INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

DISSERTATION

THREE–DIMENSIONAL ERROR CORRECTING CODES FOR VOLUME HOLOGRAPHIC DATA STORAGE

Submitted by

Terry N. Garrett

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, CO

Spring 2000

COLORADO STATE UNIVERSITY

March 2, 2000

We hereby recommend that the Dissertation prepared under our supervision by Terry N. Garrett entitled "Three–Dimensional Error Correcting Codes for Volume Holographic Data Storage" be accepted as fulfilling in part the requirements for the Degree of Doctor of Philosophy.

Committee on Graduate Work

J. K. Mabry

Pericles A. Mitkas
Adviser

Department Head
Committee Member

ABSTRACT OF DISSERTATION

THREE–DIMENSIONAL ERROR CORRECTING CODES FOR VOLUME HOLOGRAPHIC DATA STORAGE

Volume holographic data storage represents a potential technology path to meet the storage intensive application needs of the information age. Important features of these systems are large storage capacity and fast transfer rate. These storage systems store and retrieve data as two–dimensional data pages and are also referred to as page oriented memories. Conventional storage systems such as magnetic and optical disks store and retrieve data serially. Noise sources encountered in holographic data storage include cross talk, optical and electrical noise sources. During data storage and retrieval, these noise sources interact to produce several error types such as random single bit errors, burst errors and cluster errors. Two–dimensional and three–dimensional error correcting codes have been proposed to perform the data encoding and decoding required by page oriented memories.

In this dissertation, we evaluate three–dimensional error correcting codes based on array codes for volume holographic data storage. This three–dimensional encoding scheme, also called z–encoding, provides error detection and correction for error events which may span multiple pages. The attributes of two–dimensional array codes are reviewed and adapted for use as three–dimensional array codes. These codes are analyzed regarding hardware requirements, timing delay and code performance. The row and column array

code is found to be effective on random single bit errors. This code is adapted for use as a burst error correcting code and adapted again for use as a cluster error correcting code.

Interactions between array code attributes, storage capacity and decode time are analyzed. The attributes of array codes which allow it to be used either as an equal error protection code or an unequal error protection code are evaluated. An algorithm for converting an equal error protection code to an unequal error protection code is described.

Terry Nathaniel Garrett
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, CO 80523
Spring 2000

ACKNOWLEDGEMENT

This dissertation and all that it contains would not exist without the help of others whom I now wish to acknowledge and thank.

Thank you and praise you, God, without whom this project would not have been possible.

I gratefully acknowledge generous financial support from StorageTek Corporation and the Colorado Advanced Technology Institute.

I gratefully acknowledge my advisor, Dr. Pericles A. Mitkas, who provided the opportunity and served as a motivating force which allowed me to do some of my best work at times exceeding my own expectations. Moreover, the valuable contributions of my graduate committee members are appreciated and acknowledged.

For the prayers, spiritual and mental support, and loving me always as their son, thank you Sidney and Alice Garrett as well as Mack and Fern Campbell. For the most wonderful helpmate a man could ever have, thank you, my wife, Saundra. To my gifted children, Stephanie and Terence, my daughter and son, thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

1.1 Memory Hierarchies

The colossal need for and growth in information storage has been driven by information age demands and applications. To meet these demands, storage systems are required with large capacity, fast readout or transfer rates, short access times and low bit error rates. Requirements for storage intensive applications such as multimedia, video on demand, network–based computing and databases are expected to exceed 10^{20} bits (12 exabytes) in the year 2000 [MAN97]. Hierarchies of on–line, near–line and off–line storage systems will be required using diverse technologies: semiconductor random access memories, magnetic/optical disk drives and disk libraries, magnetic/optical tape drives and tape libraries.

A likely application for volume holographic data storage (VHDS) systems is as a mass storage device in the memory storage hierarchy [CHE97, PAR90, RED89] of a

computer system. Since different types of memory storage devices make up the memory storage hierarchy, these memory storage systems must be configured to provide the required attributes of storage capacity (e.g., $\geq 10^{12}$ bits), readout or transfer rate (e.g., $\geq 10^{9}$ bits/sec), access time (e.g., $\leq 40 \times 10^{-6}$ sec) and raw bit error rate (*RBER*), e.g., $\leq 10^{-4}$. A typical computer memory hierarchy includes dynamic random access memory (DRAM) devices, magnetic/optical disk and magnetic/optical tape memory storage systems.

The main memory consists of semiconductor random access memory (RAM) devices or memory cards containing arrays of DRAM devices and provides fast access to memory. Main memory is near the top or primary level of the memory storage hierarchy. Registers and cache memory are higher than main memory and provide the fastest access to memory. Storage capacity, transfer rate, access time and *RBER* are the four main parameters used to measure performance.

Magnetic/Optical disks and mass storage disk systems make up the secondary level of the memory storage hierarchy. At this level, storage capacity, throughput, access time and *RBER* are the four parameters used to measure performance. The recording density for disks continues to improve such that disk capacity has doubled every three to four years [CHE97].

A memory gap [CHE97, PAR90, RED89] exists between the primary and secondary storage levels of the memory storage hierarchy. As shown in Figure 1.1, access times can differ by over five orders of magnitude between DRAM and magnetic/optical disk systems. When acousto–optic or electro–optic deflectors steer the read and write laser beams used in VHDS systems, access times ranging from 5 μs to less than 40 μs [MCM96, PSA92, RED92] are feasible. These access times fall between the access times for DRAM devices and magnetic/optical disk systems. The memory gap for storage capacity ranges from a main memory capacity value of 256 MB to the capacity of mass storage disk systems where each

disk in a multiple disk system has greater than a 3 GB storage capacity [MAN97]. The optical digital versatile disk (DVD), also called the digital video disk, can store 4.7 GB and there is a technology path for increasing this value to 17 GB [MCL98]. When VHDS systems use angular multiplexing to store thousands of two–dimensional (2–D) holographic pages or holograms in a stack and the stacks are placed in an array of hundreds of spatial locations, storage capacity ranging from tens of gigabytes to hundreds of gigabytes becomes feasible [MOK92, PSA95, PSA98, RED92].

Magnetic/Optical tape and mass storage tape systems make up the lowest or tertiary level of the memory storage hierarchy. Again, at this level, storage capacity and access time are the main two parameters used to measure performance. The third and fourth parameters, throughput or data rate and *RBER*, are more a function of the overall memory design than the particular device. In addition, the overall volume of the memory system becomes an important issue.

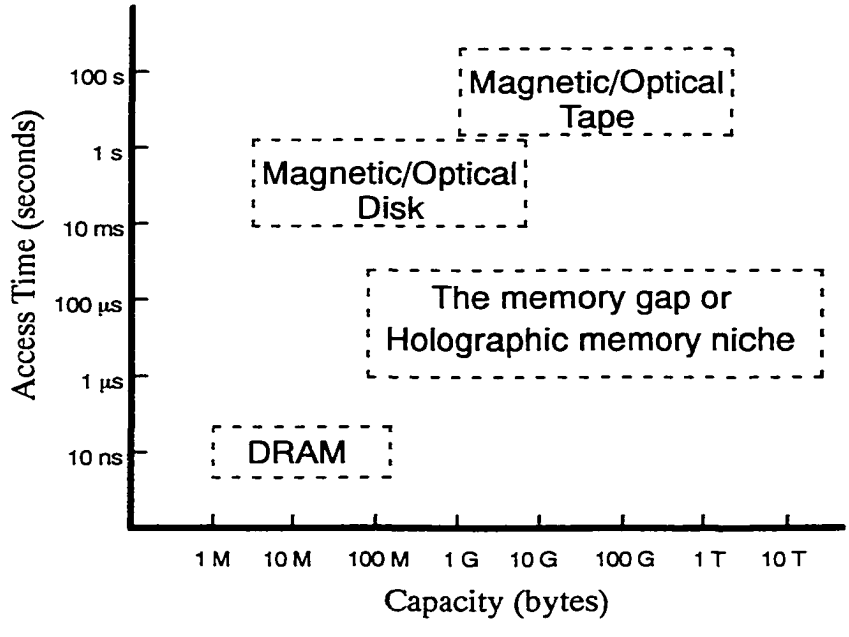


Figure 1.1: Relationship between access time and storage capacity.

1.2 Statement of the Problem

Advanced VHDS systems store and retrieve data in a 2–D format as pages or bit arrays. These storage systems are also referred to as page–oriented memories (POMs). Conventional storage systems (e.g., magnetic disks and tapes, optical disks and tapes) store and retrieve data in a serial (e.g., bit serial) fashion. Hence, POMs and conventional storage systems use markedly different data access paradigms. POMs require 2–D error detection and correction (EDC) techniques to reduce the *RBER* to acceptable levels [GOE95, HEA94].

Effective data encoding must provide EDC to adequately address problems and issues associated with digital data stored on or recalled from holographic pages. When the error event is one or more corrupted pages, data encoding using array codes can be used to address this error event. Array codes [FAR92], based mostly on exclusive–OR (XOR) gate operations, are fast and easy to implement [BLA96]. Unequal error protection (UEP) codes offer yet another approach to encoding a data page. Classes of equal error protection (EEP) array codes (e.g., cross parity check and modified cross parity check) [GOE95, GAR98] and a class of UEP array codes [PIN88, GAR99] are the most likely candidate coding techniques which can be extended to span multiple pages (i.e., z–encoding). Alternatively, Reed–Solomon codes utilize bytes corresponding to a code word distributed among different pages. After the pixels in each page are divided into bytes, encoding and decoding algorithms are performed as operations over a finite field. This approach usually leads to more complex hardware and will not be pursued as a candidate coding technique which can be extended in a z–encoding fashion.

Data encoding techniques and error control processes [GOE95] must meet several criteria to enhance and complement the storage capacity, transfer rate and access time features expected for POMs.

a) The encoding must yield codes with as high a rate as possible to minimize storage overhead and maximize error correcting capability.

b) The encoding and decoding algorithms must be fast and simple to minimize the complexity of encoding and decoding hardware.

c) The encoding must operate on data pages as large as 10^6 bits in a 2–D format as the preferred mode.

d) The encoding should be able to correct effectively a specific error type.

e) The encoding must allow corrected bit error rates lower than 10^{-12} to be obtained using an appropriate error correcting code (ECC).

The purpose of this Dissertation is to evaluate the potential of ECCs that span multiple pages and when certain error types and Gaussian noise sources occur in a VHDS [GOE95, HEA94, PSA98, WUL94] system with 2–D parallel output.

1.2.1 Noise and Error Sources

In a VHDS system, storage and retrieval of information can be modeled as data transmission across a noisy channel. Noise can be considered to be any unwanted variation that affects the performance of a given system adversely. Data communication and storage channels are subjected to noise sources. These noise sources may originate from electronic (e.g., thermal noise from the detector) and optical (e.g., light scatter) subsystems and the physical channel that separates the transmitter and receiver of information. The effect of a noisy channel on a system must be reduced to achieve a lower error rate. In principle, there are several ways in which this can be accomplished using direct and indirect techniques.

The goal of direct techniques is to reduce the relative magnitude of noise using one or a combination of techniques. For example, one may try to improve the noisy channel

characteristics or signal–to–noise–ratio(SNR), or try to find a less noisy channel, or reduce the data transfer rate of the system.

Indirect techniques add redundant information to the data in an effort to detect and correct errors that occur during storage and retrieval. These techniques are forms of coding for error control and use ECCs.

1.2.2 Small–Scale Noise and Error Sources

Small–scale noise is considered to occur when the dominant noise is additive Gaussian noise in field amplitude that is independent from pixel to pixel and the SNR shows an inverse dependence on the square of the number of holograms stored [HEA95]. This type of noise is predominantly due to scattering of the reference beam from a collection of independent scattering centers and the detector pixel size being much greater than the impulse response of the imaging optics [HEA95].

When small–scale noise occurs, binary threshold encoding can be used advantageously. This encoding associates one or more pixels with a single data bit. A pixel is switched to the on–state to represent a one or to the off–state to represent a zero. At the detector, a threshold intensity setting is selected to distinguish on–pixels from off–pixels. The threshold setting must be selected such that it minimizes the total probability of error.

1.2.3 Large–Scale Noise and Error Sources

Noise may occur on a large–scale basis due to an improper exposure schedule or to a defective storage medium. For example, use of an incorrect recording schedule can lead to significant diffraction efficiency variation from hologram to hologram. An unpredictable change in the recording environment can also lead to unacceptable diffraction efficiency variation from hologram to hologram [HEA95]. A hologram may reconstruct nonuniformly and lead to a SNR that varies across the hologram.

Large–scale noise complicates selecting an appropriate thresholding scheme. For example, large–scale noise can lead to errors because a multiregional or multihologram (i.e., nonlocal) threshold setting is used when a regional (i.e., local) threshold setting may be required. A dynamic thresholding scheme may be required to determine the threshold setting such that the threshold value is measured and set during retrieval.

1.3 Objectives of the Dissertation

This Dissertation presents work completed toward the evaluation of the potential of three–dimensional (3–D) ECCs for VHDS systems. We have given several reasons behind the selection of holographic technology for storage intensive applications, and these are summarized below.

- Large capacity is provided by storing multiplexed 2–D holographic pages into stacks and storing multiple stacks into different spatial locations.

- High data rates are provided by parallel access to 2–D holographic pages during store and recall operations.

- Acousto–optic and electro–optic deflectors are used to steer read and write laser beams providing memory access times between those of DRAM devices and magnetic/optical disk systems.

The objective, then, is to evaluate the suitability of 3–D ECCs to VHDS systems. To meet this objective, four major tasks were defined.

1. Analyze the characteristics of the holographic storage channel (e.g., noise sources, error types and ECC performance).
2. Evaluate data encoding methods which utilize 3–D ECCs for a VHDS system.

3. Analyze the potential performance gains from the application of 3–D ECCs to a VHDS system.

4. Analyze the complexity of the encoding and decoding hardware of selected 3–D ECCs.

1.4 Contents of the Dissertation

The system issues concerning VHDS systems are discussed in Chapter 2. The major components and features of a VHDS system are described. Next, the impact of various noise sources on the system are considered.

Consideration is confined to 2–D block codes and in Chapter 3 a background is given of the basic properties of these codes. Bounds on code length, number of information bits and code distance are described. The relationship of code distance to error detection and correction capability of these codes is explained.

Chapter 4 describes the parameter, properties and characteristics of 3–D array codes. The performance of the codes are described for several code classes. An analysis is performed to provide some insight on the hardware complexity and associated time delay of the encoders and decoders for these codes.

Chapter 5 describes the parameters and properties of a 3–D UEP array code by drawing on the techniques discussed in earlier chapters and adding to them. The hardware required and time delay of the encoder and decoder for this code are analyzed. The performance of this code is described. An EEP to UEP code conversion algorithm is discussed.

In Chapter 6, the impact of 3–D ECCs on the performance of VHDS systems is evaluated. Performance issues involving storage capacity and data readout or transfer rate

are discussed. Issues associated with a data encoding scheme to both modulate and encode data are described.

The last chapter contains conclusions and suggests future research directions.

Chapter 2
System Issues

In this chapter, we introduce the reader to the system issues involved in VHDS. First, the recording and reconstructing of volume holograms are described. Next, the impact of various noise sources on the system is considered. Then, we consider the interaction of noise and *RBER*. Finally, we discuss the error types produced by various noise sources acting collectively.

2.1 Recording of Volume Holograms

The components required to record a volume hologram are shown in Figure 2.1. The laser provides a coherent light source which is split into reference and signal beams using a beam splitter (not shown). An electrically addressed spatial light modulator (SLM) is used to convert the image beam into a page of digital information, (i.e., a 2–D array of on–state and off–state pixels). A lens is used to generate a spatial Fourier transform [GOO96, HEC98, MCA91] of the page which interferes with an angularly [MOK93, PSA98] encoded reference beam to produce an interference pattern or grating. The interference pattern is stored as a volume hologram in the storage medium.

In a more detailed description, a complex object wave and a plane reference wave having wavelength, λ, are incident on a storage medium (e.g., a photorefractive crystal). Assume the object wavevector, oriented in the direction of the laser beam, is incident at a face normal of the storage medium and the reference wavevector is inclined at angle θ to the object wavevector. Planes (i.e., planes of zero phase) are used to show the wavefronts in the storage medium. Where the wavefronts intersect, the amplitudes of the object and reference waves add in phase (i.e., constructively interfere) to produce an intensity interference pattern. As the wavefronts propagate in the direction of their respective wavevectors, the planes of constructive interference trace out or form a grating. The grating bisects the angle θ between the two wavevectors. If a photorefractive crystal [FUR97, MOK91, YEH93] is used as the storage medium, the grating is represented by a modulation of the dielectric constant.

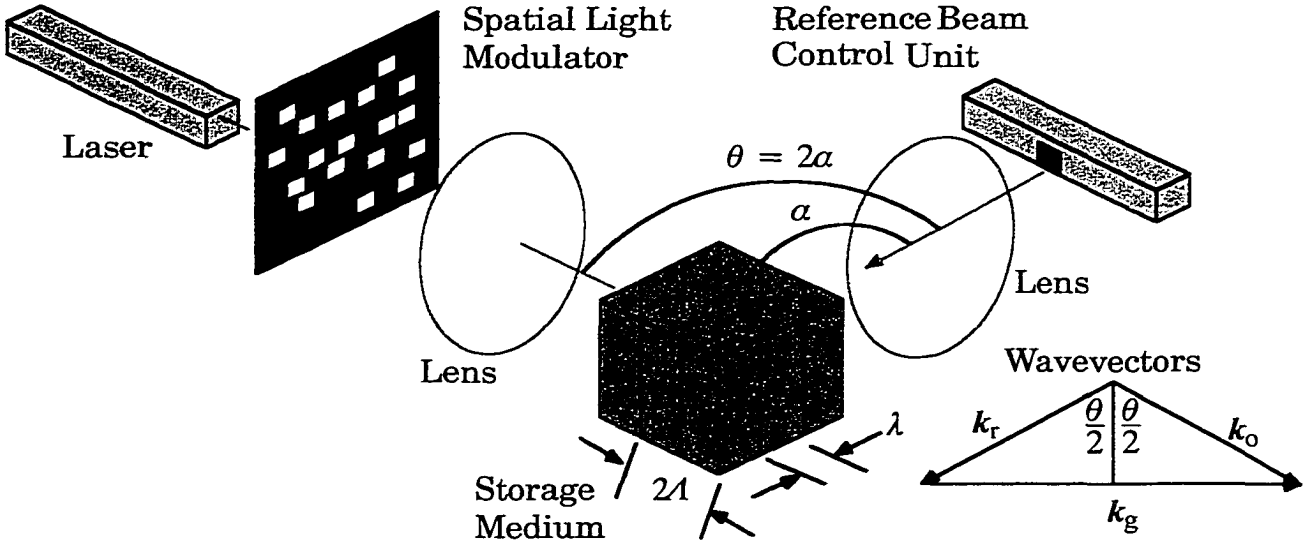


Figure 2.1: Recording of volume hologram and diagram of wavevectors.

The planar object and reference waves can be expressed as

$$U_o(\boldsymbol{r}) = \sqrt{I_o}\exp(-j\boldsymbol{k}_o \cdot \boldsymbol{r}) \qquad (2.1)$$

and

$$U_r(r) = \sqrt{I_r}\exp(-j\boldsymbol{k}_r \cdot \boldsymbol{r}) \tag{2.2}$$

where $\boldsymbol{k}_o$ is the wavevector of the object wave, and $\boldsymbol{k}_r$ is the wavevector of the reference wave and $\boldsymbol{r}$ is a position vector with elements (x,y,z). The intensity distribution of the interference pattern produced by superimposing these waves can be expressed as

$$I(\boldsymbol{r}) = |U_o + U_r|^2 = I_o + I_r + 2\sqrt{I_o I_r}\cos \boldsymbol{k}_g \cdot \boldsymbol{r} \tag{2.3}$$

where $\boldsymbol{k}_g = \boldsymbol{k}_o - \boldsymbol{k}_r$ is the grating vector. The grating vector has a sinusoidal pattern with period $\Lambda = \frac{2\pi}{|\boldsymbol{k}_g|}$ and surfaces of constant intensity normal to $\boldsymbol{k}_g$. The grating period can be expressed in terms of λ and θ by recalling that $|k_o| = |k_r| = \frac{2\pi}{\lambda}$. In Figure 2.1, the wavevectors show the direction of the grating vector and we can deduce that

$$\sin\frac{\theta}{2} = \frac{\left|\frac{\boldsymbol{k}_g}{2}\right|}{|\boldsymbol{k}_r|}. \tag{2.4}$$

Equation 2.4 can be rearranged and expressed as

$$\Lambda = \frac{\lambda}{2\sin\frac{\theta}{2}}. \tag{2.5}$$

2.2 Reconstruction of Hologram from Volume Grating

The components required to reconstruct a stored volume hologram are shown in Figure 2.2. The storage medium is illuminated with a reference beam oriented at the same angle as was used to record the hologram (i.e., the stored interference pattern). Specifically, to reconstruct the original object plane wave the volume grating must be illuminated with the reference plane wave incident at angle α. The reflected wave travels in the direction satisfying the law of reflection. The reflections occur at multiple layers of the grating. For the reflected plane waves to add in phase, the path length traveled by the reflected waves

must differ by a single optical wavelength, which corresponds to the first order diffracted wave; this condition is satisfied when the angle of incidence satisfies the Bragg condition (i.e., Bragg reflected or Bragg matched) [GOO96, SAL91] given by $\sin\alpha = \frac{\lambda}{2\Lambda}$. A lens is used to image the reconstructed hologram onto a detector array.

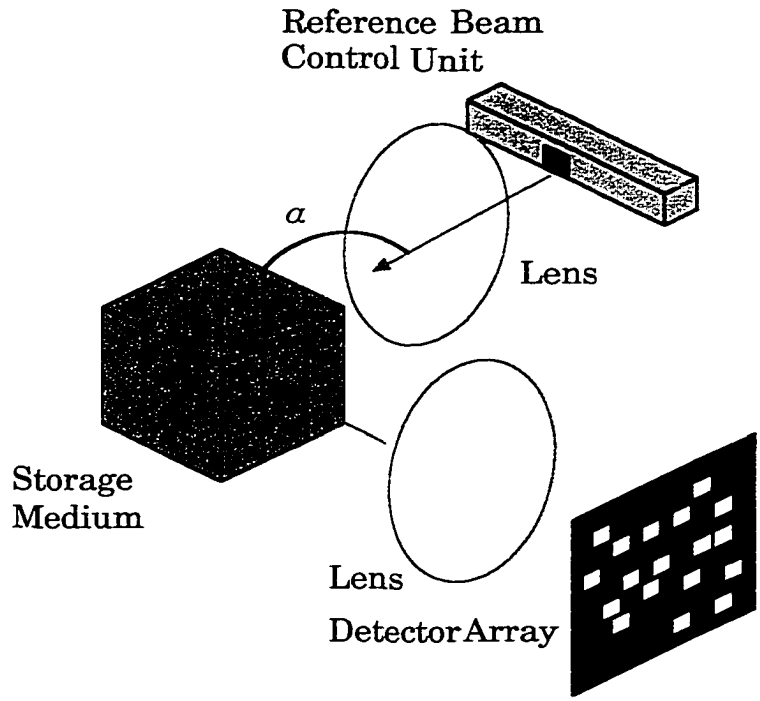


Figure 2.2: Bragg reflection of reference wave producing object.

2.3 A Typical Volume Holographic Data Storage System

Since a VHDS system must perform store and retrieve operations involving several steps, it must combine the components in Figures 2.1 and 2.2. A typical VHDS system is shown in Figure 2.3. In the store or write operation, an image beam containing a page of digital information and a reference beam oriented at a unique angle are used to record an interference pattern (i.e., a Fourier plane hologram) in the photorefractive storage medium; lens 1 and lens 2 Fourier transform the incident beams into plane waves. This procedure or

operation is repeated using different sets of image and reference beams to store additional pages in the same volume of the storage medium. During the retrieval or read operation, the storage medium is illuminated with a reference beam oriented at the same angle as was used to record the page of digital information. The retrieved page is detected or received by a photodetector array and converted to another format (e.g., electronic) for additional processing.

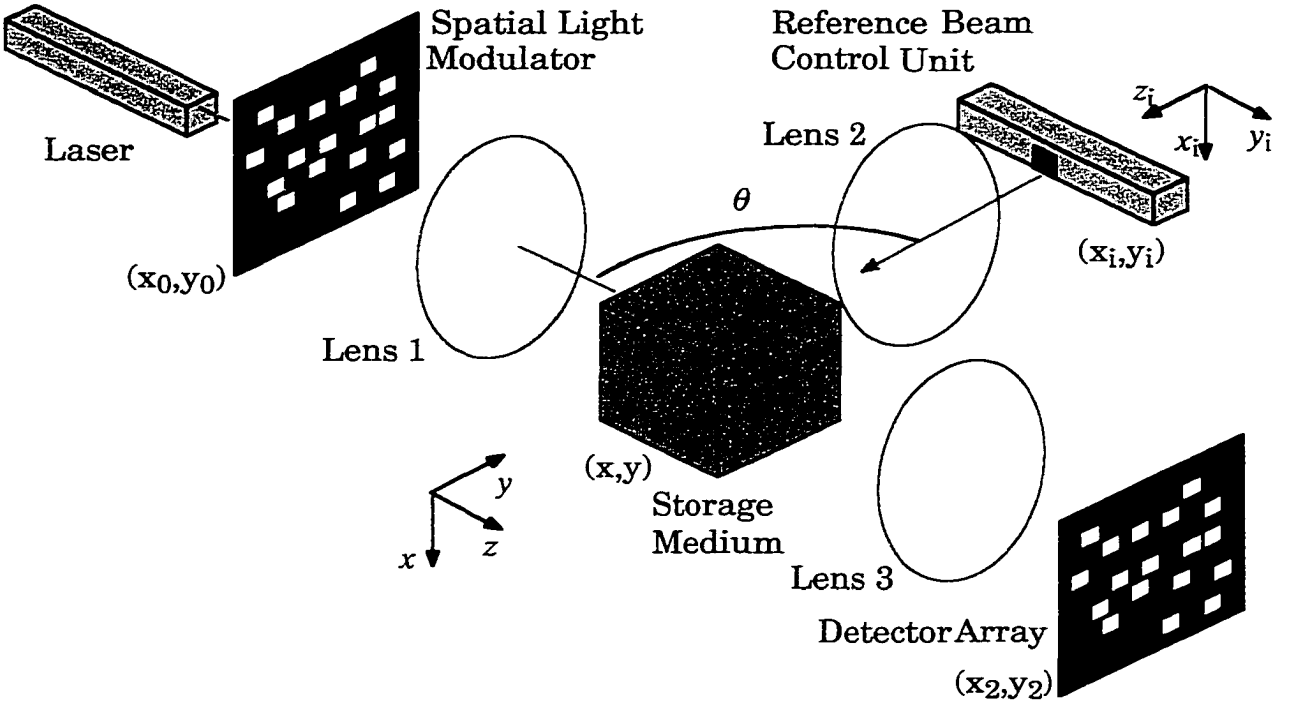


Figure 2.3: An angular multiplexed volume holographic storage system.

2.4 Holographic Storage Channel

The *RBER* is an important feature of a VHDS system. By analyzing the statistics of the noise sources encountered in a VHDS system, the *RBER* can be determined. These noise sources include cross talk noise, scatter noise and detector noise (e.g., thermal, flicker and shot noise sources). Rician statistics have been used to model cross talk noise and scattering noise [GU96, HEA94]; these noise sources are forms of optical noise with a coherent

superposition of random complex–valued amplitudes. Gaussian statistics have been used to model detector noise [AGR97]. Rician and Gaussian statistics are required when combinations of optical and electrical noise are present.

2.4.1 Cross Talk Noise Source

When the i^{th} page or hologram (i.e., m_i) is retrieved, the storage medium is illuminated with a reference beam oriented at the same angle as was used to record the i^{th} page. Since multiple pages are recorded in the same volume of the storage medium, pages other than the i^{th} page (i.e., the desired page) recorded with different reference beams are partially retrieved at greatly weakened intensity and interfere with the retrieval of the desired page. The interfering pages cause undesired diffraction of the incident light available to reconstruct the desired page and have the effect of generating a noise source called cross talk.

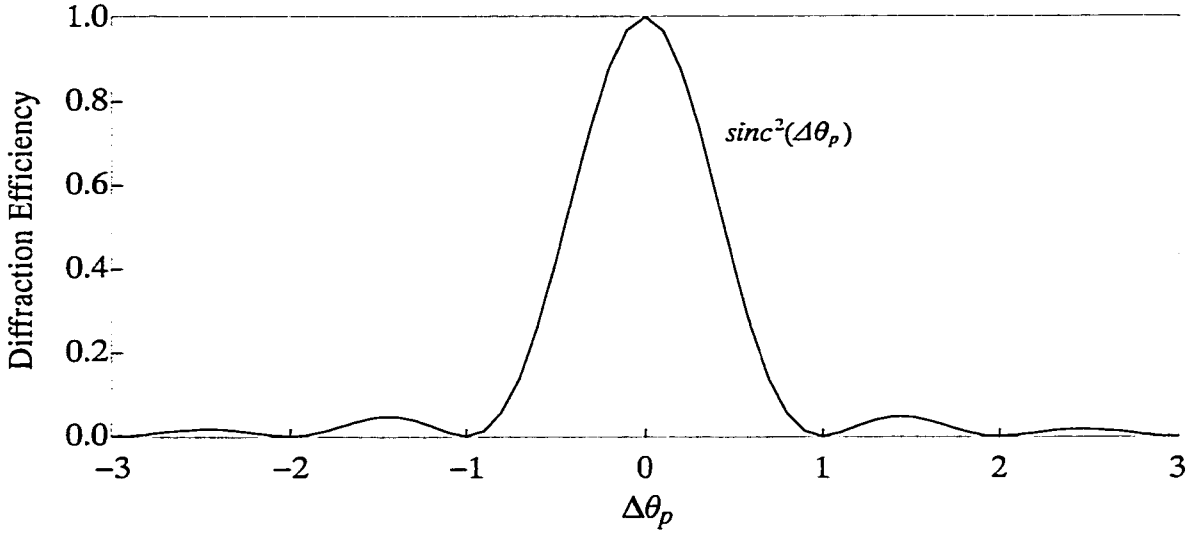


Figure 2.4: Diffraction efficiency versus angular misalignment from Bragg angle.

Diffraction efficiency, η, is the ratio of the incident optical power diffracted for reconstruction of the desired page to the total incident optical power available in the reference beam. For a single page, Figure 2.4 shows the diffraction efficiency as a function

of the difference, $\Delta\theta_p$, between the reference beam angle used to read the page and the reference beam angle (i.e., the Bragg angle) used to record the page. Incident reference beams which are Bragg–matched provide the highest values of diffraction efficiency.

As shown in Figure 2.4, the nulls are located at integer multiples of the difference, $\Delta\theta_p$, and are given approximately by

$$\Delta\theta_p \approx \frac{\lambda}{nd\sin\theta_m} \tag{2.6}$$

where λ is the light wavelength, n is the refractive index, d is the storage medium thickness and θ_m is the angle the image and reference beams make with the normal to the surface of the storage medium [RAS93]. Relation 2.6 indicates each angularly multiplexed page has a different θ_m corresponding to a different $\Delta\theta_p$. Therefore, cross talk cannot be reduced by recording each page with an angular separation corresponding to the diffraction efficiency nulls of all other pages.

A number of researchers [BAS94, CUR93, GU92, LEE88, MAN91, MOK93, NOR93, NOR94, RAS93, YI95] have studied the effects of cross talk in volume holograms. As the number of pages stored, M, increases, the diffraction efficiency decreases as $1/M^2$. The *SNR* decreases as $1/M^2$ [NEI94] and ultimately leads to the occurrence of an unacceptable number of data bit errors. Insight can be gained into the *SNR* related to cross talk by performing a more detailed analysis of cross talk.

The analysis presented here is analogous to the methods used by other researchers [CU93, GU92]. Suppose $N = 2M + 1$ holograms (i.e., pages) are stored as volume holograms in a photorefractive material. These holograms, recorded by using a suitable recording schedule [LI96, MAN91], modulate the dielectric constant of the storage material (e.g., in a fashion mentioned in the section 2.1) such that

$$\Delta\epsilon \propto \sum_{m=-M}^{M} O_m R_m + cc \tag{2.7}$$

where O_m is the amplitude of the object wave corresponding to the m^{th} object image near the Fourier plane (x,y), R_m is the amplitude of the plane reference wave corresponding to the m^{th} point at the reference plane and *cc* is the complex conjugate of the object wave.

During retrieval, a reference wave is incident on the storage medium. The modulated dielectric constant (i.e., expression 2.7) causes scattering of the incident plane wave. From scalar diffraction theory [JAC85], the electric field is given by

$$E(\boldsymbol{r}) \approx \exp(j\boldsymbol{k}_i \cdot \boldsymbol{r}) + \frac{k_o^2}{4\pi r}\exp(jkr)\int d\boldsymbol{r}' \exp(-j\boldsymbol{K}_{di} \cdot \boldsymbol{r}')\Delta\epsilon(\boldsymbol{r}') \tag{2.8}$$

where $\boldsymbol{K}_{di} = \boldsymbol{k}_d - \boldsymbol{k}_i$, $\boldsymbol{k}_d$ is the wavevector of the diffracted wave and $\boldsymbol{k}_i$ is the wavevector of the incident reference wave. In expression 2.8, the first term is the incident plane wave and the second term is the diffracted wave. The diffracted wave has both the desired output image and noise images produced by cross talk during retrieval.

To evaluate the signal and noise terms for the system shown in Figure 2.3, several expressions are required. Expressions must be derived for the reference wave and Fourier transform of the object wave. The m^{th} reference wave R_m can be expressed as

$$R_m = \exp(-j\boldsymbol{k}_m \cdot \boldsymbol{r}). \tag{2.9}$$

where $\boldsymbol{k}_m$ is the wavevector for the m^{th} reference wave. The m^{th} object wave O_m is the transform of the m^{th} image near the Fourier plane. From Fourier optics analysis [GOO68, GOO96, HEC98], O_m can be expressed as

$$O_m(\boldsymbol{r}) = \frac{1}{j\lambda f}\exp[jk(2f + n\Delta_o)]\exp(jkz) \tag{2.10}$$
$$\times \int dx_o dy_o f_m(x_o, y_o)\exp\left[\frac{-j2\pi}{\lambda f}(xx_o + yy_o)\right]\exp\left[\frac{-j\pi z}{\lambda f^2}(x_o^2 + y_o^2)\right]$$

where $f_m(x_o,y_o)$ is the m^{th} object wave pattern, (x_o,y_o) is the coordinate of the object plane, (x,y,z) corresponds to the coordinates at the rear focal plane of lens 1, f is the focal length, n is the refractive index for the lens material and Δ_o is the maximum thickness of the lens.

The diffracted wave at the front focal plane of lens 3 becomes a point at the rear focal plane of the lens. Using the paraxial approximation, the wavevector of the diffracted wave can be related to the coordinates of the point by the expression

$$k_d = [k_{dx}, k_{dy}, k_{dz}] \tag{2.11}$$

where $k_{dx} = \frac{2\pi x_2}{\lambda f}$, $k_{dy} = \frac{2\pi y_2}{\lambda f}$ and $k_{dz} = \frac{2\pi}{\lambda}\left(1 - \frac{x_2^2}{2f^2} - \frac{y_2^2}{2f^2}\right)$.

Using expression 2.7 with Equations 2.9, 2.10, and 2.11 the diffracted wave term in expression 2.8 can be written as

$$g(x_2, y_2) \propto \sum_{m=-M}^{M} \int dx_o y_o f_m(x_o, y_o) V sinc\left[\frac{l}{2\pi}\left(k_{mx} - k_{ix} + \frac{2\pi}{\lambda f}(x_o + x_2)\right)\right] \times sinc\left[\frac{w}{2\pi}\left(k_{my} - k_{iy} + \frac{2\pi}{\lambda f}(y_o + y_2)\right)\right] \times sinc\left[\frac{h}{2\pi}\left(k_{mz} - k_{iz} + \frac{\pi}{\lambda f^2}(x_2^2 - x_o^2 + y_2^2 - y_o^2)\right)\right] \tag{2.12}$$

where k_{ix}, k_{iy} and k_{iz} are the wavevector components of the incident plane wave during readout, k_{mx}, k_{my} and k_{mz} are the wavevector components of the m^{th} reference wave during recording, l, w and h represent the dimensions of the photorefractive storage medium in the x, y and z directions and $V = lwh$ is the volume of the storage medium. Since the Fourier plane of lens 1 is where the storage medium is centered, expression 2.12 represents Fourier holograms.

According to expression 2.12, the amplitude of each stored image is determined by arguments of multiple sinc functions. The x and y components of these arguments provide the spatial frequency cutoff caused by the finite size of the storage medium. By assuming the spatial bandwidth of the stored images is much smaller than the transverse dimensions of the storage medium, the sinc functions which operate on x and y components can be

approximated by δ functions. Using these approximations, expression 2.12 can be evaluated and is given by

$$g(x_2,y_2) \propto \sum_{m=-M}^{M} f_m\left(-x_2-\frac{\lambda f}{2\pi}(k_{mx}-k_{ix}),-y_2-\frac{\lambda f}{2\pi}(k_{my}-k_{iy})\right) \times h\,sinc\left[\frac{h}{2\pi}\left(k_{mz}-k_{iz}+\frac{(k_{mx}-k_{ix})x_2+(k_{my}-k_{iy})y_2}{f}+\frac{(k_{mx}-k_{ix})^2+(k_{my}-k_{iy})^2}{4\pi}\right)\right]. \quad (2.13)$$

The signal term can be separated from the noise terms in expression 2.13. For example, we can illuminate the storage medium with a reference wave oriented at the same angle as was used to record the i^{th} page (hologram). Therefore, $k_{mx} = k_{ix}$, $k_{my} = k_{iy}$ and $k_{mz} = k_{iz}$ and the reconstructed image is given by

$$g_m(x_2,y_2) \propto f_m(-x_2,-y_2). \quad (2.14)$$

where the negative x and y components refer to a reversed image. Terms in expression 2.13 with $k_{mx} \neq k_{ix}$, $k_{my} \neq k_{iy}$ and $k_{mz} \neq k_{iz}$ are the source of the cross talk noise. The sinc functions represent the shifted pattern with reduced amplitude for each noise image. The Bragg mismatch between the reading reference wave and the reference wave used for recording the i^{th} page determine the amount of shift and amplitude reduction.

For the system shown in Figure 2.3, the reference points can be distributed in the reference plane only along the y_i direction one dimensionally (i.e., $x_i = 0$). From Fourier optics [GOO96], a point (x_i,y_i) at the front focal plane of lens 2 is converted to plane wave with the wavevector given by

$$\boldsymbol{k} = [k_x, k_y, k_z] \quad (2.15)$$

where $k_x = -\frac{2\pi x_i}{\lambda f} = 0, \quad k_y = -\frac{2\pi y_i}{\lambda f}\cos\theta - \frac{2\pi}{\lambda}\left[1 - \frac{1}{2}\left(\frac{y_i}{f}\right)^2\right]\sin\theta$ and

$$k_z = \frac{2\pi y_i}{\lambda f}\sin\theta + \frac{2\pi}{\lambda}\left[1 - \frac{1}{2}\left(\frac{y_i}{f}\right)^2\right]\cos\theta.$$

Using Equation 2.15, the Bragg mismatch is given by

$$k_{mx} - k_{ix} = 0, \tag{2.16}$$

$$k_{my} - k_{iy} = \frac{2\pi(y_m - y_i)}{\lambda f}\left[-\cos\theta + \frac{1}{2}\sin\theta\frac{(y_m + y_i)}{f}\right] \tag{2.17}$$

and

$$k_{mz} - k_{iz} = \frac{2\pi(y_m - y_i)}{\lambda f}\left[-\sin\theta - \frac{1}{2}\cos\theta\frac{(y_m + y_i)}{f}\right]. \tag{2.18}$$

The noise terms in expression 2.13 can be expressed as

$$g_{noise}(x_2, y_2) \propto \sum_{m \neq i} f_m\left(-x_2 - \frac{\lambda f}{2\pi}(k_{mx} - k_{ix}), -y_2 - \frac{\lambda f}{2\pi}(k_{my} - k_{iy})\right) \tag{2.19}$$
$$\times h\,sinc\left[\frac{h}{2\pi}\left(k_{mz} - k_{iz} + \frac{(k_{mx} - k_{ix})x_2 + (k_{my} - k_{iy})y_2}{f} + \frac{(k_{mx} - k_{ix})^2 + (k_{my} - k_{iy})^2}{4\pi}\right)\right].$$

For pixels that are in the on–state, assume the object images are randomly distributed patterns with $|f_m| = 1$ and estimate the maximum noise intensity using $I_{noise} = |g_{noise}|^2$. Then the average noise intensity is given by

$$I_{noise} = \sum_{m \neq i} h^2 sinc^2\left[\frac{h}{2\pi}\left(k_{mx} - k_{ix} + \frac{(k_{mx} - k_{ix})x_2 + (k_{my} - k_{iy})y_2}{f} + \frac{(k_{mx} - k_{ix})^2 + (k_{my} - k_{iy})^2}{4\pi}\right)\right], \tag{2.20}$$

the intensity of the signal is given by

$$I_{signal}(x_2, y_2) = h^2 |f_i(-x_2, -y_2)|^2 \tag{2.21}$$

and the intensity noise–to–signal–ratio(NSR) is given by

$$NSR = \sum_{m \neq i} sinc^2 \left[\frac{h}{2\pi} \left(k_{mx} - k_{ix} + \frac{(k_{mx} - k_{ix})x_2 + (k_{my} - k_{iy})y_2}{f} + \frac{(k_{mx} - k_{ix})^2 + (k_{my} - k_{iy})^2}{4\pi} \right) \right]. \tag{2.22}$$

Substituting Equation 2.15 in Equation 2.22 leads to the expression

$$\begin{aligned} NSR = \sum_{m \neq i} sinc^2 \Bigg\{ & \frac{h(y_m - y_i)}{\lambda f} \\ & \times \sin\theta \left[1 + \cot\theta \left(\frac{2y_2 + y_m + y_i}{2f} - \cos\theta \frac{(y_m - y_i)}{2f} \right) \right. \\ & \left. + \cos\theta \frac{(y_m - y_i)(y_m + y_i)}{2f^2} - \frac{y_2(y_m + y_i)}{2f^2} - \sin\theta \frac{(y_m - y_i)(y_m + y_i)^2}{8f^3} \right] \Bigg\}. \end{aligned} \tag{2.23}$$

Keeping terms in the argument of the sinc function of order $\frac{y_m}{f}$, $\frac{y_i}{f}$ and $\frac{y_2}{f}$, Equation 2.23 becomes

$$NSR = \sum_{m \neq i} sinc^2 \left(\frac{h(y_m - y_i)}{\lambda f} \sin\theta \right). \tag{2.24}$$

Equation 2.24 is zero when the separation between adjacent reference points are selected such that the argument of the sinc function is an integer for all m. That is, the cross talk is minimized when adjacent reference points are separated by an amount given by

$$\Delta = \frac{\lambda f}{h \sin\theta}. \tag{2.25}$$

Since expression 2.25 is minimum when $\theta = 90°$, the maximum number of holograms can be stored using this arrangement and selecting the reference points as

$$y_m = m\Delta = \frac{m\lambda f}{h \sin\theta}. \tag{2.26}$$

Using expression 2.25, $\theta = 90°$, and terms up to order $\frac{y_m}{f}$, $\frac{y_i}{f}$ and $\frac{y_2}{f}$ squared, expression 2.23 can be written as

$$NSR \approx \sum_{m \neq i} sinc^2\left[\frac{h(y_m - y_i)}{\lambda f}\left(1 - \frac{y_2(y_m + y_i)}{2f^2}\right)\right]. \qquad (2.27)$$

Figure 2.5 shows the *NSR* at the detector array (i.e., the output plane) for $N = 401$, 601, 801 and 1001 stored holograms with $f = 30$cm, $\lambda = 540$nm and $h = 1$cm. The *NSR* depends on pixel position in the y–direction, y_2, and is independent of pixels in the x–direction at the detector array. Moreover, the stored holograms are referenced with respect to a y–position on the reference plane. The noise is least for pixels located at the center of the output plane and increases as the edge of the output plane is approached. This noise behavior occurs because the separation [GU92] between reference points was chosen to place the centers of the output images at the zeros of the sinc functions. For this separation between reference points, the Bragg matching condition only minimizes the cross talk at the center of the output images whereas off center (axis) pixels have a higher *NSR*. As a result, the overall cross talk noise is minimized for each output image because the sinc function is essentially symmetric near its zeros. However, the side lobes of the sinc function for the Bragg mismatched holograms on each side of the retrieved hologram have magnitude such that these neighboring holograms are the main contributors of cross talk noise.

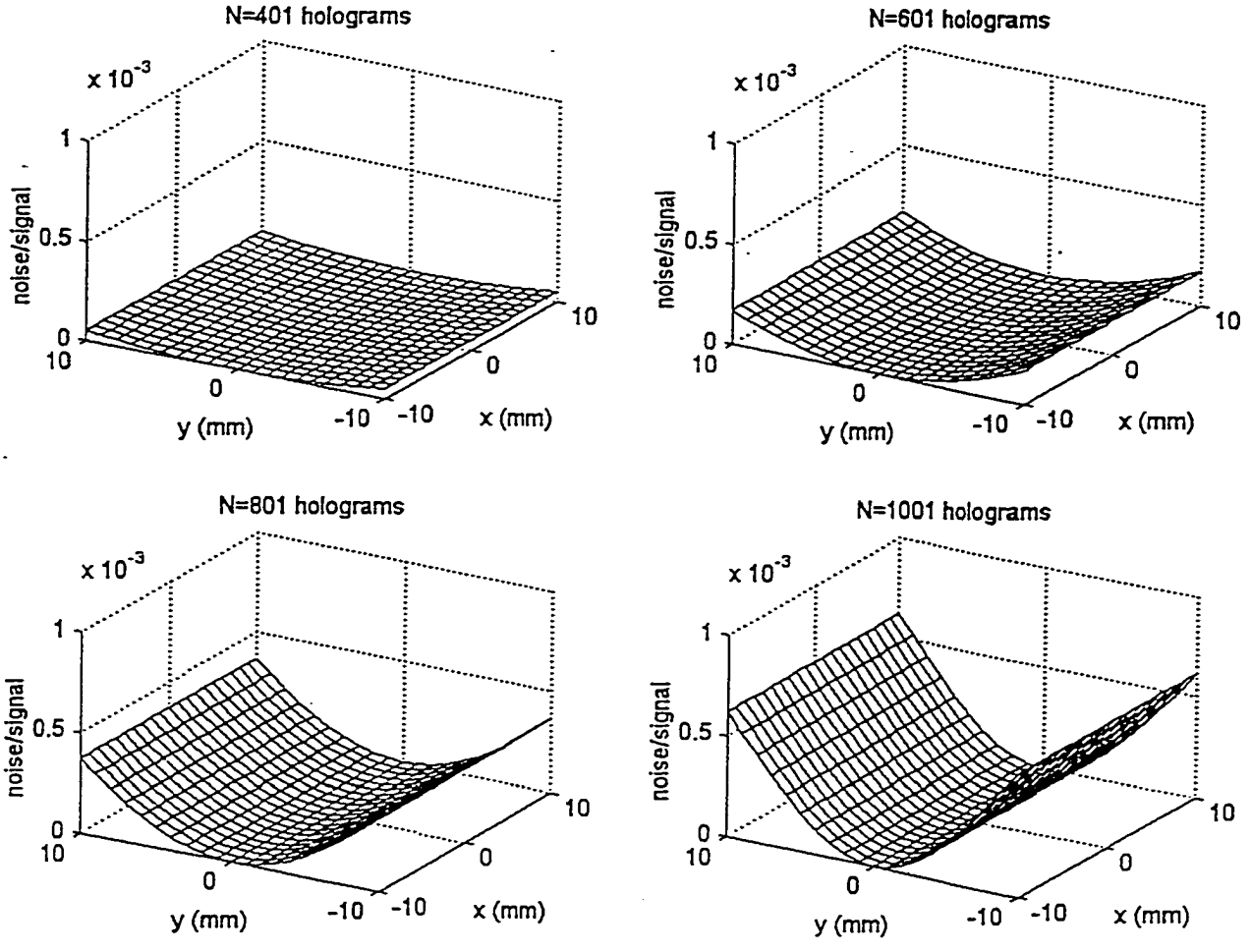


Figure 2.5: *NSR* at detector for N = 401, 601, 801 and 1001 holograms.

