Arithmetic Architectures for Finite Fields $GF(p^m)$ with Cryptographic Applications

DISSERTATION

zur Erlangung des Grades eines
Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

von

Jorge Guajardo Merchan
aus Caracas, Venezuela

Bochum, 2004

 The author can be contacted at `guajardo@alum.wpi.edu`

Dissertation submitted on:	May 20th, 2004
Dissertation defended on:	July 16th, 2004
Dissertation Advisor:	Prof. Dr.-Ing. Christof Paar
Dissertation Reader:	Prof. Dr. Colin D. Walter

To the memory of my father,
Jorge Guajardo Plantamura (1929–1997),
To my mother and sister who always have supported me,
and to my loves, Kathy and Matteo.

Dissertation Abstract

Finite fields are essential building blocks of many cryptographic schemes. Traditionally, cryptographic applications developed on hardware have tried to take advantage of the ease of implementation of fields of the form $GF(2^n)$ to reduce costs and increase performance. In recent years, however, there has been a renewed interest on the implementation of cryptographic systems based on odd characteristic finite fields $GF(p^m)$, p a prime, which have found applications in areas such as elliptic curve cryptography, identity-based encryption, and short signature schemes, to name a few. There are several reasons why these fields have become attractive. First, they provide for greater diversity at the time of implementation and this is directed related to security. For example, certain attacks which have been shown to be successful against elliptic curves defined over composite binary fields $GF((2^n)^m)$ may not carry over to elliptic curves defined over $GF((p^n)^m)$ where p is an odd prime. Thus, by considering alternative implementation options, we are, in a way, safeguarding against future attacks. Second, and perhaps more appealing to the practitioner, in certain cases fields of odd characteristic offer advantages, such as shorter signature sizes, which simply can not be achieved with fields of characteristic two. Thus, the need to provide hardware architectures for their efficient implementation. We tackle this problem in this thesis. In particular, we focus on the implementation of hardware architectures for addition, multiplication, and inversion in fields $GF(p^m)$.

The first part of the dissertation surveys previous architectures used to implement addition and multiplication over $GF(p)$ as such operations are the basic building blocks used to implement $GF(p^m)$ multipliers. We make particular emphasis on architectures for *small* $GF(p)$ fields where $p < 32$. At the end of this section, we propose a new method to design $GF(p)$ multipliers which can achieve up to a 30% improvement over previous architectures. For completeness, we also survey previous architectures for large $GF(p)$ fields such as those used in DL-based and RSA-based systems.

The second part of this thesis is concerned with multiplier architectures for fields $GF(p^m)$. We generalize architectures originally proposed for fields $GF(2^n)$ to the odd characteristic case. Both Least Significant Digit (LSD) multiplier architectures and Most Significant Digit (MSD) architectures are introduced and their time and area complexities compared. We implemented an arithmetic unit for $GF(3^m)$ fields on an FPGA and compared its performance to previous implementations and to our theoretical complexity models which agree with our practical results. In addition, we provide a thorough treatment of the cubing operation in fields of characteristic three and propose irreducible polynomials which reduce both the complexity of the multiplier and the cubing unit. In the appendix, we provide exhaustive lists of irreducible trinomials over $GF(3)$. Although LSD and MSD multiplier architectures allow the designer to trade area and performance according to his/her needs, these architectures suffer from several drawbacks: they use global signals and they are not very regular, thus not very suitable for VLSI systems. As a result, we developed systolic designs for $GF(p^m)$ fields based on our previously proposed LSD multipliers. In addition, for fixed p, we incorporate the notion of scalability, which has been extensively studied in the context of $GF(2^n)$ and $GF(p)$ based systems. Here by scalability we mean the ability to process fields $GF(p^m)$, for constant p and different values of m without recurring to changing the hardware or to reconfigurability, as in the case of FPGAs. We implemented the basic cell of an LSD-based systolic multiplier on 0.18μm CMOS technology and provided time and area complexities.

Finally, we tackle the problem of inversion in fields $GF(q^m)$, $q = p^n$, by giving a generalization of the Itoh and Tsujii inversion algorithm to fields of odd characteristic and a standard basis representation. We introduce families of irreducible polynomials which reduce the complexity of exponentiating to the

q-th power where $q = p^n$ and p is the field characteristic. By reducing the complexity of this operation, we also reduce the overall time required to compute an inverse in $GF(q^m)$.

Kurzdarstellung der Dissertation

Endliche Körper sind ein wesentlicher Bestandteil kryptographischer Verfahren. Konventionelle kryptographische Anwendungen in Hardware benutzen Binärkörper $GF(2^n)$ um den Rechenaufwand zu reduzieren und die Rechengeschwindigkeit zu steigern. In den vergangenen Jahren haben jedoch endlichen Körper ungerader Charakteristik $GF(p^m)$, wobei p eine Primzahl ist, stark an Bedeutung gewonnen. Einsatzgebiete dieser Körper sind beispielsweise kryptographische Verfahren basierend auf elliptischen Kurven, identitätsbezogene Verschlüsselung sowie kurze digitale Signaturen. Die Gründe für das Interesse an diesen Körpern sind vielseitig. Einerseits stellen sie einen wichtigen zusätzlichen Systemparameter dar, welcher sich direkt auf die kryptographische Sicherheit auswirkt. Beispielsweise sind bestimmte Angriffe gegen kryptographische Verfahren basierend auf elliptischen Kurven, welche über Binärkörpern definiert sind, über Erweiterungskörpern mit ungerader Charakteristik möglicherweise nicht erfolgreich. Andererseits weisen diese Körper in bestimmten Fällen praktische Vorteile gegenüber Binärkörpern auf. So können z.B. basierend auf diesen Körpern kürzere digitale Signaturen erzeugt werden. Um eine effiziente Implementierung zu gewährleisten, ist die Entwicklung geeigneter Hardwarearchitekturen unabdingbar. Diese Dissertation beschäftigt sich mit dem Entwurf solcher Architekturen. Insbesondere werden Implementierungen von Hardwarearchitekturen für Addition, Multiplikation und Inversion in Körpern $GF(p^m)$ entwickelt.

Der erste Teil der Dissertation untersucht bereits bekannte Architekturen für Addition und Multiplikation in Körpern $GF(p)$, welche als grundlegende Funktionsbausteine für die Implementierung von Multiplizierern in $GF(p^m)$ dienen. Insbesondere stehen Architekturen für *kleine* Körper $GF(p)$ mit $p < 32$, im Mittelpunkt. Am Ende dieses Abschnitts wird eine neue Methode für den Entwurf von Multiplizierern vorgestellt, welche bis zu 30% effizienter als bisherige Multiplizierer sind. Die Übersicht umfasst auch Architekturen für große Körper $GF(p)$, wie sie in DL-basierten und RSA-basierten Systemen eingesetzt werden.

Der zweite Teil der Arbeit behandelt Architekturen für Multiplizierer für Körper $GF(p^m)$. Die für die Multiplikation in Körpern $GF(2^n)$ vorgeschlagenen Architekturen werden für den Fall ungerader Charakteristik verallgemeinert. Es werden die Strukturen der LSD- (Least Significant Digit) und der MSD- (Most Significant Digit) Multiplizierer vorgestellt und deren Aufwand bezüglich Fläche und Laufzeit verglichen. Schließlich wird die Realisierung einer Arithmetikeinheit für Körper $GF(3^m)$ auf einem FPGA beschreiben und deren Merkmale mit bekannten Implementierungen verglichen. Bei dem Vergleich mit dem theoretisch hergeleiteten Komplexitätsmodell ist eine grundlegende Übereinstimmung zu verzeichnen. Des Weiteren wird eine sorgfältige Analyse von Cubing-Einheiten in Körpern der Charakteristik Drei durchgeführt und irreduzibles Polynome vorgestellt, welche sowohl die Komplexität des Multiplizierers als auch die der Cubing-Einheit reduzieren. Obwohl die Architekturen von LSD- und MSD-Multiplizierern dem Anwender die Wahl der Fläche und Durchsatzrate erlauben, leiden diese Architekturen an Einschränkungen: Es werden globale und unregelmäßige Verbindungen verwendet, welche für VLSI-Implementierungen nicht wünschenswert sind. Aus diesem Grund wird eine systolische Architektur für Körper $GF(p^m)$ basierend auf den existierenden Ansätzen für LSD-Multiplizierer vorgestellt. Für festes p wird zusätzlich noch der Begriff der Skalierbarkeit berücksichtigt, welcher im Kontext von $GF(2^n)$ und $GF(p)$ basierten Architekturen in der Literatur detailliert untersucht wurde. Mit Skalierbarkeit ist hier die Möglichkeit gemeint, in Körpern $GF(p^m)$ mit verschiedenen Werten m zu rechnen, ohne Änderungen an der Hardware oder, im Fall von FPGAs, Rekonfigurationen vornehmen zu müssen. Die grundlegende Struktur eines LSD-basierten systolischen Multiplizierers wurde für 0.18μm CMOS Technologie entworfen und dessen Zeit- und Flächenaufwand dargestellt.

Schließlich wird das Problem der Inversion in Körpern $GF(q^m)$, $q = p^n$, durch eine Verallgemeinerung des Inversions-Algorithmus von Itoh und Tsujii für Körper allgemeiner Charakteristik und Polynomialbasen-Repräsentation gelöst. Es werden Familien von irreduziblen Polynomen zur Reduktion des Aufwands von Exponentiationen mit Potenzen $q = p^n$ vorgestellt, wobei p die Charakteristik des Körpers ist. Durch die Reduktion der Komplexität dieser Operation wird auch die gesamte benötigte Zeit zur Berechnung einer Inversen in $GF(q^m)$ verringert.

Contents

List of Tables

List of Figures

List of Algorithms

Preface

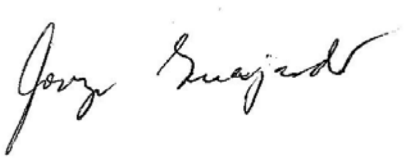
This thesis describes much of the work that I conducted while completing my Ph.D. degree at the Ruhr-Universität Bochum. I have attempted to make the treatment of architectures for finite fields of odd characteristic as complete and self-contained as possible. It is my hope that my contributions to the field and the extensive bibliographic material will make this thesis into a useful reference work for academia and industry professionals alike.

I would like to thank the following people for their support throughout the months that I worked on this thesis. First and foremost, I would like to thank my advisor Prof. Christof Paar for all his help, patience, and support throughout my graduate work at the RUB and at WPI. This work would certainly not have been possible without his help and guidance. I would also like to thank Prof. Paar for two other reasons: first for his friendship, camaraderie, and advice during difficult times in my life and, second, for introducing me, back in 1995, to the exciting field of applied cryptography and its engineering aspects, both of which I love and feel passionate about. I am also grateful to Prof. Colin Walter for taking the time to read this manuscript and for his valuable suggestions, comments, and constructive feedback, all of which helped improve the presentation of this thesis. Any errors that remain, of course, are entirely my own.

I would like thank Guido Bertoni, Gerardo Orlando, André Weimerskirch, and Thomas Wollinger for our fruitful discussions and friendship. Much of the work in this thesis was joint work with Guido and Gerardo. I learned a lot from their insights into what is useful and efficient in the hardware world. I must say that working in two different continents and three different geographic locations was definitely a challenge but also a lot of fun. I would also like to thank the members of the COSY Lab at the time. In alphabetical order they are: Marcus Heitmann, Irmgard Kühn, Sandeep Kumar, Jan Pelzl, Kai Schramm, André Weimerskirch, and Thomas Wollinger. Thanks for making the COSY Lab such a fun place to work, for your patience and understanding when I talked in my broken German, for your insights into the German culture, for your friendship, and for the many wonderful times we had together.

I must also thank Ari Juels for giving me the chance to work at RSA Laboratories in the summer of 2001. Working at the Labs was a lot of fun. Thanks for always being willing to listen to and answer my questions, for giving me the chance to learn about security protocols, and for believing in me. I am also grateful to Prof. J. E. Stine for providing me with the delay and area data for his standard cell library. My most sincere thanks to Jean-Pierre Seifert for his support and encouragement during the time of writing my thesis at Infineon. Had it not been for him, I might still be writing. Finally, I would like to give special thanks to Kathy Kreitner Guajardo, my loving wife, for proof reading my English in the manuscript that you are about to read and for her unconditional support and love.

To all of you my most sincere thanks!



CHAPTER 1

Introduction

Before 1976, Galois fields and their hardware implementation received considerable attention because of their applications in coding theory and the implementation of error correcting codes. In 1976, Diffie and Hellman [DH76] invented public-key cryptography[1] and single-handedly revolutionized a field which, until then, had been the domain of intelligence agencies and secret government organizations. In addition to solving the key management problem and allowing for digital signatures, public-key cryptography provided a major application area for finite fields. In particular, the Diffie-Hellman key exchanged is based on the difficulty of the Discrete Logarithm (DL) problem in finite fields. It is apparent, however, that most of the work on arithmetic architectures for finite fields only appeared after the introduction of two public-key cryptosystems based on finite fields: elliptic curve cryptosystems, introduced by Miller and Koblitz [Mil86, Kob87], and hyperelliptic cryptosystems, a generalization of elliptic curves introduced by Koblitz in [Kob89].

Both, prime fields and extension fields, have been proposed for use in such cryptographic systems but until a few years ago the focus was mainly on fields of characteristic 2. This is due to two main reasons. First, even characteristic fields naturally offer a straight forward manner in which field elements can be represented. In particular, elements of $GF(2)$ can be represented by the logical values "0" and "1" and thus, elements of $GF(2^n)$ can be represented as vectors of zeros and ones. For these types of fields, both software implementations (see [HMV92, SOOS95, DBV$^+$96, HHM00, LD00]) and hardware architectures (see [YP82, Mas89, HWB92, AMV93, ABMV93, FBT96, SP98, PFR99]) have been extensively studied. Second, until 1997 applications of fields $GF(p^m)$ for odd p were scarce in the literature.

In 1997, Mihăilescu [Mih97] and independently Bailey and Paar [BP98, BP01a] introduced the concept of Optimal Extension Fields (OEFs) in the context of elliptic curve cryptography. OEFs are fields $GF(p^m)$ where p is odd and both p and m are chosen to fit the particular hardware platform where the cryptosystem is being implemented. A major observation in [BP98, BP01a] is that matching of field parameters to hardware platform allows for optimized field arithmetic and an overall efficient implementation. Notice that the treatment in [Mih97, BP98, BP01a] and that of other works based on OEFs [Kob00, WBP00, Mül01] has only been concerned with efficient software implementations. Fields $GF(p^m)$ have also been proposed for cryptographic applications in [Kob98, Sma99]. In particular, [Kob98] describes an implementation of ECDSA over fields of characteristic 3 and 7. The author in [Sma99] describes a method to implement elliptic curve cryptosystems over fields of *small* odd characteristic, only considering $p < 24$ in the results section.

More recently, Boneh and Franklin [BF01] introduced a practical identity-based encryption scheme which is based on the application of the Weil and Tate pairings. Similarly, [BBS01] described a short signature scheme based on the Weil and Tate pairings (see [BB04a, BB04b] for the corresponding schemes not based on the random oracle model). Other applications include [Jou00, Ver01]. All of these applications of the Weil and Tate pairings consider elliptic curves defined over fields of characteristic 2 and 3. Because characteristic 2 field arithmetic has been extensively studied in the literature, authors have concentrated their efforts to improve the performance of systems based on characteristic 3 arithmetic[2]. For example, [BKLS02, GHS02a] describe algorithms to improve the efficiency of the pairing computations. In addition, [GHS02a] introduces some clever tricks to improve the efficiency of the underlying arithmetic in *software* based solutions.

Although, there have been several dissertations dealing with the problem of finite field arithmetic over $GF(2)$ and their hardware implementation, (see for example [Mas91, Has92, Gei93, Paa94, Wu98, Olo02]), to our knowledge there has not been a systematic treatment of finite field arithmetic in fields $GF(p^m)$ where p is odd and $m > 1$, and, in fact, very little work in general. Thus, this thesis focuses on the development of techniques to implement addition, multiplication, and inversion in fields $GF(p^m)$ for odd p and $m > 1$. We end this section by emphasizing that the efficient implementation of finite field arithmetic, whether in hardware or software, is key to the performance of the overall system and, in fact, will dictate the final performance of the system. A poor finite field implementation will result in bad system performance and vice versa. Thus, the importance of studying techniques for the efficient implementation of finite fields.

1.1 Hardware Complexity Considerations

As in the case of characteristic two finite field hardware architectures, we can also classify architectures for fields $GF(p^m)$, where $p > 2$, according to the way the finite field elements are processed as: array (also known as serial), digit, or parallel multipliers [SP98]. In this thesis, we consider only array and digit architectures for multipliers in $GF(p^m)$ which include both combinatorial and memory elements such as registers. Parallel architectures and, in particular, parallel multipliers would require immense hardware resources for cryptographic applications and, thus, they do not seem realistic in this context. Notice, however, that in building architectures for fields $GF(p^m)$, it is first required to implement arithmetic in $GF(p)$. Thus, we also consider the complexity of adders and multipliers for arithmetic in $GF(p)$, where p is small (we will make our definition of small more exact in Chapter 3). For such adders and multipliers, we consider parallel architectures.

In evaluating hardware architectures, there are several factors which need to be considered, including but not limited to:

- Space complexity (chip area)
- Time complexity (circuit performance or delay)
- Power dissipation
- Architecture regularity and modularity

Traditionally, both the area and time complexities have been the most important criteria to evaluate and compare hardware architectures for finite field arithmetic[3] (see for example [Paa94, Wu98, Olo02]). We will use primarily theoretical VLSI time and area complexities to evaluate and compare the architectures studied and proposed in this thesis. As mentioned previously, two different types of architectures are studied in this thesis: architectures to implement arithmetic in small $GF(p)$ fields and architectures to implement arithmetic in $GF(p^m)$ fields. For the first case, we have chosen to measure the space and time complexities according to:

1. The number of inverters as well as 2-input and 3-input AND, OR, and XOR gates and their corresponding delays.

2. In terms of normalized gate area and delay, where we have normalized with respect to the area and delay of a 2-input NAND gate.

The first measure has been widely used in other works where arithmetic architectures for characteristic two fields are studied [Paa94, Wu98] but limited to 2-input XOR and AND gates as it is possible to implement $GF(2)$ arithmetic using only these two types of gates. We extend the model to include OR gates and 3-input gates because these are used regularly throughout the Residue Number System (RNS) literature (for example in[CPO95, PKS01]) to estimate the complexity of $GF(p)$ adders and multipliers. Such adders and multipliers are essential building blocks of $GF(p^m)$ multipliers. In addition, by giving the area of a circuit in terms of the number of gates, we remain somewhat *technology independent*. In other words, anyone can take the number of gates required to implement a $GF(p)$ multiplier, for fixed p, and estimate the total area used given a standard cell library and CMOS technology. Notice, however, that the above measure does not allow us to perform comparisons among different designs. For example, consider a circuit which requires five 2-input XOR gates, three 2-input AND gates and six 2-input OR gates versus a circuit which requires three 2-input XOR gates and fifteen 2-input OR gates. How can we establish which circuit has the largest area? We simply can not. We need to map our gates to transistors, to equivalent gates, to their size in μm^2, or to other similar measure which allows us to compare area (time delay) in a more exact manner. We chose our measure to be the normalized area of all components with respect to the size and delay of a 2-input NAND gate. This is also known as the equivalent gate measure. Table 1.1 summarizes the assumed normalized area and delay characteristics of all basic components used in the architectures presented in this thesis.

Table 1.1. Normalized time and area complexities of basic building blocks

Component	Abbreviation	Normalized Area	Normalized Delay
Inverter	NOT	0.7	1.0
2-input AND gate	AND2	1.3	1.0
2-input NAND gate	NAND2	1.0	1.0
3-input AND gate	AND3	2.0	1.1
3-input NAND gate	NAND3	1.5	1.1
2-input OR gate	OR2	1.3	0.8
2-input NOR gate	NOR2	1.0	1.0
3-input OR gate	OR3	2.0	1.1
3-input NOR gate	NOR3	1.5	1.1
2-input XOR gate	XOR2	2.3	1.0
2-input XNOR gate	XNOR2	2.3	1.0
Complex gate implementing $\overline{((A \wedge B) \vee C)}$	AO21	1.3	1.0
Complex gate implementing $\overline{((A \wedge B) \vee (C \wedge D))}$	AO22	1.7	1.0
Complex gate implementing $\overline{((A \vee B) \wedge C)}$	OA21I	1.0	1.0
Complex gate implementing $\overline{((A \vee B) \wedge (C \vee D))}$	OA22I	1.7	1.0
D Flip-Flop	FF	4.0	1.0
Latch	LAT	2.0	1.0
1-bit 2:1 Multiplexer	MUX21	2.0	1.0
1-bit Full Adder	FA	5.0	1.1
1-bit Half Adder	HA	2.2	1.0
$2^n \times W$-bit ROM table	$\mathrm{ROM}_{n,W}$	$(2^n \times W)\cdot$ OR2	$n\cdot T_{\mathrm{FA}}$

The normalized area complexity provided in Table 1.1 has been obtained based on the normalized area characteristics of the components in the standard cell libraries from [GS03b, VLS03] which are summarized in Table D.2. We notice that the delays of most components in Table D.2 do not vary by more than 10% except for the OR2 and the FA cells. Thus, we assume that the delay of all components is the same except for the OR2, the 3-input gates, and FA cells. The delay of the OR2 gate is slightly better than other gates and the delay of the 3-input gates and FA cells is slightly worse. We notice that in our model, we assumed that 3-input gates require an area which is 1.5 times that of the corresponding 2-input gate. This is in agreement with the model of [PKS01]. We also assume that a latch requires about half the area of a flip-flop. For the delay and time complexities of ROM tables and 1-bit half adder cells, we have taken into consideration the VLSI complexity model used in [CPO95, PKS01][4]. Thus, in agreement with [PKS01], we have assumed that a HA is about 0.43 times as large as a FA. Similarly, we have adopted the ROM-complexity model from [CPO95] according to which a $2^n \times W$-bit table has an area complexity of $2^n \times W$ OR2 gates and a delay of n 1-bit FA. We hope that by considering *actual* gate sizes and delays our area and delay estimates will be more accurate.

For the $GF(p^m)$ architectures, we have chosen a single complexity measure: The number of $GF(p)$ multipliers and adders and their corresponding delays. We emphasize that the delay and area are given in terms of $GF(p)$ multipliers, adders, and, for certain designs, also registers. This measure provides us with technology and design independence. By design independence, we mean that it is very probable that in the future there will be a new and better $GF(p)$ adder or multiplier optimized according to certain measure which we might want to combine with the $GF(p^m)$ architectures proposed in this thesis. This measure, thus, allows an easy estimate of the area in terms of this future $GF(p)$ arithmetic design.

We end this section by noticing that, in some cases, we have also synthesized and simulated the proposed architectures on FPGAs. For these cases, we provide a theoretical framework to evaluate the area complexity of our FPGA designs. Our FPGA model is based on the complexity measures of [Orl02] and it is briefly described in Appendix C.

1.2 Summary of Research Contributions and Dissertation Outline

Given the research community's interest in cryptographic systems based on fields of odd characteristic and the lack of hardware architectures for general odd characteristic fields, we try to close this gap in this dissertation. We begin with an overview of the mathematics of Galois fields in Chapter 2. This introduction is meant to make the thesis self-contained and, thus, it provides the readers with facts on finite fields and construction of irreducible polynomials, all of them without proofs. We end Chapter 2 with references to other works which would provide in-depth treatment (and proofs) of the mathematical concepts here described. The remainder of the dissertation is organized into three main themes: adders and multipliers in $GF(p)$, multipliers in $GF(p^m)$, and inversion in $GF(p^m)$.

In Chapters 3 and 4, we thoroughly investigate adders and multipliers in $GF(p)$ for both small and large values of p. For small values of p, we consider architectures which have been presented in the context of digital signal processing (DSP) applications and, in particular, of residue number systems

(RNS). At the end of Chapter 3, we propose a new method to design $GF(p)$ multipliers which can achieve up to a 30% area improvement over previous architectures. For the sake of completeness, in Chapter 4 we survey multipliers for large values of *p* (between 160-bit and 2000-bit long primes) which have been proposed mainly in the context of RSA [RSA78], Discrete Logarithm (DL) systems [DH76], and, more recently, elliptic curves [Mil86, Kob87].

Chapters 5 and 6 investigate multipliers in $GF(p^m)$ and constitute the heart of this dissertation. In Chapter 5, we first generalize the work in [SP98] to fields $GF(p^m)$, p odd. In particular, we develop semi-systolic architectures for Most Significant Digit (MSD) and Least Significant Digit (LSD) multipliers. Second, we study the complexity of the architectures previously proposed for the particular case of $GF(3^m)$ and propose optimizations to the computation of the cubing operation in these fields. Fields of characteristic three are of interest for cryptographic applications such as identity-based encryption [BF01] and short signature schemes [BBS01]. As a result of the optimizations for characteristic three fields, we also provide tables of irreducible polynomials for which the complexity of the multipliers is reduced. We end this chapter by describing an implementation of an arithmetic unit used to perform operations in $GF(3^m)$ and compare it to a similar unit used to perform arithmetic over binary fields.

The methods described in Chapter 5 have the drawback of using global signals and long wires and they require reconfigurability to achieve their full potential. Thus, these solutions lack flexibility in platforms such as ASICs. In Chapter 6, we move a step forward towards the design of scalable and flexible hardware architectures for odd $GF(p^m)$ fields. In particular, we propose new systolic and scalable architectures for arithmetic in $GF(p^m)$ fields. On the one hand, the systolic nature of our architectures provides for ease of design and offers functional and layout modularity all of which are properties envisioned in good VLSI designs. On the other hand, with scalability, we are able to perform a multiplication for any value of m in $GF(p^m)$, with fixed p and fixed digit size. Thus, we can support multiple irreducible polynomials without turning to the use of reconfigurability in FPGAs, solving one of the major disadvantages of the architectures presented in Chapter 5. We end Chapter 6 with an implementation of the basic cell of the systolic architecture described in a 0.18 μm CMOS standard cell library.

In Chapter 7, we tackle the problem of inversion in fields $GF(q^m)$, $q = p^n$, by giving a generalization of the Itoh and Tsujii inversion algorithm to fields of odd characteristic and a standard basis representation. We introduce families of irreducible polynomials which reduce the complexity of exponentiating to the q-th power where $q = p^n$ and p is the field characteristic. By reducing the complexity of this operation, we also reduce the overall time required to compute an inverse in $GF(q^m)$.

1.3 Notes and Further References

1. The discovery of public-key cryptography in the intelligence community is attributed in [Ell97] to John H. Ellis in 1970. Reference [Ell97], attributes to John H. Ellis a theorem which proves the existence of a method to establish a secure communication channel between two parties who previously do not share a common secret-key. The discovery of the equivalent of the RSA cryptosystem [RSA78] is attributed to Clifford Cocks in 1973 while the equivalent of the Diffie-Hellman key exchange was discovered by Malcolm J. Williamson, in 1974. All of the authors worked at CESG at the time of their inventions and thus, their inventions were not in the public-domain at the time when Diffie and Hellman or Rivest, Shamir and Adleman published their results. In addition, it is believed (although the issue remains open) that these British scientists did not realize the practical implications of their discoveries at the time of their publication within CESG (see for example [Sch98, Dif99]).

2. The use of characteristic 3 fields is preferred in some applications due to the improved bandwidth requirements implied by the security parameters. For example, signatures resulting from pairing cryptography will be smaller in characteristic 3 than in characteristic 2.

3. Recently, power consumption has become a third important factor to evaluate the performance of hardware architectures, however, in this thesis power consumption will not be considered.

4. The standard cell library from [GS03b, VLS03] provides a 1-bit half adder cell but it is different from the definition that we use throughout this thesis. Thus, we have not consider it.

CHAPTER 2

Mathematical Background

A more appropriate name for this chapter would probably have been "A Short Introduction to Finite Fields", as we have tried to be as concise and thorough as possible. The aim was two fold. First, we wanted to make the thesis self-contained presenting all results in the theory of finite fields that were used to develop the architectures presented in this work. Second, we have found during the course of writing the present document that in the past couple of years there were several results on irreducible polynomials and their applications which were scattered throughout the literature and had not been included in previous overview articles. Thus, we wanted this chapter to be used as a reference which summarized all such results. We also hope that this chapter will be of interest not only to people working on arithmetic architectures for finite fields (the subject of the present thesis) but to a wider audience. We notice that finite fields are, by far, the most widely used algebraic structure in the construction of cryptographic schemes. Examples include: the Advanced Encryption Standard (AES) [U.S01], the Diffie-Hellman key exchange protocol [DH76] and those systems based on the difficulty of solving the Discrete Logarithm (DL) problem over finite fields [U.S00] and over elliptic curves [Mil86, Kob87] such as the Elliptic Curve Digital Signature Algorithm (ECDSA) [MJ99, U.S00]. We refer the reader to [LN97] for a comprehensive treatment of finite fields[1].

Definition 2.1. *Let S be a set. Then, the mapping from* $S \times S$ *to* S *is called a binary operation on* S*. In particular, a binary operation is a rule that assigns ordered pairs* (s, t)*, with* $s, t \in S$*, to an element of*

S. Notice that under this definition the image of the mapping is require to be also in S. This is known as the closure property.

2.1 Groups

Definition 2.2. *A group is a set G together with a binary operation $*$ on the set, such that the following properties are satisfied:*

(i) The group operation is associative. That is $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$, for all $\alpha, \beta, \gamma \in G$.

(ii) There is an element $\epsilon \in G$, called the identity element, such that $\epsilon * \alpha = \alpha * \epsilon = \alpha$ for all $\alpha \in G$.

(iii) For all $\alpha \in G$, there is an element $\alpha^{-1} \in G$, such that $\alpha * \alpha^{-1} = \alpha^{-1} * \alpha = \epsilon$. The element α^{-1} is called the inverse of α.

If the group also satisfies $\alpha * \beta = \beta * \alpha$ for all $\alpha, \beta \in G$, then the group is said to be *commutative* or *abelian*. In the remainder of this work, we will only consider abelian groups unless we explicitly say something to the contrary. Note that we have used a multiplicative group notation for the group operation. If the group operation is written additively, then we talk about an additive group, the identity element is often associated with the the zero (0) element, and the inverse element of α is written as $-\alpha$. Notation conventions are shown in Table 2.1.

Table 2.1. Notation for common group operations, where $\alpha \in G$ and n and m are integers.

Multiplicative Notation	Additive Notation
$\alpha^n = \alpha * \alpha * \cdots * \alpha$ (n factors of α)	$n\alpha = \alpha + \alpha + \cdots + \alpha$ (α added to itself n times)
$\alpha^{-n} = (\alpha^{-1})^n$	$-n\alpha = n(-\alpha)$
$\alpha^n * \alpha^m = \alpha^{n+m}$	$n\alpha + m\alpha = (n+m)\alpha$
$(\alpha^n)^m = \alpha^{nm}$	$n(m\alpha) = (nm)\alpha$

Example 2.1. (i) The set of integers $\mathbb{Z}$ forms an additive group with identity element 0.

(ii) The set of reals $\mathbb{R}$ forms a group under the addition operation with identity element 0 and under the multiplication operation with identity element 1.

(iii) The integers modulo m, denoted by $\mathbb{Z}_m$, form a group under addition modulo m with identity element 0. Notice that the group $\mathbb{Z}_m$ is, in general, not a group under multiplication modulo m, since not all its elements have multiplicative inverses.

Definition 2.3. *A group G is finite if the number of elements in it is finite, i.e., if its order, denoted $|G|$, is finite.*

Definition 2.4. *For $n \geq 1$, let $\phi(n)$ denote the number of integers in the range $[1, n]$ which are relatively prime (or co-prime) to n (i.e. an integer a is co-prime to n if $\gcd(a, n) = 1$). The function $\phi(n)$ is called the Euler phi function or the Euler totient function. The Euler phi function satisfies the following properties:*

(i) If p is prime then $\phi(p) = p - 1$.

(ii) The Euler phi function is multiplicative. In other words, if $\gcd(p, q) = 1$, then $\phi(pq) = \phi(p)\phi(q)$.

(iii) If $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, is the prime factorization of n, then $\phi(n)$ can be computed as:

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right)$$

Example 2.2. Let the set of non-zero integers modulo m which are co-prime to m be denoted by $\mathbb{Z}_m^*$. Then, the set $\mathbb{Z}_m^*$ under the operation of multiplication modulo m forms a group of order $\phi(m)$ with identity element 1. In particular, if m is prime then $\phi(m) = m - 1$.

Definition 2.5. *A group G is cyclic if there is an element $\alpha \in G$ such that for each $\beta \in G$, there is an integer i such that $\beta = \alpha^i$. Such an element is called a generator of G and we write $G =< \alpha >$. The order of $\beta \in G$ is defined to be the least positive integer t such that $\beta^t = \epsilon$, where ϵ is the identity element in G.*

Notice the difference between the order of an element $\beta \in G$, written $\mathrm{ord}(\beta)$, and the order of the group G, written $|G|$.

Example 2.3. (i) The multiplicative group of integers modulo 11, $\mathbb{Z}_{11}^*$, is a cyclic group with generators 2, $23 \equiv 8 \bmod 11$, $27 \equiv 7 \bmod 11$, and $29 \equiv 6 \bmod 11$. Notice that the powers of two which result in generators are co-prime to the order of $\mathbb{Z}_{11}^*$, i.e., 10. In fact, it can be shown that given a generator $\alpha \in \mathbb{Z}_m^*$, $\beta \equiv \alpha^i \bmod m$ is also a generator if and only if $\gcd(i, \phi(m)) = 1$.

(ii) The additive group of integers modulo 6, $\mathbb{Z}_6$, has generators 1 and 5.

2.2 Rings and Fields

Definition 2.6. *A ring, $(R, +, *)$, is a set R together with two binary operations on R, arbitrarily denoted $+$ (addition) and $*$ (multiplication), which satisfy the following properties:*

(i) $(R, +)$ is an abelian group with identity element denoted by 0.

(ii) The operation $*$ is associative, that is, $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$, for all $\alpha, \beta, \gamma \in R$.

(iii) There is a multiplicative identity element denoted by 1, with $0 \neq 1$, such that for all $\alpha \in R$, $\alpha * 1 = 1 * \alpha = \alpha$.

(iv) The operation $*$ is distributive over the $+$ operation. In other words, $\alpha*(\beta+\gamma) = (\alpha*\beta)+(\alpha*\gamma)$ and $(\beta + \gamma) * \alpha = (\beta * \alpha) + (\gamma * \alpha)$ for all $\alpha, \beta, \gamma \in R$.

If the operation $*$ is also commutative, i.e., $\alpha * \beta = \beta * \alpha$, then the ring is said to be commutative.

Example 2.4. (i) The set of integers $\mathbb{Z}$ with the usual addition and multiplication operations is a commutative ring. Similarly, the set of rational numbers $\mathbb{Q}$, the set of reals $\mathbb{R}$, and the complex numbers $\mathbb{C}$ are all examples of commutative rings with the usual addition and multiplication operations.

(ii) The set $\mathbb{Z}_m$ of residues modulo m with the addition and multiplication modulo m operations is a commutative ring.

Definition 2.7. *A field F is a commutative ring in which every non-zero element (i.e., all elements except for the additive identity element) have multiplicative inverses. A subset S of a field F which itself is a field with respect to operations in F is called a subfield of F. In this case F is said to be an extension field of S.*

Definition 2.7 implies that a field F is a set on which two binary operations are defined, called addition and multiplication, and which contains two elements, 0 and 1, which satisfy $0 \neq 1$. In particular, $(F, +, 0)$ is an abelian group with additive identity 0 and $(F^*, *, 1)$ is an abelian group under the multiplication operation with 1 as its multiplicative identity. The two operations of addition and multiplication are related to each other via the distributivity law, i.e., $\alpha * (\beta + \gamma) = (\alpha * \beta) + (\alpha * \gamma)$, and $(\beta + \gamma) * \alpha = \alpha * (\beta + \gamma) = (\beta * \alpha) + (\gamma * \alpha)$ follows automatically from the fact that $(F^*, *, 1)$ is an abelian group under multiplication.

Example 2.5. (i) The set of integers $\mathbb{Z}$ with the usual addition and multiplication operations is *not* a field since not all its elements have multiplicative inverses. In fact, only 1 and -1 have multiplicative inverses.

(ii) The set of rational numbers $\mathbb{Q}$, the set of reals $\mathbb{R}$, and the complex numbers $\mathbb{C}$ are all examples of fields.

(iii) The set $\mathbb{Z}_m$ of residues modulo m with the addition and multiplication modulo m operations is a field if and only if m is prime. For example, Z_2, Z_3, Z_5, etc., are all fields.

Definition 2.8. *The characteristic of a field is said to be 0 if* $\overbrace{1 + 1 + \cdots + 1}^{m \text{ times}}$ *is never equal to 0 for any value of* $m \geq 1$. *Otherwise, the characteristic of a field is the least positive integer* m *such that* $\sum_{i=0}^{m} 1 = 0$. *It can be shown that if the characteristic* m *of a field is not 0 then* m *is a prime.*

We end this section by noticing that $\mathbb{Z}_2, \mathbb{Z}_3, \mathbb{Z}_5, \ldots, \mathbb{Z}_p$, where p is prime, are fields of characteristic p. We notice in particular that they are fields with a finite number of elements and, thus, they have received the name of finite fields or Galois fields after its discoverer Evariste Galois, French mathematician of the 18th century. The number of elements in the field is called the *order* of the field. Finally, it is worth mentioning that $\mathbb{Z}_p$, for p prime, are just but a few of the existing finite fields. To provide constructions for other finite fields we introduce the concept of polynomial rings.

Example 2.6. (i) If p is prime then we can find the inverse of any integer a modulo p via Fermat's Little theorem which states that if $\gcd(a, p) = 1$, (this is always true if p is prime and $a < p$) then $a^{p-1} = 1 \bmod p$. It follows that a^{p-2} is the inverse of a modulo p.

(ii) The inverse of 3 modulo 7 ($3^{-1} \bmod 7$) can be found as $3^5 = 243 \equiv 5 \bmod 7$. A quick check verifies our assertion: $3 \cdot 5 = 15 \equiv 1 \bmod 7$.

(iii) A second way to find the inverse of an integer modulo p is to use the extended Euclidean algorithm which guarantees that we can find integers u and v such that $a \cdot v + p \cdot u = d = \gcd(a, p)$. It follows that if $\gcd(a, p) = 1$, we can find the inverse of $a \bmod p$ as $a \cdot v + p \cdot u = 1 \Rightarrow a \cdot v \equiv 1 \bmod p \Rightarrow a^{-1} \equiv v \bmod p$.

2.3 Polynomial Rings

Definition 2.9. *If R is a commutative ring, then a polynomial in the indeterminate x over R is an expression of the form: $A(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x_2 + a_1 x + a_0$ where each $a_i \in R$ and $n \geq 0$. As in classical algebra, the element a_i is called the coefficient of x_i in $A(x)$ and the largest n for which $a_n \neq 0$ is called the degree of $A(x)$, denoted by $\deg(A(x))$. The coefficient a_n is called the leading coefficient of $A(x)$. If $a_n = 1$ then $A(x)$ is said to be a monic polynomial. If $A(x) = a_0$ then the polynomial is a constant polynomial and has degree 0 whereas if $A(x) = 0$ (i.e. all coefficients of $A(x)$ are equal to 0), then $A(x)$ is called the zero polynomial and for mathematical convenience is said to have degree $-\infty$. Two polynomials $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{i=0}^{n} b_i x^i$ over R are said to be equal if and only if $a_i = b_i$ for $0 \leq i \leq n$.*

Example 2.7. (i) The sum of two polynomials is realized in the familiar way as:

$$A(x) + B(x) = \sum_{i=0}^{n} (a_i + b_i) x^i$$

(ii) The product of two polynomials $A(x) = \sum_{i=0}^{n} a_i x^i$ and $B(x) = \sum_{j=0}^{m} b_j x^j$ over R is defined as follows:

$$C(x) = A(x) \cdot B(x) = \sum_{k=0}^{n+m} c_k x^k$$

where

$$c_k = \sum_{\substack{i+j=k \\ 0 \leq i \leq n \\ 0 \leq j \leq m}} a_i b_j$$

and addition and multiplication of coefficients is performed in R. Together with the operations of addition and multiplication defined as above it is easily seen that the set of polynomials over R forms a ring.

Definition 2.10. *Let R be a commutative ring. Then the set of polynomials over R with addition and multiplication of polynomials defined as in Example 2.7 is called a polynomial ring and we denote it by $R[x]$.*

Notice the difference in notation between the set of all polynomials over R, together with the operations of addition and multiplication of polynomials, denoted by $R[x]$ (square brackets) and one element of $R[x]$, say $A(x)$, which we denote also with capital letters but round parenthesis. In the remainder of this work we will only consider polynomial rings $F[x]$ defined over F, where F is a field.

Elements of $F[x]$ share many properties with the integers. Thus, it is possible to talk about divisibility of a polynomial by other polynomial. In particular, a polynomial $B(x) \in F[x]$ is said to divide another polynomial $A(x) \in F[x]$ if there exists a polynomial $C(x) \in F[x]$ such that $A(x) = B(x) \cdot C(x)$. We say that $B(x)$ is a divisor of $A(x)$ or that $A(x)$ is a multiple of $B(x)$, or that $A(x)$ is divisible by $B(x)$. The idea of divisibility leads to a division algorithm for polynomials. In fact, we can prove that for any $B(x) \neq 0$ in $F[x]$, and for any $A(x) \in F[x]$, we can find polynomials $Q(x)$ and $R(x)$ such that $A(x) = Q(x) \cdot B(x) + R(x)$ where $\deg(R(x)) < \deg(B(x))$ and $Q(x)$ and $R(x)$ are unique.

Definition 2.11. *A polynomial $P(x) \in F[x]$ is said to be irreducible over F if $P(x)$ has positive degree and writing $P(x) = B(x) \cdot C(x)$ implies that either $B(x)$ or $C(x)$ is a constant polynomial. Otherwise $P(x)$ is said to be reducible.*

Much in the same way as with the integers, we say that if $A(x), B(x) \in F[x]$, then $A(x)$ is said to be congruent to $B(x)$ modulo $T(x)$ if $T(x)$ divides $A(x) - B(x)$, written $T(x)|(A(x) - B(x))$. The congruency relation is denoted as $A(x) \equiv B(x) \bmod T(x)$. For a fixed polynomial $T(x)$, the equivalence class of a polynomial $A(x) \in F[x]$ is the set of all polynomials in $F[x]$ congruent to

$A(x)$ modulo $T(x)$. It can be shown that the relation of congruency modulo $T(x)$ partitions $F[x]$ into equivalence classes. In particular, we can find a unique representative for each equivalence class as follows. From the division algorithm for polynomials we know that given any two polynomials $A(x)$ and $T(x)$ we can find unique polynomials $Q(x)$ and $R(x)$ where $\deg(R(x)) < \deg(T(x))$. Hence, every polynomial $A(x)$ is congruent modulo $T(x)$ to a unique polynomial $R(x)$ of degree less than $T(x)$. Now, we can choose the unique polynomial $R(x)$ to be the unique representative for the equivalence class of polynomials containing $A(x)$. We denote by $F[x]/(T(x))$ the set of equivalence classes of polynomials in $F[x]$ of degree less than $m = \deg(T(x))$. It turns out that $F[x]/(T(x))$ is a commutative ring and if $T(x)$ is irreducible over F, then $F[x]/(T(x))$ is a field.

Definition 2.12. *An element $\alpha \in F$, is said to be a root (or zero) of the polynomial $P(x) \in F[x]$ if $P(\alpha) = 0$.*

2.4 Construction of Finite Fields $GF(p^m)$

In previous sections, we saw how $\mathbb{Z}_p$, for p prime, was an example of a finite field (also called Galois field $GF(p)$ or $\mathbb{F}_p$) with p elements where addition and multiplication where the standard addition and multiplication modulo p operations and inversion could be achieved via Fermat's Little theorem or using the extended Euclidean algorithm for integers. In this section, we construct the remaining finite fields.

Definition 2.13. *Let m be a positive integer and $P(x)$ be an irreducible polynomial of degree m over $GF(p)$. Moreover, let α be a root of $P(x)$, i.e., $P(\alpha) = 0$. Then, the Galois field of order p^m and characteristic p, denoted $GF(p^m)$ or $\mathbb{F}_{p^m}$, is the set of polynomials $a_{m-1}\alpha^{m-1} + a_{m-2}\alpha^{m-2} + \cdots + a_2\alpha^2 + a_1\alpha + a_0$, with $a_i \in GF(p)$ together with the addition and multiplication operations defined as follows. Let $A(\alpha), B(\alpha), C(\alpha) \in GF(p^m)$, with $A(\alpha) = \sum_{i=0}^{m-1} a_i\alpha^i$, $B(\alpha) = \sum_{i=0}^{m-1} b_i\alpha^i$, and $C(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$, where $a_i, b_i, c_i \in GF(p)$ then:*

(i) $C(\alpha) = A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1}(a_i + b_i)\alpha^i$

(ii) Define $\overline{C}(\alpha)$ to be the result of multiplying $A(\alpha)$ by $B(\alpha)$ via standard polynomial multiplication as described in Example 2.7. Thus, $\overline{C}(\alpha)$ is a polynomial with $\deg(\overline{C}(\alpha)) \leq 2m - 1$. Then, we

define $C(\alpha)$ to be $\overline{C}(\alpha)$ modulo $P(\alpha)$, i.e., $C(\alpha) \equiv \overline{C}(\alpha) \bmod P(\alpha)$. Notice that $C(\alpha)$ can be found since the division algorithm guarantees that we can write $\overline{C}(\alpha)$ as $\overline{C}(\alpha) = P(\alpha)Q(\alpha) + C(\alpha)$ where $\deg(C(\alpha)) < m$.

Example 2.8. Let $p = 2$ and $P(x) = x^4 + x + 1$. Then, $P(x)$ is irreducible over $GF(2)$. Let α be a root of $P(x)$, i.e., $P(\alpha) = 0$, then the Galois field $GF(2^4)$ is defined by $GF(2^4) = \{a_3\alpha^3 + a_2\alpha^2 + a_1\alpha + a0 | a_i \in GF(2)\}$ together with addition and multiplication as defined in Definition 2.13. The field $GF(2^4)$ is of characteristic 2 and it has order $2^4 = 16$, in other words, it has 16 elements. The elements of $GF(2^4)$ can be written as shown in Table 2.2.

Table 2.2. Representation of $GF(2^4)$ elements.

As a 4-tuple	As a polynomial	As a power of α
0000	0	0
0001	1	$\alpha^{15} \equiv 1$
0010	α	α
0011	$\alpha + 1$	α^4
0100	α^2	α^2
0101	$\alpha^2 + 1$	α^8
0110	$\alpha^2 + \alpha$	α^5
0111	$\alpha 2 + \alpha + 1$	α^{10}
1000	α^3	α^3
1001	$\alpha^3 + 1$	α^{14}
1010	$\alpha^3 + \alpha$	α^9
1011	$\alpha^3 + \alpha + 1$	α^7
1100	$\alpha^3 + \alpha^2$	α^6
1101	$\alpha^3 + \alpha^2 + 1$	α^{13}
1110	$\alpha^3 + \alpha^2 + \alpha$	α^{11}
1111	$\alpha^3 + \alpha^2 + \alpha + 1$	α^{12}

To add $\alpha^3 + 1$ and $\alpha^3 + \alpha^2 + 1$ we simply perform polynomial addition and reduce the coefficients of the resulting polynomial modulo 2. Thus, $(\alpha^3 + 1) + (\alpha^3 + \alpha^2 + 1) \equiv \alpha^2$. Similarly, $(\alpha^3 + 1)$ multiplied by $(\alpha^3 + \alpha^2 + 1)$ is obtained as

$$(\alpha^3 + 1)(\alpha^3 + \alpha^2 + 1) = \alpha^6 + \alpha^5 + \alpha^3 + \alpha^3 + \alpha^2 + 1 \equiv \alpha^3 + \alpha^2 + \alpha + 1$$

Notice that $GF(2^4)^*$, in other words $GF(2^4) \setminus \{0\}$ is a cyclic group of order 15 generated by α, thus we can write $GF(2^4)^* =< \alpha >$.

Example 2.9. Let $p = 3$. Then $P(x) = x^3 + 2x + 2$ is irreducible over $GF(3)$. Let β be a root of $P(x)$. Then, the elements of $GF(3^3)$ can be written as polynomials $a_2\beta^2 + a_1\beta + a_0$ with $a_i \in GF(3)$. The order of $GF(3^3)$ is $3^3 = 27$ and the elements of $GF(3^3)$ are shown in Table 2.3.

Table 2.3. Elements of $GF(3^3)$ in polynomial representation.

0	β^2	$2\beta^2$
1	β^2+1	$2\beta^2+1$
2	β^2+2	$2\beta^2+2$
β	$\beta^2+\beta$	$2\beta^2+\beta$
2β	$\beta^2+2\beta$	$2\beta^2+2\beta$
$\beta+1$	$\beta^2+\beta+1$	$2\beta^2+\beta+1$
$\beta+2$	$\beta^2+\beta+2$	$2\beta^2+\beta+2$
$2\beta+1$	$\beta^2+2\beta+1$	$2\beta^2+2\beta+1$
$2\beta+2$	$\beta^2+2\beta+2$	$2\beta^2+2\beta+2$

Before ending this section with some basic facts about finite fields, we introduce an important mapping between an extension field $GF(q^k)$ and its ground field $GF(q)$ which we will use in the construction of some irreducible polynomials. We would like to notice that such mapping has been widely used on the efficient implementation of DL-based systems such as XTR [LV00].

Definition 2.14. *For $\alpha \in E = GF(q^m)$ and $F = GF(q)$, the trace of α from E to F, denoted by $\mathrm{Tr}_{E/F}(\alpha)$, is defined as*

$$\mathrm{Tr}_{E/F}(\alpha) = \alpha + \alpha^q + \alpha^{q^2} + \cdots + \alpha^{q^{m-1}}$$

The trace function satisfies the following properties:

1. $\mathrm{Tr}_{E/F}(\alpha + \beta) = \mathrm{Tr}_{E/F}(\alpha) + \mathrm{Tr}_{E/F}(\beta)$ *for all* $\alpha, \beta \in E$.

2. $\mathrm{Tr}_{E/F}(c\alpha) = c\mathrm{Tr}_{E/F}(\alpha)$ *for all* $c \in F$ *and* $\alpha \in E$.

3. $\mathrm{Tr}_{E/F}(c) = mc$ *for all* $c \in F$.

4. $\mathrm{Tr}_{E/F}(\alpha^q) = \mathrm{Tr}_{E/F}(\alpha)$ *for all* $\alpha \in E$.

If F is the prime subfield of E, then $Tr_{E/F}(\alpha)$ is called the *absolute trace* of α and simply denoted by $Tr_E(\alpha)$.

The following are some basic properties of finite fields:

(i) (Existence and uniqueness of finite fields) If F is a finite field then F contains p^m elements for some prime p and positive integer $m \geq 1$. For every prime power p^m, there is a unique, up to isomorphism, finite field of order p^m. Informally speaking, two finite fields are isomorphic if they are structurally the same, although the representation of their field elements may be different.

(ii) If $GF(q)$ is a finite field of order $q = p^m$, p a prime, then the characteristic of $GF(q)$ is p. In addition, $GF(q)$ contains a copy of $GF(p)$ as a subfield. Hence, $GF(q)$ can be viewed as an extension of $GF(p)$ of degree m.

(iii) Let $GF(q)$ a finite field of order $q = p^m$, then every subfield of $GF(q)$ has order p^n for some positive divisor n of m. Conversely, if n is a positive divisor of m, then there is exactly one subfield of $GF(q)$ of order p^n. An element $A \in GF(q)$ is in the subfield $GF(p^n)$ if and only if $A^{p^n} \equiv A$. The non-zero elements of $GF(q)$ form a group under multiplication called the multiplicative group of $GF(q)$, denoted $GF(q)^*$. In fact $GF(q)^*$ is a cyclic group of order $q-1$. Thus, $A^q = A$ for all $A \in GF(q)$. A generator of $GF(q)^*$ is called a primitive element of $GF(q)$.

(iv) Let $A \in GF(q)$, with $q = p^m$, then the multiplicative inverse of A can be computed as $A^{-1} \equiv A^{q-2}$. Alternatively, one can use the extended Euclidean algorithm for polynomials to find polynomials $S(\alpha)$ and $T(\alpha)$ such that $S(\alpha)A(\alpha) + T(\alpha)P(\alpha) = 1$, where $P(x)$ is an irreducible polynomial of degree m over $GF(p)$. Then, $A^{-1} = S(\alpha)$.

(v) If $A, B \in GF(q)$, with $GF(q)$ a finite field of characteristic p, then

$$(A+B)^{p^t} = A^{p^t} + B^{p^t}$$

for all $t \geq 0$.

2.5 Polynomial, Normal, and Dual Bases

Notice that in Table 2.2, we have shown two different representations of the elements of $GF(2^4)$. In one case we represent the elements of $GF(2^4)$ as polynomials, in the other case we represent the elements as powers of a suitable element, say a primitive element. In this section we describe different types of bases which can be used to represent the elements of a finite field $GF(q^m)$. Before continuing, we define the concept of conjugates.

Definition 2.15. *Let* $GF(q^m)$ *be an extension of* $GF(q)$ *and let* $\alpha \in GF(q^m)$. *Then the elements* $\alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{m-1}}$ *are called the conjugates of* α *with respect to* $GF(q)$.

Following Example 2.8, one can use different basis to represent the elements of a finite field. In particular, the two different representations from Table 2.2 lead to the ideas of polynomial basis and normal basis.

Definition 2.16. *Let* $E = GF(q^m)$ *and* $F = GF(q)$. *Then a basis of* E *over* F *of the form* $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-2}, \alpha^{m-1}\}$ *is called a polynomial basis, where* $\alpha \in GF(q^m)$ *and it is often taken to be a primitive element. Similarly a basis of* E *over* F *of the form* $\{\alpha, \alpha^q, \alpha^{q^2}, \ldots, \alpha^{q^{m-1}}\}$ *receives the name of a normal basis for a suitable element* $\alpha \in GF(q^m)$.

It can be shown that for any field $GF(q)$ and any extension field $GF(q^m)$, there exists always a normal basis of $GF(q^m)$ over $GF(q)$ [LN97, Theorem 2.35]. There has been a lot of work done on finding normal basis which are *optimal* to perform arithmetic operations. Such normal basis have received the name of *optimal* normal basis [MOVW89] because they allow efficient implementations of arithmetic operations in fields $GF(p^m)$. Notice that although there exist always a normal basis for every field, the same is not true in the case of optimal normal bases[2]. Another type of basis which has received attention in the literature is the dual basis.

Definition 2.17. $E = GF(q^m)$ *and* $F = GF(q)$. *Then two bases* $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ *and* $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ *of* E *over* F *are said to be dual or complementary bases if for* $0 \le i, j \le m-1$ *we have:*

$$\mathrm{Tr}_{E/F}(\alpha_i \beta_j) = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

References [MKW89, WB90] define the concept of a weakly dual basis as follows:

Definition 2.18. *Let* E, F, *be defined as in Definition 2.17. Then two bases* $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ *and* $\{\beta_0, \beta_1, \ldots, \beta_{m-1}\}$ *of* E *over* F *are said to be weakly dual to each other if for* $0 \le i, j \le m-1$ *we* *have:*

$$\mathrm{Tr}_{E/F}(\gamma\alpha_i\beta_j) = \begin{cases} 0 & \text{for } i \neq j \\ 1 & \text{for } i = j \end{cases}$$

for $\gamma \in E \setminus \{0\}$.

Reference [WHB02] used weakly dual basis to build finite field multipliers for fields $GF(q^m)$, where q is an odd prime power. Finally, it is important to point out that given a basis $\{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ of $GF(q^m)$ over $GF(q)$, one can always represent an element $\beta \in GF(q^m)$ as:

$$\beta = b_0\alpha_0 + b_1\alpha_1 + \cdots + b_{m-1}\alpha_{m-1}$$

where $b_i \in GF(q)$.

2.6 Irreducible Polynomials

This section summarizes numerous and recent advances on irreducible polynomials. This is interesting for two reasons. First, the results on irreducible polynomials are very dispersed throughout the literature, specially when the finite field is of odd characteristic. Second and probably more importantly, in recent years there have been improvements on generation of irreducible trinomials which are of interest to the research community since they provide efficient implementations of finite field arithmetic which in turn is the basis for numerous applications in cryptography and error correcting codes. In addition, as seen from the literature, there has been a lot of work done on fields of characteristic two but the same statement is not true when we consider fields of odd characteristic. Finally, we use some of the theorems presented in this section to generate tables and alternative construction methods for irreducible polynomials which are of interest on their own right. We use [LN97] as a convenient reference for results well established in the literature[3]. In this section, we will use the notation $\mathbb{F}_q$, q a prime power, to refer to a finite field $GF(q)$, simply because it seems less overwhelming to write $\mathbb{F}_q[x]$ to refer to the ring of polynomials over $\mathbb{F}_q$, for example, than its equivalent with the $GF(\cdot)$ notation.

2.6.1 Preliminaries

Let $\mathbb{F}_q$ denote the finite field with $q = p^m$ elements, for some prime p, and $\mathbb{F}_q[x]$ denote the set of polynomials over $\mathbb{F}_q$. As implied by Definition 2.11, we say that a polynomial $P(x) \in \mathbb{F}_q$ is irreducible if it can not be factored into a non-trivial product of lower degree polynomials in $\mathbb{F}_q[x]$.

