Elliptic Curve Cryptography for Constrained Devices

Dissertation
submitted to the Faculty of
Electrical Engineering and Information Technology
at the
Ruhr-University Bochum

for the
Degree of Doktor-Ingenieur

by

Sandeep S. Kumar
Bochum, Germany, June 2006

Author contact information:
kumar@crypto.rub.de
http://www.sandeepkumar.org

Thesis Advisor: Prof. Christof Paar
Thesis Reader: Prof. Ingrid Verbauwhede

to my parents for their love and support

Abstract

There is a re-emerging demand for low-end devices such as 8-bit processors, driven by needs for pervasive applications like sensor networks and RF-ID tags. Security in pervasive applications, however, has been a major concern for their widespread acceptance. Public-key cryptosystems (PKC) like RSA and DSA generally involve computation-intensive arithmetic operations with operand sizes of $1024 - 2048$ bits, making them impractical on such constrained devices.

Elliptic Curve Cryptography (ECC) which has emerged as a viable alternative is a favored public-key cryptosystem for embedded systems due to its small key size, smaller operand length, and comparably low arithmetic requirements. However, implementing full-size, standardized ECC on 8-bit processors is still a major challenge and normally considered to be impracticable for small devices which are constrained in memory and computational power.

The thesis at hand is a step towards showing the practicability of PKC and in particular ECC on constrained devices. We leverage the flexibility that ECC provides with the different choices for parameters and algorithms at different hierarchies of the implementation. First a secure key exchange using PKC on a low-end wireless device with the computational power of a widely used 8-bit 8051 processor is presented. An *Elliptic Curve Diffie-Hellman* (ECDH) protocol is implemented over 131-bit Optimal Extension Field (OEF) purely in software. A secure end-to-end connection in an acceptable time of 3 seconds is shown to be possible on such constrained devices without requiring a cryptographic coprocessor.

We also investigate the potential of software/hardware co-design for architectural enhancements including instruction set extensions for low-level arithmetic used in ECC, most notably to speed-up multiplication in the finite fields. We show that a standard compliant 163-bit point multiplication can be computed in 0.113 sec on an 8-bit AVR micro-controller running at 4 Mhz (a typical representative for a low-cost pervasive processor) with minimal additional hardware extensions. Our design not only accelerates the computation by a factor of more than 30 compared to a software-only solution, it also reduces the code-size and data-RAM. Two new custom instructions for the MIPS 32-bit processor architecture are also proposed to accelerate the reduction modulo a pseudo Mersenne prime. We also show that the efficiency of multiplication in an OEF can be improved by a modified multiply and accumulate unit with a wider accumulator. The proposed architectural enhancements achieve a speed-up factor of 1.8 on the MIPS processor.

In addition, different architectural enhancements and optimal digit-size choices for the Least Significant Digit (LSD) multiplier for binary fields are presented. The two different architectures, the Double Accumulator Multiplier (DAM) and N-Accumulator Multiplier (NAM) are both faster compared to traditional LSD multipliers.

Later, an area/time efficient ECC processor architecture (for the OEFs of size 169, 289 and 361 bits) which performs all finite field arithmetic operations in the discrete Fourier domain is described. We show that a small optimized implementation of ECC processor with 24k equivalent gates on a 0.35um CMOS process can be realized for 169-bit curve using the novel multiplier design. Finally we also present a highly area optimized ASIC implementation of the ECC processor for various standard compliant binary curves ranging from $133 - 193$ bits. An area between 10k and 18k gates on a 0.35um CMOS process is possible for the different curves which makes the design very attractive for enabling ECC in constrained devices.

Kurzdarstellung

Aufgrund der allgegenwärtigen Präsenz von eingebetteten Systemen, z.B. basierend auf Sensornetzwerken und RF-ID-Tag, ist wieder eine zunehmende Nachfrage nach kostengünstigen Geräten, wie z.B. 8-Bit Mikroprozessoren, zu beobachten. Dabei stellt die kryptographische Sicherheit dieser Geräte eine größe Hürde für ihre breite Akzeptanz dar. Asymmetrische Kryptosysteme (engl. public-key cryptosystems, PKC) wie RSA und DSA sind nicht für den Einsatz auf solchen beschränkten, eingebetteten Geräten geeignet, weil sie im Allgemeinen für arithmetische Operationen Langzahlen (1024-2048 Bit Operanden) verwenden.

Die Elliptische Kurven Kryptographie (engl. Elliptic Curve Cryptography, ECC) hat sich als geeignete Alternative für eingebettete Systeme herausgestellt, weil sie mit kleinen Schlüssellängen, kleineren Operanden und vergleichsweise geringen arithmetischen Anforderungen auskommt. Die Implementierung eines standardisierten ECC Algorithmus auf 8-Bit Prozessoren stellt indes immer noch eine große Herausforderung dar, die als nicht praktikabel für rechen- und speicherbeschränkte Geräte angesehen wird.

In dieser Dissertation wird die Umsetzbarkeit von asymmetrischen Verfahren, insbesondere der Elliptische Kurven Kryptographie, auf Geräten mit beschränkten Resourcen behandelt. Die Elliptische Kurven Kryptographie ermöglicht ein großes Maß an Flexibilität aufgrund möglicher Freiheitsgrade bzgl. der Wahl verschiedenener Parameter und Algorithmen, welche in dieser Arbeit diskutiert und effizient implementiert werden. So wird zuerst gezeigt, dass es möglich ist, einen sicheren Schlüsselaustausch basierend auf ECC auf einem kostengünstigen, für drahtlose Anwendungen ausgelegten Prozessor (vergleichbar mit dem dem weit verbreiteten 8-Bit 8051 Mikroprozessor) zu implementieren. Diese gänzlich auf Software basierende Implementierung des Diffie-Hellman Protokolls mit Elliptischen Kurven (Elliptic Curve Diffie-Hellman, ECDH) führt arithmetische Berechnungen in einem optimalen 131-Bit Erweiterungskörper (optimal extension field, OEF) durch. Eine kryptografisch sichere Verbindung zwischen zwei Endteilnehmern wird auf einem solchen Gerät ohne kryptographischen Co-Prozessor innerhalb von drei Sekunden hergestellt.

Desweiteren untersuchen wir die Möglichkeiten von Software/Hardware Co-Design Ansätzen um mittels Architekturmodifikationen, z.B. Befehlssatzerweiterungen (Instruction Set Extensions, ISE) für körperarithmetische Basisoperationen wie sie bei ECC zum Einsatz kommen, die Performanz zu steigern. Es wird gezeigt, dass eine standardisierte 163-Bit Punkt-Multiplikation mit minimalen zusätzlichen Hardware-

Kosten auf einem 8-Bit AVR Mikro-Controller (ein typischer, kostengünstiger Prozessor), der mit 4 MHz getaktet ist, in 0,113 Sekunden ausgeführt werden kann. Dieses Design bringt im Vergleich zu einer rein Software-basierten Implementierung einen Geschwindigkeitsgewinn um mehr als das 30-fache, während die Größe des Quelltext verringert und weniger Arbeitspeicher verbraucht wird. Zusätzlich werden zwei neue Befehle für den MIPS 32-Bit Prozessor vorgeschlagen, die Reduktionen modulo Pseudo-Mersenne Primzahlen beschleunigen. Desweiteren wird gezeigt, dass für Multiplikationen in einem OEF ein vergrösserter Akkumulator in der ALU von Vorteil ist. Die vorgestellte Architektur führt zu einem Geschwindigkeitszuwachs um 180

Darüber hinaus werden architektonische Verbesserungen sowie optimale Parameter für Least Significant Digit (LSD) Multiplizierer für Binärkörper vorgestellt. Die architektonischen Verbesserungen basieren auf einem Double Accumulator Multiplier (DAM) und N-Accumulator Multiplier (NAM), welche beide klassische LSD Multiplizierer bzgl. der Geschwindigkeit übertreffen.

Im Anschluß wird eine effiziente ECC-Prozessorarchitektur (für 169-bit, 289-bit und 361-bit OEF) vorgestellt, die alle arithmetischen Operationen im Frequenzbereich durchführt. So wird eine optimierte 169-Bit OEF ECC Implementierung mit 24K Logikgattern für einen 0.35um CMOS Prozess präsentiert.

Schließlich wird eine flächenoptimierte ECC ASIC Implementierung für Binärköper mit standardisierte 133 bis 193 Bits vorgestellt. Es wird gezeigt, dass lediglich 10K bis 18K Logikgatter für eine 0.35um CMOS Implementierung benötigt werden. Daher eignet sich diese ECC Architektur insbesondere für kostengünstige Implementierungen, wie sie z.B. in drahtlosen Netzwerken zum Einsatz kommen.

Preface

ॐ भूर्भुवः स्वः तत्सवितुर्वरेण्यं।
भर्गो देवस्य धीमहि।
धियो यो नः प्रचोदयात्॥

This thesis describes the research that I conducted during the three and half years I worked as a researcher at the Ruhr-University Bochum. I hope that the results of this work are a step towards practical acceptance of *Elliptic Curve Cryptography* (ECC) in constrained devices among researchers and in the industry.

The work, I present in this thesis, would not have been possible without the support of several people. First of all, I would like to thank my advisor, Prof. Christof Paar, who gave me the opportunity to come to Germany and work under him. His advice and expertise in the field have been indispensable throughout my research. I would also like to thank Prof. Paar for his valuable friendship and camaraderie that has developed while working together with him. I would also like to thank my thesis committee, especially Prof. Ingrid Verbauwhede for the valuable suggestions, comments and advise that they gave me.

I would also like to thank my colleagues who have now become good friends over time. A special thanks to Kathy and Jorge Guajardo for their invaluable help in adjusting to Germany, being wonderful friends, and constant support during my stay here. To Thomas Wollinger for the interesting discussions on and about Deutsch we had every morning, thereby improving my vocabulary and not to forget the work we did together on digit multipliers. Andre Weimerskirch for being understanding and helpful during our industry projects. To Jan Pelzl and Kai Schramm for initiating me to the true German culture, for all those wonderful parties and fun we had at Dresden. Andy Rupp for all the help I could get just across the door. Kerstin Lemke-Rust for all the work we did together and managing the student projects. Thanks to Tim Güneysu for checking through the last minute additions to my thesis. Thomas Eisenbath, Marcus Heitmann, Timo Kasper, Axel Poschmann, Marko Wolf, and all the students for the wonderful atmosphere at COSY. Not to forget, is the most important person at COSY, our team assistant Irmgard Khn who helped with the terrible bureaucracy. Thanks also to the computer administrators, Marcel Selhorst, Christian Röpke and Horst Edelmann for being patient with all my demands.

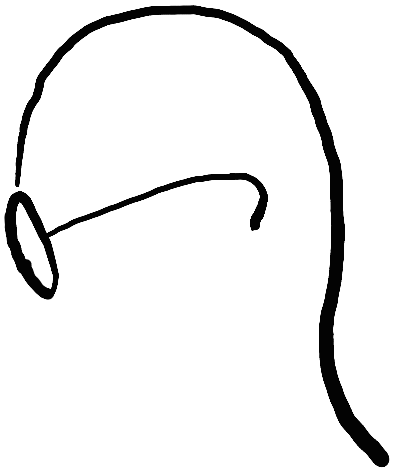
There are also several other people whom I would like to thank for the contributions to my work. Thanks to Johann Großschädl for the work done together on the extensions

for MIPS and Selcuk Baktir for the work on the frequency domain ECC. Thanks to Marco Girimondo for the patience and hard work involved in getting the Chipcon model ready for the SunNetwork 2003. Ho Won Kim for the fruitful discussions on VHDL and cryptographic hardware designs. I would also like to thank Holger Bock for the internship opportunity at Infineon Graz.

Last, but not least, I would like to thank my parents and my brother for their support, and bearing with the fact that we could not meet more often. Thanks to my cousin for her wonderful mails.

To all of you, thank you very much!!

Sandeep



Be the change you want to see in the world.
- Mahatma Gandhi

Table of Contents

Abstract iii

Kurzdarstellung v

Preface vii

1 Introduction 1
- 1.1 Security Requirements and Techniques . . . 2
  - 1.1.1 Public-Key Cryptography . . . 3
  - 1.1.2 Elliptic Curve Cryptography (ECC) . . . 5
- 1.2 Thesis Outline . . . 6

2 Mathematical Background 9
- 2.1 Introduction to Elliptic Curves . . . 9
- 2.2 Elliptic Curve Parameter Selection . . . 10
- 2.3 Elliptic Curve Arithmetic over $\mathbb{F}_p$ . . . 11
  - 2.3.1 Field Arithmetic over $\mathbb{F}_p$ . . . 13
- 2.4 Elliptic Curve Arithmetic over $\mathbb{F}_{2^m}$ . . . 17
  - 2.4.1 Field Arithmetic over $\mathbb{F}_{2^m}$ . . . 18
- 2.5 Elliptic Curve Arithmetic over $\mathbb{F}_{p^m}$ . . . 22
  - 2.5.1 Field Arithmetic over Optimal Extension Fields (OEF) . . . 23
- 2.6 Elliptic Curve Point Multiplication . . . 27
  - 2.6.1 Binary Method . . . 28
  - 2.6.2 Non-Adjacent Form (NAF) Method . . . 28
  - 2.6.3 Windowing Methods . . . 28
  - 2.6.4 Montgomery Method . . . 29
- 2.7 Elliptic Curve Key Exchange and Signature Protocols . . . 30
  - 2.7.1 Elliptic Curve Diffie-Hellman Key Exchange . . . 30

2.7.2 Elliptic Curve Digital Signature Algorithm 31

3 Software Design: ECDH Key Exchange on an 8-bit Processor 32
3.1 Motivation and Outline 32
3.2 Related Work 33
3.3 The Chipcon Architecture 33
3.4 Elliptic Curve Parameter Selection 33
3.5 Implementation aspects on the Chipcon processor 34
3.5.1 Field Arithmetic 34
3.5.2 Point Arithmetic 36
3.6 Communication Protocol 37
3.6.1 Key Establishment Phase 38
3.6.2 Normal Mode 39
3.7 Demonstration Application 39
3.8 Summary 41

4 Hardware/Software Co-design: Extensions for an 8-bit Processor 42
4.1 Motivation and Outline 42
4.2 Related Work 43
4.3 The FPSLIC Architecture 43
4.4 Implementation aspects on the AVR processor 44
4.4.1 Field Arithmetic 45
4.4.2 Point Arithmetic 46
4.5 Proposed Instruction Set Extensions 46
4.5.1 8-by-8 Bit-Parallel Multiplier 49
4.5.2 163-by-163 Bit-Serial Multiplier 49
4.5.3 163-by-163 Digit Serial Multiplier 50
4.5.4 A Flexible Multiplier 51
4.6 Summary 52

5 Hardware/Software Co-design: Extensions for a 32-bit Processor 56
5.1 Motivation and Outline 56
5.2 Related work 57
5.3 The MIPS32 architecture 58
5.4 Implementation aspects on the MIPS32 processor 59
5.5 Proposed extensions to MIPS32 61
5.5.1 Multiply/accumulate unit with a 72-bit accumulator 62

5.5.2 Custom instructions 63
5.5.3 Implementation details and performance evaluation 63
5.6 Summary 66

6 Hardware Design: Optimal Digit Multipliers for $\mathbb{F}_{2^m}$ 67
6.1 Motivation and Outline 67
6.2 Background on Digit-Serial Multipliers. 68
6.3 Architecture Options for LSD 69
6.3.1 Single Accumulator Multiplier (SAM) 70
6.3.2 Double Accumulator Multiplier (DAM) 74
6.3.3 N-Accumulator Multiplier (NAM) 78
6.4 Evaluation of the Multiplier Options 79
6.4.1 Evaluation of the SAM 80
6.4.2 Evaluation of all multiplier options 82
6.5 Summary 83

7 Hardware Design: ECC in the Frequency Domain 86
7.1 Motivation and Outline 86
7.2 Mathematical Background 87
7.2.1 Number Theoretic Transform (NTT) 87
7.2.2 Convolution Theorem and Polynomial Multiplication in the Frequency Domain 88
7.3 Modular Multiplication in the Frequency Domain 89
7.3.1 Mathematical Notation 90
7.3.2 DFT Modular Multiplication Algorithm 90
7.3.3 Optimization 91
7.4 Implementation of an ECC Processor Utilizing DFT Modular Multiplication 94
7.4.1 Base Field Arithmetic 94
7.4.2 Extension Field Multiplication 95
7.4.3 Point Arithmetic 99
7.5 Performance Analysis 100
7.6 Summary 102

8 Hardware Design: Tiny ECC Processor over $\mathbb{F}_{2^m}$ 106
8.1 Motivation and Outline 106
8.2 Mathematical Background 107

8.2.1 Squaring 108
8.2.2 Inversion 108
8.2.3 Point multiplication 110
8.3 Implementation Aspects 113
8.3.1 $\mathbb{F}_{2^m}$ Adder Unit 113
8.3.2 $\mathbb{F}_{2^m}$ Multiplier Unit 113
8.3.3 $\mathbb{F}_{2^m}$ Squarer Unit 115
8.3.4 ECC Processor design 116
8.4 Performance Analysis 118
8.5 Summary 120

9 Discussion **123**
9.1 Conclusions 123
9.2 Future Research 124

Bibliography **126**

List of Figures

1.1 Public-key encryption protocol 4

2.1 $y^2 = x^3 + a \cdot x + b$ over the reals 10
2.2 Multiply-and-accumulate strategy (m = 4) 24

3.1 Network security based on a trusted gateway. 38
3.2 Key exchange protocol. 39

4.1 Processor Core with Extension 47
4.2 ISE Interface and Structure 48
4.3 Bit-Serial LSB Multiplier 50
4.4 Digit-4 Serial LSD Multiplier 51
4.5 Bit-serial reduction circuit 52

5.1 4Km datapath with integer unit (IU) and multiply/divide unit (MDU) 58
5.2 Calculation of a column sum and subfield reduction 64

6.1 LSD-Single Accumulator Multiplier Architecture ($D = 5$) for $\mathbb{F}_{2^{163}}$ . . . 70
6.2 SAM multiplier core for $D = 5$ 71
6.3 SAM main reduction circuit for $D = 5$ 72
6.4 SAM final reduction circuit for $D = 5$ 73
6.5 DAM multiplier core for $D = 5$ 74
6.6 Overlap case 75
6.7 Underlap case 75
6.8 DAM main reduction circuit for $D = 5$ 77
6.9 DAM final reduction circuit for $D = 5$ 78
6.10 Time to complete one multiplication of the single accumulator multiplier 81
6.11 Area-Time Product of the single accumulator multiplier 82

6.12 Time to complete one multiplication of all the different multiplier implementations 83
6.13 Area-Time Product of the different multiplier implementations 84

7.1 Base Field Addition Architecture 94
7.2 Base Field Multiplication with Interleaved Reduction 96
7.3 Processing Cell for the Base Field Multiplier Core 97
7.4 DFT Montgomery Multiplication architecture 99
7.5 Top Level ECC Processor Architecture 101

8.1 $\mathbb{F}_{2^{163}}$ Most Significant Bit-serial (MSB) Multiplier circuit 115
8.2 $\mathbb{F}_{2^{163}}$ squaring circuit 116
8.3 Area optimized $\mathbb{F}_{2^n}$ ECC processor 117

List of Tables

1.1 Key length for public–key and symmetric–key cryptography 5

2.1 Operation counts for point addition and doubling on $y^2 = x^3 - 3x + b$. A = affine, P = standard projective, J = Jacobin, I = field inversion, M = field multiplication, S = field squaring [34] 13
2.2 Operation counts for point addition and doubling on $y^2 + xy = x^3 + ax^2 + b$. M = field multiplication, D = field division [34] 18

3.1 Field arithmetic performance on Chipcon (@3.68 Mhz) 36
3.2 ECC point arithmetic performance on Chipcon (@3.68 Mhz) 37
3.3 Memory map for the wireless reader implementation on CC1010 40

4.1 $\mathbb{F}_{2^{163}}$ ECC software-only performance on an 8-bit AVR μC (@4 Mhz) . 46
4.2 Software Only and Hardware Assisted Point Operation Performance (@4 Mhz) 54
4.3 ECC Scalar Point Multiplication Performance (@4 Mhz) 55

5.1 Format and description of useful instructions for OEF arithmetic 62

6.1 NIST recommended reduction polynomial for ECC and digit sizes possible 76
6.2 LSD Multiplier: Latency and critical path 80
6.3 Area and time complexity of $0.18\mu m$ CMOS standard cells 80
6.4 LSD Multiplier: Area 85

7.1 List of parameters suitable for optimized DFT modular multiplication. 92
7.2 Equivalent Gate Count Area of ECC Processor 101
7.3 Timing measurements (in clock cycles) for the ECC Processor 102
7.4 Controller Commands of ECC Processor 103
7.5 Point Doubling Instruction Sequence for ECC Processor. 104
7.6 Point Addition Instruction Sequence for ECC Processor 105

8.1 Standards recommended field sizes for $\mathbb{F}_{2^m}$ and reduction polynomial . 107
8.2 $\mathbb{F}_{2^m}$ inversion cost using Itoh-Tsuji method 110
8.3 $\mathbb{F}_{2^m}$ Squaring unit requirements 117
8.4 $\mathbb{F}_{2^m}$ ECC processor area (in equivalent gates) 119
8.5 $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *without* extra register R . 119
8.6 $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *with* extra register R . . . 120
8.7 Controller Commands of ECC Processor over $\mathbb{F}_{2^m}$ 121

Chapter 1

Introduction

Ubiquitous computing with low-cost pervasive devices have become a reality with sensor networks and RF-ID applications. Sensor networks offer tremendous benefits for the future as they have the potential to make life more convenient and safer. These constrained computing devices form large-scale collaborating networks by exchanging information. For instance, sensors can be used for climate control to reduce power consumption, for structures such as bridges to monitor the maintenance status, or for company badges to locate employees in order to increase productivity. Privacy and security of this information is important for the overall reliability of these networks and ultimately to the trustworthiness of the pervasive applications. Reliability in a lot of applications can be even more important than the secrecy of the data, because it is a protection against a failure by chance. In fact, security is often viewed as a crucial feature, a lack of which can be an obstacle to the wide-spread introduction of these applications. It is also important to note that a large share of those embedded applications will use wireless communication to reduce wiring costs, and increase flexibility and mobility. In addition, there is also a growing need for a rising number of modern appliances to be networked with each other, demanding solutions for cost-effective wireless networking. However, the main disadvantage of using a wireless network is that the communication channel is especially vulnerable to eavesdropping and the need for security becomes even more obvious.

Until a few years ago, only computers and data transferred through the Internet had been protected against unwanted harm. However, with the growing number of low-cost pervasive applications which are less secure than traditional systems, makes them an easy target for future attacks. This could include manipulating or preventing the functionality of these pervasive systems. In future, when more security critical

applications begin to depend on such devices, functionality failures could be even life-threatening.

We first present the security requirements in such systems (which are also equally applicable in general settings) and discuss the need for public key cryptography, in particular *Elliptic Curve Cryptography*.

1.1 Security Requirements and Techniques

Cryptography, or the art and science of keeping messages secure [70] involves mathematical techniques that provide the following security services:

- **Confidentiality** is a service used to keep the information accessible only to the authorized users of the communication. This service includes both protection of all user data transmitted between two points over a period of time as well as protection of traffic flow analysis.
- **Integrity** is a service that requires that system assets and transmitted information be capable of modification only by authorized users. Modification includes writing, changing the status, deleting, creating, delaying, and replaying of transmitted messages.
- **Authentication** is a service that is concerned with assuring that the origin of a message, date of origin, data content, time sent, etc are correctly identified. This service is subdivided into two major classes: entity authentication and data origin authentication. Note that the second class of authentication implicitly provides data integrity.
- **Non-repudiation** is a service which prevents both the sender and the receiver of a transmission from denying previous commitments or actions.

Symmetric-key cryptography (also known as private-key cryptography) provides the ability to securely and confidentially exchange messages between two parties. This is especially important if the data should not be revealed to any third party. Integrity can be guaranteed by using the proper *mode of operation* with the symmetric cipher. Authentication without non-repudiation can also be achieved using symmetric key cryptography if the secret key is known only to the two parties.

Some chip manufacturers have already understood these security needs and include small symmetric-key cryptographic co-processors for their low cost processors [13].

Though symmetric key algorithms do provide most of the security services, it has two major disadvantages:

- It requires secure transmission of a secret key, before being able to exchange messages.
- In a networked environment, each pair of users need a different key resulting in an n user system requiring $(\frac{n\cdot(n-1)}{2})$ key pairs .

Setting up this shared secret manually turns out to be unmanageable as such perversive applications involve a much larger number of entities than in traditional systems. It also introduces the problem of secure storage of the large number of secret key pairs.

Asymmetric-key cryptography (known generally as public-key (PK) cryptography) proposed in 1976 by Diffie and Hellman [18] introduces a new concept which can achieve the above security requirements including non-repudiation.

1.1.1 Public-Key Cryptography

Public-key cryptography is based on the idea of separating the key used to encrypt a message from the one used to decrypt it. Anyone who wants to send a message to a party, e.g., *Bob*, can encrypt that message using *Bob*'s *public key*, but only *Bob* can decrypt the message using his *private key*. The basic protocol between the two communication parties Alice and Bob can be seen in Figure 1.1, where K_{pub} denotes the public key of Bob and K_{pr} the private (not publicly available) key of Bob.

It is understood that the private key should be kept secret at all times and the public key is publicly available to everyone. Furthermore, it is computationally impossible (or in any reasonable amount of time) for anyone, except *Bob*, to derive the private key.

Public-key algorithm are not only used for the exchange of a key, but also for the authentication by using digital signatures. This enables the communication parties to prove that one of them had actually generated the message (non-repudiation). This is a crucial functionality to assure reliability in pervasive applications. It is important to note that sender non-repudiation can only be achieved using public-key cryptography.

One can realize three basic mechanisms with public–key algorithms:

- Key establishment protocols and key transport protocols without prior exchange of a joint secret,
- Digital signature algorithms, and

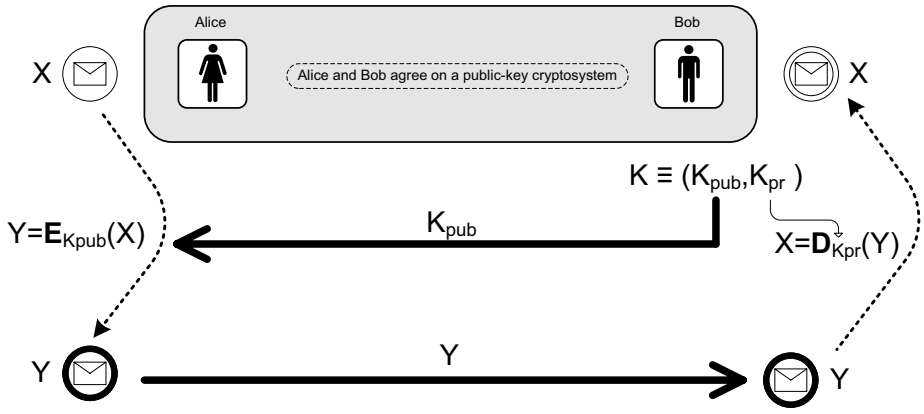


Figure 1.1: Public-key encryption protocol

- Encryption.

Though public-key schemes can provide all functionality needed in modern security protocols such as SSL/TLS, it has been the hardest to implement due to its very high computational requirements. Even when properly implemented, all PK schemes proposed to date are several orders of magnitude slower than the best known private-key schemes. Hence, in practice, cryptographic systems are a mixture of symmetric-key and public-key cryptosystems and are called *hybrid cryptosystems*. A public-key algorithm is chosen for key establishment and then a symmetric-key algorithm is chosen to encrypt the communication data, achieving in this way high throughput rates.

In general, one can divide practical public-key algorithms into three families:

- Algorithms based on the *Integer Factorization Problem (IFP):* given a positive integer n, find its prime factorization. E.g., RSA [69], the most popular public–key algorithm named after its creators — Rivest, Shamir, and Adelman

- Algorithms based on the *Discrete Logarithm Problem (DLP):* given α and β find positive integer k such that $\beta = \alpha^k \bmod p$. E.g., the Diffie-Hellman (DH) key exchange protocol [18] and the Digital Signature Algorithm (DSA) [60].

- Algorithms based on *Elliptic Curve Discrete Logarithm Problem (ECDLP):* given points P and Q on an elliptic curve defined over a finite field, find positive integer k such that $Q = k \cdot P$. E.g., the *Elliptic Curve Diffie-Hellman* (ECDH) key exchange protocol and the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [60].

In addition, there are many other public-key schemes, such as NTRU, or systems based on hidden field equations, which are not in wide spread use. The scientific community is only at the very beginning of understanding the security of such algorithms.

The computationally most intensive operation for RSA and Discrete Logarithm (DL) based public-key schemes are based on modular exponentiation, i.e., the operation $x^e \bmod n$. These operations have to be performed using very long operands, typically 1024–2048 bits in length. However for ECDLP systems, the operands are in the range of 160–256 bits in length. An extended discussion regarding key equivalences between different asymmetric and symmetric cryptosystems in given in [51]. Table 1.1 puts the public–key bit length in perspective to the symmetric key algorithms.

Table 1.1: Key length for public–key and symmetric–key cryptography

Symmetric–key	**ECC**	**RSA/DLP**	Remarks
64 bit	128 bit	700 bit	only short term security (breakable with some effort)
80 bit	160 bit	1024 bit	medium term security (excl. government attacks)
128 bit	256 bit	2048–3072 bits	long term security (excl. advances in quantum computing)

1.1.2 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography is a relatively new cryptosystem, suggested independently in 1986 by Miller [55] and Koblitz [43]. At present, ECC has been commercially accepted, and has also been adopted by many standardizing bodies such as ANSI [2], IEEE [36], ISO [38] and NIST [60].

Elliptic curve cryptosystems are based on the well known Discrete Logarithm Problem (DLP). Elliptic curves defined over a finite field provide a group structure that is used to implement the cryptographic schemes. The elements of the group are the rational points on the elliptic curve, together with a special point $\mathcal{O}$ (called the "point at infinity") acting as the identity element of the group. The group operation is the addition of points, which can be carried out by means of arithmetic operations in the underlying finite field (which will be discussed in detail in Chapter 2). A major building block of all elliptic curve cryptosystems is the *scalar point multiplication*, an operation of the form $k \cdot P$ where k is a positive integer and P is a point on the elliptic

curve. Computing $k \cdot P$ means adding the point P exactly $k-1$ times to itself, which results in another point Q on the elliptic curve[1]. The inverse operation, i.e., to recover k when the points P and $Q = k \cdot P$ are given, is known as the *Elliptic Curve Discrete Logarithm Problem* (ECDLP). To date, no subexponential-time algorithm is known to solve the ECDLP in a properly selected elliptic curve group [61]. This makes *Elliptic Curve Cryptography* a promising branch of public key cryptography which offers similar security to other "traditional" DLP-based schemes in use today, with smaller key sizes and memory requirements, e.g., 160 bits instead of 1024 bits (as shown in Table 1.1).

Many of the new security protocols decouple the choice of cryptographic algorithm from the design of the protocol. Users of the protocol negotiate on the choice of algorithm to use for a particular secure session. Hence, ECC based algorithms can be easily integrated into existing protocols to achieve the same security and backward compatibility with smaller resources. Hence more low-end constrained devices can use such protocols which till recently were considered unsuitable for such systems.

1.2 Thesis Outline

With *Elliptic Curve Cryptography* emerging as a serious alternative, the desired level of security can be attained with significantly smaller keys. This makes ECC very attractive for small-footprint devices with limited computational capacities, memory and low-bandwidth network connections. Another major advantage of ECC is that the domain parameters can be (judiciously) chosen to improve implementation performance. However ECC is still considered to be impracticable for very low-end devices. In this thesis, we show that *Elliptic Curve Cryptography* can indeed be used on such constrained devices without adversely affecting performance. This is made possible based on the flexibility that ECC provides in terms of the choice of different algorithms and parameters. We present techniques for ECC implementation in three domains: pure software based implementations on low-end processors, hardware/software co-design with extensions for such processors, and finally stand alone low area cryptographic processors.

In Chapter 2, we first present the mathematical background to the elliptic curves over different finite fields and their group operations. We also present the different algorithms that are used for the implementation of the finite field arithmetic and the

[1]Scalar multiplication in an additive group is the equivalent operation to exponentiation in a multiplicative group.

point arithmetic. We restrict our attention only to those algorithms that are relevant to this thesis. The two most common schemes, ECDH and ECDSA are also presented.

In Chapter 3, we present a public-key cryptographic implementation for secure key exchange on low-end wireless devices used in sensor networks using elliptic curves. Our implementation is based on *Optimal Extension Fields* (OEF) that are a special type of finite fields $\mathbb{F}_{p^m}$. As our platform we chose a Chipcon CC1010 chip [13] which is based on the 8051 architecture and is especially suited for secure wireless applications as it has a built-in radio transceiver as well as a hardware DES engine. We were able to establish a secure end-to-end connection between the sensor and a base station in an acceptable time of 3 seconds without requiring a cryptographic coprocessor.

In Chapter 4, we describes a proof-of-concept implementation for an extremely low-cost instruction set extension using reconfigurable logic, which enables an 8-bit AVR micro-controller running at 4 Mhz (a typical representative for a low-cost pervasive processor) to provide full size elliptic curve cryptographic capabilities. We show that a standard compliant 163-bit point multiplication can be computed in 0.113 sec on an 8-bit AVR micro-controller running at 4 Mhz with minimal extra hardware. Our design not only accelerates the computation by a factor of more than 30 compared to a software-only solution, it also reduces the code-size and data-RAM.

In Chapter 5, we investigate the potential of architectural enhancements and instruction set extensions for low-level arithmetic used in public-key cryptography, most notably multiplication in finite fields of large order. The focus of the contribution is directed towards the special Optimal Extension Fields where p is a pseudo-Mersenne (PM) prime of the form $p = 2^n - c$ that fits into a single register. Based on the MIPS32 instruction set architecture, we introduce two custom instructions to accelerate the reduction modulo a PM prime. Moreover, we show that the multiplication in an OEF can take advantage of a multiply and accumulate unit with a wide accumulator, so that a certain number of 64-bit products can be summed up without overflow. The proposed extensions support a wide range of PM primes and allow a reduction modulo $2^n - c$ to complete in only four clock cycles when $n \leq 32$.

In Chapter 6, we show different architectural enhancements in Least Significant Digit (LSD) multiplier for binary fields $\mathbb{F}_{2^m}$. Digit Serial Multipliers are now used extensively in hardware implementations of *Elliptic Curve Cryptography*. We propose two new different architectures, the Double Accumulator Multiplier (DAM) and N-Accumulator Multiplier (NAM) which are both faster compared to traditional LSD multipliers. Our evaluation of the multipliers for different digit sizes gives optimum choices and shows that presently used digit sizes are the worst possible choices. Hence,

one of the most important results of this contribution is that digit sizes of the form $2^l - 1$, where l is an integer, are preferable for the digit multipliers. Furthermore, we show that one should always use the NAM architecture to get the best timings. Considering the time area product DAM or NAM gives the best performance depending on the digit size.

In Chapter 7, we propose an area/time efficient ECC processor architecture which performs all finite field arithmetic operations in the discrete Fourier domain. The proposed architecture utilizes a class of OEF $\mathbb{F}_{p^m}$ where the field characteristic is a Mersenne prime $p = 2^n - 1$ and $m = n$. The main advantage of our architecture is that it achieves extension field modular multiplication in the *discrete Fourier domain* with only a *linear* number of base field $\mathbb{F}_p$ multiplications in addition to quadratic number of simpler operations such as addition and bitwise rotation. With its low area and high speed, the proposed architecture is well suited for *Elliptic Curve Cryptography* in constrained environments such as wireless sensor networks.

In Chapter 8, we present a stand-alone highly area optimized ECC processor design for standards compliant binary field curves. We use the fast squarer implementation to construct an addition chain that allows inversion to be computed efficiently. Hence, we propose an affine co-ordinate ASIC implementation of the ECC processor using a modified Montgomery point multiplication method for binary curves ranging from $133 - 193$ bits. An area between 10k and 18k gates on a 0.35um CMOS process is possible for the different curves which makes the design very attractive for enabling ECC in constrained devices.

We finally end this dissertation with a summary of our work and some suggestions for future research.

Chapter 2

Mathematical Background

Parts of this chapter are published in the survey articles [28] and [27]

2.1 Introduction to Elliptic Curves

An elliptic curve E over a field K is the set of solutions to the cubic equation

$$E : F(x,y) = y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0 \quad \text{where} \quad a_i \in K.$$

and the discriminant defined as

$$\begin{aligned} \Delta &= -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \neq 0 \\ &\text{where} \\ &d_2 = a_1^2 + 4a_2 \\ &d_4 = 2a_4 + a_1a_3 \\ &d_6 = a_3^2 + 4a_6 \\ &d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{aligned}$$

If L is an extension field of K, then the set of *L-rational points* on E is given as

$$E(L) = \{(x,y) \in L \times L : F(x,y) = 0\} \cup \{\mathcal{O}\} \tag{2.1}$$

where $\mathcal{O}$ is the *point at infinity*.
We construct an additive abelian group $(E, +)$ given by the points on the curve and an additive group operation defined on these points. Hence,

- **Set E:** Points on the curve given by $E(L)$.
- **Operation $+$:** $P + Q = (x_1, y_1) + (x_2, y_2) = R = (x_3, y_3)$.

