UNIVERSITY OF CALIFORNIA, SAN DIEGO

Efficient Algebraic Soft-Decision Decoding of Reed-Solomon Codes

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering
(Communications Theory and Systems)

by

Jun Ma

2007

The dissertation of Jun Ma is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007

To my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Alexander Vardy, for his guidance and inspiration throughout my study at UCSD. I feel very lucky that Alex accepted me as his part-time student. Without him, completing this dissertation would not have been possible.

I want to thank Dr. Zhongfeng Wang for the fruitful collaboration during the second half of my Ph.D. work. Zhongfeng has been both a friend and a mentor to me, and I learned a lot about VLSI system design from him. In addition, I would like to thank Professor Ralf Koetter for his help with Chapter 4 and Chapter 8 of the dissertation. Ralf has always been a source of brilliant new ideas.

I also acknowledge my dissertation committee members, Professor Ilya Dumer, Professor Alon Orlitsky, Professor Paul H. Siegel, and Professor Jack K. Wolf, for their kind support throughout my study.

In addition, I want to thank Arshad Ahmed, Qinqin Chen and Peter Trifonov for the helpful discussions and productive collaborations.

I am especially grateful to my family. My parents, Guoliang Ma and Hao Tang, and my parent-in-laws, Yuzhen Liu and Xiuqin Jiang, have encouraged and supported me for so many years. I am indebted to my wife, Aixin Liu, for so many sleepless nights she endured to care for our new-born son. William arrived in the last year of my dissertation work, and he brings a lot of joy to us. This thesis is dedicated to all of them.

Chapter 3 has been presented, at *2004 International Symposium on Information Theory (ISIT)*, Ma, Jun; Trifonov, Peter; Vardy, Alexander. The dissertation author was the primary investigator and author of the paper.

The material of Chapter 4 has been presented, in part, at *2003 International Symposium on Information Theory (ISIT)*, Koetter, Ralf; Ma, Jun; Vardy, Alexander; Ahmed, Arshad. The dissertation author was a joint investigator and co-author of the paper.

The results of the Chapter 5 have been presented, in part, at *2007 International Symposium on Information Theory (ISIT)*, Ma, Jun; Vardy, Alexander. The dissertation author was primary investigator and author of the paper.

The material of Chapter 6 has been presented, in part, at *2006 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng. The dissertation author was the primary investigator and author of this paper.

We have presented the results of Chapter 7, in part, at *2006 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng, and *IEEE Transactions on VLSI Systems, September 2006 pp. 937-950.* Wang, Zhongfeng; Ma, Jun, The dissertation author was a joint investigator and co-author of both papers.

The material of Chapter 8, in part, is published in proceedings of *2007 International Conference Acoustics, Speech and Signal Processing (ICASSP)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng; Chen Qinqin, and proceedings of *2007 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng; Chen Qinqin. It is also to be published in *IEEE Transactions on VLSI Systems* Ma, Jun; Wang, Zhongfeng; Vardy, Alexander. The dissertation author was the primary investigator and author of both papers.

VITA

1997	B. E., Biomedical Engineering, Tsinghua University, China.
2000	M. S., Electrical Engineering, University of Minnesota Twin Cities.
2007	Ph. D., Electrical Engineering (Communications Theory and Systems), University of California San Diego.

PUBLICATIONS

R. Koetter, J. Ma , A. Vardy and A. Ahmed, *Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes,* Proc. IEEE Int. Symp. Inform. Theory, Yokohama, Japan, July 2003, pg. 365.

J. Ma , P. Trifonov and A. Vardy, *Dived-and-conquer interpolation for list decoding of Reed-Solomon codes,* Proc. IEEE Int. Symp. Inform. Theory, Chicago, USA, July 2004, pg. 386.

J. Ma, A. Vardy and Z. Wang, *Efficient fast interpolation architecture for soft-decision decoding of Reed-Solomon codes*, Proc. IEEE Int. Symp. Circ. and Sys., Island of Kos, Greece, May 2006, pp. 4823-4826.

J. Ma, A. Vardy and Z. Wang, *Reencoder design for soft-decision decoding of an (255,239) Reed-Solomon code*, Proc. IEEE Int. Symp. Circ. and Sys., Island of Kos, Greece, May 2006, pp. 3550-3553.

Z. Wang and J. Ma, *High-Speed Interpolation Architecture for Soft-decision Decoding of Reed-Solomon Codes,* IEEE Transactions on VLSI Sys., September 2006 pp. 937-950.

J. Ma, A. Vardy, Z. Wang and Q. Chen, *Factorization architecture by direct root computation for algebraic soft-decision decoding of Reed-Solomon codes,* Proceeding of International Conference on Acoustics, Speech, and Signal Processing, Honolulu, HI, April 2007.

J. Ma, A. Vardy, Z. Wang and Q. Chen, *Direct root computation architecture for algebraic soft-decision decoding of Reed-Solomon codes,* Proceeding of IEEE International Symposium on Circuits and Systems, New Orleans, LA, May 2007.

J. Ma, and A. Vardy, *Reduced complexity Lee-O'Sullivan interpolation for algebraic soft-decision decoding of Reed-Solomon codes,* Accepted to IEEE International Symposium on Information Theory, Nice, France, June 2007.

J. Ma, A. Vardy and Z. Wang, *Low-latency factorization architecture for algebraic soft-decision decoding of Reed-Solomon codes,* Accepted to IEEE Transactions on VLSI Sys.

ABSTRACT OF THE DISSERTATION

Efficient Algebraic Soft-Decision Decoding of Reed-Solomon Codes

by

Jun Ma

Doctor of Philosophy in Electrical Engineering

(Communications Theory and Systems)

University of California San Diego, 2007

Professor Alexander Vardy, Chair

Algebraic soft-decision decoding of Reed-Solomon codes delivers promising gain over conventional hard-decision decoding. The major computational steps in algebraic soft-decoding (as well as Sudan-type list-decoding) are bivariate polynomial interpolation and factorization. In this thesis, we present techniques from both algorithmic and VLSI architectural level that greatly reduce the implementation complexity of a soft-decision Reed-Solomon decoder.

A divide-and-conquer approach to perform the bivariate polynomial interpolation procedure is discussed in Chapter 3. This method can potentially reduce the interpolation complexity of algebraic soft-decision decoding of Reed-Solomon code.

In Chapter 4, a computational technique, based on re-encoding coordinate transformation, is derived that can significantly reduces complexity of bivariate interpolation procedure. With this technique, the original interpolation problem is transformed into another *reduced interpolation problem*, which could be orders of magnitude smaller than the original one. A rigorous proof is presented to show that the two interpolation problems are equivalent. In addition, an efficient factorization procedure that applies directly to the reduced interpolation problem is given.

We apply the re-encoding coordinate transformation technique to the Lee-O'Sullivan interpolation algorithm in Chapter 5. A new basis construction algorithm is developed and it takes into account the additional constraints imposed by the interpolation problem that results upon the re-encoding transformation. The re-encoding coordinate transformation reduces the computational and storage complexity of the Lee-O'Sullivan algorithm by orders of magnitude, and makes it directly comparable to Koetter's algorithm in situations of practical importance.

Chapter 6 presents a VLSI design example of the re-encoding coordinate transformation technique introduced in the previous chapter. A fast and optimal algorithm to determine the re-encoding positions and an architecture that enables concurrent processing and eliminates idle time of the various hardware units are proposed. The entire design is synthesized using SMIC's $0.18-\mu$m library to a total area of 0.51mm^2. It has a throughput of approximately 500Mbps.

A high-speed interpolation architecture is presented in Chapter 7. This novel architecture applies hybrid data format to represent a finite field number, thus breaks the long critical path delay bottleneck associated with existing architectures. The proposed architecture also enables maximum overlap in time between computations at adjacent iterations. It is estimated that the proposed architecture can achieve significantly higher throughput than conventional designs with equivalent or lower hardware complexity.

Partial factorization of bivariate polynomial is also an important step of algebraic soft-decision decoding of Reed-Solomon codes, and it contributes to a significant portion of the overall decoding latency. In Chapter 8, a novel architecture based on direct root computation is proposed to greatly reduce the factorization latency. Compared with existing works, not only does our new architecture have a significantly smaller worst-case decoding latency, but it is also more area efficient since the large amount of hardware consumption for routing polynomial coefficients can be completely avoided.

CHAPTER 1

Introduction

Reed-Solomon (RS) codes are the most widely used error-correcting codes in digital communications and data storage. Standard *hard-decision decoders* correct up to $\frac{n-k}{2}$ symbol errors for an (n, k) Reed-Solomon code. Recently, several breakthroughs have been achieved in improving the error-correction capability of a RS decoder. Sudan [Sud97] showed that *list-decoding* of Reed-Solomon codes can be viewed as a bivariate interpolation problem, thereby correcting more errors than previously thought possible. Specifically, for a (n, k) Reed-Solomon code, the algorithm of [Sud97] produces all codewords whose distance to the received hard-decision vector do not exceed roughly $n - \sqrt{2kn}$. The algorithm of [Sud97] was later extended to decoding of algebraic-geometric codes by Shokrollahi-Wasserman [SW99]. A more careful analysis showed that the list-decoding algorithm of [Sud97] is asymptotically better than the standard hard-decision decoding only if the $\frac{k}{n} \leq \frac{1}{3}$. This limits the applicability of Sudan's algorithm since most practical Reed-Solomon codes are high rate codes. The second step was taken in the work of Guruswami-Sudan [GS99] which showed that one can correct even more errors by interpolating through each point not once, but m times, where m is an arbitrary integer. For $m \to \infty$, the list-decoding algorithm of [GS99] corrects up to $n - \sqrt{nk}$ errors. However, the asymptotic improvement of the Guruswami-Sudan algorithm degenerates to nothing at all for high-rate and finite length Reed-Solomon codes of practical interests. The next key achievement was the work of Koetter and Vardy [KV03a], who extended Guruswami-Sudan's technique, and more importantly, showed how the interpolation multiplicities in the algorithm of [GS99] should be chosen to achieve algebraic *soft-decision* decoding of Reed-Solomon codes. The algorithm of [KV03a] produces substantial gains for high-rate Reed-Solomon codes of finite length and significantly outperforms hard-decision list-decoding.

The goal of this research work is to bridge the gap between the high computational complexity associated with the new class of decoding algorithms and the high-throughput requirement of a practical Reed-Solomon decoder. As shown in Figure 1.1, a typical algebraic soft-decision RS decoder consists of 3 major function blocks, namely multiplicity assignment, interpolation and factorization. The multiplicity assignment block generates the set of interpolation points and their associated multiplicities based on the received soft information from the channel, and it determines the performance of the algebraic soft-decision decoder. On the other hand, most of the computational complexity of the decoder comes from bivariate polynomial interpolation and factorization, which are the focus of this thesis, where techniques from both algorithmic level and VLSI architecture level are investigated.

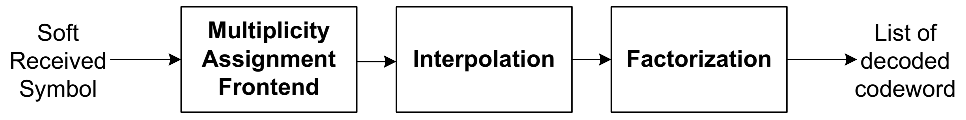

Figure 1.1: Block Diagram of the Soft-Decision Reed-Solomon Decoder

1.1 Summary of Contributions

1.1.1 Re-Encoding Coordinate Transformation Technique

It is widely recognized that bivariate polynomial interpolation is the most computationally intensive step in algebraic soft-decision decoding (or, more generally, in algebraic list-decoding) of Reed-Solomon codes. Consequently, many different algorithms for bivariate polynomial interpolation have been proposed in the past decade — see [LO06b] for a recent survey.While all these algorithms are polynomial-time, they fall short of making the required computation feasible in practical applications, involving long high-rate Reed-Solomon codes. In this thesis, we present a re-encoding coordinate transformation based technique that drastically reduce the space and time complexity of the interpolation process. The re-encoding coordinate transformation transfers the original interpolation problem into another *reduced interpolation problem*, which is orders of magnitude smaller than the original one. A rigorous proof is presented to show that the two interpolation problems are equivalent. An efficient factorization procedure that applies directly to the reduced interpolation problem is also given.

1.1.2 Reduced Complexity Lee-O'Sullivan Interpolation Algorithm

Recently, Lee and O'Sullivan proposed a new interpolation algorithm for algebraic soft-decision decoding of Reed-Solomon codes. In some cases, the Lee-O'Sullivan algorithm turns out to be substantially more efficient than alternative interpolation approaches, such as Koetter's algorithm. Herein, we combine the re-encoding coordinate-transformation technique, originally developed in the context of Koetter's algorithm, with the interpolation method of Lee and O'Sullivan. To this end, we develop a new basis construction algorithm, which takes into account the additional constraints imposed by the interpolation problem that results upon the re-encoding transformation. This reduces the computational and storage complexity of the Lee-O'Sullivan algorithm by orders of magnitude, and makes it directly comparable to Koetter's algorithm in situations of practical importance.

1.1.3 Divide-and-Conquer Interpolation Method

In [Fen99], a divide-and-conquer interpolation algorithm was proposed. This algorithm, though reducing asymptotic interpolation complexity, is still sequential in nature. Thus the divided interpolation problems can not be solved independently in parallel. In this thesis, we devise another divide-and-conquer method by utilizing some algebraic-geometric properties of the solution to the interpolation problem. Our new divide-and-conquer approach enables parallel implementation of bivariate polynomial interpolation and can potentially reduce the interpolation complexity.

1.1.4 High-Speed Interpolation Architecture

The most computationally demanding step in soft-decision decoding of RS codes is bivariate polynomial interpolation. In this thesis, we present a novel interpolation architecture which uses hybrid format representation of finite field numbers and has significantly lower complexity than the architectures proposed in [GKKG05, AKS04b] and can achieve significantly higher processing speed than all known designs for the interpolation process. We will demonstrate that the architecture can be extensively pipelined. Combined with the proposed timing scheme, the average iteration time for the interpolation process can be maximally reduced, which makes it well suited for high speed applications. In addition, the new architecture is inherently scalable. Thus it can be applied to various applications with different speed requirements.

1.1.5 Low-Latency Factorization Architecture

Existing factorization architecture [AKS03a, ZP05] uses exhaustive search based method to compute polynomial roots. In Chapter 8, a novel architecture based on direct root computation is proposed to greatly reduce the factorization latency. Direct root computation is feasible because in most practical applications of algebraic soft-decision decoding of RS codes, enough decoding gain can be achieved with a relatively low interpolation cost, which results in bivariate polynomial with low Y-degree. Compared with existing works, not only does our new architecture have a significantly smaller worst-case decoding latency, but it is also more area efficient since the large amount of hardware consumption for routing polynomial coefficients to various polynomial update engines is completely avoided.

1.1.6 Re-Encoder Design for the (255, 239) Reed-Solomon Code

Prior to this work, there were no reported implementations of the re-encoding coordinate transformation technique. Here we present such a implementation for a (255, 239) Reed-Solomon code. Key features of our implementation include: a fast algorithm to determine the re-encoding points, an area-efficient erasure-only decoding architecture and a scheduling scheme that enables concurrent processing of different steps of re-encoding and coordinate transformation process. Preliminary synthesis results show that the proposed implementation has a throughput of approximately 500Mbs with an area of 0.5mm^2.

1.2 Outline of the Thesis

The thesis is organized as follows.

In Chapter 2, we review the bivariate polynomial interpolation based Reed-Solomon decoding algorithm. The re-encoding coordinate transformation technique is introduced in Chapter 4, where a rigorous proof of the equivalence between the original interpolation problem and the reduced interpolation problem is also provided. We apply the re-encoding coordinate transformation technique to the Lee-O'Sullivan algorithm in Chapter 5. Chapter 6, Chapter 7 and Chapter 8 present architectures that can be used for practical implementation of Reed-Solomon decoders. In Chapter 6, a re-encoding front end design is presented for a $(255, 239)$ Reed-Solomon code. Chapter 7 describes a high-speed architecture for VLSI implementation of the interpolation process. A low-latency factorization architecture based on direct-root computation for low-degree polynomials is given in Chapter 8. Finally we summarize the thesis in Chapter 9 and point out open problems for future research.

CHAPTER 2

Interpolation-Based Decoding of Reed-Solomon Codes

It was recognized early on that decoding Reed-Solomon codes is equivalent to the problem of reconstructing univariate polynomials from their noisy evaluations. Conventional Berlekamp-Massey decoding [Mas69] attempts to solve this problem using *univariate polynomial interpolation*. Specifically, suppose a codeword $\big(f(x_1), f(x_2), \ldots, f(x_n)\big)$ of a Reed-Solomon code $\mathbb{C}_q(n,k)$ was transmitted and a vector $(y_1, y_2, \ldots, y_n) \in \mathbb{F}_q^n$ was received. Then the Berlekamp-Massey algorithm essentially tries to construct a univariate polynomial of degree less than k that passes through as many as possible of the received points $y_1, y_2, \ldots, y_n$. The breakthrough achieved by all algebraic list-decoding algorithms [Sud97, GS99, KV03a] is due in large part to the transition from univariate to *bivariate polynomial interpolation*. In general, the algorithms first construct a nonzero bivariate polynomial $\mathcal{Q}(X,Y)$ of least $(1, k-1)$-weighted degree that passes through $\underline{\text{all}}$ the points $(x_1,y_1), (x_2,y_2), \ldots, (x_s,y_s) \in \mathcal{P}$ with prescribed multiplicities $\{m_{x_i,y_i}\}$, then find all polynomials $f(X)$ of degree $< k$ such that $\mathcal{Q}\big(X, f(X)\big) \equiv 0$.

2.1 Two Definitions of Reed-Solomon Codes

In the original paper published by Reed and Solomon [RS60], the RS code is defined via polynomial evaluation as follows. Let $\mathbb{F}_q$ be the finite field with q elements. The ring of polynomials over $\mathbb{F}_q$ is denoted $\mathbb{F}_q[X]$. Reed-Solomon codes are obtained by evaluating certain subspaces of $\mathbb{F}_q[X]$ in a set of points $\mathcal{D} = \{x_1^*, x_2^*, \ldots, x_n^*\} \subseteq \mathbb{F}_q$. Specifically, the RS code $\mathbb{C}_q(n,k)$ of length n and dimension k is defined as follows:

$$\tilde{\mathbb{C}}_q(n,k) \overset{\text{def}}{=} \left\{ (f(x_1^*), \ldots, f(x_n^*)) \;:\; x_1^*, \ldots, x_n^* \in \mathcal{D},\; f(X) \in \mathbb{F}_q[X],\; \deg f(X) < k \right\} \tag{2.1}$$

Later on, it was realized that RS code is a special case of BCH code, belonging to the family of cyclic code. Along these lines, the RS code can be defined as follows.

$$\mathbb{C}_q(n,k) \overset{\text{def}}{=} \left\{ u(X)g(X) \;:\; u(X) \in \mathbb{F}_q[X],\; \deg u(X) < k \right\} \tag{2.2}$$

where the generator polynomial $g(X)$ is given by $g(X) = \prod_{l=0}^{n-k-1}(X - \alpha^{b+l})$, and b is an arbitrary integer.

Note that when $n < q - 1$ in the 1st definition above, the resulting RS code is referred to as truncated RS code. Correspondingly, the condition $n < q - 1$ leads to a shortened RS code with the 2nd definition. However, the code is no longer cyclic.

Thanks to the fact that a systematic encoder can be easily implemented by using LFSR's (linear feedback shift registers), the 2nd definition is used in all practical applications of RS codes. However, in view of the newly-devised algebraic list decoding algorithms, the original definition of RS code, i.e. (2.1), has to be used.

Fortunately, there exists the following one-to-one mapping between the 2 definitions of RS code given above.

$$\tilde{\mathbb{C}}_q(n,k) = \left\{ (\psi_0 c_0, \ldots, \psi_{n-1} c_{n-1}) \;:\; (c_0, \ldots, c_{n-1}) \in \mathbb{C}_q(n,k) \right\}, \tag{2.3}$$

where $\psi_0, \ldots, \psi_{n-1}$ are n constants in $\mathbb{F}_q$. With this mapping, a shortened Reed-Solomon codeword $c(X) = u(X)g(X)$, where $\deg u(X) < k$ and $g(X) = \prod_{l=0}^{n-k-1} (X - \alpha^{b+l})$, can be converted into a truncated RS codeword $\tilde{c}(X) = \sum_{i=0}^{i=n-1} f(\alpha^i) X^i$, and vice versa. Let us now determine the values of $\psi_0, \ldots, \psi_{n-1}$.

Lemma 2.1 *If $\tilde{c}(X) = \sum_{i=0}^{i=n-1} f(\alpha^i)X^i$ is a codeword of the truncated RS code, then $c(X) = \sum_{i=0}^{i=n-1} f(\alpha^i)\phi(\alpha^i)\alpha^{i(1-b)}X^i$ is a codeword of the shortened RS code, where $\phi(X) \overset{\text{def}}{=} \prod_{i=n}^{q-2}(X - \alpha^i)$.*

Proof. It is sufficient to show that $\{X^{b+l} : l = 0, 1, ..., n-k-1\}$ are roots of $c(X)$. From the definition of $\phi(X)$, we have

$$c(X) = \sum_{i=0}^{i=n-1} f(\alpha^i)\phi(\alpha^i)\alpha^{i(1-b)}X^i = \sum_{i=0}^{i=q-2} f(\alpha^i)\phi(\alpha^i)\alpha^{i(1-b)}X^i,$$

as $\phi(\alpha^i)$ is 0 for $i = n, n+1, ..., q-2$. Let us now define $h(X) = f(X)\phi(X) = \sum_{j=0}^{k-1+q-1-n} h_j X^j$, thus we have

$$\begin{aligned} c(\alpha^{b+l}) &= \sum_{i=0}^{i=q-2} \sum_{j=0}^{k-1+q-1-n} h_j \alpha^{ij}\alpha^{i(1-b)}\alpha^{i(b+l)} \\ &= \sum_{j=0}^{k-1+q-1-n} h_j \sum_{i=0}^{i=q-2} \alpha^{i(j+1-b+b+l)} \\ &= \sum_{j=0}^{k-1+q-1-n} h_j \sum_{i=0}^{i=q-2} \alpha^{i(j+1+l)} \end{aligned}$$

Since $0 \le j \le k-1+q-1-n$ and $0 \le l \le n-k-1$, we have

$$0 < j+1+l \le q-2,$$

i.e., $\alpha^{j+1+l} \ne 1$, and thus the sum $\sum_{i=0}^{i=q-2} \alpha^{i(j+1+l)}$ is always 0 for $l = 0, 1, ..., n-k-1$. ∎

Based on this lemma, the ψ_i's can be computed as follows

$$\psi_i = \frac{\alpha^{i(b-1)}}{\phi(\alpha^i)} \text{ for } i = 0, \ldots, n-1.$$

To better understand how the lemma is constructively formulated, the following steps can be taken. First of all, it is well known that the codeword space of non-shortened RS$(q-1,k)$ code is equivalent between the 2 definitions for narrow-sense RS code, i.e., $b=1$ in the generator polynomial. Next, one realizes that an auxiliary polynomial of degree $q-1-n$ that evaluates to 0 at $\alpha^n,\dots,\alpha^{q-2}$ can bridge the gap between n and $q-1$. In the end, another factor is required to convert between general (non narrow-sense) RS code and narrow sense RS code.

The lemma leads to the following corollary.

Corollary 2.2 *If $c(X)=\sum_{i=0}^{i=n-1}c_iX^i$ is a codeword of the shortened RS code, then $\tilde{c}(X)=\sum_{i=0}^{i=n-1}\frac{c_i}{\phi(\alpha^i)\alpha^{i(1-b)}}X^i$ is a codeword of the truncated RS code, where $\phi(X)\overset{\text{def}}{=}\prod_{i=n}^{q-2}(X-\alpha^i)$. In other words, there exist $f(X)$ with $\deg f(X)<k$, such that $f(\alpha^i)=\frac{c_i}{\phi(\alpha^i)\alpha^{i(1-b)}}$ for $i=0,1,...,n-1$.*

Now we can freely convert a codeword from one definition to the corresponding codeword in the other definition. let us discuss how the $f(X)$ in (2.1) can be computed from $\tilde{c}(X)$. Apparently, once $\tilde{c}(X)$ is known, we can pick any k out of n coefficients, which correspond to $f(X)$ evaluated at k of the n values of $\alpha^0,\alpha^1,...,\alpha^{n-1}$, and apply the Lagrange interpolation method. We now present another method to compute $f(X)$, where the Fourier transform in finite fields is used. Let us assume that we know the full codeword $c'(X)$ obtained by evaluating $f(X)$ at all $q-1$ non-zero numbers of the field. Note that $c'_i=\tilde{c}_i$, for $0\le i<n$. Then $f(X)$ is the inverse Fourier transform of $c'(X)$, i.e.,

$$f_i=c'(\alpha^{-i}),\ \text{for } 0\le i<q-1.$$

It's easy to see that $f_i=0$ for $i\ge k$.

Thus the key is to compute the full codeword $c'(X)$ and we have to apply the equivalence between the 2 full $(n=q-1)$ code definitions. The full codeword $c'(X)$ can also be generated from the following generator polynomial

$$g^*(X)=\prod_{i=1}^{q-1-k}(X-\alpha^i).$$

From $g^*(X)$ and $\tilde{c}(X)$, we can compute a quotient polynomial $q(X)$, such that $c'(X) = q(X)g^*(X)$. Since $q(X)$ only has k coefficients, it is possible to compute it from k lowest-degree coefficients of $\tilde{c}(X)$ and $g^*(X)$. This can be seen from the following: Since $c'(X) = q(X)g^*(X)$, we have

$$\begin{aligned}
c'_0 &= q_0 g^*_0 \\
c'_1 &= q_0 g^*_1 + q_1 g^*_0 \\
&\dots \\
c'_i &= \sum_{j=0}^{i} q_j g^*_{i-j} \\
&\dots \\
c'_k &= \sum_{j=0}^{k} q_j g^*_{k-j}
\end{aligned}$$

Given the equivalence between the 2 definitions we have established in this section, we assume, throughout the rest of the thesis that the polynomial evaluation definition of (2.1) is used.

2.2 The Bivariate Polynomial Interpolation Problem

In order to describe the bivariate polynomial interpolation problem, we need the following definitions. As in [GS99, KV03a, NH98, RR00], we define the weighted degree as follows.

Definition 2.1. *Let $\mathcal{A}(X,Y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} a_{i,j}X^iY^j$ be a bivariate rational function over $\mathbb{F}_q$ and let w_X, w_Y be real numbers. Then the (w_X, w_Y)-weighted degree of $\mathcal{A}(X,Y)$, denoted $\deg_{w_X,w_Y}\mathcal{A}(X,Y)$, is defined as the maximum over all real numbers $iw_X + jw_Y$ such that $a_{i,j} \neq 0$.*

Throughout this thesis, we only consider cases where $w_X = 1$. Thus we can use the following simplified definition of $(1,w)$-*weighted degree* of $\mathcal{A}(X,Y)$

$$\deg_w \mathcal{A}(X,Y) = \max\{i + jw : a_{i,j} \neq 0\}. \tag{2.4}$$

We let X-deg $\mathcal{A}(X,Y)$, Y-deg $\mathcal{A}(X,Y)$ denote the X-degree, respectively the Y-degree, of $\mathcal{A}(X,Y)$. In addition, we will use the following definition of *weighed-degree order* for bivariate monomials: $X^iY^j \prec_w X^aY^b$ iff

$$\big(i + wj < a + wb\big) \quad \text{or} \quad \big(i + wj = a + wb \ \text{ and } \ j < b\big)$$

Note that if $w < 0$, then $\prec_w$ is *not* a monomial order on the monomials of $\mathbb{F}_q[X,Y]$, since there is no least element. Yet, $\prec_w$ is *always* a monomial order on the set of monomials in

$$\big\langle \mathbb{F}_q[X,Y]\big\rangle_r \stackrel{\text{def}}{=} \big\{A \in \mathbb{F}_q[X,Y] \ : \ Y\text{-deg}\, A \le r\big\}\,. \tag{2.5}$$

Following Lee-O'Sullivan [LO06a,LO06b], we view $\langle \mathbb{F}_q[X,Y]\rangle_r$ as a free module over $\mathbb{F}_q[X]$ generated by the basis $1, Y, Y^2, \ldots, Y^r$.

For reasons that will become clear in Chapter 4, we do not restrict the definition of weighted degree to the usual case [GS99, KV03a, NH98] where w is a nonnegative integer. Thus the weighted degree of a polynomial $\mathcal{A}(X,Y)$ can assume negative values.

We define $K_{\alpha,\beta}$ as follows

$$\begin{aligned} K_{\alpha,\beta} \stackrel{\text{def}}{=} \quad & \textit{the ring of rational functions in } \mathbb{F}_q(X,Y) \\ & \textit{without poles at the point } (\alpha,\beta) \in \mathbb{F}_q \times \mathbb{F}_q. \end{aligned} \tag{2.6}$$

A rational function $\mathcal{A}(X,Y) \in K_{\alpha,\beta}$ has a power-series expansion in basis functions of type $(X-\alpha)^i(Y-\beta)^j$. Thus

$$\mathcal{A}(X,Y) \ = \ \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} a_{i,j}\,(X-\alpha)^i(Y-\beta)^j \tag{2.7}$$

Based on (2.7), we also define the following function:

$$\mathbf{coef}\big(\mathcal{A}(X+\alpha, Y+\beta), X^iY^j\big) \stackrel{\text{def}}{=} a_{i,j}$$

Definition 2.2. *The function $\mathcal{A}(X,Y)$ is said to pass through the point (α,β) with multiplicity m if $a_{i,j} = 0$ for all $i + j < m$ in (2.7). Define the multiplicity function $\mu_{\alpha,\beta} : K_{\alpha,\beta} \to \mathbb{N}$ as follows:*

$$\mu_{\alpha,\beta}\big(\mathcal{A}(X,Y)\big) \stackrel{\text{def}}{=} \max\{m \in \mathbb{N} : a_{i,j} = 0 \ \forall\, i + j < m\}$$

where $a_{i,j}$ are the coefficients in (2.7) and $\mathbb{N}$ is the set of natural numbers.

The following observations are obvious from the definition given above: for any two functions $\mathcal{A}(X,Y)$ and $\mathcal{B}(X,Y)$ in $K_{\alpha,\beta}$, we have

$$\mu_{\alpha,\beta}(\mathcal{A}(X,Y)\mathcal{B}(X,Y)) = \mu_{\alpha,\beta}(\mathcal{A}(X,Y)) + \mu_{\alpha,\beta}(\mathcal{B}(X,Y)) \tag{2.8}$$

$$\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) = \mu_{0,0}(\mathcal{A}(X+\alpha, Y+\beta)) \tag{2.9}$$

$$\mu_{\alpha,\beta}(\mathcal{A}(X,Y)+\mathcal{B}(X,Y)) \geq \min\{\mu_{\alpha,\beta}(\mathcal{A}(X,Y)), \mu_{\alpha,\beta}(\mathcal{B}(X,Y))\} \tag{2.10}$$

Our interest in the foregoing definitions and results is motivated by the fact that, as a consequence of Bezout's theorem [Sha95], two polynomials $\mathcal{A}(X,Y)$ and $\mathcal{B}(X,Y)$ cannot both pass through an arbitrary large number of points without having a common factor. In particular, the polynomial $Y - f(X)$, with deg $f(X)$ $< k$, passes through the n points $(x_1^*, c_1), (x_2^*, c_2), \ldots, (x_n^*, c_n)$, where $c_i = f(x_i^*)$ may be thought of as the n transmitted symbols. Then Bezout's theorem implies that any nonzero polynomial $\mathcal{Q}(X,Y)$ such that

$$\sum_{i=1}^{n} \mu_{x_i^*,c_i}(\mathcal{Q}) > \deg_{1,k-1} \mathcal{Q}(X,Y)$$

is divisible by $Y - f(X)$. This leads to the interpolation-based decoding algorithms of [Sud97], [GS99], [NH98], [KV03a]. The central idea of all these decoding algorithms is to construct a polynomial $\mathcal{Q}(X,Y)$ that passes through a prescribed set of points $\mathcal{P} = \{(x_1,y_1),(x_2,y_2),\ldots,(x_s,y_s)\}$, where $x_1,\ldots,x_s \in \mathcal{D}$ and $y_1,y_2,\ldots,y_s \in \mathbb{F}_q$, with prescribed multiplicities $m_{x_1,y_1}, m_{x_2,y_2}, \ldots, m_{x_s,y_s} \in \mathbb{N}$. If these points and multiplicities agree "sufficiently well" with the n points (x_i^*, c_i) that define the transmitted codeword, then the divisibility of $\mathcal{Q}(X,Y)$ by $Y - f(X)$ is guaranteed [KV03a]. How the two sets $\mathcal{P}$ and $M = \{m_{x_1,y_1}, m_{x_2,y_2}, \ldots, m_{x_s,y_s}\}$ are determined from the channel output is out of the scope of this thesis. Interested readers can find answers to this question in [GS99], [KV03a], [Nie03], [PV03]. In all cases, a key part of the decoding algorithm consists of solving the following interpolation problem.

Original interpolation problem: *Given a set of points* $\mathcal{P} = \{(x_1,y_1),(x_2,y_2),\ldots,(x_s,y_s)\}$ *and a set of multiplicities* $M = \{m_{x_1,y_1}, m_{x_2,y_2}, \ldots, m_{x_s,y_s}\}$, *find a nonzero polynomial* $\mathcal{Q}(X,Y)$ *of minimal* $(1,k-1)$-*weighted degree, such that* $\mu_{x_i,y_i}(\mathcal{Q}) \geq m_{x_i,y_i}$ *for* $i = 1,\ldots,s$.

We shall refer to this interpolation problem as $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$, and say that $\mathcal{Q}(X, Y)$ is a *solution to* $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$. Observe that x_i and x_j in the set $\mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_s, y_s)\}$ do not have to be distinct, all we require is that they belong to $\mathcal{D}$. In fact, in soft-decision decoding, we often interpolate through different points having the same X-coordinate [KV03a, KMVA03]. By definition, requiring that a polynomial $\mathcal{Q}(X, Y)$ passes through a point with multiplicity m imposes $\frac{1}{2}m(m+1)$ linear constraints on the vector space of polynomials in two variables (cf. (2.7)). Hence, solving $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ is tantamount to solving a system of $N(M) = \frac{1}{2}\sum_{i=1}^{s} m_{x_i,y_i}(m_{x_i,y_i} + 1)$ linear (although not necessarily linearly independent) equations. As shown in [GS99, KV03a], there are

$$\nu_{1,k-1}(\delta) = \left\lceil \frac{\delta+1}{k-1} \right\rceil \left(\delta - \frac{k-1}{2} \left\lfloor \frac{\delta}{k-1} \right\rfloor + 1 \right)$$

monomials $X^i Y^j$ with $i + (k-1)j \leq \delta$. Hence, choosing δ to be large enough will guarantee a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$. Let δ^* be the smallest integer such that $\nu_{1,k-1}(\delta^*) > N(M)$. Then $\deg_{1,k-1} \mathcal{Q}(X, Y) \leq \delta^*$, and the Y-degree of $\mathcal{Q}(X, Y)$ can be estimated as $r = \lfloor \delta^*/(k-1) \rfloor$. In principle, $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ is a linear problem that can be solved in a number of ways. In the next section, we present a brief survey of existing works that solve the interpolation problem.

2.3 Brief Survey of Existing Interpolation Algorithms

The material to be presented in this section is somewhat borrowed from the prior work in [LO06a]. Koetter's algorithm [Koe96b], later presented by McEliece [McE03a], is the first solution to the interpolation problem. It solves the interpolation problem by iteratively building a minimal solution for the linear constraints of the interpolation problem. Later, Nielsen and Høholdt came up with an algorithm [NH98] of similar nature, and a divide-and-conquer approach was proposed by Feng in [Fen99]. Based on the theory of Groebner bases, O'Keeffe and Fitzpatrik [OF02] and Farr and Gao [FG05] generalized the interpolation algorithm to decode linear codes. Olshevsky and Shokrollahi [OS99] formulated the interpolation problem as a problem of finding a nonzero element of the kernel of a certain structured matrix and devised an algorithm that solved the problem. Kuijper and Polderman [KP04] proposed an algorithm solving the interpolation problem recast from a system theory point of view. Ruatta and Trebuchet found a general and efficient approach for implicitization [RT03], and it can be applied to the interpolation problem as a specific case. In [Ale06], Alekhnovich generalized the classical Knuth-Schoenhage algorithm computing the greatest common divisor of two polynomials for solving arbitrary linear Diophantine systems over polynomials. The generalized algorithm is effective in solving weighted algebraic curve fitting problem, which is closely related to the interpolation problem. Another interpolation algorithm is proposed by Lee and O'Sullivan [LO06a], [LO06b]. This algorithm adopts a strategy, different from that of [Koe96b], [NH98], [Fen99], to compute the Groebner basis for the polynomial module defined by the interpolation problem.

2.4 Koetter's Interpolation Algorithm

Koetter's algorithm can be described as follows:

Koetter's Interpolation Algorithm

```
• Initialization:
  $\mathcal{Q}_v(X,Y) = \sum_{t=0}^{r} q_{v,t}(X)Y^t$, for $0 \le v \le r$.

• Iteration:
  Input: $\{(x_i, y_i, m_{x_i,y_i}) : (x_i, y_i) \in \mathcal{P}\}$

    – For each triple $(x_i, y_i, m_{x_i,y_i})$,
      $O_v = \deg_w \mathcal{Q}_v(X,Y)$, for $0 \le v \le r$.

      for $a = 0$ to $m_{x_i,y_i} - 1$
          for $b = 0$ to $m_{x_i,y_i} - 1 - a$
              Discrepancy Computation:
```

```
            for $v = 0$ to $r$
                $d_v^{(a,b)} = \mathbf{coef}\big(\mathcal{Q}_v(X + x_i, Y + y_i), X^a Y^b\big)$
            end
            **Polynomial Update:**
            if there exist $\eta = \operatorname{argmin}_{\substack{0 \le v \le r \\ d_v^{(a,b)} \neq 0}} \{O_v\}$
                for $v = 0$ to $r$
                    if $v \neq \eta$ and $d_v^{(a,b)} \neq 0$
                        $\mathcal{Q}_v(X, Y) := \mathcal{Q}_v(X, Y) + \frac{d_v^{(a,b)}}{d_\eta^{(a,b)}} \mathcal{Q}_\eta(X, Y)$
                    end
                end
                $\mathcal{Q}_\eta(X, Y) := \mathcal{Q}_\eta(X, Y)(X - x_i)$, and $O_\eta := O_\eta + 1$
            end
        end
    end
```

- **Result**: $\mathcal{Q}(X, Y) = \{\mathcal{Q}_\eta(X, Y)\}$, where $\eta = \operatorname{argmin}_{0 \le v \le r}\{O_v\}$. So that

$$\mathcal{Q}(X, Y) = \sum_{t=0}^{r} q_t(X) Y^t$$

As it becomes clear later, with appropriate initialization and different w values, the algorithm can also be used to efficiently solve the reduced interpolation problem ($\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$) defined in Chapter 4. It has been shown [McE03a] that the above algorithm actually computes a Groebner basis for the $F[X]$-submodule of all bivariate polynomials of Y-degree not larger than r that pass the interpolation point set with prescribed multiplicities. Proposition 11 of [LO06b] shows that the bivariate polynomials obtained by carrying out Koetter's algorithm actually form a Groebner basis for the ideal of all bivariate polynomials that satisfy the constrains of the interpolation problem. Thus we can treat the interpolation problem in the regime of $F[X, Y]$-ideals or $F[X]$-modules. However, the discussion of $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$ has to be limited to modules since $\prec_w$ is not an appropriate monomial order for bivariate polynomial ideals when $w < 0$. Fortunately, it is always a well-defined monomial order for an $F[X]$-module.

CHAPTER 3

Divide-and-Conquer Interpolation Method

The complexity of Koetter's algorithm (Chapter 2) is proportional to the number of polynomials being updated and to 2nd power of the total cost of interpolation points. Thus one may expect to reduce the complexity if the original interpolation problem is split into a number of smaller ones and later merge the solutions. This is the idea of divide-and-conquer, which is based on splitting the original set of interpolation points into a number of subsets, independently computing their interpolation polynomials and later merging the results. In addition to the desired complexity reduction, this divide-and-conquer approach enables parallel processing in interpolation. Note that the divide-and-conquer approach to be discussed is different from the one proposed in [Fen99], where the smaller interpolation problems are solved sequentially.

3.1 Matrix Interpretation of Koetter's Interpolation Algorithm

Lemma 3.1. *Let $Q_j(X,Y), j = 0,...,r$ be a set of polynomials, produced by Koetter's interpolation algorithm after processing the set $\mathcal{P}$ of interpolation* *points with corresponding multiplicities M. Then all polynomials $\mathcal{Q}(X,Y)$: $\mathrm{wdeg}_{(0,1)}\,\mathcal{Q}(X,Y) \le r$ pass through points in $\mathcal{P}$ with corresponding multiplicities M can be represented as*

$$\mathcal{Q}(X,Y) = \sum_{j=0}^{r} p_j(X)\mathcal{Q}_j(X,Y)$$

Proof. Let $\mathcal{Q}(X,Y)$: $\mathrm{wdeg}_{(0,1)}\,\mathcal{Q}(X,Y) \le r$ be a polynomial having points from $\mathcal{P}$ as roots of corresponding multiplicities from M. The following procedure is very similar both to the multivariate polynomial division [CLO96] and matrix polynomial division algorithms [Kai80, Gan88].

1. Order $\mathcal{Q}(X,Y)$ terms accordingly to $\deg_{1,k-1}$ and let $R(X,Y) = 0, p_j(X) = 0$.

2. Let $\mathrm{LT}\,\mathcal{Q}(X,Y) = \alpha x^a y^b$ and $\mathrm{LT}\,\mathcal{Q}_b(X,Y) = x^c y^b$. (Note that the Koetter's interpolation algorithm guarantees that $\mathrm{LT}\,\mathcal{Q}_b(X,Y) = X^c Y^b$ throughout the whole iterative procedure.) If $a \ge c$, then $p_b(X) := p_b(X) + \alpha x^{a-c}$, $\mathcal{Q}(X,Y) := \mathcal{Q}(X,Y) - \alpha x^{a-c}\mathcal{Q}_b(X,Y)$. Otherwise, $R(X,Y) := R(X,Y) + \mathrm{LT}\,\mathcal{Q}(X,Y)$, $\mathcal{Q}(X,Y) := \mathcal{Q}(X,Y) - \mathrm{LT}\,\mathcal{Q}(X,Y)$.

3. Repeat step 2 until $\mathcal{Q}(X,Y) = 0$.

Clearly, this procedure would lead to $\mathcal{Q}(X,Y) = \sum_j p_j(X)\mathcal{Q}_j(X,Y) + R(X,Y)$. Since all $\mathcal{Q}_j(X,Y)$ have points from $\mathcal{P}$ as roots of corresponding multiplicities M, these points will be roots of the same multiplicity of $R(X,Y)$. Thus we have obtained a polynomial with degree $j = \mathrm{wdeg}_{(0,1)}\,R(X,Y) \le r$ in y, having roots of multiplicities in M at all points of $\mathcal{P}$ such that $\deg_{1,k-1} R(X,Y) < \deg_{1,k-1}\mathcal{Q}_j(X,Y)$, which contradicts to the property of $\mathcal{Q}_j(X,Y)$ minimality (see [NH98] for proof). Thus $R(X,Y) = 0$. This lemma proves that the $\mathcal{Q}_j(X,Y)$ polynomials form a basis of a polynomial module and any polynomial $\mathcal{Q}(X,Y)$

as described in Lemma 3.1 may be represented as algorithm is

$$\mathcal{Q}(X,Y) = \mathcal{Y}\mathcal{Q}(X)P(X) = \begin{pmatrix} 1 & y & \dots & y^r \end{pmatrix} \begin{pmatrix} q_{00}(X) & q_{01}(X) & \dots & q_{0\,r}(X) \\ q_{10}(X) & q_{11}(X) & \dots & q_{1\,r}(X) \\ \dots \\ q_{r\,0}(X) & q_{r\,1}(X) & \dots & q_{r\,r}(X) \end{pmatrix} \begin{pmatrix} p_0(X) \\ p_1(X) \\ \dots \\ p_r(X) \end{pmatrix} \quad (3.1)$$

Then each step of Koetter's interpolation algorithm may be represented as multiplication of matrix polynomial $\mathcal{Q}(X)$ obtained during previous steps by matrix

$$\begin{pmatrix} 1 & 0 & \dots & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & \dots & 0 \\ \dots \\ -\frac{\Delta_0}{\Delta_{j_0}} & -\frac{\Delta_1}{\Delta_{j_0}} & \dots & (x-x_i) & \dots & -\frac{\Delta_r}{\Delta_{j_0}} \\ \dots \\ 0 & 0 & \dots & 0 & \dots & 1 \end{pmatrix}. \quad (3.2)$$

Note that the determinant of this matrix equals to $\delta(x-x_i), \delta \in GF(q)\backslash\{0\}$, which implies that for each extension of the original field $GF(q)$ it is singular only for $x = x_i$. Thus Koetter's interpolation algorithm may be considered as a process of sequential construction of a matrix polynomial $\mathcal{Q}(X)$ whose columns form a basis of a polynomial module.

It can be easily proved that application of Koetter's interpolation algorithm with different order of interpolation points leads to equivalent results. Let us assume that $\{\mathcal{Q}_j^{(1)}(X,Y)\}$ and $\{\mathcal{Q}_j^{(2)}(X,Y)\}$ are interpolation polynomials obtained for the sets $\mathcal{P}_1, \mathcal{P}_2 : \mathcal{P}_1 \bigcap \mathcal{P}_2 = \emptyset$ of interpolation points with associated multiplicities $M1$ and $M2$, respectively. Then application of Koetter's interpolation algorithm to the set of points $\mathcal{P} = \mathcal{P}_1 \bigcup \mathcal{P}_2$, with multiplicity set $M = M1 \bigcup M2$ would lead to polynomials $\{\mathcal{Q}_j(X,Y)\}$ such that

$$\mathcal{Q}(X) = \mathcal{Q}^{(1)}(X)P_1(X) = \mathcal{Q}^{(2)}(X)P_2(X), \quad (3.3)$$

where $\mathcal{Q}(X), \mathcal{Q}^{(s)}(X)$ are matrix polynomials obtained from $\mathcal{Q}_j(X,Y)$ and $\mathcal{Q}_j^{(i)}$ $(X,Y), i = 1,2$ respectively, while $P_i(X)$ are equal to products of matrices (3.2) and some uni-modular matrices (i.e. matrices with determinants in $GF(q)\setminus\{0\}$) of linear transformations required to obtain an equation. Thus the problem of "merging" two sets of interpolation polynomials may be considered as finding a *least common right multiple* of the respective matrix polynomials. Unfortunately, even the most recent algorithms for performing this task (e.g. [BL00]) appear to be too computationally expensive.

3.2 Algebraic-Geometric Interpretation of Koetter's Interpolation algorithm

As it can be seen from (3.3), the divide-and-conquer interpolation method may be considered as intersection of two modules. However, these modules possess certain additional properties which may be used to simplify computations:

1. It is possible to introduce the multiplication operation for two module elements. However, its result belongs to a module of higher dimension.
2. The module may be considered as a subset of bivariate polynomial ideal.

Thus one can perform intersection operation in an ideal which is a superset of a module, and then convert its result back into the module. Actually we have the following lemma.

Lemma 3.2. *Let $\mathcal{Q}_j(X, Y), j = 0, ..., r$ be a set of polynomials, produced by Koetter's interpolation algorithm after processing the set $\mathcal{P}$ of interpolation points with corresponding multiplicities M. If $\mathrm{LT}\, \mathcal{Q}_r(X, Y) = Y^r$, then $\langle \mathcal{Q}_0(X, Y), ..., \mathcal{Q}_r(X, Y)\rangle$ is a basis (actually a Groebner basis) for ideal of polynomials $\mathcal{Q}(X, Y)$ that pass through points in $\mathcal{P}$ with corresponding multiplicities M.*

Proof. Similar to the proof of Lemma 3.1, thus omitted here.

It is possible to generalize the definition of affine variety to accommodate the case of roots with high multiplicity. Then Koetter's interpolation algorithm may be considered as a process of adding interpolation points to an affine variety defined by the ideal of interpolation polynomials. Thus the operation of module intersection may be replaced with intersection of two ideals. But again, the ideal intersection algorithm [CLO96] appears to be too complex. However, in some cases computation of ideal intersection may be replaced with computation of their product.

Let $\mathcal{R} = \mathbb{F}_q[X, Y]$ denote the ring of bivariate polynomials over $\mathbb{F}_q$. Let $\mathcal{I} \subseteq \mathcal{R}$ be a polynomial ideal. Then the quotient ring $\mathcal{R}/\mathcal{I}$ is isomorphic to the $\mathbb{F}_q$-vector space spanned by the set

$$\left\{ X^a Y^b \ : \ X^a Y^b \notin \langle \mathrm{LT}(\mathcal{I}) \rangle \right\}$$

where $\langle \mathrm{LT}(\mathcal{I}) \rangle$ denotes the ideal generated by the leading terms of the elements of $\mathcal{I}$. This is Proposition 4 of [CLO96, p. 229]. We let $\dim_{\mathbb{F}_q} \mathcal{R}/\mathcal{I}$ denote the dimension of this vector space. The *footprint* $\Delta(\mathcal{I})$ of $\mathcal{I}$, also called the *deltaset* of $\mathcal{I}$, can be defined as the set of all monomials in $\mathcal{R}$ that are *not* the leading monomials of elements of $\mathcal{I}$. It is known [HvLP98] that $\dim_{\mathbb{F}_q} \mathcal{R}/\mathcal{I} = |\Delta(\mathcal{I})|$, provided $\Delta(\mathcal{I})$ is finite. However, we will not need this result.

Theorem 3.3 *Let $\mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_s, y_s)\}$ be a set of s distinct points in $\mathbb{F}_q^2$ and let $M = (m_1, m_2, \ldots, m_s)$ be a sequence of positive integers, called the multiplicities of the points in $\mathcal{P}$. Consider the ideal*

$$\langle \mathcal{I}(\mathcal{P}, M) \rangle \stackrel{\mathrm{def}}{=} \left\{ \begin{array}{l} Q(X,Y) \in \mathcal{R} \ : \ \mathsf{coef}\left\{ Q(X{+}x_i, Y{+}y_i), X^a Y^b \right\} = 0 \\ \textit{for } a + b < m_i \end{array} \right\} \qquad (3.4)$$

Following [KV03a]*, define the* cost *of M as $\mathcal{C}(M) = \frac{1}{2}\sum_{i=1}^{s} m_i(m_i+1)$. Then $|\Delta(\mathcal{I})| = \mathcal{C}(M)$. That is, the dimension of $\mathcal{R}/\mathcal{I}$ as a vector space over $\mathbb{F}_q$ is equal to the cost of M.*

Proof. Let $\mathcal{C}(M) = n$. We will construct a bijection between the quotient ring $\mathcal{R}/\mathcal{I}$ and an n-dimensional vector space over $\mathbb{F}_q$. To this end, let us introduce a map $\Phi : \mathcal{R} \to \mathbb{F}_q^n$, defined as follows

$$\Phi(Q) \stackrel{\mathrm{def}}{=} \Big(c_{0,0;1}(Q), c_{1,0;1}(Q), \ \ldots, c_{0,0;s}(Q), c_{1,0;s}(Q), \ldots, c_{0,m_s;s}(Q) \Big)$$

where $c_{a,b;i}(Q) \overset{\text{def}}{=} \mathsf{coef}\{Q(X + x_i, Y + y_i), X^a Y^b\}$ for all nonnegative integers a, b such that $a + b < m_i$ and for all $i = 1, 2, \ldots, s$. Thus $\Phi(Q)$ is precisely the set of coefficients in (3.4), arranged in a fixed order. Specifically, $c_{a',b';j}(Q)$ precedes $c_{a,b;i}(Q)$ in the n-dimensional vector $\Phi(Q)$ iff $j < i$ or $j = i$ and $(a', b') \prec (a, b)$, where $\prec$ is the graded lex order on $\mathbb{N}^2$ (actually, any graded order on $\mathbb{N}^2$ would suffice for our purposes). Now let us define the following polynomials:

$$G_{a,b;i}(X,Y) \overset{\text{def}}{=} (X - x_i)^a (Y - y_i)^b \prod_{x_l \neq x_i} (X - x_l)^{m_i} (Y - y_l)^{m_i} \prod_{\substack{x_l = x_i \\ y_l \neq y_i}} (Y - y_l)^{m_i} \tag{3.5}$$

for all nonnegative integers a, b such that $a + b < m_i$ and for all $i = 1, 2, \ldots, s$. By definition, these polynomials satisfy $c_{a,b;i}(G_{a,b;i}) \neq 0$. Moreover, $c_{a',b';j}(G_{a,b;i}) = 0$ if $j \neq i$ or if $j = i$ and either $a > a'$ or $b > b'$ — in particular, if $(a', b') \prec (a, b)$. Let us arrange the n polynomials $G_{a,b;i}$ in (3.5) in the same order as the n coefficients $c_{a,b;i}(Q)$ in the vector $\Phi(Q)$, and consider the $n \times n$ matrix A having $\Phi(G_{a,b;i})$ as its rows. It follows from the properties of the polynomials $G_{a,b;i}$ that the matrix A is upper triangular. Hence its rows constitute a basis for $\mathbb{F}_q^n$. This implies that for each vector $v \in \mathbb{F}_q^n$, we can construct a polynomial $Q(X,Y) \in \mathcal{R}$ such that $\Phi(Q) = v$. Indeed, if v is expressed as a linear combination of the rows of A, then $Q(X,Y)$ is just the corresponding linear combination of the polynomials $G_{a,b;i}$ in (3.5). This shows that the mapping Φ is surjective.

Now consider the mapping $\Psi : \mathcal{R}/\mathcal{I} \to \mathbb{F}_q^n$ defined by $\Psi([Q]) = \Phi(Q)$, where $[Q]$ is the equivalence class of Q in $\mathcal{R}/\mathcal{I}$. It follows from [CLO96, Chapter 5] that the mapping Ψ is well-defined. Moreover, since Φ is surjective, then so is Ψ. The mapping Ψ, on the other hand, is also injective. Indeed, if $\Psi([Q]) = \Phi(Q) = \mathbf{0}$ in $\mathbb{F}_q^n$, then $Q \in \mathcal{I}$ by (3.4) and the definition of $\Phi(Q)$. Hence $\Psi([Q]) = \mathbf{0}$ if and only if $[Q] = [0] = \mathcal{I}$. Together with the linearity of Ψ, this shows that Ψ is an injection, as claimed. We thus conclude that Ψ is a bijection from $\mathcal{R}/\mathcal{I}$ to $\mathbb{F}_q^n$. Hence $\dim_{\mathbb{F}_q} \mathcal{R}/\mathcal{I} = n$. ∎

Remark. The mapping Ψ constructed in the proof above is, in fact, a vector space isomorphism between $\mathcal{R}/\mathcal{I}$ and $\mathbb{F}_q^n$. Thus $\mathcal{R}/\mathcal{I} \simeq \mathbb{F}_q^n$, for $n = \mathcal{C}(M)$.

Corollary 3.4 *The affine variety* $\mathbf{V}(\mathcal{I})$ *defined by the ideal* $\mathcal{I}$ *in* (3.4) *is equal to*

$\mathcal{P}$.

Proof. Clearly $\mathcal{P} \subseteq \mathbf{V}(\mathcal{I})$ by definition. Assume to the contrary that there exists a point P such that $P \in \mathbf{V}(\mathcal{I})$ but $P \notin \mathcal{P}$. Let $\mathcal{P}' = \mathcal{P} \cup P$, let $M' = (m_1, m_2, \ldots, m_s, 1)$, and let $\mathcal{I}'$ be the ideal defined as in (3.4) in terms of $\mathcal{P}'$ and M'. Then the fact that $P \in \mathbf{V}(\mathcal{I})$ implies that $\mathcal{I}' = \mathcal{I}$. However, this contradicts Theorem 1, since $\mathcal{C}(M') \neq \mathcal{C}(M)$. ▮

Theorem 3.5 *Let $\mathcal{I}(\mathcal{P}, M)$ denote the ideal defined in* (3.4) *with respect to the point set $\mathcal{P}$ and multiplicity vector M. Suppose $\mathcal{P}_1, M_1$ and $\mathcal{P}_2, M_2$ are such that $\mathcal{P}_1 \cap \mathcal{P}_2 = \varnothing$. Then*

$$\mathcal{I}(\mathcal{P}_1, M_1) \cap \mathcal{I}(\mathcal{P}_2, M_2) = \mathcal{I}(\mathcal{P}_1, M_1) \cdot \mathcal{I}(\mathcal{P}_2, M_2) \tag{3.6}$$

Proof. Let $\mathcal{I}_1 = \mathcal{I}(\mathcal{P}_1, M_1)$ and $\mathcal{I}_2 = \mathcal{I}(\mathcal{P}_2, M_2)$. Then $\mathbf{V}(\mathcal{I}_1) = \mathcal{P}_1$ and $\mathbf{V}(\mathcal{I}_2) = \mathcal{P}_2$ by Corollary 2. It follows that

$$\mathbf{V}(\mathcal{I}_1 + \mathcal{I}_2) = \mathbf{V}(\mathcal{I}_1) \cap \mathbf{V}(\mathcal{I}_2) = \mathcal{P}_1 \cap \mathcal{P}_2 = \varnothing$$

Hence, by the weak Nullstellensatz [CLO96, p. 168], we have $\mathcal{I}_1 + \mathcal{I}_2 = \bar{F}[X, Y]$, where $\bar{F}$ is the algebraic closure of $\mathbb{F}_q$. Thus $\mathcal{I}_1$ and $\mathcal{I}_2$ are co-prime, which implies (3.6) by the Chinese Remainder Theorem. ▮

Suppose that the original interpolation point set $\mathcal{P}$ is split into 2 sets, namely $\mathcal{P}_1$ and $\mathcal{P}_2$. The multiplicity set M is divided into set M_1 and M_2, which are associated with $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Apparently, we have $\mathcal{I}(\mathcal{P}, M) = \mathcal{I}(\mathcal{P}_1, M_1)$ $\cap \mathcal{I}(\mathcal{P}_2, M_2)$. Furthermore, according to Theorem 3.5 of Chapter 2, we can compute $cI(\mathcal{P}, M)$ as $\mathcal{I}(\mathcal{P}, M) = \mathcal{I}(\mathcal{P}_1, M_1) \cdot \mathcal{I}(\mathcal{P}_2, M_2)$. Based on [CLO96], the basis of $\mathcal{I}(\mathcal{P}, M)$ can be constructed by taking pairwise product of basis polynomials for ideals $\mathcal{I}(\mathcal{P}_1, M_1)$ and $\mathcal{I}(\mathcal{P}_2, M_2)$. However, the basis created this way is no longer a Groebner basis. Thus one can perform interpolation as follows: Apply Koetter's algorithm to the disjoint sets $\mathcal{P}_1$ and $\mathcal{P}_2$ of interpolation points and obtain basis polynomials $Q_1 = \{Q_j^{(1)}(X, Y) > j = 0..r\}$ and $Q_2 = \{Q_j^{(2)}(X, Y), j = 0..r\}$ (they actually are Groebner bases!). Compute their pairwise product to get a basis of the ideal product and apply an elimination algorithm to reduce this basis to a Groebner basis.

