Signals and Communication Technology

Francisco Rodríguez-Henríquez
N.A. Saqib
Arturo Díaz Pérez
Çetin Kaya Koç

Cryptographic Algorithms on Reconfigurable Hardware

Springer

Springer Series on
SIGNALS AND COMMUNICATION TECHNOLOGY

SIGNALS AND COMMUNICATION TECHNOLOGY

Multimedia Database Retrieval
A Human-Centered Approach
P. Muneesawang and L. Guan
ISBN 0-387-25627-X

Broadband Fixed Wireless Access
A System Perspective
M. Engels and F. Petre
ISBN 0-387-33956-6

Distributed Cooperative Laboratories
Networking, Instrumentation, and Measurements
F. Davoli, S. Palazzo and S. Zappatore (Eds.)
ISBN 0-387-29811-8

The Variational Bayes Method in Signal Processing
V. Šmídl and A. Quinn
ISBN 3-540-28819-8

Topics in Acoustic Echo and Noise Control
Selected Methods for the Cancellation of Acoustical Echoes, the Reduction of Background Noise, and Speech Processing
E. Hänsler and G. Schmidt (Eds.)
ISBN 3-540-33212-x

EM Modeling of Antennas and RF Components for Wireless Communication Systems
F. Gustrau, D. Manteuffel
ISBN 3-540-28614-4

Interactive Video
Methods and Applications
R. I Hammoud (Ed.)
ISBN 3-540-33214-6

ContinuousTime Signals
Y. Shmaliy
ISBN 1-4020-4817-3

Voice and Speech Quality Perception
Assessment and Evaluation
U. Jekosch
ISBN 3-540-24095-0

Advanced ManMachine Interaction
Fundamentals and Implementation
K.-F. Kraiss
ISBN 3-540-30618-8

Orthogonal Frequency Division Multiplexing for Wireless Communications
Y. (Geoffrey) Li and G.L. Stüber (Eds.)
ISBN 0-387-29095-8

Circuits and Systems
Based on Delta Modulation
Linear, Nonlinear and Mixed Mode Processing
D.G. Zrilic ISBN 3-540-23751-8

Functional Structures in Networks
AMLn—A Language for Model Driven Development of Telecom Systems
T. Muth ISBN 3-540-22545-5

RadioWave Propagation for Telecommunication Applications
H. Sizun ISBN 3-540-40758-8

Electronic Noise and Interfering Signals
Principles and Applications
G. Vasilescu ISBN 3-540-40741-3

DVB
The Family of International Standards for Digital Video Broadcasting, 2nd ed.
U. Reimers ISBN 3-540-43545-X

Digital Interactive TV and Metadata
Future Broadcast Multimedia
A. Lugmayr, S. Niiranen, and S. Kalli
ISBN 3-387-20843-7

Adaptive Antenna Arrays
Trends and Applications
S. Chandran (Ed.) ISBN 3-540-20199-8

Digital Signal Processing with Field Programmable Gate Arrays
U. Meyer-Baese ISBN 3-540-21119-5

Neuro-Fuzzy and Fuzzy Neural Applications in Telecommunications
P. Stavroulakis (Ed.) ISBN 3-540-40759-6

SDMA for Multipath Wireless Channels
Limiting Characteristics and Stochastic Models
I.P. Kovalyov ISBN 3-540-40225-X

Digital Television
A Practical Guide for Engineers
W. Fischer ISBN 3-540-01155-2

Speech Enhancement
J. Benesty (Ed.)
ISBN 3-540-24039-X

Multimedia Communication Technology
Representation, Transmission and Identification of Multimedia Signals
J.R. Ohm ISBN 3-540-01249-4

continued after index

Francisco Rodríguez-Henríquez
N.A. Saqib
A. Díaz-Pèrez
Çetin Kaya Koç

Cryptographic Algorithms on Reconfigurable Hardware

Springer

Francisco Rodríguez-Henríquez
Arturo Díaz Pérez

Departamento de Computación
Centro de Investigación y de Estudios Avanzados del IPN
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco. CP 07300
México, D.F.
MEXICO

Nazar Abbas Saqib
Centre for Cyber Technology and Spectrum Management
(CCT & SM)
National University of Sciences and Technology (NUST)
#295, Street 35, F-11/3, Islamabad-44000
Pakistan

Çetin Kaya Koç
Oregon State University
Corvallis, OR 97331, USA
&
Istanbul Commerce University
Eminönü, Istanbul 34112, Turkey

Cryptographic Algorithms on Reconfigurable Hardware

Library of Congress Control Number: 2006929210

ISBN 0-387-33883-7 e-ISBN 0-387-36682-2
ISBN 978-0-387-33883-5

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1

springer.com

Dedication

A mi esposa Nareli y mi hija Ana Iremi, por su amor y estoica paciencia;
A mis padres y hermanos, por compartir las mismas esperanzas.
Francisco Rodríguez-Henríquez

To Afshan (wife),Fizza (daughter), Ahmer (son) and Aashir (son), I love you all.
Nazar A. Saqib

To Mary, Maricarmen and Liliana, my wife and daughters, my love will keep alive for you all.
Arturo Díaz-Pérez

With my love to Laurie, Murat, and Cemre.
Çetin K. Koç

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AES	Advanced Encryption Standard
AF	Affine Transformation
ANSI	American National Standard Institute
API	Application Programming Interface
ARK	Add Round Key
ASIC	Application Specific Integrated Circuit
ATM	Automated Teller Machine
BEA	Binary Euclidean Algorithm
BRAMs	Block RAMs
BS	Byte Substitution
CBC	Cipher Block Chaining
CCM	Counter with CBC-MAC
CCSA	Carry Completion Sensing Adder
CDA	Carry Delayed Adder
CFB	Cipher Feedback mode
CLB	Configurable Logic Block
CPA	Carry Propagate Adder
CPLDs	Complex PLDs
CRT	Chinese Remainder Theorem
CSA	Carry Save Adder
CTR	Counter mode
DCM	Digital Clock Managers
DEA	Data Encryption Algorithm
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithmic Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
FIPS	Federal Information Processing Standards
FLT	Fermat's Little Theorem
FPGAs	Field Programmable Gate Arrays

GAL	Generic Array Logic
GSM	Global System for Mobile Communications
HDLs	Hardware Description Languages
IAF	Inverse Affine Transformation
IARK	Inverse Add Round Key
IBS	Inverse Byte Substitution
IEEE	Institute of Electrical and Electronics Engineers
IL	Iterative Looping
IMC	Inverse Mix Column
IOBs	Input/Output Blocks
IOEs	Input/Output Elements
IPSec	Internet Protocol Security
ISE	Xilinx Integrated Software Environment
ISO	International Organization for Standardization
ISR	Inverse ShiftRow
ITMIA	Itoh-Tsujii Multiplicative Inverse Algorithm
ITU	International Telecommunication Union
JTAG	Joint Test Action Group
KOM	Karatsuba-Ofman Multiplier
LABs	Logic Array Blocks
LC	Logic Cell
LEs	Logic Elements
MAC	Message Authentication Code
MRC	Mixed-Radix Conversion
NAF	Non-Adjacent Form
NFS	Number Field Sieve
NIST	National Institute of Standards and Technology
NZWS	Nonzero Window State
OFB	Output Feedback mode
PAL	Programmable Array Logic
PC-1	Permuted Choice One
PC-2	Permutated Choice Two
PDAs	Portable Digital Assistants
PKCS	Public Key Cryptography Standard
PLA	Programmable Logic Array
PLDs	Programmable Logic Devices
SRC	Single-Radix Conversion
SSL	Secure Socket Layer
TDEA	Triple DEA
TNAF	τ-adic NAF
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
WEP	Wired Equivalent Privacy
ZWS	Zero Window State

Preface

Cryptography provides techniques, mechanisms, and tools for private and authenticated communication, and for performing secure and authenticated transactions over the Internet as well as other open networks. It is highly probable that each bit of information flowing through our networks will have to be either encrypted and decrypted or signed and authenticated in a few years from now. This infrastructure is needed to carry over the legal and contractual certainty from our paper-based offices to our virtual offices existing in the cyberspace. In such an environment, server and client computers as well as handheld, portable, and wireless devices will have to be capable of encrypting or decrypting and signing or verifying messages. That is to say, without exception, all networked computers and devices must have cryptographic layers implemented, and must be able to access to cryptographic functions in order to provide security features. In this context, efficient (in terms of time, area, and power consumption) hardware structures will have to be designed, implemented, and deployed. Furthermore, general-purpose (platform-independent) as well as special-purpose software implementing cryptographic functions on embedded devices are needed. An additional challenge is that these implementations should be done in such a way to resist cryptanalytic attacks launched against them by adversaries having access to primary (communication) and secondary (power, electromagnetic, acoustic) channels.

This book, among only a few on the subject, is a fruit of an international collaboration to design and implement cryptographic functions. The authors, who now seem to be scattered over the globe, were once together as students and professors in North America. In Oregon and Mexico City, we worked on subjects of mutual interest, designing efficient realizations of cryptographic functions in hardware and software.

Cryptographic realizations in software platforms can be used for those security applications where the data traffic is not too large and thus low encryption rate is acceptable. On the other hand, hardware methods offer high speed and bandwidth, providing real-time encryption if needed. VLSI (also known as ASIC) and FPGAs are two distinct alternatives for implementing

cryptographic algorithms in hardware. FPGAs offer several benefits for cryptographic algorithm implementations over VLSI, as they offer flexibility and fast time-to-market. Because they are reconfigurable, internal architectures, system parameters, lookup tables, and keys can be changed in FPGAs without much effort. Moreover, these features come with low cost and without sacrificing efficiency.

This book covers computational methods, computer arithmetic algorithms, and design improvement techniques needed to obtain efficient implementations of cryptographic algorithms in FPGA reconfigurable hardware platforms. The concepts and techniques introduced in this book pay special attention to the practical aspects of reconfigurable hardware design, explain the fundamental mathematics behind the algorithms, and give comprehensive descriptions of the state-of-the-art implementation techniques. The main goal pursued in this book is to show how one can obtain high-speed cryptographic implementations on reconfigurable hardware devices without requiring prohibitive amount of hardware resources.

Every book attempts to take a still picture of a moving subject and will soon need to be updated, nevertheless, it is our hope that engineers, scientists, and students will appreciate our efforts to give a glimpse of this deep and exciting world of cryptographic engineering. Thanks for reading our book.

May 2006

F. Rodríguez-Henríquez, Nazar A. Saqib, A. Díaz-Pérez, and Çetin K. Koç

1

Introduction

This chapter presents a complete outline for this Book. It explains the main goals pursued, the strategies chosen to achieve those goals, and a summary of the material to be covered throughout this Book.

1.1 Main goals

The choice of reconfigurable logic as a target platform for cryptographic algorithm implementations appears to be a practical solution for embedded systems and high-speed applications. It was therefore planned to conduct a study of high-speed cryptographic solutions on reconfigurable hardware platforms.

Both efficient and cost effective solutions of cryptographic algorithms are desired on reconfigurable logic platform. The term "efficient" normally refers to "high speed" solutions. In this Book, we do not only look for high speed but also for low area (in terms of hardware resources) solutions.

Our main objective is therefore to find high speed and low area implementations of cryptographic algorithms using reconfigurable logic devices. That implies careful considerations of cryptographic algorithm formulations, which often will lead to modify the traditional specifications of those algorithms. That also implies knowledge of the target device: device structure, device resources, and device suitability to the given task. The design techniques and the understanding of the design tools are also included in the implications imposed by efficient solutions. An optimized cryptographic solution will be the one for which every step; starting from its high-level specification down to the physical prototype realization is carefully examined.

It is known that the final performance of cryptographic algorithms heavily depends on the efficiency of their underlying field arithmetic. Consequently, we begin our investigation by first studying the algorithms, solutions and corresponding architectures for obtaining state-of-the-art finite field arithmetic

realizations. Our study was carried out for both, prime and binary extension finite fields. We investigated field arithmetic algorithms for the operations of field addition, multiplication, squaring, square root, multiplicative inverse and exponentiation among others.

Thereafter, we selected a set of three of the most important cryptographic building blocks, for their implementation on reconfigurable logic devices: hash functions, symmetric block ciphers and public key cryptosystems in the form of elliptic curve cryptography.

We described first the basic principles for attaining efficient hardware implementation of hash functions. In the subject of symmetric ciphers, we study the two most emblematic algorithms, namely, the Data Encryption Standard (DES) and the Advance Encryption Standard (AES). In the case of asymmetric cryptosystems we analyze fast implementations of Elliptic Curve operations defined over binary extension fields.

Several considerations were made to achieve high speed and economical implementations of those algorithms on reconfigurable logic platforms. One of them was to exploit high bit-level parallelism where and whenever it was possible. Similarly, we employed design techniques especially tailored for exploiting the structure of the target devices.

A variety of hash function algorithms were studied first. Emphasis was made on MD5, by providing a step-by-step analysis of its algorithm flow. An explanation of the SHA-2 family was also included. In our descriptions we pondered hardware implementation aspects of the hash algorithms.

DES was the second cryptographic building block studied in this Monograph. The basic primitives involved in block ciphers specifically for DES were analyzed for their implementations on reconfigurable logic platform. A compact one round FPGA implementation of DES was carried out exploiting high bit-level parallelism. Experiments were made for optimizing the proposed FPGA architecture with respect to hardware area.

A more detailed study was planned regarding AES due to its importance for the current security needs in the IT sector. Each step of the algorithm was investigated looking for improvements in the standard transformations of the algorithm and for an optimal mapping to the target device. Both, iterative and pipeline approaches for encryption were used for AES FPGA implementation. We attempted to reduce the critical paths for encryption/decryption by sharing common resources or optimizing the standard transformations of the algorithm.

In the case of Elliptic Curve Cryptography (ECC), we utilized a hierarchical six-layer model, but only the lower three layers were addressed in this Book. The first layer of the model deals with the efficient implementation of finite field arithmetic. The Second layer makes use of the underlying arithmetic for implement elliptic curve arithmetic main primitives: point addition and point doubling. The third layer implements elliptic curve scalar multiplication which is achieved by adding n copies of the same point P on the curve. Both the point addition and doubling operations from the second layer serve

as building blocks for the third layer. We strived for using parallel techniques for all the three layers. This way, a generic architecture for the elliptic curve scalar multiplication was proposed and implemented on the FPGA platform. We also presented parallel formulations of the scalar multiplication operation on Koblitz curves an architecture that is able to compute the elliptic curve scalar multiplication using the half-and-add method. Additionally, we presented optimizations strategies for computing a point addition and a point doubling using LD projective coordinates in just eight and three clock cycles, respectively.

1.2 Monograph Organization

Next chapters present a short introduction to the cryptographic algorithms chosen to illustrate the design strategies discussed previously as well as the mathematical background required for the correct understanding of the material to be presented. Design comparisons and conclusion remarks are presented at the end of each Chapter. A short summary of each chapter is given below.

In Chapter 2, a brief review of modern cryptographic algorithms is given. Topics addressed include: Secret-key and public-key cryptography, hash functions, digital signatures, an so forth. Furthermore, we also discuss in this Chapter potential real-world cryptographic applications and the suitability of reconfigurable hardware devices for accommodate them.

In Chapter 3 a brief introduction to reconfigurable hardware technology is given. We explain the historical development of FPGA devices and include a detailed description of the FPGA families of two major manufacturers: Xilinx and Altera. We also cover reconfigurable hardware design issues, metrics and security.

In Chapter 4, some important mathematical concepts are presented. Those concepts are particularly helpful for the understanding of cryptographic operations for AES and elliptic curve cryptosystems. Key mathematical concepts for a class of elliptic curves are also described at the end of this Chapter.

In Chapter 5, we discuss state-of-the-art arithmetic algorithms for prime fields. We present efficient hardware design alternatives for operations such as adders, modular adders, modular multipliers and exponentiation among others. We give at the end of each Section a comparison analysis with some of the most significant works reported in this topic.

In Chapter 6, state-of-the-art algorithms for binary extension fields are studied. We discuss relevant algorithms for performing efficiently field multiplication, squaring, square root, inversion and reduction among others. We give at the end of each Section a comparison analysis with some of the most significant works reported in this topic.

In Chapter 7, we study efficient reconfigurable hardware implementations of hash functions. Specifically, we carefully analyze MD5, arguably the most studied hash function ever. We give at the end of each Section a comparison analysis with some of the most significant works reported in this topic.

In Chapter 8, a general guideline for implementing symmetric block ciphers is described. Basic primitives involved in block ciphers are listed and design tips are provided for their efficient implementations on reconfigurable platform. DES is presented as a case of study. A compact and fast DES implementation on reconfigurable platform is explained. We give at the end of this Chapter a comparison analysis with some of the most significant works reported in this topic.

In Chapter 9, we explore multiple architectures for AES. Several efficient techniques for AES implementation are described. Several efficient AES encryptor and encryptor/decryptor cores based on those techniques are presented on reconfigurable platforms. The benefits/drawbacks of all AES cores are examined. We give at the end of this Chapter a comparison analysis with some of the most significant works reported in this topic.

In Chapter 10 we discuss several algorithms and their corresponding hardware architecture for performing the scalar multiplication operation on elliptic curves defined over binary extension fields $GF(2^m)$. By applying parallel strategies at every stage of the design, we are able to obtain high speed implementations at the price of increasing the hardware resource requirements. Specifically, we study the following four different schemes for performing elliptic curve scalar multiplications,

- Scalar multiplication applied on Hessian elliptic curves.
- Montgomery Scalar Multiplication applied on Weierstrass elliptic curves.
- Scalar multiplication applied on Koblitz elliptic curves.
- Scalar multiplication using the Half-and-Add Algorithm.

1.3 Acknowledgments

We would like to thank to all the long list of people who contribute to the material presented in this Book, needless to say that all of them are worthy to be mentioned. We gratefully thank our former Master's students: Juan Manuel Cruz-Alcaraz, Sabel Mercurio Hernández-Rodríguez and Emmanuel López-Trejo who contribute with their hard work and talent to the design and testing of several architectures presented in Chapters 6, 9 and 10. We would also like to thank our colleagues Guillermo Morales-Luna, Julio López-Hernández, Nareli Cruz-Cortés, Tariq Saleem, Shamim Baig, Habeel Ahmed, Erkay Savas, Tugrul Yanik, Luis Gerardo De-La-Fraga and Carlos Coello Coello who provided priceless comments and advice which greatly helped us to improve the

contents of this Book. We also acknowledge valuable contributions from Karla Gómez-Avila, Marco Negrete-Cervantes, Víctor Serrano-Hernández, Alejandro Arenas-Mendoza, Guillermo Martínez-Silva and Carlos López-Peza. We gratefully acknowledge our Springer editor, Jason Ward, for his diligent efforts and support towards the publication of this Work.

Last but not least, the first and third authors acknowledge support from CONACyT through the NSF-CONACyT project number 45306. The second author acknowledge support from the faculty and staff members of the Centre for Cyber Technology and Spectrum Management (CCT & SM), National University of Sciences and Technology (NUST), Islamabad-Pakistan.

2

A Brief Introduction to Modern Cryptography

In our Information Age, the need for protecting information is more pronounced than ever. Secure communication for the sensitive information is not only compelling for military or government institutions but also for the business sector and private individuals. The exchange of sensitive information over wired and/or wireless Internet, such as bank transactions, credit card numbers and telecommunication services are already common practices. As the world becomes more connected, the dependency on electronic services has become more pronounced. In order to protect valuable data in computer and communication systems from unauthorized disclosure and modification, reliable non-interceptable means for data storage and transmission must be adopted.

Figure 2.1 shows a hierarchical six-layer model for information security applications. Let us analyze that figure from a top-down point of view. On layer 6, several popular security applications have been listed such as: secure e-mail, digital cash, e-commerce, etc. Those applications depend on the implementation in layer 5 of secure authentication protocols like SSL/TLS, IPSec, IEEE 802.11, etc. However, those protocols cannot be put in place without implementing layer 4, which consists on customary security services such as: authentication, integrity, non-repudiation and confidentiality. The underlying infrastructure for such security services is supported by the two pair of cryptographic primitives depicted in layer 3, namely, encryption/decryption and digital signature/verification. Both pair of cryptographic primitives can be implemented by the combination of public-key and private key cryptographic algorithms, such as the ones listed in layer 2. Finally, in order to obtain a high performance from the cryptographic algorithms of layer 1, it is indispensable to have an efficient implementation of arithmetic operations such as, addition, subtraction, multiplication, exponentiation, etc.

In the rest of this Chapter we give a short introduction to the algorithms and security services listed in layers 2-4. Hence, the basic concepts of cryptography, fundamental operations in cryptographic algorithms and some im-

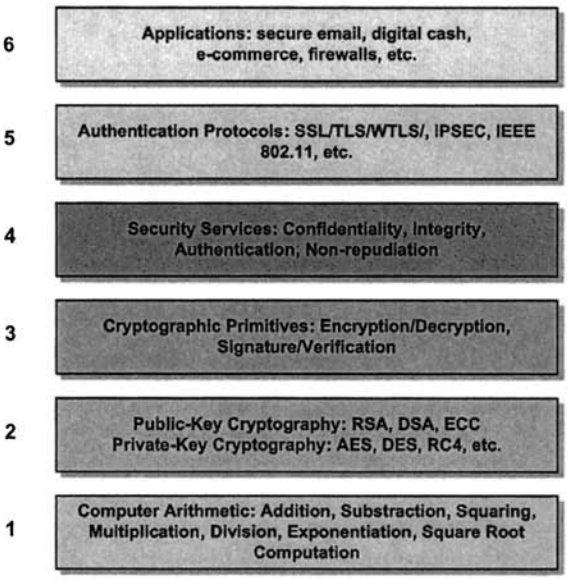

Fig. 2.1. A Hierarchical Six-Layer Model for Information Security Applications

portant cryptographic applications in the industry are studied and analyzed. Furthermore, alternatives for the implementation of cryptographic algorithms on various software and hardware platforms are also discussed.

2.1 Introduction

A cryptographic cipher system can hide the actual contents of every message by transforming (enciphering) it before transmission or storage. The techniques needed to protect data belong to the field of cryptography, which can be defined as follows.

Definition 2.1. *We define* ***Cryptography*** *as the discipline that studies the mathematical techniques related to Information security such as providing the security services of confidentiality, data integrity, authentication and non-repudiation.*

In the wide sense, cryptography addresses any situation in which one wishes to limit the effects of dishonest users [110]. Security services, which include confidentiality, data integrity, entity authentication, and data origin authentication [228], are defined below.

- **Confidentiality:** It guarantees that the sensitive information can only be accessed by those users/entities authorized to unveil it. When two or more parties are involved in a communication, the purpose of confidentiality is to guarantee that only those two parties can understand the data exchanged. Confidentiality is enforced by encryption.
- **Data integrity:** It is a service which addresses the unauthorized alteration of data. This property refers to data that has not been changed, destroyed, or lost in a malicious or accidental manner.
- **Authentication:** It is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. For these reasons this aspect of cryptography is usually subdivided into two major classes: *entity authentication* and *data origin authentication*. Data origin authentication implicitly provides data integrity.
- **Non-repudiation:** It is a service which prevents an entity from denying previous commitments or actions. For example, one entity may authorize the purchase of property by another entity and later deny such authorization was granted. A procedure involving a trusted third party is needed to resolve the dispute.

In cryptographic terminology, the message is called *plaintext*. Encoding the contents of the message in such a way that its contents cannot be unveiled by outsiders is called *encryption*. The encrypted message is called the *ciphertext*. The process of retrieving the plaintext from the ciphertext is called *decryption*. Encryption and decryption usually make use of a *key*, and the coding method use this key for both encryption and decryption. Once the plaintext is coded using that key then the decryption can be performed only by knowing the proper key.

Cryptography falls into two important categories: secret and public key cryptography. Both categories play their vital role in modern cryptographic applications. For several crucial applications, a combination of both secret and public key methods is indispensable.

2.2 Secret Key Cryptography

Definition 2.2. *Matematically, a* **symmetric key cryptosystem** *can be defined as the tuple* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$*, where [110]:*

$\mathcal{P}$ *represents the set of finitely many possible plain-texts.*
$\mathcal{C}$ *represents the set of finitely many possible cipher-texts.*
$\mathcal{K}$ *represents the key space, i.e, the set of finitely many possible keys.*
$\forall K \in \mathcal{K}\ \exists E_K \in \mathcal{E}$ *(encryption rule),* $\exists D_K \in \mathcal{D}$ *(decryption rule).*
Each $E_K : \mathcal{P} \rightarrow \mathcal{C}$ *and* $D_K : \mathcal{C} \rightarrow \mathcal{P}$ *are well-defined functions such that* $\forall x \in \mathcal{P}, D_K(E_K(x)) = x$.

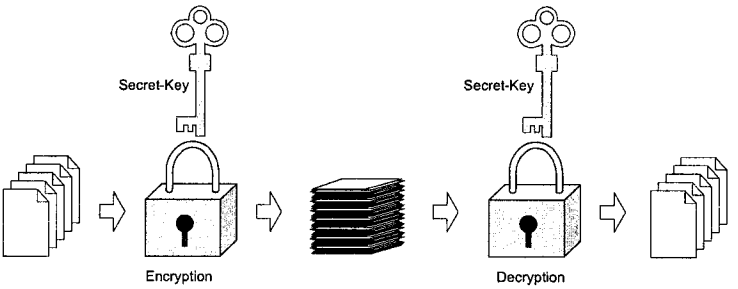

Fig. 2.2. Secret Key Cryptography

Both encryption and decryption keys (which sometimes are the same keys) are kept secret and must be known at both ends to perform encryption or decryption as is shown in Fig. 2.2. Symmetric algorithms are fast and are used for encrypting/decrypting high volume data. It is customary to classify symmetric algorithms into two types: stream ciphers and block ciphers.

- **Stream ciphers:** A stream cipher is a type of symmetric encryption algorithms in which the input data is encrypted one bit (sometimes one byte) at a time. They are sometimes called state ciphers since the encryption of a bit is dependent on the current state. Some examples of stream ciphers are SEAL, TWOPRIME, WAKE, RC4, A5, etc.
- **Block ciphers:** A block cipher takes as an input a fixed-length block (plaintext) and transform it into another block of the same length (ciphertext) under the action of a user-provided secret key. Decryption is performed by applying the reverse transformation to the ciphertext block using the same secret key. Modern block ciphers typically use a block length of 128 bits. Some famous block ciphers are DES, AES, Serpent, RC6, MARS, IDEA, Twofish, etc.

The most popular block cipher algorithm used in practice is DEA (*Data Encryption Algorithm*) defined in the standard DES [251]. The secret key used in DEA has a bit-length of 56 bits. Even though that key length was considered safe back in the middle 70's, nowadays technology can break DEA in some few hours by launching a brute-force attack. That is why DEA is widely used as Triple DEA (TDEA) which may offer a security equivalent to 112 bits. TDEA uses three 56-bit keys (namely, K_1, K_2 and K_3). If each of these keys is independently generated, then this is called the three key TDEA (3TDEA). However, if K_1 and K_2 are independently generated, and K_3 is set equal to K_1, then this is called the two key TDEA (2TDEA) [258].

On October 2000, a new symmetric cryptographic algorithm "Rijndael" was chosen as the new Advanced Encryption Standard (AES) [60] by NIST (National Institute of Standards and Technology) [253]. Due to its enhanced

security level, it is replacing DEA and triple DEA (TDEA) in a wide range of applications.

Although all aforementioned secret key ciphers offer a high security and computational efficiency, they also exhibit several drawbacks:

- **Key distribution and key exchange** The master key used in this kind of cryptosystems must be known by the sender and receiver only. Hence, both parties should prevent that this key can get compromised by unauthorized entities[1].
- **Key management** Those system having many users, must generate/manage many keys. For security reasons, a given key should be changed frequently, even in every session.
- **Incompleteness** It is impossible to implement some of the security services mentioned before. In particular, *Authentication* and *non-repudiation* cannot be fully implemented by only using secret key cryptography [317].

2.3 Hash Functions

Definition 2.3. *A Hash function H is a computationally efficient function that maps fixed binary chains of arbitrary length $\{0,1\}^*$ to bit sequences $H(B)$ of fixed length. $H(B)$ is the hash value or digest of B.*

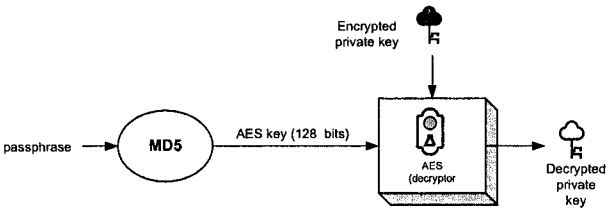

Fig. 2.3. Recovering Initiator's Private Key

In words, a hash function h maps bit-strings of arbitrary finite length to strings of fixed length, say n bits. MD5 and SHA-1 are two examples of hash functions. MD5 produces 128-bit hash values while SHA-1 produces 160-bit hash values.

Hash functions can be used for protecting user's secret key as depicted in Fig. 2.3. Fig. 2.3 shows the customary procedure used for accomplishing that

[1] This implies that in a community of n users a total of $\frac{n(n-1)}{2}$ secret keys must be created so that all users can communicate with each other in a confidential manner.

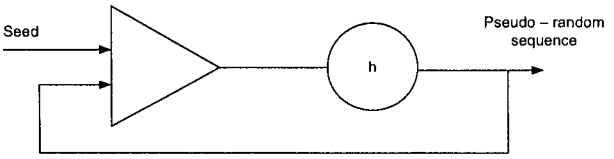

Fig. 2.4. Generating a Pseudorandom Sequence

goal. It is noticed that the AES secret key is generated by means of the hash value corresponding to the pass-phrase given by the user. Another typical application of Hash functions is in the domain of pseudorandom sequences as shown in Fig. 2.4.

Nevertheless, the main application of hash function is as a key building block for generating digital signatures as it is explained in the next Section.

2.4 Public Key Cryptography

A breakthrough in Cryptography occurred in 1976 with the invention of *public key cryptography* by Diffie and Hellman[2] [68]. This invention not only solved the key distribution and management problem but also it provided the necessary tool for implementing authentication and non-repudiation security services effectively.

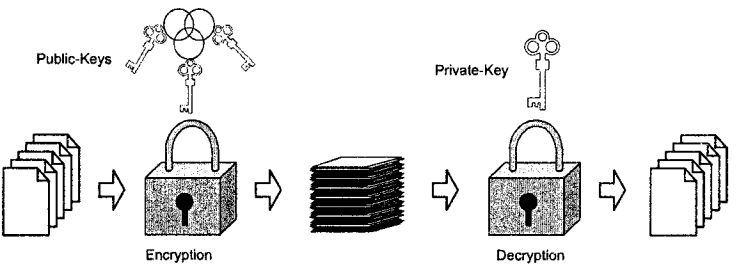

Fig. 2.5. Public Key Cryptography

[2] Although Diffie and Hellman were the first in publishing the concepts of public key cryptography in the open literature, we know now that they were not the first inventors. In 1997, a British Security agency (CESG, *National Technical Authority for Information Assurance*) published documents showing that in fact James Ellis and Clifford Cocks came out with the mechanisms needed for performing RSA-like public key cryptography in 1973. Short after that, M. Williamson discovered what is now known as Diffie-Hellman key exchange [374, 317, 206].

Asymmetric algorithms use a different key for encryption and decryption, and the decryption key cannot be easily derived from the encryption key. Asymmetric algorithms use two keys known as public and private keys as shown in Fig. 2.5.

The public key is available to everyone at the sending end. However a private or secret key is known only to the recipient of the message. An important characteristic of any public key system is that the public and private keys are related in such a way that only the public key can be used to encrypt (decrypt) messages and only the corresponding private key can be used to decrypt(encrypt) them.

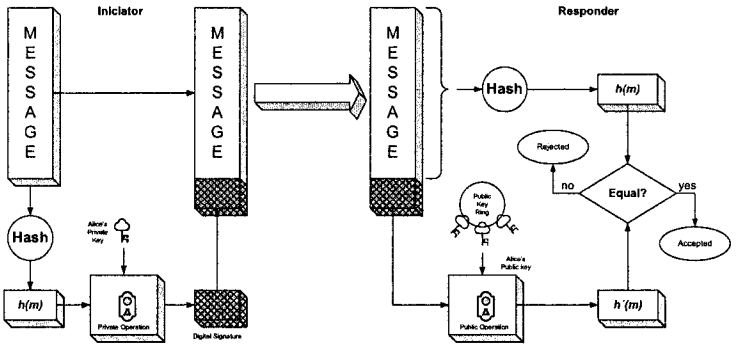

Fig. 2.6. Basic Digital Signature/Verification Scheme

Public key cryptosystems can be used for generating *digital signatures*, which cannot be repudiated. The concept of digital signature is analog to the real-world autograph signature, but it is more powerful as it also protects against malicious data modifications. A digital signature scheme is based in two algorithms: signature and verification as explained below.

- A encrypts the message m using its private key $c_1 := E_{K_{priv}(A)}(m)$
- A encrypts the result c_1 using B's public key and send the result to B,

$$c = E_{K_{pub}(B)}(c_1) = E_{K_{pub}(B)}\{E_{K_{priv}(A)}(m)\}$$

- B recovers m by performing,

$$m = D_{K_{pub}(A)}\{D_{K_{priv}(B)}(c)\}$$

Since B is able to recover m using A's public key, B can verify whether A really sign the message using its private key. Moreover, since the signature depends on the message contents, theoretically nobody else can reuse the same signature in any other message.

In practice, as is shown in Fig.2.6, a digital signature is applied not to the document to be signed itself, but to its *hash* value. This is due to efficiency reasons as public key cryptosystems tend to be computationally intensive. A hash function H is applied to the message to append its hash value $h = H(M)$, to the document itself. Thereafter, h is signed by "encrypting" it with the private key of the sender. This becomes the signature part of the message.

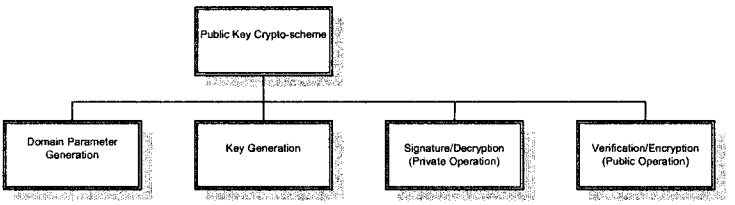

Fig. 2.7. Public key cryptography Main Primitives

As shown in Fig. 2.7 Public key cryptosystems' main primitives are:

1. **Domain Parameter Generation.** This primitive creates the mathematical infrastructure required by the particular cryptosystem to be used.
2. **Key Generation.** This primitive create users' public/private key.
3. **Public Operation.** This primitive is used for encrypting and/or verifying messages.
4. **Private Operation.** This primitive is used for decrypting and/or signing messages.

Theoretically, a public key cryptosystem can be constructed by means of specialized mathematical functions called "trapdoor one-way functions" which can be formally defined as follows.

Definition 2.4. *A One-way Function [110] is an injective function* $f(x)$

$$f : \{0,1\}^* \rightarrow \{0,1\}^*,$$

such that $f(x)$ *can be computed efficiently, but the computation of* $f^{-1}(y)$ *is computational intractable, even when using the most advanced algorithms along with the most sophisticated computer systems. We say that a one-way function is a One-way trapdoor function if is feasible to compute* $f^{-1}(y)$ *if and only if a supplementary information (usually the secret key) is provided.*

In words, a *one-way* function f is easy to compute for any domain value x, but the computation of $f^{-1}(x)$ should be computationally intractable. A trapdoor one-way function is a one-way function such that the computation $f^{-1}(x)$ is easy, provided that certain special additional information is known. The following three problems are considered among the most common for creating *trapdoor one-way* functions.

- ***Integer Factorization problem:*** Given an integer number n, obtain its prime factorization, i.e., find $n = {p_1}^{e_1}{p_2}^{e_2}{p_3}^{e_3}\cdots{p_k}^{e_k}$, where p_i is a prime number and $e_i \geq 1$.
  It is noticed that finding large prime numbers[3] is a relatively easy task, but solving the problem of factorizing the product of prime numbers is considered computationally intractable if the prime numbers are chosen carefully and with a sufficient large bit-length [196].
- ***Discrete Logarithm problem:*** Given a number p, a generator $g \in {\mathbb{Z}_p}^*$ and an arbitrary element $a \in {\mathbb{Z}_p}^*$, find the unique number i, $0 \leq i < p-1$, such that $a \equiv g^i (mod\, p)$.
  This problem is useful in cryptography due to the fact that finding discrete logarithms is difficult. The brute-force method for finding $g^j (mod\, p)$ for $1 < j \leq p-1$ is computationally unfeasible for sufficiently large prime values. However, the field exponentiation operation can be computed efficiently. Hence, $g^i (mod\, p)$ can be seen as a *trapdoor one-way function* function for certain values of p.
- ***Elliptic curve discrete Logarithm problem:*** Let $E_{\mathbb{F}_q}$ be an elliptic curve defined over the finite field $F_q$and let P be a point $P \in E_{\mathbb{F}_q}$ with primer order n. Consider the k-multiple of the point P, $Q = kP$ defined as the elliptic curve point resulting of adding P, $k-1$ times with itself, where k is a positive scalar in $[\![1, n-1]\!]$. The elliptic curve discrete logarithm problem consists on finding the scalar k that satisfies the equation $Q = kP$. This problem is considered a strong one-way trapdoor function due to the fact that computing k given Q and P is a difficult computational problem. However, given k is relatively easy to obtain the k-th multiple of P, namely, $Q = kP$.

2.5 Digital Signature Schemes

- $\mathcal{M}$ represents the set of all finitely many messages that can be signed
- $\mathcal{S}$ represents the set of all finitely many signatures (usually the signatures are fixed-length binary chains).
- $\mathcal{K}_{\mathcal{S}}$ represents the set of private keys.
- $\mathcal{K}_{\mathcal{V}}$ represents the set of public keys.
- $S_{\mathcal{E}} \colon \mathcal{M} \longrightarrow \mathcal{S}$ represents the transformation rule for an entity $\mathcal{E}$.
- $V_{\mathcal{E}} \colon \mathcal{M} \times \mathcal{S} \longrightarrow \{true, false\}$ represents the verification transformation for signatures produced by $\mathcal{E}$. It is used for other entities in order to verify signatures produced by $\mathcal{E}$.

  $S_{\mathcal{E}}$ y $V_{\mathcal{E}}$ define a digital signature scheme for $\mathcal{E}$.

Definition 2.5. *A Digital signature scheme is the triple* (Gen, Sig, Ver) *of algorithms such that,*

[3] In the cryptography domain a large prime number has a bit-length of at least 512 bits.

i. Gen is a Key generation algorithm, with input s; known as the security parameter; and possibly another extra information I, which gives as an ouptut $(\mathbf{k}_{\mathcal{S}}, \mathbf{k}_{\mathcal{V}}) \in \mathcal{K}_{\mathcal{S}} \times \mathcal{K}_{\mathcal{V}}$ *corresponding to private key, and public key, respectively.*

ii. Sig is a Signature algorithm, with input $(\mathbf{m}, \mathbf{k}_{\mathcal{S}}) \in \mathcal{M} \times \mathcal{K}_{\mathcal{S}}$*, which gives as an output an element* $\sigma \in \mathcal{S}$*, called Signature (of the message* $\mathbf{m}$ *with the private key* $\mathbf{k}_{\mathcal{S}}$*).*

iii. Ver is a Verification algorithm, with input $(\mathbf{m}, \sigma, \mathbf{k}_{\mathcal{V}}) \in \mathcal{M} \times \mathcal{S} \times \mathcal{K}_{\mathcal{V}}$*, which gives as an output the set* $\{true, false\}$ *and*

$$Ver(\mathbf{m}, Sig(\mathbf{m}, \mathbf{k}_{\mathcal{S}}), \mathbf{k}_{\mathcal{V}}) = true$$

$\forall$ *valid* $(\mathbf{k}_{\mathcal{S}}, \mathbf{k}_{\mathcal{V}})$ *obtained from Gen and for all* $\mathbf{m} \in \mathcal{M}$.

Undoubtedly, the most popular public-key algorithms are RSA (based on factoring large numbers), DSA and ElGamal (based on discrete log problem) and Elliptic Curve Cryptosystems. Elliptic curve cryptography is now popular due to the fact that it offers the same security level as offered by other contemporary algorithms at a shorter key length. It is based on elliptic curve addition operation.

2.5.1 RSA Digital Signature

The most popular algorithm for commercial applications is RSA[4]. RSA algorithm is symmetric in the sense that both, the public key and the private key can be utilized for encrypting a message.

RSA Key Generation

Algorithm 2.1 shows RSA key generation procedure. The public key is composed by the two integers (n, e), where n is called the RSA modulus and is defined as the product of two prime numbers p, q, of approximately the same bit-length. Both, p, q should be generated randomly and must be kept secret. The number e is called the *public exponent.* It must satisfy: $1 < e < \phi$ and $\gcd(e, \phi) = 1$ where $\phi = (p-1)(q-1)$. The private key d is called the *private exponent* and it must satisfy: $1 < d < \phi$ and $ed \equiv 1 (\text{mod } \phi)$. It is noticed that the problem of determining the key d given the public key (n, e) has a computacional difficulty equivalent to the integer factorization problem of finding p or q given n.

[4] RSA stands for the first letter in each of its inventors' last names: Rivest, Shamir and Addleman. These three distinguished professors were declared the 2002 A.M Turin award winners. At that time, Professor Shamir consider it "the ultimate seal of approval" for Cryptography as a Computer Science discipline [325].

Algorithm 2.1 RSA Key Generation

Require: bit-length k, a public exponent e, where e is a small prime number.
Ensure: RSA public key (n, e) and private key d.
1: Randomly find two primes $\frac{k}{2}$-bit numbers p and q.
2: $n = pq$;
3: $\phi(n) = (p-1)(q-1)$;
4: **if** $\gcd(e, \phi(n)) \neq 1$ **then**
5: Go to Step 1.
6: **end if**
7: Find d such that $d = e^{-1} \bmod \phi(n)$.
8: **Return** (n, e, d).

RSA Digital Signature

RSA encryption/decryption and Signature/verification are based in the Euler theorem identity, which establishes that,

$$m^{ed} = m \pmod{n} \tag{2.1}$$

for any arbitrary integer m. Signature and verification processes are shown in Algorithms 2.2 and 2.3. The author A of the message m computes the hash value $h = H(m)$. Then, A computes the signature $s = h^d$. Then A can send the message m along with the signature s to a verifying entity, say B. B can verify A's signature as follows. It recovers the hash value from s by computing $\hat{h} = s^e$. Thereafter, B computes once again the hash value, say, $h = H(m)$. If $\hat{h} = h$, then the signature is accepted otherwise, it is rejected.

Algorithm 2.2 RSA Digital Signature

Require: Sender's public key (n, e), Sender's private key d, message m.
Ensure: digital signature s.
1: $h = H(m)$;
2: $s = h^d \bmod n$.
3: **Return** s.

2.5.2 RSA Standards

RSA is specified in [193, 253, 255]. Additionally, there exist a number of standards where the digital signature algorithm RSA just described is utilized. The Public Key Cryptography Standard (PKCS), is a set of standards that include among others, PKCS#1[5], PKCS#3[6] and PKCS#12[7]. PKCS series

[5] RSA Cryptography Standard
[6] Diffie-Hellman key agreement Standard
[7] Personal Information Exchange Syntax Standard

Algorithm 2.3 RSA Signature Verification

Require: Sender's public key (n, e), message m, digital signature s.
Ensure: Accept/Reject.
1: $h = H(m)$;
2: $\hat{h} = s^e \bmod n$;
3: **if** $h = \hat{h}$ **then**
4: **Return**(Accept);
5: **else**
6: **Return**(Reject);
7: **end if**

have become part of many formal and de facto standards, including ANSI X9 documents, PKIX, SET, S/MIME, and SSL [193].

2.5.3 DSA Digital Signature

The Digital Signature Algorithm (DSA) is based in the crypto-scheme proponed by ElGamal in 1984, which in turn is based on the discrete logarithm problem. Many versions of the original ElGamal procedure has been proposed. In 1991, the ElGamal procedure was adopted by the U.S. National Institute of Standards and Technology and registered under the name of Digital Signature Standard (DSS).

DSA Key Generation

The prime numbers p and q and the generator g are public domain parameters. They define a multiplicative *Abelian group* modulus p. The parameter $g \in [2, p-1]$ specifies a generator of the multiplicative cyclic subgroup $\langle g \rangle$ of order q. This mathematically implies that $q|(p-1)$ and no other smaller positive integer is a prime divisor of $p-1$ satisfying $g^q \equiv 1$. The private key x is randomly selected among the subgroup elements, i.e., $x \in [1, q-1]$, whereas the corresponding public key is generated by computing $y = g^x \bmod p$, as is shown in Algorithm 2.5. The problem of finding x given the domain parameters (p, q, g) and the public key y is known as the *discrete logarithm problem.*

DSA Digital Signature Algorithm

Once that the public/private key pair has been generated, a given entity A can generate the DSA signature $S = (r, s)$ of a message m by proceeding as follows (see Algorithm 2.6). First, A must select a random number $k \in [1, q-1]$, which must be secret and should be destroyed after the DSA has been generated. Then, A must compute $T = g^k \bmod p$, and $r = T \bmod q$. Thereafter, the message m is processed using a secure hash algorithm H so that $h = H(m)$ is

Algorithm 2.4 DSA Domain Parameter Generation

Require: Security parameters l and t.
Ensure: Domain parameters (p, q, g).
1: Select a prime number q of t bits and another prime number p of l bits such that $q|(p-1)$.
2: Find an element g of order q.
3: **repeat**
4: randomly select a number $h \in [1, p-1]$ and compute $g = h^{\frac{p-1}{q}} \mod p$.
5: **until** $\{g \neq 1\}$
6: **Return** (p, q, g).

Algorithm 2.5 DSA Key Generation

Require: Domain parameters p, q, g.
Ensure: Private key x and public key y.
1: Randomly select $x \in [1, q-1]$.
2: $y = g^x \mod p$;
3: **Return** (y, x).

computed. Then, the other component of the DSA signature can be computed as,

$$s \equiv k^{-1}(h + xr) \mod q \tag{2.2}$$

DSA signature is composed by the pair (s, r). The verifying entity B can check the correctness of the DSA based on the following observation,

$$k \equiv s^{-1}(h + xr) \mod q. \tag{2.3}$$

Which implies,

$$g^k \equiv g^{s^{-1}h} g^{xs^{-1}r} \mod p \tag{2.4}$$

Finally, knowing that $T = g^k \mod p$ and $y = g^x \mod p$, we have,

$$T \equiv g^{hs^{-1}} y^{rs^{-1}} \mod p \tag{2.5}$$

Lats equation corresponds to the computation accomplished by the verifier at line 8 of Algorithm 2.7. Therefore, the verifier entity B can assess the correctness of a DSA signature by verifying that the equality $r = T \mod q$ holds. This can be done by knowing the domain parameters (p, q, g), the public key y and the DSA signature (r, s). DSA signature generation and verification are shown in Algorithms 2.6 and 2.7, respectively.

2.5.4 Digital Signature with Elliptic Curves

Elliptic curves over real numbers are defined as the set of points (x, y) which satisfy the elliptic curve equation of the form:

$$y^2 = x^3 + ax + b \tag{2.6}$$

Algorithm 2.6 DSA Signature Generation

Require: domain parameters (p, q, g), Sender's private key x, message m.
Ensure: Signature (r, s).
1: randomly select $k \in [1, q-1]$.
2: $T = g^k \bmod p$;
3: $r = T \bmod q$;
4: **if** $r = 0$ **then**
5: Go to Step 1;
6: **end if**
7: $h = H(m)$;
8: $s = k^{-1}(h + xr) \bmod q$;
9: **if** $s = 0$ **then**
10: Go to Step 1;
11: **end if**
12: **Return** (r, s).

Algorithm 2.7 DSA Signature Verification

Require: Domain parameters (p, q, g), Sender's public key y, message m and signature (r, s).
Ensure: Accept/Reject.
1: **if** r, s are not in the interval $[1, q-1]$ **then**
2: **Return**("Reject")
3: **end if**
4: $h = H(m)$;
5: $w = s^{-1} \bmod q$;
6: $u_1 = hw \bmod q$;
7: $u_2 = rw \bmod q$;
8: $T = g^{u_1} y^{u_2} \bmod p$;
9: $\hat{r} = T \bmod q$;
10: **if** $r = \hat{r}$ **then**
11: **Return**(Accept);
12: **else**
13: **Return**(Reject);
14: **end if**

$$y^2 = x^3 + ax + b \tag{2.6}$$

where a and b are real numbers. Each choice of a and b produces a different elliptic curve as shown in Figure 4.1. The elliptic curve in Equation 2.6 forms a group if $4a^3 + 27b^2 \neq 0$. An elliptic curve group over real numbers consists of the points on the corresponding elliptic curve, together with a special point $\mathcal{O}$ called the point at infinity. Elliptic curve groups are additive groups; that is, their basic function is addition. The negative of a point $P = (x, y)$ is its reflection in the x-axis: the point $-P$ is $(x, -y)$. If the point P is on the curve, the point $-P$ is also on the curve.

In elliptic curve cryptography we are only interested in elliptic curves defined over finite fields. This means that the coordinates of the points in the elliptic curve can only take values that belong to the finite field over which, the elliptic curve has been defined. In particular we define elliptic curves over binary extension fields $GF(2^m)$, using the following adjusted curve equation,

$$y^2 + xy = x^3 + ax^2 + b \tag{2.7}$$

where $a, b \in GF(2^m)$ and $b \neq 0$. Once again, the elliptic curve includes all the points (x, y) that satisfy above equation in $GF(2^m)$ arithmetic, plus the point at infinity $\mathcal{O}$. The set of point that belong to the curve E is denoted as $E(\mathbb{F}_{2^m})$[8].

Elliptic Curve Domain Parameters

The *domain parameters* needed for obtaining a public key cryptosystem based on the elliptic curve discrete logarithm problem over $\mathbb{F}_q$ are the following [133],

1. The number of field elements (finite field order) q.
2. The coeffcients $a, b \in \mathbb{F}_q$ that define the elliptic equation E over $\mathbb{F}_q$.
3. A base point $P = (x_p, y_p) \in \mathbb{F}_q$ that belongs to the curve E. P must have a prime order.
4. The *order* n of P.
5. The *cofactor* $h = \#E(\mathbb{F}_q)/n$.

ECDSA Key Generation

Let $P \in E(\mathbb{F}_q)$ with order n, where E is an elliptic curve as defined above. We consider the field order q, the elliptic curve equation E and the base point P as public domain parameters. The private key d is a randomly chosen integer in the range $[1, n-1]$ and the corresponding public key is the point $Q = dP$ as computed in Algorithm 2.8 below. The problem of defining d given P and Q is known as the *elliptic curve discrete logarithm problem.*

Algorithm 2.8 ECDSA Key Generation

Require: Elliptic curve public domain parameters (q, E, P, n).
Ensure: public/private key pair $Q = (x_Q, y_Q)$ and d.
1: Randomly choose d in the range $[1, n-1]$
2: $Q = dP$;
3: **Return** (Q, d).

[8] Elliptic curve theory is covered in Chapter 4. Reconfigurable hardware implementations of elliptic curve cryptosystems are studied in Chapter 10.

ECDSA Digital Signature

Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA) [141]. It was accepted in 1999 as an ANSI standard, and in 2000 it was accepted as IEEE and NIST standards. Unlike the ordinary discrete logarithm problem and the integer factorization problem, no subexponential-time algorithm is known for the elliptic curve discrete logarithm problem. For this reason, the strength-per-key-bit is substantially greater in an algorithm that uses elliptic curves.

Algorithm 2.9 ECDSA Digital Signature Generation

Require: Domain parameters: (q, a, b, P, n, h), Sender's private key d, message m.
Ensure: Signature (r, s).
1: Randomly Select k in the interval $[1, n-1]$
2: $kP = (x_1, y_1)$; and convert x_1 into an integer $\overline{x}_1$.
3: Compute $r = \overline{x}_1 \bmod n$.
4: **if** $r = 0$ **then**
5: goto step 1;
6: **end if**
7: $e = H(m)$;
8: $s = k^{-1}(e + dr) \bmod n$.
9: **if** $s = 0$ **then**
10: goto step 1;
11: **end if**
12: **Return**(r, s).

The ECDSA digital signature algorithm is shown in Fig. 2.9. The signature for this message is the pair (r, s). It is to be noted that the signature depends on the private key and the message. This implies that, at least in theory, no one can substitute a different message for the same signature. Note that if a message m has a valid digital signature (r, s) then,

$$s \equiv k^{-1}(e + dr) \bmod n.$$

which implies,

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}dr \equiv we + wdr \equiv u_1 + u_2 \cdot d \bmod n.$$

Thus, $X = u_1P + u_2Q = (u_1 + u_2d)P = kP$, and consequently we validate the signature iff $v = r$. Above verification process is carried out by the procedure shown in Algorithm 2.10. Notice that in line 8 of that procedure, the elliptic curve point $X = u_1 \cdot P + u_2 \cdot Q$, is computed. As explained above, if the signature to be verified is a valid one then the equality $v = \overline{x}_1 \bmod n \equiv r$ should hold.

Algorithm 2.10 ECDSA Signature Verification

Require: Domain parameters: (q, a, b, P, n, h), signature (r, s), Sender's public key Q, message m.
Ensure: Reject/Accept.
1: **if** r, s are not in the interval $[1, n-1]$ **then**
2: **Return**("Reject")
3: **end if**
4: $e = H(m)$;
5: $w = s^{-1} \bmod n$;
6: $u_1 = ew \bmod n$;
7: $u_2 = rw \bmod n$;
8: $X = u_1 \cdot P + u_2 \cdot Q$;
9: **if** $X = \mathcal{O}$ **then**
10: **Return** "Rejected".
11: **end if**
12: Convert the x coordinate of X to an integer $\overline{x}_1$.
13: $v = \overline{x}_1 \bmod n$;
14: **if** $v = r$ **then**
15: **Return**(Accept);
16: **else**
17: **Return**(Reject);
18: **end if**

2.5.5 Key Exchange

In secret key cryptography, it is necessary that both parties at the sending and receiving ends agree on a secret key for transferring data in a secure way. Thus, several key agreement protocols have been proposed in order to establish a shared secret. The first such protocol is the Diffie-Hellman protocol, which provides the key establishment of a key with two message transfers. In the following, we will describe the basic Diffie-Hellman exchange protocol followed by its elliptic curve version.

Diffie-Hellman Key Exchange Protocol

Diffie-Hellman key exchange was invented in 1976 by Whitfield Diffie, Martin Hellman and Ralph Merkle. It was the first practical method for establishing a shared secret over an unprotected communication channel. Let us suppose that A and B have already agreed on working with a group G (for example, let us say the group of integers modulo p) and a generator element g in G. Then, the protocol dataflow is as follows (Figure 2.8):

- A picks a random natural number a and sends g^a to B.
- B picks a random number b and sends g^b to A.
- A computes K=$(g^b)^a$.
- B computes K=$(g^a)^b$.

In the Diffie-Hellman protocol, g and p are the domain parameters and K is the private key for the session which can be used as a shared secret for secure communication between A and B via symmetric cryptography.

Diffie-Hellman protocol is considered secure if G and g are chosen properly, i.e., the eavesdropper has an enormous difficulty to compute the element g^{ab}, because he/she needs to solve the discrete logarithm problem over the group G.

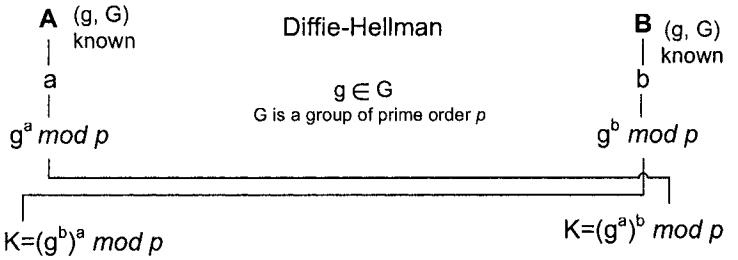

Fig. 2.8. Diffie-Hellman Key Exchange Protocol

Elliptic Curve Diffie-Hellman Key Exchange Protocol

Let A and B agree on an elliptic curve E over a large finite field F and a point P on that curve. Then the necessary steps for exchanging a secret key by using elliptic curve discrete logarithmic algorithm are as shown in Figure 2.9.

- A and B each privately choose large random integers, denoted r_1 and r_2.
- Using elliptic curve point-addition, A computes r_1P on E and sends it to B. B computes r_2P on E and sends it to A.
- Both A and B can now compute the point r_1r_2P by performing the elliptic curve scalar multiplication of the received value of r_2P, r_1P by his/her secret number r_1, r_2, respectively.

A and B agree that the x coordinate of this point will be their shared secret value.

2.6 A Comparison of Public Key Cryptosystems

Due to the high difficulty of computing the elliptic curve discrete logarithm problem, one can obtain the same security provided by other existing public-key cryptosystems, but at the price of much smaller fields, which automatically implies shorter key lengths. Having shorter key lengths means smaller

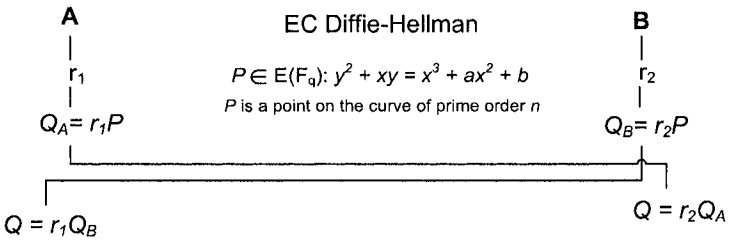

Fig. 2.9. Elliptic Curve Variant of the Diffie-Hellman Protocol

bandwidth and memory requirements. These characteristics are especially important in most embedded system applications, where both memory and processing power are constrained.

High performance implementations of elliptic curve cryptography depend heavily on the efficiency in the computation of the finite field arithmetic operations needed for the elliptic curve operations. On the other hand, the level of security offered by protocols such as the Diffie-Hellman key exchange algorithm relies on exponentiation in a large group. Typically, the implementation of this protocol requires a large number of exponentiation computations in relatively big fields. Therefore, hardware/software implementations of the group operations are, for all the practical sizes of the group, computationally intensive.

Nowadays, there exist algorithms able to solve the factorization problem as well as the discrete logarithm problems in a sub-exponential time. For instance, the Number Field Sieve (NFS) [203] is the best option for solving the integer factorization problem. The Number Field Sieve (NFS) [115] and the Pollard's rho algorithm [273] can solve the discrete logarithm problem.

In the case of RSA, the largest RSA modulus factored is a 640-bit (193-digit) integer in November, 2005 [195]. In the case of ECDSA, the largest known example was solved using the Pollard's rho method for both, prime and binary finite fields. The elliptic curve discrete logarithm problem for an elliptic curve over a 109-bit prime field was broken on November 2002 [44], whereas another elliptic curve defined over a 109-bit binary field was broken in April, 2004. The effort required 2600 computers and took 17 months [45].

2.7 Cryptographic Security Strength

Some of the major factors that determine the *security strength* of a given symmetric block cipher algorithm include, the quality of the algorithm itself, the key size used and the block size handled by the algorithm[9].

The security strength of an n-bit key symmetric block cipher algorithm, which has no known security flaws, is measured in terms of the amount of work it takes to try all possible keys, an attack traditionally known as the *brute-force* attack.

A generic cryptographic algorithm that has an m-bit key, but whose strength is comparable to an n-bit key of a strong symmetric block cipher algorithm is said to have an equivalent n-bit security strength. In general, however, the equivalent n-bit security strength of a given algorithm is less than m due to the possibility that certain specific attack to that algorithm may provide computational advantages compared with the brute-force attack [257].

Determining the security strength of an algorithm is not trivial. For example, one might expect that 3TDEA would provide $56 * 3 = 168$ bits of strength. However, the so-called *birthday and meet-in-the-middle attacks* on 3TDEA [227, 315] reduces the strength of 3TDEA to merely 112-bit equivalent security strength. In the case of 2TDEA, provided that the attacker can manage to gather approximately 2^{40} plaintext-cipher pairs, then 2TDEA would have a strength comparable to an 80-bit algorithm [257].

On the other hand and due to performance, functionality or compatibility reasons, algorithms of different strengths and key sizes are frequently combined in the same application. In general, the weakest algorithm and key size used for cryptographic protection determines the strength of the protection provided to the system. As an example, if SHA-512 is used with 1024-bit RSA, only 80-bit of security strength will be provided to data application. If the application requires 128 bits of security, then 3072-bit RSA key must be used. Alternatively, 256-bit ECC can be used to substitute RSA as a public key cryptographic engine.

Table 2.1 compares the security strengths of a set of algorithms divided into three categories: Symmetric block cipher algorithms, public key cryptosystems and hash functions. Notice, however, that novel or improved attacks and/or technologies may be developed in the future, leaving some of the algorithms included in Table 2.1 partially or completely broken. In particular, all hash functions listed in Table 2.1 have recently been subject of successful attacks, thus casting doubts on their security [368, 369, 103].

[9] The block size is also a factor that should be considered, since if a collision-attack is launched, collisions become probable after $2^{\frac{b}{2}}$ blocks have been encrypted with the same key for certain block ciphers' *modes of operation* [71, 70, 69].

Table 2.1. A Comparison of Security Strengths (Source: [258])

Private key Algorithm	**bit security**	**Expected Security lifetime**
Two-key triple DES	80	through 2010
Triple-key triple DES	112	through 2030
128-bit AES	128	beyond 2030
192-bit AES	192	beyond 2030
256-bit AES	256	beyond 2030
Public key Algorithm	**bit security**	**Expected Security lifetime**
DSA ($p = 1024, q = 160$)	80	through 2010
DSA ($p = 2048, q = 224$)	112	through 2030
DSA ($p = 3072, q = 256$)	128	beyond 2030
DSA ($p = 7680, q = 384$)	192	beyond 2030
DSA ($p = 15360, q = 512$)	256	beyond 2030
1024-bit RSA	80	through 2010
2048-bit RSA	112	through 2030
3072-bit RSA	128	beyond 2030
7680-bit RSA	192	beyond 2030
15360-bit RSA	256	beyond 2030
{160-223}-bit ECC	80	through 2010
{224-255}-bit ECC	112	through 2030
{256-383}-bit ECC	128	beyond 2030
{384-511}-bit ECC	192	beyond 2030
{512-}-bit ECC	256	beyond 2030
Hash functions	**bit security**	**Expected Security lifetime**
SHA-1	80	through 2010
SHA-224	112	through 2030
SHA-256,	128	beyond 2030
SHA-384	192	beyond 2030
SHA-512	256	beyond 2030

2.8 Potential Cryptographic Applications

During the last few years we have seen formidable advances in digital and mobile communication technologies such as cordless and cellular telephones, personal communication systems, Internet connection expansion, etc. The vast majority of digital information used in all these applications is stored and also processed within a computer system, and then transferred between computers via fiber optic, satellite systems, and/or Internet. In all those new scenarios, secure information transmission and storage has a paramount importance in the international information infrastructure, especially, for supporting electronic commerce and other security related services.

Under such a dynamic scenario, some of the most popular applications in the domain of information security include,

- *Secure e-mail*
- *World Wide Web*
- *Client-Server transactions*
- *Virtual Private Networks*
- *E-cash*
- *Electronic Financial transactions*
- *Grid Computing*

Many multinational firms now sell security products using cryptographic algorithms. Those products are in use by military or government organizations and they play a vital role in secure communications between individuals, small and large business groups.

Various international organizations have been working in developing standards for determining security and speed of products such as cellular phones, video conferencing equipment, secure telephone, etc. Examples include standards for video conferencing: H310, H323, H324 by ITU [154], for mobile communications: GSM by ETSI [87], for wireless LANs: 802.11a, 802.11b by IEEE LAN/MAN Committee [144], etc.

Numerous useful activities for increasing the security of cryptographic algorithms have happened in the few last years. The selection of the new Advance Encryption Standard (AES) 'Rijndael' and the inclusion of Elliptic curve cryptography (ECC) in international standards provide such examples.

Promising applications for cryptographic algorithms may be classified into two categories [250].

1. Processing of large amount of data at real time potentially in a high speed network. Examples include telephone conversation, telemetry data, video conferencing, streaming audio or encoded video transmissions and so forth.
2. Processing of very small amount of data at real time in a moderately high-speed network transmitted unpredictably. Examples include e-commerce or m-commerce transactions, credit card number transmission, order placement with signature, bank account information extraction, e-payments, and micro-browser-based (WAP-style) HTML page browsing and so forth.

A short list of the candidate applications corresponding to category 1 are presented in Table 2.2. Those applications belong to the "highly efficient" category of applications, thus requiring high data rates.

Table 2.2 presents both the downstream and upstream data transfer ranges on VDSL (Very high speed Digital Subscriber Line) [88, 252]. The downstream defines transmission of line terminal toward network terminal (from customer to network premise) and upstream in the reverse direction, that is, from network terminal to line terminal (from network to customer premise).

Table 2.2 can help to mark a line between high speed (highly efficient) and low speed (slow or relatively less speed) applications. The data rates for

Table 2.2. A Few Potential Cryptographic Applications

Application	Upstream	Downstream
Distance learning	384Kbps-1.5Mbps	384Kbps-1.5Mbps
Telecommuting	1.5Mbps-3.0Mbps	1.5Mbps-3Mbps
Multiple digital TV	6.0Mbps-24.0Mbps	64Kbps-640Kbps
Internet Access	400Kbps-1.4Mbps	128Kbps-640Kbps
Web hosting	400Kbps-1.5Mbps	400Kbps-1.5Mbps
Video conferencing	384Kbps-1.5Mbps	384Kbps-1.5Mbps
Video on demand	6.0Mbps-18Mbps	64Kbps-128Kbps
Interactive video	1.5Mbps-6.0Mbps	128Kbps-1.5Mbps
Telemedicine	6.0Mbps	384Kbps-1.5Mbps
High-definition TV	16Mbps	64Kbps

highly efficient applications ranges from 384Kbps to 24Mbits for upstream and 64Kbps to 3Mbps for the downstream traffic. From Table 2.2, the applications requiring a speed factor of less than 400Kbps can be grouped as low speed applications. Those applications require either stand-alone software implementations of cryptographic algorithms or the usage of software methods on embedded processors. High speed or highly efficient applications therefore reside in the range from 400Kbps onward.

Software methods on general-purpose processors cannot achieve such a high frequency gains for cryptographic algorithms. On the other hand, high speeds above 400Kbps can easily be achieved on both hardware platforms, the traditional (ASICs) and the reconfigurable hardware FPGA devices.

2.9 Fundamental Operations for Cryptographic Algorithms

Symmetric or secret key cryptographic algorithms are based on well-understood mathematical and cryptographic principles. The most common primitives encountered in various cryptographic algorithms are permutation, substitution, rotation, bit-wise XOR, circular shift, etc. This is one of the reasons for their fast encryption speed. On the other hand, asymmetric or public key cryptographic algorithms are based on mathematical problems difficult to solve. The most common primitives in various such types of algorithms include modular addition/subtraction, modular multiplication, variable length rotations, etc. Those primitives give algorithmic strength but they are hard to implement: occupy more space and consume more time.

Therefore those algorithms are not used for encrypting large data files, but rather, they are applied to other important cryptographic applications like key exchange, signature, verification, etc.

A detail survey conducted in [44], identifies the basic operations involved in several cryptographic algorithms. That survey has been slightly updated as shown in Table 2.3.

Table 2.3. Primitives of Cryptographic Algorithms (Symmetric Ciphers)

Modular addition or subtraction	Blowfish, CAST, FEAL,GOST, IDEA, WAKE RC5, RC6, TEA, SAFER K-64, Twofish, RC4 SEAL, TWOPRIME
Bitwise XOR	Blowfish, CAST, DEAL, TWOPRIME, FEAL, A5 IDEA, GOST, RC4, RC5, SAFER, SEAL, Twofish DES, WAKE, LOKI97, LOKI91, Rijndael, MISTY TEA, MMB, RC6, K-64
Bitwise AND/OR	MISTY
Variable-length rotations	CAST, Madryga, RC5, RC6
Fixed-length rotations	DEAL, DES, CAST, FEAL, GOST, Serpent, RC6 Twofish
Modular multiplication	CAST, IDEA, RC6, MMB, Rijndael,
Substitution	Blowfish, DEAL, DES, LOKI91, LOKI97, Twofish Rijndael
Permutation	DEAL, DES, ICE, LOKI91, LOKI97
Non-circular shifts	Serpent, TEA

From Table 2.3, it is clear that most cryptographic algorithms include bitwise operations such as XOR, AND/OR, etc. Those operations can be nicely implemented on hardware platforms. Long word length is another peculiarity of cryptographic algorithms, which is recommended by various international standards in order to attain sufficient security against brute force attacks.

The long key/word length of cryptographic algorithms is an obstacle for parallel dataflow on 8, 16, 32-bit general-purpose processors resulting on high time delays for the execution of crypto algorithms. This is not the case for hardware implementations. For example, in FPGAs, more than 1000 input/output pins are available for their use as either input or output buffers allowing high parallelism of data [392, 394].

In order to *confuse* the relationship between input and output, cryptographic algorithms perform a number of iterations on the same input data block for one encryption. DES performs 16 iterations or rounds and AES support 10, 12, and 14 rounds depending on the word length. In software, all iterations are performed sequentially while in hardware, all rounds can be implemented in parallel, thus ensuing significant improvements in timings.

2.10 Design Alternatives for Implementing Cryptographic Algorithms

The implementation approaches for cryptographic algorithms are based on the question: what needs to be secured?

High-speed network where large amount of data traffic must be processed in unpredictable and in real time are not supposed to be a good candidate for software implementations as data is coming at significant high speeds and must be treated in real time. Examples of such situation include telephone conversation, video conferencing, and so forth.

Hardware solutions on VLSI can accommodate high data rates but they take long development cycle for the application. Any change or modification in the design is a difficult or even impossible task.

A hardware solution that overcomes the difficulties of VLSI designs, while still allowing high dataflow, is reconfigurable hardware platforms. Indeed, Reconfigurable hardware devices such as FPGAs (Field Programmable Gate Arrays) provide fast solutions in short time with a high degree of flexibility.

Table 2.4 presents a quick comparison of reconfigurable logic against software and VLSI based solutions.

Table 2.4. Comparison between Software, VLSI, and FPGA Platforms

	Software	VLSI	FPGAs
Size	small (depends)	big	small
Cost	low	high cost	low cost
Speed	low	Very high	high
Memory	fine	fine	fine
Flexibility	highly flexible	no flexibility	highly flexible
Time-to-market	short	very high	short
Power consumption	depends	low	high
Testing/Verification	easy	difficult	easy
Run-time configuration	none	none	yes

Software implementations are low cost, easy to debug, take short time cycle but are slow. VLSI implementations are very fast but their application development cycle is too large and also they are not flexible. Reconfigurable devices are fast, highly flexible, easy to debug and take small developing cycle offering cost effective solutions.

In summary, using reconfigurable hardware for cryptographic algorithms is beneficial in several ways:

- Most cryptographic algorithms, especially symmetric ciphers, contain bit-wise logic operations whose implementation fits very well on the FPGA CLB structure.

- In Section 2.9, it was mentioned the iterative nature of most cryptographic algorithms. An iterative looping design (IL) implements only one round. Hence, n iterations of the algorithm are carried out by feeding back previous round results. For a high speed network, instead of implementing one round, n rounds of the algorithm can be replicated and registers are provided between the rounds to control the flow of data. Reconfigurable FPGA logic results useful for both design strategies due to its high speed and high-density features.
- Substitution is the fundamental operation in most block ciphers like DES or Rijndael which implies the usage of lot of memory resources. The usage of pipeline design strategies, tend to provoke significant memory requirements. Fortunately modern FPGA families like Xilinx Virtex series device are equipped with more than 280 built-in memory blocks 4K each, called *BlockRAMs* (BRAM).
- At the same time, in several contexts, designers may use reconfigurable FPGA logic to implement in the same hardware both the public key algorithm for the generation and secure exchange of key and the private key algorithm traditionally used in the bulk encryption of the underlying traffic.
- The usage of different cryptographic algorithms for various applications faces several compatibility issues. A dynamic configuration for any cryptographic algorithm on FPGA might be a good compromise solution to this problem.
- FPGA devices are ideal for debugging and fast prototyping, especially if the synthesized hardware description can be mapped by the design team from FPGA domain to ASIC.
- The flexibility for integration into larger platform together with straightforward architecture modifications are significant pluses for FPGA platform implementations.

2.11 Conclusions

In this Chapter we gave a short introduction to the algorithms and security services corresponding to layers 2-4 of Fig. 2.1. This way, basic concepts of cryptography along with a description of the main building blocks necessary for constructing security applications was given. We described the basic operation of symmetric block ciphers, hash functions, three major public key cryptosystems and the celebrated Diffie-Hellman key-exchange protocol. We also gave some comments on the security provided by the main cryptographic schemes and their equivalent security strength. Furthermore, alternatives for the implementation of cryptographic algorithms on various software and hardware platforms were also analyzed and discussed.

As a conclusion, we believe that Reconfigurable logic offers numerous useful advantages, however its usage in inexpensive consumer-oriented devices

such as electronic gadgets, wireless PDAs and handsets seems to be impossible at present time.

On the contrary, FPGA devices can be contemplated on embedded systems, large wireless equipments, electronic transmitters and receivers, repeaters, spectrum scanning devices, and intelligent equipment.

3

Reconfigurable Hardware Technology

An FPGA is an integrated circuit that belongs to a family of programmable devices called Programmable Logic Devices (PLDs). An FPGA contains tenths of thousands of building blocks, known as *Configuration Logic Blocks* (CLB) connected through programmable interconnections. Those CLBs can be reconfigured by the designers themselves resulting in a functionally new digital circuit, this way, virtually any kind of digital circuit can be implemented using FPGAs [11, 272, 304, 244].

At first, FPGA devices were mainly applied for logic design, and as a consequence of that, numerous tools were designed for synthesizing logic designs on them. Among those tools, Hardware Description Languages (HDL) and schematic diagram editors have been traditionally used as a starting point for such a synthesis process. Among the many hardware description languages available today, Verilog, and especially, VHDL, are the two most widely spread hardware languages.

In recent years, FPGAs have been used for reconfigurable computing when the main goal is to obtain high performance at a reasonable cost out of hardware implemented algorithms. The main advantage of FPGAs is their reconfigurability, i.e., they can be used for different purposes at different stages of a computation and they can be, at least partially, reprogrammed on run-time. The two most popular FPGA manufacturers are Xilinx [396] and Altera [4]. Those two makers have over 70% of the FPGA market share.

Besides Cryptography, applications of FPGAs can be found in the domains of evolvable and biologically-inspired hardware, network processors, real-time systems, rapid ASIC prototyping, digital signal processing, interactive multimedia, machine vision, computer graphics, robotics, embedded applications, and so forth. In general, FPGAs tend to be an excellent choice when dealing with algorithms that can benefit from the high parallelism offered by the FPGA fine-grained architecture.

In this chapter we present the generalities of FPGA technology. We stress that the material of this Chapter is mainly intended for those readers non-familiar with this technology.

We begin in Section 3.1 by reviewing some historical milestones of FPGA development and then we review in Section 3.2 the two most currently used FPGA technologies, namely, Xilinx and Altera. Then we compare in Section 3.3 the performance of FPGA realizations against the ones on ASICs and general-purpose processor platforms. We continue in Section 3.4 by briefly introducing the reconfigurable computing paradigm main concepts. In Section 3.5 we review several key strategies to achieve good designs for cryptographic applications. Then, we define in Section 3.6 several metrics and figures of merit needed to evaluate design performance for reconfigurable computing as well as several security concerns related to FPGA technology. In Section 3.7 we give a brief overview of some of the security concerns and attacks on FPGA technology. Finally, in Section 3.8 concluding remarks are given.

More experimented readers might be interested in reviewing more advanced material. For them, we recommend excellent sources such as the ones found in [124, 365, 217, 199, 192]. Those readers having more technology oriented interests may profit from consulting [259, 244] as well.

3.1 Antecedents

The concept of reconfigurable computing was first introduced by G. Estrin in 1960 [101]. His invention consisted of a hybrid machine composed by a general purpose microprocessor interconnected with programmable logic devices. The programmable logic could be configured for accomplishing a specific function with the characteristic efficiency of hardware designs. Once the function was completed, another task could be performed by manually reconfiguring the hardware. This resulted in a hybrid computer structure combining the best features of software (flexibility) and hardware (speed) platforms. It is nothing but remarkable how Estrin's concept come close to what is offered by nowadays modern reconfigurable devices [217].

In the mid 1970s, Programmable Logic Devices (PLDs) were introduced by companies such as IBM, Monolithic Memories, Inc (MMI) and AMD. The first PLDs were called PAL (Programmable Array Logic) or PLA (Programmable Logic Array) depending on the programming scheme utilized [272]. Earlier PLDs consisted of logic gate arrays with no clocked memory components. However, *registered* PLDs including one flip-flop at each output of the circuit, were soon available. Register PLDs allowed for the first time the design of simple reprogrammable sequential circuits.

An innovation of PAL devices was the Generic Array Logic (GAL) device, which had the same logical properties as the PAL but the functionality could be erased and reprogrammed. From the point of view of today's standards,

PALs and GALs devices are small devices having an equivalent computational power of just some few hundred logic gates.

As a consequence of Moore's law, the semiconductor technology has experienced an unrelenting improvement over the last three decades. That allowed the integration in the mid 1980s of several either GAL or PAL devices on the same chip, thus given birth to the CPLD (Complex PLD) devices. CPLDs can emulate the computational power of hundreds of thousands of logic gates and they are still very popular due to their outstanding cost-benefit compromise (some CPLD devices can be bought for less than a dollar). A typical modern CPLD device has a structure consisting of several GAL blocks whose outputs are connected to a switch matrix used for programming the interconnections as well as the Input/Output pins. Each GAL block consists of one or more programmable sum-of-products logic arrays ended with a relatively small number of registers. CPLDs are usually programmed via a serial data port that can be connected to a personal computer. Internally, the CPLD contains a decoding module that interprets the data stream in order to perform a specific logic function. The preferred standard for this programming method is the IEEE 1149.1 standard usually known as Joint Test Action Group (JTAG) interface [272]. As of 2006, most CPLDs are non-volatile electrically-erasable programmable devices.

