Master Thesis

Hardware/Software Codesign of Elliptic-Curve Cryptography on a Lightweight Microprocessor for RFID

Andrea Höller

Institute of Technical Informatics
Graz University of Technology

In collaboration with

Infineon Technologies Austria AG

Assessor:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger, TU Graz
Supervisor:	Ass. Prof. Dipl.-Ing. Dr. techn. Christian Steger, TU Graz
	Dr. Tomaz Felician, Infineon Technologies Austria AG
	Dipl.-Ing. Norbert Druml, TU Graz

Graz, March 2013

Abstract

The number of European customers becoming victims of product piracy is increasing. One technical solution to restrict the distribution of counterfeited products is Radio Frequency Identification (RFID) and authentication with asymmetric ECC. The calculation of ECC is computationally-intensive. Since the resources of an RFID tag are extremely limited, the design and implementation of ECC on RFID tags is a challenging task.
To achieve a reasonable computation time, previous implementations of ECC on RFID base mainly on pure hardware solutions. However, development teams need flexible systems to react quickly on changing demands of the market, which can be achieved with a lightweight microprocessor.
The goal of this master thesis is to answer the question "Is RFID ready for software-based ECC?". During this work several options of partitioning hardware and software are designed, implemented and evaluated. The first task is to find an efficient way to implement ECC in software. Then hardware is designed to accelerate the software implementation. A big challenge in hardware/software codesign is a trade-off between performance and area. Based on these metrics the different options are compared during this work.
In this thesis a new binary multiplication algorithm with a good performance/storage trade-off is proposed. Furthermore, a new and innovative hardware/software codesign method with virtual addressing is presented. Finally, a coprocessor is proposed that is unmatched by the other variants in terms of performance and area.
The evaluation of the implemented versions shows that the proposed approaches compare well to related work. Additionally, this work shows that software-based ECC is practicable for RFID.

Kurzfassung

In den letzten Jahren stieg die Anzahl an europäischen Konsumenten, die Opfer von Produktpiraterie wurden. Eine technische Lösung um dieses Problem zu bekämpfen, ist die Authentifizierung der Produkte mit Radio Frequency Identification (RFID). Elliptic Curve Cryptography (ECC) stellt ein dazu geeignetes Verschlüsselungsverfahren dar. ECC ermöglicht eine asymmetrische Verschlüsselung mit relativ hoher Sicherheit. Die Längen der dazu benötigten Schlüssel halten sich jedoch in Grenzen.
Dennoch ist ECC sehr rechenintensiv, da viele arithmetische Operationen in einem großen endlichen Feld ausgeführt werden müssen. Das ist der Grund, warum bisher RFID Tags mit ECC Funktionalität hauptsächlich als spezielle Hardwarelösungen entwickelt wurden. Jedoch wird dadurch der Grad an Flexibilität bei der Entwicklung stark eingeschränkt. Um schnell auf neue Anforderungen des Marktes reagieren zu können, wurde von Infineon Technologies Austria AG ein 8-Bit Mikroprozessor mit sehr kleinen Flächenanforderungen speziell für RFID Produkte entwickelt.
Das Ziel dieser Masterarbeit ist es herauszufinden, ob eine softwarebasierte Implementierung von ECC auf diesen Mikroprozessor praktisch anwendbar ist. Dazu werden verschiedene Implementierungsvarianten präsentiert und unterschiedliche Möglichkeiten der Hardware/Software Partitionierung aufgezeigt. Eine große Herausforderung bei Hardware/Software Codesign ist der Kompromiss zwischen Performanz und Fläche. Deshalb werden die Implementierungsvarianten bezüglich dieser Kenndaten verglichen.
Es wird ein neuer Algorithmus zur Multiplikation im binären Feld mit einem guten Performanz /Speicher-Verhältnis vorgestellt. Des Weiteren wird eine neue und innovative Hardware/Software Codesign Methode mit Virtueller Adressierung präsentiert. Abschließend wird eine Koprozessor Variante beschrieben, die von den anderen Varianten ungeschlagen ist was Rechenzeit und Fläche anbelangt.
Eine Analyse der implementierten Versionen zeigt, dass die vorgeschlagenen Methoden im Vergleich zu ähnlichen Implementierungen in der Literatur, ansprechende Ergebnisse liefern. Des Weiteren wird gezeigt, dass die Implementierungsvarianten auch praktisch einsetzbar sind.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz,am ....
(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Acknowledgements

First I would like to thank Dr. Tomaz Felicijan for paving the way to write this thesis at Infineon Technologies Austria AG, his constructive criticism and for examining this document. Next, I would like to thank the entire contactless memory team for their great support and the friendly working atmosphere they provided.
I would also like to thank Norbert Druml from the Graz University of Technology for his patience with questions and for his assistance to improve the quality of the thesis. In addition I would like to thank Ass. Prof. Dipl. Ing. Dr. Christian Steger from Graz University of Technology for his great support. I would like to extend my thanks to the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the project META[:SEC:] under the FIT-IT contract 829586.
Last, but most certainly not least, my special thanks go out to my boyfriend Thomas for his unwavering support and understanding. Finally, I would like to thank my parents for their aid over the course of my academic studies.

Graz, March 2013 Andrea Höller

Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
AFE	Analog Front End
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
ASICs	Application Specific Integrated Circuits
CLBs	Configurable Logic Blocks
DES	Data Encryption Standard
DMA	Direct Memory Access
DPA	Differential Power Analysis
DPH	Data Pointer High
DSP	Digital Signal Processor
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
EEPROM	Electrically Erasable Programmable Read-Only Memory
FSM	Finite State Machine
GE	NAND Gate Equivalent
GECCO	Genuine verification with Elliptic Curve Cryptography One-way authentication
GF	Galois Field
GPPs	General-Purpose Processors
HDL	Hardware Description Language
IC	Integrated Circuit

IEEE	Institute of Electrical and Electronics Engineers
IPR	Intellectual Property Rights
IR	Instruction Register
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ISO	International Organization for Standardization
LUT	Look Up Table
MIPS32	Microprocessor without Interlocked Pipeline Stages 32
MSB	Most Significant Bit
NAF	Non-Adjacent Form
NFC	Near Field Communication
NIST	National Institute of Standards and Technology
NTRU	Number Theorists aRe Us
PC	Program Counter
PCS	Program Counter Stack
RF	Radio Frequency
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RSA	Rivest, Shamir and Adleman
RTL	Register Transfer Level
SCA	Side-Channel Attack
SEC	Standards for Efficient Cryptography
SPA	Simple Power Analysis
UID	Unique Identification Number
VA	Virtual Addressing
VE	Virtual Element
WSNs	Wireless Sensor Networks
XOR	Exclusive Or

Chapter 1

Introduction

The subject of this thesis is to investigate the feasibility of Elliptic Curve Cryptography (ECC) on a lightweight microprocessor for Radio Frequency Identification (RFID). Different hardware/software partitioning variants will be designed and implemented. Thereafter the proposed options will be compared regarding performance and area.
This chapter provides a brief introduction to the theoretic background related to this thesis. First, a use case and the advantages of public-key and software-based cryptography describe the motivation of the work. The basic principles of RFID are outlined in Section 1.3. Since the implementation will be realized on a microprocessor, Section 1.4 provides an introduction to microprocessors. Section 1.5 describes the principles of cryptography. Finally, Section 1.6 introduces the reader to ECC.

1.1 Goals

The aim of this work is to answer the question „Is RFID ready for software-based ECC?“. The implementation is based on a lightweight microprocessor for RFID developed at Infineon Technologies Austria AG. Several hardware/software partitioning variants will be designed during this theses to combine the flexibility of software with the performance of hardware. Furthermore, these variants will be implemented and evaluated in terms of area and performance.

1.2 Motivation

This section describes the main motivation for this thesis by presenting a use case of the target implementation and showing the advantages of public-key cryptography.

1.2.1 Application-Scenario: Brand Protection

There is a high risk that physical products are not what they should to be. Authentication via RFID systems has a potential to restrict the distribution of counterfeited products.
Intellectual Property Rights (IPR) have been established to protect the rewards of investments in innovation. However, infringements of IPR are a widespread phenomenon. Reports about IPR presented in July 2012 by the European Commission [1, 2] show that

there is a big underground industry producing and distributing illegal product copies all over the world. The statistics in the report illustrate the impact of product piracy in Europe. There is still an upward trend of the number of interception cases reported by customers. In the year 2011 this number increased by 15% compared to 2010, and more than 1000% over the past decade (see Figure 1.1).

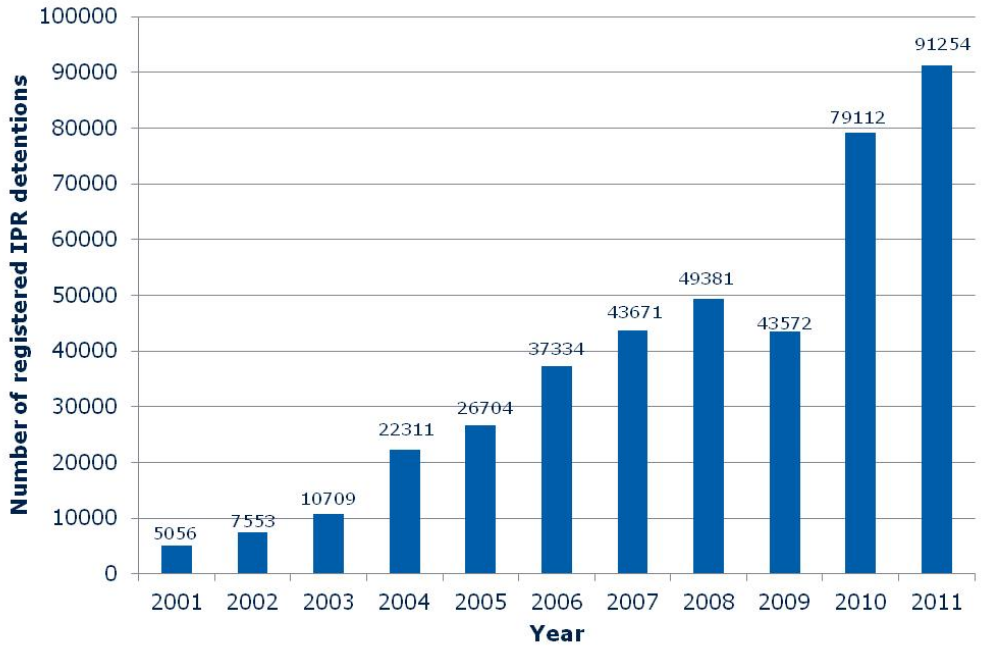


Figure 1.1: Number of IPR detentions presented by the total number of cases represented by an interception by customs over the past decade. Adapted from [1].

In total the value of the equivalent genuine products of the detected infringing articles are estimated to be over 1.2 billion US dollars. The reduced business could cause job losses and reduction of wealth creation. The most affected product categories were medicines (24%), cigarettes (18%), clothing (4%), accessories for mobile phones (3%) and labels, tags and stickers (2%). A worrying trend is that the proportion of products that could be potentially dangerous to the health and safety of customers (i.e. food and beverages, body care articles, medicines, toys) increased from 15% of the total amount in 2010 to 29% in 2011.
The trade of infringing good affects nearly everyone in Europe. The business suffers from a lower demand for legitimate products resulting in lower business revenues. Additionally, enterprises investing in research and development lose their benefits, since others copy their inventions without investment costs. The cloned products are of a lesser quality than the original products- Thus the enterprises selling the genuine goods, often suffer from a damaged reputation. Customers are concerned, because they buy expensive cloned goods, which often do not fulfill the quality standards of the original. Finally, violations of IPR cause declining research and innovation. In addition governments loose revenues of duties and taxes.

The authentication of a product could be realized by attaching the RFID tag to the product. This means that the originality of the product can be verified by the authentication of the tag. Authenticity can be proved by showing the knowledge of a certain secret key by successfully executing a challenge-response protocol. The basics of challenge-response protocols are described in Section 1.5.
The following reasons show that an authentication of products via RFID tags may be an efficient technological option as a protection against counterfeiting [3]:

- Today, RFID systems are able to perform secure authentication of tags using cryptography.
- RFID is already widespread in many applications for logistic purposes.
- Near Field Communication (NFC) technology spreads out in mobile phones. With NFC enabled phones the consumers can directly verify the authenticity of tagged products.

1.2.2 Why ECC?

Public-key cryptography provides a simpler key management than symmetric cryptography, since no secret key is required on the reader's side [4]. Thus public-key cryptography is more reasonable in open-loop applications.
However, public-key cryptography is computationally expensive and the implementation on resource limited RFID tags is challenging. ECC relies on a very hard mathematical problem and thus offers a suitable security with low key sizes. Therefore, ECC can be executed faster and requires smaller area and less energy compared to conventional integer-based public-key algorithms like RSA [4].

1.2.3 Why software-based ECC for RFID?

Implementations in literature show that ECC is practical on RFID tags ([5, 6, 7, 8, 9] etc.). However, these implementations are realized as hardware solutions in form of Application Specific Integrated Circuits (ASICs). Customized ASICs are able to satisfy specific constraints concerning size, performance and power consumption [10]. This can be achieved, since an optimal architecture is used for a specific application.
The manufacturing of an ASIC is a long and expensive process. Shrinking geometry causes the design complexity to grow exponentially. Low development costs and time-to-market are becoming increasingly important for semiconductor companies. Microprocessors are easily programmable and support a broad range of possible applications. Thus designers are able to implement complex systems in an efficient and fast way in software. This flexibility eases to change the design and the development team is able to react quickly on specific customer requirements [10]. Figure 1.2 gives an overview of the target system and the advantages of the approach of this theses.

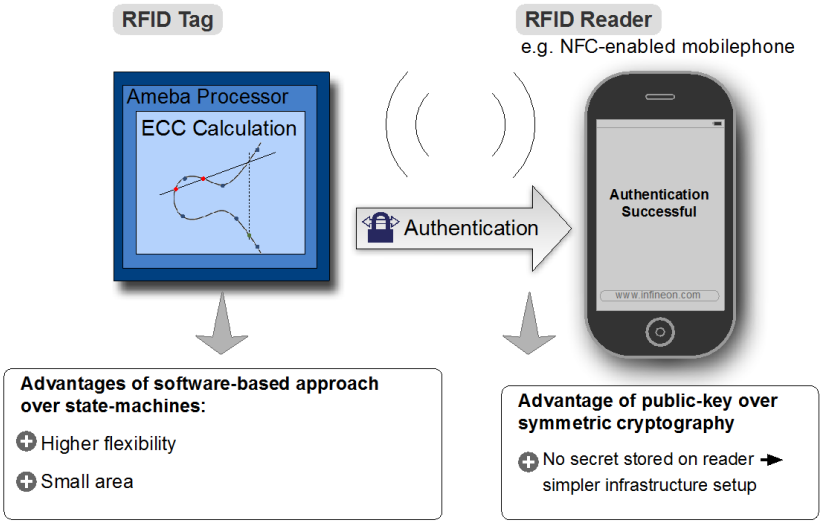


Figure 1.2: Overview of the target system.

1.3 Introduction to Radio Frequency Identification

RFID systems are one of the most pervasive computing technologies. They offer a diverse range of applications enabled by their low cost and broad applicability [11].
RFID has become popular for achieving automatic identification. RFID systems can provide information about people, animals, goods and products in transit. Common applications are supply chain management, transport ticketing, access control, animal tracking and key-less entry for automobiles.
This section does not attempt to convey all of the details of RFID, but gives a brief introduction. A full comprehensive source about RFID is the "RFID Handbook" [12].

1.3.1 Operating Principle of RFID Systems

An RFID system consists of two components: a transponder and a reader [13, p.7–9]. The working principle of a passive RFID system is illustrated in Figure 1.3.

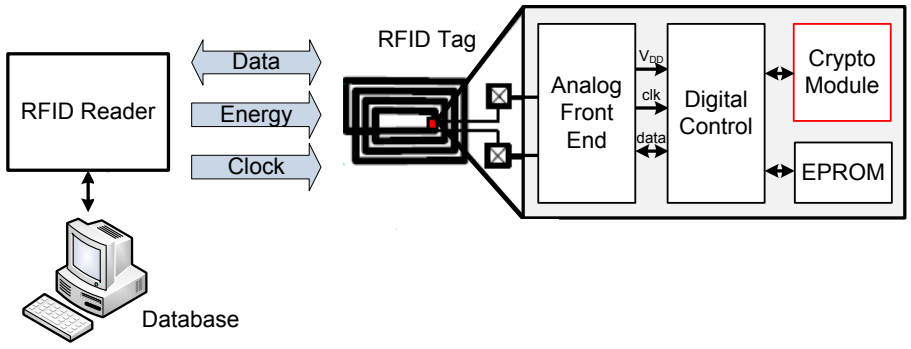


Figure 1.3: Basic scheme of a passive RFID system. Adapted from [13, p.7] and [14].

The transponder is located on the object, which should be identified. Another name for the transponder is tag. The reader reads or writes the tag and contains a Radio Frequency (RF) transmitter and receiver, a control unit and a coupling element to the tag. The tag normally includes a coupling element (like a microwave antenna) connected to an electronic microchip.
The reader can be connected to a back-end database that collects information of the tagged objects or works stand-alone [14]. The energy is transmitted over the air by means of an electromagnetic field. This field is generated by the reader and is also used for the communication to the tag.
In contrast to active RFID systems, in passive systems the RFID tag is not equipped with an internal power source [13, p.13]. Passive tags extract the required energy from the electromagnetic field. The tag is only activated within the interrogation zone of the reader. A big challenge for the designers of RFID systems is that the power supply of the tag is limited, since the energy is transmitted over the air.

The RFID Tag

The tag consists of an antenna attached to an Integrated Circuit (IC). The tag architecture can range from a low-capability device (e.g. for pet identification) to a powerful contactless smartcard (e.g. for biometric passports)[15].
The modules of a typical tag IC are shown in Figure 1.3. The Analog Front End (AFE) provides power supply and performs the communication using a modulator and a demodulator. Furthermore, the AFE recovers the clock signal out of the field.
The digital part of the tag realizes the decoding of the data and creates answers. Additionally, an anticollision algorithm is implemented in the digital control unit. An anticollision enables the support of several tags located in the interrogation zone of the reader at the same time.
The NVM holds all the information that needs to be stored, even if the tag is not supplied with power. For RFID tags an Electrically Erasable Programmable Read-Only Memory (EEPROM) is the most common way to realize the NVM. In general, tags with advanced security features also have a own crypto module [14].

Classification of RFID Systems

According to their mode of transmission RFID systems, there is a differentiation between

- full duplex (FDX),
- half duplex (HDX) and
- sequential (SEQ)

systems [13, p.11–25]. A transponder in full and half duplex systems answers to a broadcast, if the RF field is activated. In sequential systems the field of the reader is briefly switched off periodically. When the tag recognizes such a gap it sends the data. The main drawback of these procedure is the loss of power.

Another differentiation criteria is the range resulting from the transmission frequency of the reader (operating frequency). These frequencies are classified into

- LF (low frequency, 30-300 kHz),
- HF (high frequency, 3-30 MHz) and
- UHF (ultra high frequency, 300 MHz - 3 GHz).

The categories of ranges are close-coupling (0-1 cm), remote-coupling (0-1 m) and long range (>1 m).

Near Field Communication

NFC is a relatively new technology combining RFID and mobile communication [12]. An NFC enabled mobile phone can emulate both, an RFID reader and a contactless smart card. Furthermore, mobile phones can communicate with each other over the NFC interface in a similar way as it is done via Bluetooth. Data transmission with NFC is done with magnetic fields at a transmission frequency of 13.56 MHz and a maximal range of 20 cm.
The NFC communication standard is compatible with some RFID communication standards. Thus relative cheap RFID readers become available to everyone. This is the reason why, NFC offers new application scenarios for RFID technology.

1.3.2 RFID Security

In [16] RFID security is defined as the ability of keeping the information transmitted between the tag and the reader secure from non-intended receptions.
In the early years of RFID development security was not emphasized, because RFID technology was seen as an alternative identification method to barcode systems for a supply chain management [17].
Nowadays the capabilities of RFID systems have increased and they are used more often in high security applications such as access control, contactless payment or ticketing [12]. RFID tags not only send a Unique Identification Number (UID) to the reader, but they are able to perform complex calculations and protocols.
According to their cryptographic functionality RFID systems can be classified as follows [17]:

- Tags without cryptographic functions
- Tags with weak cryptographic operations such as pseudonumbergenerators or hashing
- Tags implementing symmetric ciphers (such as Advanced Encryption Standard (AES))
- Tags implementing asymmetric ciphers (such as ECC)

There have been several RFID solutions based on a custom security algorithm. One famous example of such an RFID system that gained attention because of a weak security mechanism is the widely used Mifare Classic [18]. In [19] Garcia et al. showed an easy

hack of the proprietary Crypto-I algorithm used in Mifare Classic tags.
A better approach is not to use proprietary algorithms, but to follow the Kerchoff's principle. This principle states that a cryptosystem should not be based on the secrecy of the algorithm, but on the key [20]. Thus, on-tag encryption should use standardized cryptographic algorithms (such as AES or ECC).
To enable security for low-cost tags, there is a possibility of "light-weight cryptography" meaning the redesign of existing cryptographic primitives especially for resource limited tags [21]. Until today there is no proof that these security mechanisms provide the same security as standard solutions. Thus, RFID systems using standard security algorithms are still considered as a preferred option.

1.4 Introduction to Microprocessors

The basic model of a computer is the *Von Neumann model* as shown in Figure 1.4 [22, Ch.1].

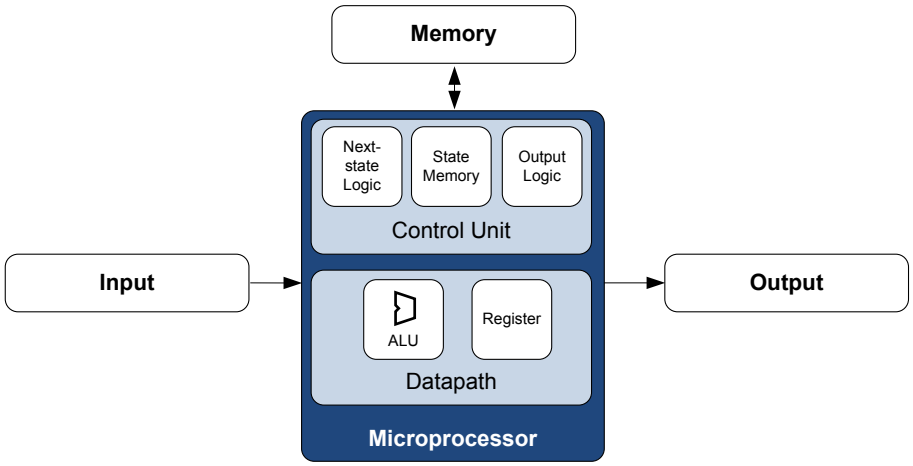


Figure 1.4: Von Neuman model. Adapted from [22, Ch.1].

In the so-called *Harvard architecture*, there are separate memories for instructions and data [23, p.3].
The microprocessor consists of a datapath and a control unit. The datapath is responsible for the execution of all data operations, which are performed by the microprocessor (e.g. adding two numbers inside the Arithmetic Logic Unit (ALU)) [22, Ch.1]. Furthermore, the datapath includes registers for temporary storing data [24].
The control unit is responsible for fetching the instruction to be executed from the memory, decoding and executing it [23, p.83]. The performance of a program executed by a microprocessor is analyzed with clock cycles [23, p.6]. The execution time can be defined as

$$execution\ time = \frac{clock\ cycles}{frequency} \Rightarrow clock\ cycles = execution\ time \cdot frequency \quad (1.1)$$

To compare different design alternatives the speed-up is determined with the following equation [23, p.10]:

$$\text{speed-up} = \frac{execution\ time\ before\ enhancement}{execution\ time\ after\ enhancement} \tag{1.2}$$

1.4.1 Instruction Set Architecture

Instruction Set Architecture (ISA) defines the boundary between hardware and software [25, p.11–15]. The instruction-set is visible to a software programmer. The basic classification of ISAs is according to their type of internal storage. The major options are a stack, an accumulator or a set of registers. The stack and accumulator approaches define the operands implicitly on top of the stack or at the accumulator.
Register architectures can be divided into register-memory architecture and load-and-store architecture. The first is to access memory as part of a instruction. The load-and-store architecture only allows to access memory with load and store instructions. The architecture is also defined by the operand size, which is today typically between 8 bit and 64 bit.

Register Set

There are two basic types of registers: general-purpose registers for multiple purposes and special-purpose registers restricted to specific functions [23, Ch.5]. For fetching an instruction and executing it, there are two registers: the Program Counter (PC) and the Instruction Register (IR). The PC contains the address of the next instruction. The IR holds the current instruction. After one instruction has been fetched, the PC is updated to point to the next instruction, which should be executed. Processors can support further registers to maintain status information or special-purpose address registers.

1.4.2 General-purpose Microprocessors vs. ASICs

ICs can be designed with General-Purpose Processors (GPPs) or dedicated hardware [22]. GPPs are able to perform a variety of computations. Not every computation is hardwired into the processor, but represented by a stored sequence of instructions. This program is executed by the microprocessor and can be easily changed. The drawback of general-purpose microprocessors is that the performance and power consumption is in general worse than dedicated hardware.
When designing ASICs, the developer has the full control over every aspect of architecture, circuit and layout design. This is the reason why it is possible to optimize the performance and energy efficiency.
The development often simultaneously requires performance, agility, low power and a modest design effort. Therefore, it is desirable to combine the throughput and energy efficiency of dedicated hardware for demanding computations with the convenience and flexibility of a microprocessor.

Coprocessors

One option to combine the flexibility of software with the performance of hardware is to use a software-controlled microprocessor cooperating with dedicated hardware units. As shown in Figure 1.5, coprocessors are specialized hardware units controlled by a software-programmable host. Each coprocessor accepts a limited number of instructions.
To exchange data the host can send the input data to the coprocessor. Another possibility is to keep the data in memory and the coprocessor accesses the memory via Direct Memory Access (DMA).

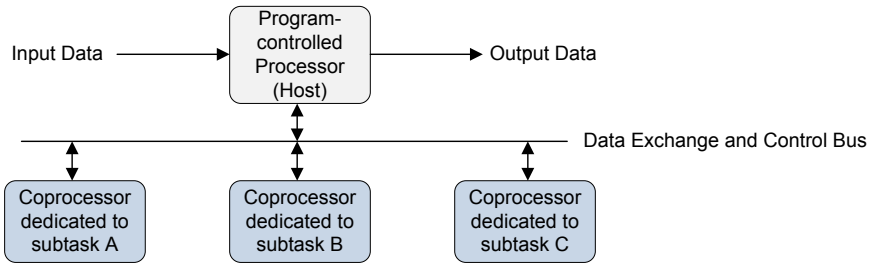


Figure 1.5: Principle of general-purpose processor with specialized coprocessors.

Instruction-Set Extension

Another option to accelerate the software is Instruction Set Extension (ISE). As stated in [26], there are several processors with an enhanced support for ISE available on the market (for example Lx designed by Helwett-Packard and STMicroelectronics [27]). Also standard microprocessor platforms can be extended by instructions to accelerate the processing of a specific problem, such as multimedia or cryptography.
Compared to coprocessors, extending the instruction set involves less communication overhead. In general the area of instruction set extensions is smaller, since the additional instructions are tightly coupled to the processor. Another advantage is that the flexibility and scalability of an additional instruction is higher than a coprocessor [28]. The major challenge of ISE is the identification of these additional instructions, which provide the largest speed-up.

1.5 Introduction to Cryptography

This section describes the basic principles of cryptography to help understanding the role of ECC in the wide range of cryptographic approaches. For more detailed information the two books "Introduction to Cryptography" [29] and the "Handbook of Applied Cryptography" [30], are recommended.
The basic scenario of secure communications is the following: two parties, for example Alice and Bob, want to talk over an insecure channel, but they do not want other parties to understand their messages. Furthermore, there is a potential eavesdropper called Eve. Alice and Bob agree to a certain method of how to obscure the information. Alice creates a ciphertext by encrypting the plaintext with a key. The ciphertext is sent over the insecure

channel to Bob, who decrypts the message using a decryption key to get the plaintext. There are four goals of cryptography [29]:

- Confidentiality: Eve should not be able to read the message
- Data integrity: Eve should not be able to manipulate the message without Alice and/or Bob noticing it
- Authentication: Deals with the identification of the two parties. For example Bob wants to be sure that only Alice could have sent the received message.
- Non-reputiation: A party cannot deny having sent a message.

The subject of this master thesis is to establish authentication.

1.5.1 Symmetric and Asymmetric Cryptography

Symmetric cryptography means that both parties share the same secret key K for encryption and decryption, or one key can be easily derived from the other [31]. The principle of symmetric cryptography is shown in Figure 1.6.

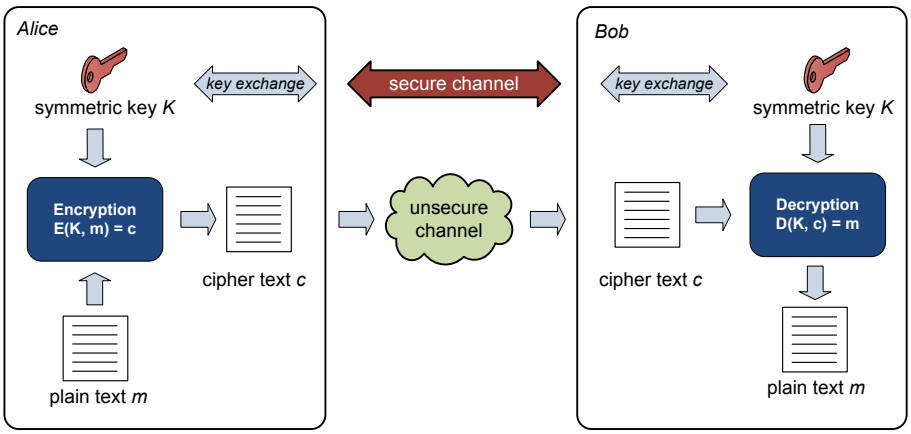


Figure 1.6: Principle of encryption using symmetric cryptography.

To send an encrypted message, Alice uses an encryption function E to calculate the cipher text based on the shared key K and the plain message m and sends it over an insecure channel. Bob calculates the plain text with the decryption function D using the shared key. The main drawback of symmetric cryptography is the key distribution problem caused by the need of both parties knowing the same key. For this key distribution a secure channel would be required. The most popular block ciphers are Data Encryption Standard (DES) [32] and the more secure Advanced Encryption Standard (AES) [33]. Stream ciphers do not collect data until a full block length is reached, but encode every character immediately.
In the year 1976 Diffie and Helmann revolutionized cryptography by presenting the idea of public-key cryptography, which is also called asymmetric cryptography [31]. The approach is to publish an encryption key, which is also called a public-key. It is computationally impossible to calculate the decryption key out of this public information, without an

additional information (private key), which is only known by the owner.
The security is not based on the secrecy of the key, but on the computational impossibility of calculating the private key. The asymmetric encryption scheme is illustrated in Figure 1.7.

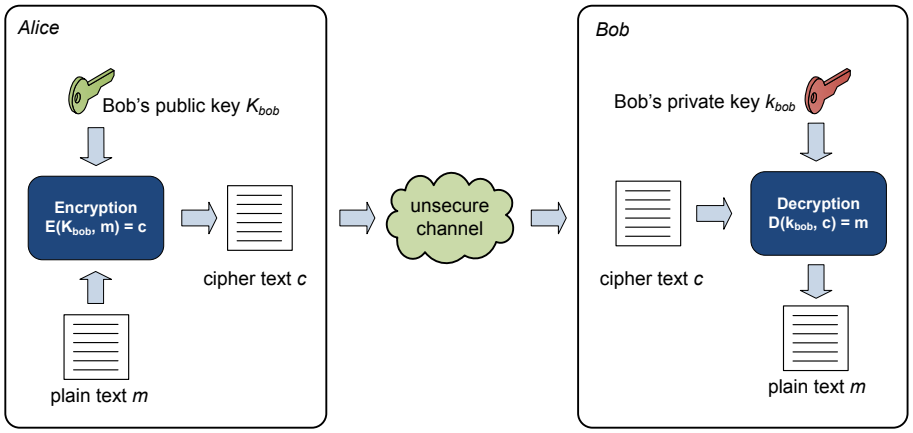


Figure 1.7: Principle of encryption using asymmetric cryptography.

Alice uses Bob's public-key K_{bob} to generate a cipher text c. This cipher text c can be sent over an insecure channel and is decrypted by Bob using his private key k_{bob}.
The first realization of such a system was Rivest, Shamir and Adleman (RSA) [34] presented in the year 1978. The mathematical problem of RSA is based on factoring large integers. Other versions of public-key cryptography are the ElGamal system based on the discrete logarithm problem, the lattice based Number Theorists aRe Us (NTRU) [35] and the error correcting codes based McEliece system [36]. ECC is a public key cryptosystem based on the discrete logarithm problem of a random elliptic curve. Public-key systems are very powerful but require much more computational effort than symmetric cryptography [31]. Thus, these methods are only recommended for encrypting small amount of data.

1.5.2 Principle of Authentication with Asymmetric Cryptography

Authenticity can either be assured on something known (i.e. a password), something possessed (i.e. a passport) or something the claimant is (i.e. proved by special biometrics) [6]. Most popular authentication methods are based on the challenge-response principle, which proves the authenticity by something known.
The basic procedure of the challenge-response authentication using asymmetric cryptography is illustrated in Figure 1.8.