Figure 2.1 shows an example of an elliptic curve over the field of real numbers. Finding $P + Q = R$ in a geometrical manner can be achieved by the following two steps and is shown in Figure 2.1:

a) $P \neq Q \rightarrow$ line through P and Q and mirror the point of third interception along the x-axis.

b) $P = Q \Rightarrow P + Q = 2Q \rightarrow$ tangent line through Q and mirror the point of second intersection along the x-axis.

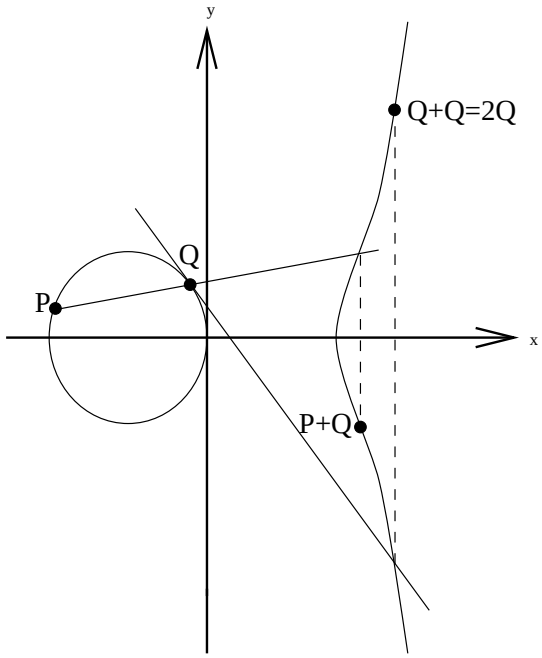


Figure 2.1: $y^2 = x^3 + a \cdot x + b$ over the reals

A more detailed discussion on elliptic curves can be found in [75].

2.2 Elliptic Curve Parameter Selection

An implementation of an elliptic curve cryptosystem requires a number of decisions to be taken at different *hierarchy* levels depending on the underlying hardware and

implementation goals that need to be achieved.

- **At the field level**
  - Selection of the underlying field (could be $\mathbb{F}_{2^m}$, $\mathbb{F}_p$ or $\mathbb{F}_{p^m}$).
  - Choosing the field representation (e.g., polynomial basis or normal basis).
  - Field arithmetic algorithms for field addition (subtraction), multiplication, reduction and inverse.
- **At the elliptic curve level**
  - Choosing the type of representation for the points (affine or projective coordinates).
  - Choosing a point addition and doubling algorithm.
- **At the protocol level**
  - Choosing the appropriate protocol (key-exchange or signature).
  - Choosing the algorithm for scalar multiplication $k \cdot P$.

These choices provide a huge flexibility and hence makes ECC viable for both constrained devices and high performance servers. We first present the arithmetic for different elliptic curves defined over three different field choices $\mathbb{F}_p$ (Section 2.3), $\mathbb{F}_{2^m}$ (Section 2.4) and $\mathbb{F}_{p^m}$ (Section 2.5). Then we present the different methods to perform the point multiplication in Section 2.6. In Section 2.7, the key-exchange and signature protocols are discussed. The algorithms that are presented here are limited only to those that are relevant to this thesis. A more exhaustive list of different algorithms can be found in references [34, 54].

2.3 Elliptic Curve Arithmetic over $\mathbb{F}_p$

An elliptic curve E over $\mathbb{F}_p$ (*characteristic* not equal to 2 or 3) is the set of solutions (x, y) which satisfy the simplified Weierstrass equation:

$$E : y^2 = x^3 + ax + b \tag{2.2}$$

where $a, b \in \mathbb{F}_p$ and $4a^3 + 27b^2 \neq 0$, together with the point at infinity $\mathcal{O}$.
The group laws are defined in terms of underlying field operations in $\mathbb{F}_p$ as follows:

Identity

$P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E(\mathbb{F}_p)$.

Negation

If $P = (x, y) \in E(\mathbb{F}_p)$, then $P + Q = \mathcal{O}$ is given by the point $Q = (x, -y) \in E(\mathbb{F}_p)$ which is the negation of P (denoted as $-P$). Note, $-\mathcal{O} = \mathcal{O}$.

Point Addition and Doubling

Let $P = (x_0, y_0) \in E(\mathbb{F}_p)$ and $Q = (x_1, y_1) \in E(\mathbb{F}_p)$, where $Q \neq -P$.
Then $P + Q = (x_2, y_2)$, where

$$\begin{aligned} x_2 &= \lambda^2 - x_0 - x_1 \\ y_2 &= \lambda(x_1 - x_2) - y_1 \end{aligned} \tag{2.3}$$

and

$$\lambda = \begin{cases} \frac{y_0 - y_1}{x_0 - x_1} & \text{if } P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P = Q \end{cases}$$

Projective coordinate representations

The coordinate representation considered so far is known as the affine representation. However, in many applications it is more convenient to represent the points P and Q in projective coordinates.

- In the *standard projective coordinates*, a point is represented by the tuple $(X : Y : Z)$, $Z \neq 0$ which corresponds to the affine point $(X/Z, Y/Z)$. The point at infinity $\mathcal{O}$ is $(0 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.
- In the *Jacobian projective coordinates*, a point is similarly represented by the tuple $(X : Y : Z)$, $Z \neq 0$ which corresponds to the affine point $(X/Z^2, Y/Z^3)$. The point at infinity $\mathcal{O}$ is $(1 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$.

This representation is advantageous when inversion is computationally much more expensive compared to multiplication in the finite field. The algorithms for projective coordinates trade inversions in the point addition and doubling operations for a larger number of multiplications followed by single inversion at the end of the algorithm.

This single inversion can be computed via exponentiation using the *Fermat's Little Theorem* [12]: $x^{-1} \equiv x^{p-2} \bmod p$.

One can derive expressions equivalent to Eq. 2.3 for point addition and doubling operations in both the projective coordinates. We refer to [34] for the actual algorithms. In Table 2.1 we present the complexity of the group operations considering different coordinates representations. The complexity of addition or doubling a point on an elliptic curve is given by the number of field multiplications, squarings and inversions (if affine coordinates are being used). Field additions are relatively cheap operations compared to multiplications or inversions, and therefore are neglected in the tables.

Table 2.1: Operation counts for point addition and doubling on $y^2 = x^3 - 3x + b$. A = affine, P = standard projective, J = Jacobin, I = field inversion, M = field multiplication, S = field squaring [34]

Doubling		**General addition**		**Mixed coordinates**	
$2A \to A$	$1I, 2M, 2S$	$A + A \to A$	$1I, 2M, 1S$	$J + A \to J$	$8M, 3S$
$2P \to P$	$7M, 3S$	$P + P \to P$	$12M, 2S$		
$2J \to J$	$4M, 4S$	$J + J \to J$	$12M, 4S$		

2.3.1 Field Arithmetic over $\mathbb{F}_p$

To perform the above introduced group operations, we have to compute the underlying field arithmetic operations on the prime field. The crucial field operations are the modular addition, subtraction, multiplication and inverse.

Addition and Subtraction

Field addition over $\mathbb{F}_p$ is performed using multi-precision integer addition followed by a reduction if required as shown in Algorithm 2.1.

Algorithm 2.1 Addition in $\mathbb{F}_p$

Input: $A, B \in \mathbb{F}_p$.
Output: $C \equiv A + B \bmod p$.
1: $C \leftarrow A + B$ { multi-precision integer addition }
2: **if** $C \geq p$ **then**
3: $C \leftarrow C - p$
4: **end if**
5: Return (C)

Similarly subtraction in $\mathbb{F}_p$ is a multi-precision integer subtraction followed by an additional addition with p if the result is negative (Algorithm 2.2).

Algorithm 2.2 Subtraction in $\mathbb{F}_p$

Input: $A, B \in \mathbb{F}_p$.
Output: $C \equiv A - B \bmod p$.
1: $C \leftarrow A - B$ { multi-precision integer subtraction }
2: **if** $C < 0$ **then**
3: $C \leftarrow C + p$
4: **end if**
5: Return (C)

Multiplication and Squaring

Modular multiplication or squaring can be done by first performing a multi-precision integer multiplication or squaring, respectively, and then reducing the double bit-length result with the prime p. Multi-precision integer multiplication and squaring can be done based on different techniques like operand scanning, product scanning [15] or Karatsuba method [41]. Each of them has its advantages and disadvantages based on the underlying implementation hardware.

Field Reduction

Field reduction can be performed very efficiently if the modulus p is a generalized Mersenne (GM) prime. These primes are sum or differences of a small number of powers of 2 and have been adopted as recommended curves in different standards like

NIST [60], ANSI [2] and SEC [1]. The normally used GM primes for different field sizes are shown here:

$$\begin{aligned} p_{160} &= 2^{160} - 2^{31} - 1 \\ p_{192} &= 2^{192} - 2^{64} - 1 \\ p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \end{aligned}$$

Fast reduction is possible using these primes since the powers of 2 translate naturally to bit locations in hardware. For e.g., $2^{160} \equiv 2^{31} + 1 \bmod p_{160}$ and therefore each of the higher bits can be wrapped to the lower bit locations based on the equivalence. The steps required to compute the fast reduction using GM primes is given in NIST [60]

However when using general primes which are not GM primes, two other different techniques can be used: Barrett reduction and Montgomery reduction.
The Barrett reduction [8] for $r = x \bmod p$ is shown in Algorithm 2.3. It requires a precomputation of $\mu = \lfloor b^{2k}/p \rfloor$ where b is the radix for the representation of x and is mostly chosen to be the word-size of the processor. This allows all divisions to be performed in the algorithm as simple word level shifts. The method is suitable if many reductions are to be done using the same modulus, which is mostly the case.

Algorithm 2.3 Barrett reduction in $\mathbb{F}_p$

Input: $x \leq b^{2k}$ where $b \geq 3$, and $\mu = \lfloor b^{2k}/p \rfloor$ where $k = \lfloor \log_b p \rfloor + 1$.
Output: $r \equiv x \bmod p$.

1: $\hat{q} \leftarrow \lfloor \lfloor x/b^{k-1} \rfloor \cdot \mu / b^{k+1} \rfloor$.
2: $r \leftarrow (z \bmod b^{k+1}) - (\hat{q} \cdot p \bmod b^{k+1})$.
3: If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4: While $r \geq p$ do: $r \leftarrow r - p$.
5: Return (r)

The Montgomery method [58] uses a special representation to perform arithmetic efficiently. The cost for changing the representation is costly and therefore this method is useful only if performing multiple reductions using the same modulus. Let $R > p$ with $gcd(R, p) = 1$, then the Montgomery reduction of $x < pR$ is $r = xR^{-1} \bmod p$. The Algorithm 2.4 shows the procedure where the elements are represented in word basis b. Choosing $R = b^n$ allows the different operations to be simple word level shifts.

Algorithm 2.4 Montgomery reduction in $\mathbb{F}_p$

Input: $x < pR$ where $R = b^n, gcd(p, b) = 1$, and $p' = -p^{-1} \bmod b$.
Output: $r \equiv xR^{-1} \bmod p$.

1: $r \leftarrow x$ $\quad$ { Notation: $(r_{2n-1} \cdots r_1 r_0)$ }
2: **for** i from 0 to $(n-1)$ **do**
3: $\quad u_i \leftarrow r_i p' \bmod b$.
4: $\quad r \leftarrow r + u_i p\ b^i$.
5: **end for**
6: $r \leftarrow r/b^n$
7: If $r \geq p$ then $r \leftarrow r - p$.
8: Return (r).

Inversion

Inversion is performed using extended binary Euclidean algorithm which replaces the expensive division with shifts as shown in Algorithm 2.5.

Algorithm 2.5 Binary Euclidean algorithm in $\mathbb{F}_p$

Input: Prime p and $x \in [1, p)$.
Output: $r \equiv x^{-1} \bmod p$.

1: $u \leftarrow x$, $v \leftarrow p$.
2: $x_1 \leftarrow 1$, $x_2 \leftarrow 0$.
3: **while** $u \neq 1$ and $v \neq 1$ **do**
4: $\quad$ **while** u is even **do**
5: $\quad\quad u \leftarrow u/2$.
6: $\quad\quad$ If x_1 is even then $x_1 \leftarrow x_1/2$; else $x_1 \leftarrow (x_1 + p)/2$.
7: $\quad$ **end while**
8: $\quad$ **while** v is even **do**
9: $\quad\quad v \leftarrow v/2$.
10: $\quad\quad$ If x_2 is even then $x_2 \leftarrow x_2/2$; else $x_2 \leftarrow (x_2 + p)/2$.
11: $\quad$ **end while**
12: $\quad$ If $u \geq v$ then $u \leftarrow u - v$, $x_1 \leftarrow x_1 - x_2$;
13: $\quad$ Else $v \leftarrow v - u$, $x_2 \leftarrow x_2 - x_1$.
14: **end while**
15: If $u = 1$ then Return $(x_1 \bmod p)$; else Return $(x_2 \bmod p)$.

2.4 Elliptic Curve Arithmetic over $\mathbb{F}_{2^m}$

An elliptic curve E over $\mathbb{F}_{2^m}$ is the set of solutions (x, y) which satisfy the simplified Weierstrass equation:

$$E: \ y^2 + xy = x^3 + ax^2 + b \tag{2.4}$$

where $a, b \in \mathbb{F}_{2^m}$ and $b \neq 0$, together with the point at infinity $\mathcal{O}$. Due to the Weil Descent attack [21], m is chosen to be a prime.
The group laws are defined in terms of underlying field operations in $\mathbb{F}_{2^m}$ as follows:

Identity

$P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E(\mathbb{F}_{2^m})$.

Negation

If $P = (x, y) \in E(\mathbb{F}_{2^m})$, then $P + Q = \mathcal{O}$ is given by the point $Q = (x, x + y) \in E(\mathbb{F}_{2^m})$ which is the negation of P (denoted as $-P$). Note, $-\mathcal{O} = \mathcal{O}$.

Point Addition and Doubling

Let $P = (x_0, y_0) \in E(\mathbb{F}_{2^m})$ and $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$, where $Q \neq -P$.
Then $P + Q = (x_2, y_2)$, where

$$\begin{aligned} x_2 &= \lambda^2 + \lambda + x_0 + x_1 + a \\ y_2 &= \lambda(x_0 + x_2) + x_2 + y_0 \end{aligned} \tag{2.5}$$

and

$$\lambda = \begin{cases} \frac{y_0 + y_1}{x_0 + x_1} & \text{if } P \neq Q \\ x_1 + \frac{y_1}{x_1} & \text{if } P = Q \end{cases}$$

Projective coordinate representations

It is more convenient to represent the points P and Q in projective coordinates when inversions are computationally more expensive compared to multiplications in $\mathbb{F}_{2^m}$.

- In the *standard projective coordinates*, a point is represented by the tuple $(X : Y : Z)$, $Z \neq 0$ which corresponds to the affine point $(X/Z, Y/Z)$. The point at infinity $\mathcal{O}$ is $(0 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$.

- In the *Jacobian projective coordinates*, a point is represented by the tuple $(X : Y : Z)$, $Z \neq 0$ which corresponds to the affine point $(X/Z^2, Y/Z^3)$. The point at infinity $\mathcal{O}$ is $(1 : 1 : 0)$ and the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$.
- In the *López-Dahab (LD) coordinates*, a point is represented by the tuple $(X : Y : Z)$, $Z \neq 0$ which corresponds to the affine point $(X/Z, Y/Z^2)$. The point at infinity $\mathcal{O}$ is $(1 : 0 : 0)$ and the negative of $(X : Y : Z)$ is $(X : X + Y : Z)$.

We refer to [34] for expressions equivalent to Eq. 2.5 for the point addition and doubling operations in the projective coordinates. In Table 2.2, we present the complexity of the group operations considering different coordinates representations.

Table 2.2: Operation counts for point addition and doubling on $y^2 + xy = x^3 + ax^2 + b$. M = field multiplication, D = field division [34]

Coordinate system	**General addition**	**Mixed coordinates**	**Doubling**
Affine	D + M	—	D + M
Standard projective	13M	12M	7M
Jacobian projective	14M	8M	4M
López-Dahab projective	14M	8M	4M

2.4.1 Field Arithmetic over $\mathbb{F}_{2^m}$

The field operations required to implement the elliptic curve group operation are addition, multiplication (squaring) and inverse in $\mathbb{F}_{2^m}$.

Polynomial Basis Representation

The *standard polynomial basis* representation is used for our implementations with the reduction polynomial $F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$ where $g_i \in \{0, 1\}$, for $i = 1, ..., m-1$ and $g_0 = 1$.
Let α be a root of $F(x)$, then we represent $A \in \mathbb{F}_{2^m}$ in polynomial basis as

$$A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i, \quad a_i \in \mathbb{F}_2 \tag{2.6}$$

This polynomial can also be represented as a bit vector: $(a_{m-1}, ...a_1, a_0)$.
The field arithmetic is implemented as polynomial arithmetic modulo $F(x)$. Notice that by assumption $F(\alpha) = 0$ since α is a root of $F(x)$. Therefore,

$$\alpha^m = -G(\alpha) = \sum_{i=0}^{m-1} g_i \alpha^i \tag{2.7}$$

gives an easy way to perform modulo reduction whenever we encounter powers of α greater than $m-1$. Throughout the text, we will write $A \bmod F(\alpha)$ to mean *explicitly* the reduction step.

Addition

$\mathbb{F}_{2^m}$ addition is the simplest of all operations, since it is a bitwise addition in $\mathbb{F}_2$ which maps to an XOR operation ($\oplus$) in software or hardware.

$$\begin{aligned} C &\equiv A + B \bmod F(\alpha) \\ &\equiv (a_{m-1} \oplus b_{m-1})\alpha^{m-1} + \ldots + (a_1 \oplus b_1)\alpha + (a_0 \oplus b_0) \end{aligned}$$

Multiplication and Squaring

The multiplication of two elements $A, B \in \mathbb{F}_{2^m}$, with $A(\alpha) = \sum_{i=0}^{m-1} a_i \alpha^i$ and $B(\alpha) = \sum_{i=0}^{m-1} b_i \alpha^i$ is given as

$$C(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i \equiv A(\alpha) \cdot B(\alpha) \bmod F(\alpha) \tag{2.8}$$

where the multiplication is a polynomial multiplication, and all α^t, with $t \geq m$ are reduced with Eq. 2.7.

The simplest algorithm for field multiplication is the shift-and-add method [42] with the reduction step inter-leaved (Algorithm 2.6). The shift-and-add method is not suitable for software implementations as the bitwise shifts are hard to implement across the words on a processor. A more efficient method for implementing the multiplier in software is the Comb method [53]. Here the multiplication is implemented efficiently in two separate steps, first performing the polynomial multiplication to obtain $2n$-bit length polynomial and then reducing it using special reduction polynomials.

Algorithm 2.7 shows the polynomial multiplication using the Comb method. The operation SHIFT$(A << k) = \sum_{i=0}^{m-1} a_i \alpha^{(i+k)}$, performs a k-bit shift across the words

Algorithm 2.6 Shift-and-Add Most Significant Bit (MSB) first $\mathbb{F}_{2^m}$ multiplication

Input: $A = \sum_{i=0}^{m-1} a_i \alpha^i$, $B = \sum_{i=0}^{m-1} b_i \alpha^i$ where $a_i, b_i \in \mathbb{F}_2$.
Output: $C \equiv A \cdot B \bmod F(\alpha) = \sum_{i=0}^{m-1} c_i \alpha^i$ where $c_i \in \mathbb{F}_2$.

1: $C \leftarrow 0$
2: **for** $i = m-1$ downto 0 **do**
3: $\quad C \leftarrow b_i \cdot (\sum_{i=0}^{m-1} a_i \alpha^i) + (\sum_{i=0}^{m-1} c_i \alpha^i) \cdot \alpha \bmod F(\alpha)$
4: **end for**
5: Return (C)

Algorithm 2.7 Comb Method for $\mathbb{F}_{2^m}$ multiplication on a w-bit processor.

Input: $A = \sum_{i=0}^{m-1} a_i \alpha^i$, $B = \sum_{i=0}^{m-1} b_i \alpha^i$ where $a_i, b_i \in \mathbb{F}_2$ and $s = \lceil \frac{m}{w} \rceil$.
Output: $C = A \cdot B = \sum_{i=0}^{2m-1} c_i \alpha^i$, where $c_i \in \mathbb{F}_2$

1: $C \leftarrow 0$
2: **for** $j = 0$ to $w-1$ **do**
3: $\quad$ **for** $i = 0$ to $s-1$ **do**
4: $\quad\quad C \leftarrow b_{wi+j} \cdot \text{SHIFT}(A << w.i) + C$
5: $\quad$ **end for**
6: $\quad A \leftarrow \text{SHIFT}(A << 1)$
7: **end for**
8: Return (C)

without reduction. It is important to note that $\text{SHIFT}(A << w.i)$ where w is the word-length of the processor, are the same original set of bytes referenced with a different memory pointer and therefore requires no actual shifts in software.

For hardware, the shift-and-add method can be implemented efficiently and is suitable when area is constrained. When the bits of B are processed from the most-significant bit to the least-significant bit (as in Algorithm 2.6), then its implemented as Most-Significant Bit-serial (MSB) multiplier. Similarly a Least-Significant Bit-serial (LSB) multiplier can be implemented and the choice between the two depends on the design architecture and goals. Digit multipliers, introduced in [76], 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 D.

The total number of digits in the polynomial of degree $m-1$ is given by $d = \lceil m/D \rceil$.

Then, we can re-write the multiplier as $B = \sum_{i=0}^{d-1} B_i \alpha^{Di}$, where

$$B_i = \sum_{j=0}^{D-1} b_{Di+j} \alpha^j \;\; 0 \leq i \leq d-1 \tag{2.9}$$

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 $\deg(B) < m$). The multiplication can then be performed as:

$$C \equiv A \cdot B \bmod p(\alpha) = A \cdot \sum_{i=0}^{d-1} B_i \alpha^{Di} \bmod p(\alpha) \tag{2.10}$$

The Least-Significant Digit-serial (LSD) multiplier is a generalization of the LSB multiplier in which the digits of B are processed starting from the least significant to the most significant. Using Eq. 2.10, the product in this scheme can be represented as follows

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

Algorithm 2.8 shows the details of the LSD Multiplier.

Algorithm 2.8 Least Significant Digit-serial (LSD) Multiplier [76]

Input: $A = \sum_{i=0}^{m-1} a_i \alpha^i$, where $a_i \in \mathbb{F}_2$, $B = \sum_{i=0}^{\lceil \frac{m}{D} \rceil - 1} B_i \alpha^{Di}$, where B_i is as in (2.9)
Output: : $C \equiv A \cdot B = \sum_{i=0}^{m-1} c_i \alpha^i$, where $c_i \in \mathbb{F}_2$

1: $C \leftarrow 0$
2: **for** $i = 0$ to $\lceil \frac{m}{D} \rceil - 1$ **do**
3: $\quad C \leftarrow B_i A + C$
4: $\quad A \leftarrow A\alpha^D \bmod p(\alpha)$
5: **end for**
6: Return $(C \bmod p(\alpha))$

Remark. *If C is initialized to value $I \in \mathbb{F}_{2^m}$ in Algorithm 2.8, then we can obtain as output the quantity, $A \cdot B + I \bmod p(\alpha)$ at no additional (hardware or delay) cost. This operation, known as a multiply/accumulate operation is very useful in elliptic curve based systems.*

Field squaring is much simpler in $\mathbb{F}_{2^m}$ when represented in polynomial basis as show here:

$$\begin{aligned} C &\equiv A^2 \bmod F(\alpha) \\ &\equiv (a_{m-1}\alpha^{2(m-1)} + a_{m-2}\alpha^{2(m-2)} + \ldots + a_1\alpha^2 + a_0) \bmod F(\alpha) \end{aligned} \tag{2.11}$$

Polynomial squaring is implemented by expanding C to double its bit-length by interleaving 0 bits in between the original bits of C and then reducing the double length result.

Field Reduction

Field reduction of a $2n$-bit size polynomial in $\mathbb{F}_{2^m}$ can be efficiently performed if the reduction polynomial $F(x)$ is a trinomial or pentanomial, i.e.,

$$F(x) = x^m + x^k + 1$$
or
$$F(x) = x^m + x^j + x^k + x^l + 1$$

Such polynomials are widely recommended in all the major standards [2, 36, 60]. For software implementation, reduction polynomials with the middle terms close to each other are more suitable while for hardware, polynomials with a smaller second highest degree are favorable. The implementation techniques using the different reduction polynomials can be found in [11].

2.5 Elliptic Curve Arithmetic over $\mathbb{F}_{p^m}$

A prime extension field is denoted as $\mathbb{F}_{p^m}$ for p prime and m a positive integer. There exist finite fields for all primes p and positive integers m. The elliptic curve group E over $\mathbb{F}_{p^m}$ (*characteristic* not equal to 2 or 3) is the set of solutions (x, y) which satisfy the simplified Weierstrass equation for $\mathbb{F}_{p^m}$:

$$E : y^2 = x^3 + ax + b \tag{2.12}$$

where $a, b \in \mathbb{F}_{p^m}$ and $4a^3 + 27b^2 \neq 0$, together with the point at infinity $\mathcal{O}$.
The group laws are same as that for $\mathbb{F}_p$ including the point addition and doubling equations. Therefore all the algorithms that are used for $\mathbb{F}_p$ at the elliptic curve level (like projective coordinate point addition and subtraction) can also be used for $\mathbb{F}_{p^m}$.

2.5.1 Field Arithmetic over Optimal Extension Fields (OEF)

$\mathbb{F}_{p^m}$ is isomorphic to $\mathbb{F}_p[x]/(P(x))$, where $P(x)$ is a monic irreducible polynomial of degree m over $\mathbb{F}_p$. The choices of p, m, and $P(x)$ can have a dramatic impact on the implementation performance. In finite fields of special form, specialized algorithms can give better performance than generic algorithms. Optimal Extension Fields (OEF), as introduced by Bailey and Paar [5], are a special family of extension fields which offer considerable computational advantages.

Definition 2.1. *An Optimal Extension Field is a extension field $\mathbb{F}_{p^m}$ such that*

1. *The prime p is a pseudo-Mersenne (PM) prime of the form $p = 2^n \pm c$ with $log_2(c) \leq \lfloor n/2 \rfloor$.*

2. *An irreducible binomial $P(x) = x^m - \omega$ exists over $\mathbb{F}_p$.*

We represent the elements of $\mathbb{F}_{p^m}$ as polynomials of degree at most $m-1$ with coefficients from the subfield $\mathbb{F}_p$, i.e., any element $A \in \mathbb{F}_{p^m}$ can be written as

$$A(t) = \sum_{i=0}^{m-1} a_i \cdot t^i = a_{m-1} \cdot t^{m-1} + \cdots + a_2 \cdot t^2 + a_1 \cdot t + a_0 \quad \text{with } a_i \in \mathbb{F}_p \tag{2.13}$$

where t is the root of $P(x)$ (i.e., $P(t) = 0$). The prime p is generally selected to be a pseudo-Mersenne prime that fits into a single processor word. Consequently, we can store the m coefficients of $A \in \mathbb{F}_{p^m}$ in an array of m single-precision words, represented as the vector $(a_{m-1}, \ldots, a_2, a_1, a_0)$.
The construction of an OEF requires a binomial $P(x) = x^m - \omega$ which is irreducible over $\mathbb{F}_p$. Reference [6] describes a method for finding such irreducible binomials. The specific selection of p, m, and $P(x)$ leads to a fast subfield and extension field reduction, respectively.

Addition and Subtraction

Addition and subtraction of two field elements $A, B \in \mathbb{F}_{p^m}$ is accomplished in a straightforward way by addition/subtraction of the corresponding coefficients in $\mathbb{F}_p$.

$$C(t) = A(t) \pm B(t) = \sum_{i=0}^{m-1} c_i \cdot t^i \quad \text{with } c_i \equiv a_i \pm b_i \bmod p \tag{2.14}$$

A reduction modulo p (i.e., an addition or subtraction of p) is necessary whenever the sum or difference of two coefficients a_i and b_i is outside the range of $[0, p-1]$ (shown in Algorithms 2.1 and 2.2). There are no carries propagating between the coefficients which is an advantage for software implementations.

Multiplication and Squaring

A multiplication in the extension field $\mathbb{F}_{p^m}$ can be performed by ordinary polynomial multiplication over $\mathbb{F}_p$ and a reduction of the product polynomial modulo the irreducible polynomial $P(t)$. The product of two polynomials of degree at most $m-1$ is a polynomial of degree at most $2m-2$.

$$\begin{aligned} C(t) &= A(t) \cdot B(t) = \left(\sum_{i=0}^{m-1} a_i \cdot t^i \right) \cdot \left(\sum_{j=0}^{m-1} b_j \cdot t^j \right) \\ &\equiv \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} (a_i \cdot b_j \bmod p) \cdot t^{(i+j)} = \sum_{k=0}^{2m-2} c_k \cdot t^k \end{aligned} \tag{2.15}$$

There are several techniques to accomplish a polynomial multiplication. The standard algorithm moves through the coefficients b_j of $B(t)$, starting with b_0, and multiplies b_j by any coefficient a_i of $A(t)$. This method, which is also referred to as *operand scanning* technique, requires exactly m^2 multiplications of coefficients $a_i, b_j \in \mathbb{F}_p$. However, there are two advanced multiplication techniques which typically perform better than the standard algorithm. The *product scanning* technique reduces the number of memory accesses (in particular store operations), whereas Karatsuba's algorithm [41] requires fewer coefficient multiplications [6].

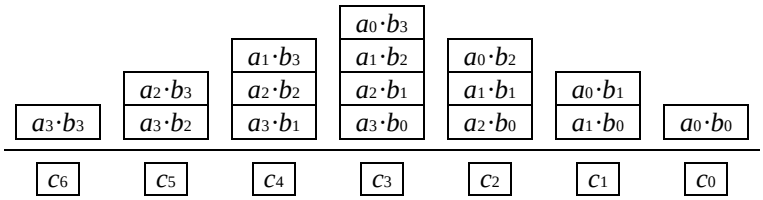


Figure 2.2: Multiply-and-accumulate strategy (m = 4)

The *product scanning* technique employs a "multiply-and-accumulate" strategy [34] and forms the product $C(t) = A(t) \cdot B(t)$ by computing each coefficient c_k of $C(t)$ at a time. Therefore, the coefficient-products $a_i \cdot b_j$ are processed in a "column-by-column" fashion, as depicted in Figure 2.2 (for $m = 4$), instead of the "row-by-row" approach used by the operand scanning technique. More formally, the product $C(t)$ and its

coefficients c_k are computed as follows.

$$C(t) = A(t) \cdot B(t) = \sum_{k=0}^{2m-2} c_k \cdot t^k \quad \text{with} \quad c_k \equiv \sum_{i+j=k} a_i \cdot b_j \bmod p \quad (0 \leq i, j \leq m-1) \tag{2.16}$$

The product scanning technique requires exactly the same number of coefficient multiplications as its operand scanning counterpart (namely m^2), but minimizes the number of store operations since a coefficient c_k is only written to memory after it has been completely evaluated. In general, the calculation of coefficient-products $a_i \cdot b_j$ and the reduction of these modulo p can be carried out in any order. However, it is usually advantageous to compute an entire column sum first and perform a single reduction thereafter, instead of reducing each coefficient-product $a_i \cdot b_j$ modulo p. The former approach results in m^2 reduction operations, whereas the latter requires only one reduction per coefficient c_k, which is $2m-1$ reductions altogether.

When $A(t) = B(t)$, the coefficient-products of the form $a_i \cdot b_j$ appear once for $i = j$ and twice for $i \neq j$. Therefore squaring of a polynomial $A(t)$ of degree $m-1$ can be obtained with only $m \cdot (m+1)/2$ coefficient multiplications

Subfield reduction

An integral part of both polynomial multiplication and polynomial squaring is the subfield reduction which is the reduction of a coefficient-product (or a sum of several coefficient-products) modulo the prime p. Pseudo-Mersenne primes are a family of numbers highly suited for modular reduction due to their special form [16]. They allow to employ very fast reduction techniques that are not applicable to general primes. The efficiency of the reduction operation modulo a PM prime $p = 2^n - c$ is based on the relation

$$2^n \equiv c \bmod p \quad (\text{for } p = 2^n - c) \tag{2.17}$$

which means that any occurrence of 2^n in an integer $z \geq 2^n$ can be substituted by the much smaller offset c. To give an example, let us assume that z is the product of two integers $a, b < p$, and thus $z < p^2$. Furthermore, let us write the $2n$-bit product z as $z_H \cdot 2^n + z_L$, whereby z_H and z_L represent the n most and least significant bits of z, respectively. The basic reduction step is accomplished by multiplying z_H and c together and "folding" the product $z_H \cdot c$ into z_L.

$$z = z_H \cdot 2^n + z_L \equiv z_H \cdot c + z_L \bmod p \quad (\text{since } 2^n \equiv c \bmod p) \tag{2.18}$$

This leads to a new expression for the residue class with a bit-length of at most $1.5n$ bits. Repeating the substitution a few times and performing final subtraction of p yields the fully reduced result $x \bmod p$. A formal description of the reduction modulo $p = 2^n - c$ is given in Algorithm 2.9.

Algorithm 2.9 Fast reduction modulo a pseudo-Mersenne prime $p = 2^n - c$ with $\log_2(c) \leq n/2$

Input: n-bit modulus $p = 2^n - c$ with $\log_2(c) \leq n/2$, operand $y \geq p$.
Output: Residue $z \equiv y \bmod p$.

1: $z \leftarrow y$
2: **while** $z \geq 2^n$ **do**
3: $\quad z_L \leftarrow z \bmod 2^n$ $\quad \{$ the n least significant bits of z are assigned to z_L $\}$
4: $\quad z_H \leftarrow \lfloor z/2^n \rfloor$ $\quad \{$ z is shifted n bits to the right and assigned to z_H $\}$
5: $\quad z \leftarrow z_H \cdot c + z_L$
6: **end while**
7: **if** $z \geq p$ **then** $z \leftarrow z - p$ **end if**
8: **return** z

Finding the integers z_L and z_H is especially easy when n equals the word-size of the target processor. In this case, no bit-level shifts are needed to align z_H for the multiplication by c.

Extension field reduction

Polynomial multiplication and squaring yields a polynomial $C(t)$ of degree $2m - 2$ with coefficients $c_k \in \mathbb{F}_p$ after subfield reduction. This polynomial must be reduced modulo the irreducible polynomial $P(t) = t^m - \omega$ in order to obtain the final polynomial of degree $m - 1$. The extension field reduction can be accomplished in linear time since $P(t)$ is a monic irreducible binomial. Given $P(t) = t^m - \omega$, the following congruences hold: $t^m \equiv \omega \bmod x(t)$. We can therefore reduce $C(t)$ by simply replacing all terms of the form $c_k \cdot t^k$, $k \geq m$, by $c_k \cdot \omega \cdot t^{k-m}$, which leads to the following equation for the residue:

$$\begin{aligned} R(t) &\equiv C(t) \bmod P(t) \\ &\equiv \textstyle\sum_{l=0}^{m-1} r_l \cdot t^l \quad \text{with} \quad r_{m-1} = c_{m-1} \\ &\qquad \text{and} \quad r_l \equiv (c_{l+m} \cdot \omega + c_l) \bmod p \quad \text{for} \quad 0 \leq l \leq m-2 \end{aligned} \tag{2.19}$$

The entire reduction of $C(t)$ modulo the binomial $P(t) = t^m - \omega$ costs at most $m - 1$ multiplications of coefficients c_k by ω and the same number of subfield reductions [5].

In summary, the straightforward way of multiplying two elements in OEF requires $m^2 + m - 1$ coefficient multiplications and $3m - 2$ reductions modulo p. Special optimizations, such as Karatsuba's method or the "interleaving" of polynomial multiplication and extension field reduction, allow to minimize the number of subfield operations (see [34] for details).

Inversion

Inversion in an OEF can be accomplished either with the extended Euclidean algorithm or via a modification of the Itoh-Tsujii algorithm (ITA) [39], which reduces the problem of extension field inversion to subfield inversion [5]. The ITA computes the inverse of an element $A \in \mathbb{F}_{p^m}$ as

$$A^{-1}(t) \equiv (A^r(t))^{-1} \cdot A^{r-1}(t) \bmod P(t) \quad \text{where} \quad r = \frac{p^m - 1}{p - 1} = p^{m-1} + \cdots + p^2 + p + 1 \tag{2.20}$$

Efficient calculation of $A^{r-1}(t)$ is performed by using an addition-chain constructed from the p-adic representation of $r - 1 = (111 \ldots 110)_p$. This approach requires the field elements to be raised to the p^i-th powers, which can be done with the help of the i-th iterate of the Frobenius map [6]. The other operation is the inversion of $A^r(t) = A^{r-1}(t) \cdot A(t)$. Computing the inverse of $A^r(t)$ is easy due to the fact that for any element $\alpha \in \mathbb{F}_{p^m}$, the r-th power of α, i.e., $\alpha^{(p^m-1)/(p-1)}$ is always an element of the subfield $\mathbb{F}_p$. Thus, the computation of $(A^r(t))^{-1}$ requires just an inversion in $\mathbb{F}_p$ which can be done using a single-precision variant of the extended Euclidean algorithm.

