

Error detection and correction in an optoelectronic memory system

Robert Hofmann, Madhulima Pandey, Steven P. Levitan, and Donald Chiarulli

ABSTRACT

This paper describes the implementation of error detection and correction logic in the optoelectronic cache memory prototype at the University of Pittsburgh. In this project, our goal is to integrate a 3-D optical memory directly into the memory hierarchy of a personal computer. As with any optical storage system, error correction is essential to maintaining acceptable system performance. We have implemented a fully pipelined, real time decoder for 60-bit Spectral Reed-Solomon code words. The decoder is implemented in reconfigurable logic, using a single Xilinx 4000-series FPGA per code word and is fully scalable using multiple FPGA's. The current implementation operates at 33Mhz, and processes two code words in parallel per clock cycle for an aggregate data rate of 4Gb/s. We present a brief overview of the project and of Spectral Reed-Solomon codes followed by a description of our implementation and performance data.

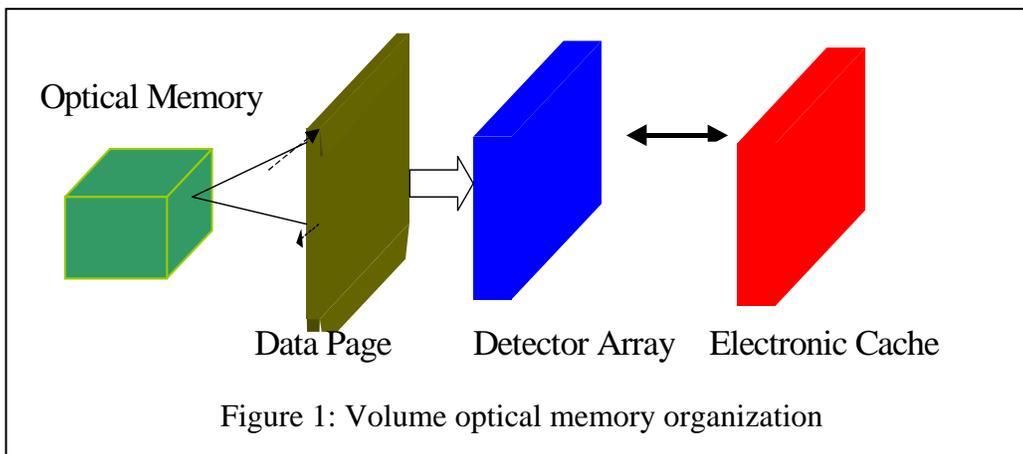
Keywords: optical storage, optoelectronic, error correcting code, Spectral Reed-Solomon, cache, FPGA, 3-D optical memory

1. INTRODUCTION

One of the ongoing trends in computer architecture is the continuous increase in the ratio between the access time for high capacity memory systems and the cycle time of clock level CPU operations. In order to reconcile this difference, system designers have adopted hierarchical memory structures for many years. These hierarchies are built using multiple memory modules of varying sizes and access latencies. By exploiting a characteristic of program behavior known as locality of reference, it is possible to build a hierarchical memory system which supports a large address space but has an average access latency many orders of magnitude smaller than the actual latency of the slowest (and largest) memory device. The system described in this paper is an implementation of an optoelectronic (OE) memory hierarchy in which the base of the hierarchy is implemented in an optical page oriented memory¹. When this memory system is interfaced to a personal computer, the optical memory becomes a part of the processor address space from which instructions and data can be fetched directly during program execution.

Optical page oriented memories such as those based on photorefractive² or two-photon³ materials have a three dimensional organization and are typically accessed in a two dimensional page format as shown in Figure 1. They combine the advantages of high storage capacity with high bandwidth due to spatially parallel data transfers.