If Bob wants to authenticate of Alice, he generates a random challenge and sends it to Alice. This challenge is then encrypted with the private key k_{alice} of Alice. The cipher text is sent to Bob, who decrypts the message using the public key k_{alice} of Alice. Only if the message is encrypted with the right private key, the output corresponds to the previous generated challenge. Since Bob can now be sure that Alice knows the right private key, the authentication of Alice is achieved.

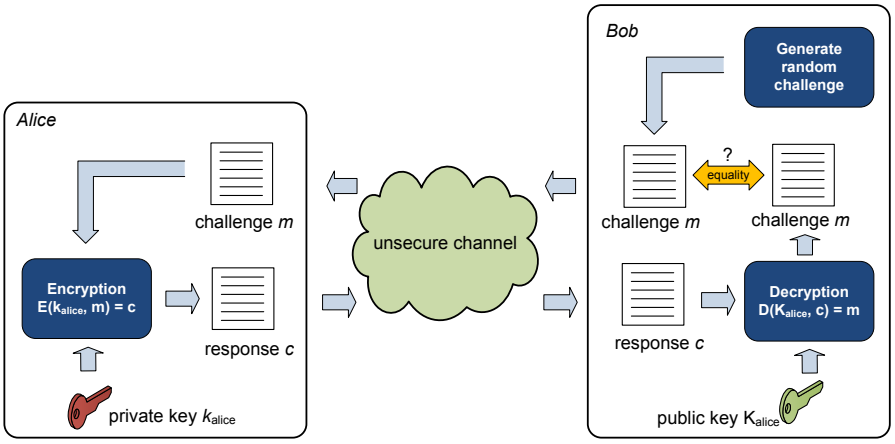


Figure 1.8: Principle of authentication using asymmetric cryptography.

1.5.3 Side-channel Attacks

When designing hardware performing cryptographic operations based on a secret key one has to keep side channel attacks in mind. This section presents a short overview of some Side-Channel Attack (SCA)s as described in [37] and [38]. An SCA does not attack the mathematical basis of the cryptographic system, but on implementation-specific aspects by analyzing physical characteristics. Examples for such characteristics are processing time, power consumptions, electromagnetic emission or faulty outputs.

Timing Attacks

Timing attacks analyze the execution time of an cryptographic algorithm processing a set of messages to get the secret parameters. To prevent timing attacks, the calculation time of the cryptographic algorithm should not depend on the secret.

Power Analysis Attacks

Power analysis attacks exploit the fact that the processed data of a cryptographic device has an impact on the power consumption [39]. The simplest form is the Simple Power Analysis (SPA), where a single power trace is visually analyzed to guess the secret.
A more advanced method is the Differential Power Analysis (DPA), which uses statistical analysis of the power consumption of different measurement for several cryptographic operations. To prevent DPA attacks, the power consumption should not depend on the processed secret data.

Fault Induction Attacks

Fault injection attacks examine cryptographic algorithms under malfunctioning. Information about the secret data can be reconstructed by analyzing the faulty output calculations. Faults can be introduced for example by freezing memory cells, by white light or by an abnormally high or low clock frequency or voltage in the power supply, etc. [40].

One method against fault induction attacks is to perform the calculation two times and compare the results.

1.6 Introduction to Elliptic Curve Cryptography

This section presents the basic principle of ECC. A comprehensive book about ECC is the "Guide to Elliptic Curve Cryptography" written by Hankerson et al. [41].
Already hundreds of years ago mathematicians deal with the specific properties of elliptic curves [41, p.1]. In the year 1985 Neal Koblitz and Victor Miller proposed independently of each other the use of elliptic curves in public-key systems. Five years later ECC attracted public attention, several standards for ECC were published and some private companies integrated ECC in their security devices.
Mathematically, an elliptic curve is defined as follows [41, Ch.3.1]:

Definition 1 *An elliptic curve E over a field K is defined by the Weierstrass equation*

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and $\Delta \neq 0$, where Δ is the discriminant of E and is defined as follows:

$$\begin{gathered}\Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6,\\ d_2 = a_1^2 + 4a_2,\\ d_4 = 2a_4 + a_1a_3,\\ d_6 = a_3^2 + 4a_6,\\ d_8 = a_1^2a_6 + 4a_2a6\_a_1a_3a_4 + a_2a_3^2 - a_4^2,\end{gathered}$$

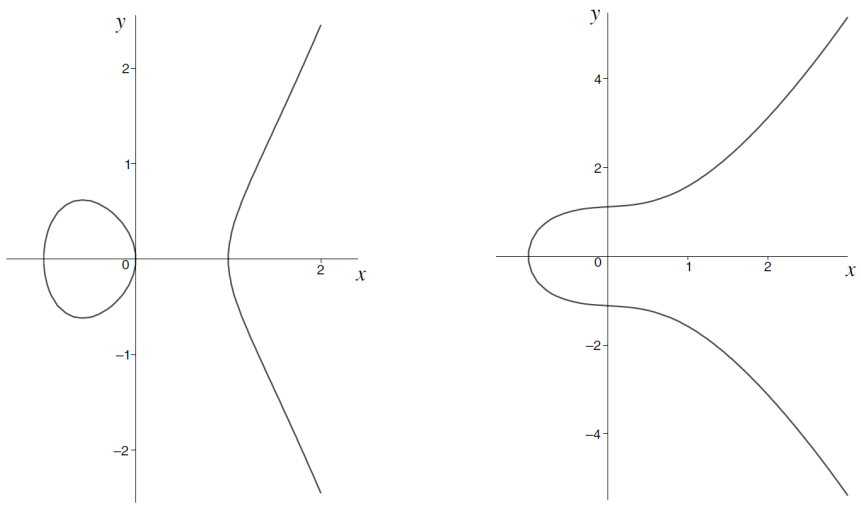
The variables x and y cover a plane and are defined over a underlying field K, the resulting elliptic curve is written as $E(K)$. The field can be any finite field like complex, integers, prime fields or binary fields. What makes the mathematic of elliptic curves deep is this underlying field [42, Ch.5.1]. Example plots of elliptic curves over real numbers on a real plane are illustrated in Figure 1.9.
An elliptic curve consists out of the points that fulfill the equation and the point at infinity. The order of an elliptic curve over a finite field $\mathbb{F}_q$ is written as $\#E(\mathbb{F}_q)$ and denotes the number of points on the curve [41, Ch.3.1.3]. It can be shown that $\#E(\mathbb{F}_q) \approx q$. The higher the order of the curve, the more security a curve provides.
A famous choice of elliptic curves are non-supersingular curves, which are defined with a simplified Weierstrass equation [41, Ch.3.1.1].

Definition 2 *A non-supersingular curve is defined as*

$$E : y^2 + xy = x^3 + ax^2 + b$$

where $a, b \in K$.

The mathematical problem ECC is based on, is the Elliptic Curve Discrete Logarithm Problem (ECDLP), which relies on the non-invertibility of the elliptic curve point multiplication [41, Ch.4.1]. The ECDLP describes that if Q and P are two given points on an elliptic curve and $Q = k \cdot P$, $k \in \mathbb{Z}$ it is hard to find k. All cryptographic functions like encryption, signature generation/verification and key agreement are based on this point multiplication.



(a) $E_1 : y^2 = x^3 - x$ (b) $E_2 : y^2 = x^3 + \frac{1}{4}x + \frac{5}{4}$

Figure 1.9: Elliptic curves over a real field [41].

1.6.1 Advantages of ECC

Asymmetric cryptosystems have different underlaying mathematical problems. Cryptographical methods with weaker mathematical problems require larger parameter sizes to provide the same level of security. Higher parameter sizes influence the performance negatively. The ECDLP is a very hard problem and thus ECC achieves high security with low parameters compared to other public-key systems like RSA. For example, ECC with a key length of 160 bits provides about the same security as RSA with 1024 bit keys. Therefore, ECC is faster and requires less space and energy than RSA [43]. This makes ECC an ideal candidate for systems like RFID, where processing power, storage, bandwidth and power consumption is constrained.
Furthermore, ECC has received commercial acceptance and is included to standards of accredited standard organizations such as American National Standards Institute (ANSI), Institute of Electrical and Electronics Engineers (IEEE), International Organization for Standardization (ISO) and National Institute of Standards and Technology (NIST).

1.6.2 Finite Fields

Thus the choice of the underlying Galois Field (GF) is essential for the efficiency of the implementation [44]. A GF is a finite set of values with the operations addition, multiplication and inversion. These operations always result in new values that are also in the field. The order of a field describes the number of elements in it.

From the mathematical point of view the underlying finite field has to fulfill the properties of an Abelian group which is defined as follows

Definition 3 *An Abelian group $(G, *)$ consists of a set G with a binary operation $*: G \times G \rightarrow$ satisfying the following properties $\forall a, b, c \in G$:*

1. *Associativity $a * (b * c) = (a * b) * c$*
2. *Existence of an identity $\exists e \in G$, such that $a * e = e * a = a$*
3. *Existence of inverses $\exists i \in G$, such that $a * b = b * a = e$*
4. *Commutativity $a * b = b * a$ [41, Ch. 1.2.3]*

An overview of the most popular fields for ECC is illustrated in Figure 1.10.

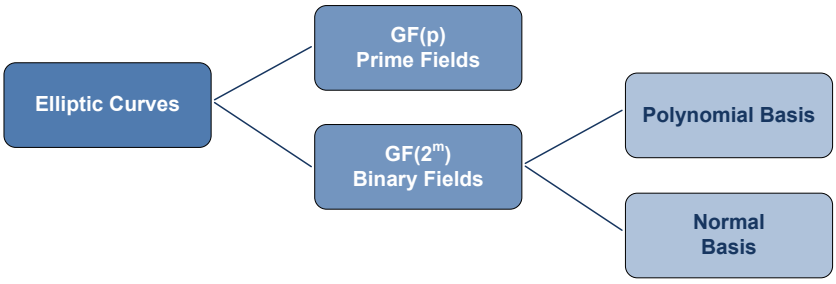


Figure 1.10: Common used underlying finite field options for ECC. Adapted from [44].

The set of a prime field is made up of every value between one and the prime number and is denoted by $\mathbb{F}_q$ [42, Ch. 5.1]. To put it simply, all operations in a prime field are performed modulo the prime number.
A binary field consists of all numbers that can be represented by a set of bits [44]. Binary fields are written as $\mathbb{F}_{2^m}$. If $m \geq 2$ the field is called binary extension field. For simplicity, throughout this text such fields will also be called binary fields. To indicate the property of being a GF, prime fields and binary fields can also be indicated by writing $GF(p)$ or $GF(2^m)$.
There are two basis representations for binary fields: polynomial and normal basis. Polynomial basis representation means that the elements of the field are constructed with binary polynomials.

Definition 4 *A binary polynomial is a polynomial of degree at most $(m-1)$ of the form*

$$\mathbb{F}_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \ldots + a_1 z^{+} a_0 : a_i \in \{0, 1\}\}$$

where the coefficients are elements of the prime field $\mathbb{F}_2$, where $a, b \in K$ [41, p.26].

It is easy to directly convert the binary polynomial to a bit representation by setting the bit m to the value of the coefficient a_m [44]. For example the bit representation `01100101` would be represented by the polynomial $z^6 + z^5 + z^2 + 1$.
For polynomial basis representation of a modulus similar to the prime number in prime fields is necessary [41, Ch. 2.1]. The modulus is represented by an irreducible polynomial

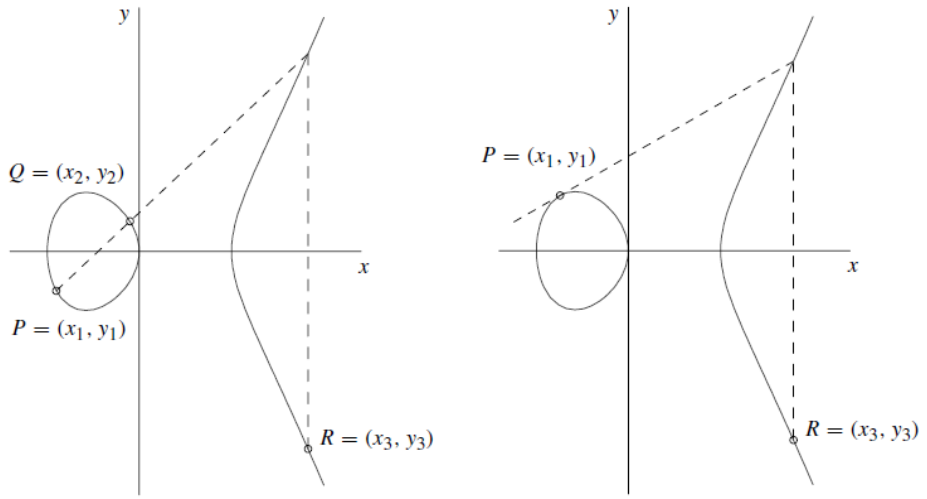
with properties comparable to prime numbers. An irreducible polynomial with degree m cannot be factored as a product of binary polynomials with a degree less than m. Another basis for binary field is the normal basis. In normal basis representation elements are expressed in terms of a basis of the form $\{b, b^2, b^{2^2}, ..., b^{2^{m-1}}\}$. For example, the bit sequence `01100101` would be represented by the polynomial $b^{2^6} + b^{2^5} + b^{2^2} + b$.

1.6.3 Elliptic Curve Arithmetic

The elliptic curve arithmetic is the arithmetic involving the points of the curve [41, Ch.3].

Group Law

The so called *chord-and-tangent* role defines how to add and double points on an elliptic curve. The addition and doubling rules can be expressed graphically (see Figure 1.11).
If P and Q are two points on the curve, the sum of these points can be determined by first drawing a line trough P and Q. This line intersects a point at the elliptic curve. The reflection of this point about the x-axis is the summation point of P and Q.
The doubling of a point can be explained similarly. If P should be doubled draw the tangent of P and reflect the intercepted point of the tangent about the x-axis. This point equals two times the point P.



(a) Addition: $P + Q = R$. (b) Doubling: $P + P = R$.

Figure 1.11: Geometric addition and doubling of elliptic curve points [41].

Together with these operations, the set of points of E defined over K forms an Abelian group with the point at infinity ∞ serving as identity element. Next, the group law for non-supersingular curves are defined.

Definition 5 *Group law for non-supersingular elliptic curves over binary fields* $\mathbb{F}_{2^m}$*:*

1. *Identity:* $P + \infty = \infty + P = P \ \forall P \in E(K)$
2. *Negatives: If* $P = (x_1, y_1) \in E(K)$*, then also* $-P \in E(K)$*,* $-P = (x_1, y_1 + x_1)$
3. *Point addition: Let* $P = (x_1, y_1) \in E(K)$ *and* $Q = (x_2, y_2) \in E(K)$*, where* $P \neq \pm Q$*. Then* $P + Q = (x_3, y_3)$*, where*

$$x_3 = (\tfrac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2 \text{ and } y_3 = (\tfrac{y_2 - y_1}{x_2 - x_1})(x_1 - x_3) - y_1$$

4. *Point doubling: Let* $P = (x_1, y_1) \in E(K)$ *, where* $P \neq -P$*. Then* $2P = (x_3, y_3)$*, where*

$$x_3 = (\tfrac{3x_1 + a}{2y_1})^2 - 2x_1 \text{ and } y_3 = ((\tfrac{3x_1^2 + a}{2y_1})(x_1 - x_3) - y_1$$

Projective Coordinates

Definition 5 shows that addition and doubling calculations require the calculation of inverses. The coordinates used in definition 5 are called *affine coordinates*.
In most implementations performing an inversion is significantly more expensive than a multiplication. The number of inversions that have to be calculated for a point multiplication can be reduced by using projective coordinates.
For the projective representation of a point a further coordinate Z is introduced. The projective point (X, Y, Z) corresponds to the affine point $(X/Z^c, Y/Z^d)$. If c and d are chosen to be one, the coordinates are called standard projective coordinates and the non-supersingular form of the elliptic curve is defined as

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3$$

Point Multiplication

The ECDLP involves the calculation of $Q = k \cdot P$, where $Q, P \in E$ and $k \in \mathbb{Z}$ [41]. Since ECC is based on the ECDLP, the point multiplication is the essential calculation in elliptic curve cryptosystems. Another name for the point multiplication is scalar multiplication. In some protocols, the point P is known a priori and thus the multiplication can exploit precomputed data depending on P.
The easiest form of multiplying an unknown point is the *double-and-add* method. For example to calculate $5P$ one can calculate $2P$ by point doubling, then again double the point $2P$ to get $4P$ and then calculate one point addition to get $5P = 4P + P$.
Montgomery found in [45] a faster method for performing the point multiplication. He showed that the sum of two points with a known difference can be computed without the y-coordinate. In [46] it is described how the y-coordinate can be recovered out of the result of the Montgomery multiplication. For the Montgomery multiplication the calculation of $6log_2(k)$ field multiplications and $5log_2(k)$ squarings are required [47].

1.6.4 Domain Parameter

ECC domain parameters describe an elliptic curve E over a finite field K, a base point $P \in E(K)$ and its order n [41, Ch. 4.2]. The domain parameter of an elliptic curve define the security of the curve. Special mathematical attacks can be prevented or complicated by choosing the right domain parameters. Furthermore, implementation issues influence the domain parameter choice. For example, the domain parameters are comprised of:

- The field order q
- The field representation K for the elements
- The coefficients $a, b \in K$ defining the equation of the elliptic curve E (e.g. $E : y^2 + xy = x^3 + ax^2 + b$)
- The base point P having prime order
- The order n of P

Well known attacks to ECC are the Pohling-Hellman attack [48] and the Pollard's rho attack [49]. The computational effort of calculating the ECDLP depends on the group order $\#E(K)$.
The Pohling-Hellmann attack reduces the calculation complexity to the greatest factor of $\#E(K)$. The challenge of this attack is to find the prime factorization of the group order, which is hard to achieve in general [41].
Pollard rho methods are algorithms to determine the period lengths of number sequences, which can be applied to calculate the discrete logarithm. Exhaustive key search of a k-bit block cipher is expected to take roughly the same time as the solution of an instance of the ECDLP using Pollard's rho algorithm over a finite field whose order has bit length $2k$. For example the 128-bit AES version can be compared to ECC over binary fields with an dimension of 283 bit [50].
According to [51] the largest broken ECDLP had a key size of 109 bit. About 2600 workstations took 17 months for the calculation. Thus, it is recommended to take a key size higher than 109 bit to ensure computational security.
To be resistant against the Pohling-Hellman and the Pollard rho attacks, $\#E(K)$, which is about m if $K = GF(2^m)$ has to be divisible by a sufficiently large prime. The maximum resistance to the mentioned attacks can be achieved by selecting E, such that $\#E(K)$ is prime.
Standard organizations like ANSI, IEEE, ISO, Standards for Efficient Cryptography (SEC) and NIST offer elliptic curve parameters [52].

1.7 Outline

This Chapter provided a basic introduction to several topics related to this thesis. The next chapter describes the related work. The design developed in this thesis is described in detail in Chapter 3. Chapter 4 provides details about the implementation. The results of the evaluation of the different algorithm variants are presented in Chapter 5. Finally, Chapter 6 concludes this thesis.

Chapter 2

Related Work

In recent years many researchers analyzed the practicability of ECC in low-end devices. This chapter describes the related work classified into different levels of hardware/software codesign.
To provide a meaningful comparison of different systems the timings are given in clock cycles, which are determined according to Formula 1.1 in the introduction. The unit of the area is the normalized NAND Gate Equivalent (GE).

2.1 Software Implementations of ECC

Many publications describe how to efficiently implement ECC in software based on high-end processors. For example, Taverne et al. recently showed in [53] that an elliptic point multiplication using binary fields of an order of 233 can be achieved in 157 thousand clock cycles on Intels Sandy Bridge Core i7 64-bit processor [54]. The following section offers an overview of 8-bit ECC implementations in software for constrained environments, which is still time demanding.
In the year 1999, Michael Rosing published the book "Implementing ECC" [42]. The book offers an introduction to ECC with the programming language C. However, the book does not describe how to implement ECC in an efficient way.
Hankerson et al. described in [50] a detailed study of software implementations with elliptic curves over binary fields recommended by NIST. They proposed several efficient algorithms for binary field operations such as the right-to-left and the left-to-right comb method for a polynomial multiplication.
In terms of software-based ECC for RFID tags, there is little literature available. However, many authors studied the efficiency of ECC in software for the Wireless Sensor Networks (WSNs) application. Although the majority of the implementations are realized on processors, which are too large for RFID tags, the comparison provides a basic understanding of the time and memory requirements. This section starts with a presentation of 8-bit microprocessors used for ECC implementations and then outlines software implementations classified with respect to the underlying field.

8-Bit Microcontroller used ECC Software Implementations

Woodbury et al. have been the first describing how to implement ECC on low-cost microprocessors [55]. They focused on the Intel 8051 family, especially on the Siemens controller SLE44C23S. Smart cards and other devices often use 8-bit microprocessors derived from the 1970s families like Intel 8051. The SLE44C23S offers only 256 bytes of internal RAM, where the lower half can be addressed directly and the upper bytes have to be referenced through a pointer register. Due to this overhead, it takes more time to address this upper part of the memory. Gura et al. compared the performance of ECC and RSA on two 8-bit processors in [56]. The first one was the Chipcon CC1010 supporting the Intel 8051 instruction set. The microprocessor contains 32 kByte Flash memory, 2 kByte external data memory and 128 bytes of internal data memory. For temporary data storage 32 bytes of the internal memory are available.
The second proposed processor was the high-performance low-power ATmega128. This processor implements the AVR architecture from Atmel [57]. The AVR family is based on a highly structured Reduced Instruction Set Computer (RISC) design [58]. The ATmega128 provides 133 instructions - most of them can be executed in one clock cycle. AVR controllers implement the Harvard architecture. The AVR pipeline has two stages: The first stage fetches an instruction and the second one executes it. The most interesting feature, especially for ECC implementations, is a large register set offering 32 8-bit general purpose registers. Most of the 8-bit ECC implementations in literature use this processor. The most similar approach to the work presented here was recently published by Wenger et al. [59]. Their architecture also targets resource-constrained RFID tags. They presented a clone of the ATmega128 called JAAVR (Just Another AVR) with a low power consumption (11 uW/MHz) and a small silicon-footprint (6.5 kGE). They showed the practicability of the ISO 14443 protocol and implemented the cryptoalgorithms AES, Grostl and ECC.

2.1.1 Software Implementations over Prime Fields

Many authors assume that software implementations of elliptic curves over prime fields can be executed faster than those over binary fields. Table 2.1 shows a summary of the performance and resource requirements of several implementations, which are presented below.
Woodbury et al. presented an ECC implementation on the Intel 8051 architecture. Using the built-in multiplier they achieved a general point multiplication in 8.37 seconds at a frequency of 12 MHz which equals to 100.44 million clock cycles.
Gura et al. showed that software based public-key cryptography is viable on a small device [56]. The evaluation of their implementation variants showed that the performance advantage of ECC over RSA increases with the decrease of the word size. They presented a hybrid multiplication algorithm, which exploits the advantages of the operand scanning and product scanning algorithm with the goal to reduce the number of memory accesses. They evaluated assembler implementations of ECC on the CC1010 and the ATmega128 processor. The execution of one scalar multiplication on the CC1010 with a frequency of about 3.7 MHz required 4.56 seconds. The CC1010 is only able to access one bank of eight registers at a time and switching between register banks requires multiple instruction cycles. This limitation does not exist on the ATmega128. Thus, the implementation on the ATmega128 was significantly faster with a runtime of 0.81 seconds at a frequency of

8 MHz, which equals to 6.5 million clock cycles. However, they used the Non-Adjacent Form (NAF)-method for point multiplication. In contrast to the Montgomery multiplication, this algorithm does not include any side channel attack countermeasures.
Kargl et al. compared in [58] implementations of ECC over different fields. They showed that binary field arithmetic can be faster on 8-bit processors. Their fastest point multiplication using prime fields had a runtime of about 13.8 million clock cycles.
In the year 2012, Wenger et al. presented several implementation variants of ECC on the JAAVR over prime fields. Their slowest point multiplication was implemented in C and used the operand scanning method for a field multiplication. This implementation variant is quite memory efficient with a code size of 3.7 kByte. However, regarding the performance of 35.1 MCycles this implementation variant is not practical. Thus, they accelerated their implementation using the operand-caching field multiplication as proposed in [60]. Furthermore, they optimized the efficiently of the code with an assembler. With this approach they reached a runtime of about 13 MCycles, but increased the ROM size to 7.6 kBytes.

Implementation	*Processor*	*Language*	*Length of scalar*	*Runtime [MCycles]*	*Codesize [KB]*	*RAM [Byte]*
Woodbury et al. [55]	SLE44C23S (8051 arch.)	C	134	111.4	13	523
Guara et al. [56]	CC1010 (8051 arch.)	asm	160	16.9	2.1	266
Guara et al. [56]	ATmega128	asm	160	6.48	3.6	280
Kargl et al. [58]	ATmega128	asm	165	13.8	9.8	-
Wenger et al. [59]	JAAVR	asm	190	13	7.6	384

Table 2.1: Comparison of existing ECC implementations over prime fields in software on 8-bit architectures. The presented values relate to one point multiplication.

2.1.2 Software Implementations over Binary Fields

Malan et al. investigated the feasibility of implementing ECC over binary fields for sensor nodes based on the algorithms presented in [42], [50] and [61]. The authors were the first presenting a performance evaluation using elliptic curves over binary fields. Their implementation is based on the MICA2 mote, which supports the ATmega128 microprocessor. They used an ECC calculation to provide a key distribution mechanism for UC Berkeley's TinySec module. For point multiplication, they used the double-and-add algorithm as proposed in [62]. For performing a polynomial multiplication they implemented the left-to-right method as described by López and Dahap in [63]. Their implementation was realized with C++ and a Java library. Their implementation was quite inefficient requiring 34 seconds for one point multiplication. Since they performed the calculation with 7.3825 MHz, this means that they needed about 2,512 million cycles.
Blaß and Zitterbart implemented a version of ECDSA over $GF2^{113}$ [51]. As starting point for their implementation they used Rosings book [42]. They accelerated their implementation with a handcrafted optimization and a precomputation of certain points. Using this approach they achieved a runtime of 6.74 seconds.

Shi and Yan showed that a sophisticated implementation of the inversion algorithm improves the performance of the ECC calculation [43]. They presented a software implementation with a performance of 13.9 seconds at a clock rate of 8 MHz. Furthermore, they analyzed the effect of different word sizes on the execution time. They showed that in general a higher word size means less time for performing one field operation. An exception is the right-to-left multiplication algorithm which is faster using 16-bit word size than 32-bit words, since the number of shift operations is smaller. In their paper they also compared different binary field algorithm variants. Their fastest version of the scalar point multiplication requires about 111 million cycles execution time and 12kByte memory. In [4] Shi and Yan improved their work. They implemented point multiplication with projective coordinates on ATmega128, which required about 97 million cycles.
One feature of the ATmega128 is the large register set of 32 8-bit general purpose registers. The authors of [64] used this feature in TinyECCK to reduce the number of memory accesses required for the binary field multiplication. They observed that the intermediate results of the polynomial multiplication are frequently stored in the same memory positions. They reduced the number of memory operations. combined additions needed for the polynomial multiplication They reduced the number of memory operations by combining two consecutive intermediate additions performed to calculate the polynomial multiplication. Using this approach they reached a runtime of about 8 million clock cycles for a scalar multiplication on a binary field with an order of 163. Furthermore, they showed that a field multiplication over binary fields can be faster than over prime fields on the ATmega128 processor requiring a comparable code size. NanoECC presented from Szeczechowiak et al. in [65] uses the same parameters as TinyECCK. NanoECC implements ECC with projective coordinates over prime and binary fields on the WSN motes MICA2 and Tmote Sky. However, NanoECC does not reach high performance of TinyECCK and requires about 16 million clock cycles for one scalar multiplication. Furthermore, their implementation has high memory requirements, since they used a comb method with pre-computed points.
Kargl et al. [58] improved the approach of Seo et al. [64] by writing the code in assembly language using the CPU registers in an optimal way. They managed to execute the binary field multiplication in about 5,000 cycles, which is the reason for the fast execution time of 6.1 million clock cycles.

Implementation	*Processor*	*Language*	*Length of scalar*	*Runtime [MCycles]*	*Codesize [KB]*	*RAM [Byte]*
Malan et al. [61]	Mica2 (ATmega128)	C	163	2,512	34	1059
Blaß and Zitterbart [51]	Mica2 (ATmega128)	C	113	47.18	75	208
Yan et al. [43]	ATmega128	C	163	111	11.6	820
Szczechowiak et al. [65]	ATmega128	C	163	15.95	32.4	1741
Seo et al. [64]	ATmega128	C	163	8.42	5.6	618
Kargl et al. [58]	ATmega128	asm	163	6.1	11	-

Table 2.2: Comparison of existing ECC implementations over binary fields in software on 8-bit architectures. The presented values relate to one point multiplication.

2.2 Hardware/Software Codesign of ECC

In recent years it has been shown that custom hardware can increase the performance off ECC significantly with low cost and energy requirements [26]. This section gives a basic overview of different hardware/software methodologies available in literature. Thereby it has to be considered that the comparison of different hardware implementations is not straight forward, since different key sizes and technologies are used.
The hardware/software codesign approaches on 8-bit processors can be classified into three levels of granularity [66]:

- Implement hardware performing the operations for the point multiplication as described in [67]. The high amount of hardware leads to very fast execution times but requires much area and limits the flexibility of the system.
- Implement the field operations in hardware and execute them in a dedicated coprocessor.
- Extend the instruction set of the microprocessor to accelerate the field arithmetic.

Much research concerning the last two points has been made. Thus these implementations using these options are discussed more detailed below.

2.2.1 Coprocessors

Available publications concerning coprocessor designs for ECC target mainly software implementations based on the Atmel AVR or the Intel 8051 architecture. Table 2.3 summarizes the performance parameters of several approaches in literature.
A popular processor supporting the AVR instruction set is the AT95K offering 62 instructions. A fast coprocessor design for the AT95K processor is reported in [68] by Ernst et al. Their implementation is scalable supporting reconfigurable parameter sizes. The hardware accelerators execute field multiplication, addition and squaring.
A coprocessor for the 8051 architecture was presented by Aigner et al. in [69]. The authors implemented all field arithmetic operations including inversion in hardware and used affine coordinates.
Kumar et al. examined in [52] if standardized elliptic curves are feasible on RFID. They assumed that there is a need for a stand-alone hardware module to meet the timing requirements of standard protocols for RFID systems. In their ECC hardware implementation they recognized that the majority of area is required to store points and temporary variables. Their processor supports multiply, square and inversion instructions. They used affine coordinates and their implementation required about 10 kGE. They used an algorithm optimized for low memory requirements.
In some works the focus of the coprocessor design is on scalability. This means that the process operands can be changed to any size without the need of re-designing the implementation. The authors of [70] designed a highly scalable and unified multiplier architecture. They showed that it is possible to design dual-field arithmetic supporting both, prime and binary fields. Their hardware accelerator was able to speed-up an ECC prime field implementation based on the ARM processor by a factor of about 4.5 requiring 15 kGE area.

A limited scalable microprocessor was designed by Koschuch et al. in [66]. They developed a system where the parameter size is scalable in hardware and software. They proposed an efficient interface between the coprocessor and the host processor via DMA.

Implementation	*Processor*	*Length of scalar*	*Runtime [MCycles]*
Ernst et al. [68]	AT94K (AVR)	113	0.014
Kumar et al. [52]	AT94K (AVR)	163	0.452
Janssens et al. [67]	AT94K (AVR)	192	0.450
Aigner et al. [69]	SLE66CX (8051)	163	0.444
Koschuch et al. [66]	Dalton (8051)	163	1.19

Table 2.3: Comparison of existing ECC coprocessor implementations over binary fields on 8-bit architectures. The presented values relate to one point multiplication.

2.2.2 Instruction Set Extension (ISE)

Hardware coprocessors for public-key cryptography often lead to a big area. This is the reason why designers of ECC tried to achieve hardware acceleration with smaller area requirements. The instruction set of a general-purpose processor can be extended with a few application-specific instructions with low-silicon footprint. Especially the acceleration of the inner loop operations of ECC can result in significant performance gains [28].
In the years 1995 and 1998 the authors of [71] and [72] were the first presenting a special multiplication instruction for binary fields called *MULGF2*. This instruction performs a multiplication of two binary polynomials of a degree $(W-1)$, where W is the word size of a processor. Many authors use this instruction to accelerate ECC. Two years later Drescher presented the idea of an unified multiply instruction [73]. This instruction integrates the *MULGF2* instruction into an available integer multiplication instruction. They analyzed the implementation using short key lengths smaller than 8-bit with a Digital Signal Processor (DSP). Polynomial arithmetic was also integrated into the datapath of modulo multipliers of cryptographic co-processors in [74] and [70].
In 2003 the authors of [47] showed that the integration of binary multiplication in the datapath of an integer multiplier does not increases the critical path significantly. The silicon area of a unified multiplier is just marginally larger than that of a conventional multiplier. Additionally, they presented a squaring algorithm, which just uses the instructions *MULGF2* and *XOR*. Their approach achieved a fast execution time of 3 million cycles for one Montgomery multiplication with projective coordinates.
One year later Grossschaedl et al. presented five custom instructions for the 32-bit RISC architecture Microprocessor without Interlocked Pipeline Stages 32 (MIPS32) [75] to accelerate arithmetic operations in prime and binary fields. They described the algorithm selection process in detail and demonstrated that it is possible to find custom instructions in an efficient manner.
The authors of [76] showed that in 64-bit systems the *MULGF2* instruction can achieve a speed-up of up to 20. The result of the $MULGF2$ instruction is two bytes long. They categorized the possibilities of implementing the *MULGF2* instruction into three approaches:

- to store the higher and lower byte in parallel in two registers,
- to implement two separate instructions for the higher and the lower byte or
- to just process the lower byte.

