Architectures for Arithmetic Operations in Galois Fields $GF(2^m)$

by

Mohamed Ahsan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

June, 1995

Architectures for Arithmetic Operations in Galois Fields $GF(2^m)$

BY

Mohamed Ahsan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

Electrical Engineering

June 1995

ARCHITECTURES FOR ARITHMETIC OPERATIONS IN GALOIS FIELDS $GF(2^m)$

MS Thesis

By

Mohamed Ahsan

June, 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

Mohamed Ahsan

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Thesis Committee :

[illegible]

Dr. Gerhard F. Beckhoff (Chairman)

[illegible] 26/6/95

Dr. Khalid Hussain Biyari (Member)

Essam Hass

Dr. Essam E. M. Hassan (Member)

[illegible]

Dr. Baher Hussein (Member)

S. Zaki Akh[illegible]

Dr. S. Z. Al-Akhdhar (Member)

[illegible]

Dr. Abdallah M. Al-Shehri
Department Chairman

[illegible]

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date 30 · 10 · 95

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

Dedicated to

my Parents and

my Wife

whose prayers, inspiration and love led to this

accomplishment

Acknowledgment

I would like to express my sincere appreciation to the folowing, whose support has been invaluable in the completion of this work

To Dr. G. F. Beckhoff, my thesis advisor; his very presence has been like the proverbial sun, and the constant encouragement and able guidance right through the course of this work has been tremendous, for which I am extremely indebted.

For having spared the time and effort in critically reviewing this work, I thank my committee members, Dr. Khalid Hussain Biyari, Dr. Baher Hussein. Dr. Essam Hassan and Dr. S. Z. Al-Akhdhar.

To all my friends and colleagues at KFUPM: their support and company made my stay infinitely enjoyable, which is gratefully acknowledged.

And lastly a word of grateful thanks to my parents, my wife, my sisters and my brother-in-law who never shackled my freedom and have always allowed me the ultimate discretion in pursuing my options.

Contents

List of Tables

List of Figures

Abstract

Name: Mohamed Ahsan

Title: Architectures for Arithmetic Operations in Galois Fields $GF(2^m)$

Major Field: Electrical Engineering

Date of Degree: June, 1995

Galois fields are used in numerous applications like Reed-Solomon (RS) codes. digital signal processing (DSP) and cryptology. There is a need for efficient multiplication and division methods that can be easliy realised on VLSI chips. Massey and Omura have recently developed a new multiplication algorithm for Galois fields based on the normal basis representation. A new bit-serial modified Massey-Omura multiplier is developed in this thesis to compute multiplications over $GF(2^m)$ In contrast to the existing multipliers. this new multiplier requires the minimum chip area. A serial-in serial-out systolic array is presented for performing element inversion with standard basis represented. The architecture is highly regular. modular and nearest neighbour connected.

Furthermore. a systolic architecture for an RS encoder. based on Cauchy representation of generator matrix of the code. is presented. consisting of $r+1$ cells. where r is the redundancy of the code. This encoder is systematic. does not require any feedback or other global signals. Its cells are of low complexity and it is easily reconfigurable for variable redundancy and changes in the choice of the generator polynomial of the code. the architecture is suitable for very high-speed applications.

Finally. a systolic array of an RS decoder. is presented. Systolic array architectures are derived for the various steps including syndrome calculation. key equation solution and error evaluation. The improvements over existing systolic implementations are discussed.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

June 1995

ملخص البحث

الاسم :- محمد إحسان .

عنوان البحث :- بناء العمليات الحسابية في حقل جالو GF (2^m) .

التخصص :- هندسة كهربائية

تاريخ الدرجة :- يونيـــو ١٩٩٥ م .

حقل جالو يستعمل في كثير من التطبيقات ، مثل " ريدسالو من كود " ، وفي معالجة الموجات الرقمية والتشـفير السـري . هناك حاجة إلى وجود طرق فعالة للضرب والقسمة والتي يمكن من خلالها تنفيذها بسهولة على شرائح " VLSI " . ماسي وأومرا طورا طريقة جديدة للضرب الحسابي لحقل جالو اعتماداً على تمثيل " نورمل Normal " . تحسين ضاربة ماسي - أومرا المتوالية وضحت في هذه الرسالة لحساب عمليات الضرب على حقل جالو . في المقابل للضاربة الموجودة ، هذه الضاربة الجديدة تحتاج إلى أقل مساحة من الشرائح . مصفوفة سستوليك المتوالية عرضت لعمل العنصر المعوكس مع تمثيل"ستاندرد Standard " الأساسي . هذا البناء منتظم جداً وتكراري ومتصل لأقرب جار .

بالإضافة إلى ذلك ، بناء سستوليك لمشفرة ريد سالومن تعتمد على تمثيل كوشي للمصفوفة المولدة المشفرة . هذه المشفرة تحتوي على (ر+١) خلية ، حيث " ر " هو الزائد من الشفرة .

المشفرة منتظمة ولا تحتاج لأي تغذية إسترجاعية . خلاياها تكون باقل تعقيد وسهولة في إعادة تشكيل المجهول الزائد . وتغيير اختيار متعددة الحدود المولدة للشفرة .

هذا البناء مناسب للتطبيقات السريعة جداً .

أخيراً ، مصفوفة سستوليك لحل الشفرة عرضت في هذه الرسالة . بناء مصفوفة سستوليك اشتقت لخطوات متعددة متضمنة حساب " سيندروم " ، حل معادلة المفتاح ، وتقييم الخطا . التحسين على تنفيذ سستوليك الموجود نوقش في هذه الرسالة .

Chapter 1

Introduction

Finite or Galois fields are important in many applications, such as error-correcting codes [1, 2, 3], switching theory [4], cryptography [5] and digital signal processing [6].

1.1 Binary Extension Fields

Galois field $GF(2^m)$ is a number system consisting of 2^m elements, where each element is represented as a vector of m bits. When considered from the hardware point of view, the field $GF(2^m)$ is of particular interest for computer applications. In particular, for digital computers and digital data transmission usually two symbols, 0 and 1 are used. Therefore, in the following text only $GF(2^m)$ fields are considered, for which addition and multiplication are defined as follows :

+	0	1
0	0	1
1	1	0

·	0	1
0	0	0
1	0	1

The addition and multiplication defined above are called *modulo-2 addition* and *multiplication* respectively. It should be noted that $1 = -1$ since $1 + 1 = 0$. The alphabet of two symbols, 0 and 1, together with modulo-2 addition and multiplication is called a *field* of two elements (or a binary field), which is usually denoted by $GF(2)$.

Definition 1.1 *Consider a polynomial $F(x)$, with one indeterminate x and with coefficients from any field F:*

$$F(x) = f_0 + f_1x + f_2x^2 + \cdots + f_nx^n.$$

The degree of a polynomial is the largest power of x in a term with a nonzero coefficient. The degree of the 0 polynomial is 0.

Definition 1.2 *A polynomial is called monic if the coefficient of the highest power of x is 1. Polynomials can be added and multiplied in the usual way, and they form a ring.*

Definition 1.3 *A polynomial of degree n which is not divisible by any polynomial of degree less than n but is greater than 0 is called irreducible.*

Definition 1.4 *The greatest common divisor of two polynomial is a monic polynomial of greatest degree which divides both of them. Two polynomials are said to be relatively prime if their greatest common divisor is 1.*

Consider computations with polynomials whose coefficients are either 0 or 1. For real numbers, if λ is a root of the polynomial $f(x)$ (that is, $f(\lambda) = 0$), $f(x)$ is divisible by $x - \lambda$. This is still true for $f(x)$ with binary coefficients. The only polynomials of degree 1 are x and $x + 1$. The only polynomials of degree 2 are x^2, $x^2 + x$, $x^2 + 1$ and $x^2 + x + 1$. Of these polynomials of degree 2, x^2 and $x^2 + x$ factor in an obvious way. Since $1^2 + 1 = 0$, $x^2 + 1$ must be divisible by $x + 1$. In fact, $(x + 1)^2 = x^2 + x + x + 1 = x^2 + 1$. However, $x^2 + x + 1$ does not have 0 or 1 as roots and so is not divisible by any polynomial except 1 and itself. A polynomial $F(x)$ of degree m is said to be *irreducible* over a binary field $GF(2)$ if $F(x)$ is not divisible by any polynomial of degree less than m and greater than zero. It can be proved that $x^4 + x + 1$ is irreducible by noting that if it is not irreducible, then there must be irreducible factor of degree equal to or less than 2. Since 0 or 1 are not roots, x and $x + 1$ are not factors. The only remaining possibility is $x^2 + x + 1$. it can be verified by long division that $x^2 + x + 1$ is not divisible by $x^2 + x + 1$. Thus, $x^4 + x + 1$ is irreducible.

Fields with 2^m symbols are called *Galois fields*, and they are denoted by $GF(2^m)$. They are important in the study of cyclic codes. In particular, they are used in decoding BCH codes and as symbols in $Reed - Solomon$ codes.

Definition 1.5 *In general, in any $GF(q)$, there is primitive element α, that is, an element of order $q-1$. Every nonzero element of $GF(q)$ can be expressed as a power of α; i.e., the multiplicative group of $GF(q)$ is cyclic.*

Definition 1.6 *An irreducible polynomial of degree m over $GF(2)$ is called primitive if it has a primitive element α from $GF(2^m)$ as a root. Then this root and hence all the roots (i.e.,α, α^2, $\cdots$ $\alpha^{2^{m-1}}$) are all primitive.*

An arithmetic field with 2^m symbols is derived as follows. First we start with an arithmetic with two symbols and an irreducible polynomial $F(x)$ of degree m over $GF(2)$. Next we introduce a new symbol, α, and assuming that $F(\alpha) = 0$. Then a table of powers of α is developed. If $F(x)$ is chosen properly, i.e., if it is irreducible in $GF(2)$, then the powers of α up to $2^m - 2$ will be different, $\alpha^{2^m-1} = 1$, and 0, 1, α, $\alpha^2, \cdots, \alpha^{2^m-2}$ will be the set of 2^m field elements. Furthermore, each element can be expressed as a sum of the elements 1, α, α^2, $\cdots$, α^{m-1}. The nonzero elements form a cyclic abelian group under multiplication, a generator of this group is the primitive element of $GF(2^m)$. For example, for m=3, $F(x) = x^3 + x + 1$ generates the field elements as shown in Table 1.1.

Exponent	Polynomial	Binary	Decimal
0		000	0
a^0	1	001	1
a^1	a	010	2
a^2	a^2	100	4
a^3	$a+1$	011	3
a^4	a^2+a	110	6
a^5	a^2+a+1	111	7
a^6	a^2+1	101	5

Table 1.1: Representation of elements of $GF(2^3)$

In general, any element of $GF(2^m)$ whose powers generate all the nonzero elements of $GF(2^m)$ is said to be primitive. It has been proved that for each positive integer m there exists at least one primitive polynomial of degree m. It is not easy to recognise a primitive polynomial. However, there are tables of polynomials in which primitive polynomials are indicated. A list of primitive polynomials are given in [1].

Definition 1.7 *Let β be an arbitrary element of the Galois field $GF(2^m)$. The polynomial $m(x)$ of smallest degree with binary coefficients such that $m(\beta) = 0$ is called the minimal polynomial of β. The minimal polynomial of β is irreducible.*

This can be seen as follows: Suppose that $m(x)$ is not irreducible, say $m(x) = m_1(x)m_2(x)$, where $m_1(x)$ and $m_2(x)$ are nontrivial. Both $m_1(x)$ and $m_2(x)$ have degree lower than the degree of $m(x)$. Since $m(\beta) = m_1(\beta)m_2(\beta) = 0$, then either $m_1(\beta)$ or $m_2(\beta)$ must be zero. This contradicts the hypothesis that $m(x)$ is the polynomial of the smallest degree such that $m(\beta) = 0$. Therefore , $m(x)$ cannot have nontrivial factors and it must be irreducible. It has been proved that every element of $GF(2^m)$ has a minimum polynomial whose degree is m or less. Since $m(\beta) = 0$ and $[m(x)]^{2^l} = m(x^{2^l})$, then

$$[m(\beta)]^{2^l} = m(\beta^{2^l}) = 0.$$

This is to say that β^{2^l} is also a root of $m(x)$. Consequently,

$$\beta, \beta^2, \beta^{2^2}, \cdots, \beta^{2^l}, \cdots$$

are all roots of $m(x)$. Since $m(x)$ has finite degree, it must have a finite number of roots. Thus, there must be a repetition in the above sequence. Let e be the degree of $m(x)$. It can be shown that $\beta, \beta^2, \beta^{2^2}, \cdots, \beta^{2^e-1}$ are all the distinct roots of $m(x)$. These elements repeat in the sequence after β^{2^e-1}.

Definition 1.8

All-One Polynomial

A polynomial $p(x) = x^m + x^{m-1} + \cdots + x + 1$ *over* $GF(2)$ *is called all one polynomial (AOP) of degree* m.

Attention is restricted to AOP's over $GF(2)$ only. An AOP of degree m is irreducible over $GF(2)$ if $(m+1)$ is a prime and 2 is the generator of $GF^*(m+1)$, where $GF^*(m+1)$ is the multiplicative group in $GF(m+1)$. Table1.2 shows some possible values of m for an AOP of degree m to be irreducible [7]. Irreducible AOPs are special irreducible polynomials, having the following interesting property : Let $p(x)$ be an irreducible AOP of degree m, then the roots $\{\alpha, \alpha^2, \alpha^{2^2}, \cdots, \alpha^{2^m-1}\}$, of $p(x) = 0$, are linearly independent over $GF(2)$. It should be noted that the set of roots of $p(x) = 0$ constructs a normal basis [3] over $GF(2)$, where $p(x)$ is an irreducible AOP over $GF(2)$.

2	28	66	138
4	36	82	148
10	52	100	162
12	58	106	172
18	60	130	178

Table 1.2: Examples of m

Arithmetic operations including the ones discussed above can be implemented in software but the speed of computations is not satisfactory. Thus, to achieve a desired rate of output, several dedicated circuits have been proposed. These are linear feedback shift register (LFSR) circuits [1, 2]. Recently many systolic architectures to perform arithmetic computations in $GF(2^m)$ have been proposed in literature. The purpose of this thesis is to study various arithmetic operations in $GF(2^m)$ and the possible systolic circuits that implement them. The application of these circuits to encode and decode Reed-Solomon codes is studied.

1.2 Bases of $GF(2^m)$ over $GF(2)$

1.2.1 Standard Basis

$GF(2^m)$ is a vector space of dimension m over the ground field $GF(2) = \{0, 1\}$. Any set of m linearly independent elements can be used as a basis for this vector space. In constructing $GF(2^m)$ from a primitive irreducible polynomial $F(x)$, the basis used is $1, \alpha, \alpha^2, \cdots, \alpha^{m-1}$, where α is a root of the primitive irreducible polynomial. This basis is known as the *standard basis representation* also called as *conventional basis representation* or *polynomial basis representation* of field elements. The elements of $GF(2^m)$ can be represented using the standard basis representation. Any element can be expressed as a polynomial of α with degree less than m. That is,

$$GF(2^m) = \{a_{m-1}\alpha^{m-1} + \cdots + a_1\alpha + a_0 | a_i \in GF(2) \text{ for } 0 \leq i \leq m-1\}.$$

Table 1.1 demonstrates an example of the standard basis representation of the field elements.

1.2.2 Normal Basis

A normal basis of $GF(2^m)$ over $GF(2)$ is of the form $\lambda, \lambda^2, \lambda^{2^2}, \cdots, \lambda^{2^{m-1}}$. It has been proved in [3] that a normal basis exists in any field $GF(q^m)$ and any field $GF(q^m)$ contains a primitive element λ such that $\lambda, \lambda^q, \cdots, \lambda^{q^{m-1}}$ is a normal basis. An element $b \in GF(2^m)$ can be represented in normal basis as

$$b = b_0\alpha + b_1\alpha^2 + \cdots + b_{m-1}\alpha^{2^{m-1}}.$$

where $b_i \in GF(2^m)$ $(0 \le i \le m-1)$. If m is odd, $GF(2^m)$ has a self- complementary normal basis. $GF(2^m)$ contains a primitive element of trace 1 or conversely, according to [8] a necessary and sufficient condition for $\lambda, \lambda^2, \lambda^{2^2}, \cdots, \lambda^{2^{m-1}}$ to be a normal basis of $GF(2^m)$ is

$$T_m(\lambda) = \lambda + \lambda^2 + \lambda^{2^2} + \cdots + \lambda^{2^{m-1}} = 1.$$

This relation implies that the normal basis representation of 1 is (1,1,1,....,1).

1.3 Systolic Arrays

Systolic arrays are integrated, special purpose processors whose dominant characteristics are:

- massive, decentralised parallelism
- local communication
- synchronous mode of operation.

1.3.1 Parallelism

The concept of parallelism breaks with the classical approach of obtaining speed by performing each operation more rapidly. In parallel computation, the speed increase comes from the simultaneous execution of operations. Multiprocessors are computers which have more than one processor. The architecture of the multiprocessor is more complex than that of the sequential machine: the problems of memory access become crucial because data have to be supplied to processors at the right time; similarly the problems of communication and synchronisation between processors are important.

1.3.2 The Locality Principle

When one tries to make best use of a parallel architecture for executing a pre-existing algorithm, the principal problem is that of distributing the algorithm across the processors in such a way that they will be actively engaged in executing the algorithm and will also be doing other *useful* work.

On close examination of the problem one comes to the conclusion that the main way in which one can be successful is not so much to employ fast processors which can perform operations quickly, but rather, to arrange the processors so that they can communicate efficiently. That is, the processors can access information as quickly as they can process it.

Every interconnection network uses the concept of *locality* in some way or another: for a given spatial distribution of processors, it is not possible to connect one processor to another unless they are neighbours (Fig. 1.1). As a consequence, an algorithm can be programmed in an efficient fashion if it can be distributed in such a way that each processor need only communicate with its neighbours.

As a second consequence of the local connection method concerns the relationship between I/O and computation. Since it is not possible to connect each processor to every other, it is, in the same way, difficult to imagine that each processor could be connected directly to the machine's external environment (that is to the *host computer system*), and it is the external environment that is supposed to supply data to the algorithm and to gather its results. In one way or another, this shows that the complexity of an algorithm must reside in its component operations and not in the number of I/O operations that have to be performed.

1.3.3 The Systolic Model

A systolic architecture is organised as a network composed of a large number of identical, elementary cells which are locally connected. Each cell receives data from its neighbouring cells, performs a simple calculation and then transmits its results

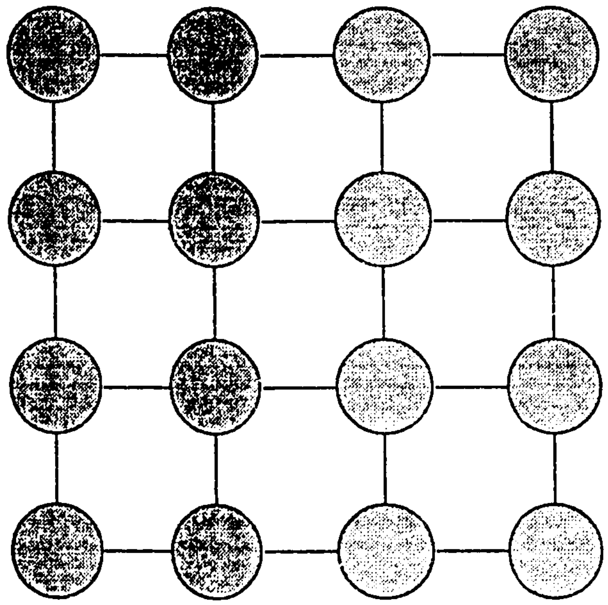


Figure 1.1: Locally connected network of processors

(again, only to its neigbours) one cycle later. Only those cells that are at the edge of the network communicate with the external world. To fix an order of size, it can be assumed that each cell is about as complex as a microprocessor.

The cells operate in parallel under the control of a global clock (i.e., they are totally synchronised): several computations are performed at the same time by the network, and several instances of the same problem can be pipelined over the network.

The name systolic comes from an analogy between the circulation of data streams in the network and the circulation of blood in the human vascular circulatory system. The clock which maintains synchronisation is the 'heart' of the system.

1.4 Literature Review

Important arithmetic operations in finite fields include addition, subtraction, multiplication, division, exponentiation and number inversion. Among these, addition which is the same as subtraction is the simplest one and inversion has been identified as the most complicated and lengthy operation [9].

Multiplication in $GF(2^m)$ is completely systolisable. One of the first studies on systolic multipliers for finite fields was published by Yeh et al.[10]. They developed a serial-in-serial-out systolic (SISOS) array and a parallel-in-parallel-out systolic (PIPOS) array for implementation of $GF(2^m)$ multipliers using the standard basis representations. Both are well suited to VLSI implementation, but the former (SISOS) requires two control signals and the latter (PIPOS) has contra flowing data streams. A system with unidirectional data flow gains advantages over a system with contra flow in terms of chip cascadability, fault tolerance and possible wafer scale integration. The systolic architecture described in [11] is also suitable for implementation using VLSI techniques, but involves broadcasting in data flow. It is often desirable to avoid broadcasting when designing a high-speed system. Wang and Lin [12] propose a parallel-in-parallel-out systolic array and a serial-in-serial-out systolic array for implementation of $GF(2^m)$ multipliers using the standard basis representation, both of which have unidirectional data flow. The parallel form structure incorporates fault-tolerant design and the serial form structure requires only one control signal. Zhou [13] proposes a new bit-serial systolic array to compute multiplication over $GF(2^m)$ which requires one control signal.

Hasan and Bhargava [9] have studied division over $GF(2^m)$. Their work has been extended by Flenn et al.[14]. A systolic solution to the problem of division has been proposed by Wang and Lin [15]. Hasan and Bhargava [9] propose bit-serial systolic divider and multiplier for finite fields $GF(2^m)$

The problem of encoding of Reed-Solomon(RS) codes using systolic arrays has been studied by Seroussi [16]. One of the first studies on decoding of RS codes using systolic arrays has been published by Shao *et al.* [17]. They use the modified Euclidean algorithm to solve the key equation. Kimura *et al.* [18] propose a systolic solution to the problem of RS decoding. They use the extended Euclidean algorithm to solve the key equation. Their solution is based on the work proposed by Brent *et al.* [19]. Several others [20, 21, 22] have proposed other *nonsystolic* versions to the problem of RS decoding. Nelson *et al.* [23] propose a systolic RS decoder which uses the extended Euclidean algorithm to solve the key equation. They use the systolic array proposed by Fitzpatrick *et al.* [24] to solve the key equation. The RS decoder proposed in [23] is least complex in terms of area and time. The systolic array of [24] to solve the key equation has been improved by Doyle *et al.* [25].

