A System for Error Control Coding using Expander-like codes and its Applications

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Technology

and

Master of Technology

by

Swadeshkumar A. Choudhary
(Roll No. 05D07018)

Under the guidance of
Prof. Sachin Patkar

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY–BOMBAY
June, 2010

Dissertation Approval

The dissertation entitled

A System for Error Control Coding using Expander-like codes and its Applications

by

Swadeshkumar A. Choudhary
(Roll No. 05D07018)

is approved for the degree of

Bachelor of Technology

and

Master of Technology

________________	________________
Examiner	Examiner
Guide	Chairman

Date: ____________

Place: ____________

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: ____________________

Swadeshkumar A. Choudhary
05D07018

Acknowledgments

I am grateful to Dr. Adiga (TCS Research) for proposing my thesis topic. I would like to thank Prof. S. Patkar (IIT Bombay) for his invaluable guidance and support during the entire course of this project. He has continuously motivated and inspired me to work harder to overcome various hurdles. I would also like to thank Prof. H. Narayanan and Prof. D. K. Sharma for their useful insights.

I would like to acknowledge that the work on proving the non-existence of certain embeddings in our construction has been done in collaboration with Mr. Hrishikesh Sharma. I want to thank Mr. Hrishikesh Sharma for his guidance and for always lending a ear to my far-fetched propositions. Moreover, he has made invaluable contributions towards the coherent presentation of ideas which has improved several sections of my report. The folding of Projective Geometry based graphs is related to his field of interest and he has independently developed an alternative scheme which is different from the one presented in this report.

I also want to thank Yash Deshpande and Tejas Hiremani for their help in implementing the Reed Solomon Product code in Matlab and discussing partitions of $GF(q^m)$ respectively. I also thank Abhishek Patil for his help in the initial stages of the project.

Date: ________________ Swadeshkumar A. Choudhary

Abstract

We present a novel error correcting code in which the symbols of the code correspond to the edges in a regular bipartite graph and the decoding algorithm is borrowed from Zemor's decoding algorithm for expander graphs [14]. We construct the graph using point and hyperplane incidences of $\mathbb{PG}$(5,GF(2)). Each vertex of the graph is a Reed-Solomon decoder and we call the collection of edges incident on each vertex as the sub-code for that vertex. We derive geometric bounds on the code for practical values of the minimum distance of the sub-code and show via MATLAB simulations that the average case performance of the code is 10 times better in 99% of the test cases for errors that are randomly distributed. We also prove the non-existence of certain kinds of sub-graphs in our construction and present an analysis of the eigenvalue based approach used in previous literature. For burst errors, we present 2 schemes which utilize our decoder to enhance the burst error correction rates of the codes used in CD-ROMs. We also present an application of the decoder to DVD-R to improve burst and random error correction. Two strategies of hardware implementation are presented. One of them involves efficiently folding the computations so that fewer number of processors are required. Design details of prototyping this strategy on a Xilinx Virtex 5 FPGA have been provided. Finally, we have presented a general folding scheme for point-hyperplane incidence based graphs that can be used for practical implementations of the code for much higher dimensions of projective geometry with a practical number of processors.

Contents

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

2 Previous work **3**

2.1 Linear Codes 3

2.2 Expander Codes 3

2.3 Expander Graphs 3

2.4 Construction of Expander Codes 4

2.5 Good Expander Codes 5

2.6 RS Codes as Good Component Codes 6

2.7 Good Expander Graphs 7

2.8 PG Graphs as Ramanujan Graphs 8

2.8.1 Advantages of Using PG Graphs 8

3 Background Information **9**

3.1 Zemor's Construction of Expander Codes 9

3.1.1 Decoding Algorithm 9

3.1.2 Error Correction 10

3.2 ECMA-130: The CD-ROM Standard 10

3.2.1 Main steps of Encoding 11

3.2.2 Cross Interleaved Reed-Solomon Code 12

3.2.3 Main steps of Decoding 13

3.3 Projective Space over Finite Fields 14

4 Detailed Description of Code Construction 16

4.1 Code Construction 16

4.2 Performance of Code for Random Errors 17

4.3 Performance of Code for Burst Errors 22

4.4 A side note on Encoding 23

5 Derivation of Bound for Random Errors 25

5.1 Introduction 25

5.2 Propositions 25

5.3 Vector Space Representation of Geometry 26

5.4 Cardinalities Related to $\mathbb{PG}(5,\mathbb{GF}(2))$ 26

5.5 Related Proofs 27

5.5.1 Important Lemmas 27

5.5.2 Main Proofs 31

5.6 An Eigenvalue Based approach for getting ξ 38

6 Applications 40

6.1 Application to CD-ROM 40

6.1.1 Scheme 1 41

6.1.2 Scheme 2 43

6.2 Application to DVD-R 44

6.2.1 Present DVD-R Error correction blocks 44

6.2.2 Analysis of Error Correction in DVD-R 45

6.2.3 A New ECC Scheme for DVD-R 47

6.3 Possibility of Building Other Expander-like Codes 49

6.3.1 Choice of Projective Space 49

6.3.2 Choice of RS Code 50

7 Detailed Design Description of Hardware Prototyping 51

7.1 Construction of the Graph 51

7.2 Recapitulation of Decoding Algorithm 53

7.3 Decoder Implementation Strategy 1 54

7.4 Decoder Implementation Strategy 2 . . . 55
7.5 Interconnect of Decoder . . . 55
7.5.1 Folding the Interconnect . . . 55
7.6 Memory Block Design . . . 61
7.7 RS Decoder Design . . . 63
7.8 Control Path Design . . . 65
7.8.1 1st State Machine . . . 65
7.8.2 2nd State Machine . . . 66
7.8.3 3rd State Machine . . . 67
7.9 Multiplexers used in Control Path . . . 67
7.10 Schedule for Decoder Iteration . . . 68
7.11 Detailed Decoding Process . . . 69
7.12 A modification to include Erasures . . . 71
7.13 Performance Modeling and Analysis . . . 73
7.13.1 Results of Implementation on the board . . . 75

8 Folding of PG based point-hyperplane graphs **76**
8.1 Introduction . . . 76
8.2 Description of the Computations . . . 77
8.3 Folding computations for $\mathbb{PG}(5, GF(2))$ - Propositions . . . 78
8.3.1 Proofs . . . 78
8.4 Generalization to $\mathbb{PG}(m, GF(q))$ . . . 90

9 Conclusion **96**
9.1 Future Work . . . 97
A Appendix . . . 99
A.1 The Graph used for the decoder . . . 99
A.2 Distribution of Data . . . 104
A.3 Generating the points of $\mathbb{PG}(5, GF(2))$ . . . 105

Bibliography **107**

List of Tables

4.1 Random errors ($\epsilon = 5$) 19

4.2 Random errors ($\epsilon = 7$) 19

4.3 Change in parameters of $\mathbb{C}$ with variation in minimum distance of subcode 20

4.4 Burst errors ($\epsilon = 5$) 23

6.1 Response to Burst errors for CIRC+RSPC 43

6.2 Response to Burst errors in Scheme 2 43

7.1 First 3 Disjoint Planes 56

7.2 Points of 9 Disjoint Planes used in Folding 57

7.3 Hyperplanes of 9 Disjoint Planes used in Folding 57

7.4 Re-ordered Graph Vertices 59

7.5 Erasure correction results 72

7.6 Resource Utilization 75

1 Point-Hyperplane Adjacency List 99

2 memory block-Edge Storage Correspondence 104

3 Points of $\mathbb{PG}(5, \mathbb{GF}(2))$ 106

List of Figures

2.1 Variables and Constraints in Expander Codes 5

3.1 CIRC Encoder 12

6.1 ECC Block of DVD-R 45
6.2 Partitioning ECC Block into Recording Frames 46

7.1 Strategy 1 54
7.2 Example to illustrate Re-ordering 58
7.3 (A) Distribution of computation to processing units (B) High-level System architecture 60
7.4 The interface of the Xilinx RS decoder IP 63
7.5 Data Path outline for each Processing Unit 65
7.6 Input Handling State Machine 66
7.7 Output Handling State Machine 67
7.8 Writeback Handling State Machine 68

Chapter 1

Introduction

Graph codes have been studied and analyzed in order to try and find codes that give good error correction capability at high code rates. At the same time, from a practical implementation point of view, such codes must have decoding and encoding algorithms that are efficient in terms of speed and hardware complexity.

Expander Codes, first suggested by Sipser and Spielman [12], proved to be theoretically capable of providing asymptotically good codes that were decodable in logarithmic time and could be implemented with a circuit whose size grew linearly with code size. [12] analyzes the code properties of expander codes and finds lower bounds on the number of errors that will always be corrected by the decoding algorithm. Zemor, in his analysis in [14], suggests using Ramanujan graphs for constructing expander codes, and provides a variation of the decoding algorithm which improves the bound by a factor of 12. In [6], Hoholdt and Justesen build on the work of Tanner on graph codes by suggesting the use of Reed Solomon codes as sub-codes for graphs that were derived from point-line incidence relations of projective planes.

The present work deals with the design, analysis and implementation of a decoding system which is based on a special bipartite graph. The bipartite graph is derived from point-hyperplane incidence relations of projective spaces of higher dimensions than those suggested by [6]. We look at various properties of the codes and in the process stumble upon several interesting properties of projective geometry. Also, we want the codes to be practically useful and hence we present a hardware design scheme for the decoder and prototype it on a FPGA.

The codes have excellent robustness to *random* as well as *burst* errors and hence we

envisage their application in data storage systems. *Disc storage* is a general category of secondary storage mechanisms, in which data are digitally recorded by various electronic, magnetic, optical, or mechanical methods on a surface layer deposited of one or more planar, round and rotating platters.The encoding pattern for most magnetic or optical recordings follows a continuous, spiral path covering the entire disc surface and extending from the innermost track to the outermost track.

Although discs are more durable than earlier storage mechanisms such as tapes, they are susceptible to environmental and daily-use damage. Unlike the now-obsolete 3.5-inch floppy disk, most removable media such as optical discs do not have an integrated protective casing and are therefore susceptible to data transfer problems due to scratches, fingerprints, and other environmental problems such as dust speckles. These data transfer problems, while the data is being read, manifests itself in form of bit errors in the digital data stream. Even mechanical issues such as vibration due to *occasional* high rotational speeds of disc motors also produce undesirable noise, and hence bit errors in fixed as well as removable media. A long sequence of bit read errors while a track is being read (e.g. a scratch on a track) can be characterized as *burst error*, while bit read error arising out of tiny dust speckle masking limited number of pits and lands on a track leads to *random error*. The occurrence of such events obviously not being rare, *recovery of data to maximum extent* in presence of such errors is an *essential* subsystem within most computing systems, such as CPU and disc players.

The next chapter gives relevant, exhaustive background information for the various concepts required in this area. It also gives the basic properties of cardinalities of projective geometry that we will be using. The subsequent chapters describe our code construction in detail, give proofs of propositions relating to bounds on error correction capacity and explain the application to two types of storage media(namely CD-ROMs and DVD-R). We then proceed to give details of the hardware design of the prototype of the decoder that has been evolved for the FPGA. Finally, we present a general hardware design approach to fold computations related to graphs based on projective geometry. It utilizes lattice embedding properties to fold the graph so that fewer processors may be used for the implementation of the computations.

Chapter 2

Previous work

2.1 Linear Codes

A q-ary linear error-correcting code of length n and rank k is a linear subspace $\mathbb{S}$ of F^n of dimension k, where $F = \mathrm{GF}(q)$. Each codeword belonging to this code will have k message symbols and the other $n - k$ symbols are determined by the message. The total number of codewords in $\mathbb{S}$ is q^k. Another important characteristic of the code is the minimum hamming distance d between any two codewords, which implies that the code can correct upto $\lfloor \frac{d-1}{2} \rfloor$ errors. The rate of the code is given by $r = \frac{k}{n}$. A linear code is generally described by the triplet (n, k, d).

2.2 Expander Codes

Expander codes are a family of asymptotically good, linear error-correcting codes. They can be decoded in sub-linear time(proportional to $\mathbf{log(n)}$, where $\mathbf{n}$ is length of codeword) using parallel decoding algorithms. Further, this can be achieved using identical component processors, whose count is linearly proportional to $\mathbf{n}$. These codes are based on a class of graphs known as *expander graphs*.

2.3 Expander Graphs

An expander graph is a graph in which every set of vertices has an unusually large number of neighbours. More formally,

Let $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ be a graph with **n** vertices. Then the graph $\mathbb{G}$ is a δ-expander, if every set of *at most* **m** vertices expands by a factor of δ. That is,

$$\forall S \subset V : |S| \leq m \Rightarrow |\{y : \exists x \in S \text{ such that } (x, y) \in E\}| > \delta \cdot |S|$$

Expander codes being a subclass of LDPC codes, for whose iterative decoding using variables and constraints a bipartite graph is required, we are interested mainly in bipartite expander graph. More specifically, a general (unbalanced) bipartite expander graph is a (c, d, ϵ, δ)-expander if it is a (c,d)-regular bipartite graph in which every subset of at most an ϵ fraction of the c-regular vertices expand by a factor of at least δ.

The degree of "goodness" of expansion, especially for regular graphs, can also be measured using its eigenvalues. The largest eigenvalue of a **k**-regular graph is **k**. If the second largest eigenvalue is much smaller from the first(**k**), then the graph is known to be a good expander.

2.4 Construction of Expander Codes

It is well known that a randomly chosen (c,d)-regular graph will be a good expander with high probability. A deterministic construction of good expander graph, that further leads to construction of good expander codes is by considering the edge-vertex incidence graph $\mathbb{B}$ of a **d**-regular graph $\mathbb{G}$. The edge-vertex incidence graph of $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, a (2,d)-regular bipartite graph, has vertex set $E \cup V$ and edge set

$$\{(e, v) \in E \times V : \text{ v is an endpoint of e}\}$$

Vertices of $\mathbb{B}$ corresponding to edges E of $\mathbb{G}$ are then associated to *variables*, while vertices of $\mathbb{B}$ corresponding to vertices of $\mathbb{G}$ are associated to constraints on these variables. Each constraint corresponds to a set of *linear* restrictions on the **d** variables that are its neighbors(see figure 2.1). In particular, a constraint will require that the variables it restricts form a codeword in some linear code of length **d**. Further, all the constraints are required to impose isomorphic codes(on different variables, of course). The default construction of expander codes requires **d** to remain constant as the order of $\mathbb{G}$ increases.

Formally, Let $\mathbb{B}$ be a (2,d)-regular graph between set of **n** nodes called *variables*, and $\frac{2}{d}$**n** nodes called *constraints*. Let *b(i,j)* be a function such that for each constraint C_i, the

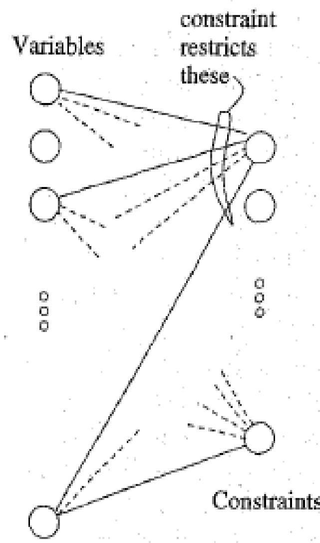

Figure 2.1: Variables and Constraints in Expander Codes

variables neighboring C_i are $v_{b(i,1)}, \cdots, v_{b(i,d)}$. Let $\mathbb{S}$ be an error-correcting code of block length **d**. The expander code $\mathbb{C}(\mathbb{B}, \mathbb{S})$ is the code of block length **n** whose codewords are the words $(x_1, \cdots, x_n)$ such that, for $1 \leq i \leq \frac{2}{d}\mathbf{n}$, $x_{b(i,1)}, \cdots, x_{b(i,d)}$ is a codeword of $\mathbb{S}$.

2.5 Good Expander Codes

As pointed out earlier, the decoding algorithm for such codes is iterative. Hence good expander codes imply *at least* the following properties:

- Better minimum distance(hence larger error-correction capability) than other codes of same length,
- Fast convergence, and
- Better code rate than other codes in the same class

We now discuss the way good codes having the above properties be derived, with help of three theorems. All the three theorems have been proved in [12], hence we just provide the statement of theorem here. For the theorems, we assume that an expander code $\mathbb{C}(\mathbb{B}, \mathbb{S})$ has been constructed having $\mathbb{S}$ as a linear code of rate **r**, block length **d**, and minimum relative distance ϵ, while $\mathbb{B}$ as the edge-vertex incidence graph of a **d**-regular graph $\mathbb{G}$ with second-largest eigenvalue λ.

Theorem 1. *The code* $\mathbb{C}(\mathbb{B}, \mathbb{S})$ *constructed as above has rate at least 2***r** *- 1, and minimum relative distance at least* $\left(\frac{\epsilon - \frac{\lambda}{d}}{1 - \frac{\lambda}{d}}\right)^2$.

Theorem 2. *If a parallel decoding round for* $\mathbb{C}(\mathbb{B}, \mathbb{S})$*, as given in [12], is given as input a word of relative distance* α *from a codeword, then it will output a word of relative distance at most* $\alpha\left(\frac{2}{3} + \frac{16\alpha}{\epsilon^2} + \frac{4\lambda}{\epsilon d}\right)$ *from that codeword.*

Theorem 3. *For all* ϵ *such that 1- 2H(*ϵ*)* > 0*, where H(*$\cdot$*) is the binary entropy function, there exists a polynomial-time constructable family of expander codes of rate 1 - 2H(*ϵ*) and minimum relative distance arbitrarily close to* ϵ^2 *in which any* $\alpha < \epsilon^2/48$ *fraction of error can be corrected by a circuit of size O(n log n) and depth O(log n).*

From theorem 1, we observe that to have high minimum relative distance, we should have ϵ as high, and $\frac{\lambda}{d}$ as low. Since $\mathbb{B}$ has been constructed out of **d**-regular graph $\mathbb{G}$, low $\frac{\lambda}{d}$ signifies high distance between first and second eigenvalues, i.e. the graph $\mathbb{G}$ has to be a "good" expander graph. Further, to have high rate, $\mathbb{S}$ has to have a high rate **r** as well, other than having high minimum relative distance ϵ.

From theorem 2, we observe that to shrink the distance of input word after one iteration maximally, we need to again have ϵ as high as possible, and $\frac{\lambda}{d}$ as low as possible. Such maximal shrinking of distance, per iteration, leads to the fastest convergence possible, as is also brought out in the proof of theorem 3.

From theorem 3, we observe that to be able to correct as high fraction of errors as possible, we need to have ϵ as high as possible, again.

2.6 RS Codes as Good Component Codes

By choosing a "good expander" graph, and fixing a code with high minimum relative distance ϵ, one can design code having the first two properties described earlier. To simultaneously have high code rate for $\mathbb{C}(\mathbb{B}, \mathbb{S})$, the component code $\mathbb{S}$ also needs to have high rate **r**. This is where bounds on rate come into picture. Given minimum relative distance ϵ, the most loose upper bound on rate is the *Singleton bound*, and Reed-Solomon codes and related ones actually achieve this bound! At the same time, we are also interested in designing *asymptotically good codes*, for which one needs to guarantee that the code rate does not fall to beyond a certain threshold at high code lengths. This

is where usage explicitly constructable class of codes, having a lower bound on code rate away from 0 as per the lone lower bound, *Gilbert-Varshamov bound*, comes to use.

If one has particular applications in mind, and is not interested in constructing an infinite family of asymptotically good codes, then another option opens up. Possibility of using codes not living by the *Gilbert-Varshamov bound* has also been mentioned in [12]. *Reed-Solomon* codes are a class of non-binary, linear codes, which for a given rate, have the best minimum relative distance(so-called *maximum distance separable* codes), **and vice-versa**. They being $(n, k, n - k + 1)$ codes, their rate need not $\rightarrow 0$, as theoretically it is possible to construct RS codes with very high distance($\approx \alpha * n$, where α is a positive fraction), as $n \rightarrow \infty$. Hence $\mathbb{C}$ are, in general, asymptotically good codes.

As a concluding remark, we had earlier observed that definition of expander code requires **d** to remain constant as **n** increases. In our construction, **d** increases as **n** increases. However, it is clear from statement of theorems 1 and 2 that higher value of **d** leads to better properties of the code $\mathbb{C}(\mathbb{B}, \mathbb{S})$. However, such codes may not be called expander codes in wake of definition of these, but just graph-based, or **expander-like** codes.

2.7 Good Expander Graphs

The construction of expander codes in [12] makes use of an **unbalanced** bipartite graph $\mathbb{B}$ made out of a d-regular graph $\mathbb{G}$. He says that a randomly chosen d-regular graph will be a good expander with high probability. Zemor pointed out that if $\mathbb{G}$ is a regular bipartite graph, then the % of errors that can be corrected using a parallel iterative decoding algorithm can be increased twelve-fold. He also pointed out that the upper bound on minimum relative distance, as pointed out in theorem 1, can also be achieved faster, if one considers *Ramanujan graphs* (Since $\frac{\lambda}{d}$ value is low for these graphs. Earlier, the particular construction of *Cayley graphs* using quadratic residues, suggested in [12] as $\mathbb{G}$, also had the Ramanujan property: λ(second-largest eigenvalue) $\leq 2\sqrt{d-1}$. However, it is noted [14] that (a) Half the constructions of the bigger class of Ramanujan graphs lead to bipartite regular graphs, and (b) Using bipartite regular graphs as $\mathbb{G}$ leads to twelve-fold improvement in error correction capability. Hence it is imperative that one focuses on using Ramanujan graphs for construction of good expander codes instead.

2.8 PG Graphs as Ramanujan Graphs

Balanced regular bipartite graphs $G_{d,d}$, which are *symmetric balanced incomplete block designs(BIBDs)* are known to be Ramanujan graphs [5]. Incidence relations of projective geometry structures give BIBDs and hence can be used to generate Ramanujan graphs. Usage of projective plane as $G_{d,d}$ along with RS codes as component codes to construct good expander-like graph codes was first reported in [6]. For our work, we do not limit to projective planes: to have better performance, we have made use of point-hyperplane incidence graphs from higher dimensional projective spaces, which also satisfy the eigenvalue properties that make a Ramanujan graph.

An example of an expander-like 1953-length code based on projective spaces is provided in section 4.1.

2.8.1 Advantages of Using PG Graphs

If one looks only at the decoding algorithm and construction of the graph, it would seem that any simple, regular bipartite graph would be a good candidate for having good expander codes and corresponding decoders. Some of the reasons for using projective geometry are as follows.

- As discussed in chapter 4, the mapping of vertices to points and hyperplanes enables us to use several projective geometry properties for disproving the existence of certain bipartite subgraphs of a fixed minimum degree. This strategy leads us to finding the minimum number of vertices required to form a complete bipartite subgraph of a given minimum degree. This number of vertices is required to calculate the bounds for error correction capability of the overall code. Thus, we don't need to use complicated eigenvalue arguments used by [12] and [14]. Also, the bounds obtained in this manner are better than our predecessors. Furthermore, Zemor had restricted the subcodes to be constrained by $d \geq 3\lambda$, λ being the second largest eigenvalue of the graph. We have *no such restriction*.

- The use of projective geometry also helps in developing a perfect *folded architecture* of the decoder for hardware implementation. Folded architectures enable efficient utilization of processors and memories, without any significant increase in scheduling complexity.

Chapter 3

Background Information

Our construction of codes is based on Zemor's ideas of Expander graphs. We briefly summarize the **basic** construction suggested by Zemor[14]. We also introduce certain concepts from projective geometry, since we use a PG-based bipartite graph construct the expander-like codes. Finally, for their applications in CDROM decoding, we overview the current existing standard for CDROM encoding and decoding, ECMA-130.

3.1 Zemor's Construction of Expander Codes

Zemor's construction is based on a **d**-regular balanced bipartite graph, $\mathbb{G} = (V, E)$. The set V is divided into two sets A and B, with $|A| = |B| = n$ such that every edge has one endpoint in A and another in B. For any vertex t, the set of edges incident on t is denoted by E_t. As the graph is bipartite, the sets E_t $\forall t \in A$ induce a partition on E. A similar partition can be created using the edge sets of the vertices belonging to B. The expander code, $\mathbb{C}(\mathbb{G}, \mathbb{S})$ is constructed by treating the edges of $\mathbb{G}$ as variables and the vertices as constraints for a binary component code $\mathbb{S}$. The block length of code $\mathbb{C}$ is $N = n \cdot d$. As before, the second largest eigenvalue of $\mathbb{G}$ is denoted by λ.

3.1.1 Decoding Algorithm

The steps in one decoding round of the algorithm suggested by Zemor are as follows:

- Each constraint t in set A completely decodes the sub-vector associated with the set of **d** variables, E_t, and replaces it with the closest codeword in $\mathbb{S}$. This step can

be carried out in parallel by all constraints in A as no symbol is shared between two constraints.

- The constraints in set B replace the sub-vector associated with its edge sets, $E_t,\ t \in B$, with the closest codeword in $\mathbb{S}$. This again can be carried out in parallel by all constraints.

3.1.2 Error Correction

The following result expresses the reduction in unsatisfied constraints in each step of this algorithm.

Suppose $\mathbb{S}$ is a linear code $(d, r \cdot d, \epsilon)$ and $\epsilon \geq 3\lambda$. Let P be a subset of A such that

$$|P| \leq \beta n(\frac{\epsilon - 2\lambda}{2d}) \tag{3.1}$$

where $\beta < 1$. Let Q be a subset of B and suppose that there exists a set $Y \subset E$ such that

1. Every edge of Y has one endpoint in P.
2. Every vertex of Q has at least $\frac{\epsilon}{2}$ edges of Y incident on it.

Then

$$|Q| \leq \frac{1}{2 - \beta}|P| \tag{3.2}$$

Using this result, Zemor proved that if $\epsilon \geq 3\lambda$, the total fraction of errors that can be corrected using the above algorithm is $\alpha \leq \beta\frac{\epsilon}{2d}(\frac{\epsilon - 2\lambda}{2d})$ for $\beta < 1$. This fraction can be seen proportional to $\frac{\epsilon^2}{4}$, an twelve-fold improvement over [12].

3.2 ECMA-130: The CD-ROM Standard

As specified by the ECMA-130 standard [7], CD-ROM decoder consists of several layers of decoding to make the codes robust towards **burst and random errors**.

The digital data to be recorded in an *Information Track* is represented by 8-bit bytes(symbols) and grouped into *Sectors*. A Sector is the smallest addressable part of the information area that can be accessed independently. It has 2352 symbols which are organized into the following categories.

Sync field The first 12 bytes are the synchronization bytes for the frame. They are used to indicate the start of a new frame.

Header field This consist of 4 bytes, a 3 byte sector address and 1 byte to indicate the mode which indicates how the data is arranged in the next fields.

User data 2048 bytes of user data are placed from byte number 16 to 2063.

Error Detection Code(EDC) field The EDC field consists of 4 bytes recorded in positions 2064 to 2067. The error detection code is a 32-bit Cyclic Redundancy Check(CRC) applied on bytes 0 to 2063.

Intermediate field It consists of 8 (00)-bytes recorded in positions 2068 to 2075. These are redundant bytes added to help improve the error correction per sector.

P & Q parity bytes These are the parity bytes added by the Reed Solomon Product code which is the first layer of encoding. The P-parity field consists of 172 bytes in positions 2076 to 2247 computed on bytes 12 to 2075. The Q-parity field consists of 104 bytes in positions 2248 to 2351 computed on bytes 12 to 2247.

3.2.1 Main steps of Encoding

Encoding is done over 2048 bytes of user data. Sync and header fields are generated, and attached ahead of the payload of user data. EDC and intermediate fields are then calculated and attached ahead of user data field. The P and Q parity bytes are calculated from the Reed Solomon Product Code(RSPC), as explained in [7]. This makes the entire block of 2352 bytes. Bytes 12 to 2351 are then passed through a scrambler, and finally the 2352 bytes are mapped onto a series of consecutive frames. Each frame consists of 24 8-bit symbols. There are 98 frames for each sector on the CD-ROM disc, and are called the **F1-Frames**.

The *F1-Frames* are fed into a Cross Interleaved Reed-Solomon Code(CIRC) encoder. This stage is responsible for the massive interleaving which corrects the majority of the burst errors. The functioning of the CIRC is explained in the next section. The result is that the *F1-Frames* of 24 bytes each are transformed into **F2-Frames** of 32 bytes each. These additional 8 bytes carry the parity information in fact. The bit pattern of each of the 24 8-bit bytes of an *F1-Frame* remains unchanged, but the bytes themselves are

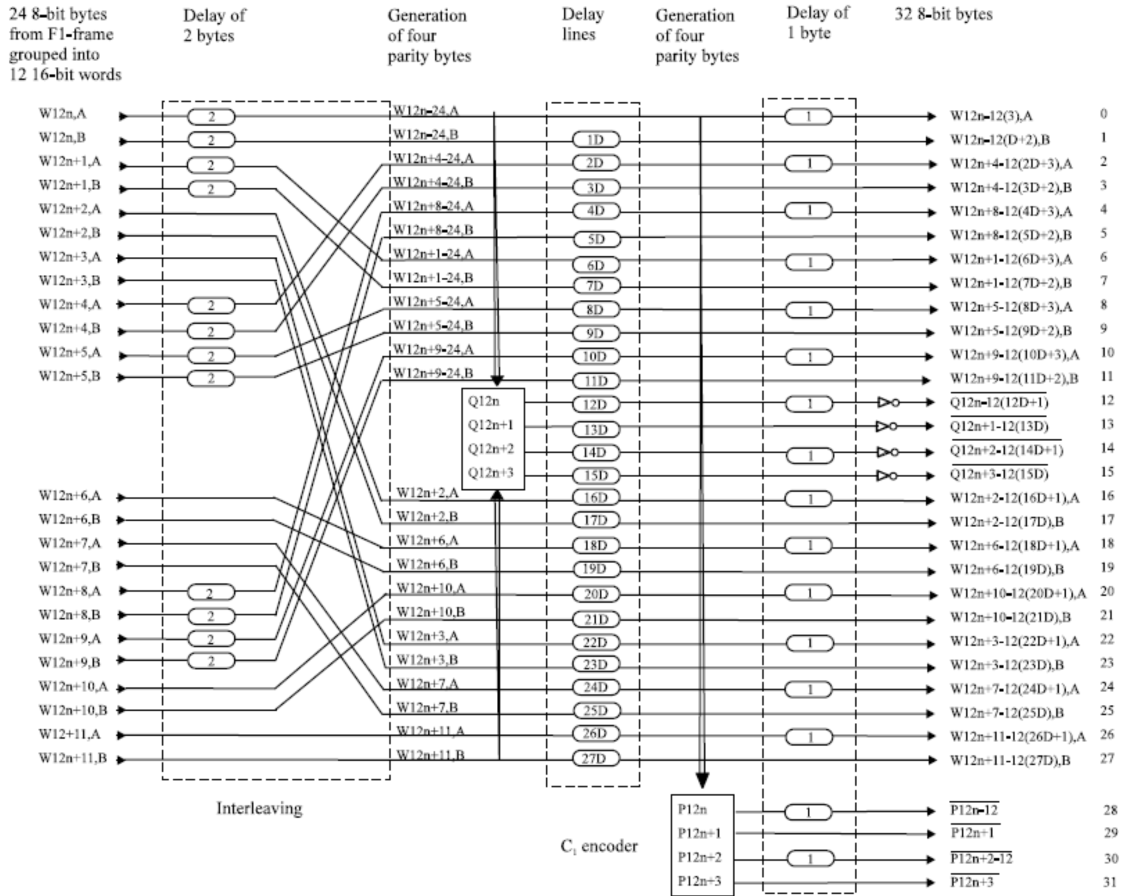

Figure 3.1: CIRC Encoder

displaced and re-distributed over 109 *F2-Frames*. A single byte called Control byte is added as first byte to each *F2-Frame*. This yields a new **F3-Frame** of 33 bytes. The information in the control byte is mainly used for addressing purposes.