Kumar and Paar presented a proof-of-concept implementation for a low-cost ISE on an 8-bit processor using reconfigurable logic [77]. They were able to offer an 8-bit microcontroller with full size ECC capabilities practical for low-cost applications. They implemented a standard-compliant 132-bit point multiplication over binary fields on the AVR microcontroller AT94K. Their extension is similar to a coprocessor consisting of a dedicated serial multiplier for binary field multiplications, which can be accessed through instructions. This instructions are multiple-cycle instructions operating on operands in memory. This approach requires about 5000 extra Configurable Logic Blocks (CLBs) and offers a runtime, which is 30 times faster than the software-only version.
In contrast to [77] the implementation described in [78] fully reuses the existing data path of an 8-bit microprocessor and uses instructions, which are consistent to the existing instruction set. The authors of [78] showed that an even faster ECC calculation as the prime field implementation in [56] is possible with binary fields and ISE. They were the first quantifying the performance of standard NIST and SECG curves on an 8-bit processor equipped with a uniform multiplier.
The authors of [26] pointed out how an ISE changes the algorithm design. First, they confirmed the fact that binary fields could be accelerated more than prime fields with an ISE. Second, they showed that the availability of an ISE can have an impact on the effectiveness of different algorithmic variants to implement the same binary field operation. Although in general the world level multiplication is much slower than the shift-and-add method, the availability of the *MULGF2* instruction makes the world level multiplication to be the faster approach.
Beside the *MULGF2* instruction, Shi et al. evaluated two additional instructions: shift by multiple bits and the determination of the Most Significant Bit (MSB) [4]. These instructions were able to achieve a speed-up of 3.9 compared to the pure software ECC implementation.
The above outlined implementations show that only a few additional instructions can lead to a significant improvement of the calculation of a ECC point multiplication.

2.3 Hardware Implementations of ECC for RFID

In terms of ECC hardware implementations, there are many publications available. However, only a few of them focus on low-resource designs for RFID [60]. Table 2.4 shows a comparison of different implementations for RFID.
In 2005, Wolkersdorfer tried to answer the question „Is Elliptic-Curve Cryptography suitable to secure RFID tags?“ [5]. He presented an ECC processor implemented in a full-precision architecture. The dual-field architecture supports prime and binary fields. This dual-field capability was achieved with almost no overhead. He showed that the power and area constrains of an RFID tag can be met, but also stated that there is room for improvement concerning the performance. The fastest implementation calculated a Montgomery multiplication in 0.8 seconds at a frequency of 545 kHz.

In 2003, Batina et al. published a survey of RSA and ECC hardware implementations [79]. They pointed out that there is room for improvement concerning ECC hardware on smart cards. Three years later they published an ECC design for RFID [80]. They presented binary field implementations and higher-layer authentication protocols based on the Schnorr and Okamoto scheme

Fürbass and Wolkersdorfer proposed an ECC processor with low die size for RFID authentication. The hardware calculates the Elliptic Curve Digital Signature Algorithm (ECDSA) over prime fields [9]. The implementation requires 23.6 kGE executing a point multiplication in 502 kCycles. Furthermore they considered several countermeasures against SPA and DPA.

One year later Lee et al. presented a processor including a tiny microcontroller for performing the Schnorr protocol for a tag authentication [7]. Their implementation is based on binary fields with a parametersize of 163. They designed an efficient modular operation algorithm with a little control overhead. Their fastest implementation performed an ECC point multiplication in 80 kCycles and required about 15 kGE area.

The same type of elliptic curve was used by Hein et al. [81] for the implementation of an ECC coprocessor.

A similar architecture was proposed by Bock et al. [8]. They used a challenge response protocol for a one way authentication based on the Diffie-Helmann key exchange and requires only one multiplication calculation on a tag. The point multiplication is realized with a slightly modified Montgomery multiplication. The modifications increased the resistance against SCAs concerning power analysis, differential power analysis and fault attacks.

The architecture of the ECC tag presented in [8] is derived from Infineons commercially available my-d$^{\text{TM}}$ light [82]. Bock et al. applied several techniques to reduce the area. The short-term storage was implemented using latches instead of flip-flops and the irreducible polynomial was hardwired. Furthermore, they optimized the number of required multiplexers and applied several measurements for energy reduction techniques, like clock gating. To enhance the performance different parallelization variants of the multiplication were evaluated. They reported that an 8-bit parallel multiplication in the arithmetic unit requires less energy than a degree of parallelization of four. Energy can be saved due to the short calculation times. The presented ECC engine is able to calculate one addition in one clock cycle and a multiplication in 41 clock cycles.

To keep the chip area small, Wenger et al. proposed a special microprocessor architecture for ECC [83]. They designed a custom architecture for a NIST elliptic curve over GF(p_{192}) and implemented the ECDSA. Although, their hardware is smaller than the architecture proposed in [8] their runtime is significantly higher requiring 1,377 kCycles execution time.

2.4 Comparison of Implementation Variants in Literature

Figure 2.1 shows a comparison of the different implementation variants of ECC presented in this chapter. All implementations are based on binary fields with a parameter size of 163. The fastest implementations available in literature are compared. One can see that a simple ISE as described in [78] can result in a significant performance improvement compared to a pure software implementation [58]. A coprocessor requires more area,

Implementation	*Area* [kGE]	*Runtime* [kCycles]	Finite field
Wolkersdorfer [5]	-	436	$GF(2^{191})$
Hein et al. [6]	13.7	306	$GF(2^{163})$
Lee et al. [7]	15.0	80	$GF(2^{163})$
Bock et al. [8]	16.2	47	$GF(2^{163})$
Fürbass et al.[9]	23.7	502	$GF(p_{192})$
Wenger et al. [83]	11.7	1,377	$GF(p_{192})$

Table 2.4: Comparison of existing ECC implementations in hardware. The presented runtime relates to one point multiplication.

but can again lead to a considerable speed-up of the execution time. A pure hardware implementation (like in [8]) is unmatched in terms of performance. However, in general such an approach requires the largest area.

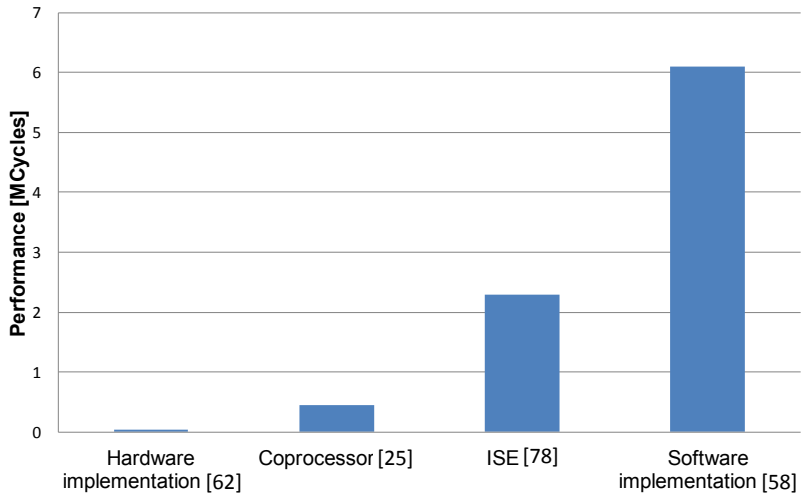


Figure 2.1: Comparison of implementation variants of 163-bit ECC in literature.

2.5 Summary

This chapter provides a survey of the related work. In particular, available software implementations of ECC on 8-bit processors are outlined. These software implementations mainly target the application of Wireless Sensor Networks. Only a few authors discussed the implementation of software-based ECC for RFID.
Many authors also studied hardware/software partitioning variants of ECC implementations. This chapter focused on literature describing hardware acceleration with instruction set extension and coprocessors.
Finally, it has been outlined that many works found in literature focus on custom designed hardware for ECC on RFID. The hardware implementations described in literature are

fast and energy efficient. Nevertheless, the high amount of special-purpose hardware introduces inflexibility.
It followed a presentation of several hardware acceleration methods. First, an innovative hardware/software partitioning approach was presented by introducing virtual addressing for ECC. Thereafter it has been shown, which additional instructions of the enhanced version of Ameba could accelerate the ECC calculation.
Finally, an implementation approach using a coprocessor was presented. It has been shown that the availability of a certain hardware influences the algorithm design and thus a new binary field multiplication algorithm was introduced. A possible way of realizing the communication with the coprocessor was outlined. Furthermore, a coprocessor architecture was proposed.

Chapter 3

Design

This chapter describes the concept of the ECC implementation. For an efficient implementation, it is important to consider the target microprocessor platform. Therefore, this chapter first presents the microprocessor Ameba. Then the chapter continues with a detailed description of the design considerations.
Figure 3.1 illustrates the procedure of this work. First, the ECC calculation is designed and optimized for software. Then hardware acceleration methods are identified. The implementation is first realized with the Ameba microprocessor and then adapted to the enhanced version of Ameba called Ameba2.

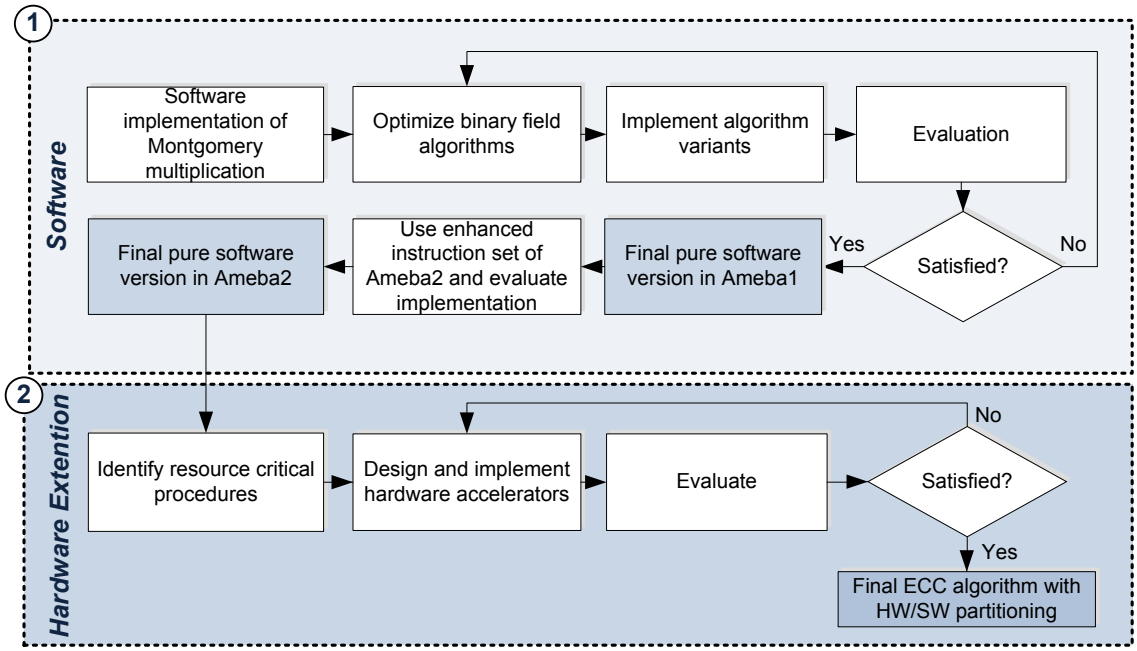


Figure 3.1: Hardware/software partitioning flow.

3.1 Special Purpose Microprocessor for RFID

Ameba is a special purpose microprocessor for RFID and was developed by Infineon Technologies. The purpose of Ameba is to offer an instruction set tailored for a typical RFID product. The processor has low requirements concerning area (about 3 kGE) and energy. The power consumption, while performing an AES encryption, is between 11.6 $\mu W/MHz$ and 25.9 $\mu W/MHz$. Compared to ad-hock state machines, the flexible and programmable framework supports a more efficient development of RFID products.

Architectural Description

The Ameba is based on the Harvard architecture with separate pathways for instructions and data. The processor can address up to 8 Kbytes of read-only instruction memory with 16 bit instruction paths. Ameba is realized with a single stage pipeline. All instructions, except memory accesses to the ROM, are executed in one clock cycle.
The Ameba supports 16 one-byte GPRs for temporal storage. According data accessing mode, Ameba can be classified as a load-and-store architecture.
The size of directly addressable data memory is 64 kBytes. For the addressing 8-bit data pathways are used. A special register called Data Pointer High (DPH) is used to hold the upper 8 bits of the address. When the RAM size is for example 512 bytes, the DPH has to be set to zero to address the lower half of the RAM. If the higher address range should be selected, then the DPH is set to one.
The Ameba also supports indirect accesses to the ROM. This could be used to implement LUTs (Look-up Tables). The position of the LUT in the ROM can be defined. This start-address has to be a multiple of 256. At every address of the LUT two bytes are stored. Instructions `MOVCH` and `MOVCL` are used to fetch either the higher or the lower byte.
Furthermore, the Ameba offers a Program Counter Stack (PCS) for subroutine instructions. When a `call` instruction is executed, the value of PC+1 is stored on the PCS. To return to this position the instruction `ret` is used. The execution then continues at the stored position on the stack.
Although the Ameba does not support interrupts, there is a halt signal to suspend the execution.

Instruction Set

The Ameba offers ALU operations with constant and variable operands, program branching and memory access with direct and indirect addressing. Table 3.1 shows a summary of the instruction set offered by the Ameba processor. Thereby, `#dataN` stands for an N-bit constant, `Rx` and `Ry` for the working registers `R0` to `R15` and `addrN` for a N-bit address. The symbol `CY` is used to indicate the carry bit. Although the processor includes a shift-left operation, there is no shift-right instruction. Furthermore, the Ameba does not include a multiplication operation.
For a better support of the symmetric cryptoalgorithm AES, an enhanced version of Ameba was designed by Infineon Technologies Austria AG. This version is called Ameba2 and includes additional instructions, which are listed in Table 3.2.

Instruction	*Description*
ADD Rx, #data8	Add immediate data and CY to register Rx
ADD Rx, Ry	Add register Ry and CY to register Rx
AND Rx, #data8	Logical AND of immediate data and register Rx
AND Rx, Ry	Logical AND of register Ry and CY to register Rx
CALL addr12	Call subroutine at addr12
JEQ addr12	Jump if CY is set
JMP addr12	Jump
JNE addr12	Jump if CY is not set
LD Rx, Ry	Load byte indirect addressed
LD Rx, addr8	Load byte
MOV Rx, #data8	Move immediate data to register Rx
MOV Rx, Ry	Move register Ry to register Rx
MOVCH Rx, Ry	Move code byte high
MOVCL Rx, Rx	Move code byte low
NOT Rx	Bitwise complement of register Rx
OR Rx, #data8	(Logical) OR of register Rx with immediate data to register Rx
OR Rx, Ry	(Logical) OR of register Rx with register Ry to register Rx
RET	Return from subroutine
ROL Rx	Rotate left
SEQB Rx, #data3	Set CY if bit Rx.n equals 1, n = #data3
SEQB Rx, Ry	Set CY if bit Rx.n equals 1, n = Ry[2:0]
SEQ Rx, #data8	Set CY if immediate data equals register Rx
SEQ Rx, Ry	Set CY if register Ry equals register Rx
SL Rx	Shift left
SLT Rx, #data8	Set CY to 1 if register Rx is less then immediate data
SLT Rx, Ry	Set CY to 1 if register Rx is less than register Ry
ST Rx, @Ry	Store byte indirect addressed
ST Rx, addr8	Store byte
SUB Rx, #data8	Subtract immediate data and CY from register Rx
SUB Rx, Ry	Subtract register Ry and CY from register Rx
XOR Rx, Ry	XOR of register Ry and register Rx

Table 3.1: Ameba Instruction Set.

Instruction	*Description*
SETB Rx, Ry	Set bit Ry of Rx
RLC Rx	Rotate Rx left with carry $Rx = \{Rx[6:0], CY\}, CY = Rx[7]$
RR Rx	Rotate Rx right, $Rx = \{Rx[0], Rx[7:1]\}$
RRC Rx	Rotate Rx left with carry $Rx = \{CY, Rx[7:1], CY = Rx[0]\}$
SR Rx	Shift Rx right
LDXR Rx, addr8	XOR of register Rx and direct addressed byte
LDXR Rx, @Ry	XOR of register Rx and indirect addressed byte

Table 3.2: Additional instructions of Ameba2.

3.2 ECC Design Decisions

Designing an ECC system includes decisions at different hierarchical levels (see Figure 3.2). The elliptic curve parameters and algorithms for field arithmetic, elliptic curve arithmetic and protocol arithmetic have to be selected.
The design decisions are influenced by

- security considerations,
- application platform (software, hardware or hardware/software codesign),
- constraints of the particular computing environment (processing speed, code size (ROM), memory size (RAM), gate count, power consumption, etc.) and
- constraints of the communications environment [41, p. 226].

An existing highly optimized hardware implementation developed at Infineon [8] provides the basis for the software implementation presented in this work. Ergo, the choice of the underlying field, protocol and point multiplication algorithm is adapted from [8].

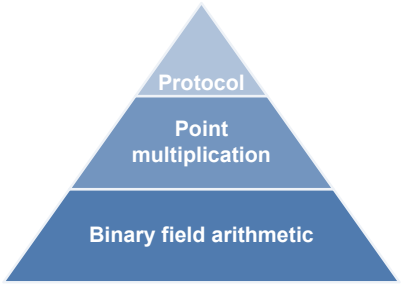


Figure 3.2: ECC design decisions. Adapted from [41, p. 226].

3.2.1 C Model

To evaluate different algorithm variants, a software model in C is written. The software models all hierarchical layers of the operations needed for authentication such as the protocol, the point multiplication and the field operations are implemented.
The data values are organized in arrays with 8-bit elements to emulate the target 8-bit architecture. The model serves as decision-support tool for the choice of the binary field operations implementation variants. Additionally, this program is used as a golden model to generate testvectors and ease the debugging.

3.2.2 Protocol

The chosen authentication protocol is a state-of-the art approach based on a Diffie-Hellman key exchange as described in [8] and illustrated in Figure 3.3. Each tag stores a random private key ξ_t and a certificate (x_T, s_t) consisting of the public key x_T and the signature s_T. The public key corresponds to the affine x-coordinate of the point $T = \xi_T \cdot P$. All RFID readers know the signature key. There is no need to store any additional secret

information in the reader.
To establish an authentication of the tag, the reader first picks a random number μ. Then the reader computes $A = \mu \cdot P$ and sends the affine x-coordinate x_A of the point A to the tag. The tag calculates $\xi_T \cdot x_A$ and sends the projective X- and Z-coordinate back to the reader. Additionally, the tag sends the certificate. The reader performs several checks with the received data and declares if the authentication of the tag was successful or not. The approach is very effective, since only one Montgomery multiplication with projective coordinates is required on the tag.
The presented work does not include the full protocol. Only the Montgomery multiplication is implemented. It is assumed that the challenge has already been received and is stored in RAM. The value of the challenge test vector will be initialized before starting the calculation. The results of the scalar multiplication are again written into the RAM. With the C model the correctness of these values is verified by checking if $X_c Z_b = X_b Z_C$. Both, the C model and the assembler implementation perform calculations with different values of the challenge and the private key.

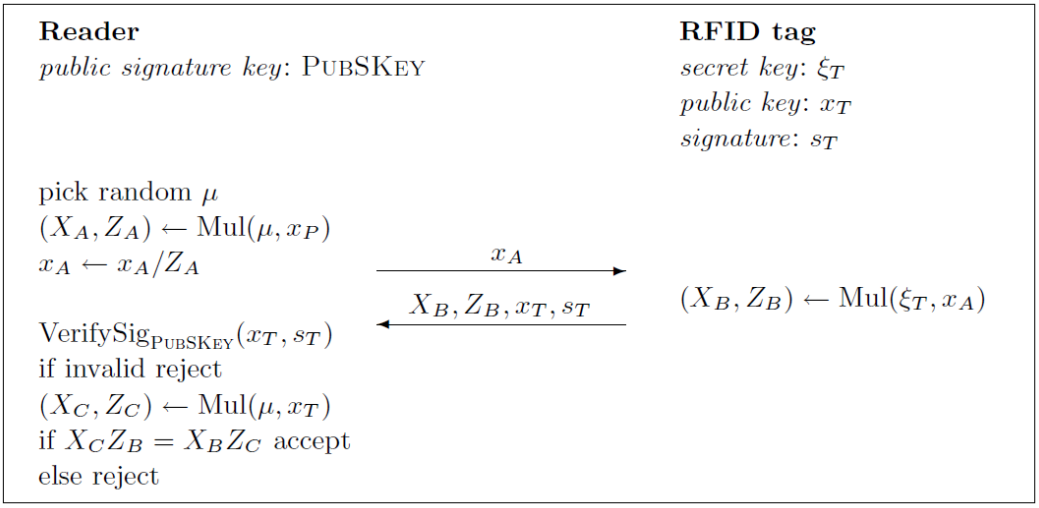


Figure 3.3: Chosen one-way authentication protocol [8].

3.2.3 Point Multiplication

To establish the authenticity of a tag, the tag has to calculate a point multiplication. Therefore, a Montgomery multiplication based on the approach described in [8] is used (see Algorithm 1). The algorithm includes several protections against SCAs. Both branches of the if-else statement require the same time for execution. Furthermore, several variables are initialized with a random number.
The generation of the random number is omitted in the presented work. The random variable is initialized with a constant value.

Algorithm 1: MONTGOMERY POINT MULTIPLICATION (for elliptic curves over $\mathbb{F}_{2^m}$).

Input: $k = (k_{t-1}, \ldots, k_1, k_0)_2$ with $k_{t-1} = 1$, $P = (x_p, y_p) \in E(\mathbb{F}_{2^m})$
Output: kP

pick random value r
$X_1 \leftarrow r, Z_1 \leftarrow 0, X_2 \leftarrow rx_p, Z_2 \leftarrow r.$
for $i \leftarrow t-2$ ***downto*** 0 **do**
    if $k_i = 1$ **then**
        $T \leftarrow Z_1, Z_1 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_1 \leftarrow xZ_1 + X_1 X_2 T Z_2.$
        $T \leftarrow X_2, X_2 \leftarrow X_2^4 + bZ_2^4, Z_2 \leftarrow T^2 Z_2^2.$
    else
        $T \leftarrow Z_2, Z_2 \leftarrow (X_1 Z_2 + X_2 Z_1)^2, X_2 \leftarrow xZ_2 + X_1 X_2 Z_1 T.$
        $T \leftarrow X_1, X_1 \leftarrow X_1^4 + bZ_1^4, Z_1 \leftarrow T^2 Z_1^2.$
if $\Delta(X_1, Z_1, X_2, Z_2, x_p) \neq 0$ **then return** *error* **else return** (X_1, Z_1)

3.3 Finite Field Operations

The performance of ECC is significantly determined by the field arithmetic [47]. This section presents the design considerations of the finite field algorithms.

3.3.1 Finite Field Choice

The choice of the underlying finite field is one of the most important decisions when designing an ECC system [47]. Basically there are two options defined in standards: prime fields $GF(p)$ or binary fields $GF(2^m)$ [56].
The hardware support of certain processors to calculate integer multiplication favours the usage of prime fields, as outlined by [41, 56, 59]. However, Seo et al. demonstrated that software-based ECC can achieve better performance results, if binary fields are used. The Ameba does not feature a hardware multiplier. Furthermore, the usage of binary fields eases future hardware accelerations [41, 47]. As a consequence, binary fields $GF(2^m)$ are chosen.
There are two options for the binary field basis: the normal and the polynomial basis. The main advantage of normal basis representation is that the squaring is a linear operation. The squaring can be implemented by a circular shift of the coefficients [84]. The disadvantage of normal basis is that the multiplication is more complicated. Since an efficient implementation of the multiplication is essential for the overall performance, the polynomial basis representation is used in this work.

3.3.2 Binary Field Operations

There are many different ways to implement each binary field operation [4]. This section reviews a selection of candidate methods and introduces algorithm variants.
To calculate the Montgomery point multiplication the following binary field operations are required:

- Addition/subtraction $c(z) = a(z) \oplus b(z)$
- Reduction $c(z) = c(z)\ mod\ p(z)$
- Multiplication $c(z) = a(z) \cdot b(z)\ mod\ p(z)$
- Squaring $c(z) = a(z)^2\ mod\ p(z)$

Another binary field operation is the inversion $c(z) = a(z)^{-1}\ mod\ p(z)$. The Montgomery multiplication operates with projective coordinates. To recover the affine coordinates out of the projective coordinates would require an inversion. However, the authentication protocol does not need to recover affine coordinates on the tag. Consequently, the expensive inversion can be omitted and will not be discussed further.
For the estimation of the runtime of algorithms, the variable $t_{[instruction]}$ presents the time required to execute an instruction.

3.3.3 Binary Field Element Representation

An element in the field $GF(2^m)$ is represented in polynomial basis in the form

$$a(z) = a_{m-1}z^{m-1} + ... + a_2z^2 + a_1z + a_0, a_i \in \{0, 1\}\ [6].$$

The degree of a polynomial $a(z)$ is written as $deg\{a(z)\}$ and is the power of the highest order term in $a(z)$. Thus the maximum degree of a polynomial in the field $GF(2^m)$ is $m - 1$. Consequently, the associated binary vector $a = (a_{m-1}, ..., a_2, a_1, a_0)$ of the polynomial $a(z)$ has the length m. The calculation $z^k a(z)$ means shifting the vector k-times left. The degree determines how many bits are required to store the element. Hence, in a W-bit architecture one element can be stored in an array A with t words, where $t = \lceil m/W \rceil$ [41, p. 47]. The chosen degree of the elliptic curves in this work is 163, thus $t = \lceil 163/8 \rceil = 21$ words are required to store one polynomial field element.
In this thesis, a common notation found in many programming languages is used: $A[i]$ stands for the i^{th} word of the array storing the vector representation of $a(z)$. As illustrated in Figure 3.4 the coefficients are stored in descending order, thus the rightmost bit of $A[20]$ stores a_0 and the leftmost unused bits of $A[0]$ are always set to zero.

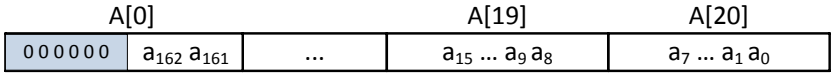


Figure 3.4: Representation of $a \in GF(2^{163})$ as an array A of 8-bit words. Adapted from [41, p. 47].

3.3.4 Polynomial Addition

Adding elements in $GF(2^m)$ can be obtained by adding the polynomials and reducing the coefficients modulo two [41, p. 27]. For example:

$$a(z) = z^3 + z^2 + 1,\ b(z) = z^2 + z + 1$$
$$c(z) = a(z) + b(z) = z^3 + 2z^2 + z + 2\ mod(2) = z^3 + z$$

Addition modulo two can be done with a bitwise Exclusive Or (XOR) operation. Consequently, the addition above would be calculated in vector representation as follows:

$$\begin{aligned} a &= 1101 \\ \underline{b &= 0111} \\ c &= 1010 \end{aligned}$$

Note, that the addition and subtraction of two elements is the same in binary field arithmetic. Since the operation needs no carry propagation, it can be implemented with a word-wise XOR function.

3.3.5 Polynomial Reduction

The results of multiplication and squaring operations are elements with a maximum degree of $(2m-2)$. This means that the results have twice the length of a standard element. Throughout this text such an element is called "double element".
The representation of a double element is shown in Figure 3.5. The lower order terms are stored in $A[2t-1]$ and the higher order terms in $A[0]$.

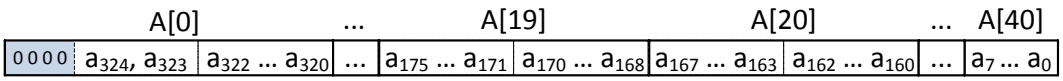


Figure 3.5: Representation of $a \in GF(2^{2m-1})$, where $m = 163$ as an array A of 8-bit words.

Mathematical Background

To reduce the double elements, modular reduction $a(z) = a_d(z)\ mod\ p(z)$ is used. The irreducible polynomial $p(z)$ is defined by the domain parameters.
From a mathematical point of view, the result of the modulo calculation is the remainder of a division by $p(z)$ [42, p. 58–64]. The degree of the remainder is always less than the prime polynomial. The following example illustrates the paper and pencil method of a polynomial division:
The polynomial $a(z) = z^{164} + z^{163} + z^6 + z$ is reduced using the NIST prime polynomial $p(z) = z^{163} + z^7 + z^5 + 1$ as follows

$$\begin{array}{lll} (z^{164} + z^{163} + z^6 + 1) & : & (z^{163} + z^7 + z^5 + 1) = z + 1 \\ \underline{z^{164} + z^8 + z^6 + z} & & \\ z^{163} + z^8 + z + 1 & & \\ \underline{z^{163} + z^7 + z^5 + 1} & & \\ z^8 + z^7 + z^5 + z & & \end{array}$$

First, the difference n between the degrees of the divisor and prime polynomial is calculated. Then, the divisor is multiplied with z^n. In a binary vector representation this

multiplication is achieved by shifting the vector n times left. The result of the multiplication is subtracted from the dividend, which cancels out the highest order term.
The next step is to check the degree of the intermediate result and repeating the previous steps. This is done until the intermediate result has a degree less than the degree of the divisor. The last calculated subtraction is the remainder.
The ideas of the reduction using a fixed prime polynomial described in [41, p. 53–54; 231–232] form the basis of the reduction algorithms, which are presented below. For an efficient implementation, the following observation is used:
If $p(z) = p_m z^m + r(z)$, then any polynomial $a_d(z)$ with $deg\{a_d(z)\} \leq 2m - 2$ can be expressed as

$$\begin{aligned} a_d(z) &= a_{2m-2}z^{2m-2} + ... + a_1 z + a_0 \\ &\equiv (a_{2m-2}z^{m-2} + ... + a_m)r(z) + a_{m-1}z^{m-1} + ... + a_1 z + a_0 \ mod\ p(z) \end{aligned} \tag{3.1}$$

Hankerson et al. propose to precompute the polynomials $z^k r(z)$, where $0 \leq k \leq W-1$ [41, p. 53]. The shifts $z^j r(z)$, where $j \geq 8$, can be achieved with array indexing. Most ECC parameters define $p(z)$ so that $r(z)$ is a low-degree polynomial, which leads to small memory requirements [43]. In this work $p(z)$ is specified so that two bytes can represent every shift of $r(z)$ denoted by $z^k r(z)$, where $0 \leq k \leq 7$. R denotes the vector representation of $r(z)$ and $R[k]$ the k^{th} shift of $r(z)$. The higher and the lower byte of $R[k]$ are indicated by $Rh[k]$ and $Rl[k]$.

Interleaved vs. Stand-alone Reduction

In general, there are two methods to perform the reduction [6]:

- Interleaved reduction: Reduce the intermediate results of a multiplication or squaring
- Stand-alone reduction: Reduce only the final result of a multiplication or squaring

The main advantage of the first option is that the length of the intermediate results is limited to one single element. Hence, intermediate result can be stored in 21 bytes of RAM.
Stand-alone reduction reduces only the final result of multiplication and squaring. This result is a double element and requires 41 bytes storage.

Bitwise Reduction

One state-of-the art reduction method is the bitwise reduction [41]. The higher terms of $a(z)$ are eliminated by processing all coefficients a_i, where $i \geq 163$. According to Equation 3.1 the following calculation has to be performed if a_i is set [43]:

$$a(z) = a(z) + r(z) \cdot z^{i-m}$$

This is achieved by shifting $r(z)$ $(i - m)$ bits left and adding the result to $a(z)$. It is assumed, that the values $R[0]$, $R[1]$,...,$R[7]$ are precalculated. Shifting a multiple of eight times, can be achieved by changing the indexes of the array when performing the addition. Figure 3.6 illustrates the calculation of $n = (i-m)$ for several bits of an array A if $W = 8$ and $m = 163$.

	...	A[18]								A[19]									A[20]	...
Array A of coefficiants a_i	...	a_{179}	a_{178}	a_{177}	a_{176}	a_{175}	a_{174}	a_{173}	a_{172}	a_{171}	a_{170}	a_{169}	a_{168}	a_{167}	a_{166}	a_{165}	a_{164}	a_{163}	a_{162} ...	
Position k of bit in word	...	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	...	
n=i-deg{$p(z)$}	...	2•8	8+7	8+6	8+5	8+4	8+3	8+2	8+1	8	7	6	5	4	3	2	1	0	...	

Figure 3.6: Calculation of n, where $n = i - deg\{p(z)\} = i - 163$ to reduce the bit a_i by adding $z^n r(z)$ to $a(z)$.

