AFRL-IF-RS-TR-2002-251
Final Technical Report
September 2002

LOW LEVEL SIGNAL PROCESSING FOR THE OPTOELECTRONIC MEMORY SYSTEM INTERFACE

University of Pittsburgh

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

APPROVED: [illegible]
BERNARD J. CLARKE
Project Engineer

FOR THE DIRECTOR: [illegible]
JOSEPH CAMERA, Chief
Information & Intelligence Exploitation Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** SEPTEMBER 2002	**3. REPORT TYPE AND DATES COVERED** Final Aug 99 – Dec 01

4. TITLE AND SUBTITLE LOW LEVEL SIGNAL PROCESSING FOR THE OPTOELECTRONIC MEMORY SYSTEM INTERFACE	**5. FUNDING NUMBERS** C - F30602-99-1-0550 PE - 62702F PR - PMEM TA - 00 WU - 01
6. AUTHOR(S) Donald M. Chiarulli and Steven P. Levitan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The University of Pittsburgh Department of Computer Science 212 MIB Pittsburgh Pennsylvania 15260	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFEC 32 Brooks Road Rome New York 13441-4114	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2002-251

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Bernard J. Clarke/IFEC/(315) 330-2106/ Bernard.Clarke@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

This report describes the development and implementation of channel coding and error detection firmware and software for a prototype optical memory system. Specific accomplishments include the development and implementation of a novel 2D coding system for managing inter-symbol interference in page oriented memory as well as clock extraction, noise filters, 5-7 coder firmware for the memory prototype. Software and firmware implementation of spectral Reed-Solomon error correcting codes were also developed and incorporated into a demonstration system. Additionally, we have identified transaction-level processing primitives that might be implemented at the storage interface to improve utilization of the high transfer rate data for image storage applications.

14. SUBJECT TERMS Optical Memory, Page Oriented Memory, Channel Codes, Error Correcting Codes, Application Oriented Storage Systems	**15. NUMBER OF PAGES** 24
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

List of Figures

1 INTRODUCTION

One of the major trends in recent years has been an explosion in the volume of data being collected from sources such as ground, air, and space based sensors. Currently, there is more data being generated than can be analyzed in real time. The DoD Image Understanding program cites a metric known as the *pixel-to-pupil* ratio to express the widening gap between the amount of data being gathered and the number of analysts available to examine it. Technological solutions are being sought that provide for greater levels of computer assisted data analysis, wider access through high bandwidth networks, and high volume data storage facilities.

Optical memory systems have the potential to provide some of the necessary data storage capacity, but significant technical issues exist with regard to the access, retrieval and distribution of this data. For example, individual sensor types within a composite image may generate data in incompatible formats. These formats need to be converted, the data registered over multiple images, and fused into a single view. It is often necessary to extract features, make comparisons, and highlight variations between current sensor data and archival copies of an image, data from other sources, or even a computer model. These are compute intensive applications, which require that very high volumes of data be delivered to the processing site. With the use of a page oriented optical memory system, such high volume transfer rates can be achieved.

1.1 SPECIFIC ACCOMPLISHMENTS

The goal of this research was to facilitate an end-to-end demonstration of data transfers from a page oriented optical memory to a program executing on a desktop personal computer. In a previous contract, F30602-96-C-0206, we designed and implemented an optoelectronic cache memory and interface to a page oriented optical memory system. Those results demonstrated that an optical memory can be tightly integrated into a conventional computer and that its high spatial bandwidth can be successfully exploited when optical memories are used as a backing store[1]. In this project, we have implemented low-level signal processing primitives to improve the density and reliability of this storage system. We are reporting on three specific accomplishments. First, we have developed a novel 2D coding system for managing inter-symbol interference in page-oriented memory. Second, in direct support of the prototype implementation we have implemented software and firmware implementations of spectral Reed-Solomon error correcting codes as well as clock extraction, noise filters, and 5-7 coder firmware. Third, we have also investigated "transaction oriented" processing primitives that might be implemented at the storage interface to improve utilization of the high transfer rate data. Each of these activities is summarized below, followed by a detailed technical overview in the next section.

1.1.1 Novel 2D Dynamic Channel Codes

Data reliability is an essential requirement for any memory system. There are a few factors for optical memories that may corrupt the data while being recorded, retained or read out from the optical memory. For example, the readout system and the material noise, optical effects such as reflection, diffusion, and inter-pixel crosstalk due to additive noise in gaussian amplitude. Mechanical noise sources are also present, such as jitter due to positioning the readout mechanism or the optical media.

The data reliability is typically ensured by using error correcting codes and media-specific data modulation. This part of the project was focused on the modulation schemes. Error correcting codes have a common characteristic that reliability is achieved at the cost of code density. For example, magnetic storage devices use encoding schemes such as EFM (eight to fourteen) modulation, which results in low utilization, 43% of the media is lost to overhead. There are several data encoding schemes proposed for page-oriented memories, such as array codes[2] and crosstalk minimization[3,4]. However, just like their counterparts from magnetic storage they also suffer from very low utilization of the media.

There are a variety of factors that can limit the set of allowable code words that are useable on an optical memory block. In this report, we will primarily consider inter-symbol interference (ISI) and the noise margins required to represent an individual bit. For example, code words must maintain a specific topological separation of "1" bits so that ISI does not raise the intensity of neighboring “0”’s above a pre-set threshold. Typically this is accomplished by a static encoding that uses a pre-selected set of code words based on these properties of the storage media and the optical system. Alternatively, our approach provides for a dynamic analysis of all data currently stored in the region surrounding a particular block and defines the allowable code words uniquely for each block.

We assume the existence of a “smart” read head that is capable of analyzing a page of data and calculating the allowable codes in real-time based on the actual data in the surrounding region. We use point-spread function based mathematical model for optical readout system to evaluate and carry out data encoding. Our experiments show 81% spatial utilization while recent publications present only 45% utilization.

1.1.2 Software and Firmware Reed Solomon ECC Code

The error correcting code (ECC) that we have chosen to implement in the prototype firmware is the Spectral Reed-Solomon (SRS) code. Several characteristics of this code make it well suited to optical memory applications. Specifically, it has a high tolerance for burst errors such as those which might be introduced by defects in a material and it is scalable, either by increasing the size of the code word for better error correction performance or by processing multiple code words in parallel for a higher aggregate data rate. Although the decoder has significant computational requirements, it can be implemented using pipelined processing techniques to achieve high throughput.

Our research in this area has focused on the theory, implementation and performance of a spectral Reed-Solomon decoder for use in the optical memory prototype. We will present a general discussion of Reed-Solomon codes and the finite field algebra on which they are based in the next section along with details of the implementation and performance data from simulations based on timing extracted from the FPGA designs.

1.1.3 Transaction Level Codes for Application Oriented Storage

High data transfer rates for specific applications can be handled by moving computational power to the interface and implementing a transaction oriented storage system. Feasibility of such system was another part of our research. We developed transaction level firmware for ray-tracing application. The firmware is executed on a set of field programmable gate arrays, which can be reconfigured according to the computation demands. The main goal for such approach is to provide high performance demanded by certain applications, such as real time ray tracing.

In ray tracing application a model of a virtual world is defined as a set of objects. The code, which we have developed, together with sufficient hardware allows rendering the objects in real time by generating ray-traced image frame. Providing sufficient number of frames can be computed in each second, it is possible to render the world as if it is filmed with a video camera.

2 TECHNICAL OVERVIEW

2.1 NOVEL 2D DYNAMIC CHANNEL CODES

Data reliability is essential requirement for any memory system. There are a few factors for optical memories that may corrupt the data while being recorded, retained or read out from the optical memory. For example, the readout system and the material noise, optical effects such as reflection, diffusion, and inter-pixel crosstalk due to additive noise in gaussian amplitude. Mechanical noise sources are also present, such as jitter due to positioning the readout mechanism or the optical media.

The data reliability is typically ensured by using error correcting codes and media-specific data modulation. Our research is focused on the modulation schemes. Error correcting codes have a common characteristic that reliability is achieved at the cost of code density. For example, magnetic storage devices use encoding schemes such as EFM (eight to fourteen) modulation, which results in low utilization, 43% of the media is lost to overhead. There are several data encoding schemes proposed for page-oriented memories, such as array codes[5] and crosstalk minimization[3,4]. However, just like their counterparts from magnetic storage they also suffer from very low utilization of the media.

One reason for utilization being so low in these codes is that code words are assigned statistically taking in account the worst-case scenario. In our approach we assign codes dynamically by taking advantage of the fact that the whole memory page is available to us for encoding and decoding. Specifically we are able to consider the actual data on regions of each page and compute the codes dynamically. We assign codes that maximize utilization while satisfying the system constraints. Therefore we assume intelligence in the read/write head with ability to process information at the source in real time. The result will be higher code densities and therefore higher media utilization while retaining the same reliability level of the data. We can also expect better performance due to parallel information processing.

