HARDWARE IMPLEMENTATION OF ERROR CONTROL DECODERS

by

BAINAN CHEN

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Adviser: Dr. Xinmiao Zhang

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August, 2008

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Bainan Chen

candidate for the Master of Science degree *.

(signed) Xinmiao Zhang
(chair of the committee)

Daniel Saab

Christos Papachristou

(date) 04/28/2008

*We also certify that written approval has been obtained for any proprietary material contained therein.

Table of Contents

List of Tables **v**

List of Figures **vi**

Acknowledgments **viii**

Abstract **ix**

Chapter 1. Introduction **1**

1.1 Overview 1

1.2 Summary of the Motivations and Contributions 1

1.2.1 FPGA Implementation of the Factorization Step in Soft-decision Reed-Solomon Decoding 1

1.2.2 Error Correction for Multilevel NAND Flash Memories Using Reed-Solomon Codes 3

1.3 Outline of the thesis 4

Chapter 2. FPGA Implementation of a Factorization Processor for Soft-decision Reed-Solomon Decoding **6**

2.1 Introduction 6

2.2 Algebraic Soft-Decision RS Decoding 9

2.3 Root-Order Prediction-Based Architecture 11

2.4 Coefficient Storage and Transfer for Parallel Polynomial Updating 16

2.5 Overall Factorization Architecture 22

2.6 Implementation Results 26

2.7 Summary 27

Chapter 3. FPGA Implementation of a Factorization Processor for Soft-decision Bit-level Generalized Minimum Distance Reed-Solomon Decoding 28
3.1 Introduction . . . 28
3.2 The BGMD Decoding Algorithm . . . 29
3.3 Factorization Architecture for BGMD Decoding . . . 31
3.4 Implementation Results . . . 39
3.5 Summary . . . 40

Chapter 4. FPGA Implementation and In-circuit Verification of the Factorization Processors 41
4.1 Introduction . . . 41
4.2 Overall Design Flow . . . 41
4.3 In-circuit Verification using FPGA Hardware Platform . . . 43
4.4 In-circuit Verification of the Factorization Processors . . . 45
4.5 Summary . . . 46

Chapter 5. Error Correction for Multilevel NAND Flash Memory using Reed-Solomon Codes 47
5.1 Introduction . . . 47
5.2 Flash Memory Models and Gray Mapping . . . 50
5.2.1 Modeling of Cell Threshold Voltage in Multi-level Flash Memory . . . 50
5.2.2 Gray Mapping and Detection . . . 51
5.3 Architectures for RS and BCH Decoders . . . 52
5.3.1 Syndrome Computation . . . 54
5.3.2 Key Equation Solver . . . 56
5.3.3 Chien Search and Forney's algorithm . . . 59
5.3.4 Computation Scheduling . . . 62
5.4 Hardware Complexity and Throughput Comparisons . . . 62
5.5 Summary . . . 66

Chapter 6. Conclusions and Future Research 67
6.1 Conclusions . . . 67
6.2 Future Research Directions . . . 68

Bibliography **70**

List of Tables

2.1 Resource Utilization on a Xilinx XC2V1000-6 FPGA 27

3.1 Resource Utilization on a Xilinx XC2V1000-6 FPGA 39

5.1 Mean and deviation for the threshold voltage levels in a 2-bit/cell MLC flash memory 50

5.2 Gate counts and critical paths for the building blocks in the (828, 820) RS decoder 63

5.3 Gate counts and critical paths for the building blocks in the (8246, 8192) BCH decoder 63

List of Figures

2.1 Algebraic soft-decision RS decoding architecture 9
2.2 An example of a root tree for the factorization 13
2.3 Detailed architectures of the RC3 unit 14
2.4 Detailed architectures of the exhaustive search unit 15
2.5 Detailed architectures of the MRC2 unit 16
2.6 Polynomial updating in the factorization 17
2.7 Detailed architectures of the PU unit 18
2.8 Coefficient storage and transfer scheme 19
2.9 Hardware architecture for the polynomial updating 21
2.10 The factorization architecture 23
2.11 Details of the top-level state machine for the factorization processor 25

3.1 Root trees for factorization with maximum Y-degree as two. . 31
3.2 Root computation for degree two polynomial (RC2) 33
3.3 The BGMD factorization architecture 34
3.4 Detailed architectures: a) RC b) PU 35
3.5 Transformations of polynomial coefficients in an iteration level with a second-order root 36
3.6 Details of the top-level state machine for the BGMD factorization processor 38

4.1 Overall design flow 42
4.2 Detailed architecture of the co-simulation 44
4.3 Detailed architecture of the high-speed prototyping 44

5.1 Cell threshold voltage distribution model for 2-bits/cell MLC flash memories 50
5.2 BER performance for 2-bit/cell MLC flash memories 51
5.3 RS/BCH decoder architecture 53

5.4 Syndrome computation: a) serial architecture b) p-parallel architecture 54
5.5 Ultra folded RiBM architecture 58
5.6 Architecture for Chien search and error magnitude computation 60

Acknowledgments

First and foremost, I wish to thank my advisor, Professor Xinmiao Zhang, for her guidance and support throughout my studies and research at Case Western Reserve University. I am very grateful for her recognition, inspiration, and the exposure and opportunities that I have received during the course of my study.

I would also like to thank Professor Christos Papachristou and Professor Daniel Saab for their support as members of my defense committee.

My thanks also go to the members of our group, particularly, Jiangli Zhu and Lan Yang for many useful discussions during the course of the work.

Hardware Implementation of Error Control Decoders

Abstract

by

BAINAN CHEN

In this thesis, an FPGA implementation of a factorization processor for algebraic soft-decision Reed-Solomon (RS) decoding is first presented. The design is based on the root-order prediction architecture and extensible for the factorization of polynomials with designated degrees. Parallel processing is exploited to speed up the polynomial updating involved in the factorization. To resolve the data dependency issue in parallel polynomial updating, an efficient coefficient storage and transfer scheme with small memory requirement and low latency is proposed. Synthesis results show that the factorization processor for a (255, 239) RS code with maximum multiplicity four can achieve an average decoding speed of 226 Mbps on a Xilinx Virtex-II FPGA device when the frame error rate is less than 10^{-2}.

Next, an FPGA implementation of a factorization processor for algebraic soft-decision bit-level generalized minimum distance (BGMD) RS decoding is presented. The BGMD factorization processor utilizes a low-latency and prediction-free scheme for root computation. Furthermore, parallel processing

architectures and efficient coefficient storage schemes are employed to reduce the latency. Synthesis results show that the BGMD factorization processor for a (255, 239) RS code with maximum multiplicity two can achieve a decoding speed of 815 Mbps on a Xilinx Virtex-II FPGA device.

Prior research efforts have been focusing on using BCH codes for error correction in multi-level cell (MLC) NAND flash memory. However, BCH codes often require highly parallel implementations to meet the throughput requirement. As a result, large area is needed. In this thesis, RS codes are proposed to be used for the error correction in MLC flash memory. A (828, 820) RS code has almost the same rate and length in terms of bits as a BCH (8248, 8192) code. Moreover, it has at least the same error-correcting performance in flash memory applications. Nevertheless, with 70% of the area, the RS decoder can achieve a throughput that is 121% higher than the BCH decoder. A novel bit mapping scheme using Gray code is also proposed. Compared to direct bit mapping, the proposed scheme can achieve 0.02 dB and 0.2dB additional gains by using RS and BCH codes, respectively, without any overhead.

Chapter 1

Introduction

1.1 Overview

Error-correcting codes (ECC) are widely used in digital communication systems and data storage to achieve highly reliable data transmission and storage. With the increasing demand for high-performance and low power systems, efficient hardware design and implementation of error-correcting codes become more and more important.

In this thesis, we will focus on the efficient hardware implementation and application of Reed-Solomon (RS) Codes. Two topics of the hardware implementation and application of RS codes are presented. These include the FPGA implementation of the factorization step for algebraic soft-decision (ASD) RS decoding and the application of RS codes in the error correction of multi-level cell (MLC) NAND flash memories.

1.2 Summary of the Motivations and Contributions

1.2.1 FPGA Implementation of the Factorization Step in Soft-decision Reed-Solomon Decoding

RS codes are among the most widely used block error-correcting codes in digital communication and storage systems. Recently, Guruswami and Su-

dan (GS) achieved a breakthrough in RS hard-decision decoding by using an algebraic interpolation technique [1, 2]. Later, the GS algorithm was extended to algebraic soft-decision decoding algorithms by incorporating the probability information from the channel into the interpolation process [3, 4]. With polynomial complexities, ASD algorithms can achieve substantial coding gain over hard-decision decoding.

Two major steps of ASD algorithms are the interpolation and factorization. In this thesis, we will focus on the high-speed FPGA implementation for the factorization step. We first present an FPGA implementation of a factorization processor for the algebraic soft-decision RS decoding. The factorization processor is based on the root-order prediction architecture, which is general and extensible. It can be easily extended to support the factorization of polynomials with any designated degree. Parallel processing is exploited to speed up the polynomial updating involved in the factorization. To resolve the data dependency issue in parallel polynomial updating, we propose an efficient coefficient storage and transfer scheme, which leads to smaller memory usage and low latency. Synthesis results show that the factorization processor for a (255, 239) RS code with maximum multiplicity four can achieve an average decoding speed of 226 Mbps on a Xilinx Virtex-II FPGA device when the frame error rate is less than 10^{-2}.

Next, we present the implementation of a factorization processor specific for the soft-decision bit-level generalized minimum distance (BGMD) RS decoding [4]. The BGMD factorization processor is based on the architecture

proposed in [5]. Among the ASD algorithms with practical multiplicity assignment schemes, the BGMD algorithm can achieve similar or higher coding gain with lower complexity. By exploiting the property that the maximum y-degree of the polynomials is low in the BGMD decoding algorithm, the BGMD factorization processor utilizes a low-latency and prediction-free root computation scheme to speed up the root computation in the factorization. Furthermore, efficient coefficient storage schemes and parallel processing architectures are employed to reduce the latency of the polynomial updating. Synthesis results show that the BGMD factorization processor for a (255, 239) RS code with maximum multiplicity two can achieve a throughput of 815Mbps.

Both of the factorization processors have been implemented and tested on the Prodesign CHIPit FPGA hardware platform.

1.2.2 Error Correction for Multilevel NAND Flash Memories Using Reed-Solomon Codes

The increasing demand for non-volatile multi-media and mass data storage has created the need for low-cost and high-capacity flash memories. In recent years, multi-level cell (MLC) flash memories [6–9] have been developed as an effective solution for increasing the storage density and reducing the cost of flash memories. In MLC flash memories, multiple bits are stored per memory cell by programming each cell with multiple threshold levels. The reduced spacing between adjacent threshold levels significantly degrades the reliability of flash memories. To ensure data integrity, error-correcting codes need to be

employed in these memories. Traditionally, single-bit error-correcting codes, such as Hamming codes, are used for single-bit cell (SBC) flash memories. For MLC flash memories, especially for NAND flash memories which have a large page size, e.g. 8K bits, single-bit error-correcting codes are no longer sufficient.

Prior research efforts have been focusing on using BCH codes for error correction in MLC NAND flash memories. However, BCH codes often require highly parallel implementations to meet the throughput requirement. As a result, large area is needed. In this thesis, we propose to use RS codes for error correction in MLC flash memory. A (828, 820) RS code has almost the same rate and length in terms of bits as a BCH (8248, 8192) code. Moreover, it has at least the same error-correcting performance in flash memory applications. Nevertheless, with 70% of the area, the RS decoder can achieve a throughput that is 121% higher than the BCH decoder. We also propose a novel bit mapping scheme using Gray code in this thesis. Compared to the direct bit mapping, our proposed scheme can achieve 0.02 dB and 0.2dB additional gain for the RS and BCH decoding, respectively, without any overhead.

1.3 Outline of the thesis

This thesis is organized as follows:

Chapter 2 is focused on the FPGA implementation for a factorization processor in algebraic soft-decision RS decoding. We first give a brief overview of the algebraic soft-decision RS decoding and the KV multiplicity assignment scheme. Next, details of the factorization step in algebraic soft-decision RS

decoding and the root-order prediction-based factorization architecture are introduced. Then the proposed coefficient storage and transfer scheme, and the overall hardware architecture are presented. Lastly, synthesis and implementation results for the proposed design are given.

Chapter 3 is focused on the FPGA implementation for the BGMD factorization processor in algebraic soft-decision RS decoding. We first give a brief overview of the BGMD multiplicity assignment scheme. Then we introduce the low-latency and prediction-free root computation scheme and the overall hardware architecture. Lastly, synthesis and implementation results for the BGMD factorization processor are given.

Chapter 4 gives a brief introduction of the overall FPGA design flow and the detailed setup of in-circuit verification for the two factorization processors. The verification results are also presented.

Chapter 5 is concerned with the application of RS codes in the error correction of MLC NAND flash memories. A Gray code-based mapping scheme is presented firstly as well as the simulation steps and modeling for MLC NAND flash memories. Next, the hardware complexity and throughput of decoding BCH and RS codes with similar code rate and bit length are compared. In addition, detailed decoder design with a (828, 820) RS code and a (8248, 8192) BCH code are presented as examples.

Chapter 6 provides the conclusion and some ideas for the future research.

Chapter 2

FPGA Implementation of a Factorization Processor for Soft-decision Reed-Solomon Decoding

2.1 Introduction

Reed-Solomon (RS) codes are popular block error-correcting codes with a wide range of applications in digital communication and storage systems. For an (n, k) RS code, traditional hard-decision algorithm, such as the Berlekamp-Massey algorithm [10, 11] and Euclid algorithms [12], can only correct up to $t = \lfloor d_{min}/2 \rfloor$ symbol errors, where $d_{min} = n - k + 1$ is the minimum distance of the code. Recently, Guruswami and Sudan (GS) achieved a breakthrough in RS hard-decision decoding by using an algebraic interpolation technique [1, 2]. The GS algorithm can correct up to $n - \sqrt{kn}$ errors. Later, the GS algorithm was extended to algebraic soft-decision decoding (ASD) algorithms by incorporating the probability information from the channel into the interpolation process [3, 4]. With polynomial complexities, ASD algorithms can achieve substantial coding gain over hard-decision decoding.

Two major steps of ASD algorithms are the interpolation and factorization. We will focus on the factorization step in this chapter. Among the various factorization algorithms [13–16], the algorithm proposed by Roth and

Ruckenstein [15] is the most efficient to solve the factorization problem. Based on this algorithm, several factorization architectures [17–20] were proposed for the hardware implementation. Each iteration of this algorithm mainly consists of root computation over finite field and polynomial updating. In [17, 18], exhaustive search is used for the root computation. This approach is very time-consuming, especially for RS codes defined over finite fields of large order. In [19], the polynomials are first transformed to affine polynomials. Then the roots of an affine polynomial are found by solving a set of linear equations. However, this approach is only applicable for polynomials with degree less than five and still has long latency. For the applications where high-speed decoding is desired, the root-order prediction-based architecture [20] can be employed. In this architecture, if the prediction is correct, the corresponding root computation in the next iteration level only involves one inversion, one multiplication and one fractional power over finite field. Root prediction failures only happen with less than 1% probability for the frame error rate (FER) in practical range. Thus, the average latency for the factorization is greatly reduced. This architecture is well-suited for high-speed applications where worse case delay is less concerned.