Field Programmable Gate Array (FPGA) devices were introduced by Xilinx in the mid 1980s. Roughly speaking, FPGA devices are built using a grid of logic gates. They differ from CPLDs in several key aspects. FPGA architectures consists of a matrix of Configurable Logic Blocks (CLBs) interconnected by an intricate array of switch matrices. This architecture provides great flexibility to hardware designers but it also implies much more sophisticated routing technologies [123]. The fact that most modern FPGAs have higher-level embedded modules such as built-in multipliers, distributed RAM blocks and so on is another important difference with CPLD devices. Moreover, in contrast to CPLD devices, most modern FPGAs support (at least partially) *in-system* reconfiguration, thus allowing designs to be changed dynamically "on run-time". This feature can be particularly useful for system updates.

Significant technical advances have led to architectures that combine FPGA's logic blocks and interconnect matrices, with one or more microprocessors and memory blocks integrated on a single chip. This hybrid technology is called Configurable System-on-Chip (CSoC). Examples of the CSoC technology are the Xilinx Virtex-II PRO, and the Virtex-4 and Virtex-5 FPGA families, which include one or more hard-core PowerPC processors embedded along with the FPGA's logic fabric [398, 396, 397].

Alternatively, *soft* processor cores that are implemented using part of the FPGA logic fabric are also available. This approach is more flexible and less costly than the CSoC technology [217]. Many soft processor cores are now available in commercial products. Some of the most notorious examples are: Xilinx 32-bit MicroBlaze and PicoBlaze, and the Altera Nios and the 32-bit

Nios II processor [394, 5]. These soft processor cores are configurable in the sense that the designer can introduce new custom instructions or processor data paths. Furthermore, unlike the hard-core processors included in the CSoC technology, designers can add as many soft processor cores as they may need (some designs could include 64 such processors or even more [130, 217]).

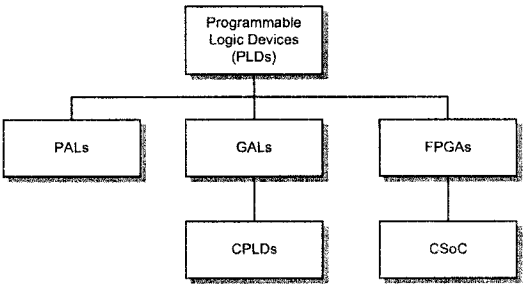

Fig. 3.1. A Taxonomy of Programmable Logic Devices

Fig. 3.1 shows the taxonomy of the programmable logic devices just discussed. In the next Section, more specific details of the FPGA device internal architecture are given.

3.2 Field Programmable Gate Arrays

In a very rough way, an FPGA can be seen as a matrix of Configurable Logic Blocks (CLBs), where not only the logic but also the connection is user programmable. The specific design of the CLB blocks varies from manufacturer to manufacturer and even, from device to device. A CLB can be as simple as just one four-input Look Up table (LUT) or as complex as a 4-input Arithmetic Logic Unit (ALU), or a 6-input Look Up Table [398]. It is customary to define the *granularity* of the reconfigurable logic as the size of the smallest functional unit that can be addressed by the programming tools.

Architectures having finer granularity tend to be more useful for data manipulation at bit level and, in general, for combinatorial circuits. On the other hand, blocks with a *coarse grain* granularity are better suited for higher levels of data manipulation, for example, for developing circuits at register-transfer level. The level of granularity has a great impact in the device configuration time. Indeed, a device with low granularity (also known as fine-grained devices) requires many configuration points producing a bigger vector data for reconfiguration. That extra routing has an unavoidable cost on power and area.

On the other hand, a coarse grained architecture tends to decrease its performance when dealing with computations smaller than what its granularity

is. For example, if for a specific application, bit-level operations are required and the smallest functional unit is four-bit wide, then a waste of three bits would occur.

FPGA interconnection has a major role in the performance of an FPGA device due to the need of fast and efficient communication highways among the different logic blocks which are organized by rows and columns. Xilinx devices[1] are equipped with four kinds of interconnects: long lines, hex lines, double lines and direct lines. Direct connect lines are intended for connecting neighbor components (for example, carry circuitry). Hex and double lines are medium length interconnects aimed for connecting many CLBs. Finally long lines interconnects are implemented along the whole chip and are normally utilized for global system signals.

In recent years, huge technological developments have had a great impact on FPGA industry. The most advanced FPGA devices operate up to 550 MHz internal clock with a gate complexity of over 10 Million gates on a single Virtex-5 FPGA chip using a technology of just 65 ηm operating at 1.0V [395]. The improvements in technology are not only limited to an ever growing internal number of logic gates but also to the addition of many functional blocks like fast access memories, multipliers or even microprocessors integrated within the same chip.

There are quite a few FPGA commercial manufacturers, and usually each one of them has developed one or more device families. Table 3.1 shows some of the most popular manufacturer families.

Table 3.1. FPGA Manufacturers and Their Devices

Manufacturer	FPGA Family	Feature
Xilinx	Virtex-5, Virtex-4, VirtexII, Spartan III	FPGA market leader 65ηm technology
Altera	Stratix, Stratix II, Cyclone	90ηm technology
Lattice	LatticeXP	first non-volatile FPGA
Actel	Fusion, M7Fusion	first mixed-signal FPGA
Quick Logic	Eclipse II	programmable-only-once FPGA
Atmel	AT40KAL	fine-grained reconfigurable
Achronix	Achronix-ULTRA	1.6GHz - 2.2GHz speed

3.2.1 Case of Study I: Xilinx FPGAs

Table 3.2 shows the main features that are included in the Xilinx FPGA families: Virtex-5, Virtex-4, Virtex II Pro and Spartan 3E. The architecture of those Xilinx FPGA families consists of five fundamental functional elements,

[1] At the time that this book was being written, Xilinx released the Virtex-5 family which has a radically different CLB interconnection pattern [395].

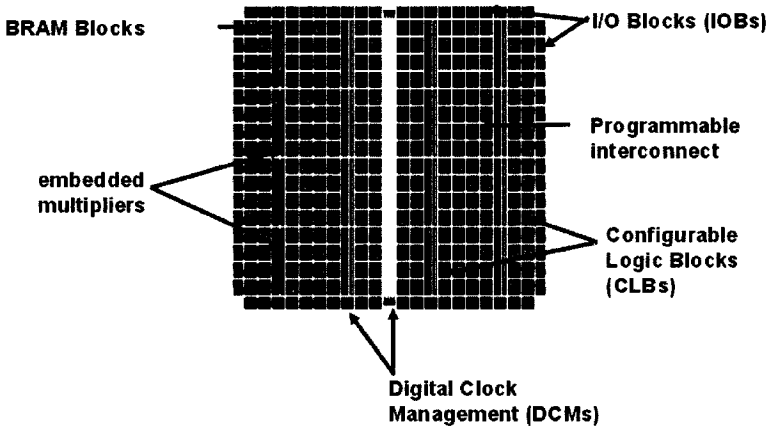

Fig. 3.2. Xilinx Virtex II Architecture

Table 3.2. Xilinx FPGA Families Virtex-5, Virtex-4, Virtex II Pro and Spartan 3E

Feature/family	Virtex-5	Virtex-4	Virtex II Pro	Spartan 3 & 3E
Logic Cells	up to 330K	12K-200K	3K-99K	1.7K-74K
BRAM (18Kbits each)	576	36-512	12-444	4-104
Multipliers	32 – 192[1]	32-512	12-444	4-104
DCM	up to 18	4-20	4-12	2-8
IOBs	up to 1200	240-960	204-1164	63-633
DSP Slices	32-192	32-192	—	–
PowerPC Blocks	N/A	0-2	0-2	–
Max. freq.	550MHz	500MHz	547 MHz	up to 300MHz
Technology	1.0V, 65ηm copper CMOS	1.2V, 90ηm, triple-oxide process	1.5V, 130ηm, 9-layer CMOS	1.2V, 90ηm, triple-oxide process
Price	N/A	From $345	From $139	From $2 up to $85

[1] 25 × 18 embedded multipliers

- Configurable Logic Block (CLB) and Slice architecture;
- Input/Output Blocks (IOBs);
- Block RAM;
- Dedicated Multipliers and;
- Digital Clock Managers (DCMs).

Those components are physically organized in a regular array as shown in Fig. 3.2. In the following we explain each one of those five elements[2].

[2] Virtex-5 devices can be considered second generation FPGA devices. In particular, a Virtex-5 slice contains four true 6-input Look Up Tables (LUTs).

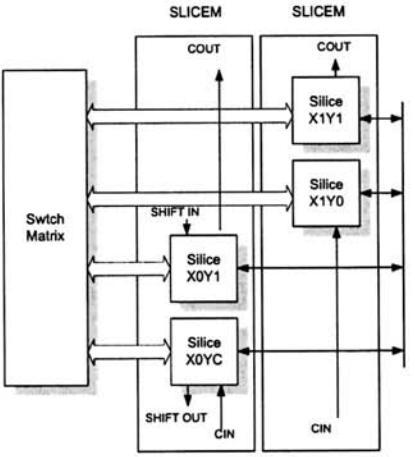

Fig. 3.3. Xilinx CLB

Configuration Logic Blocks (CLBs)

The Configurable Logic Blocs (CLBs) are the most important and abundant hardware resource of an FPGA. They are typically utilized for both, combinatorial and synchronous logic design. Each CLB is composed of four *slices*[3], which are interconnected as shown in Fig. 3.3. The slices are grouped by pairs and each pair is organized by a column with independent carry chain [395].

All four slices have the following common elements: two Look-Up Tables (LUTs), two type D flip-flops, multiplexers, logic circuits for carry handling and arithmetic logic gates. Both, the left and right pair of slices utilize those elements for providing logic functions, arithmetic and ROM. Besides that, the left pair supports two additional functions: data storage using a distributed RAM and 16-bit shift register functionality. Fig.3.4 shows the internal structure of a CLB. The atomic building block of a Virtex CLB is the logic cell (LC). An LC includes the Look-Up Table block, carry logic, and a storage element (flip-flop) as shown in Figure 3.5.

As it was mentioned, a CLB can be configured to work into two modes: logic mode and memory mode. As shown in Fig. 3.6, in logic mode, each CLB Look Up Table behaves as a combinational logic block and a one bit register. In the case of Xilinx devices those Look Up Tables can be reprogrammed to any arbitrary combinational logic function of four inputs/one output. In memory mode, Look Up Table blocks behave as two small pieces of memory blocks.

[3] Slice is a term introduced by Xilinx. It specifies a basic processing unit in a Xilinx FPGA.

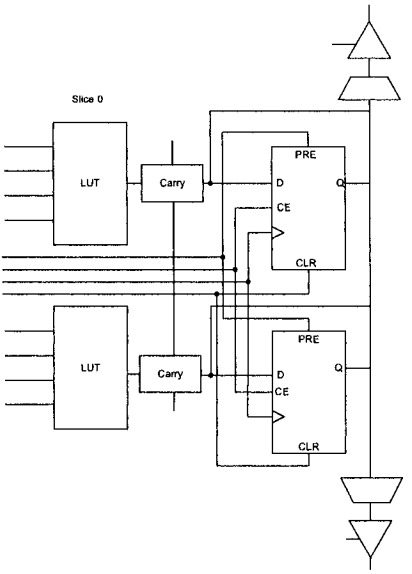

Fig. 3.4. Slice Structure

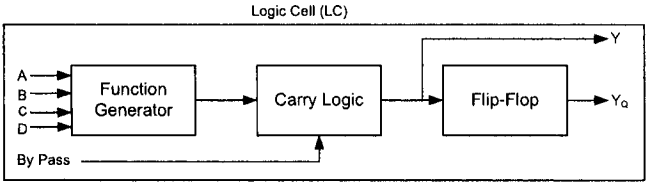

Fig. 3.5. VirtexE Logic Cell (LC)

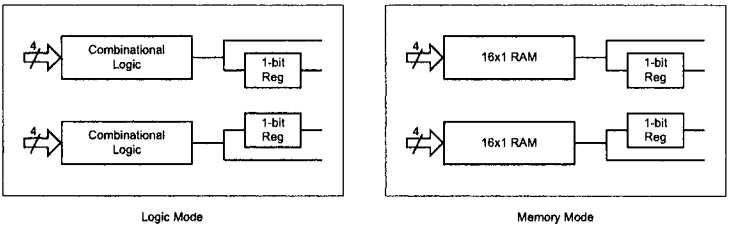

Fig. 3.6. CLB Configuration Modes

Input/Output Blocks

Input/output Blocks (IOB) provide a bidirectional programmable interface between the outside world and the internal logic structure of the FPGA device.

There exist three types of routing possibilities for an IOB: output signal, input signal and third state (high impedance) signal. Each one of those signals has their own pair of storage elements that can behave as registers or as latches [395].

Block RAM

Virtex devices include built-in 18K-bit RAM memory, called BRAM. BRAMs can be configured in a synchronous manner. BRAMs are intended for storing big amounts of data, while the distributed RAM is more useful for storing small amounts of data.

BRAMs are polymorphic blocks in the sense that its width and depth can be configured. Even multiple blocks can be connected in a back-to-back configuration in order to create wider and/or deeper memory blocks. A BRAM block supports several configuration modes, including single or double port RAM and several possible combination of data/address sizes as is shown in Table 3.3.

Table 3.3. Dual-Port BRAM Configurations

Configuration	Depth	Data bits	Parity bits
16K × 1 bit	16Kb	1	0
8K × 2 bit	8Kb	2	0
4K × 4 bit	4Kb	4	0
2K × 9 bit	2Kb	8	1
1K × 18 bit	1Kb	16	2
512 × 36 bit	512	32	4

18x18 Bit Multiplier

Xilinx FPGAs have several dedicated multiplier blocks. Those multipliers accept two 18-bit operands in two's complement form computing their product also in two's complement form. Such multipliers blocks have been optimized for performing at a high speed while their power consumption is kept low when compared with multipliers directly implemented using the CLB resources. The total number of multipliers varies from device to device as is shown in Table 3.2.

Digital Clock Managers

Digital Clock Managers (DCMs) provide a flexible control over clock frequency, phase shift and skew. The three most important functions of DCMs are: To mitigate clock skew due to different arrival times of the clock signal,

to generate an ample range of clock frequencies derived from the master clock signal and, to shift the signal of all its output clock signals with respect to the input clock signal.

3.2.2 Case of Study II: Altera FPGAs

Altera offers a wide variety of programmable hardware devices which are grouped into four categories [4].

- Complex Programmable Logic Devices(CPLDs)
- Low-Cost FPGAs
- High-density FPGAs
- Structured ASICs

CPLDs

Altera's CPLDs include MAX (EPM3032A, EPM3512A) and MAX-II (EPM 240/G, EPM 2210/G) family of devices. They are low complexity, low density and easy to use CPLD family for which software tools can be downloaded from Internet and they are free of cost.

Low-Cost FPGAs

Cyclone (EP1C3,EP1C20) and Cyclone-II (EP2C5, EP2C7) family of devices are considered low cost FPGAs. Their main features include embedded DSP blocks, on chip memory modules and support for embedded processor (NIOS).

High-Density FPGAs

The category of high density FPGAs from Altera comprises Stratix-II (EP2S15, EP2S180), Stratix (EP1S10, EP1S80), Stratix$_{GX}$-II (EP2SGX30C/D, EP2SGX130G) and Stratix$_{GX}$ (EP1SGX10C, EP1SGX40G) family of devices. Stratix and Stratix-II families are general purpose FPGAs with fast performance, large on-chip memory modules, and DSP blocks. Stratix$_{GX}$ and Stratix$_{GX}$-II families, in addition, include integrated transceivers.

Structured ASICs

Structured ASICs comprise Hardcopy (HC1S25, HC240) and Hardcopy-II (HC210W, HC240) solutions. They have similar design flow as that of Stratix and Stratix-II respectively. They are low cost structured ASIC solutions with sufficient number of gates supported by all major EDA vendors.

To provide an idea of what kinds of resources are present in Altera FPGA devices, let us discuss the structure of the Stratix family of devices. Detailed

data sheets of Stratix as well as all other Altera devices can be consulted in [4, 207, 208]. The quantitative information presented in this subsection has been extracted from [4]. Table 3.4 provides a quantitative measure of Stratix major resources, while Fig. 3.7 shows the physical distribution of those resources.

Table 3.4. Altera Stratix Devices

Feature	*Device*						
	EP1S10	EP1S20	EP1S25	EP1S30	EP1S40	EP1S60	EP1S80
Logic Elements	10,570	18,460	25,660	32,470	41,250	57,120	79,040
M512 RAM Blocks	94	194	224	295	384	574	767
M4K RAM Blocks	60	82	138	171	183	292	364
M-RAM Blocks	1	2	2	4	4	6	9
Total RAM bits	0.9205M	1.669M	1.945M	3.317M	3.423M	5.215M	7.427M
DSP Blocks	6	10	10	12	14	18	22
Embedded Multipliers	48	80	80	96	112	144	176
PLLs	6	6	6	10	12	12	12
Maximum I/O Pins	426	586	706	726	822	1022	1203

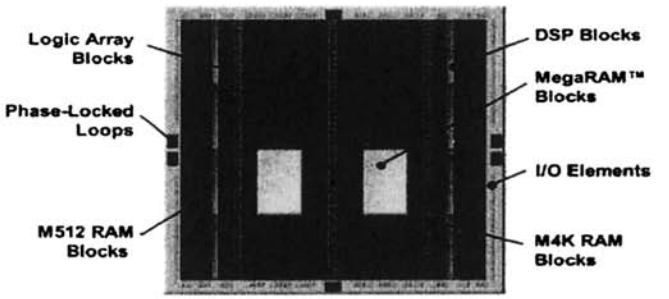

Fig. 3.7. Stratix Block Diagram

As shown in Fig. 3.7, the main building blocks in Stratix devices are the following:

- Logic Array Blocks (LABs)

- Memory Blocks
- Digital Signal Processing (DSP) Blocks
- Input/Output Elements (IOEs)
- Interconnects

Logic Array Blocks (LABs)

LABs are arranged in rows and columns across the device. Each LAB consists of 10 Logic Elements (LE). An LE is the smallest unit in Stratix architecture. It contains four input LUT, carry chain with carry select capability and a programmable register as shown in Fig. 3.8. The LUT serves as a function generator which can be programmed to any function with four variables. By using LAB-wide control signal, a dynamic addition or subtraction mode can also be selected. It is to be noted that number of resources are not fixed for an LAB in all kind of Altera devices. As an example, a LAB in Stratix-II architecture comprises 8 Adoptive Logic Modules (ALM) where each ALM contains a variety of LUT-based resources.

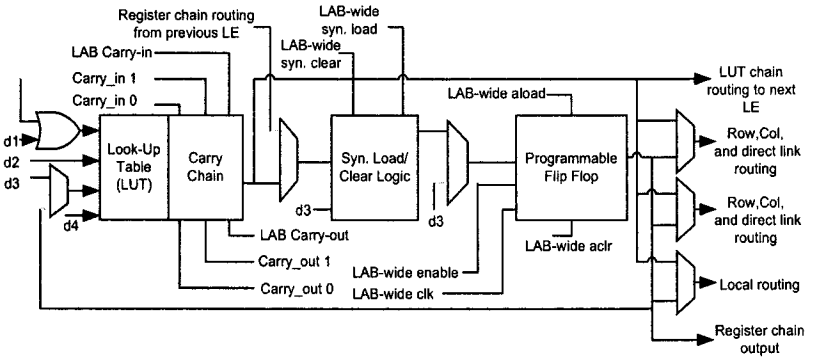

Fig. 3.8. Stratix LE

The Stratix LE can be configured into two modes:

- Normal mode
- Dynamic arithmetic mode

In normal mode, a four input LUT can be used to implement any function. The normal mode is therefore useful for implementing combinational logic and general logic functions. In dynamic arithmetic mode, an LE utilizes four 2-input LUTs which can be mapped to a dynamic adder/subtractor. First two LUTs perform two summations with possible carry-in and the other two LUTs compute carry outputs to drive two chains of the carry select circuitry. The

arithmetic mode is therefore useful for wide range of applications like adders, accumulators, wide parity functions, etc.

Memory Blocks

Three types of memory blocks are present in Stratix devices as shown in Fig. 3.7. Those are referred to as M512 RAM, M4K RAM and M-RAM (MegaRAM) blocks. M512 RAM is a simple dual port memory with sizes of 512 bits plus parity (576 bits). It can be configured as a maximum 18-bit wide single or dual port memory at up to 318 MHz. M4K is a true dual port memory with 4K bits plus parity. It can be configured as a maximum 36-bit wide dedicated dual port, simple dual or single port memory at 291 MHz. Several M-RAM blocks can also be located individually in logic arrays across the device. It is a true dual port memory with 512K bits plus parity (589,824 bits). A single M-RAM can be configured as a maximum 144-bit wide dedicated dual port, simple dual or single port memory which can operate at 269 MHz.

DSP Blocks

Those are dedicated Stratix resources which are vertically arranged into two columns in each device. DSP blocks can be configured into either eight 9×9-bit multiplier, four 18×18-bit multiplier or one full 36×36 multiplier. In addition, DSP blocks also contain 18×18-bit shift registers, Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters.

Input/Output Elements (IOEs)

Large number of IOEs can be located at the end of LAB row or column around the periphery of a Stratix device as shown in Fig. 3.7. Each I/O element comprises a bi-directional I/O buffer and six registers for buffering input, output and output-enable signals. Each Stratix I/O pin is fed by an I/O element and support several single-ended and differential I/O standards.

Interconnects

All LEs within the same LAB, or all LABs within the same device or Memory blocks or DSP blocks can be interconnected. A single LE can drive 30 other LEs through locally available fast and direct link interconnects. A direct link is also used by adjacent LABs, memory and DSP block to drive LABs local interconnects. The availability of direct links helps in reducing row and column interconnects resulting on higher performance and flexibility.

Table 3.5. Comparing Cryptographic Algorithm Realizations on different Platforms

Algorithm	FPGA		ASIC		μProcessor	
	Throughput	year	Throughput	year	Throughput	year
MD5	5.86 Gbps [156]	2005	2.09 Gbps [312]	2005	1.27Gbps (est)* [31]	1996
SHA-1	0.9 Gbps [67]	2002	2.006 Gbps [312]	2005	0.678Gbps (est)* [31]	1996
DES	21.3 Gbps [301]	2003	10Gbps [381]	1999	0.127Gbps [22]	1997
AES	25.1Gbps [113]	2005	7.5Gbps [303]	2001	0.8Gbps[109]	2004
1024-bit RSA	6.1 mS [6]	2005	1.47mS [210]	2005	22.1mS [294]	2004
ECC (binary)	17.64μS [54]	2006	190μS [313]	2003	475μS [133]	2004
ECC (prime)	3600μS [262]	2001	190μS[313]	2003	325μS [133]	2004

*Estimated for a 2GHz Pentium IV from the clock cycle count given in [31]

3.3 FPGA Platforms versus ASIC and General-Purpose Processor Platforms

Table 3.5 presents a quick performance comparison of several relevant cryptographic algorithms implemented in three different platforms: Reconfigurable hardware devices, ASIC and general purpose processors. We included implementations for hash functions (MD5 and SHA-1), block ciphers (DES and AES) and public key cryptography (RSA and ECC). All those algorithms will be studied in the next Chapters.

Referring to Table 3.5, it is noticed that software implementations are always slower than either, ASIC or FPGA implementations. The performance gap of software implementations is more noticeable for block ciphers and for the binary elliptic curve cryptosystem. On the contrary, the best reported prime elliptic curve cryptosystem is faster than the fastest FPGA design reported in [262].

We stress that the information included in Table 3.5 is intended for a first order comparison. As it has been already mentioned, it is extremely difficult to make fair performance comparisons among designs implemented in different platforms using the different technologies available at the time of their publications. In the rest of this Section we give some more insights about the advantages/disadvantages of implementing a design on reconfigurable hardware compared with other platform options.

3.3.1 FPGAs versus ASICs

Traditionally, in the design of embedded systems, the Application-Specific Integrated Circuit (ASIC) technology has played a major role for providing high performance and/or low cost building blocks necessary for the vast majority of systems during the (usually) large and sinuous design cycle. In 1980 the usage of reprogrammable components was introduced, and short after that the first FPGA device was developed by Xilinx. FPGA devices offer shorter

design cycle because of its ability of providing fast and accurate functionality testing.

However, the relatively high size and power consumption shown by FPGA devices has been the most important drawback of that technology towards an eventual substitution of the virtually ubiquitous ASIC technology. Therefore, historically FPGAs have been utilized primarily for prototyping development.

In recent years, however, FPGA manufacturers have significantly reduced the gap that still exist between FPGA and ASIC technology, paving the way for the utilization of FPGA not only as prototype tools but also as key components of embedded systems or even, becoming the system itself [364, 149, 331, 199].

However, the exact size of the performance gap between FPGAs and ASICs is currently subject of intense analysis and debate. Recently, several experimental results reported in [192], seems to suggest that for circuits designed utilizing the FPGA fabric only (i.e., LUTs and flip flops), an FPGA design is on average 40 times larger, consumes 12 times more dynamic power and it is 3.2 times slower than a standard ASIC implementation. On the other hand, in [364] it was developed a low-power FPGA core which was specially tailored for battery-powered applications such as those found in the automotive industry. The experimental results show that this solution is competitive with similar ASIC solutions.

Undoubtedly, new technological challenges must be faced for both, FPGA and ASIC platforms when the 45 ηm and 32 ηm technologies come to place. Under this scenario, it is not certain how FPGA new architectures will deal with the power consumption issue. It might be the case that manufacturers would need to trade device performance for a more flexible/predictable device power-consumption [141].

3.3.2 FPGAs versus General-Purpose Processors

The speedup that one can expect by implementing an algorithm on an FPGA device rather than using a general purpose processor (i.e. the traditional CPU) has been well documented in the literature [365, 124]. In [124], speedups of one to two orders of magnitude were measured when executing benchmarks applications in the domains of video and image processing. Roughly speaking, the same range of speedups has been confirmed in cryptographic algorithms.

From the qualitative point of view, it is interesting to study the main factors that produce this phenomenon. On the one hand, the typical maximum clock frequency achieved by FPGA designs fall in the range of 20MHz to 100MHz, while embedded microprocessors have frequencies ranging from 300 to 600 MHz and high-end workstation-class processors have frequencies of up to 3.2GHz. Hence, the clock frequency of general-purpose processors is 10-100 times faster than the typical clock frequency found in FPGA designs. On the other hand, there are two factors that help to compensate and even overcome that component, namely,

1. *FPGA Iteration-level parallelism*, obtained by, among others, loop-unrolling, pipeline and sub-pipeline techniques, and;
2. *FPGA Instruction efficiency*, obtained by carefully designed datapaths, the insertion of distributed memory blocks as needed and, taking advantage of the FPGA low granularity, the elimination of several instructions.

Those two factors combine together for obtaining a notable reduction in the total number of clock cycles required by an FPGA implementation. That reduction implies that CPU implementations may require up to 2500 times more clock cycles than that of FPGA implementations [124]. In other words, even though CPU platforms enjoy a much higher operating clock frequency, this factor is not enough for compensating the enormous clock cycle reduction that can potentially be obtained in FPGA platforms.

In the context of Moore's Law, an examination of peak floating-point performance trends for FPGA and CPU platforms is presented in [365]. The author concludes that although CPUs' performance obeys Moore's law (i.e., it doubles every 18 months), FPGA performance is growing at a rate of four times every two years. For applications using the FPGA new functionality (embedded multipliers, RAM blocks, etc.) the performance increase rate may be as high as five times every two years.

3.4 Reconfigurable Computing Paradigm

Reconfigurable computing may be defined as computer processing with highly flexible computing fabric. The main idea of reconfigurable computing is to take advantage of the best of two scenarios: flexibility from general purpose computing and speed from reconfigurable logic.

Some of the reconfigurable computing distinguished features when compared to general purpose microprocessors are [123]:

- Due to the inherent fine-grained granularity the parallelism tends to be very high.
- Registers, latches and even distributed RAM blocks can be created and distributed wherever needed by the data path. This characteristic has a tremendous impact on the device performance because reduces unnecessary re-computations and/or memory accesses.
- The amorphous nature (lack of a fixed architecture) of reconfigurable computing devices, allows the designers to tailor design's data path and control flow arbitrarily.

FPGAs can be properly used for rapid prototyping algorithms at hardware level. Considering the restrictions of FPGA devices, desirable FPGA applications should belong to one or more of the categories listed below.

1. Applications that employ only integer arithmetic or at most low precision fixed point arithmetic.

2. Applications that rely on logical operations to make decisions. Comparators, selectors and multiplexers are good examples of that.
3. Applications amenable for being decomposed in independent and pipelined stages.
4. Applications that show regularity in the way they apply a processing.
5. Applications with locality in the interconnection network they require. That means that the application modules should only have interconnections with their neighbors.

Considering FPGA capabilities and limitations some potential applications for FPGAs are:

1. Image processing algorithms such as point type operations (grey scale transformation, histogram equalization, requantization, etc.) and filtering (template matching, window techniques, convolution/correlation, median filtering, etc.) seem to be good candidates for FPGA implementation.
2. Dynamic programming algorithms requiring only integer arithmetic. Dynamic programming is in essence a bottom up procedure in which solutions to all subproblems are first calculated and then these results are used to solve the whole problem. A good example of this approach is the Floyd's shortest path algorithm.
3. Relaxation techniques requiring fixed point arithmetic. The relaxation technique is an iterative approach useful to many problems, which updates in parallel at each point and in each iteration based on the data available in the most recent updating or in the immediate preceding iteration.
4. Associative retrieval operations. Filling and retrieving data by association appears to be a powerful solution to many high volume information processing elements. An associative processing system is very adequate at recognition and recall from partial information and has remarkable error correcting capabilities. The major advantage of associative memory over RAM is its capability of performing parallel search and parallel comparison operations. There are many examples of that kind of applications: pattern matching, artificial intelligence, computer vision, data encoding, compression, and every application maintaining a dictionary data structure.
5. Highly regular and iterative applications with non-standard word lengths. Cryptography is a meaningful example of this kind of applications since it applies basic transformations mostly based on bit-level operations. Those basic operations are performed in long wordlengths starting from 128 bits to up 4096 bits or even in wordlengths non-standard, such as 163 and 233 bits (in the case of public-key cryptography). The basic transformations are repeated iteratively a number of times to process information in stages. In the following chapters we will explain how to take advantage of cryptographic algorithm features for reconfigurable computing.

3.4.1 FPGA Programming

The design cycle for programming FPGAs starts with a behavioral description of the design, using either hardware description languages (HDLs) such as VHDL or Verilog or a schematic design entry. Thereafter, the HDL code is compiled in order to produce a *netlist* which represents the mapping of the HDL code to the actual target device hardware resources. After the first compiling step, the netlist is reprocessed in order to perform the place-and-route process whose main goal is to establish how the different design's modules are going to be physically allocated and connected. This will create a binary file which is used for programming or reprogramming the FPGA device. Most designs included in this book have been compiled using the Xilinx Integrated Software Environment (ISE) version 8.1i software [393].

Hardware Description Languages (HDLs) are analogous to other high level languages (C, C++, etc.) with some significant differences. Both types are processed by a compiler, and both of them are function-oriented languages. However they differ in the way that the compiled code is executed. HDL languages are used for formal description of electronic circuits. They describe circuit's operation, its design, and tests to verify its operation by means of simulation. Typical HDL compilers tools [393], verify, compile and synthesize an HDL code, providing a list of electronic components that represent the circuit and also giving details of how they are connected.

3.4.2 VHSIC Hardware Description Language (VHDL)

The Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) was created by the US Department of Defense in the early 1980s. In December of 1987, VHDL was adopted as an IEEE Standard [272]. VHDL is a functional language that borrows much of its structure from the programming language Ada along with a set of constructs for supporting the inherent parallelism of hardware designs.

The original version of VHDL, included a wide range of data types such as, logical (bit and boolean), numerical, character and time, plus bit and character. In later versions, the std_logic data type was introduced, along with signed and unsigned types to facilitate arithmetical operations, analog and mixed-signal circuit design extensions [367].

Furthermore, the designer can know how his/her HDL instruction was mapped to FPGA components (such as slices, flip-flops, tri-state buffers, etc.). For example, an *if statement* in HDL describes a multiplexer or a flip-flop. It can occur that the frequent use of this statement would insert large number of multiplexers or flip-flops in a circuit, which is functionally correct but may or may not be efficient. As a matter of fact, HDL languages have been designed favoring a hardware designer perspective, in the sense that first the specific hardware architecture should be envisioned, and then an HDL piece of code representing it should be written. If for instance a programmer requires a

flip-flop functionality then he/she should select a suitable flip flop for the design and then he/she can write a code for it. That would generate a list of components for an electronic circuit prior to its implementation providing a designer complete control over available/used FPGA resources.

3.4.3 Other Programming Models for FPGAs

Several voices, both from the Academia and Industry sectors, have stated that the main obstacle towards a massive use of reconfigurable computing lies in the difficulty of programming FPGA devices. After all, HDLs were designed primarily from the perspective of designers trying to describe hardware structures, which quite often implies that an FPGA programmer should be primarily a hardware designer.

Considering that, it has been proposed as an alternative to HDLs as design entry tool to combine high level languages (such as C or C++) with concurrency primitives, thus allowing even faster design cycles for FPGAs than what is now possible using traditional HDLs [119, 189, 39, 229].

Table 3.6 shows some of the commercial software tools currently available in the market.

Table 3.6. High Level FPGA Programming Software

Vendor	Product	Base Language
Celoxica	Agility Compiler	Handel-C
Mentor Graphics	Catapult C	C
Impulse Accelerated Tech.	Impulse C	C
Annapolis Microsystems	Core Fire Design Suite	GUI Design Entry
Open System C Initiative (OSCI)	SystemC	C++, IEEE standard 1666

In other order of ideas, designing a complex system in FPGAs can be greatly alleviated by using existing pre-designed libraries. Those libraries, frequently called IP (Intellectual Property) cores, have been fully tested and optimized for performing commonly used building blocks, such as large multiplexers, counters, divisors, digital filters and so forth.

3.5 Implementation Aspects for Reconfigurable Hardware Designs

3.5.1 Design Flow

In general, most FPGA design tools consist of six basic steps [390] as shown in Fig. 3.9. Those steps must not be executed in a specific order but they can

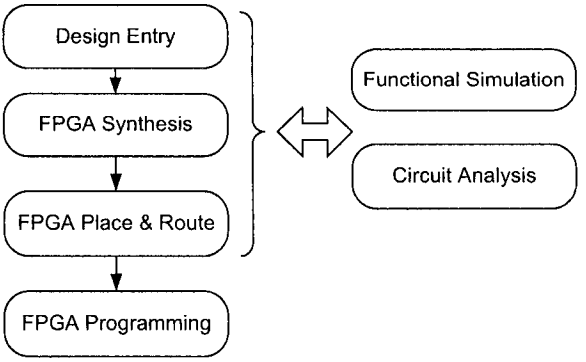

Fig. 3.9. Design flow

be repeated to improve design's performance. A short description of each step is provided below.

1. **Design Entry** : There are two standard ways to specify an FPGA design, namely,
   - Design Entry through HDLs (Hardware Description Languages): A designer can describe an FPGA design in high-level abstract language like VHDL (Very high speed integrated circuit Hardware Description Language) or Verilog. Those languages are ideal to build state machines, combinational logic, complex and large designs. Most software tools have sophisticated compilers that can efficiently translate HDL specifications to FPGA hardware resources.
   - Design Entry through Schematic: An FPGA design can also be described by using library components of the devices through a graphical interface. It is easy to optimize a circuit for speed/area and consequently it saves time and efforts of the design tool in hardware mapping, placement and routing, etc. However, it is hard to debug and modifications to the design are not straightforward as compared to design entry through HDLs.
2. **Functional verification and simulation:** In this step, the logical correctness of an FPGA design is validated. Once that the design has been specified, either by using HDLs or schematic design entry, it is necessary to verify if such description meets the design specifications.
3. **FPGA synthesis:** Synthesis converts a design entry specification into gates/blocks of an FPGA device. A netlist of basic gates is prepared from HDL/schematic design entry, which is further optimized at gate level. The next step is to map that netlist into FPGA real resources. This is an important step based on design entry. When writing HDL code or using

schematic device's libraries, an FPGA designer should always take into account the basic structure of the target device.

4. **FPGA place and route:** Place and route selects the optimal physical positioning of elementary design blocks and minimal interconnection distance among them. Place and route tools normally use device vendor specifications. Usually they provide hand-placement and also automatic features for optimizing critical paths either for speed or for area.
5. **Circuit analysis:** Circuit analysis evaluates different design performance metrics. Timing verification is made which may differ from functional simulation as it provides logical correctness taking into account all circuit delays occurring in the real device. Similarly, a power analysis evaluation provides an estimation of the design power consumption.
6. **Programming FPGA device:** Programming FPGA implies downloading bit stream codes from the last design steps onto the target FPGA device. Universal programming tools work with FPGAs from different vendors. However there are dedicated programming tools bounded only with a single family of FPGA devices.

3.5.2 Design Techniques

It has been observed that better design techniques for both design entry and design implementation play a crucial role for optimizing circuit's performance. A short description of some of those optimizing techniques is given below.

Design Strategy

Design strategy is application dependent. For some time critical applications, timing performance is the most important requirement regardless other factors such as hardware resources or device cost. On the contrary, other applications may require a design architecture as compact as possible or with a certain functionality.

Block cipher cryptographic algorithms have an iterative nature, where n iterations (or rounds) having the same functionality must be executed. It is therefore possible to implement either just one round and consume n cycles (iterative looping), or n rounds of the algorithm (using a pipeline structure) in order to achieve high timing performances. The designer choice will be made depending on design's minimum requirements in terms of speed and area.

Fig. 3.10 shows a basic methodology usually followed when implementing an FPGA design.

Choice of Target Device

Choosing the target device (FPGA) depends on the design strategy. As it is shown in Table 3.1, an ample spectrum of FPGA devices are available in

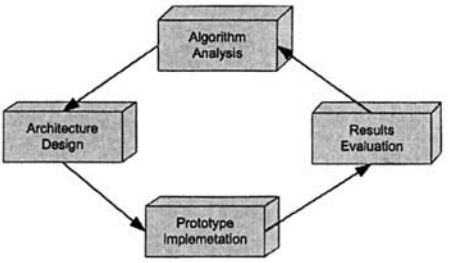

Fig. 3.10. Hardware Design Methodology

the market from various manufacturers. The basic structure of all FPGAs is similar, however some models offer additional features like built-in-memories, built-in-arithmetic functions, etc. As it is shown in Table 3.2 for Xilinx devices, different functionality and sizes are available depending on the device's cost.

For example, in the case of block cipher designs it may be useful to select an FPGA device that has embedded Block RAMs (BRAMs) on it. As it was explained above, BRAMs are fast access memories and might be excellent choices for a straightforward implementation of the characteristic S-box blocks of symmetric ciphers. Alternatively, S-Boxes can be implemented using the FPGA CLB fabric configured in memory mode.

In short, the selection of an FPGA depends upon the design size and design requirements.

Design Analysis

Design/algorithm analysis helps reducing the design's size and critical path delays. It might not be a good idea to directly implement a fast software code in hardware. Software codes are often optimized for high granularity processors, for example, 8, 16 or 32 bit general-purpose microprocessors. Due to its inherent low granularity, hardware implementations quite often can benefit from a bit-level parallelism only limited by data dependencies or resource limitations. For instance, let us consider an instruction from a software code optimized for a 32-bit word-size general-purpose microprocessor:

$$\text{work} = [((\text{left} \gg 16) \mid \text{right}) \;\&\; 0\times\; 0000\text{FFFF}];$$

That requires 16 right shifts, one logical XOR and then one logical AND with 0×0000FFFF. In software platforms, we have no option but to execute an XOR operation for the 16 most significant bits of 32-bit 'left' and 'right' registers.

On the contrary, in hardware description languages, the same instruction can be implemented almost for free, just caring for language notations. One

of the best options is to eliminate the AND operation and 16 logical Shifts by executing instead an XOR operation directly applied to the 16 most significant bits of left and right registers, that is,

$$\text{work} = \text{left}[31{:}16] \oplus \text{right}[31{:}16]$$

Selecting FPGA Resources

An FPGA designer can pick multiple options for performing a function. For example, two choices for implementing a 2-bit multiplexer are shown in Figure 3.11.

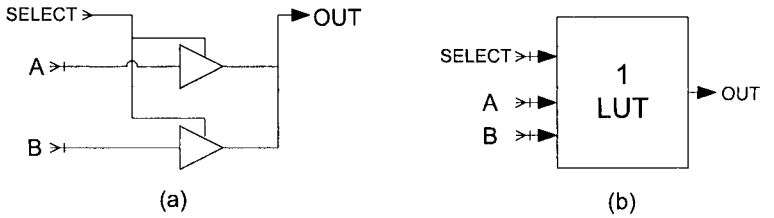

Fig. 3.11. 2-bit Multiplixer Using (a) Tristate Buffer. (b) LUT

Figure 3.11.a shows usage of tri-state buffers for a multiplexer. A large number of tri-state buffers are available in FPGAs and it seems logical to make use of them. However, experience shows that, using large number of tri-state buffers slows down the circuit. This tends to require the physical distribution of tri-state buffers all around FPGA, which requires long routing paths. A multiplexer can also be implemented using LUTs as shown in Figure 3.11.b. Using adjacent LUTs for an n to 1 multiplexer would be useful when a circuit must be optimized for speed.

Similarly, some FPGA devices contain built-in memory modules. It would be useful to utilize those memories as they provide faster access to the data as compared to distributed memories in FPGAs which are formed using several LUTs.

Hardware Approach

A careful selection and usage of the design tools results useful in our methodology for obtaining better performances. The design tools by Xilinx [390], Altera [3], Synopsis Galaxy Design Platform [351], LeonardoSpectrum and ModelSim by Mentor Graphics [231, 230], etc. provide several useful features for getting design improvements. Better placement of the components or better routing of the architecture modules can be helpful in cutting critical path delays in the circuit.

3.5.3 Strategies for Exploiting FPGA Parallelism

Achieving high-speed implementations for cryptographic algorithms is an exciting task requiring deep considerations at every stage of the design. Design strategies should therefore not only be based on the best implementing techniques on reconfigurable platforms but also on trying to innovate in the theoretical side by improving the standard transformations of cryptographic algorithms. In this sense, the designs included in this book try to take as much advantage as possible of the hardware inherent parallelism while keeping as low as possible the hardware resource requirements. In the following we discuss various strategies used by designers to implement cryptographic algorithms.

Iterative Looping (IL)

An iterative looping design (IL), implements only one round and n iterations of the algorithm are carried out by feeding back previous round results as shown in Figure 3.12a. It utilizes less area but consumes more clock cycles resulting on a relatively low speed encryption.

Loop Unrolling

Architecture with loop unrolling is shown in Figure 3.12b. In a loop unrolling or pipeline design (PP), rounds are replicated and registers are provided between the rounds to control the flow of data. The design offers high speed but area requirements tend to be too high.

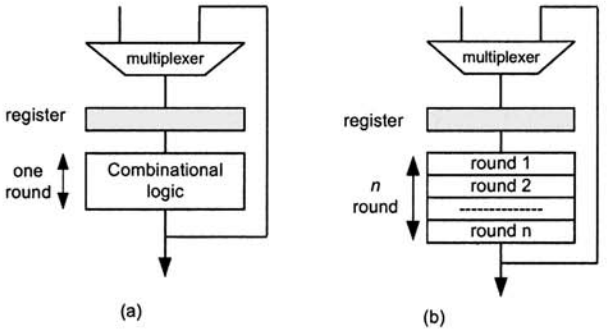

Fig. 3.12. Basic Architectures for (a) Iterative Looping (b) Loop Unrolling

Inner-Round Pipelining

Figure 3.13a shows an inner-round pipelining architecture where extra registers are provided at different stages of the same round in such a way that several blocks of data can be processed by the circuit at the same time. This approach produces high speed circuits at the cost of more hardware resources in the form of registers.

Outer-Round Pipelining

Outer-round pipelining is created through loop unrolling by adding extra registers at different stages of the same round as shown in Figure 3.13b. This approach directly trades circuit speed with circuit area.

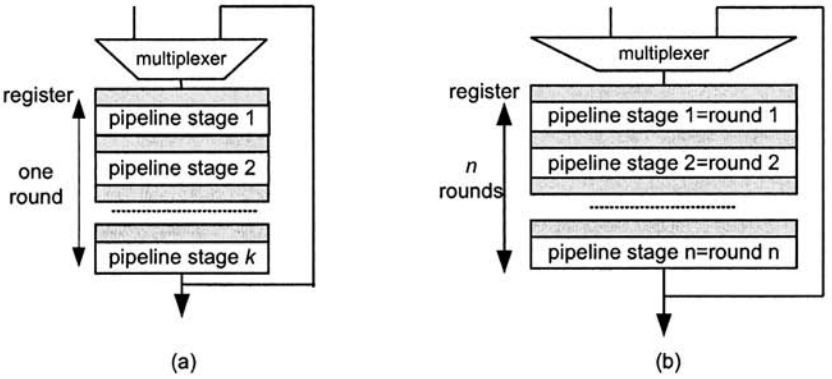

Fig. 3.13. Round-pipelining for (a) One Round (b) n Rounds

Both the iterative and pipeline architectures would be optimized for the implementation of secret-key ciphers. Public key algorithms exhibit different nature. They do not have rounds however they maintain a hierarchical structure that can be further exploited.

3.6 FPGA Architecture Statistics

Just as it occurs with software platform comparisons, comparing FPGA designs is a difficult and a bit ambiguous task. The two single most important performance metrics usually considered are the *time complexity*, sometime called design throughput and the *area complexity*.

For combinatorial designs (such as adders, squarers, fully-parallel multipliers, etc), time complexity is determined from the Maximum Clock Frequency

(MCF), which in turns is proportional to the maximum combinational path delay. In the case of sequential designs (such as block ciphers, sequential multipliers, etc.), time complexity must also consider the total number of clock cycles required before the result is ready. In the case of block cipher designs, it is customary to consider also how many bits are processed at the same time. In this work we define the throughput of a given design as follows,

Throughput

Throughput is an important factor to measure timing performances of the design [82, 103, 382]. Throughput of the design is obtained by multiplying the allowed frequency for the design with the number of bits processed per cycle. For cryptographic algorithms, throughput is defined as:

$$\text{Throughput} = \frac{\text{Allowed Frequency} \times \text{Number of Bits}}{\text{Number of cycles}} \text{ (bits/s)}$$

The higher the throughput of a design is the better its efficiency.

Area

Design statistics provided by the design software expresses hardware area occupied by the design. Unfortunately, there is no universal metric to measure the hardware costs associated with an FPGA based design. After mapping a design to a particular FPGA device, FPGA compiler provides FPGA resources utilized by that design.

Following are some common FPGA resources listed by the mapping tool:

- Number of slices
- Number of Slice Flip Flops
- Number of 4-input Look Up Tables (LUTs)
- Number of Input/Output Blocks
- Number of Clocks
- Maximum combinational path delay
- Maximum output required time after clock
- Maximum Clock Frequency (MCF)
- BlockSelect RAMs (BRAMs)

A designer, however, can report hardware area in terms of LUTs as well as CLB slices. An ideal comparison would be therefore comparing all resources on the similar FPGA device. A design using dedicated resources of the device will show less logic resources as compared to other design which implements the whole logic without using any dedicated unit of the device. It also affects the throughput statistic. It has been experimentally observed that the implementation of even the same code on different grades of the same family of devices influence the final design's throughput. That situation becomes more

crucial when the same design targets two different devices by two different manufactures. In such cases, for the purpose of classifying an FPGA design, we can ignore some of those factors.

It can be said, as a first approximation, that the fastest design is the one which achieves fastest speed no matter what type of device has been targeted for design implementation. However, when considering a compact design (a design optimized for hardware area), this criterion cannot be applied. The comparison of two compact designs can be only justified if it is made between similar devices.

Both area and throughput factors provide a measure for comparing different designs. Additionally, in order to decide how efficient a design is, we utilize the following figure of merit.

Throughput/Area

It is the ratio of the above two figures of merits and shows how efficient the design is with respect to both area and throughput. The ratio is higher in case of high throughput and less space.

3.7 Security in Reconfigurable Hardware Devices

The selection of an implementation platform in a digital system depends on many design criteria. Besides the design performance figures such as, system speed and area costs, there exist other performance and security factors that should be taken into account such as: physical security (for instance, against key recovery and algorithm manipulation), flexibility, power consumption and other secondary factors, that may as well affect the design selections.

Even though there exist a fair amount of papers reporting cryptographic implementations on FPGA devices, there are not that many papers reporting the convenience (or not) of utilizing FPGA as a target device for security applications from a system point of view. In particular, few works report the resilience of FPGA against physical or system attacks, which are potentially more dangerous than algorithm attacks [379, 342, 343].

In [380, 379] a comprehensive analysis of FPGA security aspects is given. Authors conclude that FPGA technology can provide a reasonable level of security when used properly.

The fourth generation design security of Xilinx Virtex-4 family is equipped with bit-stream encryption/decryption technology based on 256-bit AES. The user generates the encryption key and encrypted bit-stream using Xilinx ISE software. In a second step, during configuration, the Virtex-4 device decrypts the incoming bit-stream using a decryption logic module with dedicated memory for storing the 256-bit encryption key [393].

For the cryptographic applications, the most important threat is unauthorized access to a confidential cryptographic key, either a symmetric key or the private key of an asymmetric algorithm[4].

FPGA implementations are also vulnerable to side-channel attacks. A side channel attack is based on information gained directly from the physical implementation. Examples for side channels include: power consumption, timing behavior, and electromagnetic radiation. Most relevant papers on side-channel attacks and related defenses have been published in [183, 184, 182, 159, 366, 157, 278].

Power analysis attacks were introduced in 1998 by Kocher et al. [186]. The main idea behind this attack is to measure the power consumption of the FPGA device during the execution of a cryptographic operation. Thereafter, that power consumption can be analyzed in an effort for finding regions in the power consumption trace of a device that are correlated with algorithm's secret key.

In [262], the first experimental results of power analysis attack on an FPGA implementation of elliptic curve cryptosystem were presented. RSA, AES and DES FPGA implementations have also been subjects of attacks in [341, 342, 343].

3.8 Conclusions

In this chapter we presented some of the most relevant aspects related to FPGA devices considering both, technological and reconfigurable programming aspects.

The material covered in this Chapter includes a brief review of the technological antecedents that gave birth to FPGA devices. We also studied the structure of several emblematic FPGA families from the two market leaders, Xilinx and Altera. We compare the performance of FPGA realizations against the ones on ASICs and general-purpose processor platforms and we briefly introduced the main concepts related to the reconfigurable computing paradigm.

Furthermore, we reviewed several key strategies to achieve good designs when working with cryptographic applications. As a way to measure area and time performances for a given design, we defined several metrics and figures of merit. Finally, several security concerns related to FPGA technology were outlined.

[4] As it was described in the precedent chapter, most cryptographic algorithms have been standardized and therefore, they are publicly known.

4

Mathematical Background

The material presented in this Chapter, discusses several relevant mathematical concepts, fundamental for the understanding of elliptic curve public-key cryptosystems, the RSA algorithm, etc.. This material is also useful for a better understanding of the basic operations involved in the specifications of Rijndael algorithm (new Advanced Encryption Standard (AES)).

For a more detailed treatment of these aspects, the reader is referred to Number theory books like [376, 220, 47, 297], and to excellent cryptography books such as [226, 176, 129, 227, 106, 107]. The material presented in this chapter was written based on [56, 42, 289].

The rest of this Chapter is organized as follows. In Section 4.1 we give several basic definitions and theorems of the elementary theory of numbers. Then, in Section 4.2 we explain the concept of finite field, defining the associated arithmetic operations. Elliptic curves defined over $\mathbb{R}$ are described in Section 4.3. Thereafter, in Section 4.4, elliptic curves defined over binary extension fields are discussed in more detail. Several coordinate systems for representing elliptic curve points are presented in Section 4.5. Then different schemes for scalar representation are discussed in Section 4.6. Concluding remarks are given in Section 4.7.

4.1 Basic Concepts of the Elementary Theory of Numbers

Elementary theory of numbers is perhaps the single most important tool for developing cryptographic algorithms. Therefore, we start this chapter given some important definitions, theorems and results relevant to the subject of cryptography.

4.1.1 Basic Notions

Definition 4.1 (Integer Numbers). *Integer numbers are defined as the set of numbers* $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, 3, ...\}$. *Within this set we have the subset of the natural numbers,* $\mathbb{N} = \{1, 2, 3, 4, ...\}$, *i.e., the subset of all positive numbers (greater than zero)*

Definition 4.2 (Divisibility). *Let* a *and* b *be two integers with* $a \neq 0$. *We say that* a *divides* b, *that* a *is a divisor or factor of* b, *that* b *is a multiple of* a *or that* b *is divisible by* a, *if there exists an integer* k *such that* $b = ak$. *This is written as* $a|b$. *If* a *does not divide* b *we write it as* $a \nmid b$.

Let a, b, $c \in \mathbb{Z}$, some important divisibility properties are,

i. For all $a \neq 0$, $a|a$. At the same time $1|b$ for all b,
ii. If $a|b$ then $a|bc$,
iii. If $a|b$ and $b|$ c then $a|c$,
iv. If $a|b$ and $a|c$ then $a|(b \pm c)$,
v. If $a|b$ and $a \nmid c$ then $a \nmid (b \pm c)$,
vi. If $a|b$ and $a|c$ then $a|(sb + tc)$ for any arbitrary integers s and t.

Theorem 4.3 (Integer division theorem). *Let* $a \in \mathbb{Z}$ *and* $b \in N$. *Then there exist* q, $r \in \mathbb{Z}$ *with* $0 \leq r < b$ *such that* $a = mq + r$. *Additionally,* q *and* r *are unique.*

Definition 4.4 (Greatest common divisor). *Given two integers* a *and* b *different than* 0, *we say that the integer* $d > 1$ *is the greatest common divisor, or gcd, of* a *and* b *if* $d|a$, $d|b$ *and for any other integer* c *such that* $c|a$ *and* $c|b$ *then* $c|d$. *In other words,* d *is the greatest positive number that divides both,* a *and* b.

Some of the properties of the greatest common divisor are,

- $gcd(a,b) = gcd(|a|,|b|)$
- $gcd(ka,kb) = k$ gcd(a,b)
- $gcd(a,b) = d \iff d|a, d|b$ and $gcd(a/d,b/d)=1$

It is possible to compute the greatest common divisor by means of the Euclidian algorithm shown in Algorithm 4.1.

Definition 4.5 (Prime numbers). *We say that a positive integer* $p > 1$ *is a prime number if its only positive divisors are* 1 *and* p.

Definition 4.6 (Relative Primes). *We say that two integers* a *and* b *are relatively primes if* $gcd(a,b)=1$.

Definition 4.7 (Composite Numbers). *If an integer number* $q > 1$ *is not a prime, then it is a composite number. Therefore, an integer* q *is a composite number if and only if there exist* a,b *positive integers (less than* q*) such that* $q = ab$.

Algorithm 4.1 Euclidean Algorithm (Computes the Greatest Common Divisor)

Require: two positive integers a and b where $a \geq b$.
Ensure: the greatest common divisor of a and b, namely $d = gcd(a, b)$.
1: **while** $b \neq 0$ **do**
2: $r \leftarrow a \bmod b$;
3: $a \leftarrow b$;
4: $b \leftarrow r$;
5: **end while**
6: **Return** a

Theorem 4.8 (Fundamental Theorem of Arithmetic). *Any natural number $n > 1$ is either a prime number, or it can be factored as a product of powers of prime numbers p_i,*

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

with $e_i \in \mathbb{N}$, $\forall\ i\ \in\ [\![1, r]\!]$. Furthermore, except for the order of the factors, this factorization is unique.

Corollary 4.9. *If $n \in \mathbb{N}$, then the number of positive divisors of n is $(e_1 + 1)(e_2 + 1) \cdots (e_r + 1)$.*

Corollary 4.10. *If p is a prime number, $a,\ b \in \mathbb{Z}$ and $p|ab$ then $p|a$ or $p|b$.*

Notice that above result is not necessarily true if p is a composite number. For example, $10|5 \cdot 4$ but $10 \not| 5$ and $10 \not| 4$.

Let $a,\ b \in \mathbb{N} \subset \mathbb{Z}$ and $a = \prod_{i=1}^{n} p_i^{e_i}$, and $b = \prod_{j=1}^{m} q_j^{f_j}$, be their prime factorization with $1 \leq i \leq n,\ 1 \leq j \leq m$. Let $R_1, R_2, \ldots, R_s$ be the distinct prime numbers that are included in both factorizations. Rewriting a and b as $a = \prod_{i=1}^{s} R_i^{t_i},\ b = \prod_{i=1}^{s} R_i^{u_i}$ with $t_i,\ u_i \geq 0$ for $1 \leq i \leq s$, we have,

$$gcd(a, b) = \prod_{i=1}^{s} R_i^{\min\{t_i, u_i\}}$$

Example 4.11.

$$2520 = 2^3 \cdot 3^2 \cdot 5^1 \cdot 7^1$$
$$2700 = 2^2 \cdot 3^3 \cdot 5^2 \cdot 7^0$$

then $gcd(2520, 2700) = 2^2 \cdot 3^2 \cdot 5^1 = 180$.

Definition 4.12. *Let $n \in \mathbb{N}$. We define the Euler function $\phi(n)$, as the number of relatively prime numbers that n has in the interval $[1, n)$.*

In other words, $\phi(n) = |\{m \in \mathbb{N} : gcd(m, n) = 1 \text{ and } 1 \leq m < n\}|$. Let p be a prime number and $m,\ n,\ r \in \mathbb{N}$ with $r > 1$, then

i. $\phi(p^r) = p^r\left(1 - \frac{1}{p}\right) = p^{r-1}(p-1)$, In particular $\phi(p) = p - 1$,
ii. $\phi(mn) = \phi(m)\phi(n)$, if $gcd(m, n) = 1$.

Therefore, we may compute the Euler function ϕ for a given number n by obtaining first the integer factorization of n.

Example 4.13.

$$\phi(720) = \phi(2^4)\phi(3^2)\phi(5) = 2^3 \cdot (2-1) \cdot 3^1 \cdot (3-1) \cdot (5-1) = 192.$$

Theorem 4.14 (Fermat's Little Theorem). *If $(a, p) = 1$, then*

$$a^{p-1} \equiv 1 \bmod p, \quad (a^p \equiv b \bmod p)$$

equivalently,

$$a^{\phi(p)} \equiv 1 \bmod p.$$

Corollary 4.15. *If $x \equiv y \mod (p-1)$, then $a^x \equiv a^y \bmod p$.*

Theorem 4.16 (Euler Theorem). *If $a \in \mathbb{Z}$ and gcd(m,a)=1 then*

$$a^{\phi(m)} \equiv 1 \bmod m.$$

Corollary 4.17. *If $x \equiv y \mod \phi(m)$, then $a^x \equiv a^y \bmod m$.*

Definition 4.18 (Order of a number x). *If x and m are relatively primes, we say that the order of x modulo m is the smallest integer r such that*

$$a^r = 1 \bmod m.$$

Definition 4.19 (Primitive Root). *Let m be a prime number and $g \in \mathbb{Z}_m$, then we say that g is a primitive root of m, if and only if the order of g modulo m is equal to the value of the Euler function $\phi(m)$. According to Euler's theorem, there is always a primitive root since, $g^{\phi(m)} = 1 \bmod m$.*

Let g be a primitive root of a prime number p, then the following properties hold,

i. If n is an integer, then $g^n \equiv 1 \bmod p$ if and only if $n \equiv 0 \bmod p - 1$.
ii. If j and k are two integers, then $g^j \equiv g^k \bmod p$ if and only if $j \equiv k \bmod p - 1$.
iii. If a is a primitive root, then a^x is also a primitive root if and only if $gcd(x, p-1) = 1$.

iv. If $g^n \equiv 1 \bmod p$ then $n|(p-1)$.

If $p = 1223$, $p-1 = 2 \cdot 13 \cdot 47$, if a is not a primitive root, then either a^{26} or a^{94} or a^{611} must be congruent 1 modulo 1223. $a = 2, 3$ are not primitive roots, since $2^{611} \equiv 3^{94} \equiv 1 \bmod 1223$. However, $a = 5$ is a primitive root since,

$$a^{26},\ a^{94}, a^{611} \neq 1 \bmod\ 1223.$$

Furthermore, using above properties we can see that $5^2 = 25$ is not a primitive root since $gcd(2, p-1) \neq 1$. On the other hand, the element $5^3 = 125$ is a primitive root given that $gcd(3, p-1) = 1$.

4.1.2 Modular Arithmetic

Definition 4.20 (Congruency). *Given $m \in \mathbb{Z}$, $m > 1$, we say that $a, b \in \mathbb{Z}$ are congruent modulo m if and only if $m|(a-b)$. We write this relation as $a \equiv b \bmod m$. Where m is the modulus of the congruency. Notice that if m divides $a-b$, this implies that both, a and b have the same residue when divided by m.*

We define $\mathbb{Z}_m$ as the set of all positive residues modulo m, which is composed by the set, $\mathbb{Z}_m = \{0, 1, 2, ..., m-1\}$. Invoking the integer division theorem it is easy to see that for every integer a there exists a residue r that belongs to $\mathbb{Z}_m$.

If $m \in \mathbb{N}$ and $a, b, c, d \in \mathbb{Z}$ such that $a \equiv b \bmod m$ and $c \equiv d \bmod m$, then the following properties hold,

- $a + c \equiv b + d \bmod m$
- $a - c \equiv b - d \bmod m$
- $a \cdot c \equiv b \cdot d \bmod m$

The relationship of congruency modulus m is a relationship of equivalence for all $m \in \mathbb{Z}$. Let $a, b, c \in \mathbb{Z}$, then the congruence relation satisfies the following properties,

1. Reflexive: $a \equiv a \bmod m$.
2. Symmetric: If $a \equiv b \bmod m$ then $b \equiv a \bmod m$.
3. Transitivity: If $a \equiv b \bmod m$ and $b \equiv c \bmod m$ then $a \equiv c \bmod m$.

Modular Addition and subtraction If $a, b \in \mathbb{Z}_m$ then we define the modular addition operator $a + b \bmod m$ as an element within $\mathbb{Z}_m$. For example, $17 + 20 \bmod 22 = 15$. The most important properties of the modular addition are,

1. It is commutative, $a + b \bmod m = b + a \bmod m$.
2. It is associative, $(a + b) + c \bmod m = a + (b + c) \bmod m$.
3. It has a neutral element (0), such that $a + 0 = a \bmod m$.

4. For every a and b in $\mathbb{Z}_m$ there exists a unique element x in $\mathbb{Z}_m$ such that $a + x = b \bmod m$.

Using last property and $b = 0$, it can be seen that for every a in $\mathbb{Z}_m$ there exists a unique element x in $\mathbb{Z}_m$ such that $a + x \equiv 0 \bmod m$.

Modular multiplication If $a, b \in \mathbb{Z}_m$ then we define modular multiplication as, $c = a \cdot b \bmod m$, where c is an element in $\mathbb{Z}_m$. The most important properties of modular multiplication are,

1. It is conmutative $a \cdot b \bmod m = b \cdot a \bmod m$.
2. It is associative $(a \cdot b) \cdot c \bmod m = a \cdot (b \cdot c) \bmod m$.
3. It has a neutral element (1), such that $a \cdot 1 = a \bmod m$
4. If $gcd(m, c)$=1 and $a \cdot c \equiv b \cdot c \bmod m$, then $a \equiv b \bmod m$. If m is a prime number, this property always hold.

Using last property, we define the *multiplicative inverse* of a number a as follows,

Definition 4.21 (Multiplicative Inverse). *We say that an integer a has an inverse modulo m if there exists an integer b such that $1 \equiv ab \bmod m$. Then, the integer b is the inverse of a and it is written as a^{-1}. The inverse of a number $a \bmod m$ exists if and only if there exist two integer numbers x, y such that $ax + my = 1$ and these numbers exist if and only if gcd(a,m)=1.*

In order to obtain the modular inverse of a number a we may use the extended Euclidean algorithm [178], with which it is possible to find the two integer numbers x, y that satisfy the equation[1],

$$ax + my = 1.$$

Modular Division Using above definition we say that if $a, b \in \mathbb{Z}_p$ and p is a prime number, we can accomplish the division of a by b by computing $a \cdot b^{-1} \bmod m$, where b^{-1} is the multiplicative inverse of b modulo p.

For example, we can compute $\frac{17}{20} \bmod 23$, by performing $17 \cdot (20)^{-1} \bmod 23$, where $(20)^{-1} \bmod 23 = 15$. Thus,

$$\frac{17}{20} \bmod 23 = 17 \cdot 15 \bmod 23 = 2.$$

Modular Exponentiation We define modular exponentiation, as the problem of computing the number $b = a^e \bmod m$, with $a, b \in \mathbb{Z}_m$, and $e \in \mathbb{N}$. From the observation that,

$$x \cdot y \bmod m = [(x \bmod m) \cdot y \bmod m] \bmod m,$$

[1] In §6.3 we present an efficient implementation of a variation of this algorithm: the Binary Euclidean Algorithm (BEA).

Algorithm 4.2 Extended Euclidean Algorithm as Reported in [228]

Require: Two positive integers a and b where $a \geq b$.
Ensure: d =gcd(a, b) and the two integers x,y that satisfy the equation $ax + by = d$.

```
1: if b = 0 then
2:    d = a;, x = 1;, y = 0;
3:    Return (d, x, y)
4: end if
5: x1 = 0;, x2 = 1;, y1 = 1;, y2 = 0;
6: while b > 0 do
7:    q = a div b; r = a mod b;
8:    x = x2 − qx1; y = y2 − qy1;
9:    a = b; b = r; x2 = x1;
10:   x1 = x; y2 = y1; y1 = y;
11: end while
12: d = a, x = x2, y = y2;
13: Return (d, x, y)
```

it can be seen that the exponentiation problem, can be solved by multiplying numbers that never exceed the modulus m.

Rather than computing the exponentiation by performing $e - 1$ modular multiplications as,

$$b = \overbrace{a \cdot a \ldots a}^{e-1 mults.} \pmod{m},$$

we employ a much more efficient method that has complexity $O(log(e))$. For example if we want to compute 12^{26}(mod23), we can proceed as follows,

$$\begin{aligned} 12^2 &= 144 = 6 \bmod 23; \\ 12^4 &= 62 = 36 = 13 \bmod 23; \\ 12^8 &= 132 = 169 = 8 \bmod 23; \\ 12^{16} &= 82 = 64 = 18 \bmod 23. \end{aligned}$$

Then,

$$12^{26} = 12^{(16+8+2)} = 12^{16} \cdot 12^8 \cdot 12^2 = 18 \cdot 8 \cdot 6 = 864 = 13 \bmod 23.$$

This algorithm is known as the binary exponentiation algorithm [178], whose details will be discussed in §5.4.

Chinese Remainder Theorem(CRT) This theorem has a tremendous importance in cryptography. It can be defined as follows,

Let p_i for $i = 1, 2, \ldots, k$ be pairwise relatively prime integers, i.e.,

$$gcd(p_i, p_j) = 1 \text{ for } i \neq j.$$

Given $u_i \in [0, p_i - 1]$ for $i = 1, 2, \ldots, k$, the Chinese remainder theorem states that there exists a unique integer u in the range $[0, P-1]$ where $P = p_1 p_2 \cdots p_k$ such that

$$u = u_i \pmod{p_i}.$$

4.2 Finite Fields

We start with some basic definitions and then arithmetic operations for the finite fields are explained.

4.2.1 Rings

A ring $\mathbb{R}$ is a set whose objects can be added and multiplied, satisfying the following conditions:

- Under addition, $\mathbb{R}$ is an additive (Abelian) group.
- For all $x; y; z \in \mathbb{R}$ we have, $x(y + z) = xy + xz$; $(y + z)x = yx + zx$:
- For all $x; y \in \mathbb{R}$, we have $(xy)z = x(yz)$.
- There exists an element $e \in \mathbb{R}$ such that $ex = xe = x$ for all $x \in \mathbb{R}$.

The integer numbers, the rational numbers, the real numbers and the complex numbers are all rings. An element x of a ring is said to be invertible if x has a multiplicative inverse in $\mathbb{R}$, that is, if there is a unique $u \in \mathbb{R}$ such that: $xu = ux = 1$. 1 is called the *unit element* of the ring.

4.2.2 Fields

A Field is a ring in which the multiplication is commutative and every element except 0 has a multiplicative inverse. We can define a Field $\mathbb{F}$ with respect to the addition and the multiplication if:

- $\mathbb{F}$ is a commutative group with respect to the addition.
- $\mathbb{F} \setminus \{0\}$ is a commutative group with respect to the multiplication.
- The distributive laws mentioned for rings hold.

4.2.3 Finite Fields

A *finite field* or *Galois field* denoted by $\mathrm{GF}(q = p^m)$, is a field with characteristic p, and a number q of elements. Such a finite field exists for every prime p and positive integer m, and contains a subfield having p elements. This subfield is called *ground field* of the original field. For every non-zero element $\alpha \in \mathrm{GF}(q)$, the identity $\alpha^{q-1} = 1$ holds.

In cryptography the two most studied cases are: $q = p$, with p a prime and $q = 2^m$. The former case, $\mathrm{GF}(p)$, is denoted as *prime field*, whereas the latter, $\mathrm{GF}(2^m)$, is known as finite field of characteristic two or simply *binary extension field*. A binary extension field is also denoted as $\mathbb{F}_{2^m}$.

4.2.4 Binary Finite Fields

A polynomial p in $GF(q)$ is *irreducible* if p is not a unit element and if $p = fg$ then f or g must be a unit, that is, a constant polynomial.

Let $P(x)$ be an irreducible polynomial over $GF(2)$ of degree m, and let α be a root of $P(x)$, i.e., $P(\alpha) = 0$. Then, we can use $P(x)$ to construct a binary finite field $F = GF(2^m)$ with exactly $q = 2^m$ elements, where α itself is one of those elements. Furthermore, the set

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

forms a basis for F, and is called the polynomial (canonical) basis of the field [221]. Any arbitrary element $A \in GF(2^m)$ can be expressed in this basis as,

$$A = \sum_{i=0}^{m-1} a_i \alpha^i.$$

Notice that all the elements in F can be represented as $(m-1)$-degree polynomials.

The order of an element $\gamma \in F$ is defined as the smallest positive integer k such that $\gamma^k = 1$. Any finite field contains always at least one element, called a *primitive element*, which has order $q-1$. We say that $P(x)$ is a primitive polynomial if any of its roots is a primitive element in F. If $P(x)$ is primitive, then all the q elements of F can be expressed as the union of the zero element and the set of the first $q-1$ powers of α [221, 379]

$$\left\{0, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{q-1} = 1\right\}. \tag{4.1}$$

Some special classes of irreducible polynomials are more convenient for the implementation of efficient binary finite field arithmetic. Some important examples are: trinomials, pentanomials, and equally-spaced polynomials. Trinomials are polynomials with three non-zero coefficients of the form,

$$P(x) = x^k + x^n + 1 \tag{4.2}$$

Whereas pentanomials have five non-zero coefficients:

$$P(x) = x^k + x^{n_2} + x^{n_1} + x^{n_0} + 1 \tag{4.3}$$

Finally, irreducible equally-spaced polynomials have the same space separation between two consecutive non-zero coefficients. They can be defined as

$$P(x) = x^m + x^{(k-1)d} + \cdots + x^{2d} + x^d + 1\ , \tag{4.4}$$

where $m = kd$. The ESP specializes to the all-one-polynomials (AOPs) when $d = 1$, i.e., $P(x) = x^m + x^{m-1} + \cdots + x + 1$, and to the equally-spaced trinomials when $d = \frac{m}{2}$, i.e., $P(x) = x^m + x^{\frac{m}{2}} + 1$.

In this Book we are mostly interested in a *polynomial basis* representation of the elements of the binary finite fields. We represent each element as a binary string $(a_{m-1} \ldots a_2 a_1 a_0)$, which is equivalently considered a polynomial of degree less than m,

$$a_{m-1}x^{m-1} + \ldots + a_2 x^2 + a_1 x + a_0. \tag{4.5}$$

The addition of two elements $a, b \in F$ is simply the addition of two polynomials, where the coefficients are added in $GF(2)$, or equivalently, the bitwise XOR operation on the vectors a and b. Multiplication is defined as the polynomial product of the two operands followed by a reduction modulo the generating polynomial $p(x)$. Finally, the inversion of an element $a \in F$ is the process to find an element $a^{-1} \in F$ such that $a \cdot a^{-1} = \mod P(x)$.

Addition is by far the less costly field operation. Thus, its computational complexity is usually neglected (i.e., considered 0). Inversion, on the other hand, is considered the most costly field operation.

Example 4.22. The sum of the two polynomials A and B, denoted in hexadecimal representation as 57 and 83, respectively, is the polynomial denoted by D4, since:

$$\begin{aligned}
&(x^6 + x^4 + x^2 + x + 1) \oplus (x^7 + x + 1) \\
&= x^7 + x^6 + x^4 + x^2 + (1 \oplus 1)x + (1 \oplus 1) \\
&= x^7 + x^6 + x^4 + x^2
\end{aligned}$$

In binary notation we have: $01010111 \oplus 10000011 = 11010100$. Clearly, the addition can be implemented with the bitwise XOR instruction.

Example 4.23. Let us consider the irreducible pentanomial P(x), defined as,

$$P(x) = x^8 + x^4 + x^3 + x + 1 \tag{4.6}$$

Since P(x) is irreducible over $GF(2)$, we have constructed a representation for the field GF(2^8). Hence we can say that byte chains can be considered as elements of GF(2^8). For example, consider the multiplication of the field elements $A = (57)_{16}$ and $B = (83)_{16}$. The resulting field product, $C = AB \mod P(x)$, is $C = (C1)_{16}$, since,

$$\begin{aligned}
&(x^6 + x^4 + x^2 + x + 1) \times (x^7 + x + 1) \\
&= (x^{13} + x^{11} + x^9 + x^8 + x^7) \oplus (x^7 + x^5 + x^3 + x^2 + x) \\
&\quad \oplus (x^6 + x^4 + x^2 + x + 1) \\
&= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1
\end{aligned}$$

and

$$\begin{aligned}
&(x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1) \\
&\equiv x^7 + x^6 + 1 \bmod (x^8 + x^4 + x^3 + x + 1)
\end{aligned}$$

4.3 Elliptic curves

The theory of elliptic curves has been studied extensively in number theory and algebra for the past 150 years. It has been developed a rich and deep theoretical background initially tailored for purely aesthetic reasons. Elliptic curve cryptosystems were proposed for the first time by N. Koblitz [180] and V. Miller [236]. Since then a vast amount of literature has been accumulated on this topic. Recently elliptic curve cryptosystems are widely accepted for security applications like key generation, signature and verification.

Elliptic curves can be defined over real numbers, complex numbers and any other field. In order to explain the geometric properties of elliptic curves let us first examine elliptic curves defined over the real numbers $\mathbb{R}$.

Nonetheless, we stress that elliptic curves over finite fields are the only relevant ones from the cryptographic point of view. More specifically binary representation of elliptic curves will be discussed here which is directly related to the work to be presented in Chapter 10.

In the rest of this section, basic definitions and common operations of elliptic curves will be explained.

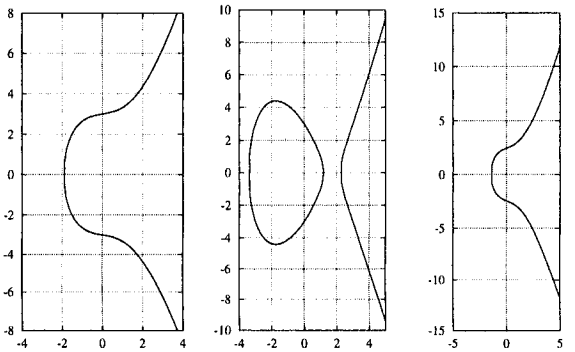

Fig. 4.1. Elliptic Curve Equation $y^2 = x^3 + ax + b$ for Different a and b

4.3.1 Definition

Elliptic curves over real numbers are defined as the set of points (x, y) which satisfy the elliptic curve equation of the form:

$$y^2 = x^3 + ax + b \tag{4.7}$$

where a and b are real numbers. Each choice of a and b produces a different elliptic curve as shown in Figure 4.1. The elliptic curve in Equation 4.7 forms a group if $4a^3 + 27b^2 \neq 0$. An elliptic curve group over real numbers consists of the points on the corresponding elliptic curve, together with a special point $\mathcal{O}$ called the point at infinity.

4.3.2 Elliptic Curve Operations

Elliptic curve groups are additive groups; that is, their basic function is addition. To visualize the addition of two points on the curve, a geometric representation is preferred. We define the negative of a point $P = (x, y)$ as its reflection in the x-axis: the point $-P$ is $(x, -y)$. Also if the point P is on the curve, the point $-P$ is also on the curve.

In the rest of this subsection the addition operation for two distinct points on the curve are explained. Some special cases for the addition of two points on the curve are also described.

- **Adding distinct P and Q**: Let P and Q be two distinct points on an elliptic curve, and $P \neq -Q$. The addition law in an elliptic curve group is $P + Q = R$. For the addition of the points P and Q, a line is drawn through the two points that will intersect the curve at another point, call $-R$. The point $-R$ is reflected in the x-axis to get a point R which is the required point. A geometrical representation of adding two distinct points on the elliptic curve is shown in Figure 4.2.

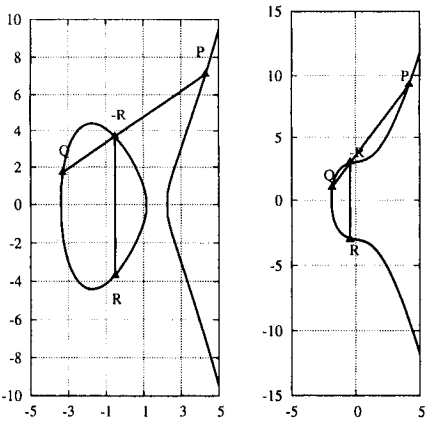

Fig. 4.2. Adding two Distinct Points on an Elliptic curve ($Q \neq -P$)

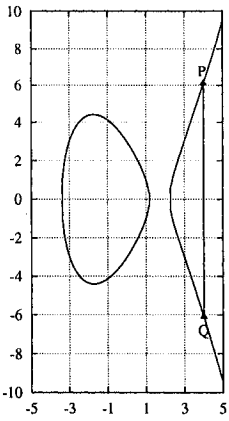

Fig. 4.3. Adding two Points P and Q when $Q = -P$

- **Adding P and $-P$:** The method for adding two distinct points P and Q cannot be adopted for the addition of the points P and $-P$ because the line through P and $-P$ is a vertical line which does not intersect the elliptic curve at a third point as shown in Figure 4.3. This is the reason why the elliptic curve group includes the point at infinity $\mathcal{O}$. By definition, $P+(-P) = \mathcal{O}$. As a result of this equation, $P+\mathcal{O} = \mathcal{P}$ in the elliptic curve group. The point at infinity $\mathcal{O}$ is called the additive identity of the elliptic curve group. All well-defined elliptic curves have an additive identity.