The *F3-Frames* are then further passed through a few more stages before they are finally ready to be recorded on the disk. One of these stages consists of an 8-to-14 encoding. In this stage, the 8-bit symbols are mapped to 14-bit symbols. The main purpose of this stage is to even out the power spectrum density. Also, one of the stages is another simpler form of interleaving which essentially doubles the error correcting capacity obtained till the CIRC stage.

3.2.2 Cross Interleaved Reed-Solomon Code

The error correction encoding of the *F1-Frames* is carried out by a **Cross Interleaved Reed-Solomon Code(CIRC)** encoder consisting of three delay sections and two en-

coders C1 and C2. It is shown in Figure 3.1. The encoder works as follows.

- The input of the encoder consists of the 24 bytes of each F1-Frame. These bytes are ordered into 12 words of two 8-bit bytes each, denoted A and B. Byte 0 of F1-Frame number **n** is indicated as W12n,A, and byte 23 as W12n+11,B.

- The interleaving scheme of the first delay section(see figure 3.1) divides the words into two groups one of which is delayed by two F1-frame times.

- The error correction encoder C2 generates a (28,24) Reed-Solomon code.

- The second delay section consists of a series of 28 delays from 0 to 27 D F1-frame times, where D equals 4. This is the main interleaving where the symbols of a input F1-Frame are spread over 109 frames. The result is that if one symbol of the input appears in the 0th frame of the output of this stage, the next symbol will appear in the 4th output frame and so on. From the decoding point of view, we can roughly estimate the burst error required to corrupt an F1-Frame. Since C2 is two error correcting, 3 symbols of a F1-Frame need to be corrupted for decoding to fail, which corresponds to a burst of $8 * 28 + 4 = 228$ symbols.

- The error correction encoder C1 generates a (32,28) Reed-Solomon code.

- The third delay section yields a delay of one F1-Frame time to every alternate byte out of the C1 encoder.

- The output of the CIRC encoder is grouped into F2-Frames as indicated in figure 3.1. All parity bits in the P and Q bytes are inverted before they leave the encoder. The longest delay for a byte between input into, and output from, the encoder is 108 F1-Frame times.

3.2.3 Main steps of Decoding

The decoding is exactly the reverse of the encoding.

3.3 Projective Space over Finite Fields

In this section, we look at how the projective spaces are generated from finite fields. Consider a finite field $\mathbb{F} = \mathbb{GF}(s)$ with s elements, where s is a power of a prime number p i.e. $s = p^k$, k being a positive integer. A projective space of dimension d is denoted by $\mathbb{P}(d, \mathbb{F})$ and consists of one-dimensional subspaces of the $(d+1)$-dimensional vector space over $\mathbb{F}$ (an extension field over $\mathbb{F}$), denoted by $\mathbb{F}^{d+1}$. Elements of this vector space are of the form $(x_1, \ldots, x_{d+1})$, where each $x_i \in \mathbb{F}$. The total number of such elements are $s^{(d+1)}$ $= p^{k(d+1)}$. An equivalence relation between these elements is defined as follows. Two non-zero elements $\mathbf{x}$, $\mathbf{y}$ are *equivalent* if there exists an element $\lambda \in \mathrm{GF}(s)$ such that $\mathbf{x} = \lambda\mathbf{y}$. Clearly, each equivalence class consists of s elements of the field ($s-1$ non-zero elements and $\mathbf{0}$), and forms a one-dimensional subspace. Such 1-d vector subspace corresponds to a **point** in the projective space. Points are the zero-dimensional subspaces of the projective space. Therefore, the total number of points in $\mathbb{P}(d, \mathbb{F})$ are

$$P(d) = \frac{\text{number of non-zero elements in the field}}{\text{number of non-zero elements in one equivalence class}} \tag{3.3}$$

$$= \frac{s^{d+1} - 1}{s - 1} \tag{3.4}$$

An m-dimensional subspace of $\mathbb{P}(d, \mathbb{F})$ consists of all the one-dimensional subspaces of an $(m+1)$-dimensional subspace of the vector space. The basis of this vector subspace will have $(m+1)$ linearly independent elements, say $b_0, \ldots, b_m$. Every element of this subspace can be represented as a linear combination of these basis vectors.

$$\mathbf{x} = \sum_{i=0}^{m} \alpha_i b_i, \text{ where } \alpha_i \in \mathbb{F}(s) \tag{3.5}$$

Clearly, the number of elements in the vector subspace are $s^{(m+1)}$. The number of points in the m-dimensional projective subspace is given by $P(m)$ defined in earlier equation. This $(m+1)$-dimensional vector subspace and the corresponding projective subspace are said to have a co-dimension of $r = d - m$ (the rank of the null space of this vector subspace).

Let us denote the collection of all the l-dimensional projective subspaces by Ω_l. Now, Ω_0 represents the set of all the points of the projective space, Ω_1 is the set of all lines, Ω_2 is the set of all planes and so on. To count the number of elements in each of these sets, we define the function

$$\phi(n, l, s) = \frac{(s^{n+1} - 1)(s^n - 1) \ldots (s^{n-l+1} - 1)}{(s - 1)(s^2 - 1) \ldots (s^{l+1} - 1)} \tag{3.6}$$

Now, the number of m-dimensional subspaces of $\mathbb{P}(d, \mathbb{F})$ is $\phi(d, m, s)$. For example, the number of points in $\mathbb{P}(d, F)$ is $\phi(d, 0, s)$. Also, the number of l-dimensional subspaces contained in an m-dimensional subspace (where $0 \leq l < m \leq d$) is $\phi(m, l, s)$, while the number of m-dimensional subspaces containing a particular l-dimensional subspace is $\phi(d - l - 1, m - l - 1, s)$.

Chapter 4

Detailed Description of Code Construction

In this chapter, we present a particular expander-like graph code. In a subsequent chapter, this code has been designed for usage in a novel high-performance CD-ROM decoder. For sake of extensibility, in chapter 6, we discuss how a family of such codes can be constructed.

4.1 Code Construction

To construct an expander-like code, we follow [14]. We generate a balanced regular bipartite graph $\mathbb{G}$ from a projective space. A projective space of dimension $\mathbf{n}$ over $\mathbb{GF}(2)$, $\mathbb{PG}(n, \mathbb{GF}(q))$, has at least following two properties, arising out of inherent duality:

1. The number of subspaces of dimension m is equal to the number of subspaces of dimension $n - m - 1$.

2. The number of m-dimensional subspaces incident on each $n - m - 1$-dimensional subspace is equal to the number of $n - m - 1$-dimensional subspaces incident on each m-dimensional subspace.

We use these two properties of projective subspaces to create balanced regular bipartite graphs. We associate one vertex of the graph with each m-dimensional subspace and one with each $n - m - 1$-dimensional subspace. Two vertices are connected by an edge if the corresponding subspaces are incident on each other. As edges lie only between subspaces of different dimensions, the graph is bipartite with vertices associated with m-dimensional

subspaces forming one set and vertices associated with $n-m-1$-dimensional subspaces forming another. Also, the two properties, listed above, ensure that both the vertex sets have the same number of elements and that each vertex has the same degree.

We consider the graph, $\mathbb{G} = (V, E)$ obtained by taking the points and hyperplanes of $\mathbb{PG}(5, \mathbb{GF}(2))$. This is the first code we construct, and use throughout. Possibilities of constructing other useful codes of the same family is discussed in section 6.3. This projective space is generated from $\mathbb{GF}(2^6)$. In this projective space, the number of points (= number of hyperplanes) is $\phi(5, 0, 2) = 63$. Each point is incident on $\phi(4, 3, 2) = 31$ hyperplanes and each hyperplane has $\phi(4, 0, 2) = 31$ points. Therefore, we have $|V| = 126$ and $|E| = 1953$. This implies that the block length of code $\mathbb{C}$ is 1953 and the number of constraints in the code is 126. The second eigenvalue of $\mathbb{G}$, λ is 4, according to formulae by Chee and Ming[4]. Hence the ratio $\frac{\lambda}{d}$ is quite small, as required for design of "good" expander codes. The calculation of elements of $\mathbb{GF}(2^6)$ and the generation of incidence relations between points and hyperplanes in $\mathbb{PG}(5, \mathbb{GF}(2))$ has been elucidated in section 7.1 later.

As the expander graph $\mathbb{G}$ is 31-regular, the block length of the component code need also be 31 to construct an expander-like code[12]. We have chosen the 31-symbol shortened Reed Solomon code as the component code, with each symbol consisting of eight bits. Our decoding algorithm is identical to Zemor's, except that if a particular vertex detects more errors than it can correct, we skip the decoding for that vertex. This is because as a side output, it is possible to compute using Berlekamp-Massey's algorithm for RS decoding[2], whether the degree of errors in the current input block of symbols to the decoder be corrected or not. If not, then the algorithm can be made to skip decoding, thus preserving the errors in the input block. This *variation in decoding* will reduce the number of extra errors introduced by that vertex if the decoding fails. Based on this decoding algorithm, a MATLAB model of decoder was first made, to observe code's performance. It uses the built-in RS decoding and encoding functions from MATLAB.

4.2 Performance of Code for Random Errors

To benchmark the error-correction performance in wake on *random errors* occurring while reading the disc, we varied the minimum distance of the component code, and simulated

the MATLAB decoder model. Random symbol errors were introduced at random locations of the zero vector. Convergence of the decoder's output back to the zero vector was checked after simulation. As the code is linear, the performance obtained in testing for zero vector is valid for the entire code. Since the errors were introduced at random locations, simulations were run over *many different rounds* of decoding for different *pseudo-random sequences* as inputs, and *averaged*, to get reliable results. These sequences differ in random positions in which the errors are introduced. Each round of decoding for particular input further involves several iterations of execution of decoding algorithm. One iteration of decoding corresponds to both sides of the bipartite graph to finish decoding the component codes.

It is *observed experimentally* that in case of a decoding failure, beyond 4 iterations, the number of errors in the codeword stabilizes(referred to as *fixed point* in [3]). In the first few iterations, as the number of errors decrease in the overall code in a particular iteration, the number of errors *on average* to be handled by RS decoders in next iteration is lesser. Hence probabilistically, and experimentally, these component decoders converge faster, thus *increasing* the percentage of errors being corrected in its next iteration. However, after maximum 4 iterations, it was seen that there is no further reduction in errors. This phenomenon could most probably be attributed to infinite oscillations of errors in an embedded subgraph, to be described in next section.

Hence we have fixed the stopping threshold of decoder to *exactly 4 iterations* not only in simulation, but also in the practical design of a CD-ROM decoder. In simulation, if there are non-zero entries remaining after 4 iterations, then the decoding is considered to have failed. In real life, if one or more component RS decoders fail to converge at 4th iteration, then again decoding is considered to have failed. The results of our simulations are presented in Tables 4.1 and 4.2. The component codes used for these simulations have minimum distance as 5 and 7, respectively. The "failures" column represents the percentage of failed decoding attempts. The "average number of iterations" column signifies the *average* number of iterations required for successful decoding of a corrupt codeword, over various rounds.

We present some worst-case bounds on rate and error-correction capability of our codes in Table 4.3. We vary the minimum distance of subcode between 3 and 15. Beyond 15, rate of the overall code $\mathbb{C}$ becomes very less and hence impractical. We also compare

No. of errors	failures	Avg. no. of iterations
50	0	1
80	1	1.71
100	18	2.33
110	40	2.72

Table 4.1: Random errors ($\epsilon = 5$)

No. of errors	failures	Avg. no. of iterations
150	0	1.6
175	0	1.99
200	0	2.19
250	23	3.82
275	64	4.5

Table 4.2: Random errors ($\epsilon = 7$)

these bounds to the bounds derived by Zemor. For calculating the Zemor bounds and making a fair comparison, we need to remove the advantage of using Reed Solomon codes as sub-codes. Zemor had derived the bounds for general codes assuming that $\geq \frac{\epsilon}{2}$ errors could not be corrected for **any** distance(even/odd). For Reed Solomon component codes used in our construction, since we use only *odd* distances, $\frac{(\epsilon+1)}{2}$ errors can never be corrected. To account for this, we replace $\epsilon/2$ by $(\epsilon+1)/2$ in Zemor's formula to calculate the bounds.

We give a geometrical analysis of process of error correction in the overall code $\mathbb{C}$. We have also used results from this analysis to derive the bounds on error correction capability of $\mathbb{C}$. First of all, since we have $\mathbb{PG}(5, \mathbb{GF}(2))$, points form the 0-dimensional subspace and hyperplanes form the 4-dimensional subspace. Moreover, planes form 2-dimensional subspaces of the projective space, and are symmetric with respect to points and hyperplanes. Finally, 7 points are contained in a plane and a plane is contained in 7 hyperplanes in $\mathbb{PG}(5, \mathbb{GF}(2))$.

To understand the limits, given the minimum distance ϵ of the subcode, we need to find the minimum number of random errors to be introduced in $\mathbb{C}$, which will cause the decoding to fail. This will happen if the vertices corresponding to the points and hyper-

Minimum distance of subcode (ϵ)	Subcode rate	Lower bound on rate of $\mathbb{C}$	Error-correcting capability of $\mathbb{C}$	Zemor's bound for $\mathbb{C}$
3	0.94	0.87	3	–
5	0.87	0.74	8	–
7	0.81	0.61	15	–
9	0.74	0.48	24	–
11	0.68	0.35	35	–
13	0.61	0.23	48	42
15	0.55	0.1	87	65

Table 4.3: Change in parameters of $\mathbb{C}$ with variation in minimum distance of subcode

planes **get locked** in such a way that in each iteration, an **equal number** of constraints fail on each side. This is the ***minimal*** *configuration of failure*. Errors can expand over iterations (more edges represent corrupt symbols), but *eventually* the codeword will be decoded if the minimal configuration of failure is not corrupted. Similarly, if errors shrink, i.e. lesser number of vertices in bipartite graph fail in next iteration, then it leads to a case of decoding convergence, not decoding failure.

For example, if we consider $\epsilon = 5$, each vertex of the graph can correct up to 2 erroneous symbols($\lfloor \frac{\epsilon}{2} \rfloor$) in the set of symbols that it is decoding. If 3 or more erroneous symbols are given to it, then either the decoder, based on Berlekamp-Massey's algorithm, skips decoding, or it outputs another codeword that in worst case has at least ϵ different symbols now(than the transmitted codeword), and hence at least ϵ errors. In either case, if we can generate a case in which decoding of the sub-code fails at vertices corresponding to 3 points, all of which are incident on 3 hyperplanes, we have a situation in which the 3 points will transfer at least 3 errors to each of vertices corresponding to the 3 hyperplanes. These vertices may also fail, or decode a different codeword, while decoding their inputs. Again in the worst case each of these hyperplane decoders will output at least 3 erroneous symbols. These corrupt symbols, or errors, are then transferred back to the vertices representing the 3 points. Thus, the errors will keep **oscillating infinitely** from one side of the graph to the other, and the decoder will never decode the right codeword. Thus, a minimum of $3 * 3 = 9$ errors are required to cause a failure of decoding. We assume the worst case scenario here also in the sense that the *wrong* decoding by a decoder *does*

not reduce the number of errors in the minimum configuration of failure. Under this assumption, the bounds can be called **"tight"**.

For any case in which less than 9 corrupt symbols exist, by pigeonhole principle, we will have *at least one* hyperplane or point having less than 3 errors incident on it. Decoder corresponding to that vertex will correctly decode the sub-code, thus reducing the total number of errors flowing in the overall decoder system of $\mathbb{C}$. This will, in next iteration, cause some other hyperplane or point to have less than 3 errors. Thus in the subsequent iterations, all the errors will definitely be removed. Therefore, *8 errors or less will always be corrected.* As we can see from Table 4.1, the worst case scenario is very unlikely to occur and for randomly placed errors, even 80 errors are found to be corrected 99% of the time.

Now the question is, when can we find a configuration in which 3 points are all incident on 3 hyperplanes? If we choose any plane in the given geometry, we can pick up any three points of that plane and find any 3 hyperplanes corresponding to the same plane. This will ensure that all the 3 points are incident on all the 3 hyperplanes. Thus, if for some input, the decoding at these 3 points fails, in the worst case they will corrupt all the edges incident on them. This in turn would cause 3 errors each, on the chosen hyperplanes. Hence the errors would oscillate between points and hyperplanes for each successive iteration. Thus, in the worst case, there need to be 9 erroneous symbols, located such that they are incident on the 3 chosen points, to cause the decoding of $\mathbb{C}$ to fail.

In general, if we are given a minimum distance ϵ of the subcode, it is known that at each vertex, more than $(\frac{\epsilon+1}{2})$ errors will not be corrected. So, in our graph we need to find the *minimum* number of vertices ξ required to get a **embedded bipartite subgraph** such that each vertex in the subgraph has a degree of at least $(\frac{\epsilon+1}{2})$ towards vertices on other side of the subgraph. Once this number of vertices in some embedded subgraph has been found, the number of errors that can always be corrected by our decoder is given by:

$$E = \xi(\frac{\epsilon+1}{2}) - 1$$

In $\mathbb{PG}(5, \mathbb{GF}(2))$, a plane has 7 points, and is contained in 7 hyperplanes. For $3 \leq \epsilon \leq 13$, ϵ being odd, the minimum number of vertices ξ corresponds to $(\frac{\epsilon+1}{2})$, and the corresponding points and hyperplanes can be picked from any plane. For $\epsilon \geq 15$, the calculation of ξ is non-trivial, since points and hyperplanes from one plane are not

sufficient. This is because $\frac{\epsilon+1}{2} = 8$, which would require us to get a subgraph of minimum degree 8. Construction of such embedded subgraph is not possible by choosing only one plane. Thus, such a subgraph with 8 vertices on each side cannot exist. We have constructed proofs to show non-existence of a minimum degree 8 embedded bipartite subgraph having order of 9 and 10, within point-hyperplane graph of $\mathbb{PG}(5, \mathbb{GF}(2))$. At a conjecture level, we suspect that an order-11, minimum degree 8 subgraph embedding is also not possible. Without using this conjecture, still, the lower bound of 87 as number of erroneous symbols in a 1953-length block of input based on $\mathbb{C}$, as in the table 4.3, is a *loose lower bound*. Since our constructions are exact, we can use these *tight* lower bounds for all practical values of ϵ, wherever calculation of it is possible. Otherwise, another *looser*, lower bound can be found using Lemma 1 of [6], but derivation of that uses eigenvalue calculations. We have outlined the derivation in the next chapter. Without that, it is still clear from this table that the bound is **much better** than the bound obtained by Zemor[14] using eigenvalue arguments.

4.3 Performance of Code for Burst Errors

The strongest applications for this code lie in the areas of mass data storage such as discs. Here, as pointed earlier in chapter 1, burst errors are the dominant cause of data corruption. Hence we have also examined the burst error correction capabilities of our code. We benchmark this capability against that of codes designed in ECMA-130 standard for CD-ROM encoding, which is considered to be very robust to burst errors.

In bipartite graph $\mathbb{G}$ constructed from $\mathbb{PG}(5, \mathbb{GF}(2))$, we label the edges with integers, to map various symbols of the codeword. Such a labeling is not required to understand/ characterize the random error correction capability of the code. But here, we label the edges with numbers to try to maximize the burst error correction capability. This is achieved if each consecutive symbol, possibly part of a burst, is mapped to edges that are incident on distinct vertices representing different component decoders. Thus, consecutive numbered edges, representing consecutively located symbols in input symbol stream, go to different vertices hosting different RS decoders. Since there are 63 vertices on one side of the graph, this scheme of numbering implies that edges incident on vertex 1 are assigned the numbers $\{1, 64, \ldots, 1890\}$. Similarly, the edges incident on Vertex 2 are assigned {2,

65, ..., 1891}, and so on. This numbering essentially achieves the effect of *interleaving* of code symbols. If the error correcting capacity of each component RS decoder is $\mu(=\lfloor\frac{\epsilon}{2}\rfloor)$, then the minimum burst error correcting capacity of $\mathbb{C}$ will be $\mu * 63$. For example, for $\epsilon = 5$, μ is 2, and the minimum burst error correcting capacity is $2 * 63 = 126$. Table 4.4 gives MATLAB simulation results for burst error correction for $\epsilon = 5$.

No. of errors	failures	Avg. no. of iterations
126	0	1
135	26	2.43

Table 4.4: Burst errors ($\epsilon = 5$)

To demonstrate the excellent burst error correction capacity of our code, we benchmark it against the massive interleaving based codes in CD-ROMs. Traditionally in ECMA-130, the encoding utilizes heavy interleaving and dependence on erasure correction to deal with burst errors. For erasure correction, one level of decoding identifies the possible locations of the error symbols. The next level of decoding uses this information to correct them. The stage/process of interleaving used in CD-ROMs makes the encoding and decoding slower. We propose two schemes in next chapter, which offer significant improvement in burst error correction at similar data rates. Our decoder, being fully parallel in its decoding, can handle larger sets of data at a time and hence could be used to increase the throughput. In our schemes, however, we wanted to fit our decoders in place of the heavy interleaving stage of the CD-ROM decoding data path, which only processes one frame at a time. Thus, in terms of throughput we will be matching the CD-ROMs but we will surpass them in burst error correction capability.

4.4 A side note on Encoding

The overall code is linear and the decoding is hard decision based, hence test on a all zero codeword suffices to test the properties of the code. To generate non-zero codewords, the following rudimentary method can be used.

For the encoding process, we derive the parity matrix and find its orthogonal matrix to get the generator matrix. Suppose $d = 5$ which means that for each sub-code 4 edges

act as parity symbols. The parity matrix for each vertex is given by:

$$\begin{pmatrix} 1 & \alpha & \alpha^2 & \dots & \alpha^{30} \\ 1 & \alpha^2 & \dots & \dots & \alpha^{60} \\ . & \dots & \dots & \dots & . \\ 1 & \alpha^4 & \dots & \dots & \alpha^{120} \end{pmatrix}$$

where $\alpha = 2$ for us. The parity matrix for the 126 vertices is generated using the above parity matrix and inserting the appropriate entries at the corresponding edge locations. Each column of the overall parity matrix corresponds to an edge and there are 126*4 rows. We then perform row operations to get it in RRE form. The generator matrix is then easily obtained. A codeword is given by the product of the message vector with the generator matrix.

The above stated method is not efficient because it uses matrix-vector multiplication and hence is of $O(n^2)$. More efficient methods could be possible by utilising the structure of the graph. Getting an efficient encoder design is one of the possibilities of future work in this area.

Chapter 5

Derivation of Bound for Random Errors

5.1 Introduction

This chapter tries to establish presence/absence of certain subgraph embeddings in bipartite graph $\mathbb{G}$ made from $\mathbb{P}(5, \mathbb{GF}(2))$. The edges of this bipartite graph signify the subsumption, or reachability relationship in the lattice representation of the projective space. The presence/absence of these embedded subgraphs leads to derivation of lower bound of error correction capability of our expander-like code for cases whenever $\epsilon \geq 15$. As discussed earlier in section 4.2, the calculation of ξ is non-trivial, since it involves choosing multiple planes. In this chapter, we will calculate minimum value of ξ by systematically eliminating the cases for which the required subgraph does not exist.

5.2 Propositions

We first state without proof, the two propositions that are needed to establish the non-existence of certain minimal embeddings.

Proposition 1. *In the construction of bipartite graph mentioned above, there exists no embedded subgraph having size of partitions as 9, the degree of each of whose vertices has a minimum degree(δ) of 8.*

Proposition 2. *In the construction of bipartite graph mentioned above, there exists no*

embedded subgraph having size of partitions as 10, the degree of each of whose vertices has a minimum degree(δ) of 8.

In the following sections, we work out the proofs of these individual propositions.

5.3 Vector Space Representation of Geometry

To recall, the points of an n-dimensional projective space over a field $\mathbb{F}$ can be taken to be the equivalence classes of nonzero vectors in the (n + 1)-dimensional vector space over $\mathbb{F}$. Vectors in an equivalance class are all scalar multiples of one-another. These vector being one-dimensional subspaces, they also represent the rays of a vector space passing through origin. The orthogonal subspace of each such ray is the **unique** n-dimensional subspace of $\mathbb{F}^{n+1}$, known as hyperplanes. Each vector h of such orthogonal subspace is linked to the ray, p, by a dot product as follows.

$$p_0 h_0 + p_1 h_1 + \cdots + p_n h_n = 0$$

where p_i is the ith coordinate of p. This uniqueness implies bijection, and hence a vector p can be used to represent a hyperplane subspace, which is exclusive of this vector as a point. Due to duality, similar thing can be said about a hyperplane subspace.

Hereafter, whenever we say that two projective subspaces of same dimension are independent, we mean the linear independence of the corresponding vector subspaces in the overall vector space.

5.4 Cardinalities Related to $\mathbb{PG}$(5,$\mathbb{GF}$(2))

1. Any line in this space is defined by any 2 points. The unique third point lying on the lines is determined by linear combination of corresponding one-dimensional subspaces. Hereafter, a line will be represented as a tuple <a, b, a+b>.

2. Similarly, or dually, 3 hyperplanes intersect in a particular 4-d projective subspace, or flat.

3. Any plane in this space is defined by 3 non-collinear points, and their 4 unique linear dependencies in the corresponding vector space. Hereafter, a plane will be

represented as <a, b, c, a+b, b+c, a+c, a+b+c>, with the non-canonical choice of non-collinear points within this plane being implicit as <a, b, c>.

4. A plane contains 7 lines, thus being a Fano plane by itself.
5. A plane reaches out to 7 hyperplanes and 7 points in the corresponding lattice structure through transitive join and meet operations. Such an hourglass structure will be critical in our proofs.
6. Similarly, a hyperplane reaches out to 31 points: 5 of them being independent, and others representing the linearly dependent vectors on these.

5.5 Related Proofs

5.5.1 Important Lemmas

Lemmas Related to Projective Space

Lemma 4. *In projective spaces over $\mathbb{GF}(2)$, any subset of points(hyperplanes) having cardinality of 4 or more has 3 non-collinear(independent) points(hyperplanes).*

Proof. The underlying vector space is constructed over GF(2). Hence, any 2-dimensional subspace of contains the zero vector, and non-zero vectors of the form $\alpha a + \beta b$. Here, a and b are linearly independent one-dimensional non-zero vectors, and α and β can be either 0 or 1:

$\alpha, \beta \in \mathbb{GF}(2) : (\alpha = \beta) \neq 0.$

Thus, any such 2-d subspace contains exactly 3 non-zero vectors. Therefore, in any subset of 4 or more points of a projective space over $\mathbb{GF}(2)$ (which represent one-dimensional non-zero vectors in the corresponding vector space), at least one point is not contained in the 2-d subspace formed by 2 randomlypicked points from the subset. Thus in such subset, a further subset of 3 independent points(hyperplanes) i.e. 3 non-collinear vectors can always be found. □

Lemma 5. *Let there be 7 hyperplanes $H1, \cdots, H7$ reachable from a plane $P1$ in $\mathbb{PG}(5, \mathbb{GF}(2))$. Let there be any other plane $P2$, which may or may not intersect with the point set of plane $P1$. Then, any point on $P2$ which is not reachable from plane $P1$, can* <u>*maximally*</u>

reach out to 3 of these 7 hyperplanes, and vice-versa. Further, these 3 hyperplanes cannot be independent. Dually, any hyperplane containing $P2$, and not containing $P1$, can maximally reach out to 3 of the 7 points contained in $P1$ and which are not independent, and vice-versa.

Proof. If a point on plane $P2$ which is not reachable from plane $P1$ lies on 4 or more hyperplanes(out of 7) reachable from plane $P1$, then by lemma 4, we can always find a subset of 3 independent hyperplanes in this set of 4. In which case, the point will also be reachable to linear combination of these 3 independent hyperplanes, and hence to all the 7 hyperplanes which lie on plane 1. This contradicts the assumption that the point under consideration does not lie on plane $P1$. The role of planes $P1$ and $P2$ can be interchanged, as well as roles of points and hyperplanes, to prove the remaining alternate propositions.

Hence if the point considered above lies on 3 hyperplanes reachable from $P1$, then these 3 hyperplanes cannot be independent, following the same argument as above. Otherwise it is indeed possible for such a point to lie on 3 hyperplanes, e.g. in the case of the planes $P1$ and $P2$ being disjoint. □

Lemma 6. *Let there be 7 hyperplanes $H1, \cdots, H7$ reachable from a plane $P1$ in $\mathbb{PG}(5, \mathbb{GF}(2))$. Further, let there be any other plane $P2$, which intersects $P1$ in a line. Then, there exist 4 hyperplanes reachable from plane $P1$ which do not contain any of the 4 points that are in plane $P2$, but not in plane $P1$.*

Proof. A line contains 3 points in $\mathbb{PG}(5, \mathbb{GF}(2))$. Hence, $P1$ and $P2$ intersect in 3 points. By duality arguments, they intersect in 3 hyperplanes as well. Hence, $P2$ contains 7-3 = 4 points that are not common to point set of $P1$. By lemma 5, these 4 points can at maximum lie on 3 hyperplanes reachable from plane $P1$. Since there are 3 hyperplanes common to $P1$ and $P2$, and hence these 4 points already lie on them, they do not further lie on any more hyperplane reachable from $P1$, but not from $P2$. □

Lemma 7. *Let there be 7 hyperplanes $H1, \cdots, H7$ reachable from a plane $P1$ in $\mathbb{PG}(5, \mathbb{GF}(2))$. Further, let there be any other plane $P2$, which intersects $P1$ in a point. By lemma 5, the 6 hyperplanes not containing both $P1$ and $P2$ still intersect $P2$ maximum in a line each. Then, (a) Such lines contain the common point to $P1$ and $P2$, and hence exactly*

2 more out of remaining 6 points of $P2$ that are not common to $P1$, and (b) 3 pairs of hyperplanes out of the 6 non-common hyperplanes intersect in a (distinct) line each out of the 3 possible lines in $P2$ containing the common point.