Sources of error for these memories include media characteristics, detector noise, and optical system tolerances. These problems make aggressive error correction techniques necessary in order to maintain acceptable system performance. Unfortunately, choosing an efficient and effective error correcting code is difficult, since many of the materials used in optical memory have not been fully characterized for error performance. There are systems for modeling error behavior which, with certain assumptions about the memories, can be used to test error correction mechanisms in simulation



environments. In our work, reconfigurable logic elements are used at the OE interface to provide a firmware test bed in which various error correction algorithms can be tested without the need to redesign the hardware.

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.

The balance of this paper will focus on the theory, implementation and performance of our spectral Reed-Solomon decoder. After a brief description of the system architecture in section two, we present a general discussion of Reed-Solomon codes and the finite field algebra on which they are based in section three. Section four then describes details of the implementation and section five presents performance data from simulations based on timing extracted from the FPGA designs.

2. PROTOTYPE SYSTEM

Figure 2 shows a block diagram of a prototype personal computer system with an optoelectronic memory system. The prototype is implemented as an add-on memory controller connected to the PCI bus of a Gateway 2000 Model P5-133 personal computer. All of the components shown in Figure 2, except for the OE cache, detector array and the optical memory, are packaged in the standard system configuration for this PC. The optical memory is accessed in 128x128 bit pages, which are captured on an external detector head and transferred by row through a ribbon cable bundle to the OE cache board. A single, long geometry PCI bus board implements the OE cache memory, bus multiplexing circuits, and the cache controller.

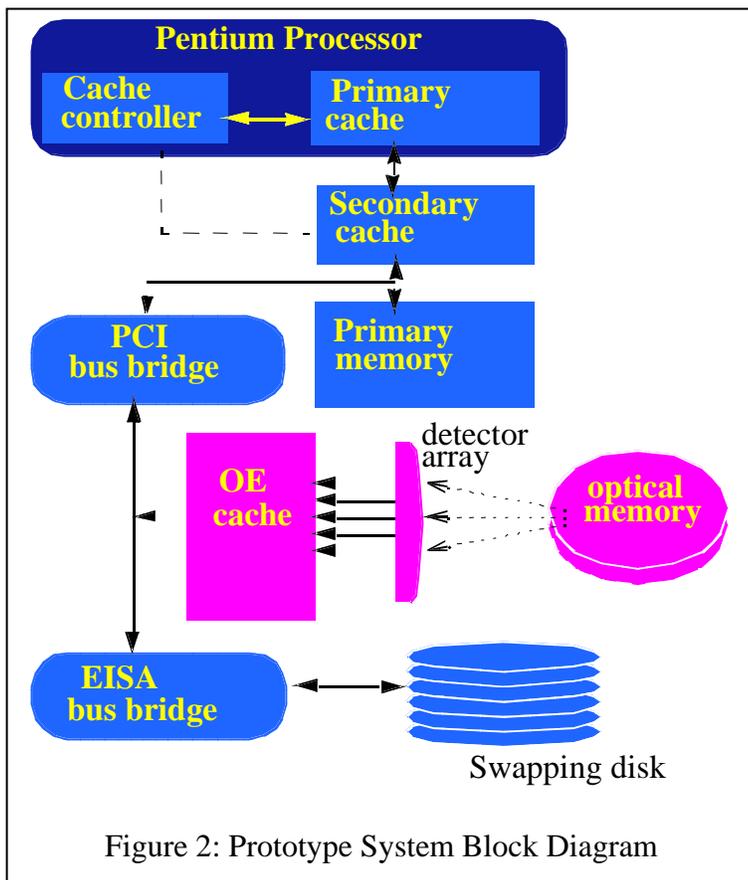


Figure 2: Prototype System Block Diagram

Figure 3 shows a block diagram of the OE cache and controller board. The design is based on a *slice* architecture that can be easily adapted to various optical memory page sizes. A single, 128-bit, slice is shown in the figure. Access to the PCI bus is obtained via an AMCC 5933 PCI controller chip that operates in pass-through mode to demultiplex address and data information and to provide synchronization between the bus and the optical cache controller. The OE cache controller is implemented in a Xilinx 4028 series FPGA. It accepts address and synchronization signals from the PCI controller and uses 4-way, set-associative mapping to determine if the requested address is available in the optical cache. If it is, the SRAM modules in the cache are accessed and data is returned directly to the PCI bus via the Bus-Mux/ECC circuitry. If not, the address of the requested page is passed on to an external optical memory controller that initiates an optical memory access. Once the requested page of optical memory is available at the detectors, incoming data is transferred in parallel by line from the detector array.

