Preface

This thesis describes the research which I conducted during my three year employment at the Institute for Experimental Mathematics. I hope that the work presented here can help serve as an example of one the institute's major goals, which is the enhancement of interdisciplinary work between engineers and mathematicians. Members of the Algebra and Number Theory Group, and the stimulating character of the Institute, contributed a great deal to the successful completion of the thesis. I wish to thank the following people from the institute, and from outside, for their support.

First of all, I am indebted to my advisor, Prof. Han Vinck, for his support and advice through all stages of the project, and the outstanding work atmosphere he has created in our group.

I am grateful to Prof. Richard Blahut from the University of Illinois and Prof. Henk van Tilborg from the Eindhoven University of Technology for their support as members of my Ph.D. committee.

From the two math groups I would particularly like to thank Dr. Wolfgang Happle, Prof. Reinhard Knörr, and Dr. Hans-Georg Rück for their many hints and ideas.

I would also like to thank the members of our group: Volker Braun, Olaf Hooijen, Dr. Tamás Horváth, Karin Rufaut, Dr. van Trung Tran, Heiner Schwarte, and Adriaan van Wijngaarden for their general support and for being such pleasant coworkers. In particular, thanks to Olaf for the fruitful work we did jointly.

Thanks also to Niko Lange from the Institute for Applied Microelectronics, Braunschweig, for his help in evaluation issues regarding chip implementation.

My thanks go also to Prof. Wayne Burleson and Jeong Yongijn from the University of Massachusetts at Amherst, for their hospitality and the insight in modern VLSI design methods.

There are also several students I would like to thank. Paul Even, Leon Grutters, Bart Jansen, Roel Pouls, Tonny Teunesen and Volker Wittelsberger posed numerous interesting technical questions and helped me stay in touch with implementational issues.

Last, but not least, I wish to thank Sarah Fowler for correcting my English, a task which certainly belonged to the less enjoyable ones needed for the completion of this thesis.

Essen, June 1994

Abstract

This thesis describes various efficient architectures for computation in Galois fields of the type $GF(2^k)$. "Efficient" refers to the fact that the architectures require a small number of elementary gates that are logical AND and exclusive OR. It is expected that, as a consequence, VLSI implementations of the architectures lead to chip designs which consume less area. All architectures are bit parallel, i.e. they apply only combinatorial logic and do not contain registers. This results in naturally fast architectures. The work focuses on the basic operations: multiplication, constant multiplication, and inversion. The architectures are based on algorithms which make extensive use of the decomposition of fields $GF(2^k)$ into $GF((2^n)^m)$, the latter of which will be called *composite fields*.

Two efficient algorithms which are related to composite fields are developed. One algorithm finds matrices which map binary field representation to composite field representations. The second algorithms performs a fast test to determine whether a polynomial over $GF(2^n)$ is primitive.

First, previous bit parallel architectures over fields $GF(2^n)$ and composite fields are reviewed. We comment on some of the previous architectures. A suboptimum algorithm for constant multiplication with a reduced number of gates is introduced. A complete list of optimized complexities for constant multiplication in the fields $GF(2^k)$, $k \leq 8$ is given in the appendix.

A general architecture for multiplication in composite fields is developed based on the Karatsuba-Ofman-Algorithm. The algorithm is closely investigated with respect to a parallel hardware implementation. It is shown that multiplication of two polynomials of degree less than m over $GF(2^n)$ is of order $\mathcal{O}((nm)^{\log_2 3})$. Through an exhaustive search, primitive polynomials are determined which perform modulo reduction with low complexity. We are able to give detailed descriptions of efficient parallel multipliers for field orders $\leq 2^{32}$.

It is shown that for certain field orders, a combined optimization of the polynomial multiplication and modulo reduction further improves the gate count and the delay of multiplier architectures. We provide suitable field polynomials and detailed descriptions of corresponding multipliers. The gate counts achieved for some field orders are the lowest ones reported in technical literature.

A comparative synthesis maps several parallel multiplier architectures to the gate-array library of TC 160G family. It is found that the comparatively low gate count of the architectures over composite fields can be transformed to netlists of gate-arrays. We conclude that the theoretical gate count is a valid measure for the number of gate equivalences of VLSI implementations if gate arrays are used. A speed estimation of the composite field multipliers results in a data throughput of up to 3.88 Gbit/sec.

An algorithm from Itoh and Tsujii for inversion over composite fields is applied to elements in standard base representation. A relationship between this algorithm and an architecture over tower fields proposed by Morii and Kasahara is developed. For the fields $GF(2^8)$ and $GF(2^{16})$ are, as an instance, architectures for parallel inverters with moderate gate count provided.

A new concept for systems involving finite field arithmetic is introduced. We propose a combined software/hardware approach which possesses the advantage of alterability. As an application, a composite field multiplier over $GF(2^{16})$ is attached to a 16 bit DSP. The external arithmetic enables the processor to perform general multiplication more than a magnitude faster than in software. We implemented a shortened (10,8) Reed-Solomon code which allows decoding at a speed of up to 1.9 Mbit/sec.

Contents

Chapter 1

Introduction

1.1 Motivation

The mathematical discipline, Algebra, includes the theory of finite fields. Its development dates back in the early nineteenth century, when Carl Friedrich Gauß and Evariste Galois worked on the general theory of finite fields. Previous work was done by Pierre de Fermat, Leonhard Euler, Joseph-Louis Lagrange and Adrien-Marie Legendre. In honor of Evariste Galois' fundamental work on the topic, finite fields are also referred to as *Galois fields*. The two names will be used interchangeably. Galois fields with q elements are denoted as $GF(q)$.

Over the last thirty years, Galois fields have gained wide spread technical applications. Areas where they have applications are:

- Algebraic codes [Bla83] [ML85]
- Cryptographic schemes [vT88] [Sch93]
- Digital signal processing [Bla85] [McC79]
- Random number generators [WP90]
- VLSI testing [GSB91]

The first two topics play an important role in modern digital communication. Since there is an increasing number of applications of communication systems expected in the near future — with increasing impacts on various aspects of our society — we will briefly explain the principals of these topics.

Transmission of digital data and its storage is often accompanied by the possibility of corruption of data. The principal of *channel coding* is that redundancy is introduced to the data before transmission or storage. Because of the extra information added, channel codes are principally capable of determining whether and where errors have occurred. In particular, BCH codes[1] and their subclass of Reed-Solomon codes (RS codes) have

[1] BCH codes are named after their inventors, Bose, Chaudhuri, and Hocquenghem.

proved to be extremely useful in technical communication systems. These codes require arithmetic in Galois fields. Most often, the use of fields with characteristic two allows a direct representation of binary data as field elements. So far, RS codes tend to perform arithmetic in fields $GF(2^8)$, while application of fields up to $GF(2^{32})$ seems promising for today's applications, since computer busses wider than eight bit have become important. More about RS codes will be said in Chapter 9.

Technical communication systems are endangered by the possibility of unauthorized reading and falsification of digital data. For the increasing number of applications of digital communication in areas such as electronic banking, *security* aspects will become a crucial issue. Several cryptographic schemes are based on the assumed difficulty of the discrete logarithm problem in finite groups or finite fields. Examples of such schemes, which have been applied widely, are the Diffie-Hellman key-exchange protocol [DH76] and the ElGamal scheme [ElG85]. A good overview on schemes applying finite field arithmetic can be found in [Odl84]. The latest recommendations for such systems suggest arithmetic in fields of order 2^{500}–2^{1000} so that security is assured. These field sizes refer to 500–1000 bit arithmetic modules. However, it should be kept in mind that it is difficult to predict the security of cryptographic systems for the future, as can be seen in the recent attack on the RSA scheme [Kol94].

Most architectures to be developed in this thesis will provide architectures with worked-out examples for fields up to an order of 2^{32}. However, the theory provided allows generalization to higher field orders, such as those needed for many cryptographic applications.

1.2 Finite Field Arithmetic in Hardware

It is often required that systems involving finite field arithmetic are *fast*. An example is channel coding in high speed data transmission. In order to meet this requirement, it might be necessary to implement the modules providing Galois field arithmetic on a semiconductor chip. Nowadays, hardware implementation usually implies a realization as a VLSI (Very Large Scale Integration) chip. VLSI modules performing Galois field arithmetic can roughly be classified into bit parallel and bit serial architectures. The former one applies only combinatorial logic, the latter one also applies registers. Generally speaking, there exist a time-space trade-off between the two types. While bit parallel architectures tend to be faster, bit serial architectures generally require less area than their parallel counterparts which provide the same function. All architectures treated in this thesis are bit parallel. For convenience, the terms "bit parallel" and "parallel" will be used interchangeably.

There are several aspects to be considered if VLSI architectures are to be evaluated. The most important ones are:

- Space complexity (chip area requirement)
- Time complexity (circuit delay or performance)

- Hierarchy
- Regularity
- Modularity

The first two aspects are unique measures for architectures which have been implemented. The architectures investigated in this thesis will be measured using *theoretical* space and time complexities. The theoretical space complexity is measured by the number of two input modulo 2 adders (logical exclusive OR, XOR,) and the number of two input modulo 2 multipliers (logical AND.) The theoretical time complexities are defined as the number of gate delays which are contained in the critical path. The architectures to be developed in this thesis focus on a low gate count, although the time complexities are also considered for most architectures. Chapter 7 deals with the relationship between theoretical gate count and area requirements of actual implementations.

The latter three aspects in the list above are structural properties [WE92]. Hierarchy is understood as the repeated division of a module into submodules. This eventually results in submodules with a comprehensible complexity. Regularity refers to architectures which are composed of *similar* modules or submodules. An example of regular structures are array architectures. Modularity is a property of architectures whose submodules possess *well defined* functions and interfaces. The architectures over composite fields $GF((2^n)^m)$, to be developed in this thesis, possess most of the structural properties. Subsection 5.5.2 discusses how the use of subfields $GF(2^n)$ results in naturally structured architectures.

1.3 Thesis Outline

Chapter 2 provides the mathematics of finite fields which is relevant to this thesis. A class of extension fields, referred to as composite fields, which is crucial for most architectures developed in the subsequent chapters, is introduced. Two algorithms related to composite fields are developed. The first algorithm finds linear mappings between different field representations. The second one determines whether a polynomial over $GF(2^n)$ is primitive. In particular, these polynomials can be used to generate composite fields.

Chapter 3 gives an overview of previous bit parallel architectures. The three classical types of multipliers, those applying standard, dual, and normal base representation of field elements, are introduced. Expressions for their space complexities are derived. Some comments on Mastrovito's standard base multiplier and on Berlekamp's dual base multiplier are given. Next, several parallel multipliers and inverters which operate over extension fields of $GF(2^n)$ are introduced.

In Chapter 4, constant multiplication over $GF(2^n)$ with reduced complexity is discussed. A locally optimum algorithm is introduced. We compare the optimized complexities with the complexities of a straightforward approach for fields up to $GF(2^{16})$. In the appendix, complete tables with optimized complexities for constant multiplication in fields up to $GF(2^8)$ are provided. The tables can be directly used for the determination

of the gate count of RS encoders. Upper bounds for the space complexity of constant multiplication in composite fields are also developed.

In Chapter 5, a general method for efficient bit parallel multiplication in composite fields is developed. The method applies the Karatsuba-Ofman algorithm, which is discussed in detail. Through an exhaustive search, field polynomials which allow modulo reduction with low complexity are found. Detailed descriptions for multiplier architectures in composite fields up to $GF(2^{32})$ are provided.

Chapter 6 derives parallel multiplier architectures for two classes of composite fields, namely $GF((2^n)^2)$ and $GF((2^n)^4)$. The two types of architectures are special cases of the previously described general method. By applying a combined optimization of polynomial multiplication and modulo reduction, the space and time complexities can be further reduced.

In Chapter 7, the VLSI syntheses of various multipliers are compared with respect to space and time complexities. The architectures compared are standard, dual, and normal base multipliers over $GF(2^k)$, and the composite field multipliers over $GF((2^n)^m)$. It is found that the latter one performs best for a gate array implementation with respect to area requirement. An estimation of the data throughput of an arithmetic module containing a composite field multiplier results in a maximum of 3.88 Gbit/sec.

Chapter 6 applies an efficient algorithm from Itoh and Tsujii for computing the inverse over composite fields to fields represented in standard base. Expressions for the space complexity are derived. A relationship to an architecture over tower fields, i.e. multiple field extensions of degree two, is developed. As examples, inverters over the fields $GF(2^8)$ and $GF(2^{16})$ are described and their space complexities are determined. It is found that implementation of parallel inverters for these fields are possible in terms of gate count.

In Chapter 9, a new concept for technical systems involving Galois field arithmetic is introduced. We propose a combined software/hardware approach. A 16 bit Reed-Solomon decoder is implemented on a digital signal processor which accesses an external multiplier over $GF(2^{16})$. Using a shortened Reed-Solomon code with code parameters (10,8) and a direct decoding algorithm, a decoding speed of up to 1.9 Mbit/sec becomes possible.

Chapter 2

Mathematical Background and Two Algorithms

2.1 Finite Fields

This section introduces the basic definitions and properties of finite fields which are relevant to the material treated later in this thesis. All statements are given without proof, but it will always be referred to the appropriate literature. Classically, books which cover algebraic coding also treat to some extend the mathematics of finite fields, as do, for instance, the references [Ber68] [PW72] [Bla83] or [LC83]. The number of mathematical books which are entirely devoted to finite fields is rather limited. Besides Lidl and Niederreiter's thorough mathematical treatment of the matter in [LN83], there are McEliece's book [McE87] and, more recently, the references [BGM+93] and [Jun93].

2.1.1 Basic Properties

We start with the definition of a fundamental algebraic structure which is called *group*. Its basic property is that it assigns to a pair of elements of a set a third element of the same set by applying one operation, denoted as $\circ$.

Definition 1 *A set G together with a binary operation $G \times G \to G$ is called a* group *if the following conditions are satisfied:*

- *The binary operation is* associative*: $(a \circ b) \circ c = a \circ (b \circ c)$, for all $a, b, c \in G$.*
- *There is an* identity *element $e \in G$ such that $a \circ e = e \circ a = a$, for all $a \in G$.*
- *For any element $a \in G$, there exists an* inverse *element $a' \in G$ such that $a \circ a' = a' \circ a = e$.*

If a group satisfies additionally the condition that $a \circ b = b \circ a$, for all $a, b \in G$, the group is said to be *commutative* or *abelian*.

Now we are in the position to define the algebraic structure *field*.

Definition 2 *[LC83] Let F be a set of elements on which two binary operations, called addition "+" and multiplication "$\cdot$", are defined. The set F together with the two binary operations $+$ and $\cdot$ is a field if the following conditions are satisfied:*

- *F is a commutative group under addition $+$. The identity element with respect to addition is called the* zero *element or the additive identity of F and is denoted by 0.*
- *The set of nonzero elements in F is a commutative group under multiplication $\cdot$. The identity element with respect to multiplication is called the* unit *element or the multiplicative identity of F and is denoted by 1.*
- *Multiplication is* distributive *over addition; that is, for any three elements a, b, and c in F: $a \cdot (b + c) = a \cdot b + a \cdot c$.*

There are fields with a finite number of elements which will be called finite or Galois fields. Such fields with q elements will be denoted by $GF(q)$. In the remainder of the thesis, only finite fields will be considered.

Definition 3 *The* order *of a field is the number of its elements.*

Theorem 1 *[McE87] The order q of a field must be a power of a prime: $q = p^m$, p prime.*

Theorem 2 *[McE87] There exists a unique field of order p^m, for any prime p and any positive integer m.*

Definition 4 *The smallest positive integer λ for which $\sum_{i=1}^{\lambda} 1 = 0$ in a field, is called the field's* characteristic.

All architectures in this thesis are based on Galois fields of characteristic two. An interesting consequence, which follows directly from the characteristic two property, is that every element a is its own additive inverse which leads to: $b - a = b + a$.

Definition 5 *Let a be an element of $GF(q)$. The smallest positive integer s for which $a^s = 1$ is called the* order *of the element.*

Definition 6 *Elements which have (maximum) order $s = q - 1$ are called* primitive elements[1].

It can be shown that elements with maximum order exist for every finite field. Primitive elements α and their powers generate the entire multiplicative group $\{1, \alpha^2, \alpha^3, \ldots, \alpha^{q-2}\}$ of a field. This power representation will be often used in this thesis in order to refer to field elements.

Theorem 3 *[LC83] Let a be a nonzero element of a finite field $GF(q)$. Then $a^{q-1} = 1$.*

Many finite field architectures for inversion are based on this theorem, since it follows immediately that $a\,a^{q-2} = 1$ and thus $a^{-1} = a^{q-2}$.

[1] There is some confusion in the literature about the terminology for these elements. Some books refer to them as "primitive elements", whereas they are some times referred to as "primitive roots."

2.1.2 Polynomials and Extension Fields

This subsection describes some properties of polynomials over finite fields. The important principle of extensions of finite fields will also be introduced. A special type of extension fields, named *composite fields*, will be defined.

A polynomial $A(x) = a_m x^m + a_{m-1} x^{m-1} + \cdots + a_0$ whose coefficients a_i are elements of a field $GF(q)$, is said to be a "polynomial over $GF(q)$." A polynomial is *monic* if its highest coefficient a_m is one.

Definition 7 *[LC83] A polynomial $A(x)$ is irreducible over $GF(q)$ if $A(x)$ is only divisible by c or by $c\,A(x)$ where $c \in GF(q)$.*

In the sequel, "$a \mid b$ " denotes "a divides b," where a and b can either be numbers or polynomials.

Definition 8 *Let $P(x)$ be a polynomial of degree m over $GF(q)$ with $P(0) \neq 0$. The smallest positive integer s for which $P(x) \mid (x^s - 1)$ is called the* order *of $P(x)$.*

Theorem 4 *[LN83] The order s of every irreducible polynomial of degree m over $GF(q)$ fulfills the condition: $s \mid (q^m - 1)$.*

A consequence of the last theorem is that the maximum possible order of an irreducible polynomial is $s = (q^m - 1)$.

Definition 9 *A monic polynomial of degree m with maximum order $s = (q^m - 1)$ is said to be a* primitive polynomial.

It can be shown that primitive polynomials of degree m over $GF(q)$ exist for any field $GF(q)$. Maximum order polynomials are of major importance for the remainder of this thesis.

An irreducible polynomial $P(x)$ of degree m over $GF(q)$ can be used to construct an *extension field* of $GF(q)$. The extension field is of order q^m and is denoted by $GF(q^m)$. The field $GF(q)$ is then a *subfield* of $GF(q^m)$ [McE87]. All q^m elements of the extension field can be represented as polynomials with a maximum degree of $m - 1$ over $GF(q)$. These q^m polynomials are the residue classes modulo $P(x)$ of all polynomials over $GF(q)$. Hence the polynomial $P(x)$ determines the algorithms for the arithmetic operations in the field.

Theorem 5 *[LN83] If a is an element of the finite field $GF(q^m)$, the element*

$$a^{\frac{q^m-1}{q-1}}$$

is in the subfield $GF(q)$.

Definition 10 *The* trace *Tr(a) of an element $a \in GF(q^m)$ relative to the subfield $GF(q)$ is defined by:*

$$Tr(a) = a + a^q + a^{q^2} + \cdots + a^{q^{m-1}}$$

It can be shown that $\mathrm{Tr}(a) \in GF(q)$.

In the following, a term introduced by Green and Taylor [GT74] will be adopted for denoting a certain type of extension fields of characteristic two:

Definition 11 *We call two pairs $\{GF(2^n), Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i\}$ and $\{GF((2^n)^m), P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i\}$ a* composite field *if*

- *$GF(2^n)$ is constructed from $GF(2)$ by $Q(y)$,*
- *$GF((2^n)^m)$ is constructed from $GF(2^n)$ by $P(x)$.*

Composite fields will be denoted by $GF((2^n)^m)$.

A composite field $GF((2^n)^m)$ is *isomorphic* to the field $GF(2^k)$, $k = nm$, in a mathematical sense [LN83]. However, although two fields of order 2^{nm} are isomorphic, their algorithmic complexity with respect to the field operations addition and multiplication may differ and depends on the choice of n and m and on the polynomials $Q(y)$ and $P(x)$. The introduction of composite fields for arithmetic in fields of order 2^{nm} will be crucial for the architectures to be developed in this thesis.

In the sequel, a root of $Q(y)$ will be denoted as ω, a root of $P(x)$ will be denoted as α. Assuming that both polynomials are primitive, the elements of the ground field $GF(2^n)$ can be represented by $\{0, 1, \omega, \omega^2, \ldots, \omega^{2^n-2}\}$, and the elements of the composite field can be represented by $\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^{nm}-2}\}$

2.1.3 Bases of Finite Fields

Although, in principle, there exist many different bases for representing elements of a Galois field, there are three bases which are of major importance from a technical point of view. This subsection provides the formal definition of the three bases, which are standard, normal, and dual base. Their application to arithmetic architectures will be treated in Section 3.1.

An extension $GF(q^m)$ of the the field $GF(q)$ can be viewed as as m-dimensional vector space over $GF(q)$. Each element of $GF(q^m)$ can be represented as a linear combination of the m elements of the base $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$. The coefficients of the linear combination are elements of the field $GF(q)$.

Definition 12 *The set*

$$\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\},$$

where α is a root of the irreducible polynomial $P(x)$ of degree m over $GF(q)$, is called standard *(or* canonical *or* polynomial*)* base.

This base is directly related to the representation of field elements as polynomials, as was stated in the previous subsection. In this case, a field element A is represented by the polynomial $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1}$, and each element represents a residue class modulo $P(x)$. Since α is a root of $P(x)$, the polynomial representation $A(x)$ is equivalent to $A(\alpha) = a_0 + a_1 \alpha + a_2 \alpha^2 + \cdots + a_{m-1} \alpha^{m-1}$.

Definition 13 *The set*

$$\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{m-1}}\},$$

where α is root of the irreducible polynomial $P(x)$ of degree m over $GF(q)$, is called normal base *if the m elements are linearly independent.*

It can be shown that normal bases exist for all Galois fields. The normal base representation is especially attractive for certain applications which involve exponentiation in finite fields, because raising to the qth power is simply a cyclic shift.

Definition 14 *Let $B = \{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ be a base of $GF(q^m)$. The* dual base *$\{\gamma_0, \gamma_1, \ldots, \gamma_{m-1}\}$ of B is a base satisfying:*

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

It can be shown that there exists a dual base for every base.

2.2 Mapping between Binary and Composite Field Representations

This section describes an algorithm which determines a binary matrix that defines the isomorphic mapping between the field representations $GF(2^k)$ and $GF((2^n)^m)$ with $k = nm$. The algorithm was developed in cooperation with the Number Theory and Algebra Group at the Institute for Experimental Mathematics. The mapping might be important in an application of composite fields, where composite field arithmetic modules have an interface to modules that operate with a binary standard base representation. In this case, simply a linear mapping at the input and output of the composite field module has to be performed. A possible scenario is, for instance, a Reed-Solomon decoder chip based on composite field arithmetic which decodes symbols generated by an encoder that uses a binary field polynomial for its arithmetic. In the following it is understood that there exists only one field of order 2^k, and that the term "different" refers to different representations of elements rather than to different fields. The mapping to be developed assumes a standard base representation of both field elements.

Our goal is the determination of a binary matrix $\mathbf{T}$ of size $(k \times k)$ which performs an *isomorphic* mapping of field elements represented with respect to $GF(2^k)$ to elements

represented with respect to $GF((2^n)^m)$. The inverse of $\mathbf{T}$, denoted by $\mathbf{T}^{-1}$, will perform the mapping in the other direction. However, the algorithm can be applied in a straightforward manner for the determination of other field mappings as well, e.g. between two isomorphic fields given by different binary field polynomials. In the sequel, we will assume that all field polynomials are primitive, i.e. they have maximum order.

First, we will provide some notations. Arithmetic in $GF((2^n)^m)$ is performed modulo the two field generators $Q(y)$ and $P(x)$. $Q(y)$ is a binary polynomial which generates the subfield $GF(2^n)$:

$$Q(y) = y^n + q_{n-1}y^{n-1} + \cdots + q_1 y + 1, \quad q_i \in GF(2).$$

$P(x)$ is a polynomial over $GF(2^n)$, which generates the composite field representation $GF((2^n)^m)$:

$$P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1 x + p_0, \quad p_i \in GF(2^n).$$

A primitive element in $GF((2^n)^m)$ which is a root of $P(x)$ will be denoted α: $P(\alpha) = 0$. A primitive element in $GF(2^n)$ will be denoted as ω, where $Q(\omega) = 0$. Every element A is represented as a vector with m components from $GF(2^n)$, while every vector element is itself a binary n vector:

$$\begin{aligned} A &= (a_{m-1}, a_{m-2}, \ldots, a_0), \quad a_i \in GF(2^n) \\ &= ((a_{m-1,n-1}, a_{m-1,n-2}, \ldots, a_{m-1,0}), (a_{m-2,n-1}, a_{m-2,n-2}, \ldots, a_{m-2,0}), \ldots, \\ &\quad (a_{0,n-1}, a_{0,n-2}, \ldots, a_{0,0})), \quad a_{ij} \in GF(2). \end{aligned} \tag{2.1}$$

Equation (2.1) shows that A is also represented by a binary $nm = k$ vector. In particular, this is how all elements from $GF((2^n)^m)$ are represented in actual digital systems, such as VLSI chips.

Arithmetic in $GF(2^k)$ is performed modulo the binary field polynomial $R(z)$ of the following form:

$$R(z) = z^k + r_{k-1}z^{k-1} + \cdots + r_1 z + 1, \quad r_i \in GF(2).$$

Let β be a root of $R(z)$ and $B_2 = (\beta^{k-1}, \beta^{k-2}, \ldots, \beta, 1)$ is the standard base with which the elements of $GF(2^k)$ are represented. Each element of $GF(2^k)$ is thus represented as a binary k vector, denoting a linear combination of the base elements.

In order to construct the isomorphic mapping, we are looking for k base elements represented with respect to $GF((2^n)^m)$, to which the k base elements from B_2 are to be mapped. Clearly, the "one" element is mapped to the "one" element. The primitive base element β must be mapped to a primitive element α^t, the base element β^2 must be mapped to α^{2t}, and so on:

$$\mathbf{T}\beta^i = \alpha^{it}, \quad i = 0, 1, \ldots, k-1$$

We are now left with the determination of the exponent t. The mapping between the two field representation must be homomorphic with respect to both field operations,

addition and multiplication. In order to assure that the mapping is homomorphic with respect to multiplication, it is not sufficient to map β to just any primitive element α^t. The condition is rather that

$$R(\alpha^t) = 0 \pmod{Q(y), P(x)}. \tag{2.2}$$

There will be exactly k primitive elements which fulfill this condition, namely α^t and its $k-1$ conjugates α^{t2^j}, $j = 1, 2, \ldots, k-1$. The exponents $t2^j$ are computed modulo $2^k - 1$.

In the following the algorithm will be stated.

1. (Initialization) Let α be the primitive element in $GF((2^n)^m)$ for which $P(\alpha) = 0$. Set $t := 1$. Prepare a list with $2^k - 1$ addresses and memory for one binary entry per address. Enter the vector $(0, 0, \ldots, 0, 1)$ into the rightmost column of $\mathbf{T}$. This provides a mapping of the one element to the one element.

2. Compute $R(\alpha^t) \pmod{Q(y), P(x)}$. If the result is zero, the element is found; goto Step 7.

3. Neither α^t nor the conjugates α^{t2^j}, $j = 1, 2, \ldots, k-1$ are the elements to which β is mapped. Therefore enter zeros in the list at addresses $t2^j \pmod{2^k - 1}$, $j = 0, 1, 2, \ldots, k-1$.

4. Set $t := t + 1$. If the list already has a zero entry at address t, goto Step 4 (i.e. repeat this step until an address t is found which does not have an entry.)

5. Check if α^t is primitive element by computing $\mathrm{GCD}(t, 2^k - 1)$. If it is not primitive, goto Step 4.

6. Goto Step 2.

7. Enter the binary vector representation (2.1) of α^t into the second rightmost column of $\mathbf{T}$. Into the next column on the right hand side, the binary vector representation of α^{2t} is entered, into the next α^{3t}, and so on until $\alpha^{(k-1)t}$ is entered into the leftmost column.

Before we comment on the performance of the algorithm, an example is given.

Example. We consider the two field representation $GF(2^8)$ and $GF((2^4)^2)$. The field polynomial of $GF(2^8)$ is $R(z) = z^8 + z^4 + z^3 + z^2 + 1$. We denote the root of R with β: $R(\beta) = 0$. The representation $GF((2^4)^2)$ is generated by $Q(y) = y^4 + y + 1$ and $P(x) = x^2 + x + \omega^{14}$, where $Q(\omega) = 0$ and $P(\alpha) = 0$.

We start by computing $R(\alpha)$:

$$R(\alpha) = \alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1 = \omega^{14}\alpha + \omega^{12} \neq 0$$

It should be stressed that all arithmetic is performed according to the rules of the composite field representation. We see that neither α nor its conjugates α^j,

$j = 2, 4, 8, 16, 32, 64, 128$ are elements to which β is mapped. Appropriate entries in the list are made.

The next element to be checked is α^7 (we know from the list that α^t, $t = 2, 4$ do not have to be tested, and the elements $t = 3, 5, 6$ are not primitive.) One obtains $R(\alpha^7) = \omega^{11} \neq 0$. Again, we make entries in the list at addresses $7, 14, 28, 56, 112, 131, 193, 224$.

The first element which fulfills condition (2.2) is α^{37}, i.e. $R(\alpha^{37}) = 0$. Now the base element β is mapped to α^{37}, base element β^2 to α^{74}, base element β^3 to α^{111}, and so on. The last pair to be mapped is base element β^7 to α^4. We compute the binary representation of the α^i with respect to the field $GF((2^4)^2)$. For instance, for the element α^{37} we obtain:

$$\alpha^{37} = \omega\alpha + \omega^{12} = (0010\,1111).$$

All 7 binary representation are entered into the matrix from right to left, which yields the transformation matrix.

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Every element E represented with respect to $GF(2^8)$ can now be mapped to a representation with respect $GF((2^4)^2)$ through $E' = \mathbf{T}E$. The mapping is homomorphic to all field operations. The inverse mapping, i.e. from a $GF((2^4)^2)$ representation to a $GF(2^8)$ representation, is performed through $\mathbf{T}$'s inverse.

The algorithm has the structure of an exhaustive search through all $2^k - 1$ elements of the multiplicative group of $GF(2^k)$. However, by applying a list with the conjugates of the elements already checked, the computation is reduced by the factor k. The most costly operation in every step is the evaluation $R(\alpha^t)$. If only these evaluations are considered, the algorithm has a complexity of order

$$\mathcal{O}\left(\frac{\Phi(2^k - 1)}{k}\right) \leq \mathcal{O}\left(\frac{2^k - 1}{k}\right),$$

where $\Phi(\cdot)$ denotes the Euler function.

It should be noted that there are always k different transformation matrices with corresponding inverse matrices. The matrices are computed from the first element found and its $k - 1$ conjugates, respectively. In the example above, it is also possible to perform homomorphic mappings from β to α^t, $t = 41, 73, 74, 82, 146, 148, 164$. These are the 7 conjugates of α^{37}. In actual implementations it might be advantageous to choose the transformation matrix with the smallest number of entries.

2.3 An Efficient Test on Primitivity

The goal of this section, is the development of a fast algorithm which tells whether a monic polynomial of degree m over $GF(2^n)$ is primitive. In the sequel, we present an implementation developed together with the Number Theory group at the Institute for Experimental Mathematics[2]. Polynomials which pass the test can be used for constructing composite fields. The algorithm was first introduced by Alanen and Knuth in their 1964 paper [AK64]. The version described here also includes a considerable speed improvement suggested in Appendix 2 of the reference.

By definition, a polynomial $P(x)$ over $GF(2^n)$ of degree m is said to be primitive if it is irreducible over $GF(2^n)$ and if it has the maximum order, in our case $2^{nm}-1$. The order of $P(x)$ is defined as the smallest integer s such that $x^s - 1 \equiv 0 \bmod P(x)$. Next, two results from [AK64] are slightly modified in order to match the finite fields considered in this thesis.

Theorem 6 (Lemma 1 in [AK64]) *If a monic polynomial of degree m over $GF(2^n)$ has order $s = 2^{nm}-1$, it is primitive (no test for irreducibility necessary.)*

The following proof for the theorem is different from the original one:

> **Proof.** We consider the ring R of polynomials in x over $GF(2^n)$ consisting of the residue classes modulo $P(x)$. The ring has exactly 2^{nm} elements (residue classes.) On the other hand, the elements $\{1, x, x^2, \ldots, x^{s-1}\}$, $s := 2^{nm} - 1$, are all distinct modulo $P(x)$, therefore belonging to s different residue classes. Including the residue class containing the zero element, the elements $\{0, 1, x, x^2, \ldots, x^{s-1}\}$ are a complete set of representatives of all residue classes of R. Since $x^s \equiv 1 \bmod P(x)$, we have $x^k\, x^{s-k} \equiv 1 \bmod P(x)$ for all $0 \leq k < s$. This means that every element x^k has a multiplicative inverse mod$P(x)$. Hence R is also a field. Since there is only one such field, $P(x)$ is field generator and in particular irreducible. $\diamond$

Theorem 7 (Theorem in [AK64]) *The constant coefficient p_0 of a primitive polynomial of degree m over $GF(2^n)$ is a primitive root (i.e. an element with order $2^n - 1$) in the ground field $GF(2^n)$.*

> **Proof.** The m roots of every primitive polynomial $P(x)$ are a primitive root α from the extension field $GF((2^n)^m)$ and its $m-1$ (primitive) conjugates:
>
> $$P(x) = (x-\alpha)(x-\alpha^{2^n})(x-\alpha^{2^{2n}})\cdots(x-\alpha^{2^{(m-1)n}}).$$
>
> Hence, the constant coefficient can be written as:
>
> $$p_0 = \prod_{i=0}^{m-1} \alpha^{2^{i\,n}} = \alpha^e$$

[2]Special thanks to Dr. Wolfgang Happle for his support

with

$$e = \frac{2^{nm} - 1}{2^n - 1}.$$

For every element $\beta \in GF((2^n)^m)$, β^e is element of $GF(2^n)$ [LN83]. Moreover, if β has maximum order $2^{nm} - 1$ in the extension field, β^e has order $(2^{nm} - 1)/e = 2^n - 1$, which is maximum order in the ground field. $\diamond$

For the test procedure we take advantage of the fact that a polynomial which divides $x^s - 1$, s integer, has an order o with either $o = s$ or with $o \mid s$, $o < s$. We are now able to state a fairly efficient algorithm for testing a polynomial $P(x)$.

1. Compute $2^{nm} - 1$ and its r maximum divisors d_i,$i = 1, 2, \ldots, r$.
2. Check if either $x = 1$ is a root or if p_0 is not a primitive root in $GF(2^n)$ (trivial checks.) If so, the polynomial is not primitive.
3. Check if $x^{2^{nm}-1} \equiv 1 \bmod P(x)$. If not, the polynomial is not primitive. Otherwise $P(x)$ is a candidate.
4. Check if $x^{d_i} \equiv 1 \bmod P(x)$, $i = 1, 2, \ldots, r$. If this is fulfilled for any i, $P(x)$ has an order less than $2^{nm} - 1$ and is thus not primitive. On the contrary, if non of the d_i satisfies the condition, the polynomial is in fact primitive.