The Divide-and-Conquer Interpolation Algorithm

```
$SplitInterpolation(s, \mathcal{P}, M)$
$\mathcal{P}_1 := \{(x_i, y_i)\}, M_1 := \{m_i\}, i = 1, ..., \lfloor s/2 \rfloor;$
$\mathcal{P}_2 := \{(x_i, y_i)\}, M_2 := \{m_i\}, i = \lfloor s/2 \rfloor + 1, ..., s;$
$\mathcal{Q}_1 := SplitInterpolation(\lfloor s/2 \rfloor, \mathcal{P}_1, M_1);$
$\mathcal{Q}_2 := SplitInterpolation(s - \lfloor s/2 \rfloor, \mathcal{P}_2, M_2);$
$\mathcal{Q}' := \{\mathcal{Q}^{(1)}_{j_1}(X, Y)\mathcal{Q}^{(2)}_{j_2}(X, Y), \mathcal{Q}^{(i)}_{j_i} \in \mathcal{Q}_i\};$
$\mathcal{Q} := Eliminate(Q');$
Return $\mathcal{Q}$
```

3.3 Conclusions

The main difference between our approach and the one suggested in [Fen99] is that interpolation subproblems are solved independently and only afterward their solutions are "merged." This allows one to solve these subproblems in parallel. Moreover, each of the subproblems has a much smaller dimension than the original problem. However, further analysis is required to find an efficient way for eliminating the redundant entries in module bases obtained during the merging step.

Chapter 3 has been presented, at *2004 International Symposium on Information Theory (ISIT)*, Ma, Jun; Trifonov, Peter; Vardy, Alexander. The dissertation author was the primary investigator and author of the paper.

CHAPTER 4

The Re-Encoding Coordinate Transformation Technique

Algebraic soft-decision decoding of Reed-Solomon codes delivers promising coding gains over conventional hard-decision decoding. As mentioned in Chapter 2, the main computational steps in algebraic soft-decoding (as well as Sudan-type list-decoding) are bivariate polynomial interpolation and factorization. In this chapter, we introduce a computational technique, based on re-encoding coordinate transformation, that significantly reduces complexity of bivariate interpolation procedure in algebraic soft decoding. The re-encoding coordinate transformation transfers the original interpolation problem into another *reduced interpolation problem*, which is orders of magnitude smaller than the original one. A rigorous proof is presented to show that the two interpolation problems are equivalent. An efficient factorization procedure that applies directly to the reduced interpolation problem is also given.

4.1 Introduction

Both list-decoding and algebraic soft-decision decoding use interpolation and factorization of bivariate polynomials, which is much more computationally intensive than hard-decision decoding. Various efficient algorithms for interpolation and factorization have been proposed by Augot-Pecquet [AP00], Feng [FG01], Nielsen-Høholdt [NH98], Olshevsky-Shokrollahi [OS99], Roth-Ruckenstein [RR00], and Wu-Siegel [WS01], among others. While polynomial-time, these algorithms fall short of making the required computation feasible in practical applications, involving long high-rate Reed-Solomon codes. In this chapter, we present a series of transformations that drastically reduce the space and time complexity of the interpolation process, by a factor of at least $\frac{n^2}{(n-k)^2}$. The main goal of this chapter is to give a streamlined formulation of this transformation process and of the corresponding factorization procedure.

Koetter's algorithm given in Chapter 2 computes $\mathcal{Q}(X,Y)$ in time $O(rN^2)$, where $N = N(M)$ is the total number of linear equations. While this is substantially faster than straightforward Gaussian elimination, the problem is that the number of equations N is often too large to make an $O(rN^2)$ computation feasible in practice. The following example sheds some light on the magnitude of this problem.

Example 4.1. *Let $\mathbb{C}$ be a RS code of length $n = 255$ and dimension $k = 239$ over $\mathbb{F}_{256}$. A typical interpolation problem arising in the algebraic soft-decision decoding [KV03a] of $\mathbb{C}_q(n,k)$ might involve the following multiplicities:*

multiplicity	*# of points*	*cost*
7	*238*	*6654*
6	*1*	*21*
6	*14*	*294*
5	*2*	*30*
4	*2*	*20*
3	*1*	*6*
2	*1*	*3*
1	*3*	*3*

for a total of $N = 7,031$ linear equations. The "cost" column illustrate the number of linear equations associated with all points of the multiplicities given in the "multiplicity" column. The corresponding value of δ^* can be found to be 1711, and the required Y-degree of $\mathcal{Q}(X,Y)$ is $r = 7$. Computing $\mathcal{Q}(X,Y)$ with the fast algorithms of [FG01] [NH98] [AKS03a] [GKKG05] thus takes about

4×10^8 *finite-field operations.* □

This example illustrates a major problem with interpolation-based decoding. While, for a fixed maximal multiplicity, the complexity of decoding is bounded by a polynomial in the length of the code, the actual complexity of computing $\mathcal{Q}(X, Y)$ is prohibitively large in practice. In the next section, we will introduce a technique based on reencoding and coordinate transformation that drastically reduce this complexity. Before describing the technique in detail, we use following example to illustrate the magnitude of savings in computational complexity that can be achieved with the technique .

Example 4.2. *Consider again the situation of Example 4.1. Judiciously choosing the re-encoding point set, we can eliminate the first* 3 *rows of the interpolation point list table in Example 4.1, i.e., the* 238 *interpolation points of multiplicity 7, and 1 out the 15 points of multiplicity 6. In other words, rather than solving the* $7,031$ *linear equations, we have reduced the problem to efficiently solving only 356 equations, which requires approximately* 8.9×10^5 *finite-field operations. This is a much more feasible task. This reduction in complexity by a factor of* 390 *is augmented by a corresponding reduction in memory requirements, due to the fact that the polynomials carried in the interpolation procedure have very small degree. In this case, the resulting polynomials have a maximum degree of about* 50*, instead of about* 1711 *before the complexity reduction approach.* □

The chapter is organized as follows: In Section 4.2, we present a birational mapping, which is the foundation of the transformation technique to be introduced in this chapter. The re-encoding and coordinate transformation technique is introduced in Section 4.3. Section 4.3 also presents a rigorous proof of the validity of the complexity reduction technique. In Section 4.4, we show how the factorization procedure involved in the decoding shall be carried out to take advantage of the complexity-reduced interpolation procedure. Conclusions are drawn in Section 4.5.

4.2 Birational Mapping

The transformation technique to be introduced later utilizes the following mappings between points in space and bivariate rational functions. bivariate rational functions can be considered as a generalization of bivariate polynomials.

Definition 4.1. *Fix a polynomial $g(X)$ of degree k over $\mathbb{F}_q$. Let $\mathscr{X}$ be the set of $\alpha \in \mathbb{F}_q$ such that $g(\alpha) = 0$. We define the following mapping pair between points in $(\mathbb{F}_q - \mathscr{X}) \times \mathbb{F}_q$:*

$$\varphi_g \colon (x, y) \to \big(x, \frac{y}{g(x)}\big) \tag{4.1}$$

$$\varphi_g^{-1} \colon (x, z) \to \big(x, zg(x)\big) \tag{4.2}$$

It is easy to see that φ_g and φ_g^{-1} are birational isomorphisms. We also define the following mapping between rational functions.

Definition 4.2. *Given the $g(X)$ and $\mathscr{X}$ defined above. We define the following mapping pair between rational functions:*

$$\Phi_g : \mathcal{A}(X, Y) = \sum_{j=0}^{\infty} \sum_{i=0}^{\infty} q_{i,j} X^i Y^j \to \mathcal{B}(X, Z) = \sum_{j=0}^{\infty} \sum_{i=0}^{\infty} q_{i,j} X^i g(X)^j Z^j$$

$$\Phi_g^{-1} : \mathcal{B}(X, Z) = \sum_{j=0}^{\infty} \sum_{i=0}^{\infty} q_{i,j} X^i Z^j \to \mathcal{A}(X, Y) = \sum_{j=0}^{\infty} \frac{\sum_{i=0}^{\infty} q_{i,j} X^i}{g(X)^j} Y^j$$

We say that $\mathcal{B}(X, Z)$ is the image of $\mathcal{A}(X, Y)$ under Φ_g and write $\mathcal{B}(X, Z) = \Phi_g(\mathcal{A}(X, Y))$, and correspondingly for the inverse mapping we write $\Phi_g^{-1}(\mathcal{B}(X, Z)) = \mathcal{A}(X, Y)$.

Theorem 4.3 *For all points $(\alpha, \beta) \in (\mathbb{F}_q - \mathscr{X}) \times \mathbb{F}_q$, and for all $\mathcal{A}(X, Y) \in K_{\alpha,\beta}$, let $(\alpha, \gamma) = \varphi_g(\alpha, \beta)$ and $\mathcal{B}(X, Z) = \Phi_g(\mathcal{A}(X, Y))$, we have $\mu_{\alpha,\beta}(\mathcal{A}(X, Y)) = \mu_{\alpha,\gamma}(\mathcal{B}(X, Z))$.*

The proof of the theorem is given in the appendix. Based on our proof, McEliece formulated a more systematic proof in [McE03b].

4.3 Complexity Reducing Transformation

In this section, we introduce the idea of re-encoding coordinate transformation step by step. A detailed examples is given to illustrate the procedures involved.

4.3.1 Re-Encoding and Shift

Rather than seeking an efficient way to solve $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$, we will modify the interpolation problem itself, by means of a shift and a coordinate transformation. Our approach is similar to the re-encoding idea of the Berlekamp-Welch [WB86]. Given the set $\mathcal{P} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_s, y_s)\}$, we will identify some k points $(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \ldots, (x_{i_k}, y_{i_k})$ in $\mathcal{P}$ such that $x_{i_1}, x_{i_2}, \ldots, x_{i_k} \in \mathcal{D}$ are all distinct. Define $\mathcal{R} = \{(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \ldots, (x_{i_k}, y_{i_k})\}$. Observe that if $\mathcal{P}$ contains at most $n - e < k$ points with distinct X-coordinates, then the resulting polynomial $\mathcal{Q}(X, Y)$ will have at least $q^{e-(n-k)}$ factors of type $Y - f(X)$. This situation corresponds to $e > n-k$ erasures, in which case the transmitted codeword cannot be uniquely determined. This shows that, unless the interpolation problem $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ is ill-conditioned by too many erasures, a set $\mathcal{R}$ with the required property always exists. In fact, there will usually be exponentially many ways to choose $\mathcal{R}$ from $\mathcal{P}$. As far as the theory developed in this chapter is concerned with, the choice of $\mathcal{R}$ is arbitrary. In practice, the set $\mathcal{R}$ will be chosen to consist of the points with the highest possible multiplicities (cf. Example 2). To simplify notation in what follows, we assume without loss of generality that $\mathcal{R}$ consists of the first k points of $\mathcal{P}$, that is $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$. The set $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$ determines a *re-encoding polynomial* $h(X)$ of degree $< k$, defined by

$$h(x_i) \;=\; y_i \qquad \text{for all } (x_i, y_i) \in \mathcal{R} \tag{4.3}$$

Note that the codeword $\underline{c}'$ obtained by evaluating $h(X)$ at $x_1^*, x_2^*, \ldots, x_n^*$ agrees with the “given” values $y_1, y_2, \ldots, y_k$ at the k positions corresponding to $x_1, x_2,$

$\ldots, x_k$. Thus computing $h(X)$ is equivalent to re-encoding through k given values at some k positions. If these k positions are consecutive and $\mathbb{C}_q(n,k)$ is cyclic, this can be achieved through division by the generator polynomial for $\mathbb{C}_q(n,k)$. Otherwise, such re-encoding is tantamount to correcting $n{-}k$ erasures in $\mathbb{C}_q(n,k)$. Various efficient algorithms for this purpose are known. Given the set $\mathcal{P} = \{(x_1,y_1),(x_2,y_2),\ldots,(x_s,y_s)\}$ and the re-encoding polynomial $h(X)$, we define

$$\mathcal{P}' \overset{\text{def}}{=} \Big\{\big(x_1, y_1{-}h(x_1)\big),\ldots,\big(x_k, y_s{-}h(x_s)\big)\Big\} \tag{4.4}$$

Notice that, by the definition of $h(X)$ in (4.3), the first k points in $\mathcal{P}'$ are of the form $(x_1,0),(x_2,0),\ldots,(x_k,0)$.

Theorem 4.4. *Let $\mathcal{Q}'(X,Y)$ be a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$. Then $\mathcal{Q}'(X,Y{-}h(X))$ is a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P},M)$.*

Proof. Consider an arbitrary point $(\alpha,\beta)\in\mathcal{P}$ and the corresponding point (α,β') $\in\mathcal{P}'$, where $\beta'=\beta-h(\alpha)$. Since $\mathcal{Q}'(X,Y)$ is a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$, (α,β') can not be a pole of $\mathcal{Q}'(X,Y)$. Similar to (2.7), the polynomial $\mathcal{Q}'(X,Y)$ can be expanded as

$$\mathcal{Q}'(X,Y) \;=\; \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q'_{i,j}\,(X-\alpha)^i\,(Y-\beta')^j$$

where $q'_{i,j}=0$ for all $i+j<m_{\alpha,\beta}$. Since $h(X)-h(\alpha)$ vanishes at α, the function $h_\alpha(X)=\frac{(h(X)-h(\alpha))}{(X-\alpha)}$ is a polynomial. Let $\mathcal{Q}(X,Y)=\mathcal{Q}'(X,Y{-}h(X))$. Then

$$\begin{aligned}\mathcal{Q}(X,Y) &= \sum_{i,j=0}^{\infty} q'_{i,j}\,(X-\alpha)^i\big(Y-h(X)-(\beta{-}h(\alpha)\big)^j\\ &= \sum_{i,j=0}^{\infty} q'_{i,j}\,(X-\alpha)^i\big((Y{-}\beta)-(h(X){-}h(\alpha)\big)^j\\ &= \sum_{i,j=0}^{\infty} q'_{i,j}\,(X-\alpha)^i\big((Y{-}\beta)-(X{-}\alpha)h_\alpha(X)\big)^j\end{aligned}$$

Since $q'_{i,j}=0$ for all $i+j<m_{\alpha,\beta}$, each *nonzero* term above passes through the point (α,β) with multiplicity at least $i+j\ge m_{\alpha,\beta}$. Therefore, for each $(\alpha,\beta)\in\mathcal{P}$, the polynomial $\mathcal{Q}(X,Y)=\mathcal{Q}'(X,Y{-}h(X))$ passes through the point (α,β) with multiplicity at least $m_{\alpha,\beta}$.

What remains to be proved is that $\mathcal{Q}(X,Y)$ is of minimum $(1,k-1)$-weighted degree. It is easy to observe that $\deg_{1,k-1}\mathcal{Q}(X,Y) = \deg_{1,k-1}\mathcal{Q}'(X,Y)$ since $\deg h(X) \leq k-1$ thus $\deg_{1,k-1} Y = \deg_{1,k-1}(Y-h(X))$. Let us assume that there exists $\mathcal{B}(X,Y)$, which passes all points in $\mathcal{P}$ with required multiplicities and is of smaller $(1,k-1)$-weighted degree than $\mathcal{Q}(X,Y)$. Thus we can determine another polynomial $\mathcal{B}'(X,Y) = \mathcal{B}(X,Y+h(X))$ that passes all points in $\mathcal{P}'$ with required multiplicities. This is because by our assumption, we can write $\mathcal{B}(X,Y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} b_{i,j}(X-\alpha)^i(Y-\beta)^j$ with $b_{i,j} = 0$ for all $i+j < m_{\alpha,\beta}$. Then we have $\mathcal{B}'(X,Y) = \mathcal{B}(X,Y+h(X)) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} b_{i,j}(X-\alpha)^i\big(Y+h(X)-\beta-h(\alpha)+h(\alpha)\big)^j = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} b_{i,j}(X-\alpha)^i\big((Y-\beta')+h(X)-h(\alpha)\big)^j = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} b_{i,j}(X-\alpha)^i\big((Y-\beta')+(X-\alpha)h_\alpha(X)\big)^j$. In addition, we have $\deg_{1,k-1}\mathcal{B}'(X,Y) = \deg_{1,k-1}\mathcal{B}(X,Y)$ and thus $\deg_{1,k-1}\mathcal{B}'(X,Y) < \deg_{1,k-1}\mathcal{Q}'(X,Y)$ by our assumption. This contradicts the fact that $\mathcal{Q}'(X,Y)$ is the minimal solution to the shifted interpolation problem. ∎

From the proof of the above theorem, we actually can obtain the following corollary.

Corollary 4.5 *$\mathcal{Q}'(X,Y)$ is a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$ if and only if $\mathcal{Q}(X,Y) = \mathcal{Q}'(X,Y-h(X))$ is a solution to $\mathbf{IP}_{1,k-1}(\mathcal{P},M)$.*

Example 4.6 *Let $q = 2^3$, so that $\mathbb{F}_q = \mathbb{F}_8 = \{0,1,\alpha,\alpha^2,\alpha^3,\alpha^4,\alpha^5,\alpha^6\}$, where α is the primitive element of $\mathbb{F}_8$ defined by primitive polynomial X^3+X+1. We take $\mathcal{D} = \{1,\alpha,\alpha^2,\alpha^3\}$ and consider the RS code $\mathbb{C}_8(4,2)$ defined as*

$$\mathbb{C}_8(4,2) \overset{\text{def}}{=} \Big\{ (f(1), f(\alpha), f(\alpha^2), f(\alpha^3)) : f(X) = a+bX \ \textit{ with } a,b \in \mathbb{F}_8 \Big\} \quad (4.5)$$

Suppose that the codeword $\underline{c} = (1,\alpha^4,\alpha^3,\alpha)$ corresponding to $f(X) = \alpha^6 + \alpha^2 X$ was transmitted, and we receive the following maximum-likelihood hard-decision vector $\underline{r} = (\alpha,\alpha^4,\alpha^6,1)$, which corresponds to hard-decision errors in positions 1, 2 and 4. For the special case of our example, the soft-decision *demodulator produces the following interpolation point set $\mathcal{P}$:*

point (x, v)	(α, α^4)	(α^2, α^6)	$(\alpha^3, 1)$	(α^3, α)	$(1, \alpha)$	$(1, 1)$	(α^2, α^3)
multiplicity	2	1	1	1	1	1	1

(4.6)

From the number of constrains defined in (4.6), we can determine that the Groebner basis in the algorithm given in the previous section has to be initialized with 4 bivariate polynomials, i.e., $r = 3$ as following:

$$\mathcal{A}_0(X,Y) = 1; \mathcal{A}_1(X,Y) = Y; \mathcal{A}_2(X,Y) = Y^2; \mathcal{A}_3(X,Y) = Y^3$$

We have $w = 1$, and the Groebner-basis interpolation algorithm proceeds as follows.

i	(x_i, y_i, m_{x_i,y_i})	$\mathcal{A}_v(X,Y)$ *for* $v = 0,1,2,3$ *after each iteration*
1	$(\alpha, \alpha^4, 2)$	$\mathcal{A}_0(X,Y) = (\alpha + X)$ $\mathcal{A}_1(X,Y) = \alpha^4 + Y$ $\mathcal{A}_2(X,Y) = \alpha + Y^2$ $\mathcal{A}_3(X,Y) = \alpha^5 + Y^3$
		$\mathcal{A}_1(X,Y) = \alpha^4 + Y$ $\mathcal{A}_0(X,Y) = (\alpha^2 + X^2)$ $\mathcal{A}_2(X,Y) = \alpha + Y^2$ $\mathcal{A}_3(X,Y) = \alpha^5 + Y^3$
		$\mathcal{A}_0(X,Y) = (\alpha^2 + X^2)$ $\mathcal{A}_1(X,Y) = (\alpha^5 + \alpha^4 X) + Y(\alpha + X)$ $\mathcal{A}_2(X,Y) = \alpha + Y^2$ $\mathcal{A}_3(X,Y) = Y(\alpha) + Y^3$
2	$(\alpha^2, \alpha^6, 1)$	$\mathcal{A}_1(X,Y) = (\alpha^6 + \alpha^4 X + \alpha^6 X^2) + Y(\alpha + X)$ $\mathcal{A}_2(X,Y) = (\alpha^3 + \alpha^5 X^2) + Y^2$ $\mathcal{A}_0(X,Y) = (\alpha^4 + \alpha^2 X + \alpha^2 X^2 + X^3)$ $\mathcal{A}_3(X,Y) = (\alpha^6 + \alpha^4 X^2) + Y(\alpha) + Y^3$

3	$(\alpha^2, \alpha^3, 1)$	$\mathcal{A}_2(X,Y) = (\alpha^4 + \alpha^4 X + \alpha X^2) + Y(\alpha + X) + Y^2$ $\mathcal{A}_0(X,Y) = (\alpha^4 + \alpha^2 X + \alpha^2 X^2 + X^3)$ $\mathcal{A}_1(X,Y) = (\alpha + \alpha^2 X^2 + \alpha^6 X^3) + Y(\alpha^3 + \alpha^4 X + X^2)$ $\mathcal{A}_3(X,Y) = (\alpha^3 + \alpha^2 X) + Y(\alpha^5 + \alpha^5 X) + Y^3$
4	$(\alpha^3, 1, 1)$	$\mathcal{A}_0(X,Y) = (1 + \alpha^3 X + X^3) + Y(\alpha^2 + \alpha X) + Y^2(\alpha)$ $\mathcal{A}_1(X,Y) = (\alpha^5 + \alpha^6 X + \alpha^5 X^2 + \alpha^6 X^3) + Y(\alpha X + X^2)$ $+Y^2(\alpha^2)$ $\mathcal{A}_2(X,Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2)$ $+Y^2(\alpha^3 + X)$ $\mathcal{A}_3(X,Y) = (\alpha^3 + \alpha^2 X) + Y(\alpha^5 + \alpha^5 X) + Y^3$
5	$(\alpha^3, \alpha, 1)$	$\mathcal{A}_1(X,Y) = (\alpha^4 + \alpha^4 X + \alpha^5 X^2 + \alpha^2 X^3) + Y(\alpha^2 + X^2) +$ $Y^2(\alpha^4)$ $\mathcal{A}_2(X,Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2)$ $+Y^2(\alpha^3 + X)$ $\mathcal{A}_3(X,Y) = (\alpha^4 + \alpha^6 X^3) + Y(\alpha^6 + \alpha^4 X) + Y^2 + Y^3$ $\mathcal{A}_0(X,Y) = (\alpha^3 + \alpha^2 X + \alpha^3 X^2 + \alpha^3 X^3 + X^4)$ $+Y(\alpha^5 + \alpha X + \alpha X^2) + Y^2(\alpha^4 + \alpha X)$
6	$(1, \alpha, 1)$	$\mathcal{A}_2(X,Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2)$ $+Y^2(\alpha^3 + X)$ $\mathcal{A}_3(X,Y) = (\alpha^5 + X + \alpha X^2 + \alpha X^3)$ $+Y(\alpha + \alpha^4 X + \alpha^3 X^2) + Y^3$ $\mathcal{A}_0(X,Y) = (\alpha^2 + \alpha^3 X + \alpha^4 X^2 + X^4)$ $+Y(\alpha^2 + \alpha X) + Y^2(1 + \alpha X)$ $\mathcal{A}_1(X,Y) = (\alpha^4 + X^2 + \alpha^3 X^3 + \alpha^2 X^4)$ $+Y(\alpha^2 + \alpha^2 X + X^2 + X^3) + Y^2(\alpha^4 + \alpha^4 X)$
7	$(1, 1, 1)$	$\mathcal{A}_2(X,Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2)$ $+Y^2(\alpha^3 + X)$ $\mathcal{A}_3(X,Y) = (\alpha^5 + X + \alpha X^2 + \alpha X^3)$ $+Y(\alpha + \alpha^4 X + \alpha^3 X^2) + Y^3$ $\mathcal{A}_1(X,Y) = (\alpha^4 + X^2 + \alpha^3 X^3 + \alpha^2 X^4)$ $+Y(\alpha^2 + \alpha^2 X + X^2 + X^3) + Y^2(\alpha^4 + \alpha^4 X)$ $\mathcal{A}_0(X,Y) = (\alpha^2 + \alpha^5 X + \alpha^6 X^2 + \alpha^4 X^3 + X^4 + X^5)$ $+Y(\alpha^2 + \alpha^4 X + \alpha X^2) + Y^2(1 + \alpha^3 X + \alpha X^2)$

After the last iteration in the above interpolation procedure, we select the following polynomial with the minimum $(1,1)$*-weighted degree in the list as a solution to the original interpolation problem.*

$$\mathcal{A}(X,Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2) + Y^2(\alpha^3 + X)$$

The total number of iterations required is 9. It is easy to verify that $\mathcal{A}(X,Y)$

passes all points listed above with associated multiplicities and that $Y - (\alpha^6 + \alpha^2 X)$ *is a factor of* $\mathcal{A}(X,Y)$. *Actually a complete factorization of* $\mathcal{A}(X,Y)$ *is as follows:*

$$\mathcal{A}(X,Y) = (\alpha^3 + X)\big(Y - (\alpha^6 + \alpha^2 X)\big)\big(Y - (\alpha^5 + \alpha^6 X)\big)$$

Now let us apply the re-encoding technique discussed in this section. We can find $h(X) = \alpha^5 + \alpha^6 X$, *such that* $\underline{c}_R = (h(1), h(\alpha), h(\alpha^2), h(\alpha^3)) = (\alpha, \alpha^4, \alpha^6, \alpha^3)$ *agrees with* $\underline{r}$ *in position 1, 2, and 3. Correspondingly the original interpolation point set* $\mathcal{P}$ *are modified to the shifted interpolation point set* $\mathcal{P}'$ *as following:*

point (x,y)	$(\alpha,0)$	$(\alpha^2,0)$	(α^3,α)	$(\alpha^3,1)$	$(1,0)$	$(1,\alpha^3)$	(α^2,α^4)
multiplicity	2	1	1	1	1	1	1

, (4.7)

where $y_i = v_i - h(x_i)$, *for* $i = 1, 2, ..., 7$. *The Groebner-basis interpolation algorithm can be carried out as follows.*

i	(x_i, y_i, m_{x_i,y_i})	$\mathcal{B}'_v(X,Y)$ *for* $v = 0,1,2,3$ *after each iteration*
1	$(\alpha, 0, 2)$	$\mathcal{B}'_0(X,Y) = \alpha + X$ $\mathcal{B}'_1(X,Y) = Y$ $\mathcal{B}'_2(X,Y) = Y^2$ $\mathcal{B}'_3(X,Y) = Y^3$
		$\mathcal{B}'_0(X,Y) = \alpha + X$ $\mathcal{B}'_1(X,Y) = (\alpha + X)Y$ $\mathcal{B}'_2(X,Y) = Y^2$ $\mathcal{B}'_3(X,Y) = Y^3$
		$\mathcal{B}'_0(X,Y) = \alpha^2 + X^2$ $\mathcal{B}'_1(X,Y) = (\alpha + X)Y$ $\mathcal{B}'_2(X,Y) = Y^2$ $\mathcal{B}'_3(X,Y) = Y^3$
2	$(\alpha^2, 0, 1)$	$\mathcal{B}'_0(X,Y) = \alpha^4 + \alpha^2 X + \alpha^2 X^2 + X^3$ $\mathcal{B}'_1(X,Y) = (\alpha + X)Y$ $\mathcal{B}'_2(X,Y) = Y^2$ $\mathcal{B}'_3(X,Y) = Y^3$

3	$(\alpha^3, \alpha, 1)$	$\mathcal{B}'_2(X,Y) = Y(\alpha^2+\alpha X)+Y^2$
		$\mathcal{B}'_0(X,Y) = (\alpha^4+\alpha^2 X+\alpha^2 X^2+X^3)+Y(\alpha^5+\alpha^4 X)$
		$\mathcal{B}'_1(X,Y) = Y(\alpha^4+X+X^2)$
		$\mathcal{B}'_3(X,Y) = Y(\alpha^3+\alpha^2 X)+Y^3$
4	$(\alpha^3, 1, 1)$	$\mathcal{B}'_0(X,Y) = (\alpha^4+\alpha^2 X+\alpha^2 X^2+X^3)+Y(\alpha+X)$ $+Y^2(\alpha^4)$
		$\mathcal{B}'_1(X,Y) = Y(\alpha^4+X+X^2)$
		$\mathcal{B}'_2(X,Y) = Y(\alpha^5+\alpha X+\alpha X^2)+Y^2(\alpha^3+X)$
		$\mathcal{B}'_3(X,Y) = Y(\alpha^2+\alpha X)+Y^2(\alpha^3)+Y^3$
5	$(1, 0, 1)$	$\mathcal{B}'_1(X,Y) = Y(\alpha^4+X+X^2)$
		$\mathcal{B}'_2(X,Y) = Y(\alpha^5+\alpha X+\alpha X^2)+Y^2(\alpha^3+X)$
		$\mathcal{B}'_3(X,Y) = Y(\alpha^2+\alpha X)+Y^2(\alpha^3)+Y^3$
		$\mathcal{B}'_0(X,Y) = (\alpha^4+\alpha X+\alpha^6 X^3+X^4)+Y(\alpha+\alpha^3 X+X^2)$ $+Y^2(\alpha^4+\alpha^4 X)$
6	$(1, \alpha^3, 1)$	$\mathcal{B}'_2(X,Y) = Y(\alpha^4+X+X^2)+Y^2(\alpha^3+X)$
		$\mathcal{B}'_3(X,Y) = Y(\alpha+\alpha^3 X+X^2)+Y^2(\alpha^3)+Y^3$
		$\mathcal{B}'_0(X,Y) = (\alpha^4+\alpha X+\alpha^6 X^3+X^4)+Y(\alpha+\alpha^3 X+X^2)$ $+Y^2(\alpha^4+\alpha^4 X)$
		$\mathcal{B}'_1(X,Y) = Y(\alpha^4+\alpha^5 X+X^3)$
7	$(\alpha^2, \alpha^4, 1)$	$\mathcal{B}'_2(X,Y) = Y(\alpha^4+X+X^2)+Y^2(\alpha^3+X)$
		$\mathcal{B}'_3(X,Y) = Y(\alpha+\alpha^3 X+X^2)+Y^2(\alpha^3)+Y^3$
		$\mathcal{B}'_1(X,Y) = (\alpha^4+\alpha X+\alpha^6 X^3+X^4)$ $+Y(\alpha^2+\alpha^2 X+X^2+X^3)+Y^2(\alpha^4+\alpha^4 X)$
		$\mathcal{B}'_0(X,Y) = (\alpha^6+\alpha^6 X+\alpha X^2+\alpha X^3+X^4+X^5)$ $+Y(\alpha^3+\alpha^6 X+\alpha^5 X^2+X^3)$ $+Y^2(\alpha^6+\alpha^3 X+\alpha^4 X^2)$

After the last iteration in the above interpolation procedure, we can obtain the following $\mathcal{B}'(X,Y)$ with the minimum $(1,1)$-weighted degree.

$$\mathcal{B}'(X,Y) = Y(\alpha^4+X+X^2)+Y^2(\alpha^3+X)$$

And it is easy to verify that $\mathcal{B}'(X,Y)$ passes through all points in (4.7) with associated multiplicities. $\mathcal{B}'(X,Y)$ can be factorized as follows:

$$\mathcal{B}'(X,Y) = (\alpha^3+X)Y\big(Y-(\alpha+X)\big)$$

Let us now shift $\mathcal{B}'(X,Y)$ with the re-encoding polynomial $h(X)$ to obtain the following polynomial:

$$\begin{aligned}\mathcal{B}(X,Y) &= \mathcal{B}'(X,Y-h(X)) = \mathcal{B}'\big(X,Y-(\alpha^5+\alpha^6 X)\big)\\ &= (1+\alpha^5 X+\alpha X^3)+Y(\alpha^4+X+X^2)+Y^2(\alpha^3+X)\end{aligned}$$

The resulting $B(X, Y)$ is the same as $\mathcal{A}(X, Y)$ found by direct interpolation through the original point set $\mathcal{P}$. □

It is worth mentioning that we have used the following monomial order to break ties between monomials in Koetter's interpolation algorithm when their weighted degree is identical

$$X^{i_1}Y^{j_1} > X^{i_2}Y^{j_2} \text{ if } i_1 + (k-1)j_1 = i_2 + (k-1)j_2 \text{ and } j_1 > j_2.$$

This monomial order ensures that the mapping $Y \to Y - h(X)$, or vice versa, preserves the monomial order. Otherwise, $\mathcal{Q}'(X, Y) = \mathcal{Q}(X, Y - h(X))$ would have different monomial order from $\mathcal{Q}(X, Y)$. And the solution to the $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ obtained by applying the mapping to a solution to the $\mathbf{IP}_{1,k-1}(\mathcal{P}', M)$ may be different from the solution obtained by carrying out Koetter's interpolation algorithm directly on the points defined by $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$. For example, if monomial order $X > Y$, instead of $Y > X$, is used in example 4.6, it is very likely that the resulting $\mathcal{B}(X, Y)$ is different from the $\mathcal{A}(X, Y)$. Correspondingly, the following monomial order is applied in later discussion of interpolation after re-encoding coordinate transformation:

$$X^{i_1}Z^{j_1} > X^{i_2}Z^{j_2} \text{ if } i_1 - j_1 = i_2 - j_2 \text{ and } j_1 > j_2.$$

Let the symbol $[\cdot]^+$ be defined as $[i]^+ = \max\{i, 0\}$. We can now proceed with the complexity reducing transformations. Let us start with a lemma.

Lemma 4.7. *The polynomial $\mathcal{A}(X, Y) = \sum_{j=0}^{\infty} a_j(X)Y^j$ passes through a point $(\alpha, 0)$ with multiplicity m if and only if the univariate polynomials $a_j(X)$ are divisible by $(X - \alpha)^{[m-j]^+}$.*

Proof. Expand $a_j(X)$ in the basis functions $(X - \alpha)^j$, that is write $a_j(X)$ as $a_j(X) = \sum_{i=0}^{\infty} a_{i,j}(X - \alpha)^i$. Then the expansion (2.7) of $\mathcal{A}(X, Y)$ at the point $(\alpha, 0)$ is given by

$$\mathcal{A}(X, Y) = \sum_{i,j=0}^{\infty} a_{i,j}(X - \alpha)^i(Y - 0)^j = \sum_{i,j=0}^{\infty} a_{i,j}(X - \alpha)^i Y^j$$

Clearly, the polynomial $a_j(X)$ is divisible by $(X-\alpha)^{[m-j]^+}$ if and only if $a_{i,j}=0$ for all $i<[m-j]^+$. This is just a reformulation of the definition of multiplicity. ∎

Lemma 4.7 can be easily extended to the following corollary.

Corollary 4.8. *The polynomial $\mathcal{A}(X,Y)=\sum_{j=0}^{\infty}a_j(X)Y^j$ passes through the k points $(x_1,0),(x_2,0),\ldots,(x_k,0)$ with multiplicities $m_{x_1,y_1},m_{x_2,y_2},\ldots,m_{x_k,y_k}$ if and only if all the polynomials $a_j(X)$ are divisible by $\prod_{i=1}^{k}(X-x_i)^{[m_{x_i,y_i}-j]^+}$.*

From Corollary 4.8, we know that the solution $\mathcal{Q}'(X,Y)$ to the interpolation problem $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$ must have the form

$$\mathcal{Q}'(X,Y)=\sum_{j=0}^{r}\left(b_j(X)\prod_{i=1}^{k}(X-x_i)^{[m_{x_i,y_i}-j]^+}\right)Y^j \tag{4.8}$$

This hints us that we can save the number of required iterations in the Groebner-basis interpolation algorithm, if we initialize the Groebner-basis polynomial as follows

$$\mathcal{Q}'_v(X,Y)=\prod_{i=1}^{k}(X-x_i)^{[m_{x_i,y_i}-v]^+}Y^v,\ \text{for } 0\le v\le r,$$

where $r=\left\lfloor\frac{\delta*}{k-1}\right\rfloor$ is the Y-degree of $Q'(X,Y)$ as defined in Section 2.2 of Chapter 2.

Example 4.9 (Continue from Example 4.6) *In this case, the Groebner basis polynomials should be initialized as following*

$$\mathcal{B}'_0(X,Y)=(X-\alpha)^2(X-\alpha^2)$$

$$\mathcal{B}'_1(X)=(X-\alpha)Y$$

$$\mathcal{B}'_2(X,Y)=Y^2;\mathcal{B}'_3(X,Y)=Y^3$$

It is easy to observe that the Groebner basis polynomials above are directly initialized to be equal to the polynomials obtained after 4 iterations in Example 4.6. Thus in this case, the Groebner-basis interpolation algorithm proceeds as *iterations 5 to 9 in Example 4.6. And we can finally obtain the same polynomial solution*

$$\mathcal{B}'(X,Y) = Y(\alpha^4 + X + X^2) + Y^2(\alpha^3 + X)$$

However, the number of iterations required is only 5 instead of 9. $\square$

4.3.2 Coordinate Transformation

Now, let the auxiliary polynomials $g(X)$, $\Phi(X)$, and the "tail" polynomials $T_j(X)$ be defined as follows:

$$g(X) \overset{\text{def}}{=} \prod_{i=1}^{k}(X - x_i) \tag{4.9}$$

$$\Phi(X) \overset{\text{def}}{=} \prod_{i=1}^{k}(X - x_i)^{m_{x_i,y_i}} \tag{4.10}$$

$$T_j(X) \overset{\text{def}}{=} \prod_{i=1}^{k}(X - x_i)^{[j-m_{x_i,y_i}]^+} \quad \text{for } j = 0,1,\ldots,r$$

where $r = \lfloor \delta^*/(k-1) \rfloor$ is the Y-degree of $\mathcal{Q}(X,Y)$, as defined in Section 2.2 of Chapter 2. From Corollary 4.8, we know that the solution $\mathcal{Q}'(X,Y)$ to the interpolation problem $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$ must have the form

$$\begin{aligned}\mathcal{Q}'(X,Y) &= \sum_{j=0}^{r}\left(b_j(X)\prod_{i=1}^{k}(X - x_i)^{[m_{x_i,y_i}-j]^+}\right)Y^j \\ &= \Phi(X)\sum_{j=0}^{r} b_j(X)T_j(X)\left(\frac{Y}{g(X)}\right)^j \end{aligned} \tag{4.11}$$

for some polynomials $b_j(X)$. Computing $\mathcal{Q}'(X,Y)$ and thereby solving both $\mathbf{IP}_{1,k-1}(\mathcal{P}',M)$ and $\mathbf{IP}_{1,k-1}(\mathcal{P},M)$ (in view of Theorem 4.4) reduce to finding $b_j(X)$. The following two propositions show that computing $b_j(X)$ is equivalent to solving a much smaller interpolation problem! In the following, the birational mapping defined in (4.1) and (4.2) are used.

Proposition 4.10. *Let $(\alpha,\beta) \in \mathcal{P}'$ be such that $g(\alpha) \neq 0$. Then $\mathcal{Q}'(X,Y)$, as defined in (4.11), passes through (α,β) with multiplicity m if and only if the polynomial $\mathcal{Q}''(X,Z) = \sum_{j=0}^{r} b_j(X)T_j(X)\,Z^j$ passes through the point $\left(\alpha, \gamma = \frac{\beta}{g(\alpha)}\right)$ with multiplicity m.*

Proof. We require that $\mu_{\alpha,\beta}\big(\mathcal{Q}'(X,Y)\big) = \mu_{\alpha,\gamma}\big(\mathcal{Q}''(X,Z)\big)$. Note that in view of (4.9) and (4.10), we have $\mu_{\alpha,\beta}(\Phi(X)) = 0$ whenever $g(\alpha) \neq 0$. Hence it follows from (4.11) and (2.8) that $\mu_{\alpha,\beta}\big(\mathcal{Q}''(X,\frac{Y}{g(X)})$ must be equal to $\mu_{\alpha,\beta}\big(\mathcal{Q}'(X,Y)\big)$. We now consider the birational mapping $\varphi_g(x,y) = (x,\frac{y}{g(x)})$ with inverse $\varphi_g^{-1}(x,z)$ $= (x, zg(x))$. By assumption $g(\alpha) \neq 0$, so the mapping $\varphi_g(x,y)$ is well-defined at $(x,y) = (\alpha,\beta)$. Then by Theorem 4.3, we have

$$\mu_{\alpha,\beta}\big(\mathcal{Q}''(X,\frac{Y}{g(X)})\big) = \mu_{\alpha,\frac{\beta}{g(\alpha)}}\big(Q''(X,Z)\big) \blacksquare$$

.

Proposition 4.10 can be applied to those points $(\alpha,\beta) \in \mathcal{P}'$ for which $g(\alpha) \neq 0$. The next proposition achieves the same goal for the case $g(\alpha) = 0$.

Proposition 4.11. *Let $(\alpha,\beta) \in \mathcal{P}'$ be such that $g(\alpha) = 0$ while $\beta \neq 0$ (if $\beta = 0$, then (α,β) is among the first k points of $\mathcal{P}'$). Let*

$$\mathcal{Q}''_\alpha(X,Z) \overset{\text{def}}{=} \sum_{j=0}^{r} b_j(X)(X-\alpha)^{m_{\alpha,0}-j}T_j(X)\, Z^j$$

Then the polynomial $\mathcal{Q}'(X,Y)$ passes through the point (α,β) with multiplicity m if and only if $\mathcal{Q}''_\alpha(X,Z)$ passes through the point $(\alpha,\frac{\beta}{g_\alpha(\alpha)})$ with multiplicity m, where polynomial $g_\alpha(X) = \frac{g(X)}{(X-\alpha)}$.

Proof. $\mathcal{Q}'(X,Y)$ in (4.11) can be rewritten as

$$\mathcal{Q}'(X,Y) = \Phi_\alpha(X)\sum_{j=0}^{r} b_j(X)(X-\alpha)^{m_{\alpha,0}-j}T_j(X)\left(\frac{Y}{g_\alpha(X)}\right)^j,$$

where $\Phi_\alpha(X) \overset{\text{def}}{=} \prod_{\substack{i=1\\ x_i\neq\alpha}}^{k}(X-x_i)^{m_{x_i,y_i}}$. The rest of the proof follows the same way as that of Proposition 4.10. First it can be observed that $\mu_{\alpha,\beta}(\Phi_\alpha(X)) = 0$, thus from (2.8) $\mu_{\alpha,\beta}\big(\mathcal{Q}'(X,Y)\big) = \mu_{\alpha,\beta}\big(\mathcal{Q}''_\alpha(X,\frac{Y}{g_\alpha(X)})\big)$. Then the birational mapping $\varphi_{g_\alpha}(x,y) = (x,\frac{y}{g_\alpha(X)})$ with inverse $\varphi_{g_\alpha}^{-1}(x,z) = (x, zg_\alpha(X))$ can be utilized. So from Theorem 4.3, we get $\mu_{\alpha,\beta}\big(\mathcal{Q}''_\alpha(X,\frac{Y}{g_\alpha(X)})\big) = \mu_{\alpha,\frac{\beta}{g_\alpha(\alpha)}}\big(\mathcal{Q}''_\alpha(X,Z)\big)$. $\blacksquare$

Propositions 4.10 and 4.11 are the cornerstone of our complexity reducing transformation, which is key to transforming the original interpolation problem to the *reduced interpolation problem* defined below.

Reduced interpolation problem: *Suppose we are given a set of points* $\mathcal{P}' = \{(x_1, y_1), (x_2, y_2), \ldots, (x_s, y_s)\}$, *such that* $y_1 = y_2 = \ldots = y_k = 0$ *and* $x_1, x_2, \ldots, x_k$ *are all distinct. We are furthermore given a set of associated multiplicities* $M = \{m_{x_1,y_1}, m_{x_2,y_2}, \ldots, m_{x_s,y_s}\}$. *Let the polynomials* $g(X)$, *and* $T_j(X)$ *be defined follows:*

- $T_j(X) = \prod_{i=1}^{k} (X - x_i)^{[j - m_{x_i,y_i}]^+}$
- $g(X) = \prod_{i=1}^{k} (X - x_i)$

Then the reduced interpolation problem consists of finding a nonzero polynomial $\mathcal{Q}''(X, Z) = \sum_{j=0}^{r} b_j(X) T_j(X) Z^j$ *of minimal* $(1, -1)$*-weighted degree, such that for all* $(x_{k+1}, y_{k+1}), (x_{k+2}, y_{k+2}), \ldots, (x_s, y_s)$, *we have*

◇ *if* $g(x_i) \neq 0$, *then*

$$\mu_{x_i, \frac{y_i}{g(x_i)}} \Big(\mathcal{Q}''(X, Z) \Big) \geq m_{x_i,y_i} \tag{4.12}$$

◇ *if* $g(x_i) = 0$, *then*

$$\mu_{x_i, \frac{y_i}{g_{x_i}(x_i)}} \Big(\mathcal{Q}''_{x_i}(X, Z) \Big), \geq m_{x_i,y_i} \tag{4.13}$$

where $g_{x_i}(X) = \frac{g(X)}{(X - x_i)}$ *and*

$$\mathcal{Q}''_{x_i}(X, Z) = \Big((X - x_i)^{m_{x_i,0}} \mathcal{Q}'' \Big(X, \frac{Z}{X - x_i} \Big) \Big).$$

We shall refer to the reduced interpolation problem above as $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$. The next theorem summarizes our results and establishes the connection between $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$ and the original interpolation problem $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$.

Theorem 4.12. *Let* $\mathcal{Q}''(X, Z)$ *be a solution to* $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$. *Then a solution to* $\mathbf{IP}_{1,k-1}(\mathcal{P}', M)$ *is given by*

$$\mathcal{Q}'(X, Y) = \Phi(X) \, \mathcal{Q}'' \left(X, \frac{Y}{g(X)} \right) \tag{4.14}$$

And thus a solution to, $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ *is given by*

$$\mathcal{Q}(X, Y) = \Phi(X) \, \mathcal{Q}'' \left(X, \frac{Y - h(X)}{g(X)} \right) \tag{4.15}$$

Proof. From Propositions 4.10 and 4.11, we know that $\mathcal{Q}'(X,Y)$ as given in (4.14) passes through all points $\{(x_i,y_i) : k < i \leq s\}$ with corresponding multiplicities. Plug the definitions of $\Phi(X), g(X)$ and $T_j(X)$'s in (4.14), we get

$$\begin{aligned}
\mathcal{Q}'(X,Y) &= \Phi(X)\sum_{j=0}^{r} b_j(X)T_j(X)(\frac{Y}{g(X)})^j \\
&= \sum_{j=0}^{r} b_j(X)\prod_{i=1}^{k}(X-x_i)^{m_{x_i,y_i}-j+[j-m_{x_i,y_i}]^+}Y^j \\
&= \sum_{j=0}^{r} b_j(X)\prod_{i=1}^{k}(X-x_i)^{[m_{x_i,y_i}-j]^+}Y^j
\end{aligned}$$

The last equality in the above equation follows from the fact that, for any integer m and n,

$$m-n = [m-n]^+ - [n-m]^+$$

Thus by Corollary 4.8, $\mathcal{Q}'(X,Y)$ also passes through points $\{(x_i,y_i) : 1 \leq i \leq k\}$ with the corresponding multiplicities.
What is left to be proved is that $\mathcal{Q}'(X,Y)$ is of minimum $(1,k-1)$-weighted degree. Let us first prove the following Lemma.

Lemma 4.13. *There exist the following weighted degree relationship between $\mathcal{Q}'(X,Y)$ and $\mathcal{Q}''(X,Z)$ as defined above:*

$$\deg_{1,k-1}\mathcal{Q}'(X,Y) = \deg\Phi(X) + \deg_{1,-1}\mathcal{Q}''(X,Z)$$

Proof. From the definition of $g(X), \Phi(X)$, and $T_j(X)$'s given in (4.9) and (4.10), we know that $\frac{\Phi(X)b_j(X)T_j(X)}{g(X)^j}$ is a well-defined polynomial for all j. Thus

$$\begin{aligned}
\deg_{1,k-1}\mathcal{Q}'(X,Y) &= \max_{0\leq j\leq r}\{\deg\frac{\Phi(X)b_j(X)T_j(X)}{g(X)^j} + (k-1)j\} \\
&= \max_{0\leq j\leq r}\{\deg\Phi(X)b_j(X)T_j(X) - \deg g(X)^j + (k-1)j\} \\
&= \max_{0\leq j\leq r}\{\deg\Phi(X) + \deg b_j(X)T_j(X) - j\} \\
&= \deg\Phi(X) + \max_{0\leq j\leq r}\{\deg b_j(X)T_j(X) - j\} \\
&= \deg\Phi(X) + \deg_{1,-1}\mathcal{Q}''(X,Z) \blacksquare
\end{aligned}$$

The proof of the Theorem can now be resumed. Let us assume that solution $\mathcal{Q}'(X,Y)$ is not minimal, thus there exist $\mathcal{A}'(X,Y)$, that passes all points in $\mathcal{P}'$ with associated multiplicities and also satisfies the following equation:

$$\deg_{1,k-1} \mathcal{A}'(X,Y) < \deg_{1,k-1} \mathcal{Q}'(X,Y). \tag{4.16}$$

According to Corollary 4.8, we know that any solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}', M)$ must have the form as in (4.8). Combining this with the definitions of $\Phi(X)$, $g(X)$ and $T_j(X)$'s, we know that $\mathcal{A}'(X,Y)$ can be expressed as:

$$\mathcal{A}'(X,Y) = \Phi(X) \sum_{j=0}^{l} p_j(X) T_j(X) \frac{Y^j}{g(X)^j}$$

Note that in the above equation, the upper limit of sum, l, does not have to be the same as in the expression of $\mathcal{Q}'(X,Y)$, however, this does not affect the proof of the theorem. Let us construct $\mathcal{A}''(X,Z)$ as follows:

$$\mathcal{A}''(X,Z) = \sum_{j=0}^{l} p_j(X) T_j(X) Z^j$$

For any point $(x_i, y_i) \in \mathcal{P}'$ such that $i > k$ and $g(x_i) \neq 0$, we know, from Proposition 4.10, that $\mathcal{A}''(X,Z)$ satisfies condition given by (4.12). For point $(x_i, y_i) \in \mathcal{P}'$ such that $i > k$ and $g(x_i) = 0$, we have

$$\begin{aligned}(X - x_i)^{m_{x_i,0}} \mathcal{A}''(X, \frac{Z}{X - x_i}) &= (X - x_i)^{m_{x_i,0}} \sum_{j=0}^{l} p_j(X) T_j(X) (\frac{Z}{X - x_i})^j \\ &= \sum_{j=0}^{l} p_j(X) (X - x_i)^{m_{x_i,0} - j} T_j(X) Z^j\end{aligned}$$

Thus from Proposition 4.11, we know that $\mathcal{A}''(X,Z)$ satisfies condition given by (4.13). From Lemma 4.13, we know that $\deg_{1,-1} \mathcal{A}''(X,Z) = \deg_{1,k-1} \mathcal{A}'(X,Y) - \deg \Phi(X)$. Combined with (4.16), we have

$$\deg_{1,-1} \mathcal{A}''(X,Z) < \deg_{1,-1} \mathcal{Q}''(X,Z). \tag{4.17}$$

However, (4.17) contradicts the fact that $\mathcal{Q}''(X,Z)$, is the minimal solution to $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$. So we conclude that $\mathcal{Q}'(X,Y)$ is the minimal solution to $\mathbf{IP}_{1,k-1}$

$(\mathcal{P}', M)$. Given equation (4.14), the 2nd part of the Theorem, i.e., equation (4.15), follows directly from Theorem 4.4. ▮

The **RIP**$_{1,-1}(\mathcal{P}', M)$ can be solved by Koetter's algorithm given in Section 2.2 of Chapter 2, too. In this case, $w = -1$, which is apparent from the proof of Lemma 4.13, and the Groebner basis polynomials should be initialized as follows:

$$\mathcal{Q}''_v(X, Z) = T_v(X)Z^v, \text{ for } 0 \leq v \leq r$$

Example 4.14 (Continue from Example 4.6) *Here the $g(X)$ and $T_i(X)$'s are, by definition, determined as following*

$$g(X) = (X - \alpha)(X - \alpha^2)$$

$$T_0(X) = 1$$

$$T_1(X) = 1$$

$$T_2(X) = (X - \alpha^2)$$

$$T_3(X) = (X - \alpha)(X - \alpha^2)^2 = \alpha^5 + \alpha^4 X + \alpha X^2 + X^3$$

Thus the Groebner-basis polynomials should be initialized as following

$$Q''_0(X, Z) = T_0(X) = 1$$

$$Q''_1(X, Z) = T_1(X)Z = Z$$

$$Q''_2(X, Z) = T_2(X)Z^2 = (X - \alpha^2)Z^2$$

$$Q''_3(X, Z) = T_3(X)Z^3 = (\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)Z^3$$

As in Example 4.9, only 5 points remain to be interpolated and these points are converted from points in set $\mathcal{P}'$ via coordinate transformation. Out of these 5 points, 4 of them listed in equation (4.18) have the property that $g(x_i) \neq 0$,

point (x_i, z_i)	(α^3, α^3)	(α^3, α^2)	$(1, 0)$	$(1, \alpha)$
multiplicity	1	1	1	1

, (4.18)

where $z_i = \frac{y_i}{g(x_i)}$, *for* $i \in \{3,4,5,6\}$. *The last point* (x_7, z_7) *given in equation (4.19) is converted from corresponding* (x_7, y_7) *in set* $\mathcal{P}'$, *where* $g(x_7) = g(\alpha^2) = 0$

$$\begin{array}{c|c} \text{point } (x_i, z_i) & (\alpha^2, 1) \\ \hline \text{multiplicity} & 1 \end{array}, \tag{4.19}$$

The coordinate transformation is carried out as $z_7 = \frac{y_7}{g_{x_7}(x_7)} = \frac{y_7}{g_{\alpha^2}(\alpha^2)}$. *The iterations of the interpolation procedure can be carried out as following:*

i	(x_i, z_i)	$\mathcal{Q}'_v(X,Z)$ *for* $v = 0,1,2,3$ *after each iteration*
3	(α^3, α^3)	$\mathcal{Q}''_2(X,Z) = Z(\alpha) + Z^2(\alpha^2 + X)$ $\mathcal{Q}''_0(X,Z) = 1 + Z(\alpha^4)$ $\mathcal{Q}''_1(X,Z) = Z(\alpha^3 + X)$ $\mathcal{Q}''_3(X,Z) = Z(\alpha^2) + Z^3(\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)$
4	(α^3, α^2)	$\mathcal{Q}''_0(X,Z) = 1 + Z(1) + Z^2(\alpha^6 + \alpha^4 X)$ $\mathcal{Q}''_1(X,Z) = Z(\alpha^3 + X)$ $\mathcal{Q}''_2(X,Z) = Z(\alpha^4 + \alpha X) + Z^2(\alpha^5 + \alpha^5 X + X^2)$ $\mathcal{Q}''_3(X,Z) = Z(\alpha) + Z^2(\alpha^5 + \alpha^3 X) + Z^3(\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)$
5	$(1, 0)$	$\mathcal{Q}''_1(X,Z) = Z(\alpha^3 + X)$ $\mathcal{Q}''_2(X,Z) = Z(\alpha^4 + \alpha X) + Z^2(\alpha^5 + \alpha^5 X + X^2)$ $\mathcal{Q}''_3(X,Z) = Z(\alpha) + Z^2(\alpha^5 + \alpha^3 X) + Z^3(\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)$ $\mathcal{Q}''_0(X,Z) = (1 + X) + Z(1 + X) + Z^2(\alpha^6 + \alpha^3 X + \alpha^4 X^2)$
6	$(1, \alpha)$	$\mathcal{Q}''_2(X,Z) = Z(\alpha^3 + X) + Z^2(\alpha^5 + \alpha^5 X + X^2)$ $\mathcal{Q}''_3(X,Z) = Z(1 + X) + Z^2(\alpha^5 + \alpha^3 X)$ $+Z^3(\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)$ $\mathcal{Q}''_0(X,Z) = (1 + X) + Z(1 + X) + Z^2(\alpha^6 + \alpha^3 X + \alpha^4 X^2)$ $\mathcal{Q}''_1(X,Z) = Z(\alpha^3 + \alpha X + X^2)$
7	$(\alpha^2, 1)$	$\mathcal{Q}''_2(X,Z) = Z(\alpha^3 + X) + Z^2(\alpha^5 + \alpha^5 X + X^2)$ $\mathcal{Q}''_3(X,Z) = Z(1 + X) + Z^2(\alpha^5 + \alpha^3 X)$ $+Z^3(\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)$ $\mathcal{Q}''_1(X,Z) = (1 + X) + Z(\alpha + \alpha^3 X + X^2)$ $+Z^2(\alpha^6 + \alpha^3 X + \alpha^4 X^2)$ $\mathcal{Q}''_0(X,Z) = (\alpha^2 + \alpha^6 X + X^2) + Z(\alpha^2 + \alpha^6 X + X^2)$ $+Z^2(\alpha + \alpha X + \alpha^4 X^2 + \alpha^4 X^3)$

After the last iteration in the above interpolation procedure, we select the following polynomial with the minimum $(1, -1)$*-weighted degree.*

$$\mathcal{Q}''(X,Z) = Z(\alpha^3 + X) + Z^2(\alpha^5 + \alpha^5 X + X^2).$$

And correspondingly

$$\mathcal{Q}''_{x_7}(X, Z) = \mathcal{Q}''_{\alpha^2}(X, Z) = Z(\alpha^2 + X) + Z^2(\alpha^3 + X)$$

It is easy to verify that the polynomial

$$\mathcal{Q}(X, Y) = (1 + \alpha^5 X + \alpha X^3) + Y(\alpha^4 + X + X^2) + Y^2(\alpha^3 + X)$$

obtained by using Theorem 4.12 is the same as the original polynomial $\mathcal{A}(X, Y)$ obtained in Example 4.6. □

In summary, we conclude that the efficient interpolation algorithms of [NH98, FG01, AKS03a, GKKG05] can be easily adapted to solve $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$. While $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$ appears to be more convoluted than the original problem $\mathbf{IP}_{1,k-1}$ $(\mathcal{P}, M)$, its complexity is often orders of magnitude lower. This is due to the fact that we *do not even need to consider* the first k points of $\mathcal{P}'$ in computing $\mathcal{Q}''(X, Z)$. In other words, these k interpolation points (which are chosen to have the largest multiplicities) are effectively pre-solved.

