Linköping Studies in Science and Technology. Dissertation
No. 242

VLSI Architectures for Computations in Galois Fields

Mastrovito Edoardo

Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden

ISBN 91-7870-737-4

Abstract

The present dissertation deals with the development of architectures for various forms of multiplication, squaring, square rooting, exponentiation and division in Galois fields GF(2^m). In particular, we are interested in architectures suitable for VLSI implementation.

Three different representations of GF(2^m) are considered with respect to the above purpose: the polynomial-basis (PB), the dual-basis (DB) and the normal-basis (NB) representation.

Our general conclusion is that the PB representation is by far the most versatile since it appears to offer suitable solutions to most computational problems.

To
Berit,
Alexander,
and
Elisabeth.

To
my parents.

Contents

Outline of the Thesis

This thesis starts off in Ch. 2 by introducing the necessary mathematical background and some of the notation to be used in later chapters.

In Ch. 3 we introduce, analyze and compare the most important PB, DB and NB architectures for bit-serial multiplication in GF(2^m). These will also serve as starting points in many later sections.

Ch. 4 deals with fast, bit-parallel multiplier architectures. We give a new, detailed treatment of an old algorithm for PB multiplication, and show that it is considerably better suited to VLSI implementation than previously believed. We present bounds on its complexity/performance, and derive simple criteria for selecting good field generators that yield low complexity and high performance. We identify and analyze thoroughly two classes of good field generators: the trinomials and the equally spaced polynomials. Further, we present the best field generators for $m \leq 16$ and show that these can be easily spotted by our selection criteria. Finally we compare with other PB and NB architectures.

In Ch. 5 we are concerned with multiplication of a variable by a constant field element. We investigate if and how the PB/DB/NB multipliers of Ch. 3-4 can be simplified, and introduce three new bit-serial PB architectures. Then we show how two of the new architectures can be used to obtained improved Reed-Solomon encoders that adopt the PB representation. Bit-parallel architectures are also investigated. The chapter ends with a comparison of the various architectures.

In Ch. 6 we investigate the properties of hybrid multipliers over GF(2^m). A hybrid multiplier is a multiplier that is designed by viewing GF(2^m) as an extension of one of its composite subfields. Hybrid multipliers fill the gap between the slow but simple bit-serial multipliers and the fast but complex fully parallel multipliers of Ch. 3-4.

In Ch. 7 we analyze squaring and square rooting in PB. We derive closed-form expressions for the functions defining the squaring operation and show that squares can often be computed by very simple, fast combinational circuits. Further, we investigate briefly the computation of square roots.

In Ch. 8 we discuss a great variety of architectures for computing α^e, some of which appears to be published for the first time. The architectures are classified according to the unknown input data and the time required by one operation.

Ch. 9 discusses and compares most known methods for computing the multiplicative inverse of an element of GF(2^m). A few, seemingly unpublished architectures are introduced.

The last topic of this thesis is treated in Ch. 10 where we consider universal architectures for multiplication, exponentiation and inversion over a range of different fields. Among other things, we introduce a novel parallel multiplier architecture which is twice as fast as, and significantly less complex than previously known architectures.

Our conclusions are found in Ch. 11.

Omissions. Some topics that belong to the field of computations in Galois fields have not been treated or even mentioned in this thesis. These are: log/antilog tables, Zech's logarithms (see e.g. [Hub89]), and the computation of the discrete logarithm (see e.g. [Odl84]).

Remark. Parts of this thesis have been published previously in [Eri86] [Mas88.1] [Mas88:2] [Mas89:1] [Mas90].

Chapter 1

Introduction

1.1 Who was Galois ?

In this thesis we are going to use the prefix *Galois* a great number of times. We feel therefore obliged to introduce the person behind this name.

Briefly stated, *Evariste Galois* (1811 - 1832) was a french mathematician who, building on the work of Lagrange, Gauss, Abel and Cauchy, greatly extended the understanding of the conditions in which an algebraic equation is solvable by radicals† and, by his methods, laid the foundations of modern group theory. He is also regarded as the father of finite fields to which he has posthumously provided the name *Galois fields*.

Galois has thus provided the mathematical basis of this dissertation.

Further details on Galois' life and work are given in Appendix E.

1.2 Applications of Galois Fields

Today Galois fields have found practical application in many important areas. Galois fields are applied to error-control codes (ECC) [Bla84:1] [Mac86], cryptography [Til88], switching theory [Ben76] and digital signal processing [Ree75] [McC79].

---

† This means that we can find its solutions by a finite number of operations performed on the coefficients of the algebraic equations. These are rational operations (i.e. addition, subtraction, multiplication, division) and extractions of roots. If a solution gained by only these operations exists, we say that the equation is *solvable by radicals*.

One of the reasons why Galois-field computations are attractive is that they display some particularly nice properties which are absent in other number systems. These are:

- all operations are carryless,
- the word length is constant,
- there is no round-off problem.

We will look closer at two areas of application.

Error-Control Codes. ECC's are often used to increase the reliability of noisy transmission channels. The transmission channel can take very different forms. It could be a telephone wire, a radio path, the fabrication process of a Compact Disc [Pee85] or of a large digital memory, the down link of a satellite [Wu87] or the deep-space environment in which spacecrafts have to function [Pos90]. In all these plus several other situations, the need arises for ECC's.

An important class of ECC's are the *cyclic codes*. Cyclic codes are normally defined in terms of a *generator polynomial* having certain prescribed zeros.

The most important cyclic codes, from a practical point of view, are the Bose-Chaudhury-Hocquenghem (BCH) codes and the related Reed-Solomon (RS) codes. RS codes are non-binary codes that have the minimum possible number of parity-check symbols, namely $2t$ symbols for a t-error correcting code.

A number of efficient algorithms are available for encoding/decoding of cyclic codes. All these algorithms require a considerable amount of operations in a Galois field.

The key idea that led to the development of the best decoding algorithms was to introduce an *error-locator polynomial* whose roots are the inverse error locations. The decoding problem can thus be split in two steps: first we find the error-locator polynomial and the error positions, and then we compute the error values. The error-locator polynomial can be computed by either the *Berlekamp-Massey algorithm* or by the *Euclidean algorithm* starting from the so-called *syndromes* which are actually the known components of the error spectrum — i.e. the error's discrete Fourier transform (DFT). The error-values can be found either by spectral

techniques whereby one determines the complete error spectrum and computes its inverse DFT, or by using the *Forney algorithm*.

Fig. 1.1 shows how the system structure of an error-erasure decoder for an RS code could look like. All boxes in the figure, except the buffer registers and the multiplexer, perform operations in a Galois field, typically in GF(2^m) with $4 \leq m \leq 8$. The most used operations are addition and multiplication, but division might also be required.

Several other decoder structures are possible but common to all is an extensive use of Galois-field computations.

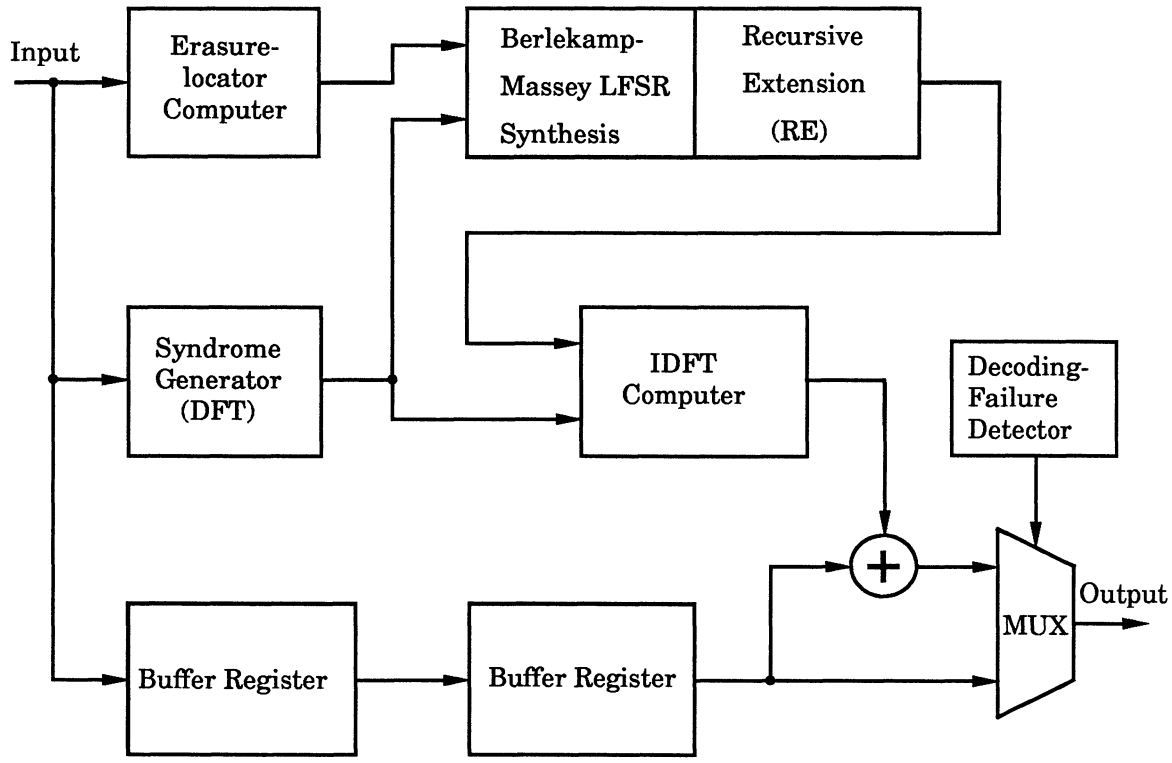


Figure 1.1 System structure of an error-erasure decoder for Reed-Solomon codes. IDFT stands for inverse discrete Fourier transform. All boxes, except buffers and multiplexer, perform operations in a Galois field, typically in GF(2^m) with m = 4, 5, 6, 7, or 8.

Other examples of classes of algebraic ECC's are: Goppa codes, Alternant codes, Quadratic Residue (QR) codes and, since a few years, algebraic-geometric (AG) codes. All these codes rely heavily on the structure of Galois fields.

Cryptography. Galois fields became highly interesting to cryptography with the introduction of public-key cryptosystems [Dif76]. The security of several such systems for authentication or key-exchange depends on the *difficulty* of inverting the exponentiation function — that is, of computing the discrete logarithm over a Galois field GF(q). An excellent overview of this subject is given in [Odl84].

Galois fields can also be used to generate pseudo-random (PN) sequences either by special shift-register configurations [Sme87] or through exponentiation [Wan90]. PN sequences are used both in cryptography and in spread-spectrum communication.

1.3 VLSI

The rapid advancements in the manufacturing techniques for integrated circuits (IC) have been the single, most important factor that has made many Galois-fields based applications technically feasible and economically viable.

In particular, Very Large Scale Integration (VLSI) allows the designers of today to allocate complex systems consisting of several hundreds thousands, or even millions transistors on one or very few chips.

The design of such complex systems could never be undertaken without extensive use of computer-aided design tools and the use of efficient design methodologies. Today, computer-aided design (CAD) and computer-aided manufacturing (CAM) are widely used in almost all aspects of electronic engineering.

The main aspects of VLSI system design are discussed in Appendix F [Wes85] [Pre88] [Wah90].

1.4 Complexity/Performance Measures in the Thesis

In Ch. 3-10 we will discuss and compare a number of architectures for computations in Galois fields. In particular, we will consider their complexity and performance. In doing so we will assume that the technology intended is CMOS (see App. F).

Complexity will be characterized both through the number of logical devices (e.g. gates and registers) required and the structural properties (e.g. modularity and regularity) of the architecture. Occasionally we will also considered routing and floor-planning related properties.

Having said the above we must warn for our somewhat ambiguous use of the word complexity. Later, we will define the number of logic components C and call it the complexity. We could have (and maybe should have) called C the component count. Clearly, C alone does not represent the *real* complexity as characterized above.

The architectures to be discussed in this thesis will consist almost exclusively of registers (or D flip-flops), AND gates and XOR gates. AND gates are normally realized as inverted NAND gates which require 6 transistors. XOR gates can be realized in different ways using between 6 and 12 transistors [Wes85]. Registers can also be realized in different ways using between 4 and 24 transistors [Wes85]. Pure delay elements (dynamic in particular) are much simpler than static set/reset D flip-flops. For sake of simplicity we have here chosen to consider registers, AND gates and XOR gates as devices of comparable complexity.

The *performance* of a logic system is determined by several factors: critical-path (CP) lengths, driving capability of heavily loaded signals, capacitances, wire lengths, signal skews, clock skews, working temperature and voltage etc.

In this thesis we will often consider the length of the critical path(s) as the major performance parameter. With the above list in mind, this appears as an obvious simplification. The CP length does not always say the whole truth about the system performance.

Finally we point out that many of the architectures treated in this thesis are to be seen as constituent parts in larger systems such as advanced, high-speed algebraic Reed-Solomon decoders. Sometimes, however, like with the large-field exponentiators of Ch. 8, we will deal with almost complete, VLSI-demanding systems.

Chapter 2

Mathematical Background

In this chapter we give a brief overview of the mathematical background of this thesis. Most of the statements are given without proof. The reader in need of proofs or further details is referred to the following textbooks: [Lid83] [Bla84:1, Ch. 2 & 4] [Ber68, Ch. 3] [Gol67] or [Gil67].

We shall begin our discussion with the fundamental concept of a group followed by the ring and the finite fields of prime order. We will then introduce the ring of polynomials over a finite field and the finite extensions of finite fields. Finally, we discuss briefly the problem of finding prime polynomials.

2.1 Groups, Rings and Fields

A group is a mathematical abstraction of an algebraic structure. In this thesis we are mainly concerned with commutative (or abelian) groups.

Definition 2.1. An *abelian group* G is a set of elements together with a binary operation $*$ satisfying the following properties:

1) *Closure*. For every a, b in the set, $c = a * b$ is in the set.

2) *Associativity*. For every a, b, c in the set,

$$a * (b * c) = (a * b) * c.$$

3) *Identity*. There is an *identity* (or *unity*) element $e \in G$ such that for all a in G,

$$a * e = e * a = a.$$

4) *Inverses*. For each $a \in G$ there exists an *inverse element* b also in G such that

$$a * b = b * a = e.$$

5) *Commutativity*. For all $a, b \in G$,

$$a * b = b * a. \square$$

It is easily verified that the identity element e and the inverse a^{-1} of a given element $a \in G$ are uniquely determined by the properties of Def. 2.1.

From now on we will drop the operator $*$ and use either $+$ (i.e. $a + b$) or juxtaposition (i.e. ab). The former notation *resembles* ordinary arithmetic addition whereas the latter *resembles* ordinary arithmetic multiplication. The word "resembles" is emphasized because we do *not* assume that the operation actually is ordinary multiplication/addition. In fact, in the rest of this thesis ordinary multiplication/addition is scarcely used.

When using the additive notation the identity element e is called "zero" and written 0, and the inverse element of a is written $-a$, so that $a + (-a) = 0$. For multiplicative notation the identity element is called "one" and written 1, and the inverse element of a is written a^{-1}, so that $aa^{-1} = 1$.

According to customary notation, we have the following rules for $a \in G$ and $n,m \in \mathbb{N}$:

Multiplicative Notation	*Additive Notation*
$a^n = aa\cdots a$ (n factors a)	$na = a + a + \cdots + a$ (n summands a)
$a^{-n} = (a^{-1})^n$	$(-n)a = n(-a)$
$a^n a^m = a^{n+m}$	$na + ma = (n + m)a$
$(a^m)^n = a^{mn}$	$n(ma) = (nm)a$
$a^0 = 1$	$0a = 0$

The "zeros" in the last row are different "zeros". The exponent is the ordinary 0 in $\mathbb{N}$ whereas the last zero is the identity element of G.

A group is called *finite* (resp. *infinite*) if it contains finitely (resp. infinitely) many elements. The number of elements in a finite group is called the *order* of the group. The integers under ordinary addition form an infinite abelian group. An example of a finite abelian group is the set {0, 1} under the exclusive–or (XOR) operation. We are mainly interested in finite groups.

A multiplicative group is said to be *cyclic* if there is an element $a \in G$ such that for any $b \in G$ there is some integer j with $b = a^j$. Such an element is called a *generator* of the cyclic group. Every cyclic group is commutative and has at least one generator (normally more). The *order* of an element a in a finite group G is the least positive integer c such that $a^c = 1$ and is denoted by ord a.

Theorem 2.2. The order of a finite group is divisible by the order of any of its elements. □

A subset H of G is a *subgroup* of G if H is itself a group with respect to the operation of G For any $a \in G$ the set of all powers of a is a subgroup of G called the subgroup *generated* by a. The order of any subgroup of the group G must divide the order of G.

Let n be a positive integer and $a, b \in \mathbb{Z}$. We say that a is *congruent* to b modulo n, and write $a \equiv b \bmod n$, if the difference $a-b$ is a multiple of n — that is, $a = b + kn$ for some integer k. The relation "congruence modulo n" is a genuine equivalence relation exhibiting reflexivity, symmetry and transitivity.

An interesting group is that formed by the set $\{\langle 0\rangle, \langle 1\rangle, \ldots, \langle n-1\rangle\}$ of equivalence classes modulo n with the operation

$$\langle a\rangle + \langle b\rangle = \langle a + b\rangle.$$

It is called the group of integers modulo n and denoted by $\mathbb{Z}_n$. $\mathbb{Z}_n$ is a group of order n with the equivalence class $\langle 1\rangle$ as a generator.

The next algebraic structure to be introduced is that of a ring. A ring is an abelian group with an additional structure.

Definition 2.3. A *ring* R is a set of elements together with two binary operations denoted by + (called *addition*) and juxtaposition (called *multiplication*) satisfying the following properties:

1) R is an abelian group under addition (+).
2) *Closure*. For every a, b in R, the product ab is in R.
3) *Associativity*. For every a, b, c in R,
$a(bc) = (ab)c$.
4) *Distributivity*. For all $a, b, c \in R$,
$a(b + c) = ab + ac$,
$(b + c)a = ba + ca$. □

A ring is called a *commutative ring* if multiplication is commutative. A ring is called a *ring with identity* if the ring has a multiplicative identity — that is, there is an element 1 such that $a1 = 1a = a$ for all a in R.

A ring is called an *integral domain* if it is a commutative ring with identity $e \neq 0$ in which $ab = 0$ implies $a = 0$ or $b = 0$.

A ring is called a *division ring* if the nonzero elements of R form a group under multiplication. A commutative division ring is called a *field*.

Having now reached the concept of a field we emphasize its definition.

Definition 2.4. A *field* is a set F together with two operations, addition (+) and multiplication (juxtaposition), satisfying the following properties:

1. The set F is an abelian group under addition with 0 as the identity element
2. The nonzero elements of F form an abelian group under multiplication with 1 as the identity element.
3. The distributive law
   $a(b + c) = ab + ac$
   holds for all a, b, c in the field. □

The set $\mathbb{R}$ of real numbers and the set $\mathbb{Q}$ of rational numbers are well–known examples of infinite fields. A few more concepts are needed in order to arrive at an example of a finite field.

A subset S of a ring R is called a *subring* of R if S is closed under addition and multiplication, and forms a ring under these operations.

A subset J of a commutative ring R is called an *ideal* if J is a subring of R and for all $a \in J$ and $r \in R$ we have $ar \in J$ and $ra \in J$ The ideal J is called a *principal* ideal if there is an element $a \in R$ such that $J = (a)$, where (a) denotes the set of all multiples of a – that is, $(a) = \{ra: r \in R\}$. We say that J is the ideal *generated by* a.

An ideal J of a ring R defines a partition of the ring into disjoint cosets, called *residue classes* modulo J The residue class of an element $a \in R$ modulo J is denoted $\langle a \rangle = a + J$ since it consists of all elements of R of the form $a + c$ for some $c \in J$. Two elements $a, b \in R$ from the same residue class modulo J are said to be congruent modulo J and is written $a \equiv b \bmod J$.

The set of residue classes of a ring R modulo an ideal J forms a ring with respect to the following operations

$$(a + J) + (b + J) = (a + b) + J$$
$$(a + J)(b + J) = ab + J$$

and it is the *residue class ring* of R modulo J, denoted R/J.

As in the case of groups we denote the residue class of the integer a modulo the positive integer n by $\langle a \rangle$, as well as by $a + (n)$, where (n) denotes the principal ideal generated by n. For example, the elements of the residue class ring $\mathbb{Z}/(n)$ are

$$\langle 0\rangle = 0 + (n),\ \langle 1\rangle = 1 + (n),\ \langle 2\rangle = 2 + (n),\ \ldots,\ \langle n-1\rangle = n - 1 + (n).$$

We are now ready to introduce the first example of a finite field:

Theorem 2.5. $Z/(p)$, the ring of residue classes of the integers modulo the principal ideal generated by a *prime* p, is a field. □

A more convenient representation for the finite fields of Th. 2.5 is obtained by introducing an isomorphism (i.e. a ono-to-one mapping that preserves operations) as indicated in the following definition.

Definition 2.6. For a prime p, let F_p be the integer set $\{0, 1, 2, \ldots, p-1\}$ and let φ be the isomorphic mapping from $Z/(p)$ to F_p defined by $\varphi(\langle a\rangle) = a$ for $a = 0, 1, 2, \ldots, p-1$. Then F_p, endowed with the field structure induced by φ, is a finite field called a *Galois field of order* p. □

It is common to denote a Galois field of order p by GF(p).

The finite field F_p has zero element 0, identity 1 and the structure of $Z/(p)$. Computations in F_p are therefore performed as in ordinary integer arithmetic with reduction modulo p.

Let R be any ring. If there exists a positive integer n such that $nr = 0$ for all $r \in R$ then the least such positive integer is called the *characteristic* of R and R is said to have characteristic n. If no such positive integer exists, R is said to have characteristic 0.

Theorem 2.7. A finite field has prime characteristic. □

The binary field F_2 (or GF(2)) is a familiar example of a field of characteristic 2. Addition and multiplication are both performed modulo 2 and are widely known as XOR and logical AND respectively.

Definition 2.8. Suppose F is a field and K is a subset of F such that K under the operations of F is a field too. Then K is called a *subfield* of F and F is called an *extension field* of K. A subfield $K \neq F$ is called a *proper subfield* of F. A field containing no proper subfields is called a *prime field.* □

The characteristic of a field F can also be seen as the number of elements in the smallest subfield of F. It is easily seen that the field F_p with prime p is

a prime field. Also, all finite extensions of the prime field $\mathbb{F}_p$ have the same characteristic p.

2.2 Polynomials over Finite Fields

Let F be an arbitrary finite field. A *polynomial* over F is a mathematical expression of the form

$$f(x) = \sum_{i=0}^{n} f_i x^i = f_0 + f_1 x + f_2 x^2 + \cdots + f_{n-1} x^{n-1} + f_n x^n$$

where n is a nonnegative integer, the *coefficients* f_i are in F, and x is an *indeterminate* over F. We call f_n the *leading coefficient* of $f(x)$ and f_0 the *constant term*, while n is called the degree of $f(x)$ and is denoted deg $f(x)$. It is convention to set deg $0 = -\infty$. Polynomials of degree 0 are called *constant polynomials*. A *monic polynomial* is a polynomial with $f_n = 1$.

The *reciprocal polynomial* of $f(x)$ is denoted by $f^*(x)$ and is defined by

$$f^*(x) = x^n f(\frac{1}{x}) = f_0 x^n + f_1 x^{n-1} + f_2 x^{n-2} + \cdots + f_{n-1} x + f_n.$$

Two polynomials $f(x)$ and $g(x) = \sum_{i=0}^{m} g_i x^i$ are said to be equal if $f_i = g_i$ for all i. Addition and multiplication are defined as the ordinary addition and multiplication of polynomials. The *sum* of $f(x)$ and $g(x)$ is thus

$$f(x) + g(x) = \sum_{i=0}^{\infty} (f_i + g_i) x^i$$

where terms with index larger than max [deg $f(x)$, deg $g(x)$] are, of course, all zero – that is, the degree of the sum is $\leq$ max [deg $f(x)$, deg $g(x)$]. The *product* of $f(x)$ and $g(x)$ is

$$f(x)g(x) = \sum_{i} \Big(\sum_{j=0}^{i} (f_i g_{i-j}) \Big) x^i .$$

and its degree is equal to the sum of the degrees of $f(x)$ and $g(x)$.

It is easily seen that with these operations the set of polynomials over F forms a ring.

Definition 2.9. The ring formed by the polynomials over F with the above operations is called the *polynomial ring* over F and is denoted by $F[x]$. □

The zero element of $F[x]$ is the *zero polynomial* $f(x) = 0$.

A polynomial ring is analogous in many ways to the ring of integers. We say that the polynomial $g(x) \in F[x]$ *divides* the polynomial $f(x) \in F[x]$ if there exists a polynomial $h(x) \in F[x]$ such that $f(x) = g(x)h(x)$. A polynomial $p(x)$ that is divisible only by $\alpha p(x)$ or α, where $\alpha \in F$, is called an *irreducible polynomial over* F. A monic irreducible polynomial of degree at least 1 is called a *prime polynomial*. A polynomial in $F[x]$ that is not irreducible over F is called *reducible over* F. The reducibility/irreducibility of a polynomial depends heavily on the field under consideration.

The *greatest common divisor* of two polynomials $f(x)$ and $g(x)$, denoted by $gcd[f(x), g(x)]$, is the monic polynomial of largest degree that divides both of them. The least common multiple of two polynomials $f(x)$ and $g(x)$, denoted by $lcm[f(x), g(x)]$, is the monic polynomial of smallest degree divisible by both of them. The greatest common divisor and the least common multiple of two polynomials in $F[x]$ are unique. If $gcd[f(x), g(x)] = 1$ then $f(x)$ and $g(x)$ are said to be *relatively prime*.

Theorem 2.10 (Division Algorithm for polynomials). Let $g(x) \neq 0$ be in $F[x]$. Then for every $f(x) \in F[x]$ there exist polynomials $q(x), r(x) \in F[x]$ such that

$$f(x) = q(x)g(x) + r(x), \text{ where } \deg r(x) < \deg g(x). \square$$

We call $q(x)$ the *quotient polynomial* and $r(x)$ the *remainder polynomial*, or simply the quotient resp. the remainder when no confusion arises.

Irreducible polynomials are of fundamental importance for the structure of the ring $F[x]$ since the polynomials in $F[x]$ can be written as products of irreducible polynomials in an unique manner as indicated below.

Theorem 2.11 (Unique Factorization of Polynomials). Any polynomial $f(x) \in F[x]$ of positive degree can be written in the form

$$f(x) = a(p_1(x))^{e_1}(p_2(x))^{e_2} \cdots (p_k(x))^{e_k}$$

where $a \in F$, $p_1(x), p_2(x), \cdots, p_k(x)$ are distinct prime polynomials in $F[x]$, and $e_1, e_2, \cdots, e_k$ are positive integers. □

The division algorithm for polynomials has an important consequence known as the Euclidean algorithm.

Theorem 2.12 (Euclidean Algorithm for Polynomials). Given two polynomials $f(x)$ and $g(x)$ in $F[x]$, their greatest common divisor can be computed by an iterative application of the division algorithm. If deg $f(x) \geq$ deg $g(x) \geq 0$, this computation is

$$\begin{aligned} f(x) &= q_1(x)g(x) + r_1(x) \\ g(x) &= q_2(x)r_1(x) + r_2(x) \\ r_1(x) &= q_3(x)r_2(x) + r_3(x) \\ &\vdots \\ r_{n-1}(x) &= q_{n+1}(x)r_n(x) \end{aligned}$$

where the process stops when a remainder of zero is obtained. Then

$$gcd[f(x), g(x)] = b^{-1}r_n,$$

where $b \in F$ is the leading coefficient of the last nonzero remainder r_n. □

An element $\alpha \in F$ is called a *root* (or a *zero*) of the polynomial $f(x)$ if and only if $f(\alpha) = 0$. By the division algorithm it follows that an element $\alpha \in F$ is a *root* of the polynomial $f(x)$ if and only if $x-\alpha$ divides $f(x)$.

The *formal derivative* of a polynomial $f(x)$ is defined as the polynomial in $F[x]$ given by

$$f'(x) = f_1 + 2f_2x + 3f_3x^2 + \cdots + nf_nx^{n-1} = \sum_{i=0}^{n} if_ix^{i-1}$$

where the factor i is interpreted as a sum over F of i ones (i.e. $1 + 1 + \cdots + 1$, i times). For example, for $f(x)$ in $F_2[x]$ the formal derivative becomes $f_1 + f_3x^2 + f_5x^4 + \cdots + f_nx^{n-1}$ for odd n, since here $2i = 0$ and $2i - 1 = 1$ for all i.

The element $\alpha \in F$ is a multiple root of $f(x) \in F[x]$ if and only if it is a root of both $f(x)$ and $f'(x)$.

2.3 Extension Fields

Previously we defined the concept of an extension field. We will see now how finite extension fields can be obtained from polynomial rings by using

constructions similar to those used to obtain finite fields from the integer ring. In particular, since the irreducible polynomials over a finite field F are the prime elements of $F[x]$, the following result appears fairly natural.

Theorem 2.13. Let $f(x) \in F[x]$. The residue class ring $F[x]/(f(x))$ is a field if and only if $f(x)$ is irreducible over F. □

The ring $F[x]/(f(x))$ consists of residue classes $g(x) + (f(x)) = \langle g(x) \rangle$. Two residue classes $\langle g(x) \rangle$ and $\langle h(x) \rangle$ are identical if $g(x) \equiv h(x) \bmod f(x)$ – that is, if $g(x)$ and $h(x)$ leave the same remainder after division by $f(x)$. Each residue class $\langle g(x) \rangle$ contains a unique representative $r(x)$ of degree less than deg $f(x)$, which is simply the remainder in the division of $g(x)$ by $f(x)$. This operation of going from $g(x)$ to $r(x)$ is called *reduction mod* $f(x)$. The residue classes in $F[x]/(f(x))$ can now be described explicitly: they are the residue classes $\langle r(x) \rangle$ where $r(x)$ runs through all polynomials in $F[x]$ of degree less than deg $f(x)$.

For example, if $F = \mathcal{F}_p$, p prime, and deg $f(x) = n \geq 0$, then the number of elements in $F[x]/(f(x))$ is exactly the number of polynomials in $\mathcal{F}_p[x]$ of degree less than n, which is p^n. Whenever we can choose an irreducible $f(x)$ of degree n, this construction yields a finite field with p^n elements, denoted $\mathcal{F}_{p^n}$ or GF(p^n). Further, in analogy with the case of prime fields $\mathcal{F}_p$, we can identify the elements of $F[x]/(f(x))$ with the polynomials over $\mathcal{F}_p$ of degree less than n. For example, the field GF(2^2) is constructed from GF(2) using the prime polynomial $p(x) = x^2 + x + 1$. The field elements are represented by the set of polynomials $\{0, 1, x, x + 1\}$. Operations on the field elements are performed modulo $p(x)$. A finite field constructed in the above way is a finite extension *of degree n* of GF(p). The extensions of GF(p) are also Galois fields. In general we have the following important result.

Theorem 2.14. Every Galois field has p^m elements for some positive integer m and prime p. □

The above construction can be extended to obtain extensions of a field GF(q) with $q = p^m$, by using an irreducible polynomial in the ring $\mathcal{F}_q[x]$. By the above theorem we are therefore able (at least in principle) to construct any Galois field. Whenever we construct a field with p^m elements we will speak of *the* (Galois) field with p^m elements since it can be shown that all fields with the same number of elements are isomorphic.

In the sequel $p(x)$ denotes the prime polynomial used to construct a finite extension of a field and will occasionally be called the *field generator*, p denotes a prime number and q a positive power of a prime number.

The field GF(q) contains the important multiplicative group GF(q)* of non-zero elements of GF(q). The group GF(q)* can be shown to be cyclic – that is, it contains generators (at least one) which are called *primitive elements*. Clearly, a primitive element of GF(q) has order $q-1$. A *primitive polynomial* is a prime polynomial having a primitive element as a zero.

The unique monic polynomial $f(x)$ over GF(q) of lowest degree such that $f(\alpha) = 0$ is called the *minimal polynomial* of α over GF(q) and is denoted $f_\alpha(x)$.

Theorem 2.15. The minimal polynomial of $\alpha \in$ GF(q^m) over GF(q) is

$$f_\alpha(x) = \prod_{j=0}^{r-1} (x - \alpha^{q^j})$$

where $r \mid m$ and is the smallest integer such that $\alpha^{q^r} = \alpha$. □

The zeros of $f_\alpha(x)$ in the above theorem are all in GF(q^m) and are called the *conjugates* of α with respect to GF(q). It is easy to see that $f_\alpha(x)$ is prime over GF(q).

Theorem 2.16. The conjugates of $\alpha \in$ GF(q^m)* with respect to any subfield of GF(q^m) have the same order in the multiplicative group GF(q^m)*. □

By the above theorem it follows that if α is a primitive element of GF(q^m) then so are all its conjugates with respect to any subfield of GF(q^m).

Theorem 2.17 Let GF(q) have characteristic p. Then for any positive integer n and for any elements α and β in GF(q),

$$(\alpha + \beta)^{p^n} = \alpha^{p^n} + \beta^{p^n}. \square$$

The square of an element of GF(q^m) is clearly also an element of GF(q^m). It is less obvious wether each field element has a square root in the same field.

Theorem 2.18 Every element of GF(2^m) has a square root in GF(2^m). Half of the nonzero elements of GF(p^m) — $p > 2$ — have a square root in GF(p^m).

Half of the nonzero elements of GF(p^m) have a square root in GF(p^{2m}) but not in GF(p^m). □

We conclude this section by introducing a linear mapping from GF(q^m) to GF(q) which involves the sum of the conjugates of a field element.

Definition 2.19 Let $\alpha \in F$ = GF(q^m) and K = GF(q). The *trace* $\mathrm{Tr}_{F/K}(\alpha)$ of α over K is defined by

$$\mathrm{Tr}_{F/K}(\alpha) = \alpha + \alpha^q + \cdots + \alpha^{q^{m-1}}. \square$$

Later when there will be no confusion about the fields F and K we will write simply $\mathrm{Tr}(\alpha)$.

Theorem 2.20 Let F = GF(q^m) and K = GF(q). Then the trace function satisfies the following properties:

1. $\mathrm{Tr}_{F/K}(\alpha+\beta) = \mathrm{Tr}_{F/K}(\alpha) + \mathrm{Tr}_{F/K}(\beta)$ for all $\alpha, \beta \in F$;
2. $\mathrm{Tr}_{F/K}(c\alpha) = c\mathrm{Tr}_{F/K}(\alpha)$ for all $c \in K$, $\alpha \in F$;
3. $\mathrm{Tr}_{F/K}(a) = ma$ for all $a \in K$;
4. $\mathrm{Tr}_{F/K}(\alpha^q) = \mathrm{Tr}_{F/K}(\alpha)$ for all $\alpha \in F$;
5. $\mathrm{Tr}_{F/K}$ is a linear transformation from F onto K, where both F and K are viewed as vector spaces over K. □

2.4 Bases of Finite Fields

A finite extension GF(q^m) of GF(q) can be viewed as a vector space of dimension m over GF(q). If $\{\beta_1, \beta_2, \ldots, \beta_m\}$ is a basis of GF(q^m) over GF(q), each element $\alpha \in$ GF(q^m) can be uniquely represented in the form

$$\alpha = a_1\beta_1 + a_2\beta_2 + \cdots + a_m\beta_m$$

with $a_i \in$ GF(q) for $1 \le i \le m$. In general there are many distinct bases of GF(q^m) over GF(q) but there are three types of basis which are of particular interest.

The first type is the *polynomial* (or *standard* or *canonical*) *basis* (PB) given by the set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$, where α is a root of the prime polynomial $p(x)$ of degree m used to construct GF(q^m) from GF(q). This basis has a direct relation to the polynomial representation of the previous section. In

particular, in this basis the elements α^i, $i = 0, 1, \ldots, m-1$ are represented by the polynomials x^i, $i = 0, 1, \ldots, m-1$, and the expression $a_0 + a_1 x + \ldots + a_{m-2}x^{m-2} + a_{m-1}x^{m-1}$ is equivalent to $a_0 + a_1\alpha + \ldots + a_{m-2}\alpha^{m-2} + a_{m-1}\alpha^{m-1}$. The above fact should explain the name *polynomial basis*.

The second type is the *normal basis* (NB) given by the set $\{\alpha, \alpha^q, \ldots, \alpha^{q^{m-1}}\}$, that is, the set of conjugates of a suitable element α of GF(q^m). If α is a root of the field generator $p(x)$, the powers $\alpha, \alpha^q, \ldots, \alpha^{q^{m-1}}$ are in GF(q^m) and constitute a complete set of roots of $p(x)$. Hence, if we want the element α to generate a normal basis we must choose a prime polynomial $p(x)$ with linearly independent roots.

Theorem 2.21 For any finite field K and any finite extension F of K, there exists a normal basis of F over K. □

Theorem 2.22 For any finite field F, there exists a normal basis of F over its prime subfield that consists of primitive elements of F. □

The third type of basis is the *dual* (or *complementary*) *basis* (DB). Two bases $\{\beta_1, \beta_2, \ldots, \beta_m\}$ and $\{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ of GF(q^m) over GF(q) are said to be dual bases if, for $1 \le i, j \le m$, we have

$$\mathrm{Tr}(\beta_i \gamma_j) = \begin{cases} 0 & , i \neq j \\ 1 & , i = j \end{cases}$$

It can be shown that for any basis $\{\beta_1, \beta_2, \ldots, \beta_m\}$ there exists a dual basis $\{\gamma_1, \gamma_2, \ldots, \gamma_m\}$.

2.5 Prime Polynomials

As we saw previously, prime polynomials over a finite field are indispensable for constructing extension fields. A natural question is therefore: do prime polynomials always exist ? The answer is positive: for every finite field GF(q) and positive integer m, there exists at least one prime polynomial over GF(q) of degree m. In fact, for every finite field GF(q) and positive integer m, there exists at least one primitive polynomial over GF(q) of degree m.

We introduce another useful concept connected with polynomials over finite fields.

Definition 2.23 Let $f(x) \in \mathcal{F}_q[x]$ be a nonzero polynomial with $f(0) \neq 0$. The least positive integer e for which $f(x)$ divides $x^e - 1$ is called the *order* of $f(x)$ and denoted by ord $f(x)$. □

The order of a polynomial $f(x)$ is also called the *period* or the *exponent* of $f(x)$. The order of a prime polynomial is directly related to the order of its roots as indicated below.

Theorem 2.24 Let $f(x) \in \mathcal{F}_q[x]$ be a prime polynomial over $\mathcal{F}_q$ of degree m with $f(0) \neq 0$. Then ord $f(x)$ is equal to the order of any root of $f(x)$ in the multiplicative group $\mathcal{F}_{q^m}^*$. □

By the above theorem it follows that ord $f(x)$ must divide $q^m - 1$. In particular, a primitive polynomial over $\mathcal{F}_q$ of degree m can be characterized as a prime polynomial of order $q^m - 1$.

Theorem 2.25 Let $f(x)$ be a nonzero polynomial in $\mathcal{F}_q[x]$ and $f^*(x)$ its reciprocal polynomial. Then ord $f(x)$ = ord $f^*(x)$. □

The number $N_q(m)$ of prime polynomials of degree m over $GF(q)$ can be computed explicitly by the well-known formula

$$N_q(m) = \frac{1}{m} \sum_{d \mid m} \mu(d) q^{m/d}$$

where $\mu(d)$ is the Möbius function. Table 2.1 shows the first twenty values of $N_q(m)$ for $q = 2$. We see, from the table, that $N_2(m+1) \approx 2N_2(m)$ for $m > 1$.

In later chapters we will be interested in prime polynomials of degree at least 2 with few nonzero coefficients — that is, prime polynomials of low *hamming weight*. In the case of binary polynomials we can easily see that their hamming weight must be odd and at least 3 (a binary polynomial of even weight has always 1 as a root). Nonbinary prime polynomials may have even weight. Further, the least significant coefficient of any prime polynomial must always be nonzero (otherwise the polynomial has 0 as a root). We call polynomials of hamming weight 2, 3, 4 and 5 for *binomials*, *trinomials*, *quadrinomials* and *pentanomials* respectively.

We denote a general trinomial over GF(q) by $\beta + \alpha x^k + x^m$, $\alpha, \beta \in$ GF(q). To our knowledge, the only known infinite class of irreducible trinomials is the following.

m	$N_2(m)$
1	2
2	1
3	2
4	3
5	6
6	9
7	18
8	30
9	56
10	99
11	186
12	335
13	630
14	1161
15	2182
16	4080
17	7710
18	14532
19	27594
20	52377

Table 2.1 The number of binary prime polynomials of degree $m \leq 20$.

Theorem 2.26 Let $m=2\cdot 3^l$, $l \in Z_+$. Then

$$f(x) = 1 + x^{m/2} + x^m$$

is irreducible over GF(2).

Proof. See [Lit82, pp.12, Th.1.1.28]. □

Another result of interest concerns so-called all-ones polynomials.

Theorem 2.27 Let e and p be two prime numbers. Then the polynomial

$$f(x) = 1 + x + \cdots + x^{e-2} + x^{e-1}$$

is irreducible over every field GF(p), where p is a generator of the group of nonzero elements of GF(e).

Proof. See [Dic58, pp.21, Th.33]. □

An interesting result, although negative, is mentioned in [Gol67, pp.96] and states that no binary irreducible trinomials of degree $8n$ exist for any value of n. More results on both binary and nonbinary irreducible polynomials can be found, for example, in [Gol67] [Lid83].

A number of lists of irreducible polynomials are available. A list containing one example of a primitive polynomial over GF(2) for each degree ≤ 100 is found in [Wat62]. A complete list of irreducible trinomials x^m+x^k+1 over $GF(2)$ for $m \leq 1000$ can be found in [Zie68]. A list of all binary irreducible trinomials of degree p, $p \leq 11213$, for which $2^p - 1$ is known to be a Mersenne prime is given in [Zie69]. In [Pet72] there is a complete list of binary irreducible polynomials of degree ≤ 16 and an incomplete list for degrees in the range [17, 34]. Information about the order and the roots of each polynomial is also given. Lists of irreducible polynomials over certain prime fields are found in [Gil67] [Lid83] [Sug79]. Lists of irreducible polynomials over certain composite fields (i.e. extension fields) are given in [Gre74]. Further references to lists of irreducible polynomials can be found in [Lid83, pp. 131 - 140].

To our knowledge, no infinite (constructive) classes of primitive polynomials have been found yet. In fact, it is not even known if such classes exist.

Chapter 3

Bit-Serial Multiplication

In this chapter we will be concerned with architectures for bit-serial (or simply serial or sequential) computation of products in a field extension $GF(2^m)$. By a serial multiplier we intend a device that performs multiplication of two *arbitrary* elements of $GF(2^m)$ in a number of steps which is linear in m — the degree of the field extension. Normally m steps will be enough but for certain architectures $2m$ steps might be required in certain applications.

We will start by describing the classical approach for polynomial-basis (PB) multipliers and a variation of it which has been unjustly neglected in the literature (and apparently also in applications). Then we present known architectures for dual-basis (DB) and normal-basis (NB) multipliers. Finally we compare the properties of all these architectures.

The chapter serves as a starting point for later chapters and contains many known results but also some new observations. At the same time, it gives a good overview of the most important algorithms for bit-serial multiplication in $GF(2^m)$ and their implementations.

Throughout this thesis we will make use of several different representations for the elements of a finite field $GF(2^m)$. We might represent the (non-zero) field elements by powers of a primitive element $\alpha \in GF(2^m)$ — i.e. $\alpha^j, j = 0, 1, \ldots, 2^m-2$. We might also represent an element as a linear combination of basis elements or by a polynomial of degree at most $m-1$ — i.e. $\beta = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$, $b_i \in GF(2)$, $\beta \in GF(2^m)$ — see also Sec. 2.4. Finally, we might use vector notation whereby only the binary co-ordinates of the actual representation are written out — i.e. $\beta = (b_0, b_1, \ldots, b_{m-1})$. In each situation we will adopt the most appropriate of the above representations .

3.1 Serial Polynomial-Basis Multipliers

The SSR multiplier. The oldest and most common solution to the problem of designing a serial multiplier ought to be the standard shift-register (SSR) multiplier introduced by Peterson in [Pet61, Ch.7] and accidentally rediscovered in [Sco86]. Let $A(x) = a_0 + a_1x + \cdots + a_{m-1}x^{m-1}$ and $B(x) = b_0 + b_1x + \cdots + b_{m-1}x^{m-1}$ be two field elements and $C(x) = c_0 + c_1x + \cdots + c_{m-1}x^{m-1}$ their product modulo the prime (and in practice primitive) polynomial $P(x) = p_0 + p_1x + \cdots + p_{m-1}x^{m-1} + x^m$ selected to generate the field GF(2^m). Then

$$C(x) = A(x)B(x) \bmod P(x) =$$

$$= [b_0A(x) + b_1x\,A(x) + \cdots + b_{m-1}x^{m-1}A(x)] \bmod P(x). \qquad (3.1)$$

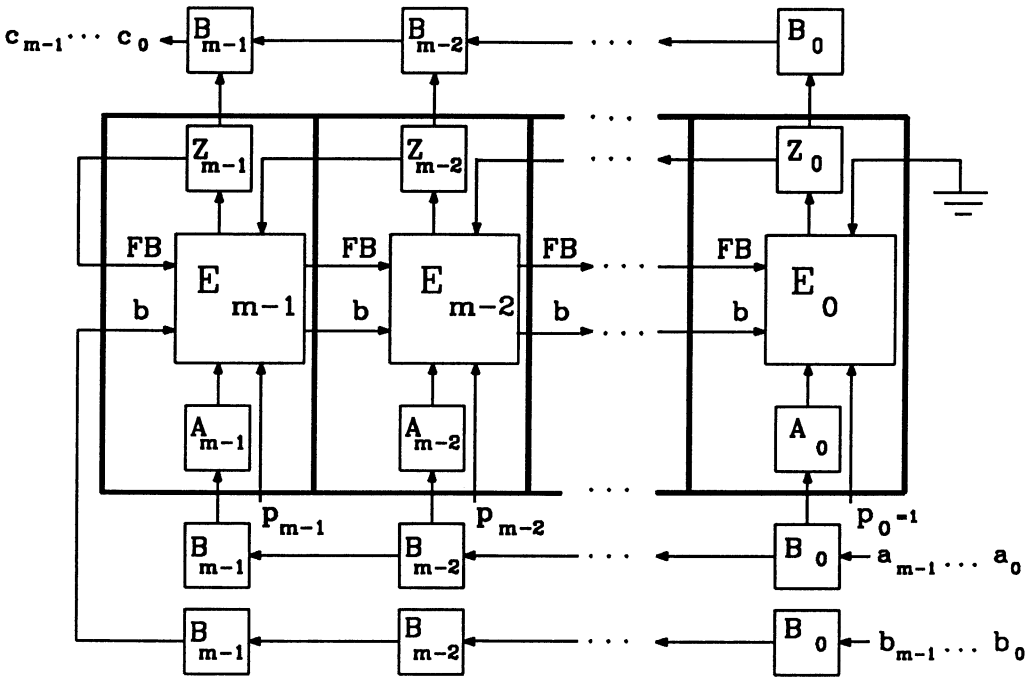
Each term $b_ix^iA(x)$ in eq. (3.1) can be computed recursively in the following way

$$b_ix^iA(x) \bmod P(x) = (\cdots((b_ix\underbrace{A(x) \bmod P(x))x \bmod P(x))\cdots)x}_{1} \underbrace{\bmod P(x)}_{2} \cdots \underbrace{\bmod P(x)}_{i} \qquad (3.2)$$

where the numbers below eq. (3.2) indicates that x appears exactly i times.

Based on eq. (3.1) and (3.2) we obtain the sequential multiplier of Fig. 3.1. The multiplier operates as follows. The Z register is initially set to zero. During the first cycle of operation, the polynomial $b_{m-1}A(x)$ is entered into the Z register. Upon the application of a clock pulse, both the Z and B registers shift left one position. The A register is left unchanged. Shifting the Z register is equivalent to multiplication by x, and since this could result in a term of degree greater than m-1, the polynomial $P(x)$ provides for a reduction of such a term by the equation $x^m = p_{m-1}x^{m-1} + p_{m-2}x^{m-2} + \ldots + p_0$. The result is $b_{m-1}xA(x) \bmod P(x)$. Added to this is the polynomial $b_{m-2}A(x)$ to produce the net result $[b_{m-1}xA(x) + b_{m-2}A(x)] \bmod P(x)$. Upon the application of the next clock cycle, the sequence of operations repeats itself to produce in the Z register the polynomial $\{[b_{m-1}xA(x) + b_{m-2}A(x)]x + b_{m-3}A(x)\} \bmod P(x)$. The product $C(x)$ is found in the Z register after m clock cycles.

We see, from Fig. 3.1, that the SSR multiplier is simple, it has a highly regular bit-slice architecture and requires simple control. Also, the structure is the same for any choice of field generator $P(x)$. To determine the speed of the multiplier we must determine the minimum length of the clock



B_i Pipelining Shift Register

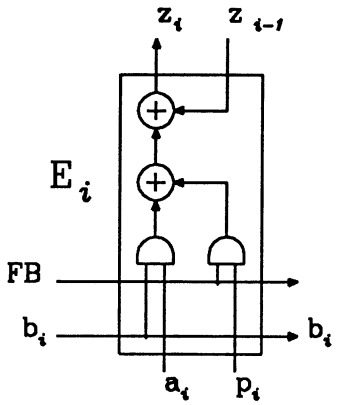


FIGURE 3.1 The SSR multiplier over GF(2^m).

cycle. This minimum length, in turn, is determined by the longest path along which a signal has to propagate within one cycle. We call this longest path the *critical path* and denote it by CP. The *length* of the CP is denoted $\mathcal{L}_{SSR}$. In Fig. 2.1 we see that $\mathcal{L}_{SSR} = 4$ (for example, the signal a_i has to drop through one register, one AND-gate and two XOR-gates), independently of the choice of $P(x)$. This means that the SSR multiplier allows the same (high) clock frequency for any m.

The complexity of SSR multiplier is denoted $\mathcal{C}_{SSR}$ and is linear in m; more exactly $\mathcal{C}_{SSR} = 6m$ ($2m$ registers and $4m$ gates).

Remark. The AND gate can be realized as a NAND since this does not change the function of the bit-slice of Fig. 3.1b.

The MSR multiplier. Another interesting architecture for bit-serial multiplication in PB is obtained by a further development of the SSR approach and results in what we call the modified shift-register (MSR) multiplier.

We start again from eq. (3.1) but develop it further:

$$C(x) = [b_0 A(x) + b_1 xA(x) + \cdots + b_{m-1}x^{m-1}A(x)] \bmod P(x) =$$

$$= [b_0 A(x) \bmod P(x)] + [b_1 xA(x) \bmod P(x)] + \cdots + [b_{m-1}x^{m-1}A(x) \bmod P(x)]. \tag{3.3}$$

We define now the polynomials $Z_{-,j}(x)$ as follows:

$$Z_{-,j}(x) = \sum_{i=0}^{m-1} z_{i,j}\, x^i = x^j A(x) \bmod P(x) \qquad j = 0, 1, \ldots, m-1 \tag{3.4}$$

where $z_{i,j} \in$ GF(2). Then

$$C(x) = b_0 Z_{-,0}(x) + b_1 Z_{-,1}(x) + \cdots + b_{m-1} Z_{-,m-1}(x) = \sum_{j=0}^{m-1} b_j Z_{-,j}(x). \tag{3.5}$$

And in matrix notation

$$C = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,m-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,m-1} \\ \vdots & \vdots & & \vdots \\ z_{m-1,0} & z_{m-1,1} & \cdots & z_{m-1,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix} = Z \cdot B \tag{3.6}$$

where Z is the m by m binary matrix in the middle of eq. (3.6). The columns of Z are just the m consecutive states of a Galois-type LFSR (Linear-Feedback Shift Register) with feedback polynomial $P(x)$ that has been initially loaded with A (= $Z_{-,0}$). The product is therefore obtained by first loading the LFSR with A, computing $b_0 Z_{-,0}$ and storing the result in an m-stage register. Next we clock the LFSR, compute $b_1 Z_{-,1}$, add it to $b_0 Z_{-,0}$, store the result and so forth. After m clock cycles the product is available in the lower register. The general form of the circuit implementing these operations is shown in Fig. 3.2*a*. The structure is simple, highly regular and independent of the field generator $P(x)$. Fig. 3.2*b* shows better the bit-slice character of the architecture. The complexity C_{MSR} is $6m$ ($2m$ registers and $4m$ gates) as for the SSR multiplier. The CP has length L_{MSR} = 3 (one register, one AND gate and one XOR gate) which is *one less* than for the SSR multiplier.

As a general multiplier the MSR architecture has all the good properties of the SSR architecture but a slightly better performance. In spite of this, the MSR multiplier is scarcely mentioned in the literature. To our knowledge it is mentioned only in [Ber69, Ch.2].

In Ch. 5 we will discuss another nice property of the MSR multiplier that is *not* shared by the SSR multiplier.

Other serial PB multipliers. A systolic multiplier based on an approach similar to the MSR approach is described in [Yeh84]. The systolic multiplier has about the same performance as the MSR multiplier but a much higher complexity of $15m$ ($10m$ registers and $5m$ gates according to [Yeh 84, Fig.3]). The systolic multiplier is thus not very attractive.

Another serial multiplier which, though, is viable only for particular choices of $P(x)$, will be described in Ch. 5.

3.2 Serial Dual-Basis Multipliers

The idea of using a dual-basis representation for performing bit-serial multiplication is due to Berlekamp [Ber82].

Let the set $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ be the dual basis of the polynomial basis $\{1, \alpha, \ldots, \alpha^{m-1}\}$ where α is a root of the field generator $P(x)$. Then, by the duality relation of Sec. 2.4 we know that $\mathrm{Tr}(\beta_i, \alpha^j) = \delta_{ij}$. Let A, B, and C be elements of GF(2^m) and $C = A \cdot B$ be the product that we wish to compute. To this end we

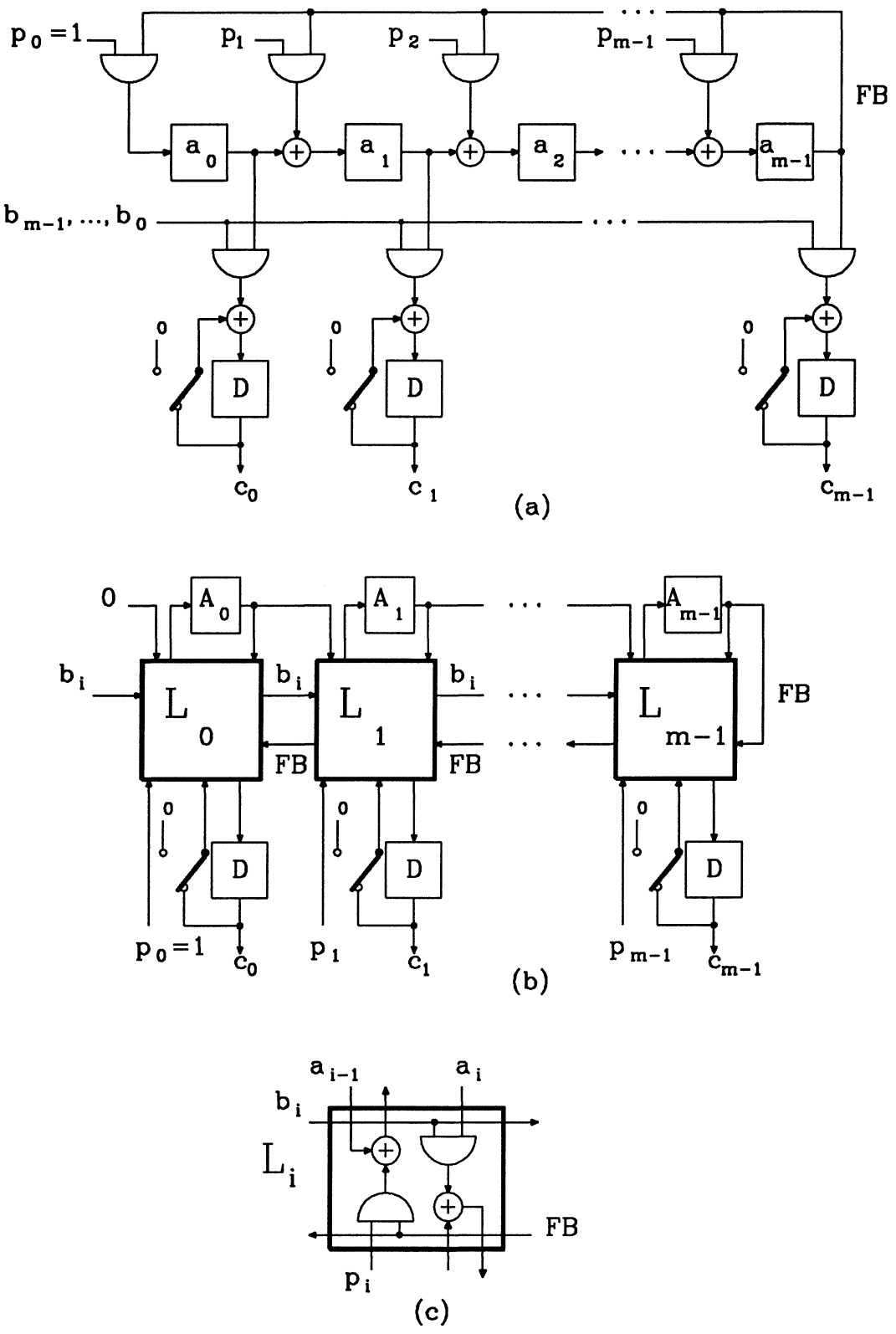


FIGURE 3.2 The MSR multiplier over GF(2^m) in original form (a), (b), (c) show an equivalent bit-slice structure.

assume A to be in DB representation and B in PB representation — i.e. $A = \sum a_i\beta_i$ and $B = \sum b_i\alpha^i$ where $a_i, b_i \in$ GF(2). By the properties of the trace function (see Th. 2.20) and the duality relation we obtain the first key fact:

$$\mathrm{Tr}(\alpha^j A) = \mathrm{Tr}(\alpha^j \sum a_i\beta_i) = \sum_{i=0}^{m-1} a_i \mathrm{Tr}(\beta_i\alpha^j) = a_j \qquad j = 0, 1, \ldots, m-1. \quad (3.7)$$

The second key fact is that multiplication by α in DB is a simple operation. Let $Y = \alpha A$. Then by eq. (3.7) we have:

$$y_j = \mathrm{Tr}(\alpha^j Y) = \mathrm{Tr}(\alpha^{j+1} A) = \begin{cases} a_{j+1}\,, & j = 0, 1, \ldots, m-2 \\ \mathrm{Tr}(\alpha^m A)\,, & j = m-1. \end{cases} \quad (3.8)$$

In order to compute $\mathrm{Tr}(\alpha^m A)$ we notice that $P(\alpha) = 0$ implies

$$\alpha^m = p_0 + p_1\alpha + \cdots + p_{m-1}\alpha^{m-1}. \quad (3.9)$$

Then by eq. (3.7), the duality relation and Th. 2.20 we have

$$\mathrm{Tr}(\alpha^m A) = \mathrm{Tr}((p_0 + p_1\alpha + \cdots + p_{m-1}\alpha^{m-1})A) =$$

$$= p_0\mathrm{Tr}(A) + p_1\mathrm{Tr}(\alpha A) + \cdots + p_{m-1}\mathrm{Tr}(\alpha^{m-1}A) = p_0a_0 + p_1a_1 + \cdots + p_{m-1}a_{m-1}$$

$$= (a_0, a_1, \ldots, a_{m-1})\cdot(p_0, p_1, \ldots, p_{m-1}) \quad (3.10)$$

where "$\cdot$" denotes the inner product. By eq. (3.8) and (3.10) we see that multiplication by α in DB can be implemented by a Fibonacci-type LFSR with feedback coefficients $p_0, p_1, \ldots, p_{m-1}$, see Fig. 3.3. We are now ready to compute the coefficients of the product C. Again by eq. (3.7) we have

$$c_j = \mathrm{Tr}(\alpha^j C) = \mathrm{Tr}((\alpha^j A)B) \qquad j = 0, 1, \ldots, m-1. \quad (3.11)$$

We determine c_0 explicitly (recall that B is in PB representation):

$$c_0 = \mathrm{Tr}(AB) = \mathrm{Tr}(b_0A) + \mathrm{Tr}(b_1\alpha A) + \cdots + \mathrm{Tr}(b_{m-1}\alpha^{m-1}A) =$$

$$= a_0b_0 + a_1b_1 + \cdots + a_{m-1}b_{m-1} = A\cdot B \quad (3.12)$$

To obtain the second coefficient c_1 we need only replace A by αA in eq. (3.12) and compute the inner product $(\alpha A)\cdot B$, the third coefficient c_2 is obtained by computing $(\alpha^2 A)\cdot B$ and so on for the remaining coefficients. We notice that the product C is in DB representation.

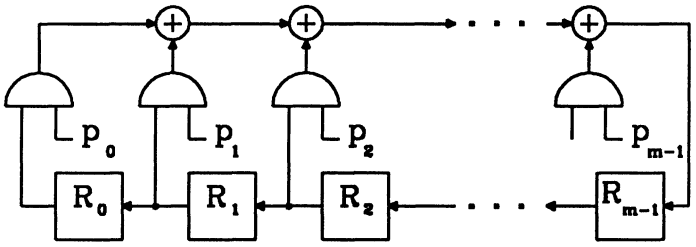


FIGURE 3.3 A Fibonacci-type LFSR of length m.