Definition 2.19. *The order (or exponent or period) of a non-zero polynomial $P(x) \in \mathbb{F}_q[x]$, with $P(0) \neq 0$, is defined to be the least positive integer e for which $P(x)$ divides $x^e - 1$, and it is denoted by $\mathrm{ord}(P) = \mathrm{ord}(P(x))$. If $P(0) = 0$, then $P(x) = x^h G(x)$ for some $h \in \mathbb{N}$ and $G(x) \in \mathbb{F}_q[x]$, with $G(0) \neq 0$, and $\mathrm{ord}(P)$ is defined to be ord(G).*

Notice that if $P(x)$ is irreducible over $\mathbb{F}_q[x]$ then $P(0) \neq 0$ by definition. A polynomial $P(x)$ is called primitive if it has degree n and $\mathrm{ord}(P) = q^n - 1$. It is sometimes convenient to define the notion of the index of $P(x)$ as $q^n - 1/e$, where $e = \mathrm{ord}(P)$.

We denote by $I_{q,n}$ the number of irreducible polynomials of degree n over $\mathbb{F}_q$. Then, it can be shown (see [LN97, Theorem 3.25]) that

$$I_{q,n} = \frac{1}{n} \sum_{d|n} \mu\left(\frac{n}{d}\right) q^d = \frac{1}{n} \sum_{d|n} \mu(d) q^{n/d} \tag{2.1}$$

where $\sum_{d|n}$ means the summation over all positive integers d divisors of n and $\mu(\cdot)$ is the Moebius function defined as follows

Definition 2.20. *The Moebius function μ is the function on $\mathbb{N}$ defined by*

$$\mu(n) = \begin{cases} 1 & \textit{if } n = 1, \\ (-1)^k & \textit{if } n \textit{ is the product of } k \textit{ distinct primes}, \\ 0 & \textit{if } n \textit{ is divisible by the square of a prime}. \end{cases}$$

Example 2.10. The number of irreducible polynomials of degree 6 over $\mathbb{F}_q$ is given by

$$I_{q,6} = \frac{1}{6}\left(\mu(1)q^6 + \mu(2)q^3 + \mu(3)q^2 + \mu(6)q\right) = \frac{1}{6}\left(q^6 - q^3 - q^2 + q\right)$$

Notice that fields of the form $\mathbb{F}_{3^{6m}}$ have been proposed for cryptographic applications with $3^{6m} \geq 2^{1024}$ in [Jou00, Ver01, BF01, BBS01, BKLS02, GHS02a, PS02].

As a side remark, notice that (2.1) can be used to show the well-known fact that for every finite field $\mathbb{F}_q$ and every integer $n \in \mathbb{N}$ there exists an irreducible polynomial in $\mathbb{F}_q[x]$ of degree n [LN97].

Next, we define the concept of the discriminant of a polynomial. Before, we do this, we introduce the notion of the splitting field of a polynomial.

Definition 2.21. *Let $F \in \mathbb{K}[x]$ be of positive degree and $\mathbb{E}$ an extension field of $\mathbb{K}$. Then, F splits in $\mathbb{E}$ if F can be written as a product of linear factors in $\mathbb{E}[x]$, that is if there exists elements $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathbb{E}$ such that*

$$F(x) = a(x - \alpha_1)(x - \alpha_2) \cdots (x - \alpha_n),$$

where a is the leading coefficient of F. The field $\mathbb{E}$ is the splitting field of F over $\mathbb{K}$ if F splits in $\mathbb{E}$ and if, moreover, $\mathbb{E} = \mathbb{K}(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

In other words, $\mathbb{E}$ is the field formed by adjoining all the roots of F to $\mathbb{K}$. It can be shown, that the splitting field of F over $\mathbb{K}$ always exists and it is unique up to isomorphism. Now, we can define the discriminant of F.

Definition 2.22. *Let $F \in \mathbb{K}$ be a polynomial of degree $n \geq 2$ and suppose that $F(x) = a(x - \alpha_1)(x - \alpha_2) \cdots (x - \alpha_n)$ with $\alpha_1, \alpha_2, \ldots, \alpha_n$ in the splitting field of F over $\mathbb{K}$. Then the discriminant $D(F)$ of F is defined by*

$$D(F) = a^{2n-2} \prod_{1 \leq i < j \leq n} (\alpha_i - \alpha_j)^2$$

From the definition, one can deduce that $D(F) = 0$ if and only if the polynomial has multiple roots. Notice, also that although the α_i's are all in $\mathbb{E}$, the splitting field of F, one can show that $D(F) \in \mathbb{K}$. For small values of n, this can be seen by explicit calculation, as Example 2.11 shows.

Example 2.11. Let $F(x) = ax^2 + bx + c = a(x - \alpha_1)(x - \alpha_2) \in \mathbb{K}[x]$ with $n = 2$. Then, from the definition of the discriminant $D(F) = a^2(\alpha_1 - \alpha_2)^2 = a^2((\alpha_1 + \alpha_2)^2 - 4\alpha_1\alpha_2) = a^2(b^2a^{-2} - 4ca^{-1})$

(the last equality was obtained by comparing coefficients of ax^2+bx+c and $a(x-\alpha_1)(x-\alpha_2)$). Thus,

$$D(F) = b^2 - 4ac$$

In general, it is not possible to determine an explicit formula for the discriminant of a polynomial over a field. However, for the case of trinomials Dalen Swan [Swa62] have found an explicit formula.

Theorem 2.1. *[Swa62] Let $F(x) = x^n + ax^k + b \in \mathbb{F}_q[x]$, for odd q, $n > k > 0$, and $d = \gcd(n,k)$ with $n = n_1 d, k = k_1 d$, then*

$$D(F) = (-1)^{n(n-1)/2} \cdot b^{k-1} \cdot \left[n^{n_1} \cdot b^{n_1-k_1} - (-1)^{n_1} \cdot (n-k)^{n_1-k_1} \cdot k^{k_1} \cdot a^{n_1}\right]^d$$

2.6.2 Irreducible Binomials

It is well known that choosing an irreducible polynomial with the least number of non-zero coefficients leads to a more efficient implementations of finite field arithmetic. Thus, this section is devoted entirely to known results regarding irreducible binomials, which in a sense are an optimal choice with respect to efficient arithmetic in odd characteristic fields, and irreducible trinomials, which for the characteristic two case are optimal since binomials are never irreducible over $GF(2)$. As we can see from Theorem 2.2, the existence of irreducible binomials is entirely established.

Theorem 2.2. *[LN97, Theorem 3.75] Let $m \geq 2$ be an integer and $\omega \in \mathbb{F}_q^*$. Then the binomial $x^m - \omega$ is irreducible in $\mathbb{F}_q[x]$ if and only if the following two conditions are satisfied: (i) each prime factor of m divides the order e of $\omega \in \mathbb{F}_q^*$, but not $(q-1)/e$; (ii) $q \equiv 1 \bmod 4$ if $m \equiv 0 \bmod 4$.*

Notice that the first condition in Theorem 2.2 implies that $\gcd(m, (q-1)/e) = 1$. An interesting corollary is given in [Men93].

Corollary 2.1. *[Men93] Let r be a prime factor of $q-1$ and $\omega \in \mathbb{F}_q$ have order e such that $r \nmid (q-1)/e$. Assume that $q \equiv 1 \bmod 4$ if $r = 2$ and $k \geq 2$. Then for any integer $k \geq 0$,*

$$x^{r^k} - \omega$$

is irreducible over $\mathbb{F}_q$.

The case where $r = 2$ and $q \equiv 3 \bmod 4$ is explicitly not allowed by Corollary 2.1. It can be shown that $x^{2^k} - \omega$ is always reducible for $k > 1$ and all $\omega \in \mathbb{F}_q$ if $q \equiv 3 \bmod 4$ (see [BGL93, BGM93] and Theorem 3.76 in [LN97]).

Example 2.12. Theorem 2.2 and Corollary 2.1 imply the following:

(i) $x^2 - 1$ is always reducible since 1 is always a root of this polynomial, i.e. $x^2 - 1 = (x+1)(x-1)$ over any finite field. Notice that there are not irreducible binomials over $\mathbb{F}_2$ since 1 is always a root of such a binomial.

(ii) Over $\mathbb{F}_3$, we can not use Corollary 2.1 because $q \equiv 3 \bmod 4$ and $r = 2$. Also, the only elements in $\mathbb{F}_3^*$ are $\{1, -1\}$ and $e = \text{order}(-1) = 2$ in $\mathbb{F}_3$. Thus, from Theorem 2.2, the only irreducible binomial over $\mathbb{F}_3$ is $x^2 + 1$, since 2 does not divide $q - 1/e$ and $m \equiv 2 \bmod 4$.

(iii) Over $\mathbb{F}_5$, we have $q \equiv 1 \bmod 4$ and $\{2, -2, -1\}$ for candidates for $\omega \in \mathbb{F}_5^*$ in Theorem 2.2. However, $\text{order}(-1) = 2$, and so there is no possible m which satisfies Theorem 2.2. A quick check, will verify that both $x^{2^k} - 2$ and $x^{2^k} + 2$ are irreducible over $\mathbb{F}_5$ for $k \geq 0$.

(iv) Over $\mathbb{F}_7$, Table 2.4 shows the possible values of ω and the corresponding degrees m.

Table 2.4. Possible degrees for irreducible binomials $x^m - \omega$ over $\mathbb{F}_7$

ω	$e = \text{order}(\omega)$	$q - 1/e$	m
2	3	2	3^k
3	6	1	3^k and $2 \cdot 3^k$
-3	3	2	3^k
-2	6	1	3^k and $2 \cdot 3^k$
-1	2	3	2

2.6.3 Irreducible Trinomials

A trinomial is a polynomial with three non-zero coefficients. They are of great interest because in many applications there are no irreducible binomials (see Example 2.12(i)) and thus, the best one can hope for to speed up the field arithmetic is to find an irreducible trinomial[4]. Interestingly enough very little is known about trinomials in general. Thus, the first part of this section is devoted to reviewing a few "classical" results as well as recursive constructions to obtain irreducible polynomials from known ones. In the second part, we treat the recent results from von zur Gathen [vzG03].

Classical Results

As mentioned in Section 2.6.2, binomials of the form $x^{2^k} - \omega$ over $\mathbb{F}_q$ are always reducible over $\mathbb{F}_q$ for all $\omega \in \mathbb{F}_q$ if $q \equiv 3 \bmod 4$. However, it is possible to construct irreducible trinomials as shown in [BGM93].

Theorem 2.3. *[BGM93, Theorem 1] Let* $p \equiv 3 \bmod 4$ *be a prime and* v *the largest integer such that* $2^v|(p+1)$. *Define* a_v *recursively by the formula*

$$a_i = \begin{cases} \pm\left(\frac{a_{i-1}+1}{2}\right)^{(p+1)/4} & \text{for } \ 2 \leq i \leq v-1 \\ \pm\left(\frac{a_{i-1}-1}{2}\right)^{(p+1)/4} & \text{for } \ i = v \end{cases}$$

with the initial $a_1 = 0$ *and at each step one can choose either* $+$ *sign or* $-$ *sign. Then*

$$x^{2^k} - 2a_v x^{2^{k-1}} - 1$$

is irreducible over $\mathbb{F}_p$ *for every integer* $k \geq 1$.

Before we continue we introduce a well known result on how to construct an irreducible polynomial over an extension field $\mathbb{F}_{q^k}$ given one which is irreducible over $\mathbb{F}_q$.

Theorem 2.4. *[LN97, Theorem 3.46] An irreducible polynomial over* $\mathbb{F}_q$ *of degree* n *remains irreducible over* $\mathbb{F}_{q^k}$ *if and only if* $\gcd(k, n) = 1$.

Example 2.13. The polynomial $P(x) = x^{11} + x^2 + 1$ is irreducible over $\mathbb{F}_2$. Thus, by Theorem 2.4 it remains irreducible over $\mathbb{F}_{2^{2^r}}$ for any $r \geq 1$, since $\gcd(2^r, 11) = 1$. This construction was used in [DBV+96] and [GP97] to perform efficient field arithmetic in the field $\mathbb{F}_{2^{176}} \cong GF((2^{16})^{11})$ in the context of elliptic curve based cryptography.

Reference [BGL93] noticed that if $q = p^m \equiv 3 \bmod 4$, for p a prime, m must be odd and thus, combining this result with Theorem 2.4, we see that $x^{2^k} - 2a_v x^{2^{k-1}} - 1$ is also irreducible over $\mathbb{F}_q$

for every integer $k \geq 1$. The previous statement together with Theorems 2.2 and 2.3 prove that when q is odd there is always an irreducible binomial or trinomial of degree 2^k over $\mathbb{F}_q$ [BGL93]. Another classical result on irreducible trinomials is the following:

Theorem 2.5. *[LN97, Theorem 3.78] Let $b \in \mathbb{F}_q$ and let p be the characteristic of $\mathbb{F}_q$. Then the trinomial $x^p - x - b$ is irreducible in $\mathbb{F}_q[x]$ if and only if the $Tr_{\mathbb{F}_q}(b) \neq 0$*

Corollary 2.2. *For $a, b \in \mathbb{F}_q^*$, the trinomial $x^p - ax - b$ is irreducible over $\mathbb{F}_q$ if and only if $a = A^{p-1}$ for some $A \in \mathbb{F}_q$ and $Tr_{\mathbb{F}_q}(b/A^p) \neq 0$*

There are several recursive constructions of irreducible polynomials which lead to new irreducible polynomials. We refer to [Men93, Chapter 3] for a nice treatment of such constructions. Here we include one result which we will use in Chapter 7 and which itself leads to new irreducible trinomials given an irreducible trinomial. Notice that if $F(x)$ is a trinomial, then $F(x^t)$ is also a trinomial.

Theorem 2.6. *[LN97, Theorem 3.35] Let $F_1(x), F_2(x), \ldots, F_N(x)$ be all the distinct monic irreducible polynomials in $\mathbb{F}_q[x]$ of degree m and order e, and let $s \geq 2$ be an integer whose prime factors divide e but not $q^m - 1/e$. Assume also that $q^m \equiv 1 \bmod 4$ if $s \equiv 0 \bmod 4$. Then $F_1(x^s), F_2(x^s), \ldots, F_N(x^s)$ are all the distinct monic irreducible polynomials in $\mathbb{F}_q[x]$ of degree ms and order es.*

Corollary 2.3. *[BGL93, Corollary 10] Let $F(x) \in \mathbb{F}_q[x]$ a polynomial of degree m with exponent e and let r be a prime factor of e such that r does not divide $q^m - 1/e$. Assume also that $q^m \equiv 1 \bmod 4$ if $r = 2$. Then $F(x^{r^k})$ is irreducible over $\mathbb{F}_q[x]$ for every integer $k \geq 1$.*

New Results on Trinomials

Fields $\mathbb{F}_{3^m}$ have received a lot of attention in the past couple of years because of their applications in cryptography. This section summarizes the results from [Loi00] and [vzG03] which we use to derive optimized cubing architectures well suited for FPGAs in Chapter 5 as well as to provide tables of irreducible polynomials over fields of odd characteristic. In the following discussion we use a theorem due to Swan [Swa62]

Theorem 2.7. *[Swa62] Let $\mathbb{F}_q$ be a finite field of odd characteristic, $n > k \geq 1$ are integers, $F(x) = x^n + ax^k + b \in \mathbb{F}_q[x]$ with $a, b \in \mathbb{F}_q^*$, r is the number of irreducible factors of F in $\mathbb{F}_q[x]$, and $D \in \mathbb{F}_q$ is the discriminant of F. Then if F is square free, then $r \equiv n \bmod 2$ if and only if D is a square in $\mathbb{F}_q$.*

In [Loi00], Loidreau studies trinomials $P(x) = x^n \pm x^k \pm 1$ over $\mathbb{F}_3$ and finds congruences for n and k which together with the number of times that 2 divides n and k characterize the property of $P(x)$ being square free and having an odd number of irreducible factors. Notice that any irreducible polynomial always satisfies such property but the converse is not true [vzG03]. Over $\mathbb{F}_3$ there are four monic trinomials of degree n over $\mathbb{F}_3$, i.e., $x^n + x^k + 1$, $x^n + x^k - 1$, $x^n - x^k + 1$, and $x^n - x^k - 1$ and the polynomial $x^n + x^k + 1$ is always reducible over $\mathbb{F}_3$ since 1 is always a root of it. Using Theorem 2.7 and the expression for the discriminant of trinomial (Theorem 2.1), reference [Loi00] computes explicitly for which values of n and k, a trinomial $P(x)$ over $\mathbb{F}_3$ has an odd number of irreducible factors. These results are summarized in Table 2.5 together with the corrections pointed out in [vzG03].

Table 2.5. Values of n and k for which the trinomials over $\mathbb{F}_3$ have an odd number of irreducible factors. Here for an integer s, $v_2(s)$ implies that $s = 2^{v_2(s)} s_1$ where s_1 is odd.

$n \bmod 12$	$x^n + x^k + 1$	$x^n + x^k - 1$	$x^n - x^k + 1$	$x^n - x^k - 1$
0	—	$k \equiv 2,4 \bmod 6$	—	$k \equiv 2,4 \bmod 6$
1	$k \equiv 4,5 \bmod 6$	$k \equiv 0,1 \bmod 3$	$k \equiv 0,1 \bmod 3$	$k \equiv 0,1 \bmod 3$
2	$k \equiv 0,2 \bmod 3$	$k \equiv 1 \bmod 6$	$k \equiv 0,2 \bmod 3$ $k \equiv 4 \bmod 6$	$k \equiv 1 \bmod 6$
3	$k \equiv 0,1 \bmod 3$	$k \equiv 2 \bmod 3$	$k \equiv 1,2 \bmod 3$	$k \equiv 1 \bmod 3$
4	—	$k \equiv 0,1 \bmod 3$ $k \equiv 2 \bmod 6, v_2(k) \neq v_2(n)$	—	$k \equiv 0,1 \bmod 3$ $k \equiv 2 \bmod 6, v_2(k) \leq v_2(n)$
5	—	$k \equiv 4 \bmod 6$	$k \equiv 1 \bmod 3$	$k \equiv 1 \bmod 6$
6	$k \equiv 1,2 \bmod 3$	$k \equiv 1,5 \bmod 6$	$k \equiv 1,2 \bmod 3$	$k \equiv 1,5 \bmod 6$
7	—	$k \equiv 2 \bmod 6$	$k \equiv 2,5 \bmod 6$	$k \equiv 5 \bmod 6$
8	—	$k \equiv 0,2 \bmod 3$ $k \equiv 4 \bmod 6, v_2(k) \neq v_2(n)$	—	$k \equiv 0,2 \bmod 3$ $k \equiv 4 \bmod 6, v_2(k) \leq v_2(n)$
9	$k \equiv 1,2 \bmod 6$	$k \equiv 1 \bmod 3$	$k \equiv 4,5 \bmod 6$	$k \equiv 2 \bmod 3$
10	$k \equiv 0,1 \bmod 3$	$k \equiv 5 \bmod 6$	$k \equiv 0,1 \bmod 3$ $k \equiv 2 \bmod 6, v_2(k) \geq v_2(n)$	$k \equiv 5 \bmod 6$
11	$k \equiv 0,2 \bmod 3$	$k \equiv 0,2 \bmod 3$	$k \equiv 0,2 \bmod 3$	$k \equiv 0,2 \bmod 3$

Reference [vzG03] further generalizes the work in [Loi00]. In particular, von zur Gathen [vzG03] gives a necessary condition (but not sufficient) for trinomials over $\mathbb{F}_q$ for q an odd prime power to be irreducible. This result is summarized in Theorem 2.8. Before stating the theorem, we make a definition which we will use in the theorem.

Definition 2.23. *We say that a polynomial $F \in \mathbb{F}_q[x]$ satisfies property (S) if F is square free and it has an odd number of irreducible factors*

Theorem 2.8. *Let q be a power of the odd prime p, $F(x) = x^n + ax^k + b$ with $a, b \in \mathbb{F}_q^*$, $n > k \geq 1$,*

$d = \gcd(n, k), n_1 = n/d, k_1 = k/d, m_2 = p(q-1)$, *and* $m_1 = \mathrm{lcm}(4, m_2)$. *Then the discriminant of* F *and property* (S) *depend only on the following residues:*

$$n \bmod m_1,\ k \bmod m_2,\ n_1 \bmod q-1 \text{ and } k_1 \bmod q-1$$

In order to minimize the search for irreducible polynomials, [vzG03] notices the following useful transformations:

1. Let $s, k_0 = 0 < k_1 < \cdots < k_{s-1} = n$ be non-negative integers and

$$R = \left\{ \sum_{0 \leq i < s} a_i x^{k_i} : \text{all } a_i \in \mathbb{F}_q^*, a_n = 1 \right\} \subseteq \mathbb{F}_q[x]$$

   be the set of monic polynomials with support $\{k_0, \ldots, k_{s-1}\}$. Notice that each $F \in R$ is s-sparse. Then its *monic reversal*, denoted $\widetilde{F}$, is defined by

$$\widetilde{F} = a_0^{-1} x^n f(x^{-1}) = a_0^{-1} \sum_{0 \leq i < s} a_i x^{n-k_i}$$

   and it is also s-sparse. This transformation preserves square freeness and the number of irreducible factors.

2. Let $u \in \mathbb{F}_q^*$ and $F \in R$, set $F_u = u^{-n} F(ux)$. Then, $F_u \in R$ and the transformation preserves square freeness and the number of irreducible factors.

3. Assume the notation as in Theorem 2.8, and $1 \leq k^* < n^*$ with $n^* \equiv -n \bmod m_1, k^* \equiv -k \bmod m_2, n_1^* \equiv -n_1 \bmod (q-1), k_1^* \equiv -k_1 \bmod (q-1)$, and $F^* = x^{n^*} + ax^{k^*} + b$. Then F has property (S) if and only if F^* does.

Notice that the concept of s-sparse refers to the number of non-zero coefficients in a given polynomial. Thus, in the above transformations if you begin with a trinomial after the transformation you still have a trinomial. Similarly, a transformation which preserves square freeness and the number of irreducible factors implies that if $F \in \mathbb{F}_q[x]$ satisfies property (S) then the transformed polynomial will also satisfy this property.

2.6.4 Irreducible AOPs and ESPs

Irreducible All One Polynomials (AOPs) and Equally Spaced Polynomials (ESPs) have been repeatedly proposed in the literature to optimize arithmetic in fields of characteristic two [IT89, Ito91, HWB92, cKKS98, WH98, LLL01] as well as arithmetic in fields of odd characteristic [GP02]. In the following, we formally define AOPs and ESPs and show two constructions for irreducible AOPs and ESPs. These definitions and constructions are used in Chapter 7 to instantiate fields which allow for an efficient implementation of the Itoh and Tsujii algorithm for inversion [IT88].

Definition 2.24. *[WW84] A polynomial $F(x) = x^k + x^{k-1} + \cdots + x + 1$ over $GF(q)$ is called an All One Polynomial (AOP) of degree k.*

Definition 2.25. *[IT89] A polynomial $G(x) = x^{sk} + x^{s(k-1)} + \cdots + x^s + 1 = F(x^s)$ over $GF(q)$, where $F(x)$ is an AOP of degree k over $GF(q)$ is called a binary s-Equally Spaced Polynomial (s-ESP) of degree sk.*

We have abused the original definitions which were for the case $q = 2$ and generalized them to $q = p^n$, p an odd prime. Notice that AOPs are just a special case of binary s-ESPs in which $s = 1$. Theorem 2.9 describes the cases when irreducible AOPs exist.

Theorem 2.9. *[Men93, Chapter 5] The polynomial $f(x) = x^m + x^{m-1} + \cdots + x + 1$ is irreducible over $GF(q)$ if and only if $m + 1$ is prime and q is primitive in $GF(m + 1)$*

We can use Theorem 2.6 to construct an irreducible s-ESP given an irreducible AOP. In particular, by choosing $GF(q)$ and m such that any of the $F_i(x)$ in Theorem 2.6 is a binary AOP, we immediately obtain a binary irreducible s-ESP, where s satisfies the conditions in Theorem 2.6. Notice, also, that if we construct an irreducible s-ESP of degree sm over $GF(q)$ using Theorem 2.6, call it $P(x)$, then $P(x)$ is also irreducible over $GF(q^k)$, for k satisfying $\gcd(k, sm) = 1$ by Theorem 2.4.

2.7 Notes and Further References

1. McEliece [McE89] also provides a nice introduction to the theory of finite fields as well as to their applications in engineering. The books by Lidl and Niederreiter [LN97] and Jungnickel [Jun93] provide

comprehensive treatments of the subject. It is also worth mentioning the book by Menezes [Men93] which has extensive treatment of topics such as constructions of normal and optimal normal bases.

2. Optimal normal bases were introduced in [MOVW89] as a way of implementing discrete logarithm-based systems over binary fields $GF(2^n)$ with $n \geq 1000$ which at the time seemed infeasible using only normal basis. Although the authors in [MOVW89] make emphasis on binary fields, their work also applies to fields $GF(p^m)$, for odd primes p. Reference [GL92] showed that the normal bases found in [MOVW89] were essentially all the normal bases. Gao and Vanstone [GV95] provide experimental results on the multiplicative orders of optimal normal basis generators in $GF(2^n)$ over $GF(2)$ for $n \leq 1200$. See also [Men93, Chapters 4 and 5] for a nice overview of optimal normal basis and some constructions. Normal and optimal normal bases have been repeatedly proposed for efficient arithmetic in finite fields of characteristic two with applications to discrete logarithm based systems [AMOV91] as well for elliptic curves [AMV93, MV93]. Parallel architectures for multiplication using normal basis have been proposed in [WTS+85, MO86, cKKS98, ScKK01, RMH02a, Kwo03]. Reference [RMH03a] offers multiplier architectures for composite fields $GF((2^n)^m)$ using normal basis while [RMH02b] and [RMH03b] propose digit-serial and sequential multipliers using normal basis, respectively. References [HTDR88, JB92, PL95] offer comparisons between binary finite field architectures based on the chosen basis for small fields while [GG90] compares polynomial and normal basis architectures for field sizes of interest in cryptography. We refer to Section 7.5 in Chapter 7 for uses of normal bases for inversion in finite fields.

3. Lidl and Niederreiter [LN97] provide a comprehensive treatment of irreducible polynomials. References [BGL93, Men93] also give interesting overviews of known results on irreducible polynomials until the year 1993. In addition, reference [BGL93] extends the tables of irreducible trinomials over $GF(2)$ in [ZB68, ZB69] up to degree $m \leq 2000$. Reference [Zie69] provides tables of primitive trinomials of degree p such that $2^p - 1$ is known to be prime and $p \leq 11213$. Tables of irreducible trinomials over $GF(2)$ are also found in [MvOV97] for degree up to $m \leq 1478$ and for primitive trinomials over $GF(2^n)$ of degree $m \leq 229$. Reference [CQS01] proved that if p is a prime and $p \equiv 13 \bmod 24$ or $p \equiv 19 \bmod 24$ then there is no irreducible trinomial of degree p over $GF(2)$. References [Loi00] and [vzG03] used the work of Swan [Swa62] to describe necessary conditions (but not sufficient) for trinomi-

als to be irreducible over $GF(3)$. Vishne [Vis97] provides some sufficient conditions for a trinomial over a field of characteristic two to be reducible. For example, [Vis97] shows that $x^m + ax^k + b \in GF(2^n)[x]$ is reducible if both m, n are even except possibly when $m = 2k$ and k is odd.

4. There has been a great deal of research done on the arithmetic advantages of irreducible polynomials of special form over $GF(2)$. In [IT89], AOPs and ESPs over $GF(2)$ are introduced. The authors show necessary and sufficient conditions for ESPs to be irreducible over $GF(2)$ and propose a new configuration of parallel multipliers for fields $GF(2^m)$, based on irreducible AOPs and ESPs over $GF(2)$. In [Ito91], necessary and sufficient conditions are given for a family of infinitely many ESPs to be irreducible over $GF(2)$. In addition, a uniqueness criteria which characterizes all irreducible ESPs over $GF(2)$ in a strict sense is presented.

CHAPTER 3

Architectures for Arithmetic in Small $GF(p)$ Fields

In this chapter, we survey previous hardware architectures for performing addition operations in $\mathbb{Z}$ and addition and multiplication in $GF(p)$ fields, where p is odd, prime, and *small*. Section 3.1 deals with integer adders which will be fundamental building blocks for the $GF(p)$ multipliers presented in Chapter 4. The remaining of the material presented in this chapter comes from the literature on Residue Number Systems (RNS) which is particularly useful in Digital Signal Processing (DSP) applications. However, we depart from these traditional applications. In Chapter 5, we use the architectures presented here to build multipliers for $GF(p^m)$ fields, for p odd, and large enough for cryptographic applications, i.e., with $p^m \geq 2^{160}$. At the end of this chapter, we describe new $GF(p)$ multipliers, for $p > 2$, specially suited for $GF(p^m)$ multiplication. Parts of this chapter appear in [GWP02].

3.1 Integer Adders

It is well known that adders constitute the basic building blocks for more complicated arithmetic operators such as multipliers. Thus, this section surveys adder architectures which we will used in future sections to implement more complicated operators. Our focus, rather than comprehensive, is on those adders to which we will refer in future chapters as we survey architectures for $GF(p)$ arithmetic in the context of cryptographic applications. For more detailed treatments of hardware architectures and computer arithmetic, we refer the reader to [Kor93, Par99].

In what follows, we consider the addition of two n-bit integers $A = \sum_{i=0}^{n-1} a_i 2^i$ and $B = \sum_{i=0}^{n-1} b_i 2^i$, with $S = c_{out} 2^n + \sum_{i=0}^{n-1} s_i 2^i = A + B$ being possibly an $(n+1)$-bit integer. We refer to A and B as the inputs (and to their bits a_i and b_i as the input bits) and to S as the sum (and to its bits s_i for $i = 0 \cdots n-1$ as the sum bits). The last bit of the sum c_{out} receives the special name of carry-out bit.

3.1.1 Ripple-Carry Adders (RCA)

Single-bit half-adders (HA) and full-adders (FA) are the basic building blocks used to synthesize more complex adders. A HA accepts two input bits a and b and outputs a sum-bit s and a carry-out bit c_{out} following (3.1) and (3.2)

$$
\begin{aligned}
s &= a \oplus b & (3.1)\\
c_{out} &= a \wedge b & (3.2)
\end{aligned}
$$

A half-adder can be seen as a single-bit binary adder that produces the 2-bit binary sum of its inputs, i.e., $a + b = (c_{out}\ s)_2$. In a similar manner, a full-adder accepts a 3-bit input a, b and a carry-in bit c_{in}, and outputs two bits: a sum-bit s and a carry-out bit c_{out}, according to (3.3) and (3.4)

$$
\begin{aligned}
s &= a \oplus b \oplus c_{in} & (3.3)\\
c_{out} &= (a \wedge b) \vee (c_{in} \wedge (a \vee b)) \\
&= (a \wedge b) \vee (a \wedge c_{in}) \vee (b \wedge c_{in}) & (3.4)
\end{aligned}
$$

Pictorially, we can view half-adders and full-adders as depicted in Figures 3.1 and 3.2. We notice that because of the importance of the FA as an arithmetic building block, there are many optimized FA designs for a variety of implementation technologies [Par99]. An n-bit ripple-carry adder (RCA) can be synthesized by concatenating n single-bit FA cells, with the carry-out bit of the ith-cell used as the

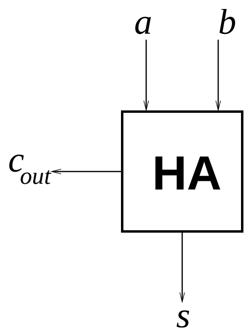


Figure 3.1. Half-adder cell

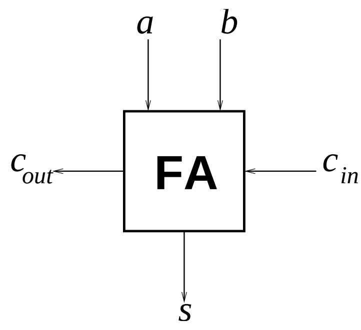


Figure 3.2. Full-adder cell

carry-in bit of the $(i+1)$th-cell. The resulting n-bit adder outputs an n-bit long sum and a carry-out bit. This is depicted in Figure 3.3. The total latency of an n-bit RCA can be approximated by $n \cdot T_{\text{FA}}$, where

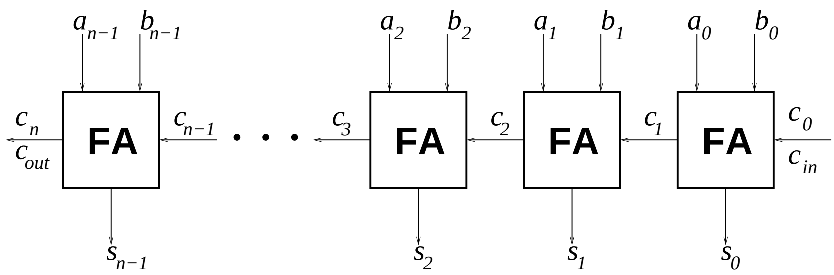


Figure 3.3. n-bit carry-ripple adder

T_{FA} refers to the delay of a single full-adder cell. Designing n-bit RCAs for any value of n is a rather simple task: simply, concatenate as many FA cells as bits of precision are required. In addition, although not directly relevant to the treatment here, RCA-based designs have two other advantages: (1) easy sign detection if one uses 2’s complement arithmetic, and (2) subtraction is accomplished by first converting the subtrahend to its 2’s complement representation and then adding the result to the original minuend. However, the delay of the RCA grows linearly with n, making it undesirable for large values of n or for high-speed applications, as it is the case in many cryptographic systems. Thus, the need to explore other designs to improve the performance of the adder without significantly increasing area-resource requirements.

3.1.2 Carry-Lookahead Adders (CLA)

As it name indicates a carry lookahead adder (CLA) computes the carries generated during an addition before the addition process takes place, thus, reducing the total time delay of the RCA at the cost of additional logic. We next make some definitions that will help us in developing a CLA.

Definition 3.1. *Let a_i, b_i be two operand digits in radix-r notation and c_i be the carry-in digit. Then, we define the generate signal g_i, the propagate signal p_i, and the annihilate (absorb) signal v_i as:*

$$\begin{aligned} g_i &= 1 \quad \textit{if} \quad a_i + b_i \geq r \\ p_i &= 1 \quad \textit{if} \quad a_i + b_i = r - 1 \\ v_i &= 1 \quad \textit{if} \quad a_i + b_i < r - 1 \end{aligned}$$

where $c_i, g_i, p_i, v_i \in \{0, 1\}$ and $0 \leq a_i, b_i < r$.

Notice that Definition 3.1 is independent of the radix used which allows one to treat the carry propagation problem independently of the number system [Par99]. Specializing to the binary case and using the signals from Definition 3.1, we can re-write g_i, p_i, and v_i as:

$$g_i = a_i \wedge b_i \tag{3.5}$$

$$p_i = a_i \oplus b_i \tag{3.6}$$

$$v_i = \overline{a_i} \wedge \overline{b_i} = \overline{a_i \vee b_i} \tag{3.7}$$

Relations (3.5), (3.6), and (3.7) have very simple interpretations. If $a_i, b_i \in GF(2)$, then a carry will be generated whenever both a_i and b_i are equal to one, a carry will be propagated if either a_i or b_i are equal to one, and a carry will be absorbed whenever both input bits are equal to zero. In some cases it is also useful to define a transfer signal ($t_i = a_i \vee b_i$) which denotes the event that the carry-out will be one given that the carry-in is equal to one [1]. Combining (3.4), (3.5), and (3.6) we can write the carry-recurrence relation as follows:

$$c_{i+1} = g_i \vee (c_i \wedge t_i) = g_i \vee (c_i \wedge p_i) \tag{3.8}$$

Proof.

$$\begin{aligned} c_{i+1} &= g_i \vee (c_i \wedge t_i) = g_i \vee (c_i \wedge (a \vee b)) \\ &= g_i \vee (c_i \wedge ((a \vee b) \wedge (a \vee \overline{a}))) = g_i \vee (c_i \wedge (a \vee (\overline{a} \wedge b))) \\ &= g_i \vee \big(c_i \wedge \big((a \wedge (b \vee \overline{b})) \vee (\overline{a} \wedge b)\big)\big) \\ &= g_i \vee \big(c_i \wedge \big((a \wedge b) \vee \big((a \wedge \overline{b}) \vee (\overline{a} \wedge b)\big)\big)\big) \\ &= g_i \vee (c_i \wedge ((a \wedge b) \vee (a \oplus b))) \\ &= g_i \vee (c_i \wedge g_i) \vee (c_i \wedge p_i) = g_i \vee (c_i \wedge p_i) \end{aligned}$$

where we have made use of the fact that $a \vee b = (a \vee (\overline{a}) \wedge (a \vee b)$. □

Intuitively, (3.8) says that there will be a non-zero carry at stage $i + 1$ if either the generate signal is equal to one or there was a carry at stage i and it was propagated (or transfered) by this stage. Notice that implementing the carry-recurrence using the transfer signal leads to slightly faster adders than using the propagate signal, since an OR gate is easier to produce than an XOR gate [Par99]. Finally, notice that from (3.3) and (3.6), it follows that

$$s_i = p_i \oplus c_i \tag{3.9}$$

3.1.3 Carry-Save Adders (CSA)

A CSA is simply a parallel ensemble of n full-adders without any horizontal connection, i.e., the carry bit from adder i is not fed to adder $i+1$ but rather, stored as c'_i. In particular given three n-bit integers $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, and $C = \sum_{i=0}^{n-1} c_i 2^i$, their sum produces two integers $C' = \sum_{i=0}^{n} c'_i 2^i$ and $S = \sum_{i=0}^{n-1} s_i 2^i$ such that

$$C' + S = A + B + C$$

where:

$$s_i = a_i \oplus b_i \oplus c_i \tag{3.10}$$

$$c'_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) \tag{3.11}$$

where $c'_0 = 0$ (notice that (3.10) and (3.11) are nothing else but (3.1) and (3.2) re-written for different inputs and outputs). An n-bit CSA is shown in Figure 3.4. We point out that since the inputs A, B, and

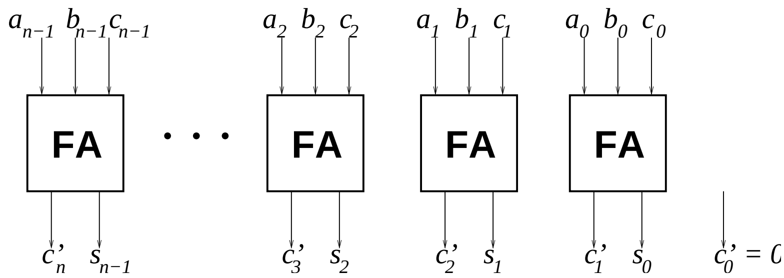


Figure 3.4. n-bit carry-save adder

C are all applied in parallel the total delay of a CSA is equal to that of a full-adder cell (i.e. the delay of (3.10) and (3.11)). On the other hand, the area of the CSA is just n-times the area of a FA cell and it scales very easily by adding more FA-cells in parallel. Subtraction can be accomplished by using 2's complement representation of the inputs. However, CSAs have two major drawbacks:

- Sign detection is hard. In other words, when an integer is represented as a carry-save pair (C', S) such that its actual value is $C' + S$, we may not know the sign of the total sum $C' + S$ unless the addition is performed in full length. In [KH91] a method for fast sign estimation is introduced and applied to the construction of modular multipliers.

- CSAs do not solve the problem of adding two integers and producing a single output. Instead, it adds three integers and produces two outputs.

We end this section with an example that illustrates the operation of a CSA.

Example 3.1. Let $A = 59 = (1\,1\,1\,0\,1\,1)_2$, $B = 61 = (1\,1\,1\,1\,0\,1)_2$, and $C = 43 = (1\,0\,1\,0\,1\,1)_2$. Then,

S and C' follow from (3.10) and (3.11) as:

$$\begin{array}{rcrcrrrrrrrr}
A & = & 59 & = & & & 1 & 1 & 1 & 0 & 1 & 1 \\
B & = & 61 & = & & & 1 & 1 & 1 & 1 & 0 & 1 \\
C & = & 43 & = & & & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
A+B+C & = & 163 & = & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
\hline
S & = & 45 & = & & & 1 & 0 & 1 & 1 & 0 & 1 \\
C' & = & 118 & = & & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
\hline
S+C' & = & 163 & = & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}$$

3.1.4 Carry-Delayed Adders (CDA)

Carry-delayed adders (CDAs) were originally introduced in [NS81] as a modification to the CSA paradigm. In particular, a CDA is a two-level CSA. Thus, adding $A = \sum_{i=0}^{n-1} a_i 2^i$, $B = \sum_{i=0}^{n-1} b_i 2^i$, and $C = \sum_{i=0}^{n-1} c_i 2^i$, we obtain the sum-pair (D, T), such that $D + T = A + B + C$, where

$$s_i = a_i \oplus b_i \oplus c_i \tag{3.12}$$

$$c'_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i) \tag{3.13}$$

$$t_i = s_i \oplus c'_i \tag{3.14}$$

$$d_{i+1} = s_i \wedge c'_i \tag{3.15}$$

with $c'_0 = d_0 = 0$. Notice that (3.14) and (3.15) are exactly the same equations that define a half-adder, thus an n-bit CDA is nothing else but an n-bit CSA plus a row of n half-adders. The overall latency is equal to the delay of a full-adder and a half-adder cascaded in series, whereas the total area is equal to n times the area of a full-adder and a half adder. The CDA scales in same manner as the CSA. Figure 3.5 depicts an n-bit CDA. We notice that t_i and d_{i+1} satisfy

$$d_{i+1} \wedge t_i = 0 \tag{3.16}$$

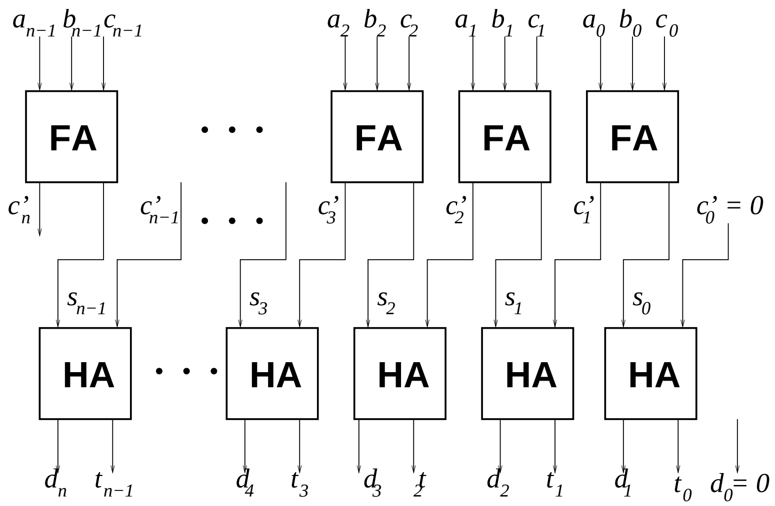


Figure 3.5. n-bit carry-delayed adder

for all $0 \leq i < n$. This property is easily proved as follows:

$$\begin{aligned} d_{i+1} \wedge t_i &= (s_i \wedge c_i') \wedge (s_i \oplus c_i') = s_i \wedge c_i' \wedge \Big((s_i \wedge \overline{c_i'}) \vee (\overline{s_i} \wedge c_i') \Big) = \\ &= (s_i \wedge c_i' \wedge \overline{c_i'}) \vee (s_i \wedge \overline{s_i} \wedge c_i') = 0 \end{aligned}$$

where we made use of fact that

$$s_i \oplus c_i' = (s_i \wedge \overline{c_i'}) \vee (\overline{s_i} \wedge c_i')$$

and of some basic Boolean algebra theorems. Property (3.16) of the CDA implies that either d_{i+1} or t_i or both are equal to zero, is exploited by Brickell in [Bri82] to reduce the complexity of a modular multiplier. We end this section with an example.

Example 3.2. Let $A = 43 = (1\,0\,1\,0\,1\,1)_2$, $B = 53 = (1\,1\,0\,1\,0\,1)_2$, and $C = 62 = (1\,1\,1\,1\,1\,0)_2$. Then,

S and C' follow from (3.10) and (3.11) as:

$$
\begin{array}{rcrccccccccc}
A & = & 43 & = & & & 1 & 0 & 1 & 0 & 1 & 1 \\
B & = & 53 & = & & & 1 & 1 & 0 & 1 & 0 & 1 \\
C & = & 62 & = & & & 1 & 1 & 1 & 1 & 1 & 0 \\
\hline
A+B+C & = & 158 & = & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
\hline
S & = & 32 & = & & & 1 & 0 & 0 & 0 & 0 & 0 \\
C' & = & 126 & = & & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
\hline
A+B+C & = & 158 & = & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
\hline
T & = & 94 & = & & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
D & = & 64 & = & & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
t_i \wedge d_{i+1} & & & = & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
T+D & = & 158 & = & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{array}
$$

3.1.5 Summary and Comparison

Section 3.1 described four different integer adders: ripple-carry adders, carry-lookahead adders, carry-save adders, and carry-delayed adders. The asymptotic complexity of the above adders is summarized in Table 3.1 and it is well known.

Table 3.1. Asymptotic area/time complexities of different n-bit adders.

Adder Type	Abbreviation	Area	Time
Ripple-Carry Adder	RCA	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Carry-Lookahead Adder	CLA	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$
Carry-Save Adder	CSA	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Carry-Delayed Adder	CDA	$\mathcal{O}(n)$	$\mathcal{O}(1)$

We, however, are interested in providing actual area estimates and, thus, asymptotics are not the right measure for this work. In terms of the components in Table 1.1, the area and delay of the RCA, CSA, and CDA are straight forward to estimate. However, the area of a CLA is very dependent on the actual implementation. Thus, we have used [NIO96] as a way of estimating the size and time delay of the CLA with respect to the RCA and CSA sizes[2]. The authors in [NIO96] have performed their study with the purpose of giving the practitioner insight into design trade-offs that can save power and enhance performance. As transistors shrink and wireless devices take over our daily lives, it is our opinion that power will probably be the optimization criteria of choice in most applications and thus, the relative size of the CLA with respect to RCA and CDA designs will be representative of current and future implementations. In addition, [NIO96] point out that by reducing area, power consumption is in general also reduced and therefore, area considerations are not completely left out in this study. Table 3.2 provides the area and delay characteristics of the RCA, CSA, CLA in [NIO96] for 16-bit, 32-bit, and 64-bit designs.

Table 3.2. Area/time complexities of RCA, CSA, and CLA according to [NIO96]. Area and delay are normalized with respect to estimated FA delay.

Adder Type	Area (transistors)			Area (normalized)			Time (nsec)			Time (normalized)		
	16-bit	32-bit	64-bit	16-bit	32-bit	64-bit	16-bit	32-bit	64-bit	16-bit	32-bit	64-bit
RCA	596	1204	2420	15.9	32.0	64.4	26.3	52.5	105	16.0	32.0	64.0
CLA	1038	2132	4348	27.6	56.7	115.6	11.0	14.2	15.8	6.7	8.7	9.6
CSA	1176	2360	4728	31.3	62.8	125.7	5.0	5.0	5.0	3.0	3.0	3.0
FA (estimate)	37.6			1.0			1.64			1.0		

We make the following observations regarding Table 3.2:

- We estimate the delay of one FA by dividing the 16-bit, 32-bit, and 64-bit RCA areas and delays by 16, 32, and 64, respectively, and averaging out the results. Our estimated FA delay is shown in the last row of Table 3.2. Based on these estimates, we calculate the normalized area and delay of the other designs.
- The area and delay of an n-bit RCA can be approximated as n FA and $n\,T_{\mathrm{FA}}$, respectively.
- The CSA design of [NIO96] includes two levels of FA. We assume that a *simple* CSA as the one presented in Section 3.1.3 requires the same resources as a RCA but half the delay of the CSA in [NIO96]. Thus, the area and delay of a *simple* CSA can be approximated as n FA and $1.5\,T_{\mathrm{FA}}$, respectively.
- The area and delay of an n-bit CLA can be approximated as $0.36n\lceil\log_2 n\rceil$ FA and $1.67\lceil\log_2 n\rceil\,T_{\mathrm{FA}}$, respectively.

Table 3.3 summarizes the above discussion and includes the time and area complexities which will be assumed in the following sections when integers adders are used.

Table 3.3. Area/time complexities of n-bit RCA, CLA, CSA, and CDA.

Adder Type	Area	Delay	Notes
RCA	n FA	$n\,T_{\mathrm{FA}}$	
CLA	$0.36\,n\log_2(n)$ FA	$1.67\ \log_2(n)\,T_{\mathrm{FA}}$	
CSA	n FA	$1.5\,T_{\mathrm{FA}}$	Needs additional RCA to generate final sum
CDA	n (FA + HA)	$T_{\mathrm{FA}} + T_{\mathrm{HA}}$	Needs additional RCA to generate final sum

3.2 $GF(p)$ Adders

As mentioned in the introduction, much of this chapter is devoted to an overview of RNS adders and multipliers, which traditionally have been used in digital signal processing applications. RNS adders, as it is the case with simple binary adders over $\mathbb{Z}$, are also key modules in the implementation of other modulo m arithmetic operations. In particular, subtracting $a - b \bmod m$ can be implemented as $a + (m - b) \bmod m$, thus only requiring a modulo adder [ST67], and multiplication modulo m can be implemented, in principle, with adders but we defer this discussion to Section 3.3.

Thus, the next sections are devoted to describing the implementation of modulo adders in hardware. Before continuing, we notice that in the RNS literature moduli have been divided into three general types: (i) moduli of the form 2^n, (ii) moduli of the form $2^n \pm 1$, and (iii) other moduli (i.e. moduli which are of no special form). In the remainder of this work, we are only concerned with the third type. Our emphasis is made on moduli of general form as they are the most interesting for our particular application, namely designing multipliers for fields $GF(p^m)$. Notice that the only primes of the form $2^n \pm 1$ between 2 and $2^{16} + 1$ are 7, 17, 31, 127, 257, 8191, and 65537. Thus, techniques specifically designed for these types of moduli can not be widely used in our context.

3.2.1 Table Look-Up Based Architectures

The simplest way to implement modular arithmetic, for moderate sizes of a modulus m, is to use look-up tables [ST67]. In particular to add (or multiply) two numbers modulo m, you need a table with $m^2 \leq 2^{2\lceil \log_2(m) \rceil}$ entries. In other words, you will need $m^2\lceil \log_2(m) \rceil$ bits of storage. According to [Jul78, BJM87b] the table look-up approach offers the best solution for high-speed realizations through pipelining. Bayoumi et al. [BJM87a] notice that the chip layout will have a direct effect on the final performance of the modular multiplier both in terms of area and speed. In particular, [BJM87a] develops a design methodology to obtain an optimized layout for table look-up-based modulo adders and multipliers. Reference [BJM87b] expands on the work of [BJM87a] and compares the table look-up based implementation to binary adder based and hybrid (combining table look-ups and combinatorial circuits) design methods and concludes that table look-up based methods are, both in terms of area and time performance, optimal for *general* moduli which can be represented with at most 5 bits. Table 3.4 summarizes the recommended implementation approach for moduli of different size according to [BJM87b].

Table 3.4. Recommended implementation approaches for different types of moduli according to [BJM87b]. LT : Look-Up Table, BA: Binary Adder, HY: Hybrid.

Modulo size (bits)	2–3		4		5		6		7		8 and up	
Modulo type	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time
$2^n \pm 1$	LT	LT	LT	LT	BA	LT	BA	HY	BA	HY	BA	HY
general	LT	LT	LT	LT	LT	LT	HY	HY	HY	HY	HY	HY

3.2.2 Combinatorial Architectures

Combinatorial architectures are based on the fact that the addition $A + B \bmod m$ can be performed as:

$$A + B \bmod m = \begin{cases} A + B & \text{if } A + B < m \\ A + B - m & \text{if } A + B \geq m \end{cases} \tag{3.17}$$

Such realization was already known in [Ban74] and it seems to date at least to 1967 [ST67]. Reference [BJM87b] reports on a VLSI implementation of this approach using two n-bit binary adders, where $m \leq 2^n$: the first adder computes $A + B$ while the second computes $A + B - m$. The carry bits of the first and second adders are combined together with an OR gate and this result is used by the multiplexer to select the correct output. The adders are implemented according to the carry-look-ahead paradigm. Figure 3.6 shows the structure of this adder and provides the reader with an example[3].

In [EB90], the authors propose a modulo adder based on carry-saved adders which accomplishes modular addition in constant time independently of the number of bits in the modulus. In particular, it requires at most 5 stages of n-bit CSA adders (for an n-bit modulus). Reference [BJS94] provides a

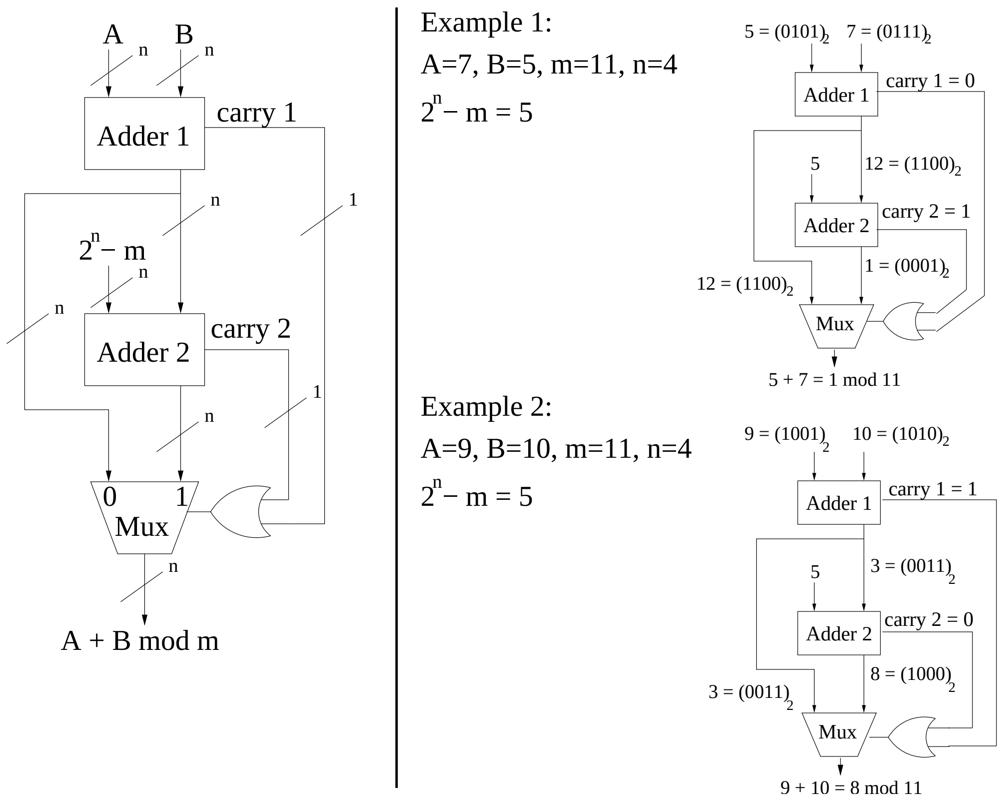


Figure 3.6. Binary adder-based RNS adder from Bayoumi [BJM87b]

formal proof of the previous statement. The adder is designed for medium to large size moduli. One problem with this adder is that it does not consider how to convert back from the CSA representation to the binary representation which, in general, requires a ripple carry-adder, thus incurring in additional delay. This fact effectively renders the advantages of the CSA approach from [EB90] impractical for normal RNS applications. It seems that it might only be convenient if one performs several additions while in CSA form before transforming back to the regular binary representation or in the final stage of a fast signal processing application for decoding and/or automatic error correction [BJS94].

We would like to point out that the same idea as in [EB90], was developed independently by [BJS94] but, this time, it is applied to an implementation of RNS arithmetic using a systolic array in the context of conversion from RNS to binary representation conversions, i.e., implementation of the Chinese Remainder Theorem (CRT). Finally, notice that a key characteristic of these architectures is the fact that certain residues modulo m do not have a unique representation. In other words, both A and $A + m$ are considered valid representations of $A \bmod m$ as long as $A + m < 2^n$, with $n = \lceil \log_2(m) \rceil$. Algorithm 3.1 summarizes the ideas of [EB90]. This type of adder seems well suited to large moduli as the delay will remain constant (i.e. 5 CSA stages [BJS94]).

Algorithm 3.1 5-stage algorithm for CSA RNS addition from [EB90] and [BJS94]

Input: $A = (A_C, A_S)$, $B = (B_C, B_S)$, and m with $A = A_C + A_S$, $B = B_C + B_S$, $2^{n-1} < m \leq 2^n$, and $0 \leq A, B < 2^n$.

Output: $R = (R_C, R_S) = A + B \bmod m$ with $R = R_C + R_S$ and $0 \leq R < 2^n$.

