Bit Serial Systolic Architectures for Multiplicative Inversion and Division over $GF(2^m)$

by

Amir K. Daneshbeh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by examiners.
I understand that my thesis may be made electronically available to the public.

Amir K. Daneshbeh

Abstract

Systolic architectures are capable of achieving high throughput by maximizing pipelining and by eliminating global data interconnects. Recursive algorithms with regular data flows are suitable for systolization. The computation of multiplicative inversion using algorithms based on EEA (Extended Euclidean Algorithm) are particularly suitable for systolization. Implementations based on EEA present a high degree of parallelism and pipelinability at bit level which can be easily optimized to achieve local data flow and to eliminate the global interconnects which represent most important bottleneck in todays sub-micron design process. The net result is to have high clock rate and performance based on efficient systolic architectures.

This thesis examines high performance but also scalable implementations of multiplicative inversion or field division over Galois fields $GF(2^m)$ in the specific case of cryptographic applications where field dimension m may be very large (greater than 400) and either m or defining irreducible polynomial may vary. For this purpose, many inversion schemes with different basis representation are studied and most importantly variants of EEA and binary (Stein's) GCD computation implementations are reviewed. A set of common as well as contrasting characteristics of these variants are discussed. As a result a generalized and optimized variant of EEA is proposed which can compute division, and multiplicative inversion as its subset, with divisor in either *polynomial* or *triangular* basis representation. Further results regarding Hankel matrix formation for double-basis inversion is provided. The validity of using the same architecture to compute field division with polynomial or triangular basis representation is proved.

Next, a scalable unidirectional bit serial systolic array implementation of this proposed variant of EEA is implemented. Its complexity measures are defined and these are compared against the best known architectures. It is shown that assuming the requirements specified above, this proposed architecture may achieve a higher clock rate performance w.r.t. other designs while being more flexible, reliable and with minimum number of inter-cell interconnects.

The main contribution at system level architecture is the substitution of all counter or adder/subtractor elements with a simpler distributed and free of carry propagation delay structures. Further a novel restoring mechanism for result sequences of EEA is proposed

using a double delay element implementation.

Finally, using this systolic architecture a CMD (Combined Multiplier Divider) datapath is designed which is used as the core of a novel systolic elliptic curve processor. This EC processor uses affine coordinates to compute scalar point multiplication which results in having a very small control unit and negligible with respect to the datapath for all practical values of m. The throughput of this EC based on this bit serial systolic architecture is comparable with designs many times larger than itself reported previously.

Acknowledgments

Thanks to Professor Hasan for his guidance, support, patience and constant presence. I specially thank him for keeping me on track to complete this work.

List of Important Acronyms

CMD	Combined Multiplier Divider PE
DPA	Differential Power Analysis
DH	Diffie-Hellman
DL	Discrete Logarithm
DSA	Digital Signature Algorithm
EC	Elliptic Curve
ECC	Elliptic Curve Cryptography
EEA	Extended Euclidean Algorithm
GF	Galois Field
GCD	Greatest Common Divisor
LFSR	Linear Feedback Shift Register
PBIA	Polynomial Basis Inversion Algorithm
PE	Processing Element
RSA	Rivest Shamir Adleman Public Key Cryptosystem
STBIA	Shifted Triangular Basis Inversion Algorithm
TBIA	Triangular Basis Inversion Algorithm

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuous innovation in diverse domains from data processing to coding theory has created an exponential growth in new applications based on means of communication. Around the world, everything and everyone can be seen connected in real time. A well-known example of these applications is secure and fraud free e-commerce over the Internet [22]. As our dependence on these new applications grows, the challenge to achieve higher performance and always available "open" systems must meet the demand for security, privacy and integrity of information. Further, in an "open" environment where new players may enter at any time, the source of information requires authenticity and non-repudiation. Mathematical algorithms to perform finite field operations are a main tool to provide privacy, integrity, authenticity and non-repudiation of data communication.

In this work, hardware implementations of a specific class of operations, the multiplicative inversion and field division over Galois fields $GF(2^m)$, required in many fundamental protocols of data communication systems, *e.g.*, public-key cryptography, is examined and some novel structures are proposed. In particular, the utilization and trade-offs of systolic implementations of inversion and division over Galois fields are investigated.

1.1 Motivation

Any challenge in the design of new communication and data processing devices aims at a higher performance as well as feasibility, reliability and scalability. A practical utilization

of the Internet and data networking relies on privacy and authenticity of information. This is achievable with the implementation of cryptographic functions. The challenge to undertake is to design an efficient hardware implementation of these functions which achieves high performance but also is reliable and scalable. Basic security functions include secret-key, public-key cryptosystems and digital signature functions. These functions can efficiently be implemented in software or hardware. A software implementation which provides flexibility can never be as fast as a special purpose hardware accelerator. Specific algorithms are optimized in order to enhance target processors [1, 6, 44, 113]. However, to reach a higher throughput, the above functions are implemented directly in hardware. In this case, basic arithmetic operations required for these functions should be optimized under the constraint of area, time and energy [75, 91, 118].

Arithmetic operations in Galois field $GF(2^m)$ where m represents the dimension of field have applications in coding theory [11, 81], computer algebra, digital signal processing [77,94] and cryptography [74,82]. However, field dimension requirements are drastically different among these applications. The specific case of cryptography incurs a continuous demand for larger field dimensions as faster data processing devices (computers) crunch higher MIPS (millions of instructions per second). In particular, in public key cryptographic applications, the field dimension may exceed 4000 for discrete logarithm cryptosystems or 500 for elliptic curve cryptosystems [82]. Such trend reflected in using larger length of keying material is shown in Table 1.1 [112].

In this work, hardware implementations of inversion and division over Galois fields are investigated and novel architectures and some practical applications of these are proposed. Not only a higher performance in terms of throughput and clock rate, but also the feasibility and scalability of such implementations are examined.

In the next few chapters inversion and division algorithms over Galois fields will be described. The feasibility, constraints, and requirements of systolic implementations of these algorithms will be discussed. In this context, the following facts will be highlighted:

- Multiplicative inversion operation over Galois fields which is used in certain standard protocols of public-key cryptography requires to be optimized for large dimension fields.

- Most efficient algorithms to compute inversion are recursive algorithms based on vari-

Asymmetric Crypt. key size (in bits)	Discrete Logarithm key size (in bits)	Elliptic Curve DL key size (in bits)	Security Margin (in MIPS years)
RSA, DH	DSA	ECC	
622	112	117	3.51×10^7
777	118	124	5.00×10^8
952	125	132	7.13×10^9
1068	129	136	3.51×10^{10}
1191	133	141	1.73×10^{11}
1369	138	146	1.45×10^{12}

Table 1.1: The security margin of cryptosystems as a function of their keying material

ants of EEA (Extended Euclidean Algorithm) and binary (Stein's) GCD (Greatest Common Divisor) Algorithm.

- As the field dimension increases, a centralized, either register based or parallel, implementation of EEA type algorithms are not efficient, scalable or even feasible under given time-area constraints.
- Systolic architectures are more suitable for recursive algorithms [70] such as variants of EEA.
- Systolic architectures achieve the highest throughput by maximizing the parallelism inherent in algorithms with use of fine-grain (bit or digit level) pipelining, and they eliminate the interconnect bottlenecks of sub-micron designs.

1.2 Previous Work

The main advantages of VLSI technology, *i.e.*, large amount of logic available at a very low cost, reduced physical size and power consumption, and increased reliability, allow the implementation of computationally intensive and recursive algorithms as part of a SoC (System on a Chip). However, as the integration density increases, the interconnection

and wiring (the routing congestion issue in sub-micron designs) dominate the design complexity. Special architectures which can increase the logic utilization without an increase in interconnection are desirable. Luckily, such architectures, the array processors in general or systolic structures in particular, exist.

Systolic architectures are specifically proposed to achieve high throughput and clock rates by maximizing the pipelining and by eliminating the global data interconnects [69, 70, 93]. Recursive algorithms such as extended Euclidean algorithm with regular data flows are particularly suitable for systolization. These present a high degree of parallelism at a bit level which can easily be optimized with few local data flow and interconnect architectures. There are many systolic architectures proposed to compute multiplication [31, 38, 67, 80, 114, 117, 120], division [48, 68] and inversion [32, 40, 108, 110, 115] over Galois fields.

Among the arithmetic operations over Galois fields the multiplicative inversion and division are the most costly ones. This complexity is even more pronounced for larger field dimensions. In general, three major schemes to compute multiplicative inverses exist: Fermat's little theorem, variants of extended Euclidean algorithm (EEA) or a solution of a set of linear equations. First scheme is efficient if either a fast squaring or multiplication method is available [57]. The extended Euclidean algorithm (EEA) based schemes to compute inversion and division are the most efficient in time and area. The third method is generally inefficient for large values of field dimension. However, it is shown that a set of linear equations formed upon triangular basis representation can be solved by schemes similar to EEA [46].

However, a limitation of a direct repetitive application of the extended Euclidean algorithm to compute multiplicative inverses is a conditionally *swap* operation which cannot easily be serialized. This may require variable size counter-like structures with carry propagation chains to keep track of the difference of the degree of polynomials, *e.g.*, [40,96,115], in which case they are not suitable for high-performance and scalable VLSI implementations (including systolic architectures). Many systolic array proposals for multiplicative inversion or division over GF(2^m) based on EEA or its dual (extended Stein's algorithm), *i.e.*, [37,56,116,119], all require counter-like structures with carry propagation delays. Since the carry propagation chain depends on the field dimension m, in general, it dominates

the critical delay path. Only counter or comparator architectures with no carry propagation chain can be considered dimension independent. Moreover, a counter with no carry propagation chain can be easily transformed into a distributed structure.

The above discussion is valid for inversion algorithms in alternative Galois field basis representations, such as a novel multiplicative inversion algorithm, by solving a set of linear equations using double-basis representation over $GF(2^m)$, [46]. In that paper, a centralized control architecture to implement this algorithm is proposed. However, again, as the field dimension increases, a VLSI implementation of such centralized control design with long global control signals becomes inefficient or even impractical for certain clock rate and throughput requirements. In this case, the possibility to incorporate a pipelined change of basis structure and a systolic inverter/divider with a distributed ring counter structure may result in an optimal solution.

1.3 Objectives

This work aims to investigate and propose a class of optimized (free of carry propagation) systolic architectures for inversion and division over Galois fields. The driving force is to improve the throughput and clock rate as a measure of performance without increasing the circuit (area) and design complexity while providing the best solution for scalability (unidirectional single cell type structure) and reliability (ease of fault-tolerant circuit insertion). Particular attention is given to minimize the interconnect at the expense of extra logic. Specifically, the proposed systolic structure will be optimized as follows:

- maximizing the regularity of the array processor,
- minimizing number of cell types,
- reducing out of order signals by elimination of data dependencies with the goal to eliminate the retiming latches between cells.

As a consequence, a feasible architecture with an extremely regular structure is sought. This architecture should be easily scalable, should deliver very high throughput, ideally with a transparent in-line computation over incoming bit streams.

Moreover, from the cryptographic and field application point of view, the design aims to process a field of any dimension, and to be independent of the choice of the field defining irreducible polynomial.

Finally, a practical deployment of such systolic structure (with a unified multiplier divider cell) as the core of an elliptic curve crypto processor will be proposed. The relative cost of the data path versus a complete state machine based control unit for EC scalar point multiplication is investigated and other performance measures are discussed.

1.4 Thesis Outline

The organization of this thesis is as follows:

In Chapter 2, three basis representations of field elements are defined and their relative importance for the implementation of particular operations over Galois fields are compared. Specifically, triangular basis is introduced which will be used for the inversion computation over double-basis. Next, a variety of inversion algorithms each best suitable in a different bases and using different number theory techniques are explained.

In Chapter 3, a detailed survey of many EEA type, binary (Stein's) GCD Algorithm and the best known application of these to perform inversion/division over Galois fields are reviewed. Common and contrasting characteristics of these are classified. Optimization techniques proposed in certain implementations and some important conclusions used in this thesis are highlighted.

In Chapter 4, inversion and division using double-basis representation by applying EEA to a Hankel matrix is reviewed. Some results regarding Hankel matrix entry formation are described. Further, a common set of algorithms to compute inversion and division based on EEA with both polynomial and triangular basis is discussed.

In Chapter 5, systolic architectures as a solution to overcome the ever increasing complexity of computational intensive algorithms are introduced. Next, an optimized (free of carry propagation) systolic array structure for EEA computation is examined which is used as a building block for Galois field inverter-divider. A unidirectional bit serial systolic architecture of inversion and division in both polynomial and triangular basis is introduced. A single type Processing Element (PE) of such architecture and its complexity measures

are investigated and evaluated against similar architectures.

In Chapter 6, a practical application of such an architecture is presented. First, optimizing Elliptic Curve Cryptographic operations in the presence of fast dividers (comparable to multipliers) is discussed. Next, a combined multiplier-divider cell structure is proposed and a unidirectional bit serial systolic architecture as the main data path unit is designed. By incorporating a state machine control unit and shift register files, an extremely high clock rate systolic processor is devised and its performance is evaluated against best known designs in the literature.

In Chapter 7, after summarizing what is accomplished, some remarks regarding future work conclude the thesis.

Chapter 2

Mathematical Background

Optimized algorithms to compute multiplicative inversion, as well as other operations over Galois fields, depend upon the choice of field element representation. In this chapter, some common basis representation of Galois field elements which are of practical use, namely *polynomial*, *dual*, *normal* and *triangular* basis will be reviewed. Next, different algorithms for inversion or division with some implementation examples suitable in each basis representation will be compared. Refer to [78] for a general background on finite prime fields, extension fields and the theory of polynomials over finite fields.

2.1 Basis representation

In the following main classes of basis representation and some variants of each are reviewed. Their relevance while optimizing certain field operations and some basis conversion techniques are discussed.

2.1.1 Polynomial or Canonical Basis

Given an irreducible polynomial $F(x)$ of degree m over the finite field GF(2), considering one of the roots of $F(x)$ such as ω, *i.e.*, $F(\omega) = 0$, it is well known that the set of elements

$$\{1, \omega, \omega^2, \cdots, \omega^{m-1}\},$$

is linearly independent and forms a basis referred to as polynomial or canonical basis. Then, any element $A \in GF(2^m)$ can be represented as

$$A = \sum_{i=0}^{m-1} a_{\Omega i}\omega^i,$$

where $a_{\Omega i} \in \{0, 1\}$ is the i-th coordinate of A with respect to the polynomial basis Ω, or in a column vector form $\underline{a}_\Omega = [a_{\Omega 0}, a_{\Omega 1}, \cdots, a_{\Omega m-1}]^T$ over GF(2).

In this representation, the *addition* of two elements is simply a pairwise bit addition. This can be easily implemented using XOR gates and hence it has a linear complexity. On the other hand, the implementation of multiplication, if not optimized, can be very costly. That is because a modulo reduction step must follow or be incorporated into polynomial multiplication operation. There are more efficient proposals which have low complexity in this basis, [24, 51, 58, 66, 75, 99, 101, 117, 118, 120, 121]. In this basis, many proposals to compute multiplicative inversion and division exist as well which will be reviewed in detail later.

2.1.2 Dual Basis

In order to have an efficient implementation of field multiplication, in some situations a specific basis, namely, dual basis relative to a primal basis can be used. In [10], Berlekamp proposed the use of combined polynomial and its *dual basis* for efficient implementation of multiplication.

Let us consider the polynomial basis $\{1, \omega, \omega^2, \cdots, \omega^{m-1}\}$ of $GF(2^m)$ where ω is a root of the irreducible polynomial $F(x)$ of degree m. Now, let

$$\{\gamma_0, \gamma_1, \cdots, \gamma_{m-1}\},$$

be the dual basis so that

$$Tr(\omega^i\gamma_j) = \lambda_{ij}, \qquad i, j = 0, 1, 2, \cdots, m-1,$$

where $Tr(\cdot)$ is the trace function from $GF(2^m)$ to $GF(2)$ and λ_{ij} is the Kronecker delta function, equal to 1 if $i = j$ and zero otherwise.

The two representations of any element, *e.g.*, x, of the field are related as follows,

$$x = \sum_{0}^{m-1} x_i \omega^i = \sum_{0}^{m-1} x'_i \gamma_i,$$

where the coordinate x'_i of x with respect to the dual basis is given by $x'_i = Tr(\omega^i x)$.

There are well-known methods to find the dual basis of a polynomial basis, *e.g.*, in [74]. Let us consider the polynomial basis $\{1, \omega, \omega^2, \cdots, \omega^{m-1}\}$, where the minimal polynomial of $\omega \in GF(2^m)$ over $GF(2)$ is $F(x)$, (the minimal polynomial is the monic irreducible polynomial over $GF(2)$ of least degree having ω as a root). Then, the dual basis can be computed by

$$\left\{ \frac{g_0}{F'(\omega)}, \frac{g_1}{F'(\omega)}, \cdots, \frac{g_{m-1}}{F'(\omega)} \right\} \tag{2.1}$$

where

$$F(x) = (x + \omega)\Big(\sum_{i=0}^{m-1} g_i x^i\Big), \quad g_i \in GF(2^m), \quad g_{m-1} = 1,$$

and $F'(x)$ is the derivative of $F(x)$ over $GF(2)$ or alternatively $F'(\omega) = \prod_{j=1}^{m-1} (\omega + \omega^{2^j})$ [109].

Among the pair of dual bases, there are those which are the dual of their own. Such basis is called a *self-dual basis* [84, 109]. It has the advantage that there is no need for a conversion from and to their respective duals. However, such self-duality does not exist for all bases.

2.1.3 Normal and Optimal Normal Bases

Considering the Galois field $GF(2^m)$ as an m-dimensional vector space over $GF(2)$, a basis for $GF(2^m)$ of the form

$$\{\beta, \beta^2, \beta^4, \cdots, \beta^{2^{m-1}}\}$$

is called a *normal basis*. The element $\beta \in GF(2^m)$ is the generator of the normal basis. It is well-known, [74], that $GF(2^m)$ has a normal basis for every $m \geq 1$. Even though normal basis can be defined for a *suitable* element of the field, there are no straightforward methods to identify such an element.

Using normal basis, the operation of *squaring* a field element is extremely simple. Considering the normal basis $\{\beta, \beta^2, \beta^4, \cdots, \beta^{2^{m-1}}\}$, any field element a can be represented as

$$a = \sum_{i=0}^{m-1} a_i \beta_i \quad \text{where } a_i \in \{0,1\} \quad \text{and} \quad \beta_i = \beta^{2^i} \quad \text{for } 0 \le i \le m-1.$$

Then

$$a^2 = \sum_{i=0}^{m-1} a_{i-1}\beta_i \quad \text{where subscripts are reduced modulo } m \tag{2.2a}$$

$$= \sum_{i=1}^{m-1} a_{i-1}\beta_i + a_{m-1}\beta_0 \tag{2.2b}$$

That is, the coordinate vector for a^2 is computed by a cyclic shift of the coordinate vector for a.

In general, multiplication in normal basis is costly. However, Massy and Omura have devised an efficient algorithm [89] for multiplication using normal basis. Further, by applying this algorithm to a specific class of normal basis, *i.e.*, *optimal normal basis*, a very efficient hardware implementation for field multiplication is obtained [85]. As in the case of *self-dual basis*, the *optimal normal basis* does not exist for all values of m.

2.1.4 Triangular Basis

The number of distinct bases of an extension field over its ground field is rather large. In practice all the bases can be transformed to each other by matrix multiplication as explained next. However, in general, such an operation has very costly implementation. Among different bases the *triangular* basis of a polynomial has the main advantage that the change of coordinates between the two bases can be easily and efficiently implemented by shift register techniques.

Let $F(x) = \sum_{i=0}^{m} f_i x^i$ be the field defining irreducible polynomial of degree m of $GF(2^m)$. Then, $GF(2^m)$ can be viewed as a vector space of dimension m over $GF(2)$ and any ordered basis

$$\Theta = (\underline{\Theta}_0, \underline{\Theta}_1, \cdots, \underline{\Theta}_{m-1})$$

can be viewed as an $m \times m$ matrix over $GF(2)$. Note that $\underline{\Theta}$'s can be viewed as column vectors over $GF(2)$. Similarly, a new basis $\Lambda = (\underline{\Lambda}_0, \underline{\Lambda}_1, \cdots, \underline{\Lambda}_{m-1})$ can be considered as another $m \times m$ matrix which can be computed by post-multiplying the matrix Θ with a *transformation* matrix, *e.g.*, M. That is

$$\Lambda = \Theta M$$

Such a transformation can be done if and only if the basis transformation matrix M is an $m \times m$ non singular matrix over $GF(2)$ [50].

Let c be an element of the filed $GF(2^m)$, then c can be written in basis Θ as $c = \underline{c}_\Theta \underline{\Theta}^T$, with $\underline{c}_\Theta \in GF(2^m)$ being the vector of coordinates of c with respect to Θ, and similarly, $c = \underline{c}_\Lambda \underline{\Lambda}^T$.

It is easy to show that $\underline{c}_\Theta = \underline{c}_\Lambda M^T$ and vice versa, $\underline{c}_\Lambda = \underline{c}_\Theta (M^T)^{-1}$. A suitable choice of M can facilitate the transformation of coordinates from one basis system to the other.

Let $M = [M_{ij}]$ with

$$M_{i,j} = \begin{cases} f_{i+j+1} & 0 \leq i+j \leq m-1, \\ 0 & m \leq i+j \leq 2m-2, \end{cases} \tag{2.3}$$

which is a Hankel matrix with constants on its back-diagonals. Then M^{-1} is a matrix with entries

$$(M^{-1})_{i,j} = \begin{cases} 0 & 0 \leq i+j \leq m-2, \\ h_{i+j+1-m} & m-1 \leq i+j \leq 2m-2, \end{cases} \tag{2.4}$$

with $h_0 = 1$ and

$$h_k = \sum_{i=0}^{k-1} f_{m-k+i} h_i.$$

Given this particular choice of M, Λ is referred to as the *triangular basis* with respect to Θ. With this particular choice of M, the change of coordinates of any element c between Θ and Λ can be easily implemented by shift register techniques using:

$$(\underline{c}_\Theta)_j = \sum_{i=0}^{m-j-1} f_{i+j+1} (\underline{c}_\Lambda)_i \quad 0 \leq j \leq m-1 \tag{2.5}$$

and

$$(\underline{c}_\Lambda)_k = \begin{cases} (\underline{c}_\Theta)_{m-1-k} & k = 0, \\ (\underline{c}_\Theta)_{m-1-k} + \sum_{i=0}^{k-1} f_{m-k+i}(\underline{c}_\Lambda)_i & 1 \le k \le m-1. \end{cases} \tag{2.6}$$

The implementation of Equation (2.5) is possible with an $(m-1)$ stage linear *feed-forward* shift register while that of Equation (2.6) requires an $(m-1)$ stage linear *feed-back* shift register.

Considering the specific case of $\Theta = \Omega$, as a *polynomial* basis, *i.e.*,

$$\Theta = \{1, \omega, \omega^2, \cdots, \omega^{m-1}\},$$

where ω satisfies an irreducible polynomial $F(\omega) = 0$, then, field element representation with respect to the corresponding triangular basis has the feature:

$$\underline{\omega}^j_\Lambda = (t_{j+1}, t_{j+2}, \cdots, t_{j+m})$$

with $t_0 = 1$, $t_1 = t_2 = \cdots = t_{m-1} = 0$, and

$$t_k = \sum_{i=0}^{m-1} f_i t_{k-m-i} \quad \text{for } k \ge m,$$

which can be implemented using a conventional m stage linear feedback shift register (LFSR) with $F(x)$ as its feedback function [50].

2.2 Multiplicative Inversion

Multiplicative inverse of element A, denoted by A^{-1}, is defined as $A \cdot A^{-1} = 1$. When polynomial basis is used, we have

$$A(x) \cdot A^{-1}(x) \equiv 1 \mod F(x),$$

where $F(x)$ is its defining irreducible polynomial. It is known that all nonzero elements of the field have distinct inverses. Commonly, multiplicative inversion is computed using one of the following techniques:

- Fermat's little theorem scheme [28, 30, 57, 106],

- extended Euclidean algorithm (EEA) [5,15,40,41], or extended binary (Stein's) GCD computation [54, 103, 110, 116]
- the solution of a set of linear equations [21, 46, 108].

Before describing algorithmic techniques a possible non algorithmic method to compute inverses and its main drawback is mentioned.

2.2.1 Non-Algorithmic Look-Up Table Inversion

The easiest and the fastest method to find the inverse of a field element would be a look-up table technique. Since the inverses of field elements are unique, a look-up table can be precomputed. In [68, 105], some variants of this method have been proposed and implemented. For example in [105], Vries et al. proposed a ROM look-up table scheme for a field dimension of ($m = 8$) which had better performance in terms of area and time when compared to algorithmic schemes of the same dimension. However, such a non-algorithmic scheme is only suitable for "small" size fields, where "small" is understood by the amount of available memory. In addition, there are proposals in which in order to compute the inverses of specific large field elements, *i.e.*, Optimal Extended Fields, first the element is reduced to a subfield element where the inverses can be easily computed by other means such as binary extended Euclidean algorithm [6], direct parallel inversion [91] or a look-up table method.

2.2.2 Fermat's Little Theorem Based Inversion

The easiest but not necessarily the most efficient method to compute the multiplicative inversion is using Fermat's little theorem which reduces inversion computation to a series of multiplication (and/or squaring). Fermat's little theorem [36] asserts,

Theorem 2.1 *If β is an element of $GF(2^m)$, then*

$$\beta^{2^m-1} = 1. \tag{2.7}$$

Then, the inverse of β is given by

$$\beta^{-1} = \beta^{2^m-2}.$$

Thus the following efficient method to compute the inverse element β^{-1} using Fermat's theorem can be devised. It is known that

$$2^m - 2 = \sum_{i=1}^{m-1} 2^i,$$

the element β^{2^m-2} may be written as

$$\beta^{-1} = \beta^{2^m-2} = \beta^{2^1} \times \beta^{2^2} \times \cdots \times \beta^{2^{m-1}}. \tag{2.8}$$

If there is a basis where either squaring or multiplication is "cheap" and/or fast, the above scheme will be useful. In *normal basis* representation, it is said that the *squaring* can be computed by simple shift operation. Hence, efficient algorithms can be devised. For example Wang et al., in [106], proposed a simple but costly implementation of Equation (2.8). In such a scheme the number of iterations are equal to the dimension of the field minus one. That is a total of $(m-1)$ squaring and $(m-2)$ multiplications are needed.

In [111], Wei uses the following recursive expression

$$2^m - 2 = 2(1 + 2 + (1 + 2(1 + \cdots))),$$

where number of iterations is $(m-1)$, to devise a polynomial basis inversion algorithm based on Fermat's theorem such that

$$\beta^{-1} = (\beta \cdots \beta(\beta(\beta(\beta^2)^2)^2)^2. \tag{2.9}$$

Also in this technique, a total of $(m-1)$ squaring and $(m-2)$ multiplications are needed.