2.1.1 Problem Statement

A typical setup for a two-photon optical memory[5,6] is shown in Figure 1. The data is read from memory in pages. The page addressing laser beam is shaped to a sheet of light and targeted to the memory page we want to read. The storage media is transparent to the laser light. The 3D pixels (voxels) that represent value ‘1’ are exited by the laser light and fluoresce. The optical readout system projects the image to the detector array and decoding system. Each pixel in the 2D page has a light intensity. The system uses an intensity threshold for each pixel value.

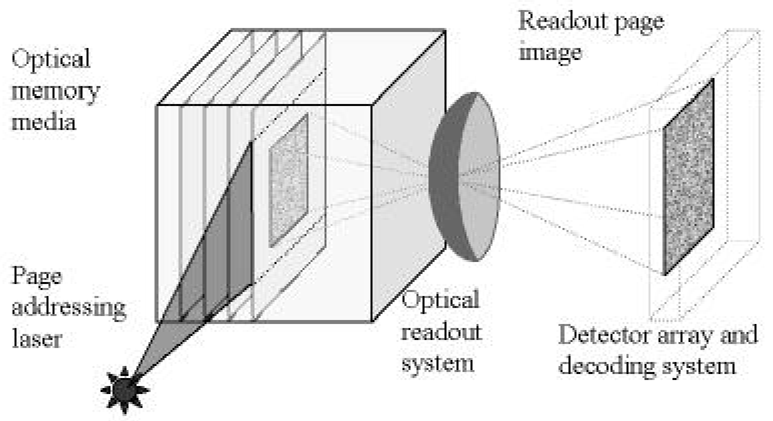


Figure 1: Two-photon optical memory

Several effects can result in corrupted information during readout:

- *Inter Symbol Interference.* The light output for each pixel almost always contributes some optical power to neighboring pixels on the detector array. This type of crosstalk should be minimized by the optical system, however a cluster of "1" pixels near a "0" pixel can produce enough crosstalk power such the "0" is detected in error as above threshold.

- *Weak ones*. If a '1' pixel is surrounded by many '0' pixels, and the readout threshold is fairly high, the '1' pixel may be too week to read out as "high" bit.

- *Misalignment and jitter*. The readout head, laser or the media may not be aligned perfectly due to mechanical nature of addressing a page. Therefore the image may be slightly shifted and pixels may lack some intensity or contribute some intensity to their neighbors.

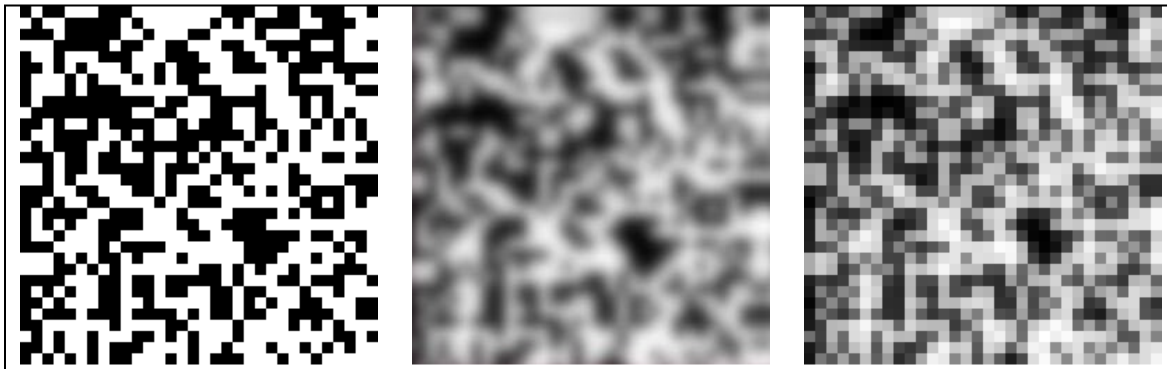

a) Binary data b) Detector image c) Sampled image

Figure 2: Raw data images

We will collectively refer to the combined effect of each of the three mechanisms as noise in the optical system. Figure 2 illustrates this with three versions of 32x32 bit image: a) the binary data written to memory; b) image as the detector array sees it and c) image as the detector array samples it. No matter what threshold you apply there is likely to be significant number of noise induced errors to the image between Figure 2b and Figure 2c.

In order to correctly detect ones and zeroes from the sampled image the relative intensity of each pixel must be clearly distinguishable according to value the pixel holds. To measure this relative intensity we gather statistics for each of the pixels on a page and plot a histogram of the number of pixels at each intensity. An example of one such histogram is shown in Figure 3. The two regions in this picture represent histograms for zeroes and ones. We observe that the regions overlap between intensities A and B. Thus in this region the same intensities may be interpreted as either '0' or '1', which results in unreliable data.

The goal of this research is to attain a histogram similar to Figure 4. The encoding should manipulate data so that the intensity values for '0' and '1' are sufficiently below or above our detection threshold T. This is achieved by ensuring that there is sufficient spatial separation between ones and zeroes on a 2D code block to limit the noise effects.

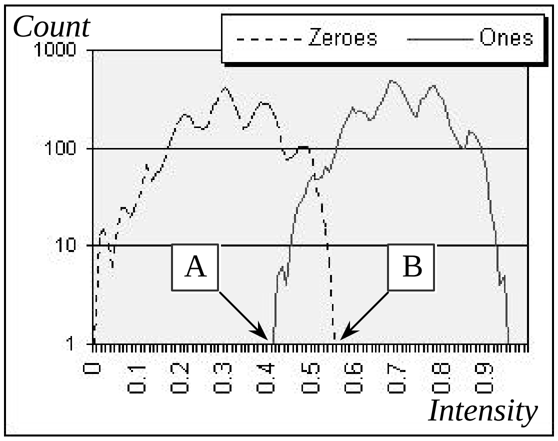


Figure 3: Raw data graph

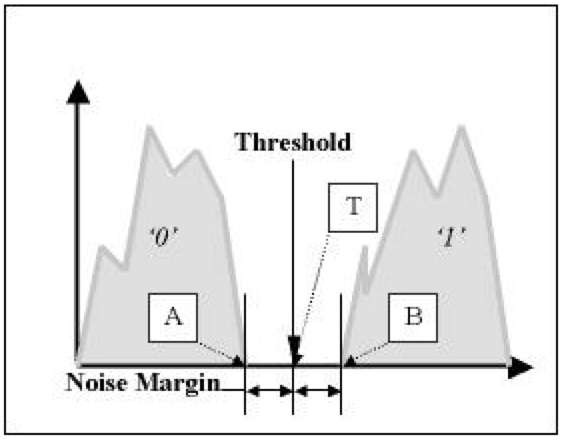


Figure 4: Noise margins

2.1.2 *Approach*

In this section we describe our mechanism for generating dynamic codes that minimize inter symbol interference (ISI) and other system noise. Specifically, ISI can cause a '0' pixel (that has many neighboring '1' pixels) to become "too bright". Similarly if zeroes surround a '1' pixel, it may become too weak to be correctly read as '1'. Therefore it will be a good idea to cluster '1' pixels by placing them in close proximity.

Since we cannot totally prevent system noise, we introduce "noise margin" as a parameter for our encoding. In this context the noise margin is a relative intensity above and below our detection threshold. Our encoding will provide that no pixels read between intensities A and B in the histogram of Figure 4. All "ones" will be encoded in such a way that their intensity should fall only above B, and "zeroes" only below A. The size of the noise margin is determined by the quality of the optics and memory system. Thus in realistic optical memory, if there will be additional intensity noise in the readout system, our encoding will tolerate it without loss of data correctness as long as it falls in the noise margin (A to T for zeroes and T to B for ones).

Figure 5 shows a section of data page during the encoding operation. The page is partitioned in rectangular code blocks (CB). In this example the light gray CB is being encoded. The dark gray CBs have been encoded previously, and the white blocks are assumed to be all zeroes (i.e. no light sources) since they will be subsequently encoded.

11001 00101 01010	01011 11010 11011	10111 10011 11010
10101 10001 11000	????? ????? ?????	00000 00000 00000
00000 00000 00000	00000 00000 00000	00000 00000 00000

Figure 5: Encoding scheme

We start with a blank page. The first CB to be encoded is the left-top most. There are no CBs to its left or above since it is beyond page boundary, therefore we assume zeroes there. We subsequently encode the whole line block by block and page line by line. Based on the surrounding data we compute intensity values and then valid codes for the CB to be encoded that will minimize cross talk and be within set noise margins. Once the valid code set is enumerated we select the k-th code from the code set depending on source data.