In summary, the efficiency of the ITA in an OEF relies mainly on the efficiency of the extension field multiplication and the subfield inversion (see [6, 34]).

2.6 Elliptic Curve Point Multiplication

The *scalar point multiplication* (an operation of the form $k \cdot P$ where k is a positive integer and P is a point on the elliptic curve) is the most basic operation in the implementation of an elliptic curve cryptosystem. There are different ways to implement point multiplication: binary, m-ary and sliding window methods, signed digit representation and combination of these methods as described in [23] and [11]. A comparison of these methods can be found in [29] and [33].

2.6.1 Binary Method

The most simplest and straightforward implementation is the binary method (as shown in Algorithm 2.10) which is similar to the square-and-multiply algorithm for exponentiation [42]. We use the Most-Significant Bit (MSB) first variant, since the point P can be kept fixed (Step 4 Algorithm 2.10) enabling mixed co-ordinates for point addition to be used. The expected running time for binary method is $\frac{m}{2} \cdot \mathbf{A} + m \cdot \mathbf{D}$ (where $\mathbf{A}$ is a point Addition, $\mathbf{D}$ is a point Doubling).

Algorithm 2.10 MSB binary method for point multiplication

Input: P, k, where $P \in E(K), k \in \mathbf{Z}^+$ and $\log_2 k < m$
Output: $Q = k \cdot P$, where $Q \in E(\mathbb{F}_{2^m})$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = m - 1$ down to 0 **do**
3: $\quad Q \leftarrow 2Q$.
4: $\quad$ If $k_i = 1$ then $Q \leftarrow Q + P$
5: **end for**
6: Return(Q)

2.6.2 Non-Adjacent Form (NAF) Method

This method uses the fact that point inverses are quite inexpensive to calculate (see elliptic curve group operations). Therefore a signed digit representation of $k = \sum_{i=0}^{t-1} k_i 2^i$, where $k_i \in -1, 0, 1$ is used for binary method. Non-Adjacent Form (NAF) is a more efficient signed digit representation in which no adjacent k_i's are nonzero. The NAF for a positive integer is calculated as shown in the Algorithm 2.11.

The advantage of NAF is that on the average $t/3$ terms of k are nonzero [59]. Hence, NAF point multiplication (shown in Algorithm 2.12) has an expected running time of $\frac{m}{3} \cdot \mathbf{A} + m \cdot \mathbf{D}$.

2.6.3 Windowing Methods

Windowing methods are useful when memory is available for storing pre-computed points (which is especially useful if the point is fixed) and hence speed up the point multiplication. The window-NAF and fixed-base Comb methods are some of the efficient windowing based point multiplication algorithms. However for constrained envi-

Algorithm 2.11 Computation of NAF of a positive integer

Input: k, a positive integer
Output: NAF(k)

1: $i \leftarrow 0$
2: **while** $k \geq 1$ **do**
3: If k is odd then $k_i \leftarrow 2 - (k mod 4)$, $k \leftarrow k - k_i$;
4: Else $k_i \leftarrow 0$
5: $k \leftarrow k/2$, $i \leftarrow i + 1$.
6: **end while**
7: Return$(k_{i-1}, k_{i-2}, \cdots, k_1, k_0)$.

Algorithm 2.12 Binary NAF method for point multiplication

Input: P, k, where $P \in E(K), k \in \mathbf{Z}^+$ and $\log_2 k < m$
Output: $Q = k \cdot P$.

1: Compute NAF$(k) = \sum_{i=0}^{l-1} k_i 2^i$. { Using Algorithm 2.11 }
2: $Q \leftarrow \mathcal{O}$
3: **for** $i = l - 1$ down to 0 **do**
4: $Q \leftarrow 2Q$.
5: If $k_i = 1$ then $Q \leftarrow Q + P$
6: If $k_i = -1$ then $Q \leftarrow Q - P$
7: **end for**
8: Return(Q)

ronments, where memory is sparse, such methods are not very attractive. A detailed description of the algorithms can be found in [34].

2.6.4 Montgomery Method

In [52], a Montgomery method for point multiplication in $E(\mathbb{F}_{2^m})$ is introduced which can be implemented in affine or projective co-ordinates. This method uses the fact that the x-coordinate of the point $P + Q$ can be derived from the x-coordinates of P, Q and $P - Q$. An iteration is setup to calculate the x-coordinates of kP and $(k+1)P$ from which the y-coordinate of kP is derived. Algorithm 2.13 shows the Montgomery point multiplication in the standard projective co-ordinates. This algorithm computes the point double and point addition in each iteration independent of k_i. Therefore the expected running time for this algorithm is $(6M + 5S + 3A)m + (1I + 10M)$.

Algorithm 2.13 Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ standard projective co-ordinates

Input: P, k, where $P = (x_1, y_1) \in E(\mathbb{F}_{2^m}), k \in \mathbf{Z}^+$ and $\log_2 k < m$
Output: $Q = k \cdot P$, where $Q = (x_3, y_3) \in E(\mathbb{F}_{2^m})$
1: $X_1 \leftarrow x_1,\ Z_1 \leftarrow 1,\ X_2 \leftarrow x_1^4 + b,\ Z_2 \leftarrow x_1^2$.
2: **for** $i = m - 2$ downto 0 **do**
3: **if** $k_i = 1$ **then**
4: $T \leftarrow Z_1,\ Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2,\ X_1 \leftarrow x_1 Z_1 + X_1 X_2 T Z_2$.
5: $T \leftarrow X_2,\ X_2 \leftarrow X_2^4 + b Z_2^4,\ Z_2 \leftarrow T^2 Z_2^2$
6: **else**
7: $T \leftarrow Z_2,\ Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2,\ X_2 \leftarrow x_1 Z_2 + X_1 X_2 Z_1 T$.
8: $T \leftarrow X_1,\ X_1 \leftarrow X_1^4 + b Z_1^4,\ Z_1 \leftarrow T^2 Z_1^2$
9: **end if**
10: **end for**
11: $x_3 \leftarrow X_1 / Z_1$
12: $y_3 \leftarrow (x_1 + X_1/Z_1)[(X_1 + x_1 Z_1)(X_2 + x_1 Z_2) + (x_1^2 + y)(Z_1 Z_2)](x_1 Z_1 Z_2)^{-1} + y_1$
13: Return (Q)

2.7 Elliptic Curve Key Exchange and Signature Protocols

The elliptic curve scalar point multiplication is used as the basic operation for constructing the ECDLP variants of popularly used DLP protocols like Diffie-Hellman (DH) key exchange [18] and Digital Signature Algorithm (DSA) [60].

2.7.1 Elliptic Curve Diffie-Hellman Key Exchange

In the *Elliptic Curve Diffie-Hellman* (ECDH) key exchange, the two communicating parties server S and client C agree in advance to use the same curve parameters and base point G. They each generate their random private keys Pr_S and Pr_C, respectively, and then compute their corresponding public keys $Pu_S = Pr_S \cdot G$ and $Pu_C = Pr_C \cdot G$.

To perform a key exchange, both the client and server first exchange their public keys. On receiving the other party's public key, each of them multiply their private key with the received public key to derive the required common shared secret:

$$Pr_C \cdot Pu_S = Pr_S \cdot Pu_C = Pr_S \cdot Pr_C \cdot G.$$

An attacker cannot determine this shared secret from the curve parameters, G or the public keys of the parties based on the hard problem, ECDLP. Normally, certified

public keys are used to maintain integrity and prevent a man-in-the-middle attack.

2.7.2 Elliptic Curve Digital Signature Algorithm

This algorithm is analogues to the Digital Signature Algorithm(DSA) for the DL systems. Signature generation for the hash of a message m and the verification of the signature are shown in Algorithm 2.14 and Algorithm 2.15, respectively. As in ECDH, both the communicating parties need to agree on the same elliptic curve parameters.

Algorithm 2.14 ECDSA signature generation

Input: Public point P in $E(K)$ of order n, private key d and message m.
Output: Signature (r, s)

1: Select a $k \in [1, n-1]$.
2: Compute $kP = (x_1, y_1)$ and convert x_1 to an integer $\hat{x}_1$.
3: Compute $r = \hat{x}_1 \bmod n$. If $r = 0$ then repeat from step 1.
4: Compute $e = H(m)$.
5: Compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ then repeat from step 1.
6: Return $((r, s))$

Algorithm 2.15 ECDSA signature verification

Input: Public point P in $E(K)$ of order n, public key Q, message m and signature (r, s).
Output: **Accept** or **Reject signature**

1: Verify if $r, s \in [1, n-1]$; else Return("**Reject signature**").
2: Compute $e = H(m)$.
3: Compute $w = s^{-1} \bmod n$.
4: Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5: Compute $X = u_1P + u_2Q = (x_1, y_1)$.
6: If $X = \mathcal{O}$ then Return("**Reject signature**");
7: Convert x_1 to an integer $\hat{x}_1$; Compute $v = \hat{x}_1 \bmod n$.
8: If $v = r$ then Return("**Accept signature**"); Else Return("**Reject signature**");

Chapter 3

Software Design: ECDH Key Exchange on an 8-bit Processor

We present here the results of the collaborative work with Sun Microsystems which was published in part in [46] and the wireless card-reader application demonstrated at SunNetwork 2003.

3.1 Motivation and Outline

The main aim of this work was to prove that public-key cryptography can indeed be used on low-end 8-bit processors (which represents the normal computational capacity of most low-end sensor networks) without the need for any extra hardware such that it provides adequate security for establishing the secret keys required for a secure wireless connection. This goal is achieved using the computational savings and the flexibility of the *Elliptic Curve Cryptography* over the special Optimal Extension Fields (OEF). A further objective was also to demonstrate the ability to setup an end-to-end secure channel between two communicating devices (across the wired and wireless domains) using the public key cryptographic implementation.

This chapter is structured as follows: In Section 3.2, we mention the previous related work. The architecture of our implementation environment *Chipcon* is described in Section 3.3. Section 3.4 shows the basis of the different elliptic curve parameter selections in particular the OEF, and Section 3.5 we describe the implementation aspects of the elliptic curve arithmetic. Section 3.6 describes the communication protocol followed by the description of the demonstration application in Section 3.7.

3.2 Related Work

ECC on an 8-bit processor have been reported in [14] and [80], both implemented over Optimal Extension Fields. In [14], the ECC implementation is over the field $\mathbb{F}_{p^m}$ with $p = 2^{16} - 165,\ m = 10$, and irreducible polynomial $f(x) = x^{10} - 2$. A performance of 122 msec at 20 Mhz is reported for a 160-bit point multiplication using the math co-processor for the sub-field multiplication. [80] implements ECC over $\mathbb{F}_{(2^8-17)^{17}}$ on an 8051 micro-controller without co-processor, but instead uses the internal 8-by-8-bit integer multiplier. The authors achieve a speed of 1.95 sec for a 134-bit fixed point multiplication using 9 pre-computed points and 8.37 sec for a general point multiplication using binary method of exponentiation.

3.3 The Chipcon Architecture

The platform chosen for the implementation is the Chipcon CC1010 chip [13]. It consists of an 8-bit 8051 processor core with a built-in radio transceiver and a hardware DES engine. The CC1010 has an optimized processor core which execute one instruction cycle every four clock cycles, offering roughly 2.5 times the performance of the original Intel 8051. The 8051 processor is widely used for low-cost applications and represents the normal computational capacity of a low-end device. The built-in radio transceiver and DES engine makes this chip an ideal platform for our proof-of-concept. Chipcon is also a very power efficient device which allows for use in mobile devices.

The CC1010 contains 32 kilobytes of flash memory for storing programs, 2048 bytes of SRAM external to the 8051 core (XRAM), and 128 bytes of internal SRAM (IRAM). The chip is clocked at 3.68 Mhz for our implementation and the wireless communication is done over the 868 Mhz radio frequency.

3.4 Elliptic Curve Parameter Selection

The choice of the finite field is dictated by the characteristics of the implementation platform. Multi-precision integer arithmetic needed in $\mathbb{F}_p$ are costly to implement in software due to the carries involved. In an 8-bit processor, the cost is further increased due to the fact that an 8-bit word size translates to a more number of words required to represent each field element and hence more number of carries. Binary fields $\mathbb{F}_{2^m}$ are also hard to implement in software due to the lack of an in-built $\mathbb{F}_2$ multiplier.

However, extension fields $\mathbb{F}_{p^m}$ are incredibly suitable for a software implementation because by a proper choice of parameters we can avoid multi-precision arithmetic that is required in $\mathbb{F}_p$. The basic idea behind the special extension field OEFs (as described in Section 2.5.1) is to select the prime p, the extension degree m, and the irreducible polynomial $x(t)$ to closely match the underlying hardware characteristics. In concrete, p has to be selected to be a pseudo-Mersenne prime with a bit-length of less than but close to 8-bits (the word size of the 8051 processor), so that all subfield operations can be conveniently accomplished with the 8051's arithmetic instructions. Based on this constraint, we chose for our implementation the prime as $p = 2^8 - 17$ and the irreducible polynomial as $P(x) = x^{17} - 2$, i.e., $m = 17$.

The field $\mathbb{F}_{(2^8-17)^{17}}$ provides a security level of 134 bits. Lenstra and Verheul [51] showed that under certain assumptions, 952-bit RSA and DSA systems may be considered equivalent in security to a 132-bit ECC system. Therefore the proposed system is far more secure than a 512-bit RSA system which has been popular for smart card applications till recently. This security level is appropriate to protect data for a medium time interval, say one year, and is sufficient for most embedded applications.

3.5 Implementation aspects on the Chipcon processor

There are two dominant performance constraints that determine the efficiency of the elliptic curve cryptosystem implementation: the efficiency of the scalar multiplication and the efficiency of the arithmetic in the underlying finite field.

3.5.1 Field Arithmetic

A field element $A \in \mathbb{F}_{(2^8-17)^{17}}$ is stored in memory as a vector of 17 byte-words $(a_{16}, \cdots, a_1, a_0)$. The field elements are stored in the external RAM (XRAM) and moved into the internal RAM (IRAM) whenever an arithmetic operation is performed on them. This is necessary since processor instructions using XRAM data are more costly (requiring more cycles) than those on IRAM data.
Here, we briefly outline the implementation of addition (subtraction), multiplication, squaring, and inversion in an OEF based on the mathematical background discussed in Section 2.5.1.

Addition and Subtraction

Addition and subtraction of two field elements is done by using the 8051's integer addition/subtraction instruction on the corresponding coefficients. A reduction modulo p is performed only if the carry is generated, i.e., the result is outside the range $[\,0, 2^8)$. Therefore, reduction steps are not performed each time as shown in Eq. 2.14, but are done only if the result cannot be stored within a single byte-word.

Multiplication and Squaring

The subfield multiplication is performed using the 8051's 8-by-8 bit integer multiplier instruction. The double-sized result coefficients generated during subfield multiplication represented as $c = c_1\ 2^8 + c_0$, where $c_0, c_1 < 2^8$ is reduced using the Eq. 2.17. Thus, the equivalence

$$c \equiv 17c_1 + c_0 \bmod (2^8 - 17)$$

leads to a very efficient reduction which requires just one multiplication by 17, one addition, and no division or inversions.

The multiplication in the extension field $\mathbb{F}_{(2^8-17)^{17}}$ is performed by polynomial multiplication in product scanning method. The extension field reduction for the double-sized polynomial is done using the relation $x^{17} \equiv 2 \bmod (x^{17} - 2)$. We can therefore represent the field multiplication implementation as:

$$\begin{aligned} C(t) &\equiv A(t) \cdot B(t) \bmod P(t) \\ &\equiv \hat{c}_{16}t^{16} + (2\hat{c}_{32} + \hat{c}_{15})t^{15} + \ldots + (2\hat{c}_{18} + \hat{c}_1)t + (2\hat{c}_{17} + \hat{c}_0) \bmod (t^{17} - 2) \\ &\quad \text{where } \ \hat{c}_i \equiv \textstyle\sum_{j+k=i} a_j b_k \bmod (2^8 - 17). \end{aligned}$$

Squaring is similarly performed except that the polynomial multiplication is much more easier because of repeating terms.

Inversion

Inversion is accomplished using the modification of the Itoh-Tsujii algorithm (ITA) described in Section 2.5.1, which reduces the problem of extension field inversion to subfield inversion. The implementation for ITA in $\mathbb{F}_{(2^8-17)^{17}}$ is as shown in Algorithm 3.1.

Due to the extreme limits in terms of memory capacity and processing power, the field operations were implemented in pure assembly for efficiency. This also allowed a fine grain control on the usage of the internal RAM. The performance for the arithmetic

Algorithm 3.1 Inversion with modified Itoh-Tsuji method over $\mathbb{F}_{(2^8-17)^{17}}$

Input: $A \in \mathbb{F}_{(2^8-17)^{17}}$ and $P(t) = t^{17} - 2$.

Output: $B \equiv A^{-1} \bmod P(t) = (A^r)^{-1}A^{r-1}$ where $r = (p^{17}-1)/(p-1) = (111...1)_p$.

1: $B_0 \leftarrow A^p = A^{(10)_p}$ { Frobenius table lookup }
2: $B_1 \leftarrow B_0 \cdot A = A^{(11)_p}$
3: $B_2 \leftarrow B_1^{p^2} = A^{(1100)_p}$ { Frobenius table lookup }
4: $B_3 \leftarrow B_2 \cdot B_1 = A^{(1111)_p}$
5: $B_4 \leftarrow B_3^{p^4} = A^{(11110000)_p}$ { Frobenius table lookup }
6: $B_5 \leftarrow B_4 \cdot B_3 = A^{(11111111)_p}$
7: $B_6 \leftarrow B_5^{p^8} = A^{(1111111100000000)_p}$ { Frobenius table lookup }
8: $B_7 \leftarrow B_6 \cdot B_5 = A^{(1111111111111111)_p}$
9: $B_8 \leftarrow B_7^{p} = A^{(1111111111111110)_p}$ { Frobenius table lookup }
10: $b \leftarrow B_8 \cdot A = A^r$
11: $b \leftarrow b^{-1}$ { subfield inverse }
12: $B \leftarrow b \cdot B_8 = (A^r)^{-1}A^{r-1}$
13: **Return** B

operations are shown in Table 3.1. The inverse operation code size is shown to be negligible as it involves only calls to the multiplication function.

Table 3.1: Field arithmetic performance on Chipcon (@3.68 Mhz)

Description	**Operation**	**Time** (μsec)	**Code size** (bytes)
Multiplication	$A(t)B(t)$	5093	5212
Squaring	$A^2(t)$	3142	3400
Inversion	$A^{-1}(t)$	24672	neg.

3.5.2 Point Arithmetic

The overall number of field additions, multiplications, and inversions for the point arithmetic depends heavily on the chosen coordinate system (see Table 2.1). Due to

the relatively low complexity of the inversion operation, the ratio for inversion time to multiplication time for our implementation (as shown in Table 3.1) is just 4.8 : 1. Therefore we choose to use affine coordinates for our point representation. Using affine coordinates also has the advantage that we require lesser amount of memory for storing temporary values, which was especially attractive for our low-cost device where limited memory resources are available.

The point addition and doubling is done as shown in Eq. 2.3. For point multiplication we use the binary double-and-add method as described in Section 2.6.1. The points are stored in XRAM, and the appropriate co-ordinates are moved to IRAM when a field operation is performed on it. The performance for the point operations are shown in Table 3.2. It takes 2.99 seconds to complete an elliptic curve point multiplication on the Chipcon platform running at 3.68 Mhz.

Table 3.2: ECC point arithmetic performance on Chipcon (@3.68 Mhz)

Operation	**Time** (msec)
Point Addition	15.395
Point Doubling	14.049
Point Multiplication	2999.8

3.6 Communication Protocol

Figure 3.1 depicts a possible application scenario. Here, a wireless *client* needs to establish a secure communication channel to a *server* on the internet using the *gateway* which bridges the wired and the wireless channel. Normally, this secure communication is achieved using a *trusted gateway* which creates two different secure channels: one between the *client* and the *gateway* (usually with a symmetric key K_2), and the second between the *gateway* and the *server* (using standard public key K_1). This is a compromise made, assuming the constrained wireless *clients* are not capable enough to run the standard public key cryptographic algorithms.

However, based on our implementation of ECC which is suitable even for constrained devices, we can now present a communication protocol to establish end-to-end security between the *client* and the *server* without the need for a *secure gateway*. Our

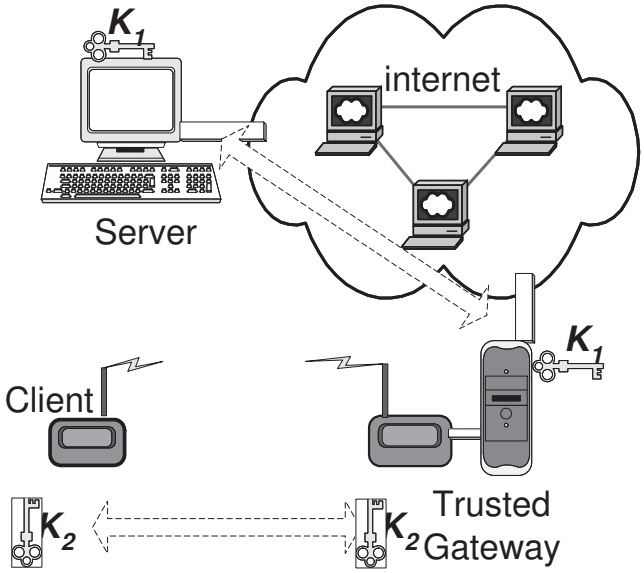


Figure 3.1: Network security based on a trusted gateway.

communication protocol consists of two phases as shown in Figure 3.2. The first one is the *key establishment phase*, which is done initially to exchange the keys using the *Elliptic Curve Diffie-Hellman* (ECDH) protocol (described in Section 2.7.1). Thereafter in *normal mode*, application data is transmitted.

3.6.1 Key Establishment Phase

The client initiates a connection by relaying a *Client Hello* with its pre-computed public key Pu_{cli} on the wireless channel. The gateway, which is in receive mode, on receiving the public key passes it to the server through the wired interface and waits for the server's public key. The server daemon, upon receiving the connection request sends a *Server Hello* with its public key Pu_{ser} to the gateway, and then starts computing the shared secret from the received public key Pu_{cli} and the server's secret key Pr_{ser}.

$$MSecret = Pr_{ser} \cdot Pu_{cli}$$

The gateway transmits the server's public key to the client and waits for the data transmission to begin. The client, on receiving the server's public key (Pu_{ser}) from the gateway, computes the *master secret*.

$$MSecret = Pr_{cli} \cdot Pu_{ser}$$

The client and server now have the same shared master secret *MSecret* of 134 bits using the ECDH key exchange. The key for the symmetric DES operation is then derived from this 134-bit secret.

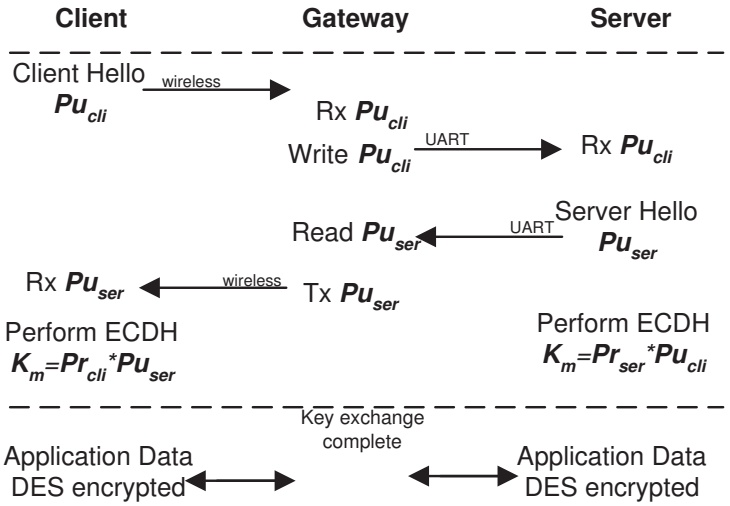


Figure 3.2: Key exchange protocol.

3.6.2 Normal Mode

Once the keys are set up, the client and the server can transmit the application data encrypted with Triple-DES in CFB mode. To close the connection securely, we use a *close connection* control message which deletes the previously generated keys.

3.7 Demonstration Application

We showed the advantage of our ECC implementation with a real-world application of a mobile wireless card reader, which needs to setup an end-to-end secure connection to a server. We used two Chipcon CC1010 evaluation modules for our demonstration setup. One of them is used as a *client*, which is connected to a portable magnetic stripe reader used normally as a point-of-sale terminal. The reader presents the encoded data through an RS-232 serial link to the CC1010. The other evaluation module is used as a *gateway*. It listens to the wireless channel, and transmits the data to a PC acting as

a *server* connected through a RS-232 serial link. The *client* device operates on three 1.5V AA batteries, and communicates with the server through the gateway on a radio frequency of 868 Mhz.

The optimized assembly implementation of ECC runs on the *client*. The wireless communication protocol on the *client* and the *gateway* was written in C and cross-compiled using Small-Devices C Compiler (SDCC) [73] tools. On the *server* side, ECC algorithms were implemented with OpenSSL [62] and the Number Theory Library (NTL) [74].

An exchange begins when a user swipes a card with a magnetic stripe, such as a credit card, on the *client* device. The *client* first saves the data encoded on the magnetic stripe, and then initiates the key exchange protocol described in Section 3.6. After the 134-bit shared secret is established, we use the first 112 bits as the secret key for the Triple-DES engine. The card data is then encrypted with Triple-DES in CFB mode on the client, and decrypted and displayed on the server side. The wireless range of this demonstration exceeded 100ft indoors. This is another strong argument for good security in these devices. The client can operate on the same set of batteries for at least 10 hours continuously, and longer battery life can be achieved if additional power management software is used.

The overall session setup time for the secure connection took 3.15 sec. The breakdown of memory usage for the implementation is shown in Table 3.3. The code size of the elliptic curve point multiplication library is 13.5 kilobytes, while the overall demonstration program occupied 22.5 kilobytes. The total RAM used also includes variables for the field arithmetic operations, storage of temporary points and the secret key (an integer coefficient), and buffers for the communication protocol.

Table 3.3: Memory map for the wireless reader implementation on CC1010

Type	**Size** (bytes)	**Function**
Code	13.5k	ECC
	9k	RF protocol
Internal RAM	128	finite field arithmetic
External RAM	406	temporary points
	34	coefficients

3.8 Summary

We showed here that security protocols based on public key cryptographic algorithms are possible even on low-end wireless devices, without the need for any additional extra hardware. We implemented a medium security ECC on an 8-bit 8051 processor, which represents the normal computational power present in such low-end devices. A complete key exchange protocol was completed in 3.15 sec, which is acceptable, considering the additional end-to-end security it enables. A working model of a wireless card reader communicating with a server was implemented to prove the concept in practice.

Chapter 4

Hardware/Software Co-design: Extensions for an 8-bit Processor

We present here results of the work which was in part published in [47].

4.1 Motivation and Outline

High-volume, low-cost, and very small power budgets of pervasive devices implies they have limited computing power, often not exceeding an 8-bit processor clocked at a few Mhz. Under these constraints, secure *standardized* public-key cryptography for authentication are nearly infeasible in software, or are too slow to be used within the constraints of a communication protocol. On the other hand, public-key cryptography offers major advantages when designing security solutions in pervasive networks. An alternative, is to use a cryptographic co-processor used in high security applications like smart-cards, but its downside are considerable costs (in terms of power and chip area) which makes it unattractive for many of the cost sensitive pervasive applications. In addition, a fixed hardware solution may not offer the cryptographic flexibility (i.e., change of parameters or key length) that is required in real-world applications. An instruction set extension (ISE) is a more viable option, because of the smaller amount of additional hardware required and its flexibility. The efficiency of an ISE is not just measured by the speed-up it achieves, but also in the decrease in code-size, data-RAM, and power consumption. We investigate here, with the use of a reconfigurable hardware attached to an 8-bit processor, to obtain reliable cost/benefit estimates for the proposed extensions.

The chapter is organized as follows: Section 4.2 discusses related work in this field. In Section 4.3, we describe FPSLIC, which is the development platform that we use.

In Section 4.4, the implementation of the arithmetic operations without extensions is showed to determine the peak performance possible, and then discuss the bottlenecks. Section 4.5, presents the proposed extensions along with the implementation and results.

4.2 Related Work

There has been considerable work on efficient implementation of ECC in software for higher-end processors [72, 17, 26, 33]. We list here only the more constrained environment implementations.

An ECDSA implementation on a 16-bit microcomputer M16C, running at 10 Mhz, is described in [35]. The authors propose the use of a field $\mathbb{F}_p$ with prime characteristic $p = e2^c \pm 1$, where e is an integer within the machine size, and c is a multiple of the machine word size. The implementation uses a 31-entry table of precomputed points to generate an ECDSA signature in 150 msec, and performs ECDSA verification in 630 msec. A scalar multiplication of a random point takes 480 msec. The authors in [26] describe an efficient implementation of ECC over $\mathbb{F}_p$ on the 16-bit TI MSP430x33x family of low-cost micro-controllers running at 1 Mhz. A scalar point multiplication over $\mathbb{F}_{2^{128}-2^{97}-1}$ is performed in 3.4 sec without any precomputation. [78] describes an implementation on a Palm Pilot running a Dragonball processor. They use a combination of 16 and 32-bit processing to implement standardized 163-bit NIST curves [60]. A maximum performance for Koblitz curves with precomputation for ECDSA signature verification is 2.4 sec, and ECDSA signature generation is 0.9 sec.

The other approach, as mentioned previously, has been to add a crypto co-processor to these micro-controllers. A survey of commercially available co-processors can be found in [32]. However, a full-size ECC co-processor may be prohibitively expensive for many pervasive applications. Hardware assistance in terms of Instruction Set Extensions (ISE) is more favorable as the cost of extra hardware is quite negligible compared to the whole processor. Previous attempts in this direction [19, 40] are only reported for ECC with not more than 133-bits.

4.3 The FPSLIC Architecture

The development platform used is the AT94K family of FPSLIC (Field Programmable System Level Integrated Circuits) device [4]. This architecture integrates an AVR 8-

bit micro-controller core, FPGA resources, several peripherals, and 36K bytes SRAM within a single chip. If the FPGA part is left unused, it functions for all practical purposes as an AVR micro-controller, which is used widely in smart-cards and sensor networks. The platform is appealing for hardware/software co-design and suited for System-on-Chip (SoC) implementations.

We use the FPGA to specifically show the effects of additional hardware extensions. However, we view the reconfigurability not only useful for prototyping purposes, but a small reconfigurable hardware extension is also an attractive platform for embedded devices, since the extension can offer many speed and power benefits for computationally intensive applications, as demonstrated in this chapter. It should also be noted that public-key operations are typically only needed at the initial or final stage of a communication session. Hence, it is perceivable that the ISE can be runtime reconfigured for other applications, when public-key operations are completed.

The AVR micro-controller is a RISC processor with an 8-bit data bus and 16-bit address bus. It has 31, 8-bit registers, each connected directly to the ALU. Six of these registers can also be used as address pointers. Almost all arithmetic and logical instructions execute in 1 clock cycle. The SRAM is divided into 20K bytes program memory, 4K bytes data memory, and 12K bytes for dynamic allocation as data or program memory. The implementations are done on the ATSTK94 FPSLIC demonstration board clocked at 4 Mhz.

4.4 Implementation aspects on the AVR processor

An efficient ISE implementation requires a tightly coupled hardware and software co-design. In a first step, we build an assembly optimized software-only implementation of ECC, to identify the functional elements and code-segments that would provide efficiency gains if implemented as an ISE. Then, a hardware model of the modified processor is used to determine the effects of the new extensions especially the running time, code-size, and data-RAM usage.

We choose the standardized 163-bit elliptic curve over $\mathbb{F}_{2^m}$ recommended in the NIST [60] and ANSI [2] standards for our implementation. We use the efficiency of the scalar point multiplication, $k \cdot P$ over this curve for determining the benefits of the ISE. First the pure software implementation is done, the arithmetic for which has been described in detail in Section 2.4.1. Here, we give only a brief overview of the implementation.

4.4.1 Field Arithmetic

Since we use the polynomial basis representation with the reduction polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, an element in $\mathbb{F}_{2^{163}}$ is represented as an array of 21 byte words in memory, with the five last most-significant bits being ignored.

Field Addition

Addition is the simplest of all operations, since it is a bitwise addition in $\mathbb{F}_2$, which directly maps to word-level XOR operation in software. Since such an XOR instruction exists on the processor, the addition can be done as 21 word-by-word operations.

Field Multiplication and Squaring

Multiplication is performed in two steps: a polynomial multiplication, followed by reduction modulo $F(x)$. As mentioned in Section 2.4.1, the polynomial multiplication can be most efficiently implemented in software using the Comb method. The implementation is done as shown in Algorithm 2.7, with $w = 8$ and $m = 163$ (giving $s = 21$) as parameters for the 8-bit AVR microprocessor.

Squaring is similarly done in two steps: a polynomial squaring, followed by reduction modulo $F(x)$. Polynomial squaring is a simple expansion by interleaving 0 bits as shown in Eq. 2.11. The simplest and efficient way to implement this in software is using a precomputed table. We use a 512 byte table to convert each byte to its 2 byte expansion.

Field Reduction

Multiplication and squaring algorithms require a second step, in which a polynomial of degree not greater than 324 is reduced to a polynomial of degree 162. The reduction uses the fact that $x^{163} \equiv x^7 + x^6 + x^3 + 1 \bmod F(x)$ to substitute terms of degree greater than 162. Using the recommended reduction polynomial $F(x) = x^{163} + x^7 + x^6 + x^3 + 1$, with its middle terms close, has the advantage that it can be effectively implemented using a table lookup for the 8 different bit locations in a byte with its reduction, which is a 2 byte word.

4.4.2 Point Arithmetic

We implement the point multiplication using the binary method (Section 2.6.1), Non-Adjacent Form(NAF) method (Section 2.6.2), and the Montgomery method (Section 2.6.4). The implementation is done over projective co-ordinates as inversion algorithms in $\mathbb{F}_{2^m}$ are very expensive due to the bit-level shifts. We use the standard projective co-ordinates because of the Montgomery point multiplication method used.

Table 4.1: $\mathbb{F}_{2^{163}}$ ECC software-only performance on an 8-bit AVR μC (@4 Mhz)

Operation	**Time** (clocks)	**Code-size** (bytes)	**Data RAM** (bytes)
Addition	151	180	42
Multiplication	15044	384	147
Squaring	441	46	63
Reduction	1093	196	63

The results of the software only implementation of the field arithmetic is given in Table 4.1. The point arithmetic performance is included in the Tables 4.2 and 4.3. The analysis of the software-only implementation shows that $\mathbb{F}_{2^m}$multiplication is the most costly operation with respect to execution time and memory requirement. Moreover, in the point multiplication algorithms, field multiplications are extremely frequent making it the bottleneck operation for ECC. A closer analysis of the multiplication block showed that the major part of the time was spent for load/store operations, because of the small number of registers available in the AVR processor which could not hold the large operands. Therefore, an hardware extension for this functional block would also have the potential to reduce the memory bottleneck and speed-up the ECC.

4.5 Proposed Instruction Set Extensions

The Instruction Set Architecture (ISA) of a microprocessor, is the unique set of instructions that can be executed on the processor. General purpose ISA are often insufficient to satisfy the special computational needs in cryptographic applications. A

more promising method is extending the ISA to build Application Specific Instruction-set Processors (ASIP).

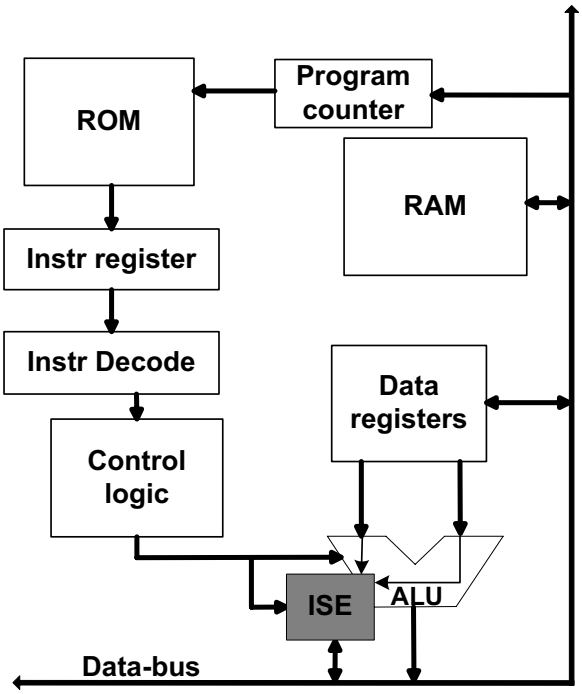


Figure 4.1: Processor Core with Extension

There are different ways of extending a processor. We consider an extension as shown in Fig. 4.1. Here, the additional hardware is closely coupled with the arithmetic logic unit (ALU) of the processor, reducing the interface circuitry. The control circuit of the processor is extended to support this extra hardware with new instructions. When the new instruction is fetched from the code-ROM during the execution of a program and decoded, the control unit can issue the required control signals to the ISE block. For multi-cycle instructions, the control logic has to take special care not to call the custom hardware until the multi-cycle operation is completed. The extension can also directly access the data-RAM, which is important for reducing delay if the computation is done over several data elements.