Proof. Let A_c be the common atom(point) between planes $P1$ and $P2$. By duality, exactly one hyperplane will be common to both $P1$ and $P2$. Let some non-common hyperplane H_{nc} reachable from plane $P1$ intersect plane $P2$ in a line $L1$, that is, $H_{nc} \cap P2 = L1$. Then, $A_c \in L1$. For if it is not, then

$|L1 \cap A_c| = 4$

Also, $L1 \cup A_c \subseteq H_{nc}$

and, $L1 \cup A_c \subseteq P2$

$\Rightarrow |H_{nc} \cap P2| = 4$, a contradiction to lemma 5. Hence, the line $L1$ contains common point A_c and 2 more out of remaining 6 points of $P2$ that are not common to $P1$.

Exactly 3 hyperplanes of the nature $H1$, $H2$, $H1+H2$ intersect in a 4-dimensional projective subspace, in $\mathbb{PG}(5, \mathbb{GF}(2))$. Such a subspace can always be formed by taking union of a projective plane and a line intersecting the plane in a point, by rank arguments. Let the common hyperplane to $P1$ and $P2$ be H_c. Let other hyperplanes reachable from $P1$ be $H1, H2, H1 + H_c, H2 + H_c, H1 + H2, H1 + H2 + H_c$. Then, the pairs of hyperplanes $<H1, H1 + H_c>$, $<H2, H2 + H_c>$, and $<H1 + H2, H1 + H2 + H_c>$, along with H_c, form 3 distinct 4-d subspaces, which leads to 3 distinct lines of meet under plane $P2$, for each pair. This can also be verified from the fact that there are exactly 3 distinct lines in plane $P2$ that have a point A_c in common. These 3 lines, and their individual unions with $P1$, leads to reachability to $H1, H_c, H1+H_c$, $H2, H_c, H2+H_c$, and $H1+H2, H_c, H1+H2+H_c$, respectively. □

Lemma 8. *Let there be two hyperplanes $H1$ and $H2$ meeting in a plane $P1$. Both $H1$ and $H2$ intersect any plane $P2$ disjoint from $P1$ in exactly a line, by lemma 5. Then these intersecting lines $L1$(of $H1$ and $P2$) and $L2$(of $H2$ and $P2$) cannot be the same.*

Proof. Let the vector space of a projective geometry flat X be denoted by $V(X)$. Flats are sets of points, each of which *bijectively* corresponds to a 1-d vector in the corresponding vector space. Also, closure of a flat(in terms of containing a point) is

defined as corresponding closure of the vector subspace. Hence, family of substructures in a projective space is bijectively intertwined to the family of subspaces in the corresponding vector space(a category-theoretic? isomorphism). Then, if $L1 = L2$ were true, then

$$V(L1) = V(L2) \tag{5.1}$$

where

$$V(L1) = V(H1) \cap V(P2) \tag{5.2}$$
$$V(L2) = V(H2) \cap V(P2) \tag{5.3}$$

Also, it is given that

$$V(H1) \cap V(H2) = V(P1) \tag{5.4}$$
$$V(P1) \cap V(P2) = \phi \tag{5.5}$$

Hence if one takes closure of set of vectors contained in $V(L1) \cup V(P1)$ ($L1$ is part of $P2$ which does not intersect with $P1$), it does generate the entire vector subspace $V(H1)$. Similarly, closure of set of vectors contained in $V(L2) \cup V(P1)$ generates the entire vector subspace $V(H2)$. Then from equation 5.1, the generated subspaces $V(H1)$ and $V(H2)$ coincide, a contradiction. □

Lemmas Related to Embedded Graphs

Lemma 9. *In a bipartite graph having 9 vertices each in both partite sets, and having a minimum degree(δ) of at least 8, any collection of 3 vertices from one side is incident on at least 6 common vertices on the other side.*

Proof. Let the vertices of one side be denoted as (a1, a2, $\cdots$, a9), and that of other side by (b1, b2, $\cdots$, b9). Given $\delta = 8$, it is obvious that minimal intersection of neighborhoods of a1 and a2 happens in some(at least) 7 vertices from the other side. Then the 2 remaining vertices are $N(a1) - N(a1) \cap N(a2)$ and $N(a2) - N(a1) \cap N(a2)$, respectively. The neighborhood of vertex a3 may either contain all these 7 vertices ($N(a1) \cap N(a2)$), or the two vertices $N(a1) - N(a1) \cap N(a2)$ and $N(a2) - N(a1) \cap N(a2)$, and at least 6 vertices out of $N(a1) \cap N(a2)$. Hence the minimal intersection of neighborhoods of arbitrarily chosen vertices a1, a2 and a3 is of cardinality 6. □

Lemma 10. *In a bipartite graph having 10 vertices each in both partite sets, and having a minimum degree(δ) of at least 8, any collection of 3 vertices from one side is incident on at least 4 common vertices on the other side.*

Proof. Let the vertices of one side be denoted as (a1, a2, $\cdots$, a9), and that of other side by (b1, b2, $\cdots$, b9). Given $\delta = 8$, it is obvious that minimal intersection of neighborhoods of a1 and a2 happens in some(at least) 6 vertices from the other side. The two vertices in $N(a1) - N(a1) \cap N(a2)$ and two more in $N(a2) - N(a1) \cap N(a2)$ count the 4 remaining vertices on the other side. The neighborhood of vertex a3 may either contain all these 6 vertices ($N(a1) \cap N(a2)$), or at most all 4 vertices $N(a1) - N(a1) \cap N(a2)$ and $N(a2) - N(a1) \cap N(a2)$, and at least 4 vertices out of $N(a1) \cap N(a2)$. Hence the minimal intersection of neighborhoods of arbitrarily chosen vertices a1, a2 and a3 is of cardinality 4. □

Lemma 11. *In a bipartite graph having 11 vertices each in both partite sets, and having a minimum degree(δ) of at least 8, any collection of 3 vertices from one side is incident on at least 2 common vertices on the other side.*

Proof. Let the vertices of one side be denoted as (a1, a2, $\cdots$, a9), and that of other side by (b1, b2, $\cdots$, b9). Given $\delta = 8$, it is obvious that minimal intersection of neighborhoods of a1 and a2 happens in some(at least) 5 vertices from the other side. The three vertices in $N(a1) - N(a1) \cap N(a2)$ and three more in $N(a2) - N(a1) \cap N(a2)$ count the 6 remaining vertices on the other side. The neighborhood of vertex a3 may either contain all these 5 vertices ($N(a1) \cap N(a2)$), or at most all 6 vertices $N(a1) - N(a1) \cap N(a2)$ and $N(a2) - N(a1) \cap N(a2)$, and at least 2 vertices out of $N(a1) \cap N(a2)$. Hence the minimal intersection of neighborhoods of arbitrarily chosen vertices a1, a2 and a3 is of cardinality 2. □

5.5.2 Main Proofs

Proving the four propositions of section 5.2 is done by demonstrating how one can incrementally construct an embedded bipartite subgraph, by improving over minimum degree of a smaller subgraph. This requires looking at the planes involved in the construction of the embedding.

Theorem 12. *In the construction of bipartite graph mentioned in section 5.1, there exists no embedded subgraph having size of partitions as 9, the degree of each of whose vertices has a minimum degree(δ) of 8.*

Proof. Assume that such a subgraph exists. Then by lemma 9, any 3 points have at least 6 hyperplanes in common, and vice-versa. By lemma 4, the set of 9 points contains at least 3 non-collinear points. The 6 common hyperplanes to them in the subgraph must contain the (unique) plane defined by the points. If one of the points is contained in some different plane, then by lemma 5, this point can only reach out to maximum 3 hyperplanes belonging to the first plane, and not (all) 6 common hyperplanes, a contradiction. Again, from lemma 4, one can pick a subset of 3 independent hyperplanes out of these 6 common hyperplanes. By dual arguments, these 3 hyperplanes also must have 6 points in common, reachable from a single unique plane defined by the 3 hyperplanes. Thus, a 6-a-side bipartite subgraph with reachability defined by a single plane of the underlying projective space, is embedded in the 9-a-side bipartite subgraph we are trying to construct.

- *Case 1:* Out of the remaining 3 points(hyperplanes) in the 9-a-side subgraph, we can at most choose 1 point such that it is contained in all the 6 hyperplanes. This 1 point is the 7th point on the 7-7 hourglass passing through a single plane. It also involves the remaining 7th hyperplane being incident on all these 6+1 points. The remaining 2 points are necessarily part of at least one other plane. This configuration of 2 remaining points and 2 remaining hyperplanes may maximally form a $K_{2,2}$ complete induced subgraph by considering whichever number of intervening planes. In terms of their minimum degree, these 2 remaining points, by lemma 3overlap, can reach out maximum to 3 more hyperplanes reachable from plane $P1$. Hence the maximum degree these two points achieve is 5, in any possible construction. This contradicts the presence of assumed subgraph having δ of 8.

- *Case 2:* On similar lines, if we choose the 3 remaining points and hyperplanes apart from the 6-a-side subgraph to form a complete bipartite subgraph $K_{3,3}$ by any construction, then again by lemma 5, they can reach out maximum to 3 more hyperplanes reachable from plane $P1$. In such a case, they maximally achieve a minimum degree of 6, which is still lesser than the requirement of 8.

□

Theorem 13. *In the construction of bipartite graph mentioned in section 5.1, there exists no embedded subgraph having size of partitions as 10, the degree of each of whose vertices has a minimum degree(δ) of 8.*

Proof. Assume that such a subgraph exists. Then by lemma 10, any 3 points have at least 4 hyperplanes in common, and vice-versa. By lemma 4, the set of 10 points contains at least 3 non-collinear points. The 4 common hyperplanes to them in the subgraph must contain the (unique) plane defined by the points. If one of the points is contained in some different plane, then by lemma 5, this point can only reach out to maximum 3 hyperplanes belonging to the first plane, and not (all) 4 common hyperplanes, a contradiction. Again, from lemma 4, one can pick a subset of 3 independent hyperplanes out of these 4 common hyperplanes. By dual arguments, these 3 hyperplanes also must have 4 points in common, reachable from a single unique plane defined by the 3 hyperplanes. Thus, a 4-a-side bipartite subgraph with reachability defined by a single plane of the underlying projective space, is embedded in the 10-a-side bipartite subgraph we are trying to construct.

By considering only one intervening plane, we can maximum go upto 7-a-side subgraph only. Hence to construct 10-a-side graph, we need to consider at least one more plane in the construction. We now individually consider the cases where the maximum number of points taken from any one of the planes is n: $4 \leq n \leq 7$.

- *Case 1:* Assume that the maximally possible set of 7 points and some number of hyperplanes are taken from the plane $P1$ intervening the 4-a-side subgraph already present. The number of hyperplanes considered from $P1$ could therefore be anywhere between 4 and 7. Hence we need to consider at least one more intervening plane between the remaining hyperplanes(minimum: 3, maximum: 6) and the 3 remaining points. In the best possible construction, these remaining hyperplanes and points form a biclique. Then, any particular hyperplane from this biclique is reachable to a maximum of all 4 points of this biclique, plus at maximum 3 more points of plane $P1$; refer lemma 5. Hence the degree requirement of these hyperplanes is unsatisfiable using a 7-* partition of the 10-point set, a

contradiction.

- *Case 2:* Next, assume that 6 points and some number of hyperplanes are taken from the plane $P1$ intervening the 4-a-side subgraph already present. The number of hyperplanes considered from $P1$ could therefore be anywhere between 4 and 7. Hence again we need to consider at least one more intervening plane between the remaining hyperplanes(minimum: 3, maximum: 6) and the 4 remaining points. In another best possible construction, these remaining hyperplanes and points form a biclique. Then, any particular hyperplane from this biclique is reachable to a maximum of all 4 points of this biclique, plus at maximum 3 more points of plane $P1$; refer lemma 5. Hence the degree requirement of these hyperplanes is unsatisfiable using a 6-* partition of the 10-point set, a contradiction.

- *Case 3:* Next, assume that 5 points and some number of hyperplanes are taken from the plane $P1$ intervening the 4-a-side subgraph already present. The number of hyperplanes considered from $P1$ could therefore be anywhere between 5 and 7. Hence again we need to consider at least one more intervening plane between the remaining hyperplanes(minimum: 3, maximum: 5) and the 5 remaining points.

  First, we claim that under this case, a subgraph $K_{5,5}$ having one intervening plane always exists. To see that, let's take the lone boundary case where 5 points and 4(minimum) hyperplanes are taken from plane $P1$, and hence a $K_{5,5}$ biclique is not provided by incidences of $P1$. Then, the remaining 6 hyperplanes must belong to plane/s different from $P1$. By lemma 5, they can at maximum reach out to 3 of the 5 points reachable from $P1$. To satisfy their requirement $\delta \geq 8$, they should therefore be reachable to all 5 of the remaining points. Hence the set of 6 remaining hyperplanes and 5 remaining points form a $K_{6,5}$ biclique, and hence also $K_{5,5}$. By lemma 4 and the fact that a plane formed by 3 independent points reaches out to 7 hyperplanes, which is simultaneously minimum and maximum, this $K_{5,5}$ biclique contains exactly 1 intervening plane different from $P1$.

  By abuse of notation, let's call the plane intervening the always-present $K_{5,5}$ subgraph as $P1$. Then, at maximum 7 hyperplanes can be considered from $P1$ in the construction, and hence 3, 4 or 5 hyperplanes and remaining 5 points need to be

added to $K_{5,5}$ by considering other planes. In case when either 3 or 4 hyperplanes are considered from other planes, the set of 5 remaining points cannot have their degree requirements satisfied. For, these points can be reachable to maximum 4 hyperplanes from other planes, and maximum 3 more, considering plane $P1$(refer lemma 5). Hence we look into the case when 5 hyperplanes, and 5 points are considered by looking into other planes.

In this case, each hyperplane out of 5 remaining hyperplanes needs to be reachable to 3 different points reachable from $P1$. These 3 different points cannot be independent, for if they were, then the corresponding hyperplane would have been on $P1$ rather than any other plane. Hence each one out of 5 such collections of 3 points forms a line. Given a plane in $\mathbb{PG}(5, \mathbb{GF}(2))$ having its point set as <a,b,c,a+b,b+c,a+c,a+b+c>, it is immediately obvious that no subset of 5 points contains 5 different lines. In fact, to have 5 different lines contained in some point subset, the minimum size of the subset required is 7. Hence all possible constructions in this case leaves at least one hyperplane not having its degree requirement satisfied, a contradiction.

- *Case 4:* Finally, assume that 4 points and some number of hyperplanes are taken from the plane $P1$ intervening the 4-a-side subgraph already present. The number of hyperplanes considered from $P1$ could therefore be anywhere between 4 and 7. Hence again we need to consider at least one more intervening plane between the remaining hyperplanes(minimum: 3, maximum: 6) and the 6 remaining points. We consider the following two cases.

  - ◇ In this case, we assume that the remaining 6 points contain at least one subset of size 3 forming a line. At least one point out of 3 remaining points of this 6-set will be not be part of this line(a line has maximum 3 points in $\mathbb{PG}(5, \mathbb{GF}(2))$). This point and the line therefore form a plane $P2$, which is maximal, by our assumption in Case 4. Since in this case, *any 3* points will have at least 4 common hyperplanes to satisfy their degree requirements, the 3 independent points of plane $P2$ will have 4 hyperplanes in common, and vice-versa. The remaining 2 hyperplanes, hereafter referred as $H1$ and $H2$, do not contain both $P1$ and $P2$. More formally, by lemmas 7 and 8, the best

case occurs when

$H\,i \cap P\,i \;=\;$ a line, for i, j=1 and 2

Hence these hyperplanes can reach out to a maximum of 6 points reachable from $P1$ and $P2$. In fact, they need to reach out to exactly 6 to satisfy their degree requirements. This reachability to 6 points by each of the 2 hyperplanes must consist of reachability to one line each from planes $P1$ and $P2$.

By lemmas 7 and 8, the intersecting lines of $H1$ and $H2$ to say, $P1$, cannot be the same. In a plane of $\mathbb{PG}(5, \mathbb{GF}(2))$ denoted as <a,b,c,a+b,b+c,a+c,a+b+c>, one can clearly see that to accomodate 2 different lines, one needs to consider a subset of at least 5 points. This contradicts our construction in which we originally had 2 collections of 4 points each contained in two planes.

◇ In this case, we assume that the remaining 6 points do not contain any subset of size 3 which is dependent. This means that any subset of triplet of points from among these define a plane. So we will arbitrarily consider two disjoint triplets from this 6 remaining points, and consider their planes $P2$ and $P3$. Also note that points from triplet of $P2$ do not lie on $P3$, and vice-versa. For, we are assuming in this sub-case that 4 coplanar points do not exist among these 6 remaining points. Again in this case, *any 3* points will have at least 4 common hyperplanes to satisfy their degree requirements, and vice-versa. Hence we again have 4 points and hyperplanes being considered in construction, for each of the planes $P1$, $P2$ and $P3$. Note that there may be common points and hyperplanes used in this construction. We consider 2 separate sub-subcases

- ⋆ In this case, all the 10 hyperplanes of the required graphs lie within the set of union of sets of 4 hyperplanes each being considered per plane.

  Consider the 3 independent points of the triple of plane $P2$. None of these points can be same, or linear combination of any point reachable from plane $P1$. These 3 points were earlier also illustrated to be independent, and hence not reachable from $P3$ as well. A join of plane $P1$ and one different point reachable from plane $P2$ in the corresponding lattice

yields one different 3-d flat each, in the lattice. With respect to plane $P1$ where we are considering a set of 4 hyperplanes, it is obvious that in $\mathbb{PG}(5, \mathbb{GF}(2))$, at least 3 of these hyperplanes are independent. If the remaining hyperplane is dependent on 2 of these 3, then a complete 3-d flat is reachable from this set of 4 hyperplanes. By lemma 5, any point of $P2$ is reachable to at most 3 of the hyperplanes reachable from $P1$, that too when the 3 hyperplanes are part of a complete 3-d flat. Hence it is possible that a particular point reachable from $P2$ also reaches out to the unique 3-d flat , whenever it exists, and thus to the 3 hyperplanes of $P1$ from this 3-d flat. Since the join of different independent points of $P2$ with plane $P1$ gives rise to different 3-d flats, and since there is at most one 3-d flat embedded in the set of 4 hyperplanes being considered w.r.t. plane $P1$, at least two points reachable from plane $P2$ can only reach out to at most 2 hyperplanes reachable from plane $P1$. A similar conclusion can be reached with respect to the same 3 points of plane $P2$, and the hyperplanes reachable from $P3$, that are under consideration for this construction. In the best case, 2 distinct points out of the 3 points reachable from $P2$ have a degree of 3 towards planes $P1$ and $P3$, respectively. Hence at least 1(the remaining one) point reachable from plane $P2$ is reachable to at most two hyperplanes each, reachable from planes $P1$ and $P3$. A similar point can also be located on plane $P3$, the 3 points reachable from which have identical relation to the point sets of remaining 2 planes.

Without loss of generality, we further claim that at most 3 of the 10 hyperplanes used in construction remain outside of those considered for planes $P1$ and $P3$(or $P2$). When planes $P1$ and $P3$ are disjoint, they cover 8 of the 10 required hyperplanes of the construction. If the planes meet in a point, then by duality arguments, they also meet in a hyperplane. In which case, they cover 7 out of 10 required hyperplanes of the construction. If $P1$ and $P3$ meet in a line(the last possible scenario), then $P3$ has 1 point exclusively belonging to itself. Hence in all cases, either $P2$ or $P3$ has at most 3 points lying on it. On both these planes, we have located at least 1 point, which reaches out to at most 2 points each to the

remaining 2 planes. Hence in all scenarios, there exists at least one point who can reach out to at most 3+2+2 = 7 hyperplanes, thus invalidating the construction of this case.

- ⋆ In this case, all the 10 hyperplanes of the required graphs do not lie within the set of union of sets of 4 hyperplanes each being considered per plane. Hence, there is at least 1 hyperplane that is not covered by planes $P1$, $P2$ and $P3$, i.e. it is not reachable to either of these. By lemma 5 and the fact that in the assumption for this case, the triplet of points of $P2$ and $P3$ are not collinear, one can see that the points of the planes $P2$ and $P3$ provide degree at most 2 each to the above hyperplane. Additionally, it may provide a reach out to maximum 3 points lying on plane $P1$. By considering the planes $P1$, $P2$ and $P3$, we have exhausted all the points of the construction, and the maximum degree this particular hyperplane has achieved so far is 3+2+2 = 7, that is clearly not sufficient.

□

5.6 An Eigenvalue Based approach for getting ξ

The arguments are very similar to the ones given by [14]. Let $\mathbf{A} = (\mathbf{a_{ij}})$ be the $2n$ x $2n$ adjacency matrix of the bipartite graph $\mathbb{G}(V, E)$ of degree d, i.e., $a_{ij} = 1$ if there is an edge between the vertices indexed by i and j and $a_{ij} = 0$ otherwise.Let $\mathbb{S}$ be the set of vertices of the graph $\mathbb{G}$ that form the minimal configuration of failure. Let $\mathbf{x_s}$ be a column vector of size $2n$ such that every coordinate indexed by a vertex of $\mathbb{S}$ equals 1 and the other co-ordinates equal 0. Now, we have,

$$\mathbf{x_s^T A x_s} = \sum_{\mathbf{v} \epsilon \mathbb{S}} \delta_{\mathbf{G}_\mathbb{S}}(\mathbf{v}) \tag{5.6}$$

where $\delta_{G_\mathbb{S}}(v)$ is the degree of vertex v in the subgraph $G_\mathbb{S}$ induced by the vertex set $\mathbb{S}$ in $\mathbb{G}$.

Let $\mathbf{j}$ be the all-ones vector. $\mathbf{j}$ is the eigenvector of $\mathbf{A}$ associated with the eigenvalue d.

Define $\mathbf{y_s}$ such that

$$\mathbf{x_s} = \frac{\mid \mathbb{S} \mid}{\mathbf{2n}} \mathbf{j} + \mathbf{y_s} \tag{5.7}$$

$\mathbf{y_s}$ has $\mid \mathbb{S} \mid$ co-ordinates equal to $1 - \frac{|\mathbb{S}|}{2n}$ and $2n- \mid \mathbb{S} \mid$ co-ordinates equal to $-\frac{|\mathbb{S}|}{2n}$ and $\mathbf{y_s}$ is orthogonal to $\mathbf{j}$. Therefore, we can write

$$\mathbf{x_s^T A x_s} = \frac{\mid \mathbb{S} \mid^2}{4n^2} d\, \mathbf{j.j} + \mathbf{y_s^T A y_s}$$

Since $\mathbf{j.j} = 2n$, we have,

$$\mathbf{x_s^T A x_s} = \frac{\mid \mathbb{S} \mid^2}{2n} d + \mathbf{y_s^T A y_s} \tag{5.8}$$

Now, $\mathbf{y_s}$ is orthogonal to $\mathbf{j}$ and the eigenspace associated to the eigenvalue d is of dimension 1 ($\mathbb{G}$ is connected). Therefore, we have, $\mathbf{y_s^T A y_s} \leq \lambda \| \mathbf{y_s} \|^{\mathbf{2}}$ where λ is the second largest eigenvalue of $\mathbf{A}$. Considering the structure of $\mathbf{y_s}$ as explained above, we have,

$$\begin{aligned} \| \mathbf{y_s} \|^2 &= \mid \mathbb{S} \mid (1 - \frac{\mid \mathbb{S} \mid}{2n})^2 + (2n- \mid \mathbb{S} \mid)(\frac{\mid \mathbb{S} \mid}{2n})^2 \\ &= \mid \mathbb{S} \mid - \frac{\mid \mathbb{S} \mid^2}{2n} \end{aligned}$$

Since we are looking for subgraphs in which the degree of each vertex is at least a certain value (say γ), after combining the various equations and inequalities above, we get,

$$\begin{aligned} \gamma \mid \mathbb{S} \mid &\leq \mathbf{x_s^T A x_s} \\ &= \frac{\mid \mathbb{S} \mid^2}{2n} d + \mathbf{y_s^T A y_s} \\ &\leq \frac{\mid \mathbb{S} \mid^2}{2n} d + \lambda \| \mathbf{y_s} \|^2 \\ &= \frac{\mid \mathbb{S} \mid^2}{2n} d + \lambda(\mid \mathbb{S} \mid - \frac{\mid \mathbb{S} \mid^2}{2n}) \end{aligned}$$

Finally, since $\mid \mathbb{S} \mid > 0$, we can cancel it from both sides and the expression that we arrive at is,

$$\mid \mathbb{S} \mid \geq \frac{2n(\gamma - \lambda)}{d - \lambda} \tag{5.9}$$

Because of duality of points and hyperplanes, we get $\xi = \frac{|\mathbb{S}|}{2}$. Thus,

$$\xi \geq \frac{n(\gamma - \lambda)}{d - \lambda} \tag{5.10}$$

The above formula is also stated in [6] in the context of finding the minimum distance of the code proposed by him using $\mathbb{PG}(2, q)$. In our case, the above formula should be used only for $\epsilon \geq 15$. For all practical values of ϵ, the combinatorial methods give a very tight bound(under the assumption stated in the previous chapter) as they have have been found by looking at the minimal configuration required for a failure to occur.

Chapter 6

Applications

In the chapter, we bring about applicability of the coding scheme in multiple areas, some of them in detail.

6.1 Application to CD-ROM

As indicated in section 4.3, we have also benchmarked the burst error correcting capability of our code against CD-ROM decoding based on ECMA-130. Based on this benchmarking, we propose 2 novel schemes for CD-ROM encoding and decoding. These schemes are based on the expander-like codes that have been discussed in chapter 4. Application of these codes at various stages of CD-ROM encoding scheme(and correspondingly in decoding scheme) *substantially* increases the burst error correcting capability of the disc drive.

To start with, we *recall* that the major part of error correction of the CD-ROM coding scheme occurs during the two stages, RSPC and CIRC. On the encoder side, RSPC stage comes before CIRC stage, while on decoder side, CIRC stage comes before RSPC stage. To get an idea of the average error correction capabilities of CDROM scheme, we simulated the CIRC and the RSPC stages of the ECMA standard in MATLAB. Because the CIRC uses delay elements to implement the interleaving, any frame arriving at the input of the second RS decoder has data symbols from the preceding 109 frames. Thus to gauge the error correction capabilities, we start with 218 frames of 32 symbols each. The 109th frame is our 0th frame. After considering the effect of interleaving, this frame will require data symbols from the previous 109 frames. Errors are distributed over the last 109 frames and the error correction capability is observed. The details of CIRC and

RSPC stages are as follows.

CIRC This stage leads to interleaving of codeword symbols. The massive interleaving done here is mainly responsible for the burst error correction. In a frame of 6976(=32*109*2) symbols, it can correct on an average 240 consecutive corrupt symbols. This amounts to approximately 2000 bits. If the burst is placed appropriately i.e. at the end of one frame of 6976 symbols and at the beginning of the next frame, then the CIRC can potentially correct 4000 bits of burst errors. The success of error correction thus *depends on the location of the burst.*

RSPC After the CIRC stage during decoding, some of the burst errors get corrected, and others get re-distributed among F1-Frames due to de-interleaving. Due to this spreading out happening due to de-interleaving, the remaining ones can be considered as *random errors.* The RSPC stage in decoding then serves to correct these errors using RS decoding as erasure decoding. Success of this stage depends heavily on the CIRC stage marking corrupt symbols as *erasures.* If we consider only error detection and correction, the CIRC + RSPC system on an average corrects a burst of 270 symbols in a frame of 6976 symbols.

The schemes we propose involve replacing one or both of the CIRC and the RSPC with encoders and decoders based on our expander-like codes. This happens in such a way, that we maximize burst error correction, without suffering too much on the data rate.

6.1.1 Scheme 1

As discussed, the CIRC+RSPC subsystem in decoder can detect and correct a burst of about 270 erroneous symbols, in a frame of 32x218(=6976) 8-bit symbols. In the first scheme, we replace this subsystem with 4 decoders for our expander-like codes, $\mathbb{C}$. Hence the RS subcodes used in $\mathbb{C}$ have block length d as 31(symbols). Further, we fix their minimum distance as $\epsilon = 7$. The output of corresponding 4 encoders is further interleaved to improve performance, and de-interleaved on receiving side. Let k be the number of message symbols in each subcode. For Reed-Solomon code, which are *maximum distance codes*, we have $n - k + 1 = \epsilon$, which implies that the data rate of the subcode is $\frac{k}{n} = \frac{25}{31} = 0.806$.

To construct these subcodes, we take a (255,249,7) RS code, and choose the first 31 message symbols; setting the other message symbols to 0. Hence we have a *shortened RS code* with each symbol (still) represented by 8 bits. We need to use this shortened RS code because each data symbol in the CD-ROM is a byte long.

For the overall code $\mathbb{C}$, the data rate is equal to $(2 * r - 1)$[12], where r is the rate of the (RS) subcode. Hence the rate for codes used in each of our encoders/decoders is $2 * 0.806 - 1 = 0.612$. Thus, the number of message symbols for each decoder is equal to $1953 * 0.612 = 1197$. The rest are therefore parity symbols.

In terms of frames, we set the cumulative input of the encoders, and correspondingly the cumulative output of decoders, to a stream of 199 frames, each having 24 symbols payload. Assuming that each symbol can be encoded in 1 byte, this leads to generation of 4776 bytes. With 12 padding bytes added to it, we can re-partition this bigger set of 4788 bytes into 4 blocks of 1197 bytes each(4*1197=4788). Each block of 1197 source symbols can then be worked upon by 4 *parallely* working encoders. After encoding each block to 1953 symbols, one of the extra added(padding) bytes is removed from each encoder giving 244 frames of 32 bytes of data. Every RS decoder has $\epsilon = 7$, which implies that it can detect and correct upto 3 errors. Thus, each encoder for $\mathbb{C}$ will give a burst error correcting capability of $63 * 3 = 189$. Since 4 of such encoders work in interleaved fashion, we get a minimum burst error detection and correction capability of 756 among 244 frames. This is opposed to 270 in 218 frames in the case of CIRC+RSPC subsystem.

The detailed design of each of the 4 corresponding decoder is provided in chapter 7.

Advantages

1. A massive improvement in burst error correction : 270 in 32x218 symbols for CIRC+RSPC system, versus 756 in 32x244 in our scheme.
2. The code rate achieved is also comparable to the one for CIRC+RSPC subsystem. In the latter case, the data rate is 24/32=0.75, whereas in our case it is 0.62.

Disadvantage

This scheme is hardware expensive due to use of many parallel RS decoders. Also, the high throughput of our decoder is not utilized. We can reduce the resource complexity by time-multiplexing the decoders, and also folding the architecture of

one decoder itself. As far as the throughput is concerned, we are limited by the stages before and after the CIRC+RSPC subsystems(see [7]). Hence, even though our decoder is faster, the advantage is not seen.