In this chapter, we present an FPGA implementation of a high-speed factorization processor based on the root-order prediction architecture proposed in [20]. The proposed factorization architecture can be easily extended to support the factorization of polynomial with any degree. Hence, it is a general architecture for the factorization in algebraic soft-decision RS decoding.

To speed up the factorization, all the polynomials in the same iteration level are updated in parallel. Additionally, in order to minimize latency, different degrees of parallelism are employed to the updating of the polynomials based on their degree. Due to the involved coefficient shifting, data dependencies exist among the polynomial coefficients during parallel polynomial updating. In this chapter, we propose an efficient coefficient storage and transfer scheme, which leads to smaller memory usage and low latency in the polynomial updating. Synthesis results show that the factorization processor for a (255, 239) RS code with maximum multiplicity four can achieve an average decoding speed of 226 Mbps when the FER is less than 10^{-2}. The proposed factorization processor is implemented and tested on the Prodesign CHIPit FPGA hardware platform. According to the best knowledge of the authors, this is the first FPGA implementation ever presented for the factorization in ASD decoding.

The rest of this chapter is organized as follows. Section 2.2 gives a brief introduction of the algebraic soft-decision RS decoding and the KV multiplicity assignment scheme. Section 2.3 briefly reviews the root-order prediction-based factorization architecture. Details of the coefficient storage and transfer scheme are described in Section 2.4. The overall architecture and FPGA implementation results are presented in Section 2.5 and 2.6 respectively. Section 2.7 provides the summary.

2.2 Algebraic Soft-Decision RS Decoding

Throughout this chapter, we consider an (n, k) RS code constructed over finite field $GF(2^q)$, where $n = 2^q - 1$ for primitive RS codes. For a (n, k) RS code, the k message symbols form a message polynomial $f(X) = f_0 + f_1 X + \cdots + f_{k-1} X^{k-1}$. The encoding is carried out by evaluating $f(X)$ at n distinct nonzero elements of $GF(2^q)$. Denote the n distinct evaluation elements as $\{\alpha_0, \alpha_1, \cdots, \alpha_{n-1}\}$, the corresponding codeword is $(f(\alpha_0), f(\alpha_1), \cdots, f(\alpha_{n-1}))$. Assume that after the jth symbol $f(\alpha_j)$ goes through the channel, θ_j is the noise-corrupted symbol observed at the receiver. Since the transmitted symbol is unknown to the receiver, it can be any field element over $GF(2^q)$. The algebraic soft-decision decoding algorithm attempts to recover the transmitted symbols (the message polynomial $f(X)$) by interpolation-based algorithms. The construction of the interpolation points is based on the reliability information observed from the channel.

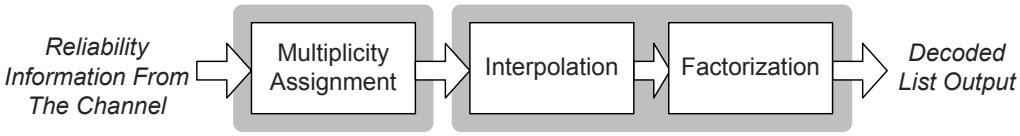

Figure 2.1: Algebraic soft-decision RS decoding architecture

Fig. 2.1 shows the overall architecture of the algebraic soft-decision RS decoding. The purpose of the multiplicity assignment is to construct a set of interpolation points based on the observation from the channel and assign each interpolation point with a 'multiplicity'. Later, the interpolation step

constructs a bivariate polynomial $Q(X, Y)$, with minimum $(1, k-1)$ weighted degree that passes all the interpolation points with the assigned multiplicities. If these points and multiplicities agree sufficiently well with the n points $(\alpha_j, f(\alpha_j))$ that define transmitted codeword, then the message polynomial $f(X)$ can be recovered in the factorization step by computing all factors of $Q(X, Y)$ in the form of $Y - f(X)$ with the degree of $f(X)$ less than k.

Various multiplicity assignment schemes have been proposed in recent years [3, 4, 18]. In the following, we brief introduce a modified version of the Koetter-Vardy (KV) multiplicity assignment scheme [18]. Denote all the distinct elements of $GF(2^q)$ by $\omega_0, \omega_1, \cdots, \omega_{2^q-1}$. Given the observation θ_j from the output of the memoryless channel, the *a posteriori* probabilities (APPs) for one of the 2^q possible transmitted symbols ω_i can be calculated as

$$\pi_{i,j} = Pr(\omega_i|\theta_j) \tag{2.1}$$

where $0 \leq i < 2^q$ and $0 \leq j < n$. There are $2^q \times n$ possible entries for $\pi_{i,j}$. All these $\pi_{i,j}$ form a $2^q \times n$ matrix called the *reliability matrix*, which represents the soft information from the channel. The multiplicity assignment step decides the interpolation point (α_j, ω_i) and its associated multiplicity $m_{i,j}$ according to the reliability matrix. All the multiplicities can be tabulated in a $2^q \times n$ *multiplicity matrix*, where the nonzero entries represent the nontrivial interpolation points and the associated multiplicities.

The original KV algorithm finds a multiplicity matrix that minimizes the decoding failure probability subject to a constraint of s, where s is the

total multiplicity. Nevertheless, the original KV algorithm is iterative and requires s passes of the reliability matrix. When it is applied to long RS codes with large s, the original KV algorithm has long latency. Alternatively, each entry of the multiplicity matrix can be generated by [18]

$$m_{i,j} = \lfloor \pi_{i,j} \times \lambda \rfloor \tag{2.2}$$

where λ is a positive number. This approach is based on the observation that for the KV algorithm, the multiplicity matrix is proportional to the reliability matrix when $s \to \infty$. It requires only one pass through the reliability matrix and can be easily implemented in hardware. In the rest of the chapter, the modified KV multiplicity assignment scheme is used. λ is set to 4.99 as a tradeoff between performance and hardware complexity. Therefore, the maximum multiplicity is four. Furthermore, it can be derived that the maximum y-degree of the bi-variate polynomial output from the interpolation is also four. It should be noted that although the algebraic soft-decision RS decoding with a maximum multiplicity of four is taken as an example, the architecture proposed in this chapter is general and can be easily extended to support other multiplicities.

2.3 Root-Order Prediction-Based Architecture

Assuming the bivariate polynomial output from the interpolation step is $Q(X, Y)$, the factorization determines all the factors of $Q(X, Y)$ in the form

of $Y - f(X)$ with $deg(f(X)) < k$. The algorithm in [15] can be described by the pseudo-code below.

Algorithm A: Factorization Algorithm

Initialization: $i = 0$

Reconstruct $(Q(X, Y), i)$ {

find the largest nonnegative integer v, such that $X^v | Q(X, Y)$

F1: $\bar{Q}(X, Y) = Q(X, Y)/X^v$

F2: find all the roots of $\bar{Q}(0, Y)$ in $GF(2^q)$

for each root γ of $\bar{Q}(0, Y)$, do

$\phi(i) = \gamma$

if $i = k - 1$

output $\Phi = [\phi(0), \phi(1), ..., \phi(k-1)]$

else

F3: $\tilde{Q}(X, Y) = \bar{Q}(X, Y + \gamma)$

F4: $\hat{Q}(X, Y) = \tilde{Q}(X, XY)$

call *Reconstruct* $(\hat{Q}(X, Y), i + 1)$

}

In Algorithm A, the k roots in each output vector, Φ, are the coefficients of a possible $f(X)$ polynomial. If the maximum Y-degree of $Q(X, Y)$ is t, then up to t vectors can be obtained from this algorithm. By employing re-encoding and coordinate transformation [21], the number of roots in each vector, and hence the number of iteration levels, can be reduced from k to 2τ in the factorization. Here τ is the number of errors intended to be corrected in the k

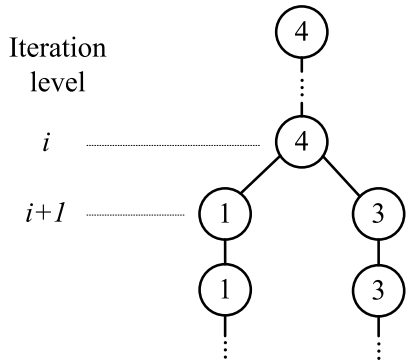

Figure 2.2: An example of a root tree for the factorization

most reliable positions. In this chapter, an FPGA implementation for a (255, 239) RS code with $t = 4$ and $2\tau = 26$ is presented as an example. The design can be easily extended to codes with different parameters.

The roots computed from Algorithm A form a tree structure. An example of the root tree is shown in Fig. 2.2 [22]. Each node represents a root with its order inside. The nodes in the same row correspond to the roots found in the same iteration level. In this example, $\bar{Q}(0, Y)$ has an order-4 root up to iteration level i. In iteration level $i+1$, two roots with order-1 and order-3 are found and the root split occurs. In [20], it is observed that the root split rarely happens and each root would most likely have a single descendant root of the same order. Based on this observation, a root-prediction scheme is proposed. In this scheme, it is first predicted that a degree-r $\bar{Q}(0, Y)$ has a single root, γ, of order r. Accordingly, γ can be computed as

$$\gamma = (q_{r-w} q_r^{-1})^{1/w}, \tag{2.3}$$

where q_i is the coefficient of X^i in $\bar{Q}(0, Y)$ and w is the minimum positive

integer such that $\binom{r}{r-w}$ is odd. It turns out that w can be only in the format of $2^e (e \in Z^*)$. In this case, $q^{1/w}$ can be derived by cyclically shifting all the bits in the normal basis representation of q, e, bit positions. Accordingly, the prediction-based root computation in (2.3) can be implemented by the RC3 unit shown in Fig. 2.3 [20]. In this figure, the inversion is implemented by a look-up table, and the Fractional Power block is used to implement the conversion to and from normal basis as well as the cyclical shift.

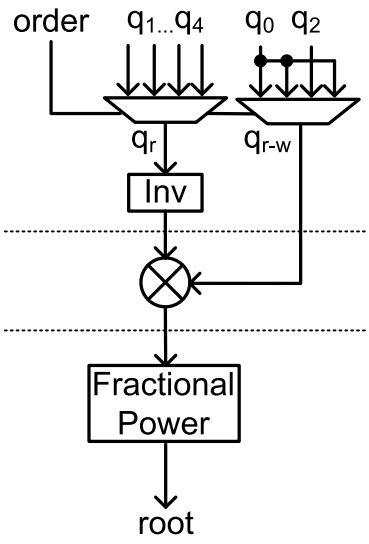

Figure 2.3: Detailed architectures of the RC3 unit

In the case that the prediction fails, if the degree of $\bar{Q}(0, Y)$ is larger than two, the roots are re-computed by the exhaustive search unit shown in Fig. 2.4. In this case, $\bar{Q}(0, Y)$ can be expressed as $q_4 X^4 + q_3 X^3 + q_2 X^2 + q_1 X + q_0$. In the exhaustive search unit, $\bar{Q}(0, Y)$ is decomposed to two polynomials $q_0 + q_1 X + q_4 X^4$ and $(q_2 + q_3 X) X^2$. At each clock cycle, these two polynomials

are evaluated with one of the 2^q field elements, respectively. When the evaluation results of the two polynomials are equal, the comparator (cmp) asserts the valid pin to indicate a root is found. The roots found from the exhaustive search output in parallel after a serial to parallel conversion (S2P).

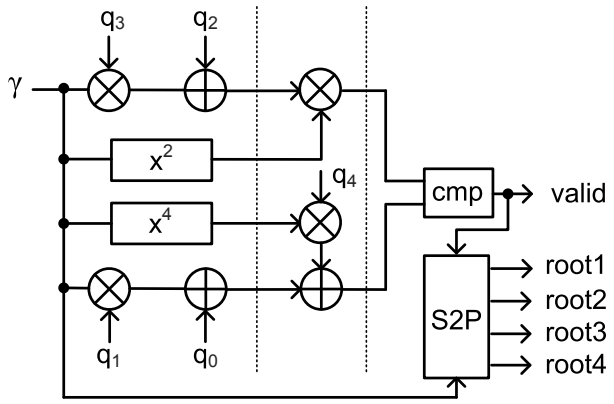

Figure 2.4: Detailed architectures of the exhaustive search unit

If the degree of $\bar{Q}(0, Y)$ is two, $\bar{Q}(0, Y)$ can be expressed as $q_2Y^2+q_1Y+q_0$. If $q_1 \neq 0$, the roots are re-computed by the MRC2 unit shown in Fig. 2.5 [20]. In this case, $q_2Y^2+q_1Y = q_0$ is transformed into the format of $Y'^2+Y' = q_2q_0(q_1^{-1})^2$ by substituting Y with $q_2^{-1}q_1Y'$. Such a degree two polynomial has two simple roots, if and only if the trace (Tr) function of $q_2q_0(q_1^{-1})^2$ is zero. The two roots can be computed by multiplying a pre-computed binary matrix with a vector formed by the standard basis representation of $q_2q_0(q_1^{-1})^2$. For detailed explanations of the MRC2 architecture, the interested reader is referred to [20].

It can be derived that for a simple root in iteration level i, the cor-

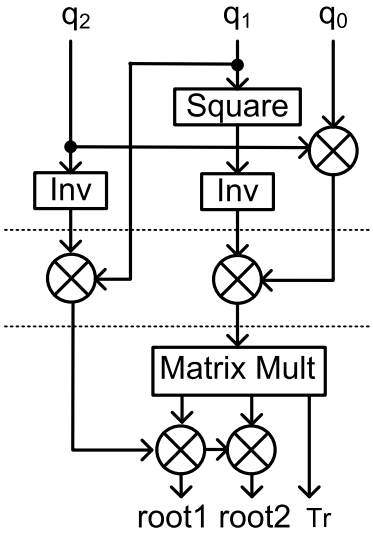

Figure 2.5: Detailed architectures of the MRC2 unit

responding $\bar{Q}(0, Y)$ in iteration level $i + 1$ is always of degree one and has a simple root. The root computation in this case can be also done by using the RC3 unit. For all the architecture addressed above (Fig. 2.3, 2.4, 2.5), pipelines shown by the dashed lines are inserted to reduce the length of critical paths of the computational units.

2.4 Coefficient Storage and Transfer for Parallel Polynomial Updating

The coefficients of a bivariate polynomial can be represented by dots in a two-dimensional array shown in Fig. 2.6 [20]. Each solid dot, hollow dot and cross represents a coefficient (zero or non-zero), a zero coefficient and a non-zero coefficient, respectively. All the dots in the same row correspond to

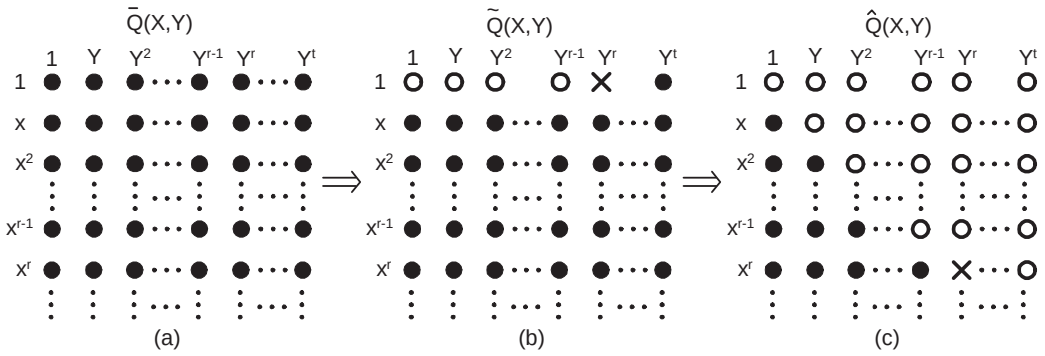

Figure 2.6: Polynomial updating in the factorization

the coefficients of the monomials with the same X-degree, while those in the same column correspond to the coefficients of the monomials with the same Y-degree. The dots in the first row of Fig. 2.6a form the coefficients of $\bar{Q}(0, Y)$. Assume an order-r root is found for $\bar{Q}(0, Y)$. After the F3 polynomial updating step in Algorithm A, the coefficients of $\tilde{Q}(0, Y)$ satisfy $\tilde{q}_{0,b} = 0 (0 \leq b < r)$ and $\tilde{q}_{0,r} \neq 0$. The F4 step shifts the jth column of $\tilde{Q}(X, Y)$ down by j positions. The coefficients in the first nonzero row of $\hat{Q}(X, Y)$ are those of $\bar{Q}(0, Y)$ in the next iteration level. If no root split occurs, the first r rows of the coefficients of $\hat{Q}(X, Y)$ are all zero and those in the row for X^r become the coefficients of the $\bar{Q}(0, Y)$ in the next iteration level [20].