By using maximum divisors in Step 4 all possible orders less than $2^{nm} - 1$ are checked. In order to obtain the maximum divisors in Step 1, the factorization of $2^{nm} - 1$ must be known. Fortunately, the numbers $2^k \pm 1$ are well studied, such that the factorization even for values $k \approx 2^{10}$ can be calculated [Rie85]. It should be noted that the tests $x^s \equiv 1 \bmod P(x)$ are basically exponentiations modulo a polynomial. Since the values of s are rather large, the algorithm implemented uses the "binary method" [Knu81] of repeated multiplying and squaring which can be applied very elegantly to operations in fields of characteristic two. The complexity of a test is thus of $\mathcal{O}(\log(nm))$.

Example. We consider the field $GF((2^4)^4)$ with $Q(y) = y^4 + y + 1$. The factorization of the field order minus one is: $2^{16} - 1 = 257 \cdot 17 \cdot 5 \cdot 3$. The corresponding $r = 4$ maximum divisors are: $d_1 = 255, d_2 = 385, d_3 = 13107, d_4 = 21845$. The $\Phi(15) = 8$ primitive roots of the ground field are: $\{\omega, \omega^2, \omega^4, \omega^7, \omega^8, \omega^{11}, \omega^{13}, \omega^{14}\}$. We implemented an exhaustive search algorithm in C, accessing a self written C-library providing Galois field arithmetic. The search determined all primitive polynomials. Running the algorithm on an IBM RS6000/580, the search was performed in 57 sec. There were $2^{16} - 1 = 65535$ polynomials tested of which $\frac{1}{4}\Phi(2^{16} - 1) = 8192$ were found to be primitive.

Chapter 3

Previous Bit Parallel Architectures

3.1 Traditional Multipliers

In this section three different approaches for bit parallel multipliers are introduced. Since this chapter, as well as the entire thesis, is restricted to bit parallel architectures, the terms "bit parallel multiplier" and "multiplier" will be used interchangeably for convenience. The expression "traditional multiplier" heading this section is a somewhat informal name for a class of Galois field multipliers defined by the author. We understand it as a class of parallel architectures with the following properties:

1. The multipliers do *not* operate over extension fields of $GF(2^n)$.
2. The space complexity of the multipliers is lower bounded by a total of $2n^2 - 1$ gates (AND + XOR).

The vast majority of the parallel multipliers proposed in technical literature, starting with the early paper of Bartee and Schneider in 1963 [BS63], possesses both properties. Since the few publications about architectures using field extension tend to be recent, we hope that the name "traditional" is meaningful to the reader. However, there are many new publications, often with important results, that describe architectures which are "traditional" according to our classification; we certainly do not intend to consider these architectures to be old-fashion or inferior.

In the sequel we introduce three different approaches for traditional parallel multipliers. Each architecture uses a different base for the representation of its operands. The three bases used — standard (SB), dual (DB), and normal base (NB) — lead to quite different architectures. Whereas the two latter ones will be explained more generally, the SB multiplier proposed by Mastrovito [Mas89] [Mas91] will be studied thoroughly. We will also comment on it, extending previous knowledge. A modified version of the DB multiplier will be introduced as well.

Chapter 7 will show the results of a gate array synthesis of the three traditional multipliers compared to the architectures developed in this thesis.

3.1.1 Mastrovito's Standard Base Multiplier

Architecture and Complexity

In this section an architecture for the multiplication of field elements given in standard base, introduced by Mastrovito in [Mas89] and [Mas91], will be developed. There are several reasons for choosing this architecture as a representative for standard base multipliers. First, it has one of the lowest gate counts among the traditional SB multipliers. Secondly, and maybe even more important, it will be used as the ground field multiplier for the architectures over composite fields to be developed in the Chapters 5 and 6. It also serves as an example for a traditional multiplier with low complexity in Chapter 7, where several architectures are compared with respect to a gate array implementation.

First, we will introduce a matrix notation for the multiplication $A(y)B(y) = C(y) \bmod Q(y)$ in the field $GF(2^n)$. All elements are binary polynomials of degree less than n:

$$c_{n-1}y^{n-1} + \ldots + c_0 = (a_{n-1}y^{n-1} + \ldots + a_0)(b_{n-1}y^{n-1} + \ldots + b_0) \bmod Q(y).$$

Alternatively, the elements $B(y)$ and $C(y)$ can be represented as column vectors containing the polynomial coefficients. By introducing the matrix $\mathbf{Z} = f(A(y), Q(y))$ the multiplication can be described as:

$$C = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = \mathbf{Z}B = \begin{pmatrix} f_{0,0} & \cdots & f_{0,n-1} \\ \vdots & \ddots & \vdots \\ f_{n-1,0} & \cdots & f_{n-2,n-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}. \tag{3.1}$$

The matrix $\mathbf{Z}$ is named "product matrix." Its coefficients $f_{ij} \in GF(2)$ depend recursively on the coefficients a_i and on the coefficients q_{ij} of the $\mathbf{Q}$ matrix which is introduced below in (3.3) as follows:

$$f_{ij} = \begin{cases} a_i & ; \quad j = 0 & ; \quad i = 0, \ldots, n-1; \\ u(i-j)a_{i-j} + \sum_{t=0}^{j-1} q_{j-1-t,i} a_{n-1-t} & ; \quad j = 1, \ldots, n-1 & ; \quad i = 0, \ldots, n-1; \end{cases} \tag{3.2}$$

where the step function u is defined as

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

The matrix-vector product in Equation (3.1) describes the entire field multiplication. The $\mathbf{Q}$ matrix which is required to build $\mathbf{Z}$ is a function of the binary primitive polynomial $Q(y)$ of degree n, generating $GF(2^n)$. Its binary entries $q_{i,j}$ are defined such that:

$$\begin{pmatrix} y^n \\ y^{n+1} \\ \vdots \\ y^{2n-2} \end{pmatrix} \equiv \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,n-1} \\ q_{1,0} & q_{1,1} & \cdots & q_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n-2,0} & q_{n-2,1} & \cdots & q_{n-2,n-1} \end{pmatrix} \begin{pmatrix} 1 \\ y \\ \vdots \\ y^{n-1} \end{pmatrix} \bmod Q(y). \tag{3.3}$$

The **Q** matrix describes the representation of the polynomials $y^n, y^{n+1}, \ldots, y^{2n-2}$ in the equivalence classes $\bmod Q(y)$, i.e. after the reduction modulo $Q(y)$.

In the following, an example for the construction of the matrix **Q** and of the product matrix **Z** is given:

Example. Let $Q(y) = y^4 + y + 1$ be the primitive polynomial generating $GF(2^4)$. Considering the equivalence classes $\bmod Q(y)$, the polynomials y^4, y^5 and y^6 are represented by:

$$\begin{aligned} y^4 &\equiv 1 + y \bmod Q(y) \\ y^5 &\equiv y + y^2 \bmod Q(y) \\ y^6 &\equiv y^2 + y^3 \bmod Q(y). \end{aligned} \tag{3.4}$$

Rewriting (3.4) in matrix notation yields the **Q** matrix:

$$\begin{pmatrix} y^4 \\ y^5 \\ y^6 \end{pmatrix} \equiv \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ y \\ y^2 \\ y^3 \end{pmatrix} \bmod y^4 + y + 1.$$

The product matrix can now be constructed by applying (3.2):

$$C = \mathbf{Z}B = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 + a_3 & a_2 + a_3 & a_1 + a_2 \\ a_2 & a_1 & a_0 + a_3 & a_2 + a_3 \\ a_3 & a_2 & a_1 & a_0 + a_3 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}. \tag{3.5}$$

The implementational complexity of the matrix-vector product (3.2) depends solely on the primitive polynomial $Q(y)$. In [Mas89] generating primitive polynomials are given for fields $GF(2^n)$, $n = 2, 3, \ldots, 16$. The polynomials are optimum with respect to the number of gates required to multiply in the field. For fields in which primitive trinomials of the form

$$Q(y) = y^n + y + 1 \tag{3.6}$$

exist, the space complexity is given by:

$$\#\text{AND} \; + \; \#\text{XOR} = 2n^2 - 1. \tag{3.7}$$

Polynomials of the form (3.6) exist for $n = 2, 3, 4, 6, 7, 9, 10, 11, 15$. However, for the trinomial $Q(y) = y^5 + y^2 + 1$, the complexity (3.7) can also be realized. For other values of n where there are no primitive trinomial, the complexity is higher, as can be seen in Table 3.1.

The delay (or time complexity) of the multiplier is upper bounded by:

$$T = T_{\text{AND}} + T_{\text{XOR}} \leq 1 + 2\lceil \log_2 n \rceil, \tag{3.8}$$

measured in gate delays.

n	$Q(y)$	AND	XOR	$\mathcal{T}_{\text{and}}$	$\mathcal{T}_{\text{xor}}$
2	2,1,0	4	3	1	2
3	3,1,0	9	8	1	3
4	4,1,0	16	15	1	3
5	5,2,0	25	24	1	5
6	6,1,0	36	35	1	4
7	7,1,0	49	48	1	4
8	8,5,3,2,0	64	84	1	5
9	9,4,0	81	80	1	6
10	10,3,0	100	99	1	6
11	11,2,0	121	120	1	6
12	12,8,5,1,0	144	207	1	7
13	13,7,6,1,0	169	202	1	6
14	14,9,7,2,0	196	282	1	7
15	15,1,0	225	224	1	5
16	16,11,6,5,0	256	281	1	6

Table 3.1: Space and time complexity of the Mastrovito multiplier in the ground fields $GF(2^n)$

Some Comments on the Mastrovito Multiplier

Next, we will state some additional facts about the Mastrovito multiplier. First we will give a formula for computing the matrix $\mathbf{Q}$. The binary entries q_{ij} of $\mathbf{Q}$ in Equation (3.3) can be computed recursively after the first row is filled with the coefficients of $Q(y) = y^n + q_{n-1}y^{n-1} + \ldots + q_1 y + 1$, i.e. $q_{0,j} = q_j$, through:

$$q_{i,j} = \begin{cases} q_{i-1,n-1} & ; \quad i = 1, \ldots, n-2 \quad ; \quad j = 0; \\ q_{i-1,j-1} + q_{i-1,n-1}q_{0,j} & ; \quad i = 1, \ldots, n-2 \quad ; \quad j = 1, \ldots, n-1. \end{cases}$$

Since the matrix-vector operation in Equation (3.1) requires exactly n^2 mod 2 multiplications, the space complexity given through (3.7) can be further specified as:

$$\#\text{AND} = n^2, \tag{3.9}$$

$$\#\text{XOR} \geq n^2 - 1, \tag{3.10}$$

where Equation (3.10) is fulfilled with equality, if the field generator is of the type stated in Equation (3.6). The time complexity can be further specified into multiples of XOR and AND gate delays. The delays will be denoted as $\mathcal{T}_{\text{xor}}$ and $\mathcal{T}_{\text{and}}$, respectively. If it is taken into consideration that each path through the multiplier contains only one mod 2 multiplier, it follows directly that the overall delay can be upper bounded by:

$$T \leq \mathcal{T}_{\text{and}} + 2\mathcal{T}_{\text{xor}}\lceil \log_2 n \rceil. \tag{3.11}$$

Using the extensions from above, it becomes possible to further specify the Mastrovito multiplier. Table 3.1 is an improved version of Table 4.5 given in [Mas91]. It contains generating polynomials $Q(y)$ for the ground fields together with the space and time complexity of multipliers in these fields. Both complexities are, unlike those in [Mas91, Table 4.5], separated into mod 2 multipliers (AND) and mod 2 adders (XOR). The row headed by $Q(y)$ contains the positions of the non-zero coefficients of the primitive polynomials.

> **Example.** We consider the multiplier in the ground field $GF(2^4)$, i.e. $n = 4$. The field polynomial used is $Q(y) = y^4 + y + 1$. The multiplier can be implemented with 16 AND gates and 15 XOR gates. This is the complexity needed for computing the matrix vector product shown in Equation 3.5. The architecture has a time complexity of 1 AND gate delay and 3 XOR gate delays.

3.1.2 Dual Base Multipliers

Architecture and Complexity

This section presents a multiplier which uses the dual base representation of one operand. The algorithm on which the multiplier is based was first described by Berlekamp in [Ber82]. In the paper, which describes the implementation of a Reed-Solomon encoder, the algorithm is applied to the multiplication of a constant field element with a variable one.

First, we recall the definition of a dual base. Let $B_s = \{1, \omega, \omega^2, \ldots, \omega^{n-1}\}$ be a standard base for a field $GF(2^n)$. A base $B_d = \{\lambda_0, \lambda_1, \lambda_2, \ldots, \lambda_{n-1}\}$ is said to be a dual base to B_s iff:

$$\mathrm{Tr}(\omega^i \lambda_j) = \begin{cases} 1 & , \text{ if } i = j \\ 0 & , \text{ if } i \neq j. \end{cases} \tag{3.12}$$

For the multiplier to be developed we represent the first operand A in the usual standard base

$$A = a_0 + \omega a_1 + \omega^2 a_2 + \cdots + \omega^{n-1} a_{n-1},$$

and the second operand in the corresponding dual base

$$B = \lambda_0 b_0 + \lambda_1 b_1 + \lambda_2 b_2 + \cdots + \lambda_{n-1} b_{n-1}.$$

Next, a formula for the multiplication of B (given in DB) with a base element ω from the SB will be derived. Consider:

$$\mathrm{Tr}(\omega^j B) = \mathrm{Tr}(\omega^j \lambda_0 b_0 + \omega^j \lambda_1 b_1 + \omega^j \lambda_2 b_2 + \cdots + \omega^j \lambda_{n-1} b_{n-1}) = b_j \ ; \ j = 0, 1, \ldots, n-1, \tag{3.13}$$

where the definition of the DB (3.12) was used. If the j-th element of the product ωB is denoted $(\omega B)_j$, we get

$$(\omega B)_j = \mathrm{Tr}(\omega^j (\omega B)) = \mathrm{Tr}(\omega^{j+1} B) = \begin{cases} b_{j+1} & , \ j = 0, 1, \ldots, n-2, \\ \mathrm{Tr}(\omega^n B) & , \ j = n-1, \end{cases} \tag{3.14}$$

where Equation (3.13) was used twice, for the rightmost and the leftmost "=". Apparently all elements $(\omega B)_j$ except the highest one are obtained simply by a shift of the elements of B. The coefficient $(\omega B)_{n-1}$ can be obtained as follows. Let $Q(y) = 1 + q_1 y + \cdots + q_{n-1} y^{n-1} + y^n$ be the binary field polynomial such that $Q(\omega) = 0$. Then

$$\omega^n = 1 + q_1\omega + \cdots + q_{n-1}\omega^{n-1}, \tag{3.15}$$

which can be used for computing

$$\begin{aligned} (\omega B)_{n-1} &= \mathrm{Tr}(\omega^n B) = \mathrm{Tr}((q_0 + q_1\omega + \cdots + q_{n-1}\omega^{n-1})B), \\ &= b_0 + q_1 b_1 + \cdots + q_{n-1} b_{n-1} = Q \circ B. \end{aligned} \tag{3.16}$$

The last term in Equation (3.16) is the dot product of the element B and the coefficients of the field polynomial. A hardware implementation of this dot product has a complexity of

$$C_1 = (h_w(Q) - 2)\,\mathrm{XOR} \;\geq 1\,\mathrm{XOR}, \tag{3.17}$$

where $h_w(Q)$ denotes the weight of the field polynomial, i.e. the number of coefficients which are one.

Now we turn to the computation of the product $C = A \cdot B$. The operand B and the product element C are both represented in DB, whereas operand A is represented in SB. Starting from Equation (3.13) one obtains:

$$c_j = \mathrm{Tr}(\omega^j C) = \mathrm{Tr}(\omega^j AB) = \mathrm{Tr}((\omega^j B)A). \tag{3.18}$$

The first coefficient is then

$$\begin{aligned} c_0 = \mathrm{Tr}(BA) &= \mathrm{Tr}(a_0 B) + \mathrm{Tr}(a_1 \omega B) + \cdots + \mathrm{Tr}(a_{n-1} w^{n-1} B) \\ &= a_0 \mathrm{Tr}(B) + a_1 \mathrm{Tr}(\omega B) + \cdots + a_{n-1} \mathrm{Tr}(w^{n-1} B) \\ &= a_0 b_0 + a_1 b_1 + \cdots + a_{n-1} b_{n-1} \\ &= A \circ B, \end{aligned} \tag{3.19}$$

which is the dot product of the two factor elements. Hence, the coefficient c_1 turns out to be:

$$c_1 = \mathrm{Tr}((\omega B)A) = A \circ (\omega B).$$

However, the term (ωB) can be easily computed through Equation (3.14) by a left shift of the coefficients and computing of (3.16). The same procedure is applied iteratively to the other coefficients:

$$\begin{aligned} c_2 &= \mathrm{Tr}((\omega^2 B)A) = A \circ (\omega(\omega B)), \\ c_3 &= \mathrm{Tr}((\omega^3 B)A) = A \circ (\omega(\omega(\omega B))), \\ &\vdots \end{aligned}$$

The formulas developed are well suited for a matrix description of a parallel version of the multiplier. For this, each element will be denoted as a vector containing n elements:

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & \cdots & b_{n-2} & b_{n-1} \\ b_1 & b_2 & \cdots & b_{n-1} & B \circ Q \\ b_2 & b_3 & \cdots & B \circ Q & (\omega B) \circ Q \\ \vdots & \vdots & & \vdots & \vdots \\ b_{n-1} & B \circ Q & \cdots & (\omega^{n-3} B) \circ Q & (\omega^{n-2} B) \circ Q \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}. \tag{3.20}$$

An example for a DB multiplier is given below:

Example. We consider multiplication in the field $GF(2^4)$. The field polynomial is $Q(y) = y^4 + y + 1$. Assuming that operand $A = (a_0, a_1, a_2, a_3)$ is given in standard base and operand $B = (b_0, b_1, b_2, b_3)$ is given in dual base, a multiplication $C = A \cdot B$ is performed by

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 \\ b_1 & b_2 & b_3 & b_0 + b_1 \\ b_2 & b_3 & b_0 + b_1 & b_1 + b_2 \\ b_3 & b_0 + b_1 & b_1 + b_2 & b_2 + b_3 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}.$$

The product element C is also given in dual base coordinates.

The complexity of a hardware implementation of the DB multiplier is composed of the complexity for performing the matrix-vector multiplication (3.20) and of the complexity for computing the products $(\omega^j B)$, $j = 0, 1, \ldots, n-2$. The complexity for Equation (3.20) is

$$C_2 = n^2 \text{AND} + n(n-1)\text{XOR} = n^2 \text{AND} + (n^2 - n)\text{XOR}.$$

The complexity C_1 for the computation of one dot product is given in Equation (3.17). Hence the over all complexity $C = (n-1)C_1 + C_2$ is:

$$\#\text{AND} = n^2, \tag{3.21}$$

$$\#\text{XOR} = (n^2 - n) + (n-1)(h_w(Q) - 2) \geq n^2 - 1. \tag{3.22}$$

The number of AND gates required is, independent of the field polynomial, equal to n^2. For the XOR complexity is equality given, iff an irreducible *trinomial* is used as field polynomial, since then $h_w(Q) = 3$. The application of a trinomial in the example above leads therefore to $4^2 = 16$ AND gates and $4^2 - 1 = 15$ XOR gates. It seems interesting that the lower complexity bound of the dual base multiplier is exactly the same as the one of the standard base multiplier introduced in the previous section. Moreover, both architectures achieve the lower bound exactly when (certain) trinomials are used as field polynomials.

Another issue which must be addressed if the complexity of a DB multiplier is evaluated, is the different representations of the inputs and the output. The architecture

requires one operand, in our derivation A, to be in SB, and the other one, in our derivation B, to be in DB. The product element C is again represented in DB. As a consequence, it is likely that base transformations are required in actual systems which apply the architecture. Thus the complexity of the transformations must be taken into account. The base transformations are linear mappings which can be represented as the multiplication of a binary n element vector with a binary $n \times n$ matrix. However, in [GG93, Theorem 5.2] a condition is stated for which the base transformation is a mere permutation of the coefficients. The condition is, for extension fields of $GF(2)$, that the field polynomial is an irreducible *trinomial*. Since permutations can be hardwired in VLSI implementations, they require no extra gates. According to our complexity measure, which is a gate count, the permutation thus do not add to the complexity.

Consequently, the lower bound in Equation (3.22) is not only exactly fulfilled if $Q(y)$ is a trinomial, but also the multiplier does not require any arithmetic operations for the base transformation. Hence, (3.22) is the exact overall measure for the number of XOR gates needed for the multiplier if $Q(y)$ is a trinomial.

A Modification

In this subsection a somewhat modified version of the dual base multiplier introduced above is developed. In the modified algorithm, the element represented in SB will be cyclicly updated, rather than the input element in DB. For the modified architecture similar operations as for the one above, shift and add, will be used. Moreover, the modification will not alter the complexity.

In the beginning, multiplication of the element $A = a_0 + \omega a_1 + \omega^2 a_2 + \cdots + \omega^{n-1} a_{n-1}$ in SB with the base element ω is considered. Again, the field polynomial is $Q(y) = 1 + q_1 y + \cdots + q_{n-1} y^{n-1} + y^n$, such that $Q(\omega) = 0$. Then Equation (3.15) holds. The multiplication ωA is:

$$\begin{aligned} \omega A &= \omega a_0 + \omega^2 a_1 + \cdots + \omega^{n-1} a_{n-2} + \omega^n a_{n-1}, \\ &= a_{n-1} + (a_{n-1} q_1 + a_0)\omega + \cdots + (a_{n-1} q_{n-1} + a_{n-2})\omega^{n-1}. \end{aligned} \tag{3.23}$$

The operation ωA requires a cyclic right shift of the vector $(a_0, a_1, \ldots, a_{n-1})$ and an addition of the shifted vector with the vector $a_{n-1}(\cdot, q_1, \ldots, q_{n-1})$. The complexity of this operation is

$$C_1' = (h_w(Q) - 2)\,\text{XOR} \;\geq 1\,\text{XOR},$$

which is exactly the complexity which was required for the operation ωB, where B is given in DB.

In order to compute the product $C = A \cdot B$, where C is in dual base, we consider

$$c_j = \text{Tr}(\omega^j C) = \text{Tr}(\omega^j A B) = \text{Tr}((\omega^j A) B),$$

which is similar to Equation (3.18), except that the parentheses in the rightmost expression are placed differently. Then, according to Equation (3.19), the first coefficient of the

product is

$$c_0 = \mathrm{Tr}(AB) = A \circ B,$$

where "$\circ$" denotes the dot product of the two factor elements. Hence the coefficient c_1 turns out to be:

$$c_1 = \mathrm{Tr}((\omega A)B) = (\omega A) \circ B.$$

The term (ωA) can be computed through Equation (3.23). The same procedure is applied iteratively to the other coefficients:

$$\begin{aligned} c_2 &= \mathrm{Tr}((\omega^2 A)B) = (\omega(\omega A)), \circ B \\ c_3 &= \mathrm{Tr}((\omega^3 A)B) = (\omega(\omega(\omega A))) \circ B, \\ &\vdots \\ c_j &= \mathrm{Tr}((\omega^j A)B) = (\omega(\cdots(\omega A)\cdots)) \circ B. \end{aligned} \tag{3.24}$$

Equation (3.24) is a recursive description of the multiplier. Since every vector element requires a dot product, the complexity for the n Equations (3.24), $j = 0, 1, \ldots, n-1$ is:

$$C_2' = n^2\mathrm{AND} + n(n-1)\mathrm{XOR} = n^2\mathrm{AND} + (n^2 - n)\mathrm{XOR}.$$

The overall complexity can now be obtained through $C' = (n-1)C_1 + C_2'$:

$$\#\mathrm{AND} = n^2, \tag{3.25}$$

$$\#\mathrm{XOR} = (n^2 - n) + (n-1)(h_w(Q) - 2) \geq n^2 - 1. \tag{3.26}$$

Unfortunately, a general matrix description for a parallel multiplier such as developed in the previous subsection in Equation (3.20) is not as elegant in this case. However, an example of the modified multiplier makes the binary operations involved in Equation (3.24) more obvious.

Example. We consider multiplication in the field $GF(2^4)$. The field polynomial is $Q(y) = y^4 + y + 1$. The operand $A = (a_0, a_1, a_2, a_3)$ is given in standard base, the operand $B = (b_0, b_1, b_2, b_3)$ is given in dual base, and the product $C = (c_0, c_1, c_2, c_3)$ will be produced in dual base coordinates. The basic operation ωT, $T = (t_0, t_1, t_2, t_3)$ is for this example

$$\omega T = (t_3, t_0, t_1, t_2 + t_3).$$

A matrix description of the multiplication $C = A \cdot B$ is thus:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_3 & a_0 & a_1 & a_2 + a_3 \\ a_2 + a_3 & a_3 & a_0 & a_1 + a_2 + a_3 \\ a_1 + a_2 + a_3 & a_2 + a_3 & a_3 & a_0 + a_1 + a_2 + a_3 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

It should be noted that every row of the matrix introduces only one new addition for the rightmost entry, the other elements are simply a shifted version of the corresponding row above.

The complexity of the modified multiplier given in (3.25) and (3.26) is exactly the same as the complexity of the multiplier introduced earlier. Moreover, the lower bound of the XOR complexity is for both multipliers reached with equality, if the Hamming weight $h_w(Q)$ is minimal, i.e. if the field polynomial is a trinomial. The problem imposed by a possibly necessary base transformation remains the same for the modified multiplier.

The architectural differences between the two multipliers can be summarized as follows:

- The modified multiplier updates the input vector given in SB rather than the one given in DB.
- The update operation requires a *cyclic right shift* of the vector elements rather than a *non cyclic left shift*.
- The update operation adds values to certain elements of the input vector, whereas the original architecture adds certain vector elements in order to obtain the new element b_{n-1}.

However, it seems as though the differences do not lead to significant improvements/drawbacks in the implementation of either architecture, so that both architectures can be considered to be of similar "quality."

3.1.3 Normal Base Multipliers

A circuit design for the multiplication of two finite field elements represented in a normal base (NB) $(\omega, \omega^2, \omega^{2^2}, \ldots, \omega^{2^{n-1}})$ was first described by Massey and Omura in a US patent application [MO84]. Due to their inventors, NB multipliers are sometimes referred to as "Massey-Omura multipliers." Although the original description focuses on a bit serial multiplier, parallelization is straightforward. A parallel architecture can for instance be found in [WTS+85]. In the sequel, we will first develop the multiplier architecture and then comment on its complexity.

Consider two field elements A, B in NB:

$$A = a_0\omega + a_1\omega^2 + a_2\omega^{2^2} + \cdots + a_{n-1}\omega^{2^{n-1}}, \tag{3.27}$$

$$B = b_0\omega + b_1\omega^2 + b_2\omega^{2^2} + \cdots + b_{n-1}\omega^{2^{n-1}}. \tag{3.28}$$

One property of the NB representation is that squaring of field elements is merely a cyclic shift of its coefficients:

$$A^2 = a_{n-1}\omega + a_0\omega^2 + a_1\omega^{2^2} + \cdots + a_{n-2}\omega^{2^{n-1}}, \tag{3.29}$$

$$B^2 = b_{n-1}\omega + b_0\omega^2 + b_1\omega^{2^2} + \cdots + b_{n-2}\omega^{2^{n-1}}. \tag{3.30}$$

The field multiplication of two elements yields the product element C:

$$\begin{aligned} C &= A \cdot B \\ &= c_0\omega + c_1\omega^2 + c_2\omega^{2^2} + \cdots + c_{n-1}\omega^{2^{n-1}}. \end{aligned} \tag{3.31}$$

First, only the highest coefficient c_{n-1} is considered. It is an as yet unspecified, bilinear function of the two sets of input coefficients a_i, b_i:

$$c_{n-1} = f(a_0, a_1, \ldots, a_{n-1}; b_0, b_1, \ldots, b_{n-1}). \tag{3.32}$$

The binary function in (3.32) is sometimes referred to as "f-function." If now both sides of Equation (3.31) are squared:

$$\begin{aligned} C^2 &= A^2 \cdot B^2 \\ &= c_{n-1}\omega + c_0\omega^2 + c_1\omega^{2^2} + \cdots + c_{n-2}\omega^{2^{n-1}}, \end{aligned} \tag{3.33}$$

we obtain an expression similar to (3.32) for the coefficient c_{n-2}:

$$c_{n-2} = f(a_{n-1}, a_0, \ldots, a_{n-2}; b_{n-1}, b_0, \ldots, b_{n-2}). \tag{3.34}$$

The function in Equation (3.34) is the same as the f-function in (3.32) but with the two sets of input values $(a_0, a_1, \ldots, a_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1})$ cyclicly shifted. The other coefficients $(c_{n-3}, c_{n-4}, \ldots, c_0)$ can also obtained from the f-function through the same procedure, i.e. through repeated cyclic shifts of the input values.

Rather than providing general formulas for obtaining the f-function for a given field polynomial $Q(y)$, the method will be explained by an example.

Example. The field considered is $GF(2^4)$ with $Q(y) = y^4 + y^3 + 1$. The normal base is $(\omega^8, \omega^4, \omega^2, \omega)$, with $Q(\omega) = \omega^4 + \omega^3 + 1 = 0$. Multiplication of two field elements $C = A \cdot B$ in normal base yields:

$$\begin{aligned} C &= c_3\omega^8 + c_2\omega^4 + c_1\omega^2 + c_0\omega \\ &= A \cdot B = (a_3\omega^8 + a_2\omega^4 + a_1\omega^2 + a_0\omega)(b_3\omega^8 + b_2\omega^4 + b_1\omega^2 + b_0\omega) \\ &= \omega^{12}(a_2b_3 + a_3b_2) + \omega^{10}(a_1b_3 + a_3b_1) + \omega^9(a_3b_0 + a_0b_3) \\ &\quad +\omega^8(a_2b_2) + \omega^6(a_2b_1a_1b_2) + \omega^5(a_2b_0 + a_0b_2) \\ &\quad +\omega^4(a_1b_1) + \omega^3(a_0b_1 + a_1b_0) + \omega^2(a_0b_0) + \omega(a_3b_3) \end{aligned}$$

The multiplication has created the elements $(\omega^{12}, \omega^{10}, \omega^9, \omega^6, \omega^5, \omega^3)$ which have to be expressed in terms of the normal base:

$$\begin{aligned} \omega^{12} &= \omega^8 + \omega^4 + \omega^2, \\ \omega^{10} &= \omega^8 + \omega^2, \\ \omega^9 &= \omega^8 + \omega^4 + \omega, \\ \omega^6 &= \omega^4 + \omega^2 + \omega, \\ \omega^5 &= \omega^4 + \omega, \\ \omega^3 &= \omega^8 + \omega^2 + \omega. \end{aligned}$$

Hence, the coefficient c_3 is:

$$\begin{aligned} c_3 &= f(a_0, a_1, a_2, a_3; b_0, b_1, b_2, b_3) \\ &= a_2b_3 + a_3b_2 + a_1b_3 + a_3b_1 + a_3b_0 + a_0b_3 + a_2b_2 + a_0b_1 + a_1b_0. \end{aligned} \tag{3.35}$$

The sum of products in Equation (3.35) is the f-function which was to be determined.

It is obvious that normal base multiplication for a given field order is determined by the corresponding f-function. In turn, the complexity of the f-function determines the overall complexity of the multiplier. The number of products, C_n, of the f-function is often taken as a complexity measure. In the example above $C_n = 9$. Since the f-function is determined by the selected field polynomial $Q(y)$, the complexity of the NB multiplier is solely a function of the field polynomial $Q(y)$ for a given field $GF(2^n)$. Similarly to the situation for the standard base multiplier, we are now left with the choice of a field polynomial that results in a low complexity multiplier.

Mullin showed in [MOVW89] that the complexity is lower bounded by $C_n \geq 2n - 1$. NB with $C_n = 2n - 1$ are said to be *optimum normal bases*. In [Gei93a] NB for fields $GF(2^n)$, $2 \leq n \leq 60$, are listed. In this reference, the smallest possible complexity was determined for most field orders 2^n. The complexity for a hardware realization of the f-function is:

$$C_n \,\text{AND} + (C_n - 1)\,\text{XOR}.$$

The overall gate count of a parallel realization of a NB multiplier is thus lower bounded by:

$$\#\text{AND} = nC_n \geq 2n^2 - n, \tag{3.36}$$

$$\#\text{XOR} = (n-1)C_n \geq 2n^2 - 3n + 1. \tag{3.37}$$

The complexities (3.36) and (3.37) of a parallel NB multiplier are approximately twice as high as the complexities of the SB multiplier from Mastrovito, if they are compared with the corresponding lower bounds (3.9) and (3.10). However, finite field architectures based on NB are still attractive, in particular for cryptographic schemes which are based on the assumed difficulty of the discrete logarithm problem [Odl84]. The basic operation to be performed in these schemes is exponentiation in rather large fields; typical are fields with $100 < n < 1000$. NB architectures are inherently advantageous for squaring operations, because the cyclic shift which performs the squaring requires hardly any area in VLSI implementations. Since most algorithms for fast exponentiation require repeated squaring and multiplication, a trade off between the good squaring and costly multiplication behavior might be found, which suggests the use of NB architectures [GG90].

3.2 Non Traditional Multipliers

In this section several finite field architectures reported in technical literature are introduced, which are not "traditional" according to the classification used in this thesis. Except the multiplier which will be introduced first, all architecture take advantage of the decomposition of Galois fields into subfields.

Two of the four architectures which will be mentioned hereafter are relevant to the thesis, and will therefore be described in some detail. They are due to V. Afanasyev from the

Institute for Problems of Information Transmission (IPPI), Moscow. The architectures are described in two remarkable, though brief, publications [Afa90] [Afa91]. Unfortunately, it seems as though the architectures have not yet been recognized by the international scientific community as they deserve to be. Some of the subsequent chapters share some ideas with Afanasyev's architectures, although they were developed independently. In addition to these two architectures, a method for efficient table look-up and a normal base multiplier, both of which apply arithmetic in subfields, will be briefly described. An inverter over extension fields will be introduced in Subsection 3.3.3.

3.2.1 Multiplication in $GF(2^k)$ using the Karatsuba-Ofman Algorithm

In [Afa90][1] a method is introduced which allows the application of the Karatsuba-Ofman Algorithm (KOA) [KO63] [Knu81] to the multiplication of finite field elements from $GF(2^k)$. The elements are represented in standard base. The architecture optimizes the polynomial multiplication, which is the major part in standard base Galois field multiplication. The KOA allows polynomial multiplication with a reduced number of multiplications, while the number of additions is increased for short polynomials. Hence, multiplication must be more costly than addition. A straightforward application of the KOA requires $\log_2 k$ iteration steps for polynomials of degree $k-1$. For a detailed description of the KOA, refer to Section 5.2.

Since multiplication and addition are approximately both as costly in the field $GF(2)$, the KOA can not be applied to multiplication of elements from $GF(2^k)$ in a straightforward manner, since the elements are polynomials with coefficients from $GF(2)$. However, the method in [Afa91] suggests to apply only $\delta < \log_2 k$ iteration steps of the KOA to the field elements. As a consequence, the elementary operations are multiplication and addition with polynomials of degree $(m/2^\delta) - 1$. For the pure polynomial multiplication, this results in a complexity of:

$$\#\text{AND} = \left(\frac{3}{4}\right)^\delta k^2, \tag{3.38}$$

$$\#\text{XOR} \leq k^\delta[(\frac{m}{2^{-\delta}} - 1)^2 + 8\frac{k}{2^{-\delta}} - 2] - 8k + 2. \tag{3.39}$$

The second step which is required to perform SB multiplication is reduction modulo the field polynomial. The architecture uses the polynomials suggested by Mastrovito [Mas91] which can also be found in Table 3.1.

The overall complexity of the architecture is considerably better than the complexities of the traditional multipliers introduced earlier. In particular, the gate count is for most cases well below the k^2 bound. However, the architectures developed in Chapter 5 and

[1] The method is also described in the later reference [Afa91].