$(c_1, D_C, D_S) \leftarrow A_S + (B_C, B_S)$
$(c_2, R_C, R_S) \leftarrow A_C + (D_C, D_S)$
if $c_1 = 0$ AND $c_2 = 0$ **then**
  Return((R_C, R_S))
else if $c_1 + c_2 = 1$ **then**
  $(c_3, R_C, R_S) \leftarrow (2^n - m) + (R_C, R_S)$
else {if $c_1 = 1$ AND $c_2 = 1$}
  $(c_3, R_C, R_S) \leftarrow 2(2^n - m) + (R_C, R_S)$
end if
if $c_3 = 0$ **then**
  Return((R_C, R_S))
else
  $(c_4, R_C, R_S) \leftarrow (2^n - m) + (R_C, R_S)$
end if
if $c_4 = 0$ **then**
  Return((R_C, R_S))
else
  $(R_C, R_S) \leftarrow (2^n - m) + (R_C, R_S)$
  Return((R_C, R_S))
end if

Reference [Dug92] implements and compares three approaches to the layout of modulo adders based on binary adders. The first approach is the one described by [BJM87b]. The second approach is shown in Figure 3.7. In this approach only one binary adder is used to perform two cycles of addition. In the first cycle, the input multiplexers select inputs A and B to be added and, both, the sum and the carry which result from the addition are stored in the latch. In the second cycle of the addition, the input multiplexers pass the sum $A + B$, together with correction factor. The correct result is chosen, as in the modular adder from [BJM87b], based on the output of a gate which ORs the carries from the two additions. The third layout approach is shown in Figure 3.8. As in the second design, the modular adder uses only one binary adder for two cycles of addition. The second cycle of addition, however, needs

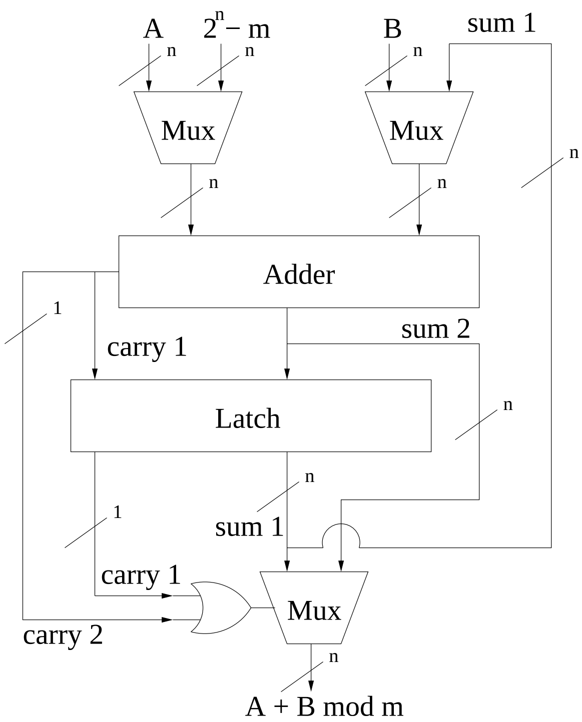


Figure 3.7. Type II modulo adder with feedback register from Dugdale [Dug92]

to be performed only if necessary. An ”add signal” selects inputs A and B to be added during the first cycle of addition. The sum and the carry of this addition are fed to an overflow detection. The add signal is disabled and the overflow signal now controls the output of the multiplexers to be added during the second cycle of addition. If an overflow is detected then the sum which resulted from the first addition cycle together with the correction factor are added together. Otherwise, the original inputs are added a second time. However, [Dug92] noticed that this approach could be modified to avoid performing the second addition if unnecessary. All adders in [Dug92] are implemented as carry-lookahead adders.

Table 3.5 summarizes the area and timing results from [Dug92] for the three designs. The author used 2 μm, single poly, double metal, static CMOS. In Table 3.5, Type I, Type II, and Type III correspond to the three different layouts that Dugdale describes in [Dug92]. We have added two columns to the this table which we have labeled AT for area-time product and $(AT)_N$ for normalized area-time product. The $(AT)_N$ values are obtained by dividing the AT values for the Type I and Type III layout approaches by the AT product of the Type II layout approach which is the one with the minimum AT value. We

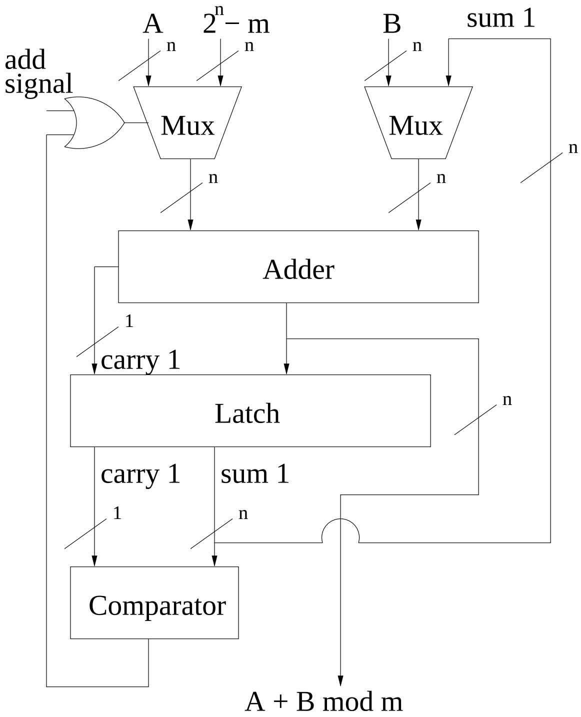


Figure 3.8. Type III modulo adder with overflow detection circuit from Dugdale [Dug92]

see that the Type II layout approach gives, on average, a 25 % improvement on the AT product over the Type I layout approach (i.e. the design from [BJM87b]) and an 8 % improvement over the Type III approach.

Table 3.5. Area-time complexities of the implementation of [Dug92] in 2 μm CMOS technology. Times are given in nsec and area in square microns $\times 10^3$

Modulo	Number of bits	Type I				Type II				Type III			
		A	T	AT	$(AT)_N$	A	T	AT	$(AT)_N$	A	T	AT	$(AT)_N$
5	3	261	34	8874	1.22	187	39	7293	1	183	43	7869	1.08
13	4	374	41	15334	1.25	267	46	12282	1	262	51	13362	1.09
29	5	452	51	23052	1.24	331	56	18536	1	324	61	19764	1.07
61	6	545	63	34335	1.27	399	68	27132	1	392	74	29008	1.07
97	7	643	69	44367	1.28	467	74	34558	1	456	82	37392	1.08
193	8	754	72	54288	1.28	550	77	42350	1	541	87	47067	1.11

Hiasat [Hia02] introduces a new design of modulo adder based on (3.17) and builds on ideas introduced by Piestrak [Pie94]. . The binary-based adder from [Hia02] is depicted in Figure 3.9. In [Pie94], the author notices that if $A + B < m$ then $A + B + 2^n - m < 2^n$ whereas if $A + B \geq m$ then $A + B + 2^n - m \geq 2^n$, where $0 \leq A, B < m$ and $2^{n-1} < m < 2^n$. Thus, based on the carry-out (c_{out})

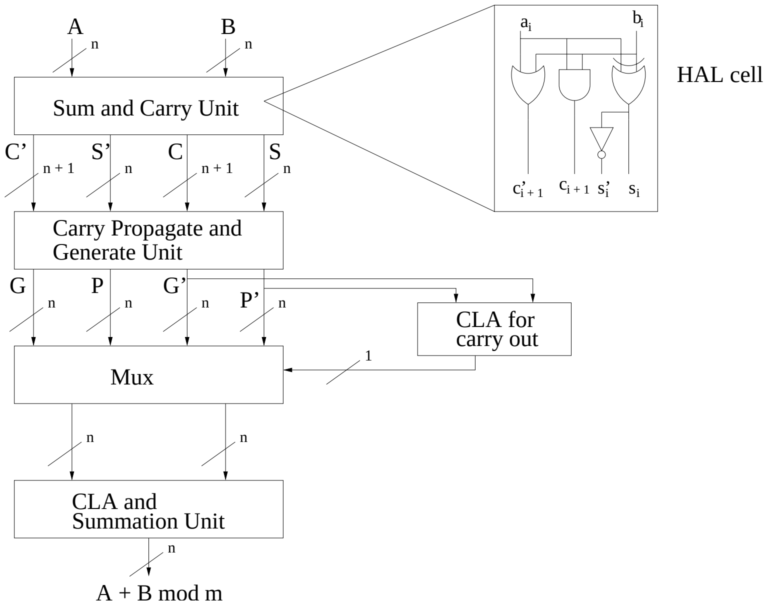


Figure 3.9. Binary-based RNS adder from Hiasat [Hia02]

of the operation $A+B+2^n-m$ (if $A+B+2^n-m<2^n$ then $c_{out}=0$, otherwise $c_{out}=1$) one can decide whether $A+B$ or $A+B+2^n-m$ is the right result for $A+B \bmod m$. Piestrak [Pie94] implements this idea in the context of residue-to-binary conversion circuits for RNS applications and uses a CSA and two CLA adders for his circuitry. Hiasat modifies this design and achieves significant savings in area and performance. The basic observation in [Hia02] is that if we use a CLA adder to implement the idea of [Pie94] we only need two sets of propagate, generate, and carry signals and a single adder circuit. In addition, one can further reduce the required hardware resources by sharing circuitry between the two Carry Propagate and Generate (CPG) units (the one corresponding to adding $X+Y$ and the one adding $X+Y+Z$). We make these ideas more explicit in the following.

The adder in [Hia02] consists of a Sum and Carry (SAC) unit, a CPG unit, a multiplexing unit, a CLA for c_{out}, and a CLA and Summation (CLAS) unit. In the following discussion we will denote by $Z=\sum_{i=0}^{n-1} z_i 2^i$ the constant 2^n-m, $A=\sum_{i=0}^{n-1} a_i 2^i$, and $B=\sum_{i=0}^{n-1} b_i 2^i$. The SAC unit outputs 4 signals corresponding to the carry and sum signals from $A+B$, denoted by $C=\sum_{i=0}^{n} c_i 2^i$, with $c_0=0$, and $S=\sum_{i=0}^{n-1} S_i 2^i$ and the carry and sum signals resulting from adding $A+B+Z$ which we denote $C'=\sum_{i=0}^{n} c'_i 2^i$, with $c'_0=0$, and $S'=\sum_{i=0}^{n-1} s'_i 2^i$. Reference [Hia02] notices that based on

(3.3) and (3.4), one can write if $z_i = 0$,

$$s_i = a_i \oplus b_i, \quad c_i = a_i \wedge b_i$$

and if $z_i = 1$

$$s'_i = \overline{a_i \oplus b_i}, \quad c'_i = a_i \vee b_i$$

A circuit which can produce the sum and carry, in both cases, i.e. when $z_i = 0$ and when $z_i = 1$, requires one XOR gate, one AND gate, and one Exclusive-NOR gate. Such a cell receives the name of half-adder-like (HAL) cell and it is also depicted in Figure 3.9. Hiasat also notices that the number of such cells will depend on the hamming weight of Z, thus, the SAC unit needs $HW(Z)$ HAL cells and $(n - HW(Z))$ normal half adder cells, where $HW(\cdot)$ denotes the hamming weight of the operand. The CPG unit is designed as explained in Section 3.1.2. Notice that the circuit in Figure 3.9 requires two CLA circuits, one for the sum $A + B$ and the second for the sum $A + B + Z$. The propagate and generate signals corresponding to the first sum are labeled P and G, whereas the ones corresponding to the second sum are labeled P' and G'. A maximum of $2n - 2$ half-adder (HA) cells are required to realize the CPG unit. However, as in the case of the SAC unit, the exact number depends on the hamming weight of Z. The more zeros in the binary representation of Z the more HA cells that can be shared between G and G' and P and P'. The CLA for the carry-out bit receives as inputs P' and G' and outputs one bit, the carry-out. The MUX requires a maximum of $(n - 1)$ 2×1 multiplexers to select between P and P' and similarly for G and G'. The number of multiplexers can be reduced depending on how many HA cells are shared by the CPG unit. The CLAS unit receives n carry bits and n propagate bits which can be combined according to (3.9) to form the final sum. We estimate the overall hardware complexity of Hiasat's adder as: one CLA, $(2HW(Z) - 1)$ MUX21, $(HW(Z) - 1)$ HA[4], and $HW(Z)$ 2-input OR gates and inverters[5].

3.2.3 Hybrid Architectures

In hybrid architectures the modulo adder is constructed using a combination of combinatorial circuits and table-lookups. One of the earliest modulo adder designs is due to Banerji [Ban74]. The author in [Ban74] notices that the set of residues modulo m form a finite cyclic group under addition modulo m and, thus, given $S = \{0, 1, \ldots, m-1\}$, adding $k \in S$ to any value in S corresponds to a permutation of S by k positions to the left. The area complexity of the adder can be further reduced by performing rotations only by powers of 2. In other words, if $k = \sum_{i=0}^{\lceil \log_2(m) \rceil} k_i 2^i$, where $k_i \in \{0, 1\}$, then you can perform a rotation by 2^i positions to the left whenever $k_i \neq 0$ and achieve the same final result. This reduction in area complexity is at the cost of a slower modular adder. The design is optimized for MSI/LSI technology. The operation of the adder is illustrated in the following example.

Example 3.3. Suppose that you want to add $4 + 5 \bmod 7$. Notice that $5 = 2^2 + 1$ and assume the availability of a register that has been preloaded with all residues modulo 7 as shown in Figure 3.10. Then, we successively rotate by one and by four positions to the left and obtain the final result of $9 \equiv 2 \bmod 7$ in register position number four. This operation is depicted in Figure 3.10.

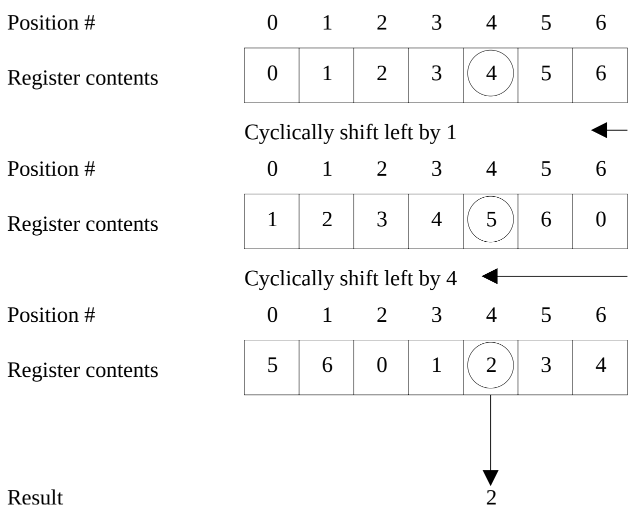


Figure 3.10. Addition $4 + 5 \bmod 7$ according to Banerji [Ban74]

The time complexity of this adder is estimated at $(\lceil \log_2(m) \rceil + 1)\Delta$, where Δ denotes a single gate delay. This is essentially given by the complexity of the shifter circuit. Although, the author does not give area estimates in *general*[6], it is easy to verify that barrel shifters have an area complexity which increases as $\mathcal{O}(m \log_2(m))$ (see for example [PSW02] for a nice overview of barrel shifters alternative hardware implementations and their corresponding time and area complexities). A subtracter can be implemented in a similar way but instead of using left shifters, one uses right shifters [Ban74].

To the author's knowledge, the only other reference which refers to the hybrid approach of implementing modulo adders is [BJM87b]. In [BJM87b], the authors use the binary adder to realize regular addition and table-lookup to correct the states which are larger than m. According to [BJM87b], this hybrid approach is only preferred for moduli which use more than six bits for their representation, as shown in Table 3.4.

3.2.4 Summary and Comparison

We summarize the complexity of the adders presented in this section. We notice that we will assume two possible table look-up implementations. The first one, referred here to as the full custom (FC) approach, allows the designer to implement ROM tables whose size is not a power of two, following [BJM87a]. The second one assumes that only ROMs whose size increases as a power of two are available. We refer to the second approach as the semi-custom (SC) approach. The SC approach is what most authors assume throughout the literature when estimating and comparing the costs of different $GF(p)$ architectures. Obviously, the second approach is faster to implement (there are already ROMs available so the engineer must only use them) but also more expensive in terms of area as there can be many unused entries. The second approach allows the practitioner to save in area but requires a lot of time in design work. We assume that a $3:1$ multiplexer requires twice as much area and delay as a $2:1$ multiplexer. All the other component complexities are taken from Table 1.1 and Table 3.3. Table 3.6 summarizes the complexities of the $GF(p)$ adder architectures discussed in this section.

Figures 3.11 and 3.12 show the area and time complexity behavior of the $GF(p)$ adders considered in this section. The area complexity increases exponentially for ROM-based adders whereas it increases logarithmically for combinatorial designs. Exception is the hybrid design from [BJM87b] whose behavior depends on the size of the prime moduli. The closer $2^n - m$ is to zero, the more the area complexity of the adder behaves logarithmically. On the other hand, the closer $2^n - m$ is to 2^n, the area com-

Table 3.6. Area/time complexities of different $GF(p)$ adders. We write $n = \lceil \log_2(m-1) \rceil$, where m is the modulus.

Adder Type	Building blocks	Area		Delay	
		Components	Normalized	Components	Normalized
n–bit table look-up $GF(p)$ adder (SC)	$(2^{2n}n)$–bit ROM	$2^{2n}n$ OR2	1.3 $2^{2n}n$	$n\,T_{\text{FA}}$	$1.1n$
n–bit table look-up $GF(p)$ adder (FC)	(m^2n)–bit ROM	m^2n OR2	1.3 m^2n	$n\,T_{\text{FA}}$	$1.1n$
n–bit hybrid $GF(p)$ adder from [BJM87b]	1 n–bit CLA + $(2^n-m)n$–bit ROM	0.36 $n\log_2(n)$ FA + $(2^n-m)n$ OR2	1.8 $n\log_2(n)$ + 1.3 $(2^n-m)n$	T_{CLA} + $n\,T_{\text{FA}}$	$1.84\log_2(n)$ + $1.1n$
n–bit CLA-based $GF(p)$ adder from [BJM87b]	2 n–bit CLAs + 1 n–bit 2:1 MUX + 1 OR2	0.72 $n\log_2(n)$ FA + n MUX21 + 1 OR2	$3.6n\log_2(n)$ + $2n$ + 1.3	$2\,T_{\text{CLA}}$ + T_{MUX21} + T_{OR2}	$3.67\log_2(n)$ + 1.8
n–bit 5-stage CSA-based $GF(p)$ adder from [EB90, BJS94]	5 n–bit CSAs + 1 n–bit 3:1 MUX + 2 n–bit MUX21	$5n$ FA + $4n$ MUX21	$33n$	4 T_{MUX21} + 7.5 T_{FA}	12.25
n–bit type II binary $GF(p)$ adder from [Dug92]	1 n–bit CLA + 3 n–bit MUX21 + n 1–bit LAT + 1 OR2	0.36 $n\log_2(n)$ FA + 3 n–bit MUX21 + n 1–bit LAT + 1 OR2	1.8 $n\log_2(n)$ + $8n$ + 1.3	$3\,T_{\text{MUX21}}$ + $2\,T_{\text{CLA}}$ + $2\,T_{\text{LAT}}$ + T_{OR2}	$3.67\log_2(n)$ + 5.8
n–bit binary $GF(p)$ adder from [Hia02]	1 n–bit CLA + $(2\,HW(2^n-m)+1)$ MUX21 + $(HW(2^n-m)-1)$ HA + $HW(2^n-m)$ (OR2 + NOT)	0.36 $n\log_2(n)$ FA + (2 $HW(2^n-m)$ $+1)$ MUX21 + $(HW(2^n-m)$ $-1)$HA + $HW(2^n$ $-m)$(OR2 + NOT)	$1.8n\log_2(n)$ + 8.2 $HW(2^n-m)$ -0.2	T_{CLA} + T_{MUX21} + T_{NOT}	$1.84\log_2(n)$ + 2

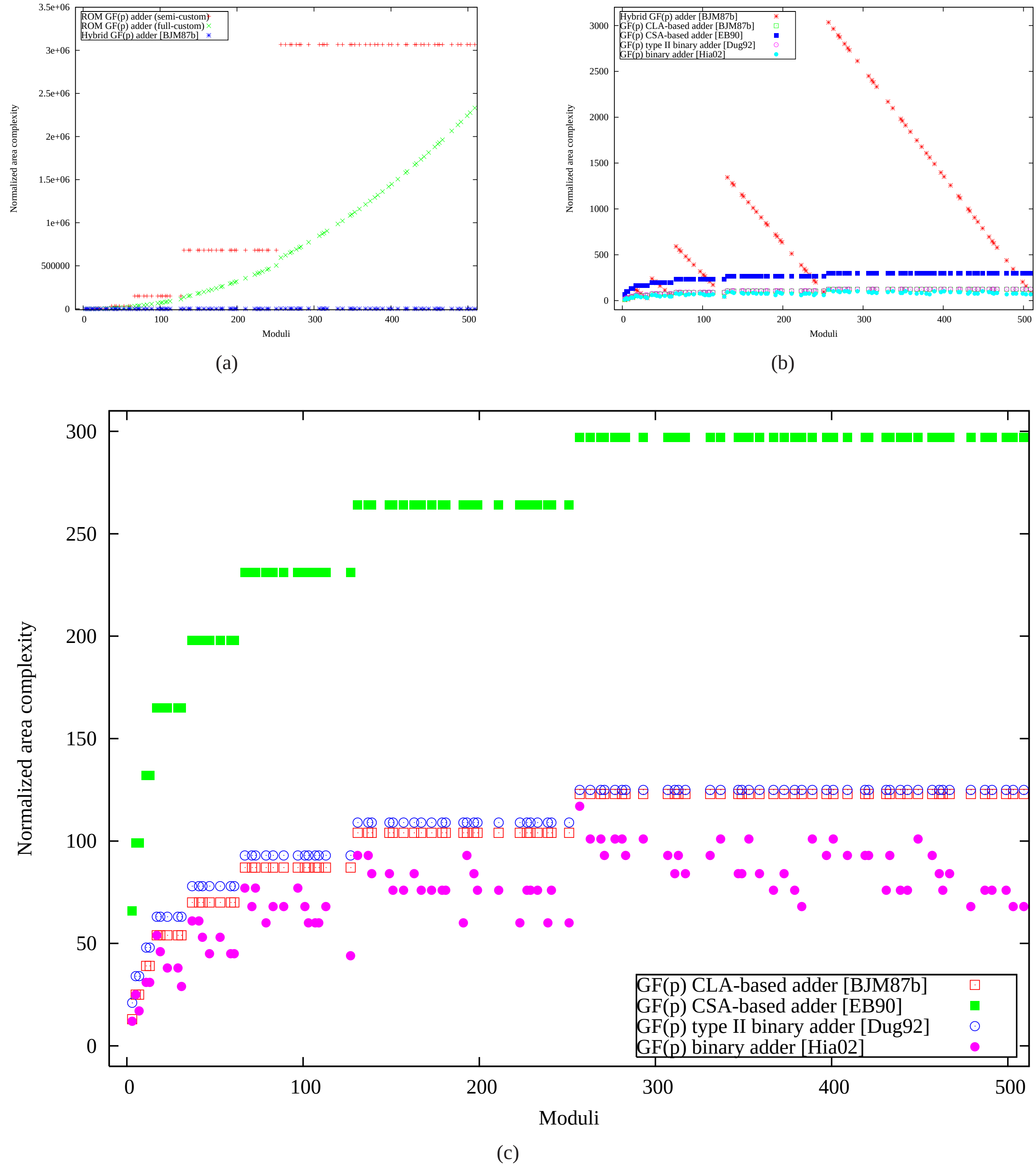


Figure 3.11. Normalized area complexity comparison of different $GF(p)$ adders. (a) ROM only vs. hybrid-base, (b) Hybrid-based vs. binary-adder-based and (c)Detail of binary-adder-based

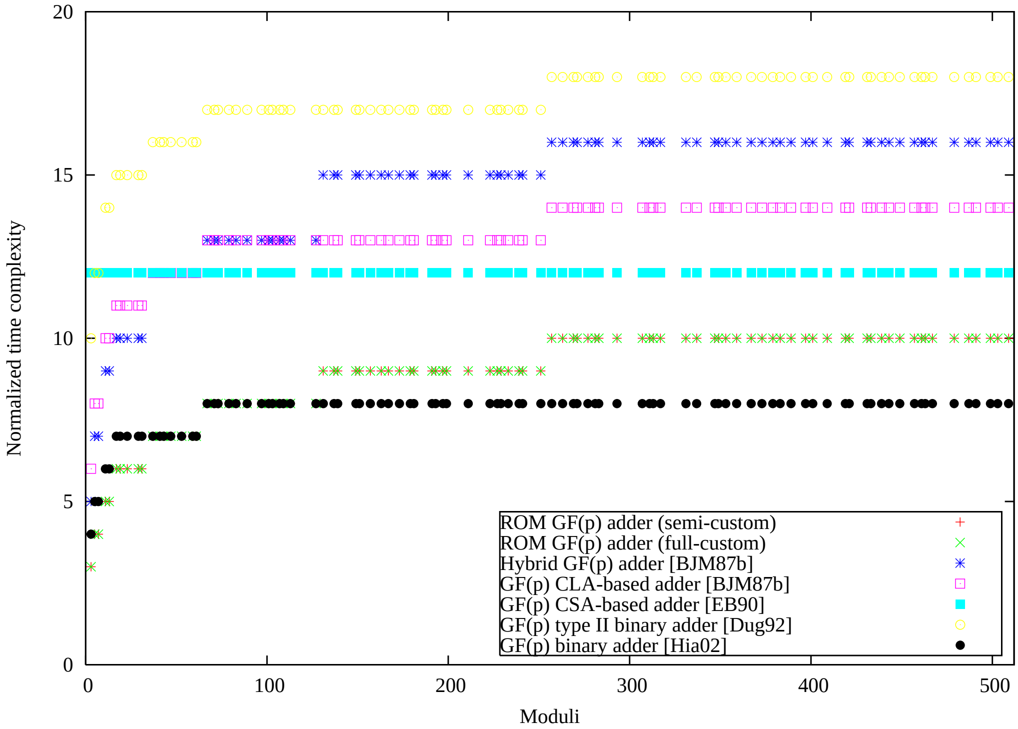


Figure 3.12. Normalized time complexity comparison of different $GF(p)$ adders.

plexity of the adder behaves exponentially, as expected. It can also be seen from Figure 3.11(c) that the area complexities of the CLA-based adder from [BJM87b], the type II binary adder from [Dug92], and the binary adder from [Hia02] are all comparable. Notice, however, that for most values of m the adder from [Hia02] has the best area complexity. With regards to time complexity, ROM-based designs together with the CLA-based design from [Hia02] show the best behaviors compared to other $GF(p)$ adders discussed in this section. For $m < 128$, ROM-based designs have the same or better time complexity than the design from [Hia02] whereas for $m \geq 128$, the adder from [Hia02] is the clear winner. Finally, notice that for the adders considered, the area/time product is essentially given by the area complexity of the adders. Thus, [Hia02] provides the best area/time trade-off of all the designs considered. For completeness, Appendix B includes tables where the normalized area, time, and area/time product complexities for all prime moduli $3 \leq p \leq 521$ are provided.

3.3 $GF(p)$ Multipliers

As in the case of modulo adders, we have also divided the multipliers according to the method of implementation. Thus, we have modulo multipliers based on table-lookups, hybrid architectures, and purely combinatorial circuits.

3.3.1 Table Look-Up and Hybrid Based Architectures

The naive method to implement modular multiplication via table look-ups would require $m^2\lceil\log_2(m)\rceil$ bits of storage as stated in Section 3.2.1. Early implementations based on table look-ups can be found in [ST67, Jul78]. However, several techniques have been developed to improve on these memory requirements. Jullien [Jul80] describes[7] the implementation of a modulo m multiplier taking advantage of the fact that there is an isomorphism between the multiplicative group $\mathbb{Z}_m^*$ and the additive group $\mathbb{Z}_{m-1}$. In particular, an element $A \in \mathbb{Z}_m^*$ can be represented as $A = G^e$ for some $e \in E$ and G a generator of $\mathbb{Z}_m^*$. Then, given $A = G^{e_1}, B = G^{e_2}$, with $A, B, C, G \in \mathbb{Z}_m^*$, the product $C \equiv A \cdot B \bmod m$ can be computed as:

$$C \equiv A \cdot B \bmod m = G^{e_1} \cdot G^{e_2} \bmod m = G^{e_1+e_2 \bmod m-1} \tag{3.18}$$

This multiplier has received the name of *index transform residue multiplier* or *index calculus multiplier*. From (3.18), we obtain the following steps to perform a modular multiplication:

Step 1. Find the index e_i for each number to be multiplied.

Step 2. Add the indexes modulo $m-1$.

Step 3. Perform the inverse index operation.

The above procedure replaces multiplication modulo m with addition modulo $m-1$ and it requires four table look-ups: two table look-ups to convert A and B to their index representation, one table look-up to perform addition of indexes modulo $m-1$, and one table look-up for the inverse index transform. The index calculus multiplier is shown in Figure 3.13.

Reference [Jul80] further simplifies the index transform method by noticing that we can perform the multiplication operation modulo m' with $m' = m_1 \cdot m_2 \geq 2m$. This restriction guarantees that the result

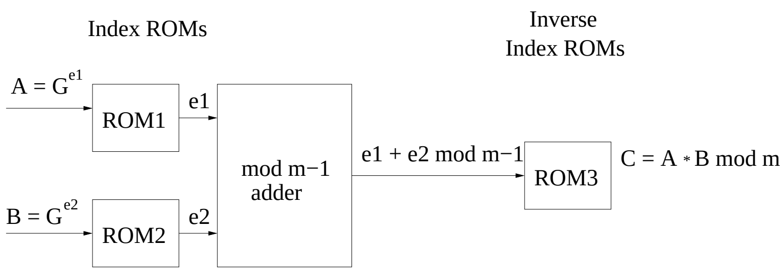


Figure 3.13. Index calculus multiplier

of multiplying two integers $A, B < m$ is less than m' and, thus, no overflow can occur. The advantage is that one can choose a modulo m' which minimizes the memory requirements of the modulo adders. Although theoretically multiplication by zero can not be performed using indexes (i.e. there is no valid index e_i for $A = 0$), [Jul80] notices that when using table look-ups one can solve the problem by adding an additional code to every ROM. According to [Jul80], this code should be greater than the largest possible index for every moduli in the factorization of $m - 1$. Since, this index will never appear as a valid submodular result, one can use it to represent the index of $A = 0$. Reference [Dug94] further simplifies the zero detection circuit by noticing that it is only necessary to include a special code in the index table corresponding to the smallest moduli in the factorization of $m - 1$. Figure 3.14 shows an implementation of a modulo 19 multiplier according to [Jul80] which incorporates the improved zero encoding of [Dug94]. Notice that the overall complexity of the multiplier with $m' = m_1 \cdot m_2$ and $m_1 < m_2$ and assuming full-custom VLSI designs[8] can be given as: $2(m(n_1' + n_2)) -$ bit ROMs $+$ $1(m_1 - 1) -$RNS adder$+1$ zero detection circuit$+1(m_2 - 1) -$RNS adder$+1(m_1(m_2 - 1)n) -$bit ROMs where $n_1' = \lceil \log_2(m_1) \rceil$, $n_2 = \lceil \log_2(m_2 - 1) \rceil$, and $n = \lceil \log_2(m - 1) \rceil$. Notice that in Figure 3.14, it is assumed that the adders are implemented using ROMs.

In [RY92] a new way to implement RNS multipliers via the index transform is presented. Reference [RY92] notices that one can use RNS and the Chinese Remainder Theorem (CRT) to further reduce the complexity of the modulo $m - 1$ adder in Figure 3.13. In other words, one can write $m - 1 = m_1 \cdot m_2 \cdots m_r$, where $\gcd(m_i, m_j) = 1$ for $i \neq j$, perform addition modulo m_i for $i = 1 \ldots r$, and finally recombine the results via the CRT[9]. As with all index-based RNS multipliers, both the conversion to the modular representation and back is done via ROMs. Since it is possible to have different

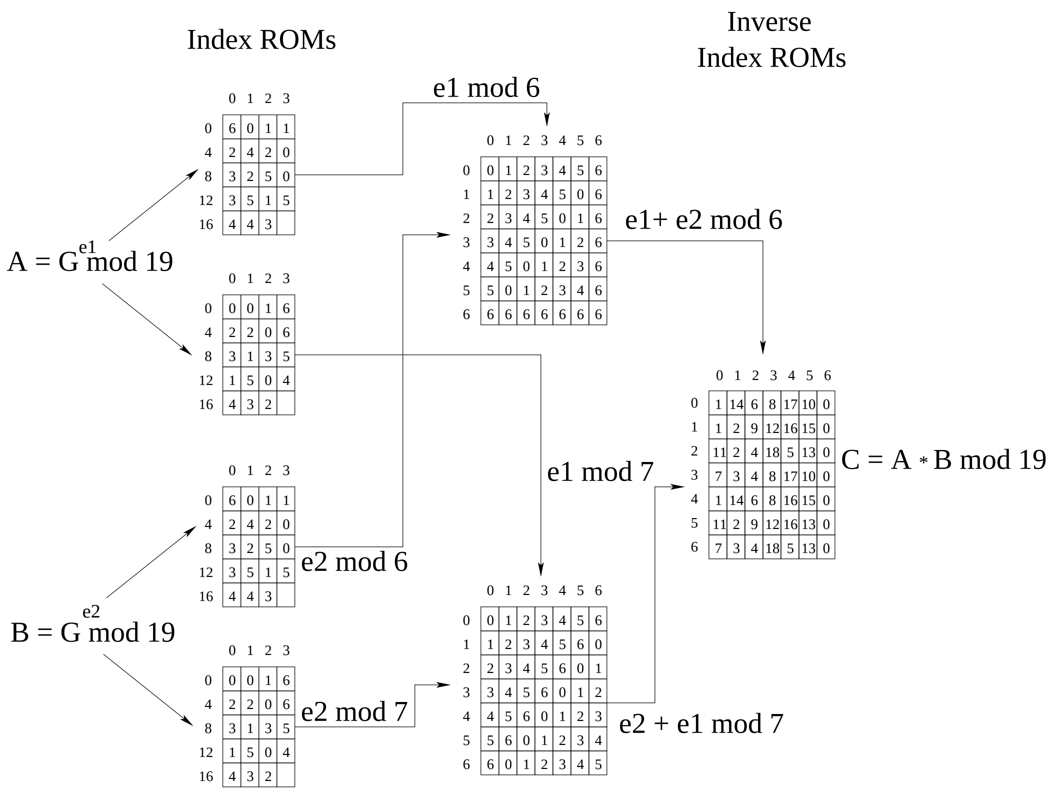


Figure 3.14. Index calculus modulo 19 multiplier from [Jul80] with zero encoding from [Dug94] and moduli set $\{6, 7\}$.

decompositions of $m-1$ in terms of the relatively prime moduli m_i's, [RY92] uses the following cost function to minimize the area/delay of the resulting RNS multiplier: $\sum_{i=1}^{r} \lceil \log_2(m_i - 1) \rceil$. Notice that minimizing the value of $\sum_{i=1}^{r} \lceil \log_2(m_i - 1) \rceil$ reduces the size of the index and inverse index ROMs. Similarly, smaller $\lceil \log_2(m_i - 1) \rceil$ imply smaller and faster adders. Depending on the design requirements one moduli set might be chosen over another. Moreover, certain moduli decompositions might result in minimum area but not minimum delay and vice versa. Figure 3.15 shows a diagram of the multiplier in [RY92]. The overall cost area cost of the multiplier in Figure 3.15 can now be estimated as: $2(m(\sum_{i=1}^{r} n_i)) -$ bit ROMs $+ 1(m_i - 1) -$ adder for each $i + 1((m-1)n) -$ bit ROM where $n_i = \lceil \log_2(m_i - 1) \rceil$.

We end this section by pointing out a second general methodology to implement ROM-based multipliers. This is known as the *quarter square residue multiplier*. Attributed in [Tay84] to [VP73] and independently discovered in [SF77], it is based on the observation that given a modulo m, multiplying

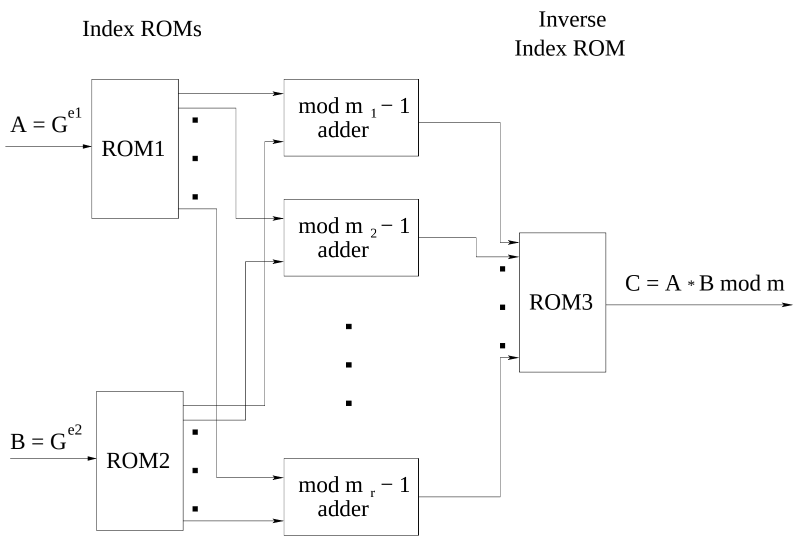


Figure 3.15. RNS multiplier from [RY92] with $m = m_1 \cdot m_2 \cdots m_r$

$A \cdot B \bmod m$ can be accomplished as:

$$\begin{aligned} C &= A \cdot B \bmod m \\ &= \left[\left((A+B)^2 \cdot 4^{-1} \bmod m\right) - \left((A-B)^2 \cdot 4^{-1} \bmod m\right)\right] \bmod m \end{aligned}$$

where m is assumed to be odd and the quantity $(\cdot)^2 \cdot 4^{-1} \bmod m$ is stored in a look-up table. However, from the literature it is apparent that other designs are better both in terms of area and time delay and, thus, the quarter square multiplier is not considered any further in this work.

3.3.2 Combinatorial Architectures

This section considers modulo multipliers based on combinational logic for *fixed* moduli. We emphasize that only architectures for fixed moduli have been considered. In addition, it would not be fair to compare architectures which can process multiple moduli to architectures optimized for a single modulus. To our knowledge, the best architectures for variable moduli in the context of RNS[10] is the one presented in [CPO95]. In [CPO95] Di Claudio et al. introduced the pseudo-RNS representation. This new representation is similar in flavor to the Montgomery multiplication technique as it defines an auxiliary modulus A relatively-prime to p. The technique allows building reprogrammable modulo multipliers, systolization, and simplifies the computation of DSP algorithms. Nevertheless, ROM-based solutions seem to be more efficient for small moduli $p < 2^6$ [CPO95].

In [SPSG97] Soudris et al. present full-adder (FA) based architectures for RNS multiply-add operations which adopt the carry-save paradigm. The paper concludes that for moduli $p > 2^5$, FA based solutions outperform ROM ones. Finally, [PKS01] introduces a new design which takes advantage of the non-occurring combinations of input bits to reduce certain 1-bit FAs to OR gates and, thus, reduce the overall area complexity of the multiplier. The multiplier outperforms in terms of area all previous designs. However, in terms of time complexity, the designs in [CPO95, RY92] as well as ROM-based ones outperform the multiplier proposed in [PKS01] for most prime moduli $p < 2^7$. Nevertheless, the combined time/area product in [PKS01] is always less than that of other designs.

Hiasat [Hia96] is the first to propose the design of modulo multipliers using the combinatorial logic approach by taking advantage of the fact that for any prime p there will always be $2^n - p$ *don't-care* positions in the truth table that defines the multiplier (where $n = \lceil \log_2 p \rceil$). In particular, one can build a truth table whose inputs are the bits of the multiplicand and multiplier and whose output corresponds to the output bits of the modular product. The truth table can then be given as input to Boolean minimization tools, such as ESPRESSO [BHH+82, BHMS84] and SIS [SSL+92], which, in turn, will output a two level logic implementation of the truth table. Reference [Hia96] only considers the *normal* approach to code the elements of $GF(p)$. For example, one considers the bit-strings $\{'000', '001', '010', '011', '100'\}$ to represent the numbers modulo 5, and the bit strings $\{'101', '110', '111'\}$ are simply used as *don't care* terms. We noticed, however, that one can choose a different representation. In other words, we could use the code words $\{'100', '101', '110', '111'\}$ to represent the set of integers $\{1,2,3,4\}$, and use the remaining *unused* bit-strings to represent the integer zero. Section 3.3.3 examines this idea in detail.

3.3.3 New $GF(p)$ Multipliers for $p < 2^5$

The following two sections are devoted to describing the methodology used to design area optimized $GF(p)$ multipliers for $p < 2^5$. The first part will concentrate exclusively on the $GF(3)$ case, which is of great interest as demonstrated by recent applications in the crypto community [Kob98, Sma99, BF01]. Our design methodology is generalized for odd prime fields, where $p < 2^5$. The work presented in this section appears in [GWP02][11].

Notation

We briefly present some basic notation and definitions used in the sequel. Let $I = \{0, 1\}$ and $O = \{0, 1, -\}$, then a logic function f in t input variables $x_{t-1}, x_{t-2}, \ldots, x_1, x_0$ and s output variables $y_{s-1}, y_{s-2}, \ldots, y_1, y_0$ can be defined as:

$$f : \; I^t \rightarrow O^s$$

where $X = [x_{t-1}, x_{t-2}, \ldots, x_1, x_0] \in I^t$ is the input and $Y = [y_{s-1}, y_{s-2}, \ldots, y_1, y_0] \in O^s$ is the output. We notice that in addition to the usual values of 0 and 1, the outputs y_i can also take on a *don't care* value $-$. Such functions are called incompletely specified logic functions.

An element $A \in GF(p)$, will be represented as a binary string $[a_{n-1}, a_{n-2}, \ldots, a_1, a_0]$ of length $n = \lceil \log_2 p \rceil$. We point out that the binary encoding of A does not necessarily imply a positional number system. Whenever we imply the representation of A in radix-2 notation we write $(A)_2 = (a_{n-1}, a_{n-2}, \ldots, a_1, a_0)_2$ explicitly. We refer to the radix-2 representation of A as the *natural* or *normal* encoding of A interchangeably throughout the text. For the purposes of this section, multiplication in $GF(p)$, is an incompletely specified logic function from I^{2n} to O^n. Here $X = [a_{n-1}, a_{n-2}, \ldots, a_1, a_0, b_{n-1}, b_{n-2}, \ldots, b_1, b_0] \in I^{2n}$ is the concatenation of the encodings of $A, B \in GF(p)$ and $C = [c_{n-1}, c_{n-2}, \ldots, c_1, c_0] \in O^n$, with $C = A{\cdot}B \bmod p$. Finally, we represent the logical AND and OR operators, by $\wedge$, $\vee$, and we will use a bar over a binary variable to denote logical negation (i.e. NOT$(a) = \overline{a}$).

$GF(3)$ Multiplier

As previously mentioned, [Hia96] only considers the *natural* encoding for $GF(p)$ elements. It is possible, however, to choose a different representation. We illustrate our approach by considering the case of $p = 3$.

Example 3.4. Let $p = 3$, then we have that $n = 2$ bits are required to represent any element of $GF(3)$. Using Hiasat's approach, one obtains the following Boolean equations to represent the product $C =$

$(c_1, c_0)_2 = A \cdot B \bmod 3$ with $A = (a_1, a_0)_2$, $B = (b_1, b_0)_2$ and $A, B, C \in GF(3)$.

$$c_1 = (a_1 \wedge b_0) \vee (a_0 \wedge b_1), \quad c_0 = (a_0 \wedge b_0) \vee (a_1 \wedge b_1) \tag{3.19}$$

This method allows for the implementation of this boolean function with four AND gates and two OR gates. However, one can do better. Notice that the above equations were obtained with the natural encoding. If instead we encode the integers 1 and 2 using the binary strings $'10'$ and $'11'$, respectively, and allow the binary strings $\{'00', '01'\}$ to both represent the integer 0, we can represent modulo 3 multiplication as shown in Table 3.7. Applying logic minimization to Table 3.7, one obtains the following

Table 3.7. Truth table representation of $C = A \cdot B \bmod 3$. The element 0 is represented as $'00'$ or $'01'$, 1 as $'10'$, and 2 as $'11'$

A $[a_1, a_0]$	B $[b_1, b_0]$	C $[c_1, c_0]$	A $[a_1, a_0]$	B $[b_1, b_0]$	C $[c_1, c_0]$
00	00	0–	10	00	0–
00	01	0–	10	01	0–
00	10	0–	10	10	1 0
00	11	0–	10	11	1 1
01	00	0–	11	00	0–
01	01	0–	11	01	0–
01	10	0–	11	10	1 1
01	11	0–	11	11	1 0

Boolean equations

$$c_1 = a_1 \wedge b_1, \quad c_0 = (\overline{a_0} \wedge \overline{b_0}) \vee (a_0 \wedge b_0) \tag{3.20}$$

Equation (3.20) can be realized with two NOT gates, three AND gates, and one OR gate. This represents an improvement of about 30% with respect to [Hia96] in the gate count if one does not take into account the circuits required to convert to and from our modified representation. At the end of this section, it is argued that for our particular application we can ignore such cost.

The optimized encoding of Example 3.4 was obtained by trying out all twelve possible encodings for $GF(3)$ elements in which we allow a redundant representation for the element 0 (i.e. one allows the element 0 to be represented by two different bit-strings[12]). In general, one has $\binom{2^n}{2^n-p+1}$ choices to encode the zero element of $GF(p)$ and $(p-1)!$ possible encodings for the non-zero elements. Multiplying these two numbers out one gets $\frac{2^n!}{(2^n-p+1)!}$ possible encodings.

Once an encoding has been chosen, one creates the corresponding truth table which is used as input to the well known Boolean minimization program ESPRESSO [BHH+82, BHMS84]. We use the SIS [SSL+92] program and its *script.algebraic* to find common terms in the Boolean functions obtained as output from ESPRESSO. One lesson we learned while performing this exhaustive search approach is that the encoding of the non-zero elements of $GF(3)$ does not matter for minimization purposes. Thus, for example, whether we encode the element 1 as $'10'$ and 2 as $'11'$ or vice versa, the same number of gates are required to implement the resulting Boolean functions. Assuming that the encoding of the non-zero elements of $GF(p)$ does not influence the complexity of the minimized Boolean functions, we are left with only $\binom{2^n}{2^n-p+1}$ possibilities to encode the zero element. This means, for example, that we would have to perform 70 Boolean minimizations for $p = 5$, 728 minimizations for $p = 11$, etc. It is obvious from the start that such a methodology is only applicable for the smallest of primes. It is, thus, necessary to develop efficient heuristics (or design criteria) that would allow a designer to choose *good* encodings, in other words, encodings that minimize our Boolean equations. The next section introduces a set of rules that allows one to find *good encodings* without trying out all possible ones.

Modular Multipliers for $p < 2^5$

Example 3.4 considered the particular case of $p = 3$ which is of the form $2^k + 1$. Notice that the encoding that produced the best results corresponded to representing an element $A \in GF(3) \setminus \{0\}$ using the *normal* binary representation of A' such that $A' = A + 1$ and the element zero is encoded as $'00'$ or $'01'$. For $p = 5$, we found $A' = A + 4$ to be optimum for $A \in GF(5) \setminus \{0\}$ and we represent zero with the strings in the set $\{'000', '001', '010', '011'\}$. A similar procedure can be applied to find a good encoding for the elements of $GF(17)$. This allows us to give our first heuristics.

Design Criterion 1 *Let $p = 2^k + 1$ be a prime. Then, we can decrease the area complexity of a combinatorial $GF(p)$ multiplier by encoding $A \in GF(p) \setminus \{0\}$ using the binary representation of A' with $A' = A + 2^n - 1$, $n = \lceil \log_2(p) \rceil$, and letting the remaining unused encodings to all represent the element zero. In addition, whenever A and/or B, in the multiplication $C = A \cdot B \bmod p$, assume any of the encodings of zero, the result C should be written as* $'0\underbrace{-\cdots-}_{n\, don't\, cares}'$.

Intuitively, it is easy to see that for primes $p = 2^k + 1$, the encoding of Criterion 1 allows one to detect the zero element by looking only at the first bit of the encoding of any $A \in GF(p)$. This is similar to the idea of diminished-1 representation, common in Number Theoretical Transform implementations. Unfortunately, the only primes $p = 2^k + 1 < 2^5$ are 3, 5, and 17. Thus, it is necessary to develop additional design criteria for primes of other forms.

In general, our experiments indicate that by choosing the encodings of the zero element such that the *don't care* terms (encodings of zero) are distributed evenly among the non-zero terms of $GF(p)$, the modulo multiplication function is minimized. We developed an easy method to accomplish this task. In particular, we built multiplication tables with $2^n \times 2^n$ entries (only $p - 1$ non-zero entries), and looked at how to divide them evenly into sections of non-zero values. Table 3.8 shows an example of the distribution of the zero-encodings that we found to be optimum for $p = 13$. This allows us to give

Table 3.8. *don't care* distribution of best encoding for $p = 13$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
1	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
2	•	•							•	•						
3	•	•							•	•						
4	•	•							•	•						
5	•	•							•	•						
6	•	•							•	•						
7	•	•							•	•						
8	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
9	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
10	•	•							•	•						
11	•	•							•	•						
12	•	•							•	•						
13	•	•							•	•						
14	•	•							•	•						
15	•	•							•	•						

our second design criterion.

Design Criterion 2 *Choose the zero-encodings in such a way that they are distributed as evenly as possible throughout the $2^n \times 2^n$ entry multiplication table.*

It is obvious that there are many such *evenly distributed* encodings which might minimize the multiplier function. Notice that we don't claim that we find the encoding that minimizes the multiplication function over all possible encodings. Rather, we give an efficient search method to produce encodings which will potentially result in functions with reduced area complexities when compared with [Hia96]. Our last heuristic is related to the way of choosing the encoding of the result of a multiplication by zero.

It is clear, that the encoding of a multiplication by zero has to belong to the set of zero encodings. However, which of the $2^n - p + 1$ possible encodings should one choose? This is particular interesting in the case in which the number of possible encodings is not a power of two. Criteria 3 and 4 provide us with guidelines to choose such encodings.

Design Criterion 3 *Let $D = 2^n - p + 1$ be equal to 2^s for some s. Then, choose sets of 2^s zero encodings by fixing s of the n possible bits to be equal to 0 and setting the remaining ones to be don't cares ($-$). If setting s bits equal to 0 does not reduce the area complexity of the resulting function, then set them equal to 1. If that still does not work, try combinations of 0's and 1's.*

Design Criterion 4 *Assume $D = 2^n - p + 1 \neq 2^s$. Then choose a set of $2^s < D$ zero encodings for the largest possible s and follow Criterion 3. Choose the remaining don't cares ($D - 2^s$) so as to satisfy Design Criterion 2. The encoding for any multiplication by zero result will correspond to the encoding of the set with 2^s elements.*

As a final remark, we notice that in some cases setting the *don't cares* to zero, can further minimize the area complexity of the resulting function.

Complexity of Proposed $GF(p)$ Multipliers

The criteria of Section 3.3.3 was used to find encodings that would reduce the complexity of $GF(p)$ multipliers for $p < 2^5$. Once the possible encodings were chosen[13], we used ESPRESSO and SIS as explained in Section 3.3.3. After Boolean minimization, we used SIS once more for technology mapping with the *stdcell2_2.genlib* CMOS cell library. Then, we chose the encoding that realizes the multiplier with the least area. The area reported by SIS is a relative figure obtained from the layout of the standard cells. We divided this value by the relative size of a pull-down or pull-up (a pull-down/pull-up is implemented with *one* transistor) according to the *stdcell2_2.genlib* library to obtain transistor counts. The number of transistors required to implement the logic functions for $2 < p < 2^5$, together with the encoding that gave the best results are summarized in Table 3.9. We have also implemented and mapped the modulo multipliers proposed in [Hia96] using the *stdcell2_2.genlib* library of SIS. We observe that

Table 3.9. Area (transistors) complexity of proposed $GF(p)$ multipliers with given encodings.

p	Area [Hia96]	Area new	%	ZERO element Encoding	Zero Encoding of Result
3	14	9	35.7	$(0)_2, (1)_2$	$'0-'$
5	65	42	35.4	$(0)_2, (1)_2, (2)_2, (3)_2$	$'0--'$
7	114	111	2.6	$(0)_2, (7)_2$	$'0\,0\,0'$
11	445	375	15.7	$(8)_2, (9)_2(10)_2,$ $(11)_2, (14)_2(15)_2,$	$'1-1-'$
13	566	520	8.1	$(0)_2, (1)_2(8)_2, (9)_2$	$'-00-'$
17	1048	907	13.5	$(0)_2, (1)_2(2)_2, (3)_2, (4)_2$ $(5)_2, (6)_2(7)_2, (8)_2, (9)_2$ $(10)_2, (11)_2(12)_2, (13)_2$ $(14)_2, (15)_2$	$'0----'$
19	1450	1343	7.4	$(0)_2, (2)_2(4)_2, (6)_2, (8)_2$ $(10)_2, (12)_2, (14)_2(16)_2,$ $(18)_2, (20)_2, (22)_2(24)_2,$ $(26)_2$	$'0---0'$
23	1984	1830	7.3	$(0)_2, (3)_2, (7)_2, (11)_2,$ $(15)_2, (19)_2, (23)_2,$ $(27)_2, (30)_2, (31)_2$	$'---11'$
29	3272	3136	4.2	$(4)_2, (12)_2, (20)_2, (28)_2$	$'00100'$
31	3806	3689	3.1	$(0)_2, (31)_2$	$'00000'$

in all cases our new encoding reduces the area complexity of the multiplier when compared to [Hia96]. Although, we do not do so explicitly here, when compared to other multipliers in the literature, our new design outperforms all previous ones except for the one presented in [PKS01].

Other Considerations

We notice that we have also implemented and mapped the circuits required to convert to/from our modified representation. The area requirements indicate that our design will only be area efficient in cases where several modulo p multipliers (for constant p) are needed but only a few conversion circuits are implemented. One such application is $GF(p^m)$ multipliers for cryptographic applications where one requires 160–1024 bit long operands. For example, the cryptosystem in [Kob98] works over the field $GF(3^{163})$. In such a field, one would require 163 $GF(3)$ multipliers to implement a $GF(3^{163})$ serial multiplier. If one uses a single conversion circuit, the conversion circuit would contribute less than one transistor per $GF(3)$ multiplier and thus, we could ignore it. Another important consideration is that in implementing a $GF(p^m)$ multiplier, we only need to change to and from our modified representation at the beginning and at the end of the field multiplication (i.e. at system input/output times). Addition and subtraction which are also required in a $GF(p^m)$ multiplier can also be done in our modified representation at no extra area penalty when compared to a normal $GF(p)$ adder/subtracter.

3.4 Notes and Further References

1. Notice that different authors use different definitions. We have followed the definitions of [Par99], however [Kor93] only defines two types of signals a generate signal, which is the same as the generate signal from [Par99], and a propagate signal which is equivalent to [Par99] transfer signal. The resulting carry recurrence relations are nevertheless the same.

2. The CDA is not considered in the study of [NIO96].

3. The design shown in Figure 3.6 is a modification of the original design in [BJM87b] due to Dugdale [Dug92]. In the original design instead of adding the constant $2^n - m$ in the second adder, one subtracts the modulus m (so the second adder acts as a subtracter), and only the carry from the second adder is considered by the multiplexer to choose the correct result.

4. These HAs come from the CPG unit and correspond to computing the propagate and generate signals for the sum $X + Y + Z$. It is $(HW(Z) - 1)$ and not $HW(Z)$ because we don't need to compute P_0, p_0, G_0, g_0 since by definition $P_0 = s_0, p_0 = s_0'$ and $G_0 = g_0 = 0$.

5. These OR2 gates and inverters correspond to the HAL cells.

6. The author in [Ban74] does provide area estimates in terms of equivalent gates for certain moduli m.

7. The idea of index table look-ups is attributed to Pollard [Pol71] by Jullien [Jul80]. Reference [RY92] attributes this idea to [Nak62] and [JL77] introduces the idea as background material without references.

8. If one can not use full-custom ROM designs, then it is assumed that only ROMs with $2^k r$-bits are available for some k and r. Thus, the overall area complexity of the multiplier in [Jul80] is $2(2^n(n_1' + n_2)) -$ bit ROMs $+ 1(m_1 - 1) -$ RNS adder $+ 1$ zero detection circuit $+ 1(m_2 - 1) -$ RNS adder $+ (2^{\lceil \log_2(m_1(m_2-1)) \rceil} n) -$ bit ROM.

9. The only primes p for which this method can not be applied are those of the form $2^n + 1$. In the range $2 \leq p \leq 2^{16} + 1$, 17, 257, and $2^{16} + 1$ are the only primes of this form.

10. Architectures for large and variable moduli such as those used in cryptographic applications are discussed in Chapter 4.

11. After presenting [GWP02], the author found out that a similar idea had been presented in [MGM99]. Interestingly enough, encoding symmetry is a key factor leading to area reduction in both works.

12. This is not to be confused with the technique of Redundant Number Representation used to limit carry propagation inherent in adder-based solutions for multipliers.
13. Following the design criteria of Section 3.3.3, we never had to try more than 25 encodings before finding one that was better than the encoding of [Hia96].