ECCs which provide more protection at the edges than at the center of the output plane are desirable. Specifically, UEP array codes can be used in this type of situation because these codes provide particular groups of information bits with different levels of protection. More will be discussed regarding the construction, attributes and capabilities of UEP array codes in Chapter 5 of this Dissertation.

2.4.2 Scattering Noise Source

Information stored in and retrieved from a VHDS system can be affected by scattering noise sources. In an angularly multiplexed VHDS system, noise can be introduced by scatter of the reference wave from a collection of independent scattering centers (i.e., scatterers). Assuming the scattering noise from each scattering center has a random phase, the noise amplitude is complex–valued and can be represented as a complex–valued random variable [GOO85, GU96, HEA95].

During information retrieval, the reconstructed wave arrives at the detector containing the object (signal) component and a noise component. The amplitude of the object component is assumed to be real with a value of 1 for an on–state pixel and α for an off–state pixel. The amplitude of the scattering noise component, $n = a + ib$, is complex–valued where a and b are assumed to be statistically independent and have the same variance, σ^2. The joint probability density function (*PDF*) for the amplitude of the reconstructed wave is [GOO85]

$$P_{AB}(a,b) = \frac{1}{2\pi\sigma^2}\exp\left[-\frac{(a-s)^2 + b^2}{2\sigma^2}\right] \tag{2.28}$$

At the photodetector array, a square law operation is used to convert the optical intensity, $I = C^2$, pattern into an electronic form.

Using a change in variables such as $a = c\cos\theta$ and $b = c\sin\theta$ to translate from rectangular to polar coordinates, the *PDF* for the random variable obtained from Equation 2.28 becomes

$$P_C(c) = \frac{c}{\sigma^2}\exp\left[-\frac{(c-s)^2}{2\sigma^2}\right]J_0\left(\frac{cs}{\sigma^2}\right) \tag{2.29}$$

where J_0 is the zero–order modified Bessel function of the first kind given by

$$J_0 = \frac{1}{2\pi}\int_{-\pi}^{\pi}\exp\left(\frac{cs}{\sigma^2}\right)\cos\theta\, d\theta \tag{2.30}$$

Using Equation 2.29 and the monotonic transformation, $C = \sqrt{I}$, the *PDF* of the detected intensity pattern, I is given by

$$P_I(I) = \frac{1}{2\sigma^2}\exp\left[-\frac{(I + I')}{2\sigma^2}\right]J_0\left(\frac{II'}{\sigma^2}\right) \tag{2.31}$$

where I' represents the intensity at the detector and is related to the output pixel value. Equation 2.31 is called the Rician density function. The Rician distribution has been shown using experimental results to accurately describe the intensity distributions found in VHDS systems [DOM88, NOR93, PSA96].

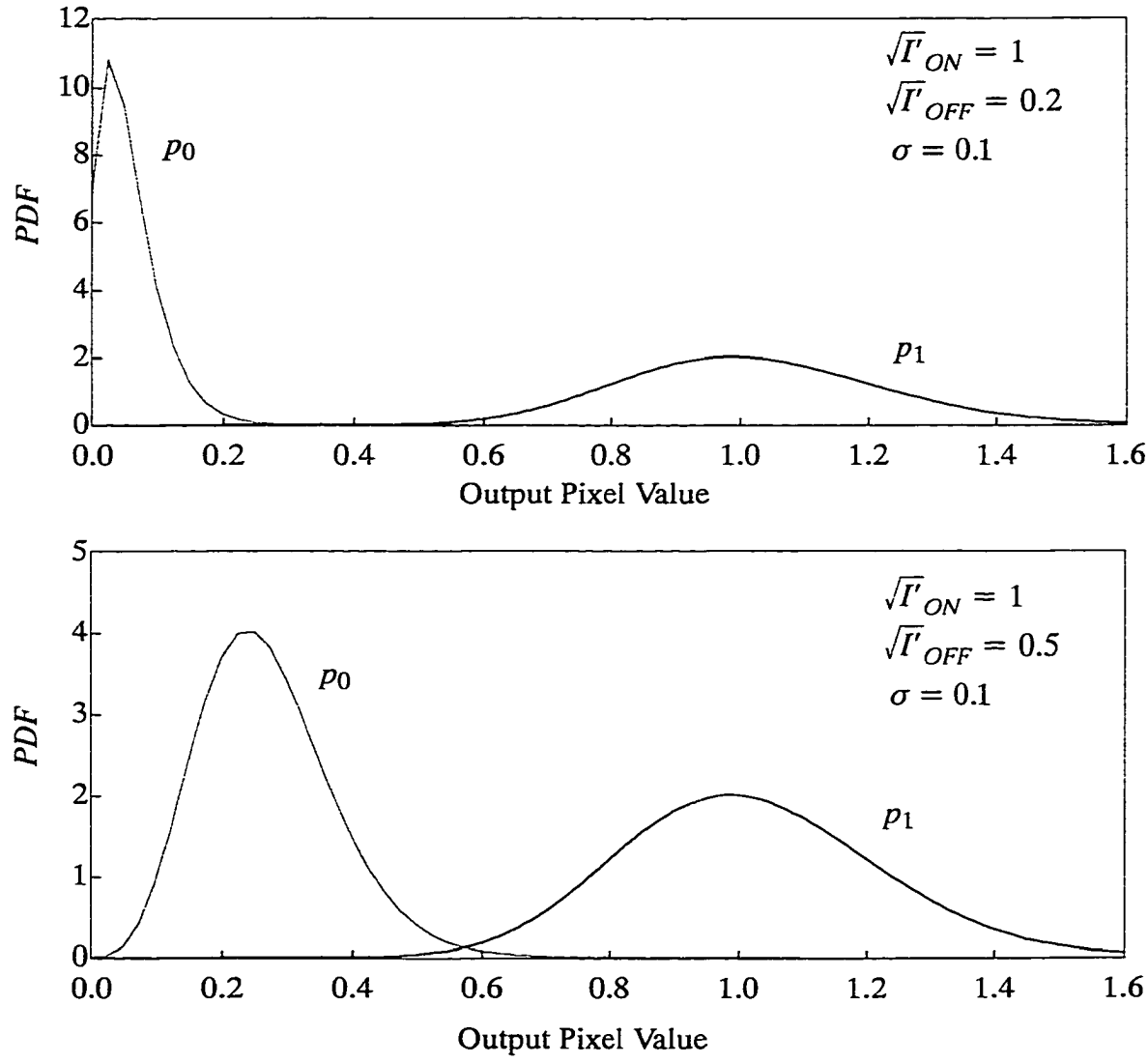


Figure 2.6: *PDF* versus output pixel value for $\sqrt{I'}_{OFF} = 0.2$ and $\sqrt{I'}_{OFF} = 0.5$.

The *PDF*s for on–state pixels, $\sqrt{I'} = 1$, and for off–state pixels $\sqrt{I'} = 0.2, 0.5$ are shown in Figure 2.6. The intensity contrast ratios represented are 25 and 4, respectively. Figure 2.6 shows the variance of the intensity distribution depends on the signal value such that the variance of p_1 (i.e., the probability of an on–state pixel) is larger than the p_0 (i.e., the probability of an off–state pixel). From Equation 2.31, the first moment (mean) and second moment are given by

$$\bar{I} = I' + 2\sigma^2, \tag{2.32}$$

$$\bar{I}^2 = (I')^2 + 8\sigma^2(\sigma^2 + I'). \tag{2.33}$$

Solving the equation, $\bar{I}^2 = (\bar{I})^2 + \sigma^2$, for σ^2 and using Equations 2.32 and 2.33, leads to the expression for the second central moment (i.e., the variance) given by

$$\sigma^2 = \bar{I}^2 - (\bar{I})^2 = 4\sigma^2(\sigma^2 + I'). \tag{2.34}$$

Mathematically, Equation 2.34, verifies that the variance increases with signal value and is due to the nonlinear nature of the square–law detection process. The intensity *SNR* is defined as

$$SNR = \frac{\bar{I}}{\sigma} = \frac{I' + 2\sigma^2}{2\sigma\sqrt{I' + \sigma^2}}. \tag{2.35}$$

2.4.3 Detection Process Related Noise Sources

The detector array is where reconstructed pages are detected, where information is extracted and where detector related noise occurs during the detection process. This array is used at the output image plane to convert incident optical power into electrical signals. A square law intensity detection process is used to convert incident photons into a photocurrent. The photocurrent, I_{ph}, which is proportional to the light intensity, is directed into a resistive load and the voltage produced across the load is measured. During this

detection process, several noise sources (e.g., thermal noise, flicker noise and shot noise) are active and add electrical noise to the output signal.

Thermal noise is the result of random variations in the photocurrent. The path length between scattering and collision events varies and leads to random variations in carrier (e.g., electrons and holes) density and mobility. Mathematically, this noise source is modelled as a stationary random process with Gaussian statistics [AGR97]. The frequency of thermal noise varies inversely with the mean time between collisions and can be on the order of 10^{-12} Hz when the carrier velocity is large. Thermal noise is also called Johnson noise or Nyquist noise or resistive noise. For a resistive load, the mean square noise current is given by

$$\bar{i}_t^2 = \frac{4k_bTB}{R} \tag{2.36}$$

where k_b is the Boltzmann constant, T is the absolute temperature, B is the bandwidth and R is the value of the resistive load. For T = 300°K, B = 100 MHz and $R = 10^4\ \Omega$, $\sqrt{\bar{i}_t^2} = 13nA$.

Flicker noise arises in transistors from generation and recombination effects (e.g., trapped carriers) on the surface of the semiconductor devices and from temperature fluctuations in the photodetector array circuitry. Material defects (e.g., traps, dislocations and stacking faults) are present in the starting materials from which devices are constructed and additional defects can be introduced during the thermal cycling involved in device processing. Since the lifetime of trapped carriers can be on the order of 1 msec, the frequency of flicker noise can be on the order of 1 kHz. The spectral amplitude of this noise varies inversely with the operating frequency, f. This type of noise is also called one–over–f noise and the mean square noise current is given by

$$\bar{i}_f^2 \propto \frac{1}{f}. \tag{2.37}$$

When random variations occur in the generation and recombination of electron–hole pairs, shot noise is produced. Mathematically, this noise source is represented as a stationary random process with Poisson statistics which in practice can be approximated by Gaussian statistics [AGR97]. The frequency of shot noise varies inversely with the carrier lifetime (e.g., 10^{-6} to 10^{-8} sec) and can range in value from 1 to 100 MHz. This type of noise is also called Schottky noise. The mean square noise current is given by

$$\bar{i}_s^{\,2} = 2qB(I_{ph} + I_d + I_b) \tag{2.38}$$

where q is the elementary charge, B is the bandwidth and I_{ph} is the photocurrent, I_d is the dark current, I_b is the background or leakage current. For $B = 100$ MHz, $I_{ph} = 20$ μA, I_d and $I_b \leq 5$ nA, $\sqrt{\bar{i}_s^{\,2}} = 25nA$.

Shot noise and thermal noise are the two fundamental noise sources which produce unwanted current fluctuations in the photodetector array [AGR97]. The thermal noise current can be modelled by Gaussian statistics with a stationary and variance, σ_t^2. The shot noise current can be approximated as a Gaussian random variable with a stationary mean and variance, σ_s^2. Since the sum of two Gaussian random variables is still a Gaussian random variable, the total noise current has a variance $\sigma_e^2 = \sigma_t^2 + \sigma_s^2$ and a Gaussian *PDF* given by

$$P(x) = \frac{1}{\sqrt{2\pi}\,\sigma_e^2}\exp\left[-\frac{(x - x')^2}{2\sigma_e^2}\right]. \tag{2.39}$$

The *PDF*s for on–state pixels, $\sqrt{x'} = 1$, and for off–state pixels $\sqrt{x'} = 0.3$ are shown in Figure 2.7. As expected, the means are different for p_1 (i.e., the probability of an on–state pixel) and p_0 (i.e., the probability of an off–state pixel). The optimum threshold value is approximately 0.7.

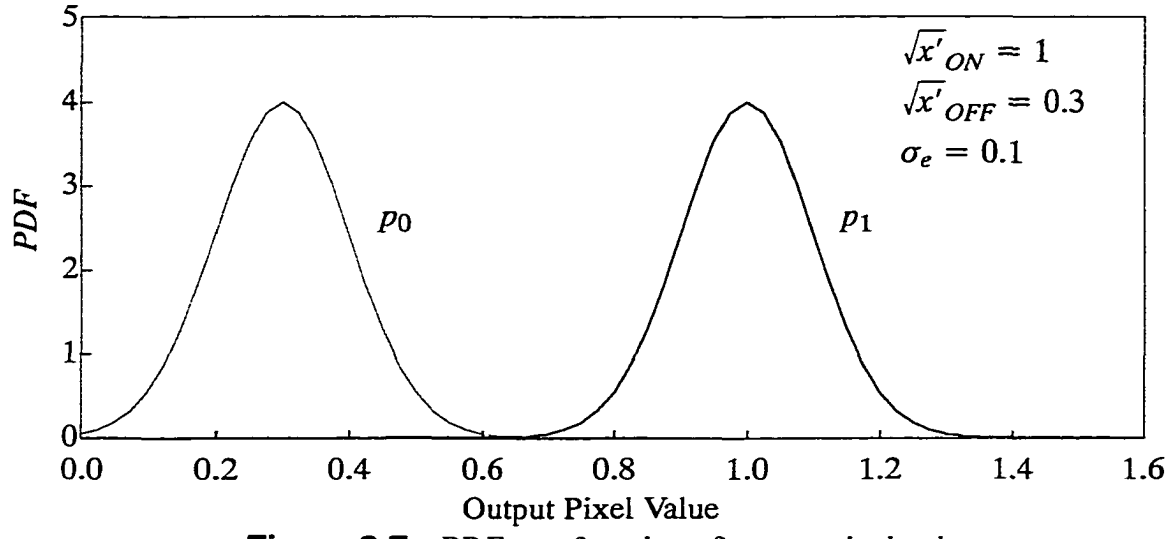


Figure 2.7: *PDF* as a function of output pixel value.

2.5 Computing the Raw Bit Error Rate

The *RBER* is determined mainly by computing the areas under the overlapping tails of the *PDF*s for pixels in the on–state and pixels in the off–state. The overlapping tails of the *PDF*s (e.g., shown in Figure 2.6) cause thresholding errors to occur. Therefore, a threshold value, I_{th}, must be carefully selected to compute the *RBER* [HEA94, HEA95, GU96]. When $I > I_{th}$, the output pixel is considered to be in the on–state and when $I < I_{th}$, the output pixel is considered to be in the off–state. The optimum I_{th} value is the value which minimizes the *RBER*. Equation 2.31 can be used to write *PDF* expressions for pixels in the on–state and for pixels in the off–state. When the on–state and off–state pixels are equally likely, the optimum I_{th} value is the intensity value obtained from setting the two *PDF* expressions equal to each other [SHA88]. A transcendental equation is produced and must be solved numerically [HOO94]. The *RBER* is defined as

$$RBER = P(I' = 1)P(I < I_{th}|I' = 1) + P(I' = \epsilon)P(I > I_{th}|I' = \epsilon) \qquad (2.40)$$

where $P(I' = 1)$ is the probability for the pixel to be in the on–state, $P(I < I_{th}|I' = 1)$ is the probability a pixel in the on–state was detected as being in the off–state, $P(I' = \epsilon)$ is the

probability for the pixel to be in the off–state and $P(I > I_{th}|I' = \epsilon)$ is the probability a pixel in the off–state was detected as being in the on–state. When the probabilities of a pixel being in the on–state or off–state are equal, $P(I' = 1) = P(I' = \epsilon) = 0.5$. The expressions for $P(I < I_{th}|I' = 1)$ and $P(I > I_{th}|I' = \epsilon)$ are given by

$$p(I < I_{th}|I' = 1) = \int_0^{I_{th}} p_I(I|I' = 1)dI = 1 - \int_{I_{th}}^{\infty} p_I(I|I' = 1)dI \qquad (2.41)$$

and

$$p(I > I_{th}|I' = \epsilon) = \int_{I_{th}}^{\infty} p_I(I|I' = \epsilon)dI, \qquad (2.42)$$

respectively.

At this point several options are available to compute the *RBER*. One option is to use the Marcum Q function [MAR60, SCH66]

$$Q(\beta,\gamma) = \int_{\gamma}^{\infty} \rho J_0(\beta\rho) \exp\left(-\frac{\beta^2 + \rho^2}{2}\right)d\rho \qquad (2.43)$$

to compute the integrals in Equations 2.41 and 2.42. Another option is to use analytical approximations [GU96] to obtain an analytical expression for the *RBER*. Direct numerical evaluation of Equation 2.31 is yet another option to compute the *RBER*. The option used depends mostly on the preference of the researcher.

2.6 Error Types within a Data Page

Channel noise and error mechanisms are related issues in VHDS systems. To design low error rate systems potential noise sources must be identified and understood. For example, a system can be characterized by a set of noise sources which depend on the specific implementation details. Noise and related error mechanisms stemming from system imperfections derive from optical and electrical sub–systems and environmental sources.

Collectively, the different noise sources interact to produce soft and hard errors. While a soft error is usually correctable by rereading the data page, a hard error type persists

upon rereading the page. Within a data page, the on–state and off–state pixels and corresponding error types are shown in Figure 2.8. The data bits are represented by square pixels with high optical intensity corresponding to an on–state pixel or logical–1 and low optical intensity corresponding to an off–state or logical–0. The error type can be a random single bit error, a burst error or a cluster error. Random single bit errors occur independently on each symbol. Burst errors occur as a sequence of single bit errors. Cluster errors occur as a 2–D array of single bit errors. These error types must be taken into consideration when designing appropriate ECCs for the VHDS system.

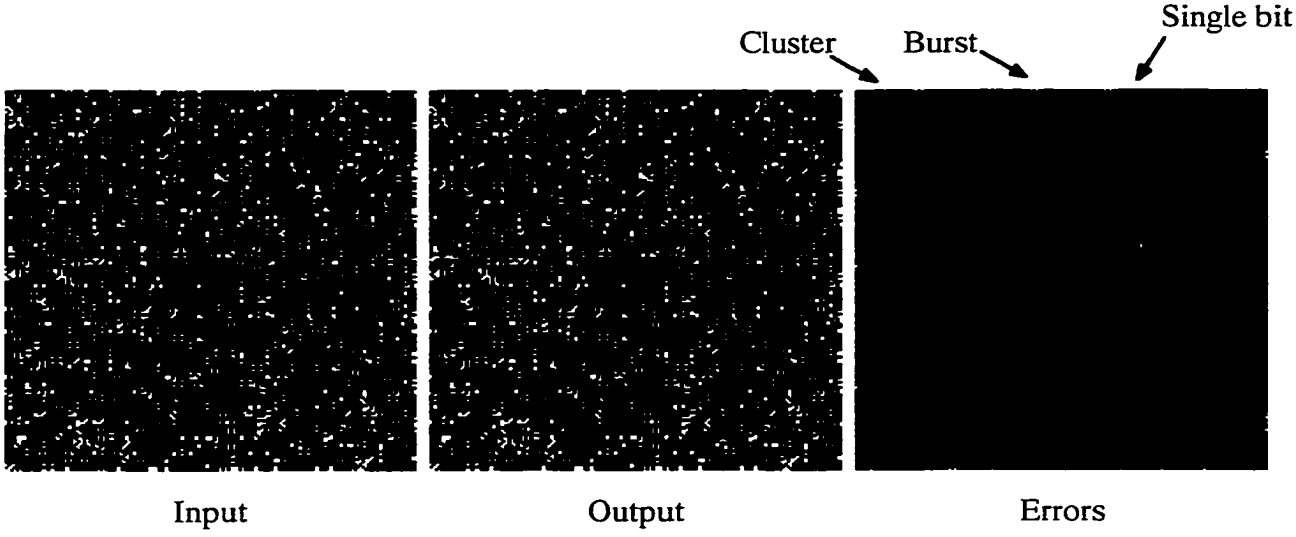


Figure 2.8: Input, output pixel values and error types within a page.

Dust and debris on optical or optoelectronic components can cause undesired artifacts (e.g., soft and hard errors) to appear in the stored and retrieved images. Dust particles on components far from the image plane will adversely impact more pixels. Dust particles on components located at the image plane affect fewer pixels and tend to block light from reaching the intended pixels. Sometimes it is helpful to think of dust and debris, which can occur on components in a random fashion, as a random error type due to environmentally related noise sources.

Chapter 3
Block 2-D Array Codes

This chapter presents an introduction to array codes. The characteristics of random error correcting array codes are considered. These codes are adapted for use as burst error correcting codes. Next, a cluster error correcting array code is described. The hardware requirements and associated timing delays are analyzed for these error correcting codes. This information will be important for understanding later chapters.

3.1 General Terminology

Error correcting codes are designed to protect the integrity of data by detecting and correcting errors occurring in the channel. For blocks of data represented as bits (i.e., binary data), redundant bits are added to the original data block producing properly formed code words. Depending on the type of code used, the redundant bits are generated using the original data block and rules such that when errors occur an improper code word is formed. Therefore, the code words represent all possible words encoded according to the rules of the code and, when decoded or checked according to these rules do not produce error conditions

(i.e., parity bit check failures). When improperly formed code words (i.e., non–code words) are decoded or checked, error conditions are produced.

The distance of a code is denoted by d and is usually referred to as the Hamming distance, d. This important parameter represents the minimum number of differing bit positions between any two code words. For example, no two code words of a $d = 2$ code differ in less than 2 bit positions; no two code words of a $d = 3$ code differ in less than 3 bit positions and so on. For one code word to be mistaken as another code word, it must experience an error in at least d of its bit positions. For example, a single parity check (SPC) code is a distance 2 code and at least 2 bit positions must be in error for an undetectable error condition to occur. A code can detect t errors and correct c errors provided $d \geq t + c + 1$ and $c \leq t$ [FAR90, RAO89].

A code is specified using the length of the code, number of information bits, distance of the code and code rate. The length of a code is denoted by n. This parameter includes both the information and parity bit checks (i.e., redundant bits). The number of information bits is denoted by k. Therefore, the number of redundant bits or parity bit checks is $n - k$. The rate of the code, r, is defined as $r = \frac{k}{n}$. Often a code is described as a (n, k, d) code with rate r.

For a given 2–D code size, the number of parity bit checks will be smaller as the code block approaches a square. Let the information part of a code have size $k = (m-1)(p-1)$. Then the overall size of the code is $n = mp$ where $m = n_1$ and $p = n_2$. Then k in terms of m and n is given by

$$k = (m - 1)(\frac{n}{m} - 1) \tag{3.1}$$

In Equation 3.1, taking the derivative of k with respect to m and setting it to zero leads to

$$0 = (\frac{n}{m} - 1) - (m - 1)\frac{n}{m^2}. \tag{3.2}$$

Solving Equation 3.2 for n leads to $n = m^2$ which is the same as $n = n_1 n_1$ (i.e., a square with sides of size n_1 by n_1).

3.2 Analysis of Row and Column Array Codes

Array codes [FAR79, FAR92, FAR96] are formed by arranging component codes into 2–D or multidimensional arrays. Using the SPC code as a component code, a 2–D array code is obtained as shown in Figure 3.1. The information bits are arranged in a $k_1 \times k_2$ array with single parity bit checks generated across the rows and columns [CAL61].

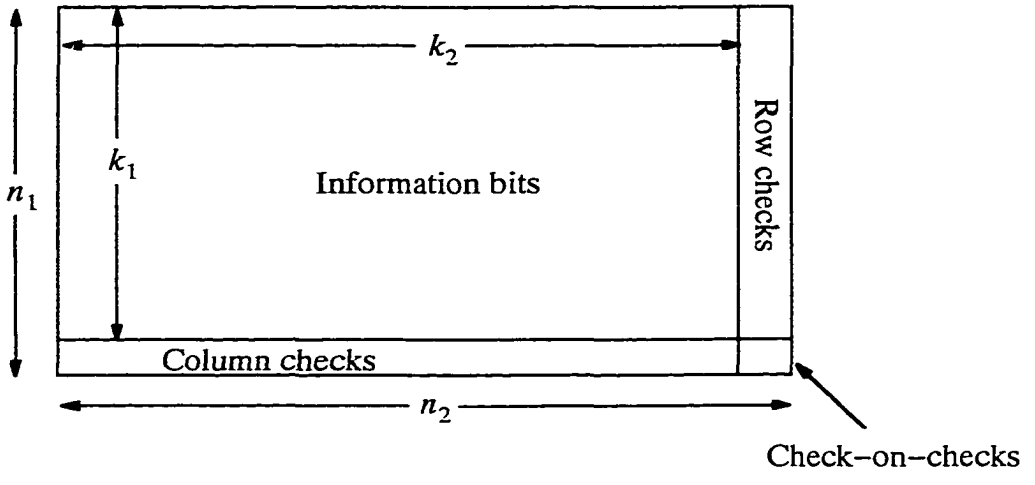


Figure 3.1: Row and column array code.

For the code shown in Figure 3.1, the parameters are given by the following expressions.

Length of code: $n = n_1 n_2 = (k_1 + 1)(k_2 + 1)$

Information bits: $k = k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = k_1 + k_2 + 1$

Number of errors that can be detected: $t = 2$

Number of errors that can be corrected: $c = 1$

Type of errors that can be corrected: random

The distance of the row and column code can be increased by including more direction sets of parity bit checks with the information bits. For example, a (15, 9, 3) code without the check on checks parity bit is shown in Figure 3.2. There are two direction sets

1	2	3	p_1
4	5	6	p_2
7	8	9	p_3
p_4	p_5	p_6	

Figure 3.2: A (15, 9, 3) row and column array code without check–on–checks.

(e.g., rows and columns) of parity bit checks included with the information bit array. Table 3.1 shows how the distance of the code is increased as additional direction sets of parity bit checks are added to the code.

Table 3.1: Direction sets of parity bit checks for corresponding (n, k, d) codes.

Direction Sets of Parity Bit Checks	n	k	d	r
Rows and columns	15	9	3	0.600
Rows, columns and one main diagonal	18	9	4	0.500
Rows, columns and two main diagonals	21	9	5	0.429

The geometric characteristics of these codes provide considerable flexibility for adapting these codes. The information bit arrays can be arranged in square or rectangular or triangular shapes. These codes can be extended in a 3–D fashion to form stacks or layers of array codes. The characteristics of 3–D array codes will be discussed in Chapter 4.

As the page size increases, issues associated with reduced error correcting capability limit the usefulness of the row and column array code. A multiblock strategy [ASH96, GOE95, GOE96, HUT96] provides a practical solution for applications involving an entire page of data which may be as large as 10^6 bits. A multiblock code represented by a $m_1 \times m_2$ array of $n_1 \times n_2$ row and column code blocks is shown in Figure 3.3. If one error occurs in each code block, $k_1 = 15$, $k_2 = 7$ and the page size is 512×512 bits, as many as 2176 errors can be corrected [GOE95].

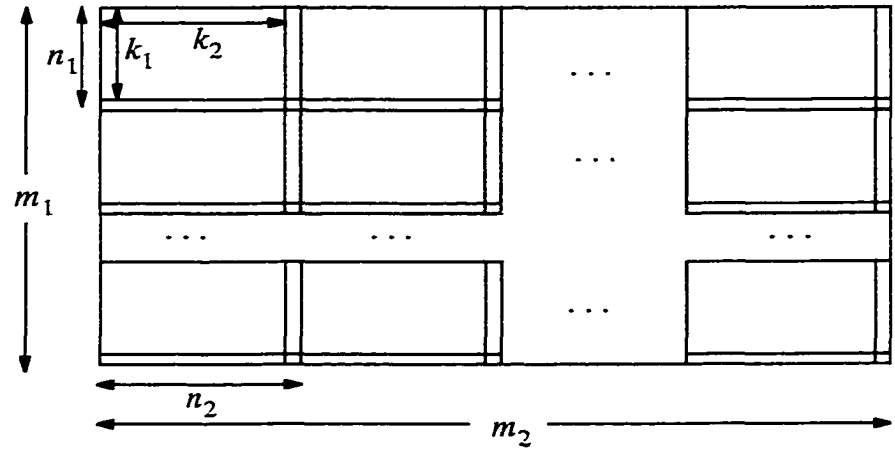


Figure 3.3: Multiblock row and column array code.

The parameters of the multiblock code shown in Figure 3.3 are given by the following expressions.

Length of code: $n = m_1 m_2 n_1 n_2 = m_1 m_2 (k_1 + 1)(k_2 + 1)$

Information bits: $k = m_1 m_2 k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = m_1 m_2 (k_1 + k_2 + 1)$

Number of errors that can be detected: $t = \{t_{min} = 2, t_{max} = 2m_1 m_2\}$

Number of errors that can be corrected: $c = \{c_{\min} = 1, c_{\max} = m_1 m_2\}$

Type of errors that can be corrected: random

3.2.1 Data Encoding for Row and Column Array Codes

Encoding requires computing the row and column parity bit checks for the data array. The parity bit checks are appended to each row and column to produce either odd or even parity. Assuming even parity, the single parity bit checks are computed across each row and across each column as shown in Figure 3.4.

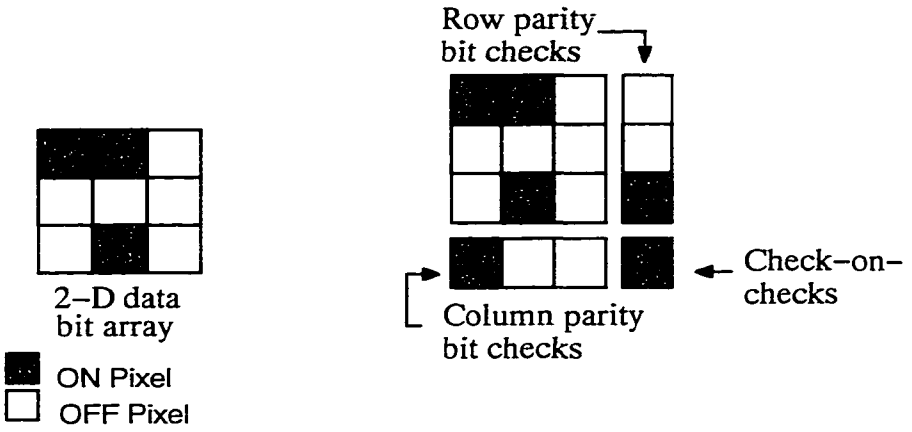


Figure 3.4: Example of data encoding applied to a 3×3 array code.

3.2.2 Data Decoding for Row and Column Array Codes

During storage or retrieval, random single bit errors may occur in a page of data. A purpose of the decoding process is to correct random single bit errors. In the decoding process, a page of data received at a detector array is latched, row and column syndromes are computed, error detection and correction are performed at the intersection of single bit failures in the row and column syndromes. Assuming even parity, Figure 3.5 shows part of the decoding process.

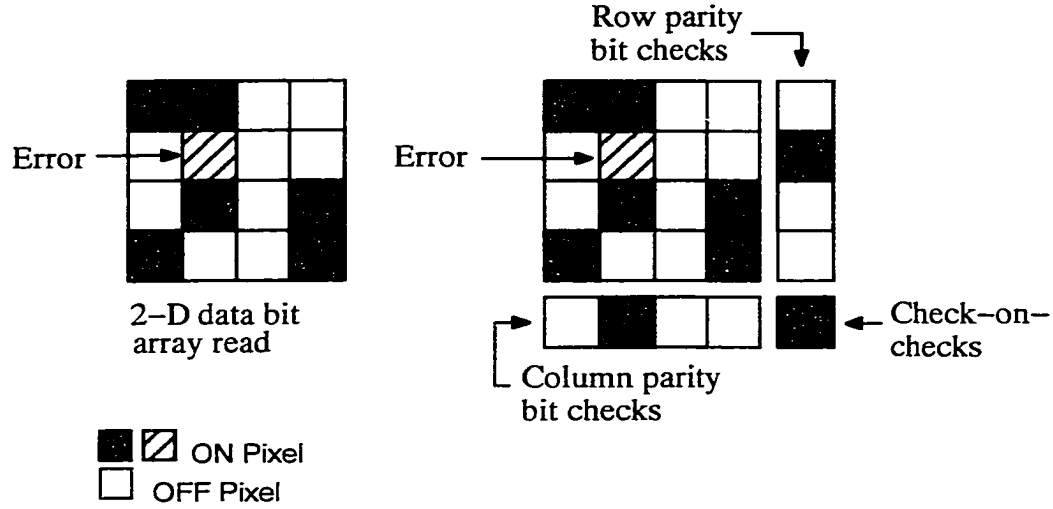


Figure 3.5: Example of data decoding applied to a 4×4 array code.

When data are read, the intersection of ones in the row and column syndromes gives the position of as many as two errors in the 2–D data bit array (i.e., code block). Within the 2–D code block, two errors in different rows and columns are detected by the presence of ones (i.e., bit failures) in the parity bit check vectors and a zero in the check–on–checks. If more than two errors occur, this code may not detect all cases, may miscorrect and may misdetect errors. For example, a pattern of three errors located at the corners of a subarray can be miscorrected by introducing a fourth error; a pattern of four errors [ROW68] located at the corners of a subarray can be misdetected such that no parity check bit differences are produced.

3.2.3 Encoder for Row and Column Array Codes

The encoder for the code shown in Figure 3.1 must compute the row and column parity bit checks for a page of data $k_1 \times k_2$ bits in size. Assuming two–input gates are used, row parity bit checks can be computed using trees of $k_2 - 1$ XOR–gates in $\lceil \log_2 k_2 \rceil$ gate delays and column parity bit checks can be computed using trees of $k_1 - 1$ XOR–gates in $\lceil \log_2 k_1 \rceil$ gate delays. The hardware required for computing the syndromes is

$H_{re} = (k_2 - 1)n_1 + (k_1 - 1)k_2$ gates and the encoding delay for computing the syndromes is $T_{re} = \max(\lceil \log_2 k_1 \rceil, \lceil \log_2 k_2 \rceil)$ gate delays.

3.2.4 Decoder for Row and Column Array Codes

For the code shown in Figure 3.1, the decoder algorithm performs four steps:

1. Latch data
2. Compute syndromes
3. Detect errors
4. Correct error.

In step 1, an array of $n_1 \times n_2$ information bits are latched. Each bit in the information bit array (i.e., the optical page of data) can be latched using the hardware shown in Figure 3.6. If six gates are used to implement the flip–flop, a total of nine gates are needed to implement the hardware. If the flip–flop introduces four gate delays, approximately six gate delays are encountered from the photodetector signal to the DATA_OUT signal. Therefore, the decoder hardware required to latch the data is $H_{rdl} = 9n_1n_2$ gates and the delay encountered is $T_l = 6$ gate delays.

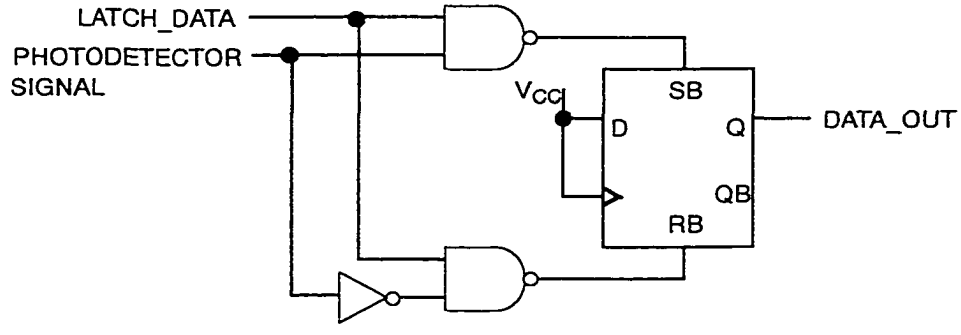


Figure 3.6: Hardware used to store one bit of the information bit array.

In step 2, trees of k_2 XOR–gates and k_1 XOR–gates can compute respectively the row syndrome in $\lceil \log_2 n_2 \rceil$ gate delays and the column syndrome in $\lceil \log_2 n_1 \rceil$ gate delays.

Figure 3.7 shows part of the hardware used to compute the syndromes. The hardware demands to compute the syndromes and the delay encountered are respectively $H_{rds} = k_2(n_1 + 1) + k_1 n_2$ gates and $T_{rds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays.

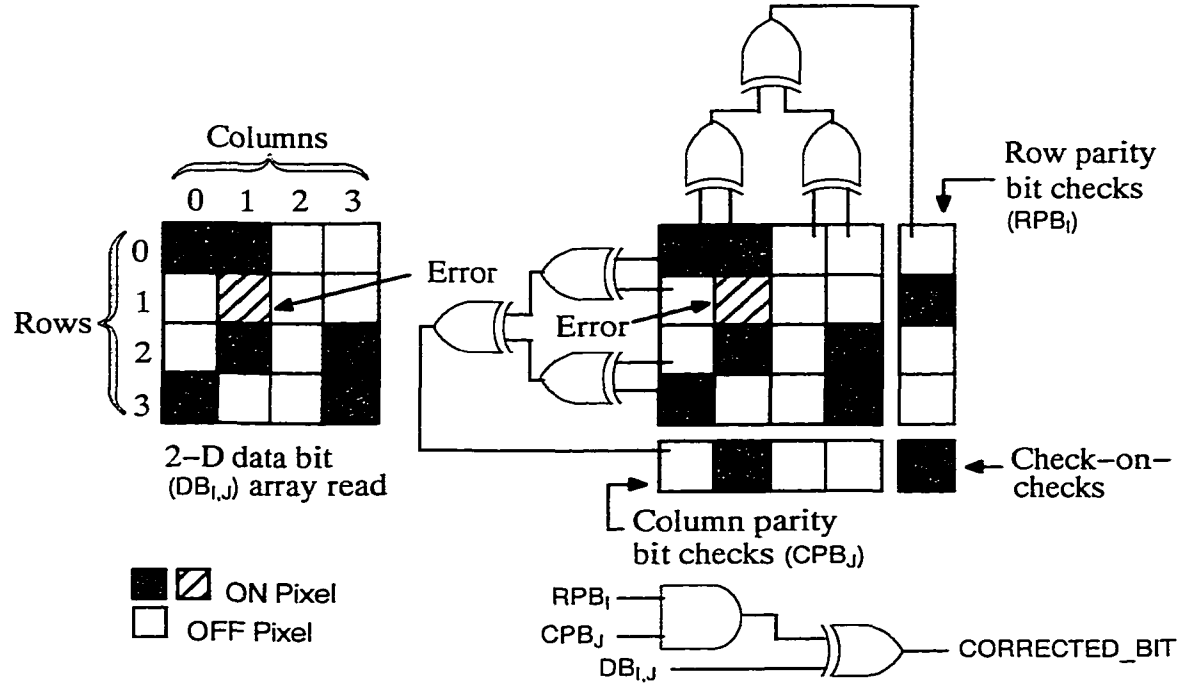


Figure 3.7: Hardware used to compute syndromes, detect and correct errors for an array code.

Several conditions must be checked to determine the error status of the system. In cases of two or more errors, the decoder cannot provide a corrective action. The no–corrective–action–taken condition must be indicated so the system can read the page again or take other corrective action. When one of the row and one of the column parity bit checks and the check–on–checks bit indicate an error, we assume a single error has been detected in the information bits. A corrective action can proceed when one error occurs. When only one of the row or one of the column parity bit checks indicates an error, we assume the check bit is in error. The page of information bits is passed to the next stage of the system.

In steps 3 and 4, the location of a data bit in error is detected and corrected. The hardware shown in Figure 3.7 can be used to perform steps 3 and 4 of the algorithm. The location of a data bit in error is detected as the intersection of single bit failures (i.e., the presence of ones in the parity check vectors and a zero in the check–on–checks) in the row and column syndromes. The data bit in error is corrected (i.e., inverted) using corresponding error correction logic (e.g., an AND–gate and XOR–gate). When multiple bit failures occur only in the row syndrome or in the column syndrome, an uncorrectable error has occurred. A tree of $n_1 - 1$ OR–gates using the row syndrome as input and another tree of $n_2 - 1$ OR–gates using the column syndrome can provide two outputs which are used as inputs to an XOR–gate to check for the uncorrectable error condition. The hardware needed for error detection and correction is $H_{rdc} = 2n_1n_2 + (n_1 - 1) + (n_2 - 1) + 1 = 2n_1n_2 + n_1 + n_2 - 1$ gates and the delay is $T_{rdc} = 2$ gate delays, if an uncorrectable error has not occurred, otherwise $T_{rdc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil) + 1$ gate delays.

In summary, if an uncorrectable error has not occurred, the decoding delay for the row and column (RAC) array code is $T_{rd} = T_1 + T_{rds} + T_{rdc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil) + 8$ gate delays otherwise $T_{rd} = 2\max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil) + 7$ gate delays. The decoding hardware needed is $H_{rd} = H_{rdl} + H_{rds} + H_{rdc} = 11n_1n_2 + k_2(n_1 + 1) + k_1n_2 + n_1 + n_2 - 1$ gates.

3.2.5 Encoding and Decoding Multiblock RAC Array Codes

An analysis can be performed to study the hardware requirements and time delays associated with the multiblock code shown in Figure 3.3. When one or no error occurs in any component (i.e., uniblock) row and column code making up the entire multiblock code, we assume the page of data have been correctly read. In addition to the encoding and decoding hardware used in the uniblock code, each component uniblock code requires

decoding hardware to generate a correct, correctable and uncorrectable error condition status.

A block serial implementation encodes and decodes one component code after another in some sequence. If a component code is encountered with an uncorrectable error condition status, the decoding process stops. If a component code is encountered with a correctable error condition status, corrective action is taken and the decoding process continues. If all component codes displayed an error condition status of correct, the page of data is assumed to have been correctly read and is passed on to the next stage of the system. Expressions for the variables T_{re}, T_{rd}, H_{re} and H_{rd} were derived in sections 3.2.3 and 3.2.4. These variables are used in the expressions shown below. The encoding delay, T_{mres}, decoding delay, T_{mrds}, H_{mres} encoding hardware requirement, H_{mres}, and decoding hardware requirement, H_{mrds}, are given by the following expressions.

Encoding delay: $T_{mres} = m_1 m_2 T_{re}$

Decoding delay: $T_{mrds} = m_1 m_2 T_{rd}$

Encoding hardware: $H_{mres} = H_{re}$

Decoding hardware: $H_{mrds} = H_{rd}$

A block parallel implementation allows component codes to be encoded and decoded in parallel. Either all component codes can be processed in parallel in a single pass or some fraction of all the component codes can be processed in a pass requiring multiple passes for process completion. Assume all component codes are processed in parallel in a single pass. Let T_{rgA} be the gate delay for the global–AND–gate and H_{rgA} be the global–AND–gate decoding hardware needed for the multiblock RAC array code. For encoding and decoding, the time delays and hardware requirements are given by the following expressions.

Encoding delay: $T_{mrep} = T_{re}$

Decoding delay: $T_{mrdp} = T_{rd} + T_{rgA}$

Encoding hardware: $H_{mrep} = m_1 m_2 H_{re}$

Decoding hardware: $H_{mrdp} = m_1 m_2 H_{rd} + H_{rgA}$

The multiblock array code has advantages over the uniblock array code regarding page size, time to process the data and hardware requirements. With respect to page size, Figure 3.8 shows the multiblock code can process data per block in less time. The lower time delay for the multiblock codes is expected because the smaller block size requires a tree of gates structure containing fewer levels of gating. Figure 3.9 indicates the hardware demands per block is significantly less for multiblock array codes. The multiblock code per block uses smaller code blocks requiring a smaller hardware gate count for implementation than the uniblock code.

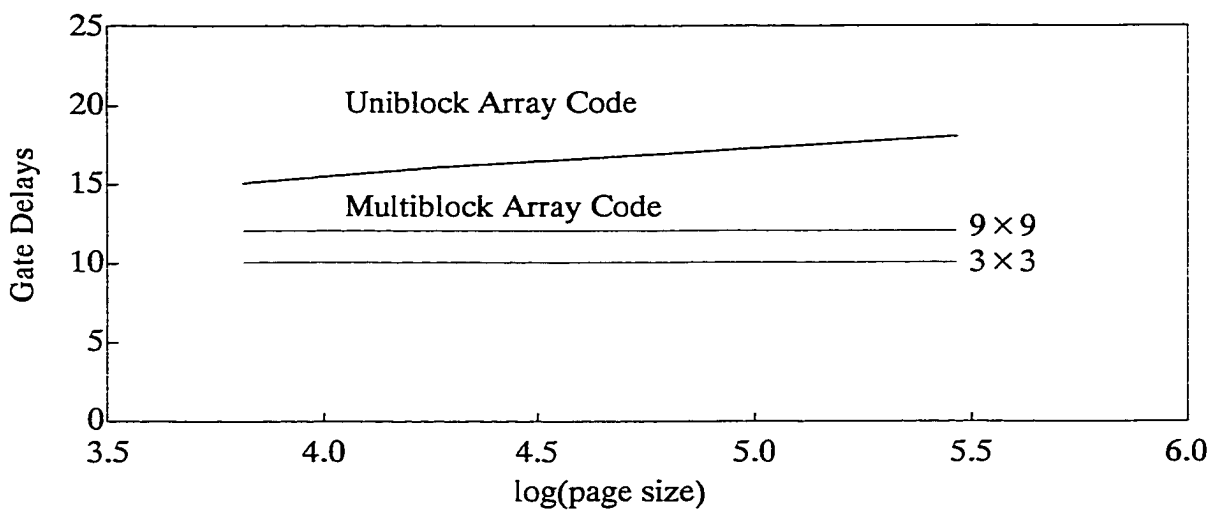


Figure 3.8: Decoding delay of uniblock and multiblock row and column array codes for different page sizes.