Chapter 6, which apply the KOA for multiplication in composite fields, perform somewhat better in terms of gate count.

The architecture is also highly modular, since all arithmetic is performed with the two kind of modules. One type of module provides multiplication of polynomials of degree $(m/2^\delta) - 1$, the other type provides addition with these polynomials.

3.2.2 Multiplication in Tower Fields

In reference [Afa91] a method for multiplication in finite fields is developed. The method is based on field extensions of degree two.

The elementary operation is the following. We consider a field $GF(2^q)$ with a field polynomial of type $P(x) = x^2 + x + p_0$ (see Theorem 11 for proof of existence.) Multiplication of two elements $A, B \in GF(2^q)$ can be performed through

$$\begin{aligned} C(x) &= A(x) \cdot B(x) = (a_1 x + a_0)(b_1 x + b_0), \\ &= (a_0 b_0 + p_0 a_1 b_1) + x([a_1 + a_0][b_1 + b_0] + a_0 b_0), \end{aligned} \tag{3.40}$$

which requires 3 general multiplications, 4 additions and 1 constant multiplication with p_0. All operations refer to arithmetic in the subfield $GF(q)$.

The basic idea of the method is to decompose the field $GF(2^k)$ of operation into subfields with (multiple) extensions of degree two. This means that for $k = n2^\delta$, the field $GF(2^k)$ is decomposed into δ subfields of the form

$$GF(2^k) \cong GF((\cdots(((2^n)^2)^2)\cdots)^2). \tag{3.41}$$

Fields of the form (3.41) are referred to as "tower fields." For multiplication in tower fields, Formula (3.40) can be applied *recursively*.

The space complexity of this architecture is remarkably low. The table given in [Afa91] contains the gate count for different decompositions of the fields $GF(2^8)$ and $GF(2^{16})$. For the field $GF(2^8)$, the best result is achieved with $\delta = 1$; the gate count is 65 XOR / 48 AND. For the field $GF(2^{16})$, the best field decomposition is found to be $\delta = 2$ which yields a gate count of 234 XOR / 144 AND. To the author's knowledge, the latter gate count is the lowest one for parallel finite field multiplier reported in technical literature. Compared to the architectures in Chapter 6, the tower field multiplier has almost exactly the same count for the fields with $k = 8, 16$. This can be seen by considering Table 6.1 and Table 6.2, respectively.

Although the gate count is extremely low, the architecture is somewhat lacking the modularity which is inherent in the architectures which apply the KOA. Both types of multiplier architectures that use the KOA, the one given above in Section 3.2.1 and the multipliers over composite fields introduced in the subsequent chapters, require only arithmetic modules from *one* subfield. On the other hand, the tower field multiplier requires arithmetic modules from δ *different* subfields, thus increasing the number of different modules.

3.2.3 Other Architectures

In this subsection references to two more schemes are provided which use subfields of Galois fields. Since the schemes are less relevant for the architectures to be developed in the thesis, they will only briefly be mentioned.

The first architecture is by Pincin [Pin89]. It is a parallel normal base multiplier over $GF(2^k)$ which uses arithmetic in subfields. The architecture is suited for a decomposition in multiple subfields, which are named "descending chain" of fields. For fields $GF(2^{2^s})$, the computational complexity of the architecture is of order $\mathcal{O}(m^{2.32})$.

The second algorithm is by Hsu et al. [HTRG88]. It deals with the use of a subfield $GF(2^{k/2})$ for performing a table lookup in the field $GF(2^k)$. Unlike the architecture described previously, the algorithm uses table lookup for all operations in the subfield. In one extensive example developed in the reference, a VLSI architecture for table lookup in the field $GF(2^8)$ is given which occupies about half the area of a straightforward implementation.

3.3 Inversion

3.3.1 Direct Inversion over $GF(2^n)$

To the author's knowledge there are only a few schemes for parallel, or direct, inversion over Galois fields $GF(2^n)$ reported in technical literature. The majority of the publications deals with bit serial architectures (see e.g. [Fen89], [HWB92a], or [KRV93] for recent references.) One recent bit parallel architecture was briefly proposed in [Mas91] by Mastrovito. This architecture will be used for the inverter over composite field from Chapter 8. Another method for direct inversion was described in an early paper by Davida [Dav72]. It will also be described briefly.

The method introduced in the sequel is based on the inversion of the product matrix of the Mastrovito multiplier, introduced earlier. It is described in [Mas91, Section 9.2]. From the matrix Equation (3.1), we can derive

$$\mathbf{Z}^{-1} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Moreover, we know from Equation (3.2) that the first column of $\mathbf{Z}$ contains the coefficients of the element which generated the matrix. Obviously, this element is $A^{-1} = (a'_0, a'_1, \ldots, a'_{n-1})^T$ in the equation above. Hence the first column of $\mathbf{Z}^{-1}$ is filled with the coefficients of A's inverse.

In order to perform inversion in parallel, the general equations for the coefficients of the first column of $\mathbf{Z}^{-1}$ must be derived. There are two method for matrix inversion

available. The first one solves a system of linear equations, the second one is based on the computation of adjoints. Both methods are computationally costly; the solution of linear equations is of order $\mathcal{O}(n^3)$, the computation of adjoints is of order $\mathcal{O}(n!)$.

For the inverter over composite field to be developed in Chapter 8, fast parallel inverters in small subfields $GF(2^n)$ are required. Although matrix inversion based on adjoints is potentially more complex, we found it better suited for the computation with *Mathematica*, a program for symbolic computation [Wol88]. For each field $GF(2^n)$ we computed a set of n equations for the coefficients a'_i, $i = 0, 1, \ldots, n-1$. We used the matrices $\mathbf{Z}$ as entries for the computation. The coefficient a'_i is obtained through

$$a'_i = \frac{\mathrm{adj}_{0,i}(\mathbf{Z})}{\det(\mathbf{Z})}.$$

However, since the inverse of $\mathbf{Z}$ always exists, the expression $\det(\mathbf{Z})$ is always equal to one and needs not to be computed. The coefficient a'_i can thus be computed through

$$a'_i = \mathrm{adj}_{0,i}(\mathbf{Z}).$$

The adjoint is a determinant of order $(n-1) \times (n-1)$. *Mathematica* was able to determine the equations for direct inversion for fields up to $n = 8$. However, the equations for $n = 8$ were found to be too complex, so that Appendix A lists only equations for fields $GF(2^n)$, $n \leq 7$. The equations provided are in "raw" form, i.e. the contain redundancies and should be further simplified for actual implementations. The field polynomials used are the same as listed in Table 3.1.

In [Dav72] another method for direct inversion is introduced. It is also based on matrix description. However, the resulting system of equation is of degree $2n - 1$. The corresponding matrix is sparse. Unfortunately, the author does not comment on the computational complexity required for solving these equations. The only example given is for the small field $GF(2^n)$.

3.3.2 Inversion in Composite Fields

Itoh and Tsujii proposed (briefly) in [IT88, Section 6] a new method for the inversion of elements of composite fields. Their approach assumes a normal base representation of the field elements. The basic idea is that inversion in the field $GF((2^n)^m)$ is replaced by inversion in the ground field $GF(2^n)$. For the latter one an approach based on Fermat's Theorem is used.

The algorithm will be described in detail in Chapter 8, where a parallel inverter for composite field elements in standard base is developed.

3.3.3 Inversion in Tower Fields

The following scheme also operates over multiple extension fields of $GF(2^n)$. An efficient parallel architecture for computing the multiplicative inverse of finite field elements was

first proposed in 1988 by Morii and Kasahara in [MK89]. The same algorithm was also proposed by Afanasyev in 1991 [Afa91], apparently unaware of the earlier publication.

The method reduces the problem of inversion in the Galois field $GF(2^k)$ to inversion in the subfield $GF(2^{k/2})$. The core part of the architecture is the following.

Let us consider an element A from $GF((2^{k/2})^2)$, represented in SB:

$$A(x) = a_0 + a_1 x; \quad a_0, a_1 \in GF(2^{k/2}).$$

The field polynomial is of the form $P(x) = x^2 + x + p_0$, where $p_0 \in GF(2^{k/2})$. If the inverse of A is denoted as $B = A^{-1}$, the equation

$$\begin{aligned} A \cdot B &= (a_0 + a_1 x)(b_0 + b_1 x) \bmod P(x) \\ &= [a_0 b_0 + p_0 a_1 b_1] + [a_0 b_1 + a_1 b_0 + a_1 b_1] x \\ &= 1, \end{aligned} \tag{3.42}$$

must be satisfied, which is equivalent to the set of two linear equations in b_0, b_1 over $GF(2^{k/2})$:

$$\left.\begin{array}{rcrcl} a_0 b_0 & + & p_0 a_1 b_1 & = & 1 \\ a_1 b_0 & + & (a_0 + a_1) b_1 & = & 0 \end{array}\right\}. \tag{3.43}$$

The solution of (3.43) is

$$\left.\begin{array}{rcl} b_0 &=& \frac{a_0 + a_1}{a_0(a_0 + a_1) + p_0 a_1^2} \\ b_1 &=& \frac{a_1}{a_0(a_0 + a_1) + p_0 a_1^2} \end{array}\right\}. \tag{3.44}$$

The variables b_0, b_1 are the coefficients of A's inverse with respect to the subfield $GF(2^{k/2})$.

The computation of the two Equations (3.44) requires 1 inversion, 3 general multiplications, 2 additions, 1 constant multiplication with p_0 and 1 squaring. All these operations are performed in $GF(2^{k/2})$. The main advantage of this algorithm is that the inversion is now performed in the subfield, which is supposed to be considerably easier than in the field $GF(2^k)$. The overhead to be paid for this are the other arithmetic operations. Both references recommend a recursive application of the algorithm, which leads to tower fields as introduced in Equation (3.41).

Neither reference provides gate counts for inverters over certain fields. However, in [Afa91] it is stated that the complexity is of order $\mathcal{O}(m^{\log_2 3} \log m)$ under certain conditions regarding the coefficient p_0.

Chapter 4

Parallel Constant Multipliers

4.1 Constant Multipliers over $GF(2^n)$

In this section an efficient scheme for performing parallel multiplication of an arbitrary element from the field $GF(2^n)$ with a fixed, i.e. constant, element is developed. The results to be obtained will be used in most of the subsequent chapters, in particular for the general multipliers in Chapters 5 and 6, and for the inverter introduced in Chapter 8. Moreover, multiplication with a constant field element is extremely important for Reed-Solomon encoders, see e.g. [LC83]. The algorithm which will be introduced here can be directly applied to Reed-Solomon encoders over fields $GF(2^n)$.

First, two greedy algorithms will be developed. The algorithms minimize the number of XOR gates which is required to implement constant multipliers. Results on the performance of the algorithms compared to a straightforward approach will be provided. In the appendix, complete lists of optimized complexities for multiplication with all elements from the fields $GF(2^n)$, $n = 4, 5, \ldots, 8$ are given, which can, for instance, be used in Reed-Solomon encoders over fields $GF(2^n)$.

4.1.1 Two Suboptimal Algorithms

The general approach taken here is the application of the Mastrovito multiplier, introduced in Section 3.1.1, to constant multiplication. This approach was previously described in [Mas91, Chapter 5.1.5]. However, the major concern of this section is the application of a greedy, i.e. locally optimum, algorithm which optimizes the gate count of the constant multipliers.

Equation (3.1) is a matrix description of the general multiplication in a field $GF(2^n)$. The product matrix $\mathbf{Z}$ is a function of the (variable) element A and the field polynomial $Q(y)$. If the element A is now chosen to be constant, a binary product matrix with *fixed* entries is obtained. The multiplication with the constant A is thus entirely described by the binary matrix. We may explain the scheme through an Example.

Example. Let $Q(y) = y^7 + y + 1$ be the primitive polynomial generating $GF(2^7)$. The primitive element of the field is denoted ω, where $Q(\omega) = 0$. The multiplication of a variable field element $B = (b_0, b_1, \ldots, b_6)$ with the fixed element $A = \omega^{47} = (1111100)$ is described by:

$$\begin{aligned} C &= \omega^{47} B = \mathbf{Z} B \qquad (4.1) \\ &= \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_6 \end{pmatrix} = \begin{pmatrix} b_0 + b_3 + b_4 + b_5 + b_6 \\ b_0 + b_1 + b_3 \\ b_0 + b_1 + b_2 + b_4 \\ b_0 + b_1 + b_2 + b_3 + b_5 \\ b_0 + b_1 + b_2 + b_3 + b_4 + b_6 \\ b_1 + b_2 + b_3 + b_4 + b_5 \\ b_2 + b_3 + b_4 + b_5 + b_6 \end{pmatrix}. \end{aligned}$$

Each operation "+" in (4.1) denotes a mod 2 addition, i.e. a two input XOR.

In the sequel we consider the space complexity of constant multiplication. As the example from above shows, the only operation required for constant multiplication is modulo 2 addition. Hence we define the space complexity as the number of XOR gates needed for the multiplier.

In [Mas91, Chapter 5.1.5] a formula for the *average* complexity for constant multiplication in the field $GF(2^n)$ is developed:

$$\overline{C}_{cnst} = \frac{n^2}{2} - n \ \ [\text{XOR}]. \qquad (4.2)$$

This value is only an estimate which presumes product matrices $\mathbf{Z}$, which have on average exactly $n^2/2$ entries. However, we determined the actual complexities for fields $n < 10$ and found the estimation accurate.

Equation (4.2) is the average of the straightforward realizations of all 2^n binary matrices of type (4.1). For instance, a straightforward realization of the constant multiplier in the example above requires 26 XOR gates, since there are 26 modulo 2 additions to be performed. However, it is rather obvious that there are redundancies in the example above, which allow a reduction of the number of XOR gates. For instance, a straightforward realization of the matrix in (4.1) would compute the sum $(b_0 + b_1)$ four times, since it appears in the rows 2,3,4, and 5. In the sequel, two greedy algorithm will be developed which finds suboptimal solutions.

The reduction of the number of XOR gates is a optimization problem on Boolean equations of form (4.1). The cost function of the optimization problem is the number of mod 2 additions required to realize a set of n equations in n variables b_i, $i = 0, 1, \ldots, n-1$, where each equation is a sum over certain b_i. The greedy algorithms operate iteratively.

The first algorithm computes in each iteration step the occurrence of all possible pairs $b_i + b_j$, $i, j = 0, 1, \ldots, p$, $i \neq j$. The most frequent occurring pair $b_{opt1} + b_{opt2}$ can be precomputed. Thus, a locally optimum solution is found. The pair is considered a new

```
lastcol := n-1;
M := zmatrix;
DO BEGIN
   hmax := 0;
   FOR i := 0 TO lastcol-1 DO BEGIN
       FOR j := i+1 TO lastcol DO BEGIN
           coli := GETCOLUMN(M,i);
           colj := GETCOLUMN(M,j);
           IF(HAMMINGWEIGHT(coli & colj) > hmax) BEGIN
               hmax := HAMMINGWEIGHT(coli & colj);
               maxi := i;
               maxj := j;
           END;
       END;
   END;
   IF (hmax > 1) DO BEGIN
       mxcoli := GETCOLUMN(M,maxi);
       mxcolj := GETCOLUMN(M,maxj);
       newcol := maxcoli & maxcolj;
       PUTCOLUMN(M,newcol,lastcol+1);
       PUTCOLUMN(M,!(newcol & maxcoli),i);
       PUTCOLUMN(M,!(newcol & maxcolj),j);
       lastcol := lastcol+1;
   END;
WHILE (hmax > 1);
```

Pseudo Code of the algorithm Greedy 1

element $b_\mu = b_{opt1} + b_{opt2}$, and the matrix is extended such that it also contains the new element. Again, in the next iteration step all possible pairs $b_i + b_j$, $i, j = 0, 1, \ldots, p+1$, $i \neq j$ are investigated, including the new element b_μ. The algorithm eventually terminates when all possible pairs occur only once. A more detailed explanation of the first greedy algorithm is given by the pseudo code description.

The pseudo code assumes the function `GETCOLUMN(M,i)`, which returns the column `i` of the passed matrix `M`, the function `PUTCOLUMN(M,col,i)`, which replaces the column `i` with the new column `col`, and the function `HAMMINGWEIGHT(col)`, which returns the Hamming weight of the passed column vector `col`. The operator `&` performs bitwise logical AND, the operator `!` computes the bitwise inverse of its argument. The algorithm in the pseudo code operates iteratively on the matrix `M`. In each iteration step one new column is appended to the matrix. This new column refers to the sum $b_i + b_j$ which can be precomputed with one XOR gate. The new column has `hmax` entries. At the same

time, the columns `maxcoli` and `maxcolj` are updated through a logical AND with the new column, thus eliminating 2 `hmax` entries. Each elimination refers to the saving of one XOR gate. Hence, every iteration step saves $2\,\texttt{hmax} - \texttt{hmax} - 1 = \texttt{hmax} - 1$ XOR gates. The number of entries is in each iteration step reduced by `hmax`. This property together with the fact that the algorithms terminates if there are no column pairs with $\texttt{hmax} > 1$ left, assures convergence of the algorithm.

To clarify the understanding of the algorithm, we apply the first greedy algorithm to the matrix belonging to the example from above.

Example. We reconsider multiplication with the element $A = \omega^{47} = (1111100)$ from $GF(2^7)$, with $Q(y) = y^7 + y + 1$ being the field polynomial. In the example above, a matrix description of the multiplication of A with the variable field element $B = (b_0, b_1, \ldots, b_6)$ was developed:

$$C = \omega^{47} B = \begin{pmatrix} b_0 + b_3 + b_4 + b_5 + b_6 \\ b_0 + b_1 + b_3 \\ b_0 + b_1 + b_2 + b_4 \\ b_0 + b_1 + b_2 + b_3 + b_5 \\ b_0 + b_1 + b_2 + b_3 + b_4 + b_6 \\ b_1 + b_2 + b_3 + b_4 + b_5 \\ b_2 + b_3 + b_4 + b_5 + b_6 \end{pmatrix}.$$

The straightforward implementation of the constant multiplication requires 26 additions. The summation from above can also be represented by the binary matrix $\mathbf{Z}$, which is the initial matrix for the greedy algorithm.

b_0	b_1	b_2	b_3	b_4	b_5	b_6
1	0	0	1	1	1	1
1	1	0	1	0	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0
1	1	1	1	1	0	1
0	1	1	1	1	1	0
0	0	1	1	1	1	1

In the sequel the iterations of the greedy algorithm on the matrix are displayed.

1. In the first iteration step it is found that the addition $b_0 + b_1$ is to be precomputed. The value `hmax` equals 4, which is the Hamming weight of the ANDed first and second column. The sum of both is considered a new element $b'_7 = b_0 + b_1$. The columns 0 and 1 are updated and the new column is added to the matrix.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b'_7
1	0	0	1	1	1	1	0
0	0	0	1	0	0	0	1
0	0	1	0	1	0	0	1
0	0	1	1	0	1	0	1
0	0	1	1	1	0	1	1
0	1	1	1	1	1	0	0
0	0	1	1	1	1	1	0

2. The second iteration step determines the new element $b'_8 = b_2 + b_3$.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b'_7	b'_8
1	0	0	1	1	1	1	0	0
0	0	0	1	0	0	0	1	0
0	0	1	0	1	0	0	1	0
0	0	0	0	0	1	0	1	1
0	0	0	0	1	0	1	1	1
0	1	0	0	1	1	0	0	1
0	0	0	0	1	1	1	0	1

3. The third iteration step determines the new element $b'_9 = b_4 + b_5$.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b'_7	b'_8	b'_9
1	0	0	1	0	0	1	0	0	1
0	0	0	1	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	0
0	0	0	0	0	1	0	1	1	0
0	0	0	0	1	0	1	1	1	0
0	1	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	1	1

4. The forth iteration step determines the new element $b'_{10} = b_4 + b'_7$.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b'_7	b'_8	b'_9	b'_{10}
1	0	0	1	0	0	1	0	0	1	0
0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	1	1	0	0
0	0	0	0	0	0	1	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	1	0	1	1	0

5. The fifth iteration step determines the new element $b'_{11} = b_6 + b'_8$.

b_0	b_1	b_2	b_3	b_4	b_5	b_6	b'_7	b'_8	b'_9	b'_{10}	b'_{11}
1	0	0	1	0	0	1	0	0	1	0	0
0	0	0	1	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	0	1	0	1

The updated matrix does not contain any column pairs with an ANDed Hamming weight that is greater one. Hence the algorithm terminates hereafter.

If the elements $(b'_7, b'_8, \ldots, b'_{11})$ are precomputed in exactly this sequence, one obtains the following equations for the constant multiplication

$$C = \omega^{47} B = \begin{pmatrix} b_0 + b_3 + (b_4 + b_5) + b_6 \\ (b_0 + b_1) + b_3 \\ ((b_0 + b_1) + b_4) + b_2 \\ (b_0 + b_1) + (b_2 + b_3) + b_5 \\ ((b_0 + b_1) + b_4) + ((b_2 + b_3) + b_6) \\ b_1 + (b_2 + b_3) + (b_4 + b_5) \\ ((b_2 + b_3) + b_6) + (b_4 + b_5) \end{pmatrix},$$

which can be computed with 16 modulo 2 additions. The greedy algorithm has thus reduced the number of XOR gates needed for a hardware implementation from 26 to 16.

The final complexity can be obtained from the matrix as follows. The entire Hamming weight, i.e. the number of all entries of the binary matrix after the last iteration step minus n, the number of rows, is the complexity of a pure matrix vector multiplication. In the case above we have $18 - 7 = 11$ additions. In order to get the exact overall complexity, the number of precomputed terms (5) must be added, which results in $11 + 5 = 16$.

The greedy algorithm has three properties which makes its application attractive:

1. The algorithm is monotone, i.e. the cost function (# XOR) is reduced in every iteration step.

2. The algorithm always converges, as was stated above.

3. The algorithm is fast. Running on an IBM PS2/486, the algorithm optimized the matrices for all elements from the field $GF(2^{15})$ in less than 2 hours, which is an average time per matrix of less than 220 msec.

As the example above showed, relatively large improvements are possible for certain field elements. However, it must be emphasized that the algorithm is only *locally* optimum and does not guarantee *globally* optimum solutions. Before the performance of the algorithm for actual Galois fields $GF(2^n)$, $n = 4, 5, \ldots, 16$, is compared with the straightforward approach, an improved version of the algorithm will be introduced.

One rather obvious improvement which can be implemented, is that the greedy algorithm checks *all* possible pairs of rows which possess a maximum `hmax`. For instance, in the example above, four pairs $(b_0 + b_1), (b_1 + b_2), (b_2 + b_3), \text{and} (b_2 + b_4)$ were possible candidates for precomputing in the very first step, since all pairs had a `hmax` of four. Hence, a second algorithm was implemented, which checked all possible pairs that have a maximum `hmax`. The principal structure of the algorithm is the same as the one given in the pseudo code description above. In order to add the new feature, the algorithm was changed such that it works recursively.

Example. Application of the second greedy algorithm to the optimization problem from above:

$$C = \omega^{47} B = \begin{pmatrix} b_0 + b_3 + b_4 + b_5 + b_6 \\ b_0 + b_1 + b_3 \\ b_0 + b_1 + b_2 + b_4 \\ b_0 + b_1 + b_2 + b_3 + b_5 \\ b_0 + b_1 + b_2 + b_3 + b_4 + b_6 \\ b_1 + b_2 + b_3 + b_4 + b_5 \\ b_2 + b_3 + b_4 + b_5 + b_6 \end{pmatrix},$$

yields another sequence of precomputations. The best sequence found is:

$$\begin{aligned} b'_7 &= b_0 + b_1, \\ b'_8 &= b_3 + b_4, \\ b'_9 &= b_2 + b_5, \\ b'_{10} &= b_6 + b_8, \\ b'_{11} &= b_2 + b'_7, \\ b'_{12} &= b_3 + b'_7. \end{aligned}$$

Application of the precomputations to the constant multiplication gives the optimized equations:

$$C = \omega^{47} B = \begin{pmatrix} b_0 + ((b_3 + b_4) + b_6) + b_5 \\ ((b_0 + b_1) + b_3) \\ ((b_0 + b_1) + b_2) + b_4 \\ ((b_0 + b_1) + b_3) + (b_2 + b_5) \\ ((b_0 + b_1) + b_2) + ((b_3 + b_4) + b_6) \\ b_1 + (b_2 + b_5) + (b_3 + b_4) \\ (b_2 + b_5) + ((b_3 + b_4) + b_6) \end{pmatrix},$$

which corresponds to a realization with $6 + 8 = 14$ XOR gates. Hence the new algorithm gives an improvement of 2 XOR gates compared to the first greedy algorithm and an improvement of 12 XOR gates compared to the straightforward approach.

n	$Q(y)$	$\overline{C}_{cnst}$	Greedy 1		Greedy 2	
		XOR	XOR	rel. impr.	XOR	rel. impr.
4	4,1,0	4	3.3	17.5%	3.3	17.5%
5	5,2,0	7	5.4	22.9%	5.3	24.3%
6	6,1,0	12	8.1	32.5%	7.9	34.2%
7	7,1,0	17	11.3	33.5%	11.0	35.3%
8	8,5,3,2,0	24	14.9	37.9%	14.4	40.0%
9	9,4,0	31	19.2	38.1%	18.5	40.3%
10	10,3,0	40	23.8	40.5%	22.8	43.0%
11	11,2,0	49	28.7	41.4%		
12	12,8,5,1,0	60	33.3	44.5%		
13	13,7,6,1,0	71	39.3	44.6%		
14	14,9,7,2,0	84	45.4	46.0%		
15	15,1,0	97	52.9	45.5%		
16	16,11,6,5,0	112	59.1	47.2%		

Table 4.1: Comparison of the average complexity of constant multiplication in the fields $GF(2^n)$: Straightforward vs. Optimized Solutions.

4.1.2 Experimental Results

In this subsection some experimental results of practical relevance regarding optimized constant multiplication are given. First, a performance measure compares the two greedy algorithms from the previous section with the complexity of the straightforward implementation. Second, the appendix lists the optimized complexity of constant multiplication with all elements from the fields $GF(2^n)$, $n = 4, 5, \ldots, 8$.

The average number of XOR gates for constant multiplication in one field $GF(2^n)$ serves as a performance measure of the algorithms. The straightforward approach requires, according to Equation (4.2), an average of $\overline{C}_{cnst} = n^2/2 - n$ XOR gates per multiplier. We applied both algorithms introduced above, Greedy 1 and Greedy 2, to all elements of the fields $GF(2^n)$, $n = 4, 5, \ldots, 16$, and computed the average number of modulo 2 additions. The results are given in Table 4.1. For the fields $n > 11$, Greedy 2 was not fast enough to optimize all elements, so that the fields $11 < n \leq 16$ were only optimized with Greedy 1.

The table lists the field polynomial $Q(y)$ next to the field exponent n. The column headed by $\overline{C}_{cnst}$ contains the average complexity of a straightforward realizations, computed with Equation (4.2). The columns headed by Greedy 1 and Greedy 2 contain the average optimized complexity (measured in XOR gates), and the improvement relative to the straightforward approach.

It can be seen that both greedy algorithms reduce the space complexity considerably. The algorithms gain more as n increases. This is due to the also increasing number of entries in the $n \times n$ matrices, which results in a higher probability of redundancies.

The actual complexities of multiplication with the elements from the fields $GF(2^n)$, $n = 4, 5, \ldots, 8$ can be found Appendix B. The lists in the appendix can, for instance, be used for the evaluation of the gate count of constant multipliers needed in an implementation of a Reed-Solomon encoder. Moreover, the lists give some insight into the complexity behavior of different field elements. For example, it can be seen that the first few $\omega, \omega^2, \ldots$ and the last few elements $\ldots, \omega^{2^n-3}, \omega^{2^n-2}$ (ordered by their exponents) of each field have a gate count which is significantly lower than the average complexity. A direct application for the optimized constant multipliers is given in the architectures of multipliers over composite fields, introduced in the Chapters 5 and 6. It was found that a careful choice of field polynomials $P(x)$, which have coefficients that possess a small constant multiplication complexity, leads to a significantly improved gate count for the operation $\mathrm{mod} P(x)$.

4.2 Constant Multipliers over $GF((2^n)^m)$

This section develops a general architecture of constant multipliers in composite fields $GF((2^n)^m)$. We will focus on the development of an upper bound for the *average* complexity of the constant multiplication. Results from the previous section will be used for this. The composite fields considered are isomorphic to $GF(2^k)$, with $k = nm$. Every element A of the composite field can be represented as a polynomial with m coefficients from $GF(2^n)$:

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_0 \;\; ; \;\; a_i \in GF(2^n) \;\; ; \;\; A \in GF((2^n)^m). \tag{4.3}$$

Multiplication of two field elements $C = A \cdot B$ can be performed in standard base as

$$C(x) = A(x) \times B(x) \bmod P(x), \tag{4.4}$$

where $P(x)$ is the field generator of degree m over $GF(2^n)$. In order to perform constant multiplication we consider one element (polynomial) to be fixed. In the remainder of this section we chose A as the fixed input element.

We can separate the two steps required for the field multiplication, which are ordinary polynomial multiplication ($\times$) and reduction modulo the field polynomial (mod). The second step, modulo reduction, will be treated thoroughly in Chapter 5, Section 5.4.

We turn now to the first step, the multiplication of a polynomial $A(x)$, with *constant* coefficient from $GF(2^n)$ with a polynomial $B(x)$ with arbitrary coefficients. Theorem 8 states that a straightforward approach allows polynomial multiplication with m^2 multiplications and $(m-1)^2$ additions. All arithmetic operations are performed in the ground field $GF(2^n)$. Each multiplication involves one constant coefficient a_i from A, and one variable coefficient b_i from B. Hence Equation 4.2 can be applied, which provides the average complexity $\overline{C}_{cnst}$ for one multiplication:

$$\otimes = \overline{C}_{cnst} = \frac{n^2}{2} - n \;\; [\mathrm{XOR}].$$

Every addition involves two products of the form $a_i \cdot b_j$, which are also variable since all b_i are variable. Therefore each addition is a general addition with a complexity of

$$\oplus = n \ \ [\text{XOR}].$$

Now an upper bound for the average complexity $\overline{C}_{pol}$ for the multiplication of two polynomials over $GF(2^n)$, one with fixed and one with variable coefficients, can be stated:

$$\begin{aligned} \overline{C}_{pol} &= m^2 \otimes + (m-1)^2 \oplus \\ &= \frac{(nm)^2}{2} - 2nm + n \ \ [\text{XOR}] \\ &= \frac{k^2}{2} - 2k + n \ \ [\text{XOR}]. \end{aligned} \tag{4.5}$$

The average complexity is of order $\mathcal{O}((nm)^2)$. This is the same order as for the constant multipliers in $GF(2^k)$, given in Equation (4.2). However, the complexity in (4.5) over $GF((2^n)^m)$ behaves slightly better than the constant multipliers over $GF(2^k)$, $k = nm$, since a comparison of the linear terms shows that

$$-2k + n < -k,$$

for all k and n.

In order to evaluate the complexity of the entire constant multiplication, the reduction mod$P(x)$ must be performed. This complexity depends heavily on the field polynomial chosen. Therefore we do not provide general expressions which would lead to complexities that are much too high. However, in the column headed by " mod " of Table 5.1 are actual complexities for the modulo reduction with *optimized* field polynomials listed. The results from there can be immediately applied to the case of constant multiplication treated here.

Finally it should be mentioned that for an actual element from $GF((2^n)^m)$ the complexity might be considerably smaller than the one in Equation (4.2). First, one should try to apply one of the greedy algorithms from the previous section to the m^2 individual constant multiplications in $GF(2^n)$. Second, constant multiplication which applies the Karatsuba-Ofman algorithm, to be developed in Section 5.2, can lead to lower complexities. This is particularly likely for large values of n. Again, it is difficult to provide general statements due to the strong dependency on the structure of the actual element considered.

Chapter 5

Multipliers over General Composite Fields $GF((2^n)^m)$

Parts of this chapter were presented in [Paa93b] and [Paa93a].

5.1 Principal

In this chapter a parallel multiplier with low complexity in the composite field $GF((2^n)^m)$ will be developed. The fields considered are of the form $GF((2^n)^{2^i})$, i integer. The elements of the field may be represented in the standard (or canonical) basis as polynomials with a maximum degree $m-1$ over $GF(q)$:

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_0 \;\; ; \;\; a_i \in GF(q) \;\; ; \;\; A \in GF(q^m).$$

The generator of the extension field is a primitive polynomial $P(x)$ of degree m over $GF(2^n)$. Multiplication of two elements A and B of the extension field can be performed in the standard representation as:

$$A(x) \times B(x) \bmod P(x). \tag{5.1}$$

The field multiplication in (5.1) may be performed in two steps:

1. Ordinary polynomial multiplication ($\times$);
2. Reduction modulo the generating polynomial (mod).

We will treat both steps separately in the following sections. The basic arithmetic operations, addition and multiplication, which are required for both steps are actually performed in the ground field $GF(2^n)$.

The basic idea of the multiplier introduced here is the application of the Karatsuba-Ofman Algorithm (KOA) [KO63] for efficient multiplication of polynomials over a field $\mathcal{F}$ to step 1. *Efficient* refers to the fact that the algorithm saves multiplications at the

cost of extra additions. Hence, if the algorithm is expected to be an improvement in complexity, multiplications must be more "costly" than additions. This condition is naturally fulfilled for polynomials over fields $GF(2^n)$. Addition in the ground field is performed by n parallel mod 2 adders since the field characteristic is still 2. On the other hand, the number of elementary gates required for a multiplier in standard representation in $GF(2^n)$ realized with traditional architectures is at least $n^2 - 1$ mod 2 adders and n^2 mod 2 multipliers, respectively. Our architecture applies the Mastrovito multiplier [Mas89] to the multiplication of the polynomial coefficients.

The outline of remainder of the chapter is as follows. Section 5.2 deals, after a brief description of some previous work, with a thorough investigation of the computational complexity of the KOA and of the time complexity of its parallel implementation. A matrix description of the algorithm, which can, for instance, be used for a VLSI description, will be developed too. Section 5.3 shows the application of the KOA to polynomials over fields of characteristic 2. Section 5.4 describes how primitive polynomials with low complexity with respect to modulo reduction can be determined. Section 5.5 shows the best multipliers found for composite fields up to $GF(2^{32})$. For each multiplier the exact complexities are given together with the composition nm and a comparison with the k^2 complexity bound of traditional multipliers. As an example, for the important field $GF(2^{16})$ a multiplier is explained and a block diagram of the architecture given.

5.2 The Karatsuba-Ofman Algorithm

5.2.1 Introduction

The Karatsuba-Ofman algorithm (KOA) is a recursive method for efficient polynomial multiplication or efficient multiplication in positional number systems (which is actually the same task.) The algorithm was first described by Karatsuba and Ofman in 1962 in the "Doklady Akademii Nauk SSSR," the English translation was published in 1963 [KO63]. The original paper aimed on the application of multiplication in positional number systems. Knuth gives a compact version of the algorithm in the second volume of his "Art of Computer Programming" [Knu81]. In the treatment of the algorithm's history Knuth states that it seems surprising that the KOA had not been discovered before 1962, despite its comparatively simplicity and its usefulness, e.g. for mental arithmetic. Sedgewick [Sed90] also provides a description of the KOA in a compact notation, though slightly different from Knuth's. An investigation of the algorithm's computational complexity is given in [Fat74], where the KOA is referred to as "Split." However, this reference contains an error in the derivation of the additive complexity, leading to a somewhat incorrect formula[1].