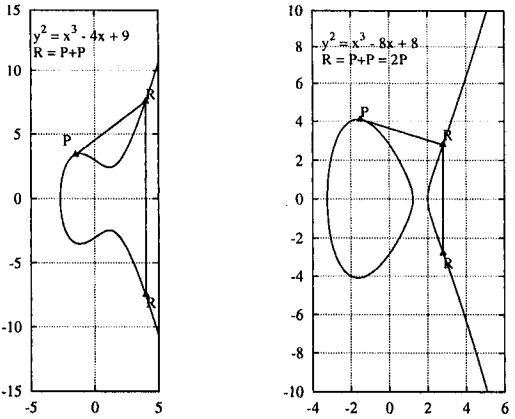

Fig. 4.4. Doubling a Point P on an Elliptic Curve

- **Doubling $P(x, y)$ when $y \neq 0$:**

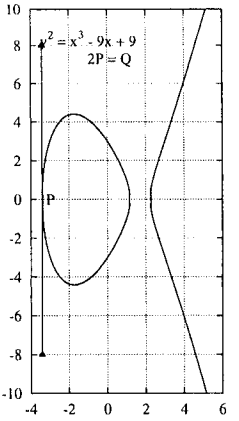

Fig. 4.5. Doubling $P(x, y)$ when $y = 0$

The law for doubling a point on an elliptic curve group is defined by: $P + P = 2P = R$. To add a point $P(x, y)$ to itself, a tangent line to the curve is drawn at the point P. If $y \neq 0$, then the tangent line intersects the elliptic curve at exactly one other point $-R$ as shown in Figure 4.4. The point $-R$ is reflected in the x-axis to R which is the required point. This operation is called doubling the point P.

- **Doubling $P(x, y)$ when $y = 0$:** If for a point $P(x, y)$, $y = 0$, then it does not intersect the elliptic curve at any other point because the tangent line to the elliptic curve at P is vertical. By definition, $2P = \mathcal{O}$ for such a point P. If one wants to find $3P$ in this situation, one can add $2P + P$. This becomes $P + \mathcal{O} = \mathcal{P}$. Thus $3P = P$, $4P = \mathcal{O}$, $5P = P$, $6P = \mathcal{O}$, $7P = P$, etc.

4.3.3 Elliptic Curve Scalar Multiplication

There is no multiplication operation in elliptic curve groups. However, the scalar product kP can be obtained by adding k copies of the same point P, which can be accomplished using the addition and doubling operations explained in the last Subsection. Thus the product $kP = P + P + ...P$ obtained in this way is referred to elliptic curve scalar multiplication. Figure 4.6 shows the scalar multiplication process for obtaining 6 copies of the point P. However for professional elliptic curve cryptosystem implementations, much higher values of k are used. Typically, the bit-length of k is selected in the range of 160-521 bits.

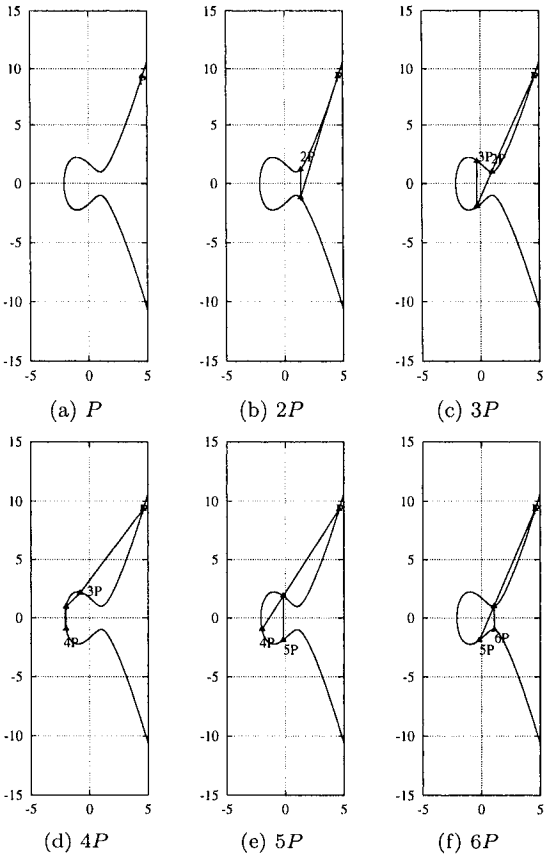

Fig. 4.6. Elliptic Curve Scalar Multiplication kP, for $k = 6$ and for the Elliptic Curve $y^2 = x^3 - 3x + 3$

4.4 Elliptic Curves over $GF(2^m)$

Because of the chracteristic two, the equation for the elliptic curve with the underlying field $GF(2^m)$ is slightly adjusted as shown in Equation 4.8. It is formed by choosing the elements a and b within $GF(2^m)$ with $b \neq 0$.

$$y^2 + xy = x^3 + ax^2 + b \tag{4.8}$$

The elliptic curve includes all points (x, y) which satisfy the elliptic curve equation over $GF(2^m)$ (where x and $y \in GF(2^m)$). An elliptic curve group over

$GF(2^m)$ consists of the points on the corresponding elliptic curve, together with a point at infinity, $\mathcal{O}$.

The points on an elliptic curve can be represented using either two or three coordinates. In affine-coordinate representation, a finite point on $E(GF(2^m))$ is specified by two coordinates x; $y \in GF(2^m)$ satisfying Equation 4.8. The point at infinity has no affine coordinates.

We can make use of the concept of a projective plane over the field $GF(2^m)$ [228]. In this way, one can represent a point using three rather than two coordinates. Then, given a point P with affine-coordinate representation x; y; there exists a corresponding projective-coordinate representation X; Y and Z such that,

$$P(x; y) \equiv P(X; Y; Z)$$

The formulae for converting from affine coordinates to Jacobian projective coordinates and vice versa are given as:

$$\begin{array}{l} \text{Affine-to-Projective: } X = x; \quad Y = y; \quad Z{=}1 \\ \text{Projective-to-Affine: } x = X/Z^2; \; y = Y/Z^3 \end{array}$$

The algebraic formulae for the group law are different for affine and projective coordinates. In the next subsections the group law over $GF(2^m)$ is explained using affine coordinates representation. The group laws for several projective coordinates representations are studied in §4.5.

4.4.1 Point Addition

The negative of a point $P = (x, y)$ is $-P = (x, x + y)$. Assuming that $P \neq Q$, then $R(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$ where:

$$\begin{array}{l} m = \frac{(y_2+y_1)}{(x_2+x_1)} \\ x_3 = m^2 + m + x_1 + x_2 + a \\ y_3 = m(x_1 + x_3) + x_3 + y_1 \end{array} \tag{4.9}$$

As with elliptic curve groups over real numbers, $P + (-P) = \mathcal{O}$, where $\mathcal{O}$ the point at infinity. Furthermore, $P + \mathcal{O} = \mathcal{P}$ for all points P in the elliptic curve group.

4.4.2 Point Doubling

Let $P(x_1, y_1)$ be a point on the curve. If $x_1 = 0$, then $2P = \mathcal{O}$. If $x_1 \neq 0$ then $R = 2P$, and $R(x_2, y_2)$ is given as:

$$\begin{array}{l} x_2 = x_1^2 + \frac{b}{x_1^2} \\ y_2 = x_1^2 + (x_1 + \frac{y_1}{x_1})x_2 + x_2 \end{array} \tag{4.10}$$

Let us recall that a is one of the parameters chosen with the elliptic curve and that m is the slope of the line through P and Q.

4.4.3 Order of an Elliptic Curve

Notice that the elliptic curve $E(\mathbb{F}_q)$, namely the collection of all the points in $\mathbb{F}_q$ that satisfy Eq. (4.10) can only be finitely many. Even if every possible pair (x, y) were on the curve, there would be only q^2 possibilities. As a matter of fact, the curve $E(\mathbb{F}_q)$ could have at most $2q+1$ points because we have one point at infinity and $2q$ pairs (x, y) (for each x we have two values of y).

The total number of points in the curve, including the point $\mathcal{O}$, is called the *order* of the curve. The order is written $\#E(\mathbb{F}_q)$. A celebrated result discovered by Hasse gives the lower and the upper bounds for this number.

Theorem 4.24. *[227] Let $\#E(\mathbb{F}_q)$ be the number of points in $E(\mathbb{F}_q)$. Then,*

$$|\#E(\mathbb{F}_q) - (q+1)| \leq 2\sqrt{q} \tag{4.11}$$

The interval $[q+1-2\sqrt{q}, q+1+2\sqrt{q}]$ is called the Hasse interval.

As we did in the case of finite fields, we can also introduce the concept of the order of an element in elliptic curves. The order of a point P on $E(\mathbb{F}_q)$ is the smallest integer n such that $nP = 0$. The order of any point it is always defined, and divides the order of the curve $\#E(\mathbb{F}_q)$. This guarantees that if r and l are integers, then $rP = lP$ if and only if $r \equiv l \pmod{n}$.

4.4.4 Elliptic Curve Groups and the Discrete Logarithm Problem

Every cryptosystem is based on a hard mathematical problem that is computationally infeasible to solve. The discrete logarithm problem is the basis for the security of many cryptosystems including Elliptic Curve Cryptosystems. More specifically the security of elliptic curve cryptosystems relies on Elliptic Curve Discrete Logarithmic Problem (ECDLP).

In the last Section we examined two elliptic curve operations: point addition and point doubling. Both point addition and doubling operations can be used to compute any number of copies of a point ($2P$, $3P$, kP, etc). The determination of a point kP in this manner is referred to as *Scalar Multiplication* of a point. In the rest of this Section we present a small example of how to compute such elliptic curve operation.

4.4.5 An Example

Let $F = GF(2^4)$ be a binary finite field with defining primitive trinomial $p(x)$ given as,

$$p(x) = x^4 + x + 1. \tag{4.12}$$

Then, if α is a root of $p(x)$, we have $p(\alpha) = 0$, which implies,

$$p(\alpha) = \alpha^4 + \alpha + 1 = 0. \tag{4.13}$$

For binary field arithmetic, addition is equivalent to subtraction. Hence, the above equation can be rewritten as

$$\alpha^4 = \alpha + 1. \tag{4.14}$$

Using equation (4.14), one can now express each one of the 15 nonzero elements of F as is shown in Table 4.1. Notice that we can define any one of the $q = 2^4$ elements of F using only four coordinates.

Element in $GF(2^m)$	Polynomial	Coordinates
0	0	(0000)
α	α	(0010)
α^2	α^2	(0100)
α^3	α^3	(1000)
α^4	$\alpha + 1$	(0011)
α^5	$\alpha^2 + \alpha$	(0110)
α^6	$\alpha^3 + \alpha^2$	(1100)
α^7	$\alpha^3 + \alpha + 1$	(1011)
α^8	$\alpha^2 + 1$	(0101)
α^9	$\alpha^3 + \alpha$	(1010)
α^{10}	$\alpha^2 + \alpha + 1$	(0111)
α^{11}	$\alpha^3 + \alpha^2 + \alpha$	(1110)
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	(1111)
α^{13}	$\alpha^3 + \alpha^2 + 1$	(1101)
α^{14}	$\alpha^3 + 1$	(1001)
α^{15}	1	(0001)

Table 4.1. Elements of the field $F = GF(2^4)$, Defined Using the Primitive Trinomial of Eq. ((4.12))

Notice that all the elements in F can be described by any of the three representations used in Table 4.1, namely, polynomial representation, coordinate representation and powers of the primitive element α.

Let us now consider a non-supersingular elliptic curve defined as the set of points $(x, y) \in F \times F$ that satisfy

$$y^2 + xy = x^3 + \alpha^{13}x^2 + \alpha^6 \tag{4.15}$$

Notice that for the coefficients a and b of equation (4.8), we have selected the values α^{13} and α^6, respectively. There exist a total of 14 solutions in such a curve, including the point at infinite $\mathcal{O}$. Using table 4.1, we can see that, for example, the point,

$$P = (x_p, y_p) = (\alpha^3, \alpha^2) \tag{4.16}$$

satisfies equation (4.15) over $\mathbb{F}_2^4$, since

$$\begin{array}{ll} y^2 + xy & = x^3 + \alpha^{13}x^2 + \alpha^6 \\ (\alpha^2)^2 + \alpha^3\alpha^2 & = (\alpha^3)^3 + \alpha^{13}(\alpha^3)^2 + \alpha^6 \\ \alpha^4 + \alpha^5 & = \alpha^9 + \alpha^{19} + \alpha^6 \\ & = \alpha^9 + \alpha^4 + \alpha^6 \\ (0011) + (0110) & = (1010) + (0011) + (1100) \\ (0101) & = (0101), \end{array} \tag{4.17}$$

Where we have used the identity $\alpha^{15} = 1$. All the thirteen finite points which satisfy equation (4.15) are shown in figure 4.7.

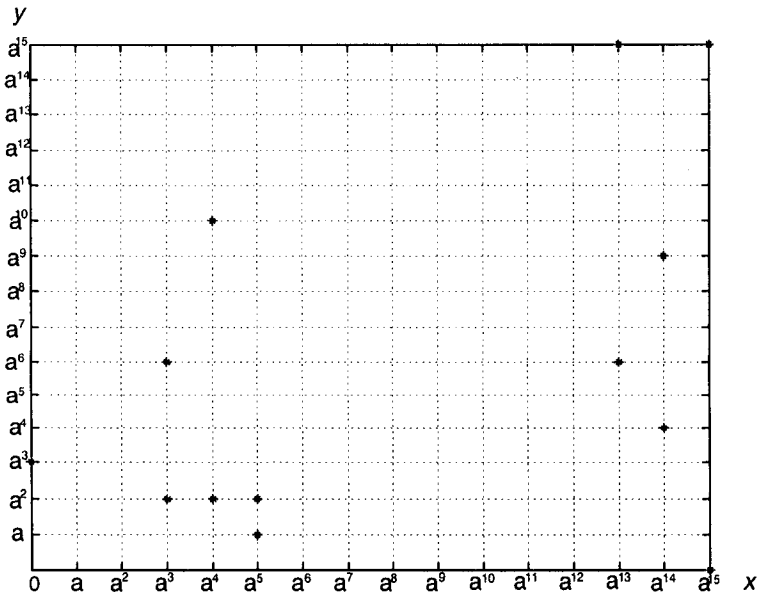

Fig. 4.7. Elements in the Elliptic Curve of Equation (4.15)

Let us now use equation (4.10) to double the point $P = (\alpha^3, \alpha^2)$. Using once again table 4.1, we obtain,

$$\begin{aligned} x_{2p} &= x_p^2 + \tfrac{b}{x_p^2} \\ &= (\alpha^3)^2 + \alpha^6 \cdot (\alpha^3)^{-2} \\ &= \alpha^6 + \alpha^6 \cdot \alpha^{-6} = \alpha^6 + 1 = \alpha^{13} \\ y_{2p} &= x_p^2 + \left(x_p + \tfrac{y_p}{x_p}\right) x_{2p} + x_{2p} \\ &= \alpha^6 + \left(\alpha^3 + \alpha^2 \cdot \alpha^{-3}\right) \alpha^{13} + \alpha^{13} \\ &= \alpha^6 + \left(\alpha^3 + \alpha^{-1}\right) \alpha^{13} + \alpha^{13} \\ &= \alpha^6 + \alpha^1 + \alpha^{12} + \alpha^{13} = \alpha^6 \end{aligned} \tag{4.18}$$

It can be verified from figure 4.7 that the result obtained above is indeed a point in the elliptic curve of equation (4.15).

As we mentioned in §4.4.3, we can keep adding P to its scalar multiples, but eventually, after $n \leq \#E(\mathbb{F}_q)$ scalar multiplications, we will obtain the point at infinite $\mathcal{O}$ as a result. Recall that the integer n is called the order of the point P. For the case in hand, P happens to have a prime order $k = 7$. Notice that as it was stated in §4.4.3, the order n of P divides the order of the curve $\#E(\mathbb{F}_q)$. Table 4.2 lists all the six finite multiples of P.

P	$2P$	$3P$	$4P$	$5P$	$6P$
(α^3, α^2)	(α^{13}, α^6)	(α^{14}, α^9)	(α^{14}, α^4)	$(\alpha^{13}, \alpha^{15})$	(α^3, α^6)

Table 4.2. Scalar Multiples of the Point P of Equation (4.16)

Obviously, in a true cryptographic application the parameter n should be chosen large enough so that efficient generation of such a look-up table approach, becomes unfeasible. In today's practice, $n \geq 2^{160}$ has proved to be sufficient.

4.5 Point Representation

In order to generate an Abelian group over elliptic curves, it was necessary to define an elliptic curve group law. More specifically, we defined the point addition and point doubling primitives of Equations (4.9) and (4.10). However, the computational cost of those equations involves the calculation of a costly field inverse operation plus several field multiplications.

Since the relation (I/M) defined as the computational cost of a field inversion over the computational cost of a field multiplication is above 8 and 20 in hardware and software implementations, respectively, there is a strong motivation for finding alternative point representations that allow the trading of the costly field inversions by less expensive field multiplications.

As we have seen at the beginning in §4.4, elliptic point representation in two coordinates is called *affine representation*, whereas the equivalent point representation in three coordinates is called *Projective representation.*

It can be shown that each affine point can be related one-to-one with a unique equivalence class. Then, each elliptic point is represented by a triple that satisfy the corresponding equivalence class. Notice that it results necessary to redefine the addition and doubling operations in the projective representation.

As it will be explained in the rest of this Section, the projective group law can be implemented without utilizing field inversions at the price of increasing the total number of field multiplications. As a matter of fact, field inversions are only required when converting from projective representation to affine representation[2], which becomes valuable in situations where we are planning to perform many point additions and doublings in a successive manner (such as in elliptic curve scalar multiplication).

4.5.1 Projective Coordinates

Let c and d be positive integers over the field K. It is possible to define an equivalent class $K^3 \setminus \{(0,0,0)\}$ as follows,

$$(X_1, Y_1, Z_1) \sim (X2, Y2, Z2) | \text{ If } X_1 = \lambda^c X_2, Y_1 = \lambda^d Y_2, Z_1 = \lambda Z_2.$$

The equivalent class

$$(X : Y : Z) = \left\{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in K^*\right\}.$$

is called a *projective point* [129], and (X, Y, Z) a *representative point* of such class, that is to say, any point within the class is a representative point. Specifically, if $Z \neq 0$, $(\frac{X}{Z^c}, \frac{Y}{Z^d}, 1)$ is a point representative of the equivalence class $(X : Y : Z)$.

Therefore, if we define the set of all projective points (equivalent classes) for each possible λ in the field K^* as,

$$P(K)^* = \{(X : Y : Z) : X, Y, Z \in K, Z \neq 0\},$$

we obtain a one-to-one correspondence between the point $P(K)^*$ and the set of affine points,

$$A(K) = \{(x, y : x, y \in K)\}.$$

Each point in the *affine coordinate system*, corresponds to the set defined by an equivalence class in particular. The set of point belonging to $P(K)^0 = \{(X : Y : Z) : X, Y, Z \in K, Z = 0\}$ is called the *line at infinity*, because this class does not correspond with any element in the set of affine points.

[2] In §4.4 the explicit conversion equations from affine to Jacobian projective coordinates and vice versa were stated.

The Weierstrass equation for an elliptic curve $E(K)$ can be defined in projective coordinates by replacing x by $\frac{X}{Z^c}$ and y by $\frac{Y}{Z^d}$. The constant values c and d will determine the characteristic of the elliptic curve arithmetic and hence, the definition of the point addition algorithm in such representation.

4.5.2 López-Dahab Coordinates

The most popular projective coordinate system are the *standard* where $c = 1$ and $d = 1$, Jacobians, with $c = 2$ and $d = 3$ and López-Dahab (LD) coordinates, , with $c = 1$ and $d = 2$. The latter system of coordinates offers algorithms for computing the addition in *mixed coordinates*, i.e., one point is given in affine coordinates while the other is given in projective coordinates. LD coordinates are highly attractive for hardware implementation because they only employ 8 field multiplications for performing a point addition operation.

In *López-Dahab (LD)* projective coordinates [210] the projective point (X: Y: Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z^2$. Therefore, the elliptic curve equation (4.8) mapped to LD projective coordinates can be written as,

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + Z^4 \tag{4.19}$$

The point at infinity is represented now as $\mathcal{O} = (1 : 0 : 0)$. For any arbitrary point P on the curve, it holds that $P+\mathcal{O} = \mathcal{O}+\mathcal{P} = \mathcal{P}$. Let $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : 1)$ be two arbitrary points belonging to the curve 4.19. Then the point $-P = (X_1 : X_1 + Y_1 : Z)$ is the addition inverse of the point P. The point doubling primitive $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$ can be performed at a computational cost of 2 general field multiplications plus two field multiplication by the elliptic curve constant b as [212],

$$\begin{aligned} Z_3 &= X_1^2 \cdot Z_1^2, \\ X_3 &= X_1^4 + b \cdot Z_1^4, \\ Y_3 &= bZ_1{}^4 Z_3 + X_3 \cdot (aZ_3 + Y_1{}^2 + bZ_1{}^4) \end{aligned} \tag{4.20}$$

Whereas if $Q \neq -P$, the point addition primitive $(X_1 : Y_1 : Z_1) + (X_2 : Y_2) = (X_3 : Y_3 : Z_3)$ can be performed at a computational cost of 8 field multiplications as,

$$\begin{aligned} A &= Y_2 \cdot Z_1^2 + Y_1; & B &= X_2 \cdot Z_1 + X_1; \\ C &= Z_1 \cdot B; & D &= B^2 \cdot (C + aZ_1^2); \\ Z_3 &= C^2; & E &= A \cdot C; \\ X_3 &= A^2 + D + E; & F &= X_3 + X_2 \cdot Z_3; \\ G &= (X_2 + Y_2) \cdot Z_3^2; & Y_3 &= (E + Z_3) \cdot F + G \end{aligned} \tag{4.21}$$

4.6 Scalar Representation

The vast majority of algorithms reported for computing the scalar multiplication in an efficient manner are based in the Horner polynomial representation,

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0 = a_0 + (a_1 + (a_2 + (\ldots + (a_{n-1} + (a_n + x)x)\ldots)x)x)x.$$

where the scalar k is represented using its binary expansion, namely, $k = b_n 2^n + b_{n-1} + 2^{n-1} + \ldots + b_1 2 + b_0$ where $b_i \in [0, 1]$.

4.6.1 Binary Representation

Algorithm 4.3 Basic Doubling & Add algorithm for Scalar Multiplication

Require: $k = (k_{m-1}, k_{m-2}, \ldots, k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(\mathbb{F}_{2^m})$
Ensure: $Q = kP$
1: $Q = P$;
2: **for** $i = m - 2$ **downto** 0 **do**
3: $Q = 2 \cdot Q$ (point doubling) ;
4: **if** $k_i = 1$ **then**
5: $Q = Q + P$ (point addition);
6: **end if**
7: **end for**
8: **Return Q**

The traditional method for computing the elliptic operation kP is based in the binary representation of k. If $k = \sum_{j=0}^{m-1} b_j 2^j$, where each $b_j \in \{0, 1\}$, then kP can be computed as [227]:

$$kP = \sum_{j=0}^{m-1} b_j 2^j P = 2(\ldots 2(2b_{m-1}P + b_{m-2}P) + \ldots) + b_0 P.$$

This method requires $m-1$ point doublings and $w_k - 1$ point additions, where w_k is the Hamming weight (total number of coefficients $b_j = 1$) of the binary representation of the scalar k.

4.6.2 Recoding Methods

It is possible to reduce the number of subsequent point additions using a *recoding* of the the exponent [154, 239, 76, 176]. The recoding techniques use the identity

$$2^{i+j-1} + 2^{i+j-2} + \cdots + 2^i = 2^{i+j} - 2^i$$

to collapse a block of 1s in order to obtain a *sparse* representation of the exponent. Thus, a redundant signed-digit representation of the exponent using the digits $\{0, 1, -1\}$ will be obtained. For example, (011110) can be recoded as

Algorithm 4.4 The Recoding Binary algorithm for Scalar Multiplication

Require: $k = (k_{m-1}, k_{m-2}, \ldots, k_1, k_0)_2$ with $k_i \in [\![-1, 0, 1]\!]$, $P(x, y, z) \in E(\mathbb{F}_{2^m})$
Ensure: $Q = kP$
1: $Q = P$;
2: **for** $i = m - 2$ **downto** 0 **do**
3: $Q = 2 \cdot Q$ (point doubling) ;
4: **if** $k_i = 1$ **then**
5: $Q = Q + P$ (point addition);
6: **else if** $k_i = \bar{1}$ **then**
7: $Q = Q - P$ (point subtraction);
8: **end if**
9: **end for**
10: **Return Q**

$$(011110) = 2^4 + 2^3 + 2^2 + 2^1$$
$$(1000\bar{1}0) = 2^5 - 2^1.$$

The recoding binary method is given in the Algorithm 4.4. Note that even though the number of bits of k is equal to m, the number of bits in the recoded exponent $\hat{k}$ can be $m + 1$, for example, (111) is recoded as $(100\bar{1})$. Thus, the recoding binary algorithm starts from the bit position m in order to compute kP by computing $\hat{k}P$ where $\hat{k}$ is the $(k + 1)$-bit recoded exponent such that $\hat{k} = k$.

Let us discuss an explicit toy example of scalar multiplication using the recoding binary method. Let $k = 119 = (1110111)$. The (nonrecoding) binary method requires 6 point doublings plus 5 point additions in order to compute $119P$. In the recoding binary method, we first obtain a sparse signed-digit representation of 119. It is easy to verify the following:

$$\text{Exponent: } 119 = 01110111,$$
$$\text{Recoded Exponent: } 119 = 1000\bar{1}00\bar{1}.$$

The recoding binary method then computes $119P$ as follows:

f_i	Step 3	Steps 4-8
1	P	P
0	$2(P) = 2P$	$2P$
0	$2(2P) = 4P$	$4P$
0	$2(4P) = 8P$	$8P$
$\bar{1}$	$2(8P) = 16P$	$16P - P = 15P$
0	$2(15P) = 30P$	$30P$
0	$2(30P) = 60P$	$60P$
$\bar{1}$	$2(60P) = 120P$	$120P - P = 119P$

Table 4.3. A Toy Example of the Recoding Algorithm

The number of point doublings plus additions is equal to $7 + 2 = 9$ which is 2 less group operations than that of the binary method. The number of

point doubling operations required by the recoding binary method can be at most 1 more than that of the binary method. The number of subsequent point additions, on the other hand, can be significantly less. This is simply equal to the number of nonzero digits of the recoded exponent. Thus, the number of point addition operations can be reduced if we obtain a sparse signed-digit representation of the scalar k.

4.6.3 ω-NAF Representation

Algorithm 4.5 ω-NAF Expansion Algorithm

Require: A positive integer k.
Ensure: $U = \omega NAF(k)$
  for $\{i = 0; k > 0; i++\}$ **do**
    if k is odd **then**
      $U_i = k \bmod 2^w$
      $k = k - U_i$;
    else
      $U_i = 0$;
    end if
    $k = k/2$;
  end for
  Return(U);

The recoding binary algorithm can be generalized for designing algorithms even more efficient at the price of using memory for storing pre-computed results. The basic *window method* ω with $\omega > 1$ expand any positive integer k using a Non-Adjacent Form (NAF) of width ω expressed as,

$$k = \sum_{i=0}^{l-1} u_i 2^i$$

Where,

- Each coefficient u_i different than zero is odd and with magnitude less than 2^{w-1};
- Given two consecutive coefficients u_i, at least one of them is nonzero;
- When using $\omega = 2$ we have the recoding binary algorithm explained above.

We write the ωNAF as,

$$\omega NAF(k) = \{u_{l-1}, ...u_0\}.$$

Algorithm 4.5 generates an ωNAF expansion of a positive scalar k. Every time that k is odd, the ω most significant bits are scanned in order to determine

the corresponding congruence class (mod 2^ω) for k. The congruence class U_i is then subtracted from k, making the new coefficient $k - U_i$ divisible by 2^w. This will guarantee a run of $w - 1$ zero coefficients in the next iterations.

In average, the Hamming weight of a ωNAF expansion is $(w+1)^{-1}$. This will directly impact the performance of the scalar multiplication algorithm because of a saving on the point additions required for computing the scalar multiplication. That saving is obtained at the price of storing multiples of the base elliptic point. Notice, however, that the total number of point doublings remains the same. Table 4.4 presents the main characteristics of the binary, recoded binary an ωNAF expansions of the scalar k, respectively.

Table 4.4. Comparing Different Representations of the Scalar k

Point Representation	Length	# PA	# PD	Pre-computation
Binary	m	$\frac{m}{2}$	m	—
recoded binary	m	$\frac{m}{3}$	$m+1$	—
ωNAF	m	$\frac{m}{w+1}$	$m+1$	Table of $2^{w-1}-1$ m-bit multiples.

4.7 Conclusions

In this Chapter we briefly reviewed some of the most important mathematical concepts useful for understanding cryptographic algorithms. We explained the most relevant definitions and theorems of the elementary theory of numbers relevant to the subject of cryptography. Moreover, we defined the concept of finite fields and related arithmetic operations. We gave a brief introduction to elliptic curve cryptography, explaining the mathematical concepts of elliptic curve group, group order, group law and point representation among others.

These concepts will be useful for understanding the material contained in the Chapters to come.

5

Prime Finite Field Arithmetic

The modular exponentiation operation is a common operation for scrambling; it is used in several cryptosystems. For example, the Diffie-Hellman key exchange scheme requires modular exponentiation [64]. Furthermore, the ElGamal signature scheme [80] and the Digital Signature Standard (DSS) of the National Institute for Standards and Technology [90] also require the computation of modular exponentiation. However, we note that the exponentiation process in a cryptosystem based on the discrete logarithm problem is slightly different: The base (M) and the modulus (n) are known in advance. This allows some precomputation since powers of the base can be precomputed and saved [35]. In the exponentiation process for the RSA algorithm, we know the exponent (e) and the modulus (n) in advance but not the base (M); thus, such optimizations are not likely to be applicable.

In the following sections we will review techniques for implementation of the modular exponentiation operation in hardware. We will study techniques for exponentiation, modular multiplication, modular addition, and addition operations. We intend to cover mathematical and algorithmic aspects of the modular exponentiation operation, providing the necessary knowledge to the hardware designer who is interested implementing modular algorithm on hardware platforms. We draw our material from computer arithmetic books [352, 138, 370, 187], collection of articles [75, 335], and journal and conference articles on hardware structures for performing the modular multiplication and exponentiations [288, 185, 322, 135, 34, 179, 180, 181, 365].

Therefore, in the remainder of this Chapter we will study algorithms for computing efficiently the most basic modular arithmetic operations. We will assume that the underlying exponentiation heuristic is either the binary method, or any of the advanced m-ary algorithm with the necessary register space already made available. This assumption allows us to concentrate on developing time and area efficient algorithms for the basic modular arithmetic operations, which is the current challenge because of the operand size.

modular arithmetic operations, which is the current challenge because of the operand size.

The literature is replete with residue arithmetic techniques applied to signal processing, see for example, the collection of papers in [337]. However, in such applications, the size of operands are very small, usually around 5–10 bits, allowing table lookup approaches. Besides the moduli are fixed and known in advance, which is definitely not the case for our application. Thus, entirely new set of approaches are needed to design time and area efficient hardware structures for performing modular arithmetic operations to be used in cryptographic applications.

5.1 Addition Operation

In this section, we study algorithms for computing the sum of two k-bit integers A and B. Let A_i and B_i for $i = 1, 2, \ldots, k-1$ represent the bits of the integers A and B, respectively. We would like to compute the sum bits S_i for $i = 1, 2, \ldots, k-1$ and the final carry-out C_k as follows:

$$\begin{array}{r} A_{k-1}\ A_{k-2} \cdots A_1\ A_0 \\ +\ B_{k-1}\ B_{k-2} \cdots B_1\ B_0 \\ \hline C_k\ S_{k-1}\ S_{k-2} \cdots S_1\ S_0 \end{array}$$

We will study the following algorithms: the carry propagate adder (CPA), the carry completion sensing adder (CCSA), the carry look-ahead adder (CLA), the carry save adder (CSA), and the carry delayed adder (CDA) for computing the sum and the final carry-out.

5.1.1 Full-Adder and Half-Adder Cells

The building blocks of these adders are the full-adder (FA) and half-adder (HA) cells. Thus, we briefly introduce them here. A full-adder is a combinational circuit with 3 input and 2 outputs. The inputs A_i, B_i, C_i and the outputs S_i and C_{i+1} are boolean variables. It is assumed that A_i and B_i are the ith bits of the integers A and B, respectively, and C_i is the carry bit received by the ith position. The FA cell computes the sum bit S_i and the carry-out bit C_{i+1} which is to be received by the next cell. The truth table of the FA cell is as follows:

A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The boolean functions of the output values are as

$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i,$$
$$S_i = A_i \oplus B_i \oplus C_i.$$

Similarly, an half-adder is a combinational circuit with 2 inputs and 2 outputs. The inputs A_i, B_i and the outputs S_i and C_{i+1} are boolean variables. It is assumed that A_i and B_i are the ith bits of the integers A and B, respectively. The HA cell computes the sum bit S_i and the carry-out bit C_{i+1}. Thus, an half-adder is easily obtained by setting the third input bit C_i to zero. The truth table of the HA cell is as follows:

A_i	B_i	C_{i+1}	S_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The boolean functions of the output values are as $C_{i+1} = A_iB_i$ and $S_i = A_i \oplus B_i$, which can be obtained by setting the carry bit input C_i of the FA cell to zero. Fig. 5.1 illustrates the FA and HA cells.

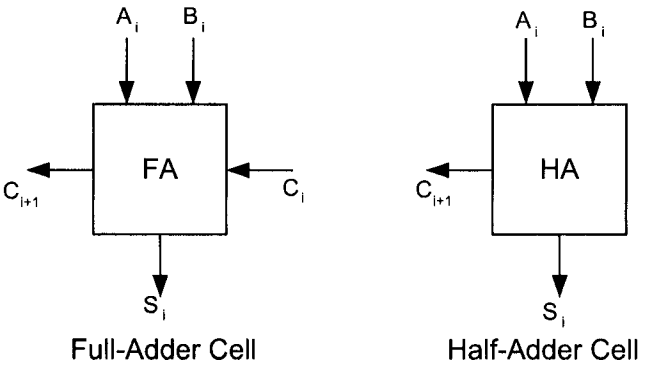

Fig. 5.1. Full-Adder and Half-Adder Cells

5.1.2 Carry Propagate Adder

The carry propagate adder is a linearly connected array of full-adder (FA) cells. The topology of the CPA is illustrated below in Fig. 5.2 for $k = 8$.
The total delay of the carry propagate adder is k times the delay of a single full-adder cell. This is because the ith cell needs to receive the correct value

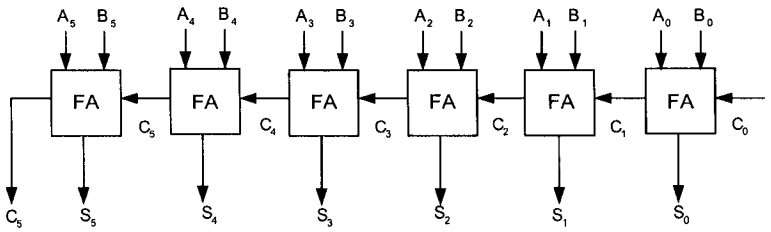

Fig. 5.2. Carry Propagate Adder

of the carry-in bit C_i in order to compute its correct outputs. Tracing back to the 0th cell, we conclude that a total of k full-adder delays is needed to compute the sum vector S and the final carry-out C_k. Furthermore, the total area of the k-bit CPA is equal to k times a single full-adder cell area. The CPA scales up very easily, by adding additional cells starting from the most significant.

The subtraction operation can be performed on a carry propagate adder by using 2's complement arithmetic. Assuming we have a k-bit CPA available, we encode the positive numbers in the range $[0, 2^{k-1} - 1]$ as k-bit binary vectors with the most significant bit being 0. A negative number is then represented with its most significant bit as 1. This is accomplished as follows: Let $x \in [0, 2^{k-1}]$, then $-x$ is represented by computing $2^k - x$. For example, for $k = 3$, the positive numbers are $0, 1, 2, 3$ encoded as $000, 001, 010, 011$, respectively. The negative 1 is computed as $2^3 - 1 = 8 - 1 = 7 = 111$. Similarly, -2, -3, and -4 are encoded as 110, 101, and 100, respectively. This encoding system has two advantages which are relevant in performing modular arithmetic operations:

- The sign detection is easy: the most significant bit gives the sign.
- The subtraction is easy: In order to compute $x - y$, we first represent $-y$ using 2's complement encoding, and then add x to $-y$.

The CPA has several advantages but one clear disadvantage: the computation time is too long for RSA computations, in which the operand size is in the order of several hundreds, up to 2048 bits. Thus, we need to explore other techniques with the hope of building circuits which require less time without significantly increasing the area.

5.1.3 Carry Completion Sensing Adder

The carry completion sensing adder is an asynchronous circuit with area requirement proportional to k. It is based on the observation that the average time required for the carry propagation process to complete is much less than the worst case which is k full-adder delays. For example, the addition of 15213 by 19989 produces the longest carry length as 5, as shown below in Fig. 5.3.

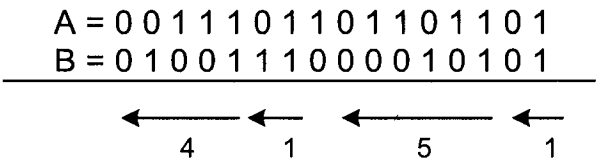

Fig. 5.3. Carry Completion Sensing Adder

A statistical analysis shows that the average longest carry sequence is approximately 4.6 for a 40-bit adder [108]. In general, the average longest carry produced by the addition of two k-bit integers is upper bounded by $\log_2 k$. Thus, we can design a circuit which detects the completion of all carry propagation processes, and completes in $\log_2 k$ time in the average.

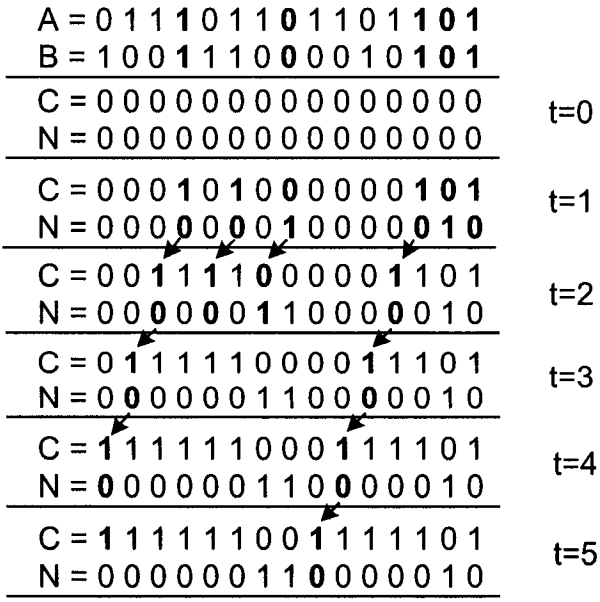

Fig. 5.4. Detecting Carry Completion

In order to accomplish this task, we introduce a new variable N in addition to the carry variable C. The value of C and N for ith position is computed using the values of A and B for the ith position, and the previous C and N values, as follows:

$$(A_i, B_i) = (0,0) \Longrightarrow (C_i, N_i) = (0,1)$$
$$(A_i, B_i) = (1,1) \Longrightarrow (C_i, N_i) = (1,0)$$
$$(A_i, B_i) = (0,1) \Longrightarrow (C_i, N_i) = (C_{i-1}, N_{i-1})$$
$$(A_i, B_i) = (1,0) \Longrightarrow (C_i, N_i) = (C_{i-1}, N_{i-1})$$

Initially, the C and N vectors are set to zero. The cells which produce C and N values start working as soon as the values of A and B are applied to them in parallel. The output of a cell (C_i, N_i) settles when its inputs (C_{i-1}, N_{i-1}) are settled. When all carry propagation processes are complete, we have either $(C_i, N_i) = (0,1)$ or $(C_i, N_i) = (1,0)$ for all $i = 1, 2, \ldots, k$. Thus, the end of carry completion is detected when all $X_i = C_i + N_i = 1$ for all $i = 1, 2, \ldots, k$, which can be accomplished by using a k-input AND gate. The procedure described above is illustrated in Fig. 5.4.

5.1.4 Carry Look-Ahead Adder

The carry look-ahead adder is based on computing the carry bits C_i prior to the summation. The carry look-ahead logic makes use of the relationship between the carry bits C_i and the input bits A_i and B_i. We define two variables G_i and P_i, named as the generate and the propagate functions, as follows:

$$G_i = A_i B_i,$$
$$P_i = A_i + B_i.$$

Then, we expand C_1 in terms of G_0 and P_0, and the input carry C_0 as

$$C_1 = A_0 B_0 + C_0(A_0 + B_0) = G_0 + C_0 P_0.$$

Similarly, C_2 is expanded in terms G_1, P_1, and C_1 as

$$C_2 = G_1 + C_1 P_1.$$

When we substitute C_1 in the above equation with the value of C_0 in the preceding equation, we obtain C_2 in terms G_0, G_1, P_0, P_1, and C_0 as

$$C_2 = G_1 + C_1 P_1 = G_1 + (G_0 + C_0 P_0) P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1.$$

Proceeding in this fashion, we can obtain C_i as function of C_0 and $G_0, G_1, \ldots, G_i$ and $P_0, P_1, \ldots, P_i$. The carry functions up to C_4 are given below:

$$C_1 = G_0 + C_0 P_0,$$
$$C_2 = G_1 + G_0 P_1 + C_0 P_0 P_1,$$
$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2,$$
$$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3.$$

The carry look-ahead logic uses these functions in order to compute all C_is in advance, and then feeds these values to an array of EXOR gates to compute the sum vector S. The ith element of the sum vector is computed using

$$S_i = A_i \oplus B_i \oplus C_i.$$

The carry look-ahead adder for $k = 3$ is illustrated in Fig. 5.5.

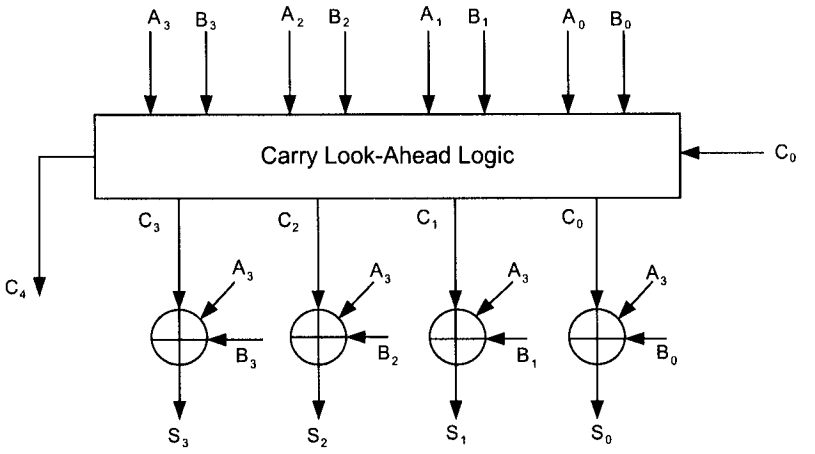

Fig. 5.5. Carry Look-Ahead Adder

The CLA does not scale up very easily. In order to deal with large operands, we have basically two approaches:

- The block carry look-ahead adder: First we build small (4-bit or 8-bit) carry look-ahead logic cells with section generate and propagate functions, and then stack these to build larger carry look-ahead adders [138, 370, 187].
- The complete carry look-ahead adder: We build a complete carry look-ahead logic for the given operand size. In order to accomplish this task, the carry look-ahead functions are formulated in a way to allow the use of the parallel prefix circuits [32, 188, 196].

The total delay of the carry look-ahead adder is $O(\log k)$ which can be significantly less than the carry propagate adder. There is a penalty paid for this gain: The area increases. The block carry look-ahead adders require $O(k \log k)$ area, while the complete carry look-ahead adders require $O(k)$ area by making use of efficient parallel prefix circuits [196, 197]. It seems that a carry look-ahead adder larger than 256 bits is not cost effective, considering the fact there are better alternatives, e.g., the carry save adders. Even by employing block carry look-ahead approaches, a carry look-ahead adder with 1024 bits seems not feasible or cost effective.

5.1.5 Carry Save Adder

The carry save adder seems to be the most useful adder for our application. It is simply a parallel ensemble of k full-adders without any horizontal connection. Its main function is to add three k-bit integers A, B, and C to produce two integers C' and S such that

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

As an example, let $A = 40$, $B = 25$, and $C = 20$, we compute S and C' as shown below:

$$\begin{array}{rl} A = 40 = & 1\,0\,1\,0\,0\,0 \\ B = 25 = & 0\,1\,1\,0\,0\,1 \\ C = 20 = & 0\,1\,0\,1\,0\,0 \\ \hline S = 37 = & 1\,0\,0\,1\,0\,1 \\ C' = 48 = & 0\,1\,1\,0\,0\,0 \end{array}$$

The ith bit of the sum S_i and the $(i+1)$st bit of the carry C'_{i+1} is calculated using the equations

$$S_i = A_i \oplus B_i \oplus C_i.$$
$$C'_{i+1} = A_i B_i + A_i C_i + B_i C_i,$$

in other words, a carry save adder cell is just a full-adder cell. A carry save adder, sometimes named a one-level CSA, is illustrated in Fig. 5.6 for $k = 6$.

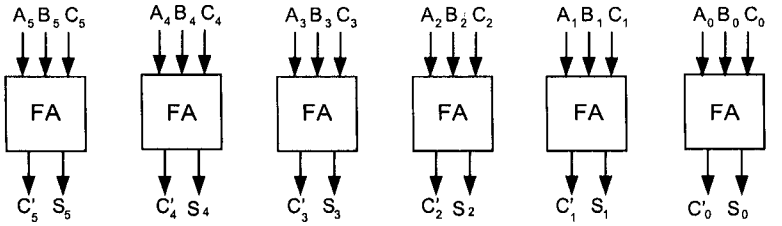

Fig. 5.6. Carry Save Adder

Since the input vectors A, B, and C are applied in parallel, the total delay of a carry save adder is equal to the total delay of a single FA cell. Thus, the addition of three integers to compute two integers requires a single FA delay. Furthermore, the CSA requires only k times the areas of FA cell, and scales up very easily by adding more parallel cells. The subtraction operation can also be performed by using 2's complement encoding. There are basically two disadvantages of the carry save adders:

- It does not really solve our problem of adding two integers and producing a single output. Instead, it adds three integers and produces two such that sum of these two is equal to the sum of three inputs. This method may not be suitable for application which only needs the regular addition.
- The sign detection is hard: When a number is represented as a carry-save pair (C, S) such that its actual value is $C + S$, we may not know the exact sign of total sum $C + S$. Unless the addition is performed in full length, the correct sign may never be determined.

We will explore this sign detection problem in an upcoming section in more detail. For now, it suffices to briefly mention the sign detection problem, and introduce a method of sign detection. This method is based on adding a few of the most significant bits of C and S in order to calculate (estimate) the sign. As an example, let $A = -18$, $B = 19$, $C = 6$. After the carry save addition process, we produce $S = -5$ and $C' = 12$, as shown below. Since the total sum $C' + S = 12 - 5 = 7$, its correct sign is 0. However, when we add the first most significant bits, we estimate the sign incorrectly.

$A = -18 =$	1 0 1 1 1 0		
$B = 19 =$	0 1 0 0 1 1		
$C = 6 =$	0 0 0 1 1 0		
$S = -5 =$	1 1 1 0 1 1		
$C' = 12 =$	0 0 0 1 1 0		
	1		(1 MSB)
	1 1		(2 MSB)
	0 0 0		(3 MSB)
	0 0 0 1		(4 MSB)
	0 0 0 1 1		(5 MSB)
	0 0 0 1 1 1		(6 MSB)

The correct sign is computed only after adding the first three most significant bits. In the worst case, up to a full length addition may be required to calculate the correct sign.

5.1.6 Carry Delayed Adder

The carry delayed adder is a two-level carry save adder. As we will see in §5.3.6, a certain property of the carry delayed adder can be used to reduce the multiplication complexity. The carry delayed adder produced a pair of integers (D, T), called a carry delayed number, using the following set of equations:

$$
\begin{aligned}
S_i &= A_i \oplus B_i \oplus C_i, \\
C_{i+1} &= A_i B_i + A_i C_i + B_i C_i, \\
T_i &= S_i \oplus C_i, \\
D_{i+1} &= S_i C_i,
\end{aligned}
$$

where $D_0 = 0$. Notice that C_{i+1} and S_i are the outputs of a full-adder cell with inputs A_i, B_i, and C_i, while the values D_{i+1} and T_i are the outputs of an half-adder cell.

An important property of the carry delayed adder is that $D_{i+1}T_i = 0$ for all $i = 0, 1, \ldots, k-1$. This is easily verified as

$$D_{i+1}T_i \;=\; S_iC_i(S_i \oplus C_i) \;=\; S_iC_i(\bar{S}_iC_i + S_i\bar{C}_i) \;=\; 0.$$

As an example, let $A = 40$, $B = 25$, and $C = 20$. In the first level, we compute the carry save pair (C, S) using the carry save equations. In the second level, we compute the carry delayed pair (D, T) using the definitions $D_{i+1} = S_iC_i$ and $T_i = S_i \oplus C_i$ as

$$\begin{array}{rr}
A = 40 = & 1\,0\,1\,0\,0\,0 \\
B = 25 = & 0\,1\,1\,0\,0\,1 \\
\hline
C = 20 = & 0\,1\,0\,1\,0\,0 \\
\hline
S = 37 = & 1\,0\,0\,1\,0\,1 \\
C = 48 = & 0\,1\,1\,0\,0\,0\,0 \\
\hline
T = 21 = & 0\,1\,0\,1\,0\,1 \\
\hline
D = 64 = & 1\,0\,0\,0\,0\,0\,0
\end{array}$$

Thus, the carry delayed pair $(64, 21)$ represents the total of $A + B + C = 85$. The property of the carry delayed pair that $T_iD_{i+1} = 0$ for all $i = 0, 1, \ldots, k-1$ also holds.

$$\begin{array}{rr}
T = 21 = & 0\,1\,0\,1\,0\,1 \\
D = 64 = & 1\,0\,0\,0\,0\,0\,0 \\
\hline
T_iD_{i+1} = & 0\,0\,0\,0\,0\,0
\end{array}$$

We will explore this property in § 5.3.6 to design an efficient modular multiplier which was introduced by Brickell [33]. Fig. 5.7 illustrates the carry delayed adder for $k = 6$.

5.2 Modular Addition Operation

The modular addition problem is defined as the computation of $S = A + B \pmod{n}$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \leq A, B < n$, i.e., they are least positives residues. The most common method of computing S is as follows:

1. First compute $S' = A + B$.
2. Then compute $S'' = S' - n$.
3. If $S'' \geq 0$, then $S = S'$ else $S = S''$.

Thus, in addition to the availability of a regular adder, we need fast sign detection which is easy for the CPA, but somewhat harder for the CSA. However, when a CSA is used, the first two steps of the above algorithm can be

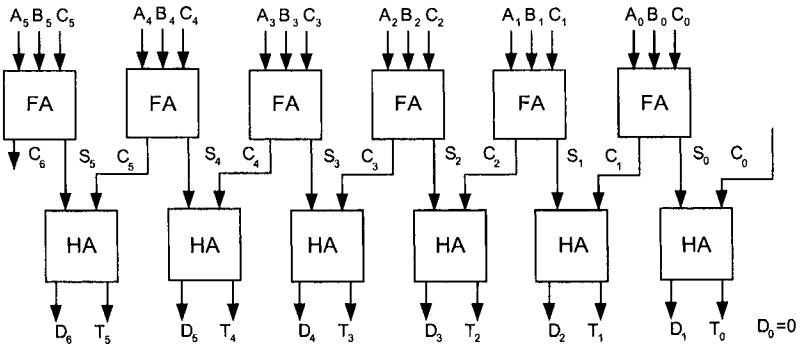

Fig. 5.7. Carry Delayed Adder

combined, in other words, $S' = A + B$ and $S'' = A + B - n$ can be computed at the same time. Then, we perform a sign detection to decide whether to take S' or S'' as the correct sum. We will review algorithms of this type when we study modular multiplication algorithms.

5.2.1 Omura's Method

An efficient method computing the modular addition, which especially useful for multioperand modular addition was proposed by Omura in [260]. Let $n < 2^k$. This method allows a temporary value to grow larger than n, however, it is always kept less than 2^k. Whenever it exceeds 2^k, the carry-out is ignored and a correction is performed. The correction factor is $m = 2^k - n$, which is precomputed and saved in a register. Thus, Omura's method performs the following steps given the integers $A, B < 2^k$ (but they can be larger than n).

1. First compute $S' = A + B$.
2. If there is a carry-out (of the kth bit), then $S = S' + m$, else $S = S'$.

The correctness of Omura's algorithm follows from the observations that

- If there is no carry-out, then $S = A + B$ is returned. The sum S is less than 2^k, but may be larger than n. In a future computation, it will be brought below n if necessary.
- If there is a carry-out, then we ignore the carry-out, which means we compute

  $$S' = A + B - 2^k.$$

  The result, which needs to be reduced modulo n, is in effect reduced modulo 2^k. We correct the result by adding m back to it, and thus, compute

$$\begin{aligned} S &= S' + m \\ &= A + B - 2^k + m \\ &= A + B - 2^k + 2^k - n \\ &= A + B - n. \end{aligned}$$

After all additions are completed, a final result is reduced modulo n by using the standard technique. As an example, let assume $n = 39$. Thus, we have $m = 2^6 - 39 = 25 = (011001)$. The modular addition of $A = 40$ and $B = 30$ is performed using Omura's method as follows:

$$\begin{aligned} A &= 40 = (101000) \\ B &= 30 = (011110) \\ S' &= A + B = 1(000110) \text{ Carry-out} \\ m &= (011001) \\ S &= S' + m = (011111) \text{ Correction} \end{aligned}$$

Thus, we obtain the result as $S = (011111) = 31$ which is equal to 70 (mod 39) as required. On the other hand, the addition of $A = 23$ by $B = 26$ is performed as

$$\begin{aligned} A &= 23 = (010111) \\ B &= 26 = (011010) \\ S' &= A + B = 0(110001) \text{ No carry-out} \\ S &= S' = (110001) \end{aligned}$$

This leaves the result as $S = (110001) = 49$ which is larger than the modulus 39. It will be reduced in a further step of the multioperand modulo addition. After all additions are completed, a final negative result can be corrected by adding m to it. For example, we correct the above result $S = (110001)$ as follows:

$$\begin{aligned} S &= (110001) \\ m &= (011001) \\ S &= S + m = 1(001010) \\ S &= (001010) \end{aligned}$$

The result obtained is $S = (001010) = 10$, which is equal to 49 modulo 39, as required.

5.3 Modular Multiplication Operation

The modular multiplication problem is defined as the computation of $P = AB \pmod{n}$ given the integers A, B, and n. It is usually assumed that A and B are positive integers with $0 \leq A, B < n$, i.e., they are the least positive residues. There are basically four approaches for computing the product P.

- Multiply and then divide.
- The steps of the multiplication and reduction are interleaved.

- Brickell's method.
- Montgomery's method.

The multiply-and-divide method first multiplies A and B to obtain the $2k$-bit number

$$P' := AB.$$

Then, the result P' is divided (reduced) by n to obtain the k-bit number

$$P := P' \mod n.$$

The result P is a k-bit or s-word number.

The reduction is accomplished by dividing P' by n, however, we are not interested in the quotient; we only need the remainder. The steps of the division algorithm can be somewhat simplified in order to speed up the process.

5.3.1 Standard Multiplication Algorithm

Let A and B be two s-digit (s-word) numbers expressed in radix W as:

$$A = (A_{s-1}A_{s-2}\cdots A_0) = \sum_{j=0}^{s-1} A_i W^i,$$

$$B = (B_{s-1}B_{s-2}\cdots B_0) = \sum_{j=0}^{s-1} B_i W^i,$$

where the digits of A and B are in the range $[0, W-1]$. In general W can be any positive number. For reconfigurable hardware implementations, we often select $W = 2^w$ where w is the word-size or granularity of the device, e.g., $w = 4$. The standard (pencil-and-paper) algorithm for multiplying A and B produces the partial products by multiplying a digit of the multiplier (B) by the entire number A, and then summing these partial products to obtain the final number $2s$-word number P'. Let P'_{ij} denote the (Carry,Sum) pair produced from the product $A_i \cdot B_j$. For example, when $W = 10$, and $A_i = 7$ and $B_j = 8$, then $P'_{ij} = (5, 6)$. The P'_{ij} pairs can be arranged in a table as

$$\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 \\
\times & & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & P'_{03} & P'_{02} & P'_{01} & P'_{00} \\
 & & P'_{13} & P'_{12} & P'_{11} & P'_{10} & \\
 & P'_{23} & P'_{22} & P'_{21} & P'_{20} & & \\
+ & P'_{33} & P'_{32} & P'_{31} & P'_{30} & & & \\
\hline
P'_7 & P'_6 & P'_5 & P'_4 & P'_3 & P'_2 & P'_1 & P'_0
\end{array}$$

The last row denotes the total sum of the partial products, and represents the product as an $2s$-word number. The standard algorithm for multiplication essentially performs the above digit-by-digit multiplications and additions. In

order to save space, a single partial product variable P' is being used. The initial value of the partial product is equal to zero; we then take a digit of B and multiply by the entire number A, and add it to the partial product P'. The partial product variable P' contains the final product $A \cdot B$ at the end of the computation. Algorithm 5.1 shows the standard procedure for computing the product $A \cdot B$.

Algorithm 5.1 The Standard Multiplication Algorithm

Require: A, B.
Ensure: $P' = A \cdot B$.
1: Initially $P'_i := 0$ for all $i = 0, 1, \ldots, 2s - 1$.
2: **for** $i = 0$ **to** $s - 1$ **do**
3: $C := 0$;
4: **for** $j = 0$ **to** $s - 1$ **do**
5: $(C, S) := P'_{i+j} + A_j \cdot B_i + C$;
6: $P'_{i+j} := S$;
7: **end for**
8: $P'_{i+s} := C$;
9: **end for**
10: **Return**$(P'_{2s-1} P'_{2s-2} \cdots P'_0)$

In the following, we show the steps of the computation of $A \cdot B = 348 \cdot 857$ using the standard algorithm.

i	j	Step	(C,S)	Partial P'
0	0	$P'_0 + A_0b_0 + C$	$(0,*)$	000000
		$0 + 8 \cdot 7 + 0$	$(5,6)$	000006
	1	$P'_1 + A_1b_0 + C$		
		$0 + 4 \cdot 7 + 5$	$(3,3)$	000036
	2	$P'_2 + A_2b_0 + C$		
		$0 + 3 \cdot 7 + 3$	$(2,4)$	000436
				002436
1	0	$P'_1 + A_0b_1 + C$	$(0,*)$	
		$3 + 8 \cdot 5 + 0$	$(4,3)$	002436
	1	$P'_2 + A_1b_1 + C$		
		$4 + 4 \cdot 5 + 4$	$(2,8)$	002836
	2	$P'_3 + A_2b_1 + C$		
		$2 + 3 \cdot 5 + 2$	$(1,9)$	009836
				019836
2	0	$P'_2 + A_0b_2 + C$	$(0,*)$	
		$8 + 8 \cdot 8 + 0$	$(7,2)$	019236
	1	$P'_3 + A_1b_2 + C$		
		$9 + 4 \cdot 8 + 7$	$(4,8)$	018236
	2	$P'_4 + A_2b_2 + C$		
		$1 + 3 \cdot 8 + 4$	$(2,9)$	098236
				298236

In order to implement this algorithm, we need to be able to execute Step 5 of Algorithm 5.1 as,

$$(C,S) := P'_{i+j} + A_j \cdot B_i + C,$$

where the variables P'_{i+j}, A_j, B_i, C, and S each hold a single-word, or a W-bit number. This step is termed as an inner-product operation which is common in many of the arithmetic and number-theoretic calculations. The inner-product operation above requires that we multiply two W-bit numbers and add this product to previous 'carry' which is also a W-bit number and then add this result to the running partial product word P'_{i+j}. From these three operations we obtain a $2W$-bit number since the maximum value is

$$2^W - 1 + (2^W - 1)(2^W - 1) + 2^W - 1 = 2^{2W} - 1.$$

Also, since the inner-product step is within the innermost loop, it needs to run as fast as possible. Of course, the best thing is to have a single microprocessor instruction for this computation; unfortunately, none of the currently available microprocessors and signal processors offers such a luxury. A brief inspection of the steps of this algorithm reveals that the total number of inner-product steps is equal to s^2. Since $s = k/w$ and w is a constant on a given computer, the standard multiplication algorithm requires $O(k^2)$ bit operations in order to multiply two k-bit numbers.

5.3.2 Squaring is Easier

Squaring is an easier operation than multiplication since half of the single-precision multiplications can be skipped. This is due to the fact that $P'_{ij} = A_i \cdot A_j = P'_{ji}$.

$$
\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
\times & & & & A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & P'_{03} & P'_{02} & P'_{01} & P'_{00} \\
 & & & P'_{13} & P'_{12} & P'_{11} & P'_{01} & \\
 & & P'_{23} & P'_{22} & P'_{12} & P'_{02} & & \\
+ & P'_{33} & P'_{23} & P'_{13} & P'_{03} & & & \\
\hline
 & & & & 2P'_{03} & 2P'_{02} & 2P'_{01} & P'_{00} \\
 & & & 2P'_{13} & 2P'_{12} & P'_{11} & & \\
 & & 2P'_{23} & P'_{22} & & & & \\
+ & P'_{33} & & & & & & \\
\hline
P'_7 & P'_6 & P'_5 & P'_4 & P'_3 & P'_2 & P'_1 & P'_0
\end{array}
$$

Thus, we can modify the standard multiplication procedure as shown in Algorithm 5.2 to take advantage of this property of the squaring operation.

Algorithm 5.2 The Standard Squaring Algorithm

Require: A.
Ensure: $P' = A \cdot A$.
1: Initially $P'_i := 0$ for all $i = 0, 1, \ldots, 2s-1$.
2: **for** $i = 0$ **to** $s-1$ **do**
3: $(C, S) := P'_{i+i} + A_i \cdot A_i$
4: **for** $j = i+1$ **to** $s-1$ **do**
5: $(C, S) := P'_{i+j} + 2 \cdot A_j \cdot A_i + C$;
6: $P'_{i+j} := S$;
7: **end for**
8: $P'_{i+s} := C$;
9: **end for**
10: **Return**$(P'_{2s-1} P'_{2s-2} \cdots P'_0)$

However, we warn the reader that the carry-sum pair produced by operation

$$(C, S) := P'_{i+j} + 2 \cdot A_j \cdot A_i + C$$

in Step 5 of Algorithm 5.2 may be 1 bit longer than a single-precision number which requires w bits. Since

$$(2^w - 1) + 2(2^w - 1)(2^w - 1) + (2^w - 1) = 2^{2w+1} - 2^{w+1}$$

and

$$2^{2w} - 1 < 2^{2w+1} - 2^{w+1} < 2^{2w+1} - 1,$$

the carry-sum pair requires $2w+1$ bits instead of $2w$ bits for its representation. Thus, we need to accommodate this 'extra' bit during the execution of the operations in Steps 5, 6, and 7 of Algorithm 5.2. The resolution of this carry may depend on the way the carry bits are handled by the particular processor's architecture. This issue, being rather implementation-dependent, will not be discussed here.

5.3.3 Modular Reduction

The multiply-and-reduce modular multiplication algorithm first computes the product $A \cdot B$ (or, $A \cdot A$) using one of the multiplication algorithms given above. The multiplication step is then followed by a division algorithm in order to compute the remainder. However, as we have mentioned before, we are not interested in the quotient; we only need the remainder. Therefore, the steps of the division algorithm can somewhat be simplified in order to speed up the process. The reduction step can be achieved by making one of the well-known sequential division algorithms. In the rest of this subsection, we describe the restoring and the nonrestoring division algorithms for computing the remainder of P' when divided by n, where n is a general modulus[1].

Division is the most complex of the four basic arithmetic operations. First of all, it has two results: the quotient and the remainder. Given a dividend P' and a divisor n, a quotient Q and a remainder R have to be calculated in order to satisfy

$$P' = Q \cdot n + R \text{ with } R < n.$$

If P' and n are positive, then the quotient Q and the remainder R will be positive. The sequential division algorithm successively shifts and subtracts n from P' until a remainder R with the property $0 \leq R < n$ is found. However, after a subtraction we may obtain a negative remainder. The restoring and nonrestoring algorithms take different actions when a negative remainder is obtained.

Restoring Division Algorithm

Let R_i be the remainder obtained during the ith step of the division algorithm. Since we are not interested in the quotient, we ignore the generation of the bits of the quotient in the following algorithm. The procedure given below first left-aligns the operands P' and n. Since P' is $2k$-bit number and n is a k-bit number, the left alignment implies that n is shifted k bits to the left, i.e., we start with $2^k n$. Furthermore, the initial value of R is taken to be P', i.e., $R_0 = P'$. We then subtract the shifted n from P' to obtain R_1; if R_1 is

[1] It is noted that Solinas proposed in [338] primes of special form for which the reduction step can be accomplished with high efficiency. However the material for Solinas special primes is not covered in this book. The interested reader may consult [37].

positive or zero, we continue to the next step. If it is negative the remainder is restored to its previous value as is shown in Algorithm 5.3 below.

Algorithm 5.3 The Restoring Division Algorithm

Require: P', n.
Ensure: $R = P' \bmod n$.
1: $R_0 := t$;
2: $n := 2^k n$;
3: **for** $i = 1$ **to** k **do**
4: $\quad R_i := R_{i-1} n$;
5: $\quad$ **if** $R_i < 0$ **then**
6: $\quad\quad R_i := R_{i-1}$;
7: $\quad$ **end if**
8: $\quad n := n/2$
9: **end for**
10: **Return**(R_k)

In Step 5 of Algorithm 5.3, we check the sign of the remainder; if it is negative, the previous remainder is taken to be the new remainder, i.e., a restore operation is performed. If the remainder R_i is positive, it remains as the new remainder, i.e., we do not restore. The restoring division algorithm performs k subtractions in order to reduce the $2k$-bit number t modulo the k-bit number n. Thus, it takes much longer than the standard multiplication algorithm which requires $s = k/w$ inner-product steps, where w is the word-size of granularity being employed.

In the following, we give an example of the restoring division algorithm for computing 3019 mod 53, where $3019 = (101111001011)_2$ and $53 = (110101)_2$. The result is $51 = (110011)_2$.

R_0	101111 001011	t
n	110101	subtract
−	000110	negative remainder
R_1	101111 001011	restore
$n/2$	11010 1	shift and subtract
+	10100 1	positive remainder
R_2	10100 101011	not restore
$n/2$	1101 01	shift and subtract
+	0111 01	positive remainder
R_3	0111 011011	not restore
$n/2$	110 101	shift and subtract
+	000 110	positive remainder
R_4	000 110011	not restore
$n/2$	11 0101	shift
$n/2$	1 10101	shift
$n/2$	110101	shift and subtract
+	000010	negative remainder
R_5	110011	restore
R	110011	final remainder

Also, before subtracting, we may check if the most significant bit of the remainder is 1. In this case, we perform a subtraction. If it is zero, there is no need to subtract since $n > R_i$. We shift n until it is aligned with a nonzero most significant bit of R_i. This way we are able to skip several subtract/restore cycles. In the average, $k/2$ subtractions are performed.

Nonrestoring Division Algorithm

The nonrestoring division algorithm allows a negative remainder. In order to correct the remainder, a subtraction or an addition is performed during the next cycle, depending on the whether the sign of the remainder is positive or negative, respectively. This is based on the following observation: Suppose $R_i = R_{i-1} - n < 0$, then the restoring algorithm assigns $R_i := R_{i-1}$ and performs a subtraction with the shifted n, obtaining

$$R_{i+1} = R_i - n/2 = R_{i-1} - n/2.$$

However, if $R_i = R_{i-1} - n < 0$, then one can instead let R_i remain negative and add the shifted n in the following cycle. Thus, one obtains

$$R_{i+1} = R_i + n/2 = (R_{i-1} - n) + n/2 = R_{i-1} - n/2,$$

which would be the same value. The steps of the nonrestoring algorithm, which implements this observation, are given in Algorithm 5.4.

Note that the nonrestoring division algorithm requires a final restoration cycle in which a negative remainder is corrected by adding the last value of n back to it.

Algorithm 5.4 The Nonrestoring Division Algorithm

Require: P', n.
Ensure: $R = P' \bmod n$.
1: $R_0 := t$;
2: $n := 2^k n$;
3: **for** $i = 1$ **to** k **do**
4: **if** $R_{i-1} > 0$ **then**
5: $R_i := R_{i-1} - n$;
6: **else**
7: $R_i := R_{i-1} + n$;
8: **end if**
9: $n := n/2$;
10: **if** $R_k < 0$ **then**
11: $R := R + n$;
12: **end if**
13: **end for**
14: **Return**(R_k)

In the following we compute $51 = 3019 \bmod 53$ using the nonrestoring division algorithm. Since the remainder is allowed to stay negative, we use 2's complement coding to represent such numbers.

R_0	0101111 001011 t	
n	0110101	subtract
R_1	1111010	negative remainder
$n/2$	011010 1	add
R_2	010100 1	positive remainder
$n/2$	01101 01	subtract
R_3	00111 01	positive remainder
$n/2$	0110 101	subtract
R_4	0000 110	positive remainder
$n/2$	011 0101	
$n/2$	01 10101	
$n/2$	0 110101	subtract
R_5	1 111110	negative remainder
n	0 110101	add (final restore)
R	0 110011	Final remainder

5.3.4 Interleaving Multiplication and Reduction

The interleaving algorithm has been known. The details of the method are sketched in papers [27, 334]. Let A_i and B_i be the bits of the k-bit positive integers A and B, respectively. The product P' can be written as

$$P' = A \cdot B = A \cdot \sum_{i=0}^{k-1} B_i 2^i = \sum_{i=0}^{k-1} (A \cdot B_i) 2^i$$

$$= 2(\cdots 2(2(0 + A \cdot B_{k-1}) + A \cdot B_{k-2}) + \cdots) + A \cdot B_0$$

This formulation yields the shift-add multiplication algorithm. Notice that we also reduce the partial product modulo n at each step of Algorithm 5.5.

Algorithm 5.5 The Interleaving Multiplication Algorithm

Require: A, B, n.
Ensure: $P = A \cdot B \bmod n$.
1: $P := 0$;
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $\quad P := 2P + A \cdot B_{k-1-i}$;
4: $\quad P := P \bmod n$;
5: **end for**
6: **Return**(P)

Assuming that $A, B, P < n$, we have

$$\begin{aligned} P &:= 2P + A \cdot B_j \\ &\leq 2(n-1) + (n-1) \;=\; 3n - 3. \end{aligned}$$

Thus, the new P will be in the range $0 \leq P \leq 3n - 3$, and at most 2 subtractions are needed to reduce P to the range $0 \leq P < n$. We can use the following algorithm to bring P back to this range:

$$P' := P - n \text{ ; If } P' \geq 0 \text{ then } P = P'$$
$$P' := P - n \text{ ; If } P' \geq 0 \text{ then } P = P'$$

The computation of P requires k steps, at each step we perform the following operations:

- A left shift: $2P$
- A partial product generation: $A \cdot B_j$
- An addition: $P := 2P + A \cdot B_j$
- At most 2 subtractions:
  $P' := P - n$; If $P' \geq 0$ then $P = P'$
  $P' := P - n$; If $P' \geq 0$ then $P = P'$

The left shift operation is easily performed by wiring. The partial products, on the other hand, are generated using an array of AND gates. The most crucial operations are the addition and subtraction operations: they need to be performed fast. We have the following avenues to explore:

- We can use the carry propagate adder, introducing $O(k)$ delay per step. However, Omura's method can be used to avoid unnecessary subtractions:

3a. $P := 2P$
3b. If carry-out then $P := P + m$
3c. $P := P + A \cdot B_j$
3d. If carry-out then $P := P + m$

- We can use the carry save adder, introducing only $O(1)$ delay per step. However, recall that the sign information is not immediately available in the CSA. We need to perform fast sign detection in order to determine whether the partial product needs to be reduced modulo n.

5.3.5 Utilization of Carry Save Adders

In order to utilize the carry save adders in performing the modular multiplication operations, we represent the numbers as the carry save pairs (C, S), where the value of the number is the sum $C+S$. The carry save adder method of the interleaving algorithm is given in Algorithm 5.6.

Algorithm 5.6 The Carry-Save Interleaving Multiplication Algorithm

Require: A, B, n.
Ensure: $P = A \cdot B \bmod n$.
1: $(C, S) := (0, 0)$;
2: **for** $i = 0$ **to** $k - 1$ **do**
3: $(C, S) := 2C + 2S + A \cdot B_{k-1-i}$;
4: $(C', S') := C + Sn$;
5: **if** SIGN ≥ 0 **then**
6: $(C, S) := (C', S')$;
7: **end if**
8: **end for**
9: **Return**(C, S)

The function SIGN gives the sign of the carry save number $C' + S'$. Since the exact sign is available only when a full addition is performed, we calculate an estimated sign with the SIGN function. A sign estimation algorithm was introduced in [185]. Here, we briefly review this algorithm, which is based on the addition of the most significant t bits of C and S to estimate the sign of $C + S$. For example, let $C = (011110)$ and $S = (001010)$, then the function SIGN produces

$$
\begin{aligned}
C &= 011110 \\
S &= 001010 \\
(t = 1)\ &\text{SIGN} = \underline{0} \\
(t = 2)\ &\text{SIGN} = \underline{0}1 \\
(t = 3)\ &\text{SIGN} = \underline{1}00 \\
(t = 4)\ &\text{SIGN} = \underline{1}001 \\
(t = 5)\ &\text{SIGN} = \underline{1}0100 \\
(t = 6)\ &\text{SIGN} = \underline{1}01000.
\end{aligned}
$$

In the worst case the exact sign is produced after adding all k bits. If the exact sign of $C+S$ is computed, we can obtain the result of the multiplication operation in the correct range $[0, n)$. If an estimation of the sign is used, then we will prove that the range of the result becomes $[0, n+\Delta)$, where Δ depends on the precision of the estimation. Furthermore, since the sign is used to decide whether some multiple of n should be subtracted from the partial product, an error in the decision causes only an error of a multiple of n in the partial product, which is corrected later. We define function $T(X)$ on an n-bit integer X as

$$T(X) = X - (X \bmod 2^t),$$

where $0 \leq t \leq n-1$. In other words, T replaces the first least significant t bits of X with t zeros. This implies

$$T(X) \leq X < T(X) + 2^t.$$

We reduce the pair (C, S) by performing the following operation Q times:

I. $(\hat{C}, \hat{S}) := C + S - n$.

J. If $T(\hat{C}) + T(\hat{S}) \geq 0$ then set $C := \hat{C}$ and $S := \hat{S}$.

In Step J, the computation of the sign bit R of $T(\hat{C}) + T(\hat{S})$ involves $n - t$ most significant bits of $\hat{C}$ and $\hat{S}$. The above procedure reduces a carry-sum pair from the range

$$0 \leq C_0 + S_0 < (Q+1)n + 2^t$$

to the range

$$0 \leq C_R + S_R < n + 2^t,$$

where (C_0, S_0) and (C_R, S_R) respectively denote the initial and the final carry-sum pair. Since the function T always underestimates, the result is never over-reduced, i.e.,

$$C_R + S_R \geq 0.$$

If the estimated sign in Step J is positive for all Q iterations, then QN is subtracted from the initial pair; therefore

$$C_R + S_R = C_0 + S_0 - QN < n + 2^t.$$

If the estimated sign becomes negative in an iteration, it stays negative thereafter to the last iteration. Thus, the condition

$$T(\hat{C}) + T(\hat{S}) < 0$$

in the last iteration of Step J implies that

$$T(\hat{C}) + T(\hat{S}) \leq -2^t,$$

since $T(X)$ is always a multiple of 2^t. Thus, we obtain the range of $\hat{C}$ and $\hat{S}$ as

$$T(\hat{C}) + T(\hat{S}) \leq \hat{C} + \hat{S} < T(\hat{C}) + T(\hat{S}) + 2^{t+1}.$$

It follows from the above equations that

$$\hat{C} + \hat{S} < 2^{t+1} - 2^t = 2^t.$$

Since in Step I we perform $(\hat{C}, \hat{S}) := C + S - n$ and in the last iteration the carry-sum pair is not reduced (because the estimated sign is negative), we must have

$$C_R + S_R = \hat{C} + \hat{S} + n,$$

which implies

$$C_R + S_R < n + 2^t.$$

The modular reduction procedure described above subtracts n from (C, S) in each of the Q iterations. The procedure can be improved in speed by subtracting $2^{k-j}n$ during iteration j, where $(Q+1) \leq 2^k$ and $j = 1, 2, 3, \ldots, k$. For example, if $Q = 3$, then $k = 2$ can be used. Instead of subtracting n three times, we first subtract $2N$ and then n. This observation is utilized in Algorithm 5.7.