Error correction is performed in real time on each line of the incoming data within the Bus-Mux/ECC FPGA. Corrected data is transferred into the SRAM chips. The OE cache controller handles

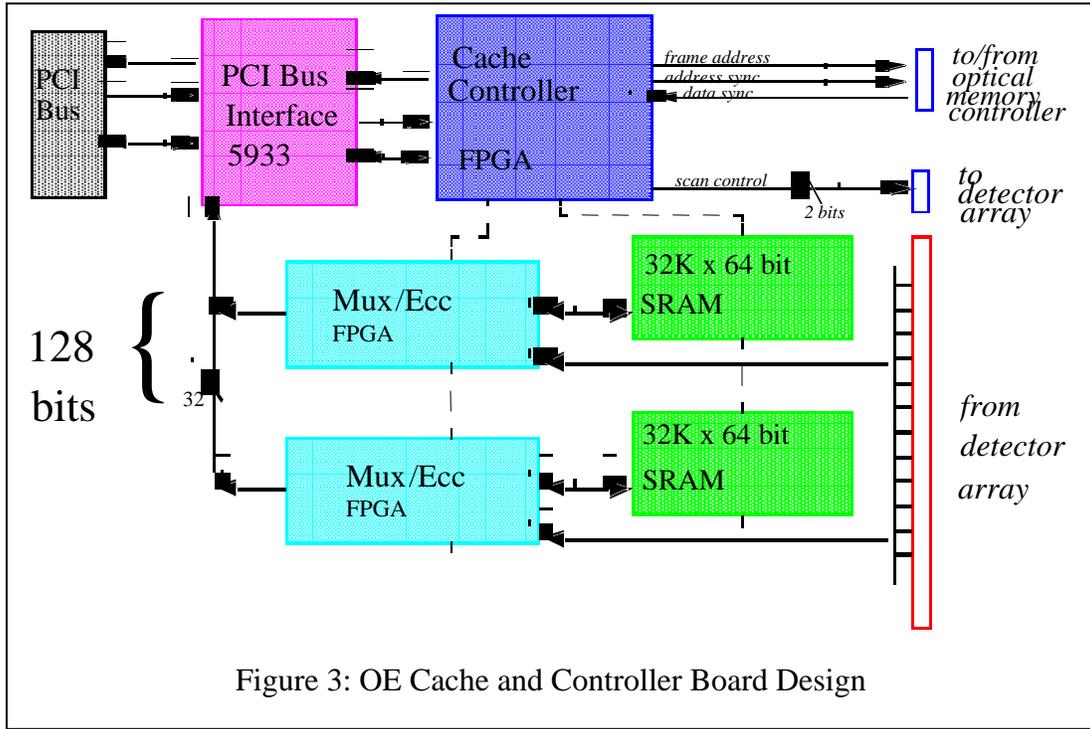


Figure 3: OE Cache and Controller Board Design

synchronization and controls the transfer. While loading the incoming optical data, the cache controller simultaneously monitors the data stream and latches the requested word into an internal buffer. This allows the memory request to be satisfied without the additional cycles required for a second access to the OE cache SRAMS. In order to understand the implementation of the ECC computation, the next section provides background on Reed-Solomon codes.