CHAPTER 4

Algorithms for Arithmetic in Large $GF(p)$ Fields

Most public-key schemes are based on modular exponentiation or repeated point addition. Both operations are in their most basic forms performed via the binary method for exponentiation or one of its variants [Gor98]. The atomic operation in the binary method for exponentiation is either modular multiplication, in the case of RSA and DL-based systems, or point addition, in the case of ECC, which in turn is performed through a combination of doublings and additions in the field of definition of the elliptic curve. In all cases, it is required that the moduli be at least 160-bit long for ECC-based systems or between 1024 and 2048-bit long for DL and RSA-based systems. Thus, this chapter surveys known methods and architectures to perform modular arithmetic with large moduli (160-bit to 2048-bit long integers). The purpose is just completeness as Chapter 3 dealt with a similar problem but for small moduli. Our focus is on algorithm developments rather than on specific architectures although, we also discuss certain implementations as examples of certain architectures.

Notation

We will refer to multi-precision integers with capital letters and to their digits in radix-b representation with lower-case letters. For example, we would write an n-digit integer in base b as $A = \sum_{i=0}^{n-1} a_i b^i$

with $b \geq 2$ and $0 \leq a_i < b$. Notice that unless otherwise stated we always assume unsigned operands.

4.1 Basic Methods

4.1.1 School-book Method for Modular Multiplication

The most naive method to perform modular multiplication is known as the multiply first and then divide method. In other words, to compute the product $Z = X \cdot Y \bmod M$, one first computes $Z' = X \cdot Y$ and then computes $Z = Z' \bmod M = Z' - \lfloor Z'/M \rfloor M$.

Multiplication

A multiplication algorithm can be derived by observing that for X and $Y = \sum_{i=0}^{n-1} y_i b^i$, the product $X \cdot Y$ can be written as:

$$X \cdot Y = \sum_{i=0}^{n-1} (X \cdot y_i) b^i = b(\cdots (b(0 + X \cdot y_{n-1}) + X \cdot y_{n-2}) + \cdots) + X \cdot y_0 \tag{4.1}$$

Algorithm 4.1 summarizes the multiplication operation (See [MvOV97, Section 14.2.3] for a more software oriented multiplication algorithm description). Algorithm 4.1 requires in every step a digit multi-

Algorithm 4.1 Multiplication

Input: $X, Y = \sum_{i=0}^{n-1} y_i b^i$
Output: $Z = X \cdot Y$

1: $Z \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad Z \leftarrow b \cdot Z + X \cdot y_{n-1-i}$
4: **end for**
5: Return(Z)

plication ($X \cdot y_i$), a multiplication[1] by b, and an adder. For $b = 2$ the algorithm reduces to left shifts by one bit and addition of X or 0, depending on the bit y_i. We also point out that if instead of initializing Z to 0 in Step 1 of Algorithm 4.1 we initialize it to $A2^{-(n-1)}$, then, the final result will be $Z = X \cdot Y + A$. This operation is useful in certain applications and it is obtained at virtually no extra cost. Finally, we notice that Algorithm 4.1 can also be re-written in terms of right-shift operations but we refer to [Kor93, Par99] for this and other hardware multipliers as these are not the most common methods to perform modular multiplication.

Division

Division can be fully characterized by the basic division equation

$$X = Q \cdot M + R, \qquad 0 \leq R < M \tag{4.2}$$

where X is the dividend, M is the divisor, Q is the quotient, and R is the residue. Just like multiplication can be expressed in terms of repeated additions and shifts, division can be written in terms of subtractions and shifts. Thus, our first version of division is called Shift/Subtract Division and it is summarized in Algorithm 4.2. At this point, several remarks are in order. First, the condition $X < 2^n M$ ensures

Algorithm 4.2 Restoring Shift/Subtract Division

Input: $M = \sum_{i=0}^{n-1} m_i 2^i$, $X = \sum_{i=0}^{2n-1} x_i 2^i < 2^n M$
Output: $X = Q \cdot M + R$ with $0 \leq R < M$ and $Q = \sum_{i=0}^{n-1} q_i 2^i$
1: $R \leftarrow X$
2: **for** $i = 0$ to $n-1$ **do**
3:   $R \leftarrow 2 \cdot R$
4:   $S \leftarrow R - 2^n M$
5:   **if** $S < 0$ **then**
6:     $q_{n-i} \leftarrow 0$
7:   **else**
8:     $q_{n-i} \leftarrow 1$
9:     $R \leftarrow S$
10:   **end if**
11: **end for**
12: $R \leftarrow R/2^n$
13: Return(Q, R)

that $Q < 2^n$. This is necessary since division of a $2n$-bit number by an n-bit number can generate a quotient which is larger than n-bits. This basically implements an overflow check. Second, the name *restoring* refers to the fact that if one guesses the next bit of the quotient wrong (by executing Step 4 of Algorithm 4.2 and obtaining a negative result), then one restores the residue R to its previous value. Restoring-dividers have timing issues which can be avoided by using non-restoring dividers [Par99].

In a non-restoring divider, one always performs Step 4 of Algorithm 4.2, however, depending on whether the result is negative or positive, one adds or subtracts $2^n M$. The result is shown in Algorithm 4.3. It is easy to see that Algorithm 4.3 produces the same result as Algorithm 4.2. For suppose that in iteration i of Algorithm 4.3, $R^{(i)} = 2 \cdot R^{(i-1)} - 2^n M < 0$ (i.e. Algorithm 4.2 would have generated $R^{(i)} = 2 \cdot R^{(i-1)}$ as a result), then in iteration $(i+1)$, $R^{(i+1)} = 2 \cdot R^{(i)} + 2^n M = 2(2 \cdot R^{(i-1)} - 2^n M) + 2^n M = 2^2 \cdot R^{(i-1)} - 2^n M$ which is the same result that Algorithm 4.2 would have generated.

Algorithm 4.3 Non-restoring Shift/Subtract Division

Input: $M = \sum_{i=0}^{n-1} m_i 2^i$, $X = \sum_{i=0}^{2n-1} x_i 2^i < 2^n M$
Output: $X = Q \cdot M + R$ with $0 \leq R < M$ and $Q = \sum_{i=0}^{n-1} q_i 2^i$

```
R ← X
for i = 0 to n − 1 do
  if R < 0 then
    q_{n−i} ← 0
    R ← 2 · R + 2^n M
  else
    q_{n−i} ← 1
    R ← 2 · R − 2^n M
  end if
end for
R ← R/2^n
Return(Q, R)
```

It is natural, to try to generalize Algorithms 4.2 and 4.3 to higher radixes. In this case, we could characterized the value of R at iteration i as:

$$R^{(i)} = b \cdot R^{(i-1)} - q_{k-i}(b^n M) \quad \text{with } R^{(0)} = X, \quad R^{(n)} = b^n R$$

where $Q = \sum_{i=0}^{n-1} q_i b^i$ and $0 \leq q_i < b$. The problem with higher-radix division methods is that of estimating q_i correctly. However, higher-radix division is outside the scope of this work and thus we refer to [Kor93, Par99] for a thorough treatment of the subject (See also [Knu81, MvOV97] for more software oriented descriptions).

4.1.2 Interleaved Multiplication Reduction Method

The details of this method are sketched in [Bla83, Slo85]. The method is based on combining (4.1) with modular reduction and making use of the distributivity property of modular reduction. Thus, (4.1) becomes:

$$\begin{aligned} X \cdot Y \bmod M &= \sum_{i=0}^{n-1} (X \cdot y_i) 2^i \bmod M \qquad (4.3) \\ &= 2(\cdots(2(0 + X \cdot y_{n-1} \bmod M) + X \cdot y_{n-2} \bmod M) + \\ &\quad \cdots) + X \cdot y_0 \bmod M \end{aligned}$$

where we have specialized (4.1) to the $b = 2$ case. Algorithm 4.4 follows easily from (4.3). We notice

Algorithm 4.4 Interleaved Multiplication Reduction Method

Input: $X, M, Y = \sum_{i=0}^{n-1} y_i 2^i$ with $X, Y < M$
Output: $Z = X \cdot Y \bmod M$

```
1: Z ← 0
2: for i = 0 to n − 1 do
3:     Z ← 2 · Z + X · y_{n−1−i}
4:     Z ← Z mod M
5: end for
6: Return(Z)
```

that restricting $X, Y < M$ does not have any practical impact as in most cryptographic applications that is a requirement. In addition, since $Z, X, Y < M$ at the beginning of every loop iteration i, we obtain that Z in Step 3 of Algorithm 4.4 is

$$Z^{(i)} = 2 \cdot Z^{(i-1)} + X \cdot y_{n-1-i} \leq 2(M-1) + (M-1) \leq 3M - 3$$

Thus, in Step 4 we need to subtract M at most twice from Z to obtain $Z \bmod M$. Step 4 can be easily achieved following Algorithm 4.5. The computation of $Z = X \cdot Y \bmod M$ following Algorithm 4.4 requires n steps. At each steps the following operations are performed:

- A left shift $(2 \cdot Z)$ in Step 3 and the conditional addition of A depending on whether y_{n-1-i} is equal to zero or one. Notice that the left shift operation can be simply achieved by wiring.

Algorithm 4.5 Modular Reduction for Step 4 of Algorithm 4.4

Input: $X, M, 0 \leq X \leq 3M - 3$
Output: $Z = X \bmod M$

$Z \leftarrow X - M$
if $Z \geq 0$ **then**
  $Y \leftarrow Z - M$
  if $Y \geq 0$ **then**
    $Z \leftarrow Y$
  end if
else
  $Z \leftarrow X$
end if
Return(Z)

- At most two subtractions for the computation of $Z \bmod M$ as indicated in Algorithm 4.5.

The crucial computations are the addition and subtraction operations, i.e., they need to be performed fast. They can be achieved following Omura's method [Omu90] introducing $O(n)$ delay or using a CSA which only introduces $O(1)$ delay. However, because in CSA sign information is not immediately available, one needs to perform fast sign detection in order to determine whether the product needs to be reduced modulo M or not. One such technique is introduced in [KH91].

4.2 Advanced Algorithms

4.2.1 Sedlak Modular Reduction

Originally introduced by Sedlak in [Sed87], this algorithm is used by Siemens, in the SLE44C200 and SLE44CR80S microprocessors, to perform modular reduction [NM96]. In the algorithm comparisons of X with $1/3, 1/6, \cdots$ of M play an important role in the final computation of $X \bmod M$. Assume that during the division of X by M, M was already shifted and subtracted from X several times. Furthermore, assume that the remainder has been stored in X. Sedlak's algorithm ensures that $|X| \leq M/3$.

This will also be true of the following steps. By assumption, M falls into one of the following ranges:

$$\begin{array}{ll}
\text{Range j=2} & \dfrac{M}{2\cdot 3} < |X| \leq \dfrac{M}{3} \\
\text{Range j=3} & \dfrac{M}{2^2\cdot 3} < |X| \leq \dfrac{M}{2\cdot 3} \\
\vdots & \vdots \\
\text{Range j=i} & \dfrac{M}{2^{(i-1)}\cdot 3} < |X| \leq 2^{-(i-2)}\dfrac{M}{2^{(i-2)}\cdot 3}
\end{array}$$

for some $i > 1$. Now M is shifted i bits to the right and for $M' = M/2^i$, we obtain:

$$\frac{2^i M'}{2^{(i-1)}\cdot 3} < |X| \leq \frac{2^i M'}{2^{(i-2)}3} \Rightarrow \frac{-M'}{3} < |X| - M' \leq \frac{M'}{3} \tag{4.4}$$

Equation 4.4 implies that $X' = X \pm M' \leq M'/3$ is satisfied again. Since, the above argument proves that the above condition is always met, M can always be shifted by at least $j = 2$ bits in every step of the algorithm. Sedlak notices that the expected value of i is 3 and therefore, the algorithm improves the reduction complexity by an average factor of $1/3$ when compared to the basic bit-by-bit reduction.

4.2.2 Barret Modular Reduction

Barret reduction was originally introduced in [Bar86], in the context of implementing RSA on a DSP processor. Algorithm 4.6 summarizes Barret's modular reduction. To clarify Algorithm 4.6, consider re-writing X as $X = Q \cdot M + R$ with $0 \leq R < M$, which is a well known identity from the division algorithm [MvOV97, Definition 2.82]. Thus

$$R = X \bmod M = X - Q \cdot M \tag{4.5}$$

Barret's basic idea is that one can write Q in (4.5) as:

$$Q = \lfloor X/M \rfloor = \left\lfloor \left(X/b^{n-1}\right)\left(b^{2n}/M\right)\left(1/b^{n+1}\right)\right\rfloor \tag{4.6}$$

Algorithm 4.6 Barret Modular Reduction

Input: $X = \sum_{i=0}^{2n-1} x_i b^i$, $M = \sum_{i=0}^{n-1} m_i b^i$, with $m_{n-1} \neq 0$, $\mu = \lfloor b^{2n}/M \rfloor$, $b > 3$
Output: $R = X \bmod M$

1: $Q_1 \leftarrow \lfloor X/b^{n-1} \rfloor$
2: $Q_2 \leftarrow Q_1 \cdot \mu$
3: $Q_3 \leftarrow \lfloor Q_2/b^{n+1} \rfloor$
4: $R_1 \leftarrow X \bmod b^{n+1}$
5: $R_2 \leftarrow Q_3 \cdot M \bmod b^{n+1}$
6: $R \leftarrow R_1 - R_2$
7: **if** $R < 0$ **then**
8: $\quad R \leftarrow R + b^{n+1}$
9: **end if**
10: **while** $R \geq M$ **do**
11: $\quad R \leftarrow R - M$
12: **end while**
13: Return(R)

and in particular Q can be approximated by

$$\widehat{Q} = Q_3 = \left\lfloor \left\lfloor \left(X/b^{n-1}\right) \right\rfloor \left(b^{2n}/M\right) \left(1/b^{n+1}\right) \right\rfloor$$

We notice that Q_3 can be at most 2 smaller than Q [MvOV97, Fact 14.43] and that the quantity $\mu = b^{2n}/M$ can be precomputed when performing many modular reductions with the same modulus, as is the case in cryptographic algorithms. Finally, Step 11 in Algorithm 4.6 is repeated at most twice [Bar86].

From the efficiency point of view, notice that all divisions by a power of b are simply performed by right-shifts and modular reduction modulo b^i, is equivalent to truncation. The complexity of Algorithm 4.6 is basically given by the number of multiplications. We notice that there are only two multi-precision multiplications: one to compute Q_2 (Step 2) and one to compute R_2 (Step 5). Both are “partial” multiplications, i.e., we don’t need to compute all digits of the result. In the case of Q_2 the $n-1$ least significant digits need not to be computed and in the case of R_2 only the $n+1$ significant digits are needed. It can be shown that Algorithm 4.6 needs at most $\frac{(n^2+5n+2)}{2} + \binom{n+1}{2} + n = n^2 + 4n + 1$ single-precision multiplications (where single-precision multiplication means multiplication of two digits)[MvOV97, Note 14.45]. We refer to [MvOV97, Section 14.3.3] for further discussion of implementation issues regarding Barret reduction.

Improved Barret Algorithm

Barret's algorithm can be further improved as shown in [Dhe94, Dhe98]. The basic idea is to re-write the quotient in (4.6) as:

$$Q = \lfloor X/M \rfloor = \left\lfloor \frac{\frac{X}{2^{n+\beta}} \frac{2^{n+\alpha}}{M}}{2^{\alpha-\beta}} \right\rfloor$$

where Barret's algorithm in radix $b = 2$ corresponds to the case $\alpha = n$ and $\beta = -1$. Then, the quotient can be estimated as

$$\widehat{Q} = \left\lfloor \frac{\left\lfloor \frac{X}{2^{n+\beta}} \right\rfloor \left\lfloor \frac{2^{n+\alpha}}{M} \right\rfloor}{2^{\alpha-\beta}} \right\rfloor$$

Then, for a given modulus M, $\mu = \frac{2^{n+\alpha}}{M}$ can be precomputed. It is shown in [Dhe98], that

$$\left\lfloor \frac{X}{M} \right\rfloor - 2^{\gamma-\alpha} - 2^{\beta+1} - 1 + 2^{\beta-\alpha} < \widehat{Q} \leq \left\lfloor \frac{X}{M} \right\rfloor \qquad (4.7)$$

for some $\gamma > 0$. Equation 4.7 implies that the estimated quotient $\widehat{Q}$ is always smaller or equal to the real quotient and that one needs to choose α, β, and γ to minimize the error of the estimate $\widehat{Q}$. In particular, [Dhe98] shows that to minimize the error one must choose $\beta \leq -2$ and $\alpha > \gamma$. Following [Dhe98], the error is at most 1, thus improving over Barret's algorithm (Algorithm 4.6) where $\widehat{Q}$ could be at most 2.

4.2.3 Brickell's Modular Reduction

Brickell's method, originally introduced in [Bri82], is dependent on the utilization of carry-delayed adders (CDAs) as introduced in Section 3.1.4. Assume that X is in CDA form, then it can be written as:

$$X = \sum_{i=0}^{n-1} (xd_i + xt_i)\, 2^i \qquad (4.8)$$

where $xd_0 = 0$. It follows that $Z = X \cdot Y$ can be computed as:

$$\begin{aligned} Z &= X \cdot Y = Y \cdot \sum_{i=0}^{n-1} (xd_i + xt_i)\, 2^i \\ &= (xt_0 \cdot Y)\, 2^0 + (xt_1 \cdot Y + xd_1 \cdot Y)\, 2^1 + \cdots + (xt_{n-1} \cdot Y + xd_{n-1} \cdot Y)\, 2^{n-1} \end{aligned}$$

Re-grouping terms, we obtain:

$$\begin{aligned} Z &= \left(2^0 \cdot xt_0 \cdot Y + 2^1 \cdot xd_1 \cdot Y\right) + \left(2^1 \cdot xt_1 \cdot Y + 2^2 \cdot xd_2 \cdot Y\right) + \cdots \\ &\quad + \left(2^{n-2} \cdot xt_{n-2} \cdot Y + 2^{n-1} \cdot xd_{n-1} \cdot Y\right) + \left(2^{n-1} \cdot xt_{n-1} \cdot Y\right) \end{aligned}$$

Finally, using (3.16), we see that at each step we will only need to add a shifted version of Y to the accumulated value Z. Reduction modulo M can be accomplished by subtracting shifted versions of M from the result Z. However, Brickell suggests to interleave multiplication and reduction steps. The result is shown in Algorithm 4.7. We make the following remarks about Algorithm 4.7:

Algorithm 4.7 Brickell's Modular Reduction

Input: $X = (Xd, Xt)_{CDA} = \sum_{i=0}^{n} xd_i 2^i + \sum_{i=0}^{n-1} xt_i 2^i$ with $xd_0 = 0$, $K = 2^n - M$ with $2^{n-1} \leq M < 2^n$, $b_1, b_2 \in \{0, 1\}$, $R = (Rd, Rt)_{CDA} = \sum_{i=0}^{n+10} (rd_i + rt_i) 2^i$ with $rd_0 = 0$

Output: $R = X \cdot Y \bmod M$

$R \leftarrow 0$, $t_1 \leftarrow 0$, t_2
for $j = 1$ to $n + 10$ **do**
  $B' \leftarrow xt_{n-1} \cdot B + xd_n \cdot 2B$
  $K' \leftarrow b_2 \cdot 2^{11} K + b_1 \cdot 2^{10} K$
  $R \leftarrow 2 \cdot (R + B' + K')$
  $X \leftarrow 2 \cdot X$
  Add the 4 MSbits of R to the 4 MSbits of $2^{11}K$
  if overflow is detected **then**
    $b_2 \leftarrow 1$
  else
    $b_2 \leftarrow 0$
  end if
  Add the 4 MSbits of R to the 3 MSBits of $2^{10}K$.
  if overflow is detected AND $t_2 = 0$ **then**
    $b_1 \leftarrow 1$
  else
    $b_2 \leftarrow 0$
  end if
end for
Return(R)

- In Algorithm 4.7, the notation $X = (Xd, Xt)_{CDA}$ implies that X is represented in the carry-delayed manner, as explained in Section 3.1.4.

- Brickell's reduction method uses basically a combination of a sign estimation technique (See for example [KH91]) and Omura's modular reduction [Omu90]. In particular, one estimates the sign of the accumulator R by looking at its four most significant bits. Since the sign is used to decide whether or not to subtract a multiple of M, the error is only by a factor of M which can be later corrected.

- Omura's method is based on the following observation. When performing a modular addition, say modulo M, one can allow a temporary register R to grow larger than M but keep it always less than an upper bound 2^k. Whenever the sum $R + C \geq 2^k$ with $R, C < 2^k$, one can instead compute $(R + C \bmod 2^k) + K$, where $K = 2^k - M$ is a pre-computed constant. In other words, whenever there is a carry-out from the sum $R + C$, one can ignore it and simply add the constant K.

- b_1 and b_2 are two bits used as control bits to subtracts multiples of K from the accumulator R.

- Carry-delayed register X, consists of an n-bit register Xt and an $(n + 1)$-bit register Xd, so that bit xd_n can be stored during the left shift operation in Step 6.

- R, the accumulator, is an $(n+11)$-bit carry-delayed register. In Step 5, it is possible that overflow occurs. However, it is shown in [Bri82] that if the overflow bits are ignored, one still obtains $R \equiv 2 \cdot (R + B' + K') \bmod M$, which is the purpose of the algorithm in the end.

- Algorithm 4.7, assumes $2^{n-1} \leq M < 2^n$. If $M < 2^{n-1}$, [Bri82] shows how to compute $X \cdot Y \bmod M$ as follows. Let $2^{s-1} \leq M < 2^s$ and $e = n - s$. To find $X \cdot Y \bmod M$, simply find $R \equiv A \cdot (B \cdot 2^e) \bmod (2^e M)$. This is possible since $2^{n-1} \leq 2^e M < 2^n$. It can be shown that $2^e | R$ and that $D/2^e \equiv X \cdot Y \bmod M$.

4.2.4 Quisquater's Modular Reduction

Quisquater's algorithm, originally introduced in [Qui90, Qui92], can be thought of as an improved version of Barret's reduction algorithm. [BD95, Wal91] have proposed similar methods. In addition, the method is used in the Phillips smart-card chips P83C852 and P83C855, which use the CORSAIR crypto-coprocessor [DQ90, NM96] and the P83C858 chip, which uses the FAME crypto-coprocessor [FMM+96].

Quisquater's algorithm, as presented in [DQ90], is a combination of the interleaved multiplication reduction method (Algorithm 4.4) and a method that makes easier and more accurate the estimation of the quotient Q in (4.5). We re-write Algorithm 4.4 for the case of general radix b as Algorithm 4.8.

Algorithm 4.8 Interleaved Multiplication Reduction Method for General Radix b

Input: $X, M, Y = \sum_{i=0}^{n-1} y_i b^i$ with $X, Y < M$ and $0 \leq y_i < b$
Output: $R = X \cdot Y \bmod M$

1: $R \leftarrow 0$
2: **for** $i = 0$ to $n-1$ **do**
3: $\quad R \leftarrow b \cdot R + X \cdot y_{n-1-i}$
4: $\quad R \leftarrow R \bmod M$
5: **end for**
6: Return(R)

Step 4 of Algorithm 4.8 can then be performed following (4.5). In particular, assume that we want to compute $R = X \bmod M = X - Q \cdot M$, then the quotient Q can be written as:

$$Q = \left\lfloor \frac{X}{M} \right\rfloor = \left\lfloor \frac{X}{2^{n+c}} \cdot \frac{2^{n+c}}{M} \right\rfloor$$

From the above, we can write

$$\widehat{Q}\delta = \left(\left\lfloor \frac{X}{2^{n+c}} \right\rfloor \right) \cdot \left(\left\lfloor \frac{2^{n+c}}{M} \right\rfloor \right) \tag{4.9}$$

where $\widehat{Q}$ is an approximation of the quotient Q. Thus, (4.9) allows us to write an approximation for $R = X \bmod M$ as

$$\widehat{R} = X - \widehat{Q} \cdot M' = X - \left\lfloor \frac{X}{2^{n+c}} \right\rfloor \cdot M'$$

where we effectively are performing a reduction modulo $M' = \delta M = \lfloor 2^{n+c}/M \rfloor M$. We notice that M' has its most significant c bits equal to 1 and that the computation of the approximate quotient $\widehat{Q}$ is immediate, i.e., it is just the most significant bits of X. Since the objective of the modular reduction is to obtain $R = X \bmod M$, we need a way to obtain R from $\widehat{R}$ which is known in the literature as de-normalization[2].

Since we have reduced modulo $M' = \delta M$, a multiple of M, we have that $R = X \bmod M =$

$\widehat{R} \bmod M = (X \bmod M') \bmod M$. Notice that we can write $\delta\widehat{R} \bmod M'$ as:

$$\delta\widehat{R} \bmod M' = \left[\delta\left(X - \left\lfloor \frac{X}{M'} \right\rfloor M'\right)\right] \bmod M' = \delta X - \left\lfloor \frac{\delta X}{M'} \right\rfloor M' = \delta X - \left\lfloor \frac{X}{M} \right\rfloor (\delta M) \quad (4.10)$$

Thus, from (4.10), we obtain the following relation to obtain R from $\widehat{R}$:

$$R = \frac{\left(\delta \cdot \widehat{R}\right) \bmod M'}{\delta}$$

The only step that is left is to compute $\delta = \lfloor 2^{n+c}/M \rfloor$. It is shown in [DJQ97] that one can approximate δ within 1 as indicated in Theorem 4.1.

Theorem 4.1. *[DJQ97] Let $k = n - c - 2$ and let $M = \sum_{i=0}^{n-1} m_i 2^i$. Putting $\widehat{M} = \sum_{i=k}^{n-1} m_i 2^{i-k}$ if we define*

$$\hat{\delta} = \left\lfloor \frac{2^{2c+2}}{\widehat{M}} \right\rfloor$$

then $\delta \leq \hat{\delta} < \delta + 1$.

Theorem 4.1 implies that with only one test we can obtain the exact value of the constant δ and, more importantly, we only need the $(c+2)$ most significant bits of M to do so. Further, details about the algorithm and the choice of δ can be found in [Dhe98].

4.3 Montgomery Modular Reduction

The Montgomery algorithm, originally introduced in [Mon85], is a technique that allows efficient implementation of the modular multiplication without explicitly carrying out the modular reduction step. The Montgomery reduction algorithm is shown in Algorithm 4.9. The idea behind Montgomery's algorithm is to transform the integers in M-residues and compute the multiplication with these M-residues. At the end, one transforms back to the normal representation. As with Quisquater's method, this approach is only beneficial if we compute a series of multiplications in the transform domain (for example, in the case of modular exponentiation). Notice that Algorithm 4.9 is just the reduction step involved in a modular multiplication. The multiplication step can be accomplished, for example, with Algorithm 4.1.

Algorithm 4.9 Montgomery Reduction

Input: $0 \le T < R \cdot M$, $R > M$, $\gcd(R, M) = 1$, and $R \cdot R^{-1} - M \cdot M' = 1$
Output: $Z = T \cdot R^{-1} \bmod M$

1: $Q \leftarrow (T \bmod R)\, M' \bmod R$
2: $Z \leftarrow \frac{T + Q \cdot M}{R}$
3: **if** $Z \ge M$ **then**
4: $\quad Z \leftarrow Z - M$
5: **end if**
6: Return(Z)

To see that Z in Step 2 is an integer, observe that $Q = T \cdot M' + k \cdot R$ and $M \cdot M' = -1 + l \cdot R$, for some integers k and l. Then, $(T + Q \cdot M)/R = (T + (T \cdot M' + k \cdot R)M)/R = l \cdot T + k \cdot M$.

In practice R in Algorithm 4.9 is a multiple of the word size of the processor and a power of two. This means that M, the modulus, has to be odd (because of the restriction $\gcd(M, R) = 1$) but this does not represent a problem as M is a prime or the product of two primes (RSA) in most practical cryptographic applications. In addition, choosing R a power of 2 simplifies Steps 1 and 2 in Algorithm 4.9, as they become simply truncation (modular reduction by R in Step 1) and right shifting (division by R in Step 2). Notice that $M' \equiv -M^{-1} \bmod R$. In [DK90] it is shown that if $M = \sum_{i=0}^{n-1} m_i b^i$, for some radix b typically a power of two, and $R = b^n$, then M' in Step 1 of Algorithm 4.9 can be substituted by $m_0' = -M^{-1} \bmod b$. The authors of [DK90] notice that although the resulting sum $T + A \cdot M$ (in Step 2 of Algorithm 4.9) might not be the same, the effect is, namely making $T + A \cdot M$ a multiple of R.

As with previous algorithms, one can interleave multiplication and reduction steps. The result is Algorithm 4.10 where the trick from [DK90] is also used. In Algorithm 4.10, we have made use of the fact that $m_0' = -M^{-1} \bmod b \equiv (b - m_0)^{-1} \bmod b \equiv -m_0^{-1} \bmod b$, where usually $b = 2^k$ for some $k > 0$. Similarly to Algorithm 4.9, one can see that Z in Step 4 of Algorithm 4.10 is an integer by substituting $q = (z_0 + x_i \cdot y_0)\, m' + l \cdot b$, for some integer l, in the expression $(Z + x_i \cdot Y + q \cdot M)\,/b$. Finally, we notice that $0 \le Z < 2M - 1$. To justify this, assume that $0 \le Z < 2M - 1$ at iteration i of Algorithm 4.10. Then, Step 4 of Algorithm 4.10 replaces Z with $(Z + x_i \cdot Y + q \cdot M)/b \le ((2M - 2) + (b - 1)(M - 1) + (b - 1)M)/b = 2M - 1 - (1/b) < 2M - 1$. This justifies Steps 6 through 8 of Algorithm 4.10 and guarantees that the output of the Algorithm is less than M.

Algorithm 4.10 Montgomery Multiplication

Input: $X = \sum_{i=0}^{n-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$, $M = \sum_{i=0}^{n-1} m_i b^i$, with $0 \le X, Y < M$, $b > 1$, $m' = -m_0^{-1} \bmod b$, $R = b^n$, $\gcd(b, M) = 1$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$

1: $Z \leftarrow 0$ {where $Z = \sum_{i=0}^{n} z_i b^i$}
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad q \leftarrow (z_0 + x_i \cdot y_0)\, m' \bmod b$
4: $\quad Z \leftarrow (Z + x_i \cdot Y + q \cdot M) / b$
5: **end for**
6: **if** $Z \ge M$ **then**
7: $\quad Z \leftarrow Z - M$
8: **end if**
9: Return(Z)

In contrast to previous algorithms, Montgomery's algorithm reverses the processing order of the digits of multiplicand X, performs a shift down instead of up on each iteration, and does an addition rather than a subtraction. These changes are used in [EW93] to simplify the combinatorial logic needed to implement Montgomery reduction. The algorithm as modified in [EW93] is shown as Algorithm 4.11 .

Algorithm 4.11 Modified Montgomery Multiplication according to [EW93]

Input: $X = \sum_{i=0}^{n-1} x_i b^i$, $Y = \sum_{i=0}^{n-1} y_i b^i$, $M = \sum_{i=0}^{n-1} m_i b^i$, with $0 \le X, Y < M$, $b > 1$ $m' = -m_0^{-1} \bmod b$, $R = b^n$, $\gcd(b, M) = 1$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$

1: $Z \leftarrow 0$ {where $Z = \sum_{i=0}^{n} z_i b^i$}
2: $q \leftarrow 0$
3: **for** $i = 0$ to $n + 1$ **do**
4: $\quad Z \leftarrow \left(Z + x_i \cdot (b^2 Y) + q \cdot M\right) / b$
5: $\quad q \leftarrow z_0 \cdot m' \bmod b$
6: **end for**
7: **if** $Z \ge M$ **then**
8: $\quad Z \leftarrow Z - M$
9: **end if**
10: Return(Z)

The idea in [EW93], is to shift Y by two digits (i.e., multiply Y by b^2) thus making q in Step 3 of Algorithm 4.10 independent of Y. Notice that one could have multiplied Y by b instead of b^2 and have also obtained a q independent of Y. However, by multiplying Y by b^2, one gets q to be dependent only on the partial product Z and on the lowest two digits[3] of the multiple of M (i.e. $q \cdot M$). The price of such a modification is two extra iterations of the for-loop for which the digits of X are zero. The architecture proposed by [EW93] is only considered for the case $b = 2$ and estimated to be twice as fast as previous modular multiplication architectures at the time of publication.

4.3.1 Towards Higher Radix Montgomery Multipliers

Reference [SV93] describes the implementation of modular exponentiation architectures on FPGAs. They utilized an array of 16 XILINX 3090 FPGAs. Their design uses several speed-up methods [SV93] including the Chinese remainder theorem, asynchronous carry completion adder, and a windowing exponentiation method. This section describes in some detail their implementation as theirs appears to be the first *working hardware implementation* of the RSA algorithm using Montgomery's method for multiplication.

The algorithm used in [SV93] is basically Algorithm 4.10. Notice that since $0 \leq Z < 2M - 1$, one only needs $n + 1$ digits[4] to represent Z as noted in Step 1 of Algorithm 4.10. However, [SV93] avoids having to perform a subtraction after every modular product of the exponentiation algorithm by letting all intermediate results have *two extra bits* of precision. [SV93] also shows that even allowing for the two extra bits of precision, one can always manage to work with intermediate results no larger than n-digits if $M < b^n/4$ and $X, Y \leq 2M$. A second improvement over previous implementations is the use of a radix $b > 2$. In particular, they use $b = 2^2$ as a radix which permits for a trivial computation of the quotient q in Step 3 of Algorithm 4.10 and it allows for the use of Booth recoded multiplications (this doubles the multipliers performance compared to $b = 2$ at an approximate 1.5 increase in hardware complexity). Higher radixes, which would offer better performance, were dismissed since they involve too great of a hardware cost and the computation of the quotient digits is no longer trivial. As in other hardware implementations and to reduce the carry-propagation time, [SV93] use carry-save adders. Upon completion of each modular multiplication stage they convert back to the non-redundant form in order to feed this result back into the modular multiplier in the next exponentiation step. In some implementations, the operands are always kept in carry-save form but this doubles the size of radix-4 multiplier compared to a non-redundant one. [SV93] solution is to convert back to the non-redundant representation by using an asynchronous carry completion detection circuit and clock the final result for as many cycles as needed to fully propagate all carries. They notice that *on average*[5], the circuit will only need to propagate through $\log_2(k)$ bits (where k is the number of bits in the result). The technique provides a valuable saving in multiplier area for a small increase in the number of cycles needed per modular multiplication. Finally, we notice that they re-write Montgomery's Algorithm in a similar way to Algorithm 4.11, to allow for pipeline execution, basically getting rid off of the q dependency on the least significant digit of the partial product Z. The cost for d levels of pipelining is d extra bits of precision and d more cycles in the computation of the final product.

RSA Implementation

The authors in [SV93] implement RSA [RSA78] in what they call a PAM (Programmable Active Memory), basically a universally configurable hardware co-processor based on FPGA technology and closely coupled to a standard host computer. For decryption (signature generation) they make use of the Chinese Remainder Theorem (CRT), since the private-key is known (i.e. the factorization of the modulus $M = P \cdot Q$, where P, Q are primes). The CRT provides their implementation with a factor of 4 speedup. This factor of 4 is achieved by using the reconfigurability capabilities of the PAM. In particular, two hardware multipliers of size $k/2$ bits each (where M is k-bits long) are instantiated in the PAM together with one of the following solutions used to recombine the output of the two multipliers into one:

1. Compute the CRT recombination in software. A 40 MIPS machine performs this operation at 600 Kbits/sec on a pair of 512-bit primes.

2. Assist a slower host computer with a fast enough hardware multiplier, running in parallel with the two exponentiators.

3. Design the two $k/2$-bit modulo P and Q multipliers so that they can be reconfigured quickly enough into one single k-bit multiplier modulo M which can be used for the recombination step in the CRT.

For the encryption (signature verification) operation, a short exponent such as $2^{16}+1$ is assumed. Finally, for the exponentiation they use the m-ary method [MvOV97, Gor98] with windows of size five bits for a 512-bit modulus (i.e. exponentiations modulo 256-bit primes) and windows of size 6-bits for 1024-bit modulus. The average number of multiplications is then $1.24k$ and $1.22k$ for $k = 512$ bits and $k = 1024$ bits, respectively, compared to $1.5k$ operations in the case of the binary method for exponentiation.

The result of all these speedup methods, is an RSA secret decryption rate of over 600 Kbits/sec for a 512-bit modulus and of 165 Kbits/sec for a 1024-bit modulus. While the previous results make full use of the PAM reconfigurability, they derive a single gate-array specification whose size is estimated under 100K gates and speed over 1Mbit/sec for RSA 512-bit keys.

4.3.2 Avoiding Quotient Determination in High Radix Montgomery

The main obstacle to the use of higher radixes in the Montgomery algorithm is that of the quotient determination. In [Oru95], the author presents a method which avoids quotient determination all together and makes higher-radix Montgomery practical.

In what follows we assume the radix b to be of the form $b = 2^k$. Our starting point for the improvements is Algorithm 4.10. Step 3 is the determination of the quotient q and it looks like

$$q \leftarrow (z_0 + x_i \cdot y_0)m' \bmod 2^k \tag{4.11}$$

Step 4 is the computation of the partial product Z as

$$Z \leftarrow (Z + x_i \cdot Y + q \cdot M)/2^k \tag{4.12}$$

[Oru95] first notices that if $k = 1$, then $m' = 1$ and the multiplication by m' in (4.11) can be avoided. Thus, the idea is to make m' always equal to 1 independent of the modulus. To this end, [Oru95] defines a new modulus $\widetilde{M} = (M' \bmod 2^k)M$ which satisfies $\widetilde{M} \equiv -1 \bmod 2^k$ and, thus, the desired property. Equations (4.11) and (4.12) become

$$q \leftarrow (z_0 + x_i \cdot y_0) \bmod 2^k \tag{4.13}$$

$$Z \leftarrow \left(Z + x_i \cdot Y + q \cdot \widetilde{M}\right)/2^k \tag{4.14}$$

Thus, we have replaced multiplication by m' in every step of the algorithm by a single multiplication (computing $\widetilde{M}$) done once at the beginning of the algorithm. The penalty of these modifications is that Z requires more precision and the loop has increased its size by at most one. Similarly to [EW93], the quotient computation in (4.13) can be further reduced by replacing Y by $2^k Y$. Then, (4.13) and (4.14) are reduced to

$$q \leftarrow z_0 \bmod 2^k \tag{4.15}$$

$$Z \leftarrow \left(Z + q \cdot \widetilde{M}\right)/2^k + x_i \cdot Y \tag{4.16}$$

The price to pay is one more iteration in the for-loop to compensate for the extra 2^k factor. The resulting algorithm is shown as Algorithm 4.12 . Reference [Oru95] notices[6] that $\widetilde{M}+1$ is divisible by $b = 2^k$.

Algorithm 4.12 Montgomery Multiplication Avoiding Quotient Determination [Oru95]

Input: $b = 2^k$, $M > 2$ with $\gcd(2, M) = 1$, $\widetilde{M} = (M' \bmod b)M$, with $4\widetilde{M} < b^n$ and $R = b^n$ and M satisfying $R \cdot R^{-1} - M \cdot M' = 1$, $X = \sum_{i=0}^{n} x_i b^i$, $x_n = 0$, and $0 \leq X < 2\widetilde{M}$, $0 \leq Y < 2\widetilde{M}$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$ and $0 \leq Z < 2\widetilde{M}$

1: $Z \leftarrow 0$
2: **for** $i = 0$ to n **do**
3: $\quad q \leftarrow Z \bmod b$
4: $\quad Z \leftarrow \left(Z + q \cdot \widetilde{M}\right)/b + x_i \cdot Y$
5: **end for**
6: Return(Z)

Thus, Step 4 of Algorithm 4.12 can be simplified as follows:

$$\begin{aligned} \frac{(Z + q \cdot \widetilde{M})}{2^k} + x_i \cdot Y &= \frac{Z - (Z \bmod 2^k)}{2^k} + \frac{q \cdot \widetilde{M} + (Z \bmod 2^k)}{2^k} + x_i \cdot Y \\ &= \frac{Z}{2^k} + \frac{q \cdot (\widetilde{M}+1)}{2^k} + x_i \cdot Y \end{aligned} \tag{4.17}$$

where we have made use of the fact that $q = Z \bmod 2^k$ and that $(Z - (Z \bmod 2^k))/2^k = Z/2^k$ if we see division by 2^k as right shifting Z to the right k positions. Notice that the $\widetilde{M}+1$ is a constant which can be pre-computed at the beginning of an exponentiation with a new value M. With the simplification of (4.17), we do not need to calculate the carry-out from the k least significant bits of $Z + q \cdot \widetilde{M}$.

The final improvement in [Oru95] is the use of quotient pipelining, similar to [SV93], the idea is to delay the use of quotient digit q_{i-d} determined from information available in iteration $i-d$, by d iterations. The result is that d iterations are available for determining a quotient. In [SV93] the price to pay is d extra iterations and an increased in the quotient range, i.e., $q_{i-d} \in \{0, 1, \cdots, 2^{k(d+1)-1}\}$. Orup improves over this by not increasing the quotient range but only increasing the number of iterations. This has the advantage of not requiring more complex hardware to compute the products $q_{i-d} \cdot (\widetilde{M}+1)$. The result is presented as Algorithm 4.13.

Algorithm 4.13 Improved Montgomery Multiplication with Quotient Pipelining [Oru95]

Input: $b = 2^k$, $M > 2$ with $\gcd(2, M) = 1$, $\widetilde{M} = (M' \bmod b^{d+1})M$, with $4\widetilde{M} < b^n$, R^{-1} such that $b^n \cdot R^{-1} \bmod M = 1$ and M' satisfying $-M \cdot M' \bmod b^{d+1} = 1$, $X = \sum_{i=0}^{n+d} x_i b^i$, $x_i = 0$ for $i \geq n$, and $0 \leq X < 2\widetilde{M}$, $0 \leq Y < 2\widetilde{M}$

Output: $Z = X \cdot Y \cdot R^{-1} \bmod M$ and $0 \leq Z < 2\widetilde{M}$

1: $Z \leftarrow 0$
2: $q_{-d} \leftarrow 0,\ q_{-d+1} \leftarrow 0, \cdots, q_{-1} \leftarrow 0$
3: **for** $i = 0$ to $n + d$ **do**
4: $\quad q_i \leftarrow Z \bmod b$
5: $\quad Z \leftarrow Z/b + q_{i-d} \cdot (\widetilde{M}+1)/2^{k(d+1)} + x_i \cdot Y$
6: **end for**
7: Return(Z)

An example architecture is also considered in [Oru95] with 3 pipeline stages and a radix $b = 2^8$. The author estimates the critical path of the architecture to be no more than 5 nsec assuming 1995 CMOS technology. It is also assumed the use of a redundant representation for the intermediate values of the Montgomery multiplier. However, the outputs have to be converted back to non-redundant representation using a carry-ripple adder with an asynchronous carry completions detection circuit as proposed in [SV93]. With these techniques, the author estimates the time of one 512-bit modular multiplication at 415 nsec. Using the binary method for exponentiation, one 512-bit exponentiation would take 319 μsec which corresponds to a 1.6 Mbit/sec throughput. Modifying the binary algorithm so as to read the bits of the exponent from the least significant bit to the most significant bit[7], one can perform multiplications and squarings in parallel as shown in [OK91], thus achieving a factor of two speedup, i.e., more than 2.4 Mbit/sec throughput. This is four times faster than the implementation of [SV93] which at the time was the fastest. Furthermore, if the modulus is composite, as in the RSA case, and its prime factorization is known, it is possible to obtain a factor of four speedup through the use of the CRT as in [SV93]. The author estimates that the architecture would require no more than 300000 transistors for 512 operands.

4.3.3 High-Radix Montgomery Exponentiation on FPGAs

This section presents the work introduced in [BP99, BP01b] which extends [Oru95] to reconfigurable hardware and a systolic array architecture as presented in [Kor94]. There have been a number of proposals for systolic array architectures for modular arithmetic. However, to our knowledge [BP99, BP01b] were the first implementations that have been reported.

In [BP99], the authors presented a version of Montgomery's algorithm optimized for a radix two hardware implementation. The version shown below is based on Algorithm 4.12, which is suitable for high radix hardware implementations of modular exponentiation. The goal of the new architecture was to design a speed efficient architecture using a systolic array which realizes Algorithm 4.12. As target devices [BP99, BP01b] use the XILINX XC4000 family [Xil96]. An XC4000 CLB consists of three look–up tables, two flip–flops and programmable multiplexers. Two Boolean functions of four inputs can be computed in one CLB. Note that Altera, Lucent, and Actel have FPGA families with related architectures and it is expected that the design from [BP01b] will be suitable for those FPGAs as well. For a more detailed description and timing diagrams of the design described in this section, we refer to [Blu99].

Design Overview

One of the major problems when implementing Algorithm 4.12 is computing multiples of Y and $\widetilde{M}$ in Step 4. Reference [Oru95] proposes a multiplexer network. This approach is not suitable for a systolic array implementation in FPGAs because of the following reasons:

1. For a radix of 2^2 the multiplexer could be implemented in one CLB per bit length. However, for $b = 2^4$, the design would require more than four CLBs per bit. This would result in unrealistically large CLB counts for secure bit lengths in cryptographic applications.

2. In a systolic array, one typically computes k bits per processing element. With a multiplexer solution the internal bit length becomes $2k$ resulting in twice the cost for adders and registers.

To avoid the doubling of the internal bit length of a unit the following approach which is optimized for the CLB architectures at hand can be taken:

- Pre-compute the multiples of Y and $\widetilde{M}$ at the beginning of Montgomery's algorithm execution and store the results for later use.
- Let the carries of this pre-computations propagate to the units to the left.

If a unit processes k bits, the stored multiples will also have k bits and the internal bit length will not exceed $k + 2$ bits (addition of 3 operands). The cost is 2^k additional clock cycles for calculating the 2^k multiples of Y. For small k values, this expense is negligible compared to the total amount of $2(n + 1)$ cycles for the whole algorithm. As storage elements, one can either use registers or RAM elements. For k larger than 2, registers are not suitable as they utilize one CLB per 2 stored bits. RAM elements are very efficient up to an address width of 4 bits [Xil96]. Their implementation requires only one CLB per two bits data width (2 CLBs for a 16×4 bit RAM). The resource requirements grow rapidly, though, for larger address width. A 64×6 bit implementation ($k = 6$) utilizes 18 CLBs, a 256×8 bit implementation ($k = 8$) utilizes 96 CLBs. Both would result in unrealistically large CLB counts for secure bit lengths. Additionally the $2^6 - 1$ or even $2^8 - 1$ clock cycles for pre-computing the multiples of Y are not negligible any more. Thus, to achieve an optimal time–area product, [BP01b] implemented an architecture with a radix $b = 2^4$ and computed 4 bits per processing element. Similar to the approaches in [Kor94], the architectures presented in [BP01b] use the binary method for exponentiation and compute squares and multiplications in parallel.

4.3.4 Performance

As target devices, [BP99, BP01b] used the XILINX XC40250XV, speed grade -09, 8464 CLBs, for the larger designs (> 5000 CLBs), and the XC40150XV, speed grade -08, 5184 CLBs, for the smaller designs. The designs were developed in VHDL and synthesized with Synopsis Design Compiler (version 1998.08). The place and route process of the synthesized designs was accomplished with the XILINX Design Manager tools (version M1.5.9). The timing results were computed by the XILINX timing analyzer from the placed and routed designs, and verified by the Synopsis VHDL debugger. They were not verified with an actual chip. Note that the XILINX tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [Alf99].

Table 4.1 shows the results from [BP99, BP01b] for a full length modular exponentiation, i.e., an exponentiation where base, exponent, and modulus have all the same bit length. We notice that [BP99, BP01b] both use the right-to-left method for exponentiation. Table 4.2 shows RSA encryption performance and resource requirements according to [BP99, BP01b]. The encryption time is calculated for the Fermat prime $F_4 = 2^{16} + 1$ exponent [Knu81], requiring $2 \cdot 19(n+2)$ clock cycles for the radix 2 design [BP99], and $2 \cdot 19(n+8)$ clock cycles if the radix 16 design is used, where the modulus has $n - 2$ bits.

Table 4.1. CLB usage and execution time for a full modular exponentiation

Radix	512 bit		768 bit		1024 bit	
	C (CLBs)	T (msec)	C (CLBs)	T (msec)	C (CLBs)	T (msec)
2 [BP99]	2555	9.38	3745	22.71	4865	40.05
16 [BP01b]	3413	2.93	5071	6.25	6633	11.95

Table 4.2. Application to RSA: Encryption

Radix	512 bit		1024 bit	
	C (CLBs)	T (msec)	C (CLBs)	T (msec)
2 [BP99]	2555	0.35	4865	0.75
16 [BP01b]	3413	0.11	6633	0.22

For decryption, [BP01b] apply the Chinese remainder theorem. They either decrypt m bits with an $m/2$ bit architecture serially, or with two $m/2$ bit architectures in parallel. The first approach uses only half as many resources, the latter is almost twice as fast. A little time is lost here because of the slower delay specifications of the larger devices.

Table 4.3. Application to RSA: Decryption

Radix	512 bit 2×256 serial		512 bit 2×256 parallel		1024 bit 2×512 serial		1024 bit 2×512 parallel	
	C (CLBs)	T (msec)	C (CLBs)	T (msec)	C (CLBs)	T (msec)	C (CLBs)	T (msec)
2 [BP99]	1307	4.69	2614	2.37	2555	18.78	5110	10.18
16 [BP01b]	1818	1.62	3636	0.79	3413	5.87	6826	3.10

4.4 Notes and Further References

1. b is in general a power of two and, thus, multiplication by b can be implemented through left shifts.

2. Notice that Quisquater's algorithm is intended for cases in which many modular reductions have to be performed such as in the case of an RSA exponentiation, thus the de-normalization is only performed at the end of the exponentiation.

3. We emphasize that q depends on the lowest two digits of M and not just on the lowest digit of M as at first sight might appear. The value of $z_0^{(i+1)}$ at iteration $i+1$ equals $((z_1^{(i)}b+z_0^{(i)})+q^{(i)}\cdot(m_1b+m_0))/b = z_1^{(i)} + m_1$.

4. More precisely, one needs n digits plus 1 bit to represent Z in Algorithm 4.10.

5. [Knu81] predicts this average and the authors measurements support the theoretical result.

6. The same observation is made in [Kor94] for a systolic Montgomery architecture in radix 2.

7. This is known as the right-to-left binary exponentiation algorithm [MvOV97]. The binary method is also known as the left-to-right binary exponentiation algorithm since it processes the bits of the exponent from left to right.

CHAPTER 5

Semi-Systolic Architectures for Arithmetic in $GF(p^m)$

In recent years, there has been increased interest in cryptographic systems based on fields of odd characteristic [PS95, Mih97, BP98, BP01a, Kob98, Sma99, LV00]. Nevertheless, there is a lack of hardware architectures for general odd characteristic fields, and thus, in this chapter we try to close this gap. Our approach is different from previous ones, in that, we propose *general* architectures which are suitable for fields $GF(p^m)$. In particular, we generalize the work in [SP98] to fields $GF(p^m)$ for odd primes p. We, then, study carefully the case of $GF(3^m)$ due to its cryptographic significance. In addition, we focused on finding irreducible polynomials over $GF(3)$ to improve the performance of the multiplier. For the problem of efficient $GF(p)$ arithmetic, we refer the reader to Chapters 3 and 4. We begin this chapter by surveying previous architectures for $GF(p^m)$ fields. Parts of this chapter appear in [BGK+03].

Notation

In the following, we will consider the field $GF(p^m)$ generated by an irreducible polynomial $q(x) = x^m + Q(x) = x^m + \sum_{i=0}^{m-1} q_i x^i$ over $GF(p)$ of degree m. We assume α to be a root of $q(x)$, thus for $A, B, C \in GF(p^m)$, we write $A = \sum_{i=0}^{m-1} a_i \alpha^i$, $B = \sum_{i=0}^{m-1} b_i \alpha^i$, $C = \sum_{i=0}^{m-1} c_i \alpha^i$, and $a_i, b_i, c_i \in$

$GF(p)$. Notice that by assumption $q(\alpha) = 0$ since α is a root of $q(x)$. Therefore,

$$\alpha^m = -Q(\alpha) = \sum_{i=0}^{m-1}(-q_i)\alpha^i \tag{5.1}$$

gives an easy way to perform modulo reduction whenever we encounter powers of α greater than $m-1$. Addition in $GF(p^m)$ can be achieved as shown in (5.2)

$$C(\alpha) \equiv A(\alpha) + B(\alpha) = \sum_{i=0}^{m-1}(a_i + b_i)\alpha^i \tag{5.2}$$

where the addition $a_i + b_i$ is done in $GF(p)$. Multiplication of two elements $A, B \in GF(p^m)$ is written as $C(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i \equiv A(\alpha) \cdot B(\alpha)$, where the multiplication is understood to happen in the finite field $GF(p^m)$ and all α^t, with $t \geq m$ can be reduced using (5.1). Notice that we abuse our notation and throughout the text we will write $A \bmod q(\alpha)$ to mean *explicitly* the reduction step described previously. Finally, we refer to A as the multiplicand and to B as the multiplier.

5.1 Previous $GF(p^m)$ Multipliers

In contrast to the $GF(p)$ case, there has not been a lot of work done on $GF(p^m)$ architectures. Our literature search yielded [PB95] and more recently [BINP03, GS03a] as the only references that explicitly treated the general case of $GF(p^m)$ multipliers, p odd[1]. Reference [OKPK99] treats explicitly the case of $GF((3^n)^3)$. We do not discuss [OKPK99] and [WHB02], who introduced parallel multipliers for $GF(p^m)$, as parallel multipliers are not well suited to cryptographic applications due to their excessive hardware requirements.

5.1.1 Non-general Multipliers

In [PB95], $GF(p^m)$ multiplication is computed in two stages:

1. The polynomial product is computed modulo a highly factorisable degree S polynomial, $M(x)$, with $S \geq 2m-1$. This restriction comes from the fact that the product of two polynomials of

maximum degree $m-1$ is at most $2m-1$. Then, the product is computed using a polynomial residue number system (PRNS), originally introduced in [ST91]. This involves converting back and forth between the normal representation and the PRNS representation.

2. The second step involves reducing modulo the irreducible polynomial $q(x)$ over which $GF(p^m)$ is defined.

In order to further simplify the complexity of these multipliers, [PB95] suggests to limit the form of $M(x)$ to being fully factorisable into degree-one polynomials. [PB95] shows that this is equivalent to requiring that the inequality $2m < p$ be satisfied. This restriction implies that for all primes $p < 67$, these multipliers can not be implemented if intended for use in cryptographic applications[2]. A second optimization in [PB95] is to consider only fields $GF(p^m)$ for which an irreducible binomial of degree m over $GF(p)$ exists. As it can be seen from Table A.1, the second optimization reduces significantly the number of fields for which these multipliers are of interest. We notice that the number of fields for which these multipliers are feasible might be increased by considering higher-dimensional PRNS as suggested in [PB95]. However, this technique requires that m be a composite integer which in many cryptographic applications is seen with skepticism because of security considerations. Thus, we do not consider this method any further in this work. The architectures presented in [BINP03] are similarly constrained, i.e., they can only be implemented for $p \geq 67$ if the desired group size is 2^{160}, however, there are no restrictions similar to [PB95].

5.1.2 General Multipliers

To our knowledge, [PS02] is the first to describe $GF(3^m)$ architectures for applications of cryptographic significance. The authors introduce a representation similar to the one used by [GHS02a] to represent their polynomials. In particular, they combine all the least significant bits of the coefficients of an element, say A, into one value and all the most significant bits of the coefficients of A into a second value (notice the coefficients of A are elements of $GF(3)$ and thus 2 bits are needed to represent each of them). Thus, $A = (a_1, a_0)$ where a_1 and a_0 are m-bit long each. Addition of two polynomials

$A = (a_1, a_0)$, $B = (b_1, b_0)$ with $C = (c_1, c_0) \equiv A + B$ is achieved as:

$$\begin{aligned} t &= (a1 \vee b0) \oplus (a0 \vee b1) \\ c1 &= (a0 \vee b0) \oplus t \\ c2 &= (a1 \vee b1) \oplus t \end{aligned} \tag{5.3}$$

where $\vee$ and $\oplus$ mean the logical OR and exclusive OR operations, respectively. Page and Smart [PS02] notice that subtraction and multiplication by 2 are equivalent in characteristic 3 and that they can be achieved as $2 \cdot A = 2 \cdot (a1, a0) = -A = -(a1, a0) = (a0, a1)$. Multiplication is achieved in the bit-serial manner, by repeatedly shifting the multiplier down by one bit position and shifting the multiplicand up by one bit position. The multiplicand is then added or subtracted depending on whether the least significant bit of the first or second word of the multiplier is equal to one. The authors do not mention what methods were used to perform modular reduction in the field. Reference [PS02] also notices that with this representation a cubing operation can only be as fast as a general multiply, whereas, using other implementation methods the cubing operation could be much faster. The implementation of multiplication in $GF((3^m)^6)$ is also discussed using the irreducible polynomial $Q(y) = y^6 + y + 2$. They use the normal method to multiply polynomials of degree 5 with coefficients in $GF(3^m)$ and then reduce modulo $Q(y)$ using 10 additions and 4 doublings in $GF(3^m)$. In addition, they suggest that using the Karatsuba algorithm for multiplication [KO63], performance can be improved at the cost of additional area. They provide timings which we further discuss in Section 5.6.2.

In [SP98] a new approach for the design of digit-serial/parallel $GF(2^k)$ multipliers is introduced. Their approach combines both array-type and parallel multiplication algorithms, where the digit-type algorithms minimize the latency for one multiplication at the expense of extra hardware inside each digit cell. In addition, the authors consider special types of polynomials which allow for efficiency in the modulo $q(x)$ reduction operation. These architectures are generalized in Section 5.3 to the $GF(p^m)$ case, where p is odd.