Chapter 2

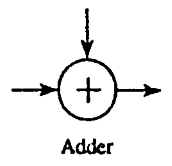
Architectures Using Linear Shift Register Circuits

Galois field arithmetic can be implemented more easily than ordinary arithmetic because there are no carries. Several circuits have been proposed in [1. 3. 2] to perform the arithmetic operations in Galois fields.

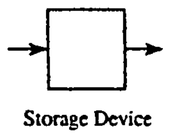
2.1 Linear Switching Circuits

In linear switching circuits, information is assumed to be some representation of elements of $GF(2^m)$. Three basic types of devices are used as shown in Fig 2.1.

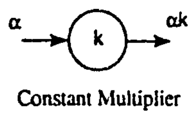
A linear finite-state switching circuit is any circuit consisting of a finite number of adders, storage devices, and constant multipliers connected in a permissible way.



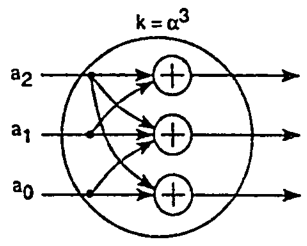
Adder can be implemented using an XOR gate.



The storage device is a D-type flip-flop.



For binary case, the constant multiplier is as follows:
1 $\longrightarrow$ Connection
0 $\longrightarrow$ No connection



Constant Multiplier for k = α^3

For nonbinary case, it can be an ordinary m-input m-output binary logic circuit, whose design is dependent on the irreducible primitive polynomial.
e.g., for GF(2^3) using $F(x) = x^3 + x + 1$.
Let $k = \alpha^3$. The input β can be represented as a polynomial
$\beta = a_2\alpha^2 + a_1\alpha + a_0$. Therefore,
$\alpha^3(a_2\alpha^2 + a_1\alpha + a_0) = (a_2+a_1)\alpha^2 + (a_1+a_2+a_0)\alpha + (a_2+a_0)$.

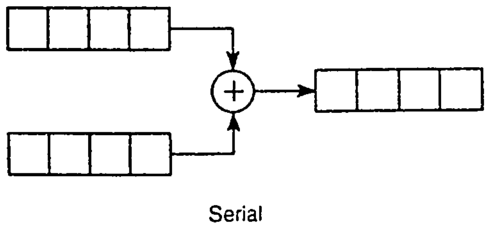
Figure 2.1: The building blocks for linear switching circuits.

2.2 Adders for $GF(2^m)$

Addition in $GF(2^m)$ is bit independent, straight-forward and is easily realised by m independent XOR gates. In order to add two field elements, their vector representations are added and the resultant vector is the vector representation of the sum of the two field elements.

Example 2.1 To add α^7 and α^{13} of $GF(2^4)$. Their vector representations are (1101) and (1011), respectively. Their vector sum is $(1101) + (1011) = (0110)$, which corresponds to the exponential representation of α^5.

Addition can be performed both serially and in parallel. Fig. 2.2 shows a circuit for addition in $GF(2^4)$ for each of the two cases: serial and parallel data flow. In each case, the addends are in shift registers to start with, and the sum is in a third shift register at the conclusion of the addition. The circuit for serial addition requires four clock times to complete the addition; the circuit for parallel addition requires only one clock time but has more wires and modulo-2 adders. If desired, the addends could be fed back to the inputs of their own registers so that at the end of the addition they could again be found in their registers for other purposes.



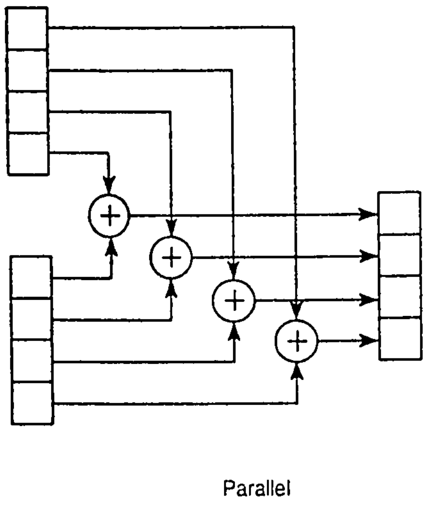


Figure 2.2: Addition of two field elements of $GF(2^4)$.

2.3 Multipliers for $GF(2^m)$

2.3.1 Multiplication of a Field Element by a Fixed Field Element

Let β be a field element in $GF(2^m)$, then it can be represented as a polynomial in α as follows:

$$\beta = b_{m-1}\alpha^{m-1} + \cdots + b_1\alpha + b_0$$

Example 2.2 Let us multiply β in, say $GF(2^4)$, by the primitive element α, whose minimal polynomial is $F(x) = x^4 + x + 1$.

Then β should be expressed as a polynomial in α as follows:

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3.$$

Multiplying both sides of the equality by α and using the fact $\alpha^4 = \alpha + 1$, the following equality is obtained:

$$\begin{aligned} \alpha\beta &= b_0\alpha + b_1\alpha^2 + b_2\alpha^3 + b_3\alpha^4 \bmod (\alpha^4 + \alpha + 1) \\ &= b_3 + (b_0 + b_3)\alpha + b_1\alpha^2 + b_2\alpha^3. \end{aligned}$$

This multiplication can be carried out by the feedback shift register shown in Fig. 2.3. First, the vector representation of (b_0, b_1, b_2, b_3) of β is loaded into the register. Then the register is pulsed once. The new contents in the register form the

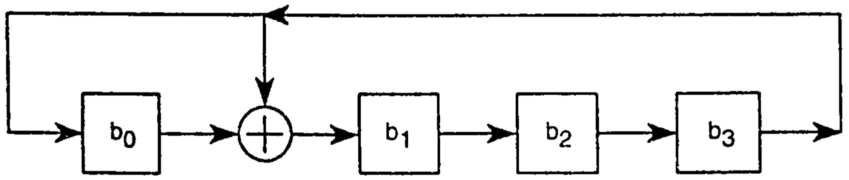


Figure 2.3: Serial multiplication implementation of $\alpha \cdot \beta$.

vector representation of $\alpha\beta$. The equivalent parallel multiplication is given in Fig. 2.4.

Example 2.3 To multiply an arbitrary element β of $GF(2^4)$ by the element α^3.

Again β is expressed in polynomial form as

$$\beta = b_0 + b_1\alpha + b_2\alpha^2 + b_3\alpha^3.$$

Multiplying both sides of the equation by α^3, the following equation is obtained:

$$\begin{aligned}
\alpha^3\beta &= b_0\alpha^3 + b_1\alpha^4 + b_2\alpha^5 + b_3\alpha^6 \\
&= b_0\alpha^3 + b_1(1+\alpha) + b_2(\alpha+\alpha^2) + b_3(\alpha^2+\alpha^3) \\
&= b_1 + (b_1+b_2)\alpha + (b_2+b_3)\alpha^2 + (b_0+b_3)\alpha^3.
\end{aligned}$$

Based on the expression above, the circuit obtained is shown in Fig. 2.5. Multiplication is carried out by first loading the vector representation (b_0, b_1, b_2, b_3) of β into the register. On pulsing the register, the new contents of the register are the vector representation of $\alpha^3\beta$. The serial implementation uses the same circuit as given in Fig.2.3. This time the register is pulsed three times. The new contents in the register form the vector representation of the product $\alpha^3\beta$.

2.3.2 Multiplication of any Polynomial by a Fixed Polynomial

The circuit shown in Fig. 2.6 is known as linear feed-forward shift register in digital system theory and is also known as a finite impulse response (FIR) filter in digital

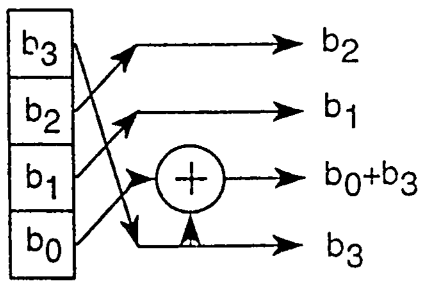


Figure 2.4: Parallel multiplication implementation of $\alpha \cdot \beta$.

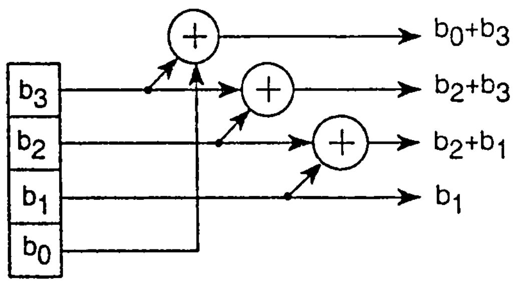
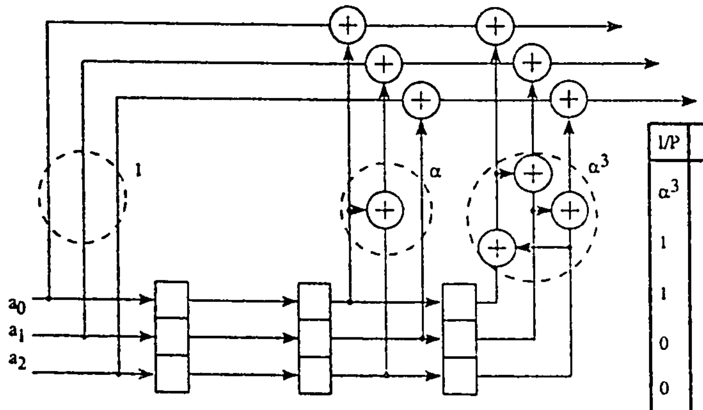
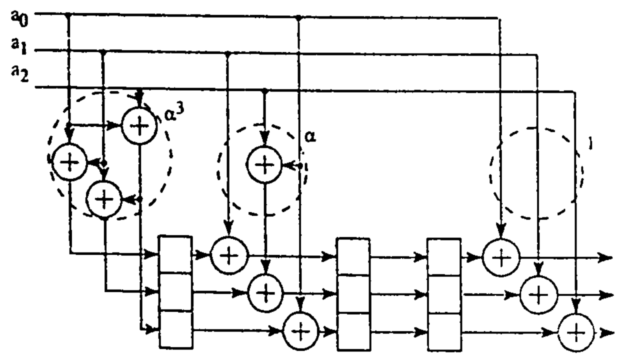


Figure 2.5: Parallel multiplication implementation of $\alpha^3 \cdot \beta$.



Type 1 multiplier

I/P	State	O/P
α^3	0 0 0	α^3
1	α^3 0 0	1
1	1 α^3 0	α^5
0	1 1 α^3	α^5
0	0 1 1	1
0	0 0 1	α^3
0	0 0 0	0



Type 2 multiplier

I/P	State	O/P
α^3	0 0 0	α^3
1	α^6 α^4 0	1
1	α^3 α^5 α^4	α^5
0	0 α^3 0	α^5
0	0 0 α^3	1
0	0 0 0	α^3

Figure 2.6: Shift Register Circuits for multiplying polynomials.

signal processing (DSP). It multiplies any input polynomial

$$a(x) = a_k x^k + a_{k-1} x^{k-1} + \cdots + a_1 x + a_0, \quad a_i \in \{0,1\}$$

by the fixed polynomial

$$h(x) = h_r x^r + h_{r-1} x^{r-1} + \cdots + h_1 x + h_0, \quad h_i \in \{0,1\}.$$

Example 2.4 Let the fixed multiplier be $x^3 + \alpha x + \alpha^3$ and the variable multiplicand be $\alpha^3 x^2 + x + 1$.

The storage devices are assumed to contain 0's initially, and the coefficients of $a(x)$ are assumed to enter high-order first.

Another circuit for multiplication is shown in Fig.2.6, which is an unconventional form of an FIR filter. The number of shifts required to obtain the complete product is equal to the sum of the degrees of the input polynomial added to 2 ($= 7$ in this case).

2.3.3 Two Input Multiplier

Circuits of the type shown above can have more than one input. The circuit shown in Fig. 2.7 has two inputs.$a_1(x)$ and $a_2(x)$, and the output is

$$b(x) = a_1(x)h(x) + a_2(x)k(x),$$

where

$$h(x) = h_r x^r + h_{r-1} x^{r-1} + \cdots + h_0.$$

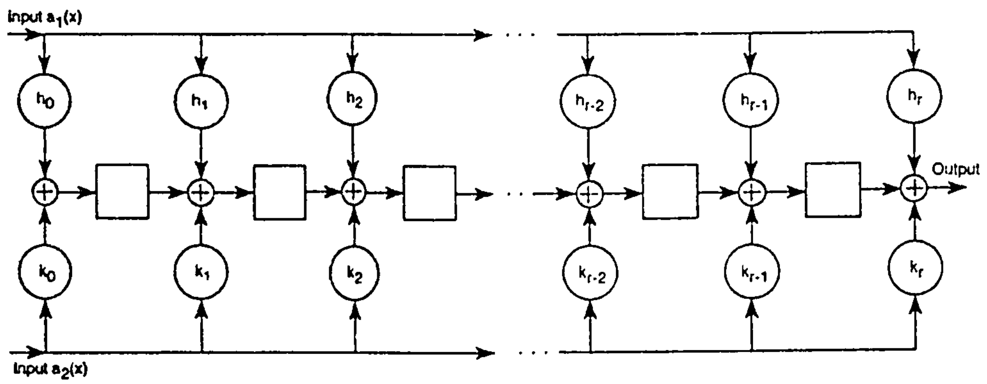


Figure 2.7: A two input multiplier

$$k(x) = k_r x^r + k_{r-1} x^{r-1} + \cdots + k_0,$$

The circuit is shown as if $h(x)$ and $k(x)$ have the same degree, but in case the degrees are not equal, r can be taken as the larger degree, and the high order coefficients of one polynomial can be 0.

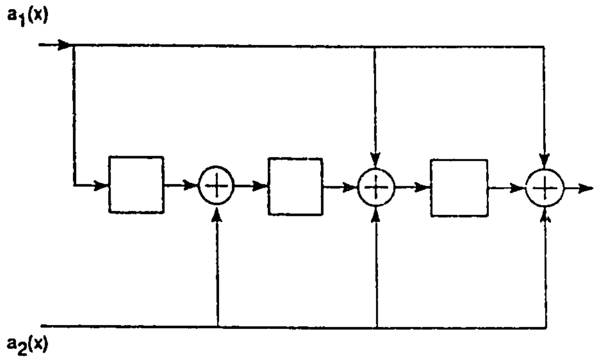
It is to be noted that in these circuits, the coefficient of the x^{r+i} comes at the same time that the coefficient of x^i goes in. The coefficient of x^r in the product comes out r units of time after the coefficients of x^0 enters the input. Thus, in a sense, the output is delayed r units of time.

Example 2.5 Let $h(x) = x^3 + x + 1$ and $k(x) = x^2 + x + 1$ be the fixed polynomials and the inputs be $a_1(x) = x^2 + x + 1$ and $a_2(x) = x^3 + x + 1$ respectively.

The resulting two input multiplier circuit is shown in Fig. 2.8.

2.3.4 Proposed Serial Modified Massey-Omura Multiplier

Massey and Omura developed a multiplier which obtains the product of two elements in the finite field $GF(2^m)$. They utilise a normal basis of form $\{\alpha, \alpha^2, \alpha^4, \cdots, \alpha^{2m-1}\}$ to represent each element in the field, where α is the root of an irreducible polynomial of degree m over $GF(2)$. In this basis, each element in the field $GF(2^m)$ can be represented by m binary digits.



a1	a2	Shift Reg	Output
1	1	000	0
1	0	110	0
1	0	101	0
0	1	111	0
0	0	000	0
0	0	000	0

Figure 2.8: A two input multiplier with $h(x) = x^3 + x + 1$ and $k(x) = x^2 + x + 1$.

Preliminaries

Multiplication in the normal basis representation requires for any one product digit the same logic circuitry as it does for any other product digit. Adjacent product-digit circuits differ only in their inputs, which are cyclically shifted versions of one another. Wang *et al.* [26] have developed a pipeline architecture for a Massey-Omura Multiplier (MOM) on $GF(2^m)$ suitable for VLSI design.

Representing an element of $GF(2^m)$ in normal basis as

$$b = b_0\alpha + b_1\alpha^2 + \cdots + b_{m-1}\alpha^{m-1}$$

where $b_i \in GF(2)$ $(0 \leq i \leq m-1)$ is the ith coordinate of b. In vector notation,

$$b = \mathbf{b}\alpha^{\mathbf{t}}.$$

where $\mathbf{b} = [b_0, b_1, \cdots, b_{m-1}]$, $\alpha = [\alpha, \alpha^2, \cdots, \alpha^{2^{m-1}}]$, and t denotes the vector transposition.

Let d be the product of any two elements b and c of $GF(2^m)$, i.e.,

$$d = \mathbf{b}\alpha^{\mathbf{t}}(\mathbf{c}\alpha^{\mathbf{t}})^{\mathbf{t}} = \mathbf{bMc^t} \tag{2.1}$$

where the *multiplication matrix* $\mathbf{M}$ is defined by

$$\alpha^{\mathbf{t}}\alpha = \begin{bmatrix} \alpha^{2^0+2^0} & \alpha^{2^0+2^1} & \cdots & \alpha^{2^0+2^{m-1}} \\ \alpha^{2^1+2^0} & \alpha^{2^1+2^1} & \cdots & \alpha^{2^1+2^{m-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{2^{m-1}+2^0} & \alpha^{2^{m-1}+2^1} & \cdots & \alpha^{2^{m-1}+2^{m-1}} \end{bmatrix}. \tag{2.2}$$

We may write M as

$$\mathbf{M} = \mathbf{M}_0\alpha + \mathbf{M}_1\alpha^2 + \cdots + \mathbf{M}_{m-1}\alpha^{2^{m-1}}. \tag{2.3}$$

where the entries of the Boolean matrix $\mathbf{M_k}$ belong to $GF(2)$. In $\mathbf{M_k}$, each row and column is numbered as $0, 1, \cdots, m-1$. Then entry (i, j) of $\mathbf{M_k}$ contains the coefficient of α^{2^k} when $\alpha^{2^i+2^j}$ is expressed in terms of the normal basis using $k = 0, 1, \cdots, m-1$. From eqn. 2.1 and eqn. 2.3, one obtains the coordinates of the product as

$$\begin{aligned} d_{m-k-1} &= \mathbf{b}\mathbf{M}_{m-k-1}\mathbf{c}^t, \qquad k = 0, 1, \cdots, m-1 \\ &= \mathbf{b}^{(k)}\mathbf{M}_{m-k}\mathbf{c}^{(k)t} \end{aligned} \tag{2.4}$$

where $\mathbf{b}^{(k)}$ is the k-fold right cyclic shift of $\mathbf{b}$. A parallel multiplier based on eqn. 2.4 is shown in Fig. 2.9 and is known as a parallel-type Massey-Omura multiplier. These are m identical blocks for the m coordinates of the product d. The adjacent blocks differ only in their inputs, which are cyclically shifted versions of one another. The complexity of the module and, consequently, that of the multiplier depends on the irreducible polynomial generating the field.

A Modified Parallel Massey-Omura Multiplier

In this section, based on the work presented in [27], finite fields $GF(2^m)$ generated by irreducible AOP are considered which lead to low complexity multipliers. Let $A_m(x)$ be the irreducible AOP over $GF(2)$. If $\alpha \in GF(2)$ satisfies $A_m(\alpha) = 0$, then $\alpha, \alpha^2, \cdots, \alpha^{2^{m-1}}$ are linearly independent roots of $A_m(x)$ and form a normal basis for

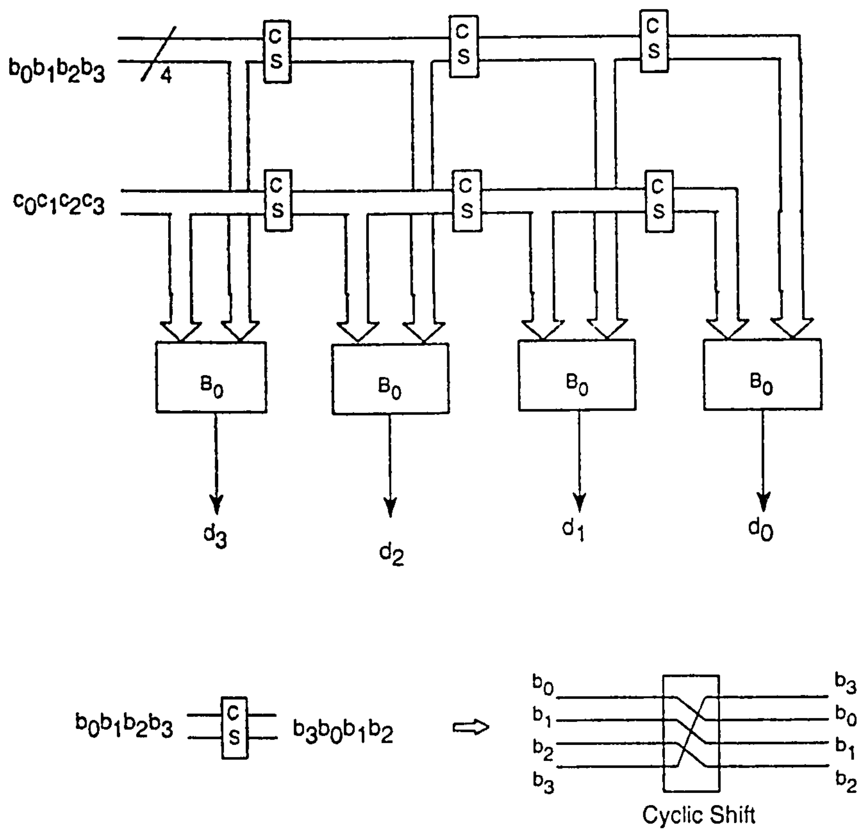


Figure 2.9: A structure for the Massey-Omura parallel multiplier over $GF(2^4)$.

$GF(2^m)$. This normal basis is an *optimal normal basis* in the sense that it results in the lowest number of 1's in $\mathbf{M_{m-1}}$ and yields a low complexity multiplier [7]. The distribution of 1's in $\mathbf{M_{m-1}}$ is given in the following theorem.