6.1.2 Scheme 2

This scheme is a more hardware economical scheme, which also increases the burst error correcting capability. Since our decoder also has a very good random error correcting capability, we can achieve an error correction advantage by replacing just the RSPC stage with our encoder/decoder. Two of our encoders can take the place of the RSPC encoder in this scheme. Data from these encoders is then interleaved, and passed on to the CIRC. In the decoding stage, after CIRC, there is correspondingly de-interleaving followed by decoding based on our code.

This scheme has the advantage that it increases the error correction capability. It also matches the code rate of CIRC: 0.75 for CIRC, versus 0.74 for our decoder. Also, it is a hardware economical scheme. MATLAB simulations show that the burst error rate goes up from 270 for CIRC+RSPC subsystem, to more than 400 for CIRC and our encoder. Tables 6.1 and 6.2 show some simulation results for this scheme.

No. of errors	failures
270	2
300	45
400	86

Table 6.1: Response to Burst errors for CIRC+RSPC

No. of errors	failures
400	7
450	17
500	26

Table 6.2: Response to Burst errors in Scheme 2

6.2 Application to DVD-R

The same class of codes can also be applied to evolve encoding and decoding for DVD-ROM. This particular application of the new coding scheme also brings out the fact that taking a bipartite graph $\mathbb{G}$ from a higher-dimensional projective space can be advantageous in terms of better rate and better error correction capacity.

6.2.1 Present DVD-R Error correction blocks

The details of the ISO/IEC standard implementation have been given in [10]. Here, we provide an overview the main error correcting block, which will be used later to derive a new correcting scheme.

The data received from the host, called *Main Data*, is formatted in a number of steps, before being recorded on the disk. It is transformed **successively** into following.

- a Data Frame,
- a Scrambled Frame,
- an ECC Block,
- a Recording Frame, and
- a Physical Sector.

Data Frames

A Data Frame consists of 2064 bytes arranged in an *array* of 12 rows, *each* containing 172 bytes. The first row starts with three fields spanning 12 bytes interval, which is followed by 160 *Main Data* bytes. The next 10 rows each contains 172 *Main Data* bytes. The last row contains 168 Main Data bytes, followed by four check bytes of Error Detection Code (EDC). Thus there are 2048 bytes of *Main Data* in each Data frame of size 2064.

Scrambled Frames

The 2048 *Main Data* bytes are later scrambled according to the procedure described in section 17 of [10]. Scrambling bytes are generated with a Linear Feedback Shift Register(LFSR), and each data byte is XOR-ed with a corresponding scrambling byte.

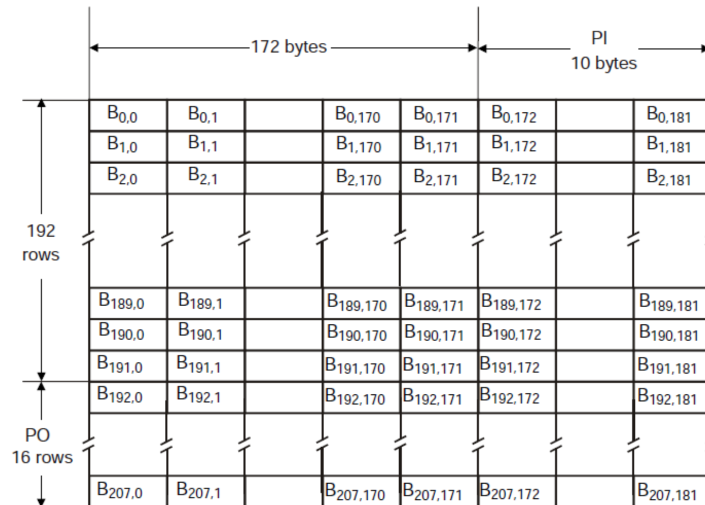

Figure 6.1: ECC Block of DVD-R

ECC Block

An ECC Block is formed by arranging 16 consecutive Data Frames, **after scrambling**, in an array of 192 rows of 172 bytes each; see figure 6.1. To each of the 172 columns, 16 bytes of *Parity* of an **Outer Code** are added. This results in a block having 208 rows having 172 bytes each. To each resulting 208 rows, 10 byte of *Parity* of an **Inner Code** are added. Thus a complete ECC Block comprises 208 rows of 182 bytes each. The bytes of this array are identified as $B_{i,j}$, where i is the row number and j the column number.

Thus the ECC block is nothing but a *Reed Solomon Product Code*, with the inner code being a RS(182,172,11) code, and the outer code being a RS(208,192,17) code.

Recording Frames

Sixteen Recording Frames are obtained by *partitioning* an ECC block into 16 frames, while *simultaneously* doing interleaving. This is achieved by interleaving one of the 16 PO rows(see figure 6.1) at a time *after every* 12 rows of an ECC Block. Figure 6.2 brings out this partitioning graphically.

The rate of this encoding can simply be calculated as $2064 * 16/37856 = 0.8724$.

6.2.2 Analysis of Error Correction in DVD-R

As mentioned earlier, the main error correction in DVD-R is provided by the RSPC block, which consists of an inner RS(182,172,11) code and an outer RS(208,192,17) code. The

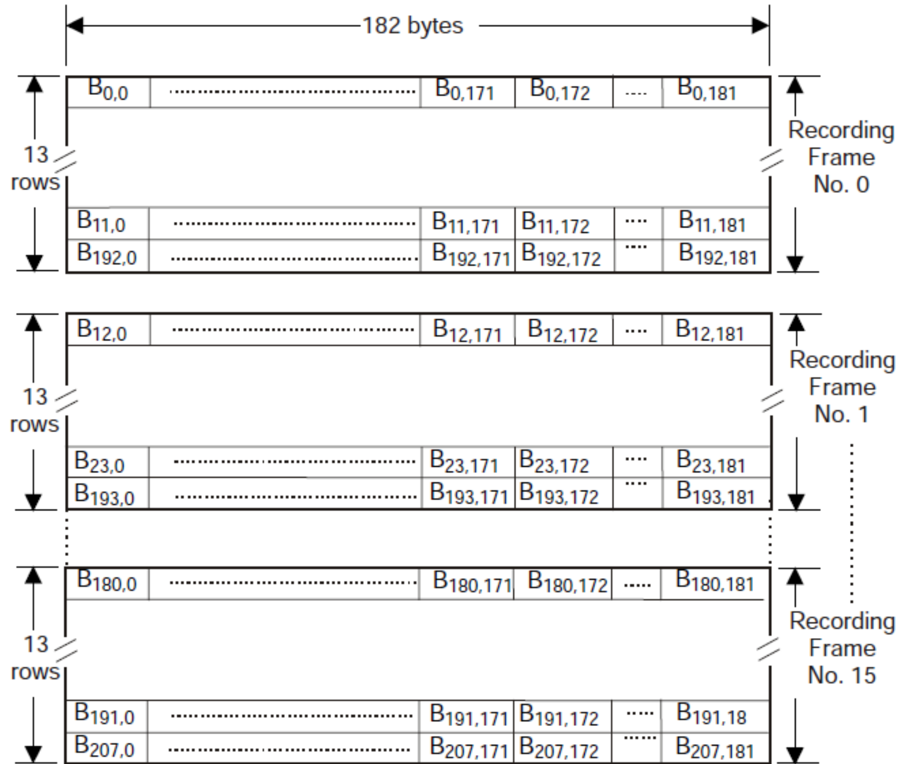

Figure 6.2: Partitioning ECC Block into Recording Frames

inner code can detect and correct up to 5 errors $((11-1)/2)$, while the outer code can detect and correct 8 errors.

Random Errors

The minimum distance of the overall code is $17*11 = 187$. Hence in a block of $(208*182) = 37856$ bytes, the number of random errors should be less than or equal to $(187-1)/2 = 93$, in order that the block be decoded completely. [10] further points out that the number of errors in 8 consecutive ECC blocks must be less than or equal to 280, for correct decoding.

Burst Errors

The RSPC code used in DVDs is very robust towards burst errors. Since the data is arranged in a matrix fashion(see figure 6.1), it is easy to calculate the *worst case* amount of errors than can be corrected by the inner and outer codes.

If we *do not* consider *erasures*, then the inner code can correct a burst of 5 errors, while the outer code can correct a burst of 8 errors. The biggest burst of errors that we can be corrected in the product code can be derived as follows.

- We can allow for 8 rows of 182 bytes to get completely corrupted. Then, these will be detected and corrected by the outer code's decoders, which operate on columns of the matrix.
- We can further have an additional 5 bytes getting corrupted in the row above and below the set of 8 rows. Since inner code decoding happens first, these errors will be corrected by the inner code's decoders, after which the outer code's decoders can correct the *remaining* 8 rows of contiguous corrupted data.

The above two points make it clear that without considering erasure decoding, we can still detect and correct $8 * 182 + 5 * 2 = 1466$ errors in a burst, in one block of 37856 bytes. If we put two ECC blocks back to back, the number of errors hat can be detected and corrected increases. We can add another 8 rows of 182 corrupt bytes in ECC block 1, 5 bytes before these 8 rows(corrected by inner codes), 8 rows of 182 corrupt bytes in ECC block 2 and 5 bytes after these 8 rows. This gives a total of $8*2*182+5*2 = 2922$ errors's burst that can be corrected simultaneously.

If we allow for the inner decoding to mark as **erasures**, all the bytes of a codeword that has more errors that it can correct, we can increase the overall burst error correction even further. With only erasures, the outer code can correct 16 erasure symbols. Thus the total error correction capacity will be $16 * 2 * 182 + 5 * 2 = 5834$ bytes. This is the **absolute maximum burst** that this stage can handle.

6.2.3 A New ECC Scheme for DVD-R

We now present a scheme based on our expander-like codes, $\mathbb{C}$. This scheme, which uses decoders for code $\mathbb{C}$, can improve the error correction capacity of the DVD scheme presented above.

In this scheme, we subtitute the RSPC stage of the DVD encoding by encoders of code $\mathbb{C}$. These encoders are therefore employed during the transformation of *Data Frames* into *Recording Frames*. In order to be *compliant* with the rest of the standard, we must output the encoded data as a block having same size as one recording frame, i.e. $2366(= 13 * 182)$ bytes. Thus, we take 2064 bytes per data frame as input, and need to output 2366 bytes in the format of recording frames.

Since we need higher rate and a better error correction capacity, we need to look at

higher dimensions of PG. Higher dimensional of PG lead to better expansion properties, which leads to better error correction.

In one such scheme, we consider $\mathbb{PG}(8, GF(2))$. This geometry contains $2^9 - 1 = 511$ points and hyperplanes. Further, each point is contained in $2^8 - 1 = 255$ hyperplanes and similarly, each hyperplane contains 255 points. Thus, the degree of each vertex in the bipartite graph we construct will be 255, and the number of vertices in each partition of the bipartite graph will be 511. Each edge of the graph represents an 8-bit symbol. Thus, the total number of symbols in the overall code $\mathbb{G}_8$ will be $255 * 511 = 130305$.

Each vertex of the graph is a RS decoder which corresponds to a RS(255,239,17) code. Thus, each vertex can detect and correct 8 errors. The rate of the sub-code in this case is $r = 239/255 = 0.9373$. The overall rate of the code is at least $2 * r - 1 = 0.8745$, which is marginally better than the rate in the ISO/IEC standard for DVD encoding. The overall burst error correction capability *without erasures* will be $8*511 = 4088$ bytes, which is much greater than 2922.

We also have $2366 * 55 = 130130$; thus, we can output 55 frames in one round of encoding. Hence we choose to encode 175 lesser source symbols, than required by $\mathbb{G}_8$. These remaining 175 source symbols will be set to 0, a padding byte value. These 175 padding bytes are later dropped after encoding, to get an overall encoded block of 130130 bytes. By using a systematic encoding matrix, the locations of these padding bytes remain intact during the process of encoding. Because of using 175 lesser source symbols for encoding, the overall rate drops to 0.8732.

It is reasonable to assume pipelining while decoding, because of the application of DVDs in many real time applications like video etc. Thus, two frames of 130305 bytes put together can detect and correct a burst of $4088 * 2 = 8176$ bytes. This number is quite bigger than the absolute maximum burst of 5834 brought out earlier. Thus, even if we do not consider erasures, we have, at a slightly **higher rate**, achieved a *much better* burst error correction capability.

Thus, at the price of extra hardware for decoding, and memory for storing the large frame sizes, we have a linear time decoder with an exceptional burst error correction capability.

As far as random errors are concerned, our average case performance will easily surpass the existing standard. The existing standard specifies that the number of random

errors in 8 consecutive ECC blocks must be less than or equal to 280. Our frame size of 130305 corresponds to approximately 3.44 consecutive ECC blocks. Preliminary MATLAB simulation results show that around 1990 random errors are always corrected in one iteration of the decoding itself.

As a concluding note, the design of expander-like code to be used in DVD-R application is not limited to choice of RS(255,239,17) code. In fact, as we increase the minimum distance from 17 to 21(i.e. RS(255, 235, 21) code), we get a burst error correction 511*2 more than 4088, that is, 5110, while the drop in overall code rate drops by just 0.03. If r expander-like encoders are used simultaneously, and their outputs interleaved, then also the error-correction capability goes up by factor of r.

6.3 Possibility of Building Other Expander-like Codes

The reason for choosing Reed Solomon codes as component codes has been that they are *maximum distance separable* codes. Hence they achieve the *maximum rate* for a given value of 'minimum distance' ϵ. Other expander-like codes, with possible future applications, can be built by using different component codes also. For example, we could use concatenated codes in place of RS codes as subcodes. Concatenated codes have been shown to achieve the *Zyablov* bound(which is tighter than the Gilbert-Varshamov bound). Hence, they could offer a greater advantage for the rate-distance trade-off.

6.3.1 Choice of Projective Space

We have given extensive results for the construction derived from $\mathbb{P}(5, \mathbb{GF}(2))$. We are in **no way restricted** to this dimension. One main reason for picking up this dimension was that the degree of each vertex in the graph derived is 31, which is very close to the F2 frame size of 32 bytes in CD-ROMs. Thus, the choice of this dimension would help us benchmark the performance of our code against an industry standard. At the same time, the size of the code was practical enough for the software implementations in MATLAB to finish in a reasonable amount of time.

One can, in fact, use higher dimensional spaces to construct expander-like codes having utility elsewhere. For example, in section 6.2, we use an 8-dimensional projective space to have an expander-like code construction, which can be potentially used for DVDs.

6.3.2 Choice of RS Code

Once again, we chose the parameters of the RS code to enable efficient schemes, that could be applicable to data storage systems. In case of application to CD-ROM, since the smallest symbol size is a byte, we had to use a RS code of size 255. The degree of each vertex in the bipartite graph used there being 31, we had to *shorten* the RS code size to 31. This shortening is done simply by dropping the message symbols from 32 onwards. More specifically, if the minimum distance is ϵ, symbols 32 to $255 - \epsilon + 1$ are assumed to be zero. Similar shortening schemes can be employed in other applications as well.

In turn, the minimum distances are chosen to get the overall rate of code to **match** the rate used by applications such as the CD-ROM encoding scheme. Additional error correction capability is achieved by adding a small amount of interleaving.

Chapter 7

Detailed Design Description of Hardware Prototyping

As discussed in section 6.1, we can use multiple decoders based on our expander-like codes to have 2 new decoding schemes, that give better performance than the existing decoders. The design details for each such decoder for new, expander-like code targeted for CD-ROM application, is provided in the remainder of chapter.

7.1 Construction of the Graph

The bipartite graph is constructed by using the point-hyperplane incidence relations of $\mathbb{P}(5, \mathbb{GF}(2))$. The points are generated using a primitive polynomial, which give the tapping points in a linear shift feedback register(LFSR). The primitive polynomial used to generate $\mathbb{GF}(2^6)$ is $x^6 + x + 1$. The points of this projective space are given in table 3 in the appendix. To identify the points lying on a particular hyperplane, first one has to construct the 5-dimensional vector subspace of the 6-dimensional vector space $\mathbb{GF}(2^6)$. Then the points that correspond to vectors lying in that subspace can be taken as the points on the particular hyperplane. One such hyperplane, represented by its point set, is $(0, 1, 2, 3, 4, 6, 7, 8, 9, 12, 13, 14, 16, 18, 19, 24, 26, 27, 28, 32, 33, 35, 36, 38, 41, 45, 48, 49, 52, 54, 56)$. The remaining 62 hyperplanes can be obtained by applying *shift automorphism*[11] to this hyperplane.

Let the hyperplanes, numbered from 0 to 62, form one side, called **A**, of the bipartite graph between points and hyperplanes. Similarly, let the points, numbered from 63 to

125, form another side of the graph, called **B**. Based on this numbering, Table 1 in the appendix gives the complete list of the hyperplanes and their adjacent vertices. As described in section 4.3, to improve the burst error capability, we need to number the edges such that consecutive edges always go to different vertices. Hence we label edge between vertex 0 and 63 as edge number 1, between vertex 1 and 64 as edge number 2 and so on...

Viewed as a *computation graph*, every vertex of this graph maps to a *RS decoding computation*. The input symbols to each of these decoders correspond to the edges which are incident to the vertex in question. During decoding, these symbols are provided as inputs in a specific order: message symbols first and then parity symbols. This order also gets reflected in the **reduced row echelon form** of the corresponding generator matrix, $\mathbb{G}$. The edges incident on a vertex of side **A**(say, $V1$) are sorted with respect to increasing index numbers of the vertices reached in side **B**. This is the order in which the corresponding symbols are fed to the RS decoder represented by $V1$. A similar strategy is used for ordering inputs for RS decoders of vertices on side **B**.

While generating the bipartite graph and its edge/vertex labels, we prefer the alternate representation described in equation 7.1. To recall, the points of an n-dimensional projective space over a field $\mathbb{F}$ can be taken to be the equivalence classes of nonzero vectors in the (n + 1)-dimensional vector space over $\mathbb{F}$. Vectors in an equivalence class are all scalar multiples of one-another. These vector being one-dimensional subspaces, they also represent the rays of a vector space passing through origin. The orthogonal subspace of each such ray is the **unique** n-dimensional subspace of $\mathbb{F}^{n+1}$, known as hyperplane. Each vector h of such orthogonal subspace is linked to the ray, p, by a dot product (addition is *modulo* 2 because of $GF(2)$) as follows.

$$p_0h_0 + p_1h_1 + \cdots + p_nh_n = 0 \tag{7.1}$$

where p_i is the ith coordinate of p. This uniqueness implies bijection, and hence a vector p can be used to represent a hyperplane subspace, which is exclusive of this vector as a point. Due to duality, similar thing can be said about a hyperplane subspace. It is important to note that the above equation **does not** imply orthogonality. It is just a way to generate the hyperplane and point subspaces and is a *convenient representation* of incidence. Using this representation, we can find the vectors representing the hyperplanes,

containing a given set of points, and then correlate them to the decimal numbers used to represent the hyperplanes.

The representation of points and hyperplanes is further dependent on the representation of the underlying vector space. For the vector space, we use a canonical representation. In this representation, we set those positions in a vector as 1 which correspond to that power of x existing in the 1-D subspace. For example, point 0 is represented by 000001, point 2 as 000100, point 8 as 001100 and so on. This representation can also easily be derived from table 3 in the appendix.

Next, we describe two possible hardware implementation strategies, for scheme 2 of alternative CDROM decoding, introduced earlier in section 6.1.2. We also give details of its prototype implementation on a Xilinx XUPV506 board based on LX110T FPGA. But first let us recall the decoding algorithm.

7.2 Recapitulation of Decoding Algorithm

Given a $\mathbf{d}$-regular PG bipartite graph $\mathbb{G}$, let the set of its vertices be $V = \mathbf{A} \cup \mathbf{B}$, where $|\mathbf{A}| = |\mathbf{B}| = n$. Every edge of $\mathbb{G}$ has one endpoint in $\mathbf{A}$ and one in $\mathbf{B}$. For any vertex v of $\mathbb{G}$, let the subset of edges incident to v be labeled as

$$E_v = \{v_1, v_2, \cdots, v_{\mathbf{d}}\}$$

Every edge of the graph represents an 8-bit symbol.

To an expander-like decoder based on this graph, we can give a received vector $\mathbf{x}$ as input, where the length of $\mathbf{x}$ is $N = \mathbf{d} \cdot n$. Every entry in $\mathbf{x}$ is an 8-bit symbol. The first iteration of the algorithm consists of applying complete RS decoding for the code induced by E_v for every $v \in \mathbf{A}$. Hence we try to replace, for every $v \in \mathbf{A}$, the vector $x_{v_1}, x_{v_2}, \cdots, x_{v_{\mathbf{d}}}$ by one of the closest codewords of $\mathbb{S}$. However, if a particular vertex detects more errors than it can correct, it skips its localized decoding. This is because while using Berlekamp-Massey's algorithm for RS decoding [2], it is usually possible to calculate whether the degree of errors in the current input block of symbols to the decoder be corrected or not. If not, then the algorithm can be made to skip decoding, thus preserving the errors in the input block. This special step in algorithm, due to decoding being RS decoding, helps in *reducing* the number of *extra errors* introduced by a vertex,

if the decoding fails. However, it is possible that if the codeword incident on a vertex is corrupt enough, the RS-decoding algorithm is fooled into outputting the wrong codeword.

7.3 Decoder Implementation Strategy 1

As discussed in previous section, decoder for $\mathbb{C}$ has inherent parallelism in the sense that the Reed Solomon decoders corresponding to vertices on one side of the bipartite graph can work in parallel, every iteration. The first strategy utilizes this parallelism.

The strategy is to have a *master processor*, which is responsible for data distribution to its slaves. The slave processors are essentially the set of RS decoders. After decoding the subcode, the slaves send back the corrected subcodes to the master. 63 of such RS decoders can work in parallel, as dictated by a master, in our case. The master processor would be required to serve all these slaves with data at the same time, to get the highest throughput possible. As such, the master would be completely idle during the time the slaves are performing their decoding. This strategy is represented in the figure 7.1.

The limitation of this strategy is that though it is easy in implementation, for every data transfer along various edges at an iteration boundary, 2 hop communication is required. Iterative decoding of all kinds being data flow intensive systems, this limitation can severely affect the system throughput. Also, another role/advantage of master processor, that of buffering data which has to be later forwarded to a different slave then the source slave, is not required in our system.

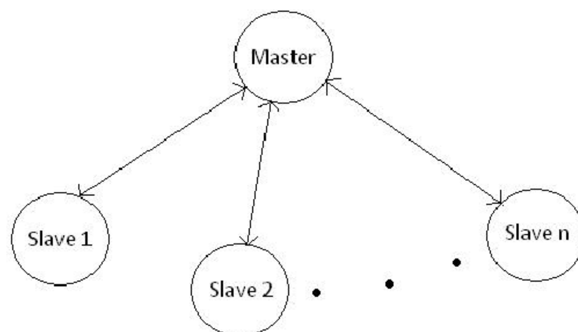

Figure 7.1: Strategy 1

7.4 Decoder Implementation Strategy 2

A more efficient strategy is to utilize projective geometry properties to **fold** the parallel vertex computations, such that the number of RS decoders required to implement the decoder for $\mathbb{C}$ is only a factor of order of $\mathbb{G}$. This saves a lot of resources, and can fit on even small FPGAs. The penalty is paid in terms of its tradeoff with decoding time. In one example of 7-fold folding, approximately 14 machine cycles are required to finish one iteration of decoding of $\mathbb{C}$, as opposed to 2, if we used 63 RS decoders used in design.

In the remaining sections, we describe the design of decoder as per this strategy.

7.5 Interconnect of Decoder

The interconnect of the current implementation is based on 63-vertex bipartite graph $\mathbb{G}|\mathbb{B}$, whose construction has been described in section 11.1. Edges of this graph depict the data to be processed by the RS decoders. The data path is inherently parallel for one partition of the graph. (No vertices share an edge in one partition). The data path has been folded onto a smaller graph, as described next.

7.5.1 Folding the Interconnect

To enable the folding of computation, we need to partition the graph $\mathbb{G}$ in a particular way. The requirement is that one should be able to assign groups of data(edges) to RS decoders in a manner which ensures conflict-free memory accesses and complete hardware utilization. The latter requirement means that no decoder remains idle while other decoders are working. Also, the folding and the scheduling should be such that data re-distribution between memory blocks should be minimized.

Projective geometry offers a way of partitioning of the graph efficiently such that data redistribution between memory blocks is not required. For efficient folding, we make use of the projective subspace which is symmetric with respect to both points and hyperplanes. For $\mathbb{P}(5, \mathbb{GF}(2))$, the plane(2-d subspace) is the corresponding subspace, that contains 7 points, and is contained in 7 hyperplanes.

The idea is in terms of reachability, one should be able to form a family of 9 subsets of the set of 63 vertices corresponding to the points, each subset defining a **disjoint plane**.

Planes being disjoint implies that the 9 subsets of points are themselves disjoint. By duality in projective space lattices, the corresponding hyperplanes also must be disjoint for folding the computations. Such a configuration exists, and can be found heuristically as explained below.

To form such partition, we start by generating a 6-dimensional vector space (excluding the zero vector), where each vector can represent a point(or its orthogonal hyperplane; see equation 7.1). Now, we can generate a list of all possible planes by selecting groups of 3 linearly independent vectors and their linear combinations as a set of 7 points that constitute a plane. The set of all planes (totally 1395) are generated by enumerative techniques. This is because the other way, that of using complete automorphism group of $\mathbb{P}(5, \mathbb{GF}(2))$ to generate all the planes, has so far not been worked out. For each plane, we can generate the list of hyperplanes associated with it.

The heuristic algorithm to find the partition starts with the first plane on the list of all the planes. After selecting this plane (let us call it $P1$), it makes a pass through the list of planes to find all the possible disjoint planes (*i.e.* all those planes which have no point in common with $P1$). If less than 9 planes are found, the second plane is selected as $P1$ and the checking is done from the 3rd plane onwards. The algorithm stops when a set of 9 disjoint planes are found.

We sped up the process in our construction by starting with a set of 3 planes, 2 orthogonal and the 3rd got by adding individual points of the first 2 planes. To identify these 3 disjoint planes was straightforward, as we are dealing with a six-dimensional vector space. These 3 planes are as follows.

Sr. No.	Points						
1	000001	000010	000011	000100	000101	000110	000111
2	001000	010000	011000	100000	101000	110000	111000
3	001001	010010	011011	100100	101101	110110	111111

Table 7.1: First 3 Disjoint Planes

The tables below present the points and hyperplanes of the 9 planes that will be used for hardware implementation. The points and hyperplanes are given by their vector space representation.

Once such a partition has been chosen, we can assign one decoder to each plane such

Sr. No.	Points						
1	001001	010010	100100	011011	101101	110110	111111
2	000001	000010	000100	000011	000101	000110	000111
3	001000	010000	100000	011000	101000	110000	111000
4	001010	010100	100011	011110	101001	110111	111101
5	001011	010110	100111	011101	101100	110001	111010
6	001100	010011	100110	011111	101010	110101	111001
7	001101	010001	100010	011100	101111	110011	111110
8	001110	010111	100101	011001	101011	110010	111100
9	001111	010101	100001	011010	101110	110100	111011

Table 7.2: Points of 9 Disjoint Planes used in Folding

Sr. No.	Hyperplanes						
1	001001	010010	100100	011011	101101	110110	111111
2	001000	010000	100000	011000	101000	110000	111000
3	000001	000010	000100	000011	000101	000110	000111
4	001011	010100	011111	100001	101010	110101	111110
5	001110	010011	011101	100111	101001	110100	111010
6	001111	010001	011110	100011	101100	110010	111101
7	001100	010101	011001	100010	101110	110111	111011
8	001101	010111	011010	100110	101011	110001	111100
9	001010	010110	011100	100101	101111	110011	111001

Table 7.3: Hyperplanes of 9 Disjoint Planes used in Folding

that it can work on behalf of all decoding computations mapped to all the points and hyperplanes reachable from that plane. Each decoder will need memory associated with it, since the decoding is multi-cycle operation. Each memory block is designed to hold $31 \times 7 = 217$ bytes(one symbol is one byte in our case) of data.

In section 7.1, while generating the bipartite graph, we had numbered the hyperplanes from 0 to 62, and the points from 63 to 125. Based on this numbering of vertices, we also number the edges in a particular order so as to maximize the burst error correcting capability. This ordering has been discussed already in section 4.3. We recall from there

that vertex 0 gets data symbols $(1, 64, 127, \ldots, 1890)$, and so on.

As shown in table 3, each decimal representation of a point has a unique vector representation. Similarly, each hyperplane also has a unique decimal and vector representation. When we divide the hyperplanes and the points as we have done in tables 7.3 and 7.2, we are essentially re-ordering the vertices of the graph so as to enable efficient folding of computation. Consider the partition of the graph that corresponds to the hyperplanes. As explained in 7.1, the vertices were labeled as $\{0, 1, \ldots, 62\}$. The re-ordered partition of the graph is given in table 7.4. Now the vertices on the hyperplane side are arranged in the order $\{0, 37, 51, \ldots, 62, 1, 2, 3, \ldots, 52\}$. A similar re-ordering has occurred on the points side as well. This re-ordering naturally changes the order of edge numbers incident on a vertex.

To **elaborate** this re-ordering, consider a small bipartite graph of 4 vertices on each side, with each vertex having a degree of 4. Figure 7.2 shows two representations of the same graph: one shows the routine labeling, while the other shows vertices re-ordered labeling. The edge numbers are written on the edges. This numbering is similar to our construction(*i.e.* consecutive edge numbers go to different vertices).

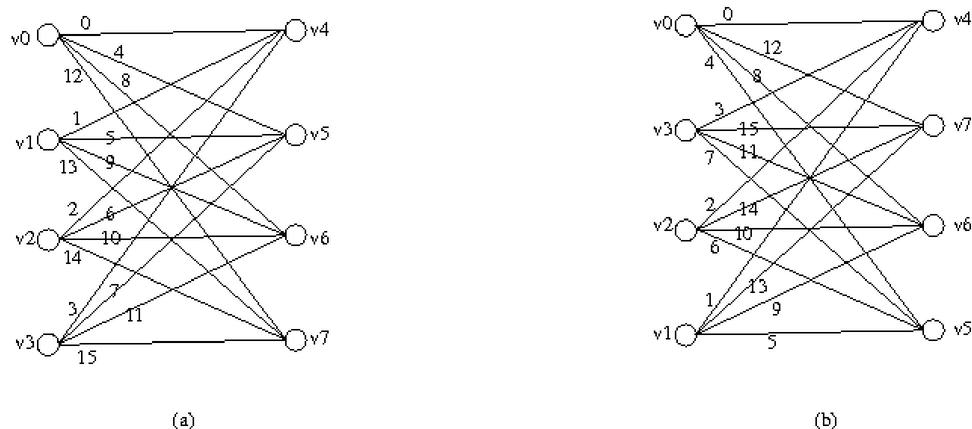

Figure 7.2: Example to illustrate Re-ordering

Consider vertex $V0$ in figure 7.2(a). If this were a RS decoder, the symbols numbers $\{0, 4, 8, 12\}$ would be given to it with 0th symbol being the first, and 12th symbol being the last. Suppose the code was designed to have two message symbols and 2 parity symbols. Then, symbols 0 and 4 would be message symbols, and symbols 8 and 12 would be corresponding parity symbols.