In our design, the F3 step is implemented by the PU unit shown in Fig. 2.7 [22], which processes one row of coefficients at a time. To reduce complexity, polynomial updating only needs to be carried out on those coefficients involved in the computations in later iteration levels. It can be derived that in iteration level i, if a root of $\bar{Q}(0, Y)$ has order r, then only the first $r \times (2\tau - i)$

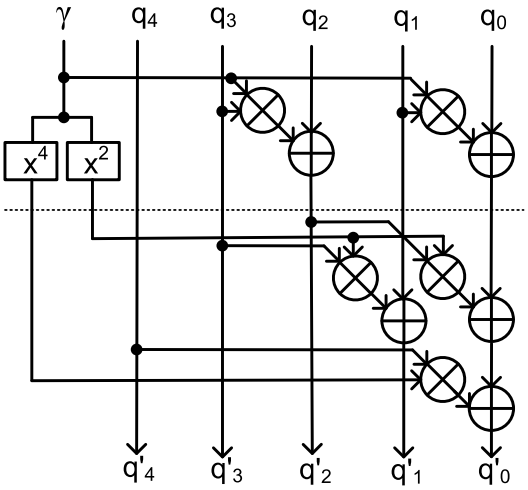

Figure 2.7: Detailed architectures of the PU unit

rows of $\bar{Q}(X, Y)$ need to be updated for this root. Hence, the roots with larger order require more coefficients to be updated. To balance the polynomial updating latency, for a root of order r, we divide the coefficients to be updated into r segments of consecutive X-degree and use r PU units to update these coefficients in parallel. For example, assume there is an order-3 root and an order-1 root in a iteration level. Then 3 PU units will be employed for the corresponding polynomial updating for the order-3 root, while the polynomial updating for the order-1 root only has one PU unit to use. Using this scheme, the polynomial updating corresponding to all the roots in a iteration level can be completed simultaneously. Since the sum of the root order in a iteration level never exceed t, t PUs would be enough for all polynomial updating.

Due to the polynomial shifting in the F4 step of Algorithm A, data dependencies exist among the coefficients with different X-degree. This leads to

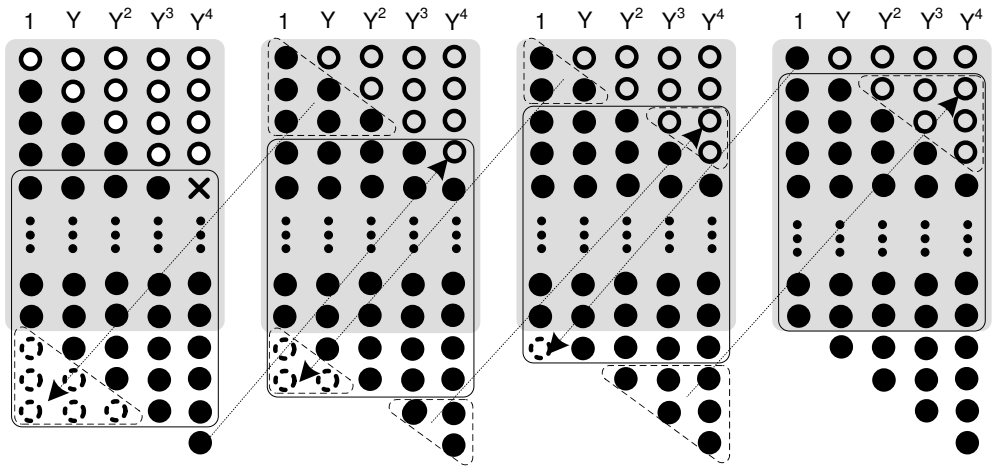

Figure 2.8: Coefficient storage and transfer scheme

data dependencies among the coefficients updated in different PU units corresponding to the same root. In this design, an efficient block coefficient storage and transfer scheme is proposed to solve this problem. Take the updating and shifting of a $\bar{Q}(X, Y)$ corresponding to an order-4 root as example, in iteration level i, the $4(2\tau - i)$ rows of coefficients are divided evenly and stored in four memory blocks shown as the shaded rectangles in Fig. 2.8 [22], with the first $2\tau - i$ rows stored in the leftmost block. After the polynomial shifting, the coefficients in each block form a parallelogram. Instead of moving the coefficients, the polynomial shifting step can be implemented by keeping track of address displacements. The coefficients in the extra last four rows generated by the shifting have the same X-degrees as the coefficients left in the first four rows of the next adjacent memory block. Hence, these coefficients need to be combined before they are fed to the PU units in the next iteration level. In addition, in the case that the corresponding $\bar{Q}(X, Y)$ in the next iteration level

still has a single root of order 4, the first four rows of coefficients in the first memory block are zero and $4(2\tau - i - 1)$ rows of coefficients need to be stored for the computations in later iterations. To minimize polynomial updating latency, the $4(2\tau - i - 1)$ coefficients also need to be divided evenly into four segments such that the same number of coefficients are updated in each PU unit. Due to these two reasons, the coefficients need to be transferred among the memory blocks as shown by the arrows in Fig. 2.8. The coefficients in the solid rectangles are those need to be stored in each memory for the next iteration level. As a result of these transformations, all coefficients of the same X-degree are put into the same row and each memory block has the same number of coefficients. The coefficient transfer patterns for other root orders can be derived in a similar way. In this scheme, we need to store at most $t \times 2\tau$ rows of coefficients in total, which is only $1/t$ of the memory requirement of the architectures proposed in [18, 19].

When the root orders do not change, the coefficients from each of the four memory blocks can be connected directly to PU units for polynomial updating. However, when root splits happen, extra care needs to be taken for the routing of the coefficients. Fig. 2.9 [22] shows an example of such a case. In this figure, the four memory blocks are denoted by M_0 to M_3. The RD_MUX is used to route the coefficients of $\bar{Q}(X, Y)$ to multiple PUs, and the WR_MUX is used for coefficient transfer between adjacent memories. Separate multiplexors are needed for each column of coefficients since the routings for each column is different. Assume in iteration level i, the $\bar{Q}(X, Y)$ corresponding to an order-3

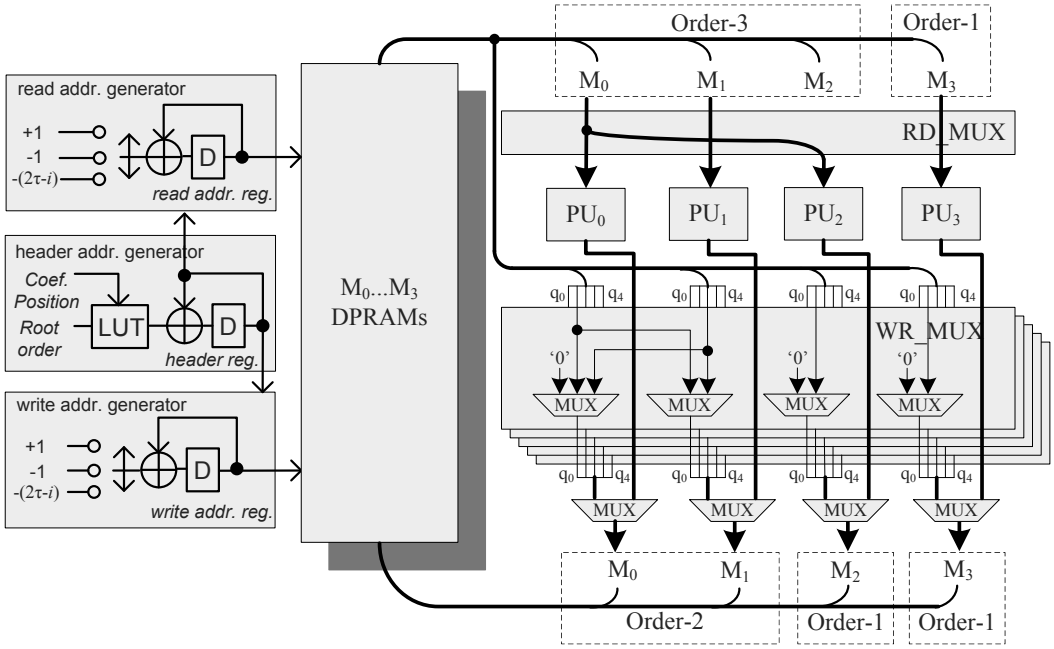

Figure 2.9: Hardware architecture for the polynomial updating

root in the previous level is stored in M_0 through M_2, and another $\bar{Q}(X, Y)$ corresponding to an order-1 root in the previous iteration level is stored in M_3. Assume that the order-3 root splits into one order-2 and one order-1 roots. Since the number of coefficients need to be updated for the order-2 root has been decreased to $2 \times (2\tau - i)$, only the coefficients in M_0 and M_1 need to be updated using the newly computed order-2 root, while the coefficients in M_2 can be discarded. Then the coefficients updated by PU_0 and PU_1 are written back into M_0 and M_1, respectively. For the order-1 root from split, the corresponding polynomial updating is only carried out on the coefficients in M_0 by PU_2. Then the updated coefficients are written back into the empty M_2 memory block. There is no root order change for the other $\bar{Q}(X, Y)$ with

order-1 root. Hence the coefficients are fed to PU_3 directly, and the updated coefficients are written back to M_3.

The detailed architectures of the address generators are also shown in Fig. 2.9. The header address generators are used to generate the starting position for each column of the memory blocks. At the beginning of each iteration level, the header addresses are loaded into both the read and write address generators. Both of these addresses are increased by one at each clock cycle until $2\tau - i$ rows of coefficients are read, updated and written back for iteration level i. After that, these address are adjusted to transfer the coefficients among memory blocks as shown in Fig. 2.8. The coefficient memory is configured as a circular memory so that the coefficients are wrapped around if they reach the memory boundary.

2.5 Overall Factorization Architecture

Fig. 2.10 [22] shows the overall architecture of the factorization processor. In this architecture, each memory block is divided into five banks and each bank stores the coefficients with the same Y-degree. Hence, a total of twenty dual-port memories (DPRAMs) are used. To simplify the address generation for the circular memory, the depth of each memory is chosen as the nearest power-of-two ceiling of 2τ, i.e. 32 bytes.

At the beginning of the factorization, the zero rows of $Q(X, Y)$ are taken out and the rest of the coefficients are divided evenly into four blocks and loaded into M_0 through M_3. In iteration level $i = 0$, $\bar{Q}(0, Y)$ is read

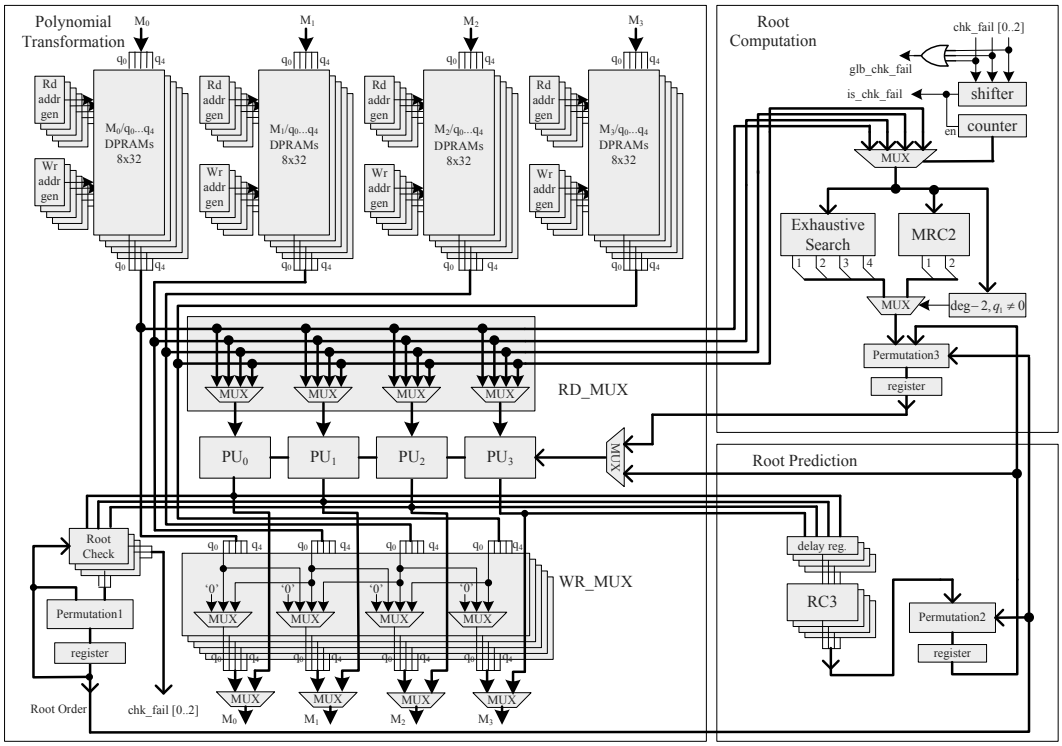

Figure 2.10: The factorization architecture

out from the first row of M_0 and fed to the Exhaustive Search unit for root computation. According to the root order definition, a root γ is a rth order root of $p(X)$ if and only if the coefficients of the terms with X-degree lower than r in $p(X+\gamma)$ are all zero, and the one for X^r is nonzero. Therefore, by checking the coefficients at the output of the PU units, the order of each computed root can be determined. In our design, the polynomial updating corresponding to a root with order r is carried out simultaneously in r PU units. In order to make the roots aligned with the parallel polynomial updating in multiple PU units, permutation blocks are added. Another function of the Root Check block is to decide which is the first nonzero row of $\hat{Q}(X,Y)$. This will be the $\bar{Q}(0,Y)$ in the next iteration level. Once the first nonzero row is decided, the root prediction for the next iteration level starts and is performed concurrently with the polynomial updating of the rest of the coefficients. Due to the polynomial shifting, the coefficients of the first nonzero row of $\hat{Q}(X,Y)$ come from the outputs of the PU units in different clock cycles. Hence, triangular arrays of registers are inserted between the PU and RC3 units.

From the second iteration level, the real orders of the roots computed through prediction are derived in the Root Check block and they are compared with the corresponding root orders in the previous iteration level. If they do not match, a prediction failure happens. The root is re-computed in either the MRC2 or the Exhaustive Search block when the degree of the failed polynomial is two or larger than two, respectively. Then polynomial updating is carried out based on the re-computed roots.