Decoding is done very similarly. We read digital data; compute intensities and valid code sets for each CB. The difference is that now we can use the code sets for reverse lookup of the encoded data from the read data. One advantage of the decoding is that it can be done in parallel for all CBs. The algorithm suggests reconfigurable mesh-connected parallel architectures, such as our optically reconfigurable field programmable gate array, developed in our laboratory[7]. Also, left-to-right approach for page fill is not necessarily the best. Filling page in a random or uniform order may actually yield better results.

There are several ways that we can generate valid codes for CBs. The most effective from the code density point of view is to enumerate all possible codes that do not violate noise margins. Unfortunately the worst time complexity for full enumeration approach is order of 2^{N-1} where N is the number of bits in a CB. The central focus of this research is to devise alternative algorithms that exploit topological properties and heuristics that are both efficient and well suited to parallel processing architectures.

2.1.3 Memory Model

We evaluate our encoding performance on two-photon volumetric optical memory model, described above. The evaluation of the encoding for other types of volumetric memories such as holography[8,9] and spectral hole burning is possible with adjustment of the model parameters and formulas to the chosen technology.

The optical memory storage is organized in 2D pages. The size and number of pages is a parameter of the model and we have run experiments for 64x64, 128x128 and 256x256 sized pages. To model the readout of the memory page the model computes an image that represents light intensities received by the readout detector array. Each of the light source intensities is computed to model light distribution assuming square aperture using the continuous point-spread function (PSF):

$$H(x, y) = 1/s^2 \operatorname{sinc}^2(x/s, y/s)$$

Where s is a parameter that models the degree of blur in the system, and

$$\operatorname{sinc}(x, y) = \sin(px)/px * \sin(py)/py$$

The optical memory page image is then generated using 2D convolution operator ? on binary data stored in the page and the PSF function.

$$r[i, j] = a[i, j] ? h[i, j] + noise[i, j]$$

where r[i, j] is the resulting intensity, a[i, j] is the source binary data, noise[i, j] is additive white Gaussian noise and h[i, j] is the effective discrete PSF computed as follows:

$$h[i, j] = ?? h(x, y)\, dxdy \qquad \textit{for each square } [i, j] \textit{ of PSF matrix.}$$

The surface integral is approximated with discrete over-sampling method using 64x64 samples per pixel. The binary output is determined by simple threshold on intensities. The degree of blur represents the quality of the optical system and is set at 1.4.

For optimization purposes we pre-compute discrete PSF matrix. We limited the dimensions of the matrix to 11x11 with the light source at the center. For every pixel beyond the dimensions of the matrix the additive intensity is assumed to be null, i.e. insignificant. This improved performance of the simulation system without significant loss of precision.

We ran experiments using compressed zip file as a data source. It was chosen for the random distribution of 0 and 1 bits due to data compression. Thus we may encounter wide variety of bit patterns (or else the file could be further compressed).

2.1.4 Simulation Results

In this section we first discuss the algorithm and then our experimental results. Our intention is to measure the improvement in utilization between conventional static encoding and our proposed dynamic encoding systems. We compute light intensity values and then build a valid code set for CB, which minimizes cross talk and complies with predefined noise margins.

The straightforward approach is to try all possible patterns of CB and *enumerate* only the valid ones. Then a bit-string from the source can be encoded using the enumeration as a lookup table. This takes about 2^N computations per CB where N is the number of pixels in a CB. One could pre-compute the lookup table to speed up the process, however the table is huge even for reasonable N values. Even though this approach is computationally very intensive, it gives us empirical upper bound for maximum utilization we could expect with the dynamic encoding approach.

An alternative way to generate a valid code is to evaluate CB *bit by bit* and decide if it can assume both ‘0’ or ‘1’ values. When this is true, we take a bit from the source data; copy it to this location and move to the next bit. Otherwise we write zero (nothing) to this pixel and go to the next. This method is much faster (order of N), however it does not generate codes as dense as the full enumeration method.

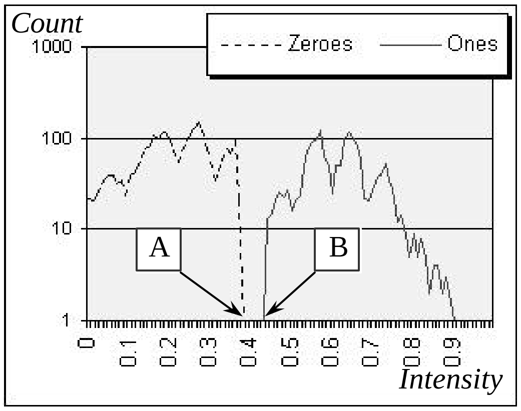


Figure 6: CB size 4x4 encoding results

We ran both algorithms for different sizes of CB, starting 1x2 up to 5x5 bits. The histogram in Figure 6 shows that the noise margins are maintained and there is no significant ISI. The utilization for the enumeration algorithm was 49-81% (as shown in Figure 7) with higher utilization for larger CBs. We also noticed that “tall” CBs do not perform as well as “wide” ones. For the same number of bits in CB, square rectangles yielded better results than non-square.

There are two interesting features shown in Figure 7. First, the best results for utilization are over 80%. This is much higher than reported results for static encoding, namely 45% for 4/9 modulation scheme[3,4]. This suggests that there may be modulation algorithms that will give good utilization and relatively low time complexity. The second feature is the trend of the surface to flatten out at CB sizes larger than 4x4. This suggests that it is not needed to go into large CB sizes to achieve good utilization results. Thus better time performance can be achieved for encoding schemes with smaller code blocks.

Utilization results for various CB sizes.

Y \ X	1	2	3	4	5	6
1		49	62	65	68	72
2	49	66	71	74	74	75
3	61	72	74	77	81	76
4	61	71	76	78	81	
5	57	73	77	80	80	
6	64	74	80			

Threshold: 0.40
Noise margin: 0.03

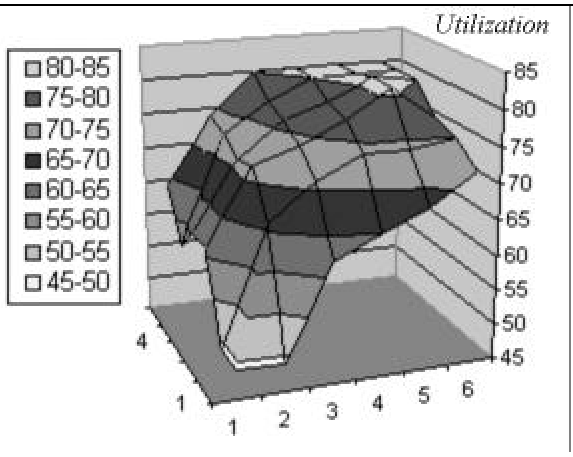


Figure 7: Full enumeration approach: Utilization vs. CB size

Returning to Figure 6 the histogram for 4x4 CB version encoding simulation shows that noise margin is maintained and there are no ambiguously read bits, i.e. no destructive crosstalk. The code density was 78%. Figure 8 presents a) binary, b) detector and c) sampled images of memory page fragment after using 4x4 encoding. There are no single isolated dark dots surrounded by bright light as an effect of encoding and thus the crosstalk is minimized. Even the detector image b) seems to have better contrast the raw data image in Figure 2b.

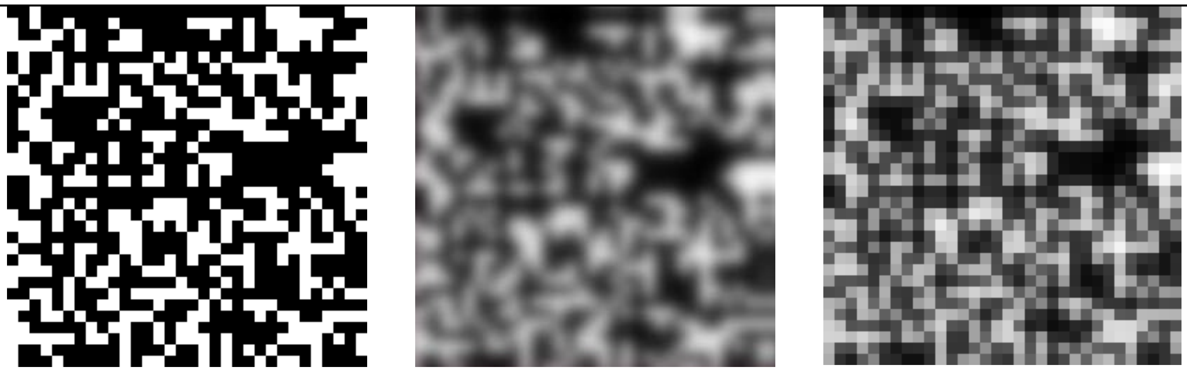

a) Binary data b) Detector image c) Sampled image

Figure 8: Encoded data images

The bit-by-bit approach gave good speed results while the utilization was between 55% and 60%. This algorithm slightly outperforms others recently published in terms of utilization. The algorithm's time complexity is linear to the number of bits in a code block. All code blocks can be decoded in parallel.