The popular approach to multimedia extensions has been to divide a large 32/64-bit data-bus into smaller 8-bit or 16-bit multimedia variables, and to run those in parallel as an SIMD instruction. However, for public-key cryptographic applications, the reverse is true: the operands are much larger than the data-path, requiring many bit

operations on long operands. For such applications, bit-parallel processing is required, where multiple data-words are operated upon simultaneously. One important issue here is the provision of the ISE with the operands. We approached this situation by implementing a complete $\mathbb{F}_{2^{163}}$ multiplier with minimum possible area.

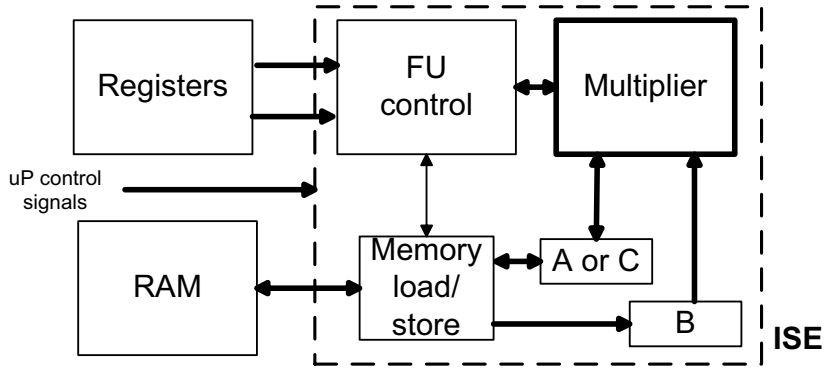


Figure 4.2: ISE Interface and Structure

Figure 4.2 shows the general layout of functional unit (FU) of the ISE, we simulate on the FPSLIC device. Four processor registers are initially loaded with the memory addresses of the two operands A and B. The ISE is then initiated by a control signal to the FU control (FUC) along with the first memory address byte. In our proof-of-concept implementation, this behavior is achieved by sending the byte over the data-lines from the processor to the FPGA, and confirming its reception through interrupt-lines from the FPGA to the processor. After the last memory address byte is received, the FUC initiates the memory load/store circuit within the ISE to load both the 21-byte operands directly from the SRAM in 21-cycles each. Then, the FUC runs the multiplier for 163-cycles to get the result C. During this period, the processor loads the memory address of C, sends it to the FPGA, and goes into polling state for the final interrupt from the FPGA. After the result C is obtained, the ISE stores it back directly in the memory in another 21-cycles and then sends the final interrupt, signalling the completion of the multiplication. This method of handshaking leads to extra control overheads, which can be reduced by having a more tightly coupled ISE to the processor without requiring confirmation interrupts. During the idle polling state, the processor could also be used in other computational work which is independent of the multiplication result. Memory access conflicts during such computation between the processor and the ISE is avoided by using a dual ported SRAM.

We implement different levels of ISE on our implementation platform as a proof-of-concept and to get a fairly accurate idea of the speed-up it can produce.

4.5.1 8-by-8 Bit-Parallel Multiplier

Most 8-bit micro-controllers, including AVR, have an 8-by-8 bit integrated integer multiplication instruction. We implement here an analogous 8-by-8 bit polynomial multiplier which is much more simpler and smaller in size, as it does not involve carries as in the integer multiplier. We implement a parallel multiplier which executes in 1 clock cycle. It can be interfaced directly to the registers to obtain a 1 cycle multiplier instruction. It has an area requirement of 64 AND and 49 XOR gates. On our test platform, the multiplier requires a total of 4 cycles due to control overheads. Using this multiplier, speed-up of the point multiplication is almost doubled as shown in Table 4.3.

4.5.2 163-by-163 Bit-Serial Multiplier

Although, 8-by-8 bit polynomial multiplier gives considerable speed-up, the bottleneck is the frequent memory access to the same set of bytes. This is caused due to the fact that the 31 registers available in the AVR processor are not enough to store the two long 21 byte operands. A more efficient method and a logical next step was to perform a 163-by-163 multiplication on the extension.

A bit-serial $\mathbb{F}_{2^m}$ hardware multiplier is the most simple solution which requires the least area. The core of the multiplier is as shown in Fig. 4.3, where reduction circuit is hardwired. A modification for implementing a more general reduction polynomial or variable size multiplication is discussed in Section 4.5.4. We implement a Least Significant Bit (LSB) first multiplier as it has a smaller critical path delay. Our bit-serial multiplier requires 163 ANDs, 167 XORs and 489 FFs (Flip-Flops). A 163-by-163 multiplication is computed in 163 clocks, excluding data input and output. In our implementation, control and memory access overheads lead to a total execution time of 313 clocks, beginning from the processor sending the memory addresses of A, B and C to the final result stored in C. Since the multiplier is much faster than the squaring in software, we use the multiplier also for squaring by loading $A = B$. The results (in Tables 4.2 and 4.3) show a drastic speed-up using this multiplier. It should be noted that the control overhead can be considerably reduced when the hardware is more tightly coupled within the processor, e.g., in an ASIC implementation of the

processor with the proposed extension.

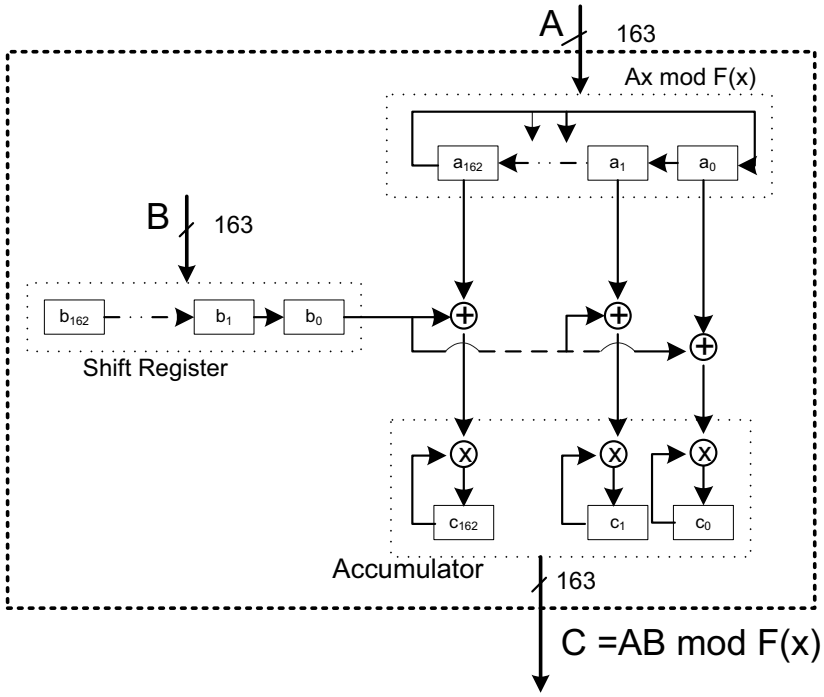


Figure 4.3: Bit-Serial LSB Multiplier

4.5.3 163-by-163 Digit Serial Multiplier

Further trade-off between area and speed is possible using a digit-serial multiplier. As previously described in Section 2.4.1, compared to the bit-serial multiplier where only one bit of operand B is used in each iteration, in a digit-serial multiplier, multiple bits (equal to the digit-size) of B are multiplied to the operand A in each iteration (Fig. 4.4). We use a digit size of 4 as it gives a good speed-up without drastically increasing the area requirements. The total area for multiplier is 652 ANDs, 684 XORs and 492 FFs. A 163-by-163 multiplication with reduction requires 42 clocks. In our implementation, the control overheads leads to a total of 193 clocks.

Our implementation gives estimates of the speed-up possible when building ISEs in small embedded 8-bit processors. In a real ISE, where the hardware is more tightly coupled, the control signal overhead can be considerably reduced and a better efficiency is possible.

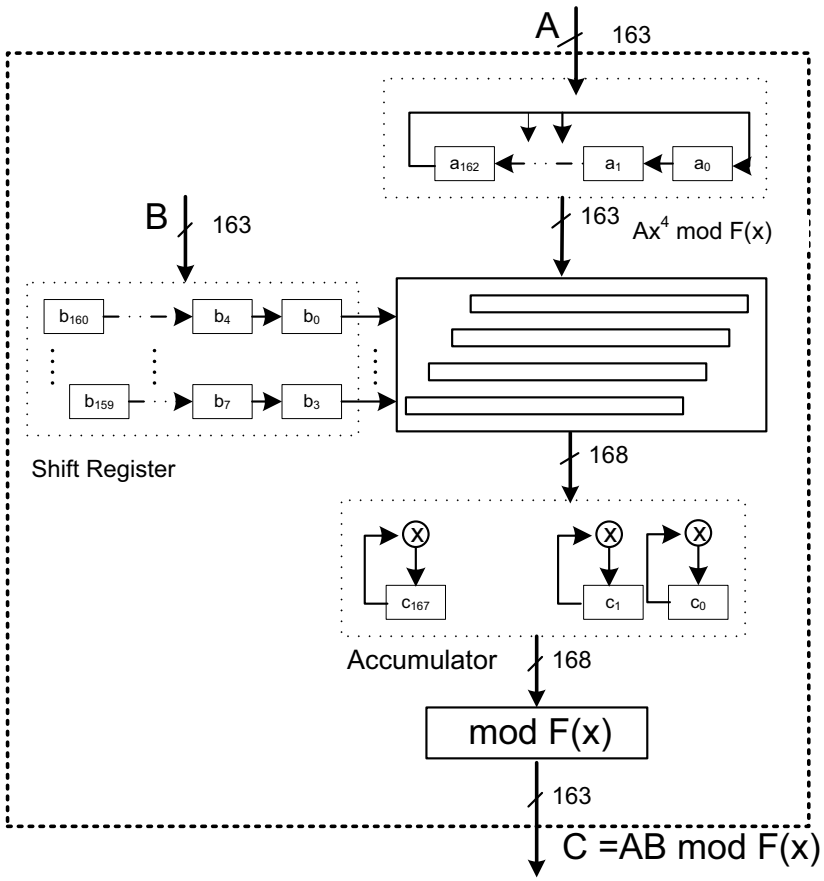


Figure 4.4: Digit-4 Serial LSD Multiplier

4.5.4 A Flexible Multiplier

Flexibility of crypto algorithm parameters, especially operand lengths, can be very attractive because of the need to alter them when deemed insecure in the future, or for providing compatibility in different applications. Considering the high-volume of pervasive devices, replacing each hardware component seems improbable. We discuss here how the multiplier can be made more flexible to satisfy these needs.

Support of a generic reduction polynomial with a maximum degree of m of the form $F(x) = x^m + G(x) = x^m + \sum_{i=0}^{m-1} g_i x^i$, requires storage of the reduction coefficients and additional circuitry as shown in Fig. 4.5 (a similar implementation for a digit-serial multiplier is straightforward). The reduction polynomial needs to be initialized only once at the beginning of the point multiplication. Thus the total number of clocks required for multiplication remains almost the same.

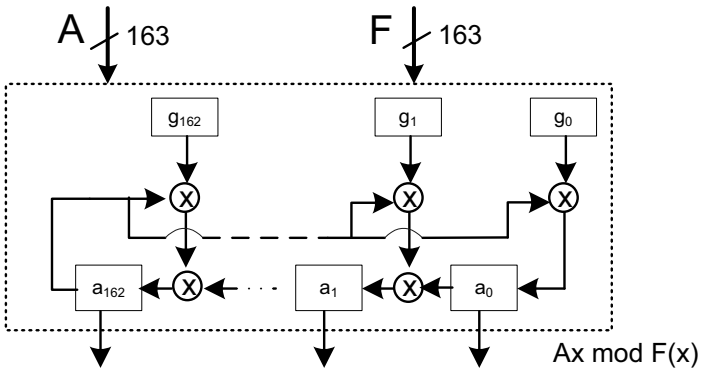


Figure 4.5: Bit-serial reduction circuit

Different bit-length multipliers, for different key-length ECC, can also be supported using this structure. We show as an example, how the 163-bit multiplier could be also used to multiply two 113-bit operands A' and B', with 113-bit reduction polynomial G'.

The operands A, B and the reduction polynomial are initially loaded as

$$\begin{aligned} A &= (a'_{112}...a'_1a'_00...0) &= A'x^{50} \\ B &= (0...0b'_{112}...b'_1b'_0) &= B' \\ G &= (g'_{112}...g'_1g'_00...0) &= G'x^{50} \end{aligned}$$

If $C' = A' \cdot B' \bmod F'(x)$ then

$$A \cdot B \bmod F(x) \;=\; A'x^{50} \cdot B' \bmod (F'(x)x^{50}) \;=\; C'x^{50}$$

Thus the result is stored in the most-significant bits of operand C after 113 clock cycles. The memory load/store circuit and the FU control unit takes care to load the operands appropriately, and to fetch the result after the required number of clocks from the multiplier to store it back appropriately in memory.

4.6 Summary

In this work, we showed that huge performance gains are possible in small 8-bit processors by introducing small amounts of extra hardware. The results show 1–2 orders of magnitude increase in speed-up for the ECC implementation. The hardware costs are in the range of 250–500 extra CLBs. Also the code size and data RAM usage is

decreased. The performance gain due to the ISE can be used to reduce the total power consumption of the devices by running the whole device at a lower frequency, which can be a major benefit in wireless pervasive applications. The proof-of-concept implementation can also be used directly as a reconfigurable ISE. The fact that the public-key exchange is done only once in the initial phase of the communication, can be used to reconfigure the FPGA at run-time for an ISE suitable for a different application (like signal processing) running later on the device. Thus two different sets of ISEs can be run on the same constrained device, accelerating both applications without increasing the total hardware cost.

Table 4.2: Software Only and Hardware Assisted Point Operation Performance (@4 Mhz)

Multiplier Type	**Point Operation**	**Time** (cycles)	**Code-size** (bytes)	**Data RAM** (bytes)	**Precomputation** (bytes)
Comb	Double	17.113	3136	274	544
	Add	39.879	6948	358	
8-by-8 multiplier	Double	10.056	2790	254	544
	Add	22.237	5146	338	
163-by-163 multiplier	Double	0.855	956	189	0
	Add	1.526	1940	273	
163-by-163 digit-4	Double	0.585	956	189	0
	Add	1.076	1940	273	

Table 4.3: ECC Scalar Point Multiplication Performance (@4 Mhz)

Field Multiplier	**CLBs**	**Point Multiplier**	**Time** (sec)	**Code-size** (bytes)	**Data RAM** (bytes)	**Precomputation** (bytes)
comb multiplier		Binary	6.039	10170	379	544
		NAF	5.031	10654	379	
		Montgomery	4.140	8208	358	
8-by-8 multiplier		Binary	3.451	7966	359	544
		NAF	2.862	8316	359	
		Montgomery	2.357	6672	338	
163-by-163 multiplier	245	Binary	0.264	2936	294	0
		NAF	0.224	3014	294	
		Montgomery	0.169	2048	273	
163-by-163 digit-4	498	Binary	0.183	2936	294	0
		NAF	0.115	3014	294	
		Montgomery	0.113	2048	273	

Chapter 5

Hardware/Software Co-design: Extensions for a 32-bit Processor

We present here results of the collaborative work with Johann Großschädl, IAIK, Graz University of Technology, Austria, part of which was published in [25].

5.1 Motivation and Outline

Public-key cryptosystems are becoming an increasingly important workload for embedded processors, driven by the need for security and privacy of communication. In the past, embedded systems with poor processing capabilities (e.g., smart cards) used dedicated hardware (co-processors) to offload the heavy computational demands of cryptographic algorithms from the host processor. However, systems which use fixed-function hardware for cryptography have significant drawbacks: they are not able to respond to advances in cryptanalysis, or to changes in emerging standards.

In this chapter, we investigate the potential of architectural enhancements and instruction set extensions for fast yet flexible implementations of arithmetic for ECC on a 32-bit embedded processor. Extending a general-purpose architecture with special instructions for performance-critical arithmetic operations, allows us to combine full software flexibility with the efficiency of a hardware solution. We opted for using OEFs in our work, since they have some specific advantages over other types of finite fields.

This chapter is organized as follows: First, some related work on the use of extensions for public-key cryptography is mentioned in Section 5.2. We use the MIPS32 (described in Section 5.3) as the base architecture and analyze in Section 5.4, how the arithmetic algorithms can be efficiently implemented on MIPS32 processors, and which functionality is required to achieve peak performance. Moreover, we also identify

disadvantageous properties of the MIPS32 architecture in this context. Different architectural extensions to support OEF arithmetic in an efficient manner are discussed in Section 5.5. The main goal was to design instruction set extensions (ISEs) that can be easily integrated into MIPS32, and entail only minor modifications to the processor core.

5.2 Related work

Most previous work is concerned with architectural enhancements and ISE for multiple-precision modular multiplication, that are needed for the "traditional" cryptosystems like RSA [65, 68, 24]. A number of microprocessor vendors have extended their architectures with special instructions targeting cryptographic applications. For instance, the instruction set of the IA-64 architecture, jointly developed by Intel and Hewlett-Packard, has been optimized to address the requirements of long integer arithmetic [37]. The IA-64 provides an integer multiply-and-add instruction (`XMA`), which takes three 64-bit operands (a, b, c) and produces the result $a \times b + c$. Either the lower or the upper 64 bits of the result are written to a destination register, depending on whether `XMA.LU` or `XMA.HU` is executed. Another example for a cryptography-oriented ISA enhancement is the `UMAAL` instruction, which has been added to version 6 of the ARM architecture (ARMv6) [3]. The `UMAAL` instruction executes a special multiply-accumulate operation of the form $a \times b + c + d$, interpreting the operands as unsigned 32-bit integers, and stores the 64-bit result in two general-purpose registers. This operation is carried out in the inner loop of many algorithms for multiple-precision modular arithmetic, e.g., Montgomery multiplication [45].

Previous work on architectural enhancements dealing with specific instructions for use in ECC have only been considered on prime fields $\mathbb{F}_p$ and binary extension fields $\mathbb{F}_{2^m}$. The work in [24], demonstrates the benefits of a combined hardware/software approach to implement arithmetic in binary fields $\mathbb{F}_{2^m}$. Efficient algorithms for multiple-precision multiplication, squaring, and reduction of binary polynomials are presented, assuming the processor's instruction set includes the `MULGF2` instruction (which performs a word-level multiplication of polynomials over $\mathbb{F}_2$). In [20], a datapath-scalable minimalist cryptographic processor architecture for mobile devices, *PAX* is introduced. PAX consists of a simple RISC-like base instruction set, augmented by a few low-cost instructions for cryptographic processing. These special instructions assist a wide range of both secret-key and public-key cryptosystems, including systems that use

binary extension fields $\mathbb{F}_{2^m}$ as underlying algebraic structure.

The *Domain-Specific Reconfigurable Cryptographic Processor (DSRCP)* [22] is loosely related to our work. Optimized for energy efficiency, the DSRCP provides an instruction set for a domain of arithmetic functions over prime fields $\mathbb{F}_p$, binary extension fields $\mathbb{F}_{2^m}$, and elliptic curves built upon the latter. However, from the perspective of design methodology, the DSRCP represents a "classical" application-specific instruction set processor (ASIP) developed from scratch, i.e., it is not an extension of an existing architecture.

In the present work, we introduce instruction set extensions to support arithmetic in OEFs.

5.3 The MIPS32 architecture

The MIPS32 architecture is a superset of the older MIPS I and MIPS II instruction set architectures and incorporates new instructions for standardized DSP operations like "multiply-and-add" (`MADD`) [57]. MIPS32 uses a load/store data model with 32 general-purpose registers of 32 bits each. The fixed-length, regularly encoded instruction set includes the usual arithmetic/logical instructions, load and store instructions, jump and branch instructions, as well as co-processor instructions. All branches in MIPS32 have an architectural delay of one instruction. The instruction immediately following a branch (i.e., the instruction in the so-called *branch delay slot*) is always executed, regardless of whether the branch is taken or not.

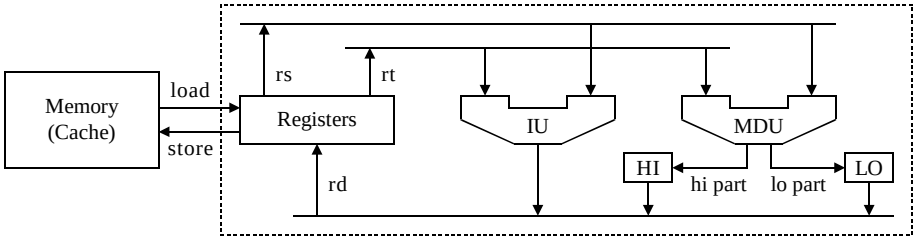


Figure 5.1: 4Km datapath with integer unit (IU) and multiply/divide unit (MDU)

The MIPS32 architecture defines that the result of a multiply (`MULT`) or multiply-and-add (`MADD`) operation is to be placed in two special result/accumulation registers, referenced by the names `HI` and `LO` (see Figure 5.1). Using the "move-from-HI" (`MFHI`) and "move-from-LO" (`MFLO`) instructions, these values can be transferred to the

general-purpose register file. The `MADD` instruction multiplies two 32-bit operands and adds the product to the 64-bit concatenated values in the `HI/LO` register pair. Then, the resulting value is written back to the `HI` and `LO` registers. MIPS32 also provides a `MADDU` ("multiply-and-add unsigned") instruction, which performs essentially the same operation as `MADD`, but interprets the operands as unsigned integers.

The *4Km processor core* is a high-performance implementation of the MIPS32 instruction set [56]. Key features of the 4Km are a five-stage, single-issue pipeline with branch control, and a fast multiply/divide unit (MDU) with a (32×16)-bit multiplier. Most instructions occupy the execute stage of the pipeline only for a single cycle. However, load operations require an extra cycle to complete before they exit the pipeline. The 4Km interlocks the pipeline when the instruction immediately following the load instruction uses the contents of the loaded register. Optimized MIPS32 compilers try to fill load delay slots with useful instructions.

The MDU works autonomously, which means that the 4Km has a separate pipeline for multiply, multiply-and-add, and divide operations (see Figure 5.1). This pipeline operates in parallel with the integer unit (IU) pipeline and does not necessarily stall when the IU pipeline stalls. Note that a (32×32)-bit multiply operation passes twice through the multiplier, i.e., it has a latency of two cycles. However, the 4Km allows to issue an IU instruction during the latency period of the multiply operation, provided that the IU instruction does not depend on the result of the multiply. This "parallelism" is possible since the `MULT` instruction does not occupy the ports of the register file in the second cycle of its execution. Therefore, long-running (multi-cycle) MDU operations, such as a (32×32)-bit multiply or a divide, can be partially masked by other IU instructions.

5.4 Implementation aspects on the MIPS32 processor

In the following section, we discuss the choice of parameters for OEF and the limitations of the MIPS32 architecture with respect to the efficient implementation of the OEF arithmetic. Since the word-size of the processor is 32, it is obvious to use an OEF defined by a 32-bit pseudo-Mersenne prime. The extension degree is chosen such that the reduction polynomial $P(t)$ exists, and an appropriate security level can be achieved. A typical example is the OEF with $p = 2^{32} - 5$ and $m = 5$, which has an order of 160 bits.

Multiplication is by far the most important field operation and has a significant

impact on the overall performance of elliptic curve cryptosystems defined over OEFs. This is the case for both the affine and projective coordinates, since the efficiency of the Itoh-Tsujii inversion depends heavily on fast extension field multiplication (see Section 2.5.1). The multiplication of two polynomials $A, B \in \mathbb{F}_{p^m}$ (as previously stated in Section 2.5.1) is performed by m^2 multiplications of the corresponding 32-bit coefficients $a_i, b_j \in \mathbb{F}_{p^m}$. Though the `MADDU` instruction facilitates the "multiply-and-accumulate" strategy for polynomial multiplication, the 64-bit precision of the result/accumulation registers `HI` and `LO` is a substantial drawback in this context. The coefficient-products $a_i \cdot b_j$ can be up to 64 bits long, which means that a cumulative sum of several 64-bit products exceeds the 64-bit precision of the `HI/LO` register pair. Therefore, the `MADDU` instruction cannot be used to implement the "multiply-and-accumulate" strategy for polynomial multiplication, simply because the addition of 64-bit coefficient-products to a running sum stored in the `HI/LO` registers would cause an overflow and loss of precision.

A straightforward way to overcome this problem is to use a smaller pseudo-Mersenne prime p, e.g., a 28-bit prime instead of a 32-bit one, so that the accumulation of m coefficient-products will not overflow the 64-bit `HI/LO` register pair. However, when the bit-length of p is less than 32 bits, we need a larger extension degree m in order to obtain an OEF of sufficient size, e.g., $m = 6$ instead of 5. The value of m determines the number of coefficient-products that have to be computed when multiplying two elements in $\mathbb{F}_{p^m}$. For instance, an extension degree of $m = 6$ requires to carry out $m^2 = 36$ coefficient multiplications (excluding the extension field reduction), which represents an increase of 44% over the 25 coefficient multiplications needed when $m = 5$.

Choosing a smaller pseudo-Mersenne prime, $p = 2^n - c$ with $n < 32$, entails a second disadvantage. The reduction of a sum of coefficient-products, according to Algorithm 2.9, requires the sum z to be of the form $z = z_H \cdot 2^n + z_L$, whereby z_L represents the n least significant bits of z and z_H includes all the remaining (i.e., higher) bits of z. Extracting the integers z_H and z_L from z is trivial when n is the same as the word size of the processor, i.e., $n = 32$, since in this case no bit-manipulations have to be performed. However, when $n < 32$, we are forced to carry out shifts of bits within words in order to obtain the higher part z_H. In addition to these bit manipulations, a number of data transfers between general-purpose registers and the accumulation registers `HI`, `LO` are required before we can do the actual reduction by computation of $z_H \cdot c + z_L$ (see line 5 of Algorithm 2.9).

The number of loop iterations in Algorithm 2.9, depends on the magnitude of z. It can be shown that the loop iterates at most twice when z is a $2n$-bit integer,

with $z < p^2$ (see [34]). Larger values of z may necessitate additional iterations. In general, any iteration of the loop decreases the length of z by $n - \lceil \log_2(c) \rceil$ bits. The "multiply-and-accumulate" strategy for polynomial multiplication requires reduction of a cumulative sum of up to m coefficient-products (see Figure 2.2), which means that the bit-length of the quantity to reduce is $2n + \lceil \log_2(m) \rceil$, provided that all coefficients a_i, b_j are at most n bits long. As a consequence, Algorithm 2.9 may need to perform more than two iterations. In our case, assuming that p is a 32-bit PM prime, i.e., $p = 2^{32} - c$, and c is no longer than 16 bits due to Definition 2.1. If we use the extension degree of $m = 5$, the cumulative sum of m coefficient-products is up to 67 bits long. We write this sum as $z = z_H \cdot 2^{32} + z_L$, whereby z_H represents the 35 most significant bits and z_L the 32 least significant bits of z, respectively. The first iteration of the while-loop reduces the length of z from 67 bits to 51 bits or even less. After the third iteration, the number z is either fully reduced or at most 33 bits long, so that a final subtraction of p is sufficient to guarantee $z < p$.

It can be formally proven that for $n = 32$, $\log_2(c) \leq 16$, and reasonable extension degrees m, at most three iterations of the while-loop (i.e., three multiplications by c) and at most one subtraction of p are necessary to bring the result within the desired range of $[0, p-1]$. We refer the interested reader to [34, 80] for a more detailed treatment.

5.5 Proposed extensions to MIPS32

As mentioned before, the `MADDU` instruction can be used to implement polynomial multiplication according to the "multiply-and-accumulate" strategy, provided that the bit-length of the coefficient is less than 32. However, this constraint competes with the optimal use of the `MADDU` instruction, and the attempt to exploit the full precision of the processor's registers and data-path, respectively. The performance of the multiplication in $\mathbb{F}_{p^m}$, $p = 2^n - c$, is mainly determined by the processor's ability to calculate a sum of up to m coefficient-products, and the ability to perform the reduction of this sum modulo p in an efficient manner. Some properties of the MIPS32 architecture — such as the 64-bit precision of the concatenated result/accumulation registers `HI` and `LO` — are clearly disadvantageous for the implementation of OEF arithmetic.

Table 5.1: Format and description of useful instructions for OEF arithmetic

Format	Description	Operation
`MADDU rs, rt`	Multiply and ADD Unsigned	$(HI/LO) \leftarrow (HI/LO) + rs \times rt$
`MADDH rs`	Multiply and ADD HI register	$(HI/LO) \leftarrow HI \times rs + LO$
`SUBC rs`	Subtract Conditionally from HI/LO	$if\ (HI \neq 0)\ then$ $(HI/LO) \leftarrow (HI/LO) - rs$

5.5.1 Multiply/accumulate unit with a 72-bit accumulator

Efficient OEF arithmetic requires exploiting the full 32-bit precision of the registers, and hence the prime p should also have a length of 32 bits. An implementation of polynomial multiplication, with 32-bit coefficients, would greatly profit from a multiply/accumulate (MAC) unit with a "wide" accumulator, so that a certain number of 64-bit coefficient-products can be summed up without overflow and loss of precision. For instance, extending the accumulator by eight guard bits, allows accumulation of up to 256 coefficient-products, which is sufficient for OEFs with an extension degree of $m \leq 256$. However, when we have a 72-bit accumulator, we also need to extend the precision of the `HI` register from 32 to 40 bits, so that the `HI/LO` register pair is able to accommodate 72 bits altogether. The extra hardware cost is negligible, and a slightly longer critical path in the MAC unit's final adder is no significant problem for most applications.

Multiplying two polynomials $A, B \in \mathbb{F}_{p^m}$, according to the product scanning technique, comprises m^2 multiplications of 32-bit coefficients and the reduction of $2m - 1$ column sums modulo p (without considering the extension field reduction). The calculation of the column sums depicted in Figure 2.2 can be conveniently performed with the `MADDU` instruction, since the wide accumulator and the 40-bit `HI` register prevent overflows. After the summation of all coefficient-products $a_i \cdot b_j$ of a column, the 32 least significant bits of the column sum are located in the `LO` register, and the (up to 40) higher bits reside in register `HI`. Therefore, the content of register `HI` and `LO` correspond to the quantities z_H and z_L of Algorithm 2.9, since p is a 32-bit prime, i.e., $n = 32$.

5.5.2 Custom instructions

Besides coefficient multiplications, also the subfield reductions can contribute significantly to the overall execution time of OEF arithmetic operations. This motivated us to design two custom instructions for efficient reduction modulo a PM prime, similar to Algorithm 2.9. Our first custom instruction is named `MADDH`, and multiplies the content of register `HI` by the content of a source register `rs`, adds the value of register `LO` to the product, and stores the result in the `HI/LO` register pair (see Table 5.1). This is exactly the operation carried out at line 5 of Algorithm 2.9. The `MADDH` instruction interprets all operands as unsigned integers and shows therefore some similarities with the `MADDU` instruction. However, it must be considered that the extended precision of the `HI` register requires a larger multiplier, e.g., a (40×16)-bit multiplier instead of the conventional (32×16)-bit variant. The design and implementation of a (40×16)-bit multiplier able to execute the `MADDH` instruction and all native MIPS32 multiply and multiply-and-add instructions is straightforward.

Our second custom instruction, `SUBC`, performs a conditional subtraction, whereby the minuend is formed by the concatenated value of the `HI/LO` register pair, and the subtrahend is given by the value of a source register `rs` (see Table 5.1). The subtraction is only carried out when the `HI` register holds a non-zero value, otherwise no operation is performed. `SUBC` writes its result back to the `HI/LO` registers. We can use this instruction to realize an operation similar to the one specified at line 7 of Algorithm 2.9. However, the `SUBC` instruction uses the content of the `HI` register to decide whether or not to carry out the subtraction, i.e., it makes a comparison to 2^n instead of $p = 2^n - c$. This comparison is easier to implement, but may entail a not fully reduced result even though it will always fit into a single register. In general, when performing calculations modulo p, it is not necessary that the result of a reduction operation is always the least non-negative residue modulo p, which means that we can continue the calculations with an incompletely reduced result.

5.5.3 Implementation details and performance evaluation

In the following, we demonstrate how OEF arithmetic can be implemented on an extended MIPS32 processor, assuming that the two custom instructions `MADDH` and `SUBC` are available. We developed a functional, cycle-accurate SystemC model of a MIPS32 4Km core in order to verify the correctness of the arithmetic algorithms and to estimate their execution times. Our model is based on a simple, single-issue pipeline and implements a subset of the MIPS32 ISA, along with the two custom instructions

```
label: LW    $t0, 0($t1)      # load A[i] into $t0
       LW    $t2, 0($t3)      # load B[j] into $t2
       ADDIU $t1, $t1, 4      # increment address in $t1 by 4
       MADDU $t0, $t2         # (HI|LO)=(HI|LO)+($t0*$t2)
       BNE   $t3, $t4, label  # branch if $t3!=$t4
       ADDIU $t3, $t3, -4     # decrement address in $t3 by 4
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       MADDH $t5              # (HI|LO)=(HI*$t5)+LO
       SUBC  $t6              # if (HI!=0) then (HI|LO)=(HI|LO)-$t6
```

Figure 5.2: Calculation of a column sum and subfield reduction

`MADDH` and `SUBC`. While load and branch delays are considered in our model, we did not simulate the impact of cache misses, i.e., we assumed a perfect cache system.

The code snippet, depicted in Figure 5.2, calculates a column sum of coefficient-products $a_i \cdot b_j$ and performs a subfield reduction, i.e., the column sum is reduced modulo a PM prime p (see Section 5.4). For e.g., the instruction sequence can be used to calculate the coefficient c_3, as illustrated in Figure 2.2, and formally specified by Equation (2.16). The first six instructions implement a loop that multiplies two coefficients a_i, b_j and adds the coefficient-product to a running sum in the `HI/LO` register pair. After termination of the loop, the column sum is reduced modulo p with the help of the last four instructions. The polynomials $A(t), B(t)$ are stored in arrays of unsigned 32-bit integers, which are denoted as `A` and `B` in Figure 5.2. Before entering the loop, register `$t1` and `$t3` are initialized with the address of a_0 and b_3, respectively. Two `ADDIU` instructions, which perform simple pointer arithmetic, are used to fill a load delay slot and the branch delay slot. Register `$t3` holds the current address of b_j and is decremented by 4 each time the loop repeats, whereas the pointer to the coefficient a_i (stored in register `$t1`) is incremented by 4. The loop terminates when the pointer to b_j reaches the address of b_0, which is stored in `$t4`.

Once the column sum has been formed, it must be reduced modulo p in order to obtain the coefficient c_3 as final result. The last four instructions of the code snip-

pet implement the reduction modulo a 32-bit PM prime $p = 2^n - c$ similar to Algorithm 2.9. As explained in Section 5.4, at most three multiplications by c and at most one subtraction of p are necessary to guarantee that the result is either fully reduced or at most 32 bits long. The `MADDH` instructions implement exactly the operation at line 5 of Algorithm 2.9, provided that register `$t5` holds the offset c. At last, the `SUBC` instruction performs the final subtraction when p is stored in `$t6`.

The execution time of the instruction sequence depicted in Figure 5.2, depends on the implementation of the multiplier. An extended MIPS32 processor, with a (40×16)-bit multiplier and a 72-bit accumulator, executes an iteration of the loop in six clock cycles, provided that no cache misses occur. The `MADDU` instruction writes its result to the `HI/LO` register pair (see Figure 5.1), and does not occupy the register file's read ports and write port during the second clock cycle. Therefore, other arithmetic/logical instructions, such as the `BNE` instruction in Figure 5.2, can be executed during the latency period of the `MADDU` operation. On the other hand, the `MADDH` instructions requires only a single clock cycle to produce its result on a (40×16)-bit multiplier, provided that the multiplier implements an "early termination" mechanism. According to Definition 2.1, the offset c is at most 16 bits long when p is a 32-bit PM prime, which means that a multiplication by c requires only one pass through the multiplier. The operation performed by the `SUBC` instruction is very simple, and thus it can be easily executed in one clock cycle. In summary, the four instructions for a subfield reduction require only four clock cycles altogether.