The KOA is based on the "divide-and-conquer" principle [Sed90]. This principle is applied to suitable algorithms with complexity greater $\mathcal{O}(n)$ by splitting the initial prob-

[1] the error will be outlined in the following section

lem, solving the partial problems separately , and combining the solutions obtained. The "price" which must be paid for the computational gain is the splitting of the input and the merging of the partial solutions. Typical examples for divide-and-conquer algorithms are Quick-Sort [Sed90] for sorting or the Fast Fourier Transform (FFT) [Str86] with its wide applications, e.g. in signal processing [Bla85]. In the case of polynomial multiplication an algorithm is considered *efficient* if it saves multiplication, often at the cost of extra additions. As a consequence, multiplication must be more "costly" than addition if the algorithm is supposed to be an improvement. It should be noted that for the two general methods for efficient polynomial multiplication, KOA and FFT, the number of extra additions is often higher than the number of multiplications saved. This situation is given in particular for short polynomials as will be shown later.

5.2.2 Recursive Description and Complexity

First, the computational complexity of the "schoolbook" or straightforward method for polynomial multiplication is given, in order to provide a measure for the algorithm to be developed.

Theorem 8 *Two arbitrary polynomials in one variable of degree less or equal* $m-1$ *with coefficients from a field* $\mathcal{F}$ *can be multiplied with not more than:*

$$\#\otimes = m^2 \tag{5.2}$$
$$\#\oplus = (m-1)^2 \tag{5.3}$$

multiplications and additions, respectively, in $\mathcal{F}$*.*

The proof can be readily obtained by induction over m.

The KOA provides a recursive algorithm which reduces the multiplicative complexity (5.2) and — for large enough m — the additive complexity (5.3). We consider the multiplication of two polynomials $A(x)$ and $B(x)$ with a maximum degree of $m-1$ over a field $\mathcal{F}$, i.e. each polynomial possesses at most m coefficients from $\mathcal{F}$. We are interested in finding the product $C'(x) = A(x)B(x)$ with $deg(C'(x)) \leq 2m-2$. The consideration here is restricted to polynomials where m is a power of two: $m = 2^t$, t integer. To apply the algorithm, both polynomials are split into a lower and an upper half:

$$\begin{aligned} A &= x^{\frac{m}{2}}(x^{\frac{m}{2}-1}a_{m-1} + \cdots + a_{\frac{m}{2}}) + (x^{\frac{m}{2}-1}a_{\frac{m}{2}-1} + \cdots + a_0) = x^{\frac{m}{2}}A_h + A_l \\ B &= x^{\frac{m}{2}}(x^{\frac{m}{2}-1}b_{m-1} + \cdots + b_{\frac{m}{2}}) + (x^{\frac{m}{2}-1}b_{\frac{m}{2}-1} + \cdots + b_0) = x^{\frac{m}{2}}B_h + B_l. \end{aligned} \tag{5.4}$$

Using (5.4), a set of auxiliary polynomials $D^{(1)}(x)$ is defined:

$$\begin{aligned} D_0^{(1)}(x) &= A_l(x)B_l(x) \\ D_1^{(1)}(x) &= [A_l(x) + A_h(x)][B_l(x) + B_h(x)] \\ D_2^{(1)}(x) &= A_h(x)B_h(x). \end{aligned} \tag{5.5}$$

The product polynomial $C'(x) = A(x)B(x)$ is achieved by:

$$C'(x) = D_0^{(1)}(x) + x^{\frac{m}{2}}[D_1^{(1)}(x) - D_0^{(1)}(x) - D_2^{(1)}(x)] + x^m D_2^{(1)}(x). \tag{5.6}$$

Thus far the procedure has reduced the number of multiplications to $3/4m^2$ in (5.5) from m^2 in (5.2). However, the algorithm becomes recursive if it is applied again to the polynomial multiplications in (5.5). The next iteration step splits the polynomials $A_l, A_h,$ and$(A_l + A_h)$ and their B counterparts again in half. With the newly halved polynomials another set of auxiliary polynomials $D_i^{(2)}, i = 0, \ldots, 8$ is obtained. The polynomials $D^{(1)}$ can now be computed by means of the $D_i^{(2)}$:

$$\begin{aligned} D_0^{(1)}(x) &= D_0^{(2)}(x) + x^{\frac{m}{4}}[D_1^{(2)}(x) - D_0^{(2)}(x) - D_2^{(2)}(x)] + x^{\frac{m}{4}} D_2^{(2)}(x) \\ D_1^{(1)}(x) &= D_3^{(2)}(x) + x^{\frac{m}{4}}[D_4^{(2)}(x) - D_3^{(2)}(x) - D_5^{(2)}(x)] + x^{\frac{m}{4}} D_5^{(2)}(x) \\ D_2^{(1)}(x) &= D_6^{(2)}(x) + x^{\frac{m}{4}}[D_7^{(2)}(x) - D_6^{(2)}(x) - D_8^{(2)}(x)] + x^{\frac{m}{4}} D_8^{(2)}(x) \end{aligned} \tag{5.7}$$

The algorithm eventually terminates after t steps. In the final step the polynomials $D^{(t)}(x)$ are degenerated into single coefficients, i.e. $deg(D^{(t)}(x)) = 0$. Since every step exactly halves the number of coefficients, the algorithm terminates after $t = \log_2 m$ steps.

The following two theorems provide expressions for the computational and the time complexity of the KOA for polynomials over fields of characteristic 2 with respect to a parallel hardware implementation.

Theorem 9 *Two arbitrary polynomials in one variable of degree less or equal* $m-1$*, where* m *is a power of two, with coefficients in a field* $\mathcal{F}$ *of characteristic 2 can be multiplied by means of the Karatsuba-Ofman algorithm with:*

$$\#\otimes = m^{\log_2 3}, \tag{5.8}$$

$$\#\oplus \leq 6m^{\log_2 3} - 8m + 2, \tag{5.9}$$

multiplications and additions, respectively, in $\mathcal{F}$*.*

Theorem 10 *A parallel realization of the Karatsuba-Ofman algorithm for the multiplication of two arbitrary polynomials in one variable of degree less or equal* $m - 1$*, where* m *is a power of two, with coefficients in a field* $\mathcal{F}$ *of characteristic 2 can be implemented with a time complexity (or delay) of:*

$$T = T_\otimes + 3\,(\log_2 m)\,T_\oplus, \tag{5.10}$$

where "$T_\otimes$*" and "*$T_\oplus$*"denote the delay of one multiplier and one adder, respectively, in* $\mathcal{F}$*.*

It should be noted that the subtractions in (5.6) are additions if $\mathcal{F}$ is of characteristic 2. For the proof of the theorems three stages of the algorithm will be distinguished:

Proof.

1. In the first stage the mere splitting of the polynomials is considered. Since splitting itself takes no computation, only the two summations in 5.5 are of interest. Taking into account that the number of polynomials triples in each iteration step, whereas the length of the polynomials is reduced by half, one obtains:

$$\#\oplus_1 = \sum_{i=1}^{\log_2 m} 3^{i-1} 2\frac{m}{2^i} = 2m^{\log_2 3} - 2m. \tag{5.11}$$

   Since all additions of one iteration can be performed in parallel in a hardware realization, the delay equals:

$$T_1 = T_\oplus \log_2 m, \tag{5.12}$$

   where "$T_\oplus$" denotes the delay for one adder in $\mathcal{F}$.

2. In the second stage the achieved $3^{\log_2 m} = m^{\log_2 3}$ polynomials (each consisting of one coefficient) are actually multiplied. This requires:

$$\#\otimes_2 = m^{\log_2 3} \tag{5.13}$$

   multiplications. The delay of a parallel implementation is:

$$T_2 = T_\otimes, \tag{5.14}$$

   where "$T_\otimes$" denotes the delay caused by one multiplier in $\mathcal{F}$.

3. The third stage merges the polynomials according to Equation (5.6). There are two kinds of additions (or subtractions) involved: Subtracting three polynomials with $2^i - 1$ coefficients and $2^i - 2$ additions due to the overlapping[2] of three terms:

$$\#\oplus_3 = \sum_{i=1}^{\log_2 m} 3^{\log_2 m - i}[2(2^i - 1) + (2^i - 2)] = 4m^{\log_2 3} - 6m + 2. \tag{5.15}$$

   The delay equals:

$$T_3 = 2\,(\log_2 m)\,T_\oplus. \tag{5.16}$$

◇

The overall complexities in the Theorems 9 and 10 are obtained by summation of the partial complexities. However, the right hand side of the additive complexity (5.9) is an upper bound rather than an exact expression, because the recursive algorithm bears

[2]Reference [Fat74] is here wrong by claiming that only $2^i - 4$ coefficients overlap.

redundancies which can be eliminated in a parallel realization. For instance, for the value $m = 4$, which is of great importance for the multiplier architectures to be developed, the upper bound in (5.9) can be reduced from 24 to 22 in a parallel implementation such as sketched in Figure (5.1).

A comparison of the computational complexities (5.8) and (5.9) of the KOA with the corresponding expressions (5.2) and (5.3) of the straightforward approach shows an improvement for both, multiplication and addition. While the number of multiplications improves for all $m \geq 2$, the additive complexity only improves for $m \geq 64$.

> **Example.** Figure 5.1 shows a block diagram of a parallel realization of the KOA over fields with characteristic 2 for the case $m = 4$, i.e. the input polynomials have degree 3. The three different stages described above can be verified easily from the figure. The adders on the left hand side of the drawing refer to the two iteration steps of stage one. There are 10 additions required (5.11), the corresponding delay is in accordance to Equation (5.12) equal to $2T_{\oplus}$. The second stage corresponds to the row of 9 multipliers, causing a delay of one $T_{\otimes}$. The third stage is given by the adders on the right hand side of Figure 5.1. As mentioned above, the number of additions could be reduced from 14 in (5.13) to 12 due to redundancies. The critical path of length $4T_{\oplus}$ for the third stage is achieved by Equation (5.16) if $m = 4$.

5.2.3 A Matrix Representation

While the previous section describes the KOA as a recursive algorithm, this section provides a description based on binary matrices. The move from the recursive to a matrix representation can be viewed as a time-space transformation. The motivation for it originated in the need for a suitable description of the KOA for a VLSI synthesis, which will be described in Chapter 7, of the multiplier to be developed. However, the matrix representation is also useful for an investigation of the algorithm's properties. For instance, one obtains information regarding the connectivity structure of a parallel realization immediately from the binary matrices, whereas this structure is somewhat hidden by the recursive description.

The investigation is again restricted to polynomials over fields of characteristic 2. The structure of the matrix representation is the following. The three different stages of the KOA, introduced in section 5.2.2, are considered:

1. Stage one takes care of the splitting of the input polynomials. The two inputs $A(x), B(x)$ are represented by two vectors $\mathbf{A}_0, \mathbf{B}_0$, each consisting of m coefficients. Every step from the recursion is represented by one matrix-vector multiplication. Hence, there are $t = \log_2 m$ such multiplications:

step	1		2		...		$\log_2 m$
operation:	$\mathbf{M}_1\mathbf{A}_0 = \mathbf{A}_1$	$\rightarrow$	$\mathbf{M}_2\mathbf{A}_1 = \mathbf{A}_2$	$\rightarrow$	...	$\rightarrow$	$\mathbf{M}_t\mathbf{A}_{t-1} = \mathbf{A}_t$
operation:	$\mathbf{M}_1\mathbf{B}_0 = \mathbf{B}_1$	$\rightarrow$	$\mathbf{M}_2\mathbf{B}_1 = \mathbf{B}_2$	$\rightarrow$	...	$\rightarrow$	$\mathbf{M}_t\mathbf{B}_{t-1} = \mathbf{B}_t$

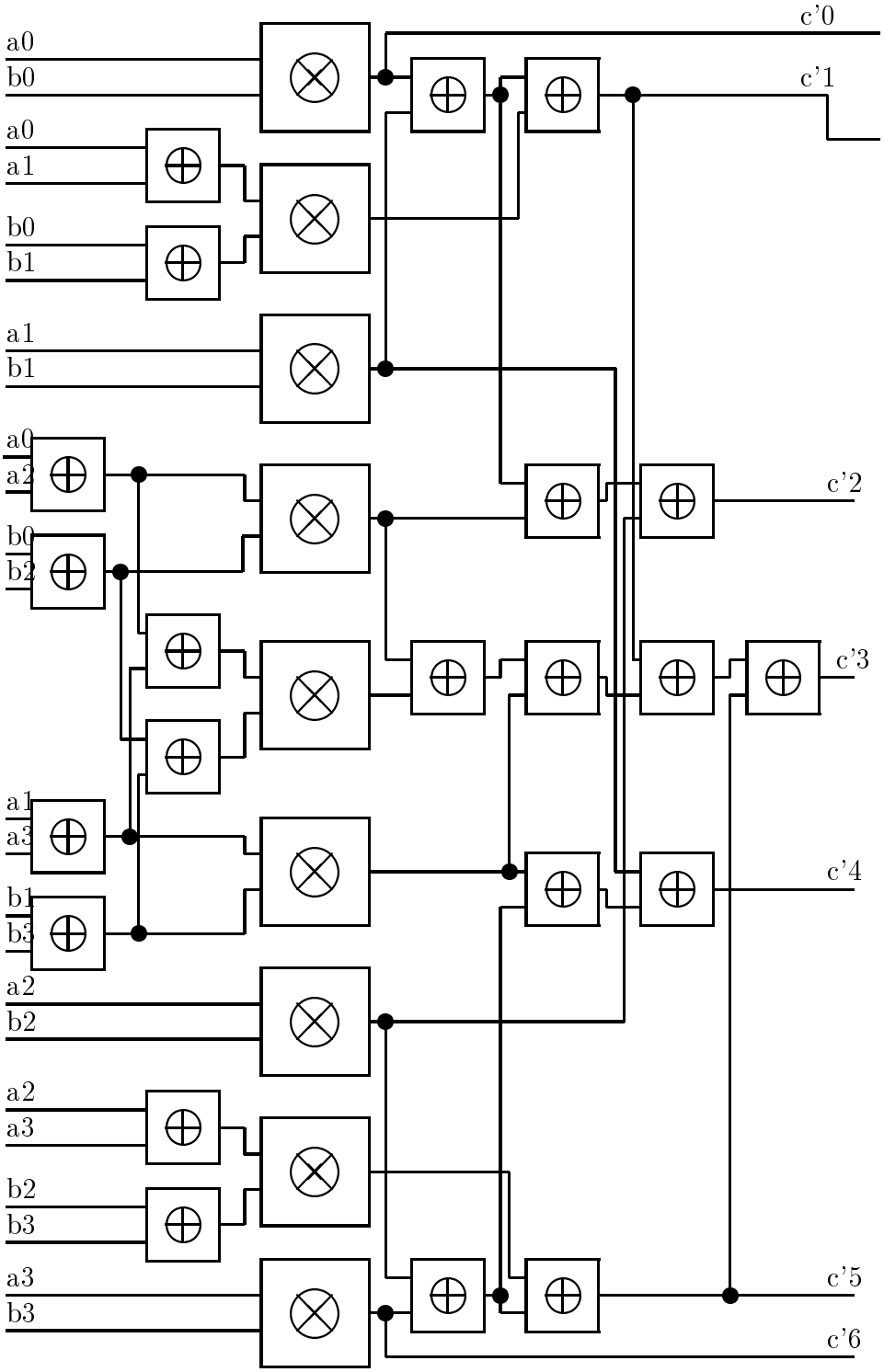

Figure 5.1: Block diagram of a parallel realization of the KOA for polynomials of degree 3 over fields with characteristic 2

The lengths of the vectors $\mathbf{A}_i, \mathbf{B}_i$ increases in every step by the factor 3/2. Since the initial vectors $\mathbf{A}_0, \mathbf{B}_0$ have m elements, the final vectors $\mathbf{A}_t, \mathbf{B}_t$ have $(3/2)^{\log_2 m} m = 3^{\log_2 m}$ elements. The one entries of the matrices $\mathbf{M}_i$ are obtained by the following algorithm in pseudo Pascal:

```
col0 := 0;
row0 := 0;
square := m 2^i;
FOR cnt := 0 TO 3^(i-1) DO  BEGIN
    FOR diag := 0 TO m/2^i DO  BEGIN
        M[diag+row0][diag+col0]=1;
        M[diag+square+row0][diag+col0]=1;
        M[diag+square+row0][diag+square+col0]=1;
        M[diag+2*square+row0][diag+square+col0]=1;
    END;
    row0 := row0 + 3 m/2^i;
    col0 := col0 + m/2^(i-1);
END;
```

2. Stage two performs element wise multiplication of the vectors $\mathbf{A}_t, \mathbf{B}_t$:

$$\mathbf{D}_0 = \mathbf{A}_t \odot \mathbf{B}_t$$

3. Stage three builds the polynomial $C'(x) = A(x)B(x)$ from the product vector $\mathbf{D}_0$. Similar to stage one, the $\log_2 m$ steps of the recursion are represented by $\log_2 m$ vector-matrix multiplications.

step	1		2		...		$\log_2 m$
operation:	$\mathbf{N}_1\mathbf{D}_0 = \mathbf{D}_1$	$\rightarrow$	$\mathbf{N}_2\mathbf{D}_1 = \mathbf{D}_2$	$\rightarrow$	...	$\rightarrow$	$\mathbf{N}_t\mathbf{D}_{t-1} = \mathbf{D}_t = C'$

The length of each vector $\mathbf{D}_i$ is given by $3^{\log_2 m - i}(2^{i+1} - 1), i = 0, 1, \ldots, \log_2 m$. The one entries of the matrices $\mathbf{N}_i$ are obtained by the following algorithm in pseudo Pascal:

```
FOR pset := 0 TO 3^log(m)-1 DO  BEGIN
   FOR cf := 0 TO 2^i-1 DO  BEGIN
      N[pset*(2^(i+1)-1)+cf][pset*(2^i-1)*3+cf]=1;
      N[pset*(2^(i+1)-1)+2^i+cf][pset*(2^i-1)*3+2*(2^i-1)+cf]=1;
      N[pset*(2^(i+1)-1)+2^(i-1)+cf][pset*(2^i-1)*3+2^i-1+cf]=1;
      N[pset*(2^(i+1)-1)+2^(i-1)+cf][pset*(2^i-1)*3+cf]=1;
      N[pset*(2^(i+1)-1)+2^(i-1)+cf][pset*(2^i-1)*3+2*(2^i-1)+cf]=1;
   END
END
```

In the sequel an instance of the matrix description is provided for polynomials with maximum degree 3.

Example. In the case $m = 4$, the initial vectors are $\mathbf{A}_0 = (a_0, a_1, a_2, a_3)$ and the $\mathbf{B}_0$ counterpart. Stage one and three are described by $t = \log_2 4 = 2$ matrix-vector multiplications. These are uniquely represented by the matrices $\mathbf{M}_1, \mathbf{M}_2, \mathbf{N}_1, \mathbf{N}_2$, which can be constructed by means of the two algorithms given above:

$$\mathbf{M}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad ; \quad \mathbf{M}_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{N}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad ; \quad \mathbf{N}_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

5.3 Efficient Polynomial Multiplication in Finite Fields

In this section the KOA is applied to efficient multiplication of polynomials $A(x), B(x)$ with maximum degree $m-1$ over fields $GF(2^n)$. This is the first and, with respect to the complexities, major step for performing the entire field multiplication (5.1) in $GF((2^n)^m)$. The goal is to minimize the number of elementary units, namely XOR- (mod 2 adder) and AND- (mod 2 multiplier) gates.

The two operations required for the KOA, addition and multiplication, refer now to arithmetic with the coefficients a_i, b_j in $GF(2^n)$. The module "$GF(2^n)$ adder" simply consists of n parallel mod 2 adders. For the module "$GF(2^n)$ multiplier" the multiplier from Mastrovito [Mas91] described in Section 3.1.1 is used. Assuming condition (3.6) for all generating polynomials $Q(y)$ of the ground field, the overall complexity for polynomial multiplication, measured in mod 2 adders/multipliers, results in:

$$\#\text{AND} \;=\; n^{2-\log_2 3}\; k^{\log_2 3} \tag{5.17}$$

$$\#\text{XOR} \;\leq\; \left(\frac{k}{n}\right)^{\log_2 3} (n^2 + 6n - 1) - 8k + 2n \;\;;\;\; \text{certain } n \tag{5.18}$$

with $k = nm$.

Both formulas (5.17), (5.18) imply that the *order* of elementary gates increases asymptotically only proportional to $k^{\log_2 3} \approx k^{1.58}$ as k increases if n is kept small. However, given a field order 2^k, the optimum value for the field partitioning $GF((2^n)^m) \cong GF(2^k)$ must be determined. Having k fixed, the optimum of Equation (5.17) is simply obtained if n is chosen as small as possible. The optimum for the XOR complexity is achieved by computing the root of the first derivative of (5.18) with respect to n. The derivative is a polynomial with irrational exponents without a closed solution. For actual values of k the solution can be obtained numerically. By using Wolfram's *Mathematica* [Wol88], the optimum value was found to be $n = 8$ for all fields with $k > 32$. For $k \leq 32$ the optimum n is between 5 ($k = 8$) and 7 ($k = 32$). However, this optimization does not take into account that m has to be a power of two. Hence, for actual field size exponents k a trade off must be made between a *possible* parameter n, usually close to the optimum one described above, which results in a reasonably low XOR complexity, and a value for n which keeps the AND complexity low.

Row six and seven of Table 5.1 in Section 5.5, headed by AND and XOR show the complexities (5.17) and (5.18), respectively, found for combinations of the form:

$$k = nm \quad \text{where} \quad k = 4, 6, \ldots, 32 \;\; ; \;\; m = 2^i.$$

To achieve an expression for the time complexity, Equation (5.10) with appropriate expressions for $T_\oplus$ and $T_\otimes$ can be applied. As mentioned before, addition in $GF(2^n)$ has a delay of one XOR gate, i.e. $T_\oplus = \mathcal{T}_{\text{xor}}$. The delay for multiplication, $T_\otimes$, in the ground field $GF(2^n)$ is upper bounded by (3.11). Hence, the overall delay for parallel multiplication of polynomials of degree $m - 1$ over $GF(2^n)$ can be upper bounded by:

$$T \leq \mathcal{T}_{\text{xor}}(2\lceil \log_2 n \rceil + 3 \log_2 m) + \mathcal{T}_{\text{and}}. \tag{5.19}$$

5.4 Reduction Modulo the Field Polynomial

This section describes the second step of (5.1), the operation "mod $P(x)$." In order to perform this operation with low complexity, it is assumed that the field polynomial can be chosen arbitrarily. From a mathematical point of view this assumption is valid anyway, since there exists only *one* field of order 2^{nm} [McE87]; different representations of fields generated by different field polynomials are always isomorphic. From a technical point of view, a situation may arise where one architecture operating on a certain polynomial has to be used in a system in which modules based on another field polynomial exist. In this case, merely an isomorphic mapping as described in Section 2.2 has to be implemented in order to fit the different representations.

The pure polynomial multiplication of two polynomials $A(x)B(x)$, both of degree $m - 1$, results in a product polynomial $C'(x)$ over $GF(2^n)$ with $deg(C'(x)) \leq 2m - 2$. In order to perform a multiplication in $GF((2^n)^m)$, $C'(x)$ must be reduced modulo the

generator polynomial $P(x)$. The modulo operation will result in a polynomial $C(x)$ with $deg(C(x)) \leq m-1$ which represents the desired field element:

$$\begin{aligned} C(x) &= C'(x) \bmod P(x) && ; \; C(x) \in GF((2^n)^m) \\ &= c'_{2m-2}x^{2m-2} + \cdots + c'_0 && \bmod P(x) \\ &= c_{m-1}x^{m-1} + \cdots + c_0 \end{aligned}$$

The reduction modulo $P(x)$ can be viewed as a linear mapping of the $2m-1$ coefficients of $C'(x)$ into the m coefficients of $C(x)$. This mapping can be represented in a matrix notation as follows:

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & r_{0,0} & \cdots & r_{0,m-2} \\ 0 & 1 & \cdots & 0 & r_{1,0} & \cdots & r_{1,m-2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & r_{m-1,0} & \cdots & r_{m-1,m-2} \end{pmatrix} \begin{pmatrix} c'_0 \\ \vdots \\ c'_{m-1} \\ c'_m \\ \vdots \\ c'_{2m-2} \end{pmatrix} \tag{5.20}$$

The matrix on the right hand side of (5.20) consists of a (m, m) identity matrix and a $(m, m-1)$ matrix $\mathbf{R}$ which we may name the *reduction matrix*. $\mathbf{R}$ is solely a function of the chosen monic generating polynomial $P(x) = x^m + \cdots + p_0$, i.e. to every $P(x)$ a reduction matrix is uniquely assigned. $\mathbf{R}$'s recursive dependency on $P(x)$ is the following:

$$r_{ji} = \begin{cases} p_i & ; \; j = 0, \ldots, m-1 \; ; \; i = 0 \\ r_{j-1,i-1} + r_{m-1,i-1}r_{j0} & ; \; j = 0, \ldots, m-1 \; ; \; i = 1, \ldots, m-2 \end{cases} \tag{5.21}$$

where $r_{j-1,i-1} = 0$ if $j = 0$. From Equation (5.21) it follows directly that $r_{ji} \in GF(2^n)$ since $p_i \in GF(2^n)$. It should be emphasized that (5.20) does not require any general multiplication but only additions and multiplications with a constant from $GF(2^n)$. Both functions require only mod 2 adders as is shown in Section 4.1. Therefore the space complexity of a realization of (5.20) can be measured by the total number of two input mod 2 adders. In order to achieve a small complexity for the reduction mod$P(x)$, an exhaustive computer based search through *all* primitive polynomials over $GF(2^n)$ of degree m was conducted. The number of primitive polynomials I_p checked is given by [GT74]:

$$I_p = \frac{1}{m}\Phi(2^{mn} - 1),$$

where $\Phi(\cdot)$ denotes the Euler function.

The complexity of multiplication with each of the I_p reduction matrices was evaluated as follows. For every matrix the number of additions and constant multiplications was computed. Redundancies within the rows of $\mathbf{R}$, i.e. at least two elements are equal: $r_{ij} = r_{ik}$, were taken into account, thus reducing the number of constant multiplications. Each addition has a weight of n mod 2 adders, the weight for constant multiplication was achieved by the optimization algorithm described in Section 4.1.

Example. The polynomial investigated is $P(x) = x^4 + x^3 + x + \omega$ over $GF(2^n)$. The corresponding reduction matrix is:

$$\mathbf{R} = \begin{pmatrix} \omega & \omega & 0 \\ 0 & \omega & \omega \\ 1 & 1 & \omega \\ 1 & 1 & 1 \end{pmatrix}.$$

The operation (5.20) is for this instance:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} c'_0 + \omega(c'_4 + c'_5) \\ c'_1 + \omega(c'_5 + c'_6) \\ c'_2 + c'_4 + c'_5 + \omega c'_6 \\ c'_3 + c'_4 + c'_6 \end{pmatrix},$$

which can be computed with 9 additions and 3 constant multiplications with ω. These operations can be implemented with $9 \cdot 4 + 3 \cdot 1 = 39$ XOR gates.

The best polynomials P_{opt} found during the search are only suboptimum. Redundancies between rows are not found by the algorithm implemented. For instance, reconsidering the example from above, the expression $(c'_4 + c'_5)$ occurs twice, so that the modulo reduction can be realized with $8 \cdot 4 + 3 \cdot 1 = 35$ XOR gates. However, the finding of *globally optimum* expressions seems to be a problem which is difficult and computational intensive. Hence, the determination of globally optimum solutions was not feasible for this application where, due to the nature of the exhaustive search, a rather large number of polynomials have to be checked. Moreover, as the results in Section 5.5 show, the modulo reduction was found to be of minor importance in terms of complexity for the multiplier, since it is responsible for less than 10% of the overall gate count for the fields investigated.

The exhaustive search was implemented as follows. The search algorithm consists of two main parts: evaluation of the complexity as described above for every polynomial $P(x)$ and, if the complexity is an improvement on the previously found ones, checking whether $P(x)$ is actually primitive. The pseudo code below describes the structure of algorithm.

```
p = x^m + 1;
bestcmpl = maxvalue;
FOR i=1 TO 2^(mn)-1 DO BEGIN
    r = GET_REDUCTION_MATRIX(p);
    cmpl = GET_COMPLEXITY(r);
    IF (cmpl <= bestcmpl) BEGIN
       IF (PRIME_TEST(p) = 1) BEGIN
          pbest = p;
          bestcmpl = cmpl;
       END;
    END;
```

```
    p = GET_NEXT_POLYNOMIAL(p);
END;
```

The function `GET_NEXT_POLYNOMIAL()` produces all $2^{nm} - 1$ possible monic polynomials in a counter like way: $P_1(x) = x^m + 1, P_2(x) = x^m + \omega, P_3(x) = x^m + \omega^2, \ldots$. The functions `GET_REDUCTION_MATRIX()` and `GET_COMPLEXITY()` apply Formula (5.21) and the evaluation procedure from above, respectively. The most sophisticated task, testing if a passed polynomial is primitive, is performed by `PRIME_TEST()`, the algorithm for which was developed in Section 2.2.

5.5 Results

5.5.1 Space and Time Complexities

In this section the overall space and time complexity of multipliers in the composite fields $GF((2^n)^m)$, where $k = nm \leq 32$, are provided. The best results achieved with values $m = 4, 8$ are listed. Although a choice of $m = 2$ is possible, in the next chapter another architecture will be introduced which provides a lower complexity and which is also simpler for this value.

The complexities are achieved by summing the partial complexities of the polynomial multiplication and of the modulo reduction, which were developed in the two previous sections. Table 5.1 gives a detailed insight in the space complexities and architectures of the parallel multipliers. For each field a generating polynomial $P(x)$ and a multiplier with a minimum complexity is given. A description of the table's contents is given below. All columns are explained from left to right, where the columns are named after their heading symbols.

k, n, m**:** k denotes the field order 2^k, where the parameters n and m determine the composition $GF((2^n)^m)$ of the field. The binary generating polynomials $Q(y)$ of the ground fields $GF(2^n)$ are listed in Table 3.1.

$P(x)$**:** Primitive polynomials over $GF(2^n)$ are given which possess minimum complexity with respect to the operation "mod$P(x)$." The character ω denotes a primitive element of the field $GF(2^n)$, such that $Q(\omega) = 0$.

$\#\otimes$**,** $\#\oplus$**:** The number of multiplications/additions is given for the pure multiplication of two polynomials of degree $m-1$ with the KOA. They refer to the formulas (5.17) and (5.18), respectively.

AND**,** XOR**:** The space complexity for the pure multiplication of two polynomials over the field $GF(2^n)$ is given in multiples of elementary gates.

mod**:** The space complexity for the operation mod$P(x)$ is given.

				$A(x) \times B(x)$				mod	$A\ B \bmod P$		
k	n	m	$P(x)$	$\#\otimes$	$\#\oplus$	AND	XOR	XOR	AND	XOR	k^2
8	2	4	$111\omega\omega$	9	22	36	71	20	**36**	**91**	64
12	3	4	$1001\omega^6$	9	22	81	138	21	**81**	**159**	144
16	4	4	1110ω	9	22	144	223	35	**144**	**258**	256
20	5	4	$100\omega\omega$	9	22	225	326	34	**225**	**360**	400
24	6	4	$1\omega^{62}\ \omega^{61}\ \omega^3\ \omega^2$	9	22	324	447	60	**324**	**507**	576
	3	8	$11111110\omega^6$	27	100	243	516	82	**243**	**598**	576
28	7	4	$100\omega^{126}\ \omega^{126}$	9	22	441	586	46	**441**	**632**	784
32	4	8	10010010ω	27	100	432	805	91	**432**	**896**	1024

Table 5.1: Composite fields $GF((2^n)^m)$ up to $GF(2^{32})$, their generating polynomials and the space complexities for parallel multipliers

$AB \bmod P$**:** The overall space complexity for a parallel multiplier in $GF((2^n)^m)$ is given in bold face letters. It is achieved by summing the complexities for the pure polynomial multiplication and the modulo reduction.

k^2**:** The complexity of many traditional architectures is lower bounded by $k^2 - 1$ XOR gates and k^2 AND gates. In order to allow comparison with other multipliers, we provide the values k^2.

Table 5.2 contains the theoretical delays of the multipliers. The time complexities are given as multiples of AND gate delays and XOR gate delays, denoted $\mathcal{T}_{\text{and}}$ and $\mathcal{T}_{\text{xor}}$, respectively. The structure is similar to Table 5.1. All columns will be described in the following:

k, n, m**:** k denotes the field order 2^k, where the parameters n, m determines the composition $GF((2^n)^m)$ of the field.

$A(x) \times B(x)$**:** The delays for the pure polynomial multiplication with the KOA are listed. The entries in these columns are achieved through Equation (5.19), where $T_\oplus = \mathcal{T}_{\text{xor}}$ and the actual delays for $T_\otimes$ were taken from Table 3.1.

mod**:** This column contains the time complexity for the operation mod$P(x)$.

$A\ B \bmod P$**:** The delay of the entire multiplier is shown in bold face letters. It is achieved by summing the time complexities for the pure polynomial multiplication and the modulo reduction.

However, delays caused by routing or high fan outs which may occur in an actual VLSI implementation are not considered.

In the following an example for a multiplier in the field $GF(2^{16})$ is described.

			$A(x) \times B(x)$		mod	$A\ B$ mod P	
k	n	m	$\mathcal{T}_{\text{and}}$	$\mathcal{T}_{\text{xor}}$	$\mathcal{T}_{\text{xor}}$	$\mathcal{T}_{\text{and}}$	$\mathcal{T}_{\text{xor}}$
8	2	4	1	8	3	**1**	**11**
12	3	4	1	9	2	**1**	**11**
16	4	4	1	9	3	**1**	**12**
20	5	4	1	11	3	**1**	**14**
24	6	4	1	10	4	**1**	**14**
	3	8	1	12	3	**1**	**15**
28	7	4	1	10	3	**1**	**13**
32	4	8	1	12	3	**1**	**15**

Table 5.2: Theoretical time complexity of multipliers over composite fields $GF((2^n)^m)$ up to $GF(2^{32})$

Example. The field considered is $GF((2^4)^4)$. The generating polynomial of the ground field $GF(2^4)$ is $Q(y) = y^4 + y + 1$. The composite field is generated by $P(x) = x^4 + x^3 + x^2 + \omega$, where $Q(\omega) = 0$. There are 3 multiplications and 4 additions required for the pure polynomial multiplication. This corresponds to 144 mod 2 multipliers and 223 mod 2 adders if the ground field multiplier from Table (3.1) with $n = 4$ is used. The operation modulo $P(x)$ requires 35 mod 2 adders. Hence, the complexity for a parallel multiplier in the composite field results in 144 mod 2 multipliers and 258 mod 2 adders. The delay of the multiplier is achieved in a similar way. The Karatsuba algorithm causes a delay of 9 $\mathcal{T}_{\text{xor}}$ plus one $\mathcal{T}_{\text{and}}$. The circuit for the reduction mod $P(x)$ causes another delay of 3 $\mathcal{T}_{\text{xor}}$, resulting in an overall delay of 12 $\mathcal{T}_{\text{xor}}$ and one $\mathcal{T}_{\text{and}}$. Figure 5.2 provides a block diagram of the multiplier's architecture. The input variables are $a_0, \ldots, a_3$ and $b_0, \ldots, b_3$, the output variables are $c_0, \ldots, c_3$. Each set of variables represents a polynomial, which is an element in $GF((2^4)^4)$ in standard representation. Each variable is actually a four bit wide bus, representing an element in the ground field. As intermediate variables the coefficients c_i' are included, which are the output of the polynomial multiplication module and the input of the module providing reduction mod $P(x)$. The blocks having an "ω" attached are multipliers with the constant element ω. As will be described in Chapter 9, the multiplier was actually implemented on an FPGA XC3142 from Xilinx. The multiplier served as a coprocessor for a digital signal processor.