Theorm 2.1 *The number of 1's in the matrix* $\mathbf{M_{m-1}}$ *of an irreducible polynomial of degree m is* $2m-1$. *These 1's appear at row i and column j satisfying*

$$2^{((i-(m/2)+1))} + 2^{((j-(m/2)+1))} = 0, \text{ or } m \bmod (m+1), \tag{2.5}$$

where $((i))$ *denotes i mod m.*

Using the above theorem, it can be shown that $i = ((\frac{m}{2}+j))$ always satisfies eqn.2.5 [46]. Separating these m 1's, one can express $\mathbf{M_{m-1}}$ as a sum of two matrices, i.e.,

$$\mathbf{M_{m-1}} = \mathbf{P} + \mathbf{Q} \pmod 2. \tag{2.6}$$

where the entry at (i,j) of $\mathbf{P} \triangleq [p_{i,j}]_{i,j=0}^{m-1}$ is given as follows:

$$p_{i,j} = \begin{cases} 1, & \text{if } i = ((\frac{m}{2}+j)), \\ 0, & \text{otherwise.} \end{cases} \tag{2.7}$$

Referring to eqn. 2.4, one can write the shifted vector $\mathbf{b}^{(k)}$ as

$$\mathbf{b}^{(k)} = \mathbf{b}\mathbf{T}^{(k)}. \tag{2.8}$$

where the entry at (i,j) of matrix $\mathbf{T}^{(k)} \triangleq [t_i^{(k)}]_{i,j=0}^{m-1}$ is

$$t_{i,j}^{(k)} = \begin{cases} 1, & \text{if } i = ((j-k)), \\ 0, & \text{otherwise.} \end{cases} \tag{2.9}$$

Now it can be easily seen that for any integer k

$$\mathbf{T}^{(k)}\mathbf{P}\mathbf{T}^{(k)t} = \mathbf{P}. \tag{2.10}$$

Using equations 2.4, 2.6, 2.8 and 2.10, for $0 \leq k \leq m-1$, we obtain

$$\begin{aligned} d_{m-1-k} &= \mathbf{b}\mathbf{T}^{(k)}\mathbf{P}\mathbf{T}^{(k)t}\mathbf{c}^t + \mathbf{b}^{(k)}\mathbf{Q}\mathbf{c}^{(k)t} \pmod 2 \\ &= \mathbf{b}\mathbf{P}\mathbf{c}^t + \mathbf{b}^{(k)}\mathbf{Q}\mathbf{c}^{(k)t} \pmod 2 \\ &\triangleq \tilde{d} + \tilde{d}_{m-1-k} \pmod 2. \end{aligned} \tag{2.11}$$

Since $\tilde{d}$, which is independent of k, is present in each coordinate d_{m-1-k}, it needs to be computed only once for $k = 0, 1, \cdots, m-1$, resulting in a reduction in the circuit complexity of a parallel-type multiplier.

Example 2.6 Consider the irreducible all-one polynomial of degree $m = 4$, i.e., $A_4(x) = x^4 + x^3 + x^2 + x + 1$. Since the order of the root α of the irreducible polynomial is $m+1$, the multiplication matrix $\mathbf{M}$ becomes

$$\begin{aligned} \mathbf{M} &= \begin{bmatrix} \alpha^2 & \alpha^3 & \alpha^5 & \alpha^9 \\ \alpha^3 & \alpha^4 & \alpha^6 & \alpha^{10} \\ \alpha^5 & \alpha^6 & \alpha^8 & \alpha^{12} \\ \alpha^9 & \alpha^{10} & \alpha^{12} & \alpha^{16} \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^3 & 1 & \alpha^4 \\ \alpha^3 & \alpha^4 & \alpha & 1 \\ 1 & \alpha & \alpha^3 & \alpha^2 \\ \alpha^4 & 1 & \alpha^2 & \alpha \end{bmatrix} \\ &\triangleq \mathbf{M}_0\alpha + \mathbf{M}_1\alpha^2 + \mathbf{M}_2\alpha^4 + \mathbf{M}_3\alpha^8 \\ &= \mathbf{M}_0\alpha + \mathbf{M}_1\alpha^2 + \mathbf{M}_2\alpha^4 + \mathbf{M}_3\alpha^3. \end{aligned}$$

As the roots α, α^2, α^4, $\alpha^8(=\alpha^3)$ are linearly independent, $\alpha+\alpha^2+\alpha^3+\alpha^4 = 1$ and the Boolean Matrix $\mathbf{M_{m-1}} = \mathbf{M_3}$ is

$$\mathbf{M}_{m-1} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

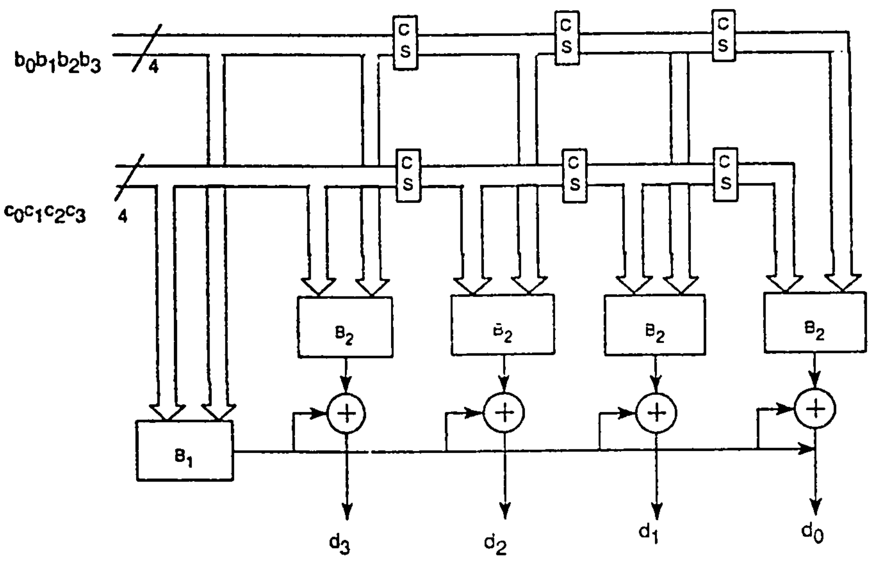
Then

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Finally,

$$\begin{aligned} \tilde{d} &= \mathbf{bPc}' = [b_0,\ b_1,\ b_2,\ b_3]\mathbf{P}[c_0,\ c_1,\ c_2,\ c_3]' \\ \tilde{d}_{m-1-k} &= \mathbf{b}^{(k)}\mathbf{Q}\mathbf{c}^{(k)t} \\ &= [b_{((-k))},\ b_{((-k+1))},\ b_{((-k+2))},\ b_{((-k+3))}] \\ &\quad \cdot\mathbf{Q}[c_{((-k))},\ c_{((-k+1))},\ c_{((-k+2))},\ c_{((-k+3))}]' \quad (0 \le k \le 3) \\ d_{m-1-k} &= \tilde{d} + \tilde{d}_{m-1-k} \quad (0 \le k \le 3). \end{aligned}$$

Fig. 2.10 shows a structure for the parallel multiplier based on eqn. 2.11. The common term $\tilde{d}$ is generated by block B_1, and the coordinate-dependent terms $\tilde{d}_{m-1-k}$ ($k = 0,\ 1,\ \cdots, m-1$) by m blocks of the same kind denoted as B_2. The inputs to two adjacent B_2 blocks are cyclically shifted versions of one another.



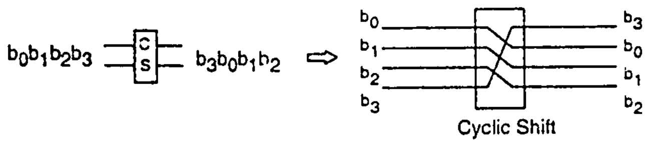


Figure 2.10: A structure for the modified Massey-Omura parallel multiplier over $GF(2^4)$ generated by irreducible all one polynomial of degree m.

Referring to eqn. 2.7, the matrix P has one 1 in each row and column, resulting in m AND gates and $m-1$ XOR gates in B_1. It has been shown in [27] that $\mathbf{M}_{m-1}$ has two 1's in all rows and columns, except the last where there is only one 1. It follows from eqn. 2.6 that matrix Q has $m-1$ 1's which are distributed in $m-1$ rows and $m-1$ columns. Each row and column has one 1, except the last row and column, resulting in $m-1$ AND gates and $m-2$ XOR gates in B_2. The total number of gates for the modified Massey-Omura multiplier is as follows:

$$\text{AND}: \; m + m(m-1) = m^2.$$

$$\text{XOR}: \; m - 1 + m + m(m-2) = m^2 - 1.$$

For comparison, the other parallel-type MOM [26] (hereafter denoted as MOM) of the same class of finite fields $GF(2^m)$, it is generated by an irreducible polynomial and uses normal basis. Like the MOM, the multiplier proposed in [27] (denoted as MOM_M) uses the normal basis. The realisation of both multipliers requires only AND and XOR gates corresponding to mod 2 additions and multiplications. The AND gates, however, can be directly substituted by NAND gates as the XOR of two AND outputs is equivalent to the XOR of two NAND outputs. Although time delays due to gates for the MOM and the MOM_M are the same, the latter requires only about half the XOR gates. The summary of the gate counts and time delays is presented in Table 2.1 [27].

Multiplier	Number of AND gates	Number of XOR gates	Time delay due to gates
MOM	m^2	$2m^2 - 2m$	$D_N + (1 + \lceil \log_2 m - 1 \rceil)D_X$
M_MOM	m^2	$m^2 - 1$	$D_N + (1 + \lceil \log_2 m - 1 \rceil)D_X$

Table 2.1: Comparison of Normal basis multipliers generated by the irreducible AOP of degree m

Proposed Serial Multiplier

For the parallel multiplier of [27], the internal description B_1 and B_2 is not given. A suggested circuit diagram for B_1 and B_2 is given in Fig. 2.11. It satisfies the gate count given in [27].

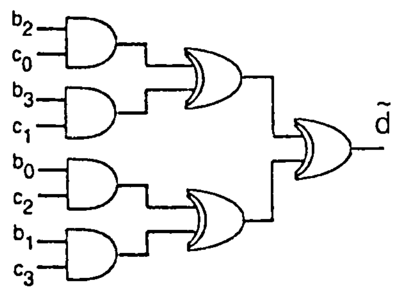
The work in [27] or [7] does not discuss any serial form implementation of the normal basis multiplier. They both use *all one* primitive irreducible polynomials. A suggested serial implementation of this multiplier is shown in Fig. 2.12. The proposed multiplier is also based on an *all one* primitive irreducible polynomial.

Operation and Implementation

It is based on the same principle as the parallel multiplier proposed in [27]. It requires first m bits to load the inputs into the shift registers. Then m shifts are requires to obtain the digits $\tilde{d}_{m-k-1}$ $(0 \leq k \leq m-1$. Additional logic circuitry is required to generate the output $\tilde{d}$ continuously as it is required to add $\tilde{d}$ to each $\tilde{d}_{m-k-1}$ $(0 \leq k \leq m-1$. The additional logic circuitry can be a simple flip-flop (JK flip-flop) which can latch the output at $\tilde{d}$ for rest of the operation.

Conclusions

The proposed multiplier can be realised by simple AND and XOR gates. It can be seen that the serial type multiplier requires the minimum number of gates. The price paid is the the delay in getting the output. It requires m shifts to compute the



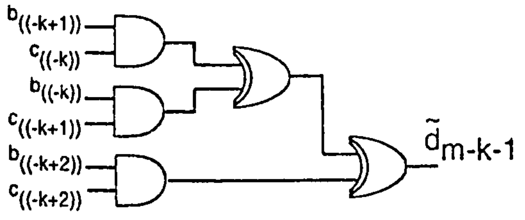


Figure 2.11: Circuit diagram of B_1 and B_2 of the modified Massey-Omura multiplier.

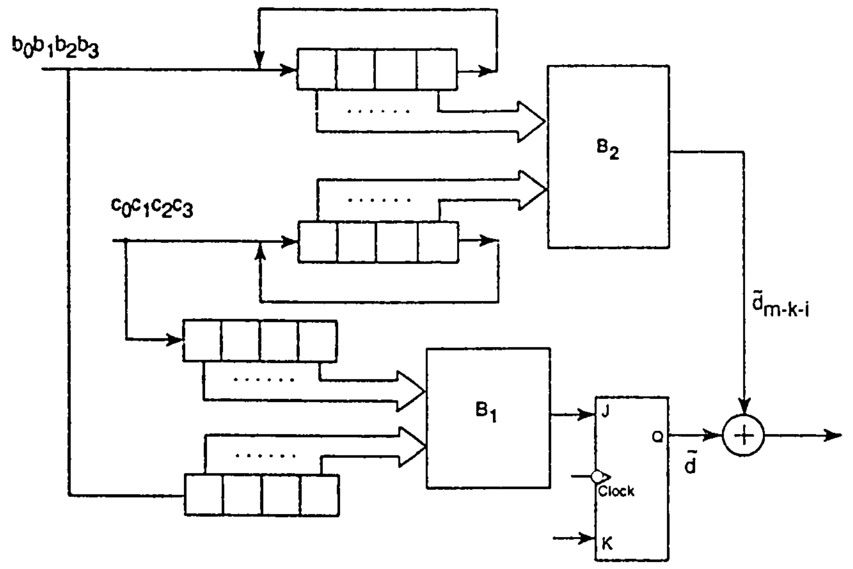


Figure 2.12: Proposed serial implementation of the normal basis multiplier.

product. The maximum switching speed is determined by the delays due to gates, which is this case is given by $D_N + (1 + \lceil \log_2 m - 1 \rceil) D_X$, where D_X is the delay due to an XOR gate and D_N is the delay due to an AND gate. Therefore, the delay due to the gates in computing the product serially is $m\{D_N + (1 + \lceil \log_2 m - 1 \rceil) D_X\}$. All the parameters are tabulated in Table 2.2

2.4 Dividers for $GF(2^m)$

Division in $GF(2^m)$ is performed by linear feedback shift registers (LFSR), also known as *infinite-impulse-response filter* (IIR). A circuit for dividing $d(x) = d_n x^n + d_{n-1}x^{n-1} + \cdots + d_0$ by $g(x) = g_r x^r + g_{r-1}x^{r-1} + \cdots + g_0$ is shown in Fig. 2.13. The storage devices must be set to 0 initially. The output is 0 for the first r shifts, that is, until the input symbol reaches the end of the shift register. Then the first nonzero output appears, and its value is $d_n g_r^{-1}$, the first coefficient of the quotient. For each quotient coefficient q_j, the polynomial $q_j g(x)$ must be subtracted from the dividend. The feedback connections accomplish this subtraction. After a total of n shifts, the entire quotient has appeared at the output, and the remainder is in the shift register.

Example 2.7 Two types of circuits which performs division in $GF(2^3)$ are presented in Figs. 2.14 and 2.15 for dividing the input polynomial $v(x) = \alpha^4 x^6 + \alpha^3 x^3 + \alpha^6 x + \alpha^6$ by the fixed polynomial $g(x) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3$.

Basis used	Normal
Number of AND gates	$2m - 1$
Number of XOR gates	$2m - 2$
Delay due to gates	$D_X + (1 + \lceil \log_2 m - 1 \rceil) D_X$
Number of shifts required	m

Table 2.2: Parameters of the proposed serial multiplier

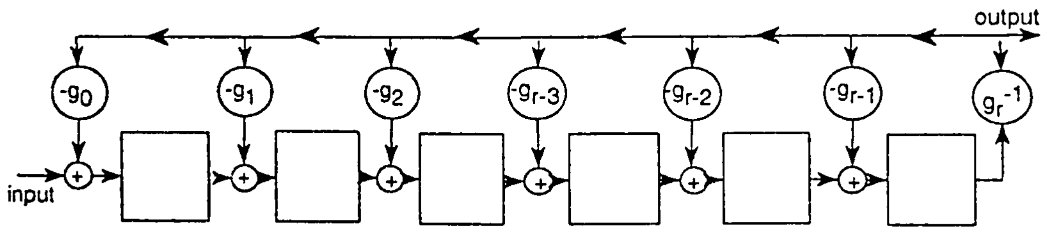
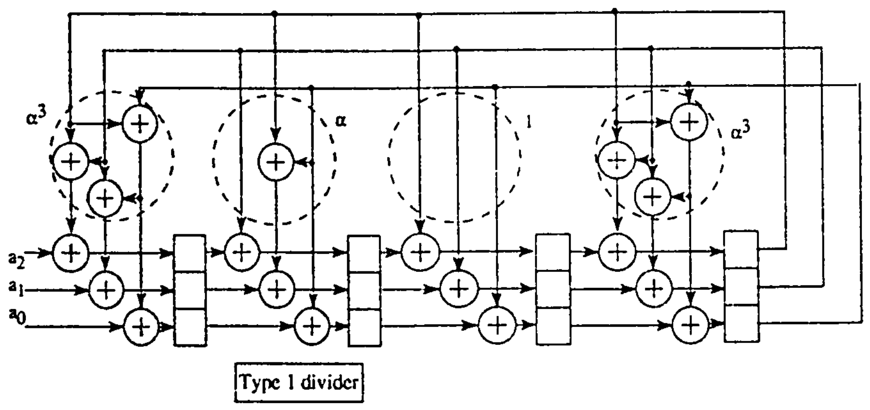


Figure 2.13: A circuit for dividing polynomials.

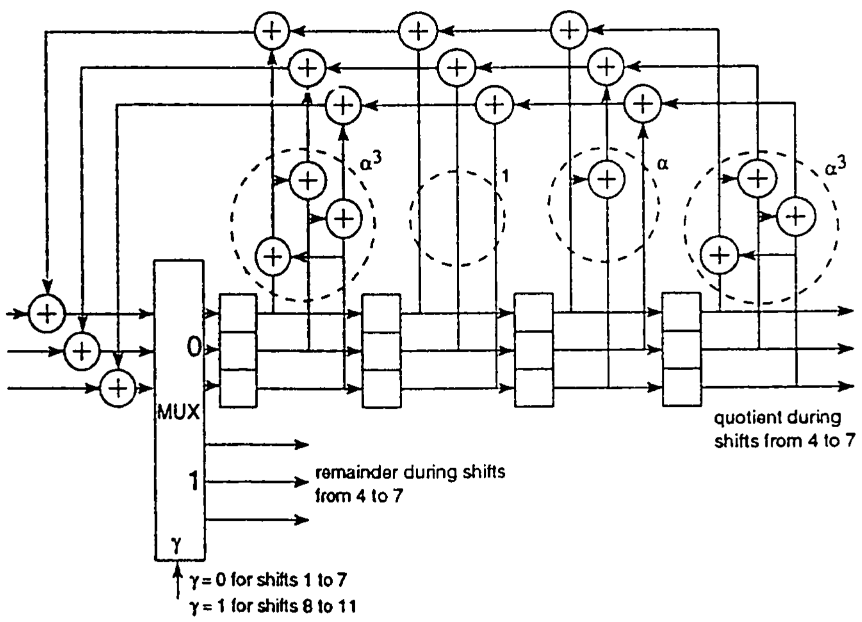


I/P	State	O/P
α^4	0 0 0 0	0
0	α^4 0 0 0	0
0	0 α^4 0 0	0
α^3	0 0 α^4 0	0
0	α^3 0 0 α^4	0
α^6	1 α^2 α^4 1	α^4
α^4	α^4 α^3 α^6 α^6	1
0	α α^5 α^4 1	α^6

Quotient = $\alpha^4x^2 + x + \alpha^6$

Remainder = $x^3 + \alpha^4x^2 + \alpha^5x + \alpha$

Figure 2.14: Type 1 circuit for dividing polynomials.


α^3
1
α
α^3
0
MUX
1
γ
remainder during shifts from 4 to 7
quotient during shifts from 4 to 7
$\gamma = 0$ for shifts 1 to 7
$\gamma = 1$ for shifts 8 to 11

I/P	State	Quotient	Rem.
α^4	0 0 0 0	0	
0	α^4 0 0 0	0	
0	1 α^4 0 0	0	
$\alpha 3$	α^6 1 α^4 0	0	
0	1 α^6 1 α^4	0	
$\alpha 6$	α^6 1 α^6 1	α^4	
$\alpha 4$	α α^5 1 α^6	1	
0	α^3 α α^6 1	α^6	
0	0 α^3 α α^6		α^6
0	0 0 α^3 α		α^3
0	0 0 0 α^3		0
0	0 0 0 0		α^6

Quotient = $\alpha^4 x + x + \alpha^6$
Remainder = $\alpha^6 x^3 + \alpha^3 x^2 + \alpha^6$

Figure 2.15: Type 2 circuit for dividing polynomials.

2.5 Applications: Reed-Solomon (RS) Codes

2.5.1 Preliminaries

Reed-Solomon (RS) codes are cyclic non-binary codes. They are used in error control coding for deep space communication and for data storage devices. The codewords can be represented as polynomials where the coefficients are m-bit symbols taken from a finite field $GF(2^m)$. RS error correction is concerned with correcting symbols and not individual bits. For a *t-error correcting* (TEC) Reed-Solomon code the generator polynomial is

$$g(x) = (x - \alpha)(x - \alpha^2) \cdots (x - \alpha^{2t}). \tag{2.12}$$

This always is a polynomial of degree $2t$. Hence a RS code satisfies $n - k = 2t$. They are Maximum Distance Separable (MDS) codes, and the minimum distance is $n - k + 1$ which is the largest minimum distance a code can have. For a fixed (n, k), no code can have a larger minimum distance than a RS code. The parameters of an RS code defined over $GF(2^m)$ are given below:

Number of bits per symbol	: m
Code length in symbols	: $n = 2^m - 1$
Number of information symbols	: $k = n - d + 1$
Error correcting capability	: t
Number of parity symbols	: $2t$

Minimum distance : $d = n - k + 1$

The cyclic property of the RS codes makes it attractive to use shift-register circuits for constructing encoders and decoders. RS codes can be implemented non-systematically by multiplication of the variable information polynomial $i(x)$ with the fixed polynomial $g(x)$ to get the codeword $c(x)$. This can be implemented with a FIR filter over $GF(2)$.

2.5.2 Systematic Encoding of RS Codes

Given the generator polynomial $g(x)$ of an (n, k) RS code, the code can be put in a *systematic form.* That is, *the first k digits of the each code word are the unaltered information digits: the last $n - k$ digits are parity check digits.* Suppose that the message of k digits to be encoded is

$$m = (m_0, m_1, m_2, \cdots, m_k - 1). \tag{2.13}$$

The corresponding message polynomial is

$$m(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{(k-1)} x^{k-1} \tag{2.14}$$

Multiplying $m(x)$ by x^{n-k} and then dividing by $g(x)$, we obtain

$$x^{n-k} m(x) = q(x) g(x) + r(x) \tag{2.15}$$

where $q(x)$ and $r(x)$ are the quotient and the remainder respectively.