3. REED-SOLOMON CODES

Reed-Solomon codes are a class of non-binary, linear block codes that offer multiple error correcting capability. They were first proposed in 1960⁴ by the mathematicians for whom they are named. Since that time, many implementations of Reed-Solomon codes have been described in literature^{5,6,7,8}. Of particular interest is recent work by Neifeld et al.^{9,10} which introduced the notion of spectral processing. The decoder described in this paper has the same overall organization as the Neifeld¹⁰, and Blahut⁸ 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 $n \times m$ -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 $\alpha, \alpha^2, \dots, \alpha^{14}$, 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

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

The periodicity of the generating polynomial defines the Finite Field Fourier Transform (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:

$$FFFT : V_k = \sum_{i=0}^{n-1} \alpha^{i \cdot k} v_i$$

$$FFFT^{-1} : v_k = \sum_{i=0}^{n-1} \alpha^{-i \cdot k} V_i$$

The specifics of the encoding and decoding operations are shown in figure 4. 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,

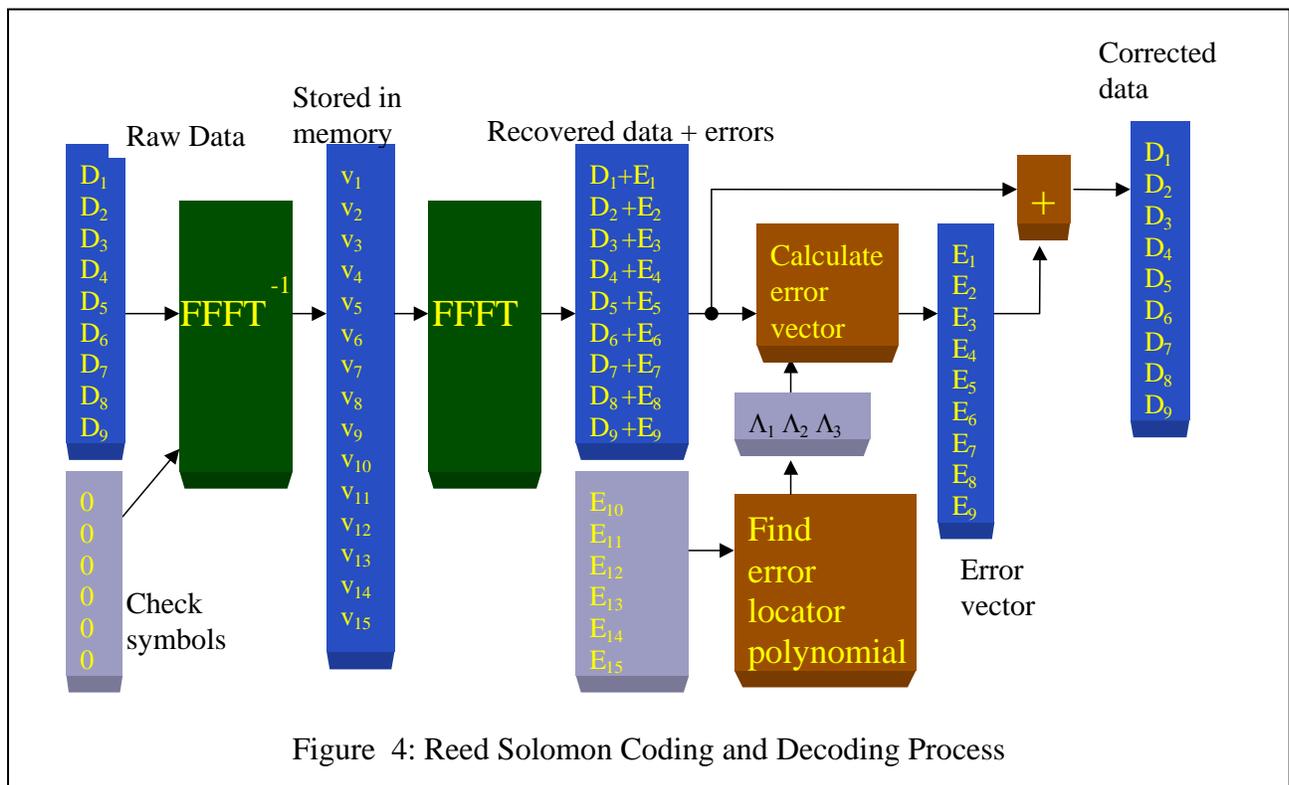


Figure 4: Reed Solomon Coding and Decoding Process

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, $\lambda(x) = \text{FFFT}^{-1}(\Lambda(x))$, has the very important property that $\lambda_i * e_i = 0, \forall i \in \{0, 1, \dots, n-1\}$. 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 :