In order to speed up the algorithm different alternatives are possible. In [28], Feng and in [57], Itoh and Tsujii have proposed variants of Fermat's theorem scheme where the number of multiplications can be reduced. In general, for a normal basis where the squaring is almost free, the multiplication operation is a costly one and all the optimization of the design must be concentrated on it. Optimal normal basis provides an alternative solution for multiplication optimization. Reducing the number of multiplications by choosing certain basis to have a low Hamming weight binary representation is an approach. For example, Feng uses the binary expression of $m-1$

$$m - 1 = m_q 2^q + m_{q-1} 2^{q-1} + \cdots + m_1 q^1 + m_0 2^0, \qquad m_i \in \{0, 1\},\ 0 \leq i \leq q \tag{2.10}$$

to devise an efficient inversion algorithm [28]. Hence, in order to compute β^{-1}, $(q+p)$ multiplications and $(m-1)$ squarings are required, where p is the number of $one's$ in the binary expression of $m-2$. On average, the computational complexity can be reduced to $O(m \times \log m)$. According to algorithm in [28], both simple squaring and square-rooting are required which can efficiently be implemented by right and left shift operations respectively.

In [57], Itoh and Tsujii have proposed an alternative approach to reduce the number of multiplication using normal basis which requires only forward squaring and its implementation is easier. The Itoh and Tsujii's scheme uses the fact that since the inverse of an element β in a field according to Fermat's little theorem is:

$$\beta^{-1} = \beta^{2^m-2} = \left(\beta^{2^{m-1}-1}\right)^2,$$

then, if m is odd, *i.e.*, $(m-1)$ is even, knowing

$$2^{m-1} - 1 = \left(2^{(m-1)/2} - 1\right)\left(2^{(m-1)/2} + 1\right),$$

then

$$\beta^{2^{m-1}-1} = \left(\beta^{2^{(m-1)/2}-1}\right)^{2^{(m-1)/2}+1},$$

and this operation requires only one multiplication to compute $\beta^{2^{m-1}-1}$ knowing $\beta^{2^{(m-1)/2}-1}$; recall that squaring is free and it is not counted. Also, if m is even, then

$$\beta^{2^{m-1}-1} = \beta^{2(2^{(m-1)/2}-1)(2^{(m-1)/2}+1)+1},$$

and it requires two multiplications to evaluate $\beta^{2^{m-1}-1}$, once $\beta^{2^{(m-1)/2}-1}$ has been computed.

Thus a recursive algorithm with minimum number of multiplications can be devised. In general, when m is the field dimension, exactly $\lfloor \log(m-1) \rfloor + \omega(m-1) - 1$ field multiplications are required, where $\omega(m-1)$ denotes the Hamming weight, the number of 1's in the binary representation of $(m-1)$. To summarize, in general, efficient inversion algorithms in normal basis require:

- Number of Multiplications $= \lfloor \log(m-1) \rfloor + \omega(m-1) \leq 2\lfloor \log(m-1) \rfloor$
- Number of Shifts $= m-1$

Both algorithms in [28] and [57] have similar computational complexities. Itoh's algorithm in [57], although simpler to implement, requires a stack structure to hold the intermediate temporary results. That is the algorithm is not completely serializable with use of a single temporary variable. In [3], Agnew et al. have discussed the hardware implementation of this scheme and its space requirements. They showed how to modify the scheme in order to trade-off an increase in number of multiplications for the space requirements. The following scheme is proposed. Suppose that $m-1 = gh$. Then

$$2^{m-1} - 1 = 2^{gh} - 1 = (2^g - 1)\Big(\sum_{i=0}^{h-1} 2^{gi}\Big). \tag{2.11}$$

Now,

$$\beta^{-1} = \beta^{2^{m-1}-2} = \beta^{2(2^{m-1}-1)} = \gamma^{(2^g-1)(\sum_{i=0}^{h-1} 2^{gi})}, \tag{2.12}$$

where $\gamma = \beta^2$. Then $\lambda = \gamma^{(2^g-1)}$ can be calculated in $g-1$ multiplications and $\lambda^{\sum_{i=0}^{h-1} 2^{gi}}$ can be computed in $h-1$ multiplications. Hence, β^{-1} can be computed in $(g+h-2)$ multiplications. It is said that the number of multiplications using this approach is minimized when g and h are about $\sqrt{m-1}$. In the case that such suitable values are not available, alternative factorizations methods can be used [3].

2.2.3 Extended Euclidean Algorithm Based Inversion

If a fast multiplier (squarer) does not exist, then the most efficient way to compute inverses is using schemes based on Extended Euclidean Algorithm (EEA) or binary (Stein's) GCD computation. The EEA is based on the classical Euclidean algorithm which computes the greatest common divisor of two integers greater than unity. The Euclidean algorithm in its polynomial form may be generalized to computes the greatest common divisor of two polynomials greater than zero polynomial. In this section a first description of EEA is provided. But in the next chapter, a detail comparison of implementations based on variants of EEA and binary GCD computation will be reviewed.

In an extended Euclidean algorithm (EEA) not only the GCD of two polynomials a and b but also two auxiliary polynomials a' and b' are computed such that

$$aa' + bb' = GCD(a, b).$$

The advantage is that the computation of three unknowns: $GCD(a, b)$, a', b', can be done simultaneously by keeping track of only two vectors of three elements each. In the following, the application of EEA over polynomial to compute the inverse of an element is presented.

Let $A(x)$ be a polynomial corresponding to element $A \in GF(2^m)$, where $GF(2^m)$ is defined by the irreducible polynomial $F(x)$. Since $F(x)$ is an irreducible polynomial, $GCD(F(x), A(x)) = 1$. In order to compute the *inverse* of $A(x)$, *i.e.*, $A'(x)$, the following relationship can be used

$$A(x)A'(x) + F(x)U(x) = 1. \tag{2.13}$$

Constructing two initial equations

$$A(x)A'^{(-1)}(x) + F(x)U^{(-1)}(x) = R^{(-1)}(x), \tag{2.14}$$

$$A(x)A'^{(0)}(x) + F(x)U^{0)}(x) = R^{(0)}(x), \tag{2.15}$$

where $A'^{(-1)}(x) = 0$, $A'^{(0)}(x) = 1$, $U^{(-1)}(x) = 1$, $U^{(0)}(x) = 0$, and $R^{(-1)}(x) = F(x)$, $R^{(0)}(x) = A(x)$. Now, applying a GCD computation sequence to pair $[R^{(-1)}(x), R^{(0)}(x)]$, at each iteration, a triple of auxiliary polynomials can be computed as

$$A'^{(i)}(x) = A'^{(i-2)}(x) + Q^{(i)}(x)A'^{(i-1)},$$

$$U^{(i)}(x) = U^{(i-2)}(x) + Q^{(i)}(x)U^{(i-1)},$$

$$R^{(i)}(x) = R^{(i-2)}(x) + R^{(i)}(x)U^{(i-1)},$$

where $Q^{(i)}(x) = \lfloor R^{(i-2)}(x)/R^{(i-1)}(x) \rfloor$. Note that addition and subtraction over $GF(2)$ are the same. With these transformations at the ith step when $Q^{(i)}(x) = GCD(A(x), F(x)) = 1$ is reached, then Equation (2.13) can be conceived as

$$A(x) \cdot A'^{(i)}(x) + F(x) \cdot U^{(i)}(x) \equiv 1 \pmod{F(x)}.$$

$F(x)$ is an irreducible polynomial, $F(x) \cdot U^{(i)}(x) \equiv 0 \pmod{F(x)}$ always, and $A'^{(i)}(x)$ is the *inverse* of element $A(x)$.

As far as numbers of iterations, in [87], two theorems by Norton show that for uniformly distributed normalized polynomials of degree m and n over $GF(2)$, the average number of iterations of a *binary left- or right-shift* GCD algorithm is

$$\frac{1}{2}m + \frac{1}{3}n + O(1) \quad \text{if } m \geq n \geq 0,$$

and the maximum number of iterations, the worst case, is

$$m + \left\lfloor \frac{n}{2} \right\rfloor + 1.$$

These results improve similar estimations by Knuth [63]. The average number of EEA iterations is less than Fermat's little theorem and also the operations needed in each iteration can be efficiently implemented, as it will be shown.

2.3 Summary

In this chapter, a brief review of basis representation of elements in extension fields has been presented. It has been mentioned that the choice of an efficient technique for inversion and other operations are intimately tied to the choice of basis representation of the field. Next, some methods for computing inverses, most importantly those based on Fermat's little theorem have been reviewed.

In the next chapter, methods to compute inverses based on variants of extended Euclidean algorithm or binary (Stein's) GCD algorithm will be compared and classified. Further, it will be shown that a method based on solving a set of linear equations using double-basis (polynomial and triangular) representation can make use of EEA type algorithms for efficient inversion and division computation.

Chapter 3

(Extended) Euclidean and Binary GCD Algorithm

3.1 Greatest Common Divisor Computation

The *Greatest Common Divisor* (GCD) computation is fundamental to algebraic and symbolic operations. Algorithms and implementations applied to integers can be easily modified for polynomials hence applied to Galois fields. In particular, the extended variants of the GCD computation has many applications in coding theory and cryptography. Because of inherent parallelism and iterative nature of many algorithms based on GCD, especially those applied to Galois fields, an iterative hardware mapping of these is common.

The *greatest common divisor* (GCD) can be defined as follows. If r and s are integers (or polynomials), not both zero, their greatest common divisor, $\gcd(r, s)$, is the larger integer (or polynomial) that evenly divides both r, s [64].

3.1.1 Basic GCD and EEA

In [64], Knuth provides the classic GCD and EEA algorithms for integers, which can readily be extended for polynomials.

It can be noted that primarily the GCD computation is a repeated remainder calculation. However, the repetition itself makes a GCD computation more complex than its basic

Algorithm 1 Classic GCD computation Algorithm.

input: r and s integers or polynomials.

output: $\gcd(r, s)$.

  while $(s \neq 0)$ **do**

    $[r, s] \leftarrow [s,\ r \pmod{s}]$;

return r

remainder calculation. Algorithm 1 shows the classic GCD computation where the value of s decreases monotonically in all steps, and $s \leq r$ always, except in the first iteration if $s > r$ at input. This condition $s \leq r$ ensures the termination of the algorithm. Let us assume that $s \leq r$ at input, then the worst case number of iterations occurs when the integer inputs are consecutive Fibonacci numbers. In this case the binary representation of an integer s decreases at most one bit at each iteration. For an $n > 0$, and $r = F_{n+2}$, $s = F_{n+1}$, Algorithm 1 requires exactly n division steps [64]. For polynomials of degree n, m, the worst case number of iterations is $(n + m + 1)$. The validity of algorithm follows from:

$$gcd(r, s) = gcd(s, r - qs), \tag{3.1}$$

$$gcd(r, 0) = |r|. \tag{3.2}$$

Equation (3.1) represents a *GCD-preserving* transformation, also described by Brent et al. in [14]. Formally, in a *GCD-preserving* transformation, a pair of polynomials $[r, s]$ are transformed into $[\overline{r}, \overline{s}]$ such that $gcd(\overline{r}, \overline{s}) = gcd(r, s)$. Other such transformations will be mentioned in all GCD and EEA variants. The extended Euclidean Algorithm (EEA) based on Algorithm 1 is shown in Algorithm 2.

In Algorithm 2, the implementation of $[w_3, u_3, t_3]$ are not required if $[w_1, u_1, t_1]$ and $[w_2, u_2, t_2]$ can be updated simultaneously, as is the case in hardware implementations. More importantly, if w_3, u_3, and t_3 can be updated simultaneously, *i.e.*, w_3 and u_3 can be transformed according to the t_3 transformation, then these remainders can be computed with no need to compute q explicitly.

In EEA, the sequence of t's represents the *gcd transformation sequence*, and the sequence of u's, the *result transformation sequence* (for inversion and division). At each

Algorithm 2 Extended Euclidean Algorithm (EEA).

input: r and s integers or polynomials.

output: w, u, t such that $w\ r + u\ s = t = gcd(r, s)$

$[w_1, u_1, t_1] \leftarrow [1, 0, r]$; $[w_2, u_2, t_2] \leftarrow [0, 1, s]$;

while $(t_2 \neq 0)$ **do**

$q \leftarrow \lfloor t_1/t_2 \rfloor$ such that $t_1 - qt_2 = t_3$;

$[w_3, u_3, t_3] \leftarrow [w_1, u_1, t_1] - [w_2, u_2, t_2]q$;

$[w_1, u_1, t_1] \leftarrow [w_2, u_2, t_2]$; $[w_2, u_2, t_2] \leftarrow [w_3, u_3, t_3]$;

return w_1, u_1, t_1

iteration i, the invariant of Algorithm 2 is

$$w_i\ r + u_i\ s = t_i. \tag{3.3}$$

The division step of Algorithm 2 required to compute a remainder, explicitly shown as modulo reduction in Algorithm 1, is computationally expensive and alternatives to it must be found. All efficient implementations, in particular for large size of operands in hardware designs, implement this remainder calculation in a *stepwise* fashion, *e.g.*, digit or bit serial. In general, this is achieved by a single step shift per iteration up to an *alignment* condition. This operation represents the first common task of all algorithms which will be reviewed in this thesis. We will refer to this as Task 1. There are two classes of architecture to perform such a stepwise shift and alignment operation. These are referred to as *left-shift* and *right-shift* algorithms. Let us start with right-shift class of algorithms.

3.2 Right-Shift GCD Algorithm, Right-Shift EEA

An alternative to the classic GCD computation without a division step is the binary GCD (or Stein's) algorithm, also called right-shift or low-order first, which uses only additions, shifts and comparisons. These are so called since they perform *GCD-preserving* transformations based on checking the value of the lowest order coefficients of polynomials or least significant bit (lsb) of integers. Polynomials (or integers) shrink from their higher order degrees (or most significant bits) but not necessarily monotonically, up to reaching

a zero remainder as expected by the classic GCD algorithm. In the following six variants of binary GCD algorithm or their extended versions are discussed.

3.2.1 Binary GCD Algorithm and its Plus-Minus Variant

The basic variant of binary GCD (Stein's) algorithm is shown in Algorithm 3.

Algorithm 3 Stein's Binary GCD Algorithm.

input: r and s non-negative integers or polynomials, where $s_0 = 1$, and s_0 represents the lsb of binary representation of integer s, or the coefficient of degree zero of polynomial s.

output: $\gcd(r, s)$.

```
while (r ≠ 0) do
  while (r_0 = 0) do
    r ← r/2;
  if (r < s) then
    [r, s] ← [s, r];      // swap //
  r ← r − s;
return s
```

In Algorithm 3, no division step is required and this implies that a newly computed r may be less or greater than s, in contrast with Algorithm 1 where it is ensured that $s \leq r$ always. Hence, the division step is eliminated at the expense of a comparison and a probable **swap** step. In Algorithm 3, the comparison step and a possible **swap** is required to ensure $s \leq r$, but the implementation complexity of such a task plus a stepwise shift and alignment operation to implement Task 1 is by far less than a division step. On the other hand, on average the number of iterations of Algorithm 3 are much greater than Algorithm 1 using a division step applied on the same set of input values, however in a hardware implementation, each iteration (cycle) time of Algorithm 3 is much shorter than Algorithm 1. In practice, all GCD and extended GCD computation algorithms, except one [56], which implement a variant of Task 1, have a **swap** condition checking mechanism as well. We will refer to this as Task 2.

A major difficulty in implementing Algorithm 3 is that the comparison $(r < s)$ requires a knowledge of all digits of r and s which is not possible in certain (bit serial) implementations

and in any case it has $O(\log r)$ computational complexity (Brent [14]). In [14], an efficient hardware alternative named plus-minus (PM) is proposed as shown in Algorithm 4.

Algorithm 4 Plus-Minus Variant of Binary GCD Algorithm for integers.

input: r, s, non-negative integers, where r is odd and $s < r$.
output: $\gcd(r, s)$.

```
d ← 0;  // This counter estimates the difference between ⌈log |r|⌉ and ⌈log |s|⌉ //
while (s ≠ 0) do
  while (s_0 = 0) do
    s ← s/2;    d ← d + 1;
  if (d ≥ 0) then
    [r, s] ← [s, r]   d ← −d;        // swap //
  if ((r + s) mod 4 = 0) then
    s ← (r + s)/2;
  else
    s ← (r − s)/2;
return |r|
```

Algorithm 4 neither has a division step nor a comparison between two integers. A hardware implementation of Algorithm 4 may run faster than Algorithm 1 even though the $s \leftarrow (r+s)/2$ operation may not allow an optimal decrease of integers at each iteration (as occurs in the classic GCD). Also, the hardware implementation of Algorithm 4 is more efficient than Algorithms 3. This is mainly due to possibility of using a counter to keep track of the number of shifts of integers. The counter value represents an estimate (an upper-bound) on the difference of the size of integers (or degree of polynomials). The computational complexity of the incrementer-decrementer of this counter is only $O(\log \log r)$ as compared to $O(\log r)$ required for the comparison step of Algorithm 3.

Algorithm 4 can be modified to compute polynomial GCD as well. In the specific case of binary polynomials, in fact, it can be simplified with no need to distinguish between plus or minus operation. Hence, a simplified variant as in Algorithm 5 may be considered.

In [87], Norton proves that in the worst case, the number of iterations of polynomial right-shift GCD algorithm over extended binary fields with polynomials of degree $m \geq n \geq$

Algorithm 5 Plus-Minus Variant of Binary GCD Algorithm for binary polynomials.

input: r, s, binary polynomials, where $r_0 = 1$ and $s < r$.

output: $\gcd(r, s)$.

```
d ← 0;  // This counter estimates the difference between deg(r) and deg(s) //
while (s ≠ 0) do
  while (s_0 = 0) do
    s ← s/x;    d ← d + 1;
  if (d ≥ 0) then
    swap (r, s);    d ← −d;
  s ← (r − s)/x;
return r
```

0 is $m + \lfloor n/2 \rfloor + 1$. It can be noted that this is less than what was said in Section 3.1.1. Achieving such a lower bound depends on the implementation feasibility and in the case of prime fields or polynomials over non-binary fields the previously stated lower bound, *i.e.*, $m + n + 1$, prevails.

The implementation of Algorithms 4 and 5 can be further optimized in hardware designs, specifically advantageous in bit serial architectures, where d can be represented by a "sign and magnitude unary" scheme. In this scheme the sign of d and the absolute value $|d|$ are kept separate and a new variable $\delta = 2^{|d|}$ represents the magnitude [17, 116]. Not only this scheme is suitable for bit-level pipelining but even in a centralized control scheme it eliminates the delay of incrementer-decrementer of a counter by using right or left shift operation. The representation of δ requires a maximum of $m + 1$ storage spaces, *i.e.*, registers, which is available and convenient to have in a bit serial implementation. In other cases, this may not be a desirable feature. This representation of d which may represent the difference of degree of polynomials as well, can be used not only in the right-shift but also in the left-shift GCD algorithms as it will be discussed later.

Based on Algorithm 4, different schemes to compute multiplicative inverses are proposed. In [103], Takagi proposed a direct extension of Algorithm 4, similar to the EEA, to compute multiplicative inverses over integer prime fields. Next, Takagi et al. in [110], and Wu et al. in [115, 116] proposed specific extended binary GCD algorithms to compute multiplicative inversion and division over binary finite fields GF(2^m).

3.2.2 Binary GCD Division Algorithm

An implementation of division over finite fields, using a direct application of Stein's algorithm is presented in [115] which is shown in Algorithm 6.

Algorithm 6 Modular Division over Prime Fields using Extended Binary GCD Algorithm.

input: p prime field modulus, b integer dividend, $a > 0$ integer divisor.
output: $b/a \mod p$.

```
[r, s] ← [p, a]   [u, v] ← [0, b]
while (s > 0) do
  if (s_0 = 1) then
    if (s < r) then
      [r, s] ← [s, r − s]
      [u, v] ← [v, (u − v) mod p]
    else
      [r, s] ← [r, s − r]
      [u, v] ← [u, (v − u) mod p]
  [r, s] ← [r, s/2]
  [u, v] ← [u, (v/2) mod p]
return v
```

In Algorithm 6 the pair $[r, s]$ represents the gcd sequence and the $[u, v]$ represents the result sequence. The gcd sequence follows the *GCD-preserving* transformation of Stein's algorithm, *e.g.*, $[r, s] \leftarrow [r, s/2]$. In Algorithm 6, also the result sequence follows the same *GCD-preserving* transformation, and a division $b/a \mod p$ can be computed since each $(u - v)$ or $(v/2)$ operation is followed by a modulo reduction if necessary. Note that the modulo reduction step of this algorithm is required either a division or solely inversion is performed.

An important distinction between the modulo reduction required in the *result transformation sequence* and the one mentioned in the *gcd transformation sequence* is in order. The modulo reduction mentioned in the gcd transformation sequence represents an integer (or a polynomial) division to compute a remainder, corresponding to a shrinking divisor at each iteration. Hence, efficient solutions to implement it with right-shift or left-shift variants are proposed. However, a probable modulo reduction in the result transformation

sequence is an effective modulo reduction by an irreducible polynomial (or a modulus) of the underlying field. This modulo reduction, when required, cannot be transformed similar to the gcd transformation sequence case; even though, a stepwise implementation of the gcd transformation sequence may facilitate the implementation of this modulo reduction.

Another variant of Algorithm 6 proposed by Takagi et al. in [110], which will not be repeated, follows the Brent's idea and compares the degree of polynomials instead of polynomial themselves. Always in [110], a divider implementation is proposed which requires two counters, first to keep track of the *exit* condition of the **while** loop of Algorithm 6, and second to hold the difference of the degree of polynomials.

In [115], a hardware optimization of Algorithm 6 to implement a systolic divider over GF(2^m) with fixed number of iterations is proposed which is shown in Algorithm 7. Clearly, this implementation does not require the first counter described in [110]. In a systolic architecture the cost of a stepwise modulo reduction is contained. In Algorithm 7, the swap condition based on the degree tracking by a counter as described by Brent et al. and Takagi et al. is used as well.

Algorithm 7 Polynomial Division over Binary Fields by Extended Binary GCD Algorithm.

input: $f(x)$ irreducible polynomial of a field of dimension m, dividend $b(x)$ and divisor $a(x)$.
output: $b(x)/a(x) \mod f(x)$.

```
d ← −1;  [r, s] ← [f(x), a(x)];  [u, v] ← [0, b(x)];
for (k = 1 to 2m − 1) do
  if (s_0 = 1) then
    if (d < 0) then
      [r, s] ← [s, r + s];
      [u, v] ← [v, u + v];   d ← −d
    else
      [r, s] ← [r, r + s];
      [u, v] ← [u, u + v];
  [r, s] ← [r, s/x];
  [u, v] ← [u, (v/x) mod f(x)];  d ← d − 1;
return v
```

In Algorithm 7, the overwriting of r and u occurs only when $d < 0$, which means the number of shifts of s is greater than r. In other words, it corresponds to the case $s < r$ of Algorithm 6. Also, in a binary field case, all subtractions are substituted with XOR operations, hence no ordering of polynomials where $d < 0$ or otherwise is shown.

Algorithm 7 is the first algorithm in which a new formulation of the *exit* condition is provided. We will refer to this *exit* condition checking as Task 3. All algorithms presented here have a variant of this. In Algorithm 7 , the **while** loop with *exit* condition checking $s > 0$ is changed into a **for** loop of $2m - 1$ since the maximum number of iterations require for polynomial s to reach zero is $2m$. This is a typical case for hardware implementations where irregular behavior and hence condition checking such as variable number of iterations are not desirable. For example, systolic architectures where an exact number of iterations can be mapped into an exact number of processing elements.

3.2.3 Almost Inverse Algorithm

In [96], another variant of the extended Binary GCD algorithm for computing multiplicative inverses over finite fields named *Almost Inverse Algorithm* is proposed which is shown as Algorithm 8.

Algorithm 8 combines ideas from Stein's binary GCD algorithm for the gcd sequence transformation and Berlekamp's result sequence transformation [12] described later. Also in this algorithm, the pair $[r, s]$ represents the gcd transformation sequence and the $[u, v]$ represents the result transformation sequence. The gcd transformation sequence follows the *GCD-preserving* transformation of Stein's algorithm. This algorithm avoids a possible costly modulo reduction of the result sequence. However, it cannot compute division since no modulo reduction is used. When Algorithm 8 ends, It returns an output ux^k such that $u \equiv a^{-1}(x) \mod f(x)$, where $f(x)$ can be either the binary representation of prime field modulus or the irreducible polynomial or a binary field, and x is the indeterminant of polynomial representation of field elements. When it terminates the inputs and the outputs are related by

$$wx^k f(x) + ux^k a(x) = x^k.$$

Hence, in order to get the correct result, it is necessary to divide ux^k by x^k, or multiply by x^{-k}.

Algorithm 8 Almost Inverse Algorithm.

input: $f(x)$ prime field modulus or irreducible polynomial, $a(x)$ a field element.
output: ux^k, k where $u \equiv a^{-1}(x) \mod f(x)$.

```
k ← 0;  [r, s] ← [f(x), a(x)];  [u, v] ← [1, 0];
while (deg(s) > 0) do
  if (s_0 ≠ 0) then
    if (deg(s) < deg(r)) then
      [r, s] ← [s, (r − s)];      // swap //
      [u, v] ← [(u − v), u];
    else
      [r, s] ← [r, (s − r)];
      [u, v] ← [(v − u), v];
    while (s_0 = 0) do
      [r, s] ← [r, (s/x)];
      [u, v] ← [u, (vx)];    k ← k + 1;
return ux^k, k
```

Algorithm 8 as shown requires three counters (more efficient hardware implementations are possible). First, two independent counters are needed to keep track of $\deg(s)$ and $\deg(r)$. Note that these counters are not simple incrementer-decrementers since at each **swap** a new value of s with a possible new degree value is computed. The difference of these two counters is used for the **swap** condition checking. Next, a third counter is required to keep track of total number of shifts of s. This third counter is represented by k in Algorithm 8. A relationship between k and the degree of r and s at each iteration can be found but its implementation may not be more efficient hence it is not explicitly shown or used here.

In Algorithm 8 the result sequence follows a transformation due to Berlekamp [12] which is identified by an reverse shifting of result sequence with respect to the gcd sequence. This is particularly useful since no modulo reductions applied to partial results are required. Only a final modulo reduction after multiplication by x^{-k} is needed. Hence, this algorithm may have efficient implementation where intermediate modulo reductions are expensive, *e.g.*, software implementations, and on the other hand, for certain classes of irreducible

polynomials the final multiplication by x^{-k} and modulo reduction can be implemented efficiently [96]. However, in [113], it is shown that Algorithm 8 is not suitable for other software implementations.

In [33], a comparison between Algorithm 8 and 7 in computing both inversion and division has shown that only under certain choices of *suitable* irreducible polynomials, Almost inverse algorithm has a better performance as reported in other papers. Let $F(x) = \sum_{i=0}^{m} f_i$ be irreducible polynomial and $l = \min\{i \geq 1 \mid f_i = 1\}$. Now, "the irreducible polynomial is said to be *suitable* if l is above some threshold which may depend on the implementation, *e.g.*, $l \geq 32$ ... since, then, less effort is required in the reduction step." [33].

3.3 Left-Shift GCD Algorithm, Left-Shift EEA

In the previous six algorithms, the right-shift (or low-end or binary or Stein's) GCD Algorithm and its extended alternatives have been reviewed. A set of algorithms, dual to above, based on a direct modification of Euclidean GCD computation exists which is called left-shift or high-end GCD class of algorithms. These algorithms perform *GCD-preserving* transformations based on checking the value of the higher order coefficients of polynomials or most significant bit (msb) of integers. This class of algorithms is advantageous where polynomial or integers must be processed from the highest degree (or msb) first. Moreover, this class of algorithms can keep track of the *exact* difference of degree of polynomials (or size of integers), and where a least or optimal number of iterations per GCD computation may be achieved. Also, a specific variant of EEA of this class can compute multiplicative inverses with no modular reduction step at all.