Since the degree of the generator polynomial $g(x)$ is $n - k$, the degree of $r(x)$ must be $n-k-1$ or less,

$$r(x) = r_0 + r_1 x + r_2 x^2 + \cdots + r_{(n-k-1)} x^{n-k-1}. \tag{2.16}$$

Rearranging eqn 2.15, we obtain

$$r(x) - x^{n-k} m(x) = q(x) g(x) \tag{2.17}$$

This indicates that $r(x) + x^{n-k}m(x)$ is a multiple of $g(x)$ and has degree $n-1$ or less. Therefore, $r(x) + x^{n-k}m(x)$ is a code polynomial of the RS code c generated by $g(x)$.

$$\begin{aligned} c(x) &= r(x) + x^{n-k} m(x) \\ &= r_0 + r_1 x + r_2 x^2 + \cdots + r_{n-k-1} x^{n-k-1} + \\ & m_0 x^{n-k} + m_1 x^{n-k+1} + \cdots + m_{(k-1)} x^{n-1} \end{aligned} \tag{2.18}$$

From eqn 2.18, it is clear that the encoding of a message block $m(x)$ of k digits is equivalent to calculating the parity check polynomial $r(x)$ which is the remainder of dividing $x^{n-k}m(x)$ by the generator polynomial $g(x)$. This is accomplished by a dividing circuit which is a shift register with feedback connections according to the generator polynomial, with $g_0 = 1$,

$$g(x) = g_0 + g_1 x + g_2 x^2 + \cdots + g_{n-k-1} x^{n-k-1} + x^{n-k}. \tag{2.19}$$

An encoding circuit with an $(n-k)$-stage shift register is shown in Fig. 2.16. The quantities $g_0, g_1, \cdots, g_r$ are the coefficients of the generator polynomial of the code.

Consider the example of a double error correcting (DEC) RS (7,3) code with symbols from $GF(2^3)$. The primitive polynomial is

$$p(z) = z^3 + z + 1$$

and the generator polynomial is given by

$$\begin{aligned} g(x) &= LCM\left[m_1(x), m_2(x), m_3(x), m_4(x)\right] \\ &= (x+\alpha)(x+\alpha^2)(x+\alpha^3)(x+\alpha^4) \\ &= \left[x^2 + (\alpha+\alpha^2)x + \alpha^3\right]\left[x^2 + (\alpha^3+\alpha^4)x + \alpha^7\right] \\ &= (x^2 + \alpha^4 + \alpha^3)((x^2 + \alpha^6 x + \alpha^0) \\ &= x^4 + (\alpha^6+\alpha^4)x^3 + (1+\alpha^3+\alpha^3)x^2 + (\alpha^4+\alpha^2)x + \alpha^3 \\ &= x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3 \end{aligned}$$

Therefore, $g(x) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3$. When implemented with binary circuits, the octal shift registers stages are three binary shift register stages in parallel. All data are 3-bit wide. The binary equivalent of the coefficients of the generator polynomial are calculated by multiplying the coefficients with a polynomial $a_2 z^2 + a_1 z + a_0$, $a_i \in GF(2)$ and taking the remainder on division by the irreducible polynomial $p(z)$. The computation is described below:

$$R_{p(z)}\left[z^3(a_2 z^2 + a_1 z + a_0)\right] = (a_1 + a_2)z^2 + (a_0 + a_1 + a_2)z + (a_0 + a_2)$$

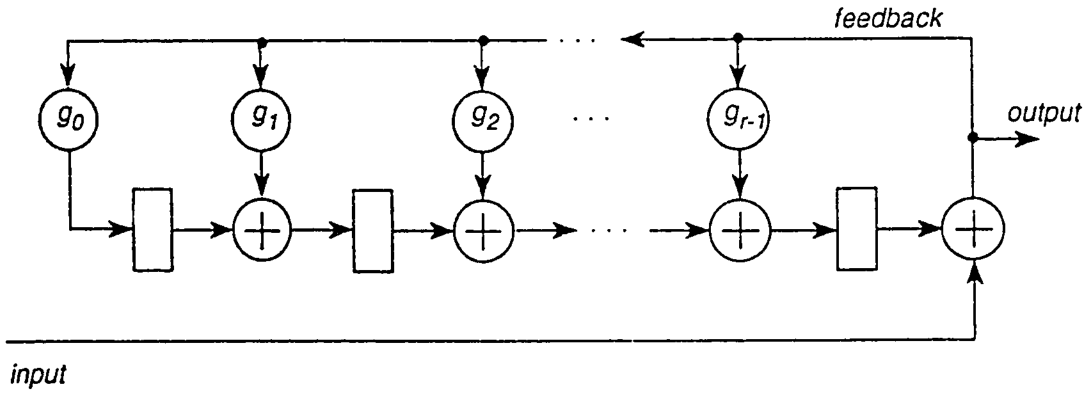


Figure 2.16: An (n-k)-stage feedback shift register encoding circuit

$$R_{p(z)}\left[z(a_2 z^2 + a_1 z + a_0)\right] = a_1 z^2 + (a_2 + a_0)z + a_0$$

The binary implementation of the (7,3) RS code is shown in Fig. 2.17.

The implementation shown above is in wide use. It is simple, and it meets the requirements for most applications. However, in some very high speed applications, the presence of a global feedback signal imposes constraints on the switching speed of the encoder. The fact that the input to all $(n-k)$ stages depend on the feedback signal, forces the $(n-k)$ stages to be synchronous, hence imposing the use of a global clock. The need to distribute the global clock and the feedback signal to all stages of the encoder might restrict the maximum switching speed achievable in a practical implementation. An alternative approach for the elimination of the feedback is to modify the FSR encoder of Fig. 2.16, by "pipelining" the feedback path so that the feedback signal goes through one delay unit before it is fed back to each shift register stage. This architecture is shown in Fig. 2.18. The resulting encoding stream is *interleaved* 2:1.

2.5.3 Decoding of RS Codes

When a codeword is transmitted over a noisy channel, it may be corrupted by noise. At the output of the channel, the received word may or may not be the same as the transmitted code word. The function of the decoder is to recover the transmitted code word from the knowledge of the received word. Let the received word be

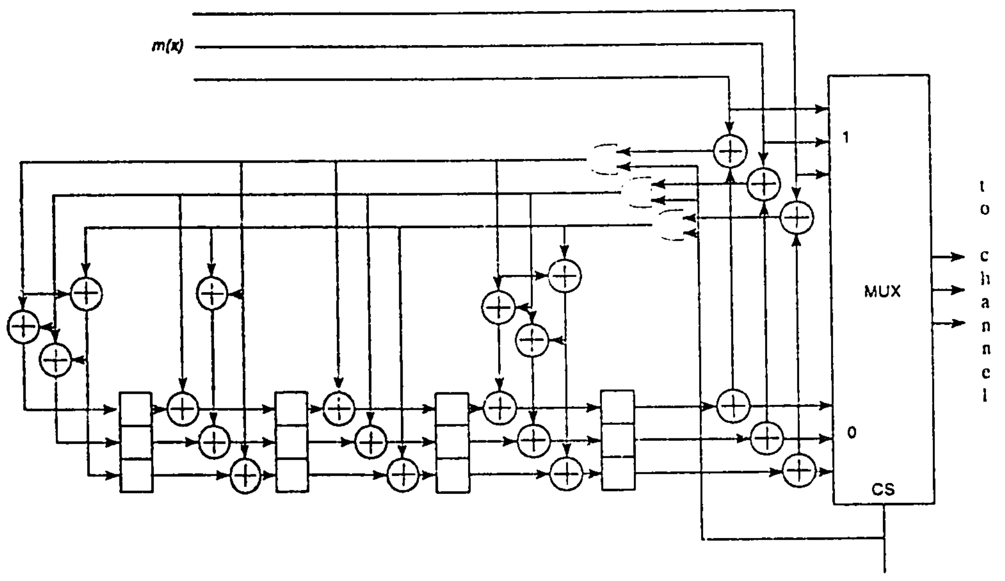


Figure 2.17: Encoder for the RS (7,3) code using binary logic elements.

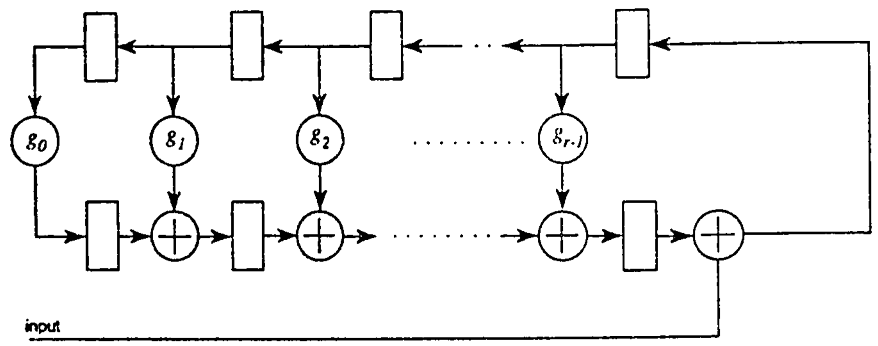


Figure 2.18: Pipelined LFSR encoder

represented by the binary polynomial

$$v(x) = v_0 + v_1 x + v_2 x^2 + \cdots + v_{n-1} x^{n-1}.$$

The decoder first checks whether or not the received polynomial corresponds to a codeword by dividing the polynomial by the generator polynomial $g(x)$ of the code. The remainder after the division is the syndrome of the received word.

$$v(x) = p(x)g(x) + s(x), \tag{2.20}$$

where $s(x)$ is a polynomial of degree $(n - k - 1)$ or less. Thus, the syndrome is a $(n - k)$–tuple. If the syndrome is zero, the received polynomial is divisible by the generator polynomial and corresponds to a codeword; the decoder will accept the received word as the transmitted code vector. If the syndrome is a non-zero vector, the received vector is not a codeword and errors have been detected. Supposing that $c(x)$ be the transmitted polynomial, then in the case of an error

$$v(x) = c(x) + e(x) \tag{2.21}$$

where $e(x)$ is the error pattern caused by the channel disturbance. Since $c(x)$ is a code polynomial, it must be a multiple of the generator polynomial $g(x)$, say

$$c(x) = m(x)g(x). \tag{2.22}$$

Combining Eqs. 2.20, 2.21 and 2.22, we obtain

$$e(x) = [p(x) - m(x)]\, g(x) + s(x) \tag{2.23}$$

That is, the syndrome of $v(x)$ is equal to the remainder resulting from dividing the error pattern by the generator polynomial $g(x)$ of the code. Thus, *the syndrome of the received word contains the information about the error pattern in the received word, which will be used for error correction.*

The syndrome calculation is accomplished by a division circuit which is identical to the encoding circuit at the transmitter. The syndrome calculator for the RS (7,3) code is shown in Fig. 2.19.

The received word is shifted into the register with all stages initially set to "0". After the entire received word has been entered into the shift register, i.e., after n shift pulses, the contents will be the syndrome.

Error Trapping

An error trapping decoder is a modification of a Meggitt decoder that can be used for certain cyclic codes including BCH codes and RS codes. Suppose that all errors in a received word occur close together. Then the syndrome, properly shifted, exhibits an exact copy of the error pattern. Defining the length of an error pattern as the smallest number of sequential stages of a shift register that must be observed so that for some cyclic shift of the error pattern, all errors are within this segment of the shift register. Supposing that the longest-length error pattern that must be corrected is no longer than the syndrome. Then for some cyclic shift of the error pattern, the syndrome is equal to the error pattern. For such codes, the syndrome

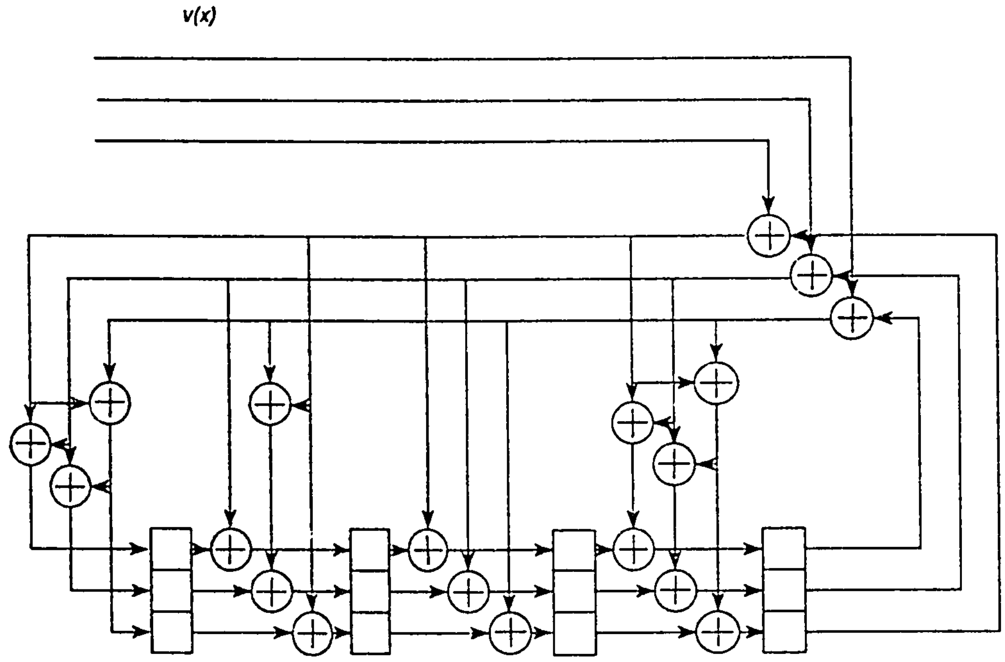


Figure 2.19: Syndrome Calculator for the RS (7,3) code.

generator is shifted until a correctable pattern is observed. The content of the syndrome generator is subtracted from the cyclically shifted received polynomial and correction is complete. Referring to the example of the Reed-Solomon (7,3) code over $GF(8)$ with generator polynomial

$$g(x) = x^4 + (z+1)x^3 + x^2 + zx + (z+1),$$

where the field elements are expressed as polynomials in z. Alternatively,

$$g(x) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3,$$

where the field elements are expressed in terms of the primitive element $\alpha = z$. This is a double error correcting (DEC) code. The error pattern $e(x)$ has at most two

$\alpha^0 = 1$	$g(x) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3$
$\alpha^1 = z$	$c(x) = \alpha^4 x^6 + \alpha^4 x^4 + \alpha^3 x^3 + \alpha^6 x + \alpha^6$
$\alpha^2 = z^2$	$e(x) = \alpha^4 x^4 + \alpha^3$
$\alpha^3 = z+1$	$v(x) = \alpha^4 x^6 + \alpha^3 x^3 + \alpha^6 x + \alpha^4$
$\alpha^4 = z^2 + z$	
$\alpha^5 = z^2 + z + 1$	
$\alpha^6 = z^2 + 1$	

nonzero terms and is a polynomial of degree 6 or less. It can always be cyclically shifted into a polynomial of degree 3 or less. Because the syndromes are of degree 3 or less, error trapping can be applied. The error trapping decoder is shown in Fig. 2.20. The binary circuit implementation uses octal shift register stages with three binary shift register stages in parallel. The multiply-by-z+1 (that is, by α^3) and

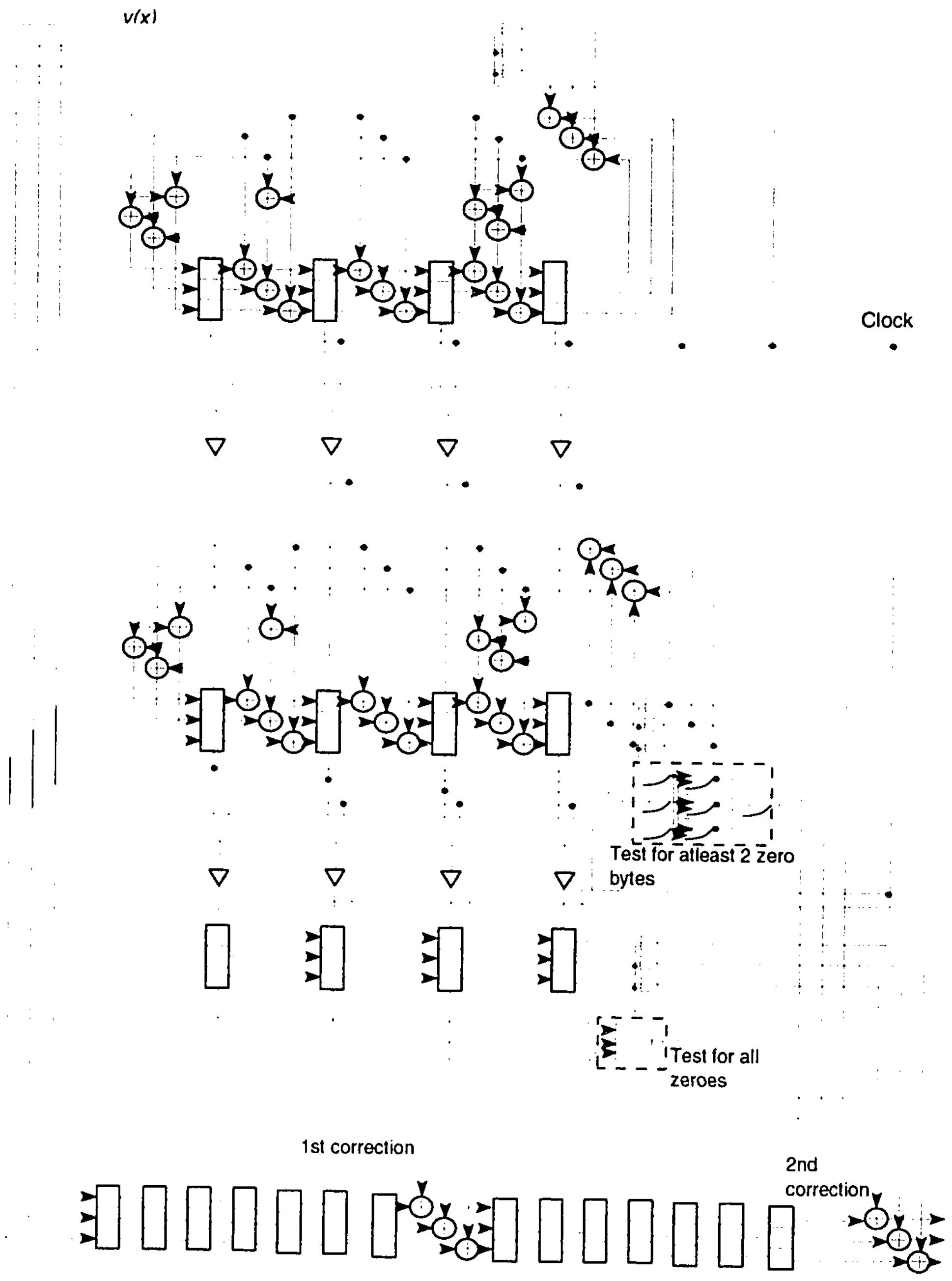


Figure 2.20: Pipelined decoder for the RS (7,3) code

-z (that is, by α) circuits in the feedback path are simple three-input three-output binary-logic circuits. A total of *21* shifts are necessary for this decoder to correct all errors. The first set of *7* shift computes the syndrome. The second set of *7* shifts corrects at least one error and the third set of *7* shifts corrects the second error.

In Table 2.3, for each shift, the content of both the syndrome register and the information buffer are listed. Beginning with the eighth shift, the circuit will correct an error whenever it sees an error-trapped pattern. This is a pattern with at most two nonzero symbols and one with one of the nonzero symbols in the right-most place. Such a pattern must always occur at least once in the second *7* shifts if no more than *2* errors occur. In the example, it occurs at shift *13*. Hence, the error is trapped. It is to be noticed that the high-order syndrome symbols are normally to the right. That the error is trapped can be seen from a *4* symbol segment of the cyclically shifted error pattern (e_4, e_5, e_6, e_0). Because the error is trapped, e_0 is corrected in the information register and set equal to zero (or subtracted from itself) in the syndrome register. After *14* shifts, the syndrome register contains the syndrome of the remaining error pattern. The process repeats through a third set of *7* shifts, after which the error correction is complete.

If the syndrome is initially nonzero but no error is trapped during the second set of *7* shifts, then more than two errors occured (but the converse is not true). Additional logic can easily included to detect an uncorrectable error pattern.

Shift	Syndrome Register		Information Register
0	$0\,0\,0\,0$		$0\,0\,0\,0\,0\,0\,0$
1	$1\,\alpha^5\alpha^4 1$		$\alpha^4 0\,0\,0\,0\,0\,0$
2	$\alpha^3\alpha^3\alpha^4\alpha^6$		$0\,\alpha^4 0\,0\,0\,0\,0$
3	$\alpha^2\alpha\,\alpha^4\alpha$		$0\,0\,\alpha^4 0\,0\,0\,0$
4	$\alpha^3\alpha^4\alpha^3\alpha^6$		$\alpha^3 0\,0\,\alpha^4 0\,0\,0$
5	$\alpha^2\alpha\,\alpha^3\alpha^5$		$0\,\alpha^3 0\,0\,\alpha^4 0\,0$
6	$\alpha^4 0\,0\,\alpha^6$		$\alpha^6 0\,\alpha^3 0\,0\,\alpha^4 0$
7	$\alpha^6 0\,\alpha^3\alpha^6$	Syndrome	$\alpha^4\alpha^6 0\,\alpha^3 0\,0\,\alpha^4$
8	$\alpha^2\alpha^2\alpha^6\alpha^5$		$\alpha^4\alpha^4\alpha^6 0\,\alpha^3 0\,0$
9	$\alpha\,1\,\alpha^3\alpha^5$		$0\,\alpha^4\alpha^4\alpha^6 0\,\alpha^3 0$
10	$\alpha\,\alpha^5\alpha^4 1$		$0\,0\,\alpha^4\alpha^4\alpha^6 0\,\alpha^3$
11	$\alpha^3 0\,\alpha^4\alpha^6$		$\alpha^3 0\,0\,\alpha^4\alpha^4\alpha^6 0$
12	$\alpha^2\alpha\,\alpha^6\alpha$		$0\,\alpha^3 0\,0\,\alpha^4\alpha^4\alpha^6$
13	$\alpha^4 0\,0\,\alpha^3$	← Trapped error	$\alpha^6 0\,\alpha^3 0\,0\,\alpha^4\alpha^4$
14	$0\,\alpha^4 0\,0$		$\alpha^6\alpha^6 0\,\alpha^3 0\,0\,\alpha^4$
15	$0\,0\,\alpha^4 0$		$\alpha^6\alpha^6 0\,\alpha^3 0\,0$
16	$0\,0\,0\,\alpha^4$	← Trapped error	$\alpha^6\alpha^6 0\,\alpha^3 0$
17	$0\,0\,0\,0$		$\alpha^6\alpha^6 0\,\alpha^3$
18	$0\,0\,0\,0$		$\alpha^6\alpha^6 0$
19	$0\,0\,0\,0$		$\alpha^6\alpha^6$
20	$0\,0\,0\,0$		α^6
21	$0\,0\,0\,0$		

Table 2.3: Error Trapping Example

Chapter 3

Architectures Using Systolic Arrays

The advantages of the LSR circuits and the designs based on the normal basis are that they take lesser area. However, they are less expandable because irreducible polynomials for different m are not related, but the design focusses on one particular m each time. Since systolic arrays have the advantages of regularity, modularity and expandability, systolic designs are worthy of consideration.