Another set of experiments determined how the dynamic encoding responds to parameter change. There are two parameters that influence the encoding: the threshold level and the noise margin. The experiments for the same input data set at different thresholds for 2x2 and 3x3 size code blocks generated the results in Figure 9. Some threshold values yield better utilization as others. The best utilization was 83% for the threshold value of 0.45 units of normalized light intensity. Even relatively small 2x2 blocks gave good utilization of 70%.

More experiments for 2x2 and 3x3 size code blocks showed how well the encoding performs for different noise margins. In the ideal case maximally wide noise margin is desirable. Unfortunately the experiments in Figure 10 show that the utilization drops quite steeply if we increase the noise margin over 0.04 units of normalized light intensity.

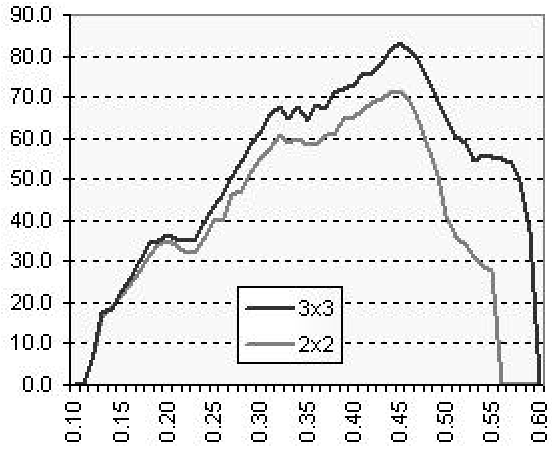


Figure 9: Utilization vs. Threshold (NM=.03)

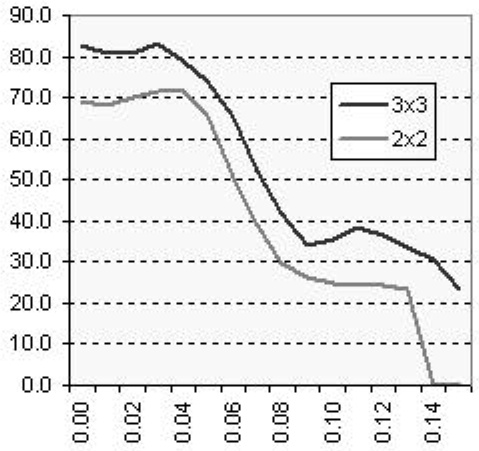


Figure 10: Utilization vs. Noise margin (T=.45)

2.2 SOFTWARE AND FIRMWARE REED SOLOMON ECC CODES

Reed-Solomon codes are a class of non-binary, linear block codes that offer multiple error correcting capability. They were first proposed in 1960[10] by the mathematicians for whom they are named. Since that time, many implementations of Reed-Solomon codes have been described in literature[11,12,13,14]. Of particular interest is recent work by Neifeld et al.[15,16] which introduced the notion of spectral processing. The decoder described in this report has the same overall organization as the Neifeld[16], and Blahut[14] coding system. However, we have adapted the pipeline structure to provide for increased performance in a minimal area FPGA implementation.

Unlike a binary code, a linear block code treats the data to be encoded as message block of *k* symbols where each symbol is encoded in *m*-bits and represents an element of a finite field of size 2^m . The codeword is built by appending (*n-k*) extra symbols to the message string to produce a block with a total length of *n* symbols. This *nxm*-bit word is referred to as an *(n,k)* Reed-Solomon (RS) code and has random error correction capability *t = (n-k)/2* symbols. In the FPGA implementation a sixteen element field was chosen yielding *m*=4, *n*=15 and *k*=9 and a 60-bit, (15,9) RS code word.

A finite field is a discrete set of symbols with operations, * (multiplication) and + (addition), over those symbols that define a well-behaved algebra. In our implementation we make use of the finite field GF(2^4) which is constructed from the elements 0 (additive identity), 1 (multiplicative identity), and $\alpha, \alpha^2, ..., \alpha^{14}$, where α is the root of special generating polynomial, P(x), of degree *m* which called a *primitive polynomial*. The choice of primitive polynomial determines the mapping of elements of GF(2^4) into four bit symbols. Our choice of $P(x) = x^4 + x + 1$ yields the following symbol alphabet for GF(16):

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

Addition of two symbols in GF(2^4) is obtained by a bitwise XOR of the bit-encoded representations of the symbols involved. Multiplication of two symbols in the field is defined as

$$A * B = \begin{cases} 0, & A = 0 or B = 0 \\ a^{(i+j) \bmod 15}, & A = a^i, B = a^j \end{cases}$$

The periodicity of the generating polynomial defines the Finite Field Fourier Transfer (FFFT) operation and its inverse (FFFT^{-1}) which are used to encode and decode codewords in a Spectral Reed-Solomon code . These operations map sets of n = 2^4 - 1= 15 symbols between the data (or temporal) space and the spectral (or frequency) space. Thus if *v* is a vector of symbols in temporal space, the FFFT computes V, which is a vector of symbols in spectral space and the FFFT^{-1} will take a vector of symbols V in spectral space and compute the vector v in temporal space according to:

$$FFFT: \quad V_k = \sum_{i=0}^{n-1} a^{i \bullet k} v_i$$

$$FFFT^{-1}: \quad v_k = \sum_{i=0}^{n-1} a^{-i \bullet k} V_i$$

The specifics of the encoding and decoding operations are shown in Figure 11. An *n*-symbol unencoded data word is assumed to consist of *k* data symbols and *n-k* check symbols. The *n-k* check symbols are initially set to the zero symbol, the additive identity element in the field. For coding purposes, the entire string of symbols is interpreted as a vector, *V*, in the spectral domain. The encoding operation is an inverse Finite Field Fourier Transform which produces a vector of symbols, *v*, in the temporal domain. This is the encoded data word that is stored in the memory system.

After being stored and retrieved, the data is processed by the multi-stage decoding and correction logic. The decoding step consists of a forward FFFT to restore the original spectral data vector. If errors were introduced during storage and retrieval, the error can be represented in temporal space as a vector *e* such that the received vector *v* is the (finite field) sum of code word *d* and error vector *e*.

$$v = d + e$$

After decoding, the linearity property of the FFFT operation provides that:

$$V = D + E$$

also holds in the spectral domain. However, since six of the D terms in the original code word were appended as zero constants, the symbols in this portion of the code word represent elements in the error vector only. These non-zero elements in the check portion of the retrieved vector V are the starting point for the correction algorithm. The goal of the correction algorithm is to calculate the remaining elements in the error vector E, such that when these elements are added to the elements of *V*, the original data is restored.

The first stage of error correction calculates the coefficients of a function called the error locator polynomial. The error locator polynomial, $\Lambda(x)$, is defined such that if an error, e_i, occurs at location *i* in the temporal space then α^{-i} is one of its roots. Due to the way in which $\Lambda(x)$ is defined, the inverse FFFT of this polynomial, $l(x) = \text{FFFT}^{-1}(\Lambda(x))$, has the very important property that $l_i * e_i = 0, \quad \forall \quad i \in \{0 \ldots n-1\}$. Application of the convolution theorem results in this set of *n* equations, which can be used to determine the coefficients of Λ and the unknown components of E:

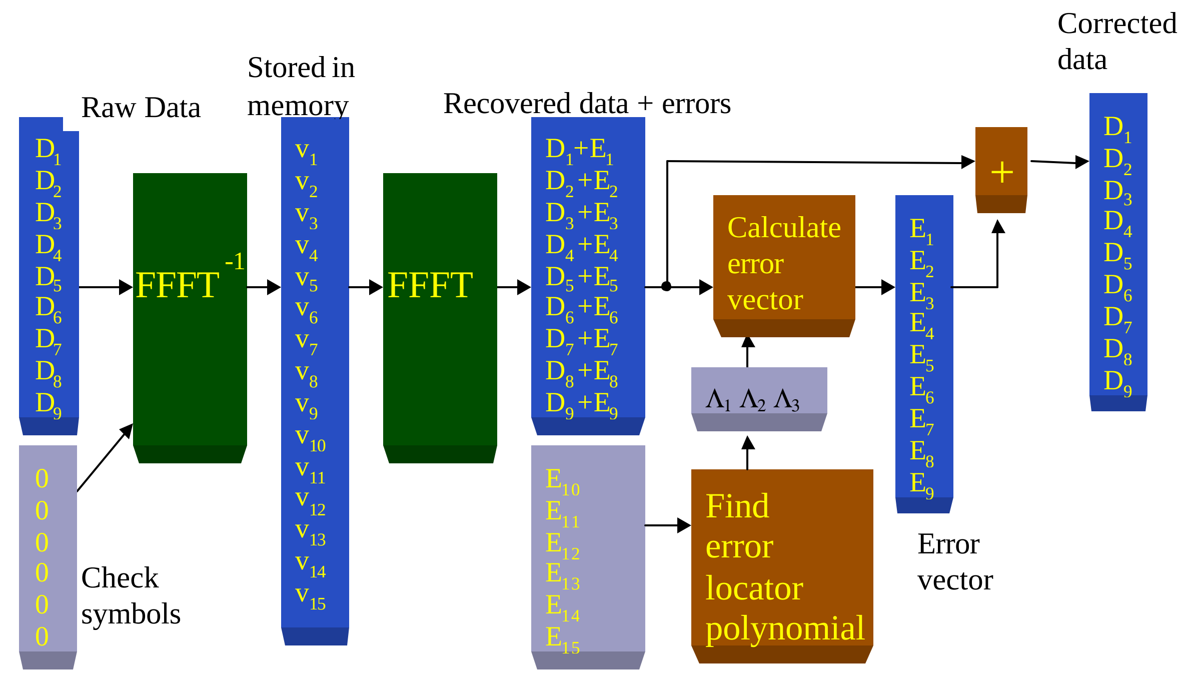

Raw Data
D1 D2 D3 D4 D5 D6 D7 D8 D9
0 0 0 0 0 0
Check symbols
FFFT-1
Stored in memory
v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15
FFFT
Recovered data + errors
D1+E1 D2+E2 D3+E3 D4+E4 D5+E5 D6+E6 D7+E7 D8+E8 D9+E9
E10 E11 E12 E13 E14 E15
Calculate error vector
Λ1 Λ2 Λ3
Find error locator polynomial
E1 E2 E3 E4 E5 E6 E7 E8 E9
Error vector
+
Corrected data
D1 D2 D3 D4 D5 D6 D7 D8 D9

Figure 11: Reed Solomon Coding and Decoding Process

$$\sum_{j=0}^{t} \Lambda_j E_{i-j} (\bmod x^n) = 0, i = 0,1,\ldots n-1$$

There will exist a subset of t of these equations which contain only the 2*t known components of E and the t + 1 coefficients of Λ. Making use of the fact that $\Lambda_0 = 1$, this reduces to a system of t equations with t unknowns which can be solved for the values of $\Lambda_1, \ldots, \Lambda_t$. This solution can be obtained through several different methods, our choice being the iterative Berlekamp -Massey (BM) algorithm.

Once the error locator polynomial has been calculated, the Λ coefficients can be used with the remaining equations from convolution result to calculate the unknown symbols of E in a process known as recursive extension (RE). The RE stage determines the remaining terms of the error vector by stepwise solution of the remaining equations, producing one new term per equation. Finally the each of calculated terms in the error vector is added (equivalent to subtraction in a finite field) to the retrieved data to regenerate the original data.

2.2.1 Spectral Reed Solomon Decoder Implementation

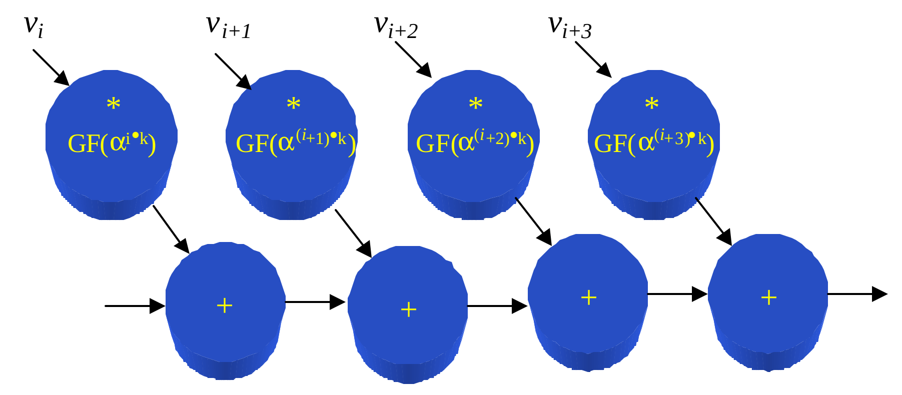
In this section, we describe the implementation of a Spectral RS decoder in a single Xilinx XC4036EX, field programmable gate array (FPGA) device. As we mentioned above, this is a firmware solution that allows different versions of the software to be loaded as necessary for testing, debugging, and optimizing alternate configurations. The circuitry itself is designed in VHDL (*V*HSIC *H*ardware *D*escription *L*anguage), which provides for a textual description of the algorithm much like a conventional programming language. Software tools synthesize the logic directly from this description and place and route tools map the logic into the combinational logic blocks (CLB's) and interconnection network of the FPGA.

2.2.1.1 FFFT Implementation

As explained above the k-th symbol of the FFFT is computed from:

$$V_k = \sum_{i=0}^{n-1} a^{i*k} v_k$$

where the operations for multiplication and addition are defined by the finite field algebra. In GF(2^4), where each symbol consists of four bits, these operations are defined by a combinational logic function of 8 inputs and 256 outputs. While addition is easily implemented as a bit-wise XOR operation, multiplication cannot be readily minimized and requires over 30 gates, (5 CLB's in an FPGA) to implement. Although the most obvious realization of the FFFT operation is to compute V_i by a multiplication addition tree, this would require some 225 multipliers and would consume most of the capacity of available reconfigurable logic devices. However, unlike a conventional FFT operation, finite field FFT's can be implemented such that one of the two input symbols, $\alpha^{i \bullet k}$, is known *a-priori* for



v_i
v_{i+1}
v_{i+2}
v_{i+3}
*
GF($\alpha^{i \bullet k}$)
GF($\alpha^{(i+1) \bullet k}$)
GF($\alpha^{(i+2) \bullet k}$)
GF($\alpha^{(i+3) \bullet k}$)
+

Figure 12: One Row of the Pipelined Computation Array

each of the 15 multiply operations of the k-th term. Thus, each of these multiply operations can be implemented as a 4 input, 16 output function which can be implemented in 2 CLB's. Using these simpler multiply operations, the multiply-add circuitry for the k-th term can be built as shown in Figure 12. Note that Figure 12 shows only a part of one row from the overall FFFT computation array. For a (15,9) code, that array consists of 15 rows and 15 columns, each column corresponding to an input term in v and each row being summed to compute a term in V.

To complete the implementation of the FFFT, two remaining issues need to be resolved. In the first case, pipeline registers must be added in order to meet timing specifications. These are arranged in two groups, one group along the rows of the FFFT array such that a new partial sum for each term in V is computed in each clock cycle, and another group that buffers the input code word to delay the input of each v term until the partial sum has propagated to the corresponding column. For example, in the first clock cycle, v_0 is broadcast to all of the multiply/adder stages in column *0* all of the partial product/sums requiring this term are calculated and buffered in pipeline registers between column 0 and 1. In the next clock cycle, the partial sums are accumulated with terms computed from v_1 in column one and stored in pipeline registers between columns one and two. Thus, it is necessary to delay the entry of the v_1 term in the FFFT array until the second clock cycle. Note also that in the second clock cycle, the v_0 terms for a different code word are being computed simultaneously. In fact, this configuration of the pipeline computes one partial result for each of 15 different code words on each clock cycle. After an initial delay to fill the pipeline, a new code word enters the pipeline and completed one exits on each clock cycle.

2.2.1.2 Berlekamp Massey Implementation

The 2t known error values from the FFFT output are used as the input to the Berlekamp-Massey(BM) stages which will iteratively solve for the error locator polynomial. A typical BM stage consists of the logic blocks shown in the inset to Figure 13. It will read up to three of the known error values from the left pipeline register as well as the previous values of Λ, B, and L from the right pipeline register, calculate the intermediate quantities Δ and δ, and use this information to refine the value of Λ and determine the new values of B and L. The values of Λ, B, and L input to the first stage are fixed so that after 2t interations, the output value of Λ will contain the true error locator polynomial.