The parameter t controls the precision of estimation; the accuracy of the estimation and the total amount of logic required to implement it decreases as t increases. After Step 7 of Algorithm 5.7, we have

$$C^{(i)} + S^{(i)} < n + 2^t,$$

which implies that after the next shift-add step the range of $C^{(i+1)} + S^{(i+1)}$ will be $[0, 3N + 2^{t+1})$. Assuming $Q = 3$, we have

$$3N + 2^{t+1} \leq (Q+1)n + 2^t = 4N + 2^t,$$

which implies $2^t \leq n$, or $t \leq n - 1$. The range of $C^{(i+1)} + S^{(i+1)}$ becomes

$$0 \leq C^{(i+1)} + S^{(i+1)} < 3N + 2^{t+1} \leq 3N + 2^n \leq 2^{n+2},$$

and after Step 4 of Algorithm 5.7, the range will be

Algorithm 5.7 The Carry-Save Interleaving Multiplication Algorithm Revisited

Require: A, B, n.
Ensure: $P = A \cdot B \bmod n$.
1: Set $S^{(0)} := 0$ and $C^{(0)} := 0$.
2: **for** $i = 1$ **to** k **do**
3: $(C^{(i)}, S^{(i)}) := 2C^{(i-1)} + 2S^{(i-1)} + A_{n-i}B$;
4: $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - 2n$;
5: **if** $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$ **then**
6: $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$.
7: **end if**
8: $(\hat{C}^{(i)}, \hat{S}^{(i)}) := C^{(i)} + S^{(i)} - n$;
9: **if** $T(\hat{C}^{(i)}) + T(\hat{S}^{(i)}) \geq 0$ **then**
10: $C^{(i)} := \hat{C}^{(i)}$ and $S^{(i)} := \hat{S}^{(i)}$;
11: **end if**
12: **end for**
13: **Return**$(C^{(i)}, S^{(i)})$

$$-2^{n+1} \leq -2N \leq C^{(i+1)} + S^{(i+1)} < n + 2^n < 2^{n+1}.$$

In order to contain the temporary results, we use $(n+3)$-bit carry save adders which can represent integers in the range $[-2^{n+2}, 2^{n+2})$. When $t = n - 1$, the sign estimation technique checks 5 most significant bits of $\hat{C}^{(i)}$ and $\hat{S}^{(i)}$ from the bit locations $n - 2$ to $n + 3$. This algorithm produces a pair of integers $(C, S) = (C^{(n)}, S^{(n)})$ such that $P = C + S$ is in the range $[0, 2N)$. The final result in the correct range $[0, n)$ can be obtained by computing $P = C + S$ and $\hat{P} = C + S - n$ using carry propagate adders. If $\hat{P} < 0$, we have $P = \hat{P} + n < n$, and thus P is in the correct range. Otherwise, we choose $\hat{P}$ because $0 \leq \hat{P} = P - n < 2^t < n$ implies $\hat{P} \in [0, n)$. The steps of the algorithm for computing $47 \cdot 48 \pmod{50}$, are illustrated in the following figure. Here we have

$$\begin{aligned} k = \lfloor \log_2(50) \rfloor + 1 &= 6, \\ A = 47 &= (000101111), \\ B = 48 &= (000110000), \\ n = 50 &= (000110010), \\ M = -n &= (111001110). \end{aligned}$$

The algorithm computes the final result

$$(C, S) = (010111000, 110000000) = (184, -128)$$

in $3k = 18$ clock cycles. The range of $C + S = 184 - 128 = 56$ is $[0, 2 \cdot 50)$. The final result is found by computing $C + S = 56$ and $C + S - n = 6$, and selecting the latter since it is positive.

		C	S	$\hat{C}$	$\hat{S}$	$T(\hat{C})+T(\hat{S})$	R
$i=0$		000000000	000000000	–	–	–	–
$i=1$	2a	000000000	000110000	–	–	–	–
	2b	000000000	000110000	000100000	110101100	111000000	1
	2c	000000000	000110000	000000000	111111110	111100000	1
$i=2$	2a	000000000	001100000	–	–	–	–
	2b	000000000	001100000	000000000	111111100	111100000	1
	2c	010000000	110101110	010000000	110101110	000100000	0
$i=3$	2a	000100000	001101100	–	–	–	–
	2b	001011000	111010000	001011000	111010000	000000000	0
	2c	001011000	111010000	110110000	001000110	111100000	1
$i=4$	2a	101100000	100100000	–	–	–	–
	2b	001000000	111011100	001000000	111011100	000000000	0
	2c	001000000	111011100	110011000	001010010	111000000	1
$i=5$	2a	101100000	100001000	–	–	–	–
	2b	101100000	100001000	000010000	111110100	111100000	1
	2c	010010000	110100110	010010000	110100110	000100000	0
$i=6$	2a	001000000	001011100	–	–	–	–
	2b	010111000	110000000	010111000	110000000	000100000	0
	2c	010111000	110000000	100010000	011110110	111100000	1

5.3.6 Brickell's Method

This method is based on the use of a carry delayed integer introduced in §5.1.6. Let A be a carry delayed integer, then, it can be written as

$$A = \sum_{i=0}^{k-1} (T_i + D_i) \cdot 2^i.$$

The product $P = AB$ can be computed by summing the terms:

$$\begin{gathered}(T_0 \cdot B + D_0 \cdot B) \cdot 2^0 + \\ (T_1 \cdot B + D_1 \cdot B) \cdot 2^1 + \\ (T_2 \cdot B + D_2 \cdot B) \cdot 2^2 + \\ \vdots \\ (T_{k-1} \cdot B + D_{k-1} \cdot B) \cdot 2^{k-1}\end{gathered}$$

Since $D_0 = 0$, we rearrange to obtain

$$\begin{gathered}2^0 \cdot T_0 \cdot B + 2^1 \cdot D_1 \cdot B + \\ 2^1 \cdot T_1 \cdot B + 2^2 \cdot D_2 \cdot B + \\ 2^2 \cdot T_2 \cdot B + 2^3 \cdot D_3 \cdot B + \\ \vdots \\ 2^{k-2} \cdot T_{k-2} \cdot B + 2^{k-1} \cdot D_{k-1} \cdot B + \\ 2^{k-1} \cdot T_{k-1} \cdot B\end{gathered}$$

Also recall that either T_i or D_{i+1} is zero due to the property of the carry delayed adder. Thus, each step requires a shift of B and addition of at most 2 carry delayed integers:

- Either: $(P_d, P_t) := (P_d, P_t) + 2^i \cdot T_i \cdot B$
- Or: $(P_d, P_t) := (P_d, P_t) + 2^{i+1} \cdot D_{i+1} \cdot B$

After k steps $P = (P_d, P_t)$ is obtained. In order to compute $P \pmod{n}$, we perform reduction:

$$\begin{aligned}
&\text{If} \quad P \geq 2^{k-1} \cdot n \text{ then} \quad P := P - 2^{k-1} \cdot n \\
&\text{If} \quad P \geq 2^{k-2} \cdot n \text{ then} \quad P := P - 2^{k-2} \cdot n \\
&\text{If} \quad P \geq 2^{k-3} \cdot n \text{ then} \quad P := P - 2^{k-3} \cdot n \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\text{If} \quad P \geq n \text{ then} \quad P := P - n
\end{aligned}$$

We can also reverse these steps to obtain:

$$\begin{aligned}
P &:= T_{k-1} \cdot B \cdot 2^{k-1} \\
P &:= P + T_{k-2} \cdot B \cdot 2^{k-2} + D_{k-1} \cdot B \cdot 2^{k-1} \\
P &:= P + T_{k-3} \cdot B \cdot 2^{k-3} + D_{k-2} \cdot B \cdot 2^{k-2} \\
&\vdots \\
P &:= P + T_1 \cdot B \cdot 2^1 + D_2 \cdot B \cdot 2^2 \\
P &:= P + T_0 \cdot B \cdot 2^0 + D_1 \cdot B \cdot 2^1
\end{aligned}$$

Also, the multiplication steps can be interleaved with reduction steps. To perform the reduction, the sign of $P - 2^i \cdot n$ needs to be determined (estimated). Brickell's solution [33] is essentially a combination of the sign estimation technique and Omura's method of correction. We allow enough bits for P, and whenever P exceeds 2^k, add $m = 2^k - n$ to correct the result. 11 steps after the multiplication procedure started, the algorithm starts subtracting multiples of n. In the following, P is a carry delayed integer of $k + 11$ bits, m is a binary integer of k bits, and t_1 and t_2 control bits, whose initial values are $t_1 = t_2 = 0$.

1. Add the most significant 4 bits of P and $m \cdot 2^{11}$.
2. If overflow is detected, then $t_2 = 1$ else $t_2 = 0$.
3. Add the most significant 4 bits of P and the most significant 3 bits of $m \cdot 2^{10}$.
4. If overflow is detected and $t_2 = 0$, then $t_1 = 1$ else $t_1 = 0$.

The multiplication and reduction steps of Brickell's algorithm are as follows:

$$\begin{aligned}
B' &:= T_i \cdot B + 2 \cdot D_{i+1} \cdot B \\
m' &:= t_2 \cdot m \cdot 2^{11} + t_1 \cdot m \cdot 2^{10} \\
P &:= 2(P + B' + m') \\
A &:= 2A.
\end{aligned}$$

5.3.7 Montgomery's Method

In 1985, P. L. Montgomery introduced an efficient algorithm [238] for computing $R = A \cdot B \bmod n$ where A, B, and n are k-bit binary numbers. The Montgomery reduction algorithm computes the resulting k-bit number R without performing a division by the modulus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n operation with division by a power of 2. This operation is easily accomplished on a computer since the numbers are represented in binary form. Assuming the modulus n is a k-bit number, i.e., $2^{k-1} \leq n < 2^k$, let r be 2^k. The Montgomery reduction algorithm requires that r and n be relatively prime, i.e., $\gcd(r, n) = \gcd(2^k, n) = 1$. This requirement is satisfied if n is odd. In the following we summarize the basic idea behind the Montgomery reduction algorithm.

Given an integer $A < n$, we define its n-residue with respect to r as

$$\bar{A} = A \cdot r \bmod n.$$

It is straightforward to show that the set

$$\{\, i \cdot r \bmod n \mid 0 \leq i \leq n-1 \,\}$$

is a complete residue system, i.e., it contains all numbers between 0 and $n-1$. Thus, there is a one-to-one correspondence between the numbers in the range 0 and $n-1$ and the numbers in the above set. The Montgomery reduction algorithm exploits this property by introducing a much faster multiplication routine which computes the n-residue of the product of the two integers whose n-residues are given. Given two n-residues $\bar{A}$ and $\bar{B}$, the *Montgomery product* is defined as the n-residue

$$\bar{R} = \bar{A} \cdot \bar{B} \cdot r^{-1} \bmod n$$

where r^{-1} is the inverse of r modulo n, i.e., it is the number with the property

$$r^{-1} \cdot r = 1 \bmod n.$$

The resulting number $\bar{R}$ is indeed the n-residue of the product

$$R = A \cdot B \bmod n$$

since

$$\begin{aligned} \bar{R} &= \bar{A} \cdot \bar{B} \cdot r^{-1} \bmod n \\ &= A \cdot r \cdot B \cdot r \cdot r^{-1} \bmod n \\ &= A \cdot B \cdot r \bmod n. \end{aligned}$$

In order to describe the Montgomery reduction algorithm, we need an additional quantity, n', which is the integer with the property

$$r \cdot r^{-1} - n \cdot n' = 1.$$

The integers r^{-1} and n' can both be computed by the extended Euclidean algorithm [178]. The Montgomery product algorithm, which computes

$$u = \bar{A} \cdot \bar{B} \cdot r^{-1} \pmod{n}$$

given $\bar{A}$ and $\bar{B}$, is given in Algorithm 5.8 below.

Algorithm 5.8 Montgomery Product

Require: $\bar{A}, \bar{B}, r, n$.
Ensure: u=MonPro$(\bar{A}, \bar{B})$=$\bar{A} \cdot \bar{B} \cdot r^{-1} \pmod{n}$.
1: $t := \bar{A} \cdot \bar{B}$;
2: $m := t \cdot n' \bmod r$;
3: $u := (t + m \cdot n)/r$;
4: **if** $u \geq n$ **then**
5: **Return**$(u - n)$
6: **else**
7: **Return**(u)
8: **end if**

The most important feature of the Montgomery product algorithm is that the operations involved are multiplications modulo r and divisions by r, both of which are intrinsically fast operations since r is a power 2. The MonPro Algorithm 5.9 can be used to compute the product of A and B modulo n, provided that n is odd.

Algorithm 5.9 Montgomery Modular Multiplication: Version I

Require: A, B, an odd number n.
Ensure: $x = A \cdot B \pmod{n}$.
1: Compute n' using the extended Euclidean algorithm.
2: $\bar{A} := A \cdot r \bmod n$;
3: $\bar{B} := B \cdot r \bmod n$;
4: $\bar{x} :=$ MonPro$(\bar{A}, \bar{B})$;
5: $x :=$ MonPro$(\bar{x}, 1)$;
6: **Return**(x)

A better algorithm can be given by observing the property

$$\text{MonPro}(\bar{A}, B) = (A \cdot r) \cdot B \cdot r^{-1} = A \cdot B \pmod{n},$$

which modifies Algorithm 5.9 as shown in Algorithm 5.10. However, the preprocessing operations, especially the computation of n', are rather time-consuming.

Algorithm 5.10 Montgomery Modular Multiplication: Version II

Require: A, B, an odd number n.
Ensure: $x = A \cdot B \pmod{n}$.
1: Compute n' using the extended Euclidean algorithm.
2: $\bar{A} := A \cdot r \bmod n$;
3: $x := \text{MonPro}(\bar{A}, B)$;
4: **Return**(x)

Nevertheless, there is an efficient algorithm for computing the single precision integer n'_0. The computation of n'_0 can be performed by a specialized Euclidean algorithm instead of the general extended Euclidean algorithm. Since $r = 2^{sw}$ and

$$r \cdot r^{-1} - n \cdot n' = 1,$$

we take modulo 2^w of the both sides, and obtain

$$-n \cdot n' = 1 \pmod{2^w},$$

or, in other words,

$$n'_0 = -n_0^{-1} \pmod{2^w},$$

where n'_0 and n_0^{-1} are the least significant words (the least significant w bits) of n' and n^{-1}, respectively. In order to compute $-n_0^{-1} \pmod{2^w}$, we use algorithm 5.11 given below which computes $x^{-1} \pmod{2^w}$ for a given odd x.

Algorithm 5.11 Specialized Modular Inverse

Require: an odd number x and w.
Ensure: $y_w = x^{-1} \pmod{2^w}$.
1: $y_1 := 1$;
2: **for** $i = 2$ **to** w **do**
3: **if** $2^{i-1} < x \cdot y_{i-1} \pmod{2^i}$ **then**
4: $y_i := y_{i-1} + 2^{i-1}$;
5: **else**
6: $y_i := y_{i-1}$;
7: **end if**
8: **end for**
9: **Return**(y_w)

The correctness of the algorithm follows from the observation that, at every step i, we have

$$x \cdot y_i = 1 \pmod{2^i}.$$

This algorithm is very efficient, and uses single precision addition and multiplications in order to compute x^{-1}. As an example, we compute $23^{-1} \pmod{64}$ using the above algorithm. Here we have $x = 23$, $w = 6$. The steps of the algorithm, the temporary values, and the final inverse are shown below:

i	2^i	y_{i-1}	$x \cdot y_{i-1} \pmod{2^i}$	2^{i-1}	y_i
2	4	1	$23 \cdot 1 = 3$	2	$1 + 2 = 3$
3	8	3	$23 \cdot 3 = 5$	4	$3 + 4 = 7$
4	16	7	$23 \cdot 7 = 1$	8	7
5	32	7	$23 \cdot 7 = 1$	16	7
6	64	7	$23 \cdot 7 = 33$	32	$7 + 32 = 39$

Thus, we compute $23^{-1} = 39 \pmod{64}$. This is indeed the correct value since

$$23 \cdot 39 = 14 \cdot 64 + 1 = 1 \pmod{64}.$$

Also, at every step i, we have $x \cdot y_i = 1 \pmod{2^i}$, as shown below:

i	$x \cdot y_i \bmod 2^i$
1	$23 \cdot 1 = 1 \bmod 2$
2	$23 \cdot 3 = 1 \bmod 4$
3	$23 \cdot 7 = 1 \bmod 8$
4	$23 \cdot 7 = 1 \bmod 16$
5	$23 \cdot 7 = 1 \bmod 32$
6	$23 \cdot 39 = 1 \bmod 64$

Montgomery Exponentiation

The Montgomery product algorithm is more suitable when several modular multiplications with respect to the same modulus are needed. Such is the case when one needs to compute a modular exponentiation, i.e., the computation of $M^e \bmod n$. Using one of the addition chain algorithms given in §5.4, we replace the exponentiation operation by a series of square and multiplication operations modulo n. This is where the Montgomery product operation finds its best use. In the following we summarize the modular exponentiation operation which makes use of the Montgomery product function MonPro. The exponentiation Algorithm 5.12 below uses the binary method.

Thus, we start with the ordinary residue M and obtain its n-residue $\bar{M}$ using a division-like operation, which can be achieved, for example, by a series of shift and subtract operations. Additionally, Steps 2 and 3 of Algorithm 5.12 require divisions. However, once the preprocessing has been completed, the inner-loop of the binary exponentiation method uses the Montgomery product operations which performs only multiplications modulo 2^k and divisions by 2^k. When the binary method finishes, we obtain the n-residue $\bar{x}$ of the quantity $x = M^e \bmod n$. The ordinary residue number is obtained from the n-residue by executing the MonPro function with arguments $\bar{x}$ and 1. This is easily shown to be correct since

$$\bar{x} = x \cdot r \bmod n$$

immediately implies that

$$x = \bar{x} \cdot r^{-1} \bmod n \;=\; \bar{x} \cdot 1 \cdot r^{-1} \bmod n \;:=\; \text{MonPro}(\bar{x}, 1).$$

Algorithm 5.12 Montgomery Modular Exponentiation

Require: A, B, and odd number n.
Ensure: $x = M^e \pmod{n}$.
1: Compute n' using the extended Euclidean algorithm.
2: $\bar{M} := M \cdot r \bmod n$;
3: $\bar{x} := 1 \cdot r \bmod n$;
4: **for** $i = k - 1$ **down to** 0 **do**
5: $\bar{x} := \text{MonPro}(\bar{x}, \bar{x})$;
6: **if** $e_i = 1$ **then**
7: $\bar{x} := \text{MonPro}(\bar{M}, \bar{x})$;
8: **end if**
9: **end for**
10: $x := \text{MonPro}(\bar{x}, 1)$;
11: **Return**(x)

The resulting algorithm is quite fast as was demonstrated by many researchers and engineers who have implemented it, for example, see [72, 200]. However, this algorithm can be refined and made more efficient, particularly when the numbers involved are multi-precision integers. For example, Dussé and Kaliski [72] gave improved algorithms, including a simple and efficient method for computing n'. We will describe these methods below.

An Example of Exponentiation

Here we show how to compute $x = 7^{10} \bmod 13$ using the Montgomery exponentiation algorithm.

- Since $n = 13$, we take $r = 2^4 = 16 > n$.
- Computation of n':
  Using the extended Euclidean algorithm, we determine that $16 \cdot 9 - 13 \cdot 11 = 1$, thus, $r^{-1} = 9$ and $n' = 11$.
- Computation of $\bar{M}$:
  Since $M = 7$, we have $\bar{M} := M \cdot r \pmod{n} = 7 \cdot 16 \pmod{13} = 8$.
- Computation of $\bar{x}$ for $x = 1$:
  We have $\bar{x} := x \cdot r \pmod{n} = 1 \cdot 16 \pmod{13} = 3$.
- Steps 5 and 7 of the ModExp routine:

e_i	Step 5	Step 7
1	MonPro$(3, 3) = 3$	MonPro$(8, 3) = 8$
0	MonPro$(8, 8) = 4$	
1	MonPro$(4, 4) = 1$	MonPro$(8, 1) = 7$
0	MonPro$(7, 7) = 12$	

  - Computation of MonPro$(3, 3) = 3$:
    $t := 3 \cdot 3 = 9$
    $m := 9 \cdot 11 \pmod{16} = 3$
    $u := (9 + 3 \cdot 13)/16 = 48/16 = 3$
  - Computation of MonPro$(8, 3) = 8$:
    $t := 8 \cdot 3 = 24$
    $m := 24 \cdot 11 \pmod{16} = 8$
    $u := (24 + 8 \cdot 13)/16 = 128/16 = 8$

- Computation of MonPro(8, 8) = 4:
  $t := 8 \cdot 8 = 64$
  $m := 64 \cdot 11 \pmod{16} = 0$
  $u := (64 + 0 \cdot 13)/16 = 64/16 = 4$
- Computation of MonPro(4, 4) = 1:
  $t := 4 \cdot 4 = 16$
  $m := 16 \cdot 11 \pmod{16} = 0$
  $u := (16 + 0 \cdot 13)/16 = 16/16 = 1$
- Computation of MonPro(8, 1) = 7:
  $t := 8 \cdot 1 = 8$
  $m := 8 \cdot 11 \pmod{16} = 8$
  $u := (8 + 8 \cdot 13)/16 = 112/16 = 7$
- Computation of MonPro(7, 7) = 12:
  $t := 7 \cdot 7 = 49$
  $m := 49 \cdot 11 \pmod{16} = 11$
  $u := (49 + 11 \cdot 13)/16 = 192/16 = 12$

- Step 7 of the ModExp routine: $x = \text{MonPro}(12, 1) = 4$
  $t := 12 \cdot 1 = 12$
  $m := 12 \cdot 11 \pmod{16} = 4$
  $u := (12 + 4 \cdot 13)/16 = 64/16 = 4$

Thus, we obtain $x = 4$ as the result of the operation $7^{10} \bmod 13$.

Hardware Implementation of the Montgomery Method

In the rest of this section, we introduce an efficient binary add-shift algorithm for computing MonPro(A, B), and then generalize it to the m-ary method. We take $r = 2^k$, and assume that the number of bits in A or B is less than k. Let $A = (A_{k-1}A_{k-2} \cdots A_0)$ be the binary representation of A. The above product can be written as

$$2^{-k} \cdot (A_{k-1}A_{k-2} \cdots A_0) \cdot B = 2^{-k} \cdot \sum_{i=0}^{k-1} A_i \cdot 2^i \cdot B \pmod{n}.$$

The product $t = (A_0 + A_1 2 + \cdots A_{k-1}2^{k-1}) \cdot B$ can be computed by starting from the most significant bit, and then proceeding to the least significant, as follows:

1. $t := 0$
2. for $i = k - 1$ to 0
2a. $t := t + A_i \cdot B$
2b. $t := 2 \cdot t$

The shift factor 2^{-k} in $2^{-k} \cdot A \cdot B$ reverses the direction of summation. Since

$$2^{-k} \cdot (A_0 + A_1 2 + \cdots A_{k-1}2^{k-1}) = A_{k-1}2^{-1} + A_{k-2}2^{-2} \cdots A_0 2^{-k},$$

we start processing the bits of A from the least significant, and obtain the following binary add-shift algorithm to compute $t = A \cdot B \cdot 2^{-k}$, as shown in Algorithm 5.13.

Procedure 5.13 computes the product $t = 2^{-k} \cdot A \cdot B$, however, we are interested in computing $u = 2^{-k} \cdot A \cdot B \pmod{n}$. This can be achieved by

Algorithm 5.13 Add-and-Shift Montgomery Product

Require: A, B.
Ensure: $t = A \cdot B \cdot 2^{-k}$.
1: $t := 0$;
2: **for** $i = 0$ to $k - 1$ **do**
3: $t := t + A_i \cdot B$;
4: $t := t/2$;
5: **end for**
6: **Return**(t)

subtracting n during every add-shift step, but there is a simpler way: We add n to u if u is odd, making new u an even number since n is always odd. If u is even after the addition step, it is left untouched. Thus, u will always be even before the shift step, and we can compute

$$u := u \cdot 2^{-1} \pmod{n}$$

by shifting the even number u to the right since $u = 2v$ implies

$$u := 2v \cdot 2^{-1} = v \pmod{n}.$$

The binary add-shift algorithm computes the product $u = A \cdot B \cdot 2^{-k} \pmod{n}$ as shown in Algorithm 5.14.

Algorithm 5.14 Binary Add-and-Shift Montgomery Product

Require: A, B, an odd number n.
Ensure: $u = A \cdot B \cdot 2^{-k} \pmod{n}$.
1: $u := 0$;
2: **for** $i = 0$ to $k - 1$ **do**
3: $u := u + A_i \cdot B$;
4: **if** u is odd **then**
5: $u := u + n$;
6: **end if**
7: $u := u/2$;
8: **end for**
9: **Return**(u)

We reserve a $(k+1)$-bit register for u because if u has k bits at beginning of an add-shift step, the addition of $A_i \cdot B$ and n (both of which are k-bit numbers) increases its length to $k + 1$ bits. The right shift operation then brings it back to k bits. After k add-shift steps, we subtract n from u if it is larger than n.

Also note that Steps 2a and 2b of the above algorithm can be combined: We can compute the least significant bit u_0 of u before actually computing the sum in Step 2a. It is given as

$$u_0 := u_0 \oplus (A_i B_0).$$

Thus, we decide whether u is odd prior to performing the full addition operation $u := u + A_i B$. This is the most important property of Montgomery's method. In contrast, the classical modular multiplication algorithms (e.g., the interleaving method) computes the entire sum in order to decide whether a reduction needs to be performed.

5.3.8 High-Radix Interleaving Method

Since the speed for radix 2 multipliers is approaching limits, the use of higher radices is investigated. High-radix operations require fewer clock cycles, however, the cycle time and the required area increases. Let 2^b be the radix. The key operation in computing $P = AB \pmod{n}$ is the computation of an inner-product steps coupled with modular reduction, i.e., the computation of

$$P := 2^b \cdot P + A \cdot B_i - Q \cdot n,$$

where P is the partial product and B_i is the ith digit of B in radix 2^b. The value of Q determines the number of times the modulus n is subtracted from the partial product P in order to reduce it modulo n. We compute Q by dividing the current value of the partial product P by n, which is then multiplied by n and subtracted from the partial product during the next cycle. This implementation is illustrated in Fig. 5.8.

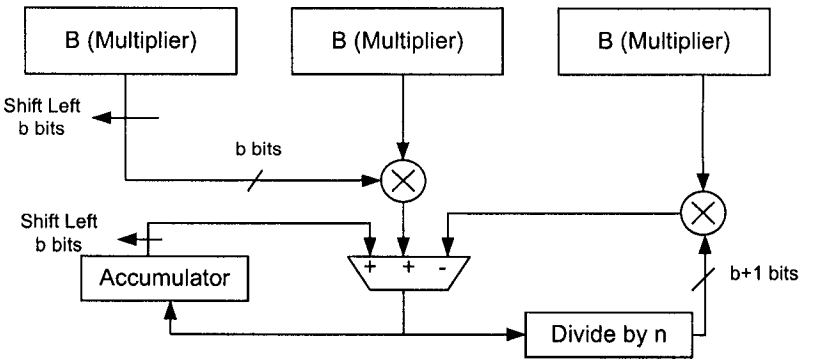

Fig. 5.8. High-Radix Interleaving Method

For the radix 2, the partial product generation is performed using an array of AND gates. The partial product generation is much more complex for higher radices, e.g., Wallace trees and generalized counters need to be used. However, the generation of the high-radix partial products does not greatly increase cycle time since this computation can be easily pipelined. The most complicated

step is the reduction step, which necessitates more complex routing, increasing the chip area.

5.3.9 High-Radix Montgomery's Method

The binary add-shift algorithm is generalized to higher radix (m-ary) algorithm by proceeding word by word, where the wordsize is w bits, and $k = sw$. The addition step is performed by multiplying one word of A by B and the right shift is performed by shifting w bits to the right. In order to perform an exact division of u by 2^w, we add an integer multiple of n to u, so that the least significant word of the new u will be zero. Thus, if $u \neq 0 \pmod{2^w}$, we find an integer m such that $u + m \cdot n = 0 \pmod{2^w}$. Let u_0 and n_0 be the least significant words of u and n, respectively. We calculate m as

$$m = -u_0 \cdot n_0^{-1} \pmod{2^w}.$$

The word-level (m-ary) add-shift Montgomery product algorithm is given in Algorithm 5.15.

Algorithm 5.15 Word-Level Add-and-Shift Montgomery Product

Require: A, B, an odd number n, $k = sw$.
Ensure: $u = A \cdot B \cdot 2^{-k} \pmod{n}$.
1: $u := 0$;
2: **for** $i = 0$ to $s - 1$ **do**
3: $\quad u := u + A_i \cdot B$;
4: $\quad m := -u_0 \cdot n_0^{-1} \bmod 2^w$;
5: $\quad u := u + m \cdot n$;
6: $\quad u := u/2^w$;
7: **end for**
8: **Return**(u)

This algorithm specializes to the binary case by taking $w = 1$. In this case, when u is odd, the least significant bit u_0 is nonzero, and thus, $m = -u_0 \cdot n_0^{-1} = 1 \pmod{2}$.

5.4 Modular Exponentiation Operation

Modular exponentiation can be defined in terms of field multiplication as follows. Let x be a positive integer in $[1, n]$. Let also e be defined as an arbitrary positive integer. Then, we define modular exponentiation as the problem of finding the number y such that,

$$y = x^e \bmod n \tag{5.1}$$

Taking advantage of the linearity property of the modular operation, (5.1) can be evaluated by performing a reduction modulo n at each step of the exponentiation thus guaranteeing that all the partial results will not grow larger than twice the length of the modulus. In the rest of this Section we will consider that every multiplication operation always includes a subsequent reduction step.

In general one can follow two strategies in order to optimize the computation of (5.1). One approach is to implement field multiplication, the main building block required for field exponentiation, as efficiently as possible. The other is to reduce the total number of multiplications needed to compute (5.1). In this Section we address the latter approach, assuming that arbitrary choices of the base x are allowed but considering that the exponent e has been previously fixed.

In this section, we include a brief review of the main deterministic heuristic proposed in the literature for computing field exponentiation.

5.4.1 Binary Strategies

Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as, $e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i$. Then,

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i} = x^{2^{m-1}} \cdot \prod_{i=0}^{m-2} \mathbf{x}^{2^i e_i} \tag{5.2}$$

Binary strategies evaluate (5.2) by scanning the bits of the exponent e one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule [2]. Both strategies require a total of $m-1$ iterations. At each iteration a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m-1$ squarings and $H(e)-1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the MSB-first and the LSB-first binary algorithms are shown in Figures 5.16 and 5.17, respectively. The computational complexity of the algorithm in Figure 5.16 is given as,

$$P(e,m) = m + H(e) - 2 = \lfloor \log_2(e) \rfloor + H(e) - 1 \tag{5.3}$$

[2] Horner's rule, named after W. G. Horner, ranks among the most efficient algorithms for the computation of nth degree polynomials of the form,
$p(x) = p_n x^n + p_{n-1} x^n - 1 + \cdots + p_1 x + u_0, p_n \neq 0$, for fixed values of x.
Horner's rule solves this problem by evaluating $p(x)$ as,
$p(x) = (\dots(p_n x + p_{n-1})x + \cdots)x + p_0$.
This elegant algorithm was discovered independently by Isaac Newton 150 years earlier than Horner and by the Chinese mathematician C. C. Chao in the 13th century [178]

An Example. Let us define $e = 1903 = (11101101111)_2$. Then $m = 11$ and $H(e) = 9$. According to (5.3) the computational complexity of the binary algorithm is given as,

$P(e) = m + H(e) - 2 = 11 + 9 - 2 = 18$.

After evaluating the algorithm of Figure 5.16, the resulting binary sequence is given as,

$$x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{29} \rightarrow x^{58}$$
$$\rightarrow x^{59} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{237} \rightarrow x^{474} \rightarrow x^{475} \rightarrow x^{950}$$
$$\rightarrow x^{951} \rightarrow x^{1902} \rightarrow x^{1903}.$$

We compare the MSB-first and the LSB-first binary algorithms in terms of time and space requirements below:

- Both methods require $m - 1$ squarings and an average of $\frac{1}{2}(m-1)$ multiplications.
- The MSB-first binary method requires two registers: x and y.
- The LSB-first binary method requires three registers: x, y, and P. However, we note that P can be used in place of M, if the value of M is not needed thereafter.
- The multiplication (Step 4) and squaring (Step 5) operations in the LSB-first binary method are independent of one another, and thus these steps can be parallelized. Provided that we have two multipliers (one multiplier and one squarer) available, the running time of the LSB-first binary method is bounded by the total time required for computing $h-1$ squaring operations on k-bit integers.

Algorithm 5.16 MSB-First Binary Exponentiation

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \ldots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: $\mathbf{y} = x$;
2: **for** $i = m - 2$ downto 0 **do**
3: $\quad \mathbf{y} = \mathbf{y}^2$;
4: $\quad$ **if** $e_i == 1$ **then**
5: $\quad\quad \mathbf{y} = \mathbf{y} \cdot \mathbf{x}$;
6: $\quad$ **end if**
7: **end for**
8: **Return(y)**

5.4.2 Window Strategies

The binary method discussed in the preceding section can be generalized by scanning more than one bit at a time. Hence, the window method (first

Algorithm 5.17 LSB-First Binary Exponentiation

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: $\mathbf{p} = \mathbf{x}$; $\mathbf{y} = 1$;
2: **for** $i = 0$ to $m - 1$ **do**
3: **if** $e_i == 1$ **then**
4: $\mathbf{y} = \mathbf{y} \cdot \mathbf{p}$;
5: **end if**
6: $\mathbf{p} = \mathbf{p}^2$;
7: **end for**
8: **Return(y)**

described in [178]) scans k bits at a time. The window method is based on a k-ary expansion of the exponent, where the bits of the exponent e are divided into k-bit words or digits. The resulting words of e are then scanned performing k consecutive squarings and a subsequent multiplication as needed. In the following we describe the window method in a more formal way.

Algorithm 5.18 MSB-First 2^k-ary Exponentiation

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$, k divisor of m such that $\Psi = m/k$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: Pre-compute and store x^j for all $j = 1, 2, 3, 4, \dots, 2^k - 1$.
2: Divide e into k-bit words W_i for $i = 0, 1, 2, \dots, \Psi - 1$.
3: $\mathbf{y} = x^{W_{\Psi-1}}$;
4: **for** $i = \Psi - 2$ downto 0 **do**
5: $\mathbf{y} = \mathbf{y}^{2^k}$;
6: **if** $W_i \neq 0$ **then**
7: $\mathbf{y} = \mathbf{y} \cdot \mathbf{x}^{W_i}$;
8: **end if**
9: **end for**
10: **Return(y)**

Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as,

$$e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i.$$

Let k be a small divisor of m. Then this binary expansion of e can be partitioned into Ψ words of length k, such that $k\Psi = m$. If k does not divide m, then the exponent must be padded with at most $k - 1$ zeros. Let us define $W_i \in [\![0, 2^k - 1]\!]$ as,

$$W_i = \left(e_{ik+(k-1)} e_{ik+(k-2)} \cdots e_{ik+1} e_{ik}\right)_2 = \sum_{j=0}^{k-1} 2^j e_{(ik+j)} \tag{5.4}$$

Then, we can equivalently represent e as, $\sum_{i=0}^{\Psi-1} W_i \cdot 2^{id}$. Using the above definition we have,

$$\mathbf{y} = \mathbf{x}^e = \mathbf{x}^{\sum_{i=0}^{\Psi-1} 2^{id} W_i} = \prod_{i=0}^{\Psi-1} \mathbf{x}^{2^{id} W_i} \tag{5.5}$$

(5.5) is the basis of the window MSB-first procedure for exponentiation described in the pseudo-code of Figure 5.18. The window method first pre-computes the values of x^j for $j = 1, 2, 3, \ldots, 2^k - 1$. Then, the exponent e is scanned k bits at a time from the most significant word ($W_{\Psi-1}$) to the least significant word (W_0). At each iteration the current partial result y is raised to the 2^k power and multiplied with x^{W_i}, where W_i is the current nonzero word being processed. Referring to Figure 5.18, it can be seen that,

- The first part of the algorithm consists on the pre-computation of the first 2^k powers of $\mathbf{x}$ at a cost of $2^k - 2$ preprocessing multiplications.
- At each iteration of the main loop, the power $\mathbf{y}^{2^k}$ can be computed by performing k consecutive squarings. The total number of squarings is given by $(\Psi - 1)k = m - k$.
- At each iteration one multiplication is performed whenever the i-th word W_i is different than zero. Since all but one of the 2^k different values of W_i are nonzero, the average number of required multiplications is given as, $(\Psi - 1)(1 - 2^{-k}) = (\frac{m}{k} - 1)(1 - 2^{-k})$.

Thus, the average number of multiplications needed by the window method in order to compute an m-bit field exponentiation is given as,

$$P(m, k) = (2^k - 2) + (m - k) + (\frac{m}{k} - 1)(1 - 2^{-k}). \tag{5.6}$$

For $k = 1, 2, 3, 4$ the window method sketched at Figure 5.18 is called, respectively, *binary*, *quaternary*, *octary* and *hexa* MSB-first exponentiation method. In particular, note that by evaluating (5.6) for $k = 1$, the average number of multiplications for the binary algorithm can be found as $\frac{3}{2}(m - 1)$ field operations on average.

One obvious improvement of the strategy just outlined is that instead of calculating and storing all the 2^k first powers of x, one can just pre-compute the windows needed for a given exponent e, thus saving some operations. This last idea is illustrated in the examples below.

Example. Once again, let us consider the exponent $e = 1903 = (11101101111)_2$ with $m = 11$. Then, the window method computational complexity and resulting sequence using $k = 2, 3, 4$ can be found as,
Quaternary: $e = 1903 = (\mathbf{01}\,11\,01\,10\,11\,11)_2$

$P(m,k) = 2$ Pre-comp mults + 10 Sqrs + 5 mults $= 17$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3$.
Main sequence:

$$x^1 \rightarrow x^2 \rightarrow x^4 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{29} \rightarrow x^{58} \rightarrow x^{116} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{472} \rightarrow x^{475} \rightarrow x^{950} \rightarrow x^{1900} \rightarrow x^{1903}.$$

Octal: $e = 1903 = (\mathbf{011}\,101\,101\,111)_2$
$P(m,k) = 4$ Pre-comp mults + 9 Sqrs + 3 mults $= 16$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^5 \rightarrow x^7$.
Main sequence:

$$x^3 \rightarrow x^6 \rightarrow x^{12} \rightarrow x^{24} \rightarrow x^{29} \rightarrow x^{58} \rightarrow x^{116} \rightarrow x^{232} \rightarrow x^{237} \rightarrow x^{474} \rightarrow x^{948} \rightarrow x^{1896} \rightarrow x^{1903}$$

Hexa: $e = 1903 = (\mathbf{0111}\,\mathbf{0110}\,1111)_2$
$P(m,k) = 6$ Pre-comp mults + 8 Sqrs + 2 mults $= 16$.
Precomp. Sequence: $x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^7 \rightarrow x^{14} \rightarrow x^{15}$.
Main sequence:

$$x^7 \rightarrow x^{14} \rightarrow x^{28} \rightarrow x^{56} \rightarrow x^{112} \rightarrow x^{118} \rightarrow x^{236} \rightarrow x^{472} \rightarrow x^{944} \rightarrow x^{1888} \rightarrow x^{1903}.$$

However, none of the above deterministic methods is able to find the shortest addition chain[3] for $e = 1903$.

5.4.3 Adaptive Window Strategy

The adaptive or sliding window strategy is quite useful for exponentiations with extremely large exponents (i.e. exponents with bit length greater than 128 bits) mainly because of its ability to adjust its method of computation according to the specific form of the exponent at hand. This adjustment is done by partitioning the input exponent into a series of variable-length zero and nonzero words called *windows*. As opposed to the traditional window method discussed in the previous section, the sliding window algorithm provides a performance tradeoff in the sense that allows the processing of variable-length zero and nonzero digits. The main goal pursued by this strategy is to try to maximize the number and length of zero words, while using relatively large values of k.

A sliding window exponentiation algorithm is typically divided into two phases: exponent partitioning and the field exponentiation computation itself.

[3] Addition chains are formally defined in §6.3.3.

In the first phase, the exponent e is decomposed into zero and nonzero words (*windows*) W_i of length $L(W_i)$ by using some partitioning strategy. Although in general it is not required that the window's lengths $L(W_i)$ must all be equal, all nonzero windows should have a length $L(W_i)$ smaller than a given number k. Let Z be the number of zero windows and NZ be the number of non-zero windows, so that their addition Ψ represents the total number of windows generated by the partitioning phase, i.e.,

$$\Psi = Z + NZ \tag{5.7}$$

It is useful to force the least significant bit of a nonzero window W_i to be equal to 1. In this way, when comparing with the standard window method discussed in the previous Section, the number of preprocessing multiplications are at least nearly halved, since x^w must only be pre-computed for w odd.

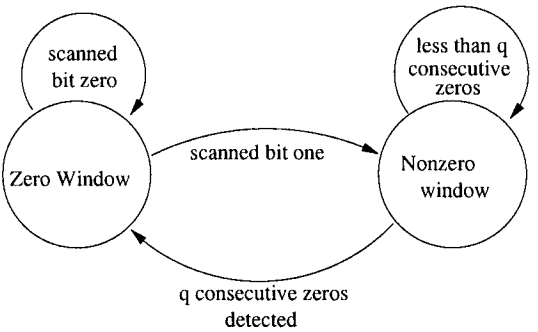

Fig. 5.9. Partitioning Algoritm

Several sliding window partitioning approaches have been proposed [116, 178, 191, 181, 30, 35]. Proposed techniques differ in whether the length of a nonzero window has to have a constant or a variable length. The partitioning algorithm instrumented in this work scans the exponent from the most significant to the least significant bit according to the finite state machine shown in Figure 5.9. Hence, at any moment the algorithm is either completing a zero window or a nonzero window. Zero windows are allowed to have an arbitrary length. However, the maximum length of any given nonzero window should not exceed the value of k bits.

Starting from the Zero Window State (ZWS), the exponent bits are checked one by one. As long as the value of the current scanned bit is zero, the algorithm stays in ZWS accumulating as many consecutive zeros as possible. If the incoming bit is one, the finite state machine switches to the Nonzero Window State (NZWS). The automaton will stay there as long as q consecutive zeros had not been collected. If this condition occurs the automaton switches to ZWS (usually q is chosen to be a small number, namely, $q \in [2, 5]$).

Otherwise, if k bits can been collected, the partitioning algorithm stores the new formed nonzero window and stays in NZWS in order to generate another nonzero window.

Algorithm 5.19 Sliding Window Exponentiation

Require: $\mathbf{x}, \mathbf{n}, e = (e_{m-1} \dots e_1 e_0)_2$.
Ensure: $\mathbf{y} = \mathbf{x}^e \bmod n$.
1: Pre-compute and store x^j for at most all $j = 1, 2, 3, 4, \dots, 2^k - 1$.
2: Divide e into zero and nonzero windows W_i of length $L(W_i)$ for $i = 0, 1, 2, \dots, \Psi - 1$.
3: $\mathbf{y} = x^{W_{\Psi-1}}$;
4: **for** $i = \Psi - 2$ downto 0 **do**
5: $\quad \mathbf{y} = \mathbf{y}^{2^{L(W_i)}}$;
6: $\quad$ **if** $W_i \neq 0$ **then**
7: $\quad\quad \mathbf{y} = \mathbf{y} \cdot \mathbf{x}^{W_i}$;
8: $\quad$ **end if**
9: **end for**
10: **Return(y)**

The pseudo-code for the sliding window exponentiation algorithm is shown in Figure 5.19. From that figure it can be seen that,

- The first part of the algorithm consists on the pre-computation of at most the first 2^k odd powers of $\mathbf{x}$ at a cost of no more than $2^{k-1}-1$ preprocessing multiplications.
- At step 2, the exponent e is partitioned using the strategy described above and depicted in Figure 5.9. As a consequence, a total of Z zero windows and NZ nonzero windows will be produced.
- At step 3, $\mathbf{y}$ is initialized using the value of the Most Significant Window as $\mathbf{y} = x^{W_{\Psi-1}}$. It is always assumed that $W_{\Psi-1} \neq 0$.
- At each iteration of the main loop, the power $\mathbf{y}^{2^{L(W_i)}}$ can be computed by performing $L(W_i)$ consecutive squarings. The total number of squarings is given by $m - L(W_{\Psi-1})$
- At each iteration one multiplication is performed whenever the i-th word W_i is different than zero. Recall that NZ represents the number of nonzero windows. Therefore, the number of multiplications required at this step of this algorithm is $NZ - 1$. Although the exact value of NZ will depend on the partitioning strategy instrumented, our experiments show that an approximate value for NZ using $q = 2, k = 5$, is about $0.15m$.

Thus, we find that the average number of multiplications needed to compute a field exponentiation for an m-bit exponent e is given as,

$$\begin{aligned} P(m,k) &= (2^{k-1} - 1) + (m - L(W_{k-1})) + NZ - 1 \\ &\approx 2^{k-1} - 1 + 1.15m - L(W_{k-1}). \end{aligned} \tag{5.8}$$

Due to the considerable high efficiency of the partitioning strategy for collecting zero words, the sliding window method significantly outperforms the standard window method when sufficiently large exponents are computed [181]. However, notice that the value of the parameter k cannot be chosen too large due to the exponentially increasing cost of pre-computing the first 2^k odd powers of $\mathbf{x}$ (step 1 of Figure 5.19). In practice and depending on the value of m, $k \in [4, 8]$ is generally adopted.

After executing the above algorithm, it is found that the modular exponentiation operation $M^e \bmod n$ with $e = 1903$, can be computed by performing 9 field squarings and 6 field multiplications, according with the sequence shown below,

$$\begin{aligned} x^1 \rightarrow x^2 \rightarrow x^3 \rightarrow x^6 \rightarrow x^{12} \rightarrow x^{24} \rightarrow x^{25} \rightarrow x^{50} \\ \rightarrow x^{100} \rightarrow x^{200} \rightarrow x^{300} \rightarrow x^{600} \rightarrow x^{900} \rightarrow x^{1800} \\ \rightarrow x^{1900} \rightarrow x^{1903}. \end{aligned} \tag{5.9}$$

Each of the deterministic heuristics just described clearly sets an upper bound on the number of field operations required for computing the modular exponentiation operation. In particular, the theoretical cost of the binary algorithm given in (5.3) implies that $l(e) \leq m + H(e) - 1$. A lower bound for $l(e)$ was found in [321] as, $\log_2 e + \log_2 H(e) - 2.13$. Therefore we can write,

$$\log_2 e + \log_2 H(e) - 2.13 \leq l(e) \leq \lfloor log_2(e) \rfloor + H(e) - 1 \tag{5.10}$$

Let us suppose that we are interested in computing the modular exponentiation for several exponents of a given fixed bit-length, say, m. Then, as it was shown in [191], the minimum number of underlying field operations is a function of the Hamming weight $H(e)$. Indeed, one can expect that on average $l(e)$ will be smaller for both, $H(e)$ closer to 0 and for $H(e)$ closer to m. On the contrary, when $H(e)$ is close to $m/2$, i.e., for those m-bit exponents having a balanced number of zeros and ones, $l(e)$ happens to be maximal [191].

5.4.4 RSA Exponentiation and the Chinese Remainder Theorem

Let us recall from Chapter 2 that the RSA algorithm requires computation of the modular exponentiation which is broken into a series of modular multiplications by the application of exponentiation heuristics. Before getting into the details of these operations, we make the following definitions:

- The public modulus n is a k-bit positive integer, ranging from 512 to 2048 bits.
- The secret primes p and q are approximately $k/2$ bits.
- The public exponent e is an h-bit positive integer. The size of e is small, usually not more than 32 bits. The smallest possible value of e is 3.

- The secret exponent d is a large number; it may be as large as $\phi(n) - 1$. We will assume that d is a k-bit positive integer.

After these definitions, we will study how the RSA modular exponentiation can be greatly benefit by applying the Chinese Remainder Theorem to it.

The Chinese Remainder Theorem

The Chinese Remainder Theorem(CRT) has a tremendous importance in cryptography. For instance, Quisquater and Couvreur proposed in [279] to use it for speeding up the RSA decryption primitive. It can be defined as follows.

Let p_i for $i = 1, 2, \ldots, k$ be pairwise relatively prime integers, i.e.,

$$gcd(p_i, p_j) = 1 \text{ for } i \neq j.$$

Given $u_i \in [0, p_i - 1]$ for $i = 1, 2, \ldots, k$, the Chinese remainder theorem states that there exists a unique integer u in the range $[0, P-1]$ where $P = p_1 p_2 \cdots p_k$ such that

$$u = u_i \pmod{p_i}.$$

In the case of RSA decryption primitive, The Chinese remainder theorem tells us that the computation of

$$M := C^d \pmod{p \cdot q},$$

can be broken into two parts as

$$M_1 := C^d \pmod{p},$$
$$M_2 := C^d \pmod{q},$$

after which the final value of M is computed (lifted) by the application of a Chinese remainder algorithm. There are two algorithms for this computation: The single-radix conversion (SRC) algorithm and the mixed-radix conversion (MRC) algorithm. Here, we briefly describe these algorithms, details of which can be found in [105, 355, 178, 209]. Going back to the general example, we observe that the SRC or the MRC algorithm computes u given $u_1, u_2, \ldots, u_k$ and $p_1, p_2, \ldots, p_k$. The SRC algorithm computes u using the summation

$$u = \sum_{i=1}^{k} u_i c_i P_i \pmod{P},$$

where

$$P_i = p_1 p_2 \cdots p_{i-1} p_{i+1} \cdots p_k = \frac{P}{p_i},$$

and c_i is the multiplicative inverse of P_i modulo p_i, i.e.,

$$c_i P_i = 1 \pmod{p_i}.$$

Thus, applying the SRC algorithm to the RSA decryption, we first compute

$$\begin{aligned} M_1 &:= C^d \pmod{p}, \\ M_2 &:= C^d \pmod{q}, \end{aligned}$$

However, applying Fermat's theorem to the exponents, we only need to compute

$$\begin{aligned} M_1 &:= C^{d_1} \pmod{p}, \\ M_2 &:= C^{d_2} \pmod{q}, \end{aligned}$$

where

$$\begin{aligned} d_1 &:= d \bmod (p-1), \\ d_2 &:= d \bmod (q-1). \end{aligned}$$

This provides some savings since $d_1, d_2 < d$; in fact, the sizes of d_1 and d_2 are about half of the size of d. Proceeding with the SRC algorithm, we compute M using the sum

$$M = M_1 c_1 \frac{pq}{p} + M_2 c_2 \frac{pq}{q} \pmod{n} \;=\; M_1 c_1 q + M_2 c_2 p \pmod{n},$$

where $c_1 = q^{-1} \pmod{p}$ and $c_2 = p^{-1} \pmod{q}$. This gives

$$M = M_1 (q^{-1} \bmod p) q + M_2 (p^{-1} \bmod q) p \pmod{n}.$$

In order to prove this, we simply show that

$$\begin{aligned} M \pmod{p} = M_1 \cdot 1 + 0 &= M_1, \\ M \pmod{q} = 0 + M_2 \cdot 1 &= M_2. \end{aligned}$$

The MRC algorithm, on the other hand, computes the final number u by first computing a triangular table of values:

$$\begin{array}{llllll} u_{11} & & & & & \\ u_{21} & u_{22} & & & & \\ u_{31} & u_{32} & u_{33} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ u_{k1} & u_{k2} & \cdots & \cdots & u_{k,k} \end{array}$$

where the first column of the values u_{i1} are the given values of u_i, i.e., $u_{i1} = u_i$. The values in the remaining columns are computed sequentially using the values from the previous column according to the recursion

$$u_{i,j+1} = (u_{ij} - u_{jj}) c_{ji} \pmod{p_i},$$

where c_{ji} is the multiplicative inverse of p_j modulo p_i, i.e.,

$$c_{ji}p_j = 1 \pmod{p_i}.$$

For example, u_{32} is computed as

$$u_{32} = (u_{31} - u_{11})c_{13} \pmod{p_3},$$

where c_{13} is the inverse of p_1 modulo p_3. The final value of u is computed using the summation

$$u = u_{11} + u_{22}p_1 + u_{33}p_1p_2 + \cdots + u_{kk}p_1p_2 \cdots p_{k-1}$$

which does not require a final modulo P reduction. Applying the MRC algorithm to the RSA decryption, we first compute

$$M_1 := C^{d_1} \pmod{p},$$
$$M_2 := C^{d_2} \pmod{q},$$

where d_1 and d_2 are the same as before. The triangular table in this case is rather small, and consists of

$$M_{11}$$
$$M_{21}\ M_{22}$$

where $M_{11} = M_1$, $M_{21} = M_2$, and

$$M_{22} = (M_{21} - M_{11})(p^{-1} \bmod q) \pmod{q}.$$

Therefore, M is computed using

$$M := M_1 + [(M_2 - M_1) \cdot (p^{-1} \bmod q) \bmod q] \cdot p.$$

This expression is correct since

$$M \pmod{p} = M_1 + 0 = M_1,$$
$$M \pmod{q} = M_1 + (M_2 - M_1) \cdot 1 = M_2.$$

The MRC algorithm is more advantageous than the SRC algorithm for two reasons:

- It requires a single inverse computation: $p^{-1} \bmod q$.
- It does not require the final modulo n reduction.

The inverse value $(p^{-1} \bmod q)$ can be precomputed and saved. Here, we note that the order of p and q in the summation in the proposed public-key cryptography standard PKCS # 1 is the reverse of our notation. The data structure [194] holding the values of user's private key has the variables:

```
exponent1 INTEGER, -- d mod (p-1)
exponent2 INTEGER, -- d mod (q-1)
coefficient INTEGER, -- (inverse of q) mod p
```

Thus, it uses $(q^{-1} \bmod p)$ instead of $(p^{-1} \bmod q)$. Let M_1 and M_2 be defined as before. By reversing p, q and M_1, M_2 in the summation, we obtain

$$M := M_2 + [(M_1 - M_2) \cdot (q^{-1} \bmod p) \bmod p] \cdot q.$$

This summation is also correct since

$$\begin{aligned} M \pmod q &= M_2 + 0 \;=\; M_2, \\ M \pmod p &= M_2 + (M_1 - M_2) \cdot 1 \;=\; M_1, \end{aligned}$$

as required. Assuming p and q are $(k/2)$-bit binary numbers, and d is as large as n which is a k-bit integer, we now calculate the total number of bit operations for the RSA decryption using the MRC algorithm. Assuming d_1, d_2, $(p^{-1} \bmod q)$ are precomputed, and that the exponentiation algorithm is the binary method, we calculate the required number of multiplications as

- Computation of M_1: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of M_2: $\frac{3}{2}(k/2)$ $(k/2)$-bit multiplications.
- Computation of M: One $(k/2)$-bit subtraction, two $(k/2)$-bit multiplications, and one k-bit addition.

Also assuming multiplications are of order k^2, and subtractions are of order k, we calculate the total number of bit operations as

$$2\,\frac{3k}{4}(k/2)^2 + 2(k/2)^2 + (k/2) + k \;=\; \frac{3k^3}{8} + \frac{k^2 + 3k}{2}.$$

On the other hand, the algorithm without the CRT would compute $M = C^d \pmod n$ directly, using $(3/2)k$ k-bit multiplications which require $3k^3/2$ bit operations. Thus, considering the high-order terms, we conclude that the CRT based algorithm will be approximately 4 times faster.

5.4.5 Recent Prime Finite Field Arithmetic Designs on FPGAs

In this Subsection, we show some of the most significant designs recently published in the open literature for modular exponentiation. All designs included in Table 5.1 were implemented either on VLSI or on reconfigurable hardware platforms. Notice also that there is a strong correlation between design's speed and the date of publication ,i.e., fastest designs tend to be the ones which have been more recently published.

Liu et al. presented in [210] a design based on the *distributed module cluster* microarchitecture especially designed to reduce long datapaths. The throughput achieved by their technique ranks as the fastest design published to date. Amanor et al. presented in [6] several designs based on different multiplier strategies. Their redundant interleaved multiplier can compute a 1024-bit RSA decryption exponentiation in just 6.1 mS. On the other hand, authors in [6] also essayed designs based on a Montgomery multiplier block,

Table 5.1. Modular Exponentiation Comparison Table

Work	year	Platform	Cost	BRAMs, 18-bit M	Freq. MHz	1024-bit time(mS)	Mult. Block Utilized
Liu et al.[210]	2005	0,13μm CMOS	221K gates	None	714	1.47	DMC Mont. Mult.
Amanor et al.[6]	2005	Virtex	4608 CLBs	None	69.4	6.1 (est.)	Interleaved Mult.
Kelley et al.[170]	2005	Virtex II	2847 LUTs	5Kb, 32	102	6.6	16-bit Scal radix 2^{16}
Mukaida et al. [243]	2004	0,11μm CMOS	61K gates	–	250	7.3	64-bit Scal radix 2^4
Amanor et al.[6]	2005	Virtex	8640 CLBs	None	42.1	9.7 (est.)	CSA Mont. Mult.
Blum et al. [29]	2001	Virtex	6613 CLBs	–	45	12	Mont. Mult. radix 2^4
Harris et al.[134]	2005	Virtex II Pro	5598 LUTs	5Kb, -	144	16	16-bit Scal radix 2
Kelley et al.[170]	2005	Virtex II	780 LUTs	5Kb, 8	102	22	16-bit Scal radix 2^{16}
Todorov[361]	2000	0,5μm CMOS	28K gates	–	64	46	16-bit Scal radix 8
Tenca et al.[359]	2003	0,5μm CMOS	28K gates	–	80	88	8-bit Scal radix 2

but the timing performance obtained was somehow lesser than that of the interleaved multiplier. Kelley et al. presented in [170] a 16-bit Montgomery scalable multiplier of radix 2^{16}, the highest radix for a Montgomery multiplier published to date. With that multiplier block, authors in [170] were able to achieve a 1024-bit exponentiation in just 6.6 mS. It is noted though, that the design by Kelley et al. utilized 32 embedded multipliers plus some 5K bit RAMs. Blum et al. designed in 2001 a high-radix Montgomery multiplier architecture able of achieving an exponentiation time of 12mS [29].

On the other side of the spectrum, designs by Todorov [361] and Tenca et al. [359] rank among the most economical of all high performance designs included in Table 5.1.

Due to the diversity of platforms and resources employed by the designs featured in Table 5.1, it results rather difficult to establish reasonable criteria for selecting the most efficient of all of them. Here, we say that a given design is efficient if it offers a great cost-benefit compromise. Nevertheless, the design by Mukaida et al. reported in [243] seems to be our best bet for this category. Utilizing a radix 16 multiplier implemented on ASIC at a clock speed of 250MHz, authors in [243] produced a design able to compute a 1024-bit exponentiation within 7.3mS at a hardware price of just 61K gates.

A final word about the performance comparison presented here. 1024-bit RSA exponentiation is one of the few major cryptographic primitives which shows a moderate performance speedup when hardware implementations of it are compared with its software counterparts. On this regard, Table 5.2 compares two RSA software designs against two of the fastest designs surveyed here.

As it can be seen, the speedup attained by the design in [210] is of 25.17 and 15.03 when compared with an XScale and a Pentium IV implementations, respectively.

Table 5.2. Modular Exponentiation: Software vs Hardware Comparison Table

Work	year	Platform	Cost	Freq. MHz	1024-bit time(mS)	Speedup
Liu et al.[210]	2005	$0{,}13\mu m$ CMOS	221K gates	714	1.47	1
Amanor et al.[6]	2005	Virtex	4608 CLBs	69.4	6.1 (est.)	4.5
Martínez-Silva et al.[219]	2005	IPAQ H5550 Intel XScale	–	400MHz	37	25.17
López-Peza et al.[294]	2004	Intel Pentium IV	–	2.4GHz	22.10	15.03

5.5 Conclusions

In this Chapter we reviewed several relevant algorithms for performing efficient modular arithmetic on large integer numbers. Addition, modular addition, Reduction, modular multiplication and exponentiation were some of the operations studied throughout the material contained in this Chapter. Strong emphasis was placed on discussing the best strategies for implementing those algorithms on hardware platforms, either in the domain of ASIC designs or reconfigurable hardware platforms.

We intended to cover some of the most significant mathematical and algorithmic aspects of the modular exponentiation operation, providing the necessary knowledge to the hardware designer who is interested implementing the RSA algorithm using the reconfigurable hardware technology.

The last Section of this Chapter contains a small survey of some of the most representative designs published in the open literature for modular exponentiation computation.

6

Binary Finite Field Arithmetic

In this Chapter we review some of the most relevant arithmetic algorithm on binary extension fields $GF(2^m)$. The arithmetic over $GF(2^m)$ has many important applications in the domains of theory of code theory and in cryptography [221, 227, 380]. Finite field's arithmetic operations include: addition, subtraction, multiplication, squaring, square root, multiplicative inverse, division and exponentiation.

Addition and subtraction are equivalent operations in $GF(2^m)$. Addition in binary finite fields is defined as polynomial addition and can be implemented simply as the XOR addition of the two m-bit operands.

That is why we begin this Section with a review of the main algorithms reported in the open literature for perhaps the most important field arithmetic operation: field multiplication.

6.1 Field Multiplication

Let $A(x), B(x)$ and $C'(x) \in GF(2^m)$ and $P(x)$ be the irreducible polynomial generating $GF(2^m)$. Multiplication in $GF(2^m)$ is defined as polynomial multiplication modulo the irreducible polynomial $P(x)$, namely,

$$C'(x) = A(x)B(x) \bmod P(x).$$

One important factor for designing multipliers in binary extension fields is the way that field elements are represented, i.e, the sort of basis that is being used[1]. Indeed, field element representation has a crucial role in the design of architectures for arithmetic operations.

Besides the polynomial or canonical basis, several other bases have been proposed for the representation of elements in binary extension fields [221, 51, 390]. Among them, probably the most studied one is the Gaussian normal basis [281, 285, 164, 89, 405].

[1] More details about field element representation can be found in §4.2.

Even though efficient bit-parallel multipliers for both canonical and normal basis representation have been regularly reported in the specialized literature, in this Section we will mainly focus on polynomial basis multiplier schemes, mostly because they are consistently more efficient than their counterparts in other bases[2].

Traditionally, the space complexity of bit parallel multipliers is expressed in terms of the number of 2-input AND and XOR gates. For reconfigurable hardware devices though, the total number of CLBs and/or LUTs utilized by the design is preferred. Depending on their space complexity, bit parallel multipliers are classified into two categories: quadratic and subquadratic space complexity multipliers.

Several quadratic and subquadratic space complexity multipliers have been reported in literature. Examples of quadratic multipliers can be found in [220, 182, 389, 390, 350, 129, 352, 315, 129, 282, 391, 112, 201, 292, 283, 284, 247, 90, 146]. On the other hand, some examples of sub-quadratic multipliers can be found in [267, 268, 269, 270, 291, 86, 298, 117, 293, 349, 16, 106, 91, 377, 239]. This latter category offers low space complexity especially for large values of n and therefore they are in principle attractive for cryptographic applications.

Among the several approaches for computing the product $C'(x)$, we will study the following strategies,

- Two-Step multipliers
- Interleaving Multiplication
- Matrix-Vector Multipliers
- Montgomery Multiplier

In the case of two-step multipliers, first the polynomial product $C(x)$ of degree at most $2m-2$ is obtained as,

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{6.1}$$

Then, in a second step, the reduction operation needs to be performed in order to obtain the $m-1$ degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) \tag{6.2}$$

It is noticed that once the irreducible polynomial $P(x)$ has been selected, the reduction step can be accomplished by using XOR gates only.

In the rest of this section different implementation aspects and several efficient methods for computing $GF(2^m)$ finite field multiplication are extensively studied. In § 6.1.1 the analysis of the school or classical method is presented. Subsection § 6.1.2 analyzes a variation of the classical Karatsuba-Ofman algorithm as one of the most efficient techniques to find the polynomial product of

[2] Examples of efficient normal basis multiplier designs recently published in the open literature can be found in [164, 89, 285, 281, 405, 352, 283].

product of Equation 6.1. In subsection § 6.1.3 we describe an efficient method to compute polynomial squaring in hardware, at a complexity cost of just $O(1)$. Subsections § 6.1.4 and § 6.1.5 explain an efficient hardware methodology that carries on the reduction step of Equation 6.2 considering three separated cases, namely, reduction with irreducible trinomials, pentanomials and arbitrary polynomials. Then in §6.1.6 a method that interleaves the steps of multiplication and reduction is presented. Subsection §6.1.7 outlines field multiplication methods that solve Equation 6.1 by reformulating it in terms of matrix-vector operations. Then, in §6.1.8, the binary field version of the Montgomery multiplier is discussed. Finally, §6.1.9 compares the most relevant binary field multiplier designs published up-to date. Designs are compared from the perspective of three different metrics, namely, speed, compactness and efficiency.

6.1.1 Classical Multipliers and their Analysis

Let $A(x), B(x)$ be elements of $GF(2^m)$, and let $P(x)$ be the degree m irreducible polynomial generating $GF(2^m)$. Then, the field product $C'(x) \in GF(2^m)$ can be obtained by first computing the polynomial product $C(x)$ as

$$C(x) = A(x)B(x) = \left(\sum_{i=0}^{m-1} a_i x^i\right)\left(\sum_{i=0}^{m-1} b_i x^i\right) . \tag{6.3}$$

Followed by a reduction operation, performed in order to obtain the $(m-1)$-degree polynomial $C'(x)$, which is defined as

$$C'(x) = C(x) \bmod P(x) . \tag{6.4}$$

Once the irreducible polynomial $P(x)$ is selected and fixed, the reduction step can be accomplished using only XOR gates. The classical algorithm formulates these two steps into a single matrix-vector product, and then reduces the product matrix using the irreducible polynomial that generates the field. The degree $2m-2$ polynomial $C(x)$ in (6.3) can be written as,

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-2} \\ c_{m-1} \\ c_m \\ c_{m+1} \\ \vdots \\ c_{2m-3} \\ c_{2m-2} \end{bmatrix} = \begin{bmatrix} a_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & 0 & \cdots & 0 & 0 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & a_{m-5} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_1 & a_0 \\ 0 & a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & a_{m-2} & \cdots & a_3 & a_2 \\ \vdots & & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix} \tag{6.5}$$

The computation of the field product $C'(x)$ in (6.4) can be accomplished by first computing the above matrix-vector product to obtain the vector C which has $2m-1$ elements. By taking into account the zero entries of the matrix, we obtain the gate complexity of the computation of $C(x)$ in Table 6.1.

Table 6.1. The Computation of $C(x)$ Using Equation (6.5)

Coordinates	AND Gates	XOR Gates	T_A	T_X
c_i for $0 \le i \le m-1$	$i+1$	i	1	$\log_2\lceil i+1 \rceil$
c_{m+i} for $0 \le i \le m-2$	$m-(i+1)$	$m-(i+1)-1$	1	$\log_2\lceil m-1-i \rceil$

Therefore, the total number of gates are found as

$$\text{AND Gates: } 1+2+\cdots+m+(m-1)+(m-2)+\cdots+2+1 = m^2 ,$$
$$\text{XOR Gates: } 1+2+\cdots+(m-1)+(m-2)+\cdots+2+1 = (m-1)^2 .$$

The AND gates operate all in parallel, and require a single AND gate delay T_A. On the other hand, the XOR gates are organized as a binary tree of depth $\log_2\lceil j \rceil$ in order to add j operands. The total time complexity is then found by taking the largest number of terms, which is equal to m for the computation of c_{m-1}. Therefore, the total complexity of computing the matrix-vector product (6.5) so that the elements c_i for $i = 0, 1, \ldots, 2m-2$ are all found is given as,

$$\begin{aligned} &\text{AND Gates} = m^2 \\ &\text{XOR Gates} = (m-1)^2 \\ &\text{Total Delay} = T_A + \lceil \log_2 m \rceil T_X . \end{aligned} \tag{6.6}$$

Notice that this computation must be followed by reduction modulo the irreducible polynomial $P(x)$. The reduction operation is discussed in Section 6.1.4.

6.1.2 Binary Karatsuba-Ofman Multipliers

Several architectures have been reported for multiplication in $GF(2^m)$. For example, efficient bit-parallel multipliers for both canonical and normal basis representation have been proposed in [136, 351, 241, 389, 20]. All these algorithms exhibit a space complexity $O(m^2)$. However, there are some asymptotically faster methods for finite field multiplications, such as the Karatsuba-Ofman algorithm [168, 268]. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under $O(m^2)$ operations [14]. Karatsuba-Ofman multipliers may result in fewer bit operations at the expense of some design restrictions, particularly in the selection of the degree of the generating irreducible polynomial m.

In [268], it was presented a Karatsuba-Ofman multiplier based on composite fields of the type $GF((2^n)^s)$ with $m = sn$, $s = 2^t$, t an integer. However, for certain applications, quite particularly, elliptic curve cryptosystems, it is important to consider finite fields $GF(2^m)$ where m is not necessarily a power of two. In fact, for this specific application some sources [145] suggest that, for security purposes, it is strongly recommended to choose degrees m primes for finite fields in the range $[160, 512]$.

In the rest of this subsection we will briefly describe a variation of the classic Karatsuba-Ofman Multiplier called *binary Karatsuba-Ofman multipliers* that was first presented in [293]. Binary Karatsuba-Ofman multipliers can be utilized arbitrarily, regardless the form of the required degree m.

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ of degree $m = rn$, with $r = 2^k$, k an integer. Let A, B be two elements in $GF(2^m)$. Both elements can be represented in the polynomial basis as,

$$\begin{aligned} A = \sum_{i=0}^{m-1} a_i x^i &= \sum_{i=\frac{m}{2}}^{m-1} a_i x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} a_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} a_i x^i = x^{\frac{m}{2}} A^H + A^L \end{aligned}$$

and

$$\begin{aligned} B = \sum_{i=0}^{m-1} b_i x^i &= \sum_{i=\frac{m}{2}}^{m-1} b_i x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i \\ &= x^{\frac{m}{2}} \sum_{i=0}^{\frac{m}{2}-1} b_{i+\frac{m}{2}} x^i + \sum_{i=0}^{\frac{m}{2}-1} b_i x^i = x^{\frac{m}{2}} B^H + B^L. \end{aligned}$$

Then, using last two equations, the polynomial product is given as

$$C = x^m A^H B^H + (A^H B^L + A^L B^H) x^{\frac{m}{2}} + A^L B^L. \tag{6.7}$$

Karatsuba-Ofman algorithm is based on the idea that the product of last equation can be equivalently written as,

$$\begin{aligned} C = {} & x^m A^H B^H + A^L B^L + \\ & (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H)) x^{\frac{m}{2}} \\ = {} & x^m C^H + C^L. \end{aligned} \tag{6.8}$$

Let us define

$$\begin{aligned} M_A &:= A^H + A^L; \\ M_B &:= B^L + B^H; \\ M &:= M_A M_B. \end{aligned} \tag{6.9}$$

Using Equation 6.8, and taking into account that the polynomial product C has at most $2m - 1$ coordinates, we can classify its coordinates as,

$$\begin{aligned} C^H &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m]; \\ C^L &= [c_{m-1}, c_{m-2}, \ldots, c_1, c_0]. \end{aligned} \tag{6.10}$$

Although (6.8) seems to be more complicated than (6.7), it is easy to see that Equation (6.8) can be used to compute the product at a cost of four polynomial additions and three polynomial multiplications. In contrast, when using equation (6.7), one needs to compute four polynomial multiplications and three polynomial additions. Due to the fact that polynomial multiplications are in general much more expensive operations than polynomial additions, it is valid to conclude that (6.8) is computationally simpler than the classic algorithm.

Algorithm 6.1 $mul2^k(C, A, B)$: $m = 2^k n$-bit Karatsuba-Ofman Multiplier

Require: Two elements $A, B \in GF(2^m)$ with $m = rn = 2^k n$, where A, B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Ensure: A polynomial $C = AB$ with up to $2m-1$ coordinates, where $C = x^m C^H + C^L$.
1: **if** $r == 1$ **then**
2: $C = mul\_n(A, B)$;
3: **Return**(C)
4: **end if**
5: **for** i from 0 to $\frac{r}{2} - 1$ **do**
6: $M_{Ai} = A_i^L + A_i^H$;
7: $M_{Bi} = B_i^L + B_i^H$;
8: **end for**
9: $mul2^k(C^L, A^L, B^L)$;
10: $mul2^k(M, M_A, M_B)$;
11: $mul2^k(C^H, A^H, B^H)$;
12: **for** i from 0 to $r - 1$ **do**
13: $M_i = M_i + C_i^L + C_i^H$;
14: **end for**
15: **for** i from 0 to $r - 1$ **do**
16: $C_{\frac{r}{2}+i} = C_{\frac{r}{2}+i} + M_i$;
17: **end for**
18: **Return**(C).

Karatsuba-Ofman's algorithm can be applied recursively to the three polynomial multiplications in (6.8). Hence, we can postpone the computations of the polynomial products $A^H B^H$, $A^L B^L$ and M, and instead we can split again each one of these three factors into three polynomial products. By applying this strategy recursively, in each iteration each degree polynomial multiplication is transformed into three polynomial multiplications with their degrees reduced to about half of its previous value.

Eventually, after no more than $\lceil \log_2(m) \rceil$ iterations, all the polynomial operands collapse into single coefficients. In the last iteration, the resulting bit

multiplications can be directly computed. Although it is possible to implement the Karatsuba-Ofman algorithm until the $\lceil \log_2 m \rceil$ iteration, it is usually more practical to truncate the algorithm earlier. If the Karatsuba-Ofman algorithm is truncated at a certain point, the remaining multiplications can be computed by using alternative techniques[3].

Let us consider the algorithm presented in Algorithm 6.1. If $r = 1$, then the product is trivially found in lines 1-3 as the result of the single n-bit polynomial multiplication $C = mul\_n(A, B)$. Otherwise, in the first loop of the algorithm (lines 4-6) the polynomials M_A and M_B of equation (6.9) are computed by a direct polynomial addition of $A^H + A^L$ and $B^H + B^L$, respectively. In lines 7-9, C^L, C^H and M, are obtained via $\frac{r}{2}$-bit polynomial multiplication. After completion of these polynomial multiplications, the final value of the lower half of C^L as well as the upper half of C^H are found. To find the final values of the upper half of the polynomial C^L and the lower half of C^H, we need to combine the results obtained from the multiplier blocks with the polynomials C^H, C^L and M, as described in equations (6.8) and (6.9). This final computation is implemented in lines 10 through 13 of figure 6.1.

Complexity Analysis

The space complexity of the Algorithm 6.1 can be estimated as follows. The computation of the loop in lines 4-6 requires $2(\frac{r}{2}) = r$ additions. The execution of lines 7-9, implies the cost of 3 $\frac{r}{2}$-bit polynomial multipliers. Finally, lines 10-13 can be computed with a total of $3r$ additions. Notice that if $n > 1$ the additions in Algorithm 6.1 need to be multi-bit operations. Noticing also that m-bit multiplications in $GF(2)$ can generate at most $(2m - 1)$-bit products, we can have an extra saving of four bit-additions in lines 11 and 13. Hence, the addition complexity per iteration of the $m = 2^k n$-bits Karatsuba-Ofman multiplier presented in Algorithm 6.1 is given as $r + 3r = 4r$ n-bit additions plus three times the number of additions needed in a $\frac{r}{2}$ multiplier block, minus four bit additions. Notice that for n-bit arithmetic, each one of these additions can be implemented using n XOR gates.

Recall that m is a composite number that can be expressed as $m = rn$, with $r = 2^k$, k an integer. Then, one can successively invoke $\frac{r}{2^i}$-bit multiplier blocks, 3^i times each, for $i = 1, 2, \ldots, \log_2 r$. After $k = \log_2 r$ iterations, all the multiplier operations will involve polynomial multiplicands with degree n. These multiplications can be then computed using an alternative technique, like the classic algorithm. By applying iteratively the analysis given above, one can see that the total XOR gate complexity of the $m = 2^k n$-bit hybrid Karatsuba-Ofman multiplier truncated at the n-bit operand level is given as

[3] such as the classical algorithm studied in §6.1.1 or other techniques

$$\begin{aligned}
\text{XOR Gates} &= M_{xor2^n} 3^{\log_2 r} + \sum_{i=1}^{\log_2 r} 3^{i-1}\left(\frac{8rn}{2^i} - 4\right) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn \sum_{i=1}^{\log_2 r} \frac{3^{i-1}}{2^i} - 4 \sum_{i=1}^{\log_2 r} 3^{i-1} \\
&= M_{xor2^n} 3^{\log_2 r} + \frac{8}{3} rn \sum_{i=1}^{\log_2 r} \frac{3^i}{2} - \frac{4}{3} \sum_{i=1}^{\log_2 r} 3^i \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn\left(\frac{3}{2}^{\log_2 r} - 1\right) - 2(3^{\log_2 r} - 1) \\
&= M_{xor2^n} 3^{\log_2 r} + 8rn(r^{\log_2 \frac{3}{2}} - 1) - 2(r^{\log_2 3} - 1) \\
&= M_{xor2^n} r^{\log_2 3} + 8n(r^{\log_2 3} - 8r) - 2(r^{\log_2 3} - 1) \\
&= r^{\log_2 3}(8n - 2 + M_{xor2^n}) - 8rn + 2 \\
&= \left(\frac{m}{n}\right)^{\log_2 3} \left(8\frac{m}{r} - 2 + M_{xor2^n}\right) - 8m + 2.
\end{aligned}$$

Where M_{xor2^n} represents the XOR gate complexity of the block selected to implement the n-bit multipliers.

Similarly, notice that no AND gate is needed in Algorithm 6.1, except when the block selected to implement the n-bit multiplier is called. Let M_{and2^n} be the AND gate complexity of the block selected to implement the n-bit multiplier. Then, since this block is called exactly $3^{\log_2 r}$ times, we conclude that the total number of AND gates needed to implement the algorithm in 6.1 is given as,

$$\text{AND gates} = r^{\log_2 3} M_{and2^n} = \left(\frac{m}{n}\right)^{\log_2 3} M_{and2^n}$$

We give the time complexity of Algorithm 6.1 as follows. The execution of the first loop in lines 4-6 can be computed in parallel in a hardware implementation. Therefore, the required time for this part of the algorithm is of just 1 n-bit addition delay, which is equal to an XOR gate delay T_X. Lines 7-9, can also be implemented in parallel. Thus, the associated cost is of one $\frac{r}{2}$-bit multiplier delay. Notice that we cannot implement this second part of the algorithm in parallel with the first one because of the inherent dependencies of the variables. Finally, lines 10-13 can be computed with a delay of just $3T_X$. Hence, the associated time delay of the $m = 2^k n$-bit Karatsuba-Ofman multiplier of figure 6.1 is given as

$$\text{Time Delay} = T_{delay2^n} + \sum_{i=1}^{\log_2 r} 3 = T_{delay2^n} + 4T_X \log_2 r.$$

In this case it has been assumed that the block selected to implement the $GF(2^n)$ arithmetic has a T_{delay2^n} gate delay associated with it.