3.1 Multipliers for $GF(2^m)$

Multipliers based on standard basis or conventional basis do not require any basis conversion. Wang and Lin [12] developed a serial-in-serial-out systolic (SISOS) array multiplier and a parallel-in-parallel-out systolic (PIPOS) array multiplier which obtains the product of two elements in the finite field $GF(2^m)$ in standard basis. The multipliers are based on the algorithm developed in [28], which is presented in the next section.

3.1.1 Parallel-In-Parallel-Out Systolic (PIPOS) Array

Let two elements of $GF(2^m)$ be $A(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0$ and $B(x) = b_{m-1}x^{m-1} + \cdots + b_1x + b_0$, and the irreducible polynomial be $F(x) = x^m + g_{m-1}x^{m-1} + \cdots + g_1x + 1$, where the coefficients a_j, b_j, and $g_j \in GF(2)$. The product $A(x)B(x)\text{mod}G(x)$ can be represented by $P(x) = p_{m-1}x^{m-1} + \cdots + p_1x + p_0$.

The multiplication $A(x)B(x)\text{mod}G(x)$, can be expanded by multiplying each term of $B(x)$ by $A(x)$ as shown below in eqn. 3.1.

$$\begin{aligned} P(x) &= A(x)B(x)\text{mod}G(x) \\ &= A(x)\{b_{m-1}x^{m-1} + \cdots + b_1x + b_0\}\text{mod}G(x) \\ &= \{A(x)b_{m-1}x^{m-1}\text{mod}G(x) \\ &\quad + \cdots + A(x)b_1x\text{mod}G(x) \\ &\quad + A(x)b_0\text{mod}G(x). \end{aligned} \tag{3.1}$$

Beginning with the first term, $A(x)b_{m-1}x^{m-1}\text{mod}G(x)$, each successive term in eqn. 3.1 is added to it and the sum reduced mod$G(x)$ until all terms have been used.

Based on the above algorithm, $P(x)$ can be computed recursively as follows:

$$T_0(x) = 0 \tag{3.2}$$

$$\begin{aligned} T_i(x) &= [T_{i-1}(x)x] \bmod G(x) \\ &\quad + A(x)b_{m-i}, \qquad i = 1, 2, \cdots, m \end{aligned} \tag{3.3}$$

$$P(x) = T_m(x), \tag{3.4}$$

where

$$T_i(x) = t_{i,m-1}x^{m-1} + t_{i,m-2}x^{m-2} + \cdots + t_{i,1}x + t_{i,0} \tag{3.5}$$

$$P(x) = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \cdots + p_1x + p_0. \tag{3.6}$$

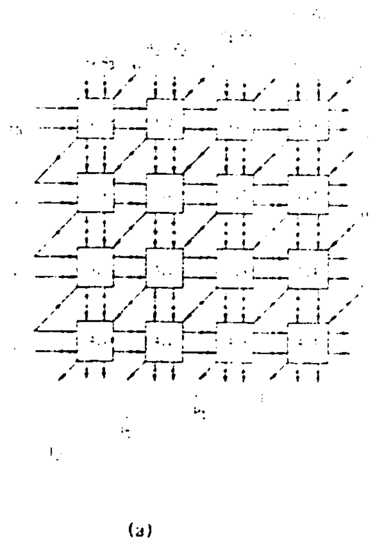
Denoting the coefficient of the highest order term of $T_i(x)$ by M_i, we can write the recurrence relation as

$$\begin{aligned} T_i(x) &= T_{i-1}(x)x + G(x)M_{i-1} \\ &\quad + A(x)b_{m-i}, \qquad i = 1, 2, 3, \cdots, m \end{aligned} \tag{3.7}$$

The above computing procedure is realised by a PIPOS array as shown in Fig. 3.1 [12], where $m \times m$ basic cells are used. The basic cell at position (i, k) performs the following logic operation:

$$\begin{aligned} t_{i,k} &= t_{i-1,k+1} \oplus (g_{m-k} \cdot M_{i-1}) \\ &\quad \oplus (a_{m-k} \cdot b_{m-i}). \end{aligned} \tag{3.8}$$

In this array, one 1-bit delay (denoted by "●") has been placed at each horizontal path and each slant path between cells, and two 1-bit latches are placed at each


(a)

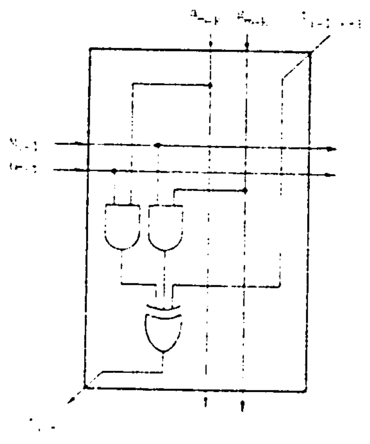

(b)

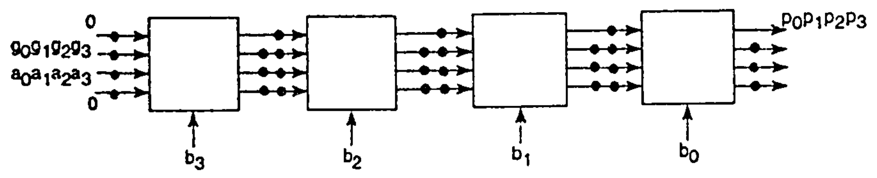
Figure 3.1: (a) Parallel-in-parallel-out bit-level systolic array for multiplication in $GF(2^4)$. (b) Circuit of the (i,k) cell in part (a).

vertical path between cells. The coefficients a_j and g_j enter the $(m-j)$th column from the top, and b_j enters the $(m-j)$th row from the left-hand side, where $j = 0, 1, 2, \cdots, m-1$. In addition, $a_j(g_j)$ is staggered by one clock cycle relative to $a_{j+1}(g_{j+1})$, and b_j is staggered by two clock cycles relative to b_{j+1}.

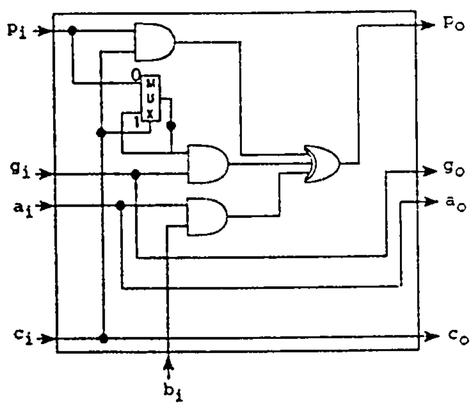
The operation of $T_i(x) = T_{i-1}(x)x + G(x)M_{i-1} + A(x)b_{m-1}$ in eqn. 3.7 is performed at the ith row of the array. All logic operations are pipelined such that each cell is doing the logic operation of eqn.3.8 for an (i, k) pair and passes data and the result to the neighbouring cells. It can be seen that the output bit p_{j+1} will emerge from the bottom of the $(m-j-1)$th column one cycle ahead of p_j which will emerge from the bottom of the $(m-j)$th column. If the input data come in continuously, the array will yield output results at a rate of one per clock cycle after an initial delay of $3m$ clock cycles (including the input/output delay).

3.1.2 Serial-In Serial-Out Systolic (SISOS) Array

The serial-in-serial-out systolic array is shown in Fig. 3.2. The coefficients of $A(x)(G(x)$ enter the array in a serial format with the MSB first. Since the MSB of each partial result $T_i(x)$ is required to control the mod$G(x)$ operation, as given in eqn. 3.7, we need some extra circuitry to latch such bits. In the basic cell shown in Fig. 3.2, an AND gate and a switch or a multiplexer (MUX) are added for this purpose. The operation of the extra circuitry are controlled by a sequence of $011 \cdots 1$ with length m. The zero bit of the control sequence enters the array one cycle ahead



(a)



(b)

Figure 3.2: (a) A serial-in-serial-out systolic array for multiplication in $GF(2^4)$. The result p_j will appear after a_j passes through the array. (b) The circuit of each basic cell in (a).

of the MSB of $A(x)$, and it is synchronised with the MSB of $T_i(x)$. If the data comes in continuously, it will yield output results at a rate of one per m cycles after an initial delay of $3m$ cycles. The output result emerges from the right-hand side of the array in serial form with the MSB first.

3.1.3 Conclusions

Both the arrays have unidirectional data flow. The parallel form incorporates fault tolerant design. and the serial form requires only one control signal. All the operations are pipelined in such a way that each block performs the same logic operation and passes the data and the result to the neighbouring cells.

3.2 Standard Basis Inversion and Division

In this section an algorithm for computing A^{-1} and $B \cdot A^{-1}$ in $GF(2^m)$ is developed based on the Gauss-Jordon elimination method [29] for a system of linear equations over $GF(2)$. Finally, a systolic array implementation of the Gauss-Jordon algorithm based on the design given by Wang and Lin [15] is presented.

3.2.1 An Algorithm for Computing A^{-1} and $B \cdot A^{-1}$ in $GF(2^m)$

Let $A(x)$ be the polynomial representing a nonzero element of the finite field $GF(2^m)$ with a primitive polynomial $G(x)$ of degree m. Then the inverse element of $A(x)$ is a polynomial $C(x)$ such that

$$A(x) \cdot C(x) \equiv 1\{\text{mod } G(x)\} \tag{3.9}$$

or

$$A(x) \cdot C(x) + H(x) \cdot G(x) = 1 \tag{3.10}$$

where $H(x)$ is also a polynomial. Since any element in $GF(2^m)$ can be represented by a polynomial of degree $m-1$ or less, $H(x)$ must have a degree satisfying

$$\deg H(x) \leq m-2. \tag{3.11}$$

Thus we can have

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0 \tag{3.12}$$

$$C(x) = c_{m-1}x^{m-1} + \cdots + c_1x + c_0 \tag{3.13}$$

$$H(x) = h_{m-2}x^{m-2} + \cdots + h_1x + h_0 \tag{3.14}$$

$$G(x) = g_mx^m + g_{m-1}x^{m-1} + \cdots + g_1x + g_0 \tag{3.15}$$

where coefficients a_j, b_j, and g_j are from $GF(2) = \{0, 1\}$ and $g_m = g_0 = 1$. On multiplying out eqn. 3.10, we obtain

$$g_0h_0 + a_0c_0 \tag{3.16}$$

$$+(g_1h_0 + a_1c_0 + g_0h_1 + a_0c_1)x \tag{3.17}$$

$$+(g_2h_0 + a_2c_0 + g_1h_1 + a_1c_1 + g_0h_2 + a_0c_2)x^2 \tag{3.18}$$

$$\vdots \tag{3.19}$$

$$+(g_mh_{m-2} + a_{m-1}c_{m-1})x^{2m-2} = 1. \tag{3.20}$$

In matrix form, this becomes eqn. 3.21. Clearly, finding an inverse element in $GF(2^m)$ can be achieved by solving a system of $2m-1$ linear equations with $2m-1$ unknowns over $GF(2)$. Since every nonzero element has an inverse, there always exists a solution for eqn. 3.21. This means that the determinant of the square matrix is nonzero if $A(x) \neq 0$.

$$\begin{bmatrix} g_0 & 0 & \cdot & 0 & a_0 & 0 & \cdot & 0 \\ g_1 & g_0 & \cdot & \cdot & a_1 & a_0 & \cdot & \cdot \\ \vdots & g_1 & \ddots & 0 & \cdot & a_1 & \ddots & 0 \\ & & & g_0 & \cdot & \cdot & & a_0 \\ g_m & \vdots & \ddots & g_1 & a_{m-1} & \cdot & & a_1 \\ 0 & g_m & & & 0 & a_{m-1} & & \cdot \\ \cdot & 0 & \ddots & \vdots & \cdot & 0 & \ddots & \cdot \\ 0 & \cdot & \cdot & g_m & 0 & \cdot & & a_{m-1} \end{bmatrix} \cdot \begin{bmatrix} h_0 \\ \vdots \\ h_{m-2} \\ c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} \qquad (3.21)$$

Let

$$B(x) = b_{m-1}x^{m-1} + \cdots + b_1x + b_0 \qquad (3.22)$$

be an element in $GF(2^m)$ such that

$$A(x) \cdot C(x) \equiv B(x)\{\bmod G(x)\} \qquad (3.23)$$

or

$$A(x) \cdot C(x) + H(x) \cdot G(x) = B(x). \qquad (3.24)$$

Then, $C(x) = B(x) \cdot A^{-1}(x) \bmod G(x)$. Analogous to eqn. 3.10, 3.24 can be expressed in matrix form as eqn. 3.25. By the same reason, the result of $B(x)A^{-1}(x)$ in $GF(2^m)$ can be obtained by solving the system of linear equations given by eqn. 3.25.

$$\begin{bmatrix} g_0 & 0 & \cdot & 0 & a_0 & 0 & \cdot & 0 \\ g_1 & g_0 & \cdot & \cdot & a_1 & a_0 & \cdot & \cdot \\ \vdots & g_1 & \ddots & 0 & \cdot & a_1 & \ddots & 0 \\ & \cdot & \cdot & g_0 & \cdot & \cdot & \vdots & a_0 \\ g_m & \cdot & \cdot & g_1 & a_{m-1} & \cdot & & a_1 \\ 0 & g_m & & & 0 & a_{m-1} & & \cdot \\ \cdot & 0 & \ddots & \vdots & \cdot & 0 & \ddots & \cdot \\ 0 & \cdot & \cdot & g_m & 0 & \cdot & & a_{m-1} \end{bmatrix} \cdot \begin{bmatrix} h_0 \\ \vdots \\ h_{m-2} \\ c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{3.25}$$

3.2.2 The Gauss-Jordan Algorithm

Given an $n \times n$ nonsingular matrix $A = [a_{ij}]$ and an n-dimensional column vector $b = [b_{ij}]$, where a_{ij} and b_j are in $GF(2)$, the solution of $Ax = b$ can be produced by using the Gauss-Jordan algorithm given as follows:

Algorithm

Step 1 For $k = 1$ to n do Steps 2 to 6.

Step 2 Search the first nonzero element of the Kth column from the first element $i = 1$.

Step 3 If $a_{ik} \neq 0$ Then go to **Step 5**.

Step 4 Row Passing: (a_{ik} cannot be used as a pivot element and the ith-row elements are not processed during the kth iteration) $i = i + 1$; go to **Step 3**

Step 5 Row Loading: (a_{ik} is chosen as the pivot element for the kth iteration and the ith- row elements are stored for use in **Step 6**)

$$r_j = a_{ij}, \quad j = k+1, k+2, \cdots, n$$

$$r_{n+1} = b_i$$

Step 6 Column Elimination and Row Rotation: (Subtract row i times $am + 1, k$ from row $m + 1$ of $[A|b]$ and store the result on row m, where $m = i, i + 1, \cdots, n - 1$, and then store the old row i on row n. all arithmetic operations are performed by taking the results modulo 2)

$$a_{mj} = a_{m+1,j} \oplus (a_{m+1,k} \cdot r_j)$$

$$m = i, i+1, \cdots, n-1 \text{ and } j = k+1, k+2, \cdots, n-1$$

$$b_m = b_{m+1} \oplus (a_{m+1,k} \cdot r_{n+1}), m = i, i+1, \cdots, n-1$$

$$a_{nj} = r_j, j = k+1, k+2, \cdots, n$$

$$b_n = r_{n+1}$$

We can see form the above that the kth iteration of the Gauss-Jordan algorithm involves c $(0 \leq c \leq n-1)$ row passing operations, one row loading operation, and $n - c - 1$ column elimination and row rotation operations. Note that the resulting $n-$dimensional vector $b = b_i$ for $i = 1, 2, \cdots, n$. To further illustrate the above algorithms, we present the following example to show the operation of inversion.

Example 3.1 Consider $G(x) = x^3 + x + 1$ and let $A(x) = x^2 + x$ over $GF(2^3)$

The polynomials $A(x)$, $G(x)$, $H(x)$ and $C(x)$, can be written as

$$A(x) = a_2x^2 + a_1x + a_0$$

$$C(x) = c_2x^2 + c_1x + c_0$$

$$H(x) = h_1x + h_0$$

$$G(x) = g_3x^3 + g_2x^2 + g_1x + g_0$$

Equation 3.21 in this case becomes

$$\begin{bmatrix} g_0 & 0 & a_0 & 0 & 0 \\ g_1 & g_0 & a_1 & a_0 & 0 \\ g_2 & g_1 & a_2 & a_1 & a_0 \\ g_3 & g_3 & 0 & a_2 & a_1 \\ 0 & g_3 & 0 & 0 & a_2 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

which represents a system of $2m - 1$ $(= 5)$ linear equations over $GF(2)$. They can be solved using the Gauss-Jordon reduction method, which results in the following

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Therefore, the inverse in this case is $C(x) = x + 1$

3.2.3 Systolic Array Implementation of the Gauss-Jordon Algorithm over $GF(2^m)$

Based on the Gauss-Jordon algorithm, a triangular systolic array is constructed for $GF(2^3)$ as shown in Fig. 3.3 for the solution of a system of n linear equations $Ax = b$ over $GF(2)$. This array consists of $(n^2 + n)/2$ main array cells and n boundary cells. The kth row from the bottom of the array is used for the kth iteration of the Gauss-Jordan algorithm. Each main array cell operates in one of three modes depending on the westbound tag bit T and the data bit E. When $T = 0$ and $E = 0$, the pivot element has not been found and the main array cell operates in the row passing mode, in which it exchanges the data stored in the R register with th. data arriving at the northbound input (D) and sends the old value to its northbound output (D'). If $T = 0$ and $E = 1$, meaning E is a pivot element, the operations of the main array cell are the same as those of the row passing mode. For each of presentation, this case will be referred to as "row loading mode" rather "row passing mode". When $T = 1$, the data E is an element to be eliminated; in this case, the main array cell operates in the column elimination and row rotation mode, where its northbound output D' is updated according to

$$D' = D \oplus (E \cdot R).$$

The three modes of operation for the main array cell are summarised in Table 3.1. The circuit for the realisation is shown in Fig. 3.3.

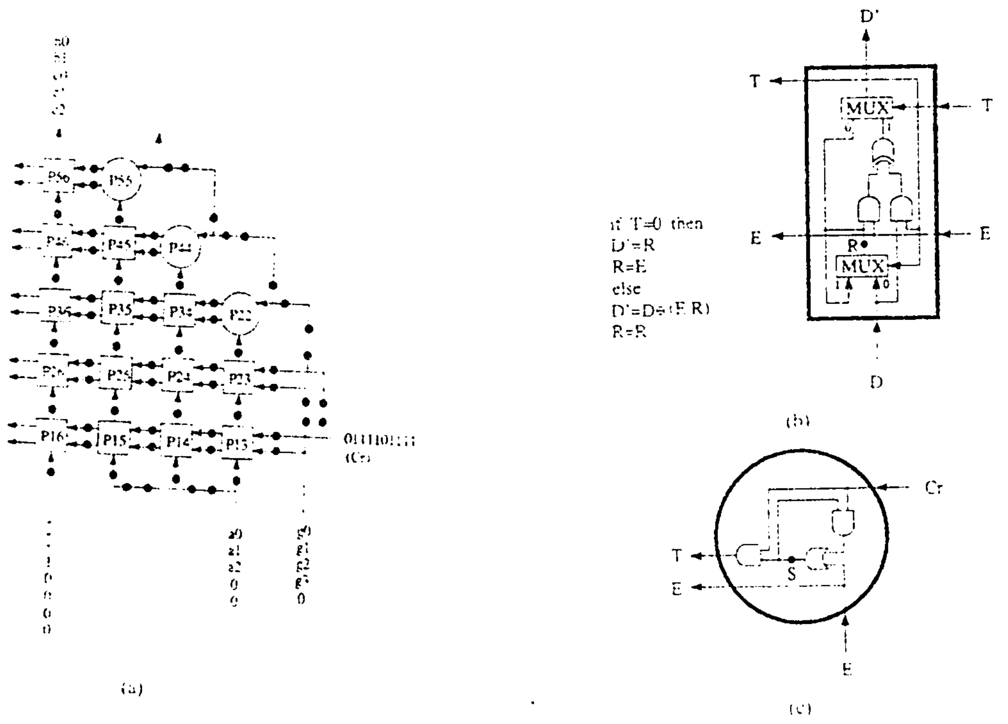


Figure 3.3: (a) Systolic array for inversion in $GF(2^3)$ (b) main array cell (c) boundary cell.

T	E	Operating Mode
0	0	Row passing
0	1	Row loading
1	0	Column elimination and Row rotation
1	1	Column Elimination and Row rotation

Table 3.1: Operating modes of the main array cell

Although the row passing mode and the row loading mode of the main array cell have the same operations, they should be followed by different operating modes. We can see from the Gauss-Jordan algorithm that a row passing is followed by either a row loading operation or a row passing operation for each iteration, while a row loading operation is always followed by a column elimination and row rotation operation except at the end of each iteration. This means that the tag bit T for the kth iteration must be initialised by 0 and then be changed based on the following two rules:

Rule 1 If $T = 0$ and $E = 0$ (row passing mode) occurs at the first clock cycle, then $T = 0$ must hold until $E = 1$ (row loading mode).

Rule 2 If $T = 0$ and $E = 0$ (row loading mode) occurs at the ith clock cycle, then $T = 1$ (column elimination and row rotation mode) must occur from the $(i + 1)$th to nth clock cycles, where we assume that it takes n clock cycles for the kth-row main array cells to complete the kth iteration.

With these two rules, it can be seen that the tag-bit pattern for the kth-row main array cell is a sequence of $c + 1$ zeroes followed by $n - c - 1$ ones as the triggering signal.

It can be seen that the latency of the of the array is $8m - 4$ clock cycles, and the throughput rate is one inverse element per $2m - 1$ clock cycles.

3.3 Application: Encoding of Reed-Solomon Codes

A systolic Reed-Solomon encoder architecture presented in [16] does not use any feedback. In this architecture, a given cell needs only be synchronous with those cells it communicates with. Hence the clock that governs the systolic computation can be propagated from cell to cell along the data, rather than being globally distributed. This clocking scheme (which has also been termed as *hypersystolic*) allows for higher switching speeds than those achievable in architectures with global clocking. This architecture is suitable for very high speed speed applications, in the Gigabit/second-order of magnitude. This performance is this encoder is enhanced by introducing several modifications.