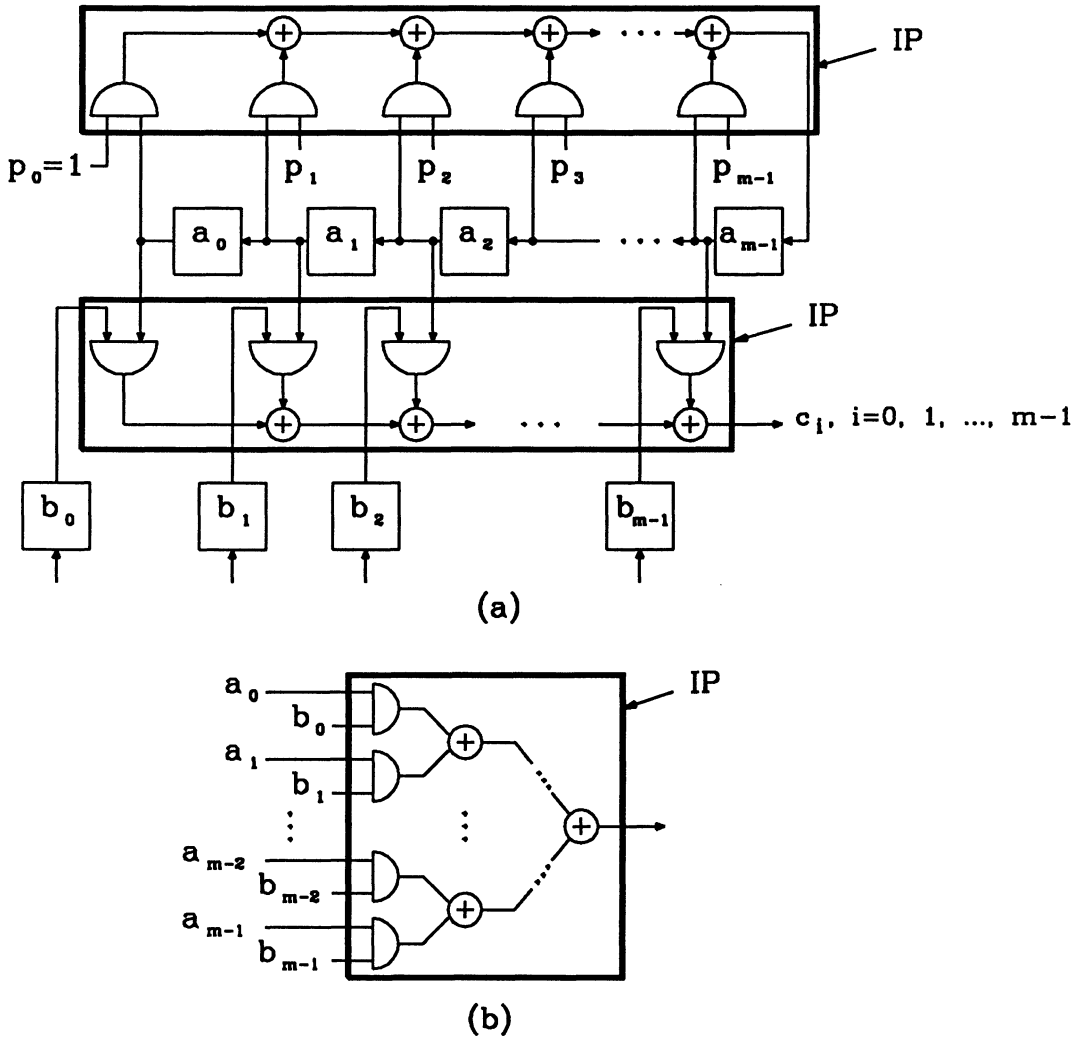


FIGURE 3.4 The DB multiplier (a) over GF(2^m). A faster realization of the inner-product (IP) logic is shown in (b).

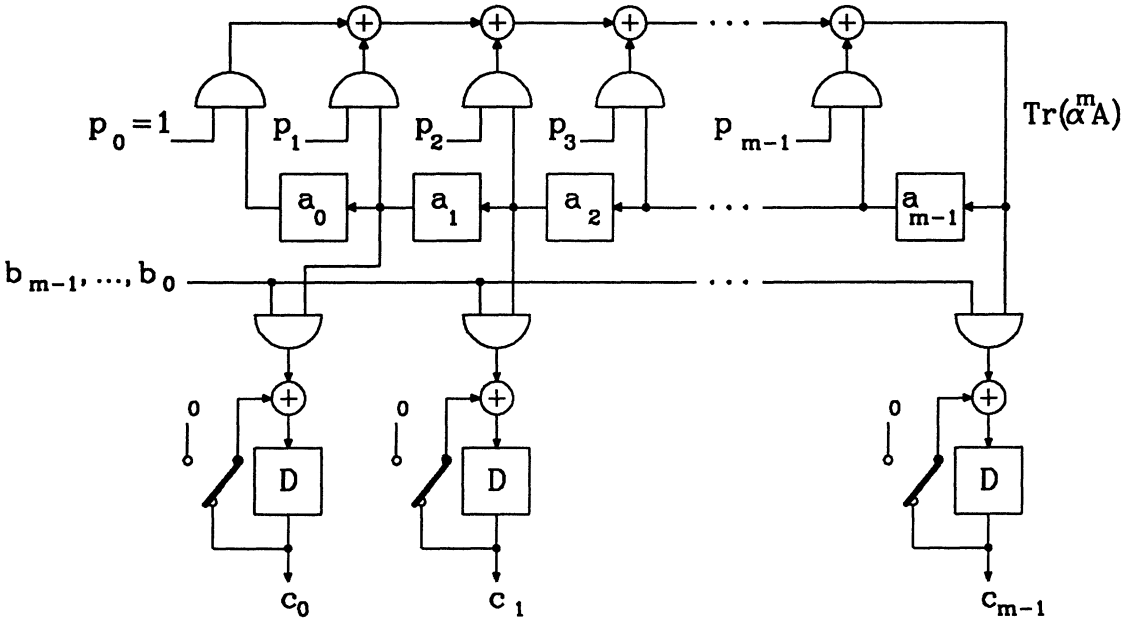


FIGURE 3.5 Another version of the DB multiplier over GF(2^m).

Figure 3.4 shows the complete DB multiplier. It consists of two m-stage registers and two identical sets of logic to compute the inner products. The upper register is initially loaded with A and the lower register with B whereby the first coefficient c_0 is computed by the lower logic. On the next clock pulse the feedback signal $\mathrm{Tr}(\alpha^m A)$ computed by the upper logic is fed into the highest stage of the upper register to produce there αA. The lower register retains B. The lower logic can now compute c_1. After m clock cycles the entire product has been produced and new elements A and B can be loaded into the registers.

The logic for computing the inner product consists of $2m$-1 gates (m AND and m-1 XOR gates). The complexity of the DB multiplier is therefore C_{BER} = $6m$-2 ($2m$ registers and $4m$-2 gates). The CP has length $L_{BER} = 2 + \lceil \log_2 m \rceil$ (one register + the inner-product logic) which means that the speed decreases, although slowly, with m. A drawback of the DB multiplier is that it requires the element A to be in dual basis whereas B must be in polynomial basis. The complete DB multiplier should therefore include some basis-changing logic for switching from dual to polynomial basis (if we assume that both A and B are given in DB). Whenever the field generator can be chosen to be a trinomial, the change of basis can be reduced to a simple permutation of the co-ordinates [Mor89].

The simultaneous presence of two different bases implies also that the field generator $P(x)$ can not be easily changed. So, although the DB multiplier has a simple structure it is not trivial to adapt it to different choices of $P(x)$ or to expand it to extension fields of different order.

It should be noted that the DB multiplier can be realized in a form similar to the MSR multiplier, as indicated in Fig. 3.5. In this alternative realization the element B is fed serially into the multiplier. The product is available in parallel form after m clock cycles.

3.3 Serial Normal-Basis Multipliers

Massey and Omura [Mae83] realized that a new architecture for bit-serial multiplication could be obtained by adopting the NB representation.

Let $P(x)$ be a prime polynomial of degree m over GF(2) with linearly independent roots and let α be one of the roots. Then, the set $\{\alpha, \alpha^2, \ldots, \alpha^{2^{m-1}}\}$ forms a normal basis of GF(2^m). The key property of the NB representation is

that squaring is a very simple operation. Let $A = a_0\alpha + a_1\alpha^2 + \cdots + a_{m-1}\alpha^{2^{m-1}}$ be in GF(2^m). Then

$$A^2 = a_0\alpha^2 + a_1\alpha^4 + \cdots + a_{m-1}\alpha^{2^m} = a_{m-1}\alpha + a_0\alpha^2 + \cdots + a_{m-2}\alpha^{2^{m-1}}. \qquad (3.13)$$

In eq. (3.13) we utilized the linearity of the squaring operation (see Th. 2.17) and the fact that $\alpha^{2^m} = \alpha$. Squaring in NB is thus a simple cyclic shift of the element's bits.

Let $C = AB = (c_0, c_1, \ldots, c_{m-1})$ be the product to be computed (everything is assumed to be in NB). Then the last coefficient c_{m-1} of C is some function f of the coefficients of A and B, i.e.

$$c_{m-1} = f(a_0, a_1, \ldots, a_{m-1}; b_0, b_1, \ldots, b_{m-1}). \qquad (3.14)$$

Now, by eq. (3.13) we have

$$C^2 = A^2B^2 = (a_{m-1}, a_0, \ldots, a_{m-2})\cdot(b_{m-1}, b_0, \ldots, b_{m-2}) =$$

$$= (c_{m-1}, c_0, \ldots, c_{m-2}) \qquad (3.15)$$

which means that the last coefficient c_{m-2} of C^2 can be obtained by applying the same function f to the components of A^2 and B^2. By squaring C repeatedly, it is now evident that

$$c_{m-1} = f(a_0, a_1, \ldots, a_{m-1}; b_0, b_1, \ldots, b_{m-1})$$

$$c_{m-2} = f(a_{m-1}, a_0, \ldots, a_{m-2}; b_{m-1}, b_0, \ldots, b_{m-2})$$

$$\vdots$$

$$c_0 = f(a_1, a_2, \ldots, a_0; b_1, b_2, \ldots, b_0). \qquad (3.16)$$

The above equations define the Massey-Omura (MO) multiplier. Fig. 3.6 shows the general structure of the serial MO multiplier. Clearly, the complexity of this multiplier depends heavily on the function f which, in turn, depends on the choice of normal basis — that is, on the choice of $P(x)$. The determination of f is cumbersome [Eri86] and is best described by an example .

Example 3.1 Let $P(x) = 1 + x^3 + x^4$ be the generator of GF(16). Fig. 3.7 shows the PB representations of the (non-zero) field elements.

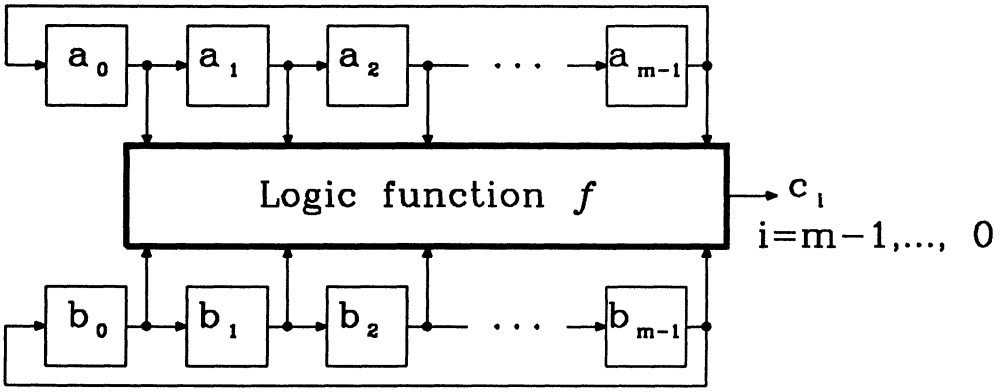


FIGURE 3.6 General structure of the serial MO multiplier.

	$1\ \alpha\ \alpha^2\ \alpha^3$		$1\ \alpha\ \alpha^2\ \alpha^3$		$1\ \alpha\ \alpha^2\ \alpha^3$
α^0	1 0 0 0	α^5	1 1 0 1	α^{10}	0 1 0 1
α^1	0 1 0 0	α^6	1 1 1 1	α^{11}	1 0 1 1
α^2	0 0 1 0	α^7	1 1 1 0	α^{12}	1 1 0 0
α^3	0 0 0 1	α^8	0 1 1 1	α^{13}	0 1 1 0
α^4	1 0 0 1	α^9	1 0 1 0	α^{14}	0 0 1 1

Figure 3.7 The PB representation of the field GF(16) with $\alpha^4 = 1 + \alpha^3$.

From Fig. 3.7 we see that

$$\begin{pmatrix} \alpha \\ \alpha^2 \\ \alpha^4 \\ \alpha^8 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \alpha \\ \alpha^2 \\ \alpha^3 \end{pmatrix} \tag{3.17}$$

and consequently,

$$\begin{pmatrix} 1 \\ \alpha \\ \alpha^2 \\ \alpha^3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \alpha^2 \\ \alpha^4 \\ \alpha^8 \end{pmatrix}. \tag{3.18}$$

We call the m by m binary matrix of eq. (3.18) the *transformation matrix* T (the matrix of eq. (3.17) is clearly T^{-1}). With the aid of the matrix T we can easily go from polynomial to normal basis.

Let us perform an ordinary (i.e. not mod $P(x)$) multiplication of A and B,

$$A \cdot B = \sum_{i=0}^{3} a_i \alpha^{2^i} \cdot \sum_{j=0}^{3} b_j \alpha^{2^j} = \sum_{i=0}^{3} \sum_{j=0}^{3} a_i b_j \alpha^{2^i + 2^j}. \tag{3.19}$$

The 16 powers of α in eq. (3.19) are first expressed in PB with the aid of Fig. 3.7 and then transformed into NB by applying the transformation matrix T. Let $\lambda_p(i,j)$ and $\lambda_n(i,j)$ be the PB and NB representation of $\alpha^{2^i + 2^j}$ respectively, then

$$\alpha^{2^i + 2^j} = \sum_{l=0}^{3} \lambda_{p,l}(i,j) \alpha^l = \sum_{l=0}^{3} \lambda_{n,l}(i,j) \alpha^{2^l}. \tag{3.20}$$

By eq. (3.19) and (3.20) we have

$$A \cdot B = \sum_{l=0}^{3} \sum_{i=0}^{3} \sum_{j=0}^{3} \lambda_{n,l}(i,j) a_i b_j \alpha^{2^l} = \sum_{l=0}^{3} c_i \alpha^{2^l} \tag{3.21}$$

and by identification

$$c_l = \sum_{i=0}^{3} \sum_{j=0}^{3} \lambda_{n,l}(i,j) a_i b_j. \tag{3.22}$$

In particular, we have

$$c_3 = f(A,B) = \sum_{i=0}^{3} \sum_{j=0}^{3} \lambda_{n,3}(i,j) a_i b_j. \tag{3.23}$$

i	j	$k=2^i+2^j$	α	α^2	α^4	α^8
0	0	2	0	1	0	0
0	1	3	1	1	0	1
0	2	5	1	0	1	0
0	3	9	1	0	1	1
1	0	3	1	1	0	1
1	1	4	0	0	1	0
1	2	6	1	1	1	0
1	3	10	0	1	0	1
2	0	5	1	0	1	0
2	1	6	1	1	1	0
2	2	8	0	0	0	1
2	3	12	0	1	1	1
3	0	9	1	0	1	1
3	1	10	0	1	0	1
3	2	12	0	1	1	1
3	3	16	1	0	0	0

Figure 3.8 The NB representation of the powers α^k.

The values of the binary coefficients $\lambda_{n,3}(i,j)$ are given in the right-most column of Fig. 3.8. Nine of these coefficients are non-zero whereby the following function f is obtained

$$f(A,B) = a_0b_1 + a_0b_3 + a_1b_0 + a_1b_3 + a_2b_2 + a_2b_3 + a_3b_0 + a_3b_1 + a_3b_2. \tag{3.24}$$

Fig. 3.9 shows the complete multiplier. □

In the above example the number of terms $b_i c_j$ in f could have been reduced to 7 by choosing $P(x) = 1 + x + x^2 + x^3 + x^4$. This polynomial is though

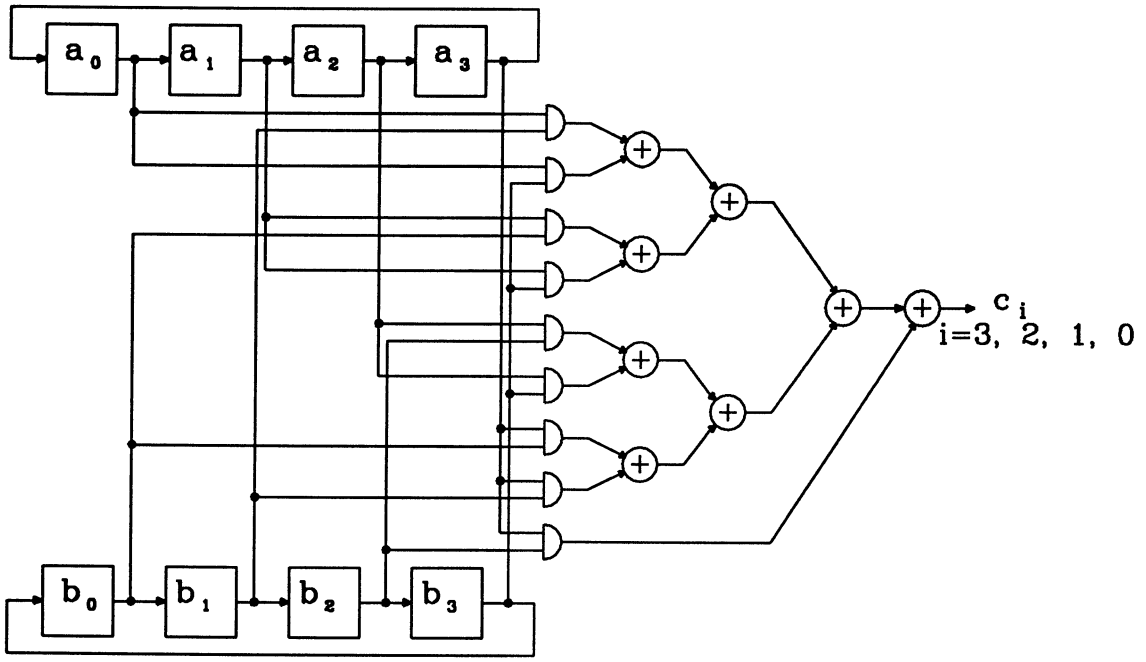


FIGURE 3.9 A realization of the serial MO multiplier over GF(16) with $P(x)=1+x^3+x^4$.

non-primitive. The field generator chosen in Ex. 3.1 is the only primitive polynomial of degree 4 with linearly independent roots [Pet61, pp.472-476] so the choice was easy. For larger degrees there are many more possible choices. As a measure of the complexity of the MO multiplier we use the number of terms $a_i b_j$ in f and denote it by N_m. We let a computer check all prime polynomials of degree ≤ 16 in order to find, for each degree, the polynomial minimizing N_m. Table 3.10 shows the result of our search.

m	$P(x)$	N_m
2	0,1,2†	3
3	0,2,3†	5
4	0,1,2,3,4*†	7
4	0,3,4	9
5	0,1,2,4,5†	9
6	0,1,4,5,6†	11
7	0,2,5,6,7	19
8	0,3,5,7,8	21
9	0,1,4,5,6,8,9†	17
10	0,1,2,3,4,5,6,7,8,9,10*†	19
10	0,1,4,9,10	37
11	0,2,3,4,8,10,11†	21
12	0,1,2,3,4,5,6,7,8,9,10,11,12*†	23
12	0,1,2,5,6,7,8,11,12	41
13	0,3,4,7,10,12,13	45
14	0,1,8,9,12,13,14†	27
15	0,2,4,5,7,9,12,14,15*	45
15	0,5,7,9,12,14,15	53
16	0,1,2,3,5,7,8,10,11,12,13,15,16	85

Table 3.10 The polynomials of degree ≤ 16 for which N_m is minimized. Only the actual powers of x are given; for example 0,1,2 means $1+x+x^2$. Non-primitive polynomials are marked by *. Polynomials yielding an optimal NB — i.e. with $N_m = 2m-1$ — are marked by †.

In [Mul89] it is shown that $N_m \geq 2m\text{-}1$ for any NB and the concept of *optimal normal basis* is introduced. An NB is said to be optimum if $N_m = 2m-1$. Some polynomials yielding an optimal NB are shown in Tab. 3.10. In [Mul89] it is shown that an optimal normal basis exists in ~ 23% of the fields GF(2^m), $2 \leq m < 1200$.

The work of [Mul89] has been extended in [Mey90] to $m \leq 2436$. In Appendix A we reproduce a table given in [Mey90, pp.96] with the values of m for which an optimal normal basis can be obtained in GF(2^m) by using the two constructions given in [Mul89, Th. 2 & Th. 3]. The first construction is

called type-I in [Mul89] and requires $P(x)$ to be an all-ones polynomial. Prime all-ones polynomials exist for example for (see Th. 2.27)

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

However, the practical importance of this class of prime polynomials is limited since we will see (in Sec. 4.5) that it contains only one primitive polynomial ($m = 2$). The second construction is called type-II in [Mul89] and seems (according to our incomplete tests) to yield primitive optimal NB's. In [Mey90, pp.95] a table on the number of optimal NB's in GF(p^m), p =2, 3, 5, 7, 11, 13, 17, 19, is presented where one finds the following figures for $p = 2$.

	$m \le 1000$	$m \le 5000$
type-I	67	255
type-II	177	702

Apparently, it is fair to say that primitive optimal NB's are rare.

Some constructions yielding non-optimal normal bases of low complexity are discussed in [Ash89]. In Appendix B we reproduce parts of a table over N_m given in [Ash89] for values of m — with 2^m-1 a (Mersenne) prime — for which a low-complexity NB can be obtained by the constructions presented in [Ash].

In order to compute the complexity C_{MO} in terms of gates and registers of the MO multiplier we write the function f in the following form

$$f(A,B) = AMB^t \tag{3.25}$$

where M is a binary m by m matrix with exactly N_m non-zero entries. The computation of AM requires N_m-m XOR gates whereas the computation of the inner product $(AM)\cdot B$ requires $2m-1$ AND/XOR gates. To this we must add the $2m$ registers and obtain $C_{MO} = 3m + N_m - 1$. By the lower bound on N_m mentioned above we get

$$C_{MO} \ge 5m - 2 \tag{3.26}$$

with equality only for optimal normal bases.

The CP goes through one register and the logic function f. The length of the CP is thus $L_{MO} = 2 + \lceil \log_2 N_m \rceil$ and, again, by the lower bound we have

$$L_{MO} \geq 2 + \lceil \log_2(2m-1) \rceil = 3 + \lceil \log_2 m \rceil. \quad (3.27)$$

Other serial NB multipliers. Some alternative serial NB architectures are presented in [Gei88]. There, the authors propose serial input/parallel output structures and also, inspired by Berlekamp, structures utilizing different basis representations (dual and normal). However, their results indicate that, in general, no significant reduction in complexity will be obtained by adopting the proposed architectures.

3.4 Discussion

In order to facilitate the discussion and comparison of the different architectures presented in the previous sections, we have compiled a table with the main properties of the multipliers. This table is shown in Fig. 3.11.

Multiplier/Basis	Complexity C	Length of CP L	Delay min	Delay max	Regularity	Easy to expand
MSR/Polynomial	$6m/4m$	3	m	$2m$	High	Yes
SSR/Polynomial	$6m/4m$	4	m	$2m$	High	Yes
Berlekamp's/Dual	$6m/4m$	$\geq 2+\lceil \log_2 m \rceil$	m	m	Low	No
MO/Normal	$\geq 5m$	$\geq 3+\lceil \log_2 m \rceil$	m	m	Low	No

Figure 3.11 The main properties of some serial multipliers over GF(2^m). The double complexity figures are for generic $P(x)$ ($6m$) and fixed $P(x)$ ($4m$). The delay indicated is that between first input and first output.

We will now comment on each of the five properties listed in Fig. 3.11.

Complexity. The left figures ($6m$) for the SSR, MSR and DB multipliers are for the generic field generator $P(x)$. The right figures ($4m$) are for field generators of low Hamming weight — that is, trinomials and pentanomials — in which case about $2m$ gates are saved. In the complexity of DB multiplier we did not include any basis-changing logic. As mentioned earlier, no extra logic is required if $P(x)$ is a trinomial. Prime trinomials exist for example in ~ 55% of the cases where $2 \leq m \leq 1000$ [Zie68]. The next-to-best choice seems to be a pentanomial of the form $1 + x^k + x^{k+1} + x^{k+2} + x^m$ in which case only a few XOR gates are required to change basis [Mor89].

However, as far as we know, no one has shown that there are enough many prime (and, in particular, primitive) pentanomials of this type.

For the MO multiplier it is not (yet) meaningful to speak of a generic $P(x)$ since we can give an exact expression of the complexity only for polynomials yielding an optimal NB; as mentioned earlier, such polynomials are rare. For non-optimal NB the minimum complexity might be significantly larger than the lower bound.

Performance. The performance of a multiplier (i.e. its maximum clock frequency) is inversely proportional to the length L of its CP. The following is a table on the lower bounds on L_{MO} and L_{BER} for $m \leq 16$.

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$L_{BER} \geq$	3	4	4	5	5	5	5	6	6	6	6	6	6	6	6
$L_{MO} \geq$	4	5	5	6	6	6	6	7	7	7	7	7	7	7	7

Comparing with the figure of the MSR multiplier ($L_{MSR} = 3, \forall\ m$) we see that the DB multiplier is at most half as fast for $m \geq 9$. The same is true for the MO multiplier already for $m \geq 5$. The three underlined m-values are those for which no prime trinomials exist [Zie68] in the above range. In these cases some basis-changing logic is needed in the DB multiplier which means that L_{BER} is at least one unit larger than indicated in the above table. By adopting the alternative architecture of Fig. 3.5, the DB multiplier can be given the same performance of the MSR multiplier as long as $P(x)$ is a trinomial.

Delay. The delay between first input and first output of the two PB multipliers might be as much as $2m$ clock cycles. This would occur under the following conditions: 1) both multiplicands are fed simultaneously into the multiplier in serial fashion *and* 2) the multiplier is operated discontinuously in time. If both conditions are met, the pipelining registers will be empty from time to time and the reloading of these will cause an extra delay of m cycles.

Regularity and expandability. The nice bit-slice architecture of the two PB multipliers simplifies significantly the VLSI design, in particular for larger values of m. Furthermore, the total independence on the choice of $P(x)$ makes them trivially expanded to any other extension field.

None of this two properties is exhibited by the DB and NB multipliers. Worst, in this sense, is the MO multiplier where $P(x)$ appears only implicitly (through the function f) in the architecture. Not even the alternative architectures of [Gei88] exhibit nice structural properties.

Our conclusion is that the PB multipliers have the most favourable properties for purpose of VLSI design of a generic multiplier over GF(2^m). Among the two, the MSR multiplier stands out since it yields maximum performance for any m.

Chapter 4

Bit-Parallel Multiplication

We turn our attention to fast bit-parallel (or simply parallel) multipliers over GF(2^m). By a parallel multiplier we intend a device that performs multiplication of two arbitrary field elements in one single step.

The chapter gives a new, detailed treatment of the parallel PB multiplier first introduced in [Bar63]. We will show that this multiplier has surprisingly and, as far as we know, unrecognized good properties, in particular for purpose of VLSI design. We present bounds on its complexity and performance, and derive simple criteria for selecting good field generators that yield low complexity and high performance. We identify and analyze thoroughly two classes of good field generators: the trinomials and the equally spaced polynomials. Further, we present the best field generators for $m \leq 16$ and show that these can be easily spotted by our selection criteria. Finally we compare with other PB and NB architectures.

4.1 Bit-Parallel Multiplication in Polynomial Basis

In Sec. 3.1 we saw that the product $C(x) = A(x)B(x) \bmod P(x)$ can be written, in matrix form, as $C = ZB$ where Z is a binary m by m matrix whose columns are the m consecutive states of a Galois-type LFSR that has been initialized by A. There we wanted to design a sequential multiplier and it was therefore natural to utilize just an LFSR to produce Z. Here, however, we are interested in fast parallel multipliers which means that we wish to generate *all* entries of Z concurrently. This will be accomplished by using dedicated combinational logic for each distinct entry. Furthermore, a set of m identical blocks will be used to concurrently compute the m inner products involved in ZB.

Since we take a different view of the matrix Z we find it convenient to change the notation slightly. We denote the entry $z_{i,j}$ of Z by $f_{i,j}(A)$, or simply $f_{i,j}$, since we will see that each entry is a linear function of the components of A.

We have thus,

$$C(x) = A(x)B(x) \bmod P(x) = \sum_{i=0}^{m-1} c_i x^i$$

$$\text{where } c_i = a_0 f_{i,0}(A) + a_1 f_{i,1}(A) + \cdots + a_{m-1} f_{i,m-1}(A) \tag{4.1}$$

with the functions $f_{i,j}$ having the form

$$f_{i,j}(A) = \sum_{\substack{\text{for some } k \\ k \in [0, 1, \ldots, m-1]}} a_k.$$

Multiplication can now be described through the functions $f_{i,j}$ which, in turn, depend on the irreducible polynomial $P(x)$. In matrix notation we can write the product

$$C = \begin{pmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,m-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,m-1} \\ \vdots & \vdots & & \vdots \\ f_{m-1,0} & f_{m-1,1} & \cdots & f_{m-1,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix} = Z\,B. \tag{4.2}$$

We call Z the the *product matrix.*

Example 4.1 Let $P(x) = 1 + x + x^4$ generate $GF(16)$. Let further $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $B(x) = b_0 + b_1x + b_2x^2 + b_3x^3$ be two field elements. Then

$$C(x) = \{a_0b_0 + x(a_1b_0 + a_0b_1) + x^2(a_2b_0 + a_1b_1 + a_0b_2) +$$

$$x^3(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3) + x^4(a_3b_1 + a_2b_2 + a_1b_3) +$$

$$x^5(a_3b_2 + a_2b_3) + x^6(a_3b_3)\} \bmod (x^4 + x + 1) \tag{4.3a}$$

and using the fact that $x^4 = x + 1$ we obtain

$$C(x) = [b_0a_0 + b_1a_3 + b_2a_2 + b_3a_1] +$$

$$x[b_0a_1 + b_1(a_0 + a_3) + b_2(a_3 + a_2) + b_3(a_2 + a_1)] +$$

$$x^2[b_0a_2 + b_1a_1 + b_2(a_0 + a_3) + b_3(a_3 + a_2)] +$$

$$x^3[b_0a_3 + b_1a_2 + b_2a_1 + b_3(a_0 + a_3)] \qquad \bmod (x^4 + x + 1). \tag{4.3b}$$

Comparing eq. (4.3*b*) with eq. (4.1) yields the product matrix

$$Z = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0+a_3 & a_3+a_2 & a_2+a_1 \\ a_2 & a_1 & a_0+a_3 & a_3+a_2 \\ a_3 & a_2 & a_1 & a_0+a_3 \end{pmatrix}. \quad \square$$

Although the determination of Z is straightforward, the calculations become soon tedious with growing m. It is therefore desirable to have explicit formulas for the functions $f_{i,j}$. In order to derive such formulas we formalize the algorithm of Ex. 4.1. For simplicity of notation we introduce the following symbol

$$s_k = \sum_{u+v=k} b_u a_v \qquad u,v \in [0, 1, 2, \ldots, m-1]. \tag{4.4}$$

Further, we define an $m-1$ by m binary matrix Q which we call the *reduction matrix*,

$$\begin{pmatrix} x^m \\ x^{m+1} \\ \vdots \\ x^{2m-2} \end{pmatrix} = Q \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} =$$

$$= \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,m-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,m-1} \\ \vdots & \vdots & & \vdots \\ q_{m-2,0} & q_{m-2,1} & \cdots & q_{m-2,m-1} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} \mod P(x). \tag{4.5}$$

The product of two polynomials of degree at most $m-1$ results in a polynomial of degree at most $2m-2$. The reduction modulo $P(x)$ of such a polynomial implies that the powers $x^m, x^{m+1}, \ldots, x^{2m-2}$ have to be represented in polynomial basis and the reduction matrix tells us explicitly how this is done. Hence, the reduction matrix contains all information we need to characterize the product matrix Z. The reduction matrix is obtained from $P(x)$ by a simple shift & add-on-overflow procedure that can be applied by hand even for moderately large m. In particular, we notice that the first row of Q is given directly by $P(x)$, i.e. $q_{0,j} = p_j$ for $j = 0, 1, \ldots, m-1$.

Proposition 4.1 Let the Galois field GF(2^m) be generated by the prime polynomial $P(x)$ and let Q be the reduction matrix associated with $P(x)$. Let $A(x)$ be an arbitrary element of GF(2^m).

Then the entries of the product matrix Z are given by

$$f_{i,0}(A) = a_i \quad i = 0, 1, \ldots, m-1 \tag{4.6a}$$

and

$$f_{i,j}(A) = \sigma(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i}a_{m-1-t} \quad i = 0, 1, \ldots, m-1, j = 1, \ldots, m-1 \tag{4.6b}$$

where $\sigma(k)$ is a step function defined by

$$\sigma(k) = \begin{cases} 1 & k \geq 0 \\ 0 & k < 0. \end{cases}$$

Proof. The product $C(x) = A(x)B(x) \bmod P(x)$ can be expressed as

$$\mathrm{C(x)} = \sum_{k=0}^{2m-2} x^k s_k \bmod P(x) = \sum_{k=0}^{m-1} x^k s_k + \sum_{k=m}^{2m-2} [x^k \bmod P(x)]s_k. \tag{4.7}$$

By the reduction matrix we have

$$x^k = \sum_{j=0}^{m-1} q_{k-m,j}x^j \bmod P(x) \qquad k = m, m+1, \ldots, 2m-2. \tag{4.8}$$

Inserting eq. (4.8) into (4.7) yields

$$C(x) = \sum_{k=0}^{m-1} x^k s_k + \sum_{k=m}^{2m-2} \left(\sum_{j=0}^{m-1} q_{k-m,j}x^j\right) s_k = \sum_{k=0}^{m-1} x^k s_k + \sum_{k=0}^{m-2} \left(\sum_{j=0}^{m-1} q_{k,j}x^j\right) s_{k+m} =$$

$$= \sum_{k=0}^{m-2} q_{k,0}s_{k+m} + \sum_{k=0}^{m-2} q_{k,1}x s_{k+m} + \cdots +$$

$$\sum_{k=0}^{m-2} q_{k,m-1}x^{m-1} s_{k+m} + \sum_{k=0}^{m-1} x^k s_k \,. \tag{4.9}$$

From eq. (4.9) we can extract the following expression for the product coefficient c_i

$$c_i = s_i + \sum_{k=0}^{m-2} q_{k,i} s_{k+m} =$$

$$= s_i + q_{0,i} s_m + q_{1,i} s_{m+1} + \cdots + q_{k,i} s_{m+k} + \cdots + q_{m-2,i} s_{2m-2}. \qquad (4.10)$$

Now we utilize the definition of s_k given in eq. (4.4) and organize the terms in the right-hand side of eq. (4.10) in a more convenient form which allows us to identify all the terms $b_u a_v$ involved there. This convenient form is shown in Tab. 4.1. The first row shows the possible range of the index u. The second row shows the range of the index v for the term s_i, the third row shows the range of the index v for the term $q_{0,i} s_m$ and so on for the other rows. For example, in the third column (i.e. $u = 2$) we can identify the term $b_2(a_{i-2} + q_{0,i} a_{m-2} + q_{1,i} a_{m-1})$. This term is thus included in eq. (4.10).

	u:	0	1	2	3	...	i	i+1	...	m-2	m-1
s_i	v:	i	i-1	i-2	i-3	...	0				
$q_{0,i} s_m$	v:		m-1	m-2	m-3	...	m-i	m-i-1	...	2	1
$q_{1,i} s_{m+1}$	v:			m-1	m-2	...	m-i+1	m-i	...	3	2
	:				:	:	:	:	:	:	:
$q_{i,i} s_{m+i}$	v:							m-1	...	i+2	i+1
	:								:	:	:
$q_{m-3,i} s_{2m-3}$	v:									m-1	m-2
$q_{m-2,i} s_{2m-2}$	v:										m-1

Table 4.1

With the help of Tab. 4.1 we can write eq. (4.10) in the following way

$$c_i = b_0 a_i + b_1(a_{i-1} + q_{0,i} a_{m-1}) + b_2(a_{i-2} + q_{0,i} a_{m-2} + q_{1,i} a_{m-1}) + \cdots +$$

$$b_i(a_0 + q_{0,i}a_{m-i} + q_{1,i}a_{m-i+1} + \cdots + q_{i-1,i}a_{m-1}) +$$

$$b_{i+1}(q_{0,i}a_{m-i-1} + q_{1,i}a_{m-i} + \cdots + q_{i,i}a_{m-1}) + \cdots +$$

$$b_{m-1}(q_{0,i}a_1 + q_{1,i}a_2 + \cdots + q_{m-2,i}a_{m-1}) =$$

$$b_0a_i + b_1(a_{i-1} + q_{0,i}a_{m-1}) + b_2(a_{i-2} + \sum_{t=0}^{1} q_{1-t,i}a_{m-1-t}) + \cdots +$$

$$b_i(a_0 + \sum_{t=0}^{i-1} q_{i-1-t,i}a_{m-1-t}) + b_{i+1}(\sum_{t=0}^{i} q_{i-t,i}a_{m-1-t}) + \cdots +$$

$$a_{m-1}(\sum_{t=0}^{m-2} q_{m-2-t,i}a_{m-1-t}). \qquad (4.11)$$

Comparison of eq. (4.11) with eq. (4.1) yields the functions $f_{i,j}$. □

With the help of Prop. 4.1 we can easily determine the product matrix associated with an arbitrary field generator $P(x)$.

Example 4.1 (Cont'd) The reduction matrix is

$$Q = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

Applying Prop. 4.1 on Q yields

$f_{i,0} = a_i \qquad i = 0, 1, \ldots, m-1$

$f_{0,1} = q_{0,0}a_3 = a_3 \qquad f_{0,2} = q_{1,0}a_3 + q_{0,0}a_2 = a_2$

$f_{0,3} = q_{2,0}a_3 + q_{1,0}a_2 + q_{0,0}a_1 = a_1$

$f_{1,1} = a_0 + q_{0,1}a_3 = a_0 + a_3 \qquad f_{1,2} = q_{1,1}a_3 + q_{0,1}a_2 = a_3 + a_2$

$f_{1,3} = q_{2,1}a_3 + q_{1,1}a_2 + q_{0,1}a_1 = a_2 + a_1$

$f_{2,1} = a_1 + q_{0,2}a_3 = a_1 \qquad f_{2,2} = a_0 + q_{1,2}a_3 + q_{0,2}a_2 = a_0 + a_3$

$f_{2,3} = q_{2,2}a_3 + q_{1,2}a_2 + q_{0,2}a_1 = a_3 + a_2$

$f_{3,1} = a_2 + q_{0,3}a_3 = a_2 \qquad f_{3,2} = a_1 + q_{1,3}a_3 + q_{0,3}a_2 = a_1$

$$f_{3,3} = a_0 + q_{2,3}a_3 + q_{1,3}a_2 + q_{0,3}a_1 = a_0 + a_3.$$

The above functions yield the same matrix Z as before. □

4.2 Realization

We consider the implementation of a parallel PB multiplier based on the algorithm of the previous section. For sake of consistency and simplicity of analysis we continue to design circuits making use of 2-input gates. A gate with $n > 2$ inputs will always be realized as a tree of 2-input gates. In general, this approach yields circuits with better performance than for special-purpose n-input gates [Wes85, Sec. 5.3.3].

By eq. (4.1) we know that the i:th product bit is obtained computing the following inner product

$$c_i = A \cdot f_{i,\cdot} = a_0 f_{i,0} + a_1 f_{i,1} + \cdots + a_{m-1} f_{i,m-1} \qquad i = 0, 1, \ldots, m-1 \tag{4.12}$$

where $f_{i,\cdot}$ denotes the i:th row of Z and addition and multiplication are taken modulo 2. The multiplier is naturally divided into three subsystems. The first subsystem computes the functions $f_{i,j}$ and is called the f-network. The second subsystem is called the IP network and consists of m identical cells where each cell computes one inner product (IP) and is called the IP cell. The third subsystem is the interconnection bus (IB) between the f-network and the IP network. Fig. 4.1a shows the general structure of the multiplier whereas Fig. 4.1b shows the circuit diagram of the general IP cell. Fig. 4.2 shows the complete multiplier of Ex. 4.1.

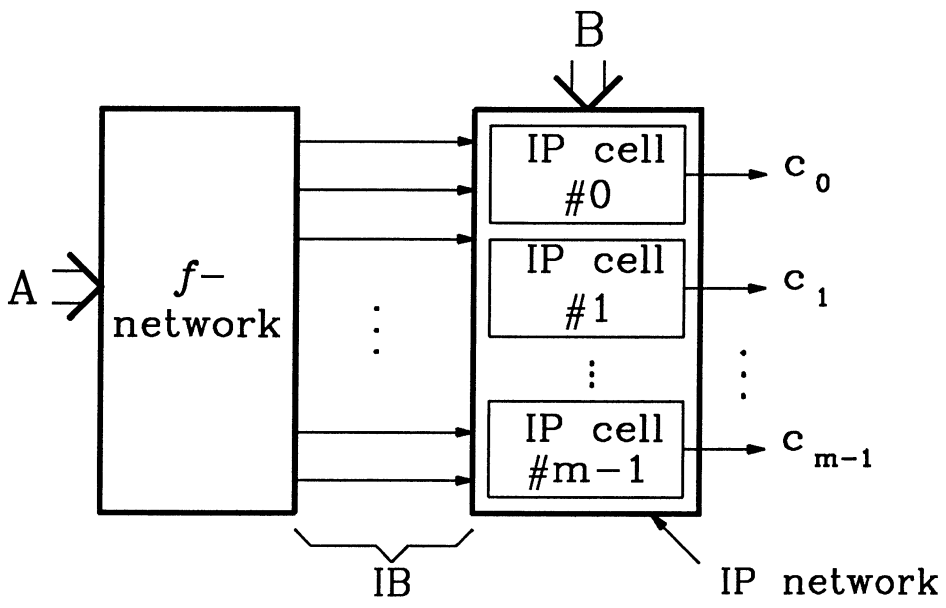
The structure of IP network displays a nice m-fold regularity. The f-network can be given the same type of regularity as follows.

We know from Sec. 3.1 that the i:th column of Z is given by $x^j A(x)$ mod $P(x)$. In particular, for $j = 1$ we have

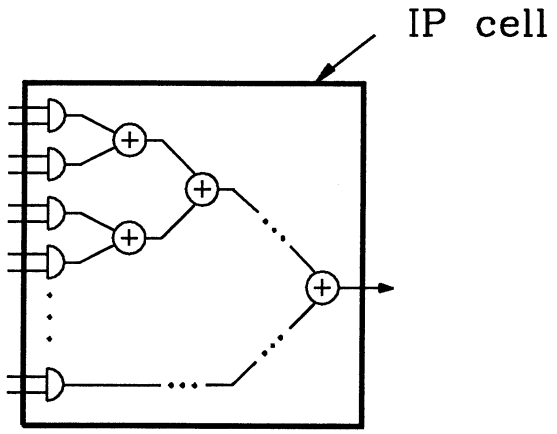
$$x \cdot A(x) = x^m a_{m-1} + x^{m-1} a_{m-2} + \cdots + x^2 a_1 + x a_0 =$$

$$a_{m-1}(x^{m-1} p_{m-1} + \cdots + x p_1 + 1) + x^{m-1} a_{m-2} + \cdots + x^2 a_1 + x a_0 =$$

$$a_{m-1} + \sum_{i=1}^{m-1} x^i (a_{m-1} p_i + a_{i-1}) \bmod P(x). \tag{4.13}$$



(a)



(b)

FIGURE 4.1 The general structure of the parallel PB multiplier over GF(2^m), (a), and the general IP cell (b).

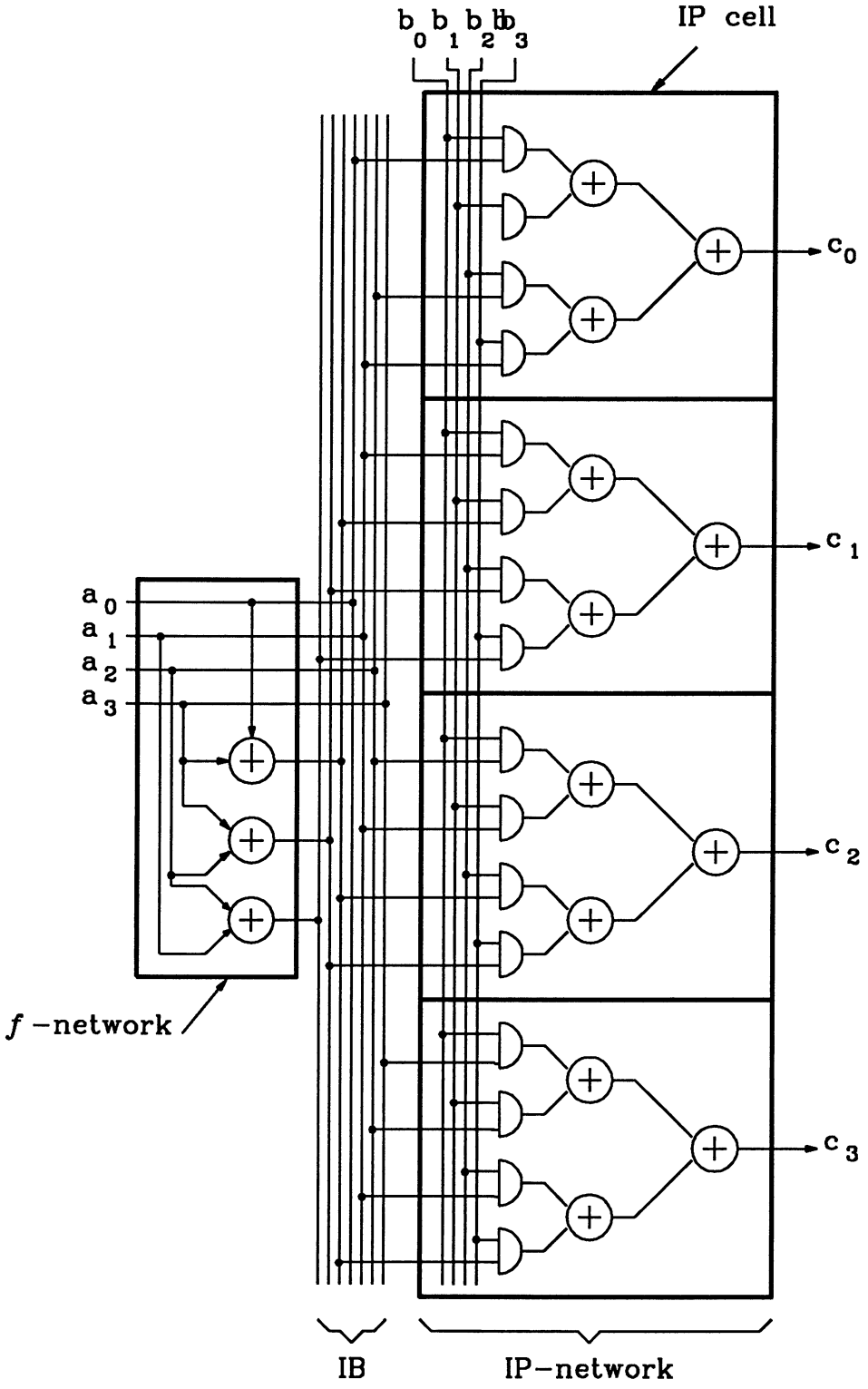


FIGURE 4.2 A parallel PB multiplier over GF(16) with $P(x)=1+x+x^4$.

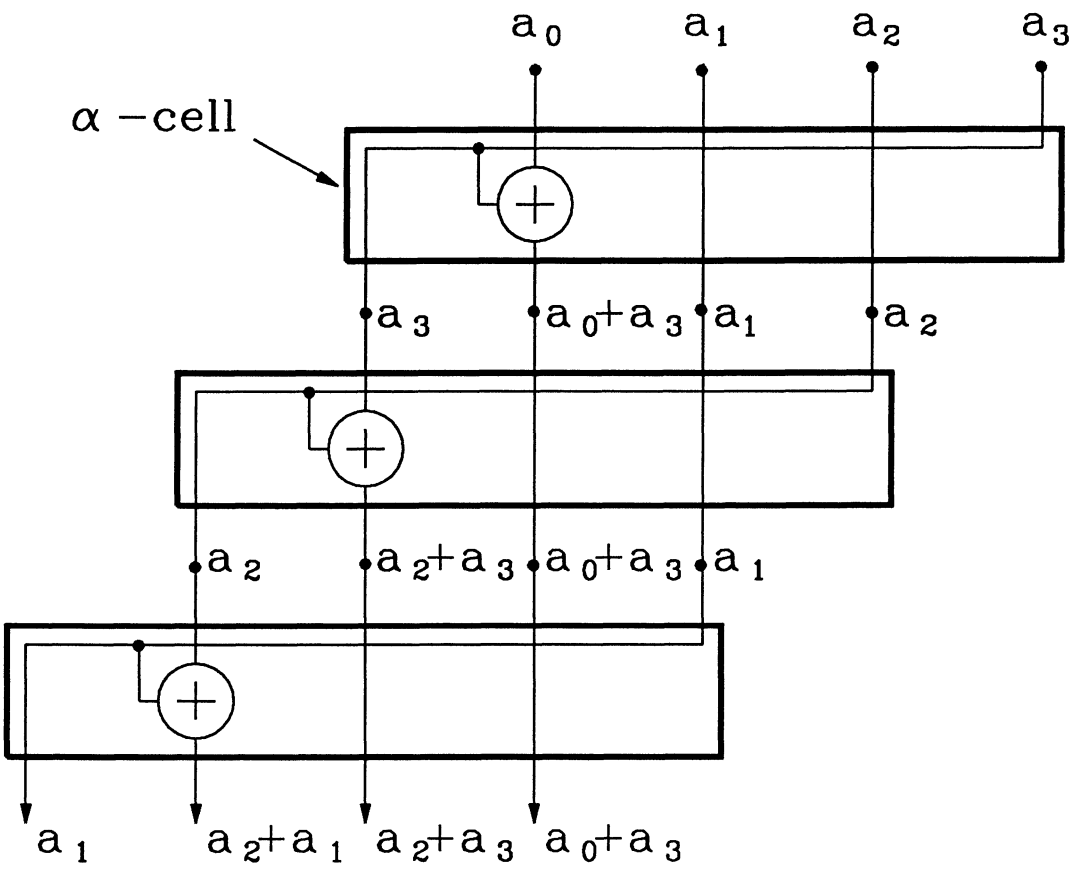


FIGURE 4.3 The α−array for $P(x)=1+x+x^4$

Let $P(x)$ have Hamming weight w_P. Since $p_m = p_0 = 1$ for binary prime polynomials, there are exactly $w_P - 2$ nonzero p_i for $i \in [1, 2, ..., m-1]$. Then by eq. (4.13) we can generate the second column ($j = 1$) of Z using a circuit consisting of $w_P - 2$ XOR gates. We call this circuit the *α-cell*. The third column ($j = 2$) can be generated by connecting the output of the α-cell to a second α-cell. Hence, we can generate Z by cascading $m-1$ α-cells. We call such a cascade the *α-array*. The α-array has complexity $(m-1)(w_P - 2)$. Fig. 4.3 shows the α-array of Ex. 4.1.

We have not found a closed-form expression for the length of the CP through an arbitrary α-array. However, this can be easily determined with the help of a computer. We wrote therefore a very short (less than one page) Pascal program which actually computes more than the length of the CP. It computes what we call the *profile* of the α-array. The profile is an m-dimensional vector $\mathcal{P} = (\lambda_0, \lambda_1, ..., \lambda_{m-1})$ where λ_i is the length of the CP between any input and the i:th output of the α-array. Clearly, we have $0 \le \lambda_i \le m-1 \ \forall \ i$. For example, the α-array of Fig. 4.3 has $\mathcal{P} = (0, 1, 1, 1)$. The Pascal program is shown in Appendix C.

Later we will see that the regularity of the α-array implies, in general, a lower performance than with the irregular f-network. However, the α-array will prove useful in other contexts as the next section where we will derive complexity and performance bounds.

4.3 Bounds on Complexity and Performance

In the previous section we saw that the parallel PB multiplier consists of three subsystems. Among them the IP network is the only one being independent of $P(x)$. The f-network and the IB are highly dependent on the choice of field generator. The former depends on $P(x)$ directly through the functions $f_{i,j}$. The latter depends on the number of distinct $f_{i,j}$ that appear in Z since each distinct $f_{i,j}$ requires a line in the IB. Consequently, in order to characterize the complexity and the performance of the parallel PB multiplier we define the following parameters.

Definition 4.2 (a) The number of coefficients a_i in a function $f_{i,j}$ is called the *width* of $f_{i,j}$ and is denoted $WD_{i,j}$.

(b) The maximum number of coefficients a_i in any function $f_{i,j}$ is called the *maximum width* and is denoted WD_{max}. More formally

$$WD_{max} = \max_{i,j} WD_{i,j}. \square$$

Definition 4.3 The number of lines in the IB is called the *bus width* and is denoted by BW. This is the same as the number of distinct $f_{i,j}$ in Z. □

Definition 4.4 (a) The number of gates required to implement the *f*-network is called the *f-complexity* and is denoted by $\mathcal{C}_f$.

(b) The number of (XOR) gates along the CP through the *f*-network is called the *f-length* and is denoted by $\mathcal{L}_f$. □

Further, we denote the total complexity (in gates) by $\mathcal{C}$ and the length (also in gates) of the CP through the whole multiplier by $\mathcal{L}$.

Example 4.1 (Cont'd) From Fig. 4.2 we see that

$$1 \le WD_{i,j} \le 2 \quad \forall\, i,j \in [0, 1, 2, 3],\ BW = 7,$$

$$\mathcal{C}_f = 3, \quad \mathcal{L}_f = 1, \quad \mathcal{C} = 31, \quad \mathcal{L} = 4. \square$$

Lower and upper bounds on the above parameters will be useful later on for determining the quality of the selection criteria and the goodness of certain class of field generators. Furthermore, bounds are interesting in general since they tell us what can be achieved in the best and worst case.

Proposition 4.5

$$1 \le WD_{i,j} \le m \qquad \forall\, i,j \in [0, 1, \ldots, m-1].$$

Proof. Since each $f_{i,j}$ is a linear combination of a-coefficients and there are m of them, the upper bound is trivial. It remains to show that $WD_{i,j} > 0$. Suppose that $WD_{i,j} = 0$. Then $f_{i,j} = 0$ and by eq. (4.2) this means that the i:th product coefficient c_i does not depend on b_i — that is, changing B does not change the product $C = AB$. Since A is an arbitrary field element, this contradicts the operation of a Galois-field multiplier. Hence, $WD_{i,j} > 0$. □

Proposition 4.6 Let $P(x)$ be a binary prime polynomial of Hamming weight w_P and degree $m \ge 2$. Then

$$m+1 \le BW \le m + (m-1)(w_P-2) \le m + (m-1)^2.$$

Proof. The lower bound. By eq. (4.6a) we know that $BW \ge m$. To complete the proof we need to show that there is at least one $f_{i,j}$ with $WD_{i,j} > 1$. By eq. (4.13) we know that the second column of Z is given in polynomial form by

$$a_{m-1} + \sum_{i=1}^{m-1} x^i(a_{m-1}p_i + a_{i-1})$$

Since w_P must be odd and at least 3 there is at least one $p_i \neq 0$ for $i \in [1, 2, \ldots, m-1]$. The associated term in the above sum is $a_{m-1} + a_{i-1}$ and equals $f_{i,1}$. Hence, $WD_{i,1} = 2$.

The upper bounds. Suppose that Z is generated by the α-array. We saw in the previous section that the α-array consists of $(m-1)(w_P - 2)$ XOR gates. Each of those gates forms eventually a new function $f_{i,j}$. Hence, there are at most $(m-1)(w_P - 2)$ functions $f_{i,j}$ with $WD_{i,j} > 1$. The proof is completed by recalling that $w_P \le m + 1$. □

Corollary 4.7

$$1 \le \mathcal{L}_f \le \lceil \log_2 m \rceil.$$

Proof. The upper bound follows directly from the upper bound of Prop. 4.5. The lower bound follows from Prop. 4.6 where we showed that there is at least one $f_{i,j}$ with $WD_{i,j} > 1$. □

Corollary 4.8

$$1 \le \mathcal{C}_f \le (m-1)(w_P - 2) \le (m-1)^2.$$

Proof. Both bounds follow from Prop. 4.6. The lower bound holds because there is at least one $f_{i,j}$ with $WD_{i,j} > 1$. The upper bounds hold because Z can always be generated by an α-array and $w_P \le m + 1$. □

Corollary 4.9

$$2 + \lceil \log_2 m \rceil \le \mathcal{L} \le 1 + 2\lceil \log_2 m \rceil.$$

Proof. The CP through an IP cell has length $1 + \lceil \log_2 m \rceil$. The rest follows from Cor. 4.7. □

Corollary 4.10

$$1 + m(2m-1) \leq \mathcal{C} \leq (m-1)(w_P - 2) + m(2m-1) \leq 3m(m-1) + 1.$$

Proof. There are m IP cells where each cell consists of $2m-1$ gates. The rest follows from Cor. 4.8. □

When dealing with bounds it is interesting to find *tight* bounds — that is, bounds that can be met in at least one case. We can anticipate that, among the above bounds, the only one which we do not know to be tight in the above sense is the lower bound of Prop. 4.6 (and consequently the lower bounds of Cor. 4.8 and 4.10). In Sec. 4.5 we will see how close to this bound we have been able to come.

4.4 Criteria for Selection of Field Generator

In the previous section we saw that the complexity $\mathcal{C}$ of the parallel PB multiplier over GF(2^m) can vary between $\sim 2m^2$ and $\sim 3m^2$, the length $\mathcal{L}$ of the CP between $2 + \lceil \log_2 m \rceil$ and $1 + 2\lceil \log_2 m \rceil$ and the bus width BW between $\sim m$ and $\sim m^2$. Obviously, it is desirable to have field generators whose $\mathcal{C}$, $\mathcal{L}$ and BW lie simultaneously on their respective lower bound. If this is not feasible or difficult to achieve they should lie as close to the lower bound as possible.

One way to find the best field generator for each degree is to go through all prime polynomials of that degree, determine for each polynomial the associated product matrix Z, $\mathcal{C}$ and $\mathcal{L}$ and keep the best one(s). However, in Sec. 2.5 we saw that the number $N_2(m)$ of binary prime polynomials grows quite rapidly with m. For example, we saw in Tab. 2.1 that $N_2(16) = 4080$ and that $N_2(m+1) \approx 2N_2(m)$. It is therefore desirable to find simpler selection criteria than the one described above.

The following is a set of "ideal goals" for our selection criteria:

G1. Maximum performance ⇔ minimum $\mathcal{L}$ ⇔ minimum $\mathcal{L}_f$
G2. Minimum complexity ⇔ minimum $\mathcal{C}$ ⇔ minimum $\mathcal{C}_f$
G3. Minimum interconnectivity ⇔ minimum BW

where minimum means meeting the lower bounds of the previous section. We anticipate that, in many cases, we will not be able to reach any of the above goals but we will often come very close.

A first general criterion for selecting a good $P(x)$ follows directly from the upper bounds of Prop. 4.6 and Cor. 4.8:

C1. Choose a field generator $P(x)$ of minimum Hamming weight w_P.

Since $w_P = 3, 5, 7, 9, \ldots$ for binary prime polynomials of degree ≥ 2, a first interesting class of field generators is the class of binary trinomials. A prime trinomial $1 + x^k + x^m$, $1 \leq k \leq m-1$, does not exist for any m and k. In [Zie68] we found that a least one prime trinomial exists in ~ 55% of the cases $m \leq 1000$. There are thus enough many of them to make this class very interesting for our purposes. The bounds of Prop. 4.6 and Cor. 4.8 become, for $w_P = 3$,

$$m + 1 \leq BW \leq 2m - 1, \qquad 1 \leq C_f \leq m - 1. \tag{4.14}$$

If we think of the f-network implemented as an α-array we see that it is not obvious that we can find cases — that is, values of k — where we can do better than the above upper bounds. Also, the fact that $w_P = 3$ does not tell us much about the length L_f. In particular, L_f can still meet the upper bound of Cor. 4.7 as shown in the following example.

Example 4.2 Let $P(x) = 1 + x^3 + x^4$. Then

$$Z = \begin{pmatrix} a_0 & a_3 & a_3+a_2 & a_3+a_2+a_1 \\ a_1 & a_0 & a_3 & a_3+a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_3+a_2 & a_3+a_2+a_1 & a_3+a_2+a_1+a_0 \end{pmatrix}$$

which shows that $WD_{max} = 4$, $L_f = 2$ and $C_f = 4$ (due to symmetries in the functions $f_{i,j}$). However, if we implemented the f-network as an α-array we would get the profile $\mathcal{P} = (2, 1, 0, 3)$ whereby L_f would be 3. C_f would though be 3. □

The field generator in the above example has been chosen to indicate that the α-array may give lower performance than the irregular f-network but

also that the α-array may require less gates. Evidently, a low w_P does not guarantee high performance. Some additional criterion is thus required.

Recall that, in order to maximize the performance of the multiplier, we must minimize the maximum function width WD_{max}. A second selection criterion should therefore be:

C2. Choose a field generator of minimum WD_{max}.

The above criterion requires the explicit computation of all functions $f_{i,j}$ which is not very practical. The following proposition presents a result by which we can determine WD_{max} directly from Q.

We define the *diagonal* of the matrix Q to be the entries $q_{i,j}$ with $i = j$.

Proposition 4.11 Let Q be the reduction matrix associated with the field generator $P(x)$. Let further w_1 be the maximum column (Hamming) weight of Q when the right-most column has been set to zero, and w_2 the maximum column (Hamming) weight of Q when all entries *on* and *below* the diagonal have been set to zero. Then

$$WD_{max} = \max\{w_1, w_2 + 1\}.$$

Proof. From eq. (4.6a) we know that

$$f_{i,j}(A) = \sigma(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i}a_{m-1-t} \qquad i = 0, 1, \ldots, m-1, j = 1, \ldots, m-1.$$

Due to the step function we check separately the cases $i < j$ and $i \geq j$:

$i < j$. Here we have

$$f_{i,j}(A) = \sum_{t=0}^{j-1} q_{j-1-t,i}a_{m-1-t} \quad j = 1, \ldots, m-1, i < j.$$

The worst-case functions are obtained for $j = m-1$

$$f_{i,m-1}(A) = (a_1, a_2, \ldots, a_{m-1}) \cdot \begin{pmatrix} q_{0,i} \\ q_{1,i} \\ \vdots \\ q_{m-2,i} \end{pmatrix} \quad i = 0, 1, \ldots, m-2.$$

The vectors ($q_{0,i}, q_{1,i}, \ldots, q_{m-2,i}$) for $i = 0, 1, \ldots, m-2$ are just the first $m-1$ columns of Z which explains w_1.