In summary, the space and time complexities of the m-bit Karatsuba-Ofman multiplier are given as

$$\begin{array}{l} \text{XOR Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3}\left(8\frac{m}{r} - 2 + M_{xor2^n}\right) - 8m + 2 \ ; \\ \text{AND Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3} M_{and2^n} \ ; \\ \text{Time Delay} \leq T_{delay2^n} + 4T_X \log_2\left(\frac{m}{n}\right) \ . \end{array} \tag{6.11}$$

As it has been mentioned above, the hybrid approach proposed here requires the use of an efficient multiplier algorithm to perform the n-bit polynomial multiplications. Let us recall that in §6.1.1 above, it was found that the space and time complexities for the classic n-bit multiplier are given as

$$\begin{array}{l} \text{XOR Gates} = (n-1)^2 \ ; \\ \text{AND Gates} = n^2 \ ; \\ \text{Time Delay} \leq T_{AND} + T_X \lceil \log_2 n \rceil \ . \end{array} \tag{6.12}$$

Combining the complexities given in equation (6.12), together with the complexities of equation (6.11) we conclude that the space and time complexities of the hybrid m-bit Karatsuba-Ofman multiplier truncated at the n-bit multiplicand level are upper bounded by

$$\begin{array}{lcl} \text{XOR Gates} & \leq & \left(\frac{m}{n}\right)^{\log_2 3}(8n - 2 + M_{xor2^n}) - 8m + 2 \\ & = & \left(\frac{m}{n}\right)^{\log_2 3}(n^2 + 6n - 1) - 8m + 2 \ ; \\ \text{AND Gates} & \leq & 3^{\log_2 r} M_{and2^n} \ = \ \left(\frac{m}{n}\right)^{\log_2 3} n^2 ; \\ \text{Time Delay} & \leq & T_{AND} + T_X(\log_2 n + 4\log_2 r) \ . \end{array} \tag{6.13}$$

Let us consider now the cases where m is a power of two, $m = rn = 2^k$, $k > 2$. Then, $n = 4$ is the most optimal selection for the hybrid Karatsuba-Ofman algorithm. For this case using equation (6.13) we obtain

$$\begin{array}{l} \text{XOR Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3}(n^2 + 6n - 1) - 8m + 2 \\ \qquad = \left(\frac{2^k}{4}\right)^{\log_2 3}(4^2 + 6 \cdot 4 - 1) - 8 \cdot 2^k + 2 \\ \qquad = 13 \cdot 3^{k-1} - 2^{k+3} + 2; \\ \text{AND Gates} \leq \left(\frac{m}{n}\right)^{\log_2 3} n^2 \ = \ \left(\frac{2^k}{4}\right)^{\log_2 3} 4^2 \ = \ 16 \cdot 3^{k-2}; \\ \text{Time Delay} \leq T_{AND} + T_X(\log_2 n + 4\log_2 r) \ = \\ \qquad = T_{AND} + T_X(\log_2 4 + 4\log_2 2^{k-2}) \ = \ T_{AND} + T_X(4k - 6) \ . \end{array} \tag{6.14}$$

Table 6.2 shows the space and time complexities for the hybrid Karatsuba-Ofman multiplier using the results found in equation (6.14). The values of m presented in Table 6.2 correspond to the first ten powers of two, i.e., $m = 2^k$ for $i = 0, 1, \ldots, 9$. Notice that the multipliers for $m = 1, 2, 4$ are assumed to be implemented using the classical method only. As we will see, the complexities of the hybrid Karatusba multiplier for degrees $m = 2^k$ happen to be crucial to find the hybrid Karatsuba-Ofman complexities for arbitrary degrees of m.

Table 6.2. Space and Time Complexities for Several $m = 2^k$-bit Hybrid Karatsuba-Ofman Multipliers

m	r	n	AND gates	XOR gates	Time delay	Area (in NAND units)
1	1	1	1	0	T_A	1.26
2	1	2	4	1	$T_X + T_A$	7.24
4	1	4	16	9	$2T_X + T_A$	39.96
8	2	4	48	55	$6T_X + T_A$	181.48
16	4	4	144	225	$10T_X + T_A$	676.44
32	8	4	432	799	$14T_X + T_A$	2302.12
64	16	4	1296	2649	$18T_X + T_A$	7460.76
128	32	4	3888	8455	$22T_X + T_A$	23499.88
256	64	4	11664	26385	$26T_X + T_A$	72743.64
512	128	4	34992	81199	$30T_X + T_A$	222727.72

Binary Karatsuba-Ofman Multipliers

In order to generalize the Karatsuba-Ofman algorithm of Algorithm 6.1 for arbitrary degrees m, particularly m primes, let us consider the multiplication of two polynomials $A, B \in GF(2^m)$, such that their degree is less or equal to $m-1$, where $m = 2^k + d$.

$$A = \underbrace{[\overbrace{0, \ldots, 0, 0}^{2^{k+1}-d}, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}}_{A^H}, \overbrace{a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0}^{A^L}];$$

$$A^H = [0, \ldots, 0, 0, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}];$$
$$A^L = [a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0];$$

Fig. 6.1. Binary Karatsuba-Ofman Strategy

As a very first approach, we could pretend that both operands have 2^{k+1} coordinates each, where their respective $2^{k+1} - d$ most significant bits are all equal to zero. Figure 6.1 shows how the sub-polynomials A^H and A^L will be redefined according with this approach. If we partition the operands A and B as shown in Figure 6.1, then, in order to compute their polynomial multiplication, we can use the regular Karatsuba-Ofman algorithm with $m = 2^{k+1}$. Although this approach is obviously valid, it clearly implies the waste of several arithmetic operations, as some of the most significant bits of the operands are zeroes. However, if we were able to identify the extra arithmetic operations and remove them from the computation, we would then be able to find a quasi-optimal solution for arbitrary degrees of m. To see how this can

be done, consider the Algorithm 6.2, which has been adapted from Algorithm 6.1 previously studied.

Algorithm 6.2 *mulgen_d*(C, A, B): m-bit Binary Karatsuba-Ofman Multiplier

Require: Two elements $A, B \in GF(2^m)$ with m an arbitrary number, and where A, B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L$, $B = x^{\frac{m}{2}} B^H + B^L$.

Ensure: A polynomial $C = AB$ with up to $2m-1$ coordinates, where $C = x^m C^H + C^L$.

1: $k = \lfloor \log_2 m \rfloor$;
2: $d = m - 2^k$;
3: **if** $d == 0$ **then**
4: $\quad C = Kmul2^k(A, B)$;
5: $\quad$ **return**(C);
6: **end if**
7: **for** i from 0 to $d-1$ **do**
8: $\quad M_{Ai} = A_i^L + A_i^H$;
9: $\quad M_{Bi} = B_i^L + B_i^H$;
10: **end for**
11: $mul2^k(C^L, A^L, B^L)$;
12: $mul2^k(M, M_A, M_B)$;
13: $mulgen\_d(C^H, A^H, B^H)$;
14: **for** i from 0 to $2^k - 2$ **do**
15: $\quad M_i = M_i + C_i^L + C_i^H$;
16: **end for**
17: **for** i from 0 to $2^k - 2$ **do**
18: $\quad C_{k+i} = C_{k+i} + M_i$;
19: **end for**
20: **Return**(C).

In lines 1-2 the values of the constants k, d such that $m = 2^k + d$, are computed. If $d = 0$, i.e, if m is a power of two, then the binary Karatsuba-Ofman Algorithm 6.2 reverts to the specialized Algorithm 6.1 presented previously. If that is not the case, Algorithm 6.2 uses the constants k and d to prevent us to compute unnecessary arithmetic operations. In lines 6-8, the d least significant bits of M_A and M_B of equation (6.9) are computed using the d non-zero coordinates of A^H and B^H. The remaining $k-d$ most significant bits of M_A and M_B are directly obtained from A^L and B^L, respectively. Notice that the operands, A^L, B^L, M_A and M_B are all 2^k-bit polynomials. Because of that, our algorithm invokes the multiplier of Algorithm 6.1 in lines 9 and 10. On the other hand, both operands A^H and B^H are d-bit polynomials, where d, in general, is not a power of two. Consequently, in line 11, the algorithm calls itself in a recursive manner. This recursive call is invoked using the operand's degree reduced to d. In each iteration the degree of the operands gets reduced,

and eventually, after a total of h iterations (where h is the hamming weight of the binary representation of the original degree m), the algorithm ends.

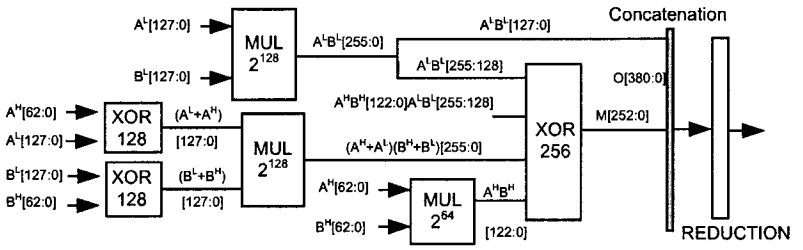

Fig. 6.2. Karatsuba-Ofman Multiplier $GF(2^{191})$

As a design example, consider the binary Karatsuba-Ofman multiplier shown in Figure 6.2. That circuit computes the polynomial multiplication of the elements A and $B \in GF(2^{191})$. Notice that for this case $m = 191 = 2^k + d = 2^7 + 63$. Since $(191)_2 = 10111111$, the Hamming weight h of the binary representation of m is $h = 7$. This implies that we would need a total of seven iterations in order to compute the multiplication using the generalized m-bit binary Karatsuba-Ofman multiplier.

However we can do much better by assuming that the d = 63 most significant leftover bits are 64 (implying $m = (192)_2 = 11000000$). Hence, algorithm 6.2 can finish the computation in only two iterations, as shown in Figure 6.2.

By using the complexity figures listed in Table 6.2, we can estimate the space and time complexities of the generalized 191-bit binary Karatsuba-Ofman multiplier as,

$$\begin{aligned} \text{Number of CLBs} &= 2MUL_X(128) + MUL_X(64) + C \\ &= 2 \cdot 3379 + 1171 + C \\ &= 7929 + C \\ \text{Delay} &= MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + O \\ &= MUL_{delay}(2^{\lfloor \log_2 191 \rfloor}) + O \\ &= MUL_{delay}(2^7) + O \end{aligned} \tag{6.15}$$

Where C and O represent the overhead in space and time, respectively, associated with the extra circuitry shown in Figure 6.2.

The generalized 191-bit binary Karatsuba-Ofman multiplier was implemented using Xilinx Foundation Series F4.1i software on Xilinx Virtex-E FPGA device XCV2600e-8bg560. The design is coded using VHDL, using library components and also by using Xilinx Coregenerator for design entry. The implementation occupied a total of 8721 slices and 576 I/O Blocks. We obtained a total path delay of 43 ηSec.

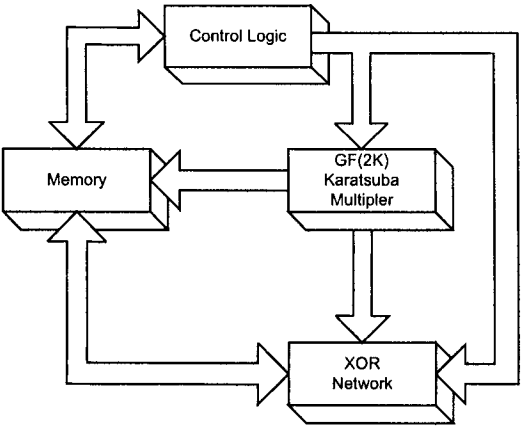

Fig. 6.3. Programmable Binary Karatsuba-Ofman Multiplier

Programmability

The schematic diagram shown in figure 6.2 illustrates two desirable characteristics of the binary Karatsuba-Ofman multipliers. First, it is possible to implement them using non-recursive architectures. In addition, since these algorithms are highly modular, it is possible to design non-parallel scalable implementations. By scalable implementations we mean configurations that allow the user to select the size m of the multiplicands that he/she wants to work with.

Consider the architecture shown in figure 6.3. We use a control logic block that allows us to execute the algorithm of figure 6.2 in a sequential manner. To do this, we also take advantage of the intrinsically modular nature of a 2^k-bit Karatsuba-Ofman multiplier, which can itself be programmed to compute multiplications that involve operands of a size that is any power of two lower than 2^k.

The partial multiplications obtained using this approach, are stored in a memory block as figure 6.3 shows. The control logic can then use these partial results to compute the remaining operations so that the total polynomial product can be obtained. Notice also, that the architecture shown in figure 6.3 can be programmed to implement multiplications with different operands' sizes.

6.1.3 Squaring

In this section we investigate some efficient methods to compute polynomial squaring, which is a special case of polynomial multiplication. Let us assume

that we have an element A given as $A = \sum_{i=0}^{m-1} a_i x^i$. Then the square of A is given as

$$C(x) = A(x)A(x) = A^2(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) = \sum_{i=0}^{m-1} a_i x^{2i}. \quad (6.16)$$

The main implication of the above equation is that the first $k < m$ bits of A completely determine the first $2k$ bits of A^2. Notice also that half the bits of A^2 (the odd ones) are zeroes. Taking advantage of this feature, the hardware implementation shown in Figure 6.4 simply interleaves a zero value between each one of the original bits of A yielding the required squaring computation which must be followed by a reduction operation to be discussed in the next Subsection.

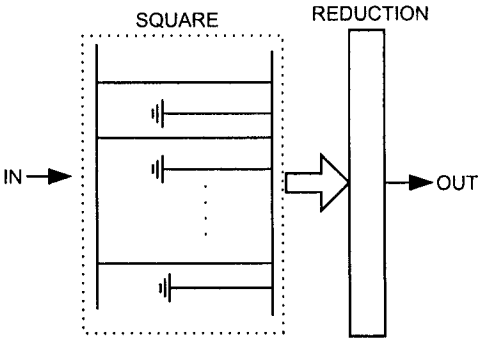

Fig. 6.4. Squaring Circuit

6.1.4 Reduction

Let the field $GF(2^m)$ be constructed using the irreducible polynomial $P(x)$ and let $A(x), B(x) \in GF(2^m)$. Assuming that we already have computed the product polynomial $C(x)$ of Equation (6.1), by using any one of the methods described in the previous two subsections, we want to obtain the modular product C' of Equation (6.2). Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{aligned} C &= [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' &= [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{aligned} \quad (6.17)$$

Once the generating polynomial $P(x)$ has been selected, the reduction step that obtains C' from C can be computed by using XOR and shift operations only.

Reduction with Trinomials and Pentanomials

Let the field $GF(2^m)$ be constructed using the irreducible trinomial $P(x) = x^m + x^n + 1$ with a root α and $1 < n < \frac{m}{2}$. Let also $A(x), B(x)$ be elements in $GF(2^m)$. In order to obtain the modular product $C'(x)$ of (6.1), we use the property $P(\alpha) = 0$, and write

$$\begin{aligned} \alpha^m &= 1 + \alpha^n \ ; \\ \alpha^{m+1} &= \alpha + \alpha^{n+1} \ ; \\ &\vdots \\ \alpha^{2m-3} &= \alpha^{m-3} + \alpha^{m+n-3} \ ; \\ \alpha^{2m-2} &= \alpha^{m-2} + \alpha^{m+n-2} \ . \end{aligned} \tag{6.18}$$

The above $m-1$ set of identities suggests a method to obtain the m-coordinates of the modular product C' of Equation (6.2). We can partially reduce the $2m-1$ coordinates of C by reducing its most significant $m-1$ bits into its first $m+n-1$ bits, as indicated by (6.18). For instance, in order to obtain the first partially reduced coordinate c'_0 we just need to add the regular product coordinate c_m to the c_0 coordinate, yielding c'_0 as $c'_0 = c_0 + c_m$.

Similarly the whole set of $m+n-2$ partially reduced coordinates can be found as,

$$\begin{aligned} c'_0 &= c_0 + c_m \ ; \\ c'_1 &= c_1 + c_{m+1} \ ; \\ &\vdots \\ c'_{n-1} &= c_{n-1} + c_{m+n-1} \ ; \\ c'_n &= c_n + c_{m+n} + c_m \ ; \\ c'_{n+1} &= c_{n+1} + c_{m+n+1} + c_{m+1} \ ; \\ &\vdots \\ c'_{m-2} &= c_{m-2} + c_{2m-2} + c_{2m-n-2} \ ; \\ c'_{m-1} &= c_{m-1} + c_{2m-n-1} \ ; \\ c'_m &= c_m + c_{2m-n} \ ; \\ &\vdots \\ c'_{m+n-3} &= c_{m+n-3} + c_{2m-3} \ ; \\ c'_{m+n-2} &= c_{m+n-2} + c_{2m-2} \ . \end{aligned} \tag{6.19}$$

Notice that in the reduction process of (6.19), the constant coefficient of the irreducible generating trinomial $P(x)$ reflects its influence in the first $m-1$ partially reduced bits. The middle term of $P(x)$, on the other hand, affects the partially reduced bits of (6.19) in the range $[c'_n, c'_{m+n-2}]$.

We say that the coefficients in (6.19) have been partially reduced because in general, if $n > 1$, we still need to reduce the $n-1$ most significant reduced coordinates of (6.19). However, this same idea can be used repeatedly until the $m-1$ modular coordinates of (6.17) are obtained. Each time that this strategy is applied we reduce $m-n$ coordinates.

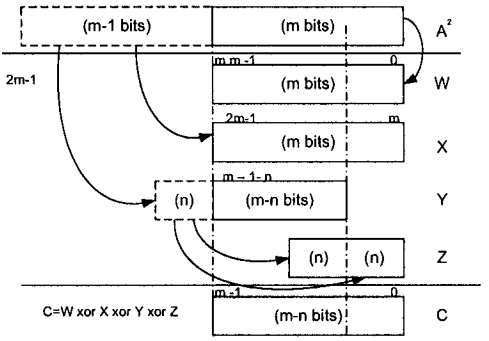

Fig. 6.5. Reduction Scheme

For hardware reconfigurable designs, we can implement above ideas as follows. According to Eq. (6.1) the polynomial product $C(x) = A(x)B(x)$, can be represented as a $2m$-coefficient vector as,

$$C(x) = \left[c'_{2m-1}\ c'_{m-2} \cdots c'_{m-1}\ c'_m\ ;\ c'_{m-1}\ c'_2 \cdots c'_1\ c'_0\right] \tag{6.20}$$

When working with an irreducible trinomial of the form $P(x) = x^m + x^n + 1$, it is convenient to consider the following four sub-vectors,

$$\begin{aligned} C' &= A \cdot B \bmod P(x) \\ &= C'_{[0,m-1]} + C'_{[m,2m-1]} + C'_{[m,2m-1-n]}x^n \\ &\quad + \left(C'_{[2m-n,2m-1]} + C'_{[2m-n,2m-1]}x^n\right) \end{aligned} \tag{6.21}$$

Thus, the reduction step can be computed by the addition of four terms,

$$\begin{aligned} W &= C'_{[0,m-1]} \\ X &= C'_{[m,2m-1]} \\ Y &= C'_{[m,2m-1-n]}x^n \\ Z &= C'_{[2m-n,2m-1]} + C'_{[2m-n,2m-1]}x^n \end{aligned}$$

This procedure is shown schematically in Fig. 6.5. Notice that for those designs implemented in hardware platforms, the modular reduction procedure just

outlined can be instrumented by using XOR logic gates only. Nevertheless, the exact computational complexity of this arithmetic operation depends on the explicit form of m and the middle coefficient n in the trinomial $P(x)$.

Although the strategy shown in Figure 6.5 has been designed for irreducible trinomials, it can be easily extended to irreducible pentanomials. For example, let us consider the finite field GF(2^{163})generated using the irreducible pentanomial $P(x) = x^{163} + x^7 + x^6 + x^3 + 1$ [4]. The corresponding reduction procedure for this pentanomial is depicted in Fig. 6.6.

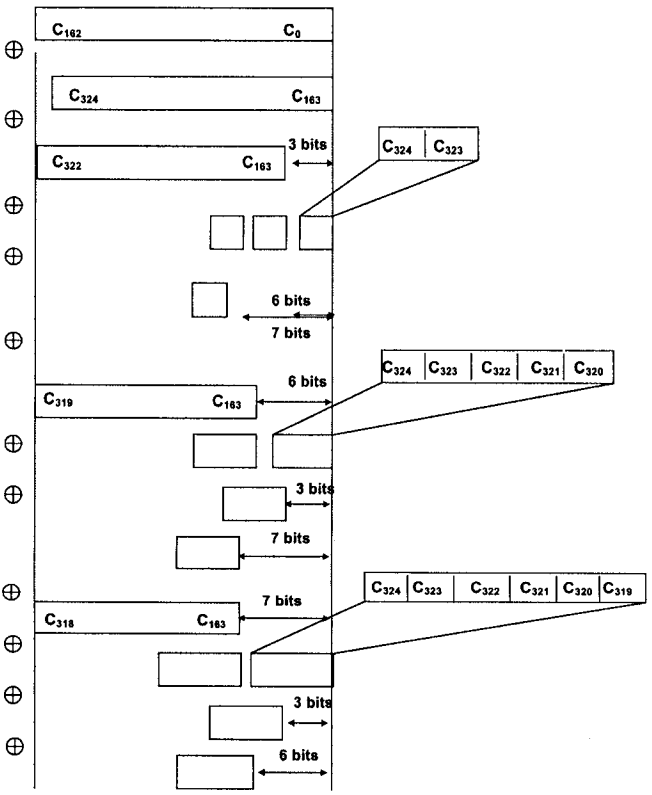

Fig. 6.6. Pentanomial Reduction

[4] This is a NIST recommended finite field for elliptic curve applications [253].

6.1.5 Modular Reduction with General Polynomials

The algorithms studied in the previous section are highly efficient for irreducible trinomials and/or pentanomials. However, when general irreducible polynomials are selected, i.e., irreducible polynomials with an arbitrary number of nonzero coefficients, the algorithms presented in last section are not efficient anymore. Because of that, we need to come out with alternative techniques to handle the reduction step. In this section we present a standard reduction method based in look-up tables specifically intended for general irreducible polynomials.

Recall that assuming that the polynomial product C with $2m-1$ coordinates is given, we would like to compute

$$C'(x) = C(x) \bmod P(x).$$

Our methods are based on the observation that since we are interested in the polynomial remainder of the above equation, we can safely add any multiple of $P(x)$ to $C(x)$ without altering the desired result. This simple observation suggests the following algorithm that can reduce k bits of the polynomial product C at once. Assume that the $m+1$ and $2m-1$ coordinates of P(x) and C(x), respectively, are distributed as follows:

$$\begin{array}{l} C = [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_1, c_0]; \\ P = [p_m, p_{m-1}, \ldots, p_1, p_0]. \end{array} \tag{6.22}$$

Then, there always exists a k-bit constant scalar S, such that

$$\begin{array}{rl} P = [& p_m,\ p_{m-1}, \ldots,\ p_{m-k+1}, p_{m-k}, \ldots, p_1, p_0]; \\ S \cdot P = [& c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, p'_{m-k}, \ldots, p'_1, p'_0]; \end{array} \tag{6.23}$$

where $1 \leq k \leq m-1$. Notice that all the most significant k bits of the scalar multiplication $S \cdot P$ become identical to the corresponding ones of the number C. By left shifting the number $S \cdot P$ exactly $Shift = 2m-2-k-1$ positions, we can effectively reduce the number in C by k bits as shown in figure 6.7.

$$\begin{array}{rl} C & [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, c_{2m-2-k}, \ldots, c_{m-2}, c_{m-3}, \ldots, c_0] + \\ 2^{Shift}(S \cdot P) & [c_{2m-2}, c_{2m-3}, \ldots, c_{2m-1-k}, p'_{m-k}, \ldots, p'_0, 0, \ldots, 0] = \\ \hline & [0, 0, \ldots, 0, c'_{2m-k}, \ldots, c'_{m-2}, c_{m-3}, \ldots, c_0] \end{array}$$

Fig. 6.7. A Method to Reduce k Bits at Once

One can apply this strategy an appropriate number of times in order to reduce all the most $m-1$ significant coordinates of C.

In summary, the main design problems that we need to solve in order to implement the reduction method discussed here are:

- Given C and P as in (6.22), find the appropriate constant S that yields the most significant k bits of the operation SP, identical to the corresponding ones in C.
- Compute the scalar multiplication $S \cdot P$ of (6.23).
- Left shift the number $S \cdot P$ by $Shift$ positions, so that the result of the polynomial addition $C + 2^{Shift}(S \cdot P)$ ends up having k leading zeroes.

Both of the first two design problems, i.e., finding the constant S and computing the scalar product $S \cdot P$, can be solved efficiently by using a look-up table approach, provided that a moderated value of k be selected. In practice, we have found that a selection of $k = 8$ yields a reasonable memory/speed trade-off in the performance of the algorithm.

For all the 2^k different values that the k most significant bits of C can possibly take, we want to guarantee that the k most significant bits of the operation SP are identical to those of C. Hence, once that k has been fixed, we need to find a set of 2^k different scalars satisfying that requirement.

Algorithm 6.3 presents a method that, given the irreducible polynomial P and its degree m and the selected value of k, constructs a table containing all the 2^k scalars needed to obtain the required result.

Algorithm 6.3 Constructing a Look-Up Table that Contains All the 2^k Possible Scalars in Equation (6.23)

Require: The irreducible polynomial P; its degree m; and k, the number of bits to be reduced at once.
Ensure: A table *highdivtable* with 2^k scalars.
1: $highdivtable = 0$;
2: $N = 2^k - 1$;
3: $PMSBk = P_m P_{m-1} \ldots P_{m-k+1}$;
4: **for** i from 0 to N **do**
5: $A = Dec2Bin(i)$;
6: **for** j from 0 to k-1 **do**
7: **if** $A_j = 1$ **then**
8: $A = A + RightShift(PMSBk, j)$;
9: $highdivtable[i] = highdivtable[i] + 2^{k-1-j}$;
10: **end if**
11: **end for**
12: **end for**
13: **Return** (highdivtable)

The Algorithm 6.3 finds all the 2^k scalars needed by *reducing* each one of them using the k most significant bits of the irreducible polynomial P. For convenience, these bits are stored in the variable $PMSBk$ (see step 3 of Algorithm 6.3). Steps 4-9 find the appropriate scalar S for each one of all the 2^k possible values that the k MSB of C can take.

In line 5 the value of C to be processed is translated to its binary representation and stored in the temporary variable A. Then, in lines 6-9 each one of the k bits of A is scanned and reduced, if necessary, by using an appropriate shift version of $PMSBk$. Finally, in line 9 the $k-1-j$-th bit of the i-th entry in table *highdivtable* is set. At the end of the inner loop in lines 6-9, the i-th entry of *highdivtable* contains the scalar S that would obtain the result in (6.23), if the k most significant bits of C where equal to the number in A.

In order to compute the scalar multiplication $S \cdot P$ of (6.23), we use once again a look-up table approach as shown in Algorithm 6.4.

Algorithm 6.4 Generating a Look-Up Table that Contains All the 2^k Possible Scalars Multiplications $S \cdot P$

Require: The irreducible polynomial P; and k, the number of bits to be reduced at once.
Ensure: A table *Paddedtable*, with all the 2^k $S \cdot P$ possible products.
1: **for** i from 0 to k-1 **do**
2: Pshift[i] = LeftShift(P, i);
3: **end for**
4: $N = 2^k - 1$;
5: **for** i from 0 to N **do**
6: $S = Dec2Bin(i)$;
7: **for** j from 0 to k-1 **do**
8: **if** $S_j = 1$ **then**
9: $Paddedtable[i] = Paddedtable[i] + Pshift[k]$;
10: **end if**
11: **end for**
12: **end for**
13: **Return** (Paddedtable)

The algorithm in 6.4 is quite similar to Algorithm 6.3. In order to obtain all the 2^k scalar products of the irreducible polynomial P, the above algorithm finds first in lines 1-2 all the first 2^j multiples of P for $j = 0, 1, \ldots, k-1$. Then, in lines 4-9 all the 2^k scalars S are examined one by one and bit by bit, so that the scalar product $i \cdot P$ is stored in the i-th entry of the table *Paddedtable* for $i = 0, 1, \ldots, N = 2^k - 1$. Notice that each entry of *Paddedtable* has a size of $m + k$ bits, where m is the degree of the irreducible polynomial P.

Using the two look-up tables generated by Algorithms 6.3 and 6.4, we can easily obtain the modular reduction of the polynomial C by repeatedly implementing the operation $C + 2^{Shift}(S \cdot P)$.

Consider now Algorithm 6.5, where it has been assumed that the tables *Highdivtable* and *Paddedtable* have been previously computed and are available.

First, in line 1 given k and the degree m of the irreducible polynomial P, the number of iterations is computed and stored in the variable N_k. In line 2 it

Algorithm 6.5 Modular Reduction Using General Irreducible Polynomials

Require: The degree m of the irreducible polynomial; the operand C to be reduced; and k the number of bits that can be reduced at once.
Ensure: The field polynomial defined as $C = C \bmod P$, with a length of m bits.
1: $N_k = \lceil \frac{m-1}{k} \rceil$;
2: $shift = 2m - 2 - k - 1$;
3: **for** i from 0 to N_k **do**
4: $A = C_{n-k \cdot i} C_{(n-k \cdot i)-1} \ldots C_{(n-k \cdot i)-k+1}$;
5: $S = Highdivtable[A]$;
6: $Pshifted = LeftShift(Paddedtable[S], shift)$;
7: $C = C + Pshifted$;
8: $shift = shift - k$;
9: **end for**
10: **Return** C

is computed the amount of shift needed to apply properly the method outlined in figure 6.7. Then, in each iteration of the loop in lines 3-9, k bits of C are reduced. In line 4 the k bits of C to be reduced are obtained. This information is used in line 5 to compute the appropriate scalar S needed to obtain the result of equation (6.23). In line 6 the S-th entry of the table *Paddedtable* is left shifted $shift$ positions so that in line 7 the operation $C + 2^{Shift}(S \cdot P)$ can be finally computed allowing the effective reduction of k bits at once. Then, in line 8 the variable $shift$ is updated in order to continue the reduction process.

Algorithm 6.5 performs a total of $N_k = \lceil \frac{m-1}{k} \rceil$ iterations. At each iteration of the algorithm the look-up tables *Highdivtable* and *Paddedtable* are accessed once each. In line 7, and XOR addition is executed, implying that the complexity cost of the general reduction method discussed in this section is given as,

$$\begin{aligned} \text{Additions} &= 2N_k, \\ \text{Look-up table size (in bits)} &= 2^k(m + 2k) \ . \end{aligned} \tag{6.24}$$

6.1.6 Interleaving Multiplication

In this Subsection we discuss one of the simplest and most economical binary field multiplier schemes: the serial interleaving multiplication algorithm.

Multiplication by a Primitive Element

Let $P(x) = p_0 + p_1 x + p_1 x^2 + \ldots + p_{m-1} x^{m-1} + x^m$ be an m-degree irreducible polynomial over $GF(2)$. Let also α be a root of $p(x)$, i.e., $p(\alpha) = 0$. Then, the set $\{1, \alpha, \alpha^2, \ldots, \alpha^{m-1}\}$ is a basis for $GF(2^m)$, commonly called the polynomial (canonical) basis of the field [221]. An element $A \in GF(2^m)$ is expressed in this basis as $A = \sum_{i=0}^{m-1} a_i \alpha^i$. Let $A(\alpha)$ be an arbitrary element of $GF(2^m)$.

Then, the product $C = \alpha \cdot A(\alpha)$ can be expressed as,

$$C = \alpha \left(a_0 + a_1\alpha + \ldots + a_{m-1}\alpha^{m-1}\right) = a_0\alpha + a_1\alpha^2 + \ldots + a_{m-1}\alpha^m. \quad (6.25)$$

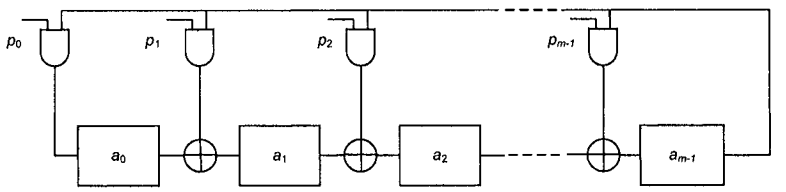

Fig. 6.8. $\alpha \cdot A(\alpha)$ Multiplication

Using the fact that α is a primitive root of the irreducible polynomial, we can write,

$$\alpha^m = p_0 + p_1\alpha + \ldots + p_{m-1}\alpha^{m-1}. \quad (6.26)$$

Substituting Eq. (6.26) into Eq. (6.25) we obtain,

$$C = c_0 + c_1\alpha + \ldots + c_{m-1}\alpha^{m-1},$$

where, $c_0 = a_{m-1}p_0$ and

$$d_i = a_{i-1} + a_{m-1}p_i,$$

for $i = 1, \ldots, m-1$. A realization of the above operation is shown in Fig. 6.8. The main building block is an m-tap LFSR register. That register is initially loaded with the m coordinates of the field element A, namely, $(a_0, a_1, a_2, \ldots, am-1)$. The signals p_i represent the coefficients of the irreducible polynomial. Notice that whenever a given polynomial coefficient is on, i.e., $p_i = 1$, then the corresponding branch of the circuit will be a *short circuit*. Otherwise, if $p_i = 0$ the branch acts as an *open circuit*. After m clock cycles, the new register content will be the value of the field element C.

Serial Multiplication

Using the multiplication procedure outlined above, the multiplication of two arbitrary field elements can be accomplished by using a procedure inspired in the well-know Horner's scheme.

Let us consider two arbitrary field elements A and B expressed in polynomial basis as,

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

Then, the product of $A \cdot B$ can be expressed as,

$$\begin{aligned} C(\alpha) &= A(\alpha)B(\alpha) \bmod P(\alpha) \\ &= A(\alpha)\left(\sum_{i=0}^{m-1} b_i\alpha^i\right) \bmod P(\alpha) \\ &= \left(\sum_{i=0}^{m-1} b_i A(\alpha)\alpha^i\right) \bmod P(\alpha) \end{aligned}$$

Therefore,

$$C(\alpha) = \left(b_0A(\alpha) + b_1A(\alpha)\alpha + b_2A(\alpha)\alpha^2 + \ldots + b_{m-1}A(\alpha)^{m-1}\right) \bmod P(\alpha).$$

Algorithm 6.6 shows the standard procedure for computing above equation using Horner's rule.

Algorithm 6.6 LSB-First Serial/Parallel Multiplier

Require: An irreducible polynomial $P(\alpha)$ of degree m, two elements $A, B \in GF(2^m)$.
Ensure: $C(\alpha) = A(\alpha)B(\alpha) \bmod P(\alpha)$.
1: $C = 0$;
2: **for** $i = 0$ to $m - 1$ **do**
3: $C = b_iA + C$;
4: $A = A\alpha^i \bmod P(\alpha)$;
5: **end for**
6: **Return(C)**.

The multiplier realization of Algorithm 6.6 is shown in Fig. 6.9. The architecture shown in Fig. 6.9 consists of two LFSR Register plus extra circuitry. As it was mentioned previously, the signals p_i in the first LFSR block represent the coefficients of the irreducible polynomial, and their values (either ones or zeroes) determine the LFSR structure. Furthermore, a gate array is included in order to compute the multiplication operation as is explained below. Initially the register C is set to zero, whereas the register in the upper part of Fig. 6.9 is loaded with the m coefficients of the field element A. Thereafter, when the clock signal is applied to the registers, the value of $A\alpha$ is generated. Then, B coefficients, namely, $b_0, b_1, b_2, \ldots, b_{m-1}$ are serially introduced in that order, thus generating the values $b_iA\alpha^i$, for $i = 0, 1, \ldots, m-1$, which are accumulated in register C until all the m product coefficients $c_0, c_1, c_2, \ldots, c_{m-1}$ are collected.

6.1.7 Matrix-Vector Multipliers

The GF(2^m) multiplication given by (6.1) can be described in terms of matrix-vector operations. There are mainly two different approaches based on matrix vector operations to compute a field product:

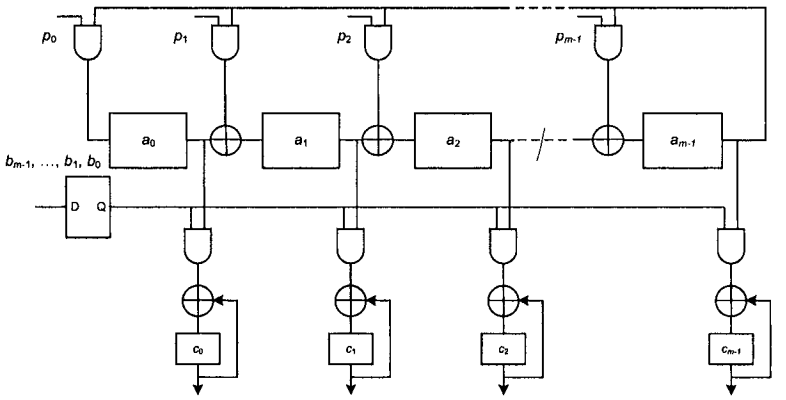

Fig. 6.9. LSB-First Serial/Parallel Multiplier

1. The polynomial multiplication part is performed by any method. Then, the resulting product is reduced by using a reduction matrix.
2. The polynomial multiplication and modular reduction parts are performed in a single step by using the so-called Mastrovito matrix.

Let $a(x)$ and $b(x)$ denote two degree m polynomials representing the elements in GF(2^m). Let $c(x) = a(x)b(x) \bmod P(x)$ denote their field product. The coefficient vectors of these polynomials are given by

$$\begin{aligned}\mathbf{a} &= [a_0, a_1, \cdots, a_{m-1}]^T \\ \mathbf{b} &= [b_0, b_1, \cdots, b_{m-1}]^T \\ \mathbf{c} &= [c_0, c_1, \cdots, c_{m-1}]^T.\end{aligned}$$

Also, let us define the polynomials

$$\begin{aligned}d(x) &= a(x)b(x) = d_0 + d_1 x + \cdots + d_{2m-2}x^{2m-2}\ , \\ d^{(L)}(x) &= d_0 + d_1 x + \cdots + d_{m-1}x^{m-1}\ , \\ d^{(H)}(x) &= d_m + d_{m+1}x + \cdots + d_{2m-2}x^{m-2}\ .\end{aligned} \tag{6.27}$$

The coefficient vectors representing these polynomials are

$$\begin{aligned}\mathbf{d} &= [d_0, d_1, \cdots, d_{2m-2}]^T\ , \\ \mathbf{d}^{(L)} &= [d_0, d_1, \cdots, d_{m-1}]^T\ , \\ \mathbf{d}^{(H)} &= [d_m, d_{m+1}, \cdots, d_{2m-2}]^T\ .\end{aligned}$$

The work in [284] reduces the polynomial multiplication $d(x)$ using an $(m \times m-1)$ reduction matrix $\mathbf{Q}$ to obtain the field product $c(x)$ as below:

$$\mathbf{c} = \mathbf{d}^{(L)} + \mathbf{Q} \cdot \mathbf{d}^{(H)} \ . \tag{6.28}$$

Mastrovito Multiplier

The so-called Mastrovito matrix is constructed from the coefficients of the first multiplicand and the irreducible polynomial defining the field. Then, the polynomial multiplication and modulo reduction steps are performed together using this matrix. The papers [351, 128, 401] follow the Mastrovito multiplication scheme outlined below,

$$\mathbf{c} = \mathbf{M} \cdot \mathbf{b} \ , \tag{6.29}$$

where $\mathbf{M}$ is the $(m \times m)$ Mastrovito matrix whose entries are the function of the coefficients of $a(x)$ and $P(x)$. The Mastrovito matrix $\mathbf{M}$ is related to the reduction matrix $\mathbf{Q}$ by

$$\mathbf{M} = \mathbf{L} + \mathbf{Q} \cdot \mathbf{U} \ , \tag{6.30}$$

where $\mathbf{L}$ and $\mathbf{U}$ are the following $(m \times m)$ and $(m-1 \times m)$ matrices:

$$\mathbf{L} = \begin{bmatrix} a_0 & 0 & 0 & \cdots & 0 & 0 \\ a_1 & a_0 & 0 & \cdots & 0 & 0 \\ a_2 & a_1 & a_0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & a_{m-4} & \cdots & a_0 & 0 \\ a_{m-1} & a_{m-2} & a_{m-3} & \cdots & a_1 & a_0 \end{bmatrix} \ ,$$

$$\mathbf{U} = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \cdots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \cdots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \cdots & 0 & a_{m-1} \end{bmatrix} \ . \tag{6.31}$$

This is because $d(x) = a(x)b(x)$ can be given in the vector notation by

$$\mathbf{d} = \begin{bmatrix} \mathbf{d}^{(L)} \\ \mathbf{d}^{(H)} \end{bmatrix} = \begin{bmatrix} \mathbf{L} \cdot \mathbf{b} \\ \mathbf{U} \cdot \mathbf{b} \end{bmatrix} \ .$$

Then, $\mathbf{c} = \mathbf{d}^{(L)} + \mathbf{Q} \cdot \mathbf{d}^{(H)} = \mathbf{L} \cdot \mathbf{b} + \mathbf{Q} \cdot \mathbf{U} \cdot \mathbf{b} = (\mathbf{L} + \mathbf{Q} \cdot \mathbf{U}) \cdot \mathbf{b} = \mathbf{M} \cdot \mathbf{b}$.

The Mastrovito and the reduction matrices are studied thoroughly in [284, 401] for various types of irreducible polynomials. In [351] a comprehensive study of the Mastrovito multiplier for irreducible trinomials was presented. Authors in [401] proposed a practical and systematic design approach for a general Mastrovito multiplier. In [388] it was shown that non-Mastrovito multipliers using direct modular reduction also provide competitive performance. Moreover, efficient non-Mastrovito multipliers for irreducible trinomials were also proposed.

6.1.8 Montgomery Multiplier

In this section we explain the Montgomery multiplication method in GF(2^m). Once again, let $P(x)$ be an irreducible polynomial over $GF(2)$ that defines the field GF(2^m). Rather than computing Eq.(6.1), the Montgomery multiplication calculates

$$C(x) = A(x)B(x)R^{-1}(x) \mod P(x) \tag{6.32}$$

where $R(x)$ is a fixed element and $gcd(R(x), P(x)) = 1$.

Because of Bezout's identity[5], one can find two polynomials $R^{-1}(x)$ and $P'(x)$ such that

$$R(x)R^{-1}(x) + P(x)P'(x) = 1 \tag{6.33}$$

where $R^{-1}(x)$ is the inverse of $R(x)$ modulo $P(x)$. These two polynomials can be calculated with the extended Euclidean algorithm. Koç and Acar [182, 388] selected $R(x) = x^m$ for high performance modular reduction in the Montgomery multiplication algorithm, which can be given as follows:

Algorithm 6.7 Montgomery Modular Multiplication Algorithm

Require: $A(x), B(x), R(x), P'(x)$
Ensure: $C(x) = A(x)B(x)R^{-1}(x) \mod P(x)$
1: $T(x) = A(x)B(x)$;
2: $U(x) = T(x)\ P'(x) \mod R(x)$;
3: $C(x) = [T(x) + U(x)P(x)]/R(x)$;
4: **Return** C

To prove the correctness of this algorithm we note that Step 2 implies that there exists a polynomial

$$U(x) = T(x)\ P'(x) + H(x)R(x)\ . \tag{6.34}$$

We write $C(x)$ in Step 3 by using (6.34) as follows:

$$\begin{aligned} C(x) &= \tfrac{1}{R(x)}[T(x) + T(x)\ P'(x)\ P(x) + H(x)R(x)\ P(x)] \\ &= \tfrac{1}{R(x)}[T(x)(1 + P'(x)\ P(x)) + H(x)R(x)\ P(x)]\ . \end{aligned}$$

From (6.33), we can write $1 + P(x)P'(x) = R(x)R^{-1}(x)$ and substitute it into our last expression

$$\begin{aligned} C(x) &= \tfrac{1}{R(x)}[T(x)R(x)R^{-1}(x) + H(x)R(x)\ P(x)] \\ &= T(x)R^{-1}(x) + H(x)\ P(x) \\ &= A(x)B(x)R^{-1} \mod P(x)\ . \end{aligned}$$

[5] For more details on Bezout's identity the reader is refer to §6.3.1.

The degree of $C(x)$ can be verified from Step 3 as follows:

$$\begin{aligned} deg[C(x)] &\leq max\{deg[T(x)], deg[U(x)] + deg[P(x)]\} - deg[R(x)] \\ &\leq max\{2m-2, deg[R(x)] - 1 + m\} - deg[R(x)] \\ &\leq max\{2m-2-deg[R(x)], m-1\} \ . \end{aligned}$$

Then, it can be concluded that $deg[C(x)] \leq m-1$, if $deg[R(x)] \geq m-1$. If we choose $R(x) = x^m$, the result $C(x)$ will be of degree $m-1$ at most.

It can be shown [182] that Algorithm 6.7 has an associated computational cost of $2m^2$ coefficient multiplications (ANDs) and $2m^2 - 3m - 1$ coefficient additions (XORs), whereas the total time complexity is $3T_A + (2\lceil \log_2 m \rceil + \lceil \log_2 (m-1) \rceil) T_X$.

6.1.9 A Comparison of Field Multiplier Designs

Table 6.3. Fastest Reconfigurable Hardware $GF(2^m)$ Multipliers

Work	Platform	Field	Cost	Cycles	$timings$	$\frac{bits}{Slices \times timings}$
KOM variant by [47], implemented by [326]	Virtex 2	$GF(2^{163})$	5307 CLBs	1	$12.56\eta S$	2.445M
KOM variant by [85], implemented by [326]	Virtex 2	$GF(2^{163})$	5409 CLBs	1	$13.37\eta S$	2.254M
KOM variant by [293], implemented by [326]	Virtex 2	$GF(2^{163})$	5840 CLBs	1	$14.73\eta S$	1.895M
KOM [106]	Virtex 2	240 bits	1480 CLBs	30	$378\eta S$	0.429M
Recursive Classical [106]	Virtex 2	240 bits	1582 CLBs	56	$523\eta S$	0.290M
KOM [117]	Virtex 2	240 bits	1660 CLBs	54	$655\eta S$	0.221M
Massey-Omura [118]	Virtex 2	240 bits	36857 LUTs	50	$800\eta S$	0.0336M (est.)

In this Subsection we compare some of the most representative designs of $GF(2^m)$ multipliers considering three metrics: speed, compactness and efficiency. Table 6.3 shows the fastest designs reported to date for $GF(2^m)$ field multiplication. It can be observed that Karatsuba-ofman Multipliers (KOM) are much faster than other schemes such as recursive classical multiplier or Massey-Omura scheme. This can be explained from the theoretical point of view from the fact that KOM algorithms enjoy of a sub-quadratic complexity.

In Table 6.4 we show a selection of some of the most compact reconfigurable hardware multiplier designs. It is noted that this category is dominated by the interleaved and Montgomery multiplier schemes.

Table 6.4. Most Compact Reconfigurable Hardware $GF(2^m)$ Multipliers

Work	Platform	Field	Cost	Cycles	$timings$	$\frac{bits}{Slices \times timings}$
Interleaved [104]	Virtex	$GF(2^{239})$	359 CLBs	239	3.1μS	0.215M
Montgomery [97]	Virtex	$GF(2^{233})$	425 CLBs (est)	466	2.81μS	0.195M
Class.+Montg. [18]	Virtex	$GF(2^{160})$	1049 CLBs	80	1.11μS	0.137M
Montgomery [18]	Virtex	$GF(2^{160})$	1427 CLBs	160	1.66μS	0.0675M
Interleaved [266]	Virtex	$GF(2^{210})$	420 CLBs (est)	210	12.3μS	0.042M

We measure efficiency by taking the ratio of number of bits processed over slices multiplied by the time delay achieved by the design, namely,

$$\frac{bits}{Slices \times timings}$$

For instance, consider the KOM variant design proposed by [47] and implemented by [326]. As is shown in Table 6.3, working over $GF(2^{163})$, that design achieved a time delay of just 12.56ηS at a cost of 5307 slices. Therefore its efficiency is calculated as,

$$\frac{bits}{Slices \times timings} = \frac{163}{5307 \times 12.56\eta} = 2.445M$$

When comparing the designs featured in Tables 6.3 and 6.4, it is noticed that the most efficient multiplier designs are the Karatsuba-Ofman multipliers variants as they were reported in [47, 85, 293]. This is a quite remarkable feature, which implies that the Karatsuba-Ofman multipliers represent both, the fastest and the most efficient of all multiplier designs studied in this Chapter.

6.2 Field Squaring and Field Square Root for Irreducible Trinomials

Let us consider binary extension fields constructed using irreducible trinomials of the form $P(x) = x^m + x^n + 1$, with $m \geq 2$. It is convenient to consider, without loss of generality, the additional restriction $1 \leq n \leq \lfloor \frac{m}{2} \rfloor$ [6].

[6] It is known that if $P(x) = x^m + x^n + 1$ is irreducible over $GF(2)$, so is $P(x) = x^m + x^{m-n} + 1$[228]. Hence, provided that at least one irreducible trinomial of degree m exists, it is always possible to find another irreducible trinomial such that its middle coefficient n satisfies the restriction $1 \leq n \leq \lfloor \frac{m}{2} \rfloor$.

The rest of this Section is organized as follows. First, in Subsection 6.2.1, we give the corresponding formulae needed for computing the field squaring operation when considering arbitrary irreducible trinomials. Those equations are then used in Subsection 6.2.2 to find the corresponding ones for the field square root operator.

6.2.1 Field Squaring Computation

Let $A = \sum_{i=0}^{m-1} a_i x^i$ be an arbitrary element of $GF(2^m)$. Then, according to Eq. (6.16) its square, A^2, can be represented by the $2m$-coefficient vector,

$$\begin{aligned} A^2(x) &= \left[0\ a_{m-1}\ 0\ a_{m-2} \cdots\ 0\ a_1\ 0\ a_0\right] \\ &= \left[a'_{2m-1}\ a'_{m-2} \cdots\ a'_{m-1}\ a'_m\ ;\ a'_{m-1}\ a'_2 \cdots\ a'_1\ a'_0\right] \end{aligned} \tag{6.35}$$

where $a'_i = 0$ for i odd. Hence, the upper half of A^2 (i.e., the m most significant bits) in Eq. (6.35) is mapped into the first m coordinates by performing addition and shift operations only.

In order to investigate the exact cost of the field squaring operation, we categorize all the irreducible trinomials over $GF(2)$ into four different types. For all four types considered and by means of Eqs. (6.35) and (6.21), the following explicit formulae for the field squaring operation were found,

Type I: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n < \frac{m}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{m+i}{2}} & i \text{ even}, i < n \text{ or } i \geq 2n, \\ a_{\frac{i}{2}} + a_{\frac{m+i}{2}} + a_{m-n+\frac{i}{2}} & i \text{ even}, n < i < 2n, \\ a_{m+1-\frac{n+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m-n+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.36}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.36) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and $2T_x$ delays.

Type II: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n = \frac{m}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{m+i}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} & i \text{ even}, i > n, \\ a_{m+1-\frac{n+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{n+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.37}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.37) has an associated cost of $\frac{m+2}{4}$ XOR gates and one T_x delay.

Type III: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, n odd numbers and $n < \frac{m+1}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+\frac{m-n}{2}} + a_{\frac{i}{2}+(m-n)} & i \text{ even}, n < i < 2n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+\frac{m-n}{2}} & i \text{ even}, i \geq 2n, \\ a_{\frac{m+i}{2}} + a_{\frac{m+i}{2}+\frac{m-n}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m+i}{2}} & i \text{ odd}, i \geq n, \end{cases} \tag{6.38}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.38) has an associated cost of $\frac{m-1}{2}$ XOR gates and $2T_x$ delays.

Type IV: Computing $C = A^2 \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m odd, n even and $n < \frac{m+1}{2}$,

$$c_i = \begin{cases} a_{\frac{i}{2}} + a_{\frac{i}{2}+m-\frac{n}{2}} & i \text{ even}, i < n, \\ a_{\frac{i}{2}} + a_{\frac{i}{2}+m-n} & i \text{ even}, n \leq i < 2n, \\ a_{\frac{i}{2}} & i \text{ even}, i \geq 2n, \\ a_{\frac{m+i}{2}} & i \text{ odd}, i < n, \\ a_{\frac{m+i}{2}} + a_{\frac{m+i}{2}-\frac{n}{2}} & i \text{ odd}, i > n, \end{cases} \tag{6.39}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.39) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and one T_x delay.

The complexity costs found on Equations (6.36) through (6.39) are in consonance with the ones analytically derived in [386, 387].

6.2.2 Field Square Root Computation

In the following, we keep the assumption that the middle coefficient n of the generating trinomial $P(x) = x^m + x^n + 1$ satisfies the restriction $1 \leq n \leq \frac{m}{2}$.

Clearly, Eqs. (6.36)-(6.39) are a consequence of the fact that in binary extension fields, squaring is a linear operation. The linear nature of binary extension field squaring, allow us to describe this operator in terms of an $(m \times m)$-matrix as,

$$C = A^2 = MA \tag{6.40}$$

Furthermore, based on Eq. (6.40), it follows that computing the square root of an arbitrary field element A means finding a field element $D = \sqrt{A}$ such that $D^2 = MD = A$. Hence,

$$D = M^{-1}A \tag{6.41}$$

Eq. (6.41) is especially attractive for fields $GF(2^m)$ with order sufficiently large, i.e., $m >> 2$, where the matrixes M corresponding to Eqs. (6.36)-(6.39) are all highly spare (each row has at most three nonzero values).

Hence, for the trinomial types I, II, III and IV as described above, the element $D = \sqrt{A}$ given by Eq. (6.41) can be found by the computation of the inverse of the corresponding matrix M. Then using $\sqrt{A} = D = M^{-1}A$, we can determine the m coordinates of the field element as described bellow.

Type I: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd, and $n < \frac{m}{2}$:

$$d_i = \begin{cases} a_{2i} + a_{2i+n} & i \leq \lfloor \frac{n}{2} \rfloor, \\ a_{2i} + a_{(2i+n) \bmod m} + a_{2i-n} & \lfloor \frac{n}{2} \rfloor < i < n, \\ a_{2i} + a_{(2i+n) \bmod m} & n \leq i < \frac{m}{2}, \\ a_{(2i+n) \bmod m} & \frac{m}{2} \leq i < m \end{cases} \tag{6.42}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.42) has an associated cost of $\frac{m+n-1}{2}$ XOR gates and $2T_x$ delays.

Type II: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m even, n odd and $n = \frac{m}{2}$:

$$d_i = \begin{cases} a_{2i} + a_{2i+\frac{m}{2}} & i < \frac{m+2}{4}, \\ a_{2i} & \frac{m+2}{4} \leq i < \frac{m}{2} \\ a_{(2i+\frac{m}{2}) \bmod m} & \frac{m}{2} \leq i < m \end{cases} \tag{6.43}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.43) has an associated cost of $\frac{m+2}{4}$ XOR gates and one T_x delay.

Type III: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, n odd numbers and $n < \frac{m+1}{2}$,

$$d_i = \begin{cases} a_{2i} & i < \frac{n+1}{2}, \\ a_{2i} + a_{2i-n} & \frac{n+1}{2} \leq i < \frac{m+1}{2}, \\ a_{2i-n} + a_{2i-m} & \frac{m+1}{2} \leq i < \frac{m+n}{2}, \\ a_{2i-m} & \frac{m+n}{2} \leq i < m \end{cases} \tag{6.44}$$

for $i = 0, 1, \cdots, m-1$. It can be verified that Eq. (6.44) has an associated cost of $\frac{m-1}{2}$ XOR gates and one T_x delay.

Type IV: Computing D such that $D^2 = A \bmod P(x)$, with $P(x) = x^m + x^n + 1$, m, odd, n even and $\lceil \frac{m-1}{4} \rceil \leq n < \lfloor \frac{m-1}{3} \rfloor$.

$$d_i = \begin{cases} a_{2i} + a_{2i+(m-n)} + a_{2i+(m-2n)} + a_{2i+(m-3n)} & i < \frac{4n-(m-1)}{2}, \\ a_{2i} + a_{2i+(m-n)} + a_{2i+(m-2n)} + a_{2i+(m-3n)} & \\ +a_{2i+(m-4n)} & \frac{4n-(m-1)}{2} \le i < \frac{n}{2}, \\ a_{2i} + a_{2i+(m-2n)} + a_{2i+(m-3n)} + a_{2i+(m-4n)} & \frac{n}{2} \le i < \frac{5n-(m-1)}{2}, \\ a_{2i} + a_{2i+(m-2n)} + a_{2i+(m-3n)} + a_{2i+(m-4n)} & \\ +a_{2i+(m-5n)} & \frac{5n-(m-1)}{2} \le i < n, \\ a_{2i} & n \le i \le \frac{m-1}{2}, \\ a_{2i-m} & \frac{m+1}{2} \le i < \frac{n+m+1}{2}, \\ a_{2i-m} + a_{2i-(m+n)} & \frac{n+m+1}{2} \le i < \frac{2n+m+1}{2} \\ a_{2i-m} + a_{2i-(m+n)} + a_{2i-(m+2n)} & \frac{2n+m+1}{2} \le i < \frac{3n+m+1}{2} \\ a_{2i-m} + a_{2i-(m+n)} + a_{2i-(m+2n)} + a_{2i-(m+3n)} & \frac{3n+m+1}{2} \le i < m \end{cases} \tag{6.45}$$

for $i = 0, 1, \cdots, m-1$. At first glance, Eq. (6.45) can be implemented with an XOR gate cost of,

$$3 \cdot \frac{4n-(m-1)}{2} + 4 \cdot \frac{m-3n-1}{2} + 3 \cdot \frac{4n-(m-1)}{2} + 4 \cdot \frac{m-3n-1}{2} + \frac{n}{2} + 2 \cdot \frac{n}{2} + 3 \cdot \frac{m-3n-1}{2} = 5 \cdot \frac{m-n-1}{2} - \frac{n}{2}.$$

However, taking advantage of the high redundancy of the terms involved in Eq. (6.45), it can be shown (after a tedious long derivation) that actually $\frac{m+n-1}{2}$ XOR gates are sufficient to implement it with a $2T_x$ gate delays.

Table 6.5. Summary of Complexity Results

Type	Trinomial $P(x) = x^m + x^n + 1$	Operation	XOR gates	Time delay
I	m even, n odd	Squaring	$(m+n-1)/2$	$2T_x$
II	m even, $n = m/2$	Squaring	$(m+2)/4$	T_x
III	m odd, n odd	Squaring	$(m-1)/2$	$2T_x$
IV	m odd, n even	Squaring	$(m+n-1)/2$	T_x
I	m even, n odd	Square root	$(m+n-1)/2$	$2T_x$
II	m even, $n = m/2$	Square root	$(m+2)/4$	T_x
III	m odd, n odd	Square root	$(m-1)/2$	T_x
IV	m odd, n even	Square root	$(m+n-1)/2$	$2T_x$

Table 6.5 summarizes the area and time complexities just derived for the cases considered. Furthermore, in Table 6.6 we list all preferred irreducible trinomials $P(x) = x^m + x^n + 1$ of degree $m \in [160, 571]$ with m a prime number. In all the instances considered the computational complexity of computing the square root operator is comparable or better than that of the field squaring.

6.2.3 Illustrative Examples

In order to illustrate the approach just outlined, we include in this Section several examples using first the artificially small finite field $GF(2^{15})$ and then more realistic fields, in terms of practical cryptographic applications.

Example 6.1. Field Square Root Computation over $GF(2^{15})$

Let us consider $GF(2^{15})$ generated with the irreducible Type III trinomial $P(x) = x^{15} + x^7 + 1$. As it was discussed before, one can find the square root of any arbitrary field element $A \in GF(2^{15})$ by applying Eq. (6.41). In order to follow this approach, based on Eq. (6.38), we first determine the matrix M of Eq. (6.40) as shown in Table 6.7. Then, the inverse matrix of M modulus two, M^{-1}, is obtained as shown in Table 6.8. Afterwards, the polynomial coefficients, in terms of the coefficients of A, corresponding to the field square $C = A^2$ and the field square root $D = \sqrt{A}$ elements can be found from Eqs. (6.40) and (6.41) as shown in Table 6.9.

As predicted by Eq. (6.38), field squaring can be computed at a cost of $(m-1)/2 = (15-1)/2 = 7$ XOR gates and one T_x delay. In the same way, the square root operation can be computed at a cost of $\frac{(m-1)}{2} = \frac{(15-1)}{2} = 7$ XOR gates with an incurred delay time of one T_x, which matches Eq. (6.44) prediction. It is noticed that in this binary extension field, computing a field square root requires the same computational effort than the one associated to field squaring.

Example 6.2. Field Square Root Computation over $GF(2^{162})$

Let us consider $GF(2^{162})$ generated using the irreducible Type II trinomial, $P(x) = x^{162} + x^{81} + 1$. Using the same approach as for the precedent example,

Table 6.6. Irreducible Trinomials $P(x) = x^m + x^n + 1$ of Degree $m \in [160, 571]$ Encoded as $m(n)$, with m a Prime Number

$m(n)$	Type	$m(n)$	Type	$m(n)$	Type
167(35)	III	281(93)	III	439(49)	III
191(9)	III	313(79)	III	449(167)	III
193(15)	III	337(55)	III	457(61)	III
199(67)	III	353(69)	III	463(93)	III
223(33)	III	359(117)	III	479(105)	III
233(74)	IV	367(21)	III	487(127)	III
239(81)	III	383(135)	III	503(3)	III
241(70)	IV	401(152)	IV	521(158)	IV
257(41)	III	409(87)	III	569(77)	III
263(93)	III	431(120)	IV		
271(70)	IV	433(33)	III		

we can obtain the square root polynomial coefficients of an arbitrary element A from the field $GF(2^{162})$ as,

$$d_i = \begin{cases} a_{2i} + a_{2i+81} & i < 41, \\ a_{2i} & 41 \leq i < 81 \\ a_{(2i+81) \bmod 162} & 81 \leq i \end{cases} \tag{6.46}$$

for $i = 0, 1, \cdots, 161$. As predicted by Eq. (6.43) the associated cost of the field square root computation for this field is given as, $\frac{(m+2)}{4} = \frac{(162+2)}{4} = 41$ XOR gates with an incurred delay time of one T_x.

Example 6.3. Field Square Root Computation over $GF(2^{233})$

Let $GF(2^{233})$ be a field generated with the Type III irreducible trinomial[7], $P(x) = x^{233} + x^{74} + 1$. The square root of any arbitrary field element A is given as,

Table 6.7. Squaring matrix M of Eq. (6.40)

$$M = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0&1&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&0&1&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1&0&0&0&1&0&0&0 \end{bmatrix}$$

[7] This is a NIST recommended finite field for elliptic curve applications [253].

$$d_i = \begin{cases} a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} & i < 32, \\ a_{2i} + a_{2i+159} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 32 \leq i < 37, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} & 37 \leq i < 69, \\ a_{2i} + a_{2i+85} + a_{2i+11} + a_{2i-63} + a_{2i-137} & 69 \leq i < 74, \\ a_{2i} & 74 \leq i \leq 116, \\ a_{2i-233} & 116 \leq i < 154, \\ a_{2i-233} + a_{2i-307} & 154 \leq i < 191 \\ a_{2i-233} + a_{2i-307} + a_{2i-381} & 191 \leq i < 228 \\ a_{2i-233} + a_{2i-307} + a_{2i-381} + a_{2i-455} & 228 \leq i < 233 \end{cases} \quad (6.47)$$

for $i = 0, 1, \cdots, 232$. Eq. (6.47) can be implemented with an XOR gate cost of $\frac{m+n-1}{2} = 153$ XOR gates with a $4T_x$ gate delay, which agrees with the value predicted by Eq. (6.45).

6.3 Multiplicative Inverse

Among customary finite field arithmetic operations, namely, addition, subtraction, multiplication and inversion of nonzero elements, the computation of the later is the most time-consuming one. Multiplicative inversion computation of a nonzero element $a \in GF(2^m)$ is defined as the process of finding the unique element $a^{-1} \in GF(2^m)$ such that $a \cdot a^{-1} = 1$.

Several algorithms for computing the multiplicative inverse in $GF(2^m)$ have been proposed in literature [153, 93, 356, 135, 399, 127, 296, 122]. In [135], multiplicative inverse is computed using an improved modification of

Table 6.8. Square Root Matrix M^{-1} of Eq. (6.41)

$$M^{-1} = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1 \\ 0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \end{bmatrix}$$

the extended Euclidean algorithm called *almost inverse algorithm*. That iterative algorithm can compute the multiplicative inverse in approximately $2m$ clock cycles [135]. In [127] an architecture able to compute the Montgomery multiplicative inverse for both, $GF(p)$, for a prime p, and $GF(2^m)$ on a unified-field hardware platform was proposed.

Based on Fermat's Little Theorem (FLT) and using an ingenious rearrangement of the required field operations, the *Itoh-Tsujii Multiplicative Inverse Algorithm* (ITMIA) was presented in [153]. Originally, ITMIA was proposed to be applied over binary extension fields with *normal basis* field element representation. Since its publication however, several improvements and variations of it have been reported [93, 356, 399, 122, 296], showing that it can be used with other field element representations too.

Unfortunately enough, cryptographic designers have historically shown some resistance to use FLT-related techniques for computing multiplicative inverses when using polynomial basis representation. This phenomenon is probably due to three frequent misconceptions:

1. Computing multiplicative inverses by using FLT-related techniques is inefficient as those methods require many field multiplication and squaring operations;
2. ITMIA is a competitive design option only when using normal basis representation and;
3. The recursive nature of the ITMIA algorithm makes the parallelization of that algorithm rather difficult if not impossible, forcing the implementation of the ITMIA procedure in a sequential manner.

In the rest of this Section we describe efficient implementations of the binary Euclidean algorithm and the Itoh-Tsujii multiplicative inverse algorithm.

Table 6.9. Square and Square Root Coefficient Vectors

$$C = \begin{bmatrix} a_0 \\ a_8 + a_{12} \\ a_1 \\ a_9 + a_{13} \\ a_2 \\ a_{10} + a_{14} \\ a_3 \\ a_{11} \\ a_4 + a_8 + a_{12} \\ a_{12} \\ a_5 + a_9 + a_{13} \\ a_{13} \\ a_6 + a_{10} + a_{14} \\ a_{14} \\ a_7 + a_{11} \end{bmatrix}, \quad D = \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 + a_8 \\ a_3 + a_{10} \\ a_5 + a_{12} \\ a_7 + a_{14} \\ a_1 + a_9 \\ a_3 + a_{11} \\ a_5 + a_{13} \\ a_7 \\ a_9 \\ a_{11} \\ a_{13} \end{bmatrix}$$

In §6.3.1 main implementation details of the binary Euclidean algorithm are explained. Then, S6.3.2 describes how the Itoh-Tsuii algorithm can be utilized for the efficient computation of multiplicative inverses.

6.3.1 Inversion Based on the Extended Euclidean Algorithm

Given two polynomials A and B, not both 0, we say that the greatest common divisor of A and B, is the highest polynomial $D = gcd(A, B)$ that divides both A and B. Based on the property $gcd = (A, B) = gcd(B \pm CA, A)$, the revered Extended Euclidean Algorithhm (EEA)[8] is able to find the unique polynomials G and H that satisfies Bezout's celebrated formula,

$$A \cdot G + B \cdot H = D,$$

where $D = gcd(A, B)$.

Several variations of the EEA have been proposed in the open literature [96, 127, 127, 10]. EEA variants include: the almost inverse algorithm, first proposed in [323], the Binary Euclidean Algorithm (BEA), the Montgomery inverse algorithm, etc. All those algorithms show a computational complexity proportional to the maximum of A and B polynomial degrees.

Algorithm 6.8 shows the binary algorithm as it was reported in [96]. That algorithm takes as inputs the irreducible polynomial P of degree m and the field element A of degree at most $m - 1$. It gives as output the field element A^{-1} such that

$$A \cdot A^{-1} \equiv 1 \bmod P.$$

In steps 4 and 10, the operands U and V are divided by x as many times as possible, respectively. Furthermore, the variables G and H are also divided by x in steps 5-8 and 11-14, respectively. Notice that in case that either G or H are not divisible by x, then an addition with the irreducible polynomial P must be performed first. Eventually, after approximately m iterations, either U or V are equal to 1, which is the condition for exiting the main loop. Either G or H will contain the required multiplicative inverse.

The number of iterations required by Algorithm 6.8 depends on several factors such as design's architecture, target platform and even the exact structure of the irreducible polynomial $P(x)$. Roughly speaking, the number of iterations N can be estimated as $N \approx m$, where m is the size of the finite field.

[8] Euclid's algorithm is proposed in his book *Elements* published 300 B.C. Nevertheless, some scholars are convinced that it was previously known by Aristotle and Eudoxus, some 100 years earlier than Euclid's times. According to Knuth, it can be considered the oldest nontrivial algorithm that has survived to modern era [178].

Algorithm 6.8 Binary Euclidean Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, A polynomial $A \in GF(2^m)$.
Ensure: $A^{-1} \bmod P(x)$.

```
1: U = A; V = P; G = 1; H = 0;
2: while (u ≠ 1 AND v ≠ 1) do
3:    while x divides U do
4:       U = U/x;
5:       if x divides G then
6:          G = G/x;
7:       else
8:          G = (G+P)/x;
9:       end if
10:   end while
11:   while x divides V do
12:      V = V/x;
13:      if x divides G_2 then
14:         H = H/x;
15:      else
16:         H = (H+P)/x;
17:      end if
18:   end while
19:   if (deg(U)>deg(V)) then
20:      U = U + V; G = G + H;
21:   else
22:      V = V + U; H = H + G;
23:   end if
24: end while
25: if U=1 then
26:   Return(G);
27: else
28:   Return(H);
29: end if
```

6.3.2 The IToh-Tsujii Algorithm

In this Section we describe the Itoh-Tsujii Multiplicative Inversion Algorithm (ITMIA). We start deriving a recursive sequence useful for finding multiplicative inverses. Then, we briefly discuss the concept of *addition chains*, which together with the aforementioned recursive sequence yield an efficient version of the original ITMIA procedure.

Since the multiplicative group of the Galois field $GF(2^m)$ is cyclic of order $2^m - 1$, for any nonzero element $a \in GF(2^m)$ we have $a^{-1} = a^{2^m-2}$. Clearly,

$$2^m - 2 = 2(2^{m-1} - 1) = 2\sum_{j=0}^{m-2} 2^j = \sum_{j=1}^{m-1} 2^j.$$

The right-most component of above equalities allow us to express the multiplicative inverse of a in two ways:

$$\left[a^{2^{m-1}-1}\right]^2 = a^{-1} = \prod_{j=1}^{m-1} a^{2^j} \tag{6.48}$$

Let us consider the sequence $\left(\beta_k(a) = a^{2^k-1}\right)_{k\in\mathbb{N}}$. Then, for instance,

$$\beta_0(a) = 1 \quad , \quad \beta_1(a) = a,$$

and from the first equality at (6.48), $[\beta_{m-1}(a)]^2 = a^{-1}$.

It is easy to see that for any two integers $k, j \geq 0$,

$$\beta_{k+j}(a) = \beta_k(a)^{2^j}\beta_j(a). \tag{6.49}$$

Namely,

$$\begin{aligned}\beta_{k+j}(a) = a^{2^{k+j}-1} &= \frac{a^{2^{k+j}}}{a} = \frac{\left(a^{2^k}\right)^{2^j}}{a} \\ &= \left(\frac{a^{2^k}}{a}\right)^{2^j} \frac{a^{2^j}}{a} = \left(a^{2^k-1}\right)^{2^j} a^{2^j-1} \\ &= \beta_k(a)^{2^j}\beta_j(a)\end{aligned}$$

In particular, for $j = k$,

$$\beta_{2k}(a) = \beta_k(a)^{2^k}\beta_k(a) = \beta_k(a)^{2^k+1}. \tag{6.50}$$

Furthermore, we observe that this sequence is periodic of period m:

$$k_2 \equiv k_1 \bmod m \;\Rightarrow\; \beta_{k_2}(a) = \beta_{k_1}(a).$$

To see this, consider $k_2 = k_1 + nm$. Then, by eq. (6.49) and FLT,

$$\beta_{k_2}(a) = \beta_{k_2}(a)^{2^{nm}}\beta_{nm}(a) = \beta_{k_2}(a)^{2^{nm}} \cdot 1 = \beta_{k_2}(a).$$

Therefore, the sequence $(\beta_k(a))_k$ is completely determined by its values corresponding to the indexes $k = 0, \ldots, m-1$.

As a final remark, notice that for any two integers k, j, by eq. (6.49):

$$\beta_k(a) = \beta_{(k-(m-j))+(m-j)}(a) = \beta_{k+j-m}(a)^{2^{m-j}}\beta_{m-j}(a).$$

Since the sequence of β's is periodic, and the rising to the power 2^m coincides with the identity in $GF(2^m)$, we have

$$\beta_k(a) = \beta_{k+j}(a)^{2^{-j}}\beta_{m-j}(a). \tag{6.51}$$

Eq. (6.49) allows the calculation of a "current" $i(= k+j)$-th term as a recursive function of two previous terms, the k-th and the j-th in the sequence.

6.3.3 Addition Chains

Let us say that an *addition chain* for an integer $m-1$ consists of a finite sequence of integers $U = (u_0, u_1, \ldots, u_t)$, and a sequence of integer pairs $V = ((k_1, j_1), \ldots, (k_t, j_t))$ such that $u_0 = 1$, $u_t = m-1$, and whenever $1 \leq i \leq t$, $u_i = u_{k_i} + u_{j_i}$.

Example 6.4. Consider the case $e = m-1 = 193-1 = 192 = (11000000)_2$. Then, a binary addition chain with length $t = 8$ for that e is,

$$\begin{aligned} U &= (\ 1, \quad 2, \quad 4, \quad 8, \quad 16, \quad 32, \quad 64, \quad 128, \quad 192) \\ V &= (\quad (0,0), (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (6,7)) \end{aligned}$$

i.e. the associated sequence is governed by the rule, $u_i = u_{i-1} + u_{i-1} = 2u_{i-1}$ for all but the final value which is obtained using $u_t = u_{t-1} + u_{t-2}$.

Another addition chain, also with length $t = 8$, is

$$\begin{aligned} U &= (\ 1, \quad 2, \quad 3, \quad 6, \quad 12, \quad 24, \quad 48, \quad 96, \quad 192) \\ V &= (\quad (0,0), (0,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7)) \end{aligned}$$

i.e. for all $i \neq 2$ the combinatorial rule is $u_i = u_{i-1} + u_{i-1} = 2u_{i-1}$, while $u_2 = u_0 + u_1$. □

The concept of addition chains leads us to a natural way to generalize the Itoh-Tsujii Algorithm, by using an addition chain for $m-1$ and relations (6.48) and (6.49) to compute $a^{-1} = [\beta_{m-1}(a)]^2$.

6.3.4 ITMIA Algorithm

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse $a^{-1} \in GF(2^m)$ of a can be found by repeatedly applying eq's. (6.49) and/or (6.50). Hence, given $\beta_{u_0}(a) = a^{2^1-1} = a$, for each $u_i, 1 \leq i \leq t$, compute

$$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \beta_{u_{i_2}}(a) = \beta_{u_{i_2}+u_{i_1}}(a) = \beta_{u_i}(a) = a^{2^{u_i}-1}$$

A final squaring step yields the required result since,

$$[\beta_{u_t}(a)]^2 = \left(a^{2^{m-1}-1}\right)^2 = (a^{2^m-2}) = a^{-1}.$$

Fig. 6.9 shows an algorithm that iteratively computes all the $\beta_{u_i}(a)$ coefficients in the exact order stipulated by the addition chain U as discussed above.

We assess the computational complexity of the algorithm shown in Fig. 6.9 as follows. The algorithm performs t iterations (where t is the length of the

addition chain U) and one field multiplication per iteration. Thus, we conclude that a total of t field multiplication computations are required. On the other hand, notice that at each iteration i, a total of $2^{u_{i_2}}$ field squarings are performed. Notice also that by definition, the addition chain guarantees that for each $u_i, 1 \leq i \leq t$, the relation $u_{i_2} = u_i - u_{i_1}$ holds. Hence, one can show by induction that the total number of field squaring operations performed right after the execution of the i-th iteration is $u_i - 1$. Therefore, at the end of the final iteration t, a total of $u_t - 1 = m - 2$ squaring operations have been performed. This, together with the final squaring operation, yield a total of $m - 1$ field squaring computations.

Summarizing, the algorithm of Fig. 6.9 can find the multiplicative inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#Multiplications &= t; \\ \#Squarings &= m - 1. \end{aligned} \tag{6.52}$$

Algorithm 6.9 Itoh-Tsujii Multiplicative Inversion Addition-Chain Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, An element $a \in GF(2^m)$, an addition chain U of length t for $m-1$ and its associated sequence V.
Ensure: $a^{-1} \in GF(2^m)$.
1: $\beta_{u_0}(a) = a$;
2: **for** i from 1 to t **do**
3: $\beta_{u_i}(a) = \left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a) \bmod P(X)$;
4: **end for**
5: **Return**$(\beta^2_{u_t}(a) \bmod P(X))$.

Example 6.5. Let us consider the binary field $GF(2^{193})$ using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, using the addition chain of Example 6.4, the algorithm of Fig. 6.9 would compute the sequence of $\beta_{u_i}(a)$ coefficients as shown in Table 6.3.4. Once again, notice that after having computed the coefficient $\beta_{u_8}(a)$, the only remaining step is to obtain a^{-1} which can be achieved as $a^{-1} = \beta^2_{u_8}(a)$. □

6.3.5 Square Root ITMIA

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse of a, $a^{-1} \in GF(2^m)$, can be found as follows [295].

Given $\gamma_{u_0}(a) = a^{1-2^{-1}} = \sqrt{a}$, for each $u_i, 1 \leq i \leq t$, compute

Table 6.10. $\beta_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_i}(a) = a^{2^{u_i}-1}$
0	1	–	–	$\beta_{u_0}(a) = a^{2^1-1}$
1	2	$2u_{i-1}$	$[\beta_{u_0}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_1}(a) = a^{2^2-1}$
2	3	$u_{i-1}+u_{i-2}$	$[\beta_{u_1}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_2}(a) = a^{2^3-1}$
3	6	$2u_{i-1}$	$[\beta_{u_2}(a)]^{2^{u_2}} \cdot \beta_{u_2}(a)$	$\beta_{u_3}(a) = a^{2^6-1}$
4	12	$2u_{i-1}$	$[\beta_{u_3}(a)]^{2^{u_3}} \cdot \beta_{u_3}(a)$	$\beta_{u_4}(a) = a^{2^{12}-1}$
5	24	$2u_{i-1}$	$[\beta_{u_4}(a)]^{2^{u_4}} \cdot \beta_{u_4}(a)$	$\beta_{u_5}(a) = a^{2^{24}-1}$
6	48	$2u_{i-1}$	$[\beta_{u_5}(a)]^{2^{u_5}} \cdot \beta_{u_5}(a)$	$\beta_{u_6}(a) = a^{2^{48}-1}$
7	96	$2u_{i-1}$	$[\beta_{u_6}(a)]^{2^{u_6}} \cdot \beta_{u_6}(a)$	$\beta_{u_7}(a) = a^{2^{96}-1}$
8	192	$2u_{i-1}$	$[\beta_{u_7}(a)]^{2^{u_7}} \cdot \beta_{u_7}(a)$	$\beta_{u_8}(a) = a^{2^{192}-1}$

$$\left[\gamma_{u_{i_1}(a)}\right]^{2^{-u_{i_2}}} \gamma_{u_{i_2}}(a) = \gamma_{u_{i_2}+u_{i_1}}(a) = \gamma_{u_i}(a) = a^{1-2^{-u_i}}$$

Where $\gamma_{\{u_t=m-1\}} = a^{1-2^{-(m-1)}} = a^{-1}$ gives the required result.