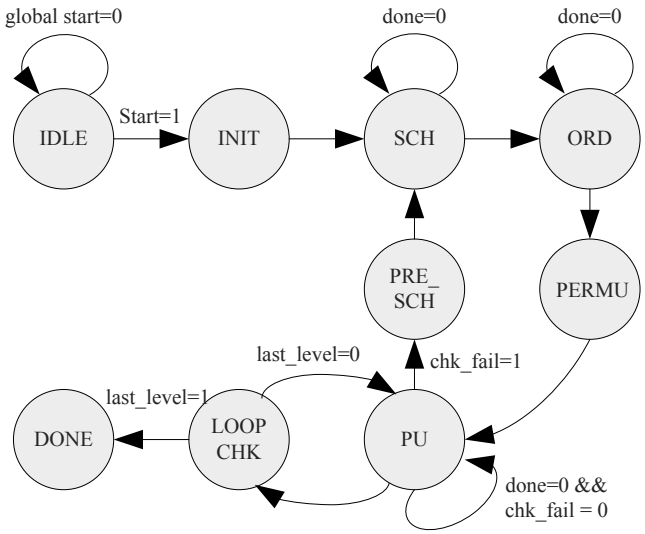

Figure 2.11: Details of the top-level state machine for the factorization processor

A hierarchical control structure is used in the factorization processor. Each computational unit has its state machine locally, which interacts with the top-level state machine through proper handshake signals. The details of the top-level state machine are shown in Fig. 2.11. The 'done' signals refer to the handshake signals from various computational units, while the 'start' signal for each computational unit is not explicitly presented. After power up, the factorization processor remains in the 'IDLE' state until the 'global start' pin is asserted. Then the internal registers are properly initialized in the 'INIT' state. Next, the exhaustive search or direct root computation is invoked in the 'SCH' state based on the degree of $\bar{Q}(0, Y)$. Once the root computation is completed, the root orders are determined in the 'ORD' state and permuted

in the 'PERMU' state. Polynomial updating and shifting are carried out in the state 'PU'. Meanwhile, root prediction for the next iteration level using the RC3 units is performed in parallel. If it is not the first iteration level, the roots predicted from the previous iteration level will be used and checked during the polynomial updating. If the predicted root order is not equal the actual root order, the 'chk_fail' signal asserts and the state is transitioned to the 'SCH' state for root search and computation. Otherwise, the state machine will check the number of loops that has run in the 'LOOP_CHK' state and decide whether it continues to the next iteration level or all the iteration levels are done and the factorization terminates. The 'PRE_SCH' state is used to insert some clock cycles to avoid memory access conflict.

2.6 Implementation Results

The factorization processor for a (255, 239) RS code with $t = 4$ was synthesized and simulated on a Xilinx Virtex-II XC2V1000-6 device. After place and route, the Xilinx ISE software reported a critical path of 9.822ns. The logic resource utilization is shown in Table 2.1 [22]. All the coefficient memories are implemented using the Xilinx distributed RAMs. Simulation results indicate that when root prediction is successful in all the iteration levels, the decoding latency is 914 clock cycles. In the case that root prediction failure happens in one iteration level, if exhaustive search is used, 1194 clock cycles are needed, if MRC2 is used, 938 clock cycles are needed. From the simulation for a (255, 239) RS code over additive white Gaussian noise (AWGN) channel,

we observed that the root prediction failure rate is less than 3×10^{-3} when the FER is less than 10^{-2}. Hence, the average decoding latency is 915 clock cycles and the decoding throughput is 226Mbps. Higher speed can be achieved by using state-of-art ASIC technology. The details of the in-circuit verification setup and results for the factorization processor will be addressed in Chapter 4.

Table 2.1: Resource Utilization on a Xilinx XC2V1000-6 FPGA

Resource Type	**Used**	**Ratio**
Slice Flip Flops	1,795	17%
4 input LUTs	6,703	65%
Block RAMs	0	0%
Total Occupied Sliced	4,195	81%

2.7 Summary

In this chapter, an FPGA implementation of a high-speed factorization processor for algebraic soft-decision RS decoding is presented. Parallel processing is employed to speed up the polynomial updating. In addition, efficient coefficient storage and transfer scheme is developed to facilitate the parallel processing and minimize the latency. This is the first hardware implementation ever presented for the factorization in the literature.

Chapter 3

FPGA Implementation of a Factorization Processor for Soft-decision Bit-level Generalized Minimum Distance Reed-Solomon Decoding

3.1 Introduction

In the previous chapter, an FPGA implementation of a factorization processor for the soft-decision Reed-Solomon decoding is presented. The design is based on the root-order prediction architecture and can be easily extended to support factorization of polynomials with designated degrees. Since the root prediction failures only happen with less than 1% probability for the FER in practical range, the average latency for the factorization is greatly reduced. However, when prediction failure happens, additional clock cycles are required to recompute the roots, which leads to extra latency.

In this chapter, we present an FPGA implementation of a factorization processor specific for soft-decision bit-level generalized minimum distance (BGMD) [4] RS decoding. The BGMD algorithm is a practical multiplicity assignment scheme in algebraic soft-decision RS decoding. Compared to the conventional multiplicity assignment scheme, such as the Koetter-Vardy(KV) scheme [3], it can achieve similar or higher coding gain with lower multiplic-

ities. As a result, the maximum Y-degree of the polynomials is low in the BGMD algorithm. Hence, the root-order prediction scheme for root computation can be replaced by a prediction-free and low latency root computation scheme. Furthermore, resource sharing techniques are employed to minimize the area requirement of the factorization architecture. The BGMD factorization processor is based on the architecture proposed in [5]. Synthesis results show that the BGMD factorization processor for a (255, 239) RS code can achieve a decoding throughput of 815 Mbps with low complexity.

The rest of this chapter is organized as follows. Section 3.2 briefly reviews the BGMD multiplicity assign scheme. Details of the factorization architecture for BGMD decoder are described in Section 3.3. The FPGA implementation results are presented in Section 3.4. Section 3.5 provides the summary.

3.2 The BGMD Decoding Algorithm

Throughout this chapter, we consider an (n, k) RS code constructed over finite field $GF(2^q)$, where $n = 2^q - 1$ for primitive RS codes. For a (n, k) RS code, the k message symbols form a message polynomial $f(X) = f_0 + f_1 X + \cdots + f_{k-1} X^{k-1}$. The encoding is carried out by evaluating $f(X)$ at n distinct nonzero elements of $GF(2^q)$. Denote the n distinct evaluation elements as $\{\alpha_0, \alpha_1, \cdots, \alpha_{n-1}\}$, the corresponding codeword is $(f(\alpha_0), f(\alpha_1), \cdots, f(\alpha_{n-1}))$. Assume that for the jth transmitted symbol $f(\alpha_j)$, θ_j is the noise-corrupted symbol observed at the receiver. Since the transmitted symbol is unknown to

the receiver, it can be any field element over $GF(2^q)$.

In the one-pass BGMD multiplicity assignment scheme, the multiplicities are assigned based on the bit-level reliability. The reliability of a bit out of a q-bit symbol after the channel can be calculated by the log-likelihood ratios $LLR = log\frac{P(0|r)}{P(1|r)}$, where r is the soft information from the channel. The received bit is considered to be erased if its $|LLR|$ is below a certain threshold. Denote the maximum multiplicity is m_{max}, the one-pass BGMD algorithm assigns multiplicities based on the number of bits erased in each received symbol. The detailed algorithm is shown below:

Algorithm A: BGMD Multiplicity Assignment

1) if no bit is erased in θ_j, assign m_{max} to (α_j, β_j), where β_j is the hard-decision of θ_j;

2) if there is only one bit erased in θ_j, assign $m_{max}/2$ to both (α_j, β_{j1}) and (α_j, β_{j2}), where β_{j1} and β_{j2} are the hard-decision of θ_j and the field element differs from the hard-decision in only the erased bit, respectively;

3) if there are more than one bit erased in θ_j, do not assign any multiplicity to the interpolation points with α_j.

It can be observed that for an evaluation element α_j: if no bit is erased in the jth received symbol, only one interpolation point with a multiplicity of 2 corresponds to α_j; if one bit is erased, then two possible interpolation points with multiplicity 1 correspond to α_j; in other cases, no interpolation point corresponds to α_j. Simulation results show that the one-pass BGMD

algorithm with maximum multiplicity two can achieve similar or higher coding gain than the KV algorithm with maximum multiplicity four. Accordingly, the maximum Y-degree of the polynomials from the interpolation output is 2 in this case.

3.3 Factorization Architecture for BGMD Decoding

In this section, the detailed architecture of a BGMD factorization processor for a (255, 239) RS code with $m_{max} = 2$ is presented. The factorization algorithm proposed by Roth and Ruckenstein [15] is employed in the BGMD factorization processor. The detailed of the algorithm is addressed in previous chapter. The maximum Y-degree is set to 2 in this case. By employing re-encoding and coordinate transformation [21], the number of the iteration levels can be reduced from k to 2τ in the factorization. Here τ is the number of errors intended to be corrected in the k most reliable positions. In the FPGA implementation, τ is set to 10 for a (255, 239) RS code.

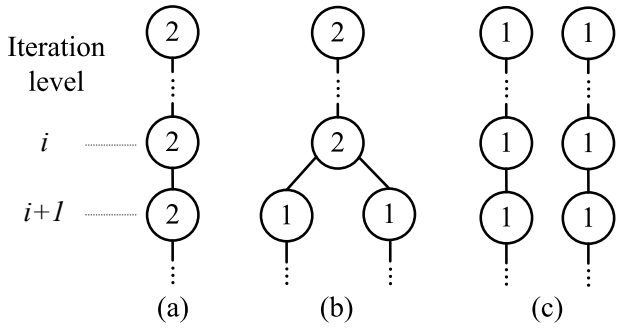

Figure 3.1: Root trees for factorization with maximum Y-degree as two.

The roots computed from the factorization algorithm form a tree structure. Taking the polynomial from the interpolation with maximum Y-degree of 2 as an example, the root tree can be one of the three formats shown in Fig. 3.1 [5]. Each node represents a root with its order inside. The nodes in the same row correspond to the roots found in the same iteration level. If in iteration level i, $\bar{Q}(0, Y)$ has one root of order two, then the corresponding $\bar{Q}(0, Y)$ may most likely have one root of order two in iteration level $i + 1$, as illustrated in Fig. 3.1a. Alternatively, $\bar{Q}(0, Y)$ may have two simple roots in iteration level $i+1$ as illustrated in Fig. 3.1b. In this case, the root tree splits. If in iteration level i, each $\bar{Q}(0, Y)$ has a simple root shown in Fig. 3.1c, it can be derived that the corresponding $\bar{Q}(0, Y)$ in iteration level $i + 1$ is always of degree one and has a simple root.

In summary, $\bar{Q}(0, Y)$ can have three types of roots in each iteration: one order-2 root, two simple roots or one simple root. The root computation of $\bar{Q}(0, Y)$ can be carried out using the RC2 architecture in Fig. 3.2 [5]. When the degree of $\bar{Q}(0, Y)$ is two, $\bar{Q}(0, Y)$ can be expressed as $q_2Y^2 + q_1Y + q_0$. If $q_1 \neq 0$, the two multiplexors in the left shaded rectangles choose q_2 and q_1 as the output. In this case, the architecture in Fig. 3.2 is similar to the MRC2 architecture in the previous chapter. If the trace (Tr) function of $q_2q_0(q_1^{-1})^2$ is zero, the two roots *roota* and *rootb* are output from the block. In addition, observing that the difference between the two roots is the identity element of the finite field, the multiplication of $q_2^{-1}q_1$ back to derive the roots of $\bar{Q}(0, Y)$ can be simplified by using only one multiplier and one adder. In the case

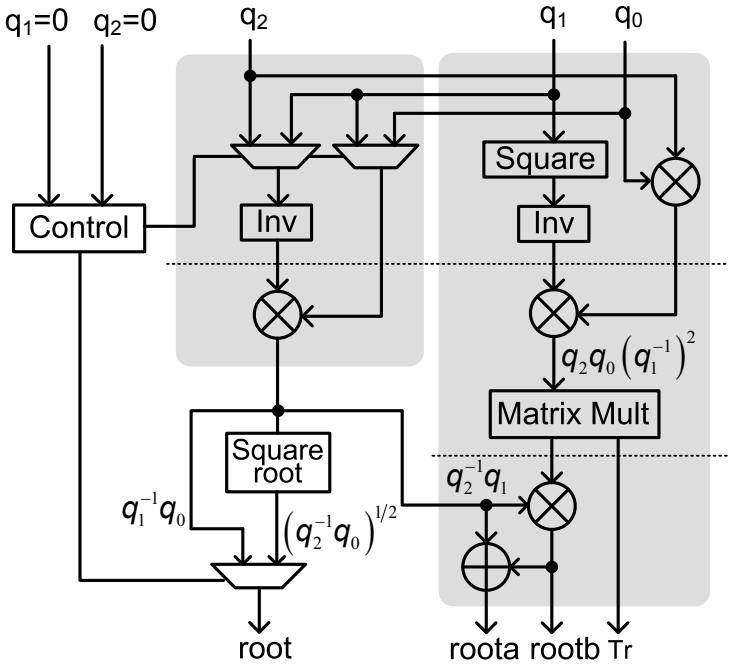

Figure 3.2: Root computation for degree two polynomial (RC2)

that $q_1 = 0$, q_2 and q_0 are selected from the multiplexors in the left shaded rectangles. The order-2 root of $q_2 Y^2 + q_0$ is $(q_2^{-1} q_0)^{1/2}$ and computed by the computational units in the left shaded rectangles and the square root block. When the degree of $\bar{Q}(0, Y)$ is one, i.e. $q_2 = 0$ and $q_1 \neq 0$, the root, $q_1^{-1} q_0$, can be computed from the computational units in the left shaded rectangles by choosing q_1 and q_0 from the multiplexors. For the detailed explanations of the RC2 architecture, the interested reader is referred to [5]. To reduce the latency, the inverter is implemented by a look-up table. The square root block is implemented by a conversion to and from normal basis as well as the cyclical shift. In Fig. 3.2, pipelines shown by the dashed lines are inserted to reduce

the length of critical paths of the computational units.

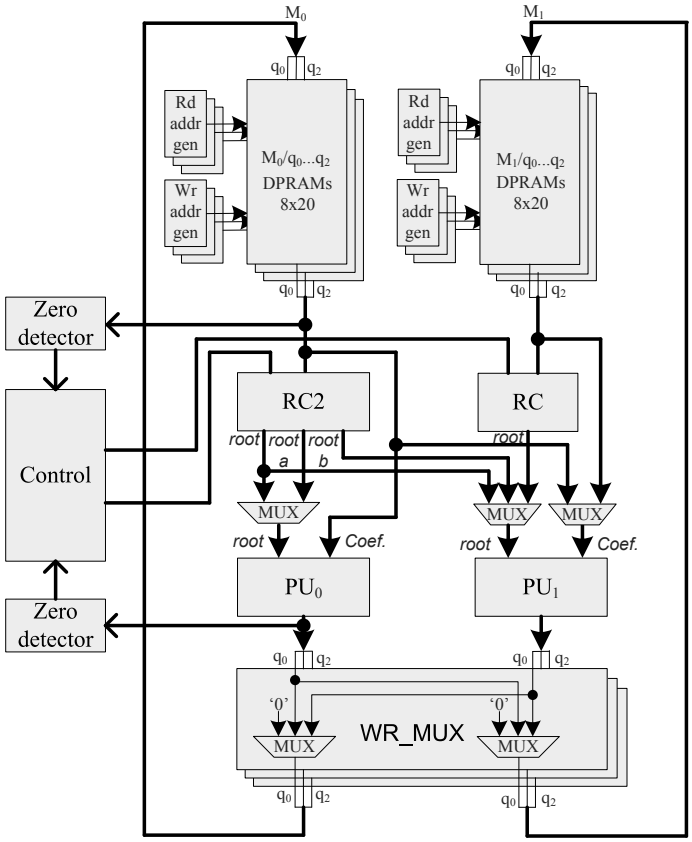

Figure 3.3: The BGMD factorization architecture

Fig. 3.3 [5] shows the overall architecture of the BGMD factorization processor. In this architecture, M_0 and M_1 are the memory blocks for storing polynomial coefficients. Each memory block is divided into three banks and each bank stores the coefficients with the same Y-degree. Hence, a total of six dual-port memories (DPRAMs) are used. The RC unit is only used for the root computation of degree one polynomials. The detailed architecture

of the RC unit is shown in Fig. 3.4a. The F3 polynomial updating step is implemented by the PU units shown in Fig. 3.4b. The F4 and F1 steps are implemented by proper address displacements from the read and write address generators.