5.2 Adder Architectures for $GF(p^m)$

This section is concerned with hardware architectures for addition and multiplication in $GF(p^m)$. Inversion can be performed through the Euclidean algorithm or by exponentiation based techniques (see for example [GP02]). We notice that in many cases inversion is avoided through the use of projective coordinates in elliptic curve based systems and through the use of explicit formulas in the hyperelliptic curve case. Thus, we do not treat inversion any further in this paper.

Addition in $GF(p^m)$ is performed according to (5.2). A parallel adder requires m $GF(p)$ adders and its critical path delay is one $GF(p)$ adder. In some multiplier architectures, such as the Most Significant Digit-Element (MSDE) first multiplier described in Section 5.4.2, the addition of two intermediate polynomials of degree larger than m might need to be performed. In these cases, a parallel adder will require $(m + D)$ $GF(p)$ adders but the critical path delay will remain that of one $GF(p)$ adder.

5.3 Serial Multipliers for $GF(p^m)$

There are three different types of architectures used to build $GF(p^m)$ multipliers: array-, digit-, and parallel-multipliers [SP98]. Array-type (or serial) multipliers process all coefficients of the multiplicand in parallel in the first step, while the coefficients of the multiplier are processed serially. Array-type multiplication can be performed in two different ways, depending on the order in which the coefficients of the multiplier are processed: Least Significant Element (LSE) first multiplier and Most Significant Element (MSE) first multiplier, described in this section. We also discuss digit-multipliers which are also divided in Most Significant and Least Significant Digit-Element first multipliers, depending on the order in which the coefficients of the polynomial are processed. Parallel-multipliers have a high critical path delay but only require one clock cycle to complete a whole multiplication. Thus, parallel-multipliers exhibit high throughput and they are best suited for applications requiring high-speed and relatively small finite fields. However, they are expensive in terms of area when compared to serial multipliers and, thus, often prohibitive for cryptographic applications. We do not discuss parallel-multipliers any further in this thesis.

5.3.1 Least Significant Element (LSE) First Multiplier

The LSE scheme processes first coefficient b_0 of the multiplier and continues with the remaining coefficients one at the time in ascending order. Hence, multiplication according to this scheme can be performed in the following way:

$$\begin{aligned} C &\equiv AB \bmod q(\alpha) \\ &\equiv b_0A + b_1(A\alpha \bmod q(\alpha)) + b_2(A\alpha^2 \bmod q(\alpha)) + \ldots + b_{m-1}(A\alpha^{m-1} \bmod q(\alpha)) \end{aligned}$$

The accumulation of the partial product has to be performed with a polynomial adder. This multiplier computes the operation according to Algorithm 5.1. The counterpart of the LSE multiplier is the Most

Algorithm 5.1 LSE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$, where $a_i, b_i \in GF(p)$
Output: $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in GF(p)$
1: $C \leftarrow 0$
2: **for** $i = 0$ to $m-1$ **do**
3: $C \leftarrow b_iA + C$
4: $A \leftarrow A\alpha \bmod q(\alpha)$
5: **end for**
6: Return (C)

Significant Element (MSE) first multiplier which is considered in the next section.

5.3.2 Most Significant Element First Multiplier (MSE)

The most significant element multiplication starts with the highest coefficient of the multiplier polynomial. Hence, the multiplication can be performed in the following way:

$$\begin{aligned} C &\equiv AB \bmod q(\alpha) \\ &\equiv (\ldots(b_{m-1}A\alpha \bmod q(\alpha) + b_{m-2}A)\alpha \bmod q(\alpha) + \ldots + b_1A)\alpha \bmod q(\alpha) + b_0A \end{aligned}$$

Algorithm 5.2 describes the operation of the MSE multiplier. Notice that in Step 3 of Algorithm 5.2 the computation of $b_{m-1-i}A$ and $C\alpha \bmod q(\alpha)$ can be performed in parallel as they are independent of each other. However, the value of C in each iteration depends on both the value of C at the previous

Algorithm 5.2 MSE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i, B = \sum_{i=0}^{m-1} b_i\alpha^i$, where $a_i, b_i \in GF(p)$
Output: $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in GF(p)$
1: $C \leftarrow 0$
2: **for** $i = 0$ to $m-1$ **do**
3: $\quad C \leftarrow C\alpha \bmod q(\alpha) + b_{m-1-i}A$
4: **end for**
5: Return(C)

iteration and on the value of $b_{m-1-i}A$. This dependency has the effect of making the MSE multiplier have a longer critical path than that of the LSE multiplier.

5.3.3 Modular Reduction

In both LSE and MSE multipliers a quantity $W\alpha$, where $W = \sum_{i=0}^{m-1} w_i\alpha^i \in GF(p^m)$, $w_i \in GF(p)$, has to be reduced $\bmod q(\alpha)$. Multiplying W by α, we obtain

$$W\alpha = \sum_{i=0}^{m-1} w_i\alpha^{i+1} = w_{m-1}\alpha^m + \sum_{i=0}^{m-2} w_i\alpha^{i+1}$$

Using (5.1) and re-writing the index of the second summation, $W\alpha \bmod q(\alpha)$ can then be calculated as follows:

$$W\alpha \bmod q(\alpha) = \sum_{i=0}^{m-1}(-q_i w_{m-1})\alpha^i + \sum_{i=1}^{m-1} w_i\alpha^i = (-q_0 w_{m-1}) + \sum_{i=1}^{m-1}(w_{i-1} - q_i w_{m-1})\alpha^i \tag{5.4}$$

where all coefficient arithmetic is done modulo p. From (5.4) and the definitions of A, B, and C, we can write expressions for A and C in Algorithm 5.1 at iteration i as follows:

$$\begin{aligned} C^{(i)} &= \sum_{j=0}^{m-1} c_j^{(i)}\alpha^j = b_i A^{(i)} + C^{(i-1)} = \sum_{j=0}^{m-1}\left(b_i a_j^{(i)} + c_j^{(i-1)}\right)\alpha^j, \quad C^{(-1)} = 0 \\ A^{(i)} &= \sum_{j=0}^{m-1} a_j^{(i)}\alpha^j \equiv A^{(i-1)}\alpha \bmod q(\alpha) = \left(-q_0 a_{m-1}^{(i-1)}\right) + \sum_{j=1}^{m-1}\left(a_{j-1}^{(i-1)} - q_j a_{m-1}^{(i-1)}\right)\alpha^j, \quad A^{(-1)} = A \end{aligned}$$

Similarly, we can write an expression for C at iteration i of Algorithm 5.2 as

$$
\begin{aligned}
C^{(i)} &= \sum_{j=0}^{m-1} c_j^{(i)} \alpha^j \equiv C^{(i-1)} \alpha \bmod q(\alpha) + b_{m-1-i} A \\
&= \left(a_0 b_{m-1-i} - q_0 c_{m-1}^{(i-1)}\right) + \sum_{j=1}^{m-1} \left(c_{j-1}^{(i-1)} - q_j c_{m-1}^{(i-1)} + a_j b_{m-1-i}\right) \alpha^j, \quad C^{(-1)} = 0
\end{aligned}
$$

where in both cases the loop index i runs from 0 to $m-1$. As a final remark, notice that if one initializes C to a value different from 0, say I, then Algorithms 5.1 and 5.2 compute $C \equiv A \cdot B + I \bmod q(\alpha)$. This multiply-accumulate operation turns out to be very useful in elliptic curve systems and it is obtained at no extra cost.

5.3.4 Area/Time Complexity of LSE and MSE Multipliers

The LSE multiplier takes m iterations to output the product $C \equiv A \cdot B \bmod q(\alpha)$. In each iteration, the following operations are performed: 1 multiplication of a $GF(p)$ element by a $GF(p^m)$ element (requires m $GF(p)$ multipliers), 1 $GF(p^m)$ addition (requires m $GF(p)$ adders), 1 multiplication by α (implemented as a $GF(p)$ coefficient shift), and 1 modulo $q(\alpha)$ reduction. This last operation could be implemented according to (5.4), and thus, it would require $(r-1)$ $GF(p)$ multipliers and $(r-2)$ adders for a fixed r-nomial (where r is the number of non-zero coefficients in the irreducible polynomial $q(x)$). However, one could envision a chip in which the modulus is loaded on demand, thus, requiring m $GF(p)$ multipliers and $(m-1)$ adders. A similar analysis for Algorithm 5.2 yields the following. The MSE multiplier takes m iterations to output the product $C \equiv A \cdot B \bmod q(\alpha)$. In each iteration, one has to perform 1 multiplication of a $GF(p)$ element by a $GF(p^m)$ element, 1 multiplication by α, 1 modulo $q(\alpha)$ reduction, and the addition of two $GF(p^m)$ elements. Notice that this last addition makes the critical path of the MSE multiplier one $GF(p)$ adder longer. The reduction in the MSE case can be implemented in the same manner as for the LSE multiplier.

Both the area and time complexities of Algorithms 5.1 and 5.2 are summarized in Table 5.1 and Table 5.2, respectively, in terms of $GF(p)$ adders and multipliers, for two types of irreducible polynomials. We chose this unusual measure because it is independent of technology and thus most general.

Table 5.1. Area complexity and critical path delay of LSE multiplier.

Irreducible polynomial	Area Complexity	Critical Path Delay	Latency (# clocks)
r-nomial	$(m+r-2)$ ADD + $(m+r-1)$ MUL	1 ADD + 1 MUL	m
General	$(2m-1)$ ADD + $2m$ MUL	1 ADD + 1 MUL	m

One immediate advantage of estimating area in terms of $GF(p)$ adders (multipliers) is that we do not need to care about the way these are implemented. In particular, there are many implementation choices depending on the application, design criteria, and size of the modulus as shown in Chapter 3. Section 5.5 gives specific complexity numbers for an FPGA implementation of $GF(3^m)$ arithmetic in terms of both, Look-Up Tables[3] (LUTs) and Configurable Logic Blocks (CLBs), thus taking into account the way $GF(p)$ arithmetic is implemented and the target technology.

Table 5.2. Area complexity and critical path delay of MSE multiplier.

Irreducible polynomial	Area Complexity	Critical Path Delay	Latency (# clocks)
r-nomial	$(m+r-2)$ ADD + $(m+r-1)$ MUL	2 ADD + 1 MUL	m
General	$(2m-1)$ ADD + $2m$ MUL	2 ADD + 1 MUL	m

In Tables 5.1 and 5.2, ADD and MUL refer to the area and delay of a $GF(p)$ adder and multiplier, respectively. We have not taken into account the delays or area requirements of storage elements (such as those needed to implement a shift register) or routing elements (such as those used for interconnections in FPGAs). In addition, we do not make any distinction between general and constant $GF(p)$ multipliers, i.e., we assume their complexities are the same. General irreducible polynomials refer to the case in which one wants to be able to change the irreducible polynomial on demand. Finally, notice that the complexity for LSE and MSE multipliers when using an r-nomial with $r = m+1$ non-zero coefficients reduces to the complexity of the multipliers when using general polynomials, as expected.

5.4 Digit-Serial/Parallel Multipliers for $GF(p^m)$

Both LSE and MSE multipliers process the coefficients of A in parallel while the coefficients of B are processed in a serial manner. Hence, these multipliers are area-efficient and suitable for low-speed applications. Digit multipliers, introduced in [SP98] for fields $GF(2^k)$, are a trade-off between speed, area, and power consumption. This is achieved by processing several of B's coefficients at the same time. The number of coefficients that are processed in parallel is defined to be the digit-size and we denote it with the letter D.

For a digit-size D, we can denote by $d = \lceil m/D \rceil$ the total number of digits in a polynomial of degree $m-1$. Then, we can re-write the multiplier as $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$, where

$$B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j \;\; 0 \le i \le d-1 \tag{5.5}$$

and we assume that B has been padded with zero coefficients such that $b_i = 0$ for $m-1 < i < d \cdot D$ (i.e. the size of B is $d \cdot D$ coefficients but $\deg(B) < m$). Hence,

$$C \equiv AB \bmod q(\alpha) = A \sum_{i=0}^{d-1} B_i \alpha^{Di} \bmod q(\alpha) \tag{5.6}$$

In the following, two generalized digit-serial/parallel multiplication algorithms are introduced. These algorithms are classified as Least Significant Digit-Element first multiplier (LSDE) and Most Significant Digit-Element first multiplier (MSDE). Here, we have introduced the word *element* to clarify that the digits correspond to groups of $GF(p)$ coefficients in contrast to [SP98] where the digits were groups of bits.

5.4.1 Least Significant Digit-Element (LSDE) Multipliers

The LSDE is an extension of the LSE multiplier. Using (5.6), the product in this scheme can be calculated as follows

$$\begin{aligned} C &\equiv AB \bmod q(\alpha) \\ &\equiv [B_0 A + B_1(A\alpha^D \bmod q(\alpha)) + B_2(A\alpha^D\alpha^D \bmod q(\alpha)) \\ &\quad + \ldots + B_{d-1}(A\alpha^{D(d-2)}\alpha^D \bmod q(\alpha))] \bmod q(\alpha) \end{aligned}$$

This is summarized in Algorithm 5.3. A similar derivation, is performed for MSDE multipliers in

Algorithm 5.3 LSDE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, where $a_i \in GF(p)$, $B = \sum_{i=0}^{\lceil \frac{m}{D} \rceil - 1} B_i\alpha^{Di}$, where B_i is as defined in (5.5)
Output: : $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in GF(p)$

$C \leftarrow 0$
for $i = 0$ to $\lceil \frac{m}{D} \rceil - 1$ **do**
  $C \leftarrow B_iA + C$
  $A \leftarrow A\alpha^D \bmod q(\alpha)$
end for
Return ($C \bmod q(\alpha)$)

the next section. We end this section by noticing that, as in Algorithm 5.1, if C is initialized to I in Algorithm 5.3, then we can obtain as an output the quantity $A \cdot B + I \bmod q(\alpha)$ at no additional (hardware or delay) cost. This operation, known as a multiply/accumulate operation, is very useful in elliptic curve cryptosystems.

5.4.2 Most Significant Digit-Element First Multiplier (MSDE)

In analogy to LSDE multipliers, one can obtain an MSDE by re-writing (5.6) as

$$\begin{aligned} C &\equiv AB \bmod q(\alpha) \\ &\equiv ((((\ldots(((AB_{d-1} \bmod q(\alpha))\alpha^D + AB_{d-2}) \bmod q(\alpha))\alpha^D + \cdots)\alpha^D) \\ &\quad +AB_1) \bmod q(\alpha))\alpha^D + AB_0) \bmod q(\alpha) \end{aligned}$$

where, we start processing the digit-elements of B from the most significant element to the least significant. Algorithm 5.4 summarizes this result. Notice that in both LSDE and MSDE multipliers, the

Algorithm 5.4 MSDE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, where $a_i \in GF(p)$, $B = \sum_{i=0}^{\lceil \frac{m}{D} \rceil - 1} B_i\alpha^{Di}$, where B_i is as defined in (5.5)
Output: $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in GF(p)$

$C \leftarrow 0$
for $i = 0$ to $\lceil \frac{m}{D} \rceil - 1$ **do**
  $C \leftarrow AB_{d-1-i} + (C \bmod q(\alpha))\alpha^D$
end for
Return ($C \bmod q(\alpha)$)

intermediate result C is of degree larger than $m = \deg(q(\alpha))$. This fact has two consequences: (i) both LSDE and MSDE multipliers require a number of storage elements which is larger than the degree of the irreducible polynomial $q(\alpha)$ and (ii) after d loop iterations, one must perform one additional reduction.

5.4.3 Modular Reduction for LSDE and MSDE Multipliers

In both LSDE and MSDE a product of the form $W\alpha^D \bmod q(\alpha)$ occurs. As in the LSE multiplier case, one can derive equations for the modular reduction for *particular* irreducible $q(\alpha)$ polynomials. However, it is more interesting to search for *families* of polynomials that minimize the complexity of the reduction operation. In coming up with these optimum irreducible polynomials we use two theorems from [SP98], adapted to the case of $GF(p^m)$ fields with p odd.

Theorem 5.1. *[SP98, Theorem 1] Assume that $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{j=0}^{k-1} q_j\alpha^j$, with $k < m$. For $t \leq m-1-k$, the degree of α^{m+t} can be reduced to be less than m in one step with the following equation:*

$$\alpha^{m+t} \bmod q(\alpha) = -\sum_{j=0}^{k} q_j\alpha^{j+t} \tag{5.7}$$

Proof. The result follows from (5.1), the assumed form of $q(\alpha)$, and the fact that for α^{k+t} where $t + k \leq m - 1$ no modular reduction is necessary. □

Theorem 5.2. *[SP98, Theorem 2] For digit multipliers with digit-element size D, when $D \leq m - k$ the degree of the intermediate results in Algorithms 5.3 and 5.4 can be reduced to be less than m in one step.*

Proof. Looking at Algorithms 5.3 and 5.4 there are two terms which will require modular reduction: one is $A \cdot B_i$ with highest degree $(m-1)+(D-1) = m+D-2$ and the other is $C\alpha^D$ with maximum degree $m + D - 1$. So assume that the intermediate value is as follows:

$$W = w_{m+D-1}\alpha^{m+D-1} + \cdots + w_m\alpha^m + w_{m-1}\alpha^{m-1} + \cdots + w_0 \tag{5.8}$$

Assuming $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{j=0}^{k-1} q_j\alpha^j$, we can substitute $\alpha^m = -\sum_{j=0}^{k} q_j\alpha^j$ into (5.8). Thus, obtaining

$$W = \left(w_{m+D-1}\alpha^{D-1} + \cdots + w_m\right)\left(-\sum_{j=0}^{k} q_j\alpha^j\right) + w_{m-1}\alpha^{m-1} + \cdots + w_0$$

When $D \le m-k$, $D-1 \le m-k-1$ and by Theorem 5.1 the degree of each intermediate result $w_{m+t-1}\alpha^{t-1}\left(-\sum_{j=0}^{k} q_j\alpha^j\right)$ for $1 \le t \le D$ is less than m. □

Theorems 5.1 and 5.2, implicitly say that for a given irreducible polynomial $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{j=0}^{k-1} q_j\alpha^j$, the digit-element size will depend on the value of k.

5.4.4 Area/Time Complexity of LSDE Multipliers for Optimal Irreducible Polynomials

Before estimating the complexity of the LSDE multiplier, it is helpful to obtain equations to describe the values of A and C at iteration i in Algorithm 5.3. Thus, assume that B_i is as in (5.5), $q(\alpha)$ as in Theorem 5.1, $A = \sum_{i=0}^{m-1} a_i\alpha^i$, and $D \le m-k$ (Theorem 5.2). Then,

$$\begin{aligned} C^{(i)} &= D^{(i)} + C^{(i-1)} = \sum_{j=0}^{m+D-2}\left(d_j^{(i)} + c_j^{(i-1)}\right)\alpha^j && (5.9)\\ A^{(i)} &= \sum_{j=D}^{m-1} a_{j-D}^{(i-1)}\alpha^j + \sum_{s=0}^{k}\sum_{j=0}^{D-1}\left(-q_s \cdot a_{j+m-D}^{(i-1)}\right)\alpha^{j+s} && (5.10)\end{aligned}$$

where $C^{(-1)} = 0$, $A^{(-1)} = A$, and

$$\begin{aligned} D^{(i)} &= \sum_{j=0}^{m+D-2} d_j^{(i)}\alpha^j = B_i \cdot A^{(i-1)} = \left(\sum_{j=0}^{m-1} a_j^{(i-1)}\alpha^j\right)\left(\sum_{s=0}^{D-1} b_{Di+s}\alpha^s\right)\\ &= \sum_{j=0}^{m-1}\sum_{s=0}^{D-1}\left(a_j^{(i-1)} \cdot b_{Di+s}\right)\alpha^{j+s} && (5.11)\end{aligned}$$

It follows from (5.11) that in each iteration one requires mD multipliers in parallel and $\sum_{j=0}^{D-2} j + \sum_{j=D-1}^{m-1}(D-1) + \sum_{j=m}^{m+D-2}(m+D-2-j) = (D-1)(m-1)$ adders to compute $D^{(i)}$. Therefore, in total we need $(D-1)(m-1)+m+D-1 = mD$ adders to compute $C^{(i)}$ according to (5.9). Using a ripple adder architecture, the critical path is given by $D-1$ adder delays from the computation of $D^{(i)}$, one adder delay from the computation $d_j^{(i)} + c_j^{(i-1)}$ in (5.9), and one multiplier. Notice, however, that it is possible to improve the critical path delay of the LSDE multiplier by using a binary tree of adders[4]. Using this technique one would reduce the length of the critical path from D $GF(p)$ adders and one $GF(p)$ multiplier delays to $\lceil \log_2(D+1) \rceil$ adders and one multiplier delays. We use this, as our complexity for the critical path. The computation of (5.10) requires only $D(k+1)$ multipliers (notice that the second term of (5.10) looks exactly the same as $D^{(i)}$ in (5.11), except that the limits in the summation are changed) and at most $(D-1)k + k = Dk$ adders (where the $(D-1)k$ term comes from the double summation and the k other adders come from adding the single summation term to the double summation one in (5.10)).

If $q(\alpha)$ is an r-nomial, the complexity of computing (5.10) is reduced to $(r-1)D$ multipliers and at most $(r-2)D$ adders (notice that the first summation in (5.10) starts at $j = D$, thus, the first D coefficients resulting from the second summation do not need to be added to anything). We say *at most* because depending on the values of D and r and the distribution of the non-zero coefficients of $q(\alpha)$, some adders may be saved. Last, we should consider the final reduction in Step 6 of Algorithm 5.3. In Step 6, we get as input $C^{(d-1)} = \sum_{j=0}^{m+D-2} c_j^{(d)} \alpha^j$ and we want to obtain $C^{(d)}$ such that $\deg(C^{(d)}) < m$. This is accomplished in a similar way to (5.10) as follows

$$\begin{aligned} C^{(d)} &= \sum_{j=0}^{m-1} c_j^{(d)} \alpha^j = \sum_{j=0}^{m-1} c_j^{(d-1)} \alpha^j + \left(\sum_{s=0}^{k} (-q_s) \alpha^s \right) \left(\sum_{j=0}^{D-2} c_{j+m}^{(d-1)} \alpha^j \right) \\ &= \sum_{j=0}^{m-1} c_j^{(d-1)} \alpha^j + \sum_{s=0}^{k} \sum_{j=0}^{D-2} \left(-q_s c_{j+m}^{(d-1)} \right) \alpha^{j+s} \end{aligned} \tag{5.12}$$

Equation (5.12) requires in general $(k+1)(D-1)$ multipliers and $k(D-2)+k+D-2+1 = (k+1)(D-1)$ adders. If we use an r-nomial, the number of multipliers will be $(D-1)(r-1)$ but the number of adders will vary, as in the case of (5.10), depending on the values i for which $q_i \neq 0$. To give an upper bound, we assume that a general optimal r-nomial $q(x) = x^m + q_{s_{r-2}} x^{s_{r-2}} + q_{s_{r-3}} x^{s_{r-3}} + \cdots + q_{s_1} x + q_0$ with $s_{r-2} > s_{r-3} > \cdots > s_1 > 0$ and $s_{r-2} \leq m - D$ also satisfies $s_i = s_{i-1} + 1$ with $s_1 = 1$. It is easy to see that this assumption will provide us with the maximum number of adders (simply look at how to compute the final product in (5.12)). Then, computing (5.12) with an r-nomial will require at most $(r+D-3)+(D-2)(r-2) = (D-1)(r-1)$ adders. These results are summarized in Table 5.3

Table 5.3. Area complexity and critical path delay of LSDE multiplier with optimal irreducible polynomials.

Irreducible polynomial	Area Complexity	Critical Path Delay	Latency (# clocks)
r-nomial	$[(m+r-2)D+(D-1)(r-1)]$ ADD + $[(m+r-1)D+(D-1)(r-1)]$ MUL	$\lceil \log_2(D+1) \rceil$ ADD + 1 MUL	$\lceil \frac{m}{D} \rceil +$ $1-\delta(D,1)$
General	$[(m+k)D+(k+1)(D-1)]$ ADD + $[(m+k+1)D+(k+1)(D-1)]$ MUL	$\lceil \log_2(D+1) \rceil$ ADD + 1 MUL	$\lceil \frac{m}{D} \rceil +$ $1-\delta(D,1)$

In Table 5.3, $\delta(D,1)$ is equal to 1 if $D=1$ and 0 otherwise. The extra clock cycle when $D>1$ comes from the modular reduction in Step 6 of Algorithm 5.3 which is necessary because the intermediate results are of degree larger than $m-1$. Table 5.3 makes the same assumptions as for the LSE case. In other words, ADD and MUL refer to the area and delay of a $GF(p)$ adder and multiplier, respectively, delay or area of storage elements are not taken into account, and no distinction is made between general and constant $GF(p)$ multipliers. We end by noticing that an LSDE multiplier with $D=1$ is equivalent to an LSE multiplier. Our complexity estimates verify this if one lets $D=1$ and $k=m-1$ in Table 5.3.

5.4.5 Area/Time Complexity of MSDE Multipliers for Optimal Irreducible Polynomials

As in the LSDE case, we first find expressions for C in Algorithm 5.4 at iteration i. Thus,

$$C^{(i)} = A \cdot B_{d-1-i} + \left(C^{(i-1)} \bmod q(\alpha)\right) \alpha^D \tag{5.13}$$

The term $A \cdot B_{d-1-i}$ is of the same complexity as (5.10), thus requiring mD parallel multipliers and $(m-1)(D-1)$ parallel adders. Computing $C^{(i-1)} \bmod q(\alpha)$ is the same operation described in (5.12), except that now we need to reduce a polynomial of degree $\le m+D-1$ instead of a polynomial of degree $\le m+D-2$. For general optimal polynomials this implies an area complexity of $(k+1)D$ multipliers and $(k+1)D$ adders. For r-nomials this means $(r-1)D$ multipliers and at most $(r-1)D$ adders. In addition, we require an additional $m-1$ adders to add $A \cdot B_{d-1-i}$ and $\left(C^{(i-1)} \bmod q(\alpha)\right) \alpha^D$. Finally, notice that since we multiply and modulo reduce all in one step, the number of adders in the critical path is at most doubled. Table 5.4 summarizes these results.

Table 5.4. Area complexity and critical path delay of MSDE multiplier with optimal irreducible polynomials.

Irreducible polynomial	Area Complexity	Critical Path Delay	Latency (# clocks)
r-nomial	$[(m+r-2)D+(D-\delta(D,1))(r-1)]$ ADD + $[(m+r-1)D+(D-\delta(D,1))(r-1)]$ MUL	$\lceil\log_2(2D+1)\rceil$ ADD + 1 MUL	$\lceil\frac{m}{D}\rceil+$ $1-\delta(D,1)$
General	$[(m+k)D+(k+1)(D-\delta(D,1))]$ ADD + $[(m+k+1)D+(k+1)(D-\delta(D,1))]$ MUL	$\lceil\log_2(2D+1)\rceil$ ADD + 1 MUL	$\lceil\frac{m}{D}\rceil+$ $1-\delta(D,1)$

5.4.6 Comments on Irreducible Polynomials of Degree m over $GF(p)$

From Theorems 5.1 and 5.2, it is obvious that choosing an irreducible polynomial should be carefully done. In this section, we give some guidelines regarding the selection of irreducible polynomials.

For fields $GF(p^m)$ with odd prime characteristic it is often possible to choose irreducible binomials $q(\alpha) = x^m - \omega$, $\omega \in GF(p)$. This is particularly interesting since binomials are never irreducible in characteristic 2 fields. Another interesting property of binomials is that they are optimum from the point of view of Theorem 5.1. In particular, for any irreducible binomial $q(\alpha) = x^m - \omega$, it holds that $k = 0$ and $D \leq m$ in Theorem 5.2. This implies that even in the degenerate case where $D = m$ (parallel multiplier case) one is able to perform the reduction in one step. In addition, reduction is virtually for free, corresponding to just a few $GF(p)$ multiplications (this follows from the fact that $\alpha^m = \omega$). A specific sub-class of these fields where q is a prime of the form $q = p = 2^n - c$, c "small", has recently been proposed for cryptographic applications in [BP01a]. We notice that the existence of irreducible binomials has been exactly established as Theorem 5.3 shows[5].

Theorem 5.3. *[LN97] Let $m \geq 2$ be an integer and $\omega \in F_q^\star$. Then the binomial $x^m - \omega$ is irreducible in $F_q[x]$ if and only if the following two conditions are satisfied: (i) each prime factor of m divides the order e of ω in $F_q^\star$, but not $(q-1)/e$; (ii) $q \equiv 1 \bmod 4$ if $m \equiv 0 \bmod 4$.*

When irreducible binomials can not be found, one searches in incremental order for irreducible trinomials, quadrinomials, etc. In [vzGN00] von zur Gathen and Nöcker conjecture that the minimal number of terms $\sigma_q(m)$ in irreducible polynomials of degree m over $GF(q)$, q a power of a prime, is for $m \geq 1$, $\sigma_2(m) \leq 5$ and $\sigma_q(m) \leq 4$ for $q \geq 3$. This conjecture has been verified for $q = 2$ and $m \leq 10000$

[BGL93, Gol67, vzGN00, Zie70, ZB68, ZB69] and for $q = 3$ and $m \leq 539$ [vzG01].

By choosing irreducible polynomials with the least number of non-zero coefficients, one can reduce the area complexity of the LSDE multiplier (this follows directly from Table 5.3). We point out that by choosing irreducible polynomials such that their non-zero coefficients are all equal to $p - 1$ one can further reduce the complexity since all the multiplications by $-q_s$ in (5.10) reduce to multiplication by 1. We point out that there is no existence criteria for irreducibility of trinomials over any field $GF(p^m)$. The most recent advances in this area are the results of Loidreau [Loi00], where a table that characterizes the parity of the number of factors in the factorization of a trinomial over $GF(3)$ is given, and the necessary (but not sufficient) irreducibility criteria for trinomials introduced by von zur Gathen in [vzG01]. Neither reference provides tables of irreducible polynomials.

5.5 Case Study: $GF(3^m)$ Arithmetic

5.5.1 $GF(3)$ Arithmetic Implementation on FPGAs

Field Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices whose basic logic elements are Look-Up Tables (LUTs), sometimes also called Configurable Logic Blocks (CLBs), flip-flops (FFs), and, for modern devices, memory elements [Act01, Alt01, Xil00]. The LUTs are used to implement Boolean functions of their inputs, that is, they are used to implement functions traditionally implemented with logic gates. In the particular case of the XCV1000E-8-FG1156 and the XC2VP20-7-FF1156, their basic building blocks are 4-bit input/1-bit output LUTs. This means that all basic arithmetic operations in $GF(3)$ (add, subtract, and multiply) can be done with 2 LUTs, where each LUT generates one bit of the output. This follows from the fact that any of these arithmetic operations over $GF(3)$ can be thought of as logic functions in 4 input variables a_1, a_0, b_1, b_0 and 2 output variables c_1, c_2 as:

$$f : \quad I^4 \longrightarrow O^2$$

where $I = \{0, 1\}$ and $O = \{0, 1\}$. Then, given three elements $a = (a_1, a_0)_2, b = (b_1, b_0)_2, c = (c_1, c_0)_2 \in GF(3)$, we can write the function "multiplication in $GF(3)$" as Table 5.5. In Table 5.5,

Table 5.5. Truth table representing multiplication in $GF(3)$.

$a_1\,a_0\,b_1\,b_0$	$c_1\,c_0$	$a_1\,a_0\,b_1\,b_0$	$c_1\,c_0$
0 0 0 0	0 0	1 0 0 0	0 0
0 0 0 1	0 0	1 0 0 1	1 0
0 0 1 0	0 0	1 0 1 0	0 1
0 0 1 1	0 0	1 0 1 1	0 0
0 1 0 0	0 0	1 1 0 0	0 0
0 1 0 1	0 1	1 1 0 1	0 0
0 1 1 0	1 0	1 1 1 0	0 0
0 1 1 1	0 0	1 1 1 1	0 0

we have assumed that $(1, 1)$ is an alternate representation for $0 \in GF(3)$. Notice that it is possible to choose different representations as shown in [GWP02]. This might minimize the complexity of the $GF(3)$ multiplier in ASIC-based designs. However, in FPGA based designs, a different encoding has no advantages because of the LUT-based structure of the FPGA.

5.5.2 Cubing in $GF(3^m)$

It is well known that for $A \in GF(p^m)$ the computation of A^p (also known as the Frobenius map) is linear. In the particular case of $p = 3$, we can write the Frobenius map as:

$$A^3 \bmod q(\alpha) = \left(\sum_{i=0}^{m-1} a_i \alpha^i\right)^3 \bmod q(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^{3i} \bmod q(\alpha) = \tag{5.14}$$

Equation (5.14) can in turn be written as the sum of three terms (notice that here we have re-written the indexes in the summation):

$$\begin{aligned} A^3 \bmod q(\alpha) &= \sum_{\substack{i=0 \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^i \bmod q(\alpha) = T + U + V \bmod q(\alpha) = \\ &= \left(\sum_{\substack{i=0 \\ i\equiv 0 \bmod 3}}^{m-1} a_{\frac{i}{3}} \alpha^i\right) + \left(\sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}} \alpha^i\right) + \left(\sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}} \alpha^i\right) \bmod q(\alpha) \end{aligned} \tag{5.15}$$

Notice that only U and V need to be reduce $\bmod q(\alpha)$. We further assume that $q(x) = x^m + q_t x^t + q_0$ and that $t < m/3$. This assumption proves to be a valid one in terms of the existence of such irreducible trinomials as we show in Section 5.5.3. Then, we obtain:

$$
\begin{aligned}
U &= \sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}}\alpha^i \bmod q(\alpha) = \sum_{\substack{i=m \\ i\equiv 0 \bmod 3}}^{2m-1} a_{\frac{i}{3}}\alpha^{i-m}\left(-q_t\alpha^t - q_0\right) \bmod q(\alpha) \\
V &= \sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}}\alpha^i \bmod q(\alpha) = \sum_{\substack{i=2m \\ i\equiv 0 \bmod 3}}^{3(m-1)} a_{\frac{i}{3}}\alpha^{i-2m}\left(\alpha^{2t} - q_t q_0\alpha^t + 1\right) \bmod q(\alpha)
\end{aligned}
$$

where we have made use of the fact that $(-q_t\alpha^t - q_0)^2 = (\alpha^{2t} - q_t q_0\alpha^t + 1)$ in $GF(3)$. It can be shown that U and V can be reduced to be of degree less than m in one extra reduction step. To estimate the complexity of this cubing circuit, we assume that $q(\alpha)$ is a trinomial with $t < m/3$, and that the circuit is implemented for fixed irreducible trinomials, i.e., that multiplications in $GF(3)$ (for example multiplying by $-p_t p_0$) can be handled by adders and subtracters. Then, it can be shown that one needs in the order of $2m$ adders/subtracters to perform a cubic operation in $GF(3^m)$.

5.5.3 Irreducible Polynomials over $GF(3)$

If we follow the criteria of Section 5.4.6 for choosing irreducible polynomials, we would try to find irreducible binomials first. Unfortunately, the only irreducible binomial over $GF(3)$ is $x^2 + 1$, thus we have to consider irreducible trinomials. Notice that $x^m + x^t + 1$ is never irreducible over $GF(3)$ since 1 is always a root of it. Therefore, we only need to search for irreducible trinomials of the following forms: $x^m - x^t - 1$ or $x^m \pm x^t \mp 1$. For $2 \leq m \leq 255$, we exhaustively searched for trinomials. Our results are provided in Tables A.3, A.4, and A.5 in Appendix A.2. There are only 23 degrees m in the range above, for which we were unable to find trinomials (this agrees with the findings in [vzG01]) and thus, we provide quadrinomials for them in Table A.6. Of these quadrinomials, only 4 correspond to m prime $(149, 197, 223, 233)$. Prime m is the most commonly used degree in cryptographic applications.

Another criteria to choose irreducible polynomials is based on the value of the non-zero coefficients of $q(x)$ and on the degree of the second-highest non-zero coefficient in $q(x)$. From our discussion in Section 5.4.6, follows that choosing irreducible polynomials whose non-zero coefficients are all equal to -1 might be advantageous, thus we make emphasis on trinomials of the form $x^m - x^t - 1$. In addition, from our cubing circuit discussion follows that polynomials $x^m + q_t x^t + q_0$, with $t < m/3$ are also desirable. Putting all these criteria together we come up with the *optimum* polynomials of Table A.2. In doing so, we follow these rules:

1. For all m, whenever a trinomial $x^m - x^t - 1$ with $t < m/3$ exists we include it in Table A.2.
2. If there is no trinomial as in 1 but there is a trinomial(s) with $t < m/3$, we include them in Table A.2.
3. If both 1 and 2 fail, we write all available trinomials for that degree.

As a final remark, we notice that of the 50 primes in the range $2 \leq m \leq 255$ which had trinomials, we were not able to find trinomials with $t < m/3$ for 9 of them (18 %).

5.6 $GF(3^m)$ Prototype Implementation and Comparisons

Figure 5.1 shows a block diagram of the prototyped[6] arithmetic unit (AU). In Figure 5.1, all bus-widths correspond to how many $GF(3)$ elements can be carried by the bus. In other words, if we write m, then it is understood that the bus is $2m$-bit wide.

The AU consists of an LSDE multiplier and a cubic circuit. The multiplier and the cubic circuit support the computation of field additions, squares, multiplications, and inversions. For addition and subtraction we take advantage of the multiply/accumulate capabilities of the LSDE multiplier and cubing circuit. In other words, the addition $C = A + B$ is done by first computing $A \cdot 1$ and then adding to it the product $B \cdot 1$. This takes two clock cycles. However, if operand A is already in the accumulator of the multiplier one can compute $C = B \cdot 1 + A$ in one clock. This addition eliminates the need for an adder. Subtractions are computed in a similar manner. The subtraction $C = A - B$ is done by first computing $A \cdot 1$ and then adding to it the product $(-1) \cdot B$ or alternatively as $C = (-1) \cdot B + A$.

AU prototypes were developed to verify the suitability of the architecture shown in Figure 5.1 for reconfigurable FPGA logic and compare the efficiency of $GF(3^m)$ and $GF(2^m)$ AUs. The prototypes were coded in VHDL at a very low level. The VHDL code was synthesized using Synopsis FPGA Compiler 3.7.1 and the component placement and routing was done using XILINX Design Manager 4.2.03i.

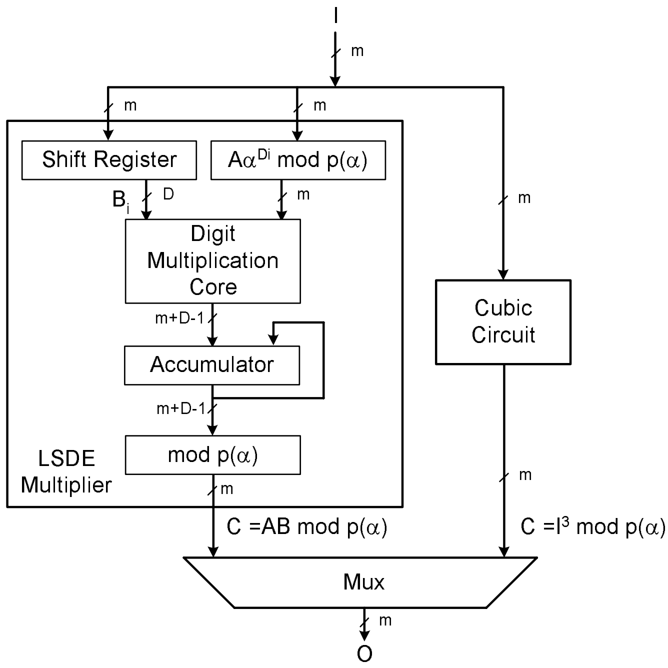


Figure 5.1. $GF(3^m)$ Arithmetic Unit Architecture

The prototypes were synthesized and routed for the XILINX XCV1000-8-FG1156 and XC2VP20-7-FF1156 FPGAs. The XCV1000E-8-FG1156 prototype allowed us to compare our AU implementations against the AU for $GF(2^m)$ used in the elliptic curve processor (ECP) documented in [OP00]. The XC2VP20-7-FF1156 prototype allowed us to verify the speed of our AU for one of the newest families of XILINX FPGAs. Three implementation were developed which support the fields $GF(3^{97})$, $GF(2^{151})$, and $GF(2^{241})$. The fields $GF(3^{97})$ and $GF(2^{241})$ are used in Weil and Tate pairing schemes for systems with comparable degrees of security (see [GHS02a, BKLS02, PS02]). The field $GF(2^{151})$ offers security comparable to that of $GF(3^{97})$ for cryptosystems based on the elliptic curve discrete logarithm problem (ECDLP).

5.6.1 $GF(3^m)$ Complexity Estimates

Table 5.6 shows the complexity estimates for the AU shown in Figure 5.1. The estimates assume the use of optimum irreducible polynomials. The estimates give the register complexity in terms of the number of flip-flops. Note that the register estimates do not account for registers used to reduce the critical path delay of a multiplier, a technique known as pipelining. This technique was used to reduce the critical path delay of the prototype implementations. The complexity estimates are based on the

Table 5.6. $GF(3^m)$ AU complexity estimates

Circuit	Area Complexity $GF(3^{97})$ ($q(x) = x^{97} + x^{12} + 2$)
LSE Mult.	$(4mD + 2m + 6D)$ LUT + $6m + 2D + 4$ FF
Cubic	$4m$ LUT
Mux	$2m$ LUT
AU (total)	$(4mD + 8m + 6D)$ LUT + $6m + 2D + 4$ FF

following assumptions:

1. A $GF(3)$ adder, subtracter, or multiplier requires two LUTs, including adders that add weighted inputs, for example, adders that compute $(a_i * c) + (b_i * d)$ where c and d are fixed constants. In addition, a 2:1 multiplexer requires one LUT.

2. From Table 5.3 the digit multiplication core and the accumulator circuits require mD $GF(3)$ multipliers and mD $GF(3)$ adders. This circuit stores the result in two $(m + D - 1)$-bit registers. An m-bit register requires m flip-flops (FFs).

3. The estimates for the $A\alpha^{Di} \bmod q(\alpha)$ circuit assume that the circuit contains two m-bit multiplexers that select between the element A and the element $A\alpha^{Di} \bmod q(\alpha)$. An m-bit multiplexer requires m LUTs. For programmable optimum irreducible trinomials, the circuitry that generates $A\alpha^{Di} \bmod q(\alpha)$ requires $2D$ $GF(3)$ multipliers and D adders (see Table 5.3). This circuit stores the result in two m-bit registers and the coefficients of $q(x)$ in two $2r$-bit registers ($r = 3$ for trinomials).

4. The estimates assume that the coefficients of B are fed in by two m-bit parallel in/serial out shift registers. Each of these shift registers contains m 2:1 multiplexers and m registers.

5. The cubic circuit requires $2m$ $GF(3)$ adders.

6. The complexity for the $GF(2^m)$ AU is done accordingly to the models in [Orl02]. We have also assumed that the $GF(2^m)$ AU contains an LSD multiplier and a squarer.

Table 5.7. AU estimated vs. measured complexity for prototypes ($D = 16$)

Circuit	$\lfloor \log_2(p^m) \rfloor$	Estimated complexity	Measured complexity (incl. pipelining registers & I/O)	LUT Estimate Error ((actual-est.)/est.)
$GF(2^{151})$	151	2366 LUT + 453 FF ($15.7m$ LUT + $3m$ FF)	2239 LUT + 1092 FF ($14.3m$ LUT + $7.2m$ FF)	-5.4 %
$GF(2^{241})$	241	3705 LUT + 723 FF ($14.9m$ LUT + $3m$ FF)	3591 LUT + 1722 FF ($15.4m$ LUT + $7.1m$ FF)	-3.1 %
$GF(3^{97})$	153	7080 LUT + 618 FF ($73.0m$ LUT + $6.4m$ FF)	7122 LUT + 2790 FF ($73.4m$ LUT + $7.2m$ FF)	1.0 %

We observe that the estimates obtained from our models are very accurate when compared to the actual measured complexities. This validates our models and assumptions.

5.6.2 Results

Table 5.8 presents the timings obtained for our three prototypes. We have tried to implement our designs in such a way that we can make a meaningful comparison. Thus, although, the clock rates are not exactly the same between the different designs (this is due to the fact that the clock rate depends on the critical path of the AU which is different for each circuit), they are not more than 10 % different. The platforms are the same and we chose same digit sizes for both $GF(2^m)$ and $GF(3^m)$ architectures. The results make sense, for the same digit size ($D = 16$) we obtain that the $GF(3^{97})$ design is about twice as big as the $GF(2^{241})$ design and more than 3 times the size of the $GF(2^{151})$ AU. This of course is offset by the gain in performance. At similar clock rates the $GF(3^{97})$ design is 2.7 times faster than the corresponding $GF(2^{241})$ AU and 1.4 times faster than the $GF(2^{151})$ one.

Table 5.8. Comparison of multiplication time for $GF(2^{151})$, $GF(2^{241})$, and $GF(3^{97})$ prototypes ($D = 16$) and the AU from [PS02]

Circuit	Mult. time for optimized mult. [PS02] (in μsec)	Mult. time for prototypes XCV1000-8-FG1156 (in μsec)	Mult. time for prototypes XC2VP20-7-FF1156 (in μsec)
$GF(2^{151})$	N/A	0.139 (@ 71.7 MHz)	0.100 (@ 100.2 MHz)
$GF(2^{241})$	37.32	0.261 (@ 61.3 MHz)	0.150 (@ 107 MHz)
$GF(3^{97})$	50.68	0.097 (@ 72 MHz)	0.074 (@ 94.4 MHz)

It is clear that by using more hardware resources for $GF(3^{97})$ we can achieve better performance than fields of characteristic 2. In particular, we point out that by choosing the same digit size for both types of fields, we are implicitly processing twice as many bits of the multiplier in $GF(3^{97})$ as in the $GF(2^m)$ cases (remember that is why we introduced the notation of LSDE, where the E refers to elements of $GF(p)$ and not bits as in the $GF(2^m)$ case). Table 5.8 also includes the results presented in [PS02]. Unfortunately, the authors in [PS02] used the Celoxica Handel-C hardware compilation system [Cel02] and a PCI resident XILINX4000XL FPGA based prototyping device. The Handel-C language allows the designer to describe hardware in a high-level language similar to C and use a compiler to map this code to synthesizable VHDL code. Thus, we do not think it is possible to make a meaningful comparison, other than point out that by coding directly in VHDL, one can improve the performance of FPGA based implementations.

5.7 Notes and Further References

1. There has been a lot work done, however, on finite field architectures for characteristic two fields. See for example [YP82, Mas89, HWB92, AMV93, ABMV93, FBT96, SP98, PFR99].

2. It is widely accepted that for cryptosystems against which the Pollard's rho algorithm or one of its variants are the best available attacks, such as elliptic curve cryptosystems, the group order should be greater or equal to 2^{160}. Thus, solving $2m < p$ and $p^m \geq 2^{160}$ for p and m, one obtains $p \geq 67$. Notice that the value of p grows as the size of the desired group grows. For groups with $|G| \geq 2^{192}$, $|G| \geq 2^{223}$, and $|G| \geq 521$, the prime p satisfies $p \geq 67$, $p \geq 79$, and $p \geq 157$, respectively.

3. Look-Up Tables are the basic building blocks of most common FPGAs [Act01, Alt01, Xil00].

4. Binary trees have been used both in [SP98] and [Orl02] in the context of $GF(2^n)$ arithmetic to reduce delay and power consumption.

5. Reference [LN97] is used in this setting as a convenient reference for well established results.

6. The synthesis was joint work General Dynamics Communication Systems, USA. The design and the running of the tools were performed at General Dynamics Communication Systems. Special thanks to Dr. Gerardo Orlando.

CHAPTER 6

Systolic and Scalable Architectures for Digit-Serial Multiplication in Fields $GF(p^m)$

The research community's interest in cryptographic systems based on fields of odd characteristic and the lack of hardware architectures for general odd characteristic fields is evident. Reference [BGK+03] has given a partial answer to this problem but their methods have the drawback of using global signals and long wires and they require reconfigurability to achieve their full potential (for example, [BGK+03] uses irreducible trinomial specific circuitry to perform modular reduction on FPGAs), and thus, these solutions lack flexibility in other hardware platforms such as ASICs. Hence, in this chapter, we move a step forward towards the design of scalable and flexible hardware architectures for odd $GF(p^m)$ fields. In particular, we propose systolic architectures for arithmetic in $GF(p^m)$ fields. Systolic architectures solve the previously mentioned problems in several ways. First, by using a systolic design we use localized routing, thus avoiding the need for global control signals and long wires. In addition, this methodology allows for ease of design and offers functional and layout modularity all of which are properties envisioned in good VLSI designs. Second, we modify the design of [BGK+03] to allow for scalability as introduced in [TcKK99]. In other words, for a fixed value of the digit-size D [SP98, BGK+03] and parameter d, we can perform a multiplication for any value of m in $GF(p^m)$, with fixed p, i.e., we support multiple irreducible polynomials making unnecessary the use of reconfigurability in

FPGAs. Thus, these architectures are well suitable for very large multipliers and a large number of hardware platforms, including FPGAs and ASICs. Parts of this chapter appear in [BGO03].

6.1 Related Work

There has been significant work on systolization of modular multiplication (i.e. multiplication in $GF(p)$). The first attempt to provide systolic architectures for modular multiplication was presented in [KH91]. However, these architectures suffered from excessive latency and a slow clock as a result of the unsuitability of the regular multiply–and–then–reduce algorithm to systolization. The first systolic architectures for Montgomery multiplication were introduced in [Wal93]. In particular, [Wal93], takes advantage of simplifications to the Montgomery algorithm presented in [EW93] to minimize the complexity of the array cells and their execution time. Throughout the years, several systolic architecture designs of the binary type (using radix 2 to represent and process operands) have been introduced, including both 1D-based arrays [Kor94, CHCW99, TSW00] and 2D-based arrays [JB97]. Higher-radix systolic arrays have also been proposed, as for example [Tak92, Wal97, FP00, BP01b]. Notice that the previously mentioned works include both Montgomery and non-Montgomery based architectures.

We also build on the concept of scalability presented in [TcKK99] and further generalized to higher radix Montgomery multipliers in [STcKK00, TcKK01]. Notice that [TcKK99, STcKK00, TcKK01] are only concerned with Montgomery multiplication over $GF(2^k)$ and $GF(p)$. Scalability, as defined in [TcKK99], means that an arithmetic unit (AU) (in our case an AU to perform arithmetic operations in $GF(p^m)$) can be used or replicated in order to generate long-precision[1] results independently of the data path precision for which the unit was originally designed. A very important design choice in [TcKK99] is the use of a word-based algorithm. Rather than processing one of the inputs (multiplier or multiplicand) in a bitwise manner, [TcKK99] uses circuits which process multiple bits of the operands at a time. Based on the data dependencies of the Montgomery algorithm, the authors define processing units which, when combined with pipelining and word-operand processing, result in scalable architectures for modular multiplication. The authors in [TcKK99] also examine trade-offs among desired performance, area, number of processing units, operand word-length, and precision.

6.2 Systolic Least-Significant Element (LSE) First Architecture

The LSE scheme from Chapter 5 processes first coefficient b_0 of the multiplier and continues with the remaining coefficients one at a time in ascending order. This multiplier computes the operation according to Algorithm 5.1 which we reproduce here as Algorithm 6.1 for ease of presentation. Step 3

Algorithm 6.1 LSE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$, $q(\alpha) = \alpha^m + \sum_{i=0}^{m-1} q_i\alpha^i$ where $a_i, b_i, q_i \in GF(p)$
Output: $C \equiv A \cdot B \bmod q(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$, where $c_i \in GF(p)$

1: $C \leftarrow 0$
2: **for** $i = 0$ to $m-1$ **do**
3: $\quad C \leftarrow b_iA + C$
4: $\quad A \leftarrow A\alpha \bmod q(\alpha)$
5: **end for**
6: Return (C)

in Algorithm 6.1 requires the accumulation of the partial product which is achieved via a polynomial adder. In Step 4 of Algorithm 6.1 the quantity $A\alpha \bmod q(\alpha)$, has to be computed. Thus, $A\alpha = \sum_{i=0}^{m-1} a_i\alpha^{i+1} = a_{m-1}\alpha^m + \sum_{i=0}^{m-2} a_i\alpha^{i+1}$. Then, using (5.1) and re-writing the index of the second summation, $A\alpha \bmod q(\alpha)$ can be calculated as follows:

$$A\alpha \bmod q(\alpha) \equiv (-q_0a_{m-1}) + \sum_{i=1}^{m-1}(a_{i-1} - q_ia_{m-1})\alpha^i \tag{6.1}$$

where all coefficient arithmetic is done modulo p. Using (6.1) we can write expressions for A and C in Algorithm 6.1 at iteration i as follows:

$$\begin{aligned} C^{(i)} &= \sum_{j=0}^{m-1} c_j^{(i)}\alpha^j \equiv b_iA^{(i)} + C^{(i-1)} = \sum_{j=0}^{m-1}(b_ia_j^{(i)} + c_j^{(i-1)})\alpha^j, && (6.2)\\ A^{(i)} &= \sum_{j=0}^{m-1} a_j^{(i)}\alpha^j \equiv A^{(i-1)}\alpha \equiv (-q_0a_{m-1}^{(i-1)}) + \sum_{j=1}^{m-1}(a_{j-1}^{(i-1)} - q_ja_{m-1}^{(i-1)})\alpha^j && (6.3)\end{aligned}$$

with $C^{(-1)} = 0$ and $A^{(-1)} = A$. Notice that if C is initialized to a value different from 0, say I, before beginning the algorithm, Algorithm 6.1 computes $C \equiv A \cdot B + I \bmod q(\alpha)$. This multiply-accumulate operation turns out to be very useful in elliptic curve systems and it is obtained at no extra cost. Using

(6.2) and (6.3), one can define Algorithm 6.2, which is a low-level version of Algorithm 6.1.

Algorithm 6.2 Low Level LSE Multiplier

Input: $A = \sum_{i=0}^{m-1} a_i\alpha^i$, $B = \sum_{i=0}^{m-1} b_i\alpha^i$, $q(\alpha) = \alpha^m + \sum_{i=0}^{m-1} q_i\alpha^i$ where $a_i, b_i, q_i \in GF(p)$
Output: $C \equiv A \cdot B \bmod q(\alpha) = \sum_{i=0}^{m-1} c_i\alpha^i$ where $c_i \in GF(p)$
1: $C \leftarrow 0$
2: **for** $i = 0$ to $m-1$ **do**
3:   **for** $j = m-1$ to 0 **do**
4:     $c_j \leftarrow b_i a_j + c_j$
5:   **end for**
6:   **for** $j = m-1$ to 0 **do**
7:     $a_j \leftarrow a_{j-1} - q_j a_{m-1}$ {Note: $a_{-1} = 0$}
8:   **end for**
9: **end for**
10: Return (C)

6.2.1 Architecture

In this section, we analyze data dependencies of Algorithm 6.2. Steps 4 and 7 in Algorithm 6.2 are completely independent of each other. In other words, at iteration i, one can calculate coefficient c_j of C and, at the same time, compute a_j for iteration $i+1$ of the outer loop. To best study the data dependencies in Algorithm 6.2, two dependency graphs (DG) are used. Figure 6.1 shows the DG for the computation of $A\alpha \bmod q(\alpha)$. Every square corresponds to the computation of Step 7 in Algorithm 6.2. Thus, each cell contains one $GF(p)$ multiplier and a $GF(p)$ adder[2]. Each column corresponds to a new iteration of the outer loop (i-loop). Notice that because of the dependence of $a_j^{(i)}$ on $a_{j-1}^{(i-1)}$, there is a two cycle delay between the processing of a column at iteration i and $i+1$ (this can be better seen on Figure 6.2, where the b_i's are labeled). Figure 6.2 shows the DG for Algorithm 6.2 as a whole. This figure is obtained by superimposing the computation of Step 4 on Figure 6.1. To make this clearer, we have used dotted lines to indicate the inputs and outputs corresponding to Step 4 while using solid lines for those that correspond to Step 7 of Algorithm 6.2. Figure 6.2, contains two types of cells. The white cells can compute the values of c_j and a_j for the next i iteration. The black cells can compute $c_{m-1}^{(i)}$ and $c_{m-2}^{(i)}$ given the values of $c_{m-1}^{(i-1)}$ and $c_{m-2}^{(i-1)}$, respectively. In other words, they compute one more c_j value than the white cells. Thus, the white cells contain two $GF(p)$ multipliers and two $GF(p)$ adders while the black cells contain three $GF(p)$ multipliers and three $GF(p)$ adders. The critical path for

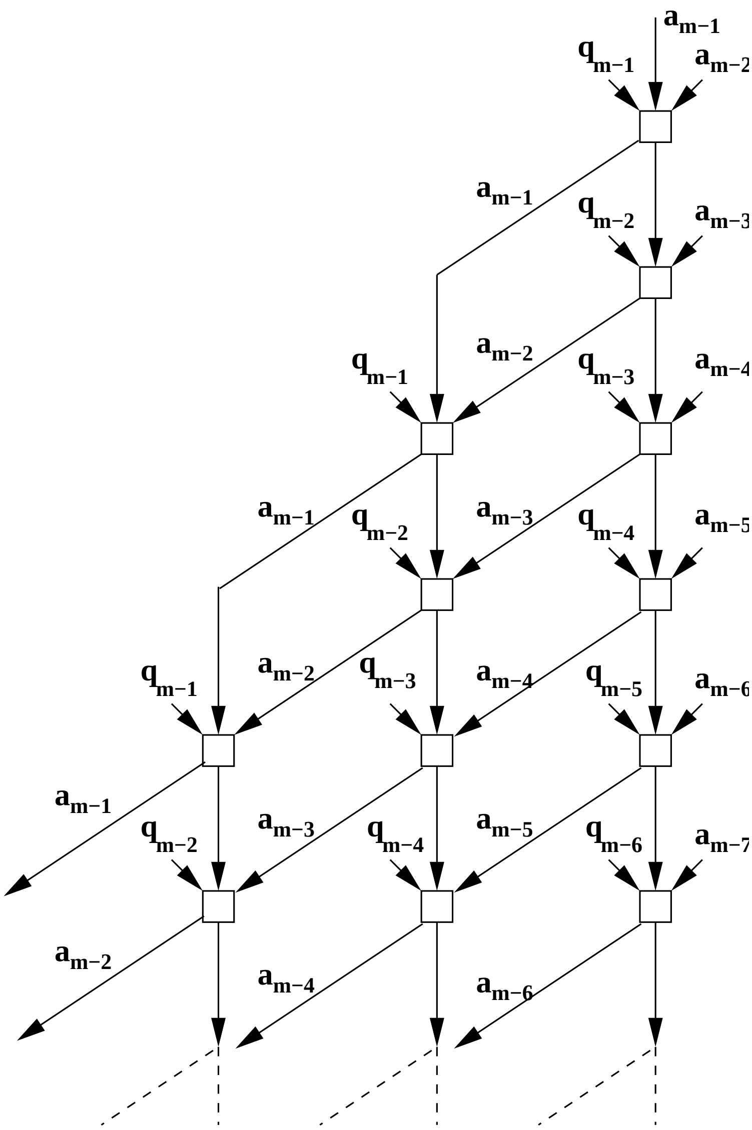


Figure 6.1. Dependency graph for $A\alpha \bmod q(\alpha)$

both types of cells is just the delay corresponding to one $GF(p)$ adder and one $GF(p)$ multiplier. As in Figure 6.1, there is a two cycle delay between columns which is a result of the dependence of $a_j^{(i)}$ on $a_{j-1}^{(i-1)}$. Similarly to [TcKK99], each column in Figure 6.2 may be computed by a different processing element (PE) and the data generated by one PE may be passed to another PE in a pipeline manner.