5.5.2 Discussion

Table 5.1 shows that the introduction of composite fields $GF((2^n)^m) \cong GF(2^k)$ leads to significantly improved parallel multipliers with respect to the number of mod 2 adders and multipliers if compared to traditional architectures such as introduced in Section 3.1 or [HWB92b] [IT89]. Moreover, the multiplier has also a lower gate count for all fields

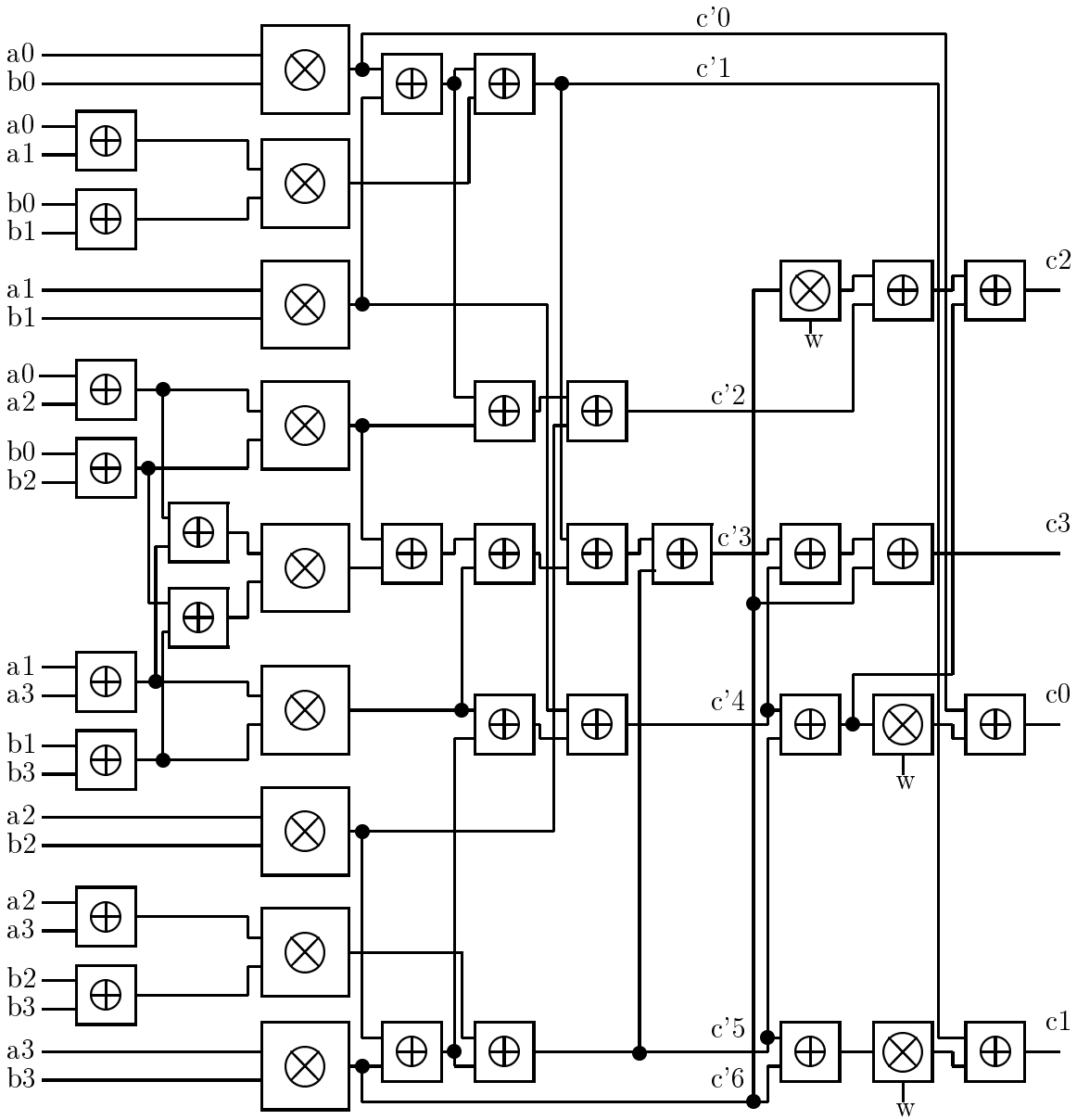

Figure 5.2: Block Diagram of a parallel multiplier in $GF((2^4)^4)$

considered than the architecture proposed in [Afa90], which applies the KOA to binary polynomials and is described in the Subsection 3.2.1.

The multiplication of two polynomials, which is the most costly step in standard based Galois field multiplication, can be performed with an asymptotical complexity of $O(k^{\log_2 3})$. The number of mod 2 adders (XOR) is improved for all fields. The number of mod 2 multipliers (AND) is improved for most fields considered. Asymptotically, the pure polynomial multiplication can be performed with $k^{\log_2 3}$ XOR and AND gates.

The complexity for the reduction modulo $P(x)$ is much smaller than the one for polynomial multiplication. For all fields considered in Table 5.1, the number of gates for modulo reduction takes less than 10% of the overall gate count for the entire field multiplication. The best field polynomials $P(x)$ found have only a few coefficients other than zero or one. All of these nontrivial coefficients possess a relatively low multiplicative complexity. In accordance to the observation of the optimization algorithm for constant multiplication, these coefficients are all among the first few or last few coefficients of the ground field $GF(2^n)$. The time complexity of the modulo reduction for the fields considered — given in Table 5.2 in the column headed by "mod" — possesses the somewhat surprising property that it is almost independent of the field size for the cases $m = 4, 8$. The greatest delay found for these values of m is 4 XOR delays, the smallest one 2 XOR delays.

From a VLSI design point of view multiplication over composite fields possesses a couple of natural advantages, namely hierarchy, modularity and — to some extent — regularity [WE92]. These properties become obvious by considering Figure 5.2. It is clear that the multiplier is divided into submodules thus assuring hierarchy. The major advantage is the high regularity, since one deals only with three types of *identical* modules, performing addition, multiplication, and constant multiplication in the ground field $GF(2^n)$. Third, the architecture is highly modular, because there are only a relatively small number of modules with *well defined* functions and interfaces. Another requirement often associated with regularity, a structure which allows a array implementation, is not naturally fulfilled by the architecture. However, as the comparative VLSI syntheses in Chapter 7 shows, the theoretical low gate count of the architecture can be used in actual gate array implementations.

Chapter 6

Multipliers over Fields with Certain Composition

In this chapter multiplier architectures for the two specific types of composite fields $GF((2^n)^2)$ and $GF((2^n)^4)$ are introduced. They differ from the general architecture over $GF((2^n)^m)$ described in the previous chapter, in that they combine the two parts of the standard base Galois field multiplication, polynomial multiplication and reduction modulo $P(x)$. For certain choices of n, in particular $n \leq 7$, the approach introduced in this chapter results in lower space and time complexities than with the general architecture.

6.1 Multipliers over $GF((2^n)^2)$

Parts of this section were presented in [Paa93a] and [Paa93b].

6.1.1 Architecture and Complexity

generalization of the architecture to be proposed in this section for multiple extension fields was previously described by Afanasyev in [Afa91]. Section 3.2.2 of this thesis contains a description of Afanasyev's architecture. However, we will provide a complete table of optimized primitive polynomials for values $n \leq 16$, together with the expected space and time complexities of multipliers in the fields $GF((2^n)^2)$.

The architecture is based on certain primitive polynomials of degree two, whose existence is given by the following theorem:

Theorem 11 *Given a ground field $GF(2^n)$, there exists always a primitive polynomial of the form*

$$P(x) = x^2 + x + p_0 \ , \ p_0 \in GF(2^n),$$

which generates the composite field $GF((2^n)^2)$.

Proof. Considering a primitive root α and its conjugate α^{2^n}, both in $GF((2^n)^2)$, a primitive polynomial can be constructed by means of its two linear factors:

$$P(x) = (x + \alpha)(x + \alpha^{2^n}) = x^2 + (\alpha + \alpha^{2^n})x + \alpha^{2^n+1}.$$

Thus, the polynomial coefficients are $p_1 = \alpha + \alpha^{2^n}$ and $p_0 = \alpha^{2^n+1}$. Now p_1 is the trace of α relative to $GF(2^n)$. It must be shown that there exists a primitive element α such that $p_1 = \alpha + \alpha^{2^n} = 1$. This problem is covered by a conjecture of Golomb [Gol84] which was later resolved in the affirmative by Moreno [Mor89]. $\diamond$

In the sequel we apply the KOA but avoid its full implementation by using a polynomial as introduced above for the reduction modulo $P(x)$.

Each field element can be represented as a polynomial with degree ≤ 1. Application of the KOA to the pure polynomial multiplication of two elements $A(x), B(x) \in GF((2^n)^2)$ results in:

$$\begin{aligned} C'(x) &= (a_1 x + a_0)(b_1 x + b_0) \\ &= a_0 b_0 + x([a_1 + a_0][b_1 + b_0] + a_0 b_0 + a_1 b_1) + x^2 a_1 b_1. \end{aligned} \tag{6.1}$$

The result from the reduction $C'(x) \bmod P(x)$ is the product field element $C(x) = A(x)B(x) \bmod P(x)$. Since $x^2 = x + p_0$, if a field polynomial as introduced in Theorem 11 is chosen, $C(x)$ is given by:

$$\begin{aligned} C(x) &= C'(x) \bmod P(x) \\ &= (a_0 b_0 + p_0 a_1 b_1) + x([a_1 + a_0][b_1 + b_0] + a_0 b_0). \end{aligned} \tag{6.2}$$

The computational complexity of (6.2) is:

$$\begin{aligned} \#\otimes &= 3, \\ \#\oplus &= 4, \\ \#\otimes_{p_0} &= 1, \end{aligned}$$

where $\otimes_{p_0}$ denotes constant multiplication by p_0. All operations refer to arithmetic in $GF(2^n)$. Figure 6.1 shows a schematic of a hardware realization of Equation (6.2).

Again, we apply the Mastrovito multiplier to the ground field multiplication. Assuming a complexity of n^2 AND / $(n^2 - 1)$ XOR gates of the ground field multiplier, the space complexity of the multiplier in the composite field is:

$$\#\text{AND} = \frac{3}{4}k^2, \tag{6.3}$$

$$\#\text{XOR} = \frac{3}{4}k^2 + 2k - 3 + C_{\otimes_{p_0}}\text{, certain n}, \tag{6.4}$$

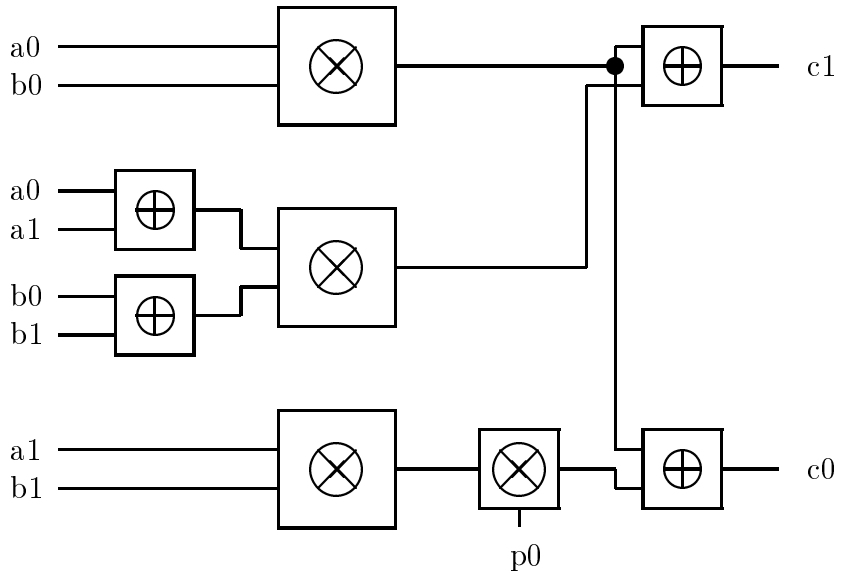

Figure 6.1: Block diagram of a parallel multiplier in $GF((2^n)^2)$

where $C_{\otimes_{p_0}}$ denotes the complexity (in XOR gates) of constant multiplication with the coefficient p_0 of the field polynomial $P(x)$. It should be noted that the XOR complexity in (6.4) is higher than stated, if the ground field multiplier requires more than $(n^2 - 1)$ XOR gates.

Expressions for the time complexity are achieved by considering the critical path in Figure 6.1. The maximal delay is composed of the delays for one general multiplication $a_1 b_1$, one constant multiplication $p_0(a_1 b_1)$, and one addition $(p_0 a_1 b_1) + (a_0 b_0)$. Using the upper bound from Equation (3.8) as the delay of the general multiplication, we obtain:

$$\#\mathcal{T}_{\text{and}} = 1, \tag{6.5}$$

$$\#\mathcal{T}_{\text{xor}} = 2\lceil \log_2 n \rceil + 1 + T_{\otimes_{p_0}}, \tag{6.6}$$

where $T_{\otimes_{p_0}}$ denotes the delay caused by the multiplication with p_0.

6.1.2 Results

As the Formulas (6.3) and (6.4) imply, the space complexity for a given field $GF(2^k)$ (i.e. k is fixed) depends solely on the coefficient p_0. Hence we performed an exhaustive search for primitive polynomials of the form $P(x) = x^2 + x + p_0$. The polynomials with the lowest complexity for constant multiplication with p_0 are considered optimum ones.

k	n	$P(x)$	$C_{\otimes_{p0}}$ XOR	$AB \bmod P$ AND	$AB \bmod P$ XOR	k^2	$AB \bmod P$ $\mathcal{T}_{\text{and}}$	$AB \bmod P$ $\mathcal{T}_{\text{xor}}$
4	2	$11\omega^{2}$	1	**12**	**18**	16	**1**	**4**
6	3	$11\omega^{6}$	1	**27**	**37**	36	**1**	**5**
8	4	$11\omega^{14}$	1	**48**	**62**	64	**1**	**5**
10	5	$11\omega^{3}$	3	**75**	**95**	100	**1**	**7**
12	6	$11\omega^{62}$	1	**108**	**130**	144	**1**	**6**
14	7	$11\omega^{124}$	3	**147**	**175**	196	**1**	**8**
16	8	$11\omega^{217}$	8	**192**	**292**	256	**1**	**9**
18	9	$11\omega^{5}$	5	**243**	**281**	324	**1**	**8**
20	10	$11\omega^{7}$	7	**300**	**344**	400	**1**	**8**
22	11	$11\omega^{2036}$	11	**363**	**415**	484	**1**	**12**
24	12	$11\omega^{4094}$	3	**432**	**672**	576	**1**	**9**
26	13	$11\omega^{8188}$	7	**507**	**665**	676	**1**	**10**
28	14	$11\omega^{5}$	12	**588**	**833**	784	**1**	**10**
30	15	$11\omega^{32766}$	1	**675**	**733**	900	**1**	**7**
32	16	$11\omega^{16948}$	16	**768**	**923**	1024	**1**	**9**

Table 6.1: Space and time complexities for multipliers in $GF((2^n)^2)$

Table 6.1 provides information regarding the space and time complexity of multipliers in the fields $GF((2^n)^2)$, $n = 2, 3, \ldots, 16$. The two leftmost columns list the field order exponents n and k, where $k = 2n$. For the ground field, the Mastrovito multiplier with the actual complexities and field polynomials $Q(y)$ as listed in Table 3.1 is assumed. The column headed by $P(x)$ contains the best primitive polynomials found by the exhaustive search. As throughout this thesis, the primitive root of the ground field polynomial $Q(y)$ is denoted as ω, such that $Q(\omega) = 0$. The symbol $C_{\otimes_{p0}}$ heads the column containing the complexities for multiplication with the coefficient p_0. The complexities were optimized with the greedy algorithm described in Section 4.1.1. The next two columns contain (in bold face letters) the overall gate count of the proposed multipliers. They refer to the Formulas (6.3) and (6.4), respectively. In order to compare the complexities with the lower bound of traditional architectures, the values k^2 are listed in the next two columns. The two rightmost columns contain the time complexities in multiples of AND and XOR gate delays.

In the sequel an example for a multiplier in the important field $GF(2^8)$ is given.

Example. As can be seen in Figure 6.1, a multiplier in the field $GF(2^8)$ is composed of 3 multipliers, 4 adders, and 1 multiplier with the constant ω^{14} from the ground field. All arithmetic is done in the field $GF(2^4)$, thus all connections are actually 4 bit wide busses. Multiplication with ω^{14} is described by the product

matrix:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

and can be realized with one XOR gate. Addition in $GF(2^4)$ requires 4 XOR gates. Since the Mastrovito multiplier in $GF(2^4)$ can be implemented with 16 AND and 15 XOR gates, the space complexity of the multiplier in $GF(2^8)$ is $3 \cdot 16 = 48$ AND gates and $3 \cdot 15 + 4 \cdot 4 + 1 = 62$ XOR gates. The time complexity is achieved by summation of the delays along the critical path: one adder ($1\ \mathcal{T}_{\text{xor}}$), one multiplier ($1\ \mathcal{T}_{\text{and}} + 3\ \mathcal{T}_{\text{xor}}$), and one constant multiplication ($1\ \mathcal{T}_{\text{xor}}$). Hence, the overall delay equals $1\ \mathcal{T}_{\text{and}} + 5\ \mathcal{T}_{\text{xor}}$.

6.1.3 Evaluation

The number of AND gates is for all fields $(3/4)\,k^2$ and thus 25% lower than the lower complexity bound of traditional architectures. The XOR complexity is in all cases higher than the AND complexity, although the number of XOR gates is for most fields considered below k^2. As a matter of fact, the number of XOR gates is considerably higher than k^2 only for the values $k = 16, 24, 28$. This effect is due to the rather high gate count of the corresponding ground field multipliers for values $n = 8, 12, 14$. However, for the values $k \leq 14$ the multiplier performs best in terms of space complexity compared to the other architectures proposed in this thesis. To the author's knowledge, the gate count for the important field $GF((2^4)^2) \cong GF(2^8)$ of 48 AND / 62 XOR gates is the best one reported in technical literature. For values $k \geq 16$ with $4|k$, the architecture operating on general composite fields $GF((2^n)^m)$ as proposed in Chapter 5 has lower gate counts. However, for all fields with k even but $4 \not| k$, the composition $GF((2^n)^2) \cong GF(2^k)$ is the only possible one under the condition that $k = n2^i$, i integer. Moreover, the routing required for this architecture is less than the one for multipliers with parameters $m \geq 4$, such that in actual VLSI implementations a field composition $GF((2^n)^2)$ might be of advantage, even for fields with $k \geq 16$.

The general advantage of multipliers over composite fields — modularity and regularity — as described in Section 5.5.2 are valid for this architecture too.

6.2 Multipliers over $GF((2^n)^4)$

Parts of the results of this section were presented in [Paa94].

This section introduces parallel multipliers over fields $GF((2^n)^4)$. As the architecture in the previous section, the approach here also combines the KOA with the reduction modulo the field polynomial $P(x)$. However, the multiplier here will reveal more clearly the principal behind both architectures. This is that a clever choice of $P(x)$ leads in conjunction with the KOA to expressions that take advantage of the fact that the field characteristic is two, i.e. we obtain expressions of the form $a + a$ which are zero and have thus not to be computed (and implemented.)

6.2.1 Architecture and Complexity

The elements $A(x), B(x)$ of the composite fields $GF((2^n)^4)$ are represented by polynomials with degree ≤ 3. First, we reconsider the KOA applied to the pure polynomial multiplication $A(x)B(x) = C'(x)$, as shown in Figure 5.1. We denote the outputs of the nine multipliers with $d_0, d_1, \ldots, d_8$, with d_0 being the top one, as can be seen in Figure 6.2.

These newly introduced intermediate variables $d_i \in GF(2^n)$ are produced by the first and the second stage of the KOA. They are obtained from the inputs through:

$$\begin{aligned}
d_0 &= a_0 b_0 \\
d_1 &= (a_0 + a_1)(b_0 + b_1) \\
d_2 &= a_1 b_1 \\
d_3 &= (a_0 + a_2)(b_0 + b_2) \\
d_4 &= (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3) \\
d_5 &= (a_1 + a_3)(b_1 + b_3) \\
d_6 &= a_2 b_2 \\
d_7 &= (a_2 + a_3)(b_2 + b_3) \\
d_8 &= a_3 b_3.
\end{aligned} \tag{6.7}$$

The third stage of the KOA constructs the product polynomial $C'(x)$ of degree ≤ 6 from the variables d_i through the following set of equations:

$$\begin{aligned}
c'_0 &= d_0 \\
c'_1 &= d_0 + d_1 + d_2 \\
c'_2 &= d_0 + d_2 + d_3 + d_6 \\
c'_3 &= d_0 + d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7 + d_8 \\
c'_4 &= d_2 + d_5 + d_6 + d_8 \\
c'_5 &= d_6 + d_7 + d_8 \\
c'_6 &= d_8.
\end{aligned} \tag{6.8}$$

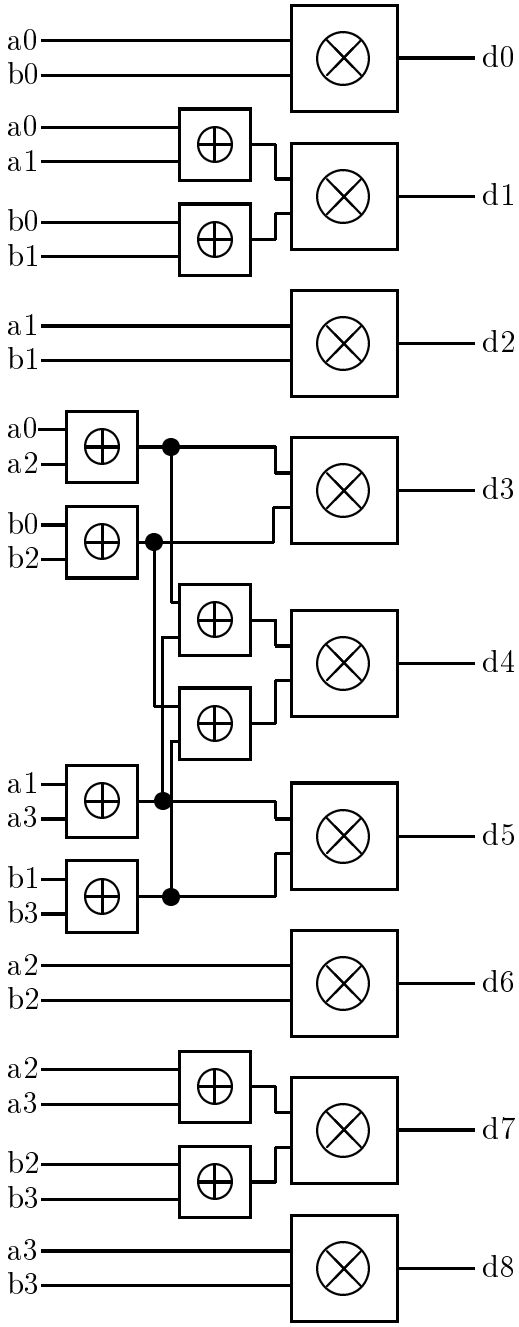

Figure 6.2: Block diagram of the first two stages of the KOA for polynomials of degree 3

In order to perform finite field multiplication, the polynomial $C'(x)$ has to be reduced modulo the field polynomial: $C(x) = C'(x) \bmod P(x)$. In Section 5.4 it was shown that the coefficients c_i are linear combinations of the coefficients c'_i. The linear combination is uniquely determined by the field polynomial $P(x)$ (Equation (5.21).) Since the coefficients c'_i are sums of the multiplier outputs d_i (Equations (6.7) above,) the coefficients c_i are also linear combinations of the d_i. Moreover, since the characteristic of the ground field $GF(2^n)$ is two, the sum of two identical variables is zero:

$$d_i + d_i = 0, \text{ for } i = 0, 1, \ldots, 8. \tag{6.9}$$

The key idea of the architecture is to find field polynomials which lead to (many) expressions of the form (6.9), thus reducing the number of additions required and improving the delay. It will be shown that the following two types of polynomials possess this property:

Definition 15 *Irreducible polynomials over $GF(2^n)$ of degree four (which generate the fields $GF((2^n)^4)$) of the following form:*

$$P_I(x) \;=\; x^4 + x^3 + p_0 \quad \textit{and} \quad P_{II}(x) \;=\; x^4 + x^3 + x^2 + p_0$$

are called Type I and Type II polynomials[1]*, respectively.*

Although we can not provide a general proof of existence for these polynomials, the existence of Type I polynomials for certain ground fields $GF(2^n)$ is provided by the following lemma.

Lemma 1 *(Irreducible) Type I polynomials exist for all ground fields $GF(2^n)$ with n odd.*

> **Proof.** We show that the specific polynomial $P(x) = x^4 + x^3 + 1$ is irreducible over all fields $GF(2^n)$ where n is odd. P is certainly irreducible over $GF(2)$ (see e.g. [LN83, Table C].) Corollary 1.3.12 in [Jun93, page 23] states that P remains irreducible over an extension field $GF(2^n)$ of $GF(2)$ if and only if $\gcd(n, \deg(P)) = \gcd(n, 4) = 1$. Obviously, this condition is fulfilled by all odd n. $\diamond$

Using Type I or Type II polynomials as field generators results in the following partial complexities of multipliers in $GF((2^n)^4)$:

[1] These polynomials should not be confused with type I and type II normal bases introduced by Mullin et al. in [MOVW89]. They are not related whatsoever.

Theorem 12 *If there exists a polynomial $P(x)$ of Type I or Type II, the third step of the KOA for polynomials of degree three and the reduction* mod$P(x)$ *can be implemented in parallel with the following space and time complexities, respectively:*

$$\mathcal{C}_{I(II)} = 14(15) \oplus +3C_{\otimes_{p_0}}, \tag{6.10}$$

$$T_{I(II)} = 3T_{\oplus} + T_{\otimes_{p_0}}, \tag{6.11}$$

where $C_{\otimes_{p_0}}$ denotes the complexity of constant multiplication with p_0 and the numbers in parenthesis the complexities for Type II polynomials.

Proof. The operation mod$P(x)$:

$$C(x) = C'(x) \bmod P(x) \quad , \quad C(x) \in GF((2^n)^4)$$

can be viewed as the linear mapping of the seven coefficients c_i' into the four coefficients c_i. The actual equations for the mapping with P_I and P_{II} polynomials are:

$$\begin{array}{lcllcl} c_0 &=& d_0 + p_0(d_2 + d_5 + d_7 + d_8) & c_0 &=& d_0 + p_0(d_2 + d_5 + d_7) \\ c_1 &=& d_0 + d_1 + d_2 + p_0(d_6 + d_7) & c_1 &=& d_0 + d_1 + d_2 + p_0(d_6 + d_7) \\ c_2 &=& d_0 + d_2 + d_3 + d_6 + p_0 d_8 & c_2 &=& d_0 + d_3 + d_5 + d_6 + d_7 + p_0 d_8 \\ c_3 &=& d_0 + d_1 + d_3 + d_4 + d_6 & c_3 &=& d_0 + d_1 + d_3 + d_4 + d_7 + d_8 \end{array} \tag{6.12}$$

Certain simplifications have applied since all arithmetic is done in $GF(2^n)$ with characteristic 2, i.e. $d_i + d_i = 0$. If the terms $(d_0 + d_1)$, $(d_3 + d_6)$ and $(d_0 + d_1)$, $(d_6 + d_7)$ are precomputed, the two set of Equations (6.12) have the space and time complexities stated in Theorem 12. $\diamond$

We are now able to state the overall complexities of multipliers based on Type I and Type II polynomials. Since the complexities in Theorem 12 refer to the third stage of the KOA and the modulo reduction, the overall complexity is obtained by summing these complexities and the complexities for the first and second step of the KOA, developed in Section 5.2.2. The KOA complexities are given by Equation (5.11) through (5.14) if $m = 4$, or can easily be obtained from Figure 6.2. The summation results in:

$$\mathcal{C}_{I(II)} = 24(25) \oplus +9 \otimes +3\otimes_{p_0}, \tag{6.13}$$

$$T_{I(II)} = 5T_{\oplus} + 1T_{\otimes} + T_{\otimes_{p_0}}. \tag{6.14}$$

Again, to the ground field multiplication the architecture of Mastrovito is applied. Assuming a complexity of n^2 AND / $(n^2 - 1)$ XOR gates for the multiplier, we obtain:

$$\#\text{AND} = \frac{9}{16}k^2 \tag{6.15}$$

$$\#\text{XOR} = \frac{9}{16}k^2 + 24(25)\frac{k}{4} - 9 + 3C_{\otimes_{p_0}} \quad \text{, certain } n \tag{6.16}$$

$$\mathcal{T}_{\text{and}} = 1 \tag{6.17}$$

$$\mathcal{T}_{\text{xor}} \leq 5 + 2\lceil \log_2 n \rceil + T_{\otimes_{p_0}}. \tag{6.18}$$

k	n	$P(x)$	type	$C_{\otimes_{p_0}}$ XOR	$AB \bmod P$ AND	$AB \bmod P$ XOR	k^2	$AB \bmod P$ $\mathcal{T}_{\text{and}}$	$AB \bmod P$ $\mathcal{T}_{\text{xor}}$
8	2	$1110\omega^2$	II	1	**36**	**80**	64	**1**	**8**
12	3	1100ω	I	1	**81**	**147**	144	**1**	**9**
16	4	1110ω	II	1	**144**	**238**	256	**1**	**9**
20	5	$1100\omega^{30}$	I	1	**225**	**339**	400	**1**	**11**
24	6	$1110\omega^{58}$	II	5	**324**	**480**	576	**1**	**11**
28	7	1100ω	I	1	**441**	**602**	784	**1**	**10**
32	8	$1110\omega^{13}$	II	14	**576**	**998**	1024	**1**	**12**

Table 6.2: Space and time complexities for multipliers in $GF((2^n)^4)$

6.2.2 Results

We determined *primitive* Type I or II polynomials for the ground fields $GF(2^n)$, $n = 2, 3, \ldots, 8$ through an exhaustive search. Type I polynomials were preferred due to their smaller implementational complexity (6.10). The search determined the polynomials which have the lowest complexity $C_{\otimes_{p_0}}$ for multiplication with the coefficient p_0.

Table 6.2 lists the space and time complexities of parallel multipliers in $GF((2^n)^4)$. For the ground field multipliers the actual complexities of the Mastrovito architecture, given in Table 3.1, were used. The table is to be interpreted as follows.

The column headed by $P(x)$ contains the best *primitive* Type I and Type II polynomials found. The next column denotes the type of the best polynomial found, i. e. either Type I or II. The column headed by $C_{\otimes_{p_0}}$ contains the complexity for multiplication with the constant p_0. The overall space complexity is given in bold face letters in the two columns headed by $AB \bmod P$. The column k^2 allows comparison with the lower complexity bound of traditional architectures. The two rightmost columns provide expressions for the time complexity in multiples of AND and XOR gate delays.

6.2.3 Evaluation

The multiplier introduced here has a gate count of $9/16k^2$ AND gates and is thus $7/16 = 44\%$ better than the lower bound of traditional architectures. The XOR complexity is higher than the AND complexity, although for most fields considered still below the k^2 bound. Only for the field $GF(2^8)$ the number of XOR gates is with 80 considerably higher than $k^2 = 64$.

For the values $k = 16, 20, 24, 28$ the architecture proposed in this section possesses the lowest gate count among the architectures proposed in this thesis. If the architecture is compared to the general architecture over composite fields $GF((2^n)^m)$ with $m = 4$ described in Chapter 5, we find that the number of XOR gates has improved between 5 and 11 %, while the number of AND gates remains the same. However, it seems as though

the major advantage of the architecture lies in its low time complexity. Compared to the general architecture with $m = 4$, the architecture here reduces the number of XOR gate delays by three for the fields $k = 16, 20, 24, 28$. This corresponds to an improvement between 20 and 25 %. For instance, the multiplier over $GF(2^{16})$ introduced here contains in its critical path only 9 XOR gates, whereas the multiplier from Table 5.1 contains 12 XOR gates.

However, for the finite field $GF(2^{32})$ the general architecture allows the composition $GF((2^4)^8)$ which results in a better gate count. For even larger values of k, i. e. $k = 64$, the general architecture will further improve because of its better asymptotical complexity. The general architecture has a complexity of order $\mathcal{O}(k^{\log_2 3})$ for the pure polynomial multiplication compared to $\mathcal{O}(k^2)$ for the architecture over $GF((2^n)^4)$ introduced in this chapter.

In [Afa91], a multiplier over tower fields together with two examples for the fields $GF(2^8)$ and $GF(2^{16})$ was introduced (see Subsection 3.2.2 for a description.) In both cases, the XOR complexity is slightly better than ours (2 and 4 gates, respectively,) when the AND complexity is the same. However, due to the field decomposition used, $GF(((2^2)^2)^2)$ and $GF(((2^4)^2)^2)$, respectively, the modularity of the architectures seems to be somewhat worse.

Chapter 7

A Comparitive Gate Array Synthesis of Multipliers

7.1 Motivation

In this chapter a VLSI synthesis of four different parallel Galois field multipliers is described[1]. The architectures were mapped to the library of the gate-array family TC 160G from Toshiba. The result was a netlist for each architecture and field order. The synthesis was performed in order to clarify the following questions regarding parallel finite field multipliers with respect to a gate-array implementation:

- What is the number of gate equivalences (or netto gates) of multipliers over composite fields compared to those of traditional architectures?
- What is the estimated time behavior of architectures over composite fields relative to traditional architectures?
- Is the theoretical gate count (in XOR/AND gates) a valid measure for the number of netto gates?
- Which maximum clock frequency can be achieved in a given, e.g. commercial, application?

We also try to contribute to closing the gap which exists between many theoretical publications describing different approaches to VLSI suitable Galois field multipliers (see e.g. the reference list) on the one hand, and the rather few reports available comparing architectures from a technical point of view on the other hand. To the author's knowledge there are only two papers with a strong comparative character: Hsu et al. compare in

[1]The synthesis was a joint project with the Institut für angewandte Mikroelektronik (IAM), Braunschweig. The design entry and the running of the design tools was performed by the IAM. Special thanks to Niko Lange.

[HTDR88] three different multipliers in dual, normal, and standard base, respectively. The architectures considered are bit serial. Their approach, which is restricted to fields $GF(2^8)$, results in actual implementations in NMOS technology. One of the paper's conclusions is that the dual base multiplier performs best with respect to area requirement. The second, more recent, paper is by Jeong and Burleson [JB92] and compares various multipliers from a high level description point of view. The VLSI synthesis is based on Dependency and Signal Flow Graphs [Kun88], focusing on systolic array architectures. Due to the abstract character of the description, the comparison is more general than the one in [HTDR88]. One of their results is that standard and dual base serial multipliers have a similar space complexity, whereas normal base multipliers require more area.

In addition to the two articles mentioned above, Geiselmann and Gollmann compare in [GG90] bit serial architectures for exponentiation. The comparison assumes a full custom design VLSI chip. The two architectures compared use standard and normal base representation of the field elements, respectively. The major conclusion is that exponentiation in normal base does not necessarily results in a lower complexity than standard base exponentiation. It is recommended that normal base architectures are only used in situations where the exponentiation requires relatively few general multiplications.

Unlike the articles from above, we will consider bit parallel architectures. Moreover, we will try to provide absolute and relative values regarding the area and time performance of different multipliers with respect to an implementation on a gate-array.

7.2 Architectures Compared and Methods

For the comparison of implementations of parallel Galois field multipliers on one specific gate-array we studied the VLSI synthesis of four different architectures. The field orders 2^k considered range from $k = 4$ to $k = 32$, thus being consistent with the architectures treated in this thesis. Assuming that fields whose elements can be represented by multiples of 8 bit are most interesting for applications, we considered, besides $k = 4$, the fields $k = 8, 16, 24$ and 32. The architectures compared are:

1. Standard base multipliers over composite fields (SB/comp. fields) as proposed in this thesis,
2. Standard base (SB) multipliers as proposed in [Mas91],
3. Dual base (DB) multipliers as described in Section 3.1.2,
4. Normal base (NB) multipliers as described in Section 3.1.3.