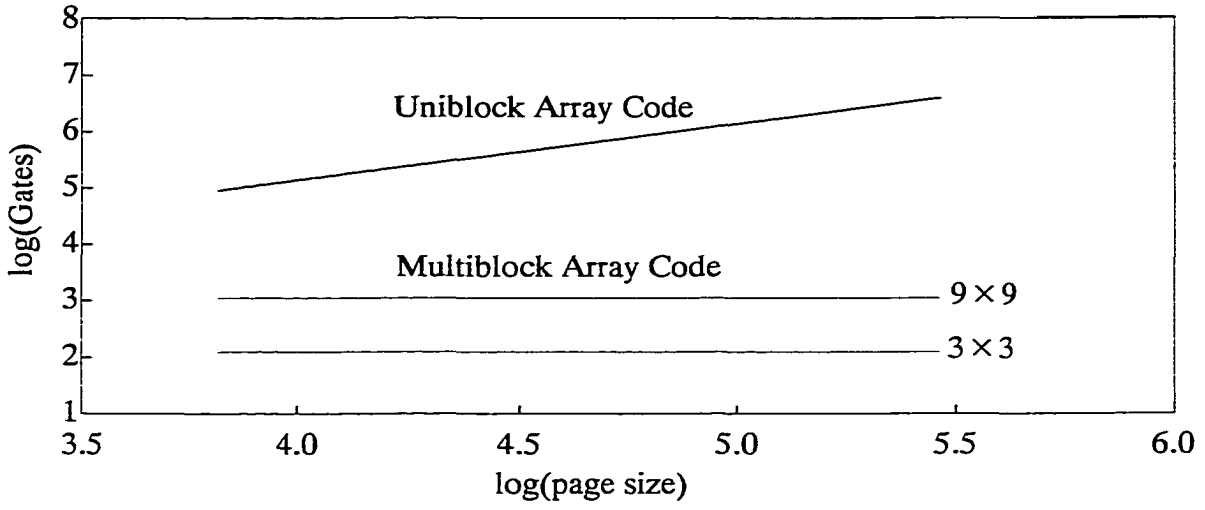


Figure 3.9: Hardware gate count of uniblock and multiblock row and column array codes for different page sizes.

3.2.6 Code Performance of Row and Column Array Codes

Simulations can be used to study the *CBER* performance of the uniblock and multiblock RAC array codes. Assume a Gaussian noise channel produces random single bit errors during the intensity detection process. A RAC code with $n_1 = 15$ and $n_2 = 8$ has parameters $(n, k, d) = (120, 98, 4)$ and $r = 0.8167$. Similarly, a RAC code with $n_1 = 10$ and $n_2 = 8$ has parameters $(n, k, d) = (80, 63, 4)$ and $r = 0.8167$. Each code supports detection of 2 errors and correction of 1 error. The probability, $P_{correct}$, that no error, e, passes through the decoder can be expressed as the sum of probabilities of 0, 1 and 2 errors in terms of the *RBER* and size of the array code parameter n:

$$P_{correct} = \sum_{e=0}^{2} \binom{n}{e} RBER^{e}(1 - RBER)^{n-e}. \tag{3.3}$$

The probability that all bits are correctly decoded, $P_{correct}$, can also be expressed in terms of *CBER* as

$$P_{correct} = (1 - CBER)^{n}. \tag{3.4}$$

The *CBER* can be expressed in terms of *n*, *e* and *RBER* by equating Equations 3.3 and 3.4 and solving for *CBER*.

Figure 3.10 shows the code performance of row and column codes of different code lengths on single bit random errors. The code with the smaller code length provides higher error correction capability. Codes with smaller block sizes tend to have more opportunities to detect and correct errors without the number of errors exceeding the error correcting capability of the code than codes of larger block size. However, codes with smaller code lengths are not as capacity efficient as codes of larger code lengths.

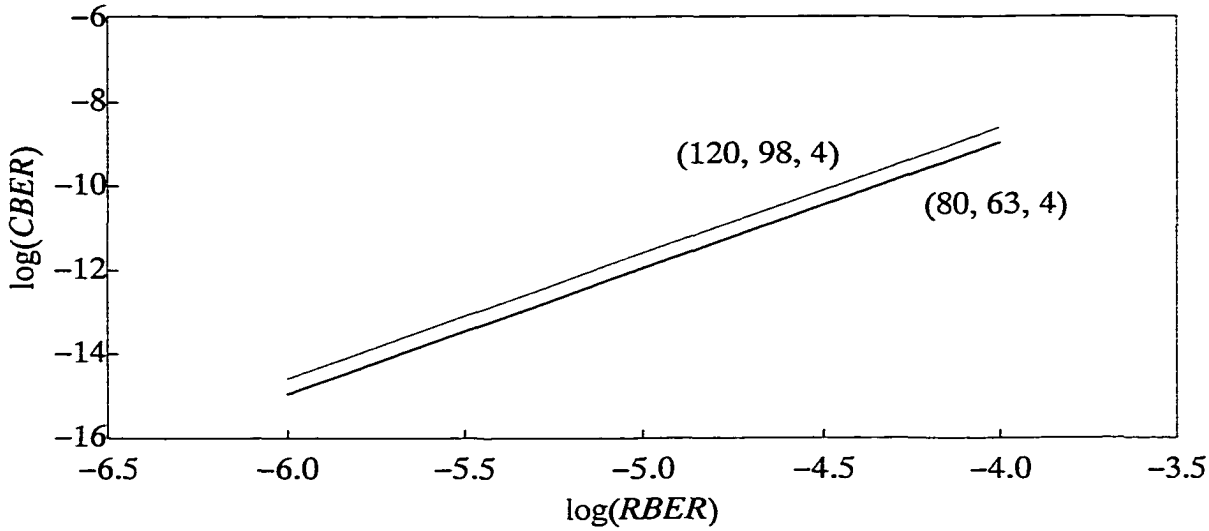


Figure 3.10: *CBER* versus *RBER* for row and column array codes on single bit random errors.

For a *RBER* value of 10^{-5}, the code performance of the uniblock and multiblock codes as page size varies is shown in Figure 3.11. For the uniblock array code, the *CBER* varies from approximately 10^{-8} to 10^{-6}. The multiblock array code is shown to have *CBER* values of 10^{-14} and 10^{-12}. While the uniblock array code is limited to small block size, the multiblock array code can be used on an entire page.

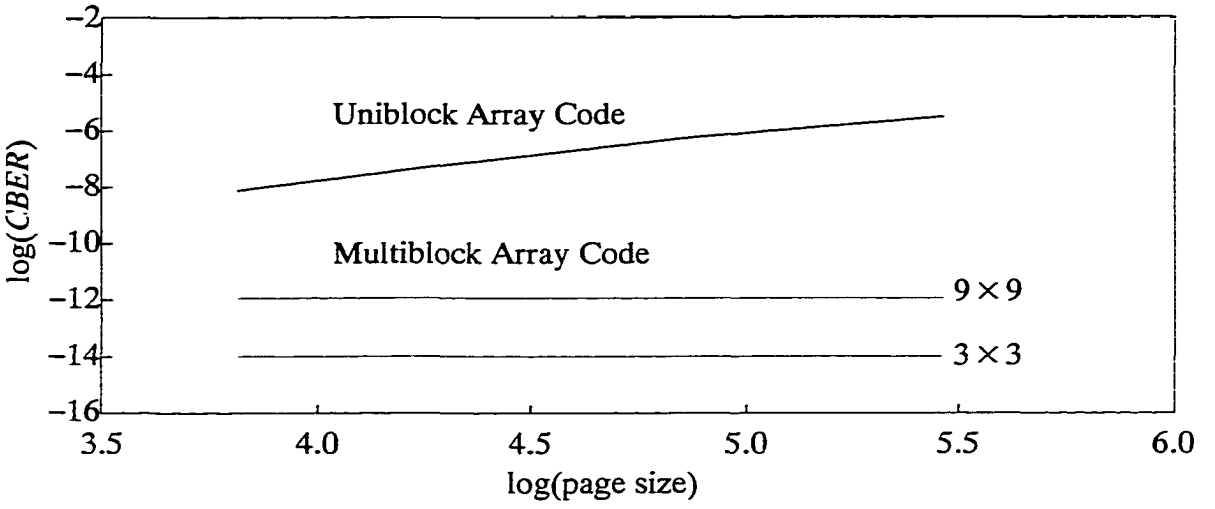


Figure 3.11: Performance of uniblock and multiblock row and column array codes for different page sizes ($RBER = 10^{-5}$).

For a given *RBER* (e.g., 10^{-5}) and a desired *CBER* (e.g., 10^{-12}), the corresponding value of n can be obtained by equating Equations 3.3 and 3.4 and solving for n. The code lengths which provide a $CBER \leq 10^{-12}$ are shown in Figure 3.12. The *RBER* are representative of values observed in holographic memory system demonstrations by different research groups [BUR95, BUR98, GOE95, HEA94, PSA96].

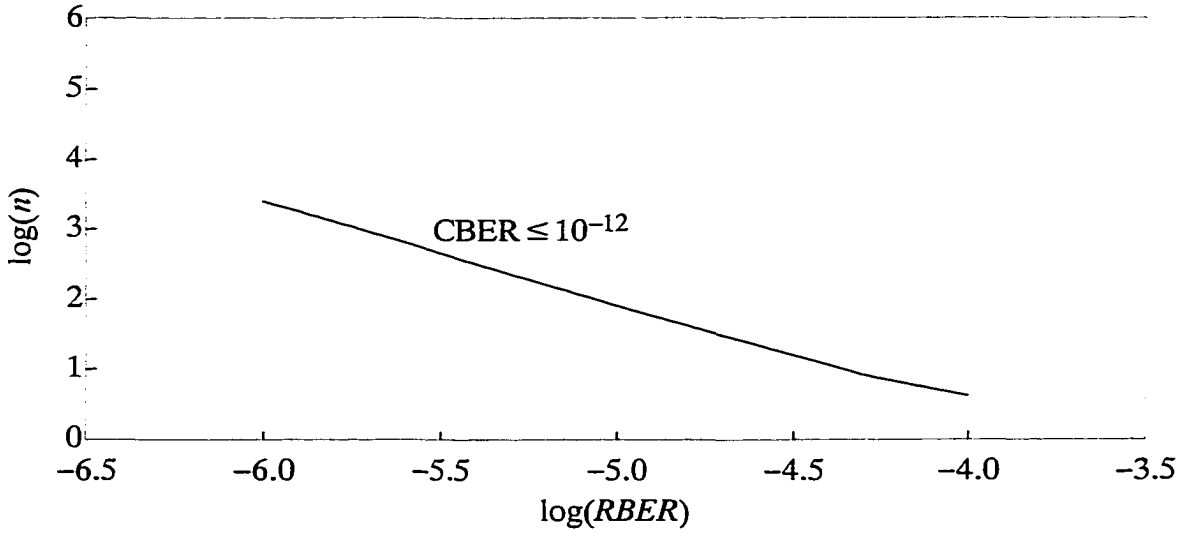


Figure 3.12: Row and column array code sizes which provide $CBER \leq 10^{-12}$.

3.3 Analysis of Wing Array Codes

A 2–D information bit array can be arranged to form a right triangular shaped (i.e., a wing) code. Wing codes are shown in Figure 3.13. The parity check bits are generated along the hypotenuse of the right triangular information bit array.

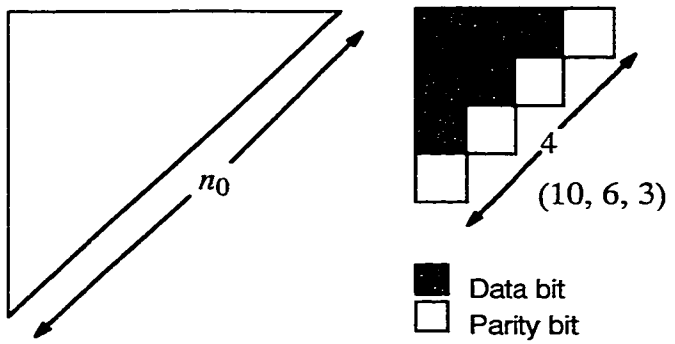


Figure 3.13: General and (10, 6, 3) wing codes.

The parameters for the general wing code shown in Figure 3.13 are given by the following expressions.

Length of code: $$n = \frac{n_0(n_0 + 1)}{2}$$

Information bits: $$k = \frac{n_0(n_0 - 1)}{2}$$

Hamming distance: $$d = 3$$

Code rate: $$r = \frac{k}{n} = \frac{n_0 - 1}{n_0 + 1}$$

Coding overhead: $$n - k = n_0$$

Number of errors that can be detected: $t = 1$

Number of errors that can be corrected: $c = 1$

Type of errors that can be corrected: random

The wing code may be used to perform error detection and correction (i.e., give protection) on the corners or lines of data that bisect the 90° angles included by the corners

of a page of data. Page regions not protected by the wing code require the use of a different code for protection. Moreover, random single bit errors cannot be expected to occur on a page of data only at a corner or on a line bisecting the 90° angle included by the corner. These observations regarding wing codes limit their usefulness when compared to the protection provided by row and column array codes. As shown in Figure 3.14, two wing codes can be arranged to protect two corners or lines of data that bisect the 90° angles included by the corners of a rectangular shaped page of data. These rectangular shaped pages can be made from a pair of wing codes having size $\frac{n_0(n_0 + 1)}{2}$.

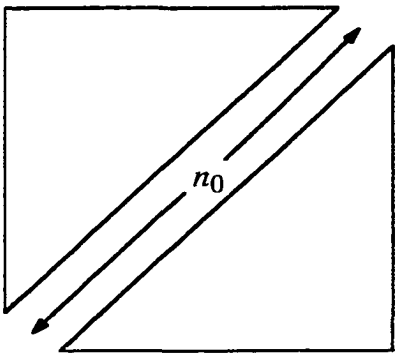


Figure 3.14: Two wing codes used for a rectangular shaped region.

The error correcting capability of a wing code pair can be exceeded as the page increases. For applications involving an entire page of data which may be as large as 10^6 bits, a multiblock strategy provides a practical solution. Assume we decide to detect and correct errors using the multiblock code shown in Figure 3.15. This multiblock code is a $m_1 \times m_2$ array of pairs of wing code blocks with each wing code having a hypotenuse of size n_0. If one error occurs in each code block, $n_0 = 15$ and the page size is 510×512 bits, as many as 2176 errors can be corrected. The parameters of this multiblock code are given by the following expressions.

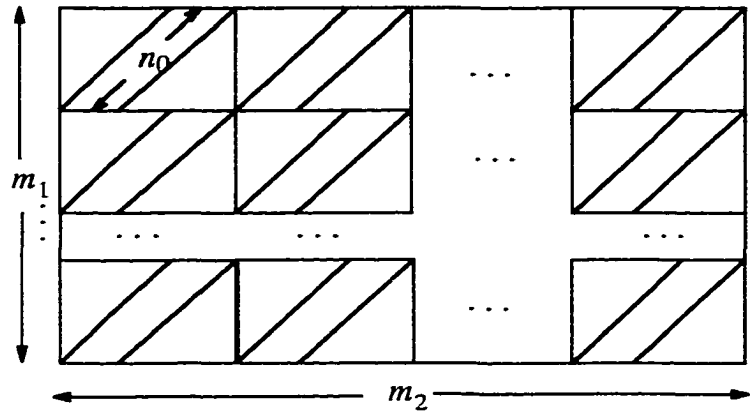


Figure 3.15: Multiblock wing array code pairs for rectangular shaped page.

The multiblock wing code shown in Figure 3.15 has parameters given by the following expressions.

Length of code: $n = m_1 m_2 n_0 (n_0 + 1)$

Information bits: $k = m_1 m_2 n_0 (n_0 - 1)$

Hamming distance: $d = 3$

Code rate: $r = \frac{k}{n} = \frac{n_0 - 1}{n_0 + 1}$

Coding overhead: $n - k = 2 m_1 m_2 n_0$

Number of errors that can be detected in each pair: $t = \{t_{\min} = 2, t_{\max} = 2 m_1 m_2\}$

Number of errors that can be corrected in each pair: $c = \{c_{\min} = 2, c_{\max} = 2 m_1 m_2\}$

Type of errors that can be corrected: random

3.3.1 Data Encoding for Wing Array Codes

Data encoding consists of computing the parity bit checks across the hypotenuse of a right triangular shaped information bit array. Using odd parity, Figure 3.16 shows the parity bit checks are composed of bits located on the hypotenuse of a right triangular information bit array. Each parity bit checks both a row and a column of the data array.

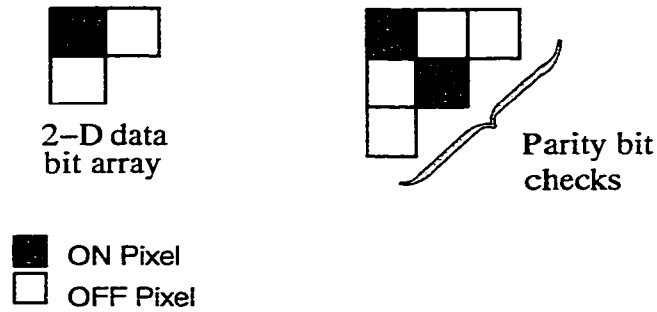


Figure 3.16: Example of data encoding applied to a wing array code.

3.3.2 Data Decoding for Wing Array Codes

When a page of data is stored or retrieved, random single bit errors can be introduced into the data page. The decoding process involves latching a page of data received at the photodetector array, computing a syndrome, detecting and correcting errors. For the wing code, an error is identified by two different bit failures in the syndrome. Using odd parity, part of the decoding process is shown in Figure 3.17.

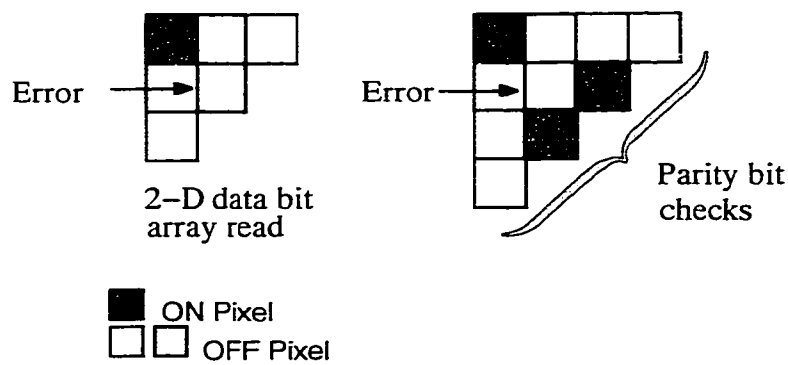


Figure 3.17: Example of data decoding applied to a wing array code.

3.3.3 Encoder for Wing Array Codes

The general wing code shown in Figure 3.13 requires a data encoding process that involves the same steps used in encoding the row and column array codes. However, the information bit array is right triangular in shape and the hypotenuse of length n_0 bits is

composed of the parity bit checks. Moreover, each parity bit check is located on the hypotenuse of the right triangular shape code block and checks both a row and a column of the information bit array. Using trees of $n_0 - 1$ XOR–gates and assuming odd parity, the syndrome can be computed in $\lceil \log_2(n_0+1) \rceil$ gate delays. The hardware needed for computing the syndrome is $H_{we} = n_0(n_0 - 1)$ gates and the corresponding time delay is $T_{we} = \lceil \log_2(n_0 + 1) \rceil$ gate delays.

3.3.4 Decoder for Wing Array Codes

The decoder for the general wing array code shown in Figure 3.13 uses four steps to perform the decoding process:

1. Latch data
2. Compute syndrome
3. Detect errors
4. Correct error.

Figure 3.6 shows hardware which can be used to accomplish step 1 of the algorithm. The time delay for latching the data is $T_l = 6$ gate delays and the required hardware is $H_{wdl} = \frac{9n_0(n_0 - 1)}{2}$ gates.

The syndrome consists of bits located on the hypotenuse of the right angle triangular code with each bit of the syndrome checking a row and a column of the data array. To accomplish step 2, an all parallel implementation can use trees of $n_0 - 1$ XOR–gates to compute the syndrome in $\lceil \log_2(n_0 + 1) \rceil$ gate delays. Figure 3.18 shows some of the hardware needed. The hardware for this implementation is $H_{wds} = n_0(n_0 - 1)$ gates and the time delay is $T_{wds} = \lceil \log_2(n_0 + 1) \rceil$ gate delays.

The error status of the system is obtained by checking several conditions. If two different bit positions of the syndrome indicate an error, we assume a single error has been detected in the information bits and corrective action can proceed. In cases of two or more errors, this code cannot provide a corrective action. The system can read the page again or take other corrective action when a no–corrective–action–taken condition occurs.

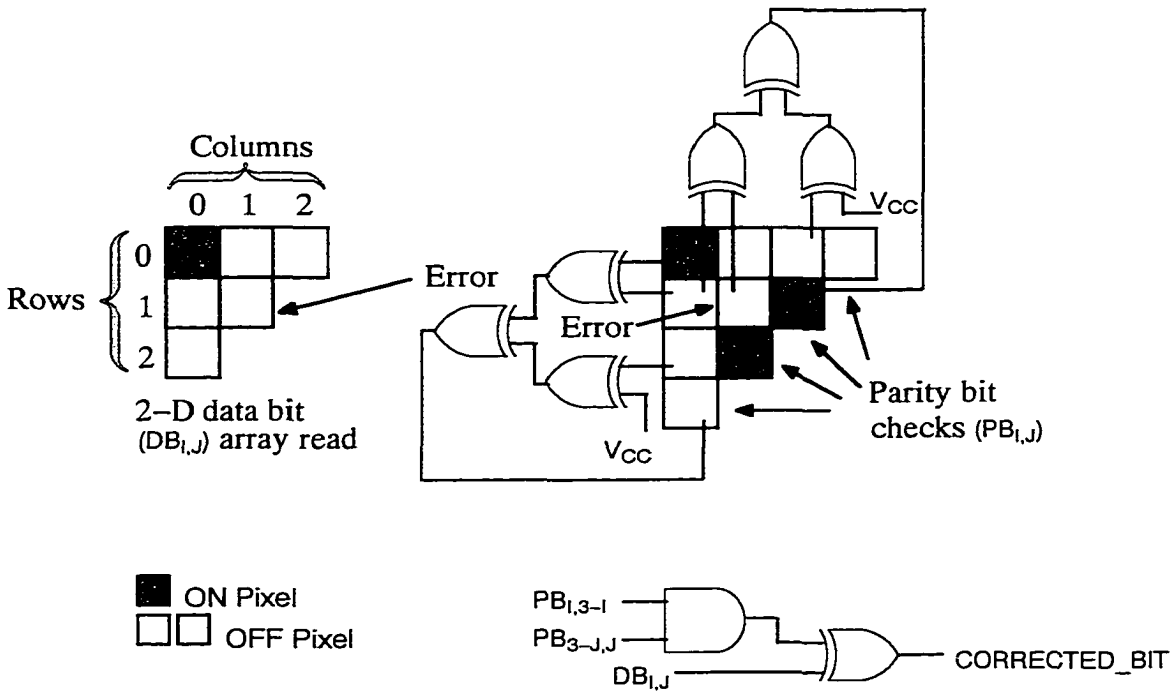


Figure 3.18: Hardware used to compute syndromes, detect and correct errors for a wing array code.

Error detection and correction are performed in the last two steps of the decoding process. Some of the required hardware is shown in Figure 3.18. A data bit in error located in the triangular information bit array is detected by the intersection of two different bit failures in the syndrome. Using corresponding error correction logic, the data bit in error can be corrected (i.e., inverted). If three or more different bit failures have occurred in the syndrome, an uncorrectable error has occurred. Two stages of AND–gates connected to trees of OR–gates can be used to test for the uncorrectable error condition. For error detection

and correction, the hardware needed is $H_{wdc} = n_0(n_0 - 1) + \frac{n_0!}{2(n_0 - 3)!} - 1$ gates. If an uncorrectable error has occurred, the time delay encountered is $T_{wdc} = 2 + \left\lceil \log_2 \frac{n_0!}{3!(n_0 - 3)!} \right\rceil$ gate delays otherwise $T_{wdc} = 2$ gate delays.

The overall delay encountered and hardware required is obtained by combining the appropriate values from each step of the decoding process. If an uncorrectable error has not occurred, the decoding delay for the wing array code is $T_{wd} = T_l + T_{wds} + T_{wdc} = \lceil \log_2(n_0 + 1) \rceil + 8$ gate delays otherwise $T_{wd} = \lceil \log_2(n_0 + 1) \rceil + \left\lceil \log_2 \frac{n_0!}{3!(n_0 - 3)!} \right\rceil + 8$ gate delays. The decoding hardware needed is $H_{wd} = H_{wdl} + H_{wds} + H_{wdc} = \frac{13n_0(n_0 - 1)}{2} + \frac{n_0!}{2(n_0 - 3)!} - 1$ gates.

3.3.5 Encoding and Decoding Multiblock Wing Array Codes

Hardware requirements and associated time delays for the multiblock wing code shown in Figure 3.15 are considered in this section. We assume the page of data have been correctly read when one or no error occurs in any wing code component (i.e., code block) component making up the entire multiblock code. Each wing code block needs encoding and decoding hardware as well as decoding hardware for generation of an error condition status (e.g., correct, correctable and uncorrectable).

A block serial implementation can be designed to encode and decode one wing code block after another sequentially. Either the decoding halts when a code block is detected with an uncorrectable error condition status or the decoding process proceeds and corrective action is taken, when a code block is detected with a correctable error condition status. The page of data is considered to have been correctly read and is sent on to the next stage of the

system when each code block has displayed an error condition status of correct. In sections 3.3.3 and 3.3.4, expressions were derived for the variables T_{we}, T_{wd}, H_{we} and H_{wd} and these variables are used in the expressions shown below. The encoding delay, T_{mwes}, decoding delay, T_{mwds}, encoding hardware needed, H_{mwes}, and decoding hardware needed, H_{mwds}, for the block serial implementation of the multiblock wing code are give by the following expressions.

Encoding delay: $T_{mwes} = 2m_1m_2T_{we}$

Decoding delay: $T_{mwds} = 2m_1m_2T_{wd}$

Encoding hardware: $H_{mwes} = H_{we}$

Decoding hardware: $H_{mwds} = H_{wd}$

A block parallel implementation can be designed to encode and decode wing code components in parallel. All code components can be processed in parallel in one pass. Alternately, some fraction of all the code components can be processed on each pass until the process is completed. Assume all component codes are processed in parallel in one pass. Let T_{wgA} be the global–AND–gate delay and H_{wgA} be the decoding hardware needed. For encoding and decoding, the time delays and hardware needed are given by the following expressions.

Encoding delay: $T_{mwep} = T_{we}$

Decoding delay: $T_{mwdp} = T_{wd} + T_{wgA}$

Encoding hardware: $H_{mwep} = 2m_1m_2H_{we}$

Decoding hardware: $H_{mwdp} = 2m_1m_2H_{wd} + H_{wgA}$

The advantages of the multiblock wing array code over the uniblock wing array code are shown in Figures 3.19 and 3.20 with respect to timing delay, hardware demands and page size. Since small block sizes are used, the time delay associated with the multiblock code

is less for a given page size as shown in Figure 3.19. For a given page size, Figure 3.20 shows the multiblock code requires less hardware per block because of the smaller block sizes used.

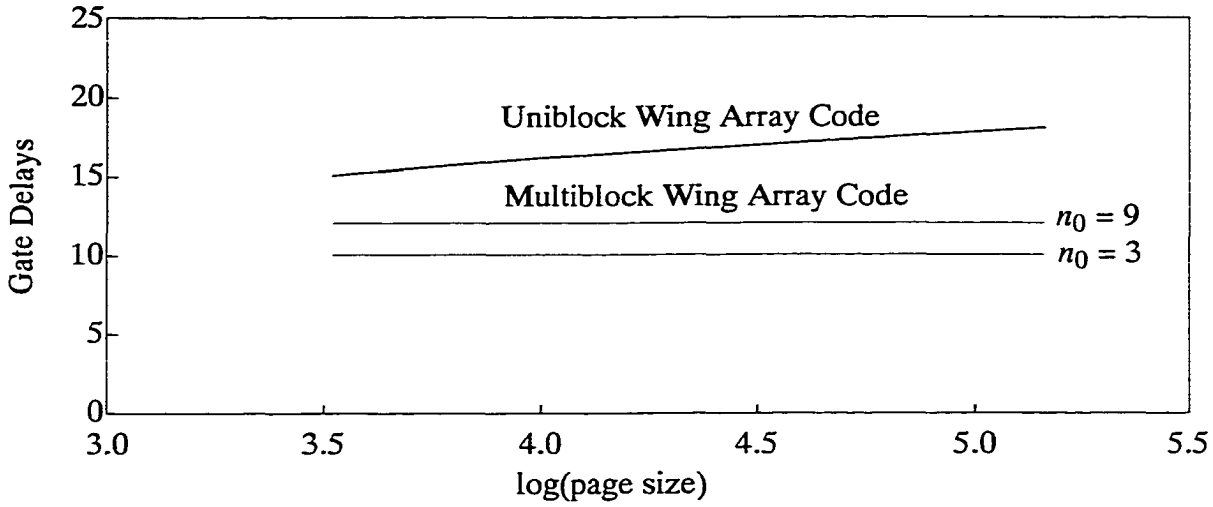


Figure 3.19: Decoding delay of uniblock and multiblock wing array codes for different page sizes.

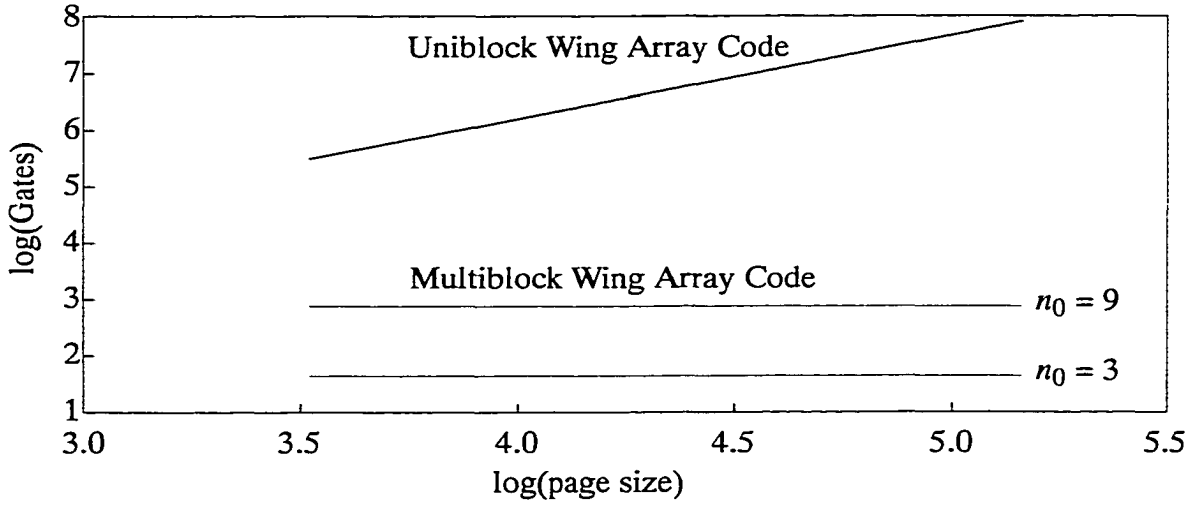


Figure 3.20: Hardware gate count of uniblock and multiblock wing array codes for different page sizes.

3.3.6 Code Performance of Wing Array Codes

The *CBER* performance of wing codes can be studied using code simulations. During the intensity detection process, assume a Gaussian noise channel produces random

single bit errors. A wing code with $n_0 = 6$ has parameters $(n, k, d) = (21, 15, 3)$ and $r = 0.7143$. Similarly, a wing code with $n_0 = 10$ has parameters $(n, k, d) = (55, 45, 3)$ and $r = 0.8182$. Each code supports detection and correction of 1 error. In terms of the *RBER*, array code parameter n and error, e, the probability, $P_{correct}$, that no error passes through the decoder can be expressed as the sum given by:

$$P_{correct} = \sum_{e=0}^{1} \binom{n}{e} RBER^{e}(1 - RBER)^{n-e}. \tag{3.5}$$

Using Equations 3.4 and 3.5, *CBER* can be expressed in terms of *RBER*, n and e.

Figure 3.21 shows the code performance of wing codes of different code lengths on single bit random errors. Even when lower values of *RBER* and codes of small code length are used, this code cannot achieve a $CBER \leq 10^{-12}$. For applications involving holographic data storage systems, these codes most likely cannot be used with other codes or in a multiblock structure when the $RBER \geq 10^{-6}$.

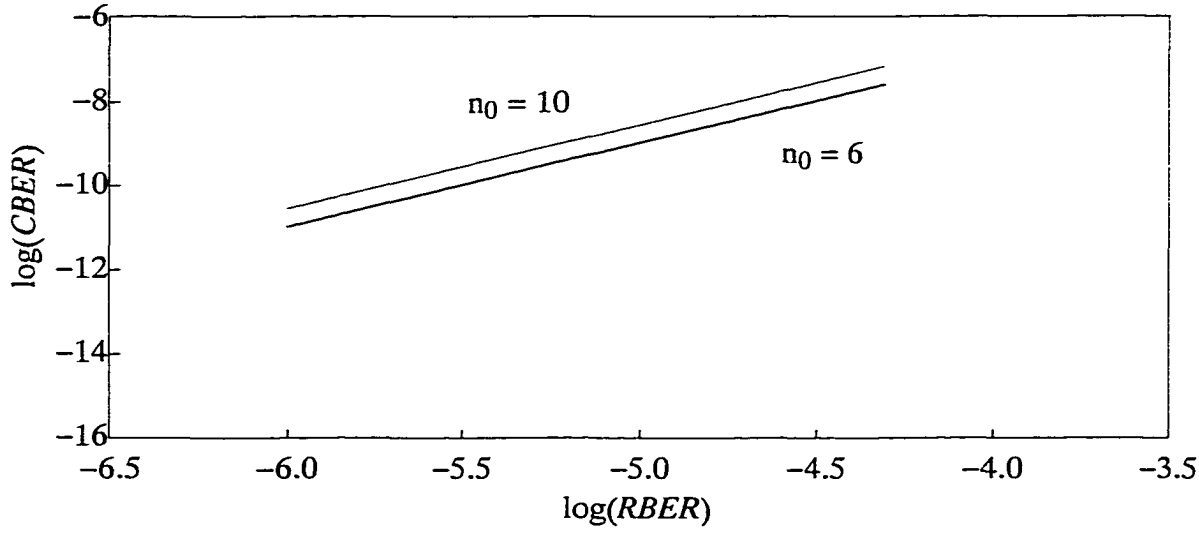


Figure 3.21: *CBER* versus *RBER* for uniblock wing array codes on single bit random errors.

3.4 Analysis of Array Codes for Correcting a Burst Error

Figure 3.22 shows a 2–D row and column code converted into a burst error correcting code [DAN85, FAR82a, MAB91]. Assume a burst of length k_2 occurs across columns.

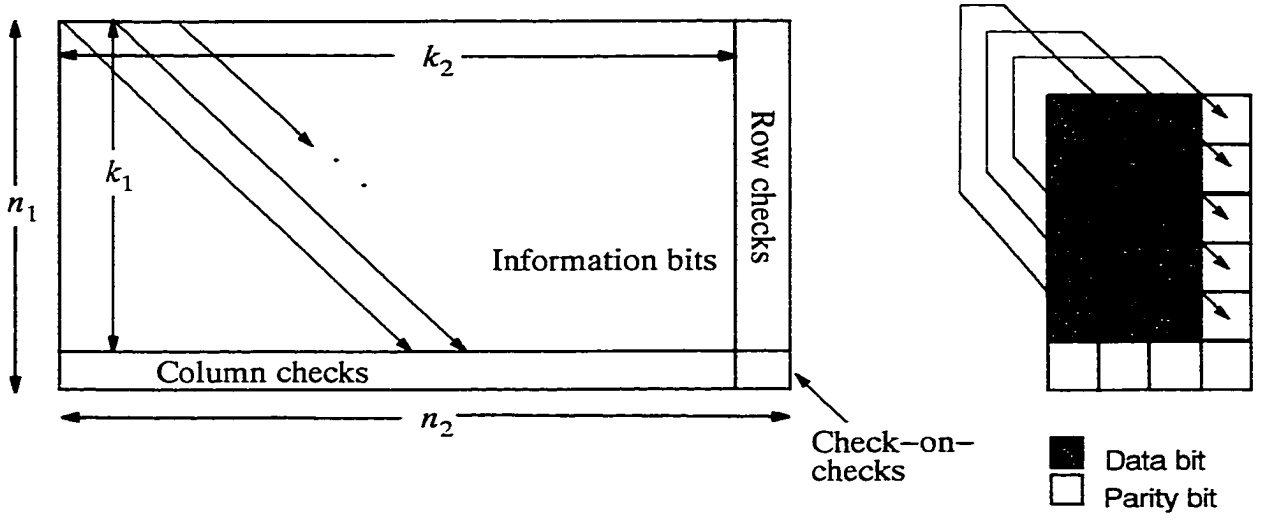


Figure 3.22: Single–burst error correcting codes and diagonal sequence pattern.

The bits of the burst error will alter specific diagonal parity bit checks and column parity bit checks. For an error burst of length k_2 bits, the information bit array must have parameters $k_1 \times k_2$ where $k_1 \geq 2(k_2 - 1)$ [FAR82a]. The parameters for this code are given by the following expressions.

Length of code: $n = n_1 n_2 = (k_1 + 1)(k_2 + 1) \geq [2(k_2 - 1) + 1](k_2 + 1)$

Information bits: $k = (n_1 - 1)(n_2 - 1) = k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = n_1 + n_2 - 1$

Number of errors that can be detected: $t = k_2$

Number of errors that can be corrected: $c = k_2$

Type of errors that can be corrected: burst

Moreover, a diagonal direction set is used because each bit in a correctable burst error is associated with distinct diagonal parity bit checks and column parity bit checks (i.e., redundant bits computed from a diagonal sequence). By using more powerful component codes or using 3–D array codes multiple bursts can be corrected [FAR92].

Applications with page sizes as large as 10^6 bits require the use of a multiblock array code strategy. A multiblock burst error correcting (BEC) code can be obtained using a $m_1 \times m_2$ array of $n_1 \times n_2$ row and column code blocks as shown in Figure 3.23. The parameters of this multiblock code are given by the following expressions.

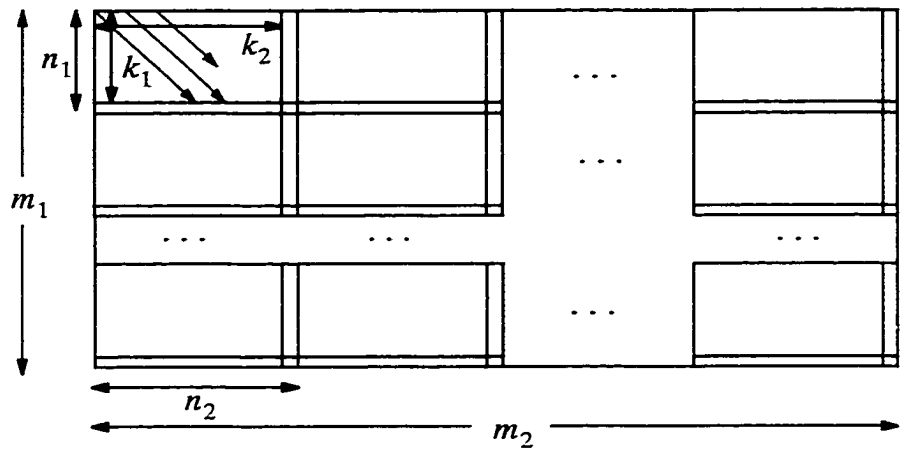


Figure 3.23: Multiblock burst error correcting array code.

Length of code: $n = m_1 m_2 (k_1 + 1)(k_2 + 1) \geq m_1 m_2 [2(k_2 - 1) + 1](k_2 + 1)$

Information bits: $k = m_1 m_2 (n_1 - 1)(n_2 - 1) = m_1 m_2 k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = m_1 m_2 (n_1 + n_2 - 1)$

Number of errors that can be detected: $t = \{t_{\min} = k_2, t_{\max} = k_2 m_1 m_2\}$

Number of errors that can be corrected: $c = \{c_{min} = k_2, c_{max} = k_2 m_1 m_2\}$

Type of errors that can be corrected: burst

3.4.1 Data Encoding for Burst Error Correcting Array Codes

The data encoding process involves the same steps used for encoding the row and column array code. However, each row parity bit check is computed in a diagonal direction as a diagonal parity bit check. Each column parity bit check is computed across a column as was done for the row and column array code. An even parity scheme is used for the encoding shown in Figure 3.24.

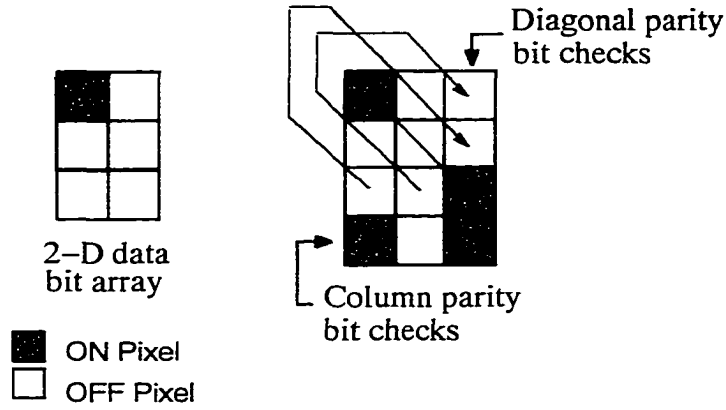


Figure 3.24: Example of data encoding applied to a burst error correcting array code.

3.4.2 Data Decoding for Burst Error Correcting Array Codes

A burst error can be introduced into the data page when a page of data is stored or retrieved. The decoding process follows essentially the same steps used by the row and column array code. Namely, these steps involve latching a page of data received at the photodetector array, computing the syndromes, detecting and correcting errors. The decoding shown in Figure 3.25 uses an even parity scheme. For the burst error correcting array code, a burst error is identified by bit failures in the column syndrome. A compare operation is performed between the column syndrome and a segment of the diagonal

syndrome equal in length to the column syndrome. If a match is not found, the diagonal syndrome is circularly shifted one bit position upward and the compare operation is repeated. A series of compare and circular shift operations are required to determine the row location of the burst error as described in the decoder section below.

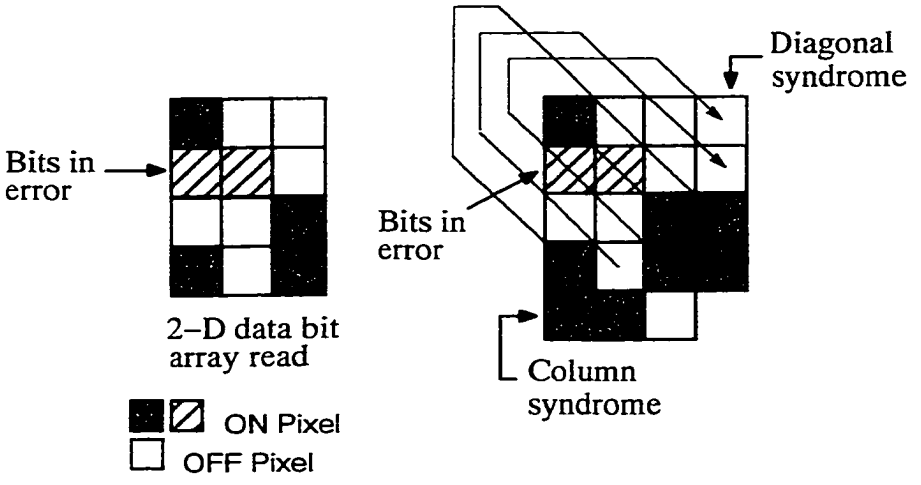


Figure 3.25: Example of data decoding applied to a burst error correcting array code.

3.4.3 Encoder for Burst Error Correcting Array Codes

Column parity bit checks and diagonal parity bit checks must be computed to accomplish the data encoding process for the burst error correcting array code shown in Figure 3.22. This encoding is similar to that used for encoding a row and column array code except diagonal parity bit checks replace the row parity bit checks. Trees of k_2-1 XOR–gates can be used to compute diagonal parity bit checks in $\lceil \log_2 k_2 \rceil$ gate delays and trees of k_1-1 XOR–gates can be used to compute column parity bit checks in $\lceil \log_2 k_1 \rceil$ gate delays. Therefore, the hardware needed for computing the syndromes is $H_{be} = (k_1 - 1)n_2 + (k_2 - 1)k_1$ gates and the corresponding encoding delay is $T_{be} = \max(\lceil \log_2 k_1 \rceil, \lceil \log_2 k_2 \rceil)$ gate delays.

3.4.4 Decoder for Burst Error Correcting Array Codes

The code in Figure 3.22 requires five steps to accomplish the decoding algorithm:

1. Latch data
2. Compute syndromes
3. Detect errors
4. Locate errors
5. Correct errors.

The control signals for the algorithm are shown in Figure 3.26. The steps of the algorithm occur sequentially. The positive edge of the LATCH_DATA pulse is used to latch a page of data into the decoder. The positive edge of the GET_DIAGONAL_SYNDROME_GET_ROW_BITS signal causes the diagonal syndrome and row enable bits to be loaded respectively into two shift registers. The detect, locate and correct error steps are initiated by the positive edge of the DETECT_LOCATE_CORRECT_ERROR pulse.

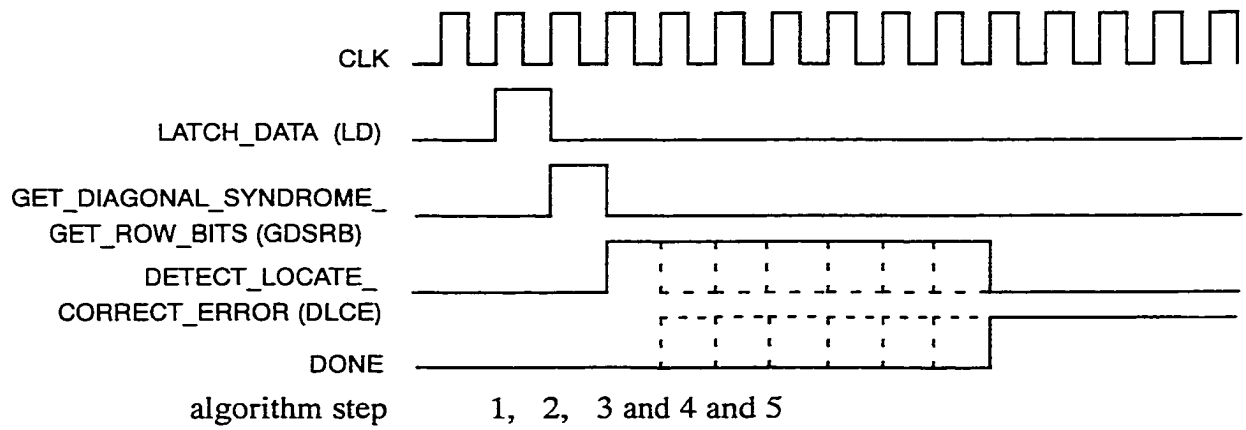


Figure 3.26: State machine timing diagram for burst error correcting array codes.

Figure 3.27 shows a state machine that can be used to produce the control signals. For a burst error k_2 bits in length, the maximum number of clock cycles required by the state machine to perform the five steps of the algorithm is $T_{bsm} = 2 + n_1$ clock cycles. Assuming the 4:1 multiplexer [NAT84] can be implemented using five gates, the counters [NAT84] can be implemented using ten gates per bit and the flip–flop consists of six gates, the hardware needed to implement the state machine is $H_{bsm} = 2[10\log_2(2^2)] + 10\lceil \log_2 n_1 \rceil + 29$ gates.