All three tasks mentioned in the right-shift class are required in left-shift algorithms as well. The main difference regards the implementation of Task 1, which is discussed next.

In left-shift class of algorithms, in order to compute the quotient q or mainly the remainder of a division step of Algorithm 2, a series of stepwise transformations is used which mimics a "long division". The "long division" algorithm described by many authors [15, 17, 40, 119] is shown in Algorithm 9. (The application of Algorithm 9 to implement a systolic structure for inversion and division over extension binary fields is described in [17].

Algorithm 9 Polynomial Long Division Algorithm.

input: r and s polynomials.

output: $r \pmod s$.

$\delta \leftarrow \deg(r) - \deg(s)$

while $(\delta \geq 0)$ **do**

$[r, s] \leftarrow [(r - x^{\delta}s), s]$;

$\delta \leftarrow \deg(r) - \deg(s)$;

return r

In Algorithm 9 consecutive values of δ, if exist, represent the coefficients of quotient polynomial q of Algorithm 2, and the corresponding partial remainders of these "long division" steps are computed using simple subtractions accordingly. Algorithm 9 ensures to end with $(\deg(r) < \deg(s))$. In fact, at **input** no pre-condition on the degrees of r and s is imposed. It can be noted that Algorithm 9 is a *GCD-preserving* transformation. Algorithm 9 can be used for integers without any modification as well.

Next, a single division step of a GCD computation or EEA, as in Algorithms 1 and 2 can be substituted by a "long division", and a complete GCD computation or EEA can be implemented by as a series of "long-division" algorithm, plus a polynomial (or integer) **swap** mechanism between each two consecutive "long divisions". Hence, only simple operations: shift, subtraction and a mechanism to keep track of difference of the degree of polynomials are sufficient to complete GCD computation or EEA which are exactly the same operations seen in right-shift binary GCD schemes.

Left-shift class of GCD computation algorithms must ensure that the *exit* condition of Algorithm 1 seen in Section 3.1.1 holds at all iterations. This is said to be $(s \leq r)$. At each iteration of Algorithm 9, if an implementation ensures that the left-shifted value $x^{\delta}s$ is computed on-the-fly or it is not written over s, the *exit* condition holds. In other implementations where the shifted value of s are saved, either an auxiliary polynomial (storage) must be used or, more interestingly, a mechanism to keep track of the number of shifts of s must be in place. The former may be called a *restoring* and the latter a *non-restoring* gcd transformation sequence. While the former can be used for both polynomial or integer GCD computation, the latter is applicable only to polynomial GCD computation.

In a *non-restoring* case, a refinement of Algorithm 9 into an exact single step shifting

with re-ordered outputs can be devised as shown in Algorithm 10.

Algorithm 10 Swapped Shifted Polynomial Long Division Algorithm.

input: r and s polynomials with $\deg(r) > \deg(s)$.
output: $x^{\delta}(r \pmod s),\ x^{\delta}s$.

```
δ ← 0
while (deg(r) > deg(s)) do
    δ ← δ + 1
    s ← xs;
[r, s] ← [s, (r − s)];      // first alignment, and overwriting of r //
while (δ > 0) do
    s ← xs;
    if (deg(r) = deg(s)) then
        s ← (r − s);
return r, s
```

Algorithm 10 is a modified version of Algorithm 9 where each polynomial is shifted one step at a time and the degree of shifted polynomials are not restored afterward. Here, a pre-condition is defined as the degree of its dividend to be greater than the degree of its divisor. And as a post-condition, it ensures that the degree of its shifted remainder is less than the degree of its shifted divisor at its termination.

In Algorithm 10, a δ times shifted version of final remainder and the divisor are returned. Initially, the degree of dividend is greater than the degree of divisor by δ units. For δ iterations a divisor polynomial is left shifted. At a first alignment iteration where the degree of shifted divisor is reached the degree of dividend, a remainder polynomial is computed which may have a degree greater, equal or less than the pre-shifted divisor, but always its degree is less than the degree of the shifted divisor. Next, a "virtual" restoring of divisor starts. That is the degree of the shifted divisor is not decreased, but rather in the next δ iterations, the remainder or partial remainders are left shifted where none, one or more other alignments may occur and new partial remainders may be computed. After exactly δ iterations, the pre-shifted divisor is "virtually" restored, and in all these δ iterations the remainder polynomial is continuously updated to a degree less than the shifted divisor. Thus, at the termination of each "long division" operation, its post-condition is ensured.

The first advantage of a stepwise "virtually" restoring implementation and a direct consequence is the possibility to compute as exact upper bound of the number of iterations of Algorithm 1, and to implement Task 3 very efficiently. Again the **while** loop of Algorithm 1 can be re-defined as a **for** loop of $2m - 1$ iterations as also seen in Algorithm 7.

The main advantage of Algorithm 10 is that Task 2 can be implemented efficiently with no **swap** of polynomials, but rather a selective overwriting of one of them. This also means that the same polynomial is always shifted. Algorithm 10 indicates that the same shifting polynomial s which represents the remainder of one "long division" operation will be the divisor of the next operation. As a consequence, no **swap** of polynomials at the end of a "long division" is needed. Thus, the end of a "long division" uniquely corresponds to a reset of δ, and δ is needed only to distinguish between an overwriting condition (or a first alignment case as indicated in Algorithm 10) and other alignment cases.

It can be seen that left-shift polynomial GCD algorithms, which process the polynomials from the higher order coefficients, may cause polynomials to shrink (or zeroed) from either higher order degrees in the restoring case, or lower order degrees in the non-restoring one. In the non-restoring case, the keeping track of the number of shifts of s, which is the same as Task 2 defined before, can be implemented using a simple counter or using alternative methods such as a shifting indicator (as a ring counter) pointing to the lowest order coefficient of s.

In the following extended Euclidean algorithms based on left-shift GCD computation and their implementation issues with regard to multiplicative inversion and division over Galois fields are discussed.

3.3.1 Berlekamp's Inversion Algorithm with No Modulo Reduction

In order to compute only multiplicative inverses over finite fields using EEA, an area efficient hardware implementation is proposed by Berlekamp in [12]. It requires a minimum number of registers, *i.e.*, no auxiliary registers, and most importantly it uses the least number of logic computations since its result transformation sequence requires no modulo reduction. This is achieved by processing (shifting) a pair of interchanged polynomials

of the result transformation sequence in the reverse order of the polynomials of the gcd transformation sequence. First, this ensures that each two corresponding polynomials of gcd and result sequence may fit in an $m+2$ storage space, where m is the degree of the larger polynomial, *i.e.*, irreducible polynomial. Next, more importantly, it ensures that the degree of the result sequence polynomials never reaches m. In [12], an implementation based on this idea which requires a maximum of $2m+1$ iterations to complete is proposed.

This compact implementation also provides a third advantage which will be described as *shifted result problem* in detail later. Specifically, in the case of Berlekamp's variant, at the *exit* condition which is recognized by the number of shifts of either polynomials of the gcd sequence reaching m, the correct result without any further manipulation is available but in the reverse order. In the centralized control implementation proposed in [12], a combined counter-like structure for the *exit* condition and also to keep track of the difference of the degree of polynomials is proposed which may not be suitable for large values of m. A systolic proposal based on this algorithm is presented in [9] with application in Reed-Solomon decoders.

In the following we review a systolic variant of Berlekamp's proposal. An algorithm based on Berlekamp's description but optimized for systolization by using a fixed number of iteration is proposed by Yan and Sarwate in [119] which is shown as Algorithm 11.

In Algorithm 11, the major innovation of Berlekamp can be seen. At initialization, the pair $[u, v]$ are initialized by $[x^m, 0]$ instead of $[0, 1]$ as it is commonly used in EEA, *i.e.*, Algorithm 2. Next, at each iteration while s is left-shifted, its counterpart v is right-shifted. In practice, as all lower order coefficients of s are moving toward m to be XORed with $r_m = 1$, the polynomial v starting from x^m generates all coefficients of degree 0 to $m-1$. Also, the sum of these two degrees is always m. Note that this is not by itself a characteristic of Berlekamp variant of inversion algorithm but proposed in other architectures,*e.g.*, [47], as well. In a centralized control design this is advantageous since both polynomials can fit in an $m+2$ bit register. This may not be trivially implemented in a distributed control, *i.e.*, systolic, implementation. But in any case, since the result sequence polynomials require no modulo reduction, the Berlekamp-type designs result in the least critical delay path circuits when distributed control (no counters) are used. On the other hand, Algorithm 11 provides two systolic optimizations w.r.t. Berlekamp's idea.

Algorithm 11 Variant of Berlekamp's EEA For Inversion.

input: $f(x)$ irreducible binary polynomial of degree m, $a(x)$ a polynomial of degree $< m$.
output: $a(x)^{-1} \mod f(x)$.

```
d ← 0;  [r, s] ← [f(x), a(x)]  [u, v] ← [x^m, 0]
for ((2m) times) do
    if ((s_m = 1) & (d < 0)) then
        d ← −d;        // swap //
        [r, s] ← [s, r];
        [u, v] ← [v, v];
    if (s_m = 1) then
        [r, s] ← [r, s + r];
        [u, v] ← [u + v, v];
    d ← d − 1;
    [r, s] ← [r, xs];
    [u, v] ← [u, v/x];
return v
```

First, it uses a fixed number of iterations for the *exit* condition. This ensures that the correct result is returned in the same polynomial v always.

A comparison with the almost inverse algorithm as described in Algorithm 8 is instructive. Also, in Algorithm 8, the result sequence polynomials $[u, v]$ are initialized with $[x^m, 0]$. Hence, no intermediate modulo reduction applied to the result sequence polynomials is required. But, in Algorithm 8, the gcd sequence follows a binary GCD algorithm type transformation. And at the *exit* condition which is recognized by s reaching zero, a variable number of shifts applied to v has to be known, hence k counter is needed.

We conclude that a generalization of extended Euclidean algorithm and extended binary GCD is possible. Two pairs $[r, s]$ and $[u, v]$ must follow the similar *GCD-preserving* transformations, but the pair $[r, s]$ may follow either right or left-shift GCD computation, while the pair $[u, v]$ may follow any of the four *shifted result* solutions which will be described next.

3.3.2 Shifted Result Problem

It has been said that there are two alternatives to implement the left-shift polynomial extended Euclidean algorithm called as restoring and non-restoring variants. In non-restoring implementations of EEA, the *shifted result* problem may be stated as "while it is advantageous to use the shifted (non-restored) polynomials $[r, s]$ to keep $\deg(r) = m$ always, the same is not true for polynomials $[u, v]$ and they must be restored." In fact, we know that for any multiplicative inversion algorithm based on Euclidean GCD, at a certain iteration, *i.e.*, $i = j$, the invariant of Equation (3.3) becomes:

$$w_j \; r + u_j \; s = 1. \tag{3.4}$$

where u_j is the inverse of $s \mod r$. In a non-restoring implementation, at *exit* condition defined as $r = s = x^m$, the invariant of Equation (3.4) becomes

$$w_j x^m \; r + u_j x^m \; s = x^m. \tag{3.5}$$

Hence, in these implementations, instead of u_j, a polynomial $u_j x^m$ may be returned.

Different strategies to tackle this problem of restoring $u_j x^m$ are proposed: The first solution applied to binary finite fields is already discussed which is the Berlekamp's variant above. In this case, initializing $[u, v] = [x^m, 0]$, next, at each iteration while the gcd sequence is updated as $r \leftarrow r + x^d s$, the result sequence is updated as $u \leftarrow u + v/x^d$.

The second solution is to initialize $[u, v] = [0, x^{-m}]$ [56] and it is discussed next. The third and fourth solutions are multiplying or dividing $U^{(i-1)}(x)$ alternatively [15] or using an auxiliary polynomial [37].

3.3.3 Inversion Algorithm without Conditional Branching.

The second solution to the shifted result problem is proposed by Huang and Wu in [56], shown in Algorithm 12.

The pair $[u, v]$ is initialized with $[0, x^{-m}]$. Note that $x^{-m} \mod f(x)$ is a polynomial with degree $< m$. Clearly, the final result after $2m$ iterations will be corrected, since according to Equation (3.5), $x^{-m} u_j x^m = u_j$. But in this case at each iteration the updated polynomial v may require a modulo reduction since its degree may reach m or more. It

Algorithm 12 Inversion Algorithm with no Conditional Branching (no **swap**).

input: $f(x)$ irreducible binary polynomial of degree m, $a(x)$ a polynomial of degree $< m$.
output: $a(x)^{-1} \mod f(x)$.

```
[r, s] ← [f(x), a(x)]   [u, v] ← [0, x^{-m}]
for ((2m) times) do
  if (s_m = 1) then
    [r, s] ← [s, x(s + r)];
    [u, v] ← [v, x(u + v) mod f(x)];
  else
    [r, s] ← [s, xs];
    [u, v] ← [v, xv mod f(x)];
return u
```

may be noted this is not a very efficient method not only for successive modulo reductions which could be avoided for only inversion algorithms, but mainly for the need for a pre-computation of the x^{-m} if a fixed irreducible polynomial is not used.

However, the major innovation of Algorithm 12 is not the solution to the shifted result problem, but rather in its GCD computation without any conditional branching (no **swap**). This results in extremely reduced logic, no counters, no swapping mechanism and a very reduced gate delay implementation. But all this is achieved at the expense of increasing the worst case number of iterations from $2m$ to m^2. Not only this increases the time complexity to $O(m^2)$, but also, in systolic architectures where each iteration is mapped into a different processing element, it may require an $O(m^2)$ area complexity. In general, such complexity figures may off balance the reduced delay path and less logic due to counter free structure completely.

3.3.4 Inversion Algorithm with Right-Left-Shift of Result Sequence

In [15], Brunner et al. proposed a multiplicative inversion algorithm over Galois fields GF(2^m) shown in Algorithm 13. *GCD-preserving* transformations.

The modified EEA in Algorithm 13, follows a direct use of "long-division" algorithm

Algorithm 13 Inversion Algorithm with Right-Left-Shift of Result Sequence.

input: $f(x), a(x)$.
output: $a^{-1}(x) \mod f(x)$.

```
d ← 0;  [r, s] ← [f(x), a(x)];  [u, v] ← [0, 1];
for (2m times) do
  if (s_m = 0) then
    s ← xs;  v ← xv mod f;  d ← d + 1;
  else
    if (r_m = 1) then
      r ← r + s;  u ← u + v;
    r ← xr;
    if (d = 0) then
      swap(r, s);  swap(u, v);
      v ← xv mod f(x);  d ← d + 1;
    else
      v ← v/x mod f(x);  d ← d − 1;
return v
```

where a stepwise *GCD-preserving* transformation of pair $[r, s]$ is shown. The implicit *exit* condition is mapped into a **for** loop of $2m$ iterations. The important aspect of Algorithm 13 is the asymmetric right and left-shift of v required to solve the shifted result problem. While the counter d is zero or increasing, $v \leftarrow vu \mod f(x)$, else while it is decreasing, then $v \leftarrow v/x \mod f(x)$.

Brunner et al. proposed a centralized control implementation with the up/down counter d which keeps track of degree of polynomials. This centralized control design has 3 global control signals, one carry for modulo reduction, and bidirectional data lines between adjacent registers. It is said that the cycle time is independent of field-size since no global data propagation occurs. That's the delay of variable size counter structure and long interconnect of global control signals are ignored. This assumption for a centralized control implementation is valid only when the worst case delay path of control unit is less than clock period requirement. It is also said that one inversion per 2m cycles is possible considering a parallel load of all operands and parallel read of the result. Both above assumptions

hold for small values of m with respect to the clock period.

In Algorithm 13, and its implementation, two constraints are imposed. First, the explicit **swap** condition occurs only at $d = 0$ where a "long-division" is complete and the other will start next. Next, the pair $[u, v]$ are implemented as m-bit registers which must not overflow. Under these constraints, even an inversion only algorithm will require a modulo reduction. Further, this inversion only algorithm cannot be extended to perform division easily. The problem is that the required modulo reduction applied to the result sequence requires two different modulo reduction at both ends of u, when its degree either overflows $m - 1$ or when it underflows 0. In this case, the asymmetric shifting of $[u, v]$ is cumbersome.

On the other hand, in other cases, such as bit serial systolic architectures, this asymmetric shifting may be handled more efficiently.

A more elaborate systolic variant of right-left-shift scheme over GF(p^m) is proposed by Fitzpatrick et al. in [32] to implement a Reed-Solomon encoder where coefficients of subfield elements are processed in each systolic cell and moreover the input polynomials may have powers of p as factors.

3.3.5 Division Algorithm with a Two-step Shifted Result Solution

A variant of Algorithm 13 is proposed by Guo and Wang in [40] which avoids such an asymmetric shifting and is able to compute division. This division variant is shown in Algorithm 14.

Algorithm 14 inherits the constraints described for Algorithm 13. But its advantage w.r.t. Algorithm 13 is the uniform left-shift operation of $u \leftarrow xu \mod f$ at all cases with $(r_m = 1)$. This simplifies the implementation of **Part a** of algorithm since no selective modulo reduction is used. Further, a uniform modulo reduction allows the inversion algorithm can be extended to perform the division at no extra cost (or modification). However, as it said before, this uniform shifting results in the shifted result problem. As a matter of fact, this algorithm is the best descriptive example of this problem. Hence, **Part b** is required explicitly to fix this problem.

Algorithm 14 Left-Shift Scheme Division with Successive Correction Step.

input: $f(x), a(x), b(x)$.
output: $b(x)/a(x) \mod f(x)$.

```
d ← 0;  [r, s] ← [f(x), a(x)];  [u, v] ← [0, b(x)];
Part a
for (2m times) do
  if (s_m = 0) then
    s ← xs;  v ← xv mod f;  d ← d + 1;
  else
    if (r_m = 1) then
      r ← r + s;  u ← u + v;
    r ← xr;  u ← xu mod f;
    if (d = 0) then
      swap(r, s);  swap(u, v);  d ← d + 1;
    else
      d ← d − 1;
Part b
for (m times) do
  v ← v/x mod f(x)
return v
```

However, a combined implementation of **Part a** plus **Part b** may be very inefficient. Specially, in an implementation where a last in first out (LIFO) stack structure between these two parts is proposed to reverse the order of polynomial v. Such implementation requires twice the number of iterations of a typical inversion/division algorithm which results in a double latency and area complexity for a systolic architecture.

3.3.6 Division Algorithm with Auxiliary Polynomial for Partial Remainder

The fourth solution to the shifted result problem is to transform the non-restoring model into a restoring one. In fact, an auxiliary polynomial t can be used to hold the partial remainders. At the start of each "long-division", it initializes $t \leftarrow u$. During each "long-

Algorithm 15 Division Algorithm with Auxiliary Polynomial for Partial Remainder.

input: $f(x), b(x)$.
output: $a(x)/b(x) \mod f(x)$.

```
d ← 0;  sign_d ← 0;  [r, s] ← [f(x), b(x)];  [v, u] ← [0, a(x)];  t = 0;
for (2m times) do
    s ← xs    t ← xt mod f(x)
    if (sign_d = 0) then
        d ← d + 1;
        if (s_m = 1) then
            [s, r] ← [r + s, s];  t ← u;    sign_d ← 1
    else
        d ← d − 1;
        if (s_m = 1) then
            s ← r + s;  t ← t + u;
        if (d = 0) then
            [u, v] ← [t + v, u];    sign_d ← 0
return v
```

division" while $(t \leq v)$, it performs $[t, v] \leftarrow [t + x^d v, v]$ and finally at the end of each "long-division", it updates the pair $[u, v] \leftarrow [u + x^d v, u]$. In practice, with this restoring method, the shifts required for the computation of partial remainders which correspond to the terms of each q in Algorithm 2 are not reflected in the dividend of the next "long-division" cycle. A variant of this solution due to Guo and Wang [37, 39, 41] optimized for systolization is shown in Algorithm 15.

In Algorithm 15, polynomial t represents the partial remainders computed during each cycle, u represents the dividend and v the divisor of each "long-division" cycle. Note that t is shifted in all $2m$ iterations, however, its content is overwritten by u at the first *alignment* of r with s equivalent to the start of a new "long-division".

Once again, a counter d is used for keeping track of the degree of polynomials $[r, s]$. For efficient systolization with no carry propagation structure, the sign and magnitude of d are separately implemented where it assumes the sign of $d = 0$ be 1, as if negative. This algorithm can be rewritten in a more compact form if we let $d \leftarrow d + 1$; and to check for

($d > 0$) instead of sign of d equals to 0 at each alignment where r is overwritten. However, in a systolic structure the same circuitry will be implemented. It can be seen that the polynomial t acts as the partial remainders while ($d < 0$), that is while the same divisor is used to compute the successive terms of the same q. At each ($d = 0$) while ($sign_d = 1$) a new dividend is generated.

In practice, this algorithm is a restoring alternative and we know that in such a case no modulo reduction is required to compute multiplicative inverses. On the other hand, in Algorithm 15, the modulo reduction is used to compute divisions.

A similar restoring solution was initially proposed by Horng and Wei in [55]. Their semi-systolic proposal is a direct map of Algorithm 2 into two semi-systolic architectures, one to compute each term of q and to pass it to semi-systolic multiplier for the consecutive updating of pair u's and v's using the intermediate polynomials t's as necessary, hence a restoring alternative. In this semi-systolic design there are 8 global control signals corresponding to 2 up-down counters, 2 loads and one add signal.

3.4 Summary

In this chapter, a comparative review of EEA and binary (Stein's) GCD computation and in particular those implementations relevant to the inversion and division over Galois fields has been presented. Some basis steps in common among these have been discussed and their differences are highlighted.

In the next chapter, a different view of computing multiplicative inversion and division using EEA will be explored. The specific case of an inversion algorithm using triangular basis will be described. Further, a common set of algorithms to compute inversion and division based on EEA with both polynomial and triangular basis will be presented.

Chapter 4

Inversion using Double-Basis representation

In this chapter, computing multiplicative inverses by solving a set of linear equations will be reviewed. The specific case of the inversion using double-basis representation, and its low complexity, area efficient implementation based on EEA will be discussed. The description follows that was introduced in [46, 49]. Part of the work in this chapter has been presented in [16, 18, 19].

4.1 Single-Basis Inversion

Let $GF(2^m)$ be an extension field of $GF(2)$ defined by the irreducible polynomial $F(x) = \sum_{i=0}^{m} f_i x^i$, $f_i \in \{0, 1\}$, and ω be a root of $F(x)$ such that $F(\omega) = 0$. Then $\Omega = \{1, \omega, \cdots, \omega^{m-1}\}$ is a polynomial basis. Let us consider two elements $A, A^{-1} \in GF(2^m)$

and rewrite the equation $A^{-1} \cdot A \equiv 1 \mod F(x)$ as

$$\begin{aligned} 1 &= \left(\sum_{j=0}^{m-1} b_{\Omega j}\omega^j\right)\left(\sum_{i=0}^{m-1} a_{\Omega i}\omega^i\right) \pmod{F(\omega)} \\ &= \sum_{j=0}^{m-1} b_{\Omega j} \sum_{i=0}^{m-1} a_{\Omega i}\omega^{i+j} \pmod{F(\omega)} \\ &= \sum_{j=0}^{m-1} b_{\Omega j} \sum_{i=0}^{m-1} a_{\Omega i} \sum_{l=0}^{m-1} p_{\Omega l}^{[i+j]}\omega^l \end{aligned} \tag{4.1}$$

where $p_{\Omega l}^{[i]}$ is the lth coordinate of ω^i.

Introducing the matrix $\mathcal{A}(\underline{A}_\Omega)$ with elements $\mathcal{A}_{i,j} = \sum_{l=0}^{m-1} a_{\Omega l} p_{\Omega i}^{[l+j]}$, Equation (4.1) can be written as

$$\underline{1}_\Omega = \mathcal{A}(\underline{A}_\Omega)\ \underline{A^{-1}}_\Omega. \tag{4.2}$$

The matrix $\mathcal{A}(\underline{A}_\Omega)$ is uniquely defined by the coefficients of defining irreducible polynomial $F(x)$ and the coordinates of element A. More precisely, it is an $m \times m$ matrix over $GF(2)$:

$$\mathcal{A}(\underline{A}_\Omega) = [\underline{A}_\Omega, \underline{A\omega}_\Omega, \cdots, \underline{A\omega^{m-1}}_\Omega]. \tag{4.3}$$

It can be shown that, with m bits memory, the formation of the matrix in Equation (4.3) requires $O(m^2)$ arithmetic operations over $GF(2)$. Then, the inverse A^{-1} is obtained by solving the system of linear equations (4.3), which requires $O(m^3)$ arithmetic operations over $GF(2)$. However, a direct application of Equation (4.3) is computationally more complex than all methods seen so far.

As an alternative implementation which does not require the matrix formation by computation but simple wiring, the following proposal can be mentioned. In [21], Davida has proposed a method to compute the multiplicative inverse of an element in extended field $GF(2^m)$ by solving an augmented set of linear equations as follows. Let consider $F(x) \cdot U(x) + A(x) \cdot A'(x) = 1$, where $U(x) = \sum_{i=0}^{m-2} u_i x^i$, and

$$deg(A(x)), deg(A'(x)) \leq m-1 \text{ and } deg(U(x)) \leq m-2.$$

In matrix form, it can be written as

$$\begin{pmatrix} f_0 & 0 & \dots & 0 & a_0 & \dots & 0 & 0 \\ f_1 & f_0 & \dots & 0 & a_1 & a_0 & \dots & 0 \\ f_2 & f_1 & f_0 & \dots & a_2 & a_1 & a_0 & \dots \\ \vdots & & & & \vdots & & & \vdots \\ \dots & \dots & f_1 & f_0 & a_{m-1} & \dots & a_1 & a_0 \\ f_m & f_{m-1} & \dots & f_1 & 0 & a_{m-1} & \dots & a_1 \\ 0 & f_m & \dots & f_2 & 0 & \dots & \dots & \vdots \\ \vdots & & & & \vdots & & & \vdots \\ 0 & 0 & \dots & f_m & 0 & 0 & 0 & a_{m-1} \end{pmatrix} \times \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{m-2} \\ a'_0 \\ a'_1 \\ \vdots \\ a'_{m-1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \quad (4.4)$$

Matrix Equation (4.4) can be transformed into $(2m-1)$ equations in $(2m-1)$ unknowns. Since every nonzero element has an inverse, a solution for $(2m-1)$ unknowns always exists. Coefficients f_i's and a_i's are known, therefore a direct solution for m coefficients a_i's can simultaneously be computed. Davida suggested a parallel implementation for computing a_i's using AND-XOR nets. This proposal would only be feasible for fields of small dimension.

Another way to obtain a low complexity and area efficient implementation is to use a double-basis inversion technique, polynomial and triangular bases at the same time.