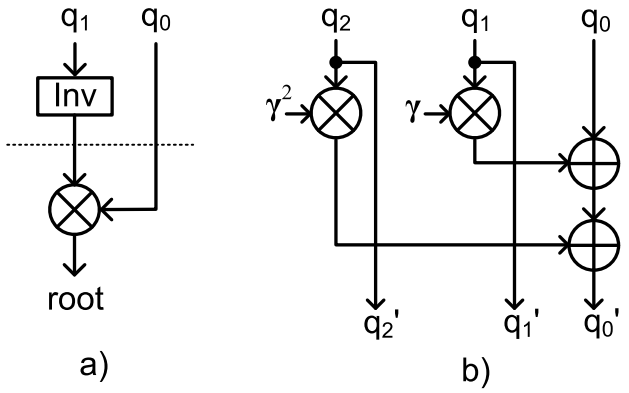

Figure 3.4: Detailed architectures: a) RC b) PU

A coefficient storage and transfer scheme similar to the one described in previous chapter is employed for the BGMD factorization processor. Assume that in iteration level i, $\bar{Q}(X,Y)$ has an order-r root, then only the coefficients of $\bar{Q}(X,Y)$ with X-degree up to $r \times (2\tau - i)$ need to be updated for this root. These coefficients are divided into r segments of consecutive X-degree and updated in r PU units. If $\bar{Q}(0,Y)$ has a single root of order two, in iteration level i, the coefficients of $\bar{Q}(X,Y)$ corresponding to the lower and higher $2\tau - i$ X-degree are stored into M_0 and M_1 respectively, as shown in Fig. 3.5 [5]. After polynomial updating, the coefficient corresponding to the term X in $\hat{Q}(X,Y)$ is tested. If it is nonzero, then there is no root in the

next iteration level and the factorization stops. Otherwise, the third row in $\hat{Q}(X,Y)$ coefficient array is first nonzero row and it becomes the first row of the $\bar{Q}(X,Y)$ coefficient array in the next iteration level. Two coefficients need to be transferred so that both M_0 and M_1 have an equal number of coefficients for the updating in the next iteration level. The coefficient transfer can be done during the write-back of the updated coefficients through the WR_MUX in Fig. 3.3. If $\bar{Q}(0,Y)$ is a degree one polynomial, $\bar{Q}(0,Y)$ has a simple root, the coefficients of $\bar{Q}(0,Y)$ occupy one of the M_0 and M_1 and are updated in one PU unit.

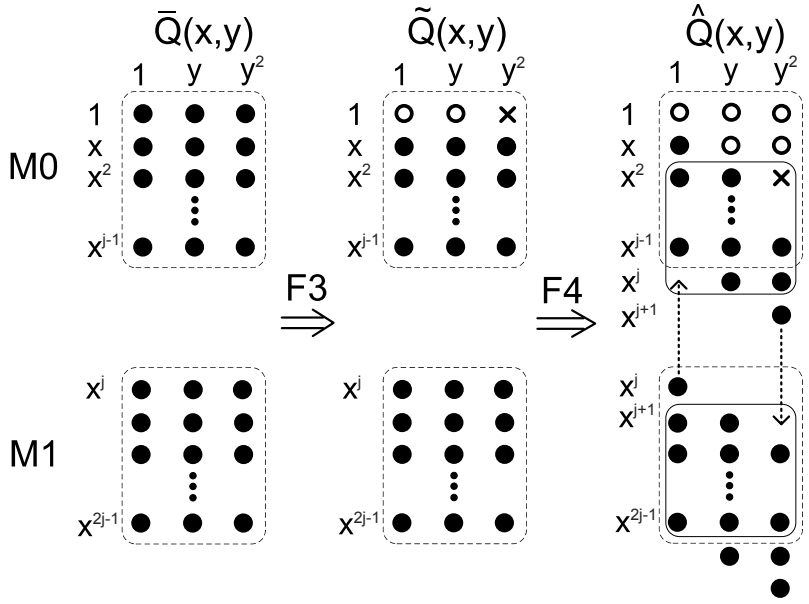

Figure 3.5: Transformations of polynomial coefficients in an iteration level with a second-order root

When the root orders do not change, the coefficients from each of the M_0

and M_1 are connected directly to a PU unit for polynomial updating. However, when root split happens, i.e. the order-2 root splits into two simple roots as shown in Fig. 3.1b, since the root order is one in all later iteration levels, the polynomial coefficients stored in M_1 are discarded, and the coefficients in M_0 are routed to both PU units and updated with two simple roots, respectively. Then the coefficients updated by PU_0 and PU_1 are written back into M_0 and M_1, respectively.

For the root computation of $\bar{Q}(0, Y)$, if $q2 \neq 0$ and $q1 = 0$ in $\bar{Q}(0, Y)$, the corresponding root tree is shown in Fig. 3.1a, the single order-2 root of $\bar{Q}(0, Y)$ is computed through the RC2 unit and routed to both PU_0 and PU_1. If $q2 \neq 0$ and $q1 \neq 0$ in $\bar{Q}(0, Y)$ in $\bar{Q}(0, Y)$, the root computation is carried out in the RC2 unit, if the trace tr from the RC2 unit is nonzero, $\bar{Q}(0, Y)$ has no root and the factorization stops. Otherwise, $roota$ and $rootb$ from the RC2 unit would routed to PU_0 and PU_1, respectively. If each of M_0 and M_1 stores a degree one polynomial, then the root for the $\bar{Q}(0, Y)$ in M_0 is computed through the RC2 unit and the root for the $\bar{Q}(0, Y)$ in M_1 is computed through the RC unit. For the case when the polynomial from the interpolation output is degree one, the coefficients of $\bar{Q}(0, Y)$ would be stored in M_0 and computed by the $RC2$ unit.

Two zero detectors are used to test the coefficients of $\bar{Q}(0, Y)$. At the beginning of the factorization, the zero detector connected to the output of M_0 is used to determine the root-order distribution in iteration level $i = 0$. Later, the zero detector connected to the PU unit is used so that the root-order

distribution is determined prior to the root computation for the next iteration level.

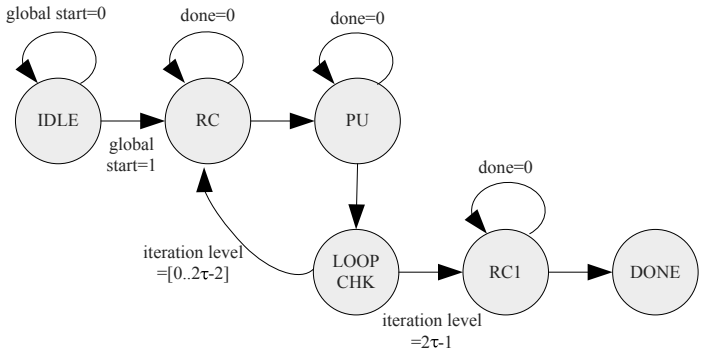

Figure 3.6: Details of the top-level state machine for the BGMD factorization processor

A hierarchical control structure is used in the BGMD factorization processor. Each computational unit has its state machine locally, which interacts with the top-level state machine through proper handshake signals. The details of the top-level state machine are shown in Fig. 3.6. The 'done' signals refer to the handshake signals from various computational units, while the 'start' signal for each computational unit is not explicitly presented. After power up, the BGMD factorization processor remains in the 'IDLE' state until the 'global start' pin is asserted. Then the root computation is carried out in the 'RC' state. Once the root computation is completed, polynomial updating and shifting are carried out in the state 'PU'. The number of iteration levels that has been run is checked in the 'LOOP_CHK' state. Root computation, polynomial updating and shifting would continue to run until $2\tau - 1$ iteration

levels are completed. Then, the root computation for the last iteration level is carried out in the 'RC1' state.

3.4 Implementation Results

The factorization processor for a (255, 239) RS code with $m_{max} = 2$ was synthesized and simulated on a Xilinx Virtex-II XC2V1000-6 device. After place and route, the Xilinx ISE software reported a critical path of 9.884ns. The logic resource utilization is shown in Table 3.1. All the coefficient memories are implemented using the Xilinx distributed RAMs. Simulation results indicate the decoding latency is 253 clock cycles. Hence, the decoding throughput is 815.7Mbps. Higher speed can be achieved by using state-of-art ASIC technology. Comparing to the synthesis result for a factorization processor, the factorization processor for the BGMD decoder occupies less than 1/4 of area as the factorization processor with $m_{max} = 4$ presented in the previous chapter.

Table 3.1: Resource Utilization on a Xilinx XC2V1000-6 FPGA

Resource Type	**Used**	**Ratio**
Slice Flip Flops	331	3%
4 input LUTs	1,580	15%
Block RAMs	0	0%
Total Occupied Sliced	1,063	20%

3.5 Summary

In this chapter, an FPGA implementation of a high-speed factorization processor for the algebraic soft-decision RS decoding using BGMD algorithm is presented. The BGMD factorization processor utilizes a low latency and prediction-free root computation scheme. Furthermore, efficient coefficient storage schemes and parallel processing architectures are employed to reduce the latency of polynomial updating. The resulted hardware implementation achieves a high decoding throughput with low complexity.

Chapter 4

FPGA Implementation and In-circuit Verification of the Factorization Processors

4.1 Introduction

In this chapter, we will first briefly describe the overall design flow for the FPGA implementation used in the factorization processor and the BGMD factorization processor. Next, the hardware setup of the in-circuit verification for the designs will be addressed in detail. Lastly, the in-circuit verification for both factorization processors is given.

4.2 Overall Design Flow

Fig. 4.1 shows the overall design flow for the FPGA hardware implementation. All the designs presented in this thesis are coded in verilog HDL. The synthesis, mapping, place and route are performed using the Prodesign CHIPit manager v3.2.1 and the Xilinx ISE v9.1 design suite. Modelsim v6.2g is used for the simulations at various design stages, which includes the behavior simulation after the HDL coding, the functional simulation after synthesis, and the timing simulation after the delay information is back-annotated to the gate-level netlist. An equivalent C model is written to generate the test

vectors as stimulus and the desired test results for comparison. Static timing analysis is performed to ensure the timing closure. After the design is fully simulated in the host platform, the bit stream file from the implementation is downloaded to the Prodesign CHIPit hardware platform (Iridium edition) for in-circuit verification.

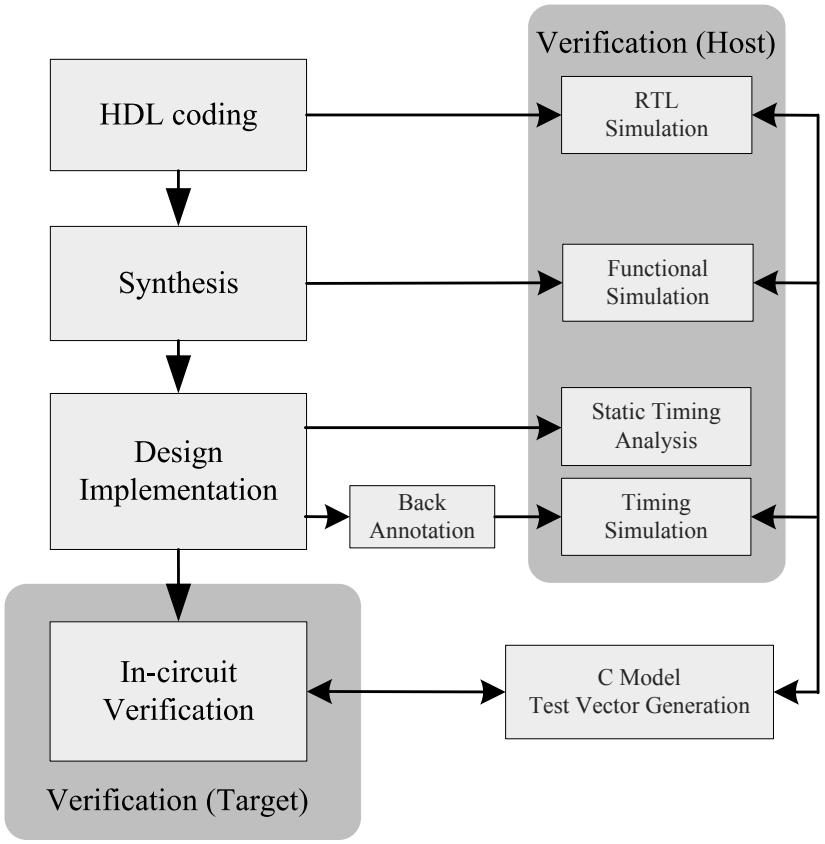

Figure 4.1: Overall design flow

4.3 In-circuit Verification using FPGA Hardware Platform

Two approaches are used in the in-circuit verification: the co-simulation and the high-speed prototyping. In the co-simulation, individual design modules or the entire design can be moved to the hardware platform to accelerate the simulation and connected to the PC host through a co-simulation link, e.g. the CHIPit HDL-Bridge. The entire design is simulated under the same simulation environment (Modelsim) using the same testbench. Fig. 4.2 shows the configuration for the design in the co-simulation. In the PC host, a pseudo DUT (design under test) is created to replace the actual DUT. The pseudo DUT has the same I/O ports and parameters as the original HDL design. It sends out the simulation commands to the hardware platform through the HDL-Bridge co-simulation link. In the hardware platform, a pseudo test bench is created to mimic the behavior of the original test bench. It receives the command sent from the host and generates the test stimulus. All the data are transferred between the PC host and the CHIPit hardware platform through the UMRBus communication system over the PCI-e card.

The co-simulation is used in the early stage of the design verification. We observed the co-simulation is at least 1000 times faster than the pure software simulation. The major reason is that the HDL codes of the DUT are run in parallel in the hardware platform during the co-simulation, while it is executed serially during pure software simulation. However, since the test-bench is run on the simulator on the PC host, the co-simulation still has

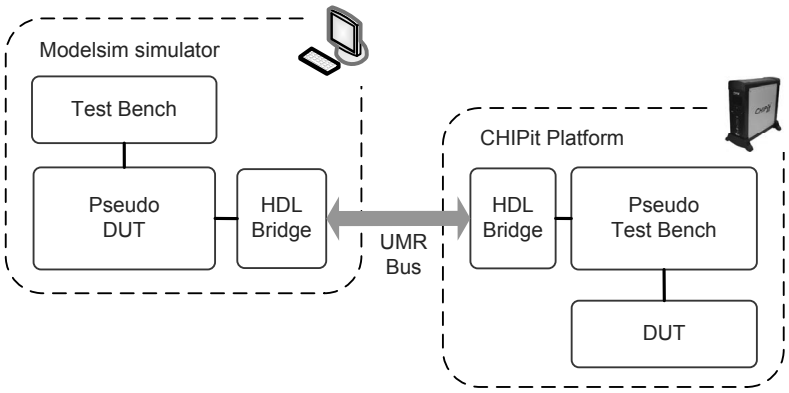

Figure 4.2: Detailed architecture of the co-simulation

a performance bottleneck in the simulator. As a result, the verification speed of the test-bench is limited to about 5KHz - 100KHz during co-simulation. To further speed up the design verification, the high-speed prototyping can be used.