3.3.1 Nonsystematic Encoder

Consider an RS code C of length $n = q - 1$ and redundancy $r < n$ over the finite field $F = GF(q)$. Let α be a primitive element of F, and let $\alpha^L, \alpha^{L+1}, \ldots, \alpha^{L+r-1}$ be the roots of the code for some integer L. Let $a \equiv 1 - L \; mod \; n$. The following is a nonsystematic generator matrix of C:

$$G = \begin{bmatrix} \alpha^{(n-1)a} & \alpha^{(n-2)a} & \cdots & \alpha^{a} & 1 \\ \alpha^{(n-1)(a+1)} & \alpha^{(n-2)(a+1)} & \cdots & \alpha^{(a+1)} & 1 \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdot & \ddots & \cdot & \cdot \\ \cdot & \cdot & \cdots & \cdot & \cdot \\ \alpha^{(n-1)(a+k-1)} & \alpha^{(n-2)(a+k-1)} & \cdots & \alpha^{(a+k-1)} & 1 \end{bmatrix} . \tag{3.26}$$

This generator matrix leads in a natural way to a nonsystematic encoder in which a message $m = (m_0, m_1, \cdots, m_{k-1})$ is encoded into a codeword $v = (v_0, v_1, \cdots, v_{n-1})$, with

$$v_j = \sum_{i=0}^{k-1} m_i \alpha^{(n-1-j)(a+i))}.$$

This encoder is readily implementable in a systolic array without global feedback signals. However, it has the disadvantage of being nonsystematic, and of requiring k (usually $\gg r$) systolic cells.

Example 3.2 For the case of the RS $(7,3)$ code over $GF(2^3)$, the nonsystematic generator matrix turns out to be

$$G = \begin{bmatrix} \alpha^{6} & \alpha^{5} & \alpha^{4} & \alpha^{3} & \alpha^{2} & \alpha & 1 \\ \alpha^{12} & \alpha^{10} & \alpha^{8} & \alpha^{6} & \alpha^{4} & \alpha^{2} & 1 \\ \alpha^{18} & \alpha^{15} & \alpha^{12} & \alpha^{9} & \alpha^{6} & \alpha^{3} & 1 \end{bmatrix} . \tag{3.27}$$

where $L = 0$, therefore $a = 1$. The above G matrix reduces to

$$G = \begin{bmatrix} \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha & 1 \\ \alpha^5 & \alpha^3 & \alpha & \alpha^6 & \alpha^4 & \alpha^2 & 1 \\ \alpha^4 & \alpha & \alpha^5 & \alpha^2 & \alpha^6 & \alpha^3 & 1 \end{bmatrix} . \qquad (3.28)$$

3.3.2 Systematic Encoder

Based on the design proposed in [16], a systolic RS encoder is developed in this section. Several modifications have been introduced for the better performance and to reduce area complexities.

Theory

Regular RS codes are a subset of a more general family known as *generalised Reed-Solomon* (GRS) codes. These have a generator matrices of the form

$$G = [G_0 G_1 \cdots G_{n-1}].$$

where

$$G_i = \begin{bmatrix} 1 \\ \alpha_i \\ \alpha_i^2 \\ \cdot \\ \cdot \\ \cdot \\ \alpha_i^{k-1} \end{bmatrix} u_i, \quad 0 \le i \le n-1$$

for some elements $\alpha_0, \alpha_1, \cdots, \alpha_{n-1}, u_0, u_1, \cdots, u_{n-1}$ of F (the α_i must be distinct, and the u_i nonzero [3]). These elements satisfy the conditions:

$$\alpha_i = \alpha^{n-1-i}, \qquad 0 \le i \le n-1 \tag{3.29}$$

$$u_i = \alpha^{(n-1-i)a}, \qquad 0 \le i \le n-1 \tag{3.30}$$

A systematic generator matrix has the form

$$G_{sys} = [I \mid A],$$

where I is the identity matrix of order k and A is a $k \times r$ matrix. The entries A_{ij} of A can be expressed in the form [30]

$$A_{ij} = \frac{c_i d_j}{x_i + y_j}, \qquad 0 \le i \le k-1, \qquad 0 \le j \le r-1, \tag{3.31}$$

for some elements c_i, d_j, x_i, y_j of F. A is called a *generalised Cauchy matrix*. For arbitrary GRS codes, the parameters c_i, d_j, x_i, y_j are related to the defining parameters α_i, u_i of the code by the formulae:

$$x_i = -\alpha_i. \qquad 0 \leq i \leq k-1.$$

$$y_j = \alpha_{k+j}. \qquad 0 \leq j \leq r-1.$$

$$c_i = \frac{u_i^{-1}}{\prod\limits_{\substack{0 \leq l \leq k-1 \\ l \neq i}} (\alpha_i - \alpha_l)}. \quad 0 \leq \mathrm{i} \leq \mathrm{k}-1.$$

$$d_j = u_{k+j} \prod_{0 \leq l \leq k-1} (\alpha_{k+j} - \alpha_l) \qquad 0 \leq j \leq r-1.$$

Combining the above equations with eqn 3.29 and eqn 3.30 for C.

$$x_i = -\alpha^{n-1-i}. \quad 0 \leq i \leq k-1. \tag{3.32}$$

$$y_j = \alpha^{n-1-(k+j)}. \quad 0 \leq j \leq r-1. \tag{3.33}$$

$$c_i = \frac{\alpha^{-(n-1-i)a}}{\prod\limits_{\substack{0 \leq l \leq k-1 \\ l \neq i}} (\alpha^{n-1-i} - \alpha^{n-1-l})}. \quad 0 \leq \mathrm{i} \leq \mathrm{k}-1. \tag{3.34}$$

$$d_j = \alpha^{(n-1-k-j)a} \prod_{0 \leq l \leq k-1} (\alpha^{n-1-k-j} - \alpha^{n-1-l}). \quad 0 \leq j \leq r-1. \tag{3.35}$$

Let $\mathbf{w} = (w_0 w_1 \cdots w_{r-1})$ denote the vector of check digits produced by a systematic encoder for a message vector $\mathbf{m} = (m_0 m_1 \cdots m_{k-1})$, i.e., $\mathbf{w} = \mathbf{mA}$, where A is the matrix defined in eqn 3.31 - eqn 3.35. Then

$$w_j = \textstyle\sum_{i=0}^{k-1} m_i A_{ij} = d_j \sum_{i=0}^{k-1} \frac{m_i c_i}{x_i + y_j}. \quad 0 \leq j \leq r-1. \tag{3.36}$$

Implementation

The proposed encoder will consist of r *Cauchy cells* $C_0, C_1, \cdots, C_{r-1}$, each computing one of the check digits according to eqn 3.36. Cell C_j computes w_j. It contains the constants y_j and d_j stored in registers that can be either "hardwired", or initialised once at encoder setup time (before processing the first codeword). C_j also contains a register that holds the value of $x_i = -\alpha^{n-1-i}$. The contents of this register are initialised to $x_0 = -\alpha^{n-1}$ at the beginning of each code block, and are multiplied by the constant α^{-1} at every clock cycle. Let $w_{j'} = w_j/d_j$. C_j computes $w_{j'}$ during the first k clock cycles of a code block. In every one of those cycles, C_j receives an input of the form $m_i c_i$, computes $m_i c_i/(x_i + y_j)$, and adds the result to a register where $w_{j'}$ is accumulated. The basic structure of C_j is shown in Fig. 3.4, and the way these cells are connected to form a basic *Cauchy encoder* is shown in Fig. 3.5. It is assumed that $m_i = 0$ for $k \leq i \leq n-1$, and that m_i with $i < 0$ corresponds to message digits of a previous block. Also, indices for the sequence of constants $\{c_i\}$ are computed modulo n.

Operation

Figs.3.4 and 3.5, and the preceding discussion, show the basic computing structure of the Cauchy encoder. Addressing the input-output issues, and the transformation of the input stream u_i into an outgoing encoded stream:

Computation of $m_i c_i$: The encoder contains an additional cell, C_{pre}, that generates

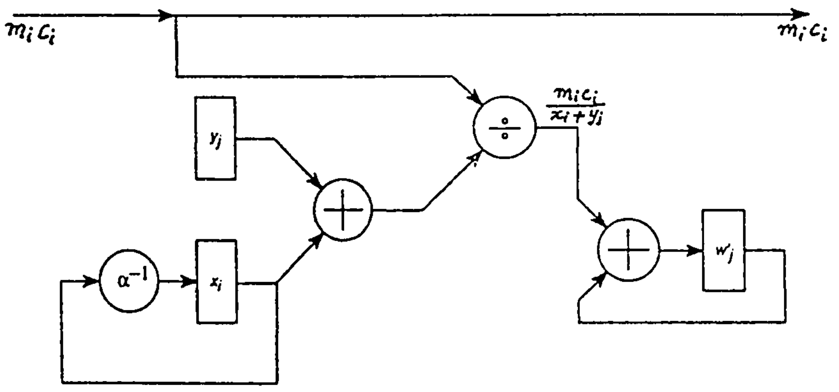


Figure 3.4: Cauchy cell C_j-basic computation

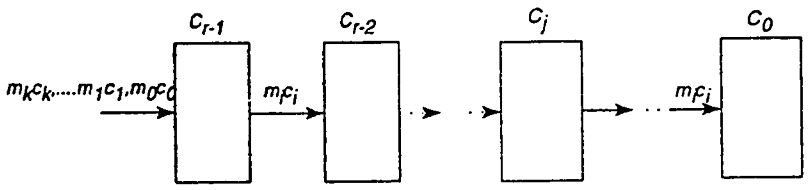


Figure 3.5: Basic Cauchy Encoder

the constants c_i and premultiplies the input stream m_i by the stream c_i, producing $m_i c_i$. The constants c_i are generated using a recursion described below. The implementation of the cell C_{pre} is not given in [16]. A suggested implementation for a particular case of (7,3) code is shown in Fig. 3.6. The modification for other codes is obvious.

Recursion for the constants c_i: From eqn. 3.34, applied to index $i+1$, the following results are obtained.

$$
\begin{aligned}
c_{i+1} &= \frac{\alpha^{-(n-1-i-1)a}}{\prod_{\substack{0 \le t \le k-1 \\ t \ne i+1}} (\alpha^{n-1-i-1} - \alpha^{n-1-t})} \\
&= \frac{\alpha^{a}\alpha^{-(n-1-i)a}}{\alpha^{-(k-1)} \prod_{\substack{0 \le t \le k-1 \\ t \ne i+1}} (\alpha^{n-1-i} - \alpha^{n-1-(t-1)})} \\
&= \alpha^{a+k-1} \frac{\alpha^{-(n-1-i)a}}{\prod_{\substack{0 \le s \le k-2 \\ s \ne i}} (\alpha^{n-1-i} - \alpha^{n-1-s})} \qquad (3.37) \\
&= \alpha^{a+k-1} \cdot \frac{\alpha^{n-1-i} - \alpha^{n-1-k+1}}{\alpha^{n-1-i} - \alpha^{n-1+1}} \cdot \frac{\alpha^{-(n-1-i)a}}{\prod_{\substack{0 \le t \le k-1 \\ t \ne i}} (\alpha^{n-1-i} - \alpha^{n-1-t})}, \\
& \qquad 0 \le \mathrm{i} \le \mathrm{k} - 2.
\end{aligned}
$$

It follows from eqn 3.34 and eqn 3.37 that

$$
c_{i+1} = \alpha^{a+k-1} \frac{\alpha^{-(i+1} - \alpha^{-k}}{\alpha^{-(i+1)} - 1} c_i. \quad 0 \le i \le k-2. \qquad (3.38)
$$

Starting with initial c_0, the sequence c_i can be generated using the recursion in eqn 3.38. The value of c_{i+1} is obtained from c_i using one division, one full multiplication,

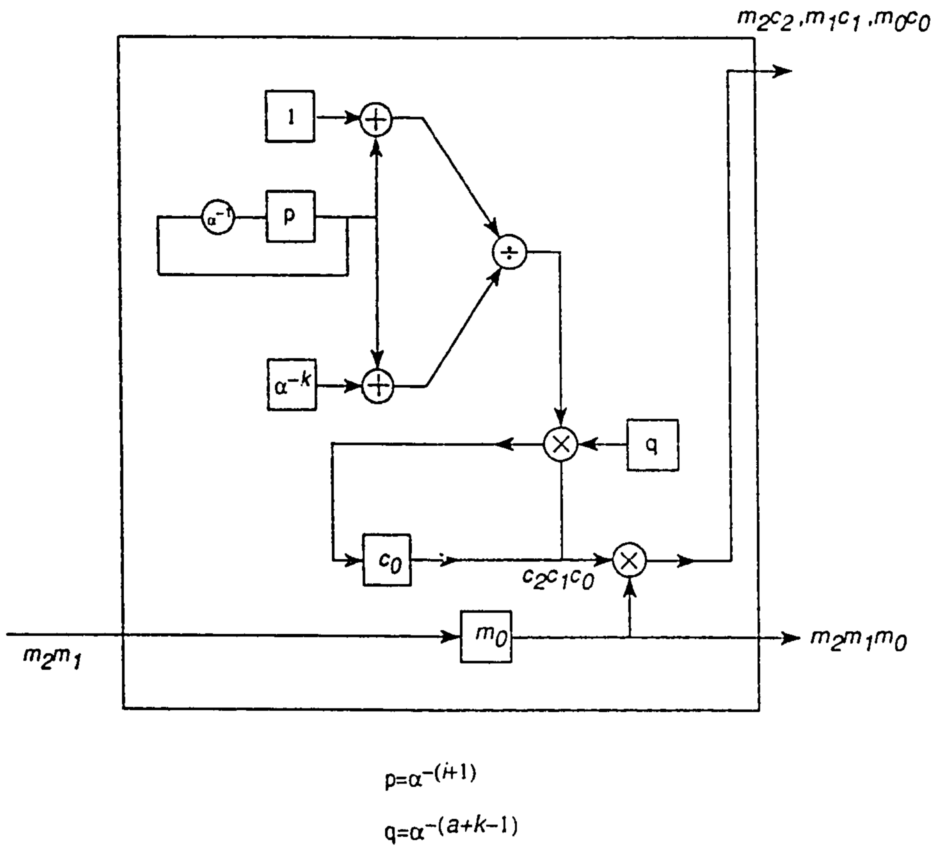


Figure 3.6: Cell C_{prc} for the (7,3) RS code

one constant multiplication, and two constant additions.

Computation of w_j: To obtain w_j, the computed w'_j is multiplied by d_j. Since the computation of w'_j takes k clock cycles out of the n cycles required to process a codeword, the product $w'_j d_j$ can be computed in the $k+1$st cycle. It should also be noted that, by storing d_j^{-1} instead of d_j, we can compute $w_j = w'_j / d_j^{-1}$ using the divider already present in the cell. The design in [16] does not indicate how it can be done. A suggested approach is to use a switch S2 to do this. The switch S2 is in position M for the first k cycles and then in position N in the $k+1$th cycle. The switch can be controlled using the same control circuit as is used for switch S1 (not shown). The complete cell C_j is shown in Fig. 3.7

Generation of the Encoded Output Stream: To generate the output stream, the encoder must reproduce the message stream $m_0, m_1, \cdots, m_{k-1}$ during the first k cycles of a code block, and produce the check stream $w_0, w_1, \cdots, w_{r-1}$ during the last r cycles. This is achieved by running an "output stream" line through the cells of the encoder. Each cell will propagate the stream m_i on its output line during the first k cycles of the block, insert its computed check digit during the $k+1$st cycle, and propagate results from previous cells (delaying them by one clock cycle) during the remaining $r-1$ cycles. The circuitry for the output stream is shown in Fig. 3.8. Switch S1 is in position A for the first k cycles of a block, in position B during the $k+1$st cycle, and in position C for the remaining cycles. Let $D_{j,i}$ denote the value at the output line of $C_j, 0 \leq j \leq r-1$, at the ith cycle of a code block.

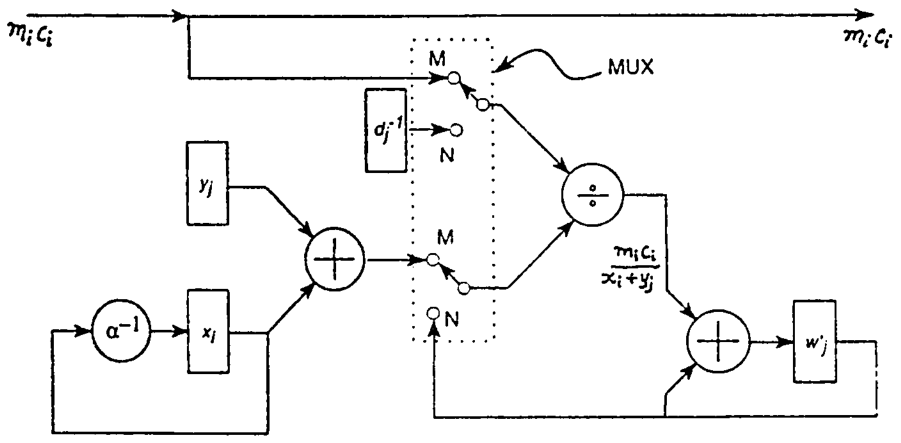


Figure 3.7: Cell C_j

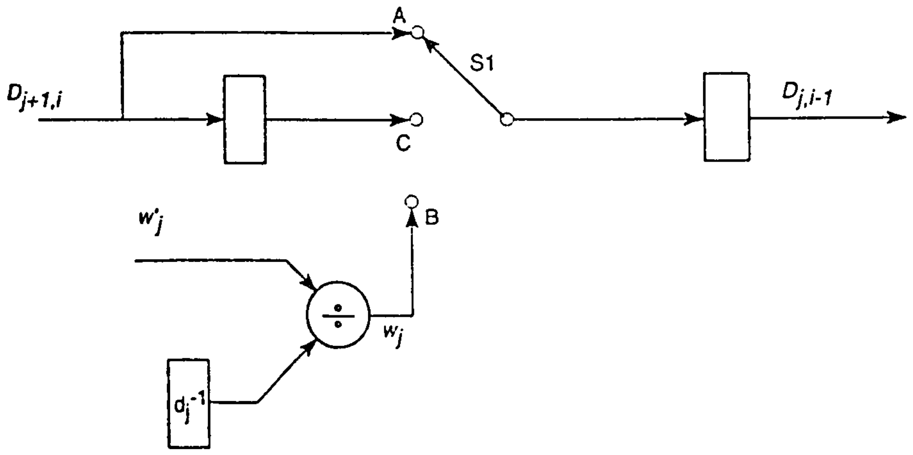


Figure 3.8: Output stream circuitry for cell C_j.

$0 \le i \le n-1$ (the cycle index i is counted according to the local clock of C_j). Also, defining $D_{r,i}$ as the value at the output line of C_{pre} at the ith cycle of a code block. Let $D_j = [D_{j,0} D_{j,1} \cdots D_{j,n-1}], 0 \le j \le r$. Then,

$$D_r = [m_0 m_1 \cdots m_{k-1} 00 \cdots 0] \, ,$$

and

$$D_j = [m_0 m_1 \cdots m_{k-1} w_j w_{j+1} \cdots w_{r-1} 0 \cdots 0] \quad 0 \le j \le r-1.$$

In particular,

$$D_0 = [m_0 m_1 \cdots m_{k-1} w_0 w_1 \cdots w_{r-1}]$$

is the desired codeword.

The computation at cell C_j can be summarised as follows:

Cycle -1	: initialise $w_j' = 0, \ x_i = x_0$
Cycles 0 to $k-1$	: compute $u_i c_i/(x_i + y_j)$, and accumulate into w_j', $0 \le i \le k-1$
Cycle k	: compute $w_j = w_j'/d_j^{-1}$: insert w_j in output stream.
Cycles k to $n-1$	: propagate results from previous cells.

This computation is synchronised by propagating a binary timing signal T_i through the cells. T_i can be generated at the preprocessing cell C_{pre}, so that T_i=1 for $0 \le i \le k-1$, and T_i=0 for $k \le i \le n-1$. This enables the Cauchy cell to recognise the beginning, the $k+1$st cycle, and the end of a code block. T_i goes through one delay unit at each Cauchy cell. The complete Cauchy encoder is shown in Fig. 3.9.

The Cauchy encoder can be greatly simplified if the constants c_i and d_j are set to 1. In this case, the resulting is not the original RS code, but a GRS code with the same error-correction capabilities. The simplified encoder does not require the preprocessing cell C_{pre}, the circuitry for transforming w'_j into w_j, or a second set of data lines (since the streams m_i and m_ic_i are identical in this case).

Another interesting property of the Cauchy encoder is that it is easily reconfigurable for variable redundancy: Any redundancy $r' \leq r$ can be accomodated by forcing the last $r - r'$ cells of the encoder to be in "message propagation" mode (switch S1 at position A) all the time, and by initialising the values of the constants d_j and y_j in each cell according to eqn 3.32 - eqn 3.35 for the desired redundancy value. The set of roots of the code is also easily reconfigurable by changing the values of d_j and y_j, as well as the value of the integer a in the initial preprocessing cell.

Example 3.3 For the running example of the RS (7,3) code, the design of the systolic encoder is as follows:The G_{sys} matrix is of the form $G_{sys} = [I \mid A]$, where I

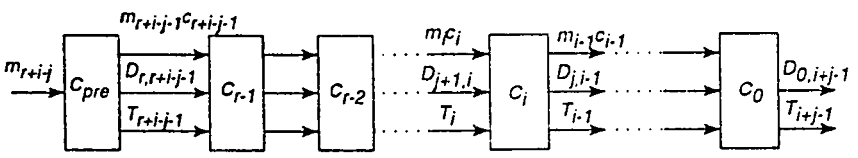


Figure 3.9: Complete Cauchy Encoder

is the identity of order 3×3 and A is a 3×4 matrix given by

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \end{bmatrix},$$

where $A_{ij} = \frac{c_i d_j}{x_i + y_j}$ $0 \leq i \leq k-1$ and $0 \leq j \leq r-1$. The calculated values of x_i, y_j, c_i and d_j are tabulated in the table below.