In part (b) of the figure, vertices $V1$ and $V3$ are interchanged. Similarly, vertices

5 and 7 are interchanged. Thus figure 7.2(b) is the same graph with vertices re-ordered. Now, the symbols to Vertex $V0$ are given in the order $\{0, 12, 8, 4\}$. Thus, symbols 0 and 12 become the message symbols, and symbols 8 and 4 become the parity symbols. A similar analysis can be done for the vertices on the other side also. This reordering of input symbols is used for pre-ordering of data distribution in the memory blocks, as explained in the next section.

The above example illustrates how the re-ordering of vertices causes the edges to also re-order. This fact must be taken into account at the time of encoding also. Appropriate re-arrangement of the columns of the generator matrix is required to be done there.

Plane No.	Hyperplane Vertices	Point Vertices
1	0,37,51,56,57,61,62	63,64,65,69,70,75,89
2	1,2,3,40,54,59,60	66,67,68,69,73,78,92
3	6,20,25,26,30,31,32	95,96,97,101,102,107,121
4	5,7,10,11,41,50,58	74,76,77,82,86,106,122
5	9,23,28,29,33,34,35	98,99,100,104,105,110,124
6	12,16,21,22,24,39,55	87,88,91,93,103,111,120
7	13,19,27,38,42,45,18	83,90,94,108,109,114,117
8	14,17,36,43,46,48,49	81,84,112,113,115,118,125
9	4,8,15,53,44,47,52	71,79,80,85,116,119,123

Table 7.4: Re-ordered Graph Vertices

To understand the impact on schedule due to folding, recall that a decoding iteration consists of RS decoding on vertices on one side of the bipartite graph. If we call iterations on 2 sides as phases, then *phase 1* corresponds to each of the 9 RS decoders decoding for the 7 hyperplanes associated with its plane. Similarly, *phase 2* corresponds to the RS decoding for the 7 points associated with each plane.

It is easy to prove that any point on a plane is incident on **exactly** 3 hyperplanes of a **disjoint plane**, and vice-versa. In Phase 1, for decoding symbols that correspond to a hyperplane, each decoder requires 31 symbols of data as input. 7 symbols out of these correspond to the edges which are also incident on the points of the corresponding plane. Then, based on 8 other disjoint planes, the same decoder can fetch 3 symbols **each** using edges that reach to each of certain points on these 8 planes. Thus, each decoder can fetch

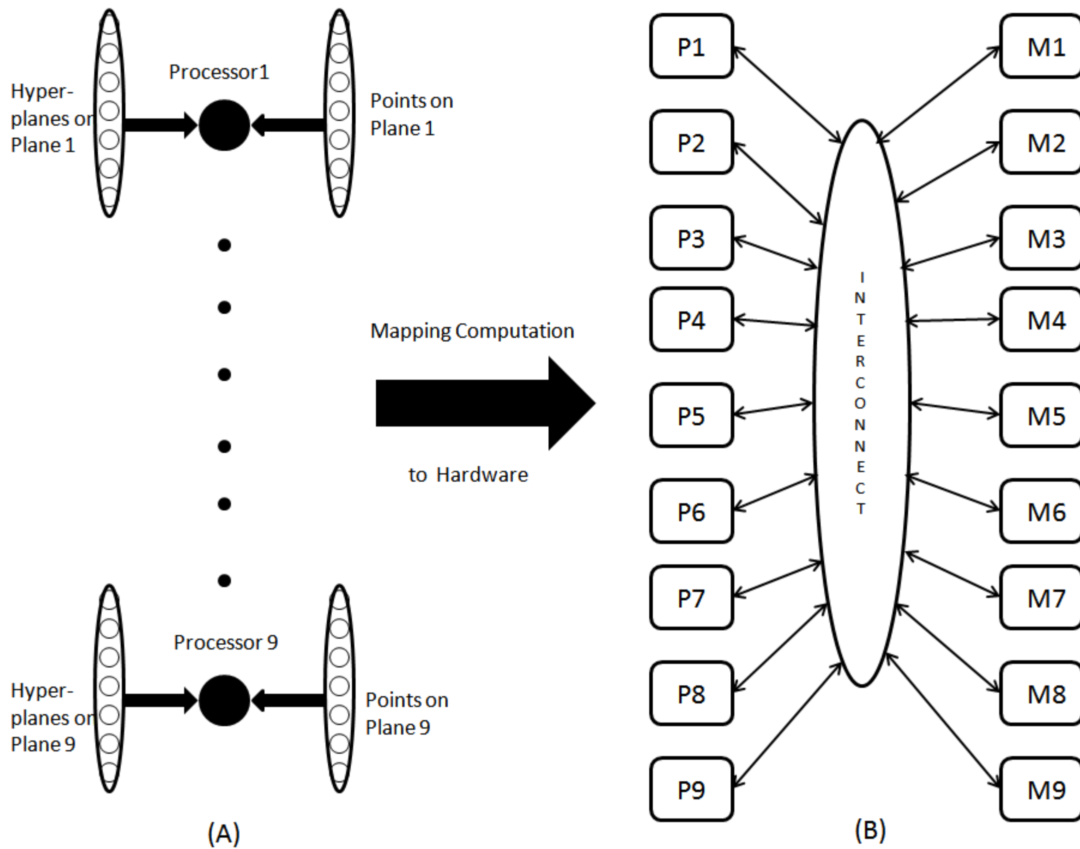

Figure 7.3: (A) Distribution of computation to processing units (B) High-level System architecture

all its 3×8+7 = 31 symbols directly.

A high level view of the mapping of computation to hardware is shown in figure 7.3. In this figure, each P_i is the **processing unit** corresponding to the Plane P_i. Further, each M_i is the **memory block** associated with the Processing unit P_i, but not contained in P_i. The interconnect, though represented as a black box, can be thought of as a big multiplexer, which controls the connections between the read/write ports of the memories, and the input/output ports of the processing units. Depending on the *Phase* of computation and the number of input symbols that have been received by the decoder so far, the interconnection is effected by establishing connections between processing units and the memories. This point becomes clearer when we explain the details of the design in the upcoming sections.

7.6 Memory Block Design

As mentioned earlier, each processing unit is co-located with an associated memory block (RAM) of size 217 bytes. The processing unit hosts an RS decoder. This decoder has an input stage, a computing stage and an output stage. Input for the decoder is read from the memory blocks, and the corresponding output must be written back to memory blocks. We use dual-port memory blocks, with one port used as a read port and the other as a write port. Usage of dual port memory blocks allows us to overlap the input and write-back stages. Hence it gives the system an advantage of increasing throughput. Our scheduling takes care that simultaneous read and write does not occur on the same memory location.

For the FPGA implementation, the memory blocks are implemented with *asynchronous read*, so that they get synthesized into distributed memories. This is because we require small-sized (217 bytes) memory blocks. Also, distributed memories, implemented using Look Up Tables (LUTs), are faster than the other on-FPGA memory elements, the block RAMs(BRAM).

The storage of data in these memory blocks is such that design of the address generators is greatly simplified. In Phase 1, we store the data such that the address generator of each processing unit simply runs a counter from 0 to 217. As seen in previous section, each decoder requires first 7 bytes corresponding to the edges with points associated with its own plane P_i in first cycle, 3 bytes next from edges to points associated with P_{i+1}, and so on. Due to cyclic shift automorphism in the projective space lattices, whenever the index of a plane, $i + j : 1 \leq j \leq 8$ becomes > 9, we convert it to $((i + j)\ modulo\ 9)$.

Thus, the data distribution is such that for the first 7 clock cycles, each decoder gets data from its own memory block. In the 8th, 9th and 10th cycle, its address and data ports are switched over to new connections, such that each decoder receives data from $\{(i + 1)\ modulo\ 9\}^{th}$ memory block. In the next 3 clock cycles, the connections are *again changed*, so that each decoder receives data from $\{(i+2)\ modulo\ 9\}^{th}$ memory block, and so on. Hence the internal layout of each memory block can be seen as follows.

Each memory block i contains the first 7 symbols required by decoder i in address locations 0 to 6. Addresses 7, 8 and 9 contain the data required by decoder $\{(i - 1)\ modulo\ 9\}$ in its 7th, 8th and 9th clock cycles. The next 3 locations store the

data required by processing unit $\{(i-2) \ modulo \ 9\}$, and so on. After address 30(data item 31), the cycle starts again with the next 7 bytes required by decoder i, now mapped to a different vertex of next fold, the next 3 by decoder $\{(i-1) \ modulo \ 9\}$ and so on.

The order of edge numbers stored in each memory block is given in table 2 in the appendix.

The obvious advantage of storing data this way is that the address generation module is just a counter in *phase 1*. It is clear that changing the order of the input symbols to the decoders has simplified data distribution and address generation in Phase 1. This is convenient because the corresponding change in the encoding process is just the re-arrangement of columns of the generator matrix. So, rather than keeping track of the original order of symbols and possibly complicating the address generator, it is simpler to modify the order according to the partition.

Another important advantage of such a data distribution is that in *phase 2*, each decoder needs to fetch data from **only** its corresponding memory block. This can be seen easily by noting that each memory block stores all the data symbols required for the points associated with the processing unit. The memory blocks layout was planned for phase 1 computation actually. Since in Phase 1, a memory block is read from, only when data is required corresponding to an edge incident on a point associated with the corresponding processing unit, it ensures per-point complete storage property.

However, in phase 2, the order in which data is required by the decoder is different. A simple counter for address generation will not work. In fact, to deal with this, we store the required order of addresses in 2 ROMs along with each decoder. Both ROMs contain the same sequence of data. One ROM is used for input address sequence, while the second is used for the output order sequence, thus providing ability to overlap the input and output phases. In Phase 2, at every clock cycle, an address is read from the ROM, and is placed on the address port of the memory block(RAM). The output of the RAM then becomes available to the decoder.

We make sure that no *data hazards* due to simultaneous reading and writing will occur. In any phase, decoders are working either on points, or on hyperplanes, at a time, and the data required for a point(hyperplane) is not required by any other point(hyperplane). In between phases, we wait for the write-back (output) to finish before starting the input stage of the next phase. Hence there are no data hazards.

7.7 RS Decoder Design

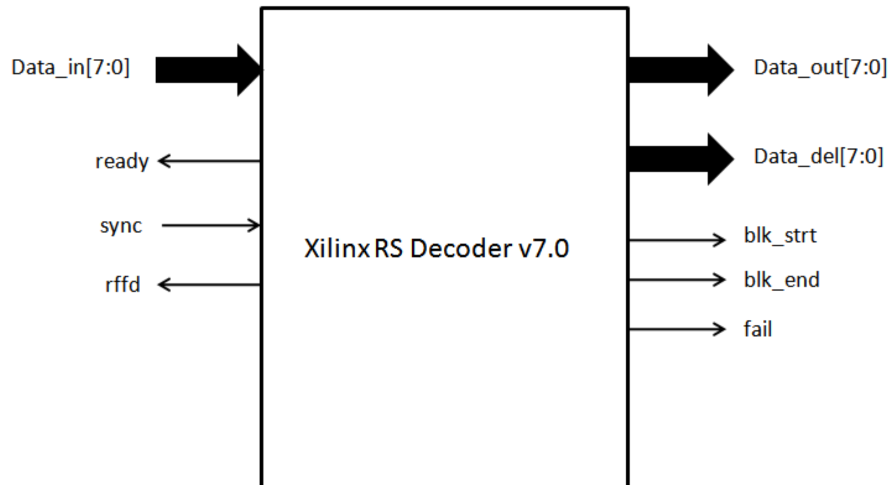

Figure 7.4: The interface of the Xilinx RS decoder IP

For rapid prototyping, we make use of IP core in form of Xilinx RS decoder v7.0. A schematic of the decoder is given in figure 7.4. The interface for input phase consists of a ready for first data (*rffd*) output from the decoder, a synchronizing (*sync*) input to the decoder and the *data_in* port. We can select the symbol width in terms of clock cycles for which a symbol is steady at *data_in*. In our design, we have chosen it to be 1 clock per symbol. Depending on the symbol width and the minimum distance of the code, we can determine the processing delay (in number of symbol periods). There is also a *ready* output, which is used to indicate when the decoder is ready to receive input symbols. This output is useful only if the processing delay (delay for processing the inputs) is greater than the number of clock cycles required to input all the input symbols. It is futher useful when block length for the RS codes being used is *variable*, since in such cases, one needs to know when to stop giving new data at the input port of the decoder, so that the input symbols are not lost. It is not required in our implementation, when the RS codes minimum distance is $\epsilon = 5$. Here, the processing delay is 30 clock cycles, which is less than the number of clock cycles required to sample the input (31 in our case). The *ready* signal always remains high in our case and only the *rffd* is required for handshaking.

Also, once the first symbol of a code is sampled by the decoder, then a **fixed** latency later, the decoded output starts appearing at the output, one symbol per clock edge. The output interface has control signals to indicate the start and end of the output word (*blk_strt* and *blk_end* respectively). It also has an output *fail* control signal, which is

asserted if the decoding fails. There is also an option for having the input passed on to an output (*data_del*) after the latency of the decoder. The blk_strt is pulled high for one clock cycle by the decoder, when the first output symbol is available on the output port. The blk_end signal is pulled high for one clock cycle, when the last output symbol is available at the output port. Along with blk_end, the fail signal is pulled high or low depending on whether the decoding was successful or not. If the fail value is asserted, then the contents of data_out are not to be considered valid. In that case, we must take the output from data_del as per requirements of our decoding algorithm.

The IP core, which is a (255,251,5) RS decoder, has been tuned/parametrized for the minimum distance of the code(ϵ) = 5 and a shortened RS code of length 31, for which the processing delay is 30 clock cycles, while the latency is 68 clock cycles. The processing delay is the delay before which the next block of input symbols can be given to the decoder. The latency is the delay between the first input symbol and the first output symbol.

Whenever we sample *rffd* to be high, we pull *sync* signal high, and place the first symbol of the corrupt codeword at the input. The decoder starts sampling the input from the clock edge at which it samples sync to be high. At the same clock edge *rffd* is pulled low by the decoder. It is responsibility of the bigger system to place the correct data symbol sequence at the subsequent clock edges, on the inputs of RS decoder subsystem.

This decoder requires a initialization period of around 100 ns when powered up. In this period the *rffd* is high irrespective of sync value. Thus, the *Reset* state of the system must last for at least 100 ns. More specific details can be found in the data sheet of the decoder provided by Xilinx[9].

So, in conclusion, our *Processing Unit* consists of the RS Decoder IP and the two ROMs along with address generation logic. Each memory block is a distributed RAM. The ports of the memory blocks and processing units are interconnected such that each processing unit can read from, and write to, every other memory block. However, at a time, only one processing unit is reading and only one processing unit is writing to it in our design. The data path of each processing unit is outlined in figure 7.5.

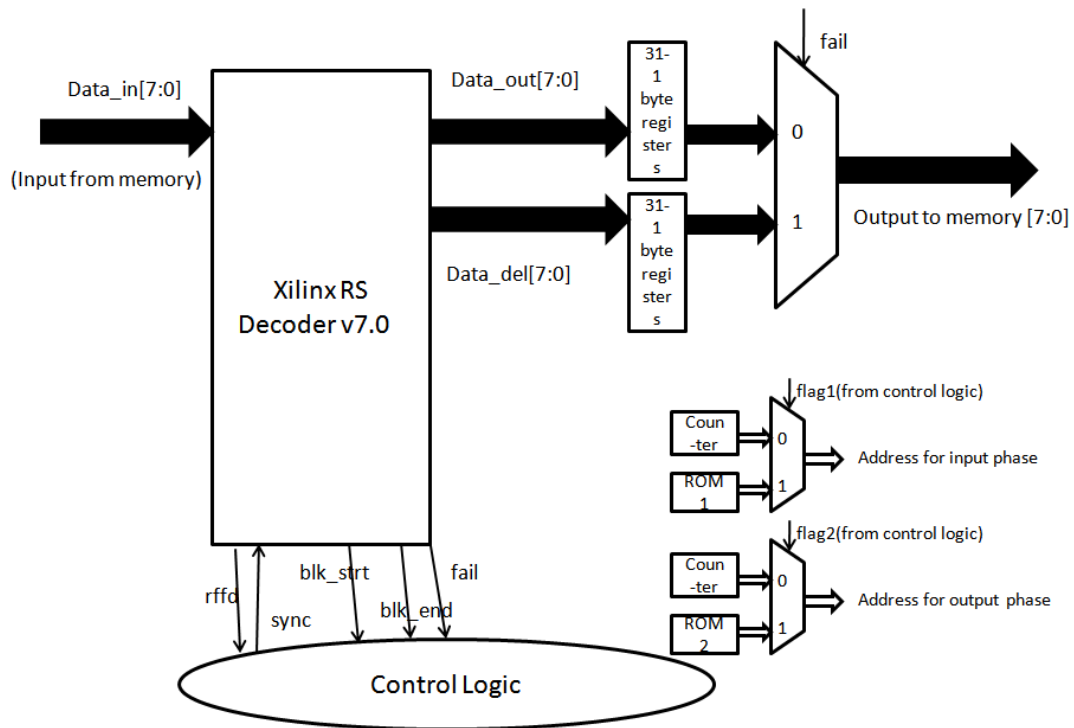

Figure 7.5: Data Path outline for each Processing Unit

7.8 Control Path Design

Three simple finite state machines have been used to implement the system. First FSM handles the input phases. The second and third FSMs handle the output and write-back phases of the decoding. Since each processing unit works perfectly in sync with the others, i.e. at any stage, all the processing units are at exactly the same stage of computing, we need to instantiate the FSMs only once and change the control signals for all the processing units simultaneously.

7.8.1 1st State Machine

This state machine handles the inputs for each RS decoder. The state variables involved in transitions are stored in various registers. The roles of these state variables(registers) are as follows.

hp_addr is the counter for the address value in Phase 1.

count_i keeps track of how many symbols have been fed to the decoder.

sync is a 9 bit register which is a handshake signal for the RS decoders.

flag1 indicates the current phase, 0 implies Phase 1; 1 implies Phase 2.

sync_flag indicates a synchronization between the input and output cycles. Phase 2 should not be started before output of Phase 1 has been written back. Similarly, in the next iteration, Phase 1 should not be started before output of Phase 2 is written back to the memory blocks.

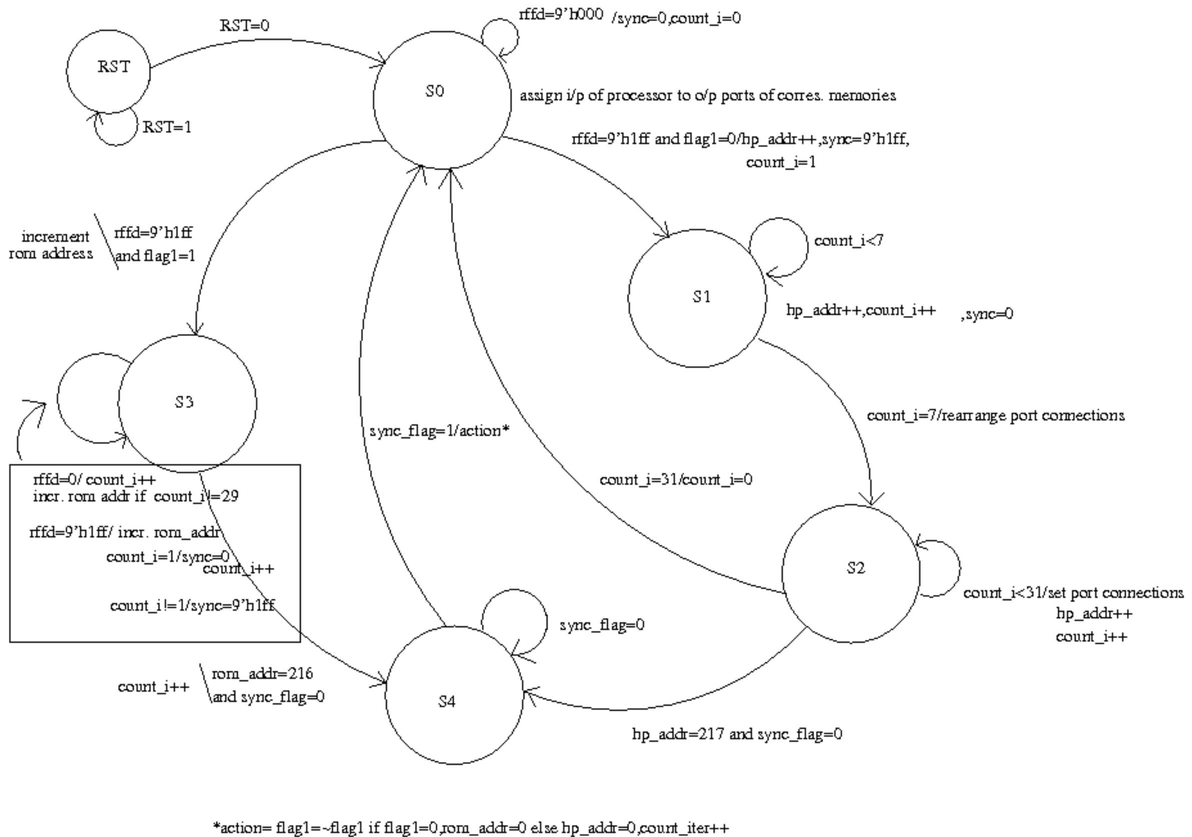

Figure 7.6: Input Handling State Machine

7.8.2 2nd State Machine

This state machine takes the decoded output, and the delayed input from each of the 9 decoders, and stores them into 2 different registers. Depending on the value of the fail signal at the end of a block, one of the registers is selected to be written back into the memory blocks. **flag3** is asserted when a block output is complete indicating to the 3rd state machine that it can begin the writeback process.

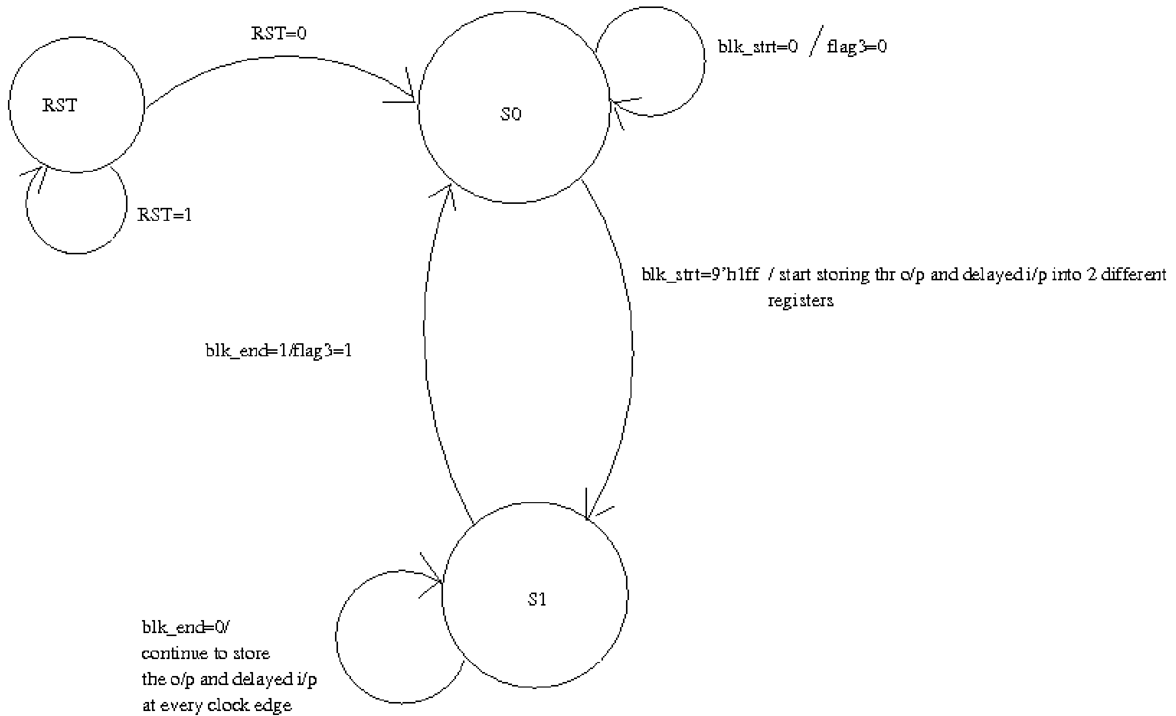

Figure 7.7: Output Handling State Machine

7.8.3 3rd State Machine

This state machine is responsible for writing back data into the memory blocks. **flag2** specifies which Phase output is being written back. If it is '0', 1st Phase output is being written back. In this case, we need to write to the memory blocks in the same order that we read in Phase 1. If **flag2** is '1', then the output is from Phase 2. In this case, the order of addresses for storing back is fetched from the 2nd ROM. In either phase, the value of the **fail** flag determines whether the decoded output or the delayed input is to be written back. The counter **count_constr** keeps track of how many constraints have been written back.

7.9 Multiplexers used in Control Path

Depending on the Phase of computation, either **hp_addr_i** (Phase 1), or **rom_output_i** (Phase 2) needed to be switched onto the address input of the read port of the 9 memory blocks. Therefore there are 9 multiplexers (with select signal as **flag1**). These muxes decide which of these two are selected to be the address inputs.

Similarly, in the output phase, we need to decide between **hp_addr_o** and **rom_output_o**

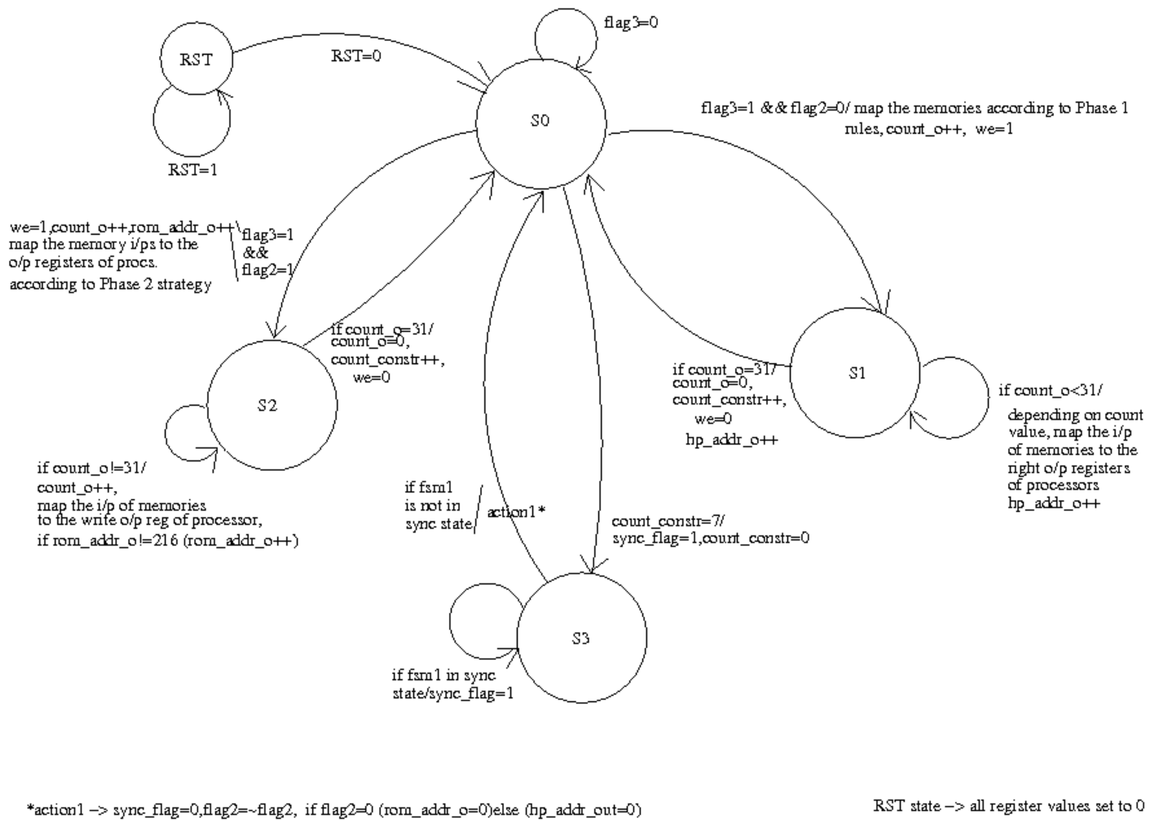

Figure 7.8: Writeback Handling State Machine

for the address input to the write port of the memory block. Here, again, we need 9 multiplexers (with select signal as **flag2**).

Further, when the decoders start the output phase, we store the decoded output and the delayed input. Depending on the value of fail, we need to select one of these for writing back. So 9 more multiplexers are required for enabling this selection.

Finally, in Phase 1 input as well as write back, we need to decide, on the basis of count value, the port connections between the processing units and the memory blocks. The logic for this has been written in a behavioral manner. It is implemented as a chain of multiplexers, which incidentally also forms the critical path of the design.

7.10 Schedule for Decoder Iteration

Even though the 3 FSMs have to be executed for each processing unit, as mentioned before, each processing unit works perfectly in sync with the others. Hence we can have only one instance of each FSM, and just replicate and distribute their output signals/transitions to **all** processing units in each cycle. The input and output FSMs are largely independent,

except during the end of Phase 1 or Phase2. Here, we must wait (for around 100 clock cycles) for the write back of the current phase to finish, before the inputs for the next phase can be read. This synchronization is achieved with the *sync_flag*. Also, the two output FSMs are synchronized in the sense that the write-back can be executed only after the outputs from the decoder have been received, and the *fail* signal has been checked to indicate the validity of the output. This synchronization is achieved via *flag3*, and can be easily seen from the FSM diagrams.

The implementation of the schedule is thus completely synchronous, with counters keeping track of how many input/output symbols have been sent/received.

7.11 Detailed Decoding Process

In this section we present a brief walkthrough of one decoding iteration, based on decoder's implementation on FPGA.

As mentioned before, the *initial* reset signal must be held high for at least 100 ns. Once the reset goes low, and the *(*rffd) output of every RS decoder goes high, decoding computations for phase 1 are started off.

At the start, the address counter for Phase 1 is set to 0. Initially, the output of each memory block is mapped to input of the *corresponding* processing unit, i.e. read port output of M_i is given to P_i. The *sync* input for each RS decoder is also set high. From the next clock edge, each decoder begins to sample the data at its input. Hence the address counter needs to start incrementing every clock edge so that the correct data symbols are available to the decoder. Note that we had stored the data in the memory so that the address generator would be just a counter. After 7 clock cycles, the connection mapping of processing units to memory blocks needs to change, as explained in section 7.6. For the next 3 clock cycles, processing unit P_i is mapped to memory block M_j, where $j = (i + 1)$ *modulo* 9. The mapping is again changed after 3 clock cycles, whereafter the processing unit P_i is connected to memory block M_j, where $j = (i + 2)$ *modulo* 9. Throughout all these connection switchings, the address output from each processing unit is also being incremented by 1. This scheme of the interconnection changing every 3 clocks is continued till each processing unit has received the complete input data required for the decoding of the first hyperplane associated with it.