$i \geq j$. Here we have

$$f_{i,j}(A) = a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i}\, a_{m-1-t} \quad j = 1, \ldots, m-1,\ i \geq j.$$

Since

$$i \geq j \quad \Rightarrow \quad j-1-t \leq i-1-t \leq i-1$$

we see that only vectors of q-coefficients lying above the diagonal of Q are involved in the above sum. These vectors are parts of Z's columns which explains w_2. The presence of the term a_{i-j} completes the proof. □

Corollary 4.12 Only the function $f_{m-1,m-1}$ may attain the maximum width m. The maximum is attained if and only if the right-most column of Z is an an all-ones vector.

Proof. From the proof of Prop. 4.11 we know that $WD_{i,j} \leq m-1$ for $i < j$. So i must be $\geq j$. Further, for $i \geq j$ the only case in which all a-coefficients are involved is for $i = j = m-1$ whereby

$$f_{m-1,m-1}(A) = a_0 + (a_1, a_2, \ldots a_{m-1}) \cdot \begin{pmatrix} q_{0,m-1} \\ q_{1,m-1} \\ \vdots \\ q_{m-2,m-1} \end{pmatrix}.$$

The maximum width is attained if and only if $q_{0,m-1} = q_{1,m-1} = \ldots = q_{m-1,m-1} = 1$. □

Prop. 4.11 provides the simple tool we need to determine L_f without explicit computation of the functions $f_{i,j}$.

Example 4.2 (Cont'd) We have

$$Q = \begin{pmatrix} 1\,0\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,1\,1 \end{pmatrix}$$

from which we obtain

$WD_{max} = \max\{3, 3+1\} = 4$. □

Example 4.3 Let $P(x) = 1 + x^2 + x^3 + x^5 + x^8$ be the generator of GF(2^8). Then

$$Q = \begin{pmatrix} 10110100 \\ 01011010 \\ 00101101 \\ 10100010 \\ 01010001 \\ 10011100 \\ 01001110 \end{pmatrix}$$

from which we obtain

$WD_{max} = \max\{4, 2+1\} = 4$

and thus $L_f = 2$. The α-array associated with the above field generator has profile $\mathcal{P} = (3, 2, 4, 4, 3, 4, 3, 3)$ which means $L_f = 4$. Hence, the irregular f-network is here 100% faster than the α-array. □

In the following we suggest a procedure for finding good polynomials out of a given set S of prime polynomials of degree m. The procedure is intended for those values of m for which no prime trinomials exist. The reason for this is that, in the next section, we are going to derive special selection criteria for prime trinomials which are much simpler than the ones derived here.

Selection Procedure

Step 1. Select the subset S_1 consisting of polynomials in S with minimum Hamming weight.

Step 2. IF S_1 is small THEN compute Z (by Prop. 4.1), L_f, C_f and BW for each polynomial in S_1 and select the best one(s) and stop, ELSE go to **step 3.**

Step 3. Compute for each $P(x) \in S_1$ the maximum width WD_{max} (by Prop. 4.11) and select the subset S_2 of polynomials of minimum WD_{max}.

Step 4. Compute Z (by Prop. 4.1), L_f, C_f and BW for each $P(x) \in S_2$ and select the best one(s). Stop. □

If no set of binary prime polynomials is available, we must perform a search. Even in this case the criteria of this section turn useful. The search should be organized as follows:

Search Procedure

I. Set $w_P = 5$ and the threshold $WD_T = m$.

II. Start searching for prime polynomials of weight w_P. There are

$$(m-1)(m-2)\cdots(m-w_P+2) = \frac{(m-1)!}{(m-w_P+1)!}$$

polynomials of degree m to be checked. Start searching from the polynomial $1 + x + x^2 + \ldots + x^{w_P-2} + x^m$.

III. Find a prime polynomial and compute its WD_{max} (by Prop. 4.11).

IV. If $WD_{max} < WD_T$ then store the polynomial and set $WD_T = WD_{max}$.

V. Repeat **III** and **IV** until WD_T has reached an acceptable low value, and then stop. If an acceptable low value is not reached, set $w_P = w_P + 2$ and go back to **II.** □

The "acceptable low value" in **V** is set in accordance with the performance requirements of the application at hand, the time one is willing to wait before the search is stopped. Also, one has to keep in mind the following lower bound on WD_{max}.

Corollary 4.14

$$WD_{max} \geq 2.$$

Proof. From the proof of Prop. 4.6 we know that there is at least one $f_{i,j}$ with $WD_{i,j} > 1$. □

4.5 Trinomials and Equally Spaced Polynomials

In this section we analyze two special classes of polynomials. The previously mentioned trinomials $1 + x^k + x^m$ and the equally spaced polynomials [Ito89].

Definition 4.14 A polynomial of degree ns of the form

$$1 + x^s + x^{2s} + \cdots + x^{(n-1)s} + x^{ns}$$

is called an *equally spaced polynomial with spacing s* and is abbreviated *s*-ESP. □

The two classes do actually overlap. The overlap is represented by the trinomials $1 + x^{m/2} + x^m$ since these are also *s*-ESP with $s = m/2$.

From the definition of *s*-ESP we see that $s \in [1, 2, \ldots, m/2]$. A 1-ESP is also called an all-ones polynomial.

Proposition 4.15 Let $P(x) = 1 + x + x^m$ be the generator of GF(2^m). Then

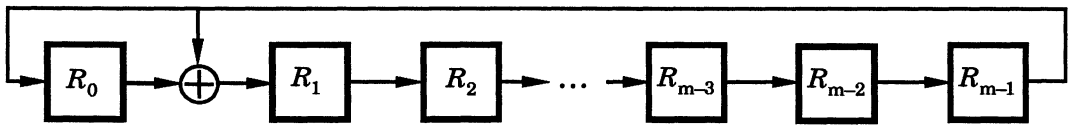
$$BW = 2m - 1,\ C_f = m - 1,\ C = 2m^2 - 1,\ WD_{max} = 2,\ L_f = 1,\ L = 2 + \lceil \log_2 m \rceil$$

and the associated product matrix Z has the following form

$$Z = \begin{pmatrix} a_0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ a_1 & a_0+a_{m-1} & a_{m-1}+a_{m-2} & \cdots & a_3+a_2 & a_2+a_1 \\ a_2 & a_1 & a_0+a_{m-1} & \cdots & a_4+a_3 & a_3+a_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0+a_{m-1} & a_{m-1}+a_{m-2} \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0+a_{m-1} \end{pmatrix}.$$

Proof. The proposition can be given two different prooves. The first one is an analytical proof based on the technique used to prove Prop. 4.1. The second proof is based on a direct investigation of the content of an LFSR with feedback polynomial $P(x)$ that has been initialized by the field element A. We provide here the latter proof and give the former in Appendix D.

The columns of Z can be generated by the following device.



The above LFSR is initially loaded with A. In the following table we show the first m states of the LFSR — that is, the contents of all registers R_i, $i = 0, 1, \ldots, m-1$, during the first m clock cycles.

R_0	R_1	R_2	...	R_{m-3}	R_{m-2}	R_{m-1}
a_0	a_1	a_2	...	a_{m-3}	a_{m-2}	a_{m-1}
a_{m-1}	a_0+a_{m-1}	a_1	...	a_{m-4}	a_{m-3}	a_{m-2}
a_{m-2}	$a_{m-1}+a_{m-2}$	a_0+a_{m-1}	...	a_{m-5}	a_{m-4}	a_{m-3}
a_{m-3}	$a_{m-2}+a_{m-3}$	$a_{m-1}+a_{m-2}$	...	a_{m-6}	a_{m-5}	a_{m-4}
:	:	:	...	:	:	:
a_2	a_3+a_2	a_4+a_3	...	$a_{m-1}+a_{m-2}$	a_0+a_{m-1}	a_1
a_1	a_2+a_1	a_3+a_2	...	$a_{m-2}+a_{m-3}$	$a_{m-1}+a_{m-2}$	a_0+a_{m-1}

The i:th row in the above table corresponds to the i:th column in the matrix Z. We see that there are $m-1$ distinct functions of width 2 and no functions of width > 2. The functions of width 2 can be read out from the last row and are

$$a_2 + a_1 \, , a_3 + a_2 \, , a_4 + a_3 \, , \ldots \, , a_{m-1} + a_{m-2} \, , a_0 + a_{m-1} . \, \square$$

Most trinomials $1 + x + x^m$ are not prime. However, among the prime ones, several are of practical interest. The following is a complete list for degrees up to 900 [Zie68]:

$m = 2, 3, 4, 6, 7, 9, 15, 22, 28, 30, 46, 60, 63, 127, 153, 172, 303, 471, 532, 865, 900.$

How about trinomials $1 + x^k + x^m$ with $k > 1$? We know by eq. (4.14) that $BW \leq 2m-1$ and $C_f \leq m-1$ for any k. What remains to find out is how WD_{max} varies with k. We investigated many trinomials of degree ≤ 30 with the help of a Pascal program that computes Q and Z. We state the results informally in Tab. 4.2 since we are not going to provide strict, formal prooves, rather,

we will indicate how the statements can be motivated with the help of previous results. The trinomials for $k = m/2$ are excluded here since these will be treated separately.

R1. $WD_{max} = 3$ for $2 \le k < m/2$.

R2. $3 \le WD_{max} \le m$ for $m/2 < k \le m-1$. (In particular, WD_{max} increases monotonically with k.)

R3. $WD_{max} = m$ for $k = m-1$.

R4. $BW = 2m-1$ for $k \ne m/2$.

R5. $C_f = m-1$ for $2 \le k < m/2$.

Table 4.2 Some experimental results for trinomials with $k > 1$, $k \ne m/2$.

R1 is motivated by realizing (with some effort) that Q's columns contain at most one 1 above the diagonal, at most two 1's on and below it and that there is at least one column of weight 3 (all this for k in the given range). Prop. 4.11 does the rest.

R2 is easy if we can show that $WD_{max} > 2$ for $k > m/2$. By Prop. 4.11 this is the same as saying that Q has at least one column with two or more 1's above the diagonal or as saying that there is at least one column of weight 3. Either of these two facts requires a certain effort to realize.

R3 follows from realizing (with little effort) that Q's last column is an all-ones vector for $k = m$ and recalling Cor. 4.12. It is about as easy to realize that this must be the only class of polynomials yielding $WD_{max} = m$.

R4-R5 are purely experimental results. (They could be motivated by an analysis of the generic α-array for $k < m/2$ but it appears to be tedious.)

From Prop. 4.15 and R1-R5 we derive the following selection criterion for the class of trinomials with $k \ne m/2$.

C3. Choose a prime trinomial $1 + x^k + x^m$ with minimum k.

We recall that the reciprocal of a prime polynomial of order e is also a prime polynomial of the same order. Consequently, we are not excluding any possibility by applying C3.

An investigation of the profile $\mathcal{P}$ of the α-arrays of many prime trinomials with $1 < k < m/2$ has indicated that the irregular f-network has the same L_f

as the corresponding α-array. Hence, for these trinomials one can as well use the α-array and further simplify the design work (we knew already that this is true for $k = 1$ by Prop. 4.15).

One of the tables given in [Zie68] is a complete list of prime trinomials of degree ≤ 1000 that can be used to check wether a prime trinomial with $m \leq 1000$ exists and, if so, to get the minimum value of k for that m.

We turn now to the s-ESP's and start with some formal results for the subclasses $s = m/2$ and $s = 1$.

The following proposition presents the only class of field generators with $BW < 2m-1$ that we have been able to find.

Proposition 4.16 Let $m = 2 \cdot 3^k$, $k = 0, 1, 2, \ldots$. Let further $P(x) = 1 + x^{m/2} + x^m$ be the generator of GF(2^m). Then

$$BW = \frac{3}{2}m,\ C_f = \frac{m}{2},\ C = 2m^2 - \frac{m}{2},\ WD_{max} = 2,\ L_f = 1,\ L = 2 + \lceil \log_2 m \rceil.$$

Proof. The irreducibility of the trinomial is guarantied by Th. 2.26. The reduction matrix is

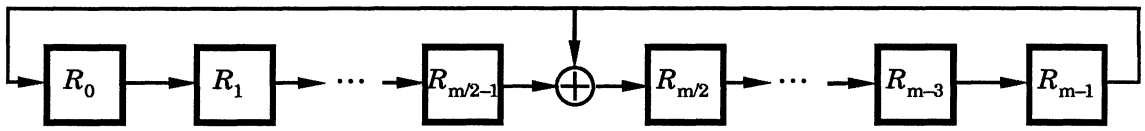
$$Q = \begin{pmatrix} 1\,0\,0 \ldots 0\,0\,1\,0\,0 \ldots 0\,0 \\ 0\,1\,0 \ldots 0\,0\,0\,1\,0 \ldots 0\,0 \\ 0\,0\,1 \ldots 0\,0\,0\,0\,1 \ldots 0\,0 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ 0\,0\,0 \ldots 0\,1\,0\,0\,0 \ldots 0\,1 \\ 1\,0\,0 \ldots 0\,0\,0\,0\,0 \ldots 0\,0 \\ 0\,1\,0 \ldots 0\,0\,0\,0\,0 \ldots 0\,0 \\ \vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ 0\,0\,0 \ldots 1\,0\,0\,0\,0 \ldots 0\,0 \end{pmatrix}.$$

The maximum width WD_{max} is, by Prop. 4.11,

$$WD_{max} = \max\{2, 1+1\} = 2$$

which is best possible and implies $L_f = 1$.

In order to determine BW we apply the same technique as in the proof of Prop. 4.15. We study thus the states of the following device.



R_0	R_1	...	$R_{m/2-1}$	$R_{m/2}$	...	R_{m-2}	R_{m-1}
a_0	a_1	...	$a_{m/2-1}$	$a_{m/2}$	...	a_{m-2}	a_{m-1}
a_{m-1}	a_0	...	$a_{m/2-2}$	$a_{m/2-1}+a_{m-1}$	...	a_{m-3}	a_{m-2}
a_{m-2}	a_{m-1}	...	$a_{m/2-3}$	$a_{m/2-2}+a_{m-2}$	...	a_{m-4}	a_{m-3}
:	:	...	:	:	...	:	:
$a_{m/2+1}$	$a_{m/2+2}$	...	a_0	$a_1+a_{m/2+1}$	...	$a_{m/2-1}+a_{m-1}$	$a_{m/2}$
$a_{m/2}$	$a_{m/2+1}$	...	a_{m-1}	$a_0+a_{m/2}$	...	$a_{m/2-2}+a_{m-2}$	$a_{m/2-1}+a_{m-1}$
$a_{m/2-1}+a_{m-1}$	$a_{m/2}$	...	a_{m-2}	$a_{m/2-1}$	...	$a_{m/2-3}+a_{m-3}$	$a_{m/2-2}+a_{m-2}$
$a_{m/2-2}+a_{m-2}$	$a_{m/2-1}+a_{m-1}$	...	a_{m-3}	$a_{m/2-2}$	...	$a_{m/2-4}+a_{m-4}$	$a_{m/2-3}+a_{m-3}$
:	:	...	:	:	...	:	:
$a_2+a_{m/2+2}$	$a_3+a_{m/2+3}$	...	$a_{m/2+1}$	a_2	...	$a_0+a_{m/2}$	$a_1+a_{m/2+1}$
$a_1+a_{m/2+1}$	$a_2+a_{m/2+2}$	...	$a_{m/2}$	a_1	...	$a_{m/2-1}$	$a_0+a_{m/2}$

We call R_0 through $R_{m/2-1}$ the *left half*, and $R_{m/2}$ through R_{m-1} the *right half* of the above LFSR. We see that the LFSR, during the first $m/2+1$ cycles, generates functions of width 2 only in the right half. These $m/2$ functions are all distinct and can be read out from the upper half of the column for $R_{m/2}$. At cycle # $m/2+2$ (which corresponds to the row in the box) the feedback signal $a_{m/2-1}+a_{m-1}$ is added to a_{m-1} (the content of $R_{m/2-1}$) to form the signal $a_{m/2-1}$ which becomes the new content of $R_{m/2}$. From now on $R_{m/2}$ will contain only functions of width 1 (due to cancellation of equal terms at the XOR gate) and no new functions of width > 1 will be generated during the remaining cycles. Hence, there are only $m/2$ distinct functions of width 2 which implies $BW = \frac{3}{2}m$ and $C_f = m/2$. The functions are

$$a_0+a_{m/2}, a_1+a_{m/2+1}, a_2+a_{m/2+2}, \ldots, a_{m/2-2}+a_{m-2}, a_{m/2-1}+a_{m-1}. \square$$

The trinomial $1 + x^{m/2} + x^m$ is rarely prime. By Th. 2.26 we know that the first seven prime ones are

$$m = 2, 6, 18, 54, 162, 486, 1458.$$

Proposition 4.17 Let $P(x) = 1 + x + x^2 + \cdots + x^{m-1} + x^m$ be the generator of GF(2^m). Then

$$BW = \frac{m^2 + m}{2},\ \mathcal{C}_f = \frac{m^2 - m}{2},\ \mathcal{C} = \frac{5m^2 - 3m}{2},$$

$$WD_{max} = 2,\ \mathcal{L}_f = 1,\ \mathcal{L} = 2 + \lceil \log_2 m \rceil.$$

Proof. The reduction matrix is

$$Q = \begin{pmatrix} 1\,1\,1\,1\,...1\,1\,1\,1 \\ 1\,0\,0\,0\,...\,0\,0\,0\,0 \\ 0\,1\,0\,0\,...\,0\,0\,0\,0 \\ :\quad\quad :\quad\quad : \\ 0\,0\,0\,0\,...\,1\,0\,0\,0 \\ 0\,0\,0\,0\,...\,0\,1\,0\,0 \end{pmatrix}.$$

The maximum width WD_{max} is, by Prop. 4.11,

$$WD_{max} = \max\{2, 1+1\} = 2$$

which is best possible and implies $\mathcal{L}_f = 1$.

In order to determine BW we apply the same technique as in the proof of Prop. 4.1 (in particular, we proceed exactly as in the alternative proof of Prop. 4.15 given in Appendix D).

From Q we see that

$$q_{0,j} = 1 \quad \forall j$$

$$q_{i,j} = 1 \quad j = 0, 1, \ldots, m-3,\ i = j + 1.$$

By eq. (4.10) and the above conditions on the $q_{i,j}$ we have

$$c_i = s_i + \sum_{k=0}^{m-2} q_{k,i} s_{k+m} = s_i + s_m + \sigma(m-3-i) s_{m+i+1}$$

where $\sigma(k)$ is the step function.

With the help of the above expression for c_i and eq. (4.4) we can create a table similar to Tab. 4.1 for each c_i and identify all products $b_u a_v$ involved there. This is done in Tab. 4.3.

We proceed to determine the number of distinct functions $f_{i,j}$ of width 2 in the Tab. 4.3. To this end we notice the following fact.

In each c_i, $i = 0, 1, \ldots, m-3$, the functions generated by the term $s_m + s_{m+i+1}$ are also generated in c_{m-i-1} by the term $s_m + s_{m-i-1}$. For example, the term $s_m + s_{m+1}$ of c_0 generates the functions

$$a_{m-1} + a_{m-2}, a_{m-2} + a_{m-3}, a_{m-3} + a_{m-4}, \ldots, a_2 + a_1$$

which are also generated by the term $s_m + s_{m-1}$ of c_{m-1}. As another example, the term $s_m + s_{2m-2}$ of c_{m-3} generates the function $a_{m-1} + a_1$ which is also generated by the term $s_m + s_2$ of c_2.

The above fact implies that, for each c_i we need only consider the functions originating from the term $s_i + s_m$. We see from the table that all such functions are distinct and that there are exactly i of them in each c_i. This, in turn, implies that the number of distinct functions $f_{i,j}$ of width 2 (= C_f here) is given by the following arithmetic series

$$\sum_{k=0}^{m-1} k = \frac{(m-1)m}{2}. \square$$

		u:	0	1	2	3	4	...	$m-3$	$m-2$	$m-1$
c_0	s_0	v:	0								
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
	s_{m+1}	v:			$m-1$	$m-2$	$m-3$	...	4	3	2
c_1	s_1	v:	1	0							
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
	s_{m+2}	v:				$m-1$	$m-2$	...	5	4	3
c_2	s_2	v:	2	1	0						
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
	s_{m+3}	v:					$m-1$	...	6	5	4
	:	:	:	:	:	:	:	:	:	:	:
c_{m-3}	s_{m-3}	v:	$m-3$	$m-4$	$m-5$	$m-6$	$m-7$	...	0		
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
	s_{2m-2}	v:									$m-1$
c_{m-2}	s_{m-2}	v:	$m-2$	$m-3$	$m-4$	$m-5$	$m-6$	...	1	0	
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
c_{m-1}	s_{m-1}	v:	$m-1$	$m-2$	$m-3$	$m-4$	$m-5$	...	2	1	0
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1

Table 4.3

In Prop. 4.6, Cor. 4.8 and Cor. 4.10 we introduced three upper bounds as functions of the Hamming weight w_P. There we also indicated their "worst-case" form for $w_P = m + 1$. By the above result we see that $w_P = m + 1$ is not the true worst case. This fact allows us to refine the mentioned upper bounds. As worst case we can now consider $w_P = m$ (for even m we could take $w_P = m - 1$) which yields the following new "worst-case" upper bounds

$$BW \le m + (m-1)(m-2) \tag{4.15}$$

$$\mathcal{C}_f \le (m-1)(m-2) \tag{4.16}$$

$$\mathcal{C} \le (m-1)(m-2) + m(2m-1). \tag{4.17}$$

The new bounds are meaningful for $m \ge 8$.

We turn now to the arbitrary s-ESP.

Proposition 4.18 Let $P(x)$ be an s-ESP of degree $m = ns$ generating GF(2^m). Then

$$WD_{max} = 2, \mathcal{L}_f = 1, \mathcal{L} = 2 + \lceil \log_2 m \rceil.$$

Proof. It is sufficient to show that $WD_{max} = 2$. To this end we notice that

$$x^m = 1 + x^s + \cdots + x^{(n-2)s} + x^{(n-1)s}$$

$$x^{m+1} = x + x^{s+1} + \cdots + x^{(n-2)s+1} + x^{(n-1)s+1}$$

$$\vdots$$

$$x^{m+s} = x^s + x^{2s} + \cdots + x^{(n-1)s} + x^m = 1$$

$$x^{m+s+1} = x$$

$$\vdots$$

$$x^{2m-2} = x^{m-s-2}$$

which means that Q has the following form (0^j denotes a sequence of j zeros)

$$Q = \begin{pmatrix} 1\,0^{s-1}\,1\,0^{s-1}\,1 & \ldots 1\,0^{s-1} \\ 0\,1\,0^{s-1}\,1\,0^{s-1} & \ldots 0\,1\,0^{s-2} \\ \vdots \quad\quad \vdots & \quad\quad \vdots \\ 0^{s-1}\,1\,0^{s-1}\,1\,0^{s-1} & \ldots 0^{s-1}\,1 \\ 1\,0^{m-1} & \\ 0\,1\,0^{m-2} & \\ \vdots & \\ 0^{m-s-2}\,1\,0^{s+1} & \end{pmatrix}.$$

By Prop. 4.11 we have finally

$$WD_{max} = \max\{2, 1+1\} = 2. \square$$

We have established that s-ESP's are a class of maximum-performance polynomials. Their complexity depends however on s as we could see for the subclasses of 1-ESP's and $m/2$-ESP's. For the arbitrary s-ESP we have the following conjecture.

Conjecture 4.18 Let $P(x)$ be an s-ESP generating GF(2^m). Then

$$BW = m + \frac{m(m-s)}{2s},\ C_f = \frac{m(m-s)}{2s},\ C = \frac{4s+1}{2s}m^2 - \frac{3}{2}m. \square$$

The above expressions have been obtained from the analysis of the structure of the matrix Z for a limited number of s-ESP's with $s \neq 1, m/2$. The analysis was performed with the help of the Pascal program computing Q and Z. A first check of the conjecture is to notice that it agrees with the results of Prop. 4.16 and 4.17 — i.e. when $s = 1, m/2$. This is a necessary but of course not sufficient condition.

In [Ito89] the authors derive some necessary and sufficient conditions for an s-ESP, $s > 1$, to be prime and present a table of values for m and s that we reproduce in Tab. 4.4. For $s = 1$ we saw already in Sec. 3.3 that a 1-ESP of degree ≤ 179 exists for

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

m	s	m	s	m	s	m	s
6	3*	110	11	500	125	2028	169
18	9*	156	13	812	29	2500	625
20	5	162	81*	1210	121	2756	53
54	27*	342	19	1332	37	3422	59
100	25	486	243*	1458	729*	3660	61

Table 4.4 Values of m and s for some prime s-ESP's, $s > 1$.
Trinomials are marked by *.

Evidently the class of s-ESP's is not very rich. This is though not the only drawback. In practice one often searches for primitive field generators. In this sense, the class of s-ESP's is of limited interest as indicated by the following result.

Proposition 4.19 Let $P(x)$ be a prime s-ESP of degree m. Then

$$\text{ord } P(x) = m + s.$$

Proof. By Th. 2.24 we know that the order of $P(x)$ is equal to the order of any of its roots. Let α be a root of $P(x)$ and consequently one of basis elements in the PB representation of GF(2^m) induced by $P(x)$. We seek the least positive integer r such that $\alpha^r = 1$. Since $\alpha^r \neq 1$ for $1 \leq r \leq m-1$ we check $r \geq m$. In the proof of Prop. 4.17 we saw that

$$x^m = 1 + x^s + \cdots + x^{(n-2)s} + x^{(n-1)s} \neq 1$$

$$x^{m+1} = x + x^{s+1} + \cdots + x^{(n-2)s+1} + x^{(n-1)s+1} \neq 1$$

$$\vdots$$

$$x^{m+s} = x^s + x^{2s} + \cdots + x^{(n-1)s} + x^m = 1.$$

The polynomial x represents just the element α. □

Since $m \geq 2$ we have that $m + s = 2^m - 1$ if and only if $m = 2$ and $s = 1$. Hence, the only primitive ESP is the trinomial $1 + x + x^2$.

4.6 An Exhaustive Search for $m \leq 16$

In this section we present the results of an exhaustive computer search through a data base consisting of all prime polynomials of degree 2 through

16 (there are 8798 of them). For each polynomial we let the computer determine WD_{max}, BW and C_f by explicit computations of all functions $f_{i,j}$. The parameter C_f was computed by the following formula

$$C_f = \sum_{i,j \in J} (WD_{i,j} - 1) \tag{4.18}$$

where J is the set of pairs (i,j) corresponding to the distinct $f_{i,j}$'s. This means that eventual residual symmetries in the $f_{i,j}$'s have not been used systematically to further reduce the complexity.

In a first selection we accepted all polynomials yielding $WD_{max} \leq 8$. For $m \leq 10$ we tabulated though the parameters of all polynomials. This table was used to verify the result of the second selection.

In the second selection we accepted only polynomials with minimum WD_{max}. These polynomials are shown in Tab. 4.5.

m	$P(x)$	WD_{max}	L	$BW-m$	C_f	C
2	0,1,2	2	3	1	1	7
3	0,1,3	2	4	2	2	17
4	0,1,4	2	4	3	3	31
5	0,2,5	3	6	4	5(4)	50(49)
6	0,3,6*	2	5	3	3	69
6	0,1,6	2	5	5	5	71
7	0,1,7	2	5	6	6	97
8	0,1,5,7,8*	4	6	15	34(20)	154(140)
8	0,2,3,5,8	4	6	21	38(28)	158(148)
9	0,1,9*	2	6	8	8	161
9	0,4,9	3	7	8	11(8)	164(161)
10	0,3,10	3	7	9	11(9)	201(199)
11	0,2,11	3	7	10	11(10)	242(241)
12	0,3,12*	3	7	11	13(11)	289(287)
12	0,1,5,8,12	6	8	33	75	351
13	0,1,6,7,13	4	7	36	68(46)	393(371)
14	0,5,14*	3	7	13	17(13)	395(391)
14	0,2,7,9,14	6	8	33	73	451
15	0,1,15	2	6	14	14	449
16	0,5,6,11,16	4	7	35	67(41)	563(537)

Table 4.5 The best field generators for $2 \leq m \leq 16$. For each polynomial we indicate the actual powers of x. Non-primitive polynomials are marked by *. For some polynomials we show, in brackets, the improved complexity figures obtained utilizing the residual symmetries in the functions $f_{i,j}$'s.

In Tab. 4.5 we have chosen to show the number $BW{-}m$ of distinct functions of width > 1 instead of BW.

The above results have led us to the following conjectures.

Conjecture 4.20 The only classes of polynomials yielding maximum performance — i.e. $WD_{max} = 2$ — are the class of trinomials $1 + x + x^m$ and the class of s-ESP's for any $s \in [1, 2, \ldots, m/2]$. □

Conjecture 4.21 No classes of prime polynomials exist for which the f-complexity C_f is less than $m/2$. □

Accordingly, the class of field generators yielding the best complexity, performance and interconnectivity should be the class of $m/2$-ESP's.

In Tab. 4.6 we show the parameters of the best and worst f-networks for the degrees 4 through 10 (C_f was computed by eq. (4.18)). For comparison we provide also the figures L_f and C_f of the worst-case α-arrays. Tab. 4.6 shows clearly the importance of choosing a good field generator.

m	L_f	$BW{-}m$	C_f	L/C_f
	Best/Worst	Best/Worst	Best/Worst	α-array (worst case)
4	1/2	3/6	3/6	3/9
5	2/2	4/12	5/19	4/12
6	1/3	3/15	3/29	5/15
7	1/3	6/30	6/62	5/30
8	2/3	15/35	34/89	5/35
9	1/4	8/49	8/122	8/56
10	2/4	9/63	11/191	7/63
11	**2/3**	10/**81**	11/**246**	10/90
12	**2/3**	11/**99**	13/**294**	11/99
13	**2/3**	**36/132**	68/**351**	**10/108**
14	**2/3**	**13/120**	17/**425**	**11/117**
15	**1/3**	14/**182**	14/**488**	14/**154**
16	**2/3**	**35/195**	67/**630**	15/195

Table 4.6 The parameters of the best/worst f-networks for $4 \le m \le 16$. For polynomials yielding worst complexity C_f we show also the parameters of the associated α-array. Boldfaced figures are lower bounds due to incomplete search.

4.7 Test of the Selection Criteria for $w_p > 3$

The best polynomials listed in Tab. 4.5 have been found by an exhaustive search through all prime polynomials of degree $2 \leq m \leq 16$. This gives us an opportunity to test the selection criteria C1 and C2 of Sec. 4.4. We tested the criteria for $m = 8, 13, 16$ since no irreducible trinomials exist for these degrees. The results are given in Tab. 4.7. The first column shows the total number of prime polynomials, the second column the number of polynomials selected by C1 whereas the third column shows the size of the final set of selected polynomials after both C1 and C2.

We see that the final set is very small. Also, in all cases the best polynomial of Tab. 4.5 is included in the final set (in two cases only the best polynomial is left !).

m	Total # of polynomials	After C1	After C1 & C2 (Final set)	Is best polynomial included in final set ?
8	30	17	3	Yes
13	630	67	1	Yes
16	4080	94	1	Yes

Table 4.7

4.8 Other Parallel Multipliers

Polynomial Basis. In [Law71] the authors describe a cellular-array multiplier which perform the same operations as the sequential SSR multiplier of Sec. 3. These operations, though, are performed iteratively in space rather than in time. The multiplier has a highly regular structure consisting of m^2 identical cells, each cell containing 4 gates which yields a complexity of $4m^2$ gates independently of $P(x)$. The major drawback of this multiplier is its speed of operation. With a critical-path length between $2m$ and $3m$ gates (see [Law71, Fig. 2]) the cellular-array multiplier loses much of its attractiveness.

A parallel systolic multiplier is presented in [Yeh84]. The multiplier has a regular structure consisting of m^2 identical cells, each cell containing 4 gates and 7 registers. Furthermore, a number of additional registers which we estimated (based on [Yeh84, Fig. 4]) to $\sim 3m^2$ is required for a proper

alignment of the input and output bits. All this sums up to a complexity of $\sim 14m^2$ and is the major drawback of this approach. The multiplier can achieve a very high operational speed *if* it is operated continuously thanks to a critical-path length of only 2. At single operation, on the other hand, the multiplier behaves as having a critical path of length between $2m$ and $3m$. The systolic multiplier is a clocked device.

In [Ito89] the authors introduce a structure for parallel PB multipliers which is very similar to the structure of parallel NB multipliers (to be discussed soon). The multiplier consists of a number > m of identical blocks and a few additional XOR gates ($\sim m$ of them) which combine the outputs of the blocks so to obtain the bits of the desired product (see [Ito89, Fig. 2 and 3]). The complexity formulas given in [Ito89, Sec. 4.3] are not directly comparable with ours. We discuss therefore the two examples given in [Ito89]. For GF(2^4) the proposed multiplier (see [Ito89, Fig. 2] requires 49 gates and has a CP of length 5. For GF(2^6) it requires (see [Ito89, Fig. 4]) 159 gates and has a CP of length 6. Comparing with the figures of Tab. 4.5 we see that this approach appears to be both more complex and slower than ours. The main advantage here is the improved regularity of the architecture. The major drawback, on the other hand, is that the proposed approach is viable only when the field generator is an s-ESP, $s \in [1, 2, \ldots, m/2]$. Consequently, recalling Prop. 4.19 we realize that the architecture proposed in [Ito89] is of limited practical interest.

Normal Basis. The Massey-Omura multiplier described in Sec. 3.3 is easily turned into a parallel multiplier. Instead of using two registers to produce the m cyclic shifts of the input elements and one logic function f to compute the output bits one at a time, one utilizes m identical blocks where each block implements the logic function f. Each of these blocks is then fed with a different cyclically shifted version of the input elements. The shifting is so-to-speak hardwired. Accordingly, the complexity is m times the complexity of one block and the performance is determined by the length of the block's CP. By eq. (3.26) and (3.27) we have the following lower bounds (we have no registers here)

$$\mathcal{C} \geq 3m^2 - 2m \tag{4.19}$$

$$\mathcal{L} \geq 2 + \lceil \log_2 m \rceil \tag{4.20}$$

with equality in the first equation only for optimal normal bases. For a generic logic function f with N_m terms we have the following formulas

$$C = m(N_m + m - 1) \tag{4.21}$$

$$L = 1 + \lceil \log_2 N_m \rceil. \tag{4.22}$$

With the help of Tab. 3.10 and the above expressions we can compile the following table for the best normal bases for $2 \le m \le 16$.

m	$P(x)$	N_m	C	L
2	0,1,2†	3	8	3
3	0,2,3†	5	21	4
4	0,1,2,3,4*†	7	40	4
4	0,3,4	9	48	5
5	0,1,2,4,5†	9	65	5
6	0,1,4,5,6†	11	96	5
7	0,2,5,6,7	19	175	6
8	0,3,5,7,8	21	224	6
9	0,1,4,5,6,8,9†	17	225	6
10	0,1,2,3,4,5,6,7,8,9,10*†	19	280	6
10	0,1,4,9,10	37	460	7
11	0,2,3,4,8,10,11†	21	341	6
12	0,1,2,3,4,5,6,7,8,9,10,11,12*†	23	408	6
12	0,1,2,5,6,7,8,11,12	41	624	7
13	0,3,4,7,10,12,13	45	741	7
14	0,1,8,9,12,13,14†	27	560	6
15	0,2,4,5,7,9,12,14,15*	45	885	7
15	0,5,7,9,12,14,15	53	1005	7
16	0,1,2,3,5,7,8,10,11,12,13,15,16	85	1600	8

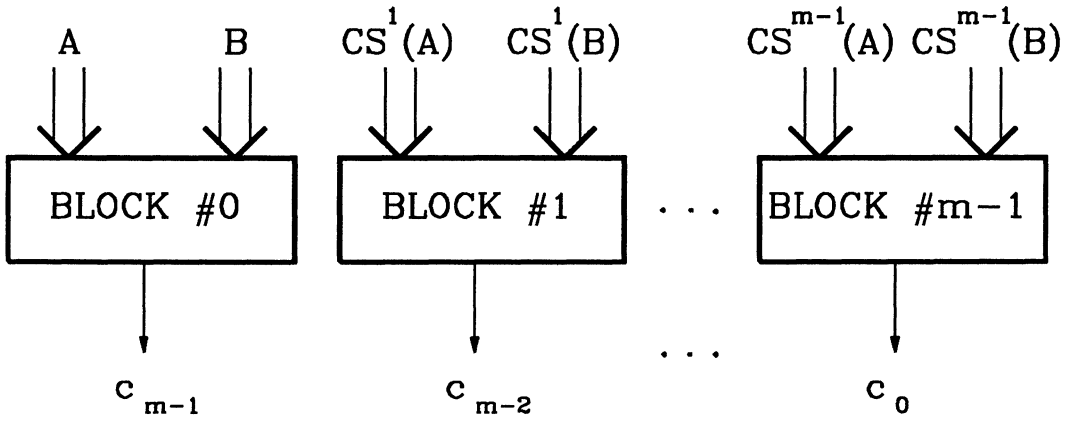
Table 4.8 Complexity and performance figures for MO multipliers constructed with prime polynomials of degree ≤ 16 for which N_m is minimized. Non-primitive polynomials are marked by *. Polynomials yielding an optimal NB are marked by †.

The complexity figures of Tab. 4.8 presuppose an implementation of the logic function f according to eq. (3.25), i.e.

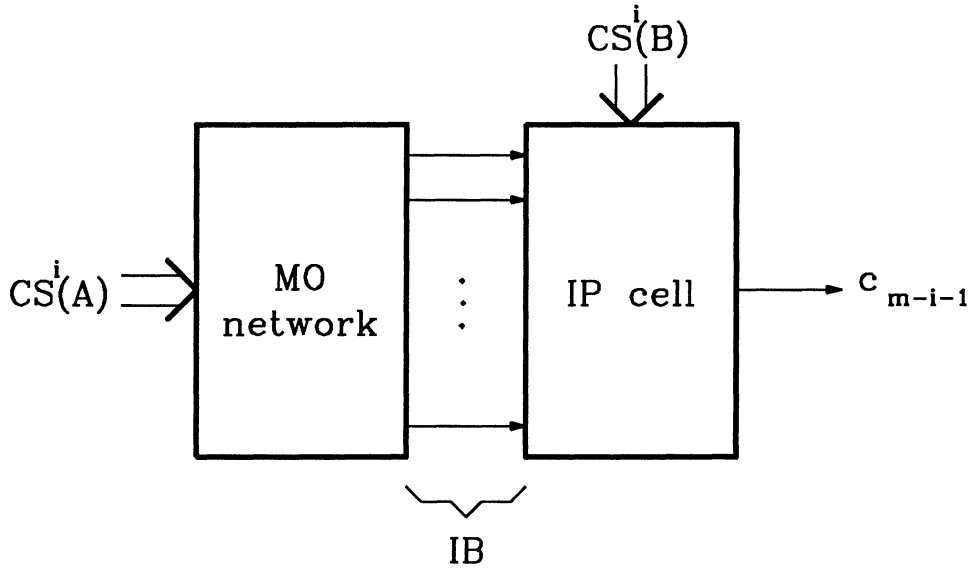
$$f(A,B) = AMB^t.$$

Fig. 4.4a shows the general structure of the MO multiplier while Fig. 4.4b shows the general structure of one of the blocks of Fig. 4.4a implemented according to eq. (3.25). Each block has a subdivision similar to our PB

$CS^i(A)$ = i:th cyclic shift of A.



(a)



(b)

FIGURE 4.4 The general structure of the parallel MO multiplier, (a), and the block #i, (b).

multiplier. It consists thus of an IP cell, an IB and a network of XOR gates (similar to our *f*-network) which we call the *MO network*. The IB has fixed $BW = m$. The MO network is the only part that depends on the field generator and consist of $N_m - m \geq m - 1$ XOR gates (equality for optimal NB).

4.9 Discussion

Our novel treatment has shown that the early PB multiplier of [Bar63] has considerably better properties than it has been believed. In particular, we have demonstrated that, by a careful choice of the field generator $P(x)$, the parallel PB multiplier can provide both low complexity and high performance.

In Tab. 4.9 we summarize the results of our investigation.

$P(x)$	C	L	$BW - m$
$1 + x + x^m$	$2m^2 - 1$	$2 + \lceil \log_2 m \rceil$	$m - 1$
$1 + x^k + x^m$ $1 < k < m/2$	$2m^2 - 1$	$3 + \lceil \log_2 m \rceil$	$m - 1$
s-ESP $1 \leq s \leq m/2$	$\frac{4s+1}{2s} m^2 - \frac{3}{2} m$	$2 + \lceil \log_2 m \rceil$	$\frac{m(m-s)}{2s}$
Pentanomial ($\neq$ ESP)	$\leq 2m^2 + 2m - 3$	$d + \lceil \log_2 m \rceil, d \geq 3$	$\leq 3(m-1)$
Generic prime ($\neq$ ESP) $w_P > 5$	$\leq 2m^2 - m + (w_P - 2)(m-1)$	$d + \lceil \log_2 m \rceil, d \geq 3$	$\leq (w_P - 2)(m-1)$

Table 4.9 Properties of the PB multiplier for different classes of field generators. C is the complexity in gates, L is the length of the critical path and $BW-m$ is the number of additional signals to be routed (i.e. the $2m$ signals of the multiplicands excluded).

Prime trinomials exist for 55% of the degrees 2 through 1000 [Zie68] and can thus be considered a common type of field generator. Prime trinomials or pentanomials exist for any degree ≤ 34 [Pet61]. For degrees > 34 for which no prime trinomials exist, it is therefore reasonable to assume that a prime pentanomial will exist with very high probability. This means that in most

cases one will be able to choose a field generator which is either a trinomial or a pentanomial. Accordingly, we can say that the complexity of the PB multiplier is ~ $2m^2$. This is ~ 50% less than the cellular-array multiplier in [Law71], ~ 85% less than the systolic multiplier in [Yeh84] and ~ 30% less than the MO multiplier in its optimal form (optimal NB).

Fig. 4.5 shows the lower bounds on the PB (Cor. 4.10) and NB (eq. (4.19)) complexities. Notice that the NB lower bound can be seen as an upper bound for PB (compare with the second upper bound of Cor. 4.10). Fig. 4.6 shows the complexities for the best PB/NB field generators of degree ≤ 16.

The performance of the PB and NB multipliers appears to be very similar, generally speaking. Depending on the actual value of m, a difference of at most two units, in either direction, in critical-path length seems to be typical. Both these multipliers are considerably faster than the cellular-array multiplier, and considerably slower, at continuous operation, than the systolic multiplier. However, by pipelining we can increase the speed of the PB/NB multiplier by introducing a certain amount of additional registers. We describe the technique by an example.

Suppose we designed a PB multiplier for $m = 8$ with a CP of length $L = 6$. The idea is to split the multiplier in r sections of equal speed. The number r will be determined by the specific requirements of the application at hand. We set $r = 2$. Since $L = 6$ we might suppose that an optimal subdivision is to split the multiplier into two sections, each with $L = 3$. The optimality of this subdivision should however be checked by simulations of the hardware. Then we introduce one register after each AND gate in the IP cells, as shown in Fig. 4.7a, and obtain a PB multiplier that can be operated twice as fast. The additional complexity is m^2 which yields a multiplier of total complexity ~ $3m^2$. Notice, though, that we get full advantage of the new performance only at continuous operation. For single multiplications the multiplier behaves like the original one, as one multiplication takes two clock cycles.

If the simulations show that an optimal subdivision is to have the registers after the f-network we need only $BW + m$ registers, i.e. typically between $3m - 1$ (trinomials) and $5m - 3$ (pentanomials) registers instead of m^2. See Fig. 4.7b.

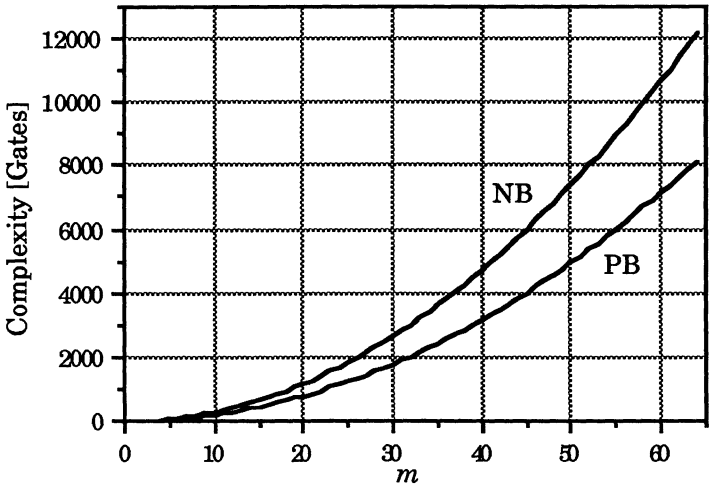


Figure 4.5 Lower bounds for parallel PB/NB multipliers.

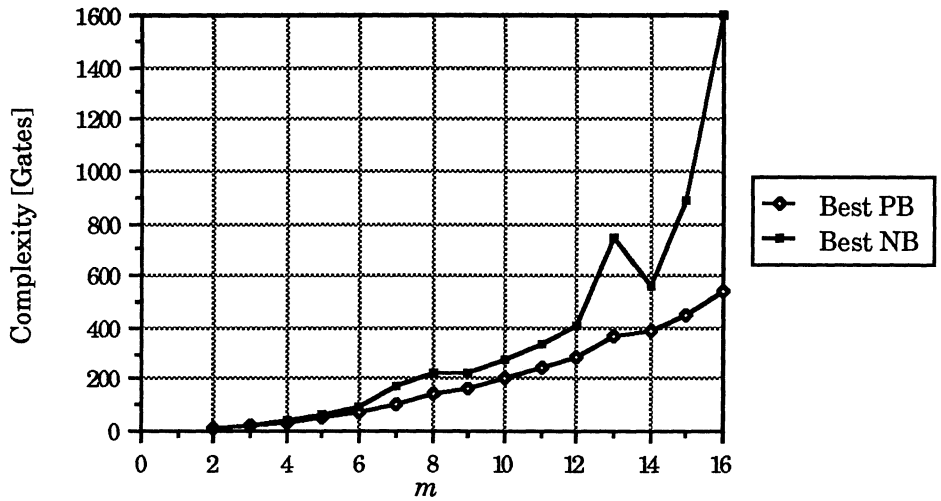


Figure 4.6 The best PB/NB complexities for $m \leq 16$.

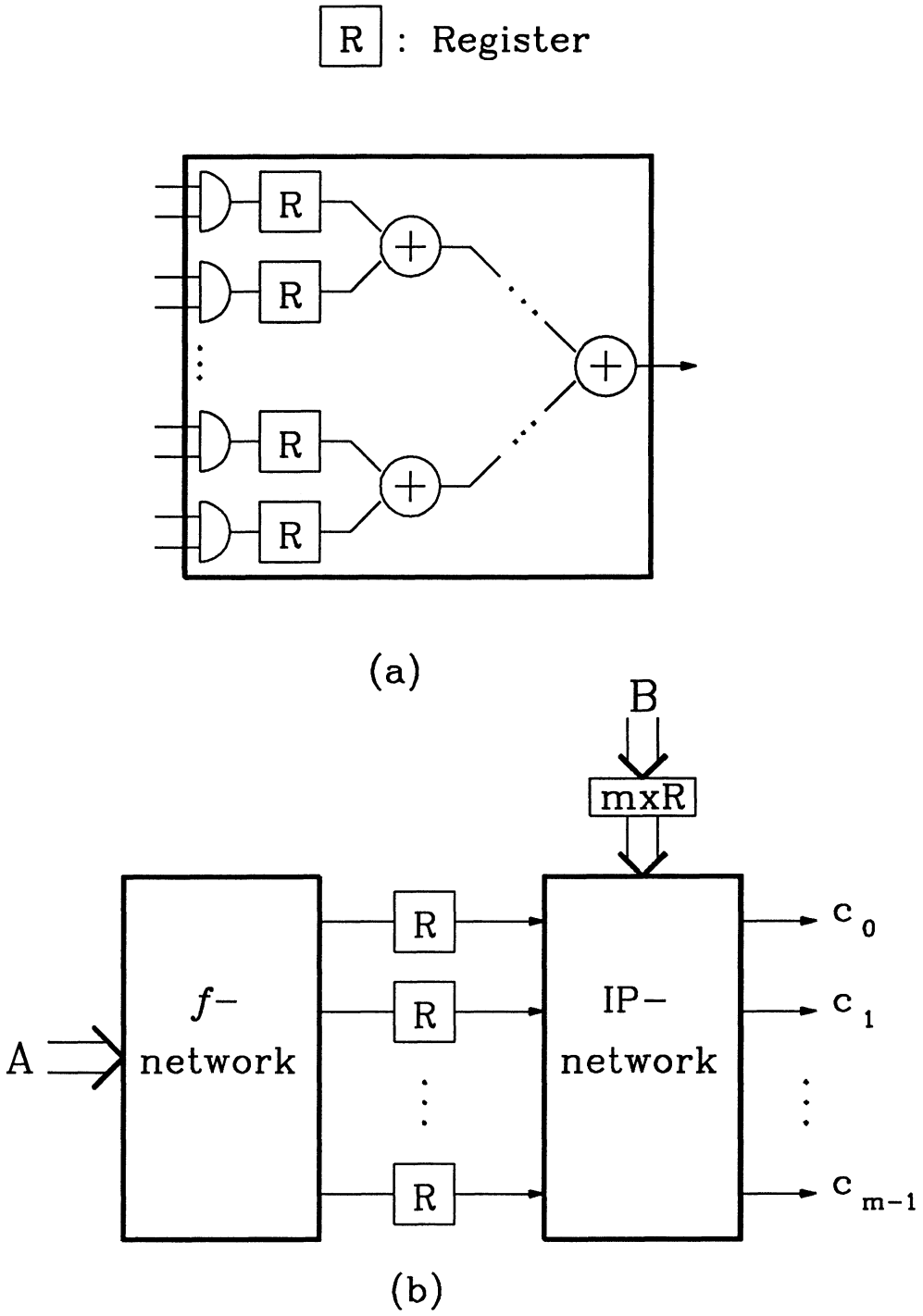


FIGURE 4.7 Pipelining an IP–cell (a), (b), another way of pipelining the parallel PB multiplier.

By pipelining we can clearly obtain multipliers with performance similar to that of the systolic multiplier but at much less expense (for example ~$3m^2$ instead of ~$14m^2$).

As far as structure is concerned, it is fair to say that the high regularity of the cellular-multiplier makes it the simplest to design and also to expand to other extensions of GF(2). The systolic multiplier is also regular but has a non-negligible fraction (~ 20%) of its complexity outside of the regular array due to the alignment registers. Both multipliers have structures that are independent of $P(x)$.

The MO multiplier with its m identical blocks displays a nice m-fold regularity but it is not as easily expanded since the block has to be redesigned for every new field extension. Also, it is generally difficult to find a low-complexity normal basis (i.e. field generator). The design effort consists in designing the MO network and the IP cell. There are $3m$ signals to be routed, i.e. A, B and the IB signals (see Fig. 4.4b).

Our PB multiplier requires the design of two different subunits: the IP cell and the f-network. The number of signals to be routed is $BW + m$ which is typically between $3m - 1$ (trinomials) and $5m - 3$ (pentanomials). The design effort is thus very similar to that of the MO multiplier.

The PB multiplier is not easily expanded by the same reasons as for the MO multiplier. The main advantage, in this sense, of the PB multiplier is that it is much easier, with the help of our results, to find a low-complexity field generator and, from this, determine the functions defining the f-network.

Chapter 5

Multiplication by a Constant and an Application

It is reasonable to expect that the architecture of a multiplier will simplify if one of the multiplicands is a constant and is, thus, known in advance. Such simplifications are common in conventional arithmetic and can be introduced also in the arithmetic of Galois fields.

Our interest in this topic stems from the fact that multiplications by constants occur frequently in the encoding/decoding of most known algebraic codes (BCH, RS etc.). We will investigate if and how the PB/DB/NB multipliers of Chap. 3 and 4 can be simplified and also introduce three new bit-serial PB architectures. Then we show how two of the new architectures can be used to obtained improved RS encoders that adopt the PB representation. A bit-parallel approach is also investigated.

The chapter is concluded by a comparison of the various architectures and a discussion.

5.1 Polynomial Basis

In the sequel we consider the computation of the product $C = AB$ over GF(2^m) where either A or B is regarded as the constant field element. The field generator is again $P(x)$ and its Hamming weight is w_P.

5.1.1 The SSR Multiplier

In the SSR multiplier of Fig. 3.1 the m registers A_i can be removed. If we give up the nice bit-slice architecture we can also reduce the number of gates. Let w_A be the (Hamming) weight of the constant A. Then the number of XOR gates required is $w_A + w_P - 1$. Low-weight constants/field generators should thus be preferred. The AND gates can be skipped since these are replaced by a wire/no wire. The complexity of the SSR multiplier for constant multiplication is thus

$$C = m + w_A + w_P - 1. \tag{5.1}$$

5.1.2 The MSR Multiplier

The MSR multiplier of Fig. 3.2 does not simplify significantly when one of the multiplicands is fixed. Both the upper and lower register are still needed to compute the product. The feedback logic can obviously be simplified for fixed $P(x)$: all AND gates can be skipped and only $w_P - 2$ XOR gates are needed between in the LFSR. The complexity of the MSR multiplier for constant multiplication is thus

$$C = 4m + w_P - 2 \tag{5.2}$$

which is about twice as much as for the SSR multiplier.

5.1.3 Two New Architectures for Arbitrary $P(x)$

In this subsection we introduce two new architectures one of which will turn useful in the design of efficient RS encoders.

In eq. (3.6) we introduced the matrix Z whose columns are the m consecutive states of a Galois-type LFSR with feedback polynomial $P(x)$ that has been initially loaded with A. Eq. (3.6) also indicated that the product bit c_i can be obtained by computing the inner product of B and the i:th row of Z. However, the i:th row of Z is not available directly after loading the LFSR with A. Let $Z_{i,-}$ denote the i:th row of Z. Our problem is to modify or redesign the multiplier of Fig. 3.2 so that the rows of Z are made available for the computation of $B \cdot Z_{i,-}$.

The Transposed Shift Register (TSR) Multiplier

This is a brute-force solution. Our goal is to produce the transpose of Z. This can always be done by means of an m by m register array as shown in Fig. 5.1a. We call this array the transposition array, TA for short. The columns of Z are loaded as rows in the registers $V_{i,j}$. When the last row has entered the TA, the whole matrix Z is moved into the registers $H_{i,j}$ from which the transpose of Z can be read out. For each column of the TA being read out, one product bit c_i is computed by the IP cell. The output bits are thus produced serially either starting from c_0 or from c_{m-1} (depending on how we read the TA). While computing one product, the matrix Z of the next

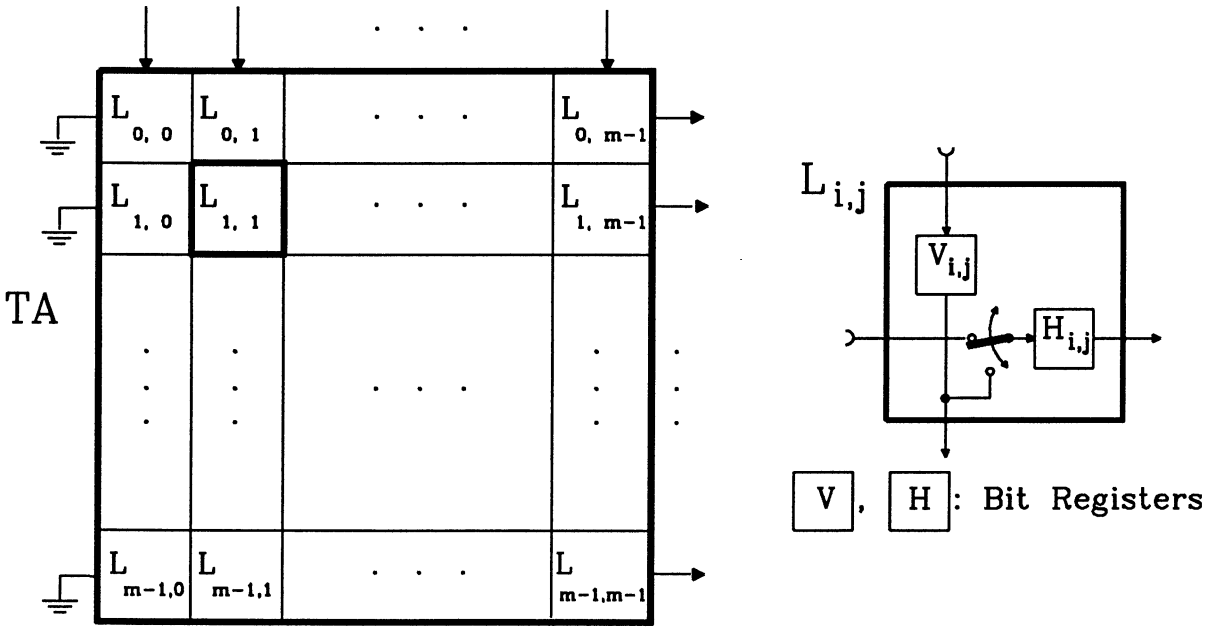


FIGURE 5.1a The transposition array TA.

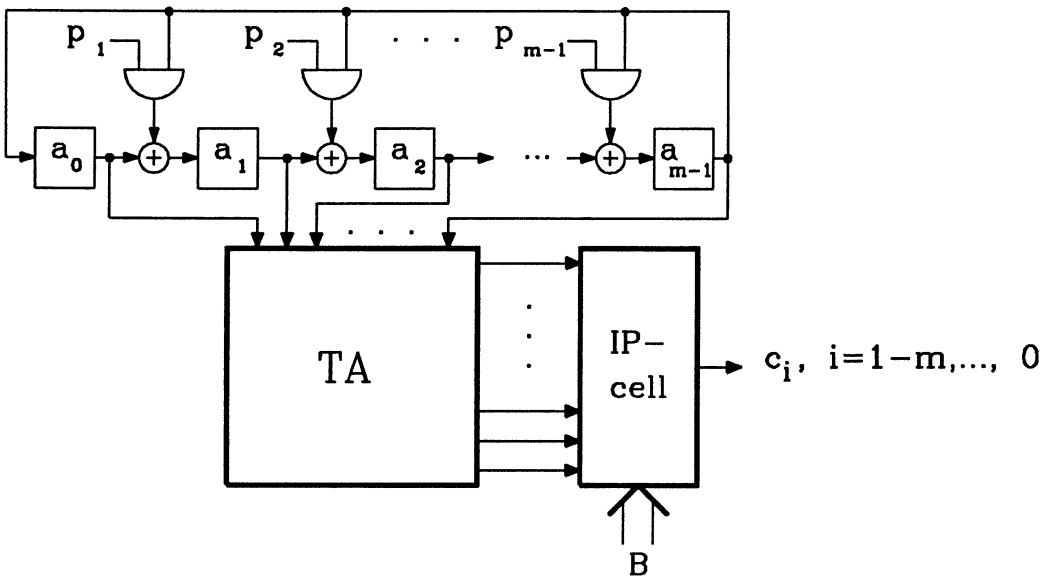


FIGURE 5.1b The general TSR multiplier.

multiplicand is being shifted into the registers $V_{i,j}$. Fig. 5.1b shows the general structure of the TSR multiplier for generic $P(x)$ and arbitrary elements A and B.

The complexity of the TSR multiplier is clearly dominated by the TA. This consists of m^2 identical cells, each cell containing 2 registers and has thus complexity $2m^2$. The total complexity is then

$$C = 2m^2 + 5m - 1 \tag{5.3}$$

which might seem exceedingly high to make a sequential multiplier interesting. However, if the element B is a constant, the IP cell can be reduced to a parity tree consisting of $w_B - 1$ XOR gates. For fixed $P(x)$ we can further simplify and the total complexity becomes

$$C = 2m^2 + m + (w_P - 2) + (w_B - 1). \tag{5.4}$$

The TSR multiplier might be attractive when the simultaneous products of a variable (A) times many different constants are required; for example when multiplying polynomials over GF(2^m), see Fig. 5.2. Then the output of the TA could be broadcast to the many parity trees embedding the constants and the additional complexity introduced by the TA would be shared by all multipliers.

The α-Array Multiplier

In Sec. 4.2 we introduced what we call the α-array. There we used the α-array to derive bounds. Here we use it to design serial multipliers as follows. Rather than generating the columns of Z recursively in time by the Galois-type LFSR, we can generate all of the them simultaneously by means of an α-array. Then we connect the α-array to an IP cell as shown in Fig. 5.3. The m-stage control register at the bottom of the α-array is used to cyclically connect the IP cell to each of the m vectors of signals which correspond to the rows of Z. This register contains simply a circulating 1 that activates the different switch lines.

The output bits c_i are produced serially starting from c_{i_0} where i_0 is the initial position of the 1 circulating in the control register. This initial position can be set to any value in the range [0, 1, ..., $m - 1$] which means that we are free to start anywhere.

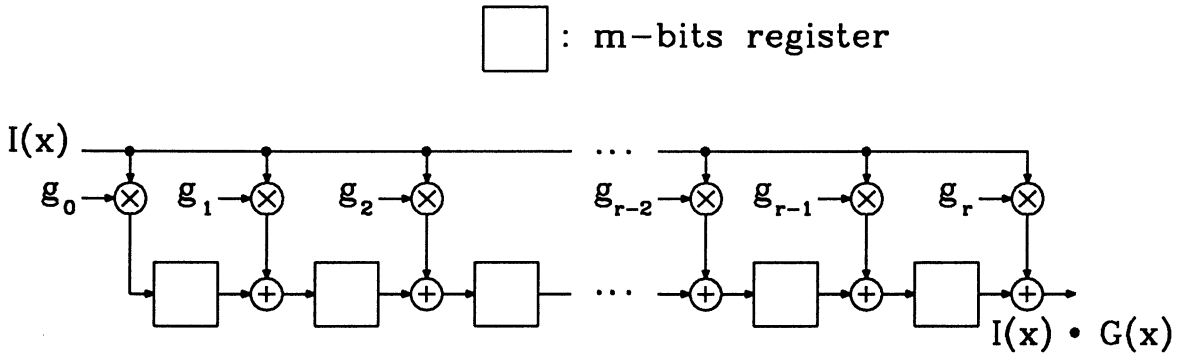


FIGURE 5.2 A circuit for multiplication by the polynomial G(x) over GF(2^m).