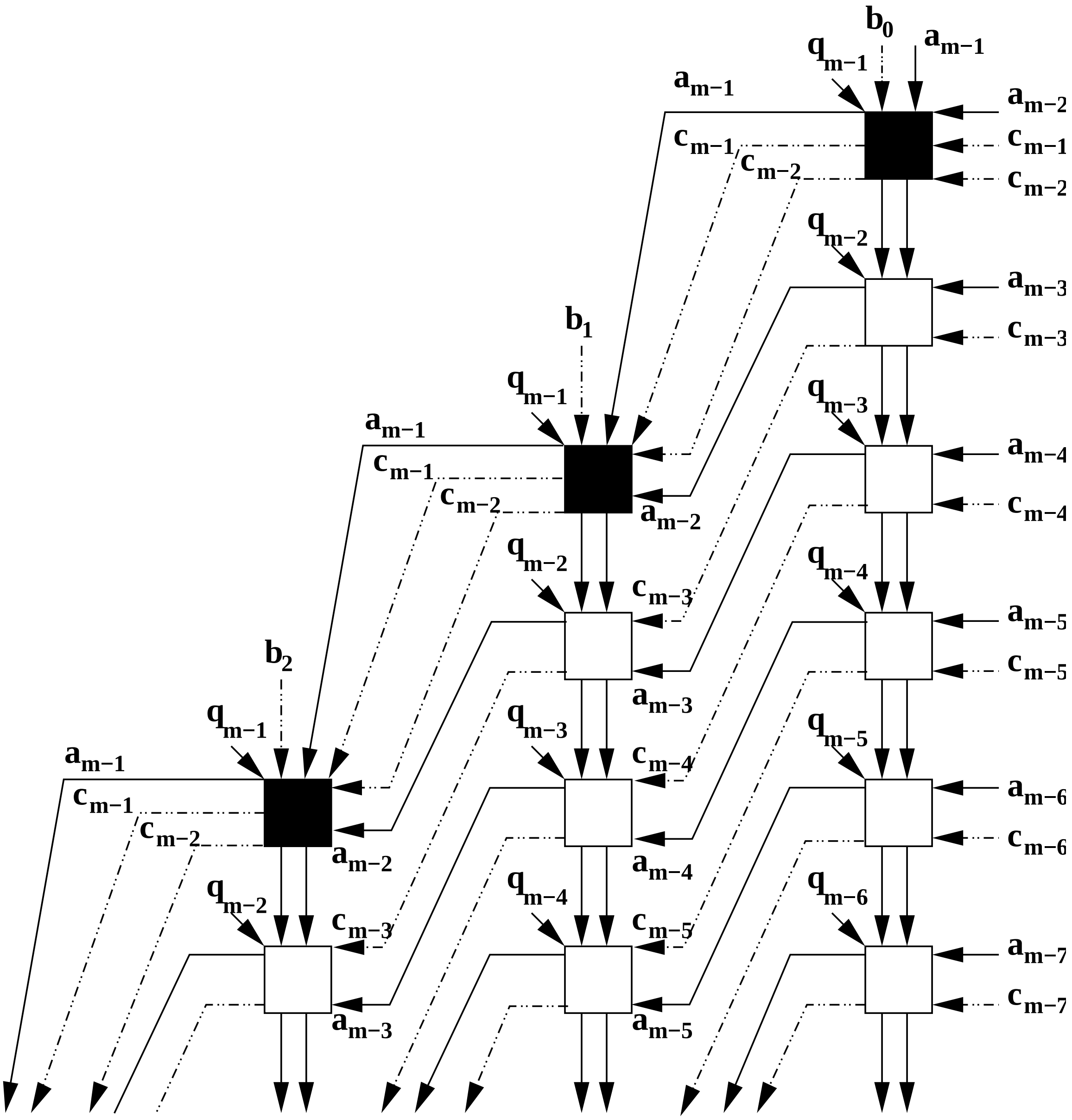


Figure 6.2. Dependency graph for the LSE algorithm

6.3 Systolic Least-Significant Digit Element (LSDE) First Architecture

This section develops the theory for the systolic Least-Significant Digit Element (LSDE) First multiplier. Digit multipliers, introduced in [SP98] in the context of fields $GF(2^k)$ and later generalized in [BGK+03], are a trade-off between speed, area, and power consumption. This is achieved by processing several of B's coefficients at the same time. The number of coefficients that are processed in parallel is defined to be the digit-size and we denote it by D. For a digit-size D, we can denote by $d = \lceil m/D \rceil$ the total number of digits in a polynomial of degree $m-1$. Thus, we can re-write the multiplier as $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$, where $B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j$ with $0 \leq i \leq d-1$ and we assume that B has been padded with zero coefficients such that $b_i = 0$ for $m-1 < i < d \cdot D$ (i.e. B's size is $d \cdot D$ coefficients but $\deg(B) < m$). Hence,

$$\begin{aligned} C &\equiv AB \bmod q(\alpha) = A\sum_{i=0}^{d-1} B_i \alpha^{Di} \bmod q(\alpha) \qquad (6.4)\\ &\equiv [B_0 A + B_1(A\alpha^D \bmod q(\alpha)) + \ldots + B_{d-1}(A\alpha^{D(d-2)}\alpha^D \bmod q(\alpha))] \bmod q(\alpha) \end{aligned}$$

Using (6.4), one can derive an algorithm similar to Algorithm 6.1, substituting b_i in Step 3 by B_i (we process now D coefficients as opposed to just one coefficient of B) and multiplying A by α^D. The authors in [BGK+03] define optimal irreducible polynomials of the form $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{i=0}^{k-1} q_i\alpha^i$ as those which satisfy the constraint[3] $k \leq m - D$. These polynomials allow one to perform the reduction of $A\alpha^D$ modulo $q(\alpha)$ in one clock cycle. We illustrate the reduction $A\alpha^D \bmod q(\alpha)$ with a small example.

Example 6.1. Although not shown in (6.4), one can also process A in digits of size D. Let $A = \sum_{i=0}^{d-1} A_i\alpha^{Di} \in GF(p^m)$ with $d = \lceil m/D \rceil$, A_i a digit (i.e., a group of D $GF(p)$ coefficients), and $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{i=0}^{k-1} q_i\alpha^i$ an optimum irreducible polynomial in the sense of [BGK+03]. Notice that A is written in digit-form whereas $q(\alpha)$ is written in terms of its coefficients. Then,

$$
\begin{aligned}
A\alpha^D &\equiv \alpha^D \sum_{i=0}^{d-1} A_i \alpha^{Di} \bmod q(\alpha) = A_{d-1}\alpha^{Dd} + \sum_{i=1}^{d-1} A_{i-1}\alpha^{Di} \bmod q(\alpha) \\
A\alpha^D &\equiv A_{d-1}\alpha^{Dd-m}\alpha^m + \sum_{i=1}^{d-1} A_{i-1}\alpha^{Di} \bmod q(\alpha) = A_{d-1}\alpha^{Dd-m}\left(-q_k\alpha^k - \sum_{i=0}^{k-1} q_i\alpha^i\right) + \sum_{i=1}^{d-1} A_{i-1}\alpha^{Di}
\end{aligned}
$$

Notice that the first term in the reduced result depends on the value of m, in other words on the field size. In fact, one needs to multiply by α^{Dd-m}, which can be instantiated as a *variable* shifter in hardware. This might be undesirable if scalability of the multiplier is desired.

Before we continue we prove a small proposition, the result of which will be used in the development of new architectures.

Proposition 6.1. *Let* $A, B \in GF(p^m)$, $q(\alpha) = \alpha^m + \sum_{i=0}^{m-1} q_i\alpha^i$, *be an irreducible polynomial over* $GF(p)$, *and* $d = \lceil m/D \rceil$. *Then,* $A \cdot B \bmod q(\alpha) \equiv [A \cdot B \bmod \overline{q}(\alpha)] \bmod q(\alpha)$, *where* $\overline{q}(\alpha) = \alpha^{Dd-m}q(\alpha)$.

Proof. If $Dd = m$, then the result is immediate, so let's consider the case $Dd > m$. Define $R \equiv A \cdot B \bmod \overline{q}(\alpha)$. Then, $A \cdot B = Q\overline{q}(\alpha) + R = Q(\alpha^{Dd-m}q(\alpha)) + R$. Thus, we can write $R = A \cdot B - Q(\alpha^{Dd-m}q(\alpha))$, which implies $R \equiv A \cdot B \bmod q(\alpha)$. □

Intuitively, Proposition 6.1 says that we can perform reductions modulo $\overline{q}(\alpha) = \alpha^{Dd-m}q(\alpha)$ and still obtain a result which when reduced modulo $q(\alpha)$ returns the correct value. We make this more formal by first introducing Algorithm 6.3.

Algorithm 6.3 Modified LSDE Multiplier

Input: $A = \sum_{i=0}^{d-1} A_i\alpha^{Di}$ with $A_i = \sum_{j=0}^{D-1} a_{Di+j}\alpha^j$, $B = \sum_{i=0}^{d-1} B_i\alpha^{Di}$ with $B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j$, $\overline{q}(\alpha) = \alpha^{Dd-m}q(\alpha)$, $a_i, b_i \in GF(p)$, and $d = \lceil \frac{m}{D} \rceil$

Output: $\overline{C} \equiv A \cdot B \bmod \overline{q}(\alpha) = \sum_{i=0}^{d} \overline{C}_i\alpha^{Di}$ with $\overline{C}_i = \sum_{j=0}^{D-1} c_{Di+j}\alpha^j$, $c_i \in GF(p)$, and $d = \lceil \frac{m}{D} \rceil$

$\overline{C} \leftarrow 0$
for $i = 0$ to $d-1$ **do**
  $\overline{C} \leftarrow B_iA + \overline{C}$
  $A \leftarrow A\alpha^D \bmod \overline{q}(\alpha)$
end for
Return ($\overline{C} \bmod \overline{q}(\alpha)$)

Algorithm 6.3 suggests the following computation strategy. Given two inputs $A, B \in GF(p^m)$ one can compute $C \equiv A \cdot B \bmod q(\alpha)$ by first computing $\overline{C} \equiv A \cdot B \bmod \overline{q}(\alpha)$ using Algorithm 6.3 and, then, computing $C \equiv \overline{C} \bmod q(\alpha)$. The second step follows as a consequence of Proposition 6.1. In practice, the second step can be performed at the end of a long range of computations, similar to the procedure used when performing Montgomery multiplication. Step 4 in Algorithm 6.3 requires a modular multiplication. There, it would be desirable to reduce $A\alpha^D$ in just one iteration as in [BGK+03] and, at the same time, make the reduction process independent of the value of m and, thus, of the field $GF(p^m)$. Given $q(\alpha) = \alpha^m + \sum_{i=0}^{m-1} q_i \alpha^i$, we define

$$\begin{aligned}
\overline{q}(\alpha) &= \alpha^{Dd-m} q(\alpha) = \alpha^{Dd} + \alpha^{Dd-m} \sum_{i=0}^{m-1} q_i \alpha^i = \alpha^{Dd} + \sum_{i=0}^{m-1} q_i \alpha^{Dd+i-m} \\
&= \alpha^{Dd} + \sum_{i=0}^{Dd-1} \overline{q}_i \alpha^{Dd+i-m} = \alpha^{Dd} + \sum_{i=0}^{d-1} \overline{Q}_i \alpha^{Di} \qquad (6.5)
\end{aligned}$$

where $\overline{q}_i = 0$ for $0 \leq i < Dd - m$, $\overline{q}_i = q_{i+m-Dd}$ for $Dd - m \leq i < Dd$, and $\overline{Q}_i = \sum_{j=0}^{D-1} \overline{q}_{Di+j} \alpha^j$. Then, we can compute $A\alpha^D \bmod \overline{q}(\alpha)$ as follows:

$$\alpha^D A \bmod \overline{q}(\alpha) = \alpha^D \sum_{i=0}^{d-1} A_i \alpha^{Di} \bmod \overline{q}(\alpha) = A_{d-1}\alpha^{Dd} + \sum_{i=0}^{d-2} A_i \alpha^{D(i+1)} \bmod \overline{q}(\alpha)$$

Using (6.5), we obtain

$$\begin{aligned}
\alpha^D A \bmod \overline{q}(\alpha) &= A_{d-1} \left(- \sum_{i=0}^{d-1} \overline{Q}_i \alpha^{Di} \right) + \sum_{i=0}^{d-2} A_i \alpha^{D(i+1)} \bmod \overline{q}(\alpha) \\
&= \sum_{i=0}^{d-2} A_i \alpha^{D(i+1)} - \sum_{i=0}^{d-1} \left(A_{d-1} \overline{Q}_i \right) \alpha^{Di} \bmod \overline{q}(\alpha) \qquad (6.6)
\end{aligned}$$

In (6.6), we have kept the $\bmod \overline{q}(\alpha)$ because it is possible that $\deg \left(A_{d-1} \overline{Q}_{d-1} \alpha^{D(d-1)} \right) \geq Dd$, in which case it would require a further reduction. This problem might be solved by defining M-LSDE optimal polynomials as those in which $\overline{Q}_{d-1} = 0$ or $\overline{Q}_{d-1} = 1$. Theorem 6.1 summarizes the above discussion.

Theorem 6.1. *Let $A = \sum_{i=0}^{d-1} A_i \alpha^{Di}$ be as defined in Algorithm 6.3 and $\overline{q}(\alpha) = \alpha^{Dd-m} q(\alpha) = \alpha^{Dd} +$*

$\sum_{i=0}^{d-1} \overline{Q}_i \alpha^{Di}$ *be as defined in (6.5), in particular* $q(\alpha)$ *is irreducible over* $GF(p)$ *of degree m. Then, if* $\overline{Q}_{d-1} = 0$ *or* $\overline{Q}_{d-1} = 1$, $A\alpha^D \bmod \overline{q}(\alpha)$ *can be computed in one reduction step. Moreover,* $\overline{Q}_{d-1} = 0$ *implies that for* $q(\alpha) = \alpha^m + \sum_{i=0}^{m-1} q_i \alpha^i$*, coefficients* $q_{m-1} = q_{m-2} = \cdots = q_{m-D} = 0$*. Similarly, when* $\overline{Q}_{d-1} = 1$ *then* $q_{m-1} = q_{m-2} = \cdots = q_{m-D+1} = 0$*.*

Proof. Notice that $\alpha^D A \bmod \overline{q}(\alpha) = \sum_{i=0}^{d-2} A_i \alpha^{D(i+1)} - \sum_{i=0}^{d-1} \left(A_{d-1}\overline{Q}_i\right) \alpha^{Di} \bmod \overline{q}(\alpha)$ from (6.6). The first summation does not require any further reduction, thus we concentrate on the second one. If $\overline{Q}_{d-1} = 1$, then the largest degree possible in (6.6) would correspond to the term $A_{d-1}\overline{Q}_{d-1}\alpha^{D(d-1)}$ and $\deg\left(A_{d-1}\overline{Q}_{d-1}\alpha^{D(d-1)}\right) \leq (D-1)+D(d-1) \leq Dd-1 < Dd$. On the other hand, if $\overline{Q}_{d-1} = 0$, (6.6) becomes

$$\alpha^D A \bmod \overline{q}(\alpha) = \sum_{i=0}^{d-2} A_i \alpha^{D(i+1)} - \sum_{i=0}^{d-2} \left(A_{d-1}\overline{Q}_i\right) \alpha^{Di} \tag{6.7}$$

Now it is easy to verify that $\deg\left(A_{d-1}\overline{Q}_{d-2}\alpha^{D(d-2)}\right) \leq (D-1)+(D-1)+(Dd-2D) = Dd-2 < Dd$. This proves the first part of the theorem. The second part is just a consequence of the definition of $\overline{Q}_i$ in terms of the coefficients of $q(\alpha)$. □

Notice that Theorem 6.1 implies that if $q(\alpha) = \alpha^m + q_k\alpha^k + \sum_{i=0}^{k-1} q_i\alpha^i$ is to be an optimal M-LSDE polynomial, then $k \leq m - D$, which agrees with the findings in [SP98, BGK+03]. Notice also that (6.7), which defines the way modular reduction is performed in Step 4 of Algorithm 6.3, is independent of the value of m and thus of the field. The price of this field independence is that now we do not obtain anymore the value of $A \cdot B \bmod q(\alpha)$ but rather $A \cdot B \bmod \overline{q}(\alpha)$ thus, requiring one more reduction at the end of the whole computation. In addition, we need to multiply *once* at initialization $q(\alpha)$ by α^{Dd-m}. This, however, can be thought of as analogous to the Montgomery initialization, and thus, can be neglected when considering the total costs of complex computations which is customary practice in cryptography. In addition, notice that multiplication by α can be easily implemented in hardware via left shifts. In the remainder of this chapter, we only consider M-LSDE optimal polynomials unless we explicitly say something to the contrary.

In what follows, we re-write the steps in Algorithm 6.3 to make it suitable to a systolic implementation. The beginning of our work is (6.6), assuming that $\overline{Q}_{d-1} = 0$ or $\overline{Q}_{d-1} = 1$, which we re-write in

(6.8) as a recurrence. We also re-write the limits of the summation for convenience.

$$A^{(i)}\alpha^D \bmod \overline{q}(\alpha) = \sum_{j=1}^{d-1} A_{j-1}^{(i-1)}\alpha^{Dj} - \sum_{j=0}^{d-1}\left(A_{d-1}^{(i-1)}\overline{Q}_j\right)\alpha^{Dj} \tag{6.8}$$

Unfortunately, (6.8) is not entirely in terms of digits. In particular, we can write $A_{d-1}^{(i-1)}\overline{Q}_j$ as

$$A_{d-1}^{(i-1)}\overline{Q}_j = R_j^{(i-1)}\alpha^D + S_j'^{(i-1)} \tag{6.9}$$

where $R_j^{(i-1)}$ is a polynomial of maximum degree $D-2$ and $S_j'^{(i-1)}$ is of maximum degree $D-1$. Plugging (6.9) into (6.8), we get:

$$\begin{aligned} A^{(i)}\alpha^D \bmod \overline{q}(\alpha) &= \sum_{j=1}^{d-1} A_{j-1}^{(i-1)}\alpha^{Dj} - \sum_{j=0}^{d-1}\left(R_j^{(i-1)}\alpha^D + S_j'^{(i-1)}\right)\alpha^{Dj} \\ &= \sum_{j=1}^{d-1} A_{j-1}^{(i-1)}\alpha^{Dj} - \sum_{j=0}^{d-1} S_j'^{(i-1)}\alpha^{Dj} - \sum_{j=0}^{d-1} R_j^{(i-1)}\alpha^{D(j+1)} \end{aligned}$$

which after re-writing the index of the last summation becomes

$$A^{(i)}\alpha^D \bmod \overline{q}(\alpha) = \sum_{j=1}^{d-1} A_{j-1}^{(i-1)}\alpha^{Dj} - \sum_{j=0}^{d-1} S_j'^{(i-1)}\alpha^{Dj} - \sum_{j=1}^{d-1} R_{j-1}^{(i-1)}\alpha^{Dj} \tag{6.10}$$

where we have made use of the fact that given that $\overline{Q}_{d-1}$ is either 0 or 1 for optimal M-LSDE polynomials, $R_{d-1}^{(i-1)} = 0$ (i.e. when the index j of the last summation is equal to d, the term $R_{j-1}^{(i-1)}$ vanishes) always, and thus the last summation in (6.10) need only run to $d-1$. Similarly, we can write an expression for Step 3 of Algorithm 6.3 at iteration i as:

$$\overline{C}^{(i)} \;=\; B_i A^{(i-1)} + \overline{C}^{(i-1)} = \sum_{j=0}^{d-1}\left(B_i A_j^{(i-1)}\right)\alpha^{Dj} + \sum_{j=0}^{d} \overline{C}_j^{(i-1)}\alpha^{Dj} \tag{6.11}$$

Notice that $B_i A_j^{(i-1)}$ is of the same form as (6.9), thus we can write $B_i A_j^{(i-1)} = R'_{j,i,(i-1)}\alpha^D + S''_{j,i,(i-1)}$ which when plugged back into (6.11), gives us

$$\begin{aligned} \overline{C}^{(i)} &= \sum_{j=0}^{d-1} \left(R'_{j,i,(i-1)}\alpha^D + S''_{j,i,(i-1)} \right) \alpha^{Dj} + \sum_{j=0}^{d} \overline{C}_j^{(i-1)} \alpha^{Dj} \\ &= \sum_{j=1}^{d} R'_{j-1,i,(i-1)}\alpha^{Dj} + \sum_{j=0}^{d-1} S''_{j,i,(i-1)}\alpha^{Dj} + \sum_{j=0}^{d} \overline{C}_j^{(i-1)} \alpha^{Dj} \qquad (6.12) \end{aligned}$$

In a similar manner, we can derive an expression for the last reduction ($\overline{C} \bmod \overline{q}(\alpha)$) of Algorithm 6.3. In particular,

$$\overline{C} \bmod \overline{q}(\alpha) = \sum_{j=0}^{d-1} \overline{C}_j \alpha^{Dj} - \sum_{j=1}^{d-1} R''_{j-1,d}\alpha^{Dj} - \sum_{j=0}^{d-1} S'''_{j,d}\alpha^{Dj} \qquad (6.13)$$

where $\overline{C}_d \overline{Q}_j = R''_{j,d}\alpha^D + S'''_{j,d}$ and because of the use of optimal M-LSDE polynomials $R''_{d-1,d} = 0$. Using (6.10), (6.12), and (6.13) we readily obtain Algorithm 6.4. Notice that the **for**-loop starting on

Algorithm 6.4 Low Level Modified LSDE Multiplier

Input: $A = \sum_{i=0}^{d-1} A_i \alpha^{Di}$ with $A_i = \sum_{j=0}^{D-1} a_{Di+j}\alpha^j$, $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$ with $B_i = \sum_{j=0}^{D-1} b_{Di+j}\alpha^j$, $\overline{q}(\alpha) = \alpha^{Dd-m} q(\alpha) = \alpha^{Dd} + \sum_{i=0}^{d-1} \overline{Q}_i \alpha^{Di}$ and $\overline{Q}_{d-1} = 0$ or $\overline{Q}_{d-1} = 1$
$a_i, b_i \in GF(p)$, and $d = \lceil \frac{m}{D} \rceil$

Output: $\overline{C} \equiv A \cdot B \bmod \overline{q}(\alpha) = \sum_{i=0}^{d} \overline{C}_i \alpha^{Di}$ with $\overline{C}_i = \sum_{j=0}^{D-1} c_{Di+j}\alpha^j$, $c_i \in GF(p)$, and $d = \lceil \frac{m}{D} \rceil$

1: $\overline{C} \leftarrow 0$
2: **for** $i = 0$ to $d-1$ **do**
3:   **for** $j = d$ to 0 **do**
4:     $R'_{j-1}\alpha^D + S''_{j-1} \leftarrow B_i A_{j-1}$ $\{ A_{-1} = 0 \}$
5:     $\overline{C}_j \leftarrow R'_{j-1} + S''_j + \overline{C}_j$ $\{ S''_d = 0 \}$
6:   **end for**
7:   **for** $j = d$ to 0 **do**
8:     $R_{j-1}\alpha^D + S'_{j-1} \leftarrow A_{d-1}\overline{Q}_{j-1}$ $\{ \overline{Q}_{-1} = 0 \}$
9:     $A_j \leftarrow A_{j-1} - S'_j - R_{j-1}$ $\{ A_{-1} = 0, S'_d = A_{d-1} \}$
10:   **end for**
11: **end for**
12: Return ($\overline{C} \bmod \overline{q}(\alpha)$)   {Reduction performed according to (6.13)}

line 7 can start at $d-1$ for optimal M-LSDE polynomials since then $A_d = 0$ always.

6.4 Architecture Description

The previous section introduced scalable LSE/LSDE multiplier architectures for fields $GF(p^m)$. These architectures are adaptations of the scalable architecture introduced in [TcKK99] to multiplication in fields $GF(p^m)$ and for multiplication algorithms that employ most significant digit reduction. Note that the architecture introduced in [TcKK99] was developed for Montgomery multiplication, which is a form of least significant digit reduction.

Addition in fields $GF(p^m)$ are carry free operations, therefore digit addition is also carry free. Two digits result from the multiplication of two digits; for example, $A_j \cdot B_i = R\alpha^D + S'$. One can consider R as a carry. Due to carry free addition, carries are consumed in the next digit position without generating further carries. This operation can be transformed so that carries flow towards the least significant digit positions using the following primitive: $A_j \cdot B_i/\alpha^D = (R\alpha^D + S')/\alpha^D$. These principles are used by the architectures presented here to perform scalar multiplications in a most significant to least significant digit order. Scalar multiplications refer to the multiplication of a $GF(p^m)$ field element by a digit. Figure 6.3 shows a dependency graph for LSDE multiplication for $d = 4$. In this graph the flow of execution progresses horizontally in the i dimension from left to right and vertically in the j dimension from top to bottom. The cut set lines, shown with dotted lines, show the timing boundaries used here to develop a systolic architecture. In general, data propagation in the i dimension need to be registered once, while data in the j direction needs to be registered twice. Data traveling diagonally need to be registered once. Dots at the top of the graphs represent the delays required to synchronize inputs with array execution. The dependency graph shows two types of cells. In the following discussion, the term type 1 refers to the cells with rounded corners and the term type 2 refers to the cells with right angle corners. Type 1 cells are used to compute Steps 2 through 11 of Algorithm 6.4 and type 2 cells are used to compute Step 12 of the same algorithm.

By folding the dependency graph along the i dimension one obtains a digit-serial multiplier where each processing unit processes a row of the dependency graph. If one then folds again along the j dimension, one obtains a scalable architecture, where processing unit x performs the functions in rows x mod e, where e represents the number of processing units in the circuit. The folding just described

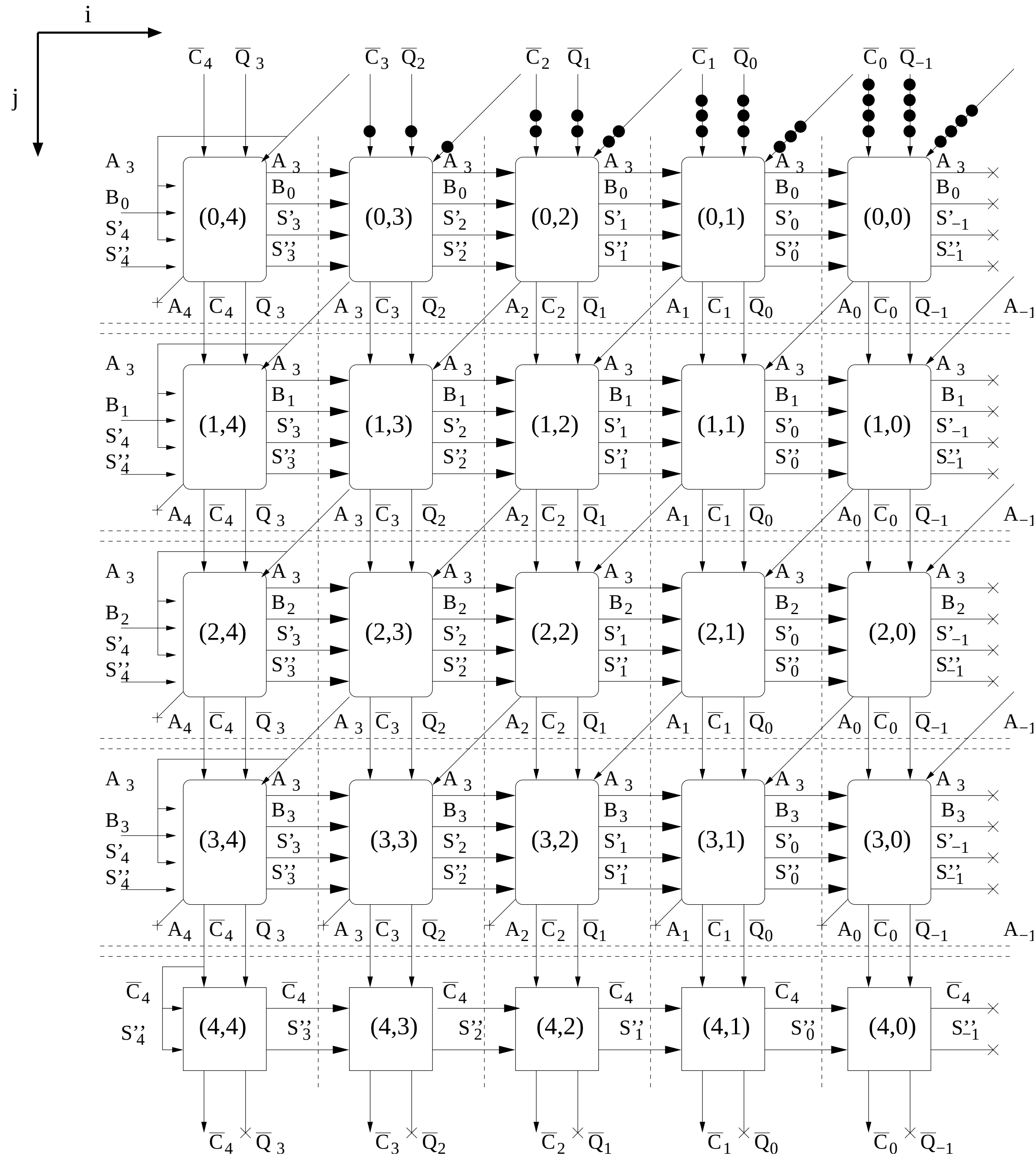


Figure 6.3. Dependency graph for LSDE multiplication ($d = 4$)

requires that one of the processing units be able to perform the functions of both type 1 and type 2 cells. LSE multipliers do not need to compute the reduction in the last row, and thus require processing units to perform the functions of type 1 cells only. Figure 6.4 shows a block diagram of an LSE/LSDE multiplier that uses two processing units ($e = 2$). Processing unit 0 performs the function of type 1 cells. Processing unit 1 performs the functions of type 2 cells. For this work we developed a single processing unit that can perform the functions of cells of type 1 and 2.

For the LSDE multipliers, the data lines, represented with solid lines, transport D elements (width of data paths is D times the width of an element). For LSE multipliers the data lines carry one element. Dotted lines represent control signals. Note that control is sent from one cell to another in a way that allows data synchronization in the processing units. The figure shows the set of inputs to processing unit 0 with the subscript 0. The same scheme is used for processing unit 1. The scheduling of operands for the LSE/LSDE algorithms require that operands $\overline{Q_i}$ be fed one digit or element at time in a cyclic manner. The rotator (Rot) in the block diagram performs this function. The digits from $\overline{Q_i}$ are bypassed by processing unit 0. Shift registers load the operands B and A into the multiplier. The multiplicand A is loaded into the multiplier through a multiplexer during the computation of the top row in the dependency graph. Thereafter, processing unit 0 gets operands from processing unit 1.

Even though it is not shown, the C_0 input can be enhanced with a circuit similar to one used to multiplex A. This circuit would allow the multiplier to perform multiply-and-accumulate operations. The FIFOs are the main components that support scalability. These FIFOs allow processing unit 1 to buffer data destined for processing unit 0. For example, if processing unit 0 is working on row 0 and processing unit 1 is working on row 1 in the dependency graph, the partial results from processing unit 1 corresponding to row 1 are stored in the FIFOs. When processing unit 0 starts working on row 2, it starts consuming the partial results corresponding to row 1 of the dependency graph. Note that partial results from processing unit 0 to processing unit 1 require one or two register delays.

6.4.1 Complexity and Performance

Table 6.1 summarizes the most significant characteristics of LSE/LSDE multipliers. The results in the table assume that d is a multiple of e for LSE multipliers and that $d + 1$ is a multiple of e for LSDE

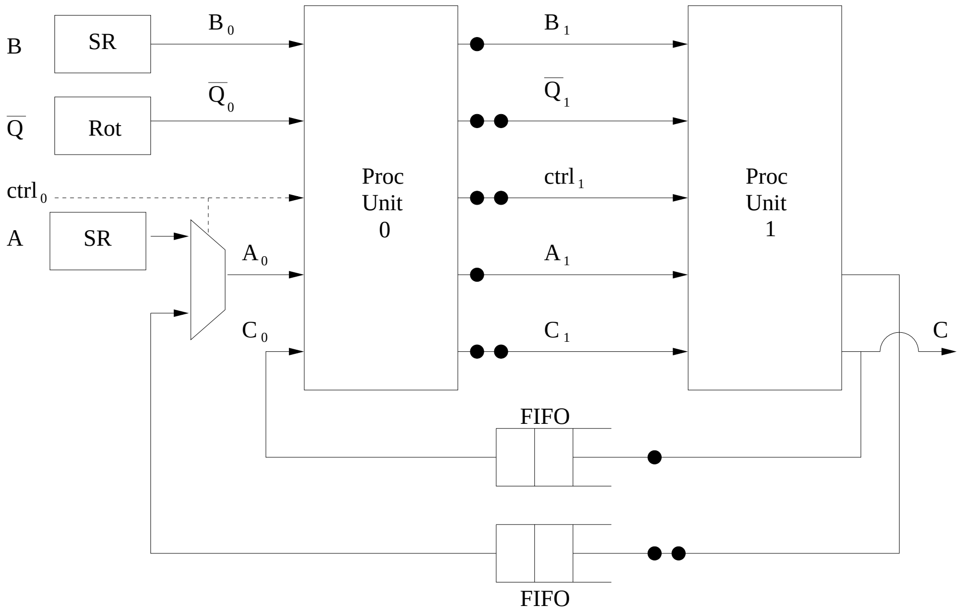


Figure 6.4. LSDE scalable multiplier ($e = 2$)

multipliers. This arrangement aligns the data so that results can be gathered from the the last processing unit in the pipeline. Note that the LSDE algorithm involves extra iterations in its processing loops.

The table identifies two cases for d. When $d = e$ the multiplication does not involve the feedback path. This is analogous to having a digit-serial systolic multiplier. When $d \geq 2e$ the feedback path is used. The timing metrics shown in the table can be deduced from the LSDE dependency graph.

In Table 6.1, M, A, R, and X represent, respectively, $GF(p)$ digit/element multiplier, adder, register, and multiplexer. T_M and T_A represent, respectively, the delay of a $GF(p)$ digit/element multiplier and adder.

Table 6.1. Characteristics of LSE/LSDE multipliers

Parameter	Condition	LSE	LSDE
Throughput (# clocks/1 mult.)	$d \geq e$	d^2/e	$(d+1)^2/e$
Latency (# clocks)	$d \geq 2e$	$(d/e)d$	$((d+1)/e)(d+1)+1$
	$d = e$	$2e$	$2e+1$
FIFO storage (# digits)	$d \geq 2e$	$d-2e$	$(d+1)-2e$
	$d = e$	0	0
Critical path delay	$d \geq e$	$T_M + T_A$	$T_M + T_A$
Cell complexity	$d \geq e$	$2M+4A+11R+5X$	$2M+4A+11R+5X$

6.5 Prototypes Description

Since the multiplier is scalable, we report synthesis[4] results for only one processing unit. The complexity of a scalar multiplier is a function of the number of processing units it contains. The prototyped processing units were synthesized with Synopsys Design Compiler and they were mapped on a 0.18 μm technology library from ST Microelectronics. For comparisons with other technology libraries, a gate density of 85 $Kgate/mm^2$ can be assumed for the ST Microelectronics library. The frequency of operation reported represents worst environment conditions ($80\,^{\circ}\mathrm{C}$).

Table 6.2 presents results for ground fields $GF(2)$ and $GF(3)$ and for digit sizes $D = 4, 8, 16$. One and two bits respectively were used to represent elements of the fields $GF(2)$ and $GF(3)$. The results

Table 6.2. Area and Frequency of the basic cell for an LSDE multiplier

	$GF(3)$		$GF(2)$	
Digit size	Frequency (MHz)	Area (μm^2)	Frequency (MHz)	Area (μm^2)
4	333	23900	454	6200
8	256	61600	357	14900
16	181	181000	344	43400

show that increases in digit size result in increases in the critical path delay and, thus, a reduction of the maximum frequency. In addition, the $GF(2^m)$ processing units exhibit superior time-area products, even when considering inputs to the processing units of the same width (for example, compare the results for $GF(2)$ with $D = 16$ and the results for $GF(3)$ with $D = 8$).

6.6 Notes and Further References

1. In our context long-precision refers to polynomials which require more than one word to be represented.

2. In Step 7 of Algorithm 6.2, the quantity $a_{j-1} - q_j a_{m-1}$ must be computed which, in principle, would require a $GF(p)$ subtracter. However, we can re-write the last operation as $a_{j-1} + \tilde{q}_j a_{m-1}$, where $\tilde{q}_j = -q_j = p - q_j$. For fixed p this is just a re-write of the coefficients of the irreducible polynomial

$q(x)$. Thus, an adder can be used instead of a subtracter.

3. This in general does not seem to be a problem. For example, 80% of irreducible polynomials over $GF(2)$ in [P1300] satisfy this requirement for practical values of D and similar results have been found in [BGK+03] for polynomials over $GF(3)$.

4. The synthesis was joint work with General Dynamics Communication Systems, USA and the Politecnico di Milano, Italy. Part of the design was coded by Gerardo Orlando at General Dynamics Communication Systems. The other part and the synthesis were performed by Guido Bertoni in Italy. Special thanks to both of them.

CHAPTER 7

An Inversion Algorithm for Fields $GF(q^m)$

This chapter is concerned with a generalization of Itoh and Tsujii's algorithm for inversion in extension fields $GF(q^m)$. Unlike the original algorithm, the method introduced here uses a standard (or polynomial) basis representation. The inversion method is generalized for standard basis representation and relevant complexity expressions are established, consisting of the number of extension field multiplications and exponentiations. In addition, for three important classes of fields we show that the Frobenius map can be explored to perform the exponentiations required for the inversion algorithm efficiently. Thus, Itoh and Tsujii's inversion method shows almost the same practical complexity for standard basis as for normal basis representation for the field classes considered. Parts of this chapter appear in [GP02] and [Gua03].

7.1 Preliminaries

The two most popular methods for large finite field inversion are based on the extended Euclidean algorithm or one of its derivatives (e.g., the almost inverse algorithm [SOOS95]), the extended binary $\gcd$, also known as Stein's algorithm [Ste67], or on Fermat's little theorem. The Itoh and Tsujii algorithm (ITA) [IT88] is an exponentiation-based inversion algorithm which reduces the complexity of computing the inverse of a non-zero element in $GF(2^n)$, when using a normal basis representation, from $n-2$ multiplications in $GF(2^n)$ and $n-1$ cyclic shifts using the binary method for exponentiation to at most

$2\lfloor \log_2(n-1) \rfloor$ multiplications in $GF(2^n)$ and $n-1$ cyclic shifts.

Itoh and Tsujii proposed in [IT88] three algorithms. The first two algorithms describe addition chains for exponentiation-based inversion in fields $GF(2^n)$ while the third one describes a method based on subfield inversion. The first algorithm is only applicable to values of n such that $n = 2^r + 1$, for some positive r, and it is based on the observation that the exponent $2^n - 2$ can be re-written as $(2^{n-1} - 1) \cdot 2$. Thus if $n = 2^r + 1$, we can compute $A^{-1} \equiv (A^{2^{2^r}-1})^2$. Furthermore, we can re-write $2^{2^r} - 1$ as

$$2^{2^r} - 1 = \left(2^{2^{r-1}} - 1\right) 2^{2^{r-1}} + \left(2^{2^{r-1}} - 1\right) \tag{7.1}$$

Equation (7.1) and the previous discussion lead to Algorithm 7.1. Notice that Algorithm 7.1 performs

Algorithm 7.1 Multiplicative inverse computation in $GF(2^n)$ with $n = 2^r + 1$ [IT88, Theorem 1]

Input: $A \in GF(2^n)$, $A \neq 0$, $n = 2^r + 1$
Output: $C = A^{-1}$
  $C \leftarrow A$
  for $i = 0$ to $r - 1$ **do**
    $D \leftarrow C^{2^{2^i}}$ {NOTE: 2^i cyclic shifts}
    $C \leftarrow C \cdot D$
  end for
  $C \leftarrow C^2$
  Return (C)

$r = \log_2(n-1)$ iterations. In every iteration, one multiplication and i cyclic shifts, for $0 \le i < r$, are performed which leads to an overall complexity of $\log_2(n-1)$ multiplications and $n-1$ cyclic shifts.

Example 7.1. Let $A \in GF(2^{17})$, $A \neq 0$. Then according to Algorithm 7.1 we can compute A^{-1} with the following addition chain:

$$\begin{aligned}
A^2 \cdot A &= A^3 \\
\left(A^3\right)^{2^{2^1}} \cdot A^3 &= A^{15} \\
\left(A^{15}\right)^{2^{2^2}} \cdot A^{15} &= A^{255} \\
\left(A^{255}\right)^{2^{2^3}} \cdot A^{255} &= A^{65535} \\
\left(A^{65535}\right)^2 &= A^{131070}
\end{aligned}$$

A quick calculation verifies that $2^{17} - 2 = 131070$. Notice that in accordance with Algorithm 7.1 we have performed four multiplications in $GF(2^{17})$ and, if using a normal basis, we would also require $2^4 = 16$ cyclic shifts.

Algorithm 7.1 can be generalized to any value of n [IT88]. First, we write $n-1$ as

$$n - 1 = \sum_{i=1}^{t} 2^{k_i}, \quad \text{where } k_1 > k_2 > \cdots > k_t \tag{7.2}$$

Using the fact that $A^{-1} \equiv (A^{2^{n-1}-1})^2$ and (7.2), one can write the inverse of A as:

$$(A^{2^{n-1}-1})^2 = \left[(A^{2^{2^{k_t}}-1}) \left(\left(A^{2^{2^{k_{t-1}}}-1} \right) \cdots \left[(A^{2^{2^{k_2}}-1})(A^{2^{2^{k_1}}-1})^{2^{2^{k_2}}} \right]^{2^{2^{k_3}}} \cdots \right)^{2^{2^{k_t}}} \right]^2 \tag{7.3}$$

An important feature of (7.3) is that in computing $A^{2^{2^{k_1}}-1}$ all other quantities of the form $A^{2^{2^{k_i}}-1}$ for $k_i < k_1$ have been computed. Thus, the overall number of multiplications required in (7.3) is equal to the number of multiplications required to compute $A^{2^{2^{k_1}}-1}$ which according to (7.1) is just k_1 plus the multiplications required to multiply out all the terms of the form $A^{2^{2^{k_i}}-1}$ in (7.1) which is equal to $t - 1 = HW(m-1) - 1$. Similarly, the number of cyclic shifts to compute $A^{2^{2^{k_1}}-1}$ are just[1] $2^{k_1} - 1$ and therefore the overall number of cyclic shifts is found as $(2^{k_1} - 1) + 2^{k_2} + \cdots 2^{k_t} + 1 = n - 1$. These results are summarized in Theorem 7.1

Theorem 7.1. *[IT88, Theorem 2] Let $A \in GF(2^n)$, $A \neq 0$. Then, there exists an algorithm which can compute A^{-1} with the following complexity*

$$\begin{aligned} \#\text{MUL} &= \lfloor \log_2(n-1) \rfloor + HW(n-1) - 1 \\ \#\text{CSH} &= n - 1 \end{aligned}$$

where $HW(\cdot)$ denotes the hamming weight of the operand, MUL refers to multiplications in $GF(2^n)$, and CSH refers to cyclic shifts over $GF(2)$ when using a normal basis.

[1] Notice that in Algorithm 7.1 we are computing A^{-1} which requires one extra squaring at the end of the loop.

Example 7.2. Let $A \in GF(2^{23})$, $A \neq 0$. Then according to (7.2) we can write $n-1 = 22 = 2^4+2^2+2$ where $k_1 = 4$, $k_2 = 2$, and $k_3 = 1$. It follows that we can compute $A^{-1} \equiv A^{2^{23}-2}$ with the following addition chain:

$$
\begin{aligned}
A^2 \cdot A &= A^{2^2-1} \\
\left(A^3\right)^{2^2} \cdot A^3 &= A^{2^4-1} \\
\left(A^{15}\right)^{2^4} \cdot A^{15} &= A^{2^8-1} \\
\left(A^{255}\right)^{2^8} \cdot A^{255} &= A^{2^{16}-1} \\
\left(A^{2^2-1} \cdot \left(A^{2^4-1} \cdot \left(A^{2^{16}-1}\right)^{2^4}\right)^{2^2}\right)^2 &= A^{2^{23}-2}
\end{aligned}
$$

The above addition chain requires 6 multiplications and 22 cyclic shifts which agrees with the complexity of Theorem 7.1.

In [IT88], the authors also notice that the previous ideas can be applied to extension fields $GF(q^m)$, $q = 2^n$. Although this algorithm does not perform a complete inversion, it reduces extension field inversion to inversion in $GF(q)$. It is assumed that subfield inversion can be done relatively easily, e.g., through table look-up or with one of the general methods mentioned earlier. The ITA is applicable to finite fields $GF(2^m)$ given in a normal basis representation. In particular, the original reference deals with composite fields $GF((2^n)^m)$. In this chapter, we apply the idea of Itoh and Tsujii to fields $GF(q^m)$ given in standard basis (or polynomial or canonical) basis representation. Although the exponentiations required in the algorithm make it rather inefficient for general fields in a standard basis representation, it can be shown that for certain classes of finite fields, the exponentiations can be computed with a very low complexity. The field classes for which efficient inversion algorithms are possible include composite fields $GF(q^m)$, $q = 2^n$, with a binary extension field polynomial; fields $GF(q^m)$ where $q = p^n$, p is an odd prime, and the field polynomial is a binomial; and fields $GF(q^m)$ where q is a prime power and the field polynomial is an equally spaced polynomial with binary coefficients.

The remainder of this chapter is organized as follows. Section 7.2 revisits the ITA for inversion in composite fields $GF((2^n)^m)$ and generalizes it to fields of any characteristic. Unlike the original algorithm, we consider the ITA in standard (or polynomial) basis representation in Section 7.3. In addition, the number of extension field multiplications and exponentiations required to perform an inverse operation is established. Finally, in Section 7.4 we show that for three important classes of fields, the Frobenius map can be explored to perform the exponentiations required for the inversion algorithm efficiently.

7.2 The Itoh-Tsujii Algorithm over $GF(q^m)$, $q = p^n$

In [IT88], the authors also propose a method for reducing inversion in $GF(q^m)$ to inversion in $GF(q)$, where $q = 2^n$. As in the case of the previous ones, this algorithm was studied in the context of a normal basis representation of the fields $GF(2^n)$ and $GF((2^n)^m)$. In the following we will review this algorithm with a new notation. Our presentation will be slightly more general as we do not require a subfield of the form $GF(2^n)$ but allow general subfields $GF(q)$, $q = p^n$, and p any prime.

Theorem 7.2. *[IT88] Let $A \in GF(q^m)$, $A \neq 0$ and $r = (q^m - 1)/(q - 1)$. Then, the multiplicative inverse of an element A can be computed as*

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

Computing the inverse through Theorem 7.2 requires four steps:

Step 1. Exponentiation in $GF(q^m)$, yielding A^{r-1}.

Step 2. Multiplication of A and A^{r-1}, yielding $A^r \in GF(q)$.

Step 3. Inversion in $GF(q)$, yielding $(A^r)^{-1}$.

Step 4. Multiplication of $(A^r)^{-1} A^{r-1}$.

Steps 2 and 4 are trivial since both A^r, in Step 2, and $(A^r)^{-1}$, in Step 4, are in the subfield $GF(q)$. Both operations can, in most cases, be done with a complexity that is well below that of one single extension field multiplication. The complexity of Step 3, subfield inversion, depends heavily on the type and order of the subfield $GF(q)$ and will not be discussed here. However, in many practical scenarios, such as in the case of cryptographic applications, the subfield can be small enough to perform inversion very efficiently, for example, through table look-up [GP97, DBV+96], or by using the Euclidean algorithm which can be applied with relatively low processing times for small subfield orders [BP01a]. What remains is Step 1, exponentiation to the $(r-1)$th power in the extension field $GF(q^m)$.

First, we notice that the exponent can be expressed in q-adic representation as

$$r - 1 = q^{m-1} + \cdots + q^2 + q = (1 \cdots 110)_q$$

This exponentiation can be computed through repeated raising of intermediate results to the q-th power and multiplications. The number of multiplications in $GF(q^m)$ can be minimized by using the addition chain proposed by Itoh and Tsujii [IT88]. Thus:

$$\#\text{MUL} = \lfloor \log_2(m-1) \rfloor + HW(m-1) - 1 \tag{7.4}$$

The number of exponentiations to the q-th power is given by

$$\#q\text{-EXP} = m - 1$$

The original paper assumes a normal basis representation of the field elements of $GF(q^m)$, $q = 2^n$, in which the exponentiations to the q-th power are simply cyclic shifts of the m coefficients that represent an individual field element. In standard basis, however, these exponentiations are, in general, considerably more expensive. Standard basis q-th power exponentiation will be considered in detail in Section 7.3 below. Notice that the algorithm performs alternating exponentiations and multiplications with previous results. For the treatment in Section 7.3 it is important to observe that in general several exponentiations to the qth power are performed between two multiplications.

7.3 Itoh-Tsujii Inversion in Standard Basis

In the following, we will consider the field $GF(q^m)$ generated by an irreducible polynomial $P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i$ over $GF(q)$ of degree m. Let α be a root of $P(x)$, then we represent $A \in GF(q^m)$ as $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i, \;\; a_i \in GF(q)$.

We will now establish the complexity of raising A to the q^e-th power, where e is a positive integer. This is the eth iterate of the Frobenius automorphism:

$$A^{q^e} = \left(\sum_{i=0}^{m-1} a_i \alpha^i \right)^{q^e} = \sum_{i=0}^{m-1} a_i \alpha^{i\, q^e}$$

This exponentiation is a $GF(q)$-linear mapping for all integers $e > 0$. We can explicitly construct the matrix which describes the mapping by considering the standard basis representations of $\alpha^{i\, q^e}$, $i = 1, 2, \ldots, m-1$:

$$\alpha^{i\, q^e} \equiv s_{0,i}^{(e)} + s_{1,i}^{(e)} \alpha + \cdots + s_{m-1,i}^{(e)} \alpha^{m-1}, \;\; i = 1, 2, \ldots, m-1 \tag{7.5}$$

together with the identity $P(\alpha) = 0$. Notice that the superscripts "(e)" are mere indexes. The matrix follows now as

$$A^{q^e} = \begin{pmatrix} 1 & s_{0,1}^{(e)} & s_{0,2}^{(e)} & \cdots & s_{0,m-1}^{(e)} \\ 0 & s_{1,1}^{(e)} & s_{1,2}^{(e)} & \cdots & s_{1,m-1}^{(e)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & s_{m-1,1}^{(e)} & s_{m-1,2}^{(e)} & \cdots & s_{m-1,m-1}^{(e)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix} \tag{7.6}$$

In general, the eth iterate of the Frobenius map has a complexity of $m(m-1)$ multiplications and $m(m-2)+1 = (m-1)^2$ additions in $GF(q)$. We note that this complexity is roughly the same as one $GF(q^m)$ multiplication, which requires m^2 subfield multiplications if we do not assume fast convolution techniques (e.g., Karatsuba's algorithm [KO63]).

An adaptation of the ITA to standard basis is straightforward. In fact, the description above is independent of the basis representation. We observe, however, that in standard basis the eth iterate of the Frobenius map, where $e > 1$, is as costly as a single exponentiation to the qth power. Thus, we change the algorithm slightly by performing as many subsequent exponentiations to the qth power in one step between multiplications. This yields the same multiplication complexity as given in (7.4), but we perform now eth iterates of the Frobenius map with the following complexity:

Theorem 7.3. *Let $A \in GF(q^m)$. One can compute A^{r-1} where $r-1 = q + q^2 + \cdots + q^{(m-1)}$ with no more than*

$$\begin{aligned} \#\mathit{MUL} &= \lfloor \log_2(m-1) \rfloor + HW(m-1) - 1 \\ \#q^e\text{-}\mathit{EXP} &= \lfloor \log_2(m-1) \rfloor + HW(m-1) \end{aligned}$$

operations, where $\#$MUL and $\#q^e$-EXP refer to multiplications and exponentiations to the q^eth power in $GF(q^m)$, respectively.

Proof. First, consider the computation of A^{s_k} where $s_k = \sum_{i=1}^{2^k} q^i = q + q^2 + \cdots + q^{2^k}$. Notice that $A^{s_k} = (A^{s_{k-1}})^{q^{2^{k-1}}} A^{s_{k-1}}$. If we denote by $M(k)$ the number of multiplications and by $E(k)$ the number of exponentiations to the q^eth power required to compute A^{s_k}, then it is easy to see that $M(k) = M(k-1) + 1$ and $E(k) = E(k-1) + 1$. Notice also that $A^{s_0} = A^q$, thus $M(k=0) = 0$ and $E(k=0) = 1$. It follows that $M(k) = k$ and $E(k) = k+1$. Furthermore, in computing A^{s_k}, we have also computed A^{s_i} for $s_i < s_k$. We now apply a similar procedure as in the proof of Theorem 2 in [IT88]. Let $m - 1 = \sum_{u=1}^{t} 2^{k_u}$ with $k_1 > k_2 > \cdots > k_t$. Then, one can re-write A^{r-1} as follows:

$$A^{r-1} = A^{q^{m-1}+\cdots+q^2+q} = (A^{s_{k_t}}) \left(\cdots (A^{s_{k_3}}) \left[(A^{s_{k_2}})(A^{s_{k_1}})^{q^{2^{k_2}}} \right]^{q^{2^{k_3}}} \cdots \right)^{q^{2^{k_t}}}$$

Since $k_1 > k_i$ for $i = 2, \cdots, t$ then if we compute $A^{s_{k_1}}$ as above, all the $A^{s_{k_i}}$ for $i = 2, \cdots, t$ will also be computed. From our previous results we see that $M(k_1) = k_1 = \lfloor \log_2(m-1) \rfloor$ and $E(k_1) = k_1 + 1 = \lfloor \log_2(m-1) \rfloor + 1$. Also notice that we go through $t - 1 = HW(m-1) - 1$ multiplications and eth iterates of the Frobenius after computing $A^{s_{k_1}}$. Adding up the partial complexities, one obtains the result in Theorem 7.3. $\square$

We would like to stress that Theorem 7.3 is just an upper bound on the complexity of this exponentiation. Thus, it is possible to find addition chains which yield better complexity as shown in [CSL00]. In addition, we see from Theorem 7.3 that Step 1 of the ITA requires about as many exponentiations to the q^eth power as multiplications in $GF(q^m)$ if a standard basis representation is being used. In the discussion earlier in this section it was established that eth iterates of the Frobenius map are roughly as costly as multiplications. Hence, if it is possible to make exponentiations to the q^eth power more efficient, considerable speed-ups of the algorithm can be expected. In the remainder of the paper we will introduce three classes of finite fields for which the complexity of the eth iterates of the Frobenius map is in fact substantially lower than that of a general multiplication in $GF(q^m)$.

7.4 Field Types with Low Complexity Inversion

This section introduces three types of finite fields for which eth iterates of the Frobenius map are substantially less costly than general field multiplications. All three field families have been proposed for use in public-key cryptosystems, mainly in the context of elliptic curve cryptosystems.