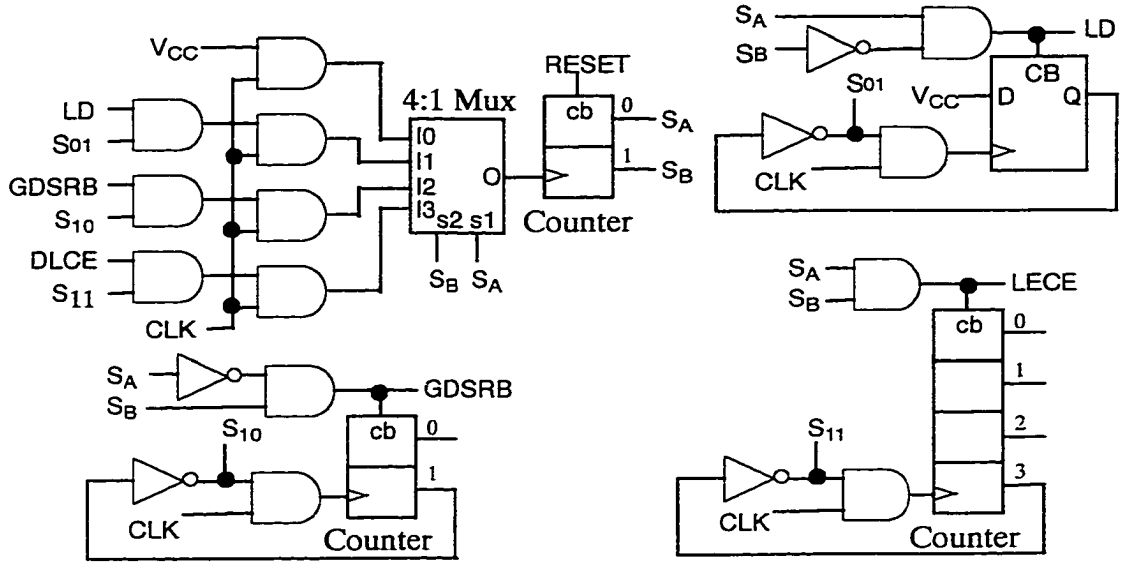


Figure 3.27: State machine hardware applied to burst error correcting array codes.

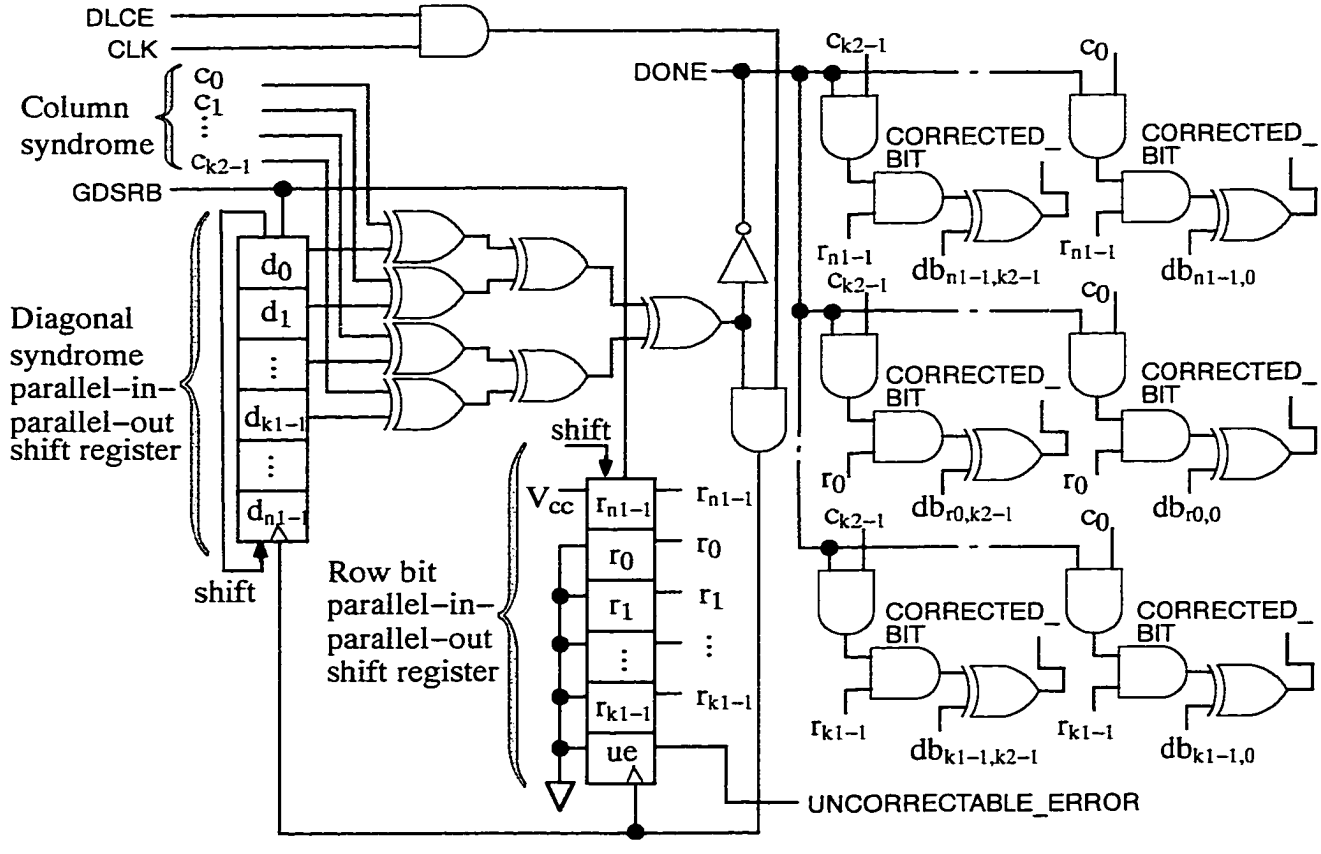


Figure 3.28: Hardware used for burst error detection and correction.

In the latch data step, an information bit array of size $n_1 \times n_2$ bits is latched. The same storage element shown in Figure 3.6 used for the row and column array code is required to latch the data. Therefore, the hardware needed is $H_{bdl} = 9n_1n_2$ gates and the time delay is $T_l = 6$ gate delays.

In a fashion similar to how the syndromes were computed for the row and column array code, trees of n_2 and n_1 XOR–gates can be used to compute respectively the diagonal and column parity bit checks in $\lceil \log_2 n_2 \rceil$ and $\lceil \log_2 n_1 \rceil$ gate delays. The hardware required and the delay encountered are respectively $H_{bds} = k_1n_2 + k_2n_1$ gates and $T_{bds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays.

To complete the detect, locate and correct error steps of the algorithm several tasks must be performed. These steps are accomplished with the hardware shown in Figure 3.28. One shift register is loaded with the diagonal syndrome and the other shift register is loaded with a bit pattern (i.e., a pattern with one bit set to one and the remaining n_1 bits set to zero) to enable one of the n_1 rows. The k_2 bit long column syndrome is compared to k_2 bits of the n_1 bit long diagonal syndrome. If the two syndromes do not match, the diagonal syndrome is circularly shifted upward one bit position, the row bit pattern is shifted downward one bit position and the syndromes are compared again. The comparisons of syndromes and circular bit shifts are repeated n_1 times provided the syndromes do not match. If the syndromes match on the i^{th} shift where $0 \leq i \leq k_1$, the row represented by i mod k_1 contains a burst error. The column syndrome, row bit pattern and done (i.e., match) signal enable error correction logic to correct the burst error. If the diagonal syndrome has been shifted $n_1 + 1$ times and the syndromes do not match, the bit set to one in the row enable bit pattern is used to indicate an uncorrectable error has been detected. Assuming the implementations for the diagonal syndrome shift register uses the hardware shown in Figure 3.29 and for the row bit shift register uses flip–flops consisting of six gates, the hardware requirement for error detection,

location and correction is $H_{bdc} = 9n_1 + 6n_1 + 2k_2 - 1 + 3 + 3k_2n_1 = n_1(3k_2 + 15) + 2k_2 + 2$ gates. If an uncorrectable error has not occurred, the maximum delay is $T_{bdc} = n_1$ clock cycles otherwise $T_{bdc} = n_1 + 1$ clock cycles.

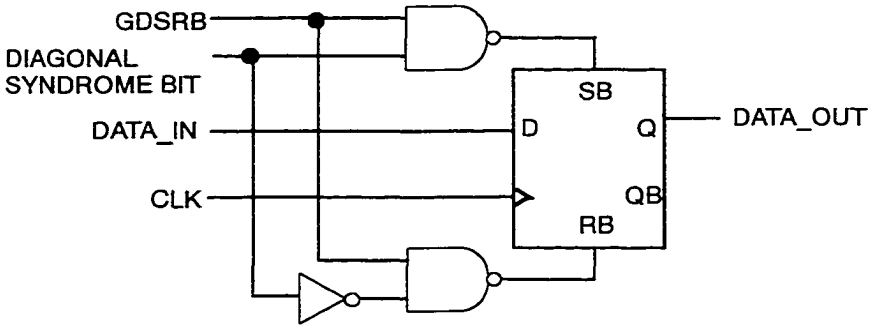


Figure 3.29: Hardware used for one bit stage of a shift register.

The hardware required for decoding the burst error correcting array code is $H_{bd} = H_{bsm} + H_{bdl} + H_{bds} + H_{bdc} = 10\lceil \log_2 n_1 \rceil + 9n_1n_2 + (4k_2 + 15)n_1 + k_1n_2 + 2k_2 + 71$ gates and the time delay is $T_{bd} = T_{bdl} + T_{bds} + T_{bdc} = 2 + n_1$ clock cycles. The time to compute the diagonal syndrome is $T_{bds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays and determines the minimum clock period or the reciprocal of the maximum clock frequency. If the minimum clock cycle is equivalent to T_{bds} gate delays, the time delay can be expressed in terms of gate delays. If an uncorrectable error has not occurred, the time delay is $T_{bd} = T_{bdl} + T_{bds} + T_{bdc} = (n_1 + 2)\max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays otherwise $T_{bd} = (n_1 + 3)\max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays.

3.4.5 Encoding and Decoding Multiblock BEC Array Codes

The time delay and hardware requirements of the multiblock BEC code shown in Figure 3.23 are obtained by repeating the analysis used for the multiblock RAC code. The same constraints regarding encoding and decoding hardware as well as needing additional hardware for the error condition status (e.g., correct and uncorrectable) still apply. A block serial and all parallel implementation are considered.

The block serial implementation encodes and decodes one component uniblock code after another sequentially as described in section 3.2.5. Expressions for the variables T_{be}, T_{bd}, H_{be} and H_{bd} were derived in sections 3.4.3 and 3.4.4 and are used in the expressions shown below. For a block serial implementation of the multiblock BEC code, the encoding delay, T_{mbes}, decoding delay, T_{mbds}, encoding hardware required, H_{mbes} and decoding hardware required, H_{mbds}, are given by the following expressions.

Encoding delay: $T_{mbes} = m_1 m_2 T_{be}$

Decoding delay: $T_{mbds} = m_1 m_2 T_{bd}$

Encoding hardware: $H_{mbes} = H_{be}$

Decoding hardware: $H_{mbes} = H_{bd}$

The block parallel implementation operates in the same way as described in section 3.2.5. For example, assume a single pass is used to encode and decode all component uniblock codes in parallel. Let T_{bgA} be the timing delay and H_{bgA} be the hardware needed for global–AND–gate. The encoding delay, T_{mbep}, decoding delay, T_{mbdp}, encoding hardware needed, H_{mbep} and decoding hardware needed, H_{mbdp}, are given by the following expressions.

Encoding delay: $T_{mbep} = T_{be}$

Decoding delay: $T_{mbdp} = T_{bd} + T_{bgA}$

Encoding hardware: $H_{mbep} = m_1 m_2 H_{be}$

Decoding hardware: $H_{mbdp} = m_1 m_2 H_{bd} + H_{bgA}$

Page size, time to process the data and hardware demands are used to compare the multiblock and uniblock array codes. Regarding page size, Figure 3.30 shows the multiblock code can process data per block in less time. The multiblock codes have a smaller time delay because the smaller block size used requires a tree of gates structure containing

less levels of gating. The hardware required per block as shown in Figure 3.31 is significantly less for multiblock array codes. The use of smaller code blocks by the multiblock codes requires a smaller hardware gate count for implementation than the uniblock code.

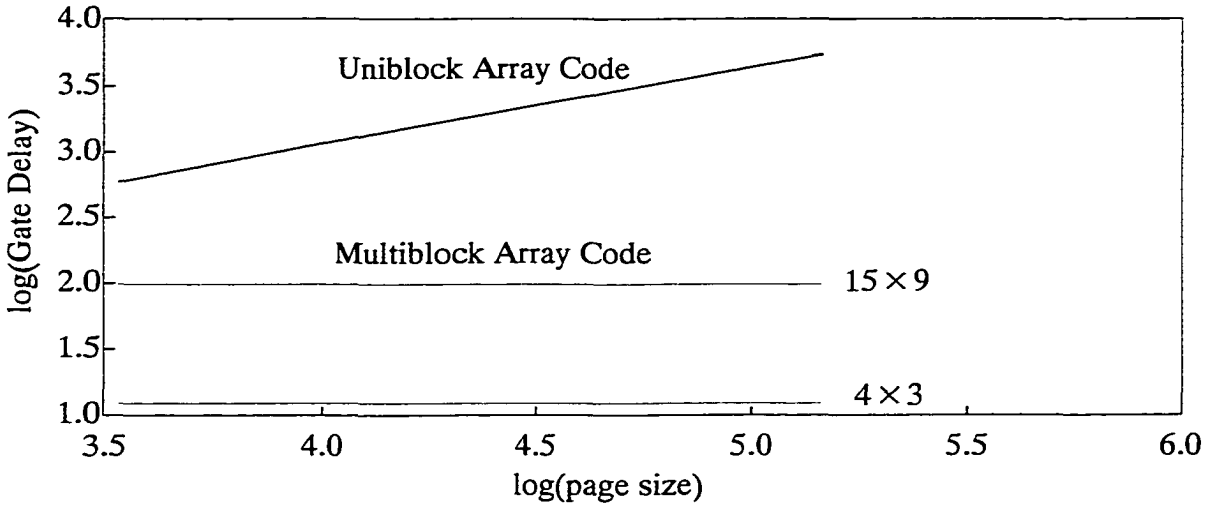


Figure 3.30: Decoding delay of uniblock and multiblock burst error correcting array codes for different page sizes.

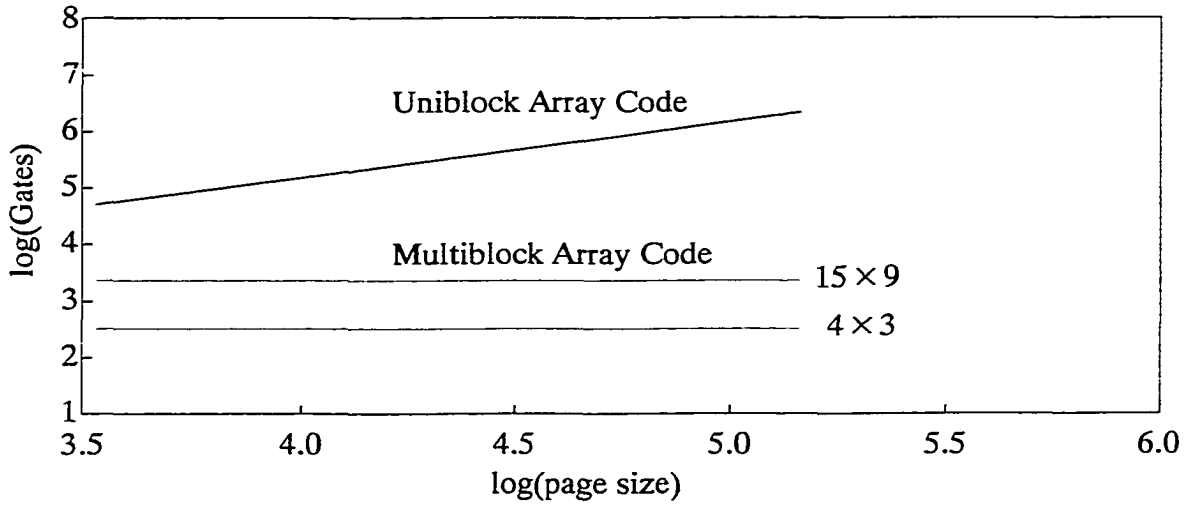


Figure 3.31: Hardware gate count of uniblock and multiblock burst error correcting array codes for different page sizes.

3.4.6 Code Performance of Burst Error Correcting Array Codes

To study the *CBER* performance of uniblock and multiblock burst error correcting array codes, code simulations can be used. Assume a burst error is caused during the storage or retrieval of a data page. For a burst error 2 bits in length, a burst error correcting array code with $n_1 = 3$, $n_2 = 3$ having parameters $(n, k, d) = (9, 4, 4)$ and $r = 0.4444$ can be used. A burst error 7 bits in length requires a burst error correcting array code with $n_1 = 15$ and $n_2 = 9$ having parameters $(n, k, d) = (135, 112, 4)$ and $r = 0.8296$. Each code supports detection and correction of a single burst error. Writing, $P_{correct}$, the probability that no error, e, passes through the decoder, as a sum of probabilities in terms of the *RBER* and size of the array code parameter n, the following expression is obtained.

$$P_{correct} = \sum_{e=0}^{k_2} \left(\frac{n(k_2 - 1)!}{e!(k_2 - e)!} \right) RBER^e (1 - RBER)^{n-e}, \qquad e > 0. \qquad (3.6)$$

The *CBER* can be expressed in terms of e, n and *RBER* using Equations 3.4 and 3.6.

Figure 3.32 shows the code performance of burst error correcting array codes of different code lengths. For lower values of *RBER* and codes of small code length, this code can almost achieve a $CBER \leq 10^{-12}$. However, for VHDS applications with $RBER \geq 10^{-6}$, these codes most likely cannot be used in either a multiblock or uniblock structure to achieve a $CBER \leq 10^{-12}$. Since, these codes are adapted versions of the row and column array codes, they cannot achieve a higher error correcting capability than the corresponding row and column array codes.

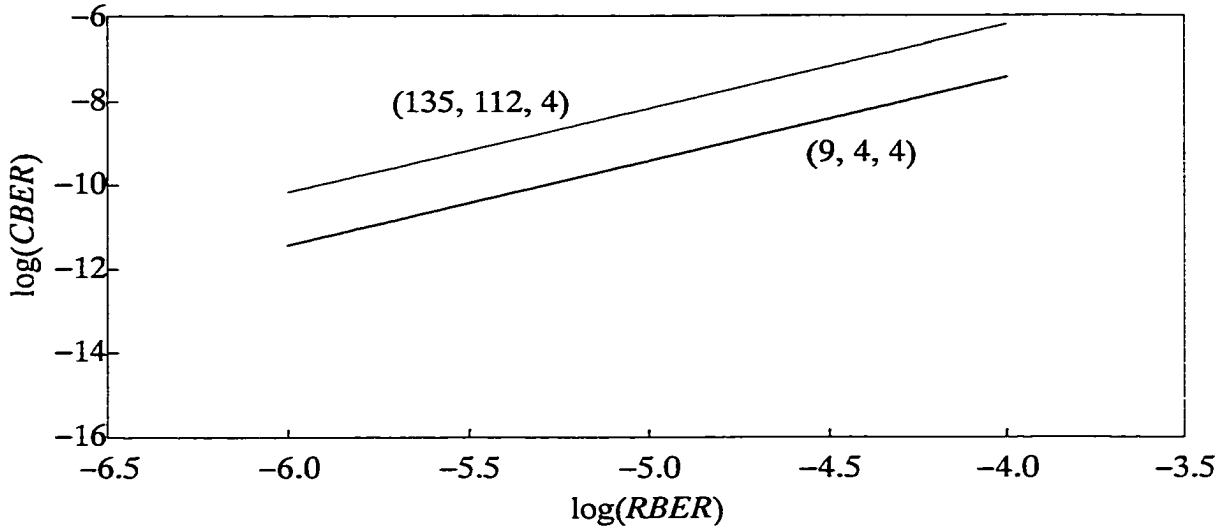


Figure 3.32: *CBER* versus *RBER* for single–burst error correcting array codes.

3.5 Analysis of a Cluster Error Correcting Array Code

The row and column array code shown in Figure 3.1 can be converted into a cluster error correcting array code [BLA94, BLA98, FAR82b]. Assume an array code consisting of information and parity bits of size $n_1 \times n_2$ contains a $b_1 \times b_2$ rectangular cluster of e errors, with constraints $1 \leq e \leq b_1 b_2$, $n_1 \geq 2b_1 b_2 - b_1$, $n_2 \geq 2b_1 b_2$, b_1 divides n_1 and b_2 divides n_2 [BLA94]. The parameters for this code are given by the following expressions.

Length of code: $n = n_1 n_2 \geq (2b_1 b_2 - b_1)2b_1 b_2$

Information bits: $k = (n_1 - 1)(n_2 - 1) = k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = n_1 + n_2 - 1$

Number of errors that can be detected: $t = b_1 b_2$

Number of errors that can be corrected: $c = b_1 b_2$

Type of errors that can be corrected: cluster

A multiblock cluster error correcting (CEC) array code can be applied to application with page sizes as large as 10^6 bits. Using a $m_1 \times m_2$ array of $n_1 \times n_2$ row and column array code as shown in Figure 3.3 constrained such that $1 \leq e \leq b_1 b_2$, $n_1 \geq 2b_1 b_2 - b_1, n_2 \geq 2b_1 b_2$, b_1 divides n_1 and b_2 divides n_2 [BLA94], a multiblock cluster error correcting array code is obtained. The following expressions describe the parameters for this code.

Length of code: $n = m_1 m_2 n_1 n_2 \geq m_1 m_2 (2b_1 b_2 - b_1) 2b_1 b_2$

Information bits: $k = m_1 m_2 (n_1 - 1)(n_2 - 1) = m_1 m_2 k_1 k_2$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = m_1 m_2 (n_1 + n_2 - 1)$

Number of errors that can be detected: $t = \{t_{\min} = b_1 b_2, t_{\max} = b_1 b_2 m_1 m_2\}$

Number of errors that can be corrected: $c = \{c_{\min} = b_1 b_2, c_{\max} = b_1 b_2 m_1 m_2\}$

Type of errors that can be corrected: cluster

The $n_1 \times n_2$ array is cyclic in both directions with the topology of a 2–D torus. These array codes allow correction of end–round cluster errors. In Figure 3.33, $n_1 = n_2 = 8$, $b_1 = b_2 = 2$ and a collection of on pixels is used to indicate the location of bits in error forming a cluster of size $b_1 \times b_2$.

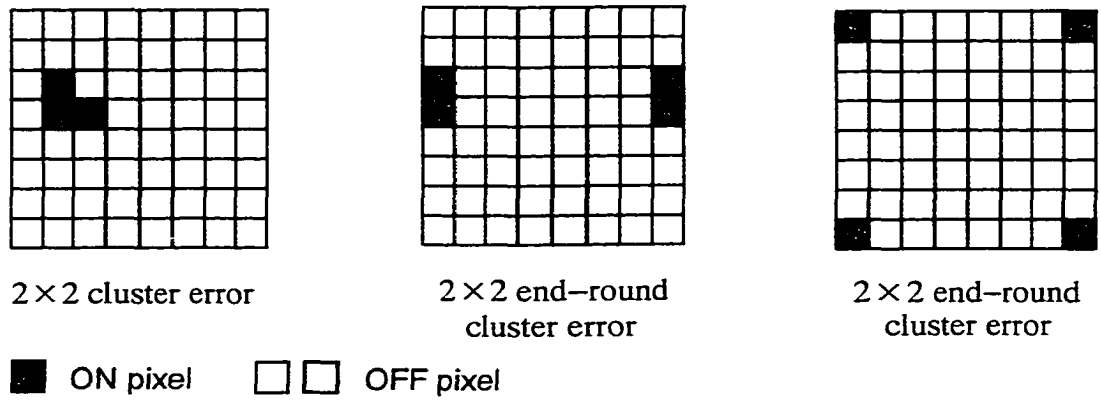


Figure 3.33: Examples of different arrangements of 2×2 cluster errors.

3.5.1 Data Encoding for a Cluster Error Correcting Array Code

For a given data array A, the encoding process involves calculating a single parity check bit across each row, across each column and staggering the data array on columns and then on rows. Specifically, a code word of size $n_1 \times n_2$ is formed by appending a single parity check bit to each row and column of a $(n_1 - 1) \times (n_2 - 1)$ information bit array I. To each row r_i of A, where $0 \leq i \leq n_1 - 1$, apply a circular shift to the right equal to $b_2(i \bmod b_1)$ bits to form a row–staggered array S^r. For each column c_j of S^r, where $0 \leq i \leq n_2 - 1$, apply a circular shift downward equal to $b_1(j \bmod b_2)$ bits to form a staggered array S. In this way, S, a staggered version of A, first on rows then on columns, is obtained for storage. Part of the encoding process is shown in Figure 3.34 for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$. To form S^r, the even rows of A are not rotated (i.e., circularly shifted) because $b_2(i \bmod 2) = 0$, whereas odd rows are rotated right two bits because $b_2(i \bmod b_1) = 2(1) = 2$ bits. To form S, the odd columns of S^r are rotated downward two bits because $b_1(j \bmod 2) = 2(1) = 2$ bits, whereas the even columns are not rotated because $b_1(j \bmod 2) = 0$. Data staggering can be implemented using shift registers configured to perform circular shifts for odd rows and columns in array A.

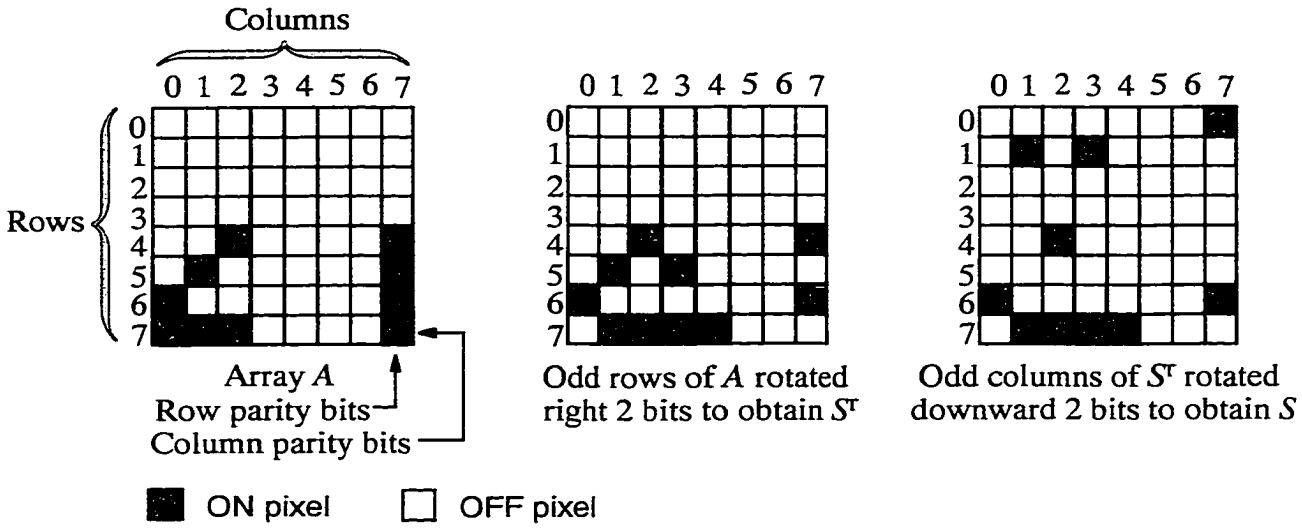


Figure 3.34: Example of data encoding applied to data array A yielding staggered arrays S^T and S for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$.

3.5.2 Data Decoding for a Cluster Error Correcting Array Code

Since the code word S may be corrupted during storage or retrieval, the retrieved array R may not equal S. During the decoding process, a cluster error of size at most $b_1 \times b_2$ can be corrected. Each column c_j of R is rotated upwardly an amount equal to $b_1(j \bmod b_2)$ bits to form a column–destaggered array D^c. Then each row r_i of D^c is rotated left an amount equal to $b_2(i \bmod b_1)$ bits to form a destaggered array D. Figure 3.35 shows array R corrupted by a 2×2 cluster error in the central region of R and how data destaggering proceeds. After data destaggering has been completed, the errors are distributed such that no more than one error occurs in a row or column.

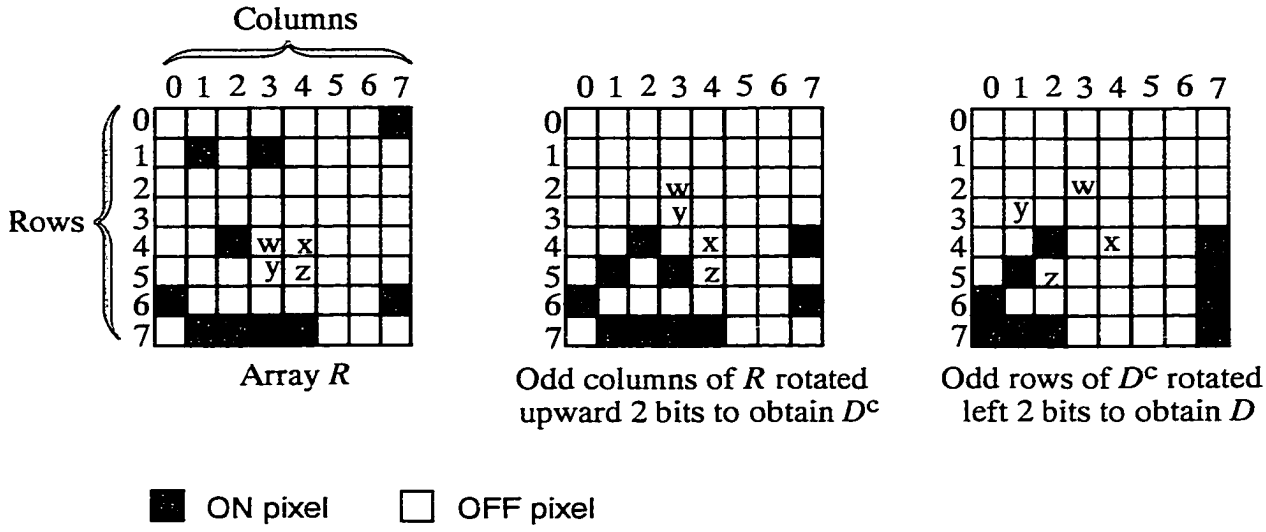


Figure 3.35: Example of data decoding applied to retrieved array R yielding destaggered arrays D^c and D for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$.

The destaggered array D is the input from which the vertical syndrome $\boldsymbol{v}$ (i.e., the exclusive–OR of the rows of D) and horizontal syndrome $\boldsymbol{h}$ (i.e., the exclusive–OR of the columns of D) are determined. If $d_{i,j}$ represents the elements of D, the vertical syndrome is given by

$$v_j = \bigoplus_{l=0}^{n_1-1} d_{l,j}, \qquad 0 \le j \le n_2 - 1 \tag{3.7}$$

and the horizontal syndrome is given by

$$h_i = \bigoplus_{l=0}^{n_2-1} d_{i,l}, \qquad 0 \le i \le n_1 - 1. \tag{3.8}$$

If array R contains no errors, both syndromes of array D will be zero. If array R contains a cluster of e errors, $1 \le e \le b_1 b_2$, the horizontal and vertical syndromes will each have weight e. The constraint $n_2 \ge 2b_1 b_2$ insures the vertical syndrome can be represented as a burst of length at most $b_1 b_2$ having a cyclic run of at least $b_1 b_2$ zeros. After the run of at least $b_1 b_2$, the first nonzero element in the vertical syndrome can be located. For example, if a cluster error occurs in array R as shown in Figure 3.35 with the values of bits 'w', 'x', 'y' and 'z' incorrectly set to logic ones, then array D can be represented as shown in Figure

3.36. The weight of both syndromes is 4 and the vertical syndrome contains a run of $b_1 b_2$ = 4 zeros. Although the rows and columns containing errors are marked by ones in the syndromes, the syndromes do not identify which 4 bits in array D are in error. Identifying the locations of the bits in error is the next task.

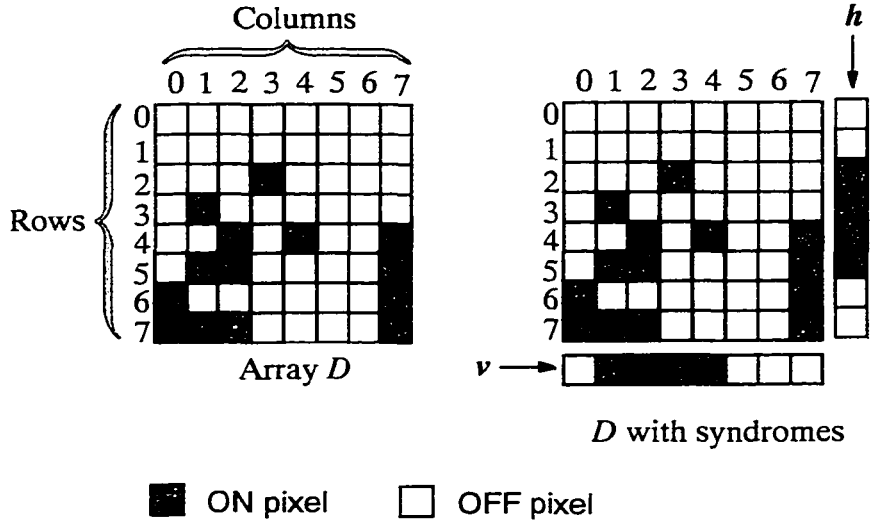


Figure 3.36: Array D is shown without and with syndromes ($\boldsymbol{h}$ and $\boldsymbol{v}$) for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$.

An algorithm for locating the cluster error proceeds as follows. Let vertical syndrome $\boldsymbol{v}$ have support (i.e., a set of nonzero coordinates) given by

$$Sup(\boldsymbol{v}) = \{j : v_j \neq 0, \qquad 0 \leq j \leq n_2 - 1\} \tag{3.9}$$

and the horizontal syndrome $\boldsymbol{h}$ have support given by

$$Sup(\boldsymbol{h}) = \{i : h_i \neq 0, \qquad 0 \leq i \leq n_1 - 1\}. \tag{3.10}$$

Both supports, $Sup(\boldsymbol{v})$ and $Sup(\boldsymbol{h})$, are assumed to have the same number of elements (i.e., the same total number of errors, $1 \leq e \leq b_1 b_2$); otherwise an uncorrectable error has occurred. Since $n_2 \geq 2b_1 b_2$ and $1 \leq e \leq b_1 b_2$, the elements of $Sup(\boldsymbol{v})$ are assumed to be ordered cyclically from the first to the last element. The destaggering process produces a burst of

length at most $b_1 b_2$ in both syndromes. Therefore, a sequence of at least $b_1 b_2$ zeros occurs before the first nonzero element in $\boldsymbol{v}$. For $\boldsymbol{v}$ shown in Figure 3.36, $Sup(\boldsymbol{v}) = \{1, 2, 3, 4\}$.

Considering $Sup(\boldsymbol{h})$, the elements can be divided into b_1 classes C_k, $0 \le k \le b_1 - 1$, given by

$$C_k = \{i \in Sup(\boldsymbol{h}) : \quad k = i \bmod b_1\}. \tag{3.11}$$

In each class C_k, the coordinates correspond to errors occurring in the same row (e.g., 'w' and 'x') before destaggering. The constraint $n_1 \ge 2b_1 b_2 - b_1$, insures there is a run with at least $b_1 b_2 - 1$ zeros before the first nonzero element in each cyclically ordered class C_k. In Figure 3.36, $Sup(\boldsymbol{h})$ can be divided into two classes, $C_0 = \{2, 4\}$ and $C_1 = \{3, 5\}$. Therefore $Sup(\boldsymbol{h})$ can be written as

$$Sup(\boldsymbol{h}) = C_0 \cup C_1 \cup \ldots C_k. \tag{3.12}$$

An algorithm for finding the set $\mathcal{S}$ of pairs of locations in error proceeds as follows.

1. Set $k \leftarrow b_1 - 1$, $\mathcal{S} \leftarrow \emptyset$ and c $\leftarrow 0$.
2. Loop: IF $C_k = \emptyset$ THEN GOTO Next.
3. ELSE, Let $C_k = \{i_1, i_2, \ldots, i_m\}$.
4. Set $H \leftarrow \{j_{c+1}, j_{c+2}, \ldots, j_{c+m}\}$.
5. Let r be such that $j_{c+r} \bmod b_2 < j_{c+l} \bmod b_2 \ \forall \ 1 \le l \le m, l \ne r$.
6. Set $\mathcal{S} \leftarrow \mathcal{S} \cup \{(i_{r-1}, j_{c+1}), (i_{r-2}, j_{c+2}), \ldots, (i_1, j_{c+r-1}), (i_m, j_{c+r}), (i_{m-1}, j_{c+r+1}), \ldots, (i_r, j_{c+m})\}$.
7. Next: IF k = 0 THEN GOTO Stop.
8. ELSE Set $k \leftarrow k - 1$, c $\leftarrow$ c + m and GOTO Loop.
9. Stop: Correct locations given by set $\mathcal{S}$ in array D.

The example in Figure 3.35 shows a cluster error of size $b_1 \times b_2$ with $b_1 = b_2 = 2$ and this example is continued in Figure 3.36 with $Sup(\boldsymbol{v}) = \{j_1, j_2, j_3, j_4\} = \{1, 2, 3, 4\}$, $C_0 = \{2,$

4} and $C_1 = \{3, 5\}$. For this example, step 1 of the algorithm assigns $k \leftarrow b_1 - 1 = 2 - 1 = 1$, $\mathcal{S} \leftarrow \emptyset$ and c $\leftarrow 0$. The loop spans steps 2 through 8. After executing step 2, step 3 assigns $C_1 = \{i_1, i_2\} = \{3, 5\}$, hence m = 2 and step 4 assigns $H \leftarrow \{j_{c+1}, j_{c+2}\} = \{j_1, j_2\} = \{1, 2\}$. In step 5, a value for r is found by observing (j_2 mod 2) = (2 mod 2) = 0 < 1 = (1 mod 2) = (j_1 mod 2); hence, $r = 2$. Step 6 assigns $\mathcal{S} \leftarrow \mathcal{S} \cup \{(i_{r-1}, j_1), (i_{r-2}, j_2)\} = \emptyset \cup \{(i_{2-1}, j_1), (i_{2-2}, j_2)\} = \{(i_1, j_1), (i_2, j_2)\} = \{(3, 1), (5, 2)\}$. Since the elements of C_k are cyclic, $C_1 = \{i_1, i_2\}$ with $i_{2-2} = i_2$. After step 7 is executed, step 8 assigns $k \leftarrow k - 1 = 1 - 1 = 0$, c $\leftarrow$ c + m = 0 + 2 =2 and the loop is executed again. After step 2 is executed, step 3 assigns $C_0 = \{i_1, i_2\} = \{2, 4\}$, hence $m = 2$ and step 4 assigns $H \leftarrow \{j_{c+1}, j_{c+2}\} = \{j_3, j_4\} = \{3, 4\}$. A value for r is determined in step 5 by observing (j_4 mod 2) = (4 mod 2) = 0 < 1 = (3 mod 2) = (j_1 mod 2); hence $r = 4$. Executing step 6 assigns $\mathcal{S} \leftarrow \mathcal{S} \cup \{(i_{r-1}, j_3), (i_{r-2}, j_4)\} = \{(3, 1), (5, 2)\} \cup \{(i_{4-1}, j_3), (i_{4-2}, j_4)\} = \{(3, 1), (5, 2)\} \cup \{(i_1, j_3), (i_2, j_4)\} = \{(3, 1), (5, 2), (2, 3), (4, 4)\}$. Since $k = 0$ in step 7, the algorithm goes to step 9 (labeled Stop) to correct the row and column pairs (r_i, c_j) in array D given by set $\mathcal{S}$ (i.e., the locations by row and column in error). The original data array A is obtained by inverting (correcting) the bits at the locations in error.

3.5.3 Encoder for a Cluster Error Correcting Array Code

During encoding, the parity bit checks are computed for the rows and columns; then the array is staggered. Therefore, an encoding algorithm can be expected to have four steps:

1. Latch data
2. Compute parity bit checks
3. Stagger row data
4. Stagger column data.

Consider a cluster error correcting array code with $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$. This code is capable of correcting a cluster error up to $b_1 \times b_2 = 2 \times 2$ bits in a $n_1 \times n_2 = 8 \times 8$ code

block. The discussion of the encoder and decoder (next section) is based on this array code as was a different implementation developed by another researcher [SCH98].

Let an 8×8 code block use an 8×8 array of pixels. Each pixel contains a storage element implemented with the hardware shown in Figure 3.37. Assume the flip–flop can be implemented using six gates. Therefore, a total of 12 gates are needed to implement the hardware. Approximately six gate delays are encountered from the photodetector signal to the data_out signal assuming the flip–flop introduces four gate delays. If the pixels located in even rows and columns were implemented with the hardware shown in Figure 3.6, the number of gates required can be reduced by 6.25%. For the encoder to latch the data, the delay encountered is $T_1 = 6$ gate delays. The hardware needed is $H_{cel} = 12n_1n_2 = 12(8)8 = 768$ gates.

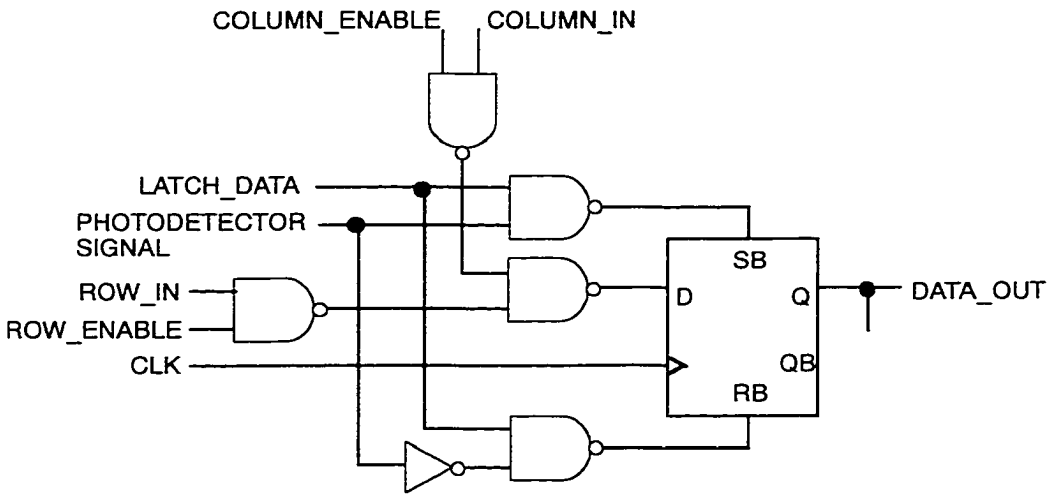


Figure 3.37: Data staggering hardware used for pixels in odd rows and columns.

Parity bit checks must be computed across each row and column. The row parity bit checks can be computed in $\lceil \log_2 k_2 \rceil$ gate delays using a tree of $k_2 - 1$ XOR–gates for each row and the column parity bit checks can be computed in $\lceil \log_2 k_1 \rceil$ gate delays using a tree of $k_1 - 1$ XOR–gates for each column. The time required to compute the row and column

parity bit checks is $T_{ces} = \max(\lceil \log_2 k_1 \rceil, \lceil \log_2 k_2 \rceil) = \max(\lceil \log_2 7 \rceil, \lceil \log_2 7 \rceil) = 3$ gate delays and the hardware needed is $H_{ces} = (k_1 - 1)n_2 + (k_2 - 1)k_1 = (7 - 1)8 + (7 - 1)7 = 97$ gates.

The data encoding operation requires data located in odd rows be staggered before the data located in odd columns is staggered to complete the data encoding process. Specifically, the odd rows of the code block must be circularly shifted right by $b_2(i \bmod b_1) = 2(1) = 2$ bits. Then the odd columns of the code block must be circularly shifted downward by $b_1(j \bmod b_2) = 2(1) = 2$ bits. If the positive edge of two clock pulses (i.e., one pulse for each bit shift) is used to clock a shift register made from the hardware shown in Figure 3.37, the time for the row staggering operation is $T_{sr} = b_2(i \bmod b_1) = 2(1) = 2$ clock cycles. In a similar way, the time for the column staggering operation is $T_{sc} = b_1(j \bmod b_2) = 2(1) = 2$ clock cycles.

The control signals required for steps 1, 3, and 4 of the algorithm can be provided by a state machine similar to the hardware shown in Figure 3.27. A 1–bit counter (i.e., a flip flop composed of six gates) can be used to provide the latch signal. Three 2–bit counters using ten gates per bit [NAT84] can be designed to provide state, row and column shift control signals. The hardware needed is $H_{csm} = 3(10\log_2 2^2) + 29 = 89$ gates.

For the cluster error correcting array code, the encoding hardware used is $H_{ce} = H_{csm} + H_{cel} + H_{src} = 97 + 768 + 89 = 954$ gates. The encoding delay is $T_{ce} = T_l + T_{ces} + T_{sr} + T_{sc} \approx T_l + T_{sr} + T_{sc} = 1 + 2 + 2 = 5$ clock cycles. Assuming a clock cycle is equivalent to 11 gate delays, the encoding delay is $T_{ce} = T_{ces} + (T_l + T_{sr} + T_{sc})11 = 3 + 5(11) = 58$ gate delays. The minimum clock cycle and equivalent gate delay value is determined in the next section.

3.5.4 Decoder for a Cluster Error Correcting Array Code

The decoding algorithm involves eight steps:

1. Latch data

2. Destagger column data

3. Destagger row data

4. Compute syndromes

5. Detect errors

6. Group errors

7. Locate errors

8. Correct errors.

The discussion of the decoding algorithm is also based on a cluster error correcting array code with a $n_1 \times n_2 = 8 \times 8$ code block size capable of correcting cluster errors of size $b_1 \times b_2 = 2 \times 2$ bits. Figure 3.38 shows the control signals for the algorithm. The positive edge of each control signal is used to initiate a corresponding step of the algorithm.