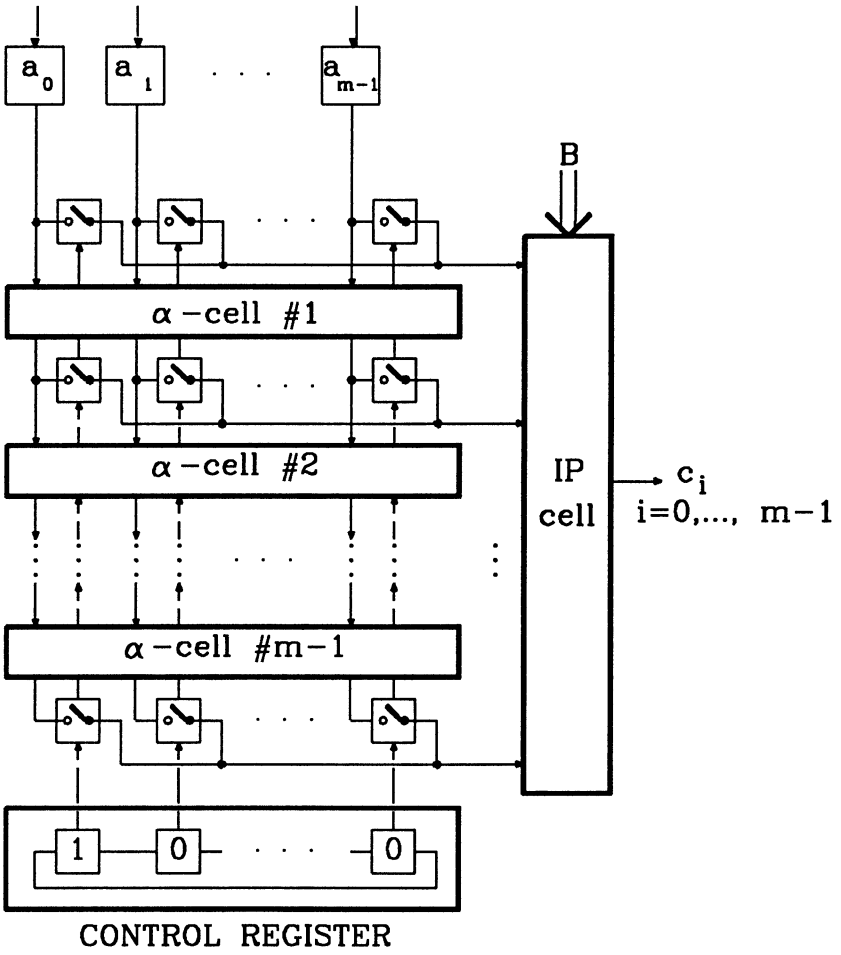


FIGURE 5.3 The general α−array multiplier.

The α-array has complexity $(m-1)(w_P-2)$ which for low w_P is much less than the $2m^2$ of the TA. The total complexity for arbitrary A and B is

$$\mathcal{C} = (m-1)(w_P-2) + 4m - 1 \tag{5.5}$$

which becomes ~ $5m$ for trinomial $P(x)$ and ~ $7m$ for pentanomial $P(x)$. If B is a constant we replace the IP cell by a parity tree and obtain

$$\mathcal{C} = (m-1)(w_P-2) + 2m + w_B - 1 \tag{5.6}$$

which becomes ~ $3m + w_B$ for trinomial $P(x)$ and ~ $5m + w_B$ for pentanomial $P(x)$.

How about performance? Since we start activating the i_0:th column of the α-array in order to produce c_{i_0} we must determine the time required to get stable signals along this column. In fact, it is enough to investigate the CP leading to the last signal at the bottom of the i_0:th column. The signals along the other columns have at least one extra clock cycle to get stable before they are accessed, so they are not critical. The CP goes thus through one register, the i_0:th column of the α-array and the IP cell alternatively parity tree and its length is

$$\mathcal{L} = 2 + \lambda_{i_0} + \lceil \log_2 m \rceil \text{ for arbitrary } B \text{ and} \tag{5.7}$$

$$\mathcal{L} = 1 + \lambda_{i_0} + \lceil \log_2 w_B \rceil \text{ for constant } B. \tag{5.8}$$

To maximize performance for given $P(x)$, i_0 should be chosen such that

$$\lambda_{i_0} = \min_j \{\lambda_j\} \triangleq \lambda_{\min} \tag{5.9}$$

(λ_i was defined at the end of Sec. 4.2). The value of $\lambda_{\min}$ can in turn be minimized by a clever choice of field generator.

In an α-cell the CP goes always through one XOR gate independently of the field generator. However, this does not automatically imply that the concatenation of m-1 α-cells will result in a CP of length m-1 along the i_0:th or, in fact, any other column. For example, it is quite easy to realize, see Fig. 4.3, that $\lambda_0 = 0$ and $\lambda_1 = \lambda_2 = \cdots = \lambda_{m-1} = 1$ for $P(x) = 1 + x + x^m$. In Tab. 5.1 we show some results that we have been able to derive by computing the profile of many prime trinomials (with the help of the Pascal program in App. C).

$P(x)$	Profile $\mathcal{P}$
$1+x+x^m$	$\lambda_0=0, \lambda_1=\lambda_2=\cdots=\lambda_{m-1}=1$
$1+x^k+x^m$, $2\le k\le m/2$	$\lambda_{k-1}=0, \lambda_0=1$, $1\le\lambda_i\le 2$ otherwise
$1+x^k+x^m$, $m/2<k\le m-2$	$\lambda_{k-1}=0$, $1\le\lambda_i\le m-2$ for $i\ne k-1$
$1+x^{m-1}+x^m$	$\lambda_{m-2}=0$, $\lambda_{m-1}=m-1$, $1\le\lambda_i\le m-2$ otherwise

Table 5.1 Some experimental results about the profile of α-arrays generated by trinomials.

It is interesting to notice that $\lambda_{\min} = 0$ and $\lambda_{m-1} \ge 1$ in the α-array generated by a trinomial.

We computed the profile also for many prime pentanomials in order to determine a lower bound on $\lambda_{\min}$. We state the result as a conjecture.

Conjecture 5.1 Let $P(x)$ be a prime polynomial over GF(2) of Hamming weight at least 5. Then for the profile of the associated α-array we have

$$\lambda_{\min} \ge 2. \square$$

Previously we mentioned that the product bits can be produced in cyclic order starting from any c_i. However, in practice it is interesting to have the bits starting from either c_0 (LSB) or c_{m-1} (MSB). While the conversion from a different ordering to any of these preferred orderings can be accomplished by using a pair of m-stage registers and some control logic, we will see that it can be avoided, at least for primitive field generators of degree ≤ 16, by a clever choice of field generator without losing in performance.

Definition 5.2 A prime (primitive) polynomial of degree m is a (primitive) λ_0-*optimal* polynomial if the first component λ_0 of the profile of the associated α-array is minimal, in the sense that no other prime (primitive) polynomial of the same degree yields a profile with lower λ_0. $\square$

Definition 5.3 Let $P(x)$ be a prime (primitive) polynomial of degree m yielding a profile $\mathcal{P} = (\lambda_0, \lambda_1, \ldots, \lambda_{m-1})$. Then $P(x)$ is a (primitive) *strictly* λ_0-*optimal* polynomial if the following conditions are met simultaneously:

a) $\lambda_0 \leq \lambda_j \; \forall \, j \in [1, 2, \ldots, m-1]$,

b) there exists no other prime (primitive) polynomial of the same degree having a profile with any of its components being $< \lambda_0$. □

Notice that strictly λ_0-optimal polynomials of every degree do not necessarily exist.

m	$P(x)$	$\mathcal{P}$ $(\lambda_0, \lambda_1, \ldots, \lambda_{m-1})$
2	0,1,2††	$\underline{0}$,1
3	0,1,3††	$\underline{0}$,1,1
4	0,1,4††	$\underline{0}$,1,1,1
5	0,2,5†	1,$\underline{0}$,2,1,1
6	0,3,6*	1,1,$\underline{0}$,2,2,1
6	0,1,6††	$\underline{0}$,1,1,1,1,1
7	0,1,7††	$\underline{0}$,1,1,1,1,1,1
8	0,1,5,7,8*	6,7,6,5,$\underline{4}$,7,6,7
8	0,2,3,5,8	3,$\underline{2}$,4,4,3,4,3,3
8	0,1,3,5,8††	$\underline{2}$,3,3,3,3,3,3,3
9	0,1,9*††	$\underline{0}$,1,1,1,1,1,1,1,1
9	0,4,9°	1,1,1,$\underline{0}$,2,2,2,1,1
10	0,3,10†	1,1,$\underline{0}$,2,2,1,1,1,1,1
11	0,2,11†	1,$\underline{0}$,2,1,1,1,1,1,1,1,1
12	0,3,12*†	1,1,$\underline{0}$,2,2,1,1,1,1,1,1,1
12	0,1,5,8,12°°	$\underline{2}$,3,3,3,3,3,3,3,4,3,3,3
13	0,1,6,7,13††	$\underline{2}$,3,3,3,3,3,3,4,4,4,4,3,3
14	0,5,14*†	1,1,1,1,$\underline{0}$,2,2,2,2,1,1,1,1,1
14	0,2,7,9,14	3,$\underline{2}$,4,3,3,3,3,4,3,4,4,4,4,3
14	0,1,3,5,14°°	$\underline{2}$,3,3,3,3,3,3,3,3,3,3,3,3,3
15	0,1,15††	$\underline{0}$,1,1,1,1,1,1,1,1,1,1,1,1,1,1
16	0,5,6,11,16	3,3,3,3,$\underline{2}$,4,5,5,5,4,3,4,4,4,4,3
16	0,1,4,6,16††	$\underline{2}$,3,3,3,3,3,4,3,3,3,3,3,3,3,3,3

Table 5.2 The profiles $\mathcal{P}$ of the α-arrays generated by the polynomials of Tab. 4.5 and three additional polynomials of degree 8, 14, 16. The λ_{min} of each profile is underlined. Non-primitive polynomials are marked by *. The entries marked by † are λ_0-optimal polynomials whereas those marked by †† are strictly λ_0-optimal polynomials. Primitive λ_0-optimal polynomials are marked by °. Primitive strictly λ_0-optimal polynomials are marked by °°.

In Tab. 5.2 we show the profiles for the polynomials of Tab. 4.5 together with some additional polynomials for $m = 8, 14, 16$. In this table we indicate also the optimality properties of the polynomials according to the results of Tab. 5.1 and the above conjecture.

We see that, for primitive $P(x)$ of degree ≤ 16, the α-array multiplier can be set to produce the output bits starting from c_0 without penalizing performance.

The results of Tab. 5.1 give us a simple selection criterion for choosing λ_0-optimal trinomials.

Selection criterion for trinomials. Choose $1 + x^k + x^m$ with lowest k. □

Our computer-aided investigation has also provided us with some clear indications for how a pentanomial should look like in order to yield a profile with $\lambda_0 = \lambda_{\min} = 2$. We state it as a conjecture.

Conjecture 5.2. (Also *Selection criterion for pentanomials*) Let $P(x) = 1 + x^{k_1} + x^{k_2} + x^{k_3} + x^m$. If the exponents k_1, k_2, k_3 can be chosen such that

I) $k_1 = 1$
II) $k_3 \leq \lceil m/2 \rceil$ for odd m,
$k_3 \leq m/2 + 1$ for even m
III) $k_1 < k_2 < k_3$

then the profile of the associated α-array has $\lambda_0 = \lambda_{\min} = 2$. □

Consequently, if m is such that no prime trinomials exist, the pentanomials selected by the above criterion are strictly λ_0-optimal. Conditions II, III are to be seen as sufficient but not necessary conditions. We have observed that polynomials with $\lambda_0 = \lambda_{\min} = 2$ can be obtained also for larger values of k_3 than those given in II. However, in those cases it was difficult to derive conditions on k_2. We could only notice that k_2 had to be restricted (from above) to a smaller interval than the one given in III.

In [Wat62] there is a table of primitive polynomials of degree ≤ 100 resulting from a search in which the polynomials were tested in their natural order. This type of search finds soon primitive polynomials that fulfil conditions I, II, III. For example, in [Wat62] we can find such pentanomials for the degrees 12, 13, 14, 18, 19, 24, 26, 27, 30 among others.

We are now able to better specify the length of the CP of the α-array multiplier.

For arbitrary B we have

$$\mathcal{L} = 2 + \lceil \log_2 m \rceil, P(x) = 1 + x^k + x^m, i_0 = k - 1, k \leq m/2 \quad (5.10)$$

$$\mathcal{L} = 3 + \lceil \log_2 m \rceil, P(x) = 1 + x^k + x^m, i_0 = 0, 1 < k \leq m/2 \quad (5.11)$$

$$\mathcal{L} = 4 + \lceil \log_2 m \rceil, i_0 = 0, P(x) = 1 + x^{k_1} + x^{k_2} + x^{k_3} + x^m, \text{ with } k_1, k_2, k_3 \text{ fulfilling I,II,III.} \quad (5.12)$$

For constant B we have

$$\mathcal{L} = 1 + \lceil \log_2 w_B \rceil, P(x) = 1 + x^k + x^m, i_0 = k - 1, k \leq m/2 \quad (5.13)$$

$$\mathcal{L} = 2 + \lceil \log_2 w_B \rceil, P(x) = 1 + x^k + x^m, i_0 = 0, 1 < k \leq m/2 \quad (5.14)$$

$$\mathcal{L} = 3 + \lceil \log_2 w_B \rceil, i_0 = 0, P(x) = 1 + x^{k_1} + x^{k_2} + x^{k_3} + x^m, \text{ with } k_1, k_2, k_3 \text{ fulfilling I,II,III.} \quad (5.15)$$

5.1.4 A New Architecture for $P(x) = 1 + x + x^m$

In Ch. 4 we saw that the entries of Z are linear combinations of the coefficients of A. We can thus write

$$C = (Z_0 + Z_1 + \ldots + Z_{m-1})\, B$$

where the Z_s, $s = 0, 1, \ldots, m-1$ are m by m matrices whose entries are either (single) coefficients of A or zeros. Most such matrices will contain only zeros and could be omitted by a clever choice of $P(x)$. In fact, the number of non-zero matrices is given by the maximum width WD_{max} as defined in Def. 4.2. When $WD_{max} = 2$, we have

$$C = (Z_0 + Z_1)\, A.$$

If Z can be chosen so to obtain two circulant or in another way suitably structured matrices, the implementation of the above equation can result in an interesting bit-serial architecture.

A class of polynomials which lends itself to such a construction is the class of trinomials $1 + x + x^m$. We demonstrate the idea by an example in GF(16).

Example 5.1 Let $P(x) = 1 + x + x^4$. By Prop. 4.15 or Ex. 4.1 we know that

$$Z = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0+a_3 & a_3+a_2 & a_2+a_1 \\ a_2 & a_1 & a_0+a_3 & a_3+a_2 \\ a_3 & a_2 & a_1 & a_0+a_3 \end{pmatrix} = Z_0 + Z_1 =$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 \\ 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & a_3 \end{pmatrix} + \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}.$$

In this case Z_1 is a pure circulant while Z_0 is not. Still Z_0 has enough structure to simplify implementation. We observe that, except for the top row, the entries of the left-most column of Z_1 "enter" the right-most column of Z_0. This observation can be exploited as follows, see Fig. 5.4a. We let a shift-register S_1 of length 4 (m in general for this class) contain the bottom row of Z_1 and another shift-register S_2 of length 3 (m-1 in general for this class) contain the bottom row of Z_0 (the left-most entry is excluded). Then we feed the output of S_1 back to its input and to the input of S_2. Also, we add the elements of S_1 to suitable elements of S_2. The idea is to perform a rowwise addition of Z_0 and Z_1 starting from the bottom row. After addition of rows number i we compute the inner product (sum of rows i)$\cdot B$ and obtain c_i, see Fig. 5.4b. The process is repeated m–1 times until the m–1 highest product bits have been computed. Now we have to take care of the top row of Z_0, the all-zeros row. A simple way to obtain this row is to have a control signal that resets the content of S_2 after m–1 clock pulses. □

In general this multiplier requires, for arbitrary multiplicands, $3m-2$ gates and $3m-1$ registers, i.e.

$$\mathcal{C} = 6m - 3$$

and the length of the CP is

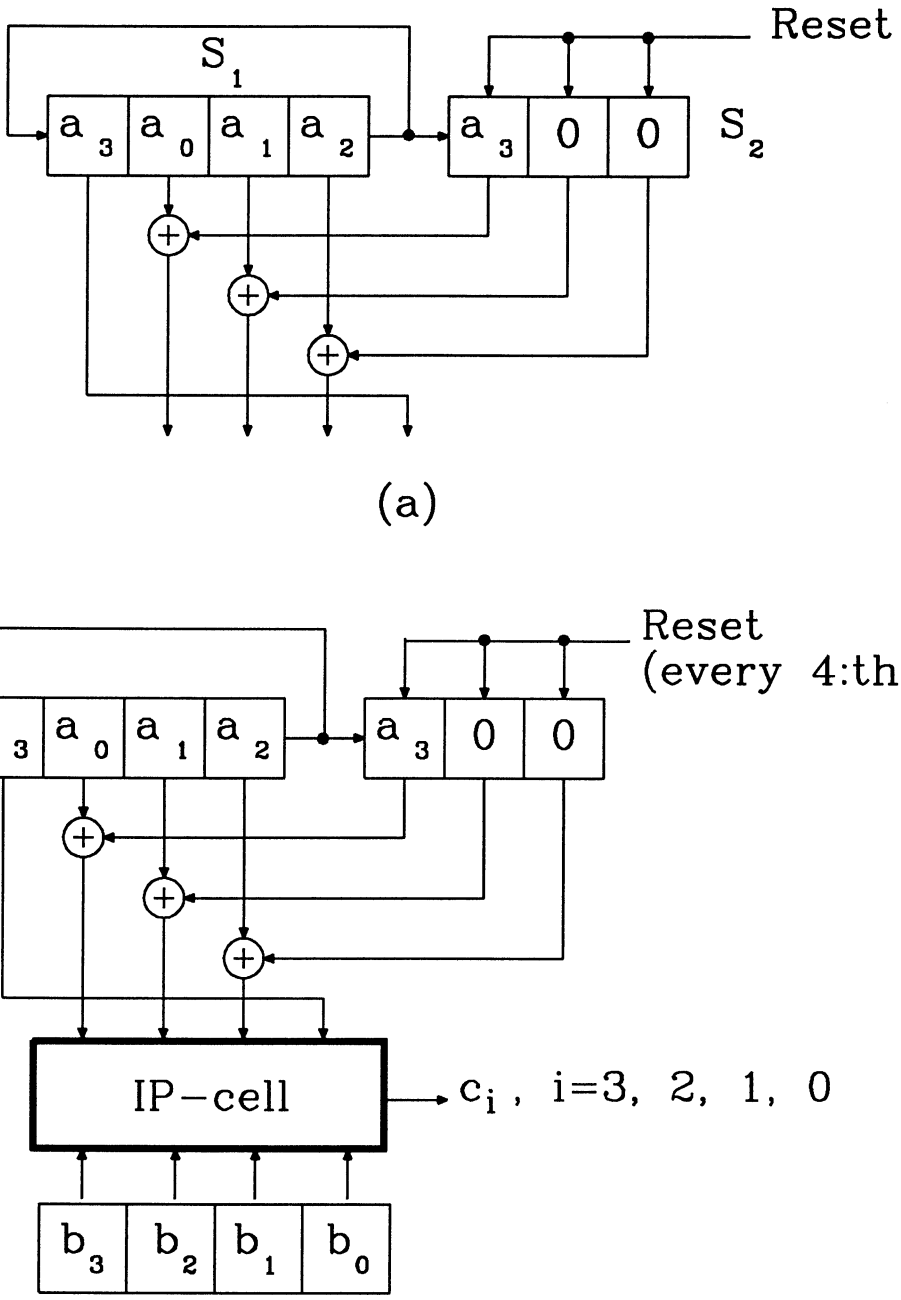


FIGURE 5.4 A circuit for generating the columns of Z with $P(x)=1+x+x^4$ (a) and the corresponding bit-serial multiplier over GF(16), (b).

$$\mathcal{L} = 2 + \lceil \log_2 m \rceil .$$

The multiplier is well suited for constant multiplication. In this case the IP-cell reduces to a parity tree and

$$\mathcal{C} = 3m - 1 + (w_B - 1), \qquad \mathcal{L} = 2 + \lceil \log_2 w_B \rceil . \tag{5.16}$$

5.1.5 Bit-Parallel Multiplication

Multiplication by a constant can be performed in a parallel fashion. The approach used to derive the MSR multiplier is well suited for this purpose.

We know that $C = Z \cdot B$ where the columns of the matrix Z are the m consecutive states of a Galois-type LFSR with feedback polynomial $P(x)$ that has been initially loaded with A.

If A is constant, Z becomes a constant binary matrix and the computation of C reduces to the computation of m parities on suitable subsets of B's coefficients.

Example 5.1 Let $1 + x + x^4$ generate GF(16) and $A = (0, 1, 0, 1)$. Then by Ex. 4.1 we have

$$C = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} b_1 + b_3 \\ b_0 + b_1 + b_2 + b_3 \\ b_1 + b_2 + b_3 \\ b_0 + b_2 + b_3 \end{pmatrix} . \square$$

With the help of a very short and simple computer program implementing the above LFSR, one can readily find the matrix Z of any field element (even for large m).

Since Z's rows have all weight ≥ 1 the number of XOR gates required to implement the multiplication is simply

$$\mathcal{C} = w_Z - m \tag{5.17}$$

where w_Z is the Hamming weight of Z (the number of XOR gates may be reduced further by exploiting eventual residual symmetries in Z). The average Hamming weight of a field element is $m/2$ which implies an

average complexity of $m^2/2 - m$. Therefore, if possible, it is important to choose the constant carefully.

The critical path goes through the parity tree corresponding to the row of Z of maximum weight and has length

$$\mathcal{L} = \lceil \log_2 rw_Z \rceil$$

where rw_Z is the maximum row weight of Z.

5.2 Dual Basis

This is the multiplier that takes best advantage of the fact that one of the multiplicands is a constant. For fixed $P(x)$ and constant B the multiplier of Fig. 3.4 has complexity

$$\mathcal{C} = m + (w_P - 2) + (w_B - 1) \tag{5.18}$$

and a CP of length

$$\mathcal{L} = 1 + \lceil \log_2 w_B \rceil$$

since both IP cells reduce to parity trees. The element A and the product C are still in DB whereas the constant B is in PB. However, B is now hardwired in a parity tree which means that no change of basis has to be performed prior to each multiplication. The change of basis is done once for ever off-line.

It is interesting to notice that the operation of the DB multiplier is basically the same as that of the MSR multiplier, as indicated by the similarity of Fig. 3.2a and Fig. 3.5.The DB multiplier can namely be described as a device computing a product of the form $Z' \cdot B$ where the columns of the m by m binary matrix Z' are just the m consecutive states of a Fibonacci-type LFSR that has been initially loaded with A (this is a direct interpretation of Fig. 3.5). However the matrix Z' displays a key property that allows the DB multiplier to be alternatively realized as shown in Fig. 3.4. This key property is *symmetry* and follows directly from the way in which the next state of a Fibonacci-type LFSR is generated (see Fig. 3.3). The transpose of a symmetric matrix is the matrix itself which explains why there is no need for transposition logic in Fig. 3.4.

5.3 Normal Basis

The NB multiplier takes no advantage of the fact that one of the multiplicands is a constant. Indeed, both multiplicands must still be shifted around to produce the output bits and the logic function f cannot be simplified (see Fig. 3.9).

5.4 Reed-Solomon Encoders

For a given RS code, the complexity of its encoder will depend strongly on how multiplication by a constant in GF(2^m) is realized. In this section we discuss the design of efficient RS encoders that make use of polynomial-basis constant multipliers. To this end, it is helpful to review briefly the theory of RS encoding.

We consider a cyclic RS code over GF(2^m) of blocklength n, dimension k and minimum distance $d = 2t + 1$ where t is the maximum error-correcting capability of the code. The generator polynomial is the monic polynomial $G(x)$ of degree $n - k = 2t$ given by

$$G(x) = \prod_{j=\mu}^{\mu+2t-1}(x - \beta^j) = \sum_{i=0}^{2t} g_i x^i \tag{5.19}$$

where μ is an integer and $\beta, g_i \in$ GF(2^m) with β being a primitive element. The codewords of the RS code are all multiples of $G(x)$ of degree at most n–1. A non-systematic encoding procedure is thus to multiply the information polynomial $I(x)$ (of degree at most $k - 1$) by $G(x)$.

In practice it is desirable to have a systematic encoder.

Type-I RS encoders. Let $C(x)$ denote an RS codeword in systematic form, i.e.

$$C(x) = R(x) + x^{n-k}I(x) \tag{5.20}$$

where $R(x)$ is a polynomial of degree $< n - k$ that denotes the redundancy. We reduce both sides modulo $G(x)$ and obtain

$$R(x) = x^{n-k}I(x) \bmod G(x). \tag{5.21}$$

The redundancy is thus given by the rest from the division of $x^{n-k}I(x)$ by $G(x)$. This operation is readily implemented by a Galois-type LFSR over GF(2^m) with feedback polynomial $G(x)$ as shown in Fig. 5.5a. This encoder makes use of $2t$ (symbol) registers and as many multipliers over GF(2^m). Half of the multipliers can be saved if $G(x)$ is chosen to be reversible (i.e. self-reciprocal). For odd d, $G(x)$ can be made reversible by choosing $\mu = 2^{m-1}-t$ [Ber82]. A reversible $G(x)$ has only t distinct coefficients g_j, see Fig. 5.5b

Type-II RS encoders. The previous encoding procedure is suitable for RS codes rate ≥ 1/2. For lower rates one can reduce the complexity by adopting an encoding procedure based on the parity-check polynomial. For a cyclic code of length n with generator polynomial $G(x)$ the following relation must hold

$$x^n - 1 = G(x)H(x) \tag{5.22}$$

where $H(x)$ is the parity-check polynomial. Clearly, $H(x)$ is a monic polynomial of degree k (= $n - 2t$). Eq. (5.22) implies

$$C(x)H(x) = 0 \bmod (x^n - 1) \tag{5.23}$$

since $C(x)$ is a multiple of $G(x)$. If we multiply out the left-hand side of eq. (5.23) we get a polynomial of degree $n - 1 + k$ whose terms of degree ≥ n must be reduced modulo $x^n - 1$ (i.e. using the fact that $x^n = 1$). We obtain thus

$$0 = (c_0h_0 + \sum_{j=n-k}^{n-1} c_j h_{n-j}) + x(\sum_{j=0}^{1} c_j h_{1-j} + \sum_{j=n-k+1}^{n-1} c_j h_{n+1-j}) + \cdots +$$

$$x^{k-1}(\sum_{j=0}^{k-1} c_j h_{k-1-j} + c_{n-1}h_k) + x^k(\sum_{j=0}^{k} c_j h_{k-j}) + x^{k+1}(\sum_{j=1}^{k+1} c_j h_{k+1-j}) + \cdots +$$

$$x^{n-1}(\sum_{j=n-k-1}^{n-1} c_j h_{n-1-j}) \bmod x^n - 1 \tag{5.24}$$

and, consequently,

$$0 = c_0h_0 + c_{n-1}h_1 + \cdots + c_{n-k+1}h_{k-1} + c_{n-k}h_k$$

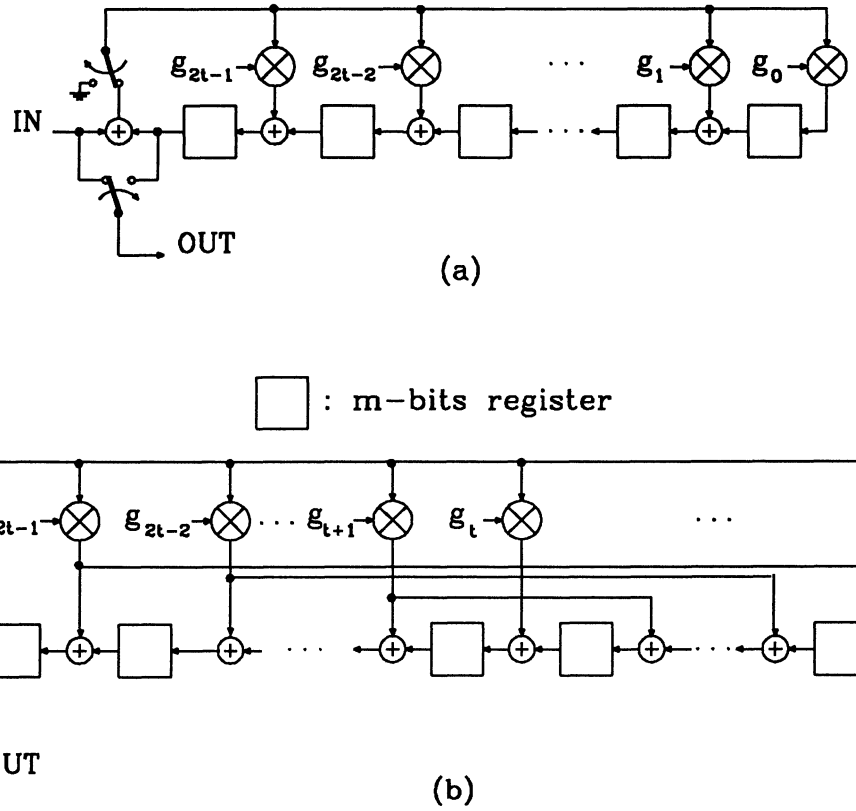


FIGURE 5.5 The type-I RS encoder for non-reversible (a) and reversible (b) generator polynomial G(x) of degree 2t.

$$0 = c_1 h_0 + c_0 h_1 + c_{n-1} h_2 + \cdots + c_{n-k+2} h_{k-1} + c_{n-k+1} h_k$$

$$\vdots$$

$$0 = c_k h_0 + c_{k-1} h_1 + \cdots + c_1 h_{k-1} + c_0 h_k$$

$$\vdots$$

$$0 = c_{n-1} h_0 + c_{n-2} h_1 + \cdots + c_{n-k} h_{k-1} + c_{n-k-1} h_k. \qquad (5.25)$$

The above n equations are just equivalent descriptions of the same linear recurring sequence which is the codeword C itself. This sequence has thus period n and can be generated by a Fibonacci-type LFSR over GF(2^m) with feedback polynomial $H(x)$. The initial state of the LFSR can be any set of k (cyclically) consecutive c_i's. If we choose the information bits $c_0 = i_0$, $c_1 = i_1$, ..., $c_{k-1} = i_{k-1}$ as initial state and clock the LFSR $n - k$ times, it will produce the redundancy $R(x)$. Notice that if h_0 is not 1 we must replace $h_1, h_2, \ldots, h_k$ by $h_1/h_0, h_2/h_0, \ldots, h_k/h_0$. The resulting encoder, see Fig. 5.6a, makes use of k registers and as many multipliers. Choosing $G(x)$ reversible makes $H(x)$ reversible (which automatically implies $h_0 = h_k = 1$) and the encoder can be simplified, see Fig. 5.6b.

A minor drawback here is the adder tree slowing down performance. Another drawback will turn out to be the fact that all multipliers have different inputs.

Type-III RS encoders. The only way to preserve the performance and the Galois-type structure of the first encoder, and still use $H(x)$, seems to be the following.

If we multiply both sides of eq. (5.21) by $H(x)$ we see that the redundancy $R(x)$ can be computed in the following three steps:

S1) $I'(x) = x^{n-k} I(x)$
S2) $R'(x) = I'(x)H(x) \mod (x^n - 1)$
S3) $R(x) = R'(x) \text{ div } H(x)$

where $R(x)$ div $H(x)$ indicates the quotient obtained from the division of $R(x)$ by $H(x)$. Fig. 5.7 shows the circuit implementing these three steps. Steps S1 and S2 are actually performed simultaneously during the first k clock cycles with the switches in the initial position. The division by $H(x)$ starts soon after, with the switches in the second position, producing the first redundant symbol. After $n - k$ additional cycles the whole $R(x)$ has been

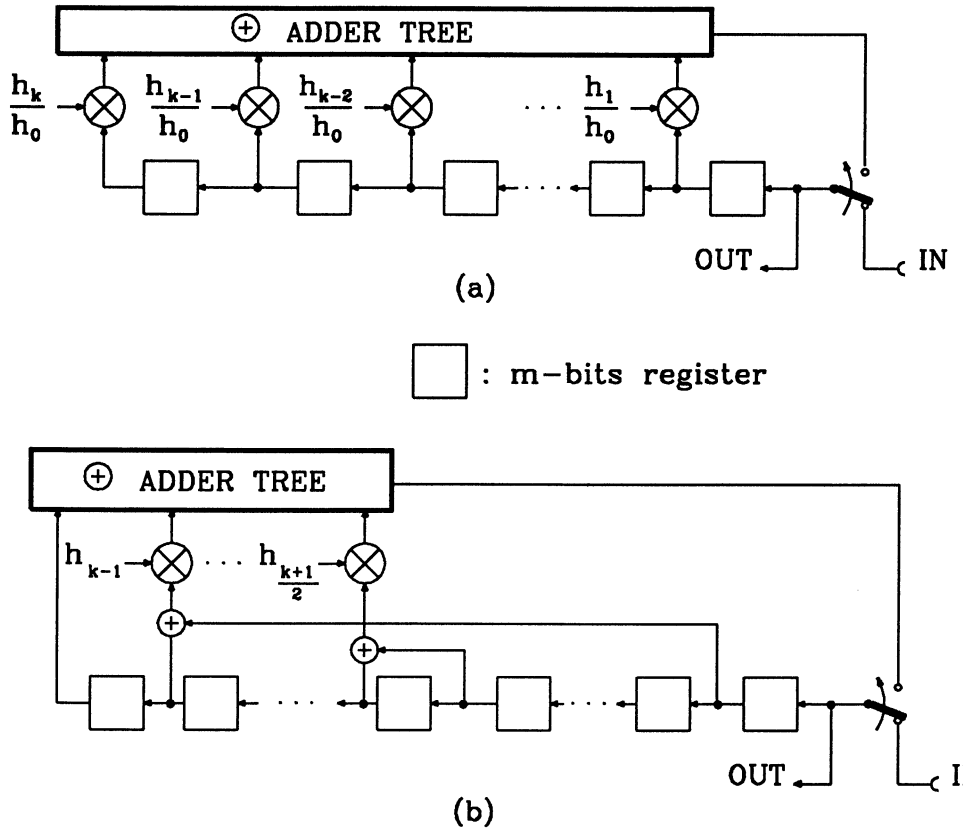


FIGURE 5.6 The type-II RS encoder for non-reversible (a) and reversible (b) parity-check polynomial H(x) of odd degree k.

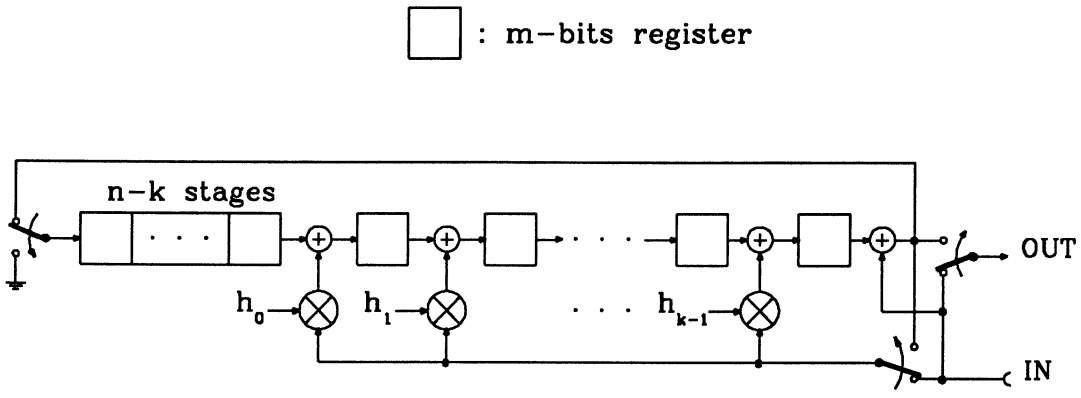


FIGURE 5.7 The type-III RS encoder for non-reversible H(x). The k input symbols are supposed to be followed by n-k zeros.

output and all registers contain zeros (the lowest $n-k$ registers are fed with zeros during the last $n-k$ cycles while the highest registers contain the rest = 0 of the division in S3) and a new encoding can start right away. For reversible $H(x)$, the encoder simplifies as in the type-I approach.

The major drawback of this architecture are the additional $n-k$ registers required to store the result of S2. This drawback is only partially softened by the fact that these registers are very simple delay elements with no special functions and whose intermediate contents need not be accessed (they can thus be compacted extremely well). In general, type-II encoders will be preferred for low-rate RS codes.

In each encoder of Fig. 5.5, 5.6 and 5.7 we have emphasized what we call the *multiplication unit* (MU). The properties of this unit depend clearly on how we choose to implement the multipliers. The other parts of the encoders, on the other hand, do not depend on the multipliers and we call them the *common section*.

In the following subsections we investigate the properties of type-I/II RS encoders based on different multiplier structures. According to the above, we will limit our investigation to the MU.

We will not consider the type-III encoders explicitly since their MU has the same structure as in the type-I encoders.

5.4.1 The TSR Multiplier

In a type-I encoder the products Ag_i could be computed by broadcasting the output of the transposition array TA to as many parity trees, each one embedding one of the constants g_i, $i = 0, 1, \ldots, 2t-1$, see Fig. 5.8. According to eq. (5.4) and Fig. 5.8 we would obtain the following complexity for the MU

$$C = 2m^2 + 2m + w_P - 2 + 2t(w_{g_i} - 1). \tag{5.26}$$

For simplicity, in the sequel we assume $w_{g_i} - 1 = m/2 \; \forall \; i$.

The above equation becomes then

$$C = 2m^2 + 2m + w_P - 2 + tm \tag{5.27}$$

or

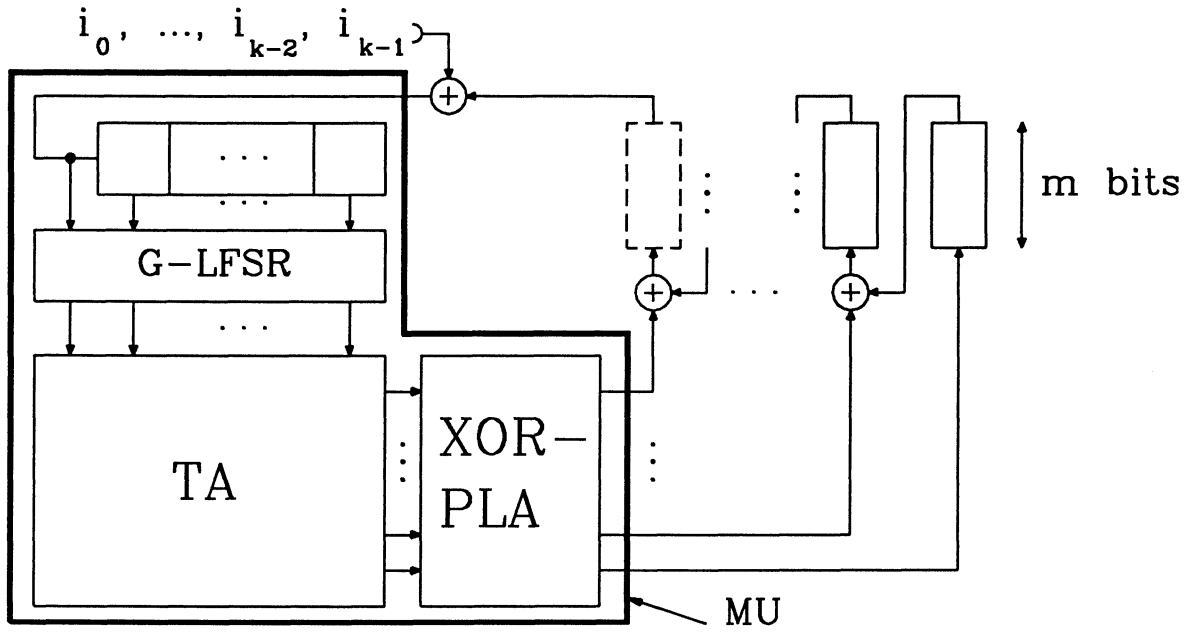


FIGURE 5.8 An attempt to RS encoder based on the TSR multiplier. Due to the 2m clocks delay of the TSR multiplier this approach does not work. G-LFSR stands for Galois-type LFSR.

$$\mathcal{C}_{rev} = 2m^2 + 2m + w_P - 2 + \frac{tm}{2} \tag{5.28}$$

where the subscript *rev* indicates reversible $G(x)$.

The parity trees embedding the constants could be realized as an XOR-PLA (Programmable Logic Array). The rest of the MU does not depend on t.

Unfortunately the additional delay of m clock cycles between first input and first output introduced by the presence of the TA causes a severe timing problem. Indeed, it takes $2m$ cycles for the first input symbol i_{k-1} to reach the parity trees embedding the constants. By this time the second input symbol i_{k-2} is entering the LFSR whereas the third i_{k-3} is just entering the encoder. This means that the wrong feedback element $i_{k-3} + i_{k-1}g_{2t-1}$ will be computed by the input adder instead of the expected $i_{k-2} + i_{k-1}g_{2t-1}$.

Since there seems to be no simple way around this timing problem we must conclude that the TSR multiplier is unsuitable to this application.

5.4.2 Bit-Serial RS Encoders Based on the α-Array Multiplier

Type-I encoders. Fig. 5.9 shows the structure of the RS encoder based on the α-array multiplier. We see that the output of the α-array is broadcast to the parity trees embedding the constants g_i. According to eq. (5.6) and Fig. 5.9 we obtain the following complexity for the MU

$$\mathcal{C} = (m-1)(w_P - 2) + 3m + tm \tag{5.29}$$

or

$$\mathcal{C}_{rev} = (m-1)(w_P - 2) + 3m + \frac{tm}{2}. \tag{5.30}$$

Also in this case the parity trees can be realized as an XOR-PLA. This PLA can be easily replaced if a new RS code with different t is to be implemented. The rest of the MU does not depend on t and can be made to a "macro-cell" to be reutilized for implementing any RS code over the same field. A change of field generator requires a new α-array — i.e. a new α-cell — which is also easily accomplished.

Type-II encoders. Since the output of the α-array can not be shared, we need one complete multiplier for each g_i which means

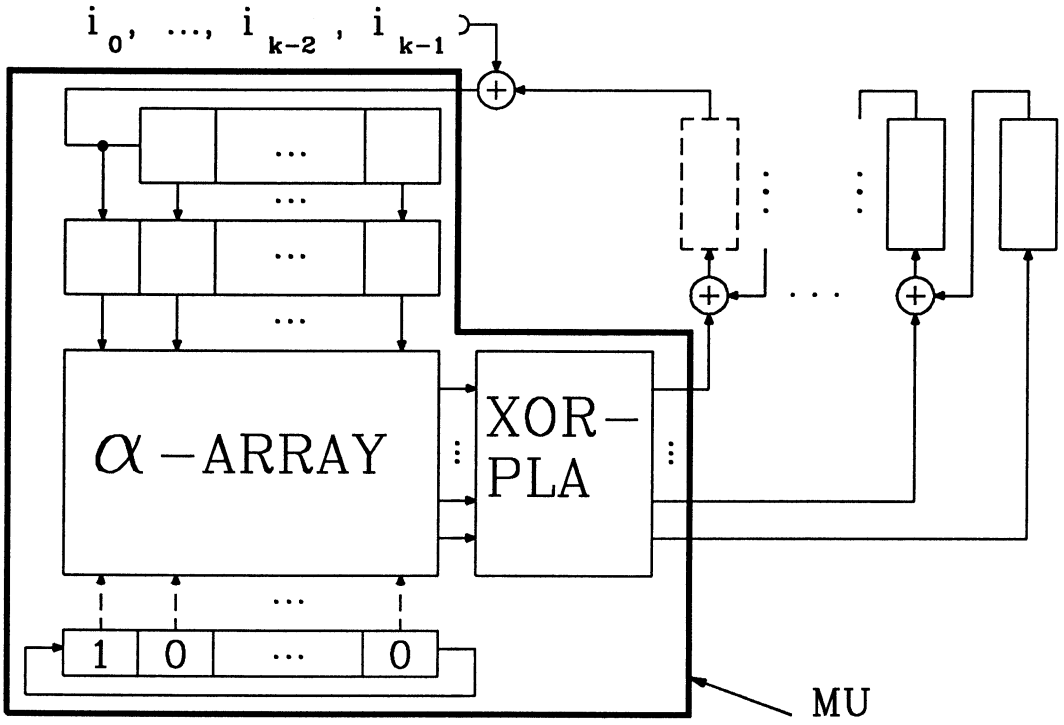


FIGURE 5.9 RS encoder based on the α−array multiplier. The dotted register is removed to allow for proper timing.

$$C = [(m-1)(w_P - 2) + 3m + \frac{m}{2}]\,(n-2t) \tag{5.31}$$

or

$$C_{rev} = [(m-1)(w_P - 2) + 3m + \frac{m}{2}]\frac{(n-2t)}{2}. \tag{5.32}$$

5.4.3 Bit-Serial RS Encoders for $P(x) = 1 + x + x^m$

Type-I encoders. Here we utilize the multiplier of Sec. 5.1.4. Also this multiplier takes advantage of the common input of the multipliers as shown in Fig. 5.10. According to Fig. 5.10 and eq. (5.16) the complexity of the MU is

$$C = 4m - 1 + mt \tag{5.33}$$

or

$$C_{rev} = 4m - 1 + \frac{mt}{2}. \tag{5.34}$$

Type-II encoders. Here we need one complete multiplier for each g_i, i.e.

$$C = (\frac{9m}{2} - 1)(n - 2t) \tag{5.35}$$

or

$$C_{rev} = (\frac{9m}{2} - 1)\frac{(n-2t)}{2}. \tag{5.36}$$

5.4.4 Bit-Serial RS Encoders Based on the SSR Multiplier

This is the conventional approach for designing polynomial-basis RS encoders [Liu82] and is included for purpose of comparison.

Type-I encoders. Recall that the SSR multiplier of Fig. 3.1 presents its output in parallel form. An additional m-stage register is thus needed to serialize the output. By eq. (5.1) each multiplier now has complexity $5m/2 + w_P$ (we set as before $w_A - 1 = m/2$). The complexity of the MU is then

$$C = (\frac{5m}{2} + w_P)2t \tag{5.37}$$

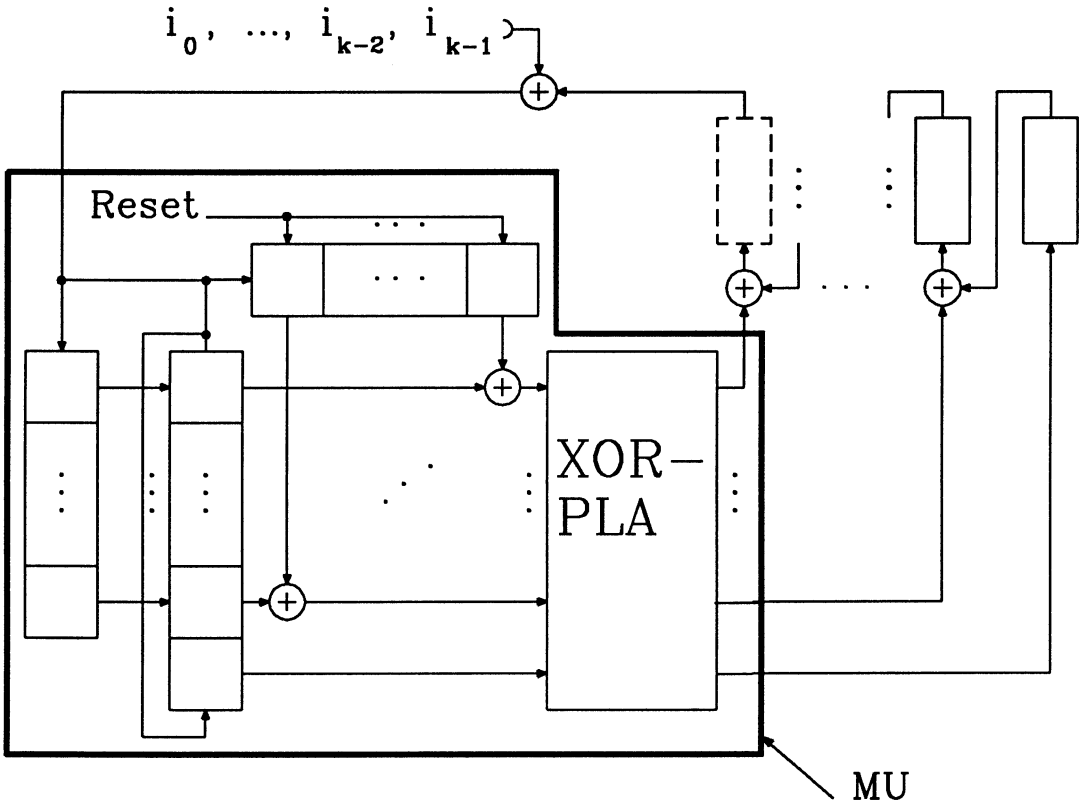


FIGURE 5.10 RS encoder based on the multiplier of Sec. 5.14 ($P(x)=1+x+x^m$).

or

$$C_{rev} = (\frac{5m}{2} + w_P)t. \tag{5.38}$$

In the above calculations we have assumed that each multiplier has been customized to multiply by a particular constant and for a particular field generator in order to minimize the complexity. This customization costs in terms of design effort and implies that all multipliers must be redesigned for a code with different t or for a new field generator. The design work can be simplified by adopting the regular SSR multiplier of Fig. 3.1 with the only modification that the registers A_i are skipped and the serializing register is added at the output. Now, however, the complexity of each multiplier is $6m$ whereby

$$C = 12mt \quad \text{and} \quad C_{rev} = 6mt. \tag{5.39}$$

Clearly, this approach takes no advantage of the fact that all multipliers have the same input.

Type-II encoders. The situation is similar to the previous case. We have thus (for customized multipliers)

$$C = (\frac{5m}{2} + w_P)(n - 2t) \tag{5.40}$$

or

$$C_{rev} = (\frac{5m}{2} + w_P)\frac{(n - 2t)}{2}. \tag{5.41}$$

5.4.5 Bit-Serial RS Encoders Based on the DB Multiplier

Also this section is included for purpose of comparison. The complexity of an RS encoder seems namely to be minimized if the constant multipliers adopts the DB representation [Ber82].

Type-I encoders. According to Fig. 5.11 and eq. (5.18) the MU has complexity

$$C = 2m + w_P - 2 + tm \tag{5.42}$$

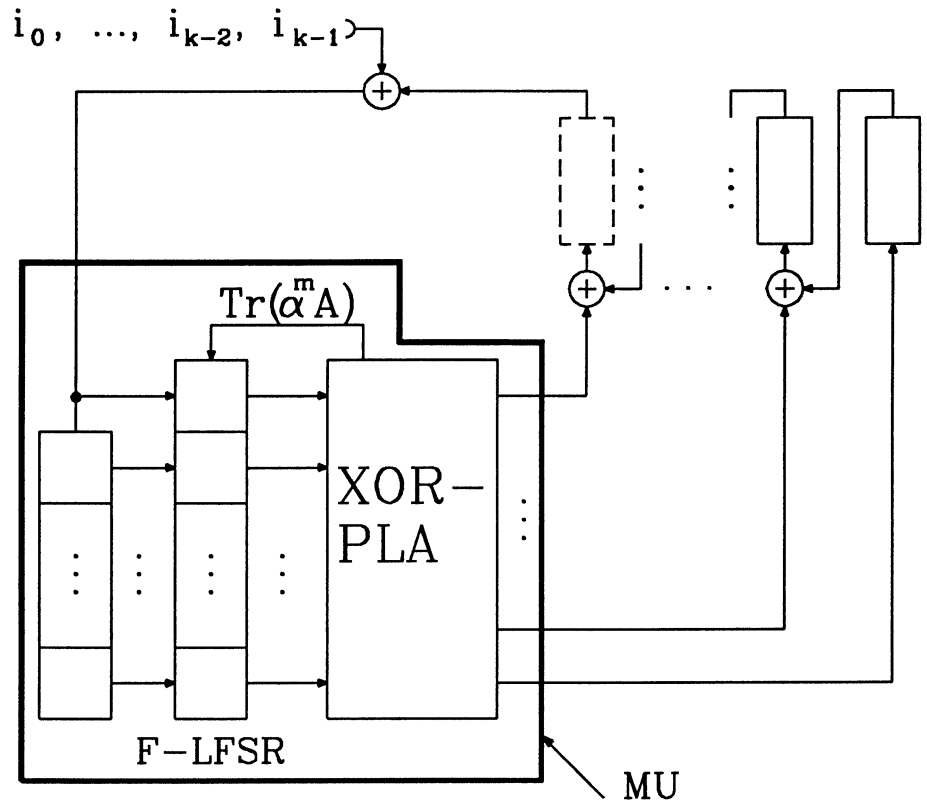


FIGURE 5.11 RS encoder based on the DB multiplier. The feedback logic of the Fibonacci-type LFSR (F-LFSR) is included in the XOR-PLA.

or

$$C_{rev} = 2m + w_P - 2 + \frac{tm}{2}. \tag{5.43}$$

Also in this case the parity trees can be realized as an XOR-PLA. The rest of the MU does not depend on t and can be made to a "macro-cell" to be reutilized for implementing any RS code over the same field. A change of field generator is easily accomplished.

Type-II encoders. One complete multiplier for each g_i is required which means

$$C = (\frac{5m}{2} + w_P - 2)(n - 2t) \tag{5.44}$$

or

$$C_{rev} = (\frac{5m}{2} + w_P - 2)\frac{(n - 2t)}{2}. \tag{5.45}$$

5.4.6 Bit-Parallel RS Encoders

In applications requiring very high throughputs one might consider bit-parallel RS encoders.

In this section we investigate bit-parallel RS encoders that adopt the PB representation of GF(2^m).

Type-I encoders. Fig. 5.12 shows the general structure of a bit-parallel type-I RS encoder. All data paths are now m bits wide and the MU consists of a linear array (LA) and a communication bus (CB). For each constant g_i, the LA computes the m parities, on suitable subsets of the feedback element A's bits, that define the product Ag_i. Since each such product requires at average $m^2/2 - m$ XOR gates, the LA has complexity

$$C = (\frac{m^2}{2} - m)2t \tag{5.46}$$

or

$$C_{rev} = (\frac{m^2}{2} - m)t. \tag{5.47}$$

In many cases of practical interest the design of a bit-parallel RS encoder can be simplified by utilizing a pre-defined general LA that generates all possible linear combinations (i.e. parities on all possible subsets) of the feedback element's bits a_i. If we exclude the null combination and the single a-coefficients there are $2^m - m - 1$ possible linear combinations which require some logic for their computation. The number of XOR gates needed to compute all these combinations is readily derived.

We start generating all double sums $a_i + a_j$, $i, j = 0, 1, \ldots, m-1, j \neq i$. There are $\binom{m}{2}$ such sums whereby as many XOR gates are needed.

Then we generate all triple sums $a_i + a_j + a_k$, $i, j, k = 0, 1, \ldots, m-1$ for which $j \neq i, i \neq k, k \neq j$. There are $\binom{m}{3}$ such sums. Since we have all double sums already available we can generate $a_i + a_j + a_k$ by adding $a_i + a_j$ to the element a_k. This operations costs one XOR gate. All triple sums require thus $\binom{m}{3}$ gates.

In the same way, we generate quadruple sums from double sums using $\binom{m}{4}$ gates and so forth for sums with 5, 6, ..., m terms. Clearly, we can generate all non-trivial linear combinations of the a_i's using

$$C = \binom{m}{2} + \binom{m}{3} + \ldots + \binom{m}{m} = 2^m - m - 1 \tag{5.48}$$

XOR gates. How does the general LA compare to the customized LA? By equating eq. (5.46) to eq. (5.48) we can find the value of t for which the two LA's yield the same complexity. We denote this value by τ. Then

$$\tau = \left\lceil \frac{2^m - m - 1}{m^2 - 2m} \right\rceil \tag{5.49}$$

and, equating eq. (5.47) to eq. (5.48),

$$\tau_{rev} = \left\lceil 2\frac{2^m - m - 1}{m^2 - 2m} \right\rceil. \tag{5.50}$$

In Tab. 5.3 we list values of τ, τ_{rev} and $C = 2^m - m - 1$ for $3 \leq m \leq 16$. We see there that the general LA is definitely viable for $m \leq 10$. Also, for many practical codes are, the general LA is less complex than the customized LA. Among these codes we have, for example, the ESA-NASA standard (255,223) RS code over GF(2^8) with $t = 16$, both with reversible and non-reversible $G(x)$.

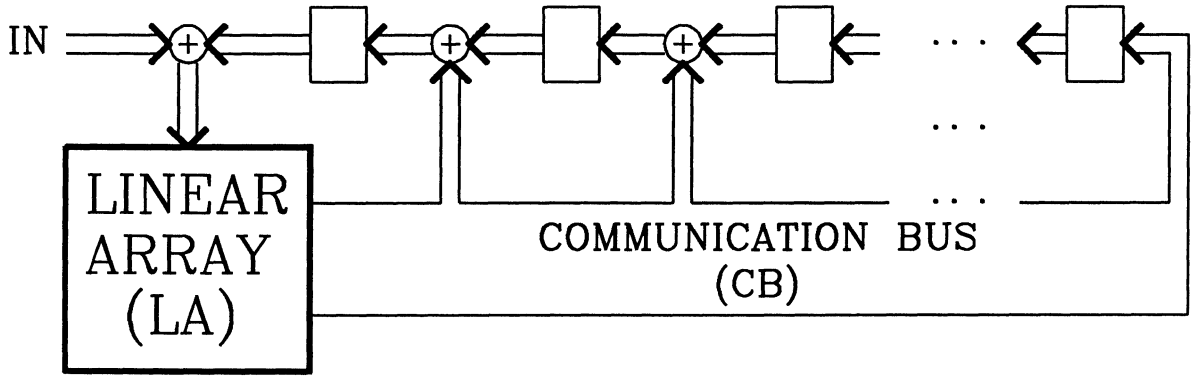


FIGURE 5.12 Bit-parallel RS encoder.

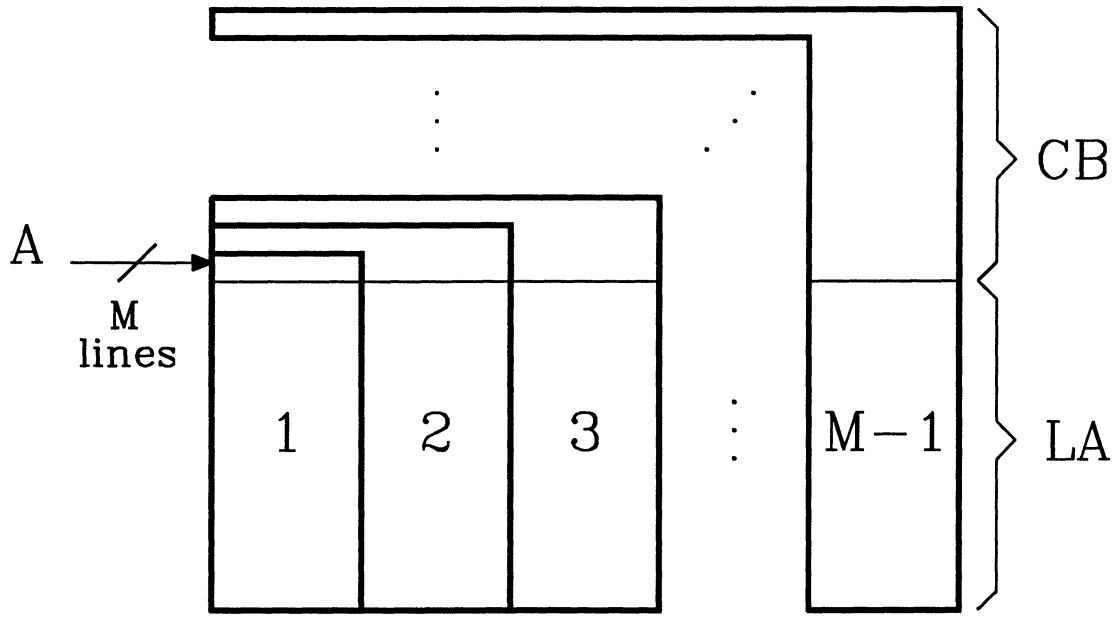


FIGURE 5.13 A modular approach for realizing the general linear array (LA). If m<M only the first m−1 modules are needed.

m	τ	τ_{rev}	$C = 2^m - m - 1$
3	2	3	4
4	2	3	11
5	2	4	26
6	3	5	57
7	4	7	120
8	6	11	247
9	8	16	502
10	13	26	1013
11	21	42	2036
12	34	68	4083
13	58	115	8178
14	98	195	16369
15	168	336	32752
16	293	585	65519

Table 5.3

Maybe the greatest advantage of the general LA is that, once it has been designed, it can be re-used to implement any RS code, over the same field, whose value of t is not much less than τ or τ_{rev}. For every new code, the only major design work left is the customization of the connections between the communication bus and the LFSR adders.

Actually, the general LA could be used for any field GF(2^m), $m \leq M$, if it is designed in a modular fashion, for example as shown in Fig. 5.13. The first module generates all possible linear combinations of a_0 and a_1. The second module generates all possible linear combinations of a_0, a_1 and a_2 except those that are already generated by the first module and so forth for the following modules. The last module generates all possible linear combinations of a_0, a_1, ..., a_{M-1} except those that are already generated by the previous modules. Each module will also contribute to the expansion of the communication bus (although this bus does not contribute to the complexity C, its width = $2^m - 1$ obviously increases the chip area significantly).

Therefore, whenever a code over a field GF(2^m), $m < M$, has to be implemented, only the first m–1 modules are necessary. In this way one does not have to use a larger/slower LA than it is necessary (the CB is also kept small).

The CP of the encoders goes through one register, the LA and one binary adder. Its length is thus

$$\mathcal{L} = 2 + \lceil \log_2 m \rceil. \tag{5.51}$$

During one clock one m-bits symbol is processed.