Fig. 6.10 shows an algorithm that iteratively computes all the $\gamma_{u_i}(a)$ coefficients in the exact order stipulated by the addition chain U as discussed above. We assess the computational complexity of the algorithm shown in Fig. 6.10 as follows. The algorithm performs one field multiplication in each of algorithm's t iterations, yielding a total of t field multiplication computations required. Furthermore, at each iteration i, a total of $2^{u_{i_2}}$ field square roots are performed. Since by definition, the addition chain guarantees that for each $u_i, 1 \leq i \leq t$, the relation $u_{i_2} = u_i - u_{i_1}$ holds, one can show that the total number of field square root operations performed right after the execution of the i-th iteration is $u_i - 1$. Therefore, a total of $u_t - 1 = m - 2$ square root operations must be performed. This, together with the initial square root operation, yield a total of $m - 1$ field square root computations.

Summarizing, the algorithm of Fig. 6.10 can find the inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#\textit{Multiplications} &= t; \\ \#\textit{Square root} &= m-1. \end{aligned} \tag{6.53}$$

Example 6.6. Following with our running example, let us consider the binary field $GF(2^{193})$ generated using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, the algorithm of Fig. 6.10 would compute the sequence of $\gamma_{u_i}(a)$ coefficients as shown in Table 6.3.5. The multiplicative inverse is given as $\gamma_{u_8} = a^{-1}$. □

Algorithm 6.10 Square Root Itoh-Tsujii Multiplicative Inversion Algorithm

Require: An irreducible polynomial $P(X)$ of degree m, An element $a \in GF(2^m)$, an addition chain U of length t for $m-1$ and its associated sequence V.
Ensure: $a^{-1} \in GF(2^m)$. **Procedure** SquareRoot_ITMIA($P(X), a, \{U, V\}$) {
1: $\gamma_{u_0}(a) = a^{1-2^{-1}} = \sqrt{a}$;
2: **for** i from 1 to t **do**
3: $\gamma_{u_i}(a) = \left[\gamma_{u_{i_1}}(a)\right]^{2^{-u_{i_2}}} \cdot \gamma_{u_{i_2}}(a) \bmod P(X)$;
4: **end for**
5: **Return**($\gamma_{u_t}(a) \bmod P(X)$)

Table 6.11. $\gamma_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\gamma_{u_{i_1}}(a)\right]^{2^{-u_{i_2}}} \cdot \gamma_{u_{i_2}}(a)$	$\gamma_{u_i}(a) = a^{1-2^{-u_i}}$
0	1	–	–	$\gamma_{u_0}(a) = a^{1-2^{-1}}$
1	2	$2u_{i-1}$	$[\gamma_{u_0}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_1}(a) = a^{1-2^{-2}}$
2	3	$u_{i-1} + u_{i-2}$	$[\gamma_{u_1}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_2}(a) = a^{1-2^{-3}}$
3	6	$2u_{i-1}$	$[\gamma_{u_2}(a)]^{2^{-u_2}} \cdot \gamma_{u_2}(a)$	$\gamma_{u_3}(a) = a^{1-2^{-6}}$
4	12	$2u_{i-1}$	$[\gamma_{u_3}(a)]^{2^{-u_3}} \cdot \gamma_{u_3}(a)$	$\gamma_{u_4}(a) = a^{1-2^{-12}}$
5	24	$2u_{i-1}$	$[\gamma_{u_4}(a)]^{2^{-u_4}} \cdot \gamma_{u_4}(a)$	$\gamma_{u_5}(a) = a^{1-2^{-24}}$
6	48	$2u_{i-1}$	$[\gamma_{u_5}(a)]^{2^{-u_5}} \cdot \gamma_{u_5}(a)$	$\gamma_{u_6}(a) = a^{1-2^{-48}}$
7	96	$2u_{i-1}$	$[\gamma_{u_6}(a)]^{2^{-u_6}} \cdot \gamma_{u_6}(a)$	$\gamma_{u_7}(a) = a^{1-2^{-96}}$
8	192	$2u_{i-1}$	$[\gamma_{u_7}(a)]^{2^{-u_7}} \cdot \gamma_{u_7}(a)$	$\gamma_{u_8}(a) = a^{1-2^{-192}}$

6.3.6 Extended Euclidean Algorithm versus Itoh-Tsujii Algorithm

In order to assess the performance differences between multiplicative inverse computation via the Extended Euclidean Algorithm and the Itoh-Tsujii Algorithm, we performed the following experiment.

Using a Virtex 2 xc2v4000-6bf957 as a target device, we implemented Algorithms 6.8 and 6.9 for computing multiplicative inverses in the field $GF(2^m)$ generated using the irreducible trinomial $P(x) = x^{193} + x^{15} + 1$. Algorithm 6.8 was implemented according to the finite-state machine shown in Fig. 6.10, whereas the Itoh-Tsujii Algorithm was implemented using the architecture shown in Fig. 6.11. The implementation statistics obtained for each algorithm are summarized in Table 6.12.

According to Table 6.12, it can be observed that the BEA scheme represents a cheaper solution in terms of hardware resource requirements. Indeed, the BEA scheme utilizes just 12.02% of the area required by the ITMIA design. On the contrary, the ITMIA scheme outperforms the BEA scheme in timing performance, with a speedup of about 3.3 times. Therefore, consider-

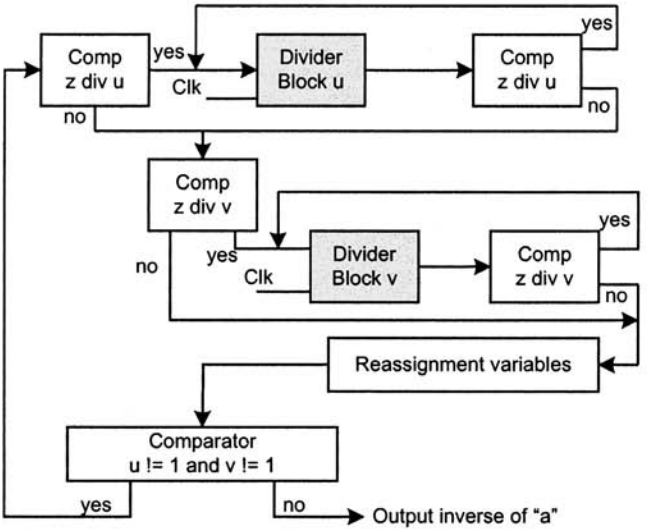

Fig. 6.10. Finite State Machine for the Binary Euclidean Algorithm

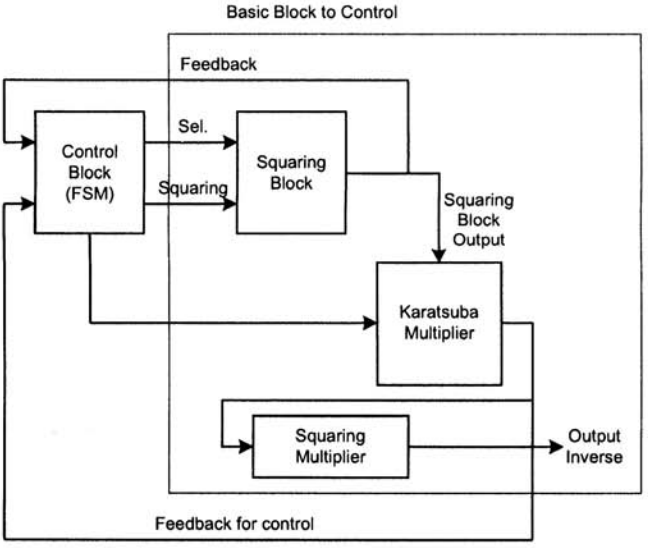

Fig. 6.11. Architecture of the Itoh-Tsujii Algorithm

Table 6.12. BEA Versus ITMIA: A Performance Comparison

Design	Cost	Cycles	Freq (MHz)	$timings$	$\frac{1}{Slices \times timings}$
BEA	1195	191	76.10	2509ηS	333.53
ITMIA	9945	40	55.25	724ηS	138.89
ITMIA without KOM Block	2345	40	55.25	724ηS	589.00

ing our customary efficiency figure of merit of $\frac{1}{Slices \times timings}$, we can see that the BEA solution is about 2.40 times more efficient than the ITMIA design.

Nevertheless, since for all practical cryptographic and code applications a binary extension field multiplier is a mandatory operator, we included the performance statistics of both, the ITMIA design considering the costs of the expensive Karatsuba-Ofman Multiplier (KOM) block and without considering it. In the case that the KOM block cost is taken out of the ITMIA statistics, Table 6.12 shows that the ITMIA solution becomes the most efficient option, providing An efficiency improvement of nearly 1.77 times with respect to the BEA design.

6.3.7 Multiplicative Inverse FPGA Designs

Table 6.13 shows the computational cost of several reported designs for the computation of multiplicative inversion over $GF(2^m)$ in hardware platforms. The *standard* Itoh-Tsujii algorithm using the architecture described here requires 28 clock cycles in the design reported in [295], thus computing the multiplicative inverse in about 1.32μS.

6.4 Other Arithmetic Operations

In this Section we briefly describe some important binary finite field arithmetic operations such as, the computation of the trace function, the half trace function and binary exponentiation. The first two operations are key building blocks for *halving* an elliptic curve point, which will be studied in §10.7.

6.4.1 Trace function

Given $C \in GF(2^m)$, the trace function can be defined as:

$$Tr(C) = C + C^2 + C^{2^2} + \ldots + C^{2^{m-1}} \tag{6.54}$$

Due to its linearity, the trace function can be implemented such that the execution time is $O(1)$ as [133],

Table 6.13. Design Comparison for Multiplicative Inversion in $GF(2^m)$

Work	Platform	Field	Cost	Cycles	Freq (MHz)	*timings*
BEA divisor [403]	0.18μm CMOS	$GF(2^{163})$	1.658 mm^2	198	400	0.495μS
BEA divisor [77]	0.18μm CMOS	$GF(2^{163})$	1.192 mm^2	326	460	0.709μS
ITMIA [248]	Xilinx Virtex 2	$GF(2^{193})$	9945	40	55.25	0.724μS
Parallel ITMIA [295]	Xilinx Virtex 2	$GF(2^{193})$	12021 CLBs	20	21.2	0.943μS
ITMIA [295]	Xilinx Virtex 2	$GF(2^{193})$	11081 CLBs	28	21.2	1.32μS
BEA [248]	Xilinx Virtex 2	$GF(2^{193})$	1195 CLBs	191	76.1	2.509μS
Montgomery Inversion [314]	0.18μm CMOS	160-bit	14.4K NANDs	1516	227.3	2.509μS
ITMIA [20]	Xilinx Virtex	$GF(2^{191})$	–	390	50	7.8μS (est.)
ITMIA [216]	Xilinx Virtex	$GF(2^{163})$		711	66	10.7μS
BEA [114]	0.25μm CMOS	$GF(2^{256})$	–	844	50	16.88μS (est.)

$$Tr(C) = Tr(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i Tr(x^i) \tag{6.55}$$

As an example, consider the field defined by $GF(2^{163})$ with the reduction polynomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$. Then, $Tr(x^i) = 1$ if and only if $i \in \{0, 157\}$. The implementation of the trace function in reconfigurable hardware only needs one XOR gate to add the bits *0 and 157* from the input polynomial.

6.4.2 Solving a Quadratic Equation over $GF(2^m)$

In order to solve a quadratic Equation (10.26), we may use the half-trace function. Let $C \in GF(2^m)$ be defined as $C(x) = \sum_{i=0}^{m-1} c_i x^i \in GF(2^m)$ with $Tr(C) = 0$ and m an odd integer, the half-trace function can be defined as:

$$H(C) = H(\sum_{i=0}^{m-1} c_i x^i) = \sum_{i=0}^{m-1} c_i H(x^i) \tag{6.56}$$

Therefore, by using the definition of the half trace equation 6.56. We can precompute the m half-traces of the field elements x^i for $i = 0, 1, \ldots, m-1$; and by arranging these Equations in a $m \times m$ matrix B, we may obtain the half-trace of an arbitrary element $C \in GF(2^m)$ by computing $H(C) = CB$.

6.4.3 Exponentiation over Binary Finite Fields

Exponentiation over binary finite fields is used for inverse computation via Fermat Little theorem [295] and key agreement schemes such as the Diffie-Hellman protocol, among other applications.

For binary extension fields $GF(2^m)$, generated using the m-degree irreducible polynomial $P(x)$, irreducible over $GF(2)$. Let e be an arbitrary m-bit positive integer e, with a binary expansion representation given as,

$$e = (1e_{m-2} \dots e_1 e_0)_2 = 2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i.$$

Then,

$$\begin{aligned} b = a^e &= a^{2^{m-1} + \sum_{i=0}^{m-2} 2^i e_i} \qquad (6.57) \\ &= a^{2^{m-1}} \cdot a^{2^{m-2} e_{m-2}} \cdot \dots \cdot a^{2^1 e_1} \cdot a^{2^0 e_0} = a^{2^{m-1}} \cdot \prod_{i=0}^{m-2} a^{2^i e_i} \end{aligned}$$

Algorithm 6.11 MSB-first Binary Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \dots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$
1: $b = a$;
2: **for** $i = m - 2$ **downto** 0 **do**
3: $b = b^2$;
4: **if** $e_i == 1$ **then**
5: $b = b \cdot a \bmod P(x)$;
6: **end if**
7: **end for**
8: **Return** b

Binary strategies evaluate (6.57) by scanning the bits of the exponent e one by one, either from left to right (MSB-first binary algorithm) or from right to left (LSB-first binary algorithm) applying the so-called Horner's rule. Both strategies require a total of $m-1$ iterations. At each iteration a squaring operation is performed, and if the value of the scanned bit is one, a subsequent field multiplication is performed. Therefore, the binary strategy requires a total of $m-1$ squarings and $H(e)-1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the MSB-first binary algorithm is shown in Algorithm 6.11.

On the other hand, it is known from Fermat Little Theorem that for any nonzero $a \in GF(2^m)$, we have $a^{2^m-1} = 1$ which implies $a^{2^m} = a$ and by taking square root in both sides of the last relation we get $a^{2^{m-1}} = \sqrt{a} = a^{2^{-1}}$. In general, the i-th square-root of a, with $i \geq 1$ can be written as,

$$a^{2^{m-i}} = a^{2^{-i}}.$$

Hence, Eq. (6.57) can be reformulated in terms of the square root operator as,

$$a^e = a^{2^{m-1}} \cdot \prod_{i=0}^{m-2} a^{2^i e_i} = a^{2^{m-1}} \cdot a^{2^{m-2} e_{m-2}} \cdot \ldots \cdot a^{2^1 e_1} \cdot a^{2^0 e_0} \tag{6.58}$$

$$= a^{2^{-1}} \cdot a^{2^{-2} e_{m-2}} \cdot \ldots \cdot a^{2^{-(m-1)} e_1} \cdot a^{2^0 e_0} = \sqrt{a} \cdot \prod_{i=2}^{m-1} a^{2^{m-i} e_i} \cdot a^{e_0}$$

Algorithm 6.12 Square root LSB-first Binary Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \ldots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$
1: $b = a$;
2: $e_m = e_0$;
3: **for** $i = 1$ **to** m **do**
4: $b = \sqrt{b}$;
5: **if** $e_i == 1$ **then**
6: $b = b \cdot a \bmod P(x)$;
7: **end if**
8: **end for**
9: **Return** b

Therefore, the novel square root LSB-first binary strategy requires a total of $m - 1$ square root computations and $H(e) - 1$ field multiplications, where $H(e)$ is the Hamming weight of the binary representation of e. The pseudo-code of the square root LSB-first binary algorithm is shown in Algorithm 6.12. Algorithms 6.11 and 6.12 suggest a parallel version that can combine both ideas. This parallel version is especially attractive for hardware platforms implementations. Algorithm 6.13 shows this suggesting algorithm. Notice that both loop computations can be performed in parallel provided that the architecture has two independent field multiplier units. The computational time speedup can be estimated in about 50% when compared with Algorithms 6.11 and 6.12.

6.5 Conclusions

In this chapter, we addressed the problem of how to implement efficiently finite field arithmetic algorithms for reconfigurable hardware platforms. We included detailed analysis of complexities for binary field operations such as: multiplication, squaring, square root, multiplicative inverse computation, among others.

Algorithm 6.13 Squaring and Square Root Parallel Exponentiation

Require: The irreducible polynomial $P(x)$, $a \in GF(2^m)$, $e = (e_{m-1} \dots e_1 e_0)_2$
Ensure: $b = a^e \bmod P(x)$

```
1: b = c = 1 ;
2: e_m = 0 ;
3: N = ⌊m/2⌋ ;
4: for i = N downto 0 do          for j = N + 1 to m do
5:    b = b^2 ;                      c = √c;
6:    if e_i == 1 then               if e_j == 1 then
7:       b = b · a;                     c = c · a;
8:    end if
9: end for
10: b = b · c;
11: Return b
```

In §6.1, field multipliers algorithms were studied covering the whole spectrum of state-of-the-art strategies for computing that crucial arithmetic operation as efficiently as possible. That spectrum goes from the mighty fully bit-parallel Karatsuba-Ofman multiplier to the ultra compact interleaving multiplier which can be quite useful for constrained environments.

The most attractive feature of the Karatsuba-Ofman algorithm variation analyzed in §6.1.2, is that the degree m of the generating irreducible polynomial can be arbitrarily selected by the designer, allowing the usage of prime degrees. In addition, the new field multiplier leads to architectures which show a considerably improved space complexity when compared to traditional approaches. Moreover, the binary Karatsuba-Ofman multiplier leads to highly modular architectures that are well suited for both, VLSI and reconfigurable hardware implementations.

We studied in §6.1.4 a method able to perform the reduction step of field multipliers when an irreducible trinomial or pentanomial is used to generate the field. Moreover, we also presented a general method for accomplishing reduction when dealing with arbitrary irreducible polynomials.

In §6.2 a low-complexity bit-parallel algorithm for computing square roots over binary extension fields was studied. Although the method presented can be applied for any type of irreducible polynomials, we were particularly interested in studying the case of irreducible trinomials. Hence, in order to investigate the exact cost of the square root operator, we categorized irreducible trinomials over $GF(2)$ into four different types. For all four types considered, explicit area and time complexity formulae were found for both, field squaring and field square root operators. It was shown that for the important practical case of finite fields generated using irreducible trinomials, the square root operation can be performed with no more computational cost than the one associated to the field squaring operation.

In §6.3 we presented a performance comparison of two of the most popular algorithms for computing the field multiplicative inverse operation: the

Binary Euclidean Algorithm (BEA) and the Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA). It was shown that the Itoh-Tsujii strategy offers a competitive performance when implemented in hardware platforms. Furthermore, we combined the standard Itoh-Tsuii algorithm with the concept of addition chains. Then, we showed that for this version of the Itoh-Tsuii algorithm the multiplicative inverse of an arbitrary nonzero field element in GF(2^m) can be computed by performing exactly $m-1$ field squarings and t multiplications, where t is the step-length of the optimal addition-chain for m-1. One of the main conclusions of this Section is that according to Table 6.12 there is not a clear winner when comparing the BEA and the ITMIA methods.

Finally, in §6.4 some less popular field arithmetic operations were studied, such as, the computation of the trace function, the half trace function and binary field exponentiation. The first two operations are key building blocks for *halving* an elliptic curve point, which will be studied in §10.7.

7

Reconfigurable Hardware Implementation of Hash Functions

This Chapter has two main purposes. The first purpose is to introduce readers to how hash functions work. The second purpose is to study key aspects of hardware implementations of hash functions. To achieve those goals, we selected MD5 as the most studied and widely used hash algorithm. A step-by-step description of MD5 has been provided which we hope will be useful for understanding the mathematical and logical operations involved in it. The study and analysis of MD5 will be utilized as a base for explaining the most recent SHA2 family of hash algorithms.

We start this Chapter given a brief introduction to hash algorithms in Section 7.1. A survey of some famous hash algorithms is presented in Section 7.2. Then we provide a detailed discussion of the MD5 algorithm in Sec. 7.3. All MD5 steps are explained by means of an illustrative example which is explained at a bit level. In Section 7.4, we describe the SHA2 family of hash algorithms and some tips are provided with respect to their hardware implementation. In Section 7.5 design strategies to achieve efficient hash algorithms when implemented on reconfigurable devices are discussed. Section 7.6 presents a review of recent hash function hardware implementations. Finally, in Section 7.7 concluding remarks are drawn.

7.1 Introduction

As it was explained in Chapter 2, a Hash function H is a computationally efficient function that maps fixed binary chains of arbitrary length $\{0,1\}^*$ to bit sequences $H(B)$ of fixed length. $H(M)$ is the hash value, hash code or digest of M [110].

In words, let M be a message of an arbitrary length. A *hash function* operates on M and returns a fixed-length value, h, as shown in Fig. 7.1. The value h is commonly called *hash code*. It is also referred to as a message

digest or hash value. The main application of hash functions lies on producing fingerprint of a file, message or other blocks of data.

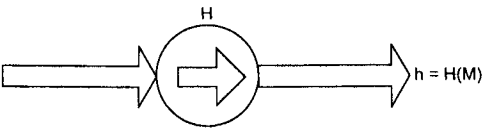

Fig. 7.1. Hash Function

Hash functions do not use a particular key, but instead, it is a highly non linear function of all message bits. The code changes with the change of any bit or bits in the input message and thus it provides error detection capabilities.

In practice, modern hash functions are specifically designed for having a short bit-length hash code h (usually from around 128 bits up to 512 bits). This characteristic is especially attractive for the application of hash functions in virtually every digital signature algorithm. Therefore, rather than attempting to sign the whole message (which by definition has arbitrary length), it becomes more practical to sign the hash code of the message as it was depicted in the basic digital signature/verification scheme shown in Figure 2.6.

As a way of illustration, let us suppose that Ana received \$500 from Bill, and that afterwards, she proceeded signing the hash code $h1$ of the message $M1$ as shown below,

M1 = Ana received \$500 from Bill

h1 = H(M1) = 89CB0C238A3C7A78D0DD7063C4153B65

Bill can never claim that Ana received \$5000 as the hash code $h2$ of message $M2$ using the same hash function vastly differs,

M2 = Ana received \$5000 from Bob.

h2=H(M2)=CCD40B907C543D96FDB7203979E55E8B

Alternatively, Bill may try to find another message M3 whose hash value corresponds to the hash value of message M1, and then claim that Ana actually signed message M3, not M1.

If we can find any two messages producing the same message digest, we say that we have found a *collision*. *Collision* is a not desired characteristic of hash functions but at the same time is unavoidable. All that one can hope is that no matter how determined an adversary may be, it should result computational unfeasible for him/her to find collisions. Therefore, a hash function H is said to be strong enough against collision and thus useful for message authentication, if it has the following properties [342, 246],

- H applies to any block of data.
- H returns a fixed-length output.
- For any given value x, $H(x)$ is relatively easy to compute. That feature makes hash function implementations more practical in both software and hardware platforms (Fig. 7.2a).

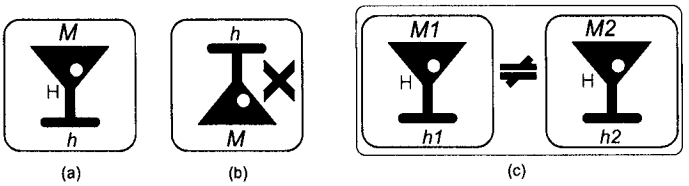

Fig. 7.2. Requirements of a Hash Function

- Given x, it is easy to compute $H(x)$. Given h, it is computationally infeasible to find x such that $H(x) = h$. That is sometimes referred to as *one way* property of hash functions (Fig. 7.2b).
- For any given block x, it is computationally infeasible to find y ($y \neq x$), with $H(y) = H(x)$. This is sometimes referred to as *weak collision resistance.*
- To find a pair (x, y) such that $H(x) = H(y)$, is computationally infeasible. This is sometimes referred to as *strong collision resistance* (Fig. 7.2c).

7.2 Some Famous Hash Functions

The overall structure of a typical hash function is shown in Fig. 7.3.

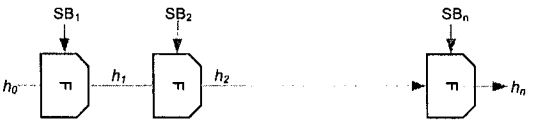

Fig. 7.3. Basic Structure of a Hash Function

The structure was first proposed by Merkle [233, 234] and then followed by most hash function designs in use today including MD5, SHA-1 and RIPEMD-160 [342].

It is apparent from Fig. 7.3 that a typical hash function is iterative in nature. That is, it partitions (*hashes*) a given input message to L sub blocks SBs of some fixed length m bits and operates sequentially on each SB. Those message blocks shorter in length than m are padded as necessary with zeroes.

Table 7.1. Some Known Hash Functions

Name	Author(s)	Year	Block Size	Digest Size
AR	ISO [151]	1992		
Boognish	Daemen[58]	1992	32	up to 160
Cellhash	Daemen, Govaerts, Vandewalle [59]	1991	32	up to 256
FFT-Hash I	Schnorr [318]	1991	128	128
GOST R 34.11-94	Government Committee of Russia for Standards [257]	1990	256	256
FFT-Hash II	Schnorr [319]	1992	128	128
HAVAL	Zheng, Pieprzyk, Seberry [402]	1994	1024	128, 160, 192, 224, 256
MAA	ISO [150]	1988	32	32
MD2	Rivest [162]	1989	512	128
MD4	Rivest [288]	1990	512	128
MD5	Rivest [289]	1992	512	128
N-Hash	Miyaguchi, Ohta, Iwata [237]	1990	128	128
PANAMA	Daemen, Clapp [56]	1998	256	unlimited
Parallel FFT-Hash	Schnorr, Vaudenay [320]	1993	128	128
RIPEMD	The RIPE Consortium [287]	1990	512	128
RIPEMD-128	Dobbertin, Bosselaers, Preneel [70]	1996	512	128
RIPEMD-160	Dobbertin, Bosselaers, Preneel [70]	1996	512	160
SHA-0	NIST/NSA [61]	1991	512	160
SHA-1	NIST/NSA [255]	1993	512	160
SHA-224	NIST/NSA [255]	2004	512	224
SHA-256	NIST/NSA [255]	2000	512	256
SHA-384	NIST/NSA [255]	2000	1024	384
SHA-512	NIST/NSA [255]	2000	1024	512
SMASH	Knudsen [177]	2005	256	256
Snefru	Merkle [235]	1990	512-m	m = 128, 256
StepRightUp	Daemen [55]	1995	256	256
Subhash	Daemen [57]	1992	32	up to 256
Tiger	Anderson, Biham [8]	1996	512	192
Whirlpool	Barreto, Rijmen [286]	2000	512	512

The heart of a hash algorithm is the so-called *compression function* F. A repeated use of function F is made by the hash algorithm. F takes two inputs: an m-bit input block message and; an n-bit input from previous step, called hash h of that message block. The output is an n-bit hash h, namely [317],

$$h_j = F(Sb_j, h_{j-1}) \tag{7.1}$$

For j=1, 2, ..., L, where L is the total number of SB message blocks. For $j = 1$, the function F takes the first sub block SB_1 and h_0, where h_0 is a fixed value provided by the algorithm. For h_n, (i.e. $j = n$), the two inputs are SB_n and h_{n-1}, h_n is the hash value of the entire message.

The term compression comes from the fact that the hash output has a much shorter bit-length n than the original input message bit-length m. Although it has not been formally proved, some authors consider that the security of a hash function strongly depends upon the security of its compression function [234, 62, 245]. Indeed, if the compression function is strongly collision resistant, then hashing a message using that method is also secure. Modern hash functions strive for improving the internal logic of their compression functions. At the same time, extensive research has been carried out on the issue of how many repetitions of the compression function are essential for obtaining an acceptable security and how those repetitions could be sequenced.

Table 7.1 features a list of known hash functions prepared by [17]. Detailed discussions about the design of most of those hash functions can be found in [165, 275, 234, 19, 276, 277, 276, 278, 347, 348, 360, 28, 119, 119, 138].

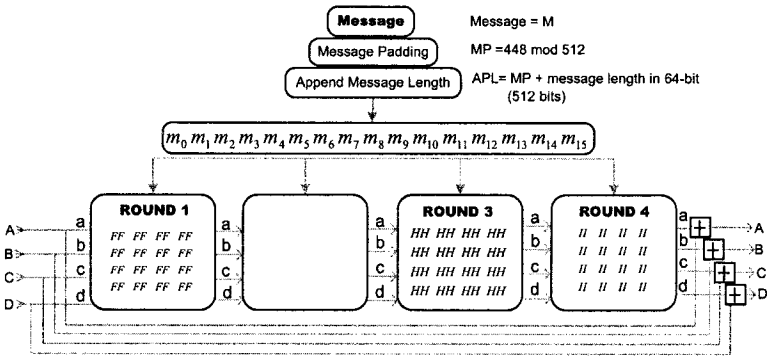

Fig. 7.4. MD5

7.3 MD5

The series of Message Digest (MD) hash algorithms is due to Rivest[289]. The original message digest algorithm was simply called MD. MD was quickly followed by MD2 [162]. Nevertheless, MD2 was soon found to be quite weak. Rivest then started working on MD3, which however was never released. MD4 [288] was the next family member. Soon MD4 was also found to be imperfect, but it provided the theoretical foundations for its successors MD5 (designed in 1992) and also for SHA-0 [61] and RIPEMD [287], from other

authors. Then, in 2004, the never ending battle between hash function designers and crypto analysts had yet another episode, when several advances for finding collisions on MD5 were announced in [24, 159].

Short after that, Wang et al. without revealing their method, presented on the rump session of [98] evidence of MD5 colliding messages [370]. Wang et al. method was later published in [372]. Before that happened though, several experimental results were presented in [174], showing for the first time how MD5 could be break. Recently, it has been proved that collisions on MD5 can be found (under certain conditions) within a minute using a standard laptop [175].

Operating on 512-bit input blocks, MD5 produces 128-bit message digests from input messages of arbitrary length. For longer messages, a partition into sub blocks is performed. The algorithm then operates iteratively on all message sub-blocks as shown in Fig. 7.4. In the following Subsection, MD5 steps for hashing a message are described in detail.

7.3.1 Message Preprocessing

First, original message is preprocessed. The message is padded such that its length (in bits) is congruent to 448 mod 512. Messages shorter than 448 bits are padded with the first bit set to '1' and all the rest set to zero. The remaining 64 bits for completing a block of 512 bits are reserved for appending message length. For instance, a message with 200-bit length would require a padding of 228 bits. The padding would comprise a single '1' at the most significant position followed by 227 zeroes. The last 64 bits are all zeroes except for the last byte which is "11001000" denoting message length of 200. As a way of illustration, we show below how a sub block of 512-bit is obtained from an input message. Let our input message M be,

"MD5 was proposed by Ron Rivest in 1992."

The ASCII representation of the message M (39 characters) is shown in Table 7.2.

Table 7.2. Bit Representation of the Message M

01001101 01000100 00110101 00100000 01110111 01100001 01110011 00100000
01110000 01110010 01101111 01110000 01101111 01110011 01100101 01100100
00100000 01100010 01111001 00100000 01010010 01101001 01110110 01100101
01110011 01110100 00100000 01101001 01101110 00100000 00110001 00111001
00111001 00110010 00101110

The first step consists on padding the Message M in order to complete a block of 512 bits as shown in Table 7.3. Notice the location of the padding

start bit (i.e. bit '1') and the message length (given in a 64-bit representation) appended into the last 64 bits (shaded). As it was explained above, the padding process assures that the block message length will always be an exact multiple of 512. Thereafter the main loop starts. A message parsing is required for this loop. This is accomplished by dividing the 512-bit input message block into sixteen 32 bit words.

Table 7.3. Padded Message (M)

01001101	01000100	00110101	00100000	01110111	01100001	01110011	00100000
01110000	01110010	01101111	01110000	01101111	01110011	01100101	01100100
00100000	01100010	01111001	00100000	01010010	01101001	01110110	01100101
01110011	01110100	00100000	01101001	01101110	00100000	00110001	00111001
00111001	00110010	00101110	10000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000	00000000	00000001	00011000

In the case of hardware implementations, designers can use various options for message preprocessing. One of the possible approaches is to use sixteen 32 bit shift registers which are initialized with zeroes except for the first one which has its first bit set to '1'. All the 16 registers are cascaded in such a way that the output of one is placed as the input of the next register.

Thus, whenever a message is read, all message bits are sequentially transferred to shift registers. The start bit '1' of the first shift register is now the end bit of the message as shown in Fig. 7.5. Since there is no need to cascade final register (SR15) with the other registers it can be reserved for appending the message length. That register arrangement also completes message parsing as all 16 registers contain 32-bit words.

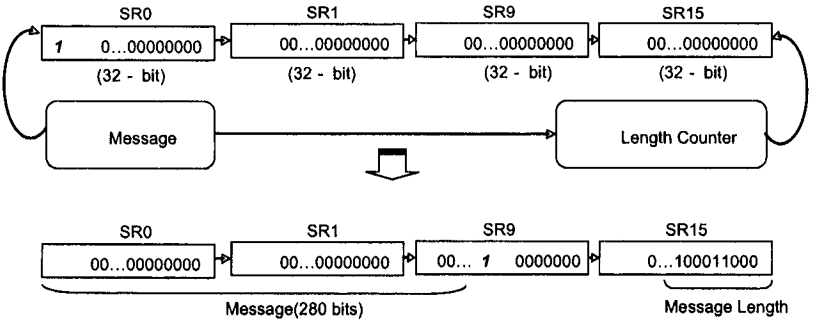

Fig. 7.5. Message Block = 32 × 16 =512 Bits

Rivest selected a *little-endian architecture* for interpreting a message as a sequence of 32-bit words. A little endian architecture stores the least significant byte of a word into the lowest byte address. This design decision was taken due to Rivest observation that several processor architectures with little endian format offer faster processing [342]. This way, the first block message is converted into sixteen 32-bit words, which are then written into hex little endian format as shown in Table 7.4.

Table 7.4. Message in Little Endian Format

Message in Hex	Message little endian format
0x4d443520	0x2035444d
0x77617320	0x20736177
0x70726f70	0x706f7270
0x6f736564	0x6465736f
0x20967920	0x20796220
0x526f6e20	0x206e6f52
0x52697665	0x65766952
0x69207473	0x69207473
0x6e203139	0x3931206e
0x39322e80	0x802e3239
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000	0x00000000
0x00000000,0x00000138	0x00000138,0x00000000

Appending bits to message blocks according to the Little endian format is intended for 32-bit word rather than one byte words. Therefore, the 64 bits that are reserved for keeping the message length are divided into two 32-bit words. By applying said convention, the lower order 32-bit word is appended first as shown in Table 7.4 (observe the last two 32-bit words).

7.3.2 MD Buffer Initialization

As it has been already mentioned, internally MD5 operates on two inputs: the input message block and the output hash from the previous step. In the first step, the initial hash values are constants provided by the algorithm. The initial values for MD5 are provided into four 32-bit words. A four-word buffer (a, b, c, d) is used to store those values which are then replaced by the output hash values after each step. MD5 a, b, c, d four words, are also referred to as *chain variables*. The initial values for the MD5 chain variables are shown in Table 7.5.

Table 7.5. Initial Hash Values in Little Endian Format

Normal Values	Little endian format
a = 0x01234567	a = 0x67452301
b = 0x89abcdef	b = 0xefcdab89
c = 0xfedcba98	c = 0x98badcfe
d = 0x76543210	d = 0x10325476

7.3.3 Main Loop

The Main loop is composed of four rounds. Each round has as a 512-bit message block as an input. As it was mentioned, message blocks are grouped into sixteen 32-bit words. The second input comes in the form of chain variables which are also grouped as four words of 32-bit each (totaling 128 bits). All the four rounds use an auxiliary function, which takes three 32-bit inputs producing a single 32-bit output. Table 7.6 presents the four non-linear functions F, G, H, and I, that are utilized in rounds 1 to 4.

Table 7.6. Auxiliary Functions for Four MD5 Rounds

F(A,B,C) = (A AND B) OR ((NOT A) AND C)
G(A,B,C) = (A AND C) OR (B AND (NOT C))
H(A,B,C) = (A XOR B XOR C)
I(A,B,C) = (B XOR (A OR (NOT C)))

All the four non-linear functions are simple and can be easily constructed in reconfigurable hardware. The architecture of those four functions maps well to those reconfigurable devices having a 4-bit input/1-bit output Look Up Tables (LUTs) as a basic unit. On such devices, all the four functions occupy a single LUT, thus using a total of 4 LUTs for one bit manipulation as shown in Fig. 7.6.

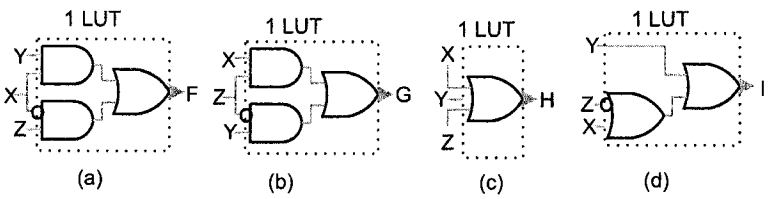

Fig. 7.6. Auxiliary Functions in Reconfigurable Hardware (a) F(X,Y,Z) (b) G(X,Y,Z) (c) H(X,Y,Z) (d) I(X,Y,Z)

Let $\ll S$ denote a left circular shift by S bits and let m_i represent the ith sub-block (0 to 15) of the message. Provided that there is a constant K_j for the jth state of a round, the four operations corresponding to four MD5 rounds are shown in Table 7.7.

Table 7.7. Four Operations Associated to Four MD5 Rounds

$$\begin{aligned}
\text{FF(a,b,c,d, } m_i\text{, S, } K_j) &\Rightarrow \text{a} = \text{b} + ((\text{a} + \text{F(b,c,d)} + m_i + K_j) \ll \text{S}) \\
\text{GG(a,b,c,d, } m_i\text{, S, } K_j) &\Rightarrow \text{a} = \text{b} + ((\text{a} + \text{G(b,c,d)} + m_i + K_j) \ll \text{S}) \\
\text{HH(a,b,c,d, } m_i\text{, S, } K_j) &\Rightarrow \text{a} = \text{b} + ((\text{a} + \text{H(b,c,d)} + m_i + K_j) \ll \text{S}) \\
\text{II(a,b,c,d, } m_i\text{, S, } K_j) &\Rightarrow \text{a} = \text{b} + ((\text{a} + \text{I(b,c,d)} + m_i + K_j) \ll \text{S})
\end{aligned}$$

The architecture of a single MD5 operation can be optimized for reconfigurable devices by re-ordering some steps as shown in Fig. 7.7.

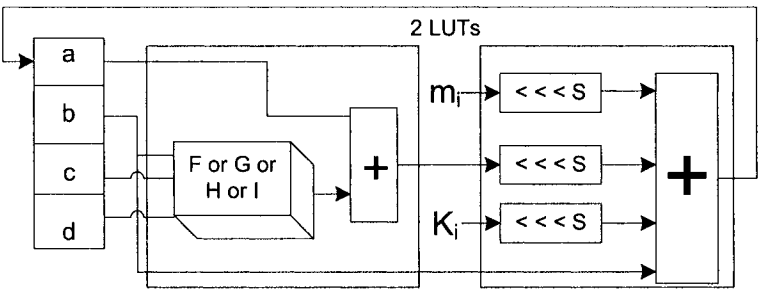

Fig. 7.7. One MD5 Operation

Two changes are introduced. First, summation of word a is appended with the manipulation of the non-linear function, this occupies a single LUT. Similarly, instead of a single shift operation by S bits, a total of three shift operations have been introduced. That does not cost other logic resources but only the routing resources of the target reconfigurable device.

There are a total of 64 steps in the four MD5 rounds. The output of each round for our example message is presented in Table 7.8, Table 7.9, Table 7.10, and Table 7.11 for round 1, round 2, round3, and round 4, respectively. The constant values K_i can be computed by taking the integer part of $2^{32} \times abs(sin(i))$, where i is in radians.

7.3.4 Final Transformation

The last step consists on adding the initial and final hash values. Here addition is a simple integer addition modulo 2^{32} and not an 'XOR' operation. The

Table 7.8. Round 1

Function	Output
FF (a, b, c, d, m_0, 7, 0xd76aa478)	a = 0xbfc20e04
FF (d, a, b, c, m_1, 12, 0xe8c7b756)	d = 0x2445ea9a
FF (c, d, a, b, m_2, 17, 0x242070db)	c = 0xbada24bf
FF (b, c, d, a, m_3, 22, 0xc1bdceee)	b = 0xdae8f105
FF (a, b, c, d, m_4, 7, 0xf57c0faf)	a = 0xd3e2a4f
FF (d, a, b, c, m_5, 12, 0x4787c62a)	d = 0x618adec1
FF (c, d, a, b, m_6, 17, 0xa8304613)	c = 0x605da696
FF (b, c, d, a, m_7, 22, 0xfd469501)	b = 0xb10d4538
FF (a, b, c, d, m_8, 7, 0x698098d8)	a = 0xf0ce7848
FF (d, a, b, c, m_9, 12, 0x8b44f7af)	d = 0xadc2ea19
FF (c, d, a, b, m_{10}, 17, 0xffff5bb1)	c = 0x8ca10c71
FF (b, c, d, a, m_{11}, 22, 0x895cd7be)	b = 0xd06eda96
FF (a, b, c, d, m_{12}, 7, 0x6b901122)	a = 0xcfc79c1a
FF (d, a, b, c, m_{13}, 12, 0xfd987193)	d = 0xef0992d6
FF (c, d, a, b, m_{14}, 17, 0xa679438e)	c = 0x419bb7da
FF (b, c, d, a, m_{15}, 22, 0x49b40821)	b = 0xa41613f9

Table 7.9. Round 2

Function	Output
GG (a, b, c, d, m_1, 5, 0xf61e2562)	a = 0x01816d6a
GG (d, a, b, c, m_6, 9, 0xc040b340)	d = 0x8d2b14de
GG (c, d, a, b, m_{11}, 14, 0x265e5a51)	c = 0xf0ec903d
GG (b, c, d, a, m_0, 20, 0xe9b6c7aa)	b = 0xfbb03b00
GG (a, b, c, d, m_5, 5, 0x0d62f105d)	a = 0x3c1fe25e
GG (d, a, b, c, m_{10}, 9, 0x02441453)	d = 0x53c87df3
GG (c, d, a, b, m_{15}, 14, 0xd8a1e681)	c = 0xefcf863a
GG (b, c, d, a, m_4, 20, 0xe7d3fbc8)	b = 0x7a06c30d
GG (a, b, c, d, m_9, 5, 0x21e1cde6)	a = 0x00fb73e8
GG (d, a, b, c, m_{14}, 9, 0xc33707d6)	d = 0x968fd037
GG (c, d, a, b, m_3, 14, 0xf4d50d87)	c = 0x14952739
GG (b, c, d, a, m_8, 20, 0x455a14ed)	b = 0xcf0e19b2
GG (a, b, c, d, m_{13}, 5, 0xa9e3e905)	a = 0xeec09e98
GG (d, a, b, c, m_2, 9, 0xfcefa3f8)	d = 0xe0cb123e
GG (c, d, a, b, m_7, 14, 0x676f02d9)	c = 0xadfb03b9
GG (b, c, d, a, m_{12}, 20, 0x8d2a4c8a)	b = 0x3d9b93ef

Table 7.10. Round 3

Function	Output
HH (a, b, c, d, m_5, 4, 0xfffa3942)	a = 0x3ae82d36
HH (d, a, b, c, m_8, 11, 0x8771f681)	d = 0xf21c9795
HH (c, d, a, b, m_{11}, 16, 0x6d9d6122)	c = 0x8043a89c
HH (b, c, d, a, m_{14}, 23, 0xfde5380c)	b = 0x3985c48b
HH (a, b, c, d, m_1, 4, 0xa4beea44)	a = 0xf8dd0bbf
HH (d, a, b, c, m_4, 11, 0x4bdecfa9)	d = 0x7a6540bb
HH (c, d, a, b, m_7, 6, 0xf6bb4b60)	c = 0x7263dc17
HH (b, c, d, a, m_{10}, 23, 0xbebfbc70)	b = 0x79d86ca3
HH (a, b, c, d, m_{13}, 4, 0x289b7ec6)	a = 0xaf5015ec
HH (d, a, b, c, m_0, 11, 0xeaa127fa)	d = 0xe9e2e73d
HH (c, d, a, b, m_3, 16, 0xd4ef3085)	c = 0x860d260
HH (b, c, d, a, m_6, 23, 0x4881d05)	b = 0xddfa26e9
HH (a, b, c, d, m_9, 4, 0xd9d4d039)	a = 0x3aace80d
HH (d, a, b, c, m_{12}, 11, 0xe6db99e5)	d = 0xdf9a1e0c
HH (c, d, a, b, m_{15}, 16, 0x1fa27cf8)	c = 0xffda7edc
HH (b, c, d, a, m_2, 23, 0xc4ac5665)	b = 0x4d718018

Table 7.11. Round 4

Function	Output
II (a, b, c, d, m_0, 6, 0xf4292244)	a = 0xbc2cf190
II (d, a, b, c, m_7, 10, 0x432aff97)	d = 0xc43bf785
II (c, d, a, b, m_{14}, 15, 0xab9423a7)	c = 0x9d557285
II (b, c, d, a, m_5, 21, 0xfc93a039)	b = 0xbf063e88
II (a, b, c, d, m_{12}, 6, 0x655b59c3)	a = 0xc5ec3319
II (d, a, b, c, m_3, 10, 0x8f0ccc92)	d = 0x20d2175b
II (c, d, a, b, m_{10}, 15, 0xffeff47d)	c = 0xc6863889
II (b, c, d, a, m_1, 21, 0x85845dd1)	b = 0xf70ea106
II (a, b, c, d, m_8, 6, 0x6fa87e4f)	a = 0x12f76270
II (d, a, b, c, m_{15}, 10, 0xfe2ce6e0)	d = 0xd40a121f
II (c, d, a, b, m_6, 15, 0xa3014314)	c = 0xe4c960a4
II (b, c, d, a, m_{13}, 21, 0x4e0811a1)	b = 0x2fb93bf8
II (a, b, c, d, m_4, 6, 0xf7537e82)	a = 0xadf1d7b5
II (d, a, b, c, m_{11}, 10, 0xbd3af235)	d = 0xfd93443b
II (c, d, a, b, m_2, 15, 0x2ad7d2bb)	c = 0x5a402c56
II (b, c, d, a, m_9, 21, 0xeb86d391)	b = 0x9f2895cb

resultant four words a, b, c, and d would be in little-endian format. They need to be converted back to its original format. Finally, four words a, b, c, and d are concatenated to give the 128-bit hash of the given message as shown in Table 7.12.

Table 7.12. Final Transformation

Initial Hash Values	Round Output	Final Transformation	Conversion from Little Endian
a = 0x67452301	b = 0xefcdab89	c = 0x98badcfe	d = 0x10325476
a = 0xadf1d7b5	b = 0x9f2895cb	c = 0x5a402c56	d = 0xfd93443b
a = 0x1536fab6	b = 0x8ef64154	c = 0xf2fb0954	d = 0x0d508c19
a = 0xb6fa3615	b = 0x5441f68e	c = 0x5409fbf2	d = 0xb198c50d

Final Hash = b6fa36155441f68e5409fbf2b198c50d

7.4 SHA-1, SHA-256, SHA-384 and SHA-512

The FIPS 180-2 [255] supersedes FIPS 180-1 [95]. It includes four secure hash algorithms SHA-1, SHA-224, SHA-384 and SHA-512. SHA-1 is identical to SHA-1 specified in FIPS 180-1[1].

Some notational changes have been introduced to make it consistent with the other three algorithms. All four algorithms are one way iterative hash functions. They differ in terms of block and word size. They also differ in the size of the message digest, which redounds in different levels of security. Table 7.13 compares basic specifications of the four secure hash algorithms.

Table 7.13. Comparing Specifications for Four Hash Algorithms

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest (bits)	Security (bits)
SHA-1	$< 2^{64}$	512	32	160	80
SHA-256	$< 2^{64}$	512	32	256	128
SHA-384	$< 2^{128}$	1024	64	384	192
SHA-512	$< 2^{128}$	1024	64	512	256

[1] Just as it happened with MD5, the SHA family of hash algorithms has been successfully attacked in several recent papers [371, 107].

7.4.1 Message Preprocessing

Preprocessing is always done before hash computation begins. Preprocessing comprises three main steps,

Step 1: Padding the message
Step 2: Parsing the padded message
Step 3: Setting the initial hash values

The hash computation for SHA-1 and SHA-256 requires 512-bit block. A 1024-bit input block is processed by SHA-384 and SHA-512 hash computation. Preprocessing for both categories is discussed separately.

SHA-1 and SHA-256

Step 1: Padding the Message

Let l be the length of the message M in bits. Append bit '1' to the end of the message followed by k zeroes such that the length of the resulting block is 64 bits short of 512 bits, i.e.,

$$\text{Result} = M + 1 + k = 448 \bmod 512.$$

The remaining 64 bits are reserved for adding the message length l in its binary representation. As an example, the message 'try' has an ASCII representation of 24 bits (8×3). Therefore, it requires 423 more bits to be padded at the end of the message in addition to the leading bit '1' in order to complete a block of 448 bits. The message length $l = 24$ in its 64-bit Boolean representation is appended at the end, as shown in Fig. 7.8.

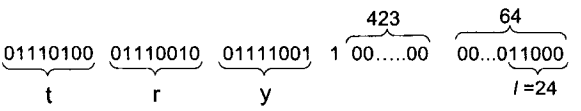

Fig. 7.8. Padding Message in SHA-1 and SHA-256

Padding is always made even if the message block is of 448 bits. For a 448-bit message, a single bit '1' is appended at the end followed by 447 zeroes. Thus, in that case, an apparent single block message would be treated as two separated blocks.

Step 2 : Parsing the message

A padded message is parsed to N 512-bit blocks, namely, $M_0, M_1, \ldots, M_N$. Where each M_i block is organized into sixteen 32-bit blocks, namely, $M_i^0, M_i^1, \ldots, M_i^{15}$. Therefore, the first sixteen 32-bit blocks are: $M_0^0, M_0^1, \ldots, M_0^{15}$.

Step 3: Setting the initial hash values

Before beginning the actual hash function computation, initial values must be set. Those values are provided by the algorithm. Table 7.14 and Table 7.15 show in hex format five 32-bit words for SHA-1 and eight 32-bit words for SHA-256, respectively.

Table 7.14. Initial Hash Values for SHA-1

a = 0x67452301
b = 0xefcdab89
c = 0x98badcfe
d = 0x10325476
e = 0xc3d2e1f0

Table 7.15. Initial Hash Values for SHA-256

a = 0x6a09e667
b = 0xbb67ae85
b = 0x3c6ef372
c = 0xa54ff53a
d = 0x510e527f
e = 0x9b05688c
f = 0x1f83d9ab
g = 0x5be0cd19

SHA-384 and SHA-512

Step 1: Padding the message

Padding procedure for SHA-384 and SHA-512 is similar to those of SHA-1 and SHA-256. However, let us recall that both SHA-384 and SHA-512 operate on 1024-bit message blocks, which consequently causes a change in other lengths. Let l be the length of the message M in bits. In this case, after appending a single bit '1' to the end of the message, k zeroes are added such that the length of the resulting block is 120 bits short of 1024 bits,

$$\text{Result} = M + 1 + k = 896 \bmod 1024$$

The remaining 120 bits are reserved for appending the message length l in its binary representation. Once again, let us consider the same example

message "try" (24 bits). In this case, 871 more bits are required to be padded at the end of the message in addition to the mandatory leading bit '1' to complete a block of 896 bits. The remaining 120 bits represent the message length as shown in Fig.7.9.

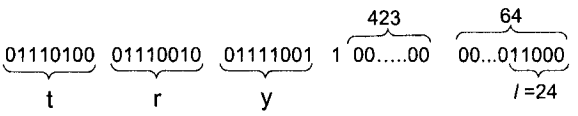

Fig. 7.9. Padding Message in SHA-384 and SHA-512

Step 2 : Parsing the message

Padded messages are parsed to N 1024-bit blocks: $M_0, M_1, \ldots, M_N$. Where each M_i comprises thirty-two 32-bit blocks, namely, $M_i^0, M_i^1, \ldots, M_i^{31}$. The first thirty-two 32 blocks are $M_0^1, M_0^2, \ldots, M_0^{31}$, and so on.

Step 3: Setting the initial hash values

The initial values SHA-384 and SHA-512 comprises two sets of eight 64-bit words as shown in Table 7.16 and Table 7.17.

Table 7.16. Initial Hash Values for SHA-384

a = 0xcbbb9d5dc1059ed8
b = 0x629a292a367cd507
c = 0x9159015a3070dd17
d = 0x152fecd8f70e5939
e = 0x67332667ffc00b31
f = 0x8eb44a8768581511
g = 0xdb0c2e0d64f98fa7
h = 0x47b5481dbefa4fa4

7.4.2 Functions

The auxiliary functions used in SHA-1 differ to those functions used in SHA-256, SHA-384 and SHA-512. Functions used in SHA-256, SHA-384 and SHA-512 are identical but they operate on different word sizes.

Table 7.17. Initial Hash Values for SHA-512

a	= 0x6a09e667f3bcc908
b	= 0xbb67ae8584caa73b
c	= 0x3c6ef372fe94f82b
d	= 0xa54ff53a5f1d36f1
d	= 0x510e527fade682d1
e	= 0x9b05688c2b3e6c1f
f	= 0x1f83d9abfb41bd6b
g	= 0x5be0cd19137e2179

7.4.3 SHA-1

The function F_t in SHA-1 takes three 32-bit words X, Y, and Z, producing a single 32-bit word output, where the variable t ranges from 0 to 79. It is defined as indicated below.

$$F_t = \begin{cases} Ch(X,Y,Z) & = (X\ OR\ Y) \oplus ((NOT\ X)\ OR\ Z) & 0 \leq t \leq 19 \\ Parity(X,Y,Z) & = X \oplus Y \oplus Z & 20 \leq t \leq 39 \\ Maj(X,Y,Z) & = (X\ OR\ Y) \oplus (X\ OR\ Z) \oplus (Y\ OR\ Z) & 40 \leq t \leq 59 \\ Parity(X,Y,Z) & = X \oplus Y \oplus Z & 60 \leq t \leq 79 \end{cases}$$

A reconfigurable hardware architecture for the F_t is illustrated in Fig. 7.10. It is noted that all three, Ch, $Parity$, and Maj, occupy a single LUT when 1-bit operand is processed.

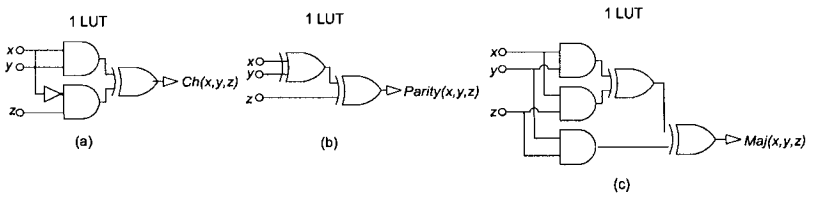

Fig. 7.10. Implementing SHA-1 Auxiliary Functions in Reconfigurable Hardware

SHA-256, SHA-384 and SHA-512

All three, SHA-256, SHA-384 and SHA-512, use six logical functions. Each function operates on three words X, Y, and Z producing a new word of the same size as output. SHA-256 operates on 32-bit long words X, Y and Z. However, both SHA-384 and SHA-512 operates on 64-bit words. The six functions are,

$$
\begin{aligned}
Ch(X,Y,Z) &= (X\ OR\ Y) \oplus ((NOT\ X)\ OR\ Z) \\
Maj(X,Y,Z) &= (X\ OR\ Y) \oplus (X\ OR\ Z) \oplus (Y\ OR\ Z) \\
\Sigma_0(X) &= ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X) \\
\Sigma_1(X) &= ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X) \\
\sigma_0(X) &= ROTR^7(X) \oplus ROTR^{18}(X) \oplus ROTR^3(X) \\
\sigma_1(X) &= ROTR^{17}(X) \oplus ROTR^{19}(X) \oplus ROTR^{10}(X)
\end{aligned}
$$

The architectures for $Ch(X,Y,Z)$ and $Maj(X,Y,Z)$ are identical to the architectures presented in Fig. 7.10. The architectures for Σ_0, Σ_1, σ_0, and σ_1, are also simple. Since the rotation operation can be implemented in reconfigurable hardware by only using routing resources, each of the aforementioned functions can be accommodated into a single LUT as shown in Fig. 7.11.

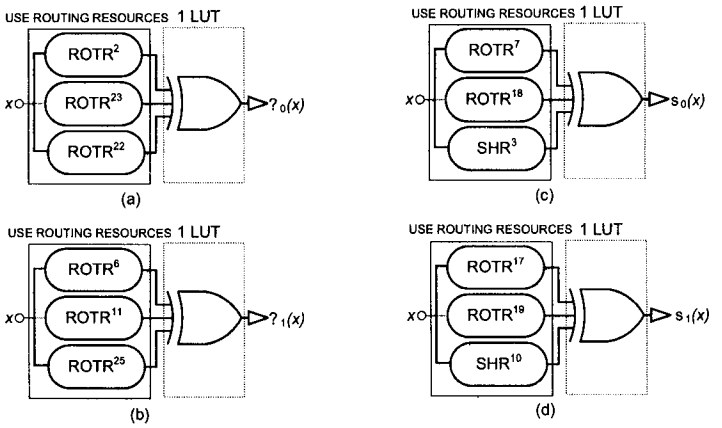

Fig. 7.11. Σ_0, Σ_1, σ_0, and σ_1 in Reconfigurable Hardware

7.4.4 Constants

Constants for SHA-1 and SHA-256 differ. On the other hand, SHA-384 and SHA-512, share the same constant values.

SHA-1

SHA-1 uses eighty 32-bit constant words K_0,K_1,, K_{79} which are given below, in hex format.

$$
K_t = \begin{cases} 0x5a827999 & 0 \le t \le 19 \\ 0x5a827999 & 20 \le t \le 39 \\ 0x8f1bbcdc & 40 \le t \le 59 \\ 0xca62c1d6 & 60 \le t \le 79 \end{cases}
$$

SHA-256

SHA-256 uses sixty four 32-bit different constant words, $K_0, K_1, \ldots, K_{63}$. Those constants are extracted from the first 32 bits of the fractional parts of the first 64 prime numbers' cube roots. They are shown in hexadecimal format in Table 7.18.

Table 7.18. SHA-256 Constants

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1	923f82a4	ab1c5ed5
d807aa98	12835b01	243185be	550c7dc3	72be5d74	80deb1fe	9bdc06a7	c19bf174
e49b69c1	efbe4786	0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147	06ca6351	14292967
27b70a85	2e1b2138	4d2c6dfc	53380d13	650a7354	766a0abb	81c2c92e	92722c85
a2bfe8a1	a81a664b	c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a	5b9cca4f	682e6ff3
748f82ee	78a5636f	84c87814	8cc70208	90befffa	a4506ceb	bef9a3f7	c67178f2

SHA-384 & SHA-512

SHA-384 and SHA-512 use eighty 64-bit different constant words $K_0, K_1, \ldots, K_{79}$. Those constants are extracted from the first 64 bits of the fractional parts of the first 80 prime numbers' cube roots. They are shown in hexadecimal format in Table 7.19.

7.4.5 Hash Computation

The main procedure for hash calculation in SHA-256, SHA-384, and SHA-512 is similar, only the word size varies. SHA-1 hash computation is however different. We can classify the hash calculation procedure of the SHA algorithm family into 3 major steps.

1. Define Word
2. Repeat Operation
3. Final Transformation

SHA-1

- Define Word: After performing message preprocessing for SHA-1, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 80 words for next steps as follows:

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-16}) & 16 \leq t \leq 79 \end{cases}$$

Table 7.19. SHA-384 & SHA-512 Constants

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

- Repeat Operation: A single operation for SHA-1 is shown in Fig. 7.12 which must be repeated 80 times. Let us recall that for the first sub block message, initial values for words a, b,c,d, and e are provided by the algorithm. For the next message sub-blocks, the output hash value of an i^{th} message block serves as initial vector for the hash computation process of the next sub block message. The symbol K_t represents SHA-1 constant values.

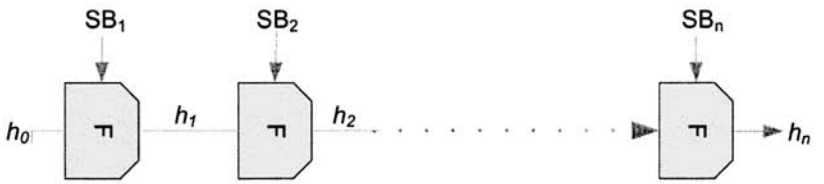

Fig. 7.12. Single Operation for SHA-1

- Final Transformation: Final transformation is simply the addition (modulo 2^{32}) of the initial hash value with the final output hash value of the N^{th} sub block message. A 160-bit hash of the message is then obtained by concatenating five 32-bit words, namely,

$$a \parallel b \parallel c \parallel d \parallel e$$

SHA-256

- Define Word: After performing message preprocessing for SHA-256, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 64 words for next steps as follows[2]:

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) & 16 \leq t \leq 63 \end{cases}$$

- Repeat Operation: A single operation for SHA-256 is shown in Fig. 7.13 which is repeated for 60 times. Similarly as in SHA-1, for the first sub block message, initial values for 8 words a, b,c,d,e,f,g, and h are provided by the algorithm. For next message blocks, output hash values for an i^{th} block message serve as initial vectors for hash calculating process on next sub block message. The symbol K_t represents constant values for SHA-256.

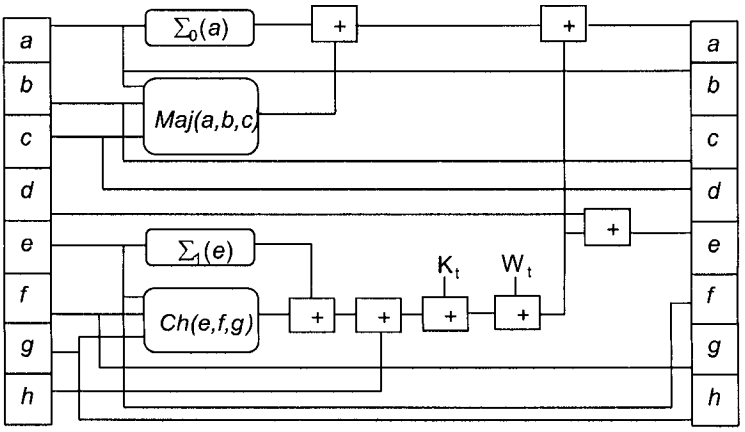

Fig. 7.13. Single Operation for SHA-256

- Final Transformation: Final transformation is simply the addition (modulo 2^{32}) of the initial hash values with the final output hash values of N^{th} message sub block. A 256-bit hash of the message is then obtained by concatenating eight 32-bit words, namely,

$$a \parallel b \parallel c \parallel d \parallel e \parallel f \parallel g \parallel h$$

[2] The operations $\oplus$ and $+$, must not be mixed.

SHA-384

- Define Word: After performing message preprocessing for SHA-384, an i^{th} block message M_n^i $(0 \leq n \leq 15)$, is used to get 80 words for the next steps as follows[3],

$$W_t = \begin{cases} M_t^i & 0 \leq t \leq 19 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) & 16 \leq t \leq 63 \end{cases}$$

  Here addition is performed modulo 2^{64}.
- Repeat Operation: A single operation for SHA-384 is similar to that of SHA-256 as shown in Fig. 7.13. The difference lies in the number of repetitions which are 80, instead of the 60 repetitions of SHA-256.
- Final Transformation: Final transformation consists on the addition (modulo 2^{64}) of the initial hash values with the final output hash values of N^{th} sub block message. A 384-bit message digest is then obtained by truncating the last 2 words. The first six 64-bit words are concatenated as follows.

$$a \parallel b \parallel c \parallel d \parallel e \parallel f$$

SHA-512

The process of hash computation for SHA-512 is quite similar to that of SHA-384. There are only two exceptions. The first one is due to loading the initial values for the 8 words a, b,c,d,e,f,g, and h, which are different for both SHA-384 and SHA-512. The second difference is that a 512-bit message digest is obtained by concatenating all 8 words. Last 2 words are not truncated as it is in the case of SHA-384.

$$a \parallel b \parallel c \parallel d \parallel e \parallel f \parallel g \parallel h$$

7.5 Hardware Architectures

The main moral of the preceding Sections is that hash function computation is iterative in nature. To calculate hash values, several rounds must be performed where each round comprises a certain number of steps. The output of a step serves as input to the next step and the output of a round serves as the input of the next round.

That characteristic does not prevent us from designing a fully pipeline or sub pipeline architecture for hash functions. Let us recall that the input message M is divided into N blocks. Hash computation of a new block cannot start until the hash computation of the previous block has been fully completed. The hash values (output) of the first block are now the initial values

[3] It is noticed that the word size for SHA-384 is 64-bit as compared to SHA-256 which is 32-bit long.

for the hash computation of the second block message. That restricts us from start processing the second block although only a single stage is active and all others are idle during hash computation.

However, different strategies have been proposed by designers in order to improve the data flow at different stages of the design so that high speed gains can be obtained. The different design strategies are discussed in the rest of this Section.

7.5.1 Iterative Design

An iterative design is a natural approach for the implementation of hash functions on hardware platforms. Fig. 7.14 presents an iterative approach for implementing hash algorithms in hardware.

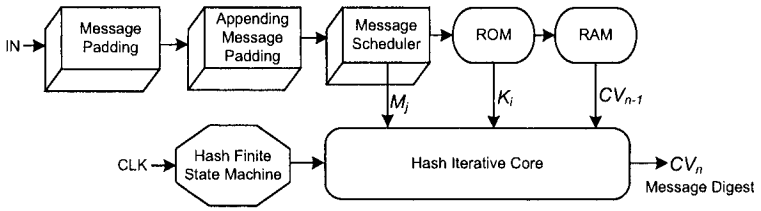

Fig. 7.14. Iterative Approach for Hash Function Implementation

The input message is formatted according to the algorithm requirements in two steps. Those are message padding, and then appending the message length on it. Message scheduler shall provide a sub block or a word derived from some sub blocks for any given algorithm step. Constants provided by the algorithm can be stored in a memory block (ROM). The initial hash values are required till the end of one iteration of the algorithm. This is in order to perform the final transformation (simple XOR with the final output of the iteration). Hence, at the end of a given iteration, partial results must update the input parameters for the next iteration. BRAMs can be used for accomplishing this operation.

The block labeled: "Hash Iterative Core" in Fig. 7.14, includes all logical steps needed for accomplishing a particular compression function computation. The exact sequence of those logical steps (i.e., when should they be executed and with which parameters), is synchronized by the module labeled "Hash Finite State Machine" block. Clearly, the main building blocks of Fig. 7.14 can be altered/combined/modified using different techniques according to the characteristics of the target device and the hash algorithm in hand.

7.5.2 Pipelined Design

In pipeline architectures, registers are provided at different stages of the algorithm. At each clock cycle, the output of a stage is shifted to the next stage. Thus, at the first clock cycle, one input block should be loaded. At the next clock cycle, a second block must be loaded and so on. Once the pipeline is filled, i.e., the final stage outputs a data, then an output value will be ready at each clock cycle.

Pipeline is a fast approach but cost has to be paid in terms of hardware resources. Unfortunately, that approach cannot be fully utilized for hash function computation due to the inherent dependencies. As it was explained, the second iteration cannot be started until the computations for first iteration have been completed. However a sort of pipelining can be achieved for different operations of the similar stage.

7.5.3 Unrolled Design

Unrolled design approach is a useful technique used on the implementation of hash algorithms in order to improve their performance on time. In this approach, all or part of the stages of a hash algorithm are unrolled as is shown in Fig. 7.15a. That however produces long critical paths which causes undesirable long path delays in the circuit. Most designers therefore prefer to unroll some k stages and then to cascade them for the implementation of the whole algorithm as is shown in Fig. 7.15b.

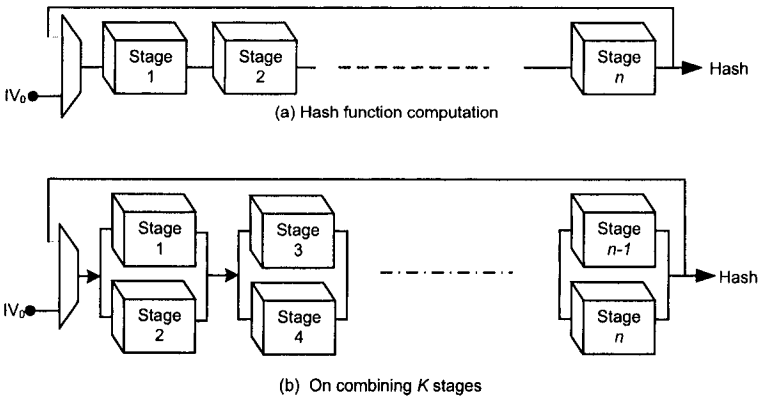

Fig. 7.15. Hash Function Implementation (a) Unrolled Design (b) Combining k Stages

7.5.4 A Mixed Approach

Designing circuits with long critical paths is not useful especially if the target devices are FPGAs. The propagation of long time delays usually implies a performance diminishing. However some registers can be provided as interface buffers between neighbor stages of the hash algorithm. That can be also helpful for producing a more compact design, which will help the mapping synthesis tool. Another enhancement can be made by combining an unrolled design structure with the provision of registers between different stages as shown in Fig. 7.16.

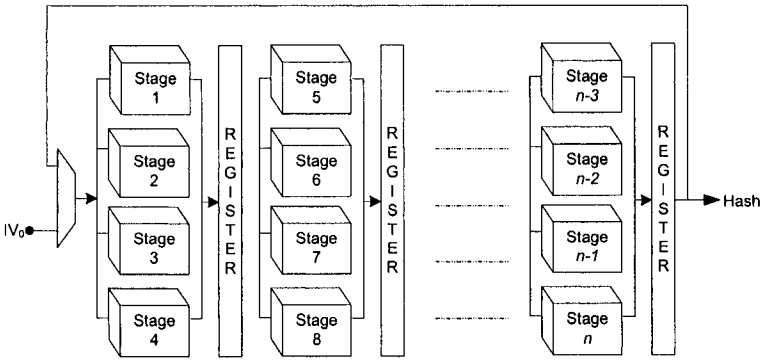

Fig. 7.16. A Mixed Approach for Hash Function Implementation

7.6 Recent Hardware Implementations of Hash Functions

Various hardware implementations of hash algorithms have been reported in literature. Some of them focus on speed optimization while others concentrate on saving hardware resources. Some authors have also tried to exploit parallelism in operations whenever this can be done. Some designs present a tradeoff between time and hardware resources. It has been shown that by adding few registers or few memory units, considerable timing improvements can be obtained.

In the rest of this Section we review some of the most representative hash function hardware designs recently reported. In total, we review six hash function algorithms, namely, MD4, MD5, SHA-1, RIPEMD-160, SHA-2 and Whirpool.

MD4

A single MD4 FPGA architecture has been reported in the open literature [328]. The distinct feature of this design is to try to exploit as much parallelism and pipelining for the MD4 hash algorithm as possible. That design implements arithmetic, logic and circular shift operation using a pipelined parallel processor. It takes 94.07 μS to compute the message digest of a 512-bit input message block at 6.67 MHz frequency consuming only 252 CLB slices.

Table 7.20. MD5 Hardware Implementations

Author(s)	Target Device	Cost	Freq. MHz	Cycles	T† Mbps	T/S
Fastest ASIC MD5 Cores						
Satoh et al. [312]	0.13μm ASIC	17.7K gates	277.8	68	2091	0.117
Compact ASIC MD5 Cores						
Satoh et al. [312]	0.13μm ASIC	10.3K gates	133.3	68	1004	0.097
Helicon [358]	0.18μm ASIC	16K gates	145	65	1140	0.072
Sandra [71]	0.6μm ASIC	10.9K gates + RAM	59	206	146	0.013
Fastest FPGA MD5 Cores						
Jarvinen et al. [156]	Virtex-II XC2V4000-6	11.5K(10) slices(RAM)	75.5	66	5857	0.509
Compact FPGA MD5 Cores						
Helicon [358]	Virtex-II	613(1) slices(RAM)	96	66	744	1.213
Other FPGA MD5 Cores						
Jarvinen et al. [156]	Virtex-II XC2V4000-6	5.7K(0) 647(2) slices(RAM)	80.7 75.5	66 66	2395 586	0.417 0.905
Helicon [358]	Spartan3	630(1) slices(RAM)	63	66	488	0.774
Sandra [71]	Virtex XCV300E	2008 slices	42.9	206	107	0.053
Kang et al. [166]	Apex EP20K1000E	10.5K logic cells	18	65	142	0.0134
Deepak. et al. [65]	Virtex XCV1000-6	880(2) slices(RAM)	21	65	165	0.187

† Throughput

MD5

A considerable number of MD5 hardware implementations have been reported in the open literature. Table 7.20 presents some selected designs. However, due to the availability of a large number of FPGA devices by different manufacturers, with different logic complexity within the basic building block, a comparison of different hash cores becomes complicated.

The ASIC MD5 design in [312] is the fastest one in its category, with a throughput of 2.09 Gbps at a cost of 17,764 gates on a $0.13\mu m$ chip.

The authors in [156] designed several MD5 architectures by unrolling a variable number of MD5 stages. A fully unrolled MD5 architecture is their fastest design, achieving a throughput of 5.8 Gbps by occupying 11498 slices plus 10 BRAMs on a Xilinx Virtex-II XC2V4000-6.

A commercially available MD5 core designed by [358] is a compact design that occupies only 630 slices plus 1 BRAM and reports a throughput of 744 Mbps on a Xilinx Virtex-II device. The throughput over area factor (our figure of merit for measuring efficiency) achieved in [358] is the best one of all designs considered in Table 7.20.

Other MD5 architectures on different FPGA chips using different design approaches are also reported in Table 7.20.

SHA-1

Numerous SHA-1 FPGA implementations have been reported in the literature. A representative group of them are shown in Table 7.21.

The authors in [312] presented two SHA-1 architectures in ASIC hardware, one of them is the fastest architecture reported in the literature, achieving a throughput of 2 Gbps by utilizing 9859 gates in a $0.13\mu m$ chip.

In the reconfigurable hardware category, the fastest design, reported in [67] achieves a throughput of 899.8 Mbps. That is also a compact design with the best throughput over area performance.

A SHA-1 architecture in [120] is the 2^{nd} fastest FPGA core. It utilizes carry save adders to speed up multi-operand additions and to minimize delays with carry propagation. This design reduces the number of operands in a round by pre-computing addition of Constants (K) and Words(W) $(K_t + W_t)$ and also it eliminates the final round which is incorporated as a conditional addition within a round. The throughput for this design is reported as 462 Mbps when operating at a 75.8 MHz clock frequency.

The most compact design for SHA-1 was presented in [71] using as a target device a Xilinx V300E. It proposes a pipelined parallel structure by implementing two arithmetic logic units for SHA-1, achieving a throughput of 119 Mbps at a 59 MHz clock frequency.

The design in [404] utilizes 1622 slices on an Altera EPIK100QC208-1 achieving a throughput of 268.99 Mbps. That is another compact hardware SHA-1 core on Altera devices.

Table 7.21. Representative SHA-1 hardware Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
Fastest ASIC SHA-1 Cores						
Satoh et al [312]	0.13μm ASIC	9.9K gates	333.3	85	2006	0.203
Compact ASIC SHA-1 Cores						
Satoh et al [312]	0.13μm ASIC	7.9K gates	154.3	85	929	0.116
Helicon [358]	0.18μm ASIC	20K gates	166	81	1000	0.050
Sandra [71]	0.6μm ASIC	10.9K + RAM gates	59	255	119	0.011
Compact & Fastest FPGA SHA-1 Cores						
Diez et al [67]	Virtex-II XC2V3000	1.55K slices	38.6	22	899.8	0.580
Grembowski et al [120]	Virtex XCV1000-6	2.2K slices	75.76	84	462	0.210
Other FPGA SHA-1 Cores						
Sandra [71]	Virtex V300E	2.0K slices	42.9	255	86	0.042
Zibin et al [404]	Apex EPIK100Q	1.6K logic cells	43.08	82	268.99	0.165
Kang et al [166]	Apex EP20K1000	10.5K logic cells	18	81	114	0.011
Sklavos [332]	Virtex XCV300	2.6K slices	37		233	0.089

† Throughput

Additionally, there exist other SHA-1 cores [67, 404, 166, 332] which propose some effective techniques to save hardware resources and to increase time factor. In [166], a significant saving of resources was achieved. This design implements a switching matrix by using multiplexers for an appropriate word (W) selection. It can operate at 18 MHz and achieves a throughput of 114 Mbps.

The SHA-1 implementation in [332] was used as a pseudo-random number generator. It is actually a VLSI architecture which was first captured in VHDL and synthesized on FPGAs. That design allows a system frequency of 37 MHz and can run at the rate of 233 Mbps.

Finally, the SHA-1 core in [404] explores three Altera FPGA grades for the same SHA-1 code.

RIPEMD-160

Table 7.22 presents two FPGA architectures for RIPEMD-160, which were implemented on devices made by different manufacturers. The design in [249] is a unified architecture in Altera EPF10K50SBC356-1 for two different hash algorithms:RIPEMD-160 and MD5. That design achieves a throughput over 200 Mbps for MD5 and 84 Mbps for RIPEMD-160 when operating at 26.66 MHz and it stands as the compact and the fastest RIPMD architecture in FPGAs. In [71], a RIPEMD-160 FPGA implementation on Xilinx V300E can run at a 42.9 MHz frequency and achieves a data rate of 89 Mbps.