For example, if bit a_{170} is set, then $n = 170 - 163 = 7$ and thus $R[7]$ is added to A. More precisely, this means that $Rl[7]$ is added to $A[40]$ and $Rh[7]$ is added to $A[39]$. When considering the next bit a_{171}, then $n = 171 - 163 = 8$. The required shifting is done with indexing. This means that, $Rh[0]$ is added to $A[39]$ and $Rl[0]$ to $A[38]$.
It is assumed that the values are equally distributed and thus every second bit is one. The control overhead is estimated to be 20%. Thus the calculation time is estimated as follows: $t_{red\_bitwise} = [21t_{LD} + 163 \cdot (t_{SEQB} + t_{JNE} + \frac{1}{2}(2t_{LD} + 2t_{XOR} + 2t_{ST}))]1.2 \approx 1,005$ cycles

Bytewise Reduction

To achieve a faster runtime, the author of this work proposes an approach to perform the reduction bytewise.
When looking at an illustration of the reduction procedure described above (see Figure 3.7), one can observe that it is necessary to change three bytes of A to reduce one byte.

More specifically, for the reduction of one byte $A[i]$, where $i \leq 19$, which is presented by the bits $\{a_7, ..., a_1, a_0\}$, it is necessary to perform the following calculations:

$$A[i+21] \leftarrow A[i+21] \oplus R_0(a_2, a_1, a_0) \tag{3.2}$$
$$A[i+20] \leftarrow A[i+20] \oplus R_1(a_2, a_1, a_0) \oplus R_2(a_7, ..., a_4, a_3) \tag{3.3}$$
$$A[i+19] \leftarrow A[i+19] \oplus R_3(a_7, ..., a_4, a_3) \tag{3.4}$$

The functions R_0 and R_1 depend on least significant three bits of the byte $A[i]$. The value of the other two functions R_2 and R_3 is determined with the most significant five bits of $A[i]$.

The values of R_0, R_1 and R_2 can be determined, as stated below:

$$\begin{aligned} R_0(a_2, a_1, a_0) &= a_2 \cdot R_l[7] \oplus a_1 \cdot R_l[6] \oplus a_0 \cdot R_l[5] \\ R_1(a_2, a_1, a_0) &= a_2 \cdot R_h[7] \oplus a_1 \cdot R_h[6] \oplus a_0 \cdot R_h[5] \\ R_2(a_7, ..., a_4, a_3) &= a_7 \cdot R_l[4] \oplus a_6 \cdot R_l[3] \oplus a_5 \cdot R_l[2] \oplus a_4 \cdot R_l[1] \oplus a_3 \cdot R_l[0] \\ R_3(a_7, ..., a_4, a_3) &= a_7 \cdot R_h[4] \oplus a_6 \cdot R_h[3] \oplus a_5 \cdot R_h[2] \oplus a_4 \cdot R_h[1] \oplus a_3 \cdot R_h[0] \end{aligned} \tag{3.5}$$

The used $r(z)$ is defined so that the value of $R_3(a_7, ..., a_4, a_3)$ always equals to $\{a_7, ..., a_4, a_3\}$. There are several options how to determine R_0, R_1 and R_2. They could be either calculated at runtime or implemented as a Look Up Table (LUT) in Read Only Memory (ROM). Different implementation variants are discussed below.

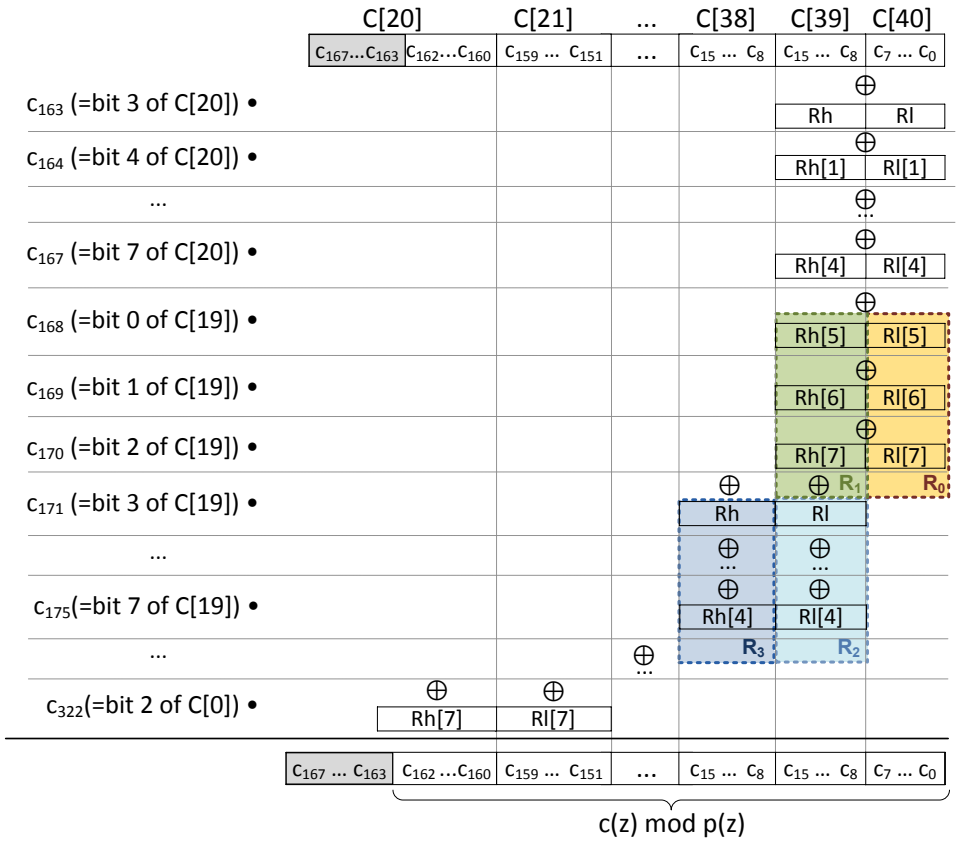


Figure 3.7: Illustration of required additions for bytewise reduction.

Variant 1 One option is to store the different possible values of R_0, R_1 and R_2 in LUTs. Table *red_low_lut* could store R_0 in the lower bytes and R_1 in the higher bytes. The values of the table are addressed by the bits $\{a_2, a_1, a_0\}$.
Additionally, the different values of R_2 could be stored in a separate table *red_high_lut*, which is addressed with the five bits $\{a_7, ..., a_4, a_3\}$. This address can be calculated by shifting $C[i]$ three bits to the right, which corresponds to R_3. Algorithm 2 illustrates this procedure.

Algorithm 2: Reduction using two LUTs.

for $i \leftarrow 0$ ***to*** 19 **do**
  $l \leftarrow A[i]$ & $0x07$
  $h \leftarrow A[i] \gg 3$
  $A[21+i] \leftarrow A[21+i] \oplus red\_low\_lut(l)_{highbyte}$
  $A[20+i] \leftarrow A[20+i] \oplus red\_low\_lut(l)_{lowbyte}$
  $A[20+i] \leftarrow A[20+i] \oplus red\_high\_lut(h)_{highbyte}$
  $A[19+i] \leftarrow A[19+i] \oplus h$

Ameba does not support a shift-right instruction, thus line 3 is realized with five rotate left instruction and clearing the upper three bits with an AND. The runtime of Algorithm 2 is estimated as follows:

$$\begin{aligned} t_{lines2,3} &= t_{LD} + 2t_{AND} + 5t_{ROL} = 8\ cycles \\ t_{lines4,5,6} &= 3 \cdot (t_{LD} + t_{MOVCH/MOVCL} + t_{XOR} + t_{ST}) = 15\ cycles \\ t_{line7} &= t_{LD} + t_{XOR} + t_{ST} = 3\ cycles \\ t_{red\_v1} &= 20 \cdot (t_{lines2,3} + t_{lines4,5,6} + t_{lines8...1}) + t_{lines8...11} \approx 520\ cycles \end{aligned} \quad (3.6)$$

The overhead for pointer calculation is approximated by $t_{pointer} = 100\ cycles$. Inclusive control overhead, the estimated execution time of this variant is

$$t_{algo\_v1_{total}} = (t_{algo2} + t_{pointer}) \cdot 1.2 \approx 745\ cycles \quad (3.7)$$

The ROM size required for each half of the LUT red_low_lut is 2^3 bytes. The table red_high_lut requires 2^5 bytes. Hence, the total LUT size is 48 bytes.

Variant 2 Another option is to combine R_1 and R_2 in a new table called red_r4_lut that stores

$$R_4(a_7, ..., a_1, a_0) = R_1(a_3, a_2, a_1) \oplus R_2(a_7, ..., a_4, a_3). \quad (3.8)$$

This table stores 2^8 possible values of R_4, which are addressed by the bytes of A. This means that lines 5 and 6 of Algorithm 2 could be replaced by

$$A[20 + i] \leftarrow A[20 + i] \oplus red\_r4\_lut(l)_{lowbyte}. \quad (3.9)$$

The execution time of $t_{lines4,5,6}$ can be estimated as

$$t_{lines4,5,6} = 2 \cdot (t_{LD} + t_{MOVCH/MOVCL} + t_{XOR} + t_{ST}) = 10\ cycles. \quad (3.10)$$

These lines have been accelerated by five clock cycles and are executed 20 times. Thus the total runtime is reduced by 100 cycles. Considering the control overhead, the runtime of this variant is estimated as follows:

$$t_{algo\_v2_{total}} = (t_{red\_v1} - 100\ cycles + t_{pointer}) \cdot 1.2 \approx 625\ cycles \quad (3.11)$$

The lower bytes of the table red_low_lut and the table red_r4_lut require $2^8 = 256$ bytes. To sum it up, 264 bytes for the LUTs are necessary for this reduction variant.

Variant 3 The fastest variant is to determine all values of R_0, R_4 and R_3 with LUTs. Each table requires 256 bytes. Thus, 768 bytes have to be stored in ROM. The required R_0, R_3 and R_4 can be looked up in tables without the need to manipulate the current processed byte of A. Thus, the lines 2 to 7 of Algorithm 2 could be replaced by the following calculations:

$$\begin{aligned} a &\leftarrow A[i] \\ A[21 + i] &\leftarrow A[21 + i] \oplus red\_r0\_lut(a) \\ A[20 + i] &\leftarrow A[20 + i] \oplus red\_r4\_lut(a) \\ A[19 + i] &\leftarrow A[19 + i] \oplus red\_r3\_lut(a) \end{aligned} \quad (3.12)$$

The following runtime is expected

$$t_{xorLUT} = t_{LD} + 3(t_{LD} + t_{MOVCH/MOVCL} + t_{XOR} + t_{ST}) = 16\ cycles. \tag{3.13}$$

The total runtime of this variant is approximately:

$$t_{red\_v3} = 20 \cdot (t_{xorLUT}) \approx 320\ cycles \tag{3.14}$$

Considering the control overhead and pointer calculations, this results in the following estimated runtime:

$$t_{red\_v3_{total}} = (t_{red\_v3} + t_{pointer}) \cdot 1.2 \approx 505\ cycles \tag{3.15}$$

Comparison of Reduction Methods

Both time and storage requirements have to be considered when choosing the reduction algorithm. Looking at Table 3.3, the word-wise reduction variant 2 offers a good trade-off and is thus chosen for implementation.

Method	*Time [Cycles]*	*LUT [Byte]*
Bitwise	1,005	0
Wordwise variant 1	745	48
Wordwise variant 2	625	264
Wordwise variant 3	505	768

Table 3.3: Comparison of stand-alone reduction methods regarding approximated execution time and ROM storage requirements.

3.3.6 Polynomial Multiplication

Multiplication accounts for the majority of runtime and should be implemented in an efficient way [64].
The multiplication of two polynomials $a(z)$ and $b(z)$ can be written as

$$c(z) = a(z) \cdot b(z) = a_{m-1}z^{m-1}b(z) + ... + a_2z^2b(z) + a_1zb(z) + a_0b(z)\ \text{[41, p. 48]}.$$

The most straight-forward method for $GF(2^m)$ multiplication is the *shift-and-add* algorithm [43]. To multiply two polynomials $a(z)$ and $b(z)$, the product $c(z)$ is first set to zero if $a_0 = 0$ or to $b(z)$ if $a_0 = 1$. Then the algorithm scans the bits of $a(z)$ from a_1 to a_{m-1}. Whenever a bit of $a(z)$ is processed, $b(z)$ is shifted to the left by one bit. If the scanned bit in $a(z)$ is one, the new value of $b(z)$ is added to the product $c(z)$. Typically, a hardware implementation can shift the whole element in one clock cycle. Thus this method is well-suited for a hardware implementation. A software implementation requires many memory accesses to perform the shift. This results in a poor performance. Therefore, this section presents more efficient methods for a polynomial multiplication in software.

Left-to-right multiplication

The left-to-right (l-t-r) multiplication is based on the observation that if $z^k b(z)$ has been computed for $k \in [0, W-1]$, then $z^{Wj+k} b(z)$ can be determined by appending j zero words to the right of the vector $z^k b(z)$ [41, p. 53–54]. This observation is used to reduce the number of required shifts from $(m-1)$ to $(W-1)$.
The approach is shown in Algorithm 3. Unlike the shift-and-add algorithm, which scans the bits in $a(z)$ one by one sequentially, the l-t-r algorithm first tests the most significant bit of all the words in $a(z)$ from $A[t-1]$ to $A[0]$. If the scanned bit is one, the shifted value of $b(z)$ is added to $c(z)$. Note, that $c(z) + b(z) \cdot z^{jW}$ can be performed by aligning $b(z)$ with proper words in $c(z)$. Then the product $c(z)$ is shifted left by one bit. Then the second most significant bit is processed, and so on.

Algorithm 3: Left-to-right polynomial multiplication. Adapted from [41, p.50].

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$

```
1 C ← 0
2 for k ← 7 to 0 do
3     for j ← t − 1 to 0 do
4         if k^th bit of A[j] is 1 then
5             for i ← 0 to t − 1 do  C[i + j] ← C[i + j] ⊕ B[i]
6     if k ≠ 0 then  C ← C · z
```

Runtime estimation The first step in Algorithm 3 is to initialize all words of $c(z)$ to zero and requires $t_{resetC} = 41 \cdot t_{ST} = 41$ cycles.
The check in line 4 can be achieved by the following steps:

- load $A[j]$
- check if the bit is set with a $SEQB$ instruction
- skip the four-loop with a JNE instruction if the bit is not set

Hence, inclusive pointer addition the runtime for this line can be estimated as follows:

$$t_{checkbit} = t_{LD} + t_{SEQB} + t_{JNE} + t_{ADD} = 5 \text{ cycles.}$$

The addition in line 5 requires $t_{wordadd} = 2t_{LD} + t_{XR} + t_{ST} + 2t_{ADD} = 6$ cycles.
The shift in line 6 could be implemented in Ameba as follows:

```
LD   R1, address of C[n]
SEQB R2, #7
MOV  R2, R1
SL   R1
ADD  R1, #0 (add carry)
ST   R1, address of C[n]
```

Therefore, the runtime of shifting a double element is approximately:

$$t_{shiftC} = 41 \cdot (t_{LD} + t_{SEQB} + t_{MOV} + t_{SL} + t_{ADD} + t_{ST}) \approx 250 \text{ cycles.}$$

Assuming that every second bit is one and 20% control overhead, the runtime of this approach is approximated as follows:

$$t_{l-t-r} = (t_{resetC} + 8 \cdot 21(t_{checkbit} + \tfrac{1}{2} \cdot 21 \cdot t_{wordadd}) + 7 \cdot t_{shiftC}) \cdot 1.2 \approx 15,830 \text{ cycles.}$$

The result is 41 byte long and has to be reduced with a stand-alone-reduction algorithm. Hence, the total runtime is approximately

$$t_{l-t-r_{total}} = t_{l-t-r} + t_{red\_v2_{total}} \approx 16,455 \text{ cycles.}$$

Left-to-right Multiplication with Windows

At the expense of some storage overhead, the l-t-r multiplication method can be accelerated [85]. The multiplication $u(z) \cdot b(z)$ for all polynomials $u(z)$ of degree less than the window size W is calculated (see Algorithm 4) [41, p.50-51].
Then the algorithm steps through all words of $a(z)$. If a window size of w is chosen, the windowing method processes w bits of every word at once. The processed w bits of a word of A represent the coefficients of $u(z)$. According to this $u(z)$, the appropriate B_u is added to C. After processing the first window of every word, the result is shifted left by w bits. This procedure is repeated until all windows are processed.
The choice of w comes with a trade-off between memory requirements and performance. The number of precalculated elements is $2^w - 1$. A small w leads to low RAM requirements, but has a negative impact on the performance, since the loop at line 3 of Algorithm 4 is executed more often.

Algorithm 4: Left-to-right multiplication with windows of width w [41, p.50].

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m - 1$
Output: $c(z) = a(z) \cdot b(z)$
1 $C \leftarrow 0$
2 Compute $B_u = u(z) \cdot b(z)$ for all polynomials $u(z)$ of degree at most $w - 1$
3 **for** $k \leftarrow (W/w) - 1$ ***to*** 0 **do**
4 **for** $j \leftarrow t - 1$ ***downto*** 0 **do**
5 Let $u = (u_{w-1}, ..., u_1, u_0)$, where u_i is bit $(wk + i)$ of $A[j]$
6 **for** $i \leftarrow 0$ ***to*** $t - 1$ **do** $C[i + j] \leftarrow C[i + j] \oplus B_u[i]$
7 **if** $k \neq 0$ **then** $C \leftarrow z^w \cdot C$

Example: An example of a l-t-r multiplication with $w = 4$ is presented below. The vector of $a(z)$ is shown in Figure 3.8. The factor $b(z)$ can take any value in $GF(2^{163})$.

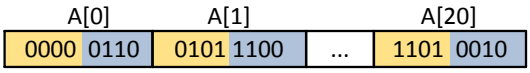


Figure 3.8: Value of factor $a(z)$ for l-t-r multiplication with windows example.

In the precalculation phase 15 elements are calculated and stored in RAM. Table 3.4 shows the polynomial, binary vector and integer representation of the precalculated elements.

Precalculated Element b_u	*Representation*		
	Polynomial	*Binary vector*	*Integer*
b_1	$b(z)$	$0001 \cdot b$	$1 \cdot b$
b_2	$z \cdot b(z)$	$0010 \cdot b$	$2 \cdot b$
b_3	$(z+1) \cdot b(z)$	$0011 \cdot b$	$3 \cdot b$
b_4	$z^2 \cdot b(z)$	$0100 \cdot b$	$4 \cdot b$
b_5	$(z^2+1) \cdot b(z)$	$0101 \cdot b$	$5 \cdot b$
...			
b_{15}	$(z^3+z^2+z+1) \cdot b(z)$	$1111 \cdot b$	$15 \cdot b$

Table 3.4: Precalculated elements of l-t-r multiplication with windows example.

After the precalculation, four bits of every word determine u. The words are processed as follows:

- The outer loop in line 3 of Algorithm 4 starts with $k = 1$:
  - Process $A[0] \Rightarrow u = 0000 \Rightarrow$ continue
  - Process $A[1] \Rightarrow u = 0101 \Rightarrow$ add $(B_5[0], B_5[1], ..., B_5[20])$ to $(C[19], C[20], ..., C[40])$
  - ...
  - Process $A[20] \Rightarrow u = 1101 \Rightarrow$ add $(B_{13}[0], B_{13}[1], ..., B_{13}[20])$ to $(C[0], C[1], ..., C[20])$
- Shift all words of C four times left (see line 7 of Algorithm 4)
- Repeat the loop with $k = 0$:
  - Process $A[0] \Rightarrow u = 0110 \Rightarrow$ add $(B_6[0], B_6[1], ..., B_6[20])$ to $(C[20], C[21], ..., C[41])$
  - Process $A[1] \Rightarrow u = 1100 \Rightarrow$ add $(B_{12}[0], B_{12}[1], ..., B_{12}[20])$ to $(C[19], C[20], ..., C[40])$
  - ...
  - Process $A[20] \Rightarrow u = 0010 \Rightarrow$ add $(B_2[0], B_2[1], ..., B_2[20])$ to $(C[0], C[1], ..., C[20])$

Multiplication with Windows of Width w=2 The l-t-r multiplication with windows of width $w = 2$ is illustrated in Algorithm 5. Three 21-byte elements (B_1, B_2 and B_3) have to be precalculated and stored in RAM. This means that, including the storage requirement for the result, 104 bytes of RAM is needed. Thereafter, the algorithm steps through all words of $a(z)$ and processes two bits each time.

The runtime of the algorithm can be estimated as described below.
Copy one element to another (see line 3 of Algorithm 5) requires about

$$t_{copy} = 21(t_{LD} + t_{ST} + 2t_{ADD}) = 84 \text{ cycles}$$

Algorithm 5: Left-to-right multiplication with windows of width $w = 2$.

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m - 1$
Output: $c(z) = a(z) \cdot b(z)$

```
1  C ← 0
2  B_0 ← 0
3  B_1 ← B
4  B_2 ← B · z
5  B_3 ← B_2 ⊕ B_1
6  for d ← 3 downto 0 do
7      for j ← t − 1 downto 0 do
8          switch d do
9              case 0   u ← A[j] & 0x03
10             case 1   u ← (A[j] & 0x0C) ≫ 2
11             case 2   u ← (A[j] & 0x30) ≫ 4
12             case 3   u ← (A[j] & 0xC0) ≫ 6
13         for i ← 0 to t − 1 do
14             C[i + j] ← C[i + j] ⊕ B_u[i]
15     if d ≠ 0 then
16         for j ← 0 to 1 do
17             C ← z^2 · C
                                          // Shift C ≪ 2
```

The shift operation in line 4 operates on 21 bytes and can be approximated with t_{shift} as follows:

$$t_{shift} = t_{shiftC}/2 = 125 \text{ cycles.}$$

The addition in line 4 is a simple word-wise addition and thus the whole precalcuation requires approximately:

$$t_{precalc} = t_{copy} + t_{shiftC}/2 + 21 \cdot t_{wordadd} = 335 \text{ cycles.}$$

Again, the shift-right operations in lines 11 to 12 have to be realized with `ROL` instructions. These runtimes are estimated as follows:

$$\begin{aligned} t_{line9} &= t_{LD} + t_{AND} = 2 \text{ cycles.} \\ t_{line10} &= t_{LD} + t_{AND} + 6t_{ROL} = 8 \text{ cycles.} \\ t_{line11} &= t_{LD} + t_{AND} + 4t_{ROL} = 6 \text{ cycles.} \\ t_{line12} &= t_{LD} + t_{AND} + 2t_{ROL} = 4 \text{ cycles.} \end{aligned}$$

The runtime including 20 % control overhead is approximately:

$$\begin{aligned} t_{ltr\_w2} &= [t_{resetC} + t_{precalc} + 21 \cdot (t_{line9} + t_{line10} + t_{line11} + t_{line12} + 4 \cdot 21 \cdot t_{wordadd}) + 3 \cdot t_{shiftC}] \cdot 1.2 \\ &\approx 14,555 \text{ cycles.} \end{aligned}$$

The total runtime inclusive stand-alone reduction can be estimated as follows:

$$t_{ltr\_w2\_total} = t_{ltr\_w2} + t_{red\_v2_{total}} = 15,180 \text{ cycles.}$$

The runtime approximation shows that a speed-up compared to the l-t-r method without windows can be achieved.

Multiplication with Windows of Width w=4 The performance can be improved further by increasing the window with to $w = 4$. Then the algorithm steps through all words of $a(z)$ only twice, but requires 15 precalculated elements. Thus, 315 bytes of RAM are needed.
Algorithm 6 shows the determination of all B_u, where u is even. The remaining B_u can be calculated as shown in Algorithm 7. The remaining procedure is the same as in Algorithm 4.

Algorithm 6: Precalculation of even $B_u = u(z)$ for window multiplication with width $w = 4$.

Input: Binary polynomial $b(z)$
Output: $b_u = b(z) \cdot u(z)$ for all $u(z)$
with even degree lower than 4

1 $B_1 \leftarrow B$
2 $B_2 \leftarrow B \cdot z$
3 $B_4 \leftarrow B_2 \cdot z$
4 $B_6 \leftarrow B_2 \oplus B_4$
5 $B_8 \leftarrow B_4 \cdot z$
6 $B_{10} \leftarrow B_8 \oplus B_2$
7 $B_{12} \leftarrow B_8 \oplus B_4$
8 $B_{14} \leftarrow B_{12} \oplus B_2$

Algorithm 7: Precalculation of odd $B_u = u(z)$ for window multiplication with width $w = 4$.

Input: Binary polynomial $b(z)$
Output: $b_u = b(z) \cdot u(z)$ for all $u(z)$
with even degree lower than 4

1 **for** $u \leftarrow 1$ ***to*** 7 **do**
2 $\quad B_{2u+1} \leftarrow B_{2u} \oplus B_1$

The runtime of the precalculation is estimated as follows:

$$t_{precalc_{even}} = 4 \cdot 21 \cdot t_{wordadd} + 3 \cdot t_{shift} \approx 950 \text{ cycles}$$

$$t_{precalc_{odd}} = 7\dot{2}1 \cdot t_{wordadd} \approx 880 \text{ cycles.}$$

The shift in line 6 of Algorithm 4, where $w = 4$, is achieved by shifting every byte of C four bits left. With Ameba this is realized by executing a rotate-left instruction four times. Then the upper nibble of the byte already represent the right values and lower nibble equals the carry to the next byte. The estimated runtime of the shift operation overhead is

$$t_{shift4C} = 41 \cdot (t_{LD} + 4t_{ROL} + 2t_{MOV} + 2t_{AND} + t_{OR} + t_{ST}) \approx 375 \text{ cycles.}$$

Assuming a control overhead of 20% the runtime is approximately

$$t_{ltr_w4} = [t_{reset} + t_{precalc_{odd}} + t_{precalc_{even}} + 21 \cdot (t_{AND} + 4 \cdot t_{ROL} + \cdot 2 \cdot 21 \cdot t_{wordadd}) + t_{shift4C}] \cdot 1.2 \approx 9,170 \text{ cycles.}$$

Considering the state-alone reduction the runtime increases to:

$$t_{ltr_w4_total} = t_{ltr_w4} + t_{red\_v2_{total}} = 9,795 \text{ cycles.}$$

The estimation shows that increasing the windowsize achieves a significant performance increase.

Adapted Left-to-right Multiplication with Windows

The author of this work designed an algorithm to reduce the storage requirement of the l-t-r multiplication and provide a better performance/storage trade-off. This can be achieved by performing fewer precalculations and calculating more on runtime as shown in Algorithm 8.
The idea is to ignore the last term of $u(z)$ during precalculation phase. Thus, only B_u satisfying

$$deg\{u(z)\} < w \text{ and } u(z) = u_{w-1}z^{w-1} + ... + u_2z^2 + u_1z + 0z^0$$

are determined. In vector representation this means that B_u, where u is even and B_1, are precalculated. As a consequence, only $(2^w/2 - 1 + 1) = 2^{w-1}$ elements have to be stored. The inner loop has to be executed only W/w times. If the processed window of $a(z)$ is even, it is necessary to add the corresponding B_u, to the accumulator $c(z)$. If the value of the window is odd, the last bit of u is set to zero for the determination of the required B_u. Additionally, the value of B_1 is added to $c(z)$. This can be done efficiently, since every manipulated word of $c(z)$ has to be loaded and stored only once.

Algorithm 8: Adapted window method with windows of width w.

Input: $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$ of degree at most $2m-2$

```
1  C ← 0
2  Compute b_u = b(z) · u(z) for all u(z) with even degree lower than w − 1
3  for k ← (W/w) − 1 downto 0 do
4      for j ← t − 1 downto 0 do
5          Let u = (u_{w−1}, ..., u_1, 0), where u_i is bit (wk + i) of A[j]
6          if bit wk of A[j] is set then
7              if u = 0 then u ← 1
8              for i ← 0 to t − 1 do C[i + j] ← C[i + j] ⊕ B_u[i]
9          else if u ≠ 0 then
10             for i ← 0 to t − 1 do C[i + j] ← C[i + j] ⊕ B_1[i] ⊕ B_u[i]
11     if k ≠ 0 then C ← z^w · C
```

Example: The example value of A is shown in Figure 3.8 at page 59.
First B_1, B_2, B_4, B_6, B_8, B_{10}, B_{12} and B_{14} are precalculated as shown in Table 3.4.
Thereafter, three bits of every word are considered to determine u as follows:

- The outer loop in line 4 of Algorithm 8 starts with $k = 1$:
  - Process $A[0] \Rightarrow u = 000 \Rightarrow$ bit 4 of $A[0]$ is not set $\Rightarrow$ continue
  - Process $A[1] \Rightarrow u = 010 \Rightarrow$ bit 4 of $A[1]$ is set $\Rightarrow$ add $(B_4[0], B_4[1], ..., B_4[20]) \oplus (B_1[0], B_1[1], ..., B_1[20])$ to $(C[19], C[20], ..., C[40])$
  - ...
  - Process $A[20] \Rightarrow u = 110 \Rightarrow$ bit 4 of $A[20]$ is set $\Rightarrow$ add $(B_{13}[0], B_{13}[1], ..., B_{13}[20]) \oplus (B_1[0], B_1[1], ..., B_1[20])$ to $(C[0], C[1], ..., C[20])$
- Shift all words of C four times left
- Repeat the loop with $k = 0$:
  - Process $A[0] \Rightarrow u = 011 \Rightarrow$ bit 0 of $A[1]$ is not set $\Rightarrow$ add $(B_6[0], B_6[1], ..., B_6[20])$ to $(C[20], C[21], ..., C[41])$
  - Process $A[1] \Rightarrow u = 110 \Rightarrow$ bit 0 of $A[1]$ is not set $\Rightarrow$ add $(B_{12}[0], B_{12}[1], ..., B_{12}[20])$ to $(C[19], C[20], ..., C[40])$
  - ...
  - Process $A[20] \Rightarrow u = 001 \Rightarrow$ bit 0 of $A[1]$ is not set $\Rightarrow$ add $(B_2[0], B_2[1], ..., B_2[20])$ to $(C[0], C[1], ..., C[20])$

Runtime Analysis First, the general runtime of the new multiplication method is analyzed. Thereafter, the runtime of an Ameba implementation with a window size of $w = 4$ is approximated.
The performance of the precalculation was accelerated from $\mathcal{O}(2^w t)$ to $\mathcal{O}(2^{w-1} t)$ and is twice as fast.
The runtime of the addition loop of the left-to-right multiplication is estimated to be $\mathcal{O}(\frac{W}{w} t^2)$. Below, the average, worst and best case of the runtime of the adapted additions in the loop are approximated.

Average Case For the average case estimation, it is assumed that the values of A are equally distributed (which should be the case in cryptography, where the values are like random values). Thus in the average case bit wk of the words A is set every second time, which causes an additional addition. Thus, the addition loop of the adapted version (lines 3 to 10 of Algorithm 8) requires a runtime of

$$\mathcal{O}(\tfrac{W}{w} t(\tfrac{1}{2} t + \tfrac{1}{2} 2t)) = \mathcal{O}(\tfrac{W}{w} \tfrac{3}{2} t^2).$$

This means that the runtime of the addition loop is in average case 1.5 times slower as the state-of-the art windowing approach. However, the storage requirement is reduced almost by a half.

Worst Case In the worst case the bit wk of every word in A, and every possible value of w is set. This means that each iteration of the addition loop requires one additional addition of B_1 and thus the runtime is:

$$\mathcal{O}(\frac{W}{w}t(2t)) = \mathcal{O}(2\frac{W}{w}t^2).$$

In the worst case the runtime is two times slower as the state-of-the-art approach.

Best Case In the best case bit wk of every word in A, and every possible value of w is zero. Since no further additions are required, the runtime stays the same.

Adapted Multiplication with Windows of Width w=4 When using a window size of four, only eight elements (168 bytes) have to be stored in RAM.
The runtime estimation can be done similarly to the left-to-right window method with $w = 4$. Additionally the check of the bit wk of the current processed word of A has to be considered. A `SEQB` and a conditional jump-instruction can realize the if-branch in line 6 of Algorithm 8. It is assumed the check is fulfilled every second time.
The overall runtime can be approximated as follows:

$$t_{ltr\_adapted\_w4} = (t_{reset} + t_{precalc_{even}} + 2 \cdot 21 \cdot (4t_{ROL} + t_{AND} + t_{SEQB} + t_{JNE} + \frac{1}{2} \cdot (4 \cdot t_{ROL} + +t_{AND}) + 21 \cdot (\frac{1}{2}t_{add} + \frac{1}{2} \cdot (2 + t_{add})) + t_{shift4C}) \cdot 1.2 \approx 9,400 \text{ cycles.}$$

Inclusive the stand-alone reduction one adapted multiplication requires approximately: $t_{ltr\_adapted\_w4\_total} + t_{red\_v2\_total} = 10,025$ cycles.

This means that this approach is estimated to be about 20% slower than the standard windowing method with windows of a width $w = 4$. However, the storage requirement for the precalulated values has been reduced by almost a half.

Comparison of Polynomial Multiplication Methods

Table 3.5 shows a comparison between different polynomial multiplication methods. To implement the fastest multiplication method 356 bytes of RAM are required. Since the goal is to implement ECC with very small silicon-footprint this storage requirement is considered to be too high.
The windowing method with windows of width $w = 2$ and the adapted window method with windows of width $w = 4$ offer good storage/performance trade-offs. Thus, these two variants will be implemented and evaluated.