4.4 The Factorization Procedure

The reductions in complexity obtained in Section 4.3 would be less significant if one has to actually compute the original polynomial $\mathcal{Q}(X, Y)$ (using (4.14), say) in order to find a factor of type $Y - f(X)$. Fortunately, the lemma below shows that rather than factoring $\mathcal{Q}(X, Y)$, we can directly factor the much smaller polynomial $\mathcal{Q}''(X, Z)$ to recover the transmitted codeword.

Lemma 4.15. *If $\mathcal{Q}(X, Y)$, the solution to $\mathbf{IP}_{1,k-1}(\mathcal{P}, M)$ has a factor of $Y - f(X)$, where $f(X)$ is the information polynomial, then the solution to $\mathbf{RIP}_{1,-1}(\mathcal{P}', M)$ $\mathcal{Q}''(X, Z)$ has a factor $Z - \frac{\omega(X)}{\sigma(X)}$, where $\sigma(X)$ is the error-locator polynomial for the re-encoding positions in the received codeword.*

Proof. If $\mathcal{Q}(X, Y)$ has a factor equal to $Y - f(X)$, where $f(X)$ is the information polynomial, then $\mathcal{Q}'(X, Y) = \mathcal{Q}(X, Y + h(X))$ must have a factor equal to $Y-(f(X)-h(X))$, where $h(X)$ is the re-encoding polynomial as defined in (4.3). Let us introduce $\eta(X) = f(X) - h(X)$ and from the definitions of $f(X)$ and $h(X)$, we know that $\deg \eta(X) = k-1$. It is easy to observe that $\eta(X)$ evaluates to zero in exactly those positions i where y_i was the transmitted symbol, for all $i = 1, 2, \ldots, k$. Thus, with the substitution $Z = \frac{Y}{g(X)}$, a factor of type $Y-\eta(X)$ in $\mathcal{Q}'(X,Y)$ translates into a factor of type $Z-\frac{\eta(X)}{g(X)}$ in $\mathcal{Q}''(X,Z)$. The Roth-Ruckenstein factorization procedure of [RR00] can be applied to reveal the power-series expansion of the rational function $\frac{\eta(X)}{g(X)}$. Let us assume that there are ν correctable errors occurred among the k re-encoded positions of the RS codeword and denote the indices of these positions as a set $\mathbf{I_e} \overset{\text{def}}{=} \{i' : y_{i'} \neq f(x_{i'}) \text{ and } 1 \leq i' \leq k\}$. Due to the re-encoding procedure defined early in the chapter, it is easy to see that $\eta(X)$ evaluates to 0 at all those non-error positions, thus $\eta(X)$ can be written as

$$\eta(X) = \omega(X) \prod_{\substack{i' \notin \mathbf{I_e} \\ 1 \leq i' \leq k}} (X - x_{i'}) \tag{4.20}$$

Given $g(X)$ as defined in (4.9) and the above equation, we have, by canceling common terms,

$$\frac{\eta(X)}{g(X)} = \frac{\omega(X)}{\prod_{i' \in \mathbf{I_e}} (X - x_{i'})}$$

It now becomes clear that the denominator of the right side of the above equation can be treated as the error-locator polynomial, $\sigma(X)$, whose roots are the X coordinates of the error locations. ∎

Given the error locations, the corresponding error magnitudes can be found by observing that $e_i = y_i - f(x_i) = h(x_i) - f(x_i) = \eta(x_i)$ for $i \in \mathbf{I_e}$. In addition, we have $\frac{\eta(X)}{g(X)} = \frac{\omega(X)}{\eta(X)}$. Since $g(x_i) = \eta(x_i) = 0$, we have, by the L'Hôpital rule:

$$e_i = \left. \frac{g'(X)\omega(X)}{\sigma'(X)} \right|_{x_i}$$

whenever $\sigma(x_i) = 0$. And finally after obtaining all $f(x_i)$'s for $1 \leq i \leq k$, re-encoding can be applied again to recover the entire transmitted codeword. Note that $\omega(X), \sigma(X)$ can be found from the power-series expansion of $\frac{\omega(X)}{\sigma(X)}$ by a Padé approximation procedure, such as the Berlekamp-Massey algorithm. In practice, a reasonable bound of the maximum number of errors intended to be corrected by the soft-decision decoder can be set to be $n-k$, thus $\deg \sigma(X) \leq n-k$. According to [Mas69], only the first $2(n-k)$ coefficients in the expansion of rational function $\frac{\omega(X)}{\sigma(X)}$ need to be generated. This is similar to applying the Berlekamp-Massey algorithm to the hard-decision decoding of Reed-Solomon codes, where the maximum number of correctable errors is equal to $\frac{n-k}{2}$ and $(n-k)$ syndromes are used to compute the error-locator polynomial. Let us define the power series $S(X) = \sum_{i=0}^{\infty} S_i X^i \overset{\text{def}}{=} \frac{\omega(X)}{\sigma(X)}$ and here we refer S_i's as syndromes, too. In the following, a formal description of the reduced factorization algorithm is presented.

The Reduced Factorization Algorithm

Input: $\mathcal{Q}''(X,Z) = \sum_{j=0}^{r} b_j(X) T_j(X) Z^j$.

- Apply the Roth-Ruckenstein algorithm [RR00] to generate the first $2(n-k)$ coefficients of all Z-roots of $\mathcal{Q}''(X,Z)$ and store them in the following power series $S_t(X)$, for $t = 0, 1, ..., r$.

- Initialize $N_R = 0$. For $t = 0, 1, ..., r$

  1. $N_R = N_R + 1$;
  2. Use $S_t(X) = \sum_{i=0}^{2(n-k)-1} S_{t,i} X^i$ as syndrome polynomial and apply Berlekamp-Massey algorithm to obtain $\sigma_t(X)$. If $\deg \sigma_t(X) > n-k$, go to 6.
  3. Use Chien search to find the roots $x_{t,i}$'s. If the number of valid roots is smaller than $\deg \sigma_t(X)$, go to 6.
  4. Compute $\omega_t(X) = \sigma_t(X) S_t(X)$. If $\deg \omega_t(X) \geq \deg \sigma_t(X)$, go to 6.
  5. Compute $\eta_t(X) = \frac{g(X)\omega_t(X)}{\sigma_t(X)}$.
  6. $N_R = N_R - 1$.

```
    end
```

- Return: All valid $\eta_t(X)$'s.

Not all power series returned from the Roth-Ruckenstein factorization algorithm correspond to valid Z-root, where by valid Z-root, we refer to the power series which can be mapped to a valid codeword. Similar to the Berlekamp-Massey algorithm based hard-decision decoding, usually the following 3 conditions indicate that the power series whose first $2(n-k)$ coefficients are generated from the Roth-Ruckenstein factorization algorithm does not map to a valid Z-root:

- $\deg \sigma_t(X) > (n-k)$;
- $\sigma_t(X)$ has fewer number of roots in $\mathbb{F}_q$ than its own degree;
- $\deg \omega_t(X) > \deg \sigma_t(X)$.

All 3 conditions are used as false root detector in the factorization algorithm given above. Most of the time, only 1 polynomial $\eta(X)$ and 1 set of roots x_i's are returned from the reduced factorization algorithm. When the factorization procedure defined above returns multiple candidate polynomials, the spurious roots can be eliminated by applying the soft information to determine which root evaluates to the codeword with the largest maximum likelihood probability. The following example illustrate how the reduced factorization algorithm works.

Example 4.16 (Continue from Example 4.14) *Performing factorization on $\mathcal{Q}''(X,Z) = \alpha Z + (\alpha^2 X + X^2)Z^2 + (\alpha^5 + \alpha^4 X + \alpha X^2 + X^3)Z^3$ as described in [RR00], we can get the following 3 syndrome power series:*

$$S_0(X) = 0$$

$$S_1(X) = \alpha^5 + \alpha^3 X + \alpha X^2 + \alpha^6 X^3 + \alpha^4 X^4 + \alpha^2 X^5 + X^6 + \dots$$

$$S_2(X) = \alpha^5 + \alpha^6 X + \alpha^6 X^2 + \alpha X^3 + \alpha^5 X^4 + \alpha X^5 + \dots$$

Applying the Berlekamp-Massey algorithm to the 3 syndrome polynomials given above, we can obtain the following 3 solutions:

t	$\sigma_t(X)$	$\omega_t(X)$	$\frac{g(x)}{\sigma_t(X)}$	$\eta_t(X)$
0	1	0	$\alpha^3 + \alpha^4 X + X^2$	0
1	$\alpha^5(X - \alpha^2)$	α^5	$\alpha^3 + \alpha^2 X$	$\alpha + X$
2	$\alpha^4(X - \alpha)(X - \alpha^2)$	α^5	α^3	α

In this case, $\eta_1(X) = \alpha + X$ is the desired root, since it is easy to verify that the true information polynomial $f(X) = h(X) + \eta_1(X)$, where $f(X) = \alpha^6 + \alpha^2 X$ and $h(X) = \alpha^5 + \alpha^6 X$ are given in Example 4.6.

4.5 Conclusions

A proof of the applicability of the re-encoding coordinate transformation technique to bivariate polynomial interpolation process of the algebraic soft-decision decoding of Reed-Solomon codes is presented. A detailed example is also given to illustrate the entire re-encoding coordinate transformation process and its saving in interpolation complexity. In addition, we show how the factorization procedure should be modified to accommodate the *reduced interpolation problem*. Factorization complexity is also significantly reduced for high rate Reed-Solomon codes, since the number of iterations required is reduced to 2δ from $n - k$, where δ is the number of errors to be corrected in a received codeword of length n.

4.6 Appendix:Proof of Theorem 4.3

In the appendix, we give a proof of Theorem 4.3. *Proof.* The proof of the theorem consists of 2 parts. In the 1st part, we prove that $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \geq \mu_{\alpha,\gamma}(\mathcal{B}(X,Z))$. We then prove that $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \leq \mu_{\alpha,\gamma}(\mathcal{B}(X,Z))$ in the 2nd part.

Lemma 4.17. *For any given non-negative integer m, if $\mu_{\alpha,\gamma}(\mathcal{B}(X,Z)) = m$, then $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \geq m$.*

Proof. We start by writing $\mathcal{B}(X,Z)$ as

$$\mathcal{B}(X,Z) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(Z-\gamma)^j \tag{4.21}$$

By definition of the multiplicity function, we have

$$q_{i,j} = 0, \ \forall\, i+j < m \tag{4.22}$$

From the mapping defined in section 4.2, rational function $\mathcal{A}(X,Y)$ can be expressed as following

$$\begin{aligned}\mathcal{A}(X,Y) &= \varphi_g^{-1}(\mathcal{B}(X,Z)) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(\frac{Y}{g(X)}-\gamma)^j \\ &= \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(\frac{Y-\gamma g(X)}{g(X)})^j\end{aligned} \tag{4.23}$$

By definition of the mapping ϕ_g, $g(\alpha) \neq 0$. In addition, $X-\alpha \mid g(X)-g(\alpha)$, so we can define $h(X) \overset{\text{def}}{=} \frac{g(X)-g(\alpha)}{X-\alpha}$ and obviously $\deg h(X) = k-1$. Thus $g(X)$ can be expressed as

$$g(X) = g(\alpha) + (X-\alpha)h(X) \tag{4.24}$$

Let us apply (4.24) to (4.23) and we get

$$\begin{aligned}\mathcal{A}(X,Y) &= \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(\frac{Y-\gamma(g(\alpha)+(X-\alpha)h(X))}{g(X)})^j \\ &= \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(\frac{Y-\beta-\gamma(X-\alpha)h(X)}{g(X)})^j \\ &= \sum_{j=0}^{\infty}(Y-\beta-\gamma(X-\alpha)h(X))^j\frac{\sum_{i=0}^{\infty} q_{i,j}(X-\alpha)^i}{(g(X))^j},\end{aligned}$$

where $\beta = \gamma g(\alpha)$. From above, we can get

$$\mathcal{A}(X+\alpha, Y+\beta) = \sum_{j=0}^{\infty}(Y-\gamma X h(X+\alpha))^j\frac{\sum_{i=0}^{\infty} q_{i,j}X^i}{(g(X+\alpha))^j} \tag{4.25}$$

By definition, $g(X)$ is a polynomial of degree k, so we can write $(g(X+\alpha))^j$ in the following form

$$(g(X+\alpha))^j = \sum_{s=0}^{jk} \lambda_{j,s} X^s = \lambda_j(X) \text{ for all } j \tag{4.26}$$

Also from definition of $g(X)$, $g(\alpha) \neq 0$. Thus in (4.26), we have $\lambda_{j,0} \neq 0$ for all j, which implies that

$$\mu_{0,0}(\lambda_j(X)) = 0 \text{ for all } j \tag{4.27}$$

Let us plug (4.26) into (4.25), we get

$$\begin{aligned} \mathcal{A}(X+\alpha, Y+\beta) &= \sum_{j=0}^{\infty} (Y - \gamma X h(X+\alpha))^j \frac{\sum_{i=0}^{\infty} q_{i,j} X^i}{\lambda_j(X)} \\ &= \sum_{j=0}^{\infty} (Y - \gamma X h(X+\alpha))^j \sum_{i=0}^{\infty} \rho_{i,j} X^i, \end{aligned} \tag{4.28}$$

where obviously rational function $\rho_j(X) = \sum_{i=0}^{\infty} \rho_{i,j} X^i = \frac{\sum_{i=0}^{\infty} q_{i,j} X^i}{\lambda_j(X)}$. From (4.22), we can get

$$\mu_{0,0}(\sum_{i=0}^{\infty} q_{i,j} X^i) = [m-j]^+ \text{ for all } j.$$

Combining with (4.27) and applying the property of the multiplicity function as given in (2.8), we get

$$\mu_{0,0}(\rho_j(X)) = \mu_{0,0}(\sum_{i=0}^{\infty} q_{i,j} X^i) - \mu_{0,0}(\lambda_j(X)) = [m-j]^+.$$

By property (2.8), we also have the following obvious observation

$$\mu_{0,0}((Y - \gamma X h(X+\alpha))^j) = j\mu_{0,0}(Y - \gamma X h(X+\alpha)) = j \text{ for all } j.$$

Thus applying another property of the multiplication function defined in (2.10), we have

$$\begin{aligned} \mu_{0,0}(\mathcal{A}(X+\alpha, Y+\beta)) &\geq \min_j \{\mu_{0,0}((Y - \gamma X h(X+\alpha))^j) + \mu_{0,0}(\sum_{i=0}^{\infty} \rho_{i,j} X^i)\} \\ &= \min_j \{j + [m-j]^+\} = m \end{aligned}$$

Finally by applying property (2.9), we can conclude that $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \geq m$. ∎
So far we have proved Lemma 4.17, which shows that

$$\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \geq \mu_{\alpha,\gamma}(\mathcal{B}(X,Z)).$$

Lemma 4.18. *For any given non-negative integer m, if $\mu_{\alpha,\gamma}(\mathcal{B}(X,Z)) < m$, then $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) < m$.*

Proof. As we have done in the proof of Lemma 4.17, we start by writing $\mathcal{B}(X,Z)$ as

$$\mathcal{B}(X,Z) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} q_{i,j}(X-\alpha)^i(Z-\gamma)^j \tag{4.29}$$

And similarly, we can apply the mapping to find

$$\mathcal{A}(X+\alpha, Y+\beta) = \sum_{j=0}^{\infty}(Y-\gamma Xh(X+\alpha))^j \sum_{i=0}^{\infty}\rho_{i,j}X^i \tag{4.30}$$

Since $\mu_{\alpha,\gamma}(\mathcal{B}(X,Z)) < m$, by definition, there exist at least one $q_{i,j}$ in (4.29), such that $q_{i,j} \neq 0$ and $i+j < m$. And since $\mu_{0,0}(\rho_j(X)) = \mu_{0,0}(\sum_{i=0}^{\infty} q_{i,j}X^i)$, there exist at least one $\rho_{i,j}$ in (4.28), such that $\rho_{i,j} \neq 0$ and $i+j < m$. Let us expand (4.30) further to the following form

$$\begin{aligned}
\mathcal{A}(X+\alpha, Y+\beta) &= \sum_{j=0}^{\infty}\sum_{b=0}^{j}\sum_{i=0}^{\infty}\binom{j}{b} Y^b(-\gamma h(X+\alpha))^{j-b}X^{j-b}\rho_{i,j}X^i \\
&= \sum_{j=0}^{\infty}\sum_{b=0}^{j}\sum_{i=0}^{\infty}\binom{j}{b} Y^b X^{j-b}\rho_{i,j}X^i v_{j,b}(X)
\end{aligned} \tag{4.31}$$

Note that in the 2nd equality of the above equation, we have implicitly defined a new rational function, i.e.,

$$v_{j,b}(X) \overset{\text{def}}{=} (-\gamma h(X+\alpha))^{j-b}.$$

Let us write

$$v_{j,b}(X) = \sum_{r\geq 0} v_{j,b,r}X^r \tag{4.32}$$

and clearly the following holds for all $v_{j,j}(X)$'s:

$$v_{j,j}(X) \overset{\text{def}}{=} (-\gamma h(X+\alpha))^0 = v_{j,j,0} = 1 \text{ for all } j \tag{4.33}$$

Plugging (4.32) back to (4.31), we get

$$\mathcal{A}(X+\alpha, Y+\beta) = \sum_{j=0}^{\infty}\sum_{b=0}^{j}\sum_{i=0}^{\infty}\sum_{r\geq 0}\binom{j}{b}\rho_{i,j}\nu_{j,b,r}X^{i+j-b+r}Y^{b}$$

By a substitution of variables $\delta = j-b$ and $a = i+\delta+r$, we get

$$\begin{aligned}\mathcal{A}(X+\alpha, Y+\beta) &= \sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\sum_{\delta=0}^{a}\sum_{r\geq 0}\binom{\delta+b}{b}\rho_{a-\delta-r,\delta+b}\nu_{\delta+b,b,r}X^{a}Y^{b}\\ &= \sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\pi_{a,b}X^{a}Y^{b}\end{aligned}$$

where

$$\pi_{a,b} = \sum_{\delta=0}^{a}\sum_{r\geq 0}\binom{b+\delta}{b}\rho_{a-\delta-r,b+\delta}\nu_{b+\delta,b,r}. \tag{4.34}$$

Definition 4.3. *"$\prec$" is an order on the coefficients $\rho_{i,j}$'s such that*

$$\rho_{i_1,j_1}\prec\rho_{i_2,j_2} \text{ if } \begin{cases} i_1+j_1 < i_2+j_2 \\ or \\ i_1+j_1 = i_2+j_2 \text{ and } i_1 < i_2 \end{cases}$$

As we concluded above that there exist at least one nonzero $\rho_{i,j}$ such that $i+j < m$. Let us now order all such coefficients (if there are more than one) $\rho_{i,j}$ by the order defined above, and select

$$\rho_{\sigma,\tau} = \min_{\prec}\{\rho_{i,j} : i+j < m \text{ and } \rho_{i,j} \neq 0\}. \tag{4.35}$$

Let us now examine the corresponding coefficient $\pi_{\sigma,\tau}$ s as given in (4.34), and we have

$$\begin{aligned}\pi_{\sigma,\tau} &= \sum_{r\geq 0}\rho_{\sigma,\tau}\nu_{\tau,\tau,r} + \sum_{\delta=1}^{\sigma}\sum_{r\geq 0}\binom{\tau+\delta}{\tau}\rho_{\sigma-\delta-r,\tau+\delta}\nu_{\tau+\delta,\tau,r}\\ &= \sum_{r\geq 0}\rho_{\sigma,\tau}\nu_{\tau,\tau,r} \qquad (4.36)\\ &= \rho_{\sigma,\tau}. \qquad (4.37)\end{aligned}$$

The 2nd equality follows from (4.35) and the fact that all $\rho_{\sigma-\delta-r,\tau+\delta}$'s in the double sum term are 0, because $\rho_{\sigma-\delta-r,\tau+\delta}\prec\rho_{\sigma,\tau}$ according to Definition 4.3.

The 3rd equality above follows from equation (4.33). Thus we conclude that $\pi_{\sigma,\tau} \neq 0$. Thus $\mathcal{A}(X+\alpha, Y+\beta)$ contains monomial with degree less than m, so we conclude $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) < m$. ▮
Lemma 4.18 infers that $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) \leq \mu_{\alpha,\gamma}(\mathcal{B}(X,Z))$. Thus combining these 2 Lemmas, we conclude that $\mu_{\alpha,\beta}(\mathcal{A}(X,Y)) = \mu_{\alpha,\gamma}(\mathcal{B}(X,Z))$.

The material of Chapter 4 has been presented, in part, at *2003 International Symposium on Information Theory (ISIT)*, Koetter, Ralf; Ma, Jun; Vardy, Alexander; Ahmed, Arshad. The dissertation author was a joint investigator and co-author of the paper.

CHAPTER 5

Reduced Complexity Lee-O'Sullivan Interpolation Algorithm

Recently, Lee and O'Sullivan proposed a new interpolation algorithm for algebraic soft-decision decoding of Reed-Solomon codes. In some cases, the Lee-O'Sullivan algorithm turns out to be substantially more efficient than alternative interpolation approaches, such as Koetter's algorithm. Herein, we combine the re-encoding coordinate-transformation technique, originally developed in the context of Koetter's algorithm, with the interpolation method of Lee and O'Sullivan. To this end, we develop a new basis construction algorithm, which takes into account the additional constraints imposed by the interpolation problem that results upon the re-encoding transformation. This reduces the computational and storage complexity of the Lee-O'Sullivan algorithm by orders of magnitude, and makes it directly comparable to Koetter's algorithm in situations of practical importance.

5.1 Introduction

It is widely recognized that bivariate polynomial interpolation is the most computationally intensive step in algebraic soft-decision decoding (or, more generally, in algebraic list-decoding) of Reed-Solomon codes. Consequently, many different algorithms for bivariate polynomial interpolation have been proposed in the past decade — see [LO06b] for a recent survey.While all these algorithms are polynomial-time, they fall short of making the required computation feasible in practical applications, involving long high-rate Reed-Solomon codes. In all cases that we are aware of where algebraic soft-decision decoding of Reed-Solomon codes has been reduced to practice — either in software or in hardware — the bivariate interpolation is carried out using the algorithm of Koetter [Koe96b]. This is due in large part to the fact that Koetter's algorithm is amenable to the re-encoding coordinate transformation, developed in [KV03b, KMVA03], which reduces the complexity of the interpolation problem by orders of magnitude.

Recently, Lee and O'Sullivan [LO06a, LO06b] proposed a new algorithm for bivariate polynomial interpolation. Unlike Koetter's algorithm, which incrementally constructs a Groebner basis for the ideal of $\mathbb{F}_q[X,Y]$ defined by the interpolation constraints, the Lee-O'Sullivan algorithm computes a Groebner basis for the corresponding module over $\mathbb{F}_q[X]$ in two steps. The first step produces a basis, which is not necessarily a Groebner basis but can be quickly computed, while the second step iteratively reduces this basis to a Groebner basis with respect to the desired monomial order. Notably, for high-rate Reed-Solomon codes, the Lee-O'Sullivan algorithm is often more efficient than Koetter's algorithm. This is illustrated in the following example.

Example 5.1. *Let $\mathbb{C}$ be the $(255, 239, 17)$ RS code over GF(2^8). A typical interpolation problem arising in algebraic soft-decision decoding of $\mathbb{C}$ might involve the following multiplicities:*

multiplicity	7	6	5	4	3	2	1
# of points	229	12	10	4	3	10	10
# of constraints	6412	252	150	40	18	30	10

(5.1)

for a total of 6912 *linear constraints. Using Koetter's algorithm to solve this interpolation problem requires* 159.56×10^6 *finite field multiplications. In comparison, the Lee-O'Sullivan algorithm accomplishes this task using only* $45.37 \times$

10^6 *finite-field multiplications. Of these,* 32.11×10^6 *multiplications are expended in Step 1 (basis construction) while* 13.26×10^6 *multiplications are expended in Step 2 (basis reduction) of the algorithm. The figures above are precise; they were obtained by actually implementing both algorithms, and counting the number of finite-field multiplications in software.* □

The foregoing example shows that the Lee-O'Sullivan algorithm is about 3.5 times more efficient than Koetter's algorithm in this case. However, *both algorithms* are clearly infeasible in practice: there is simply no way to solve a system of 6912 linear equations in reasonable time with reasonable cost. This is where the re-encoding coordinate transformation of [KMVA03] and [KV03b] comes in. This transformation, briefly reviewed in the next section, converts the original interpolation problem to a *reduced interpolation problem*, which is orders of magnitude smaller.

Example 5.2. *Consider again the situation in Example 5.1. Judiciously choosing the re-encoding point set, we can eliminate from (5.1) the* $k = 239$ *points with the highest multiplicities: the* 229 *points of multiplicity* 7 *as well as* 10 *of the* 12 *points of multiplicity* 6*. This leaves only* 290 *linear equations to solve, rather than the original* 6912*.* □

The resulting reduced interpolation problem can be solved using Koetter's algorithm, as explained in [KV03b, KMVA03]. However, as illustrated in Example 1, the Lee-O'Sullivan algorithm is potentially much more efficient. Can this algorithm be applied to solve the reduced interpolation problem? This is precisely the subject of the present chapter.

While the second step of the Lee-O'Sullivan algorithm is generic, it is not at all clear how to construct a basis for the relevant module in the first step. In addition to the usual multiplicity constraints, the reduced interpolation problem imposes two more types of constraints (equations (5.13) and (5.15) of the next section) on the interpolation polynomial. Herein, we develop an efficient basis construction algorithm which takes all three types of constraints into account (Algorithm 2). Using this algorithm in conjunction with the Lee-O'Sullivan basis reduction method, we can solve the reduced interpolation problem.

Example 5.3. *Consider the situation described in Example 2. Using Algorithm 2 of Section 5.3 to compute a basis for the relevant module requires* 86×10^3 *finite-field multiplications. The second step (basis reduction) of the Lee-O'Sullivan algorithm then takes* 544×10^3 *multiplications. To summarize, we have:*

	Step 1	Step 2	Total
Original Lee-O'Sullivan	*32.11*	*13.26*	45.37
This Paper	*0.086*	*0.544*	*0.63*

The reduction in total complexity by a factor of $\sim$*72 is augmented by a corresponding reduction in memory requirements, due to the fact that the polynomials we need to deal with in the interpolation procedure now have much smaller degree.* □

The rest of this chapter is organized as follows. In the next section, we review the re-encoding coordinate transformation technique of [KMVA03, KV03b]. In Section 5.3, we develop an efficient basis construction algorithm (Algorithm 2) for the resulting reduced interpolation problem, and prove its correctness. In Section 5.5, we compare the complexity of the Lee-O'Sullivan and Koetter's algorithms, as applied to the reduced interpolation problem, and conclude with a brief discussion of the results.

5.2 Re-Encoding Coordinate Transformation Revisited

In this section, we review the re-encoding coordinate transformation technique presented in Section 4.3 of Chapter 4. This is necessary as some new notation has to be introduced for exposition of the algorithms to be discussed in this chapter. Throughout this chapter, we will use the definitions of *weighted degree*

$\deg_w A(X,Y)$ of a bivariate polynomial, *weighed-degree order* for bivariate monomials of *weighed-degree order* for bivariate monomials, and *multiplicity function* $\mu_{\alpha,\beta}(\cdot)$ for polynomials in $\mathbb{F}_q[X,Y]$ as given in Chapter 2. A brief review of the interpolation and coordinate transformation technique (Chapter 4) is given below. *Note that to accommodate the algorithms to be described in this chapter, we use slightly different notation from that used in Chapter 2 and Chapter 4.*

The interpolation problem that arises in algebraic soft-decision decoding of $\mathbb{C}$ can be formulated as follows.

Definition 5.1. (Interpolation problem). *Let $\ell_1, \ell_2, \ldots, \ell_n$ be positive intergers, and consider a set $\mathcal{P}$ of $\ell_1+\ell_2+\cdots+\ell_n \le nq$ distinct points in $\mathbb{F}_q \times \mathbb{F}_q$ given by*

$$\mathcal{P} \overset{\text{def}}{=} \left\{ (x_i, y_{i,j}) \ : \ i = 1,2,\ldots,n \ \text{ and } \ j = 1,2,\ldots,\ell_i \right\} \tag{5.2}$$

Let $M = \left\{ m_{x_i,y_{i,j}} : (x_i, y_{i,j}) \in \mathcal{P} \right\}$ denote the associated multiplicities, which are positive integers. The interpolation problem consists of computing a polynomial $\mathcal{Q}(X,Y) \not\equiv 0$ such that

$$\mu_{x_i,y_{i,j}}\big(\mathcal{Q}(X,Y)\big) \ge m_{x_i,y_{i,j}} \ \text{ for all } (x_i, y_{i,j}) \in \mathcal{P} \tag{5.3}$$

and $\deg_{k-1}\mathcal{Q}(X,Y)$ is minimal among all bivariate polynomials that satisfy the interpolation constraints (5.3). Note that multiplicity function $\mu_{x_i,y_{i,j}}$ is defined in Definition 2.2

We let $\mathcal{I}(\mathcal{P}, M)$ denote the ideal of $\mathbb{F}_q[X,Y]$ consisting of all the polynomials that satisfy (5.3). It is easy to see that

$$\langle \mathcal{I}(\mathcal{P}, M) \rangle_r \overset{\text{def}}{=} \mathcal{I}(\mathcal{P}, M) \cap \langle \mathbb{F}_q[X,Y] \rangle_r \tag{5.4}$$

is a submodule of $\langle \mathbb{F}_q[X,Y] \rangle_r$, where $\langle \mathbb{F}_q[X,Y] \rangle_r$ is defined in (2.5). Clearly, if the Y-degree of the interpolation polynomial $\mathcal{Q}(X,Y)$ is at most r, then $\mathcal{Q}(X,Y)$ belongs to this submodule. Most bivariate interpolation algorithms, including those of Koetter [Koe96b] and Lee-O'Sullivan [LO06a,LO06b] solve the interpolation problem by computing a Groebner basis for $\langle \mathcal{I}(\mathcal{P}, M) \rangle_r$ with respect to the $\prec_{k-1}$ monomial order.

The re-encoding coordinate transformation of [KMVA03,KV03b] begins with a set $\mathcal{R} \subset \mathcal{P}$ consisting of some k interpolation points with distinct X-coordinates.

We call the k points in $\mathcal{R}$ the *re-encoding points*. Although any k points in $\mathcal{P}$ with distinct X-coordinates can be chosen as the re-encoding points, one usually selects the k points with the highest multiplicity (cf. Example 5.2). The re-encoding process consists of computing the unique polynomial $h(X) \in \mathbb{F}_q[X]$ of degree $\leq k-1$ such that

$$h(x_i) = y_{i,j} \ \text{ for all } (x_i, y_{i,j}) \in \mathcal{R}. \tag{5.5}$$

and then shift the Y-coordinates of all $(x_i, y_{i,j}) \in \mathcal{P}$ by the following operation:

$$y_{i,j} := y_{i,j} - h(x_i).$$

In the rest of the chapter, we still use $\mathcal{P}$ to denote the set of interpolation points after the "shift" operation defined above. Correspondingly, $\mathcal{R}$ now contains the re-encoding points whose Y-coordinates have been "shifted" to zero. Since the equivalence between the interpolation problem defined on the original interpolation point set and the shifted interpolation point set has been established in Theorem 4.4 of Chapter 4, the latter one is used throughout the rest of the chapter.

To describe the coordinate transformation, we need some more notation. Let

$$\begin{aligned}
\mathcal{A} &\overset{\text{def}}{=} \textit{the set of points in } \mathcal{P}\backslash\mathcal{R} \textit{ whose } X\textit{-coordinates} \\
&\quad\ \textit{differ from those of the re-encoding points} \\
\mathcal{B} &\overset{\text{def}}{=} \textit{the set of points in } \mathcal{P}\backslash\mathcal{R} \textit{ whose } X\textit{-coordinates} \\
&\quad\ \textit{coincide with those of the re-encoding points}
\end{aligned}$$

Thus the sets $\mathcal{A}$ and $\mathcal{B}$ form a partition of $\mathcal{P}\backslash\mathcal{R}$. Given a subset $\mathcal{S}$ of $\mathcal{P}$, let $\{\mathcal{S}\}_X \subseteq \{1, 2, \ldots, n\}$ denote the set of *indices* of the X-coordinates of the points in $\mathcal{S}$, so that $\{\mathcal{B}\}_X \subseteq \{\mathcal{R}\}_X$ while $\{\mathcal{A}\}_X = \{1, 2, \ldots, n\}\backslash\{\mathcal{R}\}_X$. We define

$$\psi(X) \overset{\text{def}}{=} \prod_{i \in \{\mathcal{R}\}_X} (X - x_i), \tag{5.6}$$

$$\text{and } \phi(X) \overset{\text{def}}{=} \prod_{i \in \{\mathcal{R}\}_X} (X - x_i)^{\nu_i}. \tag{5.7}$$

In addition, we use $\psi'(X)$ to denote the first-order Hasse derivative of $\psi(X)$. The coordinate transformation consists of converting the set $\mathcal{P}$ in (5.2) into the set $\mathcal{P}' = \big\{(x_i, z_{i,j}) : (x_i, y_{i,j}) \in \mathcal{P}\backslash\mathcal{R}\big\}$, where

$$z_{i,j} \overset{\text{def}}{=} \begin{cases} \dfrac{y_{i,j}}{\psi(x_i)} & \text{if } (x_i, y_{i,j}) \in \mathcal{A} \\ \dfrac{y_{i,j}}{\psi'(x_i)} & \text{if } (x_i, y_{i,j}) \in \mathcal{B} \end{cases} \tag{5.8}$$

We let $\mathcal{A}'$ and $\mathcal{B}'$ denote the sets of points in $\mathcal{P}'$ transformed from the points in $\mathcal{A}$ and $\mathcal{B}$, respectively.

$$\begin{aligned} \mathcal{A}' &\overset{\text{def}}{=} \textit{the set of points in } \mathcal{P}'\backslash\mathcal{R} \textit{ whose } X\textit{-coordinates} && (5.9)\\ &\quad\ \textit{differ from those of the re-encoding points} \\ \mathcal{B}' &\overset{\text{def}}{=} \textit{the set of points in } \mathcal{P}'\backslash\mathcal{R} \textit{ whose } X\textit{-coordinates} && (5.10)\\ &\quad\ \textit{coincide with those of the re-encoding points} \end{aligned}$$

Let $M' \subset M$ denote the multiplicities of these points. Thus

$$M' \overset{\text{def}}{=} \big\{m_{x_i,z_{i,j}} = m_{x_i,y_{i,j}} : (x_i, z_{i,j}) \in \mathcal{P}'\big\} \tag{5.11}$$

For the multiplicities of the k re-encoding points $(x_i, y_{i,j}) \in \mathcal{R}$, we will introduce the simplified notation $\nu_i \overset{\text{def}}{=} m_{x_i,y_{i,j}}$. Finally, we need to define the polynomials $T_0(X), T_1(X), \ldots, T_r(X)$. These are known as *tail polynomials*, and given by

$$T_j(X) \overset{\text{def}}{=} \prod_{i\,\in\,\{\mathcal{R}\}_X} \big(X - x_i\big)^{[j-\nu_i]^+} \qquad \text{for } j = 0, 1, \ldots, r \tag{5.12}$$

where the operation $[\cdot]^+$ is defined by $[a]^+ = \max\{a, 0\}$. We are now ready to define the reduced interpolation problem.

Definition 5.2. (Reduced interpolation problem). *Given the sets $\mathcal{P}'$ and M', the reduced interpolation problem consists of computing a polynomial $\mathcal{Q}'(X,Z) \not\equiv 0$ which can be expressed as*

$$\mathcal{Q}'(X,Z) = \sum_{j=0}^{\infty} q_j(X)\, Z^j T_j(X) \tag{5.13}$$

and satisfies

$$\mu_{x_i,z_{i,j}}\big(\mathcal{Q}'(X,Z)\big) \geq m_{x_i,z_{i,j}} \quad \forall (x_i, z_{i,j}) \in \mathcal{A}' \tag{5.14}$$

$$\mu_{x_i,z_{i,j}}\Big(\big(X - x_i\big)^{\nu_i} \mathcal{Q}'\Big(X, \tfrac{Z}{X-x_i}\Big)\Big) \geq m_{x_i,z_{i,j}} \quad \forall (x_i, z_{i,j}) \in \mathcal{B}' \tag{5.15}$$

such that $\deg_{-1}\mathcal{Q}'(X,Z)$ *is minimal among all bivariate polynomials that satisfy the constraints* (5.13), (5.14), *and* (5.15).

Due to the elimination of the k re-encoding points in $\mathcal{R}$, the reduced interpolation problem is often orders of magnitude smaller than the original interpolation problem (cf. Examples 5.2 and 5.3). It is shown in Chapter 4 that the solution to the "shifted" interpolation problem (or, better yet, the corresponding list of codewords of $\mathbb{C}$) can be efficiently reconstructed from the solution $\mathcal{Q}'(X,Z)$ to the reduced interpolation problem. In particular, we have

$$\mathcal{Q}(X,Y) \;=\; \phi(X)\,\mathcal{Q}'\!\left(X,\,\frac{Y}{\psi(X)}\right) \tag{5.16}$$

where $\psi(X)$, and $\phi(X)$ are as defined in (5.6) and (5.7). We let ξ denote the mapping in (5.16), and use χ to denote its inverse mapping. Thus $\mathcal{Q}(X,Y) = \xi\big(\mathcal{Q}'(X,Z)\big)$ and

$$\mathcal{Q}'(X,Z) = \chi\big(\mathcal{Q}(X,Y)\big) \overset{\text{def}}{=} \frac{\mathcal{Q}\big(X,\,\psi(X)Z\big)}{\phi(X)} \tag{5.17}$$

In principle, the mappings ξ and χ are defined on the field of bivariate rational functions over $\mathbb{F}_q$, but in the case of $\mathcal{Q}(X,Y)$ and $\mathcal{Q}'(X,Z)$, they take polynomials to polynomials.

Our goal herein is to use (an appropriate modification of) the Lee-O'Sullivan algorithm [LO06a, LO06b] for the solution of the reduced interpolation problem. Let $\mathcal{J}(\mathcal{P}',M')$ denote the set of all polynomials in $\mathbb{F}_q[X,Z]$ that satisfy the constraints (5.13), (5.14), (5.15). Observe that $\mathcal{J}(\mathcal{P}',M')$ is *no longer an ideal* of $\mathbb{F}_q[X,Z]$ (e.g. if $A(X,Z) \in \mathcal{J}(\mathcal{P}',M')$, then $ZA(X,Z)$ is not necessarily in $\mathcal{J}(\mathcal{P}',M')$, since (5.13) could be violated). Nevertheless, the set

$$\big\langle \mathcal{J}(\mathcal{P}',M')\big\rangle_r \overset{\text{def}}{=} \mathcal{J}(\mathcal{P}',M') \cap \big\langle \mathbb{F}_q[X,Z]\big\rangle_r \tag{5.18}$$

can be still regarded as a free module over $\mathbb{F}_q[X]$. In order to apply the Lee-O'Sullivan algorithm to the reduced interpolation problem, we need to construct a basis for this module.

5.3 Basis Construction Algorithms

We assume that an upper bound r on the Z-degree of the solution $\mathcal{Q}'(X,Z)$ to the reduced interpolation problem is known. Several such bounds can be found in the literature [KV03b]. Given r, the Lee-O'Sullivan algorithm would proceed as follows.

Step 1: Construct a basis $\mathcal{G} = \{G_0, G_1, \ldots, G_r\}$ for the module $\langle \mathcal{J}(\mathcal{P}', M') \rangle_r$ in (5.18), with the additional property that Z-deg $G_s = s$ for all $s = 0, 1, \ldots, r$.

Step 2: Reduce $\mathcal{G}$ to a Groebner basis with respect to the monomial order $\prec_{-1}$. Output as $\mathcal{Q}'(X, Z)$ the minimal (with respect to $\prec_{-1}$) element of this Groebner basis.

This general approach was developed by Lee and O'Sullivan [LO06a,LO06b] in the context of the original interpolation problem (cf. Definition 5.1). Step 2 of the Lee-O'Sullivan algorithm is generic, in the sense that it works for any module over $\mathbb{F}_q[X]$ and any monomial order. Step 1 of the algorithm is easy in the case of the module $\langle \mathcal{I}(\mathcal{P}, M) \rangle_r$.

Step 1 of the algorithm is relatively easy in the case of the original interpolation problem. Let

$$l_i(X) \overset{\text{def}}{=} \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{X - x_j}{x_i - x_j} \qquad \text{for } i \in \{1, 2, \ldots, n\} \tag{5.19}$$

be the relevant Lagrange interpolation polynomials. Lee and O'Sullivan [LO06a] construct a basis for $\langle \mathcal{I}(\mathcal{P}, M) \rangle_r$ as follows.

Algorithm 1 ***(The Lee-O'Sullivan Basis Construction)***
Initialize the sets $\mathcal{P}_0$ and M_0 as follows $\mathcal{P}_0 := \mathcal{P}$ and $M_0 := M$. Set $s := 0$, and proceed iteratively through the steps below for $s = 0, 1, \ldots, r$ (until exiting with $s = r + 1$ at Step 4).

$\boxed{\mathbf{1}}$ *For each code position i in $\{\mathcal{P}_s\}_X$, find the largest multiplicity in this position. That is, set*

$$m_i \overset{\text{def}}{=} \max_j \{ m_{x_i, y_{i,j}}(s) \} \quad \textit{for all } i \in \{\mathcal{P}_s\}_X \tag{5.20}$$

where $m_{x_i,y_{i,j}}(s)$ are elements of M_s. Let y_i be the Y-coordinate of the point in $\mathcal{P}_s$ with this largest multiplicity.

$\boxed{2}$ *First, compute the polynomials $a_s(X)$ and $b_s(X)$ defined as follows:*

$$a_s(X) \overset{\text{def}}{=} \prod_{i\,\in\,\{\mathcal{P}_s\}_X} (X - x_i)^{m_i} \quad \textit{and} \quad b_s(X) \overset{\text{def}}{=} \sum_{i\,\in\,\{\mathcal{P}_s\}_X} y_i l_i(X)$$

Note that the empty product should be interpreted as one and the empty sum as zero, throughout. Next, compute

$$B_s(X,Y) \overset{\text{def}}{=} a_s(X) \prod_{i=0}^{s-1} \Big(Y - b_i(X)\Big) \tag{5.21}$$

$\boxed{3}$ *Decrease by one the multiplicity of those points that were processed at the current iteration. That is, set*

$$m_{x_i,y_{i,j}}(s+1) := \begin{cases} m_{x_i,y_{i,j}}(s) - 1 & \textit{if } y_{i,j} = y_i \\ m_{x_i,y_{i,j}}(s) & \textit{if } y_{i,j} \neq y_i \end{cases} \tag{5.22}$$

If for some points $(x_i, y_{i,j})$, this results in a zero multiplicity (i.e. $m_{x_i,y_{i,j}}(s+1) = 0$) remove these points from $\mathcal{P}_s$. Let $\mathcal{P}_{s+1}$ and M_{s+1} denote the sets thereby obtained.

$\boxed{4}$ *Set $s := s+1$. If $s \le r$, go back to $\boxed{1}$. Otherwise, stop and output the set*

$$\big\{B_0(X,Y), B_1(X,Y), \ldots, B_r(X,Y)\big\}.$$

It is proved in [LO07] that the set of polynomials produced by Algorithm 1 constitutes a basis for the module $\langle \mathcal{I}(\mathcal{P}, M)\rangle_r$ of (5.4).

However, in the case of $\langle \mathcal{J}(\mathcal{P}', M')\rangle_r$, it is not at all clear how to construct the required basis. The situation is rather more complicated than in the interpolation problem of Definition 5.1, in view of the additional constraints (5.13) and (5.15). In this section, we develop a basis construction algorithms appropriate for the reduced interpolation problem.

In oder to describe Algorithm 2, let us first define the following Lagrange interpolation polynomials:

$$h_i(X) \overset{\text{def}}{=} \prod_{\substack{j \in \{\mathcal{A}\}_X \\ j \neq i}} \frac{X - x_j}{x_i - x_j} \qquad \text{for all } i \in \{\mathcal{A}\}_X \tag{5.23}$$

$$g_i(X) \overset{\text{def}}{=} \prod_{j \in \{\mathcal{A}\}_X} \frac{X - x_j}{x_i - x_j} \qquad \text{for all } i \in \{\mathcal{B}\}_X \tag{5.24}$$

where the sets $\mathcal{A}$ and $\mathcal{B}$ and the notation $\{\cdot\}_X$ are as defined in the previous section. In addition, let

$$\Theta_j(X) \overset{\text{def}}{=} \prod_{i \in \mathcal{D}} \big(X - x_i\big)^{[j-\nu_i]^+} \qquad \text{for } j = 0, 1, \ldots, r \tag{5.25}$$

where $\mathcal{D} = \{\mathcal{R}\}_X \backslash \{\mathcal{B}\}_X$. Thus $\mathcal{D} \subset \{1, 2, \ldots, n\}$ is the set of the X-coordinate indices of those re-encoding points which are alone in their code position: there are no other points in $\mathcal{P}$ with these X-coordinates. Algorithm 2 can be now stated as follows.

Algorithm 2 ***(The New Basis Construction Algorithm)***

Initialize the sets of points $\mathcal{A}_0$, $\mathcal{B}_0$ as $\mathcal{A}_0 := \mathcal{A}'$ and $\mathcal{B}_0 := \mathcal{B}'$, where $\mathcal{A}'$ and $\mathcal{B}'$ are as in (5.10) and (5.11). Let $M_0 := M'$ where M' is given by (5.11), and $\nu_i(0) := \nu_i$ for all $i \in \{\mathcal{R}\}_X$. Set $s := 0$, and proceed through the steps below for $s = 0, 1, \ldots, r$ (until exiting with $s = r + 1$ at Step 4).

$\boxed{\textbf{1}}$ *For each code position i in $\{\mathcal{A}_s\}_X \cup \{\mathcal{B}_s\}_X$, find the largest multiplicity in this position. That is, set*

$$m_i^{(s)} \overset{\text{def}}{=} \max_j \{m_{x_i, z_{i,j}}\} \quad \textit{for all } i \in \big(\{\mathcal{A}_s\}_X \cup \{\mathcal{B}_s\}_X\big) \tag{5.26}$$

where $m_{x_i, z_{i,j}}$ are the elements of M_s. Let z_i be the Z-coordinate of the point with the largest multiplicity, that is z_i is such that $m_i^{(s)} = m_{x_i, z_i}$. For every position $i \in \{\mathcal{B}_s\}_X$, also check whether $\nu_i(s) \geq m_i^{(s)}$, where $\nu_i(s)$ is the (current) multiplicity of the corresponding re-encoding point. If so, set $z_i := 0$ and $\nu_i(s + 1) := \nu_i(s) - 1$. Finally, let

$$\mathcal{E}_s \overset{\text{def}}{=} \{\mathcal{B}_s\}_X \setminus \big\{i \in \{\mathcal{B}_s\}_X \,:\, z_i = 0\big\}$$

(if $\nu_i^{(s)} \geq m_i^{(s)}$, we decrease $\nu_i^{(s)}$ by one and do not *process the i-th position in $\{\mathcal{B}_s\}_X$ at the current iteration). In addition, define the set $\mathcal{F}_s \stackrel{\text{def}}{=} \{\mathcal{B}'\}_X \backslash \mathcal{E}_s$. Thus $\mathcal{F}_s \subset \{1, 2, \ldots, n\}$ is the set of indices of the X-coordinates of those points in $\mathcal{B}'$ that are* not *processed at the current iteration.*

$\boxed{2}$ *First, compute the polynomials $v_s(X)$ and $w_s(X)$ defined as follows:*

$$v_s(X) \stackrel{\text{def}}{=} \begin{cases} \prod_{i \in \mathcal{E}_s}(X - x_i) & \textit{if } \mathcal{E}_s \neq \emptyset \\ 1 & \textit{otherwise} \end{cases}, \tag{5.27}$$

and

$$w_s(X) \stackrel{\text{def}}{=} \sum_{i \in \{\mathcal{A}_s\}_X} z_i h_i(X) v_s(X) + \sum_{i \in \mathcal{E}_s} z_i g_i(X) \frac{v_s(X)}{(X - x_i)} \tag{5.28}$$

Note that the empty product should be interpreted as one and the empty sum as zero, throughout.

For $i \in \{\mathcal{B}\}_X$, let $\sigma_i^{(s)}$ be the largest integer such that $(X - x_i)^{\sigma_i^{(s)}}$ divides $\prod_{j=0}^{s-1} v_j(X)$. Define the auxiliary polynomials

$$u'_s(X) \stackrel{\text{def}}{=} \prod_{i \in \mathcal{F}_s} \left(X - x_i\right)^{[s - \nu_i - \sigma_i^{(s)}]^+} \tag{5.29}$$

$$u''_s(X) \stackrel{\text{def}}{=} \prod_{i \in \mathcal{E}_s} \left(X - x_i\right)^{[m_i^{(s)} + s - \nu_i - \sigma_i^{(s)}]^+} \tag{5.30}$$

where ν_i is the original *multiplicity of the re-encoding point at the i-th code position. Then define $u_s(X)$ as follows:*

$$u_s(X) \stackrel{\text{def}}{=} u'_s(X)\, u''_s(X) \prod_{i \in \{\mathcal{A}_s\}_X} \left(X - x_i\right)^{m_i^{(s)}} \Theta_s(X) \tag{5.31}$$

With $u_s(X)$ defined by (5.29)–(5.31), the basis polynomials can be computed as follows:

$$G_s(X, Z) \stackrel{\text{def}}{=} u_s(X) \prod_{i=0}^{s-1} \Big(Z v_i(X) - w_i(X) \Big) \tag{5.32}$$

$\boxed{3}$ *Update the sets of points and multiplicities, as follows. First, decrease by one the multiplicity of those points that were processed at the current iteration, that is*

$$m_{x_i, z_{i,j}} := \begin{cases} m_{x_i, z_{i,j}} & \textit{if } z_{i,j} \neq z_i \\ m_{x_i, z_{i,j}} - 1 & \textit{if } z_{i,j} = z_i \end{cases}$$

If for some points $(x_i, z_{i,j})$ this results in a zero multiplicity ($m_{x_i,z_{i,j}} = 0$), remove these points from $\mathcal{A}_s$ and $\mathcal{B}_s$. Also purge the corresponding zero multiplicities from M_s. Let $\mathcal{A}_{s+1}$, $\mathcal{B}_{s+1}$, and M_{s+1} denote the sets thereby obtained.

$\boxed{4}$ *Set $s := s+1$. If $s \leq r$, go back to $\boxed{1}$. Otherwise, stop and output the set $\{G_0(X,Z), G_1(X,Z), \ldots, G_r(X,Z)\}$.*

Actually at any iteration s of Algorithm 2, we always have $m_i^{(s)} + s - \nu_i - \sigma_i^{(s)} \geq 0$ for all $i \in \mathcal{E}_s$. This can be derived from the following properties of the algorithm: The value represented by $(s - \sigma_i^{(s)})$, which is always non-negative, indicate how many iterations, out of the s iterations, that the re-encoding point at x_i is the one with the largest remaining multiplicity. Thus $\nu_i - (s - \sigma_i^{(s)})$ denotes the remaining multiplicity of the re-encoding point at x_i. Since $i \in \mathcal{E}_s$ at iteration s, we must have $m_i^{(s)} \geq (\nu_i - (s - \sigma_i^{(s)}))$, where $m_i^{(s)}$ is the remaining multiplicity of point (x_i, z_i). Thus throughout the rest of this chapter, we write $(m_i^{(s)} + s - \nu_i - \sigma_i^{(s)})$ instead of $[m_i^{(s)} + s - \nu_i - \sigma_i^{(s)}]^+$.

Hereafter, let $\mathcal{G} = \{G_0(X,Z), G_1(X,Z), ..., G_r(X,Z)\}$ denote the set of polynomials produced by Algorithm 2. It is obvious from (5.32) that Z-deg $G_s(X,Z) = s$ for $s = 0, 1, \ldots, r$. This property also implies that the $r+1$ polynomials in $\mathcal{G}$ are linearly independent over $\mathbb{F}_q[X]$. Now we need to prove that $\langle \mathcal{G} \rangle = \langle \mathcal{J}(\mathcal{P}', M') \rangle_r$. We start by showing that $G_s(X,Z)$ belongs to $\mathcal{J}(\mathcal{P}', M')$ for all $s = 0, 1, \ldots, r$. This is established in a series of lemmas in what follows.

Lemma 5.4. *Write $\Gamma_s(X,Z) \overset{\text{def}}{=} Zv_s(X) - w_s(X)$, where $v_s(X)$ and $w_s(X)$ are as defined in* Algorithm 2. *Suppose that a point $(x_i, z^*) \in \mathcal{A}'$ is processed at iteration s of the algorithm (that is, $i \in \{\mathcal{A}_s\}_X$ and $z_i = z^*$ at this iteration). Then $\Gamma_s(x_i, z^*) = 0$.*

Proof. It should be obvious from (5.24) that $g_j(x_i) = 0$ for all $j \in \{\mathcal{B}_s\}_X$. Similarly, $h_j(x_i) = 0$ for all $j \neq i$ and $h_i(x_i) = 1$ in view of (5.23). Hence $w_s(x_i) = z_i h_i(x_i) v_s(x_i) = z^* v_s(x_i)$, and $\Gamma_s(x_i, z^*) = z^* v_s(x_i) - w_s(x_i) = 0$. ∎

Lemma 5.5. *Each of the polynomials produced by* Algorithm 2 *satisfies* (5.14).

That is, for all $s = 0, 1, \ldots, r$, we have

$$\mu_{x_i,z_{i,j}}\big(G_s(X,Z)\big) \;\geq\; m_{x_i,z_{i,j}} \qquad \forall (x_i, z_{i,j}) \in \mathcal{A}'$$

Proof. Throughout the proof of this lemma, $m_{x_i,z_{i,j}}$ denotes the original multiplicity of point $(x_i, z_{i,j}) \in \mathcal{B}'$. Suppose that a point $(x_i, z_{i,j}) \in \mathcal{A}'$ was processed η times during iterations $0, 1, \ldots, s-1$ of Algorithm 2. Then the multiplicity of this point during iteration s is $m_{x_i,z_{i,j}} - \eta$, which implies that $m_i^{(s)} \geq m_{x_i,z_{i,j}} - \eta$. It now follows from (5.31) that $\mu_{x_i,z_{i,j}}\big(u_s(X)\big) \geq m_{x_i,z_{i,j}} - \eta$. On the other hand

$$\mu_{x_i,z_{i,j}}\left(\prod_{t=0}^{s-1}\Big(Zv_t(X) - w_t(X)\Big)\right) \;=\; \mu_{x_i,z_{i,j}}\left(\prod_{t=0}^{s-1}\Gamma_t(X,Z)\right)$$

is at least η by Lemma 5.4. Hence, the lemma follows from (5.32) together with the fact that $\mu_{\alpha,\beta}(AB) = \mu_{\alpha,\beta}(A) + \mu_{\alpha,\beta}(B)$ for all $A, B \in \mathbb{F}_q[X,Z]$ and all $\alpha, \beta \in \mathbb{F}_q$. ∎

Lemma 5.6. *Let $\Gamma_s(X,Z)$ be as defined in* Lemma 5.4*, and suppose that a point $(x_i, z^*) \in \mathcal{B}'$ is processed at iteration s of* Algorithm 2 *— that is, $i \in \mathcal{E}_s$ and $z_i = z^*$ at this iteration. Define $\Gamma'_s(X,Z) = \Gamma_s\big((X, Z/(X - x_i)\big)$. Then $\Gamma'_s(X,Z)$ is a bivariate polynomial, and moreover $\Gamma'_s(x_i, z^*) = 0$.*

Proof. The fact that $\Gamma_s\big((X, Z/(X - x_i)\big)$ is indeed a well-defined polynomial follows immediately from the observation that $X - x_i$ is a factor of $v_s(X)$ for all $i \in \mathcal{E}$. Also notice that $v_s(x_i) = 0$, and therefore

$$w_s(x_i) \;=\; z_i g_i(x_i) \prod_{j \in \mathcal{E}\setminus\{i\}} (x_i - x_j) \;=\; z^* \prod_{j \in \mathcal{E}\setminus\{i\}} (x_i - x_j) \tag{5.33}$$

where the second equality follows from the fact that $g_i(x_i) = 1$ in view of (5.24). Evaluating the polynomial $Zv_s(X)/(X - x_i)$ at (x_i, z^*), we get exactly the expression on the right-hand side of (5.33), and the lemma follows. ∎

Lemma 5.7. *Each of the polynomials produced by* Algorithm 2 *satisfies* (5.15). *That is, for all $s = 0, 1, \ldots, r$, we have*

$$\mu_{x_i,z_{i,j}}\left((X - x_i)^{\nu_i}\, G_s\Big(X, \tfrac{Z}{X - x_i}\Big)\right) \geq m_{x_i,z_{i,j}} \quad \forall (x_i, z_{i,j}) \in \mathcal{B}'$$

Proof. Throughout the proof of this lemma, $m_{x_i,z_{i,j}}$ denotes the original multiplicity of point $(x_i, z_{i,j}) \in \mathcal{B}'$. For iteration s of the algorithm, depending on the value of i, there are 2 cases to be considered.

Case 1. Suppose that $i \in \mathcal{E}_s$, and we also assume that this point has been processed η times during iterations $0, 1, \ldots, s-1$ of Algorithm 2. Let us define the following auxiliary polynomial

$$u_{s,i}(X) = \frac{u_s(X)}{(X - x_i)^{(m_i^{(s)}+s-\nu_i-\sigma_i^{(s)})}}.$$

From the definition of $u_s(X)$ given in (5.31) and the fact that $i \in \mathcal{E}_s$, we can see that $u_{s,i}(X)$ is a well-defined polynomial. Let us also define $\Lambda(X, Z)$ as follows:

$$\Lambda(X, Z) = \prod_{0 \le t < s} \Gamma_t(X, \frac{Z}{X - x_i})(X - x_i)^{(m_i^{(s)}+s-\nu_i-\sigma_i^{(s)})+\nu_i}, \tag{5.34}$$

where $\Gamma_t(X, Z)$'s are as defined in Lemma 5.4.