In order to provide comparable conditions, the field polynomials of all multipliers were chosen to be *primitive*. The following polynomials were selected:

1. For the multipliers over composite field the architectures from Section 6.1 were used for the field orders $k = 4, 8$. For the larger fields the architectures from Section 6.2 was chosen.

2. For the multipliers [Mas91] the polynomials given in Table 3.1 were used for field orders $k = 4, 8, 16$. For the two large fields investigated the polynomials $Q(y) = y^{24} + y^4 + y^3 + y^2 + y + 1$ and $Q(y) = y^{32} + y^7 + y^5 + y^3 + y^2 + y + 1$ were used, respectively.

3. We used the following primitive polynomials for the normal base multiplier:

$$\begin{aligned}
Q(y) &= y^4 + y^3 + 1 \\
Q(y) &= y^8 + y^7 + y^5 + y^3 + 1 \\
Q(y) &= y^{16} + y^{15} + y^{13} + y^{12} + y^{11} + y^{10} + y^8 + y^7 + y^5 + y^3 + y^2 + y + 1 \\
Q(y) &= y^{24} + y^{23} + y^{22} + y^{21} + y^{19} + y^{17} + y^{15} + y^{14} + y^{13} + y^{11} + y^{10} + y^8 + \\
&\quad y^6 + y^4 + y^3 + y + 1 \\
Q(y) &= y^{32} + y^{31} + y^{29} + y^{28} + y^{27} + y^{26} + y^{23} + y^{22} + y^{20} + y^{18} + y^{16} + y^{15} + \\
&\quad y^{14} + y^{13} + y^{12} + y^{10} + y^8 + y^5 + 1
\end{aligned}$$

   The first polynomial was chosen as suggested in [Mas91]. The polynomials for $k = 8, 16, 32$ were taken from [Gei93a] and they are optimal with respect to the theoretical gate count. The polynomial for $k = 24$ was provided by W. Geiselmann [Gei93b]. It is the best primitive polynomial with respect to multiplier complexity. However, the multiplier over $GF(2^{32})$ was found to be too large for the VLSI tools used, so that it could not be synthesized.

4. For the dual base multiplier primitive polynomials with the lowest possible coefficient weight were chosen. Since primitive trinomials does not exist if $8|k$, the polynomials have $(3 + 2i)$, i integer, coefficients for $k = 8, 16, 24, 32$:

$$\begin{aligned}
Q(y) &= y^4 + y + 1 \\
Q(y) &= y^8 + y^4 + y^3 + y^2 + 1 \\
Q(y) &= y^{16} + y^5 + y^3 + y^2 + 1 \\
Q(y) &= y^{24} + y^4 + y^3 + y + 1 \\
Q(y) &= y^{32} + y^7 + y^5 + y^3 + y^2 + y + 1
\end{aligned}$$

The target hardware of the comparative synthesis was the TC 160G, which is a modern and often applied family of gate-arrays from Toshiba. The TC 160G's are sea-of-gates chips realized in $0.8\mu m$ CMOS technology. All architectures were entered in *Verilog-HDL* (hardware description language) into the computer. The mappings onto the gate-array library was performed automatically by the synthesis tool *Synopsys*, resulting in corresponding netlists. This approach is of great practical importance, since the application of a highly automated design process leads to a shortened development time (faster "time-to-market") and in turn to reduced development costs. However, the automatic technology mapping does not guarantee optimal results.

Each synthesis resulted in an absolute measure for the netto gate consumption which is the number of gate equivalences (g.e.). The value is an approximate measure of the chip area needed in an actual implementation of the architecture. Second, each synthesis provided a measure of the time complexity, or delay, of each multiplier. Although the delay is given in absolute units (nanoseconds, ns,) the time behavior of an actual implementation depends heavily on the surrounding circuitry and the chip size. These parameters influence the interconnection delay. Therefore, the delay values are only a rough estimate of the speed of the multipliers in hardware implementations. However, all delay times given in this study are valid *relative* measures, with which the speed of different multipliers can be compared. In order to underline the limited relevance of the absolute delay values, all numbers denoting multiples of nanoseconds and values derived from those, are given in parenthesis in all tables below.

All syntheses were performed twice, once with each of the two compiler options "smallest" or "fastest." When the "smallest" option is set, *Synopsys* tries to realize the architecture with the smallest number of gate equivalences. The "fastest" option leads to architecture mappings which possess a minimized critical path. Generally speaking, it was found that the gate complexity increases only by 10–20% when the "smallest" architectures were compared to the corresponding "fastest" ones. On the other hand, the delay was reduced up to 50%, depending on the architecture and the field order, when the compiler option was switched from "smallest" to "fastest".

7.3 Results

7.3.1 Comparison of the Gate Consumption

This section shows a comparison of the number of gate equivalences, required for the multiplier architectures. In order to achieve architectures with a minimized number of netto gates, the compiler option was set to "smallest." Table 7.1 shows the complexity as absolute values in gate equivalences. In addition, these number were normalized with respect to the best multiplier for each field order 2^k, thus providing a relative measure for comparing the different architectures. The two measures are both printed in bold face numbers. The table also provides the delays as computed by *Synopsys*. As described above, these values are most useful for *comparing* the different architectures. Since their usefulness as absolute values is limited, they are given in parenthesis.

The table shows that the composite field multiplier performs best in terms of gate consumption for all fields except for the smallest field $GF(2^4)$. The Mastrovito multiplier requires between 26 and 42% more gate equivalences than the composite field multiplier. The dual base multiplier shows a similar behavior; it requires between 27 and 34% more gate equivalences. The normal base multiplier requires by far the most gates. Relative to the multiplier over composite fields, it takes between 160–582% more gate equivalences.

With respect to the time performance, the different architecture are not as easy to classify. For the fields with $k = 8, 16$ the composite field multiplier is somewhat faster

	SB/comp. fields			SB			NB			DB		
	gates		delay	gates		delay	gates		delay	gates		delay
	abs.	rel.	abs.	abs.	rel.	abs.	abs.	rel.	abs.	abs.	rel.	abs.
k	[g.e.]		[ns]	[g.e.]		[ns]	[g.e.]		[ns]	[g.e.]		[ns]
4	**69**	**1.11**	(3.00)	**62**	**1.00**	(2.30)	**103**	**1.66**	(2.86)	**62**	**1.00**	(2.30)
8	**243**	**1.00**	(3.81)	**307**	**1.26**	(4.49)	**508**	**2.09**	(4.30)	**322**	**1.33**	(5.31)
16	**885**	**1.00**	(6.26)	**1290**	**1.46**	(6.61)	**3709**	**4.19**	(14.84)	**1120**	**1.27**	(6.88)
24	**1819**	**1.00**	(10.09)	**2576**	**1.42**	(8.09)	**10581**	**5.82**	(13.17)	**2445**	**1.34**	(6.09)
32	**3554**	**1.00**	(11.36)	**4650**	**1.31**	(10.37)				**4536**	**1.28**	(8.73)

Table 7.1: Comparison of the netto gate consumption of parallel finite field multipliers over $GF(2^k)$ on the gate-array TC 160G (compiler option set to "smallest")

	SB/comp. fields			SB			NB			DB		
	delay		gates	delay		gates	delay		gates	delay		gates
	abs.	rel.	abs.	abs.	rel.	abs.	abs.	rel.	abs.	abs.	rel.	abs.
k	[ns]		[g.e.]	[ns]		[g.e.]	[ns]		[g.e.]	[ns]		[g.e.]
4	**(1.84)**	**1.18**	95	**(1.60)**	**1.03**	85	**(1.94)**	**1.24**	169	**(1.56)**	**1.00**	99
8	**(2.74)**	**1.00**	322	**(3.04)**	**1.11**	395	**(2.96)**	**1.08**	654	**(3.77)**	**1.38**	392
16	**(4.50)**	**1.09**	1009	**(4.31)**	**1.05**	1498	**(4.11)**	**1.00**	4929	**(4.27)**	**1.04**	1277
24	**(7.13)**	**1.58**	1920	**(4.73)**	**1.05**	3125	**(5.25)**	**1.16**	14173	**(4.52)**	**1.00**	2829
32	**(8.24)**	**1.52**	3976	**(6.08)**	**1.12**	5332				**(5.43)**	**1.00**	5189

Table 7.2: Comparison of the estimated delay of parallel finite field multipliers over $GF(2^k)$ on the gate-array TC 160G (compiler option set to "fastest")

than the other architectures, for the field with $k = 4$ the SB and the DB architecture perform best, and for $k = 24, 32$ the DB is clearly the fastest. The NB multiplier has for the values $k = 16, 24$ a considerably longer delay than all other architectures.

7.3.2 Comparison of the Time Behaviors

This section compares the delays of the synthesized parallel multipliers. In order to achieve minimal delays, the compiler option was set to "fastest." Table 7.2 provides absolute and relative measures for the delays. The absolute values, measured in nanoseconds and printed in bold face numbers, were computed by the synthesis tool *Synopsys*. Since the absolute values are not exact measures of the physical delays in actual implementations, but rather estimations, they are given in parenthesis. For each field order parameter k, the speed of all multipliers was normalized with respect to the fastest one. These relative values are also given in bold face numbers. The third parameter for each architecture and field order is the absolute number of gate equivalences.

The table shows that certain architectures perform best for certain field orders. In

compiler option:	"fastest"			"smallest"		
	delay	o	tp	delay	o	tp
k	[ns]	[Mop/s]	[Gbit/s]	[ns]	[Mop/s]	[Gbit/s]
4	(1.84)	**(543)**	**(2.17)**	(3.00)	**(333)**	**(1.33)**
8	(2.74)	**(365)**	**(2.92)**	(3.81)	**(262)**	**(2.10)**
16	(4.50)	**(222)**	**(3.65)**	(6.26)	**(160)**	**(2.56)**
24	(7.13)	**(140)**	**(3.37)**	(10.09)	**(99)**	**(2.38)**
32	(8.24)	**(121)**	**(3.88)**	(11.36)	**(88)**	**(2.82)**

Table 7.3: Estimated speed and data throughput of parallel multiplier modules over composite fields on the gate-array TC 160G

this respect the behavior is different from the comparison of the gate complexity, where the composite field architectures possess the lowest gate count for almost all fields. The dual base multiplier achieves the smallest delays for $k = 4, 24, 32$. For $k = 16$ the normal base multiplier is the fastest, and for $k = 8$ the one over composite fields. However, the architecture from Mastrovito is for all fields only slightly slower than the best ones. For fields with $k = 24, 32$ the multiplier over composite fields is considerably slower than the other architectures.

The relative netto gate requirements are similar to the situation where the compiler option was set to "smallest." The SB and DB multipliers show a very similar behavior, while the NB multiplier needs considerably more gate equivalences. The multiplier over composite fields performs again best for all fields but for $k = 4$.

7.3.3 Estimation of the Theoretical Throughput of Multipliers over Composite Fields

In this section an estimation of the maximal achievable operational speed and the corresponding data throughput for the multiplier architectures over composite fields is provided in Table 7.3. The maximum operational speed, measured in operations per second [op/s], is also the maximum clock frequency, measured in clocks per second [Hz]. This is due to the fact that the architectures only contain combinatorial logic but do not possess any registers. The approach here is based on the assumption that the clock frequency is the reciprocal of the absolute delays estimated by *Synopsys*. It should be emphasized that the investigation is restricted to the consideration of the theoretical clock frequency of a single multiplier module. Issues such as the time behavior of entire systems, e.g. Reed-Solomon decoders, and delays caused by data I/O are not considered.

We compared each architecture compiled with both options, "smallest" and "fastest." The time complexity, in nanoseconds, obtained this way are given in the two columns headed by "delay." If the delay values are denoted with t, the number of operations per

second o is achieved by:

$$o = \frac{1}{t} \quad [\text{op/s}].$$

The values calculated are given in Mop/s. In the rightmost columns we compare the theoretical data throughput of the multiplier modules. We define the data throughput as the number of bits per seconds which can be obtained from the multiplier output when the architecture is clocked with its maximal speed. Since a multiplier produces exactly k bits of output per operation, the data throughput tp is:

$$tp = k\, o = \frac{k}{t} \quad [\text{bit/s}].$$

The values achieved are given in the table in Gbit/s.

As expected, the maximal clock frequency decreases as the field order increases. For the architectures compiled with the "fastest" option set, the maximal frequency ranges from 543 down to 121 MHz. The corresponding values for architectures compiled with the "smallest" option range from 333 down to 88 MHz. However the data throughput increases as k grows. This indicates that the maximal clock frequency, which behaves reciprocal to the length of the critical path, decreases slower than the logarithm of the field order: $\log_2(2^k) = k$.

7.4 Conclusions

We investigated the synthesis of different parallel multiplier architectures with respect to the gate-array family TC 160G. The comparison included three traditional multipliers and architectures over composite fields. The results from this comparison supports the major achievement of this thesis, which is the development of parallel Galois field multipliers with a low gate count. In particular it is shown that the theoretically low gate count can be transformed to the netto gate count of gate-arrays under the given conditions which are:

- the target hardware is the sea-of-gate chip family TC 160G, i.e. semi-custom chips,
- the architectures are entered in HDL and the synthesis is performed automatically using the general purpose tool *Synopsys*.

In particular, the use of a general purpose synthesis tool does not take into account most of the structural properties of the different architectures.

It was found that the SB and DB multipliers show a very similar behavior with respect to both, time and gate requirements. The NB architecture requires significantly more gates than all other multipliers. Hence it is doubtful if a parallel NB architecture is well suited for large field orders, where $k \geq 16$. The results regarding the netto gate requirements support the conclusions drawn for serial architectures in [JB92], where it

was also found that SB and DB possess a similar performance, while the NB performs worse. However, it should be noted that our comparison did not evaluate the base transformation necessary for the DB multiplier. In order to perform the base transformation, computations are required. A pure permutation is not possible, since this requires the existence of irreducible trinomials (see Section 3.1.2), which do not exist if $8|k$ [Gol67].

Another set of syntheses aimed on *fast* multipliers. It was found that the architectures are approximately equally fast for fields with $k = 4, 8, 16$, if delay optimized architectures were synthesized. For larger fields the composite field multiplier was found to be considerably slower under the given conditions. However, comparing the "fastest" composite field architectures with the corresponding "smallest" traditional architectures shows that the first one outperforms the traditional multipliers with respect to both, gate *and* time complexity. The only exception is the field $GF(2^{24})$, where the "smallest" DB multiplier is somewhat faster than the "fastest" composite field multiplier. As a conclusion, it seems as though the traditional multipliers compiled with the "smallest" option are not the best choices under the given conditions.

An estimation of the maximal data throughput for the composite field multiplier resulted in values between 1.33 and 3.88 Gbit/sec. These values correspond to clock frequencies from 333 down to 121 MHz. Due to the achievable high data throughput, the architectures seem to be attractive for many high speed applications, such as fast special purpose Reed-Solomon decoder chips or dedicated arithmetic units for general purpose processors.

Chapter 8

Parallel Inverters over Composite Fields

8.1 Introduction

This chapter introduces an architecture for parallel inversion over composite fields which is based on an idea of Itoh and Tsujii [IT88, Section 6]. The original algorithm was applied to composite fields $GF((2^n)^m)$ represented in normal base. However, we will investigate the algorithm's application to composite fields in standard base representation. Unlike the original algorithm, we propose that inversion in the subfield is performed by a direct method rather than by Fermat's Theorem. It will be shown that the significantly lower complexity of the algorithm compared to other architectures allows the implementation of parallel inverters. Moreover, we will show that the basic algorithm proposed by Morii and Kasahara [MK89] is a special case of the algorithm explained hereafter.

For the complexity of the algorithm, the following measures will be used.

- $\otimes_n$ and $\otimes_{nm}$ denote general multiplication in the fields $GF(2^n)$ and $GF((2^n)^m)$, respectively.
- $\otimes_n^{cnst}$ and $\otimes_{nm}^{cnst}$ denote constant multiplication in the fields $GF(2^n)$ and $GF((2^n)^m)$, respectively.
- $\otimes_n^{-1}$ denotes inversion in the field $GF(2^n)$.
- $\oplus_n$ denotes addition in the field $GF(2^n)$.

The goal is the determination of the inverse of $A \in GF((2^n)^m)$, $A \neq 0$. A is given as

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \;\; a_i \in GF(2^n).$$

As do many other architectures, we also apply Fermat's Theorem which is in the notations used here

$$A^{2^{nm}-1} = 1, \;\; \forall A \in GF((2^n)^m) \setminus \{0\}. \tag{8.1}$$

Equation (8.1) is equivalent to

$$AA^{2^{nm}-2} = 1,$$

from which it follows that

$$A^{-1} = A^{2^{nm}-2}. \tag{8.2}$$

Equation (8.2) shows that computing the inverse of a field element A can be accomplished by raising it to the power of $2^{nm}-2 = 2+2^2+2^3+\cdots+2^{nm-1}$. A straightforward application of the well known "binary method" [Knu81] of repeated squaring and multiplying results in a computational complexity of $nm-1$ squarings and $nm-2$ multiplications. These operations refer to arithmetic in the field $GF((2^n)^m)$. However, the binary method does not produce optimum results. In [IT88, Section 4, Theorem 2] an improved method is proposed, which reduces the number of multiplications to

$$\lfloor \log_2(nm-1) \rfloor + H_w(nm-1) - 1 \leq 2\lfloor \log_2(nm-1) \rfloor, \tag{8.3}$$

where $H_w(\cdot)$ denotes the Hamming weight of the operand's binary representation.

The outline of the remainder of this chapter is as follows. The next section develops the algorithm given in [IT88] in our notation. After the algorithm is introduced, expressions for the computational complexity with respect to a composite field representation in standard base will be derived. In Section 8.4 it will be shown that the algorithm results in the architecture [MK89] if we choose $m = 2$. In the last section, two examples for parallel inversion in the important fields $GF(2^8)$ and $GF(2^{16})$ will be given.

8.2 Itoh and Tsujii's Algorithm for Inversion in Composite Fields

The basic property of the algorithm explained in this section is that inversion in $GF((2^n)^m)$ is reduced to inversion in the subfield $GF(2^n)$. Itoh and Tsujii's algorithm will be developed with a different notation, starting with the following lemma.

Lemma 2 *The multiplicative inverse of an element A of the composite field $GF((2^n)^m)$ can be computed by*

$$A^{-1} = (A^r)^{-1} A^{r-1},$$

where $A^r \in GF(2^n)$.

Proof. First, the auxiliary parameter[1] r is defined as

$$r = \frac{2^{nm}-1}{2^n-1}.$$

[1] r corresponds to the parameter a in the original paper.

An important property of r is that [LN83] :

$$A^r \in GF(2^n), \quad \forall A \in GF((2^n)^m), \tag{8.4}$$

from which Lemma 2 follows directly. $\diamond$

We are also able to establish a relationship between the inversion formula given above and the inversion based on Fermat's Theorem. The exponent $2^{nm} - 2$, needed for the inversion according to Equation (8.2), can be expressed in terms of r:

$$2^{nm} - 2 = \frac{2^{nm} - 1}{2^n - 1}(2^n - 1) - 1 = r(2^n - 1) - 1 = r(2^n - 2) + r - 1.$$

Inserting the new expressions into Equation (8.2) yields

$$A^{-1} = A^{r(2^n-2)+r-1} = A^{r(2^n-2)}A^{r-1} = (A^r)^{-1}A^{r-1},$$

where for the final step property (8.4) was used.

Lemma 2 implies a new method for computing the multiplicative inverse for a composite field element. The method will be divided into four steps:

Step 1 Compute A^{r-1} (Exponentiation in $GF((2^n)^m)$.)

Step 2 Compute $A^{r-1}A = A^r$ (Multiplication in $GF((2^n)^m)$, where the product is an element of $GF(2^n)$.)

Step 3 Compute $(A^r)^{-1} = A^{-r}$ (Inversion in $GF(2^n)$.)

Step 4 Compute $A^{-r}A^{r-1} = A^{-1}$ (Multiplication of an element from $GF(2^n)$ with an element from $GF((2^n)^m)$.)

For the remainder of the chapter, a parallel implementation of the corresponding architecture will be investigated. Figure 8.1 shows a block diagram of a parallel realization of the architecture. It is assumed that all blocks work bit parallel.

8.3 Analysis of the Complexity of a Parallel Realization

In this section the complexity of the algorithm's four steps are analyzed. We will use the complexity measures which were given in the introduction.

Figure 8.1: Block diagram of an inverter over composite fields

8.3.1 Complexity of Step 1

Step 1 of the algorithm above is the following operation

$$A^{r-1}, \quad A \in GF((2^n)^m),$$

where r is defined in Lemma 2. The operation is clearly an exponentiation in the field $GF((2^n)^m)$. The special structure of r, together with the fact that A is element of a composite field, will lead to an efficient method.

The parameter r can be expressed as a sum of powers:

$$r-1 = \frac{2^{nm}-1}{2^n-1} - 1 = 2^n + 2^{2n} + 2^{3n} + \cdots + 2^{(m-1)n}.$$

This representation is similar to the binary representation of the number $2^{nm}-2 = 2+2^2+2^3+\cdots+2^{nm-1}$. Hence the optimized method from [IT88] with the computational complexity given in Equation (8.3) can be applied. The method requires $\lfloor \log_2(m-1) \rfloor + H_w(m-1) - 1$ general multiplications and $m-1$ exponentiations to the power of 2^n, with both types of operations performed in $GF((2^n)^m)$. Efficient structures for general multiplication are studied in detail in Chapters 5 and 6. In the following the exponentiation will be studied.

Let B and C be elements of $GF((2^n)^m)$. We wish to perform the exponentiation of $C(x) = B^{2^n}$, where $B(x) = \sum_{i=0}^{m-1} b_i x^i$. This can be performed as follows (the proof is based on [McE87, Lemma 5.12]:)

$$C(x) = \sum_{i=0}^{m-1} c_i x^i = \left(\sum_{i=0}^{m-1} b_i x^i \right)^{2^n} = \sum_{i=0}^{m-1} b_i^{2^n} x^{i2^n} = \sum_{i=0}^{m-1} b_i x^{i2^n}, \quad b_i \in GF(2^n). \tag{8.5}$$

In order to achieve general expressions for the complexity, we assume $2^n > m-1$. With this assumption, there are $m-1$ powers of x which must be reduced modulo the field polynomial $P(x)$, namely the powers x^{i2^n}, $i = 1, 2, \ldots, m-1$. We use the following notation for the representation of these powers in the residue classes modulo $P(x)$:

$$x^{i2^n} = s_{0,i} + s_{1,i}x + \cdots + s_{m-1,i}x^{m-1} \bmod P(x), \quad i = 1, 2, \ldots, m-1.$$

Using the coefficients $s_{j,i}$, the exponentiations in Equation (8.5) can be expressed in matrix form as

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} 1 & s_{0,1} & s_{0,2} & \cdots & s_{0,m-1} \\ 0 & s_{1,1} & s_{1,2} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & s_{m-1,1} & s_{m-1,2} & \cdots & s_{m-1,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix}.$$

Since all $s_{j,i}$ are (constant) elements from $GF(2^n)$, the complexity of one exponentiation is $(m-1)m$ constant multiplications ($\otimes_n^{cnst}$) and $m(m-2)+1 = (m-1)^2$ additions ($\oplus_n$). The complexity of the first step is therefore

$$\begin{aligned} C_1 &= (m-1)[(m-1)m \otimes_n^{cnst} + (m-1)^2 \oplus_n] + [\lfloor \log_2(m-1) \rfloor + H_w(m-1) - 1] \otimes_{nm} \\ &= (m-1)^2 m \otimes_n^{cnst} + (m-1)^3 \oplus_n + [\lfloor \log_2(m-1) \rfloor + H_w(m-1) - 1] \otimes_{nm}. \end{aligned} \quad (8.6)$$

8.3.2 Complexity of Step 2

Step 2 performs the operation

$$A^r = A^{r-1}A, \quad (8.7)$$

where $A^r \in GF(2^n)$, and the two operands are elements in $GF((2^n)^m)$. The operand A^{r-1} is the result of the computations of Step 1. One possibility for computing (8.7) is to apply a general multiplier over $GF((2^n)^m)$, such as suggested in [IT88]. In this case, the complexity of Step 2 is

$$C_2' = \otimes_{nm}.$$

However, it is possible to take advantage of the à priori knowledge that A^r is an element of the subfield. For small values of m, this leads to a reduced complexity. In the sequel, complexity expressions for this approach will be developed.

In order to provide general expressions, we consider the multiplication of $B \cdot C = D \bmod P(x)$, with $B, C \in GF((2^n)^m)$ and $D \in GF(2^n)$. First, we consider the pure polynomial multiplication of B and C:

$$\begin{aligned} D'(x) = B(x)C(x) &= \left(\sum_{i=0}^{m-1} b_i x^i \right) \left(\sum_{i=0}^{m-1} c_i x^i \right) \\ &= \left(\sum_{i=0}^{m-2} d_i x^i \right). \end{aligned} \quad (8.8)$$

We know that $D'(x) = d_0 \bmod P(x)$, i.e. that all but the zero coefficient vanish after reduction modulo $P(x)$. If the matrix representation from Chapter 5, introduced in Equation (5.20), is used for the modulo reduction, the coefficient d_0 can be expressed as

$$\begin{aligned} D = d_0 &= d'_0 + r_{0,0}c'_m + r_{0,1}c'_{m+1} + \cdots + r_{0,m-2}c'_{2m-2} \\ &= d'_0 + \sum_{i=0}^{2m-2} r_{0,i}d'_{m+i} \bmod P(x). \end{aligned} \tag{8.9}$$

The coefficients $r_{0,i}$ are the entries of the uppermost row of the reduction matrix $\mathbf{R}$ in Equation (5.20). There are $m-1$ constant multiplications with the coefficients $r_{0,i}$ involved. Equation (8.9) reveals that the computation of D only requires the coefficients d'_i, $i = m, m+1, \ldots, 2m-2$ and d'_0. Application of the straightforward method for polynomial multiplication to the computation of these coefficient results in an overall complexity for Step 2 of:

$$C''_2 \leq \frac{m(m-1)+2}{2} \otimes_n + \frac{m(m-1)}{2} \oplus_n + (m-1) \otimes_n^{cnst}. \tag{8.10}$$

For small values of m, the complexity (8.10) is lower than the complexity of a general multiplier over $GF((2^n)^m)$. Let's consider $m = 4$ as an example. In this case, a general multiplier based on the KOA, such as described in Chapter 5, requires 9 $\otimes_n$ and 22 $\oplus_n$ (if the modulo reduction is neglected.) According to Equation (8.10), the improved method requires only 7 $\otimes_n$ and 6 $\oplus_n$ (if the constant multiplications are neglected.)

For a given polynomial $P(x)$, further improvements are possible. These improvements stem from the specific structure of the associated reduction matrix together with the à priori knowledge that all d_i except d_0 will be zero. We will explain the approach with an example.

Example. We consider a composite field $GF((2^n)^4)$, generated by a Type II polynomial as introduced in Definition 15. The reduction modulo $P(x) = x^4 + x^3 + x^2 + p_0$ after the polynomial multiplication can be expressed as

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & p_0 & p_0 & 0 \\ 0 & 1 & 0 & 0 & 0 & p_0 & p_0 \\ 0 & 0 & 1 & 0 & 1 & 1 & p_0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} d'_0 \\ d'_1 \\ \vdots \\ d'_6 \end{pmatrix},$$

where $d_1 = d_2 = d_3 = 0$. The product D could be computed through $D = d_0 = d'_0 + p_0(d'_4 + d'_5)$, which would take $6\otimes_n$, $5\oplus_n$, and 1 $\otimes_n^{cnst}$. This is already an improvement of the complexity of Equation (8.10). However, using the relations

$$\begin{aligned} d_2 &= d'_2 + d'_4 + d'_5 + p_0 d'_6 = 0, \\ d'_4 + d'_5 &= d'_2 + p_0 d'_6, \\ d_0 &= d'_0 + p_0(d'_4 + d'_5) = d'_0 + p_0 d'_2 + p_0^2 d'_6, \end{aligned} \tag{8.11}$$

it becomes possible to compute $D = d'_0$ with $5\otimes_n$, $4\oplus_n$, and $2\otimes_n^{cnst}$.

8.3.3 Complexity of Step 3

The operation of Step 3 is

$$(A^r)^{-1},$$

which is inversion in the subfield $GF(2^n)$ because of Lemma 2. Since our goal is the development of a parallel architecture, inversion in the ground field must also be performed bit parallel. For inversion in small subfields the direct method based on matrix inversion, which is described in Subsection 3.3.1, is suited. In particular, for values $n \leq 6$ direct inversion has a lower gate count than a parallel implementation of the architectures based on Fermat's Theorem, such as proposed in [IT88]. Formulas for direct inversion in the field $GF(2^n)$, with $n \leq 7$ are listed in Appendix A. It should be noted that the formulas listed are not optimized. For instance, the inversion formulas for $GF(2^4)$ require 23 XOR and 22 AND gates if implemented in a straightforward manner. On the other hand, in [Mas91, Section 9.2] optimized formulas for this case are given, which have an estimated gate count of 25, involving XOR, AND, and binary inverters.

The complexity of Step 3 is in our notation:

$$C_3 = \otimes_n^{-1}.$$

8.3.4 Complexity of Step 4

Step 4 requires the operation

$$A^{-r} \cdot A^{r-1} = A^{-1},$$

where $A^{-r} \in GF(2^n)$and $A^{r-1} \in GF((2^n)^m)$. In order to determine the complexity of this step, we denote $B = b_0 \in GF(2^n)$and $C(x) = \sum_{i=0}^{m-1} c_i x^i \in GF((2^n)^m)$. The operation $B \cdot C = D \bmod P(X)$ is:

$$\begin{aligned} D = B \cdot C &= b_0 \sum_{i=0}^{m-1} c_i x^i = \sum_{i=0}^{m-1} b_0 c_i x^i. \qquad (8.12) \\ &= \sum_{i=0}^{m-1} d_i x^i. \end{aligned}$$

Equation (8.12) has a complexity of m multiplications in the ground field. There is no reduction modulo $P(x)$ required. Hence the complexity of Step 4 equals:

$$C_4 = m \otimes_n.$$

8.3.5 Overall Complexity

The overall complexity is achieved by summation of the four partial complexities:

$$\begin{aligned} C &= C_1 + C_2' + C_3 + C_4 \\ &= [\lfloor \log_2(m-1) \rfloor + H_w(m-1)] \otimes_{nm} \\ &\quad + \otimes_n^{-1} + m \otimes_n + (m-1)^3 \oplus_n + (m-1)^2 m \otimes_n^{cnst}, \qquad (8.13) \end{aligned}$$

where the non-optimized complexity of Step 2 was assumed.

In terms of complexity, the most important steps are the first and the third ones. The major gain of the new method is that the exponent of A is reduced from $2^{nm}-2$ to $r-1=(2^{nm}-1)/(2^n-1)-1$ in Step 1. In Subsection 8.3.1 it was shown that the number of operations in $GF((2^n)^m)$, required for Step 1, is of order $\mathcal{O}(\log_2 m)$, whereas it was stated in Equation (8.2) that raising A to the power of $2^{nm}-2$ is of order $\mathcal{O}(\log_2(nm))$. The "price" which must be paid for the gain is Steps 2, 3 and 4, of which Step 3 is the most critical one. However, as can be seen in Appendix A, the direct inversion $GF(2^n)$ required by Step 3 can be implemented with a relatively small number of gates if n is kept small. Generally speaking, we must find a trade off between the two parameters n and m of the composite field $GF((2^n)^m)$. The first one determines the complexity of the inversion in the subfield, while the latter one determines the number of multiplications in the composite field $GF((2^n)^m)$. Section 8.5 will give two examples for the decomposition of the fields $GF(2^8)$ and $GF(2^{16})$, which leads to parallel inverters with moderate complexity.

8.4 A Relationship with Morii and Kasahara's Inverter

This section establishes a relationship between Itoh and Tsujii's algorithm for inversion in $GF((2^n)^m)$ in standard base, and the core algorithm of Morii and Kasahara's architecture for inversion over tower fields. The first one is described in the previous section, the latter one in Subsection 3.3.3. It will be shown that the core algorithm of the architecture is the same as Itoh and Tsujii's method for the case $m=2$.

Morii and Kasahara's architecture is based on consecutive field extensions of degree 2. However, the core algorithm is based on one field extension. In order to establish the relation, the composite fields considered are $GF((2^n)^2)$. The field polynomial is $P(x)=x^2+x+p_0$. An arbitrary field element is represented by $A(x)=a_1x+a_0$, its inverse B by $B:=A^{-1}=b_1x+b_0$. Let's recall the core algorithm of the inverter proposed by Morii and Kasahara, given in Equation (3.44):

$$\left.\begin{array}{rcl} b_0 & = & \frac{a_0+a_1}{a_0(a_0+a_1)+p_0a_1^2} \\ b_1 & = & \frac{a_1}{a_0(a_0+a_1)+p_0a_1^2} \end{array}\right\}.$$

It was stated that there are 1 inversion, 3 general multiplications, 2 additions, 1 constant multiplication with p_0 and 1 squaring required to compute the inverse from the equations above.

In the sequel, we investigate the algorithm from Section 8.2 for the case $m=2$ with the field polynomial $P(x)$. The parameter r is now $r=(2^{2n}-1)/(2^n-1)=2^n+1$. Step 1 of the algorithm is:

$$A^{r-1}=(a_1x+a_0)^{r-1}=a_1x^{r-1}+a_0=[a_1s_{1,1}]x+[a_1s_{0,1}+a_0]. \tag{8.14}$$

The computation in Step 2 is:

$$A^r = A^{r-1}A = [a_0a_1s_{1,1} + a_1^2s_{0,1} + a_0a_1 + a_1^2s_{1,1}]x + [a_0a_1s_{0,1} + a_0^2 + a_1^2s_{1,1}p_0]. \quad (8.15)$$

However, A^r is an element of the subfield and therefore its coefficient at x is zero. Using this, a relation between the coefficients can be established:

$$\begin{aligned} 0 &= a_0a_1s_{1,1} + a_1^2s_{0,1} + a_0a_1 + a_1^2s_{1,1} \\ \Leftrightarrow 0 &= a_0s_{1,1} + a_1s_{0,1} + a_0 + a_1s_{1,1} \\ \Leftrightarrow a_1s_{0,1} + a_0 &= (a_0 + a_1)s_{1,1} \end{aligned} \quad (8.16)$$

Inserting the auxiliary relation (8.16) into the Equations (8.14) and (8.15) results in new expressions for Step 1 and Step 2, respectively

$$\begin{aligned} A^{r-1} &= s_{1,1}(a_1x + [a_1 + a_0]), \\ \text{and} & \\ A^r &= s_{1,1}[a_0(a_1 + a_0) + a_1^2p_0]. \end{aligned} \quad (8.17)$$

Step 3 is the inversion of A^r:

$$(A^r)^{-1} = s_{1,1}^{-1}\,[a_0(a_1 + a_0) + a_1^2p_0]^{-1}.$$

The result in Step 4 is computed as $B = A^{r-1}(A^r)^{-1}$:

$$\begin{aligned} B(x) &= b_1x + b_0 \\ &= A^{r-1}(A^r)^{-1} = \frac{a_1x + (a_1 + a_0)}{a_0(a_1 + a_0) + a_1^2p_0}. \end{aligned} \quad (8.18)$$

Equation (8.18) is exactly the same as the resulting Equations (3.44) of the core algorithm of Morii and Kasahara. If only one field extension is used for the tower inverter, its architecture is the same as the architecture of Itoh and Tsujii's inverter in standard base in the field $GF((2^n)^2)$. Moreover, the architecture described in the previous section can be viewed as a generalization of Morii and Kasahara's core algorithm. However, it is not true that it is a generalization of Morii and Kasahara's *architecture*, since this is based on tower fields, i.e. multiple field extensions of degree two.