$$\sum_{j=0}^t \Lambda_j E_{i-j} \pmod{x^n} = 0, i = 0, 1, \dots, n-1$$

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 $\Lambda_1, \dots, \Lambda_t$. 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.

4. SPECTRAL REED-SOLOMON DECODER IMPLEMENTATION

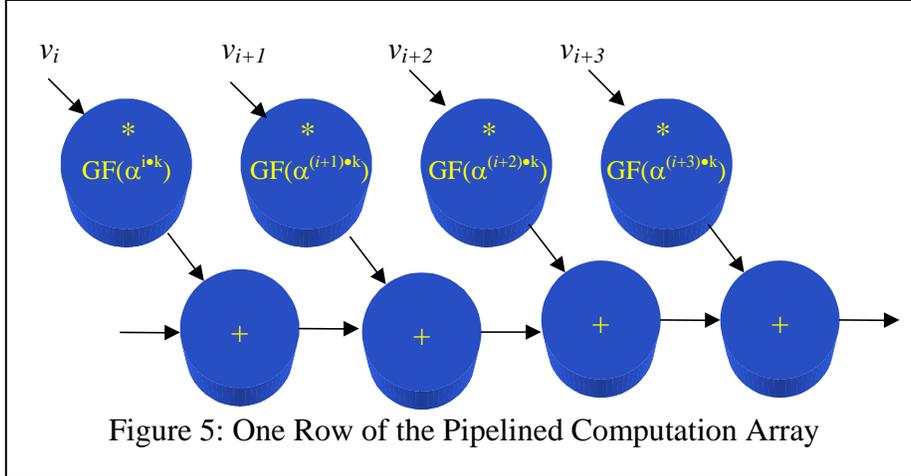
In this section, we describe the implementation of a Spectral RS decoder in a single Xilinx XC4036EX, field programmable gate array (FPGA) device. As we mentioned above, this is a firmware solution that allows different versions of the software to be loaded as necessary for testing, debugging, and optimizing alternate configurations. The circuitry itself is designed in VHDL (*VHSIC Hardware Description Language*), 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.

4.1 FFFT IMPLEMENTATION

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

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

where the operations for multiplication and addition are defined by the finite field algebra. In $\text{GF}(2^4)$, 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, $\alpha^{i \cdot k}$, is known *a-priori* for 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 5. Note that figure 5 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.

4.2 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 6. 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 $2t$ iterations, the output value of Λ will contain the true error locator polynomial.