Thus according to (5.32), $(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X-x_i})$ can be written as follows:

$$(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X - x_i}) = \Lambda(X, Z) u_{s,i}(X)$$

To facilitate the proof, we define the following index sets

$$\begin{aligned} \mathcal{S} &= \{i : 0 \le i < s\} && (5.35)\\ \mathcal{S}' &= \{i : 0 \le i < s \text{ and a point with X-coordinate} \\ &\qquad \text{equal to } x_i \text{ is processed at iteration i}\} && (5.36)\\ \mathcal{S}'' &= \{i : 0 \le i < s \text{ and the point } (x_i, z_{i,j}) \\ &\qquad \text{is processed at iteration } i\} && (5.37) \end{aligned}$$

By their definitions, we have

$$\mathcal{S}'' \subseteq \mathcal{S}' \subseteq \mathcal{S},$$

and $|\mathcal{S}'| = \sigma_i^{(s)}$, $|\mathcal{S}''| = \eta$.

Apparently $(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X-x_i})$ is a well-defined bivariate polynomial if we can establish that $\Lambda(X))$ is one. According to the definitions of $v_i(X)$'s and $\mathcal{S}'$, $(X - x_i)\,|v_t(X)$ for $t \in \mathcal{S}'$, so $\left(\prod_{t\,\in\,\mathcal{S}'}\left(\frac{Z}{X-x_i}v_t(X) - w_t(X)\right)\right)$ is a well-defined bivariate polynomial. In addition, one can verify that

$$(m_i^{(s)} + s - \nu_i - \sigma_i^{(s)}) + \nu_i - (s - \sigma_i^{(s)}) \geq m_i^{(s)}, \tag{5.38}$$

so $\prod_{t\,\in\,\mathcal{S}\backslash\mathcal{S}'}\Gamma_t\big(X, \frac{Z}{X-x_i}\big)(X - x_i)^{(m_i^{(s)}+s-\nu_i-\sigma_i^{(s)})+\nu_i}$is also a well-defined bivariate polynomial. Thus $\Lambda(X))$ is a well-defined bivariate polynomial.

We now proceed to prove the lemma for all points $(x_i, z_{i,j}) \in \mathcal{B}'$ such that $i \in \mathcal{E}_s$, while distinguishing between the following 2 sub-cases:

Sub-Case 1.1. Suppose that $z_{i,j} = z_i$, i.e., the point $(x_i, z_{i,j})$ is processed at iteration s. Thus we have $m_i^{(s)} = m_{x_i,z_{i,j}} - \eta$. It follows from (5.38) that

$$\mu_{x_i,z_{i,j}}\Big(\prod_{t\,\in\,\mathcal{S}\backslash\mathcal{S}'}\Gamma_t\Big(X, \frac{Z}{X-x_i}\Big)(X - x_i)^{(m_i^{(s)}+s-\nu_i-\sigma_i^{(s)})+\nu_i}\Big) \geq m_i^{(s)}$$

On the other hand, since $\mathcal{S}'' \subseteq \mathcal{S}'$, we have

$$\mu_{x_i,z_{i,j}}\left(\prod_{t\,\in\,\mathcal{S}'}\Gamma_t(X,Z)\right) \geq \mu_{x_i,z_{i,j}}\left(\prod_{t\,\in\,\mathcal{S}''}\Gamma_t(X,Z)\right).$$

And the right hand side of the above equation is at least η by Lemma 5.4. Hence for points considered in this sub-case, the lemma follows from (5.34) together with the fact that $\mu_{\alpha,\beta}(AB) = \mu_{\alpha,\beta}(A) + \mu_{\alpha,\beta}(B)$ for all $A, B \in \mathbb{F}_q[X,Z]$ and all $\alpha, \beta \in \mathbb{F}_q$.

Sub-Case 1.2. Suppose that $z_{i,j} \neq z_i$, i.e., the point$(x_i, z_{i,j})$ is not processed at iteration s. The enumeration of all points $(x_i, z_{i,j})$ with the same X coordinate in the algorithm dictates that $m_i^{(s)} \geq m_{x_i,z_{i,j}} - \eta$. Hence for points considered in this sub-case, the lemma follows from the same arguments applied in Sub-Case 1.1.

Case 2. Suppose $i \notin \mathcal{E}_s$, or equivalently, $i \in \mathcal{F}_s$, and we also assume that this point has been processed η times during iterations $0, 1, \ldots, s-1$ of Algorithm 2. Let us define the following auxiliary polynomial

$$u_{s,i}(X) = \frac{u_s(X)}{(X - x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+}}.$$

From the definition of $u_s(X)$ given in (5.31) and the fact that $i \in \mathcal{F}_s$, we can see that $u_{s,i}(X)$ is a well-defined polynomial. Let us also define $\Lambda(X,Z)$ as follows:

$$\Lambda(X,Z) = \prod_{0 \le t < s} \Gamma_t(X, \frac{Z}{X - x_i})(X - x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+ + \nu_i}, \tag{5.39}$$

where $\Gamma_t(X,Z)$'s are as defined in Lemma 5.4.

Thus according to (5.32), $(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X-x_i})$ can be written as follows:

$$(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X - x_i}) = \Lambda(X,Z) u_{s,i}(X)$$

Here we adopt the index sets defined in (5.35), (5.36), and (5.37). Apparently $(X - x_i)^{\nu_i} G_s(X, \frac{Z}{X-x_i})$ is a well-defined bivariate polynomial if we can establish that $\Lambda(X))$ is one. According to the definitions of $v_i(X)$'s and $\mathcal{S}'$, $(X - x_i) \,|\, v_t(X)$ for $t \in \mathcal{S}'$, so $\left(\prod_{t \in \mathcal{S}'} \left(\frac{Z}{X-x_i} v_t(X) - w_t(X)\right)\right)$ is a well-defined bivariate polynomial. In addition, since $(s - \sigma_i^{(s)})$ is positive, one can verify that

$$[s - \nu_i - \sigma_i^{(s)}]^+ + \nu_i - (s - \sigma_i^{(s)}) = [\nu_i - (s - \sigma_i^{(s)})]^+, \tag{5.40}$$

so $\prod_{t \in \mathcal{S} \setminus \mathcal{S}'} \Gamma_t(X, \frac{Z}{X-x_i})(X - x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+ + \nu_i}$is also a well-defined bivariate polynomial. Thus $\Lambda(X))$ is a well-defined bivariate polynomial.

We now proceed to prove the lemma for all points $(x_i, z_{i,j}) \in \mathcal{B}'$ such that $i \in \mathcal{F}_s$, while distinguishing between the following 2 sub-cases:

Sub-Case 2.1. Suppose that the remaining multiplicity of point $(x_i, z_{i,j})$ is 0 at iteration s. The correct execution of Algorithm 2 guarantees that this point has been processed exactly $m_{x_i,z_{i,j}}$ times in the previous s iterations, i.e. $\eta = m_{x_i,z_{i,j}}$. Since $\mathcal{S}'' \subseteq \mathcal{S}'$, we have

$$\mu_{x_i,z_{i,j}}\left(\prod_{t \in \mathcal{S}'} \Gamma_t(X,Z)\right) \ge \mu_{x_i,z_{i,j}}\left(\prod_{t \in \mathcal{S}''} \Gamma_t(X,Z)\right).$$

And the right hand side of the above equation is at least η by Lemma 5.4. Hence for points considered in this sub-case, the lemma follows from (5.39).

Sub-Case 2.2. Suppose that the remaining multiplicity of point $(x_i, z_{i,j})$ is greater than 0 at iteration s. It follows from (5.40) that

$$\begin{aligned} & \mu_{x_i,z_{i,j}}\Big(\prod_{t \in \mathcal{S}\backslash\mathcal{S}'} \Gamma_t\Big(X, \frac{Z}{X-x_i}\Big)(X-x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+ + \nu_i}\Big) \\ = \; & [\nu_i - (s-\sigma_i^{(s)})]^+. \end{aligned}$$

Similar to the previous cases, since $\mathcal{S}'' \subseteq \mathcal{S}'$, we have

$$\mu_{x_i,z_{i,j}}\left(\prod_{t \in \mathcal{S}'} \Gamma_t(X,Z) \right) \geq \mu_{x_i,z_{i,j}}\left(\prod_{t \in \mathcal{S}''} \Gamma_t(X,Z) \right).$$

And the right hand side of the above equation is at least η by Lemma 5.4.

If $\sigma_i^{(s)} = 0$, then $[\nu_i - (s-\sigma_i^{(s)})]^+ = [\nu_i - s]^+$, so no point at x_i has been processed up to iteration s. This happens only if the following condition is satisfied

$$\nu_i - s \geq m_{x_i,z_{i,j}},$$

thus the lemma follows from from (5.39) together with the fact that $\mu_{\alpha,\beta}(AB) = \mu_{\alpha,\beta}(A) + \mu_{\alpha,\beta}(B)$ for all $A, B \in \mathbb{F}_q[X,Z]$ and all $\alpha, \beta \in \mathbb{F}_q$. .

Otherwise, if $\sigma^{(s)} > 0$, then we must have

$$(s - \sigma_i^{(s)}) - \eta = \nu_i - m_{x_i,z_{i,j}}, \tag{5.41}$$

since $(s - \sigma_i^{(s)})$ is the number of times, during the previous s iterations, that the re-encoding point with X coordinate x_i has a remaining multiplicity that is not smaller than that of any other point with the same X coordinate. Then the lemma follows due to the fact that

$$\begin{aligned} & \eta + [\nu_i - (s - \sigma_i^{(s)})]^+ \\ = \; & \eta + [m_{x_i,z_{i,j}} - \eta]^+ \\ = \; & m_{x_i,z_{i,j}}, \end{aligned}$$

where the 1st equality above is derived from (5.41). ∎

Lemma 5.8. *Each of the polynomials produced by* Algorithm 2 *satisfies* (5.13). *In other words, for all $s = 0, 1, \ldots, r$, the polynomial $G_s(X, Z)$ in* (5.32) *can be expressed as*

$$G_s(X,Z) \;=\; \sum_{i=0}^{r} q_i(X)\, Z^i T_i(X)$$

Proof. For $i = 0, 1, \ldots, s$, let $p_i(X) \in \mathbb{F}_q[X]$ denote the coefficient of Z^i in $G_s(X, Z)$. Thus

$$p_s(X) \;=\; u_s'(X)\, u_s''(X)\, \Theta_s(X) \prod_{i\,\in\,\{\mathcal{A}_s\}_X} \big(X - x_i\big)^{m_i^{(s)}} \prod_{j=0}^{s-1} v_j(X)$$

We first show that $T_s(X)$ divides $p_s(X)$. To this end, we consider the k terms in the product of (5.12) and prove that each of them divides $p_s(X)$, while distinguishing between three cases.

Case 1. Suppose that $i \in \mathcal{D}$. Then the term $(X - x_i)^{[s-\mu_i]^+}$ divides $\Theta_s(X)$ by (5.25); hence, it also divides $p_s(X)$.

Case 2. Suppose that $i \in \mathcal{F}_s$. Then $(X - x_i)^{[s-\mu_i-\sigma_i^{(s)}]^+}$ divides $u_s'(X)$, while $(X - x_i)^{\sigma_i^{(s)}}$ divides $\prod_{j=0}^{s-1} v_j(X)$ by the definition of σ_i. Since $[s - \mu_i - \sigma_i^{(s)}]^+ + \sigma_i^{(s)} \geq [s - \mu_i]^+$, it follows that the term $(X - x_i)^{[s-\mu_i]^+}$ divides $p_s(X)$.

Case 3. Suppose that $i \in \mathcal{E}_s$. Then $(X - x_i)^{(m_i^{(s)}+s-\mu_i-\sigma_i^{(s)})}$ divides $u_s''(X)$ and $(X - x_i)^{\sigma_i^{(s)}}$ divides $\prod_{j=0}^{s-1} v_j(X)$. Again, since

$$(m_i^{(s)} + s - \mu_i - \sigma_i^{(s)}) + \sigma_i^{(s)} \;\geq\; [s - \mu_i]^+$$

it follows that the term $(X - x_i)^{[s-\mu_i]^+}$ divides $p_s(X)$. This exhausts all the possible cases for a code position $i \in \{\mathcal{R}\}_X$.

We are still required to show that $T_j(X)$ divides $p_j(X)$ for all $j = 1, 2, \ldots, s-1$ (note that $T_0(X) = 1$). This follows by the same argument as above, along with the following observation. Let $\mathcal{S}$ be an arbitrary subset of $\{0, 1, \ldots, s-1\}$ of size $s - \delta$; if $(X - x_i)^{\sigma_i}$ divides $\prod_{j=0}^{s-1} v_j(X)$, then $(X - x_i)^{\sigma_i - \delta}$ necessarily divides $\prod_{j\,\in\,\mathcal{S}} v_j(X)$. Hence the foregoing argument is applicable for all $j = s - \delta$, with $\delta = 1, 2, \ldots, s - 1$. ∎

We can summarize the lemmas proved so far into the following theorem.

Theorem 5.9 *The $G_s(X,Z)$'s constructed from Algorithm 2 has the following property*

$$\langle \mathcal{G} \rangle \subseteq \langle \mathcal{J}(\mathcal{P}', M') \rangle_r.$$

We now proceed to show that $\langle \mathcal{J}(\mathcal{P}', M') \rangle_r \subseteq \langle \mathcal{G} \rangle$.

Lemma 5.10 *Any bivariate polynomial $A(X,Z)$ that passes point (x,z) with multiplicity m can be written as*

$$A(X,Z) = (Z-z)B(X,Z) + (X-x)^m b(X).$$

Proof. Divide $A(X,Z)$ by $(Z-z)$, we get $A(X,Z) = (Z-z)B(X,Z) + a(X)$. Since $A(X+x, Z+z) = ZB(X+x, Z+z) + a(X+z)$ and the 1st term on the right hand side of this equation does not have any monomials in X only, we must have $X^m \, |a(X+z)$, i.e. $(X-x)^m \, |a(X)$. ∎

Lemma 5.11 *For any bivariate polynomial $A(X,Z) \in \langle \mathcal{J}(\mathcal{P}', M') \rangle_r$ of Y-degree s, where $0 \le s \le r$. If we write $A(X,Z)$ as $A(X,Z) = \sum_{j=0}^{s} q_j(X) Z^j$, we must have*

$$u_s(X) \prod_{i=0}^{s-1} v_i(X) \, |q_s(X) \, ,$$

where $u_s(X)$ and the $v_i(X)$'s are defined in Algorithm 2.

Proof. By definitions of $u_s(X)$ and $v_i(X)$'s given in Algorithm 2, and the fact that $\mathcal{E}_s \bigcup \mathcal{F}_s = \mathcal{B}'$, we have

$$\begin{aligned} & u_s(x) \prod_{i=0}^{s} v_i(X) \qquad (5.42) \\ = & \prod_{i \in \{\mathcal{A}_s\}_X} (X-x_i)^{m_i^{(s)}} \prod_{i \in \mathcal{F}_s} (X-x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+ + \sigma_i^{(s)}} \\ & \prod_{i \in \mathcal{E}_s} (X-x_i)^{(m_i^{(s)}+s-\nu_i-\sigma_i^{(s)})+\sigma_i^{(s)}} \prod_{i \in \mathcal{D}} (X-x_i)^{[s-\nu_i]^+} \end{aligned}$$

We will show that every $(X-x_i)^{(\bullet)}$ term in the above equation divides $q_s(X,Z)$. To this end, we fix i in the rest of the proof while distinguishing between the following cases. let us define the following series of sub-modules, for

$t = 0, 1, \ldots, s$,

$$\mathcal{J}_t(x_i) \overset{\text{def}}{=} \left\{ \begin{array}{c|c} Q(X,Z) & \mu_{x_i,z_{i,j}}\big(Q(X,Z)\big) \geq m_{x_i,z_{i,j}} \\ \in \langle \mathbb{F}_q[X,Y] \rangle_r & \text{for all } (x_i, z_{i,j}) \in \mathcal{A}_t \text{ or } \mathcal{B}' \end{array} \right\}, \tag{5.43}$$

where the $\mathcal{A}_t$'s, $\mathcal{B}'$'s and $m_{x_i,z_{i,j}}$'s are as in Algorithm 2.

Case 1. Suppose that $i \in \{\mathcal{A}_s\}_X$. We need to show that $(X - x_i)^{m_i^{(s)}} | q_s(X, Z)$. For the fixed i, let $m_i^{(s)}$ be as defined in (5.26) for each iteration. Since $A(X, Z) \in \langle \mathcal{J}(\mathcal{P}', M') \rangle_r$, it is also in $\mathcal{J}_0(x_i)$. We can write $A(X, Z)$ as

$$A(X,Z) = (Z - z_{i,j})A_0(X,Z) + (X - x_i)^{m_i^{(0)}} a_0(X),$$

according to Lemma 5.10. We should emphasize that the $m_i^{(0)}$ is as in iteration 0 of Algorithm 2. Apparently the 2nd term on the right hand side of the equation above belongs to $\mathcal{J}_0(x_i)$, thus we can subtract it from $A(X, Z)$. Now we have Y-deg $A_0(X, Z) = s - 1$ and $A_0(X, Z) \in \mathcal{J}_1(x_i)$, where $\mathcal{J}_1(x_i)$ is as defined in (5.43). Now we can write $A_0(X, Z)$ as $A_0(X, Z) = (Z - z_{i,j})A_1(X, Z) + (X - x_i)^{m_i^{(1)}} a_1(X)$, and here $m_i^{(1)}$ is as in iteration 1 of Algorithm 2. Repeat the argument $s - 1$ times, and we finally have $A_{s-1}(X, Z) \in \mathcal{J}_s(x_i)$. Thus $(X - x_i)^{m_i^{(s)}} | A(X, Z)$.

Case 2. Suppose that $i \in \{\mathcal{B}'\}_X$. We still need to differentiate between the following 2 sub-cases.

Sub-Case 2.1. Suppose that $i \in \mathcal{E}_s$ and let $A'(X, Z) = (X - x_i)^{\nu_i} A(X, \frac{Z}{X - x_i})$. Here we need to prove that $(X - x_i)^{(m_i^{(s)} + s - \nu_i - \sigma_i^{(s)}) + \sigma_i^{(s)}} | q_s(X)$

Since $A(X, Z) \in \langle \mathcal{J}(\mathcal{P}', M') \rangle_r$, we must have $A'(X, Z) \in \mathcal{J}_0(x_i)$ as well. Given Lemma 5.10, we can write $A'(X, Z)$ as follows

$$A'(X,Z) = (Z - z_{i,j})A'_0(X,Z) + (X - x_i)^{m_i^{(0)}} a_0(X).$$

Applying the same arguments as for Case 1, we can prove that

$$(X - x_i)^{m_i^{(s)}} \big| A'(X,Z) \, .$$

Let us write $A'(X, Z) = \sum_{j=0}^{s} q'_j(X) Z^j$, thus $(X - x_i)^{m_i^{(s)}} \Big| q'_j(X)$ for $0 \leq j \leq s$. Since $A(X, Z) = \sum_{j=0}^{s} q_j(X) Z^j$, $q_j(X)$ must have the following form

$$q_j(X) = q'_j(X)(X - x_i)^{j - \nu_i}.$$

The fact that $(X-x_i)^{m_i^{(s)}} \Big| q_j'(X)$ leads to $(X-x_i)^{m_i^{(s)}+j-\nu_i} | q_j(X)$ for $0 \le j \le s$. In particular, we have $(X-x_i)^{m_i^{(s)}+s-\nu_i} | q_s(X)$ as the desired result.

Sub-Case 2.2. Suppose that $i \in \mathcal{F}_s$. Here we need to prove that

$$(X-x_i)^{[s-\nu_i-\sigma_i^{(s)}]^+ + \sigma_i^{(s)}} | q_s(X)\ .$$

The arguments to be used here are similar in spirit to those used in Sub-Case 2.1. There are 2 possibilities:

If $\sigma_i^{(s)} = 0$, i.e., no point in $\mathcal{B}'$ at x_i has been processed in the previous s iterations, we must have $s \le \nu_i$, thus $[s-\nu_i-\sigma_i^{(s)}]^+ + \sigma_i^{(s)} = 0$ and there is nothing to be proved.

Otherwise, if $\sigma_i^{(s)} > 0$, it is still true that $s-\nu_i-\sigma_i^{(s)} \le 0$. Thus $[s-\nu_i-\sigma_i^{(s)}]^+ + \sigma_i^{(s)} = \sigma_i^{(s)}$. Let $A'(X,Z) = (X-x_i)^{\nu_i} A(X, \frac{Z}{X-x_i})$. By applying the same arguments as we have for Case 2.1, we can show that

$$(X-x_i)^{(m_i^{(s)}+s-\nu_i)} | q_s(X)\ .$$

To see that

$$m_i^{(s)} + s - \nu_i = \sigma_i^{(s)}, \tag{5.44}$$

we can write $m_i^{(s)} + s - \nu_i = \left(m_i^{(s)} - (\nu_i - (s - \sigma_i^{(s)}))\right) + \sigma_i^{(s)}$. As we have mentioned earlier, the value $(s-\sigma_i^{(s)})$ represents the number of times that the reencoding point at x_i has the largest multiplicity during the previous s iterations, and the value $(\nu_i - (s-\sigma_i^{(s)}))$ denotes the remaining multiplicity of the corresponding re-encoding point. The enumeration of Algorithm 2 at i, where the re-encoding point is always given priority when a tie in multiplicity occurs, guarantees that $\left(m_i^{(s)} - (\nu_i - (s-\sigma_i^{(s)}))\right) = 0$, so (5.44) follows.

Case 3. Suppose that $i \in \mathcal{D}$. Given that $A(X,Z) \in \langle \mathcal{J}(\mathcal{P}', M') \rangle_r$, it can be expressed in the form of (5.13). Thus $(X-x_i)^{[s-\nu_i]^+}$ must be a factor of $q_s(X)$.

In summary, we have shown that every $(X-x_i)^{(\bullet)}$ term in (5.42) divides $q_s(X)$, thus the lemma is proved. ∎

The lemma is key to the proof of the following theorem.

Theorem 5.12. *For any $A(X,Z) \in \langle \mathcal{J}(\mathcal{P}', M')\rangle_r$, it can expressed as a linear combination of the $G_s(X,Z)$'s constructed from Algorithm 2, i.e., $A(X,Z) \in \langle \mathcal{G}(X,Z)\rangle$.*

Proof. . Without loss of generality, let us assume that Y-deg $A(X,Z) = t$. Given Lemma 5.11, we know that $A(X,Z)$ can be written as follows

$$\begin{aligned} A(X,Z) &= a_t(X)u_t(X)\prod_{s=0}^{t}\big(Zv_s(X) - w_s(X)\big) + A_{t-1}(X,Z) \\ &= a_t(X)G_t(X,Z) + A_{t-1}(X,Z). \end{aligned}$$

Since $G_t(X,Z) \in \langle \mathcal{J}(\mathcal{P}', M')\rangle_r$, we have $A_{t-1}(X,Z) \in \langle \mathcal{J}(\mathcal{P}', M')\rangle_r$ and Y-deg $A_{t-1}(X,Z) < t$, thus the theorem is proved by induction on t. ∎

Given Theorem 5.9 and Theorem 5.12, it immediately follows that

$$\left\langle \mathcal{J}(\mathcal{P}', M')\right\rangle_r = \langle G_0, ..., G_r\rangle\,. \tag{5.45}$$

5.4 Properties of the Bases

The basis computed from the new basis construction algorithm (Algorithm 2) and the one constructed by the original Lee-O'Sullivan algorithm (Algorithm 1) are closely related. This is the subject for the rest of the section.

Lemma 5.13. *Let the polynomials $v_s(X)$ and $w_s(X)$ be as defined in* Algorithm 2*. Let $b_s(X)$ be the polynomial defined in* Algorithm 1*. Then for all $s = 0,1,\ldots,r$, we have*

$$\frac{w_s(X)\psi(X)}{v_s(X)} = b_s(X) \tag{5.46}$$

Proof. From the definition of $w_s(X)$ given in (5.28), we can write the left hand side of (5.46) as follows

$$\begin{aligned}\omega_s(X)\frac{\psi(X)}{v_s(X)} &= \sum_{j\,\in\,\{\mathcal{A}_s\}_X} z_j h_j(X)\psi(X) \\ &+ \sum_{j\,\in\,\mathcal{E}_s} z_j g_j(X)\frac{\psi(X)}{(X-x_j)} \\ &= \sum_{j\,\in\,\{\mathcal{A}_s\}_X} z_j\psi(x_j)\frac{h_j(X)\psi(X)}{\psi(x_j)} \\ &+ \sum_{j\,\in\,\mathcal{E}_s} z_j\psi'(x_j)g_j(X)\frac{\psi(X)}{\psi'(x_j)(X-x_j)}.\end{aligned}$$

In addition, from the definitions of $\psi(X)$, $h_j(X)$ and $g_j(X)$ given by (5.6), (5.23) and (5.24), we can get the following

$$\frac{h_j(X)\psi(X)}{\psi(x_j)} = \prod_{l\,\in\,\{\mathcal{A}\}_X, l\neq j} \frac{X-x_l}{x_j-x_l}\frac{\prod_{l'\,\in\,\{\mathcal{R}\}_X}(X-x_{l'})}{\prod_{l'\,\in\,\{\mathcal{R}\}_X}(x_j-x_{l'})}$$

$$\frac{g_j(X)\psi(X)}{\psi'(x_j)(X-x_j)} = \prod_{l\,\in\,\{\mathcal{A}\}_X} \frac{X-x_l}{x_j-x_l}\frac{\prod_{j'\,\in\,\{\mathcal{R}\}_X, j'\neq j}(X-x_{j'})}{\prod_{j'\,\in\,\{\mathcal{R}\}_X, j'\neq j}(x_j-x_{j'})}.$$

Thus by definition of $l_i(X)$'s given by (5.19), we have

$$l_j(X) = \begin{cases} h_j(X)\psi(X)/\psi(x_j) & \text{for } i\in\{\mathcal{A}\}_X \\ g_j(X)\psi(X)/\psi'(x_j)(X-x_j) & \text{for } i\in\{\mathcal{B}\}_X \end{cases}. \tag{5.47}$$

Combining all of above, we obtain

$$\frac{w_s(X)\psi(X)}{v_s(X)} = \sum_{j\,\in\,\{\mathcal{A}_s\}_X} z_j\psi(x_j)l_j(X) + \sum_{j\,\in\,\mathcal{E}_s} z_j\psi'(x_j)l_j(X) \tag{5.48}$$

We now use the inverse of (5.8) to express $z_i\psi(x_i)$ and $z_i\psi'(x_i)$ as y_i, where y_i is as defined in Algorithm 1. Note that the difference between $\{\mathcal{P}_s\}_X$ and $\{\mathcal{A}_s\}_X \cup \mathcal{E}_s$ corresponds precisely to the re-encoding points, for which $y_i = 0$ by (5.5). Hence, the right-hand side of (5.48) can be expressed as $\sum_{i\,\in\,\{\mathcal{P}_s\}_X} y_i l_i(X)$, and the lemma follows from the definition of $b_s(X)$ in Algorithm 1. ∎

Lemma 5.14. *Let the polynomials $u_s(X)$ and $v_s(X)$ be as defined in* Algorithm 2. *Then for all $s = 0, 1, \dots, r$, we have*

$$\frac{u_s(X)\phi(X)}{\big(\psi(X)\big)^s}\prod_{i=0}^{s-1} v_i(X) = \prod_{i\,\in\,\{\mathcal{P}_s\}_X}\big(X - x_i\big)^{m_i} = a_s(X) \tag{5.49}$$

Proof. From the definitions of $v_i(X)$'s given in (5.27), we can write $\prod_{i=0}^{s-1} v_i(X)$ as

$$\prod_{i=0}^{s-1} v_i(X) = \prod_{l'\,\in\,\mathcal{F}_s}(X - x_{l'})^{\sigma_{l'}^{(s)}}\prod_{l\,\in\,\mathcal{E}_s}(X - x_l)^{\sigma_l^{(s)}}.$$

Combining with the definition of $u_s(X)$, $\psi(X)$ and $\phi(X)$ in (5.31), (5.6) and (5.7), we can write out the leftmost term in (5.49) as follows

$$\begin{aligned}
&\frac{u_s(X)\phi(X)}{\big(\psi(X)\big)^s}\prod_{i=0}^{s-1} v_i(X)\\
=\ &\prod_{l'\,\in\,\mathcal{F}_s}(X - x_{l'})^{[s-\nu_{l'}-\sigma_{l'}^{(s)}]^+ + \sigma_{l'}^{(s)} + \nu_{l'} - s}\\
&\prod_{l\,\in\,\mathcal{E}_s}(X - x_l)^{(m_l+s-\nu_l-\sigma_l^{(s)}) + \sigma_l^{(s)} + \nu_l - s}\\
&\Theta_s(X)\prod_{j\,\in\,\{\mathcal{A}_s\}_X}(X - x_j)^{m_j}
\end{aligned}$$

As we have shown before,

$$\sigma_{l'}^{(s)} + \nu_{l'} - s + [s - \nu_{l'} - \sigma_{l'}^{(s)}]^+ = [\nu_{l'} + \sigma_{l'}^{(s)} - s]^+,$$

and $\sigma_l^{(s)} + \nu_l - s + (m_l + s - \nu_{l'} - \sigma_l^{(s)}) = m_l$ for all $l' \in \mathcal{F}_s$ and $l \in \mathcal{E}_s$ respectively, the equation above can be rewritten as

$$\begin{aligned}
&\frac{u_s(X)\phi(X)}{\big(\psi(X)\big)^s}\prod_{i=0}^{s-1} v_i(X)\\
=\ &\prod_{l\,\in\,\mathcal{F}_s}(X - x_l)^{[\nu_l+\sigma_l^{(s)}-s]^+}\prod_{l\,\in\,\mathcal{E}_s}(X - x_l)^{m_l}\\
&\prod_{l\,\in\,\mathcal{D}}(X - x_l)^{[\nu_l-s]^+}\prod_{j\,\in\,\{\mathcal{A}_s\}_X}(X - x_j)^{m_j}.
\end{aligned}$$

We can write $\prod_{l\,\in\,\mathcal{D}}(X - x_l)^{[\nu_l-s]^+}$ as $\prod_{l\,\in\,\mathcal{D}_s}(X - x_l)^{\nu_l-s}$, where $\mathcal{D}_s = \{l \in \mathcal{D} : \nu_l > s\}$. A careful examination of the enumeration of points at each index reveals that $\{\mathcal{P}_s\}_X = \{\mathcal{A}_s\}_X \bigcup \mathcal{D}_s \bigcup \mathcal{E}_s \bigcup \mathcal{F}_s$, and the lemma follows. ∎

Lemma 5.15. *The basis polynomials produced by* Algorithm 1 *and* Algorithm 2 *are related via the maps* ξ *and* χ *in* (5.16) *and* (5.17). *That is, for all* $s = 0, 1, \ldots, r$, *we have*

$$B_s(X, Y) = \xi\big(G_s(X, Z)\big) \quad \text{and} \quad G_s(X, Z) = \chi\big(B_s(X, Z)\big) \tag{5.50}$$

Proof. Let $H_s(X, Y) \overset{\text{def}}{=} \xi\big(G_s(X, Z)\big)$. Combining the definition of $G_s(X, Z)$ in (5.32) with the definition of the mapping ξ in (5.16), we find that $B_s(X, Y)$ can be expressed as

$$\frac{u_s(X)\phi(X)}{\big(\psi(X)\big)^s} \prod_{i=0}^{s-1} v_i(X) \prod_{i=0}^{s-1} \left(Y - \frac{w_i(X)\psi(X)}{v_i(X)} \right)$$

It now follows from Lemma 5.13 and Lemma 5.14, along with (5.21), that $H_s(X, Y) = B_s(X, Y)$ as claimed. Note that, once again, ξ and χ take polynomials to polynomials in this case. ∎

If $\mathcal{G}(X, Y) = a(X)G_s(X, Y) + b(X)G_t(X, Y)$ and $\mathcal{B}(X, Y) = a(X)B_s(X, Y) + b(X)P_t(X, Y)$, for arbitrary $a(X)$ and $b(X)$,it's easy to verify that

$$\mathcal{B}(X, Y) = \phi(X)\mathcal{G}(X, \frac{Y}{\psi(X)}). \tag{5.51}$$

Thus the mapping ξ and χ define an isomorphism between the 2 submodules:

$$\begin{aligned} \langle \mathcal{I}(\mathcal{P}, M) \rangle_r &= \langle B_0(X, Y), ...B_r(X, Y) \rangle \\ \big\langle \mathcal{J}(\mathcal{P}', M') \big\rangle_r &= \langle G_0(X, Y), ..., G_r(X, Y) \rangle . \end{aligned}$$

Let us assume that the $G_s(X, Y)$'s and the $B_s(X, Y)$'s can be expanded as follows, for $0 \le s \le r$:

$$G_s(X, Z) = \sum_{i=0}^{s} g_{s,i}(X) Z^i,$$

$$B_s(X, Z) = \sum_{i=0}^{s} p_{s,i}(X) Y^i.$$

Apparently we have

$$p_{s,i}(X) = \frac{\phi(X) g_{s,i}(X)}{\psi(X)^i}. \tag{5.52}$$

Thus we have

$$\begin{aligned} & \deg_{1,k-1} p_{s,i}(X)Y^i \qquad (5.53)\\ = \; & \deg \frac{\phi(X)g_{s,i}(X)}{\psi(X)^i} + (k-1)i \\ = \; & \deg \phi(X)g_{s,i}(X) - \deg \psi(X)^i + (k-1)i \\ = \; & \deg \phi(X) + \deg g_{s,i}(X) - ki + (k-1)i \\ = \; & \deg \phi(X) + \deg g_{s,i}(X) - i \\ = \; & \deg \phi(X) + \deg_{1,-1} g_{s,i}(X)Z^i \end{aligned}$$

Let us use $\mathrm{LT}_{a,b}()$ to denote the leading term of a bivariate polynomial, with respect to the (a,b)-weighted monomial order. From (5.53), we can further derive that, for $0 \le s \le r$,

$$\text{y-deg}\big(\mathrm{LT}_{1,k-1}(B_s(X,Y))\big) = \text{y-deg}\big(\mathrm{LT}_{1,-1}(G_s(X,Y))\big) \qquad (5.54)$$

The following curious property of the basis produced by Algorithm 1 was apparently overlooked by Lee and O'Sullivan [LO06a].

Theorem 5.16. *The basis* $\mathcal{B} = \{B_0(X,Z), B_1(X,Z), \ldots, B_r(X,Z)\}$ *produced by* Algorithm 1 *is a Groebner basis for the module* $\langle \mathcal{I}(\mathcal{P}, M)\rangle_r$ *with respect to the* $\prec_{n-1}$ *monomial order.*

Proof. By Proposition 12 of Lee and O'Sullivan [LO06b], it would suffice to show that the Y-degrees of the leading terms of the polynomials $B_0(X,Y), B_1(X, Y), \ldots, B_r(X,Y)$ with respect to $\prec_{n-1}$ are all distinct. But since $\deg b_i(X) \le n-1$, the leading term of $B_s(X,Y)$ with respect to $\prec_{n-1}$ is $a_s(X)Y^s$. ∎

Theorem 5.17. *The basis* $\mathcal{G} = \{G_0(X,Z), G_1(X,Z), ..., G_r(X,Z)\}$ *produced by* Algorithm 2 *is a Groebner basis for the module* $\langle \mathcal{J}(\mathcal{P}', M')\rangle_r$ *with respect to the* $\prec_{n-k-1}$ *monomial order.*

Proof. Again, in view of Proposition 12 of [LO06b], it would suffice to show that for all $s = 0, 1, \ldots, r$, the Z-degree of the leading term of $G_s(X,Z)$ is precisely s. To this end, let us analyze the degree of the polynomials $w_i(X)$ in (5.32).

By (5.23), we have

$$\deg h_i(X) = |\{\mathcal{A}\}_X| - 1 \; = \; n - |\{\mathcal{R}\}_X| - 1 \; = \; n - k - 1$$

for all $i \in \{\mathcal{A}\}_X$. Hence, the degree of the first sum in (5.28) is at most $\deg v_s(X)$ $+n-k-1$. Similarly, we conclude from (5.6) that $\deg g_i(X) = n-k$ for all i. Hence, the degree of the second sum in (5.28) is also bounded by $\deg v_s(X) + n$ $-k-1$. It follows that $\deg w_i(X) \le \deg v_i(X) + (n-k-1)$ for all i, so that $w_i(X) \prec_{n-k-1} Zv_i(X)$. Therefore, the leading term of $G_s(X,Z)$ with respect to $\prec_{n-k-1}$ is $u_s(X)Z^s\prod_{i=0}^{s-1} v_i(X)$. ∎

Thus in both cases, Step 2 of the Lee-O'Sullivan algorithm converts a Groebner basis with respect to a "wrong" monomial order (either $\prec_{n-1}$ or $\prec_{n-k-1}$) into a Groebner basis with respect to the desired monomial order (either $\prec_{k-1}$ or $\prec_{-1}$). The algorithm is repeated below.

Algorithm 3 (Groebner Basis Algorithm) *Input: $B_s(X,Y)$ for $s = 0,1,...,r$.*

```
I1. Set $l \leftarrow 0$.
I2. Set $l := l+1$. If $l \le r$, then proceed; otherwise go to step I6.
I3. Find $s = y\text{-}deg(LT(P_l))$. If $s = l$, then go to step I2.
I4. Set $d \leftarrow \deg(p_{l,s}) - \deg(p_{s,s})$ and $c \leftarrow \frac{LC(p_{l,s})}{LC(p_{s,s})}$.
I5. $\begin{pmatrix} P_r \\ B_s \end{pmatrix} := \begin{cases} \begin{pmatrix} 1 & -cX^d \\ 0 & 1 \end{pmatrix}\begin{pmatrix} P_r \\ B_s \end{pmatrix} & \text{if } d \ge 0 \\ \begin{pmatrix} X^{-d} & -c \\ 1 & 0 \end{pmatrix}\begin{pmatrix} P_r \\ B_s \end{pmatrix} & \text{else} \end{cases}$, then go back to step I3.
I6 Let $P(X,Y)$ be the $B_s(X,Y)$ with the smallest leading term. Output $P(X,Y)$
    and the algorithm terminates.
```

In the following, we show that the above algorithm has the following property:

Theorem 5.18 *For any 2 sets of polynomials $B_s(X,Y)$'s and $G_s(X,Y)$'s related by the mappings χ and ξ, when the Groebner basis algorithm is applied to $B_s(X,Y)$'s with respect to $(1,k-1)$-weighted degree, and applied to $G_s(X,Y)$'s with respect to $(1,-1)$-weighted degree, the resulting $B_s(X,Y)$'s and $G_s(X,Y)$'s still have the relationship as defined by (5.50). In addition, all intermediate $B_s(X,Y)$'s and $G_s(X,Y)$'s have the same relationship.*

Proof. The theorem essentially says that the mappings χ and ξ are preserved throughput the execution of the algorithm. Let us now examine a side-by-side application of the algorithm, with appropriate weighted-degree, to $B_s(X,Y)$'s and $G_s(X,Y)$'s. In step I3, due to (5.54), the same s will be found on both sides. In step I4, we have

$$\begin{aligned} & \deg p_{l,s}(X) - \deg p_{s,s}(X) \\ = \; & \deg \frac{\phi(X) g_{l,s}(X)}{\psi(X)^s} - \deg \frac{\phi(X) g_{s,s}(X)}{\psi(X)^s} \\ = \; & \deg g_{l,s}(X) - \deg g_{s,s}(X) \end{aligned}$$

Thus the same d will result from both sides. In addition, we have

$$\begin{aligned} \frac{\mathrm{LC}(p_{l,s})}{\mathrm{LC}(p_{s,s})} &= \frac{\mathrm{LC}(\frac{\phi(X) g_{l,s}(X)}{\psi(X)^s})}{\mathrm{LC}(\frac{\phi(X) p_{s,s}(X)}{\psi(X)^s})} \\ &= \frac{\frac{\mathrm{LC}(\phi(X))\mathrm{LC}(g_{l,s}(X))}{\mathrm{LC}(\psi(X)^s)}}{\frac{\mathrm{LC}(\phi(X))\mathrm{LC}(g_{s,s}(X))}{\mathrm{LC}(\psi(X)^s)}} = \frac{\mathrm{LC}(g_{l,s})}{\mathrm{LC}(g_{s,s})} \end{aligned}$$

This shows that the same c will be obtained from both sides. Since d and c computed at both sides are identical, the mapping remains after the operation done at step I5 if we further apply the property of (5.51). In summary, since (5.50) holds for the input to both sides and it is preserved after each iteration of the algorithm, the mapping certainly holds for the output from both sides. ▮

5.5 Final Remarks

As we can see from Example 5.3, significant savings in computational complexity can be achieved with the modified Lee-O'Sullivan algorithm. However, the reduction in complexity is less impressive than for Koetter's algorithm. Actually, it only takes 350×10^3 multiplications for Koetter's algorithm to solve the reduced interpolation problem in Example 5.2. So why is the Lee-O'Sullivan algorithm more efficient in solving the original interpolation problem but loses to Koetter's algorithm with respect to the reduced interpolation problem? A heuristic explanation for this is given below.

Example 5.3, shows that more reduction in complexity can be achieved for Step 1 (basis construction) than for Step 2 (basis reduction) of the Lee-O'Sullivan algorithm. Without the re-encoding coordinate transformation, Step 1 of the algorithm takes many more multiplications than Step 2. However, after re-encoding, the majority of the modified Lee-O'Sullivan algorithm's computational complexity comes from the second step. With the re-encoding coordinate transformation, the basis construction algorithm presented in Section 5.3 requires the same number of iterations as the original basis construction algorithm of [LO06a]. In addition, in the process of proving Theorem 5.18, we have shown that the number of iterations required for the Groebner basis algorithm in solving the reduced interpolation problem is the same as what is required to solve the original interpolation problem. In other words, when the modified Lee-O'Sullivan algorithm is applied to the reduced interpolation problem, the complexity reduction only comes from the fact that the polynomials become much smaller. However, this is not the case for Koetter's algorithm, wherein the number of iterations are also significantly reduced when the algorithm is applied to the reduced interpolation problem.

Thus with the re-encoding coordinate transformation, Koetter's algorithm appears to be more efficient than the modified Lee-O'Sullivan algorithm. However, this assumes that polynomial multiplication, which is the main operation in the second step of the Lee-O'Sullivan algorithm, is done in the naïve way. An interesting open problem for future research is whether such polynomial multiplication can be made more efficient (in the non-asymptotic sense). We should also mention that even in the case of solving $\mathbf{IP}_{1,k-1}(\mathcal{P},\mathcal{M})$, the Lee-O'Sullivan's algorithm is not always more efficient than Koetter's algorithm.

Actually, it has larger computational complexity when the RS code of interest has a much lower code rate.

The results of the Chapter 5 have been presented, in part, at *2007 International Symposium on Information Theory (ISIT)*, Ma, Jun; Vardy, Alexander. The dissertation author was primary investigator and author of the paper.

CHAPTER 6

Re-Encoder Design for an (255,239) Reed-Solomon Code

The most computationally demanding step in soft-decision decoding of RS codes is bivariate polynomial interpolation. The re-encoding and coordinate transformation based technique can significantly reduce the computation complexity of the original interpolation problem, thus making the algebraic soft-decision decoder practically feasible. In this chapter, an implementation of the re-encoding coordinate transformation procedure is presented. The novelties of our design include a fast algorithm to determine the re-encoding points, an area efficient erasure-only RS decoding architecture, and an overlapped scheduling of the various procedures required for the re-encoding process to reduce the overall latency. The synthesis result shows that the proposed design is sufficiently fast for any existing or developing interpolation architecture.

6.1 Introduction and Background

Algebraic soft-decision decoding of Reed-Solomon (RS) codes delivers promising coding gains over conventional hard-decision decoding. The most computationally demanding step in soft-decision decoding of RS codes is bivariate polynomial interpolation. The re-encoding coordinate transformation based technique [KMVA03] can significantly reduce the computation complexity of the interpolation problem. Though a number of literatures have presented efficient architectures for the interpolation and factorization algorithms, there is no prior work on implementation of the re-encoding coordinate transformation techniques, which is a crucial first-step of any practical soft-decision RS decoders. In this chapter, an efficient implementation of the re-encoding coordinate transformation function of a $(255, 239)$ soft-decision RS decoder is presented. The proposed design can be easily extended to RS codes of other rates and is sufficiently fast for practical applications.

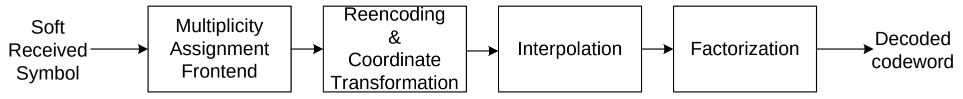
Soft Received Symbol
Multiplicity Assignment Frontend
Reencoding & Coordinate Transformation
Interpolation
Factorization
Decoded codeword

Figure 6.1: Block Diagram of the Soft-Decision Reed-Solomon Decoder

A block diagram of soft-decision RS decoder is shown in Figure 6.1. In our implementation, the re-encoding coordinate transformation block assumes that the original set of interpolation points are generated by a multiplicity-assignment function such that there are at most 2 interpolation points with the same X coordinate. This has been shown to have very negligible performance loss compared to schemes with no limitations on the number of interpolation points for the same X coordinate. Let $\{x_i, i = 0, 1, ..., 254\}$ denote the set of X coordinates, $\{y_{i,j}, i = 0, 1, ..., 254, j = 0, 1\}$ for the set of the Y coordinates, and $\{m_{i,j} : i = 0, 1, ..., 254, j = 0, 1\}$ denote the corresponding multiplicities and we assume $m_{i,0} \geq m_{i,1}$, i.e., $(x_i, y_{i,0})$ is of higher multiplicity than $(x_i, y_{i,1})$ for all i. The re-encoding coordinate transformation block first determines 239 interpolation points of highest multiplicities with distinct X coordinates. Let us denote the indices i of the 239 X coordinates as a set $\{\mathcal{R}\}_X$ and denote the rest of the 255 indices as a set $\{\mathcal{A}\}_X$. Apparently we have $\{\mathcal{R}\}_X \Cup \{\mathcal{A}\}_X = \{i : i = 0, ..., 254\}$. We also assume in the scope of this work that $m_{i,1} = 0$ for all $i \in \{\mathcal{R}\}_X$. In the rest of the chapter, we refer points $\{(x_i, y_{i,0}) : i \in \{\mathcal{R}\}_X\}$ as re-encoding points, X coordinates $\{x_i : i \in \{\mathcal{R}\}_X\}$ as re-encoding X coordinates and indices $\{i : i \in \{\mathcal{R}\}_X\}$ as re-encoding indices or re-encoding positions. Correspondingly, we refer $\{(x_i, y_{i,0}) : i \in \{\mathcal{A}\}_X\}$, $\{x_i : i \in \{\mathcal{A}\}_X\}$ and $\{i : i \in \{\mathcal{A}\}_X\}$ as interpolation points, interpolation X coordinates and interpolation indices, respectively. The re-encoding function then finds a valid codeword $C'(X) = \sum_{i=0}^{i=254} c'_i X^i$ such that $c'_i = y_{i,0}$ for all $i \in \{\mathcal{R}\}_X$. This can be achieved with an erasure-only decoding algorithm as suggested in [GKKG05]. After this the Y coordinates for all of the original interpolation points are *shifted* as follows:

$$(x_i, y_{i,j}) \rightarrow (x_i, y'_{i,j} = y_{i,j} - c'_i). \tag{6.1}$$

As can be seen, points $(x_i, y'_{i,j})$ have a non-zero Y coordinate only for $i \in \{\mathcal{A}\}_X$. We then carry out a coordinate transform for these points with non-zero Y coordinates as follows:

$$(x_i, y'_{i,j}) \rightarrow (x_i, z_{i,j} = \frac{y'_{i,j}}{V(x_i)}), \tag{6.2}$$

where $i \in \{\mathcal{A}\}_X$ and $V(X) \stackrel{\text{def}}{=} \prod_{i \in \{\mathcal{R}\}_X} (X - x_i)$.

An architectural block diagram of our implementation of the re-encoding coordinate transformation algorithm is given in Figure 6.2. We assume that the two Y coordinates and associated multiplicities of each interpolation point are generated by a multiplicity-assignment block and stored in a 256×24 FERAM (front-end RAM) as shown. The overall re-encoding coordinate transformation algorithm consists of 4 sub-function blocks and they are implemented as different hardware processors and a central controller. An optimal scheduling scheme that enables maximum parallel processing among different hardware processors is illustrated in Figure 6.3. In Section 6.2, we describe a novel classification algorithm. Section 6.3 shows how the evaluation of $V(X)$ at the 16 interpolation X coordinates is implemented. The erasure-only decoding algorithm and its implementation are presented in Section 6.4. In Section 6.5, we describe how the coordinate shift and transformation are implemented. An overall hardware complexity and latency estimate, including synthesis results, are given in Section 6.6. Finally, conclusion is drawn in Section 6.7.

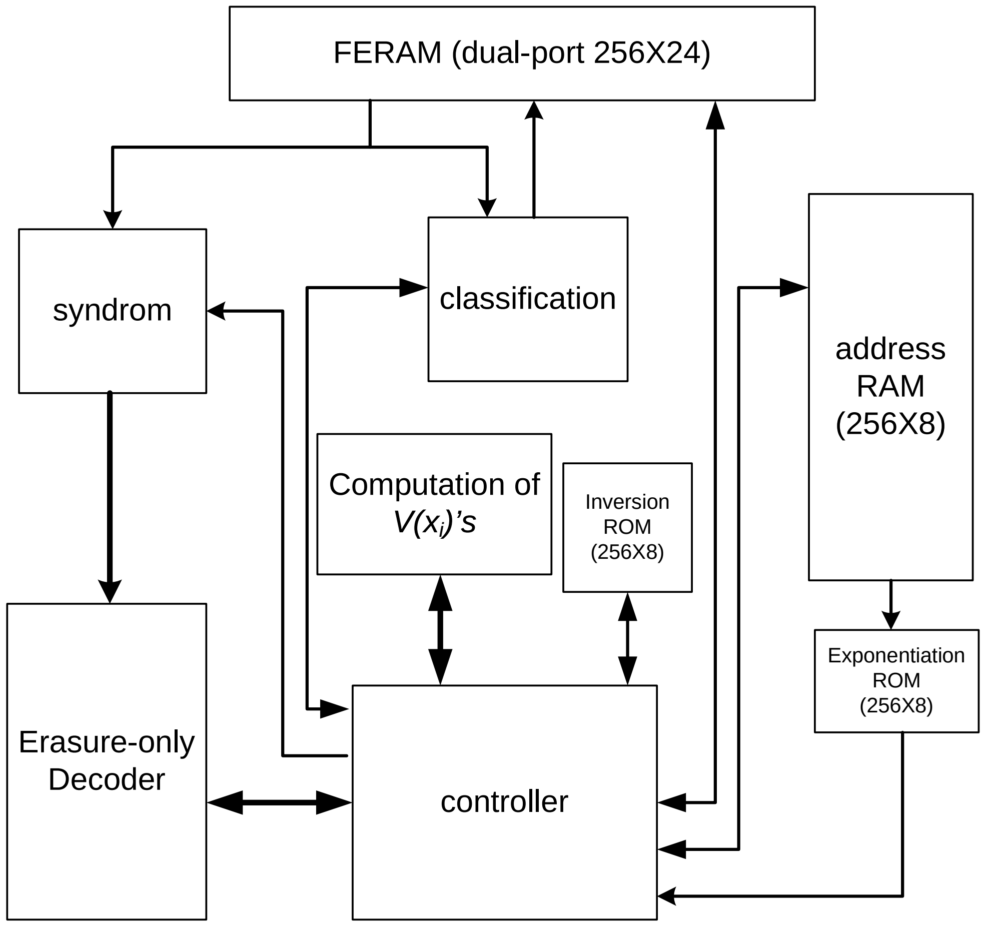

Figure 6.2: Block diagram of the implementation of re-encoding coordinate transformation algorithms

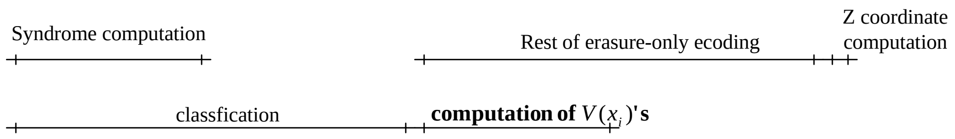

Figure 6.3: Timing Diagram for the Reencoding Process

6.2 The Classification Algorithm and Implementation

The classification block determines the 239 re-encoding indices corresponding to points of largest multiplicities, as well as the 16 interpolation indices. To avoid prohibitively complex sorting of the 255 indices, $\{i : i = 0, ..., 254\}$, according to their associated multiplicities, i.e., the $m_{i,0}$'s, we apply the following algorithm. Without loss of generality, we assume the maximum multiplicity is m_{max}.

The Classification Algorithm

- **Initialization:**
  $c[i] = 0,$ for $0 \leq i \leq m_{max}$, $s = 0$ and $t = 0$.

- **Iteration:**
  Input: $\{(x_i, y_{i,0}, m_{i,0}) : i = 0, 1, ..., 254\}$

  – First loop

```
for $i = 0$ to 254
    $c[m_{i,0}] := c[m_{i,0}] + 1$;
end
```

  – Decide the boundary multiplicity m_b

```
for $i = m_{max}$ to 0
    $s := s + c[i]$;
    if $s \geq 239$ break;
end
$m_b = i$;
```

  – Second loop

```
for $i = 0$ to 254
    if $m_{i,0} > m_b$, assign $i$ to $\{\mathcal{R}\}_X$;
    else if $m_{i,0} = m_b$, and $t < 239$, assign $i$ to $\{\mathcal{R}\}_X$;
    else assign $i$ to $\{\mathcal{A}\}_X$;
    $t := t + t$;
end
```

- **Result:** $\{\mathcal{R}\}_X$ and $\{\mathcal{A}\}_X$ with $|\{\mathcal{R}\}_X| = 239$ and $|\{\mathcal{A}\}_X| = 16$.

When the classification engine stores the last 16 indices into the address RAM, it also converts the indices into X coordinates and stores the 16 X coordinates into 16 8-bit registers for future use. The conversion is defined as an antilogarithm function in $\mathbb{F}_{2^8}$:

$$x_i = \alpha^i \tag{6.3}$$

This can be implemented as a 256×8 ROM-based look up table (LUT).

6.3 Computation of the Birational Mapping

As can be seen from (6.2), we need to compute the polynomial $V(X)$ and evaluate it at each of the $\{x_{i_l}:\ \text{for } l = 0, ..., 15,\ \text{and } i_l \in \{\mathcal{A}\}_X\}$ to carry out the coordinate transformation. This can be done after we finish the classification process and the 16 interpolation X coordinates are stored in a 16-element 8-bit register array. We compute

$$v_l = \prod_{m=0, j_m \in \{\mathcal{R}\}_X}^{238} (x_{j_m} - x_{i_l}),\ \text{for each of the 16 } i_l \in \{\mathcal{A}\}_X$$

in parallel with 16 $\mathbb{F}_{2^8}$ multipliers as illustrated in Figure 6.4. This computation require 239 clock cycles. However, since the computation is done in parallel with the construction of $\Lambda(X)$ and $\Omega(X)$ in the erasure decoding function and the same antilogarithm circuit is shared between these two functions, 16 more clock cycles are required. After evaluation, the 16 v_l's given in the equation above are stored in another 16-element 8-bit register array.

6.4 Erasure Decoding Algorithm

Most erasure decoding algorithms consist of 3 steps, namely computing syndromes, solving key-equations and computing the values of erasure symbols.

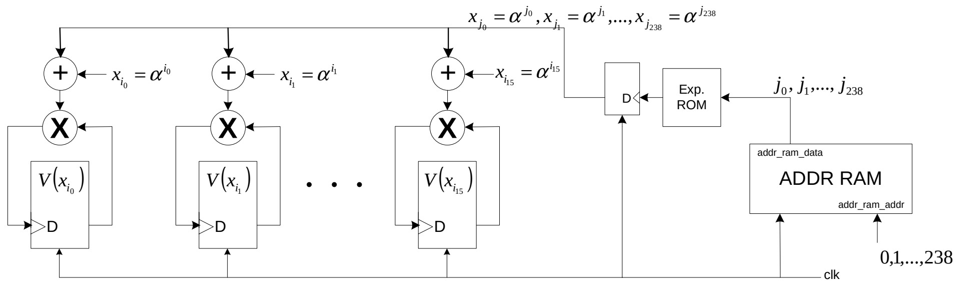

Figure 6.4: Architecture for evaluation of $V(X)$ at the interpolation X coordinates

Due to the nature of the algorithms applied to the 2nd step and 3rd step, we simply refer the 2nd step as the "construction" step and the 3rd step as the "evaluation" step.

The 1st step, syndrome computation, is as follows:

$$s_i = Y(\alpha^i) \tag{6.4}$$

where α^i's are roots of the generator polynomial and $Y(X) = y_{0,0} + y_{1,0}X + ... + y_{254,0}X^{254}$ represents the received hard-decision word. This procedure can be carried out in parallel with the classification step and reuse the multipliers an adders used to compute $V(x_i)$'s.

In the construction step, we treat the 16 indices $\{i \in \{\mathcal{A}\}_X\}$ as erasure locations. The iterative erasure decoding algorithm to be presented in the following is derived from the Berlekamp decoding algorithm in [Bla02]. Let us define syndrome polynomial $S(X)$, erasure locator polynomial $\Lambda(X)$ and erasure evaluator polynomial $\Omega(X)$ as follows

$$S(X) = \sum_{i=1}^{16} s_i X^i \tag{6.5}$$

$$\Lambda(X) = \prod_{i \in \mathbf{I}} (1 - x_i X) \tag{6.6}$$

$$\Omega(X) = S(X)\Lambda(X) \mod X^{17} \tag{6.7}$$

Both polynomial $\Lambda(X)$ and polynomial $\Omega(X)$ can be constructed iteratively and stored in an array of 16 shift registers. (Although both polynomials have a degree of 16, there is no need to store their constant coefficients.) Let us define $\Lambda^{(r)}(X) = \prod_{j=1}^{r}(1 - x_{i_j}X)$, where $i_j \in \{\mathcal{A}\}_X$, and define $\Omega^{(r)}(X) = S(X)\Lambda^{(r)}(X) \mod X^{r+1}$, both for $r = 1, 2, ..., 16$. In addition, we introduce a new variable $\Delta^{(r-1)}$, which is defined as follows

$$\Delta^{(r-1)} = \text{coef}\{\Lambda^{(r-1)}(X)S(X), X^r\} = \sum_{j=1}^{r} s_j \lambda_{r-j}^{(r-1)} \tag{6.8}$$

The iterative algorithm is given below.