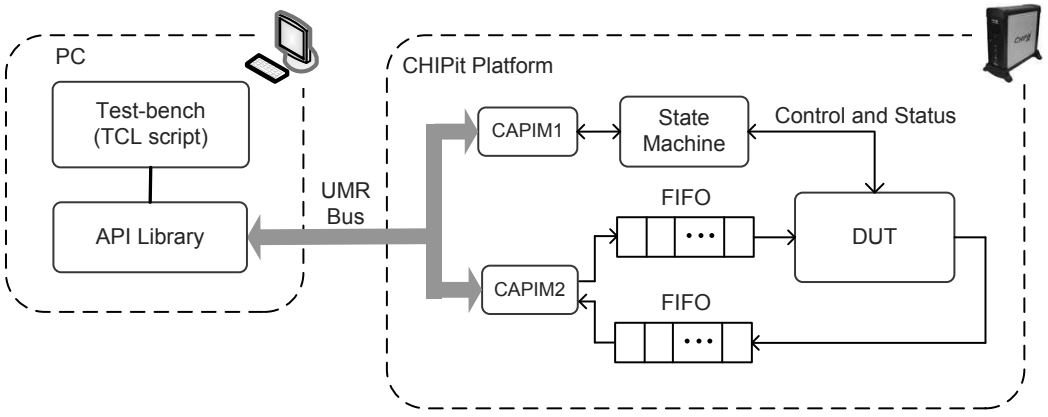

Figure 4.3: Detailed architecture of the high-speed prototyping

Fig. 4.3 shows the detailed architecture of the high-speed prototyping. In this mode, the DUT is run at the maximum operating speed during the

verification. In the PC host, a test-bench written in TCL script is used to send out test vectors and fetch the test results. The API library is a software library to provide an interface between the host application and the UMRBus communication system. In the hardware platform, two client application interface modules (CAPIM) from Prodesign are used for the data and command exchanges between the DUT and the UMRBus. the CAPIM1 is used for the control path and the CAPIM2 is used for the data path. A state machine is employed to generate the controls signals based on the commands from the CAPIM1 and fetch the status of the DUT. Since the UMRBus runs at a fixed clock frequency of 70MHz and the DUT needs to run at a frequency up to 100MHz, two asynchronous FIFO are inserted for data transfer between the two different clock domains. Furthermore, 4-phase handshake scheme is employed to the control signals between the two clock domains to avoid metastability.

4.4 In-circuit Verification of the Factorization Processors

Both the factorization processor and the BGMD factorization processor are verified using the methods addressed above. We first apply the computer generated test vectors to factorization processors. To ensure the test coverage, the test cases for various root distributions with or without root splitting are selected or constructed manually and tested. Both factorization processors successfully pass all the test cases at the frequency of 100MHz.

4.5 Summary

In this chapter, the overall design flow for the FPGA design and implementation is presented. The setup of the in-circuit verification for the factorization processors and the test results are given.

Chapter 5

Error Correction for Multilevel NAND Flash Memory using Reed-Solomon Codes

5.1 Introduction

The increasing demand for non-volatile multi-media and mass data storage has created the need for low-cost and high-capacity flash memories. In recent years, multi-level cell (MLC) flash memories [6–9] have been developed as an effective solution to increasing the storage density and reducing the cost of flash memories. In MLC flash memories, multiple bits are stored per memory cell by programming each cell with multiple threshold levels. The reduced spacing between adjacent threshold levels significantly degrades the reliability of flash memories. To ensure data integrity, error-correcting codes (ECC) need to be employed in these memories. Traditionally, single-bit error-correcting codes, such as Hamming codes, are used for single-bit cell (SBC) flash memories. For MLC flash memories, especially for NAND flash memories which have a large page size, e.g. 8K bits, single-bit error-correcting codes are no longer sufficient.

Several works [9, 23–25] have proposed to use strong Bose-Chaudhuri-Hocquenghem (BCH) codes for MLC NAND flash memories. BCH codes

provide a wide range of error-correcting capability and support flexible code length through the shortening process. Efficient decoding algorithms, such as the Berlekamp-Massey algorithm (BMA) [10, 11] and Euclid algorithms [12], exist for decoding BCH codes. However, the large page size of NAND flash memories demands codes of long length. Hence, the binary BCH codes to be used in NAND flash memories are constructed over finite fields of very large order. Consequently, highly parallel implementations are necessary in order to meet the decoding speed requirement. As a result, large area is required for the decoder implementation.

In this thesis, we propose to use Reed-Solomon (RS) codes for error correction in flash memories. RS codes are good at correcting burst errors since they are non-binary codes constructed over extension fields. For a RS code constructed over $GF(2^m)$, if a symbol error can be corrected, then all the m bit errors in this symbol are corrected. From simulations, it was found that the errors in flash memory applications are random errors, *i.e.* errors happen one bit at a time, if proper bit mapping is used. Although the burst error-correcting capability of RS codes are wasted by having a t-symbol error-correcting code to correct t random bit errors, the RS decoder has much lower complexity than the decoder of a t-bit error-correcting BCH code with about the same code rate and length in terms of bits. The major reason is that in order to construct a code of the same bit length, the required order of the finite field for RS codes is much smaller than that required for BCH codes. Hence, the syndrome computation in RS decoding involves much less number of coefficients and

Chien search [26] needs to try much smaller number of finite field elements. Accordingly, much smaller parallel processing factor is required by the RS decoder to achieve a given throughput. Smaller parallel processing factor translates to smaller number of computational units. As a result, RS decoders require much less area than BCH decoders, despite the extra step of error magnitude computation. As an example, a (828, 820) RS code constructed over $GF(2^{10})$ and a binary (8248, 8192) BCH code constructed over $GF(2^{14})$ are selected to correct 4 bit errors in an 8K data block for flash memories. It can be computed that with 70% of the area, the RS decoder can achieve 121% higher throughput than the BCH decoder.

Another contribution of this thesis is that, for the first time, a Gray code-based mapping scheme is proposed to map the multi-level voltage value in a memory cell to binary numbers. Compared to the direct mapping scheme that was employed previously, our scheme can achieve 0.02dB and 0.2dB coding gains for the (828, 820) RS code and (8248, 8192) BCH code, respectively, without any overhead.

This chapter is organized as follows. Our proposed bit mapping scheme is presented in Section 5.2. This section also describes the simulation setups. Section 5.3 presents the decoder design for RS codes and BCH codes. Complexity analyses and comparisons for the two decoders are provided in Section 5.4. Section 5.5 draws conclusions.

5.2 Flash Memory Models and Gray Mapping

5.2.1 Modeling of Cell Threshold Voltage in Multi-level Flash Memory

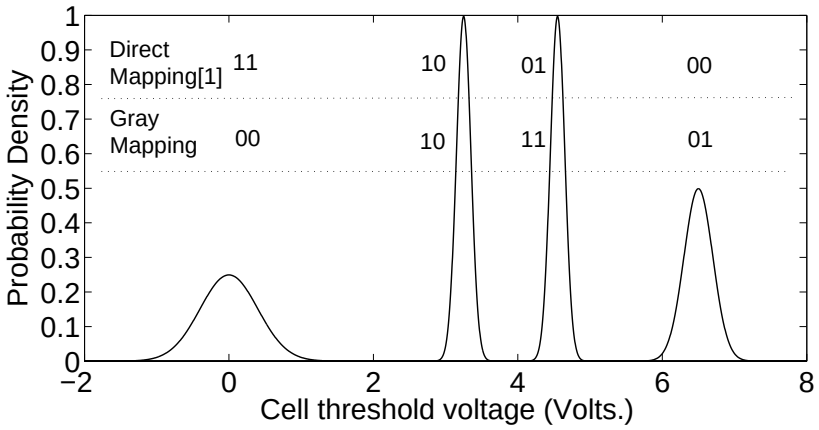

Figure 5.1: Cell threshold voltage distribution model for 2-bits/cell MLC flash memories

Table 5.1: Mean and deviation for the threshold voltage levels in a 2-bit/cell MLC flash memory

Level	**3**	**2**	**1**	**0**
Direct Mapping [7]	11	10	01	00
Gray Mapping (This thesis)	00	10	11	01
Mean (V)	0	3.25	4.55	6.5
Deviation (V)	4σ	σ	σ	2σ

The threshold voltage distribution measurement results for 2-bit/cell MLC flash memories are given by Intel in [7]. For simulation purpose, we use an approximate simulation model [23] shown in Fig. 5.1 [27]. Each of the four threshold levels follows a Gaussian distribution. The mean values and deviations of the four threshold levels are listed in Table 5.1 [27].

5.2.2 Gray Mapping and Detection

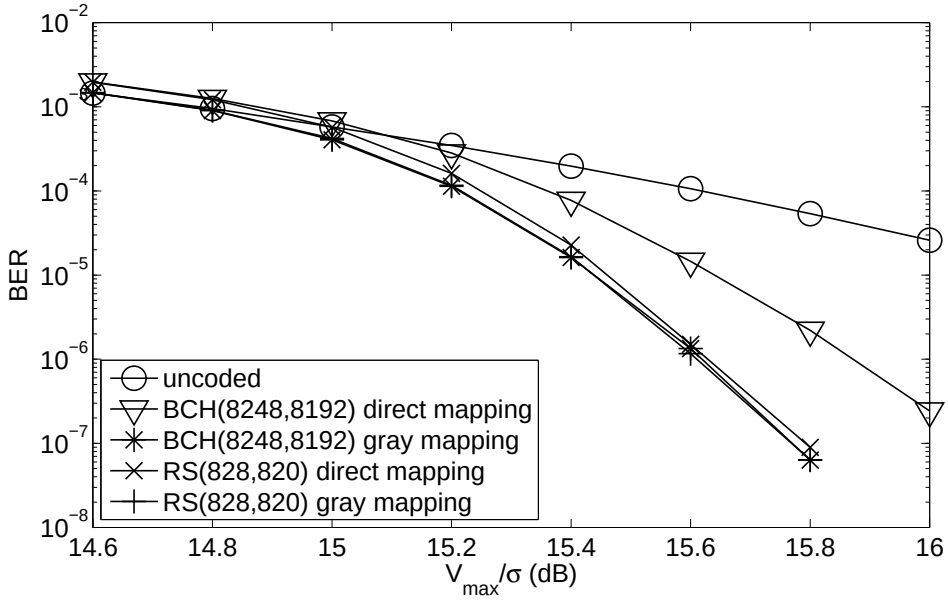

Figure 5.2: BER performance for 2-bit/cell MLC flash memories

For 2-bit/cell MLC flash memories, data are grouped into 2-bit pairs, and each pair is mapped to one of the four threshold levels. In prior works, a direct mapping scheme is used. In the direct mapping scheme, the bits mapped to a threshold level are the binary representation of the threshold number, as shown in Table 5.1. From Fig. 5.1, when errors happen, it is more likely that a voltage level will be taken as one of its adjacent levels in the detector. In addition, the two voltage levels in the middle have both bits different. Hence two bit errors will happen if one of the middle voltage levels is taken as another by mistake. Alternatively, Gray codes can be used for the bit-mapping. As a result, adjacent voltage levels have only one different bit

as shown in Fig. 5.1, and bit error rate (BER) can be reduced. Fig. 5.2 [27] shows some simulation results of BERs for using BCH and RS codes in flash memory applications. In our simulations, the decision threshold for signal detection is set to a voltage, whose distances from the two adjacent means are proportional to the corresponding deviations. Accordingly, the three decision thresholds are set to 2.6V, 3.9V and 5.2V. As it can observed from Fig. 5.2, around 0.2 dB coding gain can be achieved by the BCH code at BER of 10^{-7} if Gray code mapping is used. The coding gain can be achieved by the RS code using Gray mapping is only 0.02 dB. The coding gain for RS codes is trivial because RS codes are symbol-based. In most of the cases, the two bit errors resulted from taking one voltage level in the middle as another by mistake are located in one symbol and can be corrected. Despite the small coding gain for RS codes, the employment of Gray mapping does not require any overhead.

5.3 Architectures for RS and BCH Decoders

In [9, 23, 25], binary BCH codes are employed for error correction in MLC flash memories. To protect a page size of 8K bits, a (8248, 8192) BCH code is used. This code is a shortened 4-bit error correcting (16383, 16327) BCH code constructed over $GF(2^{14})$. To shorten an (n, k) systematic code, s message bits/symbols can be deleted. The resulted shortened code is an $(n - s, k - s)$ code. RS codes can be also employed for error correction. From Fig. 5.2, it can be observed that the (828, 820) RS code can achieve similar or higher coding gain than the (8248, 8192) BCH code. The (828, 820) RS code

is a shortened 4-symbol error correcting (1023, 1015) RS code constructed over $GF(2^{10})$. In addition, compared to the (8248, 8192) BCH code, the (828, 820) RS code has about the same rate and code length in terms of bits.

No prior work has compared the complexity of RS and BCH decoders. In the remaining of this section, the decoder complexity for the BCH and RS codes are compared. The algebraic decodings of RS and BCH codes share three major steps: (a) syndrome computation; (b) key equation solver; (c) error location computation using the Chien search. In addition, since RS codes are non-binary, an extra step is required to determine the error magnitudes using Forney's algorithm [28]. The overall decoder architecture is illustrated in Fig. 5.3 [27].

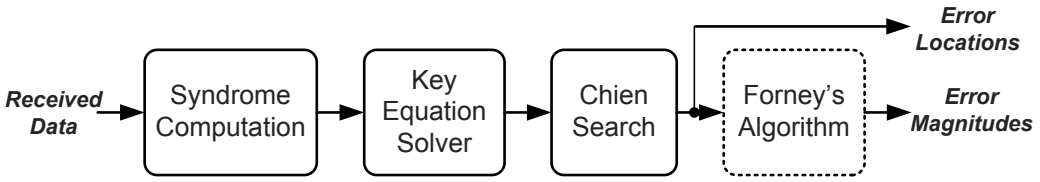

Figure 5.3: RS/BCH decoder architecture

The decoding of a shortened code can be accomplished by the same circuit used for the unshortened code with minor modifications. Hereafter, we consider the decoder architecture for primitive t-error correcting (n, k) code constructed over $GF(2^m)$, where $n = 2^m - 1$. The architecture modifications for the shortened codes will be described when necessary. Throughout this chapter, the parameters n, k, t and s denote bit numbers and symbol numbers in the context of BCH codes and RS codes, respectively.