4.2 Double-Basis Inversion

Recall the irreducible polynomial $F(x)$, the polynomial basis Ω and the triangular basis Λ as defined in Section 2.1.4. Further let us recall the linear transformation matrix (M^{-1}) as in Equation (2.4). Then, conversion of a given element $A(x)$ from polynomial to triangular

basis can be written in matrix form $\underline{A}_\Lambda = (M^{-1})\underline{A}_\Omega$ where M^{-1} is

$$M^{-1} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 1 & h_1 \\ 0 & 0 & 0 & \dots & 1 & h_1 & h_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 1 & h_1 & \dots & h_{m-4} & h_{m-3} & h_{m-2} \\ 1 & h_1 & h_2 & \dots & h_{m-3} & h_{m-2} & h_{m-1} \end{pmatrix}, \tag{4.5}$$

and $h_i = \sum_{j=0}^{i-1} f_{m-i+j} h_j$, for $1 \le i \le m-1$, and $h_0 = 1$.

For an inversion operation, the above transformation is not necessary directly, and this transformation can be combined into Equation (4.2) by pre-multiplying both sides of Equation (4.2) by the matrix in (4.5), which results in

$$\underline{1}_\Lambda = H(\underline{A}_\Lambda)\underline{A^{-1}}_\Omega \tag{4.6}$$

where $\underline{1}_\Lambda$ is the triangular basis representation of the multiplicative *identity* element of $GF(2^m)$ and

$$H(\underline{A}_\Lambda) = [\underline{A}_\Lambda, \underline{A\omega}_\Lambda, \cdots, \underline{A\omega^{m-1}}_\Lambda]$$

is a Hankel matrix built upon $\underline{A}_\Lambda$. Given the element $A(x)$ in polynomial basis, the coefficients of inverse element $A^{-1}(x)$ in polynomial basis can be computed by solving Equation (4.6).

In general, the formation of coefficients of the Hankel matrix requires recursive formulae such as

$$h_i = \begin{cases} a_{\Lambda i} & 0 \le i \le m-1, \\ \sum_{j=0}^{m-1} h_{i-m+j} f_j & m \le i \le 2m-2, \end{cases} \tag{4.7}$$

which can be implemented by using linear feedback shift registers (LFSR) with $F(x)$ as the *feedback* polynomial and $S(x)$ as the *seed* polynomial defined by

$$S(x) = \sum_{j=0}^{m-1} a_{\Lambda j} x^j = \sum_{j=0}^{m-1} h_j x^j \tag{4.8}$$

$$= H(x) \pmod{(x^m)}. \tag{4.9}$$

In [46], it is shown that the computational complexity for the formation of the Hankel matrix is equal to that of $A(\underline{A}_\Omega)$ in Equation (4.2), *i.e.*, $O(m^2)$, and it is proposed that by a suitable choice of field defining irreducible polynomial, such as low Hamming weight trinomials of type x^m+x+1, the formation of the coefficients of the Hankel matrix requires no recursion and it can be generated using single XOR gate paths. That is, since $f_j = 0$ for $2 \leq j \leq m-1$, and only $f_0 = f_1 = 1$, then

$$h_i = h_{i-m+1} + h_{i-m} = a_{\Lambda i-m+1} + a_{\Lambda i-m} \qquad m \leq i \leq 2m-2.$$

For all practical purposes, there exists an irreducible trinomial or pentanomial [98], hence a suitable choice of them will be useful. For general trinomials, $x^m + x^k + 1$, with $k \leq \lfloor m/2 \rfloor$, each coefficient can be generated by 2 XOR gates at maximum (worst case delay path). With a similar reasoning as above, only $f_0 = f_k = 1$ and all other $f_i = 0$, then let $\Delta = m - k$, for $m \leq i \leq m + k - 1$, $h_i = h_{i-m+k} + h_{i-m}$, with both right hand terms directly available from the coordinates of A. For $m + k \leq i \leq 2m - 2$,

$$h_i = h_{i-m+k} + h_{i-m} = (h_{i-\Delta}) + h_{i-\Delta-k} \text{ with } i - \Delta > m,$$

$$h_i = (h_{i-2\Delta} + h_{i-2\Delta-k}) + h_{i-\Delta-k},$$

with all three right hand terms directly available from A. These complexity figures and ones that follow can be computed in the same manner as those of entries for the multiplication matrix in Mastrovito multipliers [43].

For pentanomials, $x^m + x^k + x^j + x^i + 1$, where $m > k > j > i > 0$, it can be shown that the critical path is dependent on $\Delta = m - k$. More precisely, the number of XOR gates required to generate each entry are always less than or equal to $C = \frac{m-2}{\Delta} + 3$, with a worst case delay path $\lceil \log_2 C \rceil$. Specifically, the worst case corresponds to pentanomials of type $x^m + x^{m-1} + x^j + x + 1$. However, in [98], it is shown that for pentanomials up to degree 10000, for the degrees where no trinomial exists, the value of k can be quite low which will result in very low values of the critical path.

In [34], Furness et al. have shown that for trinomials, the triangular basis is a simple permutation of the polynomial basis, and for pentanomials this conversion requires a minimal hardware and a reordering of the coordinates of the element. For general irreducible

polynomials, the conversion between a polynomial basis representation and its corresponding triangular basis representation can be systolized with techniques similar to the one used in bit serial dual basis systolic multipliers [29]; One may conclude that the Hankel matrix formation has a lower computational complexity compared to solving Equation (4.6), and either computing or pre-computing coefficients of the Hankel matrix, Equation (4.6) can be solved for $a'_{\Omega i}$'s with $O(m^2)$ $GF(2)$ arithmetic operations [100], as opposed to $O(m^3)$ required by Equation (4.2).

4.3 Inversion by applying EEA to a Hankel Matrix

In [100], Sugiyama has proposed some generalized algorithms to solve Discrete-Time Wiener-Hopf (DTWH) equations based upon Euclidean algorithm. The DTWH equations are represented by a set of t linear inhomogeneous equations with t unknowns λ_k $(k = 0, 1, \cdots, t-1)$, $(2t-1)$ constant coefficients s_k $(k = 0, 1, \cdots, 2t-2)$, not all zeros, and t constants b_k $(k = 0, 1, \cdots, t-1)$ such that

$$\begin{pmatrix} s_{t-1} & s_{t-2} & \cdots & s_1 & s_0 \\ s_t & s_{t-1} & \cdots & s_2 & s_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ s_{2t-3} & s_{2t-4} & \cdots & s_{t-1} & s_{t-2} \\ s_{2t-2} & s_{2t-3} & \cdots & s_t & s_{t-1} \end{pmatrix} \times \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{t-2} \\ \lambda_{t-1} \end{pmatrix} = \begin{pmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_{t-2} \\ a'_{t-1} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) can be viewed in a polynomial form as

$$S(x)\Lambda(x) = A^{-1}(x), \tag{4.11}$$

where polynomials $S(x)$ and $A^{-1}(x)$ are defined such as

$$0 \leq deg S(x) \leq 2t-2, \tag{4.12}$$

$$deg A^{-1}(x) \leq t-1, \tag{4.13}$$

for any fixed t. Then, Equation (4.11) can be solved to obtain a trio of polynomials $\{\Gamma(x), \Lambda(x), \Psi(x)\}$ which satisfies the following conditions:

1. $S(x)\Lambda(x) = \Psi(x)x^{2t-1} + A^{-1}(x)x^{t-1} + \Gamma(x)$,
2. $deg\Gamma < t-1$,
3. $deg\Lambda < t$.

Applying EEA type sequence, this trio of polynomials can be computed, [100], such that it requires $O(t^2)$ arithmetic operations for a polynomial of degree $(2t-1)$. Let us recall the Hankel matrix built upon $\underline{A}_\Lambda$ and rewrite Equation (4.6) in matrix form

$$\begin{pmatrix} h_0 & h_1 & \cdots & h_{m-2} & h_{m-1} \\ h_1 & h_2 & \cdots & h_{m-1} & h_m \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ h_{m-2} & h_{m-1} & \cdots & h_{2m-4} & h_{2m-3} \\ h_{m-1} & h_m & \cdots & h_{2m-3} & h_{2m-2} \end{pmatrix} \times \begin{pmatrix} a'_{\Omega 0} \\ a'_{\Omega 1} \\ \vdots \\ a'_{\Omega(m-2)} \\ a'_{\Omega(m-1)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}, \tag{4.14}$$

then, a Hankel matrix with entries $h_{i,j} = h_{i+j}$ is recognized. Let us consider two polynomials

$$\mathcal{H}(x) = h_{2m-2}x^{2m-2} + h_{2m-3}x^{2m-3} + \cdots + h_1 x + h_0,$$

and

$$\underline{A^{-1}}_\Omega(x) = a'_{\Omega(m-1)}x^{m-1} + a'_{\Omega(m-2)}x^{m-2} + \cdots + a'_{\Omega 1}x + a'_{\Omega 0},$$

where $h_i,\ 0 \le i \le 2m-2$, and $a'_j,\ 0 \le j \le m-1$, are elements of the Hankel matrix and column vector of the left hand side of Equation (4.14), respectively. Noting that $0 \le deg\mathcal{H}(x) \le 2m-2$, and $0 \le deg A_\Omega^{-1}(x) \le m-1$, the procedure to solve general DWTH equations using EEA can be applied to solve Equation (4.14).

In [46], a more efficient algorithm which improves upon those in [100] for the special case of computing inverses is proposed.

However, this improved algorithm will be presented in the following in the context of a common set of algorithms based on EEA for computing inversion and division with both polynomial and triangular basis as follows. This common set of EEA algorithms is the one that will be systolized in the next chapter.

4.4 Generalized Polynomial Inversion and Division

In this section two generalized polynomial basis inversion and divsion algorithms using EEA will be discussed. Next, these will be transformed into a novel division algorithm optimized for systolization.

4.4.1 Polynomial Basis Inversion using EEA

Algorithm 16 Polynomial Basis Inversion Algorithm (PBIA) using EEA.

input: $F(x)$, $A(x)$.
output: The inverse of $A(x)$.
Initialization:
$U(x) \leftarrow 0;\ V(x) \leftarrow 1;$
$R(x) \leftarrow F(x);\ S(x) \leftarrow A(x);$
Polynomial Updating:
while $(\deg R(x) > 0)$ **do**
$Q(x) \leftarrow \left\lfloor \frac{R(x)}{S(x)} \right\rfloor;$
$R(x) \leftarrow R(x) - Q(x)S(x);$
$U(x) \leftarrow U(x) - Q(x)V(x);$
swap $(R(x),\ S(x));$ **swap** $(U(x),\ V(x));$
return $U(x)$

The Extended Euclidean Algorithm (EEA) as in Algorithm 16 can be used to compute the inverse of an element. Inputs to Algorithm 16 are $F(x)$, defining irreducible polynomial, and $A(x)$, the polynomial basis representation of element A. At each iteration, Algorithm 16 performs a polynomial division of $R(x)$ by $S(x)$ where the quotient and the remainder are $Q(x)$ and (the new) $R(x)$, respectively. The **swap** instructions of Algorithm 16 make sure that at the end of each iteration $\deg R(x) > \deg S(x)$.

In a hardware efficient design of Algorithm 16, the computation of $Q(x)$ and the following operations inside the **while** can be implemented bitwise. In such implementation, it will be discussed that remainders $R(x)$, $U(x)$ can be computed directly bitwise without a need to compute $Q(x)$ explicitly.

4.4.2 Triangular Basis Inversion Using EEA

In [46], an inversion algorithm is proposed which uses a triangular basis representation for input A and returns its inverse in the polynomial basis according to Equation (4.6). This equation can be solved using different methods for solving general Discrete-Time Wiener-Hopf equations as in [100]. A simplified version of algorithm given in [46] is repeated here as Algorithm 17.

Algorithm 17 Triangular Basis Inversion Algorithm (TBIA) using EEA.

input: $H(x) = \sum_{i=0}^{2m-2} h_i x^i$
output: The inverse element.
Initialization:
$U(x) \leftarrow 0;\ V(x) \leftarrow 1;\ R(x) \leftarrow x^{2m-1};\ S(x) \leftarrow x^{2m-2}H(x^{-1});$
Polynomial Updating:
while $(\deg R(x) > m-1)$ **do**
$Q(x) \leftarrow \left\lfloor \frac{R(x)}{S(x)} \right\rfloor;$
$R(x) \leftarrow R(x) - Q(x)S(x); \quad U(x) \leftarrow U(x) - Q(x)V(x);$
swap $(R(x),\ S(x)); \quad$ **swap** $(U(x),\ V(x));$
return $U(x)$

4.5 Key Results on Inversion and Division using Triangular Basis

In this section, some key results are presented that enable us to develop efficient hardware for finite field inversion and division.

4.5.1 Inversion Algorithm Revisited

The following discussions are mainly centered around Algorithm 17. However, they can be equally applied to Algorithm 16 and EEA in general.

The total number of iterations in Algorithm 17 depends on the initializations of $R(x)$ and $S(x)$. Let us denote these initial polynomials as $R^{(0)}(x)$ and $S^{(0)}(x)$, respectively, In the exit condition of the **while** loop, if the value, which deg $R(x)$ is compared with, is changed then the total number of iterations is also affected. Let us denote this comparison value as μ. Thus the triplet $\mathcal{T} = \left\langle R^{(0)}(x),\ S^{(0)}(x),\ \mu \right\rangle$ completely determines the total number of iterations and, hereafter, the latter is denoted as $I(\mathcal{T})$. In Algorithm 17, this triplet is $\langle x^{2m-1},\ x^{2m-2}H(x^{-1}),\ m-1\rangle$. For any integer k, define another triplet $\tilde{\mathcal{T}} = \left\langle x^k R^{(0)}(x),\ x^k S^{(0)}(x),\ \mu + k \right\rangle$. Note that when k is negative, deg $x^k R^{(0)}(x)$ and/or deg $x^k S^{(0)}(x)$ can be negative. The degree of the zero polynomial is assumed to be ∞. Now, we state the following lemma.

Lemma 1 *Using the notations given above, we have*

$$I(\mathcal{T}) = I\left(\tilde{\mathcal{T}}\right). \tag{4.15}$$

Before giving a proof of the above lemma, we consider another aspect of Algorithm 17. It is clear that this algorithm generates a sequence of quotient polynomials, one quotient in each iteration. These quotients are determined by the initial polynomials: $R^{(0)}(x)$ and $S^{(0)}(x)$. The length of the sequence depends on and is equal to the total number of iterations. This, in turn, implies that the sequence is determined by $\mathcal{T}$. Let us denote the sequence as $\mathcal{Q}(\mathcal{T})$.

Lemma 2 *With the notations given above, the following holds.*

$$\mathcal{Q}(\mathcal{T}) = \mathcal{Q}\left(\tilde{\mathcal{T}}\right). \tag{4.16}$$

It is worth pointing out a direct implication of Lemmas 1 and 2. For any integer k, either positive or negative, if we multiply the initialization polynomials $R^{(0)}(x)$ and $S^{(0)}(x)$ by x^k

and add k to the comparison value of the exit condition, then the resultant total number of iterations and the sequence of quotient polynomials are same as those without the above mentioned changes.

Proof of Lemmas 1 and 2: Let $e = I(\mathcal{T})$ and $\tilde{e} = I\left(\tilde{\mathcal{T}}\right)$. For $1 \leq i \leq \min\{e, \tilde{e}\}$, let $Q^{(i)}(x)$ and $\tilde{Q}^{(i)}(x)$ denote the quotients after i iterations resulting from the use of $\mathcal{T}$ and $\tilde{\mathcal{T}}$, respectively. Similarly, denote $R^{(i)}(x)$, $S^{(i)}(x)$, $\tilde{R}^{(i)}(x)$, $\tilde{S}^{(i)}(x)$, etc. Clearly,

$$\tilde{Q}^{(1)}(x) = \left\lfloor \frac{\tilde{R}^{(0)}(x)}{\tilde{S}^{(0)}(x)} \right\rfloor = \left\lfloor \frac{x^k R^{(0)}(x)}{x^k S^{(0)}(x)} \right\rfloor = Q^{(1)}(x),$$

$$\tilde{R}^{(1)}(x) = \tilde{S}^{(0)}(x) = x^k S^{(0)}(x) = x^k R^{(1)}(x),$$

$$\begin{aligned} \tilde{S}^{(1)}(x) &= \tilde{R}^{(0)}(x) - \tilde{Q}^{(1)}(x)\tilde{S}^{(0)}(x) \\ &= x^k R^{(0)}(x) - Q^{(1)}(x) x^k S^{(0)}(x) \\ &= x^k S^{(1)}(x). \end{aligned}$$

Using the above three relationships, it is then easy to show that the following holds after the 2nd iteration.

$$\tilde{Q}^{(2)}(x) = Q^{(2)}(x); \quad \tilde{R}^{(2)}(x) = x^k R^{(2)}(x); \quad \tilde{S}^{(2)}(x) = x^k S^{(2)}(x).$$

If similar relationships hold after $i-1$ iterations, by induction then they also do after i iterations, i.e.,

$$\tilde{Q}^{(i)}(x) = Q^{(i)}(x); \quad \tilde{R}^{(i)}(x) = x^k R^{(i)}(x); \quad \tilde{S}^{(i)}(x) = x^k S^{(i)}(x). \tag{4.17}$$

Thus, the two sequences $\mathcal{Q}(\mathcal{T})$ and $\mathcal{Q}(\tilde{\mathcal{T}})$ are equal up to a length of $\min\{e, \tilde{e}\}$. To show that these sequences have the same length, we need to show that $e = \tilde{e}$, i.e. Lemma 1.

Note that e is the smallest integer such that $\deg R^{(e)}(x) \leq m-1$. Similarly, $\tilde{e}$ is the smallest integer such that $\deg \tilde{R}^{(\tilde{e})}(x) \leq m-1+k$. Without loss of generality assume that $e \leq \tilde{e}$. From (4.17), it follows that $\deg \tilde{R}^{(e)}(x) = k + \deg R^{(e)}(x) \leq m-1+k$. Thus $e = \tilde{e}$ and the proof is complete.

Now we consider the effect of replacing $\mathcal{T}$ by $\tilde{\mathcal{T}}$ on $U(x)$. The latter is the polynomial through which the required result is returned at the termination of Algorithm 17. Let $U(x)$ and $V(x)$ are initialized with $U^{(0)}(x)$ and $V^{(0)}(x)$. Then, after i iterations, $1 \leq i \leq e$, polynomial $U(x)$ can be written as

$$U^{(i)}(x) = \mathcal{G}_i\left(U^{(0)}(x), V^{(0)}(x), \mathcal{T}\right)$$

for some function $\mathcal{G}_i$ which depends on $U^{(0)}$(x), $V^{(0)}(x)$, and the first i quotients of the sequence $\mathcal{Q}(\mathcal{T})$. If the triplet is changed to $\tilde{\mathcal{T}}$, it follows from Lemmas 1 and 2 that neither the total number of iterations nor the resulting quotient sequence changes. Hence, for $1 \leq i \leq e$, we have

$$\mathcal{G}_i\left(U^{(0)}(x), V^{(0)}(x), \mathcal{T}\right) = \mathcal{G}_i\left(U^{(0)}(x), V^{(0)}(x), \tilde{\mathcal{T}}\right).$$

In other words, for given $U^{(0)}(x)$ and $V^{(0)}(x)$, the results returned by Algorithm 17 using $\mathcal{T}$ and $\tilde{\mathcal{T}}$ are the same. Then, using $k = -(m-1)$ in $\tilde{\mathcal{T}}$ we can restate Algorithm 17 as follows. This particular choice of k makes the comparison value of the exit condition equal to that of Algorithm 16. This feature allows us to map this new algorithm on a structure that is designed for Algorithm 16 by simply changing the initializations of $R(x)$ and $S(x)$.

Algorithm 18 Shifted Triangular Basis Inversion Algorithm (STBIA).

input: $H(x) = \sum_{i=0}^{2m-2} h_i x^i$

output: The inverse element.

Initialization:

$U(x) \leftarrow 0;\ V(x) \leftarrow 1;\ R(x) \leftarrow x^m;\ S(x) \leftarrow x^{m-1}H(x^{-1});$

Polynomial Updating:

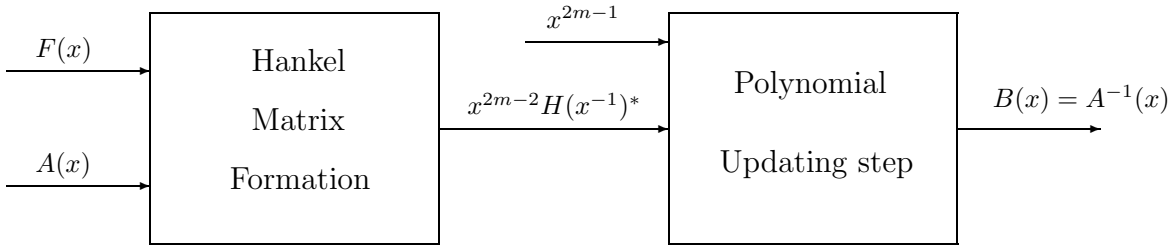
while (deg$R(x) > 0$) **do**

$Q(x) \leftarrow \left\lfloor \frac{R(x)}{S(x)} \right\rfloor;$

$R(x) \leftarrow R(x) - Q(x)S(x); \quad U(x) \leftarrow U(x) - Q(x)V(x);$

swap $(R(x),\ S(x)); \quad$ **swap** $(U(x),\ V(x));$

return $U(x)$



* Hankel vector of length $(2m-1)$

Figure 4.1: Two step block diagram of Algorithm TBIA.

4.5.2 Comments

- In any practical implementation of Algorithms 16 and 18 the computation of $Q(x)$ is done by a bitwise polynomial long-division of $R(x)$ by $S(x)$ where nonzero terms of the quotient are implicitly calculated bitwise.
- Algorithms 16 and 18 are quite similar. Despite the differences of the input polynomials in the initialization step, the polynomial updating step of both algorithms are the same.
- Algorithm 18 can be considered as a two-step process represented in block diagram in Figure 4.1. The Hankel matrix entry formation for the initialization step must precede, pipelined or otherwise, the polynomial updating step. It is shown that the computational complexity of the first step is much less than the second in all specific (practical) cases.

4.5.3 Algorithms for Division

It is mainly the finite field inversion that we have discussed so far. Here we will consider its generalized operation, i.e., finite field division $B = C/A$, where $A, B, C \in \mathrm{GF}(2^m)$, $A \neq 0$. First we briefly present the following well known method for division using polynomial basis. This will eventually help us with division using triangular basis.

Polynomial basis division: Given polynomial basis representations of A and C, it is well known that Algorithm 16 can be modified to obtain the required B in the polynomial basis [11]. To this end, all that need to be changed in Algorithm 16 are to replace "$V(x) \leftarrow 1$" by "$V(x) \leftarrow C(x)$" and "$U(x) \leftarrow U(x) - Q(x)V(x)$" by "$U(x) \leftarrow U(x) - Q(x)V(x) \mod F(x)$". The new initialization of $V(x)$ may cause the degree of $U(x)$ to exceed $m-1$ and hence the modular operation is needed.

Triangular basis division: When A are C are both represented w.r.t. the triangular basis, Equation (4.6) can be extended to

$$\underline{c}_\Lambda = \mathbf{H}(\underline{a}_\Lambda)\underline{b}_\Omega. \tag{4.18}$$

Like Equation (4.6) the above equation can be solved by the scheme proposed in [100] which has a computational complexity twice more than that of Algorithm 18. The scheme of [100] also requires twice as many polynomials as in Algorithm 18. We present a much more efficient solution that uses two separate bases– triangular basis for A and polynomial basis for C.

Let $H'(x) = x^{m-1}H(x^{-1})$. Noting that $R^{(0)} = x^m$, one can see that the following relationship holds at each iteration of Algorithm 18.

$$H'(x)U(x) \equiv R(x) \mod x^m.$$

Let us denote A' as the inverse of A, i.e., $A'(x)A(x) \equiv 1 \mod F(x)$. Assume that Algorithm 17 takes e iterations to return $A'(x)$. Then

$$H'(x)A'(x) \equiv R^{(e)}(x) \mod x^m. \tag{4.19}$$

Now, we proceed to modify Algorithm 18 as we have done with Algorithm 16, i.e., "$V(x) \leftarrow 1$" is replaced by "$V(x) \leftarrow C(x)$", and $U(x) \leftarrow Q(x)V(x)$" by "$U(x) \leftarrow U(x) - Q(x)V(x) \mod F(x)$". Then it is not difficult to see that the following is preserved at each iteration of this modified algorithm.

$$H'(x)U(x) \equiv R(x)C(x) \mod x^m.$$

The above mentioned two changes do not change the updating of $R(x)$ and the new algorithm still terminates after e iterations. As a result,

$$H'(x)U^{(e)} \equiv R^{(e)}(x)C(x) \mod x^m. \tag{4.20}$$

Then equations (4.19) and (4.20) yield

$$\begin{aligned} U^{(e)}(x) &\equiv A'(x)C(x) \mod F(x) \\ &= B(x) \end{aligned}$$

4.6 Summary

In this chapter, inversion and division using double-basis representation by applying EEA to a Hankel matrix has been reviewed. Some results regarding Hankel matrix entry formation have been described. Some key results on inversion and division using triangular basis have been presented. Further, a common set of algorithms to compute inversion and division based on EEA with both polynomial and triangular basis has been discussed.

In the next chapter, systolization as an answer to the VLSI design complexity will be discussed and finally a unidirectional bit serial systolic architecture for computing inversion and division over Galois fields based on a common algorithm will be proposed.

Chapter 5

Systolic Architectures

In this chapter, after introducing some complexity measures in VLSI design, systolic architectures as a solution to VLSI complexity issues are introduced. Next, Brent and Kung systolic array structure for polynomial GCD computation [13] as an example is presented. Finally, a novel unidirectional bit serial systolic architecture for computing inversion and division over Galois fields is proposed. It will be seen that the same architecture can be used for inversion and division in both polynomial and double-basis designs. it will be shown that this unidirectional bit serial systolic implementation with no carry propagation structure is suitable for large values of m. Part of the work in this chapter has been presented in [17, 18, 20].

5.1 Complexity measures in VLSI Design

It is customary to view the complexity of algorithms by defining a *computational complexity* metric and it is measured in order of number of elementary arithmetical operations necessary to complete the algorithm as a function of their binary representation in bits.

An implementation oriented measure of complexity is the *AT-complexity* or Area-Time

product complexity. In general, area is measured as the number of elementary gates and time as the worst case delay time to complete the algorithm. It is easy to see that there is a trade-off between area and time complexity, hence the need for their product as a comprehensive measure of implementation complexity.

Although *AT-complexity* measure is very useful to highlight implementation issues such power and energy consumption, but it cannot cover other important complexity issues such as design flow, IP (Intellectual Property) re-use, reliability (fault tolerance) and neither the detailed area or timing issues derived from interconnect bottleneck of sub-micron VLSI process. For example, on a practical chip, the datapath logic area is just a fraction of total area dominated by power grid, memory bus, interconnect (switches) and on-chip test and fault circuitry [70, 79, 92]. Also the time complexity cannot simply be measured by the number of gates along the logic delay path, but it must be scaled by each gate's fan-out and the effective on-chip routing path between gates.

As the transistor feature of VLSI process is reduced to sub-micron, and multi-million gates designs become feasible, the only answer to the complexity of such designs is to have architectures with regularity, modularity, local communication, massive parallelism and scalability. Concepts such as multi-core, throughput flow pipelining and macro-cell on PLD (Programmable logic Design) become common place. The basic techniques to prevail are the use of regular and repetitive architectural structures. The array processors [70] and systolic architectures are inherently suitable for such capabilities.