The Iterative Algorithm

- **Initialization:** $\Omega^{(0)}(X) = 0$ and $\Lambda^{(0)}(X) = 1$.
- **Iteration:**

```
For $r = 1$ to 16, compute the following:
    $\Delta^{(r-1)} = \sum_{j=1}^{r} s_j \lambda_{r-j}^{(r-1)}$
    $\Lambda^{(r)}(X) = (1 - x_{i_r}X)\Lambda^{(r-1)}(X)$
    $\Omega^{(r)}(X) = (1 - x_{i_r}X)\Omega^{(r-1)}(X) + \Delta^{(r-1)}X^r$
```

- **Output:** $\Lambda(X) = \Lambda^{(16)}(X), \Omega(X) = \Omega^{(16)}(X)$

As can be seen from above, both $\Lambda(X)$ and $\Omega(X)$ can be constructed after 16 iterations. By definition, the coefficients of $\Lambda^{(r)}(X)$ can be computed from those of $\Lambda^{(r-1)}(X)$ as follows:

$$\lambda_i^{(r)} = \begin{cases} \lambda_{i-1}^{(r-1)} x_{i_r} & \text{for } i = r \\ \lambda_{i-1}^{(r-1)} x_{i_r} + \lambda_i^{(r-1)} & \text{for } i = 2, ..., r-1 \\ x_{i_r} + \lambda_i^{(r-1)} & \text{for } i = 1 \end{cases} \tag{6.9}$$

Thus a total of $r - 1$ multiplications and $r - 1$ additions are required to compute all r coefficients of $\Lambda^{(r)}(X)$ during the rth iteration. If we have a total of 15 multipliers and 15 adders to accommodate for the computation of $\Lambda^{(16)}(X)$ during iteration 16, we can finish all iterations in 16 clock cycles. However, this

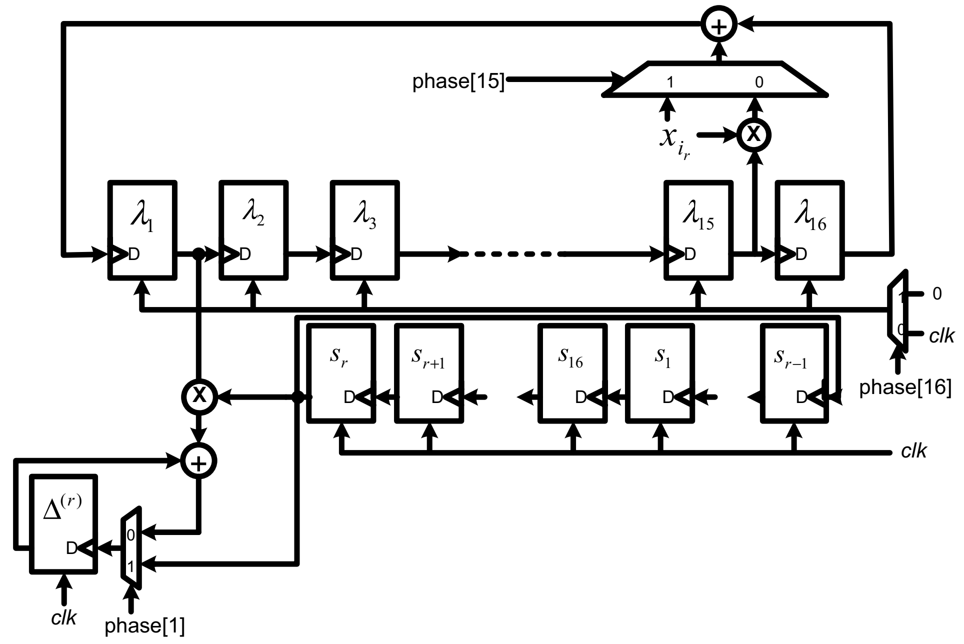

Figure 6.5: Architecture for construction of $\Lambda(X)$ and Δ's

iterative procedure is carried out in parallel with evaluation of $V(X)$ at the 16 interpolation X coordinates, which takes at least 239 clock cycles. This means that the 15 multipliers and 15 adders are idle most of the time. Since all $\lambda_i^{(0)}$'s are initialized to 0, we can make slight modification on (6.9) to obtain the following update formula for coefficients of $\Lambda^{(r)}(X)$:

$$\lambda_i^{(r)} = \begin{cases} \lambda_{i-1}^{(r-1)} x_{i_r} + \lambda_i^{(r-1)} & \text{for } i = 2, ..., 16 \\ x_{i_r} + \lambda_i^{(r-1)} & \text{for } i = 1 \end{cases} \tag{6.10}$$

This modification leads to a very regular and area-efficient architecture, where only 1 multiplier and 1 adder are used. Similarly for the computation of the coefficients of $\Omega^{(r)}(X)$, we can do the following:

$$\omega_i^{(r)} = \begin{cases} \omega_{i-1}^{(r-1)} x_{i_r} + \omega_i^{(r-1)} & \text{for } i = 2, ..., 16 \\ x_{i_r} + \omega_i^{(r-1)} & \text{for } i = 1 \end{cases} \tag{6.11}$$

$$\omega_r^{(r)} = \omega_r^{(r)} + \Delta^{(r-1)} \tag{6.12}$$

The equations above suggest that at iteration r, computation of $\Omega^{(r)}(X)$'s coefficients consists of 2 parts. The 1st part is exactly the same as the $\Lambda(X)$ update. In the 2nd part, we add a correction factor to $\omega_r^{(r)}$. Equation (6.8) can be modified as follows:

$$\Delta^{(r-1)} = \sum_{j=1}^{16} s_j \lambda_{(r-j)_{16}}^{(r-1)} = s_r + \sum_{j=1}^{15} s_{(r+j)_{16}} \lambda_{16-j}^{(r-1)}, \tag{6.13}$$

where $(x)_{16} \overset{\text{def}}{=} x \mod 16$. The equations (6.10) (6.11) (6.12) (6.13) enable us to compute $\Lambda(X)$ and $\Omega(X)$ with the architecture shown in Figure 6.5 and Figure 6.6.

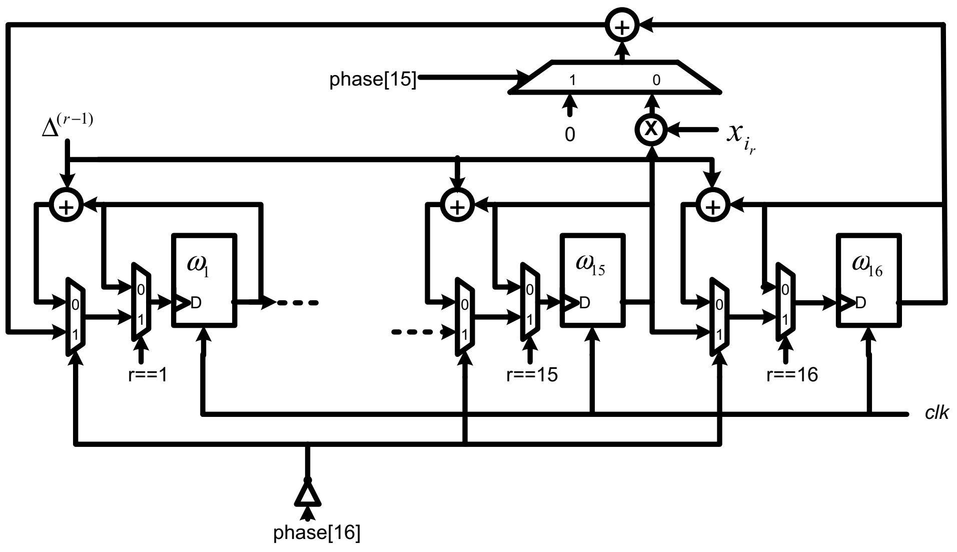

Figure 6.6: Architecture for construction of $\Omega(X)$

In the evaluation step, the Forney's algorithm [Bla02] is applied to generate erased symbols as follows:

$$c_i' = \frac{x_i \Omega(x_i^{-1})}{\Lambda'(x_i^{-1})} + y_{i,0} \text{ for } i \in \{\mathcal{A}\}_X \tag{6.14}$$

Since $\Lambda(X)$ can be expressed as $\Lambda(X) = \lambda_0 + \lambda_1 X + \lambda_2 X^2 + ... + \lambda_{16} X^{16}$, we have $\Lambda'(X) = \lambda_1 + \lambda_3 X^2 + ... + \lambda_{15} X^{14}$. Thus (6.14) can be expressed as

$$c_{i_r}' = \frac{\Omega(x_{i_r}^{-1})}{x_{i_r}^{-1} \Lambda'(x_{i_r}^{-1})} + y_{i_r,0} = \frac{\sum_{j=1}^{16} \omega_j x_{i_r}^{-j}}{\sum_{j=1}^{15} \lambda_j x_{i_r}^{-j}} + y_{i_r,0} \text{ for } i_r \in \{\mathcal{A}\}_X \tag{6.15}$$

The circuit computing $\Omega(x_{i_r}^{-1})$ and $x_{i_r}^{-1}\Lambda'(x_{i_r}^{-1})$ are illustrated in Figure 6.7 and Figure 6.8. To simplify the notation, we define $N_r \overset{\text{def}}{=} \Omega(x_{i_r}^{-1})$ and $D_r \overset{\text{def}}{=} x_{i_r}^{-1}\Lambda'(x_{i_r}^{-1})$. Horner's rule is applied for evaluating these 2 polynomials. In addition, this architecture enables us to reuse the multipliers and adders used for the construction of these 2 polynomials. Evaluating the 2 polynomials at the inverse of each of the interpolation X coordinates requires 16 clock cycles and due to the fact that $\deg X\Lambda'(X) = \deg \Omega(X) - 1$ and the pre-shift of λ_j's at the end of "construction" procedure, the computation of $x_{i_r}^{-1}\Lambda'(x_{i_r}^{-1})$ is finished 1 clock cycle ahead of the computation of $\Omega(x_{i_r}^{-1})$. This ensures that the reciprocal of $x_{i_r}^{-1}\Lambda'(x_{i_r}^{-1})$, which is obtained by a 256X8 ROM based LUT, and $\Omega(x_{i_r}^{-1})$ are available at the same time. They can then be multiplied together and added to $y_{i_r,0}$ to obtain the codeword symbol c'_{i_r} at the interpolation position i_r of the re-encoding codeword.

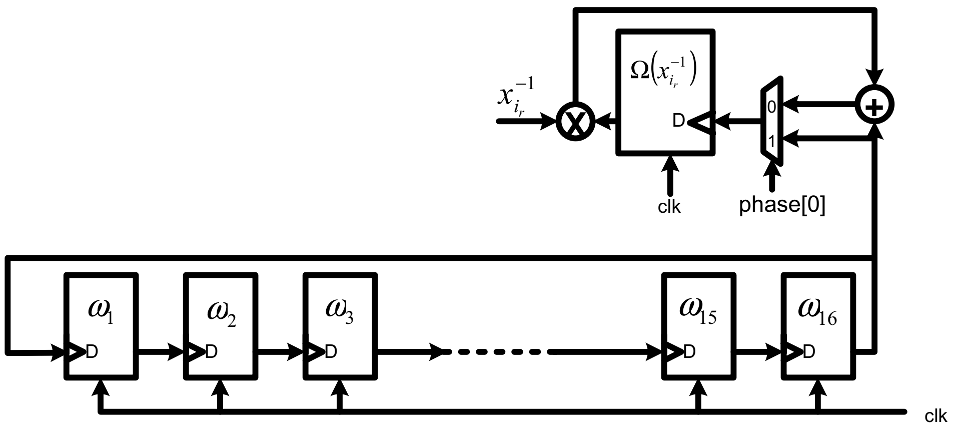

Figure 6.7: Architecture for computation of $\Omega(x_{i_r}^{-1})$

6.5 Coordinate Shift and Transformation

Once symbol c'_{i_r} is available, the "shift" operation is carried out by adding symbol c'_{i_r} to Y coordinates $y_{i_r,1}$ and $y_{i_r,0}$, which is pre-fetched from the FERAM. The 2 shifted Y coordinates $y'_{i_r,1}$ and $y'_{i_r,0}$ are then multiplied with the inverse of the corresponding $V(x_{i_r})$ to get the 2 Z coordinates. Note that this computation

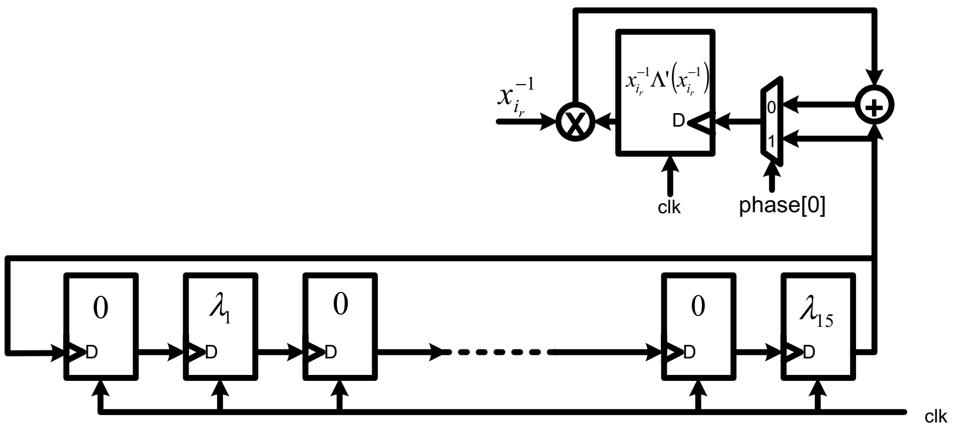

Figure 6.8: Architecture for computation of $x_{i_r}^{-1}\Lambda'(x_{i_r}^{-1})$

is pipelined with the circuits shown in Figure 6.7 and Figure 6.8. In addition, the multiplier with label 2 is a reuse of the multiplier used in computation of $V(x_{i_{15}})$ shown in Figure 6.4. The inversion of the 16 $V(x_{i_r})$'s is done using the circuit shown in the upper left part of Figure 6.9. It should be emphasized that though it seems, from Figure 6.9, that 2 inversion LUTs are required, actually only 1 inversion LUT is used and is time-shared among multiple computations. After the coordinate “shift” and transformation, the words in the interpolation addresses of FERAM are modified such that the original Y coordinates are replaced with the re-encoded and coordinated transformed Z coordinates.

6.6 Overall Hardware Complexity and Latency Estimate

A macro-level estimate of the hardware units required to implement the re-encoding coordinate transformation is given in Table 6.6. In the 2nd row of the table, the estimate is given for general (n, k) RS codes defined on $\mathbb{F}_{2^8}$. Plugging in $n = 255$ and $k = 239$ for this example, we can obtain the estimate shown in the 3rd row of the table. Note that the required hardware for the classification engine and the controller is not included. We have synthesized the entire design, including the 256×24 FERAM, with SMIC $0.18\mu m$ library. The overall number

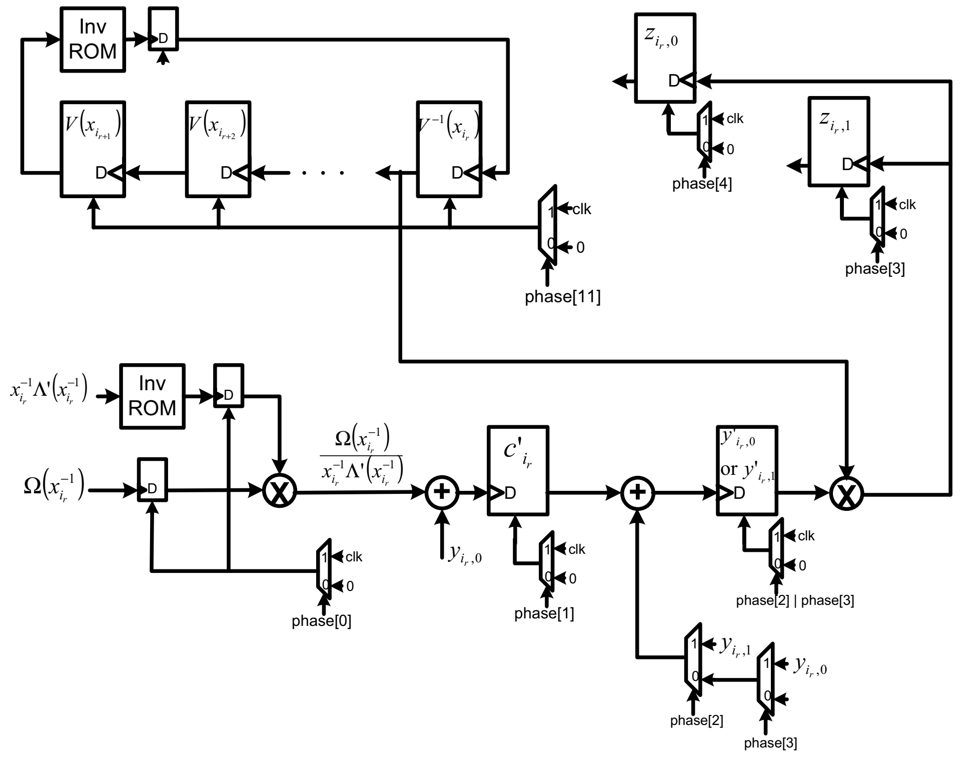

Figure 6.9: Architecture for computation of the Z coordinates for the interpolation points

of cells used is 9812, and the total area is 0.51 mm^2.

The synthesis tool is run with a clock frequency constraint of 250MHz. As shown in Section 6.2, it takes about 510 clock cycles for the classification engine to finish its processing. The erasure decoding step takes $272 + 256 = 528$ clock cycles. The computation of the 16 syndromes are done in parallel with the classification process and the computation of the $V(x_i)$'s are carried in parallel with the construction of $\Lambda(X)$ and $\Omega(X)$ as shown in Figure 6.3, thus they don't contribute any more delay to the re-encoding process. In addition, the coordinate "shift" and transformation are pipelined with the erasure decoding, thus they do not add any more delay to the whole process either. In summary, the total number of clock cycles required to finish the re-encoding coordinate transformation is 1038, which translate to a throughput equal to $\frac{255\times8\times250}{1038} \approx 500Mbps$. This throughput can satisfy requirement of any existing or developing interpo-

Table 6.1: Macro-Level Hardware Estimate for the Datapath

$\mathbb{F}_{2^8}$ multiplier	$\mathbb{F}_{2^8}$ adder	256×8 ROM	256×8 RAM	8-bit register
$(n-k)$ $+5$	$2(n-k)$ $+7$	2	1	$4(n-k)$ $+11$
21	39	2	1	75

lation processors to our best knowledge.

6.7 Conclusion

An efficient implementation of the re-encoding coordinate transformation process for the soft-decision decoding of Reed-Solomon codes has been presented for the first time. The proposed architecture, though illustrated for a practical example, can be easily extended to other Reed-Solomon codes. Highlights of our design include:

- An novel fast "classification" algorithm to determine the reencoding points.
- An area-efficient erasure-only RS decoder architecture.
- Parallel processing of various tasks to reduce the overall re-encoding latency.

Future work will be focused on scalability of the design so that different soft-decision decoding throughput requirement can be met with minimum area.

The material of Chapter 6 has been presented, in part, at *2006 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng. The dissertation author was the primary investigator and author of this paper.

CHAPTER 7

Fast Interpolation Architecture

Algebraic soft-decision decoding of Reed-Solomon (RS) codes delivers promising coding gains over conventional hard-decision decoding. The most computationally demanding step in soft-decision decoding of RS codes is bivariate polynomial interpolation. In this chapter, we present a hybrid data format based interpolation architecture that is well suited for high-speed implementation of the soft-decision decoders. It will be shown that this architecture is highly scalable and can be extensively pipelined. It also enables maximum overlap in time for computations at adjacent iterations. It is estimated that *the proposed architecture can achieve significantly higher throughput than conventional designs with equivalent or lower hardware complexity*.

7.1 Introduction

Reed-Solomon (RS) codes are the most widely used error-correcting codes in digital communications and data storage. Recently Sudan [Sud97] and Guruswami and Sudan [GS99] discovered hard-decision list decoding algorithms, which have larger decoding radius than conventional hard-decision decoding algorithms, such as the Berlekamp-Massey algorithm. The hard-decision list-decoding algorithms were later extended by Koetter and Vardy [KV03a] to an algebraic soft-decision decoding algorithm. All of these algorithms involve interpolation and factorization of bivariate polynomials. The interpolation step has been shown to be the most computation-intensive process for all list decoders. Thus it is crucial to develop efficient hardware architectures to implement the interpolation procedure, which will enable practical applications of the list decoding algorithms. However, among existing literatures, only a few [AKS03a] [AKS03b] [GKKG05] are devoted to study VLSI architectures to implement the interpolation process. In this chapter, we discuss a novel architecture which has significantly lower complexity than the architectures proposed in [AKS04b] and can achieve significantly higher processing speed than all known designs for the interpolation process. We will demonstrate that the architecture can be extensively pipelined. Combined with the proposed timing scheme, the average iteration time for the interpolation process can be maximally reduced, which makes it well suited for high speed applications. In addition, the new architecture is inherently scalable. Thus it can be applied to various applications with different speed requirements. Part of this work is also presented in [MVW06a].

This chapter is organized as follows: Section 7.2 gives some background information. The interpolation algorithm and the proposed interpolation architecture are discussed in Section 7.3. Section 7.4 presents a soft-decision decoder design example for a (255,239) Reed-Solomon code with the proposed interpolation architecture. Conclusions are drawn in Section 7.5.

7.2 Background and Preliminaries

As shown in Figure 1.1 of Chapter 1, the soft-decision Reed-Solomon decoder consists of three major blocks. They are usually referred to as the multiplicity assignment block, the interpolation block and the factorization block, respectively. It is well-known that the interpolation block has an order of magnitude higher computation complexity than other blocks and thus is the focus of this chapter. For a description of the multiplicity assignment and factorization algorithms, interested readers may refer to [PV03] [RR00]. Since the interpolation problem and Koetter's interpolation algorithm have been discussed in detail in Chapter 2, we will not repeat them here. All definitions and notation given in Chapter 2 apply to this chapter.

We now briefly discuss representation of numbers in a finite field. All practical RS codes are defined in finite fields with characteristic equal to 2. Hence, in the rest of the chapter, the discussion will be limited to finite field with 2^p elements denoted as $\mathbb{F}_{2^p}$. Most commonly, a number in $\mathbb{F}_{2^p}$ is represented as a binary polynomial of degree $(p-1)$ or a p-tuple binary vector. It is also possible to express a finite field number as a power of the primitive element in the field. With the former representation, implementation of addition operation in the field is straightforward as it only requires bitwise exclusive-or operation. However, implementation of multiplication operation in the field is more complicated as it usually involves large combinational circuit for bit-parallel implementation or significant number of clock cycles for bit-serial implementation. If the latter representation is used instead, multiplication operation can be implemented as addition with modulo operation. However, implementation of addition operation will be complicated. ¿From now on, we refer the 1st representation as *regular representation* and the 2nd representation as *power representation*. The power representation for $\mathbb{F}_{2^p}$ numbers is rarely used in existing Reed-Solomon codec implementations. In this chapter, we show that representing the $\mathbb{F}_{2^p}$ numbers in both regular and power formats, i.e. hybrid-format, can lead to an efficient decoder architecture. For the rest of the chapter, we use lower case letter for regular representation of a finite field number and the same lower case letter with a tilde symbol on top for power representation of the same finite field number. For example, if x and y are regular representations of finite field numbers, then $\tilde{x}$ and $\tilde{y}$ are corresponding power representations. The power representations and regular representations are related as follows: $\tilde{x} \overset{\text{def}}{=} \log_\alpha x$ and $\tilde{y} \overset{\text{def}}{=} \log_\alpha y$, or equivalently $x = \alpha^{\tilde{x}}$ and $y = \alpha^{\tilde{y}}$. In the formula above, $log_\alpha(.)$ represents the operation to convert a finite field number in regular representation to the power of the field's primitive element α, where $\alpha^{(\cdot)}$ denotes the operation to convert a finite field element represented as power of the primitive element back to its regular representation. We refer these 2 operations as *logarithm* and *antilogarithm*, respectively. To make the definition "complete", we use value $(2^p - 1)$ as power representation for finite field number 0.

7.3 Interpolation Architecture

The most computationally complex step in algebraic soft decision decoding of Reed-Solomon codes is bivariate polynomial interpolation. This step is carried out by applying Koetter's interpolation algorithm, which is presented at Chapter 2 and repeated as follows.

Koetter's Interpolation Algorithm

```
• Initialization:
  $\mathcal{Q}_v(X,Y) = Y^v$, for $0 \le v \le r$.

• Iteration:
  Input: $\{(x_i, y_i, m_{x_i,y_i}) : (x_i, y_i) \in \mathcal{P}\}$

    – For each triple $(x_i, y_i, m_{x_i,y_i})$,
      $O_v = \deg_w \mathcal{Q}_v(X,Y)$, for $0 \le v \le r$.

      for $b = 0$ to $m_{x_i,y_i} - 1$
          for $a = 0$ to $m_{x_i,y_i} - 1 - b$
              Discrepancy Coefficient Computation:
              for $v = 0$ to $r$
                  $d_v^{(a,b)} = \mathbf{coef}\big(\mathcal{Q}_v(X + x_i, Y + y_i), X^a Y^b\big)$
              end
              Polynomial Update:
              if there exist $\eta = \operatorname{argmin}_{0 \le v \le r,\ d_v^{(a,b)} \neq 0} \{O_v\}$
                  for $v = 0$ to $r$
                      if $v \neq \eta$ and $d_v^{(a,b)} \neq 0$
                          $\mathcal{Q}_v(X,Y) := d_\eta^{(a,b)} \mathcal{Q}_v(X,Y) +$
```

```
                        $d_v^{(a,b)} \mathcal{Q}_\eta(X,Y)$
                end if
            end
            $\mathcal{Q}_\eta(X,Y) := \mathcal{Q}_\eta(X,Y)(X - x_i)$, and
            $O_\eta := O_\eta + 1$
        end if
    end
end
```

- **Result:**
  $\mathcal{Q}(X,Y) = \{\mathcal{Q}_\eta(X,Y)\}$, where $\eta = \text{argmin}_{0 \le v \le r}\{O_v\}$.
  So that

$$\mathcal{Q}(X,Y) = \sum_{t=0}^{r} q_t(X) Y^t$$

The algorithm presented above can solve the interpolation problem given in Section 7.2 iteratively, one constraint at a time, thus it is referred to as constraint-serial interpolation algorithm. Another version of the iterative algorithm called *point-serial* interpolation algorithm is introduced in [AKS04b]. This algorithm is more efficient only when all constraints associated with an interpolation point need to be enforced in the interpolation procedure. In addition, the point-serial interpolation algorithm favors certain architectures as shown in [AKS04b], thus it is less flexible than the constraint-serial interpolation algorithm. In this chapter, we will focus on the constraint-serial interpolation algorithm. The benefits will be evident from later discussion.

Recently, revolutionary algorithmic changes applying re-encoding and coordinate transformation techniques [GKKG02] [KMVA03] are proposed to reduce the original interpolation problem to a much smaller counterpart. The reduced interpolation problem can also be solved with a slight variant of the iterative interpolation algorithm presented above. In this section, we start with discussing interpolation architecture to solve the original interpolation problem. But it can be shown that small modifications can be made to the architecture to solve the reduced interpolation problem. For a detailed description of the reduced interpolation problem and its solution, interested readers can refer to [KMVA03]. In the iterative algorithm given above, each iteration involves two major operations, namely the *discrepancy coefficient computation* (DCC) and *polynomial update* (PU). Subsection 7.3.1 will present a novel architecture for the DCC operation. Subsection 7.3.2 will discuss architectures for the PU operations. Between the DCC and PU operations at each iteration, one needs to determine which polynomial has the minimum weighted degree among all polynomials with non-zero discrepancy coefficient. This necessary step that connects the DCC and PU processes has been ignored by all previous works. In this chapter, an implementation of this step will be presented in Subsection 7.3.3. Following that, Subsection 7.3.4 will discuss how maximum concurrency can be achieved between DCC and PU processes. Finally, Subsection 7.3.5 will show that adjustments can be made to the interpolation architecture to accommodate re-encoded interpolation. For the discussion throughout the rest of the chapter, it is assumed that a total of $(r+1)$ polynomials are used for the interpolation algorithm.

7.3.1 Architecture for Discrepancy Coefficient Computation

Existing architecture for the DCC operation [AKS04b] applies Horner's rule to evaluate bivariate polynomials at a certain point (x, y). It has a MAC unit in the critical path. Due to the recursive nature of Horner's rule, it is meaningless to directly pipeline the MAC operation to reduce the critical path delay. Thus the critical path delay is always lower bounded by the delay associated with the MAC unit. For Reed-Solomon codes defined on a large finite field, the delay associated with a MAC unit is significant. A folded-pipelined architecture for DCC operation is also proposed in [AKS04b]. This folded-pipelined architecture, though fast and area-efficient for DCC, is bonded with the point-serial interpolation algorithm [AKS04b], and it requires that the multiplicity associated with each interpolation point is less than but close to the value of p.

The folded-pipelined architecture is efficient only when all $(m+1)m/2$ constraints associated with one interpolation point with multiplicity m need to be enforced. Thus it can not be applied to other variants of the interpolation algorithm, which might require the interpolation constraints to be solved in a different order. For example, it was shown in [AKS04a] that carrying out interpolation with gradually-increasing number of constraints for each interpolation point and performing factorization on intermediate interpolation results improve decoder performance with lower overall interpolation complexity. The point-serial interpolation algorithm and the folded-pipelined architecture can not be applied in this case. In addition, when the DCC and PU operations are carried out concurrently as proposed in Section 7.3.4, the folded-pipelined architecture can not reduce the overall latency of the interpolation process, which is lower bounded with latency of the PU process. Thus it is desirable to have a flexible interpolation architecture that is high-speed and not tied to any specific order of the interpolation constraint enforcement. In this chapter, we present a novel architecture to implement the DCC operation, which applies hybrid representation of the finite field numbers used in the computation. This new approach effectively breaks the high-speed bottleneck formed by the MAC-unit with the Horner's rule based recursive method. A detailed description of the architecture will be presented in the following.

The equation used for discrepancy coefficients computation can be expressed as follows:

$$\begin{aligned} d_v^{(a,b)} &= \mathbf{coef}\big(\mathcal{Q}_v(X+x, Y+y), X^aY^b\big) \\ &= \textstyle\sum_{t=b}^{r}\sum_{s=a}^{w_{v,t}} \binom{s}{a}\binom{t}{b} q_{s,t}^{(v)} x^{s-a}y^{t-b} \text{ for } v = 0, 1, ..., r. \end{aligned} \tag{7.1}$$

Instead of applying Horner's rule, we can evaluate each monomial, i.e., $q_{s,t}^{(v)}X^{s-a}Y^{t-b}$, for $s \geq a$ and $t \geq b$, independently at point (x, y) and then sum them up to obtain the final coefficient $d_v^{(a,b)}$. Please note that evaluation of monomial $x^{s-a}y^{t-b}$ is only valid for $s \geq a$ and $t \geq b$ as defined in (7.1), so in the rest of the section, we implicitly assume that $s \geq a$ and $t \geq b$ whenever we use notation $(s-a)$ or $(t-b)$. A direct realization of $x^{s-a}y^{t-b}$ is not easy, especially when the value of $(s-a)$ gets very large. This is because it requires $(s-a)$ consecutive multiplication of x in the finite field. One way around it is to convert x and y from their regular representation to power representation. Let us also define $\{i\}_m \overset{\text{def}}{=} i \mod m$, where i and m are non-negative and positive integers, respectively. This leads to the following evaluation formula.

$$d_v^{(a,b)} = \sum_{t=b}^{r} \sum_{s=a}^{w_{v,t}} \binom{s}{a} \binom{t}{b} q_{s,t}^{(v)} \alpha^{\{(s-a)\tilde{x}+(t-b)\tilde{y}\}_{(2^p-1)}} \quad \text{for } v = 0, 1, ..., r, \tag{7.2}$$

where α is the primitive element of the Galois field, and $\tilde{x} = \log_\alpha x$ and $\tilde{y} = \log_\alpha y$ are as defined at the end of Section 7.2. As it is declared at the beginning of the chapter, we assume the following condition is true: $x \neq 0$. However, the Y coordinate y could be zero. Let us define $[s-a] \overset{\text{def}}{=} \max(s-a, 0)$ and $[t-b] \overset{\text{def}}{=} \max(t-b, 0)$. We also define the following boolean function $c(a, b, s, t, \tilde{y})$ as

$$c(a,b,s,t,\tilde{y}) = \begin{cases} \left\{\binom{s}{a}\binom{t}{b}\right\}_2 & \text{if } \tilde{y} \neq (2^p-1) \text{ or } \tilde{y} = (2^p-1) \text{ and } t = b, \\ 0 & \text{otherwise.} \end{cases} \tag{7.3}$$

Thus we can slightly modify (7.2) as follows:

$$d_v^{(a,b)} = \sum_{t=0}^{r} \sum_{s=0}^{w_{v,t}} c(a,b,s,t,\tilde{y}) q_{s,t}^{(v)} \alpha^{\{[s-a]\tilde{x}+[t-b]\tilde{y}\}_{2^p-1}} \quad \text{for } v = 0, 1, ..., r. \tag{7.4}$$

In (7.4), the polynomial coefficients are still represented with their regular form because this eases the implementation of polynomial update process. With this formula, computation of $x^{s-a}y^{t-b}$ only takes 2 integer multiplications, 1 modulo addition and 1 conversion operation to convert $x^{s-a}y^{t-b}$, from its power representation, i.e., $\{\tilde{x}(s-a) + \tilde{y}(t-b)\}_{(2^p-1)}$ back to regular representation.

Though the X and Y coordinates of the interpolation points have to be converted from regular representation to power representation and convert the intermediate result, $\{\tilde{x}(s-a)+\tilde{y}(t-b)\}_{(2^p-1)}$ from power representation back to normal representation, evaluating all $x^{s-a}y^{t-b}$'s independently in logarithm domain eliminates the need for Horner's rule based recursive algorithm, which makes it easier to pipeline the circuit. Depending on the speed requirement, multi-stage pipeline can be introduced to integer multiplication, integer addition and finite field multiplication operations. In fact, it will be shown later in the chapter that only p-bit adders and fixed-coefficient p-bit integer multiplier, which has significantly smaller area than regular p-bit integer multiplier, is required to implement (7.4). In the rest of this subsection, an efficient way to implement $\{\tilde{x}[s-a]+\tilde{y}[t-b]\}_{(2^p-1)}$ is discussed first, then the implementation of the antilogarithm operation mentioned above is discussed and finally the architecture for the overall DCC operation is presented. It is assumed that the operation of converting the X and Y coordinates of the interpolation points from regular representation to power representation is combined into the multiplicity assignment block shown in Figure 1.1 of Chapter 1, thus its discussion is omitted in this chapter.

Implementation of $\left(\tilde{x}[s-a]+\tilde{y}[t-b]\right)$ modulo (2^p-1) operation

A straightforward implementation would use two regular integer multipliers and one adder followed by a circuit that implements modulo (2^p-1). However, one may observe, from (7.4), that s is a running index and that, for practical applications of the soft-decision algebraic decoder, the value of r is usually a small number. Thus the circuit shown in Figure 7.1 can be used to implement it.

In the figure above, the integer adders with a square box around it represents p-bit modulo (2^p-1) integer adder. However, these two adders are of different types, thus different subscripts are assigned to them in the figure. A detailed description of the implementation of the these two types of adders will be given later. The $(r-1)$ multipliers in the figure are p-bit modulo (2^p-1) integer multipliers. Note that these $(r-1)$ multipliers have a fixed multiplicand

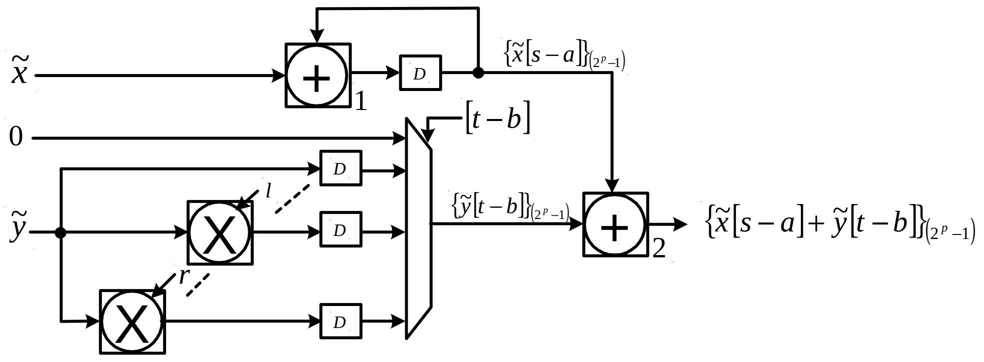
$\tilde{x}$
$\{\tilde{x}[s-a]\}_{(2^p-1)}$
D
1
0
$[t-b]$
D
l
$\{\tilde{y}[t-b]\}_{(2^p-1)}$
$\tilde{y}$
X
D
2
$\{\tilde{x}[s-a]+\tilde{y}[t-b]\}_{(2^p-1)}$
r
X
D

Figure 7.1: Implementation of $\left(\tilde{x}[s-a]+\tilde{y}[t-b]\right)$ modulo (2^p-1) operation.

as input, and the fixed multiplicand is usually a smaller number. Therefore these multipliers normally cost significantly less hardware than a regular p-bit multiplier. Instead of using a regular multiplier, here we choose to use $(r-1)$ such “small” multipliers with a MUX to implement $\{\tilde{y}[t-b]\}_{(2^p-1)}$ because it will be shown in Subsection 7.3.1 that these “small” multipliers can be shared for simultaneously computing the discrepancy coefficients for all $(r+1)$ polynomials. In addition, since r is a fixed number, all $\{\tilde{y}l\}$ for $l=2,3,...,r$ can be pre-computed and stored in the registers shown in Figure 7.1. In the following, we describe the implementation of the 3 types of devices embedded with modulo (2^p-1) operation.

We start with describing a circuit that implements $w=\{f+g\}_{(2^p-1)}$, where f and g are both non-negative integers and satisfy $f,g<2^p$. This guarantees that $\{f+g\}_{2^p}+1\leq 2^p-1$, or equivalently, $\{f+g+1\}_{2^p}\leq 2^p-1$. In other words, there is no carry out signal for this adding 1 operation. Thus we have

$$w=\begin{cases}\{f+g+1\}_{2^p} & \text{if } f+g>(2^p-1),\\ f+g & \text{if } f+g<(2^p-1),\\ 0 & \text{otherwise.}\end{cases}$$

The equation above can be mapped to the block diagram shown in Figure 7.3.1. Note that the portion represented by dotted lines ensures that the output is zero when $(g+f)=(2^p-1)$. However, this may not be necessary as it will be

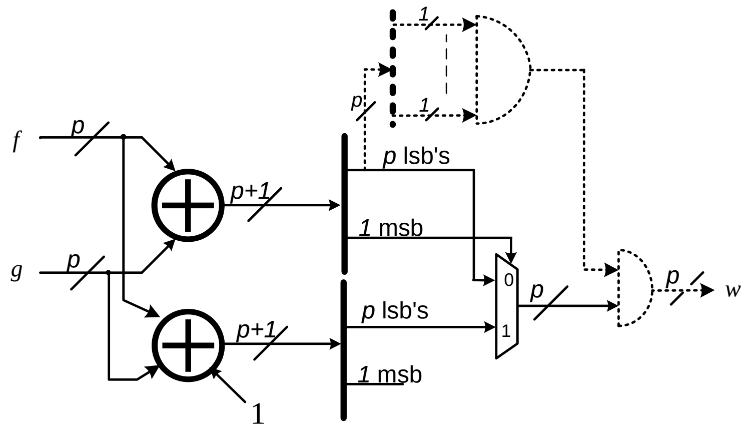

Figure 7.2: Implementation of modulo $(2^p - 1)$ addition.

shown later that an output equal to $(2^p - 1)$ can be appropriately handled by the antilogarithm unit. Thus to reduce the overall delay, the dotted portion can be omitted. Now the delay of the circuit is equal to the sum of the delay of the adder and the delay of a 2:1 MUX. Fast adders, such as the ones based on carry-look-ahead architecture, should be used in the circuit. In addition, if the modulo $(2^p - 1)$ addition circuit is not placed within a loop, pipelining technique can be used to reduce the critical path delay.

Note that if both f and g are equal to $(2^p - 1)$, the output from the circuit shown in Figure 7.3.1 generates $(2^p - 1)$ at its output.

Now let us first describe the implementation of the modulo $(2^p - 1)$ multiplier that produces $\{\tilde{y}l\}_{(2^p-1)}$ for $l = 2, 3, ..., r$. As it is mentioned earlier, r is usually a small number for practical applications of the algebraic soft-decision decoding algorithm. Here we assume that the fixed multiplicand l satisfy the condition $l < 10$. In binary format, l can be represented as $l_3 l_2 l_1 l_0$. We can also assume the $p > 3$ as most practical Reed-Solomon codes are defined over large Galois field. Thus we have

$$\begin{aligned} &\{\tilde{y}l\}_{(2^p-1)} \\ =&\left\{\{2^3 l_3 \tilde{y}\}_{(2^p-1)} + \{2^2 l_2 \tilde{y}\}_{(2^p-1)} + \{2 l_1 \tilde{y}\}_{(2^p-1)} + l_0 \tilde{y}\right\}_{(2^p-1)} \end{aligned} \tag{7.5}$$

and modulo $(2^p - 1)$ multiplication in form of $\{2^i\tilde{y}\}_{(2^p-1)}$ can simply be implemented by cyclically shifting p-bit number $\tilde{y}$ left by i bits. The cyclic shift operation can be justified as follows: Let $\tilde{y} = \tilde{y}_2 2^{p-i} + \tilde{y}_1$, i.e., $\tilde{y}_2$ is the number represented by the i msb's of $\tilde{y}$ while $\tilde{y}_1$ is the number represented by the $(p-i)$ lsb's of $\tilde{y}$. Note that $\{2^p\tilde{y}_2\}_{(2^p-1)} = \tilde{y}_2$, we then have $\{2^i\tilde{y}\}_{(2^p-1)} = \{\{2^p\tilde{y}_2\}_{(2^p-1)} + 2^i\tilde{y}_1\}_{(2^p-1)} = 2^i\tilde{y}_1 + \tilde{y}_2$.

Figure 7.3 shows an implementation example of $\tilde{y} * 7$ *mod* $(2^8 - 1)$, where $\tilde{y} = Y[7..0]$, S[7..0] and C[7..0] represent the sum and carry out components, respectively, for the carry-save operation (see top "+" sign in the figure). The modulo operation is easily handled by moving some most significant bits to least significant positions as shown in the figure. Since $l < 10$, we will, in any case, have no more than 3 partial products as we have in this example. This means (7.5) takes at most one full adder delay and the delay to compute two p-bit addition with modulo $(2^p - 1)$ operation, which can be implemented as shown in Figure 7.3.1. Note that if $\tilde{y} = (2^p - 1)$, all modulo $(2^p - 1)$ multipliers will generate $(2^p - 1)$ at its output. However, as it will become clear at the end of this subsection, this output will be "filtered" out by another modulo $(2^p - 1)$ addition implemented as the modulo $(2^p - 1)$ adder with subscript 2 in Figure 7.1.

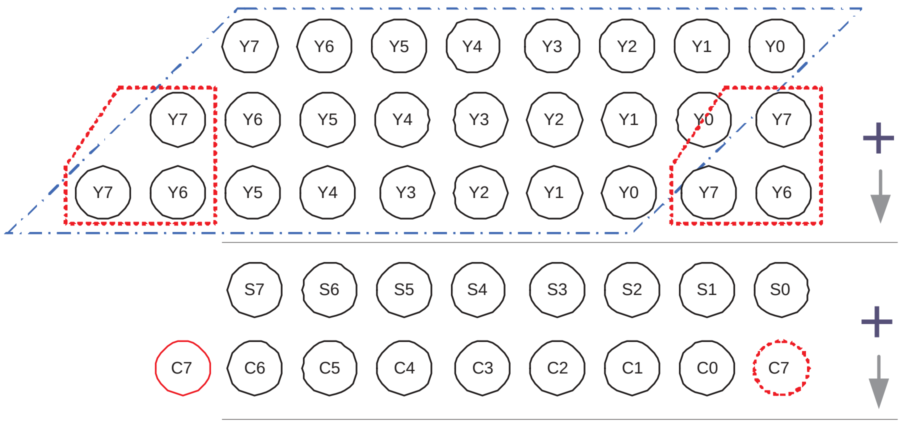

Figure 7.3: Implementation example of $\{\tilde{y} * 7\}_{(2^8-1)}$.

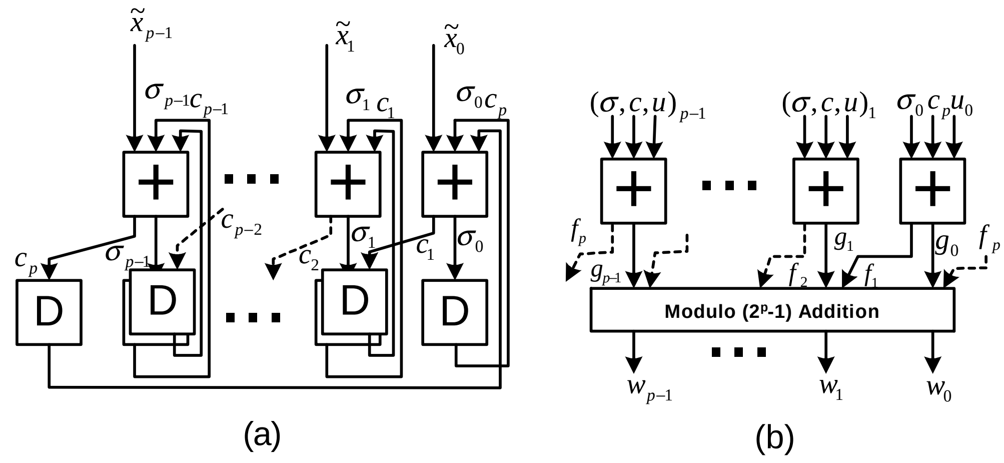

Figure 7.4: Implementation of (a) $\{\tilde{x}(s-a)\}_{(2^p-1)}$ and (b) $\{\tilde{x}(s-a)+\tilde{y}(t-b)\}_{(2^p-1)}$.

Next we discuss the implementation of the modulo (2^p-1) embedded accumulator that recursively generates $\{\tilde{x}[s-a]\}_{(2^p-1)}$, *i.e.*, the modulo adder with subscript 1 shown in Figure 7.1. Let us define $A_{\tilde{x}}(s) \overset{\text{def}}{=} \{\tilde{x}[s-a]\}_{(2^p-1)}$, then $A_{\tilde{x}}(s+1) = \{(A_{\tilde{x}}(s)+\tilde{x})\}_{(2^p-1)}$. So $A_{\tilde{x}}(s+1)$ can be computed with the modulo (2^p-1) addition circuit shown in Figure 7.3.1 with $f = A_{\tilde{x}}(s)$ and $g = \tilde{x}$. However, as it is mentioned earlier, the delay of the circuit shown in Figure 7.3.1 can be quite large, thus placing it in the accumulator loop will significantly limit the clock frequency of the entire design. This can be tackled by applying the "carry-and-save" method, which is described as follows. At each cycle, only 1-bit full addition is performed as illustrated in Figure 7.4 (a). The temporary sum $\sigma = \sigma_{p-1}...\sigma_1\sigma_0$, and carry out, $\{c = c_p...c_1\}$, are both stored in p-bit registers. Note that both σ and c are equal to (2^p-1) if and only if $\tilde{x} = (2^p-1)$. The accumulator output $A_{\tilde{x}}(s)$ can be expressed as $A_{\tilde{x}}(s) = \left\{\sigma + \{2c\}_{(2^p-1)}\right\}_{(2^p-1)}$ and since $\{2c\}_{(2^p-1)} = c_{p-1}...c_1c_p$, this leads to the implementation shown in Figure 7.4 (a), where bit c_p is pulled back as an input to the last full adder.

At last, the implementation of the modulo (2^p-1) adder with subscript 2 shown in Figure 7.1 is described. This adder takes 3 numbers as input which are the temporary sum σ, cyclicly left-shifted carry bits, i.e., $c_{p-1}...c_1c_p$, and

$\{\tilde{y}[t-b]\}_{(2^p-1)}$. The former two come from the modulo (2^p-1) adder with subscript 1 and the third one comes from the selected output from one of the modulo (2^p-1) multipliers described earlier. The implementation of this 3-input adder is illustrated in Figure 7.4 (b), where $u = u_{p-1}...u_1u_0$ represents the selected modulo (2^p-1) multiplier output. The carry-and-save method is used here as well to reduce the complexity and delay of the circuit. The modulo (2^p-1) addition circuit shown in Figure 7.3.1 is used to carry out modulo (2^p-1) addition of the two numbers f and g produced from the p 1-bit full adders. Note that the number f consisting of the carry bits $f_p...f_1$ is also cyclically shifted left by one bit to carry out an inherent modulo (2^p-1) operation as explained earlier in this subsection.

The total delay of the circuit presented in this subsection that computes $\left\{\tilde{x}[s-a]+\tilde{y}[t-b]\right\}_{(2^p-1)}$ for a particular pair of s and t can be estimated as follows. The delay associated with computing all $\{\tilde{y}l\}_{(2^p-1)}$'s can be ignored due to the fact that they can be pre-computed. The total delay is equal to the delay of the modulo (2^p-1) adder with subscript 2 and the delay associated with the modulo (2^p-1) adder with subscript 1 in Figure 7.1. Note that here we purposely ignored the delay associated with the (r+1):1 MUX, because it will be shown later in Subsection 7.3.1 that we can get around using any MUX to select the appropriate $\{\tilde{y}l\}_{(2^p-1)}$ to send to the modulo (2^p-1) adder with subscript 2. Thus the total delay is equal to the sum of the delays associated with circuits shown in (a) and (b) in Figure 7.4, which is equal to two times of the delay of a 1-bit full adder plus the delay of the circuit shown in Figure 7.3.1.

Implementation of antilogarithm

As shown in 7.2, the coefficients of the polynomials are represented in regular format, thus an antilogarithm operation is required. The most straightforward way to implement the antilogarithm operation in $\mathbb{F}_{2^p}$ is to use a 2^p-entry LUT. However, for large p, a ROM-based 2^p-entry table implies long critical-path delay, which is undesirable. Combinational logic based implementation with pipeline can reduce the critical-path delay, however, when p is a large number, a p-bit input p-bit output combinational logic also infers large gate count. In order to reduce the hardware consumption, non-negative integer i, where $i < 2^p$, can be expressed as $i = 2^{p'}i' + i''$, where $i'' < 2^{p'}$ and $p' = \lfloor \frac{p}{2} \rfloor$. We then have $\alpha^i = \alpha^{2^{p'}i'} \alpha^{i''}$. This enables us to use a $(p - p')$-bit input combinational logic to realize $\alpha^{2^{p'}i'}$ and a p'-bit input combinational logic to implement $\alpha^{i''}$. Then a finite field multiplier is needed to compute α^i as shown in Figure 7.5. Note that both the combinational logic blocks and the finite field multiplier can be pipelined to meet any critical-path delay requirement.

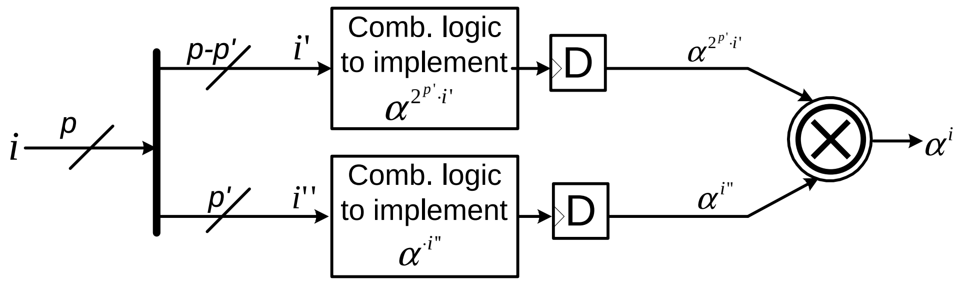
p-p'
i'
Comb. logic to implement $\alpha^{2^{p'}\cdot i'}$
D
$\alpha^{2^{p'}\cdot i'}$
i
p
p'
i''
Comb. logic to implement $\alpha^{\cdot i''}$
D
$\alpha^{i''}$
α^i

Figure 7.5: Implementation of the antilogarithm based on LUT and multiplier.

Architecture for DCC operation

Now we have described the circuits used to implement the $\{\tilde{x}[s - a] + \tilde{y}[t - b]\}_{(2^p-1)}$ and the antilogarithm operations, the architecture shown in Figure 7.6 can be used to evaluate the discrepancy coefficients for one bivariate polynomial. The coefficients of a bivariate polynomial $Q_v(X, Y) = \sum_{t=0}^{r} b_t^v(X)Y^j = \sum_{t=0}^{r}\sum_{s=0}^{w_{v,t}} q_{s,t}^{(v)} X^s Y^t$ can be stored in $(r + 1)$ banks of RAM. In the figure, the multiplier and adder with double circle represent finite-field multiplier and adder. An efficient implementation of $\{\binom{s}{a}\binom{t}{b}\}_2$ can be found in [GKKG05], thus the computation of $c(s, b, s, t, \tilde{x}, \tilde{y})$ can be implemented with combinational logic. Apparently the circuit can be extensively pipelined to ensure that the critical-path delay is less than a certain desired amount. It can be assumed that the hybrid multiplication, which includes computing $\tilde{x}[s - a] + \tilde{y}[t - b]$ modulo $(2^p - 1)$ as described in Subsection 7.3.1 and LUT-based antilogarithm described in Subsection 7.3.1, uses ξ_0-level pipeline. We further assume the rest of the datapath, which includes the $\mathbb{F}_{2^p}$ multipliers, the $\mathbb{F}_{2^p}$ accumulator and the $(r+1)$-input $\mathbb{F}_{2^p}$ adder, uses ξ_1 pipelining stages. Note that the delay associated with the combinational logic that computes $c(s, b, s, t, \tilde{x}, \tilde{y})$ is not counted because the computation can be carried out in parallel with the antilogarithm operation. If the X-degree of the polynomial is d_X, the total number of clock cycles required to finish 1 DCC operation is $\xi_0 + \xi_1 + d_X$.

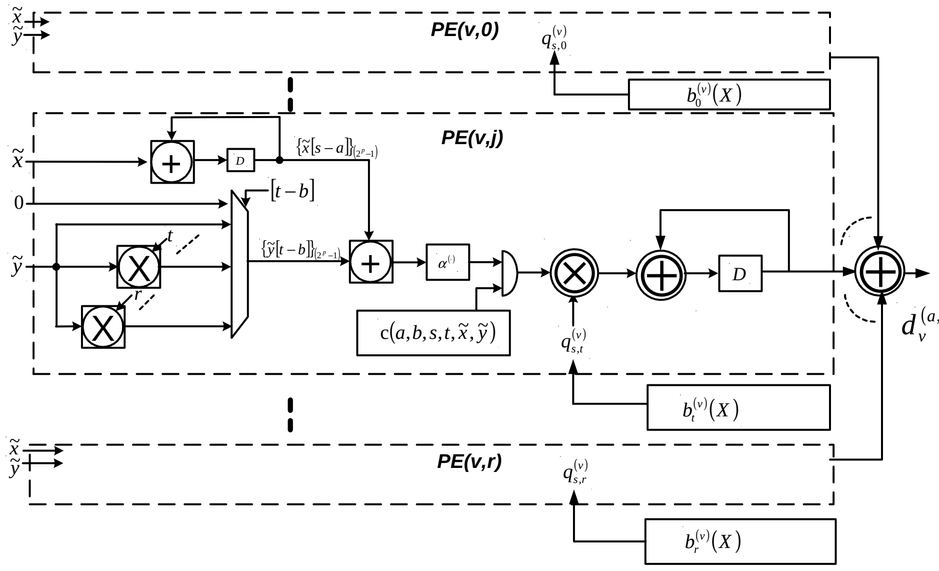

Figure 7.6: Architecture of the DCC for $Q_v(X, Y)$.

Let us assume that there are a total of $r+1$ bivariate polynomials used in the iterative interpolation algorithm and each bivariate polynomial has a Y-degree equal to r. We may further assume $r < 10$ for all practical applications of the soft-decision decoder. To implement DCC operations for all bivariate polynomials in parallel, we actually do not need $(r+1)^2$ copies of the PE's (processing engine) shown in Figure 7.6. This is because a lot of hardware sharing is possible. Since all monomials in a bivariate polynomial are computed sequentially for all $(r+1)$ bivariate polynomials, the $(r+1)^2$ PE's can share the accumulator with embedded modulo operation that is used compute $\{\tilde{x}[s-a]\}_{(2^p-1)}$, and only 1 copy of the array of smaller finite field multipliers with one fixed multiplicand used to compute $\{\tilde{y}[t-b]\}_{(2^p-1)}$ is needed and are shared among the PE's. This translates into the following required arithmetic hardware units. A total of p 1-bit full adders are needed to compute $\{\tilde{x}[s-a]\}_{(2^p-1)}$ as shown in Figure 7.4 (a). We also need $(2r+1)$ copies of modulo (2^p-1) adder shown in Figure 7.3.1. Out of these $(2r+1)$ modulo (2^p-1) adders, at most r of them are used to compute $\{\tilde{y}l\}_{(2^p-1)}$ for $l = 2, ..., r$ since 2 of them are needed to compute $\{7\tilde{y}\}_{(2^p-1)}$. The rest $(r+1)$ of them are used in computing $\{\tilde{x}[s-a]+\tilde{y}[t-b]\}_{(2^p-1)}$ as shown in Figure 7.4 (b). Note that each hardware unit shown in Figure 7.3.1 contains two p-bit carry-look-ahead adders. Thus a total of $(4r+2)$ p-bit carry-look-ahead adders are needed. In addition, $p*(r+1)$ 1-bit full adders are required for $(r+1)$ copies of the hardware unit shown in Figure 7.4 (b). In the same manner, only $(r+1)$ copies of the hardware module used to carry out antilogarithm as shown in Figure 7.5 are needed. In summary, the proposed architecture requires the following arithmetic units: $(r+1)(r+2)$ p-bit finite-field multipliers and $(r+1)r$ p-bit finite-field adders, $p(r+2)$ 1-bit full adders, $(4r+2)$ p-bit integer carry-look-ahead adders. In addition, $2(r+1)$ combinational logic units are needed to implement the LUT's used in the antilogarithm operation as described in Section 7.3.1. This architecture is shown in Figure 7.7. In this architecture, the hardware module computing all monomial $\binom{s}{a}\binom{t}{b}x^{s-a}y^{t-b}$'s is called HME (hybrid multiplication engine) and the rest is divided into $(r+1)$ MACE (multiplication accumulation engine). In Figure 7.7, there are r MUXes in the HME with incremental input sizes that are used to select the desired $\tilde{y}l$ for $l = 0, 1, ..., r$. These MUXes are put in the figure only for illustrative purpose as they can be avoided from implementation due to the following property of the Groebner-basis interpolation algorithm. At each interpolation point (x_i, y_i) with multiplicity m_{x_i,y_i}, a double loop of iterations is performed with an outer loop on b and an inner loop on a. It is easy to observe the regularity at the output of the r MUXes as the value of b traverse from 0 to $(m_{x_i,y_i}-1)$. Thus the r registers that store $\tilde{y}l$ for $l = 0, 1, ..., r$ should be implemented as shift registers, which at the end of each inner loop, should downshift by one entry and a zero should be stuffed into the top register.