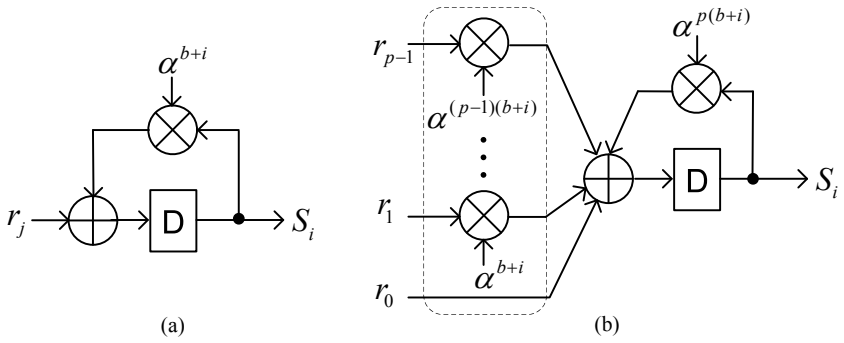

Figure 5.4: Syndrome computation: a) serial architecture b) p-parallel architecture

5.3.1 Syndrome Computation

The received bits/symbols can be considered as the coefficients of a degree $n-1$ polynomial $r(x) = \sum_{i=0}^{n-1} r_i x^i$. The first step of the decoding is to compute the $2t$ syndrome values, i.e. evaluate $r(x)$ over $2t$ consecutive finite field elements starting from α^b:

$$S_i = r(\alpha^{b+i}) = \sum_{j=0}^{n-1} r_j (\alpha^{b+i})^j, 0 \leq i < 2t$$

where α is a primitive element of the finite field and b is an integer. These $2t$ consecutive elements are roots of the generator polynomial of the BCH or RS code.

Applying the Horner's rule, each syndrome can be computed by using a recursive architecture as shown in Fig. 5.4a [27], which includes a constant finite field multiplier and an adder. In the RS decoder, $2t$ such units are required to compute $2t$ syndromes in parallel. In this case, $2t$ constant finite field multipliers and $2t$ adders are required. One clock cycle is required to

process each received symbol. In shortened RS codes, the deleted message symbols can be considered as zero. Hence only $n - s$ clock cycles are required to compute the syndromes.

For binary BCH codes, the syndrome computation can be simplified by using the property that $S_{2i} = S_i^2$. Taking $b = 1$, only t syndromes need to be computed explicitly and the rest can be derived through simple squaring computations [29]. However, the number of coefficients in $r(x)$ for the (8248, 8192) BCH decoding is much larger than that in the (828, 820) RS decoding. Hence, higher level of parallel processing needs to be employed in the BCH syndrome computation in order to achieve the same throughput as the RS decoder. A p-parallel processing syndrome computation unit [30] is shown in Fig. 5.4b, where p received bits are input at each clock cycle and weighted before the recursive summation. If p-parallel processing is employed, then the syndrome computation for BCH decoding requires pt constant multipliers, pt 2-input adders and t squarers. In addition, since each input r_j is binary, the constant multipliers in the dashed block in Fig. 5.4b can be replaced by directly wiring the inputs to the adder according to the nonzero bits in the $\alpha^{b+i}, \ldots, \alpha^{(p-1)(b+i)}$ coefficients. Further area reduction can be achieved by applying sub-structure sharing [31] among all constant multipliers in the syndrome computation using the iterative matching algorithm (IMA)[32]. This optimization depends on the finite field order and the irreducible polynomials used for field construction.

5.3.2 Key Equation Solver

Assume ν errors occurred in the received data. The key equation solver step finds the error locations $i_1, i_2 \cdots, i_\nu$. This is achieved by making use of an error locator polynomial defined as

$$\Lambda(x) = \prod_{l=1}^{\nu}(1 - \alpha^{i_l}x) = 1 + \lambda_1 x + \ldots + \lambda_\nu x^\nu,$$

where α is a primitive element. It can be derived that

$$S_j - \sum_{i=1}^{\nu} \lambda_i S_{j-i} = 0 \tag{5.1}$$

for $1 \leq j < 2t$. In addition, for RS codes, an error magnitude polynomial $\Omega(x)$ is calculated in this step. Define the syndrome polynomial as $S(x) = S_1 x + S_2 x^2 + \cdots + S_{2t} x^{2t}$, $\Omega(x)$ is the polynomial that satisfies the following condition:

$$(1 + S(x))\Lambda(x) = \Omega(x) \ mod \ x^{2t+1}.$$

Efficient algorithms, such as BMA, exist to compute $\Lambda(x)$ and $\Omega(x)$. Starting from $j = 0$, the BMA constructs the $\Lambda(x)$ iteratively to satisfy one more equation described by (5.1) each time. In each iteration, a discrepancy value $S_j - \sum_{i=1}^{\nu} \lambda_i S_{j-i}$ is first computed. If it is not zero, then the $\Lambda(x)$ is modified by making use of a correction polynomial to make the discrepancy zero. It can be observed that the computation of the discrepancy involves polynomial convolutions. In addition, the updating of $\Lambda(x)$ depends on the discrepancy. Therefore, long latency is required in each iteration loop. To

increase the speed, a reformulated inversion-less BMA (RiBM) [33] has been proposed. The RiBM algorithm introduces a discrepancy polynomial to calculate the discrepancy value recursively. Moreover, it breaks the dependency between the error locator polynomial and the discrepancy. As a result, the coefficients of both polynomials can be updated simultaneously in each iteration. The RiBM algorithm leads to a simple control scheme and a systolic architecture, which has $3t + 1$ processing elements (PEs) with one multiplier and one adder in the critical path. To reduce the area, the RiBM architecture can be folded by a factor of $3t + 1$ and use only one PE [34]. The folded architecture is shown in Fig. 5.5.

In the architecture shown in Fig. 5.5, the $2t$ syndromes are loaded into the first $2t$ registers in both rows at the beginning. The rest of the registers are initialized as '0' except that the rightmost registers in both rows are initialized as '1'. Since $2t$ iterations are required and each iteration takes $3t + 1$ clock cycles, the key equation solver step for RS decoding requires $2t \times (3t+1)$ clock cycles. After $2t \times (3t + 1)$ clock cycles, the coefficients of $\Omega^{(h)}(x)$ and $\Lambda(x)$ are available in first t and the following $t + 1$ registers, respectively, in the top row. Here $\Omega^{(h)}(x)$ satisfies $x^{2t}\Omega^{(h)}(x) + \Omega(x) = 0|_{x=\alpha^{-i_l}}$. Although $\Omega^{(h)}(x)$ does not equal $\Omega(x)$, it has the same evaluation values on α^{-i_l} as $\Omega(x)$. These evaluation values are what actually needed to compute the error magnitudes in the Forney's algorithm. For the detailed description of this architecture, the interested reader is referred to [33, 34]. The critical path of this architecture has one multiplier, one adder and one multiplexor.

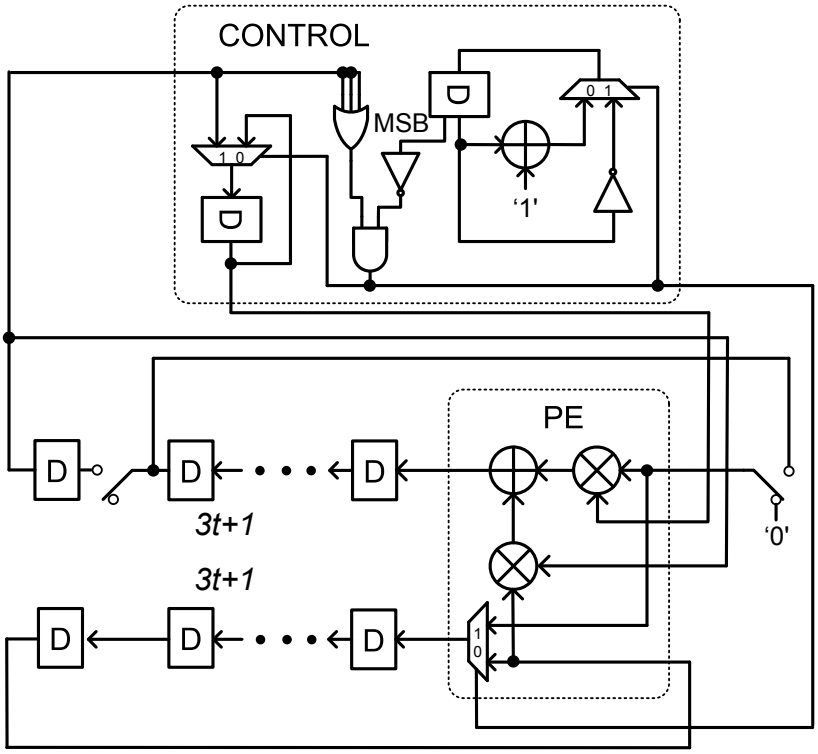

Figure 5.5: Ultra folded RiBM architecture

For binary BCH codes, the discrepancy is always zero in each odd iteration. Hence, all the odd iterations can be skipped and only t iterations are required [29]. In addition, since $\Omega^{(h)}(x)$ does not need to be computed, only t registers is required in each row in Fig. 5.5 and each iteration takes $2t$ clock cycles [25]. Therefore $t \times (2t) = 2t^2$ clock cycles are required for the key equation solver step in BCH decoding.

5.3.3 Chien Search and Forney's algorithm

After $\Lambda(x)$ is computed, the error locations can be found through computing the roots of $\Lambda(x)$. Root computations over finite fields can be carried out by Chien search, which is based on exhaustive search. In addition, RS decoding requires an additional step to determine the error magnitudes. The Forney's algorithm can be used to calculate the error magnitude for the error located at position i_l:

$$e_{i_l} = \left. \frac{x^b \Omega(x)}{x\Lambda'(x)} \right|_{x=\alpha^{-i_l}}. \tag{5.2}$$

In (5.2), $x\Lambda'(x) = \lambda_1 x + \lambda_3 x^3 + \ldots + \lambda_{2t-1} x^{2t-1}$ is the sum of the odd degree terms of $\Lambda(x)$. It can be directly obtained during the computation of $\Lambda(x)$. Since the error magnitude polynomial $\Omega^{(h)}(x)$ from the RiBM algorithm satisfies $x^{2t}\Omega^{(h)}(x) + \Omega(x) = 0|_{x=\alpha^{-i_l}}$, $\Omega^{(h)}(x)$ is used instead of $\Omega(x)$ in (5.2), the term x^b can be eliminated by setting $b = -t$.

The architecture for the Chien search and error magnitude computation in RS decoding is shown in Fig. 5.6 [27]. In this architecture, $\omega_0, \omega_1, \cdots, \omega_{t-1}$ are the coefficients of $\Omega^{(h)}(x)$. The upper part of this architecture carries out the Chien search and the lower part is for the error magnitude computation. The evaluations of the even terms and odd terms of $\Lambda(x)$ are done by the upper left and right parts, respectively. Then the outputs of these two parts are added up. If the sum is zero, it indicates that a root has been found. Since the evaluation values of the odd terms are the evaluation values of the denominator in (5.2), the output of the upper right part can be directly used

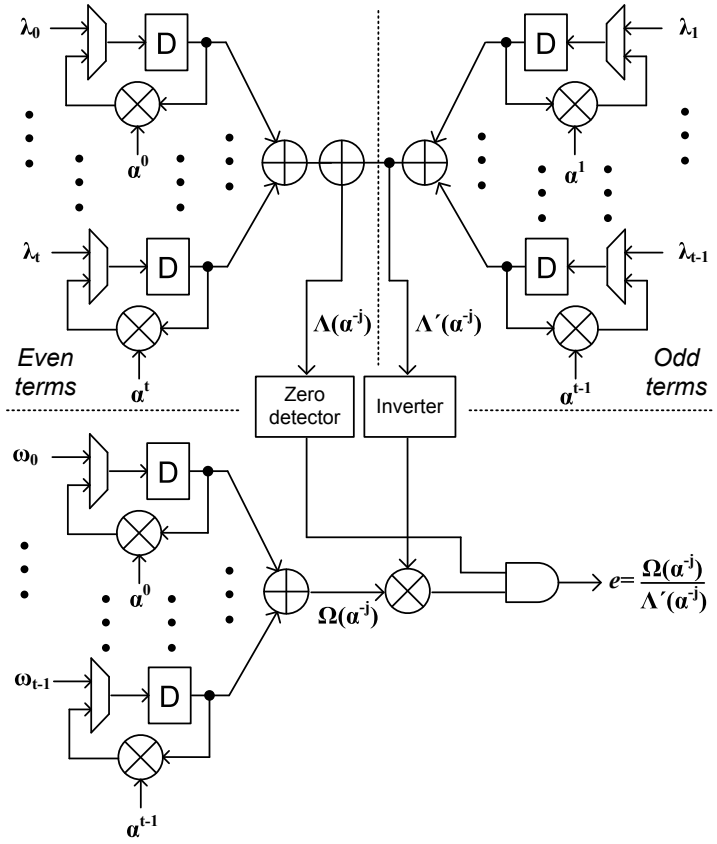

Figure 5.6: Architecture for Chien search and error magnitude computation

in the error magnitude computation. From (5.2), the Forney's algorithm requires one finite field inversion. To reduce the area requirement of inversion, composite field arithmetic can be employed [35]. In addition, since the inversion is located in a feed-forward path, pipelining can be applied to reduce the length of the critical path. The hardware complexity for the Chien search and error magnitude computation block is $2t$ constant multipliers, $2t-1$ 2-input adders, one multiplier and one inverter. The architecture of Fig. 5.6 requires the search to begin with the field element α or 1. Since the roots of $\Lambda(x)$ are

the reciprocals of the error locations, the error correction would always begin from the α^{2^m-2} position. One field element is tried in each clock cycle. Hence, $2^m - 1$ clock cycles are required to find all the error locations.

In binary BCH decoding, there is no need to compute the error magnitudes. In addition, the same architecture in the upper part of Fig. 5.6 can be used for the Chien search to find the error locations. However, since the BCH code is constructed over a much larger finite field than the RS code, a much larger number of field elements need to be checked in the Chien search. Therefore, to achieve the same decoding throughput, a large parallel processing factor needs to be employed to the Chien search for BCH decoding. Assuming a parallel processing factor of p is used, the hardware complexity of the Chien search block is pt constant multipliers and $p(t-1)$ adders. To further reduce the hardware, sub-structure sharing can be applied among all the constant multipliers. In addition, the Chien search can be carried out on $\tilde{\Lambda}(x) = \lambda_\nu + \lambda_{\nu-1}x + \ldots + \lambda_1 x^{\nu-1} + x^\nu$ in binary BCH decoding since the computation of $x\Lambda'(x)|_{x=\alpha^{-i_l}}$ is not required. In this case, the roots found for $\tilde{\Lambda}(x)$ are reciprocals of the roots of $\Lambda(x)$. Hence, the computed roots are already the error locations and the error correction can start from the α position. For shortened codes, there is no need to find the errors in the deleted message bits. Therefore, only $n - s$ field elements need to be checked in the Chien search. Employing p-parallel processing, the Chien search for the BCH decoding requires $\lceil \frac{n-s}{p} \rceil$ clock cycles.

5.3.4 Computation Scheduling

From the architectures discussed above, the latencies of both the syndrome computation and Chien search are proportional to the code length. On the other hand, the latency for the key equation solver is proportional to t^2. For the error correction in MLC flash memories, the error-correcting capability t is much smaller than the code length. Hence, the latencies of the syndrome computation and Chien search dominate. Accordingly, although pipelining can be employed to break the computation of all three steps, pipelining will be only applied to separate the Chien search step from the others in order to increase the hardware utilization efficiency. In this case, the clock cycle number required for decoding each block is the maximum of the clock cycle numbers required for syndrome computation and key equation solver and that of the Chien search (and error magnitude computation for RS decoding).