In our case, the processing delay for the input is 30 clock cycles, which is less than the number of input symbols for one sub-code decoding. Thus, it is possible to begin the input phase of the next hyperplane as soon as the input of the first hyperplane has been given. But, to make the design robust and to ensure complete synchronization between processing units, we wait till the *rffd* output of each decoder becomes high before asserting the *sync* of each decoder again, and starting the input phase of the next hyperplane. Once the input phase is started, the process repeats with the address being incremented every clock cycle, and the connections between processing units and memory blocks change similarly, according to which input symbol needs to be received at the different processing units.

After 68 clock cycles of latency since the first input symbol was sampled by each decoder, the output and the delayed input start appearing at their respective ports. Both these output need to be stored in separate registers until the 31st output symbol (i.e. *blk_end* signal is asserted). With the *blk_end* signal asserted, the *fail* signal indicates if the decoding was a failure or not. If the decoding fails, we need to write-back the delayed input. If the decoding was successful, we need to write back the valid output of the decoder. The write-back schedule is exactly the same as the input schedule, with the address being a counter and the connections between the memory block write ports and the processing units exactly mimicking the steps during the input schedule.

The above mentioned schedule repeats 7 times for each processing unit. Thus, each processing unit decodes the symbols corresponding to the 7 hyperplanes associated with it. The write-back for a particular hyperplane finishes $(68 + 31 = 99)$ clock cycles after the first input symbol has been sampled by the decoder. Thus, after the input schedule for Phase 1 is complete, we must wait for 99 clock cycles before initiating the input schedule for Phase 2. This is done to ensure that no *write-after-read* data hazards occur.

In Phase 2, the processing units work on decoding the symbols associated with the points. The interconnections between the read and write ports of the memory blocks to the input and output ports of the processing unit is greatly simplified, with every processing unit P_i only communicating with its own memory block M_i. The address output of each processing unit that is fed to the two address ports of the memory blocks is got from the two ROM outputs. One ROM for the input schedule, while the other ROM for the output schedule. The address input for the ROMs is a simple counter which is set to zero initially. Once the *rffd* outputs of all decoders are sampled to be high, the *sync* signal

is asserted for all the decoders and the first input symbol is ready to be read from the memory block read ports. Once 31 symbols have been read, we check if the *rffd* signals of all the decoders is high. Only then RS decoders are sequentially provided with the inputs for the next "point". Here also, the output symbols and the delayed input symbols start appearing at the respective ports, 68 clock cycles after the first input symbol is sampled by the decoder. These must be stored in registers, as before, until we get the *blk_end* signal. Once the *blk_end* signal is asserted by the decoder, we must begin the write back of the output or the delayed input depending on the value of the *fail* signal.

The write back is performed similar to the input schedule, the address for the memory block write ports is got from the ROMs and the ROM address itself is just a counter. This schedule is repeated till each processing units finish the decoding associated with all the 7 points assigned to them. The completion of Phase 2 results in the completion of one iteration of the overall decoding. The entire Phase 1 and Phase 2 cycles are repeated 4 times, before the result of the decoding algorithm can be read from the memory blocks.

7.12 A modification to include Erasures

So far, we have not discussed the effects of adding the ability of erasure corrections to our decoder. This partly due to the lack of easy erasure decoding blocks in MATLAB. However, the Xilinx RS decoder IP does have the option of including erasure correction facility. You can choose to give an additional one bit input to the decoder core to specify if the input symbol being sampled at the current clock edge is an erasure or not.

In the present design, if erasure correction is enabled, we just store an extra bit with every data symbol to indicate if it is an erasure or not. If the decoding at a decoder core fails(indicated by the *fail* signal), we mark all the edges going to that vertex as erasures.

Enabling erasure correction increases the latency and processing delay of the core, thereby decreasing the throughput. Also, it necessitates a slight change in the input handling FSM to ensure the correct working of the design. Since we had designed for $\epsilon = 5$, without erasures, the processing delay was 30 clock cycles and we could afford to give continuous input to the core. Including erasures and/or increasing ϵ, we get processing delay to be greater than the number of input symbols. Now, we must wait till the processing delay of one input block is over before the next input block can be

ϵ	Latency	Processing Delay	Random errors	Burst errors	Clocks for 4 iterations
5	83	45	141	143	3428
7	115	77	218	219	5218
9	155	117	328	295	7459

Table 7.5: Erasure correction results

given to the decoder. This does not change anything in *Phase 1* of the input because our address generator is just a counter and hence there is only one clock cycle delay between us asserting the address and the data being available for input to the core. The input handling FSM does that based on the *rffd* signal and hence does not depend on the processing delay. However, in *Phase 2*, we are using a look-up for the address as well and hence there is a delay of 2 clock cycles between the assertion of the look-up table address and the availability of data at the output. To ensure a continuous stream of input symbols after the first data has been sampled by the decoder, we need to invest an extra clock cycle in which we just sample the *rffd* after it goes high and get the pipeline started. This has been incorporated by using an additional 'flag5' signal to delay the assertion of the "sync" signal by one clock cycle. Investing 7 extra clock cycles per iteration, we have made the design generic in the sense that it will work for all values of processing delay and hence all valid values of ϵ.

The table 7.5 gives approximate error correction results with respect to the Model-SIM simulations that have been run in order to gauge the error correction capabilities for different ϵ values, with erasures. The clock period was 20 ns.

A further improvement of the design has been done with respect to the storage required for address look-up in Phase 2. It is not possible to infer a Dual Port ROM in the version of Xilinx that we are using and hence the best way to save storage space is to use a dual port distributed RAM within each processing unit rather that 2 ROMs described above.

7.13 Performance Modeling and Analysis

In this section we first present a calculation of the peak throughput of the design, give order estimates for number of computations and storage involved and finally present some results of the current implementation of the design.

Throughput

Let us assume that the design has been implemented such that the operating frequency is f Hz. Let the underlying bipartite graph have N edges. Let the dimension of the projective geometry used be m. Thus, the number of points is $2^{m+1} - 1 = n$, which is also equal to the number of hyperplanes. Each vertex in the bipartite graph formed out of the incidence relations of points and hyperplanes has a degree of $2^m - 1 = \delta$.

Thus, $N = (2^{m+1} - 1) * (2^m - 1) = n * \delta$. Let there be x number of processing units working in parallel. Further, let each processing unit work on y pointsm, and hence on y hyperplanes.

Let the processing delay(time for processing the inputs) of the RS decoder be p. Let the latency of the RS decoder be l symbol periods. Let the number of clock cycles per symbol period be c. Once the first input symbol is sampled, then l*c clock cycles later, the first output symbol appears. The size of each block of RS code is δ. So, it takes $\delta * c$ clock cycles to collect the output, and further δ clock cycles to write it back to the memories. For y points, since we can give input symbols for the next block *immediately* after the processing delay whenever $\delta \leq p$, $y * p * c$ clock cycles are required to provide all the inputs per phase to a particular RS decoder. Consider the last RS decoding corresponding to last point that is scheduled on a particular RS decoder. After the first input symbol of this point is sampled, l*c clock cycles later, the output symbols start appearing, where they are registered and then either the delayed input or the corrected output is written back to the memory, depending on the fail signal. Hence, the number of clock cycles C required per *Phase* of computation is as given below:

$$C = (y-1) * p * c + (l * c) + \delta * c + \delta$$

For both phases, the number of clock cycles required is *2C*. So, for one iteration *2C* clock cycles are required. Since we perform 4 iterations, *8C* clock cycles are required for

one decoding round.

Since there are x processing units, there are x two port memory units. The size of each memory unit is $y * \delta$. In a pipelined implementation, the decoded output of the present codeword needs to be read out, and the input symbols of the new codeword needs to written to the memories. Since the memory units are two port and we can implement them in the *read first* mode, totally $y * \delta$ clock cycles are required for reading the present memories and writing in the symbols of the next codeword.

Thus, over k blocks of streaming encoded data, the throughput can be given as follows.

$$\begin{aligned} \text{Throughput} &= \frac{k * N}{k * (8C + y * \delta)} f \quad \text{bytes/sec} \\ &= \frac{N}{(8C + y * \delta)} f \quad \text{bytes/sec} \end{aligned}$$

Power

The maximum power is used up during the RS decoding steps. Within RS decoding, we can estimate power by giving the order of field computations involved for RS decoding algorithm. As the size of the Galois field increases, the complexity required for addition, multiplication etc. of two field elements increases. Also, as the number of parity symbols for the code increases, the number of computations also increase and become more complex. Given below is the order of additions/subtractions, multiplications and comparisons involved if it is a t error correcting code.

$$\begin{aligned} \text{Number of Additions/Subtractions} &\propto t^2 \\ \text{Number of Multiplications} &\propto t^2 \\ \text{Number of Comparisons and Updates} &\propto t \end{aligned}$$

Note that by using the Singleton bound for RS codes, we have t $= \frac{(1-R)}{2} \cdot n$. Hence we can coarsely represent the computational complexities as follows.

$$\begin{aligned} \text{Number of Additions/Subtractions} &= O(n^2) \\ \text{Number of Multiplications} &= O(n^2) \\ \text{Number of Comparisons and Updates} &= O(n) \end{aligned}$$

Storage

It is clear from the discussions so far that the amount of memory required varies *linearly* with the size of the code.

7.13.1 Results of Implementation on the board

The design was implemented on a Xilinx virtex 5 LX110T FPGA with a speed grade of -3 (maximum available for this device). About 25% of the slices were used to implement the folded computations which corresponds to 4,367 out of 17,280. We used distributed RAM to implement the memory modules. The results are tabulated in 7.6 for both with and without erasure designs. The post PAR frequency is 180.83 MHz for the design without erasures and 180.79 MHz for the design with erasures.

The above numbers are for $\epsilon = 5$ for which it takes 2611 clock cycles to finish 4 iterations, without erasure decoding. Adding 217 clock cycles to write data into the memory, we get a throughput of $\frac{1953}{2828} * 181 \approx 125 Mbytes/s$. For a 72x CD-ROM read system, the data transfer rate is $10.8 Mbytes/s$. Hence, the decoder can easily be incorporated without hurting throughput. Moreover, in an ASIC implementation, we would expect the performance to be better.

To test the design on the board, we simply output the memory contents to the serial port provided on the board after 4 iterations of decoding. This data is transmitted to the PC and can be read via the Realterm program.

		Without Erasure		With Erasures	
Resource	Available	Usage	Percentage	Usage	Percentage
Occupied Slices	17280	4367	25	5889	34
Total Memory(KB)	5328	324	6	324	6

Table 7.6: Resource Utilization

Chapter 8

Folding of PG based point-hyperplane graphs

8.1 Introduction

As the dimension of the projective space is increased, the corresponding graphs derived from the incidence relations of the projective subspaces grows in size (in terms of number of vertices and degree of each vertex). If each vertex of the graph represents a processor then the number of processors grows exponentially with the dimension. For practical implementations it is not possible to have a large number of processors running in parallel.

Folding the computation efficiently allows practical implementation with the advantage that all the processors are utilized continuously throughout the computation cycle. The folding allows us to create a schedule which ensures that there are no memory access conflicts and no shifting of data between memories is required.

We begin by giving a brief introduction to the cardinalities associated with projective geometry. We then proceed to demonstrate our folding schemes using $\mathbb{PG}(5, GF(2))$ as an example. It is then very easy to extend the scheme to general projective geometry dimensions and we can also determine the conditions required to extend the scheme to projective spaces over other Galois fields.

8.2 Description of the Computations

The computations we consider are based on bipartite graphs that are created using point-hyperplane incidence relations of $\mathbb{PG}(m, GF(p))$. The points and hyperplanes are represented by the vertices of the graph and there is an edge between two vertices if one of them represents a point and the other represents a hyperplane containing the point.

Thus, the vertices corresponding to the points form one partition of the graph and the vertices corresponding to the hyperplanes corresponds to the other partition. We envisage two possible mappings of computations based on such graphs.

- *Type 1*: The vertices corresponding to processors and the edges correspond to the data symbols/memories. Each processor performs the same computation but on different sets of data i.e. the data symbols or the data from the memories that correspond to the edges incident on it. This sort of mapping is useful for iterative decoding applications.

- *Type 2*: The points correspond to processing units and the hyperplanes correspond to the memories that store data required by the points that are contained in it. A dual and equivalent version of this exists with the roles of points and hyperplanes interchanged. In this case, there could be computation scheduled that required a hyperplane to work on binary operands, i.e. it accesses two points at a time, performs some computation and writes back the result. A perfect access pattern mentioned in [11] could be generated.

Both of the above types of computations can be easily folded so that fewer processors are required at the cost of throughput. The folding schemes and schedules we talk about are perfect in the sense that if the graph is folded by a factor of 'x', then the amount of time required for the computation increases exactly by a factor 'x'(Assuming same number of memories in both cases). No processor is left idle at any point during the computation. This is under the assumption that once the processors acquire the data required for a particular computation cycle, each processing unit takes exactly the same amount of time to finish the computation. Moreover, the special folding scheme ensures that the data doesn't need to be redistributed between memories in between computations thus avoiding an overhead of time. The schedule becomes structured and can be used to simplify the address generation circuits within processing units. We will use dual port

memories where one port is used for reading and one port for writing. This helps to overlap the input and output phases between folds and thus save time.

8.3 Folding computations for $\mathbb{PG}(5, GF(2))$ - Propositions

In this section we outline two schemes to demonstrate the possibilities of folding computations related to point hyperplane incidence graphs derived from $\mathbb{PG}(5, GF(2))$.

Proposition 3. *When considering computations based on the point-hyperplane incidence graph of* $\mathbb{PG}(5, GF(2))$*, it is possible to fold the computations and arrange a scheduling that can be executed using 9 processing units and 9 dual port memories.*

Proposition 4. *When considering computations based on the point-hyperplane incidence graph of* $\mathbb{PG}(5, GF(2))$*, it is possible to fold the computations and arrange a scheduling that can be executed using 21 processing units and 21 dual port memories.*

8.3.1 Proofs

We first present some important numbers associated with $\mathbb{PG}(5, GF(2))$ that we will use in proving the functional correctness of the folded computations.

Projective geometry of dimension 5 over $GF(2)$

- No. of points = No. of hyperplanes (4-d projective subspace) = $\phi(5, 0, 2) = 63$.
- No. of points contained in a particular hyperplane = $\phi(4, 0, 2) = 31$
- No. of points contained in a line(1-d projective subspace) = $\phi(1, 0, 2) = 3$
- No. of points contained in a plane(2-d projective subspace) = $\phi(2, 0, 2) = 7$
- No. of points contained in a 3-d projective subspace = $\phi(3, 0, 2) = 15$
- No. of hyperplanes containing a particular plane = $\phi(5 - 2 - 1, 4 - 2 - 1, 2) = 7$
- No. of hyperplanes containing a particular line = $\phi(5 - 1 - 1, 4 - 1 - 1, 2) = 15$

- No. of hyperplanes containing a particular 3-d projective subspace = $\phi(5-3-1, 4-3-1, 2) = 3$
- No. of lines contained in a 3-d projective subspace = $\phi(3, 1, 2) = 21$

We will need the following lemmas to establish the feasibility of folding using proposition 1.

Lemma 14. *The point set of a projective space of dimension 5 over $GF(2)$ (represented by the non-zero elements of a vector space V over $GF(2)$) can be partitioned into disjoint sets such that each set contains the non-zero elements of a 3-d vector subspace of V and thus represents a unique plane (2-d projective subspace). This leads to the set of hyperplanes (4-d projective subspaces) also being partitioned into disjoint sets.*

Proof. The vector space V is represented by the field $GF(2^6)$ and has an order of 63. Since 3 is a divisor of 6, $GF(2^3)$ is a subfield of $GF(2^6)$. The multiplicative cyclic group of $GF(2^3)$ (of order 7) is isomorphic to a subgroup of the multiplicative cyclic group of $GF(2^6)$ and we can form a coset decomposition to generate 9 *disjoint* partitions of V into subsets such that each subset is a 3-d vector space (-$\{0\}$) and hence represents a 2-d projective space i.e. a plane.

Lets say that α is a generator for the multiplicative group of $GF(2^6)$. Then $(1, \alpha^9, \alpha^{18}, \alpha^{27}, \alpha^{36}, \alpha^{45}, \alpha^{54})$ is the sub-group that we are looking for. The distinct cosets corresponding to this sub-group give the partition we need.

Each plane is contained in 7 hyperplanes and these hyperplanes are unique to the plane since they represent the 7 hyperplanes that are common to the set of points that form the plane. More explicitly, if two planes do not have any point in common, they will not have any hyperplanes in common and vice-versa. Thus, the 9 disjoint planes partition the hyperplane set into 9 disjoint subsets. □

Also, we have proved earlier in lemma 5 that a point that is not contained in a plane lies in 3 hyperplanes associated with that plane.

Thus, in our case, a point on a plane $P1$ gets a degree of 7 through $P1$, and a degree of 3 from each of the remaining 8 planes. (Total degree is 8*3 + 7 =31) This is true for all points with respect to the planes that they lie in and all hyperplanes with respect to the plane they contain.

Main Proof of Proposition 3

Here, we present the folding scheme and the schedule for proposition 1; thereby proving its existence.

We have shown above in lemma 14 that we can partition the set of points into 9 disjoint subsets (each corresponding to a plane) and there will be a corresponding partition of the hyperplane set. Let the planes be $P1, P2 \ldots, P9$. We will assign a processor to each of the planes. Let us abuse notation and call the processors by the name corresponding to the plane assigned to them. We will assign a dual port memory to each of the processing units and provide a schedule that **avoids memory conflicts**. The distribution of data among the memories can be done such that the address generation circuits become simplified (counters/look-up tables). The address generating units are incorporated with the processors to form *processing units.*

Type 1 Computation

Here, the processors are represented by the points as well as the hyperplanes. The data symbols are represented by the edges of the bipartite graph. The overall computation is broken into two phases. *Phase 1* corresponds to the points performing the computation over the edges and updating the necessary data symbols. *Phase 2* corresponds to the hyperplanes performing the computations and updating the necessary symbols.

The schedule and data distribution among the memories (say $M1, M2 \ldots, M9$) is done as follows:

- In *Phase 1*, processing unit Pi performs the computations corresponding to the points that are contained in the plane Pi in a sequential fashion.The data corresponding to edges incident on the hyperplane-vertices containing plane Pi is stored in memory Mi. Thus for each hyperplane h containing plane Pi, there will be 7 units of data corresponding to the points associated with plane Pi that will be stored in Mi. In addition to this, Mi will have to store 3 units of data for each of the points **not** a part of Pi corresponding to the 3 hyperplanes that are reachable from those points and contain plane Pi.

  Suppose processing unit $P1$ is beginning the computation cycle corresponding to point a. It needs to fetch data from the memories, perform some computation and

possibly write back the output of the computation to the same memories. First, it collects the 7 units of data corresponding to a in $M1$. This corresponds to the edges that exist between point a and the hyperplanes that contain plane $P1$. Next, it fetches 3 units of data from each of the other Mis that correspond to a. This consists of the edges between a and the 3 hyperplanes from each of the planes not containing a. Since none of the points share any data (have no common edges in the graph), we can have all the 9 processors working in parallel and each of them follows the same schedule. For processing unit Pi, first, 7 units of data from Mi are fetched. Further, 3 units of data are fetched from each $M(i + j)mod9$, j going from 1 to 8. In this fashion, no two processing units will be trying to access the same memory at the same time i.e. no memory conflicts will occur. The writing if the output is done in a similar fashion. If dual port memories are used, we can overlap the writing of the output of one point with the reading of the input of the next point.

- In *Phase 2*, processing unit Pi performs the computation corresponding to the hyperplanes that are associated with plane Pi. If the data is distributed as explained in the previous point, then Mi already contains all the data required for the hyperplanes associated with plane Pi. In this case the processing unit communicates only with its own memory and performs the computation.

The data can be distributed such that the address generator circuit in Phase 1 is just a counter and in Phase 2 it becomes a look up table. This sort of folding is specially useful if the order in which the data is received by the processing units has an effect on the computation involved.

Type 2 Computation

In this case, the hyperplanes represent the processors and the points represent memory units/data to be operated on. There is an equivalent mapping with the roles of the points and hyperplanes reversed.

Here, we place the data corresponding to the points associated with a plane Pi in memory Mi. There is only one phase of computation wherein the processing unit performs the functions of each of the hyperplanes associated with in a sequential fashion.

The schedule of data retrieval has to be developed depending on the kind of computation involved. Suppose, as mentioned earlier, binary operands are required. Each hyperplane takes data from two of its points at a time and goes through all possible combinations in one phase. Then an exhaustive schedule can be thought of for one processing unit and the remaining processing units follow the same schedule but talk to different memories. Specifically, if $P0$ is talking to Mi, Mj then $P1$ will talk to $M(i+1mod9)M(j+1mod9)$ and so on in order to ensure no memory conflicts. The assumption here is that all hyperplanes of the projective space are allowed to work in parallel. So, if two hyperplanes are working with data associated with a particular point and they make relevant updates to that data, it should not make a difference to the overall computation. Alternatively, the computation could be such that the data from the points is used for computation, but the results of the computation are accumulated/updated in a completely different section of the memory.

Lemmas related to Proposition 4

Lemma 15. *The point set of a projective space of dimension 5 over $GF(2)$ (represented by the non-zero elements of a vector space V over $GF(2)$) can be partitioned into disjoint sets such that each set contains the non-zero elements of a 2-d vector subspace of V and thus represents a unique line (1-d projective subspace). The set of hyperplanes (4-d projective subspaces) can also be partitioned into disjoint sets of 3 hyperplanes each such that each set represents a unique 3-d projective subspace by the relation that it is contained in the 3 hyperplanes only.*

Proof. The proof is very similar to lemma 14. As before, V is represented by $GF(2^6)$ and since 2 divides 6, $GF(2^3)$ is a subfield and thus V can be partitioned into disjoint vector subspaces(-{0}) of dimension 2 each (using coset decomposition). Each of these subsets represents a 1-d projective subspace (line) and contains 3 points.

By duality of projective geometry, it follows that we can partition the hyperplanes into disjoint sets of 3 each such that each set represents a unique 3-d projective subspace. This can be achieved by performing a suitable coset decomposition of the dual vector space. □

Thus, we have partitioned the set of 63 points into 21 sets of 3 points each. In this

way we have a one-to-one correspondence between a point and the line that contains it. The following theorem is critical in showing that a conflict-free schedule can be developed.

We now provide certain lemmas, that will be needed to prove theorem 19 later.

Lemma 16. *The "union" of two disjoint lines(1-d projective subspaces) leads to a 3-d projective subspace. The union operation is defined as taking all possible linear combinations of the points in the two lines.*

Proof. Let the two disjoint lines be L_1 and L_2. Being disjoint, they have no points in common.

Each line, being a 2-d vector space contains exactly two independent points. Thus, two disjoint lines will contain 4 independent points. Taking a union, we get all possible linear combinations of the 4 independent points which corresponds to a 4-d vector space(lets call it T_{12}). The 4 independent points have been taken from the points of the 6-d vector space V, used to describe the projective space. Thus, the 4-d vector space, got from the linear combinations of the 4 points, is a subspace of V.

Being a 4-d vector subspace of V, T_{12} represents a 3-d projective subspace. □

Lemma 17. *For $\mathbb{PG}(5)$, let $\mathbb{L} = \{L_0, L_1, \ldots, L_{20}\}$ be the set of 21 disjoint lines obtained after coset decomposition of V. Let T_{ij} be the 3-d projective space obtained after taking the union of the lines L_i, L_j, both taken from the set $\mathbb{L}$. Then, any line L_k from the set $\mathbb{L}$, is either contained in T_{ij} or does not share any point with T_{ij}. Specifically, if it shares one point with T_{ij}, then it shares all its points with T_{ij}.*

Proof. Let α be the generator of the cyclic multiplicative group of $GF(2^6)$. Then the points of the projective space will be given by $\{\alpha^0, \alpha^1, \ldots, \alpha^{62}\}$ and for any integer i, $\alpha^i = \alpha^{(i \bmod 63)}$.

Lines are formed by 2-d vector subspaces. After the relevant coset decomposition of $GF(2^6)$, without losing generality, we may generate a correspondence between lines of $\mathbb{L}$ and the cosets as follows:

$$L_0 \equiv \{\alpha^0, \alpha^{21}, \alpha^{42}\}$$
$$L_1 \equiv \{\alpha^1, \alpha^{22}, \alpha^{43}\}$$
$$L_2 \equiv \{\alpha^2, \alpha^{23}, \alpha^{44}\}$$
$$\vdots$$
$$L_{19} \equiv \{\alpha^{19}, \alpha^{40}, \alpha^{61}\}$$
$$L_{20} \equiv \{\alpha^{20}, \alpha^{41}, \alpha^{62}\}$$

Now, $L_i \equiv \{\alpha^i, \alpha^{i+21}, \alpha^{i+42}\}$, where, $i + 21 \cong ((i + 21)\ mod\ 63)$ and $i + 42 \cong ((i + 42)\ mod\ 63)$.

Similarly, $L_j \equiv \{\alpha^j, \alpha^{j+21}, \alpha^{j+42}\}$

Now, T_{ij} is given by the union of L_i, L_j. Thus, T_{ij} contains all possible linear combinations of the points of L_i and L_j. Let us divide the points of T_{ij} into two parts:

1. The first part X_1 is given by the 6 points of L_i and L_j
2. The second part X_2 contains 9 points obtained by the linear combinations of the form $a\alpha^u + b\alpha^v$ where $\alpha^u \in L_i$ and $\alpha^v \in L_j$ and a, b take the non-zero values of $GF(2)$, i.e. $a = b = 1$.

Consider any line $L_k \in \mathbb{L}$.

- *Case 1:*

  If $k = i$ or $k = j$, then by the given construction, it is obvious that $L_k \subset T_{ij}$ and the lemma holds.

- *Case 2:*

  Here, $k \neq i, j$

  We have, $L_k \equiv \{\alpha^k, \alpha^{k+21}, \alpha^{k+42}\}$. Also, $L_k \in \mathbb{L}$ and $k \neq i, k$ implies that L_k is disjoint from L_i and L_j. Thus, it has no points in common with L_i and L_j.

  Since L_k has no points in common with L_i and L_j, it *cannot* have any points in common the set of points X_1 of T_{ij} defined above.

Now, we will prove that if L_k has even a single point in common with the set of points X_2 defined in point 2 above, then it has all its points in common with the set X_2 which implies that $L_k \subset T_{ij}$. If no points are in common, then L_k is not contained $T_{i,j}$ as required by the lemma.

Without loss of generality, let $\alpha^k = \alpha^u + \alpha^v$
where, $\alpha^u \in L_i$ and $\alpha^v \in L_j$
From the coset decomposition given above, it is clear that if $\alpha^k \in L_k$, then $\alpha^{k+21} \in L_k$, and $\alpha^{k+42} \in L_k$. Here again, $k+21 \cong ((k+21)\ mod\ 63)$ and $k+42 \cong ((k+42)\ mod\ 63)$

Since α is a generator of a multiplicative group, $\alpha^{k+21} = \alpha^k.\alpha^{21}$ and $\alpha^{k+42} = \alpha^k.\alpha^{42}$

Also, one of the fundamental properties of finite fields states that the elements are abelian with respect to multiplication and addition and the multiplication operator distributes over addition, i.e.

$$a.(b+c) = (b+c).a = a.b + a.c = b.a + c.a \quad (8.1)$$

where a, b, c are elements of the field.
Consider, α^{k+21}, We have,

$$\alpha^{k+21} = \alpha^k.\alpha^{21} \quad (8.2)$$

$$\Longrightarrow \alpha^{k+21} = (\alpha^u + \alpha^v).\alpha^{21} \quad (8.3)$$

$$\Longrightarrow \alpha^{k+21} = \alpha^u.\alpha^{21} + \alpha^v.\alpha^{21} \quad (8.4)$$

$$\Longrightarrow \alpha^{k+21} = \alpha^{u+21} + \alpha^{v+21} \quad (8.5)$$

Here, 8.4 follows because of 8.1 and as usual, the addition in the indices is taken modulo 63.

Since, $\alpha^u \in L_i$ and $\alpha^v \in L_j$, from the coset decomposition scheme, we have, $\alpha^{u+21} \in L_i$ and $\alpha^{v+21} \in L_j$. Thus, $\alpha^{u+21} \in T_{ij}$ and $\alpha^{v+21} \in T_{ij}$. And finally, $(\alpha^{u+21} + \alpha^{v+21}) \in T_{i,j}$ which has the straightforward implication that $\alpha^{k+21} \in T_{ij}$.

Analogous arguments for α^{k+42} prove that $\alpha^{k+42} \in T_{ij}$. Thus, all three points of L_k are contained in $T_{i.j}$ and $L_k \subset T_{i,j}$.

The arguments above show that any line $L_k \in \mathbb{L}$ either is completely contained in T_{ij} or has no intersecting points with it. For the sake of completeness, we present the points in T_{ij} so that is easy to "see" the lemma:

$$T_{ij} = \begin{bmatrix} \alpha^i & \alpha^{i+21} & \alpha^{i+42} \\ \alpha^j & \alpha^{j+21} & \alpha^{j+42} \\ \alpha^j + \alpha^i & \alpha^{j+21} + \alpha^i & \alpha^{j+42} + \alpha^i \\ \alpha^j + \alpha^{i+21} & \alpha^{j+21} + \alpha^{i+21} & \alpha^{j+42} + \alpha^{i+21} \\ \alpha^j + \alpha^{i+42} & \alpha^{j+21} + \alpha^{i+42} & \alpha^{j+42} + \alpha^{i+42} \end{bmatrix}$$

□

Lemma 18. *Given the set $\mathbb{L}$ of 21 disjoint lines that cover all the points of $\mathbb{PG}(5, GF(2))$, pick any $L_i \in \mathbb{L}$ and take its union with the remaining 20 lines in $\mathbb{L}$ to generate 20 3-d projective subspaces. Of these 20, only 5 distinct 3-d projective subspaces will exist.*

Proof. Any 3-d projective subspace has 3 hyperplanes associated with it. If a line is contained in a 3-d projective subspace, then it is contained in all the 3 hyperplanes associated with that 3-d subspace.