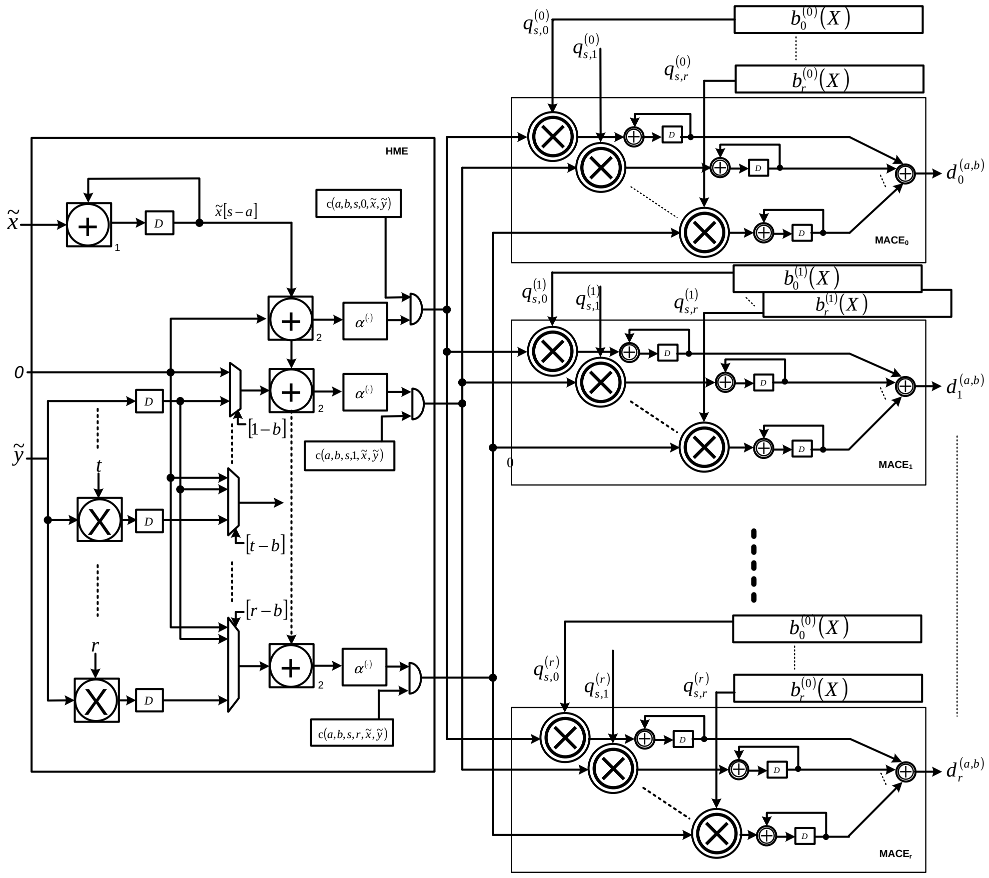

Figure 7.7: Architecture of the overall DCC Processor.

We also want to mention that the proposed DCC architecture is very scalable. With our hybrid representation of the $\mathbb{F}_{2^p}$ number, the DCC operation consists of evaluating all monomials independently at a certain point as shown in (7.4). Thus scaling the architecture is a matter of how many monomial evaluations one wants to perform in parallel. So the DCC architecture can be easily adapted to different decoding speed requirement.

7.3.2 Architecture for Polynomial Update

In this subsection, we propose a fast and scalable polynomial update architecture. It can be shown that our architecture achieves a smaller latency than prior efforts [AKS04b]. In addition, a new memory access scheme is presented so that requirement on storage area can be greatly reduced.

The equation used for polynomial update in the Groebner-basis interpolation algorithm is reiterated as follows.

$$\mathcal{Q}_v(X,Y) := \begin{cases} d_\eta^{(a,b)}\mathcal{Q}_v(X,Y) + d_v^{(a,b)}\mathcal{Q}_\eta(X,Y) & \text{if } v \neq \eta \\ X\mathcal{Q}_v(X,Y) + x\mathcal{Q}_v(X,Y) & \text{if } v = \eta \end{cases} \tag{7.6}$$

for $v = 0, 1, ..., r$.

In this chapter, the polynomial with index η is referred to as the "pivot" polynomial and note that the "pivot" polynomial may change from iteration to iteration. An architecture implementing (7.6) with a total of $(r+1)$ PUE's (polynomial update engine) operating in parallel is shown in Figure 7.8. Each PUE consists of $2(r+1)+1$ $\mathbb{F}_{2^p}$ multipliers and $(r+1)+1$ $\mathbb{F}_{2^p}$ adders. This architecture has a redundant update for the "pivot" polynomial, $\mathcal{Q}_\eta(X,Y)$, at each iteration as the "pivot" polynomial is also updated as a "non-pivot" polynomial. This redundancy costs 2 extra multipliers and 1 extra adder for each PUE. However, the amount of multiplexing and routing required is greatly reduced compared with "non-redundant" implementation. Without the "redundant" update, for each PUE in Figure 7.8, p copies of $(r+1):(r+1)$ MUXes are required to route the appropriate polynomial coefficients to the multipliers and another p copies of $(r+1):(r+1)$ MUXes are required to route the updated coefficients back to the appropriate RAM banks, where p is the number of binary bits needed to represent one polynomial coefficient in $\mathbb{F}_{2^p}$. A standard implementation of the $(r+1):(r+1)$ MUX requires $(r+1)$, $(r+1):1$ MUXes. For example, let us assume that $r=5$ and $p=8$, as in the example we will fully develop in Section 7.4. An $6:1$ MUX is equivalent to 5, $2:1$ MUXes. The total number of 2:1 MUXes required to route polynomial coefficients to the multipliers amounts to $2 \times 8 \times 6 \times 5 = 480$ for each PUE. If our "redundant" approach is applied, only $2 \times 8 \times 5 = 80$, $2:1$ MUXes are needed to route the polynomial coefficients to the multipliers and $8 \times 6 = 48$, $2:1$ MUXes are needed to route the update polynomial coefficients back to the corresponding RAM banks. The 2 "redundant" multipliers and 1 "redundant" adder consume $2 \times 77 + 8 = 162$ 2-input XOR gates and $2 \times 64 = 128$ 2-input AND gates. Thus in addition to the number of arithmetic operators required by equation (7.6), our method requires $80 + 48 = 128$, $2:1$ MUXes, 162 2-input XOR gates and 128 2-input AND gates in total, which is, in general, smaller in area compared with 480 $2:1$ MUXes, as a $2:1$ MUX is comparable in area with a 2-input XOR gate.

The coefficients of the $(r+1)$ polynomials are stored in $(r+1)^2$ banks of RAM as the coefficients of monomials with the same Y-degree in a polynomial are stored in the same bank. From Figure 7.8, it seems that the RAM's need to be implemented as dual-port RAM's because coefficients are read from the RAM's by the PUE's and, at the same time, updated coefficients are written back to the RAM's. The dual-port RAM solution is also used in polynomial update architecture of [AKS04b]. Since dual-port RAM usually costs significantly more area than its single-port counterpart, a single-port solution is desirable. The word size of each RAM can be doubled to store 2 coefficients at the same location and introduce an input buffer and an output buffer to each bank of RAM. With a simple controller, the following RAM access schedule can be applied: At an even clock cycle, 2 coefficients are read from each bank of RAM, one of them is sent to the corresponding PUE and the other one is stored in the output buffer. At the same time, the updated coefficient is temporarily buffered at the input. At an odd clock cycle, the coefficient stored at the output buffer is sent to the corresponding PUE. At the same time, the newly updated coefficient, combined with the coefficient previously stored in the input buffer, is written in to the bank of RAM. For fast application, the finite-field multipliers and the MUXes shown in Figure 7.8 can all be properly pipelined to meet a desired critical-path delay requirement. Here let us assume that a total of ξ_2 stages of pipelining is used for the $\mathbb{F}_{2^p}$ multiplier and the $(r+1):1$ MUX . In addition, another $(r+1):1$ MUX sits at the input of each bank of RAM and it can be assumed that ξ_{mux} stages of pipeline is used for those MUXes. Thus if the maximum X-degree of all $(r+1)$ polynomials is equal to d_X at a certain iteration, the number of clock cycles required to update all polynomials is approximately equal to $d_X + \xi_2 + \xi_{mux}$.

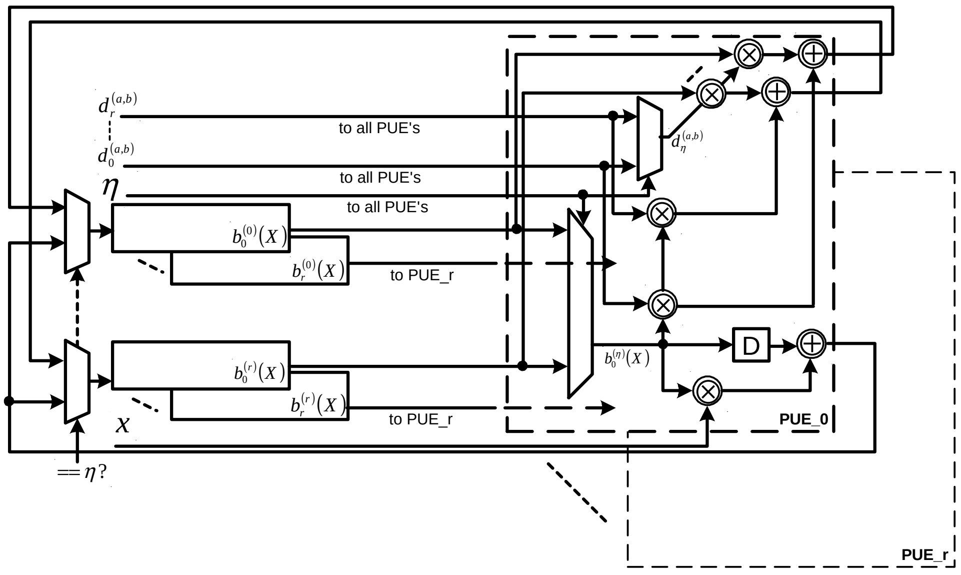

Figure 7.8: Polynomial Update Architecture-I

It should be mentioned that the PU architecture is very scalable. Here we choose to update $(r+1)^2$ coefficients for monomials with the same X-degree in parallel. But the degree of parallelism can be easily scaled up or down to meet different update speed requirement.

The polynomial update architectures presented in [AKS04b] is based on a reformulation of equation (7.6) as follows.

$$\mathcal{Q}_v(X,Y) := \begin{cases} \mathcal{Q}_v(X,Y) + \frac{d_v^{(a,b)}}{d_\eta^{(a,b)}} \mathcal{Q}_\eta(X,Y) & \text{if } v \neq \eta \\ X\mathcal{Q}_v(X,Y) + x\mathcal{Q}_v(X,Y) & \text{if } v = \eta \end{cases} \tag{7.7}$$

for $v = 0, 1, ..., r$.

Compared with our newly proposed architecture, the architectures based on (7.7) has 1 more $\mathbb{F}_{2^p}$ inverter and 1 more $\mathbb{F}_{2^p}$ multiplier in the longest path. If we assume that ξ_{inv} levels of pipelining is used for the inverter, then with the same level of parallelism, the architectures given in [AKS04b] have $\xi_{inv} + \xi_2$ more clocks of latency in each interpolation iteration than our architecture. Since hundreds of iterations are usually needed in soft-decision decoding of a single received codeword, this $\xi_{inv} + \xi_2$ clocks of latency in each iteration translates to a huge decoding latency. However, architectures presented in [AKS04b] can lead to hardware area savings compared to our new architecture, thus those PU architectures should be chosen in an area-constraint driven design.

7.3.3 Polynomial Update Controller

The polynomial update engines shown in the previous section assumes, as an input, the index of the minimum weighted degree polynomial, among all polynomials with non-zero discrepancy coefficients, i.e., the “pivot” polynomial. This index, η, needs to be computed based on the output of the computed $d_v^{(a,b)}$'s and the weighted degrees of the polynomials. Here the hardware module performing this function is referred to as PUC (polynomial update controller). Since the index η has to be computed for each iteration of the interpolation process, efficient design is important to ensure that it only introduces the minimum overhead clock cycles to the whole interpolation process. Let us assume, without loss of generality, that an $(r+1)$-entry register array is allocated to store the weighted degrees and the indices of the $(r+1)$ polynomials used in the interpolation process with each register storing the (O_v, v) pair (O_v represents the weighed degree of polynomial $Q_v(X,Y)$ as shown in our algorithm description). In order to minimize the computation delay that is required to generate the index η, we propose an PUC architecture assuming that the polynomial indices are sorted according to their associated weighted degrees before the final selection. Obviously the sorting operation can be performed in parallel with the DCC operation. It is known that only one polynomial will increase its weighted degree at each iteration. The sorting operation at each iteration is equivalently to one round bubble sort process, which consists of comparison and data swapping operations. Since the total number of polynomials used in the interpolation algorithm is usually a small number, thus the sorting operation will not introduce any further delay in general. Once all $(r+1)$ discrepancy coefficients are ready, each sorted index needs to be "labeled" to rule out all indices associated with polynomials with zero discrepancy coefficients for this iteration. This can be performed with the circuit shown in Figure 7.9 (a), where the π_v's represent registers that store the sorted polynomial indices with π_0 storing the one associated with the minimum weighted degree among all polynomials and the l_v's represent the 1-bit registers that store flags indicating whether the corresponding polynomial has a zero discrepancy coefficient at this iteration. For example, if $l_0 = 1$ after the "labeling" operation, we know that the polynomial whose index is stored in register π_0 has a non-zero discrepancy coefficient at this iteration. The overall delay of this circuit is a sum of the delay of the logic used to decide whether the corresponding discrepancy coefficient is non-zero, which is essentially a p-input OR gate, and the delay of a $(r+1) : 1$ MUX.

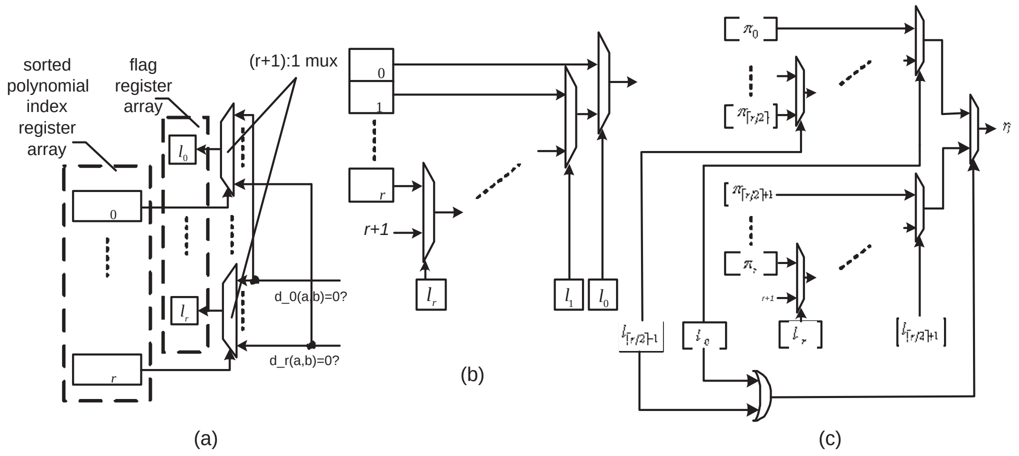
sorted polynomial index register array
flag register array
(r+1):1 mux
0
r
l_0
l_r
d_0(a,b)=0?
d_r(a,b)=0?
(a)
0
1
r
r+1
l_r
l_1
l_0
(b)
π_0
$\pi_{\lceil r/2 \rceil}$
$\pi_{\lceil r/2 \rceil+1}$
π_r
r+1
η
$l_{\lceil r/2 \rceil+1}$
l_0
l_r
$l_{\lceil r/2 \rceil+1}$
(c)

Figure 7.9: Implementation of the "labeling" and "select" operations.

After the labeling operation, the index η can be determined by selecting the 1st register, among the π_0, π_1, ..., π_r sorted list, with non-zero labeling. This can be implemented with the following circuit shown in Figure 7.9 (b). The overall delay of this circuit is equal to the delay of $(r+1)$ serially concatenated $2 : 1$

MUXes. When r is large, the delay may be longer than desired. In this case, an alternative parallel architecture shown in Figure 7.9 (c) can be used. This implementation reduces the delay of the circuit to $(\lceil r/2 \rceil + 1)$ serially concatenated $2:1$ MUXes. Actually this architecture can be further extended all the way to a binary tree type of MUX array to reduce the delay to a minimum.

With the implementation of “labeling” and “select” functions given above, a total of ξ_3 stages of pipelining is needed. Thus it takes ξ_3 clock cycles to generate the index η of the polynomial with minimum weighted degree and non-zero discrepancy coefficient.

7.3.4 Concurrent Discrepancy Coefficient Computation and Polynomial Update

From the architectures for DCC and PU given in previous subsections, the polynomial coefficients are generated sequentially from the PUE’s. Once a coefficient is generated, it is no longer needed for this iteration of the polynomial update procedure. In addition, the MACE’s also expect sequential input of the polynomial coefficients. Thus the DCC and polynomial update processes can be carried out concurrently, i.e., once a coefficient of a polynomial has been updated for the current iteration, it can be immediately used for next generation’s discrepancy computation of the corresponding polynomial. This can be illustrated with the following block diagram in Figure 7.10. Note that the same level of parallelism should be used for the DCC architecture and the PU architecture in order to achieve the maximum overlapping between these two operations in consecutive interpolation iterations.

Now let us estimate the number of clock cycles required for each iteration of the interpolation process. As it is mentioned earlier, if we adopt the architecture shown in Figure 7.8 for polynomial update process, there is ξ_2 clock cycles of pipeline overhead. Note that the pipeline overhead ξ_{mux} associated with the $(r+1):1$ MUXes at the input to the coefficient RAM banks is not counted because after ξ_2 clock cycles, coefficients are available for use in discrepancy co-

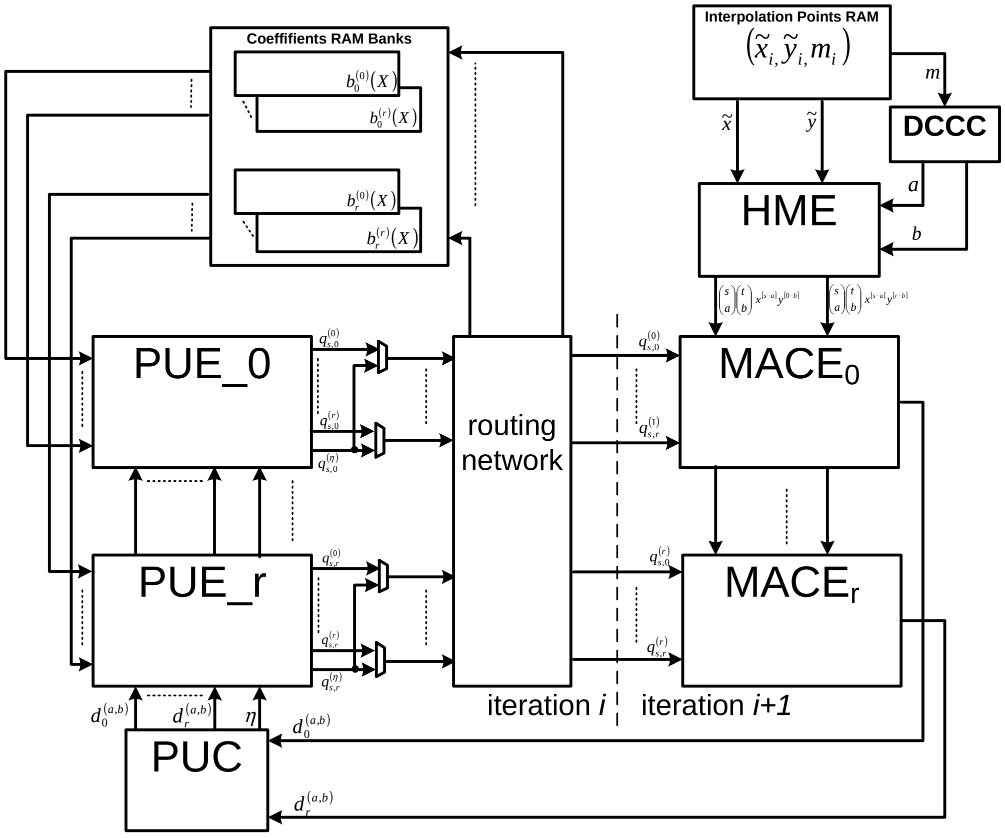

Figure 7.10: Overall Interpolation Architecture with concurrent DCC and PU operations.

efficient computation. From our discussion of the DCC architecture, the DCC procedure has $\xi_0 + \xi_1$ clock cycles of pipeline overhead. Among the these clock cycles, ξ_0 cycles are caused by the pipeline overhead of the HME's. Since the computation in the HME's does not depend on the polynomial coefficients, but only depend on the X and Y coordinates, these computations don't have to wait until the coefficients are available from the PUE's to start but can be carried out in parallel with the last ξ_0 clock cycles of the polynomial update procedure. This enables a significant saving of required clock cycles for each iteration of the interpolation process. With this saving, it takes $d_X + \xi_1 + \xi_2 + \xi_3$ cycles to finish 1 iteration of the interpolation algorithm assuming the maximum X-degree of the polynomials in this iteration is equal to d_X. This can be illustrated with the timing diagram shown in Figure 7.11. With the above estimate, we can now further estimate the total number of clock cycles required to implement the iterative interpolation algorithm. Let us assume the total number of interpolation constraints to be enforced is C, usually referred to as the interpolation cost [KV03a], thus a total of C iterations are needed for the iterative interpolation algorithm. The interpolation algorithm has the property that at each iteration, at most 1 of the $(r+1)$ polynomials grows its X-degree by 1. Thus when the interpolation process stops, the maximum X-degree among all $(r+1)$ polynomials is at most $\lceil \frac{C}{r+1} \rceil$. Due to the uniform growth property of the X-degrees of the polynomials in the interpolation process, we can assume that on average, the X-degree of the polynomials involved in the DCC operation is equal to $\lceil \frac{C}{2(r+1)} \rceil$. Thus the total number of clock cycles required for all interpolation operations is approximately $(\lceil \frac{C}{2(r+1)} \rceil + \xi_1 + \xi_2 + \xi_3)C$.

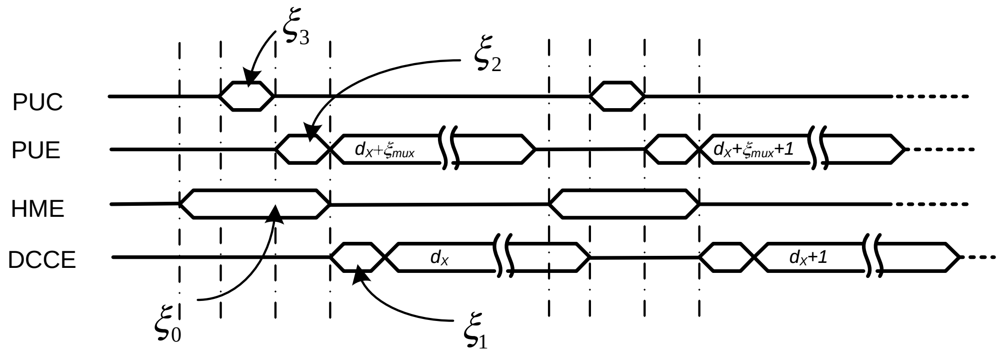

Figure 7.11: Concurrent DCC and PU Timing Diagram.

7.3.5 Architecture for Re-encoded Interpolation

The re-encoding coordinate transformation techniques have been presented in [GKKG02] [KMVA03]. In this subsection, we first highlight the algorithmic changes from the iterative interpolation algorithm point of view and then discuss modifications that can be made to the interpolation architecture presented in earlier subsections to handle re-encoded interpolation.

First of all, in the re-encoded interpolation, the bivariate polynomial sets are usually initialized with some “tail” polynomials as follow

$$Q_v(X,Z) = T_v(X)Z^v \; for \; v = 0,1,...,r,$$

where the ”tail” polynomials depend on the X coordinates and multiplicities of the re-encoding interpolation points and the value of v. Secondly, after re-encoding coordinate transformation techniques are applied to the original interpolation point set $\mathcal{P}$, the remaining interpolation points can be classified into two sets: Set I and Set II. Set I includes the points whose X coordinates are different from all re-encoding points’ X coordinates. For these points, the DCC operation for polynomial $\mathcal{Q}_v(X,Z)$ uses exactly the formula as its non-re-encoded counterpart, i.e.,

$$\begin{aligned} d_v^{(a,b)} &= \mathbf{coef}\big(\mathcal{Q}_v(X+x, Z+z), X^a Z^b\big) \\ &= \sum_{t=b}^{r} \sum_{s=a}^{w_{v,t}} \binom{s}{a} \binom{t}{b} q_{s,t}^{(v)} x^{s-a} z^{t-b} \qquad (7.8) \\ &\text{for } v = 0,1,...,r. \end{aligned}$$

When computing the discrepancy coefficients for points in Set I, the weighted degree is defined as follows $\mathcal{O}_v = \deg_{1,-1} \mathcal{Q}_v(X,Z)$. Set II include points whose X coordinates coincides with the re-encoding points’ X coordinate. For a point (x,z) in this set, the DCC operation for polynomial $\mathcal{Q}_v(X,Z)$ uses the following formula

$$d_v^{(a,b)}$$

$$\begin{aligned}
&= \mathbf{coef}\big(\mathcal{Q}_v^{(x)}(X+x, Z+z), X^a Z^b\big) \\
&= \sum_{t=0}^{r} \mathbf{coef}\{\mathcal{Q}_{v,t}(X+x)(Z+z)^t, X^{a-(m-t)} Z^b\} \\
&= \sum_{t=t_0}^{r} \mathbf{coef}\{\mathcal{Q}_{v,t}(X+x)(Z+z)^t, X^{a-(m-t)} Z^b\} \\
&= \sum_{t=t_0}^{r} \mathbf{coef}\{\sum_{t'=0}^{t} \binom{t}{t'} z^{t-t'} Z^{t'} \sum_{s=0}^{w_{v,t}} q_{s,t}^{(v)} (X+x)^s, X^{a-(m-t)} Z^b\} \\
&= \sum_{t=t_0}^{r} \sum_{s=s_0}^{w_{v,t}} \binom{s}{a-(m-t)} \binom{t}{b} q_{s,t}^{(v)} x^{s-\left(a-(m-t)\right)} z^{t-b}
\end{aligned}$$

$$\text{for } v = 0, 1, ..., r. \tag{7.9}$$

In the equation shown above, $t_0 = max([m-a], b)$ and $s_0 = a-(m-t)$. $\mathcal{Q}_{v,t}(X) = \sum_{s=0}^{w_{v,t}} q_{s,t}^{(v)} X^s$ is a polynomial in X such that $\mathcal{Q}_v(X, Z) = \sum_{t=0}^{r} \mathcal{Q}_{v,t}(X) Z^t$. $\mathcal{Q}_v^{(x)}(X, Z) = \sum_{t=0}^{r} (X-x)^{m-t} \mathcal{Q}_{v,t}(X) Z^t$ is a "scratch" polynomial and needs not to be stored and m is the multiplicity assigned to the re-encoding point with X coordinate equal to x. The second equality above follows from the trivial observation that $\mathbf{coef}\{P(X)X^i, X^j\} = \mathbf{coef}\{P(X), X^{j-i}\}$ for any integer i, j as long as $P(X)X^i$ is a valid polynomial. The third equality, with the index to the summation changed to t_0 from 0, follows from the fact that if $t < m-a$, then $a-(m-t) < 0$, and $\mathcal{Q}_{v,t}(X+x)(Z+z)^t$ does not have any monomial with negative X power, and that if $t < b$, $\mathcal{Q}_{v,t}(X+x)(Z+z)^t$ does not have any monomial with Y degree equal to b. It is easy to verify that in the expression of last equality above, all powers of x and z are non-negative since it is always true that $a+b < m$. When computing the discrepancy coefficients for points in Set II, we need to use the weighted degree $\mathcal{O}_v = \deg_{1,0} \mathcal{Q}_v^{(x)}(X, Z)$.

For re-encoded interpolation, it can be assumed that a hardware block called "re-encoding frontend" that performs the re-encoding coordinate transformation operations described in [KMVA03]. An example implementation of the "re-encoding frontend" is given in [MVW06b]. This block also transforms all X and Z coordinates from regular representation to power representation as needed for the HME block. In addition, the re-encoding frontend initializes the interpolation polynomials with the appropriate tail polynomials as shown earlier. The "re-encoding frontend" can be combined into the multiplicity-assignment block shown in Figure 1.1 of Chapter 1 and operates in a pipelined fashion with the interpolation block, i.e., it works on the most recently received codeword while the interpolation engine works on the previously received codeword. From (7.8) and (7.9), it can be seen that the HME and MACE presented in Section 7.3.1 can also be applied to re-encoded interpolation. For interpolation points in set I, no change needs to be made to the HME and MACE's. For interpolation points in set II, $(r+1)$ copies of the accumulator are required to compute $\tilde{x}[s-a'(t)]$, for $a'(t) \overset{\text{def}}{=} a-(m-t)$ and $t = 0, 1, ..., r$. This is because this accumulator can no longer be shared for all PE's shown in Figure 7.6 as the power of x is a function of both running index s and t.

In summary, a new interpolation architecture is presented in this section and an estimate of its latency and hardware complexity is given in Table 7.1. In the table, C denotes the total number of interpolation constraints, r is the largest Y-degree of the interpolation polynomials. (note that r is, in general, proportional to the square root of the value of C.) ξ_0, x_1, x_2 and x_3 are the number of pipelining stages used for the HME's, the MACE's, the PUE's and the PUC, respectively. In addition, the number L represents the degree of parallelism that can be applied to the scalable architecture. The proposed architecture has the following advantages over prior efforts [AKS04b] [GKKG05] [GKKG02] [GKG04].

- The architecture can be highly pipelined and is very scalable. It has smaller overall interpolation latency.
- With our architecture, the DCC and PU operations can be overlapped to the maximum extent.
- The architecture requires less storage area as single-port RAM, instead of dual-port RAM [AKS04b] or registers [GKG04], is used to store the polynomial coefficients.

Table 7.1: Latency and Hardware Complexity Estimate of the Interpolation Architecture

	latency	Hardware	
		Device	Quantity
DCC	$(\lceil\frac{C}{2(r+1)L}\rceil$ $+\xi_0+\xi_1)C$	$\mathbb{F}_{2^p}$ multiplier	$(r+1)+(r+1)^2L$
		$\mathbb{F}_{2^p}$ adder	$r(r+1)L$
		1-bit full adder	$p(r+2)$
		p-bit CLA	$4r+2$
		Antilog LUT	$2(r+1)$
		MUX (2:1)	$p(r+1)$
		1-bit register	$p(r+1)\xi_0$ $+p(r+1)^2\xi_1L$
		SRAM (bit)	p(r+1)C
PU	$(\lceil\frac{C}{2(r+1)L}\rceil$ $+\xi_2+\xi_3)C$	$\mathbb{F}_{2^p}$ multiplier	$(r+1)(2r+3)L$
		$\mathbb{F}_{2^p}$ adder	$(r+1)(r+2)L$
		MUX (2:1)	$p(r+1)(3r+1)L$ $+r(r+1+\log_2 r)$
		1-bit register	$p(r+1)^2\xi_2L$ $+(r+1)^2\xi_3$

7.4 Example: Interpolation Architecture for a $(255, 23\ 9)$ Reed-Solomon Code

In this section, the new interpolation architecture presented in Section 7.3 is applied to the soft-decision decoder for a (255,239) RS code defined on $\mathbb{F}_{2^8}$. Our new architecture is also compared with other existing ones in terms of hardware complexity and decoding latency. This section is organized as follows: The algorithm-level interpolation complexity associated with the Groebner-basis interpolation algorithm is estimated first. we then present the hardware requirement and latency estimate for DCC, PU and PUC blocks, respectively. Finally an overall gate count and decoding throughput estimate are given and are compared with prior works. In this section, without specific mentioning, all logic gates are assumed to be 2-input gates.

7.4.1 Algorithm-Level Interpolation Complexity

Previous simulations [AKS04a] have shown that soft-decision decoder with a total interpolation cost, i.e., the total number of constraints to be enforced, equal to 3800 can provide more than 0.5dB coding gain at a codeword error rate of 10^{-5} over the conventional hard-decision decoder. The number of bivariate polynomials required for the Groebner-basis interpolation algorithm shown in Section 7.3 can be computed as follows.

$$r = \min\{t \in Z : (t+1)(\frac{t(k-1)}{2} + k) > C\}$$

where Z represents the set of all integers and C is the targeted interpolation cost. Plugging $k = 239$ and $C = 3800$ in the above equation, we obtain $r = 5$. Thus a total of 6 bivariate polynomials are needed for the iterative interpolation procedure, which translates to 6×6 banks of RAM as described in Subsection 7.3.2. For the re-encoding coordinate-transformation based technique, it has been shown in [KMVA03] that the total number of required bivariate polynomials required is the same as the regular interpolation procedure. In the worst case, the remaining constraint, after interpolation, can be estimated as $C_{RE} = C \times \frac{n-k}{n} = 3800 \times \frac{16}{255} \cong 239$. Thus the expected X-degree of all polynomials at the end of the re-encoded interpolation is around $\frac{C_{RE}}{(r+1)} = \frac{239}{6} \approx 40$. Let us conservatively assume that the maximum X-degree is 50, which translates to $6 \times 6 \times 50 = 1800$ bytes of RAM required to store all coefficients.

7.4.2 Area and Latency Estimate of Discrepancy Coefficient C-omputation

As it has been mentioned in Subsection 7.3.1, a total of $(r+1)(r+2) = 42$ $\mathbb{F}_{2^8}$ multipliers, $(r+1)r = 30$ $\mathbb{F}_{2^8}$ adders, $p(r+2) = 56$ 1-bit full adders and $(2r+1) = 11$ 8-bit modulo 255 adders as shown in Figure 7.3.1 are required. Due to the application of re-encoding coordinate transformation, $pr = 40$ extra 1-bit full adders have to be used to accommodate the fact that the accumulator with modulo operation can not be shared as explained in the end of Subsection 7.3.5.

Let us assume that the primitive polynomial that defines $\mathbb{F}_{2^8}$ is $p(X) = 1 + X^2 + X^3 + X^4 + X^8$, a straightforward bit-parallel $\mathbb{F}_{2^8}$ multiplier can be implemented in combinational logic with 77 XOR and 64 AND gates and the longest path consists of 1 AND gate and 6 XOR gates. Since addition in $\mathbb{F}_{2^8}$ can simply be implemented by bitwise XOR operation, each $\mathbb{F}_{2^8}$ consists of 8 parallel XOR gates. A 1-bit full adder can be realized with 2 AND gates, 2 XOR gates and 1 OR gate with the longest path consisting of 1 XOR gate, 1 AND gate and 1 OR gate. Each 8-bit modulo 255 adder consists of 2 8-bit carry-look-ahead adders (CLA) and 8 MUXes. It is a common practice to break large CLA into parallel concatenated smaller ones to reduce the longest path delay. Thus we propose the following architecture shown in Figure 7.12 for the 8-bit modulo 255 adder based on CLA's. The architecture works as follows. The two 8-bit numbers, namely f and g, are partitioned into two parts with each part consisting of two 4-bit numbers. We compute the summations for each part independently while considering two possible carry-in's, i.e., 0 and 1, for each summation. The final result depends on the output of MUX3 and that of MUX2. The output of MUX3 indicates if there is a carry out from the original addition, i.e., $f + g$. The output of MUX2 tells which carry in signal should be used for the upper 4-bit addition. As can be seen for the figure, the overall computation delay for this part is equal to a 4-bit CLA delay and 3 MUX delay. It can be shown that 21 AND gates, 10 OR gates and 8 XOR gates are required to implement a 4-bit CLA with the longest path consisting of 3 AND gates, 3 OR gates and 1 XOR gate.

In addition to the arithmetic units described above, 12 combinational logic based LUT's are required for the antilogarithm operation. Each LUT has 16 entries and the synthesis reports show that the top LUT in Figure 7.5 requires an area about 25% of an $\mathbb{F}_{2^8}$ multiplier and the bottom LUT in Figure 7.5 maps to an area about 31% of an $\mathbb{F}_{2^8}$ multiplier. The longest paths of both LUT's have a delay that is about 50% of that of the $\mathbb{F}_{2^8}$ multiplier. Thus we may conservatively assume that the top LUT has 35 XOR gates and the bottom LUT has 43 XOR gates in area, and that both of them have 4 XOR gates in their longest paths.

In summary, the longest path of the HME unit consists of 2 1-bit full adders,

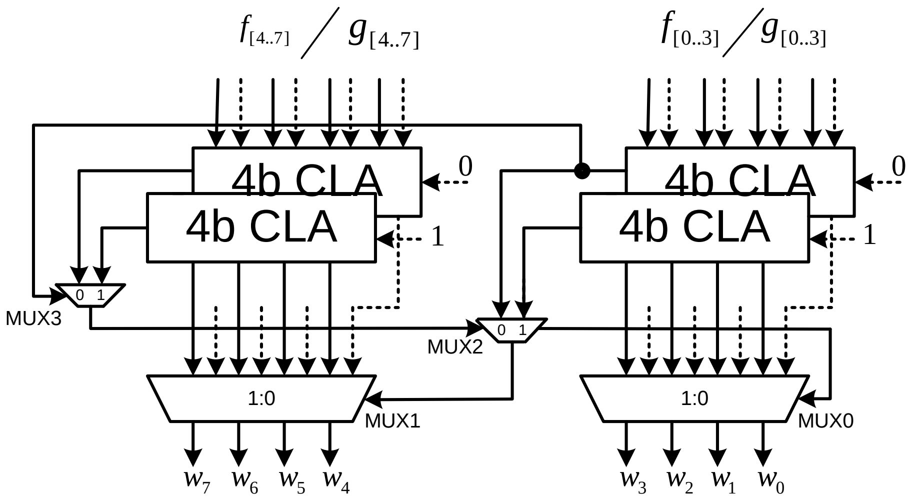

Figure 7.12: Architecture for Fast 8-bit Modulo 255 Addition.

1 4-bit CLA and 3 MUXes, 1 combinational-logic LUT, 1 $\mathbb{F}_{2^8}$ multiplier and 1 AND gate, which translates to 7 AND gates, 5 OR gates, 13 XOR gates, and 3 MUXes. Thus 6 pipelining stages can be applied to ensure that the critical path does not contain more than 4 serially-concatenated XOR gates' equivalent delay. Note that the pipelining overhead associated with the HME unit does not contribute to the overall delay of the interpolation process as it has been shown in Subsection 7.3.4 that the HME unit can finish its computation in parallel with previous iteration's polynomial update. We may assume that $(r+1) = 6$ 8-bit numbers need to be stored for each pipelining stages, thus the total number of registers required is equal to $6 \times 8 \times 6 = 288$. The longest path of the MACE unit consists of a $\mathbb{F}_{2^8}$ multiplier, a $\mathbb{F}_{2^8}$ adder and a 6-input $\mathbb{F}_{2^8}$ adder, which amounts to 10 XOR gates and 1 AND gate. a total of 3 pipelining stages can be applied to the MACE units, i.e., we have $\xi_1 = 3$. For each pipelining stages in MACE, $(r+1)^2 = 36$ 8-bit numbers need to be stored. Thus a total of $36 \times 8 \times 3 = 864$ registers have to be used.

7.4.3 Area and Latency Estimate of Polynomial Update

When $r = 5$, each PUE shown in Figure 7.8 requires 16 6:1 MUXes, which in area, is equivalent to $16 \times 5 = 80$, 2:1 MUXes. Each PUE also includes 13 $\mathbb{F}_{2^8}$ multipliers 7 $\mathbb{F}_{2^8}$ adders. Besides the 6 PUE's, 288 MUXes are required to appropriately route the updated coefficients back to the coefficient RAM banks. The longest path from where the coefficients are read out from the RAM's to after they have been updated consists of 1 6:1 MUX, 1 $\mathbb{F}_{2^8}$ multiplier and 1 $\mathbb{F}_{2^8}$ adder, which translates to 3 MUXes, 7 XOR gates and 1 AND gate. Thus we may choose $\xi_2 = 3$ pipelining stages in our architecture and a total of $36 \times 8 \times 3 = 864$ registers are needed to store the intermediate results of the pipelining stages.

7.4.4 Area and Latency Estimate of Polynomial Update Controller

The major role of the PUC unit is to figure out, after the DCC operation, which polynomial is the "pivot" polynomial. The PUC works on sorted weighted degrees of the polynomials. As we mentioned earlier, the worst-case X-degree can be at most 50, thus 8 bits are enough to store the weighted degrees. With $r = 5$, the "labeling" operation requires 6 6:1 MUXes and 6 8-input AND gates. Note that the 8-input AND gate is used to decide whether a computed discrepancy coefficient, $d_v(a, b)$, is equal to 0 or not. They can be mapped to 30, $2 : 1$ MUXes and 42 AND gate with the longest path consists of 3 MUXes and 3 AND gates. From Figure 7.9 (c), the "select" operation requires 15 MUXes and longest path has 3 MUXes on it. Overall, the delay of the PUC unit is equal to that of 6 MUXes and 3 AND gates. We can choose $\xi_3 = 2$ pipelining stages. it can be assumed that the 1st pipelining stage is inserted into the 6:1 MUXes of the "labeling" operation thus requires no more than $6 \times 6 = 36$ registers. In addition, 3 registers are required to store the value of η at the output of the 2nd pipelining stage.

Table 7.2: Estimation for Gate Counts and Critical Paths

	area	longest path	pipeline stages
$\mathbb{F}_{2^8}$ multiplier	77XOR+64AND	6XOR+1AND	N/A
$\mathbb{F}_{2^8}$ adder	8XOR	1XOR	N/A
1-bit full adder	2XOR+2AND+1OR	1XOR+1AND+1OR	N/A
4-bit CLA	8XOR+21AND+10OR	1XOR+3AND+3OR	N/A
Comb. Logic LUT1	35XOR	4XOR	N/A
Comb. Logic LUT2	43XOR	4XOR	N/A
8-bit Mod. 255 Adder	32XOR+84AND+40OR+10MUX	1XOR+3AND+3OR+3MUX	N/A
HME	1362XOR+1280AND +408OR+80MUX	13XOR+7AND +5OR+3MUX	6
MACE's	3252XOR+2304AND	10XOR+1AND	3
PUE's	1057XOR+832AND+768MUX	7XOR+1AND+3MUX	3
PUC	42AND+45MUX	3AND+6MUX	2
total	5713XOR+4416AND +408OR+893MUX	N/A	N/A

7.4.5 Overall Area and Throughput Estimate of the Interpolation Architecture

Table 7.4.4 gives the gate count and critical path of each building block, except the memory and control block, for our proposed interpolation architecture. In the table, the gate count and critical path of fundamental hardware modules, such as the $\mathbb{F}_{2^8}$ multiplier, is given first, followed by those of large building blocks, such as the HME. All building blocks are pipelined such that their resulting critical path is no longer than 4 concatenated XOR gates. In Table 7.3, we give an estimate of the register storage required for the pipelining stages associated with the hardware blocks given in Table 7.4.4. With start-of-the-art 90nm CMOS technology, our architecture can support a clock frequency of over 1.5 GHz considering the critical path delay is only 4 XOR operations. As we have mentioned in Subsection 7.3.4, the total number of clock cycles required to finish the interpolation process with a cost equal to C is $(\lceil \frac{C}{2(r+1)} \rceil + \xi_1 + \xi_2 + \xi_3)C$. In this example, we have $C = 239$, $r = 5$ and $\xi_1 + \xi_2 + \xi_3 = 8$, thus we need

Table 7.3: Storage Requirement Estimate

Block	HME	MACE's	PUE's	PUC	Total
Register Count	288	864	864	39	2095

$(\lceil \frac{239}{12} \rceil + 8)239 = 6692$ clock cycles to finish the interpolation process. Thus the throughput of the interpolation architecture is at least $\frac{1.5*10^9 \times 8 \times 255}{6692} \approx 450$ Mbps.

We would like to mention that the proposed interpolation architecture is highly scalable, thus it is possible apply more parallelism to both the discrepancy computation and polynomial update engines to obtain even faster design. For instance, we can process 6 adjacent coefficients in both DCC and PU operations, the overall processing time for the above example is reduced to $(8 + [239/(12 * 6)]) * 235 = 2820$ cycles. Therefore, the throughput is increased to $6692/2820 * 450$Mbps ≈ 1.1 Gbps.

As we have not completed a real design (FPGA or ASIC) using the proposed architecture, we can only provide a rough comparison with other published works.

For the design presented in [GKG04], the PU process for each candidate polynomial is completed in one cycle while different candidate polynomials are updated sequentially. For the same example discussed above, a total of $235 * 6 = 1410$ cycles is needed for the PU process. There was no technical details given about the DCC process in the paper. We optimistically assume that their DCC process only contributes 30% of time that their PU process used to the overall computation delay. Thus the design requires a total of $1410 * 1.3 = 1833$ cycles. However, the critical path of the design is quite long. The paper reported a maximum clock speed of 35 Mhz with Xilinx Virtex II 8000. With 90nm CMOS technology, the maximum clock speed of the corresponding ASIC design will hardly exceed 500 Mhz. Therefore, the maximum throughput with their architecture is no more than $2390/1833 * 500/1800 * 1.5 = 543$ Mbps. On the other hand, their architecture requires significantly more hardware in the PU part and consumes more power in the overall process. This can be explained as follows. As the X degrees of all candidate polynomials incrementally grow, the hardware utilization efficiency (HUE) would be approximately 50% on average if we update an entire candidate polynomial at one cycle, where the hardware has to accommodate for the possibly largest X degree. In our design, we serially update a small number of adjacent X coefficients for each candidate polynomial. The HUE is very close to 100%. Due to the significantly higher HUE with our design, we can save nearly 50% hardware and thus 50% power consumption as well for the same target throughput. In addition, as the critical path of our design is significantly shorter than that of the design presented in [GKG04], we can further save power by lowering the supply voltage, which will lead to quadratically reduced power consumption.

Now a comparison between our new interpolation architecture and the architecture given in [AKS04b] is provided. First of all, the new architecture presented in this chapter can be extensively pipelined, while the architecture in [AKS04b] has an F_{2^8} multiplier, an F_{2^8} adder and a $2:1$ MUX in its critical path, thus the new architecture can support a much higher clock frequency. Second of all, the architecture in [AKS04b] does not use overlapped DCC and PU operations. According to the data provided in Table 3 of [AKS04b], the computation delay of the DCC operation is comparable to that of the PU operation and is completely added to the overall computation delay. This scenario is generally true. However, due to the maximally overlapped decoding, the DCC part in our design only contributes a very small portion in the overall computation delay for pragmatic cases. Note that the PU operation is basically straightforward. Although the proposed PU architecture is faster than the one presented in [AKS04b], we could use the same PU architecture without changing the overall data flow. In other words, we can assume both designs take the same amount of cycles in the PU process with equivalent hardware. Overall, our design will, in general, require less number of cycles for an entire interpolation process. Considering the shorter critical path of the proposed design, we can fairly claim that our design can generally achieve more than twice higher throughput than the one presented in [AKS04b]. Another minor point is that the DCC process presented in [AKS04b] generally requires more hardware than the proposed one.

7.5 Conclusions

In this chapter, a novel interpolation architecture for soft-decision Reed-Solomon decoders is presented. Based on the hybrid data representation, the proposed architecture not only breaks the bottleneck of the recursive computation with Horner's rule, but also enables a maximum overlapping in time between sequential iteration steps of the interpolation algorithm. By exploring the inherent property of interpolation polynomial growing, the proposed architecture is very efficient in both area and power. It was also shown that the proposed architecture is highly scalable and is thus well suited for applications with various speed requirement. Analysis has been provided to show that the proposed architecture can achieve significantly higher throughput than other published works with equivalent or lower hardware complexity.

We have presented the results of Chapter 7, in part, at *2006 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng, and *IEEE Transactions on VLSI Systems, September 2006 pp. 937-950.* Wang, Zhongfeng; Ma, Jun, The dissertation author was a joint investigator and co-author of both papers.

CHAPTER 8

Factorization Architecture

Bivariate polynomial factorization is an important step of algebraic soft-decision decoding of Reed-Solomon codes and contributes to a significant portion of the overall decoding latency. With the exhaustive search based root computation method, factorization latency is dominated by root computation, especially for RS codes defined over very large finite fields. The root-order prediction method proposed by Zhang and Parhi only improves average latency, but does not have any effect on the worst-case latency of the factorization procedure. Thus neither approach is well-suited for delay-sensitive applications. In this paper, a novel architecture based on direct root computation is proposed to greatly reduce the factorization latency. Direct root computation is feasible because in most practical applications of algebraic soft-decision decoding of RS codes, enough decoding gain can be achieved with a relatively low interpolation cost, which results in a bivariate polynomial with low Y-degree. Compared with existing works, not only does our new architecture have a significantly smaller worst-case decoding latency, but it is also more area efficient since the corresponding hardware for routing polynomial coefficients is eliminated.

8.1 Introduction

Reed-Solomon (RS) codes are the most widely used error-correcting codes in digital communications and data storage. Recently Sudan and Guruswami made a breakthrough discovery of a list decoding algorithm [Sud97], [GS99], which has larger decoding radius than conventional hard-decision decoding algorithms, such as the Berlekamp-Massey algorithm. The hard-decision list-decoding algorithm was later extended by Koetter and Vardy [KV03a] to an algebraic soft-decision decoding algorithm. The Koetter-Vardy (KV) algorithm has polynomial complexity in codeword length and can achieve significant coding gain for RS codes of all rates. All of these algorithms involve interpolation and factorization of bivariate polynomials. The interpolation procedure normally is more computationally complex than factorization. Thus more research has been directed at efficient hardware implementation of the interpolation procedure [AKS03b], [AKS04b], [GKKG02], [GKG04], [MVW06a].

In conventional KV decoding, factorization is performed only once after all interpolation constraints have been enforced. It has been shown recently in [AKS03a] that attempting factorization of judiciously selected intermediate interpolation results multiple times leads to significantly reduced interpolation complexity in algebraic soft-decision decoding of RS codes. This can be seen from Fig. 8.1, where simulation results for a $(458, 410)$ RS code defined over $\mathbb{F}_{2^{10}}$ are shown. The simulations assume BPSK modulation over the AWGN channel. If factorization is performed over intermediate interpolation results, the KV decoder can achieve 0.4dB decoding gain, over the Berlekamp-Massey decoder, at codeword error rate of 10^{-6} with a maximum multiplicity value of four. Otherwise, the maximum multiplicity value needs to be at least six. According to [KV03a], the interpolation cost is approximately $\frac{n(m+1)m}{2} = \frac{458\times5\times4}{2} = 4580$ in the former case, while it amounts to $\frac{458\times7\times6}{2} = 9618$ in the latter case. Thus about 52 percent reduction in interpolation complexity can be achieved by performing factorization on intermediate interpolation results. With the advent of this iterative interpolation and factorization technique, the computational complexity is more balanced between interpolation and factorization, and factorization latency becomes a more significant portion in the overall decoding latency. Thus a low-latency factorization architecture is of great practical value.

One major step of the factorization procedure is root computation for polynomials. The factorization architecture of [AKS04b] uses Chien search to find roots of a polynomial at the beginning of each iteration. This approach is very time consuming, especially for RS codes defined over a large finite field. In [ZP05], a root-order prediction based method was proposed by Zhang and Parhi, who observed that the orders of roots seldom change between factorization iterations. The VLSI architecture based on this observation can improve the average factorization latency. However, the worst case latency of [ZP05] is not any better than that of [AKS04b], because the root-order prediction has a non-zero failure rate and one has to resort to Chien search after detecting a root-order prediction failure. Thus the root-order prediction based architecture cannot be used in applications with a stringent latency requirement.

In this paper, we present a fast factorization architecture based on direct computation of polynomial roots. Direct root computation is only feasible for low-degree polynomials. Fortunately, this is not a problem for most practical applications of algebraic soft-decision decoding, where significant decoding gain can be achieved with relatively low interpolation cost. Low interpolation cost results in bivariate polynomials with Y-degree lower than five. This is especially true when the repeated interpolation and factorization method [AKS03a] is applied. For soft-decision decoding RS codes with a fixed cost C, the highest Y-degree of candidate bivariate polynomials, r, can be computed from the following formula [KV03a]:

$$r = \min\{t \in Z : (t+1)(\frac{t(k-1)}{2} + k) > C\}$$

For $C = 4580$ and $k = 419$ in the previous example of decoding the $(458, 410)$ RS code, we obtain $r = 4$. Actually, interpolation costs up to 6139 can be supported with this choice of r.

Now the applicability of the direct root computation method to bivariate

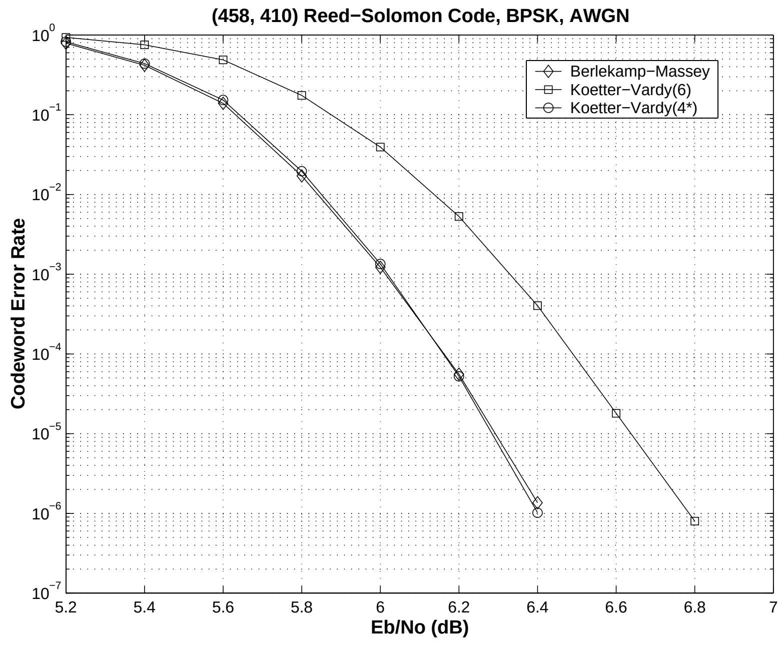

Figure 8.1: Decoding performance of a (458, 410) RS code over AWGN channel with BPSK modulation.

polynomial factorization in practical soft-decision decoding of RS code is established. Let us assume, through out the rest of this paper, that we work on a finite field of 2^p elements, i.e., $\mathbb{F}_{2^p}$. Chien search based root computation has a latency on the order of 2^p, which is very inefficient for solving low-degree polynomial equations in a large finite field. As will be shown later, the direct root computation can be implemented in hardware with a latency on the order of only $2p$. Another advantage of using the direct root finding method over the conventional exhaustive search method is that the order (multiplicity) of each root can be precisely determined. This leads to a factorization architecture where only roots of polynomials need to be routed to desired hardware resources. Compared to the architecture of [ZP05], where large number of MUXes are used to route both roots and polynomial coefficients, our new architecture is more area efficient.

The rest of the paper is organized as follows. In Section 8.2, we give some background information on the factorization procedure. Direct root computation methods for quadratic, cubic and quartic polynomials defined over $\mathbb{F}_{2^p}$ are presented in Section 8.3. An efficient VLSI architecture for direct root computation is also introduced there. An overall factorization architecture is given in Section 8.4. Section 8.5 gives an example of factorization architecture for decoding a $(458, 410)$ RS code. Conclusions are drawn in Section 8.6.

Throughout the rest of the paper, only factorization of bivariate polynomials with Y-degree lower than five are considered.

8.2 Factorization Algorithm and Fast Shift Transform

Currently, the most practical factorization algorithm was proposed by Roth and Ruckenstein [RR00] and it is repeated below.

Algorithm 4 ***The Roth-Ruckenstein factorization algorithm***

```
• Input: the bivariate polynomial $A(X,Y)$ and total number of factorization level $\Delta$.

• Initialization: iteration level $i = 0$

• Reconstruct $(A(X,Y), i)$
  {
  S1: find the largest integer l, such that $X^l | A(X,Y)$.
      $Q(X,Y) := A(X,Y)/X^l$
  S2: find all roots $\gamma_0^{(i)}, \gamma_1^{(i)}, ...$ of $Q(0,Y)$ in $\mathbb{F}_{2^p}$.
  if $i = \Delta - 1$, exit;
  else, for each root $\gamma_j^{(i)}$, do
      S3: $\hat{Q}(X,Y) = Q(X, Y + \gamma_j^{(i)})$.
      S4: $\tilde{Q}(X,Y) = \hat{Q}(X, XY) = Q(X, XY + \gamma_j^{(i)})$.
  S5: call Reconstruct$(\tilde{Q}(X,Y), i+1)$.
  }

• Output: all sequences $\{\gamma_j^{(0)}, \gamma_j^{(1)}, ..., \gamma_j^{(\delta-1)}\}$.
```

Let us assume that the bivariate polynomial $Q(X,Y)$ has a Y-degree equal to r and it can be expressed as $Q(X,Y) = \sum_{t=0}^{r} Y^t \sum_s q_{s,t} X^s$. The two major steps involved in the factorization algorithm given above are as follows:

1. Root Computation: Find all roots of $Q(0,Y)$ in $\mathbb{F}_{2^p}$, where the RS code is defined.

2. Polynomial Update: For each root γ of $Q(0,Y)$, compute $\hat{Q}(X,Y) = Q(X,Y+\gamma)$ and then $\tilde{Q}(X,Y) = \hat{Q}(X,XY)$ as follows:

$$\hat{q}_{s,t} = \sum_{t'=t}^{r} \binom{t'}{t} \gamma^{t'-t} q_{s,t'} \tag{8.1}$$

$$\tilde{q}_{s,t} = \hat{q}_{s-t,t} = \sum_{t'=t}^{r} \binom{t'}{t} \gamma^{t'-t} q_{s-t,t'} \tag{8.2}$$

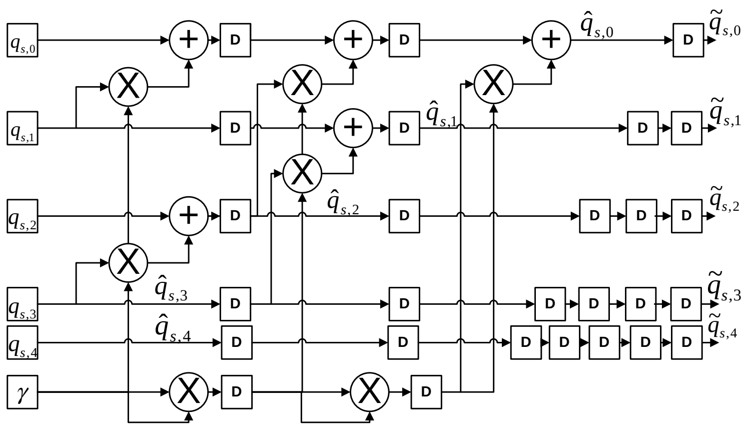

Figure 8.2: Hardware architecture for the fast shift transform.