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

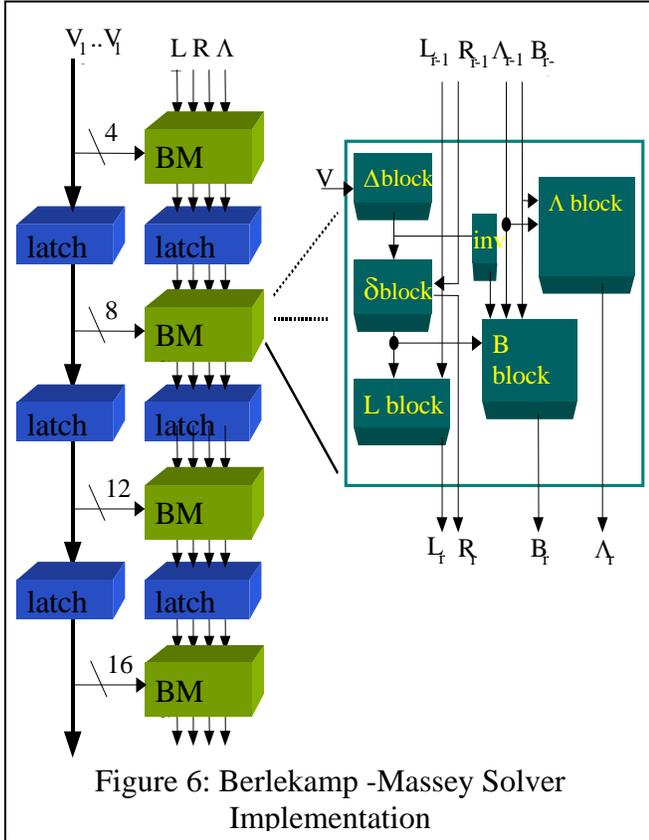


Figure 6: Berlekamp-Massey Solver Implementation

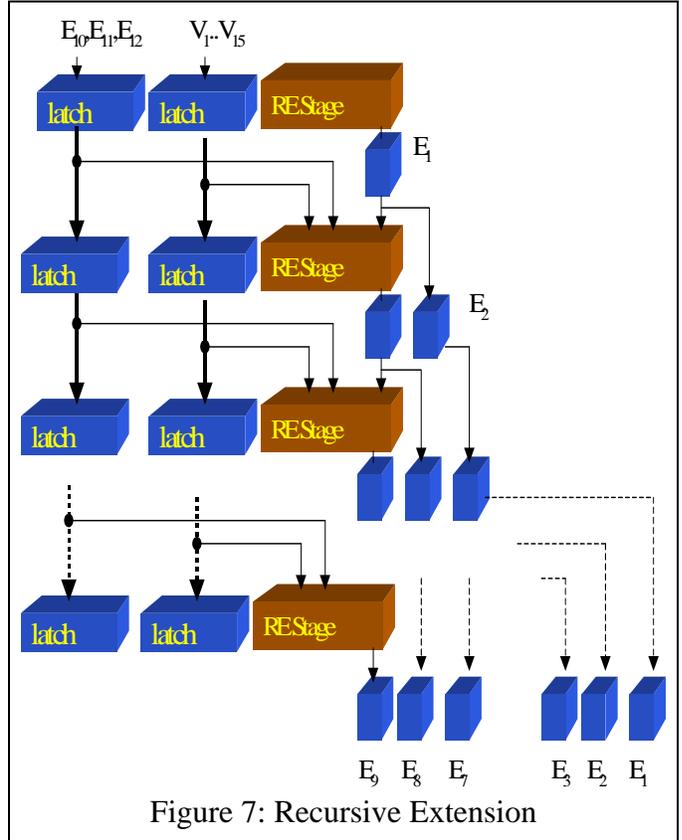


Figure 7: Recursive Extension

$B^r = (\Lambda^{r-1} / \Delta^r) \delta_r + (xB^{r-1})(1 - \delta_r)$. 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.

4.3 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 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. At the output of each stage, the error locator 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 .

5. PERFORMANCE DATA

Figure 8 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.