Type-II encoders. The complexity of the MU is clearly

$$\mathcal{C} = (\frac{m^2}{2} - m)(n - 2t) \tag{5.52}$$

or

$$\mathcal{C}_{rev} = (\frac{m^2}{2} - m)\frac{(n - 2t)}{2}. \tag{5.53}$$

A general LA can not be adopted in this case. Due to the adder tree the CP of the encoder has length

$$\mathcal{L} = 1 + \lceil \log_2 m \rceil + \lceil \log_2 (n - 2t) \rceil. \tag{5.54}$$

5.5 Discussion

In Tab. 5.4 we have summarized the main properties of the constant multipliers investigated in this chapter.

Multiplier/Basis	Complexity $\mathcal{C}$	Length of CP $\mathcal{L}$	Delay
Berlekamp's/Dual	$m + w_P + w_B$	$1 + \lceil \log_2 w_B \rceil$	m
SSR/Pol.	$2m + w_P + w_B$	3	m
Sec. 5.1.4/Pol.	$3m + w_B$	$2 + \lceil \log_2 w_B \rceil$	m
α-Array/Pol.	$(m-1)(w_P - 2) + 2m + w_B$	$d + \lceil \log_2 w_B \rceil$, $d \geq 1$	m
MSR/Pol.	$5m + w_P$	3	m
TSR/Pol.	$2m^2 + m + w_P + w_B$	$1 + \lceil \log_2 w_B \rceil$	$2m$

Table 5.4 Some properties of bit-serial constant multipliers over GF(2^m). The delay indicated is that between first input and first output. All complexity figures refer to multipliers with serialized output. w_B stands for the Hamming weight of the constant.

We see that, as far as multiplication by a single constant is concerned, the DB multiplier has the lowest complexity ($\sim m$) followed by the SSR multiplier ($\sim 2m$) while our new architectures have complexity at least $\sim 3m$.

The new architectures are, on the other hand, better suited than the SSR multiplier to multiple constant multiplication. In this case even the TSR multiplier is often an improvement on the SSR approach. Suppose, for example, that we want to multiply an incoming variable by μ distinct constants (everything is in GF(2^m)). Then for the SSR approach we have (we set for simplicity $w_P = 3$ and $w_B = m/2$)

$$C_{\mathrm{SSR}} = \frac{5\mu m}{2}$$

and for the TSR approach

$$C_{\mathrm{TSR}} = 2m^2 + m + \frac{\mu m}{2}.$$

Equating the above expressions we obtain

$$\mu = m + \frac{1}{2}.$$

Hence, the number of distinct constants needs only be $\geq m$ to make the TSR approach favourable.

If we adopt the α-array approach for the same purpose we have

$$C_\alpha = 3m + \frac{\mu m}{2} \qquad \text{for } w_P = 3$$

and

$$C_\alpha = 5m + \frac{\mu m}{2} \qquad \text{for } w_P = 5.$$

In the former case we gain on the SSR approach already for $\mu = 2$ while in the latter case we gain first for $\mu = 3$.

If we switch to the DB approach we improve further and obtain

$$C_{\mathrm{DB}} = m + \frac{\mu m}{2}.$$

Clearly, for low-weight $P(x)$, the difference between C_{DB} and C_{α} becomes quickly less and less significant with growing μ.

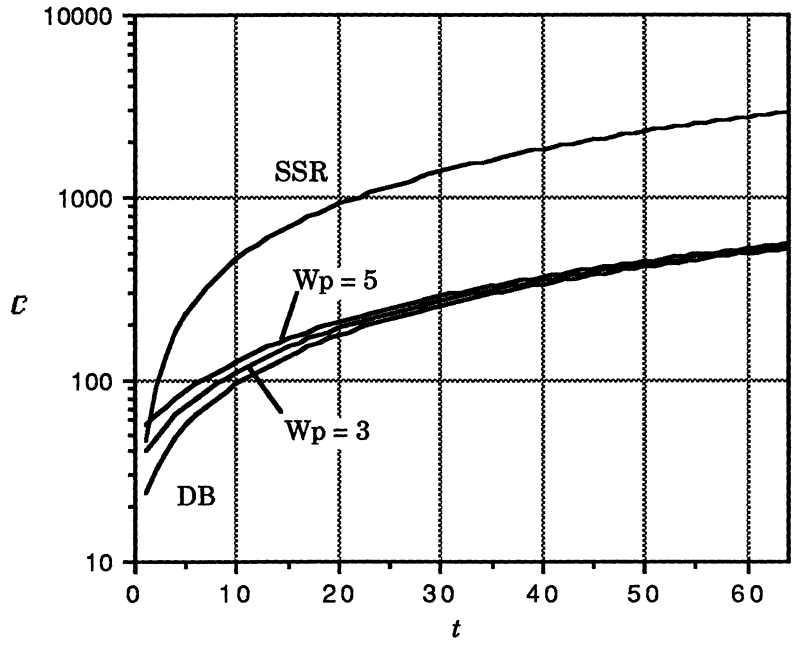
As an application of the above results we have considered the design of RS encoders. We found that two of the new architectures can be used to obtain bit-serial RS encoders of type-I/III that improve on the corresponding encoders based on the SSR multiplier. In Tab. 5.5 we summarize the properties of the multiplication units of the four best encoder architectures.

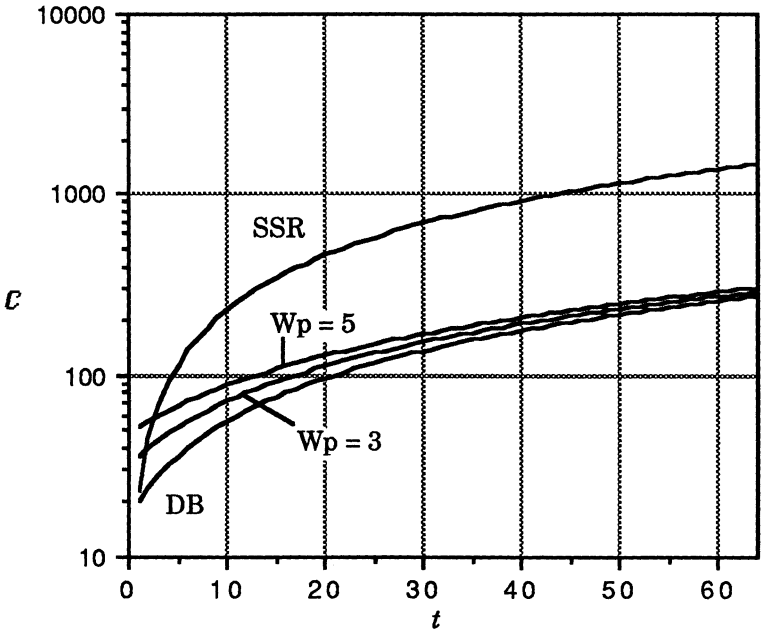
Approach/Basis	Complexity		Length of CP
	C	C_{rev}	L
SSR/PB	$(5m+6)t$	$(5m+6)\frac{t}{2}$	3
Sec. 5.1.4/PB	$4m+mt$	$4m+\frac{tm}{2}$	$2+\lceil \log_2 m \rceil$
α-Array/PB			
w_P=3	$4m+mt$	$4m+\frac{tm}{2}$	$d+\lceil \log_2 m \rceil, 1 \le d \le 2$
w_P=5	$6m+mt$	$6m+\frac{tm}{2}$	$3+\lceil \log_2 m \rceil$
Berlekamp's/DB	$2m+tm$	$2m+\frac{tm}{2}$	$1+\lceil \log_2 m \rceil$

Table 5.5 Properties of MU of bit-serial RS encoders of type-I/III over GF(2^m) for four different approaches. The code has parameters $n, k, d = 2t + 1$. For simplicity we have set $w_B = m/2$ and $w_P = 3$ unless otherwise indicated. The subscript *rev* indicates reversible $G(x)$.

We see from Tab. 5.5 that the two new PB architectures improve on the SSR approach already for $t = 2$ (or $t = 3$ for reversible $G(x)$). Fig. 5.14 shows a plot of the complexities of Tab. 5.5 for $m = 8$ where the complexity of the α-array approach is indicated by "Wp=3" for trinomial $P(x)$ and by "Wp=5" for pentanomial $P(x)$.

Concerning performance we notice that the SSR approach has the advantage of yielding the same performance for any m whereas the performance of the remaining architectures decreases (slowly) with m. The performance of the new approaches is very close to that of the DB approach, often (i.e. for trinomial $P(x)$) it is just the same.





(Reversible G(x))

Figure 5.14 Plot of the complexities of Tab. 5.5 for m=8.

For type-II RS encoders the situation is quite different. Since each multiplier has a distinct input and according to Tab. 5.4, the SSR approach requires fewer components than our new PB architectures. Best is still the DB approach.

Finally, we considered the complexity of bit-parallel RS encoders. In Sec. 5.4.6 we investigated the average performance. In Fig. 5.15 we show a plot of the complexity of type-I/III encoders for $m = 8$ where we included the following four curves:

C_max = the maximum complexity of an RS encoder with customized LA (obtained by a computer search, through all primitive elements of GF(2^8), for the worst β).

C_min = the minimum complexity of an RS encoder with customized LA (obtained by a computer search, through all primitive elements of GF(2^8), for the best β).

C_min_R = the minimum complexity of an RS encoder with customized LA and *reversible* $G(x)$ (obtained by a computer search, through all primitive elements of GF(2^8), for the best β).

Gen. LA = complexity of the general linear array.

In Fig. 5.15 we have thus chosen to compare the general LA with the best possible $G(x)$. Still, we see that the general LA is favourable for many values of t. In particular, it is less complex than the customized LA for $t \geq 7$ (non-reversible $G(x)$) or $t \geq 13$ (reversible $G(x)$). A closer look at Fig. 5.14 and 5.15 shows actually that the general LA, for larger values of t, can even compare favourably with the bit-serial approaches (for example, for $t \geq 30$ with non-rev. $G(x)$).

Our conclusion is that the PB representation is well suited for applications where the simultaneous multiplication of a variable by many different constants is required. In the bit-serial case, the complexity of the PB approach is not as low as that of Berlekamp's DB approach, but the difference between the two is in many cases negligible. This small difference in complexity might very well be compensated for by the fact that the PB approach does not mix different field representations whereby the problem of basis conversion never arises.

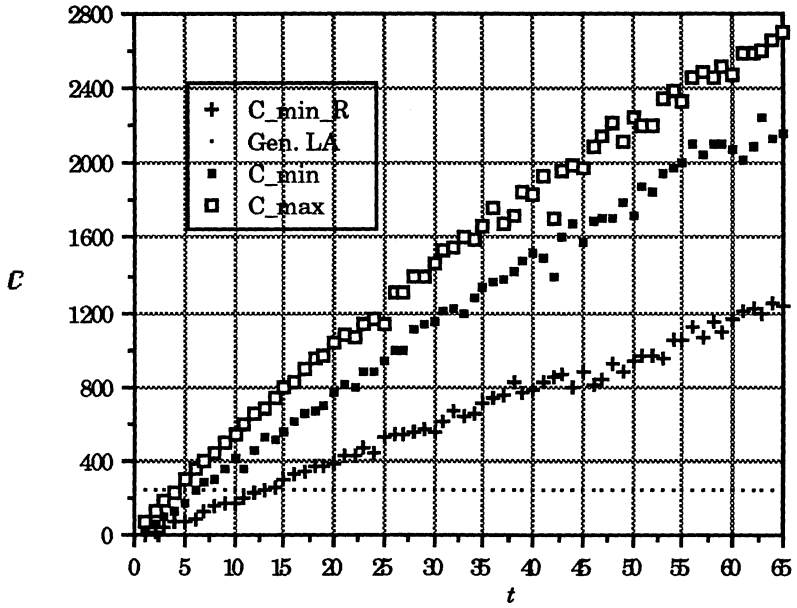


Figure 5.15 Complexity of bit-parallel type-I/III RS encoders for m=8.

Chapter 6

Hybrid Multipliers

In this chapter we investigate the properties of hybrid multipliers over GF(2^m). Let $m = m_1 m_2 \cdots m_l$. A hybrid multiplier is a multiplier that is designed by viewing the composite (i.e. nonprime) field GF(2^m) as an extension of the another composite field, namely the subfield GF($2^{m'}$) where m' is a non-trivial factor of m. A hybrid multiplier has properties which lie between those of the conventional (i.e. non-hybrid) serial and parallel approaches. The hybrid multiplier fills thus the gap between the slow but simple bit-serial multiplier and the fast but complex fully parallel multiplier described in Ch. 3 and 4.

To our knowledge, this type of multipliers has not been previously investigated and compared with the conventional approach.

6.1 Hybrid Multiplication

Hybrid multiplication in GF(q^r) is basically the same as in the conventional approach.

Let GF(q^r) be a finite extension of GF(q), $q = 2^s$, generated by a prime polynomial $\Pi(y) = \pi_0 + \pi_1 y + \cdots + \pi_{r-1} y^{r-1}$, $\pi_i \in$ GF(q). We assume a polynomial-basis representation of GF(q^r) whereby all field elements are represented as linear combinations of the basis elements $\{1, \gamma, \ldots, \gamma^{r-1}\}$ where γ is a root of $\Pi(y)$. Alternatively we identify the field elements with the polynomials of degree $< r$ over GF(q). The field GF(q) is generated by the binary prime polynomial $P(x) = p_0 + p_1 x + \ldots + p_{s-1} x^{s-1}$.

Let $C(y) = A(y)B(y) \bmod \Pi(y)$, $A(y), B(y) \in$ GF(q^r), be the product we wish to compute. Then just as in Sec. 3.1

$$C(y) = [b_0 A(y) + b_1 yA(y) + \cdots + b_{r-1} y^{r-1} A(y)] \bmod \Pi(y) =$$

$$= [b_0 A(y) \bmod \Pi(y)] + [b_1 yA(y) \bmod \Pi(y)] + \cdots + [b_{r-1} y^{r-1} A(y) \bmod \Pi(y)]$$

which leads to the following matrix form

$$C = \begin{pmatrix} c_0 \\ c_1 \\ : \\ c_{r-1} \end{pmatrix} = \begin{pmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,r-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,r-1} \\ : & : & & : \\ z_{r-1,0} & z_{r-1,1} & \cdots & z_{r-1,r-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ : \\ b_{r-1} \end{pmatrix} = Z \cdot B \tag{6.1}$$

where $a_i, b_i, c_i, z_i \in \mathrm{GF}(q)$. The columns of Z are the r consecutive states of a Galois-type LFSR over GF(q) with feedback polynomial $\Pi(y)$ that has been initially loaded with A. We can thus realize a hybrid multiplier with the structure of the earlier MSR multiplier. The main difference is that all coefficients operation are in GF(q) whereby all data lines are s-bits wide. Fig. 6.1 shows the structure of the general hybrid multiplier over GF(q^r), $q = 2^s$.

Let $\mathcal{C}_q$ and $\mathcal{L}_q$ denote the complexity respectively the length of the CP of the generic GF(q) multiplier. Further, let $\mathcal{C}_{\mathrm{FB}}$ denote the complexity of the feedback logic, see Fig. 6.1. Then, for the hybrid multiplier, we have

$$\mathcal{C} = r\mathcal{C}_q + 3rs + \mathcal{C}_{\mathrm{FB}} \tag{6.1}$$

and

$$\mathcal{L} = \mathcal{L}_q + 2. \tag{6.2}$$

If GF(q) can be generated by a trinomial $1 + x^k + x^s$ with $k < s/2$ we know from Tab. 4.9 that $\mathcal{C}_q = 2s^2 - 1$ whereas $\mathcal{L}_q = \mathrm{d} + \lceil \log_2 s \rceil$ with d = 2 for $k = 1$ and d = 3 for $k > 1$. Then

$$\mathcal{C} = 2rs^2 + r(3s - 1) + \mathcal{C}_{\mathrm{FB}} \tag{6.3}$$

and

$$\mathcal{L} = 4 + \lceil \log_2 s \rceil \qquad k = 1 \tag{6.4a}$$

$$\mathcal{L} = 5 + \lceil \log_2 s \rceil \qquad 1 < k < s/2. \tag{6.4b}$$

6.2 Prime Polynomials over Composite Galois Fields

Prime and, in particular, primitive polynomials over composite Galois fields are needed to apply the idea of the previous section.

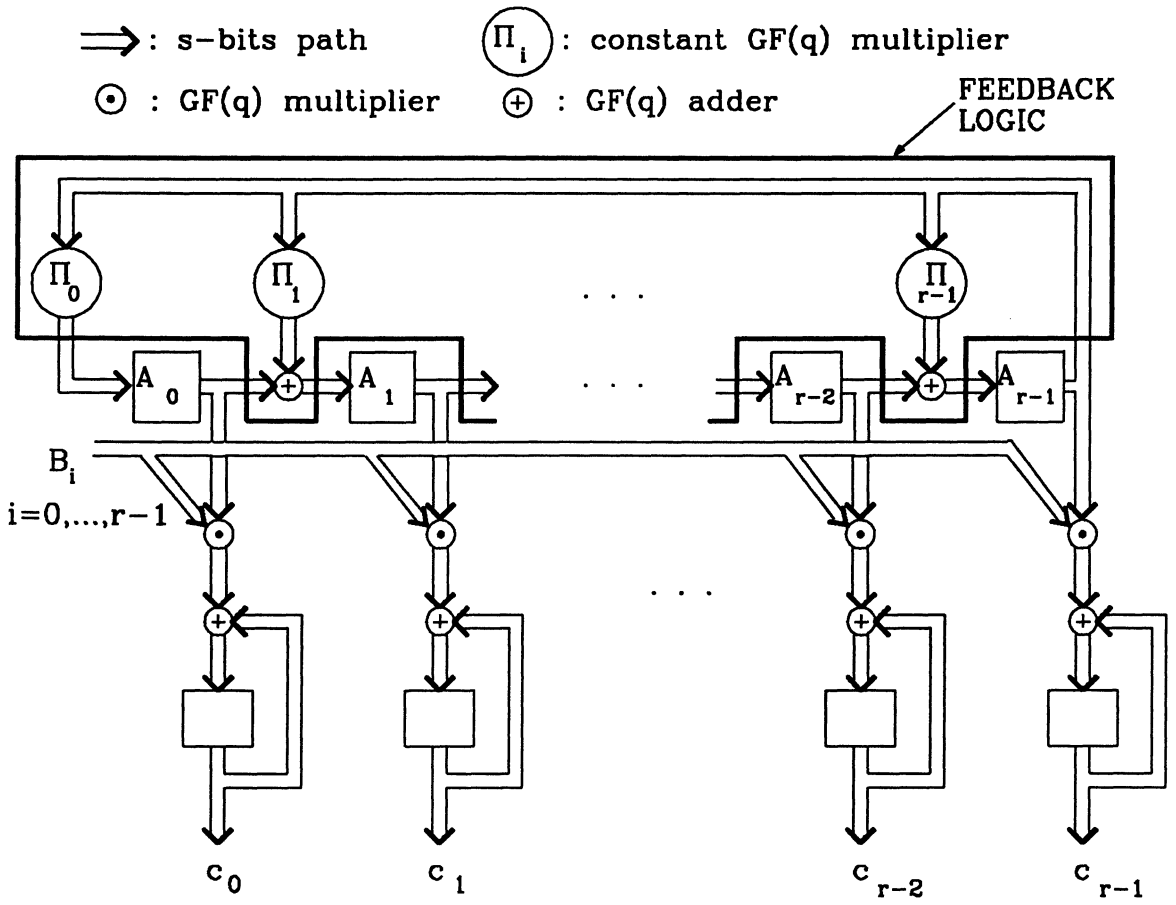


FIGURE 6.1 The general hybrid multiplier over $GF(q^r)$, $q=2^s$. All registers are s-bits wide.

In order to simplify implementation one should choose prime polynomials of low Hamming weight whose nonzero coefficients are simple to multiply by in a bit-parallel fashion. Recall from Sec. 5.1.5 that bit-parallel multiplication by a constant α^j in GF(2^m) requires $C = w_Z - m$ XOR gates where Z is the product matrix associated with the element α^j, and that its CP has length $L = \lceil \log_2 rw_Z \rceil$ where rw_Z is the maximum row weight of Z.

Some good field elements yielding low C and L are easily found:

1: the identity element yields obviously $C = L = 0$

α: we can multiply by α using an α-cell. We know (see Ch. 4 or 5) that this requires $w_P - 2$ gates and has $L = 1$.

$\alpha^{2^m-2} = \alpha^{-1}$: since $P(\alpha) = 0$ and $p_0 = 1$ we have

$$1 = \sum_{i \neq 0: p_i = 1} \alpha^i$$

or

$$\alpha^{-1} = \sum_{i \neq 0: p_i = 1} \alpha^{i-1}$$

which implies that the first column of Z has weight $w_P - 1$ while the last $m-1$ columns (representing 1, α, ..., α^{m-1}) have all weight 1. Hence, we obtain $C = w_P - 2$ and $L = 1$.

We see that multiplication by α or α^{2^m-2} requires only one XOR gate when $P(x)$ is a trinomial.

In [Gre74] a number of prime polynomials, along with their periods, over GF(4), GF(8), and GF(16) are listed. In Tab. 6.1 we show the values of C and L for all elements of GF(4), GF(8), and GF(16) where the field generators are the binary primitive polynomials $1 + x + x^2$, $1 + x + x^3$ and $1 + x + x^4$ respectively (these are the same as in [Gre74]).

Based on Tab. 6.1 and the above discussion, we selected from the lists in [Gre74] most of the primitive polynomials shown in Tab. 6.2 (only the polynomial α^{14},1,0,1 over GF(16) comes from a computer search of ours).

	C, L		
j in α^j	GF(4), $\alpha^2 = \alpha+1$	GF(8), $\alpha^3 = \alpha+1$	GF(16), $\alpha^4 = \alpha+1$
0	0,0	0,0	0,0
1	1,1	1,1	1,1
2	1,1	2,1	2,1
3		4,2	3,1
4		4,2	5,2
5		3,2	5,2
6		1,1	5,2
7			6,2
8			6,2
9			8,2
10			9,2
11			8,2
12			6,2
13			3,2
14			1,1

Table 6.1 Properties of bit-parallel multiplication by α^j in three small fields.

Degree	GF(4) $\alpha^2 = \alpha+1$	GF(8) $\alpha^3 = \alpha+1$	GF(16) $\alpha^4 = \alpha+1$
2	α,1,1	α^6,1,1	α^{14},1,1
3	α,1,1,1	α,1,0,1	α^{14},1,0,1
4	α,1,0,1,1	α^3,1,0,0,1	α^2,α,1,0,1
5	α,1,0,0,0,1	α^3,1,1,0,0,1	α^2,α,0,0,0,1
6	α,1,1,0,0,0,1	α,1,0,0,0,0,1	
7	α^2,α,1,0,0,0,0,1	α^3,α,1,0,0,0,0,1	
8	α,1,0,1,0,0,0,0,1		
9	α,1,1,0,0,0,0,0,0,1		
10	α,α,α,1,0,0,0,0,0,0,1		
11	α,1,0,0,0,0,0,0,0,0,0,1		

Table 6.2 Primitive polynomials over three small composite fields. Example: the polynomials $\alpha + y + y^3 + y^4$ and $\alpha + y + y^{11}$ are primitive over GF(4).

In Tab. 6.3 we show a few primitive polynomials over GF(32), GF(64) and GF(128) that we found by computer search.

Degree	GF(32) $\alpha^5 = \alpha^2+1$	GF(64) $\alpha^6 = \alpha+1$	GF(128) $\alpha^7 = \alpha+1$
2	α,α^{30},1	α^{62},1,1	α^{126},α,1
3	α^{30},α,0,1		

Table 6.3 Primitive polynomials over some composite fields.

6.3 Examples

In this section we present some examples of hybrid multipliers. In Tab. 6.4 we show examples for some moderately large fields while Tab. 6.5 shows examples for a few very large fields. In the tables, the conventional bit-serial MSR multiplier is found for $r = m$ and $s = 1$; it has $C = 4m$ and $L = 3$ (from Fig. 3.11). The bit-parallel approach of Ch. 4 is found for $r = 1$ and $s = m$.

The parameter R denotes the throughput of a multiplier and has been computed by the following expression

$$R = \frac{s}{LT} \tag{6.5}$$

where T is the time delay that the CP signal experiences when propagating through one level of logic (we assume that all logic levels behave the same way). In the tables we have set $T = 2$ ns.

m	r	s	C	L	R [Mbits/s]
8	8	1	32	3	167
	4	2	57	5	200
	2	4	91	6	333
	1	8	140	6	667
12	12	1	48	3	167
	6	2	83	5	200
	4	3	111	6	250
	3	4	134	6	333
	2	6	185	7	429
	1	12	351	8	750
14	14	1	56	3	167
	7	2	96	5	200
	2	7	244	7	500
	1	14	451	8	875
16	16	1	64	3	167
	8	2	113	5	200
	4	4	187	6	333
	2	8	312	8	500
	1	16	537	7	1143

Table 6.4 Examples of hybrid multipliers for some moderately large fields. The conventional serial and parallel approaches are found for $r = m, 1$ respectively.

The values of C, L and R for the cases $1 < s \leq m$ of Tab. 6.4 have been calculated with the help of Tab. 4.5 according to eq. (6.1) and (6.2).

The values of C and L for the cases $1 < s < m$ of Tab. 6.5 have been calculated according to eq. (6.3) and (6.4) with the value of C_{FB} set to $3s + 3$ (we assume a $\Pi(y)$ with $w_\Pi \leq 5$ and low-complexity coefficients). For $s = m$ we used $C = 2m^2$ and $L = 3 + \lceil \log_2 m \rceil$.

m	r	s	C	L	R [Mbits/s]
289	289	1	1156	3	167
	17	17	10730	10	850
	1	289	167000	12	12042
300	300	1	1200	3	167
	150	2	1959	5	200
	100	3	2612	6	250
	75	4	3240	6	333
	60	5	3858	8	313
	50	6	4471	7	429
	30	10	6903	9	556
	25	12	8114	10	600
	20	15	9928	8	938
	15	20	12948	10	1000
	12	25	15966	10	1250
	10	30	18983	10	1500
	6	50	31047	11	2273
	5	60	37078	10	3000
	4	75	46124	12	3125
	3	100	61200	12	4167
	2	150	91351	13	5769
	1	300	180000	12	12500
961	961	1	3844	3	167
	31	31	62534	10	1550
	1	961	1800000	13	36962

Table 6.5 Examples of hybrid multipliers for some very large fields. The conventional serial and parallel approaches are found for $r = m, 1$ respectively.

The m-values of Tab. 6.5 represent two cases where m is the square of a prime number (289 & 961) and one case where m is highly composite (300). The conventional bit-parallel multiplier is not viable in these three cases but is included in the table for purpose of comparison.

6.4 Discussion

The hybrid approach has proven powerful for designing multipliers over GF(2^m) with higher throughputs than the conventional bit-serial approach but with lower complexity than the conventional bit-parallel approach.

Accordingly, the designer gains larger freedom to adapt the complexity and performance of the multiplier to the specific requirements of the application.

The number of possible realization is large when m is highly composite. For example, Fig. 6.2 shows the throughput and complexity of all possible hybrid realizations of a multiplier over GF(2^{300}). We see that the throughput grows almost linearly with the complexity.

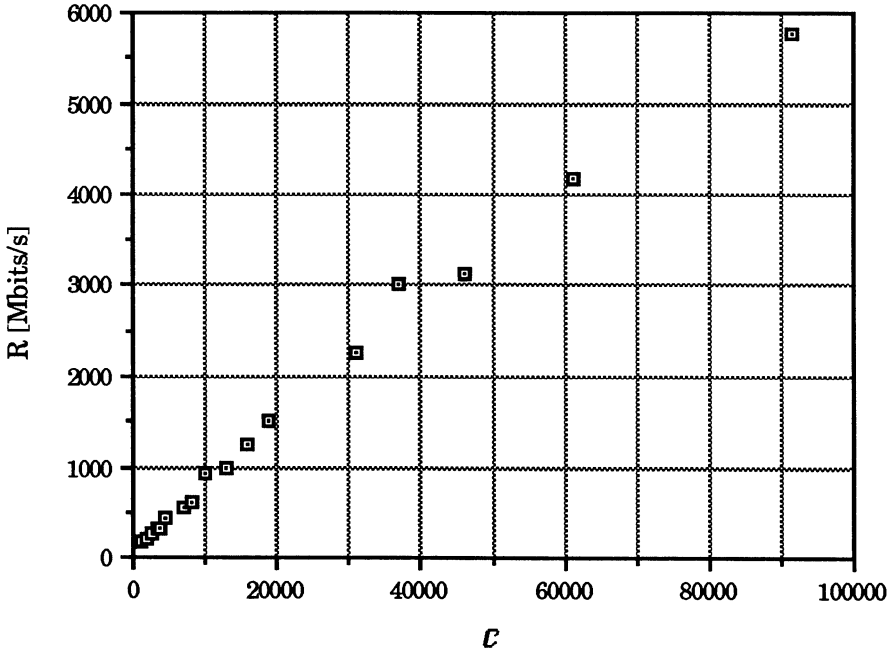


Figure 6.2 Throughput vs. complexity for different hybrid realizations of a multiplier over GF(2^{300}).

Natural applications for hybrid multiplication are the nonbinary BCH codes and the nonbinary pseudo-noise sequences.

Chapter 7

Squares and Square Roots

Squaring is an important special case of multiplication. Its importance derives primarily from the fact that squares are needed when dealing with exponentiation or inversion of field elements. While it is well-known that squaring in GF(2^m) is a linear operation — which thus can be described by a binary matrix (see for example [Ber68, pp.50-51]) — no closer analysis of the subject appears to be available. Two interesting questions that we will treat in this chapter are: if we assume a PB representation, how does the above matrix depends on $P(x)$ and, how compact can we make the squaring circuitry by a clever choice of $P(x)$? Also, we will look briefly at the inverse operation — that is, the computation of square roots.

7.1 Squaring in Polynomial Basis

Let $C(x)$ be the square of $A(x)$ modulo $P(x)$, that is

$$C(x) = A^2(x) \bmod P(x). \tag{7.1}$$

We anticipated that squaring can be described by a binary matrix, i.e.

$$C = S \cdot A \tag{7.2}$$

where the m by m binary matrix S is here called the *squaring matrix*. It is useful to derive a closed-form expression relating the squaring matrix to $P(x)$. To this end we define the following symbols

$$\hat{m} = \left\lceil \frac{m}{2} \right\rceil , \; \check{m} = \left\lfloor \frac{m}{2} \right\rfloor . \tag{7.3}$$

Further, we define a new m–$\hat{m}$–1 by m reduction matrix Q' by

$$\begin{pmatrix} x^{2\hat{m}} \\ x^{2(\hat{m}+1)} \\ \vdots \\ x^{2m-2} \end{pmatrix} = Q' \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} =$$

$$\begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,m-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,m-1} \\ \vdots & \vdots & \cdots & \vdots \\ q_{m-\hat{m}-1,0} & q_{m-\hat{m}-1,1} & \cdots & q_{m-\hat{m}-1,m-1} \end{pmatrix} \begin{pmatrix} 1 \\ x \\ \vdots \\ x^{m-1} \end{pmatrix} \mod P(x). \tag{7.4}$$

The matrix Q' has a simple relation to the reduction matrix Q of eq. (4.5): Q' consists of the $\check{m}$ rows of Q describing the reduction of the even powers of x in the range m through $2m-2$.

Proposition 7.1 Let S be the squaring matrix associated with the binary prime polynomial $P(x)$ of degree m. Then

$$S = [I \mid Q'^{\mathrm{T}}]$$

where Q'^{T} is the transpose of Q' and I is an m by $\hat{m}$ matrix with ones only at coordinates (i,j) such that $i = 2j, j = 0, 1, \ldots, \check{m}-\mu(m)$, where the binary-valued function $\mu(m)$ is defined by

$$\mu(m) = (m + 1 \mod 2). \tag{7.5}$$

Proof. By eq. (7.1) we have

$$C(x) = \sum_{k=0}^{\hat{m}-1} a_k x^{2k} + \sum_{k=\hat{m}}^{m-1} a_k [x^{2k} \bmod P(x)]. \tag{7.6}$$

From eq. (7.4) we obtain

$$x^{2k} = \sum_{n=0}^{m-1} q_{k-\hat{m},n} x^n \mod P(x), \quad k = \hat{m}, \hat{m}+1, \ldots, m-1$$

which inserted in eq. (7.6) yields

$$C(x) = \sum_{k=0}^{\hat{m}-1} a_k x^{2k} + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} \Big(\sum_{n=0}^{m-1} q_{k,n} x^n \Big) =$$

$$\sum_{k=0}^{\hat{m}-1} a_k x^{2k} + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,0} + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,1} x + \cdots + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,m-1} x^{m-1}$$

and, finally

$$C(x) = (a_0 + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,0}) + x(\sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,1}) + \cdots +$$

$$x^{2i}(a_i + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,2i}) + x^{2i+1}(\sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,2i+1}) + \cdots +$$

$$x^{2\hat{m}-2}(a_{\hat{m}-1} + \sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,2\hat{m}-2}) + \mu(m) x^{2\hat{m}-1}(\sum_{k=0}^{m-\hat{m}-1} a_{k+\hat{m}} q_{k,2\hat{m}-1}) \qquad (7.7)$$

where $\mu(m)$ is 1 for even m and 0 otherwise.
Examination of eq. (7.7) completes the proof. □

By Prop. 7.1 we can easily determine the squaring matrix of any prime polynomial, since the determination of Q' is just as simple as the determination of Q in Ch. 4.

Example 7.1 Let $P(x) = 1 + x + x^4$ generate GF(16). Then

$$\begin{pmatrix} x^4 \\ x^6 \end{pmatrix} = \begin{pmatrix} 1\,1\,0\,0 \\ 0\,0\,1\,1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ x^2 \\ x^3 \end{pmatrix}$$

and, by Prop. 7.1,

$$S = \begin{pmatrix} 1\,0\,1\,0 \\ 0\,0\,1\,0 \\ 0\,1\,0\,1 \\ 0\,0\,0\,1 \end{pmatrix}.$$

The square of A is

$$C = A^2 = S \cdot A = \begin{pmatrix} a_0 + a_2 \\ a_2 \\ a_1 + a_3 \\ a_3 \end{pmatrix}. \square$$

A squarer is clearly a device that computes m parity checks on suitable subsets of A's bits. A bit-parallel implementation has complexity

$$C_{\square} = w_S - m \tag{7.8}$$

where w_S is the Hamming weight of S. The speed of the squarer is determined by the critical-path length $L_{\square}$ given by

$$L_{\square} = \lceil \log_2 w_{S\_row} \rceil \tag{7.9}$$

where w_{S_row} is the maximum row weight of S. Both $C_{\square}$ and $L_{\square}$ can be determined directly from Q'.

Proposition 7.2 Let Q' be the reduction matrix associated with $P(x)$. Then

$$C_{\square} = w_{Q'} - \left\lfloor \frac{m}{2} \right\rfloor$$

where $w_{Q'}$ is the Hamming weight of Q'.

Proof. By Prop. 7.1 and eq. (7.8) we have

$$C_{\square} = w_I + w_{Q'} - m.$$

We need only determine the weight of I. The fact that I has ones only at coordinates (i,j) such that $i = 2j, j = 0, 1, \ldots, m-\mu(m)$, implies that the even numbered rows have weight one while the remaining rows have weight zero. In other words, I's weight increases by one every second row. Therefore, of the m rows of I, only $\left\lceil \frac{m}{2} \right\rceil$ has weight one. □

Proposition 7.3 Let Q' be the reduction matrix associated with $P(x)$. Then

$$L_{\square} = \lceil \log_2 \mathrm{MAX}\{w_{\mathrm{odd}}, w_{\mathrm{eve}} + 1\} \rceil$$

where w_{odd} is the maximum column weight among the odd numbered columns of Q' and w_{eve} the maximum column weight among the even numbered columns of Q'.

Proof. By eq. (7.9) we need the maximum row weight of S. By Prop. 7.1 the i:th row of S is the direct sum (or concatenation) of the i:th row of I and the i:th row of Q'. In the proof of Prop. 7.2 we noted that only the even numbered rows of I has nonzero weight and that their weight is 1. □

The above results, plus the fact that squaring is a special case of multiplication, imply that polynomials yielding good multipliers yield also good squarers. The following propositions are therefore of interest.

Proposition 7.4 Let $P(x) = 1 + x + x^m$ generate GF(2^m). Then

$$C_{\square} = \left\lfloor \frac{m}{2} \right\rfloor,$$

$$L_{\square} = 1.$$

Furthermore, the components of $C = A^2$ can be expressed in the following simple form

$$c_0 = a_0 + a_{\hat{m}}\mu(m)$$

$$c_{2i+1} = a_{\hat{m}+i} \quad 0 \le i \le \hat{m} - 1$$

$$c_{2i} = a_i + a_{\hat{m}+i-1+\mu(m)} \quad 1 \le i \le \hat{m} - 1,$$

where $\mu(m)$ is 1 for even m and 0 otherwise.

Proof. The complexity. By Prop. 7.2 we need only determine the weight of Q'. Clearly, all rows of Q' have weight 2. The number of rows is $m/2$ for even m and $(m-1)/2$ for odd m — that is, $\left\lfloor \frac{m}{2} \right\rfloor$ rows in general. Q' has thus weight $2 \cdot \left\lfloor \frac{m}{2} \right\rfloor$.

The CP length. The maximum column weight of Q' is 1 (cfr. Ex. 7.1). Prop. 7.3 does the rest.

The components of C are obtained by using eq. (7.7) and observing that Q' has a 1 in column 0 only for even m, and that column $j, j \ge 1$, has a 1 in row $\left\lfloor \frac{j-1+\mu(m)}{2} \right\rfloor$. □

Example 7.2 Let $1 + x + x^{15}$ generate GF(2^{15}). Then, by Prop. 7.4,

$c_0 = a_0$ $c_1 = a_8$ $c_2 = a_1 + a_8$ $c_3 = a_9$ $c_4 = a_2 + a_9$

$c_5 = a_{10}$ $c_6 = a_3 + a_{10}$ $c_7 = a_{11}$ $c_8 = a_4 + a_{11}$ $c_9 = a_{12}$

$c_{10} = a_5 + a_{12}$ $c_{11} = a_{13}$ $c_{12} = a_6 + a_{13}$ $c_{13} = a_{14}$ $c_{14} = a_7 + a_{14}$. □

Proposition 7.5 Let $P(x) = 1 + x^{m/2} + x^m$ generate GF(2^m), $m = 2 \cdot 3^l$. Then

$$\mathcal{C}_{\square} = \left\lceil \frac{m}{4} \right\rceil$$

$$\mathcal{L}_{\square} = 1.$$

Proof. The complexity. Again, we need only determine the weight of Q'. As m is even and $m/2$ odd, we see from the proof of Prop. 4.16 (the structure of Q in particular) that Q' has $(m/2 + 1)/2$ rows of weight 2 and $(m/2 - 1)/2$ rows of weight 1, i.e.

$$w_{Q'} = (\frac{m}{2}+1)\frac{1}{2}2 + (\frac{m}{2}-1)\frac{1}{2} = \frac{3m}{4} + \frac{1}{2}.$$

Prop. 7.2 and the fact that $m = 2 \cdot 3^l$ do the rest.

The CP length. From the structure of Q, see Prop. 4.16, we see that Q' has maximum column weight 1. Prop. 7.3 completes the proof. □

Proposition 7.6 Let $P(x) = 1 + x + \cdots + x^{m-1} + x^m$ generate GF(2^m). Then

$$\mathcal{C}_{\square} = m - 1 \qquad \mathcal{L}_{\square} = 1.$$

Proof. The complexity. Again, we need only determine the weight of Q'. As m is even (see Th. 2.27) we see from the proof of Prop. 4.17 (the structure of Q in particular) that Q' has one row of weight m and $(m-2)/2$ rows of weight 1, i.e.

$$w_{Q'} = m + \frac{m-2}{2} = m - 1 + \frac{m}{2}.$$

Prop. 7.2 does the rest.

The CP length. From the structure of Q, see Prop. 4.17, we see that Q' has $w_{\text{odd}} = 2$ and $w_{\text{eve}} = 1$. Prop. 7.3 completes the proof. □

Applying the same technique as in the prooves of the last three propositions one can proof the following general result for ESP's.

Proposition 7.7 Let $P(x)$ be a prime ESP of degree m and spacing s. Then

$$\mathcal{C}_{\square} = \left\lceil \frac{s}{2} \right\rceil \cdot (\frac{m}{s} - 1) \qquad \mathcal{L}_{\square} = 1. \square$$

It is easily seen that the above classes of polynomials are optimal in the sense that they yield minimum $\mathcal{L}_{\square} = 1$ ($\mathcal{L}_{\square}$ can not possibly be 0, for that would mean that squaring would be just a permutation of bits).

We have identified another class of trinomials with the above optimal property, namely the trinomials $1 + x^k + x^m$ for even k, $2 \le k < m/2$. We prove the result for $k = 2$.

Proposition 7.8 Let $P(x) = 1 + x^2 + x^m$ generate GF(2^m). Then m must be odd and we have

$$\mathcal{C}_{\square} = \left\lceil \frac{m}{2} \right\rceil \qquad \mathcal{L}_{\square} = 1.$$

Proof. The complexity. The matrix Q has the following form

$$Q = \begin{pmatrix} 1\,0\,1\,0\,0\,0 \ldots 0\,0\,0 \\ 0\,1\,0\,1\,0\,0 \ldots 0\,0\,0 \\ 0\,0\,1\,0\,1\,0 \ldots 0\,0\,0 \\ 0\,0\,0\,1\,0\,1 \ldots 0\,0\,0 \\ \vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots \ldots \vdots\;\vdots\;\vdots \\ 0\,0\,0\,0\,0\,0 \ldots 1\,0\,1 \\ 1\,0\,1\,0\,0\,0 \ldots 0\,1\,0 \end{pmatrix}$$

whereby Q' becomes

$$Q' = \begin{pmatrix} 0\,1\,0\,1\,0\,0 \ldots 0\,0\,0 \\ 0\,0\,0\,1\,0\,1 \ldots 0\,0\,0 \\ \vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots \ldots \vdots\;\vdots\;\vdots \\ 0\,0\,0\,0\,0\,0 \ldots 0\,1\,0 \\ 1\,0\,1\,0\,0\,0 \ldots 0\,1\,0 \end{pmatrix}.$$

We see that Q' has $(m-1)/2$ rows: one row has weight 3 and $(m-3)/2$ rows have weight 2 whereby $w_{Q'} = m$. Prop. 7.2 does the rest.

The CP length. We see that Q' has $w_{\text{eve}} = 1$ and $w_{\text{odd}} = 2$. Prop. 7.3 completes the proof. □

The first nine prime trinomials with $k = 2$ are [Zie68]

$m = 5, 11, 21, 29, 35, 93, 123, 333, 845.$

Through computer tests we found that

$$\mathcal{C}_{\square} = \left\lfloor \frac{m + k - 1}{2} \right\rfloor \tag{7.10}$$

for any $1 \leq k < m/2$ and that $\mathcal{L}_{\square} = 2$ for odd k, $2 < k < m/2$.

In Tab. 7.1 we show the properties of the squarers associated with the polynomials of Tab. 4.5, 5.2 and a few new polynomials.

m	$P(x)$	$\mathcal{L}_{\square}$	$\mathcal{C}_{\square}$
2	0,1,2	1	1
3	0,1,3	1	1
4	0,1,4	1	2
5	0,2,5	1	3
6	0,3,6*	1	2
	0,1,6	1	3
7	0,1,7	1	3
8	0,1,5,7,8*	2	10
	0,2,3,5,8	2	11
	0,1,3,5,8	2	14
9	0,1,9*	1	4
	0,4,9	1	6
10	0,3,10	2	6
11	0,2,11	1	6
12	0,3,12*	2	7
	0,3,5,6,12	2	20
	0,1,5,8,12	3	22
13	0,2,3,4,5,6,7,8,9,10,11,12,13	2	12
	0,1,6,7,13	2	18
14	0,5,14*	2	9
	0,3,5,6,7,10,12,13,14	2	20
	0,2,7,9,14	2	25
	0,1,3,5,14	2	26
15	0,1,15	1	7
16	0,5,6,11,16	2	21
	0,1,4,6,16	3	35

Table 7.1 Properties of parallel squarers for some prime polynomials of degree $m \leq 16$. Non-primitive polynomials are marked by *. A primitive polynomial of minimum $\mathcal{L}_{\square}$ and/or $\mathcal{C}_{\square}$ is given for each degree.

The figures in Tab. 7.1 do not account for eventual residual symmetries (in the squaring matrices) that could be used to further reduce the complexity. For each degree, a primitive polynomial of minimum complexity and/or minimum CP length is included in the table.

We see in Tab. 7.1 that a primitive polynomial with $\mathcal{C}_{\square} \leq 2m$ is found for all $m \leq 16$. Actually, if we exclude $m = 12$, we can find a primitive polynomial with $\mathcal{C}_{\square} \leq 1.5m$ in that range of degrees.

Finally we point out that Prop. 7.2 and 7.3 can be used for selecting good field generators out of a given set or, for interrupting a computer search for prime/primitive polynomials of a certain degree (normally a degree for which prime/primitive trinomials do not exist).

7.2 Square-Rooting in Polynomial Basis

The square root of an element $A \in \mathrm{GF}(2^m)$ is always in $\mathrm{GF}(2^m)$, since either $A^{1/2}$ or $A^{2^m/2}$ is in $\mathrm{GF}(2^m)$ (see Th. 2.18 for fields with odd characteristic). By eq. (7.2) we have

$$\sqrt{A} = S^{-1}A. \tag{7.11}$$

The presence of the inverse of S in eq. (7.11) does not allow us to find a general, explicit formula relating A to Q' (i.e. to $P(x)$). Though, we can derive such an expression for the special case $P(x) = 1 + x + x^m$.

Proposition 7.9 Let $1 + x + x^m$ be the generator of $\mathrm{GF}(2^m)$ and let $A = \sqrt{C}$. Then

$$a_0 = c_0 + c_1\,\mu(m)$$

$$a_i = c_{2i+\mu(m)-1} + c_{2i+\mu(m)} \quad 1 \le i \le \hat{m} - 1$$

$$a_i = c_{2(i-\hat{m})+1} \quad \hat{m} \le i \le m - 1$$

where $\mu(m)$ is 1 for even m and 0 otherwise. Clearly, we have also

$$\mathcal{C}_{\sqrt{\ }} = \left\lfloor \frac{m}{2} \right\rfloor \qquad \mathcal{L}_{\sqrt{\ }} = 1.$$

Proof. Follows directly from the expressions for c_i given in Prop. 7.4. □

Example 7.3 Let $1 + x + x^{15}$ generate $\mathrm{GF}(2^{15})$. Then, by Prop. 7.9,

$a_0 = c_0$ $\quad a_1 = c_1 + c_2$ $\quad a_2 = c_3 + c_4$ $\quad a_3 = c_5 + c_6$ $\quad a_4 = c_7 + c_8$

$a_5 = a_9 + c_{10}$ $\quad a_6 = c_{11} + c_{12}$ $\quad a_7 = c_{13} + c_{14}$ $\quad a_8 = c_1$ $\quad a_9 = c_3$

$a_{10} = c_5$ $\quad a_{11} = c_7$ $\quad a_{12} = c_9$ $\quad a_{13} = c_{11}$ $\quad a_{14} = c_{13}$. □

The inverse of a binary matrix can be computed very quickly by a modern computer, even for fairly large dimensions. We checked therefore many polynomials and determined the properties of some interesting classes of field generators. This was done by means of a Pascal program that computes S and S^{-1} for given $P(x)$. The results are shown in Tab. 7.2.

We see that the trinomial $1 + x^{m/2} + x^m$ is still best followed by $1 + x + x^m$. The all-ones polynomials have also nice properties. Furthermore, we noticed that trinomials with high k values have surprisingly good properties.

$P(x)$	$L_{\sqrt{\ }}$	$C_{\sqrt{\ }}$
$1 + x^{m/2} + x^m$	1	$\lceil \frac{m}{4} \rceil$
$1 + x + x^m$	1	$\lfloor \frac{m}{2} \rfloor$
$1 + x^{m-1} + x^m$	1	$m-1$
$1 + x + \cdots + x^{m-1} + x^m$	1	$m-1$
$1 + x^k + x^m$, $2 \le k < m-1$		
m, k odd	1	$\lfloor \frac{m}{2} \rfloor$
m odd, k even	$\le \lceil \log_2 \frac{m}{2} \rceil$	*
m even, k odd	2	$\lfloor \frac{m+k-1}{2} \rfloor$
m, k even	**	**

Table 7.2 Properties of parallel square-rooters for some classes of generator polynomials. All results obtained through computer tests. *) In this case we have not been able to extrapolate a formula for complexity; an upper bound on $C_{\sqrt{\ }}$ for $m \le 30$ is $5m$. However, the higher the k the better the properties. **) In this case $1 + x^k + x^m$ is reducible.

In Tab. 7.3 we show the properties of the polynomials of Tab. 7.1 plus those of a few new polynomials. Our investigations indicate that, except for most trinomials, polynomials yielding good squarers are likely to yield bad square-rooters and viceversa (see for example the cases $m = 12, 14, 16$). Also, we noted that there are normally more polynomials with $L_{\sqrt{\ }} \le 2$ than polynomials with $L_{\square} \le 2$ (this fact became very evident in our exhaustive searches for degrees ≥ 12).

The figures of Tab. 7.3 also show that there exists a primitive polynomial of degree ≤ 16 with $C_{\sqrt{\ }} \le 1.5m$.

m	$P(x)$	$L_{\sqrt{}}$	$C_{\sqrt{}}$
2	0,1,2	1	1
3	0,1,3	1	1
4	0,1,4	1	2
5	0,2,5	2	5
6	0,3,6*	1	2
	0,1,6	1	3
7	0,1,7	1	3
8	0,1,5,7,8*	2	15
	0,1,2,7,8	2	9
	0,1,3,5,8	2	13
	0,2,3,5,8	2	19
9	0,1,9*	1	4
	0,5,9	1	4
	0,4,9	2	10
10	0,3,10	2	6
11	0,9,11	1	5
	0,2,11	3	20
12	0,3,12*	2	7
	0,1,2,8,12	2	18
	0,1,5,8,12	3	26
	0,3,5,6,12	3	40
13	0,2,3,4,5,6,7,8,9,10,11,12,13	2	12
	0,1,6,7,13	2	21
14	0,5,14*	2	9
	0,4,5,8,10,11,14	2	19
	0,2,7,9,14	3	45
	0,3,5,6,7,10,12,13,14	3	54
	0,1,3,5,14	3	56
15	0,1,15	1	7
16	0,5,10,11,16	2	19
	0,1,4,6,16	2	24
	0,5,6,11,16	3	50

Table 7.3 Properties of parallel square-rooters for some prime polynomials of degree $m \leq 16$. Non-primitive polynomials are marked by *. A primitive polynomial of minimum $C_{\sqrt{}}$ and/or $L_{\sqrt{}}$ is given for each degree.

7.3 Squares and Square Roots in Normal Basis

The major feature of the NB representation of a Galois field is that squaring and square-rooting become very simple operations. Indeed, let A be an element of GF(2^m), i.e. $A = a_0\alpha + a_1\alpha^2 + \cdots + a_{m-1}\alpha^{2^{m-1}}$.

Then

$$A^2 = (a_0\alpha + a_1\alpha^2 + \cdots + a_{m-1}\alpha^{2^{m-1}})^2 = (a_0\alpha^2 + a_1\alpha^4 + \cdots + a_{m-1}\alpha^{2^m}).$$

Since $\alpha^{2^m} = \alpha$ we have

$$A^2 = a_{m-1}\alpha + a_0\alpha^2 + \cdots + a_{m-2}\alpha^{2^{m-1}},$$

which means that squaring is a simple cyclic shift of the element's bits. Square-rooting is obviously a cyclic shift in the opposite direction, i.e.

$$\sqrt{A} = a_1\alpha + a_2\alpha^2 + \cdots + a_0\alpha^{2^{m-1}}.$$

7.4 Discussion

In this chapter we have established that squaring, in polynomial basis, can often be accomplished by a simple and very fast combinational network. For trinomials we found that squaring requires between $\frac{m}{4}$ and m binary additions (i.e. XOR gates). As noted in previous chapters, there are plenty of prime trinomials [Zie68] [Zie69]. Tab. 7.4 summarizes the main results about squaring.

For some of the fields where the results of Tab. 7.4 can not be applied, we found that squaring can be performed in at most 1.5 m binary additions, (see Tab. 7.1). In these cases the best polynomials were found by computer search. A prime polynomial that yields simple squaring circuits shall normally have low Hamming weight. This follows naturally from the fact that the squaring matrix S consists of an identity-like matrix and the new reduction matrix Q' which, in turn, consists of the even numbered rows of the product matrix Q of Ch. 4.

$P(x)$	$L_{\square}$	$C_{\square}$
$1 + x^{m/2} + x^m$	1	$\lceil \frac{m}{4} \rceil$
$1 + x + x^m$	1	$\lfloor \frac{m}{2} \rfloor$
$1 + x + \cdots + x^{m-1} + x^m$	1	$m-1$
$1 + x^k + x^m$, $2 \le k < m/2$		
m odd, k even	1	$\lfloor \frac{m+k-1}{2} \rfloor$
k odd	2	$\lfloor \frac{m+k-1}{2} \rfloor$

Table 7.4 Properties of parallel squarers for some classes of generator polynomials.

For large m, the implementation of the squarer as a fast combinational network can be accomplished either using an XOR-PLA or through a module generator. Using a module generator is simple since we have closed-form expressions (eq. (7.7)) for the m boolean functions defining the components of the square. These boolean functions could be generated by a simple computer program and input (in a suitable format) to the module generator together with a set of directives for optimization and manipulation of the functions, the circuit realization and the layout [Pre88, Ch. 7]. This possibility is of particular importance when m is large.

For small to moderately large m, the combinational network can alternatively be handcrafted. The trinomial $1 + x + x^m$ yields a particularly nice and regular structure as shown in Fig. 7.1 for $m = 15$ (Ex. 7.2).

Squarers, for the PB representation, can be implemented as sequential devices if the field generator is a trinomial (see also [Bet87] [Gom90)]. These squarers require m registers, a switch and a few gates (they are thus more complex than the corresponding parallel squarers discussed here). Fig. 7.2 shows a sequential version of the squarer of Fig. 7.1.

Square roots are not frequently used in applications of Galois fields. Nevertheless, we investigated briefly the computation of square roots (in PB) by fast combinational networks. Due to the presence of a matrix inversion in the calculations of the square root, the investigation had to be based on computer tests. The results showed that square-rooting is normally as simple as squaring — that is, the square root can often be computed in at most $1.5m$ binary additions. However, we also found that, except for most trinomials, the best square-rooting polynomials are likely to be unsuitable for squaring and viceversa. Tab. 7.5 shows some primitive polynomials of degree ≤ 16 with good properties in both cases.

Fig. 7.3 shows a bit-parallel and a bit-serial version of the square-rooting circuit based on Ex. 7.3. Notice that the combinational network of the parallel circuit is identical to that of the squarer of Fig. 7.1. This feature applies in general to $1 + x + x^m$ and implies that the squaring circuit can be used also for square-rooting.

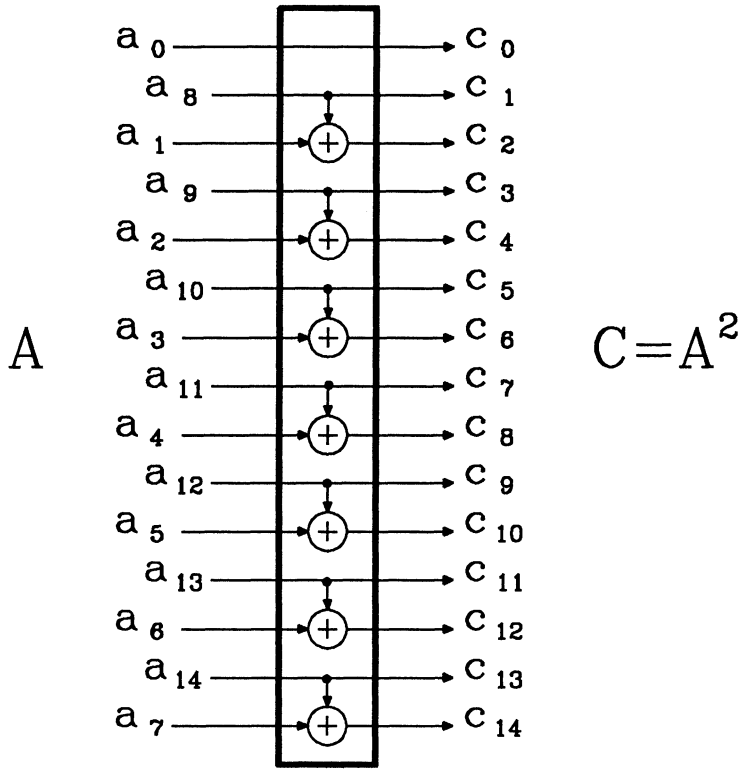


FIGURE 7.1 A parallel squarer for $P(x)=1+x+x^{15}$.

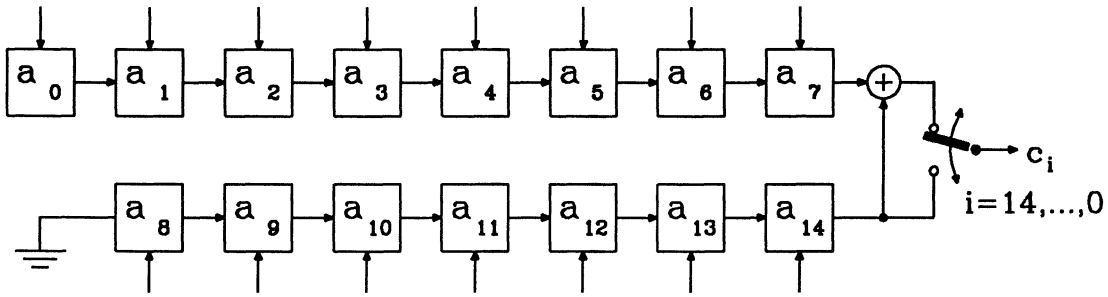
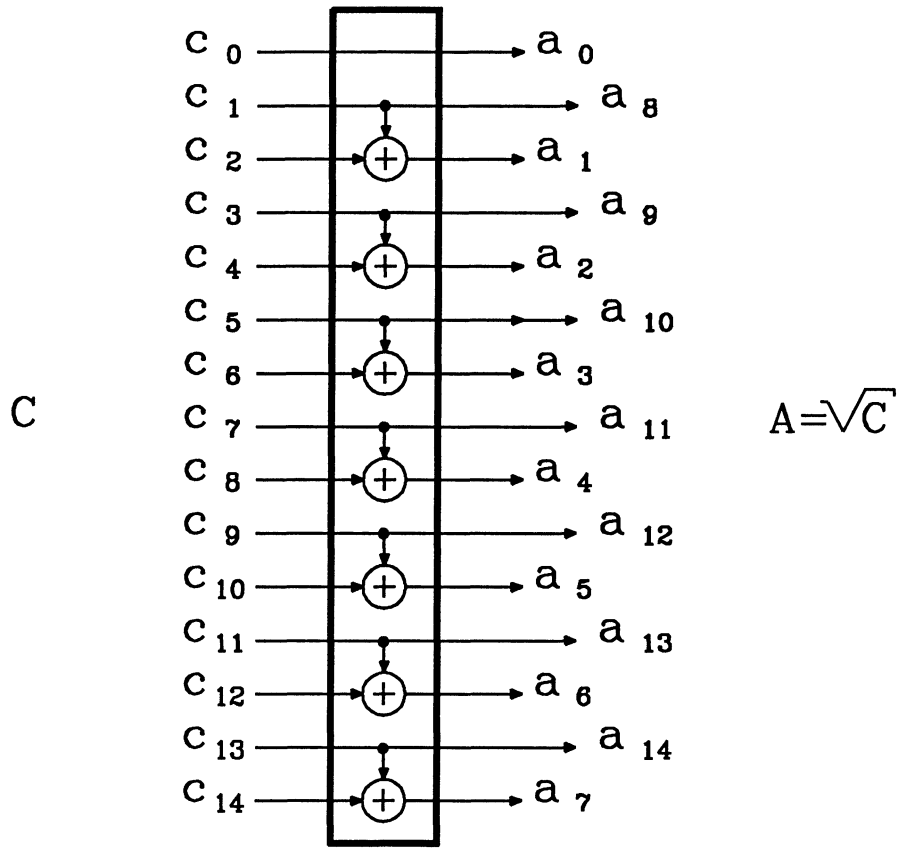
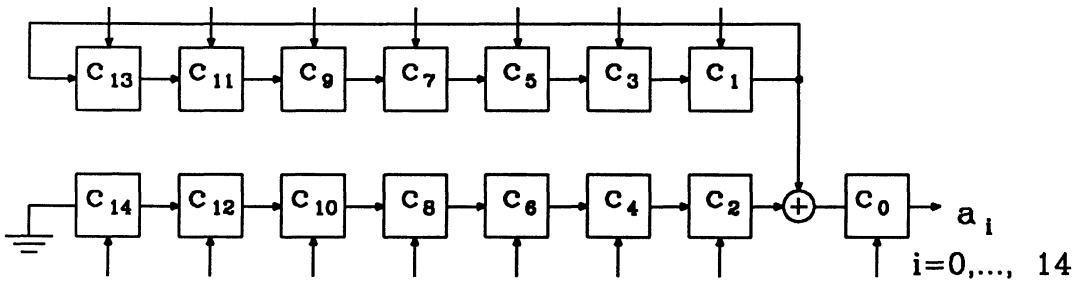


FIGURE 7.2 A bit-serial squarer for $P(x)=1+x+x^{15}$.