Step 1 will be discussed in detail in Section 8.3. For Step 2, an efficient method, called FST (fast shift transform) is proposed in [AKS04b]. However, the hardware architecture given in [AKS04b] only carries out the operation defined by (8.1), while (8.2) is ignored. We present a complete FST architecture as shown in Fig. 8.2 for bivariate polynomials of Y-degree four. The incremental number of delay elements (registers) on the output lines are used to convert the coefficients of $\hat{Q}(X,Y)$ to those of $\tilde{Q}(X,Y)$. This can be illustrated with Fig. 8.3, where the contents of all registers in Fig. 8.2 of the FST architecture are shown for the initial seven clock cycles of the polynomial update step. (The registers represented by the darkened boxes store some intermediate results.) As one can see, the coefficients of $\tilde{Q}(X,Y)$ with the same Y degree always show up at the rightmost column of the registers simultaneously. Thus Step S1 of the factorization algorithm can be implemented by checking the contents of the registers in the rightmost column of Fig. 8.3. And the first non-zero column of coefficients are coefficients of $Q(0,Y)$ of the next iteration. As proved in [RR00], when γ is a root of order δ, it is always true that $\deg Q(0,Y) \leq \delta$ for the next iteration. Thus depending on the multiplicity of the root γ (at most four in our case), it may take from three to seven clock cycles for the first non-zero column to arise. We should emphasize that even if $\delta = 4$ for the root γ of a certain iteration, the degree of the corresponding $Q(0,Y)$ of the next iteration is still uncertain. Note that all the delay elements should be initialized to 0 at the beginning of each iteration.

Since only the $\tilde{q}_{l,t}$'s, for $t = 0, 1, ..., r$, are required in computing the roots for the next iteration level and all $\tilde{q}_{l,t}$'s are generated together with the FST architecture, we can start the root-finding process while the rest of the $\tilde{q}_{s,t}$'s, for $s > l$, are being computed. The index l is defined by Step S1 of the factorization algorithm. This concurrency in the root finding and FST processes can significantly reduce the latency associated with the root-finding procedure. Actually the latency contribution from the root computation step can be completely discounted, except for the very first iteration and for iterations when the polynomial update takes fewer clock cycles than the root computation process. The overlap in the root computation and polynomial update steps can be illustrated in Fig. 8.4. In the figure, the number δ represents the number of clock cycles the elapse before the coefficients are ready for the root computation process.

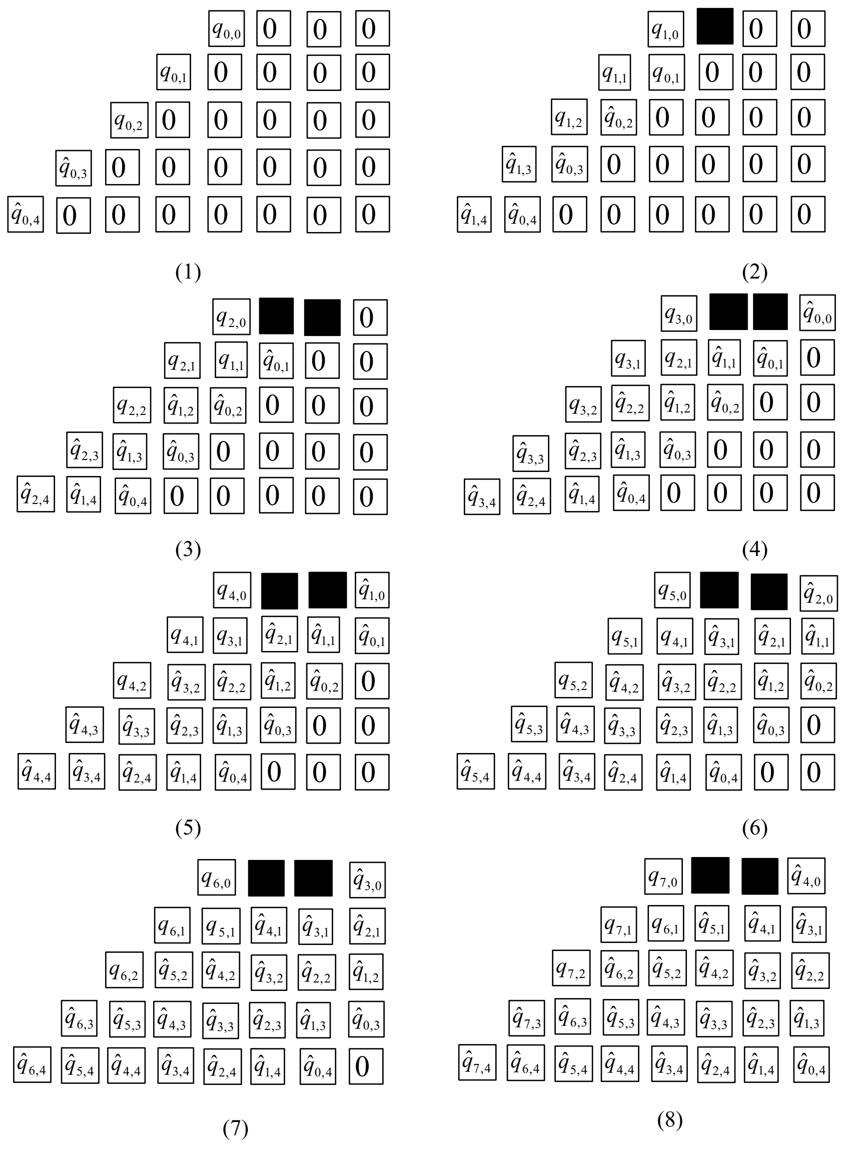

Figure 8.3: The values stored in the registers of the FST architecture during the first few clock cycles.

8.3 Direct Root Computation for Polynomials of Degree Lower than Five

A method for directly computing roots of affine polynomials over $\mathbb{F}_{2^p}$ is given in [Ber68]. To find roots for a non-affine polynomial, one can apply a

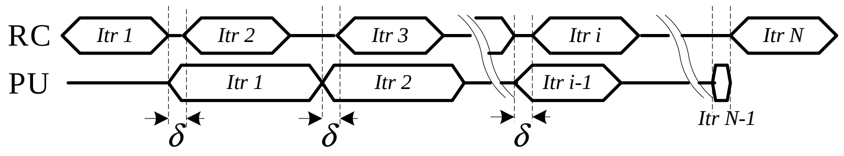

Figure 8.4: Timing diagram for the concurrent root computation and polynomial update steps.

transformation to convert the non-affine polynomial to an affine polynomial or derive the minimum affine multiple of the polynomial. The second approach is usually very complicated and may not have any advantage in complexity over the exhaustive search. Fortunately, for low degree(<5) polynomials, the transformation is sufficient. In this section, method and apparatus for solving polynomial equations of degree lower than five are described. We will show that the problem of finding roots of any cubic or quartic polynomial can be converted to the problem of finding roots of a quartic affine polynomial by polynomial transformation. This section is organized as follows: The direct root computation for affine polynomials is discussed in Subsection 8.3.1. In Subsection 8.3.2, the method and apparatus for solving simultaneous linear equations are presented. Transformation of general cubic and quartic polynomials to affine quartic polynomials is discussed in Subsection 8.3.3.

Any number $\beta \in \mathbb{F}_{2^p}$ can be expressed as $\beta = \sum_{j=0}^{p-1} \beta_j \alpha^j$, where α is the primitive element of the field. Thus the binary vector $\underline{\beta} = [\beta_0\ \beta_1\ ...\ \beta_{p-1}]$ can be used to represent the number β. Throughout this section, an underlined symbol denotes a binary vector and a double-underlined symbol denotes a matrix. In logic function expressions, “&” and “|” are used to represent binary AND and OR operations, respectively. A bar placed on top of a letter indicates a logic inversion.

8.3.1 Direct Root Computation for Affine Polynomials over F_{2^p}

A polynomial $f(Y)$, over $\mathbb{F}_{2^p}$, is said to be an affine polynomial iff $f(Y)$ can be expressed as $f(Y) = \sum_{j\geq 0} f_j Y^{2^j} + h$. For an element $y \in \mathbb{F}_{2^p}$, which can be represented with standard basis as $y = \sum_{i=0}^{p-1} y_i \alpha^i$, we have

$$\begin{aligned} f(y) &= \sum_{j\geq 0} f_j (\sum_{i=0}^{p-1} y_i \alpha^i)^{2^j} + h \\ &= \sum_{j\geq 0} f_j (\sum_{i=0}^{p-1} y_i (\alpha^i)^{2^j}) + h \\ &= \sum_{i=0}^{p-1} y_i \sum_{j\geq 0} f_j (\alpha^{2^j})^i + h. \end{aligned}$$

The second equality can be derived from a property of the linearized polynomial $\sum_{j\geq 0} f_j Y^{2^j}$. If we define $h_i \overset{\text{def}}{=} \sum_{j\geq 0} f_j (\alpha^{2^j})^i$ for $i = 0, 1, ..., p-1$, the above equation can be rewritten as $f(y) = \sum_{i=0}^{p-1} y_i h_i + h$. In other words, the roots of polynomial $f(Y)$ satisfy the following linear equation array

$$\underline{y} \begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ ... \\ \underline{h}_{p-1} \end{pmatrix} = \underline{h}, \tag{8.3}$$

where $\underline{h_i}$'s are the vector notation representation of h_i's defined above. Thus the problem of finding roots of an affine polynomial $f(Y)$ can be converted to the problem of solving p simultaneous linear equations as shown in (8.3). Since all α^{2^i}'s are known *a priori*, the $p \times p$ binary matrix can be constructed with the circuit shown in Fig. 8.5 once the coefficients f_i's are available.

8.3.2 Solving Binary Linear Equation Array

As discussed in the previous section, finding roots of an affine polynomial with coefficients in $\mathbb{F}_{2^p}$ is equivalent to solving the following linear equation array $\underline{y}\, \underline{\underline{M}} = \underline{z}$, where $\underline{y}$ and $\underline{z}$ are binary p-tuple row vectors and $\underline{\underline{M}}$ is a binary p-by-p matrix. An efficient algorithm for solving simultaneous linear equations has been derived by Berlekamp [Ber68]. In this subsection, we present a fast VLSI architecture to implement the algorithm.

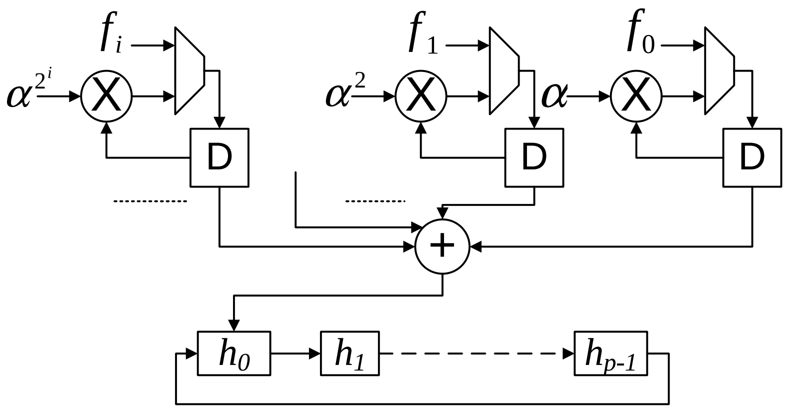

Figure 8.5: Circuit for constructing the $p \times p$ binary matrix for affine polynomials.

According to [Ber68], the matrix $\underline{\underline{M}}$ should be transformed to the reduced triangular idempotent (RTI) form. A p-by-p matrix is in the RTI form iff every entry below the main diagonal is 0, every entry on the main diagonal is either 0 or 1, and every entry in the same column as a main-diagonal 0 or the same row as a main-diagonal 1 is 0. It can be easily proved that such a matrix $\underline{\underline{\hat{M}}}$ has the property that $\underline{\underline{\hat{M}}}^2 = \underline{\underline{\hat{M}}}$. Berlekamp [Ber68] also gives the following theorems:

Theorem 8.1. *If $\underline{\underline{\hat{M}}}$ is in RTI form, then the row vector $\underline{y}$ is a solution to the equation $\underline{y}\ \underline{\underline{\hat{M}}} = \underline{0}$ iff $\underline{y}$ is a linear combination of rows of the matrix $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$, where $\underline{\underline{I}}$ is the identity matrix. Similarly, the row vector $\underline{y}$ is a solution to the equation $\underline{y}(\underline{\underline{\hat{M}}} - \underline{\underline{I}}) = \underline{0}$ iff $\underline{y}$ is a linear combination of rows of the matrix $\underline{\underline{\hat{M}}}$ and $\underline{y}$ is such a linear combination iff the product of each component of $\underline{y}$ and the corresponding diagonal component of $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$ is 0.*

Proof. The theorem can be proved by re-writing the property of the RTI-form matrix, i.e., $\underline{\underline{\hat{M}}}^2 = \underline{\underline{\hat{M}}}$, as $\underline{\underline{\hat{M}}}(\underline{\underline{\hat{M}}} - \underline{\underline{I}}) = \underline{0}$ or $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})\underline{\underline{\hat{M}}} = \underline{0}$. ∎

Theorem 8.2. *Any square matrix can be transformed to the RTI form by appropriate column operations.*

Theorem 8.2 can be proved in a constructive way by applying the matrix-reduction algorithm given in [Ber68] and deduction. The matrix-reduction algorithm is outlined as follows:

Algorithm 5 ***The matrix-reduction algorithm*** *For an p-by-p binary matrix, the procedure consists of repeating the following three steps p times:*

1. *If the topmost leftmost component is 1, do nothing. Otherwise, exchange the leftmost column with the leftmost of columns whose top component is 1 and whose main-diagonal component is 0. If no such column exists, exchange the leftmost column with the leftmost of columns whose top component is 1 and whose main-diagonal component is 1. If there is no column whose top component is 1, do nothing. We refer the column chosen to exchange with the leftmost column the pivot column.*

2. *Zero the top component of all columns whose top components is 1, except the leftmost column, by adding (modulo 2) the leftmost column to these columns.*

3. *Circularly shift the rows upward and circularly shift the columns leftward.*

Berlekamp [Ber68] gives the following lemma without a proof. In the following, the lemma and a complete proof is presented.

Lemma 8.3 *After the three steps listed above have been performed k times, the first $p-k$ columns contain 0's in their bottom k rows, and the lower right k-by-k matrix is in RTI form.*

Proof. The proof is carried out by induction on k. For $k=1$, after Step 2, the top row either has all 0's or its leftmost component is 1 with the rest equal to 0. After Step 3, the bottom row either has all 0's or its rightmost component is 1 with the rest equal to 0. Thus the condition is satisfied. Now let us assume that the condition is satisfied for $k=l>1$ and we use $\underline{\underline{M}}^{(l)}$ to represent the matrix after l rounds of three-step operations defined above. When $k=l+1$, we consider the following three cases. Case 1, assume the top row is an all-zero row. In this case, after Step 3, the top l rows of the lower left $(l+1)$-by-$(p-l-1)$ matrix are all 0's since the components come from the lower left l-by-$(p-l)$ matrix after the l rounds of operations. And this matrix is an all-zero matrix from our assumption. The last row of the lower left $(l+1)$-by-$(p-l-1)$ matrix are all 0's since the components come from the top row, which is assumed to be all 0's. For the lower right $(l+1)$-by-$(l+1)$ matrix, its upper left l-by-l sub-matrix comes from the lower right l-by-l RTI-form matrix in $\underline{\underline{M}}^{(l)}$ and components in its bottom row and rightmost column are all 0's since they come from the top row and the l bottom components of the leftmost column of $\underline{\underline{M}}^{(l)}$. Thus by definition of the RTI form, the lower right (l+1)-by-(l+1) matrix is still in the RTI form. Case 2, let us assume that either the topmost and leftmost component of $\underline{\underline{M}}^{(l)}$ is 1 or the column chosen to exchange with the leftmost column is one of the $(p-l)$ leftmost columns. In either case, after Step 2, the top row of $\underline{\underline{M}}^{(l)}$ becomes all 0's except the leftmost component of the top row and the lower l rows of $\underline{\underline{M}}^{(l)}$ are intact since the bottom l rows of the $(p-l)$ leftmost columns of $\underline{\underline{M}}^{(l)}$ are all 0's. After the circular row and column shifts of Step 3, the only difference from Case 1 is that a 1 shows up in the bottommost and rightmost component. This still makes the lower right $(l+1)$-by-$(l+1)$ matrix RTI form by definition. Case 3, assume that in Step 1 the leftmost column needs to be exchanged with a column among the l rightmost columns. If the main-diagonal component of the pivot column is 1, then after exchange, the corresponding column in the lower right l-by-l matrix is replaced with an all-zero column from the bottom l components of the leftmost column of $\underline{\underline{M}}^{(l)}$. Then in Step 2, only columns to the right of the original pivot column may be added by the leftmost column and all such columns have a 1 as their main-diagonal component since otherwise, the pivot column would not have been chosen to be that one in first place! Thus after Step 2, the lower right l-by-l matrix is still in the RTI form. The lower l components of the leftmost column are not all-zero but after the shifts, they become the top l components in the rightmost column of the lower right $(l+1)$-by-$(l+1)$ matrix and all conditions of the lemma are satisfied. If the main-diagonal component of the pivot column is 0, the lower l components of the pivot column are all 0's according to the definition of the RTI form. This is no different from Case 2. Thus we conclude that after the $(l+1)$ rounds of three-step operations, the conditions in the lemma are still satisfied. ∎

Therefore, Theorem 8.2 can be proved with a combination of Algorithm 5 and Lemma 8.3. The equation $\underline{y}\ \underline{\underline{M}} = \underline{z}$ for $\underline{z} \neq 0$ can be augmented as $[\underline{y}\ 1] \begin{bmatrix} \underline{\underline{M}} \\ \underline{z} \end{bmatrix} = \underline{0}$. Let us assume that p-by-p matrix $\underline{\underline{M}}$ is transformed to a p-by-p matrix $\underline{\underline{\hat{M}}}$ of the RTI form and the same column operations are applied to row vector $\underline{z}$ to transform it to $\underline{\hat{z}}$. Solutions to the original equation are equivalent to solutions of the transformed equation $[\underline{y}\ 1] \begin{bmatrix} \underline{\underline{\hat{M}}} \\ \underline{\hat{z}} \end{bmatrix} = \underline{0}$, or $\underline{y}\ \underline{\underline{\hat{M}}} = \underline{\hat{z}}$. If there exists a non-zero element in $\underline{\hat{z}}$, say z_i, such that the corresponding column (the i^{th} column) of $\underline{\underline{\hat{M}}}$ is an all-zero column, we have the following equation:

$$y_0 0 + y_1 0 + ... + y_{p-1} 0 + z_i = 0.$$

This equation is a contradiction, thus neither the original equation array nor the transformed equation array has a solution. Otherwise, every all-zero column in $\underline{\underline{\hat{M}}}$ corresponds to a zero element in $\underline{\hat{z}}$. This translates to the fact that the product of each component of $\underline{\hat{z}}$ and the corresponding main-diagonal component of $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$ is 0, which, according to Theorem 8.1, infers that $\underline{\hat{z}}$ is a linear combination of the rows of matrix $\underline{\underline{\hat{M}}}$. From the same theorem, we also get $\underline{\hat{z}} = \underline{\hat{z}}\ \underline{\underline{\hat{M}}}$, thus $\underline{y} = \underline{\hat{z}}$ is a solution to equation $\underline{y}\ \underline{\underline{\hat{M}}} = \underline{\hat{z}}$. We also know that solutions to the equation $\underline{y}\ \underline{\underline{\hat{M}}} = \underline{\hat{z}}$ also include $\underline{y} = \underline{\hat{z}} + \underline{y}'$, where $\underline{y}'$ is a solution to equation $\underline{y}\ \underline{\underline{\hat{M}}} = \underline{0}$.

In summary, solving equation $\underline{y}\ \underline{\underline{M}} = \underline{z}$ for $\underline{z} \neq 0$ consists of three steps.

1. Apply Algorithm 5 to transform $\underline{\underline{M}}$ to $\underline{\underline{\hat{M}}}$ of the RTI form. Same column operations are applied simultaneously to row vector $\underline{z}$ to convert it to $\underline{\hat{z}}$.

2. Form matrix $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$.

3. Check whether the products of all components of $\underline{\hat{z}}$ and the corresponding main-diagonal component of $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$ are 0. If not, declare "no solution". Otherwise, solutions consist of $\underline{\hat{z}}$ plus any linear combination of the rows of matrix $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$.

The most computation-intensive step in the procedure given above is the matrix reduction as described by Algorithm 5. It turns out that a key part of the algorithm is to decide which column to exchange with the leftmost column and this can be done with the method given in [Ber68]. To facilitate the description of the algorithm and its circuit implementation, we introduce the following notation. Let $\{i : i = 0, 1, ..., p-1\}$ denote the indices of the columns of the matrix, where the leftmost column has index 0, and define the following three sets of variables:

- $\{D_i\}$: the diagonal element on the ith column.
- $\{T_i\}$: the ith leftmost component on the top row.
- $\{E_i\}$: whether the ith column should be exchanged with the leftmost column (the 0th column).

In [Ber68], the E_i's are computed from the D_i's and T_i's. First, intermediate variables A_i's, for $i = -1, 0, ..., 2p-1$, are introduced. A_i', for $i = 0, 1, ..., p-1$, are set to 0 if no column with index less than or equal to i contains a 1 in the top position and a 0 in the diagonal position . If the ith column has a 1 in the top component and a 0 in the diagonal component, A_i is set to 1; otherwise, A_i is set to be the same as A_{i-1}. A_i, for $i = p, p+1, ..., 2p-1$, is set to 0 iff all $A_0 = A_1 = ... = A_{p-1} = 0$ and no column with indices less than or equal to $i-p$ has a 1 in the top position. For $i = p, p+1, ..., 2p-1$, if column $i-p$ has a 1 as its top component, A_i is set to 1; otherwise, A_i is set to be the same as A_{i-1}. Thus the A_i's can be computed with the following formula:

$$\begin{cases} A_{-1} = 0 & \text{initialization} \\ A_i = (T_i \& \bar{D}_i) | A_{i-1} & \text{if } i = 0, 1, ..., p-1 \\ A_i = T_{i-p} | A_{i-1} & \text{if } i = p, p+1, ..., 2p-1 \end{cases} \tag{8.4}$$

For the special case of $i = 0$, we define $D_0 = 0$, thus we have $T_0 \& \bar{D}_0 = T_0$, and the E_i's can be generated by the following formula

$$E_i = (T_i \& \bar{A}_{i+p-1}) | ((T_i \& \bar{D}_i) \& \bar{A}_{i-1}), \text{for } i = 0, 1, ...p-1.$$

It is easy to verify that at most one of the E_i's can be set as 1. The circuit [Ber68] shown in Fig. 8.6 and Fig. 8.7 can be used to compute all E_i's and decide which column to exchange with the leftmost column. The circuit, though straightforward, has a long critical path delay. As can be seen from Fig. 8.6 and Fig. 8.7, the longest path consists of $2p$ OR gates, two AND gates and two inverters.

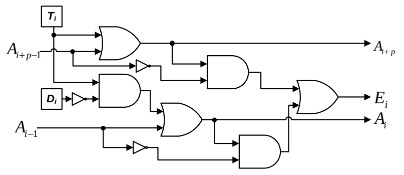

Figure 8.6: Logic circuit design of one cell.

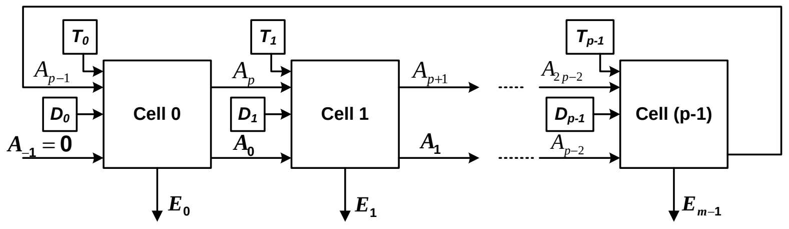

Figure 8.7: Logic circuitry for deciding which column of the binary matrix to exchange with the leftmost column.

The long critical path delay results from the recursive computation of A_i's as given in (8.4). In the rest of the subsection, we present a faster implementation that significantly reduces the critical path delay of the design presented in [Ber68]. First, let us define $S_i = T_i \& \bar{D}_i$, $V_i = T_i \& \bar{S}_i = T_i \& D_i$ for $i = 0, 1, ..., p-1$. Instead of letting the A_i's ripple across the longest path, we choose to compute them all in parallel as follows:

$$A_0 = S_0$$

$$A_1 = S_0 | S_1$$

$$\cdots$$

$$A_{p-1} = S_0|S_1|...|S_{p-1}$$

$$A_p = T_0|S_1|S_2|...|S_{p-1}$$

$$A_{p+1} = T_0|T_1|S_2|S_3|...|S_{p-1}$$

$$...$$

$$A_{2p-1} = T_0|T_1|...|T_{p-1}.$$

Thus the E_i's can be computed as follows:

$$E_0 = (T_0\&\bar{A}_{p-1})|(S_0\&\bar{A}_{-1})$$

$$\begin{aligned} E_1 &= (T_1\&\bar{A}_p)|(S_1\&\bar{A}_0) \\ &= (T_1\&\bar{T}_0\&\bar{S}_1\&...\&\bar{S}_{p-1})|(S_1\&\bar{S}_0) \\ &= (\bar{T}_0\&V_1\&\bar{S}_2\&...\&\bar{S}_7)|(S_1\&\bar{S}_0) \end{aligned}$$

$$...$$

$$\begin{aligned} E_{p-1} &= (T_{p-1}\&\bar{A}_{2p-2})|(S_{p-1}\&\bar{A}_{p-2}) \\ &= (T_{p-1}\&\bar{T}_0\&\bar{T}_1\&...\&\bar{T}_{p-2}\&\bar{S}_{p-1}) \\ &\quad |(S_{p-1}\&\bar{S}_0\&\bar{S}_1\&...\&\bar{S}_{p-2}) \\ &= (\bar{T}_0\&\bar{T}_1\&...\&\bar{T}_{p-2}\&V_{p-1}) \\ &\quad |(S_{p-1}\&\bar{S}_0\&\bar{S}_1\&...\&\bar{S}_{p-2}). \end{aligned}$$

It is easy to see that for the special case of $i = 0$, we have $E_0 = T_0$. Thus the E_i's can be generated from the circuitry shown in Fig. 8.8 and Fig. 8.9. The AND array in Fig. 8.9 is made of binary-tree type AND gates of at most $\lceil log_2 p\rceil$ levels. The longest path in this implementation consists of one inverter, one OR gate, and $\lceil log_2 p\rceil + 1$ AND gates, which is much shorter than the one in Fig. 8.7, especially for large p. The required hardware can be estimated as follows: two inverters, two AND gates and one OR gate are required for one logic cell shown in Fig. 8.8, which amounts to $2p$ inverters, $2p$ AND gates and p OR gates for p such logic cells. The number of AND gates required for the binary-tree type AND gate array is equal to $(p-1)^2 + 0.5p(p-1) = 1.5p^2 - 2.5p + 1$. In addition, $(p-1)$ OR gates have to be used. In summary, to compute all E_i's, a total of $(1.5p^2 - 0.5p + 1)$ AND gates, $(2p-1)$ OR gates and $2p$ inverters are needed.

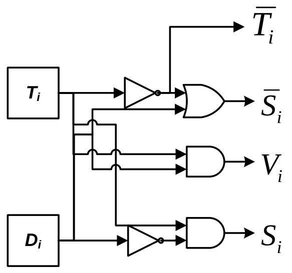

Figure 8.8: Logic circuit design of one cell.

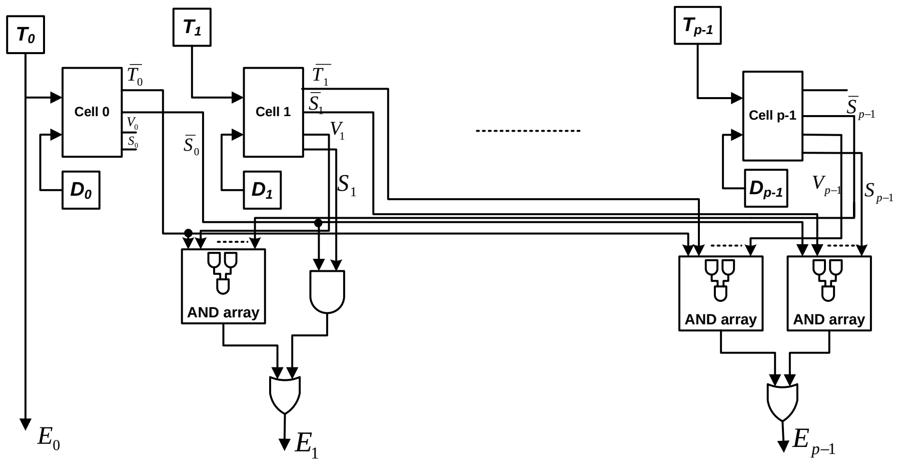

Figure 8.9: Logic circuit for deciding which column of the binary matrix to exchange with the leftmost column.

After all the E_i's are determined by the logic circuit discussed above, the column exchange operation and the other two steps of Algorithm 5 can be implemented with the circuit shown in Fig. 8.10, where the Steps 2 and 3 in Algorithm 5 are carried out in one clock cycle. These two steps are combined because each of them has a shorter critical path delay than Step 1 of Algorithm 5. Combining these two steps not only makes the design more balanced, in terms of

Table 8.1: Hardware unit counts for implementation of the matrix reduction algorithm

Unit Type	Inverter	AND	OR
Count	$2p$	$3.5p^2 - 1.5p$	$p^2 + 2p - 2$
Unit Type	XOR	MUX	Register
Count	$p^2 - p$	$2p^2 + p - 1$	$p^2 + p$

critical path delay, but it also reduces the total number of clock cycles required for the matrix reduction algorithm by 33%. In the figure, only the ith row of the $(p+1) \times p$ register matrix is shown and the rest of the circuitry is similar. The input to each register in the figure is controlled by a 2:1 MUX, corresponding to the two clock cycles required for each iteration. Astute readers might have realized that for the pth row, all registers, except the rightmost one will be loaded with 0's at the end of the second clock cycle during each iteration. This results from Step 2 of the algorithm. Thus we can save $p - 1$ copies of AND and XOR gates used to generate input to the 2:1 MUX during the second clock cycle. A careful counting shows that the circuit in Fig. 8.10 requires a total of $(p+1)(p-1) + p(p-1) = 2p^2 - p - 1$ AND gates, $(p+1)(p-1) = p^2 - 1$ OR gates, $p(p-1) = p^2 - p$ XOR gates and $(p+1)p + (p+1)(p-1) = 2p^2 + p - 1$ MUXes. Combined with the logic required to compute all E_i's and the $(p+1)p$ registers used to store the matrix, an estimate of the number and type of hardware units required to implement the matrix reduction algorithm is given in Table 8.1.

From Figures 8.8, 8.9 and 8.10, we can see that the critical path of the entire matrix reduction circuitry consists of one 2:1 MUX, $\lceil log_2 p \rceil + 2$ AND gates, $\lceil log_2 p \rceil + 1$ OR gates, and one inverter.

At the end of the matrix reduction procedure, the solutions are embedded in the resulting matrix $\underline{\underline{\hat{M}}} - \underline{\underline{I}}$ and vector $\hat{z}$. Since the quartic (or lower degree) affine polynomial has at most four distinct roots, at most two out of the p rows of matrix $\underline{\underline{\hat{M}}} - \underline{\underline{I}}$ can be non-zero. However, the exact locations of these two rows are unknown. If we use z_0 and z_1 to denote the two numbers represented by the two non-zero rows, the architecture shown in Fig. 8.11 can be used to

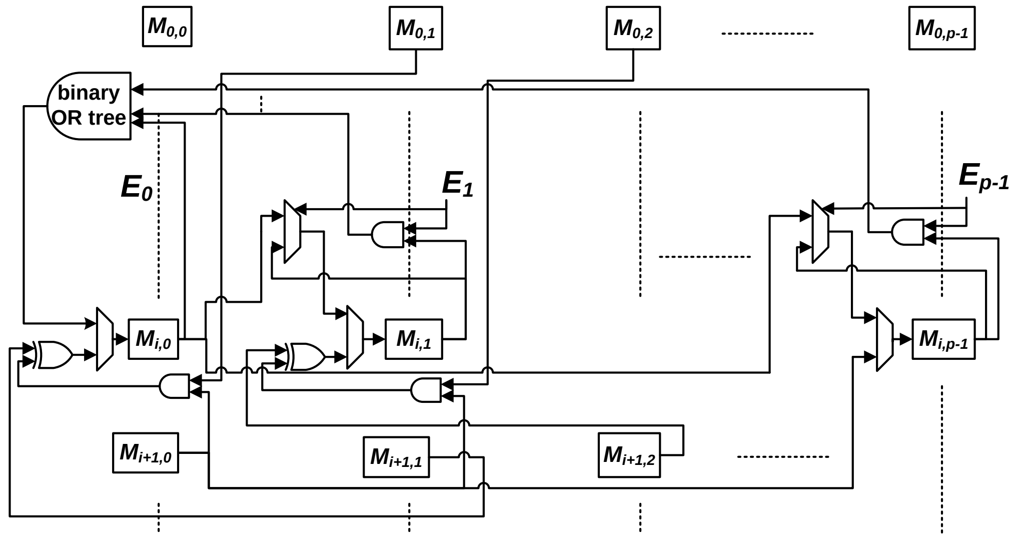

Figure 8.10: Architecture for the two-step matrix operations.

"extract" z_0 and z_1 from the matrix. In the figure, M_i's denote the F_{2^p} numbers represented by rows of the matrix $\underline{\underline{\hat{M}}} - \underline{\underline{I}}$ and B_i's are binary variables such that

$$B_i = \begin{cases} 1, & \text{if } M_i \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Note that z_0 and z_1 can be distinct or identical, and both of them can be equal to 0, depending on the coefficients of the polynomial equation. Various outcomes of z_0 and z_1 will be exploited in Subsection 8.3.4 to determine the root conditions, such as total number of distinct roots and orders of each root found.

8.3.3 The Linear, Quadratic, Cubic and Quartic Polynomials

In this subsection, techniques described in earlier subsections are applied to root computation for polynomials of degree lower than five. For a linear polynomial, finding its root only takes a division operation. The quadratic polynomial $f(Y) = aY^2 + bY + c$, where $a, b, c \in \mathbb{F}_{2^p}$, is affine in nature. Let us represent root y of $f(Y) = 0$ as $y = \sum_{i=0}^{p-1} y_i \alpha^i$. From the property of linearized polynomial,

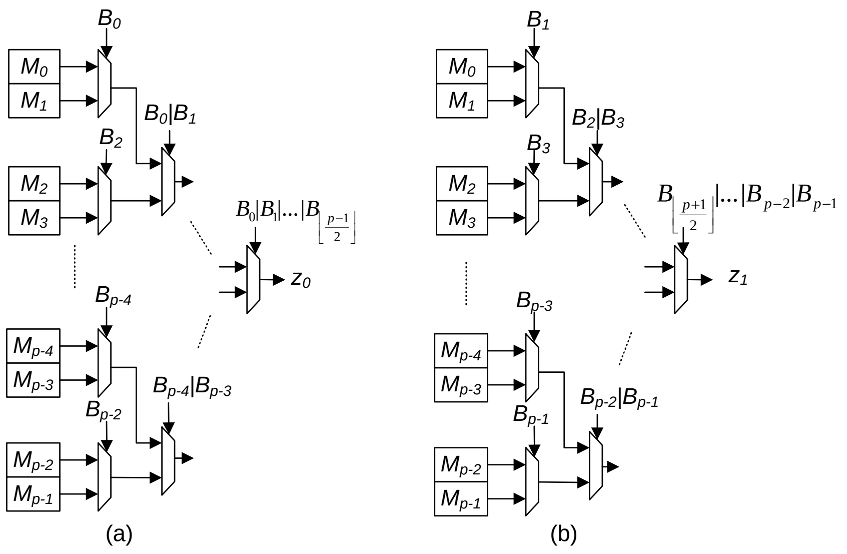

Figure 8.11: Architecture for computing roots from the reduced triangular idempotent matrix.

one can write

$$ay^2 + by = \sum_{i=0}^{p-1} y_i(a\alpha^{2i} + b\alpha^i).$$

Let $h_i = a\alpha^{2i} + b\alpha^i$ for $i = 0, 1, ..., p-1$, then equation $ay^2 + by = c$ can be re-written as

$$\underline{y}\begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ ... \\ \underline{h}_{p-1} \end{pmatrix} = \underline{c},$$

where $\begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ ... \\ \underline{h}_{p-1} \end{pmatrix}$ is a p by p matrix. All α^{2i}'s and α^i's can be pre-computed, once a and b are known, the h_i's forming the matrix can be generated, thus finding root translates into solving the linear binary equation array with p unknowns above.

For a cubic polynomial $f(Y) = f_3Y^3 + f_2Y^2 + f_1Y + f_0$, $f(Y) = 0$ can be re-written as $Y^3 + \frac{f_2}{f_3}Y^2 + \frac{f_1}{f_3}Y + \frac{f_0}{f_3} = 0$. This can be further modified as $f_3^2(Y + \frac{f_2}{f_3})^3 + (f_2^2 + f_1 f_3)(Y + \frac{f_2}{f_3}) + f_1 f_2 + f_0 f_3 = 0$. A replacement of variable $Z = Y + \frac{f_2}{f_3}$ and defining $a \overset{\text{def}}{=} f_3^2$, $b \overset{\text{def}}{=} f_2^2 + f_1 f_3$ and $c \overset{\text{def}}{=} f_1 f_2 + f_0 f_3$ lead to

$$aZ^3 + bZ + c = 0.$$

The computation of a, b and c can be achieved with the circuit shown in Fig. 8.12.

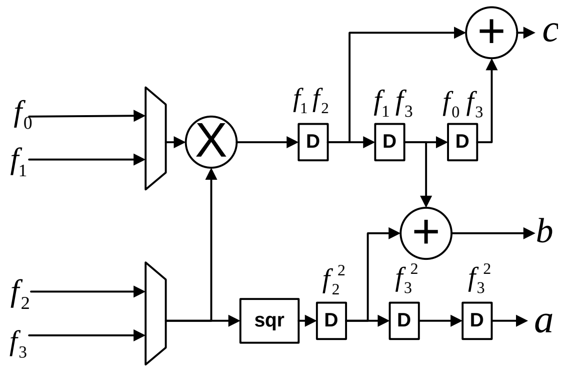

Figure 8.12: Architecture for transforming a general cubic polynomial to an affine quartic polynomial.

Multiplying both sides by Z in the above equation, we get

$$aZ^4 + bZ^2 + cZ = 0.$$

This is a linearized polynomial and from its property, its root $z = \sum_{i=0}^{p-1} z_i \alpha^i$ satisfies

$$az^4 + bz^2 + cz = \sum_{i=0}^{p-1} z_i (a\alpha^{4i} + b\alpha^{2i} + c\alpha^i)$$

In matrix form, defining $h_i = a\alpha^{4i} + b\,\alpha^{2i} + c\alpha^i$ for $i = 0, 1, ..., p-1$, the roots of $aZ^4 + bZ^2 + cZ = 0$ satisfy the following equation

$$\underline{z} \begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ \dots \\ \underline{h}_{p-1} \end{pmatrix} = \underline{0}$$

All α^{4i}'s, α^{2i}'s and α^{i}'s can be pre-computed, once a, b and c are computed from the coefficients f_i's, the h_i's that form the matrix can be generated, thus finding root z translates into solving the linear binary equation array with p unknowns. Once a root z is found, it can be shifted to get root y for the original cubic polynomial.

Even though a method and apparatus to convert a general cubic polynomial to an affine polynomial is given here, we will show later that cubic polynomial can be handled with the same hardware resource that transforms a general quartic polynomial to an affine polynomial.

In the case of a quartic polynomial $f(Y) = f_4Y^4 + f_3Y^3 + f_2Y^2 + f_1Y + f_0$, if $f_3 \neq 0$, $f(Y) = 0$ can be re-written as

$$Y^4 + \frac{f_3}{f_4}(Y^3 + \frac{f_2}{f_3}Y^2 + \frac{f_1}{f_3}Y + \frac{f_0}{f_3}) = 0.$$

The following "shift" transformation can be applied:

$$\begin{aligned}
& Y^4 + \frac{f_3}{f_4}(Y^3 + \frac{f_2}{f_3}Y^2 + \frac{f_1}{f_3}Y + \frac{f_0}{f_3}) \\
= \quad & Y^4 + \frac{f_3}{f_4}((Y + \sqrt{\frac{f_1}{f_3}})^3 + (\sqrt{\frac{f_1}{f_3}} + \frac{f_2}{f_3})Y^2 + (\sqrt{\frac{f_1}{f_3}})^3 + \frac{f_0}{f_3}) \\
= \quad & Y^4 + \frac{f_3}{f_4}\left((Y + \sqrt{\frac{f_1}{f_3}})^3 + (\sqrt{\frac{f_1}{f_3}} + \frac{f_2}{f_3})(Y + \sqrt{\frac{f_1}{f_3}})^2 + \frac{f_2f_1}{f_3^2} + \frac{f_0}{f_3}\right) \\
= \quad & (Y + \sqrt{\frac{f_1}{f_3}})^4 + \frac{f_3}{f_4}(Y + \sqrt{\frac{f_1}{f_3}})^3 + \frac{f_3}{f_4}(\sqrt{\frac{f_1}{f_3}} + \frac{f_2}{f_3})(Y + \sqrt{\frac{f_1}{f_3}})^2 \\
+ \quad & \frac{f_3}{f_4}\frac{f_2f_1}{f_3^2} + \frac{f_0}{f_4} + (\frac{f_1}{f_3})^2
\end{aligned}$$

Note that $\sqrt{\frac{f_1}{f_3}}$ always exists in the same field that the polynomial is defined on due to the following reason. Let $h = \frac{f_1}{f_3} = \alpha^t$, where α is the primitive element that defines the field. If t is even, than $\sqrt{h} = \alpha^{t/2}$; otherwise, $\alpha^t = \alpha^{(t+2^p-1)}$ and $(t + 2^p - 1)$ is an even number, thus $\sqrt{h} = \alpha^{(t+2^p-1)/2}$. We now define $a \overset{\text{def}}{=} f_3$, $b \overset{\text{def}}{=} (\sqrt{f_1f_3} + f_2)$, $c \overset{\text{def}}{=} \frac{f_2f_1}{f_3} + f_0 + (\frac{f_1}{f_3})^2 f_4$, and define the variable substitution $Z \overset{\text{def}}{=} \frac{1}{Y+\sqrt{\frac{f_1}{f_3}}}$. Note that the variable substitution through shifting and reciprocation is valid only if $y = \sqrt{\frac{f_1}{f_3}}$ is not a root of polynomial $f(Y)$, or equivalently $c \neq 0$. In this case, finding roots y of $f(Y) = 0$ is equivalent to finding roots z of polynomial $g(Z) = cZ^4 + bZ^2 + aZ + f_4 = 0$ followed by reciprocation and shifting operations. On the other hand, if $c = 0$, then $y = \sqrt{\frac{f_1}{f_3}}$ is a root of the original polynomial, and this root cannot be found by solving $g(Z) = 0$. In this case, root $y = \sqrt{\frac{f_1}{f_3}}$ has a multiplicity of at least two. It may have a multiplicity of three if $b = 0$ and $a \neq 0$, and it may have a multiplicity of four if $b = 0$ and $a = 0$. In this case, roots other than $y = \sqrt{\frac{f_1}{f_3}}$, if exist, can still be derived from solutions of $g(Z) = 0$.

Polynomial $cZ^4 + bZ^2 + aZ + f_4 = 0$ is an affine polynomial and its root $z = \sum_{i=0}^{p-1} z_i\alpha^i$ satisfies

$$cz^4 + bz^2 + az = \sum_{i=0}^{p-1} z_i(c\alpha^{4i} + b\alpha^{2i} + a\alpha^i) = f_4$$

In matrix form, defining $h_i = c\alpha^{4i} + b\alpha^{2i} + a\alpha^i$ for $i = 0, 1, ..., p-1$, the roots of equation $cZ^4 + bZ^2 + aZ = 1$ satisfy the following equation

$$\underline{z} \begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ ... \\ \underline{h}_{p-1} \end{pmatrix} = \underline{f}_4$$

All α^{4i}'s, α^{2i}'s and α^i's can be pre-computed, once a , b and c are computed from the coefficients f_i's, the h_i's that form the matrix can be generated, thus finding root z translates into solving the linear binary equation array with p unknowns above. Once a root z is found, it can be inverted and shifted to obtain root y for the original quartic polynomial.

Note that the above transform is necessary only if $f_3 \neq 0$. If $f_3 = 0$, i.e., the polynomial equation to be solved is $f_4Y^4 + f_2Y^2 + f_1Y + f_0 = 0$, which is an affine polynomial in the first place. In matrix form, defining $h_i = f_4\alpha^{4i} + f_2\alpha^{2i} + f_1\alpha^i$ for $i = 0, 1, ..., p-1$, the roots $y = \sum_{i=0}^{p-1} y_i\alpha^i$ of the affine polynomial satisfy:

$$\underline{y}\begin{pmatrix} \underline{h}_0 \\ \underline{h}_1 \\ \ldots \\ \underline{h}_{p-1} \end{pmatrix} = \underline{f_0}.$$

The transform from general quartic to affine quartic polynomials can be performed by the architecture given in Fig. 8.13, which is highly optimized for area and latency. As shown by the timing diagram in Fig. 8.14, it takes only five clock cycles for the architecture to complete the transform. The hardware complexity is reduced to the minimum, as only one multiplier and one inverter are used. The multiplier is time-shared among several multiplications required by the transform, which are arranged in such an optimal way that no de-multiplexer is required at the multiplier's output and that the multiplier has an 80% utilization efficiency. The inputs to the multiplier are selected by two 4:1 MUXes, which can be controlled by the same set of signal.

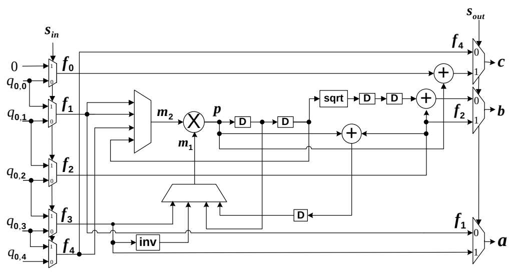

Figure 8.13: Architecture for transforming a general quartic polynomial to an affine quartic polynomial.

To optimize resource utilization, the quartic polynomial equation solver can be configured to solve cubic polynomial equations. The MUXes at the input and output of Fig. 8.13 serves this purpose. As will be shown in Section 8.4, there is no need for a separate cubic polynomial equation solver. The control signals s_{in} and s_{out} are defined as follows:

$$s_{in} = \begin{cases} 1, & \text{if } \deg Q(0,Y) = 3; \\ 0, & \text{otherwise.} \end{cases} \qquad s_{out} = \begin{cases} 1, & \text{if } f_3 == 0; \\ 0, & \text{otherwise.} \end{cases}$$

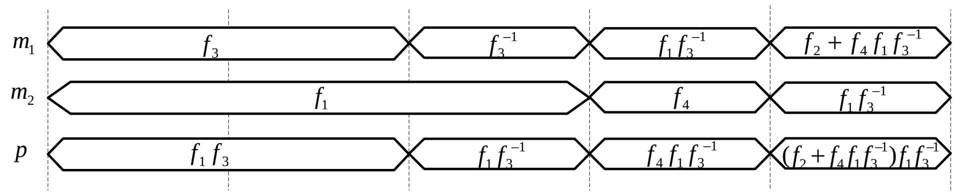

Figure 8.14: Timing diagram for transforming a general quartic polynomial to an affine quartic polynomial.

8.3.4 Root Order Determination

The orders of the roots found at a certain iteration level of the factorization algorithm are key to resource allocation and scheduling of the next iteration level. In [ZP05], the order of each root found at a certain iteration is predicted based on the statistics collected from simulations. Polynomial and root scheduling for succeeding iterations are made accordingly. The prediction, though accurate most of the time, has a non-zero failure rate. Thus a mechanism to check the correctness of the prediction has to be applied. Once a prediction fails, an exhaustive root search has to be carried out and hardware resources need to be reallocated, which leads to longer latency for the factorization procedure. In addition, a large number of MUXes are needed to route polynomial coefficients to different polynomial update engines. This issue can be completely circumvented by using the direct root computation since the order of the roots found by direct computation can be precisely determined. To illustrate this for a quartic polynomial, we start with the following theorem.

Theorem 8.4 *The roots of a quartic affine polynomial of the following form $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = 0$, where $\mu_0, \mu_1, \mu_2 \in \mathbb{F}_{2^p}$, have the following properties:*

- *The polynomial can have no root, one single root, two distinct roots, or four distinct roots in $\mathbb{F}_{2^p}$.*
- *If the polynomial only has one root, the root can only be of multiplicity one or four.*

- *If the polynomial has two distinct roots, either both roots are of multiplicity one or both roots have a multiplicity of two.*

Proof. From Section 8.3.2, if there is no solution to the binary linear equation array, the polynomial $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = 0$ does not have any root in $\mathbb{F}_{2^p}$. On the other hand, if a solution exists for $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = 0$, matrix $(\underline{\underline{\hat{M}}} - \underline{\underline{I}})$ can have zero, one or two non-zero rows. These correspond to the one-root, two-root and four-root cases, respectively. In other words, it is impossible for the equation to only have three distinct roots in $\mathbb{F}_{2^p}$.

If only one root, $\gamma \in \mathbb{F}_{2^p}$, is found, γ can certainly be a root of multiplicity four, and a necessary condition is that $\mu_1 = \mu_2 = 0$. Let us now assume that γ has a multiplicity of three. Thus $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0$ can be factorized, in a larger field, as $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = (X + \gamma)^3 (X + \eta)$ where $\eta \notin \mathbb{F}_{2^p}$. However, we then have $\eta = \frac{\mu_0}{\gamma^3} \in \mathbb{F}_{2^p}$, which contradicts our assumption. Thus the single root γ cannot have a multiplicity of three. If γ has a multiplicity of two, then we have $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = (X + \gamma)^2 (X^2 + \eta_1 X + \eta_0)$ where $\eta_0, \eta_1 \in \mathbb{F}_{2^p}$ and $X^2 + \eta_1 X + \eta_0$ can not be further factorized in $\mathbb{F}_{2^p}$. Apparently, $\eta_1 \neq 0$ otherwise the square root of η_0 always exists in $\mathbb{F}_{2^p}$ and $X^2 + \eta_1 X + \eta_0$ can be further factorized. Then it can be derived that

$$\begin{aligned} & X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = (X^2 + \gamma^2)(X^2 + \eta_1 X + \eta_0) \\ = \; & X^4 + \eta_1 X^3 + (\gamma^2 + \eta_0) X^2 + \gamma^2 \eta_1 X + \gamma^2 \eta_0 \end{aligned}$$

and a contradiction occurs since $\eta_1 \neq 0$ in the right hand side (RHS) but there is no cubic term in the left hand side (LHS) of the equation above. At last, if a single root γ exists and either $\mu_1 \neq 0$ or $\mu_2 \neq 0$, γ can only have a multiplicity equal to one.

Now let us consider the case when two distinct roots γ_1 and γ_2 are found. If one of them, say γ_1, has a multiplicity of three and the other is of multiplicity one, we have

$$\begin{aligned} & X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = (X + \gamma_1)^3 (X + \gamma_2) \\ = \; & X^4 + (\gamma_1 + \gamma_2) X^3 + (\gamma_1^2 + \gamma_1 \gamma_2) X^2 + (\gamma_1^3 + \gamma_1^2 \gamma_2) X + \gamma_1^3 \gamma_2 \end{aligned}$$

Since the LFS of the equation above does not have any cubic term, we must have $\gamma_1 = \gamma_2$ in the field of characteristic two, which contradicts the fact that $\gamma_1 \neq \gamma_2$. Now if both roots are of multiplicity two, it can be easily derived that a necessary condition for this to happen is that $\mu_1 = 0$. In addition, if one of the roots, say γ_1, has a multiplicity of two and the other root γ_2 has a multiplicity equal to one, we must have the following factorization: $X^4 + \mu_2 X^2 + \mu_1 X + \mu_0 = (X+\gamma_1)^2(X+\gamma_2)(X+\eta)$, where $\eta \notin \mathbb{F}_{2^p}$. By the same arguments used earlier, there is a contradiction. At last, if two distinct roots exist and $\mu_1 \neq 0$, both of the roots must have a multiplicity equal to one.

Note that given the number of distinct roots already known, all necessary conditions mentioned above are also sufficient conditions. ∎

Now we apply the theorem above to the original polynomial $f(Y) = f_4 Y^4 + f_3 Y^3 + f_2 Y^2 + f_1 Y + f_0$ of Subsection 8.3.3. Though $f(Y)$ may not be affine in nature, the condition of its roots can be inferred from the theorem and from z_0, z_1 and $\hat{z}$. (One may recall from Subsection 8.3.2 that z_0 and z_1 are the two numbers "extracted" from matrix $\underline{\underline{\hat{M}}} - \underline{\underline{I}}$, which, as well as vector $\underline{\hat{z}}$, are outcome of the matrix reduction procedure.) Let us define binary variables $\{C_i, i = 0, ..., 6\}$ as follows:

$$C_0 = \begin{cases} 1, & \text{if equation } \underline{y}\,\underline{\underline{\hat{M}}} = \underline{z} \text{ has solution;} \\ 0, & \text{otherwise.} \end{cases}$$

$$C_1 = \begin{cases} 1, & \text{if } z_0 = z_1; \\ 0, & \text{otherwise.} \end{cases} \quad C_2 = \begin{cases} 1, & \text{if } z_0 = 0; \\ 0, & \text{otherwise.} \end{cases}$$
$$C_3 = \begin{cases} 1, & \text{if } f_3 = 0; \\ 0, & \text{otherwise.} \end{cases} \quad C_4 = \begin{cases} 1, & \text{if } a = 0; \\ 0, & \text{otherwise.} \end{cases}$$
$$C_5 = \begin{cases} 1, & \text{if } b = 0; \\ 0, & \text{otherwise.} \end{cases} \quad C_6 = \begin{cases} 1, & \text{if } c = 0 \\ 0, & \text{otherwise.} \end{cases}$$

For the case of $\deg Q(0,Y) = 4$, the resulting roots and their associated orders from solving quartic equation $f(Y) = f_4 Y^4 + f_3 Y^3 + f_2 Y^2 + f_1 Y + f_0 = 0$ are listed in the following table. In a word, the root conditions can be derived from evaluating the logic functions in the second column of the table. To keep the brevity of the paper, the combinational logic circuit used to evaluate those logic functions is not given. Since the quartic polynomial equation solver is also

Table 8.2: Root conditions for quartic polynomials

Case	Condition	Root(s) and Order(s)
0	$(C_0\&C_3\&C_4\&C_5\&\bar{C}_6)$	1 root ($\hat{z}$) of order 4
1	$(\bar{C}_0\&\bar{C}_3\&C_4\&C_5\&C_6)$	1 root ($\sqrt{\frac{f_1}{f_3}}$) of order 4
2	$\bar{C}_0\&\bar{C}_3\&\bar{C}_5\&C_6$	1 root ($\sqrt{\frac{f_1}{f_3}}$) of order 2
3	$C_0\&C_1\&C_2\&C_3\&\bar{C}_6$	1 root ($\hat{z}$) of order 1
4	$C_0\&C_1\&C_2\&\bar{C}_3\&\bar{C}_6$	1 root ($\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$) of order 1
5	$(C_0\&C_1\&\bar{C}_2\&C_3\&C_4\&\bar{C}_6)$	2 roots ($\hat{z}$ and $\hat{z} + z_0$) both of order 2
6	$(C_0\&C_1\&C_2\&\bar{C}_3\&C_4\&C_6)$	2 roots ($\sqrt{\frac{f_1}{f_3}}$ and $\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$) both of order 2
7	$C_0\&\bar{C}_3\&C_5\&C_6$	2 roots, one ($\sqrt{\frac{f_1}{f_3}}$) of order 3 and the other ($\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$) of order 1
8	$C_0\&C_1\&\bar{C}_2\&C_3\&\bar{C}_4\&\bar{C}_6$	2 roots ($\hat{z}$ and $\hat{z} + z_0$) both of order 1
9	$C_0\&C_1\&\bar{C}_2\&\bar{C}_3\&\bar{C}_4\&\bar{C}_6$	2 roots ($\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$ and $\frac{1}{\hat{z}+z_0} + \sqrt{\frac{f_1}{f_3}}$) both of order 1
10	$C_0\&C_1\&\bar{C}_2\&C_6$	3 roots, one ($\sqrt{\frac{f_1}{f_3}}$) of order 2, the other two ($\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$ and $\frac{1}{\hat{z}+z_0} + \sqrt{\frac{f_1}{f_3}}$) of order 1
11	$C_0\&\bar{C}_1\&\bar{C}_2\&C_3$	4 roots ($\hat{z}$, $\hat{z} + z_0$, $\hat{z} + z_1$, and $\hat{z} + z_0 + z_1$), all of order 1
12	$C_0\&\bar{C}_1\&\bar{C}_2\&\bar{C}_3$	4 roots ($\frac{1}{\hat{z}} + \sqrt{\frac{f_1}{f_3}}$, $\frac{1}{\hat{z}+z_0} + \sqrt{\frac{f_1}{f_3}}$, $\frac{1}{\hat{z}+z_1} + \sqrt{\frac{f_1}{f_3}}$, and $\frac{1}{\hat{z}+z_0+z_1} + \sqrt{\frac{f_1}{f_3}}$), all of order 1
13	$\bar{C}_0\&\bar{C}_6$	no roots

used to find roots for lower-degree polynomials, similar tables can be obtained for the cases of $\deg Q(0,Y) < 4$.