Method	*Time [Cycles]*	*Data Memory [Byte]*
Left-to-right	15,180	41
Window Method (w=2)	16,075	104
Windows Method (w=4)	9,795	356
Adapted Window Method (w=4)	10,025	209

Table 3.5: Comparison of variants of the left-to-right polynomial multiplication algorithm.

3.3.7 Polynomial Squaring

Squaring of a polynomial can be achieved much faster than multiplying two polynomials, due to the following fact [41, p. 52]:
If $a(z) = a_{m-1}z^{m-1} + ... + a_2z^2 + a_1z + a_0$, then

$$a^2(z) = a_{m-1}z^{2m-2} + ... + a_2z^4 + a_1z^2 + a_0$$

A binary vector representation of $a^2(z)$ can be determined by inserting zeros between two consecutive bits of the binary representation of $a(z)$.
Several variants of squaring with stand-alone and interleaved reduction are discussed below.

Squaring with Stand-alone Reduction

This method first expands an array A to the 41-byte counterpart $A2$ by inserting zeros between two consecutive bits of the vector representation. Thereafter, $A2$ is reduced with the stand-alone reduction method. An efficient way to expand the element is to use a LUT [41, p. 52]. Different variants concerning the size of the LUT are discussed below.

Variant 1 A 512 byte LUT can be used to hold the expanded 16-bit value of every possible 8-bit binary vector. The resulting 16-bit value can be stored in the higher and lower byte of the table.
Consequently, to extend one byte of A it is necessary to load this byte, fetch the corresponding higher and lower bytes from the LUT and store them into the result array $A2$. Thereafter, the expanded array $A2$ is reduced with the stand-alone reduction method.
To address the bytes of A and $A2$ several pointer additions are necessary. When accessing the ROM the DPH of the Ameba has to be set to the startaddress of the LUT with a MOV instruction. Afterwards the RAM is accessed and thus the DPH has to be resetted again. In sum, this requires four MOV instructions. The runtime can be estimated as follows:

$$t_{square\_sa\_v1} = 21(t_{LD} + t_{MVCL} + t_{MVCH} + 2t_{ST} + 3t_{ADD} + 4t_{MOV}) = 315\ cycles.$$

Estimating 20% control overhead and using the stand-alone reduction variant 2 as described in section 3.3.5, the following runtime is estimated:

$$t_{square\_sa\_v1_total} = t_{square\_sa\_v1} \cdot 1.2 + t_{red\_v3_total} \approx 1,005\ cycles.$$

Variant 2 Another approach is to expand every nibble of a byte separately to eight bits. A 32-byte LUT is necessary to expand four bits to one byte. Again, the procedure processes every byte as shown in Algorithm 9. The shift-right operation in line 3 can be implemented with four rotate-left instructions.

The runtime of this method, can be estimated similar to the variant described above:

$$t_{square\_sa\_v2} = [21 \cdot (t_{LD} + 2t_{MVCL} + 5t_{MOV} + 2t_{AND} + 4t_{ROL} + 2t_{ST} + 3t_{ADD})] \cdot 1.2 \approx 530\ cycles.$$

Algorithm 9: Squaring with stand-one reduction. The 32-bythe LUT *square_table* contains 8-bit counterparts with inserted zeros between consecutive bits of a 4-bit value.

Input: $c(z) = \{c_{m-1}z^{m-1} + \cdots + c_1 z + c_0\}$
Output: $c^2(z)$
1 **for** $i \leftarrow 20$ ***downto*** 0 **do**
2 $C[2i-20] \leftarrow square\_table[C[i] \ \& \ 0x0F]$
3 $C[2i-21] \leftarrow square\_table[C[i] >> 4]$
4 $C[0] \leftarrow square\_table[C[10] \ \& \ 0x03]$
5 *Stand-alone reduction of C*

Considering the time for the stand-alone reduction, the performance of this variant is approximately:

$$t_{square\_sa\_v2\_total} = t_{square\_sa\_v2} + t_{red\_v3\_total} \approx 1,155 \ cycles.$$

Although the estimated runtime of this variant is higher, it needs 480 bytes less storage for the LUT.

Squaring with Interleaved Reduction

The author of this work proposes an approach for squaring with interleaved reduction using the fact that every second bit is zero. The lower half of the vector representation

$$a_{(m-1)/2}z^{(m-1)/2} + ... + a_1 z + a_0$$

stored in $A[\frac{t-1}{2}]...A[t-1]$ is expanded with LUTs similar to the squaring with stand-alone reduction. The result is the array $A2[0]...A2[t-1]$. If a 16-bit LUT is used, then the execution time of this step is approximately:

$$t_{square\_int\_step1} = t_{square\_sa\_v2\_total}/2 \approx 265 \ cycles.$$

The reduction is performed interleaved with a reduction rule similar to the standard reduction. If one bit k_i of $A[j]$, where $j \leq 10$, should be squared and reduced interleaved, it is necessary to add $R_h[n]$ to $A[index_{high}]$ and $R_l[n]$ to $A[index_{low}]$ according to Table 3.3.7.

i	0	1	2	3	4	5	6	7
n	5	7	1	3	5	7	1	3
$index_{low}$	2j		2j-1		2j-1		2j-2	
$index_{high}$	2j+1		2j		2j		2j-1	

Table 3.6: Interleaved reduction rule for squaring.

It can be seen that five bytes have to be manipulated to reduce one byte. More precisely, to reduce $A[j]$ represented by the bits $\{a_7, ..., a_1, a_0\}$, the following calculations

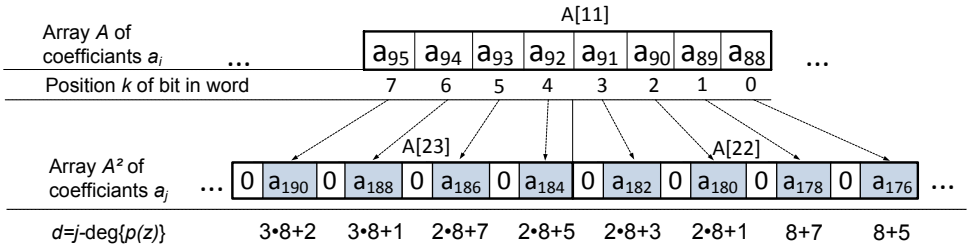


Figure 3.9: Squared counterpart of byte $A[11]$ and determination of the corresponded reduction rule. To reduce the squared bit of a_i the binary vector representation of $r(z)$ has to be shifted d times left and added to A^2.

have to be performed:

$$\begin{aligned} A[2j+1] &\leftarrow A[2j+1] \oplus R_{s0}(a_1, a_0) \\ A[2j\ \ \ ] &\leftarrow A[2j\ \ \ ] \oplus R_{s1}(a_5, ..., a_1, a_0) \\ A[2j-1] &\leftarrow A[2j-1] \oplus R_{s2}(a_7, ..., a_3, a_2) \\ A[2j-2] &\leftarrow A[2j-2] \oplus R_{s3}(a_7, a_6) \end{aligned} \tag{3.16}$$

The values of R_{s0}, R_{s1} and R_{s2} can be obtained as follows:

$$\begin{aligned} R_{s0} &= a_1 \cdot R_h[7] \oplus a_0 \cdot R_h[5] \\ R_{s1} &= a_5 \cdot R_h[7] \oplus a_4 \cdot R_h[5] \oplus a_3 \cdot R_h[3] \oplus a_2 \cdot R_h[1] \\ &\quad \oplus a_1 \cdot R_l[7] \oplus a_0 \cdot R_l[5] \\ R_{s2} &= a_7 \cdot R_h[3] \oplus a_6 \cdot R_h[1] \oplus a_5 \cdot R_l[7] \oplus a_4 \cdot R_l[5] \\ &\quad \oplus a_3 \cdot R_l[3] \oplus a_2 \cdot R_l[1] \\ R_{s3} &= a_7 \cdot R_l[6] \oplus a_6 \cdot R_l[1] \end{aligned}$$

Similar to the standard reduction operation, there are several variants to implement the calculation of R_{s0}, R_{s1}, R_{s2} and R_{s3}.

Variant 1 The first option is to calculate the values of $R_{s0}, ..., R_{s3}$ at runtime. It is assumed that the values of $R_h[0]$,...,$R_h[7]$ and $R_l[0]$,...,$R_l[7]$ are stored as constants. To determine R_{s0},...,R_{s3} all bits of the current processed word of A are scanned. The values of R_{s0},...,R_{s3} are stored in temporary registers. If the current processed bit is one, the corresponding value of Rh or Rl is XORed to the registers according to Equation 3.17. For example the code segment written in Ameba assembler to perform the calculation $R_{s0} \leftarrow R_{s0} \oplus a_1 \cdot R_h[7]$ could look like:

```
LD Rc, C[j]
SEQB Rc, #1
JNE  next calculation
XOR  Rs0, #value of Rh[7]
...
```

The runtime of this code segment can be estimated as $t_{addR} = 4$ cycles.
As a result the runtime to calculate $R_{s0}, ..., R_{s3}$ can be approximated as follows:

$$\begin{aligned} t_{Rs0} &= 2t_{addR} = 8\ cycles \\ t_{Rs1} &= 6t_{addR} = 24\ cycles \\ t_{Rs2} &= 6t_{addR} = 24\ cycles \\ t_{Rs3} &= 2t_{addR} = 8\ cycles \end{aligned} \tag{3.17}$$

Considering the load and store instructions, pointer additions and overhead required for Equation 3.16 and the first step for squaring, the estimated runtime of this variant is

$$\begin{aligned} t_{square\_int\_v1} &= 10 \cdot (5t_{LD} + 4t_{ST} + 4t_{ADD} + t_{Rs0} + t_{Rs1} + t_{Rs2} + t_{Rs3}) \cdot 1.2 \\ &+ t_{square\_int\_step1} \approx 1,420\ cycles \end{aligned} \tag{3.18}$$

Variant 2 The usage of LUTs can accelerate the interleaved squaring method. The calculation of R_{s1} and R_{s2} require many cycles. One option is to store the values of R_{s1} in the lower column and R_{s2} in the higher column of a LUT. This requires 128 bytes of ROM.
Then the runtimes t_{Rs1} and t_{Rs2} are $t_{MOVCL/MOVCH} + 2t_{MOV} = 4cycles$. Performing the same estimation as in Equation 3.18 with the new values of t_{Rs1} and t_{Rs2}, the runtime is approximated as follows:

$$t_{square\_int\_v2} = \approx 930\ cycles \tag{3.19}$$

The runtime of this option is estimated to be about 40% faster than the first variant.

Comparison of Squaring Methods

Since interleaved reduction do not extend the result to a double element, this method is used. The second interleaved reduction variant seems to offer the best trade-off between the ROM size and execution time.

Method	*Time*	*LUT [Byte]*
Stand-alone V1	850	512
Stand-alone V2	1,010	32
Interleaved V1	1,470	32
Interleaved V2	920	160

Table 3.7: Comparison of squaring methods.

3.4 Virtual Addressing

Typically, acceleration of ECC is achieved by dedicated coprocessors or instruction set extension (ISE). Here, the author of this work explores an innovative acceleration approach by using Virtual Addressing (VA).
Many procedures access memory consecutively causing many pointer calculations. This could be avoided by encoding the manipulated memory positions with fixed values. However, this would require duplicated code, which would increase the code size enormously. Virtual addressing allows to write a code segment with fixed virtual addresses without significantly increasing the code size.
Typically, a microprocessor has an address bus (`addr_o`) and a data bus (`data_o`), which are directly connected to RAM as shown in the upper part of Figure 3.10. In case of a virtual addressing, additional logic is implemented between the microprocessor and the RAM.

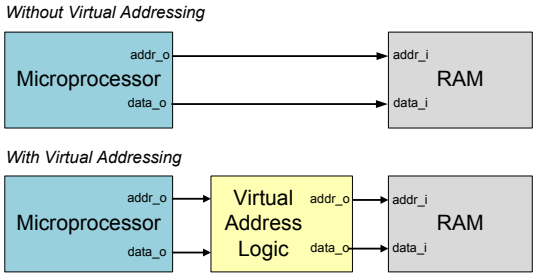


Figure 3.10: Principle of Virtual Addressing. By default the output-address of the microprocessor is connected to the RAM. Virtual addressing manipulates the address before it forwards it to the RAM.

3.4.1 Address Organization

The Ameba is able to address 256 bytes of memory using a single load or store instruction. To increase the memory space an additional data pointer is available (DPH), which determines the upper bits of the memory address. In this work two memory banks of 256 bytes are used.
The temporary values of the Montgomery multiplication are stored in the lower bank. The upper bank is used by the polynomial multiplication.

3.4.2 Virtual Addressing for Addition, Copying and Resetting

A Virtual Element (VE) represents continuous address range of 21 words. The virtual addressing logic maps these addresses to 21 adjoined physical words. The conversion between virtual element and physical addresses is influenced by parameters. These parameters can be set from the microprocessor by writing to certain addresses.
Figure 3.11 shows the principle of VA.

The parameters `element0` and `element1` indicate to which addresses the VEs point to. The remaining parameters are used for polynomial multiplication and will be discussed later in this section.

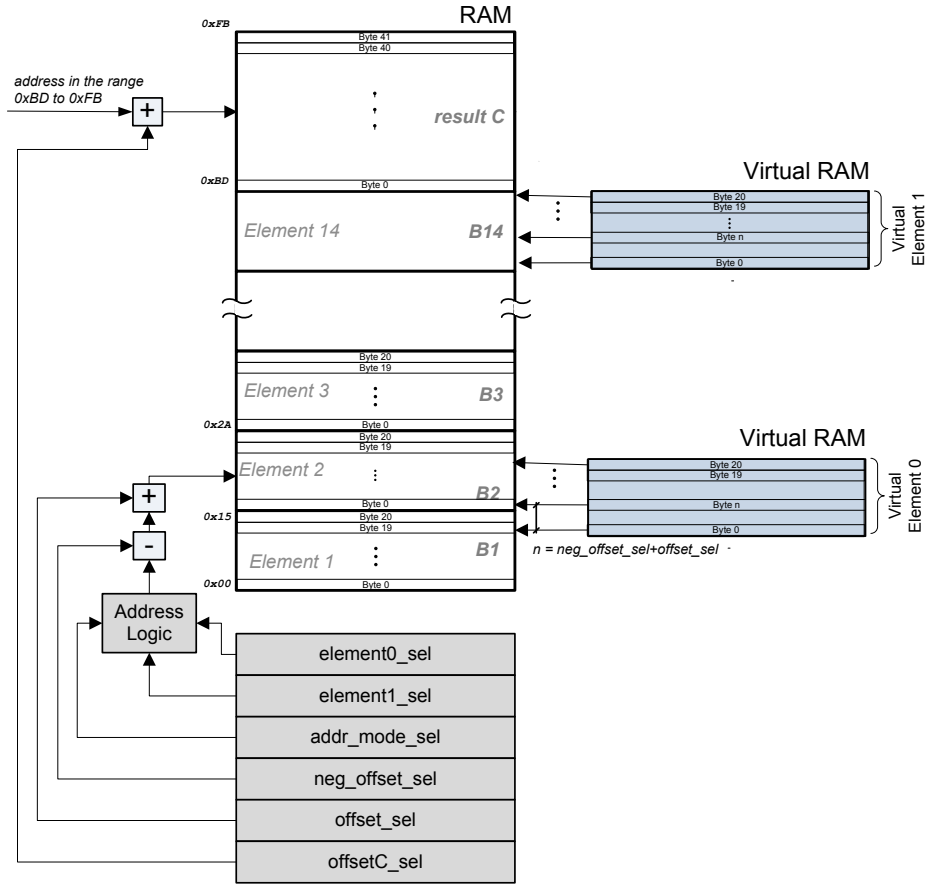


Figure 3.11: Virtual addressing with two virtual elements and six parameters.

The VA allows to write procedures with fixed addresses of the VEs. An example of an addition using the VA is shown in Algorithm 10. Virtual addressing can also be used for copying one element to the position of another and initializing elements.

Algorithm 10: Addition of binary field elements with virtual addressing.

Output: $X_1 = X_1 + Z_1$

```
1  MOV  r0, #20                              // loop counter
2  ST   #6, ELEMENT0_SEL                     // X1 has the element number 6
3  ST   #8, ELEMENT1_SEL                     // Z1 has the element number 8
4  MULT_LOOP:                                // addition loop
5    LD   Ra, VE0[i]
6    LD   Rb, VE1[i]
7    XOR  Ra, Rb,
8    ST   Ra, VE0[i]
9    SUB r0, #1
10   JNE MULT_LOOP                           // jump if r0 > 0
```

3.4.3 Virtual Addressing for Polynomial Multiplication

VA accelerates the word-wise additions of the adapted l-t-r multiplication by reducing the number of pointer calculations and memory accesses. Figure 3.12 shows the additions required to process the first window and how they are realized with VA. The algorithm steps through all words of A, determines u and adds B_u to C. If the processed window is odd, also B_1 is added.

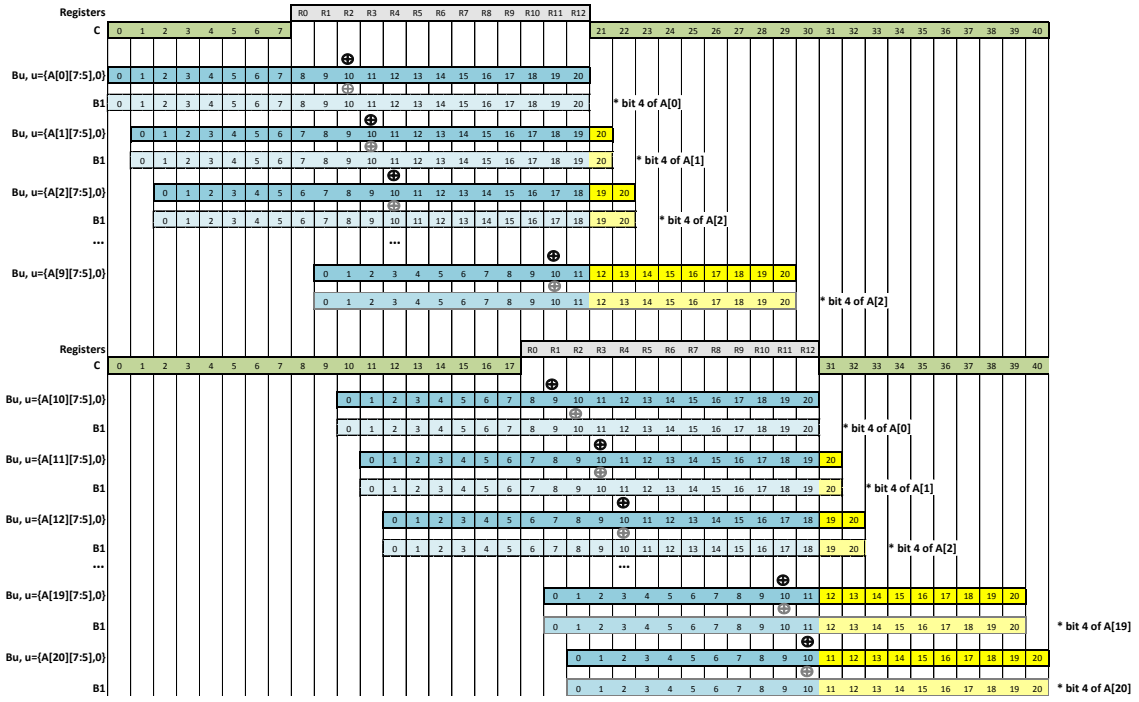


Figure 3.12: Visualization the first window of the adapted l-t-r multiplication using virtual addressing. The processing of the second window is similar. However, $A[i][3:1], 0$ defines u and the bit 0 of $A[i]$ determines if an additional addition of B_1 is necessary.

Normally, for each word-wise addition pointer additions are necessary to calculate the addresses of the processed words. These pointer calculations can be omitted by using VEs. Figure 3.12 shows that the operations on these bytes of C, which are altered most frequently, are performed with registers: instead of loading values from memory and storing the altered content back to the same position, all operations targeting these addresses are performed with registers. This significantly reduces the number of memory accesses.
The pattern for the addition of the first ten and the second eleven additions is similar and thus is implemented only once. The processing of the first half is implemented and whenever the second half is considered, the addresses of C are increased by an offset of 10.

The details of the VA concept for polynomial multiplication are described below.

Concept Details

For polynomial multiplication only one VE (Virtual Element 0) and five parameters are used. This section first describes the parameters and then shows how the l-t-r algorithm is executed step-by-step.

Processing of the Windows The software implementation requires several instructions to determine u and calculate the startaddress of B_u. This can be outsourced to the virtual address logic.
The VA includes two parameters `element0` and `addr_mode`. The parameter `element` is set to the processed word of A. The parameter `addr_mode` defines which bits are used to determine u. If `addr_mode` is one, the value of u is {`element[7:5],0`} else u is {`element[3:1],0`}. The VE points to the associated B_u.

Shifting B_u As stated above, the parameters `addr_mode` and `element` determine, to which B_u the virtual element points to. However, there are two additional parameters, which influence the address mapping.
A parameter `neg_offset` is subtracted from the virtual address and the parameter `offset` is added to the address. Figure 3.13 illustrates the mapping of the virtual addresses. Some addresses of the virtual element could point to the words in the RAM, which do not belong to B_u. However, these addresses are never used.

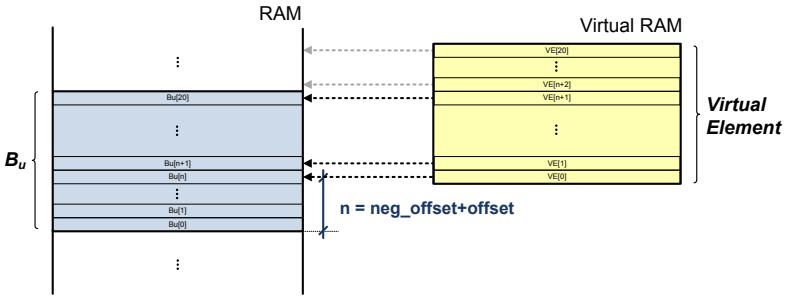


Figure 3.13: Address mapping using the virtual element.

Subroutines for Word-wise Addition of B_u to C The word-wise additions are implemented in four subroutines, which offer several entry points. The number of word-wise additions performed by one subroutine depends on the entry point, which is called.

ADD_B1_E_[n] This subroutine implements the addition of the VE to C from left-to-right as shown in Figure 3.14. If the entry point $ADD\_B1\_E\_[n]$ is called $(21 - n)$ additions are executed. The last 13 additions are done with registers representing the values of $C[8]$ to $C[20]$.

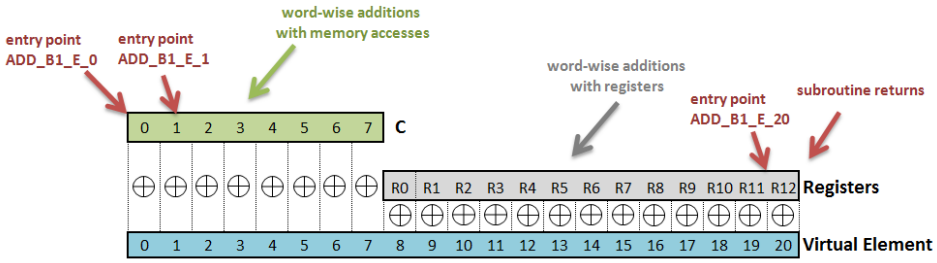


Figure 3.14: Illustration subroutine `ADD_B1_E_[n]` for the multiplication with virtual addressing.

ADD_B1_O_[n] This subroutine is called, if the window is odd. It is similar to the subroutine `ADD_B1_E_[n]`. However, it additionally adds the element B_1.

ADD_B2_E_[n] This subroutine implements nine word-wise additions and processes the VE from right-to-left as shown in Figure 3.15. If the entry point $ADD\_B2\_E\_[n]$ is called, n additions are executed.

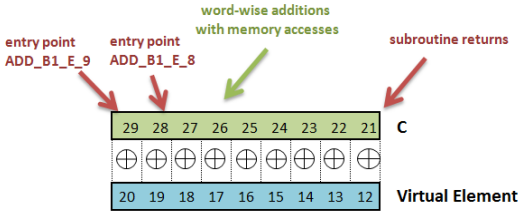


Figure 3.15: Illustration of subroutine `ADD_B2_E_[n]`.

ADD_B2_O_[n] This subroutine is implemented in the same way as `ADD_B2_E_[n]`. However, also $B_1[20]$ to $B_1[12]$ is added.

Shifting the Accumulator C To shift C, there is a parameter `offsetC`. Whenever an address of C is accessed, this parameter is added to the pointer. Figure 3.16 illustrates the memory additions of the subroutine `ADD_B1_EVEN` with the setting `offsetC = 10`.

Algorithm for Binary Field Multiplication with Virtual Addressing The adapted l-t-r multiplication algorithm is designed for a windowsize of $w = 4$, a wordsize of $W = 8$ and 13 available register.
To calculate $C = A \cdot B$ the algorithm precalculates eight different B_u's and stores them in to RAM. Thereafter, the algorithm steps through all words of A twice and performs several additions of B_u and B_1. The addressing of A is not influenced by VA. The result is 41 bytes long and is stored at a fixed RAM location as shown Figure 3.11.

Step 1: Process the first window The algorithm starts processing the first window of every word of A. Therefore, the parameter `addr_mode` is set to one.

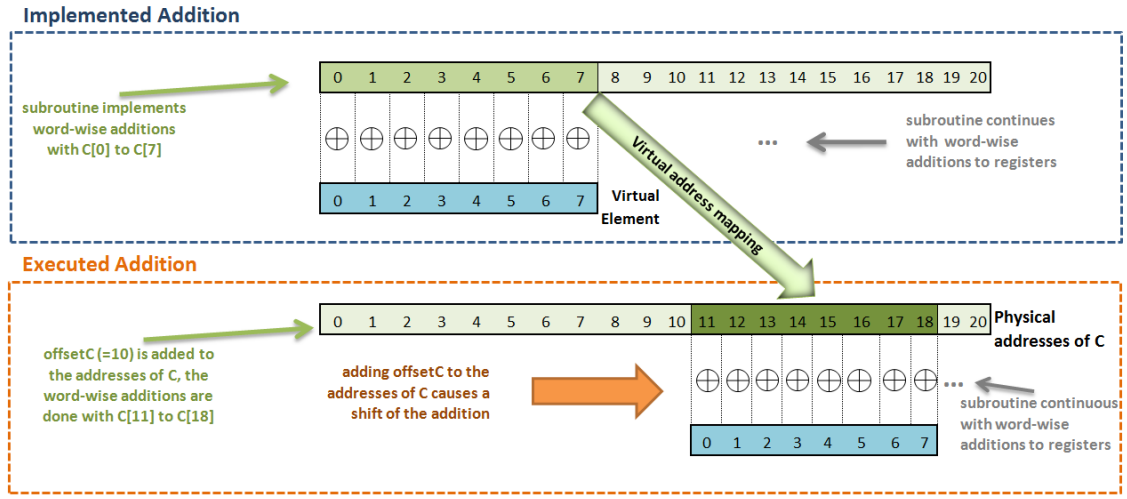


Figure 3.16: Illustration subroutine `ADD_B1_EVEN` with a shifted C by setting `offsetC = 10`.

Step 1.1: Process the first half (A[0] to A[9]) All registers, which store the intermediate results in the subroutines are set to zero. The subsequent addition subroutines are executed without shifting C. Thus, the `offsetC` is set to zero.

Step 1.1.1: Process A[0] The algorithm starts processing the first word of A as shown in Figure 3.17.

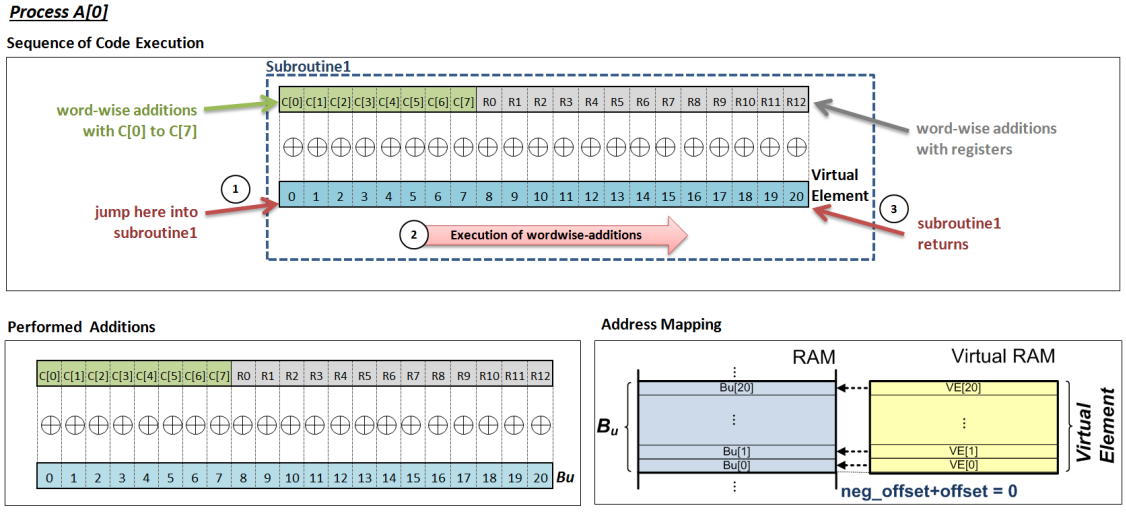


Figure 3.17: Illustration of processing the first word of A for the case that the analyzed window is even. All word-wise additions of the first subroutine are executed. The 21 words of the virtual element point to the 21 words of B_u in RAM.

To address the right B_u, the parameter `element` is set to $A[0]$. Thereafter, the bit 4 of $A[0]$ is analyzed. If it is one, the addition of the element B_u to $\{C[0]...C[20]\}$ is done by

calling the first subroutine at the entry point `ADD_B1_E_0`. Otherwise, `ADD_B1_O_0` is called. The parameters `neg_offset` and `offset` are set to zero. Thus, all words of the VE point to the words of B_u.

Step 1.1.2: Process A[1] The next step is to analyze the second word of A. Therefore, the value of $A[1]$ is loaded from RAM. Again, the addressing of B_u is achieved by setting `element` to $A[1]$.
Thereafter, the words $\{B_u[0], B_u[1], ..., B_u[20]\}$ have to be added to $\{C[1], C[2], ..., C[21]\}$. The following text describes the process if the window is even. If the window is odd, the procedure is the same, but `ADD_B1_O_1` and `ADD_B2_O_1` are called.
Figure 3.18 illustrates the processing of $A[1]$. It is not necessary to manipulate the value of $C[0]$. Thus, the algorithm jumps into the second word-wise addition of the first subroutine by calling `ADD_B1_E_1`. Thus the subroutine starts adding $VE[1]$ to $C[1]$, then adds $VE[2]$ to $C[2]$ and so on. However, $B_u[0]$ should be added to $C[1]$, $B[1]$ to $C[2]$ and so on. This can be achieved by setting `neg_offset=1`. Then the words $\{B_u[0], B_u[1], ..., B_u[19]\}$ are added.
The addition of $B_u[20]$ is done with the second subroutine. The algorithm jumps in to the last word-wise addition, which adds $VE[12]$ to $C[21]$. To achieve the mapping of $VE[12]$ to $B_u[20]$, `offset` is set to 9. Before the subroutine returns this parameter is set to zero again.

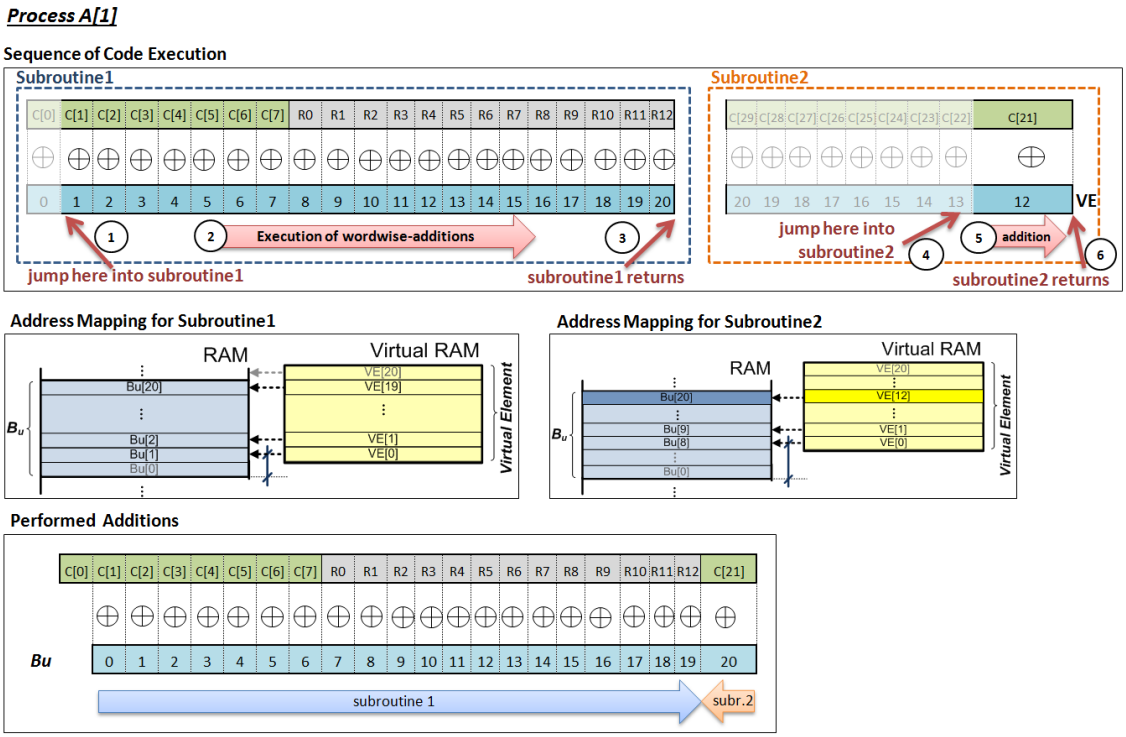


Figure 3.18: Illustration of processing the second word of A for the case that the processed window is even.