In ASIC hardware, the fastest RIPEMD architecture is due to [312]. That design can run at 1.442 Gbps by occupying 24755 gates on a $0.13\mu m$ chip.

Table 7.22. Representative RIPEMD-160 FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
Fastest ASIC RIPEMD Cores						
Satoh et al [312]	$0.13\mu m$ ASIC	24775 gates	270.3	96	1442	0.058
		17446 gates	142.9	96	762	0.044
Sandra [71]	$0.6\mu m$ ASIC	10,900 gates + RAM	59	337	89	0.008
Compact & Fastest FPGA RIPEMD Cores						
Ng et al [249]	Apex EPF10K50S-1	1964 logic elements	26.66	162	84	0.042
Sandra [71]	Virtex V300E	2008 slices	42.9	337	65	0.032

† Throughput

SHA-2

Table 7.23 shows several representative SHA-2 hardware cores reported in the open literature.

Authors in [312] reported four ASIC architectures for SHA-224, SHA-256, SHA-384, and SHA-512 implemented on a $0.13\mu m$ chip. The fastest among them is the SHA-512 architecture that achieves a throughput of 2.9 Gbps by using 27297 gates. That is also the fastest ASIC hardware architecture of any SHA-2 family of hash algorithms.

The fastest FPGA SHA-2 architectures have been proposed in [222]. It achieves a throughput of 1466 Mbps on a Xilinx Virtex-II device. The architecture employed for that SHA-2 (512-bit) design consisted on a two-step (2x) unrolled implementation. Authors in [222] essayed six variants of the same design which are named as SHA2 (256) basic, SHA2 (256) 2x-unrolled, SHA2 (256) 4x-unrolled, SHA2 (512) basic, SHA2 (512) 2x-unrolled and SHA2 (512)

Table 7.23. Representative SHA-2 FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
ASIC SHA-2 Cores						
Satoh et al [312]						
SHA-224	0.13μm ASIC	11484 gates	154.1	72	1096	0.095
SHA-256	0.13μm ASIC	15329 gates	333.3	72	2370	0.154
SHA-384	0.13μm ASIC	23146 gates	125.0	88	1455	0.062
SHA-512	0.13μm ASIC	27297 gates	250.0	88	2909	0.106
Helicon [358]						
SHA-256	0.18μm ASIC	22K gates	200	65	1575	0.072
Fastest FPGA SHA-2 Cores						
McEvoy [222]	Virtex-II	4107 slices	65.893	46	1466	0.357
SHA-2(512)	XC2V2000					
Compact FPGA SHA-2 Cores						
Sklavos et al [333]	Virtex	1060 slices	83		326	0.307
SHA-2(256)	XCV200-6					
Other FPGA SHA-2 Cores						
Sklavos et al [333]	Virtex	1966 slices	74		350	0.178
SHA-2(384)	XCV200-6					
Sklavos et al [333]	Virtex	2237 slices	75		480	0.214
SHA-2(512)	XCV200-6					
McLoone et al [224]	Virtex	2914 slices +	38	80	479	0.164
SHA-2(384)	XCV600E-8	2 BRAMs				
McLoone et al [224]	Virtex	2914 slices	38	80	479	0.164
SHA-2(512)	XCV600E-8	2 BRAMs				
McEvoy [222]						
SHA-2(256)						
(Basic)	Virtex-II XC2V2000	1373 slices	133.06	68	1009	0.734
(2x-unrolled)	Virtex-II XC2V2000	2032 slices	73.975	38	996.7	0.490
(4x-unrolled)	Virtex-II XC2V2000	2898 slices	40.833	23	908.9	0.313
McEvoy [222]						
SHA-2(512)						
(Basic)	Virtex-II XC2V2000	2726 slices	109.03	84	1329	0.487
(4x-unrolled)	Virtex-II XC2V2000	5807 slices	35.971	27	1364	0.234

† Throughput

4x-unrolled. Those architectures optimize time performances by combining pipelining and unrolling techniques.

In [333], a common architecture is customized for three SHA2 algorithms: SHA2 (256), SHA2 (384) and SHA2 (512). The design compares three implementations in terms of operating frequency, throughput and area-delay product. Among them, SHA2 (256) FPGA implementation consumes least hardware resources in the literature, achieving a throughput of 326 Mbps on a Xilinx V200PQ240-6.

In [224], a single chip FPGA implementation is also presented for SHA2 (384) and SHA2 (512). That architecture optimizes time factor and hardware area by using shift registers for message scheduler and compression block. Similarly, block select RAMs (BRAMs) are used to store the compression function constants.

Table 7.24. Representative Whirlpool FPGA Implementations

Author(s)	Target Device	Hardware	Freq. MHz	Cycles	T† Mbps	T/S
Fastest FPGA Whirlpool Cores						
McLoone et al [226] 2×unrolled	Virtex-4 X4VLX100	13210 slices	47.8		4896	0.370
Kitsos et al [173] LUT based Time optimized	Virtex XCV1000E	5585 slices	87.5	10	4480	0.802
Compact FPGA Whirlpool Cores						
Pramstaller et al [274]	Virtex-2P XC2VP40	1456 slices	131		382	0.262
Other FPGA Whirlpool Cores						
Kitsos et al [173] Boolean expression based	VirtexE XCV1000E	3815 slices	75	20	1920	0.503
Kitsos et al [173] LUT based	VirtexE XCV1000E	3751 slices	93	20	2380	0.634
Kitsos et al [173] Boolean expression based Time optimized	VirtexE XCV1000E	5713 slices	72	10	3686	0.645
McLoone [226]	Virtex-4 X4VLX100	4956 slices	93.56		4790	0.966

† Throughput

Whirlpool

Table 7.24 lists various Whirlpool FPGA-based architectures. The fastest Whirlpool core has been reported in [226]. That is a 2 stages (2x) unrolled Whirlpool architecture implemented on a Xilinx Virtex-4 which achieves a throughput of 4896 Mbps by consuming 13210 CLB slices.

Another Whirlpool core showing similar throughput to the design in [226] is due to [173] which reports a throughput of 4480 Mbps on a Xilinx XCV1000 by occupying 5585 CLB slices and also some dedicated memory modules. Three more variants of that design are also presented. Those architectures implement Whirlpool mini boxes by using Boolean expressions, referred to as BB (Boolean expressions Based) and by using FPGA LUTs, referred to as LB (LUT Based) respectively. Let us call them as Whirlpool BB and Whirlpool LB. Both Whirlpool BB and Whirlpool LB can operate at rates of 1920 Mbps and 2380 Mbps. Both architectures are further optimized for time, increasing throughputs to 3686 Mbps and 4480 Mbps.

In contrast to the aforementioned architectures, a compact FPGA implementation of Whirlpool hash function was reported in [274]. That architecture focuses on saving considerable hardware resources by using LUT-based RAM for Whirlpool state. Authors report a hardware cost of just 1456 CLB slices achieving a data rate of 382 Mbps.

7.7 Conclusions

In this chapter, various popular hash algorithms were described. The main emphasis on that description was made on evaluating hardware implementation aspects of hash algorithms.

MD5 description included in this Chapter can be regarded as a step by step example of how intermediate values are being updated during algorithm execution. We have mentioned that MD5 design methodology has a strong influence in almost all modern hash functions. The explanation provided for SHA family of hash algorithms can be regarded as an evidence that the structure of current hash algorithms borrows basic rules and principles from their predecessors.

A fair number of hash function implementations in reconfigurable Hardware have been reported so far. Those architectures do not pretend to be a universal solution for all the universe of hash applications such as, secure web traffic (https /SSL), encrypted e-mail(PGP, S/MIME), digital certificates, cryptographic document authenticity, secure remote access (ssh/sftp), etc.

However, the usage of reconfigurable hardware for hash function implantations can provide a unique benefit of reconfiguring customized hardware architecture according to the specifications of end users. Furthermore, given the fact that most hash functions are enduring difficult times, where several emblematic hash functions have been critically attacked, new security patches could be easily incorporated.

8

General Guidelines for Implementing Block Ciphers in FPGAs

This chapter pretends to provide general guidelines for the efficient implementation of block ciphers in reconfigurable hardware platforms. The general structure and design principles for block ciphers are discussed. Basic primitives in block ciphers are identified and useful design techniques are studied and analyzed in order to obtain efficient implementations of them on reconfigurable devices. As a case of study, those techniques are applied to the Data Encryption Standard (DES), thus producing a compact DES core.

8.1 Introduction

Block ciphers are based on well-understood mathematical problems. They make extensive use of non-linear functions and linear modular algebra [227]. Most block ciphers exhibit a highly regular structure: same building blocks are applied a predetermined number of times. Generally speaking, block ciphers are symmetric in nature. Sometimes encryption and decryption only differ in the order that sub-keys are used (either ascending or descending order). Thus, quite often pretty much the same machinery can be used for both processes.

Implementation of block ciphers mainly use bit-level operations and table look-ups. The bit-level operations include standard combinational logic operations (such as XORs, AND, OR, etc.), substitutions, logical shifts and permutations, etc. Those operations can be nicely mapped to the structure of FPGA devices. In addition, there are built-in dedicated resources like memory modules which can be used as a Look Up Tables (LUTs) to speedup the substitution operation, which is one of the key transformations of modern block ciphers. Furthermore, contemporary FPGAs are capable of accommodating big circuits making possible to generate highly parallel crypto cores. All these features combine together for providing spectacular speedups on the implementation of crypto algorithms in reconfigurable devices.

In this chapter, we analyze key block ciphers characteristics. We explore general strategies for implementing them on FPGA devices. We search for the most frequent operations involved in their transformations and develop strategies for their implementations in reconfigurable devices. It has been already pointed out how bit level parallelism can be greatly exploited in FPGAs. As we will see, this fact is especially true for block ciphers. As a way of illustration, we test our methodology in one specific case of study: the Data Encryption Standard (DES). Furthermore, in the next Chapter our strategies are also applied to the Advanced Encryption Standard (AES).

DES is the most popular, widely studied and heavily used block cipher. It has been around for quite a long time, more than thirty years now [64, 92]. It was developed by IBM in the mid-seventies. The DES algorithm is organized in repetitive rounds composed of several bit-level operations such as logical operations, permutations, substitutions, shift operations, etc. Although those features are naturally suited for efficient implementations on reconfigurable devices, DES implementations can be found on all platforms: software [64, 92, 169, 25, 23], VLSI [78, 76, 381] and reconfigurable hardware using FPGA devices [204, 384, 167, 99, 225, 381, 271]. In this Chapter, we present an efficient and compact DES architecture especially designed for reconfigurable hardware platforms.

The rest of this Chapter is organized as follows. Section 8.2 describes the general structure and design principles behind block ciphers. Emphasis is given on useful properties for the implementation of block ciphers in FPGAs. An introduction to DES is presented in Section 8.3. In Section 8.4, design techniques for obtaining an efficient implementation of DES are explained. In Section 8.5 a survey of recently reported DES cores is given. Finally, concluding remarks are drawn in Section 8.6.

8.2 Block Ciphers

In cryptography, a block cipher is a type of symmetric key cipher which operates on groups of bits of some fixed length, called blocks. The block size is typically of 64 or 128 bits, though some ciphers support variable block lengths. DES is a typical example of a block cipher, which operates on 64-bit plaintext block. Modern symmetric ciphers operate with a block length of 128 bits or more. Rijndael (selected in October, 2000 as the new Advanced Encryption Standard), for instance, allows block lengths of 128, 192, or 256 bits.

A block cipher makes use of a key for both encryption and decryption. Not always the key length matches the block size of the input data. For example, in triple DES or 3DES for short (a variant of DES), a 64-bit block is processed using a 168-bit key (three 56-bit keys) for encryption and decryption. Rijndael allows various combinations of 128, 192, and 256 bits for key and input data blocks.

As it was already mentioned in §2.7 Some of the major factors that determine the *security strength* of a given symmetric block cipher algorithm include, the quality of the algorithm itself, the key size used and the block size handled by the algorithm. Block lengths of less than 80 bits are not recommended for current security applications [253].

In the rest of this Section, general structure and design principles of the block ciphers are discussed. We explain several primitives which commonly form part of the repertory of block cipher transformations. Finally, we give some comments about their hardware implementation, specifically on reconfigurable type of hardware.

8.2.1 General Structure of a Block Cipher

As is shown in Figure 8.1, there are three main processes in block ciphers: encryption, decryption and key schedule. For the encryption process, the input is *plaintext* and the output is *ciphertext*. For the decryption process, ciphertext becomes the input and the resultant output is the original plaintext. A number of rounds are performed for encryption/decryption on a single block. Each round uses a round key which is derived from the cipher key through a process called *key scheduling*. Those three processes are further discussed below.

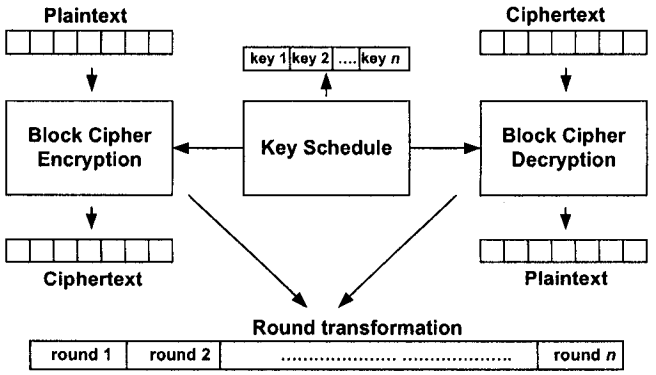

Fig. 8.1. General Structure of a Block Cipher

Block Cipher Encryption

Many modern block ciphers are Fiestel ciphers [342]. Fiestel ciphers divide input block into two halves. Those two halves are processed through n number of rounds. In the final round, the two output halves are combined to produce a single ciphertext block. All rounds have similar structure. Each round uses

a round key, which is derived from the previous round key. The round key for the first round is derived from the user's master key. In general all the round keys are different from each other and from the cipher key.

Many modern block ciphers partially or completely employ a similar Fiestel structure. DES is considered a perfect Fiestel cipher. Modern block ciphers also repeat n rounds of the algorithm but they do not necessarily divide the input block into two halves. All the rounds of the algorithm are generally similar if not identical. Round operations normally include some non-linear transformations like substitution and permutation making the algorithm stronger against cryptanalytic attacks.

Block Cipher Decryption

As it was explained, one of the main characteristics of a Fiestel cipher is the usage of a similar structure for encryption and decryption processes. The difference lies on the order that the round keys are applied. For decryption, round keys are used in reverse order as that of encryption. Modern block ciphers also use round keys following a similar style, however, encryption and decryption processes for some of them may not be the same. In any case, they preserve the symmetric nature of the algorithm by guaranteeing that each transformation will always have its corresponding inverse. As a result both, the encryption and decryption processes tend to appear similar in structure.

Key Schedule

The round keys are derived from the user key through a process called *key scheduling*. Block ciphers define several transformations for deriving the round keys to be utilized during the encryption and decryption processes. For some of them, round keys for decryption are derived using reverse transformations. Alternatively, keys derived for encryption can be simply used during the decryption process in reverse order.

8.2.2 Design Principles for a Block Cipher

During the last two decades both, theoretical new findings as well as innovative and ingenious practical attacks have significantly increase the vulnerability of security services. Every day, more effective attacks are launched against cryptographic algorithms. We also have seen a tremendous boost in computational power. Successful exhaustive key search engines have been developed in software as well as in hardware platforms. As a consequence of this, old cryptographic standards were revised and new design principles were suggested to improve current security features. In this subsection, we analyze some of the key features that directly impact the design of a block cipher.

Key Size

If a block cipher is said to be highly resistant against brute force attack, then its strength is determined by its key length: the longer the key, the longer it takes before a brute force search can succeed. This is one of the reasons why, modern block ciphers employ key lengths of 128 bits or more.

Variable Key Length

On the one hand, longer keys provide more security against brute force attacks. On the other hand, a large key length may slow down data transmission due to low encryption speed. Modern block ciphers therefore offer variable key lengths in order to support different security and encryption speed compromises. All the five finalists of the 2000 competition for selecting the new advance encryption standard, namely, RC6, Twofish, Serpent, MARS and Rijndael, provide variable key lengths.

Mixed Operations

In order to make the job of a cryptanalyst more complex, it is considered useful to apply more than one arithmetic and/or Boolean operators into a block cipher. This approach adds more non-linearity producing complex functions as an alternative to S-boxes (substitution boxes). Mixed operations are also used in the construction of S-boxes to add non-linearity thus making them produce more unpredictable results.

Variable Number of Rounds

Round functions in crypto algorithms add a great deal of complexity, which implies that the crypto-analysis process becomes significantly less amenable. By increasing the number of rounds larger *safety margins* are provided. On the contrary, a large number of rounds slows cipher encryption speed. Modern block ciphers provide variable number of rounds allowing users to trade security by time. It should be noticed that the strength of a given crypto algorithm is also linked with the other design parameters. For example, AES with 10 rounds provides higher security as compared to DES with 16 rounds.

Variable Block Length

The security of a block cipher against brute force attacks is dependent upon key and block lengths. Longer keys and block lengths obviously imply a bigger search space, which tend to give more security to a cipher algorithm. As it has been said, modern ciphers support variable key and block lengths, thus assuring that the algorithm becomes more flexible according to different security requirement scenarios.

Fast Key Setup

Blowfish uses a lengthy key schedule. Therefore, the process of generating round keys for encrypting/decrypting a single data block may take a significant amount of time. On the other hand, this characteristic also adds security to Blowfish in the sense that it greatly magnifies the time to search all possibilities for round keys. However for those applications where the cipher key must be changed frequently, a fast key setup is needed. For example, overheads due to key setup during the encryption of the security Internet protocol (IPSec) packets are quite considerable. That is why most modern block ciphers offer simple and fast key schedule algorithms. Rijndael Key schedule algorithm is a good example of an efficient process for round key generation.

Software/Hardware Implementations

It was the time when crypto algorithms were designed to get an efficient implementation on 8-bit processors. Most of their arithmetic/logical functions were designed to operate on byte level. Perhaps, encryption speed was not a *must have* issue as it is now. Those times has gone for good. There are applications which require high encryption speeds either for software or for hardware platforms. This is why cryptographers started to include those functions in crypto algorithms which can be efficiently executed in both software and hardware platforms. For example, the XOR operation can be found in virtually all modern block ciphers, among other reasons, because of its efficiency when implemented in software as well as in hardware platforms.

Simple Arithmetic/Logical Operations

A complex crypto algorithm might not be strong enough cryptographically. The attribute of *simplicity* can be seen in most of the strong block ciphers used nowadays. They mainly include easily understandable bit-wise operations.

Table 8.1 describes key features for some famous block ciphers including the five finalists (AES, MARS, RC6, Serpent, Twofish) of the NIST-organized contest for selecting the new Advanced Encryption Standard. It can be seen that modern block ciphers use high block lengths of 128 bits or more. Similarly they provide high key lengths up till 448 bits. Both block and key lengths in block ciphers are often variable to trade the security and speed for the chosen algorithm. Number of rounds ranges from 8 to 32. For some block ciphers the number of round is fixed but for some others that number can vary depending on the chosen block and key lengths.

It is noticed that most block ciphers can be efficiently implemented in software and hardware platforms. All block ciphers generally include bit-wise (XOR, AND) and shift or rotate operations. Excluding a small minority of block ciphers, most algorithms use the so-called *S-boxes* for substitution. Fast key set-up is an important feature among modern block ciphers. They are

Table 8.1. Key Features for Some Famous Block Ciphers

Properties	DES	Blowfish	IDEA	AES	MARS	RC6	Serpent	TwoFish
Block length	64	64	64	128-256	128	128	128	128
Key length	64	32-448	128	128-256	128-448	128-256	256	128-192
No. of rounds	16	16	8	10-14	32	20	32	16
Software	√	√	√	√	√	√	√	√
Hardware	√	√	√	√	√	√	√	√
Symmetric	√	√	√	×	×	×	×	√
Bit-operations	√	√	√	√	√	√	√	√
Permutation	√	×	×	×	×	×	√	√
S-Box	√	√	×	√	√	×	√	√
Shift/rotate	√	×	√	√	√	√	√	√
Fast key setup	√	×	√	√	√	√	√	√

not always symmetric, that is, same building blocks used for encryption not necessarily can be used for decryption.

8.2.3 Useful Properties for Implementing Block Ciphers in FPGAs

Hardware implementations are intrinsically more physically secure: key access and algorithm modification is considerably harder. In this subsection we identify some useful properties in symmetric ciphers that have the potential of being nicely mapped to the structure of reconfigurable hardware devices.

Bit-Wise Operations

Most of the block ciphers include bit-level operations like AND, XOR and OR which can be efficiently implemented and executed in FPGAs. Indeed, those operations utilize a relatively modest amount of hardware resources. The primitive logic units in most of the FPGAs are based on 4-input/1-ouput configuration. This useful feature of FPGAs allow to build 2, 3, or 4 input Boolean function using the same hardware resources as shown in Figure 8.2.

Substitution

Substitution is the most common operation in symmetric block ciphers which adds maximum non-linearity to the algorithm. It is usually constructed as a look-up table referred to as substitution box (S-Box). The strength of DES heavily depends on the security robustness of its S-boxes. AES S-box is used in both encryption and decryption processes and also in its key schedule algorithm.

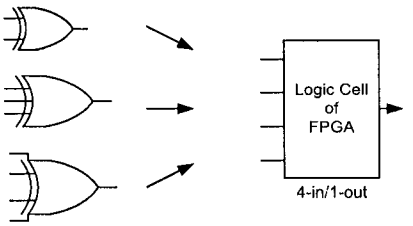

Fig. 8.2. Same Resources for 2,3,4-in/1-out Boolean Logic in FPGAs

Formally, an S-box can be defined as a mapping of n input to m output bits, i.e., $F : Z_2^n \rightarrow Z_2^m$. When $n = m$ the mapping is reversible and therefore it is said to be bijective. AES has only one S-Box, which happens to be reversible, but all eight DES S-boxes are not[1].

FPGA devices offer various solutions for the implementation of substitution operation as shown in Figure 8.3.

- The primitive logic unit in FPGAs can be configured into memory mode. A 4-in/1-out LUT provides 16×1 memory. A large number of LUTs can be combined into a big memory. This might be seen as a fast approach because the S-Box pre-computed values can be stored, thus saving valuable computational time for S-Box manipulation.
- The values for S-boxes in some block ciphers can also be calculated. In this case, if the target device does not contain enough memory, then one can use combinational logic to implement S-boxes. That could be rather slow due to large routing overheads in FPGAs.
- Some FPGA devices contain built-in memory modules. Those are fast access memories which do not make use of primitive logic units but they are integrated within FPGAs. The pre-computed values for S-boxes can be stored in those dedicated modules. That could be faster as compared to store S-box values in primitive logic units configured into memory mode. As it was described in Chapter 3, many FPGA devices from different manufacturers contain those memory blocks, frequently called BRAMs.

Permutation

Permutation is a common block cipher primitive. Fortunately, there is no cost associated with this operation since it does not make use of FPGA logic

[1] It is noticed that the number of candidate Boolean functions for building an n bit input/m bit output S-box is given as 2^{m2^n}. It follows that even for moderated values of n and m, the size of the search space becomes huge. However, not all Boolean functions are suitable for building robust S-Boxes. Some of the desired cryptographic properties that good candidate Boolean functions must have are: High non-linearity, high algebraic degree and low auto-correlation, among others.

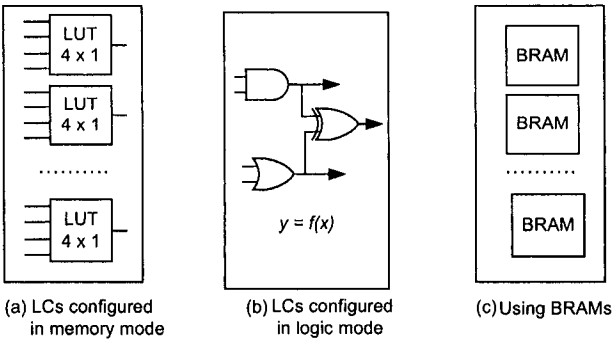

Fig. 8.3. Three Approaches for the Implementation of S-Box in FPGAs

resources. It is just rewiring and the bits are rearranged (concatenated) in the required order. Figure 8.4 demonstrates a simple example of permuting 6 bits only. That strategy can be extended for the permutation operation over longer blocks.

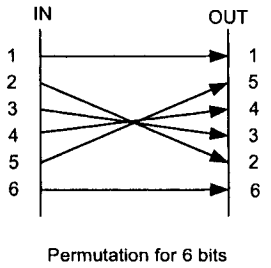

Fig. 8.4. Permutation Operation in FPGAs

Shift & Rotate

Shift is simpler than the permutation operation. Shift operation is normally performed by extracting some particular bit/byte values from a larger register. One practical example of this situation is: retrieving a 6-bit sub-vector from a 48-bit state register for their further substitution in DES. This operation can be implemented using wide data buses, which are then divided into small buses carrying the required bit/byte values. A typical byte-level shift operation is shown in Figure 8.5a.

In some cases, the input data is shifted n bits and n zeroes are added, a process known as *zero padding*. In FPGAs, zero padding for n bits is achieved by simply connecting n bits to the ground as shown in Figure 8.5b.

Most block ciphers (such as AES, RC6, DEAL, etc.) use the rotation operation. It is similar to shift operation but with no zero padding. Instead, bit wires are re-grouped according to a defined setup. For example, for a 4-bit buffer, shifting left $a_0a_1a_2a_3$ by 1-bit becomes $a_1a_2a_30$, whereas rotating left by 1-bit produces $a_1a_2a_3a_0$.

Fixed rotation is trivial and there is no cost associated with it. Variable rotation is also used by some cryptographic algorithms (RC5, RC6, CAST) however this is not a trivial operation anymore.

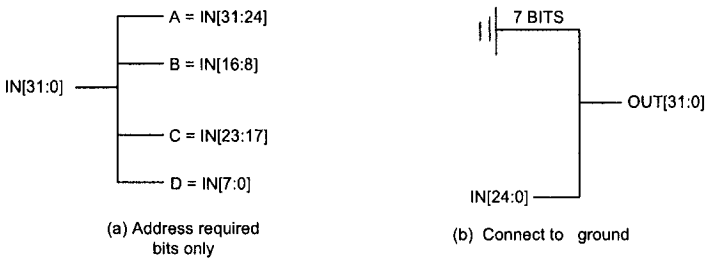

Fig. 8.5. Shift Operation in FPGAs

Iterative Design Strategy

Block ciphers are naturally iterative, that is, n iterations of the same transformations, normally called *rounds*, are made for a single encryption/decryption. An iterative design strategy is a simple approach which implements the cipher algorithm by executing n iterations of its rounds. Therefore, n clock cycles are consumed for encrypting/decrypting a single block, as shown in Figure 8.6. Obviously, this is an economical approach in terms of required hardware area. But it slows cipher speed which is n times slower for a single encryption. Such architectures would be useful for those applications where available hardware resources are limited and speed is not a critical factor.

Pipeline Design Strategy

In a pipeline design, all the n rounds of the algorithm are unrolled and registers are provided between two consecutive rounds as shown in Figure 8.7. All the intermediate registers are triggered at the same clock by shifting data to the next stage at the rising/falling edge of the clock. Once all the pipeline stages are filled, the output blocks starts appearing at each successive clock cycle.

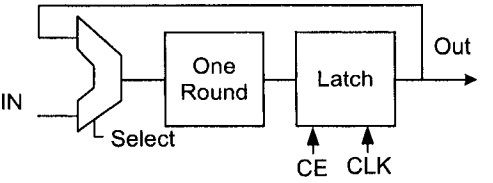

Fig. 8.6. Iterative Design Strategy

This is a fast solution which increases the hardware cost to approximately n times as compared to an iterative design.

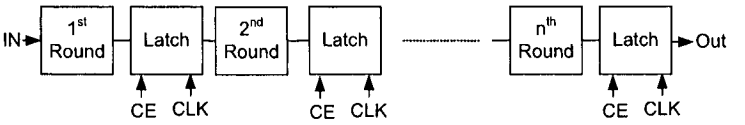

Fig. 8.7. Pipeline Design Strategy

Sub-pipelining Design Strategy

Figure 8.8 represents a sub-pipeline design strategy. As shown in Figure 8.8, Sub-pipelining is implemented by placing the registers between different stages of a single round for a pipeline architecture. That improves performance of the pipeline architecture as those internal registers shift the results within the round when outputs of a round are being transferred to the next round. It has been experimentally demonstrated that careful placement of those registers within a round may produce a significant increase in the design performance.

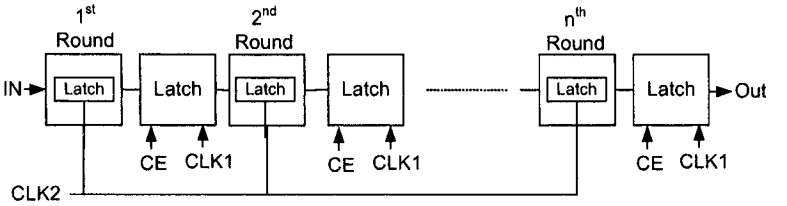

Fig. 8.8. Sub-pipeline Design Strategy

Managing Block Size

Modern block ciphers operate on data blocks of 128 bits or more. Unlike software implementations on general-purpose microprocessors, FPGAs allow parallel execution of the whole data block provided that there is no data dependency in the algorithm. Therefore, it is always useful to dissection the cipher algorithm looking for possible parallelization versions of it. Furhtermore, FPGAs offer more than 1000 external pins to be programmable for inputs or outputs. This is advantageous when the communication is needed with several peripheral devices on the same board simultaneously.

Key Scheduling

Fast key setup is one of the characteristics in modern block ciphers. The keys are required to be changed rapidly in some cryptographic applications. It is possible to reconfigure FPGA device for the key schedule module only whenever a change in the key is desired

Key Storage

It is recommendable for cryptographic applications to make use of different secret keys for different sessions. FPGAs provide enough memory resources to store various session keys. As the keys are stored inside an FPGA, it is therefore valid to say that FPGA implementations are physical secure[2].

8.3 The Data Encryption Standard

On August, 1974, IBM submitted a candidate (under the name LUCIFER) for cryptographic algorithm in response to the 2nd call from National Bureau of Standards (NBS), now the National Institute of Standards & Technology (NIST)[253], to protect data during transmission and storage.

NBS launched an evaluation process with the help of National Security Agency (NSA) and finally adopted on July 15, 1977, a modification of LUCIFER algorithm as the new Data Encryption Standard (DES). The Data Encryption Standard [392], known as Data Encryption Algorithm (DEA) by the ANSI [392] and the DEA-1 by the ISO [152] remained a worldwide standard for a long time until it was replaced by the new Advanced Encryption Standard (AES) on October 2000.

DES and TripleDES provide a basis for comparison of new algorithms. DES is still used in IPSec protocols, ATM encryption, and the secure socket layer (SSL) protocol. It is expected that DES will remain in the public domain

[2] See §3.7 for more details on the security offered by contemporary reconfigurable hardware devices.

for a number of years. DES expired as a federal standard in 1998 and it can only be used in legacy systems. Nevertheless, DES continues to be the most widely deployed symmetric-key algorithm. Its variant, Triple-DES, which consists on applying three consecutive DES without initial (direct and inverse) permutations between the second and the third DES, coexists as a federal standard along with AES.

A detail description of the DES algorithm can be seen in [317, 228, 362]. The description of DES in this chapter it closely follows that of [317].

Description

DES uses a 64-bit long key. The eight bits of that key are used for odd parity and therefore they are not counted in the key length. The effective key length is therefore 56 bits, providing 2^{56} possible keys. DES is a block cipher: It encrypts/decrypts data in 64-bit blocks using a 56-bit key. DES is a symmetric algorithm: the same algorithm and the key are used for both encryption and decryption. DES is an iterative cipher: the basic building block (a substitution followed by a permutation) called a *round* is repeated 16 times. For each DES round, a sub-key is derived from the original key through the process of key scheduling. Although the key scheduling algorithm for encryption and decryption is exactly the same, produced round keys for decryption are used in reverse order. Figure 8.9 shows the basic algorithm flow for both the encryption and key schedule processes.

Encryption begins with an *initial permutation* (IP), which scrambles the 64-bit plain-text in a fixed pattern. The result of the initial permutation is sent to two 32-bit registers, called the *right half* register, R_0 and *left half* register, L_0. Those registers hold the two halves of the intermediate results through successive 16 applications of the function f_k which is given by ($n =$ 0 to 15):

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} + f(R_{n-1}, K_n) \end{aligned} \tag{8.1}$$

After 16 iterations, the contents of the right and left half registers are passed through the final permutation IP^{-1}, which is the inverse of the initial permutation. The output of IP^{-1} is the 64-bit ciphertext.

A detailed explanation of those three operations is provided in the rest of this Subsection. The key sechedule algorithm of DES is explained at the end.

3.3.1 The Initial Permutation (IP^{-1})

The initial permutation is the first operation applied to the input 64-bit block before the main iterations of the algorithm start. It transposes the input block as described in Table 8.2. For example, the initial permutation moves bit 58 to bit position 1, bit 50 to bit position 2, bit 42 to bit position 3, and so forth.

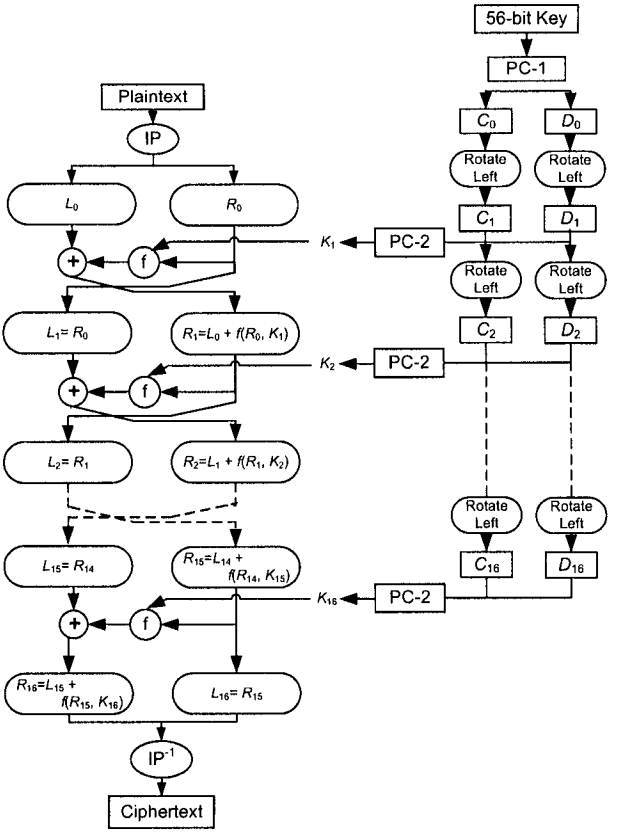

Fig. 8.9. DES Algorithm

The initial permutation has no cryptographic relevance on DES security. Its primary purpose is to make it easier for an application to load plain-text into a DES chip in byte-sized pieces. Initial permutation implementation in hardware is trivial but cumbersome in software.

8.3.2 Structure of the Function f_k

The 64-bit output from the initial permutation is divided into two halves L_0 and R_0 of 32 bits each as shown in Figure 8.9. Both halves go through the 16 iterations of the function f_k (Eq. 8.1) which is described below.

For the first iteration, R_0 and 48-bit round key are the two inputs. We first expand R_0 from 32 bits to 48 bits by using the expansion permutation (Permutation E).

Table 8.2. Initial Permutation for 64-bit Input Block

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

The Expansion Permutation (Permutation E)

This operation expands 32-bit right half R_i to 48 bits. Some bits are therefore repeated and the order of the bits is also changed as shown in Table 8.3.

Table 8.3. E-bit Selection

32	1	2	3	4	**5**	**4**	5	6	7	8	**9**
8	9	10	11	12	**13**	**12**	13	14	15	16	**17**
16	17	18	19	20	**21**	**20**	21	22	23	24	**25**
24	25	26	27	28	**29**	**28**	29	30	31	32	**1**

Table 8.3 shows the position of input bits after applying the permutation E. For example, the bit in position 3 of input block moves to position 4, bit 21 moves to position 30 and 32 of the output block. The redundant bits and their positions in the output block can be easily seen as they are outside the squares in boldface letter as shown in Table 8.3.

This operation has two purposes. First, it makes the size of right half register equal to the size of the key to perform XOR operation. Second, the 48-bit expanded register can be compressed during the substitution operation.

The output 48-bit is XORed with the 48-bit round key which is then divided into 6-bit long eight groups. The eight groups each of six bits are replaced to eight groups of four bits each by applying the substitution boxes (S-Boxes) whose values are provided by the algorithm.

The S-Box Substitution

DES S-box is a 64-entry table arranged into four rows and sixteen columns. The input is a 6-bit address and the output is 4-bit long. The first and last bits a_0a_5 of 6-bit address $a_0a_1a_2a_3a_4a_5$ represent the row number while the middle four bits $a_1a_2a_3a_4$ denote the column number. Thus the S-box will substitute 101011 with the entry at row 4th (11) and column 6th (0101). To

Table 8.4. DES S-boxes

Row	Column																S-Box
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	S_1
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0	
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13	
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10	S_2
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15	
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9	
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8	S_3
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7	
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12	
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15	S_4
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4	
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14	
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9	S_5
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14	
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3	
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11	S_6
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6	
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13	
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1	S_7
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2	
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12	
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7	S_8
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8	
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11	

substitute a 48-bit word, DES uses eight S-boxes each of size $64 \times 4 = 256$ bits occupying a total of 2 Kbits memory as shown in Table 8.4

The 32-bit S-Box output undergoes through another permutation, which is called P-Box Permutation.

The P-Box Permutation

In this step, the input 32-bit (output of the S-box) is permuted to get the 32-bit output. The bit position for P-Box permutation is shown in Table 8.5.

As shown in Table 8.5, bit 21 moves to bit 4, bit 4 moves to bit 31 and so on. There is no repetition in bits and none of them is ignored.

Table 8.5. Permutation P

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

The 32-bit output after P-Box permutation is XORed with L_0. In the next iteration, we will have $L_2 = R_1$, which is the block we just calculated and then we must calculate R_2, repeating the same procedure as it was used for R_1. At the end of the 16^{th} iteration we have the blocks L_{16} and R_{16}. The order of these blocks is reversed and two blocks are concatenated into a 64-bit block $R_{16}L_{16}$. The final permutation IP^{-1} is then applied.

The Final Permutation IP^{-1}

Table 8.6 provides the bit positions for the final permutation which operates on 64-bit input block producing 64-bit output block. This completes the encryption process for a single block.

Table 8.6. Inverse Permutation

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Decryption is simply the inverse of encryption which is carried out by repeating the same steps as they were explained above. Only the round keys are applied in the reverse order.

8.3.3 Key Schedule

The round keys for all the 16 rounds are derived from the original key as shown in Figure 8.9. First the 64-bit DES key is reduced to 56 bits by ignoring every 8^{th} bit governed by the Table 8.7. This is referred to as Permuted Choice One (PC-1). The 48-bit round keys are then derived as follows.

Table 8.7. Permuted Choice one PC-1

57	49	41	33	25	17	9	1	58	50	42	34	26	18
10	2	59	51	43	35	27	19	11	3	60	52	44	36
63	55	47	39	31	23	15	7	62	54	46	38	30	22
14	6	61	53	45	37	29	21	13	5	28	20	12	4

The 56-bit output after PC-1 is divided into two halves C_0 and D_0. In each round, the two halves undergo a circular left shift or rotation by either one or two bits, depending on the round as shown in Table 8.8.

Table 8.8. Number of Key Bits Shifted per Round

Round No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

After the shift operation, the two halves are concatenated and serve as input to Permuted Choice Two (PC-2) as given in Table 8.9. The resulting 48-bit block is the required round key. Both halves before permutation PC-2 are also used as the two inputs to generate round keys for the next round as is shown in Figure 8.9.

Table 8.9. Permuted Choice two (PC-2)

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

8.4 FPGA Implementation of DES Algorithm

In this section DES implementation is described on reconfigurable hardware platform. The design steps for the development of FPGA architecture are explained along with some useful design techniques for the improvement of design performance. Performance results and comparison with the previous FPGA implementations of DES are presented at the end of this Section.

8.4.1 DES Implementation on FPGAs

Figure 8.10 is a block diagram representation of DES implementation in FPGAs. As it has been mentioned before, permutation operations do not occupy

logical resources of the device and it can be implemented by rearranging bit positions for the outgoing bus (change of wires), hence it is free of cost. DES includes several permutations (initial, final, permutation E, permutation P). The building blocks for those operations in Figure 8.10 are therefore symbolic representations having no logic inside.

Each DES S-Box occupies 64 × 4=256-bit memory. Hence, a total of 2K (2048 bits) memory is required for the construction of eight S-Boxes. If it is not intended to use dedicated memory resources, only 32 CLB slices are needed for an S-Box on Xilinx VirtexE devices. Some other fabric resources of the device were occupied for the implementation of latches (Slice Flip Flops) and logic blocks for XOR operation.

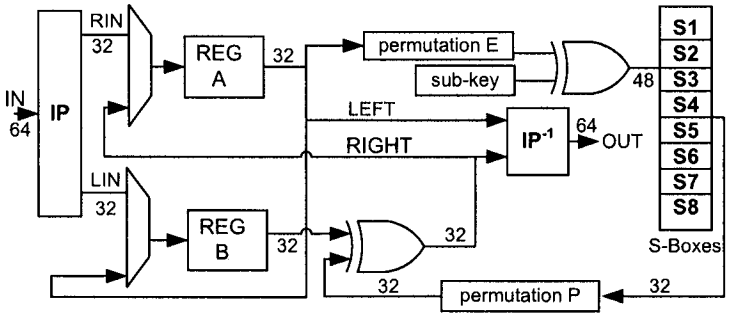

Fig. 8.10. DES Implementation on FPGA

DES chip consists of four I/O pins: three inputs and one output. The three input pins are Chip Enable (CE), Clock (CLK), and input data (IN). The single output pin is named as OUT. The CE signal activates the DES chip, whereas the CLK is the only master clock in charge of driving the whole circuitry. It is used to generate all control signals needed for the synchronization of the data flow.

When the CE signal is in low, it enables the circuit. As a consequence, the input 64-bit block after passing through initial permutation (bit wires rearranged) is partitioned into two halves RIN and LIN. At the first rising edge of CLK, both RIN and LIN are transferred to the output of the two registers REGA and REGB. The REGA output (RIN) goes through a number of operations: Permutation E, addition with sub-key, substitution (through S-Boxes), Permutation P, and then finally addition with the initial REGB output (LIN). On the next clock, the old right half (RIGHT) is the input of register REGB and the new left half (LEFT) is the input of register REGA.

In the 16^{th} clock cycle the two RIGHT and LEFT halves are concatenated (two buses joined together) and they are pass through the inverse permutation (IP^{-1}) producing a valid 64-bit DES ciphertext at OUT pin.

It is to be noted that the parallel structure for the eight DES S-Boxes contributes with a significant reduction of the critical path for encryption/decryption.

8.4.2 Design Testing and Verification

DES implementation was made on XCV400e-8-bg560 VirtexE device using Xilinx Foundation Series F4.1i. The design tool provides two options for design testing and verification: functional simulation and timing verification. Functional verification tests the logical correctness of the design. It is performed after the design entry has been completed using VHDL or using library components of the target devices. It detects logical errors without considering circuit overheads like path delays, synchronization, etc. A netlist of the logic components in the design is created by the design tool, which is then mapped to the available resources of the actual target device. Timing verifications are made at this stage.

Both functional and timing verifications must be performed for a successful design implementation. For both cases, test vectors are used for result verification and testing. Table 8.10 shows a simple test vector used to verify DES chip.

Table 8.10. Test Vectors

Input Block	First Permutation	f(R,K)	Second permutation
LIN=0×FFFF0000	LFOUT=0×06060606	LEFT=0×49DE9DF2	LOUT=0×17F77A33
RIN=0×AAAAAAAA	RFOUT=0×E7E7E7E7	RIGHT=0×C7EEC966	ROUT=0×7B7AB72A

Figure 8.11 and Figure 8.12 show the results for the functional simulation and the timing verification for DES implementation in FPGA. Notice that the difference between Figure 8.11 and 8.12. Time delays in Figure 8.12 are clearer.

8.4.3 Performance Results

FPGA implementation of DES algorithm was accomplished on a VirtexE device XCV400e-8-bg560 using Xilinx Foundation Series F4.1i as synthesis tool. The design was coded using VHDL language. It occupied 165 (3%) CLB slices, 117 (1%) slice Flip Flops and 129 (41%) I/Os. The design achieves a frequency of 68.05 MHz (14.7 ηS). It takes 16 clock cycles to encrypt one data block (64-bits). Therefore, the achieved throughput is $(68.05 \times 64)/16$=274 Mbits/s.

8.5 Other DES Designs

Several FPGA implementations of DES have been reported in the literature achieving throughput ranges from 26 Mbps to 21.3 Gbps. In Table 8.11 we

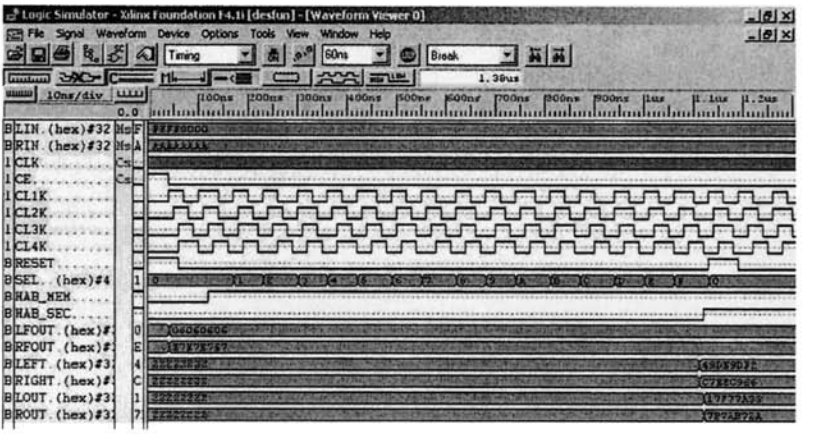

Fig. 8.11. Functional Simulation

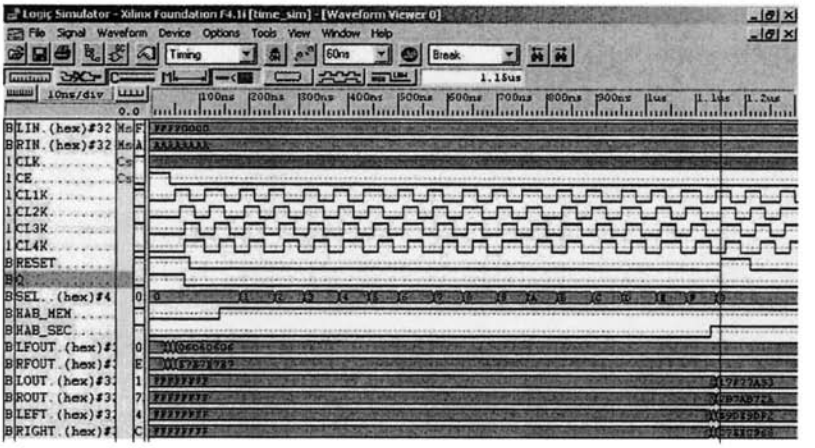

Fig. 8.12. Timing Verification

review the fastest designs reported in the literature. They are sorted in descending order. The design reported in [299] by Rouvrou *et al* achieves a throughput of 21.3 Gbps and it is the fastest design reported up to this book publication's date. It consist on a pipeline architecture with a pipeline depth of 37 stages. The 37 stages for that design were developed by introducing a different formulation of DES in which a new mathematical expression especially tailored for FPGA devices is proposed. In the same paper, authors proposed a different grouping of the stages resulting in a pipeline architecture of 21 stages. The throughput for the second architecture is reported as 14.5 Gbps.

Table 8.11. DES Comparison: Fastest Designs

Author	Device	Design Strategy	Area Slices	Freq. (Mhz)	Throughput (Mbps)	T/A
Rouvroy et al.[301]	XC2V1000-6	Pipeline 37 cycles	2965	333	21300	7.18
Xilinx [148]		Pip. 48 stages	3900	237	15100	3.87
Rouvroy et al.[301]	XC2V1000-6	Pipeline 21 cycles	3775 LUT 2904 FF	227	14500	N.D.
Trimberger et al.[363]	XCV300 E8	Pip. 48 stages (3DES)	4216 LUT 5573 FF	188.7	12000	N.D.
Patterson et al.[271]	XCV150-6	Jbits and RTR	1584	168	10752	6.75
Swankoski et al.[353]	Virtex-II Pro	17 parallel DES blocks	5544	140.6	9000	1.65
Trimberger et al.[363]	XCV300 E8	Pip. 16 stages	4216 LUT 1943 FF	132	8400	N.D.
McLoone et al.[225]	XCV1000	Pipeline 16 stages	6446	59.5	3808	0.59
FreeIP-Proy [99]	XCV400-6	Pipeline 16 stages	2528	47.7	3100	1.22

The first architecture is also the most efficient architecture with a throughput over area ratio of 7.18.

Trimberger *et al* [363, 148] presented three of the fastest DES designs ever reported. They are pipelined designs with 48 and 16 stages. A Java-based (Jbits) DES implementation in [271] achieves the encryption rate of 10752 Mbps. It implements all DES primitives in FPGA while the key schedule is processed in software. The communication between the two operations is made through a Java-based Application Programming Interface (API) which is intended for runtime creation and modification of the bit-stream.

Initial high-performance designs were reported by McLoone *et al* [225] and the free IP project [99]. Both are 16-stage pipeline architectures that report throughputs around 3 Gbps. The architecture reported by Swankoski *et al* in [353] consists of several independent DES blocks (17).

In Table 8.12 we review the most compact designs reported in the literature. They are sorted in ascending order. In general, a throughput greater than 1 Gbps is well beyond reach of compact designs which, otherwise, is not the main goal of such designs. On the contrary hardware area is the major concern for such type of architectures. Most of them implement one DES round and iteratively process a block to perform encryption/decryption.

Most recent reported designs [309], [300] and [353] implement both, ciphering and key scheduling. The most compact design was reported by Nazar *et al* [309] with a design that occupies just 167 slices. The next one, reported by Rouvroy *et al* [300], occupies 189 CLB slices attaining better performance results. Some other designs implement more than one round in order to increase performance [167]. FPGA implementation of DES in [167] implement

Table 8.12. DES Comparison: Compact Designs

Author	Device	Design Strategy	Area Slices	Freq. (Mhz)	Throughput (Mbps)	T/A
Nazar et al.[309]	XCV400E	one round	167	68.5	274	1.64
Rouvroy et al.[300]	XC2V1000-5	one round	189	274	974	5.15
CAST [147]	Virtex-II 5	one round	238	N.A.	816	3.43
Kaps et al.[167]	XCV4013 E3	one round	262	23.9	91.2	0.35
Swankoski et al.[353]	Virtex-II Pro	one round	343	203.3	765.7	2.23
Wong et al.[384]	XCV4020E	one round	438	10	26.7	0.06
Kaps et al.[167]	XCV4013 EX3	2 stage pipe	433	23.0	183.8	0.42
Leonard et al.[204]	XCV4025-4	key spec.	640	6.0	24	0.04
Kaps et al.[167]	XCV4013 EX3	4 stage pipe	741	25.2	402.7	0.54

both 2-stage and 4-stage pipeline approaches obtaining throughput of 183.8 Mbps and 402.7 Mbps, respectively. The design in [384] is a one round DES implementation on a single-chip FPGA. A fair comparison is not possible with this design and the one reported in [204], because they did not consider key scheduling.

In Table 8.13 we select those designs presented in Tables 8.11 and 8.12 showing a throughput over area ratio greater than one. In this sense, the most efficient designs are also high-performance designs.

Table 8.13. DES Comparison: Efficient Designs

Author	Device	Design Strategy	Area Slices	Freq. (Mhz)	Throughput (Mbps)	T/A
Rouvroy et al.[301]	XC2V1000-6	Pip. 37 cycles	2965	333	21300	7.18
Patterson et al.[271]	XCV150-6	Jbits and RTR	1584	168	10752	6.75
Rouvroy et al.[300]	XC2V1000-5	one round	189	274	974	5.15
Xilinx [148]		Pip. 48 stages	3900	237	15100	3.87
CAST [147]	Virtex-II 5	one round	238	N.A.	816	3.43
Swankoski et al.[353]	Virtex-II Pro	one round	343	203.3	765.7	2.23
Swankoski et al.[353]	Virtex-II Pro	17 parallel DES blocks	5544	140.6	9000	1.65
Nazar et al.[309]	XCV400E	one round	167	68.5	274	1.64
FreeIP-Proy [99]	XCV400-6	Pip. 16 stages	2528	47.7	3100	1.22

Finaly, in Table 8.14 we show some other designs for TripleDES. They are sorted in descending order with respect to performance. Pipeline strategy

Table 8.14. TripleDES Designs

Author	Device	Design Strategy	Area Slices	Freq. (Mhz)	Throughput (Mbps)	T/A
Panu et al.[131]	Virtex V800 FG 676-6	Pip. 3DES 16 round	6689	45.55	2912	0.43
Rouvroy et al.[300]	XC2V1000-5	It. 3DES	604	258	917	1.51
Swankoski et al.[353]	Virtex-II Pro	It. 3 Blocks 3DES Parallel	819	195.1	734.9	0.089
Panu et al.[131]	Virtex V800 FG 676-6	3DES two round	1257	25.09	59.44	0.04
Panu et al.[131]	Virtex V800 FG 676-6	3DES wireless app.	1107	43.90	55.12	0.04
Leitold et al.[202]	VLSI 0.6 μm CMOS	3DES CBC QoS apps.	N.A.	275	155	N.A

is applied by Panu *et al* in order to develop a TripleDES Core specifically targeting wireless communications. The design reported by Leitold *et al* [202] is not an FPGA design, but rather, it is a VLSI design specifically targeted to ATM communications.

8.6 Conclusions

This chapter provides a general guideline for the implementation of block ciphers in reconfigurable logic platform. The general structure of block ciphers was discussed. Most frequent operations in block ciphers were presented, and at the same time, several useful properties for implementing block ciphers in FPGAs were discussed. We described the design steps and some design techniques for obtaining fast and/or compact and/or efficient FPGA implementations. A general guideline, was therefore developed for the implementation of block ciphers in reconfigurable devices. Our methodology was then applied for DES implementation resulting on an efficient and compact DES core on reconfigurable hardware platform.

We also showed a very compact DES architecture which can be adequate for embedded and mobile systems. We presented a review of several DES designs reported in the technical literature. We walked through high-performance designs to compact ones. We also reviewed efficient DES designs as well as several TripleDES designs, which were classified according to their performance metrics.

9

Architectural Designs For the Advanced Encryption Standard

In this chapter we present some of the most common architectural alternatives to implement Advanced Encryption Standard (AES) in reconfigurable hardware. The first factor to be considered on implementing AES is the application. There are high speed applications like High Definition TV (HDTV) and video conferencing where high performance is required. The target throughput, expressed in gigabits per second (Gbps), must be specified, and to achieve such a high performance we can replicate several functional units to increase parallelism. That would however imply higher power and hardware area requirements.

On the other hand, high speed designs are not always desired solutions. In some applications, such as mobile computing and wireless communications, smaller throughput is demanded. Then a good balance between hardware area and design performance should be achieved. In addition, since there are trends to incorporate secure electronic data exchange into low-end consumer products, inexpensive AES implementations are needed for PDAs (personal digital assistant), wireless devices and many other embedded applications. Furthermore, it has been suggested that applications in the domain of radio frequency identifiers (RFID), low-power AES chip may be needed, thus demanding extremely compact AES implementations.

9.1 Introduction

Two main factors impact an AES implementation for a given application: hardware area and timing performance. Quite frequently, both factors have an opposite effect: Although compact designs tend to occupy a small amount of hardware resources, they generally show low performances. On the contrary, achieving high speed gains requires that many modules should work simultaneously, thus demanding greater hardware area requirements.

Another important feature to be considered when choosing an architectural alternative for AES is related to its mode of use. Many applications use AES in the Electronic Code Book (ECB) mode in which a complete block is ciphered independently of all other blocks. Then, several blocks can be processed in parallel or pipeline strategies can be applied to increase performance. Nevertheless, it is noticed that ciphering is only a part of a secure application and that there exist applications for which ciphering is accomplished with authentication [214]. For those scenarios, a feedback mode is required. For example, in Cipher Block Chaining (CBC), a previous ciphered block is used to encrypt the present block. That however, prevents us from using pipeline architectures. Therefore, an iterative architecture with some authentication logic could be a solution.

From its evaluation process to post selection period, the Advanced Encryption Standard (AES) has been implemented on all kind of hardware and software platforms. Gladman [109] and Bertoni et al. [21], reported software implementations in which AES specification is manipulated to increase performance. AES software implementations have a throughput that ranges from 300 to 800 Mbps depending on the specific architecture and platform selected by the developers. Some efficient AES encryptor/decryptor core VLSI implementations have been also reported in [143, 376, 215, 303]. Performance of VLSI implementations ranges from 2 to 7.5 Gbps.

Similarly, various AES FPGA implementations have been reported in [102, 63, 83, 223]. Those are one round (*iterative*) or n rounds (*pipeline*) FPGA implementations optimized for encryption or encryption/decryption processes. Since published works have utilized an ample variety of FPGA devices, reported performance results are broadly variable ranging from 300 Mbps to up to 25 Gbps.

Clearly, modern FPGA technology has a great impact in implementation performances. Nonetheless, there are algorithmic and architectural strategies for different target applications that also influence the final performance. The asymmetric characteristics of AES encryption and decryption processes limit the implementation of high-performance AES cores. Each step for AES encryption has its inverse counterpart for decryption. Designing separated architectures for encryption and decryption processes would imply the allocation of a large amount of FPGA resources and the area requirements of such design might be difficult or even impossible to meet in several FPGA families of devices.

Published work about AES FPGA implementation covers a wide spectrum. Some designs [102, 63, 83] have considered only the encryption part of AES. For example, in [102, 63] an iterative design implementing one round is reported. In [63] key scheduling is also considered, however, in [102] key scheduling was ignored. The design in [83] implements all AES rounds with a pipeline organization but without key scheduling, whereas the design in [223] reported an FPGA implementation of a fully pipeline AES encryptor/decryptor core.

In this Chapter, various FPGA architectures of AES are presented. Those implementations cover all three basic processes: key scheduling, encryption and decryption. All are single-chip FPGA implementation. Different design architectures are considered by implementing AES encryptor, decryptor and encryptor/decryptor cores separately. Both iterative and pipeline techniques are applied showing diverse time-area tradeoffs. All AES implementations were optimized for low cost, high efficiency and/or high portability.

The rest of this Chapter is organized as follows. An introduction to AES algorithm is presented in Section 9.2. The basic transformations of the algorithm and their effects on the algorithm cryptographic strength are also explained in this Section. Section 9.3 gives a brief explaination of the AES modes of use. Section 9.4 describes various algorithmic optimization for implementing AES basic transformations on FPGAs. Those techniques help to improve overall algorithm performance by modifying the most costly operations of the algorithm. Section 9.5 deals with general architectures for AES implementation on FPGAs. Then, the algorithmic optimizations are mixed with architectural alternatives to obtain several different AES designs. Section 9.6 presents performance results for each design and compare them with published works. Finally, in Section 9.6.1 some recent trends on AES cores are reviewed providing a classification of several relevant designs. Concluding remarks are drawn in Section 9.7.

9.2 The Rijndael Algorithm

On October 2000, Rijndael was selected as a new Advanced Encryption Standard (AES) by NIST [253] replacing Data Encryption Standard (DES). The name 'Rijndael' is a rearrangement of the names of its two inventors Rijmen and Daemen [60].

Rijndael is a symmetric block cipher which takes two inputs, namely, the plaintext block to be encrypted and the secret key. It applies an iterative procedure at the end of which an output ciphertext block is produced. During a single iteration, a set of transformations, called a *round*, are applied to the state data block. For each round, a *round key* is generated through a process called key scheduling.

In this Section we give a short explanation of the algorithm behavior. We start explaining the difference between AES and Rijndael. Then, we describe AES basic structure and building blocks. Thereafter, the round transformation of the algorithm is specified. Finally, the process of key generation is described.

9.2.1 Difference Between AES and Rijndael

AES fixes the block sizes and key lengths from the range supported by Rijndael. Rijndael can process variable block and key lengths of 128, 192, and 256 bits. Moreover, Rijndael supports all possible combinations of those sizes for

block and key lengths. The number of rounds depends upon the combination of the selected block and key lengths as shown in Table 9.1. It can be seen that the number of rounds ranges from 10 to 14.

key length *(bits)*	Block length *(bits)*		
	128	192	256
128	10	12	14
192	12	12	14
256	14	14	14

Table 9.1. Selection of Rijndael Rounds

On the other hand, AES fixes the block length to 128 bits and supports key lengths of 128, 192 or 256 bits only. The most frequent AES case of use is with block and key lengths of 128 bits. In the rest of this chapter whenever we use the word AES, it means block and key lengths of 128 bits and therefore with the number of rounds equal to 10. Moreover, In the rest of this Chapter the names AES and Rijndael are used indistinctly.

9.2.2 Structure of the AES Algorithm

The basic structure of AES algorithm is shown in Figure 9.1.

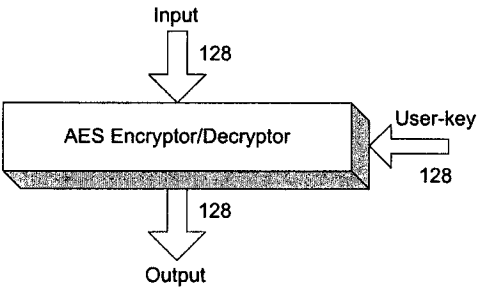

Fig. 9.1. Basic Structure of Rijndael Algorithm

For encryption, the input is a plaintext block and a key, and the output is a ciphertext block. For decryption, the input is a ciphertext block and a key (the same key used for encryption), and the output is the original plaintext. The basic algorithm flow for encrypting a single block of data is shown in Figure 9.2.

The AES cipher treats the input 128 bit block as a group of 16 bytes organized in a 4×4 matrix called *State* matrix. The algorithm consists of an initial

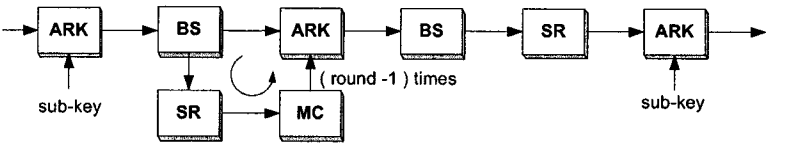

Fig. 9.2. Basic Algorithm Flow

transformation, followed by a main loop where nine iterations, called *rounds*, are executed. Each *round transformation* is composed of a sequence of four transformations: ByteSubstitution (BS), ShiftRows (SR), MixColumns (MC) and AddRoundKey (ARK). For each round of the main loop, a round key is derived from the original key through a process called *Key Scheduling*. At the last round MC step is skipped and consequently just three transformations, namely, BS, SR and ARK, are executed.

AES decryption can be performed by using same algorithm flow. However all four steps in the round transformation are replaced with their own inverses and the round keys for encryptions are used in the reverse order.

9.2.3 The Round Transformation

The round transformation is a sequence of four transformations BS, SR, MC and ARK. All four transformations contribute in AES strength by inducing *confusion* and *diffusion*, which are arguably the two most important properties that a strong symmetric cipher must have. Confusion makes the output dependent on the key. Ideally, every key bit influences every output bit. Diffusion makes the output dependent on previous input (plain/ciphertext). Ideally, each output bit is influenced by every (previous) input bit. Roughly speaking, those characteristics correspond to cipher's substitution and permutation.

Symmetric ciphers need to be complex, so they could not be analyzed easily. Also, their transformations need to be simple enough to be implemented efficiently in hardware or software. For AES, the general criteria for round transformation was inverse function and simplicity besides the step-specific criteria.

9.2.4 ByteSubstitution (BS)

It is a non-linear transformation where each input byte of the State matrix is independently replaced by another byte. BS can be seen as a highly non-linear function. There are a great finite number of possible BS functions, however some of them are more appropriate than others. In [60] some important properties about designing a BS function are discussed. *Non-linearity* and *algebraic complexity* being the most important of them.

The BS transformation of an input byte (8-bit vector) a is defined by two substeps:

1. **Inverse:** Let $x = a^{-1}$, the multiplicative inverse in GF(2^8) (except if $a = 0$ then $x = 0$).
2. **Affine Transformation:** Then the output is $y = M \times x \oplus b$, with the constant bit matrix M and byte b shown below:

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1\,1\,1\,1\,1\,0\,0\,0 \\ 0\,1\,1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,0\,1\,1\,1\,1\,1 \\ 1\,0\,0\,0\,1\,1\,1\,1 \\ 1\,1\,0\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,0\,0\,1\,1 \\ 1\,1\,1\,1\,0\,0\,0\,1 \end{bmatrix} \times \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{9.1}$$

All bit operations are performed modulo 2.

BS is decomposed into two transformations. First each input byte is replaced with its multiplicative inverse (MI) in GF(2^8) with the element {00} being mapped to itself and then the affine transformation is applied as shown in Equation 9.1.

From the implementation point of view, BS can be considered as a look-up table, called *S-Box*, in which the input byte is considered as the address of the table where its substitution is found. Then an S-Box can be seen as a 256×8 look up table as shown in Figure 9.3. This is the easiest way to implement BS and for many applications it is enough to consider this way of implementing it[1].

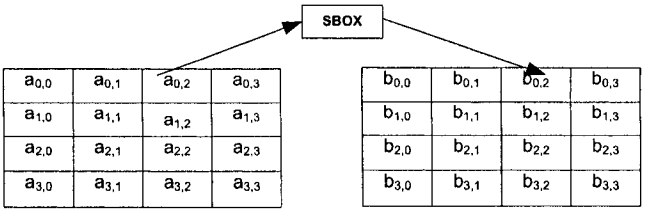

Fig. 9.3. BS Operates at Each Individual Byte of the State Matrix

If we look for a very compact or a high efficient design, we need to look for the calculation of BS. Multiplicative inverse can be found using the extended Euclidean algorithm [228][2]. Let x be the input byte and let us assume that we

[1] It has been proposed that also the multiplications associated to the MixColumn transformation can be implemented using the Look-up Table methodology [81].

[2] Formal definition of field multiplicative inverse and the extended Euclidean algorithm can be found in §4.1.2. Efficient computations of the multiplicative inverse were discussed in §6.3.

look for the inverse of the polynomial $a(x)$. The extended Euclidean algorithm can be used to find two polynomials $b(x)$ and $c(x)$ such that:

$$a(x) \times b(x) + m(x) \times c(x) = \gcd(a(x), m(x)) \tag{9.2}$$

where $\gcd(a(x), m(x))$ represents the *greatest common divisor* of the polynomials $a(x)$ and $m(x)$. If $m(x)$ is irreducible then we know for sure that $\gcd(a(x), m(x)) = 1$. Applying modular reduction to Equation 9.2 we get,

$$a(x) \times b(x) = 1 \bmod m(x) \tag{9.3}$$

which means that $b(x)$ is the inverse element of $a(x)$. The non-linearity of the AES S-box is introduced by applying the multiplicative inverse in $GF(2^8)$. The affine transformation has no impact on the non-linearity but it contributes in increasing the algebraic complexity.

Inverse Operation (IBS)

The inverse BS is obtained by applying inverse affine transformations followed by the multiplicative inverse in $GF(2^8)$. Therefore, the inverse of the affine transformation in Eqn. 9.1 is defined as follows.

$$x = M^{-1} \times y \oplus d$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,1\,0\,0\,1 \\ 1\,0\,0\,1\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1\,0\,1 \\ 1\,0\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,1\,0\,0\,1 \\ 1\,0\,1\,0\,0\,1\,0\,0 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{9.4}$$

For both affine and inverse affine transformations, multiplicative inverse is taken in $GF(2^8)$ with irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

9.2.5 ShiftRows (SR)

It is a cyclic shift operation where each row is rotated cyclically to the left using 0,1,2 and 3-byte offset for encryption as shown in Figure 9.4. *Diffusion optimality* is the design criteria for selecting the offsets which requires the four offsets to be different.

Inverse Operation (ISR)

The inverse operation of ShiftRows is called Inverse ShiftRows (ISR). It is a cyclic shift operation used for decryption where each row is rotated cyclically to the right using 0,1,2 and 3-byte offset.

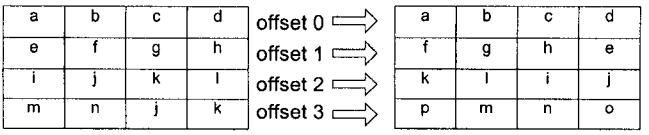

Fig. 9.4. ShiftRows Operates at Rows of the State Matrix

9.2.6 MixColumns (MC)

In this transformation, each column of the State matrix is considered a polynomial over GF(2^8) and is multiplied by a fixed polynomial $c(x)$ modulo $x^4 + 1$. The polynomial $c(x)$ is given by:

$$c(x) = 03.x^3 + 01.x^2 + 01.x + 02 \tag{9.5}$$

Let $b(x) = c(x) \cdot a(x) \bmod x^4 + 1$, then the modular multiplication with a fixed polynomial can be written as shown in Equation 9.6.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{9.6}$$

MixColumns operates on the columns of the state matrix as shown in Figure 9.5.

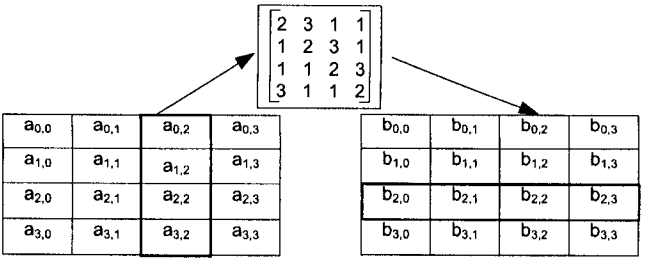

Fig. 9.5. MixColumns Operates at Columns of the State Matrix

The design criteria for MixColumns step includes *dimensions*, *linearity*, *diffusion* and *performance* on 8-bit processor platforme. The *Dimension* criterion it is achieved in the transformation operation on 4-byte columns.

Inverse Operation IMC

The inverse of MixColumns is called (IMC). The constant polynomial $c(x)$ given in Eqn. 9.5 is co-prime to $x^4 + 1$ and therefore invertible. Let $d(x)$ be the inverse of $c(x)$ and written as follows.

$$(03.x^3 + 01.x^2 + 01.x + 02).d(x) \equiv 01 \ (mod\ x^4 + 1) \tag{9.7}$$

From Eqn. 9.7, it can be seen that $d(x)$ is given by:

$$d(x) = 0B.x^3 + 0D.x^2 + 09.x + 0E \tag{9.8}$$

Similarly to MC, in IMC each column of the state matrix is transformed by multiplying with constant polynomial $d(x)$ written as a matrix multiplication as shown in Equation 9.9.

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{9.9}$$

9.2.7 AddRoundKey (ARK)

In the last step, the output of MC is XOR-ed with the corresponding round key. This step is denoted as ARK. Figure 9.6 illustrates the effect of key addition on the state matrix.

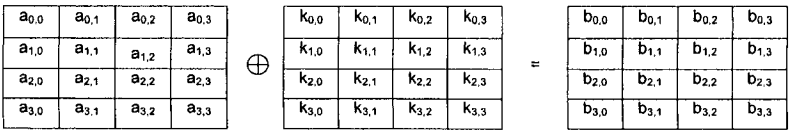

Fig. 9.6. ARK Operates at Bits of the State Matrix

Inverse Operation IARK

Inverse of ARK, called IARK, is essentially the same for encryption and decryption[3]. The only important thing to remember is that keys are applied for decryption in reverse order as in encryption.

[3] However, as is explained in §9.5.2, efficient implementations of AES encryptor/decryptor cores, require to append the IMC step to the generation of round keys for decryption.

9.2.8 Key Schedule

Both, encryption and decryption require the generation of round keys. Round keys are obtained through the expansion of secret user key by attaching each $j-th$ round a 4-byte word $k_j = (k_{0,j}, k_{1,j}, k_{2,j}, k_{3,j})$ to the user key. The original user key, consisting of 128 bits, is arranged as a 4×4 matrix of bytes.

Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key. Then, these four columns are recursively expanded to obtain 40 more columns. Let us assume we have computed columns up to $w[i-1]$. Then, we can compute the $i-th$ column, $W[i]$, as follows,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \tag{9.10}$$

where $T(w[i-1])$ is a non-linear transformation of $w[i-1]$ calculated as follows:

Let w, x, y, and z be the elements of column $w[i-1]$ then,

1. Shift cyclically the elements to obtain z, w, x, and y.
2. Replace each of the byte with the byte from BS $S(z)$, $S(w)$, $S(x)$ and $S(y)$.
3. Compute the round constant $r(i) = 02^{(i-4)/4}$ in GF(2^8).

Then, T($w[i-1]$) is the column vector, $(S(z) \oplus r(i), S(w), S(x), S(y))$. In this way, columns from w[4] to w[43] are generated from the first four columns.

The 16-byte round key for the $j-th$ round consists of the columns

$$(w[4j], w[4j+1], w[4j+2], w[4j+3])$$

Sometimes it results convenient to pre-compute the round keys once and for all and then store them. A similar process is utilized for generating round keys for the decryption process, although they should be used in the reverse order.

After the explanation of all four AES transformations and key schedule, we can write the sequence of those transformations when performing encryption and decryption as follows,

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: IARK→ IMC→ ISR→ IAF→ MI

9.3 AES in Different Modes

Most of the published work on AES implementation considers AES in Electronic Book Mode (ECB). In ECB mode, an individual plaintext block is converted to ciphertext block. Thus by collecting several plaintext and their ciphertext blocks, one can produce some pattern information which could

be helpful in recovering the original plaintext. ECB mode in some cases, is therefore not considered secure. The Cipher Block Chaining mode (CBC), the Cipher Feedback mode (CFB), and the Output Feedback mode (OFB) offer better security than ECB, but encryption of the block depends on the feedback of its previous block encipherment [253]. This property prevents using pipelining in which many different blocks are encrypted simultaneously. The encryption speed in CBC, CFB, and OFB modes is much slower as in ECB. Fortunately, there exists another mode, called Counter mode (CTR) which increases the security of ECB and has not dependencies among different blocks, thus allowing all operations to be fully pipelined to achieve high performance.

9.3.1 CTR Mode

In [100] a CTR mode implementation of AES is reported. In CTR mode, a plaintext is processed by encrypting a counter value with key 'K' and then by XORing the output with the plaintext to get the ciphertext. Figure 9.7 presents the counter mode. Decryption procedure takes the same process to recover the plaintext from the ciphertext. The counter value has no dependencies with previous output, thus pipelining can be fully used. Counter mode has no padding overhead which is required for ECB, CBC, and CFB modes when the data is not a multiple of block length. Counter mode does not propagates error and restrict the error to the specific block as compared to CBC and CFB modes which pass the error to the subsequent blocks.

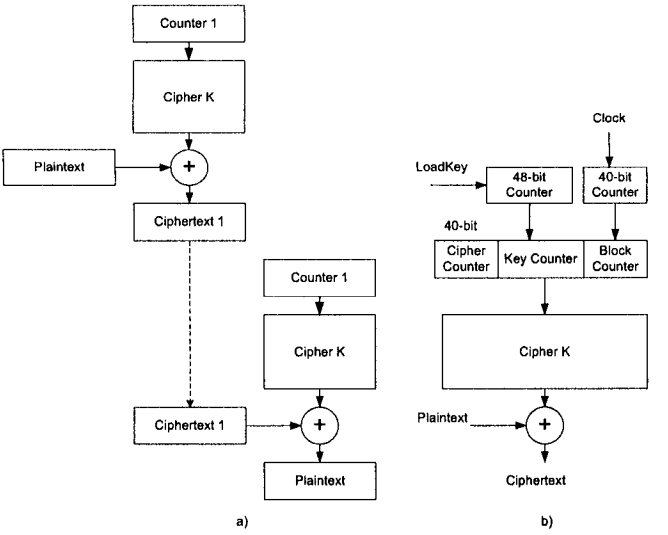

Fig. 9.7. Counter Mode Operations

Figure 9.7b, presents different counter blocks for obtaining cipher key 'K'. A three stage counter, 40-bit cipher identification, 48-bit key counter and 40-bit block counter, are used for each plaintext block. For each cipher artifact, there is a pre-assigned cipher ID. The key counter increases whenever a new key has been updated. Block counter increases for each block. The search space for each part is, although finite, large enough. If the block counter is exhausted, the key counter will be increased to avoid the use of the same key with the same counter value. Then, we guarantee that produced keys are all distinct. The counter value pairs can be used more than once.

The special requirement for CTR mode is that the same counter value and key should not be used to encrypt more than one block of data. If this happens, the plaintext would be recovered by XORing the two ciphertext, which in fact, equals to XORing the two plaintext. Especially when one of the plaintext is already known, the other one can be easily recovered by XORing the known plaintext with the output ciphertext after XOR.

9.3.2 CCM Mode

For applications in which more robustness is required, there is no choice and a feedback mode is mandatory. For example, the Wired Equivalent Privacy (WEP) protocol has been the most widely security tool used for protecting information in wireless environments. However, this protocol was broken in 2001 by Fluhrer et al. [1]. Based on that attack, nowadays there exist a variety of programs that can be downloaded from Internet to break the WEP Protocol in few seconds and with almost no effort. This situation has led to a search for new security mechanisms for guaranteeing reliable ways of protecting information in wireless mobile environments.

AES in CCM (Counter with CBC-MAC) proposed by Whiting et. al. in [378], has become one of the most promising solutions for achieving security in wireless networks. This mode simultaneously offers two key security services, namely, data Authentication and Encryption [214]. CCM means that two different modes are combined into one, namely, the CTR mode and the CBC-MAC. CCM is a generic authenticate-and-encrypt block cipher scheme that has been specifically designed for being use in combination with a 128-bit block cipher, such as AES. Currently, CCM mode has become part of the new 802.11i IEEE standard.

CCM Primitives

Before sending a message, a sender must provide the following information [378]:

1. A suitable encryption key K for the block cipher to be used.
2. A nonce N of $15 - L$ bytes. Nonce value must be unique, meaning that the set of nonce values used with any given key shall not contain duplicate values.