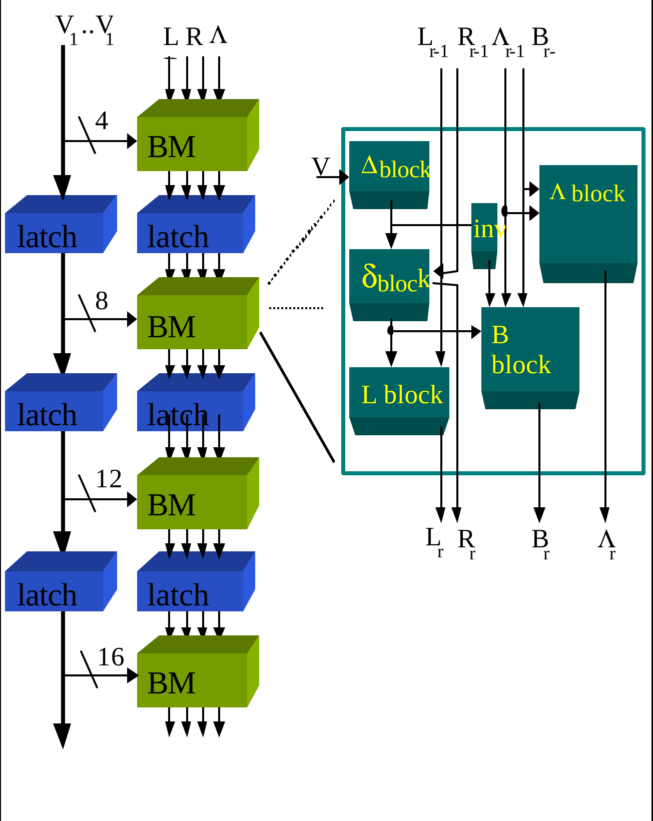


Figure 13: Berlekamp -Massey Solver Implementation

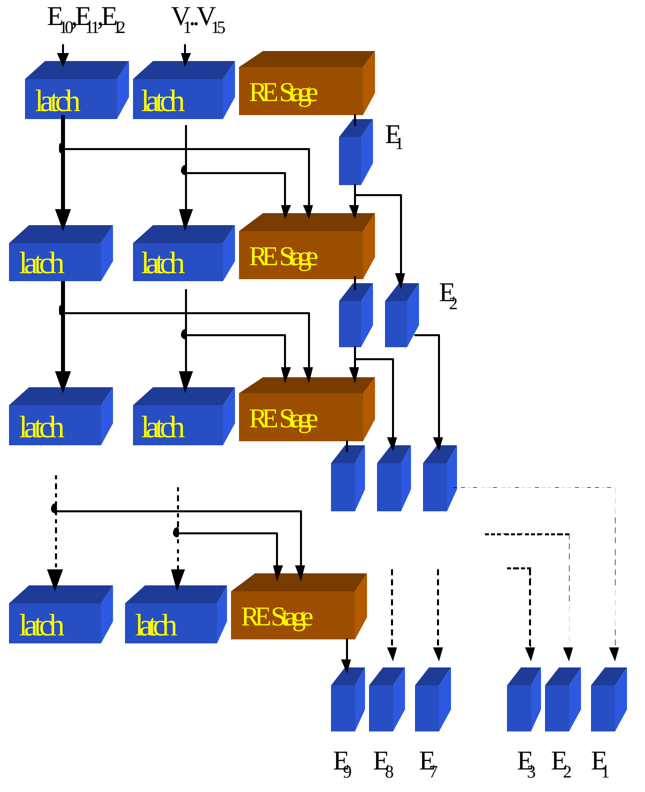


Figure 14: Recursive Extension

The dominant hardware requirements for the BM stages will be the GF multipliers and inverters. Some properties of the error locator polynomial can help us to characterize the number of such components needed. First, the maximum degree of Λ is t, and consequently the maximum degree of B must be t-1. Thus, we will need t+1 symbols to represent Λ and t symbols for B. Second, the first element of Λ is always 1 and never needs to be explicitly calculated, so the number of symbols needed for Λ is reduced to t. As a result, the Λ block requires at most t full multipliers to compute the term $\Lambda^r = \Delta_r * x * B^{r-1}(x)$ and the B block requires at most t full multipliers to compute the term $B^r = (\Lambda^{r-1}/\Delta^r)d_r + (xB^{r-1})(1-d_r)$. The B block also requires an inverter to calculate Δ^{-1}. Finally, the Δ block will multiply up to t known error symbols by coefficients of Λ requiring another t multipliers. Thus, at the worst case a BM stage requires 3t full GF multipliers and a GF inverter. Furthermore, by pipelining the BM implementation the total hardware requirement can be reduced significantly because each copy of the BM stage computes a fixed level of the iteration and in the beginning and end stages some portions of the computation are unnecessary.

2.2.1.3 Recursive Extension Implementation

At the output of the Berlekamp-Massey stage we obtain all of the coefficients of the error locator polynomial. With these values and the known error values we can calculate the unknown terms of the error vector in the Recursive Extension (RE) stages. Each stage uses the t error locator coefficients and the previous three coefficients of the error vector to calculate the next symbol, which is fed forward to the next stage. An RE stage requires t full

The simulation was clocked at 33mhz and a sequence of five coded data words were introduced in the *raw optical data*

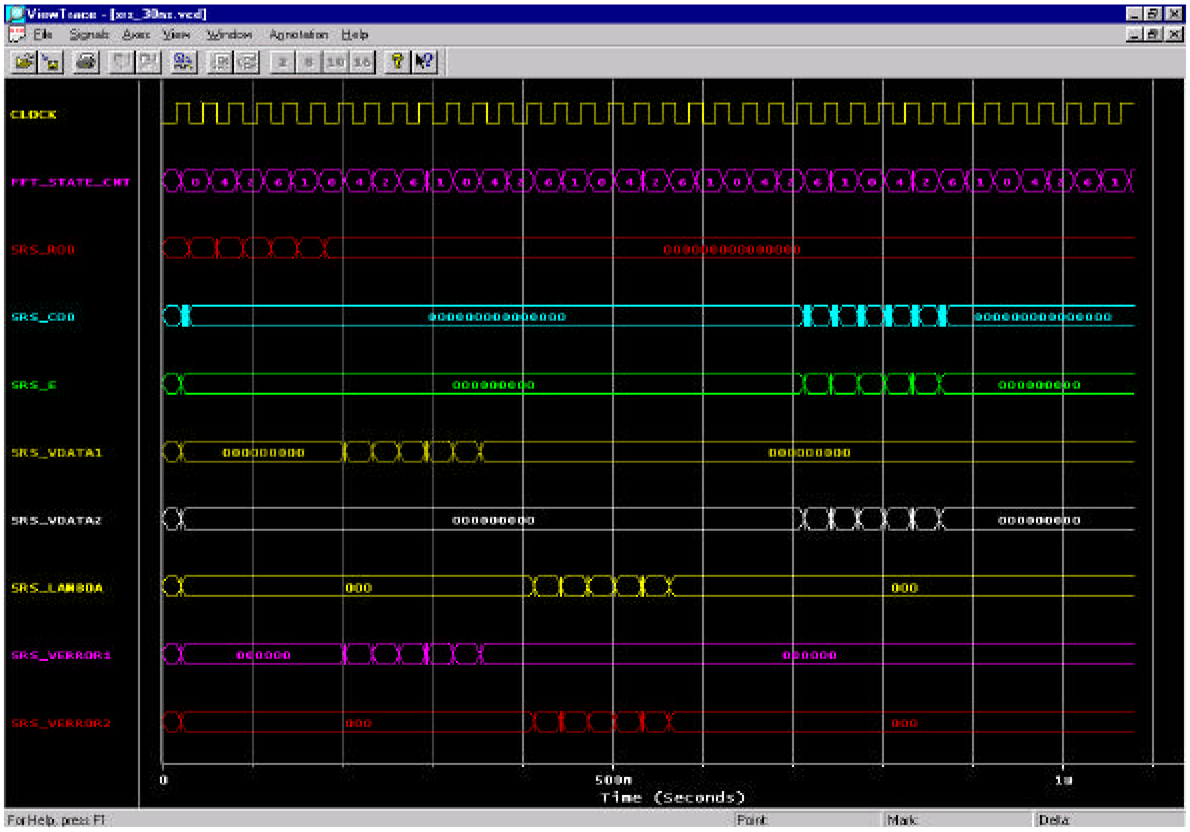


Figure 15: Simulation from FPGA Implementation

GF multipliers which operate in parallel on the inputs and a t symbol input GF adder which sums the outputs of the multipliers to produce the new symbol. Our pipelined implementation contains k stages separated by latches (Figure 14). At the output of each stage, the error locater coefficients, the previously computed terms of the error vector, and the new error vector symbol are latched. The last stage produces the k symbols of the unknown part of the error vector. This is then subtracted from the k data symbols in the received vector V to recover the original vector C.

2.2.2 Performance Data

Figure 15 shows the output from a timing annotated simulation of the Spectral RS decoder. This simulation is based on the netlist generated from the synthesized, placed, and routed logic exactly as it is downloaded to the FPGA for circuit operation. The netlist is annotated by the extraction tool with highly accurate timing data for the specific device used and the interconnection paths required.