Experimental results

We implemented the arithmetic operations for a 160-bit OEF defined by the following parameters: $p = 2^{32} - 5$, $m = 5$, and $x(t) = t^5 - 2$. Our simulations show that a full OEF multiplication (including extension field reduction) executes in 406 clock cycles, which is almost twice as fast as a "conventional" software implementation that uses only native MIPS32 instructions. The OEF squaring executes in 345 cycles on our extended MIPS32 model. These timings were achieved without loop unrolling and without special optimizations like Karatsuba's algorithm. An elliptic curve scalar multiplication $k \cdot P$ can be performed in 940k clock cycles when projective coordinates are used in combination with the binary NAF method (see [34] for details). On the other hand, we were not able to implement the scalar multiplication in less than 1.75M cycles on a standard MIPS32 processor. In summary, the proposed architectural enhancements achieve a speed-up factor of 1.8.

Reduction modulo PM primes of less than 32 bits

In order to ensure compatibility with other systems, it may be necessary to handle PM primes with a bit-length of less than 32. The proposed extensions are also useful for shorter primes, e.g., the 31-bit prime $p = 2^{31} - 1$. This is possible by performing all subfield reduction operations modulo a 32-bit *near-prime* $q = d \cdot p$ instead of the original prime p. A near-prime is a small multiple of a prime, e.g., $q = 2 \cdot (2^{31} - 1) = 2^{32} - 2$. All residues obtained through reduction by q are congruent to the residues obtained through reduction by the "original" prime p. Therefore, we can carry out a full elliptic curve scalar multiplication with a 32-bit near-prime q instead of p, using the same software routines. However, at the very end of the calculation, an extra reduction of the coefficients modulo p is necessary.

5.6 Summary

In this work, we proposed simple extensions for efficient OEF arithmetic on MIPS32 processors. A wide accumulator allows a convenient calculation of column sums with the help of the `MADDU` instruction, whereas a (40×16)-bit multiplier along with the two custom instructions `MADDH` and `SUBC` makes it possible to perform a reduction modulo a PM prime in only four clock cycles. Our simulations show that an extended MIPS32 processor is able to execute a multiplication, in a 160-bit OEF, in only 406 clock cycles, which is almost twice as fast as a conventional software implementation with native MIPS32 instructions. A full elliptic curve scalar multiplication over a 160-bit OEF requires approximately 940k clock cycles. The proposed extensions are simple to integrate into a MIPS32 core since the required modifications/adaptions are restricted to the instruction decoder and the multiply/divide unit (MDU). A fully parallel (i.e., single-cycle) multiplier is not necessary to reach peak performance. The extra hardware cost for a (40×16)-bit multiplier is marginal when we assume that the "original" processor is equipped with a (32×16)-bit multiplier.

Chapter 6

Hardware Design: Optimal Digit Multipliers for $\mathbb{F}_{2^m}$

Parts of this work have been published in [49] and to be published in [50]

6.1 Motivation and Outline

Hardware based implementations of curve based cryptography, especially *Elliptic Curve Cryptography* (ECC), are becoming increasingly popular. A comprehensive overview on hardware implementations of RSA and ECC can be found in [10]. Characteristic two fields $\mathbb{F}_{2^m}$ are often chosen for hardware realizations as they are well suited for hardware implementation due to their "carry-free" arithmetic. The over-all time and area complexity of ECC and HECC implementations heavily depends on the $\mathbb{F}_{2^m}$ multiplier architecture used. Most commonly cited implementations of ECC over characteristic two fields in literature [30, 63] use digit multipliers with digit sizes of power of 2. Using a digit multiplier, allows implementations to do a tradeoff between speed and area, enabling fast implementations tuned to the available resources of the hardware.

In this chapter we present different architectures like Single Accumulator Multiplier (SAM), Double Accumulator Multiplier (DAM) and N-Accumulator Multiplier (NAM) for the implementation of the digit multiplier. The naming is based on the number of internal accumulators used to store the intermediate result. We use these extra accumulators to reduce the critical path delay of the multipliers and hence increase the maximum operating frequency. We also give the necessary conditions that need to be satisfied by the irreducible polynomial for such implementations. We find that all the standardized NIST polynomials satisfy the required conditions for implementing these techniques. Evaluating the multiplication speed and the area-time product for

the different architectures leads to the optimum digit sizes for an implementation. These results show that the cryptosystems can be implemented more efficiently than had been done in the past. E.g., for NIST B-163 [60], the most optimum architectures are SAM with digit-size=3 and DAM with digit-size=6, giving the developer a good choice between area and time.

The remaining of the chapter is organized as follows. Section 6.2 gives an overview of the conditions for choosing efficient reduction polynomials. Section 6.3 introduces our different architectures for the Digit-Serial multiplier. Section 6.4 summarizes and evaluates optimum digit sizes for the different architectures.

6.2 Background on Digit-Serial Multipliers.

Finite field multiplication in $\mathbb{F}_{2^m}$ of two elements A and B to obtain a result $C = AB \bmod p(\alpha)$ (where $p(\alpha)$ is the irreducible polynomial), can be done in various ways based on the available resources. As described in Section 2.4.1, bit-serial multipliers processes the coefficients of multiplicand A in parallel, while the coefficients of the multiplier B are processed in a serial manner. Hence, these multipliers are area-efficient and suitable for low-speed applications.

Reduction $\bmod \mathbf{p}(\alpha)$ for bit-serial multipliers

In the Least Significant Bit (LSB) multiplier (similar to shift-and-add MSB Algorithm 2.6), a quantity of the form $W\alpha$, where $W(\alpha) = \sum_{i=0}^{m-1} w_i\alpha^i \in \mathbb{F}_{2^m}$, has to be reduced mod$p(\alpha)$. Multiplying W by α, we obtain

$$W\alpha = \sum_{i=0}^{m-1} w_i\alpha^{i+1} = w_{m-1}\alpha^m + \sum_{i=0}^{m-2} w_i\alpha^{i+1} \tag{6.1}$$

Using the property of the reduction polynomial $p(\alpha) = 0 \bmod p(\alpha)$, we obtain:

$$\alpha^m = \sum_{i=0}^{m-1} p_i\alpha^i \bmod p(\alpha) \tag{6.2}$$

Substituting for α^m and re-writing the index of the second summation in Equation (6.1), $W\alpha \bmod p(\alpha)$ can then be calculated as follows:

$$W\alpha \bmod p(\alpha) = \sum_{i=0}^{m-1}(p_i w_{m-1})\alpha^i + \sum_{i=1}^{m-1} w_i\alpha^i = (p_0 w_{m-1}) + \sum_{i=1}^{m-1}(w_{i-1} + p_i w_{m-1})\alpha^i \tag{6.3}$$

where all coefficient arithmetic is done modulo 2.

Reduction $\mathrm{mod}\ \mathbf{p}(\alpha)$ for Digit Multipliers

Digit Multipliers are a trade-off between area and speed. In a digit-serial multiplier (see Section 2.4.1) multiple bits (equal to the digit-size) of B are multiplied to the operand A in each iteration In the Least Significant Digit (LSD) multiplier, products of the form $W\alpha^D$ occurs (as seen in Step 4 of Algorithm 2.8) which have to be reduced mod $p(\alpha)$. As in the LSB multiplier case, one can derive equations for the modular reduction for *general* irreducible polynomials $p(\alpha)$. However, it is more interesting to search for polynomials that minimize the complexity of the reduction operation. For determining optimum irreducible polynomials we use two theorems from [76].

Theorem 6.1. *Assume that the irreducible polynomial is of the form* $p(\alpha) = \alpha^m + p_k\alpha^k + \sum_{j=0}^{k-1} p_j\alpha^j$, *with* $k < m$. *For* $t \leq m-1-k$, α^{m+t} *can be reduced to a degree less than* m *in one step with the following equation:*

$$\alpha^{m+t} \bmod p(\alpha) = p_k\alpha^{k+t} + (\sum_{j=0}^{k-1} p_j\alpha^{j+t}) \tag{6.4}$$

Proof. *The result follows from Equation (2.7), the assumed form of* $p(\alpha)$, *and the fact that for* α^{k+t} *where* $k+t \leq m-1$, *no modular reduction is necessary.*

Theorem 6.2. *For digit multipliers with digit-element size* D, *when* $D \leq m-k$, *the intermediate results in Algorithm 2.8 (Step 4 and Step 6) can be reduced to degree less than* m *in one step.*

Proof. *Refer [76].*

Theorems 6.1 and 6.2 implicitly say that for a given irreducible polynomial $p(\alpha) = \alpha^m + p_k\alpha^k + \sum_{j=0}^{k-1} p_j\alpha^j$, the digit-element size D has to be chosen based on the value of k, the degree of the second highest coefficient in the irreducible polynomial.

6.3 Architecture Options for LSD

In this section, we provide different architectural possibilities for the implementation of the LSD multiplier. The architectures are named based on the number of accumulators present in the multiplier.

6.3.1 Single Accumulator Multiplier (SAM)

The Single Accumulator Multiplier (SAM) is similar to the Song/Parhi multiplier architecture as introduced in [76]. This kind of architecture is most commonly used in cryptographic hardware implementations [30, 63]. This architecture consists of three main components as shown in the Fig. 6.1.

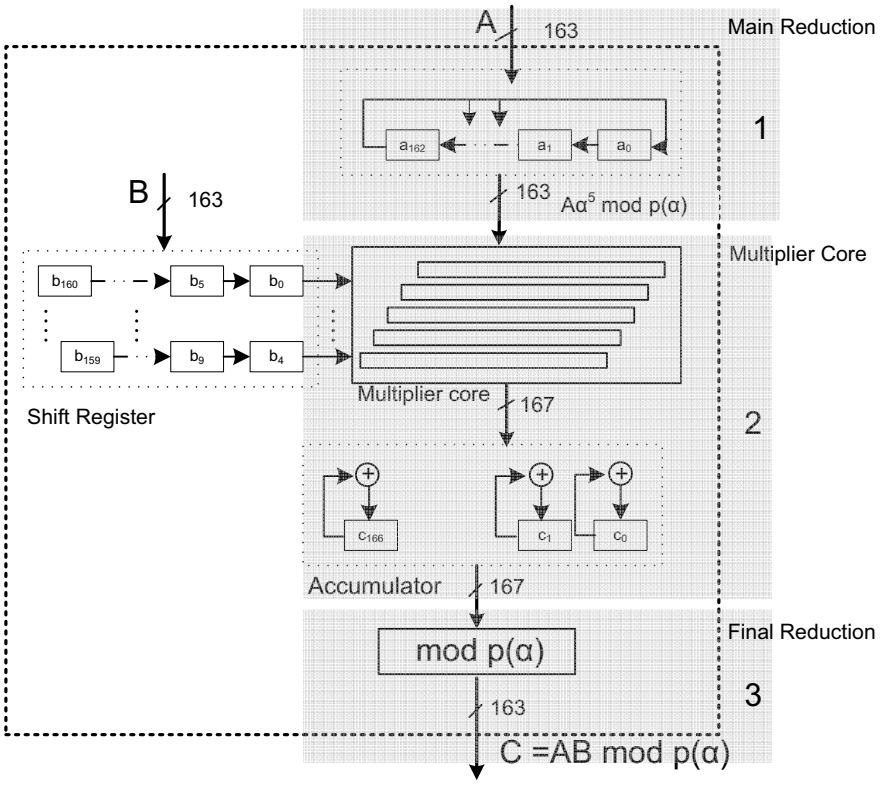


Figure 6.1: LSD-Single Accumulator Multiplier Architecture ($D = 5$) for $\mathbb{F}_{2^{163}}$

- The *main reduction circuit* to shift A left by D positions and to reduce the result mod$p(\alpha)$ (Step 4, Algorithm 2.8).
- The *multiplier core* which computes the intermediate C and stores it in the accumulator (Step 3, Algorithm 2.8).
- The *final reduction circuit* to reduce the contents in the accumulator to get the final result C (Step 6, Algorithm 2.8).

All the components run in parallel, requiring one clock for each step, and the critical path of the whole multiplier normally depends on the critical path of the multiplier core.

We give here a further analysis of the area requirements and the critical path of the different components of the multiplier. In the figures, we will denote an AND gate with a filled dot, and elements to be XORed by a vertical line over them. The number of XOR gates and the critical path is based on the binary tree structure that has to be built to XOR the required elements. For n elements, the number of XOR gates required is $n-1$ and the critical path delay comes out to be the binary tree depth $\lceil \log_2 n \rceil$. We calculate the critical path as a function of the delay of one XOR gate (Δ_{XOR}), and for one AND gate (Δ_{AND}). This allows our analysis to be independent of the cell-technology used for the implementation.

Multiplier core

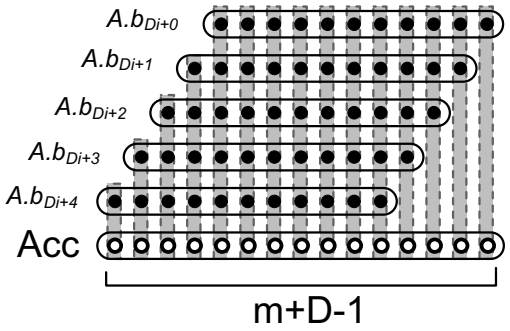


Figure 6.2: SAM multiplier core for $D = 5$

The multiplier core performs the operation $C \leftarrow B_i A + C$ (Step 3 Algorithm 2.8). The implementation of the multiplier core is as shown in Fig. 6.2 for a digit size $D = 5$. It consists of ANDing the multiplicand A with each element of the digit of the multiplier B, XORing the result with the accumulator Acc, and storing it back in Acc. The multiplier core requires mD AND gates (denoted by the black dots), mD XOR gates (for XORing the columns denoted by the vertical line plus the XOR gates for the accumulator) and $m + D - 1$ Flip-Flops (FF) for accumulating the result C.

It can be seen that the multiplier core has a maximum critical path delay of one Δ_{AND} (since all the ANDings in one column are done in parallel), and a delay for

XORing $D+1$ elements as shown in Fig. 6.2. Thus the total critical path delay of the multiplier core is $\Delta_{AND} + \lceil log_2(D+1) \rceil \Delta_{XOR}$.

Main reduction circuit

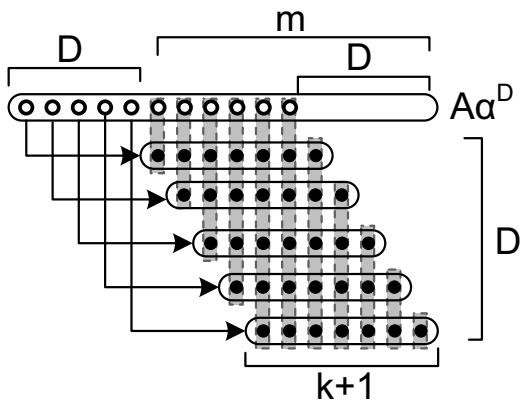


Figure 6.3: SAM main reduction circuit for $D = 5$

The main reduction circuit performs the operation $A \leftarrow A\alpha^D \bmod p(\alpha)$ (Step 4 Algorithm 2.8) and is implemented as shown in Fig. 6.3. Here, the multiplicand A is shifted left by the digit-size D, which is equivalent to multiplying by α^D. The result is then reduced with the reduction polynomial by ANDing the higher D elements of the shifted multiplicand with the reduction polynomial $p(\alpha)$ (shown in the figure as pointed arrows), and XORing the result. We assume that the reduction polynomial is chosen according to Theorem 6.2, which allows reduction to be done in one single step. It can be shown that the critical path delay of the reduction circuit can be at most equal or less than that for the multiplier core.

The main reduction circuit requires $(k+1)$ ANDs and k XORs gates for each reduction element. The number of XOR gates is one less because the last element of the reduction are XORed to the empty elements in the shifted A. Therefore, a total of $(k+1)D$ AND and kD XOR are needed for D digits. Another m Flip-Flops(FF) are needed to store A, and $k+1$ FFs to store the general reduction polynomial.

The critical path of the main reduction circuit (as shown in Fig. 6.3) is one AND (since the ANDings occur in parallel) and the critical path for summation of the D reduction components with the original shifted A. Thus the maximum possible critical

path delay is $\Delta_{AND} + \lceil log_2(D+1) \rceil \Delta_{XOR}$, which is the same as the critical path delay of the multiplier core.

Final reduction circuit

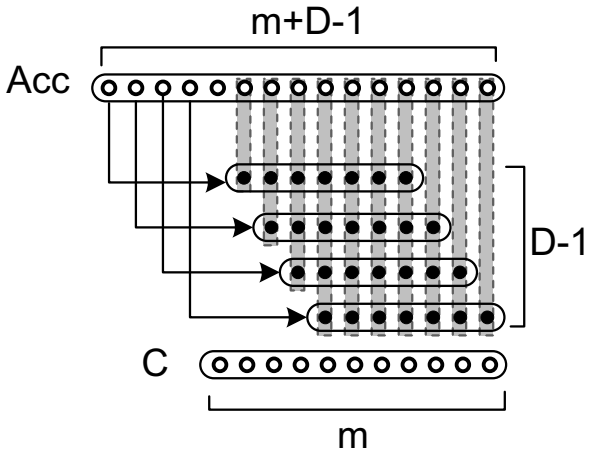


Figure 6.4: SAM final reduction circuit for $D = 5$

The final reduction circuit performs the operation $C \bmod p(\alpha)$, where C is of size $m + D - 1$. It is implemented as shown in Fig. 6.4, which is similar to the main reduction circuit without any shifting. Here, the most significant $(D-1)$ elements are reduced using the reduction polynomial $p(\alpha)$, as shown with the arrows. The area requirement for this circuit is $(k+1)(D-1)$ AND gates and $(k+1)(D-1)$ XOR gates. The critical path of the final reduction circuit is $\Delta_{AND} + \lceil log_2(D) \rceil \Delta_{XOR}$ which is less than that of the main reduction circuit since the degree of the polynomial reduced is one less (Fig. 6.4).

An r-nomial reduction polynomial satisfying Theorem 6.2, i.e., $\sum_{i=0}^{k} p_i = (r-1)$, is a special case and hence the critical path is upper-bounded by that obtained for the general case given here. For a fixed r-nomial reduction polynomial, the area for the main reduction circuit is $(r-1)D$ ANDs and $(r-2)D$ XORs. In addition, we require m flip flops to store intermediate result A. However, no flip flops are needed to store the reduction polynomial as it can be hardwired.

Thus, the total area is given in Table 6.4, and our analysis of the optimum digit

size for critical path and area can be found in Section 5.

6.3.2 Double Accumulator Multiplier (DAM)

The DAM multiplier is the new variant of the SAM multiplier that we propose. They differ in the multiplier core only. Here, we use two accumulators to store the partial product C, such that we can reduce the critical path of the multiplier core. The architecture is shown in the Fig. 6.5. The first accumulator Acc1 adds $\lceil D/2 \rceil$ of the elements, and the other accumulator Acc2 adds the remaining $\lfloor D/2 \rfloor$ elements.

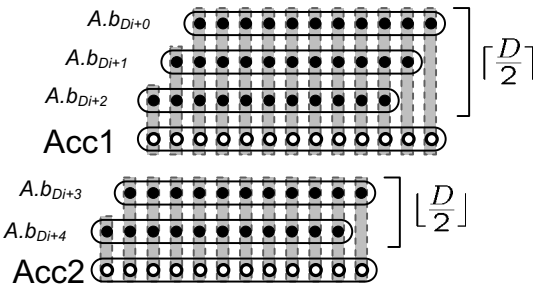


Figure 6.5: DAM multiplier core for $D = 5$

Therefore, the longest critical path in the DAM core is due to the part involving Acc1. The delay here is one AND gate (since ANDings occur in parallel) and the delay for accumulating $\lceil D/2 \rceil + 1$ elements. Thus the critical path delay of the multiplier core is $\Delta_{AND} + \lceil log_2(\lceil D/2 \rceil + 1) \rceil \Delta_{XOR}$. The lower delay has an advantage only if the critical path of the other components (reduction circuits) also have a smaller or equal delay. Therefore, the conditions on the reduction polynomial are more stringent than in the SAM case. We provide here a theorem which shows how to choose such a reduction polynomial.

Theorem 6.3. *Assume an r-nomial irreducible reduction polynomial* $p(\alpha) = \alpha^m + p_k\alpha^k + \sum_{i=0}^{k-1} p_i\alpha^i$, *with* $k \leq m - D$ *and* $\sum_{i=0}^{k} p_i = (r-1)$. *For a digit multiplier implemented using two accumulators for the multiplication core (DAM), the reduction polynomial* $p(\alpha)$ *satisfying the following condition can perform reduction with a smaller critical path than the multiplier core:*

$$\begin{array}{ll} D \leq (m+1)/2\text{:} & \sum_{i=0+j}^{D+j} p_i \leq \lceil D/2 \rceil \quad \textit{for } 0 \leq j < m - 2D + 2 \\ D > (m+1)/2\text{:} & (r-1) \leq \lceil D/2 \rceil \end{array} \tag{6.5}$$

Proof. *There are two different cases which affect how r can be chosen based on the number of reduction elements in the XOR tree. The first, we call the overlap case when there are a maximum of D reducing elements in a column (Fig. 6.6), and the second, the underlap case, where the maximum number of reducing elements in a column is less than D (Fig. 6.7).*

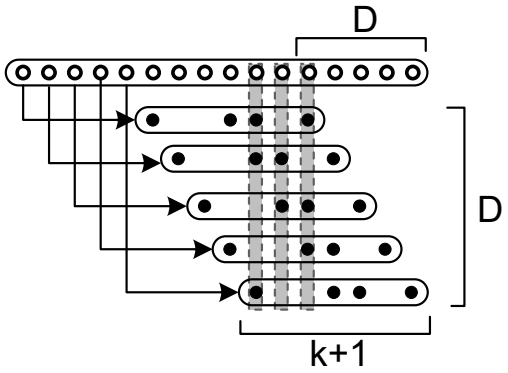


Figure 6.6: Overlap case

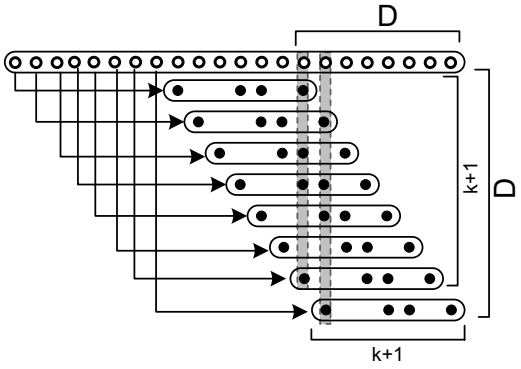


Figure 6.7: Underlap case

Case 1: Overlap

For the reduction elements to overlap, as shown in the Fig. 6.6, the second highest degree of the reduction polynomial k, should satisfy the condition $(k+1) \geq D$. *We denote the number of overlapping columns (shown by the shaded lines) as* $q = k+1-(D-1) \leq m-2D+2$. *If we now analyze each of these columns in the overlapping region, it consists of XORing D consecutive coefficients of* $p(\alpha)$ *with the shifted A. For e.g., in the rightmost column, it is* $(\sum_{i=0}^{D-1} p_i+1)$, *in the next column, it is* $(\sum_{i=1}^{D} p_i+1)$, *and so on. Therefore, the critical path of the circuit is the maximum critical path of any of these columns and should be less than equal to that of the multiplier core. This can be expressed as*

$$\lceil log_2(\textstyle\sum_{i=0+j}^{D-1+j} p_i + 1)\rceil \Delta_{XOR} \leq \lceil log_2(\lceil D/2 \rceil + 1)\rceil \Delta_{XOR} \quad for\ all\ 0 \leq j < m-2D+2 \quad (6.6)$$

The result in Equation (6.5) can be easily obtained from this. The implication of the result is that the sum of any D consecutive coefficients in the reduction polynomial that lie in the overlap region should be less than equal to $\lceil D/2 \rceil$.

Case 2: Underlap

The reduction elements underlaps, as shown in the Fig. 6.7, for the remaining possible values of k, i.e., $(k+1) < D$. The number of reduction elements being added in the underlap region is not more than k. The maximum number of reduction elements that can be present along any column can be $(r-1)$ (like the shaded columns in the figure) since the reduction polynomial is an r-nomial. The condition on the critical path delay is now

$$\lceil log_2(\sum_{i=0}^{k} p_i + 1)\rceil \Delta_{XOR} \leq \lceil log_2(\lceil D/2 \rceil + 1)\rceil \Delta_{XOR} \tag{6.7}$$

This leads to the condition Equation (6.5) given in the theorem. This implies that the sum of all the non-zero coefficients (except the highest degree) in the reduction polynomial should be less than equal to $\lceil D/2 \rceil$.

In the Table 6.1, we provide the possible digit sizes for NIST recommended ECC reduction polynomials. We see that the only difference in the condition between SAM and DAM is for 163-bit reduction polynomial where the digit-size $D = 2$ is not possible. This shows that NIST curves implemented in the SAM architecture can be easily converted to the DAM architecture and does not require any extra constraints.

Table 6.1: NIST recommended reduction polynomial for ECC and digit sizes possible

Reduction Polynomial	possible D	
$p(\alpha)$	**SAM**	**DAM**
$x^{163} + x^7 + x^6 + x^3 + 1$	≤ 156	≤ 156 except $\{2\}$
$x^{233} + x^{74} + 1$	≤ 159	≤ 159
$x^{283} + x^{12} + x^7 + x^5 + 1$	≤ 271	≤ 271
$x^{409} + x^{87} + 1$	≤ 322	≤ 322
$x^{571} + x^{10} + x^5 + x^2 + 1$	≤ 561	≤ 561

The addition of an extra accumulator also increases the size of the multiplier. Therefore, we give an exact count of gates for the new multiplier which allows us to perform a realistic comparison in terms of area-time product with the other multiplier

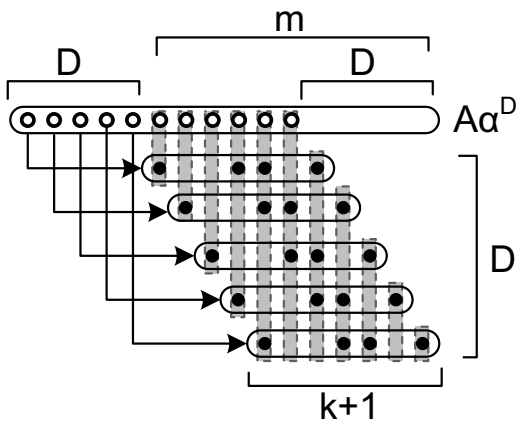


Figure 6.8: DAM main reduction circuit for $D = 5$

designs. Evaluating the size of the multiplier, the three different components of the multiplier require the following area:

- The *multiplier core* needs mD AND gates, and for the XORing we have two accumulators, where accumulator Acc1 needs to XOR $\lceil D/2 \rceil + 1$ elements and accumulator Acc2 needs to XOR $\lfloor D/2 \rfloor + 1$ elements. Hence, the total XOR-gates required are $\lceil D/2 \rceil * m + \lfloor D/2 \rfloor * m = mD$.

  We require $(m + \lceil D/2 \rceil - 1) + (m + \lfloor D/2 \rfloor - 1) = 2m + D - 2$ FFs for the two accumulators. Adding up the two accumulators (which is done with the final reduction) requires additional $m - 1$ XORs which is the overlapping region of the two accumulators.

- The *main reduction circuit* (Fig. 6.8) area is the same as discussed for SAM r-nomial irreducible polynomial: $(r-1)D$ AND, $(r-2)D$ XOR gates and m FF for A.

- *Final reduction* (Fig. 6.9) is done using $(r-1)(D-1)$ AND and $(r-1)(D-1)$ XOR gates.

Remark. *The addition of the two accumulators is done as part of the final reduction circuit in the last cycle. Since the final reduction circuit has a critical path smaller that of the main reduction circuit, the overall critical path is not greater than that of the multiplier core.*

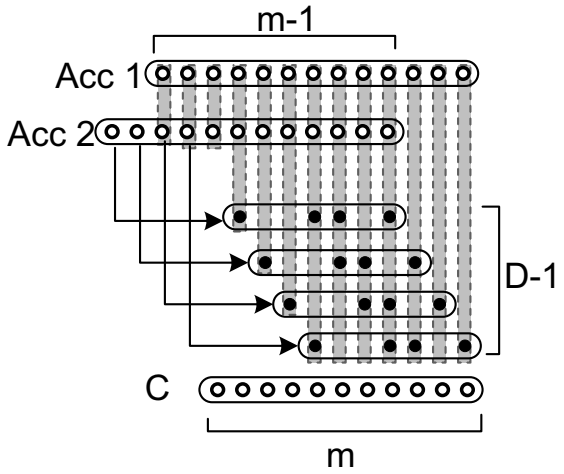


Figure 6.9: DAM final reduction circuit for $D = 5$

6.3.3 N-Accumulator Multiplier (NAM)

The N-Accumulator Multiplier is a more generalized version of the DAM. Here, we try to reduce the critical path further (assuming additional conditions on the reduction polynomial) by having multiple accumulators calculating the partial sum C.

Assuming n accumulators summing the partial sum, the largest critical path in the multiplier core is $\Delta_{AND} + \lceil log_2 \lceil D/n \rceil + 1 \rceil \Delta_{XOR}$. The accumulators are themselves XORed using a different tree of critical path $\lceil log_2 n \rceil \Delta_{XOR}$ in an extra clock at the end (hence the overall latency of the multiplier will be increased by one clock cycle). Care has to be taken that the final accumulation critical path is not greater than that of the multiplier core, i.e., $\lceil log_2 n \rceil \leq \lceil log_2 \lceil D/n \rceil + 1 \rceil$. This is true when the number of accumulators is less than equal to the maximum number of elements XORed in any of the accumulators.

The condition on the reduction polynomial such that the reduction circuit has lesser critical path delay than the multiplier core is an extension of Theorem 6.3 as given below.

Theorem 6.4. *Assume that r-nomial reduction polynomial* $p(\alpha) = \alpha^m + p_k \alpha^k + \sum_{i=0}^{k-1} p_i \alpha^i$, *with* $k \leq m - D$ *and* $\sum_{i=0}^{k} p_i = (r-1)$. *For a digit multiplier implemented using n accumulators for the multiplication core (NAM), the reduction polynomial* $p(\alpha)$ *satisfying the following condition can perform reduction with a smaller*

critical path than the multiplier core:

$$\begin{array}{ll} D \leq (m+1)/2: & \sum_{i=0+j}^{D-1+j} p_i \leq \lceil D/n \rceil \quad \textit{for } 0 \leq j < m-2D+2 \\ D > (m+1)/2: & (r-1) \leq \lceil D/n \rceil \end{array} \tag{6.8}$$

Proof. *Similar to the proof for Theorem 6.3.*

For the calculation of the area requirement for NAM, we assume that each of the accumulator accumulates $q_i,\ 1 \leq i \leq n$ elements such that $\sum_{i=1}^{n} q_i = D$.

- The multiplier core needs mD AND gates and for the XORing we require $q_1 * m + q_2 * m + \ldots + q_n * m = mD$ XOR gates. The FFs required for the accumulators are $(m + q_1 - 1) + (m + q_2 - 1) + \ldots + (m + q_n - 1) = nm + D - n$.
- The main reduction is same as before $(r-1)D$ AND, $(r-2)D$ XOR and m FF for A.
- The final reduction is done using $(r-1)(D-1)$ AND and $(r-1)(D-1)$ XOR gates.
- For the final accumulation, any two adjacent accumulators have only $(m-1)$ elements overlapping. Therefore the total number of XORs for the accumulation tree is $(m-1)(n-1)$. An additional $m + D - 1$ FF are required to store the result as unlike the DAM, the addition is done in a separate clock cycle.

6.4 Evaluation of the Multiplier Options

In this section we first summarize the different architecture options. Table 6.2 shows the latency (in clocks) and the critical path of three different architectures assuming the digit-sizes satisfy the required conditions. The latency for NAM is greater due an extra last cycle to sum all the accumulators.

Table 6.4 shows the area requirement for the proposed architectures. As expected the area is larger for the new architectures, but the area-time product is better in the DAM and NAM case as will be shown in Section 5.

We evaluated the multipliers with the NIST B-163 polynomial (which is in wide-spread use in real-world applications) for different digit sizes to find the optimum values. We use the critical path estimation and latency to calculate the time required for one multiplication. The area is calculated using the estimation we made for each component of the multiplier (tabulated in Table 6.4). Real world estimations are done using the standard cell library from [77] as shown in Table 6.3.

Table 6.2: LSD Multiplier: Latency and critical path

	Latency	**Critical Path**
SAM ($D \geq 2$) r-nomial	$\lceil m/D \rceil + 1$	$1\Delta_{AND} + \lceil \log_2(D+1) \rceil \Delta_{XOR}$
DAM ($D \geq 2$) r-nomial	$\lceil m/D \rceil + 1$	$1\Delta_{AND} + \lceil \log_2(\lceil D/2 \rceil + 1) \rceil \Delta_{XOR}$
NAM ($n \geq 3$, $D > n$) r-nomial	$\lceil m/D \rceil + 2$	$1\Delta_{AND} + \lceil \log_2(\lceil D/n \rceil + 1) \rceil \Delta_{XOR}$

Table 6.3: Area and time complexity of $0.18\mu m$ CMOS standard cells

	A(μm^2)	T (ns)
2-input AND	32	0.6
2-input XOR	56	0.6
D Flip Flop	96	0.6
2:1 Mux	48	0.6

6.4.1 Evaluation of the SAM

Our single accumulator multiplier architecture is very similar to the digit multipliers used in the open literature. In order to allow an analysis of this multiplier, we plot Fig. 6.10. This plot shows the time taken to complete one multiplication for different digit sizes for SAM.

Concluding from Fig. 6.10, one realizes that the digit-size D equal to a powers of 2, like $4, 8, 16, 32, 64$, are the local worse values. However, these kind of D values are normally used as digit-sizes for ECC implementations [30, 63].

In addition, one can see that values of D of the form $2^l - 1$ are optimum because of the optimum binary tree structure they generate. Since, the multiplier is the most important component in these cryptosystems which also dictates the overall efficiency,

changing the digit sizes to the more optimum values can give a much better performance.

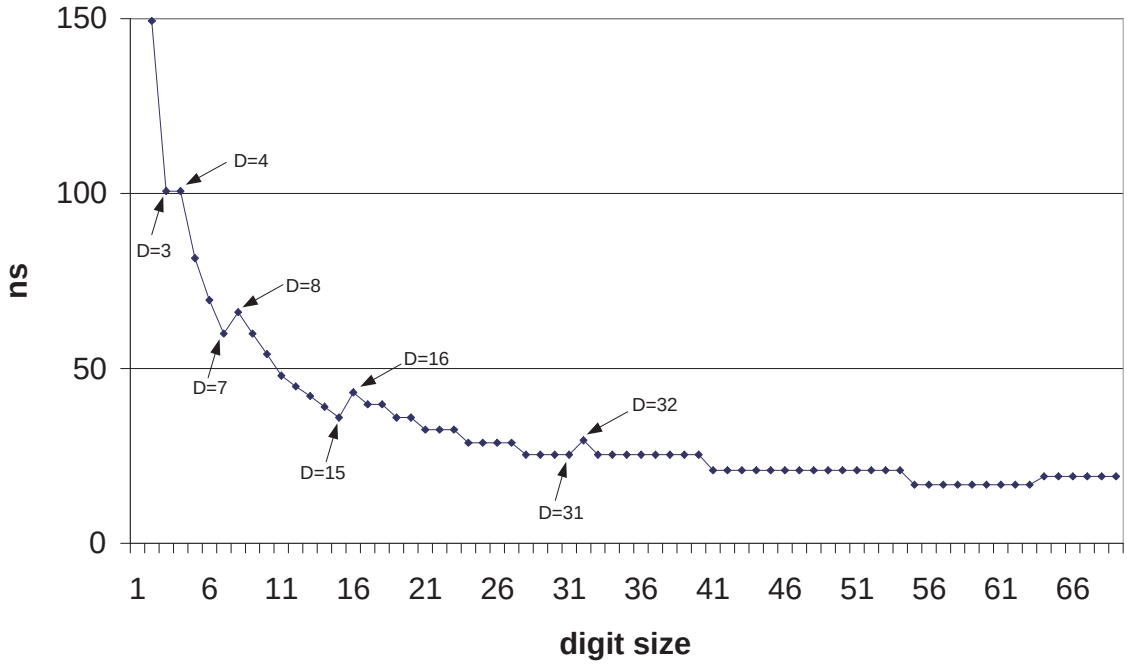


Figure 6.10: Time to complete one multiplication of the single accumulator multiplier

We can generate a very fast design by using a lot of resources, hence a large digit-size leads to a faster multiplier. Thus, one has to consider speed and area in order to get the optimum multiplier, like using the area-time product. Fig. 6.11 we draw the area-time product over different digit-sizes of the single accumulator multiplier. This plot clearly shows that most commonly used digit sizes are not only slower but also inefficient in terms of the area-time product used. Better and faster implementations of public key cryptography can be obtained using traditional LSD SAM multiplier by choosing $D = 2^l - 1$. For e.g., a SAM multiplier with D=3 can compute a multiplication in the same time as D=4, but will require much smaller area. For D=7, the multiplier would compute the result faster than D=8 with 10% lesser area.