5.2 Systolization of Polynomial Updating Step

For large values of m, the complexity of a centralized implementation of the polynomial updating step of Algorithm 16 is dominated by the presence of carry propagation structures (counters or comparators) and long control interconnects. For a scalable VLSI implementation with large values of m and at reasonable clock rates a bit/digit serial systolic architecture is more suitable. To this end, we identify the following four tasks that are performed in each iteration of Algorithm 16.

- Task 1. Compute the remainder of $R(x)$ divided by $S(x)$ to update $R(x)$.

- Task 2. Swap pair of polynomials $R(x),\ S(x)$.
- Task 3. Check for the *exit* condition: $degR(x) > 0$.
- Task 4. Update $U(x),\ V(x)$ according to Tasks 1 and 2 transformation.

Note that Tasks 1, 2 and 3 are exactly the same tasks as defined in Chapter 3.

5.2.1 Task 1 and Shifted Remainder

Among tasks listed above, the most complex one is a *stepwise* computation of remainder of $R(x)$ divided by $S(x)$. To this end, a *stepwise* "long division" operation is described by many authors, *e.g.*, [15, 119]. We will consider a variant of a "long division" operation which returns shifted results

$$x^d(R(x) \mod S(x)) \text{ and } x^d(S(x)),$$

where $d = \deg(R(x)) - \deg(S(x))$. This variant of "long division" operation is shown as Algorithm 19.

In Algorithm 19, initially, the degree of dividend is greater than the degree of divisor by d units. For d iterations a divisor polynomial is left shifted, *i.e.*, multiplied by x. At a first alignment where the degree of shifted divisor has reached the degree of dividend, a remainder polynomial is computed. This remainder may have a degree which is not necessarily less than the degree of the pre-shifted divisor, but its degree will be always less than that of the shifted divisor. Next, a "virtual" restoring of divisor starts. That is the degree of the shifted divisor is not decreased, but rather in the next d iterations, the remainder or partial remainders are left shifted where none, one or more other alignments may occur and new partial remainders may be computed. After exactly d iterations, the pre-shifted divisor is "virtually" restored, and in all these d iterations the remainder polynomial is continuously updated to a degree less than the shifted divisor.

The following facts always hold for Algorithm 19 and they are stated without proof since they are straightforward.

Algorithm 19 Computing $R(x) \pmod{S(x)}$ returning shifted results.

input: $R(x)$ and $S(x)$ polynomials with $\deg R(x) \geq \deg S(x)$.
output: $x^d(R(x) \pmod{S(x)})$, $x^d S(x)$,
where $d = \deg R(x) - \deg S(x)$.

```
δ ← 0;
while (degR(x) > degS(x)) do
    S(x) ← xS(x);
    δ ← δ + 1;
δ ← −δ;          // first alignment and δ = −d //
R(x) ← (R(x) − S(x));
while (δ < 0) do
    R(x) ← xR(x);
    if (degR(x) = degS(x)) then
        R(x) ← (R(x) − S(x));
    δ ← δ + 1;
return R(x), S(x)
  //degree of returned S(x) is equal to degree of input R(x)//
```

Fact 1 *In each iteration within the* **while** *loops of Algorithm 19 the number of shift operations (i.e., $S(x)$ or $R(x)$ is multiplied by x) is one.*

Fact 2 *Algorithm 19 requires $2d$ iterations to complete where d is the difference of the degrees of dividend and divisor. Specifically, d iterations are required for its divisor to be aligned with its dividend and d more iterations are needed to "virtually" restore the divisor. The algorithm starts and ends with $\delta = 0$ and the maximum (absolute) value that variable δ can take is d.*

The sign of variable δ will be used in Section 5.2.3 to propose an efficient division algorithm.

Algorithm 20-a Swapped shifted long division algorithm.

input: $R(x)$ and $S(x)$ polynomials with deg$R(x) \geq$ deg$S(x)$.
output: $x^d S(x), x^d(R(x) \pmod{S(x)})$,
where $d =$ deg$R(x) -$ deg$S(x)$.

```
δ ← 0;
for (d times) do
    S(x) ← xS(x);   // (degR(x) > degS(x)) and δ ≥ 0 //
    δ ← δ + 1;
δ ← −δ;             // first alignment and δ = −d //
[R(x), S(x)] ← [S(x), (R(x) − S(x))];
for (d times) do
    S(x) ← xS(x);   // (degR(x) > degS(x)) and δ < 0 //
    if (degR(x) = degS(x)) then
        S(x) ← (R(x) − S(x));
    δ ← δ + 1;
return R(x), S(x)
```

5.2.2 Task 2 and Swapped Shifted Long Division Algorithm

Shown below is Algorithm 20-a which is a slight variation of Algorithm 19. The purpose of Algorithm 20-a is not only to update $R(x)$ but also to swap $R(x)$ and $S(x)$ as needed in the second last statement of Algorithm 16.

From computational view point, there are a number of differences between Algorithms 19 and 20-a. In Algorithm 20-a, whenever $\deg(R(x)) = \deg(S(x))$, then $S(x) \leftarrow (R(x) - S(x))$, and at the first occurrence of $\deg(R(x)) = \deg(S(x))$, $R(x)$ is updated as $R(x) \leftarrow S(x)$. Based on Fact 2, the **while** loops of Algorithm 19, have been replaced with **for** loops in Algorithm 20-a. The **for** loops have made the use of variable δ redundant. Nevertheless, δ is still shown here, since it will be used later. We now define the following two terms which will be useful for describing our hardware architectures later on.

- *Alignment*: Whenever deg$R(x) =$ deg$S(x)$, we call this an alignment of $R(x)$ and $S(x)$.
- *Swap with partial update* (SPU): From an initial condition of deg$R(x) >$ deg$S(x)$, let

$S(x)$ be repeatedly updated as $S(x) \leftarrow xS(x)$ to reach an *alignment*. Then, an SPU corresponds to the following:

$$[R(x),\ S(x)] \leftarrow [S(x),\ x(R(x) - S(x))].$$

There are $2d$ shift operations in Algorithm 20-a. For hardware implementation of Algorithm 20-a, it is desirable to perform one shift in each clock cycle which will result in balancing the maximum critical path of each cycle. To this end, Algorithm 20-a can be restated as in Algorithm 20-b.

The final **if** statement in Algorithm 20-b ensures that the degree of returned $S(x)$ is less than the degree of $R(x)$. When the returned polynomials of Algorithm 20-b are to be used as inputs of another round of Algorithm 20-b, and this process is to be repeated, then this **if** statement can be omitted from all but the final round of Algorithm 20-b. Moreover, if the final result is only $R(x)$ which we are interested in, then this **if** statement can be removed even from the final round of Algorithm 20-b as well.

Fact 3 *For Algorithm 20-b, let $R(x)$ be a polynomial of degree m and $S(x) \neq 0$, then the degree of returned $R(x)$ is also m.*

5.2.3 Tasks 3, 4 and Putting All Together

In this section all remaining tasks are discussed and a new variant of Algorithm 16 to compute finite field division is presented.

Task 3

For Algorithm 16, assume that there are n iterations in the **while** loop before its *exit* condition, *i.e.*, Task 3, is met. Also, let us assume that Algorithm 20-b is used for updating of $R(x)$ of Algorithm 16. Thus in Algorithm 16, Algorithm 20-b runs a total of n times.

Algorithm 20-b Enhanced variant of Algorithm 20-a with an SPU.

input: $R(x)$ and $S(x)$ polynomials with deg$R(x) \geq$ deg$S(x)$.

output: $x^d S(x), x^d(R(x) \pmod{S(x)})$,
where $d =$ deg$R(x) -$ deg$S(x)$.

$\delta \leftarrow 0$;
for ($2d$ times) **do**
  if (deg$R(x) >$ deg$S(x)$) & ($\delta \geq 0$) **then**
    // occurs d times //
    $S(x) \leftarrow xS(x)$;
    $\delta \leftarrow \delta + 1$;
  else if (deg$R(x) =$ deg$S(x)$) & ($\delta \geq 0$) **then**
    // occurs only once //
    $\delta \leftarrow -\delta$; // first alignment and $\delta = -d$ //
    $[R(x),\ S(x)] \leftarrow [S(x),\ x(R(x) - S(x))]$; // SPU//
    $\delta \leftarrow \delta + 1$;
  else if ((deg$R(x) >$ deg$S(x)$) & ($\delta < 0$)) **then**
    $S(x) \leftarrow xS(x)$;
    $\delta \leftarrow \delta + 1$;
  else if ((deg$R(x) =$ deg$S(x)$) & ($\delta < 0$)) **then**
    $S(x) \leftarrow x(R(x) - S(x))$;
    $\delta \leftarrow \delta + 1$;
if (deg$R(x) =$ deg$S(x)$) **then**
  $S(x) \leftarrow (R(x) - S(x))$;
return $R(x),\ S(x)$

For the ith run of Algorithm 20-b, let us denote the value of its parameter d by d_i. Since inputs of Algorithm 16 have $\deg(F(x)) = m > \deg(A(x))$, we can write

$$\sum_{i=1}^{n} d_i = m. \tag{5.1}$$

Based on (5.1) and noting that the ith round of Algorithm 20-b has a **for** loop of $2d_i$ iterations, we can effectively replace the **while** loop of Algorithm 16 by a **for** loop of $2m$ iterations. This is shown in Algorithm 21. When compared with Algorithm 20-b, one can easily see that Algorithm 21 has additional statements. These are explained later on.

Note that based on Fact 3, after the completion of the **for** loop of Algorithm 21, degree of $R(x)$ is m, *i.e.*, m times shifted version of what we get from Algorithm 16. Let s_i denote the ith coefficient of $S(x)$. Then the condition $(\deg(R(x)) = \deg(S(x)))$ corresponds to $(s_m = 1)$. Thus all occurrences of $(\deg(R(x)) = \deg(S(x)))$ and $(\deg(R(x)) > \deg(S(x)))$ have been replaced by $s_m = 1$ and $s_m = 0$, respectively. A design which has no carry propagation structure in order to to update δ of Algorithm 21 will be discussed in Section 5.3.

Task 4

As seen above, when Algorithm 20-b is applied to Algorithm 16, polynomial $R(x)$ is updated without explicitly computing polynomial $Q(x) = \left\lfloor \frac{R(x)}{S(x)} \right\rfloor$. The latter appears to be needed to update polynomial $U(x)$. The computation of $Q(x)$ can be however avoided if $U(x)$ is updated (*i.e.*, transformed) in the same way we update $R(x)$ using Algorithm 20-b. This is used and shown in Algorithm 21. Like $R(x)$, the final $U(x)$ would be an m times shifted version of what we get from Algorithm 16. We call this the shifted result problem. To solve this problem the returned $U(x)$ must be restored by dividing it by x^m.

Different strategies to perform $x^m U(x)/x^m$, have been proposed: initializing $U(x) = x^{-m}$ [56]; initializing $U(x) = x^m$, $V(x) = 0$, but at each iteration divide $V(x)$ instead of multiply such that $U(x) \leftarrow U(x) - V(x)/x^d$ ([11], Section 2.3), [119]; multiplying or dividing $V(x)$ alternatively [15]; or using an auxiliary polynomial [37]. The first method requires pre-computation for each different irreducible polynomial. The second and third proposals compute only inverses. The fourth method requires an auxiliary polynomial.

Algorithm 21 Division Variant of Algorithm 16 with Restoring Result Polynomial.

input: irreducible polynomial $F(x)$, divisor $A(x)$ and
dividend $C(x)$ in polynomial basis.

output: $B(x) = C(x)/A(x) \mod F(x)$.

Initialization:

$\delta \leftarrow 0$; $U(x) \leftarrow 0$; $V(x) \leftarrow C(x)$;

$R(x) \leftarrow F(x)$ $S(x) \leftarrow A(x)$;

Polynomial Updating:

for $2m$ times **do**

if ($s_m = 0$ & $\delta \geq 0$) **then** { // shift and delay //}

$[R(x), S(x)] \leftarrow [R(x),\ xS(x)]$;

$[U(x), V(x)] \leftarrow [U(x)/x,\ V(x)]$;

else if ($s_m = 1$ & $\delta \geq 0$) **then** {// SPU //}

$\delta \leftarrow -\delta$;

$[R(x), S(x)] \leftarrow [S(x), x(R(x) - S(x))]$;

$[U(x), V(x)] \leftarrow [V(x), x(U(x) - V(x)) \mod F(x)]$;

else if ($s_m = 0$ & $\delta < 0$) **then** {// shift and shift //}

$[R(x), S(x)] \leftarrow [R(x),\ xS(x)]$;

$[U(x), V(x)] \leftarrow [U(x),\ xV(x) \mod F(x)]$;

else if ($s_m = 1$ & $\delta < 0$) **then** {// alignment (no **swap**) //}

$[R(x), S(x)] \leftarrow [R(x), x(R(x) - S(x))]$;

$[U(x), V(x)] \leftarrow [U(x), x(U(x) - V(x)) \mod F(x)]$;

$\delta \leftarrow \delta + 1$;

return $U(x)$

Here, we take a new and simple approach which appears to be a more efficient scheme when implemented in a bit serial systolic structure. We allow $U(x)$ to shift and expand in both directions, *i.e.*, left shifted (multiply by x) and right shifted (divide by x). In Algorithm 21, in all **if** case statements except the first one, pair $[U(x),\ V(x)]$ is updated as pair $[R(x),\ S(x)]$. For the first **if** case statement which occurs exactly m times, pair $[U(x),\ V(x)]$ is updated by one less shift with respect to pair $[R(x),\ S(x)]$. That is, instead of $[U(x),\ xV(x)]$ we use $[U(x)/x,\ V(x)]$. For the first **if** case statement, while $S(x)$ is shifted and $R(x)$ is unchanged, for their counterparts, $V(x)$ is not shifted, and more importantly $U(x)$ is delayed (mathematically equivalent to divided by x, but in a bit serial architecture it is sufficient that its progress is delayed by one cycle, hence, no modulo reduction is needed). Thus, in all four cases, Algorithm 21 conserves the same difference of degree relationship between two polynomials of each pair. By the time, $R(x)$ becomes x^m (*i.e.*, non-restored and m-fold shifted version of 1), $U(x)$ becomes finite field division result but in restored form.

For field division, $V(x)$ is initialized with a polynomial which may have a maximum degree $m-1$. In this case, at any iteration when $V(x)$ is left shifted, a modulo reduction may be needed. On the other hand, for an inversion computation, $U(x)$ and $V(x)$ are initialized with 0 and 1, respectively. We know that, in Algorithm 21, $V(x)$ is left shifted exactly m times. Hence, $\deg(V(x))$ may reach m only at $2m$th iteration, but this may be ignored since at this iteration the final result, *i.e.*, $U(x)$, is already computed. Thus, for an inversion only architecture, no modulo reduction is required.

An example of an inversion as a special case of division to highlight the stepwise restoring action of Algorithm 21 will be shown next.

5.2.4 An Example of Stepwise Restoring Action of Algorithm 21

In Table 5.1, the inverse of element $A(x) = 1 + x$ in GF(2^3) defined by $F(x) = 1 + x + x^3$ is $x + x^2$ which is returned in $U(x)$ at $i = 6$ if a restoring is applied. Otherwise, its shifted version by exactly m position is computed. In this example, the overwriting cases are recognized at $i = 2$ and 5.

In the non-restoring case, $[U(x),\ V(x)]$ transformations follow that of $[R(x),\ S(x)]$. In

Table 5.1: Example of inversion over GF(2^3).

				non-restoring		*restoring*	
i	δ	$R(x)$	$S(x)$	$U(x)$	$V(x)$	$U(x)$	$V(x)$
0	0	$1+x+x^3$	$1+x$	0	1	0	1
1	1	$1+x+x^3$	$x+x^2$	0	x	0	1
2	2	$1+x+x^3$	x^2+x^3	0	x^2	0	1
3	-1	x^2+x^3	$x+x^2+x^3$	x^2	x^3	$1=x^0$	x
4	0	x^2+x^3	x^2	x^2	x^3+x^4	x^0	$x+x^2$
5	1	x^2+x^3	x^3	x^2	x^4+x^5	x^{-1}	$x+x^2$
6	0	x^3	x^3	x^4+x^5	$x^3+x^5+x^6$	$x+x^2$	$x^0+x^2+x^3$

the restoring one, at *overwriting* iteration and while $\delta < 0$, $V(x)$ is shifted similar to $S(x)$. Otherwise, $U(x)$ is divided by x while $V(x)$ is not shifted. Thus the relative degree of pair $[U(x),\ V(x)]$ follows that of pair $[R(x),\ S(x)]$ always. As counter δ increases and decreases exactly m times, hence, after $2m$ iterations, the pair $[U(x),\ V(x)]$ is divided by exactly x^m.

It must be re-emphasized that in a fixed size register base design, it is necessary that $U(x)/x$ is modulo reduced. If not, a nonzero shifted least significant coefficient of $U(x)$ may be lost. Hence a restoring implementation with a fixed size register would be as inefficient as a non-restoring. However, in a bit serial systolic architecture with a double delay element on the path of $U(x)$, no such modulo reduction is needed. In other words, double delay elements provide a computationally equivalent register whose size is as large as $2m$. Hence, keeping track of $U(x)/x$ with a negative sign and its restoring afterward, as shown in row 5 of Table 5.1 is feasible.

The example in Table 5.1 is simple but rather a special case where the correct result is written in $U(x)$ exactly at $i = 2m$ iteration, since $\delta = 1$ at $i = 5$. In general this is not the case, and the last SPU case may occur at $i < 2m$, with $\delta > 1$. Afterward, the sign of δ is negated, and till it reaches zero no further SPU occurs. Hence, Algorithm 21 does

not modify $U(x)$ anymore by branching to the last **else if** case repeatedly. This ensures that the correct result is always returned after $2m$ iterations. During these last iterations between the last SPU and until $i = m$, degree of $V(x)$ may reach m. However, since $V(x)$ is not used any more, a modulo reduction is not needed.

5.3 Bit Serial Unidirectional Systolic Architectures

In this section, after presenting a generalized bit serial systolic structure, different PE architectures for inversion or division are discussed.

5.3.1 Bit Serial Unidirectional Systolic Structure

In Figure 5.1, a generalized bit serial unidirectional systolic architecture for inversion and/or division over polynomial basis is shown. It is based on Algorithm 21. In all descriptions of this section we refer to this maximum $2m$ identical PE model. However, in the next section optimizations are described which result in fewer number of PEs. In Figure 1, no external latches are shown since all leading coefficients of sequences enter and exit each PE in the same cycle, and no such latches are needed.

In Figure 5.1, there are two sets of inputs to each PE: the *datapath* inputs consisting of $r,\ s,\ u,\ v,\ f$; the *control* inputs consisting of $dseq,\ dec,\ update$ and $start$. For division using polynomial basis (*i.e.*, Algorithm 16), both r and f inputs to PE_0 consist of the coefficients of $F(x)$. For inversion only architecture the input f is not needed and the input r is as described before. The highest order coefficients, degree m of polynomials enter first. In a bit serial systolic structure, each PE must be initialized once per each computation round.

All *control* signals, other than $start$, will be introduced in the following sections. The $start$ signal is used to set up other *control* signals in the initialization cycle. The $start$ is defined as a one followed by m zeroes. It takes 2 cycles for the $start$ signal to pass through each PE and after $4m$ cycles, the coefficient of the highest degree, *i.e.*, $m-1$, of the result synchronized with ($start = 1$) exits the last PE; $m-1$ more cycles and all the coefficients

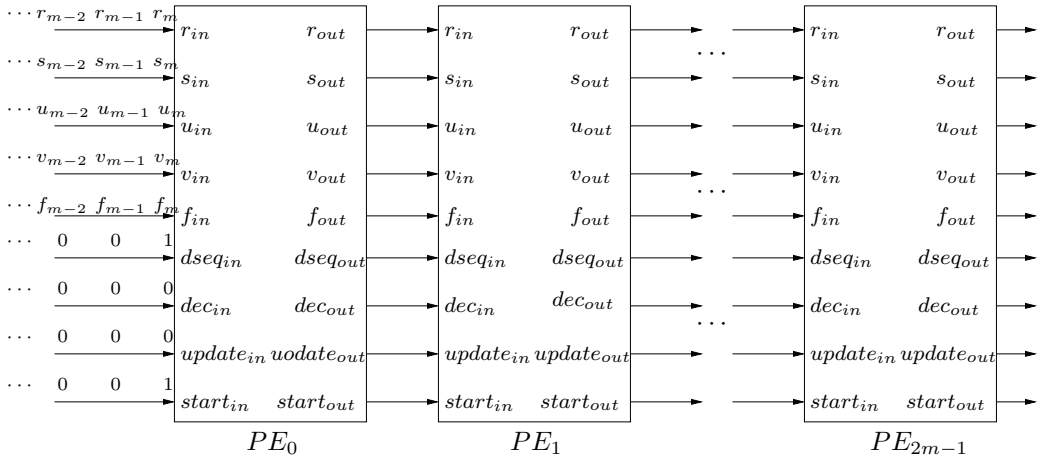


Figure 5.1: Bit serial unidirectional systolic architecture for Inversion/Division.

of the result are available. Thus, in an implementation with $2m$ PEs the latency is $5m-1$, and in an optimized implementation with $2m-1$ PE, it would be $5m-2$.

The throughput of this bit serial architecture, in either case of polynomial basis inversion or division where $R(x) = F(x)$, is one computation (inversion or division) per $m+1$ cycles since the longest input sequence is f which is $m+1$ bits. Hence, in a back to back computation the distance between two consecutive $start = 1$ signals must be $m+1$.

5.3.2 Processing Element for Inversion

Figure 5.2 represents the single type PE based on Algorithm 21. Two of the *control* signals, *update* and *dec* are precomputed and saved for the next PE. The *update* is the latched value of incoming $s_{in} = s_m$ at ($start = 1$). The pre-computation of *update* serves to reduce the delay of the datapath. The sign of δ as defined before must be computed separately, and is called *dec* as described next.

The PE of Figure 5.2 incorporates a serial-in implementation of a ring-counter structure with no carry propagation delay to compute the difference of degree of polynomials is proposed. A mathematical formulation similar to [115] but simpler follows. The signed

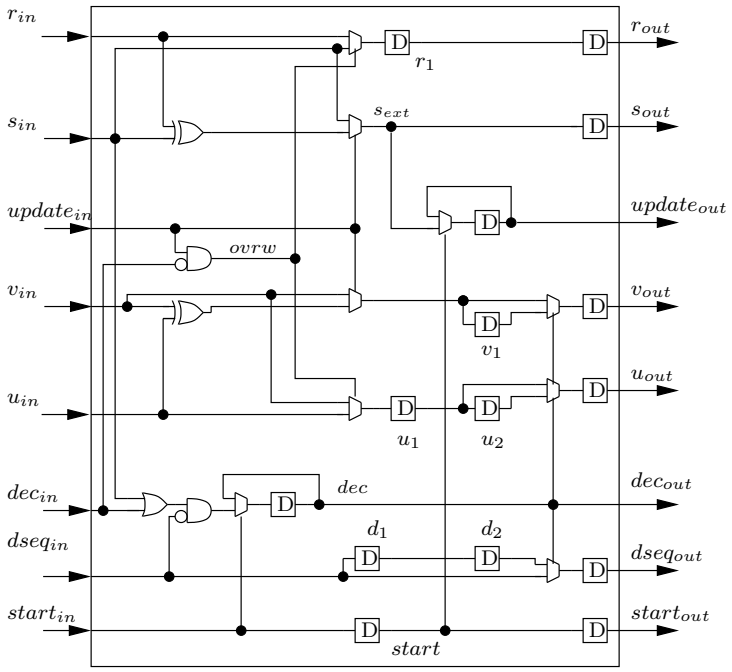


Figure 5.2: Processing element for the inverter where D is a delay element.

integer δ may be represented by a sign bit, *dec*, and by a $(m+1)$-bit sequence *dseq* where $dseq = 2^{|\delta|}$ and

$$dec = \begin{cases} 0 & \delta \geq 0, \\ 1 & \text{otherwise.} \end{cases}$$

At *initialization* $\delta = 0$, hence, $dec = 0$, and $dseq = 00 \cdots 001$. At each shift of $S(x)$ while $dec = 0$, *dseq* is left shifted. At SPU iteration, *dec* is set, and *dseq* is right shifted. In all consecutive iterations until *dseq* returns back to 1, *dec* remains set. It will reset at $dseq = 1$. Thus, *dec* can be computed at $start = 1$ as

$$dec \leftarrow \overline{dseq}_{in} \ \& \ (dec_{in} \mid s_{in})$$

where the overline, & and | represent a NOT, AND and OR logic operators.

It is said that *dec* is needed to check for an SPU condition. An internal *state* signal $ovrw \leftarrow (update_{in} \ \& \ \overline{dec}_{in})$ is computed accordingly.

In Figure 5.2, a single delay element, r_1 causes the xS action. Double delay elements u_1, u_2 synchronized with v_1, while *dec* is set, provide the restoring action as described. The maximum gate delay of Figure 5.2 is

$$t_g = max(2T_{A2}, T_{X2} + T_{M2}) + T_{M2},$$

where T_{A2}, T_{X2} and T_{M2} represent the delay of a 2-input AND gate or OR gate, a 2-input XOR gate and a 2-input MUX respectively.

This can be further shortened at the expense of two extra latches (on the path of *dec* and *ovrw* and the propagation of the latched *ovrw* signal. In this case, the critical delay path will be as short as $t_g = T_{A2} + T_{M2} + T_D$. However, in general, such a reduction may not be a substantial improvement which justifies the trade-off. The critical delay path of an architecture required to compute the effective clock period must take account not only the gate delay shown above but also the delay of the latches, interconnect and the clock network skew as well. In a standard synchronous design, the *setup* time of flip-flops and the clock skew may be much larger than single gate delay and may dominate the effective clock period especially where such reduced logic paths exist. In the following we only consider the gate delays which are architecture dependent and not the critical path delay.

5.3.3 Processing Element for Division

In Algorithm 21, in order to perform a division, V must be initialized with the dividend. Next, at each iteration where a new value of V is computed, specifically at an SPU iteration and while $\delta < 0$, a modulo reduction of this polynomial may be needed. In Algorithm 21, $U \leftarrow U/x$ is used only for a pre-shifted restoring and never requires a modulo reduction. Up to reaching an SPU condition its value is zero, and always the number of right shifts (divide by x) of U is compensated by number of multiplication of V at SPU iteration or while $\delta < 0$. The *reduction* condition for the division algorithm is detected when the leading coefficient of V, *dec* and *start* are all set.