(SRS_ROD) input lines. After a pipeline delay of 21 cycles, the corrected data begins to appear one symbol per clock cycle on the *corrected optical data* (SRS_COD) output lines. The remaining traces show a number of intermediate values, which are monitored, between the stages of the decoder logic.

In addition to the timing simulations, we have also verified the functional correctness of the algorithm using a software implementation. Figure 16 shows a sequence of images beginning with raw unencoded data, followed in sequence by the encoded data, encoded data with errors introduced, decoded and uncorrected data, and corrected data.

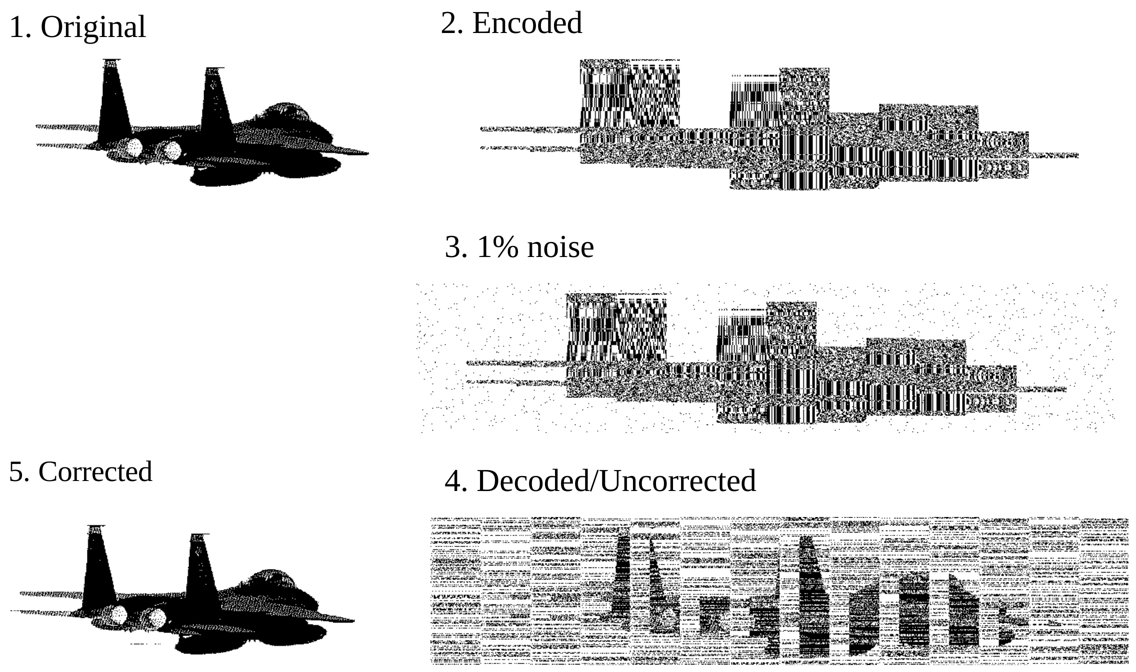


Figure 16: Sequence of Images Showing Coding, Noise Injection, and Decoding Results

2.3 TRANSACTION LEVEL CODES FOR APPLICATION ORIENTED STORAGE

The third area of our research was an investigation of an approach for dealing with the mismatch between the high data transfer rates from page oriented memory and the ability of a processor to do perform useful computations on the data in real time. Our solution is based on the addition of computation power at the memory interface such that routine computations on specific types of stored data, such as images can be preformed before the data in transferred into system memory. This type of access would be inherently application specific and would reduce the computation load on the processor to generating and processing a series of transaction with the new, intelligent, memory interface.

The application that we chose to study was image scene reconstruction and rendering using a ray-tracing algorithm. Thus, a scheme might be stored as a collection of objects and object relationships that can be restored into a viewable rendered image at read time. Such a system would have wide application in both military and consumer settings.

2.3.1 What is Ray Tracing

Ray Tracing is a method that allows you to create photo-realistic images on a computer, dedicated hardware, or mix of it. The images are created from a model of the world, viewpoint and the viewing direction and angle.

First the model must be defined. It can contain various kinds of primitive objects, including spheres, boxes, planes, etc. Complex objects can be built uniting, intersecting and subtracting other primitive or complex objects. Every surface of an object has specific properties, such as light reflection and refraction coefficients, color and texture. Some special objects are created, acting as light sources. All this can be done using an interactive modeling system, like a CAD package, or by creating a text file that has a programming language-like syntax to describe the elements.

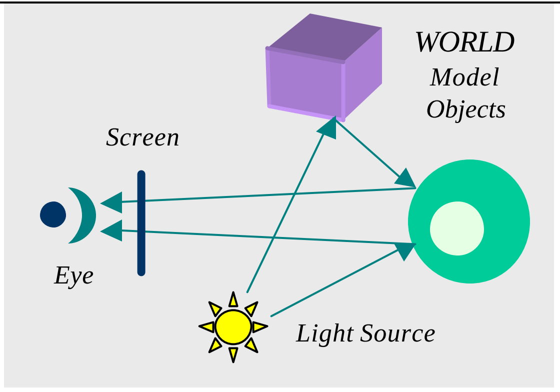


Figure 17: Ray tracing concept

The model is the input for the ray tracing system. The system computes physical effects of light rays traveling, reflecting and refracting through the model environment (Figure 17). Finally, the system generates an image as if looking at the model from the preset viewpoint.

The main drawback of ray tracing is that it is computationally very intensive. Using conventional software programs such as Persistence-of-Vision (POV) for tracing images can take anything from a few minutes to many days. Therefore custom hardware and firmware solutions are developed to speed up the process. Even greater challenge is to apply ray tracing for video generation purposes, where several image frames have to be generated for each second of playback. The ultimate challenge is real-time high-definition video with ray tracing.

2.3.2 Algorithm

The ray tracing method generates a two-dimensional image from a set of objects, which represent a world model. The result is an image, as it would be realistically seen from a predefined viewpoint, be it an eye of an observer or an imaginary camera.

The image is interpreted as a two-dimensional pixel array. Each pixel of the image is computed by tracing a ray of light, which hit the view-plane at this pixel. The path of the ray is tested against all the objects for an intersection point. If an intersection point is found, any of the following can happen:

- The ray is absorbed
- The ray is reflected: a new ray is generated.
- The ray is refracted: a new ray is generated.

All the newly generated rays are processed recursively until a limited recursion depth is reached.

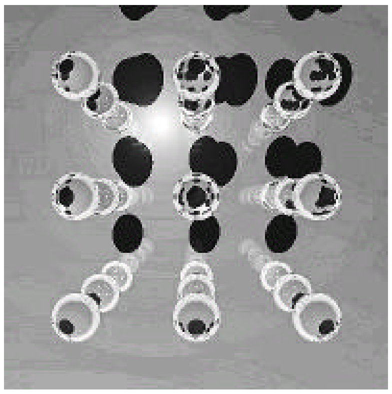


Figure 18: Ray tracing example: 3x3 glass ball grid against white wall, one light source.

At the each intersection point new rays can be generated, but also the light contributing to the originating pixel is computed (see example image in Figure 18). The amount and color of light is determined depending on the distance from the light source, light source properties and the intersection point surface properties.

The intersection test specifics between a ray and an object depend on the type of the object. Essentially a different mathematical formula is used for each type of object.

In order to achieve realistic results, high precision computing is required; therefore it has to be done using floating-point arithmetic as opposed to faster and simpler integer arithmetic.

2.3.3 Complexity Analysis

To illustrate how demanding the ray tracing implementation can be, consider the following parameters for the ray tracing system:

- Image size: 1000x1000 pixels
- Frame rate: 30 fps
- Number of primitive objects: 1000
- Number of light sources: 1
- Recursion depth of ray tracing: 3

Simplistically speaking, the complexity of the ray tracing method boils down to a number of intersections between a ray and an object that are computed. In our example each frame has 10”6 pixels, which leads to the same number of outgoing rays. The recursion depth multiplies this number by 3, which leads to 3*10”6 rays in the system. Each ray has to be tested against each primitive object for intersection, thus we have 3*10”9 intersection tests per frame. Taking in account the frame rate 9*10”10 intersections must be computed every second.

Every intersection is computed differently for different types of objects. Let us say it takes about 10 floating-point operations on the average for each computation. The total complexity, which has to be matched, is 9*10”11 flops/second.

If we assume that modern pipelined processors can issue a floating-point operation each clock cycle, the clock speed we need a single processor system to be able to meet our requirements is **900GHz**. Moreover, this result was obtained on relaxed assumptions. Realistically there are more computations to be performed, such as:

- Searching for objects in the model database.
- Operations for object temporal maintenance, such as changing their position in time, modeling movement.
- In realistic models there may be more than 1000 primitive objects.
- Floating point operations may take more than 1 clock cycle to issue.
- Operating system overhead.