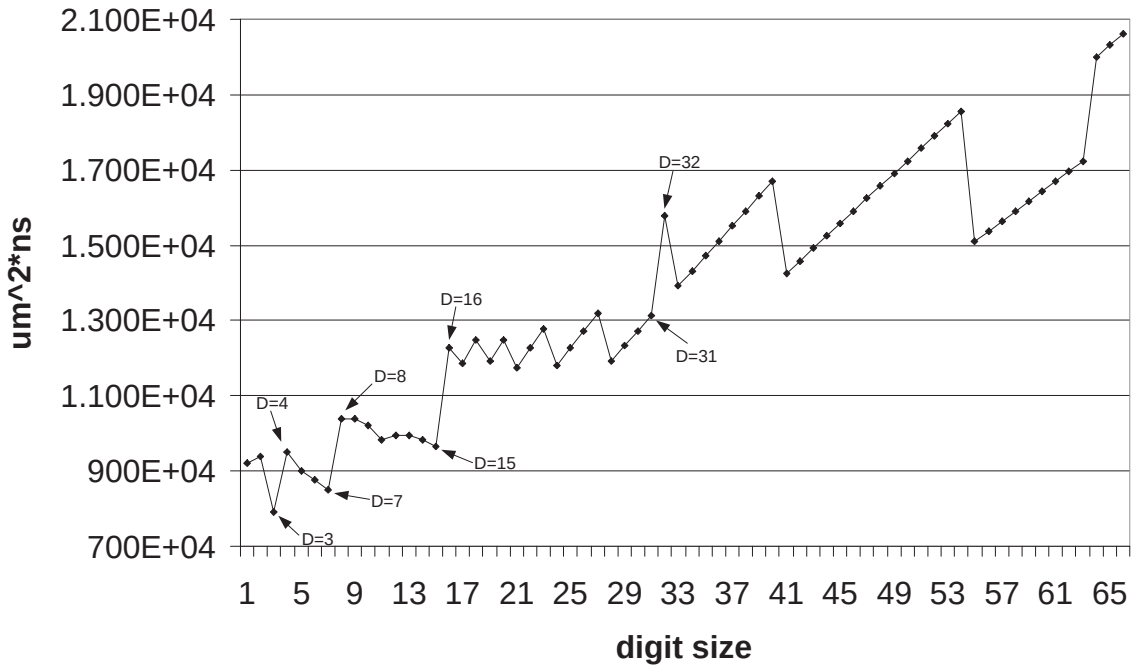


Figure 6.11: Area-Time Product of the single accumulator multiplier

6.4.2 Evaluation of all multiplier options

In this subsection, we compare all our introduced multiplier options. Fig. 6.12 shows the time requirements for the different multipliers. As expected, DAM and NAM (n=3) architectures are faster than SAM. Its important to note that the optimum digit sizes changes for different architectures. This is because of the optimum tree structures which are formed at different sizes of D within the DAM and NAM architecture. For e.g., one should rather use D=4 for NAM, whereas this digit size will be not optimal for DAM and SAM. For DAM and SAM we rather would use D=3.

The area-time product of the multipliers is plotted in Fig. 6.13. This shows that DAM and NAM are also efficient architectures when we consider speed and resources of hardware used.

NAM can be inefficient for small D sizes because of the extra overhead in area due to the registers and the extra clock cycle in the last step. The designer has to choose the right architecture based on the various constraints like area and speed. For e.g., if the designer had a compromise of area and speed at D=8 for a SAM architecture, then

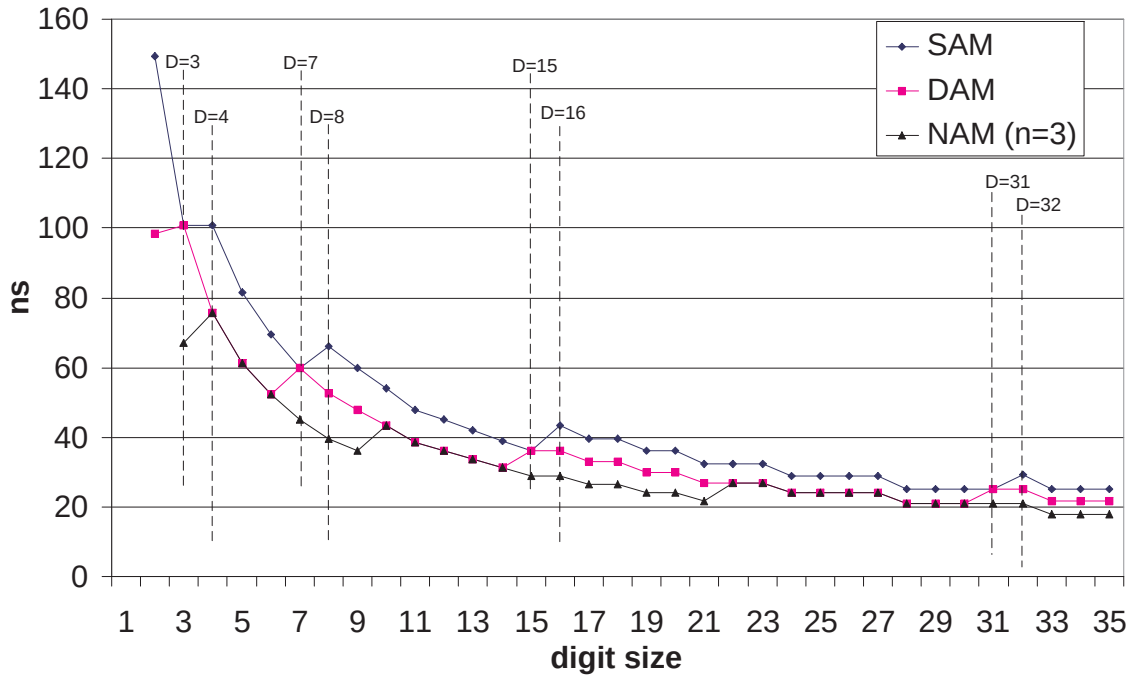


Figure 6.12: Time to complete one multiplication of all the different multiplier implementations

he can either implement it with the same or smaller area with SAM (D=7) or DAM with D=6 or NAM (n=3) with D=3 with much better speed. This extra flexibility eventually allows the designer to build more optimum cryptosystems than presently available.

6.5 Summary

In this contribution, we showed new architectures for implementing LSD multipliers. The conditions that apply on the irreducible polynomial to successfully implement such architectures are given. It can be seen that all NIST recommended polynomials easily satisfy these conditions, which make these architectures very promising for implementing curve based public key cryptosystems. An evaluation of the multipliers for different digit sizes provide optimum values of D which give the best efficiency for a required speed. This has enabled us to show that present digit-sizes being used are the

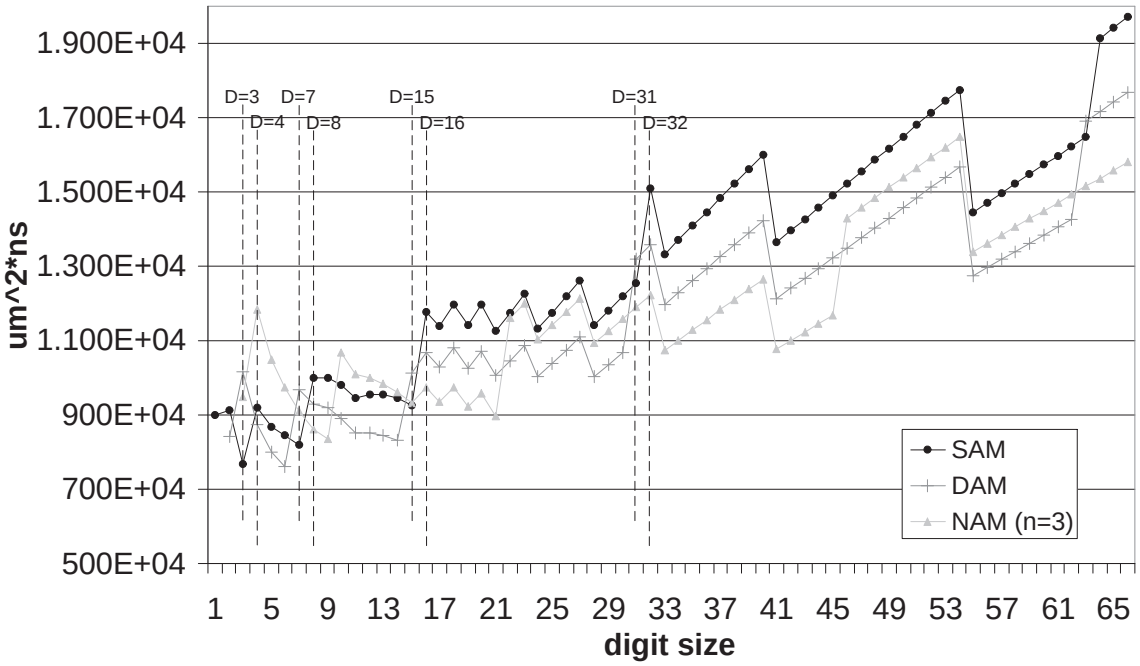


Figure 6.13: Area-Time Product of the different multiplier implementations

worst choices and much better implementations are possible. The different possible architectures also provide the designer with more flexibility in making the compromise between area and time which is inherent in all implementations.

Table 6.4: LSD Multiplier: Area

	# XOR	**# AND**	**# FF**
SAM ($D \geq 2$)			
general	$(m+k)D + (k+1)(D-1)$	$(m+k+1)D + (k+1)(D-1)$	$2m + D + k$
r-nomial	$(m+r-2)D + (r-1)(D-1)$	$(m+r-1)D + (r-1)(D-1)$	$2m + D - 1$
DAM ($D > 2$)	$(m+r-2)D + (r-1)(D-1)$	$(m+r-1)D + (r-1)(D-1)$	$3m + D - 2$
r-nomial	$+(m-1)$		
NAM ($n \geq 3$,	$(m+r-2)D + (r-1)(D-1)$	$(m+r-1)D + (r-1)(D-1)$	$(n+2)m + 2D$
$D > n$), r-nomial	$+(m-1)(n-1)$		$-(n+1)$

Chapter 7

Hardware Design: ECC in the Frequency Domain

We present here the results of the collaborative work with Selcuk Baktır, Worcester Polytechnic Institute, USA.

7.1 Motivation and Outline

Efficiency of an elliptic curve cryptosystem is highly dependent on the underlying finite field arithmetic. When performing ECC in $\mathbb{F}_{p^m}$, field multiplication can be achieved with a quadratic number of multiplications and additions in the base field $\mathbb{F}_p$ using the classical polynomial multiplication method. Using the Karatsuba algorithm, this complexity can be reduced significantly, however one still needs to do a subquadratic number of multiplications and additions in the base field $\mathbb{F}_p$. Multiplication operation is inherently much more complex than other operations such as addition, therefore it is desirable that one performs as small number of base field multiplications as possible for achieving an extension field multiplication. The *DFT modular multiplication*[7] achieves multiplication in $\mathbb{F}_{p^m}$ in the frequency domain with only a linear number of base field $\mathbb{F}_p$ multiplications in addition to a quadratic number of simpler base field operations such as additions/subtractions and bitwise rotations.

In an ECC processor the multiplier unit usually consumes a substantial area on the chip, therefore it is crucial that one uses an area/time efficient multiplier, particularly in environments such as wireless sensor networks where resources are precious. In this work, we address this issue by proposing an area/time efficient ECC processor architecture utilizing *DFT modular multiplication* in optimal extension fields (OEF)[5, 6] with the Mersenne prime field characteristic $p = 2^n - 1$ and the extension degree

$m = n$.

This chapter is organized as follows. In Section 7.2, we provide some background information on OEF multiplication in the frequency domain. In Section 7.3, we overview the *DFT modular multiplication* algorithm and also present some optimization ideas for efficient implementation of this algorithm in hardware. In Section 7.4, we present an efficient ASIC implementation of the elliptic curve cryptographic processor design using AMI Semiconductor $0.35\mu m$ CMOS technology which utilizes an *optimized DFT modular multiplier architecture* over $\mathbb{F}_{(2^{13}-1)^{13}}$, $\mathbb{F}_{(2^{17}-1)^{17}}$ and $\mathbb{F}_{(2^{19}-1)^{19}}$. Finally in Section 7.5, we present our implementation results.

7.2 Mathematical Background

OEFs are found to be successful in ECC implementations in software where resources such as computational power and memory are constrained, as shown in Chapter 3 and Chapter 5. In this work, we propose an efficient hardware architecture which performs the extension field multiplication operation (described in Section 2.5.1), in the frequency domain. For this, we need to first represent the operands in the frequency domain. To convert an element in $\mathbb{F}_{p^m}$ into the frequency domain representation, the *number theoretical transform* is used, which is explained next.

7.2.1 Number Theoretic Transform (NTT)

Number theoretic transform over a ring, also known as *the discrete Fourier transform (DFT) over a finite field*, was introduced by Pollard [66]. For a finite field $\mathbb{F}_p$ and a sequence (a) of length d whose entries are from $\mathbb{F}_p$, the forward DFT of (a) over $\mathbb{F}_p$, denoted by (A), can be computed as

$$A_j = \sum_{i=0}^{d-1} a_i r^{ij} \ , \ 0 \leq j \leq d-1 \ . \tag{7.1}$$

Here, we refer to the elements of (a) and (A) by a_i and A_i, respectively, for $0 \leq i \leq d-1$. Likewise, the inverse DFT of (A) over F_q can be computed as

$$a_i = \frac{1}{d} \cdot \sum_{j=0}^{d-1} A_j r^{-ij} \ , \ 0 \leq i \leq d-1 \ . \tag{7.2}$$

We will refer to the sequences (a) and (A) as the *time and frequency domain representations*, respectively, of the same sequence. The above DFT computations over the finite field $\mathbb{F}_p$ are defined by utilizing a d^{th} primitive root of unity, denoted by r, from $\mathbb{F}_p$ or a finite extension of $\mathbb{F}_p$. In this work, we will use $r = -2 \in \mathbb{F}_p$ as it enables efficient implementation in hardware which will be further discussed in Section 7.3.3. We will consider only finite fields $\mathbb{F}_{p^m}$ with a Mersenne prime characteristic $p = 2^n - 1$ and odd extension degree $m = n$. This makes the sequence length $d = 2m$, since $r = -2$ is a $(2m)^{th}$ primitive root of unity in the base field $\mathbb{F}_{(2^n-1)}$. In this case when $r = -2$ and $p = 2^n - 1$ a modular multiplication in $\mathbb{F}_p$ with a power of r can be achieved very efficiently with a simple bitwise rotation (in addition to a negation if the power is odd). The discrete Fourier transform computed modulo a Mersenne prime, as in our case, is called *Mersenne transform* [67].

7.2.2 Convolution Theorem and Polynomial Multiplication in the Frequency Domain

A significant application of the Fourier transform is convolution. Convolution of two d-element sequences (a) and (b) in the time domain results in another d-element sequence (c), and can be computed as follows:

$$c_i = \sum_{j=0}^{d-1} a_j b_{i-j \bmod d} \ , \quad 0 \leq i \leq d-1 \ . \tag{7.3}$$

According to the convolution theorem, the above convolution operation in the time domain is equivalent to the following computation in the frequency domain:

$$C_i = A_i \cdot B_i \ , \quad 0 \leq i \leq d-1 \ , \tag{7.4}$$

where (A), (B) and (C) denote the discrete Fourier transforms of (a), (b) and (c), respectively. Hence, convolution of two d-element sequences in the time domain, with complexity $O(d^2)$, is equivalent to simple pairwise multiplication of the DFTs of these sequences and has a surprisingly low $O(d)$ complexity.

Note that the summation in Eq. 7.3) is the *cyclic convolution* of the sequences (a) and (b). We have seen that this cyclic convolution can be computed very efficiently in the Fourier domain by pairwise coefficient multiplications. Multiplication of two polynomials on the other hand is equivalent to the *acyclic (linear) convolution* of the polynomial coefficients. However, if we represent elements of $\mathbb{F}_{p^m}$, which are $(m-1)^{st}$

degree polynomials with coefficients in $\mathbb{F}_p$, with at least $d = (2m-1)$ element sequences by appending zeros at the end, then the cyclic convolution of two such sequences will be equivalent to their acyclic convolution and hence gives us their polynomial multiplication.

One can form sequences by taking the ordered coefficients of polynomials. For instance,

$$a(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-1} x^{m-1} \ ,$$

an element of $\mathbb{F}_{p^m}$ in polynomial representation, can be interpreted as the following sequence after appending $d - m$ zeros to the right:

$$(a) = (a_0, a_1, a_2, \ldots, a_{m-1}, 0, 0, \ldots, 0) \ . \tag{7.5}$$

For $a(x), b(x) \in \mathbb{F}_{p^m}$, and for $d \geq 2m - 1$, the cyclic convolution of (a) and (b) yields a sequence (c) whose first $2m - 1$ entries can be interpreted as the coefficients of a polynomial $c(x)$ such that $c(x) = a(x) \cdot b(x)$. The computation of this cyclic convolution can be performed by simple pairwise coefficient multiplications in the discrete Fourier domain.

Note, that using the convolution property the polynomial product $c(x) = a(x) \cdot b(x)$ can be computed very efficiently in the frequency domain, but the final reduction by the field generating polynomial is not performed. For further multiplications to be performed on the product $c(x)$ in the frequency domain, it needs to be first reduced modulo the field generating polynomial. *DFT modular multiplication* algorithm, which will be mentioned briefly in the following section, performs both polynomial multiplication and modular reduction in the frequency domain and thus makes it possible to perform consecutive modular multiplications in the frequency domain.

7.3 Modular Multiplication in the Frequency Domain

To the best of our knowledge, *DFT modular multiplication* algorithm [7], which performs Montgomery multiplication in $\mathbb{F}_{p^m}$ in the frequency domain, is the only existing frequency domain multiplication algorithm to achieve efficient modular multiplication for operand sizes relevant to *Elliptic Curve Cryptography*. In this section, we give a brief overview of this algorithm and the notation used.

7.3.1 Mathematical Notation

Since the DFT modular multiplication algorithm runs in the frequency domain, the parameters used in the algorithm are in their frequency domain sequence representations. These parameters are the input operands $a(x), b(x) \in \mathbb{F}_{p^m}$, the result $c(x) = a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) \in \mathbb{F}_{p^m}$, irreducible field generating polynomial $f(x)$, normalized irreducible field generating polynomial $f'(x) = f(x)/f(0)$, and the indeterminate x. The time domain sequence representations of these parameters are $(a), (b), (c), (f), (f')$ and (x), respectively, and their frequency domain sequence representations, i.e., the discrete Fourier transforms of the time domain sequence representations, are $(A), (B), (C), (F), (F')$ and (X). We will denote elements of a sequence with the name of the sequence and a subscript for showing the location of the particular element in the sequence, e.g., for the indeterminate x represented as the following d-element sequence in the time domain

$$(x) = (0, 1, 0, 0, \cdots, 0) \ ,$$

the DFT of (x) is computed as the following d-element sequence

$$(X) = (1, r, r^2, r^3, r^4, r^5, \ldots, r^{d-1}) \ ,$$

whose first and last elements are denoted as $X_0 = 1$ and $X_{d-1} = r^{d-1}$, respectively.

7.3.2 DFT Modular Multiplication Algorithm

DFT Modular Multiplication shown in Algorithm 7.1, consists of two parts: Multiplication (Steps 1 and 3) and Montgomery reduction (Steps 4 through 13). Multiplication is performed simply by pairwise multiplication of the two input sequences (A) and (B). This multiplication results in (C) (which corresponds to $c(x) = a(x) \cdot b(x)$), a polynomial of degree at most $2m - 2$. For performing further multiplications over $c(x)$ using the same method in the *frequency domain*, one needs to first reduce it modulo $f(x)$, so that its time domain representation is of degree at most $m - 1$.

DFT modular multiplication algorithm performs reduction in the frequency domain by *DFT Montgomery Reduction* (Steps 3 to 9). The inputs to DFT Montgomery reduction are the frequency domain sequence representation (C) of $c(x) = a(x) \cdot b(x)$ and its output is the sequence (C) corresponding to $c(x) = a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) \in \mathbb{F}_{p^m}$. DFT Montgomery reduction is a direct adaptation of Montgomery reduction. In the frequency domain, the value S is computed such that $(c(x) + S \cdot (f'(x)))$ is a multiple of

Algorithm 7.1 DFT modular multiplication algorithm for $\mathbb{F}_{p^m}$

Input: $(A) \equiv a(x) \in \mathbb{F}_{p^m}$, $(B) \equiv b(x) \in \mathbb{F}_{p^m}$
Output: $(C) \equiv a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) \in \mathbb{F}_{p^m}$

1: **for** $i = 0$ to $d-1$ **do**
2: $\quad C_i \leftarrow A_i \cdot B_i$
3: **end for**
4: **for** $j = 0$ to $m-2$ **do**
5: $\quad S \leftarrow 0$
6: $\quad$ **for** $i = 0$ to $d-1$ **do**
7: $\quad\quad S \leftarrow S + C_i$
8: $\quad$ **end for**
9: $\quad S \leftarrow -S/d$
10: $\quad$ **for** $i = 0$ to $d-1$ **do**
11: $\quad\quad C_i \leftarrow (C_i + F'_i \cdot S) \cdot X_i^{-1}$
12: $\quad$ **end for**
13: **end for**
14: Return (C)

x. Note that $(c(x)+S\cdot(f'(x)))$ is equivalent to $c(x) \bmod f(x)$. The algorithm then divides $(c(x)+S\cdot f'(x))$ by x and obtains a result which is congruent to $c(x)\cdot x^{-1} \bmod f(x)$. By repeating this $m-1$ times (Steps 4 to 13) the initial input which is the $(2m-2)^{nd}$ degree input polynomial $c(x) = a(x)\cdot b(x)$ is reduced to the final $(m-1)^{st}$ degree result which is congruent to $a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x)$. Hence, for the inputs $a(x) \cdot x^{m-1}$ and $b(x) \cdot x^{m-1}$, both in $\mathbb{F}_{p^m}$, the DFT modular multiplication algorithm computes $a(x) \cdot b(x) \cdot x^{m-1} \in \mathbb{F}_{p^m}$ and thus the Montgomery residue representation is kept intact and further computations can be performed in the frequency domain using the same algorithm.

7.3.3 Optimization

In this work, we show that for the case of $r = -2$, odd m and $n = m$, i.e., when the bit length of the field characteristic $p = 2^n - 1$ is equal to the field extension degree, the DFT modular multiplication can be optimized by precomputing some intermediary values in the algorithm. Our optimization takes advantage of the fact that when $r = -2$, $p = 2^n - 1$, the field generating polynomial is $f(x) = x^m - 2$ and hence $f'(x) = -\frac{1}{2} \cdot x^m + 1$, m is odd and $m = n$, the following equalities hold in $\mathbb{F}_p$:

$$F_i' = -\frac{1}{2} \cdot (-2)^{mi} + 1 = \begin{cases} -\frac{1}{2} + 1 = \frac{1}{2}\,, & i \ \ even \\ \\ \frac{1}{2} + 1\,, & i \ \ odd \end{cases} \tag{7.6}$$

(7.7)

This equality holds since

$$(-2)^{mi} \equiv (-2)^{ni} \equiv (-1)^{ni}(2^n)^i \equiv (-1)^{ni} \pmod{p}\,.$$

Note that in this case F_i' has only two distinct values, namely $-\frac{1}{2} + 1 = \frac{1}{2}$ and $\frac{1}{2} + 1$ for the irreducible field generating polynomial $f(x) = x^m - 2$. Hence, $F_i' \cdot S$ in Step 11 of the Algorithm 7.1 can attain only two values for any distinct value of S and these values can be precomputed outside the loop avoiding all such computations inside the loop. The pre-computations can be achieved very efficiently with only one bitwise rotation and one addition. With the suggested optimization, both the number of base field additions/subtractions and the number of base field bitwise rotations required to perform an extension field multiplication are reduced by $d(m-1) = 2m(m-1)$.

n	$p = 2^n - 1$	m	d	r	equivalent binary field size
13	8191	13	26	-2	$\sim 2^{169}$
17	131071	17	34	-2	$\sim 2^{289}$
19	524287	19	38	-2	$\sim 2^{361}$

Table 7.1: List of parameters suitable for optimized DFT modular multiplication.

In Algorithm 7.2, we present the optimized algorithm for the irreducible polynomial $f(x) = x^m - 2$. In Table 7.1, we suggest a list of parameters for implementation of the optimized algorithm over finite fields of different sizes. Note that these parameters are perfectly suited for *Elliptic Curve Cryptography*. In Section 7.5 we provide implementation results for all the finite fields listed in Table 7.1 and thus show the relevance of the optimized algorithm for area/time efficient hardware implementation of ECC in constrained environments.

Algorithm 7.2 Optimized DFT modular multiplication in $\mathbb{F}_{p^m}$ for $r = -2$, $p = 2^n - 1$, m odd, $m = n$ and $f(x) = x^m - 2$

Input: $(A) \equiv a(x) \in \mathbb{F}_{p^m}$, $(B) \equiv b(x) \in \mathbb{F}_{p^m}$
Output: $(C) \equiv a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x) \in \mathbb{F}_{p^m}$

1: **for** $i = 0$ to $d - 1$ **do**
2: $C_i \leftarrow A_i \cdot B_i$
3: **end for**
4: **for** $j = 0$ to $m - 2$ **do**
5: $S \leftarrow 0$
6: **for** $i = 0$ to $d - 1$ **do**
7: $S \leftarrow S + C_i$
8: **end for**
9: $S \leftarrow -S/d$
10: $S_{half} \leftarrow S/2$
11: $S_{even} \leftarrow S_{half}$
12: $S_{odd} \leftarrow S + S_{half}$
13: **for** $i = 0$ to $d - 1$ **do**
14: **if** $i \bmod 2 = 0$ **then**
15: $C_i \leftarrow C_i + S_{even}$
16: **else**
17: $C_i \leftarrow -(C_i + S_{odd})$
18: **end if**
19: $C_i \leftarrow C_i/2^i$
20: **end for**
21: **end for**
22: Return (C)

7.4 Implementation of an ECC Processor Utilizing DFT Modular Multiplication

The DFT modular multiplication algorithm trades off computationally expensive modular multiplication operations for simple bitwise rotations which can be done practically for free in hardware by proper rewiring. In this section, we present a hardware implementation of an ECC processor using the DFT modular multiplication algorithm. We exemplarily use the field $\mathbb{F}_{(2^{13}-1)^{13}}$ to explain our design, although the design is easily extendable for other parameter sizes mentioned in Table 7.1 and the implementation results for all parameter sizes are given in Section 7.5.

We first describe the implementation of the base field arithmetic in $\mathbb{F}_{(2^{13}-1)}$ and then make parameter decisions based on the Algorithm 7.2 to implement an efficient DFT modular multiplier. Then, we present the overall processor design to compute the ECC scalar point multiplication.

7.4.1 Base Field Arithmetic

Base field arithmetic consists of addition, subtraction (or negation) and multiplication in $\mathbb{F}_{(2^{13}-1)}$. The arithmetic architectures are designed to ensure a area/time efficiency.

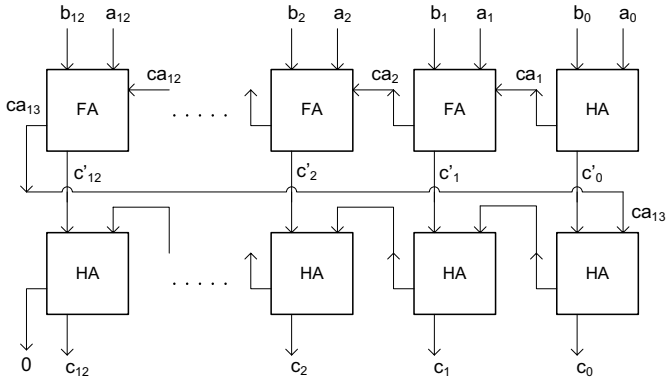


Figure 7.1: Base Field Addition Architecture

Base Field Addition and Subtraction

Addition in the base field is implemented using a ripple carry adder. Reduction with the Mersenne prime $p = 2^{13} - 1$ is just an additional addition of the carry generated. This additional addition is performed independently of the value of the carry to avoid any timing related attacks. Fig. 7.1 shows the design of the base field adder built using half adders (HA) and full adders (FA).
Negation in the base field is extremely simple to implement with a Mersenne prime p as the field characteristic. Negation of $B \in \mathbb{F}_p$ is normally computed as $B' = p - B$. However, when p is a Mersenne, it is easy to see from the binary representation of $p = 2^{13} - 1$ (which are all 1's) that this subtraction is equivalent to flipping (NOT) of the bits of B. Hence, subtraction in this architecture can be implemented by using the adder architecture with an additional bitwise NOT operation on the subtractor.

Base Field Multiplication

Base field multiplication is a 13×13-bit integer multiplication followed by a modular reduction with $p = 2^{13} - 1$. Since p is a Mersenne prime, we can easily implement the integer multiplication with interleaved reduction. Fig. 7.2 shows the design of our multiplier architecture. It consists of the multiplication core (which performs the integer multiplication with interleaved reduction of the carry) and a final reduction of the result which is performed using a ripple carry adder.

The processing cell of the multiplier core, which is shown in Fig. 7.3, is built with a full adder (FA). Here, a_i and b_i represent the inputs, ca_i represents the carry, and i and k are the column and row numbers, respectively, in Fig. 7.2.

7.4.2 Extension Field Multiplication

Finite field multiplication in an extension field is computed using a polynomial multiplier with coefficients in the base field, and then reducing the result with the reduction polynomial as described in Section 2.5.1. Using an extension field $\mathbb{F}_{p^m}$ can reduce the area of a finite field multiplier in a natural way since in this case only a smaller base field multiplier is required. For instance, for performing multiplication in $\mathbb{F}_{(2^{13}-1)^{13}}$ one would only need a 13×13-bit base field multiplier. However, an implementation in the time domain has the disadvantage of a quadratic time complexity because a total number of $13 \times 13 = 169$ base field multiplications need to be computed. In our design,

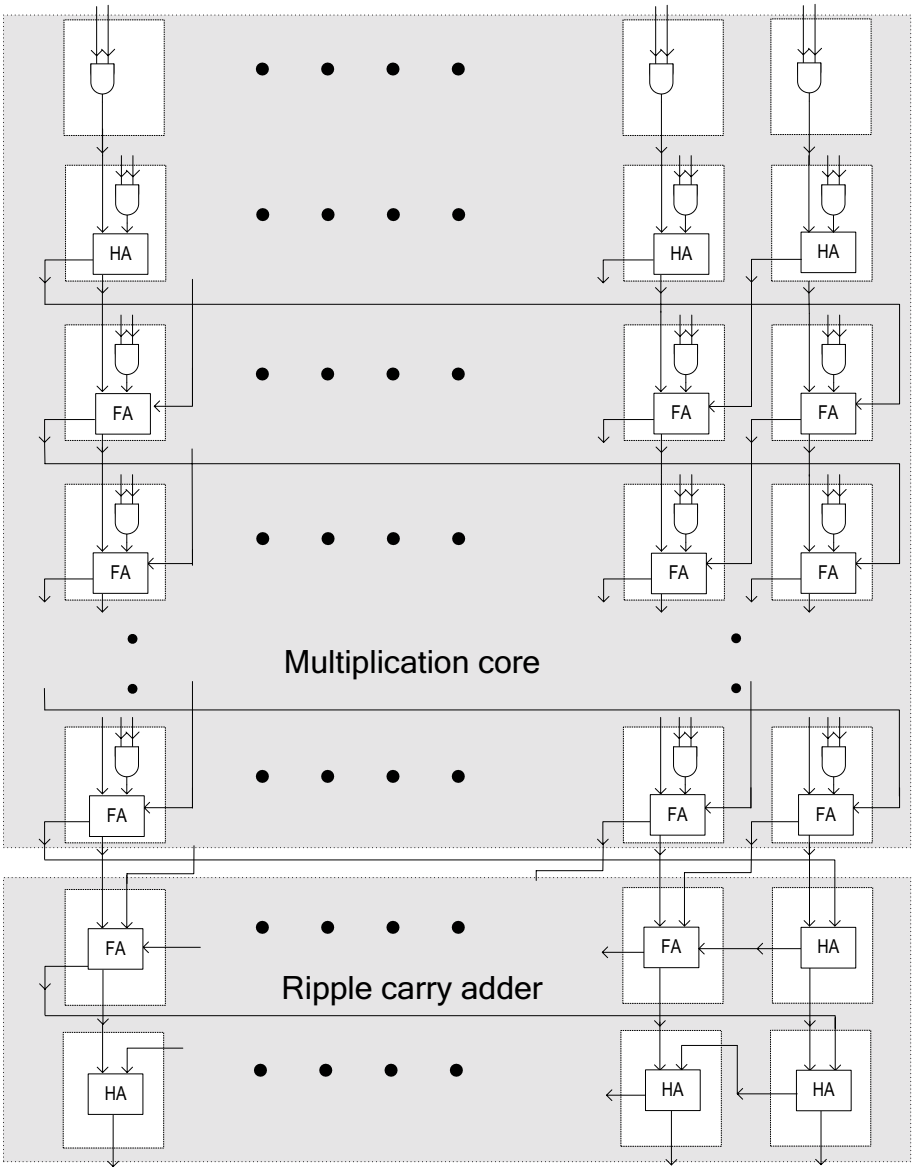


Figure 7.2: Base Field Multiplication with Interleaved Reduction

we save most of these base field multiplications by utilizing *DFT modular multiplication* since we now have to perform only a linear number of 26 base field multiplications (Steps 1-3, Algorithm 7.2).

DFT Montgomery Multiplication

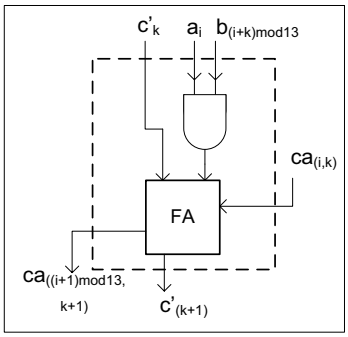


Figure 7.3: Processing Cell for the Base Field Multiplier Core

For the application of DFT modular multiplication, Algorithm 7.2 is modified for an optimized hardware implementation. The main design decision here is to use a single base field multiplier to perform Step 2 (for pairwise coefficient multiplications) and Step 9 (for multiplications with the constant $-1/d$). Next, the two loops Steps 6-8 (for accumulating C_i's) and Steps 13-20 (for computing C_i's) were decided to be performed together simultaneously. The final design that emerged is as shown in Fig. 7.4 and the functionality is represented by the pseudo-code in Algorithm 7.3. During Steps 2-5 (Algorithm 7.3), the multiplexer select signal *red* is set to 0 and later it is set to 1 for the remaining steps. This allows the *MUL* (the base field multiplier) to be used for the initial multiplication with the proper results accumulated in the register S. Registers S_{even} and S_{odd} cyclically rotate its contents every clock cycle performing the Steps 17 and 18 (Algorithm 7.3), respectively.

Step 19 in Algorithm 7.2, which involves different amounts of cyclic rotations for different C_i would normally be inefficient to implement in hardware. In this work, this problem is solved in a *unique* way with the *FIFO (First-In First-Out) cyclic register block*. This temporary memory location for storing C_i values pushes in values till it is completely full. In the next loop, as the values are moved out of the *FIFO*, each of them is cyclically rotated at each clock cycle as it moves up. Hence the different C_i values are cyclically rotated by different number of bits with no extra cost. The pseudo-code which shows this functionality of the memory block is given in Steps 19-21 of Algorithm 7.3. Steps 14-18 (Algorithm 7.2) are implemented using the two multiplexors with the select signal o_e.