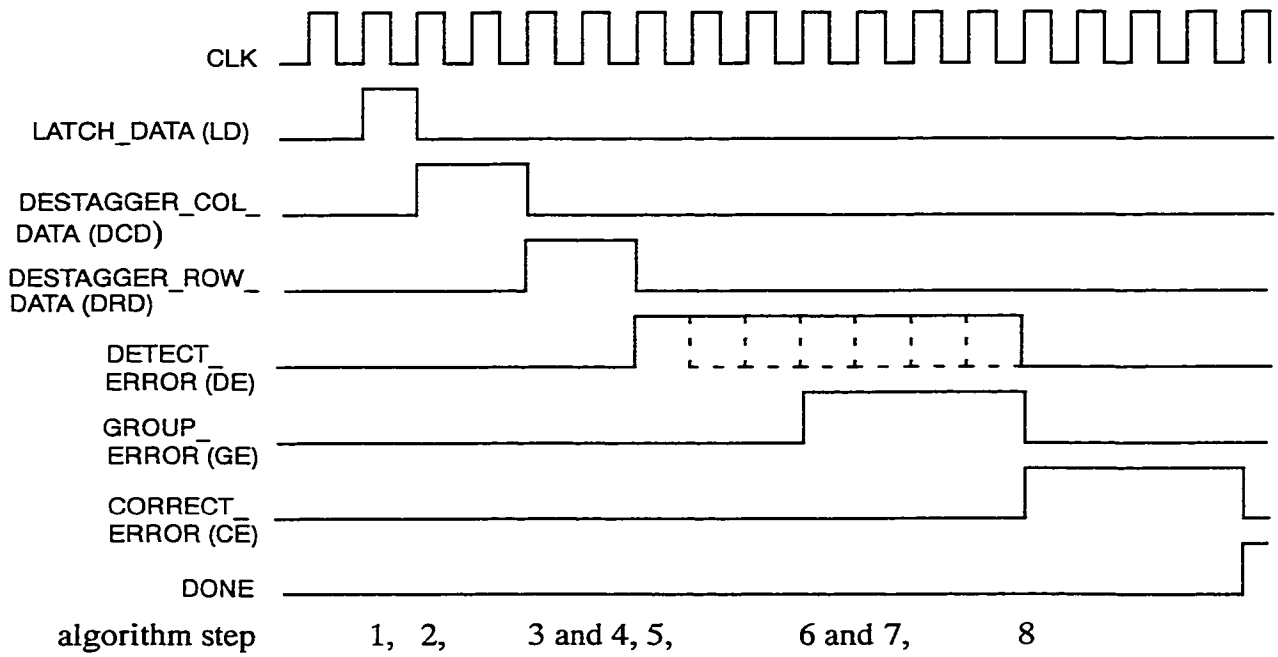


Figure 3.38: State machine timing diagram for cluster error correcting array codes.

A state machine with 6 states can be designed to provide the control signal for steps 1, 2, 3, 5, 6 and 8 of the decoding algorithm. Following an approach similar to that used to

produce the hardware shown in Figure 3.27, three 2–bit counters are used in steps 1, 2 and 3, a four bit counter is used in step 5 and two 3–bit counters are used in steps 6 and 8. The hardware required is $H_{dsm} = 10\log_2(2^4) + 3[10\log_2(2^3)] + 3[10\log_2(2^2)] + 34 = 224$ gates.

The decoding operation requires an optical page of information be latched into a detector array. For the code block size being considered, the detector array must have a corresponding 8×8 array of pixels. The storage element in each pixel can be implemented using the hardware shown in Figure 3.37. From the discussion on encoding, 12 gates can be used to implement the hardware in Figure 3.37 and approximately six gate delays are associated with signals using this hardware. The time delay involved is $T_l = 6$ gate delays and the hardware needed to latch the data is $H_{dl} = 12n_1n_2 = 12(8)8 = 768$ gates.

In the data destaggering steps, data located in columns are destaggered and then data located in rows are destaggered. After data in odd columns are circularly shifted upwardly an amount equal to $b_1(j \bmod b_2) = 2(1) = 2$ bits, data in odd rows are circularly shifted to the left an amount equal to $b_2(i \bmod b_1) = 2(1) = 2$ bits. If one clock cycle is used for each bit shift, the time required to destagger a column is $T_{sc} = b_1(j \bmod b_2) = 2(1) = 2$ clock cycles and to destagger a row is $T_{sr} = b_2(i \bmod b_1) + 2(1) = 2$ clock cycles.

The row and column syndromes are computed respectively using trees of $n_2 - 1$ and $n_1 - 1$ XOR–gates. The time to compute these syndromes is $T_{ds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays and represents the minimum period of the system clock or the reciprocal of the maximum system clock frequency. The required hardware is $H_{ds} = k_2(n_1 + 1) + k_1n_2 = 7(8 + 1) + 7(8) = 119$ gates.

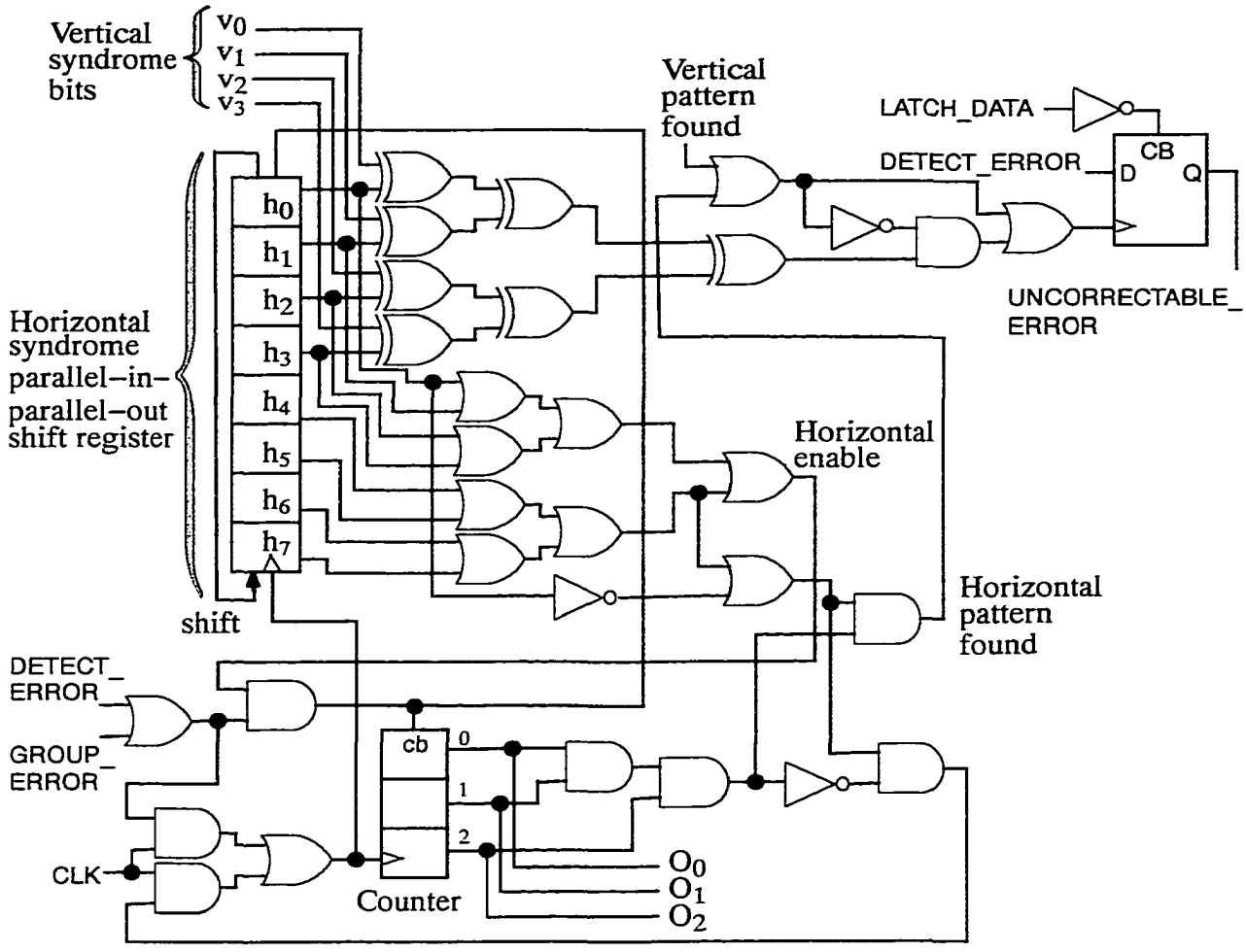


Figure 3.39: Hardware applied to horizontal part of the detect error circuit.

The detect errors step of the algorithm involves several tasks. The row and column syndromes are loaded into two different shift registers. In parallel fashion, the row and column syndromes are tested for bit patterns with all zeros (i.e., no bit failures implies no errors have been detected), for bit patterns with more than $b_1 b_2 = 2(2) = 4$ bit failures and for bit patterns having a different number of bit failures between the row and column syndromes. The latter two bit patterns indicate an uncorrectable error has been detected. Part of the error detection hardware is shown in Figure 3.39. The shift register in Figure 3.39 can be implemented using the hardware in Figure 3.29. Each bit stage making up the counter

used in the detection hardware can be implemented using ten gates [NAT84] and the associated time delay from the clock input to an output is five gate delays. Inspecting Figure 3.39 indicates the hardware needed for this step is $H_{dde} = [9(8) + 26 + 10(3)]2 = 267$ gates and the maximum time delay is $T_{dde} = 2b_1b_2 - 1 = 2(4) - 1 = 7$ clock cycles. The time delay from the output of the shift register to the output of the counter, $T_{sco} = (5 + \lceil \log_2 b_1 b_2 \rceil + 4) = 5 + \lceil \log_2 4 \rceil + 4 = 11$ gate delays, determines the minimum clock period of the system clock or the reciprocal of the maximum clock frequency.

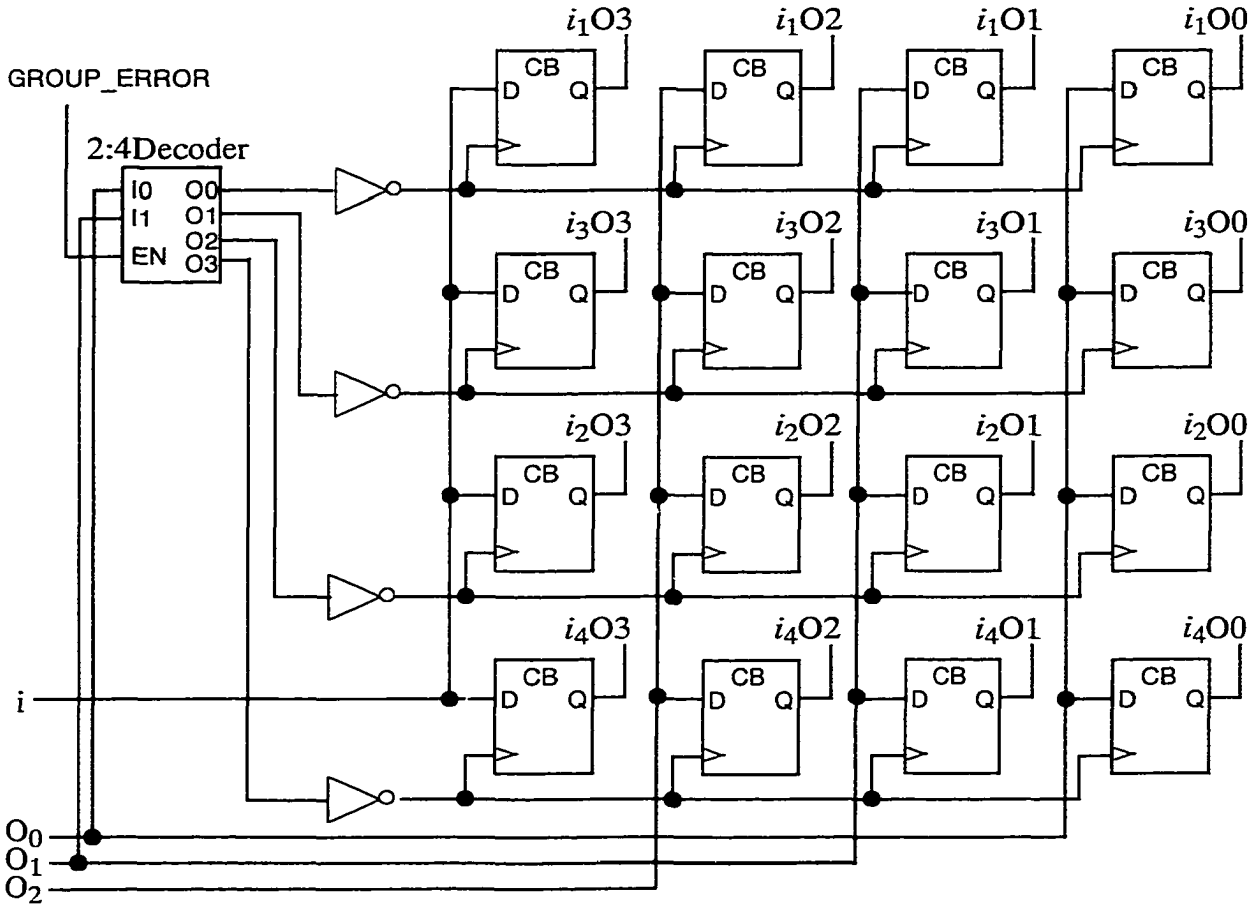


Figure 3.40: Hardware used for horizontal part of group error circuit.

The group error step of the algorithm separates the horizontal syndrome, ***h***, into classes but not the vertical syndrome. From the discussion in section 3.5.1, ***h*** must be

grouped into two classes. The hardware shown in Figure 3.40 accomplishes this task. Class C_0 is obtained when $O_1O_0 = 00$ or 10 (i.e., when the least significant bit is zero) and class C_1 is obtained when $O_1O_0 = 01$ or 11 (i.e., when the least significant bit is one). The four flip–flops store a row address where the value of i (e.g., i_1O3) is the most significant bit and the value of $O_2O_1O_0$ occupies the three remaining bits. Although the vertical syndrome, ν, is not grouped into classes, similar hardware is used to develop a column address with j (e.g., j_1O3) being the most significant bit in the address.

Assume the decoder [NAT84] shown in Figure 3.40 can be implemented using nine gates and six gates are used to implement each flip–flop. The group errors hardware requires $H_{dge} = [9 + 4 + 6(4)4]2 = 218$ gates. The time delay to cycle through the row and column addresses is $T_{dge} = b_1b_2 = 2(2) = 4$ clock cycles.

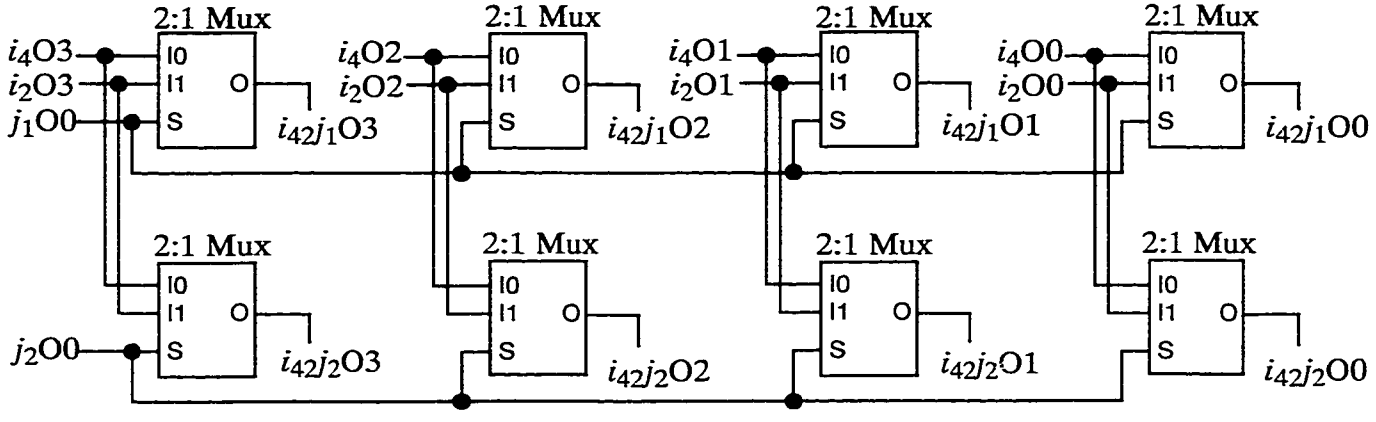


Figure 3.41: Part of the hardware required for the locate error circuit.

Part of the hardware used to perform the locate errors step of the algorithm is shown in Figure 3.41. The least significant bit of the column address (e.g., j_1O_0) is used to associate specific i and j values to locate each error making up the cluster. This hardware must be replicated one more time with i_4 replaced by i_3 , i_2 replaced by i_1, j_1 replaced by j_3 and j_2 replaced by j_3. Four of the 2:1 multiplexers [NAT84] shown in Figure 3.41 can be

implemented using 14 gates with a time delay of four gate delays. The hardware needed is $H_{dle} = 14b_1b_2 = 14(4) = 56$ gates and the time delays is $T_{dle} = 4$ gate delays.

Figure 3.42 shows part of the hardware applied to the correct errors step of the algorithm. Four 4:1 multiplexers are used to steer row address bits to a row address 3:8 line decoder and similar hardware steers column address bits to a column address decoder. The row and column addresses are cycled through using a counter. The 4:1 multiplexers [NAT84] can be implemented with 11 gates, the 8:1 multiplexers [NAT84] can be implemented with 16 gates and the 3:8 line decoder [NAT84] can be implemented with 15 gates. Using ten gates per bit, the counter requires 20 gates. The number of gates used in the correct errors hardware is $H_{dce} = 11(4)2 + 16(8) + 15(2) + 10(2) + 8(4) = 298$ gates the time delay experienced in cycling through the row and column addresses is $T_{dce} = b_1b_2 = 2(2) = 4$ clock cycles.

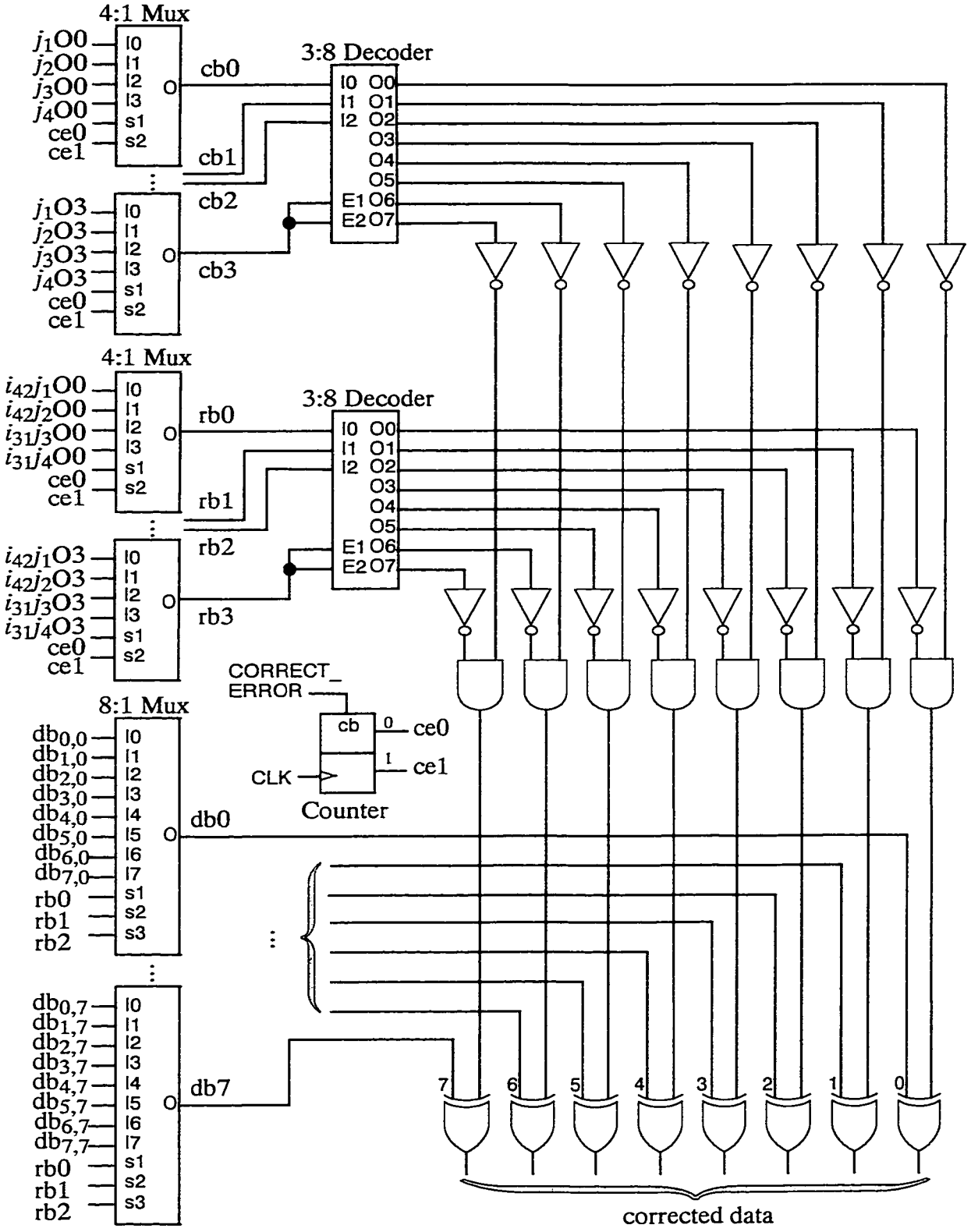


Figure 3.42: Hardware applied to error correction circuit.

The overall hardware and time delay is obtained by considering all the steps making up the decoding algorithm. The decoding hardware required for the cluster error correcting array code is $H_{cd} = H_{dsm} + H_{dl} + H_{ds} + H_{dde} + H_{dge} + H_{dle} + H_{dce} = 224 + 768 + 119 + 267 + 218 + 56 + 298 = 1950$ gates. The decoding delay is $T_{cd} = T_l + T_{sc} + T_{sr} + T_{ds} + T_{dde} + T_{dge} + T_{dle} + T_{cce} \approx T_l + T_{sc} + T_{sr} + T_{dde} + T_{dge} + T_{dce} = 1 + 2 + 2 + 7 + 4 + 4 = 20$ clock cycles. For a clock cycle equivalent to 11 gate delays, the equivalent decoding delay is $T_{cd} = T_{ds} + (T_l + T_{sc} + T_{sr} + T_{dde} + T_{dge} + T_{dle} + T_{cce})11 = 11 + 20(11) = 231$ gate delays.

3.5.5 Encoding and Decoding of a Multiblock CEC Array Code

Using the multiblock structure shown in Figure 3.3, the hardware and time delays for a multiblock cluster error correcting array code are considered. The analysis used for the row and column code is followed. In addition to the encoding and decoding hardware, hardware is needed to indicate an error condition status of correct and uncorrectable.

The variables T_{ce}, T_{cd}, H_{ce} and H_{cd} were described in sections 3.5.3 and 3.5.4. These variables are used in the expressions given below describing the time delay and hardware needed for the block serial implementation of the multiblock CEC array code where T_{mces} is the encoding delay, T_{mcds} is the decoding delay, H_{mces} is the encoding hardware needed and H_{mcds} is the decoding hardware needed.

Encoding delay: $T_{mces} = m_1 m_2 T_{ce}$

Decoding delay: $T_{mcds} = m_1 m_2 T_{cd}$

Encoding hardware: $H_{mces} = H_{ce}$

Decoding hardware: $H_{mces} = H_{cd}$

The block parallel implementation of the multiblock cluster error correcting array code encodes and decodes all component codes in parallel as described in section 3.2.5. For the global–AND–gate, let T_{cgA} be the timing delay and H_{cgA} be the hardware needed. The

following expressions describe T_{mcep}, the encoding delay, T_{mbcp}, the encoding delay, H_{mcep}, the encoding hardware needed, and H_{mcdp}, decoding hardware needed.

Encoding delay: $T_{mcep} = T_{ce}$

Decoding delay: $T_{mcdp} = T_{cd} + T_{cgA}$

Encoding hardware: $H_{mcep} = m_1 m_2 H_{ce}$

Decoding hardware: $H_{mcdp} = m_1 m_2 H_{cd} + H_{cgA}$

We know a multiblock cluster array code can be used on larger page sizes by selecting the size of the component block code appropriately. A component code with $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$, can be used on larger pages (e.g., 256×256, 512×512 and 1024×1024) that are integer multiples of $n_1 = n_2 = 8$. In this way, the time delay, time to process the data and hardware required for the smaller block sizes can be applied to larger page sizes.

3.5.6 Code Performance of a Cluster Error Correcting Array Code

Code simulations can be used to study the *CBER* performance of cluster error correcting array codes. During storage and retrieval, cluster errors may be introduced into a page of data. For a cluster error correcting array code with $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$, important parameters are $(n, k, d) = (64, 49, 4)$ and $r = 0.7656$. A cluster error correcting array code with $n_1 = n_2 = 16$, $b_1 = 4$ and $b_2 = 2$ has parameters $(n, k, d) = (256, 225, 4)$ and $r = 0.8789$. Each code supports detection and correction of a single cluster error. Using a sum of probabilities written in terms of *RBER*, code size n and error e, the probability, $P_{correct}$, that no error passes through the decoder can be expressed as:

$$P_{correct} = \sum_{e=0}^{b_1 b_2} \left(\frac{n(b_1 b_2 - 1)!}{e!(b_1 b_2 - e)!} \right) RBER^e (1 - RBER)^{n-e}, \qquad e > 0. \quad (3.13)$$

Using Equations 3.4 and 3.13, *CBER* can be expressed in terms of n, e and *RBER*.

Figure 3.45 shows the code performance of 4×2 and 2×2 cluster error correcting array codes. Even for relatively low values of *RBER*, these codes can not achieve a

CBER $\leq 10^{-12}$. Regarding VHDS applications with *RBER* $\geq 10^{-6}$, it is not likely these codes can be used as either a multiblock or uniblock code structure and achieve a *CBER* $\leq 10^{-12}$. These codes are adapted versions of the row and column array codes and cannot achieve a higher error correcting capability than the corresponding row and column array codes. It is important to mention that some of the literature on VHDS systems provides information regarding bit error rates [HEA94, BER96, GOE96, IBM96, BUR97, KIN99, TAO99]. However, the cluster error rates for these systems have not been widely reported.

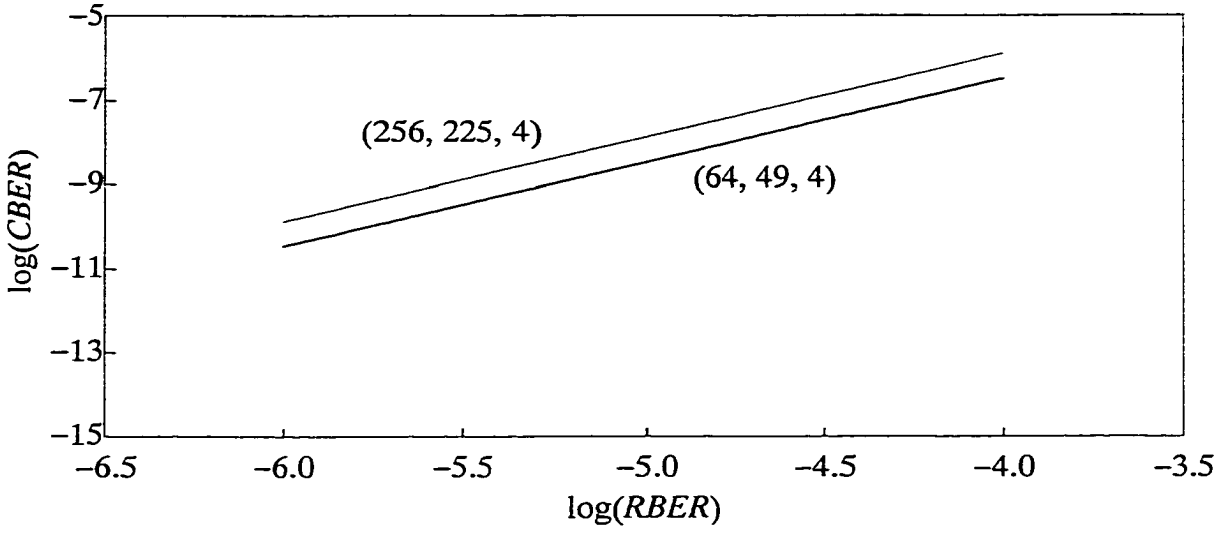


Figure 3.43: *CBER* versus *RBER* for cluster error correcting array codes.

Chapter 4
Block 3–D Array Codes

This chapter presents an introduction to block 3–D array codes. The characteristics of random error correcting array codes are considered. Next, these codes are adapted for use as BEC array codes. Finally, these codes are adapted for use as a CEC array code.

4.1 Advantages and Disadvantages of 3–D Array Codes

There are several advantages for 3–D array codes. These codes have a higher error correcting capability than 1–D and 2–D codes. For example, a single parity check code has a Hamming distance of 2, a simple row and column code constructed from two single parity check codes has a hamming distance of 4, while a 3–D code constructed from three single parity check codes has a hamming distance of 8. The hamming distance parameter provides the single parity check code the ability to detect one error, the row and column code the ability to detect two errors and correct one error and the 3–D code the ability to detect four errors and correct three errors. The higher error correcting capability of 3–D codes permits a lower *CBER* to be attained for a given *RBER* than is possible with single parity bit and row

and column codes. For example, the equivalent of the VHDS system shown in Figure 2.3 is replicated multiple times to produce a multiple channel memory with each channel assigned to a data page. During a single access of memory where multiple pages of memory are stored or retrieved, an uncorrectable error condition may occur on some pages while other pages have no detected errors or a correctable error condition. A 3–D code can be designed to use its third dimension to provide additional error protection such that only the pages containing an uncorrectable error condition are rejected and correct or corrected pages are sent to the next stage of the system. A 3–D code can be used to correct a burst error occurring in each layer making up the 3–D information bit array of the code. Similarly, a 3–D code can correct a cluster error in each layer making up a 3–D array code. An analysis of the code parameters (e.g., code length, code rate, Hamming distance, etc.) for these 3–D array codes is given later.

There are two disadvantages for 3–D array codes. They have a higher hardware complexity for implementing the encoder and decoder as compared to the single parity check and row and column codes. An analysis of hardware complexity is given later. These codes also use more parity bit checks than the single parity check and row and column codes.

4.2 Analysis of 3–D Row and Column Array Codes

For a given 3–D code size, the number of parity check bits will be smaller as the code block approaches a cube. Let the information part of a code have size, $k = (m-1)(p-1)^2$, then the overall size of the code is $n = mp^2$, where $m = n_1$ and $p = n_2 = n_3$. Then k in terms of m and n is given by

$$k = (m - 1)(\sqrt{\frac{n}{m}} - 1)^2. \tag{4.1}$$

In Equation 4.1, taking the derivative of k with respect to m and setting it to zero leads to

$$0 = (\sqrt{\frac{n}{m}} - 1)^2 - (m - 1)(\sqrt{\frac{n}{m}} - 1)\sqrt{\frac{n}{m^3}}. \quad (4.2)$$

Solving Equation 4.2 for m leads to $n = m^3$ which is the same as $n = n_1 n_1 n_1$ (i.e., a cube that has sides of n_1 by n_1 by n_1.

Assume error detection and correction is performed using the 3–D row and column code (i.e., a 3–D uniblock row and column array code) shown in Figure 4.1. The parameters for this code are given by the following expressions [GAR98].

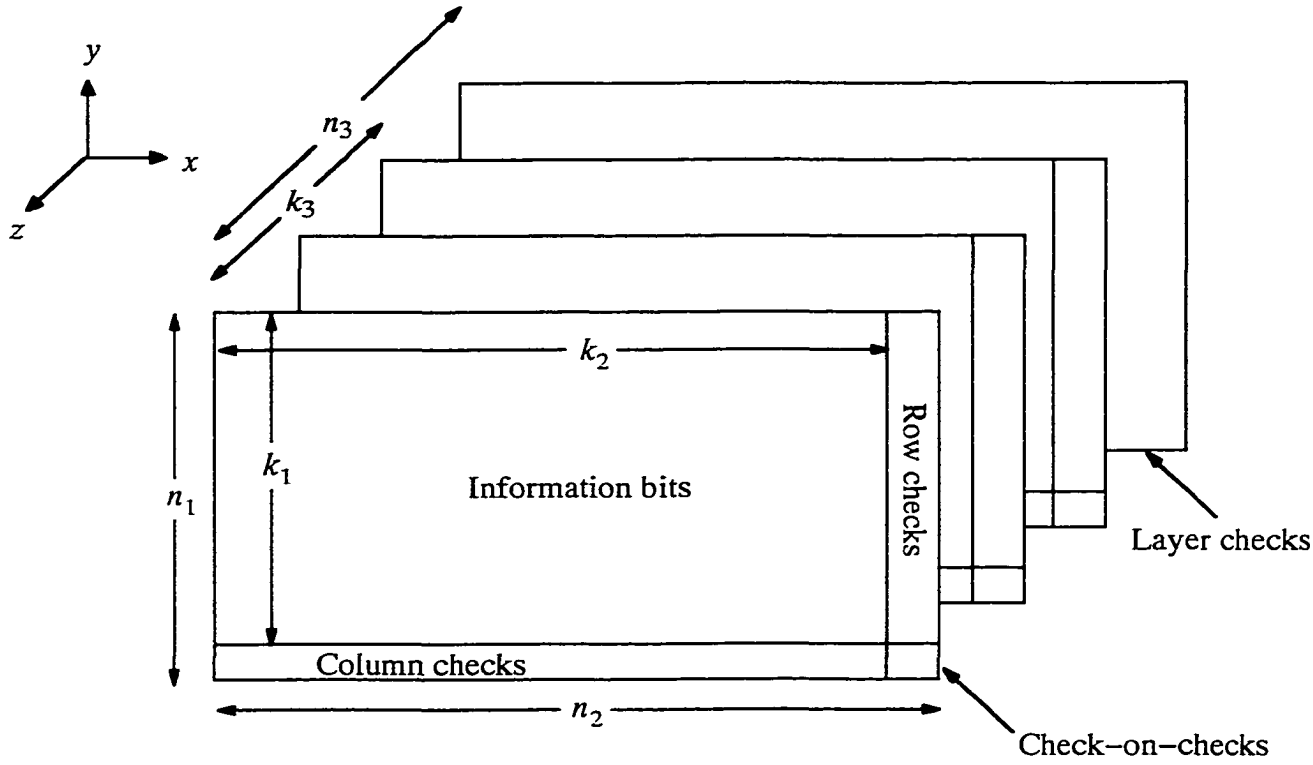


Figure 4.1: Uniblock 3–D row and column array code.

Length of code: $n = n_1 n_2 n_3 = (k_1 + 1)(k_2 + 1)(k_3 + 1)$

Information bits: $k = k_1 k_2 k_3$

Hamming distance: $d = 8$

Code rate: $r = \frac{k}{n} = \frac{k_1 k_2 k_3}{(k_1 + 1)(k_2 + 1)(k_3 + 1)}$

Coding overhead: $n - k = k_1k_2 + k_1k_3 + k_2k_3 + k_1 + k_2 + k_3 + 1$

Number of errors that can be detected: $t = 4$

Number of errors that can be corrected: $c = 3$

Type of errors that can be corrected: random

The parity check bits are generated along the rows, columns and layers of a $n_1 \times n_2 \times n_3$ bit data array. When data are read, the intersection of ones in the parity check vectors give the position of as many as three errors in the $(k_1+1) \times (k_2+1) \times (k_3+1)$ array. Four errors within the 3–D code block are detected by the presence of ones in the parity check vectors and a zero in the check–on–check–on–checks. If more than fives errors occur, this code may not detect all cases, may miscorrect and may misdetect errors. For example, a cubic pattern of five errors may be miscorrected producing a sixth error; whereas, a cubic pattern of eight errors may fail to produce a difference in the parity check bits.

Several 3–D row and column codes are shown in Table 4.1. The 3–D array codes with two information layers and a parity bit layer have a higher error detecting and correcting capability than do similar codes without a parity bit layer. However, when three or more layers are involved, the code not using a parity check layer is able to detect and correct more errors than the same code containing a parity bit check layer. We use this idea later.

Table 4.1: Comparison of 3–D uniblock array codes with and without parity bit layers.

Information layers	Row Parity Bits	Column Parity Bits	Parity Bit Layer	Hamming distance	Errors Detected	Errors Corrected
2	yes	yes	yes	8	4	3
3	yes	yes	yes	8	4	3
2	yes	yes	no	4	2/layer $\Rightarrow$ 4	1/layer $\Rightarrow$ 2
3	yes	yes	no	4	2/layer $\Rightarrow$ 6	1/layer $\Rightarrow$ 3

Issues associated with decoding hardware complexity and reduced error correcting capability cause the usefulness of the uniblock row and column code to be exceeded as the page size increases. Therefore, a multiblock strategy can be applied to applications where data page sizes may be as large as 10^6 bits. For example, for $n_1 = 16$, $n_2 = 8$, $n_3 = 3$ and a 3–D page size of $512 \times 512 \times 3$ as many as 4352 errors can be corrected if each occurs in a different 3–D block.

Assume error detection and correction is performed using the multiblock 3–D row and column array code shown in Figure 4.2. Let the 3–D information bit array represent a $m_1 \times m_2$ array of $n_1 \times n_2 \times n_3$ 3–D code blocks. The following expressions describe the parameters for this multiblock code.

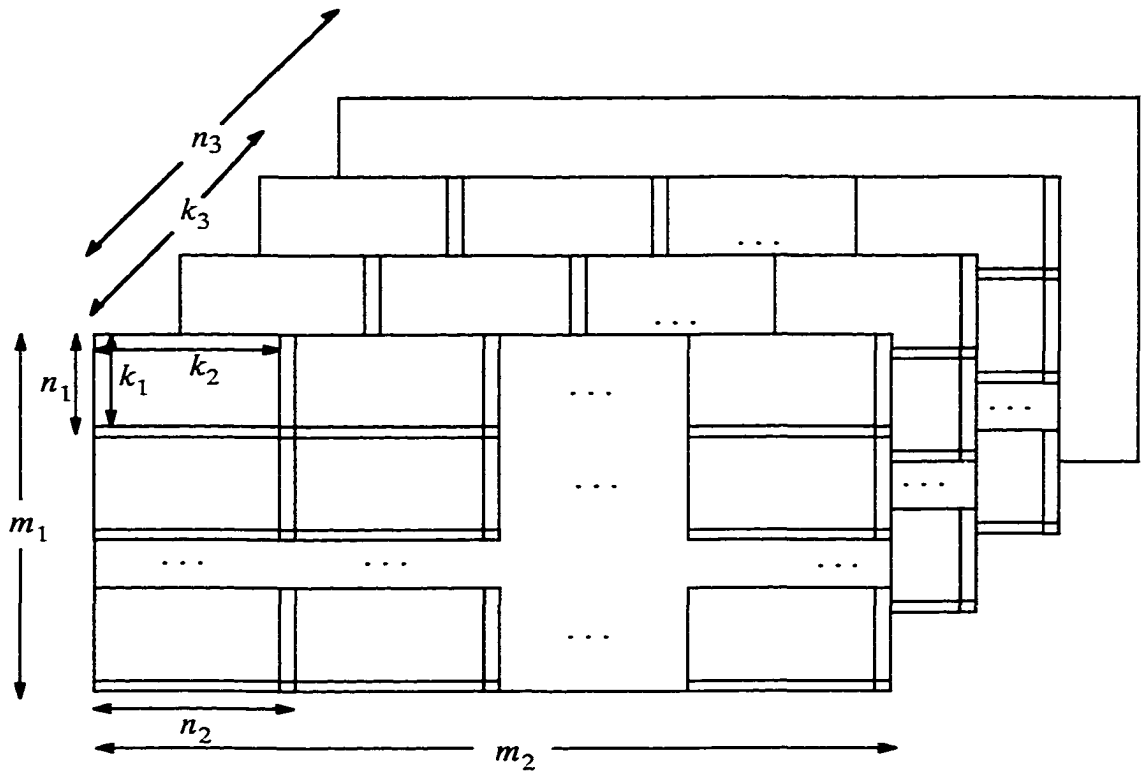


Figure 4.2: Multiblock 3–D row and column array code.

Length of code: $n = m_1 m_2 n_1 n_2 n_3 = m_1 m_2 (k_1 + 1)(k_2 + 1)(k_3 + 1)$

Information bits: $k = m_1 m_2 k_1 k_2 k_3$

Hamming distance: $d = 8$

Code rate: $r = \frac{k}{n} = \frac{k_1 k_2 k_3}{(k_1 + 1)(k_2 + 1)(k_3 + 1)}$

Coding overhead: $n - k = m_1 m_2 (k_1 k_2 + k_1 k_3 + k_2 k_3 + k_1 + k_2 + k_3 + 1)$

Number of errors that can be detected: $t = \{t_{\min} = 4, t_{\max} = 4m_1 m_2\}$

Number of errors that can be corrected: $c = \{c_{\min} = 3, c_{\max} = 3m_1 m_2\}$

Type of errors that can be corrected: random

When three or fewer errors occur in any block, this code can correct up to $3m_1 m_2$ errors. When four or fewer errors occur, this code can detect up to $4m_1 m_2$ errors. Otherwise no error detection or correction using the multiblock approach is possible.

4.2.1 Encoding for 3–D Row and Column Array Codes

For example, consider the array code shown in Figure 4.3. Using an even parity strategy, the row, column and layer parity checks in Figure 4.3(b) are generated from the information bits in Figure 4.3(a). The data array and parity checks form the code block stored in the optical memory.

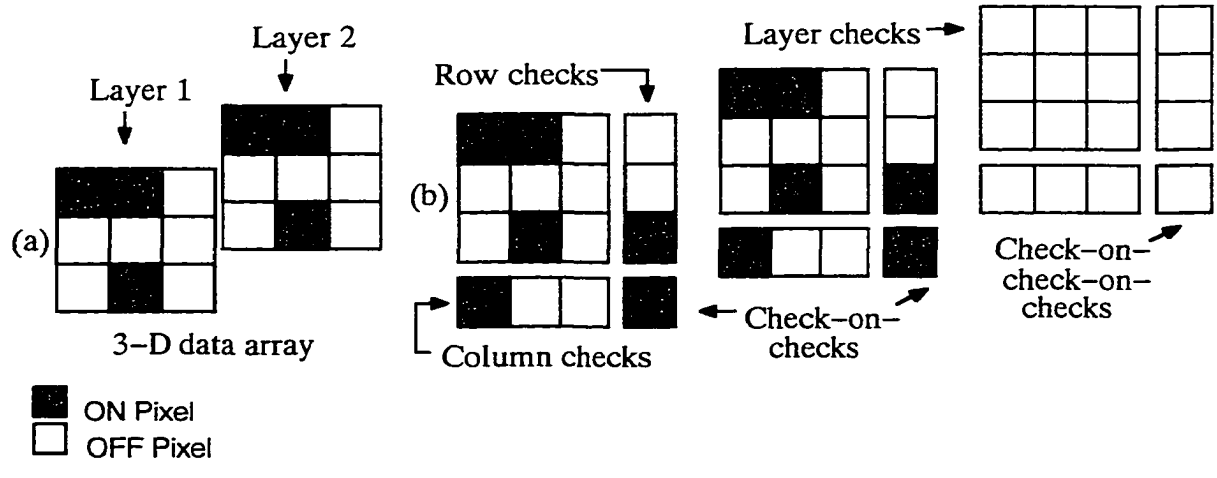


Figure 4.3: Data encoding applied to a 3×3×2 array code.

4.2.2 Decoding for 3–D Row and Column Array Codes

The occurrence of an error in a retrieved data block is shown in Figure 4.4(a). To decode the code block in Figure 4.4(a), the row, column and layer parity checks are computed as shown in Figure 4.4(b). The parity checks shown in Figure 4.4(b) point to the location of the bit in error. A code block with no errors will have all of the layer parity checks off.

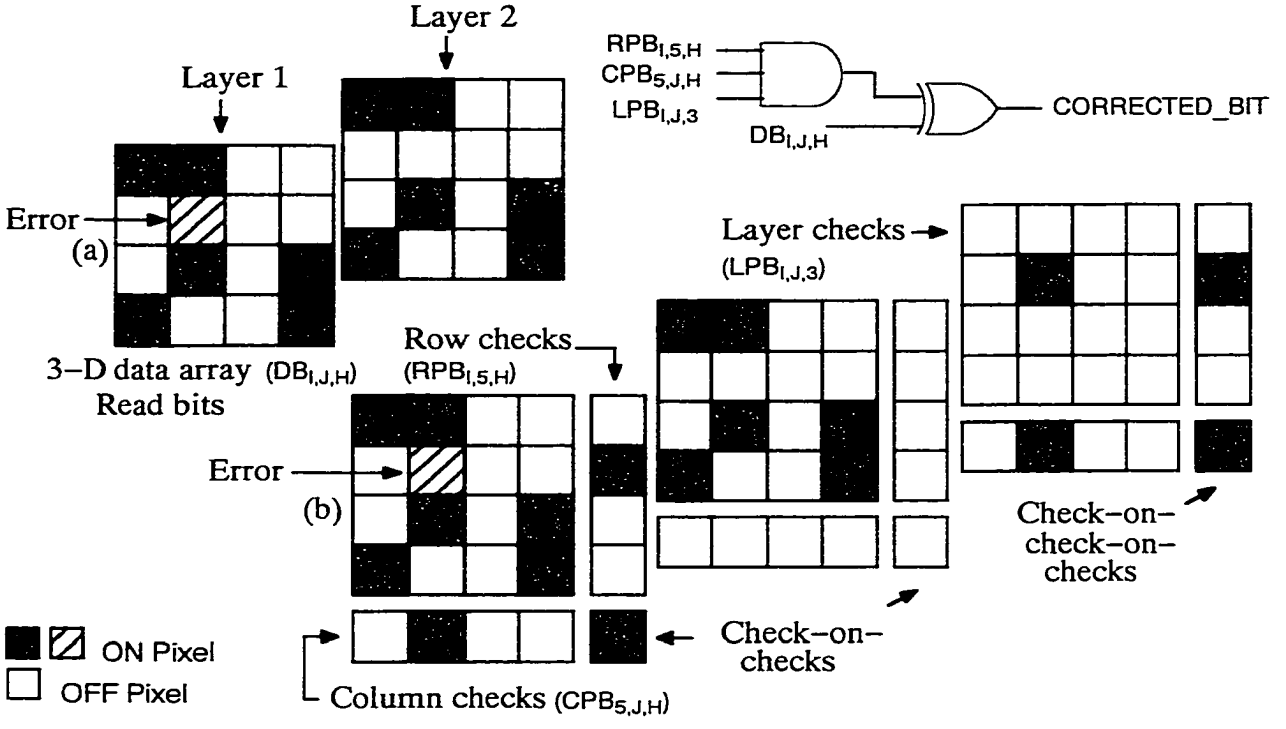


Figure 4.4: Data decoding applied to a $4\times4\times2$ array code.

4.2.3 Encoder for 3–D Row and Column Array Codes

Referring to Figure 4.1, the encoder computes the row, column and layer parity bit checks for multiple pages of data $n_1\times n_2\times n_3$ bits in size. Except for the need to consider the n_3–direction of the 3–D row and column code, the results of the row and column codes described in section 3.2.3 are adapted for the 3–D row and column code. Therefore, the hardware needed is $H_{3re} = (k_2-1)n_1k_3 + (k_1-1)k_2k_3 + (k_3-1)n_1n_2$ gates and the encoding time delay is $T_{3re} = \max(\lceil\log_2 k_1\rceil, \lceil\log_2 k_2\rceil, \lceil\log_2 k_3\rceil)$ gate delays.