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

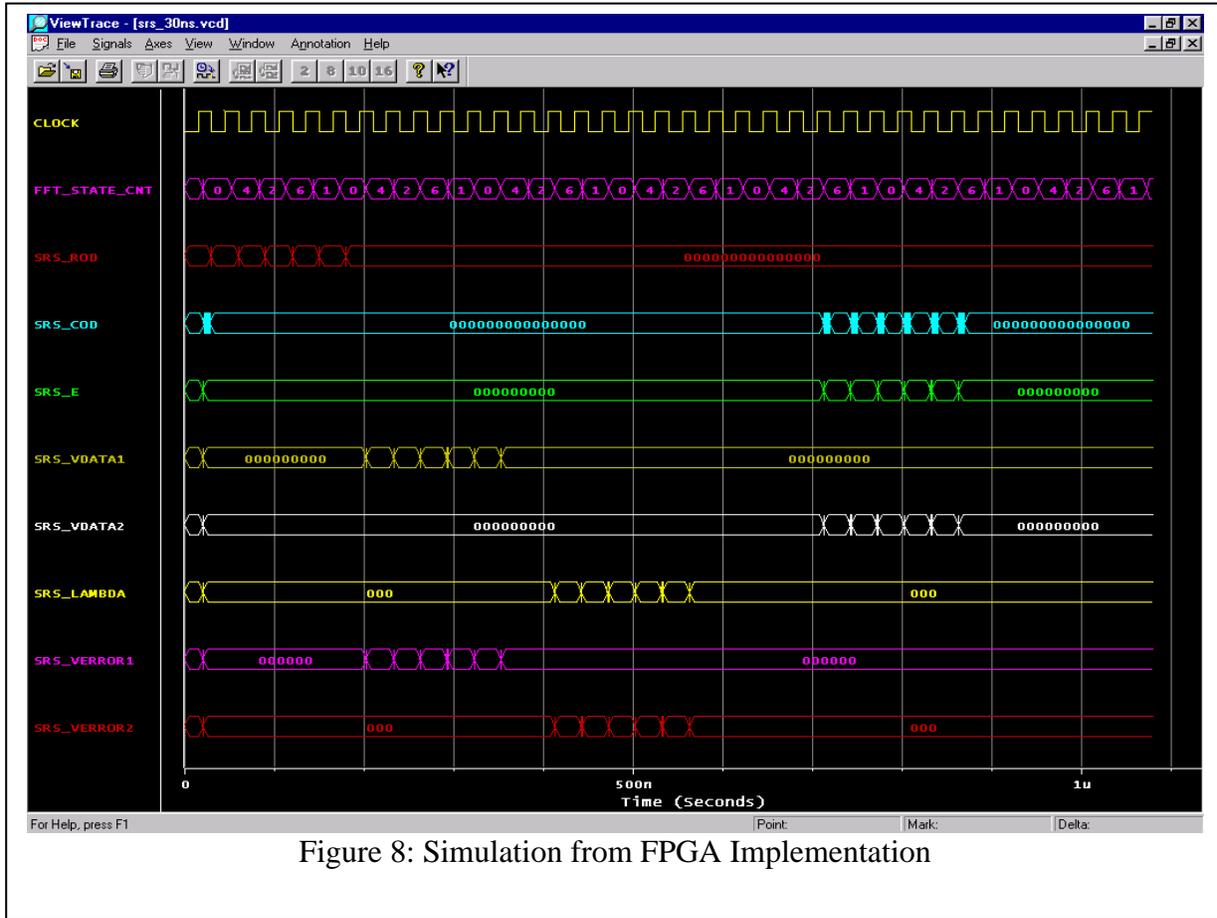


Figure 8: Simulation from FPGA Implementation

(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 9 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.

6. SUMMARY AND CONCLUSIONS

One of the most challenging issues in the design of an optical memory system interface is the error detection and correction mechanism. This paper 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 its 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 decode a 60-bit, (15,9), SRS code can be implemented in a single device and operating at a clock rate of 33 MHz.