Thus, in this work the steps of the original DFT modular multiplication algorithm (Algorithm 7.2) are reordered so as to fine tune it to generate a hardware efficient

Algorithm 7.3 Pseudo-code for hardware implementation of DFT multiplier

Input: (A), (B)

Output: $(C) \equiv a(x) \cdot b(x) \cdot x^{-(m-1)} \bmod f(x)$

1: $S \leftarrow 0$
2: **for** $i = 0$ to $d - 1$ **do**
3: $\quad C_i \leftarrow A_i \cdot B_i$
4: $\quad S \leftarrow S + C_i$
5: **end for**
6: **for** $j = 0$ to $m - 2$ **do**
7: $\quad S \leftarrow -S/d$
8: $\quad S_{even} \leftarrow S/2$
9: $\quad S_{odd} \leftarrow S + S/2$
10: $\quad S \leftarrow 0$
11: $\quad$ **for** $i = 0$ to $d - 1$ **do**
12: $\quad\quad$ **if** $i \bmod 2 = 0$ **then**
13: $\quad\quad\quad C_i \leftarrow C_i + S_{even}$
14: $\quad\quad$ **else**
15: $\quad\quad\quad C_i \leftarrow -(C_i + S_{odd})$
16: $\quad\quad$ **end if**
17: $\quad\quad S_{even} \leftarrow S_{even}/2$
18: $\quad\quad S_{odd} \leftarrow S_{odd}/2$
19: $\quad\quad$ **for** $k = i + 1$ to $d - 1$ **do**
20: $\quad\quad\quad C_k \leftarrow C_k/2$
21: $\quad\quad$ **end for**
22: $\quad\quad S \leftarrow S + C_i$
23: $\quad$ **end for**
24: **end for**
25: Return (C)

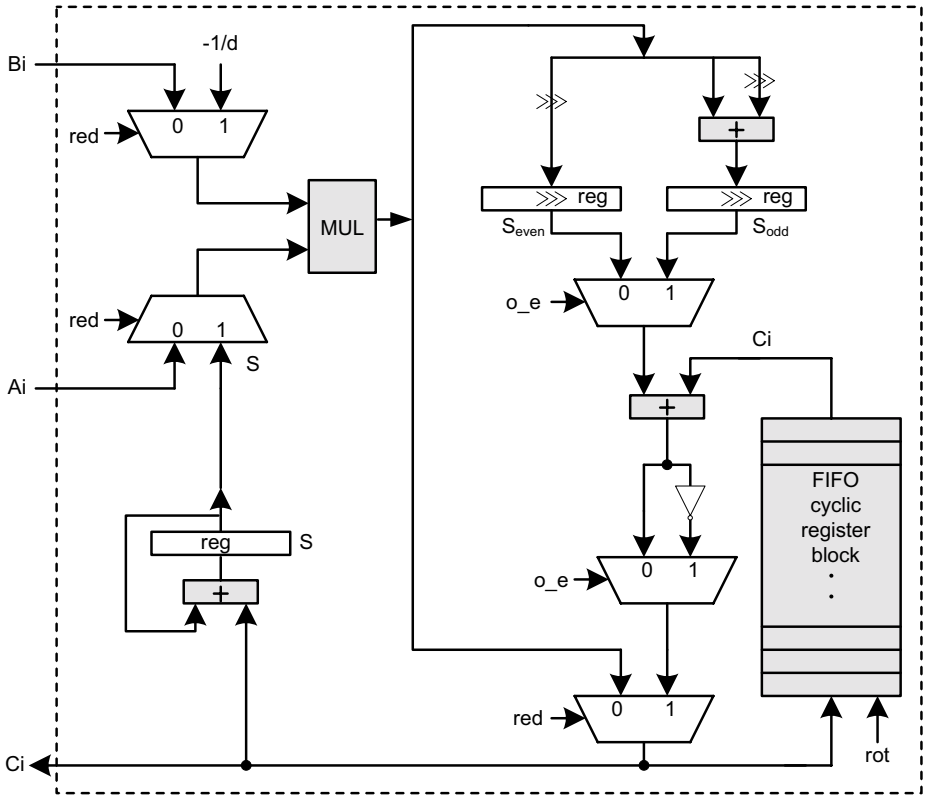


Figure 7.4: DFT Montgomery Multiplication architecture

architecture. The area is optimized by reusing the various components. Also, since all the bus signals are 13-bit wide, signal routing is made extremely easy in the design.

7.4.3 Point Arithmetic

The overall architecture of the ECC processor is shown in Fig. 7.5. The *Arithmetic Unit* consists of the *DFT modular multiplier* and the base field adder which has a negation unit on one of its inputs for doing subtraction. All the necessary point variables are stored in the *Memory* component. We use FIFO registers here, because the DFT multiplication and addition(subtraction) operate only on 13-bits of the data on each clock cycle. This enables our processor to use 13-bit wide buses throughout the design. To avoid losing the contents of the memory when being read out, they are looped back in the FIFO if new values are not being written in.

The *Control Unit* is the most important component which performs the point arith-

metic by writing the required variables onto the A and B busses, performing the required operations on them and storing the result back to the proper memory register. The *Control Unit* is also responsible for interacting with the external world by reading in inputs and writing out the results. The instruction set of the *Control Unit* is given in Table 7.4.

The ECC point arithmetic is performed using mixed Jacobian-affine coordinates[34] to save area on inversion. Here, we assume the elliptic curve is of the form $y^2 = x^3-3x+b$. We use the binary NAF method (Algorithm 2.12) with mixed co-ordinates to perform the point multiplication. The point arithmetic is performed such that the least amount of temporary storage is required. Since, the point multiplication algorithm allows overwriting of the inputs while performing the point doubling and addition, we require only three extra temporary memory locations. The command sequence for the point doubling and addition issued by the *Control Unit* are given in Table 7.5 and Table 7.6, respectively, with temporary variables denoted as T_1, T_2 and T_3. The point doubling is performed in Jacobian coordinates and requires 8 DFT multiplications and 12 sequence additions (or subtractions). Point addition is performed in mixed Jacobian and affine coordinates, and requires 11 DFT multiplications and 7 additions. Point subtraction (for NAF method) is easily implemented by flipping the bits of y_2 in Step 6 of the point addition. The total size of the *Memory* is therefore eight sequence FIFO registers for $(X_1, Y_1, Z_1, x_2, y_2, T_1, T_2$ and $T_3)$ or $26 * 13 * 8 = 2704$ bits.

The inversion required for final conversion from projective to affine coordinates is performed using Fermat's Little Theorem. The conversion from time to frequency domain and vice-versa is done by simple rotations which is done using the *FIFO cyclic register block* inside the DFT multiplier.

7.5 Performance Analysis

We present here the implementation results for the ECC processor design for three different finite fields: $\mathbb{F}_{(2^{13}-1)^{13}}$, $\mathbb{F}_{(2^{17}-1)^{17}}$ and $\mathbb{F}_{(2^{19}-1)^{19}}$. For our performance measurements we synthesized for a custom ASIC design using AMI Semiconductor $0.35\mu m$ CMOS technology using the Synopsys Design Compiler tools. Timing measurements were done with Modelsim simulator against test vectors generated with Maple.

Table 7.2 shows the area requirements for the ECC processor in terms of the equivalent number of NAND gates. The areas required for each of the three main components of the processor are also shown individually.

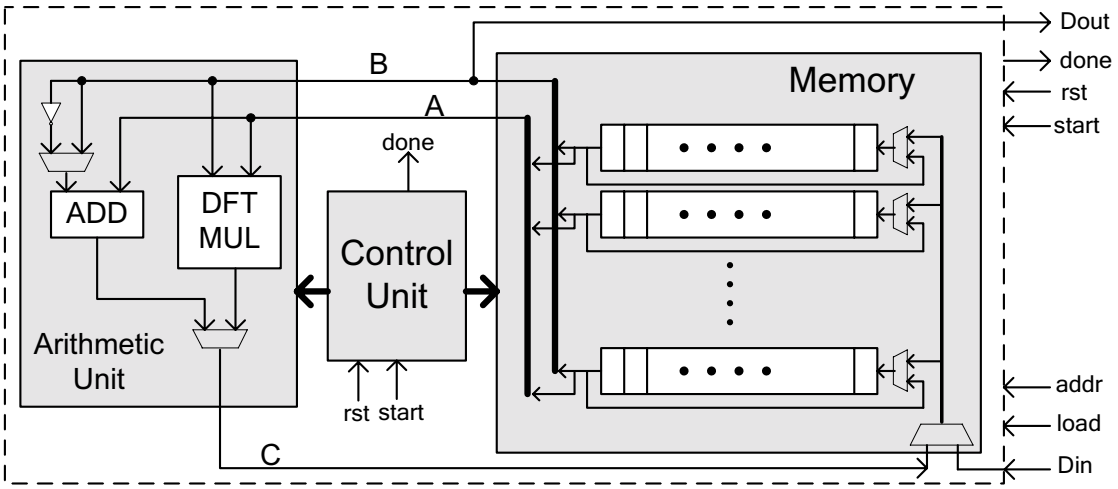


Figure 7.5: Top Level ECC Processor Architecture

Table 7.2: Equivalent Gate Count Area of ECC Processor

Field	Arithmetic Unit	Control Unit	Memory	Total Area
$\mathbb{F}_{(2^{13}-1)^{13}}$	5537.06	351.26	18768.66	24754.62
$\mathbb{F}_{(2^{17}-1)^{17}}$	6978.95	362.56	31794.52	39243.00
$\mathbb{F}_{(2^{19}-1)^{19}}$	10898.82	362.89	39586.72	50959.02

Table 7.3 presents the number of clock cycles required for the DFT multiplication and ECC point arithmetic. It also shows the maximum frequency of the processor and total time required to perform a point multiplication.

We attempt here to compare our results to VLSI implementations of ECC which are openly available in literature and oriented towards light-weight implementations. To the best of our knowledge, the closest ASIC implementation of an OEF is proposed in [14] which uses a CalmRISC 8-bit micro-controller with MAC2424 math co-processor to implement ECC point multiplication over $\mathbb{F}_{(2^{16}-165)^{10}}$. It computes one point multiplication in 122 ms at 20 Mhz which is comparatively much slower than our results for the 169-bit OEF. The paper does not mention any area figures for the device. A low power elliptic curve digital signature chip over the binary field $\mathbb{F}_{2^{178}}$ is presented

Table 7.3: Timing measurements (in clock cycles) for the ECC Processor

Field	**DFT multi.**	**Point Double**	**Point Addition**	**Max. Frequency** (Mhz)	**Point Multiplication**
$\mathbb{F}_{(2^{13}-1)^{13}}$	354	3180	4097	238.7	3.47 ms
$\mathbb{F}_{(2^{17}-1)^{17}}$	598	5228	6837	226.8	10.33 ms
$\mathbb{F}_{(2^{19}-1)^{19}}$	744	6444	8471	221.7	16.34 ms

in [71]. It uses $0.5\mu m$ CMOS technology and the point multiplication occupies 112K gates which is almost double the size of our 361-bit OEF implementation. The performance is however much better requiring just $4.4ms$ to compute a signature at 20Mhz. An implementation of ECC over the prime field $\mathbb{F}_{(2^{167}+1)/3}$ is shown in [64]. It occupies an area of around 30K gates and performs a point multiplication in 6.3ms at 100Mhz. A very small implementation of ECC over 192, 224 and 256-bit prime fields is mentioned in [79] using $0.35\mu m$ CMOS technology. The area requirements are between 23K gates and 31K gates. The performance is around 10.2 ms at 66 Mhz for the 192-bit ECC. Based on these implementations, we can easily confirm that the proposed implementation is area efficient without compromising on speed.

7.6 Summary

We have proposed a novel hardware architecture for DFT modular multiplication which is area efficient. We have also presented an ECC processor architecture to perform point multiplication in the frequency domain using this multiplier. We have synthesized our architecture for custom VLSI CMOS technology to estimate the area and the time performance, and shown that the proposed ECC processor is time/area efficient and useful in constrained environments such as sensor networks.

Table 7.4: Controller Commands of ECC Processor

Command	**Action**
LOAD $[addr_A]$	Load data into the register $[A]$
READ $[addr_A]$	Read data out of the register $[A]$
DFT_MULT $[addr_A]$ $[addr_B]$ $[addr_C]$	Perform DFT multiplication on sequences in $[A]$, $[B]$ and stores the result in $[C]$
ADD_SEQ $[addr_A]$ $[addr_B]$ $[addr_C]$	Perform base field addition on sequences in $[A]$, $[B]$ and stores the result in $[C]$
SUB_SEQ $[addr_A]$ $[addr_B]$ $[addr_C]$	Perform base field subtraction on sequences in $[A]$, $[B]$ and stores the result in $[C]$
MOVE_SEQ $[addr_A]$ $[addr_B]$	Move data from register $[A]$ to $[B]$
DFT_SEQ $[addr_A]$	Convert data sequence at $[A]$ to frequency domain
TIME_SEQ $[addr_A]$	Convert data sequence at $[A]$ to time domain

Table 7.5: Point Doubling Instruction Sequence for ECC Processor.
Input:P=(X_1, Y_1, Z_1) in Jacobian.
Output:2P=(X_1, Y_1, Z_1) in Jacobian.

Command	input $< addr_A >$	input $< addr_B >$	output $< addr_C >$
DFT_MUL	Z_1	Z_1	T_1
SUB_SEQ	X_1	T_1	T_2
ADD_SEQ	X_1	T_1	T_3
DFT_MUL	T_2	T_3	T_1
ADD_SEQ	T_1	T_1	T_2
ADD_SEQ	T_1	T_2	T_1
DFT_MUL	Y_1	Z_1	Z_1
ADD_SEQ	Z_1	Z_1	$\mathbf{Z_1}$
DFT_MUL	Y_1	Y_1	Y_1
ADD_SEQ	Y_1	Y_1	T_2
ADD_SEQ	T_2	T_2	T_2
DFT_MUL	Y_1	T_2	T_3
ADD_SEQ	T_3	T_3	T_3
DFT_MUL	X_1	T_2	T_2
DFT_MUL	T_1	T_1	X_1
SUB_SEQ	T_1	T_2	X_1
SUB_SEQ	T_1	T_2	$\mathbf{X_1}$
SUB_SEQ	T_2	X_1	Y_1
DFT_MUL	Y_1	T_1	Y_1
SUB_SEQ	Y_1	T_3	$\mathbf{Y_1}$

Table 7.6: Point Addition Instruction Sequence for ECC Processor
Input:P=(X_1, Y_1, Z_1) in Jacobian, Q=(x_2, y_2) in affine
Output:P+Q=(X_1, Y_1, Z_1) in Jacobian.

Command	input $< addr_A >$	input $< addr_B >$	output $< addr_C >$
DFT_MUL	Z_1	Z_1	T_1
DFT_MUL	x_2	T_1	T_2
SUB_SEQ	T_2	X_1	T_2
DFT_MUL	Z_1	T_1	T_1
DFT_MUL	Z_1	T_2	$\mathbf{Z_1}$
DFT_MUL	y_2	T_1	T_1
SUB_SEQ	T_1	Y_1	T_1
DFT_MUL	T_2	T_2	T_3
DFT_MUL	T_2	T_3	T_2
DFT_MUL	X_1	T_3	T_3
DFT_MUL	T_1	T_1	X_1
SUB_SEQ	X_1	T_2	X_1
SUB_SEQ	X_1	T_3	X_1
SUB_SEQ	X_1	T_3	$\mathbf{X_1}$
DFT_MUL	Y_1	T_2	Y_1
SUB_SEQ	T_3	X_1	T_3
DFT_MUL	T_1	T_3	T_3
SUB_SEQ	T_3	Y_1	$\mathbf{Y_1}$

Chapter 8

Hardware Design: Tiny ECC Processor over $\mathbb{F}_{2^m}$

We present here results of the work which was in part published in [48].

8.1 Motivation and Outline

A number of hardware implementations for standardized *Elliptic Curve Cryptography* have been suggested in literature, but very few of them are aimed for low-end devices. Most implementations focus on speed and are mostly only suitable for server end applications due to their huge area requirements. A survey of different ECC implementations can be found in [10]. An ISE based implementation as shown in Chapter 4 provides a simple solution if a processor is already available on the device. However, there is an equally important need for a stand-alone ECC engines in small constrained devices used for different applications, like sensor networks and RF-ID tags. This is normally dictated by the needs for better performance required by a communication protocol or energy constraints (as a stand-alone engine can be selectively switched off when not in use).

The different ECC processor implementations that have been suggested for such low-end applications normally use non-standardized curves and hence are not acceptable for commercial applications. A few of these implementations have been previously discussed in Section 7.5. The work in [79] presents an ECC implementation aimed for low-area for both $\mathbb{F}_{2^m}$ and $\mathbb{F}_p$ curves. The implementations use field sizes in the range of 191 to 256-bits for certain standardized curves.

In this work, we try to find the limits of a low-area stand-alone public-key processor for standardized ECC curves. Therefore, we tradeoff flexibility in a design for a spe-

cific standardized binary field curve which is quite reasonable for constrained devices. We also note from previous implementations ([79] and Chapter 7), that the memory requirements for storage of points and temporary variables can contribute substantially (more than 50%) to the overall size of the ECC processor. Hence, we aim for algorithms that require less area even if it leads to a small computational drawback.

The chapter is organized as follows: In Section 8.2, we give the proper choice and tweaks of the different algorithms that allow to reduce area without drastically affecting the performance. Section 8.3 presents the implementation design for the different arithmetic units, memory unit and the overall processor design. Finally, we analyze the area and performance in Section 8.4.

8.2 Mathematical Background

Characteristic two fields $\mathbb{F}_{2^m}$ are often chosen for hardware realizations [10] as they are well suited for hardware implementation due to their "carry-free" arithmetic. This not only simplifies the architecture but reduces the area due to the lack of carry arithmetic. For the implementation of our stand-alone ECC processor, we use standardized binary fields that provide short-term security, and also fields which are required for high security applications. The four different field sizes chosen for our implementation range from 113 to 193-bits, and are recommended in the standards SECG [1] and NIST [60]) (the different recommended curves and the reduction polynomials are shown in Table 8.1). For some of the constrained devices, short-term keys in 113-bit fields can provide the adequate security required for the application and therefore are a good option when area is extremely constrained.

Table 8.1: Standards recommended field sizes for $\mathbb{F}_{2^m}$ and reduction polynomial

Standard	**Field Size** (m)	**Reduction polynomial**
SECG	113	$F_{113}(x) = x^{113} + x^9 + 1$
SECG	131	$F_{131}(x) = x^{131} + x^8 + x^3 + x^2 + 1$
NIST, SECG	163	$F_{163}(x) = x^{163} + x^7 + x^6 + x^3 + 1$
SECG	193	$F_{193}(x) = x^{193} + x^{15} + 1$

A second reason for the use of the binary fields, is the simplified squaring structure, which is a central idea used in the algorithms chosen for the processor design.

8.2.1 Squaring

As described in Section 2.4.1, squarings in $\mathbb{F}_{2^m}$ could be done in two steps, first an expansion with interleaved 0's (Eq. 2.11), and then reducing the double-sized result with the reduction polynomial using the equivalence in Eq. 2.7. However, in hardware these two steps can be combined if the reduction polynomial has a small number of non-zero co-efficients (which is the case with the trinomial and pentanomial reduction polynomials as in Table 8.1). Hence, the squaring can be efficiently implemented to generate the result in *one single clock cycle* without huge area requirements. The implementation and the area costs are discussed in detail in the implementation section.

8.2.2 Inversion

It is well known that performing point arithmetic in affine co-ordinates requires lesser number of temporary variables. This is a very good argument to help reduce the memory requirements. However, the disadvantage is that the point operations in the affine co-ordinates requires an inversion operation (see Section 2.4). Dedicated inversion units using binary Euclidean methods are themselves costly to implement and require extra storage variables. The other more simpler method to perform inversion is using the *Fermat's Little Theorem* [12]:

$$A^{-1} \equiv A^{2^m-2} = (A^{2^{m-1}-1})^2 \bmod F(x) \quad \text{for } A \in \mathbb{F}_{2^m}. \tag{8.1}$$

Since $2^m-2 = 2^1+2^2+\cdots+2^{m-1}$, a straightforward way of performing this exponentiation would be a binary square-and-multiply (similar to double-and-add Algorithm 2.10) as $A^{-1} = A^{2^1} \cdot A^{2^2} \cdots A^{2^{m-1}}$, requiring a total of $(m-2)$ multiplications and $(m-1)$ squarings. This is an extremely costly due to the large number of multiplications and can considerably slow down an implementation of an ECC point multiplication. Therefore, projective co-ordinates are chosen even for low-area implementations [79].

However, Itoh and Tsujii proposed in [39], a construction of an addition chain such that the exponentiation could be performed in $O(log_2 m)$ multiplications. Though the algorithm was proposed for optimal normal basis implementations where squarings are almost for free (cyclic rotations), the area requirements for the squaring structure in

our implementation is within bounds. However, the timing efficiency of 1-clock cycle is same as in the normal basis.

We first present here the addition chain construction and discuss the arithmetic costs for the different fields that we use. Representing $m-1$ in binary format, we can write

$$m-1 = 2^{q-1} + m_{q-2}2^{q-2} + \cdots + m_1 2 + m_0$$

where $m_i \in 0, 1$ and $q = \lfloor log_2(m-1) \rfloor + 1$, the bit-length of $m-1$. We can then represent $m-1$ as a bit vector: $[1m_{q-2} \cdots m_1 m_0]_2$.
The Itoh-Tsujii method is based on the idea that we can represent

$$\begin{aligned} 2^{m-1}-1 &= 2^{[1m_{q-2}\cdots m_1 m_0]_2} - 1 \\ &= 2^{m_0}(2^{2\cdot[1m_{q-2}\cdots m_1]_2} - 1) + 2^{m_0} - 1 \\ &= 2^{m_0}(2^{[1m_{q-2}\cdots m_1]_2} - 1) \cdot (2^{[1m_{q-2}\cdots m_1]_2} + 1) + m_0 \end{aligned} \tag{8.2}$$

Thus this process can be recursively iterated. If we define

$$T_i = (2^{[1m_{q-2}\cdots m_i]_2} - 1),$$

the recursive equation can be represented as

$$T_i = 2^{m_i} T_{i+1} \cdot (2^{[1m_{q-2}\cdots m_{i+1}]_2} + 1) + m_i \quad \text{for} \;\; 0 \leq i \leq (q-2)$$

and $T_{q-1} = 1$.
The exponentiation $A^{2^{m-1}-1} = A^{T_0}$ can then be shown as an iterative operation:

$$\begin{aligned} A^{T_i} &= A^{2^{m_i} T_{i+1} \cdot (2^{[1m_{q-2}\cdots m_{i+1}]_2} + 1) + m_i} \\ &= \{(A^{T_{i+1}})^{2^{[1m_{q-2}\cdots m_{i+1}]_2}} \cdot (A^{T_{i+1}})\}^{2^{m_i}} \cdot (A)^{m_i} \quad \text{for} \;\; 0 \leq i \leq (q-2) \end{aligned} \tag{8.3}$$

Thus each iterative step requires $[1m_{q-2} \cdots m_{i+1}]_2$ squaring + 1 multiplication, and if $m_i = 1$, an additional squaring and multiplication. It can be easily showed that the inverse A^{-1} can then be obtained in $(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1)$ multiplications and $(m-1)$ squaring using this addition chain, where $H_w(.)$ denotes the Hamming weight of the binary representation.

Algorithm 8.1, shows the steps involved for calculation of the inverse for the field size $163 = [10100011]_2$. As shown, an inverse in $\mathbb{F}_{2^{163}}$ requires 9 field multiplications and 162 squaring, and one extra variable (denoted as T here) for the temporary storage (which is blocked only during the inversion process and hence can be used later as a temporary variable in the point multiplication algorithm). As we already mentioned, a squaring can be computed in a single clock cycle, and hence the overall cost for

inverse is approximately 10 multiplications (assuming multiplication takes 163 clock cycles). Hence, we take the different approach of using the affine co-ordinates for our implementation of the ECC processor. Similar addition chains can be achieved for the other field sizes. We give here only the costs in terms of the field multiplications and squarings for each in the Table 8.2

Table 8.2: $\mathbb{F}_{2^m}$ inversion cost using Itoh-Tsuji method

Field size (m)	**Cost**
113	8 $\mathbb{F}_{2^{113}}$ **M** + 112 $\mathbb{F}_{2^{113}}$ **S**
131	8 $\mathbb{F}_{2^{131}}$ **M** + 130 $\mathbb{F}_{2^{131}}$ **S**
163	9 $\mathbb{F}_{2^{163}}$ **M** + 162 $\mathbb{F}_{2^{163}}$ **S**
193	9 $\mathbb{F}_{2^{193}}$ **M** + 192 $\mathbb{F}_{2^{193}}$ **S**

8.2.3 Point multiplication

Montgomery point multiplication is a very efficient algorithm which is used widely because of the computational savings it gives in projective co-ordinates (see Section 2.6.4). It also has the added advantage that it computes only over the x co-ordinates in each iteration and hence requires lesser storage area. It is based on the fact that a running difference $P = (x, y) = P_1 - P_2$ can be used to derive the x co-ordinate of $P_1 + P_2 = (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ as:

$$x_3 = \begin{cases} x + (\frac{x_1}{x_1+x_2})^2 + \frac{x_1}{x_1+x_2} & \text{if } P_1 \neq P_2 \\ x_1^2 + \frac{b}{x_1^2} & \text{if } P_1 = P_2 \end{cases} \tag{8.4}$$

In affine co-ordinates, the Montgomery algorithm as shown in Algorithm 8.2.3, has the disadvantage that it requires two inversions to be computed in each iteration.
The overall cost of the point multiplication using this algorithms is:

$$\sharp INV. = 2\lfloor log_2 k \rfloor + 2, \quad \sharp MUL. = 2\lfloor log_2 k \rfloor + 4,$$
$$\sharp ADD. = 4\lfloor log_2 k \rfloor + 6, \quad \sharp SQR. = 2\lfloor log_2 k \rfloor + 2.$$

We can reduce the inversions required by performing a simultaneous inversion. For $a_1, a_2 \in \mathbb{F}_{2^m}$ and non-zero, we first compute the product $A = a_1 \cdot a_2 \bmod F(\alpha)$ and

Algorithm 8.1 Inversion with Itoh-Tsuji method over $\mathbb{F}_{2^{163}}$

Input: $A \in \mathbb{F}_{2^{163}}$ and irreducible polynomial $F(t)$.

Output: $B \equiv A^{-1} \bmod F(t) = A^{2^m-2}$ where $m = 163$.

1: $B \leftarrow A^2 = A^{(10)_2}$ { 1 SQR }

2: $T \leftarrow B \cdot A = A^{(11)_2}$ { 1 MUL }

3: $B \leftarrow T^{2^2} = A^{(1100)_2}$ { 2 SQR }

4: $T \leftarrow B \cdot T = A^{(1111)_2}$ { 1 MUL }

5: $B \leftarrow T^2 = A^{(11110)_2}$ { 1 SQR }

6: $T \leftarrow B \cdot A = A^{(11111)_2}$ { 1 MUL }

7: $B \leftarrow T^{2^5} = A^{(1111100000)_2}$ { 5 SQR }

8: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{10})_2}$ { 1 MUL }

9: $B \leftarrow T^{2^{10}} = A^{(\underbrace{1\cdots1}_{10}\underbrace{0\cdots0}_{10})_2}$ { 10 SQR }

10: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{20})_2}$ { 1 MUL }

11: $B \leftarrow T^{2^{20}} = A^{(\underbrace{1\cdots1}_{20}\underbrace{0\cdots0}_{20})_2}$ { 20 SQR }

12: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{40})_2}$ { 1 MUL }

13: $B \leftarrow T^{2^{40}} = A^{(\underbrace{1\cdots1}_{40}\underbrace{0\cdots0}_{40})_2}$ { 40 SQR }

14: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{80})_2}$ { 1 MUL }

15: $B \leftarrow T^2 = A^{(\underbrace{1\cdots1}_{80}0)_2}$ { 1 SQR }

16: $T \leftarrow B \cdot A = A^{(\underbrace{1\cdots1}_{81})_2}$ { 1 MUL }

17: $B \leftarrow T^{2^{81}} = A^{(\underbrace{1\cdots1}_{81}\underbrace{0\cdots0}_{81})_2}$ { 81 SQR }

18: $T \leftarrow B \cdot T = A^{(\underbrace{1\cdots1}_{162})_2}$ { 1 MUL }

19: $B \leftarrow T^2 = A^{(\underbrace{1\cdots1}_{162}0)_2}$ { 1 SQR }

20: **Return** B

perform a single inversion $A^{-1} \bmod F(\alpha)$. Then the individual inverses are obtained by the two multiplication $a_1^{-1} = A^{-1} \cdot a_2 \bmod F(\alpha)$ and $a_2^{-1} = A^{-1} \cdot a_1 \bmod F(\alpha)$. Hence, we can trade-off one inversion for three extra multiplications. From Table 8.2,

Algorithm 8.2 Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates [52]

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$
Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$
1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: Set $x_1 \leftarrow x$, $x_2 \leftarrow x^2 + b/x^2$.
3: **for** $i = l - 2$ downto 0 **do**
4: Set $t \leftarrow \frac{x_1}{x_1 + x_2}$.
5: **if** $k_i = 1$ **then**
6: $x_1 \leftarrow x + t^2 + t, \quad x_2 \leftarrow x_2^2 + \frac{b}{x_2^2}$.
7: **else**
8: $x_2 \leftarrow x + t^2 + t, \quad x_1 \leftarrow x_1^2 + \frac{b}{x_1^2}$.
9: **end if**
10: **end for**
11: $r_1 \leftarrow x_1 + x$, $r_2 \leftarrow x_2 + x$
12: $y_1 \leftarrow r_1(r_1 r_2 + x^2 + y)/x + y$
13: Return $(Q = (x_1, y_1)$

we know that inversions for our implementation are more costly than 3 multiplications and therefore a simultaneous inversion always gives a better performance.

There is however, the cost for one extra memory location to temporarily save the product A during the computation of the inverse (apart from the temporary location T). We use two different options here: a) we allocate an extra memory location (denoted as R here) to store the temporary variable A, and b) the product A is computed each time it is required during the inverse computation.

Based on the discussion on the inverse computation and as seen in Algorithm 8.1, the value of A is required at $H(m-1)$ different steps in the inverse operation and hence has to be recomputed $H(m-1)-1$ times (since the computation at the beginning would have to be done anyways). This is quite low and if area is the main constraint, replacing A with extra multiplications would not drastically affect performance. Therefore, the steps in the computation of the inverse in Algorithm 8.1:

$T \leftarrow B \cdot A$;

are replaced with the computational sequence:

$T \leftarrow a_1 \cdot a_2$; $\{T = A\}$
$T \leftarrow B \cdot T$;

As we mentioned, simultaneous inversion requires three extra multiplications to trade-off one inversion. However, for the Montgomery point multiplication algorithm, we can

reduce this to just two multiplications based on the observation that the x co-ordinate of $P_1 - P_2$ (as in Eq. 8.4) can as well be replaced with x co-ordinates of $P_2 - P_1$ as shown here (based on the $\mathbb{F}_{2^m}$ group operation defined in Section 2.4):

$$x_3 = \begin{cases} x + (\frac{x_2}{x_1+x_2})^2 + \frac{x_2}{x_1+x_2} & \text{if } P \neq Q \\ x_1^2 + \frac{b}{x_1^2} & \text{if } P = Q \end{cases} \tag{8.5}$$

The modified Montgomery algorithm is as shown in Algorithm 8.2.3. We now require one inversion and four multiplications in each iteration. The total cost of the $\mathbb{F}_{2^m}$ point multiplication using this algorithms is:

$$\sharp INV. = \lfloor log_2 k \rfloor + 2, \quad \sharp MUL. = 4\lfloor log_2 k \rfloor + 4,$$
$$\sharp ADD. = 4\lfloor log_2 k \rfloor + 6, \quad \sharp SQR. = 3\lfloor log_2 k \rfloor + 2.$$

The algorithm also allows us to compute each iteration without the need for any extra temporary memory locations (apart from the memory location T for inversion, and based on the implementation option the memory location R)

8.3 Implementation Aspects

Based on the mathematical analysis, the main units that are required for the ECC processor are the adder, multiplier and squaring units in $\mathbb{F}_{2^m}$.

8.3.1 $\mathbb{F}_{2^m}$ Adder Unit

Addition is a simple bit wise XOR operation implemented using XOR gates. Therefore, a $\mathbb{F}_{2^m}$ addition is implemented in our design using m XOR gates with the output latency of 1 clock cycle.

8.3.2 $\mathbb{F}_{2^m}$ Multiplier Unit

Multipliers are normally the next biggest component in an ECC processor and therefore the appropriate multiplier design needs to be chosen based on the implementation goals (speed or area). When implementing for constrained devices, which requires extreme savings in area, bit-serial multipliers are the most efficient that reduce area and maintain good performance. The Least Significant Bit-serial (LSB) architecture,

Algorithm 8.3 Modified Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$

Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$

1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: Set $x_1 \leftarrow x$, $x_2 \leftarrow x^2 + b/x^2$.
3: **for** $i = l - 2$ downto 0 **do**
4: Set $r_0 \leftarrow x_1 + x_2$.
5: **if** $k_i = 1$ **then**
6: $R = \frac{1}{(x_1+x_2)\cdot x_2}$
7: $x_1 \leftarrow x + (x_2^2 \cdot R)^2 + (x_2^2 \cdot R), \quad x_2 \leftarrow x_2^2 + b \cdot (r_0 \cdot R)^2$.
8: **else**
9: $R = \frac{1}{(x_1+x_2)\cdot x_1}$
10: $x_2 \leftarrow x + (x_1^2 \cdot R)^2 + (x_1^2 \cdot R), \quad x_1 \leftarrow x_1^2 + b \cdot (r_0 \cdot R)^2$.
11: **end if**
12: **end for**
13: $r_1 \leftarrow x_1 + x$, $r_2 \leftarrow x_2 + x$
14: $y_1 \leftarrow r_1(r_1 r_2 + x^2 + y)/x + y$
15: Return $(Q = (x_1, y_1)$

as shown in Fig 4.3, is not suitable for the present implementation since the reduction is performed on the operand A, and therefore A has to moved and stored within the multiplier to perform this reduction. This requires extra area, equivalent to the storage of one operand memory register. We get rid of this extra area cost by using the Most-Significant Bit-serial (MSB) multiplier. The structure of the 163-bit MSB multiplier is as shown Fig 8.1.

Here, the operand A can be enabled onto the data-bus A of the multiplier, directly from the memory register location. The individual bits of b_i can be sent from a memory location by implementing the memory registers as a cyclic shift-register (with the output at the most-significant bit). The value of the operand register remains unchanged after the completion of the multiplication as it makes one complete rotation.

The reduction within the multiplier is now performed on the accumulating result c_i, as in step 4 in Algorithm 2.6. The taps that are feeded back to c_i, are based on the reduction polynomial. In Fig 8.1, which shows the implementation for $F_{163}(x)$ reduction polynomial, the taps XOR the result of c_{162} to c_7, c_6 c_3 and c_0.

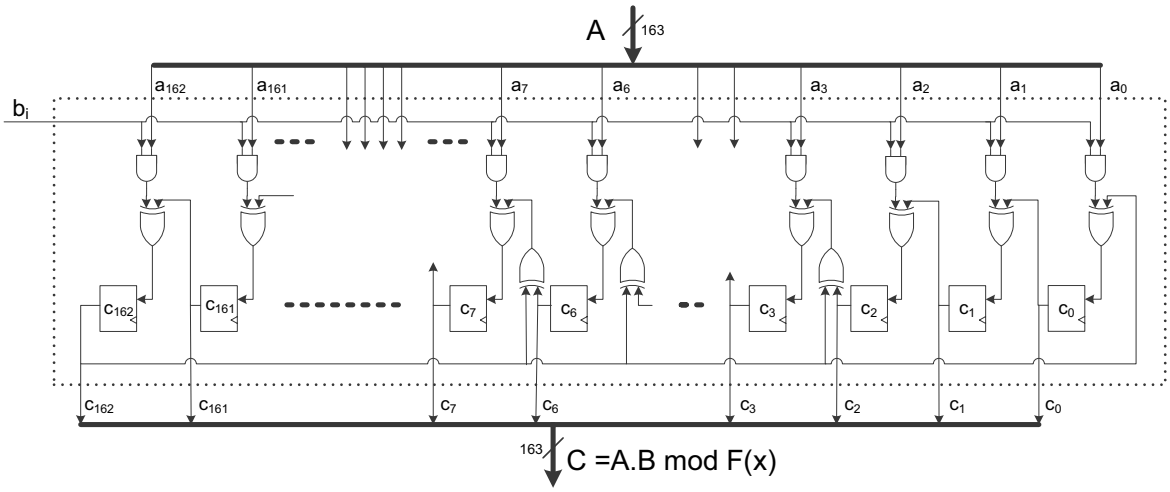


Figure 8.1: $\mathbb{F}_{2^{163}}$ Most Significant Bit-serial (MSB) Multiplier circuit

The complexity of the multiplier is n AND + $(n + t - 1)$ XOR gates and n FF where $t = 3$ for a trinomial reduction polynomial ($F_{113}(x)$ and $F_{193}(x)$) and $t = 5$ for a pentanomial reduction polynomial ($F_{131}(x)$ and $F_{163}(x)$). The latency for the multiplier output is n clock cycles. The maximum critical path is $2\Delta_{XOR}$ (independent of n) where, Δ_{XOR} represents the delay in an XOR gate.

8.3.3 $\mathbb{F}_{2^m}$ Squarer Unit

As mentioned previously, we can implement a very fast squarer with a latency of a single clock cycle. Squaring involves first the expansion by interleaving with 0's, which in hardware is just an interleaving of 0 bit valued lines on to the bus to expand it to $2n$ bits. The reduction of this polynomial is inexpensive, first, due to the fact that reduction polynomial used is a trinomial or pentanomial, and secondly, the polynomial being reduced is sparse with no reduction required for $\lfloor n/2 \rfloor$ of the higher order bits (since they have been set to 0's). The squarer is implemented as a hard wired XOR circuit as shown in Fig. 8.2. The XOR requirements and the maximum critical path (assuming an XOR tree implementation) for the four reduction polynomials used are given in the Table 8.3.