4.2.4 Decoder for 3–D Row and Column Array Codes

The decoder performs the same four steps described in section 3.2.4 for the row and column code. While accounting for the n_3–direction of the 3–D row and column code shown in Figure 4.1, the expressions developed for the row and column codes in section 3.2.4 can be modified and applied to the 3–D row and column code. Using the hardware shown in Figure 3.6, a 3–D row and column code can be shown to have a hardware requirement of $H_{3rdl} = 9n_1n_2n_3$ gates with a decoding time delay $T_l = 6$ gate delays. This timing delay is the same as the row and column code described in section 3.2.4.

The syndromes must be computed across the rows, columns and layers. The expressions from section 3.2.4. for the row and column code, show the hardware needed is $H_{rds} = k_2(n_1 + 1) + k_1n_2$ gates and the corresponding delay is $T_{rds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil)$ gate delays. Therefore, the hardware needed for the 3–D row and column code is $H_{3rds} = k_2(n_1 + 1)n_3 + k_1n_2n_3 + k_3n_1n_2$ gates and the delay is $T_{3rds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil)$ gate delays.

The error detection and correction steps of the algorithm are considered next. A data bit in error is detected and corrected using the hardware shown in Figure 4.4. Additional hardware is needed to detect the uncorrectable error which occurs when two or more bit failures occur only in the row or column syndromes of the information layers (i.e., first k_3 of the n_3 layers). For each of the k_3 layers, the uncorrectable error condition is detected using the row and column syndromes as input to trees of OR–gates as was done for the row and column code described in section 3.2.4. Therefore, the hardware needed is $H_{3rdc} = 2n_1n_2k_3 + (n_1 - 1)k_3 + (n_2 - 1)k_3 + k_3 - 1$ gates. If an uncorrectable error has not occurred, the time delay is $T_{3rdc} = 2$ gate delays otherwise $T_{3rdc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 2$ gate delays.

The overall hardware required for decoding is $H_{3rd} = H_{3rdl} + H_{3rdd} + H_{3rdc} = (9n_1 + k_1)n_2n_3 + (n_1 + 1)k_2n_3 + [3n_1n_2 + (n_1 - 1) + (n_2 - 1) + 1]k_3 - 1$ gates. If an uncorrectable error has not occurred, the overall time delay is $T_{3rd} = T_1 + T_{3rds} + T_{3rdc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 8$ gate delays otherwise $T_{3rd} = 2\max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 8$ gate delays.

An analysis was performed to gain insight into the hardware complexity of the 3–D uniblock code shown in Figure 4.1. In cases of four or more errors, this code can not provide a corrective action. The no–corrective–action–taken condition must be indicated so the system can reread the pages again or take other corrective action. However, we can proceed with a corrective action when cases of one, two or three errors occur. Several conditions must be checked to determine the error status of the system. If no errors are indicated in the n_1, n_2 and n_3 direction check bits or only the check–on–check–on–checks bit is in error, then we assume we correctly read a page of data. The pages of information bits are passed on to the next stage of the system. If the n_1, n_2 and n_3 direction check bits and the check–on–check–on–checks bit indicate an error, then we assume we have detected a single error in the information bits. The n_1, n_2 and n_3 direction where the errors in the check bits occur indicates the position of the error in the information array and we can correct the error by inverting the bit stored at this position. If one n_1 or n_2 or n_3 direction check bit indicates an error, then we assume the check bit is in error. Again the pages of information bits are passed to the next stage of the system. The strategy described will locate up to three errors occurring in a 3–D information array.

4.2.5 Encoding and Decoding Multiblock 3–D RAC Array Codes

We consider the timing delay and hardware required for the multiblock 3–D row and column code shown in Figure 4.2 in this section. The occurrence of three or fewer errors

in any 3–D component code block making up the multiblock code means either multiple pages of data have been correctly read or component codes containing errors can be corrected. Therefore, each component code needs hardware for encoding and decoding hardware plus decoding hardware to generate a correct and uncorrectable error condition status. The parameters T_{3re}, T_{3rd}, H_{3re} and H_{3rd} were derived in sections 4.2.3 and 4.2.4. These parameters are used in expressions given below describing the time delay and hardware required to implement the multiblock 3–D row and column code.

Component 3–D code blocks are encoded and decoded sequentially in the block serial implementation. Using T_{3mres}, as the encoding delay, T_{3mrds}, as the decoding delay, H_{3mres}, as the encoding hardware required and H_{3mrds} as the decoding hardware required, the following expression for time delay and hardware requirements are obtained.

Encoding delay: $T_{3mres} = m_1 m_2 T_{3re}$

Decoding delay: $T_{3mrds} = m_1 m_2 T_{3rd}$

Encoding hardware: $H_{3mres} = H_{3rc}$

Decoding hardware: $H_{3mrds} = H_{3rd}$

A block parallel implementation encodes and decodes component code blocks making up the multiblock 3–D row and column code in parallel. For this implementation, assume a single pass is used to process all 3–D component block codes in parallel. Let T_{3rgA} be the delay encountered and H_{3rgA} be the hardware required for the global–AND–gate. In the following expressions, T_{3mrep}, is the encoding delay, T_{3mrdp}, is the decoding delay, H_{3mrep}, is the encoding hardware required and H_{3mrdp} is the decoding hardware required.

Encoding delay: $T_{3mrep} = T_{3re}$

Decoding delay: $T_{3mrdp} = T_{3rd} + T_{3rgAg}$

Encoding hardware complexity: $H_{3mrep} = m_1 m_2 H_{3re}$

Decoding hardware complexity: $H_{3mrdp} = m_1 m_2 H_{3rd} + H_{3rgA}$

4.2.6 Code Performance of 3–D Row and Column Array Codes

Several 3–D row and column codes were simulated to study the *CBER* performance of these codes. The simulations were performed assuming a Gaussian noise channel produces random errors during the intensity detection process. For a 3–D array code, let $n_1=n_2=n_3=5$ then $k_1=k_2-1=k_3=4$. The code has triplet $(n,k,d) = (125,64,8)$, $r = 0.512$ and supports detection of 4 errors and correction of 3 errors. We know from section 3.2.6, the probability, $P_{correct}$, that an error, e, does not pass through the decoder in terms of e, *RBER* and array code size n can be expressed as

$$P_{correct} = \sum_{e=0}^{4} \binom{n}{e} RBER^{e}(1 - RBER)^{n-e}. \tag{4.3}$$

In terms of the *CBER*, the probability, $P_{correct}$, that all bits are correctly decoded is given by Equation 3.4 and is repeated here for convenience

$$P_{correct} = (1 - CBER)^{n}. \tag{4.4}$$

For a given *RBER* (e.g., 10^{-5}) and a desired *CBER* (e.g., 10^{-12}), the corresponding value of n can be obtained by equating Equations 4.3 and 4.4 and solving for n. Figure 4.5 shows the values of n which yield $CBER \leq 10^{-12}$ and $CBER \leq 10^{-14}$. Codes with smaller block sizes (i.e., codes with smaller code length, n) correspond to the lower *CBER* for a given *RBER*. As will be discussed shortly, codes with smaller block sizes are associated with codes of lower rate. Therefore, a tradeoff is involved between the error correcting capability of the code and the code rate.

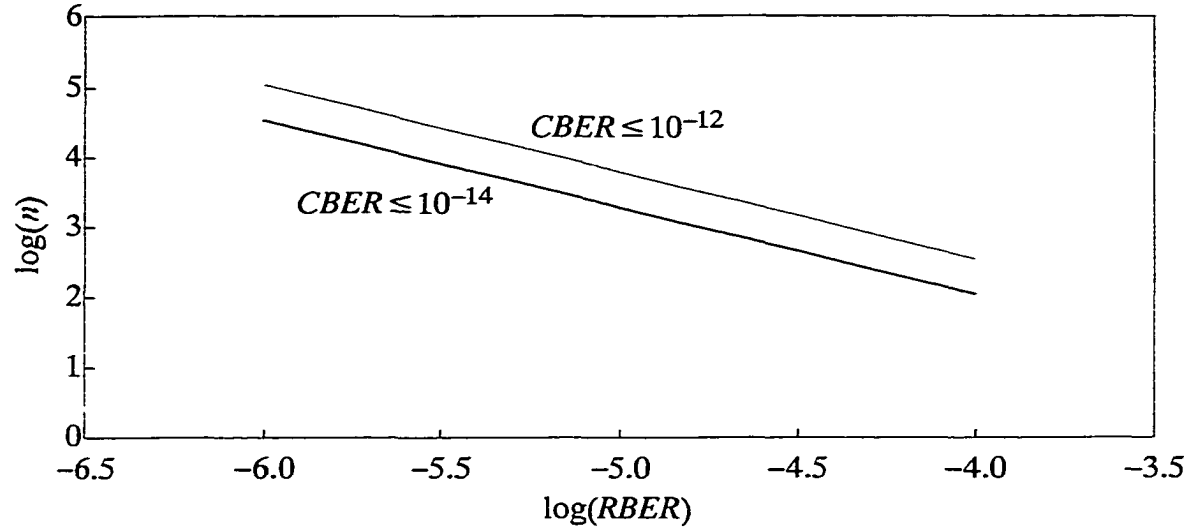


Figure 4.5: Sizes of 3–D row and column array codes which provide $CBER \leq 10^{-14}$ and $CBER \leq = 10^{-12}$ on random single bit errors.

An expression for *CBER* in terms of *RBER* and n, can be obtained by equating Equations 4.3 and 4.4 and solving for *CBER*. The code performance of 3–D array codes of different block sizes against single bit random errors is shown in Figure 4.6. The codes with smaller block sizes provide higher error correction capability.

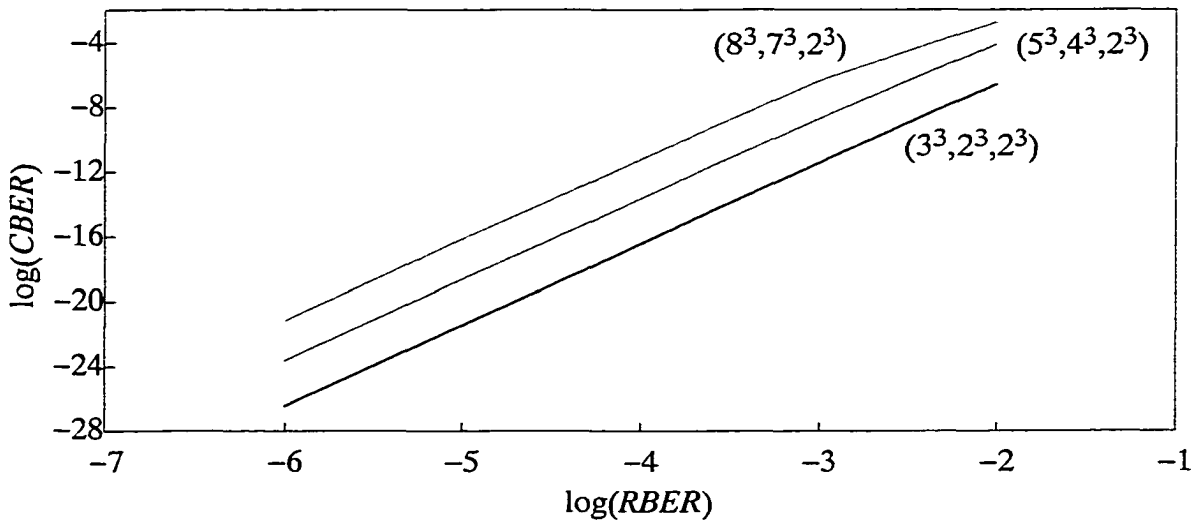


Figure 4.6: Performance of 3–D row and column array codes on random single bit errors.

A code with a larger block size is more capacity efficient as shown in Table 4.2. Efficient codes usually need a more complex decoder which operates at relatively low speed. If a simple or fast decoder is required, efficiency must be sacrificed to a certain extent.

Table 4.2: Code parameters for uniblock 3–D array codes.

code	n	k	d	r
$(8^3,7^3,2^3)$	512	343	8	0.670
$(5^3,4^3,2^3)$	125	64	8	0.512
$(3^3,2^3,2^3)$	27	8	8	0.296

To obtain a target *CBER* value of 10^{-14} to 10^{-12} when the typical *RBER* values range from 10^{-5} to 10^{-4}, the uniblock 3–D row and column array code is limited to a small 3–D block size (e.g., $8\times8\times8$). This fact is indicated by the performance curves shown in Figure 4.6. The multiblock 3–D row and column array code provides a way to obtain the target *CBER* values for typical *RBER* values mentioned previously while using larger page sizes. The code performance of the multiblock 3–D row and column array code is shown in Figure 4.7 for different 3–D block sizes (e.g., $81\times81\times3$, $135\times135\times3$, $270\times270\times3$, etc.).

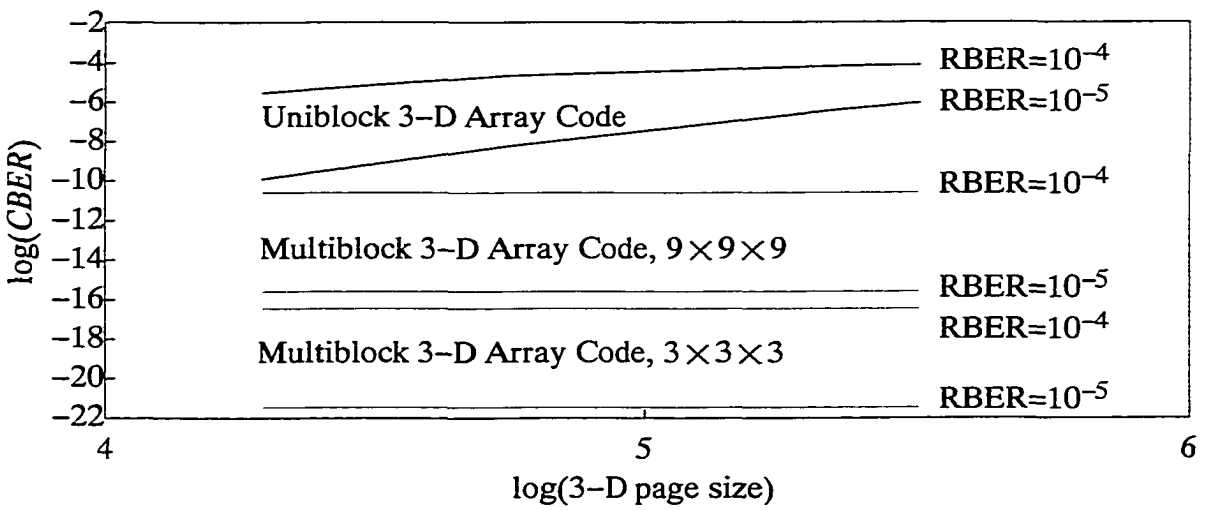


Figure 4.7: Performance of uniblock and multiblock 3–D row and column array codes for different *RBER* versus 3–D page size.

4.3 Analysis of 3–D Wing Array Codes

Figure 4.8 shows a 3–D wing (i.e., right triangular) code which is considered in this section for error detection and correction. Observe that this code does not have a parity check bit layer. The reason for choosing this code structure is explained later. Each layer of the 3–D wing code shown in Figure 4.8 is equivalent to the wing code described in section 3.3. Therefore, the same expressions used in section 3.3 apply to each layer of the 3–D wing array code shown in Figure 4.8.

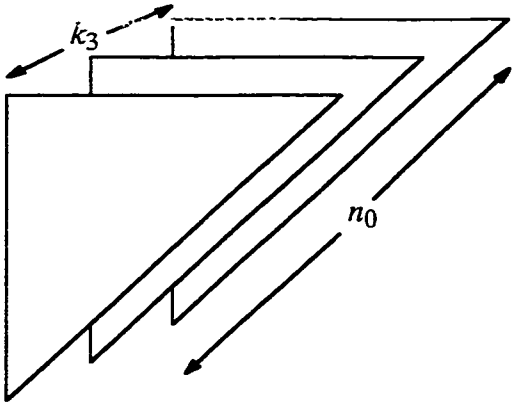


Figure 4.8: Uniblock 3–D wing array code.

Length of code: $n = \frac{n_0(n_0 + 1)k_3}{2}$

Information bits: $k = \frac{n_0(n_0 - 1)k_3}{2}$

Hamming distance: $d = 3$

Code rate: $r = \frac{k}{n} = \frac{(n_0 - 1)}{(n_0 + 1)}$

Coding overhead: $n - k = n_0 k_3$

Number of errors that can be detected: $t = k_3$

Number of errors that can be corrected: $c = k_3$

Type of errors that can be corrected: random

The parity check bits are generated along the hypotenuse of the right triangular information bit arrays. Each parity check bit located on the hypotenuse checks both a row and a column of the information bit array. This code detects and corrects one error per layer. When data are read, the intersection of two different ones in the parity check vectors give the position of the error in the information bit array. More than one error within the information bit array causes the presence of three or more ones in the parity check vectors and this code may not detect all cases, may miscorrect and may misdetect errors.

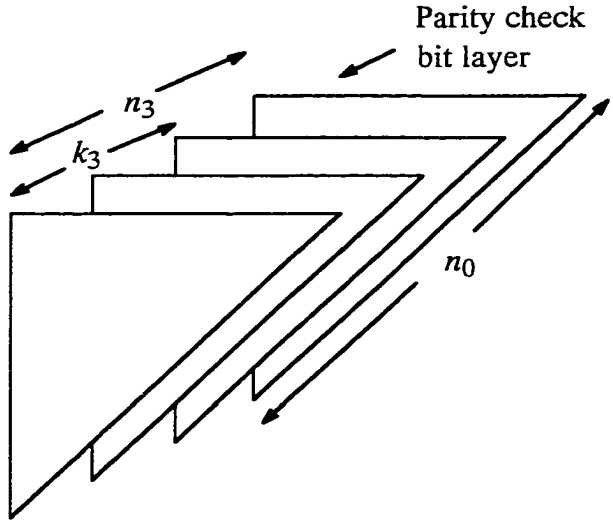


Figure 4.9: Uniblock 3–D wing array code with a parity bit check layer.

The reason the parity check bit layer is not used in the 3–D wing code is mainly because of the small increase in the error correcting capability of this code. Since the wing code in section 3.3 has distance, $d = 3$, it is capable of detecting and correcting one error. A 3–D wing code constructed as shown in Figure 4.9 has distance, $d = 5$. Therefore, this code can detect and correct two errors. However, the same number of errors (i.e., for $k_3 = 2$) or more errors (i.e., $k_3 > 2$) can be detected and corrected by using the wing code discussed in

section 3.3 and shown in Figure 3.13 for each layer of the 3–D wing code and not using a parity bit check layer.

The 3–D wing code may be used to perform EDC (i.e., protect) on data located at the corner and on a line which bisects the 90° angle included by the corner for multiple pages or layers of data. The multiple page regions not protected by the wing code require the use of a different code for protection. Two wing codes can be arranged, as shown in Figure 4.10, to form an 3–D rectangular shaped structure to protect multiple pages or layers of data. These 3–D rectangular shaped structures can be made from pairs of 3–D wing codes with each wing code having size $\frac{n_0(n_0 + 1)k_3}{2}$.

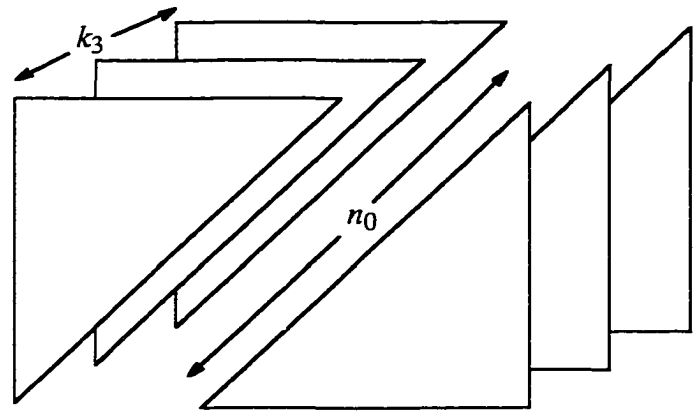


Figure 4.10: Uniblock 3–D wing array code applied to a 3–D rectangular shape.

The multiblock 3–D wing code shown in Figure 4.12 can be used to perform error detection and correction on applications with page sizes as large as 10^6 bits. This code is obtained by using the code shown in Figure 4.10 as a component code arranged in a $m_1 \times m_2$ array of 3–D code blocks. This multiblock 3–D wing code has parameters described by the following expressions.

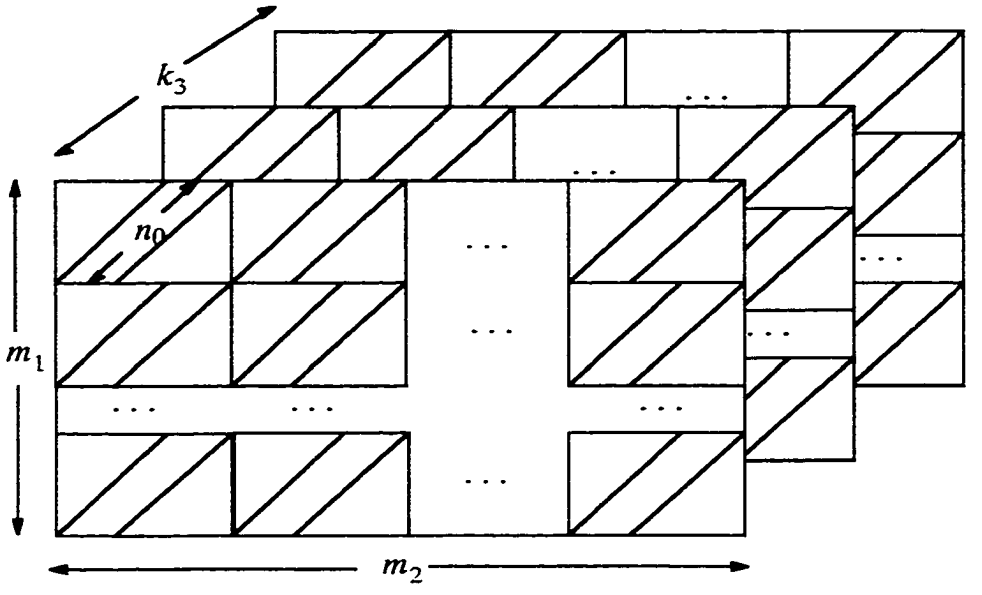


Figure 4.11: Multiblock 3–D wing array code.

Length of code: $n = m_1 m_2 n_0 (n_0 + 1) k_3$

Information bits: $k = m_1 m_2 n_0 (n_0 - 1) k_3$

Hamming distance: $d = 3$

Code rate: $r = \frac{k}{n} = \frac{(n_0 - 1)}{(n_0 + 1)}$

Coding overhead: $n - k = 2 m_1 m_2 n_0 k_3$

Number of errors that can be detected per triangular pair:

$$t = \{t_{\min} = 2k_3,\ t_{\max} = 2k_3 m_1 m_2\}$$

Number of errors that can be corrected per triangular pair:

$$c = \{c_{\min} = 2k_3,\ c_{\max} = 2k_3 m_1 m_2\}$$

Type of errors that can be corrected: random

When one error occurs in each triangular block, this code can detect and correct up to $2k_3 m_1 m_2$ errors. Otherwise no error detection or correction using the multiblock approach is possible.

4.3.1 Encoding for 3–D Wing Array Codes

Data encoding for a 3–D wing array code is shown in Figure 4.12. For each layer, data encoding is performed in the same way as was done for the wing code described in section 3.3.1. The parity check bits for each layer are located on the hypotenuse of a right triangular shape and are computed using odd parity.

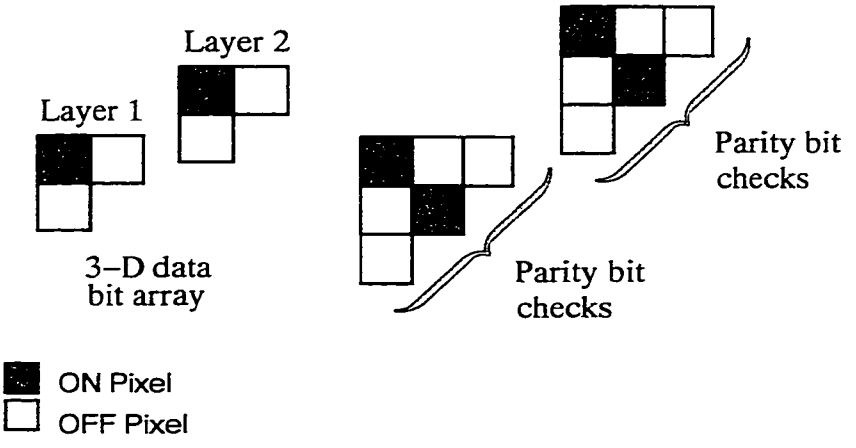


Figure 4.12: Example of data encoding applied to a 3–D wing array code.

4.3.2 Decoding for 3–D Wing Array Codes

Figure 4.13 shows that data decoding for a 3–D wing array code is performed in a way similar to that used by the wing code described in section 3.3.2. After a page of data is received at the photodetector array, the syndrome is computed using odd parity and error detection and correction are performed. The occurrence of two bit failures in the syndrome are used to identify the location of an error.

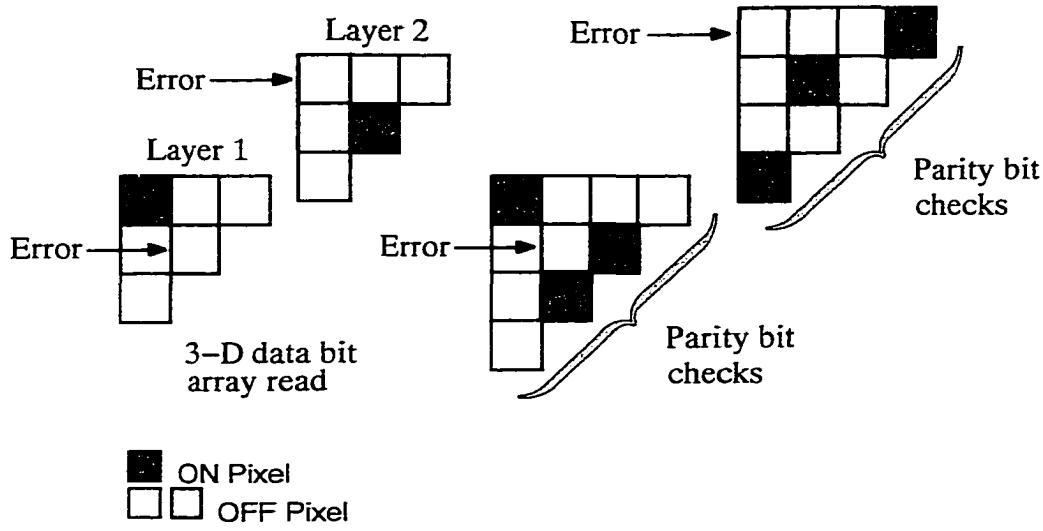


Figure 4.13: Example of data decoding applied to a 3–D wing array code.

4.3.3 Encoder for 3–D Wing Array Codes

Each layer of the 3–D wing array code shown in Figure 4.12 is equivalent to the wing code described in section 3.3.3. Although each layer of the 3–D wing array code is used in parallel with the other layers, each layer can be analyzed separately for time delays and hardware requirements. Using this analysis strategy, allows all of the expressions for the encoder time delays and hardware requirements developed in section 3.3.3 to apply to each layer of the 3–D wing array code shown in Figure 4.12. Refer to section 3.3.3 for a discussion of the hardware and time delay expressions.

4.3.4 Decoder for 3–D Wing Array Codes

A version of the wing code described in section 3.3.4 is replicated in each layer of the 3–D wing array code shown in Figure 4.13. While the layers of the 3–D wing array code are used in parallel, the time delays and hardware needed are analyzed for each layer separately. The expressions for the decoder time delays and hardware demands obtained from the analysis given in section 3.3.4 hold for each layer of the 3–D wing array code shown

in Figure 4.12. Therefore, the reader is referred to section 3.3.4 for a discussion of the hardware and time delay expressions.

4.3.5 Encoding and Decoding Multiblock 3–D Wing Array Codes

Expressions regarding hardware requirements and time delays for the multiblock wing code described in section 3.3.5 apply to each layer (i.e., layers occur in the k_3–direction) of the multiblock 3–D wing array code shown in Figure 4.11. This is true because each layer of the multiblock 3–D wing array code is a replicated version of the multiblock wing code described in section 3.3.5. For a discussion of the hardware and time delay expressions, refer to section 3.3.5.

4.3.6 Code Performance for 3–D Wing Array Codes

We know from section 4.3.4 that each layer of the 3–D wing array code is a replicated version of the wing array code discussed in section 3.3.4. From section 4.3.5, we know each layer of the multiblock 3–D wing array code is equivalent to the multiblock wing array code discussed in section 3.3.5. Therefore, each layer of the uniblock and multiblock 3–D wing array codes have a *CBER* performance which is equivalent to the *CBER* performance of the uniblock and multiblock wing array codes discussed in section 3.3.6. Refer to section 3.3.6 for a discussion of the *CBER* performance of wing array codes which are used as layers in 3–D wing array codes.

4.4 Analysis of Burst Error Correcting 3–D Array Codes

A 3–D array row and column array code used for correcting burst errors is shown in Figure 4.14. No parity bit check layer is utilized in this code because of hardware requirement and time delay issues discussed in sections 3.4.4 and 3.4.5. The BEC array code discussed in section 3.4 are used for each layer making up the BEC 3–D array codes.

Therefore, the expressions developed in section 3.4 apply to each layer of the BEC 3–D array code. The reader is urged to read section 3.4 as needed to understand the discussion given here.

Assume we want to detect and correct errors using the 3–D array code shown in Figure 4.14. If a burst error of length k_2 bits occurs, the information bit array must have parameters $k_1 \times k_2 \times k_3$ where $k_2 \geq 2(k_1 - 1)$. The parameters of BEC code discussed in section 3.4 are combined with the factor k_3 to obtain the following expressions for the parameter of the BEC 3–D array code.

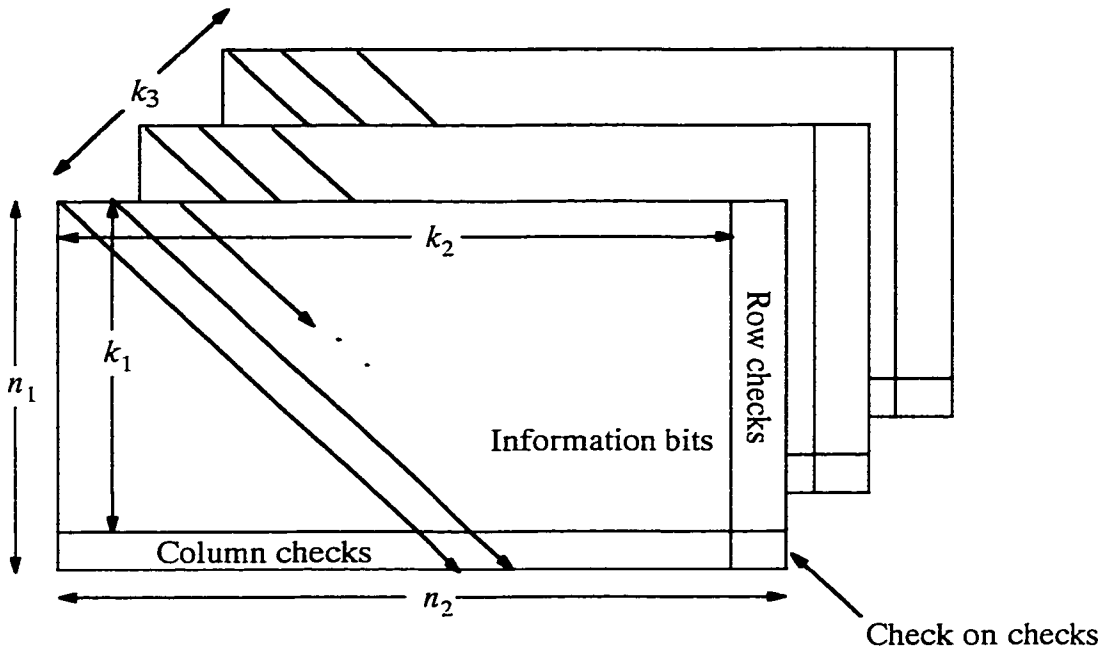


Figure 4.14: Burst error correcting 3–D array code.

Length of code:

$$n = n_1 n_2 k_3 = (k_1 + 1)(k_2 + 1)k_3 \geq (k_1 + 1)[2(k_1 - 1) + 1]k_3$$

Information bits: $k = (n_1 - 1)(n_2 - 1)k_3$

Hamming distance: $d = 4$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2}$

Coding overhead: $n - k = (n_1 + n_2 - 1)k_3$

Number of errors that can be detected: $t = k_1 k_3$

Number of errors that can be corrected: $c = k_1 k_3$

Type of errors that can be corrected: burst

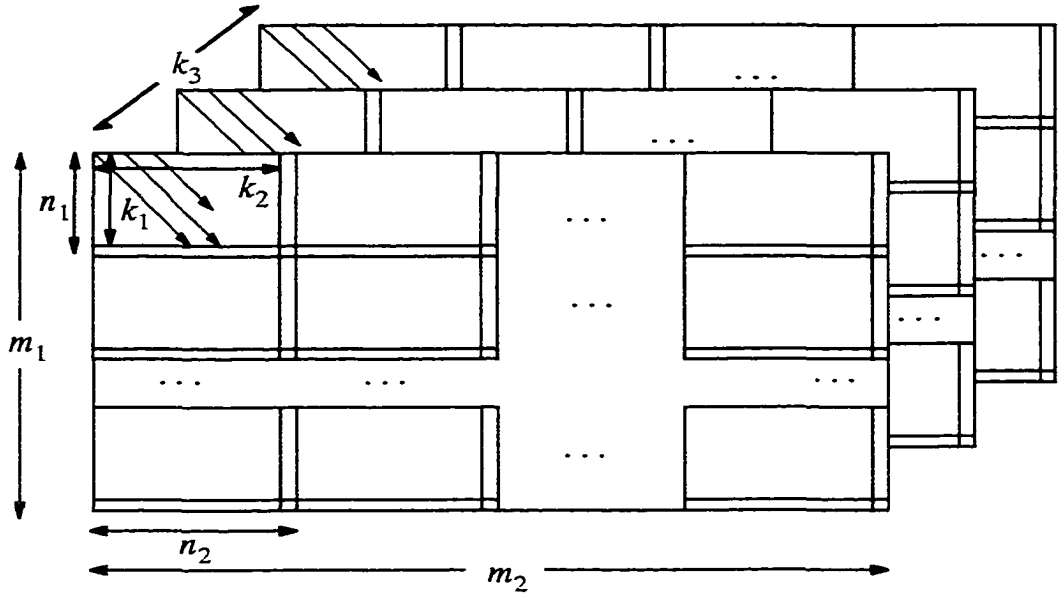


Figure 4.15: Multiblock burst error correcting 3–D array code.

Burst error detection and correction can be performed on page sizes as large as 10^6 bits using the multiblock BEC 3–D array code shown in Figure 4.15. This code is constructed from a $m_1 \times m_2$ array of $n_1 \times n_2 \times k_3$ code blocks. Including the factor k_3 in the expressions describing the parameters of the the multiblock BEC array code discussed in section 3.4, leads to the following expressions for the multiblock BEC 3–D array code.

Length of code:

$$n = m_1 m_2 (k_1 + 1)(k_2 + 1)k_3 \geq m_1 m_2 (k_1 + 1)[2(k_1 - 1) + 1]k_3$$

Information bits: $k = m_1 m_2 (n_1 - 1)(n_2 - 1)k_3$

Hamming distance: $d \geq 4$

Code rate: $r \geq \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2}$

Coding overhead: $n - k = m_1 m_2 (n_1 + n_2 - 1) k_3$

Number of errors that can be detected: $t = \{t_{min} = k_2, t_{max} = m_1 m_2 k_2 k_3\}$

Number of errors that can be corrected: $c = \{c_{min} = k_2, c_{max} = m_1 m_2 k_2 k_3\}$

Type of errors that can be corrected: burst

The occurrence of a burst error in each code block allows this code to detect and correct up to $m_1 m_2 k_2 k_3$ errors. Otherwise this code can not perform error detection and correction.

4.4.1 Encoding Burst Error Correcting 3-D Array Codes

Each layer of the BEC 3-D array code shown in Figure 4.16 performs data encoding in the same way as described in section 3.4.1. Even parity is used to compute diagonal and column parity bit checks. Row parity bit checks are computed as diagonal parity bit checks using a diagonal direction and column parity bit checks are computed across each column.

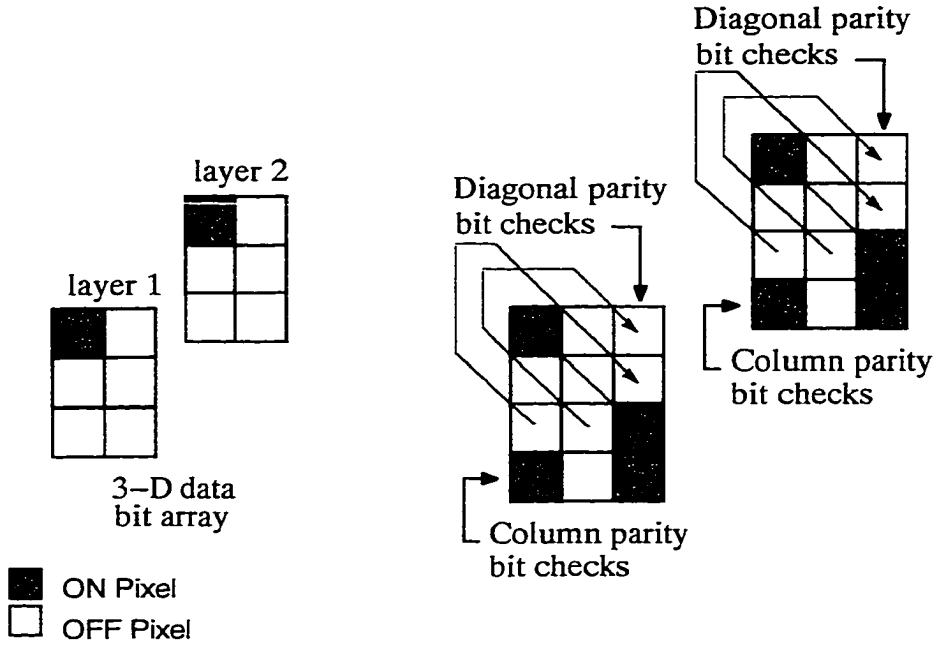


Figure 4.16: Data encoding applied to a burst error correcting 3–D array code.

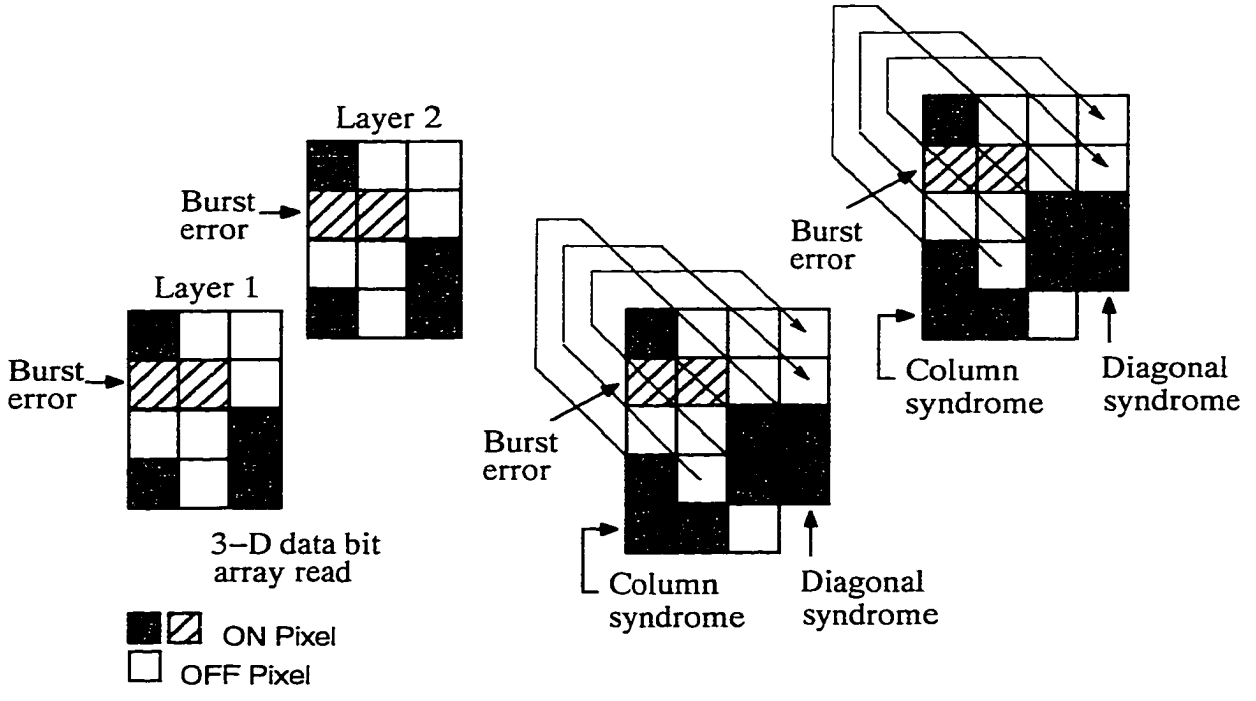


Figure 4.17: Data decoding applied to a burst error correcting 3–D array code.

4.4.2 Decoding Burst Error Correcting 3–D Array Codes

Data decoding as described in section 3.4.2 is performed on each layer of the BEC 3–D array code shown in Figure 4.17. The data decoding process involves receiving a page of data at the photodetector array, computing the syndromes (diagonal and column) and performing burst error detection and correction. Refer to section 3.4.2 for additional information on data decoding.

4.4.3 Encoder for Burst Error Correcting 3–D Array Codes

The BEC 3–D array code shown in Figure 4.16 is composed of layers equivalent to the BEC array code described in section 3.4.3. The encoder time delays and hardware requirements developed in section 3.4.3 apply to each layer of the BEC 3–D array code. Therefore, the reader is referred to section 3.4.3 for a discussion of these expressions.

4.4.4 Decoder for Burst Error Correcting 3–D Array Codes

Figure 4.15 shows a BEC 3–D array code made of layers which are replicated versions of the BEC array code discussed in section 3.4.4. The decoder time delays and hardware requirements described in section 3.4.4 are valid for each layer of the BEC 3–D array code. Refer to section 3.4.4 for a discussion of these time delay and hardware expressions.

4.4.5 Encoding and Decoding Multiblock BEC 3–D Array Codes

Using the multiblock BEC array code discussed in section 3.4.5, a multiblock BEC 3–D array code is obtained as shown in Figure 4.15. The hardware requirement and time delay expressions developed in section 3.4.5 apply for each layer of the multiblock BEC array code shown in Figure 4.15. For a discussion of these expressions, the reader is referred to section 3.4.5.

4.4.6 Code Performance of BEC 3–D Array Codes

The discussion given in section 4.4.4 used the BEC code described in section 3.4.4 to construct the uniblock BEC 3–D array code. Similarly, the multiblock BEC 3–D array code described in section 4.4.5 used the multiblock BEC array code described in section 3.4.5 to construct the multiblock BEC array code. Hence, the code performance of each layer of the BEC 3–D array codes will have the same code performance of the BEC array codes described in section 3.4.6. For a discussion of the code performance of these BEC array codes used as layers in the BEC 3–D array codes, refer to section 3.4.6.

4.5 Analysis of a Cluster Error Correcting 3–D Array Code

Figure 4.18 shows a 3–D array row and column array code which can be used as a CEC array code. Because of hardware demands and time delay issues discussed in sections 3.5.4 and 3.5.5, no parity bit check layer is used in this code. The CEC array code described in section 3.5 is used for each layer of the CEC 3–D array codes. This type of code structure allows the expressions developed in section 3.5 apply to each layer of the cluster correcting 3–D array code. Refer to section 3.5 as needed to understand the discussion given here.

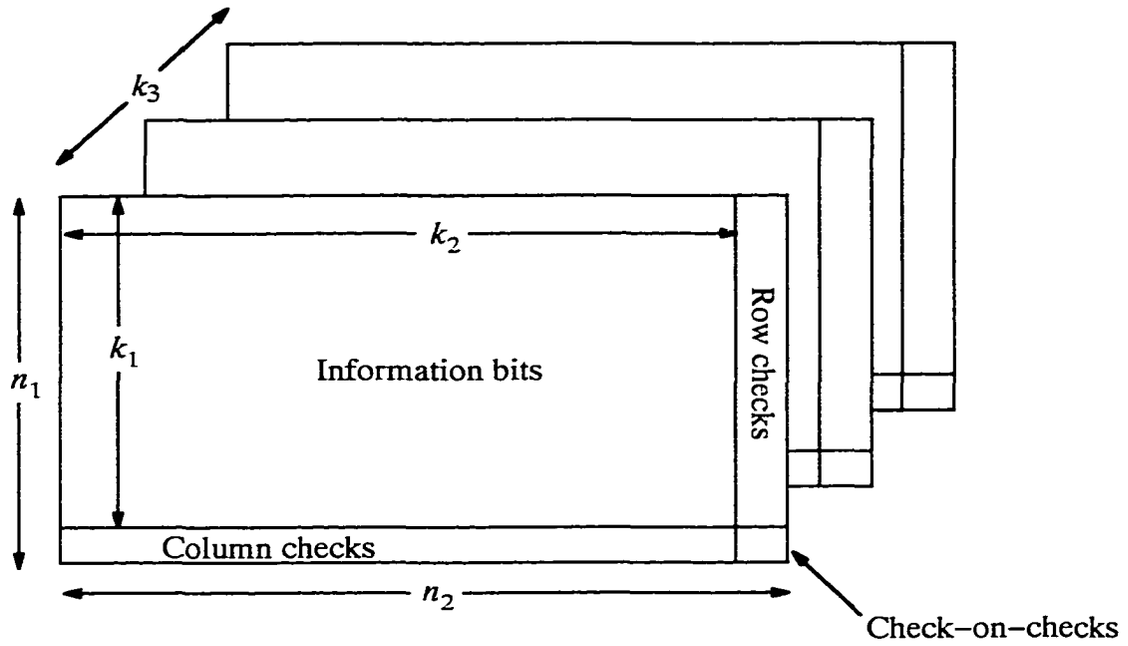


Figure 4.18: Uniblock cluster error correcting 3–D array code.

Assume a rectangular cluster error of size $b_1 \times b_2$ occurs. The array code consisting of the information and parity bits must have parameters $n_1 \times n_2 \times k_3$, where $n_1 \geq 2b_1b_2-b_1$, $n_2 \geq 2b_1b_2$, b_1 divides n_1 and b_2 divides n_2. For this code, the Hamming distance, d, of each layer is four and additional code parameters are given by the following expressions.

Length of code: $n = n_1n_2k_3 \geq (2b_1b_2 - b_1)2b_1b_2k_3$

Information bits: $k = (n_1 - 1)(n_2 - 1)k_3 = k_1k_2k_3$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1n_2} = \frac{k_1k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = (n_1 + n_2 - 1)k_3$

Number of errors that can be detected: $t = b_1b_2k_3$

Number of errors that can be corrected: $c = b_1b_2k_3$

Type of errors that can be corrected: cluster

Assume this code uses staggering first across rows and then across columns to achieve single cluster correction. A bit shift to the right by b_2 (i.e., $i \bmod b_1$) places is applied

to each row i. Then a bit shift downward by b_1 (i.e., j mod b_2) places is applied to each column j. This staggering process interleaves several row and column parity bits; however, no error within the cluster (block) of errors affect more than a single row and column parity bit. Therefore, the location of the cluster and the errors in it can be determined and corrected.