(a)



(b)

FIGURE 7.3 A parallel (a) and serial (b) square-rooter for GF(2^{15}) with $P(x)=1+x+x^{15}$.

m	$P(x)$	$L_{\square}$	$C_{\square}$	$L_{\sqrt{}}$	$C_{\sqrt{}}$
2	0,1,2	1	1	1	1
3	0,1,3	1	1	1	1
4	0,1,4	1	2	1	2
5	0,3,5	2	3	1	2
6	0,1,6	1	3	1	3
7	0,1,7	1	3	1	3
8	0,1,2,7,8	2	15	2	9
9	0,5,9	2	6	1	4
10	0,3,10	2	6	2	6
11	0,2,5,8,11	2	11	2	13
12	0,1,4,6,12	3	23	2	18
13	0,2,3,4,5,6,7,8,9,10,11,12,13	2	12	2	12
14	0,1,3,4,5,7,8,10,11,13,14	2	21	2	22
15	0,1,15	1	7	1	7
16	0,5,10,11,16	3	37	2	19

Table 7.5 Some primitive polynomials suitable for both squaring and square-rooting.

Clearly, squaring/square-rooting is not as easy in PB as in NB. The additional complexity required by the PB squarers/square-rooters can though be made small by a clever choice of field generator. In general, a PB squarer/square-rooter will not display a regular structure but this drawback is largely compensated for by the low number of gates to be handled.

Furthermore, the availability of simple closed-form expressions defining the boolean functions of the squarer allows the design to be automatic to a large extent.

Hence, we conclude that the NB representation is not an obvious choice in applications requiring squares and/or square roots, unless the application at hand makes frequent use of these operations and very little use of other more complex operations (e.g. multiplication). However, even in such applications both representations should be considered and a comparison performed, before a choice is made.

Chapter 8

Exponentiation

In this chapter we consider the computation of expressions of the following form,

$$\gamma = \alpha^e \qquad 0 \le e \le 2^m - 2 \tag{8.1}$$

where α is a primitive element of GF(2^m). Eq. (8.1) defines the *exponentiation function* over the Galois field GF(2^m) in an unambiguous way since we have a one-to-one correspondence between e and γ due to the fact that α is primitive. Henceforth we call α the *base* and e the *exponent*.

Let e be in binary form, i.e.

$$e = e_0 2^0 + e_1 2^1 + \cdots + e_{m-2} 2^{m-2} + e_{m-1} 2^{m-1}.$$

Then, by eq. (8.1) we have

$$\alpha^e = \alpha^{e_0} \alpha^{e_1 2} \cdots \alpha^{e_{m-2} 2^{m-2}} \alpha^{e_{m-1} 2^{m-1}} \tag{8.2}$$

or, equivalently

$$\alpha^e = (\cdots ((\alpha^{e_{m-1}})^2 \alpha^{e_{m-2}})^2 \alpha^{e_{m-3}} \cdots \alpha^{e_1})^2 \alpha^{e_0}. \tag{8.3}$$

Exponents can thus be computed by repeated square-and-multiply (S & M).

It is interesting to notice that eq. (8.2) has been known for several centuries (eq. (8.3) is a simple circumscribing of eq. (8.2)). Yet, as far as we know, no substantially better ways to compute α^e have been found since then. Consequently, the above equations appear still to be the most suitable for implementation purposes.

In the following sections we consider a number of different architectures, some of which appear to be reported for the first time, for computing α^e according to the above equations. Since squaring and multiplication are the required operations we will consider both the PB (suitable for multiplication) and NB (suitable for squaring) representation of GF(2^m). An architecture mixing the DB and PB representations can be found in [Bet87].

8.1 Exponentiation with Fixed Base

We start off by considering the simpler case when the base α is a constant. In particular, we let α be a root of the primitive polynomial $P(x)$ generating the field GF(2^m).

8.1.1 Stored Conjugates

In eq. (8.2) we see that γ is the product of a subset of α's conjugates. Since α is fixed and known in advance, its conjugates can be precomputed, stored in circulating registers or in a ROM (Read Only Memory) and multiplied together according to the exponent [Sco88].

The multiplication can be performed in a parallel fashion by a tree consisting of $m-1$ multipliers as shown in Fig. 8.1 for $m = 8$. If we adopt bit-parallel multipliers (Ch. 4) we obtain γ very quickly but the multiplier tree has complexity $\sim 2m^3$ (PB) or $\sim 3m^3$ (NB) which is acceptable only for small values of m. The device storing the conjugates has complexity $\sim m^2$.

Alternatively, we can adopt pipelined bit-serial multipliers (Ch. 3) and pipeline the operation of the whole exponentiator so to obtain a rate of one exponentiation per m clock cycles. This rate presupposes continuous operation. At discontinuous operation parts of the pipeline will be empty from time to time and an additional (variable) delay will be incurred. The complexity of the multiplier tree is now $\sim 4m^2$ (PB) or $\sim 5m^2$ (NB).

Processor-time tradeoffs can be made in the above architecture. Fig. 8.2 shows, for $m = 8$, the opposite extreme where we have reduced the number of multipliers to 1. The single multiplier of Fig. 8.2 can, up to moderate values of m, be a fast, bit-parallel multiplier whereby we obtain a rate of one exponentiation per m clock cycles. For large m only a bit-serial multiplier is viable and the rate becomes one exponentiation per m^2 clock cycles. In the former case the complexity of the multiplier ($\mathcal{C} \sim 2m^2$ in PB or $\mathcal{C} \sim 3m^2$ in NB) is of the same order of magnitude of the storing device ($\mathcal{C} \sim m^2$) while in the latter case the complexity is dominated by the storing device (the multiplier has $\mathcal{C} \sim 4m$ in PB and $\mathcal{C} \sim 5m$ in NB).

Solutions lying between the above extreme cases — i.e. where a subset of the full multiplier tree is used — are also possible [Sco88].

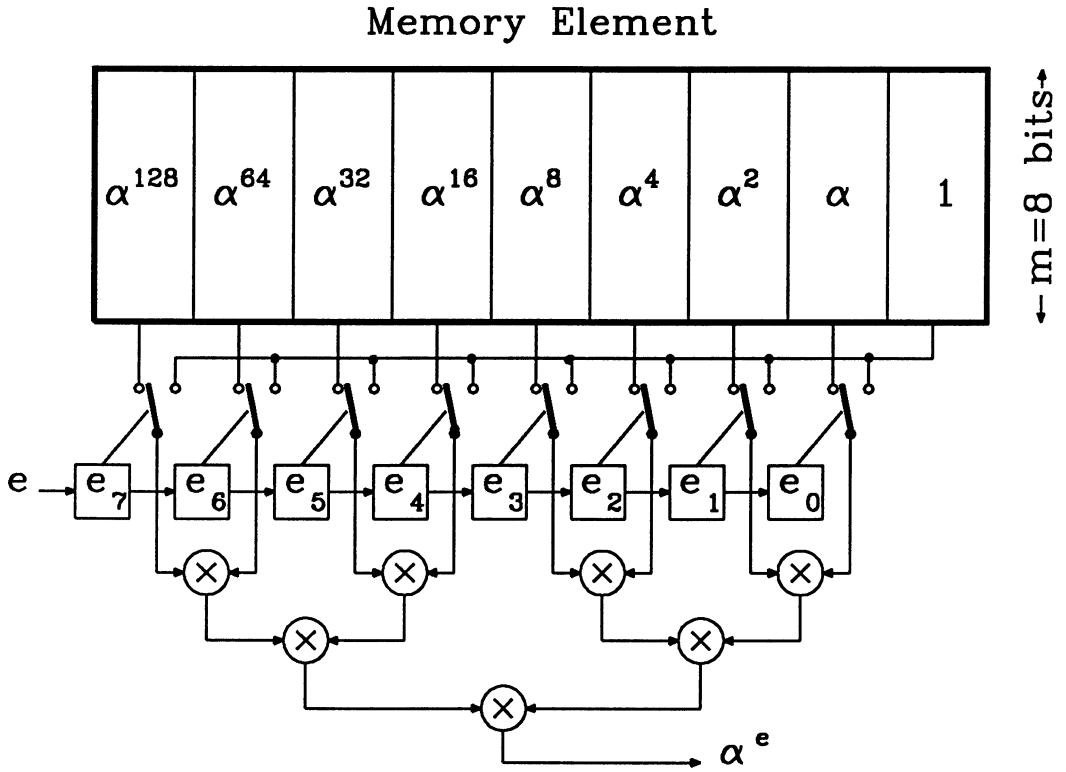


FIGURE 8.1 Exponentiator with stored conjugates and full multiplier tree for m=8.

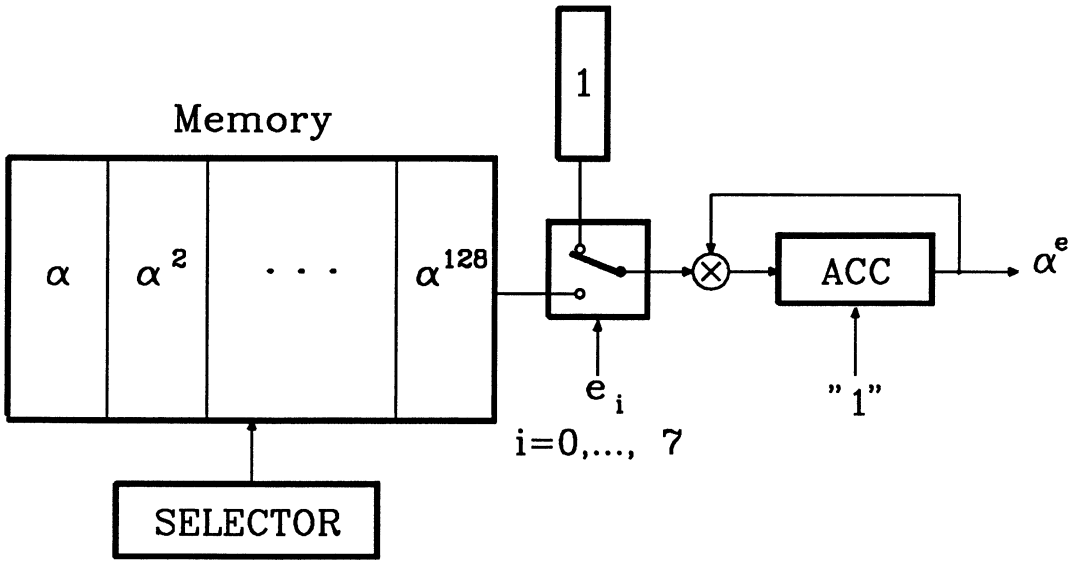


FIGURE 8.2 Exponentiator with stored conjugates and one multiplier for m=8.

8.1.2 Consecutive S & M with Bit-Parallel Squaring

The fact that the base α is fixed and known can be utilized in another way through eq. (8.3). There γ is obtained by repeated squaring and multiplication by α. The multiplication by α is required in step i only if bit $e_i = 1$. The general structure of the exponentiator based on eq. (8.3) is shown in Fig. 8.3. Notice that the operation is pipelined and that all input/output is bit-serial while the internal operation is bit-parallel. Also, in a practical realization, the two buses going back to the top of the exponentiator would be routed *through* the α-cell and the squarer instead of running on the sides.

The rate is one exponentiation per m clock cycles.

Polynomial Basis

As we saw in Ch. 4, multiplication by α in PB is a very simple operation that requires only $w_P - 2$ binary additions, and can be implemented by the α-cell introduced in Ch. 4 (see e.g. the α-cell of Fig. 4.3). Hence, the implementation of the multiplication by α can be considered an easy task even for large m.

Squaring was discussed in Ch. 7 where we found that whenever we can choose $P(x) = 1 + x^k + x^m$, $k < m/2$, only $\left\lfloor \frac{m+k-1}{2} \right\rfloor \le \frac{3m}{4}$ binary adders are required to implement a fast bit-parallel squarer. For other choices of $P(x)$ we found that the number of binary adders is normally at most $2m$.

Let C denote the complexity of the squarer/α-cell pair. Then

$$C \le \frac{3m}{4} + w_P - 2\,, \quad P(x) = 1 + x^k + x^m, k \le m/2 \tag{8.4}$$

$$C \le 2(m-1) + w_P\,, \quad w_P \ge 5. \tag{8.5}$$

Let L denote the length of the critical path through the squarer/α-cell pair. In Ch. 7 we found that squarers based on trinomials had a CP length ≤ 2. For degrees ≤ 16 for which no prime trinomials exist, we could still find polynomials with $L_\square \le 2$. Therefore, we can assume that it is likely to find a field generator with $L_\square \le 2$ also for most degrees > 16. Then

$$L = 1 + d\,, \quad 1 \le d \le 2. \tag{8.6}$$

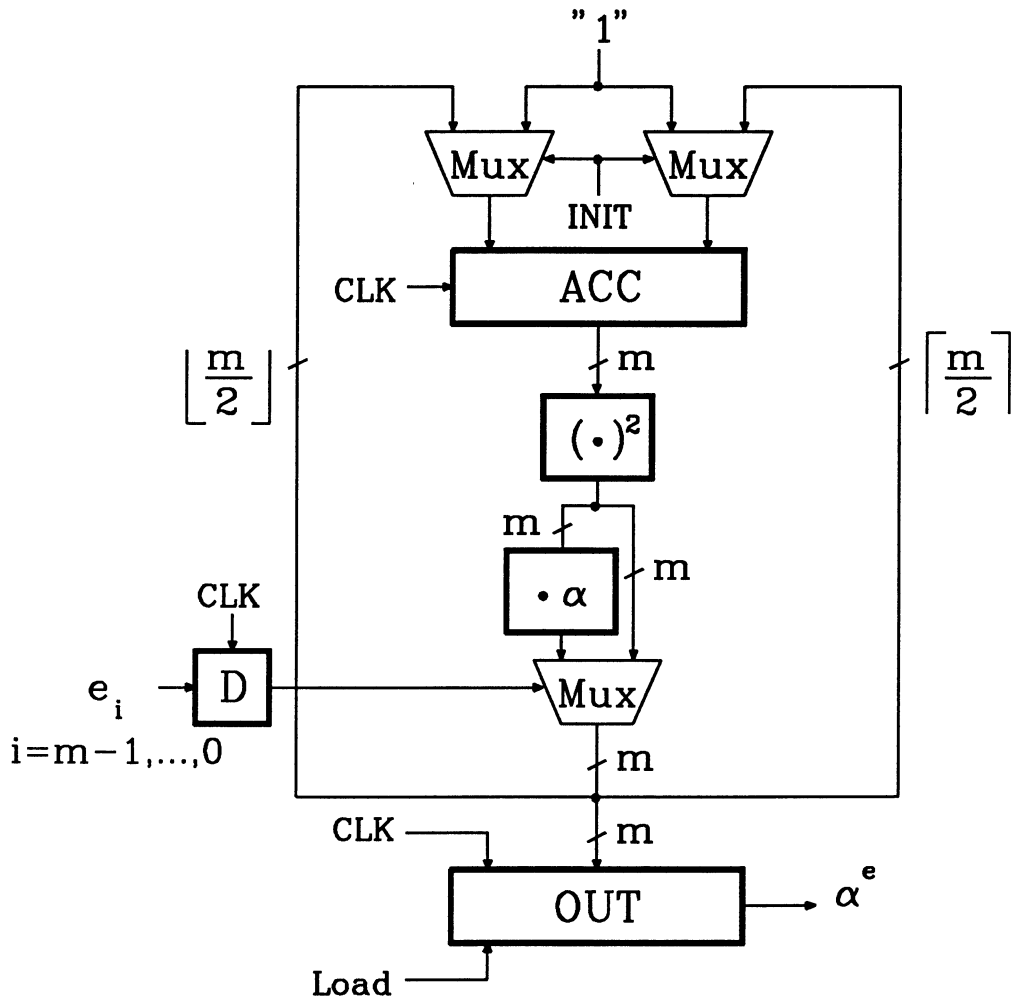


FIGURE 8.3 General structure of a fast exponentiator over GF(2^m) for fixed base α.

Normal Basis

In NB, squaring is a cyclic shift which, in our case, is readily hardwired. Multiplication by α is more complicated.

Let A and B be two field elements in an NB representation, and let C denote their product (also in NB). Then, by eq. (3.22), the product bits c_i can be computed by

$$c_i = f(\mathrm{CS}^{m-i-1}(A), \mathrm{CS}^{m-i-1}(B)) \quad i = 0, 1, \ldots, m-1$$

where $\mathrm{CS}^n(A)$ is the n:th right cyclic shift of A. In NB, multiplication by α is the same as multiplication by $B = (1, 0, \ldots, 0)$ (which is not the identity element). Then, by eq. (3.25), the product bits c_i are given by

$$c_i = \mathrm{CS}^{m-i-1}(A) M \mathrm{CS}^{m-i-1}(1, 0, \ldots, 0), \quad i = 0, 1, \ldots, m-1 \tag{8.7}$$

or

$$c_i = \mathrm{CS}^{m-i-1}(a_0, a_1, \ldots, a_{m-1}) \cdot (m_{0,m-i-1}, m_{1,m-i-1}, \ldots, m_{m-1,m-i-1}),$$
$$i = 0, 1, \ldots, m-1 \tag{8.8}$$

where $\cdot$ denotes inner product and $(m_{0,j}, m_{1,j}, \ldots, m_{m-1,j})$ is the j:th column of M. Clearly, each c_i is a modulo-2 sum of a-coefficients. Totally, the computation of αA requires $N_m - m$ binary additions where N_m was defined in Sec. 3.3 as the number of terms in f which, in turn, is the same as the Hamming weight of M.

Since $N_m \geq 2m - 1$ the complexity of the NB squarer/α-cell pair is

$$\mathcal{C} \geq m - 1 \tag{8.9}$$

with equality only for optimal normal bases. The CP length $\mathcal{L}$ of the NB α-cell is easily shown to be 1 for optimal NB and at least 2† for non-optimal NB, i.e.

$$\begin{cases} \mathcal{L} = 1 & \text{for optimal NB} \\ \mathcal{L} \geq 2 & \text{otherwise.} \end{cases} \tag{8.10}$$

† Although we do not have a formal proof for it, we have checked many NB's and observed that N_m is always odd which implies that $N_m \geq 2m + 1$ for non-optimal NB. This, in turn, implies that the matrix M must have at least one column of weight 3 (and that equality in eq. (4.20) is valid only for optimal NB).

Tab. 8.1 shows, for both representations, the values of C and L for the best choice of primitive polynomials of degree ≤ 16 (the polynomials are found in Tab. 3.10 and 7.1). Tab. 8.2 shows the corresponding figures for five large values of m, four of which are Mersenne exponents (i.e. $2^m - 1$ is prime).

m	C_{PB}	C_{NB}	L_{PB}	L_{NB}
2	2	1	2	1
3	2	2	2	1
4	3	5	2	2
5	4	4	2	1
6	4	5	2	1
7	4	12	2	2
8	14	13	3	2
9	7	8	2	1
10	7	27	3	3
11	7	10	2	1
12	23	29	3	2
13	21	32	3	2
14	27	13	3	1
15	8	38	2	2
16	24	69	3	3

Table 8.1. The properties of the PB/NB squarer/α-cell pair of Fig. 8.3 for the best choice of primitive field generators of degree ≤ 16 (see Tab. 3.10 and 7.1 for the polynomials).

m	$P(x)$ for PB	C_{PB}	C_{NB}	L_{PB}	L_{NB}
127	0,1,127	64	374	2	≥ 2
333	0,2,333	168	> 332	2	≥ 2
521	0,32,521	277	≤ 16150	2	≥ 2
607	0,105,607	356	3014	3	≥ 2
1279	0,216,1279	748	≤ 11510	2	≥ 2

Table 8.2. The properties of the PB/NB squarer/α-cell pair of Fig. 8.3 for some field generators of large degree. 127, 521, 607 and 1279 are Mersenne exponents. For these four degrees we used the list of low-complexity NB given in Appendix B. No optimal NB exist for the degrees in the table. The trinomials listed in the table are all primitive [Zie68] [Zie69].

Fig. 8.4a shows a PB exponentiator over GF(16) with $P(x) = 1 + x + x^4$ while Fig. 8.4b shows the corresponding exponentiator for the NB consisting of the roots of the primitive trinomial $1 + x^3 + x^4$ ($N_m = 9$).

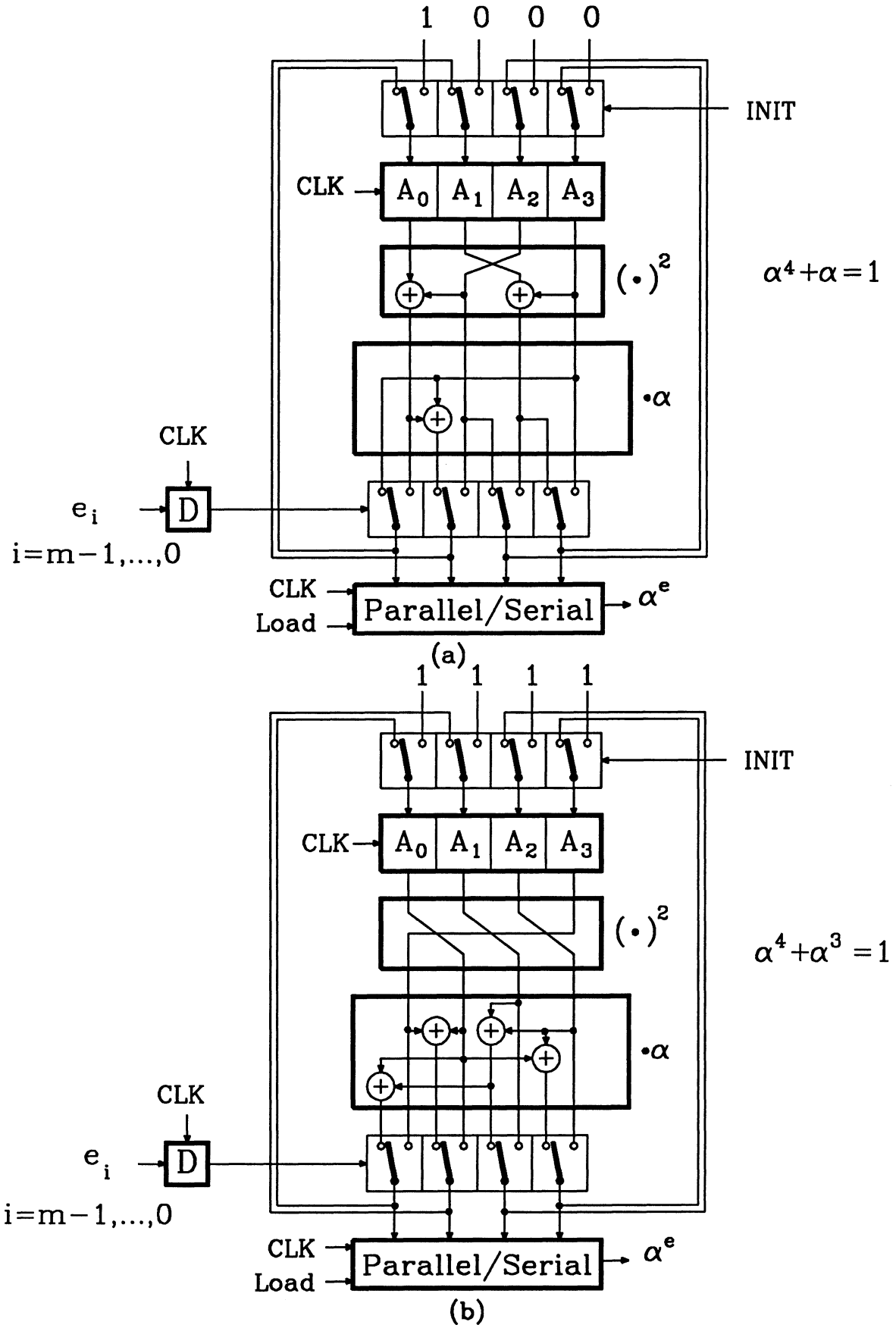


FIGURE 8.4 Fast exponentiator for fixed base α over GF(16) in PB (a) and NB (b).

In the design of the NB α-cell one gate was saved due to symmetries in the expressions for the c_i's of eq. (8.8).

8.1.3 Consecutive S & M with Squaring Multiplier

This is a variation of the PB architecture of the previous section where the parallel squarer is replaced by a generic sequential multiplier. For example, the MSR multiplier of Sec. 3.1 is well suited for this purpose. Each squaring takes now m clock cycles whereby the rate is reduced to one exponentiation per m^2 clock cycles.

The major advantage of this architecture is the regular structure that makes it easily expanded to any (large) value of m.

The complexity is dominated by the MSR multiplier whose complexity is $4m$. The performance is limited mainly by the performance of the MSR multiplier whose CP has length 3 and to which we must add 1 to account for the CP of the α-cell.

Fig. 8.5 shows an example over GF(16) with $P(x) = 1 + x + x^4$. Notice that there are only two types of cells and that these differ only by two XOR gates.

8.1.4 Exponentiation by LFSR

If time is not an issue and m small or moderately large, exponentiation can be performed by a simple Galois-type LFSR with feedback polynomial $P(x)$. The LFSR is initially loaded with the element 1 (= α^0) and then clocked $e = \sum_{i=0}^{m-1} e_i 2^i$ times. The operation is asynchronous and the average rate is one exponentiation per 2^{m-1} clock cycles.

The LFSR has complexity $m + w_P - 2$ and can be clocked very fast. A counter, controlled by e, for clocking the LFSR is also required. Further, a certain amount of buffer space might be needed for handling the queue of incoming exponents.

This low-complexity exponentiator is best suited for applications where the average time between two consecutive exponentiations is relatively large (like > 2^{m-1} clock cycles).

8.1.5 Table Look-up

For small values of m it is possible to use a table-look-up method of exponentiation. A ROM containing all nonzero field elements is addressed

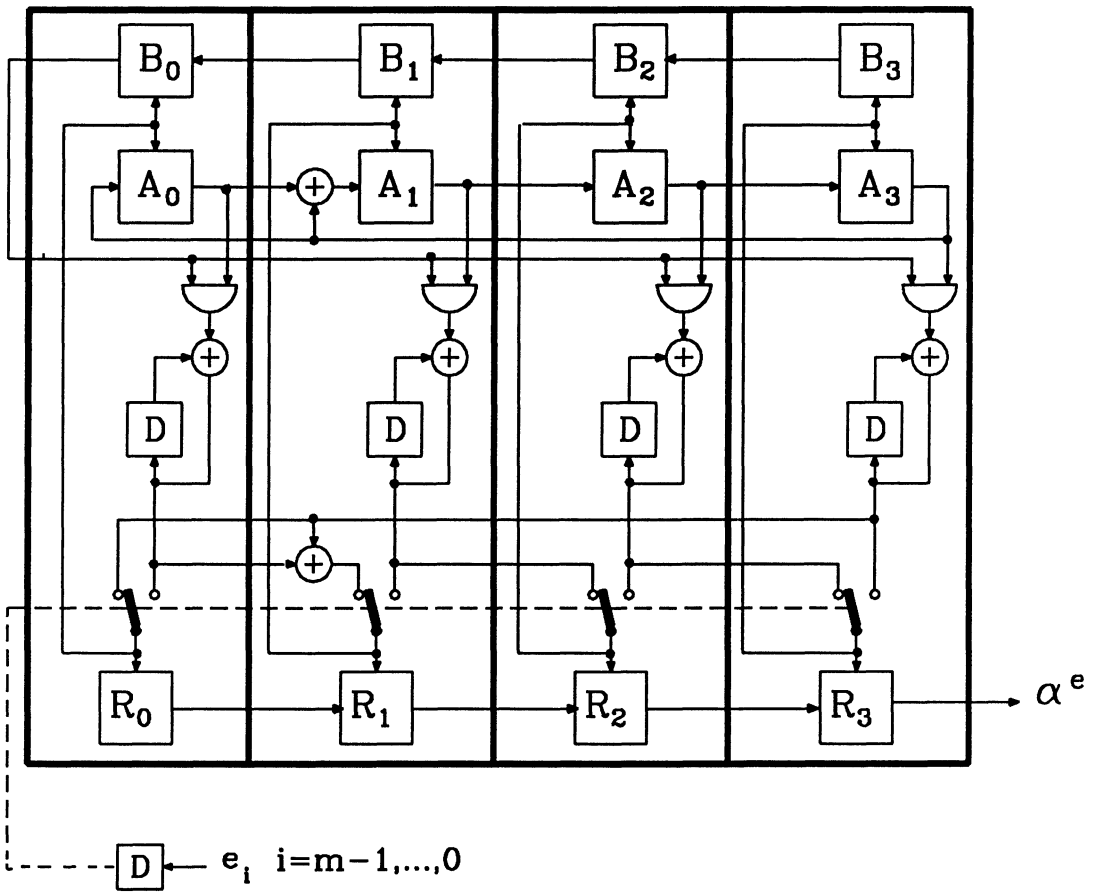


FIGURE 8.5 Exponentiator over GF(16) with squaring multiplier and α-cell for fixed base α, $(\alpha^4+\alpha=1)$. The A- and B-registers are initially loaded with (1,0,0,0).

by the exponent and a rate of one exponentiation per clock cycle is obtained. If fast bit-serial communication is desired, the memory is equipped with a serial/parallel addressing register and a parallel/serial output register.

Memories are well-studied, compact and regular structures well suited for VLSI implementation. The complexity of an m-bits wide exponentiation ROM is clearly $\sim m2^m$.

8.2 Exponentiation with Variable Base

We assume next that both the exponent *and* the base are variables. Accordingly, we can no longer perform the required multiplications by means of the simple α-cell — now we must adopt a generic multiplier.

8.2.1 Consecutive S & M

The exponentiator of Fig. 8.3 can work with variable base if we replace the α-cell with a generic multiplier over GF(2^m) as shown in Fig. 8.6.

Polynomial Basis. In PB we can use the MSR multiplier ($\mathcal{C} \sim 4m$) as generic multiplier and a bit-parallel squarer ($\mathcal{C} \sim 2m$). The base α is input serially into the MSR multiplier while the output of the squarer is input in parallel. At synchronous operation we obtain a rate of one exponentiation per m^2 clock cycles. At asynchronous operation the average rate is higher since we perform the multiplication by α in step i only if $e_i = 1$. The CP has length

$$\mathcal{L} = \mathcal{L}_{\text{MSR}} + \mathcal{L}_{\square} = 3 + \mathcal{L}_{\square}$$

which will normally be at most 5 by the results of Ch. 7.

If m is small or moderately large large we can use a bit-parallel multiplier ($\mathcal{C} \sim 2m^2$) and obtain a rate of one exponentiation per m clock cycles. The clock period would though be longer due to a CP of length

$$\mathcal{L} = d + \lceil \log_2 m \rceil + \mathcal{L}_{\square},$$

where we normally have $3 \leq d + \mathcal{L}_{\square} \leq 5$ (see Tab. 4.9 and Ch. 7).

An other interesting configuration is to use a single serial multiplier alternatively for squaring and multiplication. The rate for asynchronous

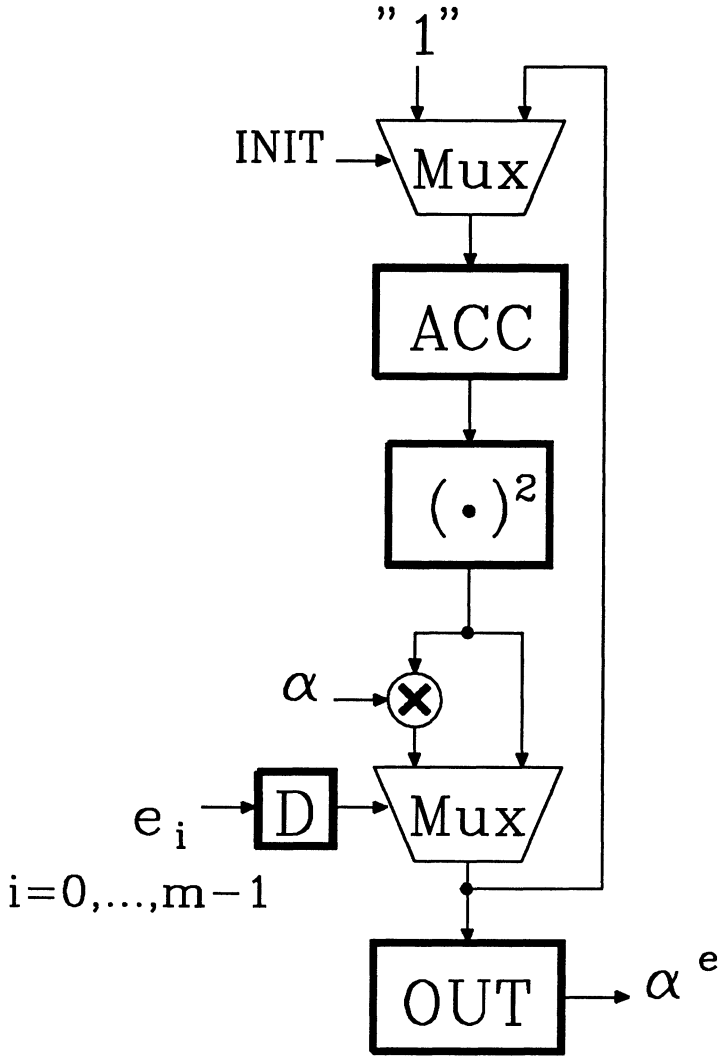


FIGURE 8.6 General structure of an exponentiator with variable base α and consecutive square and multiply.

operation is then reduced to one exponentiation per $m(m + w_e)$ clock cycles where w_e is the Hamming weight of e.

Normal Basis. In NB the squarer "vanishes". With the bit-serial multiplier, complexity ~ $5m$ and a CP of length $\geq 3 + \lceil \log_2 m \rceil$ (see Tab. 3.11), the rate is one exponentiation per m^2 (synchronous) respectively $m(w_e - 1) + m$ (asynchronous) clock cycles.

With the bit-parallel multiplier, complexity ~ $3m^2$ (optimal NB) and a CP of length $2 + \lceil \log_2 m \rceil$ (eq. (4.20)), the rate becomes one exponentiation per m clock cycles.

8.2.2 Concurrent S & M

Squaring and multiplication can be performed in concurrency if we operate by eq. (8.2). Fig. 8.7 shows the general structure of the exponentiator. The algorithm implemented is the following. Let

$P^{(i)}$ = product term
$S^{(i)}$ = square term.

Initialize

$$P^{(-1)} = 1 \qquad S^{(-1)} = \alpha.$$

Then, for $i = 0, 1, \ldots, m-1$, compute

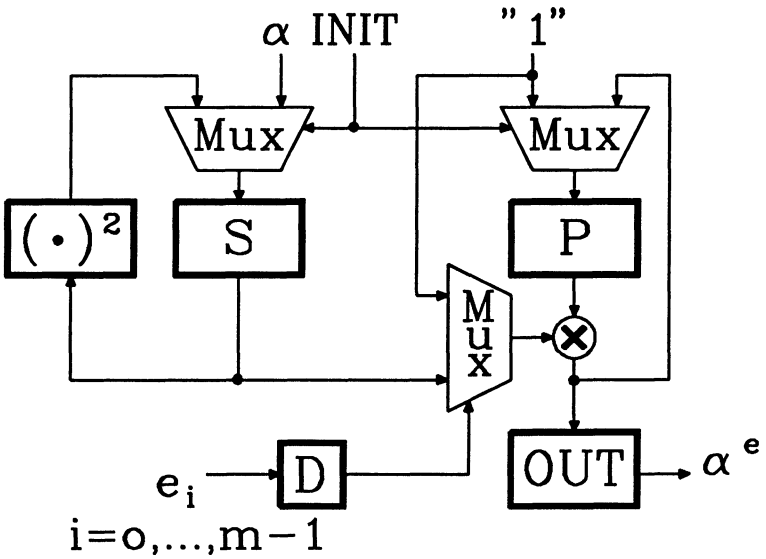
$$P^{(i)} = \begin{cases} 1 \cdot P^{(i-1)} & \text{if } e_i = 0 \\ S^{(i-1)} \cdot P^{(i-1)} & \text{if } e_i = 1 \end{cases} \qquad S^{(i)} = [S^{(i-1)}]^2.$$

The final result is

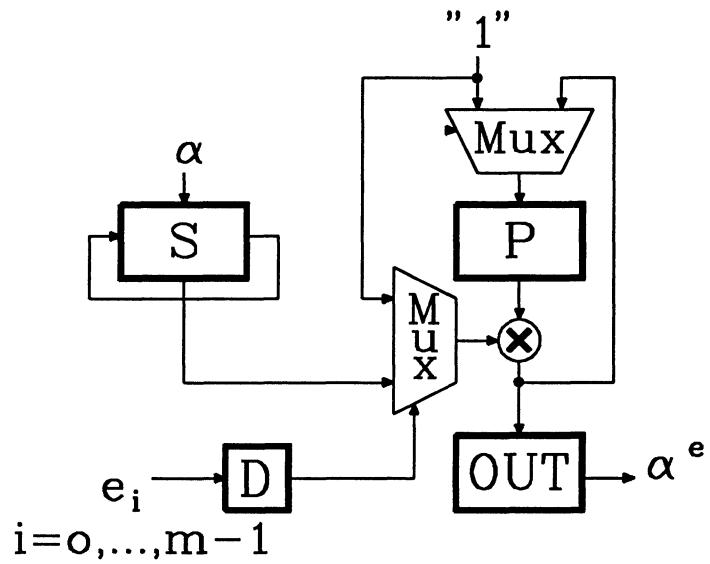
$$\alpha^e = P^{(m-1)}.$$

The squaring unit simply produces the conjugates of α iteratively starting from α and feeds them to the multiplication unit where they are multiplied and accumulated according to the exponent's bits. The exponent is read from LSB to MSB.

For large m the multiplier has to be of bit-serial type. Then it is not really meaningful to have a fast bit-parallel squarer since it would not contribute to



(a)



(b)

FIGURE 8.7 General structure of exponentiator with variable base α in PB (a) and (b).

increase the exponentiation rate which is set by the multiplier to one exponentiation per m^2 clock cycles.

Polynomial Basis. In PB the squarer can be realized as a bit-serial multiplier which simplifies the design work since it is already available in the multiplication unit. Only minor modifications are required to make the multiplier work as a squarer, see Fig. 8.8 for an example over GF(16). We see that the architecture is highly regular and thus well suited for large-field exponentiation. The complexity of the entire device (with pipelining registers etc.) for arbitrary m is $\sim 14m$ (see the previous section for the complexities of other combinations of representation (PB/NB) and data structure (serial/parallel)).

The fact that the squaring is carried out concurrently with the multiplication implies that the performance of the PB exponentiator is limited exclusively by the slowest of the two operations. In this case squaring and multiplication are as fast (or slow) — the CP has length 3.

Normal Basis. In NB the squarer is a simple register for cyclically shifting the base α. Fig. 8.9 shows an NB exponentiator over GF(16). The module realizing the logic function f is not easily implemented for large m. As indicated in Tab. 8.2 an optimal NB is a necessity to make the NB a reasonable alternative. With an optimal NB the f-module has complexity $\sim 3m$ and the whole exponentiator $\sim 10m$ — the CP has length $3 + \lceil \log_2 m \rceil$.

8.2.3 Stored Conjugates

If α does not change too frequently the architectures of Sec. 8.1.1 can be easily modified to allow for variable exponentiation base. The ROM storing the conjugates is simply replaced by a writable memory into which the new conjugates can be stored. If circulating registers are used, these are easily reconfigured to allow for loading of new conjugates [Sco88]. The rate will depend on the actual realization of the multiplier tree (see Sec. 8.1.1).

The computation of the new conjugates has complexity $O(m^2)$ in PB and $O(m)$ in NB. In PB this computation has to be accomplished by some additional (external) device or by the exponentiator itself as an initialization procedure prior to the next sequence of exponentiations. In NB a cyclically shifting register is enough.

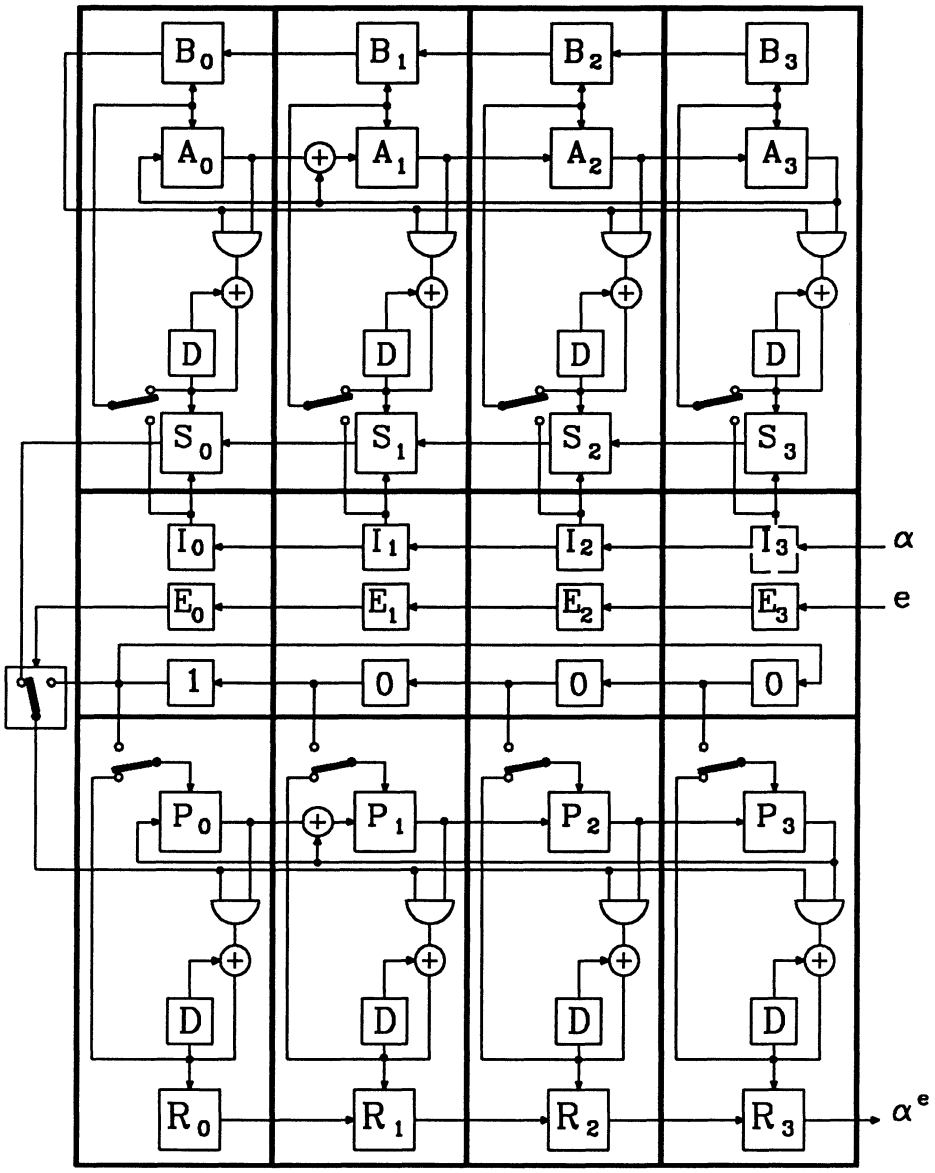


FIGURE 8.8 Pipelined exponentiator in PB over GF(16) with concurrent square and multiply. The upper multiplier acts as a squarer.

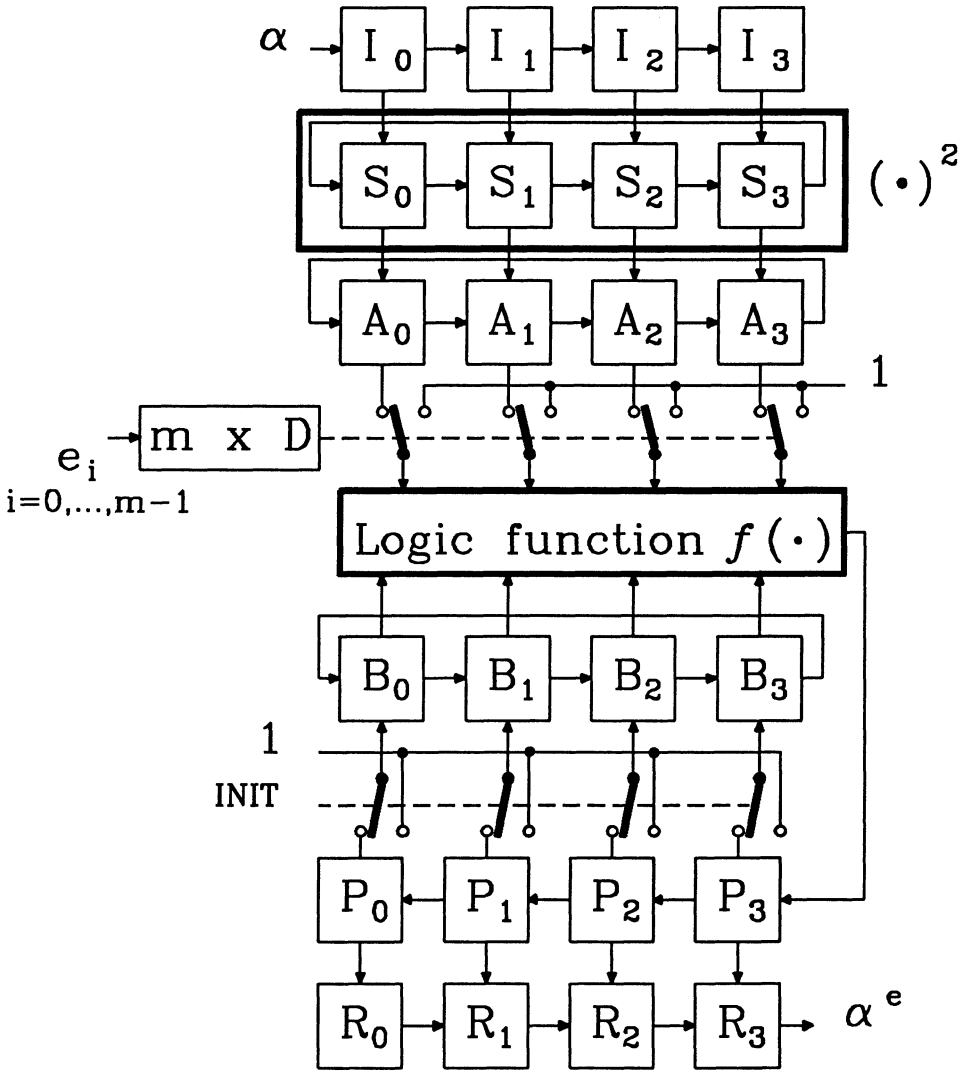


FIGURE 8.9 Pipelined exponentiator in NB over GF(16) with concurrent square and multiply and bit-serial multiplier.

8.3 Exponentiation with Fixed or Slowly Varying Exponent

If the exponent is fixed, or varies relatively slowly, and the base is variable, it is possible, in certain configurations, to systematically minimize the number of multiplications involved in the exponentiation process.

The exponent is input to a programmable control unit that, prior to the next batch of exponentiations, sets up a sequence of control signals for the squaring and multiplication units that corresponds to a particular, fixed sequence of operations. The idea is to let a fast squaring unit produce each of the conjugates α^{2^i} for which $e_i = 1$ in concurrency with a non-trivial bit-serial multiplication. Fig. 8.10 shows the general structure of such a device.

Polynomial Basis. A suitable configuration is that of Fig. 8.7a with the MSR multiplier and a bit-parallel squarer (Ch.7). Let

$$e = \sum_{n=1}^{w_e} e_{i_n} 2^{i_n}.$$

Then the sequence of operations is the following.

I. The element α^k with $k = e_{i_1}$ is computed in $i_1 \leq m$ squarings.
II. α^k is passed to the multiplier for multiplication and accumulation. Simultaneously the next element α^k with $k = e_{i_2}$ is computed in $i_2 - i_1 \leq m$ squarings.
III. α^k is passed to the multiplier for multiplication and accumulation. Simultaneously the next element α^k with $k = e_{i_3}$ is computed in $i_3 - i_2 \leq m$ squarings.
IV. And so on...

The number of multiplications (all non-trivial) is w_e and the rate one exponentiation per $i_1 + mw_e$ clock cycles where the length of the clock cycle is determined by the CP of the MSR multiplier.

Normal Basis. Here the squarer becomes a cyclically shifting register (see Fig. 8.7b). The rate is one exponentiation per $i_1 + mw_e$ clock cycles where the length of the clock cycle is determined by the CP of the serial NB multiplier.

If the exponent is chosen at random then $w_e \approx m/2$, at average, and the rate is one exponentiation per $i_1 + m^2/2$ clock cycles.

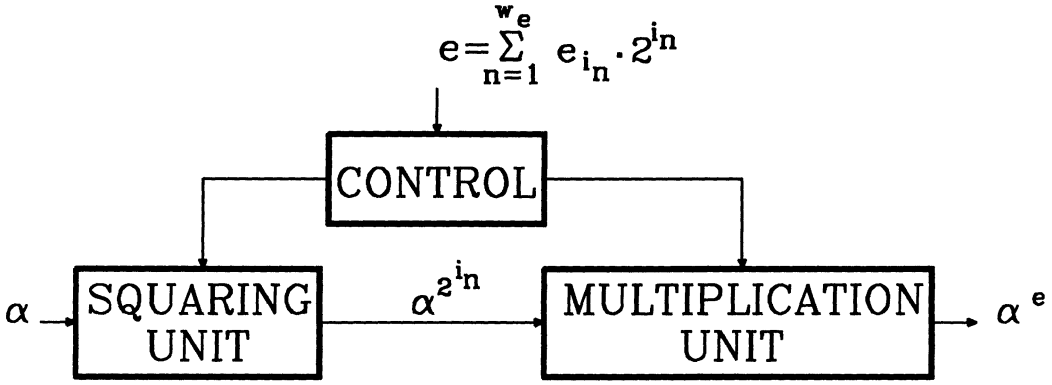


FIGURE 8.10 System structure of exponentiator with fixed exponent e.

8.4 Discussion

In this chapter we have discussed a great variety of exponentiation architectures, each architecture with different properties (data structure and representation, complexity, performance etc.) and fitting different applications (fixed base, variable base, fixed exponent).

As long as the field order m is small or moderately large, the designer is relatively free to choose among the above variety of architectures.

However, as m grows large like in certain applications of cryptography where $m \geq 1000$ is envisaged, finding and implementing a suitable exponentiation architecture becomes much more involved. In these situations it is not sufficient to look at the parameters mentioned in the previous sections, rather there are other issues that become as important as, or even more important than the amount of components required or the number of gates along the critical path.

For a successful large-field design, the following additional issues must be given great attention:

- structure
- routing and driving of long and/or heavily loaded lines
- wide data paths
- distribution of clocks and other control signals
- power dissipation
- floor-planning and placement.

In the sequel we adopt the following abbreviations: f.b. = fixed base, v.b. = variable base, f.e. = fixed exponent, exp = exponentiation, c.c. = clock cycles.

Structure

This issue has actually been taken into account to a certain extent. The architectures of Fig. 8.5 (f. b., 1 exp/m^2 c.c.) and Fig. 8.8 (v.b., 1 exp/m^2 c.c.) were suggested mainly because of their nice bit-slice structure.

None of the other architectures presented above offer such a high degree of regularity.

A high degree of regularity simplifies design and leads to high density in the physical layout.

Long and/or heavily loaded lines

The presence of long and/or heavily loaded lines affects directly the maximum clock frequency and consequently the system performance. For example, the serial input of the MSR multiplier is a long line that has to drive m AND gates. For large m, a number of refresh amplifiers is clearly needed to manage such a heavy load (see also Sec. 1.3). Furthermore, the layout must be chosen so to minimize (or at least reduce) the time skew between the nearest and the most distant unit to be driven (see also Floor-planning).

A similar problem arises also in the design of the fast PB squarer, the NB α-cell and the block realizing the f-function (Fig. 8.9) where certain input signals might have to drive ~ $m/2$ XOR gates.

Wide data paths

Wide data paths and particularly *long* such paths occupy large areas and act as large capacitive loads. Moreover, their routing and connection is a time-consuming task. The best way to deal with them is therefore to avoid them. For example, the two $\frac{m}{2}$-bits wide buses running on the sides of Fig. 8.3 become very inconvenient for large m. They should therefore be routed *through* the α-cell and the squarer.

Distribution of clocks and other control signals

A careful design of the clock distribution network is necessary to avoid or reduce clock skew. This is of particular importance when the number of clocked units is large. For example, all architectures based on bit-serial multipliers (e.g. Fig. 8.8 and 8.9) contain a large fraction of registers.

Since clocks are normally distributed in wide metal wires, the area occupied by the clock distribution network may be substantial.

Similar considerations are valid for the distribution of control signals.

Power dissipation

Since large-field exponentiators have to adopt bit-serial arithmetic, we saw that the exponentiation time can not be better than $O(m^2)$. If short exponentiation times are desired we must therefore increase the system's clock frequency.

However, the power dissipated in a CMOS circuit increases linearly with the total capacitance and the switching frequency (see Sec. 1.3). Even if these are difficult to estimate we can easily deduce that the total capacitance can not be small if m is large (i.e. the number of clocked units is large). Similarly, we can easily deduce that the switching frequency of most capacitances must increase with the system's clock frequency.

A rule-of-thumb says that the total power dissipation of a single chip should not exceed 1 W.

Floor-planning and placement

Integrated circuits can not be too tall or too wide. The modules/cells of a large-field exponentiator are therefore likely to be organized and placed in a somewhat unnatural way in order to obtain the desired aspect ratio (normally corresponding to a ~ square). This fact might complicate the layout work and eventually limit our choice of architecture.

For large m, the bit-slices of Fig. 8.5 or 8.8 can hardly be laid out as a single row, rather they must be organized in a two dimensional array connected as a serpentine. Clocks, control, feedbacks (LFSR) and the serial inputs to the multipliers would then be distributed through a regular, two-dimensional network across the array rather than through long wires along the serpentine. This will reduce time skews and set-up times and improve the performance of the system.

The same arrangement is less suited to the architectures of e.g. Fig. 8.4a, 8.4b or 8.9 where the presence of random-logic blocks (PB squarer, NB α-cell, f-module) complicates the folding of the layout and the distribution/routing of the required signals.

Our conclusions concerning exponentiation in GF(2^m) are the following:

- For small to moderately large m the designer can choose among a large variety of architectures while for large m the choice is very limited.
- Variable-base exponentiation in $O(m)$ time is feasible only for small to moderately large m.
- Fixed-base exponentiation in $O(m)$ time is feasible for large m. PB architectures require often less components than NB architectures. When

the field generator is a trinomial PB architectures require always less components than NB architectures. For large m, NB architectures require optimal or (true) low-complexity NB. None of the architectures display a high degree of structural regularity.

- Variable-base exponentiation for large m is feasible only in $O(m^2)$ time. Both PB and NB architectures are feasible. The PB architecture is superior mainly due to a high structural regularity, independently of field generator, which simplifies the implementation significantly. A necessary condition for implementing an NB architecture is the availability of an optimal NB.

Chapter 9

Division

Up to now, division in Galois fields has been used almost solely in applications of coding theory — that is, in small fields. However, it cannot be excluded that future cryptographic or other applications will require division in large fields.

The division of a field element γ by another field element β is normally viewed as a multiplication of γ by β^{-1}, i.e.

$$\frac{\gamma}{\beta} = \gamma\beta^{-1}.$$

Since each element of a Galois field has a unique multiplicative inverse we need *only* find this inverse and feed it to a multiplier along with the other element. However, we will see that, in general, finding the inverse is not a simple task.

The principal object of this chapter is to cover the above subject by presenting most known methods/architectures for finding the inverse of an element of GF(2^m). Along the way, a few, seemingly unpublished architectures are presented.

9.1 Inversion by Exponentiation

Let $\beta \in$ GF(2^m) be the element whose inverse we wish to determine. Since

$$\beta = \beta^{2^m}$$

we can divide both sides by β^2 and obtain

$$\beta^{-1} = \beta^{2^m-2}. \tag{9.1}$$

Finding the inverse β^{-1} is thus equivalent to an exponentiation with variable base β and fixed exponent $e = 2^m - 2$.

The binary representation of the integer $2^m - 1$ is clearly 111...11 — that is, $2^m - 1 = 2^0 + 2^1 + 2^2 + \cdots + 2^{m-1}$ or,

$$2^m - 2 = 2 + 2^2 + \cdots + 2^{m-1}. \tag{9.2}$$

Eq. (9.1) and (9.2) yield

$$\beta^{-1} = \beta^2 \beta^{2^2} \beta^{2^4} \cdots \beta^{2^{m-1}} \tag{9.3}$$

which can also be written as

$$\beta^{-1} = (\cdots((\beta)^2)\beta)^2 \cdots \beta)^2 \tag{9.4}$$

where β appears exactly $m - 1$ times. Hence, the inverse of β can be viewed as the product of its conjugates and can be computed by square and multiply (S & M).

Notice the similarity between eq. (9.3)-(9.4) and eq. (8.2)-(8.3).

9.1.1 Iterative S & M, $O(m)$ Time

Fig. 9.1 and 9.2 show the system structure of the general inverter based on eq. (9.3) and (9.4) respectively. We see that, in both systems, the main *ingredients* are the multiplier and the squarer.

If the inverse is to be obtained in ~ m clock cycles, both the multiplier and the squarer must be of parallel type which implies that m must be small or moderately large.

Polynomial Basis. The parallel multiplier (Ch. 4) has complexity ~ $2m^2$ while the squarer (Ch. 7) has complexity ~ $2m$ — altogether $\mathcal{C} \sim 2(m^2 + m)$. The performance is determined by the sum of the CP lengths of the multiplier and the squarer, i.e. $\mathcal{L} = \lceil \log_2 m \rceil + d + \mathcal{L}_{\square}$ where we normally have $3 \le d + \mathcal{L}_{\square} \le 5$ (see Tab. 4.9 and Ch. 7).

Normal Basis. Here the squarer "vanishes". The squaring is done either by a cyclically shifting register [Wan85] — B in Fig. 9.1 — or by an hard-wired cyclic shift (Fig. 9.2). The parallel multiplier (Sec. 4.8) has complexity ~ $3m^2$ for optimal NB and a CP of length $2 + \lceil \log_2 m \rceil$ (also for optimal NB).

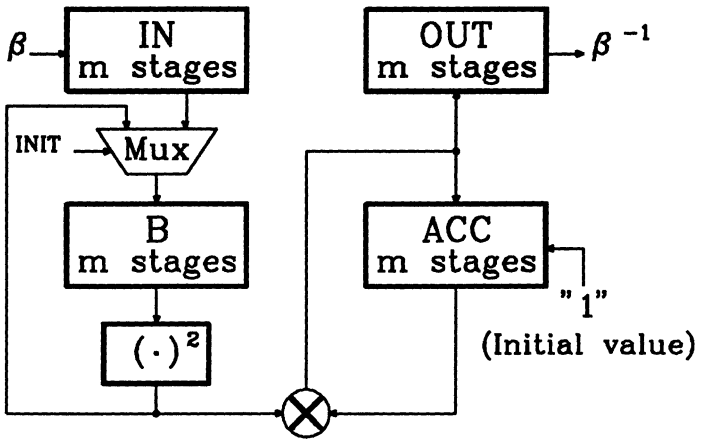


FIGURE 9.1 System structure of a pipelined inverter over GF(2^m) based on eq.(9.3).

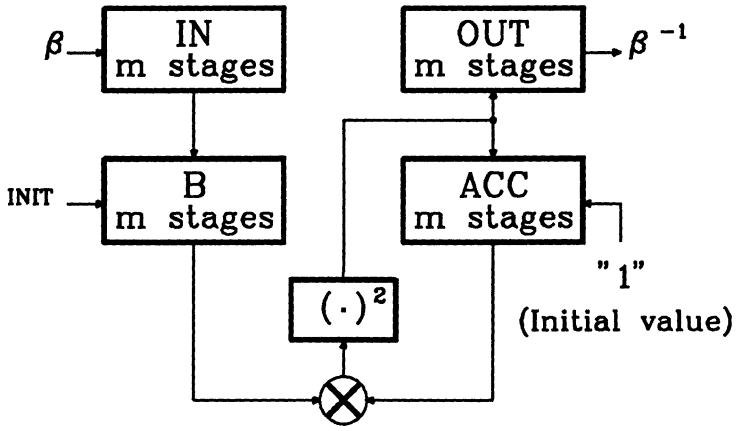


FIGURE 9.2 System structure of a pipelined inverter over GF(2^m) based on eq.(9.4).

9.1.2 Inversion with Multiplier Tree, $O(m)$ Time

We can also compute β^{-1} adopting structures similar to the stored-conjugates exponentiator of Fig. 8.1. The conjugates of β can be generated by a cascade of $m-1$ squarers (or square rooters) as shown in Fig. 9.3a ($m = 5$) or by two different cascades, one consisting of squarers and the other consisting of square rooters, for example as shown in Fig. 9.3b. The conjugates are then multiplied together by a tree of $m-2$ bit-serial multipliers. With pipelining and continuous operation, a rate of one inversion per m clock cycles is obtained.

Polynomial Basis. The cascade generating the conjugates of β in Fig. 9.3a uses only one type of module — the squarer or the square rooter depending on which one is smallest/fastest (see Ch. 7) — but is slower, due to a longer CP, than the mixed cascade of Fig. 9.3b. The actual combination of squarers/square rooters in the latter cascade is chosen so to reduce complexity and CP length (e.g. with the help of Tab. 7.5). Typically, such a mixed cascade will have complexity at most $\sim 2(m^2 - m)$ and a CP of length $\leq m-1$.

A tree of e.g. MSR multipliers has complexity $4(m^2 - 2m)$ whereby the total complexity becomes $\sim 6m^2$.

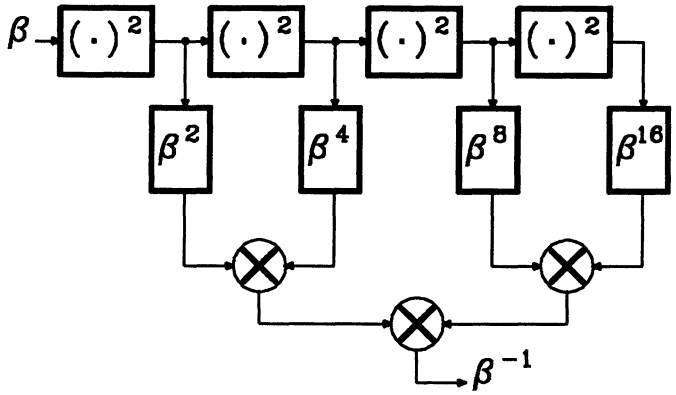
The performance of the inverter is determined by the sum of the CP lengths of the above cascade and of the MSR multiplier ($\mathcal{L}_{\text{MSR}} = 3$), i.e. typically a CP of length $\sim m + 2$.