Step 1.1.3 to Step 1.1.10: process A[2] to A[9] The next steps are done in the same way as step 1.1.2. The difference between the executions is that the number of word-wise additions performed by the first and the second subroutine is different. The additions in Figure 3.12, which are illustrated in blue color are performed by the subroutines $ADD\_B1\_[E/O]\_[n]$ from left-to-right. The additions, which are yellow, are done from right-to-left with the subroutines $ADD\_B2\_[E/O]\_[n]$. For processing the word $A[i]$ the subroutine is called at the entry points `ADD_B1_[E/O]_[i]` and `ADD_B2_[E/O]_[i]`. Before calling the first subroutine the parameter `neg_offset` is set to the value of i. The parameter `offset` is set to 9 before calling the second subroutine.

Step 1.2: Process the second half (A[10] to A[20]) Before processing the second half, the registers keeping the intermediate values are stored to C. The value of $R1$ is stored to $C[8]$, $R2$ is stored to $C[9]$ and so on.
The next step is to process the second half. The additions can be done in the same way as described so far, but the accumulator C has to be shifted by setting `offsetC=10` (see Figure 3.12). The registers $R1$ to $R12$ now represent the intermediate values of $C[18]$ to $C[30]$. Thus, these values are first loaded from RAM. Thereafter, the same code that performs the additions of step 1.1 can be executed.

Step 2: Process the second window Next, the accumulator C is shifted four bits left. This shifting is not influenced by the virtual addressing mechanism.
Thereafter, the second window of every word in A is analyzed. This is achieved by setting `addr_mode=0`. The remaining procedure is the same as the steps 1.1 and 1.2. A summary of the whole procedure is shown in Algorithm 11.

Algorithm 11: Algorithm for implementing the polynomial multiplication using virtual addressing.

```
1  ADDR_MODE_SEL ← 1
2  call PROCESS_WINDOW
3  call SHIFT_C
4  ADDR_MODE_SEL ← 0
5  call PROCESS_WINDOW
6  ret
7
8  PROCESS_WINDOW:
9  Reset registers
10 OFFSETC_SEL ← 0
11 for k ← 0 to 9 do call MULT_LOOP
12 Store and load registers from/to C
13 OFFSETC_SEL ← 10
14 for k ← 0 to 10 do call MULT_LOOP
15
16 MULT_LOOP:
17 ELEMENT_SEL ← value stored in A_ptr                    // A_ptr... address of A
18 A_ptr ← A_ptr + 1
19 NEG_OFFSET_SEL ← k
20 if bit wk of ELEMENT_SEL is one then
21     call ADD_B1_ODD[k]
22 else call ADD_B1_EVEN[k] if k ≠ 0 then
23     OFFSET_SEL ← 9
24     if bit wk of ELEMENT_SEL is one then
25         call ADD_B2_ODD[k]
26     else call ADD_B2_EVEN[k]
27 ret
28
29 ADD_B1_EVEN_0: C[0] ← C[0] ⊕ VE[0]
30 ADD_B1_EVEN_1: C[1] ← C[1] ⊕ VE[1]
31 ...
32 ADD_B1_EVEN_7: C[7] ← C[7] ⊕ VE[7]
33 ADD_B1_EVEN_8: R0 ← R0 ⊕ VE[8]
34 ADD_B1_EVEN_9: R1 ← R1 ⊕ VE[9]
35 ...
36 ADD_B1_EVEN_20: R12 ← R12 ⊕ VE[20]
37 ret
38
39 ADD_B2_EVEN_9: C[29] ← C[29] ⊕ VE[20]
40 ADD_B2_EVEN_8: C[28] ← C[28] ⊕ VE[19]
41 ...
42 ADD_B2_EVEN_1: C[21] ← C[21] ⊕ VE[12]
43 OFFSET_SEL ← 0
44 ret
45 ADD_B1_ODD_0: C[0] ← C[0] ⊕ B1[0] ⊕ VE[0]
46 ADD_B1_ODD_1: C[1] ← C[1] ⊕ B1[1] ⊕ VE[1]
47 ...
48 ADD_B1_ODD_7: C[7] ← C[7] ⊕ B1[0] ⊕ VE[7]
49 ADD_B1_ODD_8: R0 ← R0 ⊕ B1[0] ⊕ VE[8]
50 ADD_B1_ODD_9: R1 ← R1 ⊕ B1[0] ⊕ VE[9]
51 ...
52 ADD_B1_ODD_20: R12 ← R12 ⊕ VE[20]
53 ret
54
55 ADD_B2_ODD_9: C[29] ← C[29] ⊕ B1[0] ⊕ VE[20]
56 ADD_B2_ODD_8: C[28] ← C[28] ⊕ B1[0] ⊕ VE[19]
57 ...
58 ADD_B2_ODD_1: C[21] ← C[21] ⊕ B1[0] ⊕ VE[12]
59 OFFSET_SEL ← 0
60 ret
```

3.5 Coprocessor

In this section an implementation variant using a coprocessor is proposed. The approach is to accelerate the calculation by outsourcing the field multiplication to dedicated hardware. The coprocessor works independently of the microprocessor and consists of a 4x8-bit multiplier and control logic.
First, the multiplication algorithm used by the coprocessor is described. Then, the concept of the communication and the architecture of the coprocessor is presented.

3.5.1 Comba's Method

In literature [47, 28, 26] it is proposed to use the Comba's method, if a partial multiplication is supported by the hardware. The procedure of the Comba's multiplication is shown in Algorithm 12. Each word of the result C is calculated at a time starting at the least significant word $C[41]$. The algorithm includes two nested loops: the first one calculates the least significant words $C[20]$ to $C[40]$ and the second one calculates the remaining words $C[0]$ to $C[19]$.
The difference to the l-t-r multiplication is the order in which the products are generated [26]. Figure 3.19 shows the multiplication of two elements, which are four words long. The order of the l-t-r algorithm is the same as the order of the schoolbook method, which writes several times to several words of the result [26]. The Comba's method needs store operations by writing every word of the result only once. However, the amount of partial multiplications stays the same.
Two words are multiplied in each iteration. The product is two words long and is stored u (higher word) and v (lower word).
In summary each iteration of the inner loop performs a multiply operation and accumulates the result. Therefore, two load operations are executed. In the outer loop the store operations are performed.

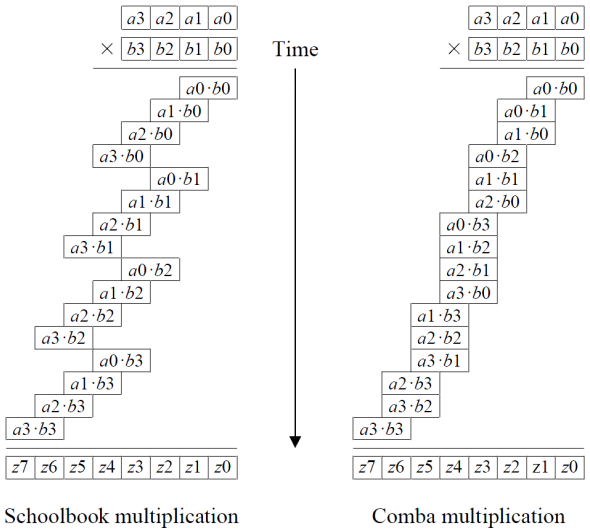


Figure 3.19: Comparison of schoolbook and Comba multiplication [26].

Algorithm 12: Comba's method for binary field multiplication. Adapted from [47]. The operator $\otimes$ stands for a word-level multiplication.

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$

```
1  (u, v) ← 0
2  for i ← 20 downto 0 do
3      for j ← 20 downto i do
4          (u, v) ← (u, v) ⊕ (A[j] ⊗ B[i − j + 20])
5      C[i + 20] ← v
6      v ← u, u ← 0
7  for i ← 19 downto 0 do
8      for j ← i downto 0 do
9          (u, v) ← (u, v) ⊕ (A[j] ⊗ B[i − j])
10     C[i] ← v
11     v ← u, u ← 0
12 C[0] ← v
```

3.5.2 Comba's method with interleaved reduction

The second outer loop of the Comba's method calculates the most significant words of the result which have to be reduced. Interleaved reduction can be applied efficiently. Instead of storing a word of the result and reading the word again in the subsequent reduction procedure, the reduction of the word can be done immediately.
Note that the reduction of one word of the result $C[i]$ requires the following calculations:

$$\begin{aligned} C[i+21] &\leftarrow C[i+21] \oplus R_0(c_2, c_1, c_0) \\ C[i+20] &\leftarrow C[i+20] \oplus R_4(c_7, ..., c_1, c_0) \\ C[i+19] &\leftarrow C[i+19] \oplus R_3(c_7, ..., c_4, c_3) \end{aligned} \tag{3.20}$$

The second outer loop of the Comba's method can be changed to perform these calculations.
The loop first calculates $C[20]$, then $C[19]$ and so on. This means that if one word $C[i]$ is processed, only the value of $C[i+21]$ remains unchanged in the next iteration. This is the reason why it is only necessary to store this value. The results of the remaining calculations $R_4(a_7, ..., a_1, a_0)$ and $R_3(a_7, ..., a_4, a_3)$ can be stored in the registers. This means that the number of store operations remains the same as in the standard Comba's method.
Algorithm 13 shows the approach. Since the interleaved reduction is performed, the result is just 21 bytes long. Thus the indices of the result C are adapted in line 5 and 10.

Algorithm 13: Comba's method for binary field multiplication with interleaved reduction.The operator $\otimes$ stands for a word-level multiplication.

Input: Binary polynomials $a(z)$ and $b(z)$ of degree at most $m-1$
Output: $c(z) = a(z) \cdot b(z)$

```
1  (u, v) ← 0
2  for i ← 20 downto 0 do
3      for j ← 20 downto i do
4          (u, v) ← (u, v) ⊕ (A[j] ⊗ B[i − j + 20])
5      C[i] ← v
6      v ← u, u ← 0
7  for i ← 19 downto 0 do
8      for j ← i downto 0 do
9          (u, v) ← (u, v) ⊕ (A[j] ⊗ B[i − j])
10     C[i] ← C[i] ⊕ R0(v2, v1, v0) ⊕ r2
11     r2 ← R1(v7, ..., v1, v0) ⊕ r1
12     r1 ← R3(v7, ..., v1, v0)
13     v ← u, u ← 0
14 C[19] ← C[19] ⊕ R1(r1) ⊕ v
15 v ← v ⊕ r2
16 C[20] ← C[20] ⊕ R1(v & 0xF8) ⊕ R0(r1)
17 C[0] ← v & 0x07
```

3.5.3 Memory Access and Communication

The Ameba, the RAM and the coprocessor are connected at top level as shown in Figure 3.20. Direct Memory Access (DMA) for the coprocessor is used.

To keep the communication overhead low, DMA is used. The coprocessor directly communicates with the RAM. The coprocessor multiplies two elements, which are stored in the RAM, and writes the result back to RAM.
The Ameba provides the information, where the elements, which have to be multiplied, are located in the RAM. Furthermore, the Ameba also determines the memory location of the result. The Ameba transfers this information by writing to predefined addresses. The address logic forwards the data to the coprocessor. The coprocessor has several ports for the communication with RAM. Hence, it is able to directly read the values of the two factors, which should be multiplied. Additionally, the coprocessor writes the result to the specified position.
The coprocessor has *halt_o* signal to indicate that the microprocessor should be halted. When the halt signal is set the clock of the Ameba is disabled. Thus the Ameba pauses the processing. This ensures that there is only the coprocessor or the Ameba that can access the RAM at the same time.

3.5.4 Coprocessor Architecture

The coprocessor is implemented as a stand-alone hardware block with ports as shown in Figure 3.20. It consists of a combinatorial and control logic.

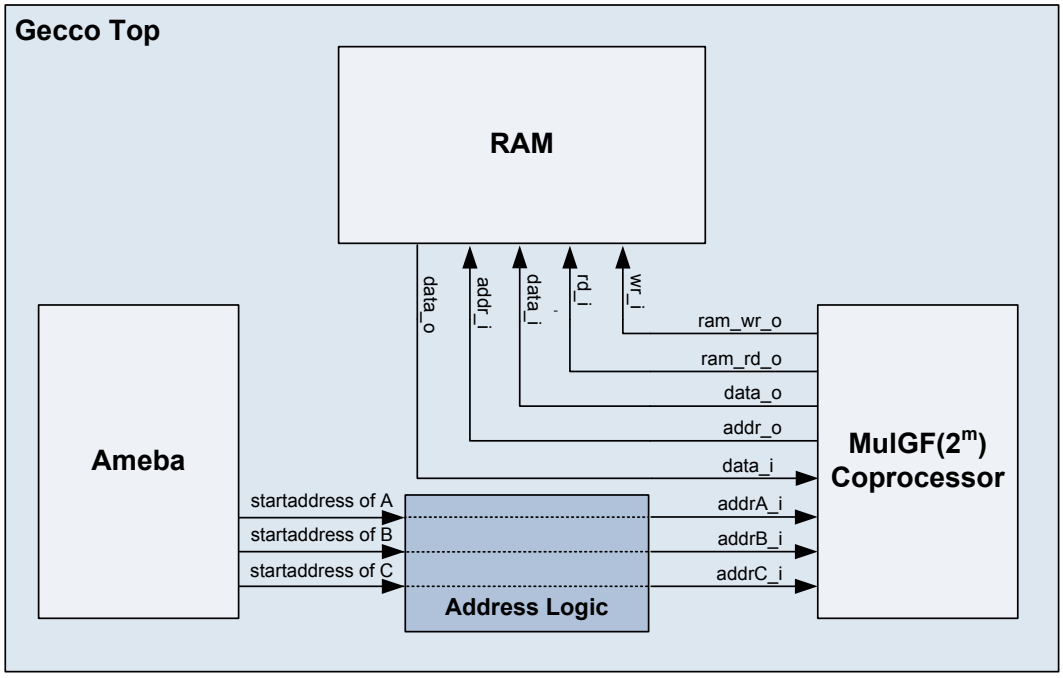


Figure 3.20: Interfaces between Ameba, RAM and coprocessor. The connection of the modules is realized in the topfile.

Partial multiplication

The partial 8x8-bit multiplication required for the Comba's multiplication algorithm is realized as combinatorial logic. The partial multiplier calculates $A \cdot B = C$, where the lengths of A and B is 8 bit and the length of the result C is 16 bit. The coprocessor uses registers to store the values of A, B and C.
To keep the area small not a 8x8-bit multiplication, but the control logic of a 4x8-bit multiplier is implemented. The interfaces of the multiplier are illustrated in Figure 3.21. The multiplier calculates $a \cdot b = c$, where a is four bit, b is eight bit and c is ten bits long.

Figure 3.21: Illustration of inputs and outputs of a 4x8-bit multiplication.

Figure 3.22 shows how the multiplication is realized. There are 32 AND and 12 XOR gatters required.

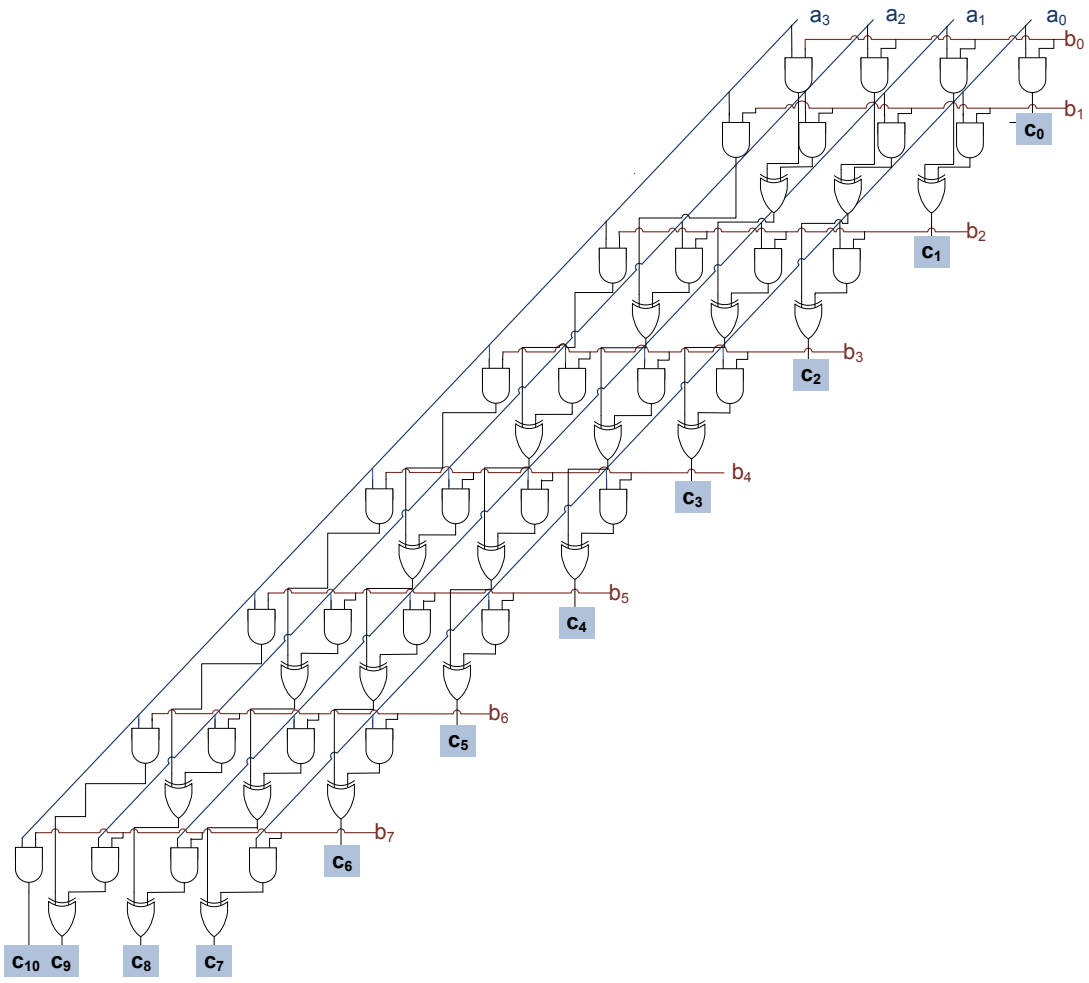


Figure 3.22: Combinatorial logic of a 4x8-bit multiplication.

The 8x8-bit partial multiplication, which is required in the Comba's multiplication algorithm is realized with the 4x8-bit multiplication function as shown in Figure 3.23.

First, the most significant nibble of the 8-bit register that stores the value of the first multiplicand A is taken as input of the multiplication function. The second input is the 8-bit register that stores B. The 4x8-bit multiplication function generates a ten bit output. The most significant three bit of this output represent the higher three bit of the final result.

Then, the multiplication function is used again. Now, the lower nibble of the register A is used as input. The function again generates ten bits. The least significant three represent the least significant three bits of the final result. The other bits have to be $XORed$ with the least significant seven bits of the previous multiplication.

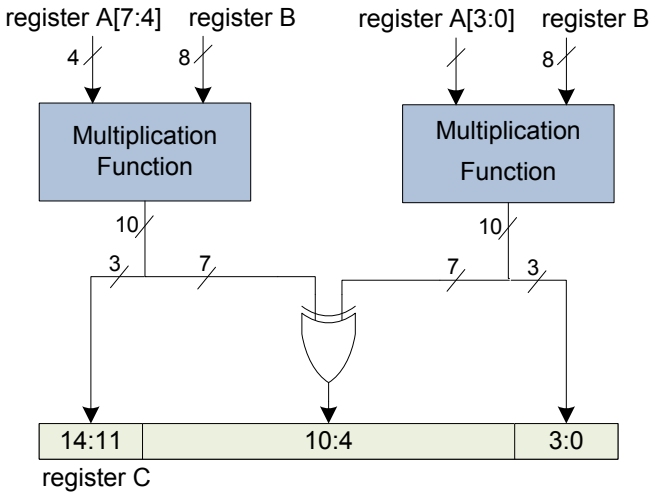


Figure 3.23: Construction of a 8x8-bit partial multiplication with a 4x8-bit multiplication function.

3.5.5 Reduction

The reduction functions R_0, R_4 and R_3 are implemented hardwired. These functions depend on the least significant 8-bits of the register holding the result C.
The functions R_0 and R_4 require several XOR gatter. The interfaces of these two functions are illustrated in Figure 3.24. It is assumed that about 15 XOR gatters will be required to implement these two functions.
The function R_3 can simply be realized by rewiring the least significant four bits of the register storing C.

Figure 3.24: Illustration of inputs and outputs of the reduction functions.

Control Logic

The Comba's algorithm is implemented as Finite State Machine (FSM). The coprocessor gets the startaddresses of the factors for the multiplication and the address, where the result should be stored, from the Ameba. As shown in Figure 3.20 the coprocessor has three inputs to get these addresses. Whenever one of these inputs is changed, the input value is stored into a register.
The Ameba microprocessor first provides the addresses of the factor A and the result C. The memory mapping forwards these values to the inputs $addr\_a\_i$ and $addr\_c\_i$ of the

coprocessor. Then the Ameba gives the address of the second factor B, which is forwarded to the port *addr_b_i*. Three 8-bit registers are used to store these addresses. Whenever the value of one of these inputs changes, the value is either stored in the register *addr_a_s*, *addr_b_s* or *addr_c_s*.
The FSM stays in an idle state until the input *addr_b_i* is changed. Then the FSM starts processing Algorithm 13.
Two registers are provided to count the iterations of the outer and inner loop. Whenever an iteration of the inner loop is executed, new bytes from the 21-byte elements A and B are required (see Line 4 of Algorithm 13). Therefore, the required addresses are calculated and the registers *addr_a_s* and *addr_b_s* are updated. The coprocessor directly communicates with the RAM and gets the values stored in the memory locations *addr_a_s* and *addr_b_s*. These values are stored in two separate registers *b_s* and *c_s*. Each read access is realized in an own state.
The temporary values of u and v are stored in one 16-bit register *c_s*. If the outer loop is executed, then the lower byte of the register gets the value of the higher byte. The higher byte is set to zero. This implements the lines 6 and 13 of Algorithm 13. Furthermore, the byte of the result C, which has been determined in the current outer loop has to be stored. Again, this is done by directly communicating with the RAM.
The 8x8-bit partial multiplication in line 4 is realized with the 4x8-bit multiplication functions in the control logic with two states as described above. The reduction functions in the lines 10 to 12 use the reduction functions R_0 and R_4, which are implemented as logical circuits.

3.6 Summary

When designing an ECC system many decisions have to be made during the design phase. In this chapter, the design decisions made for the ECC implementation are presented.
The chapter provided a detailed description of the main issues relating to the algorithm design of binary field operations. Several implementation variants were analyzed by estimating the performance and storage requirements. The focus lied on the most demanding operation: the binary field multiplication. The state-of-the art method for binary field multiplication in software was introduced. In addition, a new multiplication approach that provides a better performance/storage trade-off as the standard method was presented.
This was followed by the presentation of possible ways of accelerating the implementation with hardware extensions. First, an innovate hardware/software codesign approach of virtual addressing for ECC was proposed. Thereafter a coprocessor that performs the binary field multiplication in hardware was introduced.

Chapter 4

Implementation

The performance of the proposed algorithms in Chapter 3 was proven with implementation and simulation. This chapter shows how the hardware extensions and the software were implemented. The implementation is named "Genuine verification with Elliptic Curve Cryptography One-way authentication (GECCO)".
The implementation procedure is shown in Figure 4.1. First, a golden model was written with C using the Eclipse development environment. The book "Implementing Elliptic Curve Cryptography" [42] served as primary source for the C model.
Thereafter the proposed software versions were implemented using the Ameba. Finally, the calculation was accelerated with hardware extensions.

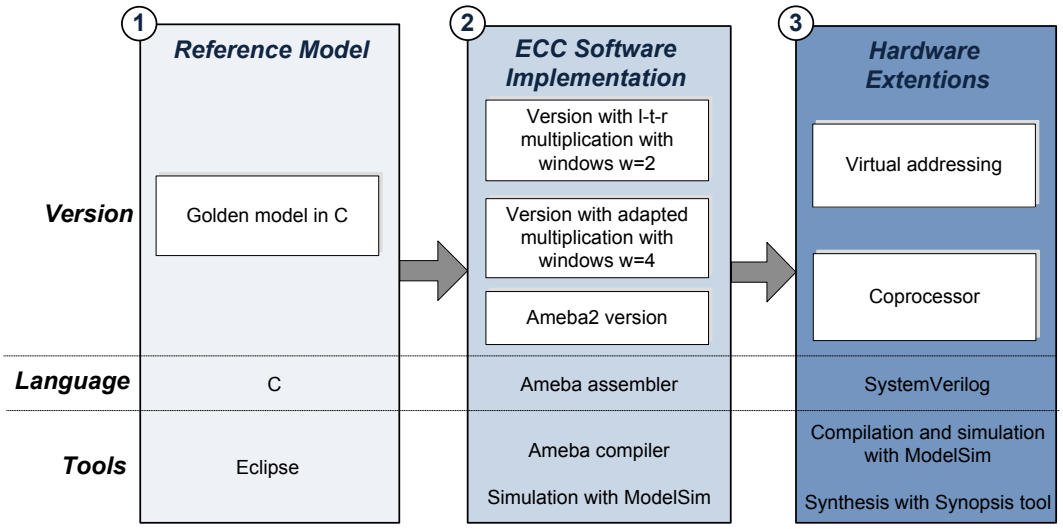


Figure 4.1: Illustration of the implemented versions and the development environment.

4.1 Development Environment

This section presents the languages and tools used during the implementation phase.

4.1.1 Software Implementation

Figure 4.2 provides an overview of the implementation tool chain.

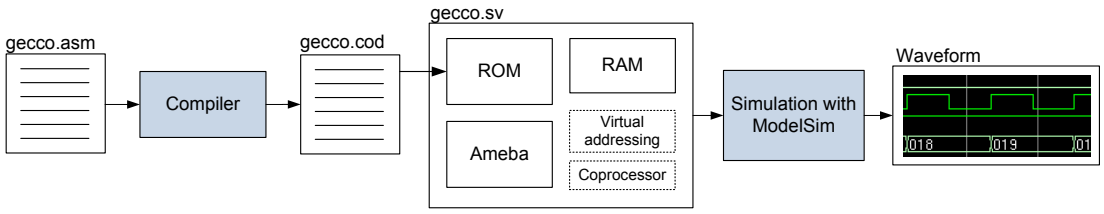


Figure 4.2: Toolchain for software implementation.

An assembler converts the source code into machine code and stores it in a *.cod* file. The machine code is loaded into the ROM and is executed by the Ameba microprocessor. The implementation was evaluated with simulation.

Common Optimization Considerations for Assembler Implementation

Several optimization techniques described in [41, p. 215–217] and [86] were considered during the implementation phase:

- **Loop Unrolling:** Loop unrolling is one of the most basic and profitable strategies to improve the performance. This extends the loops in such a way that more operations are done per iteration. Thus, the condition of the loop has to be checked less often. Hence, the number of executed instructions is reduced, but the code size is increased.
- **Duplicated Code:** Sometimes it can be effective to duplicate code. The performance can be improved by writing case-specific code fragments to reduce the number of conditionals. However, duplicated coding can involve significant code expansion.
- **Optimum Use of Registers:** An enhanced use of registers can prevent frequent accesses to RAM.

4.1.2 Hardware Implementation

The software implementation was accelerated with hardware extensions as described in Chapter 3. The hardware was implemented at Register Transfer Level (RTL) with the SystemVerilog language.
SystemVerilog extends the Hardware Description Language (HDL) Verilog and can be used for simulation, verification and synthesis. [87, p. 8]. The language can describe different levels of abstraction ranging from a system to gate level.

Modules

SystemVerilog provides modules, which are similar to Verilog modules. These modules can be seen as black boxes with inputs, outputs and internal logic [88].

Variable Assignment

To model combinatorial elements SystemVerilog offers `assign` and `always` statements [88]. For sequential elements, only `always` statements can be used.
An `always` block has a sensitivity list. A change of an variable in the sensitivity list causes the execution of the block.
Combinatorial logic is modeled with `assign` statements, which are executed continuously.

4.1.3 Simulation

The implementation was verified with simulation. Additionally, time measurements were performed with the simulation results. Therefore, the tool ModelSim [89] from Mentor Graphics was used.
ModelSim allows a clock synchronous simulation and shows the values of the different signals over the time in a waveform (see Figure 4.3).

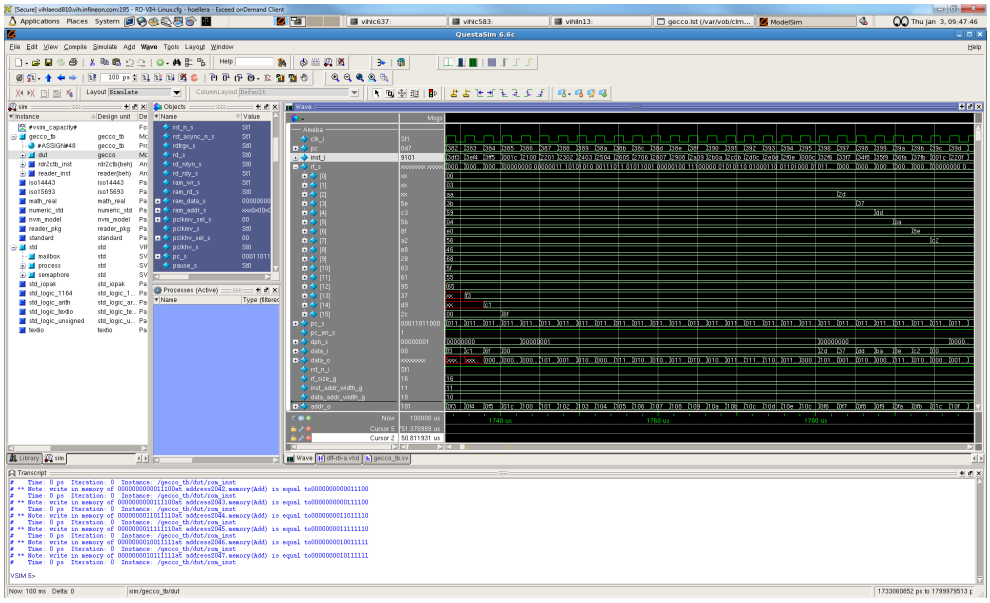


Figure 4.3: Screenshot of a ModelSim simulation including a waveform.

4.2 Software Implementation

The algorithms discussed in Chapter 3 were implemented using the Ameba.

4.2.1 Binary Field Operations

The entry point of the ECC calculation is the Montgomery multiplication. Each binary field operation was implemented in a separate subroutine. The Montgomery multiplication calls these subroutines according to Algorithm 1. This section presents details about the implementation of the binary field operations.

Addition

The addition routine adds one 21-byte element to another one. The routine steps through all words of both elements, XORs the words and stores the result. To avoid overhead caused by a loop, each of the 21 word-wise additions, were implemented separately. Thus, duplicated code was used.

Reduction

The reduction was realized with the second proposed variant using a 264-byte LUT as described in Section 3.3.5. The implementation uses registers to prevent frequent memory accesses.
To reduce the i^{th} byte of an element C, it is necessary to alter the three bytes $C[i+21]$, $C[i+20]$ and $C[i+19]$. The algorithm steps through the 21 most significant words of C starting at $C[0]$. This means that $C[i+21]$ and $C[i+20]$ are changed again at the next iteration. Hence, these values were stored in registers and to avoid accesses to memory.

Multiplication

Two variants of the binary field multiplication were implemented. The following section describes implementation details of these variants with a special focus on the word-wise additions, which mainly determine the performance.

Left-to-right multiplication with windows of width w=2 Recall that for the calculation of $C = A \cdot B$, the algorithm first calculates $B_1 = B$, $B_2 = 2 \cdot B$ and $B_3 = 3 \cdot B$ and stores these values into the RAM.
Thereafter, an addition loop, which processes each byte of the element A, is executed four times. The value of the current window of A determines the value of u. The associated B_u is added to the accumulator C. Depending on the number of iterations of the loop, the 21 bytes of B_u are added to 21 subsequent bytes of C. The address of the word of C, to which the first byte of B_u should be added, is written into a register.
To avoid pointer calculations, three subroutines were implemented. Each of the subroutine either adds B_1, B_2 or B_3 to C. These routines use hard coded values indicating the addresses of the words of B_1, B_2 and B_3. Figure 4.4 shows an extract of the subroutine adding B_1.

```
...                 ;r2...address of current processed word of C
ADD r2, #1          ;determine addr. of next word of C
LD  r11, B1_1       ;load B1[1]
LD  r12, @r2        ;load C
XOR r12, r11        ;word-wise addition
ST  r12, @r2        ;store C
...                 ;the pattern is repeated for the remaining additions
```

Figure 4.4: Assembler implementation of word-wise addition for left-to-right algorithm with windows of width w=2.