5.4 Hardware Complexity and Throughput Comparisons

Table 5.2 and 5.3 [27] summarize the gate count and critical path of each building block except the control block in the (828, 820) RS decoder and (8246, 8192) BCH decoder, respectively. All gates in these tables refer to 2-input gates except the single input NOT gates, and the MUXes refer to 1-bit 2:1 multiplexors. In the (8246, 8192) BCH decoder, a parallel processing factor of 8 is employed in both the syndrome computation and Chien search blocks. Multiple irreducible polynomials can be used to construct $GF(2^{10})$ and $GF(2^{14})$. $p(x) = x^{10} + x^3 + 1$ and $p(x) = x^{14} + x^{10} + x^6 + x + 1$ are picked

in our design in order to minimize the hardware complexity.

Table 5.2: Gate counts and critical paths for the building blocks in the (828, 820) RS decoder

-	**Area**	**Critical path**
Multiplier	101XOR+100AND	5XOR+1AND
Adder	10XOR	1XOR
Inverter	164XOR+160AND+36OR+5NOT	16XOR+5AND+3OR+1NOT
Syndrome Computation	116XOR	-
Key Equation Solver	212XOR+200AND+30MUX	6XOR+1AND+1MUX
Chien Search	50XOR+9OR+90MUX	-
Forney algorithm	301XOR+270AND+36OR+5NOT	-
Total Gate Count	679XOR+470AND+45OR+5NOT+120MUX	6XOR+1AND+1MUX

Table 5.3: Gate counts and critical paths for the building blocks in the (8246, 8192) BCH decoder

-	**Area**	**Critical path**
Multiplier	221XOR+196AND	14XOR+1AND
Adder	14XOR	1XOR
Squarer	21XOR	3XOR
Syndrome Computation	339XOR	-
Key Equation Solver	456XOR+392AND+42MUX	15XOR+1AND+1MUX
Chien Search	727XOR+104OR+56MUX	-
Total Gate Count	1522XOR+392AND+104OR+98MUX	15XOR+1AND+1MUX

From Table 5.2 and 5.3, it can be observed that the multiplier in $GF(2^{10})$ requires less than 50% of the area of the multiplier in $GF(2^{14})$ and the critical path is also much shorter. The major reason is that a trinomial exists as an irreducible polynomial for the construction of $GF(2^{10})$ but not $GF(2^{14})$. As it was mentioned before, the inverter required for the error magnitude computation in (5.2) is implemented by using composite field arithmetic [35]. In addition, the critical path of the composite field inverter is almost three times

of that in the overall design. Hence, the inverter is pipelined into three stages.

As it was mentioned in the previous section, the constant multipliers for the syndrome computation in BCH decoding can be replaced by wiring the input bits, and substructure sharing can be applied among the multipliers. In addition, the number of syndromes needs to be computed explicitly is only t for a t-error-correcting BCH code. After taking these optimizations into account, the syndrome computation in the (8246, 8192) BCH decoding still requires more than twice of the area as that in the (828, 820) RS decoding due to the 8-parallel processing employed. From Table 5.2 and 5.3, it can be also observed that the key equation solver block in the RS decoder has much lower complexity than that in the BCH decoder. This is due to the simpler multipliers in $GF(2^{10})$. The Chien search in the RS decoder has a much smaller area than the one in BCH decoder. The major reason is that in the Chien search of BCH decoding, high parallel processing is employed to search a much larger number of finite field elements.

In summary, the overall hardware complexity for a (828, 820) RS decoder is 679 XOR gates, 470 AND gates, 45 OR gates, 5 NOT gates, 120 MUXes, and 534 registers, the implementation of the (8246, 8192) BCH decoder requires 1522 XOR gates, 392 AND gates, 104 OR gates, 98 MUXes, and 638 registers. Assume that each AND or OR gate occupies 3/4 of the area of an XOR, each NOT gate occupies half of the area of an XOR, a MUX has the same area as an XOR, and each register requires about 3 times of the area of an XOR. The (828, 820) RS decoder occupies 70% of the area of the

(8246, 8192) BCH decoder, despite the extra step required to compute the error magnitudes using the Forney's algorithm.

The minimum achievable clock period of both the RS and BCH decoders is limited by the feedback loop in the key equation solver block. It contains one multiplier, one adder and one MUX. Since there are no feedback loops involved in other parts of the decoder architecture, pipelining can be employed to have a critical path of one multiplier, one adder and one MUX. Hence, the critical path of the RS decoder is about 47% of that in the BCH decoder. The (828, 820) RS decoder requires 828, 104 and 1024 clock cycles for the syndrome computation, key equation solver, Chien search and error magnitude computation, respectively. The (8248, 8192) BCH decoder requires 1031, 32 and 1031 clock cycles for the three decoding steps, respectively. Applying the scheduling scheme mentioned in the previous section, it can be derived that decoding one block of (828, 820) RS code requires 1024 clock cycles, while 1063 clock cycles are required for the (8248, 8192) BCH code. Accordingly, the RS decoder can achieve 121% higher throughput than the BCH decoder.

When codes constructed over other fields are considered, the area and throughput differences may become less if there is no trinomial that can be used to construct the finite field for RS codes. However, it is still expected that, with similar error-correcting performance, RS codes of similar code rate and length requires smaller area and can achieve higher efficiency in the decoder implementation.

5.5 Summary

In this chapter, a novel bit-mapping scheme using the Gray code is proposed. The proposed scheme can achieve additional coding gain without any overhead. In addition, the complexity of RS codes and BCH codes with similar code rate and length are compared. It can be concluded that the RS codes can achieve similar error-correcting performance as the BCH codes in MLC flash memory applications with lower decoding complexity. Therefore, RS codes are more suitable for these applications.

Chapter 6

Conclusions and Future Research

6.1 Conclusions

This thesis considers several hardware design and implementation issues for error-correcting coding, including the FPGA implementation of the factorization step for algebraic soft-decision RS decoding and the application of RS codes for MLC NAND flash memories.

For the FPGA implementation of the factorization step for soft-decision RS decoding, we first present an FPGA implementation of a factorization processor for algebraic soft-decision RS decoding in general case. The proposed factorization architecture is general and extensible. It can be easily modified to support factorization of polynomial with designated degree. Parallel processing is employed to speed up the polynomial updating. In addition, efficient coefficient storage and transfer scheme is developed to facilitate the parallel processing and minimize the latency. Next, we present an FPGA implementation of a factorization processor specific for the BGMD decoding. The implementation is based on the architecture proposed in [5]. A low-latency and prediction-free scheme is utilized for the root computation in the factorization. Moreover, parallel processing and efficient coefficient storage scheme are

employed to reduce the latency of the polynomial updating. The BGMD factorization processor achieves a high decoding throughput and has low hardware area requirement, although it is only applicable to the polynomial factorization with low Y-degree.

Another topic covered in this thesis is on the application of RS codes in the error correction for MLC NAND flash memories. First, we propose a novel bit-mapping scheme using the Gray code. The proposed scheme can achieve additional coding gain without any overhead. In addition, the complexity of RS codes and BCH codes with similar code rate and length are compared. It can be concluded that the RS codes can achieve similar error-correcting performance as the BCH codes in MLC flash memory applications with lower decoding complexity. Therefore, RS codes are more suitable for these applications.

6.2 Future Research Directions

In the factorization processor presented in Chapter 2, a root-order prediction-based root computation scheme is utilized to circumvent the exhaustive-search-based root computation from the second iteration level. However, the root computation in the first iteration level is still carried out by exhaustive search, which contributes to a significant portion of the overall factorization latency. In [36], a novel iterative prediction scheme is proposed for the root computation in the first iteration level. The resulted factorization architecture can achieve significant speedup over prior works. Thus it is of practical in-

terest to incorporate the iterative prediction root computation scheme to the factorization processor to further reduce the factorization latency.

In the BGMD factorization processor presented in Chapter 3, since the updated polynomial coefficients for the next iteration level are available after at most three clock cycles of polynomial updating, further speedup can be achieved by performing the root computation and the polynomial updating simultaneously in each iteration level. However, the design throughput is still limited by the long critical path and overall clock cycles. Thus, it is highly desirable to develop new architectures to further break the critical path and reduce the overall clock cycles of the BGMD factorization processor.

For the error correction in the MLC flash memories, future studies will be directed to develop more efficient decoders for flash memories applications. The investigation of the silicon cost and implementation for both error-correcting coding schemes are also interesting topics.

Bibliography

[1] M. Sudan, "Decoding of Reed-Solomon codes beyond the error-correction bound," *Journal of Complexity*, vol. 13, no. 1, pp. 180–193, 1997.

[2] V. Guruswami and M. Sudan, "Improved decoding of Reed-Solomon and algebraic-geometric codes," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 1755–1764, 1999.

[3] R. Koetter and A. Vardy, "Algebraic soft-decision decoding of Reed-Solomon codes," *IEEE Transactions on Information Theory*, vol. 49, no. 11, pp. 2809–2825, 2003.

[4] J. Jiang and K. R. Narayanan, "Algebraic soft decision decoding of Reed-Solomon codes using bit-level soft information," *Proc. Allerton Conference on Communications, Control and Computing*, 2006.

[5] J. Zhu and X. Zhang, "Efficient VLSI architecture for soft-decision decoding of Reed-Solomon codes," *IEEE Transactions on Circuit and System - I: Regular Papers*, submitted.

[6] M. Bauer, R. Alexis, G. Atwood, B. Baltar, A. Fazio, K. Frary, M. Hensel, M. Ishac, J. Javanifard, M. Landgraf, D. Leak, K. Loe, D. Mills, P. Ruby, R. Rozman, S. Sweha, S. Talreja, and K. Wojciechowski, "A multilevel-cell

32 mb flash memory," *IEEE International Solid-State Circuits Conference (ISSCC), 1995.*, pp. 132–133, 351, 15-17 Feb 1995.

[7] G. Atwood, A. Fazio, D. Mills, and B. Reaves, "Intel StrataFlash memory technology overview," *Intel Technology Journal*, pp. 1–8, 4th Quarter, 1997.

[8] B. Ricco, G. Torelli, M. Lanzoni, A. Manstretta, H. Maes, D. Montanari, and A. Modelli, "Nonvolatile multilevel memories for digital applications," *Proceedings of the IEEE*, vol. 86, no. 12, pp. 2399–2423, Dec 1998.

[9] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. Di Martino, L. D'Onofrio, A. Gambardella, E. Grillea, G. Guerra, D. Kim, C. Missiroli, I. Motta, A. Prisco, G. Ragone, M. Romano, M. Sangalli, P. Sauro, M. Scotti, and S. Won, "A 4Gb 2b/cell NAND flash memory with embedded 5b BCH ECC for 36MB/s system read throughput," *IEEE International Solid-State Circuits Conference (ISSCC), 2006.*, pp. 497–506, Feb. 6-9, 2006.

[10] E. R. Berlekamp, *Algebraic Coding Theory.* McGraw-Hill, 1968.

[11] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, Jan 1969.

[12] Y. Sugiyama, M. Kasahara, S. Hirawawa, and T. Namekawa, "A method for solving key equation for decoding Goppa codes," *Information and*

Control, vol. 27, pp. 87–99, 1975.

[13] G.-L. Feng and K. Tzeng, "A generalization of the Berlekamp-Massey algorithm for multisequence shift-register synthesis with applications to decoding cyclic codes," *IEEE Transactions on Information Theory*, vol. 37, no. 5, pp. 1274–1287, Sep 1991.

[14] S. Gao and M. A. Shokrollahi, "Computing roots of polynomials over function fields of curves," *Coding Theory and Cryptography, D. Joyner, Ed.*, pp. 114–228, 1999.

[15] R. M. Roth and G. Ruckenstein, "Efficient decoding of Reed-Solomon codes beyond half the minimum distance," *IEEE Transactions on Information Theory*, vol. 46, no. 1, pp. 246–257, 2000.

[16] X.-W. Wu and P. Siegel, "Efficient root-finding algorithm with application to list decoding of algebraic-geometric codes," *IEEE Transactions on Information Theory*, vol. 47, no. 6, pp. 2579–2587, Sep 2001.

[17] A. Ahmed, R. Koetter, and N. Shanbhag, "VLSI architectures for soft-decision decoding of Reed-Solomon codes," *2004 IEEE International Conference on Communications*, vol. 5, pp. 2584–2590, 20-24 June 2004.

[18] W. J. Gross, *Implementation of algebraic soft-decision Reed-Solomon decoders*. PhD thesis, E.C.E., Univ. of Toronto, Canada, 2003.

[19] J. Ma, A. Vardy, Z. Wang, and Q. Chen, "Factorization architecture by direct root computation for algebraic soft-decision decoding of Reed-Solomon," *ICASSP 2007*, pp. II–1–4, April 2007.

[20] X. Zhang and K. K. Parhi, "Fast factorization architecture in soft-decision Reed-Solomon decoding," *IEEE Transactions on VLSI System*, vol. 13, no. 4, pp. 413–426, 2005.

[21] K. Ralf and A. Vardy, "A complexity reducing transformation in algebraic list decoding of Reed-Solomon codes," *Proc. ITW2003*, pp. 10–13, April 2003.

[22] B. Chen and X. Zhang, "FPGA implementation of a factorization processor for soft-decision Reed-Solomon decoding," *IEEE International Symposium on Circuits and Systems*, May 2008.

[23] F. Sun, K. Rose, and T. Zhang, "On the use of strong BCH codes for improving multilevel NAND flash memory storage capacity," *IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2006.

[24] F. Sun, S. Devarajan, K. Rose, and T. Zhang, "Design of on-chip error correction systems for multilevel NOR and NAND flash memories," *IET Circuits, Devices and Systems*, vol. 1, no. 3, pp. 241–249, Jun. 2007.

[25] W. Liu, J. Rho, and W. Sung, "Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories," *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 248–253, Oct. 2006.

[26] R. Chien, "Cyclic decoding procedure for the bose-chaudhuri-hocquenghem codes," *IEEE Transactions on Information Theory*, vol. 10, pp. 357–363, October 1964.

[27] B. Chen and X. Zhang, "Error correction for multi-level NAND flash memory using Reed-Solomon codes," *IEEE Workshop on Signal Processing Systems (SiPS)*, submitted.

[28] G. D. Forney, "On decoding BCH codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 549–557, Oct 1965.

[29] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 2nd ed., 2004.

[30] L. Song, M.-L. Yu, and M. Shaffer, "10- and 40-Gb/s forward error correction devices for optical communications," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1565–1573, Nov 2002.

[31] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley, Jan. 1999.

[32] Y. Chen and K. K. Parhi, "Small area parallel Chien search architectures for long BCH codes," *IEEE Transactions on VLSI System*, vol. 12, no. 5, pp. 545–549, 2004.

[33] D. V. Sarwate and N. R. Shanbhag, "High-speed architectures for Reed-Solomon decoders," *IEEE Transactions on VLSI System*, vol. 9, no. 5, pp. 641–655, 2001.

[34] K. Seth, K. N. Viswajith, S. Srinivasan, and V. Kamakoti, "Ultra folded high-speed architectures for Reed Solomon decoders," *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pp. 1063–9667, 3-7 Jan. 2006.

[35] C. Paar, *Efficient VLSI Architecture for Bit-Parallel Computations in Galois Field*. PhD thesis, Institute for Experimental Mathematics, University of Essen, Germany, 1994.

[36] X. Zhang, "Further exploring the strength of prediction in the factorization of soft-decision Reed-Solomon decoding," *IEEE Transactions on VLSI System*, vol. 15, no. 7, pp. 811–820, 2007.