Also, if a line is not contained in a 3-d subspace, it is not contained in any of the hyperplanes associated with it. This is because:

$$dim(L_1 \cup T_1) = dim(L_1) + dim(T_1) - dim(L_1 \cap T_1)$$

$$dim(L_1) = 2\ ,\ dim(T_1) = 4$$

$$dim(L_1 \cap T_1) = 0$$

$$\implies dim(L_1 \cup T_1) = 6$$

where L_1 is the line, and T_1 is the 3-d projective subspace not containing the line. A hyperplane is a 5-d vector space and so if $dim(L_1 \cup T_1) > 5$, $L1$ is not contained in any hyperplane associated with T_1.

Let T_{ij} and T_{jk} be 2 4-d vector subspaces of V that represent 2 3-d projective subspaces. $dim(T_{ij}) = dim(T_{jk}) = 4$. Also,

$$dim(T_{ij} \cup T_{jk}) = dim(T_{ij}) + dim(T_{jk}) - dim(T_{ij} \cap T_{jk})$$

It is known that T_{ij} and T_{jk} have 0,1 or 3 common hyperplanes. No other case is possible. If they have three common hyperplanes, then $T_{ij} = T_{jk}$. This implies that $dim(T_{ij} \cap T_{jk}) = 4$.

If they have one hyperplane common, the 5-dimensional vector space corresponding to that hyperplane must contain both the 4-dimensional vector spaces. This is possible iff $dim(T_{ij} \cup T_{jk}) = 5 \Rightarrow dim(T_{ij} \cap T_{jk}) = 3$.

If they have no hyperplane in common, then $dim(T_{ij} \cup T_{jk}) = 6 \Rightarrow dim(T_{ij} \cap T_{jk}) = 2$.

Consider the union of line L_i with $L_j \in \mathbb{L}, j \neq i$. By Lemma 16, the union generates a 3-d projective space. Lets call it T_{ij}. Similarly, let the union of line L_i with $L_k \in \mathbb{L}, k \neq i, j$ be called T_{jk}.

By lemma 17, either $L_k \subset T_{ij}$ or $L_k \cap T_{ij} = 0$.

If $(L_i, L_k) \subset T_{ij}$, then $T_{jk} = T_{ij}$.

If $L_k \cap T_{ij} = 0$, then T_{jk} is distinct from T_{ij} and it adds 3 hyperplanes to L_i. Moreover, $dim(T_{ij} \cap T_{jk}) = 2$, which implies that T_{ij} and T_{jk} do not share any hyperplanes.

Applying this argument iteratively for the 20 3-d subspaces we see that a **maximum** of **5** distinct 3-d projective subspaces can be generated, each of which gives a cardinality of 3 hyperplanes to L_i, thus making 15 hyperplanes.

Each 3-d projective subspace, e.g. T_{ij} has 15 points and hence, it can contain a maximum of 5 disjoint lines. One of them is L_i, and another 4 need to be accounted for. So, when the union of L_i is taken with the remaining 20 lines, a maximum of 4 lines, out of these 20 lines, can give rise to same 3-d projective subspace, T_{ij}. This implies that a **minimum** of $20/4 = \mathbf{5}$ 3-d projective subspaces can be generated from the remaining 20 lines.

Since a maximum and minimum of 5 3-d projective subspaces can be generated, exactly 5 distinct 3-d projective subspaces are generated. Moreover, none of these subspaces

share any hyperplanes. □

Finally, the main theorem behind the construction of schedule mentioned in proposition 4, is as following.

Theorem 19. *In $\mathbb{PG}(5, GF(2))$, given a set of 21 disjoint lines (1-d projective subspaces) that cover all the points, a set of 21 disjoint 3-d projective subspaces can be created such that they cover all the hyperplanes.In this case, each point attains its degree of 31 hyperplanes in the following manner:*

1. *it gets a degree of 3 hyperplanes each from 5 3-d projective subspaces that contain the line corresponding to it.*

2. *it gets a degree of 1 hyperplane each from the remaining 16 3-d projective subspaces that necessarily cannot contain the line corresponding to it.*

The dual argument with the roles of points and hyperplanes interchanged also holds.

Proof. Generate the set of 21 disjoint lines $\mathbb{L}$ according to the coset decomposition corresponding to the subgroup isomorphic to $GF(2^2)$. We choose this subgroup as the *canonical* subgroup mentioned in theorem 19, i.e. $\{\alpha^0, \alpha^{21}, \alpha^{42}\}$.

Choose any line L_i from this set and take its union with the remaining 20 lines in the set to generate 5 distinct 3-d projective subspaces(as proved in lemma 18). Call these subspaces $T_1, T_2, \ldots, T_5$. Choose 5 distinct lines, each **NOT equal to** L_i, to represent each of these subspaces. Such a choice exists by lemma 16. Pick the line representing T_1, and take its union with the other 4 lines to generate 4 new 3-d projective subspaces. Choose 4 more lines(distinct from the 5 lines used earlier), to represent these 4 new subspaces. Again, such distinct lines exist by lemma 16, and there are overall 21 distinct lines. Pick another line from T_1 (not equal to the previously used lines), and take its union with the 4 newly chosen lines to form yet more 4 new 3-d projective subspaces. Repeat this process 2 more time, till you get 21 different 3-d subspaces, each contributing 3 distinct hyperplanes.

The following facts now hold for the so generated partitions of hyperplanes and points:

1. Each line in the set $\mathbb{L}$ lies in 5 3-d projective subspaces and each 3-d projective subspace contains 5 lines from the set $\mathbb{L}$.

2. A point in a line gets a degree of 3 hyperplanes from every 3-d subspace that contains the line and a degree of 1 hyperplane from every subspace that doesn't contain the line (since in this case if the subspace doesn't contain the line, it doesn't contain the point and hence will only contribute one hyperplane corresponding to the union of the point with the projective subspace).

From the above facts, all the points of the theorem follow. The dual argument holds in exactly the same way. You could have started with a partition of 3-d subspaces and generated lines by completely working in the dual vector space and using the exact same arguments. Hence, there are two ways of folding for each partition of V into disjoint lines. □

Proof of Proposition 4

From the lemmas stated above, it is quiet clear that we have a system such that the graph can be folded easily and a scheduling similar to the one used for proposition 1 can be developed.

We begin by assigning one processing unit to every line of the disjoint set. Each processing unit has an associated memory. After the 3-d subspaces have been created as explained above, we can assign a 3-d space to each of the memories. For Type 1 computation, the computation is again divided into two phases. In phase 1, the points on a particular line are scheduled on the processor corresponding to that line in a sequential manner. A point gets 3 data units from a memory if the 3-d projective space corresponding to that memory contains the line, otherwise it gets 1. In phase 2, the memory already has data corresponding to the hyperplanes that contain the 3-d subspace representing the memory and the communication is just between the processing unit and its own memory. The output write-back cycles follow the schedule of input reads in both phases.

In Type 2 computation, the schedule is similar to the phase 1 schedule used in Type 1 computation, but depends on the application. No phase two exists. Thus, using 21 processors with appropriate address generation schemes, we can fold the computation efficiently in this case also.

8.4 Generalization to $\mathbb{PG}(m, GF(q))$

In this section, we generalize the above propositions for projective geometries of dimension m, when $m + 1$ is not a prime number.

A $\mathbb{PG}(m, GF(q))$ is represented using the elements of a vector space V of dimension $m + 1$ over $GF(q)$. If $m + 1$ is not prime, it can be factored into prime factors $p1, p2, p3, \ldots, pn$ such that $p1 * p2 * \ldots * pn = m + 1$. Also, the dimensions of the projective subspaces vary from 0 to $(m - 1)$ and the dimensions of the corresponding vector subspaces of V vary from 1 to m. The points are the 0 dimensional projective subspaces (represented by the elements of V) and the hyperplanes are the $m - 1$ dimensional projective subspaces.

It is convenient to describe the folding in two separate cases; even though the first case is a sub-case of the second one.

- *Case 1* :

  Suppose $m + 1$ is even. Then we can partition the set of points (has cardinality $\frac{q^{m+1}-1}{q-1}$) into disjoint sets; each of dimension $\frac{m+1}{2}$ and thus containing $\frac{q^{\frac{(m+1)}{2}}-1}{q-1}$ each. The existence of such a partition can easily be explained by using the arguments similar to that of lemma 1. The vector space V would be represented by $GF(q^{m+1})$. Since $m+1$ is even, $\frac{m+1}{2}$ divides it and $GF(q^{\frac{m+1}{2}})$ is a sub-field of $GF(q^{m+1})$. Thus, V can be partitioned into disjoint subspaces (say S_is), each of dimension $\frac{m+1}{2}$ and by property of vector sub-spaces, if $x \in S_i$ then $\lambda x \in S_i, \lambda \epsilon GF(q)$. So the equivalence relation of points holds in the subspaces also. Each subspace represents a $\frac{m-1}{2}$ dimensional projective subspace.

  Now because of duality of points and hyperplanes, there are a equal number of hyperplanes associated with each $\frac{m+1}{2} - 1$ dimensional projective subspace. Moreover, since we have a disjoint partition of points, there will exist a corresponding disjoint partition of hyperplanes. The number of partitions equals $\frac{q^{m+1}-1}{q^{\frac{(m+1)}{2}}-1}$.

  Each point has a total of $\frac{q^{m}-1}{q-1}$ hyperplanes containing it; $\frac{q^{\frac{(m+1)}{2}}-1}{q-1}$ hyperplanes from the partition that contains it and the rest from the remaining partitions in the following manner:

$$\begin{aligned}\text{No. of partitions remaining} &= \frac{q^{m+1}-1}{q^{\frac{(m+1)}{2}}-1} - 1 \\ &= q^{\frac{(m+1)}{2}}\end{aligned}$$

Each partition is formed by $\frac{m+1}{2}$ independent points and all their linear combinations over coefficients from $GF(q)$. The total number of points in a $\frac{m-1}{2}$ dimensional vector space is $\frac{q^{\frac{(m-1)}{2}}-1}{q-1}$. Therefore, in any set of $\frac{q^{\frac{(m+1)}{2}}-1}{q-1}+1$ points, it is possible to find $\frac{m+1}{2}$ independent points.

Any point that does not lie in a $\frac{m-1}{2}$ dimensional projective subspace reaches out to exactly $\frac{q^{\frac{(m-1)}{2}}-1}{q-1}$ hyperplanes through that projective subspace. Any more and we could find $\frac{m+1}{2}$ independent points which would imply that the point lies in that subspace. Any less, and the point would not achieve the established number of hyperplanes in the above partition as:

$$\begin{aligned}\text{Hyperplanes from the sets not containing the point} &= \frac{q^{\frac{(m-1)}{2}}-1}{q-1} * q^{\frac{(m+1)}{2}} \\ &= \frac{q^m - q^{\frac{(m+1)}{2}}}{q-1} \\ \text{Total degree} &= \frac{q^{\frac{(m+1)}{2}}-1}{q-1} + \frac{q^m - q^{\frac{(m+1)}{2}}}{q-1} \\ &= \frac{q^m-1}{q-1} \\ &\text{as required}\end{aligned}$$

Given the above construction, it is easy to develop an architecture and scheduling strategy as in proposition 1. One processor could be assigned to each of the disjoint sets of points. Additional folds are possible by assigning multiple disjoint sets to each processing unit as long as the "symmetry" of the schedule (with regards to the amount of data required from local memory and data required from memories of other processing units) is maintained.

- *Case 2*:

  Let $m+1 = (k+1)*t$, where $k > 0$ and $t \geq 3$ ($t = 2$ comes under case 1). There exists a projective subspace of dimension k and its dual space will be of dimension $m-k-1$. We will use these subspaces to partition the points and hyperplanes into disjoint sets and then assign these sets to processing units.

Now, V has dimension $m+1$ and the vector subspace corresponding to the projective subspace of dimension k has a dimension of $k+1$. Since, $k+1$ divides $m+1$, we can partition the points into disjoint sets, each set having $k+1$ independent points. The sets are obtained by coset decomposition of multiplicative group of $GF(q^{m+1})$. Let $\mathbb{S}$ denote the collection of these sets and let the ith set be denoted by S_i. Since $t \geq 3$, $k \leq \lfloor \frac{m+1}{2} \rfloor$.

We will construct the dual projective subspaces from these sets such that they do not share any hyperplanes and together cover all the hyperplanes, thus creating a disjoint partition. $m-k$ independent points are required to create a $m-k-1$ dimensional projective subspace. Since $\frac{m-k}{k+1} = \frac{(m+1)-(k+1)}{k+1} = t-1$, the union of $t-1$ disjoint sets taken from $\mathbb{S}$, each having $k+1$ independent points, and the points that are all possible linear combinations over $GF(q)$ of these, will form a $m-k-1$ dimensional projective subspace.

Let $S_0, S_1, S_2, S_3, \ldots, S_{t-2} \in \mathbb{S}$ be combined to make the $m-k-1$ dimensional projective space T_1.

$S_0, S_1, S_2, S_3, \ldots, S_{t-2}$ are sets of cosets that have been obtained by the coset decomposition of the nonzero elements of $GF(q^{m+1})$. Further, an *equivalent* set of points of projective space can be obtained from each coset S_i as the set of equivalence classes using the equivalence relation $a_i = \lambda \cdot a_j$, where a_i, $a_j \in S_i$, and $\lambda \in \mathbb{GF}(q)$.

Therefore, they can be written as:

$$S_i \equiv \{\alpha^i, \alpha^{i+\beta}, \alpha^{i+2\beta}, \ldots (q^{k+1}-1) \text{ terms}\}$$

where, α is the generator of the multiplicative group of $GF(q^{m+1})$, and $\{0, \alpha^0, \alpha^\beta, \alpha^{2\beta}, \ldots (q^{k+1}-1) \text{ terms}\}$, where $\beta = \frac{q^{m+1}-1}{q^{k+1}-1}$ forms a subfield of $GF(q^{m+1})$ that is isomorphic to $GF(q^{k+1})$.

Moreover, any $S_i \in \mathbb{S}$ contains only the **non-zero** elements of a vector space over $GF(q)$, and their all possible linear combinations of the form $(c_0a_0+c_1a_1+\ldots+c_na_n)$ $\forall c_0, c_1, \ldots, c_n \in GF(q)$. Here, $a_0, a_1 \ldots, a_n \in S_i$, and all c's are not all simultaneously 0.

Consider $S_k \in \mathbb{S}, k \neq 0, 1, 2, 3, \ldots, (t-2)$. Similar to lemma 17, we will show that if even one point of S_k is common with T_1, then all points of S_k must be common and thus, $S_k \subset T_1$.

Divide the set of points of T_1 into two parts:
X_1 consists of the points of $S_0, S_1, \ldots, S_{(t-2)}$.
X_2 is the set of points of the form $c_0\alpha^{u_0} + c_1\alpha^{u_1} + \ldots + c_{(t-2)}\alpha^{u_{(t-2)}}$
where $\alpha_i^u \in S_i$ and $c_i \in GF(q)$ and there are at least two non-zero c_i's.

Moreover, if $\alpha^{u_i} \in S_i$, then $c_i\alpha^{u_i} \in S_i \forall c_i \in GF(q)$. Therefore, we can abuse notation and simply write

$$X_2 = \alpha^{u_0} + \alpha^{u_1} + \ldots + \alpha^{u_{(t-2)}} \tag{8.6}$$

such that there are at least two non-zero terms in the summation.

Let $S_k \equiv \{\alpha^k, \alpha^{k+\beta}, \alpha^{k+2\beta}, \cdots\}$. Consider $\alpha^k \in S_k$. It is clear that since S_k is disjoint from $S_i,\ i \in 0, 1, \ldots, (t-2)$, $\alpha^k \notin X_1$.

Suppose if $\alpha^k \in X_2$, then we have,

$$\alpha^{k+\beta} = \alpha^k.\alpha^\beta \tag{8.7}$$

$$\implies \alpha^{k+\beta} = (\alpha^{u_0} + \alpha^{u_1} + \ldots + \alpha^{u_{(t-2)}}).\alpha^\beta \tag{8.8}$$

$$\implies \alpha^{k+\beta} = \alpha^{u_0+\beta} + \alpha^{u_1+\beta} + \ldots + \alpha^{u_{(t-2)}+\beta} \tag{8.9}$$

Now, if $\alpha^i \in S_j$ for some i, j, then $\alpha^{(i+\beta)} \in S_j$. Therefore, it is clear that equation 8.9 represents some linear combination of elements of $S_0, \ldots, S_{t-2}$ and hence *must* be contained in T_1.

Proceeding in a similar way for all multiples of β, we find that all points $\in S_k$ are eventually found part of T_1, where we started just by having one point being part of T_1.

Thus, if T_1 is generated by the union of $S_0, \ldots, S_{(t-2)}$, the remaining S_i's are either contained in T_1 or have no intersection with T_1. If some S_i has no intersection with T_1, then replacing S_0 *or* S_1 *or* ... *or* $S_{(t-2)}$ with S_i will each generate a different (m-k-1)-dimensional projective space T, which is distinct from T_1 for each substitution that is made.

Let T_i and T_j be two distinct subspaces, each generated by union of disjoint S_i's. Between these two sets, there will be some sets S_k which are common as per the construction mentioned above. The number of shared independent points, that are contained in these shared sets S_k, is $(t-2)*(k+1)$. Thus, we have, $dim(T_i \cap T_j) = (t-2)*(k+1)$

$$\begin{aligned} dim(T_i \cup T_j) &= dim(T_i) + dim(T_j) - dim(T_i \cap T_j) \\ &= 2m - 2k - (k+1)*(t-2) \\ &= 2m - 2k - (k+1)*t + 2(k+1) \\ &= m + 1 \quad \text{(Since, m+1=(k+1)*t)} \end{aligned}$$

Since $dim(T_i \cup T_j) > m$, they have no hyperplanes in common.

Therefore, if we generate all the distinct T_i's, we will get a disjoint partition of the hyperplane set. The number of hyperplanes containing a T_i ($m-k-1$ dimensional projective subspace) is equal to $\phi(m-(m-k-1)-1, (m-1)-(m-k-1)-1, q) = \phi(k, k-1, q)$ where ϕ is the function defined in section 3.3.

A construction and schedule analogous to the one used for Theorem 19 may be generated. For Type 1 Computation, we begin by assigning one processor for each S_i. To each of these processors, we also assign one T_i containing the corresponding S_i. The points in S_i are executed on the corresponding processor in a sequential fashion. Each point gets $\phi(k, k-1, q)$ hyperplanes from every T_k that contains S_i. The remaining degree will be evenly distributed among the remaining T_k's. Specifically,

If $S_i \cap T_i = 0$, then :

$$\begin{aligned} dim(T_i \cup S_i) &= dim(T_i) + dim(S_i) - dim(T_i \cap S_i) \\ &= m - k + k + 1 - 0 \\ &= m + 1 \end{aligned}$$

Therefore, if $S_i \cap T_i = 0$, then S_i gets no hyperplanes from T_i.

Each S_i gets $\phi(m-k-1,(m-1)-k-1,q)$ hyperplanes in all, out of $\phi(m,(m-1),q)$ hyperplanes, which it must get from $N = \frac{\phi(m-k-1,(m-1)-k-1,q)}{\phi(k,k-1,q)}$ T's reachable from it.

So every point A, gets $\phi(m-k-1,(m-1)-k-1,q)$ hyperplanes from $N = \frac{\phi(m-k-1,(m-1)-k-1,q)}{\phi(k,k-1,q)}$ T_k's containing the S_i which itself contains this point.

Moreover, for the remaining T_j's (the ones not containing the point A) have the following property:

$$\begin{aligned} dim(T_j \cup A) &= dim(T_j) + dim(A) - dim(T_j \cap A) \\ &= m - k + 1 \\ &= m + 1 - k \end{aligned}$$

Therefore, the number of hyperplanes got from T_j will be the number of hyperplanes containing the subspace $(T_j \cup A)$. It is a $m+1-k$ vector subspace and therefore is a $m-k$ projective subspace. The number of hyperplanes containing it is given by: $\phi(m-(m+-k)-1, m-1-(m-k)-1, q) = \phi(k-1, k-2, q)$

So now a schedule can easily be generated utilizing the incidence relations stated above.

For Type 2 computations, a schedule for the phase 1 is to be generated. Moreover, the schedule will be application dependent. It depends on the number of operands needed from memory corresponding to each point and a corresponding perfect access pattern generation.

Chapter 9

Conclusion

We have presented the construction, performance analysis and hardware design strategies for a expander-like decoder that is based on a bipartite graph. The graph if derived from the incidence relations of projective spaces offers unique advantages in terms of deriving lower bounds on error correction capabilities, and there is a fundamental advantage in terms of hardware design. As the size of the graph increases, practical implementation of the code become difficult. Projective geometry through lattice embedding properties offers a natural way of folding the computations which leads to using fewer processors. The folding is special because it does not involve any data transfer between memories between computation cycles, every processor is utilized completely during the computation cycle and the address generation circuits can be simplified due to the structured nature of the memory accesses.

The code performance is better than previously stated in literature as it relaxes some restrictions that were imposed with respect to the second largest eigenvalue of the graph. Derivation of bounds of error correction have been presented and the average case performance of the code is shown to be up to 10 times better through simulations. Moreover, the code has special implicit interleaving due to the numbering of the edges and this offers great advantage in burst error corrections. A natural application of these codes with respect to data storage media (namely CD-ROMs and DVD-R) has been explored and we have presented schemes that improve the burst error performance in comparison to existing standards.

The hardware design for the decoder prototype has been completely worked out. We use Xilinx RS decoder IPs as the processors. The computations have been efficiently

folded in order to make them fit on a Xilinx LX110T FPGA. We have tested the design on the FPGA and also exploited the erasure correction ability of the RS code. The ASIC implementation of the decoder is expected to be much more optimal and we would expect a much higher throughput. Moreover, a general folding strategy has been developed for higher dimensions of projective geometry that provides a methodology for practically implementing decoders of higher dimensions.

9.1 Future Work

In this section we would like to present a couple of open problems that could prove to be challenging and exciting for anyone interested in this area.

- As stated before, the combinatorial bounds hold for the practical values of ϵ. However, beyond a certain point, we are forced to revert back to eigenvalue arguments to obtain the bounds. A general formulation of the problem is as follows :
  We would like to solve the following general optimization problem using the properties of projective geometry. The problem is as follows,

$$\begin{aligned} \text{Given,} \quad \mathbf{x_s^T A x_s} - \gamma \mathbf{x_s^T x_s} &\geq 0, \quad \mathbf{x_s} \,\epsilon\, \{0,1\}^{2n} \\ \text{which is equivalent to,} \quad (\mathbf{A} - \gamma \mathbf{I})\mathbf{x_s} &\geq 0, \quad \mathbf{x_s} \,\epsilon\, \{0,1\}^{2n} \\ \text{minimize} \quad \mathbf{x_s^T x_s} \end{aligned}$$

  We need to find non-trivial solutions of the above problem. $\mathbf{x_s}$ and $\mathbf{A}$ are the same as mentioned in chapter 5. The value of ξ is then given by $\mathbf{x_s^T x_s}$.

- The second problem is with respect to the rate of the code. It should be noted that the value of the rate of the overall code $(2 * r - 1)$ is a lower bound and is obtained by assuming that all the constraints corresponding to the vertices of the graph are independent.

  In general, the performance of our decoding algorithm is independent of the assignment of edges as message or parity symbols of the subcodes. Thus, we would like to see if by changing this arrangement, can we get a higher rate for the code while keeping the error correction capacity the same. Hoholdt [6] has addressed the problem of the rate of the code without explicitly looking at the edge assignment

problem. Intuitively it looks like if we were able to arrange the code to 'look like' a product code (The parity symbols of one side of the graph become the message symbols of the other side), we would be able to get a higher rate. If we map the decoding algorithm to mimic decoding of product codes, we would be able to achieve D/4 error correction (D is the minimum distance of the overall code and D/4 would be the correction capability of the corresponding product code). But to do this, we must be able to manipulate the parity matrix of the overall code to make it the parity matrix of a product code. At the sub-code level, we need to figure out which edges to put as message symbols of the subcode and which to put as parity. The parity symbols due to the left hand side of the graph must appear as the message symbols in the right hand side of the graph. This assignment problem itself could be very difficult.

If d1=d2=d, rate of a product code for RS subcodes would be as follows: $r(\text{subcode}) = \frac{(n-(d-1))}{n} = (1 - [\frac{(d-1)}{n}]) = 1 - z(\text{say})$ for product code:$R = r^2 = 1 - 2z + z^2$
which is greater than $(2r-1)(= 1 - 2z)$
Thus, it should be possible to get the same error correction capacities for our code at higher rates.

- Thirdly, the problem of designing a linear time encoding system is still open. Using properties of projective geometry, we should be able to utilize the structure of the parity/generator matrices to develop a practically efficient encoding system

A Appendix

A.1 The Graph used for the decoder

Table 1: Point-Hyperplane Adjacency List

Hyperplane	Adjacent Vertices
0	63, 64, 65, 66, 67, 69, 70, 71, 72, 75, 76, 77, 79, 81, 82, 87, 89, 90, 91, 95, 96, 98, 99, 101, 104, 108, 111, 112, 115, 117, 119
1	64, 65, 66, 67, 68, 70, 71, 72, 73, 76, 77, 78, 80, 82, 83, 88, 90, 91, 92, 96, 97, 99, 100, 102, 105, 109, 112, 113, 116, 118, 120
2	65, 66, 67, 68, 69, 71, 72, 73, 74, 77, 78, 79, 81, 83, 84, 89, 91, 92, 93, 97, 98, 100, 101, 103, 106, 110, 113, 114, 117, 119, 121
3	66, 67, 68, 69, 70, 72, 73, 74, 75, 78, 79, 80, 82, 84, 85, 90, 92, 93, 94, 98, 99, 101, 102, 104, 107, 111, 114, 115, 118, 120, 122
4	67, 68, 69, 70, 71, 73, 74, 75, 76, 79, 80, 81, 83, 85, 86, 91, 93, 94, 95, 99, 100, 102, 103, 105, 108, 112, 115, 116, 119, 121, 123
5	68, 69, 70, 71, 72, 74, 75, 76, 77, 80, 81, 82, 84, 86, 87, 92, 94, 95, 96, 100, 101, 103, 104, 106, 109, 113, 116, 117, 120, 122, 124
6	69, 70, 71, 72, 73, 75, 76, 77, 78, 81, 82, 83, 85, 87, 88, 93, 95, 96, 97, 101, 102, 104, 105, 107, 110, 114, 117, 118, 121, 123, 125
7	70, 71, 72, 73, 74, 76, 77, 78, 79, 82, 83, 84, 86, 88, 89, 94, 96, 97, 98, 102, 103, 105, 106, 108, 111, 115, 118, 119, 122, 124, 63
8	71, 72, 73, 74, 75, 77, 78, 79, 80, 83, 84, 85, 87, 89, 90, 95, 97, 98, 99, 103, 104, 106, 107, 109, 112, 116, 119, 120, 123, 125, 64
9	72, 73, 74, 75, 76, 78, 79, 80, 81, 84, 85, 86, 88, 90, 91, 96, 98, 99, 100, 104, 105, 107, 108, 110, 113, 117, 120, 121, 124, 63, 65
10	73, 74, 75, 76, 77, 79, 80, 81, 82, 85, 86, 87, 89, 91, 92, 97, 99, 100, 101, 105, 106, 108, 109, 111, 114, 118, 121, 122, 125, 64, 66
11	74, 75, 76, 77, 78, 80, 81, 82, 83, 86, 87, 88, 90, 92, 93, 98, 100, 101, 102, 106, 107, 109, 110, 112, 115, 119, 122, 123, 63, 65, 67

Continued on next page

Table 1 – continued from previous page

Hyperplane	Adjacent Vertices
12	75, 76, 77, 78, 79, 81, 82, 83, 84, 87, 88, 89, 91, 93, 94, 99, 101, 102, 103, 107, 108, 110, 111, 113, 116, 120, 123, 124, 64, 66, 68
13	76, 77, 78, 79, 80, 82, 83, 84, 85, 88, 89, 90, 92, 94, 95, 100, 102, 103, 104, 108, 109, 111, 112, 114, 117, 121, 124, 125, 65, 67, 69
14	77, 78, 79, 80, 81, 83, 84, 85, 86, 89, 90, 91, 93, 95, 96, 101, 103, 104, 105, 109, 110, 112, 113, 115, 118, 122, 125, 63, 66, 68, 70
15	78, 79, 80, 81, 82, 84, 85, 86, 87, 90, 91, 92, 94, 96, 97, 102, 104, 105, 106, 110, 111, 113, 114, 116, 119, 123, 63, 64, 67, 69, 71
16	79, 80, 81, 82, 83, 85, 86, 87, 88, 91, 92, 93, 95, 97, 98, 103, 105, 106, 107, 111, 112, 114, 115, 117, 120, 124, 64, 65, 68, 70, 72
17	80, 81, 82, 83, 84, 86, 87, 88, 89, 92, 93, 94, 96, 98, 99, 104, 106, 107, 108, 112, 113, 115, 116, 118, 121, 125, 65, 66, 69, 71, 73
18	81, 82, 83, 84, 85, 87, 88, 89, 90, 93, 94, 95, 97, 99, 100, 105, 107, 108, 109, 113, 114, 116, 117, 119, 122, 63, 66, 67, 70, 72, 74
19	82, 83, 84, 85, 86, 88, 89, 90, 91, 94, 95, 96, 98, 100, 101, 106, 108, 109, 110, 114, 115, 117, 118, 120, 123, 64, 67, 68, 71, 73, 75
20	83, 84, 85, 86, 87, 89, 90, 91, 92, 95, 96, 97, 99, 101, 102, 107, 109, 110, 111, 115, 116, 118, 119, 121, 124, 65, 68, 69, 72, 74, 76
21	84, 85, 86, 87, 88, 90, 91, 92, 93, 96, 97, 98, 100, 102, 103, 108, 110, 111, 112, 116, 117, 119, 120, 122, 125, 66, 69, 70, 73, 75, 77
22	85, 86, 87, 88, 89, 91, 92, 93, 94, 97, 98, 99, 101, 103, 104, 109, 111, 112, 113, 117, 118, 120, 121, 123, 63, 67, 70, 71, 74, 76, 78
23	86, 87, 88, 89, 90, 92, 93, 94, 95, 98, 99, 100, 102, 104, 105, 110, 112, 113, 114, 118, 119, 121, 122, 124, 64, 68, 71, 72, 75, 77, 79
24	87, 88, 89, 90, 91, 93, 94, 95, 96, 99, 100, 101, 103, 105, 106, 111, 113, 114, 115, 119, 120, 122, 123, 125, 65, 69, 72, 73, 76, 78, 80
25	88, 89, 90, 91, 92, 94, 95, 96, 97, 100, 101, 102, 104, 106, 107, 112, 114, 115, 116, 120, 121, 123, 124, 63, 66, 70, 73, 74, 77, 79, 81