Figure 5.3 shows as augmented version of Figure 5.2 with added circuitry to check the *reduction* condition, and to perform modulo reduction accordingly. However, in order to reduce the gate delay path of the *reduction* condition checking, a balanced delay path for

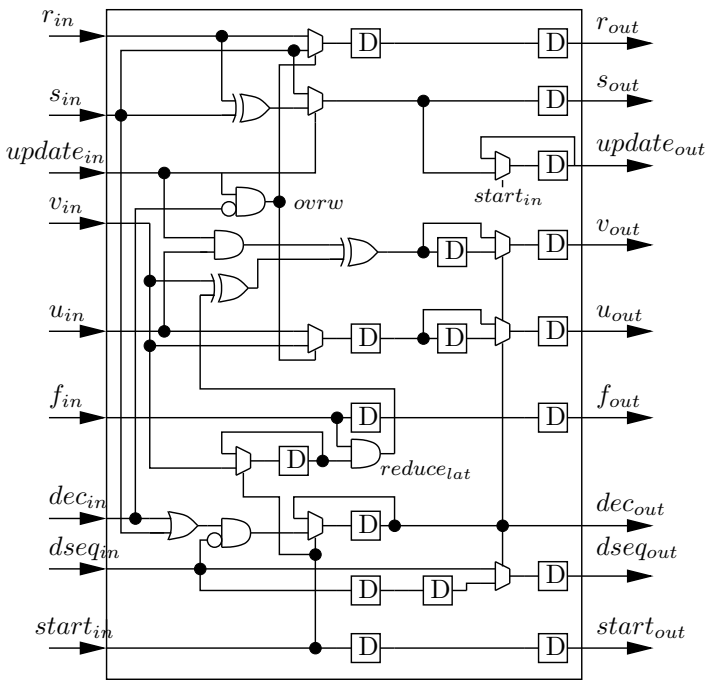


Figure 5.3: Processing element for the divider.

the input of three latches u_1, v_1, and $reduce_{lat}$ is computed, assuming similar delays for AND and XOR gates, as shown in Figure 5.3. Accordingly, the maximum gate delay is

$$max\ (3T_{A2},\ 2T_{A2} + T_{X2},\ T_{A2} + 2T_{X2}) + 2T_{M2},$$

where $T_{A2},\ T_{M2}$ and T_{X2} were defined before.

5.3.4 Bit Serial Inverter-Divider in Triangular Basis

The PE architectures in Figure 5.2 and 5.3 can be used to perform inversion and division with a divisor in triangular basis without any modification or extra control signals. The input polynomials $R(x),\ S(x)$ should be initialized as in Algorithm 18.

Because of negative degree terms of $x^{m-1}H(x^{-1})$, this input sequence is almost twice as long as their polynomial basis counterpart. This fact does not change the latency but

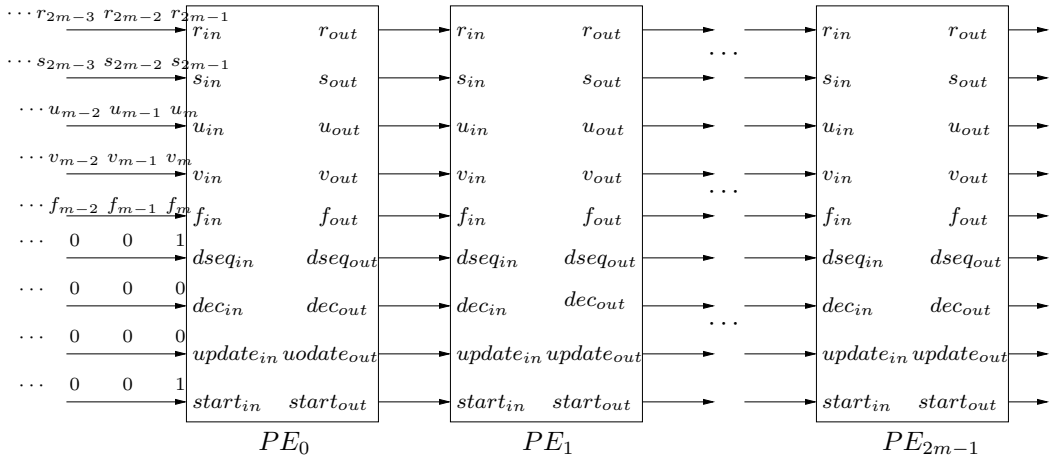


Figure 5.4: Bit serial systolic architecture for Inversion/Division in Double-basis.

rather the throughput of this architecture for inversion or division using triangular basis which is almost half of polynomial basis. All other inputs to both polynomial and triangular inverter or divider will be the same.

Let consider a bit serial systolic architecture as in Figure 5.4. In fact, this is the same as Figure 5.1 with the exception of its *datapath* input set.

In Figure 5.4, the subscripts follow those of Algorithm 17 to better highlight the highest order of coefficients needed in a triangular basis architecture versus a polynomial basis. However, from a practical point of view and in order to use the same set of control signals as in the case of polynomial basis architecture, then more conveniently, the *datapath* input set may follow those defined in the initialization step of Algorithm 18. Specifically, $r,\ s,\ u,\ v$ can be coefficients of x^m, $x^{m-1}H(x^{-1})$, 0 and 1 respectively.

One concludes that the same PE architectures in Figure 5.2 and 5.3 can be used to perform inversion and division with a divisor in triangular basis and this returns an inversion or division result in polynomial basis without any modification or extra control signals.

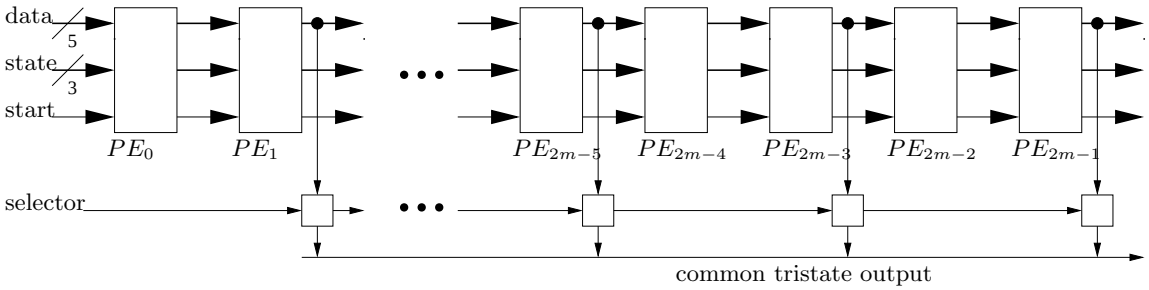


Figure 5.5: Variable dimension divider using selectors over common tristate output.

5.4 Generalization and Optimization

The unidirectional bit serial systolic architecture described in Figure 5.1 not only is independent of the defining irreducible polynomial of the field but also it can compute the inverses in Galois fields of any dimension. Two different aspects of this assertion are discussed next.

5.4.1 Universal Bit Serial Systolic Inverter-Divider

We may generalize the *exit* condition as defined in Section 5.2, where it is mapped to an exact number of PEs. Let N represents the number of PEs in the architecture. In a bit serial unidirectional systolic architecture as in Figure 5.1, it is easy to see that if $N \geq 2m$, where m represents the field dimension, the coefficients of the inverse element start to appear at the output of the $2m$th PE after $4m$ cycles. Hence, it is possible to process any field dimension $m \leq N/2$ as far as the correct result can be captured at the right cycle. Here two such schemes are described.

Either the output of the $2m$th PE should be accessible directly, or a mechanism should be in place to de-activate all remaining PEs from $2m+1$ to N, and to capture the result at the output of Nth PE always. The former does not require any inner PE modification and provides the optimal latency. One such scheme using a selector circuitry over a common tristate output line is shown in Figure 5.5.

One of the selector latches can be set such that only the specific output corresponding

to this latch is passed to the common output, and all other outputs are kept in tristate. Figure 5.5 represents a serial-in mechanism to set the latch corresponding to the field dimension. A parallel load is possible as well, since the serial-in *selector* must not be used synchronized with *start*. In fact, the given latch is set only once per each change of field dimension. Note that only the output of every other PE requires a selector latch.

5.4.2 Trading off Throughput for Storage Area

If some external buffering is used, it is easy to see that any bit serial systolic architecture with $N < m$ where N represents the number of PEs, can be used to perform inversion/division over fields of variable dimension m. In this case, at the expense of throughput, the same PEs are initialized more than once for each inversion computation to process different blocks of data. A limiting case is of particular interest where no external buffering is needed.

In Section 5.2, it is assumed that $2m$ PEs are required to perform the inversion but this is a case where a minimum number of $(m + 1)$ cycles per inversion is a design requirement. On the other hand, if this constraint is relaxed, and no back to back inversion is required, it is possible to use exactly m PEs to perform an inversion without any external buffering. This is achievable through a MUXing mechanism at the input of the first PE. At the $2m$th cycle, by the time the $(start = 1)$ signal exits the mth PE, all input coefficients have already entered the first PE. In the next cycle, *i.e.*, $2m + 1$, it is possible to feedback the outputs of the mth PE synchronized with $(start = 1)$ into the first PE which will be re-initialized for a second time. In this case, throughput is sacrificed, and the input of a new set of input should be delayed by $2m$ cycles.

5.4.3 Area and Latency Optimization without Throughput Loss

A slight optimization in area and latency of the architecture in Figure 5.1 is achievable without degrading throughput. In Sections 5.2, 5.3.1, it is always assumed that the highest order coefficient of polynomials to be processed are the mth for the polynomial. In fact,

among all data inputs of Algorithm 21 only $R(x)$ has a nonzero coefficient of this degree. As a result the inputs to the first PE of the proposed architecture is always the same, most importantly, $s_{in} = s_m = 0$, $dseq_{in} = 1$, and according to Algorithm 21, the first PE always branches to same **else if** ($s_m = 0$ & $d \geq 0$). Hence, it is easy to compute the output of PE_0, and initialize the algorithm with inputs of PE_1 directly. Specifically, a new architecture with only $2m - 1$ PEs with the following inputs to its new first PE can be devised: $start = 1$, $update = 1$, $dec = 0$, $dseq = 00 \cdots 010$, f_m, and data inputs r_m, s_{m-1}, v_{m-1}, u_{m-1}. Hence, r, u, f, $start$, and dec are not changed (u is initially always zero). Also, s_m and v_m are always zero.

Thus, only two *control* signals $dseq$ and $update$ will have modified values. In this $2m-1$ PE architecture, the latency is reduced to $5m-4$. In the literature, most systolic proposals for inversion/division use such a $(2m - 1)$ PE based structure, we refer to this optimized model for comparison purposes.

5.4.4 Implementation Results

Exhaustive test patterns for values of m up to thirteen for inputs and outputs using a C model have been generated. Different variants of the architecture (inverter and divider, basic or optimized in only polynomial basis or double-basis) have been coded in Verilog-HDL language which has been simulated and verified. The Verilog model has been synthesized using Synopsys with a standard CMOS 0.18μ library for a clock period of 1.1ns for the inverter and 1.4ns for the divider. The average setup time of the flip-flops has been 0.32ns and the clock skew set at 10%. The area reports, including the flip-flops, are equivalent to 90 and 120 2-input NAND gates for each inverter and divider of Figure 5.2, 5.3. It should be pointed out that non-combinational area, as defined by Synopsys, is the dominant part. For the divider, it is more than 75%, and for the inverter more than 80%.

Further implementation results will be discussed in the next section where above complexity measured in the context of a practical application of this bit serial systolic structure as part of a complete elliptic curve processor is discussed.

5.5 Comparison

In this section, the architectures proposed in this work are evaluated against some relevant polynomial basis field inversion and division systolic architectures based on extended Euclidean and binary (Stein's) GCD algorithm [37, 40, 41, 56, 115, 119] which were already mentioned in Chapter 3. In the following, as before, T_{X2}, T_{M2}, T_{A2} represent a 2-input XOR, 2-input MUX and a 2-input AND/OR delay time, respectively.

In [40], Guo and Wang proposed a systolic implementation of a modified Brunner et al. [15] inversion algorithm based on a variant of EEA. They have enhanced that algorithm by eliminating the asymmetric looping due to alternating division or multiplication of the algorithm to solve the shifted result problem. In [41], they proposed an improved version of an in-place parallel-in parallel-out systolic inverter/divider. It is a row of 2 types of cells with bidirectional signals. This proposal has a lower number of cells since no extra steps for degree restoring are required at the expense of using an extra polynomial and an extra control signal. The reported area complexity is said to be $O(m)$ but in fact it is $O(m \log m)$ due to inner cell adder/subtractor. In [37], they further improved the bit serial implementation of the above design by introducing a single cell type unidirectional systolic design. The shifted result problem is again addressed by using an extra polynomial as before. In this design a distributed degree tracking control mechanism is introduced.

In [119], Yan and Sarwate proposed two parallel-in systolic inversion architectures, centralized and distributed control, based on a modified EEA. The distributed control architecture has a bidirectional degree tracking mechanism requiring 2 control signals and a $2m$ bits register but results in only two gates on its critical path. These are inversion only architectures.

Table 5.2 shows a comparison of the divider with PE of Figure 5.3 and the inverter with the PE of Figure 5.2 with two representative designs reviewed above. The design in [37] is chosen since it is a divider with best overall characteristics and also in the same category as ours (bit serial unidirectional systolic using a variant of EEA with a counter which has no carry propagation chain). The second design [40] is chosen to show why single row parallel-in designs in particular when they use counter-like structures with carry propagation chains are not suitable for large values of m.

As it can be seen in Table 5.2, our PEs for division (Figure 5.3) and inversion (Figure

Table 5.2: Comparison of bit serial systolic dividers and inverter.

Features	Guo et al. [40] [a]	Guo et al. [37] [b]	Figure 5.3	Figure 5.2
Time Complexity	$O(m)$	$O(m)$	$O(m)$	$O(m)$
# of cycles per inversion/division	m	$m+1$	$m+1$ [c]	$m+1$
Latency	$8m-1$	$5m-4$	$5m-4$ [d]	$5m-4$
Max Gate Delay	$2T_{A2}+T_{X3}$ $+2T_{M2}$	$T_{A2}+T_{X2}$ $+T_{X3}+T_{M2}$	$2T_{A2}+T_{X2}$ $+T_{M2}$	$2T_{A2}+T_{M2}$
Area Complexity	$O(m\log m)$	$O(m)$	$O(m)$	$O(m)$
Gate Count:	total[e]	per cell	per cell	per cell
IO Latch[f]	$4m$	-	-	-
inner Latch[g]	$46m+$ $4m\lceil\log_2 m+1\rceil$	22	18	15
MUX	$35m+2$	11	10	9
XOR	$11m$	5	3	3
AND/OR	$26m$	8	7	3
others	adder, zero-check	-	-	-
Single Cell	no	yes	yes	yes
I/O pins	$O(m)/O(m)$	10/10	9/9	8/8
Unidirectional	yes	yes	yes	yes
Basis	poly.	poly.	poly./triang.	poly./triang.

[a] single row parallel-in architecture, with adder
[b] distributed control
[c] for polynomial basis input
[d] for a $2m-1$ PEs architecture
[e] different type of cells
[f] IO Synchronization and Reordering
[g] inner and inter cell latches

5.2) have a better area as well as time complexities. In addition, it can be shown that the same structure can be incorporated in a design to compute inversion and division with a divisor in triangular basis.

5.6 Summary

In this chapter, the use of systolic architectures as a solution to the VLSI design complexity has been reviewed. Next, a novel and optimized division algorithm suitable for systolization has been proposed. A unidirectional bit serial systolic architecture for computing polynomial basis division based on this algorithm has been described. It has been said that the same structure can be used for double-basis inversion and division with appropriate input initialization. Finally, after discussing further generalization and optimization aspects some implementation results and comparative tables have been provided.

In the next chapter an application of this bit serial systolic structure to implement a systolic Elliptic Curve (EC) crypto processor will be discussed.

Chapter 6

Systolic Elliptic Curve Processor

6.1 Background

Many algorithms of public-key cryptography are based on finite fields arithmetic, either prime fields GF(p) or extension binary fields GF(2^m), where m represents the dimension of the field. For example, the Diffie-Hellman key exchange protocol [23], Digital Signature Algorithm [86] El-Gamal Cryptosystem [25] and systems which use elliptic [83] and hyperelliptic [65] curves can be implemented using operations in extension binary fields. In contrast to the schemes based on the discrete logarithm problem, *e.g.*, [25] (similar to their prime field counterparts, *e.g.*, RSA [95]) which may require field dimensions as large as 4000 bits to be secure, the elliptic (and hyperelliptic) curve cryptosystems provide a higher security strength [71] having a dimension less than 256 bits. This aspect of the latter group make them specially attractive in applications where the computation, power, storage and communication bandwidth is a prime concern such as embedded System on Chip architectures used in Personal Digital Assistant (PDA) or wireless devices.

On the other hand, the elliptic curve cryptosystems are more cumbersome to implement rather than their RSA-type counterpart, particularly in hardware implementation. The

reason is that not only they require a complete suite of finite field arithmetic operations (multiplication, squaring, and inversion or division) to implement point operations on the curve, but also they have a more complex algorithm to be implemented. In a hardware implementation this means multiple functional units and/or complex and dedicated control units. Moreover, among the field operations, multiplicative inversion (or division) is known to have a very poor performance and to allocate a dedicated hardware unit for it may not be cost effective.

It is known that the effect of the poor performance of the field division can be mitigated at the expense of many more multiplications using projective coordinates to represent the points on the curve. Several variants of point operation using the projective coordinates exist [7], which however require extra storage space and more complex control units than the non-projective (affine) coordinates point representation. In fact, these schemes are best suitable either for software implementations over general purpose processors or embedded systems with core processors where complex control flow can be added and extra storage, *e.g.*, register and/or cache memory is available [45]. Many hardware mapping of these variants, mostly into FPGAs (Field Programmable Gate Array) with large data and instruction memory usage and complex control units, are implemented [8, 27, 72, 73, 90] which achieve order of magnitude better performance than their software counterparts.

In this chapter a very small footprint (area efficient) EC hardware processor will be proposed, whose datapath is based on a combined multiplier divider (CMD) bit serial systolic architecture which computes both multiplication and division over GF(2^m) in a single hardware unit and may achieves high performance by running at extreme clock rates. This combined multiplier divider is based on the architecture described in the previous chapter, and its systolic structure allows using the EC affine coordinates which requires a very small control unit and the least amount of storage space.

Also, this bit serial structure coupled with a shift register file, instead of single or dual port RAM model, allows an efficient bit-level pipelining among successive field multiplication and division operations, or even consecutive EC full point additions when multiple CMD units are used. Such a memory organization can be easily expanded independent of the datapath to provide adequate external buffering for processing field dimensions much larger and independent of the number of PEs present in the datapath. This scalability fea-

ture comes at the expense of reducing the overall throughput. The area-time complexity of such a structure outperforms that of all proposals known to us not at the field operation level but most importantly at the system level when variable field dimension and a random irreducible polynomial usage is a requirement.

On the other hand, it will be shown that for high performance, high throughput applications, multiple CMD units can be pipelined due to their unidirectional bit serial architecture. Furthermore, this unidirectional single type PE structure makes it suitable for fault-tolerant designs. A unified EC full point add and double operation makes it attractive for SPA (Simple Power Analysis) resistant applications as well. The work being presented here has appeared in [19].

6.2 Related Work

Due to the multi-layer computation structure of the ECC over GF(2^m) (not considering the prime fields), many flavors of dedicated hardware implementation have been proposed *e.g.*, [4, 8, 27, 42, 52, 60, 72, 73, 90] which are reviewed and classified next.

Most of these proposals are implemented on the FGPA either as a prototype for a final ASIC design or as a final product to use the reconfigurability feature of the FPGA to provide a varying field implementation (its size and its defining polynomial). Almost all of them, except two, use the projective coordinates since they either lack an independent division unit or the performance of their division (inversion) is order of magnitude less than multiplication. Basically, all these proposals are based on a fast multiplier (and/or squarer) and assume that the complexity associated with the control unit is much less, hence transfer the burden of such complexity to a main processor (SW-HW co-design [26, 59, 88]). The proposal in [73] discusses the relative cost of a control unit to perform EC curve operations using projective coordinates by comparing a microcode and an FSM model of such a control unit. It is shown that equivalent of 512 rows of 16-bit words of opcode is needed for an EC *scalar* multiplication.

Many proposals are based on (optimal) normal basis representation of the underlying filed elements *e.g.*, [4, 27, 35, 53, 72, 102]. The (optimal) normal basis is known to have

an almost free squarer structure and it is possible to optimize its multiplier efficiently. Hence, the above proposals propose to couple a fast squarer and multiplier structure with either a generic or a DSP processor (using its control and general purpose registers). The implementation in [53] goes further to propose an asynchronous wave pipelined datapath (multiplier) coupled with synchronous control and memories.

Those based on the polynomial basis try to achieve a better performance by using special classes of field defining polynomials [42, 90]. In [42], a recent implementation is reported which consider an end-to-end performance analysis including the system level data exchange in the presence of a hybrid of a fast field multiplier for special class of polynomials plus a slow field multiplier for generic ones. This design which has an independent (slow) division unit, none the less, uses projective coordinates over polynomial basis. It achieves the best system level performance by optimizing the multiplier for specific irreducible polynomials.

Non-systolic implementations using affine coordinates exists as well [52, 60]. The proposal in [60] (over polynomial basis) is similar to our architecture in the sense that a combined multiplier divider is being proposed. However, its datapath requires m bit buses and multiple m bit parallel input units, hence its critical delay path is dimension dependent as reported in [60]. Further, an area limiting factor in many of the above prototypes, *e.g.*, [90], is the FPGA interconnect issue.

On the other hand, many stand-alone fast binary field multipliers, including systolic structures are proposed which do not consider the system level and control unit complexities, *e.g.*, [62, 104]. The proposal in [62] is a representative of a class of systolic implementations of $AB^2 + C \mod F(x)$ algorithm. In this case, extreme clock rates can be achieved, however, the inversion must be performed by using time consuming Fermat's theorem and necessitates the use of projective coordinates and the complex control units associated with.

6.3 Elliptic Curve Cryptography and EC Arithmetic

Elliptic curve cryptography (ECC) [83] is based on the discrete logarithm problem applied to elliptic curves over a finite field. An elliptic curve E is defined to be the set of points satisfying a defining equation $E(x, y) = 0$ with cubic degree in x where the coefficients of the defining equation are elements of a finite field GF(q), and the points on the curve are of the form $P = (x, y)$, where x and y are elements of GF(q) as well.

There are several kinds of defining equations for elliptic curves, defined over prime field or extension binary fields. Here, we are interested exclusively on hardware implementation of elliptic curve cryptosystems based on binary fields, hence a Weierstrass defining equation [2]

$$E(x, y): \quad y^2 + xy = x^3 + ax^2 + b \tag{6.1}$$

is used where $a, b \in \text{GF}(2^m),\ b \neq 0$. Given Equation (6.1), the elliptic curve E consists of the solutions $P(x, y)$ over GF(2^m) to the defining equation, along with an additional element called the *point at infinity* (denoted $\mathcal{O}$). The fundamental operation for ECC is *scalar* multiplication, *i.e.*, a point is added to itself k times.

$$Q = kP \tag{6.2}$$

$$= \underbrace{P \oplus P \oplus \cdots \oplus P}_{k \text{ times}} \tag{6.3}$$

Commonly, *scalar* multiplication is performed by successive *doubling* and *adding* the base point P similar to the binary method of exponentiation. Here, point addition is defined geometrically by the "chord-tangent" law of composition $\oplus$, *i.e.*, $P_2 = P_0 \oplus P_1$, where all three points will be on the curve E. Point doubling is defined by adding a point to itself, equivalent to a tangent to the curve, *i.e.*, $P_2 = 2P_0 = P_0 \oplus P_0$.

6.3.1 EC Point Add and Point Double over GF(2^m)

Algorithms to perform EC point add and double not only depend on the curve defining equation, *e.g.*, in the case of binary extension fields the Weierstrass Equation 6.1, but also they differ depending on the coordinate system of point representation as well. The affine

coordinate representation provides the simplest algorithms but its implementation is efficient only if a fast field division (relative to the field multiplication) is available. A unified EC full point add algorithm [2] using affine coordinates is modified such that its implementation to have identical steps and running time for both an EC point add and point double. This modification simplifies the control unit, reduces the storage requirements and also results in an SPA-resistant implementation. This is shown as Algorithm 22.

Algorithm 22 EC Full Point Add for Extension Binary Fields.

input: coefficient a of the curve, points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$.
output: Point $P_2 = (x_2, y_2) = P_0 \oplus P_1$
Part a: Exception Handling
if $(P_0 = \mathcal{O})$ **then**
return P_1
if $(P_1 = \mathcal{O})$ **then**
return P_0
if $((x_0 = x_1)$ AND $[(y_0 \neq y_1)$ OR $(x_0 = 0)])$ **then**
return $\mathcal{O}$
Part b: Main Computation
if $(x_0 \neq x_1)$ **then**
$\lambda \leftarrow \frac{y_1+y_0}{x_1+x_0} + 0$; // point add
else if $(x_0 = x_1)$ **then**
$\lambda \leftarrow \frac{y_1+0}{x_1+0} + x_1$; // point double
$x_2 \leftarrow \lambda^2 + \lambda + a + x_0 + x_1$;
$y_2 \leftarrow (x_1 + x_2)\lambda + x_2 + y_1$;
return $P_2 = (x_2, y_2)$

In Algorithm 22, the "0" represents the identity element of GF(2^m). Its computation part requires one division, one multiplication and one squaring and ten field additions. Further Algorithm 22 requires a very simple control (state machine) unit.

In Table 6.1, a stepwise implementation view of Algorithm 22 is shown to better point out the differences in each common steps of the EC point add and double. First, Table 6.1 shows that the same sequence of operations can be used to perform a point add or double. In other words, the three complex and multi-cycle operations *divide*, *square* and

Table 6.1: Comparison of Stepwise Computation Step of Algorithm 22.

Operation	Point Add	Point Double	Comment
*add	$w_0 = x_0 + x_1$	$w_0 = 0 + x_1$	
*add	$w_1 = y_0 + y_1$	$w_1 = 0 + y_1$	
divide	$w_1 = \frac{w_1}{w_0}$	$w_1 = \frac{w_1}{w_0}$	
*add	$w_1 = w_1 + 0$	$w_1 = w_1 + x_1$	λ (saved)
square	$w_0 = w_1 \times w_1$	$w_0 = w_1 \times w_1$	λ^2
add	$w_0 = w_0 + w_1$	$w_0 = w_0 + w_1$	$\lambda^2 + \lambda$
add	$w_0 = w_0 + a$	$w_0 = w_0 + a$	$\lambda^2 + \lambda + a$
*add	$w_0 = w_0 + x_0$	$w_0 = w_0 + x_1$	$\lambda^2 + \lambda + a + x_0$
add	$w_0 = w_0 + x_1$	$w_0 = w_0 + x_1$	x_2 (saved)
add	$w_2 = w_0 + x_1$	$w_0 = w_0 + x_1$	$x_2 + x_1$
multiply	$w_2 = w_2 \times w_1$	$w_0 = w_2 \times w_1$	$(x_2 + x_1)\lambda$
add	$w_2 = w_2 + w_0$	$w_2 = w_2 + w_0$	$(x_2 + x_1)\lambda + x_2$
add	$w_2 = w_2 + y_1$	$w_2 = w_2 + y_1$	y_2 (saved)

multiply are common and only the input to the intermediate *add* operations may vary. The binary field addition can be simply implemented by an XOR operator, hence it is cost free. If a selective input to an XOR for *add* operation at exactly four steps (highlighted by an * in Table 6.1) are permitted, then a single bit control signal suffices to distinguish between point add and double. The XOR operation for *add* is free relative to multi-cycle multiplication or division and it can be overlapped with a pipelined multiplier and divider functional unit in a bit serial architecture. Hence, in this setting, the selection between the point add and double is transparent with no running time distinction.

Further, in a bit serial architecture, where all *add* operations between consecutive *divide*, *square* and *multiply* operations can be performed bitwise and in parallel, then three temporary variables w_0, w_1, w_2, as shown in Table 6.1, are not needed. In fact, a single shift register for λ is sufficient since this is the only computed variable needed more than once in each full point add computation. This results in the least amount of the storage space for an EC point add computation.