The architecture shown in Fig. 8.15 can then be used to generate the roots of the polynomial under consideration. Depending on the degree and coefficients of the polynomial, the total number of roots and the orders of the roots are different. As will become clear in the context of the overall factorization architecture, the control signal of the switch in Fig. 8.15 should be designed in such a way that appropriate roots are routed to the four output ports, γ_0 through γ_3. In the worst case, four 5:1 MUXes for p-bit inputs are needed to implement the switch.

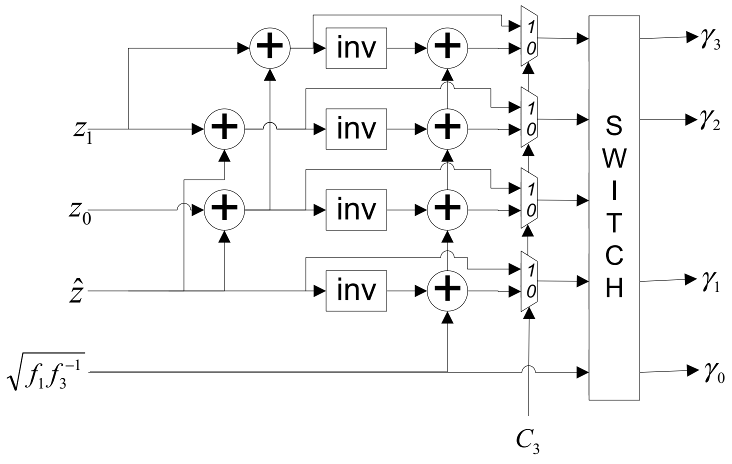

Figure 8.15: Root computation from results of the matrix reduction procedure.

8.4 Overall Factorization Architecture

A parallel factorization architecture, where root computation and polynomial update by FST (fast shift transform) for all $Q(X,Y)$ in the same iteration level are carried out simultaneously, is given in Fig. 8.16. Since we only deal with bivariate polynomials of Y-degree four, a maximum of four copies of the bivariate polynomial coefficient buffer and corresponding FST engines are needed. In Fig. 8.16, the superscript (i) in $Q_j^{(i)}(X,Y)$, $Q_j^{(i)}(0,Y)$ and $\gamma_{j,j'}^{(i)}$ indicates the iteration level, the subscript j identifies the bivariate polynomial at the iteration level, and the second subscript j' in $\gamma_{j,j'}^{(i)}$ is used as root indices. For example $\gamma_{1,1}^{(i)}$ refers to the second root found from solving equation $Q_1^{(i)}(0,Y)$ at iteration level i. There are three types of equation solvers in our architecture, namely, linear, quadratic and quartic equation solvers. As mentioned in Subsection 8.3.3, the quartic equation solver unit can be configured to compute roots for lower degree polynomials. Though it is more area efficient to solely use the quartic equation solver to handle all polynomial equations of degree lower than five, applying the quartic polynomial equation solver to linear polynomial is certainly an “overkill” and causes unnecessary delay. In addition, our simulations indicate that, with a high probability, only linear equations arise in subsequent iteration levels. Thus a linear equation solver is used as a “slave” engine to the quartic equation solver. By doing so, the worst-case factorization latency is not improved, but the average factorization delay is greatly reduced. The same argument applies to the linear equation solver bundled with the quadratic equation solver in Fig. 8.16. In summary, a total of four linear equation solvers, one quadratic equation solver and one quartic equation solver are used in our architecture. For the linear equation solver, the RC1 architecture of [ZP05] can be used. As one can see, the two extra linear equation solvers only cost two $\mathbb{F}_{2^p}$ inverters and multipliers, two 2:1 MUXes and two p-bit registers. The root condition check block associated with the quartic equation solver consists of combinational logic used to evaluate the logic functions given in the second column of Table 8.2. It takes the polynomial coefficients (a, b and c), results of the matrix reduction procedure ($\hat{z}$, z_0 and z_1), f_3, etc., as inputs and generates a control signal for the switch given in Fig. 8.15. A similar root condition check block is used for the quadratic equation solver as well. In addition, the root MUX controller block is used to generate controlling signals, to be defined later, for the MUXes that select the roots at the input of the four FST engines. At last, the four root buffers store all possible factorization output sequences.

It should be emphasized that in our new architecture, each of the four FST

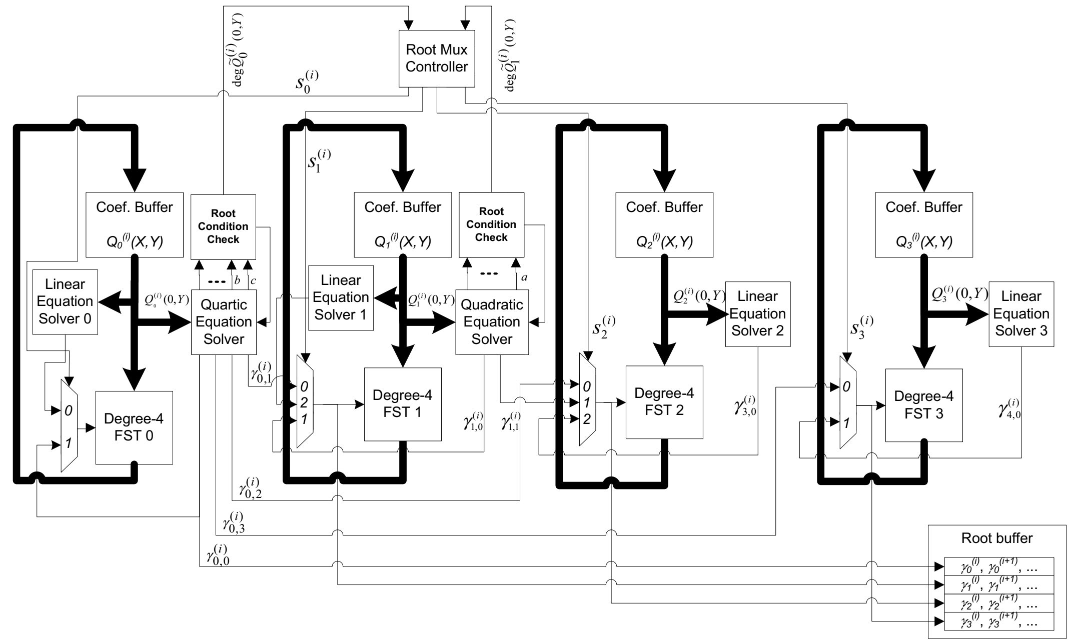

Figure 8.16: Factorization architecture for bivariate polynomial of Y-Degree four.

engines is tied to a polynomial coefficient buffer. Compared to the architecture of [ZP05], where a large number of MUXes and DeMUXes are used in the root and polynomial scheduling and de-scheduling blocks to route polynomial coefficients from polynomial buffers to FST engines, our new architecture only needs to route appropriate roots from the equation solvers to FST engines, thus significantly reducing MUX consumption. At each iteration of the factorization procedure, the appropriate root is routed to each FST engine and bivariate polynomial coefficients at the output of the FST engine are stored back to the same buffer. Our factorization architecture utilizes the following root routing scheme.

The root routing algorithm

- At the beginning of the factorization procedure, the coefficients of the bivariate polynomial $A(X, Y)$ are "broadcast" to the four coefficient buffers.
- In the ensuing iteration level i, the following cases are possible for $Q_0^{(i)}(0, Y)$.
  - Case 1: $\deg Q_0^{(i)}(0, Y) = 4$

    - If a single root of order four is found, this root is applied to all of the four FST engines.
    - If two roots, both of order two are found, one root is sent to both FST0 and FST3 and the other root is sent to both FST1 and FST2.
    - If two roots, one of order three and the other of order one, are found. The order three roots are sent to FST0, FST1 and FST2, while the order one root is used by FST3.
    - If three roots, one of order two and the other two of order one, are found, the order two roots are sent to FST1 and FST2, while the other two roots are sent to FST0 and FST3, respectively.
    - If four distinct roots are found, they are sent to the four FST engines, respectively.
    - If none of the above, not all roots of $Q_0^{(i)}(0, Y)$ are in $\mathbb{F}_{2^p}$. The valid roots can still be sent to a subset of the FST engines following the principle used by the previous cases. And some, or all, of the FST engines can be disabled for the ensuring iterations.
- Case 2: $\deg Q_0^{(i)}(0, Y) = 3$
    - If a single root of order three is found, it is sent to FST0, FST1 and FST2.
    - If two roots, one of order two and the other of order one, are found. The order two roots are sent to FST1 and FST2, while the other root is sent to FST0.
    - If three distinct roots are found, they are sent to FST0, FST1 and FST2, respectively.
    - If none of the above, not all roots of $Q_0^{(i)}(0, Y)$ are in $\mathbb{F}_{2^p}$. The valid roots can still be sent to a subset of the first three FST engines following the principle used by the previous cases. And some, or all, of the FST engines can be disabled for the ensuring iterations.
- Case 3: $\deg Q_0^{(i)}(0, Y) = 2$

    * If a single root of order two is found, it is sent to both FST0 and FST3.
    * If two distinct roots are found, they are sent to FST0 and FST3, respectively.
    * If none of the above, both FST0 and FST3 can be disabled for the ensuring iterations.
  - Case 4: $\deg Q_0^{(i)}(0,Y) = 1$, the root (computed from the "slave" linear equation solver) is sent to FST0.

- In the ensuing iteration level i, the following cases are possible for $Q_1^{(i)}(0,Y)$.
  - Case 1: $\deg Q_0^{(i)}(0,Y) = 2$
    * If a single root of order two is found, it is sent to both FST1 and FST2.
    * If two distinct roots are found, they are sent to FST1 and FST2, respectively.
    * If none of the above, both FST1 and FST2 can be disabled for the following iterations.
  - Case 2: $\deg Q_0^{(i)}(0,Y) = 1$, the root (computed from the "slave" linear equation solver) is sent to FST1.

- For linear equation solver 2 and 3, their roots are always sent to FST2 and FST3, respectively.

- Since multiple roots may be sent to a FST engine from multiple equation solvers, the final root input to the FST engines are selected by the four MUXes with the following control signals:

$$s_0^{(i)} = \begin{cases} 0, & \text{if } \deg Q_0^{(i)}(0,Y) == 1; \\ 1, & \text{otherwise.} \end{cases}$$

$$s_1^{(i)} = \begin{cases} 0, & \text{if } \deg Q_0^{(i)}(0,Y) > 2; \\ 1, & \text{else if } \deg Q_1^{(i)}(0,Y) == 2; \\ 2, & \text{otherwise.} \end{cases}$$

$$s_2^{(i)} = \begin{cases} 0, & \text{if } \deg Q_0^{(i)}(0,Y) > 2; \\ 1, & \text{else if } \deg Q_1^{(i)}(0,Y) > 1; \\ 2, & \text{otherwise.} \end{cases}$$

$$s_3^{(i)} = \begin{cases} 0, & \text{if } \deg Q_0^{(i)}(0,Y) > 1; \\ 1, & \text{otherwise.} \end{cases}$$

The validity of the routing algorithm given above is guaranteed by the Corollary 6.3 of [RR00] that if a root of order r is found at iteration i, the degree of corresponding $Q(0,Y)$ in the ensuing iteration cannot be larger than r. The implementation of the routing algorithm is feasible because of the precise knowledge of the root conditions from the direct root computation method given in Section 8.3. Accordingly, a switch can be designed so that appropriate roots appear at the four output ports of Fig. 8.15.

8.5 Example: Factorization Architecture for a (458, 410) Reed-Solomon Code

As an illustrative example, our new factorization architecture is applied to soft-decision decoding of a $(458, 410)$ RS code defined on $\mathbb{F}_{2^{10}}$. This code is used in some magnetic recording products. A reason for selecting this code as an example is that the large field size makes it easier to demonstrate the superiority of our direct root computation based factorization architecture over prior works. Throughout this section, without specific mentioning, all logic gates are assumed to be two-input gates.

8.5.1 Algorithm-Level Factorization Complexity

As shown in [KMVA03], [KMV06], the re-encoding and coordinate transformation technique also significantly reduces factorization complexity for high-rate RS codes. At most 2Δ iterations are needed in the factorization process, where Δ is the maximum number of errors to be corrected in the received hard-decision vector. Otherwise, at least k, the number of information symbols in a codeword, iterations are required for the factorization algorithm. According to our simulations carried out for the $(458, 410)$ RS code in a binary AWGN channel, as many as 32 symbol errors, eight more than a hard-decision decoder's error-correcting capability, can be corrected by the soft-decision decoder at codeword error rate of 10^{-6}. Thus for practical applications of soft-decision decoding to this RS code, we may assume that $\Delta = 32$, thus a total of 64 iterations are required for the factorization procedure.

8.5.2 Hardware Complexity and Factorization Latency Estimate

In this section, we provide an area and latency estimate for our factorization architecture. For area estimate, the gate counts of all building blocks shown in Fig. 8.16, except the controller blocks, are given. The factorization procedure involves many arithmetic operations in $\mathbb{F}_{2^{10}}$, such as multiplication, inversion, squaring, finding the square roots, etc. Detailed information regarding their VLSI implementation, including hardware complexity estimate, are discussed in the Appendix.

The critical path of our factorization architecture is determined by the critical path of the matrix reduction block discussed in Section 8.3.2. In this case, there are six AND gates, five OR gates, one inverter and one MUX in the critical path, which is comparable to the critical path in earlier designs [AKS04b], [ZP05]. Necessary pipelining is implemented for all blocks that have longer critical path. A summary of gate counts and critical paths of all building blocks, except the controllers, are given in Table 8.3.

Based on Table 8.4 of the Appendix, we see that at most three stages of

Table 8.3: Gate counts and Critical Path for the Building Blocks in Factorization Architecture

Unit	Area	Critical Path
Converting General Quartic Polynomial to Affine Quartic Polynomial (Fig. 8.13)	373XOR+260AND +36OR+5INV +90MUX+100REG	7XOR+1AND
Matrix Construction (Fig. 8.5)	27XOR+130REG	3XOR
Matrix Reduction (Fig. 8.10)	90XOR+335AND +118OR+20INV +209MUX+110REG	6AND+5OR +1INV+1MUX
Linear Equation Solver	265XOR+260AND +36OR+5INV +10MUX+10REG	6XOR+1AND
Quadratic Equation Solver	127XOR+335AND +118OR+20INV +299MUX+240REG	6AND+5OR +1INV+1MUX
Quartic Equation Solver	1265XOR+1570AND +416OR+65INV +758MUX+450REG	6AND+5OR +1INV+1MUX
FST Engine (Fig. 8.2)	567XOR+500AND +260REG	6XOR
Equation Solver to FST MUXes	60MUX	–
total	4720XOR+4945AND +678OR+105INV +1157MUX+1770REG	

pipelining are required for the datapath that includes the 2:1 MUX and $\mathbb{F}_{2^{10}}$ inverter, thus transforming a general quartic equation to an affine quartic polynomial takes six clock cycles. Constructing the matrix with the architecture shown in Fig. 8.5 needs 10 clock cycles. Matrix reduction needs 20 clock cycles. With the architecture shown in Fig. 8.15, it takes four clock cycles to route appropriate roots to the output ports. Thus a total of $6 + 10 + 20 + 4 = 40$ clock cycles are required to solve the quartic equation and route the appropriate roots to the FST engines.

We now present a worst-case latency estimate for the new factorization architecture. Since, at any iteration level, polynomial update step defined by (8.1) and (8.2) only needs to be applied to the coefficients required for root computation in future iterations, the number of clock cycles required for polynomial update decreases linearly. Section 8.2 shows that up to seven clock cycles are required for the FST engines to generate $Q(0, Y)$ for the next iteration level. Thus in the worst case, polynomial update takes $64 + 7 = 71$ clock cycles at iteration level $i = 0$ and requires seven clock cycles for the last iteration. Since root computation and routing by the quartic equation solver takes 40 clock cycles, it can completely overlap with polynomial update from iteration level $i = 0$ up until iteration level $i = 24$. For the rest 38 iterations, each iteration needs $7 + 40 = 47$ clock cycles as solving quartic equation dominates the total delay. Thus the worst case clock cycle count can be estimated as $40 + ((7 + 64) + (7 + 40)) \times 25/2 + (7 + 40) \times 38 = 3301$. If exhaustive root search based architectures [AKS03a], [ZP05] are applied, without any overlap between root computation and polynomial update, the worst-case latency is at least $1024 + ((1024 + 7 + 64) + (1024 + 7 + 1)) \times 63/2 = 680245$ clock cycles.

Our simulations indicate that high order roots are very rare in practice, and with a very high probability, roots of order one are the only roots from the initial root computation. In this case, root computation takes only one clock cycle with the linear equation solver, from iteration level $i = 2$ and onwards. Thus it takes $40 + ((4 + 63) + (4 + 1)) \times 63/2 = 2308$ clock cycles to finish factorization procedure most of the time.

8.6 Conclusion

A novel architecture based on direct root computation is proposed to speed up the factorization process of algebraic soft-decision decoding of RS codes. Even though direct root computation can only be applied to bivariate polynomials with Y degree lower than five, it is sufficient for most practical applications of algebraic soft-decision decoding. With the new architecture, there is only a small variation in decoding latency and the worst-case latency is significantly reduced compared to previous architectures. Due to precise knowledge of root orders from direct root computation, there is no need to multiplex polynomial coefficients to multiple parallel polynomial update (FST) engines, which can cost a large amount of MUXes. Thus the new architecture should be more area efficient as well.

8.7 Appendix: Arithmetics in $\mathbb{F}_{2^{10}}$

Throughout the appendix, we assume that primitive polynomial $p(X) = X^{10} + X^3 + 1$ is used to generate $\mathbb{F}_{2^{10}}$ and α is a root of $p(X)$. As it should be clear from the context, "$a\oplus$ b", "$a + b$" and "ab" denote binary XOR, OR and AND operations, respectively.

8.7.1 Multiplier Complexity

In this subsection, we show how the gate count for various types of $\mathbb{F}_{2^{10}}$ multipliers used in the factorization architecture is obtained. There are three types of constant multipliers. Let $b \in \mathbb{F}_{2^{10}}$ and define $c = \alpha^4 b$, $d = \alpha^2 b$, and $e = \alpha b$. It can be derived that:

$$\begin{array}{llll} c_0 = b_6 & c_1 = b_7 & c_2 = b_8 & c_3 = b_6 \oplus b_9 \\ c_4 = b_0 \oplus b_7 & c_5 = b_1 \oplus b_8 & c_6 = b_2 \oplus b_9 & \\ c_7 = b_3 & c_8 = b_4 & c_9 = b_5 & \end{array}$$

$$\begin{array}{lllll} d_0 = b_8 & d_1 = b_9 & d_2 = b_0 & d_3 = b_1 \oplus b_8 & d_4 = b_2 \oplus b_9 \\ d_5 = b_3 & d_6 = b_4 & d_7 = b_5 & d_8 = b_6 & d_9 = b_7 \end{array}$$

$$\begin{array}{lllll} e_0 = b_9 & e_1 = b_0 & e_2 = b_1 & e_3 = b_2 \oplus b_9 & e_4 = b_3 \\ e_5 = b_4 & e_6 = b_5 & e_7 = b_6 & e_8 = b_7 & e_9 = b_8 \end{array}$$

Thus the three constant multipliers only use four, two and one XOR gates, respectively. The general multiplier $c = ab$, where a and b are arbitrary numbers in $\mathbb{F}_{2^{10}}$, can be implemented with the following combinational logic:

$$\begin{aligned}
t_0 &= a_0b_0 \\
t_1 &= a_0b_1 \oplus a_1b_0 \\
t_2 &= a_0b_2 \oplus a_1b_1 \oplus a_2b_0 \\
t_3 &= a_0b_3 \oplus a_1b_2 \oplus a_2b_1 \oplus a_3b_0 \\
t_4 &= a_0b_4 \oplus a_1b_3 \oplus a_2b_2 \oplus a_3b_1 \oplus a_4b_0 \\
t_5 &= a_0b_5 \oplus a_1b_4 \oplus a_2b_3 \oplus a_3b_2 \oplus a_4b_1 \oplus a_5b_0 \\
t_6 &= a_0b_6 \oplus a_1b_5 \oplus a_2b_4 \oplus a_3b_3 \oplus a_4b_2 \oplus a_5b_1 \oplus a_6b_0 \\
t_7 &= a_0b_7 \oplus a_1b_6 \oplus a_2b_5 \oplus a_3b_4 \oplus a_4b_3 \oplus a_5b_2 \oplus a_6b_1 \\
&\oplus a_7b_0 \\
t_8 &= a_0b_8 \oplus a_1b_7 \oplus a_2b_6 \oplus a_3b_5 \oplus a_4b_4 \oplus a_5b_3 \oplus a_6b_2 \\
&\oplus a_7b_1 \oplus a_8b_0 \\
t_9 &= a_0b_9 \oplus a_1b_8 \oplus a_2b_7 \oplus a_3b_6 \oplus a_4b_5 \oplus a_5b_4 \oplus a_6b_3 \\
&\oplus a_7b_2 \oplus a_8b_1 \oplus a_9b_0 \\
t_{10} &= a_1b_9 \oplus a_2b_8 \oplus a_3b_7 \oplus a_4b_6 \oplus a_5b_5 \oplus a_6b_4 \oplus a_7b_3 \\
&\oplus a_8b_2 \oplus a_9b_1 \\
t_{11} &= a_2b_9 \oplus a_3b_8 \oplus a_4b_7 \oplus a_5b_6 \oplus a_6b_5 \oplus a_7b_4 \oplus a_8b_3 \\
&\oplus a_9b_2 \\
t_{12} &= a_3b_9 \oplus a_4b_8 \oplus a_5b_7 \oplus a_6b_6 \oplus a_7b_5 \oplus a_8b_4 \oplus a_9b_3 \\
t_{13} &= a_4b_9 \oplus a_5b_8 \oplus a_6b_7 \oplus a_7b_6 \oplus a_8b_5 \oplus a_9b_4 \\
t_{14} &= a_5b_9 \oplus a_6b_8 \oplus a_7b_7 \oplus a_8b_6 \oplus a_9b_5 \\
t_{15} &= a_6b_9 \oplus a_7b_8 \oplus a_8b_7 \oplus a_9b_6 \\
t_{16} &= a_7b_9 \oplus a_8b_8 \oplus a_9b_7 \\
t_{17} &= a_8b_9 \oplus a_9b_8 \\
t_{18} &= a_9b_9
\end{aligned}$$

The t_i's, for $i = 0, \ldots, 18$, are intermediate variables and the final outputs can be expressed as follows:

$$\begin{array}{ll} c_0 = t_0 \oplus t_{10} \oplus t_{17} & c_1 = t_1 \oplus t_{11} \oplus t_{18} \\ c_2 = t_2 \oplus t_{12} & c_3 = t_3 \oplus t_{10} \oplus t_{13} \oplus t_{17} \\ c_4 = t_4 \oplus t_{11} \oplus t_{14} \oplus t_{18} & c_5 = t_5 \oplus t_{12} \oplus t_{15} \\ c_6 = t_6 \oplus t_{13} \oplus t_{16} & c_7 = t_7 \oplus t_{14} \oplus t_{17} \\ c_8 = t_8 \oplus t_{15} \oplus t_{18} & c_9 = t_9 \oplus t_{16} \end{array}$$

The general multiplier requires 100 AND gates and 101 XOR gates in area with one AND gate and five XOR gates in the critical path. For squaring operations, if we define $c = a^2$ for any $a \in \mathbb{F}_{2^{10}}$, then we have

$$\begin{array}{llll} c_0 = a_0 \oplus a_5 & c_1 = a_9 & c_2 = a_1 \oplus a_6 & c_3 = a_5 \\ c_4 = a_2 \oplus a_7 \oplus a_9 & c_5 = a_6 & c_6 = a_3 \oplus a_8 & c_7 = a_7 \\ c_8 = a_4 \oplus a_9 & c_9 = a_8 & & \end{array}$$

The squaring operation only takes six XOR gates to implement with two XOR gates in the critical path.

8.7.2 Conversion Matrix for Composite Field Representation of $\mathbb{F}_{2^{10}}$

In this section, the method of [SSK03] is used to convert an element of $\mathbb{F}_{2^{10}}$ represented by standard polynomial basis to composite field representation. Instead of using α, the root of $p(X) = X^{10} + X^3 + 1$ to construct a primitive polynomial over $\mathbb{F}_{2^5}$, we choose to use another primitive element of $\mathbb{F}_{2^{10}}$, namely, $\tilde{\alpha} = \alpha^{343}$. The minimal polynomial of the later with respect to $\mathbb{F}_{2^5}$ has a linear coefficient equal to 1 as shown below.

$$\begin{aligned} M_{2^5,\alpha^{343}}(X) &= (X + \alpha^{343})(X + \alpha^{343\times 2^5}) \\ &= X^2 + (\alpha^{343} + \alpha^{746})X + \alpha^{66} \\ &= X^2 + X + \alpha^{66}. \end{aligned} \tag{8.5}$$

And $B_{(2^5)^2} = \left[1\ \tilde{\alpha}\right]$ is a basis of $\mathbb{F}_{(2^5)^2}$ over $\mathbb{F}_{2^5}$. From Theorem 1 of [SSK03], $\gamma = \alpha^{343\times 33} = \alpha^{66}$ is primitive in $\mathbb{F}_{2^5}$. Thus a standard basis of $\mathbb{F}_{2^5}$ can be defined as

$$B_{2^5} = \left[1\;\gamma\;\gamma^2\;\gamma^3\;\gamma^4\right] = \left[1\;\alpha^{66}\;\alpha^{132}\;\alpha^{198}\;\alpha^{264}\right]. \tag{8.6}$$

The primitive polynomial that generates this field can be computed as the minimal polynomial of γ with respect to $\mathbb{F}_2$.

$$\begin{aligned} & (X+\gamma)(X+\gamma^2)(X+\gamma^4)(X+\gamma^8)(X+\gamma^{16}) \\ = \;& (X^2+(\gamma+\gamma^2)X+\gamma^3)(X^2+(\gamma^4+\gamma^8)X+\gamma^{12})(X+\gamma^{16}) \\ = \;& (X^2+(\alpha^{66}+\alpha^{132})X+\alpha^{198}) \\ & (X^2+(\alpha^{264}+\alpha^{528})X+\alpha^{792})(X+\alpha^{33}) \\ = \;& X^5+X^4+X^3+X^2+1. \end{aligned} \tag{8.7}$$

With the primitive polynomial given above, it can be easily established that general multiplication in $\mathbb{F}_{2^5}$ requires 25 AND gates and 29 XOR gates, with the critical path consisting of one AND and four XOR gates. For an element $\beta \in \mathbb{F}_{2^{10}}$, it has two different representations:

$$\beta = \sum_{i=0}^{9} \beta_i \alpha^i \tag{8.8}$$

$$\beta = \sum_{i=0}^{1} \tilde{\beta}_i \tilde{\alpha}^i, \tag{8.9}$$

where $\beta_i \in \mathbb{F}_2$ and $\tilde{\beta}_i \in \mathbb{F}_{2^5}$. Since $\tilde{\beta}_i$'s belong to $\mathbb{F}_{2^5}$, they can be represented by the basis given in (8.6), i.e.,

$$\tilde{\beta}_i = \sum_{j=0}^{4} \hat{\beta}_{i,j} \alpha^{66j}. \tag{8.10}$$

Substituting (8.10) back into (8.9), we get

$$\begin{aligned} \beta \;=\; & \sum_{i=0}^{1}\sum_{j=0}^{4} \hat{\beta}_{i,j}\alpha^{66j}\tilde{\alpha}^i = \sum_{i=0}^{1}\sum_{j=0}^{4} \hat{\beta}_{i,j}\alpha^{343i+66j} \\ =\; & \hat{\beta}_{0,0} + \hat{\beta}_{0,1}\alpha^{66} + \hat{\beta}_{0,2}\alpha^{132} + \hat{\beta}_{0,3}\alpha^{198} + \hat{\beta}_{0,4}\alpha^{264} \\ +\; & \hat{\beta}_{1,0}\alpha^{343} + \hat{\beta}_{1,1}\alpha^{409} + \hat{\beta}_{1,2}\alpha^{475} \\ +\; & \hat{\beta}_{1,3}\alpha^{541} + \hat{\beta}_{1,4}\alpha^{607}. \end{aligned} \tag{8.11}$$

Thus by applying (8.8) and (8.9), we get

$$\begin{aligned}[\hat{\beta}_{0,0} \dots \hat{\beta}_{0,4}\ \hat{\beta}_{1,0} \dots \hat{\beta}_{1,4}]\underline{\underline{\Pi}}[1\ \alpha \dots \alpha^9]^t \\ = [\beta_0\ \beta_1 \dots \beta_9][1\ \alpha \dots \alpha^9]^t.\end{aligned}$$

The matrix $\underline{\underline{\Pi}}$ that converts an element from its composite field representation to a direct representation, can be expressed as follows:

$$\underline{\underline{\Pi}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Correspondingly, the matrix $\underline{\underline{\Psi}}$ that converts an element from the direct representation to its composite field representation can be expressed as follows:

$$\underline{\underline{\Psi}} = \underline{\underline{\Pi}}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Implementation of multiplication by $\underline{\underline{\Pi}}$ requires 28 XOR gates; and implementation of multiplication by $\underline{\underline{\Psi}}$ requires 32 XOR gates. The critical path in both multiplication circuits consists of three XOR gates.

8.7.3 Direct Inversion in $\mathbb{F}_{2^5}$

With $p_{2^5}(X) = X^5 + X^4 + X^3 + X^2 + 1$ serving as the generating polynomial for $\mathbb{F}_{2^5}$, the direct inversion, $b = a^{-1}$ for any $a \in \mathbb{F}_{2^5}$, can be expressed with the following logic equations:

$$\begin{aligned}
b_0 &= \bar{a}_1 a_3 a_4 + \bar{a}_1 \bar{a}_2 a_3 + \bar{a}_0 (a_1 \bar{a}_2 \bar{a}_3 + a_1 \bar{a}_2 a_4) \\
&+ a_0 (\bar{a}_1 \bar{a}_2 + \bar{a}_1 a_4 + \bar{a}_1 a_3 + a_1 \bar{a}_3 \bar{a}_4 + a_2 a_3 a_4) \\
b_1 &= a_2 \bar{a}_3 a_4 + a_1 \bar{a}_2 \bar{a}_3 + a_1 \bar{a}_2 a_4 + a_0 a_1 \bar{a}_3 \\
&+ \bar{a}_0 (\bar{a}_1 a_2 + \bar{a}_1 a_3 \bar{a}_4 + a_2 a_3 \bar{a}_4) \\
b_2 &= a_1 a_2 a_3 a_4 + a_0 (\bar{a}_1 \bar{a}_3 a_4 + \bar{a}_1 \bar{a}_2 a_4 + a_1 \bar{a}_2 \bar{a}_4) \\
&+ \bar{a}_0 (a_2 \bar{a}_3 + a_1 a_2 + \bar{a}_1 \bar{a}_2 a_3 + a_1 \bar{a}_3 a_4 + a_1 a_3 \bar{a}_4) \\
b_3 &= \bar{a}_1 a_2 \bar{a}_4 + a_1 \bar{a}_2 a_3 + a_0 (\bar{a}_1 \bar{a}_3 + \bar{a}_1 \bar{a}_2 a_4 + a_1 a_3 \bar{a}_4) \\
&+ \bar{a}_0 (\bar{a}_1 a_3 \bar{a}_4 + \bar{a}_1 a_2 a_3 + a_1 a_2 \bar{a}_3 a_4) \\
b_4 &= \bar{a}_1 \bar{a}_2 \bar{a}_3 a_4 + a_0 (\bar{a}_1 a_2 + a_2 a_4 + a_1 \bar{a}_2 a_3) \\
&+ \bar{a}_0 (a_1 a_2 \bar{a}_4 + a_2 \bar{a}_3 \bar{a}_4 + a_1 \bar{a}_2 \bar{a}_3 + a_1 \bar{a}_2 a_4).
\end{aligned}$$

The logic functions given above can be implemented with $17 + 14 + 19 + 18 + 17 = 85$ AND gates, $8 + 6 + 8 + 7 + 7 = 36$ OR gates, and five inverters. Note that further area optimization is possible by taking advantage of common subexpressions. The critical path in this implementation has one inverter, three AND gates and three OR gates.

8.7.4 $\mathbb{F}_{2^{10}}$ Inversion in Composite Field

Let us assume that $\beta \in \mathbb{F}_{2^{10}}$ can be represented in the composite field as $\beta = \tilde{\beta}_0 + \tilde{\beta}_1 X$ and its inverse is $\eta = \tilde{\eta}_0 + \tilde{\eta}_1 X$, where $\tilde{\beta}_0, \tilde{\beta}_1, \tilde{\eta}_0, \tilde{\eta}_1 \in \mathbb{F}_{2^5}$. The following equation

$$\begin{aligned}\eta\beta &= (\tilde{\eta}_0 + \tilde{\eta}_1 X)(\tilde{\beta}_0 + \tilde{\beta}_1 X) \mod M_{2^5,\alpha^{343}}(X)\\ &= (\tilde{\eta}_0\tilde{\beta}_0 + \tilde{\eta}_1\tilde{\beta}_1\alpha^{66}) + (\tilde{\beta}_1\tilde{\eta}_0 + \tilde{\beta}_0\tilde{\eta}_1 + \tilde{\beta}_1\tilde{\eta}_1)X\\ &= 1,\end{aligned}$$

needs to be satisfied. Thus $\tilde{\eta}_0$ and $\tilde{\eta}_1$ can be computed as

$$\begin{aligned}\tilde{\eta}_0 &= \frac{\tilde{\beta}_0 + \tilde{\beta}_1}{\tilde{\beta}_0(\tilde{\beta}_0 + \tilde{\beta}_1) + \tilde{\beta}_1^2\alpha^{66}}\\ \tilde{\eta}_1 &= \frac{\tilde{\beta}_1}{\tilde{\beta}_0(\tilde{\beta}_0 + \tilde{\beta}_1) + \tilde{\beta}_1^2\alpha^{66}}.\end{aligned} \tag{8.12}$$

In $\mathbb{F}_{2^5}$, it can be estimated that general multiplication needs 25 AND gates and 29 XOR gates with the critical path consisting of one AND gate and four XOR gates, and multiplication by constant α^{66} takes three XOR gates with only one XOR gate on the critical path. For the squaring operation of any number $a \in \mathbb{F}_{2^5}$, let us define $c = a^2$, then with the primitive polynomial given in (8.8), we have

$$\begin{array}{lll} c_0 = a_0 \oplus a_3 & c_1 = a_3 & c_2 = a_1 \oplus a_3 \oplus a_4\\ c_3 = a_4 & c_4 = a_2 \oplus a_4. & \end{array}$$

Thus squaring in $\mathbb{F}_{2^5}$ only requires four XOR gates with the critical path consisting of two XOR gates.

8.7.5 Conversion between Standard Basis and Normal Basis in $\mathbb{F}_{2^{10}}$

Computing the square root of a number in a finite field can be carried bout by first converting the number from its standard basis representation to normal basis representation followed by a cyclic shift and a conversion back to standard basis representation. It has been shown in [MS81] that any finite field $\mathbb{F}_{2^p}$ contains an element γ such that $\{\gamma, \gamma^2, ..., \gamma^{2^{p-1}}\}$ is a normal basis of the field.

Table 8.4: Gate counts and critical paths of function blocks in the implementation of $\mathbb{F}_{2^{10}}$ composite field inversion

	number of gates	critical path
$\times\Psi$	32XOR	3XOR
$\times\Pi$	28XOR	3XOR
$\times\alpha^{66}$	3XOR	1XOR
squaring in $\mathbb{F}_{2^5}$	4XOR	2XOR
general multiplier	25AND+29XOR	1AND+4XOR
inversion in $\mathbb{F}_{2^5}$	85AND+36OR +5INV	3AND+3OR +1INV
total	160AND+164XOR +36OR+5INV	5AND+3OR +1INV+16XOR

However, γ may not be a primitive element of the field, or even if γ is a primitive element, it may not be the root of primitive polynomial that generates the field. For example, if $p(X) = X^{10} + X^3 + 1$ is used to generate $\mathbb{F}_{2^{10}}$, the root of this polynomial, α, cannot be directly used to construct a normal basis. Using an exhaustive search method, we find that $\gamma = \alpha^7$ can be used to form a normal basis. In this case, let us introduce two binary 10×10 matrices $\underline{\underline{\Xi}}$ and $\underline{\underline{\Gamma}}$ such that

$$[\gamma\, \gamma^2 \,...\, \gamma^{2^9}]^t = \underline{\underline{\Xi}}[1\, \alpha \,...\, \alpha^9]^t$$

$$[1\, \alpha \,...\, \alpha^9]^t = \underline{\underline{\Gamma}}[\gamma\, \gamma^2 \,...\, \gamma^{2^9}]^t.$$

In this case, the standard to normal basis conversion matrix $\underline{\underline{\Xi}}$ is:

$$\underline{\underline{\Xi}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix},$$

and the normal to standard basis conversion matrix $\underline{\underline{\Gamma}}$ can be computed as follows:

$$\underline{\underline{\Gamma}} = \underline{\underline{\Xi}}^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

If $\underline{\beta}_n$ is the representation of an element $\beta \in \mathbb{F}_{2^{10}}$ on a normal basis, then the standard basis representation $\underline{\beta}_s$ can be computed as $\underline{\beta}_s = \underline{\beta}_n \, \underline{\underline{\Xi}}$. This vector and matrix multiplication can be implemented with 22 XOR gates. On the other hand, the standard to normal representation conversion has the form $\underline{\beta}_n = \underline{\beta}_s \, \underline{\underline{\Gamma}}$, and 19 XOR gates are needed for this vector and matrix multiplication, too.

As a summary, gate counts and critical paths of all $\mathbb{F}_{2^{10}}$ arithmetic operators discussed in the Appendix are presented in Table 8.5.

Table 8.5: Gate counts and critical paths of arithmetic units for $\mathbb{F}_{2^{10}}$

Unit	Area	Critical Path
General Multiplier	101XOR+100AND	5XOR+1AND
α Multiplier	1XOR	1XOR
α^2 Multiplier	2XOR	1XOR
α^4 Multiplier	4XOR	1XOR
Inverter	160AND+164XOR +36OR+5INV	5AND+3OR +1INV+16XOR
Square	6XOR	2 XOR
Square Root	41XOR	7XOR

The material of Chapter 8, in part, is published in proceedings of *2007 International Conference Acoustics, Speech and Signal Processing (ICASSP)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng; Chen Qinqin, and proceedings of *2007 International Symposium on Circuits and Systems (ISCAS)*, Ma, Jun; Vardy, Alexander; Wang, Zhongfeng; Chen Qinqin. It is also to be published in *IEEE Transactions on VLSI Systems* Ma, Jun; Wang, Zhongfeng; Vardy, Alexander. The dissertation author was the primary investigator and author of both papers.

CHAPTER 9

Conclusions and Future Research

In this chapter, we conclude the thesis by discussing some open problems.

9.1 Divide-and-Conquer Interpolation

One can see that the complexity of the divide-and-conquer algorithm presented in Chapter 3 is

$$C(n, r, \rho) = 2C(n/2, r, \rho/2) + C_m + C_e, , \tag{9.1}$$

where C_m is the cost of polynomial multiplication and C_e is the cost of the elimination step. The sum if C_m and C_e can be considered as the cost of "merge" step in the divide-and-conquer approach. The divide-and-conquer is practical only if C_m and C_e are not too large to offset the complexity reduction achieved by dividing the original interpolation problem into smaller ones. For the multiplication step there is a problem of efficient multiplication of large polynomials which may appear after processing of many interpolation points. For the elimination step, the task is to find a Groebner basis of the polynomial ideal. Design of such an elimination algorithm remains an open problem. In addition, how to combine the divide-and-conquer approach with the re-encoding coordinate transformation techniques presented in Chapter 4 is also not clear at all.

9.2 Re-encoding through *n* Points

In Chapter 4, a re-encoding coordinate transformation technique is proposed to reduce the interpolation complexity of algebraic soft decoding of Reed-Solomon code. The key idea of Chapter 4 is to shift the interpolation points by a polynomial of of degree $< k$, which then makes it possible to eliminate the k points with the largest multiplicities from the interpolation problem. A natural question to ask is: can we re-encode through n points, thereby eliminating $n-k$ more points from the interpolation problem? If this is possible, it will lead to significantly more decoding complexity reduction for low-rate RS codes. Let us see how this could be done.

The re-encoding point set will include 1 point for each of the X coordinates that define the code, i.e., $\mathcal{R} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. The n Y coordinates $y_1, y_2, \ldots, y_n$ are precisely the hard-decision received vector.

First of all, one needs to find a polynomial $h(X)$ of degree smaller than n, such that

$$h(x_i) \;=\; y_i \qquad \text{for all } (x_i, y_i) \in \mathcal{R} \tag{9.2}$$

Then the original interpolation point set $\mathcal{P}$ can be modified into the following "shifted" interpolation point set:

$$\mathcal{P}' \overset{\text{def}}{=} \Big\{ \big(x_1, y_1 - h(x_1)\big), \ldots, \big(x_k, y_s - h(x_s)\big) \Big\} \tag{9.3}$$

Finding a bivariate polynomial that passes all points in $\mathcal{P}'$ with prescribed multiplicities is less complex than finding a bivariate polynomial that satisfies all constraints defined by the original interpolation point set $\mathcal{P}$, since the Groebner-basis polynomials can be *initialized* as

$$Q'_v(X, Y) = \prod_{i=1}^{n} (X - x_i)^{[m_{x_i, y_i} - v]^+} Y^v, \text{ for } v = 0, 1, ..., r. \tag{9.4}$$

Thus the constraints defined by the first n points in $\mathcal{P}'$ are solved simply by initialization. Next, Koetter's iterative algorithm can be carried out as usual to solve the interpolation problem defined by the rest of the points in $\mathcal{P}'$ and their prescribed multiplicities. Note that there might be several interpolation points with the same X coordinate. The one with the highest multiplicity is handled by (9.4), but the interpolation constraints associated with the other points have to be still enforced (via Koetter's interpolation algorithm, for example). The end result is a set of polynomials

$$Q'_v(X,Y) = \sum_{j=0}^{r} b_{v,j}(X) \prod_{i=1}^{n} (X - x_i)^{[m_{x_i,y_i}-j]^+} Y^j, \text{ for } v = 0, 1, ..., r. \tag{9.5}$$

Theorem 3 of Chapter 4 still holds – that is, the polynomials $Q_v(X,Y) = Q'_v(X, Y - h(X))$, for $v = 0,1,...,r$ satisfy the constraints defined by the original set of interpolation points $\mathcal{P}$ and their prescribed multiplicities. However, does the set $Q_v(X,Y)$, for $v = 0,1,...,r$ contain a polynomial of minimum $(1, k-1)$-weighted degree that satisfies all the interpolation constraints? Unfortunately, the answer is NO. Note that the $Q'_v(X,Y)$'s are initialized in (9.4) to a set of polynomials of minimal $(1, n-1)$-weighted degree that satisfy the constraints defined by the first n points in $\mathcal{P}'$. Thus, no matter what monomial order is chosen in the ensuing interpolation process for the remaining points in $\mathcal{P}'$, $Q_v(X,Y)$ won't have the desired property.

What can be done? One can use, for example, the algorithm of [LO06a] can be used to convert $Q_v(X,Y)$'s from a Groebner basis, with respect to $(1, n-1)$-weighted monomial order, to a Groebner basis with respect to $(1, k-1)$-weighted monomial order. The algorithm of [LO06a] is much simpler than the general Buchberger's algorithm.

In summary, the following algorithm can be applied to carry out the interpolation:

Step 1. Find a polynomial $h_r(X)$ of degree less than n such that $h_r(x_i) = y_i$, for $i = 1, ..., n$.

Step 2. Shift the points in set $\mathcal{P}$ as follows: $y_i := y_i - h_r(x_i)$, which results in a new point set $\mathcal{P}'$.

Step 3. Initialize a set of basis polynomial as shown in (9.4) and then apply Koetter's algorithm to find a Groebner basis that satisfies all interpolation constraints defined by the points in $\mathcal{P}'$.

Step 4. Shift the resulting polynomials to obtain a new set of polynomials $Q_v(X,Y) = Q'_v(X, Y - h_r(X))$, for $v = 0, 1, ..., r$.

Step 5. Apply the algorithm of [LO06a], or alternatives, to compute a Groebner basis, with respect to $(1, k-1)$-weighted monomial order, for the polynomial ideal defined by the $Q_v(X,Y)$'s.

Step 6. Select the polynomial with minimal $(1, k-1)$-weighted as the final output of the interpolation.

Though the foregoing algorithm works, it is not simpler, in terms of arithmetic complexity, than the re-encoding through-k-point method described in Chapter 4.

The following steps of the algorithm described above are the major contributors to complexity. First of all, the polynomial set resulting from running Koetter's interpolation algorithm has to be "shifted" back in Step 4 by re-applying the re-encoding polynomial. Secondly, a "reduction" algorithm such as [LO06a] has to be performed in Step 5. So the questions remaining to be answered are as follows:

Q1. Is is possible to somehow bypass step 4 of the algorithm described above?

Q2. Is there a simpler algorithm for step 5?

In Chapter 4, question Q1 is resolved by applying a coordinate transformation technique, then directly factoring polynomial obtained from solving the "reduced" interpolation problem (without shifting back to the original point set). Something similar would have to be developed for the re-encoding-through-n-points method, if it is to be practical. However, the answer to Q1 is independent of that to Q2. Thus one can postpone this issue for later, and focus on Q2.

9.3 Multivariate Interpolation

Multivariate interpolation is the enabling force behind the newly devised techniques that correct errors beyond the Guruswami-Sudan list-decoding radius [PV04], [PV05]. The multivariate interpolation can be carried out by extending Koetter's interpolation algorithm to multiple variables. The extended Koetter's algorithm still runs polynomial-time. However, it is much more complex than the bivariate polynomial case. For example, in the case of trivariate interpolation, it takes $O(n^2m^8/R^{2/3})$ operations (additions and multiplications) in $\mathbb{F}_q$. Thus it will be of great practical value to reduce the complexity of multivariate interpolation. In principle, the re-encoding coordinate transformation technique described in Chapter 4 should help. However, the details remain to be worked out.

In addition, no VLSI architecture has been devised for multivariate interpolation. Thus it would be interesting to extend the architectures for bivariate polynomials presented in this thesis to multivariate interpolation.

9.4 More Efficient Decoder Architecture

VLSI architectures for major computational blocks of algebraic soft-decision Reed-Solomon decoder have been proposed in this thesis. However, further improvement are possible for these architectures.

The re-encoding coordinate transformation frontend architecture proposed in Chapter 6 is for a fixed choice of code parameters. A generic design would be more interesting. In addition, we have ignored the interpolation points whose X coordinates coincide with those of the re-encoding points. Interpolating through those points actually provides further decoding gain. Thus future architectures should handle those interpolation points.

The factorization architecture of Chapter 8 has a relatively large critical path delay. And it is not clear how to pipeline the architecture, especially the Degree-4 polynomial equation solver.

Beyond all of these architecture-level improvement, implementing the entire algebraic soft-decision Reed-Solomon decoder into an ASIC is a very challenging task.

Reference

[AKS03a] Arshad Ahmed, Ralf Koetter, and Naresh R. Shanbhag, *VLSI Architectures for soft-decision decoding of Reed-Solomon Codes*, IEEE Transactions on VLSI Systems (2003), submitted for publication.

[AKS03b] Ahmed Arshad, Ralf Koetter, and Naresh R. Shanbhag, *Systolic interpolation architectures for soft-decoding Reed-Solomon codes*, IEEE Workshop on Signal Processing Systems, August 2003, pp. 81–86.

[AKS04a] ______, *Reduced complexity interpolation for soft-decoding of Reed-Solomon codes*, IEEE Symposium on Information Theory (ISIT), vol. 5, June 2004, p. 385.

[AKS04b] ______, *VLSI architectures for soft-decision decoding of Reed-Solomon codes*, IEEE International Conference on Communications, vol. 5, June 2004, pp. 2584–2590.

[Ale06] Michael Alekhnovich, *Linear diophantine equations over polynomials and soft decoding of Reed-Solomon codes*, IEEE Transactions on Information Theory **51** (2006), no. 7, 2257–2265.

[AP00] Daniel Augot and Lancelot Pecquet, *A Hensel lifting to replace factorization in list-decoding of algebraic-geometric and Reed-Solomon codes*, IEEE Transactions on Information Theory **46** (2000), 2605–2614.

[Ber68] Elwyn R. Berlekamp, *Algebraic Coding Theory*, McGrow-Hill, New York, 1968.

[BL00] Bernhard Beckermann and George Labahn, *Fraction-free computation of matrix rational interpolants and matrix GCDs*, SIAM Journal on Matrix Analysis and Applications **22** (2000), no. 1, 114–144.

[Bla02] Richard E. Blahut, *Algebraic Codes for Data Transmission*, Cambridge University Press, May 2002.

[CLO96] David A. Cox, John B. Little, and Don O'Shea, *Ideals, Varieties, and Algorithms*, Springer-Verlag, Berlin, 1996.

[Fen99] Gui-Liang Feng, *Fast algorithms in Sudan decoding procedure for Reed-Solomon codes*, 37th Annual Allerton Conference on Communication, Control and Computing (Monticello, IL), October 1999, pp. 545–554.

[FG01] Gui-Liang Feng and Xavier Giraud, *Fast algorithms in Sudan decoding procedure for Reed-Solomon codes*, submitted for publication, 2001.

[FG05] Jeffrey B. Farr and Shuhong Gao, *Groebner basis, Pade approximation, and decoding of linear codes*, Contemporary Mathematics American Mathematics Society **381** (2005), In: Coding Theory and Quantum Computing.

[Gan88] Feliks R. Gantmakher, *Matrices Theory*, 4th ed., Moscow: Nauka, 1988, In Russian.

[GKG04] Warren J. Gross, Frank R. Kschischang, and P. Glenn Gulak, *An FPGA interpolation processor for soft-decision Reed-Solomon decoding*, Proceedings of IEEE Workshop on Field-Programmable Custom Computing Machines (FCCM'04) (Napa, CA), April 2004, pp. 310–311.

[GKKG02] Warren J. Gross, Frank R. Kschischang, Ralf Koetter, and P. Glenn Gulak, *A VLSI architecture for interpolation in soft-decision list decoding of Reed-Solomon codes*, Proceedings of IEEE Workshop on Signal Processing Systems (San Diego, CA), October 2002, pp. 39–44.

[GKKG05] ______, *Towards a VLSI architecture for interpolation-based soft-decision Reed-Solomon decoders*, Journal of VLSI Signal Processing **39** (2005), no. 1–2, 93–111.

[GR06] Philippe Gaborit and Olivier Ruatta, *Improved Hermite multivariable polynomial interpolation*, IEEE International Symposium on Information Theory, 2006.

[GS99] Venkatesan Guruswami and Madhu Sudan, *Improved decoding of Reed-Solomon and algebraic-geometric codes*, IEEE Transactions on Information Theory **45** (1999), 1755–1764.

[HS99] Didier Henrion and Michael Sebek, *Reliable numerical methods for polynomial matrix triangularization*, IEEE Transactions on Automatic Control **44** (1999), no. 3, 497–508.

[HvLP98] T. Høholdt, J. H. van Lint, and R. Pellikaan, *Algebraic geometry of codes*, Handbook of coding theory, Vol. I, II, North-Holland, Amsterdam, 1998, pp. 871–961.

[Kai80] Thomas Kailath, *Linear Systems*, Prentice Hall, 1980.

[KMV06] Ralf Koetter, Jun Ma, and Alexander Vardy, *A complexity reducing transformation in algebraic soft decoding of Reed-Solomon codes*, manuscript in preparation, 2006.

[KMVA03] Ralf Koetter, Jun Ma, Alexander Vardy, and Ahmed Arshad, *Efficient interpolation and factorization in algebraic soft-decision decoding of Reed-Solomon codes*, Proceedings of IEEE Symposium on Information Theory (Yokohama, Japan), 2003, p. 365.

[Koe96a] Ralf Koetter, *Fast generalized minimum distance decoding of algebraic geometric and Reed-Solomon codes*, IEEE Transactions on Information Theory **42** (1996), 721–738.

[Koe96b] ______, *On algebraic decoding of algebraic-geometric and cyclic codes*, Ph.D. thesis, University of Linköping, Sweden, 1996.

[KP04] M. Kuijper and J. W. Polderman, *Reed-Solomon list decoding from a system-theoretic perspective*, IEEE Transactions on Information Theory **50** (2004), no. 2, 259–271.

[KV02] Ralf Koetter and Alexander Vardy, *Decoding of Reed-Solomon codes for additive cost functions*, IEEE Symposium on Information Theory (ISIT) (Lausanne, Switzerland), July 2002.

[KV03a] ______, *Algebraic soft-decision decoding of Reed-Solomon codes*, IEEE Transactions on Information Theory **49** (2003), no. 11, 2809–2825.

[KV03b] ______, *A complexity reducing transformation in algebraic list decoding of Reed-Solomon codes*, Proceedings of IEEE Information Theory Workshop (Paris, France), April 2003, pp. 10–13.

[LO06a] Kwankyu Lee and Michael O'Sullivan, *An interpolation algorithm using Groebner basis for soft-decision decoding of reed-solomon codes*, July 2006, pp. 2032–2036.

[LO06b] ______, *Sudan's list decoding of Reed-Solomon codes from a Groebner basis perspective*, arXiv:math.AC / 0601022 **2** (2006).

[LO07] ______, *Private communications*, January 2007.

[Mas69] James L. Massey, *Shift-register synthesis and BCH decoding*, IEEE Transaction on Information Theory **15** (1969), 122–127.

[McE03a] Robert J. McEliece, *The Guruswami-Sudan decoding algorithm for Reed-Solomon codes*, JPL Interplanetary Network Progress Report (2003), no. 42-153.

[McE03b] ______, *The Guruswami-Sudan decoding algorithm for Reed-Solomon codes*, Private Communications (2003).

[MS81] F. Jessie MacWilliams and Neil J. A. Sloane, *The Theory of Error-Correcting Codes*, Elsevier / North-Holland, Amsterdam, 1981.

[MTV04] Jun Ma, Peter Trifonov, and Alexander Vardy, *Divide-and-conquer interpolation for list decoding of Reed-Solomon codes*, Proceedings of IEEE Symposium on Information Theory (ISIT), June 2004, p. 386.

[MVW06a] Jun Ma, Alexander Vardy, and Zhongfeng Wang, *Efficient fast interpolation architecture for soft-decision decoding of Reed-Solomon codes*, Proceedings of IEEE Symposium on Circuits and Systems(ISCAS), May 2006, pp. 4823–4826.

[MVW06b] ______, *Re-encoder design for soft-decision decoding of an (255,239) Reed-Solomon code*, Proceedings of IEEE Symposium on Circuits and Systems(ISCAS), May 2006, pp. 3550–3553.

[NH98] R. Refslund Nielsen and Tom Høholdt, *Decoding Reed-Solomon codes beyond half the minimum distance*, Proceedings of International Conference on Coding Theory, Cryptography, and Related Areas (Gaunajuato, Mexico), 1998, pp. 221–236.

[Nie01] R. Refslund Nielsen, *List decoding of linear block codes*, Ph.D. thesis, Technical University of Denmark, Denmark, 2001.

[Nie03] R. Refslund Nielsen, *Decoding concatenated codes with Sudan's algorithm*, submitted for publication, 2003.

[OF02] Henry O'Keeffe and Patrick Fitzpatrick, *Groebner basis solutions of constrained interpolation problems*, Linear Algebra Applications (2002), no. 351–352, 533–551.

[OS99] Vadim Olshevsky and M. Amin Shokrollahi, *A displacement structure approach to efficient decoding of Reed-Solomon and algebraic-geometric codes*, Proceedings of 31st ACM Symposium on Theory of Computing (STOC) (Atlanta, GA), May 1999, pp. 235–244.

[Paa94] Christof Paar, *Efficient vlsi architecture for bit-parallel computations in galois field*, Ph.D. thesis, University of Essen, Germany, 1994.

[PV03] Farzad Parvaresh and Alexander Vardy, *Multiplicity assignments for algebraic soft-decision decoding of Reed-Solomon codes*, Proceedings of IEEE Symposium on Information Theory (ISIT) (Yokohama, Japan), July 2003, p. 205.

[PV04] ______, *Multivariate interpolation decoding beyond the Guruswami-Sudan radius*, Proceedings of 42-nd Annual Allerton Conference on Communications, Control and Computing, October 2004.

[PV05] ______, *Correcting errors beyond the Guruswami-Sudan radius in polynomial time*, Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), October 2005, pp. 285–294.

[RR00] Ron M. Roth and Gitit Ruckenstein, *Efficient decoding of Reed-Solomon codes beyond half the minimum distance*, IEEE Transaction on Information Theory **46** (2000), no. 1, 246–257.

[RS60] Irving S. Reed and Gustave Solomon, *Polynomial codes over certain finite fields*, SIAM: Journal of the Society for Industrial and Applied Mathematics **8** (1960), no. 2, 300–304.

[RT03] Olivier Ruatta and Philippe Trebuchet, *A general and efficient approach to implicitization*, 2003.

[Sha95] Igor R. Shafarevich, *Basic Algebraic Geometry*, Springer-Verlag, New York Berlin Heidelberg, 1995.

[SSK03] Berk Sunar, Erkay Savas, and Cetin K. Koc, *Constructing composite field representations for efficient conversion*, IEEE Transaction on Computers **52** (2003), no. 11, 1391–1398.

[Sud97] Madhu Sudan, *Decoding of Reed-Solomon codes beyond the error-correction bound*, Journal of Complexity **13** (1997), no. 1, 180–193.

[SW99] M. Amin Shokrollahi and Hal Wasserman, *List decoding of algebraic-geometric codes*, IEEE Transactions on Information Theory **45** (1999), 432–437.

[TJR01] Trieu-Kien Truong, Jyh-Horng Jeng, and Irving S. Reed, *Fast algorithm for computing the roots of error locater polynomials up to degree 11 in reed-solomon decoders*, IEEE Transaction on Communications **49** (2001), no. 5, 779–783.

[WB86] Lloyd R. Welch and Elwyn R. Berlekamp, *Error correction for algebraic block codes*, US Patent No. 4,633,470, 1986.

[WS01] Xin-Wen Wu and Paul H. Siegel, *Efficient root-finding algorithm with application to list decoding of algebraic-geometric codes*, IEEE Transactions on Information Theory **47** (2001), 2579–2587.

[ZP05] Xinmiao Zhang and Keshab Parhi, *Fast factorization architecture in soft-decision Reed-Solomon decoding*, IEEE Transactions on VLSI Systems **13** (2005), no. 4, 413–426.