Continued on next page

Table 1 – continued from previous page

Hyperplane	Adjacent Vertices
26	89, 90, 91, 92, 93, 95, 96, 97, 98, 101, 102, 103, 105, 107, 108, 113, 115, 116, 117, 121, 122, 124, 125, 64, 67, 71, 74, 75, 78, 80, 82
27	90, 91, 92, 93, 94, 96, 97, 98, 99, 102, 103, 104, 106, 108, 109, 114, 116, 117, 118, 122, 123, 125, 63, 65, 68, 72, 75, 76, 79, 81, 83
28	91, 92, 93, 94, 95, 97, 98, 99, 100, 103, 104, 105, 107, 109, 110, 115, 117, 118, 119, 123, 124, 63, 64, 66, 69, 73, 76, 77, 80, 82, 84
29	92, 93, 94, 95, 96, 98, 99, 100, 101, 104, 105, 106, 108, 110, 111, 116, 118, 119, 120, 124, 125, 64, 65, 67, 70, 74, 77, 78, 81, 83, 85
30	93, 94, 95, 96, 97, 99, 100, 101, 102, 105, 106, 107, 109, 111, 112, 117, 119, 120, 121, 125, 63, 65, 66, 68, 71, 75, 78, 79, 82, 84, 86
31	94, 95, 96, 97, 98, 100, 101, 102, 103, 106, 107, 108, 110, 112, 113, 118, 120, 121, 122, 63, 64, 66, 67, 69, 72, 76, 79, 80, 83, 85, 87
32	95, 96, 97, 98, 99, 101, 102, 103, 104, 107, 108, 109, 111, 113, 114, 119, 121, 122, 123, 64, 65, 67, 68, 70, 73, 77, 80, 81, 84, 86, 88
33	96, 97, 98, 99, 100, 102, 103, 104, 105, 108, 109, 110, 112, 114, 115, 120, 122, 123, 124, 65, 66, 68, 69, 71, 74, 78, 81, 82, 85, 87, 89
34	97, 98, 99, 100, 101, 103, 104, 105, 106, 109, 110, 111, 113, 115, 116, 121, 123, 124, 125, 66, 67, 69, 70, 72, 75, 79, 82, 83, 86, 88, 90
35	98, 99, 100, 101, 102, 104, 105, 106, 107, 110, 111, 112, 114, 116, 117, 122, 124, 125, 63, 67, 68, 70, 71, 73, 76, 80, 83, 84, 87, 89, 91
36	99, 100, 101, 102, 103, 105, 106, 107, 108, 111, 112, 113, 115, 117, 118, 123, 125, 63, 64, 68, 69, 71, 72, 74, 77, 81, 84, 85, 88, 90, 92
37	100, 101, 102, 103, 104, 106, 107, 108, 109, 112, 113, 114, 116, 118, 119, 124, 63, 64, 65, 69, 70, 72, 73, 75, 78, 82, 85, 86, 89, 91, 93
38	101, 102, 103, 104, 105, 107, 108, 109, 110, 113, 114, 115, 117, 119, 120, 125, 64, 65, 66, 70, 71, 73, 74, 76, 79, 83, 86, 87, 90, 92, 94
39	102, 103, 104, 105, 106, 108, 109, 110, 111, 114, 115, 116, 118, 120, 121, 63, 65, 66, 67, 71, 72, 74, 75, 77, 80, 84, 87, 88, 91, 93, 95

Continued on next page

Table 1 – continued from previous page

Hyperplane	Adjacent Vertices
40	103, 104, 105, 106, 107, 109, 110, 111, 112, 115, 116, 117, 119, 121, 122, 64, 66, 67, 68, 72, 73, 75, 76, 78, 81, 85, 88, 89, 92, 94, 96
41	104, 105, 106, 107, 108, 110, 111, 112, 113, 116, 117, 118, 120, 122, 123, 65, 67, 68, 69, 73, 74, 76, 77, 79, 82, 86, 89, 90, 93, 95, 97
42	105, 106, 107, 108, 109, 111, 112, 113, 114, 117, 118, 119, 121, 123, 124, 66, 68, 69, 70, 74, 75, 77, 78, 80, 83, 87, 90, 91, 94, 96, 98
43	106, 107, 108, 109, 110, 112, 113, 114, 115, 118, 119, 120, 122, 124, 125, 67, 69, 70, 71, 75, 76, 78, 79, 81, 84, 88, 91, 92, 95, 97, 99
44	107, 108, 109, 110, 111, 113, 114, 115, 116, 119, 120, 121, 123, 125, 63, 68, 70, 71, 72, 76, 77, 79, 80, 82, 85, 89, 92, 93, 96, 98, 100
45	108, 109, 110, 111, 112, 114, 115, 116, 117, 120, 121, 122, 124, 63, 64, 69, 71, 72, 73, 77, 78, 80, 81, 83, 86, 90, 93, 94, 97, 99, 101
46	109, 110, 111, 112, 113, 115, 116, 117, 118, 121, 122, 123, 125, 64, 65, 70, 72, 73, 74, 78, 79, 81, 82, 84, 87, 91, 94, 95, 98, 100, 102
47	110, 111, 112, 113, 114, 116, 117, 118, 119, 122, 123, 124, 63, 65, 66, 71, 73, 74, 75, 79, 80, 82, 83, 85, 88, 92, 95, 96, 99, 101, 103
48	111, 112, 113, 114, 115, 117, 118, 119, 120, 123, 124, 125, 64, 66, 67, 72, 74, 75, 76, 80, 81, 83, 84, 86, 89, 93, 96, 97, 100, 102, 104
49	112, 113, 114, 115, 116, 118, 119, 120, 121, 124, 125, 63, 65, 67, 68, 73, 75, 76, 77, 81, 82, 84, 85, 87, 90, 94, 97, 98, 101, 103, 105
50	113, 114, 115, 116, 117, 119, 120, 121, 122, 125, 63, 64, 66, 68, 69, 74, 76, 77, 78, 82, 83, 85, 86, 88, 91, 95, 98, 99, 102, 104, 106
51	114, 115, 116, 117, 118, 120, 121, 122, 123, 63, 64, 65, 67, 69, 70, 75, 77, 78, 79, 83, 84, 86, 87, 89, 92, 96, 99, 100, 103, 105, 107
52	115, 116, 117, 118, 119, 121, 122, 123, 124, 64, 65, 66, 68, 70, 71, 76, 78, 79, 80, 84, 85, 87, 88, 90, 93, 97, 100, 101, 104, 106, 108
53	116, 117, 118, 119, 120, 122, 123, 124, 125, 65, 66, 67, 69, 71, 72, 77, 79, 80, 81, 85, 86, 88, 89, 91, 94, 98, 101, 102, 105, 107, 109

Continued on next page

Table 1 – continued from previous page

Hyperplane	Adjacent Vertices
54	117, 118, 119, 120, 121, 123, 124, 125, 63, 66, 67, 68, 70, 72, 73, 78, 80, 81, 82, 86, 87, 89, 90, 92, 95, 99, 102, 103, 106, 108, 110
55	118, 119, 120, 121, 122, 124, 125, 63, 64, 67, 68, 69, 71, 73, 74, 79, 81, 82, 83, 87, 88, 90, 91, 93, 96, 100, 103, 104, 107, 109, 111
56	119, 120, 121, 122, 123, 125, 63, 64, 65, 68, 69, 70, 72, 74, 75, 80, 82, 83, 84, 88, 89, 91, 92, 94, 97, 101, 104, 105, 108, 110, 112
57	120, 121, 122, 123, 124, 63, 64, 65, 66, 69, 70, 71, 73, 75, 76, 81, 83, 84, 85, 89, 90, 92, 93, 95, 98, 102, 105, 106, 109, 111, 113
58	121, 122, 123, 124, 125, 64, 65, 66, 67, 70, 71, 72, 74, 76, 77, 82, 84, 85, 86, 90, 91, 93, 94, 96, 99, 103, 106, 107, 110, 112, 114
59	122, 123, 124, 125, 63, 65, 66, 67, 68, 71, 72, 73, 75, 77, 78, 83, 85, 86, 87, 91, 92, 94, 95, 97, 100, 104, 107, 108, 111, 113, 115
60	123, 124, 125, 63, 64, 66, 67, 68, 69, 72, 73, 74, 76, 78, 79, 84, 86, 87, 88, 92, 93, 95, 96, 98, 101, 105, 108, 109, 112, 114, 116
61	124, 125, 63, 64, 65, 67, 68, 69, 70, 73, 74, 75, 77, 79, 80, 85, 87, 88, 89, 93, 94, 96, 97, 99, 102, 106, 109, 110, 113, 115, 117
62	125, 63, 64, 65, 66, 68, 69, 70, 71, 74, 75, 76, 78, 80, 81, 86, 88, 89, 90, 94, 95, 97, 98, 100, 103, 107, 110, 111, 114, 116, 118

A.2 Distribution of Data

Table 2: memory block-Edge Storage Correspondence

memory Block No.	Edge numbers in the order in which they are stored
1	0, 63, 126, 315, 378, 567, 1008, 6, 69, 321, 130, 193, 445, 1715, 1904, 581, 1777, 1903, 643, 1776, 12, 705, 1836, 1899, 198, 68, 131, 383, 1, 64, 316, 1045, 1108, 1171, 1234, 1297, 1486, 1801, 1595, 1721, 335, 1898, 260, 827, 1655, 1781, 521, 1593, 1782, 459, 1654, 1717, 1843, 1535, 1787, 212, 1897, 7, 889, 2, 254, 947, 618, 681, 744, 870, 933, 996, 1500, 1474, 1600, 88, 1653, 1716, 1842, 1107, 1170, 1296, 1594, 1909, 397, 1659, 1722, 1848, 1351, 1414, 1540, 1837, 136, 766, 192, 255, 507, 434, 497, 560, 686, 749, 938, 1316, 1475, 1727, 26, 926, 1052, 1619, 1051, 1114, 1240, 1413, 1476, 1665, 1534, 1660, 274, 1352, 1415, 1541, 1775, 1838, 74, 985, 1363, 1741, 372, 435, 498, 624, 687, 876, 1254, 1290, 1353, 1605, 803, 866, 1181, 865, 928, 991, 1046, 1109, 1235, 1536, 1599, 150, 1230, 1419, 1923, 986, 1175, 1679, 558, 810, 1377, 187, 250, 313, 502, 565, 754, 1195, 1228, 1291, 1480, 619, 682, 871, 804, 1119, 1560, 1113, 1176, 1302, 984, 1047, 1425, 1357, 1420, 1546, 680, 743, 932, 311, 374, 815, 125, 188, 251, 440, 503, 692, 1133, 1229, 1292, 1481, 620, 809, 1439, 742, 805, 1057, 864, 927, 990, 496, 559, 748, 1169, 1358, 1862, 373, 436, 625, 249, 312, 564
2	127, 190, 253, 442, 505, 694, 1135, 189, 252, 504, 195, 258, 510, 4, 67, 319, 1778, 1841, 77, 1840, 139, 769, 1839, 1902, 201, 9, 72, 324, 5, 257, 950, 65, 128, 191, 380, 443, 632, 1073, 1360, 1423, 1549, 1658, 1784, 524, 71, 134, 386, 1718, 1907, 584, 1656, 1719, 1845, 1780, 1906, 646, 1598, 1724, 338, 133, 196, 448, 3, 66, 129, 318, 381, 570, 1011, 807, 1122, 1563, 1537, 1663, 277, 1779, 15, 708, 1233, 1422, 1926, 1657, 1720, 1846, 1596, 1785, 462, 1477, 1603, 91, 1900, 10, 892, 1048, 1111, 1174, 1237, 1300, 1489, 1804, 623, 812, 1442, 1538, 1790, 215, 989, 1178, 1682, 988, 1366, 1744, 1539, 1602, 153, 1597, 1912, 400, 1478, 1730, 29, 1901, 263, 830, 621, 684, 747, 873, 936, 999, 1503, 561, 813, 1380, 1416, 1479, 1668, 929, 1055, 1622, 1054, 1117, 1243, 1172, 1361, 1865, 1662, 1725, 1851, 1293, 1356, 1608, 1049, 1112, 1238, 437, 500, 563, 689, 752, 941, 1319, 376, 439, 628, 1354, 1417, 1543, 745, 808, 1060, 867, 930, 993, 987, 1050, 1428, 1110, 1173, 1299, 1231, 1294, 1483, 806, 869, 1184, 375, 438, 501, 627, 690, 879, 1257, 314, 377, 818, 1355, 1418, 1544, 683, 746, 935, 868, 931, 994, 1116, 1179, 1305, 622, 685, 874, 1232, 1295, 1484, 499, 562, 751
3	320, 446, 509, 698, 824, 1454, 1832, 568, 631, 820, 630, 693, 882, 384, 447, 636, 382, 508, 886, 14, 518, 1589, 13, 76, 328, 75, 138, 390, 135, 261, 702, 259, 322, 385, 574, 763, 1393, 1771, 506, 569, 1514, 1612, 1738, 352, 1847, 1910, 209, 197, 323, 1331, 143, 332, 1025, 1908, 81, 1530, 205, 394, 1087, 1850, 23, 1409, 73, 199, 262, 514, 640, 1270, 1711, 444, 759, 1893, 1059, 1374, 492, 1726, 1789, 844, 267, 456, 1149, 1485, 1548, 414, 19, 271, 964, 1911, 147, 1470, 1666, 1729, 1855, 11, 137, 200, 452, 578, 1208, 1649, 1426, 229, 922, 875, 1064, 245, 1664, 1916, 1286, 1241, 1304, 1493, 1303, 43, 799, 1728, 783, 1224, 1786, 1849, 85, 1604, 1667, 722, 1301, 1364, 1427, 1553, 1616, 167, 860, 1188, 1251, 1818, 939, 1758, 183, 1794, 1920, 660, 1118, 1370, 614, 1180, 1432, 676, 1424, 1487, 1676, 1788, 906, 1347, 1545, 1734, 1041, 995, 1058, 1121, 1247, 1436, 1940, 554, 878, 1130, 59, 691, 817, 1636, 1606, 598, 1165, 997, 1879, 430, 1056, 1182, 1497, 1239, 1365, 105, 1362, 1488, 291, 1672, 1798, 538, 814, 877, 940, 1003, 1192, 1696, 121, 753, 816, 1068, 629, 755, 1007, 1607, 1859, 1103, 998, 1313, 368, 1120, 1183, 1309, 1242, 1557, 738, 937, 1126, 307, 1547, 476, 980
4	1017, 1080, 1143, 1206, 1269, 1458, 1773, 1202, 1391, 1895, 1324, 1387, 1513, 1323, 1386, 1512, 1329, 1392, 1518, 1201, 1264, 1453, 1085, 1148, 1274, 958, 1147, 1651, 957, 1335, 1713, 590, 653, 716, 842, 905, 968, 1472, 1141, 1330, 1834, 1262, 1325, 1577, 37, 289, 982, 776, 1091, 1532, 1079, 1142, 1268, 836, 899, 962, 837, 900, 963, 898, 1024, 1591, 406, 469, 532, 658, 721, 910, 1288, 1018, 1081, 1207, 1200, 1263, 1452, 1689, 1752, 1878, 592, 781, 1411, 1023, 1086, 1212, 36, 99, 351, 775, 838, 1153, 714, 777, 1029, 344, 407, 470, 596, 659, 848, 1226, 956, 1019, 1397, 103, 166, 418, 1694, 1757, 1883, 530, 782, 1349, 1871, 1934, 233, 1933, 295, 862, 468, 531, 720, 652, 715, 904, 159, 222, 285, 474, 537, 726, 1167, 41, 104, 356, 1629, 1944, 432, 1569, 1695, 309, 345, 408, 597, 1811, 47, 740, 1810, 1873, 109, 227, 290, 542, 591, 654, 843, 97, 160, 223, 412, 475, 664, 1105, 1688, 1751, 1877, 1571, 1634, 185, 1510, 1762, 61, 283, 346, 787, 1690, 1816, 556, 1812, 1938, 678, 1932, 42, 924, 165, 228, 480, 35, 98, 161, 350, 413, 602, 1043, 1570, 1822, 247, 1509, 1635, 123, 1448, 1511, 1700, 221, 284, 536, 1628, 1817, 494, 1750, 1939, 616, 1872, 171, 801, 1630, 1756, 370
5	579, 642, 768, 831, 1146, 1398, 1587, 765, 891, 1647, 887, 1328, 1769, 946, 1072, 1891, 945, 1134, 1638, 825, 888, 951, 949, 1012, 1390, 707, 770, 1022, 580, 1084, 1336, 457, 520, 583, 709, 961, 1213, 1528, 86, 149, 401, 826, 1267, 1519, 1010, 1136, 1451, 1864, 1927, 226, 272, 461, 1154, 764, 1205, 1709, 395, 458, 647, 333, 396, 585, 210, 273, 399, 525, 903, 1092, 1407, 28, 154, 595, 703, 829, 1459, 1074, 1578, 1830, 1437, 1815, 366, 25, 214, 1222, 519, 645, 1275, 1800, 288, 603, 334, 523, 1468, 148, 211, 337, 463, 841, 1030, 1345, 92, 911, 1163, 641, 704, 893, 1678, 40, 481, 1253, 1379, 119, 152, 278, 719, 1745, 296, 674, 1618, 1681, 736, 90, 216, 657, 24, 87, 276, 339, 780, 969, 1284, 1860, 411, 978, 1805, 419, 797, 1314, 1755, 243, 1443, 1884, 57, 30, 849, 1101, 1559, 1937, 110, 1558, 1621, 172, 1739, 164, 920, 1677, 1740, 1803, 1866, 102, 543, 858, 1861, 349, 727, 1499, 1562, 428, 1193, 1256, 1823, 1069, 1132, 1258, 1921, 535, 1039, 1375, 1438, 1564, 1623, 48, 552, 1617, 1743, 357, 1252, 1315, 1441, 1504, 1693, 1945, 181, 1799, 1925, 665, 1318, 1381, 1633, 1131, 1194, 1320, 1070, 1574, 1763, 1922, 473, 788, 1376, 1502, 305, 1498, 1876, 490, 1683, 234, 612

Continued on next page

Table 2 – continued from previous page

memory block No.	Edge Numbers and the order in which they are stored
6	391, 706, 832, 1210, 1273, 1462, 1525, 453, 894, 1272, 828, 1395, 1584, 1013, 1517, 1706, 883, 1009, 1576, 1071, 1575, 1827, 699, 1581, 1644, 760, 1075, 1516, 329, 644, 1211, 144, 522, 648, 1089, 1152, 1278, 1404, 268, 1339, 1465, 275, 464, 1157, 637, 952, 1456, 821, 1703, 1766, 478, 541, 730, 20, 398, 1028, 575, 890, 1457, 206, 710, 1151, 82, 460, 586, 1027, 1090, 1216, 1342, 336, 966, 1281, 217, 847, 1036, 1333, 1396, 1522, 948, 1137, 1641, 1248, 51, 240, 151, 340, 1033, 582, 771, 1401, 1863, 540, 855, 1917, 27, 279, 846, 909, 972, 1098, 526, 967, 1219, 1856, 155, 785, 515, 767, 1334, 1867, 355, 733, 1127, 1505, 1820, 89, 908, 1160, 107, 170, 422, 169, 232, 484, 1613, 1802, 1928, 416, 479, 668, 794, 213, 402, 1095, 600, 663, 852, 1742, 293, 671, 1440, 1881, 54, 1065, 1317, 1821, 93, 786, 975, 1433, 299, 425, 1684, 46, 487, 1554, 1680, 1806, 231, 294, 546, 609, 354, 417, 606, 1735, 1924, 601, 1310, 113, 302, 1004, 1382, 1760, 1321, 1699, 1951, 1795, 31, 724, 1501, 1942, 178, 1371, 237, 363, 1494, 1620, 1746, 45, 108, 360, 549, 1189, 1378, 1882, 1673, 791, 917, 1255, 1444, 1948, 1698, 1761, 1887, 1196, 1259, 1826, 662, 725, 914, 1565, 1943, 116, 1561, 1624, 175
7	266, 392, 1337, 1400, 1463, 1526, 1652, 454, 1399, 1714, 327, 516, 1461, 513, 576, 1521, 635, 761, 1580, 1639, 1702, 1828, 819, 1701, 1764, 573, 1707, 1896, 697, 1579, 1642, 80, 269, 1214, 1277, 1340, 1466, 1592, 18, 207, 1215, 142, 1276, 1402, 1031, 1094, 1220, 700, 1582, 1645, 758, 884, 1640, 604, 667, 856, 83, 1217, 1343, 638, 1520, 1835, 1611, 1674, 666, 729, 792, 918, 1044, 145, 1279, 1405, 21, 1155, 1533, 1918, 973, 1099, 451, 1585, 1774, 822, 1704, 1767, 1311, 114, 303, 1915, 970, 1096, 204, 330, 1338, 1492, 1555, 358, 421, 547, 610, 925, 1854, 1161, 1350, 1093, 1156, 1282, 1793, 1037, 1289, 389, 1460, 1523, 1552, 544, 607, 1190, 1946, 371, 971, 1034, 1412, 359, 485, 863, 1369, 1495, 235, 298, 361, 550, 802, 605, 731, 983, 1032, 1158, 1473, 1671, 789, 915, 482, 545, 734, 1125, 117, 495, 1002, 1128, 1947, 1857, 912, 1227, 173, 236, 488, 1308, 1434, 111, 174, 300, 426, 741, 420, 483, 672, 1614, 669, 795, 790, 853, 1168, 50, 176, 617, 1886, 1949, 248, 1825, 1888, 124, 850, 913, 976, 1249, 52, 241, 1246, 1372, 49, 112, 238, 364, 679, 1431, 297, 423, 1063, 55, 433, 1736, 728, 1106, 1066, 1885, 310, 1005, 1824, 186, 944, 1952, 62, 1733, 1796, 851, 1187, 179, 557
8	256, 571, 634, 823, 1705, 1768, 1894, 140, 203, 455, 202, 265, 517, 264, 1524, 1650, 387, 450, 639, 194, 572, 1643, 379, 757, 1765, 441, 756, 1890, 132, 762, 1833, 8, 449, 512, 701, 1583, 1646, 1772, 1844, 17, 1403, 270, 1341, 1467, 16, 79, 331, 1661, 1913, 1283, 70, 511, 1708, 317, 695, 1829, 1675, 793, 919, 146, 1280, 1406, 1905, 78, 141, 393, 1464, 1527, 1590, 1359, 1737, 981, 1783, 208, 1531, 84, 1218, 1344, 1792, 1162, 1225, 325, 388, 577, 633, 696, 885, 1185, 177, 555, 1852, 1159, 1348, 1115, 1367, 1430, 1556, 548, 611, 800, 1177, 1429, 673, 1791, 1035, 1287, 1723, 22, 1471, 1919, 974, 1100, 326, 1586, 1712, 1615, 670, 796, 1001, 56, 308, 1601, 1853, 1097, 992, 1244, 1307, 1496, 362, 551, 677, 1306, 424, 739, 1298, 1550, 857, 1914, 1221, 1410, 1482, 1797, 1104, 1490, 608, 923, 1062, 180, 369, 750, 1191, 246, 1542, 1731, 1038, 934, 1123, 1186, 1312, 115, 304, 493, 1245, 489, 615, 1491, 735, 861, 1236, 1551, 732, 1609, 916, 1042, 1373, 239, 365, 626, 1067, 122, 880, 943, 1006, 1669, 1732, 1858, 872, 1061, 1124, 1250, 53, 242, 431, 1435, 301, 427, 1053, 1368, 486, 811, 1000, 118, 1421, 1610, 854, 688, 1129, 184, 942, 1950, 60, 566, 881, 1889, 1670, 977, 1166
9	63, 64, 65, 66, 67, 69, 70, 71, 72, 75, 76, 77, 79, 81, 82, 87, 89, 90, 91, 95, 96, 98, 99, 101, 104, 108, 111, 112, 115, 117, 119, 64, 65, 66, 67, 68, 70, 71, 72, 73, 76, 77, 78, 80, 82, 83, 88, 90, 91, 92, 96, 97, 99, 100, 102, 105, 109, 112, 113, 116, 118, 120, 65, 66, 67, 68, 69, 71, 72, 73, 74, 77, 78, 79, 81, 83, 84, 89, 91, 92, 93, 97, 98, 100, 101, 103, 106, 110, 113, 114, 117, 119, 121, 66, 67, 68, 69, 70, 72, 73, 74, 75, 78, 79, 80, 82, 84, 85, 90, 92, 93, 94, 98, 99, 101, 102, 104, 107, 111, 114, 115, 118, 120, 122, 67, 68, 69, 70, 71, 73, 74, 75, 76, 79, 80, 81, 83, 85, 86, 91, 93, 94, 95, 99, 100, 102, 103, 105, 108, 112, 115, 116, 119, 121, 123, 68, 69, 70, 71, 72, 74, 75, 76, 77, 80, 81, 82, 84, 86, 87, 92, 94, 95, 96, 100, 101, 103, 104, 106, 109, 113, 116, 117, 120, 122, 124, 69, 70, 71, 72, 73, 75, 76, 77, 78, 81, 82, 83, 85, 87, 88, 93, 95, 96, 97, 101, 102, 104, 105, 107, 110, 114, 117, 118, 121, 123, 125

A.3 Generating the points of $\mathbb{PG}(5, GF(2))$

The table gives the points of $GF(2^6)$ generated via the primitive polynomial $x^6 + x + 1$.

Index	1-D subspace
0	{0,1}
1	$\{0,\ x^1\}$
2	$\{0,\ x^2\}$
3	$\{0,\ x^3\}$
4	$\{0,\ x^4\}$
5	$\{0,\ x^5\}$
6	$\{0,\ x^1+1\}$
7	$\{0,\ x^2+x\}$
8	$\{0,\ x^3+x^2\}$
9	$\{0,\ x^4+x^3\}$
10	$\{0,\ x^5+x^4\}$
11	$\{0,\ x^5+x^1+1\}$
12	$\{0,\ x^2+1\}$
13	$\{0,\ x^3+x\}$
14	$\{0,\ x^4+x^2\}$
15	$\{0,\ x^5+x^3\}$
16	$\{0,\ x^4+x^1+1\}$
17	$\{0,\ x^5+x^3+x^2\}$
18	$\{0,\ x^3+x^2+x+1\}$
19	$\{0,\ x^4+x^3+x^2+x\}$
20	$\{0,\ x^5+x^4+x^3+x^2\}$
21	$\{0,\ x^5+x^4+x^3+x+1\}$
22	$\{0,\ x^5+x^4+x^2+1\}$
23	$\{0,\ x^5+x^3+1\}$
24	$\{0,\ x^4+1\}$
25	$\{0,\ x^5+x^1\}$
26	$\{0,\ x^2+x^1+1\}$
27	$\{0,\ x^3+x^2+x^1\}$
28	$\{0,\ x^4+x^3+x^2\}$
29	$\{0,\ x^5+x^4+x^3\}$
30	$\{0,\ x^5+x^4+x+1\}$
31	$\{0,\ x^5+x^2+1\}$

Index	1-D subspace
32	$\{0,\ x^3+1\}$
33	$\{0,\ x^4+x\}$
34	$\{0,\ x^5+x^2\}$
35	$\{0,\ x^3+x+1\}$
36	$\{0,\ x^4+x^2+x\}$
37	$\{0,\ x^5+x^3+x^2\}$
38	$\{0,\ x^4+x^3+x+1\}$
39	$\{0,\ x^5+x^4+x^2+x\}$
40	$\{0,\ x^5+x^3+x^2+x+1\}$
41	$\{0,\ x^4+x^3+x^2+1\}$
42	$\{0,\ x^5+x^4+x^3+x\}$
43	$\{0,\ x^5+x^4+x^2+x+1\}$
44	$\{0,\ x^5+x^3+x^2+1\}$
45	$\{0,\ x^4+x^3+1\}$
46	$\{0,\ x^5+x^4+x\}$
47	$\{0,\ x^5+x^2+x+1\}$
48	$\{0,\ x^3+x^2+1\}$
49	$\{0,\ x^4+x^3+x\}$
50	$\{0,\ x^5+x^4+x^2\}$
51	$\{0,\ x^5+x^3+x+1\}$
52	$\{0,\ x^4+x^2+x\}$
53	$\{0,\ x^5+x^3+x\}$
54	$\{0,\ x^4+x^2+x+1\}$
55	$\{0,\ x^5+x^3+x^2+x\}$
56	$\{0,\ x^4+x^3+x^2+x+1\}$
57	$\{0,\ x^5+x^4+x^3+x^2+x\}$
58	$\{0,\ x^5+x^4+x^3+x^2+x+1\}$
59	$\{0,\ x^5+x^4+x^3+x^2+1\}$
60	$\{0,\ x^5+x^4+x^3+1\}$
61	$\{0,\ x^5+x^4+1\}$
62	$\{0,\ x^5+1\}$

Table 3: Points of $\mathbb{PG}(5, \mathbb{GF}(2))$

References

[1] Noga Alon. Eigenvalues, geometric expanders, sorting in rounds, and Ramsey theory. *Combinatorica*, 6(3):207–219, 1986.

[2] John B. Anderson and Mohan Seshadri. *Source and Channel Coding: An Algorithmic Approach*. Kluwer Academic Publishers, 1991.

[3] Alexander Barg and Gilles Zemor. Error Exponents of Expander Codes. *IEEE Transactions on Information Theory*, 48(6):1725–1729, 2002.

[4] Yeow Meng Chee and San Ling. Highly Symmetric Expanders. *Finite Fields and Their Applications*, 8(3):294 – 310, 2002.

[5] Tom Hoholdt and Heeralal Janwa. Optimal Bipartite Ramanujan Graphs from Balanced Incomplete Block Designs: Their Characterizations and Applications to Expander/LDPC Codes. *International Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 53–64, 2009.

[6] Tom Hoholdt and Jorn Justensen. Graph Codes with Reed-Solomon Component Codes. *International Symposium on Information Theory*, pages 2022–2026, 2006.

[7] http://www.ecma-international.org/. *Standard ECMA-130: Data Interchange on Read-only 120 mm Optical Data Disks (CD-ROM)*, 1996.

[8] http://www.ecma-international.org/. *Standard ECMA-267: 120 mm DVD - Read-Only Disk*, 2001.

[9] Xilinx Inc. Reed-Solomon Decoder v7.0 Datasheet. `http://www.xilinx.com/support/documentation/ip_documentation/rs_decoder.pdf`, 2009.

[10] ISO,International Organization for Standardization, and IEC, International Electrotechnical Commission. *ISO/IEC 23912:2005, Information technology 80 mm (1,46 Gbytes per side) and 120 mm (4,70 Gbytes per side) DVD Recordable Disk (DVD-R)*, 2005.

[11] Narendra Karmarkar. A New parallel architecture for sparse matrix computation based on finite projective geometries. *Proceedings of Supercomputing*, 1991.

[12] Michael Sipser and Daniel Spielman. Expander Codes. *IEEE Transactions on Information Theory*, 42(6):1710–1722, 1996.

[13] Wikipedia. Disc Storage Devices. `http://en.wikipedia.org/wiki/Disk_storage`, 2010.

[14] Gilles Zemor. On Expander Codes. *IEEE Transactions on Information Theory*, 47(2):835–837, 2001.