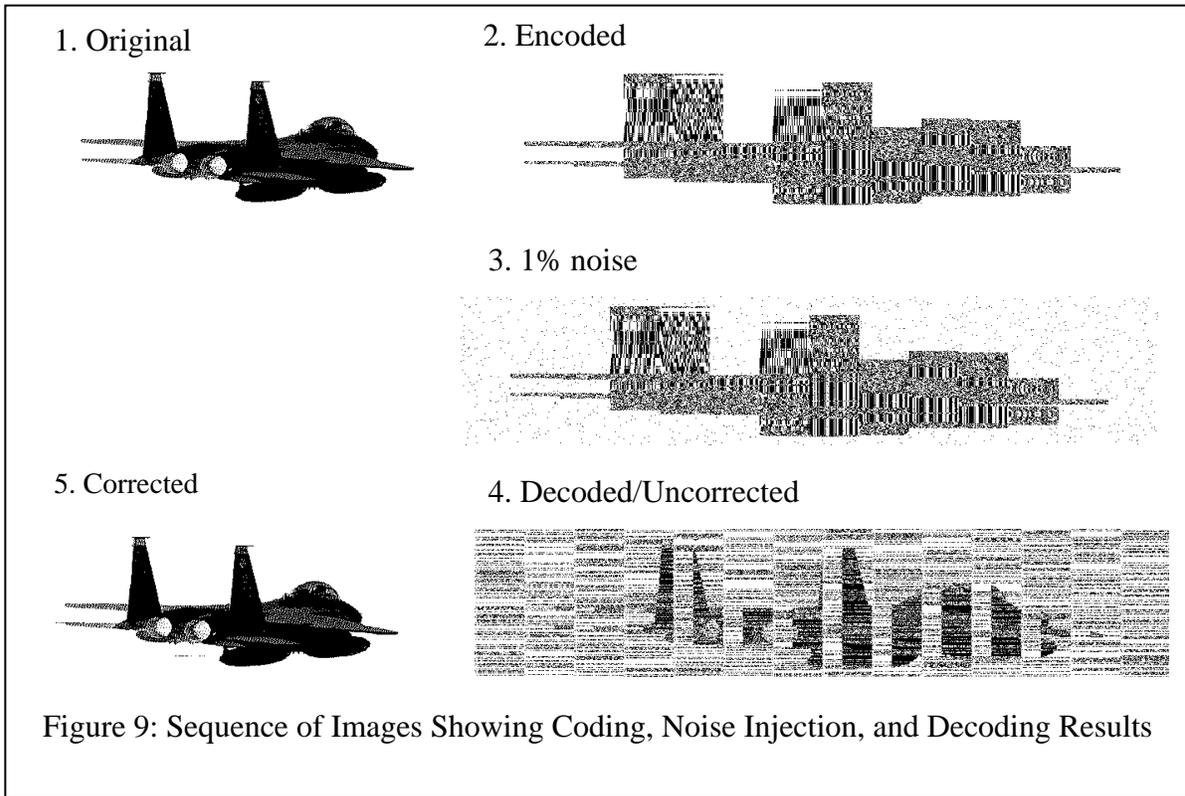


Figure 9: Sequence of Images Showing Coding, Noise Injection, and Decoding Results

7. REFERENCES

1. Donald M. Chiarulli and Steven Levitan. Optoelectronic Cache Memory System Architecture. *Applied Optics*, 35(14):2449-2456, May 1996.
2. G. Burr, F. H. Mok, and D. Psaltis. Storage of 10000 holograms in LiNbO₃: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.
3. 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.
4. I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *J. Soc. Ind. Appl. Math.*, 8:300-304, June 1960.
5. 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.
6. 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.
7. Po Tong. A 40-mhz Encoder-Decoder Chip Generated by a Reed-Solomon Code Compiler. In *IEEE Custom Integrated Circuits Conference*, 1990.
8. Richard E. Blahut. A Universal Reed-Solomon Decoder. *IBM J. Res. Develop.*, 28(2):150-158, March 1984.
9. Mark A. Neifeld and Jerry D. Hayes. Error-correction schemes for volume optical memories. *Applied Optics*, pages 8183-8191, December 1995.
10. Mark A. Neifeld and Satish K. Sridharan. Parallel Error Correction using Spectral Reed-Solomon Codes. *Journal of Optical Communications*, to be published.