7.4.1 Fields $GF((2^n)^m)$ with Binary Field Polynomials

Fields of characteristic two with two field extensions $GF(q^m)$, $q = 2^n$, sometimes referred to as *composite fields*, had been proposed repeatedly for applications in elliptic curve cryptosystems [HMV92, DBV$^+$96, GP97]. We notice that Gaudry et al. [GHS02b] have shown an attack on these types of curves. However, it is not entirely certain how plausible the attack is and if it is practical at all (see for example [CQS01, MQ01, MMT02]). Even if these fields were not usable to construct applications based on elliptic curve cryptosystems, composite fields have also proved useful in the implementation of error correcting codes [Paa96] and in the implementation of power and space efficient architectures for the Advanced Encryption Standard (AES) [MS02].

Let the field polynomial $P(x)$ be irreducible over $GF(2)$ and of degree m. Then, according to Theorem 2.4, $P(x)$ will also be irreducible over $GF(2^n)$ if and only if $\gcd(n, m) = 1$. Notice that since $P(x)$ is binary, all the powers α^{iq^e} in (7.5) can also be represented as binary polynomials. Hence, the matrix coefficients $s_{i,j}$ are elements of $GF(2)$ and no general multiplications are required in the matrix multiplication shown in (7.6). Assuming on average an equal number of ones and zeros in the matrix, an eth iterate of the Frobenius map can be computed with an average complexity of

$$\left(\frac{m-1}{2}-1\right)m+1=\frac{(m-1)(m-2)}{2}\leq\frac{m^2}{2}$$

additions in $GF(2^n)$ and no $GF(2^n)$ multiplications. Since $GF((2^n)^m)$ multiplications require (in a straight forward realization) m^2 subfield multiplications and $(m-1)^2$ subfield additions, the dominant complexity for computing A^{r-1} in $GF((2^n)^m)$ is now determined by the number of extension field multiplications as given in (7.4).

Example 7.3. As an example we consider the special case where $n = 16$ and $m = 11$ which is of interest for cryptographic systems that are based on the discrete logarithm problem for elliptic curves [DBV+96, GP97]. We chose as field polynomial the trinomial $P(x) = x^{11} + x^2 + 1$. We can now apply Theorem 7.3 to compute $A^{r-1} = A^{2^{16}+2^{2\cdot 16}+\cdots+2^{10\cdot 16}}$. Note that $m-1 = 10 = 2^3 + 2 = 2^{k_1} + 2^{k_2}$ and that $q = 2^n$. Then

$$A^{r-1}=\left(A^{s_{k_2}}\right)\left(A^{s_{k_1}}\right)^{q^{2^{k_2}}}=\left(A^{2^n+2^{2n}}\right)\left(A^{2^n+2^{2n}+\cdots+2^{8n}}\right)^{2^{2n}}=A^{2^n+2^{2n}+\cdots+2^{10n}}$$

$A^{s_{k_1}}$ can be computed using the following addition chain:

$$\begin{array}{rcl}
 & A & \\
 & A^{2^n} & \\
(A^{2^n})^{2^n} & = & A^{2^{2n}} \\
A^{2^n}A^{2^{2n}} & = & A^{2^n+2^{2n}} \\
(A^{2^n+2^{2n}})^{2^{2n}} & = & A^{2^{3n}+2^{4n}} \\
A^{2^n+2^{2n}}A^{2^{3n}+2^{4n}} & = & A^{2^n+2^{2n}+2^{3n}+2^{4n}} \\
(A^{2^n+2^{2n}+2^{3n}+2^{4n}})^{2^{4n}} & = & A^{2^{5n}+2^{6n}+2^{7n}+2^{8n}} \\
A^{2^n+2^{2n}+2^{3n}+2^{4n}}A^{2^{5n}+2^{6n}+2^{7n}+2^{8n}} & = & A^{s_{k_1}}
\end{array}$$

Notice that in computing $A^{s_{k_1}}$, we also computed $A^{s_{k_2}} = A^{2^n+2^{2n}}$ and that in the overall process we performed $\lfloor \log_2(10) \rfloor + HW(10) - 1 = 3 + 2 - 1 = 4$ multiplications and 5 exponentiations to a power 2^{ne} as predicted by Theorem 7.3. Furthermore, each exponentiation to a power 2^{ne} will only require $(m-1)(m-2)/2 = 10 \times 9/2 = 45$ additions in $GF(2^n)$ on average.

7.4.2 Fields $GF(q^m)$ with Binomials as Field Polynomials

For extension fields with odd prime characteristic it is often possible to choose irreducible binomials $P(x) = x^m - \omega$, $\omega \in GF(q)$. A specific sub-class of these fields where q is a prime of the form $q = p = 2^n - c$, c "small", has recently been proposed for cryptographic applications in [BP98, LKL98, KMKH99, BP01a, Kob00] (Also see [BP01a] for tabulated tables with values for n, c, m, and ω). We will show that for the general case of fields $GF(q^m)$ with binomials as field polynomials, the eth iterates of the Frobenius map in the ITA are computationally inexpensive. [LN97, Theorem 3.75] describes the conditions necessary for irreducible binomials to exist. The computational savings are due to the following theorem:

Theorem 7.4. *Let $P(x)$ be an irreducible polynomial of the form $P(x) = x^m - \omega$ over $GF(q)$, e an integer, $P(\alpha) = 0$, and it is understood that $q = p^n$, $p \geq 3$. Then:*

$$\alpha^e \equiv \omega^t \alpha^s$$

where $s \equiv e \bmod m$ and $t = \frac{e-s}{m}$

Proof. First, notice that since $P(\alpha) = 0$, then $\alpha^m \equiv \omega$. Now $\alpha^e = \alpha^{tm+s}$, where t and s are as defined above. Then, $\alpha^e = \alpha^{tm}\alpha^s \equiv \omega^t \alpha^s$ $\square$

It follows immediately from Theorem 7.4 that (7.5) has only one non-zero coefficient $s_{i,j}^{(e)}$ and thus, the exponentiation matrix in (7.6) has also only one non-zero entry per column. Moreover, the theorem also provides an efficient method for computing these entries. Again, the dominant complexity of Step 1 of the ITA algorithms is determined by the number of extension field multiplications as given in (7.4).

Example 7.4. In [BP01a], we find that $p = 2^7 - 1$ is prime and that $P(x) = x^{21} - 3$ is irreducible over $GF(p)$. Then, $P(x)$ is also irreducible over $GF(p^k)$ for $\gcd(k, m) = 1$. Let $k = 2$ and $q = p^2$, then using the construction shown in the proof of Theorem 7.3, it is easy to see that, for any $A \in GF(q^{21})$, computing $A^{r-1} = A^{q+q^2+\cdots+q^{20}}$ can be performed as:

$$\begin{aligned}
(A^q)^q \cdot A^q &= A^{q^2+q} \\
\left(A^{q^2+q}\right)^{q^2} \cdot A^{q^2+q} &= A^{q^4+q^3+q^2+q} \\
\left(A^{q^4+q^3+q^2+q}\right)^{q^4} \cdot A^{q^4+q^3+q^2+q} &= A^{q^8+q^7+q^6+q^5+q^4+q^3+q^2+q} \\
\left(A^{q^8+q^7+\cdots+q^2+q}\right)^{q^8} \cdot A^{q^8+q^7+\cdots+q^2+q} &= A^{q^{16}+q^{15}+\cdots+q^2+q} \\
\left(A^{q^{16}+q^{15}+\cdots+q^2+q}\right)^{q^4} \cdot A^{q^4+q^3+q^2+q} &= A^{q^{20}+q^{19}+\cdots+q^2+q}
\end{aligned}$$

The above addition chain requires 5 multiplications in $GF(q^{21})$ and 6 exponentiations in $GF(q^{21})$ which is in complete agreement with Theorem 7.3. Finally notice that for α a root of $P(x)$ and $\omega = 3 \in GF(p)$, and $q = p^2 \equiv 1 \bmod 21$, we have the following identities: $\alpha^{iq} \equiv (73\alpha)^i$, $\alpha^{iq^2} \equiv (122\alpha)^i$, $\alpha^{iq^3} \equiv (25\alpha)^i$, and $\alpha^{iq^8} \equiv (117\alpha)^i$, for $i = 1, 2, \cdots m - 1 = 20$. This implies that in computing an exponentiation to the q^eth power, one will perform at most $m - 1$ multiplications by an element of $GF(q)$ as mentioned above.

7.4.3 Fields $GF(q^m)$ with Binary s-ESP Field Polynomials

Irreducible All One Polynomials and Equally Spaced Polynomials have been proposed in [IT89, Ito91, HWB92, cKKS98, WH98, LLL01] to optimized the arithmetic in fields of characteristic 2. Nevertheless, these types of polynomials have not been treated in the literature for the case of odd characteristic extension fields. This section considers fields with binary irreducible s-ESPs as their field polynomial. Section 2.6.4 provides the reader with the necessary definitions regarding AOPs and ESPs as well as methods for the construction of irreducible polynomials of this form.

In the following we show the computational advantages derived from choosing a s-ESP as our field polynomial. We consider fields $GF(q^{sk}) \cong GF(q^m)$, $m = sk$, with a binary irreducible s-ESP as their field polynomial. Again, we will show that raising elements of these fields to the q^e-th power is computationally inexpensive. We look again at the representatives of the residue classes which contain $\alpha^{i\,q^e}$ in (7.5).

Theorem 7.5. *Let* $P(x) = x^{sk} + x^{s(k-1)} + \cdots + x^s + 1$ *be a binary irreducible* s*-ESP over* $GF(q)$ *and* $P(\alpha) = 0$*. Then an element* $\alpha^l \in GF(q^{sk})^*$*,* $l > 0$*, has the following polynomial representation:*

$$\alpha^l \equiv \begin{cases} \alpha^r, & \textit{if} \quad 0 \le r < sk \\ \sum_{i=0}^{k-1} -\alpha^{is+(r-k)}, & \textit{if} \quad sk \le r < sk+s \end{cases}$$

where $l \equiv r \bmod (sk+s)$ *and* $ord(\alpha) = sk + s$*.*

Proof. Let $P(x)$ and α be as defined in the theorem. Then, all α^l with $0 \le l < sk$ are distinct monomials, elements of $GF(q^{sk})^*$. For $l \ge sk$, we have the following equivalences:

$$\begin{aligned}
\alpha^{sk} \quad & \equiv -1 - \alpha^s - \cdots - \alpha^{sk-s} \\
\alpha^{sk+1} \quad & \equiv -\alpha - \alpha^{s+1} - \cdots - \alpha^{sk-s+1} \\
& \vdots \\
\alpha^{sk+s-1} \quad & \equiv -\alpha^{s-1} - \alpha^{2s-1} - \cdots - \alpha^{sk-1} \\
\alpha^{sk+s} \quad & \equiv 1
\end{aligned}$$

It follows from the congruences above that $\mathrm{ord}(\alpha) = s(k+1)$ and therefore $\alpha^l \equiv \alpha^{l \bmod (sk+s)}$. The upper part of the congruence, where $0 \le (l \bmod (sk+s)) < sk$, is now clear. For the other case, where $sk \le (l \bmod (sk+s)) < sk+s$, α^l is a polynomial with equally space coefficients of the form shown above. Finally, notice that since α is a root of $P(x)$, the above holds also true for fields of the form $GF((q^t)^{sk})$ where $\gcd(t, sk) = 1$.

It follows from Theorem 7.5 that the matrix in (7.6), which describes the eth iterates of the Frobenius map contains entries equal to -1, 0, or 1. Hence, multiplication by the matrix does not require any subfield multiplications but only additions and subtractions.

7.5 Notes and Further References

Reference [WTS+85] examines the hardware implementation of a Massey-Omura parallel multiplier using a normal basis representation and apply their design to the computation of the inverse in $GF(2^n)$, using a pipeline design. However, the inversion algorithm in [WTS+85] is impractical for sizes of n needed in cryptographic applications. The treatment in [IT88] is concerned only with fields of characteristic two and a normal basis representation of the field elements. Feng [Fen89] proposes an algorithm for inversion in fields $GF(2^n)$ using normal basis which requires $\rfloor \log_2(n-1)\rfloor + HW(n-1)$ multiplications and a number of cyclic shifts which are not considered by the author. The method in [Fen89] yields addition chains which are similar to the ones of the ITA. Reference [MK00] notice that in a hardware implementation the latency of the ITA can be reduced from $\lfloor \log_2(m-1)\rfloor + HW(m-1) - 1$ in a field $GF(2^m)$ to $\lfloor \log_2(m-1)\rfloor - 1$ by performing certain operations in parallel. Reference [HWB92] applies parallel multipliers to efficiently compute the inverse of elements in $GF(2^n)$ using a variant of Fermat's Little Theorem. Finally, [GP97] describes a version of the ITI algorithm applied to fields $GF((2^n)^m)$ in a polynomial basis representation applied to elliptic curve cryptosystems.

Both [HWB92] and [WH98] use irreducible AOPs and ESPs over $GF(2)$ to implement efficient parallel multipliers. The complexity of the proposed multipliers in both of these contributions is less than that of the parallel multiplier proposed in [IT89]. Furthermore, since inversion can be achieved by repeated multiplications, [HWB92] applies parallel multipliers to efficiently compute the inverse of elements in $GF(2^m)$ using a variant of Fermat's Little Theorem.

Other possibilities to compute the inverse of an element in a finite field have also been extensively explored. For example, non-systolic hardware implementations based on the extended Euclidean algorithm have been proposed [AFM89, BCH93] as well as systolic implementations [GW98b, GW98a], and VLSI algorithms based on the extended binary $\gcd$ algorithm [WTT02, WmWSH02]. Notice that all of the above techniques except for [GP02] study inversion in binary fields only.

CHAPTER 8

Discussion

8.1 Summary and Conclusions

In this thesis, we have focused on the development of hardware architectures for addition, multiplication, and inversion in fields $GF(p^m)$. In studying arithmetic in $GF(p^m)$ fields we have taken a bottom-up approach. First, we discussed thoroughly architectures to implement addition and multiplication over $GF(p)$. We make particular emphasis on architectures for *small* $GF(p)$ fields where $p < 32$. In particular, we proposed a new method to design small $GF(p)$ multipliers which can achieve up to a 30% improvement over previous architectures. The method is based on two observations: (i) for moduli which can be represented with less than 5 bits, it is very efficient to implement the modulo arithmetic operations as Boolean equations of the input bits and (ii) if the implementation method of (i) is assumed, then it is possible to reduce the complexity of the resulting Boolean equations by using a redundant encoding of the value zero.

The second part of this thesis is concerned with multiplier architectures for fields $GF(p^m)$. We investigated two strategies towards the implementation of these architectures. First, we explored a generalization of digit architectures originally proposed for fields $GF(2^n)$ to the odd characteristic case. Both Least Significant Digit (LSD) multiplier architectures and Most Significant Digit (MSD) architectures were considered. As in the case of characteristic two, Most Significant Digit architectures tend to have increased area and delay requirements. We implemented an arithmetic unit for $GF(3^m)$ fields on an FPGA and compared its performance to previous implementations and to our theoretical complexity models which agreed with our practical results. Not surprisingly, we find that in odd characteristic fields, it is possible to achieve comparable performance to binary fields at twice the hardware cost. The result is particularly interesting because fields of characteristic three allow for shorter signatures than binary fields (at twice the hardware cost but with comparable performance). We also showed that in characteristic three the cubing operation can be implemented using a number of adders/subtracters linear on the degree m of the irreducible polynomial defining the field. Finally, we introduced optimal irreducible polynomials in the sense that they optimize the reduction operation during the normal operation of the multiplication algorithm as well as when a cubing operation is being computed.

The second approach that we investigated towards the implementation of architectures for fields $GF(p^m)$ was based on systolization and scalability. In other words, we proposed architectures which are able to process fields $GF(p^m)$, for constant p and different values of m without recurring to changing the hardware or to reconfigurability, as in the case of FPGAs. To achieve scalability we introduced a new algorithm, which at the cost of initial and final pre-computations, allows us to perform modular reductions for any irreducible polynomial which satisfies a certain optimality criteria. The criteria is analogue to the definition optimal irreducible polynomials in Chapter 5 and does not constitute a major restriction for choosing suitable irreducible polynomials in practical applications. Notice that the method is also analogue to the Montgomery method of multiplication in the sense that both require initial and final computation stages. We implemented the basic cell of an LSD-based systolic multiplier on 0.18μm CMOS technology and provided time and area complexities.

Finally, we tackle the problem of inversion in fields $GF(q^m)$, $q = p^n$, by giving a generalization of the Itoh and Tsujii inversion algorithm to fields of odd characteristic and a standard basis representation. We introduce families of irreducible polynomials which reduce the complexity of exponentiating to the q-th power where $q = p^n$ and p is the field characteristic. By reducing the complexity of this operation, we also reduce the overall time required to compute an inverse in $GF(q^m)$. In particular, for the families of fields proposed, the inverse computation time complexity is essentially given by $\lfloor \log_2(m-1) \rfloor + HW(m-1) - 1$ multiplications.

8.2 Recommendations for Further Research

During the research which lead to this thesis, several questions and open problems remained unanswered. This section summarizes these questions and suggests several avenues to extend the results presented in this work.

- **Finite field hardware implementation.** One natural avenue for further research is to generalize the Montgomery method for $GF(2^n)$ fields to $GF(p^m)$ fields and compare it to our previously proposed architectures. The main contribution here would be to allow for scalable support of any field $GF(p^m)$, for any value of m. In previous work, it is assumed that the ground field is fixed, which for small values of p (by small values, we mean $p < 32$) means arithmetic done through table-lookups. For larger values of p, it would be interesting to investigate how the Montgomery technique (for fields $GF(p)$) compares to RNS-based techniques. A second possibility is to investigate the cost of incorporating $GF(p^m)$ into unified architecture units which already support $GF(p)$ and $GF(2^n)$arithmetic. Such architectures are important since standards support ECC based on fields $GF(p)$, $GF(2^n)$, and $GF(p^m)$, for p odd and prime.

- **Elliptic curves and Pairing-based cryptosystems.** A logical next step to this thesis would be to implement ECC and pairing-based crypto-processors based on $GF(p^m)$ arithmetic. These processors are important since the main application of finite field arithmetic and, in particular, of $GF(p^m)$ fields (in this context) is the implementation of ECC and of ID-based encryption and short signature schemes based on parings. To the author's knowledge there are not documented hardware implementations of such schemes in the literature. Although, throughput is always a main factor in such hardware implementations, it would be also important to consider implementations which can trade-off power, throughput, and area according to application requirements. Another, interesting area of further research is the implementation of the above cryptographic processors on systems which combine programmable cores and reconfigurable logic. This is specially interesting for schemes based on $GF(p^m)$ arithmetic because it is conceivable that one could take advantage of the programmable cores to implement the basic $GF(p)$ arithmetic and use the reconfigurable logic to implement the polynomial arithmetic required to support operations in

$GF(p^m)$. Such implementations have not been studied yet in the research community.

- **Hyperelliptic curves cryptosystems (HECC).** Recently, there has been a lot of work on HECC because they use smaller operands than ECC and thus, they might be better suited to constrained environments. There is already some hardware implementations of HECC over $GF(2^n)$as well as the first embedded processor implementations over $GF(2^n)$. Because of HECC's smaller operands, it seems natural to extend the work of OEFs on ECC to HECC and to examine implementation aspects on systems which combine programmable cores and reconfigurable components. Notice that no work on hardware or embedded processor implementations of HECC over odd characteristic fields has been documented in the literature. Similarly, it would be interesting to explore the performance of OEF-based HECC on general purpose processors and embedded processors, since OEFs where originally introduced as a way of obtaining fast implementations of cryptographic systems on software platforms.

APPENDIX A

Irreducible Polynomials over $GF(p)$

A.1 Irreducible Binomials over $GF(p)$

The generation of irreducible binomials over $GF(p)$ follows easily from Theorem 2.2 as their existence is entirely established. Reference [BP01a] provides an algorithm based on Theorem 2.2 which on input accepts a field order and the approximate prime size and as output provides a prime of the form $p = 2^n - c$, for small c, and an irreducible binomial of the form $x^m - 2$. We used a modification of such algorithm whose implementation is shown in Section A.1.1 of this appendix. Differences include: the generation of the prime p of special form, which in our case does not apply since we are interested in irreducible polynomials for all primes in the range of interest and not limiting the irreducible binomial to the form $x^m - 2$ but rather accepting binomials of the form $x^m - \omega$ for the smallest ω possible. Notice that [BP01a] also provides tables with irreducible binomials in the range $3 \leq p \leq 521$, p prime, but only for *some* p, whereas Table A.1 does the same for *all* primes in this range. The number theory and multi-precision library NTL [Sho01] was used to implement the search algorithm in C++.

Table A.1. Irreducible binomials of the form $x^m - \omega$ over $GF(p)$, for all m such that $2^{160} \leq p^m \leq 2^{256}$ and $3 \leq p \leq 521$ with $n = \lfloor \log_2 p \rfloor$.

p	n	m	$n \cdot m$	ω	p	n	m	$n \cdot m$	ω	p	n	m	$n \cdot m$	ω	p	n	m	$n \cdot m$	ω	p	n	m	$n \cdot m$	ω
3	1	—	—	—	101	6	32	192	2	211	7	27	189	2	337	8	21	168	2	449	8	28	224	3
5	2	128	256	2			40	240	2			30	210	4		8	24	192	10			32	256	3
7	2	81	162	2	103	6	27	162	2			35	245	2		8	27	216	2	457	8	24	192	13
11	3	—	—	—			34	204	2	223	7	27	189	3		8	28	224	5			27	216	3
13	3	54	162	2	107	6	—	—	—	227	7	—	—	—		8	32	256	5			32	256	5
		64	192	2	109	6	27	162	3	229	7	24	168	6	347	8	—	—	—	461	8	20	160	2
		72	216	2			32	192	2			27	189	3	349	8	24	192	2			23	184	2
		81	243	2			36	216	6			32	224	2		8	27	216	2			25	200	2
17	4	64	256	3	113	6	28	168	3			36	252	6		8	29	232	2			32	256	2
19	4	54	216	4			32	192	3	233	7	29	203	2		8	32	256	2	463	8	21	168	2
23	4	—	—	—	127	6	27	162	3			32	224	3	353	8	22	176	3			22	176	2
		49	196	2			42	252	9	239	7	34	238	2		8	32	256	3			27	216	2
		56	224	2	131	7	25	175	2	241	7	24	168	7	359	8	—	—	—	467	8	—	—	—
		64	256	2			26	182	3			25	175	3	367	8	27	216	2	479	8	—	—	—
31	4	45	180	3	137	7	32	224	3			27	189	2	373	8	24	192	2	487	8	27	216	2
		50	200	2			34	238	3			30	210	7		8	27	216	2	491	8	25	200	2
		54	216	5	139	7	23	161	2			32	224	7		8	31	248	2	499	8	27	216	5
37	5	32	160	2			27	189	2			36	252	7		8	32	256	2	503	8	—	—	—
		36	180	2	149	7	32	224	2	251	7	25	175	3	379	8	21	168	2	509	8	32	256	2
		48	240	2	151	7	25	175	3	257	8	32	256	3		8	27	216	2	521	8	20	160	3
41	5	32	160	3			27	189	2	263	8	—	—	—	383	8	—	—	—			25	200	2
		40	200	6			30	210	5	269	8	32	256	2	389	8	32	256	2			26	208	3
		50	250	6	157	7	24	168	5	271	8	25	200	2	397	8	22	176	2					
43	5	42	210	9			26	182	2			27	216	2		8	24	192	5					
		49	245	2			27	189	3			30	240	2		8	27	216	3					
47	5	46	230	2			32	224	2	277	8	23	184	2		8	32	256	2					
53	5	32	160	2			36	252	5			24	192	5	401	8	20	160	3					
59	5	—	—	—	163	7	27	189	2			27	216	3		8	25	200	2					
61	5	32	160	2	167	7	—	—	—			32	256	2		8	32	256	3					
		36	180	2	173	7	32	224	2	281	8	20	160	3	409	8	24	192	7					
		40	200	2	179	7	—	—	—			25	200	2		8	27	216	2					
		45	225	2	181	7	24	168	2			28	224	3		8	32	256	7					
		48	240	2			25	175	2			32	256	3	419	8	22	176	3					
		50	250	2			27	189	2	283	8	27	216	3	421	8	20	160	2					
67	6	27	162	2			30	210	2	292	8	32	256	2		8	21	168	2					
		33	198	2			32	224	2	307	8	27	216	5		8	24	192	2					
71	6	35	210	2			36	252	2	311	8	25	200	2		8	25	200	2					
73	6	27	162	2	191	7	25	175	2			31	248	2		8	27	216	2					
		32	192	5	193	7	24	168	5	313	8	24	192	10		8	28	224	2					
		36	216	5			27	189	2			26	208	7		8	30	240	2					
79	6	27	162	2			32	224	5			27	216	2		8	32	256	2					
		39	234	2			36	252	5			32	256	5	431	8	25	200	5					
83	6	41	246	2	197	7	28	196	2	317	8	32	256	2	433	8	24	192	5					
89	6	32	192	3			32	224	2	331	8	22	176	5		8	27	216	3					
97	6	27	162	2	199	7	27	189	2			25	200	2		8	32	256	5					
		32	192	5			33	231	2			27	216	2	439	8	27	216	5					
		36	216	5	211	7	25	175	2			30	240	4	443	8	26	208	3					

A.1.1 C++ Source Code Used to Generate Table A.1

```
/*--------------------------------------------------------------------------
 *
 * Filename: GenIrreducBinomials
 *
 * Description:  This program generates and tests the generated polynomials
 * for irreducibility.  The irreducible polynomials are binomials
 *
 * The algorithm is based on Theorem 3.75 from Lidl and Niederreiter
 * Let t >= 2 be an integer and a in F^*_q. Then the binomial x^t - a
 * is irreducible in F_q if and only if the follwoign two conditions are
 * satisfied
 * (i)  Each prime factor of t divides the order of a in F^*_q but not
 *      (q-1)/e
 * (ii) q = 1 mod 4 if t = 0 mod 4.
 *
 *
 * Functions:
 *
 * Notes:
 *
 *
 * Revision History:
 * Person             Date            Comment
 *--------------------------------------------------------------------------
 * Jorge Guajardo      06/26/2002      Creation date
 *
 *------------------------------------------------------------------------*/

#include <NTL/ZZ_pXFactoring.h>
#include <NTL/ZZ_pEX.h>
#include <NTL/ZZ.h>

#define NUM_ODDPRIMES 55

const unsigned short primelist[]={
2, 3,      5,      7,     11,      13,     17,     19,     23,     29,   31,
37,     41,     43,     47,     53,     59,     61,     67,     71,   73,
79,     83,     89,     97,    101,    103,    107,    109,    113,   127,
131,    137,    139,    149,    151,    157,    163,    167,    173,  179,
181,    191,    193,    197,    199,    211,    223,    227,    229,  233,
239,    241,    251,    257};


void FactorLong(vec_ZZ &prime_list,
vec_ZZ &prime_multiplicity,
const ZZ  &integer_Z);

void orderOfLong(ZZ &order,
 const ZZ &element,
 const ZZ &prime,
 const vec_ZZ &prime_list,
 const vec_ZZ &prime_multiplicity);

void PrintFactorization(const vec_ZZ &prime_list, const vec_ZZ &prime_multiplicity);


int main()
{

    vec_ZZ prime_list, prime_multiplicity;
    vec_ZZ primesOforder_L, primesOforder_multiplicity_L;
    ZZ dummy_number;
    ZZ order, prime_Z;
    PrimeSeq s;
    long p;
    int indx, jndx, commaCounter;
    int sizeOfprime_list;

    //This factors the order of GF(p)*, so this basically gives us the
    //possible orders of elements in GF(p)*
    p = s.next();
    cout << "p \t p-1 \t\t a \t ord(a) \t poss. p's in t " << endl;
    while (p <= 259) {
if (p != 2)
{
    cout << "---------------------------------------------------------------------" << endl;

    dummy_number = p -1;
    FactorLong(prime_list,prime_multiplicity,dummy_number);

    // Now find out the order e of all elements between 2 and p-1
    // Factor the order (e) of each element and check if each prime p_i in the
    // factorization of the order of the element divides p-1/e.  If it does
    // then  p_i can not be a factor in t
```

```
    cout << p << " \t ";
    PrintFactorization(prime_list,prime_multiplicity); cout << "\t";
    for (indx = 2; indx < p; indx ++)
    {
dummy_number = indx;
prime_Z = p;
orderOfLong(order,dummy_number, prime_Z, prime_list, prime_multiplicity);
cout << "\t " << dummy_number << " \t " << order << " \t \t";
//Factor the order of the element
FactorLong(primesOforder_L, primesOforder_multiplicity_L,order);
//Check divisibility of p-1/e by prime factors of e
commaCounter = 0;
for (jndx=0; jndx < primesOforder_L.length(); jndx++)
{
    if ((divide(dummy_number,((prime_Z -1)/order),primesOforder_L[jndx])) == 0)
    {
if (commaCounter != 0)
    cout << ",";
cout << primesOforder_L[jndx];
commaCounter++;
    }
}
cout << endl;
cout << "  \t     \t " ;
    }

}
cout << endl;
p = s.next();
    }
    cout << endl;


}


/*-------------------------------------------------------------------------
 * Description:  Find the order of element in GF(prime)
 *
 * Note:  Only works for small primes really, because assumes that you can
 *        factor prime -1
 *        Because we will be doing this for all elements is GF(p) then it
 *        assumes that we precompute the factorization of p-1
 *-----------------------------------------------------------------------*/
void orderOfLong(ZZ &order, const ZZ &element, const ZZ &prime,
 const vec_ZZ &prime_list,
 const vec_ZZ &prime_multiplicity)
{
    int indx, jndx;
    int sizeOfPrimeList;
    ZZ result;

    if (IsOne(element))
order = 1;
    else if (IsZero(element))
    {
order = 0;
cout << "Warning: you asked to calculate the order of ZERO" << endl;
    }
    else
    {

sizeOfPrimeList = prime_list.length();
order = prime -1;
for (indx = 0; indx < sizeOfPrimeList; indx++)
{
    for (jndx = 1; jndx <= prime_multiplicity[indx]; jndx++)
    {
order = order / prime_list[indx];
PowerMod(result, element, order, prime);
if (!IsOne(result))
{
    order = order *prime_list[indx];
    break;
}
    }

}
    }

}


/*-------------------------------------------------------------------------
```

```
 * Description:  Factor integers less than 259 by dividing by list of primes
 * less than 259
 * prime_list = p_1 p_2 p_3 ... p_n
 * prime_multiplicity = e_1 e_2 ... e_n
 *
 * where integer = p_1^e_1 p_2^e_2 ... p_n^e_n
 *
 * Note:  Only works for small primes
 *
 *--------------------------------------------------------------------------*/
void FactorLong(vec_ZZ &prime_list,
vec_ZZ &prime_multiplicity,
const ZZ  &integer_Z)
{
    ZZ dummy_Z;
    ZZ quotient_Z;
    int flag = 1;
    int indx = 0;
    int counter_numPrimes;
    int counter_multiplicity;
    long remainder;

    /*Just make sure that we don't try to factor bigger numbers for now */
    if (integer_Z > 259)
Error("The integer you are trying to factor is greater than 259\n");

    dummy_Z = integer_Z;
    counter_numPrimes = 0;
    while(flag)
    {
if (IsOne(dummy_Z))
    flag = 0;
//Tests if dummy_Z is prime, if the function returns 1
//then dummy_Z is prime
else if (ProbPrime(dummy_Z) == 1)
{
    prime_list.SetLength(counter_numPrimes+1);
    prime_multiplicity.SetLength(counter_numPrimes+1);
    prime_list[counter_numPrimes] = dummy_Z;
    prime_multiplicity[counter_numPrimes] = 1;
    flag = 0;  //end the big while loop
}
else
{
    //This takes care of dividing by the small primes and multiplicities
    remainder = DivRem(quotient_Z, dummy_Z ,primelist[indx]);

    if (remainder == 0)
    {
dummy_Z = quotient_Z;
prime_list.SetLength(counter_numPrimes+1);
prime_multiplicity.SetLength(counter_numPrimes+1);
prime_list[counter_numPrimes] = primelist[indx];
counter_multiplicity = 1;
while (remainder == 0)
{
    remainder = DivRem(quotient_Z, dummy_Z ,primelist[indx]);
    if (remainder == 0)
    {
dummy_Z = quotient_Z;
counter_multiplicity++;
//cout << counter_multiplicity << endl;
    }
    else
prime_multiplicity[counter_numPrimes] = counter_multiplicity;
}
counter_numPrimes++;
    }
    else
indx++;
}
    }//end big while

}


/*--------------------------------------------------------------------------
 * Description: Print Factorization of a number given its prime factors and
 * their multiplicity
 *
 *-------------------------------------------------------------------------- */
void PrintFactorization(const vec_ZZ &prime_list, const vec_ZZ &prime_multiplicity)
{
    long sizeOfprime_list;
    int indx;

    sizeOfprime_list = prime_list.length();
```

```
    for (indx=0; indx < sizeOfprime_list; indx++)
    {
cout << prime_list[indx];
if (prime_multiplicity[indx] != 1)
    cout << "^" << prime_multiplicity[indx] << " ";
else
    cout << " ";
    }
}
```

A.2 Irreducible Trinomials and Quadrinomials over $GF(3)$

The irreducible polynomials shown in Tables A.3, A.4, A.5, and A.6 were generated as explained in Section 5.5.3 performing an exhaustive search. Notice that this exhaustive search could be improved by including the irreducibility criteria described in [vzG01]. In the range $2 \leq m \leq 255$, there are only 23 degrees m for which we were unable to find trinomials (this agrees with the findings in [vzG01]) and thus, we provide irreducible quadrinomials for them in Table A.6. Finally, Table A.2 corresponds to optimal irreducible polynomials, where optimality is defined as described in Section 5.5.3. Example of the code used to generate Tables A.3, A.4, A.5, and A.6 is shown in Section A.2.1. The code was written by Guido Bertoni of the Politecnico di Milano and it uses the NTL library [Sho01].

Table A.2. Optimal irreducible trinomials of the form $x^m + p_t x^t + p_0$ over $GF(3)$, for m prime and $2 \leq m \leq 255$.

m	(p_t, t, p_0)	m	(p_t, t, p_0)	m	(p_t, t, p_0)
2	(2,1,2)	67	(1,2,2),(2,2,1),(2,11,2)	157	(1,22,2),(2,22,1)
3	(2,1,2)	71	(1,20,2),(2,20,1)	163	(2,59,2)
5	(2,1,2)	73	(2,1,2)	167	(2,71,2)
7	(1,2,2),(2,2,1)	79	(1,26,2),(2,26,1)	173	(2,7,2)
11	(1,2,2),(2,2,1),(2,3,2)	83	(2,27,2)	179	(2,59,2)
13	(2,1,2)	89	(2,13,2)	181	(2,37,2)
17	(2,1,2)	97	(1,12,2),(2,12,1)	191	(2,71,2)
19	(1,2,2),(2,2,1)	101	(2,31,2)	193	(1,12,2),(2,12,1)
23	(2,3,2)	103	(2,47,2)	197	—
29	(2,4,1),(1,4,2)	107	(2,3,2)	199	(2,35,2)
31	(2,5,2)	109	(2,9,2)	211	(2,89,2)
37	(1,6,2),(2,6,1)	113	(2,19,2)	223	—
41	(2,1,2)	127	(1,8,2),(2,8,1)	229	(1,72,2),(2,72,1)
43	(2,17,2)	131	(2,27,2)	233	—
47	(2,15,2)	137	(2,1,2)	239	(2,5,2)
53	(2,13,2)	139	(2,59,2)	241	(1,88,2),(2,88,1), (2,117,2)
59	(2,17,2)	149	—	251	(2,9,2)
61	(2,7,2)	151	(1,2,2),(2,2,1)		

Table A.3. Irreducible trinomials of the form $x^m + x^t + 2$ over $GF(3)$, $2 \leq m \leq 255$

m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t
2	1	22	5	43	26	71	20	93	70	117	52	143	108	167	92	187	8	211	122	238	5
3	2	23	8	44	3	72	28	95	48	119	2	144	56	168	28	188	11	214	65	239	24
4	1	24	4	45	28	73	30	96	16	120	4	145	24	169	24	191	116	215	36	240	8
5	4	25	6	46	5	76	9	97	12	121	40	147	8	170	43	192	32	216	4	241	88
6	1	26	7	47	32	77	16	99	74	124	25	148	3	171	20	193	12	217	132	242	115
7	2	27	20	48	8	78	13	100	25	125	52	150	73	172	19	194	55	219	26	243	122
8	2	28	2	51	50	79	26	101	70	126	49	151	2	173	166	195	26	220	15	244	31
9	4	29	4	52	7	80	2	102	25	127	8	152	18	174	73	196	79	222	89	245	148
11	2	30	1	53	22	81	40	103	50	128	6	153	94	176	12	198	29	224	12	246	13
12	2	31	20	54	1	83	32	104	5	131	48	155	12	177	52	199	164	225	16	247	122
13	4	32	5	55	26	84	14	107	32	133	88	156	26	178	11	200	3	227	68	248	50
14	1	33	28	56	3	85	16	108	2	134	61	157	22	179	104	201	88	228	14	249	76
15	2	35	2	59	20	86	13	109	88	135	44	158	61	180	38	203	8	229	72	251	26
16	4	36	14	60	2	87	26	111	2	136	57	159	32	181	40	204	50	230	73	252	98
17	16	37	6	61	30	88	6	112	6	137	136	160	4	182	25	205	78	232	30	253	12
18	7	39	26	63	26	89	64	113	70	139	80	162	19	183	2	206	61	234	91	254	73
19	2	40	1	64	3	90	19	114	7	140	59	163	80	184	20	208	10	235	26	255	26
20	5	41	40	67	2	91	74	115	32	141	64	164	15	185	64	209	40	236	9		
21	16	42	7	69	52	92	10	116	15	142	65	165	22	186	47	210	7	237	70		

Table A.4. Irreducible trinomials of the form $x^m + 2x^t + 1$ over $GF(3)$, $3 \leq m \leq 255$

m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t
3	1	31	5	62	10	93	23	126	52	159	32	191	71	227	11
5	1	33	5	63	26	94	30	127	8	162	80	193	12	229	72
6	2	34	2	66	10	95	47	131	27	163	59	194	24	230	64
7	2	35	2	67	2	97	12	133	15	165	22	195	26	234	104
9	4	37	6	69	17	99	19	134	4	166	54	198	38	235	26
10	2	38	4	70	4	101	31	135	44	167	71	199	35	237	70
11	2	39	7	71	20	102	2	137	1	169	24	201	88	238	4
13	1	41	1	73	1	103	47	138	34	170	32	202	62	239	5
14	4	42	10	74	12	106	26	139	59	171	20	203	3	241	88
15	2	43	17	77	16	107	3	141	5	173	7	205	9	242	2
17	1	45	17	78	14	109	9	142	40	174	52	206	94	243	121
18	8	46	6	79	26	110	22	143	35	177	52	209	40	245	97
19	2	47	15	81	40	111	2	145	24	178	26	211	89	247	122
21	5	50	6	82	2	113	19	146	2	179	59	214	6	249	59
22	4	51	1	83	27	115	32	147	8	181	37	215	36	250	104
23	3	53	13	85	16	117	52	151	2	182	34	217	85	251	9
25	3	54	14	86	34	118	34	153	59	183	2	218	18	253	7
26	2	55	11	87	26	119	2	154	32	185	64	219	25	254	16
27	7	58	8	89	13	121	1	155	12	186	46	222	4	255	26
29	4	59	17	90	34	122	14	157	22	187	8	225	16		
30	4	61	7	91	17	125	52	158	52	190	94	226	38		

Table A.5. Irreducible trinomials of the form $x^m + 2x^t + 2$ over $GF(3)$, $2 \leq m \leq 255$

m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t	m	t
2	1	22	5	43	17	71	51	93	23	117	65	143	35	167	71	187	65	211	89	238	5
3	1	23	3	44	3	72	28	95	47	119	3	144	56	168	28	188	11	214	65	239	5
4	1	24	4	45	17	73	1	96	16	120	4	145	73	169	37	191	71	215	59	240	8
5	1	25	3	46	5	76	9	97	81	121	1	147	43	170	43	192	32	216	4	241	117
6	1	26	7	47	15	77	25	99	19	124	25	148	3	171	151	193	81	217	85	242	115
7	5	27	7	48	8	78	13	100	25	125	73	150	73	172	19	194	55	219	25	243	121
8	2	28	2	51	1	79	53	101	31	126	49	151	125	173	7	195	49	220	15	244	31
9	5	29	25	52	7	80	2	102	25	127	119	152	18	174	73	196	79	222	89	245	97
11	3	30	1	53	13	81	41	103	47	128	6	153	59	176	12	198	29	224	12	246	13
12	2	31	5	54	1	83	27	104	5	131	27	155	129	177	83	199	35	225	209	247	125
13	1	32	5	55	11	84	14	107	3	133	15	156	26	178	11	200	3	227	11	248	50
14	1	33	5	56	3	85	31	108	2	134	61	157	69	179	59	201	113	228	14	249	59
15	7	35	17	59	17	86	13	109	9	135	91	158	61	180	38	203	3	229	79	251	9
16	4	36	14	60	2	87	37	111	13	136	57	159	127	181	37	204	50	230	73	252	98
17	1	37	13	61	7	88	6	112	6	137	1	160	4	182	25	205	9	232	30	253	7
18	7	39	7	63	37	89	13	113	19	139	59	162	19	183	181	206	61	234	91	254	73
19	11	40	1	64	3	90	19	114	7	140	59	163	59	184	20	208	10	235	83	255	229
20	5	41	1	67	11	91	17	115	83	141	5	164	15	185	121	209	49	236	9		
21	5	42	7	69	17	92	10	116	15	142	65	165	77	186	47	210	7	237	167		

Table A.6. Irreducible quadrinomials $x^m + p_{t_1}x^{t_1} + p_{t_2}x^{t_2} + p_0$ over $GF(3)$ for degrees with no trinomials.

m	t_1	t_2	(p_{t_1}, p_{t_2}, p_0)
49	3	2	$(-1, \pm 1, \pm 1)$
57	7	2	$(1, \pm 1, \mp 1)$
65	5	3	$(1, 1, \pm 1$
68	3	2	$(\pm 1, 1, 1)$
75	5	4	$(-1, \pm 1, \pm 1)$
98	4	3	$(1, \pm 1, 1)$
105	6	2	$(\pm 1, \pm 1, \pm 1)$
123	7	4	$(1, \pm 1, \mp 1)$
129	3	2	$(-1, \pm, \pm 1)$
130	10	6	$(1, 1, 1)$
132	10	1	$(1, \pm 1, 1)$
149	11	10	$(-1, \pm 1, \pm 1)$
161	9	5	$(1, 1, \pm 1)$
175	10	8	$(\pm 1, \pm 1, \pm 1)$
189	9	7	$(1, 1, \pm 1)$
197	9	4	$(-1, \pm 1, \pm 1)$
207	11	8	$(-1, \pm 1, \pm 1)$
212	14	3	$(1, \pm 1, 1)$
213	12	1	$(\pm 1, 1, \mp 1)$
221	12	2	$(\pm 1, \pm 1, \pm 1)$
223	8	5	$(\pm 1, -1, \pm 1)$
231	8	7	$(\pm 1, 1, \mp 1)$
233	6	2	$(\pm 1, \pm 1, \pm 1)$

A.2.1 C++ Source Code Used to Generate Irreducible Trinomials

```
/*--------------------------------------------------------------------------
 *
 * Filename: GenIrreducTrinom.C
 *
 * Description:  This program generates and tests the generated polynomials
 * for irreducibility.  The irreducible polynomials are trinomials
 *
 *
 *
 * Revision History:
 * Person              Date            Comment
 *--------------------------------------------------------------------------
 * Guido Bertoni      06/26/2002      Creation date
 *
 *-------------------------------------------------------------------------*/

#include <NTL/ZZ_pXFactoring.h>

#define MAX_DEGREE_MINE 256;

//#define debug_interactive

int main()
{
   ZZ p;
   //cin >> p;
   p=3;

   ZZ_p::init(p);

   ZZ_pX degree,d,coeffzero,poly_sum;
   vec_pair_ZZ_pX_long factors;
   long irreducibile;
   int counter,counter2,flag;

   flag=0;

   SetCoeff(degree,2,1);
   SetCoeff(d,1,1);

   SetCoeff(coeffzero,0,2);

   for(counter=2;counter<256;counter++){
     //cout << "cycle" << counter << "\n";
     degree=0;
     d=0;
     SetCoeff(degree,counter,1);
     SetCoeff(d,0,1);
     for(counter2=1;counter2<counter;counter2++){
       poly_sum=0;
       LeftShift(d,d,1);
       poly_sum=degree+d;
       poly_sum=poly_sum+coeffzero;
       irreducibile=DetIrredTest(poly_sum);

#ifdef debug_interactive
       cout << "d ";
       cout << d << "\n";
       cout << "irre " << poly_sum << "\n";
       cout << "degree d" << counter2 << "\n";
       cout << "irreduc ???" << irreducibile << "\n";
#endif

       if(irreducibile!=0){
 flag=1;
 cout << counter << " " << counter2 << "\n";
 break;
       }
     }
   }
   cout <<"\n any irr: " << flag;
}
```

APPENDIX B

$GF(p)$ Adder Complexities

Table B.1. Normalized area complexity of different $GF(p)$ adders for $3 \leq p \leq 521$

p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]	p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]
3	2	41	18	7	13	66	21	12	239	8	681574	456968	220	104	264	109	60
5	3	249	75	21	25	99	34	25	241	8	681574	464648	200	104	264	109	76
7	3	249	147	13	25	99	34	17	251	8	681574	504008	96	104	264	109	60
11	4	1331	484	41	39	132	48	31	257	9	3067084	594441	3035	123	297	125	117
13	4	1331	676	30	39	132	48	31	263	9	3067084	622521	2965	123	297	125	101
17	5	6656	1445	119	54	165	63	54	269	9	3067084	651249	2895	123	297	125	101
19	5	6656	1805	106	54	165	63	46	271	9	3067084	660969	2872	123	297	125	93
23	5	6656	2645	80	54	165	63	38	277	9	3067084	690561	2801	123	297	125	101
29	5	6656	4205	41	54	165	63	38	281	9	3067084	710649	2755	123	297	125	101
31	5	6656	4805	28	54	165	63	29	283	9	3067084	720801	2731	123	297	125	93
37	6	31948	8214	239	70	198	78	61	293	9	3067084	772641	2614	123	297	125	101
41	6	31948	10086	208	70	198	78	61	307	9	3067084	848241	2450	123	297	125	93
43	6	31948	11094	192	70	198	78	53	311	9	3067084	870489	2404	123	297	125	84
47	6	31948	13254	161	70	198	78	45	313	9	3067084	881721	2380	123	297	125	93
53	6	31948	16854	114	70	198	78	53	317	9	3067084	904401	2333	123	297	125	84
59	6	31948	20886	67	70	198	78	45	331	9	3067084	986049	2170	123	297	125	93
61	6	31948	22326	52	70	198	78	45	337	9	3067084	1022121	2099	123	297	125	101
67	7	149094	31423	591	87	231	93	77	347	9	3067084	1083681	1982	123	297	125	84
71	7	149094	35287	555	87	231	93	68	349	9	3067084	1096209	1959	123	297	125	84
73	7	149094	37303	536	87	231	93	77	353	9	3067084	1121481	1912	123	297	125	101
79	7	149094	43687	482	87	231	93	60	359	9	3067084	1159929	1842	123	297	125	84
83	7	149094	48223	445	87	231	93	68	367	9	3067084	1212201	1748	123	297	125	76
89	7	149094	55447	391	87	231	93	68	373	9	3067084	1252161	1678	123	297	125	84
97	7	149094	65863	318	87	231	93	77	379	9	3067084	1292769	1608	123	297	125	76
101	7	149094	71407	282	87	231	93	68	383	9	3067084	1320201	1561	123	297	125	68
103	7	149094	74263	263	87	231	93	60	389	9	3067084	1361889	1491	123	297	125	101
107	7	149094	80143	227	87	231	93	60	397	9	3067084	1418481	1397	123	297	125	93
109	7	149094	83167	209	87	231	93	60	401	9	3067084	1447209	1351	123	297	125	101
113	7	149094	89383	172	87	231	93	68	409	9	3067084	1505529	1257	123	297	125	93
127	7	149094	112903	45	87	231	93	44	419	9	3067084	1580049	1140	123	297	125	93
131	8	681574	137288	1344	104	264	109	93	421	9	3067084	1595169	1117	123	297	125	93
137	8	681574	150152	1281	104	264	109	93	431	9	3067084	1671849	1000	123	297	125	76
139	8	681574	154568	1260	104	264	109	84	433	9	3067084	1687401	976	123	297	125	93
149	8	681574	177608	1156	104	264	109	84	439	9	3067084	1734489	906	123	297	125	76
151	8	681574	182408	1136	104	264	109	76	443	9	3067084	1766241	859	123	297	125	76
157	8	681574	197192	1073	104	264	109	76	449	9	3067084	1814409	789	123	297	125	101
163	8	681574	212552	1011	104	264	109	84	457	9	3067084	1879641	695	123	297	125	93
167	8	681574	223112	969	104	264	109	76	461	9	3067084	1912689	649	123	297	125	84
173	8	681574	239432	907	104	264	109	76	463	9	3067084	1929321	625	123	297	125	76
179	8	681574	256328	844	104	264	109	76	467	9	3067084	1962801	578	123	297	125	84
181	8	681574	262088	824	104	264	109	76	479	9	3067084	2064969	438	123	297	125	68
191	8	681574	291848	720	104	264	109	60	487	9	3067084	2134521	344	123	297	125	76
193	8	681574	297992	699	104	264	109	93	491	9	3067084	2169729	298	123	297	125	76
197	8	681574	310472	657	104	264	109	84	499	9	3067084	2241009	204	123	297	125	76
199	8	681574	316808	636	104	264	109	76	503	9	3067084	2277081	157	123	297	125	68
211	8	681574	356168	512	104	264	109	76	509	9	3067084	2331729	87	123	297	125	68
223	8	681574	397832	387	104	264	109	60	521	10	13631488	2714410	6599	141	330	142	126
227	8	681574	412232	345	104	264	109	76	—	—	—	—	—	—	—	—	—
229	8	681574	419528	324	104	264	109	76	—	—	—	—	—	—	—	—	—
233	8	681574	434312	283	104	264	109	76	—	—	—	—	—	—	—	—	—

Table B.2. Normalized time complexity of different $GF(p)$ adders for $3 \leq p \leq 521$

p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]	p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]
3	2	3	3	5	6	12	10	4	239	8	9	9	15	13	12	17	8
5	3	4	4	7	8	12	12	5	241	8	9	9	15	13	12	17	8
7	3	4	4	7	8	12	12	5	251	8	9	9	15	13	12	17	8
11	4	5	5	9	10	12	14	6	257	9	10	10	16	14	12	18	8
13	4	5	5	9	10	12	14	6	263	9	10	10	16	14	12	18	8
17	5	6	6	10	11	12	15	7	269	9	10	10	16	14	12	18	8
19	5	6	6	10	11	12	15	7	271	9	10	10	16	14	12	18	8
23	5	6	6	10	11	12	15	7	277	9	10	10	16	14	12	18	8
29	5	6	6	10	11	12	15	7	281	9	10	10	16	14	12	18	8
31	5	6	6	10	11	12	15	7	283	9	10	10	16	14	12	18	8
37	6	7	7	12	12	12	16	7	293	9	10	10	16	14	12	18	8
41	6	7	7	12	12	12	16	7	307	9	10	10	16	14	12	18	8
43	6	7	7	12	12	12	16	7	311	9	10	10	16	14	12	18	8
47	6	7	7	12	12	12	16	7	313	9	10	10	16	14	12	18	8
53	6	7	7	12	12	12	16	7	317	9	10	10	16	14	12	18	8
59	6	7	7	12	12	12	16	7	331	9	10	10	16	14	12	18	8
61	6	7	7	12	12	12	16	7	337	9	10	10	16	14	12	18	8
67	7	8	8	13	13	12	17	8	347	9	10	10	16	14	12	18	8
71	7	8	8	13	13	12	17	8	349	9	10	10	16	14	12	18	8
73	7	8	8	13	13	12	17	8	353	9	10	10	16	14	12	18	8
79	7	8	8	13	13	12	17	8	359	9	10	10	16	14	12	18	8
83	7	8	8	13	13	12	17	8	367	9	10	10	16	14	12	18	8
89	7	8	8	13	13	12	17	8	373	9	10	10	16	14	12	18	8
97	7	8	8	13	13	12	17	8	379	9	10	10	16	14	12	18	8
101	7	8	8	13	13	12	17	8	383	9	10	10	16	14	12	18	8
103	7	8	8	13	13	12	17	8	389	9	10	10	16	14	12	18	8
107	7	8	8	13	13	12	17	8	397	9	10	10	16	14	12	18	8
109	7	8	8	13	13	12	17	8	401	9	10	10	16	14	12	18	8
113	7	8	8	13	13	12	17	8	409	9	10	10	16	14	12	18	8
127	7	8	8	13	13	12	17	8	419	9	10	10	16	14	12	18	8
131	8	9	9	15	13	12	17	8	421	9	10	10	16	14	12	18	8
137	8	9	9	15	13	12	17	8	431	9	10	10	16	14	12	18	8
139	8	9	9	15	13	12	17	8	433	9	10	10	16	14	12	18	8
149	8	9	9	15	13	12	17	8	439	9	10	10	16	14	12	18	8
151	8	9	9	15	13	12	17	8	443	9	10	10	16	14	12	18	8
157	8	9	9	15	13	12	17	8	449	9	10	10	16	14	12	18	8
163	8	9	9	15	13	12	17	8	457	9	10	10	16	14	12	18	8
167	8	9	9	15	13	12	17	8	461	9	10	10	16	14	12	18	8
173	8	9	9	15	13	12	17	8	463	9	10	10	16	14	12	18	8
179	8	9	9	15	13	12	17	8	467	9	10	10	16	14	12	18	8
181	8	9	9	15	13	12	17	8	479	9	10	10	16	14	12	18	8
191	8	9	9	15	13	12	17	8	487	9	10	10	16	14	12	18	8
193	8	9	9	15	13	12	17	8	491	9	10	10	16	14	12	18	8
197	8	9	9	15	13	12	17	8	499	9	10	10	16	14	12	18	8
199	8	9	9	15	13	12	17	8	503	9	10	10	16	14	12	18	8
211	8	9	9	15	13	12	17	8	509	9	10	10	16	14	12	18	8
223	8	9	9	15	13	12	17	8	521	10	11	11	18	14	12	18	9
227	8	9	9	15	13	12	17	8	—	—	—	—	—	—	—	—	—
229	8	9	9	15	13	12	17	8	—	—	—	—	—	—	—	—	—
233	8	9	9	15	13	12	17	8	—	—	—	—	—	—	—	—	—

Table B.3. Normalized area/time product of different $GF(p)$ adders for $3 \le p \le 521$

p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]	p	n	ROM (SC)	ROM (FC)	[BJM87b] (hybrid)	[BJM87b] (CLA)	[EB90] (CSA)	[Dug92] (type II)	[Hia02]
3	2	123	54	35	78	792	210	48	239	8	6134166	4112712	3300	1352	3168	1853	480
5	3	996	300	147	200	1188	408	125	241	8	6134166	4181832	3000	1352	3168	1853	608
7	3	996	588	91	200	1188	408	85	251	8	6134166	4536072	1440	1352	3168	1853	480
11	4	6655	2420	369	390	1584	672	186	257	9	30670840	5944410	48560	1722	3564	2250	936
13	4	6655	3380	270	390	1584	672	186	263	9	30670840	6225210	47440	1722	3564	2250	808
17	5	39936	8670	1190	594	1980	945	378	269	9	30670840	6512490	46320	1722	3564	2250	808
19	5	39936	10830	1060	594	1980	945	322	271	9	30670840	6609690	45952	1722	3564	2250	744
23	5	39936	15870	800	594	1980	945	266	277	9	30670840	6905610	44816	1722	3564	2250	808
29	5	39936	25230	410	594	1980	945	266	281	9	30670840	7106490	44080	1722	3564	2250	808
31	5	39936	28830	280	594	1980	945	203	283	9	30670840	7208010	43696	1722	3564	2250	744
37	6	223636	57498	2868	840	2376	1248	427	293	9	30670840	7726410	41824	1722	3564	2250	808
41	6	223636	70602	2496	840	2376	1248	427	307	9	30670840	8482410	39200	1722	3564	2250	744
43	6	223636	77658	2304	840	2376	1248	371	311	9	30670840	8704890	38464	1722	3564	2250	672
47	6	223636	92778	1932	840	2376	1248	315	313	9	30670840	8817210	38080	1722	3564	2250	744
53	6	223636	117978	1368	840	2376	1248	371	317	9	30670840	9044010	37328	1722	3564	2250	672
59	6	223636	146202	804	840	2376	1248	315	331	9	30670840	9860490	34720	1722	3564	2250	744
61	6	223636	156282	624	840	2376	1248	315	337	9	30670840	10221210	33584	1722	3564	2250	808
67	7	1192752	251384	7683	1131	2772	1581	616	347	9	30670840	10836810	31712	1722	3564	2250	672
71	7	1192752	282296	7215	1131	2772	1581	544	349	9	30670840	10962090	31344	1722	3564	2250	672
73	7	1192752	298424	6968	1131	2772	1581	616	353	9	30670840	11214810	30592	1722	3564	2250	808
79	7	1192752	349496	6266	1131	2772	1581	480	359	9	30670840	11599290	29472	1722	3564	2250	672
83	7	1192752	385784	5785	1131	2772	1581	544	367	9	30670840	12122010	27968	1722	3564	2250	608
89	7	1192752	443576	5083	1131	2772	1581	544	373	9	30670840	12521610	26848	1722	3564	2250	672
97	7	1192752	526904	4134	1131	2772	1581	616	379	9	30670840	12927690	25728	1722	3564	2250	608
101	7	1192752	571256	3666	1131	2772	1581	544	383	9	30670840	13202010	24976	1722	3564	2250	544
103	7	1192752	594104	3419	1131	2772	1581	480	389	9	30670840	13618890	23856	1722	3564	2250	808
107	7	1192752	641144	2951	1131	2772	1581	480	397	9	30670840	14184810	22352	1722	3564	2250	744
109	7	1192752	665336	2717	1131	2772	1581	480	401	9	30670840	14472090	21616	1722	3564	2250	808
113	7	1192752	715064	2236	1131	2772	1581	544	409	9	30670840	15055290	20112	1722	3564	2250	744
127	7	1192752	903224	585	1131	2772	1581	352	419	9	30670840	15800490	18240	1722	3564	2250	744
131	8	6134166	1235592	20160	1352	3168	1853	744	421	9	30670840	15951690	17872	1722	3564	2250	744
137	8	6134166	1351368	19215	1352	3168	1853	744	431	9	30670840	16718490	16000	1722	3564	2250	608
139	8	6134166	1391112	18900	1352	3168	1853	672	433	9	30670840	16874010	15616	1722	3564	2250	744
149	8	6134166	1598472	17340	1352	3168	1853	672	439	9	30670840	17344890	14496	1722	3564	2250	608
151	8	6134166	1641672	17040	1352	3168	1853	608	443	9	30670840	17662410	13744	1722	3564	2250	608
157	8	6134166	1774728	16095	1352	3168	1853	608	449	9	30670840	18144090	12624	1722	3564	2250	808
163	8	6134166	1912968	15165	1352	3168	1853	672	457	9	30670840	18796410	11120	1722	3564	2250	744
167	8	6134166	2008008	14535	1352	3168	1853	608	461	9	30670840	19126890	10384	1722	3564	2250	672
173	8	6134166	2154888	13605	1352	3168	1853	608	463	9	30670840	19293210	10000	1722	3564	2250	608
179	8	6134166	2306952	12660	1352	3168	1853	608	467	9	30670840	19628010	9248	1722	3564	2250	672
181	8	6134166	2358792	12360	1352	3168	1853	608	479	9	30670840	20649690	7008	1722	3564	2250	544
191	8	6134166	2626632	10800	1352	3168	1853	480	487	9	30670840	21345210	5504	1722	3564	2250	608
193	8	6134166	2681928	10485	1352	3168	1853	744	491	9	30670840	21697290	4768	1722	3564	2250	608
197	8	6134166	2794248	9855	1352	3168	1853	672	499	9	30670840	22410090	3264	1722	3564	2250	608
199	8	6134166	2851272	9540	1352	3168	1853	608	503	9	30670840	22770810	2512	1722	3564	2250	544
211	8	6134166	3205512	7680	1352	3168	1853	608	509	9	30670840	23317290	1392	1722	3564	2250	544
223	8	6134166	3580488	5805	1352	3168	1853	480	521	10	149946368	29858510	118782	1974	3960	2556	1134
227	8	6134166	3710088	5175	1352	3168	1853	608	—	—	—	—	—	—	—	—	—
229	8	6134166	3775752	4860	1352	3168	1853	608	—	—	—	—	—	—	—	—	—
233	8	6134166	3908808	4245	1352	3168	1853	608	—	—	—	—	—	—	—	—	—

APPENDIX C

FPGAs and Their Complexity Model

C.1 Virtex Architecture

This section gives a brief overview of the XILINX Virtex FPGA architecture which are the devices used in this thesis for prototyping. The XILINX Virtex family is the most used FPGA series in academia concerning cryptographic implementations [WGP03].