$x_0 = \alpha^6$	$y_0 = \alpha^3$	$c_0 = \alpha^4$	$d_0 = \alpha$
$x_1 = \alpha^5$	$y_1 = \alpha^2$	$c_1 = \alpha$	$d_1 = \alpha^6$
$x_2 = \alpha^4$	$y_2 = \alpha$	$c_2 = 1$	$d_2 = \alpha^5$
	$y_3 = 1$		$d_3 = \alpha^4$

Based on these values, the parity check bits w_j are calculated as given below.

$$w_0 = m_0\alpha + m_1\alpha^6 + m_2\alpha^2$$

$$w_1 = m_0\alpha^3 + m_1\alpha^4 + m_2\alpha^5$$

$$w_2 = m_0\alpha^4 + m_1 + m_2\alpha^3$$

$$w_3 = m_0\alpha^6 + m_1\alpha + m_2\alpha^6$$

The G_{sys} matrix is therefore given by

$$G_{sys} = \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & \alpha & \alpha^3 & \alpha^4 & \alpha^6 \\ 0 & 1 & 0 & 1 & \alpha^4 & 1 & \alpha \\ 0 & 0 & 1 & \alpha^2 & \alpha^5 & \alpha^3 & \alpha^6 \end{array}\right].$$

The proposed encoder has *4 Cauchy cells*, C_0, C_1, C_2 and C_3 computing w_0, w_1, w_2 and w_3 respectively. Each cell contains a register to hold the value α^6 which is multiplied by α^{-1} at every clock cycle. The complete Cauchy encoder is shown in Fig. 3.10. The cell C_{pre} is shown in Fig. 3.6.

3.3.3 Conclusions

A modification to the circuit given in [16] is that the delay element in the path of the stream u_ic_i has been removed since u_ic_i should be available to all the cells at the same time during the first k cycles. Another modification is the suggested circuitry for the use of same divider to divide the term w_j' by d_j^{-1}. By doing so we are reducing area. This reduction comes as the same control circuitry is being used which is used at the output.

The foregoing proposed encoder for the RS codes does not require any global feedback as is required in the case of the traditional feedback shift register encoder. This enhances the throughput of the encoder. The encoder described is systematic, noninterleaved and uses only forward data path. The architecture is suitable for very high-speed applications of the order of Gigabits/sec.

3.4 Application: Decoding of Reed-Solomon Codes

Systolic array architectures can be applied to the various steps involved in decoding non-binary block codes. Systolic array architectures can be used in all stages of the decoding process including the syndrome calculation, key equation solution using

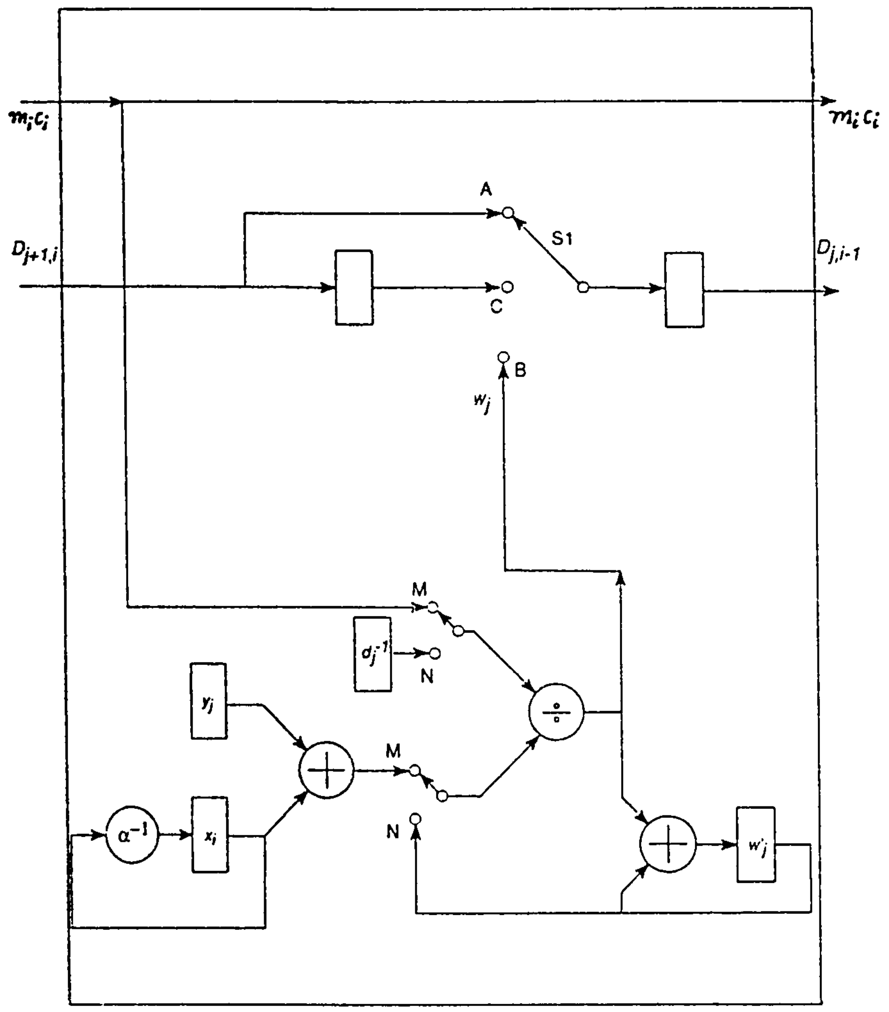


Figure 3.10: PE for the systolic encoder of (7,3) RS code

Euclid's method and error evaluation. In this section a modified decoder for the running example of the (7,3) RS code is presented. The throughput of the decoder is effectively determined by the speed of the multipliers used. Fig. 3.11 shows an overall block diagram for a pipeline or systolic based RS decoder. At the highest level the pipeline consists of separate blocks for calculating the syndrome, solving the key equation and locating the errors. Each of these blocks is in turn made up of linear systolic arrays. In the following sections the implementations of each block will be described.

3.4.1 Syndrome Calculation

Defining the received polynomial as

$$v(x) = \sum_{i=0}^{n-1} v_i x^i$$

The syndrome computation

$$S_k = \sum_{i=0}^{n-1} v_i (\alpha^k)^i \qquad 1 \leq k \leq 2t$$

is an evaluation of a polynomial of length n on $2t$ points. Since $n > 2t$, it is best to compute all syndromes simultaneously. S_k is computed in the following manner:

$$S_k = \cdots (v_{n-1}\alpha^k + v_{n-2})\alpha^k + \cdots + v_1)\alpha^k + v_0.$$

It should be noted that v_{n-1} is the first received symbol. Starting from the innermost parenthesis, syndrome S_k is gradually computed as the v_i are received. After v_0 is

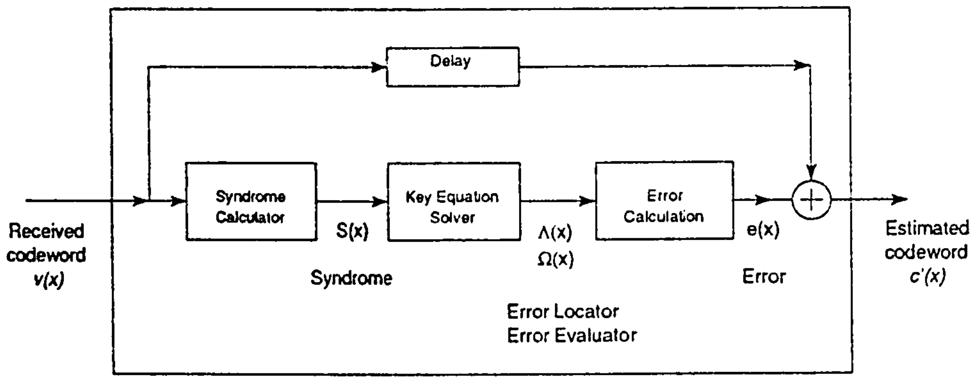


Figure 3.11: Block diagram for a pipeline RS decoder

entered, all $2t$ syndrome computations are completed at the same time. They are ready to be shifted out serially at that point. A systolic array design of syndrome computation circuit is shown in Fig. 3.12.

Initially the accumulator registers A_j are cleared and the argument registers contain the value of the argument α^1 to α^{2t}, which are constants and may be hardwired. The symbols from the received codeword are then sent to all the cells simultaneously. When the complete codeword has been processed, the syndromes are sent to the key equation solver.

3.4.2 Key Equation Solver

Implementing the Extended Euclidean Algorithm using Systolic Arrays

The *extended Euclidean algorithm* (XEA) is a well known method for finding the *greatest common divisor* (or *gcd*) of two polynomials $f(x)$ and $g(x)$ whose coefficients lie in a field K as well as polynomials $\alpha(x)$ and $\beta(x)$ such that

$$gcd[f(x), g(x)] \;=\; \alpha(x)f(x) + \beta(x)g(x) \;=\; \gamma(x).$$

Brent and Kung [19] have given a systolic design for the (ordinary, unextended) Euclidean algorithm and indicated how their methods might be developed to carry out the extended algorithm. Several authors have proposed systolic RS decoders [18, 31, 17] in which the key equation is solved using a systolic XEA which improved on existing designs. In [24], a systolic XEA is described which improved on existing

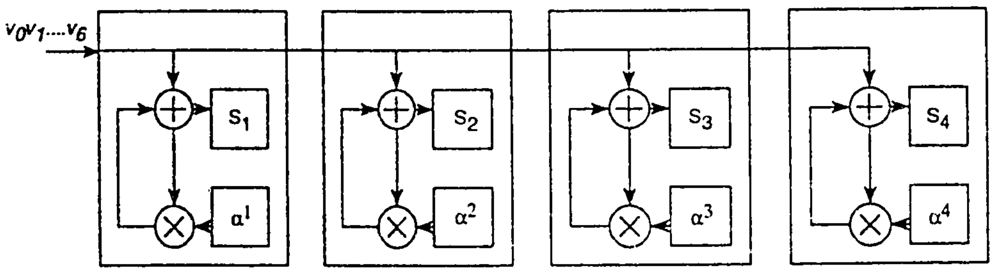


Figure 3.12: Systolic syndrome calculator for the RS (7,3) code

designs. Below is presented the systolic design to implement the key equation solver presented in [25].

The Algorithm

The basis of the XEA is the *polynomial remainder sequence* $\gamma_{-1}, \gamma_0, \cdots, \gamma_{n+1}$ derived by initialising $\gamma_{-1}(x) = f(x), \gamma_0(x) = g(x)$ (assuming $\delta f \geq \delta g$ where δ denotes degree) and setting γ_{j+2} to be the remainder on division of γ_j by γ_{j+1}. Here $\gamma_{n+1} = 0$ and $\gamma_n = gcd(f, g)$. Viewing division as repeated shifting, cross multiplication by leading coefficients and subtraction one obtains the *left-shift* XEA in which two vectors of three polynomials **f**=$(f1, f2, f3)$ and **g**=$(g1, g2, g3)$ are carried along representing successive iterates of the polynomials $f1(x), f2(x), f3(x)$. This is converted into a *separated action* form XARD, in which each operation is one of ADVANCE(**g**), REDUCE(**f**), DELAY(**g**) or SWAP(**f**,**g**). For a systolic implementation, these actions are combined in order to force each processing element (PE) to reduce the degree of the current $f3(x)$ by at least 1. The *combined action* version COMB_XARD algorithm is given in Fig. 3.13. The parameter t appearing in the definition of the variable $< decwina >$ corresponds to the error-correction capability of an RS code. For the full XEA algorithm its value is taken as 0.

Implementation

The circuit of the PE for the systolic COMB_XARD algorithm uses 3-bit coefficients corresponding to elements from the Galois field $GF(8)$. The cell architecture in Fig. 3.14 comprises the polynomial reduction circuitry and subsequent delay control to

```
Input: f, g polynomials with coefficients in K, not both zero.
Output: f where f1f + f2g = f3 = gcd(f,g).
procedure comb_XARD gcd(f,g)
{Initialisation}
f3:=f; g3:=g;
if $\delta$f3 < $\delta$ g3 then swap(f3,g3);
f1:=1; f2:=0; g1:=0: g2:=1;
$start$ :=$x^{\delta f3}$; {align start - regarded as a polynomial - with the
leading coefficient of f3}
$gshift$ :=0 { g is multiplied by $x^{gshift}$ }
$decwina$ := $\delta$f3 - $t$ + 1 - $gshift$;
$decwini$ := $decwina$;
{Calculation}
{f,g and control signals are passed through a pipeline of
systolic cells whose algorithm follows}
{Cell operation}
{L'() denotes the coefficient coincident with the start signal asserted}
if $decwina$ > 0 then
    case (L'(f3) = 0, L'(g3) = 0) of
    {Adjust} (T,T):begin
            $decwina$ − −;
            delay(f1, f2, g1, g2, $start$);
                   end;
    {Reduce} (F, F):
                (T, F): begin
                    f := L'(g3)f - L'(f3)g;
                    if ($gshift$ > 0) then
                      begin
    {Delay}             delay g:
                          delay f1, f2, g1, g2, $start$:
                          $gshift$ − −;
                          end;
                     else
                          begin
    {Swap}              swap(f,g)
    {Advance}           delay(f, $start$):
                        $gshift$ + +;
                        $decwina$ − −;
                          end
```

```
  {Advance}(F,T):    begin
                       delay(f, $start$);
                       $gshift++$;
                       $decwina--$;
                     end;
  end{case}
else
  {Null}
{Completion}
if ($gshift > 0$)
{g multiplied by $x^{gshift}$ }
begin
   for n:=1 to $gshift$ do
        begin
        delay(g);
        $decwina++$;
   end;
$swap$(f,g
{Result always in f }
end;
{f3 is too far advanced by $decwina$ }
for n:=1 to ($decwini - decwina$) do
     delay(f3)
return (f/L(f3));
```

Figure 3.13: COMB_XARD

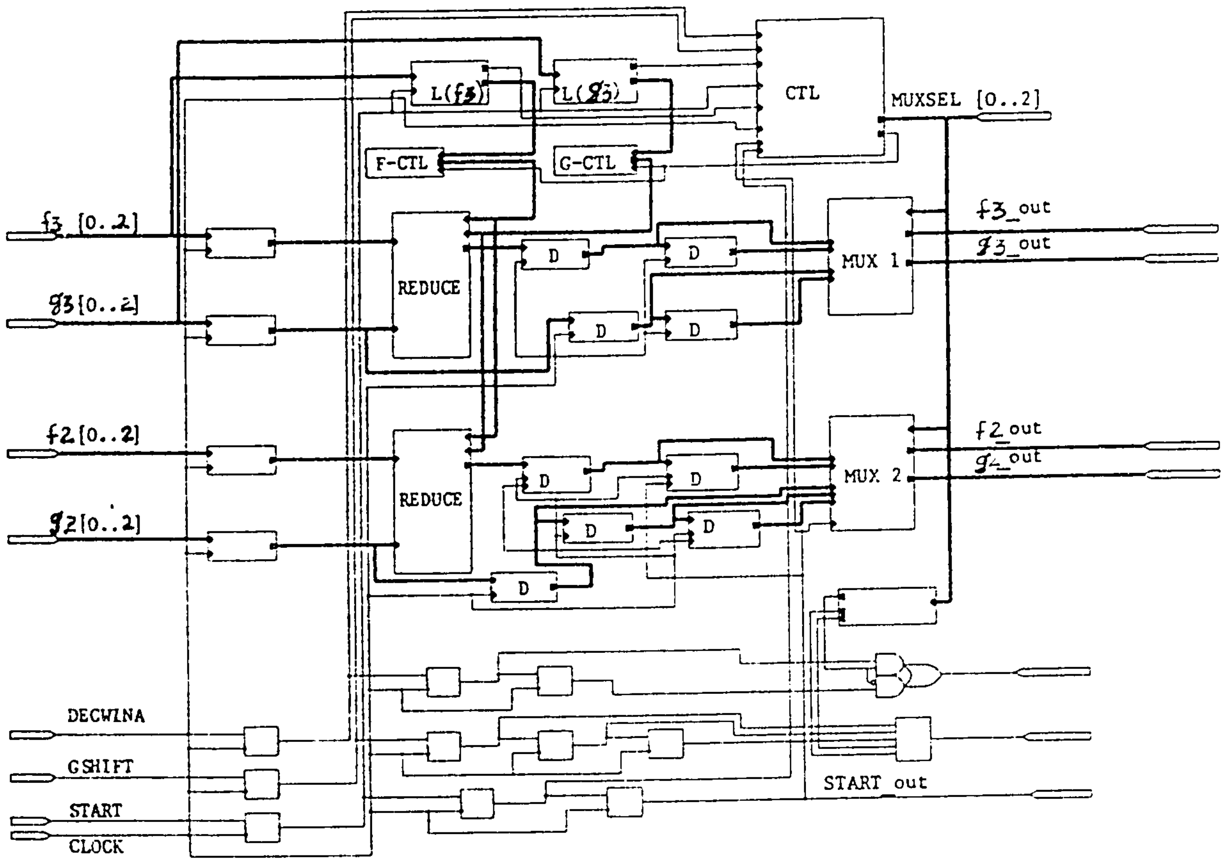


Figure 3.14: Processing Element

implement the advance, delay and swapping functions. The action of the cell is determined by the zero/non-zero value of the leading coefficients and the $< gshift >$ and $< decwina >$ signals.

The operation of the PE is as follows. On the rising edge of a $< start >$ signal the leading (highest degree) coefficients of f3 and g3 are latched for subsequent multiplication in a reduction stage and at the same time tested for zero/non-zero status. The result of this comparison along with the values (1 or 0) of $< gshift >$ and $< decwina >$ determine the contents of the MUXSEL[0:2] bus for the current operation. MUXSEL[0:2] defines the input selection of the multplixers MUX1, MUX2 which ensure correct time alignment of the output polynomials. The increment and decrement of the $< gshift >$ and $< decwina >$ signals is also determined by the output multiplexers which select the appropriately delayed input signal. These multiplexers are controlled by logical function of the MUXSEL[0:2] bus value determined in the CTL2 block.

The reduction circuitry comprises two multipliers which cross multiply **f** and **g** by the leading coefficients of f_3 and g_3 respectively and a subtracter to obtain the difference of the results. The result is the new **f** in which the degree of f_3 has been reduced by at least 1. The multiplier uses an array of binary inner product cells combining the summation and reduction associated with finite field multiplication. The coefficients G[0..2] of the field generator polynomial are hardwired in this case but they can easily be brought out to pads for external programmability. The mul-

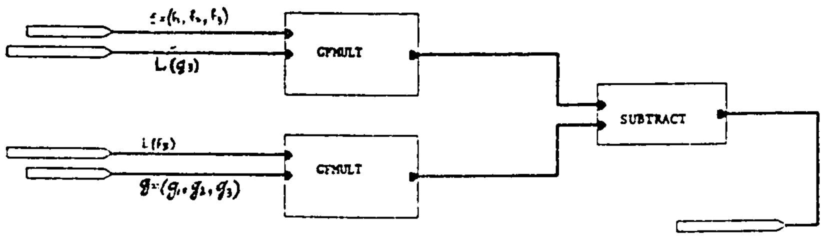

$\underline{f}=(f_1, f_2, f_3)$
$L(\bar{g}_3)$
$L(f_3)$
$\underline{g}=(g_1, g_2, g_3)$
CFMULT
CFMULT
SUBTRACT

Figure 3.15: Reduction Circuit

tiplier can be one of the available multipliers e.g., the one proposed by Wang and Lin [12]. Subtraction is a bitwise XOR operation. Exceptions to the normal multiplication occur in the NULL and ADJUST operations when the leading coefficient values (indicated by L) are overwritten by $L(f_3)$=0 and $L(g_3)$=1 to ensure that f passes unchanged. These operations are performed by the blocks labelled F_CTL. G_CTL

Fig. 3.16 gives a block diagram of the connections of the PE's to calculate the error locator polynomial and the error evaluator polynomial, where f1 and g1 have been suppressed as they are not involved in this particular computation.

This design improves over the one presented in [23] which uses the design given in [24]. The systolic array for the XEA used in the proposed design of the decoder is an improved version of that given in [24] in terms of cell complexity.

3.4.3 Error Calculation

The error calculation part requires the evaluation of the error locator polynomial $\Lambda(x)$ for α^{-1}. $0 \leq i \leq n-1$ to find its inverse roots. If

$$\Lambda(\alpha^{-1}) = 0$$

then v_i is a corrupted symbol. The inverse elements of $GF(2^3)$ are generated by an LFSR, as shown in Fig. 3.17 for the generator polynomial $x^3 + x + 1$. Note the correspondence between the feedback connections and the nonzero coefficients

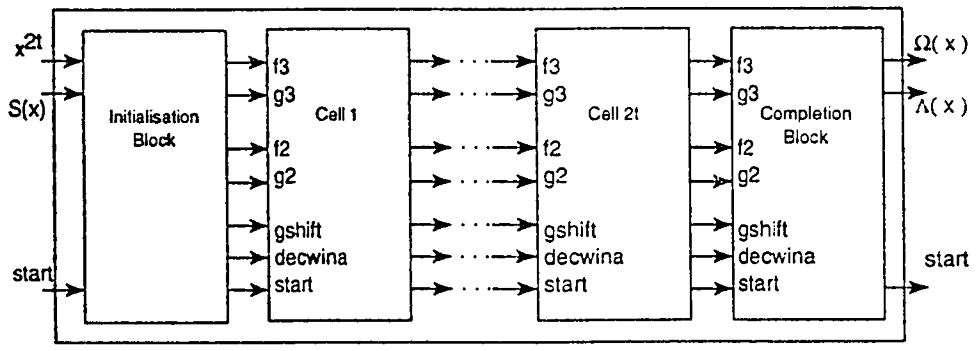


Figure 3.16: Block diagram of the key equation solver based on the COMB_XARD algorithm

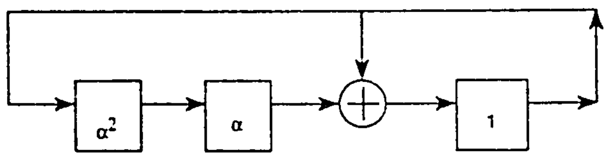


Figure 3.17: A circuit that generates the elements of $GF(2^m)$ in reverse order

of the generator polynomial. This is as shown in [1].

The corresponding error magnitudes are computed by evaluating $\Omega(x)$ and $\Lambda'(x)$ for α^{-i}, $i = 0, 1, \cdots, n-1$. That is the error magnitude is given by

$$e_i = \frac{\Omega(\alpha^{-i})}{\Lambda'(\alpha^{-i})} \qquad 0 \leq i \leq n-1.$$

The error equation requires that $\Lambda(x)$ is formally differentiated and this is relatively straightforward to implement for $GF(2^m)$. Consider the polynomial $A(x)$ over $GF(2^m)$:

$$\begin{aligned} A(x) &= a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1 x + a_0 \\ A'(x) &= (m-1)a_{m-1}x^{m-2} + (m-2)a_{m-2}x^{m-3} + \cdots + a_1 \\ &= a_{m-1}x^{m-2} + a_{m-3}x^{m-4} + \cdots + a_1 \qquad \text{for } m \text{ even} \end{aligned} \tag{3.39}$$

All the coefficients of even powers of x in $\Lambda(x)$ and the odd powers remain the same, since addition corresponds to the XOR operation: and the resultant polynomial is divided by x, which can be implemented as a shift right or delay.