Adapted multiplication with windows of width w=4 The second multiplication algorithm was implemented as described in Section 3.3.6.
The implementation is similar to the first version. One difference is that eight elements (B_1, B_2, B_4,B_6, ..., B_{14}) are first precalculated and stored. In contrast to the previous version, a separate subroutine for each possible addition of a precalculated element would significantly expand the code size. Again, each iteration of the addition loop determines the value of u depending on the current processed word of A. The implementation of this version stores the startaddress of the required B_u in a register. The address of the word in C, where the addition of B_u should start, is also written into a register.
For the addition two subroutines were implemented. One subroutine adds B_u to C and the other one adds B_u and B_1 to C. During each iteration of the addition loop, one of these addition routines is called depending on the last bit of the processed window of A. The two registers storing the addresses are used in the subroutines. The first subroutine adds B_u to C and is called if the processed window is even. The implementation is similar to the addition subroutines of the previous presented multiplication version. However, the subroutine needs one additional instruction. The reason for this is that the address of B_u is not hard coded any longer (as shown in Figure 4.4). Thus, the address of the current processed word of B_u has to be determined for each word-wise addition (see Figure 4.5). If the processed window of A is odd, a subroutine that also adds B_1 is called. This requires two additional instructions per word-wise addition as shown in Figure 4.5.

```
;r1...address of current processed word of Bu
;r2...address of current processed word of C
ADD_B_EVEN:
...              ;add first words
;the following code sequence is repeated 20 times
ADD r2, #1       ;determine addr. of next word of C
ADD r1, #1       ;<-- add. instr. to determine addr. of next word of Bu
LD  r11, @r1     ;load Bu
LD  r12, @r2     ;load C
XOR r12, r11     ;word-wise addition
ST  r12, @r2     ;store C
...

ADD_B_ODD:
...              ;add first words
ADD r2, #1       ;determine addr. of next word of C
ADD r1, #1       ;determine addr. of next word of Bu
LD  r11, B1_1    ;<-- load B1[1] (fix address)
LD  r12, @r1     ;load Bu[1]
XOR r11, r12     ;<-- B1[1] XOR Bu[1]
LD  r12, @r2     ;load C
XOR r12, r11     ;word-wise addition
ST  r12, @r2     ;store C
...              ;the pattern is repeated for the remaining additions
```

Figure 4.5: Assembler implementation of the addition subroutines for the adapted multiplication with windows of width w=4.

Squaring

The squaring operation was implemented with two LUTs as described in Section 3.3.7. The 32-byte LUT was integrated in the free available columns of the LUT used for the reduction.
To square element A, the squaring routine starts expanding the first ten bytes using the 32-byte LUT. The remaining bytes are expanded with the interleaved reduction. To expand byte $A[i]$, the bytes $A[2i+1]$, $A[2i]$, $A[2i-1]$ and $A[2i-2]$ have to be changed. The expansion is done byte per byte. Similar to the reduction procedure, it can be observed that when scanning the next byte $A[i+1]$, only the byte $A[2i-2]$ remains unchanged. Thus, the values of $A[2i+1]$, $A[2i]$ and $A[2i-1]$ are kept in registers and are used again at the next iteration step. This approach reduces the number of memory accesses.

4.3 Implementation in Ameba2

The source code implementing the adapted multiplication with windows of width w=4, was adapted to the Ameba2 instruction set.
The sequences where a XOR follows a LD instruction are substituted by one $LDXR$ instruction. Such sequences frequently occur at the addition loop of the field multiplication. Furthermore, the Ameba2 offers a shift-right instruction (SR). This instruction was mainly used to accelerate the squaring operation.

4.4 Virtual Addressing

The address logic for the VA was implemented in the top module. It manipulates the output address of the microprocessor before forwarding it to the RAM. The address conversion is influenced by several parameters. The VA implementation stores these parameters in registers. Table 4.1 shows a list of these registers. The values of the registers are set by the microprocessor. For the setting of each parameter a specific address is reserved. Whenever the microprocessor writes to such an address, the output is stored in the associated register.

Register name	*Width [Bits]*	*Purpose*
`elem1_v`	8	Start address of VE 1
`elem2_v`	8	Start address of VE 2
`offset_s`	5	Positive offset added when addressing one of the B_u's
`neg_offs_s`	4	Negative offset added when addressing one of the B_u's
`offsetC_s`	5	Positive offset added when addressing the accumulator C.
`addr_mode_s`	1	Indicates the window.

Table 4.1: Registers for the virtual addressing mechanism.

As described in Chapter 3, the virtual addressing (VA) concept includes two virtual elements (VEs) within a range of 21 words. Each physical 21-byte address range is associated with an element number. Whenever such an element number is set, the physical

start address of the element is determined. For example, if the element number is two, the start address is 21, if the element number is four the start address is 42, and so on. If the element number of the first VE is set, also the parameter *addr_mode* has to be considered. This parameter defines which bits are used to determine the start address.

The address logic was implemented as described in Section 3.4. Therefore, an `assign` statement that defines the input address of the RAM was used as shown in Figure 4.6. The logic first analyses the range of the given address. If the address is a virtual address, the corresponding physical address is calculated.

```
//ram_addr_s  ... input address of the RAM
//addr_s      ... output address of the Ameba
assign ram_addr_s  =
  //upper address range
  (addr_s >= 10'h01E7 & addr_s <= 10'h01FB) ?  //VE 1
    ({addr_s[8], elem1_v[7:0] + (addr_s[4:0] - 5'h07)}) +
    offset_s - neg_offs_s :

  (addr_s >= 10'h01A8 & addr_s <= 10'h01D1) ?  //accumulator C
     addr_s + offset_c_s :

   //lower address range
  (addr_s >= 10'h00E7 & addr_s <= 10'h00FB) ?  //VE 1
    ({addr_s[8], elem1_v[7:0] + (addr_s[4:0] - 5'h07)}) :

  (addr_s >= 10'h00D2 & addr_s <= 10'h00E6) ?  //VE 2
    ({addr_s[8], ecc_elem2_v[7:0] + (addr_s[4:0] - 6'h02)})   :

   addr_s;    //not a virtual address --> forward unchanged address
```

Figure 4.6: Implementation of the virtual address mapping. The output address of the microprocessor is manipulated before forwarding to the RAM.

4.4.1 Software Implementation with Virtual Addressing

The software implementation was adapted to take advantage of VA as described in Section 3.4. The number of instructions required for the word-wise additions are reduced as shown in Figure 4.7.

```
...
ADD_B1_E_7: LD   r11, VE_7  ;addition with memory accesses
            LD   r12, C_7   ;load 8th word of C and XOR with Bu
            XOR  r11, r12
            ST   r11, C_7   ;store C

ADD_B1_E_7: LD   r11, VE_8  ;addition with register
            XOR  r0, r11
...
```

Figure 4.7: Assembler implementation of addition subroutines for VA as described in Algorithm 11.

4.5 Coprocessor

Finally, the most hardware-intense version was implemented. Therefore, the software implementation is altered to use a coprocessor for the field multiplication.
The coprocessor was implemented as a dedicated SystemVerilog module. To store temporary values, the registers listed in Table 4.2 were used.

Register name	*Width [Bits]*	*Purpose*
`addr_a_s`	8	Stores the address of current processed byte of element A
`addr_b_s`	8	Stores the address of current processed byte of element B
`addr_c_s`	8	Stores the address of current processed byte of result C
`c_s`	16	Stores the partial product of an 8x8-bit multiplication
`a_s`	8	Stores the value of the current processed byte of factor A
`b_s`	8	Stores the value of the current processed byte of factor B
`red_r0_s`	8	Stores the temporary result of the reduction function R_0
`red_r4_s`	6	Stores the temporary result of the reduction function R_4
`red_r3_s`	8	Stores the temporary result of the reduction function R_3
`half_s`	1	Shows if the first or the second outer loop of the Comba's multiplication is executed.
`counter_i`	5	Counter for the outer loop of the Comba's multiplication
`counter_j`	5	Counter for the inner loop of the Comba's multiplication
`state`	6	Stores the current state of the FSM.
`next_state`	6	Stores the future state of the FSM.

Table 4.2: Registers used by the coprocessor.

4.5.1 Combinatorial Part

The combinatorial part was implemented with assign statements. Figure 4.5.1 shows the implementation of the 4x8-bit binary multiplication.
The reduction functions are implemented in a similar way.

```
//4x8 bit multiplication
assign c_w[0]=(a_s[0] & b_s[0]);
assign c_w[1]=(a_s[0] & b_s[1])^(a_s[1] & b_s[0]);
assign c_w[2]=(a_s[0] & b_s[2])^(a_s[1] & b_s[1])^(a_s[2] & b_s[0]);
assign c_w[3]=(a_s[0] & b_s[3])^(a_s[1] & b_s[2])^(a_s[2] & b_s[1])^(a_s[3] & b_s[0]);
assign c_w[4]=(a_s[0] & b_s[4])^(a_s[1] & b_s[3])^(a_s[2] & b_s[2])^(a_s[3] & b_s[1]);
assign c_w[5]=(a_s[0] & b_s[5])^(a_s[1] & b_s[4])^(a_s[2] & b_s[3])^(a_s[3] & b_s[2]);
assign c_w[6]=(a_s[0] & b_s[6])^(a_s[1] & b_s[5])^(a_s[2] & b_s[4])^(a_s[3] & b_s[3]);
assign c_w[7]=(a_s[0] & b_s[7])^(a_s[1] & b_s[6])^(a_s[2] & b_s[5])^(a_s[3] & b_s[4]);
assign c_w[8]=                  (a_s[1] & b_s[7])^(a_s[2] & b_s[6])^(a_s[3] & b_s[5]);
assign c_w[9]=                                   (a_s[2] & b_s[7])^(a_s[3] & b_s[6]);
assign c_w[10]=                                                    (a_s[3] & b_s[7]);
```

Figure 4.8: SystemVerilog implementation of the 4x8-bit multiplication.

4.5.2 Control Logic

The control logic was realized with an FSM.The start addresses of the processed elements are stored in the registers `addr_a_s`, `addr_b_s` and `addr_c_s`. If the address of the element B is updated, the FSM starts.
Figure 4.9 shows the state machine implementing the Comba's multiplication as shown in Algorithm 13. The coprocessor reads the values of the elements directly from RAM and stores the result back to RAM by using the given addresses. `STATE_MUL_LOW` handles the processing of the first outer-loop and `STATE_MUL_HIGH` handles the second outer-loop of the Comba's algorithm. The latter implements the interleaved reduction.
The states `STATE_M1` and `STATE_M2` determine the 8x8-bit multiplication by using the combinatorial 4x8-bit multiplication.
The calculation of the last byte has to be handled differently. The reduction of the last byte is performed as described in the lines 14 to 17 of Algorithm 13 and isrealized with the states `STATE_RED0`, `STATE_RED19`, `STATE_RED19_STORE`, `STATE_RED20` and `STATE_R20_STORE`.
Finally, the FSM goes into an idle state and the microprocessor is resumed.

4.6 Summary

In this chapter, the implementation aspects were outlined. The languages and tools used for the implementation of hard- and software were presented.
Furthermore, implementation details of the binary field operations with the Ameba were described. It has been shown how the different implementation options of the field multiplication influence the word-wise addition, which occurs frequently. The adapted multiplication version needs more instructions to perform one word-wise addition.
Thereafter, it is shown how the VA mechanism was implemented in SystemVerilog.
Finally, the coprocessor implementation was described. A detailed description about the implementation of the combinatorial part and the control logic was provided.

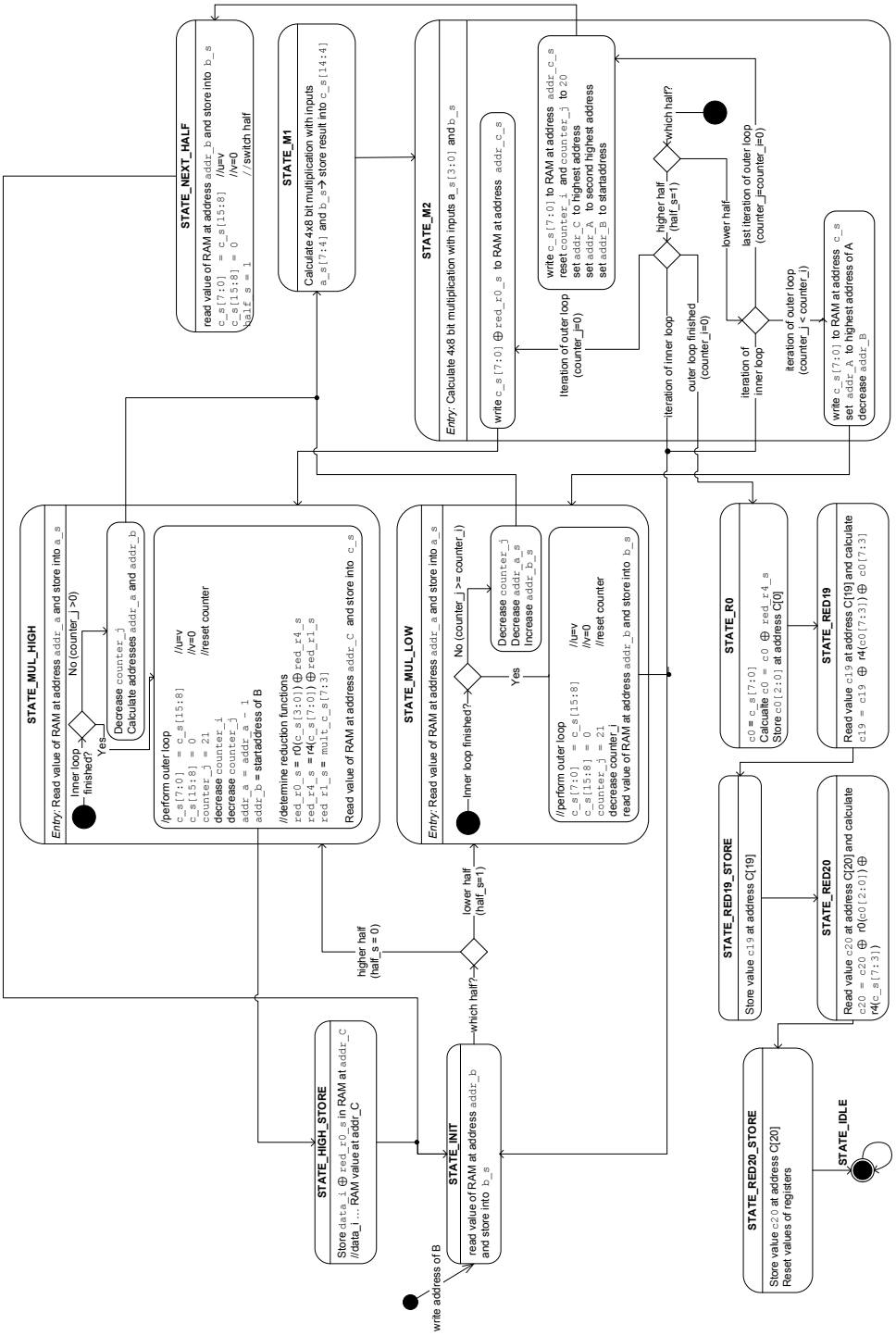


Figure 4.9: FSM of the coprocessor.

Chapter 5

Results

This chapter presents an evaluation of the different implementation variants and compares the performance and storage requirements to similar implementations available in literature. The execution times are determined with simulation and are scaled to clock cycles using the Equation 1.1 described in Chapter 4.

5.1 Software Implementation

This section presents the results of the software implementation without hardware extensions. First, the performance and storage requirements of each binary field operations is analyzed. Then the requirements of the whole implementation are regarded.

5.1.1 Binary Field Operations

The measured runtimes and storage requirements of the binary field operations addition, reduction, multiplication and squaring are described below.

Addition

The field addition is the field operation that requires the least amount of storage and execution time. The addition needs about 115 bytes of code and the execution of one addition is done in about 150 cycles. Te runtime is data independent.

Reduction

The implemented reduction variant does not depend on the input data. Thus, every reduction requires the same amount of time. The measured execution time is about 590 cycles. This measured time is close to the estimation in Chapter 3. The execution is done about 6% faster as approximated.
The code size is about 235 bytes and requires a 264 byte LUT.

Multiplication

This section compares the two implemented field multiplication variants.

Multiplication with windows of width w=2

The three precalculated elements need 105 bytes of RAM. Additionally, 41 bytes of RAM are needed to store the result.
The code size of the implementation is 1,745 bytes. Compared to the other field operations, this storage requirement is by far the highest.
The implementation first performs the precalculation, executes the addition loop four times and shifts C three times left by two bits. The simulation of this field multiplication is shown in Figure 5.1. The waveform shows the accumulator in RAM, which stores the result. It can be seen clearly that the addition loops frequently write to the RAM and that they count for the majority of runtime.

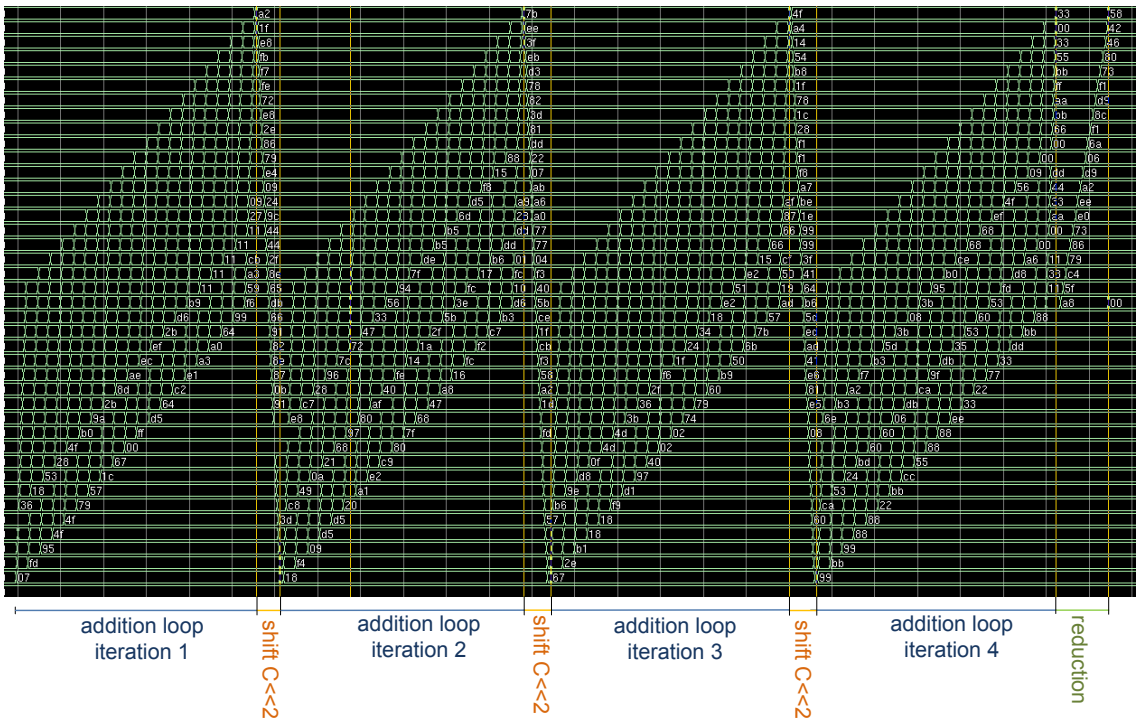


Figure 5.1: Simulation of one left-to-right multiplication with windows w=2. The waveform shows the content of the accumulator in the RAM storing the result.

The average runtimes of the calculation steps are outlined in Table 5.1. The word-wise additions can be done relatively fast by using three separate subroutines. This accelerates the addressing of the B_u's. This is the reason why the measured runtime is about 50 % faster as estimated in the design phase in Section 3.3.6. The total average time required for one field addition is 9,750 cycles.
The execution time of the precalculation and shifting steps are data independent. However, the loop addition depends on the scanned window. If the window is zero, the addition of a precalculated element can be skipped. This causes data dependency. To evaluate the dependency, the runtimes of 20 field multiplications operating on different elements are measured. The highest measured execution time is about 10,550 cycles and the fastest execution is done in about 9,630 cycles. Furthermore, the standard deviation is calculated

to show how widely the measured times are dispersed from the average execution time [90]. The value of the standard deviation is determined with the following equation, where $\overline{x}$ is the average time, x is the sample and n is the number of measurements [90]:

$$s = \sqrt{\frac{\sum x - \overline{x}}{n-1}}. \tag{5.1}$$

The calculation of the standard deviation for the measured execution times gives $s = 100$. This shows that the spreading of the execution times is within relative small ranges.

Calculation step	*Runtime [clock cycles]*	*# executions*	*Total runtime [clock cycles]*	*% of* runtime
Precalculation	250	1	250	2
Loop Addition	2,150	4	8,600	88
Shift C	300	3	900	10
Total runtime			**9,750**	

Table 5.1: Performance of the left-to-right multiplication with windows of width $w = 2$.

Adapted multiplication with windows of width w=4

It is expected that the performance of this field multiplication variant is better than the field multiplication with windows of width $w = 2$.
However, the method needs more RAM. It requires 210 bytes RAM, which is 105 bytes more than the previous presented version. The code size can be reduced by 250 bytes, since less duplicated code for the loop addition is used.
An analysis of the runtime of this method is shown in Table 5.2.

Calculation step	*Runtime [clock cycles]*	*# executions*	*Total runtime [clock cycles]*	*% of* runtime	*Speed-up*[1]
Precalculation	780	1	780	10	0.36
Loop Addition	3,290	2	6,580	86	1.47
Shift C	300	1	300	4	3.5
Total runtime			**7,640**		1.43

Table 5.2: Performance measures of adapted left-to-right polynomial multiplication with windows of width $w = 4$.
[1]The speed-up relates to the left-to-right multiplication with windows of with $w = 2$ (see Table 5.1).

The addition in the loop is about 1.47 times slower. This is due to the additional instructions required for one word wise addition. However, the addition loop has to be executed half as often. This results in a total performance speed-up of about 1.28 for the multiplication. The implementation uses a duplicated code for the word-wise addition. Thus, the runtime is about 25% faster as approximated in Section 3.3.6.
Again, the execution time of the addition step depends on the value of the processed data. Compared to the standard left-to-right multiplication with windows $w = 2$, additional data

dependency is introduced by treating the last bit of a window differently. If this bit is one, additional calculations have to be performed. This increased data dependency can be seen by looking at twenty different execution time measurements. The highest measured time is about 6,790 cycles and the lowest about 5,565 cycles. The calculation of the standard deviation according Equation 5.1 gives $s = 260$. This shows that the measured execution times are more widespread. One has to keep this in mind, when considering side channel attacks, which exploit the data dependency of execution times.

Squaring

The total average runtime measured for one squaring operation is 1,560 cycles. The implementation is about 35% slower as estimated during the design phase (see Section 3.3.7). This can be reasoned by the control overhead. The runtime of a square operation does not depend on the data.
The simulation of one square operation is shown in Figure 5.2.

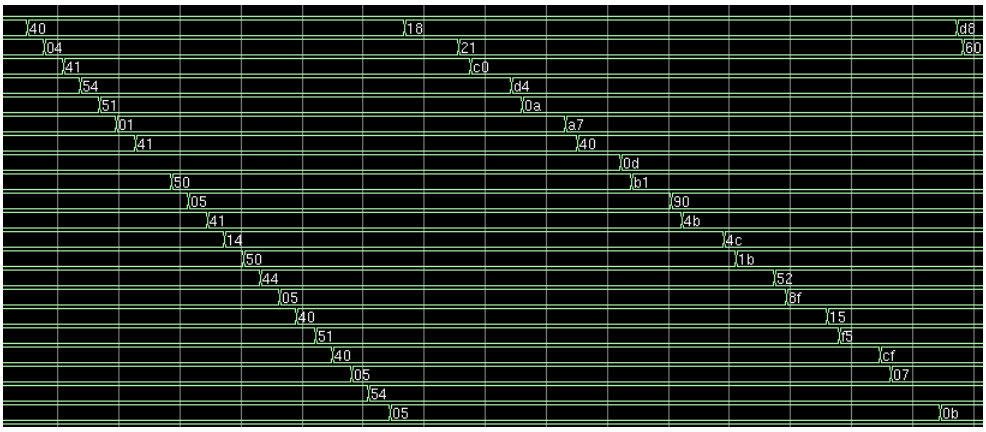


Figure 5.2: Simulation of one square operation. The waveform shows the content of an element in RAM, which is squared.

The waveform shows the content of an element in RAM, which is squared. It can be seen that write operations are performed on all words of the RAM twice. First, the intermediate results computed from the squaring of the first half of the element are written. Thereafter, the upper half is squared with the interleaved reduction. This is the reason why the squaring of the upper half takes longer than the squaring of the lower half. Finally, the most significant five bits are reduced, which requires two write operations.
The code size for storing the square procedure is about 300 bytes. Additionally, the two LUTs require 160 bytes.

5.1.2 Montgomery multiplication

The Montgomery multiplication calls the binary field operations. The code size of the Montgomery multiplication implementation is about 170 byte. The algorithm requires ten 21-byte elements stored in RAM.
The code size of the whole implementation using the adapted multiplication method is

about 2.3 kBytes without LUTs. Figure 5.3 shows the composition of the required ROM size. It can be seen that the field multiplication is the determining factor regarding the storage requirements.
The average runtime of a Montgomery multiplication is about 9.4 million cycles. Table 5.3 and Figure 5.3 show the partitioning of the runtime. It can be seen clearly that the multiplication is also the determining factor for the overall performance. The polynomial multiplication and reduction, which is only needed for the multiplication, require more than 85% of the runtime.

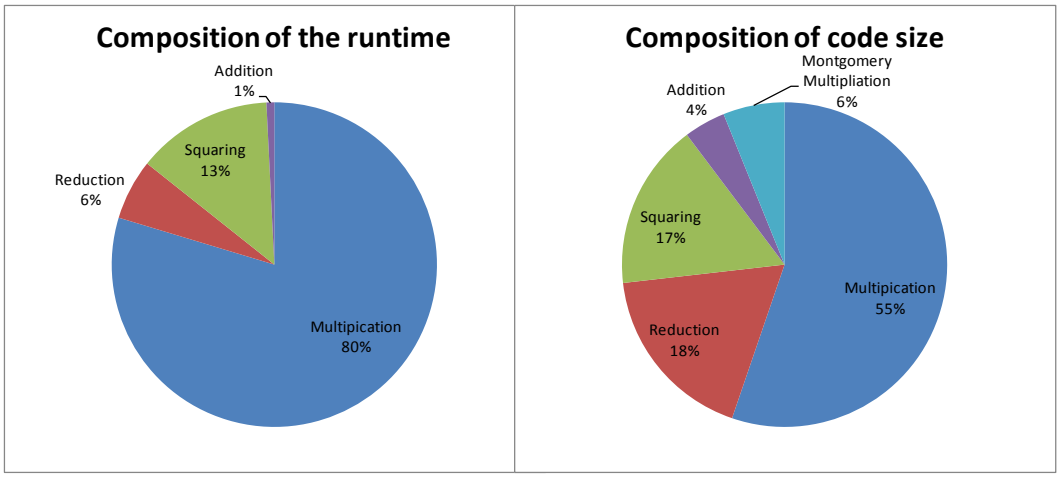


Figure 5.3: Composition of runtime and code size. The code sizes of the binary field operations include the associated LUTs.

Field Operation	*Runtime [clock cycles]*	*# executions per Montgomery mult.*	*Total runtime* [clock cycles]	*% of* total runtime
Multiplication	7,660	978	7,491,480	79.7
Reduction	580	978	567,240	6.0
Squaring	1,560	815	1,271,400	13.5
Addition	150	489	73,350	0.8
Total runtime for one Montgomery multiplication			**9,403,470**	

Table 5.3: Partitioning of runtime for one Montgomery multiplication with Ameba.

5.2 Virtual addressing

The virtual addressing approach requires additional hardware. The area of this hardware is estimated to be about 1kGE. Furthermore, the assembler code was adapted and extended. However, the additional code requires only 46 bytes of ROM.
At the expense of this small area overhead, a significant performance improvement was achieved. The composition of the runtime is shown in Table 5.4.

The virtual addressing approach reduces the time for inner loop addition by almost a half. This is the most expensive step of the field multiplication. The total runtime of the

Calculation step	*Runtime [clock cycles]*	*Speed-up*[2]
Precalculation	750	1.04
Loop Addition	4,230	1.56
Shift C	300	1
Total runtime	**5,280**	1.45

Table 5.4: Performance measures of polynomial multiplication virtual addresses.
[2]The speed-up relates to the software implementation using the left-to-right multiplication with windows of with $w = 4$ (see Table 5.2).

multiplication with virtual addressing is about 44% faster compared to the multiplication without any hardware acceleration.
Figure 5.4 shows the simulation of one adapted field multiplication with virtual addressing. Compared to the previous multiplication variants (see Figure 5.1), fewer write operations during one addition loop are performed. This can be achieved since registers are used to store the intermediate values as shown in Figure 3.12.

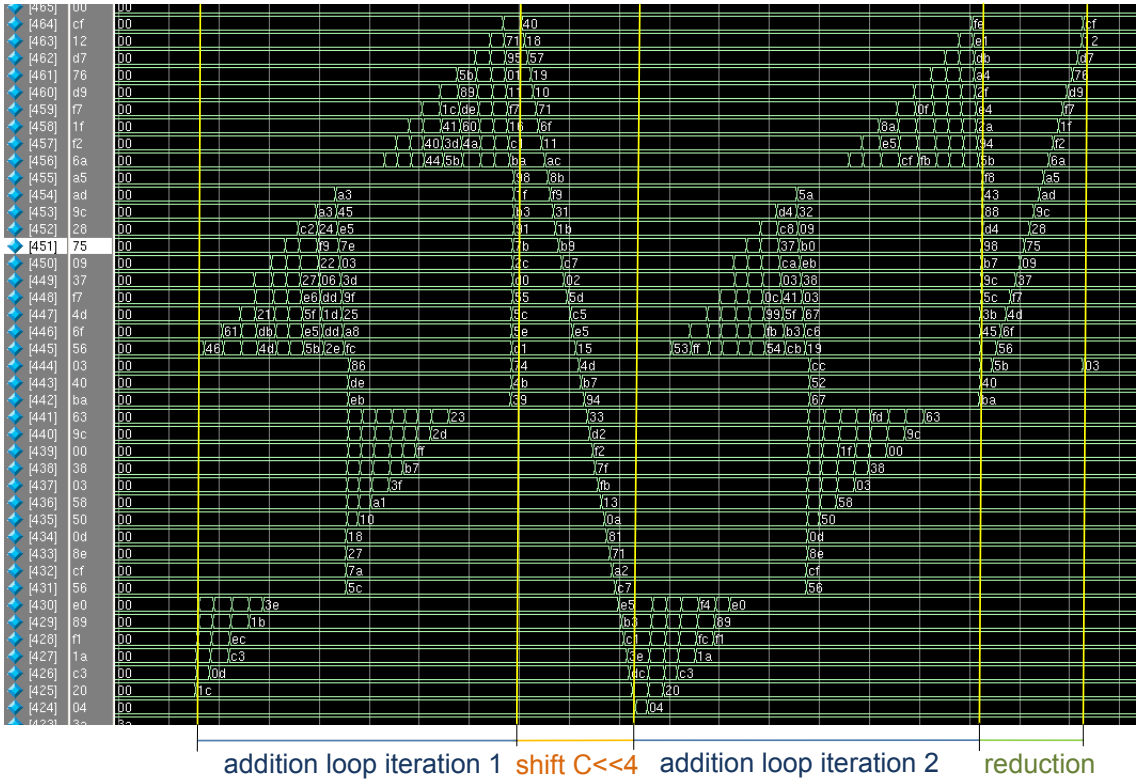


Figure 5.4: Simulation of one field multiplication with virtual addressing. The waveform shows the content of those words in the RAM, where the result is written to.

The binary field operations reduction and addition were accelerated as shown in Table 5.5. The virtual addressing approach achieves an execution time for one Montgomery multiplication, which is 23% faster than without virtual addressing.

Field Operation	*Runtime [clock cycles]*	*# executions per Montgomery mult.*	*Total runtime [clock cycles]*	*Speed-up*[3]
Multiplication	5,280	978	5,163,840	1.45
Reduction	570	978	557,460	1.02
Squaring	1,540	815	1,255,100	1.01
Addition	90	489	44,010	1.67
Total runtime for one Montgomery multiplication			**7,020,410**	

Table 5.5: Performance measures of the virtual addressing implementation.
[3]The speed-up relates to the software implementation using the adapted field multiplication (see Table 5.3).

5.3 Ameba2 implementation

The extended instruction set of Ameba2 allows combining several consecutive instructions to one single instruction. This decreased the storage requirement for the program code by 171 bytes.
Furthermore, the performance was improved. Mainly the word-wise additions were accelerated by the *LDXR* instruction. The execution time of one addition loop is about 33% faster to the previous presented version. Table 5.6 outlines the execution times of every step required for one field multiplication.

Calculation step	*Runtime [clock cycles]*	*Speed-up*[3]
Precalculation	700	1.07
Loop Addition	2,820	1.5
Shift C	300	1
Total runtime	**3,820**	1.38

Table 5.6: Performance measures of polynomial multiplication virtual addresses and Ameba2.
[3]The speed-up relates to Ameba implementation with VA (see Table 5.4).