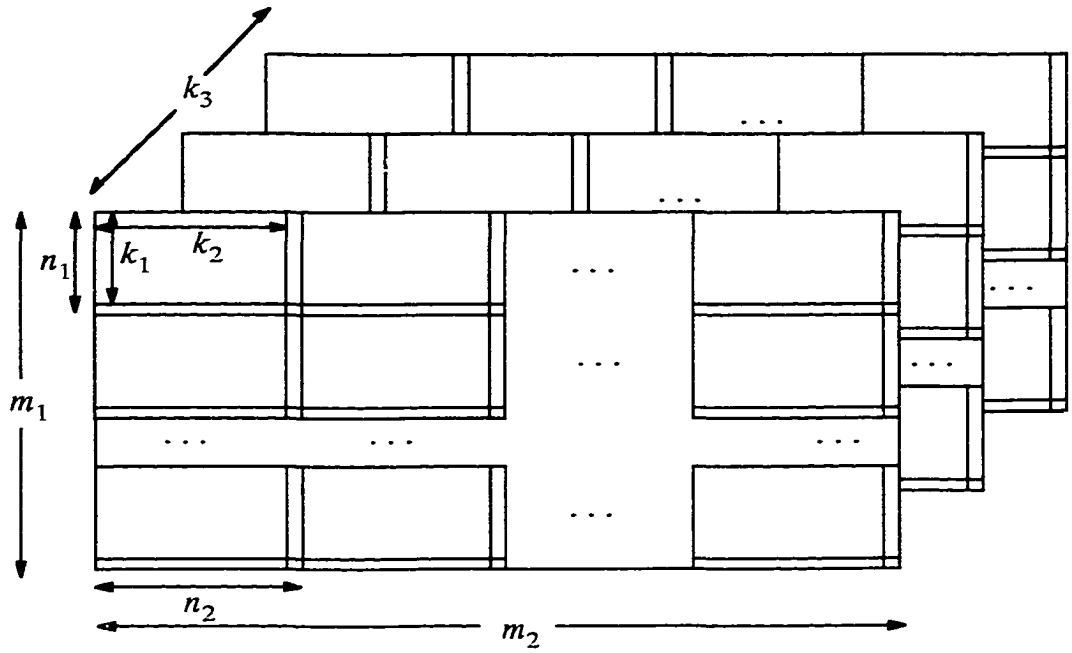


Figure 4.19: Multiblock cluster error correcting 3–D array code.

For large page sizes (e.g., a page containing 10^6 bits), the multiblock CEC 3–D array code shown in Figure 4.19 can be used for error detection and correction. Each layer has a Hamming distance of four. Additional parameters are given by the following expressions.

Length of code: $n = m_1 m_2 n_1 n_2 k_3 \geq m_1 m_2 (2b_1 b_2 - b_1) 2b_1 b_2 k_3$

Information bits: $k = m_1 m_2 (n_1 - 1)(n_2 - 1) k_3 = m_1 m_2 k_1 k_2 k_3$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)}{n_1 n_2} = \frac{k_1 k_2}{(k_1 + 1)(k_2 + 1)}$

Coding overhead: $n - k = m_1 m_2 (n_1 + n_2 - 1) k_3$

Number of errors that can be detected: $t = \{t_{\min} = b_1 b_2 k_3,\ t_{\max} = m_1 m_2 b_1 b_2 k_3\}$

Number of errors that can be corrected: $c = \{c_{\min} = b_1 b_2 k_3, c_{\max} = m_1 m_2 b_1 b_2 k_3\}$

Type of errors that can be corrected: cluster

4.5.1 Encoding for a Cluster Error Correcting 3–D Array Code

The data encoding of a CEC 3–D array code is shown in Figure 4.20. This encoding is performed for each layer making up the CEC 3–D array code as described in section 3.5.1. Row and column parity bit checks are computed using even parity. Since, the encoding has been described previously, the reader is referred to section 3.5.1 for additional information.

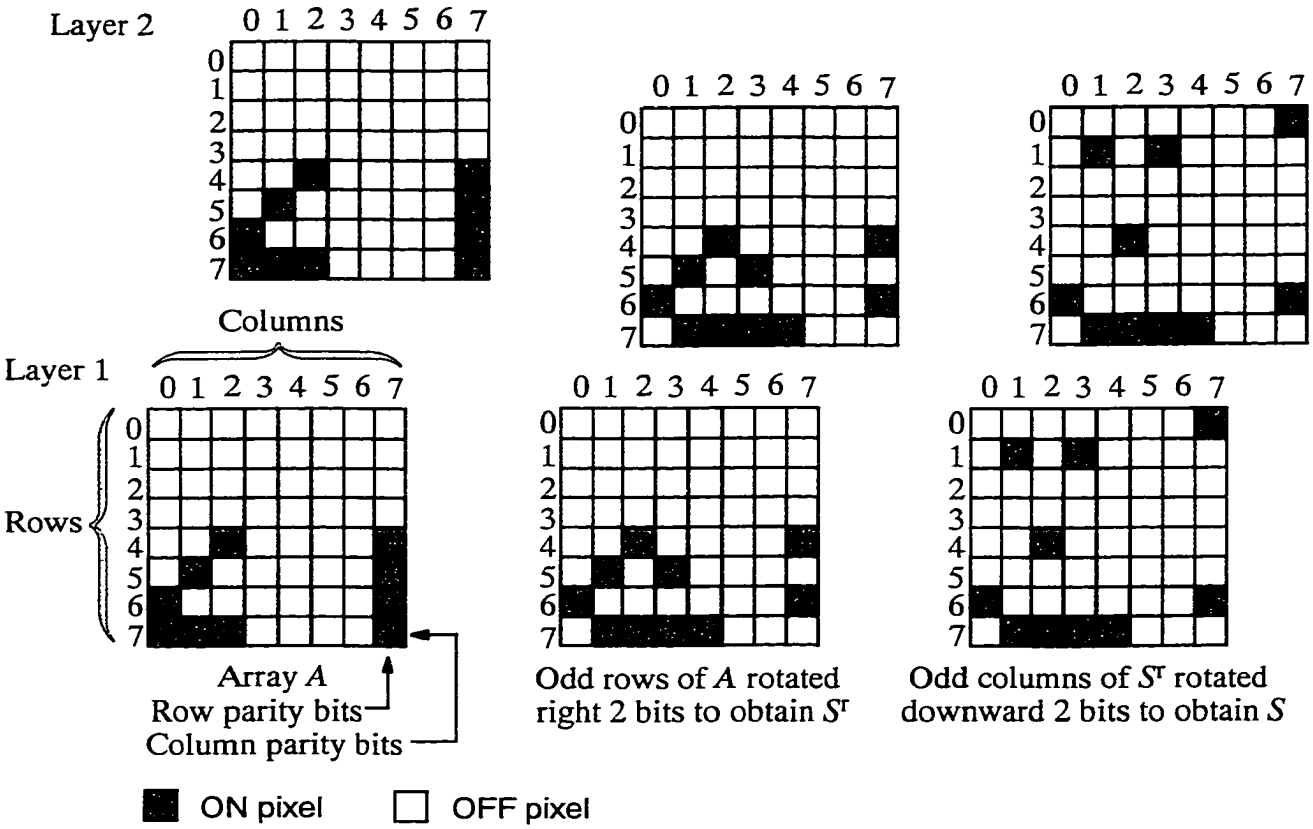


Figure 4.20: Example of data encoding applied to 3–D data array A yielding staggered arrays S^T and S for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$.

4.5.2 Decoding for a Cluster Error Correcting 3–D Array Code

Figure 4.21 shows how data decoding is accomplished for a CEC 3–D array code. Data decoding is performed on each layer of the 3–D array code as described in section 3.5.2.

The row and column syndromes are computed using even parity. Refer to the discussion in section 3.5.2, to review data decoding for a CEC array code.

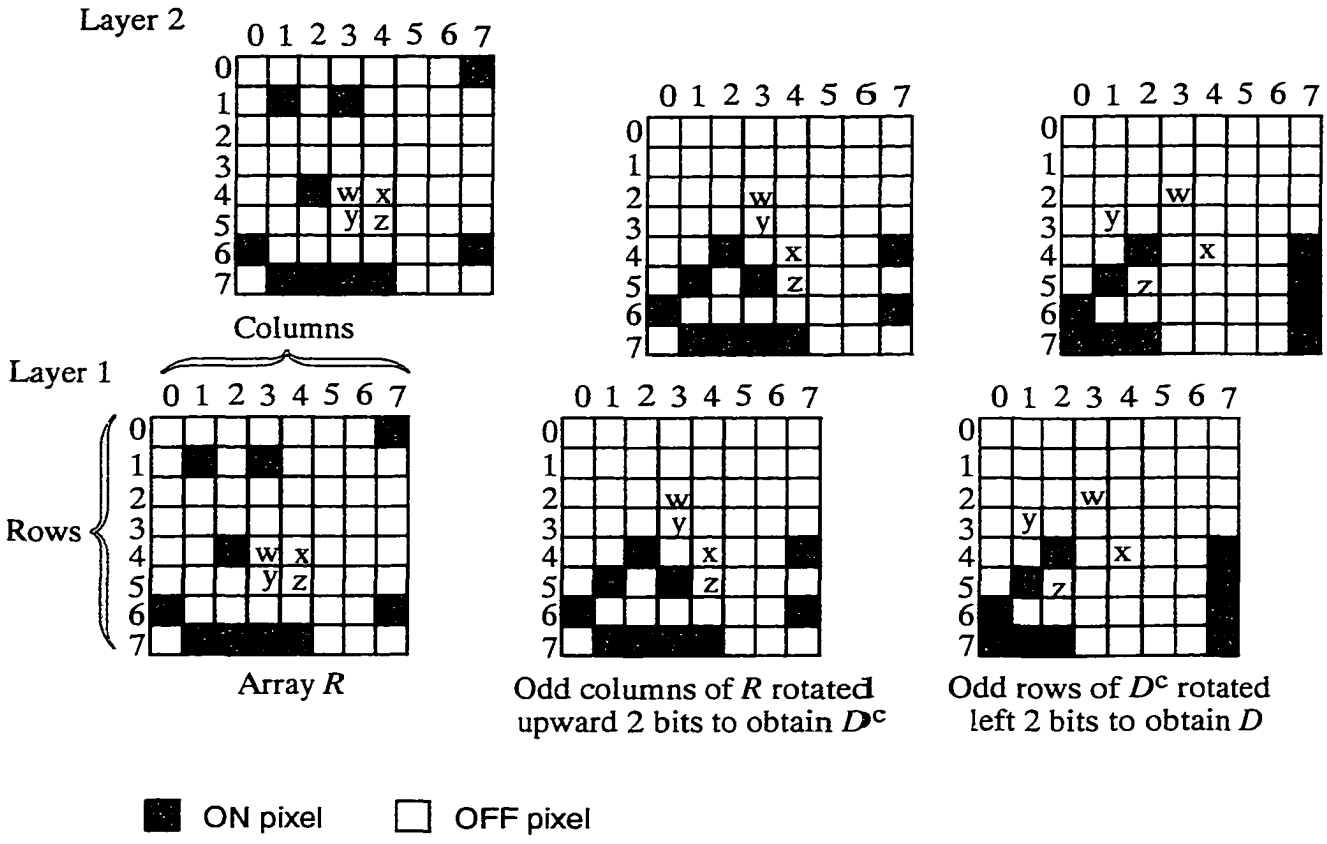


Figure 4.21: Example of data decoding applied to retrieved 3–D array R yielding destaggered arrays D^c and D for $n_1 = n_2 = 8$ and $b_1 = b_2 = 2$.

4.5.3 Encoder for a Cluster Error Correcting 3–D Array Code

The CEC 3–D array code shown in Figure 4.20 is built using layers. Each layer is equivalent to the CEC array code discussed in section 3.5.1. The layers of the 3–D array code are used in parallel. However, the time delay and hardware requirement for each layer can be analyzed separately. Therefore, the expressions developed in section 3.5.3 for the encoder time delay and hardware needed apply to each layer of the CEC 3–D array code. See section 3.5.3 for a discussion of these expressions.

4.5.4 Decoder for a Cluster Error Correcting 3–D Array Code

The CEC array code described in section 3.5.2 is used as a component to construct the CEC 3–D array code shown in Figure 4.21. The analysis given in section 3.5.4 regarding the decoder time delays and hardware requirements hold for each layer of the CEC 3–D array code shown in Figure 4.21. Therefore, refer to section 3.5.4 for the expressions describing hardware requirements and time delays.

4.5.5 Encoding and Decoding a Multiblock CEC 3–D Array Code

Figure 4.19 shows a multiblock CEC 3–D array code constructed from layers which are equivalent to the CEC array code discussed in section 3.5.5. For each layer of the multiblock CEC 3–D array code, the expressions for hardware demands and time delays described in section 3.5.5 apply. These expressions can be reviewed by referring to section 3.5.5.

4.5.6 Code Performance of CEC 3–D Array Codes

According to section 4.5.4, each layer of the uniblock CEC 3–D array code is equivalent to the CEC array code discussed in section 3.5.4. From section 4.5.5, each layer making up a multiblock CEC 3–D array code is a replicated version of the multiblock CEC array code discussed in section 3.5.5. Therefore, the CEC 3–D array codes have code performances for each layer which are equivalent to the CEC array codes discussed in section 3.5.6. The reader is referred to section 3.5.6 for a discussion of the *CBER* performance of CEC array codes which are used as layers in CEC 3–D array codes.

Chapter 5
Block 3–D Unequal Error Protection Array Codes

This chapter presents an introduction to block 3–D UEP array codes. The construction and characteristics of these codes for use on single bit random errors are considered. Next, an analysis is performed to provide some insight on the hardware complexity of the encoders and decoders for UEP codes. The *CBER* performance of 3–D EEP and UEP codes is discussed. Finally, an algorithm is presented for converting 3–D EEP codes into 3–D UEP codes programmatically.

5.1 Terminology

When some information bits are expected to have a greater number of errors than other information bits, UEP codes may provide a more optimal solution than EEP codes. Because selected or particular groups of information bits in UEP codes are provided different levels of protection, this leads to a more complicated expression for the distance parameter [DUN88, MOR95].

The separation vector [PIN88] for a linear (n,k) code C over the alphabet $GF(q)$ is denoted by $s(G) = s(G)_1, s(G)_2, ..., s(G)_k$ and is defined with respect to a generator G of C by Equation 5.1.

$$s(G)_i = \min\{wt(mG) | m \in GF(q)^k, m_i \neq 0\}, i = 1, 2, ...k \tag{5.1}$$

where $wt(mG)$ is the Hamming weight of $GF(q)^k$. If no more than $\frac{s(G)_i - 1}{2}$ errors have occurred in the transmitted code word and decoding is based on a maximum likelihood algorithm, $s(G)$ provides the correct interpretation of the i^{th} message symbol. For a linear UEP code $(n,k,s(G))$, let the protection level [PIN88], l_j be defined by

$$l_j = \left\lfloor \frac{s(G)_i - 1}{2} \right\rfloor \tag{5.2}$$

where j is the information bit position.

5.2 Analysis of 3–D Unequal Error Protection Array Codes

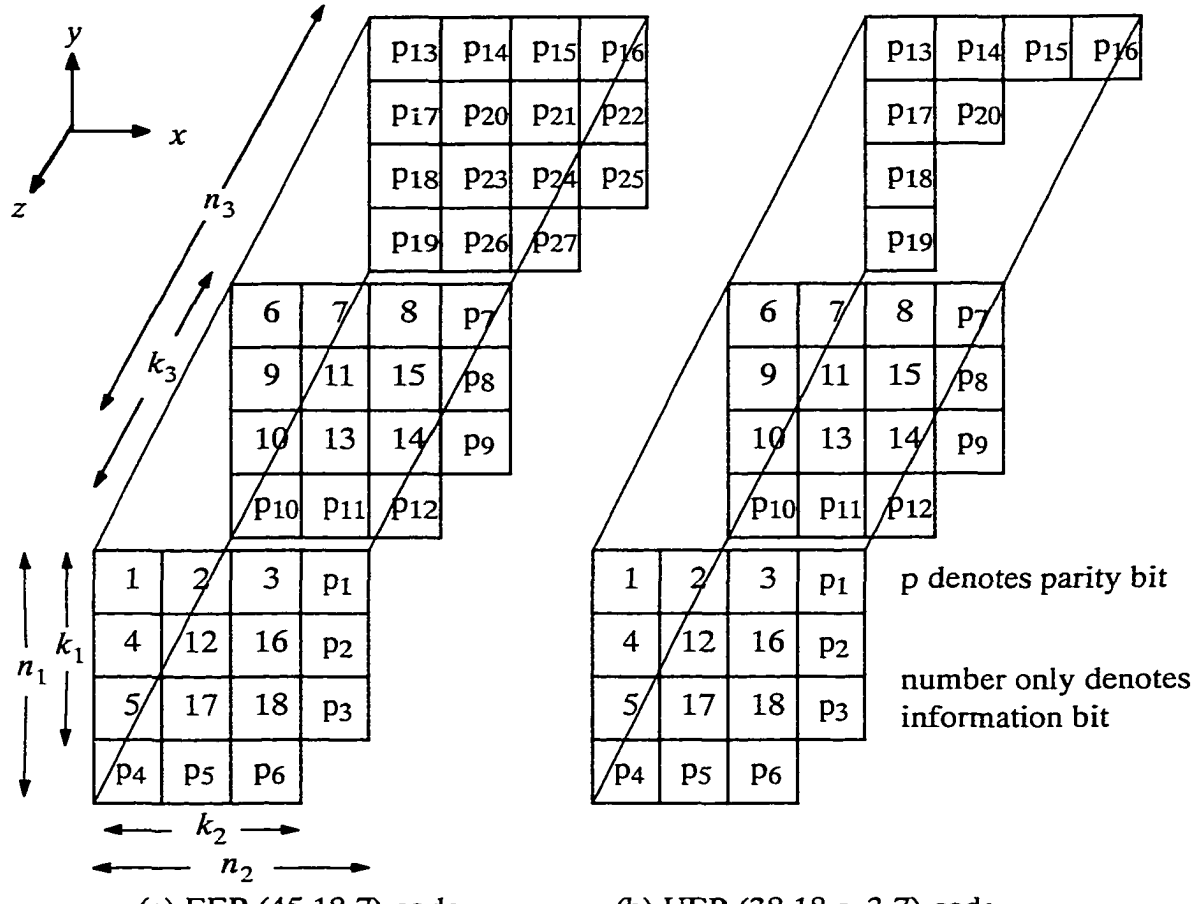
Consider the 3–D array code shown in Figure 5.1(a). There are several direction sets (e.g., rows, columns and layers) of parity bit checks. Since the direction sets used are orthogonal to each other, these direction sets can be used to increase the minimum distance of the code. By using the direction sets in combination, EEP codes with different (n,k,d) parameters can be formed. For the codes shown in Table 5.1, neither the check–on–checks nor the check–on–check–on–checks is used. However, if all of the parity bit checks making

Table 5.1: Direction sets of parity bit checks for corresponding 3–D EEP array codes.

Direction Sets of Parity Bit Checks	n	k	d
Rows	24	18	2
Rows and columns	30	18	3
Rows, columns and layers	45	18	7

up a direction set or combination of direction sets are not used collectively, UEP codes are produced [GAR98].

Assume a 3–D UEP code (n,d,$s(G)$) shown in Figure 5.1(b) is constructed for an application that requires information bit positions 1 to 12 be protected at level l_1 and information bit positions 13 to 18 be protected at level l_2 where $l_1 > l_2$. For information bit positions 1 to 12 and 13 to 18, s is 7 and 3, respectively. The parameters of the UEP code are (38,18,s) where s has the values 3 and 7.



(a) EEP (45,18,7) code (b) UEP (38,18,s=3,7) code

Figure 5.1: Information bit arrays with parity bit checks for 3–D (a) EEP and (b) UEP array codes.

Length of code: $n = (n_1 n_2 - 1)n_3 - 7$

Information bits: $k = (n_1 - 1)(n_2 - 1)(n_3 - 1)$

Distance vector: $s = 3, 7$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)(n_3 - 1)}{(n_1 n_2 - 1)n_3 - 7}$

Coding overhead: $n - k = (n_1 + n_2 - 2)n_3 + (n_1 - 1)n_2 - n_1 - 6$

Number of errors that can be detected: $t = 1$ (for $s = 3$), 3 (for $s = 7$)

Number of errors that can be corrected: $c = 1$ (for $s = 3$), 3 (for $s = 7$)

Type of errors that can be corrected: random

The parity check bits are generated along the rows, columns and layers of a $k_1 \times k_2 \times k_3$ data array as was done for the 3–D row and column array code discussed in section 4.2. When more than three errors occur in locations protected using a distance vector s = 7, or more than one error occurs in locations protected using a distance vector s = 3, this code can not provide a corrective action.

A multiblock 3–D UEP array code can be constructed for applications having page sizes as large as 10^6 bits. Using a $m_1 \times m_2$ array of 3–D UEP code block components having size $n_1 \times n_2 \times n_3$, the multiblock 3–D UEP array code obtained is shown in Figure 5.2. The parameters of this code are given by the following expressions.

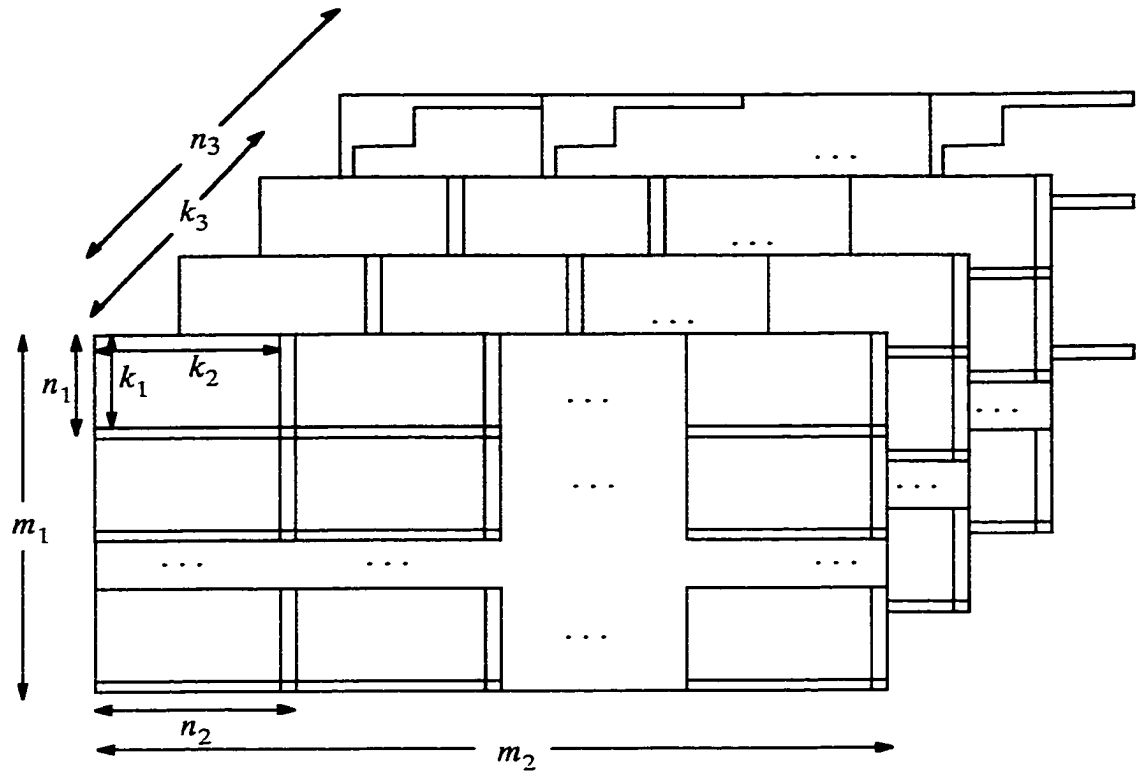


Figure 5.2: Multiblock 3–D UEP array code.

Length of code: $n = m_1 m_2 (n_1 n_2 - 1) n_3 - 7$

Information bits: $k = m_1 m_2 (n_1 - 1)(n_2 - 1)(n_3 - 1)$

Distance vector: $s = 3, 7$

Code rate: $r = \frac{k}{n} = \frac{(n_1 - 1)(n_2 - 1)(n_3 - 1)}{(n_1 n_2 - 1) n_3 - 7}$

Coding overhead: $n - k = m_1 m_2 [(n_1 + n_2 - 2) n_3 + (n_1 - 1) n_2 - n_1 - 6]$

Number of errors that can be detected: $t = \{t_{\min} = 1, t_{\max} = m_1 m_2\}$ for $s = 3$

Number of errors that can be detected: $t = \{t_{\min} = 3, t_{\max} = 3 m_1 m_2\}$ for $s = 7$

Number of errors that can be corrected: $c = \{c_{\min} = 1, c_{\max} = m_1 m_2\}$ for $s = 3$

Number of errors that can be corrected: $c = \{c_{\min} = 3, c_{\max} = 3 m_1 m_2\}$ for $s = 7$

Type of errors that can be corrected: random

This code can provide a corrective action as long as three or fewer errors occur in locations which are associated with distance vector s = 7. For locations associated with distance vector s = 3, a corrective action can be performed when a single error occurs. Otherwise, no corrective action can be taken.

5.2.1 Encoding a 3–D Unequal Error Protection Array Code

Figure 5.3 shows data encoding for a 3–D UEP array code. The row, column and layer parity bit checks are computed using even parity. The code block stored in memory consists of the data array and parity bit checks.

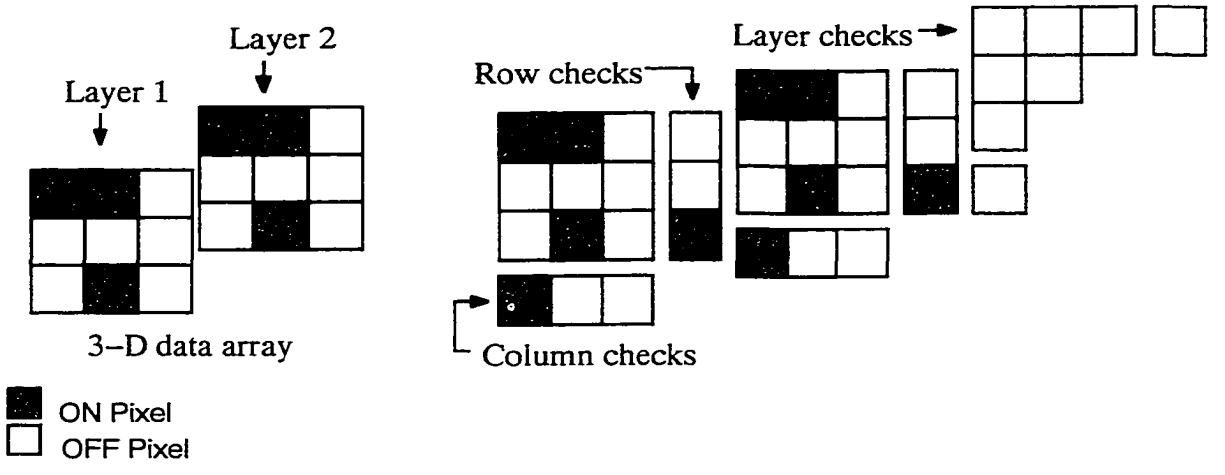


Figure 5.3: Data encoding for a 3–D UEP array code.

5.2.2 Decoding a 3–D Unequal Error Protection Array Code

Data decoding for a 3–D UEP array code is shown in Figure 5.4. After a page of data has been received at the photodetector array, computation of the row, column and layer checks occurs. The row, column and layer checks can be used to locate the occurrence of errors.

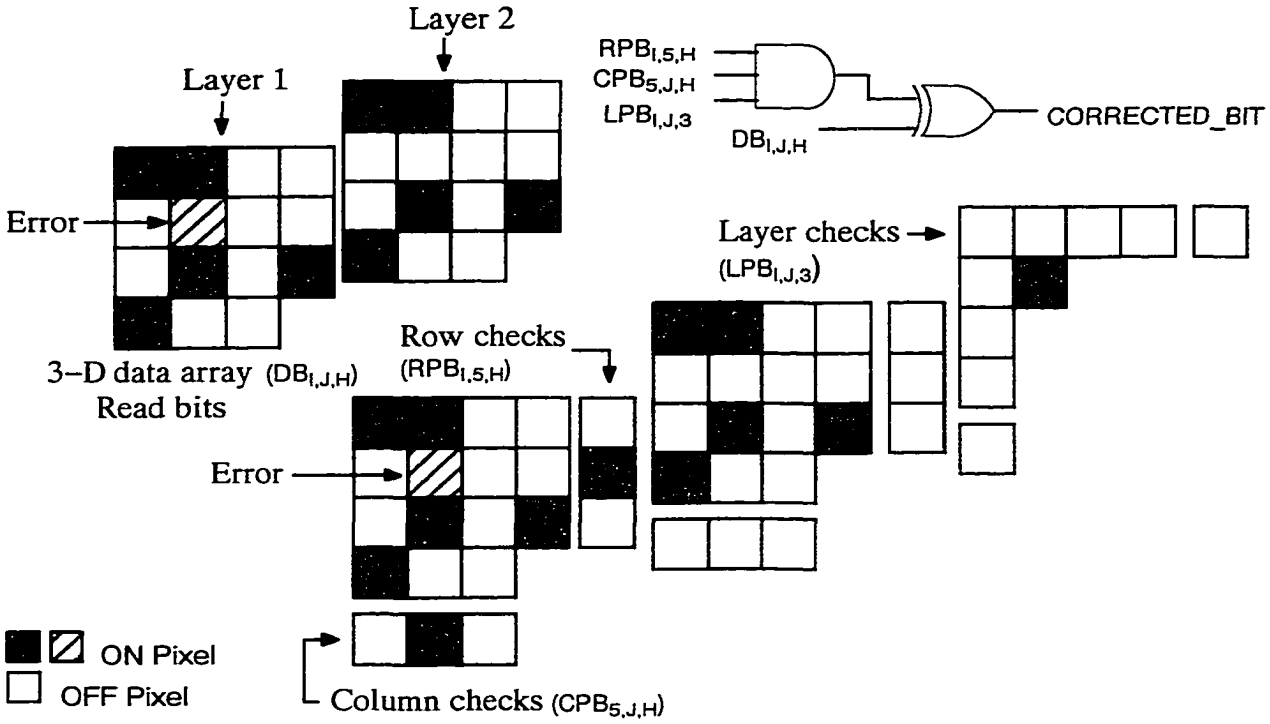


Figure 5.4: Data decoding for a 3–D UEP array code.

5.2.3 Encoder for a 3–D Unequal Error Protection Array Code

The encoder for the array code shown in Figure 5.1(b) must compute parity bit checks across rows, columns and layers. After adjustments are made for not using all of the parity bit checks in the parity bit check layer, the expressions for the time delay and hardware demands described in section 4.2.3 can be used here. Specifically, the hardware needed is $H_{3ae} = (k_1 - 1)k_2k_3 + (k_1 - 1) + (k_2 - 1)k_1k_3 + (k_2 - 1) + (k_3 - 1)(n_1n_2 - 8)$ gates. The time delay is $T_{3ae} = T_{3re} = \max(\lceil \log_2 k_1 \rceil, \lceil \log_2 k_2 \rceil, \lceil \log_2 k_3 \rceil)$ gate delays and T_{3re} is defined in section 4.2.3. The time delay expression for the 3–D unequal error protection array code is identical to the 3–D row and column array code (i.e., equal error protection array code) developed in section 4.2.3 because the number of layers used in these codes are the same.

5.2.4 Decoder for a 3–D Unequal Error Protection Array Code

The decoder hardware expression discussed in section 4.2.4 need some modification before they can be applied to the unequal protection array code being discussed. However, the decoder time delay expressions developed in section 4.2.4 can be used unchanged for the UEP array code considered here. For example, the hardware required to latch the data is $H_{3adl} = (n_1n_2 - 1)k_3 + (n_1n_2 - 8)$ gates. From section 4.2.4, the time delay experienced in latching the data is $T_{3adl} = T_l = 6$ gate delays.

The expressions for the hardware needed and the time delay encountered to compute the row, column and layer syndromes make use of the corresponding expressions developed in section 4.2.4. The hardware needed is $H_{3ads} = 2k_1k_2k_3 + k_1 + k_2 + (k_3 - 1)(n_1n_2 - 8)$ gates. The time delay is $T_{3ads} = T_{3rds} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil)$ gate delays and the meaning of T_{3rds} is explained in section 4.2.4.

The analysis given in section 4.2.4 for error detection an correction applies to Figure 5.1(b). Accounting for hardware needed to detect an uncorrectable error, the hardware required can be shown to be $H_{3adc} = 2k_1k_2k_3 + 2(k_1k_2 - 8) + (k_1 + k_2 - 1)k_3 + k_1 + k_2 - 1$ gates. The time delay expressions from section 4.2.4 are repeated here for convenience. Specifically, if an uncorrectable error has not occurred, $T_{3adc} = T_{3rdc} = 2$ gate delays otherwise $T_{3adc} = T_{3rdc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 2$ gate delays.

Therefore, the total hardware required is $H_{3ad} = H_{3adl} + H_{3ads} + H_{3adc} = [9(n_1n_2 - 1) + 4k_1k_2 + k_1 + k_2 + n_1n_2 - 9]k_3 + 2(k_1 + k_2) + 8n_1n_2 + 2k_1k_2 - 81$ gates. If an uncorrectable error has not occurred, the total time delay is $T_{3ad} = T_l + T_{3ads} + T_{3adc} = \max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 8$ gate delays otherwise $T_{3ad} = 2\max(\lceil \log_2 n_1 \rceil, \lceil \log_2 n_2 \rceil, \lceil \log_2 n_3 \rceil) + 8$ gate delays.

5.2.5 Encoding and Decoding a Multiblock 3–D UEP Array Code

The hardware requirements and time delay associated with the multiblock 3–D UEP array code follows the same analysis given in section 4.2.5 with minor changes. The expressions given in section 4.2.5 are valid for the multiblock 3–D UEP array code after replacing the encoding delay T_{3re} with T_{3ae}, the decoding delay T_{3rd} with T_{3ad}, the encoding hardware needed H_{3rd} with H_{3ad}, the decoding hardware needed H_{3rd} with H_{3ad}. For the global–AND–gate the time delay T_{3gAr} must be replace with T_{3gAa} and the associated hardware required H_{3gAr} must be replaced with H_{3gAa}.

5.2.6 Code Performance of 3–D UEP and EEP Array Codes

Code simulations were used to study the code performance of the 3–D EEP and UEP array codes. These simulations assume a Gaussian noise channel introduces random single bit errors during the intensity detection process. The EEP code shown in Figure 5.1(a) has parameters (n,k,d)=(45,18,7) and supports detection and correction of three errors. However, the UEP code has parameters $(n,k,s(G))$=(38,18,s=3,7) with s = 3 supporting detection and correction of 1 error and with s = 7 supporting detection and correction of three errors. From section 3.2.6, we know the probability, $P_{correct}$, that no error, e, passes through the decoder can be expressed in terms of e, n, and $RBER$ given by

$$P_{correct} = \sum_{e=0}^{t} \binom{n}{e} RBER^{e}(1 - RBER)^{n-e}. \tag{5.3}$$

with t = 3 corresponding to d = 7 for the EEP code, t = 1 corresponding to s =3 for the UEP code and t = 3 corresponding to s = 7 for the UEP code. According to Equation 3.4, repeated below for convenience, the probability, $P_{correct}$, that all bits are correctly decoded can also be expressed in terms of $CBER$ and n as

$$P_{correct} = (1 - CBER)^{n}. \tag{5.4}$$

Using Equations 5.3 and 5.4, *CBER* can be expressed in terms of e, n and *RBER*. One equation with $t = 3$ is obtained for the EEP array code. Two equations, one for $t = 1$ and another for $t = 3$, are obtained for the UEP array code.

The simulated performance of the 3–D EEP (45,18,7) and UEP (38,18,s) array codes, where s has values of 3 and 7, against random single bit errors is shown in Figure 5.5. Comparing different codes with the same separation (i.e., distance structure) in Figure 5.5, shows *CBER* depends on both the distance structure and rate of the code. The protection received by certain information bits in the UEP (38,18,$s = 7$) code is the same as that received by the information bits in the EEP (45,18,7) code.

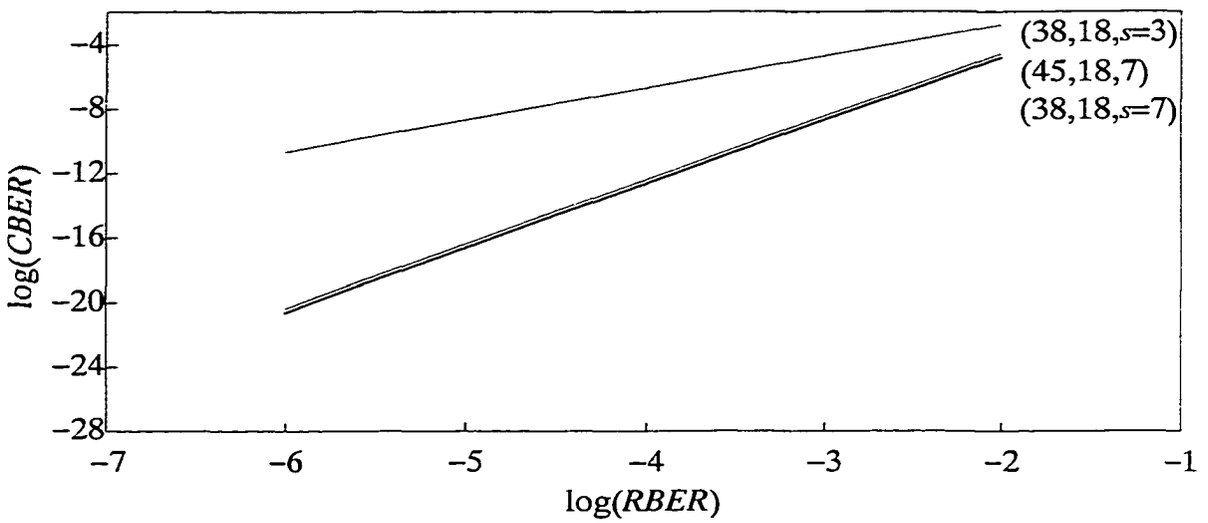


Figure 5.5: Performance of 3–D EEP and UEP array codes on single bit random errors.

When cross talk noise occurs, this noise source increases the *NSR* by a greater amount at the edges than at the center of the output plane and decreases the storage capacity of the memory system. The code parameters and *CBER* performance of EEP and UEP codes offer different approaches to providing error detection and correction for this type of noise source. The 3–D EEP array codes provide the same level of error protection to all the information bits in a page. The 3–D UEP codes provide selected groups of information bits with different

levels of protection in a page. The algorithm presented in the next section shows how EEP codes can be changed to UEP codes programmatically [GAR99]. The *CBER* performance of the codes depends on both the distance structure and rate of the code.

5.3 EEP Code to UEP Code Conversion Algorithm

Suppose the probability density function [GOO85, JAI89] $p_U(u)$ of the random variable $U \geq 0$ is known. To transform U to random variable $V \geq 0$ with a specified probability density function $p_V(v)$, a uniform random variable can be defined as

$$c = \int_0^u p_U(y)dy = G_U(u) \tag{5.5}$$

that also satisfies the expression

$$c = \int_0^v p_V(z)dz = G_V(v) \tag{5.6}$$

Eliminating c in Equations 5.5 and 5.6 leads to the expression [JAI89, CAS96]

$$v = G_V^{-1}(G_U(u)). \tag{5.7}$$

When U and V are discrete values with values u_i and v_j with probability density functions $p_U(u_i)$ and $p_V(v_j)$ respectively, Equation 5.7 can be implemented approximately using equations

$$c = \sum_{i=1}^{k} p_U(u_i) \tag{5.8}$$

and

$$cs = \sum_{j=1}^{k} p_V(v_j). \tag{5.9}$$

For the smallest value of j, let h be the value cs such that $cs - c \geq 0$; then the input u maps to a corresponding $h = v_j$.

The major steps of an algorithm are described which provides a way to change from the original histogram of parity bit checks for an EEP code to a desired histogram of parity bit checks for a UEP code programmatically. First, for the direction set of interest, compute the original histogram (i.e., the relative frequency measure) of parity bit checks. Second, compute the cumulative distribution function, (CDF) for the original histogram of parity bit checks. Third, use a specified or desired histogram of parity bit checks. Fourth, compute the desired cumulative distribution function, $CDF_{specified}$. Fifth, compare the cumulative distribution values and use the single parity bit check rule for proper assignment of the parity bit checks.

The algorithm in a more detailed form is given below:

1. yes ← 1, no ← 0, first ← startNum, last ← endNum
2. Initialize arrays parityBitNum[ ], parityBitFreq[ ], specParityBitFreq[ ], tempNum[ ], tempFreq[ ]
3. IF parityBitFreq[ ] = specParityBitFreq[ ] THEN exit algorithm
4. Compute cumulative distribution function (CDF) using Equation 5.8
5. Compute specified cumulative distribution function ($CDF_{specified}$) using Equation 5.9
6. FOR i = first to last
7. j ← 1, stop ← no
8. WHILE stop = no
9. IF cs[j] – c[i] ≥ 0 THEN
10. tempFreq[i] ← 1
11. tempNum[i] ← parityBitNum[j]
12. stop ← yes
13. ELSE increment j

14. FOR i = first to last

15. IF tempNum[i] = parityBitNum[i] THEN continue

16. ELSE

17. tempFreq[i] ← 0

18. tempNum[i] ← parityBitNum[i]

An example is used to illustrate this algorithm. A section of the information bits and parity bit checks for the array code from Figure 5.1(a) are selected and shown in Figure 5.6. The desired or specified histogram of parity bit checks shown in Figure 5.7 is a modified version of the original histogram shown in Figure 5.6. The relative frequency measure is defined in Equation 5.10

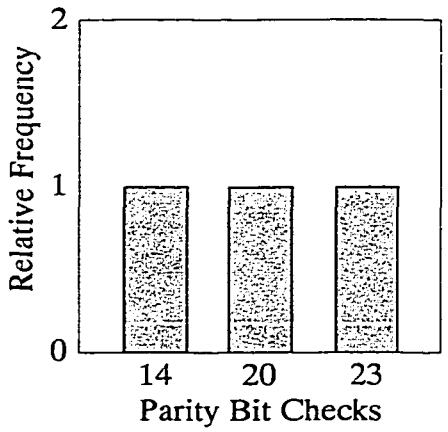


Figure 5.6: Histogram of parity bit checks for a section of the EEP (45,18,7) array code.

$$p_U(u_i) = \frac{n_i}{n} \tag{5.10}$$

where n_i is the ith parity bit check and n is the total number of parity bit checks for a direction set. From the histogram in Figure 5.6, the specific values for $p_U(u_i)$ are shown in Equation 5.11.

$$p_U(u_{14}) = p_U(u_{20}) = p_U(u_{23}) = \frac{1}{3} = 0.3333 \tag{5.11}$$

The cumulative distribution function, CDF, is given by

$$c_i = \sum_{i=1}^{k} p_U(u_i) \tag{5.12}$$

and from the histogram in Figure 5.6, the specific values for c_i are shown in Equations 5.13 to 5.15.

$$c_1 = 0.3333 \tag{5.13}$$

$$c_2 = 0.3333 + 0.3333 = 0.6666 \tag{5.14}$$

$$c_3 = 0.6666 + 0.3333 = 0.9999 \tag{5.15}$$

Assume the desired or specified histogram is as shown in Figure 5.7.

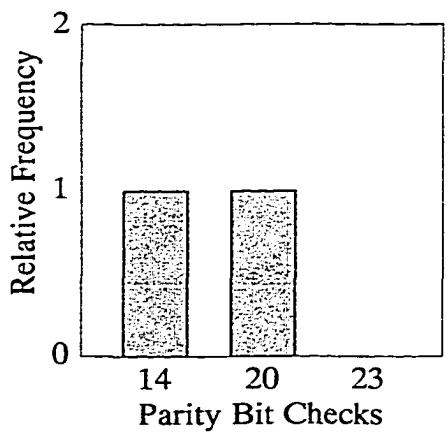


Figure 5.7: Specified histogram of parity bit checks for desired UEP array code.

The cumulative distribution function, $CDF_{specified}$, is given by

$$cs_j = \sum_{j=1}^{k} p_U(u_j) \tag{5.16}$$

and from the specified histogram in Figure 5.7, the specific values for cs_j are shown in Equations 5.17 to 5.19.

$$cs_1 = 0.5 \tag{5.17}$$

$$cs_2 = 0.5 + 0.5 = 1 \tag{5.18}$$

$$cs_3 = 1 + 0 = 1 \tag{5.19}$$

Observe the single parity bit check rule shown in Figure 5.8. This rule states that parity bit check, p_{14}, is computed from information bit positions 2 and 7, parity bit check, p_{20}, is computed from information bit positions 12 and 11 and parity bit check, p_{23}, is computed from information bit positions 17 and 13.

2	7	p_{14}
12	11	p_{20}
17	13	p_{23}

Figure 5.8: Information bit array with parity bit checks for a section of the EEP (45,18,7) array code.

From Equations 5.13 and 5.17, $cs_1 - c_1 \geq 0$, $c_1 = 0.3333 < 0.5 = cs_1 \Rightarrow h_1 = p_{14}$, use p_{14}.

From Equations 5.14 and 5.18, $cs_2 - c_2 \geq 0$, $c_2 = 0.6666 < 1 = cs_2 \Rightarrow h_2 = p_{20}$, use p_{20}.

From Equations 5.15 and 5.19, $cs_2 - c_3 \geq 0$, $c_3 = 0.9999 < 1 = cs_2 \Rightarrow h_2 = p_{20}$, use p_{20}.

Using the single parity bit check rule, parity bit check p_{20} can not be used or assigned to row 3. Therefore the parity bit check, p_{20}, for row 3 is omitted and the desired histogram shown in Figure 5.7 is obtained.

The algorithm presented shows how EEP codes can be changed to UEP codes programmatically. Although, the UEP codes can be used in place of EEP codes to provide different levels of protection to information bits stored in volume optical memories, UEP codes will most likely be used to provide different levels of error protection in other applications. For some numerical data applications (e.g., target data may have reliability

requirements that vary within a block of data), errors in the sign or in the most significant digits can be more serious than errors in the least significant digits. For some applications involving multiuser channels or computer networks, messages may require more protection against different noise levels with different levels of importance.

Chapter 6
Performance Analysis

In this chapter, a theoretical analysis of several features of a holographic data storage system is described. Specifically, storage capacity, decode time, transfer rate and data encoding are the features considered. Storage capacity and transfer rate are features in which volume holographic storage systems promise to provide significant gains over conventional (e.g., magnetic and optical disks) storage methods. From this analysis, insight is gained regarding the interactions of these features.

6.1 Storage Capacity

For a holographic data storage system storing uncoded holograms, the storage capacity, N_u, in bits can be expressed as

$$N_u = nmM_uL \tag{6.1}$$

where n is the length (i.e., the number of bits) of a code block,

m is the number of code blocks,

M_u is the number of uncoded holograms which are multiplexed pages per location and L is the number of locations in which holograms are stored.