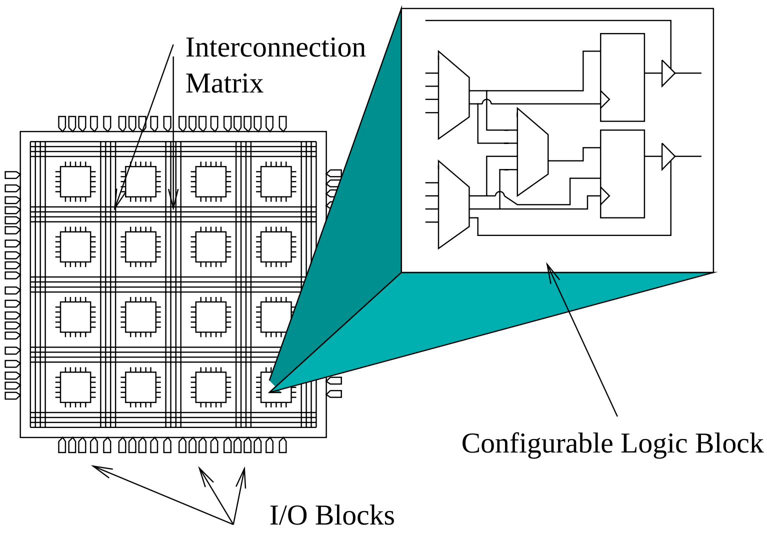


Figure C.1. Virtex FPGA

Three logic components are common in modern FPGAs: Configurable Logic Blocks (CLBs), interconnections, and I/O blocks as shown in Figure C.1. In addition, certain FPGA families and, in particular the Virtex family, contain embedded memory blocks. The I/O blocks of FPGAs are very similar to the I/O pads in an ASIC and act as buffers to the outside world. The CLBs are the core logic element in an FPGA. The main block in a Virtex CLB is the logic cell. Each Virtex CLB contains four logic cells organized in two similar slices. A logic cell includes a 4-input function generator, carry logic, and a storage element. The output from the function generator in each logic cell drives both the CLB output and the D-input of the flip-flop. The function generators in the slice can be thought of as 4-input look-up tables (LUTs). Each LUT can be configured to provide a 16×1-bit synchronous RAM or the two LUTs within a slice can be combined to create a 16×2-bit, 32×1-bit synchronous RAM, or a 16×1-bit dual-port synchronous RAM. The F5 multiplexer provides the ability to combine the function generator outputs, either to a function generator (implementing any 5-input function), to a 4:1 multiplexer, or to selected functions of up to nine inputs. The F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs. The XOR gate provides the possibility to implement a 1-bit full adder in one LC and the AND gate allows a efficient multiplier implementation.

In addition, large blocks of RAM memories which are organized in columns are provided. Virtex devices have two columns that extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Virtex device 64 CLBs high contains 16 memory blocks per column, and a total of 32 blocks. A summary of the number of system gates, CLBs, LC, available I/O and RAM for the Virtex devices considered in this thesis can be found in Table C.1.

Table C.1. Virtex FPGA Family Members

Device	System Gates	CLB Array	Logic Cells	Maximum I/O	Block RAM Bits	Maximum RAM Bits
$XCV1000$	$1,124,022$	64x96	$27,648$	512	$131,072$	$393,216$

C.2 Complexity Considerations for the Virtex FPGA

The main focus of this thesis is on providing architectures for $GF(p^m)$ fields, amenable to VLSI implementations. However, in order to validate our designs, we have in some cases implemented prototypes on FPGAs. To this end, we also provide size complexity estimates which we compare to the actual size of the circuit after synthesis by the FPGA tools. We do not provide delay estimates for several reasons, which include the lack of an accurate delay model for FPGAs as pointed out in [Orl02] and the fact that our implementation of the circuits on FPGAs is mainly for verification purposes rather than as target technology.

We have used extensively the area models for FPGAs introduced in [Orl02], since we have used similar architectures to the ones introduced in [Orl02] to implement the $GF(p^m)$ multipliers presented in this thesis and, more importantly, because the models in [Orl02] turn out to be remarkably accurate. Notice that the prototype implementations presented in this work correspond to fields $GF(2^n)$ and $GF(3^m)$, i.e., fields of characteristic two and three respectively. Thus, the models and assumptions set out in this section are specific to these two fields (although they can probably be easily generalized to other fields).

The following assumptions have been made in coming up with the FPGA area estimates throughout this thesis. We notice that the models from [Orl02] have been extended to account for the characteristic three case.

- A $GF(2)$ adder, subtracter, or multiplier requires one LUT.
- A $GF(3)$ adder, subtracter, or multiplier requires two LUTs, including adders that add weighted inputs, for example, adders that compute $(a_i * c) + (b_i * d)$ where c and d are fixed constants.
- A 2:1 multiplexer requires one LUT. A 2:1 multiplexer that handles m-bit inputs requires m 2:1 multiplexers and thus m LUTs
- A shift-register cell requires one LUT and one flip-flop (FF). Thus, an m-bit shift-register requires m LUTs and m flip-flops. Alternatively, an m-bit shift-register can be thought of as containing m 2:1 multiplexers and m registers.

- An m-bit register requires m flip-flops.
- The register estimates do not account for registers used to reduce the critical path delay of a multiplier through pipelining.

APPENDIX D

Standard Cell Library Data

Table D.1. Area and time complexities of different circuits for three different standard cell libraries.

Function	[GS03b, VLS03] 0.25 μm CMOS		[GS03b, VLS03] 0.18 μm CMOS	
	A (μm^2)	T (nsec)	A (μm^2)	T (nsec)
$\overline{A}$	36	0.9	16	0.7
$(A \wedge B)$	72	—	32	0.6
$\overline{(A \wedge B)}$	54	0.9	24	0.7
$\overline{(A \wedge B \wedge C)}$	81	1.0	36	0.7
$(A \vee B)$	72	0.8	32	0.5
$\overline{(A \vee B)}$	54	0.8	24	0.6
$\overline{(A \vee B \vee C)}$	144	0.8	64	0.6
$(A \oplus B)$	126	0.8	56	0.6
$\overline{(A \oplus B)}$	126	0.8	56	0.6
$\overline{((A \wedge B) \vee C)}$	72	0.8	32	0.6
$\overline{((A \wedge B) \vee (C \wedge D))}$	90	0.8	40	0.6
$\overline{((A \vee B) \wedge C)}$	51	0.9	23	0.7
$\overline{((A \vee B) \wedge (C \vee D))}$	90	0.9	40	0.7
D Flip-Flop	216	0.8	96	0.6
2:1 Multiplexer	108	0.8	48	0.6
Full Adder	270	1.0	120	0.7

Notes:

- The standard cell library from [GS03b, VLS03] has a half adder cell, but the definition of the half adder cell is different from the one that we have used throughout this work, thus we do not include it in Table D.1. Similarly, we have not included delay for the 2-input AND gate in [GS03b, VLS03] because in the documentation there is only delay data for an AND gate designed with an output driving strength which is twice as large as all other components included in Table D.1.

- Timings reported for the 0.25 μm CMOS library from [GS03b, VLS03] are worst case delay assuming a 0.3 pF load.
- Timings reported for the 0.18 μm CMOS library from [GS03b, VLS03] are worst case delay assuming a 0.15 pF load.

Table D.2. Normalized area and time complexities of different components in different standard cell libraries. Normalization done with respect to the area/delay of a 2-input NAND gate in the given library.

Component	[VLS03] 0.25 μm CMOS		[VLS03] 0.18 μm CMOS	
	A	T	A	T
Inverter	0.7	1.0	0.7	1.0
2-input AND gate	1.3	—	1.3	0.9
2-input NAND gate	1.0	1.0	1.0	1.0
3-input NAND gate	1.5	1.1	1.5	1.0
2-input OR gate	1.3	0.9	1.3	0.7
2-input NOR gate	1.0	0.9	1.0	0.9
3-input NOR gate	2.7	0.9	2.7	0.9
2-input XOR gate	2.3	0.9	2.3	0.9
2-input XNOR gate	2.3	0.9	2.3	0.9
Complex gate implementing $\overline{((A \wedge B) \vee C)}$	1.3	0.9	1.3	0.9
Complex gate implementing $\overline{((A \wedge B) \vee (C \wedge D))}$	1.7	0.9	1.7	0.9
Complex gate implementing $\overline{((A \vee B) \wedge C)}$	0.9	1.0	1.0	1.0
Complex gate implementing $\overline{((A \vee B) \wedge (C \vee D))}$	1.7	1.0	1.7	1.0
D Flip-Flop	4.0	0.9	4.0	0.9
2:1 Multiplexer	2.0	0.9	2.0	0.9
Full Adder	5.0	1.1	5.0	1.0

Bibliography

[ABMV93] G. Agnew, T. Beth, R. Mullin, and S. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6:3–13, 1993.

[Act01] Actel Corporation. *Actel's ProASIC Family, The Only ASIC Design Flow FPGA*, 2001.

[AFM89] K. Araki, I. Fujita, and M. Morisue. Fast inverters over finite field based on Euclid's algorithm. *Transactions of the IEICE*, E-72(11):1230–1234, November 1989.

[Alf99] P. Alfke. Xilinx M1 Timing Parameters. Electronic Mail Personal Correspondance, December 1999.

[Alt01] Altera Corporation. *APEX 20KC Programmable Logic Device Data Sheet*, 2001.

[AMOV91] G. B. Agnew, R. C. Mullin, I. M. Onyschuk, and S. A. Vanstone. An implemenation for a fast public-key cryptosystem. *Journal of Cryptology*, 3:63–79, 1991.

[AMV93] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected areas in Communications*, 11(5):804–813, June 1993.

[Ban74] D.K. Banerji. A Novel Implementation Method for Addition and Subtraction in Residue Number Systems. *IEEE Transactions on Computers*, pages 106–108, January 1974.

[Bar86] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume LNCS 263, pages 311–323, Berlin, Germany, August 1986. Springer-Verlag.

[BB04a] D. Boneh and X. Boyen. Short Signatures Without Random Oracles. In C. Cachin and J. Camenisch, editors, *Advances in Cryptology — EUROCRYPT 2004*, volume LNCS 3027, pages 56–73. Springer-Verlag, 2004.

[BB04b] D. Boneh and X. Boyen. Short Signatures Without Random Oracles. In M. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume LNCS 3152, pages 443–459. Springer-Verlag, 2004.

[BBS01] D. Boneh, B.Lynn, and H. Shacham. Short signatures from the Weil pairing. In J. Boyd, editor, *Advances in Cryptology — Asiacrypt 2001*, volume LNCS 2148, pages 514–532. Springer-Verlag, 2001.

[BCH93] H. Brunner, A. Curiger, and M. Hofstetter. On Computing Multiplicative Inverses in $GF(2^m)$. *IEEE Transactions on Computers*, 42(8):1010–1015, August 1993.

[BD95] J. Benaloh and W. Dai. Fast modular reduction. Rump session of CRYPTO '95, 1995.

[BF01] D. Boneh and M. Franklin. Identity-Based Encryption from the Weil Pairing. In J. Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume LNCS 2139, pages 213–229. Springer-Verlag, 2001.

[BGK$^+$03] G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger. Efficient $GF(p^m)$ Arithmetic Architectures for Cryptographic Applications. In M. Joye, editor, *Topics in Cryptology — CT-RSA 2003*, volume LNCS 2612, pages 158–175. Springer-Verlag, April 13-17, 2003.

[BGL93] Blake, Gao, and Lambert. Constructive problems for irreducible polynomials over finite fields. In T.A. Gulliver and N.P. Secord, editors, *Information Theory and Applications*, volume LNCS 793, pages 1–23. Springer-Verlag, 1993.

[BGM93] I. F. Blake, S. Gao, and R. C. Mullin. Explicit factorization of $x^{2^k}+1$ over f_p with prime $p \equiv 3 \bmod 4$. *Applicable Algebra in Engineering, Communication, and Computation*, 4:89–94, 1993.

[BGO03] G. Bertoni, J. Guajardo, and G. Orlando. Systolic and Scalable Architectures for Digit-Serial Multiplication in Fields $GF(p^m)$. In T. Johansson and S. Maitra, editors, *Progress in Cryptology — INDOCRYPT 2003*, volume LNCS 2904, pages 349–362. Springer-Verlag, December 8-10, 2003.

[BHH$^+$82] R.K. Brayton, G.D. Hachtel, L. Hemanchandra, R. Newton, and A. Sangiovanni-Vincentelli. A Comparison of Logic Minimization Strategies Using ESPRESSO: An APL Program Package for Partitioned Logic Minimization. In *Proceedings of the International Symposium on Circuits and Systems*, pages 42–48, April 1982. Latest version of the ESPRESSO software is available from http://www-cad.eecs.berkeley.edu/software.html.

[BHMS84] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni. ESPRESSO-II: A New Logic Minimizer for Programmable Logic Arrays. In *Proc. 1984 Custom Integrated Circuits Conference*, May 1984.

[BINP03] J.-C. Bajard, L. Imbert, C. Nègre, and T. Plantard. Efficient Multiplication in $GF(p^k)$ for Elliptic Curve Cryptography. In J.-C. Bajard and M. Schulte, editors, *Proceedigns of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 181–187, June 15-18, 2003.

[BJM87a] M.A. Bayoumi, G.A. Jullien, and W.C. Miller. A Look-Up Table VLSI Design Methodology for RNS Structures Used in DSP Applications. *IEEE Transactions on Circuit and Systems*, CAS-34(6):604–616, June 1987.

[BJM87b] M.A. Bayoumi, G.A. Jullien, and W.C. Miller. A VLSI Implementation of Residue Adders. *IEEE Transactions on Circuit and Systems*, CAS-34(3):284–288, March 1987.

[BJS94] S. Bandyopadhyay, G.A. Jullien, and A. Sengupta. A Fast VLSI Systolic Array for Large Modulus Residue Addition. *Journal of VLSI Signal Processing*, 8:305–318, 1994.

[BKLS02] P. S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient Algorithms for Pairing-Based Cryptosystems. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, volume LNCS 2442, pages 354–368. Springer-Verlag, 2002.

[Bla83] G. R. Blakley. A computer algorithm for the product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.

[Blu99] T. Blum. Modular exponentiation on reconfigurable hardware. Master's thesis, ECE Department, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, May 1999.

[BP98] D. V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume LNCS 1462, pages 472–485, Berlin, Germany, 1998. Springer-Verlag.

[BP99] T. Blum and C. Paar. Montgomery modular multiplication on reconfigurable hardware. In *Proceedigns of the 14th IEEE Symposium on Computer Arithmetic (ARITH-14)*, pages 70–77, 1999.

[BP01a] D. V. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[BP01b] T. Blum and C. Paar. High radix Montgomery modular exponentiation on reconfigurable hardware. *IEEE Transactions on Computers*, 50(7):759–764, July 2001.

[Bri82] E. F. Brickell. A fast modular multiplication algorithm with applications to two key cryptography. In D. Chaum and R. L. Rivest and A. T. Sherman, editor, *Advances in Cryptology — CRYPTO '82*, pages 51–60, New York, USA, 1982. Plenum Publishing.

[Cel02] Celoxica. *Celoxica Handel-C Language Overview*, 2002. Manual available from http://www.celoxica.com.

[CHCW99] C.Y.Su, S.A. Hwang, P.S. Chen, and C.W. Wu. An improved Montgomery's algorithm for high-speed RSA public-key cryptosystems. *IEEE Transactions on VLSI Systems*, 7(2):282–286, 1999.

[cKKS98] Ç. K. Koç 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.

[CPO95] E. D. Di Claudio, F. Piazza, and G. Orlandi. Fast Combinatorial RNS Processors for DSP Applications. *IEEE Transactions on Computers*, 44(5):624–633, May 1995.

[CQS01] M. Ciet, J.-J. Quisquater, and F. Sica. A Secure Family of Composite Finite Fields Suitable for Fast Implementation of Elliptic Curve Cryptography. In C.P. Rangan and C. Ding, editors, *Progress in Cryptology — Indocrypt 2001*, volume LNCS 2247, pages 108–116, Berlin, December 16-20, 2001. Springer-Verlag.

[CSL00] Jae Wook Chung, Sang Gyoo Sim, and Pil Joong Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 57–70, Berlin, August 17-18, 2000. Springer-Verlag.

[DBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In Kim, K. and Matsumoto, T., editor, *Advances in Cryptology — Asiacrypt '96*, volume LNCS 1233, pages 65–76. Springer-Verlag, 1996.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[Dhe94] J.-F. Dhem. Modified version of the Barret modular multiplication algorithm. UCL Technical Report CG-1994/1, Université catholique de Louvain, July 18, 1994.

[Dhe98] J.-F. Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards*. PhD thesis, UCL — Université catholique de Louvain, Louvain-la-Neuve, Belgium, May 1998.

[Dif99] W. Diffie. Subject: Authenticity of Non-Secret Encryption documents. World Wide Web, October 6, 1999. Email message sent to John Young. Available at http://cryptome.org/ukpk-diffie.htm.

[DJQ97] J.-F. Dhem, M. Joye, and J.-J. Quisquater. Normalisation in diminished-radix modulus transform. *Electronic Letters*, 33(23):1931, 1997.

[DK90] S. R. Dussé and B. S. Kaliski. A Cryptographic Library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume LNCS 473, pages 230–244, Berlin, Germany, May 1990. Springer-Verlag.

[DQ90] D. De Waleffe and J.-J. Quisquater. CORSAIR: A smart card for public key cryptosystems. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90*, volume LNCS 537, pages 502–514, Berlin, 1990. Springer-Verlag.

[Dug92] M. Dugdale. VLSI Implementeation of Residue Adders Based on Binary Adders. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(5):325–329, May 1992.

[Dug94] M. Dugdale. Residue Multipliers Using Factored Decomposition. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 41(9):623–627, September 1994.

[EB90] K. M. Elleithy and M. A. Bayoumi. A $\Theta(1)$ Algorithm for Modulo Addition. *IEEE Transactions on Circuits and Systems*, 37(5):628–631, May 1990.

[Ell97] J. H. Ellis. The story of non-secret encryption. Available at http://jya.com/ellisdoc.htm, December 16th, 1997.

[EW93] S. E. Eldridge and C. D. Walter. Hardware implementation of Montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, July 1993.

[FBT96] S. T. J. Fenn, M. Benaissa, and D. Taylor. $GF(2^m)$ multiplication and division over the dual base. *IEEE Transactions on Computers*, 45(3):319–327, March 1996.

[Fen89] G. L. Feng. A VLSI Architecture for Fast Inversion in $GF(2^m)$. *IEEE Transactions on Computers*, C-38(9):1989, October 1989.

[FMM+96] R. Ferreira, R. Malzahn, P. Marissen, J.-J. Quisquater, and T. Wille. FAME: A 3rd generation coprocessor for optimising public key cryptosystems in smart card applications. In P. H. Hartel, P. Paradinas, and J.-J. Quisquater, editors, *Proceedings of CARDIS 1996, Smart Card Research and Advanced Applications*, pages 59–72, CWI, Amsterdam, The Netherlands, September 16–18, 1996. Stichting Mathematisch Centrum.

[FP00] W.L. Frecking and K. K. Parhi. Performance-Scalable Array Architectures for Modular Multiplication. In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors — ASAP'00*, pages 149–162, July 10 - 12, 2000.

[Gei93] W. Geiselmann. *Algebraische Algorithmenentwicklung am Beispiel der Arithmetik in Endlichen Körpern*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, Institut für Algorithmen und Kognitive Systeme, Karlsruhe, Germany, 1993.

[GG90] W. Geiselmann and D. Gollmann. VLSI design for exponentiation in $GF(2^n)$. In J. Seberry and J. Pieprzyk, editors, *Advances in Cryptology — AUSCRYPT '90*, volume LNCS 453, pages 398–405, Sydney, Australia, January 1990. Springer-Verlag.

[GHS02a] S. D. Galbraith, K. Harrison, and D. Soldera. Implementing the Tate Pairing. In C. Fieker and D. Kohel, editors, *Algorithmic Number Theory — ANTS-V*, volume LNCS 2369, pages 324–337. Springer-Verlag, 2002.

[GHS02b] P. Gaudry, F. Hess, and N. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. *Journal of Cryptology*, 15(1):19–46, January 2002.

[GL92] S. Gao and H. W. Lenstra, Jr. Optimal normal bases. *Designs, Codes and Cryptography*, 2:315–323, 1992.

[Gol67] S.W. Golomb. *Shift Register Sequences*. Holden-Day, San Francisco, USA, 1967.

[Gor98] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.

[GP97] J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In B. Kaliski, Jr., editor, *Advances in Cryptology — CRYPTO '97*, volume LNCS 1294, pages 342–356, Berlin, Germany, August 1997. Springer-Verlag.

[GP02] J. Guajardo and C. Paar. Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes. *Design, Codes, and Cryptography*, 25(2):207–216, February 2002.

[GS03a] W. Geiselmann and R. Steinwandt. A Redundant Representation of $GF(q^n)$ for Designing Arithmetic Circuits. *IEEE Transactions on Computers*, 52(7):848–853, July 2003.

[GS03b] J. Grad and J. E. Stine. A Standard Cell Library for Student Projects. In *International Conference on Microelectronic Systems Education*, pages 98–99. IEEE Computer Society, 2003.

[Gua03] J. Guajardo. Itoh-Tsujii Inversion Algorithm. In H.C.A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Kluwer Academic Publishers, 2003. To appear.

[GV95] S. Gao and S.A. Vanstone. On Orders of Optimal Basis Generators. *Mathematics of Computation*, 64(211):1227–1233, July 1995.

[GW98a] J.-H. Guo and C.-L. Wang. Hardware-efficient systolic architecture for inversion and division in $GF(2^m)$. *IEE Proceedings — Computers and Digital Techniques*, 145(4):272–278, July 1998.

[GW98b] 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*, 145(4):272–278, July 1998.

[GWP02] J. Guajardo, T. Wollinger, and C. Paar. Area Efficient $GF(p)$ Architectures for $GF(p^m)$ Multipliers. In *Proceedings of the 45th IEEE International Midwest Symposium on Circuits and Systems — MWSCAS 2002*, August 2002.

[Has92] M. A. Hasan. *Efficient Computations in Galois Fields*. PhD thesis, Department of ECE, University of Victoria, Canada, April 1992.

[HHM00] D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Ç. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 1–24, Berlin, August 17-18, 2000. Springer-Verlag.

[Hia96] A. A. Hiasat. Semi-Custom VLSI Design for RNS Multipliers Using Combinatorial Logic Approach. In *Third IEEE International Conference on Electronics, Circuits, and Systems — ICECS '96*, pages 935–938, October 13–16, 1996.

[Hia02] A. A. Hiasat. High-Speed and Residue-Area Modular Adder Structures for RNS. *IEEE Transactions on Computers*, 51(1):84–89, January 2002.

[HMV92] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT '92*, volume LNCS 658, pages 163–173, Berlin, Germany, May 1992. Springer-Verlag.

[HTDR88] I. S. Hsu, T. K. Truong, L. J. Deutsch, and I. S. Reed. A comparison of VLSI architecture of finite field multipliers using dual-, normal-, or standard bases. *IEEE Transactions on Computers*, 37(6):735–739, June 1988.

[HWB92] M. A. Hasan, M. Wang, and V. K. Bhargava. Modular construction of low complexity parallel multipliers for a class of finite fields $GF(2^m)$. *IEEE Transactions on Computers*, 41(8):962–971, August 1992.

[IT88] T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Information and Computation*, 78:171–177, 1988.

[IT89] T. Itoh and S. Tsujii. Structure of parallel multipliers for a class of fields $GF(2^k)$. *Inform. and Comp.*, 83:21–40, 1989.

[Ito91] T. Itoh. Characterization for a family of infinitely many irreducible equally spaced polynomials. *Information Processing Letters*, 37(5):273–277, March 1991.

[JB92] Y. Jeong and W. Burleson. Choosing VLSI algorithms for finite field arithmetic. In *IEEE Symposium on Circuits and Systems*, pages 799–802. IEEE, Inc., 1992.

[JB97] Y.J. Jeong and W.P. Burleson. VLSI array algorithms and architectures for RSA modular multiplication. *IEEE Transactions on VLSI Systems*, 5(2):211–217, 1997.

[JL77] W.K. Jenkins and B.J. Leon. The Use of Residue Number Systems in the Design of Finite Impulse Response Digital Filters. *IEEE Transactions on Circuits and Systems*, CAS-24(4):191–201, April 1977.

[Jou00] A. Joux. A one-round protocol for tripartite Diffie-Hellman. In W. Bosma, editor, *Algorithmic Number Theory — ANTS-IV*, volume LNCS 1838, pages 385–394. Springer-Verlag, 2000.

[Jul78] G. A. Jullien. Residue number scaling and other operations using ROM arrays. *IEEE Transactions on Computers*, C-27:325–337, April 1978.

[Jul80] G. A. Jullien. Implementation of Multiplication, Modulo a Prime Number, with Applications to Number Theoretic Transforms. *IEEE Transactions on Computers*, C-29(10):899–905, October 1980.

[Jun93] D. Jungnickel. *Finite Fields*. B.I.-Wissenschaftsverlag, Mannheim, Pennsylvania, USA, 1993.

[KH91] Ç. K. Koç and C. Y. Hung. Bit-level systolic arrays for modular multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.

[KMKH99] T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino. Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic. In *Advances in Cryptology — EUROCRYPT '99*, Berlin, Germany, 1999. Springer-Verlag.

[Knu81] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1981.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl. (English translation)*, 7(7):595–596, 1963.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Kob89] N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):129–150, 1989.

[Kob98] N. Koblitz. An elliptic curve implementation of the finite field digital signature algorithm. In H. Krawczyk, editor, *Advances in Cryptology — CRYPTO 98*, volume LNCS 1462, pages 327–337. Springer-Verlag, 1998.

[Kob00] T. Kobayashi. Base–ϕ Method for Elliptic Curves over OEF. *IEICE Transaction on Fundamentals of Electronics, Communications and Computer Sciences*, E83-A(4):679–686, April 2000.

[Kor93] I. Koren. *Computer Arithmetic Architectures*. Prentice-Hall, 1993.

[Kor94] P. Kornerup. A systolic, linear-array multiplier for a class of right-shift algorithms. *IEEE Transactions on Computers*, 43(8):892–898, August 1994.

[Kwo03] S. Kwon. A Low Complexity and a Low Latency Bit Parallel Systolic Multiplier over GF(2m) Using an Optimal Normal Basis of Type II. In J.-C. Bajard and M. Schulte, editors, *Proceedigns of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 196–203, June 15-18, 2003.

[LD00] J. López and R. Dahab. High-Speed Software Multiplication in F_{2^m}. In — *Indocrypt 2000*, volume LNCS, Berlin, 2000. Springer-Verlag.

[LKL98] E. Lee, D. Kim, and P. Lee. Speed-up of F_{p^m} Arithmetic for Elliptic Curve Cryptosystems. In *Proceedings of ICICS '98*, Berlin, Germany, 1998. Springer Verlag, Lecture Notes in Computer Science.

[LLL01] C.-Y. Lee, E.-H. Lu, and J.Y. Lee. Bit-Parallel Systolic Multipliers for $GF(2^m)$ Fields Defined by All-One and Equally Spaced Polynomials. *IEEE Transactions on Computers*, 50(5):385–393, May 2001.

[LN97] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, Great Britain, second edition, 1997.

[Loi00] P. Loidreau. On the Factorization of Trinomials over F_3. Rapport de recherche no. 3918, INRIA, April 2000.

[LV00] A. Lenstra and E. Verheul. The XTR Public-Key Cryptosystem. In M. Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, volume LNCS 1423, pages 1–19, Berlin Heidelberg, 2000. Springer-Verlag.

[Mas89] E. D. Mastrovito. VLSI design for multiplication over finite fields $GF(2^m)$. In *Lecture Notes in Computer Science 357*, pages 297–309, Berlin, Germany, March 1989. Springer-Verlag.

[Mas91] E. D. Mastrovito. *VLSI Architectures for Computation in Galois Fields*. PhD thesis, Linköping University, Department of Electrical Engineering, Linköping, Sweden, 1991.

[McE89] R. J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 2nd edition, 1989.

[Men93] A. J. Menezes, editor. *Applications of Finite Fields*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1993.

[MGM99] M.N. Mahesh, S. Gupta, and M. Mehendale. Improving Area Efficiency of Residue Number System based Implementation of DSP Algorithms. In *Proceedings of the International Conference on VLSI Design (VLSID)*, Goa, India, 1999.

[Mih97] P. Mihăilescu. Optimal Galois Field Bases which are not Normal. Recent Results Session — FSE '97, 1997.

[Mil86] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, volume LNCS 218, pages 417–426, Berlin, Germany, 1986. Springer-Verlag.

[MJ99] A. Menezes and D. Johnson. The elliptic curve digitial signature algorithm (ECDSA). Technical report CORR 99-34, Department of C $\&$ O, University of Waterloo, Ontario, Canada, August 1999.

[MK00] S. Morioka and Y. Katayama. $O(\log_2 m)$ Iterative Algorithm for Multiplicative Inversion in $GF(2^m)$. In *Proceedings of the 2000 IEEE International Symposium on Information Theory*, page 449, June 25-30, 2000.

[MKW89] M. Morii, M. Kasahara, and D.L. Whiting. Efficient bit-serial multiplication and discrete-time Wiener-Hoph equation over finite fields. *IEEE Transactions on Information Theory*, IT-35:1177–1184, November 1989.

[MMT02] M. Maurer, A. Menezes, and E. Teske. ANALYSIS OF THE GHS WEIL DESCENT ATTACK ON THE ECDLP OVER CHARACTERISTIC TWO FINITE FIELDS OF COMPOSITE DEGREE. *LMS Journal of Computation and Mathematics*, 5:127–174, November 2002.

[MO86] J. L. Massey and J. K. Omura. Computational method and apparatus for finite field arithmetic. United States Patent, Patent Number 4587627, May 6, 1986. Filed September 14, 1982.

[Mon85] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[MOVW89] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics, North Holland*, 22:149–161, 1988/89.

[MQ01] A. Menezes and M. Qu. Analysis of the Weil Descent Attack of Gaudry, Hess and Smart. In D. Naccache, editor, *Topics in Cryptology - CT-RSA 2001*, volume LNCS 2020, pages 308–318, Berlin, April 8-12 2001. Springer-Verlag.

[MS02] S. Morioka and A. Satoh. An Optimized S-Box Circuit Architecture for Low Power AES Design. In B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2002*, volume LNCS 2523, pages 172–186. Springer-Verlag, 2002.

[Mül01] V. Müller. Efficient Point Multiplication for Elliptic Curves over Special Optimal Extension Fields. In Walter de Gruyter, editor, *Public-key Cryptography and Computational Number Theory*, pages 197–207, September 2001.

[MV93] A. Menezes and S. Vanstone. Elliptic curve cryptosystems and their implementation. *Journal of Cryptology*, 6:209–224, 1993.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, USA, 1997.

[Nak62] A. Nakl, editor. *Decimal Arithmetic Unit*. Stroje Na Zprocovani, Prague, CSAU, Czechoslovakia, 1962.

[NIO96] C. Nagendra, M.J. Irwin, and R.M. Owens. Area-Time-Power Tradeoffs in Parallel Adders. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 43(10):689–702, October 1996.

[NM96] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, 1996.

[NS81] M. J. Norris and G. J. Simmons. Algorithms for high-speed modular arithmetic. *Congressus Numeratium*, 31:153–163, 1981.

[OK91] H. Orup and P. Kornerup. A High-Radix Hardware Algorithm for Calculating the Exponential M^E Modulo N. In P. Kornerup and D. W. Matula, editors, *Proceedigns of the 10th IEEE Symposium on Computer Arithmetic (ARITH 10)*, pages 51–56, 1991.

[OKPK99] Jin Young Oo, Young-Gern Kim, Dong-Young Park, and Heung-Su Kim. Efficient Multiplier Architecture Using Optimized Irreducible Polynomial over $GF((3^n)^3)$. In *Proceedings of the IEEE Region 10 Conference – TENCON 99. "Multimedia Technology for Asia-Pacific Information Infrastructure"*, volume 1, pages 383–386, 1999.

[Olo02] M. Olofsson. *VLSI Aspects on Inversion in Finite Fields*. PhD thesis, Department of Electrical Engineering — Linköpings universitet, Linköpings, Sweden, 2002.

[Omu90] J. K. Omura. A public key cell design for smart card chips. In *International Symposium on Information Theory and its Applications*, pages 983–985, USA, November 27-30, 1990.

[OP00] G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. In Ç. K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 41–56. Springer-Verlag, August 17-18, 2000.

[Orl02] G. Orlando. *Efficient Elliptic Curve Processor Architectures for Field Programmable Logic*. PhD thesis, Dept. of ECE, Worcester Polytechnic Institute, March 2002.

[Oru95] H. Orup. Simplifying quotient determination in high-radix modular multiplication. In *Proceedigns of the 12th IEEE Symposium on Computer Arithmetic (ARITH 12)*, pages 193–9, 1995.

[P1300] *IEEE P1363-2000: IEEE Standard Specifications* *for Public Key Cryptography*, 2000. Available at http://standards.ieee.org/catalog/olis/busarch.html.

[Paa94] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, (Engl. transl.), Institute for Experimental Mathematics, University of Essen, Essen, Germany, June 1994. ISBN 3–18–332810–0.

[Paa96] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[Par99] B. Parhami. *COMPUTER ARITHMETIC — Algorithms and Hardware Designs*. Oxford University Press, New York, USA, 1999.

[PB95] M. G. Parker and M. Benaissa. $GF(p^m)$ Multiplication Using Polynomial Residue Number Systems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(11):718–721, November 1995.

[PFR99] C. Paar, P. Fleischmann, and P. Soria Rodriguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, October 1999.

[Pie94] S.J. Piestrak. Design of High-Speed Residue-to-Binary Number System Converter Based on Chinese Remainder Theorem. In *Proceedings of the International Conference on Computer Design — ICCD'94*, pages 508–511. IEEE Computer Society Press, October 14-16, 1994.

[PKS01] V. Paliouras, K. Karagianni, and T. Stouraitis. A Low-Complexity Combinatorial RNS Multiplier. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 48(7):675–683, July 2001.

[PL95] C. Paar and N. Lange. A comparative VLSI synthesis of finite field multipliers. In *3rd International Symposium on Communication Theory and its Applications*, Lake District, UK, July 10–14 1995.

[Pol71] J.M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25:365–374, April 1971.

[PS95] P.Smith and C. Skinner. A public-key cryptosystem and a digital signature system based on the lucas function analogue to discrete logarithms. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptology — ASIACRYPT'94*, volume LNCS 917, pages 357–364, Berlin, 1995. Springer-Verlag.

[PS02] D. Page and N. P. Smart. Hardware implementation of finite fields of characteristic three. In B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2002*, volume LNCS 2523, pages 529–539. Springer-Verlag, 2002.

[PSW02] M.R. Pillmeier, M.J. Schulte, and E.G. Walters III. Design alternatives for barrel shifters. In F.T. Luk, editor, *Proceedings of SPIE — Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, volume 4791, pages 436–447, Seattle, Washington, USA, July, 2002. SPIE - The International Society for Optical Engineering.

[Qui90] J.-J. Quisquater. Fast modular exponentiation without division. Rump session of EUROCRYPT '90, 1990.

[Qui92] J.-J. Quisquater. Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system. United States Patent, Patent Number 5166978, November 24 1992.

[RMH02a] A. Reyhani-Masoleh and M.A. Hasan. A New Construction of Massey-Omura Parallel Multiplier over $GF(2^m)$. *IEEE Transactions on Computers*, 51(5):511–520, May 2002.

[RMH02b] A. Reyhani-Masoleh and M.A. Hasan. Efficient Digit-Serial Normal Basis Multipliers over $GF(2^M)$. In *IEEE International Symposium on Circuits and Systems — ISCAS 2002*, pages 781–784, May 2002.

[RMH03a] A. Reyhani-Masoleh and M.A. Hasan. Efficient Multiplication Beyond Optimal Normal

Bases. *IEEE Transactions on Computers. Special Section on Cryptographic Hardware and Embedded Systems*, 52(4):428–439, April 2003.

[RMH03b] A. Reyhani-Masoleh and M.A. Hasan. Low Complexity Sequential Normal Basis Multipliers over $GF(2^m)$. In J.-C. Bajard and M. Schulte, editors, *Proceedigns of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 188–195, June 15-18, 2003.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[RY92] D. Radhakrishnan and Y. Yuan. Novel Approaches to the Design of VLSI RNS Multipliers. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 39(1):52–57, January 1992.

[Sch98] B. Schneier. Crypto-Gram Newsletter. World Wide Web, May 15, 1998. Available at http://www.schneier.com/crypto-gram-9805.html.

[ScKK01] B. Sunar and Ç. K. Koç. An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Transactions on Computers*, 50(1):83–87, January 2001.

[Sed87] H. Sedlak. The rsa cryptography processor. In D. Chaum and W. L. Price, editors, *Advances in Cryptology – EUROCRYPT ’87*, volume LNCS 304, pages 95–105, Berlin, Germany, 1987. Springer-Verlag.

[SF77] M.A. Soderstrand and E.L. Fields. Multipliers and residue number arithmetic digital filters. *Electronic Letters*, 13(6):164–166, March 1977.

[Sho01] V. Shoup. NTL: A Library for doing Number Theory, version 5.1, July 19th, 2001. Available at http://www.shoup.net/ntl/.

[Slo85] K. R. Sloan, Jr. Comments on “A computer algorithm for the product AB modulo M”. *IEEE Transactions on Computers*, 34(3):290–292, March 1985.

[Sma99] N. Smart. Elliptic Curve Cryptosystems over Small Fields of Odd Characteristic. *Journal of Cryptology*, 12(2):141–151, Spring 1999.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO '95*, volume LNCS 963, pages 43–56, Berlin, Germany, 1995. Springer-Verlag.

[SP98] L. Song and K. K. Parhi. Low energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing*, 19(2):149–166, June 1998.

[SPSG97] D. J. Soudris, V. Paliouras, T. Stouraitis, and C. E. Goutis. A VLSI Design Methodology for RNS Full Adder-Based Inner Product Architectures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 44(4):315–318, April 1997.

[SSL$^+$92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgay, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, May 4th 1992.

[ST67] N. Szabó and R. Tanaka. *Residue Arithmetic and its Applications to Computer Technology*. McGraw-Hill, New York, 1967.

[ST91] A. Skavantzos and F.J. Taylor. On the polynomial residue number system. *IEEE Transactions on Signal Processing*, 39:376–382, February 1991.

[STcKK00] E. Savaş, A. F. Tenca, and Ç K. Koç. A Scalable and Unified Multiplier Architecture for Finite Fields $GF(p)$ and $GF(2^m)$. In Ç K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, volume LNCS 1965, pages 277–292, Berlin, Germany, August 17-18, 2000. Springer-Verlag.

[Ste67] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1:397–405, 1967.

[SV93] M. Shand and J. Vuillemin. Fast implementations of RSA cryptography. In E. Swartzlander, Jr., M. J. Irwin, and G. Jullien, editors, *Proceedigns of the 11th IEEE Symposium on Computer Arithmetic (ARITH-11)*, pages 252–259, 1993.

[Swa62] R.G. Swan. Factorization of trinomials over finite fields. *Pacific Journal of Mathematics*, 12:1099–1106, 1962.

[Tak92] N. Takagi. A radix-4 modular multiplication hardware algorithm for modular exponentiation. *IEEE Transactions on Computers*, 41(8), 1992.

[Tay84] F. Taylor. Residue Arithmetic: A tutorial with examples. *IEEE Transactions on Computers*, C-32:50–62, May 1984.

[TcKK99] A. F. Tenca and Ç K. Koç. A Scalable Architecture for Montgomery Multiplication. In Ç K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES'99*, volume LNCS 1717, pages 94–108, Berlin, Germany, August 12-13, 1999. Springer-Verlag.

[TcKK01] A. F. Tenca and G. Todorov Ç K. Koç. High-Radix Design of a Scalable Modular Multiplier. In Ç K. Koç, D. Naccache, and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, volume LNCS 2162, pages 185–201, Berlin, Germany, May 14-16, 2001. Springer-Verlag.

[TSW00] W.C. Tsai, C.B. Shung, and S.J. Wang. Two systolic architectures for modular multiplication. *IEEE Transactions on VLSI Systems*, 8(1):103–110, 2000.

[U.S00] U.S. Department of Commerce/National Institute of Standard and Technology. *FIPS 186-2, Digital Signature Standard (DSS)*, February 2000. Available at http://csrc.nist.gov/encryption.

[U.S01] U.S. Department of Commerce/National Institute of Standard and Technology. *FIPS PUB 197, Specification for the Advanced Encryption Standard (AES)*, November 2001. Available at http://csrc.nist.gov/encryption/aes.

[Ver01] E. Verheul. Self-blindable Credential Certificates from the Weil Pairing. In C. Boyd, editor, *Advances in Cryptology — Asiacrypt 2001*, volume LNCS 2248, pages 533–551. Springer-Verlag, 2001.

[Vis97] U. Vishne. Factorization of Trinomials over Galois Fields of Characterisitic 2. *Finite Fields and Their Applications*, 3(4):370–377, 1997.

[VLS03] VLSI Computer Architecture, Arithmetic, and CAD Research Group — Department of Electrical and Computer Engineering, IIT, Chicago, IL 60616, USA. *IIT Standard Cells for AMI 0.5 μm and TSMC 0.25μm/0.18 μm (Version 1.6.0)*, 2003. Library and documentation available from http://www.ece.iit.edu/~cad/.

[VP73] V.A. Vyshysky and V.D. Petushchak. Algorithms for Determination of the Reciprocal of a Number in a Residue Class System. *Soviet Automatic Control*, 6(3):58–61, May 1973.

[vzG01] J. von zur Gathen. Irreducible Trinomials over Finite Fields. In B. Mourrain, editor, *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation — ISSAC2001*, pages 332–336. ACM Press, 2001.

[vzG03] J. von zur Gathen. Irreducible Trinomials over Finite Fields. *Mathematics of Computation*, posted on February 3, 2003. PII: S 0025-5718(03)01515-1 (to appear in print).

[vzGN00] J. von zur Gathen and M. Nöcker. Exponentiation in Finite Fields: Theory and Practice. In T. Mora and H. Mattson, editors, *Applied Algebra, Agebraic Algorithms and Error Correcting Codes — AAECC-12*, volume LNCS 1255, pages 88–113. Springer-Verlag, 2000.

[Wal91] C. D. Walter. Faster modular multiplication by operand scaling. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91*, volume LNCS 576, pages 313–323, Berlin, 1991. Springer-Verlag.

[Wal93] C. D. Walter. Systolic Modular Multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.

[Wal97] C. D. Walter. Space/time trade-offs for higher radix modular multiplication using repeated addition. *IEEE Transactions on Computers*, 46(2), 1997.

[WB90] M. Wang and I.F. Blake. Bit serial multiplication in finite fields. *SIAM Discrete Mathematics*, 3(1):140–148, 1990.

[WBP00] A. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *IFIP CARDIS 2000, Fourth Smart Card Research and Advanced Application Conference*, Bristol, UK, September 20–22 2000. Kluwer.

[WGP03] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State of the Art — Implementations and Attacks. *ACM Transactions on Embedded Systems*, 2003. To appear.

[WH98] H. Wu and M.A. Hasan. Low Complexity Bit-Parallel Multipliers for a Class of Finite Fields. *IEEE Transactions on Computers*, 47(8):883–887, August 1998.

[WHB02] H. Wu, M.A. Hasan, and I.F. Blake. Low Complexity Parallel Multiplier in F_{q^n} Over F_q. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 49(7):1009–1013, July 2002.

[WmWSH02] C.-H. Wu, C. m. Wu, M.-D. Shieh, and Y.-T. Hwang. Novel Algorithms and VLSI Design for Division over $GF(2^m)$. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E85-A(5):1129–1139, May 2002.

[WTS+85] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed. VLSI Architectures for Computing Multiplications and Inverses in $GF(2^m)$. *IEEE Transactions on Computers*, C-34:709–717, August 1985.

[WTT02] Y. Watanabe, N. Takagi, and K. Takagi. A VLSI Algorithm for Division in $GF(2^m)$ Bassed on Extended Binary GCD Algorithm. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E85-A(5):994–999, May 2002.

[Wu98] H. Wu. *Efficient Computations in Finite Fields with Cryptographic Significance*. PhD thesis, Department of Electrical Engineering — University of Waterloo, Waterloo, Ontario, Canada, 1998.

[WW84] P.K.S. Wah and M.Z. Wang. Realization and application of the Massey-Omura lock. In *Proc. International Zurich Seminar*, Switzerland, 1984.

[Xil96] Xilinx, Inc., San Jose, California, USA. *The Programmable Logic Data Book*, 1996.

[Xil00] Xilinx, Inc. *The Programmable Logic Data Book*, 2000.

[YP82] K. Yiu and K . Peterson. A single-chip VLSI implemenation of the discrete exponential public-key distribution system. *IBM Systems Journal*, 15(1):102–116, 1982.

[ZB68] N. Zierler and J. Brillhart. On Primitive Trinomials $(\mathrm{mod} 2)$. *Information and Control*, 13:541–554, 1968.

[ZB69] N. Zierler and J. Brillhart. On Primitive Trinomials $(\mathrm{mod} 2)$, II. *Information and Control*, 14:566–569, 1969.

[Zie69] N. Zierler. Primitive Trinomials Whose Degree is a Mersenne Exponent. *Information and Control*, 15:67–69, 1969.

[Zie70] N. Zierler. On $x^n + x + 1$ over $GF(2)$. *Information and Control*, 16:67–69, 1970.

Curriculum Vitae

Personal Data

Born on August 1st, 1973 in Caracas, Venezuela. Venezuelan Citizen.

Secondary Education

09.1985 - 13.07.1990	Colegio Champagnat, Caracas, Venezuela. Degree: High School Diploma.
09.1990 - 01.06.1991	South Kent School, South Kent, CT 06785, USA English and Science Language Courses.
08.1991 - 20.05.1995	Worcester Polytechnic Institute, Worcester, MA 01609, USA. Degree: B.S. in Electrical and Computer Engineering and Physics.
09.1995 - 24.05.1997	Worcester Polytechnic Institute, Worcester, MA 01609, USA. Degree: M.S. in Electrical and Computer Engineering.
09.1998 - 28.09.2001	Worcester Polytechnic Institute, Worcester, MA 01609, USA. Research assistant and Ph.D. candidate.

Professional Experience

08.1993 - 03.1995	Teaching assistant, WPI, Worcester, MA 01609, USA.
08.1994 - 05.1995	Senior tutor, WPI, Worcester, MA 01609, USA.
07.1995 - 08.1995	Instructor for the 1995 STRIVE program, WPI,Worcester, MA 01609, USA.
07.1995 - 06.1997	Research and teaching assistant, WPI, Worcester, MA 01609, USA.
05.1996 - 08.1996	Engineer intern, General Dynamics, Needham, MA 02494, USA.
08.1997 - 08.1998	Systems/Software engineer, Baltimore Technologies, Needham, MA 02494, USA.
05.2000 - 08.2000	Engineer intern, cv cryptovision gmbh, 45886 Gelsenkirchen, Germany.
09.1998 - 05.2001	Research assistant, WPI, Worcester, MA 01609, USA.
06.2001 - 08.2001	Research intern, RSA Laboratories, Bedford, MA 01730, USA.
10.2001 - 07.2003	Research assistant at the Lehrstuhl Kommunikationssicherheit, 44807 Bochum, Germany.
08.2003 - present	Member of the Technical Staff, Infineon Technologies AG, Secure Mobile Solutions Division, 81609 Munich, Germany.

Publications

Journals

- T. Wollinger, J. Guajardo, C. Paar, Security on FPGAs: State of the Art Implementations and Attacks, *ACM Transactions on Embedded Computing Systems — Special Issue on Embedded Systems Security*, 2004. To appear.

- J. Guajardo, C. Paar, Itoh-Tsujii Inversion in Standard Basis and Its Applications to Cryptography, *Design, Codes and Cryptography*, 25(2):207-216, February, 2002.

Book Chapters

- J. Guajardo, Itoh-Tsujii Inversion Algorithm, In H.C.A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*, Kluwer Academic Publishers, 2004. To appear.

- G. Bertoni, J. Guajardo, C. Paar, Architectures for Advanced Cryptographic Systems, In C. Bellettini and M. G. Fugini, editors, *Information Security Policies and Actions in Modern Integrated Systems*, Idea Group Publishing, 2004. To appear.

Conferences

- G. Bertoni, J. Guajardo, G. Orlando, Systolic and Scalable Architectures for Digit-Serial Multiplication in Fields $GF(p^m)$, In T. Johansson and S. Maitra (Eds.): *Progress in Cryptology — INDOCRYPT 2003*, volume LNCS 2904, pp. 349–362, New Delhi, India, December 8-10, 2003.

- J. Pelzl, T. Wollinger, J. Guajardo, C. Paar, Hyperelliptic Curve Cryptosystems: Closing the Performance Gap to Elliptic Curves. In C. D. Walter, Ç. K. Koç, and C. Paar(Eds.): *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2003*, volume LNCS 2779, pp. 351-365, Cologne, Germany, September 7-10, 2003.

- G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, T. Wollinger, Efficient $GF(p^m)$ Arithmetic Architectures for Cryptographic Applications, In Marc Joye (Ed.): *The Cryptographers' Track at the RSA Conference - CT-RSA 2003*, volume LNCS 2612, pp. 158-175, San Francisco, CA, USA, April 13-17, 2003.

- T. Wollinger, J. Guajardo, C. Paar, Cryptography in Embedded Systems: An Overview (Invited Paper), In *proceedings of the Embedded World 2003 Exhibition and Conference*, pp.735-744, Design & Elektronik, Nuernberg, Germany, February 18-20, 2003.

- J. Guajardo, T. Wollinger, C. Paar, Area Efficient $GF(p)$ Architectures for $GF(p^m)$ Multipliers. In proceedings of *the 45th IEEE International Midwest Symposium on Circuits and Systems*, Tulsa, Oklahoma, August 4-7, 2002.

- A. Juels, J. Guajardo, RSA Key Generation with Verifiable Randomness, In D. Naccache and P. Paillier (Eds.), *Fifth International Workshop on Practice and Theory in Public Key Cryptography — PKC 2002*, volume LNCS 2274, February 12-14 2002, Paris, France.

- J. Guajardo, R. Bluemel, U. Krieger, C. Paar, Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers, In Kwangjo Kim (Ed.), *Fourth International Workshop on Practice and Theory in Public Key Cryptography - PKC 2001*, volume LNCS 1992, February 13-15 2001, Cheju Island, Korea.

- T. Wollinger, M. Wang, J. Guajardo, C. Paar, How Well Are High-End DSPs Suited for the AES Algorithms?: AES Algorithms on the TMS320C6x DSP, *The Third Advance Encryption Standard (AES3) Candidate Conference*, April 13-14, 2000, New York, USA. (Also published as TI Application Report SPRA740, February 2001).

- D. V. Bailey, W. Cammack, J. Guajardo, C. Paar, Cryptography in Modern Communication Systems (Extended Abstract),Wireless Symposium, *TI DSPS FEST '99,* Houston, Texas, August 1999, USA.

- J. Guajardo, C. Paar Fast Inversion in Composite Galois Fields $GF((2^n)^m)$, *1998 IEEE International Symposium on Information Theory*, MIT, August 16-21, 1998, Cambridge, MA, USA.

- J. Guajardo, C. Paar, Efficient Algorithms for Elliptic Curve Cryptosystems, In Burton Kaliski, editor, *Advances in Cryptology - CRYPTO '97,* volume LNCS 1294, August 17-21 1997, Santa Barbara, CA, USA.

Other Manuscripts

- J. Bömer, J. Guajardo, V. Krummel, Provably Secure Masking of AES, Cryptology ePrint Archive, Report 2004/101. April 30th, 2004.

- J. Guajardo, C. Paar, A Modified Squaring Algorithm. 1998.

Patents

- C. Paar, J. Guajardo, Method and system for point multiplication in elliptic curve cryptosystems. US Patent No. 6,252,959. Issue Date: June 26, 2001.

Honors and Awards

- Scholarship received from the Venezuelan government in 1990 to study in the United States for five years. Selected among 3500 applicants.

- Second place 1995 ECE Senior Project Competition.

- 1995 Salisbury Prize Award given to the best senior projects in each academic department.

- Graduated with High Honors from WPI in 1995.

- GTE Fellowship to sponsor my second year of graduate studies at WPI. (Fall 1996-Spring 1997)

- USENIX Scholars Fellowship awarded for one year (9.2000 - 9.2001)

Professional Society Memberships and Activities

- Refereed Papers for: the Workshop on Cryptographic Hardware and Embedded Systems (CHES '99), CHES 2000, CHES 2001, CHES 2003, CHES 2004, the RSA Conference's Cryptographers' Track (CT-RSA 2003), the IEEE Transactions on Computers, the IEEE Transactions on Circuits and Systems I, IEE Proceedings: Computers and Digital Techniques, The Computer Journal, and International Journal of Computers and Applications.

- Member of the IACR (International Association for: Cryptographic Research), Tau Beta Pi (National Engineering Honor Society), Sigma Pi Sigma (Physics Honor Society), and Eta Kappa Nu (Electrical Engineering Honor Society - WPI Chapter President Fall 94-Spring 95)

Index