8.5 Two Examples

In this section the space complexity of parallel inverters in the technically important fields $GF(2^8)$ and $GF(2^{16})$ will be investigated. The measure which will be used is the number of modulo 2 adders (XOR) and multipliers (AND). It will be shown that the same field decomposition which was used for the multiplier architectures in the previous chapters can be applied. Hence these multipliers can be used as modules within the inverter architecture.

8.5.1 A Parallel Inverter over $GF(2^8)$

For inversion in the field $GF(2^8)$ we choose a decomposition into $GF((2^4)^2)$. The primitive field polynomial for the ground field is $Q(y) = y^4 + y + 1$, and the extension field is generated by $P(x) = x^2 + x + \omega^{14}$. The choice of $Q(y)$ allows the application of the Mastrovito multiplier for the ground field, which has a complexity of 15 XOR/16 AND gates. The coefficient $p_0 = \omega^{14}$ of $P(x)$ has a complexity of 1 XOR gate for constant multiplication, as can be seen from the first table in Appendix B.

As Section 8.4 showed, the optimized equations of the tower inverter can be used for this composite field. The inverse, B, of an element $A = a_1 x + a_0$ can be computed from Equation (8.18):

$$\begin{aligned} B(x) &= b_1 x + b_0 \\ &= A^{r-1}(A^r)^{-1} = \frac{a_1 x + (a_1 + a_0)}{a_0(a_1 + a_0) + a_1^2 p_0}. \end{aligned}$$

In this case, all arithmetic operations are performed in the ground field $GF(2^4)$. The operations required are 1 inversion, 3 general multiplications, 2 additions, 1 constant multiplication with p_0 and 1 squaring. Next, the gate count of the arithmetic operations in a parallel hardware implementation will be determined.

- For computing the inverse in $GF(2^4)$, the direct method described in Section 3.3.1 is developed. Appendix A lists formulas for direct inversion. For the ground field considered here, a straightforward realization of the formulas would require 23 XOR/22 AND gates. This is certainly an upper bound, because redundancies in the formulas have not been used to improve the gate count.

- For the three general multiplications, the Mastrovito multiplier from Subsection 3.1.1 is used. This results in a gate count of $3(15\,\text{XOR} + 16\,\text{AND}) = 45\,\text{XOR} + 48\,\text{AND}$.

- The two additions in $GF(2^4)$ require $2 \cdot 4 = 8$ XOR gates.

- Constant multiplication with $p_0 = \omega^{14}$ requires 1 XOR gate.

- Squaring of an element $c = \sum_{i=0}^{3} c_i y^i$ in $GF(2^4)$ involves the following operation:

  $$c^2 = c_3 y^6 + c_2 y^4 + c_1 y^2 + c_0 = c_3 y^3 + (c_3 + c_1) y^2 + c_2 y + (c_2 + c_0),$$

  where $y^6 = y^3 + y^2 \bmod Q(y)$ and $y^4 = y + 1 \bmod Q(y)$. The complexity of the operation is 2 XOR.

The overall complexity for parallel inversion in $GF((2^4)^2)$ is obtained by summation of the partial complexities. By denoting the complexity with C_{2^4} we get:

$$C_{2^4} < (23 + 45 + 8 + 1 + 2)\,\text{XOR} + (22 + 48)\,\text{AND} = 79\,\text{XOR} + 70\,\text{AND}$$

This complexity is remarkably low. It is interesting to compare this complexity with standard base *multiplication*. The Mastrovito multiplier, which is one of the best standard base architectures, has a gate count of 84 XOR/64 AND (see Table 3.1.) We conclude that inversion in $GF(2^8)$ can be performed with almost the same gate count as multiplication, if composite fields are introduced.

8.5.2 A Parallel Inverter over $GF(2^{16})$

For inversion in the field $GF(2^{16})$ we choose a decomposition into $GF((2^4)^4)$. A decomposition into $GF((2^8)^2)$ is *not* advisable, because this would require direct inversion in $GF(2^8)$ which is already very costly if direct inversion is applied.

The primitive field polynomials chosen are $Q(y) = y^4+y+1$ and $P(x) = x^4+x^3+x^2+p_0$. Again we can apply the Mastrovito architecture to the ground field multiplication. For multiplication in $GF((2^4)^4)$, the architecture developed in Chapter 6 will be used. Table 6.2 shows that the complexity of the multiplier is 144 XOR and 238 AND gates.

We apply the inversion algorithm described in Section 8.2. The parameter for the decomposition is $m = 4$. In the sequel, the gate counts of the four steps of the algorithm are evaluated.

- According to Equation (8.6), there are 36 constant multiplications, 9 additions and 3 general multiplications required for Step 1. The general multiplications refer to arithmetic in the composite field $GF((2^4)^4)$, the two first types of operation to arithmetic in the ground field $GF(2^4)$. In order to find a measure for the constant multiplication, we use the optimized average complexity given in Table 4.1. For this ground field, it is 3.3 XOR gates per constant multiplication. Using this estimation, we obtain a complexity for Step 1 of $36 \cdot 3.6 + 9 \cdot 4 + 3 \cdot 144 = 658$ XOR and $3 \cdot 238 = 714$ AND.

- For Step 2, the optimized complexity developed in the example in Subsection 8.3.2 is valid. It is 5 multiplications, 4 additions and 2 constant multiplications. All arithmetic is performed in the ground field. Hence the complexity equals $5 \cdot 15 + 4 \cdot 4 + 2 \cdot 3.3 = 98$ XOR and $5 \cdot 16 = 80$ AND.

- As in the previous example, an upper estimate for one inversion in the ground field, required for Step 3, is 23 XOR and 22 AND.

- The four ground field multiplications required by Step 4 have a complexity of $4 \cdot 15 = 60$ XOR and $4 \cdot 16 = 64$ AND.

The overall gate count is obtained by summation of the partial complexities. By denoting the complexity with $C_{2^{16}}$ we get:

$$C_{2^{16}} < 839 \text{ XOR} + 880 \text{ AND}.$$

As expected, the complexity has increased dramatically if compared to the inverter over $GF(2^8)$. However, the complexity is still in the range of the complexities of the multipliers which were synthesized in Chapter 7. As matter of fact, the multiplier over $GF((2^8)^4)$, which was synthesized, has a theoretical gate count 576 XOR and 998 AND. This complexity is comparable to the gate count of the inverter. Since the synthesis resulted in an actual gate consumption of 3554 gate equivalences, we can expect that the inverter will have a similar area complexity. This rough estimation implies that parallel implementations of inverters over $GF(2^{16})$ are still possible.

Chapter 9

An Application: A DSP Based Reed-Solomon Decoder with External Arithmetic Unit

Parts of this chapter were presented in [PH94].

9.1 Motivation

In this chapter the implementation of a Reed-Solomon decoder) (RS decoder) on a digital signal processor (DSP) is discussed. The DSP has an external, field programmable gate array (FPGA) attached, containing the parallel finite field multiplier over $GF((2^4)^4)$ which was developed in Chapter 5. We will show that the external FPGA enables the processor to perform Galois field multiplication more than one magnitude faster than in software.

The major goals which were pursued by the implementation can be described as follows:

- Development and verification of a new concept for systems involving finite field operations, consisting of a general purpose processor and a dedicated external finite field arithmetic unit.
- Verification of the architectures for parallel multiplication over composite fields developed in this thesis.
- Development of an RS code operating with symbols longer than 8 bits and short block length.
- Implementation of a reasonably fast ($\geq$ 1 Mbps) RS decoder which is reprogrammable and whose code parameters are thus alterable.

9.2 Introduction

Reed-Solomon codes are error control codes, belonging to the important class of cyclic codes [LC83] [Bla83]. They are a special case of BCH (Bose-Chaudhury-Hocquenghem) codes. RS codes are, in addition to cryptography and signal processing, one of the most important technical areas where finite fields are applied. Over the last twenty years they have gained widespread application, ranging from space communication [LL84] [WHPH87] [PRM90] to error correction on compact discs [Pee85] [SI91]. RS codes perform arithmetic in Galois fields of the form $GF(2^k)$, where each field element is represented by k binary bits. The vast majority of applications so far has operated over fields with $k \leq 8$ [SI91] [Mes91]; RS codes over $GF(2^8)$ were actually standardized as part of a concatenated coding scheme of the ESA (European Space Agency) and NASA [Kum83]. On the other hand, today's digital systems tend to possess binary word lengths that are longer than 8 bits. Typical are 16 or 32 bits, while an extension to 64 or more bits is expected in the near future. RS codes which operate over Galois fields with field order exponents k equal to 16 or 32 are therefore certainly attractive for many applications.

Another advantage of an increased symbol length matching the bus width of today's processors is, that faster software implementations of RS decoders and RS encoders become possible. Although the use of a multi purpose processor as a decoder limits the possible data throughput to one or two magnitudes of a VLSI solution [PD90] [Mes91], a DSP inhibits several advantages. In addition to the shortened development time and costs, programmable decoders offer much more flexibility if changes in the coding scheme become necessary. An impressive example of the drawbacks of fixed coding schemes is given by the Galileo spacecraft's flight to the planet Jupiter, and the problems caused by its non-unfolding antenna [CDD+93].

9.3 Implementational Aspects

9.3.1 Code Specification and Decoding Algorithm

The parameters of the implemented code are as follows:

- RS-code characteristics:
  $n = 10$: number of symbols per RS codeword
  $k = 8$: number of information symbols per RS codeword
  $t = 1$: number of symbol errors that can be corrected per RS codeword
  code rate : 0.8

- The generator polynomial of the the RS-code is:

$$g(X) = (X - \alpha^0)(X - \alpha^1) = (X - 1)(X - \alpha), \tag{9.1}$$

  with α being a primitive element of $GF((2^4)^4)$. The pair of consecutive roots $\{1, \alpha\}$ of this generator polynomial were taken as suggested in [Ber82]. They indeed proved

to be optimal with respect to the encoder complexity (gate count) after an exhaustive search through all possible pairs of roots.

- Arithmetic is performed in the composite field $GF((2^4)^4)$, with $Q(y) = y^4 + y + 1$ and $P(x) = x^4 + x^3 + x^2 + \omega$ being the field polynomials of the ground and extension field, respectively.

Standard RS-codes over a field $GF(q)$ have a code word length n that is equal to $n = q - 1$ [Bla83], [LC83]. Using a field $GF(2^{16})$ would result in a code word length of $(2^{16} - 1) \times 16\text{bit} = 131$ kByte. This kind of block size is extremely difficult to handle with respect to the required memory and introduced delay. Therefore the approach taken here uses a shortened RS-code.

In the case of a shortened RS-code only $n < q - 1$ symbols of the code word are used to carry information; the others are considered to be zero symbols. If a systematic code is used, the decoder has the à priori information about which of the information symbols are in fact the zero symbols. These symbols carry no information and can therefore be neglected in decoding algorithms.

Shortened RS-codes are, just like standard RS-codes, Maximum Distance Separable (MDS) Codes, because they meet the Singleton bound with equality: $d_{min} = n - k + 1$ [Bla83]. With a symbol error probability equal to p_s and using the weight distribution of MDS-codes [ML85], the undetected error probability for a (n,k) (shortened) RS-codes over $GF(q)$ is upperbounded by :

$$P_U(E) = \sum_{i=0}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) (q-1)^{1-i} \sum_{j=0}^{i-d} (-1)^j \left(\begin{array}{c} i-1 \\ j \end{array} \right) q^{i-d-j} p_s^i (1-p_s)^{n-i},$$

In order to keep the introduced decoding delay low, we constructed a code with a relatively short block length of 10 symbols and $d_{min} = n - k + 1 = 3$. To avoid the time consuming solving of the key equations by means of Euclid's or Massey/Berlekamp's algorithm, a design parameter of $t = 1$ was chosen. Using a code with $t = 1$ results in a pair of syndromes (9.2) and (9.3), which allow the determination of error location and error magnitude by a direct method.

Code words are multiples of the generator polynomial (9.1). At the decoder we receive words $r(X)$, which can be considered code words $c(X)$ with possible error words e(X) added to them:

$$r(X) = c(X) + e_i X^i, \qquad 0 \le i \le 9.$$

Evaluation of this received word at the roots of the generator polynomial $g(X)$, gives the syndromes:

$$\begin{aligned} Y &:= r(1) = c(1) + e_i 1^i = e_i = S_1 \\ &= r_9 + r_8 + \cdots + r_1 + r_0 = \textstyle\sum_{i=0}^{9} r_i \end{aligned} \tag{9.2}$$

and

$$\begin{aligned} YX &:= r(\alpha) = c(\alpha) + e_i \alpha^i = e_i \alpha^i = S_2 \\ &= (\cdots (r_9 \alpha + r_8)\alpha \cdots + r_1)\alpha + r_0, \end{aligned} \tag{9.3}$$

with $Y := e_i$ (, $e_i \in GF(2^{16})$) being the error value and $X := \alpha^i$ indicating an error at position i [Bla83].

With the calculation of S_1, which takes 9 additions (i.e. bitwise XOR,) the value Y of the error is known. Furthermore, when this value is zero, we assume that no error has occurred. For the calculation of S_2 in (9.3), which is actually the evaluation of a polynomial with known powers x^j and variable coefficients r_j, Horner's rule [Knu81] is well suited. In this particular case it has a complexity of 9 additions and 9 multiplications.

In the second decoding step the error position X can be found by systematically trying which one of the possible values of X satisfies $S_2 = YX$, with S_2 and Y given. The complexity of this method is in the worst case, i.e. an error at position $i = 7$, eight multiplications. When none of the possible X satisfy the equation, an uncorrectable error (i.e. more than two symbol errors) is detected. Using this kind of systematic search we avoid the inversion of Y, (i.e. $X = S_2 \times Y^{-1}$) which would take 15 multiplications and 16 squaring operations if the standard binary method for exponentiation is applied [Knu81].

9.3.2 The Hardware Concept

As mentioned in the introduction, the decoder was implemented on a DSP TMS320 C25 from Texas Instruments which has internally and externally a 16 bit wide bus. The TMS320 C25 is an enhanced version of the TMS320 20 with an instruction cycle of $100ns$ when running at full speed (40 MHz.) In the field $GF(2^{16})$, addition is simply performed by a bitwise XOR of the two operands which takes only one instruction cycle. In order to overcome the bottleneck imposed by slow finite field multiplication in software, an external multiplier was attached. A block diagram of the decoder is shown in Figure 9.1.

From the available methods for finite field multiplication – table look-up, calculation in software, serial hardware multiplication and parallel hardware multiplication – only the last one is suited for this application. The frequently applied table look-up [TM90], [YACD89] would easily exceed the memory available on chip for the field size used. External memory would not only be penalized by a higher access time, but will not even be sufficient if the proposed system design is generalized for processors with longer word size. For instance, if TI's third of forth generation DSPs with 32 bit are applied. In general, the memory size required for a table look up in $GF(2^k)$ is of $\mathcal{O}(2^k)$, where k is also the processor's word size.

We used the multiplier architecture proposed in Chapter 5 which has a gate count of 144 AND / 258 XOR and which could easily be implemented on a XC3142 FPGA from Xilinx. The multiplier chip is located on a printed circuit board which can be accessed by the DSP via its regular I/O ports through fast interface logic. However, the access time of the DSP I/O ports by assembler instructions is rather long (both read and write need 2 machine cycles, i.e. $200ns$), such that the overall time for performing an entire multiplication adds up to $700ns$ when using indexed addressing, or even $1000ns$ when multiplying two numbers that were not previously in registers.

Figure 9.1: Block diagram of the DSP based RS decoder with external finite field multiplier

Hardware		*Software*			
		C-library			*MAPLE*
pure FPGA	TMS320C25 + FPGA	IBM RS6000/580	80486DX2 66 MHz	TMS320C25 (estimated)	IBM RS6000/580
80*ns*	700 *ns*	4.8 μs	6.1 μs	12 μs	2.6 *ms*

Table 9.1: Speed comparison of various methods for general multiplication in the finite field $GF(2^{16})$

9.4 Results and Comparison

Table 9.1 shows a comparison of hardware/software solutions for general multiplication in $GF(2^{16})$. For reasons stated above, we assume that table look-up is not feasible and therefore all methods given in the table *calculate* the result of the multiplication. The software methods are based on a self written C-library with an especially optimized multiplication routine and the multi purpose program MAPLE for algebraic computation. For a generalization of the results for larger fields it should be noted that the speed of the library function increases at least proportionally with the logarithm of the field order 2^k. For instance, multiplication in $GF(2^{32})$ would at least double the multiplication time.

The maximum data rate of the decoder is limited by the time required for the decoding algorithm plus some overhead for the interrupt handling. For the system implemented, a CD based signal was used which results in a clock speed of $10/8 \times 1.41$Mbps $= 1.76$Mbps. This correspondents to 91 μs for the receiving of an entire block of 160 bits. The DSP needs 68 μs for input, output, and decoding in a worst case situation, which is 75 % of the available 91 μs. Therefore the data rate can be increased theoretically up to 1.9 Mbps. The DSP I/O and decoding process cause a delay of two blocks or 320 bits, which is equal to approximately 0.2 milliseconds.

Table 9.2 shows a comparison of our system with the two DSP-based RS-decoders proposed in [TM90] and [YACD89]. These decoders apply table look-up for multiplication in the fields $GF(2^8)$ and $GF(2^4)$, respectively. The coding schemes implemented are standard, i.e. non-shortened, RS-codes. We are aware of the fact that it is difficult to provide a fair comparison due to progress in processor technology and due to different code parameters used. However, the significantly increased data rate of our approach compared to those in [TM90], [YACD89] seems to prove the success of the new system design, using dedicated external arithmetic units that allow arithmetic in fields which match the processor's bus structure.

A complete version of the system, including an FPGA encoder and a simulated channel which corrupts digital data from a compact disc player at a speed of 1.76 Mbps, was exhibited in the research section of the CeBIT computer fair, held in March 1994 in Hannover, Germany.

System	DSP	Coding Scheme	Code rate	Maximum data rate	Delay	Decoding algorithm
Proposed	Texas Inst. TMS320C25	(10,8) over $GF(2^{16})$	0.80	1.9 Mbps	0.2 ms	Direct solution
[TM90] 1990	NEC μPD77220	(255,223) over $GF(2^8)$	0.87	275 kbps	14.5 ms	Euclid
[YACD89] 1989	Texas Inst. TMS32010	(15,k) over $GF(2^4)$	0.67 - 0.93	16 - 80 kbps	4.4 - 0.9 ms	Massey-Berlekamp

Table 9.2: Comparison of DSP based RS decoders

9.5 Outlook

Our approach looks promising for an extension of the error correction capability to $t > 1$. For this, in addition to the multiplier, a fast external finite field divider/inverter such as proposed in Chapter 3.3 should be attached to the DSP. This would make more general and more powerful DSP based decoders possible. The inverter would allow a fast solution of the key equations either through Euclid's algorithm, Massey-Berlekamp's algorithm or an extended version of the direct methods such as the algorithms proposed in [DZ85]. A further improvement can be made if the access time of the external hardware is accelerated. We used the standard I/O ports of the DSP, but a sophisticated DMA-based access scheme may result in faster multiplication/inversion.

Chapter 10

Discussion

10.1 Summary and Conclusions

This thesis describes various bit parallel VLSI architectures for computation in Galois fields of characteristic two. The arithmetic functions considered are: multiplication with a constant, general multiplication and inversion. The architectures make extensive use of a decomposition of fields $GF(2^k)$ into a subfield $GF(2^n)$, together with an extension of degree m, where $nm = k$. These fields are referred to as "composite fields." The architectures are based on algorithms which lead to small theoretical gate counts.

The architectures use a polynomial representation of composite fields elements. This means, the elements are represented by polynomials with a maximum degree of $m - 1$ and coefficients of $GF(2^n)$. Thus, computation in $GF((2^n)^m)$ is performed by applying arithmetic modules from $GF(2^n)$. This setup possesses several natural advantages for VLSI implementations:

- The architectures are modular, with modules performing $GF(2^n)$ arithmetic with well defined functions and interfaces.
- Since multiplication is considerably more "costly" than addition in $GF(2^n)$ in terms of gate count and delay, efficient algorithms known from integer arithmetic can be applied to arithmetic in $GF((2^n)^m)$. This may result in an improved gate count.
- Since the complexity of inversion in finite fields $GF(2^k)$ increases dramatically with the field order, algorithms over composite field which reduce inversion in $GF((2^n)^m)$ to inversion in $GF(2^n)$ are potentially very efficient.

There are two efficient algorithms described which can be used as tools for the practical application of composite field architectures. The first algorithm describes a linear mapping between a binary (traditional) field representation and a composite field representation, such that the architectures introduced here can be used together with other architectures. The second algorithm performs a test in order to determine whether a

polynomial over $GF(2^n)$ is primitive. Polynomials which pass the test can be used for constructing composite fields. The latter algorithm seems to be especially useful, since tables of these polynomials are very rare in literature. The tables available usually contain irreducible polynomials over $GF(2)$ or polynomials over other prime fields; one of the few exceptions is reference [GT74].

After the introduction of a locally optimum algorithm for gate optimization of constant multipliers, different architectures for multiplication of two arbitrary elements in composite fields are developed. A general architecture can be applied to fields $GF((2^n)^m)$, where m is a power of two. The architecture is based on the Karatsuba-Ofman algorithm, whose application to the multiplication of polynomials over fields $GF(2^n)$ is studied in detail. It is shown that the computational complexity of this operation is of order $\mathcal{O}((nm)^{\log_2 3})$, and the theoretical delay of order $\mathcal{O}(\log_2(nm))$. Applying an exhaustive search results in primitive polynomials which perform modulo reduction in fields up to $GF(2^{32})$ with low complexity. For two types of composite fields, those with the fixed compositions $GF((2^n)^2)$ and $GF((2^n)^4)$, improved architectures are provided. For all field orders, architectures were found that are below the $2k^2-1$ lower complexity bound of traditional architectures. Moreover, the complexities are also below the complexities of an architecture that applies the Karatsuba-Ofman algorithm to elements in a binary field representation [Afa90]. The multiplier over $GF(2^8)$, introduced in Section 6.1, has the lowest gate count reported in technical literature. For larger fields, the architectures perform slightly worse than those over tower fields, but possess the advantage of a higher modularity.

A VLSI synthesis compares three traditional multipliers, which uses standard, dual, and normal base representation, respectively, with composite field multipliers. The synthesis performs an automatic mapping of the architectures to a sea-of-gates chip. The major result is that the theoretically improved gate count of the composite field multiplier can be transformed to actual gate array implementations under the given conditions. We conclude that the proposed multiplier architectures are of great interest for technical applications.

An algorithm of Itoh and Tsujii for inversion in composite fields is applied to elements in standard base representation. The algorithm reduces inversion in $GF((2^n)^m)$ to inversion in the ground field $GF(2^n)$. The algorithms is divided into four steps, each of which are investigated with respect to space complexity. It is proposed that inversion in $GF(2^n)$ is performed by a direct method. Two examples show that parallel inversion with elements in standard base is possible with a surprisingly low gate count.

A new concept for systems involving finite field arithmetic is proposed. The concept is based on a general purpose processor together with dedicated hardware for computation in finite fields. As an application for the parallel architectures over relatively large fields developed in this thesis, a fast Reed-Solomon (RS) decoder with 16 bit symbols was implemented on a digital signal processor. The processor has an external finite field multiplier over $GF(2^{16})$ attached. It is shown that a shortened (10,8) RS code allows a decoding speed of up to 1.9 MHz/sec.

10.2 Recommendations for Further Research

It has been demonstrated in this thesis that it is advantageous to apply composite fields to bit parallel VLSI architectures. During the research, several questions regarding the architectures presented arose which are as yet unanswered. Moreover, some extensions of the results can be suggested.

1. Chapter 5 develops a general architecture for multiplication over $GF((2^n)^m)$. Whereas it was possible to provide general expression for the complexity of the multiplication of two polynomials over $GF(2^n)$, field polynomials $P(x)$ which perform the operation $\mathrm{mod} P(x)$ with low complexity were determined through an exhaustive search. It would be interesting to provide general expression for modulo reduction with *low complexity* as well. A possible approach would be to investigate the existence of field polynomials over $GF(2^n)$ with low coefficient weight.

2. In Chapter 6, only the existence of Type I polynomials over $GF(2^n)$, n odd, could be proved. We could not find a similar proof of existence for Type II polynomials over $GF(2^n)$, n even. Since the fields $GF((2^n)^4)$, n even, are especially interesting for technical applications, it would be helpful to provide such a proof.

3. It seems interesting that Type I polynomials do not exist over fields $GF(2^n)$ when $n = 2, 4, 6, 8$. These observations suggest a further study of the question *"Do Type I polynomials exist only over fields $GF(2^n)$ with n odd"?* An answer to this question would extend Lemma 1 on page 66. A more generalized question is: *"Do primitive trinomials of degree four exist over fields $GF(2^n)$ with n even?"* In order to answer this question it might be helpful to study [Gol67], where the non-existence of binary trinomials of degree $8i$, i integer, is stated.

4. Both types of architectures in Chapter 6, over fields $GF((2^n)^2)$ and $GF((2^n)^4)$, improve the general architecture of Chapter 5 over $GF((2^n)^m)$ by combining the third stage of the Karatsuba-Ofman Algorithm and the modulo reduction. An extension of this approach to composite fields with $m = 8, 16, \ldots$ seems promising.

5. Up to which field order the parallel multiplier developed in Chapter 5 can realistically be implemented in VLSI should be investigated. For instance, is parallel multiplication in fields of order $\approx 2^{500}$ possible?

6. A locally optimum algorithm for optimized constant multiplication over $GF(2^n)$ was introduced in Chapter 4. In order to obtain solutions which are *globally* better, it might be worth while to apply the principle of simulated annealing [AK89] to the algorithm.

7. In [Mas91, Chapter 6] hybrid multipliers were introduced. These multipliers are architectures based on composite fields which perform the ground field multiplication in parallel, but perform the extension field algorithm in a serial manner. In

particular, hybrid architectures seem to be attractive for arithmetic in Galois fields of very higher order, such as needed for some applications in cryptography. With the material provided in this thesis, the construction of hybrid multipliers over large Galois fields should be possible: The algorithm of Section 2.3 can be used for the determination of suitable field polynomials; the results of the gate array synthesis described in Chapter 7 can be used for an estimation of the time performance.

8. As stated in Section 9.5, a programmable RS decoder should be implemented which, in addition to a finite field multiplier, has also a parallel hardware inverter attached. What is the achievable speed of the hybrid software/hardware decoder?

Appendix A

Direct Inversion in $GF(2^n)$

The following formulas describe direct inversion in the fields $GF(2^n)$, $n = 3, 4, 5, 6, 7$. The notation of the variables refers to

$$B = (b_{n-1}, \ldots, b_1, b_0) = A^{-1} = (a_{n-1}, \ldots, a_1, a_0)^{-1},$$

where $A, B \in GF(2^n)$. The formulas are not optimized, i.e. they contain redundancies. For an implementation, a gate optimization is recommended. The field polynomials are given in Table 3.1.

1. Equations for inversion in $GF(2^3)$:

$$\begin{aligned} b_0 &= a_0 + a_1 + a_2 + a_1a_2 \\ b_1 &= a_0a_1 + a_2 \\ b_2 &= a_1 + a_2 + a_0a_2 \end{aligned}$$

2. Equations for inversion in $GF(2^4)$:

$$\begin{aligned} b_0 &= a_0 + a_1 + a_2 + a_0a_2 + a_1a_2 + a_0a_1a_2 + a_3 + a_1a_2a_3 \\ b_1 &= a_0a_1 + a_0a_2 + a_1a_2 + a_3 + a_1a_3 + a_0a_1a_3 \\ b_2 &= a_0a_1 + a_2 + a_0a_2 + a_3 + a_0a_3 + a_0a_2a_3 \\ b_3 &= a_1 + a_2 + a_3 + a_0a_3 + a_1a_3 + a_2a_3 + a_1a_2a_3 \end{aligned}$$

3. Equations for inversion in $GF(2^5)$:

$$\begin{aligned} b_0 &= a_0 + a_1 + a_2 + a_1a_2 + a_3 + a_1a_3 + a_0a_2a_3 + a_1a_2a_3 + a_4 + a_0a_1a_4 + a_2a_4 + \\ &\quad a_1a_2a_4 + a_3a_4 + a_1a_3a_4 + a_2a_3a_4 + a_1a_2a_3a_4 \\ b_1 &= a_0a_1 + a_0a_2 + a_1a_2 + a_0a_1a_2 + a_0a_1a_3 + a_2a_3 + a_0a_1a_2a_3 + a_4 + a_1a_4 + a_0a_1a_4 \\ &\quad + a_1a_2a_4 + a_0a_3a_4 \\ b_2 &= a_0a_1 + a_1a_2 + a_3 + a_0a_3 + a_0a_1a_3 + a_4 + a_0a_4 + a_1a_4 + a_0a_1a_4 + a_2a_4 + a_0a_2a_4 \\ &\quad + a_1a_2a_4 + a_0a_1a_2a_4 + a_0a_3a_4 + a_1a_3a_4 + a_2a_3a_4 \end{aligned}$$

$$
\begin{aligned}
b_3 &= a_0a_1 + a_2 + a_3 + a_0a_3 + a_1a_3 + a_0a_1a_3 + a_0a_2a_3 + a_1a_2a_3 + a_4 + a_0a_4 + a_1a_4 \\
&\quad + a_0a_2a_4 + a_3a_4 + a_0a_3a_4 + a_1a_3a_4 + a_0a_1a_3a_4 \\
b_4 &= a_1 + a_2 + a_0a_2 + a_0a_1a_2 + a_3 + a_0a_3 + a_2a_3 + a_0a_2a_3 + a_4 + a_0a_4 + a_2a_4 \\
&\quad + a_0a_2a_4 + a_1a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4
\end{aligned}
$$

4. Equations for inversion in $GF(2^6)$:

$$
\begin{aligned}
b_0 &= a_0 + a_1 + a_2 + a_0a_2 + a_1a_2 + a_0a_1a_2 + a_3 + a_0a_3 + a_0a_2a_3 + a_4 + a_1a_4 \\
&\quad + a_0a_1a_4 + a_0a_2a_4 + a_1a_2a_4 + a_0a_1a_2a_4 + a_3a_4 + a_1a_3a_4 + a_0a_1a_3a_4 + \\
&\quad a_2a_3a_4 + a_0a_2a_3a_4 + a_1a_2a_3a_4 + a_0a_1a_2a_3a_4 + a_5 + a_1a_5 + a_0a_1a_5 + a_0a_2a_5 + \\
&\quad a_1a_2a_5 + a_3a_5 + a_1a_3a_5 + a_2a_3a_5 + a_0a_4a_5 + a_1a_4a_5 + a_2a_4a_5 + a_3a_4a_5 + \\
&\quad a_0a_3a_4a_5 + a_1a_3a_4a_5 + a_2a_3a_4a_5 + a_1a_2a_3a_4a_5 \\
b_1 &= a_0a_1 + a_0a_2 + a_1a_2 + a_0a_1a_2 + a_0a_1a_3 + a_2a_3 + a_1a_2a_3 + a_0a_1a_2a_3 + a_0a_4 \\
&\quad + a_1a_4 + a_2a_4 + a_0a_1a_2a_4 + a_3a_4 + a_0a_3a_4 + a_0a_1a_3a_4 + a_5 + a_2a_5 + a_0a_2a_5 \\
&\quad + a_3a_5 + a_0a_3a_5 + a_0a_1a_3a_5 + a_2a_3a_5 + a_0a_2a_3a_5 + a_1a_2a_3a_5 + a_0a_1a_2a_3a_5 \\
&\quad + a_1a_4a_5 + a_3a_4a_5 + a_0a_3a_4a_5 + a_1a_3a_4a_5 + a_2a_3a_4a_5 \\
b_2 &= a_0a_1 + a_1a_2 + a_0a_1a_2 + a_0a_3 + a_1a_3 + a_2a_3 + a_4 + a_1a_4 + a_2a_4 + a_1a_2a_4 \\
&\quad + a_0a_1a_2a_4 + a_0a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_1a_2a_3a_4 + a_5 + a_0a_5 + a_1a_5 \\
&\quad + a_0a_1a_5 + a_2a_5 + a_0a_2a_5 + a_0a_1a_2a_5 + a_1a_3a_5 + a_0a_1a_3a_5 + a_0a_2a_3a_5 \\
&\quad + a_1a_4a_5 + a_0a_2a_4a_5 + a_1a_2a_4a_5 + a_0a_1a_2a_4a_5 + a_3a_4a_5 \\
b_3 &= a_0a_1 + a_0a_2 + a_1a_2 + a_3 + a_0a_3 + a_1a_3 + a_0a_1a_3 + a_1a_2a_3 + a_0a_1a_2a_3 + a_4 \\
&\quad + a_0a_4 + a_1a_4 + a_0a_2a_4 + a_0a_3a_4 + a_0a_1a_3a_4 + a_2a_3a_4 + a_5 + a_0a_5 + a_0a_1a_5 \\
&\quad + a_3a_5 + a_0a_1a_3a_5 + a_2a_3a_5 + a_4a_5 + a_0a_4a_5 + a_0a_1a_4a_5 + a_0a_2a_4a_5 \\
&\quad + a_1a_2a_4a_5 + a_3a_4a_5 + a_1a_3a_4a_5 + a_0a_1a_3a_4a_5 \\
b_4 &= a_0a_1 + a_2 + a_0a_2 + a_0a_1a_2 + a_3 + a_0a_3 + a_1a_2a_3 + a_4 + a_2a_4 + a_1a_2a_4 + a_3a_4 \\
&\quad + a_0a_1a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_5 + a_0a_5 + a_2a_5 + a_0a_2a_5 + a_0a_1a_2a_5 \\
&\quad + a_1a_3a_5 + a_0a_2a_3a_5 + a_0a_4a_5 + a_2a_4a_5 + a_0a_2a_4a_5 + a_0a_3a_4a_5 + a_0a_2a_3a_4a_5 \\
b_5 &= a_1 + a_2 + a_0a_1a_2 + a_3 + a_1a_3 + a_0a_1a_3 + a_2a_3 + a_1a_2a_3 + a_4 + a_1a_4 + a_0a_2a_4 \\
&\quad + a_1a_3a_4 + a_5 + a_0a_5 + a_0a_1a_5 + a_2a_5 + a_1a_2a_5 + a_1a_3a_5 + a_2a_3a_5 + a_0a_2a_3a_5 \\
&\quad + a_1a_2a_3a_5 + a_4a_5 + a_0a_1a_4a_5 + a_2a_4a_5 + a_1a_2a_4a_5 + a_3a_4a_5 + a_1a_3a_4a_5 \\
&\quad + a_2a_3a_4a_5 + a_1a_2a_3a_4a_5
\end{aligned}
$$