All this makes ray tracing very compute expensive and unavailable for real time video rendering using current off the shelf computers and software.

2.3.4 Parallel Processing, Firmware Implementation

Our approach takes advantage of the high parallelism of operations performed by ray tracing. Many pixels and respective ray properties can be computed in parallel, independently of each other. We also want to customize the hardware for maximally efficient computation.

The model of the world is stored on an optical memory media (Figure 19). Once the request for it is made, the appropriate subset of the objects in immediate vicinity of the observer is brought to the cache memory for fast access.

The ray tracing computation is done on reconfigurable FPGA devices. Since the FPGA budget is limited, they are reconfigured to perform intersection test for different object type and rays. Since every object has to potentially deal with every ray, the reconfiguration will not occur often and will not create a bottleneck for the total computation time.

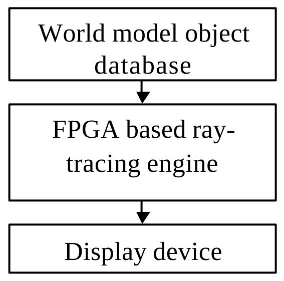


Figure 19: Approach

2.3.5 Simulation Results

The investigation shows that for ray tracing method implementation for intersection test requires an FPGA of size 467 rows of CLBs, which is also the total pipeline depth by 1010 columns. Since presently there is no FPGA available of this size, the system can only be built based on an array of FPGAs.

Ray-tracing software was developed for simulation purposes. The simulation experiments show that the cost of one intersection on a Pentium class 300MHz computer is on the average 9.34us. The computer system was also running at a faster clock speed than FPGA based, but the computer also had to perform other computations, such as the operating system overhead.

Meanwhile, for the FPGA based implementation running at 100MHz clock speed the throughput time of an intersection test issue is 10ns. Since every computation is fully pipelined, one test can be issued every clock cycle. The FPGA based system outperforms the standard computer/software solution by almost 3 orders of magnitude in terms of computation time.

The FPGA system can perform 10”8 intersection computations each second. This is sufficient for the real time video, if the given parameters are somewhat relaxed to 100x100 pixel frame, and the number of objects is limited to 100. However, using several instances of the FPGA based system can increase these numbers. Developing and implementing more efficient algorithms can also improve the performance. For instance, one approach is to subdivide space so that not all the objects have to be tested for intersection against each ray.

One challenge of an FPGA based ray tracing is the large number of FPGAs to accommodate 467x1010 CLB requirement. If an FPGA of size 100x100 CLBs is used, then 5x11 array of FPGAs is needed to build it. The implementation of such system would have to deal with high power, electronic noise and data communication issues.

3 SUMMARY AND CONCLUSIONS

Optical memories are good candidates to match requirements for large, fast and reliable data storage. We have presented dynamic data modulation encoding technique to attain the reliability with lower spatial overhead than conventional methods. Our experiments show utilization up to 81% while recent publications[3,4] present 4/9 encoding (45% utilization). The 2x2 version of the full enumeration approach seems to be a good candidate for implementation due to its utilization of 70% and relatively good performance (only 16 different codes to test for each code block).

Unfortunately the computational complexity of full pattern enumeration algorithm is higher than desired; therefore there is a need to search for alternative algorithms and heuristics to speed up the encoding and decoding. We have shown one such algorithm achieving 60% utilization. We assumed existence of "smart" optical readout head for our method implementation. We have previously presented a proof of concept of an optically reconfigurable FPGA[7], which may serve as a prototype for the readout head. To further improve the performance of the algorithms parallel implementations of the encoding and the decoding for mesh type architectures should be investigated.

Another challenging issues in the design of an optical memory system interface is the error detection and correction mechanism. This report has presented the approach that was taken in the OE memory hierarchy prototype at the University of Pittsburgh. We have chosen to implement a Spectral Reed-Solomon code for its advantages of burst error tolerance and for the pipeline processing advantages of it decoding algorithm. The implementation technique used a hardware description language and reconfigurable logic blocks thus providing maximum flexibility to adapt the implementation to specific materials or error characteristics as optical memory material properties are refined. All of the logic necessary to decide a 60-bit, (15,9), SRS code can be implemented in a single device and operating at a clock rate of 33 MHz.

Finally, one approach to handling high data transfer rates is to move computational power to the interface and implement a transaction oriented storage system. We have investigated the feasibility of implementing such a system for real time rendering of stored scene data. In our investigation the ray tracing method was mapped to the architecture based on an array of FPGAs. The computation is distributed over the array of FPGAs. The FPGAs are reconfigured as needed to accommodate different computations and compensate for lack of resources.

The simulation experiments of the developed ray tracing software show that the cost of one ray tracing operation (intersection) on a Pentium class 300MHz computer is on the average 9.34us. The FPGA based implementation running at one third of the speed of the computer the throughput is 10ns. This indicates that FPGA based system outperforms the standard computer/software solution by almost 3 orders of magnitude in terms of computation time.

The FPGA system performance is sufficient for the real time video representation of the defined world model for a medium size (100x100 pixel) frame and limited number of objects. However, by using several instances of the FPGA based system these parameters can be increased due to scalability of the system. Developing and implementing more efficient algorithms can further improve the performance.

BIBLIOGRAPHY

1. Donald M. Chiarulli and Steven Levitan. Optoelectronic Cache Memory System Architecture. *Applied Optics*, 35(14):2449-2456, May 1996.
2. John F. Hutton, George A. Betzos, Maureen Schaffer, Pericles A. Mitkas, Error correcting codes for page-oriented optical memories. *Proceedings of SPIE, Materials, Devices, and Systems for Optoelectronic Processing, Volume 2848*, 1996.
3. Dhawat E. Pansatiankul, Alexander A. Sawchuk, Two-dimensional modulation codes for page-oriented optical data storage systems. *Proceedings of Optics in Computing*, p.79-81, 2001.
4. Dhawat E. Pansatiankul, Alexander A. Sawchuk, Multidimensional modulation codes and error correction for page oriented optical data storage. *Optical data storage topical meeting 2001*, p.94-97, 2001.
5. J. E. Ford, S. Hunter, R. Piyaket, and Y. Fainman. 3-d Two Photon Memory Materials and Systems. In *Proceedings of SPIE – The International Society for Optical Engineering*, volume 1853, pages 5-13. SPIE, 1993.
6. Haichuan Zhang et al., Multi-layer Optical Data Storage Based on Two-photon Recordable Fluorescent Disk Media. *Eighteenth IEEE Symposium on Mass Storage Systems*, 2001.
7. Leo Selavo, Steven P. Levitan and Donald M. Chiarulli, An Optically Reconfigurable Field Programmable Gate Array. *Proceedings of Optics in Computing*, 1999.
8. G. Burr, F. H. Mok, and D. Psaltis. Storage of 10000 holograms in LiNb03:Fe. In *Technical Digest Series: Proceedings of 1994 Conference on Lasers and Electro-Optics and The International Electronics Conference CLEO/IQEC*, volume 8, page 9. Optical Society of America, May 1994.
9. Joshua L. Kann et al., Mass Storage and retrieval at Rome Laboratory. *Proceedings of 5th NASA Goddard Mass Storage Systems and Technologies Conference*, pp. 389-406, 1996.
10. I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *J. Soc. Ind. Appl. Math*., 8:300-304, June 1960.
11. Howard M. Shao, T. K. Troung, Leslie J Duetsch, Joseph H. Yuen, and Irving S. Reed. A VLSI Design of a Pipeline Reed-Solomon Decoder. *IEEE Transactions on Computers*, C-34(5):393-403, May 1985.
12. Howard M. Shao, T. K. Troung, Leslie J Duetsch, Joseph H. Yuen, and Irving S. Reed. On the VLSI Design of a Pipeline Reed-Solomon Decoder using Systolic Arrays. *IEEE Transactions on Computers*, 37(10):1273-1280, October 1988.
13. Po Tong. A 40-mhz Encoder-Decoder Chip Generated by a Reed-Solomon Code Compiler. In *IEEE Custom Integrated Circuits Conference*, 1990.
14. Richard E. Blahut. A Universal Reed-Solomon Decoder. IBM J. Res. Develop., 28(2):150-158, March 1984.
15. Mark A. Neifeld and Jerry D. Hayes. Error-correction schemes for volume optical memories. *Applied Optics*, pages 8183-8191, December 1995.
16. Mark A. Neifeld and Satish K. Sridharan. Parallel Error Correction using Spectral Reed-Solomon Codes. *Journal of Optical Communications*, Vol.18, No.4, pp.144-150, 1997.