Normal Basis. The squarers/square rooters become hardwired cyclic shifts that cause no additional delay. The bit-serial multiplier has complexity $\sim 5m$ (optimal NB). The multiplier tree has then complexity $\sim 5(m^2 - 2m)$. The performance is determined by the CP of the multiplier which is $3 + \lceil \log_2 m \rceil$ (optimal NB).

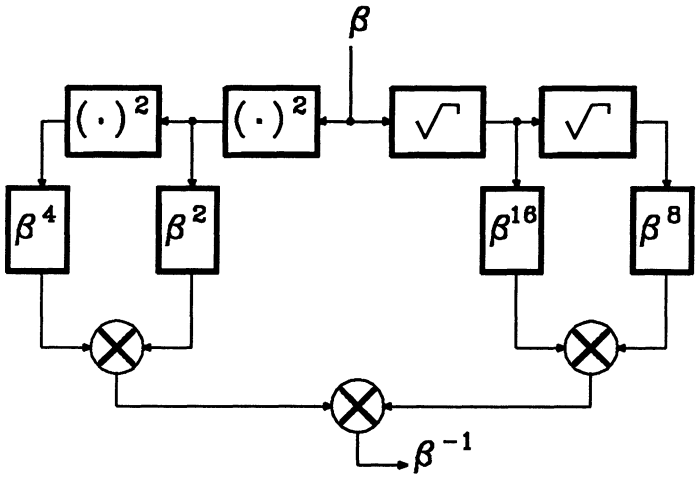
9.1.3 Iterative S & M, $O(m \log m)$ Time

An algorithm for fast inversion by exponentiation was introduced in [Fen89]. Although valid for any representation of GF(2^m), the algorithm becomes highly efficient first in conjunction with an NB representation.

The key idea is a novel decomposition of the exponent $2^m - 2$. Let



(a)



(b)

FIGURE 9.3 System structure of an inverter over GF(2^5) vith squaring cascade (a) and mixed cascade (b). In (a) the squarers can be replaced by square rooters.

$$m-1 = m_0 + m_1 2 + m_2 2^2 + \cdots + m_{q-1} 2^{q-1} + m_q 2^q \qquad (9.5)$$

where $m_q = 1$. Then, we have the following decomposition (see [Fen89, Appendix] for a proof)

$$2^1 + 2^2 + \cdots + 2^{m-1} =$$

$$2^{m-m_0}[((\cdots((m_q 2^{-m_q 2^q}(1 + 2^{2^{q-1}}) + m_{q-1})2^{-m_{q-1}2^{q-1}}(1 + 2^{2^{q-2}}) + m_{q-2})2^{-m_{q-2}2^{q-2}}(1 +$$

$$2^{2^{q-3}}) + \cdots)2^{-m_2 2^2}(1 + 2^{2^1}) + m_1)2^{-m_1 2^1}(1 + 2^{2^0}) + m_0]. \qquad (9.6)$$

Combining eq. (9.6) and (9.3) leads to the following inversion algorithm:

Step I: $\delta := \beta$
Step II: FOR $i = q$ TO 1 DO
Step III: BEGIN IF $m_i = 1$ THEN $\delta := (\delta)^{2^{-2^i}}$
Step IV: $\delta := \delta \times \delta^{2^{2^{i-1}}}$
Step V: IF $m_{i-1} = 1$ THEN $\delta := \delta \times \beta$
END
Step VI: $\beta^{-1} := (\delta)^{2^{m-m_0}}$. STOP.

We see that multiplications are required only in steps IV and V while steps III and VI require only cyclic shifts (in NB). The computation of β^{-1} requires thus $(q + p)$ multiplications, where p is the number of ones in $(m_0, m_1, \ldots, m_{q-1})$. Fig. 9.4a shows the general structure of the inverter. The number of signals controlling the multiplexer is $\lceil \log_2(q + p) \rceil$ [Fen89, Sec. III].

The above inverter makes use of a special serial-in/parallel-out NB multiplier introduced in [Fen89] whose general structure is shown in Fig. 9.4b. The multiplier accepts both inputs A and B in serial form and produces the result $A \times B$ in parallel form after m clock cycles. Of the five sections composing the multiplier, the NB α-cell is the only one having an irregular structure. Examination of [Fen89, Fig. 1] shows that the multiplier has complexity $\sim 7m + \mathcal{C}_{\alpha\text{-cell}} \geq 8m - 1$ while its CP has length $5 + \mathcal{L}_{\alpha\text{-cell}} \geq 6$ with equality in both cases only for optimal NB (see eq. (8.9) and (8.10)).

Clearly, the computation of β^{-1} can be performed in $m(q + p) \sim m \log m$ clock cycles. The algorithm is in its simplest form when $m = 2^k + 1$ — i.e.

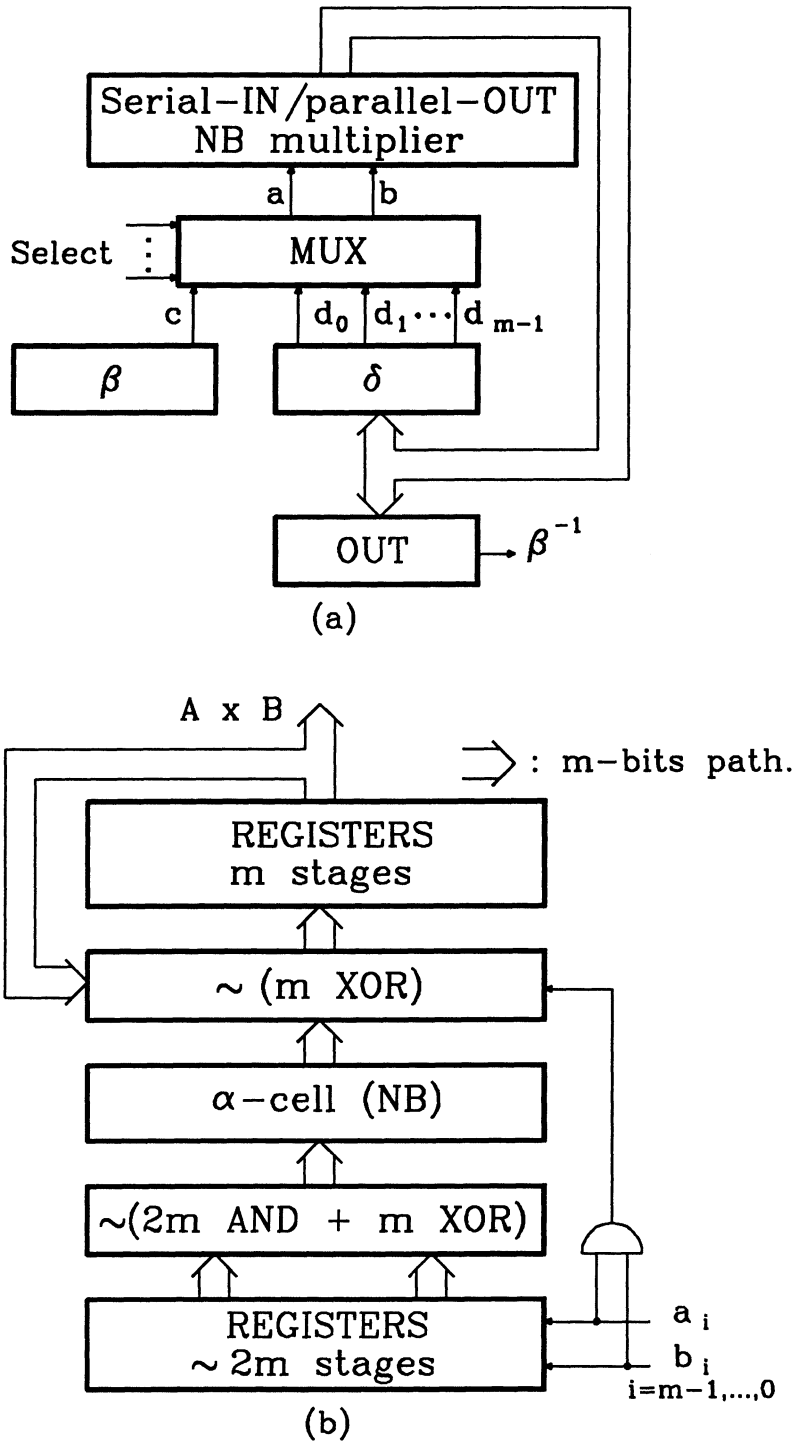


FIGURE 9.4 System structure of an NB inverter over GF(2^m) for inversion in O(m log m) time [Fen 89], (a). The system structure of the serial/parallel NB multiplier is shown in (b).

when $m-1=2^k$ whereby $p = 0$ ($\Leftrightarrow$ step V vanishes) and $q = \lceil \log_2(m-1) \rceil = k$ — and in its most complex form when $m = 2^k$ — i.e. when $m - 1 = 2^k - 1$ whereby $p = q = \lceil \log_2(m-1) \rceil = k - 1$ ($\Leftrightarrow$ step V is performed every time). The inverter has complexity ~ $10m$.

9.1.4 Iterative S & M, $O(m^2)$ Time

The similarity of eq. (9.3) and (8.2) suggests that the exponentiator of Fig. 8.9 should be easily turned into an inverter over GF(2^m). Indeed, in Fig. 8.8 we need only remove the registers for the exponent and the element 1, and the multiplexer between the output of the upper (squaring) multiplier and the input of the lower multiplier since all $m-1$ conjugates of β are required. Fig. 9.5 shows the architecture of the resulting inverter for $m = 4$. The operation is as follows.

1) The A and B registers are loaded with the input element β.

2) During the first m clock cycles the upper multiplier computes β^2. β^2 is then loaded in the S register and simultaneously loaded into the A and B registers. The P register is set to the element 1.

3) During the second m clock cycles the lower multiplier computes $1 \cdot \beta^2$ while the upper multiplier computes β^4. β^2 is then loaded into the P register while β^4 is loaded into the S register and simultaneously loaded into the A and B registers.

4) During the third m clock cycles the lower multiplier computes $\beta^2 \cdot \beta^4 = \beta^6$ while the upper multiplier computes β^8. β^6 is then loaded into the P register while β^8 is loaded into the S register.

5) During the last m clock cycles the lower multiplier computes $\beta^6 \cdot \beta^8 = \beta^{14} = \beta^{-1}$. β^{-1} is then loaded into the R register. (GOTO 1)

The architecture is highly regular and has complexity ~ $12m$. The performance is determined by the CP length (= 3) of the MSR multiplier. The inversion time is $O(m^2)$.

Also the NB exponentiator of Fig. 8.9 is easily transformed into an inverter. This is done by removing the upper array of multiplexers and the registers buffering the exponent. The complexity is $\geq 9m$. The performance is determined by the CP length ($\geq 3 + \lceil \log_2 m \rceil$) of the NB multiplier. The inversion time is again $O(m^2)$.

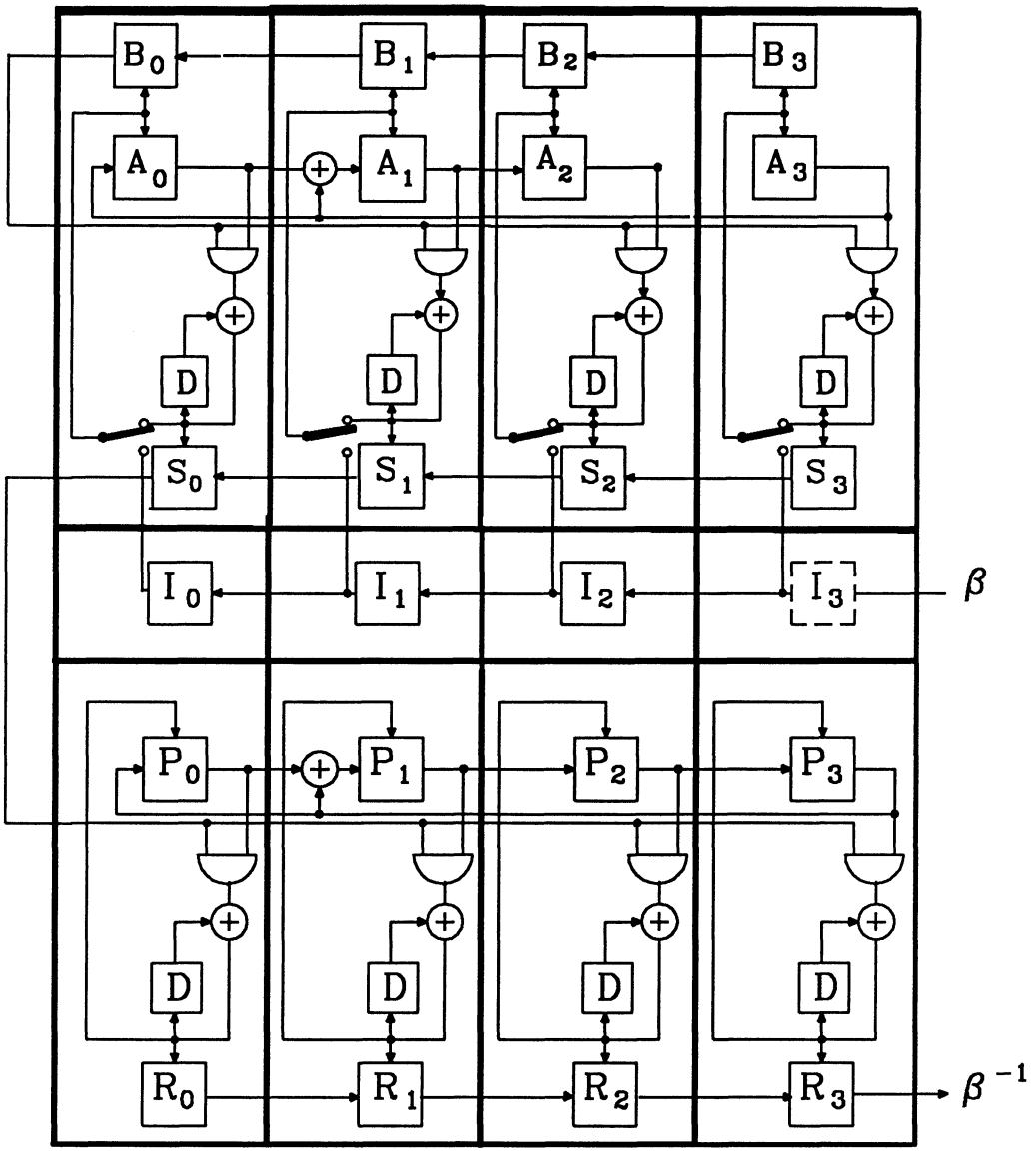


FIGURE 9.5 Pipelined inverter in PB over GF(16) with concurrent square and multiply. The upper multiplier acts as a squarer. The inversion time is $O(m^2)$.

9.2 Direct Inversion

In Ch. 3 we noted that multiplication by an arbitrary field element A can be described through the product matrix Z whose entries are linear combinations of A's coefficients. In particular, the left-most column of Z consists of A itself. The matrix Z is invertible and its inverse Z^{-1} corresponds to multiplication by A^{-1}. In particular, the left-most column of Z^{-1} is A^{-1} itself. Therefore, if we determine the left-most column of Z^{-1} we have also found the m (non-linear) boolean functions by which we can compute, in one clock cycle, the inverse of A.

However, the computation of the left-most column of Z^{-1} is about as difficult as the inversion of Z which, in general, is a hard computational task due to the particular form of the entries of Z. Inverting this matrix by hand is recommended only for $m < 6$. Hence, although this approach potentially can give fast, parallel inversion circuits of polynomial complexity (we believe the complexity $\mathcal{C}$ to be $O(m^2)$), it is viable only for small m. Furthermore, we expect the resulting logic to be of random type and, as such, hardly suited for VLSI implementation.

Example 9.1 Let $P(x) = 1 + x + x^4$ generate GF(16) and let Z be the associated product matrix — Z can be found in Ex. 4.1. Then we obtain the following boolean functions for computing the components d_i of the inverse of A:

$$d_0 = a_2 + a_3 + (a_2)'(a_0 + a_1) + a_1a_2(a_0 + a_3)$$

$$d_1 = a_0(a_1 + a_2) + a_1(a_2 + a_3) + (a_0a_1)'a_3$$

$$d_2 = a_0a_2a_3 + a_0a_1 + (a_0)'(a_2 + a_3)$$

$$d_3 = a_1 + (a_0)'(a_2 + a_3) + a_1(a_3 + a_0a_2)$$

where $(\cdot)'$ denotes binary complementation. The above functions can be implemented in about 25 gates (if we utilize the various symmetries in the functions). The CP has length 4. □

Another method for determining the boolean functions defining the inverse is given in [Dav72]. There, one has to solve a system of $2m - 1$ equations in $2m - 1$ unknowns which is about as hard as finding the left-most column of Z^{-1}.

9.3 Table Look-up

For small m, the inverse of a field element can be looked up in table. A ROM containing all inverse elements is simply addressed by the element to be inverted. Inversion can thus be performed in one clock cycle. The complexity of the ROM is $\sim m2^m$.

9.4 Inversion with the Euclidean Algorithm

Finally, we would like to mention that the inverse can be computed by means of the Euclidean algorithm for polynomials. The standard form of the algorithm is given in Th. 2.12.

Let $A(x)$ be the polynomial representation of the field element A. Finding the inverse of A is equivalent to finding the polynomial $B(x)$ such that

$$A(x)B(x) \equiv 1 \mod P(x) \tag{9.7}$$

or, equivalently

$$A(x)B(x) + Q(x)P(x) = 1. \tag{9.8}$$

Since $P(x)$ is irreducible we have that the GCD of $P(x)$ and $B(x)$ is 1. The above equation can thus be written as

$$A(x)B(x) + Q(x)P(x) = \mathrm{GCD}[A(x), P(x)]. \tag{9.9}$$

The Euclidean algorithm is just a procedure for computing GCD's, which involves polynomial division as a subprocedure. The determination of both GCD$[A(x), P(x)]$ and the unknown polynomials $B(x)$ and $Q(x)$ is called the *extended GCD problem.*

A method that combines the steps of the polynomial-division algorithm with the steps of the Euclidean algorithm in a single procedure for computing $B(x)$ in eq. (9.9) is given in [Ber68, pp. 41]. It requires $\sim 2m$ registers and $\sim 4m$ additions/shifts. The procedure appears to require complicated control and its suitability for large fields has to be investigated.

9.5 Discussion

Since division is equivalent to multiplication by inverse we presented a number of architectures for finding the multiplicative inverse of an element of GF(2^m). Also, we recognized that inversion can be viewed as an exponentiation with fixed exponent ($2^m - 2$) and variable base. Consequently, some of the architectures suggested in this chapter are simple modifications of certain exponentiation architectures presented in Ch. 8.

For small to moderately large m, inversion in $O(m)$ time is feasible in both PB and NB, for example by iterative square & multiply with bit-parallel arithmetic. NB architectures are potentially faster than PB architectures due to the fact that squaring does not increase the CP length.

When bit-parallel arithmetic is not viable or too costly, an interesting alternative is the NB inverter introduced in [Fen89] whose complexity is linear in m ($\mathcal{C} \geq 10m$) and whose computation time is $O(m \log m)$. This architecture is also suitable for large m if an optimal or (true) low-complexity NB is available. The necessity of a good NB stems from the fact that an NB α-cell has to be designed. As noted in Ch. 8 the cost of a large-field NB α-cell might be prohibitive. Moreover, such a cell has typically a random-logic structure whereby a great design effort might be required.

If speed is not the main issue, a highly regular PB architecture based on e.g. the MSR multiplier can be implemented up to large values of m. The complexity is $\sim 12m$ and the inversion time is $O(m^2)$. The clock frequency is, in principle, independent of m.

The general conclusion we can draw is that the NB appears advantageous for purpose of multiplicative inversion over GF(2^m).

Chapter 10

Universal Architectures

The last topic of this thesis is the development of universal architectures. By *universal* we mean that it is possible to operate over a *range* of fields rather than over a single one. Since the word length of any computing device must be finite, the above range must also be finite.

When dealing with universal architectures it is mandatory to adopt a field representation that will not require basis-conversion hardware since this hardware cannot generally be made universal in reasonable complexity.

Of the three representations treated in this thesis (PB, NB, DB), the PB representation is the only one that fulfils the above requirement, and will therefore be assumed throughout this chapter.

10.1 Universal Galois Multipliers

The availability of good universal Galois multipliers (UGM) is crucial for the development of both universal Galois processors and other Galois operators. Moreover, multiplication is by far the most commonly used operation in applications of Galois fields.

10.1.1 Bit-Serial UGM

A bit-serial UGM can be obtained by modifying the MSR multiplier of Sec. 3.1.

Suppose we wish to design an UGM for operation in GF(2^m), $2 \leq m \leq M$. Then we can modify the MSR multiplier as shown in Fig. 10.1 for $M = 4$. There, we have introduced two new registers and a device consisting of a triplet of switches.

The p-register stores the coefficients $p_1, p_2, \ldots, p_{M-1}$ of the field generator $P(x)$. When $m < M$, the unused a- and p-coefficients are set to zero.

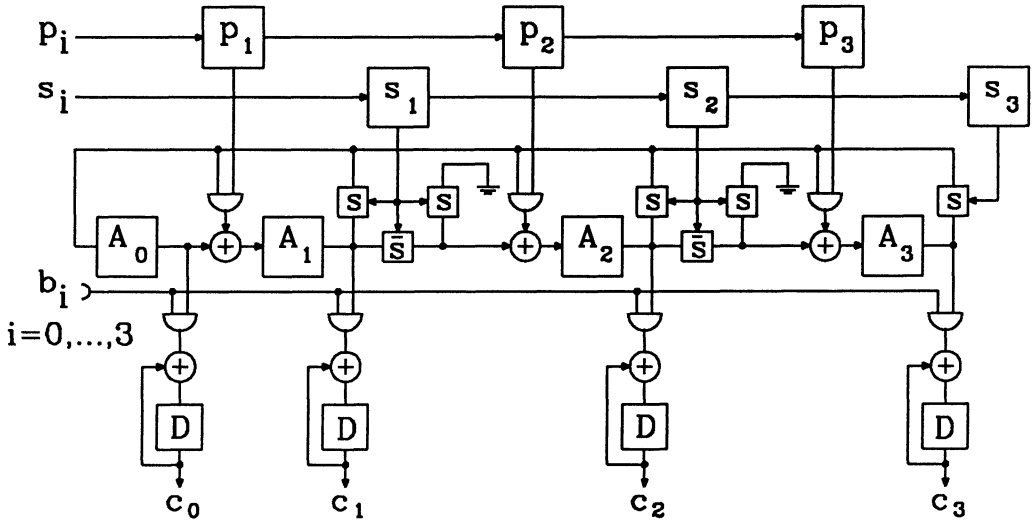


FIGURE 10.1 Architecture of universal MSR multiplier over GF(16).

The triplet consists of one $\bar{S}$ and two S switches. These are complementary switches in the sense that they operate in a complementary fashion when controlled by the same signal. Consider the triplet of the i:th bit-slice. Then the switches operate as follows:

$s_i = 1 \Rightarrow$ S open , $\bar{S}$ closed

$s_i = 0 \Rightarrow$ S closed, $\bar{S}$ open

where s_i is the i:th component of the control vector $S = (s_1, s_2, \ldots, s_{M-1})$ stored in the s-register. If the UGM is to be operated over GF(2^m), the control vector is set as follows:

$$\begin{cases} s_{m-1} = 1 \\ s_i = 0 \quad \text{otherwise.} \end{cases}$$

Of the three switches, the upper-left one is used to select the correct feedback signal. The upper-right switch is used to tie the input of the (unused) XOR gate to zero. This, together with the lower switch $\bar{S}$, will ensure that the unused part of the multiplier will produce a zero-output, i.e. $c_i = 0$ for $i \geq m$ (recall that all unused p-coefficients are zero whereby the unused AND gates produce zeros). If this feature is not needed, both the upper-right switch and the lower switch can be omitted.

When m grows large, one has to be careful with the driving capability of the feedback signal and of the b-input since both have to drive m AND gates. The switches connected to the feedback line will also contribute (with their capacitances) to load the feedback signal. Good driving capability is crucial for performance.

If not all values of m in the range [2, M] are to be used, the above architecture can be simplified by excluding all triplets corresponding to the those unused values. This will improve the performance by reducing the capacitance on the feedback line which is particularly important in applications where only a set of large values of m is interesting.

Finally we notice that the S vector could be derived from P by some simple logic whereby the s-register would vanish and the programming of the UGM simplify.

10.1.2 Bit-parallel UGM

If the fields to be used are not too large it is possible to adopt a bit-parallel UGM. We will develop an architecture based on the MSR approach.

In Sec. 3.1 we saw that the product $C = A \times B$ in GF(2^m) could be expressed in matrix form as

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,m-1} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,m-1} \\ \vdots & \vdots & & \vdots \\ z_{m-1,0} & z_{m-1,1} & \cdots & z_{m-1,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix} \tag{10.1}$$

where $\{z_{i,j}\}$ is the product matrix Z. We know also that the columns of Z can be generated by the α-array introduced in Sec. 4.2.

A parallel UGM can be obtained by connecting a universal α-array capable of operating over GF(2^m), $2 \le m \le M$, to a network of IP-cells designed to operate over GF(2^M). The general structure of the UGM is shown in Fig. 10.2. The UGM requires the input field elements to have zeros in the unused positions, i.e. $a_i = b_i = 0$, $i \ge m$.

The universal α-array consists of $m - 1$ universal α-cells. The operation of the ordinary α-cell is described by the following expression (from eq. (4.13))

$$a_{m-1} + \sum_{i=1}^{m-1} x^i (a_{m-1} p_i + a_{i-1}) \tag{10.2}$$

which is the polynomial corresponding to αA. Based on eq. (10.2) we can design a universal α-cell as shown in Fig. 10.3. The universal α-cell can be programmed to operate over any of the fields GF(2^m), $2 \le m \le M$, by means of the binary control vectors $P = (p_1, p_2, \ldots, p_{M-1})$ and $S = (s_1, s_2, \ldots, s_{M-1})$.

Suppose we want to program the UGM for operation over GF(2^m) where m is a particular value in the usable range. Then the components of the vector S are set as follows:

$$s_i = \begin{cases} 1 & i = m-1 \\ 0 & i \ne m-1. \end{cases} \tag{10.3}$$

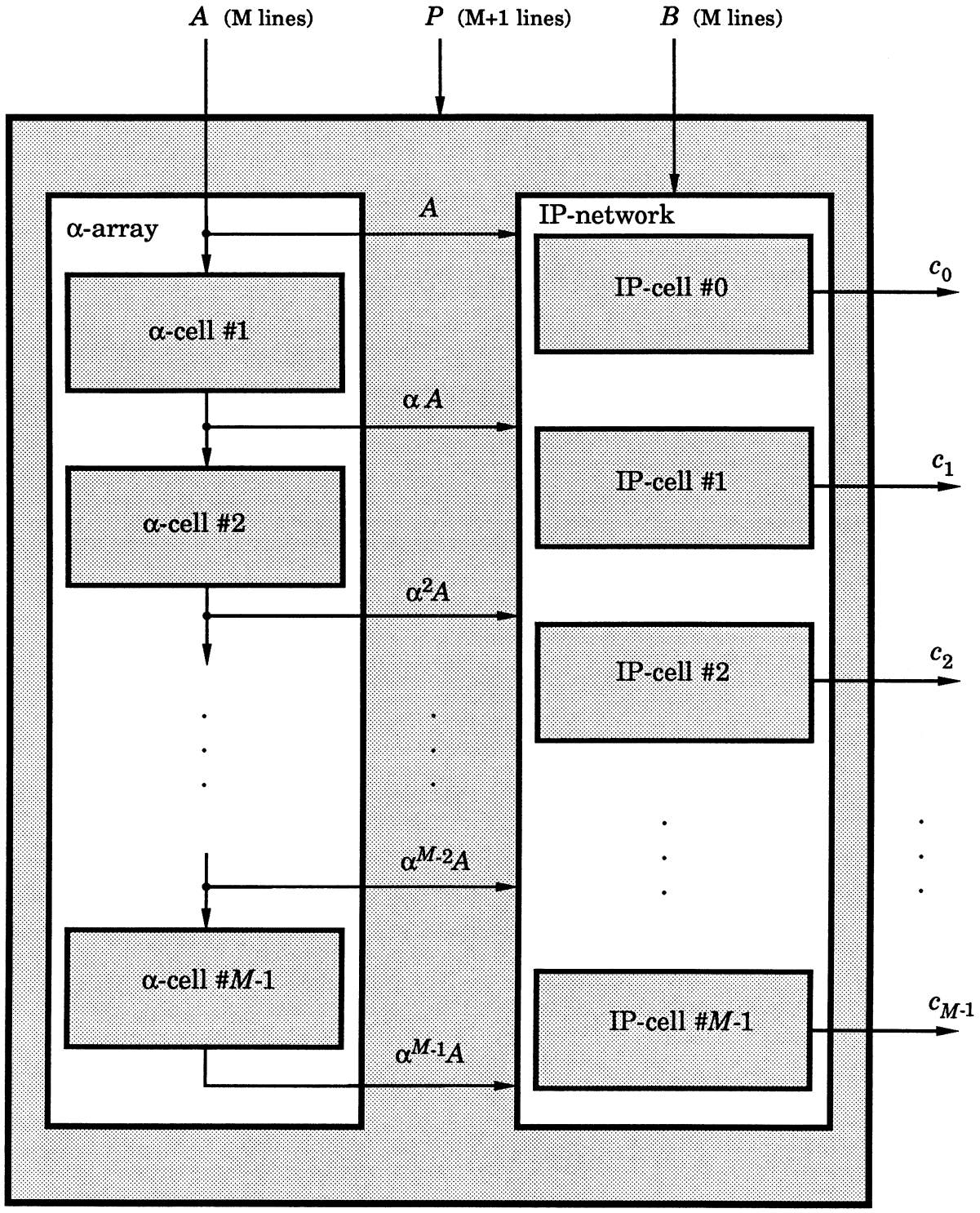


FIGURE 10.2 System structure of the bit-parallel UGM.

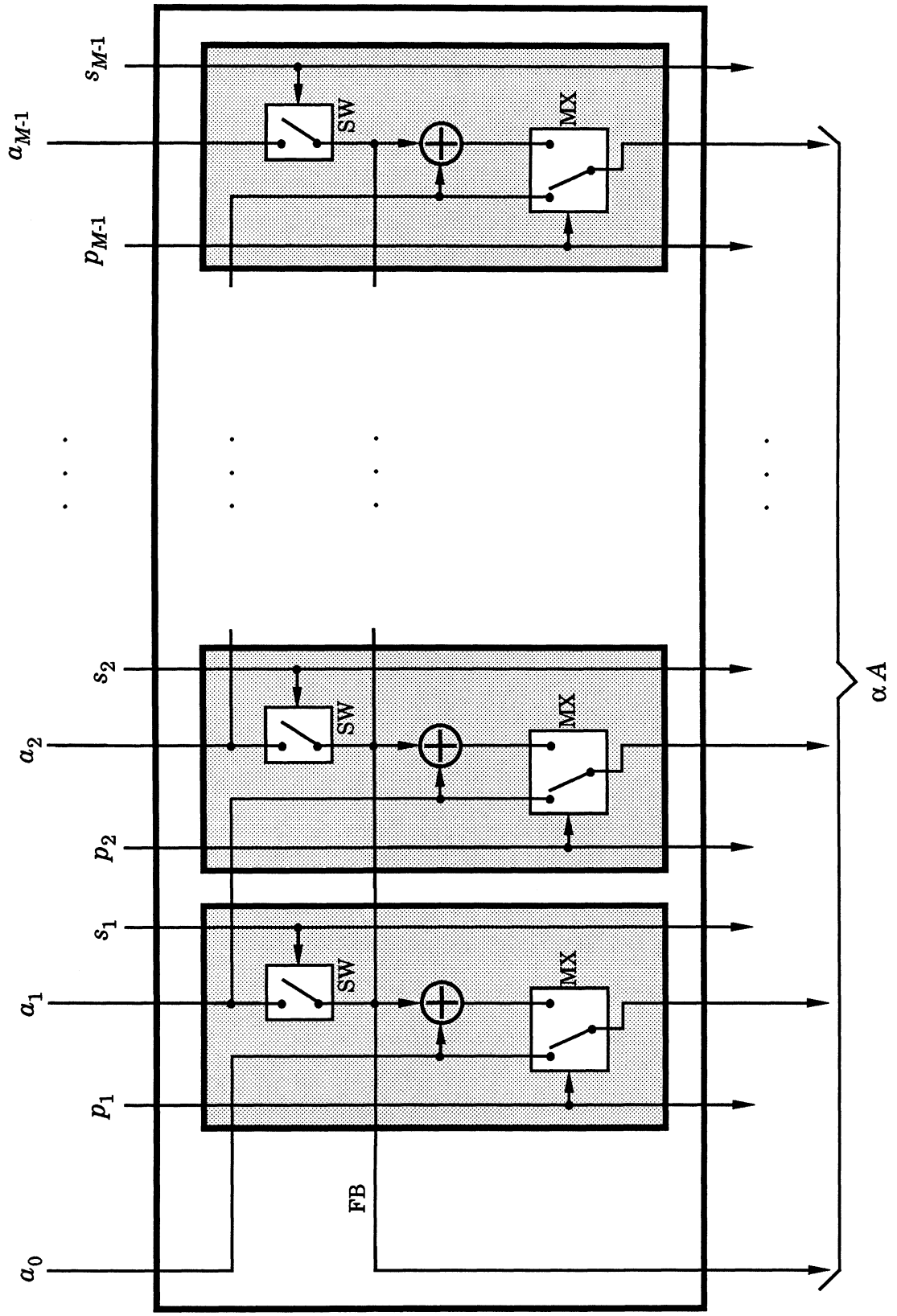


FIGURE 10.3 The universal α-cell.

The vector S determines the feedback signal FB shown in Fig. 10.3. The first $m-1$ components of the vector P are just the coefficients of the field generator $P(x)$. The unused coefficients p_m through p_{M-1} are set to zero.

We see that the α-cell has a regular bit-slice structure consisting of $m-1$ identical subcells. The switch SW in subcell i is controlled by the signal s_i in the following way:

$s_i = 1 \Rightarrow$ SW is closed
$s_i = 0 \Rightarrow$ SW is open.

The multiplexer MX is controlled by the signal p_i in the following way:

$p_i = 1 \Rightarrow$ MX transfers $a_{m-1} + a_{i-1}$
$p_i = 0 \Rightarrow$ MX transfers a_{i-1}.

Fig. 10.4 shows a particular realization of the IP-cell. The multiplexer MX appended to the output of each IP-cell ties the unused lines c_i, $i \geq m$, to zero. The signal v_i is the i:th component of a vector $V = (v_0, v_1, \ldots, v_{M-1})$ that is set as follows

$$v_i = \begin{cases} 0 & i \leq m-1 \\ 1 & i > m-1. \end{cases} \tag{10.4}$$

The multiplexer MX connects the output c_i to zero if $v_i = 1$. If $v_i = 0$ the output of the XOR-tree is connected to c_i.

Operating the UGM for $m < M$ means that only a part of α-array is used. This fact can be easily illustrated by eq. (10.1). First we define the vectors C_L, C_U, B_L and B_U as follows

$$C_L = (c_0, \ldots, c_{m-1})^T \qquad C_U = (c_m, \ldots, c_{M-1})^T$$

$$B_L = (b_0, \ldots, b_{m-1})^T \qquad B_U = (b_m, \ldots, b_{M-1})^T$$

where the superscript T indicates transposition. Then we have

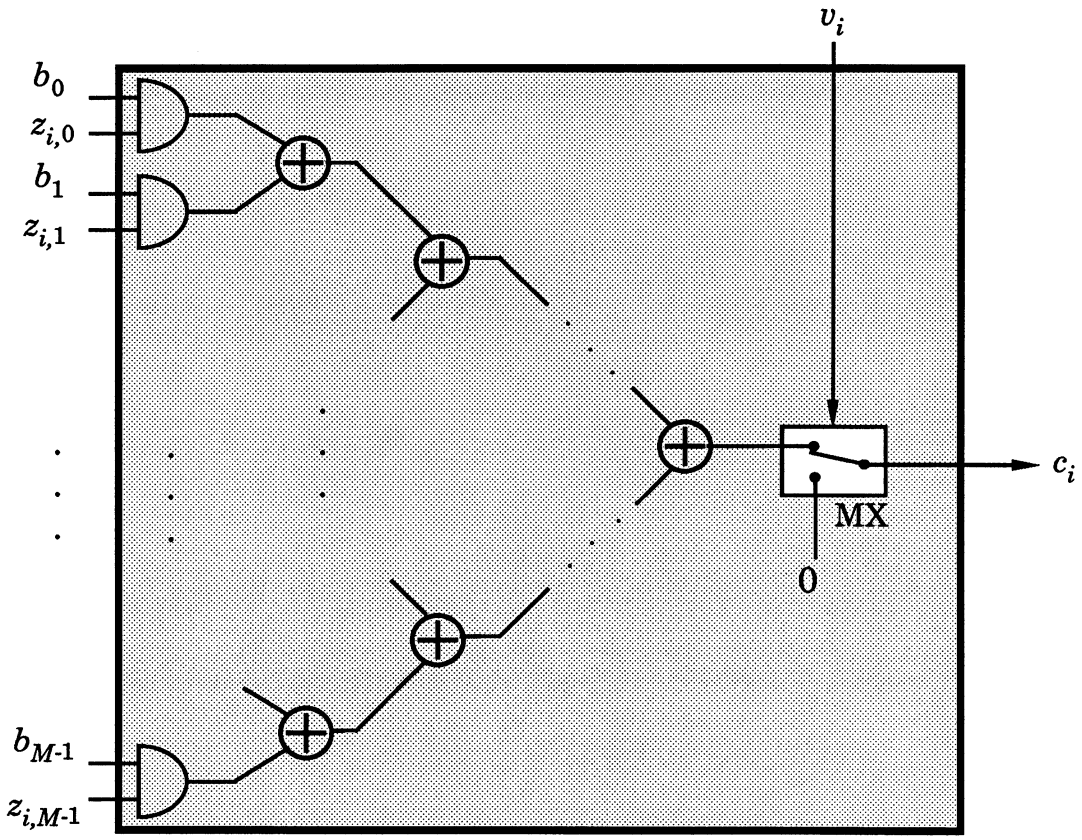


FIGURE 10.4 The i:th IP cell.

$$\begin{pmatrix} C_L \\ C_U \end{pmatrix} = \left(\begin{array}{ccc|ccc} z_{0,0} & \cdots & z_{0,m-1} & z_{0,m} & \cdots & z_{0,M-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ z_{m-1,0} & \cdots & z_{m-1,m-1} & z_{m-1,m} & \cdots & z_{m-1,M-1} \\ \hline z_{m,0} & \cdots & z_{m,m-1} & z_{m,m} & \cdots & z_{m-1,M} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ z_{M-1,0} & \cdots & z_{M-1,m-1} & z_{M-1,m} & \cdots & z_{M-1,M-1} \end{array}\right) \begin{pmatrix} b_0 \\ \vdots \\ b_{m-1} \\ b_m \\ \vdots \\ b_{M-1} \end{pmatrix} =$$

$$\left(\begin{array}{c|c} Z_1 & Z_3 \\ \hline Z_2 & \end{array}\right) \begin{pmatrix} B_L \\ B_U \end{pmatrix} \qquad (10.5)$$

where Z_1, Z_2 and Z_3 are submatrices of Z defined according to the subdivision of Z indicated in equation (10.5). The product of interest is $Z_1 \cdot B_L$ and we want it to appear on the lines of C_L. To have this product correctly computed, the product $Z_3 \cdot B_U$ must always be zero which is ensured by the fact that B_U is zero by requirement. What remains to take care of, is the product $Z_2 \cdot B_L$ since this is normally nonzero and it would appear on the lines of C_U (the unused lines that we wish to be zero). These lines are tied to zero by the multiplexer MX and the control signal v_i mentioned above and shown in Fig. 10.4.

Performance

The performance of the UGM is related to the CP. We will give an upper bound on the length L_{UGM} of the CP. To this end, we approximate the delay of a switch/multiplexer pair by that of an XOR gate.

The CP through the UGM must go through $m-1$ α-cells and one IP-cell.

The length of the CP through the IP-cell is fixed equal to $1 + \lceil \log_2 M \rceil$ gates.

The CP through the α-array depends on the choice of $P(x)$. It consists however of three parts: switches, XOR gates and multiplexers. Furthermore, since switches and multiplexers have poor driving capability we must include a line driver (e.g. a double inverter) in each subcell. We approximate the delay of a line driver by that of an XOR gate.

The number of XOR gates along the CP can be much less than $m-1$ by clever choice of $P(x)$. Tab. 10.1 shows the number of XOR gates along the CP

through the α-array for the best primitive field generators of degree ≤ 16 (the polynomials are taken from Tab. 5.2).

We see that the number of XOR gates is less than $\frac{m}{2}$ for $m \leq 8$. For $m > 8$ a better upper bound seems to be $\frac{m}{3}$. We use $\frac{m}{2}$ as an upper bound for all m.

The number of switches, drivers and multiplexers along the CP is not easily determined exactly. We assume worst case and say that the CP goes through m–1 switches, $m - 1$ drivers and $m - 1$ multiplexers. According to the above approximations these correspond to $2m - 2$ XOR gates.

m	$P(x)$	# of XOR gates
2	0,1,2	1
3	0,1,3	1
4	0,1,4	1
5	0,2,5	2
6	0,1,6	1
7	0,1,7	1
8	0,1,3,5,8	3
9	0,4,9	2
10	0,3,10	2
11	0,2,11	2
12	0,1,5,8,12	4
13	0,1,6,7,13	4
14	0,1,3,5,14	3
15	0,1,15	1
16	0,1,4,6,16	4

Table 10.1 The minimum number of XOR gates along the CP of the α-arrays generated by primitive polynomials of degree ≤ 16.

The total CP length for $m < M$ can now be upper bounded by

$$\mathcal{L}_{\text{UGM}} \leq 2m - 2 + m/2 + 1 + \lceil \log_2 M \rceil = 2.5m + \lceil \log_2 M \rceil - 1 \quad \text{[Gates]}$$

which is considerably better than the ~ $7m$ [Gates] of a previous UGM [Law71].

Complexity

The α-array consists of $M-1$ α-cells where each cell contains $M-1$ XOR gates, $M-1$ switches, $M-1$ multiplexers and $M-1$ drivers. Since switches and multiplexers are much simpler than XOR gates we approximate the complexity of a switch/multiplexer pair by that of one XOR gate. We let also the driver correspond to an XOR gate.

Then the complexity of the α-array can be estimated to $3(M-1)^2$ gates. The IP-network consists of M IP-cells where each cell contains $2M-1$ gates. Totally $2M^2-M$ gates for the IP-network. Finally we need $3M$ registers to store the vectors P, S and V needed to program the UGM (these registers are written from an external unit). The complexity $\mathcal{C}_{\text{UGM}}$ for the whole UGM can therefore be estimated by

$$\mathcal{C}_{\text{UGM}} = 3(M-1)^2 + 2M^2 - M + 3M \approx 5M^2 - 4M.$$

Compared to a previous UGM [Law71] with complexity $\approx 7M^2+3M$ the present UGM requires about 30% less components.

10.1.3 A CMOS Implementation of a Fast UGM for $2 \le m \le 16$

A bit-parallel UGM for $M = 16$, based on the ideas presented above, has been implemented at our department by M. Johannesson. The results of this implementation work have been reported in [Joh90]. We reproduce here the main results and conclusions.

The Layout

In Fig. 10.2 we have a large communication bus consisting of M^2 lines connecting the α-array to IP network. This bus is clearly inconvenient since it occupies a large area and is tedious to connect. The bus can be skipped completely by adopting a *distributed* IP-cell.

First we modify the IP-cell to have the parity-generating XOR gates organized in a chain rather than a tree, as shown in Fig. 10.5 where the switches are used to output the product coefficient c_i as soon as it is available, i.e. without having to propagate through the whole IP cell when $m < M$.

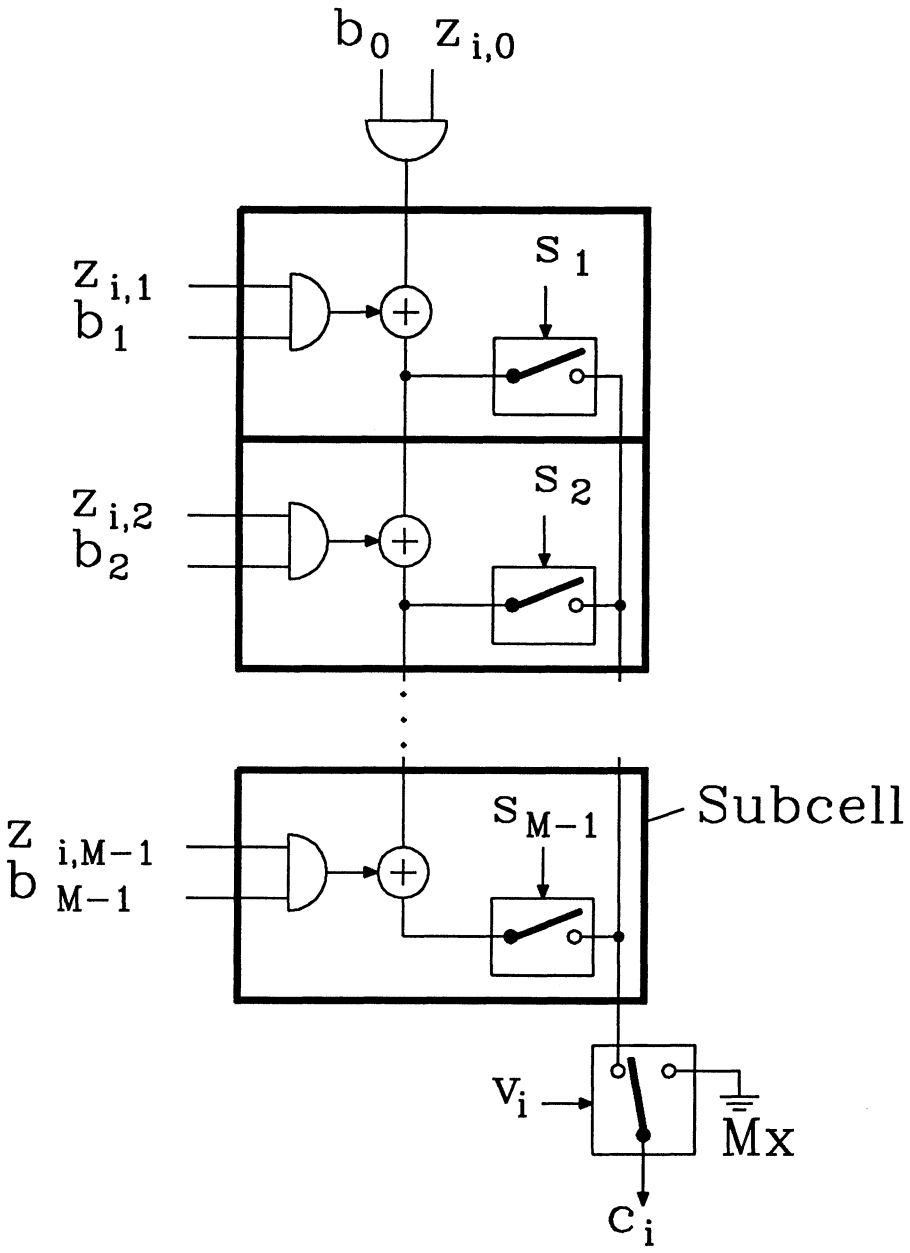


FIGURE 10.5 The new IP-cell for computing the inner product $Z_{i,-} \cdot B$.

The new arrangement implies that the CP through the IP-cell has length $m-1$ which is considerably more than the CP length of the original tree arrangement ($\sim \log m$). However, the computation of the inner products can now be carried out in concurrence with the computation of the matrix Z whereby the system performance should improve. The new IP-cell is distributed along the columns of the α-array in a way such that each of the $m-1$ subcells of Fig. 10.5 lies adjacent to one of the bit-slices of an α-cell. In this way we obtain a new bit-slice that includes also a subcell of the new IP-cell.

After various tests with different configurations it was decided to realize the new bit-slice as shown in Fig. 10.6 which we now comment.

The switch SW and the multiplexer MX are those indicated in Fig. 10.3. Driver 1 is used to refresh the signal coming from the multiplexer MX of the previous stage. The switch S1 is used to reduce the capacitance on the feedback line. Simulations have shown that having the XOR gates direct on the FB line results in longer propagation time. Driver 2 is a consequence of S1. The logic below MX is the IP subcell (from Fig. 10.5). The switch S2 is used to output the product coefficient c_i as soon as it is available.

The final floor plan of the UGM is shown in Fig. 10.7. The upper row of AND gates are the top gates of the IP cells (see Fig. 10.5). The left-most column of IP subcells (also from Fig. 10.5) is used to compute c_0. The bottom row is called the *S/V* generator and is used to derive the control vectors S and V from the input vector P (the field generator). There are thus no registers storing S, V and P. The *S/V* generator is an array of 15 identical bit-slices one of which is shown in Fig. 10.8. The 16 multiplexers (MX in Fig. 10.5) connecting the unused c_i's to zero are also contained in the *S/V* generator. This simplifies considerably the distribution of the v_i's.

The largest unit is the 15 by 15 matrix of bit-slices from Fig. 10.6. The physical layout of one such bit-slice is shown in Fig. 10.9.

The complete layout of the UGM is shown in Fig. 10.10. For purpose of comparison the architecture of the UGM in [Law71, Fig. 3] has also been implemented and its final layout is shown in Fig. 10.11. We call the latter UGM the *reference UGM*.

The layout work was done using the Electric CAD system. The technology adopted is 2-μm CMOS.

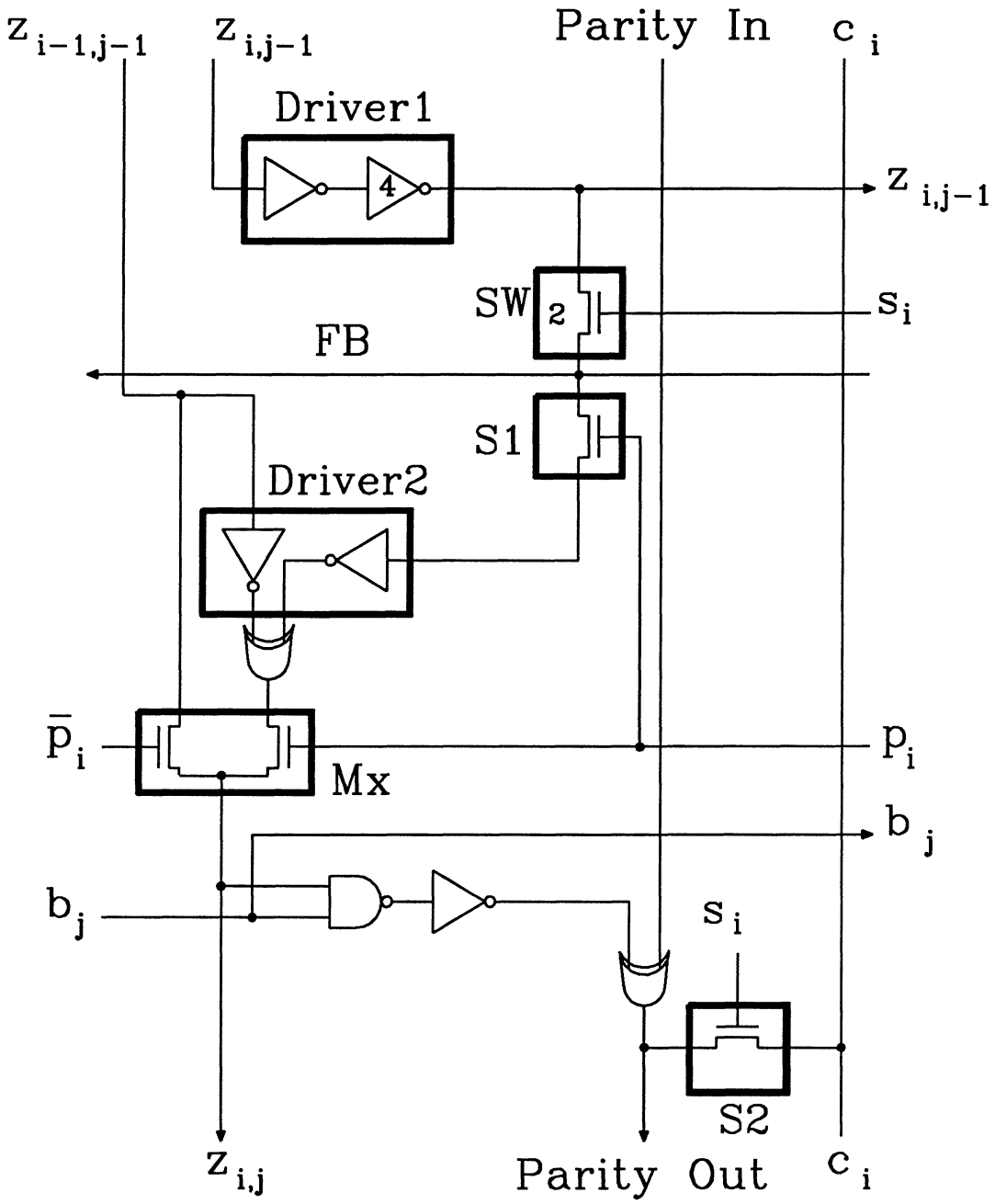


FIGURE 10.6 The new bit-slice.

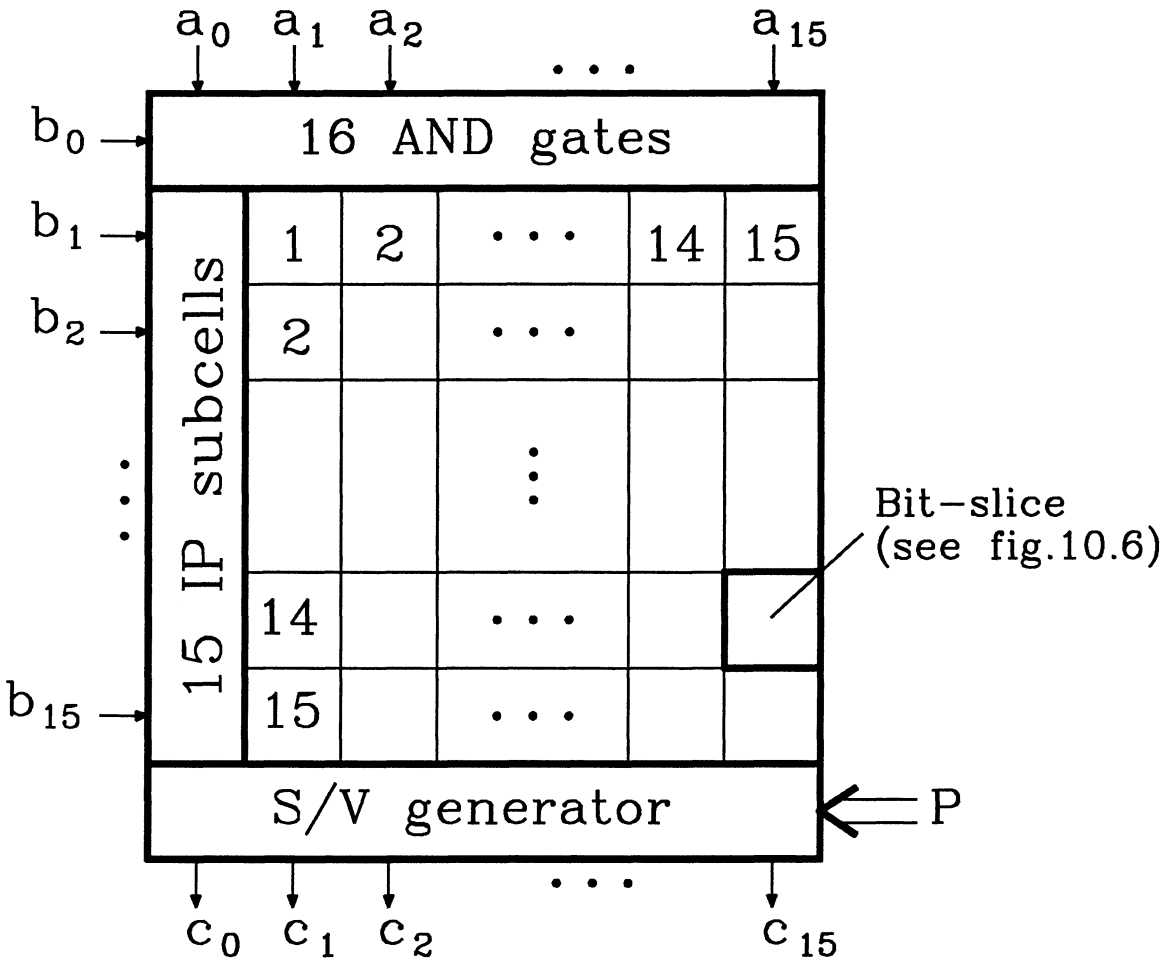


FIGURE 10.7 Final floor plan of a bit-parallel UGM for M=16.

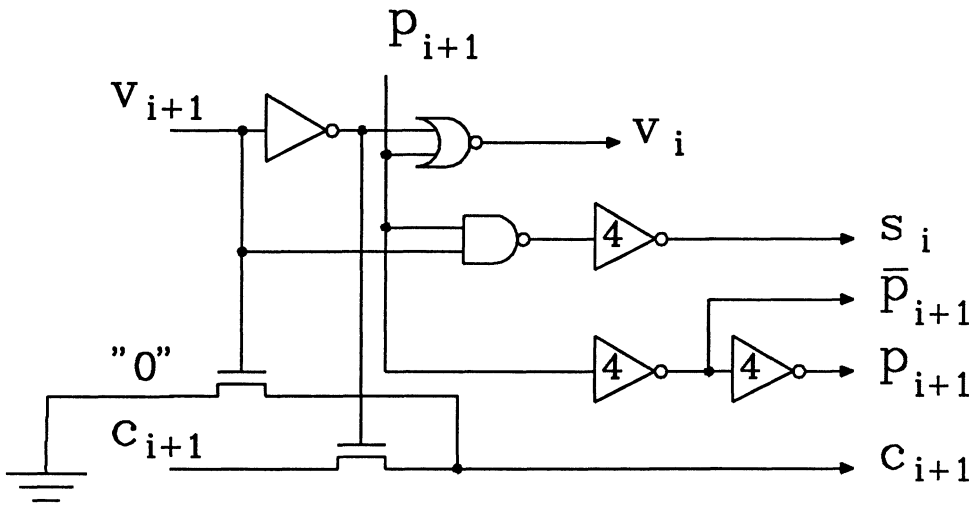


FIGURE 10.8 The bit-slice of the s/v generator.

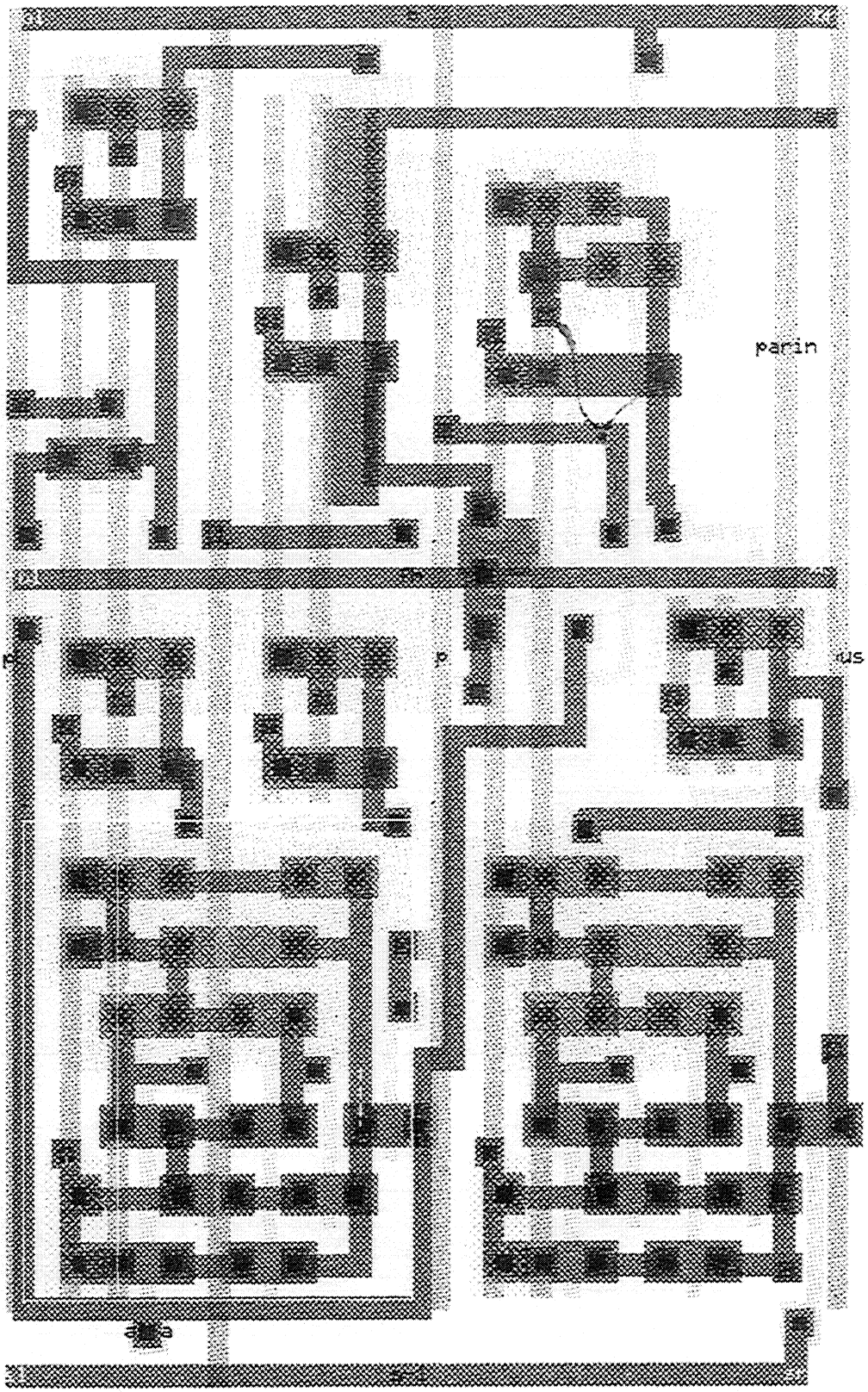


Figure 10.9 Final layout of the bit-slice of Fig. 10.6.