Table 5.7 summarizes the measured runtimes of the finite field operations using Ameba2 and compares these runtimes to the previous version. The multiplication and addition operation were mainly accelerated with the *LDXR* instruction. However, Ameba2 also enables a significant performance improvement of the squaring algorithm. This is due to the implementation of the missing shift-right instruction.
In sum, these two additional instructions lead to a total runtime, which is about 27% faster as the previous implementation.

5.3.1 ROM Storage Requirements

The implementation of the Montgomery multiplication using the adapted field multiplication and virtual addressing with Ameba2 requires in total about 4 kBytes ROM. About 3kBytes are required to store the program and the LUTs need about 1kByte (see Figure 5.5). The field multiplication requires by far the most size of the ROM.

Field Operation	*Runtime [clock cycles]*	*# executions per Montgomery mult.*	*Total runtime [clock cycles]*	*Speed-up*[4]
Multiplication	3,820	978	3,735,960	1.38
Reduction	500	978	489,000	1.14
Squaring	1,070	815	872,050	1.44
Addition	75	489	36,675	1.2
Total runtime for one Montgomery multiplication			**5,133,685**	1.37

Table 5.7: Performance measures of binary field operations virtual addresses and Ameba2. [4]The speed-up relates to Ameba implementation with virtual addresses (see Table 5.5).

The startaddress of a LUT has to be a multiple of 256 and is organized in two halves. This introduces unused storage gaps as shown in Figure 5.5. The total size of unused ROM is about 850 bytes.

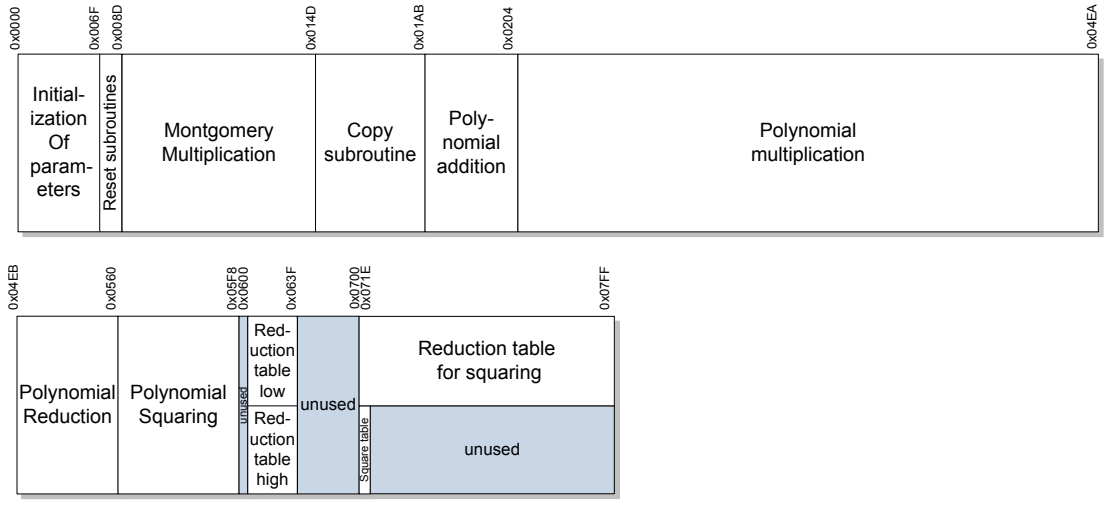


Figure 5.5: Partitioning of program code implementing Montgomery multiplication using the adapted left-to-right multiplication with windows of size $w = 4$ and virtual addressing.

5.4 Coprocessor

This section presents the area and performance requirements of the coprocessor implementation.

5.4.1 Area

To synthesize the coprocessor the Synopsis Design Compiler tool was used. The values given by the tool only represent the space needed for the standard cell area alone. Additionally about 20% for routing has to be considered.
The result of the synthesis shows that the coprocessor requires about 2.13 kGE area. Table 5.8 shows how the combinational and non-combinational parts influence the area.

	Area [kGE]	*[%] of total cell area*
Combinational area	1.416	79
Non- combinational area	364	21
Total cell area	1.78	
Total area incl. routing	2.13	

Table 5.8: Combinational and non-combinational area of the coprocessor. The total area includes an overhead of 20% for the cell interconnections.

Since the field multiplication with interleaved reduction is outsourced to the coprocessor, the assembler code for the multiplication and reduction is not required anymore. This reduced the code size to about 1 kByte.
Additionally, the virtual addressing for the field multiplication is not needed either. This means that just the virtual addressing that has an effect on the lower 256 byte RAM is implemented. This reduces the area of VA to approximately 0.4 kGE. The RAM required to store temporary values for the field multiplication can also be omitted. The total required RAM storage amount is now 214 bytes.

5.4.2 Performance

As expected, the performance of the hardware executing one field multiplication is unmatched by the other implementation versions.
Figure 5.6 shows the simulation of one multiplication with the coprocessor. It can be seen that the calculation of the most significant and least significant words is fast. The calculation of the words in the middle takes longer, since the inner loop is executed more often (also see Figure 3.19). The coprocessor writes twice to each of the 21 words in RAM, which store the result. First, the intermediate result of the first outer loop is written and thereafter the second outer loop writes the final result.
Every calculation performed by the coprocessor needs the same amount of time. The coprocessor is able to calculate the field multiplication with the interleaved reduction in 1,830 cycles. The previous fastest implementation version (Ameba2 with virtual addressing) required 4,320 cycles for multiplication and reduction. This means that the coprocessor

achieves a runtime for field multiplication, which is about 2.4 times faster than the previous version.
How the field operations influence the total performance is shown in Table 5.9.

Operation	*Runtime [clock cycles]*	*Speed-up*[4]
Multiplication with interleaved reduction	1,830	2.4
Squaring	1,070	1
Addition	75	1
Montgomery multiplication	**2,698,460**	1.9

Table 5.9: Performance measures of binary field operations using Ameba2 and the coprocessor for multiplication.
[4]The speed-up relates to the software implementation with virtual addresing (see Table 5.7).

It can be seen that the percentage of the time required for a multiplication is reduced compared to the previous versions. A Montgomery multiplication can be calculated in about 2.7 million cycles using the coprocessor. This is about 47 % faster than the fastest version without the coprocessor.

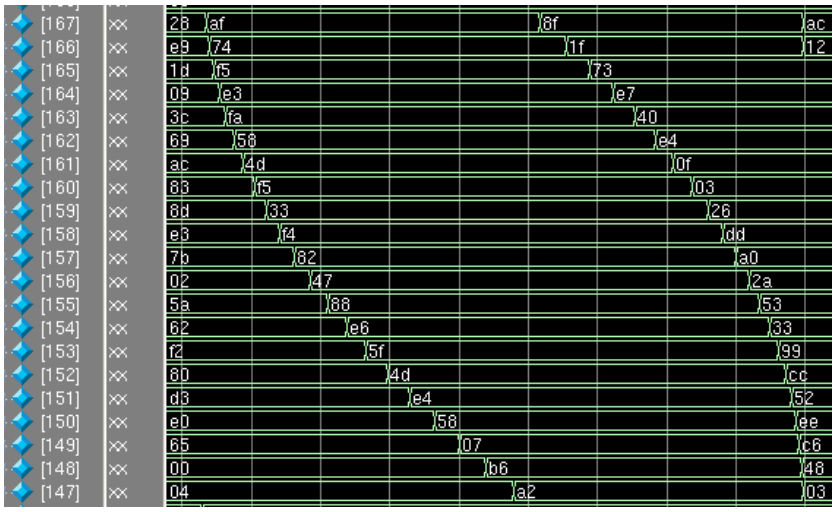


Figure 5.6: Simulation result of one mutliplication with the coprocessor. The waveform shows the content of those words in the RAM, where the result is written to.

5.5 Comparison of Implementation Variants

Table 5.10 and Figure 5.7 summarizes the performance of the Montgomery multiplication using the different previously presented methods.

Implementation	*Code Size*		*RAM*		*Extensions*	*Area*	*Runtime*
Variant	*[Byte]*	*[kGE]*	*[Byte]*	*[kGE]*	*[kGE]*	*[kGE]*	*[MCycles]*
L-t-r mult.	3,205	3.41	318	4.1	-	7.51	12.1
Adapted mult.	3,123	3.32	423	5.46	-	8.92	9.4
Virtual addr.	3,840	4.09	423	5.46	1	10.55	7.0
Ameba2$^{\#}$	3,594	3.83	423	5.46	1	10.29	5.1
Coprocessor	1,023	1.09	214	2.76	2.53*	6.38	2.7

Table 5.10: Comparison of implementation variants.
*coprocessor and virtual addressing for lower half of the RAM
#Ameba2 and virtual addressing

Figure 5.7 shows, that the improved variants mainly reduced the execution time of the most expensive field operation - the multiplication. The fastest implementation is about 4.5 times faster than the slowest one. The area requirements are summarized in Figure 5.8.
Furthermore, the area of the implementation variants is outlined in Table 5.10. The area of the microprocessor was not considered. The adapted field multiplication requires 95 bytes of RAM more than the left-to-right multiplication with windows of width $w = 2$. However the execution time is decreased by about 22%.
Adding VA causes a small additional area overhead, but the performance improves by a factor of about 1.34.
Using two additional instructions of Ameba2 improves the computation performance even further to 5.1 MCycles. In sum, the virtual addressing and Ameba2 extensions improve the performance by a factor of 1.9.
Table 5.10 shows that the implementation always includes a trade-off between performance and area. The faster the execution gets, the more area is needed. However, the implementation with a coprocessor is an exception. The outsourcing of the field multiplication significantly lowers the storage requirements. The area reduction caused by the lower storage requirement is higher than the additional area needed by the proprietary hardware. This makes that the coprocessor variant a very good option. It is unmatched in terms of performance and area.

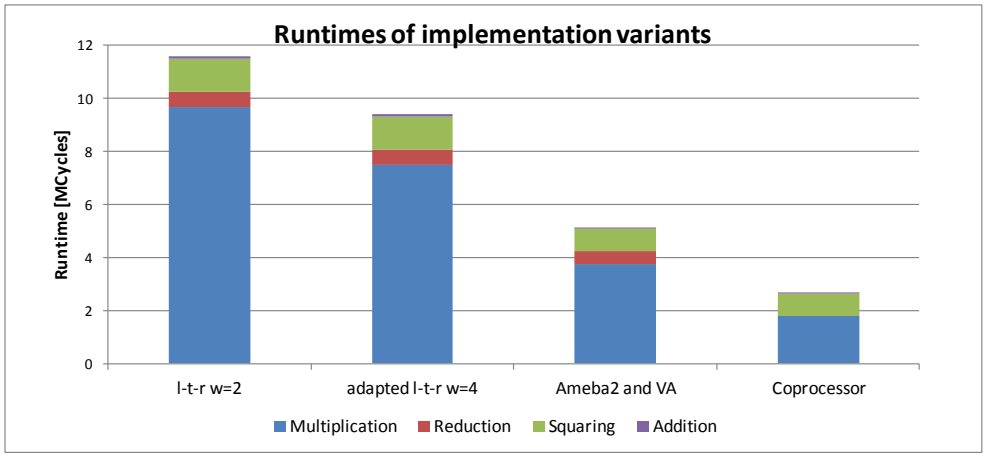


Figure 5.7: Comparison of runtimes of different implementation variants.

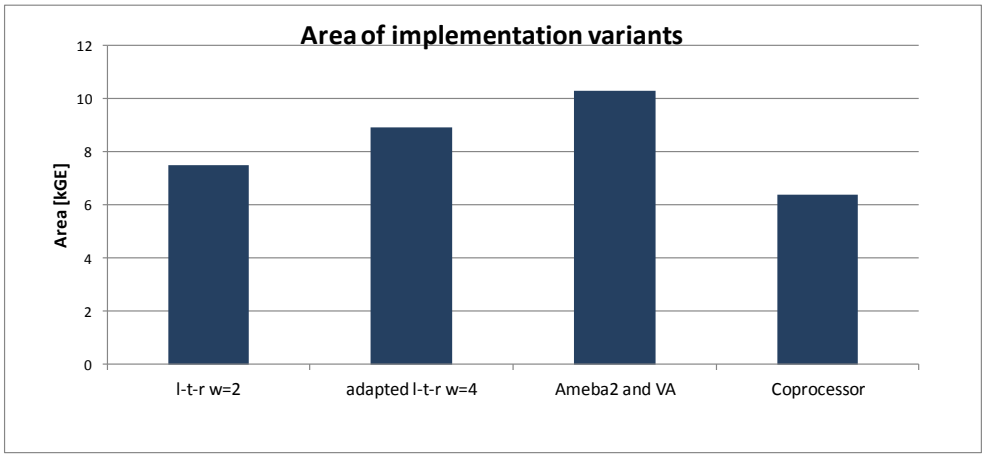


Figure 5.8: Comparison of area requirements for the different implementation variants. The areas include the area for storage (RAM and ROM) and extensions.

5.6 Comparison to Implementations in Literature

This section compares the implementation presented in this thesis with available software implementations in literature.

5.6.1 Comparison of Low-area Processor Architectures

Most of the 8-bit ECC software implementations found in literature are based on the ATmega128 processor. Compared to the 30 instructions of the Ameba, the ATmega128 features 133 instructions and is therefore larger and more powerful. Among these 133 instructions, there is also an area-intensive on-chip multiplication. Furthermore, this processor provides a larger register set consisting of 32 GPRs, which is twice as large as the register set of the Ameba.
The most similar approach compared to this work was published by Wenger et al. [59].

Their architecture also targets resource-constrained RFID tags. They presented a clone of the ATmega128 called JAAVR. The silicon-footprint of JAAVR is 6.5 kGE, which is more than two times larger than the Ameba.

5.6.2 Comparison to Prime Field Implementations

The ECC hardware/software architectures designed and implemented during this work compares favorably with works described in literature. Table 5.11 highlights different 8-bit ECC software implementations. The table does not consider the coprocessor implementation, since a comparison to other software implementations without hardware acceleration would not be meaningful.

Implementation Variant	*GF*	*ROM [kBytes]*	*RAM [Bytes]*	*Runtime [MCycles]*
Guara et al. [56]	p_{160}	3.6	280	6.48
Wenger et al. [59]				
slowest version	p_{160}	3.86	384	35.1
fastest version	p_{160}	7.76	384	13.0
Yan et al. [43]	2^{163}	11.6	820	111
[43] with ISE	2^{163}	-	-	13.5
Szczechowiak et al. [65]	2^{163}	32.4	1741	15.95
Kargl et al. [58]	2^{167}	11	>588	6.1
This work				
Software implementation with the Ameba	2^{163}	3.05	423	9.7
Ameba2 and virtual addressing	2^{163}	3.51	423	5.1

Table 5.11: Comparison to implementations available in literature.

All listed implementations base on the ATmega128 processor [57]. The implementations over prime fields exploit the ATmega128's hardware multiplier.
Guara et al. [56] reached considerable performance results. However, they used a Non-Adjacent Form (NAF)-method for point multiplication, which is in contrast to the Montgomery multiplication not resistant against side channel attacks [59].
The most similar approach compared to this work was published by Wenger et al. [59], since their target application is also RFID. Their slowest point multiplication is implemented in C with operand scanning. Their fastest presented version was optimized in assembler and used the operand-caching, as described in [60]. The approach presented in this work requires comparable memory resources. However, their execution time for one point multiplication is about 3.3 MCycles higher. The Ameba2 with virtual addressing is even able to execute an ECC calculation about 2.6 times faster as the presented JAAVR implementation. Although the area of the Ameba2 and virtual addressing (∼4kGE) is smaller than the area of the JAAVR processor (6.5 kGE).

5.6.3 Comparison to Binary Field Software Implementations

The ECC calculations in [65] and [58] were performed over binary fields. Consequently, they could not take advantage of the multiplier. Szczechowiak et al. used a comb method

with pre-computed points for point multiplication, which causes high memory requirements.
The implementation of Kargl et al. [58] is comparable to the implementation presented in this work, since they use a similar field and point multiplication method. By fully utilizing the 32 registers available on the ATmega128, they reduced the number of memory accesses and reached a runtime of 6.1 MCycles. Thus, the execution time is faster than the software approach presented in this thesis, but requires significantly more ROM.

5.7 Summary

In this chapter the implementation versions were evaluated. The total execution time of the enhanced version was mainly reduced by accelerating the expensive field multiplication. The analysis of the performance and area of the implementation variants showed that the versions, which offer better performances introduce a bigger area. Thus, the choice of an implementation variant includes a trade-off between performance and area.
It is remarkable that the coprocessor implementation variant constitutes an exception. The availability of a hardware multiplier influenced the algorithm design. This resulted in the lower storage requirements, which made it possible that the most hardware-intense version requires the smallest silicon footprint. As expected, the performance of the coprocessor version is unreached by the other proposed implementation versions.
Finally, it has been shown that the approaches presented in this thesis compare well to other 8-bit software implementations available in literature.

Chapter 6

Conclusion

The ambition of this thesis was to answer the question "Is RFID ready for software-based ECC?". Up to this date not much research has been done in the field of software-based ECC for RFID. The thesis proposes several hardware/software implementation variants and evaluates the approaches in terms of area and performance.
Different algorithm variants were specifically designed for the implementation on a light-weight 8-bit microprocessor for RFID. First, two software-based field multiplication variants were evaluated: The first one is the left-to-right multiplication with windows, which is the state-of-the art approach. The other one was designed by the author of this work to improve the storage/performance trade-off of the standard method.
Furthermore, an innovative hardware/software codesign approach was presented by introducing virtual addressing for ECC. The proposed hardware acceleration decreases the execution time of one ECC calculation by about a quarter, while introducing very little area overhead.
Next, the implementation was adapted to an enhanced instruction set of the Ameba. The evaluation shows that the availability of only two additional instructions, leads to a 27 percent faster execution time.
Finally, the binary field multiplication was outsourced to a hardware coprocessor. The availability of a certain hardware functionality has an impact on the algorithm design. The change of the binary field multiplication algorithm reduced the size of the needed RAM storage. Furthermore, outsourcing of functionalities reduced the required ROM for storing the machine code to one third. The impact on the storage requirements makes this most hardware-intense version to be the option with the lowest silicon footprint. Additionally, the execution time of the proposed coprocessor version is nearly twice as fast as the virtual addressing approach. Thus, this implementation variant is unmatched in terms of area and performance.
The performance and storage requirements of the presented implementations compare favorable to other 8-bit software ECC implementations in literature. The version with only a little additional hardware (Ameba2 and Virtual Addressing) needs about 5.1 million clock cycles for one authentication. If the frequency of the RFID tag is for example 13.56 MHz, this corresponds to 0.38 seconds. The fastest version with a coprocessor would take 0.2 seconds (2.7 million clock cycles).
This might be too slow for applications, which require fast response times, such as ticketing applications. However, the runtime is suitable for other applications such as brand

protection. These applications have relaxed requirements in terms of performance.
To sum up, it can be concluded that software-based ECC on RFID tags is applicable. By using a lightweight microprocessor and the presented hardware extensions, a good runtime performance and low memory requirements are possible. Furthermore, software-based approach maintains a high level of flexibility.

6.1 Future Work

In this thesis a flexible design approach of ECC on an RFID tag was presented. Different implementation versions were proposed and analyzed in terms of performance and area. Future work could evaluate the energy efficiency of the proposed variants with power simulation.
Furthermore, the quality of the approach could be reviewed by manufacturing the implementation and analyze the chip in terms of performance, energy consumption and side-channel attack resistance.

Bibliography

[1] European Union. Report on EU Customs Enforcement of Intellectual Property Rights. Available: `http://ec.europa.eu/taxation_customs/resources/documents/customs/customs_controls/counterfeit_piracy/statistics/2012_ipr_statistics_en.pdf`, July 2012. Accessed: 02/08/2012.

[2] Taxation and Customs Union. Counterfeit and Piracy. Available: `http://ec.europa.eu/taxation_customs/customs/customs_controls/counterfeit_piracy/combating/index_en.htm`, 2012. Accessed: 02/08/2012.

[3] M.O. Lehtonen, F. Michahelles, and E. Fleisch. Trust and Security in RFID-based Product Authentication Systems. *Systems Journal, IEEE*, 1(2):129–144, 2007.

[4] Z.J. Shi and H. Yan. Software Implementations of Elliptic Curve Cryptography. *International Journal of Network Security*, 7(2):157–166, 2008.

[5] J. Wolkerstorfer. Is Elliptic-Curve Cryptography Suitable to Secure RFID Tags?, 2005. Slides of a talk given at Workshop on RFID and Light-Weight Crypto, July 14–15, Graz.

[6] Daniel Hein. Elliptic Curve Cryptography ASIC for Radio Frequency Authentication. Master's thesis, Graz University of Technology, 2008.

[7] Lee et al. Elliptic-curve-based Security Processor for RFID. *IEEE Transactions on Computers*, 2008.

[8] H. Bock, M. Braun, M. Dichtl, E. Hess, J. Heyszl, W. Kargl, H. Koroschetz, B. Meyer, and H. Seuschek. A Milestone Towards RFID Products Offering Asymmetric Authentication Based on Elliptic Curve Cryptography. *Invited talk at RFIDsec*, 2008.

[9] F. Furbass and J. Wolkerstorfer. ECC Processor with Low Die Size for RFID Applications. In *IEEE International Symposium on Circuits and Systems*, pages 1835–1838. IEEE, 2007.

[10] C. Galuzzi and K. Bertels. The Instruction-set Extension Problem: A Survey. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2011.

[11] A. Mitrokotsa, M.R. Rieback, and A.S. Tanenbaum. Classifying RFID Attacks and Defenses. *Information Systems Frontiers*, 2010.

[12] K. Finkenzeller et al. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication*. Wiley, 2010.

[13] K. Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification*. Wiley, 2003.

[14] M. Aigner, M. Feldhofer, and M. Lamberger. RFID Security, 2010. Course Material IT Security.

[15] Gildas Avoine. Privacy Challenges in RFID. In J. Garcia-Alfaro, G. Navarro-Arribas, N. Cuppens-Boulahia, and S. de Capitani di Vimercati, editors, *Data Privacy Management and Autonomous Spontaneus Security*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012.

[16] Jerry Banks. Understanding RFID Part 9: RFID Privacy and Security. Available: `http://www.rfidnews.org/2008/05/30/understanding-rfid-part-9-rfid-privacy-and-security`, May 2008. Accessed: 06/08/2012.

[17] P. Hock. RFID Security and Privacy. *Institute of Media Informatics Ulm University*, page 25, 2012.

[18] NXP Semiconductors. Security of MIFARE Classic. Available: `http://www.mifare.net/technology/security/mifare-classic`, 2012. Accessed: 06/08/2012.

[19] F. Garcia, G. de Koning Gans, R. Muijrers, P. Van Rossum, R. Verdult, R. Schreur, and B. Jacobs. Dismantling Mifare Classic. *Computer Security-ESORICS 2008*, 2008.

[20] A. Kerckhoff. La Cryptographie Militaire, Journel des Sciences militaires, 1883.

[21] T. Eisenbarth and S. Kumar. A Survey of Lightweight-Cryptography Implementations. *Design & Test of Computers, IEEE*, 2007.

[22] E.O. Hwang. *Digital Logic and Microprocessor Design with VHDL*. 2006. Accessed: 03/11/2012.

[23] M. Abd-El-Barr and H. El-Rewini. *Fundamentals of Computer Organization and Architecture*. Wiley-Interscience, 2004.

[24] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[25] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann, 2011.

[26] J. Großschädl, S. Tillich, P. Ienne, L. Pozzi, and A.K. Verma. When Instruction Set Extensions Change Algorithm Design: A Study in Elliptic Curve Cryptography. 2005.

[27] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 203–213. ACM, 2000.

[28] J. Großschädl and E. Savaş. Instruction Set Extensions for Fast Arithmetic in Finite Fields GF(p) and GF(2^m). *Cryptographic Hardware and Embedded Systems – CHES*, pages 161–169, 2004.

[29] W. Trappe and L.C. Washington. *Introduction to Cryptography.* Pearson Prentice Hall, 2006.

[30] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC, 1997.

[31] Erich Wenger. Neptun - ECC Processor for RFID Tags and Smart Cards. Master's thesis, ETH Zürich, University of Technology Graz, 2010.

[32] U.S department of commerce/NIST. Data Encryption Standard, FIPS PUB 46-3, 1999.

[33] J. Daemen and V. Rijmen. Rijndael, AES Proposal. In *Proceedings from the First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, 1998.

[34] R.L. Rivst, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, 1977.

[35] J. Hoffstein, J. Pipher, and J. Silverman. NTRU: A Ring-based Public Key Cryptosystem. *Algorithmic Number Theory*, pages 267–288, 1998.

[36] R.J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *DSN Progress Report 42–44*, 1978.

[37] G. Joy Persial, M. Prabhu, and R. Shanmugalakshmi. Side channel Attack-Survey. *Int J Adva Sci Res Rev*, 2011.

[38] B. Badrignans, J.L. Danger, V. Fischer, G. Gogniat, and L. Torres. *Security Trends for FPGAs: From Secured to Secure Reconfigurable Systems.* Springer, 2011.

[39] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer-Verlag New York Inc, 2007.

[40] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[41] S.A. Vanstone D.R. Hankerson and A.J. Menezes. *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York Inc, 2004.

[42] Michael Rosing. *Implementing Elliptic Curve Cryptography.* Manning, 1998.

[43] H. Yan and Z.J. Shi. Studying Software Implementations of Elliptic Curve Cryptography. In *Third International Conference on Information Technology: New Generations*, pages 78–83. IEEE, 2006.

[44] M. Estes and P. Hines. Efficient Implementation of an Elliptic Curve Cryptosystem over Binary Galois Fields in Normal and Polynomial Bases. *George Massion University*, 2006.

[45] P.L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[46] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. In *Cryptographic Hardware and Embedded Systems*, pages 724–724. Springer, 1999.

[47] J. Großschädl and G.A. Kamendje. Instruction Set Extension for Fast Elliptic Curve Cryptography over Binary Finite Fields GF(2^m). In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 455–468. IEEE, 2003.

[48] S. Pohlig and M. Hellman. An Improved Algorithm for Computing Logarithms over GF(p) and its Cryptographic Significance. *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[49] J.M. Pollard. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[50] D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Cryptographic Hardware and Embedded Systems – CHES*, pages 243–267. Springer, 2000.

[51] E.O. Blaß and M. Zitterbart. Towards Acceptable Public-key Encryption in Sensor Networks. In *ACM 2nd International Workshop on Ubiquitous Computing*, pages 88–93. INSTICC Press Miami, USA, 2005.

[52] S. Kumar and C. Paar. Are Standards Compliant Elliptic Curve Cryptosystems Feasible on RFID? In *Workshop on RFID Security*, pages 12–14, 2006.

[53] J. Taverne, A. Faz-Hernández, D. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López. Software Implementation of Binary Elliptic Curves: Impact of the Carry-less Multiplier on Scalar Multiplication. *Cryptographic Hardware and Embedded Systems – CHES*, pages 108–123, 2011.

[54] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2012.

[55] A. Woodbury, D.V. Bailey, and C. Paar. Elliptic Curve Cryptography on Smart Cards without Coprocessors. *Smart Card Research and Advanced Applications*, 2000.

[56] N. Gura, A. Patel, A. Wander, H. Eberle, and S.C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. *Cryptographic Hardware and Embedded Systems – CHES*, pages 925–943, 2004.

[57] Atmel Corp. *8-bit Microcontroller with 128K Bytes In-System Programmable Flash: ATmega 128*, 2004.

[58] A. Kargl, S. Pyka, and H. Seuschek. Fast Arithmetic on ATmega128 for Elliptic Curve Cryptography. *Context of the SMEPP Project (Secure Middleware for Embedded Peer-to-Peer Systems, FP6 IST-5-033563)*, 2008.

[59] Erich Wenger, Thomas Baier, and Johannes Feichtner. JAAVR: Introducing the Next Generation of Security-enabled RFID Tags. *Euromicro Conference on DSD*, 2012.

[60] M. Hutter and E. Wenger. Fast Multi-precision Multiplication for Public-key Cryptography on Embedded Microprocessors. *Cryptographic Hardware and Embedded Systems – CHES*, pages 459–474, 2011.

[61] D.J. Malan, M. Welsh, and M.D. Smith. A Public-key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography. In *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pages 71–80. IEEE, 2004.

[62] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*, volume 265. Cambridge University Press, 1999.

[63] J. López and R. Dahab. High-Speed Software Multiplication in GF(2^m). *Progress in Cryptology – INDOCRYPT*, pages 93–102, 2000.

[64] S.C. Seo, HAN Dong-Guk, H.C. Kim, and H. Seokhie. TinyECCK: Efficient Elliptic Curve Cryptography Implementation over GF(2^m) on 8-bit Micaz Mote. *IEICE Transactions on Information and Systems*, 91(5):1338–1347, 2008.

[65] P. Szczechowiak, L. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. *Wireless sensor networks*, pages 305–320, 2008.

[66] M. Koschuch, J. Lechner, A. Weitzer, J. Großschädl, A. Szekely, S. Tillich, and J. Wolkerstorfer. Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller. *Cryptographic Hardware and Embedded Systems – CHES*, pages 430–444, 2006.

[67] S. Janssens, J. Thomas, W. Borremans, P. Gijsels, I. Verbauwhede, F. Vercauteren, B. Preneel, and J. Vandewalle. Hardware/Software Co-design of an Elliptic Curve Public-key Cryptosystem. In *IEEE Workshop on Signal Processing Systems*, pages 209–216. IEEE, 2001.

[68] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blumel. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n). *Lecture Notes in Computer Science*, pages 381–399, 2003.

[69] H. Aigner, H. Bock, M. Hütter, and J. Wolkerstorfer. A Low-cost ECC Coprocessor for Smartcards. *Cryptographic Hardware and Embedded Systems – CHES*, pages 65–78, 2004.

[70] E. Savaš, A. Tenca, and Ç. Koç. A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2^m). In *Cryptographic Hardware and Embedded Systems – CHES*, pages 277–292. Springer, 2000.

[71] E. Nahum. *Towards High Performance Cryptographic Software*. PhD thesis, Citeseer, 1995.

[72] C.K. Koc and T. Acar. Montgomery Multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, 1998.

[73] W. Drescher, K. Bachmann, and G. Fettweis. VLSI Architecture for Datapath Integration of Arithmetic over GF(2^m) on Digital Signal Processors. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 631–634. IEEE, 1997.

[74] J. Goodman and A.P. Chandrakasan. An Energy-efficient Reconfigurable Public-key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, 2001.

[75] MIPS Technologies, Inc. MIPS website. Available: `http://mips.com`. Accessed: 04/12/2012.

[76] A.M. Fiskiran and R.B. Lee. Evaluating Instruction Set Extensions for Fast Arithmetic on Binary Finite Fields. In *15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 125–136. IEEE, 2004.

[77] S. Kumar and C. Paar. Reconfigurable Instruction Set Extension for Enabling ECC on an 8-bit Processor. *Field Programmable Logic and Application*, pages 586–595, 2004.

[78] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *16th IEEE International Conference on Application-Specific Systems, Architecture Processors*, pages 343–349. IEEE, 2005.

[79] L. Batina, S. Berna Örs, B. Preneel, and J. Vandewalle. Hardware Architectures for Public Key Cryptography. *Integration, the VLSI journal*, 34(1):1–64, 2003.

[80] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede. Low-cost Elliptic Curve Cryptography for Wireless Sensor Networks. *Security and Privacy in Ad-Hoc and Sensor Networks*, pages 6–17, 2006.

[81] D. Hein, J. Wolkerstorfer, and N. Felber. ECC is Ready for RFID–A Proof in Silicon. In *Selected Areas in Cryptography*, pages 401–413. Springer, 2009.

[82] Infineon. *SRF 55V01P my-d light Manual*. Chip Card & Security ICs, 2005.

[83] Erich Wenger, Martin Feldhofer, and Norbert Felber. Low-Resource Hardware Design of an Elliptic Curve Processor for Contactless Devices, 2011.

[84] M. Hutter, J. Großschadl, and G.A. Kamendje. A Versatile and Scalable Digit-Serial/Parallel Multiplier Architecture for Finite Fields GF(2^m). In *International Conference on Information Technology: Coding and Computing*, pages 692–700. IEEE, 2003.

[85] J. Luo, K.D. Bowers, a. Oprea, and L. Xu. Efficient Software Implementation of Large Finite Fields GF(2^n) for Secure Storage Applications. *ACM Transactions on Storage (TOS)*, 2012.

[86] M. Arora, N.C. Lahane, and P. Srinivasan. Assembly Code Optimization Techniques for Real Time DSP Implementation of Speech Codecs. In *National Conference on Communication, Indian Institute of Technology Bombay, India*, 2002.

[87] Doulos Ltd. *SystemVerilog Golden Reference Guide*. 2006.

[88] K. T. Deepak. SystemVerilog Tutorial. Available: `http://www.asic-world.com/systemverilog/tutorial.html`. Accessed: 12/02/2013.

[89] M. Graphics. ModelSim Website. Available: `http://www.mentor.com/products/fv/modelsim`. Accessed: 03/01/2013.

[90] E. Weisstein. Standard Deviation Distribution. Available: `http://mathworld.wolfram.com/StandardDeviationDistribution.html`, 2012. Accessed: 04/02/2013.