When coded pages are stored the storage capacity, N_c, in bits can be expressed as

$$N_c = kmM_cL \tag{6.2}$$

where k is the number of information bits in a code block, M_c is the number of coded holograms which are multiplexed pages per location and the other factors have the same meaning mentioned above.

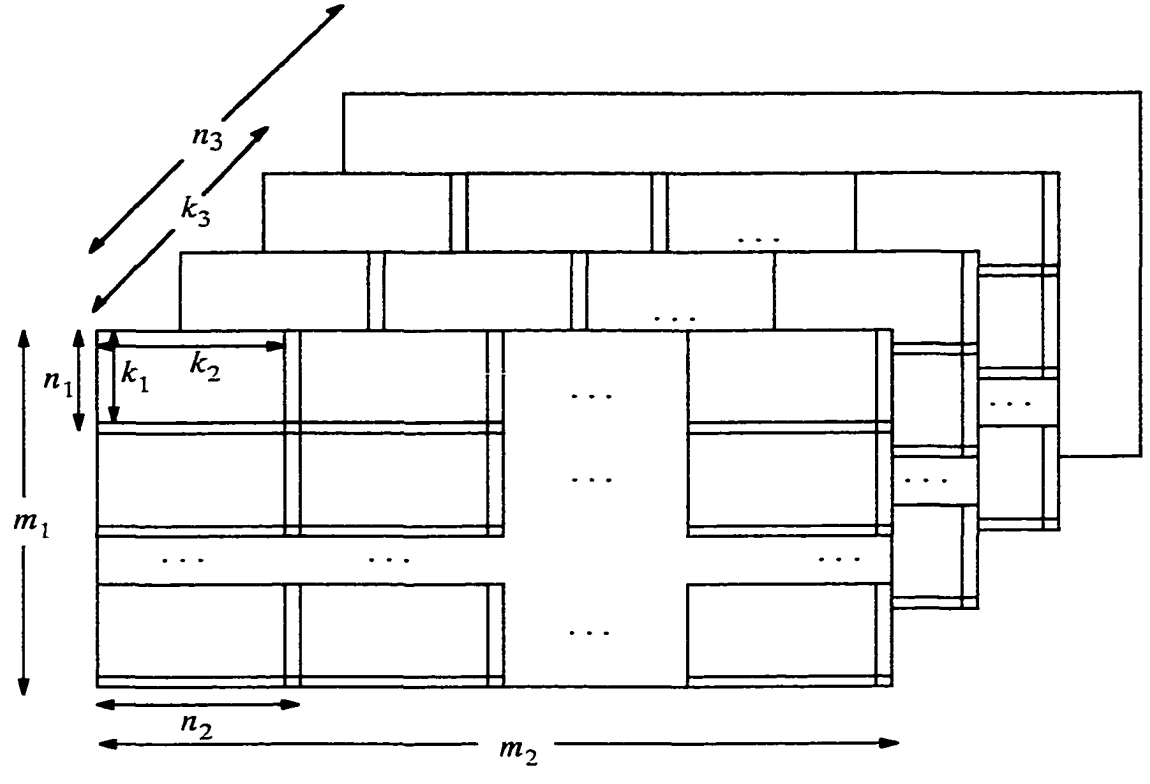


Figure 6.1: Multiblock 3–D row and column array code.

For the multiblock 3–D array code shown in Figure 4.3 and duplicated in Figure 6.1, the pages of data are represented as a $m_1 \times m_2$ array of $n_1 \times n_2 \times n_3$ 3–D code blocks. Equation 6.1 can be rewritten as

$$N_u = m_1 m_2 n_1 n_2 n_3 M_u L \tag{6.3}$$

and Equation 6.2 can be rewritten as

$$N_c = m_1 m_2 k_1 k_2 k_3 M_c L. \tag{6.4}$$

The page size is given by the product $m_1 m_2 n_1 n_2$. For $n_1 = n_2 = 8$ and $m_1 = m_2 = 64$, a page size equal to 512^2 bits is obtained. Similarly, when $n_1 = n_2 = 8$ and $m_1 = m_2 = 128$, a page size equal to 1024^2 bits is obtained. The simulation shown in Figure 6.2 is obtained using Equation 6.3 with $L = 100$ and page sizes of 512^2 and 1024^2 bits. A memory capacity exceeding 30 Gbits is attainable where 40,000 to 50,000 1Mbit pages are stored.

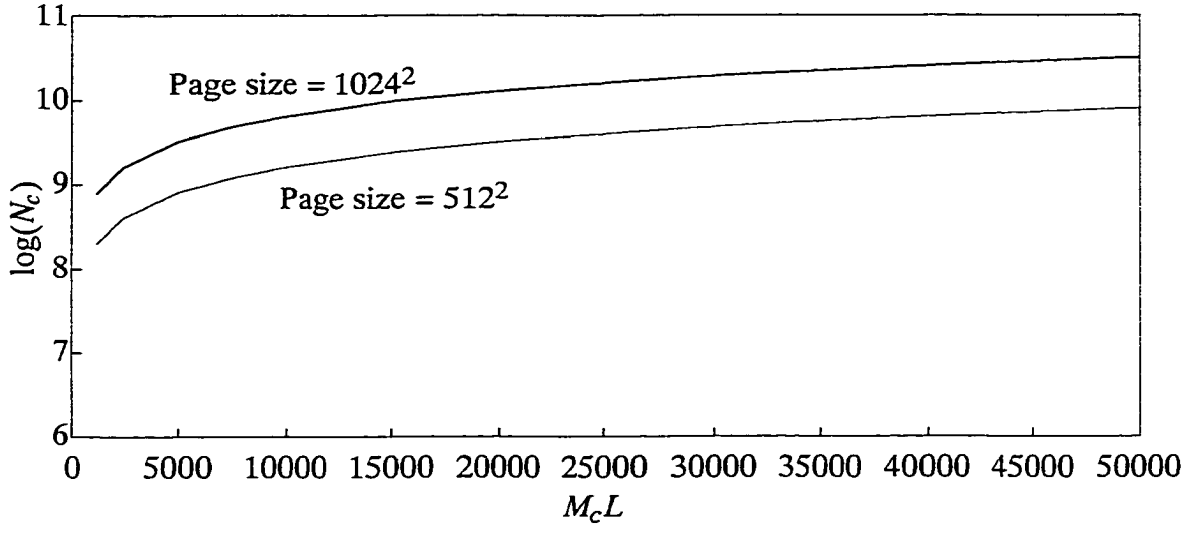


Figure 6.2: Storage capacity for different page sizes and number of pages.

The memory capacity N_u and N_c given by Equations 6.1 and 6.2 ignore constraints imposed by the *SNR* and loss in diffraction efficiency. Important issues regarding error correcting codes such as code rate, code gain and Hamming distance of the code impact memory capacity. These constrains and issues are now taken into account.

The loss in diffraction efficiency, η, goes as $1/M^2$ where M is the number of multiplexed pages or holograms [NEI94]. Loss in diffraction efficiency is one factor which limits the useful storage capacity of a holographic memory. For a given noise floor, cross talk effects also act to limit memory storage capacity [MOK93]. To establish an upper limit

on storage capacity assume the cross talk effect can be ignored. However, the value of η must be chosen to meet a desired *SNR* for a given optical readout power level and required *CBER*. Since the *SNR* associated with interpage cross talk effects decreases as $1/M$ for angular multiplexed holograms, this is a valid assumption for large M (e.g., $M > 200$). Error correcting codes can be used to increase *SNR* so a required *CBER* can be obtained. The improved storage capacity can be obtained by selecting an appropriate code rate to produce a corresponding coding gain value.

Let the *SNR* expression for uncoded and coded pages be given respectively by

$$SNR_u = \frac{\gamma}{M_u^2} \tag{6.5}$$

and

$$SNR_c = \frac{\gamma}{M_c^2} \tag{6.6}$$

where γ depends on physical and optical system quantities which are constant and independent of the encoding [NEI94]. After dividing Equation 6.4 by Equation 6.3 and using Equations 6.5 and 6.6, a ratio is obtained given by

$$\frac{N_c}{N_u} = r\sqrt{\frac{SNR_u}{SNR_c}} \tag{6.7}$$

where r is the code rate (i.e., $r = k/n$). As mentioned previously, coding gain, g, is a parameter of an error correcting code used to measure the error correcting capability of a code and can be defined as

$$g = 10\log_{10}\frac{SNR_u}{SNR_c}. \tag{6.8}$$

Using Equations 6.7 and 6.8, a storage capacity coded–to–uncoded–ratio, *CUR*, containing coding gain can be derived and is given by

$$CUR = \frac{N_c}{N_u} = r10^{\frac{g}{20}}. \tag{6.9}$$

Using Equation 6.9, simulations can be performed to determine an optimum code rate for an error correcting code. A plot of the simulation for code size $n = 512$ is shown in Figure 6.3. For a given code size, this simulation shows the *CUR* approaches an optimum code rate range. The specific component codes and code parameters used to produce a code of a given size are shown in Tables 6.1.

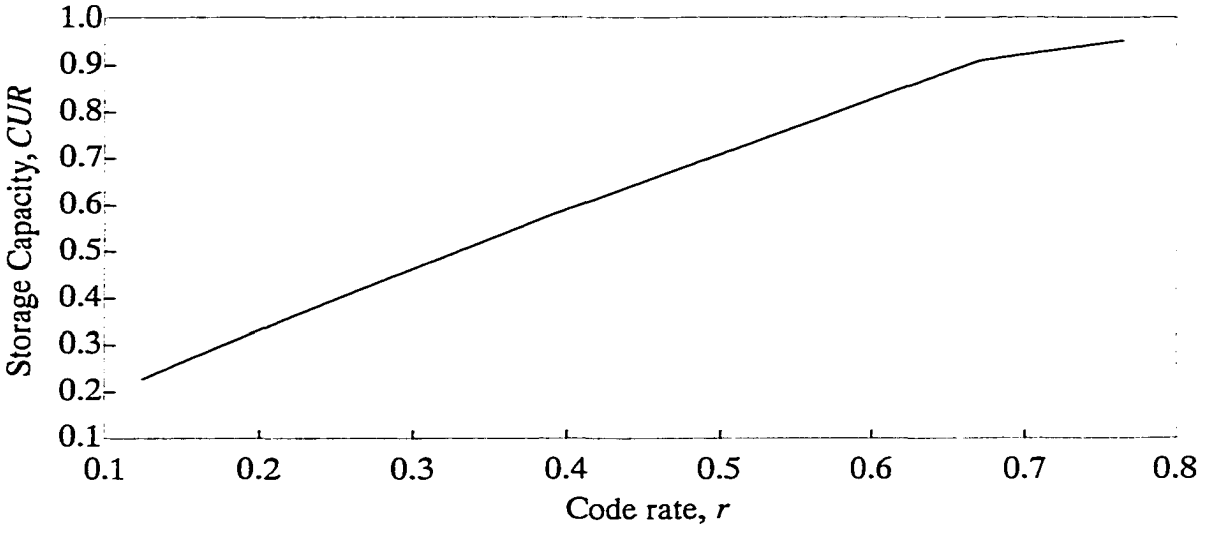


Figure 6.3: Storage capacity *CUR* versus code rate for $n = 512$ and $CBER \leq 10^{-12}$.

Table 6.1: Parameters for 3–D row and column array codes with $n = 512$.

n_1 code	n_2 code	n_3 code	n	k	d	c	r
8,8,1	8,7,2	8,7,2	512	392	4	1	0.7656
8,7,2	8,7,2	8,7,2	512	343	8	3	0.6699
8,7,2	8,7,2	8,4,4	512	196	16	7	0.3828
8,7,2	8,4,4	8,4,4	512	112	32	15	0.2188
8,4,4	8,4,4	8,4,4	512	64	64	31	0.125

6.2 Decoder Time Delay

The decoder time delay is inversely related to the transfer rate of the decoder. Therefore, to maximize the transfer rate of the decoder, the decoder time delay must be

minimized. To have a technology independent measure, the time delay is measured in gate delays. However, if a particular technology is selected, a specific gate delay can be used or specified. The time delays of decoders developed for multiblock array code range from 12 to 231 gate delays and are shown in Figure 6.4 for ease of comparison. The decoder for the random single bit error correction code exhibits the lowest time delay. However, it can only correct one error type. The decoder for the cluster error correcting code has the longest delay. In addition to correcting cluster errors, this decoder can correct burst and random single bit error that may occur in place of the cluster errors.

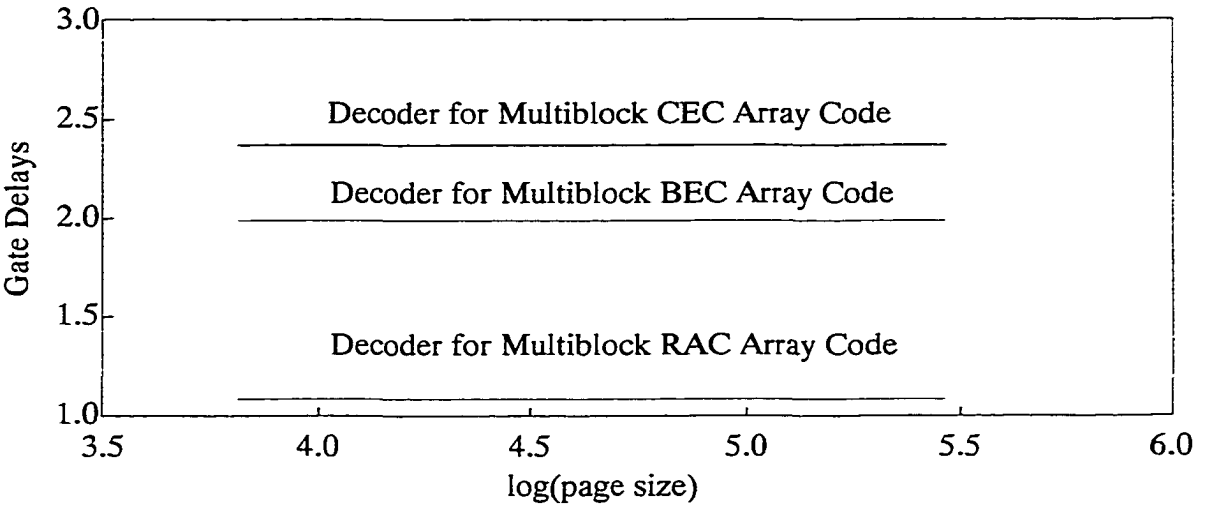


Figure 6.4: Decoding delay of different decoders for different page sizes.

6.3 Transfer Rate

In general, transfer rate for a VHDS system is design dependent. The design in turn is based on selected technologies which constrains the architecture and components selected for use in the decoder. The transfer rate for the read operation is given by

$$T = \frac{rC}{\tau_{decode}} \tag{6.10}$$

where r is the code rate, C is the storage capacity of a page (i.e., page size) and τ_{decode} is the decode time for the decoder. Figure 6.5 is a plot of Equation 6.10.

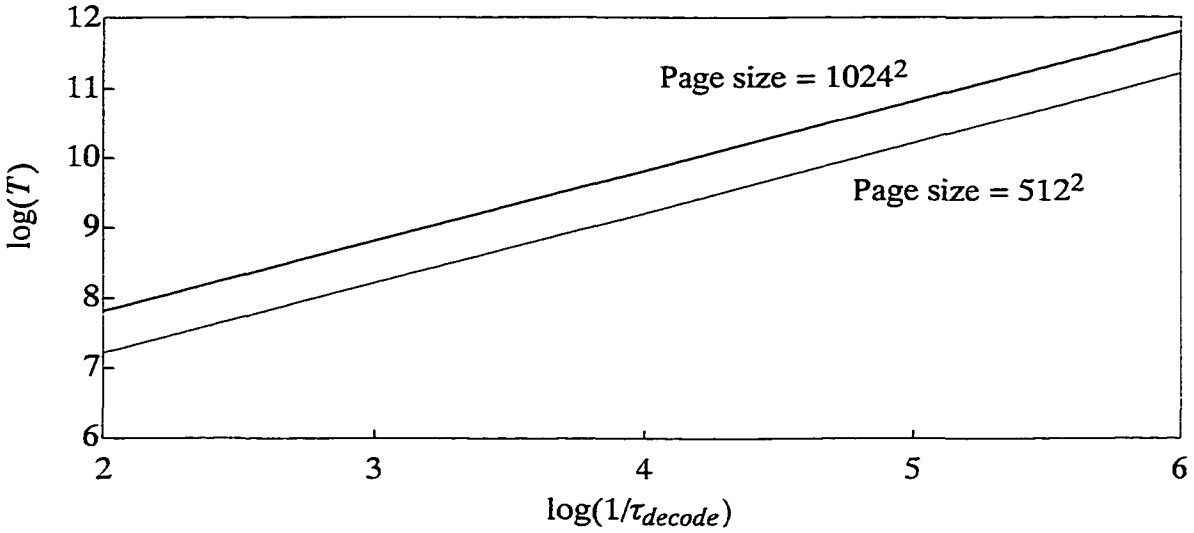


Figure 6.5: Transfer rate versus decode time.

6.4 Data Encoding Scheme

We know from sections 2.4.2 and 2.4.3 that the threshold detection process is susceptible to errors caused by noise sources such as optical scatter and detection noise. Modulation codes offer a way to reduce detection errors caused by spatial intensity variation [BUR97, VAD99]. These codes impose constraints on the number of on–pixels to off–pixels which can be used in a spatial region to represent a data symbol. For example, balanced modulation codes require the same number of on–pixels to off–pixels in representing a data symbol. The adverse effects of intensity variation can be reduced by the modulation code constraints such that the *CBER* is reduced. A disadvantage of using a modulation code is the loss in storage capacity caused by a decrease in code rate. For example, a 6:8 balance code (i.e., six user bits get encoded as eight balanced bits) has a code rate of 75% (i.e., a storage capacity loss of 25%).

As shown in Figure 6.6, array codes can provide a data encoding scheme to both modulate and encode information for storage in a volume holographic storage system. This

encoding converts each symbol of a character set into a column of 14 binary bits [GAR99]. In each column, only three bits are ON (high intensity). One of the first six bits being ON specifies four 4–symbol character groups out of six 4–symbol character groups and one of the next four bits being ON specifies which 4–symbol character group out of four 4–symbol character groups and one of the last four bits being ON specifies the symbol within the 4–symbol character group. The total number of characters that can be represented using this encoding is limited to 96 but it is adequate for representing character sets used in word processing and database applications. A sparse encoding is produced which provides a high reconstruction contrast ratio between ON and OFF bits [NEI93].

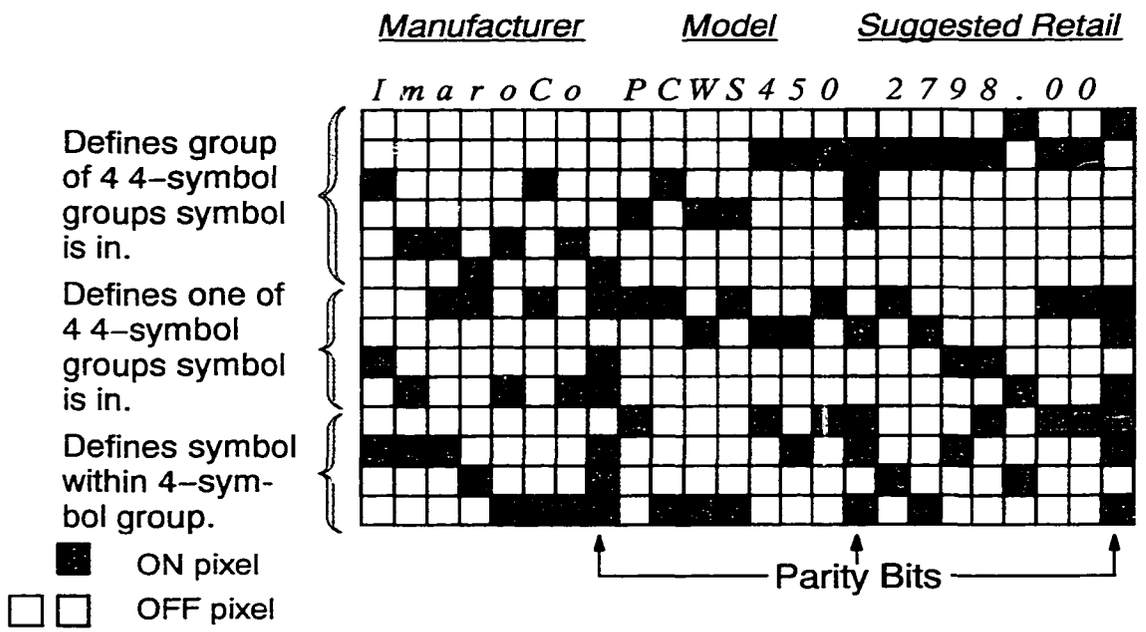


Figure 6.6: Array code applied to arrays of 7 symbols without column parity bits.

The 24 4–symbol groups are shown in Table 6.2. The selection of which symbols to include in 4–symbol groups and 4 4–symbol groups is somewhat arbitrary. However, to minimize the chance of creating horizontal lines across numerical symbol fields (e.g., checking and saving accounts numbers) and fields of upper and lower case alphabetical symbols, the symbol groups can be assigned in an essentially pseudorandom fashion.

Table 6.2: One arrangement of 4–symbol groups (not using pseudorandom number strategy) for encoding.

Symbol Group	*1*	*2*	*3*	*4*
4 4–Symbol Group 1	! " #	$ % & '	( ) * +	, – . /
4 4–Symbol Group 2	0 1 2 3	4 5 6 7	8 9 : ;	< = > ?
4 4–Symbol Group 3	@ A B C	D E F G	H I J K	L M N O
4 4–Symbol Group 4	P Q R S	T U V W	X Y Z [	\ ] ^ _
4 4–Symbol Group 5	` a b c	d e f g	h i j k	l m n o
4 4–Symbol Group 6	p q r s	t u v w	z y z {	| } ~ "

Table 6.3: Arrangement of 4–symbol groups (using pseudorandom number strategy) for encoding.

Symbol Group	*1*	*2*	*3*	*4*
4 4–Symbol Group 1	< c # s	S \ u 8	y : d N	R 1 & Y
4 4–Symbol Group 2	o 5 ` Q	T . ' U	h k a D	7 f * P
4 4–Symbol Group 3	A " E n	[l x $	~ w z 9	| v m)
4 4–Symbol Group 4	,] " q	{ e O j	C ! M b	(? 4
4 4–Symbol Group 5	; i V L	+ B g _	> 2 K X	6 @ / }
4 4–Symbol Group 6	H ^ t Z	I 3 % J	p G = r	W – 0 F

Using a pseudorandom number strategy the symbol groups shown in Table 6.3 can be obtained. The corresponding encoding is shown in Figure 6.7. If the ratio of horizontal and vertical lines containing two or more ON pixels to the number of ON pixels in the region of interest is used as a figure of merit, the pseudorandom number strategy may provide a small improvement. In Figure 6.6, out of the 84 ON pixels there are approximately 23 pixel lines, giving a pixel lines to ON pixels ratio (PLOR) of 27.38%. In Figure 6.7, out of the 82 ON pixels there are approximately 20 pixel lines giving a PLOR of 24.39%.

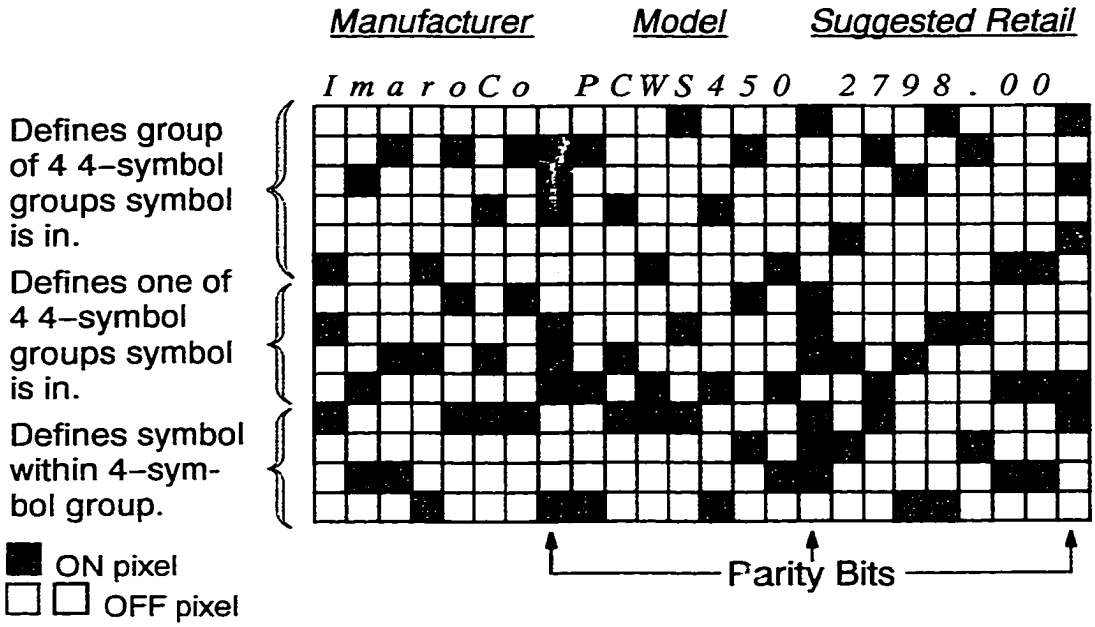


Figure 6.7: Array code applied to arrays of 7 symbols (using pseudorandom number strategy) without column parity bits.

Chapter 7
Conclusion and Future Work

We conclude this dissertation by briefly discussing goals for future work and summarizing the analyses and results previously detailed.

As storage intensive applications (e.g., databases, multimedia and network–based computing) continue to proliferate, greater demand for storage systems with large capacity, fast transfer rates, short access times and low bit error rate is expected. The volume holographic storage system is a mass storage system which can be configured to provide the required attributes of storage capacity ($\geq 10^{12}$ bits), transfer rate (10^9 bits/sec) access time ($\leq$ 40 μsec), $RBER \leq 10^{-4}$ and $CBER \leq 10^{-12}$. These promising attributes have led researchers and companies to investigate the performance or capabilities of VHDS systems for storage intensive applications.

7.1 Conclusion

The main goal of the research described in this dissertation has been to evaluate the potential of error correcting codes that span multiple pages for use in a VHDS system.

Several key interests motivated this research. First, a VHDS system operates on data in a 2–D format as pages or bit arrays and requires at least 2–D error detection and correction techniques. Second, the *RBER* for a typical VHDS system does not meet the industry standard for binary data storage and leads to a need to use error correcting codes. Third, error events can span multiple pages thereby generating a need to study EDC techniques to address these error events. EDC techniques using 3–D EEP and UEP array codes were shown to be effective on error events spanning multiple pages.

An introduction to some of the system issues regarding a typical VHDS system were provided. The system interaction during recording and reconstructing holograms were reviewed. Important system considerations were found to be cross talk, scattering and detector noise sources. These noise sources interact complicating the intensity detection process through the introduction of different error types. The specific error types mentioned were the random single bit, burst and cluster errors. These error types require effective error correcting codes be used for encoding and decoding data pages to achieve acceptable *CBER* performance in a VHDS system.

The attributes of block 2–D array codes for detecting and correcting the error types were described. The row and column array code used to detect and correct random single bit errors were shown to be effective in providing a $CBER \leq 10^{-12}$ for a $10^{-5} \leq RBER \leq 10^{-4}$. Although, the wing array code can correct random single bit errors, the code proved to be the least flexible of the array codes considered. For example, the wing code required the use of an odd parity strategy for data encoding and decoding. Additional codes are required to protect regions not protected by this code. The row and column array code was adapted to be a burst error detection and correction code. Compared to random single bit error correcting array codes, the burst error correcting array code required up to an order of magnitude more hardware and increase in the time delay for data encoding and

decoding. The cluster error correcting array code was found to be the most complex of the array codes considered. For an 8×8 array code capable of correcting a 2×2 cluster error, approximately 1950 gates are required to implement the decoder design considered. However, the time delay was approximately the same as for the burst error correcting array code. Using a decoding scheme which limits the number of on-state pixels per symbol, leads to a more simple decoding process and allows encoding and modulation to be combined.

The discussion of 3-D array codes focussed on the structure and attributes of these codes. A uniblock 3-D array code for detecting and correcting random single bit errors were shown to have a Hamming distance of 8. This distance value allows this code to detect four errors and correct three errors. For application with page sizes as large as 10^6 bits, a multiblock version of the row and column array code was described which used a collection of uniblock 3-D array codes arranged in a row and column format. The 3-D wing array code constructed to have a parity bit check layer was found to have a Hamming distance equal to five allowing this code to detect and correct two errors. The analysis showed a 3-D wing array code using two layers and no parity bit check layer was also capable of detecting and correcting two errors. These observations provided the motivation for using the more simple 3-D wing code structure for analysis. This idea of not using a parity bit check layer was utilized in developing the structure of the burst and cluster error correcting array codes. By not using a parity bit check layer, the hardware required and associated time delay are reduced for the burst and cluster error correcting 3-D array codes.

The nature and attributes of the UEP 3-D array code allows this code to provide particular groups of information bits different levels of protection. Since this code is constructed such that not all of the parity bit checks in the 3-D row and column are used, this code is a subset of the 3-D row and column array code. The UEP code in effect sacrifices structural regularity and corresponding error correcting capability to achieve the

characteristic of unequal error protection. This type of code can be utilized in application (e.g., numerical data, multiuser channels and computer networks) where data may have reliability requirements that vary within a block of data or require more protection against noise levels with different levels of importance. The EEP to UEP conversion algorithm presented in this work we believe is for the first time. This algorithm contributes to the flexibility of array codes.

The performance analysis point to several important features of a VHDS system. A VHDS system offers simultaneously large storage capacity, high transfer rate and fast access times. The storage capacity of the system is expected to be $\geq 10^{12}$ bits, a transfer rate $\geq 10^{9}$ bits/sec, and a read access time $\leq$ 40 μsec.

7.2 Future Work

The noise sources and related error mechanism require further study. The design of lower bit error rate VHDS systems is highly desirable. Therefore, a better understanding of potential noise sources and characterization of error statistics are required. We know from section 2.5 that several error types (e.g., random single bit errors, burst errors, and cluster errors) occur in a VHDS system. Since potential noise sources are implementation dependent, a database of system implementations, probable noise sources and corresponding error statistics will be helpful to VHDS system designers.

Data encoding schemes to both modulate and encode data for storage in a VHDS system are desirable. Discussions of these codes exist in the literature but the performance of these codes on the various error types needs more study. For example, which modulating and encoding scheme, provides better code performance on a specific error type?

More efficient decoding will likely involve more sophistication in manipulating the redundancy of the array code. For the burst error correcting array code, instead of computing

the parity bit checks over a diagonal direction set which has a slope, m, of either $m = 1$ or $m = -1$, a diagonal direction set with a higher slope value (e.g., either $m = 2$ or $m = -2$) can be investigated. Moreover, there seems to be an endless need for more sophisticated interleaving techniques regarding burst and cluster error correcting array codes.

REFERENCES

[AGR97] P. G. Agrawal, Fiber–Optic Communication Systems, John Wiley & Sons, Inc., New York, 1997.

[ASH96] J. Ashley, M. Blaum and B. Marcus, "Report on coding techniques for holographic storage," IBM Research Report RJ 10013 (89104), Mar. 25, 1996.

[BAS94] M. C. Bashaw, J. F. Heanue, A. Aharoni, J. G. Walkup and L. Hesselink, "Cross–talk considerations for angular and phase–encoded multiplexing in volume holography," *J. Opt. Soc. Am. B*, Vol. 11, No. 9, Sep. 1994, pp. 1820–1836.

[BER96] M. Bernal, H. Coufal, R.Grygier, J. Hoffnagle, C. M. Jefferson, R. Macfarlane, R. Shelby, G. Sincerbox, P. Wimmer and G. Wittmann, "A precision tester for studies of holographic optical storage materials and recording physics," *Applied Optics*, Vol. 35, No. 14, May 1996, pp. 2360–2370.

[BLA94] M. Blaum and P. G. Farrell, "Array codes for cluster–error correction," *Electronics Letters*, Vol. 30, No. 21, Oct. 1994, pp. 1752–1753.

[BLA96] M. Blaum, "Array codes for holographic storage," *Proceedings of the 1996 Workshop on Data Encoding for Page–oriented Optical Memories*, Phoenix, AZ, March 27–28, 1996, published by Colorado State University, College of Engineering, P. A. Mitkas Editor, pp. 59–62.

[BLA98] M. Blaum, J. Bruck and A. Vardy, "Interleaving schemes for multidimensional cluster errors," *IEEE Trans. on Info. Theory*, Vol. 44, No. 2, Mar. 1998, pp. 730–743.

[BUR95] G. W. Burr, F. H. Mok, and D. Psaltis, "Angle and space multiplexed holographic storage using the 90° geometry," *Opt. Comm.*, Vol. 117, May 1995, pp. 49–55.

[BUR97] G. W. Burr, J. Ashley, H. Coufal, R. Grygier, J. Hofnagle, C. M. Jefferson and B. Marcus, "Modulation coding for pixel–matched holographic data storage," *Optics Letters*, Vol. 22, No. 9, May 1997, pp. 639–641.

[BUR98] G. W. Burr, W. Chou, M. A. Neifeld, H. Coufal, J. A. Hoffnagle and C. M. Jefferson, "Experimental evaluation of user capacity in holographic data–storage systems," *Applied Optics*, Vol. 37, No. 23, 10 Aug. 1998, pp. 5431–5443.

[CAL61] P. Calingaert, "Two-dimensional parity checking," *J. of the ACM*, Vol. 8, No. 1, Jan. 1961, pp. 186–200.

[CAS96] K. R. Castleman, Digital Image Processing, Prentice–Hall, Inc., New Jersey, 1996.

[CHE97] P. P. Chen, M. Plesset and D. Theis, "Computer Storage Technology," in McGraw–Hill Encyclopedia of Science and Technology, Eighth Edition, 1997, pp. 298–308.

[CUR93] K. Curtis, C. Gu, D. Psaltis, "Cross talk in wavelength-multiplexed holographic memories," *Optics Letters*, Vol. 18, No. 12, Jun. 15, 1993, pp. 1001–1003.

[DAN85] J.Daniels and P. G. Farrell, "Burst–error–correcting array codes: further developments," International Conference on Digital Processing of Signals in Communications, Loughborough, UK, Publication No. 62, 22–26 Apr. 1985, pp. 261–4.

[DOM88] V. Dombrovskii, S. Dombrovskii and E. Pen, "Reliability of data readout in a holographic channel with constant characteristics," *Optoelectron. Instrum. Data Process*, Vol. 6, 1988, pp. 69–77.

[DUN88] L. Dunning and W. Robbins, "Optimal encodings of linear block codes for unequal error protection," *Information and Control*, Vol. 37, 1988, pp. 150–177.

[FAR79] P. G. Farrell, "Array Codes," in Algebraic Coding Theory, (ed. by G. Longo), Springer-Verlag, 1979, pp. 231–242.

[FAR82a] P. G. Farrell and S. J. Hopkins, "Burst-error-correcting array codes," *The Radio and Electron. Engineer*, Vol. 52, No. 4, Apr. 1982, pp. 188–192.

[FAR82b] P. G. Farrell, "Array codes for correcting cluster–error patterns," International Conference on Electronic Image processing, University of York, UK, 26–28 Jul. 1982, pp. 83–87.

[FAR90] P. G. Farrell, "Coding as a cure for communication calamities: the successes and failures of error control," *Electron. and Commun. Eng. J.*, Vol. 2, No. 6, Dec. 1990, pp. 213–220.

[FAR92] P. G. Farrell, "A survey of array error control codes," *European Trans. on Telecommun.*, Vol. 3, No. 5, Sep.-Oct. 1992, pp. 441–454.

[FAR96] P. G. Farrell, "Array codes: a tutorial summary," *Proceedings of the 1996 Workshop on Data Encoding for Page–oriented Optical Memories*, Phoenix, AZ, March 27–28, 1996, published by Colorado State University, College of Engineering, P. A. Mitkas Editor, pp. 29–34.

[FUR97] Y. Furukawa, K. Kitamura, Y. Ji, G. Montemezzani, M. Zgonik, C. Medrano and P. Gunter, "Photorefractive properties of iron–doped stoichiometric lithium niobate," *Optics Letters*, Vol. 22, No. 8, Apr. 1997, pp. 501–503.

[GAR98] T. N. Garrett and P. A. Mitkas, "Three–dimensional data encoding for volumetric optical memories," *SPIE Proceedings*, Vol. 3468, SPIE Annual Meeting, San Diego, CA, Jul. 1998, pp. 116–124.

[GAR99] T. N. Garrett and P. A. Mitkas, "Equal and unequal error correcting codes for volume holographic storage systems," *SPIE Proceedings*, Vol. 3802, SPIE Annual Meeting, Denver, CO, Jul. 1999.

[GOE95] B. J. Goertzen, "Volume holographic storage for large relational databases," Masters thesis, Colorado State University, 1995.

[GOE96] B. J. Goertzen and P. A. Mitkas, "Volume holographic storage for large relational databases," *Optical Egineering*, Vol. 35, No. 7, Jul. 1996, pp. 1847–1853.

[GOO68] J. W. Goodman, Introduction to Fourier Optics, McGraw–Hill, Inc., New York, 1968.

[GOO85] J. W. Goodman, Statistical Optics, John Wiley & Sons, New York, 1985.

[GOO96] J. W. Goodman, Introduction to Fourier Optics, The McGraw–Hill Companies, Inc., New York, 1996.

[GU92] C. Gu, J. Hong, I. McMichael, R. Saxena, and F. Mok, "Cross-talk-limited storage capacity of volume holographic memory," *J. Opt. Soc. Am. A*, Vol. 9, No. 11, Nov. 1992, pp. 1978–1983.

[GU96] C. Gu, G. Sornat and J. Hong, "Bit–error rate and statistics of complex amplitude noise in holographic data storage," *Optics Letters*, Vol. 21, No. 14, Jul. 1996, pp. 1070–1072.

[HEA94] J. F. Heanue, M. C. Bashaw, and L. Hesselink, "Volume holographic storage and retrieval of digital data," *Science*, Vol. 265, No. 5173, Aug. 1994, pp. 749–752.

[HEA95] J. F. Heanue, M. C. Bashaw, and L. Hesselink, "Channel codes for digital holographic data storage," *J. Opt. Soc. Am. A*, Vol. 12, No. 11, Nov. 1995, pp. 2432–2439.

[HEC98] E. Hecht, Optics, Addison–Wesley, Reading, Massachusetts, 1998.

[HOO94] P. W. Hooijmans, Coherent Optical System Design, John Wiley & Sons, Inc., New York, 1994.

[HUT96] J. F. Hutton, M. Schaffer, G. A. Betzos and P. A. Mitkas,"Error–correcting codes for page oriented memories," *SPIE Proceedings*, Vol. 2848, SPIE Annual Meeting, Denver, CO Aug. 1996.

[IBM96] IBM Holographic Optical Storage Team, "Holographic storage promises high data density," *Laser Focus World*, Vol. 32, No. 11, Nov. 1996, pp. 81–93.

[JAC85] J. D. Jackson, Classical Electrodynamics, John Wiley & Sons, New York, 1975.

[JAI89] A. K. Jain, Fundamentals of Digital Image Processing, Prentice–Hall, Inc., New Jersey, 1989.

[KIN99] B. M. King and M. A. Neifeld, "Unequal a–priori probabilities holographic storage," *SPIE Proceedings*, Vol. 3802, SPIE Annual Meeting, Denver, CO, Jul. 1999.

[LEE88] H. Lee, "Cross-talk effects in multiplexed volume holograms," *Optics Letters*, Vol. 13, No. 10, Oct. 1988, pp. 874–876.

[LI96] H. S. Li and J. Hong, "Nonuniformity in hologram diffraction efficiency from time–constant error in the recording schedule," *J. Opt. Soc. Am. B*, Vol. 13, No. 5, May 1996, pp. 894–899.

[MAB91] A. O. Mabogunje and P. G. Farrell, "Burst Error Correction Capability of Square Array Codes," *Electronics Letters*, Vol. 27, No. 13, Jun. 1991, pp. 1215–1216.

[MAN91] E. S. Maniloff and K. M. Johnson, "Maximized photorefractive holographic storage," *J. Appl. Phys.*, Vol. 70, No. 9, Nov. 1, 1991, pp. 4702–4707.

[MAN97] M. Mansuripur and G. Sincerbox, "Principles and techniques of optical data storage," *Proceedings of the IEEE*, Vol. 85, No. 11, Nov. 1997, pp. 1780–1796.

[MAR60] J. Marcum, "Studies of target detection by pulsed radar," Special Monograph Issue, *IRE Trans. on Info. Theory*, Vol. IT–6, No. 2, Apr. 1960.

[MCA91] A. D. McAulay, Optical Computer Architectures, John Wiley & Sons, Inc., New York, 1991.

[MCL98] S. W. McLaughlin, "Shedding light on the future of SP for optical recording," *IEEE Signal Processing Magazine*, Jul. 1998, pp. 83–94.

[MCM96] I. McMichael, W. Christian, D. Pletcher, T. Chang and J. Hong, "Compact holographic storage demonstrator with rapid access," *Applied Optics*, Vol. 35, No. 14, May 1996, pp. 2375–2379.

[MOK91] F. H. Mok, M. C. Tackitt, and H. M. Stoll, "Storage of 500 high-resolution holograms in a $LiNbO_3$ crystal," *Optics Letters*, Vol. 16, No. 8, Apr. 1991, pp. 605–607.

[MOK92] F. Mok, D. Psaltis, and G. Burr, "Spatially- and angle-multiplexed holographic random access memory," in Photonics for Processors, Neural Networks, and Memories, (ed. by W. J. Miceli, J. A. Neff, and S. T. Kowel), *Proc. Soc. Photo-Opt. Instrum. Eng.*, Vol. 1773, 1992, pp. 334–345.

[MOK93] F. H. Mok, "Angle-multiplexed storage of 5000 holograms in lithium niobate," *Optics Letters*, Vol. 18, No. 11, Jun. 1, 1993, pp. 915–917.

[MOR95] R. Morelos–Zaragoza and S. Lin, "QPS block–modulation codes for unequal error protection," *IEEE Trans. on Info. Theory*, Vol. 41, No. 2, Mar. 1995, pp. 576–581.

[NAT84] National Semiconductor Corporation, Logic Databook, Vol. 1, 1984.

[NEI93] M. A. Neifeld, "Computer generated holography for optical memory using sparse data words: capacity and error tolerance," *Applied Optics*, Vol. 32, No. 26, Sep. 1993, pp. 5125–5134.

[NEI94] M. A. Neifeld and M. McDonald, "Error correction for increasing the usable capacity of photorefractive memories," *Optics Letters*, Vol. 19, No. 18, Sep. 1994, pp. 1483–1485.

[NOR93] G. P. Nordin and P. Asthana, "Effects of cross talk on fidelity in page-oriented volume holographic optical data storage," *Optics Letters*, Vol. 18, No. 18, Sep. 1993, pp. 1553–1555.

[NOR94] G. P. Nordin and P. Asthana, "Achieving a minimum signal-to-noise ratio in angularly multiplexed volume holographic optical data storage systems," in Photonics for Processors, Neural Networks, and Memories II, (ed. by J. L. Horner, B. Javidi, and S. T. Kowel), Proc. Soc. Photo-Opt. Instrum. Eng., Vol. 2297, 1994, pp. 392–401.

[PAR90] T. Parish, "Crystal clear storage," *Byte*, Vol. 15, No. 11, Nov. 1990, pp. 283–288.

[PIN88] F. Pingzhi, C. Zhi and J. Fan, "Linear unequal error–protection array codes," *Electronics Letters*, Vol. 24, No. 6, 1988, pp. 333–335

[PSA92] D. Psaltis, "Parallel optical memories," *Byte*, Vol. 17, No. 9, Sep. 1992, pp. 179–182.

[PSA95] D. Psaltis, "Holographic memories," *Scientific American*, Vol. 273, No. 5, Nov. 1995, pp. 70–76.

[PSA96] D. Psaltis and A. Pu, "Performance characteristics of holographic 3–D disks," *Proceedings of the 1996 Workshop on Data Encoding for Page–oriented Optical Memories*, Phoenix, AZ, March 27–28, 1996, published by Colorado State University, College of Engineering, P. A. Mitkas Editor, pp. 37–40.

[PSA98] D. Psaltis and G. Burr, "Holographic data storage," *Computer*, Vol. 31, No. 2, Feb. 1998, pp. 52–60.

[RAO89] T. R. N. Rao and E. Fugiwara, Error-Control Coding for Computer Systems, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[RAS93] K. Rastani, "Storage capacity and cross talk in angularly multiplexed holograms: two case studies," *Applied Optics*, Vol. 32, No. 20, Jul. 10, 1993, pp. 3772–3777.

[RED89] S. Redfield and L. Hesselink, "Data storage in photorefractives revisited," in Optical Computing, *Proc. Soc. Photo-Opt. Instrum. Eng.*, Vol. 963, 1989, pp. 35–45.

[RED92] S. Redfield, "Holographic storage: not a device but a storage class," in Enabling Technologies for High-Bandwidth Applications, *Proc. Soc. Photo-Opt. Instrum. Eng.*, Vol. 1785, 1992, pp. 45–51.

[ROW68] R. Rowland, "Error-detecting capabilities of two-coordinate parity codes," *Electron. Eng.*, Vol. 40, No. 479, Jan. 1968, pp. 16–20.

[SAL91] B. E. A. Saleh and M. C. Teich, Fundamentals of Photonics, John Wiley & Sons, Inc., New York, 1991.

[SCH66] M. Schwartz, W. Bennel and S. Stein, Communications Systems and Techniques, McGraw–Hill, Inc., New York, 1966.

[SCH98] M. E. Schaffer, "Smart photodetector arrays for error control in page–oriented optical memory," Ph.D. Dissertation, Colorado State University, 1998.

[SHA88] K. S. Shanmugan and A. M. Breipohl, Random Signals: Detection, Estimation and Data Analysis, John Wiley & Sons, Inc., New York, 1988.

[TAO99] S. Tao, B. Tang, Y. Zhou and L. Shen, "Quantitative study of the gray–scale fidelity of volume holographic images," *Applied Optics*, Vol. 38, No. 17, Jun. 10, 1999, pp. 3767–3777.

[VAD99] V. Vadde and B. V. K. Kumar, "Parity coding for page–oriented optical memories with intrapage intensity variations," *Optics Letters*, Vol. 24, No. 8, Apr. 15, 1999, pp. 546–548.

[WUL94] J. R. Wullert II and Y. Lu, "Limits of the capacity and density of holographic storage," *Applied Optics*, Vol. 33, No. 11, Apr. 1994, pp. 2192–2196.

[YEH93] P. Yeh, Introduction to photorefractive nonlinear optics, John Wiley & Sons, Inc., New York, 1993.

[YI95] X. Yi, S. Campbell and P. Yeh, "Statistical analysis of cross–talk noise and storage capacity in volume holographic memory: image plane holograms," *Optics Letters*, Vol. 20, No. 7, Apr. 1995, pp. 779–781.