5. Equations for inversion in $GF(2^7)$:

$$
\begin{aligned}
b_0 &= a_0 + a_1 + a_2 + a_1a_2 + a_3 + a_1a_3 + a_0a_1a_2a_3 + a_4 + a_1a_4 + a_2a_4 + a_1a_2a_4 \\
&\quad + a_3a_4 + a_0a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_1a_2a_3a_4 + a_5 + a_0a_5 + a_0a_1a_5 \\
&\quad + a_2a_5 + a_0a_2a_5 + a_1a_2a_5 + a_0a_1a_2a_5 + a_0a_3a_5 + a_1a_3a_5 + a_0a_1a_3a_5
\end{aligned}
$$

$$
\begin{aligned}
& +a_2a_3a_5 + a_0a_1a_4a_5 + a_0a_2a_4a_5 + a_1a_2a_4a_5 + a_0a_2a_3a_4a_5 + a_1a_2a_3a_4a_5 \\
& +a_6 + a_0a_1a_6 + a_2a_6 + a_3a_6 + a_0a_3a_6 + a_2a_3a_6 + a_0a_2a_4a_6 + a_1a_2a_4a_6 \\
& +a_3a_4a_6 + a_1a_3a_4a_6 + a_0a_1a_3a_4a_6 + a_1a_2a_3a_4a_6 + a_5a_6 + a_0a_5a_6 \\
& +a_0a_1a_5a_6 + a_1a_2a_5a_6 + a_0a_1a_2a_5a_6 + a_3a_5a_6 + a_0a_3a_5a_6 + a_2a_3a_5a_6 \\
& +a_1a_2a_3a_5a_6 + a_4a_5a_6 + a_1a_4a_5a_6 + a_1a_2a_4a_5a_6 + a_3a_4a_5a_6 + a_1a_3a_4a_5a_6 \\
& +a_2a_3a_4a_5a_6 + a_1a_2a_3a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_1 \;=\;& a_0a_1 + a_0a_2 + a_1a_2 + a_0a_3 + a_0a_1a_3 + a_2a_3 + a_1a_2a_3 + a_0a_1a_4 + a_2a_4 \\
& +a_0a_2a_4 + a_3a_4 + a_0a_3a_4 + a_0a_1a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_1a_2a_3a_4 \\
& +a_0a_1a_2a_3a_4 + a_0a_5 + a_1a_5 + a_0a_1a_5 + a_0a_2a_5 + a_0a_1a_2a_5 + a_0a_3a_5 \\
& +a_1a_3a_5 + a_0a_2a_3a_5 + a_0a_1a_2a_3a_5 + a_4a_5 + a_2a_4a_5 + a_0a_2a_4a_5 \\
& +a_0a_1a_2a_4a_5 + a_3a_4a_5 + a_0a_3a_4a_5 + a_0a_1a_3a_4a_5 + a_2a_3a_4a_5 \\
& +a_0a_1a_2a_3a_4a_5 + a_6 + a_1a_6 + a_0a_1a_6 + a_1a_2a_6 + a_0a_1a_2a_6 + a_0a_3a_6 \\
& +a_1a_3a_6 + a_0a_4a_6 + a_1a_4a_6 + a_0a_1a_4a_6 + a_2a_4a_6 + a_1a_2a_4a_6 + a_0a_3a_4a_6 \\
& +a_1a_3a_4a_6 + a_0a_1a_3a_4a_6 + a_1a_2a_3a_4a_6 + a_0a_1a_5a_6 + a_2a_5a_6 + a_1a_2a_5a_6 \\
& +a_0a_1a_2a_5a_6 + a_3a_5a_6 + a_0a_3a_5a_6 + a_2a_3a_5a_6 + a_0a_2a_3a_5a_6 + a_4a_5a_6 \\
& +a_0a_4a_5a_6 + a_1a_4a_5a_6 + a_0a_1a_4a_5a_6 + a_2a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_2 \;=\;& a_0a_1 + a_1a_2 + a_0a_1a_2 + a_1a_3 + a_2a_3 + a_1a_2a_3 + a_0a_1a_2a_3 + a_0a_4 + a_0a_2a_4 \\
& +a_3a_4 + a_1a_3a_4 + a_2a_3a_4 + a_1a_2a_3a_4 + a_5 + a_0a_5 + a_0a_1a_5 + a_0a_2a_5 + a_0a_3a_5 \\
& +a_1a_3a_5 + a_0a_1a_3a_5 + a_0a_2a_3a_5 + a_0a_1a_2a_3a_5 + a_1a_4a_5 + a_0a_1a_4a_5 + a_2a_4a_5 \\
& +a_1a_2a_4a_5 + a_3a_4a_5 + a_0a_3a_4a_5 + a_1a_3a_4a_5 + a_6 + a_0a_6 + a_0a_2a_6 + a_1a_2a_6 \\
& +a_3a_6 + a_0a_3a_6 + a_1a_3a_6 + a_0a_2a_3a_6 + a_1a_2a_3a_6 + a_0a_1a_2a_3a_6 + a_4a_6 \\
& +a_0a_4a_6 + a_0a_1a_4a_6 + a_0a_1a_2a_4a_6 + a_0a_1a_3a_4a_6 + a_0a_2a_3a_4a_6 + a_1a_2a_3a_4a_6 \\
& +a_0a_1a_2a_3a_4a_6 + a_0a_5a_6 + a_0a_1a_5a_6 + a_3a_5a_6 + a_0a_3a_5a_6 + a_1a_3a_5a_6 \\
& +a_2a_3a_5a_6 + a_0a_2a_3a_5a_6 + a_1a_2a_3a_5a_6 + a_4a_5a_6 + a_0a_4a_5a_6 + a_0a_1a_4a_5a_6 \\
& +a_2a_4a_5a_6 + a_1a_2a_4a_5a_6 + a_3a_4a_5a_6 + a_0a_3a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_3 \;=\;& a_0a_1 + a_0a_2 + a_1a_2 + a_0a_1a_2 + a_2a_3 + a_0a_2a_3 + a_1a_2a_3 + a_0a_1a_2a_3 + a_4 \\
& +a_0a_2a_4 + a_0a_3a_4 + a_1a_3a_4 + a_0a_1a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_5 + a_0a_1a_5 \\
& +a_2a_5 + a_0a_2a_5 + a_0a_1a_2a_5 + a_3a_5 + a_0a_1a_2a_3a_5 + a_2a_4a_5 + a_0a_2a_4a_5 \\
& +a_1a_2a_4a_5 + a_0a_1a_2a_4a_5 + a_3a_4a_5 + a_1a_3a_4a_5 + a_0a_1a_3a_4a_5 + a_2a_3a_4a_5 \\
& +a_6 + a_0a_6 + a_0a_1a_6 + a_2a_6 + a_0a_1a_2a_6 + a_3a_6 + a_1a_3a_6 + a_2a_3a_6 \\
& +a_1a_2a_3a_6 + a_0a_4a_6 + a_1a_4a_6 + a_2a_4a_6 + a_0a_2a_4a_6 + a_0a_1a_2a_4a_6 \\
& +a_0a_3a_4a_6 + a_1a_3a_4a_6 + a_5a_6 + a_1a_2a_5a_6 + a_0a_1a_2a_5a_6 + a_0a_3a_5a_6 \\
& +a_1a_3a_5a_6 + a_0a_1a_3a_5a_6 + a_2a_3a_5a_6 + a_0a_2a_3a_5a_6 + a_1a_2a_3a_5a_6 \\
& +a_0a_1a_2a_3a_5a_6 + a_0a_4a_5a_6 + a_0a_3a_4a_5a_6 + a_1a_3a_4a_5a_6 + a_2a_3a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_4 \;=\;& a_0a_1 + a_1a_2 + a_0a_1a_2 + a_3 + a_0a_2a_3 + a_1a_2a_3 + a_4 + a_1a_4 + a_2a_4 + a_1a_3a_4
\end{aligned}
$$

$$
\begin{aligned}
& +a_0a_1a_3a_4 + a_2a_3a_4 + a_0a_2a_3a_4 + a_1a_2a_3a_4 + a_5 + a_1a_5 + a_2a_5 + a_0a_2a_5 \\
& +a_1a_2a_5 + a_0a_1a_2a_5 + a_0a_3a_5 + a_1a_3a_5 + a_0a_2a_3a_5 + a_4a_5 + a_0a_1a_4a_5 \\
& +a_0a_2a_4a_5 + a_1a_2a_4a_5 + a_0a_1a_2a_4a_5 + a_0a_2a_3a_4a_5 + a_1a_2a_3a_4a_5 \\
& +a_6 + a_0a_6 + a_1a_6 + a_0a_1a_6 + a_1a_2a_6 + a_0a_1a_2a_6 + a_1a_3a_6 + a_2a_3a_6 \\
& +a_0a_2a_3a_6 + a_1a_2a_3a_6 + a_4a_6 + a_0a_4a_6 + a_0a_1a_4a_6 + a_1a_2a_4a_6 \\
& +a_0a_1a_2a_4a_6 + a_1a_3a_4a_6 + a_0a_1a_3a_4a_6 + a_0a_2a_3a_4a_6 + a_0a_5a_6 \\
& +a_1a_5a_6 + a_0a_1a_5a_6 + a_2a_5a_6 + a_1a_2a_5a_6 + a_0a_1a_3a_5a_6 + a_0a_1a_4a_5a_6 \\
& +a_2a_4a_5a_6 + a_0a_2a_4a_5a_6 + a_1a_2a_4a_5a_6 + a_0a_1a_2a_4a_5a_6 + a_3a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_5 \quad = \quad & a_0a_1 + a_2 + a_0a_1a_2 + a_3 + a_0a_3 + a_1a_3 + a_0a_2a_3 + a_1a_2a_3 + a_0a_1a_2a_3 \\
& +a_4 + a_0a_4 + a_1a_4 + a_0a_1a_4 + a_0a_2a_4 + a_3a_4 + a_0a_1a_3a_4 + a_0a_1a_2a_3a_4 \\
& +a_5 + a_0a_5 + a_0a_1a_5 + a_0a_2a_5 + a_1a_2a_5 + a_0a_1a_2a_5 + a_3a_5 + a_0a_1a_3a_5 \\
& +a_0a_2a_3a_5 + a_0a_4a_5 + a_1a_4a_5 + a_0a_1a_4a_5 + a_1a_3a_4a_5 + a_0a_1a_3a_4a_5 \\
& +a_2a_3a_4a_5 + a_6 + a_0a_6 + a_0a_1a_6 + a_3a_6 + a_0a_2a_3a_6 + a_4a_6 + a_1a_4a_6 \\
& +a_0a_1a_4a_6 + a_1a_2a_4a_6 + a_3a_4a_6 + a_0a_3a_4a_6 + a_1a_3a_4a_6 + a_0a_1a_3a_4a_6 \\
& +a_5a_6 + a_0a_5a_6 + a_0a_1a_5a_6 + a_2a_5a_6 + a_0a_3a_5a_6 + a_0a_1a_3a_5a_6 \\
& +a_0a_2a_3a_5a_6 + a_1a_2a_3a_5a_6 + a_4a_5a_6 + a_0a_4a_5a_6 + a_1a_4a_5a_6 + a_2a_4a_5a_6 \\
& +a_0a_2a_4a_5a_6 + a_3a_4a_5a_6 + a_0a_3a_4a_5a_6 + a_1a_3a_4a_5a_6 + a_0a_1a_3a_4a_5a_6
\end{aligned}
$$

$$
\begin{aligned}
b_6 \quad = \quad & a_1 + a_2 + a_0a_2 + a_0a_1a_2 + a_3 + a_0a_3 + a_2a_3 + a_4 + a_0a_1a_4 + a_2a_4 + a_0a_3a_4 \\
& +a_0a_2a_3a_4 + a_1a_2a_3a_4 + a_5 + a_2a_5 + a_3a_5 + a_0a_3a_5 + a_0a_1a_3a_5 + a_2a_3a_5 \\
& +a_0a_2a_3a_5 + a_4a_5 + a_1a_4a_5 + a_0a_1a_2a_4a_5 + a_3a_4a_5 + a_0a_3a_4a_5 \\
& +a_1a_3a_4a_5 + a_2a_3a_4a_5 + a_0a_2a_3a_4a_5 + a_6 + a_0a_6 + a_1a_6 + a_0a_1a_6 \\
& +a_1a_2a_6 + a_0a_1a_2a_6 + a_3a_6 + a_0a_3a_6 + a_1a_3a_6 + a_2a_3a_6 + a_0a_2a_3a_6 \\
& +a_1a_2a_3a_6 + a_0a_1a_2a_3a_6 + a_0a_4a_6 + a_1a_4a_6 + a_0a_1a_4a_6 + a_2a_4a_6 \\
& +a_0a_2a_4a_6 + a_1a_2a_4a_6 + a_3a_4a_6 + a_0a_3a_4a_6 + a_0a_2a_3a_4a_6 + a_0a_5a_6 \\
& +a_1a_2a_5a_6 + a_0a_3a_5a_6 + a_1a_3a_5a_6 + a_2a_3a_5a_6 + a_0a_2a_3a_5a_6 + a_0a_4a_5a_6 \\
& +a_0a_2a_4a_5a_6 + a_1a_3a_4a_5a_6 + a_2a_3a_4a_5a_6 + a_0a_2a_3a_4a_5a_6
\end{aligned}
$$

Appendix B

Complexities of Constant Multipliers

This appendix contains complete tables of the complexities for constant multiplication with elements from $GF(2^n)$, $n = 4, 5, 6, 7, 8$. The complexities are measured in XOR gates. The complexities were optimized with the second greedy algorithm described in Chapter 4. Due to the nature of the algorithm, the complexities are suboptimum. The average complexity for each field is given in Table 4.1 in the text.

ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR
ω^1	1	ω^4	4	ω^7	4	ω^{10}	6	ω^{13}	2
ω^2	2	ω^5	5	ω^8	4	ω^{11}	5	ω^{14}	1
ω^3	3	ω^6	4	ω^9	5	ω^{12}	3		

Table B.1: Space complexities for multiplication with elements of $GF(2^4)$ generated by $Q(y) = y^4 + y + 1$, where $Q(\omega) = 0$

ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR
ω^1	1	ω^6	6	ω^{11}	7	ω^{16}	5	ω^{21}	8	ω^{26}	5
ω^2	2	ω^7	7	ω^{12}	8	ω^{17}	5	ω^{22}	8	ω^{27}	4
ω^3	3	ω^8	7	ω^{13}	7	ω^{18}	6	ω^{23}	8	ω^{28}	3
ω^4	4	ω^9	7	ω^{14}	7	ω^{19}	7	ω^{24}	7	ω^{29}	2
ω^5	5	ω^{10}	7	ω^{15}	5	ω^{20}	7	ω^{25}	6	ω^{30}	1

Table B.2: Space complexities for multiplication with elements of $GF(2^5)$ generated by $Q(y) = y^5 + y^2 + 1$, where $Q(\omega) = 0$

ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR
ω^1	1	ω^{12}	8	ω^{23}	9	ω^{33}	7	ω^{43}	9	ω^{53}	9
ω^2	2	ω^{13}	9	ω^{24}	8	ω^{34}	8	ω^{44}	9	ω^{54}	10
ω^3	3	ω^{14}	9	ω^{25}	9	ω^{35}	9	ω^{45}	10	ω^{55}	10
ω^4	4	ω^{15}	10	ω^{26}	10	ω^{36}	10	ω^{46}	11	ω^{56}	10
ω^5	5	ω^{16}	11	ω^{27}	10	ω^{37}	12	ω^{47}	11	ω^{57}	9
ω^6	6	ω^{17}	12	ω^{28}	8	ω^{38}	11	ω^{48}	9	ω^{58}	5
ω^7	7	ω^{18}	12	ω^{29}	8	ω^{39}	10	ω^{49}	8	ω^{59}	4
ω^8	7	ω^{19}	10	ω^{30}	7	ω^{40}	11	ω^{50}	8	ω^{60}	3
ω^9	7	ω^{20}	10	ω^{31}	6	ω^{41}	10	ω^{51}	8	ω^{61}	2
ω^{10}	7	ω^{21}	10	ω^{32}	6	ω^{42}	10	ω^{52}	8	ω^{62}	1
ω^{11}	7	ω^{22}	9								

Table B.3: Space complexities for multiplication with elements of $GF(2^6)$ generated by $Q(y) = y^6 + y + 1$, where $Q(\omega) = 0$

ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR
ω^1	1	ω^{22}	13	ω^{43}	14	ω^{64}	7	ω^{85}	13	ω^{106}	13
ω^2	2	ω^{23}	12	ω^{44}	15	ω^{65}	8	ω^{86}	12	ω^{107}	12
ω^3	3	ω^{24}	12	ω^{45}	16	ω^{66}	9	ω^{87}	12	ω^{108}	12
ω^4	4	ω^{25}	11	ω^{46}	15	ω^{67}	10	ω^{88}	13	ω^{109}	12
ω^5	5	ω^{26}	10	ω^{47}	14	ω^{68}	11	ω^{89}	13	ω^{110}	11
ω^6	6	ω^{27}	10	ω^{48}	13	ω^{69}	13	ω^{90}	12	ω^{111}	10
ω^7	7	ω^{28}	11	ω^{49}	12	ω^{70}	14	ω^{91}	11	ω^{112}	9
ω^8	8	ω^{29}	12	ω^{50}	12	ω^{71}	14	ω^{92}	12	ω^{113}	9
ω^9	8	ω^{30}	13	ω^{51}	11	ω^{72}	13	ω^{93}	13	ω^{114}	10
ω^{10}	8	ω^{31}	14	ω^{52}	11	ω^{73}	12	ω^{94}	14	ω^{115}	11
ω^{11}	8	ω^{32}	14	ω^{53}	10	ω^{74}	12	ω^{95}	15	ω^{116}	12
ω^{12}	8	ω^{33}	14	ω^{54}	10	ω^{75}	13	ω^{96}	15	ω^{117}	12
ω^{13}	8	ω^{34}	14	ω^{55}	11	ω^{76}	14	ω^{97}	14	ω^{118}	12
ω^{14}	9	ω^{35}	15	ω^{56}	12	ω^{77}	13	ω^{98}	14	ω^{119}	12
ω^{15}	10	ω^{36}	15	ω^{57}	11	ω^{78}	13	ω^{99}	14	ω^{120}	11
ω^{16}	10	ω^{37}	14	ω^{58}	11	ω^{79}	15	ω^{100}	13	ω^{121}	6
ω^{17}	11	ω^{38}	13	ω^{59}	10	ω^{80}	16	ω^{101}	13	ω^{122}	5
ω^{18}	12	ω^{39}	14	ω^{60}	10	ω^{81}	15	ω^{102}	12	ω^{123}	4
ω^{19}	13	ω^{40}	13	ω^{61}	9	ω^{82}	14	ω^{103}	11	ω^{124}	3
ω^{20}	14	ω^{41}	13	ω^{62}	8	ω^{83}	13	ω^{104}	11	ω^{125}	2
ω^{21}	14	ω^{42}	13	ω^{63}	7	ω^{84}	13	ω^{105}	12	ω^{126}	1

Table B.4: Space complexities for multiplication with elements of $GF(2^7)$ generated by $Q(y) = y^7 + y + 1$, where $Q(\omega) = 0$

ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR	ω^i	XOR
ω^{1}	3	ω^{44}	15	ω^{87}	16	ω^{129}	17	ω^{171}	17	ω^{213}	19
ω^{2}	5	ω^{45}	13	ω^{88}	18	ω^{130}	16	ω^{172}	16	ω^{214}	17
ω^{3}	8	ω^{46}	12	ω^{89}	15	ω^{131}	14	ω^{173}	16	ω^{215}	16
ω^{4}	10	ω^{47}	11	ω^{90}	16	ω^{132}	14	ω^{174}	16	ω^{216}	16
ω^{5}	12	ω^{48}	10	ω^{91}	15	ω^{133}	14	ω^{175}	16	ω^{217}	13
ω^{6}	14	ω^{49}	10	ω^{92}	15	ω^{134}	13	ω^{176}	16	ω^{218}	11
ω^{7}	15	ω^{50}	10	ω^{93}	15	ω^{135}	14	ω^{177}	16	ω^{219}	10
ω^{8}	16	ω^{51}	11	ω^{94}	15	ω^{136}	13	ω^{178}	16	ω^{220}	10
ω^{9}	15	ω^{52}	12	ω^{95}	14	ω^{137}	12	ω^{179}	15	ω^{221}	11
ω^{10}	15	ω^{53}	14	ω^{96}	13	ω^{138}	12	ω^{180}	14	ω^{222}	13
ω^{11}	15	ω^{54}	15	ω^{97}	12	ω^{139}	14	ω^{181}	15	ω^{223}	11
ω^{12}	15	ω^{55}	16	ω^{98}	12	ω^{140}	14	ω^{182}	16	ω^{224}	12
ω^{13}	15	ω^{56}	17	ω^{99}	12	ω^{141}	14	ω^{183}	16	ω^{225}	13
ω^{14}	14	ω^{57}	17	ω^{100}	12	ω^{142}	14	ω^{184}	17	ω^{226}	15
ω^{15}	15	ω^{58}	17	ω^{101}	13	ω^{143}	13	ω^{185}	17	ω^{227}	15
ω^{16}	16	ω^{59}	17	ω^{102}	14	ω^{144}	14	ω^{186}	17	ω^{228}	17
ω^{17}	18	ω^{60}	16	ω^{103}	15	ω^{145}	14	ω^{187}	16	ω^{229}	15
ω^{18}	17	ω^{61}	17	ω^{104}	17	ω^{146}	14	ω^{188}	14	ω^{230}	13
ω^{19}	16	ω^{62}	17	ω^{105}	17	ω^{147}	14	ω^{189}	14	ω^{231}	13
ω^{20}	15	ω^{63}	16	ω^{106}	18	ω^{148}	15	ω^{190}	13	ω^{232}	13
ω^{21}	14	ω^{64}	16	ω^{107}	16	ω^{149}	16	ω^{191}	13	ω^{233}	14
ω^{22}	12	ω^{65}	16	ω^{108}	16	ω^{150}	16	ω^{192}	14	ω^{234}	12
ω^{23}	10	ω^{66}	16	ω^{109}	16	ω^{151}	17	ω^{193}	14	ω^{235}	13
ω^{24}	9	ω^{67}	16	ω^{110}	17	ω^{152}	16	ω^{194}	13	ω^{236}	14
ω^{25}	10	ω^{68}	15	ω^{111}	17	ω^{153}	15	ω^{195}	13	ω^{237}	16
ω^{26}	11	ω^{69}	15	ω^{112}	16	ω^{154}	15	ω^{196}	13	ω^{238}	15
ω^{27}	12	ω^{70}	15	ω^{113}	17	ω^{155}	15	ω^{197}	14	ω^{239}	17
ω^{28}	12	ω^{71}	15	ω^{114}	17	ω^{156}	15	ω^{198}	14	ω^{240}	17
ω^{29}	12	ω^{72}	16	ω^{115}	17	ω^{157}	15	ω^{199}	16	ω^{241}	17
ω^{30}	12	ω^{73}	15	ω^{116}	16	ω^{158}	14	ω^{200}	15	ω^{242}	18
ω^{31}	11	ω^{74}	16	ω^{117}	14	ω^{159}	15	ω^{201}	14	ω^{243}	17
ω^{32}	11	ω^{75}	18	ω^{118}	14	ω^{160}	16	ω^{202}	12	ω^{244}	16
ω^{33}	11	ω^{76}	18	ω^{119}	14	ω^{161}	17	ω^{203}	11	ω^{245}	17
ω^{34}	11	ω^{77}	19	ω^{120}	14	ω^{162}	17	ω^{204}	11	ω^{246}	17
ω^{35}	11	ω^{78}	19	ω^{121}	14	ω^{163}	17	ω^{205}	10	ω^{247}	17
ω^{36}	11	ω^{79}	17	ω^{122}	14	ω^{164}	16	ω^{206}	10	ω^{248}	16
ω^{37}	12	ω^{80}	18	ω^{123}	14	ω^{165}	17	ω^{207}	10	ω^{249}	14
ω^{38}	14	ω^{81}	19	ω^{124}	15	ω^{166}	18	ω^{208}	12	ω^{250}	12
ω^{39}	16	ω^{82}	17	ω^{125}	17	ω^{167}	19	ω^{209}	13	ω^{251}	10
ω^{40}	16	ω^{83}	16	ω^{126}	17	ω^{168}	18	ω^{210}	15	ω^{252}	8
ω^{41}	18	ω^{84}	15	ω^{127}	17	ω^{169}	17	ω^{211}	15	ω^{253}	5
ω^{42}	17	ω^{85}	16	ω^{128}	17	ω^{170}	17	ω^{212}	17	ω^{254}	3
ω^{43}	15	ω^{86}	15								

Table B.5: Space complexities for multiplication with elements of $GF(2^8)$ generated by $Q(y) = y^8 + y^5 + y^3 + y^2 + 1$, where $Q(\omega) = 0$

Bibliography

[Afa90] V.B. Afanasyev. Complexity of VLSI implementation of finite field arithmetic. In *II. Intern. Workshop on Algebraic and Combinatorial Coding Theory*, pages 6–7, Leningrad, USSR, September 1990.

[Afa91] V.B. Afanasyev. On the complexity of finite field arithmetic. In *5th Joint Soviet-Swedish Intern. Workshop on Information Theory*, pages 9–12, Moscow, USSR, January 1991.

[AK64] J.D. Alanen and D.E. Knuth. Tables of finite fields. *Sankhyā, the Indian Journal of Statistics, Series A*, 26(part 4):1964, Dezember 1964.

[AK89] E. Aarst and J. Korst. *Simulated Annealing and Boltzmann Machines*. Wiley & Sons Inc., 1989.

[Ber68] E.R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, 1968.

[Ber82] E.R. Berlekamp. Bit-serial Reed-Solomon encoders. *IEEE Transactions on Information Theory*, IT-28(6):869–874, November 1982.

[BGM+93] Blake, Gao, Mullin, Vanstone, and Yaghgoobin. *Applications of Finite Fields*. Kluwer Academic Publisher, 1993.

[Bla83] R.E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, Massachusetts, 1983.

[Bla85] R.E. Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, Reading, Massachusetts, 1985.

[BS63] T.C. Bartee and D.I. Schneider. Computation with finite fields. *Information and Control*, 6:79–98, 1963.

[CDD+93] K. Cheung, D. Divsalar, S. Dolinar, I. Onyszchuk, F. Pollara, and L. Swanson. Changing the coding system on a spacecraft in flight. In *Proceedings of the 1993 IEEE International Symposium on Information Theory*, page 381, San Antonio, TX, USA, January 1993.

[Dav72] G.I. Davida. Inverse of elements of a Galois field. *Electronic Letters*, 8(21):518–520, October 1972.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[DZ85] S.M. Dodunekov and V. Zinoviev. On fast decoding of Reed-Solomon codes over $GF(2^m)$ correcting $t \leq 4$ errors. Technical Report LiTH-ISY-I-0750, Dept. of Electrical Engineering, Linköping University, S-58183 Linköping, Sweden, 1985.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[Fat74] R.J. Fateman. Polynomial multiplication, powers and asymptotic analysis: Some comments. *SIAM J. Comput.*, 7(3):196–21, September 1974.

[Fen89] G.L. Feng. A VLSI architecture for fast inversion in $GF(2^m)$. *IEEE Transactions on Computers*, C-38(9):1989, Oct 1989.

[Gei93a] W. Geiselmann. *Algebraische Algorithmenentwicklung am Beispiel der Arithmetik in Endlichen Körpern.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Institut für Algorithmen und Kognitive Systeme, Karlsruhe, Germany, 1993.

[Gei93b] W. Geiselmann. Primitive normal bases with low complexity for the field $GF(2^{24})$. Personal correspondence, November 1993.

[GG90] W. Geiselmann and D. Gollmann. VLSI design for exponentiation in $GF(2^n)$. In J. Seberry and J. Pieprzyk, editors, *Lecture Notes in Computer Science 453: Advances in Cryptology — AUSCRYPT '90*, pages 398–405, Sydney, Australia, January 1990. Springer-Verlag, Berlin.

[GG93] W. Geiselmann and D. Gollmann. Self-dual bases in F_{q^n}. *Designs, Codes and Cryptography*, 3:333–345, 1993.

[Gol67] S.W. Golomb. *Shift Register Sequences.* Holden-Day, San Francisco, 1967.

[Gol84] S.W. Golomb. Algebraic construction for costas arrays. *J. Comb. Theory*, A 37:13–21, 1984.

[GSB91] T.A. Gulliver, M. Serra, and V.K. Bhargava. The generation of primitive polynomials in $GF(q)$ with independent roots and their application for power residue codes, VLSI testing and finite field multipliers using normal bases. *Int. J. electronics*, 71(4):559–576, 1991.

[GT74] D.H. Green and I.S. Taylor. Irreducible polynomials over composite Galois fields and their applications in coding techniques. *Proc. IEE*, 121(9):935–939, September 1974.

[HTDR88] 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 Transactions on Computers*, 37(6):735–739, June 1988.

[HTRG88] I.S. Hsu, T.K. Truong, I.S. Reed, and N. Glover. A VLSI architecture for performing finite field arithmetic with reduced table lookup. *Linear Algebra and its Applications*, 98:249–262, 1988.

[HWB92a] M.A. Hasan, M. Wang, and V.K. Bhargava. Division and bit-serial multiplication over $GF(q^m)$. *IEEE Transactions on Computers*, 41(8):972–980, August 1992.

[HWB92b] M.A. Hasan, M. Wang, and V.K. Bhargava. Modular construction of low complexity parallel multipliers for a class of finite fields $GF(2^m)$. *IEEE Transactions on Computers*, 41(8):962–971, August 1992.

[IT88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[IT89] T. Itoh and S. Tsujii. Structure of parallel multipliers for a class of fields $GF(2^k)$. *Inform. and Comp.*, 83:21–40, 1989.

[JB92] Y. Jeong and W. Burleson. Choosing VLSI algorithms for finite field arithmetic. In *IEEE Symposium on Circuits and Systems, ISCAS 92*, 1992.

[Jun93] D. Jungnickel. *Finite Fields.* B.I.-Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich, 1993.

[Knu81] D.E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys.-Dokl. (Engl. transl.)*, 7(7):595–596, 1963.

[Kol94] G. Kolata. The assault on $114,381,\ldots$. In *New York Times*, page C1, March 22, 1994.

[KRV93] M. Kovac, N. Ranganathan, and M. Varanasi. SIGMA: A VLSI systolic array implementation of a Galois field $GF(2^m)$ based on multiplication and division algorithm. *IEEE Transactions on VLSI Systems*, 1(1):1993, March 1993.

[Kum83] H. Kummer. *Recommendation for space data system standards: Telemetry channel coding: Issue-1.* Consult. Comm. Space Data Syst., September 1983.

[Kun88] S.Y. Kung. *VLSI Array Processing.* Prentice-Hall, 1988.

[LC83] S. Lin and D.J. Costello. *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[LL84] K. Liu and J. Lee. Recent results on the use of concatenated Reed-Solomon / Viterbi channel coding and data compression for space communication. *IEEE Transactions on Communication*, COM-32:518–523, May 1984.

[LN83] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications.* Addison-Wesley, Reading, Massachusetts, 1983.

[Mas89] E.D. Mastrovito. VLSI design for multiplication over finite fields $GF(2^m)$. In *Lecture Notes in Computer Science 357*, pages 297–309. Springer-Verlag, Berlin, March 1989.

[Mas91] E.D. Mastrovito. *VLSI Architectures for Computation in Galois Fields.* PhD thesis, Linköping University, Dept. Electr. Eng., Linköping, Sweden, 1991.

[McC79] J.H. McClellan. *Number Theory in Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, 1979.

[McE87] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, 1987.

[Mes91] R. Mester. *Reed Solomon Encoder/Decoder Chip Set.* BTS GmbH, Darmstadt, Germany, November 1991.

[MK89] M. Morii and M. Kasahara. Efficient construction of gate circuit for computing multiplicative inverses over $GF(2^m)$. *Transactions of the IEICE*, E 72(1):37–42, January 1989.

[ML85] A.M. Michelson and A.H. Levesque. *Error-Control Techniques for Digital Communication.* Wiley & Sons Inc., 1985.

[MO84] J.L. Massey and J.K. Omura. Apparatus for finite field computation. *US Patent Application*, pages 21–40, 1984.

[Mor89] O. Moreno. On the existence of a primitive quadratic of trace 1 over $GF(p^m)$. *J. Comb. Theory*, A 51:104–110, 1989.

[MOVW89] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics, North Holland*, 22:149–161, 1988/89.

[Odl84] A.M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Lecture Notes in Computer Science 209*, pages 224–316. Springer-Verlag, Berlin, 1984.

[Paa93a] C. Paar. Fast finite field arithmetic for VLSI design. In *3rd Benelux-Japan Workshop on Coding and Information Theory*, page 7, Institute for Experimental Mathematics, University of Essen, Germany, August 30 1993.

[Paa93b] C. Paar. A parallel Galois field multiplier with low complexity based on composite fields. In *6th Joint Swedish-Russian Workshop on Information Theory*, pages 320–324, Mölle, Sweden, August 22–27 1993.

[Paa94] C. Paar. Low complexity parallel multipliers for Galois fields $GF((2^n)^4)$ based on special types of primitive polynomials. In *1994 IEEE International Symposium on Information Theory*, Trondheim, Norway, June 27 – July 1 1994.

[PD90] J. Politano and D. Deprey. A 30 Mbits/s (255,223) Reed-Solomon decoder. In *Eurocode '90, International Symposium on Coding Theory and Applications*, pages 385–392, Udine, Italy, November 1990.

[Pee85] J.B.H. Peek. Communication aspects of the compact disc digital audio system. *IEEE Commun. Magaz.*, 23(2):7–15, February 1985.

[PH94] C. Paar and O. Hooijen. Implementation of a reprogrammable Reed-Solomon decoder over $GF(2^{16})$ on a digital signal processor with external arithmetic unit. In *Fourth International ESA Workshop on Digital Signal Processing Techniques Applied to Space Communications*, page 3.11, King's College, London, September 26–28 1994.

[Pin89] A. Pincin. A new algorithm for multiplication in finite fields. *IEEE Transactions on Computers*, 38(7):1045–1049, July 1989.

[PRM90] E.C. Posner, L.L. Rauch, and B.D. Madsen. Voyager mission telecommunication first. *IEEE Commun. Magaz.*, 28(9):22–27, September 1990.

[PW72] W.W. Peterson and E.J. Weldon. *Error-Correcting Codes*. MIT Press, Cambridge, Massachusetts, 1972.

[Rie85] H. Riesel. *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, Boston, Basel, Stuttgart, 1985.

[Sch93] B. Schneier. *Applied Cryptography.* Wiley & Sons, 1993.

[Sed90] R. Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, Massachusetts, 1990.

[SI91] K.A. Schouhamer-Immink. *Coding Techniques for Digital Recorders.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Str86] G. Strang. *Introduction to Applied Mathematics.* Wellesley-Cambridge Press, Wellesley, Massachusetts, 1986.

[TM90] T. Todoroki and S. Miura. Design of a Reed-Solomon decoder using a DSP. In *SBT/IEEE International Telecommunication Symposium ITS, Symposium Record*, pages 443–445, Rio de Janeiro, Brazil, September 1990.

[vT88] H.C.A. van Tilborg. *An Introduction to Cryptology.* Kluwer Academic Publishers, 1988.

[WE92] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design, A Systems Perspective.* Addison-Wesley Publishing Company, second edition, 1992.

[WHPH87] W.W. Wu, D. Haccoun, R. Peile, and Y. Hirata. Coding for satellite communication. *IEEE Jour. Selec. Ar. Commun.*, SAC-5(4):1987, May 1987.

[Wol88] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, 1988.

[WP90] C.C. Wang and D. Pei. A VLSI design for computing exponentiation in $GF(2^m)$ and its application to generate pseudorandom number sequences. *IEEE Transactions on Computers*, C-39(2):258–262, February 1990.

[WTS+85] 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 Transactions on Computers*, C-34:709–717, August 1985.

[YACD89] M.D. Yücel, F. Atmaca, S. Can, and T. Doğan. Implementation of a real time Reed-Solomon encoder and decoder using TMS32010. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Conference Proceedings*, pages 341–344, Victoria, BC, Canada, June 1989.