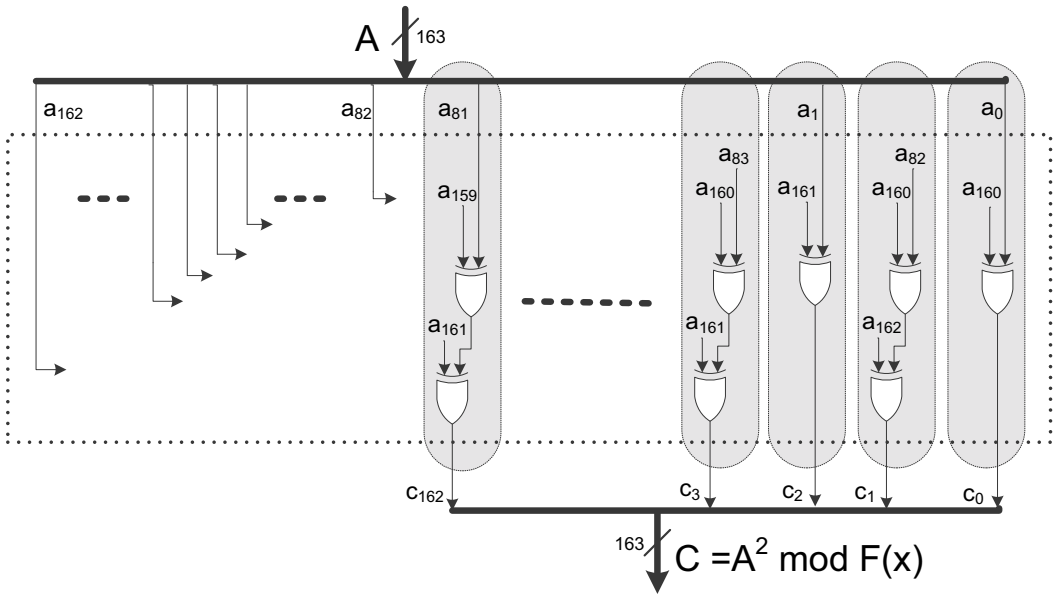


Figure 8.2: $\mathbb{F}_{2^{163}}$ squaring circuit

8.3.4 ECC Processor design

The three units: $\mathbb{F}_{2^m}$ addition (ADD), $\mathbb{F}_{2^m}$ multiplication (MUL), and $\mathbb{F}_{2^m}$ squaring (SQR) are closely interconnected inside a single *Arithmetic Unit* (as shown in Fig. 8.3) sharing the common input data-bus A. The appropriate result is selected at the output data-bus C by the *Controller* signal C_{sel}. The adder needs an additional data-bus B for the second operand and the multiplier requires a single bit b_i signal for the multiplicand. The operands are stored in the *Memory* as registers (some of them as cyclic registers) with the output being selected for A, B and b_i using multiplexors with control signals (A_{sel}, B_{sel} and b_{i_sel}) from the *Controller*. All the operand register are connected in parallel to the data-bus C, with the appropriate register being loaded based on the *Controller* load signal C_{ld_reg}.

Inversion is done as mentioned using the Itoh-Tsuji method and thus requires no additional hardware apart from the multiplier and squarer unit with some additional control circuitry to enable the proper variables to the appropriate unit. We define a single INV instruction which performs the required sequence of steps. Since the inversion can be computed with $(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1)$ multiplications and $(m-1)$ squaring, the latency of the inversion in clock cycles is:
$(\lfloor log_2(m-1) \rfloor + H_w(m-1) - 1) \cdot m + (m-1) = (\lfloor log_2(m-1) \rfloor + H_w(m-1)) \cdot m - 1$

Table 8.3: $\mathbb{F}_{2^m}$ Squaring unit requirements

Reduction Polynomial	**XOR gates**	**Critical Path**
$x^{113} + x^9 + 1$	56 XOR	2 Δ_{XOR}
$x^{131} + x^8 + x^3 + x^2 + 1$	205 XOR	3 Δ_{XOR}
$x^{163} + x^7 + x^6 + x^3 + 1$	246 XOR	3 Δ_{XOR}
$x^{193} + x^15 + 1$	96 XOR	2 Δ_{XOR}

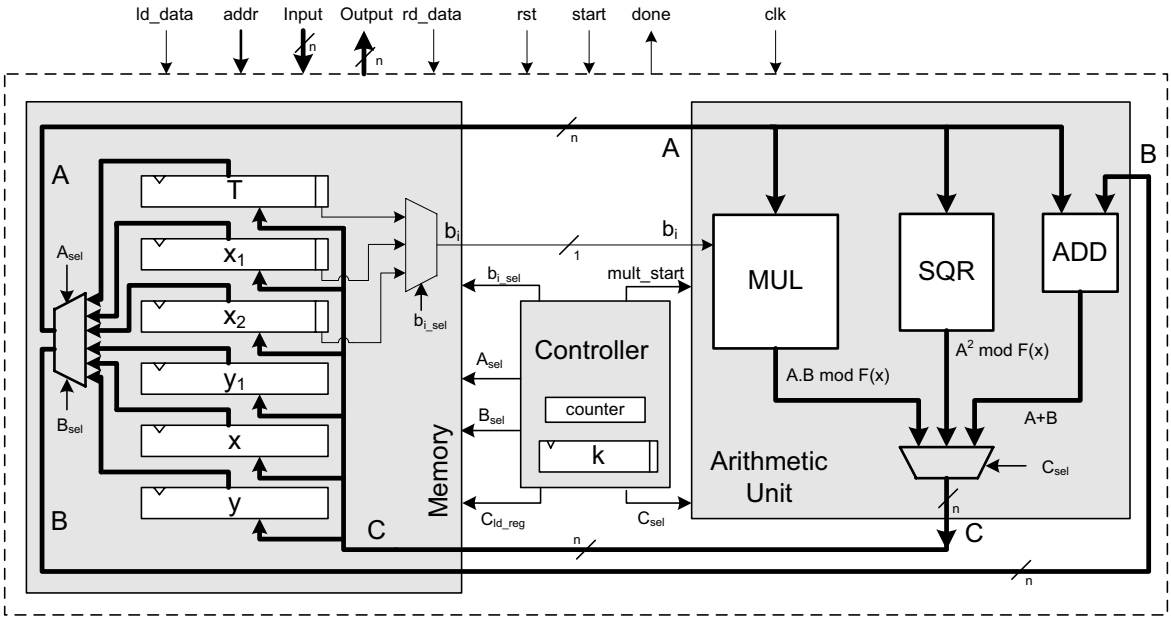


Figure 8.3: Area optimized $\mathbb{F}_{2^n}$ ECC processor

The instruction set of the processor is as shown in Table 8.7. The *Controller* executes the required instructions by enabling the appropriate control signals to the *Arithmetic Unit* and *Memory*. The scalar k is stored within the *Controller* as a shift register. The input operands x,y, and k are loaded externally on an n bit data-bus using the *addr* signal. The final results x_1 and y_1 is similarly read out using the *addr* signal.

The next important aspect of the design was to find the optimum sequence of computation steps such that the least number of temporary memory is required. Another

requirement during this optimization process, was to make sure that not all memory variables be connected to the data-bus B and b_i signal. This reduces the area requirement for the selection logic that is implemented with multiplexors and additionally, the fact that memory variables which are implemented without a cyclic shift requires much lesser area.

We use the modified Algorithm 8.2.3 to construct the sequence of instructions as shown in Algorithm 8.5. As mentioned before we can get rid of the extra register R by performing additional multiplications and hence Fig. 8.3 is shown without this register. The processor requires only 6 memory locations, the inputs x, y , the outputs x_1, x_2, and registers x_2 and T. No other extra temporary memory registers are needed during the computation. Only the registers x_1, x_2 and T are connected to the b_i signal and hence the others are implemented as simple registers with no shift operation. Similarly, only the register x_1 and x_2 are connected to the data-bus B. Data-bus A is connected to all the memory locations.

8.4 Performance Analysis

Based on the implementation described in the last section, we build two ECC processors: a) with the extra register R (not a cyclic shift register) and hence fewer multiplications and b) without the extra register R but smaller area. The implementations were synthesized for a custom ASIC design using AMI Semiconductor $0.35\mu m$ CMOS technology using the Synopsys Design Compiler tools. Timing measurements were done with Modelsim simulator against test vectors generated with a Java based model.

The area requirements (in terms of equivalent gate counts) for individual units of both the processor implementations is shown in the Table 8.4. The *Controller* area differs only slightly between the implementations and we give here only for the case without the register R. The area requirements for the *Memory* is given in Table 8.5 (without the extra R register) and Table 8.6 (with the extra R register). As can be seen, memory requirements are more than 50% of the whole design. The designs have a total area ranging from 10k equivalent gates for 113-bit field for short-term security, to 18k for 193-bit field for high security applications.
The latency of the ECC processor in clock cycles for a single scalar multiplication is shown in Tables 8.5 and 8.6. Since the structure is extremely simple, it can be clocked at very high frequencies, but we present here the absolute timings at 13.56 Mhz which

Table 8.4: $\mathbb{F}_{2^m}$ ECC processor area (in equivalent gates)

Field size	**MULT**	**SQR**	**ADD**	**Controller**
113	1210.58	188.38	226.00	1567.51
131	1402.46	406.00	262.00	1833.23
163	1743.58	502.00	326.00	2335.61
193	2068.38	321.98	386.00	2957.83

is normally the frequency used in RF-ID applications. We also make a comparison of the efficiency between the two ECC processor using the area-time product (normalized to the implementation with register R) and shown in Table 8.5. We see that the ECC processor with the extra register has a better efficiency, and so should be the preferred implementation if a slight area increase is acceptable.

Table 8.5: $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *without* extra register R

Field size	**Memory**	**Total Area**	**Cycles**	**Time** (ms)	**Area-Time**
113	6686.43	10112.85	195159	14.4	1.07
131	7747.17	11969.93	244192	18.0	0.99
163	9632.93	15094.31	430654	31.8	1.06
193	11400.83	17723.12	564919	41.7	1.00

Comparing the results presented here to the work in [79] (the only standardized curve implementation for constrained devices), the clock cycles for our 193-bit size implementation is 19% more than for the 191-bit Montgomery projective co-ordinate algorithm, which was expected because we trade-off performance slightly to save on area. However, the implementation is 4.4 times faster than the affine implementation presented. The area requirements of 18k gates for our 193-bit implementation is much smaller (22%) than the 23k gates for the 191-bit field mentioned in [79] (though the design is slightly more versatile due the dual field multipliers used). However, as mentioned before, not all constrained devices require such high security and can therefore use the smaller key size ECC processor implementations.

Table 8.6: $\mathbb{F}_{2^m}$ ECC processor performance @13.56 Mhz *with* extra register R

Field size	**Memory**	**Total Area**	**Cycles**	**Time** (ms)
113	7439.01	10894.34	169169	12.5
131	8619.63	12883.51	226769	16.8
163	10718.49	16206.67	376864	27.9
193	12686.21	19048.22	527284	38.8

8.5 Summary

We showed here an extremely small area implementation of an ECC processor in the affine co-ordinates. Though affine co-ordinate implementations are not normally favored for constrained devices due to the need for an inverter, we show through the use of an addition chain and fast squarer, they are equally good for use in low-area implementations. Further savings are possible at the point arithmetic level by tweaking the algorithm to save on temporary storage variables. Hence, we show that an ECC processor implementation for four different standardized binary fields ranging from 113 to 193 bits with area requirements ranging between 10k and 18k gates, which makes the design very attractive for enabling ECC in constrained devices. The proposed ECC processor is also secure against timing attacks due to the regular structure of the instructions used independent of the scalar.

Table 8.7: Controller Commands of ECC Processor over $\mathbb{F}_{2^m}$

Command	**Action**
LOAD $[addr_A]$	Load data into the register $[A]$ (in parallel)
READ $[addr_A]$	Read data out of the register $[A]$ (in parallel)
MUL $[addr_A]$, $[addr_B]$, $[addr_C]$	Perform $\mathbb{F}_{2^m}$ multiplication on $[A]$ (loaded in parallel), $[B]$ (loaded in serially) and stores the result in $[C]$ (in parallel)
ADD $[addr_A]$ $[addr_B]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ addition on $[A]$, $[B]$ (both loaded in parallel) and stores the result in $[C]$ (in parallel)
SQR $[addr_A]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ squaring on $[A]$ (loaded in parallel) and stores the result in $[C]$
INV $[addr_A]$ $[addr_C]$	Perform $\mathbb{F}_{2^m}$ inversion on $[A]$ (loaded in parallel) and stores the result in $[C]$ (in parallel)

Algorithm 8.4 Instruction sequence for the Modified Montgomery method for scalar point multiplication in $\mathbb{F}_{2^m}$ in affine co-ordinates

Input: P, k, where $P = (x, y) \in E(\mathbb{F}_{2^m}), k = [k_{l-1} \cdots k_1 k_0]_2 \in \mathbf{Z}^+$ and $\log_2 k < m$
Output: $Q = k \cdot P$, where $Q = (x_1, y_1) \in E(\mathbb{F}_{2^m})$

1: if $k = 0$ or $x = 0$ then Return$Q = (0, 0)$ and stop.
2: $x_1 \leftarrow x$,
3: $y_1 \leftarrow \text{SQR}\ (x)$
4: $x_2 \leftarrow \text{INV}\ (y_1)$.
5: $x_2 \leftarrow \text{MUL}\ (b, x_2)$.
6: $x_2 \leftarrow \text{ADD}\ (y_1, x_2)$.
7: **for** $i = l - 2$ downto 0 **do**
8: **if** $k_i = 1$ **then**
9: $x_1 \leftarrow \text{ADD}\ (x_1, x_2)$
10: $R \leftarrow \text{MUL}\ (x_1, x_2)$
11: $y_1 \leftarrow \text{INV}\ (R)$
12: $x_1 \leftarrow \text{MUL}\ (y_1, x_1)$
13: $x_2 \leftarrow \text{SQR}\ (x_2)$
14: $y_1 \leftarrow \text{MUL}\ (y_1, x_2)$
15: $x_1 \leftarrow \text{SQR}\ (x_1)$
16: $x_1 \leftarrow \text{MUL}\ (b\ , x_1)$
17: $x_2 \leftarrow \text{ADD}\ (x_1, x_2)$
18: $x_1 \leftarrow \text{SQR}\ (y_1)$
19: $x_1 \leftarrow \text{ADD}\ (y_1, x_1)$
20: $x_1 \leftarrow \text{ADD}\ (x, x_1)$
21: **else**
22: $x_2 \leftarrow \text{ADD}\ (x_1, x_2)$
23: $R \leftarrow \text{MUL}\ (x_1, x_2)$
24: $y_1 \leftarrow \text{INV}\ (R)$
25: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
26: $x_1 \leftarrow \text{SQR}\ (x_1)$
27: $y_1 \leftarrow \text{MUL}\ (y_1, x_1)$
28: $x_2 \leftarrow \text{SQR}\ (x_2)$
29: $x_2 \leftarrow \text{MUL}\ (b\ , x_2)$
30: $x_1 \leftarrow \text{ADD}\ (x_1, x_2)$
31: $x_2 \leftarrow \text{SQR}\ (y_1)$
32: $x_2 \leftarrow \text{ADD}\ (y_1, x_2)$
33: $x_2 \leftarrow \text{ADD}\ (x, x_2)$
34: **end if**
35: **end for**
36: $y_1 \leftarrow \text{ADD}\ (x, x_1)$
37: $x_2 \leftarrow \text{ADD}\ (x, x_2)$
38: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
39: $T \leftarrow \text{SQR}\ (x)$
40: $x_2 \leftarrow \text{ADD}\ (T\ , x_2)$
41: $x_2 \leftarrow \text{ADD}\ (y\ , x_2)$
42: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
43: $y_1 \leftarrow \text{INV}\ (x)$
44: $x_2 \leftarrow \text{MUL}\ (y_1, x_2)$
45: $y_1 \leftarrow \text{ADD}\ (y, x_2)$
46: Return $(Q = (x_1, y_1)$

Chapter 9

Discussion

In this chapter, we summarize the findings of the thesis and give some recommendations for future research.

9.1 Conclusions

The main focus of the thesis has been to show the feasibility of *Elliptic Curve Cryptography* (ECC) in constrained devices with limited memory and computational capacity. We were able to demonstrate a practical public-key exchange protocol using ECDH on an 8-bit 8051 processor using special Optimal Extension Fields. The key exchange was possible in an acceptable time, which showed the viability of ECC based protocols even on low-end processors without the need for extra hardware. The work in [31], extends to show that standardized prime field ECC on 8-bit processors are also possible within acceptable timings.

Further improvements in the ECC arithmetic was shown by providing small extensions to the processor, which could drastically improve the performance. We could show performance improvements by a factor of 30 on an 8-bit AVR processor and a factor of 1.8 for a 32-bit MIPS processor by proposing appropriate Instruction Set Extensions (ISE). The work in [9], shows a similar use of hardware/software co-design for implementing an *Hyperelliptic Curve Cryptographic* (HECC) algorithm [44] on an 8051 microprocessor.

For the stand-alone implementations of the ECC processor, we first presented some architectural enhancements for the Least Significant Digit-serial (LSD) multipliers, which are the most commonly used multipliers for an area/time optimized implementation. We showed that the new architectures, the Double Accumulator Multiplier

(DAM) and N-Accumulator Multiplier (NAM) are both faster compared to traditional LSD multipliers. Then we proposed an area/time efficient ECC processor architecture for the OEFs of size 169, 289 and 361 bits, which performs all finite field arithmetic operations in the discrete Fourier domain. A 169-bit curve implementation of the ECC processor using the novel multiplier design required just 24k equivalent gates on a 0.35um CMOS process. Finally we presented a highly area optimized ECC design in ASIC, especially attractive for constrained devices. An area between 10k and 18k gates was required on a 0.35um CMOS process for the different standardized binary curves ranging from $133 - 193$ bits.

Hence, we showed different design options for enabling ECC for constrained devices in the three implementation domains: low-end processor (software), extensions for low-end processors (hardware/software co-design) and stand-alone low area ECC processors (hardware). We found that ECC, with its relatively high computational demands, is barely manageable for constrained processors with the help of proper parametric choices. However, a slight extension in the processor can drastically improve the performance of even standard compliant curves. This opens up a completely new domain for low-end processors specifically suited for public-key applications. With the stand-alone implementations, we were able to show that ECC is manageable for constrained devices which till recently were considered to be impracticable. This allows the use of (and design new) protocols based on public-key algorithms to be used even for such constrained applications.

9.2 Future Research

Implementation of cryptographic systems presents several other requirements and challenges especially for constrained environments, other than the memory and area requirements discussed in this thesis. Most importantly is the power and energy requirement for the public key algorithms. This is especially a challenge in pervasive devices running on their own energy storage and placed in the field for long periods of time without any maintenance or possible physical access. Also, the large number of these devices makes replacing the batteries a highly cumbersome process. On the other hand, RF-ID tag applications derive the required power from the electromagnetic field of the reader to run its applications. Hence, such systems also have to be extremely power efficient. Therefore, real world estimates of the power requirements for cryptographic process are extremely important including systems running PKC on processors with

extensions. The underlying arithmetic algorithms could then be chosen and fine-tuned more efficiently for a low power ECC design.

Unlike traditional systems which cannot be physically accessed by an attacker, pervasive systems have to also consider the physical security as they are placed in insecure surroundings easily accessible for tampering. Therefore, storing the private key securely on such devices are still a big challenge and the usual solutions are too expensive for such low-cost devices.

Even when physically secured, these devices can be passively attacked using side-channel (time and power) methods. Well known side-channel resistant algorithms normally require almost double the execution time, large memory and more hardware resources. Thus these measures are unsuitable for such low-end devices which require highly optimized implementation (in time, memory and power) and therefore is an open problem that needs to be further investigated.

Bibliography

[1] *SEC 2. Standards for Efficient Cryptography Group: Recommended Elliptic Curve Domain Parameters. Version 1.0*, 2000.

[2] *ANSI X9.62: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).* American National Standards Institute, New York, USA, 1999.

[3] *ARM: Architecture Reference Manual.* ARM Limited, ARM Doc No. DDI-0100, H edition, October 2003.

[4] *AT94K Series Field Programmable System Level Integrated Circuit — FPSLIC Datasheet, (rev. H).* Atmel Corp., San Jose, CA, USA, July 2005. Available for download at `http://www.atmel.com/products/FPSLIC`.

[5] 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 1462 of *Lecture Notes in Computer Science*, pages 472–485, Springer-Verlag Inc., Berlin, Germany, August 1998.

[6] 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.

[7] S. Baktır and B. Sunar. Finite Field Polynomial Multiplication in the Frequency Domain with Application to Elliptic Curve Cryptography.

[8] 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 263 of *Lecture Notes in Computer Science*, pages 311–323, Springer-Verlag Inc., Berlin, Germany, 1986.

[9] L. Batina, D. Hwang, A. Hodjat, B. Preneel, and I. Verbauwhede. Hardware/software co-design for hyperelliptic curve cryptography (HECC) on the 8051μP. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems—CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 106–118, Springer-Verlag Inc., Berlin, Germany, August 2005.

[10] L. Batina, S. B. Ors, B. Prenee, and J. Vandewalle. Hardware architectures for public key cryptography. *Integration, the VLSI journal*, 34(6):1–64, 2003.

[11] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society lecture note series*. Cambridge University Press, Cambridge, UK, 1999.

[12] D. M. Burton. *Elementary Number Theory*. Allyn and Bacon, 1980.

[13] *Single Chip Very Low Power RF Transceiver with 8051-Compatible Microcontroller —CC1010, Datasheet (rev. 1.3)*. Chipcon, Texas Instruments, Norway., 2003. Available for download at `http://www.chipcon.com/files/CC1010_Data_Sheet_1_3.pdf`.

[14] J. W. Chung, S. G. Sim, and P. J. Lee. Fast Implementation of Elliptic Curve Defined over $GF(p^m)$ on CalmRISC with MAC2424 Coprocessor. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 57–70, Springer-Verlag Inc., Berlin, Germany, August 2000.

[15] P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[16] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent #5,159,632, US Patent and Trade Office, October 1992.

[17] 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 K. Kim and T. Matsumoto, editors, *Advances in Cryptology—ASIACRYPT '96*, volume 1163 of *Lecture Notes in Computer Science*, pages 65–76, Springer-Verlag Inc., Berlin, Germany, November 1996.

[18] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[19] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Bluemel. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $GF(2^n)$. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 381–399, Springer-Verlag Inc., Berlin, Germany, August 2002.

[20] A.M. Fiskiran and R.B. Lee. *Embedded Cryptographic Hardware: Design and Security*, chapter PAX: A Datapath-Scalable Minimalist Cryptographic Processor for Mobile Environments, pages 19–33. Nova Science, NY, USA, September 2004.

[21] P. Gaudry, F. Hess, and N. P. Smart. Constructive and destructive facets of weil descent on elliptic curves. *Journal of Cryptology*, 15(1):19–46, 2002.

[22] J. R. Goodman. *Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, USA, 2000.

[23] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.

[24] J. Großschädl and G.-A. Kamendje. Architectural enhancements for montgomery multiplication on embedded risc processors. In J. Zhou, M. Yung, and Y. Han, editors, *First International Conference Applied Cryptography and Network Security — ACNS 2003*, volume 2846 of *Lecture Notes in Computer Science*, pages 418–434, Springer-Verlag Inc., Berlin, Germany, December 2003.

[25] J. Großschädl, S. S. Kumar, and C. Paar. Architectural support for arithmetic in optimal extension fields. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors — ASAP'04*, volume asap, pages 111–124, IEEE Computer Society Press, Silver Spring, MD, USA, September 2004.

[26] J. Guajardo, R. Bluemel, U. Krieger, and C. Paar. Efficient Implementation of Elliptic Curve Cryptosystems on the TI MSP430x33x Family of Microcontrollers. In K. Kim, editor, *Practice and Theory in Public Key Cryptography — PKC 2001*, volume 1992 of *Lecture Notes in Computer Science*, pages 365–382, Springer-Verlag Inc., Berlin, Germany, February 2001.

[27] J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, and J. Pelzl. Efficient Hardware Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, 93(1–3):75–118, September 2006.

[28] J. Guajardo, S. S. Kumar, C. Paar, and J. Pelzl. Efficient Software-Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, 93(1–3):3–32, September 2006.

[29] J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In B. Kaliski, Jr., editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 342–356, Springer-Verlag Inc., Berlin, Germany, August 1997.

[30] N. Gura, S. Chang, H. Eberle, G. Sumit, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila. An End-to-End Systems Approach to Elliptic Curve Cryptography. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 351–366, Springer-Verlag Inc., Berlin, Germany, May 2001.

[31] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems—CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132, Springer-Verlag Inc., Berlin, Germany, August 2004.

[32] H. Handschuh and P. Paillier. Smart Card Crypto-Coprocessors for Public-Key Cryptography. In J.-J. Quisquater and B. Schneier, editors, *Smart Card Research and Applications—CARDIS '98*, volume 1820 of *Lecture Notes in Computer Science*, pages 372–379, Springer-Verlag Inc., Berlin, Germany, September 2000.

[33] D. Hankerson, J. López, and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In Ç. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 1–24, Springer-Verlag Inc., Berlin, Germany, August 2000.

[34] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag Inc., Berlin, Germany, 2004.

[35] T. Hasegawa, J. Nakajima, and M. Matsui. A Practical Implementation of Elliptic Curve Cryptosystems over $GF(p)$ on a 16-bit Microcomputer. In H. Imai and Y. Zheng, editors, *Practice and Theory in Public Key Cryptography — PKC'98*,

volume 1431 of *Lecture Notes in Computer Science*, pages 182–194, Springer-Verlag Inc., Berlin, Germany, February 1998.

[36] *IEEE P1363-2000: IEEE Standard Specifications for Public-Key Cryptography*. IEEE Computer Society Press, Silver Spring, MD, USA, 2000.

[37] *The Itanium Processor Family - High Performance on Security Algorithms*. Intel Corporation, April 2003. Available for download at `http://intel.com/cd/ids/developer/asmo-na/eng/dc/itanium/optimization/19909.htm`.

[38] *ISO/IEC 15946: Information Technology — Security Techniques: Cryptographic Techniques based on Elliptic Curves*. International Organization for Standardization, Geneva, Switzerland, 2002.

[39] 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.

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

[41] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics — Doklady (English translation)*, 7:595–596, 1963.

[42] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, volume 2. Addison-Wesley, MA, USA, second edition, 1973.

[43] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[44] N. Koblitz. A Family of Jacobians Suitable for Discrete Log Cryptosystems. In S. Goldwasser, editor, *Advances in Cryptology — CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 94 – 99, Springer-Verlag Inc., Berlin, Germany, 1988.

[45] Ç. K. Koç, T. Acar, and B. S. Kaliski. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.

[46] S. S. Kumar, M. Girimondo, A. Weimerskirch, C. Paar, A. Patel, and A. S.Wander. Embedded End-to-End Wireless Security with ECDH Key Exchange. In *Proceedings of the 46th IEEE International Midwest Symposium on Circuits and Systems*

— *MWSCAS 2003*, volume 2, pages 786–789, IEEE Computer Society Press, Silver Spring, MD, USA, December 2003.

[47] S. S. Kumar and C. Paar. Reconfigurable instruction set extension for enabling ECC on an 8-bit processor. In J. Becker, M. Platzner, and S. Vernalde, editors, *14th International Conference Field Programmable Logic and Application — FPL 2004*, volume 3203 of *Lecture Notes in Computer Science*, pages 586–595, Springer-Verlag Inc., Berlin, Germany, August 2004.

[48] S. S. Kumar and C. Paar. Are standards compliant elliptic curve cryptosystems feasible on rfid. In *Workshop on RFID Security 2006*, July 2006.

[49] S. S. Kumar, T. Wollinger, and C. Paar. Optimized digit multipliers for elliptic curve cryptography. In *Cryptographic Advances in Secure Hardware — CRASH*, Leuven, Belgium., September 2005.

[50] S. S. Kumar, T. Wollinger, and C. Paar. Optimum hardware gf(2m) multipliers for curve based cryptography. *IEEE Transactions on Computers*, 55(10):1306–1311, October 2006.

[51] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. In H. Imai and Y. Zheng, editors, *Practice and Theory in Public Key Cryptography—PKC 2000*, volume 1751 of *Lecture Notes in Computer Science*, pages 446–465, Springer-Verlag Inc., Berlin, Germany, January 2000.

[52] J. López and R. Dahab. Fast multiplication on elliptic curves over GF(2 m) without precomputation. In Ç. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327, Springer-Verlag Inc., Berlin, Germany, August 1999.

[53] J. López and R. Dahab. High-Speed Software Multiplication in F_{2^m}. In B. Roy and E. Okamoto, editors, *International Conference in Cryptology in India—INDOCRYPT 2000*, volume 1977 of *Lecture Notes in Computer Science*, pages 203–212, Springer-Verlag Inc., Berlin, Germany, December 2000.

[54] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, FL, USA, 1997.

[55] V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in cryptology — CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426, Springer-Verlag Inc., Berlin, Germany, August 1986.

[56] *MIPS32 4Km^{TM} Processor Core Datasheet*. MIPS Technologies, Inc., September 2001. Available for download at `http://www.mips.com/publications/index.html`.

[57] *MIPS32 ^{TM} Architecture for Programmers*. MIPS Technologies, Inc., March 2001. Available for download at `http://www.mips.com/publications/index.html`.

[58] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[59] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24(6):531–543, 1990.

[60] *FIPS 186-2: Digital Signature Standard (DSS). 186-2*. National Institute for Standards and Technology, Gaithersburg, MD, USA, February 2000. Available for download at `http://csrc.nist.gov/encryption`.

[61] A. M. Odlyzko. Discrete logarithms: the past and the future. *Designs, Codes, and Cryptography*, 19(2-3):129–145, March 2000.

[62] *OpenSSL*. Available for download at `http://www.openssl.org/`.

[63] G. Orlando and C. Paar. A High-Performance reconfigurable Elliptic Curve Processor for $GF(2^m)$. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems—CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 41–56, Springer-Verlag Inc., Berlin, Germany, August 2000.

[64] E. Öztürk, B. Sunar, and E. Savas. Low-power elliptic curve cryptography using scaled modular arithmetic. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems—CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 92–106, Springer-Verlag Inc., Berlin, Germany, August 2004.

[65] B. J. Phillips and N. Burgess. Implementing 1, 024-bit rsa exponentiation on a 32-bit processor core. In *12th IEEE International Conference on Application-Specific*

Systems, Architectures and Processors — ASAP'04, volume asap, pages 127–137, IEEE Computer Society Press, Silver Spring, MD, USA, July 2000.

[66] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25:365–374, 1971.

[67] C. M. Rader. Discrete Convolutions via Mersenne Transforms. *IEEE Transactions on Computers*, C-21:1269–1273, December 1972.

[68] Srivaths Ravi, Anand Raghunathan, Nachiketh R. Potlapally, and Murugan Sankaradass. System design methodologies for a wireless security processing platform. In *Proceedings of the 39th Design Automation Conference —DAC*, pages 777–782, ACM Press, New York, USA, June 2002.

[69] 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.

[70] B. Schneier. *Applied Cryptography*. John Wiley, New York, NY, USA, 2nd edition, 1996.

[71] R. Schroeppel, C. L. Beaver, R. Gonzales, R. Miller, and T. Draelos. A low-power design for an elliptic curve digital signature chip. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 366–380, Springer-Verlag Inc., Berlin, Germany, August 2002.

[72] 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 963 of *Lecture Notes in Computer Science*, pages 43–56, Springer-Verlag Inc., Berlin, Germany, August 1995.

[73] *SDCC - Small Device C Compiler*. Available for download at `http://sdcc.sourceforge.net/`.

[74] *NTL*. Available for download at `http://www.shoup.net/ntl`.

[75] J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate texts in Mathematics*. Springer-Verlag Inc., 1986.

[76] 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.

[77] VLSI Computer Architecture, Arithmetic, and CAD Research Group. *IIT Standard Cells for AMI* $0.5\mu m$ *and TSMC* $0.25\mu m/0.18\mu m$ *(Version 1.6.0)* . Deparment of Electrical Engineering, IIT, Chicago, IL, 2003. Library and documentation available for download from `http://www.ece.iit.edu/~vlsi/scells/home.html`.

[78] A. Weimerskirch, C. Paar, and S. Chang Shantz. Elliptic Curve Cryptography on a Palm OS Device. In V. Varadharajan and Y. Mu, editors, *The 6th Australasian Conference on Information Security and Privacy — ACISP 2001*, volume 2119 of *Lecture Notes in Computer Science*, pages 502–513, Springer-Verlag Inc., Berlin, Germany, 2001.

[79] J. Wolkerstorfer. *Hardware Aspects of Elliptic Curve Cryptography.* PhD thesis, Graz University of Technology, Graz, Austria, 2004.

[80] A. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In J. D.-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Applications—CARDIS 2000*, volume 180 of *IFIP Conference Proceedings*, pages 71–92, Kluwer Academic Publishers, Bristol, UK, September 2000.

Curriculum Vitae

Personal Data

Born on May 13th, 1980 in Kerala, India.
Contact information: kumar@crypto.rub.de

Secondary Education:

1995 – 1997	Little Flower Junior College, Hyderabad, India.

University Education:

1997 – 2001	*Bachelor of Technology* in Electrical Engineering, Indian Institute of Technology, Bombay, India.
2001 – 2002	*Master of Technology* in Electrical Engineering with specialization in Communication and Signal Processing, Indian Institute of Technology, Bombay, India.

Work Experience:

06.2001 – 05.2002	*Teaching Assistant* at the Communication Group, Indian Institute of Technology, Bombay, India.
09.2002 – present	*Researcher* at the Chair for Communication Security, Ruhr University Bochum, Germany.

Publications:

Journal:

- S. S. Kumar, T. Wollinger, and C. Paar, "Optimum Hardware GF(2m) Multipliers for Curve Based Cryptography", IEEE Transactions on Computers, Volume 55, Issue 10, pp. 1306-1311, October 2006.
- J. Guajardo, S. S. Kumar, C. Paar, J. Pelzl, "Efficient Software-Implementation of Finite Fields with Applications to Cryptography", Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications, Volume 93, Numbers 1-3, pp. 3-32, September 2006.
- J. Guajardo, T. Güneysu, S. S. Kumar, C. Paar, J. Pelzl, "Efficient Hardware Implementation of Finite Fields with Applications to Cryptography", Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications, Volume 93, Numbers 1-3, pp. 75-118, September 2006.

Book Chapters:

- S. Kumar, and T. Wollinger, "Fundamentals of Symmetric Cryptography", chapter in Embedded Security in Cars - Securing Current and Future Automotive IT Applications, editors K. Lemke, C. Paar, M. Wolf, Springer-Verlag, 2005.
- T. Wollinger, and S. Kumar, "Fundamentals of Asymmetric Cryptography", chapter in Embedded Security in Cars - Securing Current and Future Automotive IT Applications, editors K. Lemke, C. Paar, M. Wolf, Springer-Verlag, 2005.

Conferences:

- G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger, "Efficient GF(pm) Arithmetic Architectures for Cryptographic Applications", In M. Joye editor, The Cryptographers' Track at the RSA Conference — CT-RSA 2003, volume 2612 of LNCS, pages 158-175, Springer- Verlag Inc., Berlin, Germany, April 2003.
- S. Kumar, M. Girimondo, A.Weimerskirch, C. Paar, A. Patel, and A. S.Wander, "Embedded End-to-End Wireless Security with ECDH Key Exchange", IEEE

International Midwest Symposium on Circuits and Systems — MWSCAS 2003, volume 2, pages 786–789, IEEE Computer Society Press, USA, December 2003.

- S. Kumar and C. Paar, "Reconfigurable instruction set extension for enabling ECC on an 8-bit processor", In J. Becker, M. Platzner, and S. Vernalde, editors, Field Programmable Logic and Application — FPL 2004, volume 3203 of LNCS, pages 586–595, Springer- Verlag Inc., Berlin, Germany, August 2004.
- J. Großschädl, S. S. Kumar, and C. Paar, "Architectural support for arithmetic in optimal extension fields", In 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors — ASAP 2004, pages 111–124, IEEE Computer Society Press, USA, September 2004.
- S. Kumar, K. Lemke, and C. Paar, " Some Thoughts about Implementation Properties of Stream Ciphers", State of the Art of Stream Ciphers Workshop — SASC 2004, Brugge, Belgium, October 2004.
- S. Kumar, T. Wollinger and C. Paar, "Optimized Digit Multipliers for Elliptic Curve Cryptography", Cryptographic Advances in Secure Hardware — CRASH 2004, Leuven, Belgium, September 2005.
- S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, A. Rupp, M. Schimmler, "How to Break DES for Euro 8,980", Special-purpose Hardware for Attacking Cryptographic Systems — SHARCS 2005, Cologne, Germany, April 2006.
- S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, M. Schimmler, "COPACOBANA - A Cost-Optimized Special-Purpose Hardware for Code-Breaking", IEEE Symposium on Field-Programmable Custom Computing Machines — FCCM 2006, Napa, USA, April 2006.
- S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, M. Schimmler, "A Configuration Concept for a Massive Parallel FPGA Architecture" , International Conference on Computer Design — CDES 2006, Las Vegas, USA, June 2006.
- S. Kumar, C. Paar, "Are standards compliant elliptic curve cryptosystems feasible on RFID", Workshop on RFID Security 2006, Graz, Austria, July 2006.
- S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, M. Schimmler, "Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker", to be presented at Cryptographic Hardware and Embedded Systems (CHES 2006), Japan, October 2006.