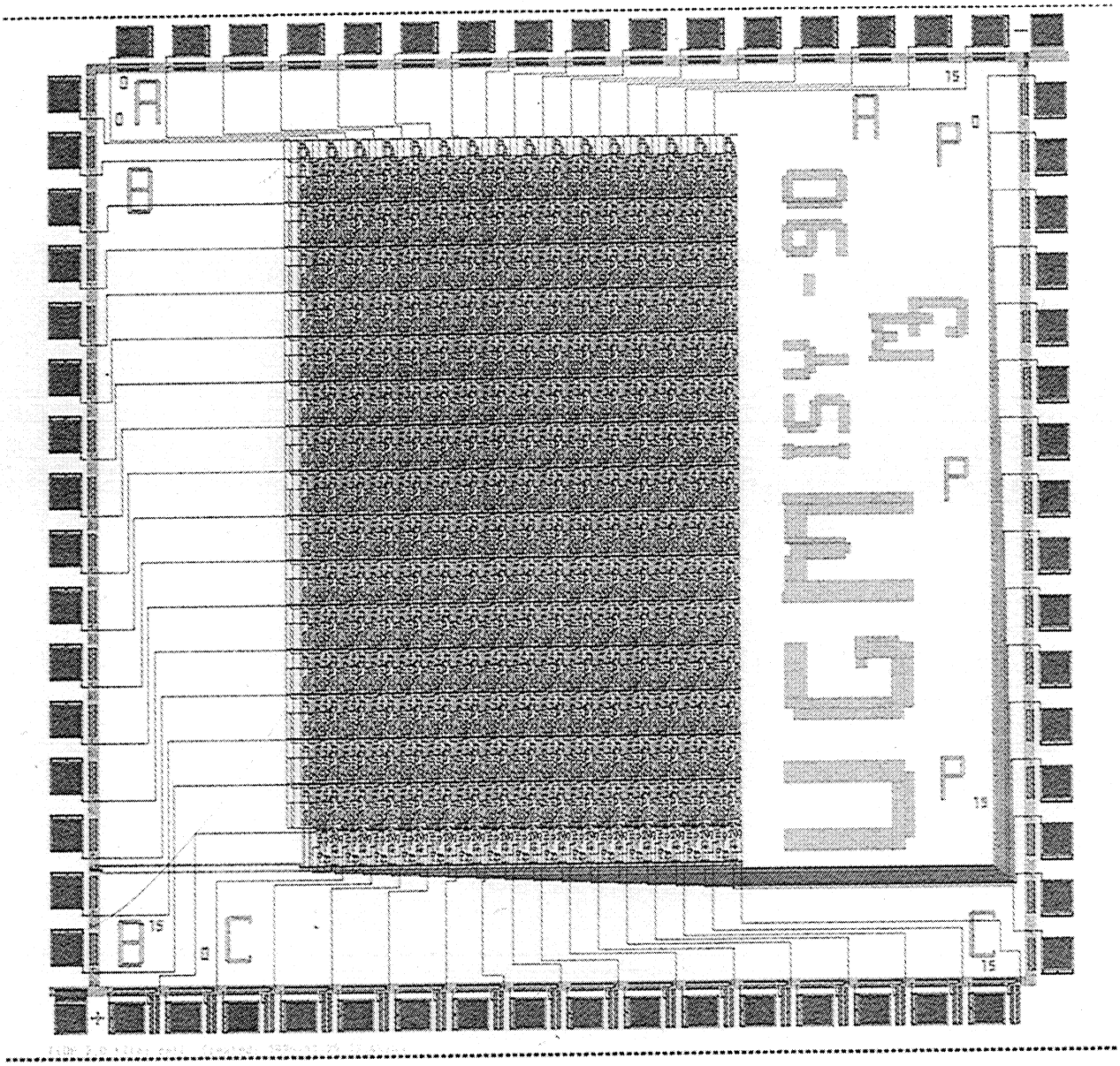


Figure 10.10 Final layout of the new UGM.

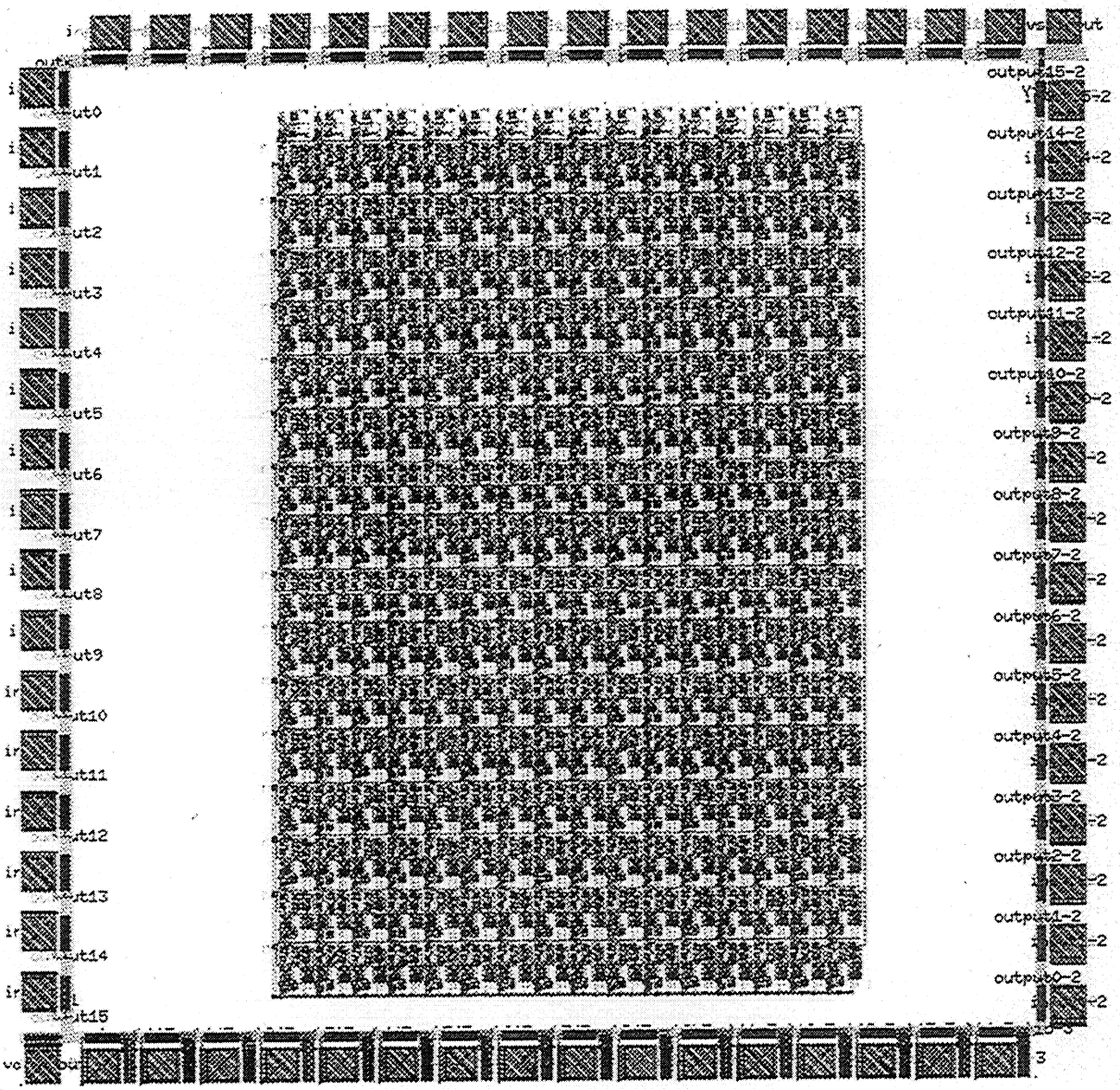


Figure 10.11 Layout of the reference UGM.

Complexity

The UGM used four different cells and a total of ~ 10000 transistors. The reference UGM used two different cells and a total of ~ 13000 transistors which makes the new UGM about 30% more efficient in component use.

Without pads, the UGM used an area of ~ 2×3.3 m^2 which translates into a density of ~ 1500 transistors/m^2. The reference UGM used ~ 2.6×3.8 m^2 with a density of ~ 1300 transistors/m^2.

Performance

Due to the new IP cell, the CP length has been estimated to ~ $3m$ [gates]. The CP length of the reference UGM has been found to be $8m$. The new UGM should thus offer ~ 63% better performance.

In order to verify the importance of choosing a good field generator the UGM was simulated with both the best $P(x)$ of Tab. 10.1 and the all-ones polynomial (i.e. worst-case for the α-array). The simulations were performed on coded versions of the multipliers where all logical devices were input as routines/functions in the C language. The results are shown in Tab. 10.2. We see that the performance of the best/worst case polynomial can differ by a factor of 2. We notice also that the reference UGM, although always slower, is much less sensitive to the particular choice of $P(x)$.

m	$P(x)$	New UGM	Reference UGM
16	best	63.6	153.4
16	all-ones	126.5	163.8
8	best	42.6	86.0
8	all-ones	64.4	98.1

Table 10.2 Simulated performance for different $P(x)$. The table gives the propagation times [ns] through the multipliers. The results concern C-programmed models of the multipliers, not the physical layouts.

Simulations based on the physical layouts without pads were also performed and indicated considerably better performance (consistently for

both UGM's) as shown in Fig. 10.12. A simulation of the new UGM with pads and $m = 16$ showed that the pads introduce an additional delay of ~ 7 ns.

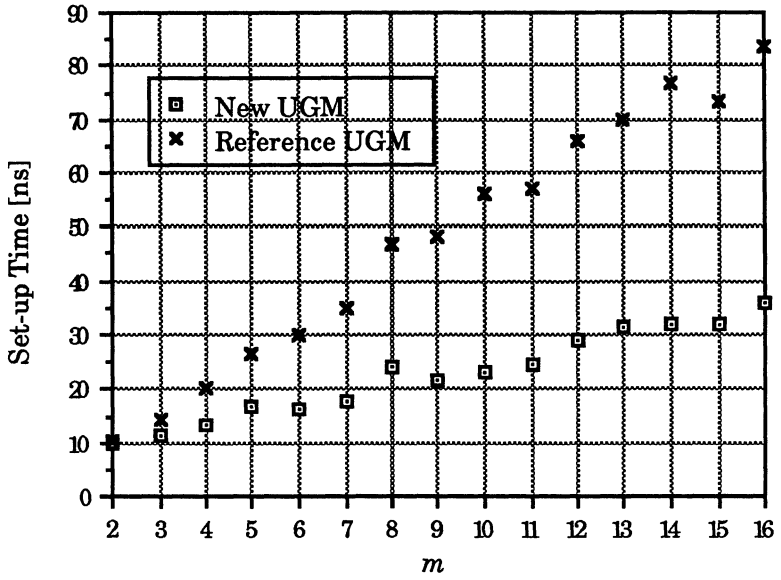


Figure 10.12 Simulated performance based on the physical layouts (no pads).

Fig. 10.12 shows that the new UGM is about 50% (instead of 63% as previously estimated) faster than the reference UGM. The missing 13% is believed to be a consequence of the capacitances on the feedback line FB and the output lines c_i.

The propagation time with pads for $m = 16$ has been simulated to ~ 43 ns which translates into a rate of ~ 23.2 millions multiplications per second.

The functionality of the new UGM has been tested through a limited number of randomly chosen multiplications in all fifteen fields, and was found to work properly.

The layout will be sent for fabrication in January 1991 and the chip will be available for tests/measurements a few months later. First by that time the real performance of the UGM will be established.

The chip has 66 pins, 16 for each of A, B, C and P plus 2 for power supply. The 16 (M in general) pins of P can be reduced to 4 ($\lceil \log_2 M \rceil$ in general) and used to address an on-chip memory storing the 15 ($M - 1$ in general) primitive polynomials.

The chip is estimated to dissipate ~ 0.05 W when clocked at 20 MHz.

10.2 Universal Galois Exponentiators

The PB exponentiation architecture of Sec. 8.2.2 (Fig. 8.8) makes use of two fixed MSR multipliers. It is therefore possible to realize a universal Galois exponentiator (UGE) with variable base and exponent by replacing the fixed multipliers with the bit-serial UGM of Sec. 10.1.1.

If the base α is fixed it is possible to modify the architecture of Fig. 8.5. We replace the MSR multiplier with the bit-serial UGM of Sec. 10.1.1 and the fixed α-cell with the universal α-cell of Fig. 10.3.

Sometimes, like in certain algorithms for erasure decoding of RS codes (see e.g. [Mas89:2, Fig. 5]), only consecutive powers of the element α, where α is a root of the field generator, are required. Then it is enough to provide a universal Galois-type LFSR, i.e. the upper half of Fig. 10.1. If the bit-serial UGM is already available in the system, this can be used as a universal Galois-type LFSR by setting the serial input to a constant 1.

10.3 Universal Galois Inverters

The PB inverter of Fig. 9.5 makes use of two fixed MSR multipliers. It is therefore possible to realize a universal Galois inverter (UGI) by replacing the fixed multipliers with the bit-serial UGM of Sec. 10.1.1.

10.4 Universal Galois Processors

Ordinary, commercially available digital signal processors (DSP) — e.g. TMS320C25, DSP32C or MB 8764 — are badly suited to applications of Galois fields. It would therefore be interesting to develop efficient *universal Galois*

processors (UGP) tailored to different fields of application like algebraic error-control codes (ECC) or cryptography. Universal Galois operators, multipliers in particular, are clearly fundamental for the development of UGP's.

Although a detailed specification, design and realization of a programmable UGP is beyond the scope of this dissertation, we are confident that such a UGP, while providing the flexibility of an ordinary DSP, would offer considerably better performance than an ordinary DSP. Moreover, the end-user would find the programming of the UGP less tedious since its instruction set would be matched to the field of application.

10.5 Discussion

In this chapter we have introduced several new universal architectures capable of operating in a range of Galois fields GF(2^m), $2 \leq m \leq M$. All architectures are based on the PB representation of a Galois field.

The most important universal operator is the UGM which has been realized in both serial and parallel form.

The serial UGM displays a high structural regularity and simplicity which makes it well suited for both small and large values of m.

The parallel UGM is modular and regular but its quadratic complexity makes it viable as along as m is relatively small. A CMOS implementation [Joh90] for $M = 16$ has shown that the proposed UGM is well suited for VLSI implementation. With the cells created for the above implementation it would be a very simple matter to implement an UGM for a new value of M.

We introduced also universal exponentiators and inverters with nice structural properties which make them viable for a wide range of m. Universal exponentiators are interesting in public-key cryptosystems where the possibility to change field from one session to another has the effect to increase the security of the system [Odl84].

The idea of creating special Galois processors is not new. In [Ber79] a (fixed) Galois field computer is described which operates over GF(32).

We have proposed the realization of *universal* Galois processors (UGP) that would offer a much higher degree of flexibility than the their fixed

relatives while still providing considerably higher performance than ordinary DSP's.

Consider for example the field of algebraic ECC. Adopting the above parallel UGM one could realize an UGP that makes use of pipelining, concurrency and other modern techniques of microprocessor design. Such a UGP could be used for the encoding/decoding of many, many different ECC's of practical interest. If implemented in a small CMOS technology (like 0.7 μm) the UGP is believed to allow for throughputs in the range of 1-5 Mbits/s depending on the actual ECC.

Universal Galois operators could also be used as coprocessors to ordinary DSP's.

In our discussion we did not mention the possibility of designing universal Galois squarers since squaring would normally be performed as a multiplication by an UGM. However, it might very well be possible to design a universal squarer, e.g. based on the results of Sec. 7.1, that is both faster and less complex than the parallel UGM.

Finally, we point out that an universal RS encoder could be realized basically by replacing the (fixed) α-array of Fig. 5.9 with an (universal) α-array consisting of universal α-cells.

Chapter 11

Conclusions and Further Research

In this thesis we have investigated the three major representations (polynomial-basis PB, normal-basis NB, dual-basis DB) of Galois fields with respect to the development of architectures for computations in GF(2^m).

In particular, we have sought *fixed* and *universal* architectures for

- parallel, serial, constant and hybrid multiplication,
- squaring and square rooting,
- exponentiation,
- division

that are suitable for implementation in current VLSI technologies.

Our *general conclusion* is that the PB representation is by far the most versatile since it appears to offer excellent or, at least, good solutions to most computational problems. Furthermore, the fact that good PB's are easily found, that no basis-conversion problems arise, and that simple, compact descriptions of the algorithms are often available, makes the PB the easiest to use.

The DB representation offers the best solution for bit-serial multiplication by a constant. We found however that the advantages over the corresponding PB solution may, in certain applications (e.g. RS encoders), be very marginal.

The main feature offered by the NB representation is the fact that squares and square roots are easily computed. This feature is unfortunately penalized by multiplier architectures which are heavily dependent on the availability of optimal or, at least, true low-complexity NB's. Optimal NB's are rare while there are at present very few constructions of true low-complexity NB's. On-going research might soften the last drawback in a near future.

Given a good basis, the NB representation offers good architectures for exponentiation and multiplicative inversion. The best structural properties, however, are found in the corresponding PB architectures.

Remark: During the preparation of the last chapters, we became aware of some recent results which, due to time constraints, have not been included in this thesis. These recent results are reported in [Afa90] and [Gom90].

In [Afa90] the author (very briefly) introduces a new algorithm for fast multiplication which appears to significantly reduce the number of gates required. This reduction in component count seems to be penalized by an equally significant increase in CP length. However, the structural properties of the architectures have to be investigated before the attractiveness of the algorithm is assessed.

In [Gom90] the author discusses (in German) various VLSI architectures for cryptographic applications. Some of the conclusions drawn there *seem* to agree with some of the conclusions drawn in Ch. 3 and Ch. 8 in this thesis (we say "*seem*" because our knowledge of the German language is unfortunately rather limited).

Further Research

In order to push further the results of our investigation, it is necessary to gain extensive hands-on experience through involvement in physical VLSI implementations of various architectures. Such experience is almost mandatory when large-fields architectures are considered. We are therefore planning to extend our small-fields VLSI experience to fields GF(2^m) with $m \sim 1000$ through implementation of some exponentiation architecture.

During our investigations, a number of questions/problems remained without answers/solutions and new questions/problems arose:

- It appears to be very difficult to find good structures for fast, parallel computation of multiplicative inverses. Actually, it is difficult to even find the Boolean functions defining the inverse.

- The use of the Euclidean algorithm for computing inverses has not really been investigated. It would be interesting to establish if it is a relevant alternative in the context of VLSI design.

- Hybrid architectures have been introduced only for sequential multiplication. How about the other operations ?

- There seems to be very little published about prime and, in particular, primitive polynomials over composite fields. Since these are fundamental to hybrid architectures, it would be interesting to extend the few known (and small) tables.

- In [Mul89] the authors introduce an optimal NB which they call the type-II construction. Our limited tests indicate that this construction leads to *primitive* NB's. It would be interesting to know if this is always the case. If so, one has also found a nearly systematic way to identify a relatively large number of primitive polynomials.

- Galois fields with odd characteristic p are very important to public-key cryptography. They are, on the other hand, never considered in the practice of error-control codes. It is maybe time to change this attitude by investigating, for instance, the possibility of adopting p-level logic.

- The three bases considered here have strong algebraic structures. Are there other bases with useful algebraic structures ?

Appendix A

Table of Optimal Normal Bases [Mey90]

2*	3	4	5	6	9	10	11	12	14	[18]	23
26	28	29	30	33	35	36	39	41	50	51	52
53	58	60	65	66	69	74	81	82	83	86	89
90	95	98	99	100	105	106	113	119	130	131	134
135	138	146	148	155	158	162	172	173	174	178	179
180	183	186	189	191	194	196	209	[210]	221	226	230
231	233	239	243	245	251	254	261	268	270	273	278
281	292	293	299	303	306	309	316	323	326	329	330
338	346	348	350	354	359	371	372	375	[378]	386	388
393	398	410	411	413	414	418	419	420	426	429	431
438	441	442	443	453	460	466	470	473	483	490	491
495	508	509	515	519	522	530	531	540	543	545	546
554	556	558	561	562	575	585	586	593	606	611	612
614	615	[618]	629	638	639	641	645	650	651	652	653
658	659	660	676	683	686	690	700	708	713	719	723
725	726	741	743	746	749	755	756	761	765	771	772
774	779	783	785	786	791	796	803	809	810	818	820
826	828	831	833	834	846	852	858	866	870	873	876
879	882	891	893	906	911	923	930	933	935	938	939
940	946	950	953	965	974	975	986	989	993	998	1013
1014	1018	1019	1026	1031	1034	1041	1043	1049	1055	1060	1065
1070	1090	1103	1106	1108	1110	1116	1118	1119	1121	1122	1133
1134	1146	1154	1155	1166	1169	1170	1178	1185	1186	1194	1199
1211	1212	1218	1223	1228	1229	1233	1236	1238	1251	1258	1265
1269	1271	1274	1275	1276	1278	1282	1289	1290	1295	1300	1306
1310	1323	1329	1331	1338	1341	1346	1349	1353	1355	1359	1370
1372	1380	1394	1398	1401	1409	1418	1421	1425	1426	1430	1439
1443	1450	1451	1452	1454	1463	1469	1478	1481	1482	1492	1498
1499	1505	1509	1511	1518	1522	1530	1533	1539	1541	1548	1559
1570	1583	1593	1601	1618	1620	1626	1636	1649	1653	1659	1661
1666	1668	1673	1679	1685	1692	1703	1706	1730	1732	1733	1734
1740	1745	1746	1749	1755	1758	1763	1766	1769	1773	1778	1779
1785	1786	1790	1791	1806	1811	1818	1821	1829	1835	1838	1845
1850	1854	1859	1860	1863	[1866]	1876	1883	1889	1898	1900	1901
1906	1923	1925	1926	1930	1931	1938	1948	1953	1955	1958	1959
1961	1965	1972	1973	1978	1983	1986	1994	1996	2001	2003	2006
2009	2010	2026	2028	2039	2045	2046	2049	2052	2055	2063	2066
2068	2069	2078	2079	2082	2098	2109	2114	2115	2121	2126	2129
[2130]	2140	2141	2163	2174	2178	2181	2186	2195	2198	2212	2220
2223	2225	2231	2236	2241	2242	2246	2253	2258	2266	2268	2273
2291	2292	2295	2301	2308	2310	2318	2319	2332	2338	2339	2345
2351	2356	2361	2370	2388	2391	2393	2394	2399	2406	2415	2436

The above table shows the values of $m \leq 2346$ for which an optimal NB exists. The underlined values denote type-I NB's. The values in the boxes are such that both a type-I and a type-II NB exists. The remaining values are all type-II NB's.

Appendix B

Complexities for Normal Bases in Mersenne Prime Fields [Ash89]

m	N_m
2	3
3	5
5	9
7	21
13	45
17	81
19	≤189
31	≤309
61	345
89	177
107	621
127	501
521	≤16671
607	3621
1279	≤12789
2203	13197
2281	13665
3217	≤51471
4253	≤85059
4423	26517
9689	19377
9941	≤119291
11213	≤224259
19937	≤279117
21701	43401
23209	≤232089
44497	266961
86243	517437
132049	528189

Appendix C

```
(* Program for computing the complete profile of the alfa-array associated *)
(* with the prime polynomial px. Created on a μVAX-II in VAX-VMS enviroment. *)
program profile(input,output);
const    max_m=32;
var  i,j,k,m:integer;
       px:array[1..max_m] of integer; (* The prime polynomial  *)
  prof_mat:array[1..max_m-1,1..max_m] of integer; (* The profile matrix *)
procedure init_input; (* Initialize and get prime polynomial from keyboard*)
begin
 for i:=1 to max_m do px[i]:=0;
 for i:=1 to max_m-1 do for j:=1 to max_m do prof_mat[i,j]:=0;
 writeln;
 write('enter x-powers of non-zero terms of P(x), except LSB, MSB first:');
 read(i); (* Read highest coefficient => degree of P(x)*)
 px[i+1]:=1;
 m:=i;
 while not eoln do
 begin
   read(i);
  px[i]:=1;
  prof_mat[1,i+1]:=1;
 end;
 readln;
 writeln;
end;
procedure gen_prof_mat; (* Compute profile matrix *)
begin
 for i:=2 to m-1 do
 begin
   prof_mat[i,1]:=prof_mat[i-1,m];
  for j:=2 to m do
   if (prof_mat[1,j] = 0) then prof_mat[i,j]:=prof_mat[i-1,j-1]
    else prof_mat[i,j]:=MAX(prof_mat[i-1,j-1]+1,prof_mat[i-1,m]+1)
 end
end;
procedure output_prof; (* Output result on screen *)
begin
 writeln;
 for i:=1 to m-1 do
 begin
   for j:=1 to m do write(prof_mat[i,j]:3);
   writeln
 end;
 writeln;
end;
begin (* Main Program *)
 while true do
 begin
   init_input;
   gen_prof_mat;
  output_prof
 end
end.
```

Appendix D

An Alternative Proof of Proposition 4.15

The reduction matrix is

$$Q = \begin{pmatrix} 1\,1\,0\,0 \ldots 0\,0\,0\,0 \\ 0\,1\,1\,0 \ldots 0\,0\,0\,0 \\ 0\,0\,1\,1 \ldots 0\,0\,0\,0 \\ : \; : \; : \; : \; : \; : \; : \; : \; : \; : \\ 0\,0\,0\,0 \ldots 0\,1\,1\,0 \\ 0\,0\,0\,0 \ldots 0\,0\,1\,1 \end{pmatrix}$$

and has ones only on the diagonal (as defined in Sec. 4.4) and the entries soon above it. That is,

$$q_{k,i} = 1 \qquad \text{if } i = k \text{ or } i = k+1\,,\ k = 0, 1, \ldots, m-2.$$

The maximum width WD_{max} is, by Prop. 4.11,

$$WD_{max} = \max\{2, 1+1\} = 2$$

which is best possible and implies $L_f = 1$.

In order to determine Z we look at the coefficient c_i as we did in eq. (4.10)

$$c_i = s_i + \sum_{k=0}^{m-2} q_{k,i} s_{k+m} = \begin{cases} s_0 + s_m & i = 0 \\ s_i + s_{m+i-1} + s_{m+i} & i = 1, 2, \ldots, m-2 \\ s_{m-1} + s_{2m-2} & i = m-1. \end{cases} \qquad \text{(D.1)}$$

With the help of eq. (D.1) and eq. (4.4) we can create a table similar to Tab. 4.1 for each c_i and identify all products $b_u a_v$ involved there. This is done in Tab. D.1.

		u:	0	1	2	3	4	...	$m-3$	$m-2$	$m-1$
c_0	s_0	v:	0								
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
c_1	s_1	v:	1	0							
	s_m	v:		$m-1$	$m-2$	$m-3$	$m-4$	...	3	2	1
	s_{m+1}	v:			$m-1$	$m-2$	$m-3$	...	4	3	2
c_2	s_2	v:	2	1	0						
	s_{m+1}	v:			$m-1$	$m-2$	$m-3$	...	4	3	2
	s_{m+2}	v:				$m-1$	$m-2$	...	5	4	3
	:						:	:	:	:	:
c_{m-2}	s_{m-2}	v:	$m-2$	$m-3$	$m-4$	$m-5$	$m-6$	...	1	0	
	s_{2m-3}	v:								$m-1$	$m-2$
	s_{2m-2}	v:									$m-1$
c_{m-1}	s_{m-1}	v:	$m-1$	$m-2$	$m-3$	$m-4$	$m-5$	...	2	1	0
	s_{2m-2}	v:									$m-1$

Table D.1

From the above table we obtain

$$c_0 = b_0a_0 + b_1a_{m-1} + b_2a_{m-2} + \cdots + b_{m-2}a_2 + b_{m-1}a_1$$

$$c_1 = b_0a_1 + b_1(a_0 + a_{m-1}) + b_2(a_{m-2} + a_{m-1}) + \cdots + b_{m-2}(a_3 + a_2) + b_{m-1}(a_2 + a_1)$$

$$c_2 = b_0a_2 + b_1a_1 + b_2(a_0 + a_{m-1}) + \cdots + b_{m-2}(a_{m-2} + a_{m-1}) + b_{m-1}(a_3 + a_2)$$

:

$$c_{m-2} = b_0a_{m-2} + b_1a_{m-3} + b_2a_{m-4} + \cdots + b_{m-2}(a_0 + a_{m-1}) + b_{m-1}(a_{m-2} + a_{m-1})$$

$$c_{m-1} = b_0a_{m-1} + b_1a_{m-2} + b_2a_{m-3} + \cdots + b_{m-2}a_1 + b_{m-1}(a_0 + a_{m-1})$$

which yield the matrix Z. A closer look at Tab. D.1 reveals that there are $m-1$ distinct functions $f_{i,j}$ of width 2 and that these can be read out from c_1. The functions are

$$a_2 + a_1\,,\ a_3 + a_2\,,\ a_4 + a_3\,,\ \ldots\ ,\ a_{m-1} + a_{m-2}\,,\ a_0 + a_{m-1}.\ \square$$

Appendix E

Summary of Galois' Life and Work

Evariste Galois was born in Bourg-la-Reine near Paris, on 25 October 1811. His father, M. Nicolas Gabriel Galois, was first headmaster of the local boarding school and later mayor of the town. His mother, Adélaide-Marie Demante Galois, came from a family of jurists, and it was she who gave Evariste his early, mainly classical education. Aged 12, Galois was sent to the Collége Louis-le-Grand in Paris in 1823. Four years later he started to make himself familiar with the work of Lagrange and Legendre and by 1828 he was busy mastering the most recent works on the theory of equations, number theory, and elliptic functions. He read, among other books, Lagrange's *Résolution des équations numériques* where he found the following definition of algebra in the introduction (transl. from [Inf48]):

"Algebra, as usually understood, is the art of determining unknown quantities as functions of known quantities or those assumed to be known; and also the art of finding a general solution of equations. Such a solution consists in finding, for all the equations of the same degree, such functions of the coefficients of the algebraic equations that represent all its roots.

Up to present, this problem can be regarded as solved only for equations of the first, second, third and fourth degrees. ..."

Galois quickly came to believe that he had solved the general fifth-degree equation. Like Abel before him, he discovered his error, and that discovery launched him on his search for a solution to the general problem of solvability of algebraic equations.

In July 1829 his father, victim of a political plot which unjustly discredited him, committed suicide. In August Galois failed to enter the Ecole Polytechnique because of his inability to submit to the generally required methods of procedure. Later that year he entered the Ecole Normale Supérieur (ENS).

In February 1830 a paper submitted to the Académie des Sciences was not only rejected, the examiner even lost his manuscript and Galois again felt he was being persecuted, as his father had been. Four months later, however, he succeeded in having a paper on number theory published in the

prestigious *Bulletin des sciences mathématiques* [Gal97, pp. 14-23]. It was this paper that contained the highly original and diverting theory of "Galois imaginaires".

Wishing to solve congruences $f(x) \equiv 0 \pmod{p}$ where p is a prime and f is irreducible of degree n (over Z_p) he introduced, in analogy with the complex numbers where i acts as a root of $x^2 + 1 = 0$, a hypothetical element j to act as a root for f and considered the p^m formally distinct expressions $a_0 + a_1 j + \cdots + a_{n-1} j_{n-1}$ where the a_k's are integers of Z_p. These are the expressions leading to a finite field, the Galois field $GF(p^n)$.

After this, Galois, who was a fierce republican, became involved in the July Revolution of 1830 that drove Charles X off the throne. The same year he was expulsed from the ENS for openly criticizing its director.

In 1831 he was imprisoned twice for his political activities. In prison he continued his mathematical research. Shortly after his release in 1832 he was persuaded into a duel, perhaps arising from a love affair. The event remains mysterious. It is though evident that he expected to be killed. On 29 May 1832, in a letter written in fervish haste, he outlined the principal results of his mathematical investigations.

Galois' principal achievement was to arrive at a definitive solution to the problem of the solvability of the algebraic equations, and in doing so to produce such a breakthrough in the understanding of fields of algebraic numbers and also of groups that he is considered as the chief founder of modern group theory. What has come to be known as the Galois theorem made immediately demonstrable the insolubility of higher-than-fourth-degree equations by radicals. The theorem is found in [Gal97, pp. 33]:

"... The reader will find here the general *condition* which must be *satisfied by all equations solvable by radicals* and which, conversely, assures their resolvability. An application is made only to equations whose degree is a prime number. Here is the theorem given by our analysis:

For an equation of prime degree, which has no rational divisor, to be solvable by radicals, it is necessary and sufficient that all its roots be rational functions of any two among them. ..."

The letter [Gal97, pp. 24-31] was sent to Auguste Chevalier, almost certainly in the expectation that it would find its way to Karl Gauss and Karl Jacobi. The duel took place on the following morning. Galois was severely wounded in the stomach and died in hospital on 31 May 1832.

So brief was Galois' life he was only 20 when he died and so dismissively were his ideas treated by the Academy, that he was known to his contenporaries only as an ardent republican.

To the mathematicians of today, familiar with the words "Galois group", "Galois field", "Galois theory", he is known as one of greatest mathematicians of all ages, who died in his youth in a duel.

But as long as he lived he was both.

Further details on Galois' life and work can be found in [Inf48], [Gal62], [Gal97], [Abb85].

Appendix F

Aspects of VLSI Design

The Design Process

The design process can be viewed as a sequence of transformations on behavioral, structural and physical design representations, at various levels of abstractions. For example, *functional* (or *behavioral*) *design* defines the behaviour of system outputs as function of the inputs; *logic design* manipulates structural descriptions (e.g. schematic or logic diagram) while preserving functionality. *Physical design* concerns either transformations on or transformations into information to be used in the manufacturing process.

Design Representations

In the design process several different representations are used to show different aspects of the system under design.

Behavioral representations describe a circuit's function. For example, Boolean expressions and procedural descriptions (e.g. IF $enable = high$ THEN $a := a \times b$) are behavioral representations.

Structural representations describe the composition of circuits in terms of cells, components and interconnections among the components. Such descriptions are normally hierarchical, i.e. a component at one level may be decomposed into constituent components until elementary components are reached. Block diagrams, schematic drawings, and net lists of logic gates are examples of structural descriptions.

Structural representations say nothing explicit about the functionality of a circuit.

Physical representations are characterized by information relevant to the manufacture or fabrication of physical systems. Physical information may be a geometric layout or a topological constraint.

Physical representations provide only an implicit description of a circuit's behaviour.

Methodologies

A structured design methodology is mandatory for the design of VLSI circuits, not only to cope with the high design complexity but also to increase the design efficiency and the probability for an error-free design, and to reduce the amount of design data.

Fig. F.1 depicts the different phases of the commonly used *top-down* design methodology.

Specification is normally a manual step. Some important factors to be considered here are the application of the system, the performance requirements, the architecture of the system, the external interfaces, the economic and temporal constraints, the manufacturing technology, and the design tools available. This phase can be very time-consuming.

Functional design. In this phase a functional behaviour is synthesized to meet the specifications. The output may, for example, be a behavioral representation.

Logic design results in a logic structure that implements the functional design. The result may for example be a schematic description.

The *circuit design* phase is concerned with the detailed, electrical behaviour of the basic circuit elements (e.g. transistors, capacitors, resistors). Here we may, for example, have to resize transistors to meet propagation time requirements.

In the *physical design* phase we transform the result of the previous phase into the geometrical shapes used in the fabrication process.

Each of the above design phases is in turn subdivided in synthesis, analysis and verification.

Synthesis derives a new or improved representation based on the representation derived in the previous phase.

Analysis is concerned with the evaluation of a design representation against its requirements (e.g. area, performance, power consumption).

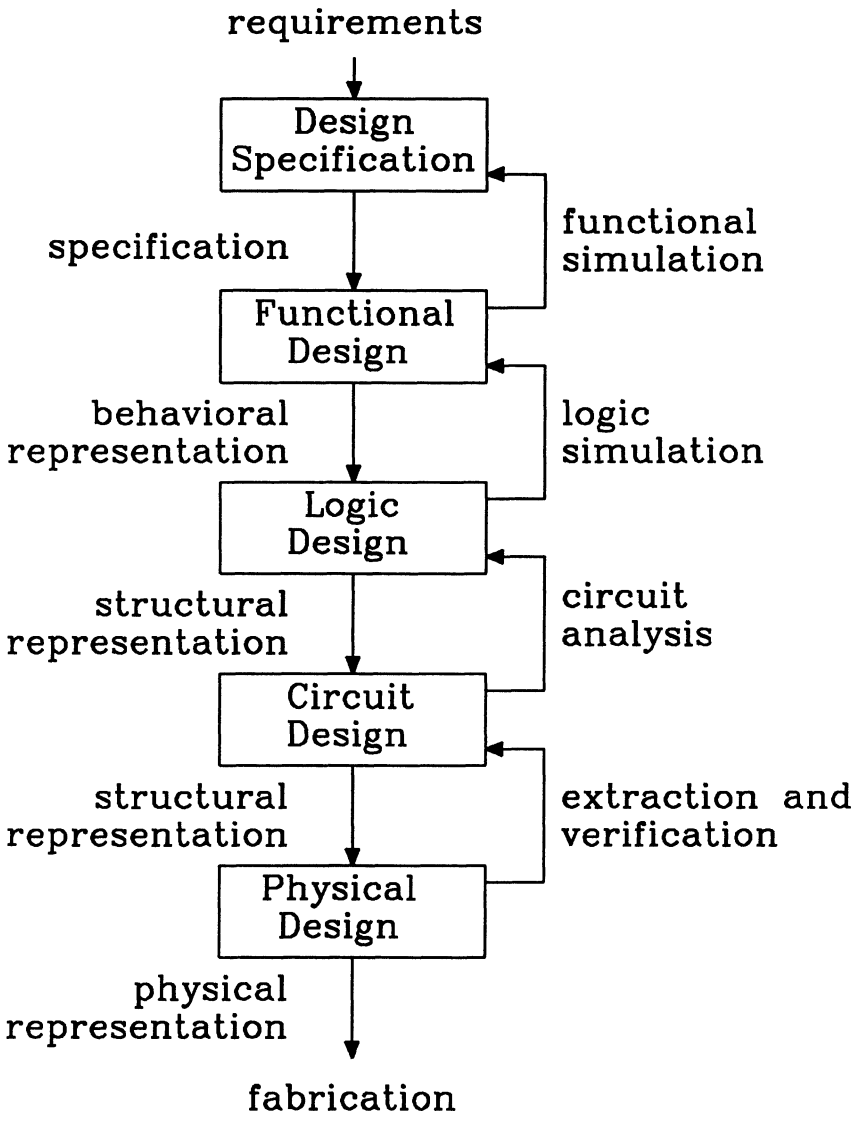


FIGURE F.1 The phases of electronic system design for a top-down design methodology.

Verification involves a formal process for proving that the synthesized representation is equivalent, under all specified conditions, to another representation. This is often an impossible task whereby designers in practice apply a less rigorous correctness check called *validation*.

Degree of Automation

Although computer aid in electronic design is widely spread, the degree of automation varies considerably and can be coarsely classified as follows.

Handcrafted design is a less constrained design method, seldom used to create large systems. Its use is limited, for example, to creating small cells or to finishing any incomplete wiring left by an automatic router.

Computer-aided design systems provide tools to synthesize, analyse and verify portions of the design but require active participation of the designer. The designer must thus supply a degree of creativity while the CAD system takes care of the tedious tasks.

In *automated design* the designer provides an abstract specification of the object to be designed and the design systems generates the physical design automatically and verifies that the design meets the specifications. Today no genuine automatic design systems exist rather these are subject of intensive research (e.g. [Bra89]).

Complexity Management

The only way to manage the complexity of large systems is by adoption of a structured design method.

A structured design method provides the simplification necessary to successfully complete the design process. Good methods for simplification are abstraction, regularity and standardization.

Abstraction replaces an object with a simplified model that defines the interaction of the object with its environment. Any details of internal organization or implementation are hidden in the model. When dealing with large systems, one has to introduce several levels of abstractions in order to reduce objects to manageable sizes. One speaks then of an *hierarchy* of information where the greatest detail is evident only at the lowest levels. Two important rules to follow in order to make a hierarchical decomposition of greatest value are modularity and locality.

Modularity. Hierarchy involves dividing the system into a set of modules. A system whose modules are characterized by a well-defined, unambiguous functional interface is called *modular*. Modularity helps a designer to understand and document his design, and also allows a design system to be of greater utility by checking attributes of a module as it is constructed. Furthermore, modularity facilitates team design where a number of designer work on different portions of a chip.

Locality implies that the details outside a component are not important when considering the interior of a component.

Regularity is often used to reduce, sometimes dramatically, complexity. A modular design having a few types of modules has a high degree of regularity. Regularity can exist at all levels of abstraction. At circuit level, uniform transistors might be used rather than individually optimized devices. A the structural level, identical gate structures might be employed. At a higher level, one might design architectures that use several identical processor structures.

Standardization that restricts the design domain can be applied at all levels of the design to simplify modules and thereby reduce complexity and the possibility of errors.

Layout Styles

There are several major design styles which are coarsely divided into *semi-custom* and *full-custom* approaches. A better classification is obtained by identifying the styles according to the amount of standardization imposed on the layout.

Gate arrays and *sea of gates* are *programmable arrays* in that they do not implement any specific function when they are manufactured. The array is such that it can be later customized to obtain the specified function. The customization of programmable arrays involve only those mask layers that are unique to the particular function, normally one to four mask layers.

In *gate arrays* the design involves only customization of devices into gates, placement of circuit functions (gates), and interconnection of the gates. The placement of the transistors, contacts, and routing areas is defined by the vendor. Efficient tools exist today that take as input a list of gates and a net list defining the interconnections, and produce a complete gate-array layout.

In the *sea-of-gates* approach the areas dedicated for wiring have been removed and the gate pattern has been made uniform. The device density is therefore much higher than for ordinary gate arrays. The wiring is done over unused devices, and contacts to active devices are created whenever needed. Sea of gates can provide high performance.

In *standard-cell*, *unconstrained-cell* and *unconstrained design* all mask layers are required whereby the manufacturing cost is higher than for programmable arrays.

Standard-cell systems rely on a set of predefined logic cells. The cells have fixed height, variable length, and are placed in rows in a standardized floor plan. Routing areas are provided between the rows. The device density and performance are higher than for gate arrays. Efficient tools for automatic layout of standard-cell chips exist today.

In an *unconstrained-cell* design no restrictions are put on the shape, area and placement of the cells/modules, or the wire routing. The layout of low-level cells can be done in detail but is time-consuming. Device density and performance are very high.

Unconstrained design is the most powerful and flexible design method. The designer has full freedom to specify all design parameters, transistor sizes, placement, routing, etc. This gives full control over area, speed, power consumption, etc. for the devices and the whole chip. The price is that the design effort becomes very large and costly, and the lack of any standardization makes most design tools useless. Consequently, unconstrained design is suitable only for critical circuits where performance is a primary concern.

Procedural Design

Procedural (or *program driven*) *design* systems may be algorithmic, such as module generators or silicon compilers, or may be rule based, expert systems. These are all semi-automatic or automatic design tools which are current topics of research.

The design of cells or large modules can be done by a *module generator* which directly generates the layout. The input description is typically a set of boolean equations. Module generators are particularly useful for layout of

regular arrays (e.g. ROM, RAM) but can also be applied on random logic modules [Bra89].

A *silicon compiler* is an automatic design system that can produce a complete layout of a whole or a part of a chip from a high-level functional or behavioral description.

Expert systems are useful for solving complex and unstructured problems for which an algorithmic solution is unknown.

Technologies

En early VLSI circuit technology was the *n-channel metal oxide semiconductor* (*n*MOS) technology based on the *n*-channel MOS field effect transistor (MOSFET).

Presently, the most important and widely used VLSI circuit technology is the *complementary metal oxide semiconductor* (CMOS) technology which makes use of both *n*-channel and *p*-channel MOSFET transistors. The main advantages of CMOS over *n*MOS are: lower power dissipation, higher noise margin, ratioless logic, higher speed, and the possibility to use circuit techniques based on transmission gates that are almost perfect switches. Disadvantages of CMOS compared to *n*MOS are: more devices per logic function and a more complex manufacturing process. CMOS is expected to be the main VLSI technology for several years to come.

Another interesting, emerging technology is the Gallium Arsenide (GaAs) technology [Cat90]. Because electrons move faster through the GaAs crystal lattice than through silicon crystal, GaAs devices are two to four times faster than their silicon rival, including state-of-the-art ECL (Emitter-Coupled Logic). Power consumption is lower than in ECL.

GaAs circuits are better suited than CMOS in environments subject to ionizing radiation which makes the technology highly interesting for e.g. satellite and deep-space communications. Optical communications is also a potential beneficiary since the high bandwidth of optical-fiber transmission demand speeds beyond the capabilities of CMOS technology. Today, digital GaAs complexity in volume production is limited to ~ $3 \cdot 10^4$ gates which limits the applicability to smaller systems. Furthermore, the yield is lower than for silicon circuits since the technology is not as mature. The two latter drawbacks are however just a matter of time.

Bibliography

[Abb85] D. Abbott (Gen. Editor), *Mathematicians*, London: Blond Educational, 1985.

[Afa90] V.B. Afanasyev, *Complexity of VLSI Implementation of Finite Field Arithmetic*, Proceed. of II Intern. Workshop on Algebraic and Combinatorial Coding Theory, pp. 6 - 8, Leningrad, USSA, 1990.

[Aho74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley 1974.

[All83] R.B.J.T Allenby, *Rings, Fields and Groups*, London, Edward Arnold Publ., 1983.

[Ash89] D.W. Ash, I.F. Blake, S.A. Vanstone, *Low Complexity Normal Bases*, Discrete Appl. Math. 25, pp. 191-210, 1989.

[Bar63] T.C. Bartee and D.I. Schneider, *Computations with Finite Fields*, Inform. Contr., vol. 6, pp. 79-98, Mar. 1963.

[Ben76] B. Benjauthrit, I.s. Reed, *Galois Switching functions and their Applications*, IEEE Trans. Comput., Vol. C-25, pp. 78-86, January 1976.

[Ber68] E.R. Berlekamp, *Algebraic Coding Theory*, New York: McGraw-Hill, 1968.

[Ber79] E.R. Berlekamp, *Galois Field Computer*, US Patent No. 4162480, July 1979.

[Ber82] E.R. Berlekamp, *Bit-Serial Reed-Solomon Encoders*, IEEE Trans. on Inform. Theory, Vol. IT-28, No. 6, pp.869-874, November 1982.

[Bet87] T. Beth, B.M. Cook, D. Gollmann, *Architectures for Exponentiation in GF*(2^m), Lecture Notes in Computer Science 263, Springer-Verlag, pp.277-301, 1987.

[Bla84:1] R.E. Blahut, *Theory and Practice of Error Control Codes*, Reading, MA: Addison-Wesley, 1984.

[Bla84:2] R.E. Blahut, *A Universal Reed-Solomon Decoder*, IBM J. Res. Develop., vol.28 no.2, pp.150-158, 1984.

[Bla85] R.e. Blahut, *Fast Algorithms for Digital Signal Processing*, Reading, MA: Addison-Wesley, 1985.

[Bra89] L. Brange, M. Torkelson, *A Flexible CAD Tool for Module Generation*, Internal Report LUTEDX/(TETE-7041)/1-58, Lund University, Lund, Sweden, 1989.

[Bre84] R.P. Brent, H.T. Kung, *Systolic VLSI Arrays for Polynomial GCD Computation*, IEEE Trans. Comput., Vol. C-33, No. 8, August 1984.

[Bur71] H.O. Burton, *Inversionless Decoding of Binary BCH Codes*, IEEE Trans. Inform. Theory, vol. IT-17, no.4, pp. 464-466, July 1971.

[Cat90] R. Cates, *Gallium arsenide finds a new niche*, IEEE Spectrum, pp. 25-28, April 1990.

[Dav72] G.I. Davida, *Inverse of Elements of a Galois Field*, Electron. Letters, Vol.8, No.21, pp.518-520, October 1972.

[Dic58] L.E. Dickson, *Linear Groups with an Exposition of the Galois Field Theory*, Leipzig 1901; reprinted New York: Dover, 1958.

[Dif76] W. Diffie, M.E. Hellman, New directions in Cryptogrphy, IEEE Trans. Inform. Theory, IT-22, pp. 644-654, 1976.

[Dod85] S.M. Dodunekov, V. Zinoviev, *On Fast Decoding of Reed-Solomon Codes over GF(2^m) Correcting $t \leq 4$ Errors*, Internal Report LiTH-ISY-I-0750, Linköping University, Sweden, 1985.

[Eri86] H. Eriksson, E.D. Mastrovito, *A CMOS Implementation of a Parallel Multiplier over GF(64)*, Proc. of 2nd Nordic Symposium on Computers & Communications, Linköping, Sweden, June 1986.

[Fen89] G.L. Feng, *A VLSI Architecture for Fast Inversion in GF*(2^m), IEEE Trans. on Comput., Vol. C-38, No. 10, pp.1383-1386, October 1989.

[Für89] M. Fürer, K. Mehlhorn, AT^2*-Optimal Galois Field Multiplier for VLSI*, IEEE Trans. on Comput., Vol. C-38, No. 9, pp.1333-1336, September 1989.

[Gal62] E. Galois, *Écrits et Mémoires Mathematiques d'Évariste Galois*, Paris, Gauthier-Villars, 1962.

[Gal97] E. Galois, *Oeuvres Mathématiques*, Gauthier-Villars, Paris, 1897.

[Gei88] W. Geiselmann, D. Gollmann, *Symmetry and Duality in Normal Basis Multiplication*, Lecture Notes in Computer Science 357, Springer-Verlag, pp.230-238, March 1989.

[Gel89] P.P. Gelsinger, P.A. Gargini, G.H. Parker, A.Y.C. Yu, *Microprocessors circa 2000*, IEEE Spectrum, pp. 43-47, October 1989.

[Gil67] A. Gill, *Linear Sequential Circuits*, New York: MacGraw-Hill, 1967.

[Gol67] S.W. Golomb, *Shift Register Sequences*, San Francisco: Holden-Day, 1967.

[Gom90] D. Gollman, *Algorithmenentwurf in der Kryptographie*, Habil., University of Karlsruhe, 1990 [Preprint].

[Gre74] D.H. Green, I.S. Taylor, *Irreducible Polynomials over Composite Galois Fields and Their Application in Coding Techniques*, Proc. IEE, Vol. 121, N0. 9, pp. 935-939, September 1974.

[Gu89] J. Gu, K.F. Smith, *A Structured Approach for VLSI Circuit Design*, Computer, Vol. 22, No. 11, pp. 9-22, November 1989.

[Hag87] J. Hagenauer, E. Lutz, *Forward Error Correction Coding for Fading Compensation in Mobile Satellite Channels*, IEEE Jour. Selec. Ar. Commun., vol. SAC-5, no.2, pp. 215-225, Feb. 1987.

[Hsu84] I. Hsu, I.S. Reed, T.K. Truong , K. Wang, C. Yeh, L.J. Deutsch, *The VLSI Implementation of a Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm*, IEEE Trans. on Comput., Vol. C-33, No. 10, pp. 906-911, October 1984.

[Hsu88] I.S. Hsu, T.k. Truong, L.J. Deutsch and I.S. Reed, *A Comparison of VLSI Architecture of Finite Field Multipliers Using Dual, Normal or Standard Bases*, IEEE Trans. Comput., Vol.37, No.6, pp.735-739, June 1988.

[Hub89] K. Huber, *Some Comments on Zech's Logarithms*, Proc. of Fourth Joint Swedish-Sovjet Intern. Workshop on Inform. Theory, Gotland, Sweden, 1989, pp. 65-69.

[Inf48] L. Infeld, *Whom the Gods Love*, New York, McGraw-Hill 1948.

[Isa90] M. Isaksson, *Block Codes with Expanded Signal-Sets for PSK Modulation*, PhD. Dissertation TRITA-TTT-9002, Royal Institute of Technology, Stockhom, February 1990.

[Ito89] T. Itoh, S. Tsujii, *Structure of Parallel Multipliers for a Class of Fields GF*(2^m), Inform. and Comput. 83, pp. 21-40, 1989.

[Joh90] M.F. Johannesson, *A VLSI Implementation of a Parallel Universal Galois Multiplier*, Master Thesis, LiTH-ISY-EX-1053, December 1990.

[Knu69] D.E. Knuth, *The Art of Computer Programming*, Vol.2, Reading, MA: Addison-Wesley 1969.

[Law71] B.A. Laws and C.K. Rushforth, *A Cellular-Array Multiplier for GF*(2^m), IEEE Trans. Comput., vol. C-20, pp. 1573-1578, Dec. 1971.

[Lid83] R. Lidl, H. Niederreiter, *Finite Fields*, Reading, MA: Addison-Wesley, 1983.

[Lin83] S. Lin, D.J. Costello Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice-Hall 1983.

[Lit82] J.H. v. Lint, *Introduction to Coding Theory*, New York: Springer, 1982.

[Liu82] K.Y. Liu, *Architecture for VLSI Design of Reed-Solomon Encoders*, IEEE Trans. Comput., vol. C-31, No.2, pp.170-175, February 1982.

[Liu84] K.Y. Liu, *Architecture for VLSI Design of Reed-Solomon Decoders*, IEEE Trans. Comput., vol. C-33, pp.178-189, 1984.

[Mac86] F.J. MacWilliams and N.J. Sloane, *The Theory of Error-Correcting Codes*, Amsterdam: North-Holland 1986.

[Mae83] J.L. Massey, J.K. Omura, *Computational Method and Apparatus for Finite Field Arithmetic*, European Patent No. 0080528, June 1983.

[Man89] D.M. Mandelbaum, On Iterative Arrays for the Euclidean Algorithm over Finite Fields, IEEE Trans. Comput., Vol. 38, No.10, pp. 1473-1478, 1989.

[Mas88:1] E.D. Mastrovito, *VLSI Designs for Multiplication over Finite Fields GF(2^m)*, Lecture Notes in Computer Science 357, Springer-Verlag, pp. 297-309, March 1989.

[Mas88:2] E.D. Mastrovito, *VLSI Designs for Computations over Finite Fields GF(2^m)*, Lic. Thesis No: 159, LiU-TEK-LIC-1988:32, Linköping University, December 1988.

[Mas89:1] E.D. Mastrovito, *Exponentiation over Finite Fields GF(2^m). A comparison of Polynomial and Normal Basis Designs with Respect to VLSI Implementation*, Proc. of Fourth Joint Swedish-Sovjet Intern. Workshop on Inform. Theory, Gotland, Sweden, 1989, pp. 260-266.

[Mas89:2] E.D. Mastrovito, *Architectures for VLSI Design of a Fast Bounded Minimum Distance Decoder for First Order Concatenated Codes*, Int. Report LiTH-ISY-I-1037, Linköping University, November 1989.

[Mas90] E.D. Mastrovito, *Universal Galois Multiplier*, Swedish Patent Application No. 9002124-7, submitted June 1990.

[McC79] J.H. McClellan, C. M. Rader, *Number Theory in Digital Signal Processing*, Englewood Cliffs: Prentice-Hall, 1979.

[McE87] R.J. McEliece, *Finite Fields for Computer Scientists and Engineers*, Boston: Kluwer 1987.

[Mey90] F. Meyer, *Normalbasismultiplikation in endlichen Körpern*, Diplomarbeit, University of Karlsruhe (TH), February 1990.

[Mic80] J. Mick, J. Brick, *Bit-Slice Microprocessor Design*, New York: McGraw-Hill, 1980.

[Mor89] M.Morii, M. Kasahara, D.L. Whiting, *Efficient Bit-Serial Multiplication and the Discrete-Time Wiener-Hopf Equation over Finite Fields*, IEEE Trans. on Inform. Theory, Vol. IT-35, No. 6, pp. 1177-1183, November 1989.

[Mul89] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, R.M. Wilson, *Optimal Normal Bases in* GF(p^n), Discrete Appl. Math. 22, pp. 149-161, 1988/89.

[Mur89] T. Murakami, K. Kamizawa, M. Kameyama, S. Nakagawa, *A DSP Architectural Design for Low Bit-Rate Motion Video Codec*, IEEE Trans. Circuits & Systems, Vol. 36, No. 10, pp. 1267-1274, October 1989.

[Odl84] A.M. Odlyzko, *Discrete Logarithms in Finite Fields and their Cryptographic Significance*, Adv. Cryptol., Proc. Eurocypt '84, Paris, France, pp. 224-314, April 1984.

[Ort87] G.A. Orton, M.P. Roy, P.A. Scott, L.e: Peppard, S.E. Tavares, *VLSI Implementation of Public-Key Encryption Algorithms*, Lecture Notes in Computer Science 263, Springer-Verlag, pp.302-310, 1987.

[Pee85] J.B.H Peek, Communications Aspects of the Compact Disk Digital Audio System, IEEE Commun. Magaz., Vol. 23, No. 2, pp. 7-15, February 1985.

[Pei86] D.Y. Pei, C.C. Wang, J.K. Omura, *Normal Basis of Finite Fields GF(2^m)*, IEEE Trans. Inform. Theory, vol. IT-32, No.2, pp.285-287, March 1986.

[Pet72] W.W. Peterson and E.J. Weldon, *Error-Correcting Codes*, Cambridge, MA: MIT Press, 1972.

[Pin89] A. Pincin, *A New Algorithm for Multiplication in Finite Fields*, IEEE Trans. on Comput., Vol. C-38, No. 7, pp.1045-1049, July 1989.

[Pio89] M. Piontas, *Algorithm for Squaring in GF(2^m) in Standard Basis*, Electr. Letters, Vol. 25, No. 18, pp. 1262-1263, August 1989.

[Pos90] E.C. Posner, L.L. Rauch, B.D. Madsen, Voyager Mission Telecommunication Firsts, IEEE Commun. Magaz., Vol. 28, No. 9, pp. 22-27, September 1990.

[Pre88] Edited by B. Preas, M. Lorenzetti, *Physical Design Automation of VLSI Systems*, Menlo Park, CA: Benjamin/Cummings, 1988

[Ree75] I.S. Reed and T.K. Truong, The Use of Finite Fields to Compute Convolutions, IEEE Trans. Inform. Theory, vol. IT-21, No.2, pp.208-213, March 1975.

[Sch89] E. Schulz, *Algebraische Decodierung über die halbe BCH-Grenze hinaus*, PhD. Dissertation, Fortschr.-Ber. VDI Reihe 10, No. 107, Düsseldorf: VDI-Verlag 1989.

[Sco86] P.A. Scott, S.E. Tavares and L.E. Peppard, *A Fast VLSI Multiplier for GF(2^m)*, IEEE Jour. Selec. Ar. Commun., vol. SAC-4, no.1, pp. 62-66, Jan. 1986.

[Sco88] P.A. Scott, S.J. Simmons, S.E. Tavares and L.E. Peppard, *Architectures for Exponentiation in GF(2^m)*, IEEE Journal Sel. Areas Commun., Vol.6, No.3, pp.578-586, April 1988.

[Ség90] G.E. Séguin, *Low Complexity Normal Bases for $F_{2^{mn}}$*, Discrete Appl. Math. 28, pp. 309-312, 1990.

[Sha85] H.M. Shao, T.K. Truong, L.J. Deutsch, J.H. Yen, I.S. Reed, *A VLSI design of a Pipeline Reed-Solomon Decoder*, IEEE Trans. Comput., vol. C-34, no. 5, pp. 393-403, May 1985.

[Sme87] B. Smeets, *Some Results on Linear Recurring Sequences*, PhD Dissertation LUTEDX/(TEDD-1007)/1-129 (1987), Lund University, Lund, March 1987.

[Sug79] E. Sugimoto, *A Short Note on New Indexing Polynomials of Finite Fields*, Inform. Control 41, pp. 243-246, 1979.

[Sun86] R. Sundblad, *Contributions to System Design in CMOS Technology*, PhD Dissertation No. 152, Linköping University, Linköping, Sweden, December 1986.

[Til88] H.C.A. van Tilborg, *An Introduction to Cryptology*, Boston: Kluwer Academic Publ., 1988.

[Wag88] Y. Wang, X. Zhu, *A Fast Algorithm for the Fourier Transform over Finite Fields and its VLSI Implementation*, IEEE Jour. Selec. Areas in Commun., Vol. 6, No. 3, pp. 572-577, April 1988.

[Wah90] L. Wanhammar, B. Sikström, *DSP Integrated Circuits*, Preprint [to be printed by Prentice-Hall].

[Wan85] C.C. Wang, T.K. Truong, H.M. Shao, L.J. Deutsch, J.K. Omura and I.S. Reed, *VLSI Architectures for Computing Multiplications and Inverses in GF(2^m)*, IEEE Trans. Comput., vol. C-34, pp. 709-717, Aug. 1985.

[Wan89] C.C. Wang, *An Algorithm to Design Finite Field Multipliers Using a Self-Dual Normal Basis*, IEEE Trans. on Comput., Vol. C-38, No. 10, pp.1457-1460, October 1989.

[Wan90] C.C. Wang, D. Pei, *A VLSI Design for Computing Exponentiations in GF(2^m) and Its Application to Generate Pseudorandom Number Sequences*, IEEE Trans. on Comput., Vol. C-39, No. 2, pp.258-262, February 1990.

[Wat62] E.J. Watson, *Primitive Polynomials (Mod 2)*, Math. of Computation, No. 16, pp. 368-369, 1962.

[Wes85] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Reading, MA: Addison-Wesley, 1985.

[Wu87] W.W. Wu, D. Haccoun, R. Peile, Y. Hirata, *Coding for Satellite Communication*, IEEE Jour. Selec. Ar. Commun., vol. SAC-5, no.4, pp. 724-748, May 1987.

[Yaj86] S. Yajima, *VLSI-Oriented Hardware Algorithm*, IFIP VLSI'85, E. Hörbst edt., Elsevier Science Pub. B.V., North-Holland, 1986.

[Yeh84] C.S. Yeh, I.S. Reed and T.K. Truong, *Systolic Multipliers for Finite Fields GF(2^m)*, IEEE Trans. Comput., vol. C-33, pp. 357-360, Apr. 1984.

[Zie68] N. Zierler, J. Brilluart, *On Primitive Trinomials (Mod 2)*, Inform. Contr. vol. 13, pp. 541-554, 1968.

[Zie69] N. Zierler, *Primitive Trinomials Whose Degree is a Mersenne Exponent*, Inform. Contr. vol. 15, pp. 67-69, 1969.