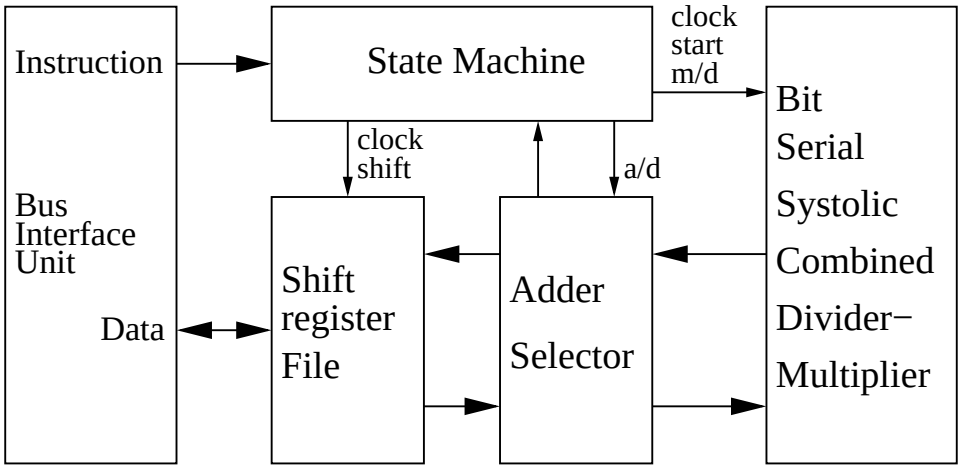


Figure 6.1: System level integration.

In Section 6.4, a bit serial systolic architecture for a combined multiplier and divider (CMD) is proposed. In this proposal the same unit will perform multiplication, squaring and division. The multiplication and squaring are exactly the same, and a division requires a running time twice as a multiplication. What is more interesting is that two distinct operations *divide* and *multiply* can be pipelined during the same EC point add computation.

6.3.2 System Level Block Diagram of an EC Cryptoprocessor

Before describing a CMD datatpath design it is important that a system level and a top view of an EC cryptoprocessor to be considered. Figure 6.1 depicts such a design consists of four units: bus interface unit, control unit, register file and datapath.

In Figure 6.1, the datapath is shown in two blocks: adder selector and bit serial systolic combined multiplier-divider (CMD). The register file will be a set of ten shift registers, including all input and output registers (irreducible polynomial, curve parameters, key register, input points and output point coordinates) accessible to the host system through the bus interface unit. The shift registers act as input-output host accessible registers as well as buffering registers for the CMD. Hence, the size of each shift register is the same

as the field size in a simple implementation but may be larger if external buffering to the CMD is required.

The bus interface unit (BIU) is mainly and simply responsible for the address decoding of the registers, and its handshake signal generation to the host is minimal. The registers are parallel write and readable by the host using the bus word size which is in general much smaller than the field size. Hence, the shift registers act as a buffer to resolve the data size and the data rate mismatch between the host and the CMD, (note that the CMD may run at multiple clock rate of the host). Internally, the registers are accessed exclusively bit serially.

The EC cryptoprocessor can perform a suite of low level field operations, as well as high level EC curve full point add, and most importantly, the EC *scalar* multiplication. A specific instruction register in the control unit will be used to decode the operation to perform. This register will be directly accessible by the host. No interaction between the host and the EC cryptoprocessor is required except to write and read the data and the operation name.

The peculiar and novel aspect of this EC cryptoprocessor is the simplicity and the reduced size of its control unit which is based on a robust state machine (with all error handling mechanism). This unit includes a very simple binary key parsing to perform the *scalar* multiplication. In fact, the control unit of Figure 6.1 generates only 3 controls signals in addition to the clock and the set of shift signals for the registers.

In the next section the major functional unit of this EC cryptoprocessor will be discussed.

6.4 EC Bit-Serial Systolic Accelerator over GF(2^m)

In Chapter 5, it has been said that systolic architectures are suitable for computations where inherent parallelism can be exploited by bit level pipelining. Computations like exponentiation, or EC *scalar* multiplication with their repetitive multiplication squaring steps are good candidates. However, the inversion or division step for EC routines may complicate the design if a similar multiplier and divider pipelined structure does not exist.

On the other hand, the area requirement of a bit parallel systolic structure is prohibitive specially for large size operands such as those used in cryptographic applications. In practice, many digit serial alternatives have been proposed and deployed. In an area constraint application such as an embedded system a bit serial systolic architecture may be the best choice if the throughput requirements are relaxed and it can be met.

A drawback of systolic architectures in general, even in a bit serial design is the high ratio of the latch area versus the logic area (typical values between 70% to 90% of the total area). On the other hand, to achieve higher clock rates, it is known that the gate delay between two consecutive latches (or active edge of the clock of flip-flops) should be shorten. However, it is easy to compute that at a certain point (depending on the technology and its standard library) the combined delay of latches, setup time of flops, skew of clock and the interconnect delay, will exceed the delay of a specific number of gates. This means that any further reduction of the number of gates between two consecutive latches, at the expense of adding a new row of pipeline latches will become counter productive. Hence, a flexible design which must consider the technology variations is desirable.

In the following, a combined multiplier-divider (CMD) architecture is proposed which not only exploits the pipelining among different operations, but most importantly it reduces the disadvantage of the latch area overhead by combining more computational logic in the same PE and reusing the latches among multiplication and division. Figure 6.2 depicts a bit serial unidirectional systolic architecture for such a combined multiplication-division design. As one may notice once again Figure 6.2 has the same structure as Figure 5.1.

In Figure 6.2, three sets of inputs: *datapath* signals (R, S, U, V, F), *state* signals (*dec*, *dseq*, *update*), and *control* signals *start* and m/d are distinguished. These signals will be defined in the following two sections where systolic multiplier and divider architectures are described individually.

6.4.1 Bit Serial Systolic Architecture for Field Multiplication

The binary finite field multiplication and division can be systolized easily due to the recursive feature of their algorithms. There are two class of binary finite field multiplication algorithms, MSB-first and LSB-first [58]. Algorithm 23 is a reformulating of the one pro-

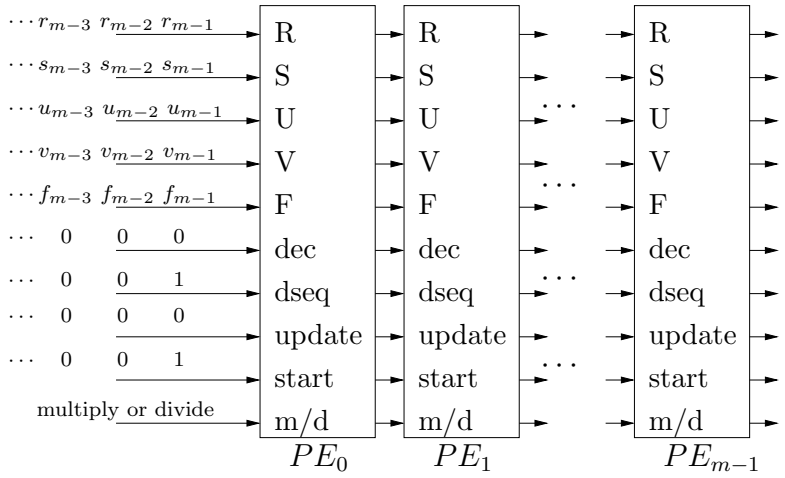


Figure 6.2: Bit serial systolic architecture for a CMD.

posed initially by Scott et al. [97] which has been reused by many authors in digit-serial and partitioned forms, *e.g.*, [61], where $R(x)$ and $S(x)$ represent two elements of GF(2^m) of degree at most $m-1$, and $F(x)$ is the irreducible defining polynomial of degree m used to generate the field. Let $U(x)$ be the product of $R(x)S(x) \mod F(x)$ of degree at most $m-1$ at well.

In an MSB-first bit serial systolic multiplier with identical processing elements (PE), the index i of Algorithm 23 will be mapped into the spatial location (the specific PE) and the j index will be distributed over the temporal sequence of computations in each PE. The coefficients of $U(x)$ will be returned bitwise MSB-first. Such a systolic implementation has been proposed by Wang and Lin [107]. Figure 6.3 depicts one PE of such a bit serial multiplier.

In Figure 6.3, as well as in Algorithm 23, the polynomials are named such that to be consistent with an MSB-first divider architecture presented in the next section. The most significant bit (msb) of the $U(x)$, *i.e.*, u_{m-1} exits the mth PE at $2m$th cycle after a ($start = 1$) enters the first PE. The control signal *start* consists of a one followed with $m-1$ zeroes. It is used to latch the most significant bits of accumulator u and input s. Clearly, in this architecture the overhead of 10 latches versus only 6 logical operators per PE can be noted. On the other hand, the critical delay path (gate delay) is as short as

Algorithm 23 MSB-first Multiplication in GF(2^m)

input: $R(x)$, $S(x)$, $F(x)$.

output: $U(x)$

$u_j^{(0)} = 0 \qquad$ for $0 \leq j \leq m-1$

$u_0^{(i)} = 0 \qquad$ for $1 \leq i \leq m-1$

for $(i = 0$ to $m-1)$ **do**

for $(j = m-1$ to $0)$ **do**

$u_j^{(i+1)} \leftarrow u_{m-1}^{(i)} f_j + s_{m-i+1} r_j + u_{j-1}^{(i)};$

return $U(x) = u_j^{(m-1)} \qquad$ for $0 \leq j \leq m-1$

$t_g = T_{2A} + T_{3X}$, where T_{2A} is the gate delay of a 2-input AND gate and T_{3X} that of 3-input XOR gate.

6.4.2 Bit Serial Systolic Architecture for Field Division

For division the same PE as defined in Section 5.3.3 can be used. It is said that a bit serial systolic field divider with $2m$ PE of Figure 5.3 computes a division result in $4m$ cycles. However, it is also described that only m PEs are sufficient if the output of the mth PE is fed back to the first PE at the $2m$th cycle. In this proposal initially such an arrangement of m PEs is followed. Hence, a result is computed always in $4m$ cycles after the same m PEs are initialized and used twice. Such an architecture computes the result $U(x)$ bitwise MSB-first according to Equation 6.4.

$$U(x) = \frac{V(x)}{S(x)} \mod F(x), \tag{6.4}$$

The input $V(x)$ represents the dividend and the input $S(x)$ the divisor. The other input $R(x)$ as well as input $F(x)$ is initialized with $F(x)$, the irreducible polynomial. The input $U(x)$ is initialized with 0.

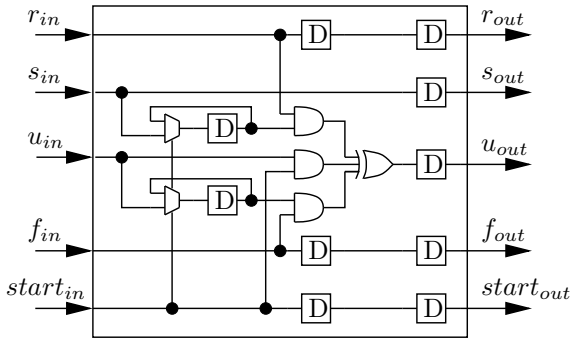


Figure 6.3: Processing element for the multiplier.

Recall that PE presented in Figure 5.3 keeps track of the degree of polynomials in a distributed fashion with no carry propagation structures (counters). This results in a linear time complexity of the architecture independent of the size of m. The control signals *start*, *dec* and *dseq* are those described in Section 5.3.2. The critical delay path of the divider is dominant with respect to the multiplier and the entire system level architecture as will be discussed later. The maximum gate delay of the divider is balanced to correspond to a maximum of the four terms: ($3T_{A2}$, $2T_{A2}+T_{X2}$, $T_{A2}+T_{X2}+T_{M2}$, plus T_{M2}), where T_{A2}, T_{M2} and T_{X2} are defined as the delay of a 2-input AND gate, MUX and XOR gate respectively. As mentioned before, the total critical delay path must include the latch delay, the clock skew, and the interconnect delay which we may assume are common between multiplier PE, divider PE and a bit serial XORing for field additions.

6.4.3 Processing Element of a Combined Multiplier Divider

In order to reduce the area overhead of the latches of Figures 6.3 and 5.3, a combined architecture as shown in Figure 6.4 can be used.

The PE in Figure 6.4 has only 20 latches for the combined structures. It also includes twelve 2-input MUX gates, three 2-input and one 3-input XOR gate, and ten 2-input AND/OR gates. In this structure only one control signal m/d is added to select between

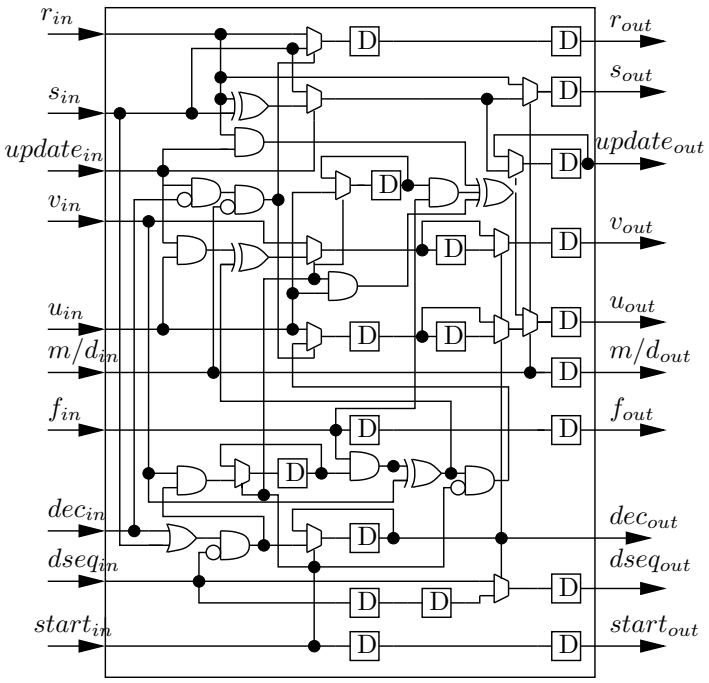


Figure 6.4: Processing element of a CMD.

multiplication and division. The maximum re-use of resources (latches, gates and IO pins) are applied without affecting the maximum gate delay, hence the critical delay path, of the PE which remains the same as the divider PE described in Section 6.4.2. In fact, extra MUXs required for the m/d selection are not on the critical delay path of the PE.

6.5 Implementation Issues

In this section the architectural alternatives and implementation issues regarding the three main sub-blocks of the EC cryptoprocessor are discussed.

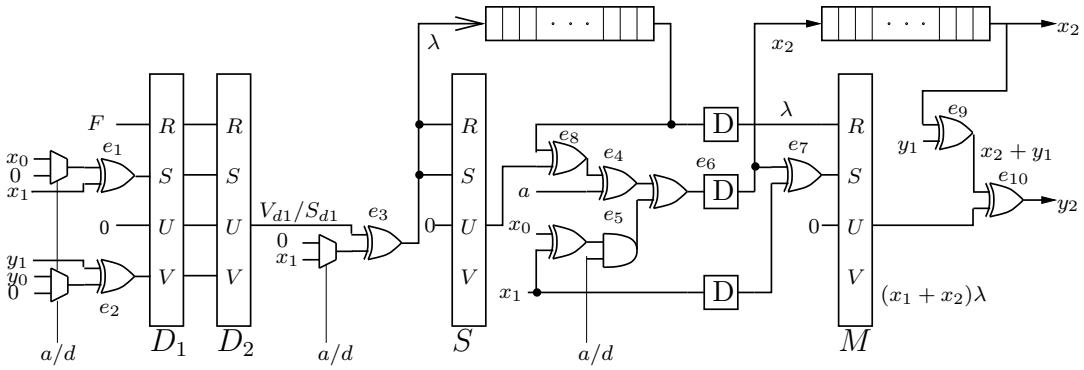


Figure 6.5: Bit serial functional sequence of an EC full point add computation.

6.5.1 EC Full Point Add using the CMD Datapath

Considering the bit serial systolic architecture of Figure 6.2, which consist of single type PEs of a CMD as shown in Figure 6.4, a bit serial computation of an EC full point add can be viewed as the concatenation of four systolic CMDs. Such a functional arrangement is shown in Figure 6.5.

In a bit serial architecture as in Figure 6.5, four identical CMDs can be used in a pipelined fashioned to perform the division in two slots of $2m$ cycles each and also the squaring and the multiplication in $2m$ cycles each. On the other hand, a single CMD can be reused four times at a lower throughput if adequate MUXing at its inputs are provided. In Figure 6.5, three MUXs and an AND gate with a select signal a/d are used to choose between the EC point add and point double. It is said that this selection is external to the CMD and applies only to the XOR gates which perform the field addition. More importantly, all ten field additions required in an EC full add operation can be computed on the fly between the pipelined computation of division, squaring and multiplication by using exactly 10 XOR gates.

In an area constraint case where only a single CMD is used, the inputs to the four functional CMDs must be MUXed. Figure 6.6 shows the selection of the four cases: division step1, division step2, squaring and the multiplication for all inputs.

Considering the effect of 2 layers of MUXs at all inputs of the same CMD in this case,

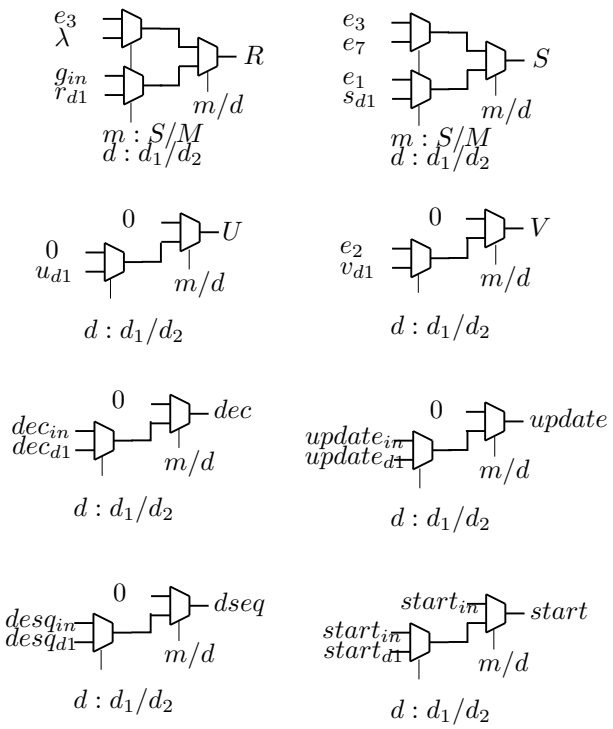


Figure 6.6: Inputs selection of a CMD for four operational cases.

the insertion of three delay latches in Figure 6.5 ensures that the maximum gate delay, hence the critical delay path, external to the CMD remains less than that of a single PE as discussed in Section 6.4.2. The XOR and MUX gates of Figures 6.5 and 6.6 are grouped together in the Adder-Selector unit shown in Figure 6.1 which interface both the CMD and the shift register file. According to Figures 6.2, 6.5 and 6.6, an EC full point add operation requires only four *control* signals, and three *state* signals for the division, in addition to the clock and the shift signals to the corresponding shift registers. This is an indication of the simplicity of an area efficient state machine required for this architecture.

6.5.2 An FSM-type Control Unit

It is mentioned that one of the advantageous to use Algorithm 22 to compute the EC full point add in affine coordinates is the simplicity of the control unit. In fact, by presetting a/d signal, the EC full point add algorithm has no conditional branching. Consequently, all 13 steps of Table 6.1 to compute an EC full point add can be hardwired as shown in Figure 6.5. In fact, a simple sequencer is used to generate the control signal *start* for time slots of $2m$ cycles, and to check the exit of this signal out of the mth PE of the CMD by comparing with an independent ring-counter of size $2m$. This comparison is used to generate the *done* signal of the operation to the host and also for a partial sanity check of the correctness of CMD functionality.

The other signals generated by the sequencer are: select lines for the MUXs in the input of the CMD (coded as a two bit value) and the selective shift signals to the specific shift register at each step of full point add according to the Table 6.1.

As a preliminary EC *scalar* multiplication design, *e.g.*, $Q = kP$, it is decided to use a simple binary scheme to parse the k to set the a/d signal directly based on its bit values. This results in the only conditional branching required in this control unit. Better schemes to speedup the *scalar* multiplications exists. These will be considered in future improvements of the design.

The area requirement of the control unit is negligible with respect to the CMD or the shift register file. Also, its design required no pipelining (as customarily used in the microcode or wired FSM designs of control units) to reduce its critical path. You may noticed that a main large counter required in the sequencer was designed as a ring-counter with no carry propagation structure in the control unit.

6.5.3 Scalability and Dealing with Varying Dimension

In general, the bit serial unidirectional architecture in Figure 6.2 may process either fields of dimension m or any field dimension less than m as far as the correct result can be captured at the right clock cycle. In an architecture where the same m PEs are used twice to perform a single division, any field dimension between $m/2$ and m can be processed.

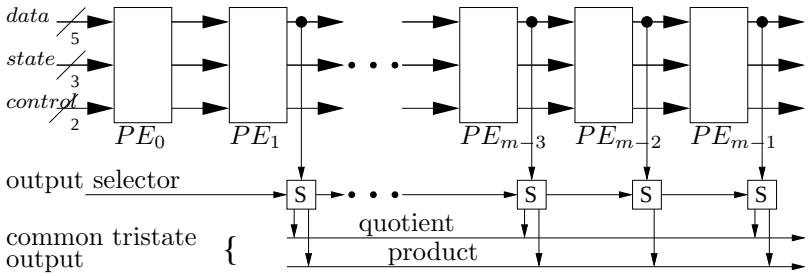


Figure 6.7: Variable dimension architecture using selectors over common tristate output.

One such area efficient scheme using a selector circuitry over a common tristate output line similar to Figure 5.5 is shown in Figure 6.7. The main difference is a double output bus which must distinguish between the result of division (quotient) and multiplication.

One of the selector latches can be set such that only the specific output corresponding to this latch is passed to the common output, and all other outputs are kept in tristate. Figure 6.7 represents a serial-in mechanism to set the latch corresponding to the field dimension.

The given latch is set only once per each change of field dimension and before the EC *scalar* multiplication starts. In a CMD structure, where the divider uses the same functional unit twice, two such common tristate outputs are required, since the product and the quotient must be captured at different PE output. For the division only the output of every other PE requires a selector latch.

6.5.4 Implementation Results

A prototype architecture of a CMD for different values of m is coded in Verilog-HDL, simulated and verified. Recall that the value of m in itself is no limitation to computation of the critical delay path of the circuit which is independent of m. In general, for larger values of m the area of the CMD is linearly proportional to m, while the clock rate does not increase, the throughput is not changed, although the (initial) latency increases.

One of the main characteristics of this bit serial systolic architecture is its ease to the

scalability. The Verilog model of a complete architecture based on Figures 6.1, 6.2, 6.4, 6.5, and 6.6 has been synthesized using Synopsys with a standard CMOS 0.18μ library for a clock period of 1.4 ns. The critical delay path of the complete architecture was in fact internal to a PE of Figure 6.4. The area report of a single PE, including the flip-flops, are equivalent to 145 2-input NAND gates. It should be pointed out that non-combinational area is still the dominant part, more than 65%. The register file shown in Figure 6.1 consists of 10 shift registers of size m.

Two other configurations are modeled as well to better estimate the area-throughput-latency trade-offs. These alternatives are modeled using a fixed $m = 163$. One alternative with four copies of the CMD to increase pipelining and throughput with a simpler control module. The area requirement for the CMD, even though it has increased linearly, becomes much more dominate with respect to the shift register storage space. However, as expected the throughput of this configuration reaches optimal value for such systolic architecture while further simplifying the control unit.

Next, a model with a CMD of exactly 163 PEs, but with adequate shift registers as buffers to process a curve with an underlying field of size 233 is modeled. In this case, not only the throughput (as well as latency) was slower, but more importantly, it required a more complex control unit, with a significant added complexity for its verification.

On the other hand, a complete parametrized bit serial unidirectional systolic architecture with a small control and a set of shift registers for input and output operands has been designed. For larger values of m, *e.g.*, Elliptic Curve Cryptography (NIST recommended trinomials and pentanomials with field dimensions 163, 233, 283, 409, 571), the area of the control unit is a negligibly smaller constant compared to the systolic path and the area complexity of the entire design becomes linearly dependent on m, *e.g.*, equivalent to $120 \times 2m$ 2-input NAND gates (plus $4 \times m$ flip-flops for input and output shift registers).

As one may guess, the area is independent of the Hamming weight of the defining irreducible polynomial used.

The time complexity of this design remains constant and is bounded to the time complexity measure of a CMD as described earlier. For example, considering only module level constraints, requirement to perform back to back divisions or in a pipelined setting with other bit serial operations and a field size of $m = 163$, one operation every 0.24 μs

or equivalent to more than 4 million operations per second is feasible. As said before, if necessary an initial latency of almost four times above value must be added. For other field dimensions a simple extrapolation with no hidden overheads can be used.

Last, the feasibility of reusing the same m PE's to compute the division without the use of any external buffering has been tested. In this case, the extra complexity of the state machine of the control unit is relatively negligible and as anticipated, the area of the data path is reduced by half while all operations took twice to complete.

6.5.5 Comparison

Two of the implementations described in Section 6.2 provide valid metrics for our proposal. The implementation in [42] is the best performer which computes a field multiplication using polynomials basis and the projective coordinates for a specific irreducible polynomial over GF(2^m) in only 4 cycles (at 66 MHz) which translates in almost 7000 *scalar* multiplication per second. However, it is reported that for defining polynomials (with the second MSB less than $m/2$), the performance decreases by more than an order of magnitude. In [42], it is implied that the maximum usage in area of a Xilinx XCV2000E with a dominant multiplier is done.

In the other implementation on the same FPGA proposed in [60], a combined multiplier divider design using polynomial basis and the affine coordinates is used. In this implementation a *scalar* multiplication is computed in $9m^2$ clock cycles over any defining polynomial of any dimension up to a maximum size, where its 256 bit datapath can run at 40 MHz and has a reported gate count of 74103 (this does not include the SRAMS module and the control required). In contrast to these two designs, our proposal has an estimated area of 35K gates for a CMD of 256 bits, it computes a *scalar* multiplication in $8m^2$ cycles and most importantly it critical path is 1.4 ns which may result in a device running at over 700 MHz. Considering this clock rate and its area estimate, our design outperforms two above implementations when a generic defining polynomial implementation is required.

6.6 Summary

In this chapter an area efficient elliptic curve processor has been investigated whose datapath is based on a combined multiplier divider (CMD) bit serial systolic architecture which implements both multiplication and division over GF(2^m) in a single hardware unit and may achieve high performance by running at high clock rates. This combined multiplier divider and its systolic structure allows using the EC affine coordinates, hence having a very small and simple control unit and the least amount of storage space.

This structure coupled with a shift register file provides an efficient bit-level pipelining among successive field multiplication and division operations with in-line XORing which results in further pipelining of consecutive EC full point additions.

A trade-off between scalability and throughput has been studied: using adequate external buffering for processing field dimensions larger and independent of the number of PEs present in the datapath at the expense of lower throughput. It has been shown that for high throughput applications, multiple CMD units can easily be pipelined due to their unidirectional bit serial architecture.

This structure outperforms other EC processors when variable field dimension and any irreducible polynomial usage is a requirement. Furthermore, this unidirectional single type PE structure makes it suitable for fault-tolerant designs.