The circuit that performs the differentiation of $\Lambda(x)$ in $GF(2^3)$ is shown as a part of Fig. 3.19. Fig.3.19 contains a block diagram showing the various arrays and the auxiliary blocks required. The implementation is based on Fig. 3.18 where the argument and the accumulator are passed through an array of cells holding the polynomial coefficients.

The approach presented here is opposite to the one given in [23]. This paper

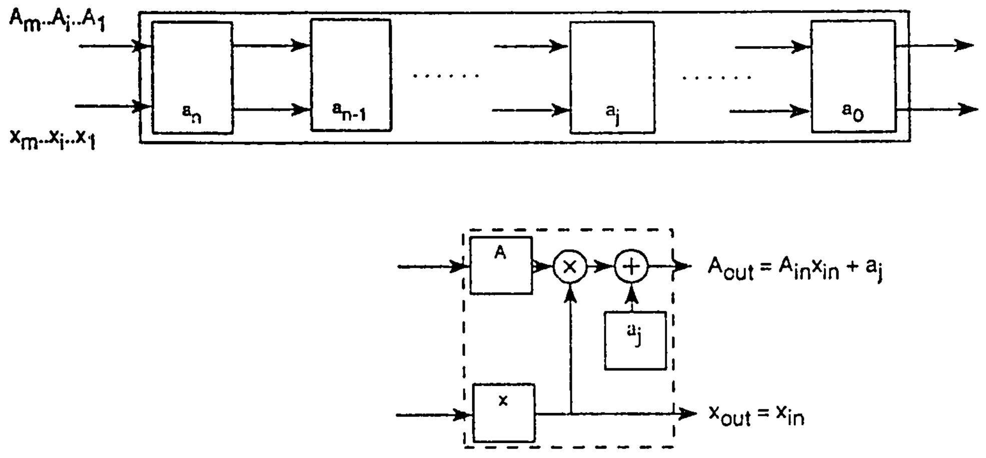


Figure 3.18: A systolic architecture for polynomial evaluation

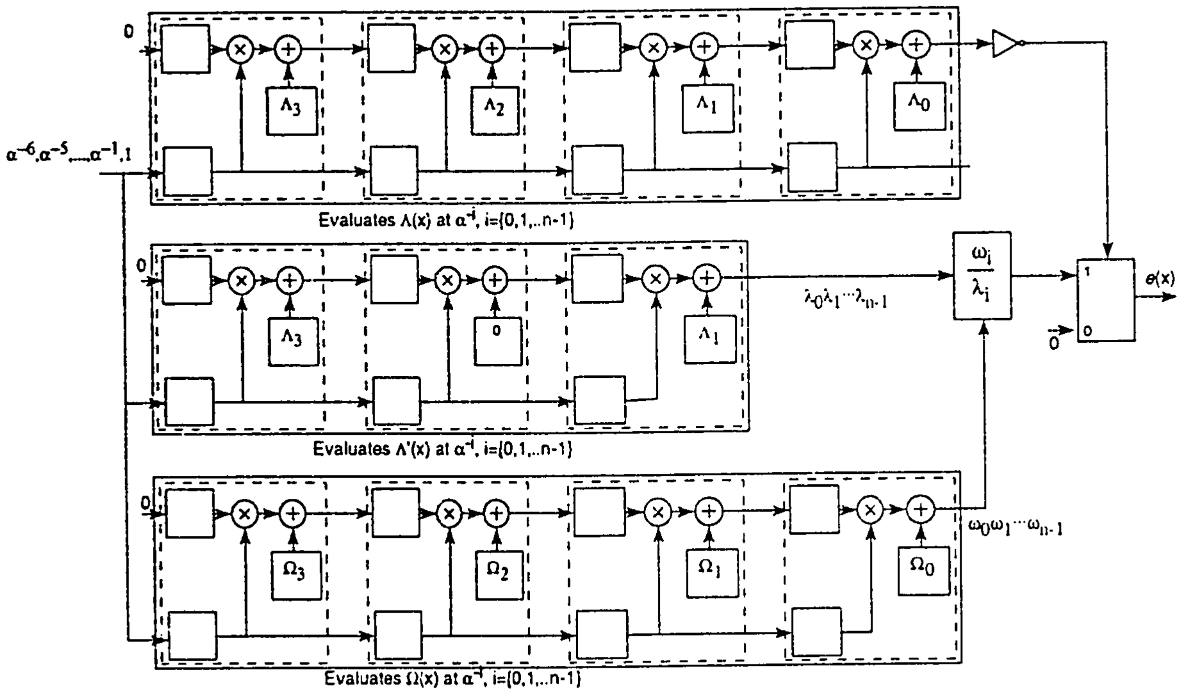

0
α⁻⁶,α⁻⁵,...,α⁻¹,1
Λ3
Λ2
Λ1
Λ0
Evaluates Λ(x) at α⁻ⁱ, i={0,1,..n-1}
0
Λ3
0
Λ1
λ0λ1···λn-1
Evaluates Λ'(x) at α⁻ⁱ, i={0,1,..n-1}
ωi/λi
1
0
e(x)
0
Ω3
Ω2
Ω1
Ω0
ω0ω1···ωn-1
Evaluates Ω(x) at α⁻ⁱ, i={0,1,..n-1}

Figure 3.19: Error Evaluation Block diagram

evaluates the error locator at α^i and assumes that an error has occured at α^i,i.e.,

$$\Lambda(\alpha^i) = 0 \qquad \longrightarrow \text{ error at } \alpha^i$$

$$e_i = \frac{\Omega(\alpha^i)}{\Lambda'(\alpha^i)}.$$

Applying this result to the running example has not given correct results.

Besides differentiation, the remainder of the error location operation involves only polynomial evaluations and the generation of the sequence α^{n-1} to α^0. Polynomial evaluation is required for $\Lambda(x)$, $\Lambda'(x)$ and $\Omega(x)$. For pipeline implementations it is easier to evaluate all three for all values of α^{-i} and to ignore $\Lambda'(\alpha^{-i})$ and $\Omega(\alpha^{-i})$ except where $\Lambda(\alpha^{-i}) = 0$.

3.4.4 Error Correction

The error correction is implemented by adding the error and received codeword. The received codeword is delayed until its corresponding error polynomial has been calculated. The adder is implemented using XOR gates.

3.4.5 Exceptional Errors

There are two special error conditions that must be considered: no errors and more than t errors. In the case of no errors the syndrome will be the zero polynomial, $S(x) = 0$. When the key equation solver terminates, then $\Lambda(x) = 1$ and $\Omega(x) = 0$.

Therefore, the error locator polynomial will equal 1 for all error locations which results in no error correction of the received codeword.

When more than t errors occur, the design may be expanded to detect incorrect decoding. A suggested method of implementing this is to count the number of error locations and compare this with the degree of the error locator polynomial. If the two are not equal an uncorrectable codeword has been detected.

Example 3.4 For the running example of the RS (7,3) code, let the received word be $v(x) = \alpha^4 x^6 + \alpha^3 x^3 + \alpha^6 x + \alpha^4$ and the generator of the code be $g(x) = x^4 + \alpha^3 x^3 + x^2 + \alpha x + \alpha^3$

The step by step procedure of error correction is shown in the Appendix.

3.4.6 Conclusions

Decoding RS codes can be implemented using systolic arrays. The architecture presented here improves upon the previous designs. Moreover, the PE in the calculation of XEA is significantly simpler than those in previous designs. For example the degrees of the given polynomials f and g and their updates are not carried and the stopping conditions (in the RS decoder application) has been reduced to a simple zero/nonzero comparison. A further improvement lies in the way, the growth in the degrees on the auxilliary polynomials f1 and f2 are handled, avoiding the possibility of these polynomials becoming "too far advanced" and the consequent loss of data.

Bibliography

[1] W. W. Peterson and Jr. E. J. Weldon. *"Error-Correcting Codes"*. MIT Press. Cambridge. MA. 1972.

[2] Shu Lin. *"An Introduction to Error-Correcting Codes "*. Prentice Hall. Inc. Englewood Cliffs. NJ. 1970.

[3] F. J. Macwilliams and N. J. A. Sloane. *"The Theory of Error-Correcting Codes "*. North Holland. NY. 1977.

[4] B. Benjauthrit and I. S. Reed. "Galois Switching Functions and their Applications ". *IEEE Transactions on Computers*. C-25:78–86. January 1976.

[5] D. E. R. Denning. *"Cryptography and Data Security "*. Addison-Wesley. Reading. MA. 1983.

[6] I. S. Reed and T. K. Truong. "The use of Finite Finite Fields to Compute Convolutions ". *IEEE Transactions on Information Theory*. IT-21:208–213. March 1975.

[7] Toshiyo Itoh and Shigeo Tsujii. "Structure of Parallel Multipliers for a Class of Fields $GF(2^m)$". *Information and Computation*, 83:21–40, 1989.

[8] Din Y. Pei, Charles C. Wang, and Jim K. Omura. "Normal Basis of Finite Field $GF(2^m)$". *IEEE Transactions on Information Theory*, IT-32(2):285–287, March 1986.

[9] M. Anwarul Hasan and Vijay K. Bhargava. "Bit Serial Systolic Divider and Multiplier for Finite Fields $GF(2^m)$ ". *IEEE Transactions on Computers*, 41(8):972–980, August 1992.

[10] C.S.Yeh, Irving S. Reed, and T.K.Truong. "Systolic Multipliers for Finite Fields $GF(2^m)$". *IEEE Transactions on Computers*, 33:357–360, April 1984.

[11] A. Sengupta S. Bandyopadhyay. "Algorithms for multiplication in Galois field for implementation using systolic arrays". *IEE Proceedings*, 135(6):336–339, November 1988.

[12] Chin-Liang Wang and Jung-Lung Lin. "Systolic Array Implementation of Multipliers for Finite Fields $GF(2^m)$". *IEEE Transactions on Circuits and Systems*, 38(7):796–800, July 1991.

[13] B. B. Zhou. " A New Bit-Serial Systolic Multiplier Over $GF(2^m)$". *IEEE Transactions on Computers*, 37(6):749–751, June 1988.

[14] S.T.J. Flenn, D. Taylor, and M. Benaissa. "Division over $GF(2^m)$". *Electronics Letters*, 28(24):2259–61, November 1992.

[15] Chin-Liang Wang and Jung-Lung Lin. "A Systolic Architecture for computing Inverses and Divisions in Finite Fields $GF(2^m)$". *IEEE Transactions on Computers*, 42(9):1141–46, September 1993.

[16] Gadiel Seroussi. "A Systolic Reed-Solomon Encoder". *IEEE Transactions on Information Theory*, 37:1217–1220, July 1991.

[17] Howard M. Shao and Irving S. Reed. "On the VLSI Design of a Pipeline Reed-Solomon Decoder Using Systolic Arrays". *IEEE Transactions on Computers*, 37(10):1273–1280, October 1988.

[18] Masayuki Kimura, Hideki Imai, and Yasanuri Dohi. "Systolic Decoder for Reed-Solomon Codes". *Electronics and Communications in Japan, Part 1*, 70(8):731–6, 1987.

[19] Richard P. Brent and H. T. Kung. "Systolic VLSI Arrays for Polynomial GCD Computation". *IEEE Transactions on Computers*, 33(8):731–6, August 1984.

[20] Tetsuo Iwaki, Toshihisa Tanaka, Eiji Yamada, Tohru Okuda, and Taizoh Sasada. "Architecture of a High Speed Reed-Solomon Decoder". *IEEE Transactions on Consumer Electronics*, 40(1):75–81, February 1994.

[21] Hirokazu Okano and Hideki Imai. "A Construction Method of High Speed Decoders Using ROM's for Bose-Chaudhuri-Hocquenghem and Reed-Solomon Codes". *IEEE Transactions on Computers*, 36(10):1165–71, October 1987.

[22] Shyue-Win Wei and Che-Ho Wei. "High Speed Decoder of Reed-Solomon Codes". *IEEE Transactions on Communications*, 41(11):1588–93, November 1993.

[23] John Nelson, Abdur Rahman, and Eamonn McQuade. "Systolic Architectures for Decoding Reed-Solomon Codes". *International Conference on Application Specific Specific Array Processors*, pages 67–77, 1990.

[24] P. Fitzpatrick, J. Nelson, and G. Norton. "A Systolic Version of the Extended Euclidean Algorithm". *Proceedings of the Conference on Systolic Array Processors - Killarney 1988*, pages 477–486, 1989.

[25] Rory Doyle, Patrick Fitzpatrick, and John Nelson. "An Improved Systolic Extended Euclidean Algorithm for Reed-Solomon Decoding: Design and Implementation". *International Conference on Application Specific Specific Array Processors*, pages 448–456, 1990.

[26] Charles C. Wang, T.K.Truong, Howard M. Shao, Leslie J. Deutsch, Jim K. Omura, and Irving S.Reed. "VLSI Architectures for Computing Multiplica-

tions and Inverses in $GF(2^m)$". *IEEE Transactions on Computers*. 34:709–717. August 1985.

[27] M. Z. Wang M. A. Hasan and V. K. Bhargava. "A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields ". *IEEE Transactions on Computers*. 42(10):1278–1280. October 1993.

[28] P. Andrew Scott. Stanford E tavares. and Lloyd Peppard. "A Fast VLSI Multiplier for $GF(2^m)$". *IEEE Journal on Selected Areas in Communications*. SAC-4:62–66. January 1986.

[29] G. Strang. "*Linear Algebra and Its Applications* ". Academic. New York. 1980.

[30] R. M. Roth and G. Seroussi. "On Generator Matrices of MDS codes". *IEEE Transactions on Information Theory*. IT-31(6):826–830. November 1985.

[31] H. M. Shao. T. K.Truong. L. J. Deutch. J. H. Yeun. and I. S. Reed. "AVLSI Design of a Pipeline Reed-Solomon Decoder". *IEEE Transactions on Computers*. 34:393–402. 1985.

Appendix

Syndrome Calculation:

$$S(x) = \sum_{j=1}^{2t=4} S_j x^j \quad \text{and} \quad S_j = v(\alpha^j) = \sum_{i=0}^{n-1=6} v_i \alpha^{ji}$$

$S_1 = 1$

$S_2 = \alpha^2$ $\qquad$ $\underline{S(x) = \alpha^4 x^3 + \alpha^5 x^2 + \alpha^2 x + 1}$ is the syndrome polynomial

$S_3 = \alpha^5$

$S_4 = \alpha^4$ $\qquad$ $2t = 4$, therefore $x^{2t} = x^4$

Key Equation Solving:

{ *Initialisation* }

$\mathbf{f} = (f1, f2, f3) = (1, 0, x^4)$

$\mathbf{g} = (g1, g2, g3) = (0, 1, \alpha^3 x^3 + \alpha^5 x^2 + \alpha^2 x + 1)$

start = x^4 , gshift = 0, decwina = 4 - 2 + 1 - 0 = 3, decwini = 3

{ *Calculation* }

```
if ( decwina >0 )
Case( L'(f3) = 0, L'(g3) = 0) of
FT      delay(f, start) -> g = gx, therefore g = $(0, x, \alpha^4x^4 + \alpha^5x^3 + \alpha^2x^2 + x)$
        gshift ++ = 1, decwina -- = 2

FF      f = L'(g3)f - L'(f3)g
          $= (\alpha^4, 0, \alpha^4x^4) - (0, x, \alpha^4x^4 + \alpha^5x^3 + \alpha^2x^2 + x) = (\alpha^4, x, \alpha^5x^3 + \alpha^2x^2 + x)$
        gshift > 0,
           delay g -> (f = fx) -> f = $(\alpha^4, x, \alpha^5x^3 + \alpha^2x^2 + x)$
           delay(f1,f2,g1,g2,start)
           f = $(\alpha^4, x, \alpha^5x^4 + \alpha^2x^3 + x^2)$, g = $(0, 1, \alpha^4x^4 + \alpha^5x^3 + \alpha^2x^2 + x)$
        gshift -- =0

FF      f = L'(g3)f - L'(f3)g
          $= (\alpha, \alpha^4x, \alpha^2x^4 + \alpha^6x^3 + \alpha^4x^2) - (0, \alpha^5, \alpha^2x^4 + \alpha^3x^3 + x^2 + \alpha^5x)$
          $= (\alpha, \alpha^4x + \alpha^5, \alpha^4x^3 + \alpha^5x^2 + \alpha^5x)$
        gshift = 0
           swap(f,g)
           g = $(\alpha, \alpha^4x + \alpha^5, \alpha^4x^3 + \alpha^5x^2 + \alpha^5x)$
           f = $(0, 1, \alpha^4x^4 + \alpha^5x^3 + \alpha^2x^2 + x)$
           delay(f, start) -> g = gx
           g = $(\alpha x, \alpha^4x^2 + \alpha^5x, \alpha^4x^4 + \alpha^5x^3 + \alpha^5x^2)$
        gshift ++= 1, decwina -- = 1

FF      f = L'(g3)f - L'(f3)g
           $(0, \alpha^4, \alpha x^4 + \alpha^2x^3 + \alpha^6x^2 + \alpha^4x) - (\alpha^5x, \alpha x^2 + \alpha^2x, \alpha x^4 + \alpha^2x^3 + \alpha^2x^2)$
           f = $(\alpha^5x, \alpha x^2 + \alpha^2x + \alpha^4, x^2 + \alpha^4x)$
        gshift > 0, delay g -> (f = fx) -> f = $(\alpha^5x^2, \alpha x^3 + \alpha^2x^2 + \alpha^4x, x^3 + \alpha^4x^2)$
        delay(f1, f2, g1, g2, start)
        f = $(\alpha^5x, \alpha x^2 + \alpha^2x + \alpha^4, x^3 + \alpha^4x^2)$ ,  g = $(\alpha, \alpha^4x + \alpha^5, \alpha^4x^4 + \alpha^5x^3 + \alpha^5x^2)$
        gshift -- = 0
```

```
TF    f = L'(g3)f - L'(f3)g
        = ($\alpha^2x$, $\alpha^5x^2 + \alpha^6x + \alpha$, $\alpha^4x^3 + \alpha x^2$) - (0)
      gshift = 0
        swap(f,g)
      f = ($\alpha$, $\alpha^4x + \alpha^5$, $\alpha^4x^4 + \alpha^5x^3 + \alpha^5x^2$),  g = ($\alpha^2x$, $\alpha^5x^2 + \alpha^6x + \alpha$, $\alpha^4x^3 + \alpha x^2$)
        delay(f, start) -> (g = gx) -> g = ($\alpha^2x^2$, $\alpha^5x^3 + \alpha^6x^2 + \alpha x$, $\alpha^4x^4 + \alpha x^3$)
      gshift ++ = 1
      decwina -- = 0
```

{ *Completion* }

```
gshift > 0,
for n = 1 to 1
        begin :   delay(g) -> g = g/x = ($\alpha^2x$, $\alpha^5x^2 + \alpha^6x + \alpha$, $\alpha^4x^3 + \alpha x^2$)

decwina = 1
        swap(f,g) : f = ($\alpha^2x$, $\alpha^5x^2 + \alpha^6x + \alpha$, $\alpha^4x^3 + \alpha x^2$),
                    g = ($\alpha$, $\alpha^4x + \alpha^5$, $\alpha^4x^4 + \alpha^5x^3 + \alpha^5x^2$)

for  n = 1 to (3 - 1) do

        delay(f3)          n = 1:  $\alpha^4x^2 + \alpha x$
                           n = 2:  $\alpha^4x + \alpha$

return (f / L(f3) = ($\alpha^2x$, $\alpha^5x^2 + \alpha^6x + \alpha$, $\alpha^4x + \alpha$) /  $\alpha^4$
```

$f2 = \alpha x^2 + \alpha^2x + \alpha^4 = \Lambda(x)$

$f3 = x + \alpha^4 = \Omega(x)$

$\Lambda(x) = \alpha x^2 + \alpha^2x + \alpha^4$

Evaluating the error locator polynomial at $1, \alpha^{-1}, \ldots\ldots, \alpha^{-6}$

$\Lambda(1) = \alpha + \alpha^2 + \alpha^4 = 0$

$\Lambda(\alpha^{-1}) = \alpha^6 + \alpha + \alpha^4 = 1$

$\Lambda(\alpha^{-2}) = \alpha^4 + 1 + \alpha^4 = 1$

$\Lambda(\alpha^{-3}) = \alpha^2 + \alpha^6 + \alpha^4 = \alpha^5$

$\Lambda(\alpha^{-4}) = \alpha^7 + \alpha^5 + \alpha^4 = 0$

$\Lambda(\alpha^{-5}) = \alpha^5 + \alpha^4 + \alpha^4 = \alpha^5$

$\Lambda(\alpha^{-6}) = \alpha^3 + \alpha^3 + \alpha^4 = \alpha^4$

Therefore the error locations are $\alpha^0 = 1$ and $\alpha^{-3} = \alpha^4$

$\Omega(x) = x + \alpha^4$

$\Lambda'(x) = 2\alpha x + \alpha^2 = \alpha^2$

Error calculation

$e_i = \Omega(x) / \Lambda'(x)$

$e_1 = \Omega(1) / \Lambda'(1) = (1 + \alpha^4) / \alpha^2 = \underline{\alpha^3}$

$e_4 = \Omega(\alpha^3) / \Lambda'(\alpha^3) = (\alpha^3 + \alpha^4) / \alpha^2 = \underline{\alpha^4}$

Therefore the error polynomial is $\mathbf{e(x)} = \alpha^4\mathbf{x}^4 + \alpha^3$

Error correction

The corrected codeword is, therefore, $c(x) = v(x) + e(x)$

$(\alpha^4x^6 + \alpha^3x^3 + \alpha^6x + \alpha^4) + (\alpha^4x^4 + \alpha^3) = (\alpha^4x^6 + \alpha^4x^4 + \alpha^3x^3 + \alpha^6x + \alpha^6)$

Therefore, the corrected codeword is $c(x) = \alpha^4x^6 + \alpha^4x^4 + \alpha^3x^3 + \alpha^6x + \alpha^6$.

Vitae

- Mohamed Ahsan
- Born in June 1970 at Bijapur (Karnataka), India
- Received Bachelor of Engineering (**B.E.**) degree in Electronics and Communication Engineering from M. J. College of Engg. and Tech. (MJCET), Osmania University (**OU**), Hyderabad, India in June 1992
- Joined the Department of Electrical Engineering at King Fahd University of Petroleum and Minerals (**KFUPM**), Dhahran, Saudi Arabia as a Research/Teaching Assistant in December 1992
- Received Master of Science (**M.S.**) degree in Electrical Engineering from KFUPM, Saudi Arabia in 1995