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusions

In this work efficient hardware implementation of multiplicative inversion and division over Galois fields with polynomial or triangular basis representation of elements has been investigated. Specifically, a detailed survey of many implementations of EEA (Extended Euclidean Algorithm) and binary (Stein's) GCD algorithm to compute multiplicative inversion or field division with polynomial basis has been reviewed. A set of common tasks among all these variants is defined and some major implementation issues have been highlighted. Further, a common set of algorithms has been described which could be used to compute inversion or division when the input divisor is in either polynomial or triangular basis. In this case, some comments regarding the formation of coefficients of the Hankel matrix used in triangular basis inversion (or division) has been summarized, and initialization modifications have been discussed which enable us to use exactly the same architecture in both bases. Further some key results on inversion and division using triangular basis have been presented.

Next, systolization as a solution to the VLSI design complexity of computational inten-

sive and iterative algorithms has been reviewed. A major complexity issue, the interconnect bottleneck in the sub-micron design of algorithms like EEA has been explained.

Then, a novel class of unidirectional bit serial systolic architecture to implement multiplicative inversion and division over Galois fields has been proposed. This architecture uses a variant of EEA optimized for unidirectional systolization with no carry propagation structure. The same algorithm and architecture can be used to perform inversion and division where the input divisor is in triangular basis. Main contribution is the restoring mechanism introduced in Algorithm 21 and its implementation using double delay elements in Figure 5.3. Also, a simpler distributed counter structure has been proposed. This architecture is suitable for applications where the field dimension may be large or variable (such as cryptographic applications). This architecture is well suitable for high clock rates applications.

Finally, In Chapter 6, the design of a high performance systolic elliptic curve processor has been investigated. It has been stated that if a field division can be performed at the speed of a field multiplication, then the complexity of a control unit of an EC scalar multiplication processor using affine coordinates becomes negligible with respect to its datapath. Hence, a datapath based on a combined multiplier divider (CMD) bit serial systolic architecture is implemented where both multiplication and division over GF(2^m) are performed in a single hardware unit in a pipelined organization. This has resulted in a high performance by achieving high clock rates.

This unidirectional bit serial datapath structure coupled with a shift register file provides an efficient bit level pipelining among successive field multiplication and division operations which has resulted in further pipelining of consecutive EC full point additions.

A trade-off between scalability and throughput has been studied: using adequate external buffering for processing field dimensions larger and independent of the number of PEs present in the datapath at the expense of lower throughput. It has been shown that for high throughput applications, multiple CMD units can easily be pipelined due to their unidirectional bit serial architecture.

This structure outperforms other EC processors at a system level specially when variable field dimension and irreducible polynomial usage is a requirement.

7.2 Future Work

In this thesis a very specific objective, to design a high performance but also scalable architecture of multiplicative inversion and division for large and variable values of field dimension, m, was set. Many schemes have been investigated and a specific implementation, a bit serial unidirectional systolic structure, has been selected mainly for its suitability to reduce the interconnect complexity of VLSI design. However, improving some aspects of this architecture can be further studied, and certain features of this design can be generalized in other applications. A list of optimization and generalizations follows:

1. *Circuit level optimization:* Major issue with any systolic architecture in general and this bit serial architecture in particular is the high ratio of latch elements (flip-flops) versus the logic. Considering advantages of a bit level pipelining, a possible investigation to use or design practical alternative latch models instead of standard flip-flops is desirable. It is known that AT-complexity is highly dependent on the delay elements used. Both delay or area optimizations of a latch design may be explored further.

2. *Architecture level generalization:* A possible alternative to mitigate the high cost of latch elements versus the logic is the case of generalizing the bit serial architecture to an optimal digit serial design. This may not necessarily be a solution to the above point. However, any bit serial structure can be transformed into a digit serial one, by folding techniques, [39, 70], if a higher throughput is a requirement. A digit serial alternative can be further studied to formulate an optimal throughput-area relationship.

3. *Algorithmic generalization:* In this thesis a specific implementation of EEA, *i.e.*, left-shift GCD algorithms, applied to computing inversion and division over Galois fields is studied. However, the same structure with slight modifications can be used for right-shift GCD algorithm and thus can be extended to perform a unified dual field (prime and binary) inversion and division as well.

4. *Fault-tolerant design optimization:* The unidirectionality of this architecture makes it suitable for fault-tolerant design paradigm [76]. Further with simple extra circuitry

it is possible to verify the correctness of the result of this structure.

5. *SPA (Simple Power Analysis):* A unified EC full point add and multiply operation makes this systolic EC crypto processor resistant to SPA attack. However, this has not be proved. Further, studying an efficient key parsing methodology to be resistant to both SPA and DPA (Differential Power Analysis) type attacks for this specific architecture would be interesting.

Bibliography

[1] Tolga Acar. *High-Speed Algorithms & Architectures For Number-Theoretic Cryptosystems*. PhD thesis, Oregon State University, Dec. 1997.

[2] IEEE 1363-2000: adopted IEEE Standard. "IEEE Standard Specifications for Public Key Cryptography". *http://standards.ieee.org/catalog/olis/busarch.html*, 2000.

[3] G.B. Agnew, T. Beth, R.C. Mullin, and S.A. Vanstone. "Arithmetic Operations in GF(2^m)". *Journal of Cryptology*, 6(1):3–13, June 1993.

[4] G.B. Agnew, R.C. Mullin, and S.A. Vanstone. "An Implementation of Elliptic Curve Cryptosystems Over $F_{2^{155}}$". *IEEE J. Selected Areas of Communications*, 11(5):804–813, June 1993.

[5] K. Araki, I. Fujita, and M. Morisue. "Fast Inverter Over Finite Field Based on Euclid's Algorithm". *Tran. Inst. Electronics, Information. And Comm. Engineers*, E-72:1230–1234, Nov. 1989.

[6] Daniel Bailey. *Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms*. Worcester Polytechnic Institute, May 1998. Master's Thesis.

[7] M. Bednara, M. Daldrup, J. Teich, J. von zur Gathen, and J. Shokrollahi. "Tradeoff analysis of FPGA based elliptic curve cryptography". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, 5:797–800, 2002.

[8] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich. "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms". *16th International Parallel and Distributed Processing Symposium*, pages 154–167, 2002.

[9] E. Berlekamp, G. Seroussi, and P. Tong. *Reed-Solomon Codes and Their Applications, Editors S.B. Wicker and V.K. Bhargava.* Chapter 10. IEEE Press, Piscataway NJ, 1994. A Hypersystolic Reed-Solomon Decoder.

[10] E. R. Berlekamp. "Bit-Serial Reed-Solomon Encoders". *IEEE Transactions on Information Theory*, 28:869–874, Nov. 1982.

[11] E.R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill Book Company, 1968.

[12] E.R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill Book Company, 1968. Section 2.3.

[13] R.P. Brent and H.T. Kung. "Systolic VLSI Arrays for Polynomial GCD Computation". *IEEE Transactions on Information Theory*, c-33(8):731–736, Aug. 1984.

[14] R.P. Brent, H.T. Kung, and F.T. Luk. "Some Linear-Time Algorithms for Systolic Arrays". *"Proceedings of International Federation of Information Processing (IFIP), 9th World Computer Congress, Paris, France"*, pages 865–876, Sept. 1983.

[15] H. Brunner, A. Curiger, and M. Hofstetter. "On Computing Multiplicative Inverses in GF(2^m)". *IEEE Transactions on Computers*, 42(8):1010–1015, Aug. 1993.

[16] A. Daneshbeh and M.A. Hasan. "On Double-Basis Inversion and Division over GF(2^m)". Manuscript under preparation.

[17] A. Daneshbeh and M.A. Hasan. "A Class of Scalable Unidirectional Bit Serial Systolic Architectures for Multiplicative Inversion and Division over GF(2^m)". `http://www.cacr.math.uwaterloo.ca`, Dec 2002. Technical Report CORR 2002-35.

[18] A.K. Daneshbeh and M.A. Hasan. "A Unidirectional Bit Serial Systolic Architecture for Double-Basis Division over GF(2^m)". *Proceedings of the 16th. IEEE Symposium on Computer Arithmetic, Arith-16*, pages 174–187, June 2003. A conference article based on [17].

[19] A.K. Daneshbeh and M.A. Hasan. "Area Efficient High Speed Elliptic Curve Systolic Cryptoprocessor for Random Curves". *Proceedings of the International Conference on Information Technology, Coding and Computation. ITCC'04*, 2:588–592, April 2004.

[20] A.K. Daneshbeh and M.A. Hasan. "A Class of Unidirectional Bit Serial Systolic Architectures for Multiplicative Inversion and Division over GF(2^m)". *IEEE Transactions on Computers*, 54(3):–, March 2005. Accepted to be published. An extended journal article based on [17].

[21] G.I. Davida. "Inverse of Elements of a Galois Field". *Electronics Letters*, 8(2):22–30, Mar. 1972.

[22] E. Delgado. e-Commerce and Trustmarks: Results from the ALPINE Working Group. *URL: http://elab.vanderbilt.edu/elib/upload/196/A roadmap on Trustmarks.pdf*, December 2002. European Software Institute.

[23] W. Diffie and M. Hellman. "New Directions in Cryptography". *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[24] Mohamed El-Gebaly. *Finite field Multiplier Architectures for Cryptographic Applications*. PhD thesis, University of Waterloo, 2000.

[25] T. ElGamal. "A Public-key Cryptosystem and a Signature Scheme based on Discrete Logarithms". *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[26] M.M. Elgebaly and M.A. Hasan. "Elliptic Curve Diffie-Hellman Key Exchange Coprocessor". *20th Biennial Symposium on Communications, Kingston, Ontario, Canada*, pages 54–58, May 2000.

[27] M. Ernst, S. Klupsch, O. Hauck, and S.A. Huss. "Rapid prototyping for hardware accelerated elliptic curve public-key cryptosystems". *12th International Workshop on Rapid System Prototyping, RSP '01*, pages 24–29, 2001.

[28] G. Feng. "A VLSI Architecture for Fast Inversion in GF(2^m)". *IEEE Transactions on Information Theory*, 38(10):1383–1386, Oct. 1989.

[29] S.T.J. Fenn, M. Benaissa, and D. Taylor. "Bit-serial Dual Basis Systolic Multipliers for GF(2^m)". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'95*, 3:2000–2003, 1995.

[30] S.T.J. Fenn, M. Benaissa, and D. Taylor. "Finite Field Inversion Over the Dual Basis". *IEEE Transactions on VLSI Systems*, 4(1):134–137, March 1996.

[31] S.T.J. Fenn, M. Benaissa, and D. Taylor. "Dual Basis Systolic Multipliers for GF(2^m)". *IEE Proceedings. Computers and Digital Techniques*, 144(1):43–46, Jan. 1997.

[32] P. Fitzpatrick, J. Nelson, and G. Norton. "A Systolic Version of The Extended Euclidean Algorithm". *Systolic Array processors*, pages 477–486, 1989. Edited by J. McCanny and J. McWhirter.

[33] K. Fong, D. Hankerson, J. Lopez, and A. Menezes. "Field Inversion and Point Halving Revisited". *IEEE Transactions on Computers*, 53(8):1047–1059, August 2004.

[34] R. Furness, M. Benaissa, and S.T.J. Fenn. "GF(2^m) Multiplication over Triangular Basis for Design of Reed-Solomon Codes". *IEE Proceedings. Computers and Digital Techniques*, 145(6):437–443, Nov. 1998.

[35] L. Gao, S. Shrivastava, H. Lee, and G.E. Sobelman. "A compact fast variable key size elliptic curve cryptosystem coprocessor". *7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '99*, pages 304–305, 1999.

[36] W.J. Gilbert and S.A. Vanstone. *Classical Algebra*. Waterloo Mathematics Foundation, 1993.

[37] J.-H. Guo and C.-L. Wang. "Bit-serial Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in $GF(2^m)$". *Proceedings of Technical Papers, International Symposium on VLSI Technology, Systems, and Applications*, pages 113–117, June 1997.

[38] J.-H. Guo and C.-L. Wang. "Digit-serial Systolic Multiplier for Finite Fields GF(2^m)". *IEE Proceedings. Computers and Digital Techniques*, 145(2):143–148, Mar. 1998.

[39] J.-H. Guo and C.-L. Wang. "Novel Digit-serial Systolic Array Implementation of Euclid's algorithm for Division in $GF(2^m)$". *Proceedings of the IEEE International Symposium on Circuits and Systems, 1998. ISCAS'98*, 2:478–481, 1998.

[40] J.-H. Guo and C.-L. Wang. "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)". *IEEE Transactions on Computers*, 47(10):1161–1167, Oct. 1998.

[41] J.-H. Guo and C.-L. Wang. "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in GF(2^m)". *IEE Proceedings Computers and Digital Techniques*, 145(4):272–278, July 1998.

[42] N. Gura, S. H. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchenstein, E. Goupy, and D. Stebila. "An End-to-End Systems Approach to Elliptic Curve Cryptography". *4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002, LNCS 2523, Springer-Verlag*, pages 351–366, 2002.

[43] A. Halbutogullari and C.K. Koc. "Mastrovito Multiplier for General Irreducible Polynomials". *IEEE Transactions on Computers*, 49(5):503–518, May 2000.

[44] Alper Halbutogullari. *Fast Bit-Level, word-Level and Parallel Arithmetic in Finite Fields for Elliptic Curve Cryptosystems*. PhD thesis, Oregon State University, Nov. 1998.

[45] D. Hankerson, J.L. Hernandez, and A. Menezes. "Software Implementation of Elliptic Curve Cryptography over Binary Fields". *2nd International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, LNCS 1965, Springer-Verlag*, pages 1–24, August 2000.

[46] M.A. Hasan. "Double-Basis Multiplicative Inversion over GF(2^m)". *IEEE Transactions on Computers*, 47(9):960–970, Sep. 1998.

[47] M.A. Hasan. "Efficient Computation of Multiplicative Inverses for Cryptographic Applications". *Proceedings of the 15th. IEEE Symposium on Computer Arithmetic, Arith-15*, pages 66–72, June 2001.

[48] M.A. Hasan and V.K. Bhargava. "Bit-Serial Systolic Divider and Multiplier for Finite Fields GF(2^m)". *IEEE Transactions on Computers*, 41(8):972–980, Aug. 1992.

[49] M.A. Hasan and V.K. Bhargava. "Division and Bit-Serial Multiplication over GF(q^m)". *IEE proceedings of Electronics*, 139:230–236, May 1992.

[50] M.A. Hasan and V.K. Bhargava. "Architecture for a Low Complexity Rate-Adaptive Reed-Solomon Encoder". *IEEE Transactions on Computers*, 44(7):938–942, July 1995.

[51] M.A. Hasan, M.Z. 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, Aug. 1992.

[52] M.A. Hasan and A.G. Wassal. "VLSI Algorithms, Architectures and Implementation of a Versatile GF(2^m) Processor". *IEEE Transactions on Computers*, 49(10):1064–1073, Oct. 2000.

[53] O. Hauck, A. Katoch, and S.A. Huss. "VLSI system design using asynchronous wave pipelines: a 0.35 μm CMOS 1.5 GHz elliptic curve public key cryptosystem chip". *6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, ASYNC 2000*, pages 188–198, 2000.

[54] C.P. Hong, C.H. Kim, S.H. Kwan, and I.G. Nam. "Efficient Bit-Serial Systolic Array for Division Over GF(2^m)". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'03*, II:252–255, May 2003.

[55] Y-T. Horng and S-W. Wei. "Fast Inverters and Dividers for Finite Field GF(2^m)". *The 1994 IEEE Asia-Pacific Conference on Circuits and Systems, APCCAS '94*, pages 206–211, 1994.

[56] C-T. Huang and C-W. Wu. "High-Speed C-Testable Systolic Array Design for Galois-field Inversion". *Proceedings of the European Design and Test Conference*, Paris:342–346, March 1997.

[57] T. Itoh and S. Tsujii. "A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Basis". *Information and Computation*, 78:171–177, 1988.

[58] S.K. Jain, L. Song, and K.K. Parhi. "Efficient Semi-Systolic Architectures for Finite fields Arithmetics". *IEEE Transactions on VLSI Systems*, 6(1):101–113, Mar. 1998.

[59] S. Janssens, J. Thomas, W. Borremans, P. Gijsels, I. Verbauwhede, F. Vercauteren, B. Preneel, and J. Vandewalle. "Hardware/software co-design of an elliptic curve public-key cryptosystem". *2001 IEEE Workshop on Signal Processing Systems, SPS 2001*, pages 209–216, 2001.

[60] T. Kerins, E. Popovici, W. Marnane, and P.Fitzpatrick. "Fully Parameterizable Elliptic Curve Cryptography Processor over GF(2^m)". *12th International Conference on Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream, FPL 2002, LNCS 2438, Springer-Verlag*, pages 750–759, 2002.

[61] H.-S. Kim, K.-J. Lee, J. Kim, and K.-Y. Yoo. "Partitioned Systolic Multiplier for GF(2^m)". *1999 International Workshops on Parallel Processing, ICCP '99*, pages 192–197, Sept. 1999.

[62] N.-Y. Kim, W.-H. Lee, and K.-Y. Yoo. "Efficient Power-Sum Systolic Architectures for Public-Key Cryptosystems in GF(2^m)". *8th International Conference on Computing and Combinatorics, COCOON 2002, LNCS 2387, Springer-Verlag*, pages 153–161, August 2002.

[63] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1973. sec. 3.2.

[64] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1973. Section. 4.5.2, 4.5.3.

[65] N. Koblitz. "Hyperelliptic Cryptosystems". *Journal of Crptology*, 1(3):139–150, 1989.

[66] C.K. Koc and B.Sunar. "Low Complexity Bit Parallel Canonical and Normal Basis Multipliers for a Class of Finite Fields". *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[67] C.K. Koc and C.Y. Hung. "Bit-Level Systolic Arrays for Modular Multiplication". *Journal of VLSI Signal Processing*, 3:210–223, 1991.

[68] M. Kovac, N. Ranganathan, and M. Varanasi. "SIGMA: A VLSI Systolic Array Implementation of a Galois Field GF(2^m) Based Multiplication and Division Algorithm". *IEEE Transactions on VLSI Systems*, 1(1):22–30, Mar. 1993.

[69] H.T. Kung. "Why Systolic Architectures?". *Computer*, 15(1):37–46, Jan. 1982.

[70] S.Y. Kung. *VLSI Array Processor*. Prentice Hall Information and System Sciences Series, 1988.

[71] A.K. Lenstra and E.R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 14(4):255–293, 2001.

[72] K.H. Leung, K.W. Ma, W.K. Wong, and P.H.W. Leong. "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor". *8th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2000*, pages 68–76, 2000.

[73] P.H.W. Leung and I.K.H. Leong. "A Microcoded Elliptic Curve Processor using FPGA Technology". *IEEE Transactions on VLSI Systems*, 10(5):550–559, Oct 2002.

[74] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, 1994.

[75] E.D. Mastrovito. *VLSI Architectures for computation in Galois Fields*. PhD thesis, Linkoping Univ., Linkoping Sweden, 1991.

[76] J.V. McCanny, R.A. Evans, and J.G. McWhirter. "Use of Unidirectional Data Flow in Bit-level Systolic Array Chips". *Electronics Letters*, 22:540–541, May 1986.

[77] J.H. McClellan and C.M. Rader. *Number Theory in Digital Signal Processing*. Prentice-Hall, Inc., Englewood Cliffs, 1979.

[78] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

[79] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley Publishing Company, 1980.

[80] M.C. Mekhallati, M.K. Ibrahim, and A.S. Ashur. "New Low Complexity Bidirectional Systolic Structures for Serial Multiplication over the Finite Field GF(q^m)". *IEE Proceedings. Circuits, Devices and Systems Series*, 145(1):55–60, Feb. 1998.

[81] A.J. Menezes, I.F. Blake, X.-H. Gao, R.C. Mullin, S.A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields*. SECS199. Kluwer International Series in Engineering and Computer Science. Communications and Information Theory, 1993.

[82] A.J. Menezes, P. van Ooreschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[83] V. Miller. "Use of Elliptic Curves in Cryptography". *Advances in Cryptology, CRYPTO '85, LNCS 218, Springer-Verlag*, pages 417–426, 1985.

[84] M. Morii, M. Kasahara, and D.L. Whiting. "Efficient Bit-Serial Multiplication and the Discrete-Time Wiener-Hopf Equation over Finite Fields". *IEEE Transactions on Information Theory*, 35(6):1177–1183, Nov. 1989.

[85] R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson. "Optimal Normal Bases in GF(p^n)". *Discrete Applied Mathematics*, 22:149–161, 1988-89.

[86] NIST. FIPS 186-2, Digital Signature Standard (DSS). `http://csrc.nist.gov/publications/fips/fips186-2-change1.pdf`, Feb 2000. Specification of the Digital Signature Algorithm (DSA).

[87] G.H. Norton. "Precise Analyses of The Right- and Left-Shift Greatest Common Divisor Algorithms For GF(q)[x]". *SIAM Journal on Computing*, 18(3):608–623, June 1989.

[88] S. Okada, N. Torii, K. Itoh, and M. Takenada. "A high-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)". *2nd International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, LNCS 1965, Springer-Verlag*, pages 25–40, August 2000.

[89] J. Omura and J. Massey. "Computational Method and Apparatus for Finite Field Arithmetic". *U.S. Patent Number 4,587,627*, May 1986.

[90] G. Orlando and C. Paar. "A high-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)". *2nd International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000, LNCS 1965, Springer-Verlag*, pages 41–56, August 2000.

[91] Christof Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, University of Essen, 1994.

[92] D.A. Pucknell and K. Eshraghian. *Basic VLSI Design*. Prentice Hall, third edition, 1994.

[93] P. Quinton and Y. Robert. *Systolic Algorithms & Architectures*. Prentice-Hall, Inc., 1990.

[94] T.R.N. Rao and E. Fujiwara. *Error Correcting Codes for Computer Systems*. Prentice-Hall, Inc., Englewood Cliffs, 1989.

[95] R.L. Rivest, A. Shamir, and L. Adleman. "A Method of Obtaining Digital Signature and Public-key Cryptosystems". *Communication of the ACM*, 21(2):120–126, Feb. 1978.

[96] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. "Fast Key Exchange with Elliptic Curve Systems". *Advances in Cryptology, EUROCRYPT '95, LNCS 921, Springer-Verlag*, pages 43–56, 1995.

[97] P.A. Scott, S.E. Tavares, and L.E. Peppard. "A Fast VLSI Multiplier for GF(2^m)". *IEEE J. Selected Areas of Communications*, 4(1):62–66, Jan. 1986.

[98] G. Seroussi. "Table of Low-weight Irreducible Polynomials over F_2". *Hewlett-Packard Laboratories Technical Report No. HPL-98-135*, August 1998.

[99] L. Song and K.K. Parhi. "Low-Energy Digit-Serial/Parallel Finite Field Multipliers". *Journal of VLSI Signal Processing.*, 19:149–166, 1998.

[100] Y. Sugiyama. "An Algorithm for Solving Discrete-Time Wiener-Hopf Equations based on Euclid's Algorithm". *IEEE Transactions on Information Theory*, 32(5):394–409, May 1986.

[101] B. Sunar and C.K. Koc. "Mastrovito Multiplier for All Trinomials". *IEEE Transactions on Computers*, 48(7):522–527, July 1999.

[102] S. Sutikno, R. Effendi, and A. Surya. "Design and implementation of arithmetic processor $F_{2^{155}}$ for elliptic curve cryptosystems". *The 1998 IEEE Asia-Pacific Conference on Circuits and Systems, APCCAS '98*, pages 647–650, 1998.

[103] N. Takagi. "A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm". *Transactions of The Institute of Electronics, Information and Communication Engineers, 1998. IEICE'98*, E81(5):724–728, 1998.

[104] W.-C. Tsai and S.-J. Wang. "A systolic architecture for elliptic curve cryptosystems". *5th International Conference on Signal Processing, WCCC-ICSP 2000*, 1:591–597, 2000.

[105] L.B. Veries, K.A. Imink, J.G. Nibor, H. Hoeve, T. Doi, K. Okada, and H. Ogawa. "The Compact Disc Digital Audio System - Modulation and Error Correction". *Procedings Sixy-Seventh AES Convention*, Oct. 1980.

[106] 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*, 34(8):709–716, Aug. 1985.

[107] C.-L. Wang and J.L. Lin. "Systolic Array Implementation of Multipliers for Finite Fields GF(2^m)". *IEEE Transactions on Circuits and Systems*, 38(7):796–800, July 1991.

[108] C.-L. Wang and J.L. Lin. "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields GF(2^m)". *IEEE Transactions on Computers*, 42(9):1141–1146, Sept. 1993.

[109] M.Z. Wang and I.F. Blake. "Bit-Serial Multiplication in Finite Fields". *SIAM Journal on Discrete Mathematics.*, 3(1):140–148, Feb. 1990.

[110] Y. Watanabe, N. Takagi, and K. Takagi. "A VLSI Algorithm for Division in GF(2^m) Based on Extended Binary GCD Algorithm". *Transactions of The Institute of Electronics, Information and Communication Engineers, 2002. IEICE'02*, E85(5):994–999, 2002.

[111] S.-W. Wei. "VLSI Architectures for Computing Exponentiations, Multiplicative Inverses and Divisions in GF(2^m)". *IEEE International Symposium Circuits and Systems*, pages 4.203–4.206, 1994.

[112] Certicom ECC Whitepapers. "Remarks on the Security of the Elliptic Curve Cryptosystem". *URL: www.comms.scitech.susx.ac.uk/fft/crypto/EccWhite3.pdf*, Sept. 1997.

[113] E. De Win, A. Bosselaers, S. Vandenberghe, P. DeGersem, and J. Vandewalle. "A Fast Software Implementation for Arithmetic Operations in GF(2^n)". *Advances in Cryptology, ASIACRYPT '96, LNCS 1163, Springer-Verlag*, pages 65–76, Nov. 1996.

[114] J.J. Wozniak. "Systolic Dual Basis Serial Multiplier,". *IEE Proceedings. Computers and Digital Techniques*, 145(3):237–241, May 1998.

[115] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. "Systolic VLSI Realization of a Novel Iterative Division Algorithm over GF(2^m): A High Speed Low-Complexity Design". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'01*, 4:33–36, 2001.

[116] C-H. Wu, C-M. Wu, M-D. Shieh, and Y-T. Hwang. "An Area-Efficient Systolic Division Circuit over GF(2^m) for Secure Communication". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, 5:733–736, 2002.

[117] C.-W.. Wu and M.-K. Chang. "Bit-Level Systolic Arrays for Finite-Field Multiplications". *Journal of VLSI Signal Processing*, 10(1):85–92, June 1995.

[118] Huapeng Wu. *Efficient Computations in Finite Fields with Cryptographic Significance*. PhD thesis, University of Waterloo, Nov. 1998.

[119] Z. Yan and D.V. Sarwate. "Systolic Architectures for Finite Field Inversion and Division". *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS'02*, V:789–792, June 2002.

[120] C.-S. Yeh, I.S. Reed, and T.K. Truong. "Systolic Multipliers for Finite Fields GF(2^m)". *IEEE Transactions on Computers*, 33(4):357–359, April 1984.

[121] T. Zhang and K.K. Parhi. "Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible Polynomials'. *IEEE Transactions on Computers*, 50(7):734–749, July 2001.