3. The message m, consisting of a string of $l(m)$ bytes where $0 \leq l(m) < 2^{8L}$.
4. Additional authenticated data a, consisting of a string of $l(a)$ bytes where $0 \leq l(a) < 2^{64}$. This additional data is authenticated but not encrypted, and is not included in the output of this mode.

Figure 9.8 shows CCM authentication and verification processes dataflow. Notice that because of the CBC feedback nature of the CCM mode a pipeline approach for implementing AES is not possible, therefore there is no option but to implement AES encryption core in an iterative fashion.

CCM Authentication consists on defining a sequence of blocks $B_0, B_1, \cdots, B_n$ and thereafter CBC-MAC is applied to those blocks so that the authentication field T can be obtained. Blocks B_is are defined as explained below.

First, the authentication data a is formatted by concatenating the string that encodes $l(a)$ with a itself, followed by organizing the resulting string in chunks of 16-byte blocks. The blocks constructed in this way are appended to the first configuration block B_0 [375]. Then, message blocks are added right after the (optional) authentication blocks a. Message blocks are formatted by splitting the message m into 16-byte blocks which will be the main part of the sequence of blocks

$$B_0, B_1, ..., B_n$$

needed by the authentication mode. Finally, the CBC-MAC is computed as,

$$\begin{aligned} X_1 &:= AES_E(K, B_0) \\ X_{i+1} &:= AES_E(K, X_i \oplus B_i) \text{ for } i = 1, ..., n \\ T &:= firstMbytes(X_{n+1}) \end{aligned} \tag{9.11}$$

Where AES_E is the AES block cipher selected for encryption, and T is the MAC value defined as above. If it is needed, the ciphertext would be truncated in order to obtain T.

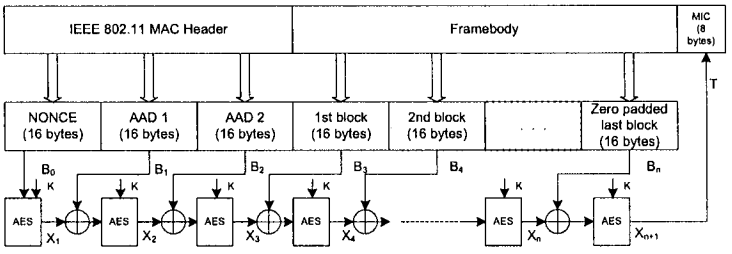

Fig. 9.8. Authentication and Verification Process for the CCM Mode

Figure 9.9 shows the CCM encryption/decryption process dataflow. CCM encryption is achieved by means of Counter (CTR) mode as,

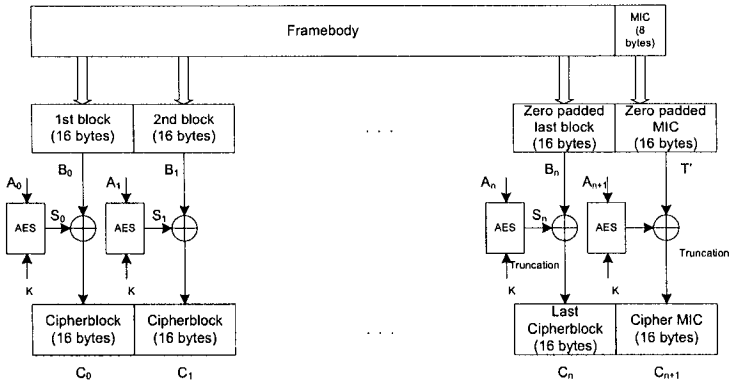

Fig. 9.9. Encryption and Decryption Processes for the CCM Mode

$$S_i := AES_E(K, A_i) \text{ for } i = 0, 1, 2, ..., \tag{9.12}$$
$$C_i := S_i \oplus B_i$$

where A_i stands for counters. See [378, 100] for more technical details about how to build the counters.

Plaintext m is encrypted by XORing each of its bytes with the first $l(m)$ bytes of the sequence resulting from concatenating the cipher blocks $S_1, S_2, S_3, ...$, produced by Eq. 9.12. The authentication value is computed by encrypting T with the key stream block S_0 truncated to the desired length as,

$$U := T \oplus firstMbytes(S_0) \tag{9.13}$$

The final result c consists of the encrypted message m, followed by the encrypted authentication value U.

At the receiver side, the decryption process starts by recomputing the key stream to recover the message m and the MAC value T. Figure 9.9 shows how the decryption process is accomplished in CCM Mode.

Message and additional authentication data is then used to recompute the CBC-MAC value and check T. If the T value is not correct, the receiver should not reveal the decrypted message, the value T, or any other information. Figure 9.8 describes how the verification process is accomplished.

It is important to notice that the AES encryption process is used in encryption as well as in decryption. Therefore, AES decryption functionality is not necessary in CCM-mode, which leads to save valuable hardware resources.

9.4 Implementing AES Round Basic Transformations on FPGAs

Strategies for efficient hardware implementation of AES on FPGA devices can be classified into two types: algorithmic and architectural optimizations. Algorithmic optimizations try to obtain some mathematical expressions to take advantage of FPGA structure. Architectural optimizations exploit design techniques such as iterative, pipelining and sub-pipelining. In addition, AES hardware implementation poses a challenge since encryption and decryption processes are not completely symmetrical which forces to have some additional observations while implementing a single encryptor/decryptor core.

In Subsection 9.2.3 it was described the basic round transformations, BS, SR, MC, and ARK, and their corresponding inverse transformations IBS, ISR, IMC, and IARK. That Subsection also describes the key schedule process to generate the necessary subkeys during an encryption or decryption process.

But before start discussing how to implement a full encryption or decryption core, let us analyze, from the algorithmic optimization point of view, some important implementation properties shown by the basic round transformations.

The most important operations for the basic transformations include polynomial multiplication in $GF(2^8)$ for BS/IBS, fixed-rotation for SR/ISR, constant polynomial multiplication in $GF(2^8)$ for MC/IMC, and simple addition (XOR) for ARK/IARK. Fixed-rotation is hardwired and does not consume FPGA's logic resources. The addition used in ARK/IARK is a simple XOR operation. Hence, BS/IBS and MC/IMC are the two key functional units in AES implementations. It has been estimated that BS/IBS and MC/IMC take more than 65% of the total area in the entire AES encryptor/decryptor implementation.

Perhaps, the most costly operation for BS/IBS is polynomial multiplication in $GF(2^8)$. We also need to perform a polynomial multiplication in $GF(2^8)$ for MC/IMC but we can take advantage from the fact that is a constant multiplication. Even though the latter transformation is relatively less costly than the former still it occupies considerable FPGA's resources. Therefore, both BS/IBS and MC/IMC are good candidates for improving overall performance of the round transformation.

In the rest of this Section, we present various approaches for implementing BS/IBS and MC/IMC.

Regarding BS/IBS two alternatives are considered. In the first approach pre-computed values are simply stored on the FPGA's built-in memory modules. This might be seen as an expensive solution but it helps to save valuable computational time. The second approach provides an alternative for constrained memory requirements and it is based on an on-fly computation strategy.

Similarly, two approaches for MC/IMC implementations are presented. First approach, that we have called *standard approach*, deals with the struc-

tural organization of MC/IMC transformations. The second approach called *modified approach* introduces a small modification before MC to perform IMC step. Finally, some structural changes are proposed in key schedule algorithm which can improve hardware performance by cutting path delays.

9.4.1 S-Box/Inverse S-Box Implementations on FPGAs

The straightforward approach for implementing BS is by using a look-up table in which pre-computed values are stored in memories. That requires memory modules with fast access. In FPGAs, there are two ways to organize memory: by using flip-flops and CLBs (i.e., FPGA fabrics), or by using FPGAs built-in memory modules called BRAMs (BlockRAMs).

Implementing BS/IBS by look-up tables is simple, fast and in many cases desirable. A single BS/IBS table would require 8-bit wide 256 entries. We can make some few observations about implementing BS/IBS using look-up tables.

Firstly, for the implementation of both encryption and decryption on a single chip two different separated look-up tables are required, thus duplicating memory requirements.

Secondly, if we want to increase performance, BS/IBS can be performed in parallel for the sixteen bytes of the state matrix. The fully parallelization of BS/IBS would therefore require 16 copies of the same look-up table, one per state matrix element. Finally, if high performance is required, unfolding the 10 rounds of AES to construct a pipeline architecture, would require 160 copies of the same look-up table.

In the following, we discuss some other alternatives to implement BS/IBS in FPGAs.

I. S-Box and Inverse S-Box Implementation

To avoid utilization of a considerable amount of FPGA resources, BS/IBS can be implemented using a look-up table. The look up table would be used for MI by implementation affine (AF) and inverse affine (IAF) transformations using some logic gates for BS and IBS respectively. The combination MI + AF implements BS for encryption and the combination IAF + MI gives IBS for decryption. For constructing an encryptor/decryptor core, two separated designs for encryption and decryption would result in high area requirements. From Section 9.2.4, we know that only one MI transformation in addition to AF and IAF transformations is required for both encryption and decryption. Therefore, a multiplexer can be used to switch the data path for either encryption or decryption as shown in Figure 9.10

II. S-Box and Inverse S-Box Based on Composite Field Techniques

BS/IBS implementations can be made using composite field techniques e.g. BS can be manipulated in $GF((2^4)^2)$ and even $GF(((2^2)^2)^2)$ instead of $GF(2^8)$.

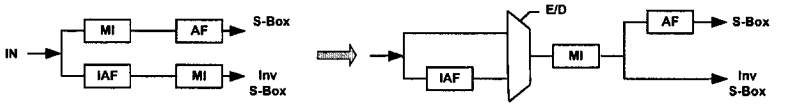

Fig. 9.10. S-Box and Inv. S-Box Using Same Look-Up Table

That would reduce memory requirements to 16×4 bits in $\mathrm{GF}(2^4)$ as compared to 256×8 bits in $\mathrm{GF}(2^8)$ for a single LUT. More hardware resources would be however used to implement the required logic in $\mathrm{GF}(2^4)$. Several authors [267, 242, 303] have designed AES S-Box based on the composite field techniques reported first in [267]. Those techniques use a three-stage strategy:

1. Map the element $A \in \mathrm{GF}(2^8)$ to a smaller composite field F by using an isomorphism function δ.
2. Compute the multiplicative inverse over the field F.
3. Finally, map the computations back to the original field.

In [242], an efficient method to compute the inverse multiplicative based on Fermat's little theorem was outlined. That method is useful because it allows us to compute the multiplicative inverse over a composite filed $\mathrm{GF}(2^m)^n$ as a combination of operations over the ground field $\mathrm{GF}(2^m)$. It is based on the following theorem:

Theorem 1 *[267, 121] The multiplicative inverse of an element A of the composite field $GF(2^m)^n$, $A \neq 0$, can be computed by,*

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma - 1} \bmod P(x) \tag{9.14}$$

$$\texttt{where} \qquad A^{\gamma} \in \mathrm{GF}(2^n) \;\&\; \gamma = \frac{2^{nm} - 1}{2^m - 1}$$

An important observation of the above theorem is that the element A^{γ} belongs to the ground field $\mathrm{GF}(2^m)$. This remarkable characteristic can be exploited to obtain an efficient implementation of the inverse multiplicative over the composite field. By selecting $m = 4$ and $n = 2$ in the above theorem, we obtain $\gamma = 17$ and,

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma - 1} = (A^{17})^{-1} A^{16} \tag{9.15}$$

In case of AES, it is possible to construct a suitable composite field F, by using two degree-two extensions based on the following irreducible polynomials.

$$\begin{array}{ll} F_1 = \mathrm{GF}(2^2) & P_0(x) = x^2 + x + 1 \\ F_2 = \mathrm{GF}((2^2)^2 & P_1(y) = y^2 + y + \phi \\ F_3 = \mathrm{GF}(((2^2)^2)^2 & P_2(z) = z^2 + z + \lambda \end{array} \tag{9.16}$$

$$\text{where } \phi = \{10\}_2,\ \lambda = \{1100\}_2$$

The inverse multiplicative over the composite field F_2 defined in the Equation 9.15, can be found as follows.

Let $A \in F_2 = \text{GF}(2^2)^2$ be defined in polynomial basis as $A = A_H y + A_L$, and let the Galois Fields F_1, F_2, and F_3 be defined as shown in Equation 9.16, then it can be shown that,

$$\begin{aligned} A^{16} &= A_H y + (A_H + A_L) \\ A^{17} &= A^{16} \cdot A = 0.y + (\lambda(A_H)^{16} A_H + (A_L)^{16} A_L) \\ &= \lambda(A_H)^2 + (A_L)^{16} A_L \end{aligned} \tag{9.17}$$

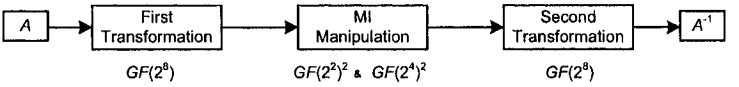

Fig. 9.11. Block Diagram for 3-Stage MI Manipulation

Figures 9.11 and 9.12 depict block diagram to three-stage inverse multiplier represented by Equations 9.15 and 9.17.

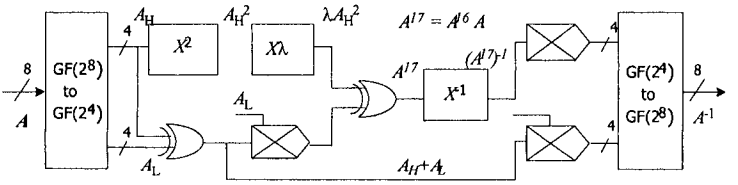

Fig. 9.12. Three-Stage Approach to Compute Multiplicative Inverse in Composite Fields

As it was explained before, in order to obtain the multiplicative inverse of the element $A \in F$ =GF(2^8), we first map A to its equivalent representation (A_H, A_L) in the isomorphic field $F_2 = \text{GF}\left((2^2)^2\right)$ using the isomorphism δ (and its corresponding inverse δ^{-1}). In order to map a given element A from the finite field F to its isomorphic composite field F_2 and vice versa, we only need to compute the matrix multiplication of A, by the isomorphic functions shown in Equation 9.18 given by [242]:

$$\delta = \begin{bmatrix} 1\,0\,1\,0\,0\,0\,0\,0 \\ 1\,1\,0\,1\,1\,1\,1\,0 \\ 1\,0\,1\,0\,1\,1\,0\,0 \\ 1\,0\,1\,0\,1\,1\,1\,0 \\ 1\,1\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,1\,1\,1\,0 \\ 0\,1\,0\,1\,0\,0\,1\,0 \\ 0\,1\,0\,0\,0\,0\,1\,1 \end{bmatrix} \quad \delta^{-1} = \begin{bmatrix} 1\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,0\,0\,0\,1\,0\,0 \\ 0\,1\,1\,0\,0\,0\,1\,0 \\ 0\,1\,1\,1\,0\,1\,1\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0 \\ 0\,0\,1\,1\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1\,1 \\ 0\,1\,1\,1\,0\,1\,0\,1 \end{bmatrix} \tag{9.18}$$

The isomorphism function δ and δ^{-1} can be constructed as follows:

Let α and β be roots of a same primitive irreducible polynomial ($m(x) = x^8 + x^4 + x^3 + x^2 + 1$ can be used). First search for primitive element α in the field A and then search for β in the field B. Once δ and δ^{-1} are founded, the matrix representation can be obtained, where α^k is mapped to β^k or vice versa. Note that there could be more than one eligible isomorphism.

Also by taking advantage of the fact that A^{17} is an element of F_2, the final operation $(A^{17})^{-1}A^{16}$ of Equation 9.15 can be easily computed with further gate reduction. Last stage of algorithm consists of mapping computed value in the composite field, back to the field GF(2^8).

To further increase the depth of a pipeline architecture, MI can be calculated by a composite field approach dealing MI manipulation in GF(2^2) and GF(2^4) instead of GF(2^8).

In [113], BS has been computed rather than using a look-up table. The main goal of using this formulation is to get a high-performance AES encryptor core without depending on look-up tables.

Using the composite field technique, BS arithmetic in GF(2^8) is performed via several arithmetic blocks in GF(2^4). This effectively reduces an 8-bit calculation to a 4-bit one, resulting on several stages of computation with lower delays. That allows obtaining a sort of sub-pipelining architecture in which, instead of having 11 unfolded stages (each stage corresponding to a single round), each single round is further unfolded into several stages. Thus, BS is (sub)divided into four pipeline stages where the first round takes only one stage, each middle round takes seven stages, and the final round, in which MC is not required, takes six stages.

In order to keep all stages balanced, i.e., propagating similar delays, a pipeline architecture with a depth of 70 stages was proposed in [113]. After 70 clock cycles when the pipeline is full, each clock cycle will deliver a ciphered block. This technique achieves a throughput of 25.107 Gbps, the fastest one reported up to date of this book publication.

The idea of dividing computations in sub fields is further exploited to its extreme in [42], where 4-bit calculations are broken into several 2-bit ones. Authors in [42] explored as many as 432 different isomorphisms. Polynomial as well as normal basis were considered and using an exhaustive tree- search algorithm [153], those isomorphisms requiring the minimum number of gates were selected. Logic optimizations both at the hierarchical level of the Galois

Field arithmetic and at the low level of individual logic gates were performed. The authors also reused common expressions to save space and noticing that NAND gates take less space than other ones, they rewrite all expressions in terms of such gates. Authors reported results exploring a family of 432 implementations depending on the selected basis ranking from 138 to 195 gates. Such compact $S-$box implementations can be used in security for low-end customer products, such as PDAs, wireless devices and other embedded applications.

9.4.2 MC/IMC Implementations on FPGA

The MC/IMC transformations are essentially the inner-product operations on GF(2^8) expressed in equations 9.6 and 9.9. They can be realized using byte-level or bit-level substructure sharing methods [140].

For an encryptor/decryptor core MC/IMC steps are implemented separately and they can be realized in a small series of instructions. In case of FPGAs, these instructions can be realized by keeping in mind the basic CLB structure (4 input/1 output) in order to limit path delays and to save space. Let us call this approach the MC/IMC standard approach. Fortunately, there exists another approach for which the implementation of IMC is made by introducing small modification before MC. The first approach is efficient but needs separate implementation for MC and IMC. The MC/IMC modified approach reuses some modules which eliminates the need for separated implementation of MC/IMC.

MC and IMC Transformation: Standard Approach

Observing that constant terms in equations 9.6 and 9.9 are the same, it is possible to consider only the inner product that generates one output byte, Z in MC and Z_{inv} in IMC, for an input column $[ABCD]^T$:

$$Z = \{01\}A \oplus \{01\}B \oplus \{02\}D \oplus \{03\}E \tag{9.19}$$

Using the property of $\{02\}D = \{02\}D \oplus 0 = \{02\}D \oplus D \oplus D$, we can rewrite equation 9.19 in the following manner:

$$Z = (A \oplus B \oplus D \oplus E) \oplus \{02\}(D \oplus E) \oplus D) \tag{9.20}$$

We can use an efficient implementation of constant multiplication by 02 in GF(2^8) calculated by the functional block $xtime(v)$ and extracting the common factor in all columns $t = (A \oplus B \oplus D \oplus E)$, then equation 9.19 can be rewritten as:

$$Z = t \oplus xtime(D \oplus E) \oplus D) \tag{9.21}$$

Therefore, full MC transformation can be efficiently computed by using only 3 steps [21, 60]: an addition step, a doubling step and a final addition step.

Let us consider a complete output row of MC transformation. Consider now the element of *State* matrix's column one $a[0]$, $a[1]$, $a[2]$, and $a[3]$, then the transformed MC column $a'[0]$, $a'[1]$, $a'[2]$, and $a'[3]$ can be efficiently obtained as shown in Equation 9.22.

$$\begin{array}{c} t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; \\ v = a[0] \oplus a[1];\ v = xtime(v);\ a'[0] = a[0] \oplus v \oplus t; \\ v = a[1] \oplus a[2];\ v = xtime(v);\ a'[1] = a[1] \oplus v \oplus t; \\ v = a[2] \oplus a[3];\ v = xtime(v);\ a'[2] = a[2] \oplus v \oplus t; \\ v = a[3] \oplus a[0];\ v = xtime(v);\ a'[3] = a[3] \oplus v \oplus t; \end{array} \tag{9.22}$$

Observe that t is a common expression for the four outputs and it needs to be calculated just once. Next four rows are calculated in parallel and the circuit is the same except for some input data. Finally, the sum of three terms requires only eight CLBs, one per bit. Given that CLBs can compute 4-input/1-output functions, it is possible to embed the ARK transformation, which is just a sum, to the final expression. This does not require additional CLBs and improves performance since MC and ARK are computed at the same stage. This is expressed in the following manner:

$$\begin{array}{ll} \multicolumn{1}{c}{Step1} & \multicolumn{1}{c}{Step2} \\ v = a[1] \oplus a[2] \oplus a[3]; & xt_0 = xtime(a[0]); \\ v = a[0] \oplus a[2] \oplus a[3]; & xt_1 = xtime(a[1]); \\ v = a[0] \oplus a[1] \oplus a[3]; & xt_2 = xtime(a[2]); \\ v = a[0] \oplus a[1] \oplus a[2]; & xt_3 = xtime(a[3]); \end{array}$$

$$\begin{array}{c} Step3 \\ a'[0] = k[0] \oplus v \oplus xt_0 \oplus xt_1; \\ a'[1] = k[1] \oplus v \oplus xt_1 \oplus xt_2; \\ a'[2] = k[2] \oplus v \oplus xt_2 \oplus xt_3; \\ a'[3] = k[3] \oplus v \oplus xt_3 \oplus xt_0; \end{array} \tag{9.23}$$

The same strategy applied above for MC can be used to compute IMC. Considering again an input column $[ABCD]^T$, we can expressed Z_{inv} as:

$$Z_{inv} = \{0d\}A \oplus \{09\}B \oplus \{0e\}D \oplus \{0b\}E \tag{9.24}$$

Using the same property for constant multiplication by $\{02\}$, we can rewrite Equation 9.24 in the following manner:

$$Z_{inv} = D \oplus N \oplus xtime(M \oplus N) \oplus xtime(D \oplus E) \tag{9.25}$$

where:

$$
\begin{aligned}
T_0 &= A \oplus B \oplus D \oplus E \\
T_1 &= T_0 \oplus xtime(xtime(T_0)) \\
N &= T_1 \oplus xtime(xtime(B \oplus E)) \\
M &= T_1 \oplus xtime(xtime(A \oplus D))
\end{aligned}
$$

Full IMC transformation can be computed by using seven steps: four sum steps and three doubling steps. The difference is due to the fact that coefficients in Equation 9.9 have a higher Hamming weight than the ones in Equation 9.6. To overcome this drawback, we use the strategy depicted in Equation 9.25 where IMC manipulation is restructured and seven steps are cut to five steps. Moreover, as explained above, IARK is embedded into IMC resulting in six total steps. For final round (Round 10), MC/IMC steps are not executed; therefore a separated implementation of ARK can be made. Let us consider now a complete output row of IMC transformation embedded with and IARK transformation, where a, and a' stand as before.

Step 1	*Step 2*	*Step 3*
$t = a[0] \oplus a[1] \oplus a[3]$		$u = s'_0 \oplus s'_1 \oplus s'_2 \oplus s'_3;$
$s_0 = xtime(a[0]);$	$s'_0 = xtime(s_0);$	$v = s_0 \oplus s_1 \oplus s'_0 \oplus s'_2;$
$s_1 = xtime(a[1]);$	$s'_1 = xtime(s_1);$	$v = s_1 \oplus s_2 \oplus s'_1 \oplus s'_3;$
$s_2 = xtime(a[2]);$	$s'_2 = xtime(s_2);$	$v = s_2 \oplus s_3 \oplus s'_0 \oplus s'_2;$
$s_3 = xtime(a[3]);$	$s'_3 = xtime(s_3);$	$v = s_3 \oplus s_0 \oplus s'_1 \oplus s'_3;$

Step 4	*Step 5*	*Step 6*
$u = xtime(u);$	$t' = t \oplus u;$	$a'[0] = a[0] \oplus t' \oplus v \oplus k[0];$
		$a'[1] = a[1] \oplus t' \oplus v \oplus k[1];$
		$a'[2] = a[2] \oplus t' \oplus v \oplus k[2];$
		$a'[3] = a[3] \oplus t' \oplus v \oplus k[3];$

(9.26)

MC and IMC Transformation: Modified Approach

The strategy utilized above for MC and IMC yields up to three and six computational steps for encryption and decryption respectively. In order to minimize difference in number of steps, the following strategy can be used.

Observe that it should exist a 4×4 byte matrix $D(x)$ in $\mathrm{GF}(2^8)$ such that the constant MC matrix of Equation 9.6 can be related to the constant matrix of Equation 9.9 as,

$$
\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} D(x) \tag{9.27}
$$

Using the fact that both constant matrices in Equation 9.27 are the inverse of each other in the finite field $F = \mathrm{GF}(2^8)$, equation 9.27 can be solved using the AES irreducible pentanomial $m(x) = x^8 + x^4 + x^3 + x + 1$ [60] for the first column of $D(x)$ as shown in Equation 9.28.

$$\begin{bmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 0E \\ 09 \\ 0D \\ 0B \end{bmatrix} \tag{9.28}$$

where $d_{i,0}$, $i = 0, 1, 2, 3$ represent the four coefficients of the first column of $D(x)$. It follows that Equation 9.28 has a unique solution in the finite field F as given in Equation 9.29,

$$d_{0,0} = 5 \quad d_{1,0} = 0 \quad d_{2,0} = 4 \quad d_{3,0} = 0 \tag{9.29}$$

Hence, Equation 9.27 can be re-written as shown in Eq. 9.30.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix} \tag{9.30}$$

Equation 9.30 suggests an efficient way to compute IMC by re-using the MC transformation to obtain IMC constant matrix. This is useful since constant elements of second matrix in the right side of Equation 9.30 have a less Hamming weight as compared to the constants of the original matrix for IMC.

9.4.3 Key Schedule Optimization

Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key arranged into 4 x 4 matrix of bytes. Then, those four columns are recursively expanded to obtain 40 more columns as follows. Let the columns up to $w[i-1]$ have been determined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \tag{9.31}$$

Where $T(w[i-1])$ a is non-linear transformation based on the application of the S-Box to the four bytes of the column. It involves also an additional cyclic rotation of the bytes within the column and the addition of a round constant ($rcon$) for symmetric elimination [60]. Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be represented as:

$$w[0] = \begin{bmatrix} k_0 \\ k_4 \\ k_8 \\ k_{12} \end{bmatrix} \quad w[1] = \begin{bmatrix} k_1 \\ k_5 \\ k_9 \\ k_{13} \end{bmatrix}$$
$$w[2] = \begin{bmatrix} k_2 \\ k_6 \\ k_{10} \\ k_{14} \end{bmatrix} \quad w[3] = \begin{bmatrix} k_3 \\ k_7 \\ k_{11} \\ k_{15} \end{bmatrix} \tag{9.32}$$

Then according to the above expressions, the new columns $w'[0]$, $w'[1]$, $w'[2]$, and $w'[3]$ of the next round key can be calculated as shown in Equation 9.33.

$$\begin{array}{ll} \textit{Step 1} & \textit{Step 2} \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_4 = k_4 \oplus k'_0; \\ k'_1 = k_0 \oplus SBox(k_{14}); & k'_5 = k_5 \oplus k'_1; \\ k'_2 = k_0 \oplus SBox(k_{15}); & k'_6 = k_6 \oplus k'_2; \\ k'_3 = k_0 \oplus SBox(k_{12}); & k'_7 = k_7 \oplus k'_3; \end{array}$$

$$\begin{array}{ll} \textit{Step 3} & \textit{Step 4} \\ k'_8 = k_8 \oplus k'_4; & k'_{12} = k_{12} \oplus k'_8; \\ k'_9 = k_9 \oplus k'_5; & k'_{13} = k_{13} \oplus k'_9; \\ k'_{10} = k_{10} \oplus k'_6; & k'_{14} = k_{14} \oplus k'_{10}; \\ k'_{11} = k_{11} \oplus k'_7; & k'_{15} = k_{15} \oplus k'_{11}; \end{array} \tag{9.33}$$

But it was mentioned before that in a typical FPGA device, a 4 input look-up table can be configured indistinctly to handle 2, 3, or 4 input logic gates. Hence, we can save some time by parallelizing the above computation using only two steps. By applying redundant computations, Equation 9.33 can be rewritten as it is shown in Equation 9.34 for the first row. Parallel computations are applied to obtain k'_4, k'_8, and k'_{12}.

$$\begin{array}{ll} \textit{Step1} & \textit{Step2} \\ & k'_4 = k_4 \oplus k'_0; \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_8 = k_4 \oplus k_8 \oplus k'_0; \\ & k'_{12} = k_4 \oplus k_8 \oplus k_{12} \oplus k'_0; \end{array} \tag{9.34}$$

9.5 AES Implementations on FPGAs

The basic organization of the hardware implementation of the AES algorithm is shown in Figure 9.13 which represents three blocks: encryptor/decryptor

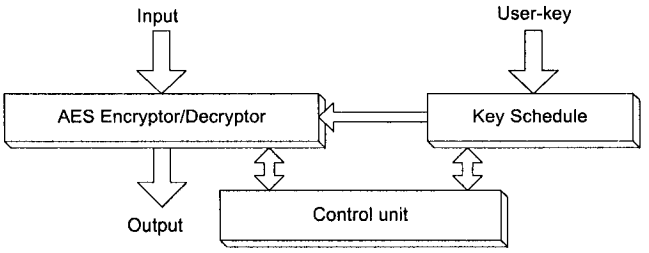

Fig. 9.13. Basic Organization of a Block Cipher

unit, key scheduling unit, and a control unit for synchronizing the flow of data between them.

Three main processes participate in AES:

- Key Schedule
- Encryption
- Decryption

The above three processes can be implemented using different design strategies showing distinct time-area tradeoffs. Depending on the application specification, the AES implementation can be carried out for just encryption, encryption/decryption on the same chip, separate encryption and decryption cores, or simply decryption. A separate implementation of AES encryptor or decryptor core would be less complex and efficient. Implementing AES encryptor/decryptor core on a single chip FPGA by mixing their common blocks, will give out an area efficient solution but one of them, either encryption or decryption could be performed at a time. To develop a full duplex operation having a capability to perform both encryption and decryption simultaneously would require relatively high hardware resources and consequently would become a bit slow.

For AES, key schedule implementations are different for an encryptor, decryptor or encryptor/decryptor cores. The usage of internal memory resources of an FPGA for storing pre-computed round-keys would be a simple approach. For encryption/decryption processes however it is recommendable not to use the same key for long time. A key schedule implementation will therefore provide a user the added flexibility of selecting encryption/decryption key of his own choice at any given time.

9.5.1 Architectural Alternatives for Implementing AES

Several approaches can be followed to implement AES on hardware achieving variable performance results [218].

Iterative architectures implement a reduced number of rounds (typically one) in an independent fashion. This kind of architectures occupy small area

of circuits but at the expense of low throughput. Unrolled architectures have a large number of rounds that are independently implemented in hardware. Pipelining allows to process multiples blocks of data at the same time at different stages to have higher throughput. Pipelining is achieved by putting rows of registers among different stages. Sub-pipelining inserts registers inside the round transformation to create sub-stages.

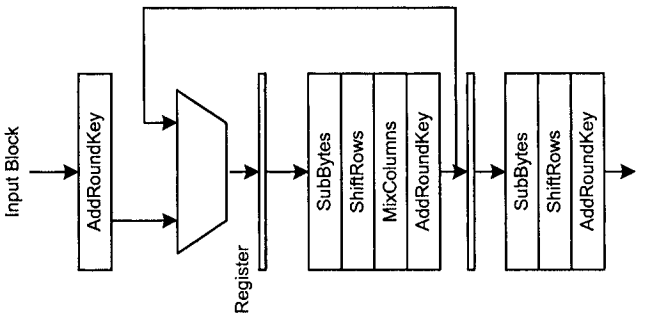

Fig. 9.14. Iterative Design Strategy

Block ciphers are of iterative nature, that is, n iterations of the same algorithm are made for a single encryption/decryption. An iterative design strategy would be a straightforward approach to implement the algorithm which executes n iterations of it by consuming n clock cycles for a single encryption/decryption as shown in Figure 9.14. The first round only considers ARK, the next nine rounds implement the four basic transformation, BS, SR, MC and ARK. The last round implements all but MC transformation. Clearly, it is an economical approach with respect to the hardware area and the cost has to be paid in terms of design speed which gets reduced with a factor of n. Such architectures would be useful for applications where hardware area is limited and speed is not more critical.

If reconfigurable platform is the choice for the implementation of a block cipher, a high speed architecture would result by implementing n rounds of the algorithm as modern FPGAs have enough logic density to accommodate massive circuits. The simplest way to improve performance is to use loop unrolling that expand the iterative structure by replicating rounds and conecting the output to the input of two consecutive rounds. This architecture is shown in Figure 9.15. By eliminating switches (multiplexers) and registers the accumulated delay can be reduced, but the duplication of multiple rounds incurs in large critical paths, which implies lower clock frequencies.

By putting registers between two consecutive rounds, which operate at the same clock cycle, we can achieve a pipeline architecture as shown in Figure 9.16.

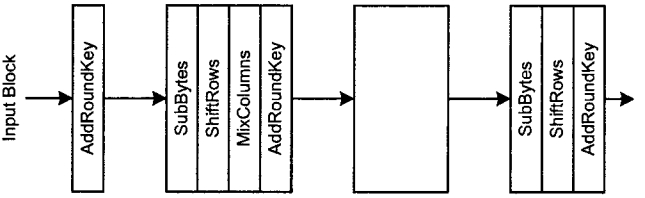

Fig. 9.15. Loop Unrolling Design Strategy

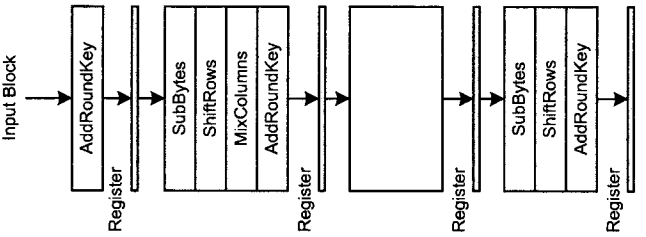

Fig. 9.16. Pipeline Design Strategy

Each round forms a pipeline stage of the data flow. The critical path is cut into stages although it is not diminished. The main advantage is that several different blocks can be processed at the same time but in different rounds of the encryption/decryption process. Once the pipeline is filled, the output blocks appear at each successive clock cycle. This allows to increase performance multiplied by the number of rounds or stages in the pipeline (typically eleven). This architecture increases throughput but it becomes costly in terms of hardware area.

FPGAs provide large number of flip-flops, which can be used to put several registers inside the different steps of a single round for a pipeline design strategy. This improves the performance of a pipeline architecture as those registers shift the internal results of a round while the final results are being transferred to the next round. It has been observed that careful use of those registers inside a round causes a significant increase in design performance. Figure 9.17 represents a sub-pipeline design strategy. This approach increases the depth of the pipeline up to 40 stages.

Although one can think that the increase in performance is folded as many times as the number of stages this is not completely exact. The problem is that all stages must have similar delays which is not true for AES. According to the formulation of BS, it is clear that its implementation takes longer delays than other basic transformations.

To keep balanced stages and at the same time to increase the depth of pipeline, we can break BS calculation by a four-stage composite field approach as it was explained in Section 9.4.1 and it is shown in Figure 9.18. Each middle

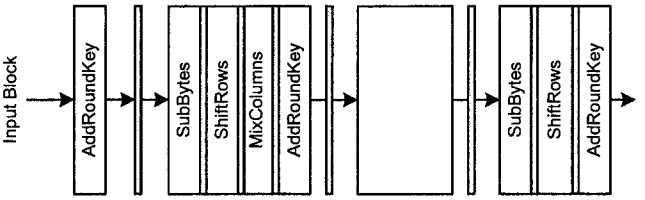

Fig. 9.17. Sub-pipeline Design Strategy

round is decomposed into seven stages, four from BS and one for SR, MC and ARK, each. That gives a 70 stages pipeline approach which reports high performance at the expense of great area requirements.

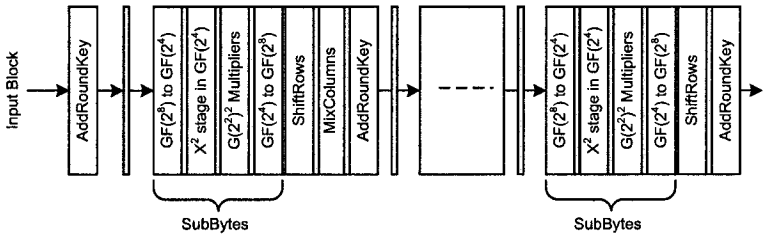

Fig. 9.18. Sub-pipeline Design Strategy with Balanced Stages

Pipelining and sub-pipelining are useful only when the cipher block is used in the ECB mode (electronic code book). As it was mentioned in Section 9.3, in the Output Feedback Mode (OFB) and in the CCM mode (Counter with CBC-MAC), pipelining looses its potential since a cipherblock is used to encrypt the next block. The only acceptable architecture for feed back modes is the iterative one, also called loop architecture.

In the rest of this section we disccuss some alternatives for implementing AES. All of them are intended to be implemented on a single-chip FPGA. There exists multi-chip implementations but as FPGA density is increasing, those implementations would be less meaningful in the future.

Varieties for AES implementation include encryptor, decryptor, and encryptor/decryptor cores using iterative or pipeline approaches. Each AES implementation targets specific criteria composed of factors like efficiency, cost, effectiveness and portability. Table 9.2 provides a roadmap to all implemented AES designs. It consideres four parameters: design (Sec.9.5), based on Section (Sec. 9.4), E/D/K module (encryption/decryption/key schedule) and architecture (encryptor, decryptor or encryptor/decryptor core). Key schedule implementations for encryptor, decryptor and encryptor/decryptor cores are also presented.

Table 9.2. A Roadmap to Implemented AES Designs

Design	Based on the Section	E/D/K Module	Architecture
Sec. 9.5.2	Sec. 9.4.3	(Key schedule)	For iterative & pipeline encryptor cores only
Sec. 9.5.2	Sec. 9.4.3	(Key schedule)	For Pipeline encryptor/decryptor cores
Sec. 9.5.3	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Iterative)
Sec. 9.5.3	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. 9.5.4	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor/decryptor core (Pipeline)
Sec. 9.5.4	Sec. 9.4.1 Sec. 9.4.2	S-box Composite field MC classic	Encryptor/decryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table Modified MC/IMC	Encryptor/decryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. 9.5.5	Sec. 9.4.1 Sec. 9.4.2	S-box Look-up table Modified IMC	Decryptor core (Pipeline)

All designs presented in this section were completely synthesized and succesfully implement using Xilinx Foundation Tool F4.1i. All designs are either coded in VHDL or by using libraries of the target devices. CoreGenerator is another tool used for design entry.

9.5.2 Key Schedule Algorithm Implementations

Let the user key consisting of 16 bytes be arranged as:

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix} \tag{9.35}$$

The process of generating next round key is optimized as discussed in Section 9.4.3 and is shown in Figure 9.19. The KGEN block consists of four similar units where each unit contains an S-Box and four XORs. The first block is slightly different as a constant predefined value (*rcon*) is XOR-ed in each round. As shown in Figure 9.19, last four bytes $k_{12}, k_{13}, k_{14}, k_{15}$, of each round key are substituted with the bytes from S-Box and then various XOR operations are performed to get the next round key.

The KGEN block is the basic building block used to generate round Keys for all AES implementations. However, the key management for producing

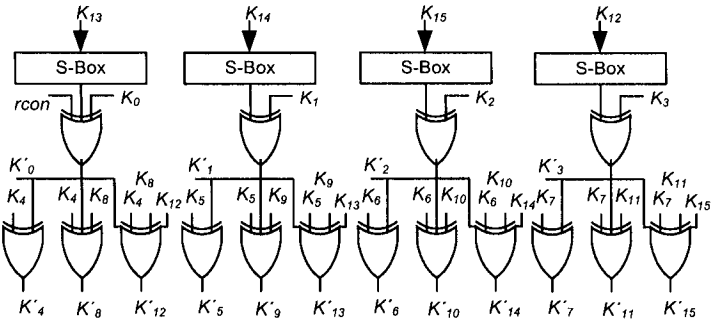

Fig. 9.19. KGEN Architecture

round keys differs depending on the particular implementation's strategy being used. For an encryptor core in iterative mode, round keys are also generated in iterative mode. For fully pipeline encryptor core, all round keys must be available before the encryption process starts. In a fully pipeline encryptor/decryptor core, the round keys for decryption are stored in reverse order as that of encryption.

Key Schedule for Iterative and Pipeline Encryptor Cores

For an encryptor core in iterative mode, a single round key is generated. The round key is fed to perform ARK step and also latched to feed back to KGEN block in order to get prepared for processing the next round key as shown in Figure 9.20. A multiplexer is used to switch the user-key first time and then for all rounds, each round key is used to generate the next round key.

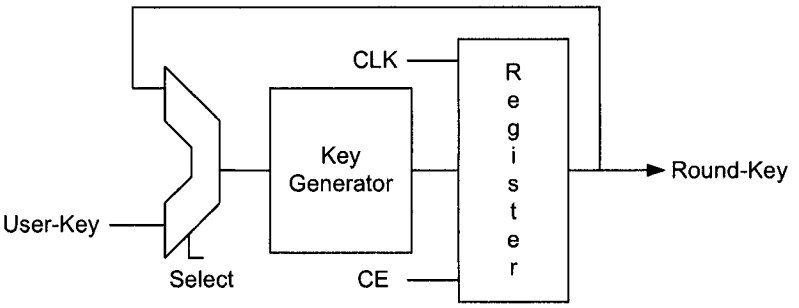

Fig. 9.20. Key Schedule for an Encryptor Core in Iterative Mode

For a fully pipelined encryptor core, the round keys must be available for each round permanently. The key generation process for a fully pipeline encryptor core is shown in Figure 9.21. The internal structure of each block is the same as shown in Figure 9.20, however, same block is replicated n (number of rounds) times. Once the round keys are generated, there is no need to repeat this process again and again. The same round keys serve for the whole session. For a fully pipeline encryptor core, the encryption process can be started in a parallel way, and there is no need to wait for the completion of all round keys.

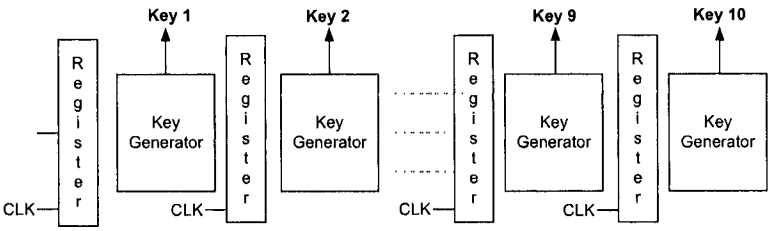

Fig. 9.21. Key Schedule for a Fully Pipeline Encryptor Core

Key Schedule for Encryptor/Decryptor Cores

For an encryptor/decryptor core on a single-chip FPGA, all the round keys must be generated and latched before the encryption/decryption processes start. The reason why round keys cannot be generated in a parallel way is because they are required in reverse order for decryption. The process of key generation is the same as explained above, however, round keys are stored in the registers for encryption and decryption in ascending or descending order respectively as shown in Figure 9.22. Besides this difference, the same blocks can be used for encryption and decryption processes.

As shown in Figure 9.22, round keys are generated by KGEN block as it was explained above by introducing two modifications. The first one deals with the generation of select signals (s_i) through an up/down counter. The main purpose of having those select signals is to choose the correct order for round keys either for the encryption or for the decryption process.

The second modification is the addition of IMC step which is required for generating round keys for decryption. It is applied through a multiplexer that allows passing round keys directly for encryption and switches the other line for applying IMC operation for the decryption round keys. IMC operation is performed before all the round keys are latched in their registers. Obeying algorithm description of the AES decryption process, this modification is not applied to first and last round keys.

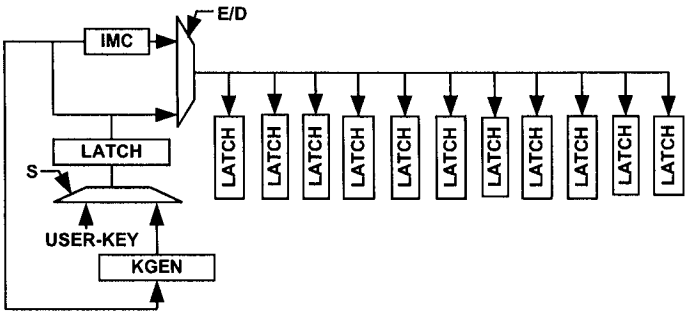

Fig. 9.22. Key Schedule for a Fully Pipeline Encryptor/Decryptor Core

IMC modifications discussed in Section 9.4.2 are applied in the IMC step for key scheduling as shown in Figure 9.23. This module is part of the second AES encryptor/decryptor core to be explained in the next Section.

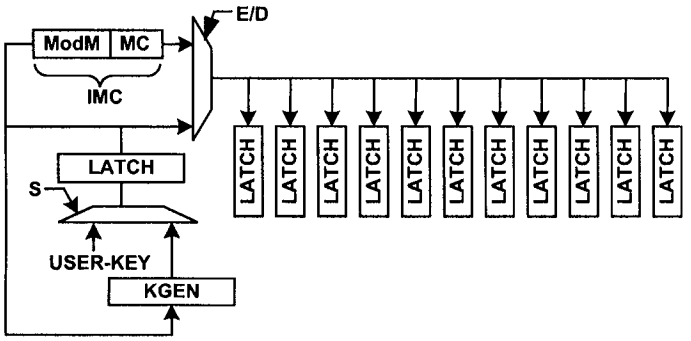

Fig. 9.23. Key Schedule for a Fully Pipeline Encryptor/Decryptor Core with Modified IMC

9.5.3 AES Encryptor Cores - Iterative and Pipeline Approaches

FPGAs implementations of AES encryptor cores are carried out using two strategies: iterative and pipeline.

AES Encryptor Core Using an Iterative Approach

For an iterative approach, instead of implementing n iterations of the algorithm, one iteration is implemented and n clock cycles are consumed to achieve final output. An AES iterative approach is shown in Figure 9.24.

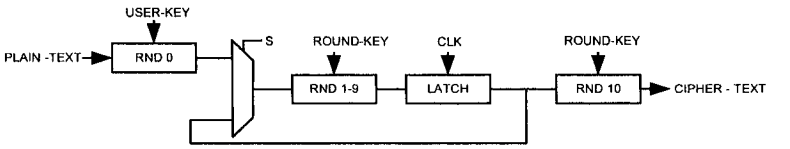

Fig. 9.24. Iterative Approach for AES Encryptor Core

The encryption process is presented in Figure 9.24, where RND0 is a simple ARK step: the user-key and plain-text are added. The RND1-9 block includes the four AES steps, namely, BS,SR,MC,ARK. Round keys are generated for all iterations of the algorithm. A multiplexer selects RND0 output at the first cycle and then selects the latch output for RND1-9 during the next nine cycles. RND10 is implemented separately without including the MC step.

The latch output is connected to the RND10 block and it is also fed-back to the multiplexer. All latch outputs passes through RND10 block but only during the tenth cycle its output is collected giving the final result. No clock cycle is therefore consumed to perform RND10.

Sixteen ROMs (256×8) are configured by using CLB in memory mode for performing the BS step of RND1-9. Since RND10 also includes the BS step, sixteen more ROMs are required for this step. The key scheduling algorithm also includes the BS step for the last four bytes of each round key (See Section 9.5.2) as shown in Figure 9.19, occupying four extra ROM blocks. A total of 36 ROM blocks are used for encryption part only. The SR step is combined with BS step. The MC and ARK steps are combined to reduce area requirements as discussed in Section 9.4.2.

The design was implemented on Xilinx VirtexE FPGA devices (XCV812BEG). It utilizes 36 ROMs, 385 I/O Blocks (95%) and 2744 slices (28%) to achieve a throughput of 258.5 Mbits/sec at 20.192 MHz. An encryption is completed in 10 clock cycles. That design does not make use of FPGA dedicated resources (BRAMs, etc.), hence it has a high portability and can be implemented virtually in every commercial FPGA device.

Fully Pipeline AES Encryptor Core

For a pipeline architecture, all AES rounds are unrolled. That is achieved by repeating one AES round 11 times as shown in Figure 9.25.

Similar to the iterative architecture, RND0 is just ARK step. The RND1-9 block includes all four steps BS, SR, MC, and ARK. The RND10 includes three steps BS, SR, ARK excluding MC step. 160 ROMs are required for 10 AES rounds instead of 16 ROMs occupied by the iterative architecture to perform BS step. Typically, the critical data path in pipeline architecture is longer, which implies that the design can run at lower speeds. However, by using dedicated memory modules BRAMs, as explained in the introduction Section, it is possible to reduce critical path delays.

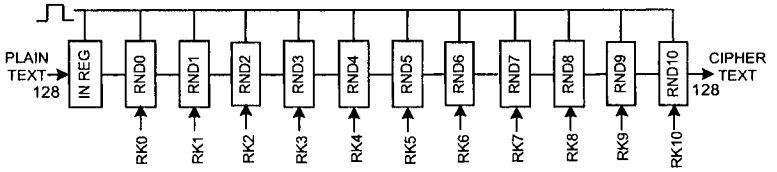

Fig. 9.25. Fully Pipeline AES Encryptor Core

The Virtex and VirtexE FPGA devices [397, 396] contain more than 280 BRAMs each of 4K. Each dual port BRAM can be configured as two single port BRAMs which reduces half of the memory requirements. A total of 80 BRAMs are therefore used to perform BS step. The same approach is used for key schedule implementation by occupying 20 BRAMs instead of 40 ROMs.

The design is targeted to Xilinx VirtexE FPGA devices (XCV812BEG) and occupies 2136 CLB slices (22%), 385 I/O Blocks (95%) and 100 BRAMs (35%). It uses a system clock of 22.41 MHz and data is processed at a rate of 2868 Mbits/sec. For a fully pipeline encryptor core, encryption starts from first clock cycle without initial delay. The round keys are generated in parallel. It takes 11 clock cycles to fill the pipeline first and then encrypted blocks start appearing at each consecutive clock cycle.

At first look, a comparison of the iterative and pipeline architectures suggests that the number of CLB slices occupied by the pipeline architecture seems to be less as compared to an iterative architecture. But this is accomplished at the price of occupying extra memory (100 BRAMs) needed to achieve desired fully pipeline architecture. The usage of dedicated memory resources (BRAMs) makes the pipeline design importable as it can only be targeted to those FPGA devices equipped with embedded memory functionality.

9.5.4 AES Encryptor/Decryptor Cores- Using Look-Up Table and Composite Field Approaches for S-Box

For an encryptor/decryptor core, each encryption step (BS, SR, MC, ARK) has its own inverse (IBS, ISR, IMC, IARK) which has to be implemented separately. The implementation of BS and IBS on a single chip is the most costly operation for AES implementation on FPGAs. In this design, two architectures are proposed for the BS/IBS implementation on FPGAs. First architecture proposes high performance implementations of BS/IBS step and second architecture is based on on-fly architecture scheme which tries to reduce memory requirements. The implementation of the remaining three steps SR, MC, and ARK is the same as the one described in Section 9.5.3. In the following, BS/IBS implementation strategies are discussed.

For encryption, BS implementation can be made by computing the Multiplicative Inverse (MI) of the input byte in $GF(2^8)$ followed by the affine

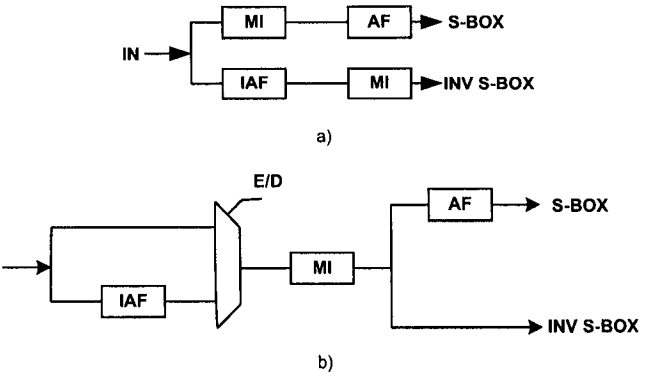

Fig. 9.26. S-Box and Inv S-Box Using (a) Different MI (b) Same MI

transformation (AF). For decryption, inverse affine transformation (IAF) is applied first followed by MI step. Implementing MI as look-up table requires memory modules, therefore, a separated implementation of BS/IBS causes the allocation of high memory requirements especially for a fully pipelined architecture. We can reduce such requirements by developing a single data path which uses one MI block for encryption and decryption. Figure 9.26 shows the BS/IBS implementation using single block for MI.

There are two design approaches for implementing MI: look-up table method and composite field calculation.

MI Using Look-Up Table Method

MI can be implemented using memory modules (BRAMs) of FPGAs by storing pre-computed values of MI. By configuring a dual port BRAM into two single port BRAMs, 8 BRAMs are required for one stage of a pipeline architecture, hence a total of 80 BRAMs are used for 10 stages. A separated implementation of AF and IAF is made. Data path selection for encryption and decryption is performed by using two multiplexers which are switched depending on the E/D signal. A complete description of this approach is shown in Figure 9.27

The data path for both encryption and decryption is, therefore, as follows:

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: ISR→ IAF→ MI→ IMC→IARK

The design targets Xilinx VirtexE FPGA devices (XCV2600) and occupies 80 BRAMs (43%), 386 I/O blocks (48%), and 5677 CLB slices (22.3%). It runs at 30 MHz and data is processed at 3840 Mbits/s.

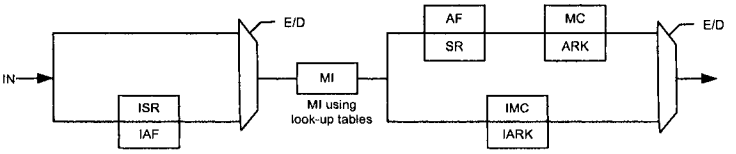

Fig. 9.27. Data Path for Encryption/Decryption

The data blocks are accepted at each clock cycle and then after 11 cycles, output encrypted/decrypted blocks appear at the output at consecutive clock cycles. It is an efficient fully pipeline encryptor/decryptor core for those cryptographic applications where time factor really matters.

MI with Composite Field Calculation

This is composite field approach that deals with MI manipulation in $GF(2^2)$ and $GF(2^4)$ instead of $GF(2^8)$ as it was explained in Section 9.4.1. It is a 3-stage strategy as shown in Figure 9.28.

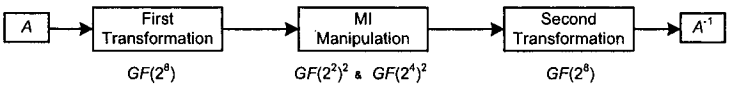

Fig. 9.28. Block Diagram for 3-Stage MI Manipulation

First and last stages transform data from $GF(2^8)$ to $GF(2^4)$ and vice versa. The middle stage manipulates inverse MI in $GF(2^4)$. The implementation of the middle stage with two initial and final transformations is represented in Figure 9.29 which depicts a block diagram of the three-stage inverse multiplier represented by Equations 9.15 and 9.17. It is noted that the Data path for encryption/decryption for this approach remains the same as the change in this approach is introduced in the MI manipulation.

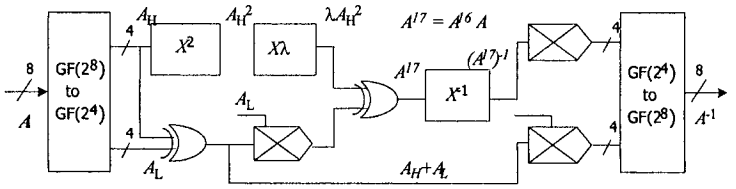

Fig. 9.29. Three-stage to Compute Multiplicative Inverse in Composite Fields

The circuit shown in Figure 9.30 and Figure 9.31 present a gate level implementation of the aforementioned strategy.

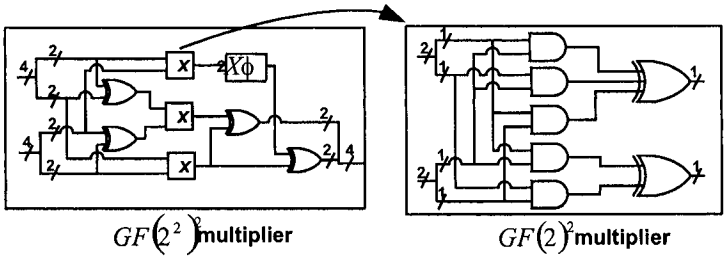

Fig. 9.30. $GF(2^2)^2$ and $GF(2^2)$ Multipliers

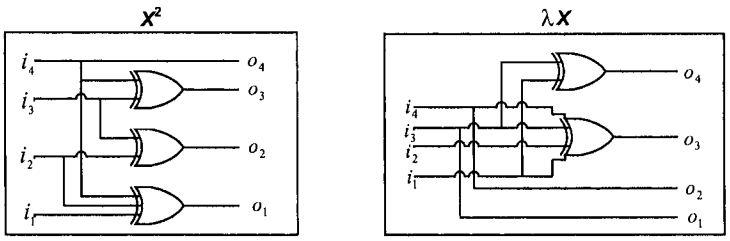

Fig. 9.31. Gate Level Implementation for x^2 and λx

The architecture is implemented on Xilinx VirtexE FPGA devices (XCV2600BEG) and occupies 12,270 CLB slices (48%), 386 I/O blocks (48%). It runs at 24.5 MHz and throughput achieved is 3136 Mbits/s. The increment on CLB slices utilized for this design is due to the manipulation for MI instead of using BRAMs. The increased design complexity causes the throughput to decrease when compared against the first design.

9.5.5 AES Encryptor/Decryptor, Encryptor, and Decryptor Cores Based on Modified MC/IMC

Three AES cores are presented in this Section. First design is an encryptor/decryptor core based on the ideas discussed in Section 9.4.2 for MC/IMC implementations. The second and third designs implement encryption and decryption paths separately for that design. There are two main reasons for the

separate implementation of encryption and decryption paths. First, to realize the effects of the modifications introduced in MC/IMC transformations. Second, most of reported AES implementations are either encryptor cores or encryptor/decryptor cores and few attention has been put to decryptor only cores.

Encryptor/Decryptor Core

This architecture reduces the large difference between the encryption/decryption time by exploiting the ideas explained in Section 9.4.2 for MC/IMC transformations. For this design, BS/IBS implementations are made by storing pre-computed MI values in FPGA's memory modules (BRAMs) with separate implementation of AF/IAF as explained in Section 9.5.4. The MC and ARK are combined together for encryption and a small modification ModM is applied before MC+ARK to get IMC operation as shown in Figure 9.32. Two multiplexers are used to switch the data path for encryption and decryption.

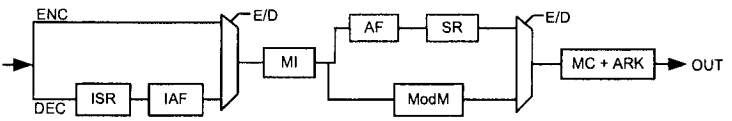

Fig. 9.32. AES Algorithm Encryptor/Decryptor Implementation

The data path for both encryption and decryption is, therefore, as follows:

Encryption: MI→ AF→ SR→ MC→ ARK
Decryption: ISR→ IAF→ MI→ ModM→ MC→ ARK

This AES encryptor/decryptor core occupies 80 BRAMs (43%), 386 I/O Blocks (48%) and 5677 slices (22.3%) by implementing on Xilinx VirtexE FPGA devices (XCV812BEG). It uses a system clock of 34.2 MHz and the data is processed at the rate of 4121 Mbits/sec. This is a fully pipeline architecture optimized for both time and space that performs at high speed and consumes less space.

Encryptor Core

It is a fully pipeline AES encryptor core. As it was already mentioned, the encryptor core implements the encryption path for AES encryptor/decryptor core explained in the last Section. The critical path for one encryption round is shown in Figure 9.33.

For BS step, pre-computed values of the S-Box are directly stored in the memories (BRAMs), therefore, AF transformation is embedded into BS. For

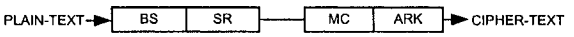

Fig. 9.33. The Data Path for Encryptor Core Implementation

the sake of symmetry, BS and SR steps are combined together. Similarly MC and ARK steps are merged to use 4-input/1-output CLB configuration which helps to decrement circuit time delays. The encryption process starts from the first clock cycle as the round-keys are generated in parallel as described in Section 9.5.2. Encrypted blocks appear at the output 11 clock cycles after, when the pipeline got filled. Once the pipeline is filled, the output is available at each consecutive clock cycle.

The encryptor core structure occupies 2136 CLB slices(22%), 100 BRAMs (35%) and 386 I/O blocks (95%) on targeting Xilinx VirtexE FPGA devices (XCV812BEG). It achieves a throughput of 5.2 Gbits/s at the rate of 40.575 MHz. A separated realization of this encryptor core provide a measure of timings for encryption process only. The results shows huge boost in throughput by implementing the encryptor core separately.

Decryptor Core

It is a fully pipeline decryptor core which implements the separate critical path for the AES encryptor/decryptor core explained before. The critical path for this decryptor core is taken from Figure 9.32 and then modified for IBS implementations. The resulting structure is shown in Figure 9.34.

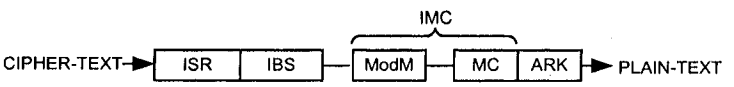

Fig. 9.34. The Data Path for Decryptor Core Implementation

The computations for IBS step are made by using look-up tables and precomputed values of inverse S-Box are directly stored into the memories (BRAMs). The IAF step is embedded into IBS step for symmetric reasons which is obtained by merely rewiring the register contains. The IMC step implementation is a major change in this design, which is implemented by performing a small modification ModM before MC step as discussed in Section 9.4.2. The MC and ARK steps are once again merged into a single module.

The decryption process requires 11 cycles to generate the entire round keys, then 11 cycles are consumed to fill up the pipeline. Once the pipeline is filled, decrypted plaintexts appear at the output after each consecutive clock cycle. This decryptor core achieves a throughput of 4.95 Gbits/s at the rate of 38.67 MHz by consuming 3216 CLB slices(34%), 100 BRAMs (35%) and 385

I/Os (95%). The implementation of decryptor core is made on Xilinx VirtexE FPGA devices (XCV812BEG).

A comparison between the encryptor and decryptor cores reveals that there is no big difference in the number of CLB slices occupied by these two designs. Moreover, the throughput achieved for both designs is quite similar. The decryptor core seems to be profited from the modified IMC transformation which resulted in a reduced data path. On the other hand, there is a significant performance difference between separated implementations of encryptor and decryptor cores against the combination of a single encryptor/decryptor implementation.

We conclude that separated cores for encryption and decryption provide another option to the end-user. He/she can either select a large FPGA device for combined implementation or prefer to use two small FPGA chips for separated implementations of encryptor and decryptor cores, which can accomplish higher gains in throughput.

Table 9.3. Specifications of AES FPGA implementations

	Core	Type	Device (XCV)	BRAMs	CLB(S) Slices	Throughput Mbits/s (T)	T/S
Sec. 9.5.4 [308]	E/D	P	2600E	80	6676	3840	0.58
Sec. 9.5.4 [308]	E/D	P	2600E		13416	3136	0.24
Sec. 9.5.5 [297]	E/D	P	2600E	100	5677	4121	1.73
Sec. 9.5.3 [311]	E	IL	812E		2744	258.5	0.09
Sec. 9.5.3 [311]	E	P	812E	100	2136	5193	2.43
Sec. 9.5.5 [307]	E	P	812E	100	2136	5193	2.43
Sec. 9.5.5 [306]	D	P	812E	100	3216	4949	1.54

9.5.6 Review of This Chapter Designs

The performance results obtained from the designs presented throughout this chapter are summarized in Table 9.3.

In Section 9.5.4 we presented two encryptor/decryptor cores. The first one utilized a Look-Up Table approach for performing the BS/IBS transformations. On the contrary, the second encryptor/decrpytor core computed the BS/IBS transformations based on an on-fly architecture scheme in $GF(2^4)$ and $GF(2^2)^2$ and does not occupy BRAMs. The penalty paid was on an increment in CLB slices.

The encryptor/decryptor core discussed in Section 9.5.5 exhibits a good performance which is obtained by reducing delay in the data paths for MC/IMC transformations, by using highly efficient memories BRAMs for BS/IBS computations, and by optimizing the circuit for long delays.

The encryptor core design of Section 9.5.3 was optimized for both area/time parameters and includes a complete set-up for encryption process. The user-

key is accepted and round-keys are subsequently generated. The results of each round are latched for next rounds and a final output appears at the output after 10 rounds. This increases the design complexity which causes a decrement in the throughput attained. However this design occupies 2744 CLB slices, which is acceptable for many applications.

Due to the optimization work for reducing design area, the fully pipeline architecture presented in Sections 9.5.3 and 9.5.5 consumes only 2136 CLB slices plus 100 BRAMs. The throughput obtained was of 5.2 Gbits/s. Finally, the decryptor core of (Sec. 9.5.5) achieves a throughput of 4.9 Gbits/s at the cost of 3216 CLB slices.

9.6 Performance

Since the selection of new advanced encryption standard was finalized on October, 2000, the literature is replete with reports of AES implementations on FPGAs. Three main features can be observed in most AES implementations on FPGAs.

1. **Algorithm's selection:** Not all reported AES architectures implement the whole process, i.e., encryption, decryption and key schedule algorithms. Most of them implement the encryption part only. The key schedule algorithm is often ignored as it is assumed that keys are stored in the internal memory of FPGAs or that they can be provided through an external interface. The FPGA's implementations at [102, 83, 63] are encryptor cores and the key schedule algorithm is only implemented in [63]. On the other hand the AES cores at [223, 366, 357] implement both encryption and decryption with key schedule algorithm.
2. **Design's strategy:** This is an important factor that is usually taken based on area/time tradeoffs. Several reported AES cores adopted various implementation's strategies. Some of them are iterative looping (IL) [102], sub-pipeline (SP) [83], one-round implementation [63]. Some fully pipeline (PP) architectures have been also reported in [223, 366, 357].
3. **Selection of FPGA:** The selection of FPGAs is another factor that influences the performance of AES cores. High performance FPGAs can be efficiently used to achieve high gains in throughput. Most of the reported AES cores utilized Virtex series devices (XCV812, XCV1000, XCV3200). Those are single chip FPGA implementations. Some AES cores achieved extremely high throughput but at the cost of multi-chip FPGA architectures [366, 357].

9.6.1 Other Designs

Comparing FPGA's implementations is not a simple task. It would be a fair comparison if all designs were tested under the same environment for all implementations. Ideally, performances of different encryptor cores should be

compared using the same FPGA, same design's strategies and same design specifications.

In this Section a summary of the most representative designs for AES in FPGAs is presented. We have grouped them into four categories: speed, compactness, efficiency, and other designs.

Table 9.4. AES Comparison: High Performance Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Good et al. [113]	E/D	P	XC3S2000-5	ECB	17425(0)	25107	1.44
Good et al. [113]	E/D	P	XCV2000e-8	ECB	16693(0)	23654	1.41
Zambreno et al. [400]	E	P	XC2V4000	ECB	16938(0)	23570	1.39
Saggese et al. [305]	E	P	XCVE2000-8	ECB	5819(100)	20,300	1.09
Standaert et al. [346]	E	P	VIRTEX3200E	ECB	15112(0)	18560	1.22
Jarvinen et al. [157]	E	P	XCV1000e-8	ECB	11719(0)	16500	1.40

*Throughput

In the first group, shown in Table 9.4, we present the fastest cores reported up to date. Throughput for those designs goes from 16.5 Gbps to 25.1 Gbits/s. To achieve such performances designers are forced to utilize pipelined architectures and, clearly, they need large amounts of hardware resources.

Up to this book's publication date, the fastest reported design achieved a throughput of 25.1 Gbits/s. It was reported in [113] and it applies a subpipelining strategy. The design divides BS transformation in four steps by using composite field computation. BS is expressed in computational form rather than as a look-up table. By expressing BS with composite field arithmetic, logic functions required to perform GF(2^8) arithmetic are expressed in several blocks of GF(2^4) arithmetic. That allows obtaining a sort of subpipelining architecture in which each single round is further unfolded into several stages with lower delays. This way, BS is divided into four subpipeline stages. As a result, there is a single stage in the first round, each middle round is composed of seven stages, while the final round, in which MC is not required, takes six stages. To keep balanced stages with similar delays, a pipeline architecture with a depth of 70 stages was developed. After 70 clock cycles once that the pipeline is full, each clock cycle delivers a ciphered block.

In the second group shown in Table 9.5 compact designs are shown. The bigger one in [297] takes 2744 slices without using BRAMs. The most compact design reported in [113] needs only 264 slices plus 2 BRAMS and it has a 2.2 Mbps throughput. In order to have a compact design it is necessary to have an iterative (loop) design. Since the main goal of these designs is to reduce hardware area, throughputs tend to be low. Thus, we can see that in general, the more compact a design is the lower its throughput.

Table 9.5. AES Comparison: Compact Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Good et al. [113]	E	IL	XCS2S15-6	ECB	264(2)	2.2	.008
Amphion CS5220 [7]	E	IL	XVE-8	ECB	421(4)	290	0.69
Weaver et al. [375]	E	IL	XVE600-8	ECB	460(10)	690	1.5
Chodowick et al. [52]	E	IL	XC2530-6	ECB	522(3)	166	0.74
Chodowick et al. [52]	E	IL	XC2530-5	ECB	522(3)	139	0.62
Rouvry et al. [302]	E	IL	XC3S50-4	ECB	1231(2)	87	0.07
Saqib [297]	E	IL	XCV812E	ECB	2744	258.5	0.09

*Throughput

Since BS is the most expensive transformation in terms of area, the idea of dividing computations in composite fields is further exploited in [113] to break 4-bit calculations into several 2-bit calculations. It is therefore a three stage strategy: mapping the elements to subfields, manipulation of the substituted value in the subfield and mapping of the elements back to the original field. Authors in [113] explored as many as 432 choices of representation both, in polynomial as well as normal basis representation of the field elements.

In the third group, a list of several designs is presented. We sorted the designs included according to the throughput over area ratio as is shown in Table 9.6[4]. That ratio provides a measure of efficiency of how much hardware area is occupied to achieve speed gains. In this group we can find iterative as well as pipelined designs. Among all designs considered, the design in [297] only included the encryption phase and the most efficient design in [223] reporting a throughput of 6.9 Gbps by occupying some 2222 CLB slices plus 100 BRAMs for BS transformation. We stress that we have ignored the usage of BRAMs in our estimations. If BRAMs are taken into consideration, then the design in [346] is clearly more efficient than the one in [223].

The designs in the first three categories implement ECB mode only. The fourth one, which is the shortest, reports designs with CTR and CBC feedback modes as shown in Table 9.7. Let us recall that a feedback mode requires an iterative architecture. The design reported in [214] has a good throughput/area tradeoff, since it takes only 731 slices plus 53 BRAMs, achieving a throughput of 1.06 Gbps.

As we have seen, most authors have focused on encryptor cores, implementing ECB mode only. There are few encryptor/decryptor designs reported. However, from the first three categories considered, we classified AES cores according to three different design criteria: a high throughput design, a compact design or an efficient design.

[4] In this figure of merit, we did not take into account the usage of specialized FPGA functionality, such as BRAMs.

Table 9.6. AES Comparison: Efficient Designs

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
McLoone et al. [223]	E	P	XCV812E	ECB	2222(100)	6956	3.10
Standaert et al. [346]	E	P	VIRTEX2300E	ECB	542(10)	1450	2.60
Saqib et al. [307]	E	P	XCV812E	ECB	2136(100)	5193	2.43
Saggese et al. [305]	E	IL	XCVE2000-8	ECB	446(10)	1000	2.30
Amphion CS5230 [7]	E	P	XVE-8	ECB	573(10)	1060	1.90
Rodríguez et al. [297]	E/D	P	XCV2600E	ECB	5677(100)	4121	1.73
López et al [214]	E	IL	Spartan 3 3s4000	ECB	633(53)	1067	1.68
Segredo et al. [325]	E	IL	XCV600E-8	ECB	496(10)	743	1.49
Segredo et al. [325]	E	IL	XCV-100-4	ECB	496(10)	417	0.84
Calder et al. [41]	E	IL	Altera EPF10K	ECB	1584	637.24	0.40
Labbé et al. [193]	E	IL	XCV1000-4	ECB	2151(4)	390	0.18
Gaj et al. [102]	E	IL	XCV1000	ECB	2902	331.5	0.11

*Throughput

Table 9.7. AES Comparison: Designs with Other Modes of Operation

Author	Core	Type	Device	Mode	Slices (BRAMs)	T* (Mbps)	T/A
Fu et al [100]	E	IL	XCV2V1000	CTR	2415 (NA)	1490	0.68
Charot et al. [49]	E	IL	Altera APEX	CTR	N/A	512	N/A
López et al [214]	E	IL	Spartan 3 3s4000	CBC	1031(53)	1067	1.03
López et al [214]	E	IL	Spartan 3 3s4000	CTR	731(53)	1067	1.45
Bae et al [15]	E	IL	Altera Stratix	CCM	5605(LC)	285	NA

*Throughput

After having analyzed the designs included in this Section, we conclude that there is still room for further improvements in designing AES cores for the feedback modes.

9.7 Conclusions

A variety of different encryptor, decryptor and encryptor/decryptor AES cores were presented in this Chapter. The encryptor cores were implemented both in iterative and pipeline modes. Some useful techniques were presented for the implementations of encryptor/decryptor cores, including: composite field approach for BS/IBS, look-up table method for BS/IBS, and modified MC/IMC approach.

All the architectures described produce optimized AES designs with different time and area tradeoffs. Three main factors were taking into account for implementing diverse AES cores.

- High performance: High performances can be obtained through the efficient usage of fast FPGA's resources. Similarly, efficient algorithmic techniques enhance design performance.
- Low cost solution: It refers to iterative architectures which occupy less hardware area at the cost of speed. Such architectures accommodate in smaller areas and consequently in cheaper FPGA devices.
- Portable architecture: A portable architecture can be migrated to most FPGA devices by introducing minor modifications in the design. It provides an option to the end-user to choose FPGA of his own choice. Portability can be achieved when a design is implemented by using the standard resources available in FPGA devices, i.e., the FPGA CLB fabric. A general methodology for achieving a portable architecture, in some cases, implies lesser performance in time.

For AES encryptor cores, both iterative and fully pipeline architectures were implemented. The AES encryptor/decryptor cores accomplished the BS/IBS implementation using two techniques: look-up table method and; composite fields. The latter is a portable and low cost solution.

The AES encryptor/decryptor core based on the modified MC/IMC is a good example of how to achieve high performance by using both efficient design and algorithmic techniques. It is a single-chip FPGA implementation that exhibits high performance with relatively low area consumption.

In short, time/area tradeoffs are always present, however by using efficient techniques at both, design and algorithm level, the always present compromise between area and time can be significantly optimized.

10

Elliptic Curve Cryptography

In this chapter we discuss several algorithms and their corresponding hardware architecture for performing the scalar multiplication operation on elliptic curves defined over binary extension fields $GF(2^m)$. By applying parallel strategies at every stage of the design, we are able to obtain high speed implementations at the price of increasing the hardware resource requirements. Specifically, we study the following four different schemes for performing elliptic curve scalar multiplications,

- Scalar multiplication applied on Hessian elliptic curves.
- Montgomery Scalar Multiplication applied on Weierstrass elliptic curves.
- Scalar multiplication applied on Koblitz elliptic curves.
- Scalar multiplication using the Half-and-Add Algorithm.

10.1 Introduction

Since its proposal in 1985 by [179, 236], many mathematical evidences have consistently shown that, bit by bit, Elliptic Curve Cryptography (ECC) offers more security than any other major public key cryptosystem.

From the perspective of elliptic curve cryptosystems, the most crucial mathematical operation is the *elliptic curve scalar multiplication*, which can be informally stated as follows. Let k be a positive integer and P a point on an elliptic curve. Then we define *elliptic curve scalar mutiplication* as the operation that computes the multiple $Q = kP$, defined as the point resulting of adding $P + P + \ldots + P$, k times. Algorithm 10.1 shows one of the most basic methods used for computing a scalar multiplication, which is based on a double-and-add algorithm isomorphic to the Horner's rule. As its name suggests, the two most prominent building blocks of this method are the *point*

doubling and *point addition* primitives. It can be verified that the computational cost of Algorithm 10.1 is given as $m-1$ point doublings plus an average of $\frac{m-1}{2}$ point additions.

The security of elliptic curve cryptosystems is based on the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP) that can be formulated as follows. Given an elliptic curve E defined over a finite field $GF(p^m)$ and two points Q and P that belong to the curve, where P has order r, find a positive scalar $k \in [1, r-1]$ such that the equation $Q = kP$ holds. Solving the discrete logarithm problem over elliptic curves is believed to be an extremely hard mathematical problem, much harder than its analogous one defined over finite fields of the same size.

Scalar multiplication is the main building block used in all the three fundamental ECC primitives: *Key Generation*, *Signature* and *Verification* schemes[1].

Although elliptic curve cryptosystems can be defined over prime fields, for hardware and reconfigurable hardware platform implementations, binary extension finite fields are preferred. This is largely due to the carry-free binary nature exhibit by this type of fields, which is a valuable characteristic for hardware systems leading to both, higher performance and lesser area consumption.

Many implementations have been reported so far [128, 334, 261, 333, 20, 311, 327, 46], and most of them utilize a six-layer hierarchical scheme such as the one depicted in Figure 10.1. As a consequence, high performance implementations of elliptic curve cryptography directly depend on the efficiency in the computation of the three underlying layers of the model.

The main idea discussed throughout this chapter is that each one of the three bottom layers shown in Figure 10.1 can be implemented using parallel strategies. Parallel architectures offer an interesting potential for obtaining a high timing performance at the price of area, implementations in [333, 20, 339, 9] have explicitly attempted a parallel strategy for computing elliptic curve scalar multiplication. Furthermore, for the first time a pipeline strategy was essayed for computing scalar multiplication on a $GF(P)$ elliptic curve in [122].

In this Chapter we present the design of a generic parallel architecture especially tailored for obtaining fast computation of the elliptic curves scalar multiplication operation. The architecture presented here exploits the inherent parallelism of two elliptic curves forms defined over GF(2^m): The Hessian form and the Weierstrass non-supersingular form. In the case of the Weierstrass form we study three different methods, namely,

- Montgomery point multiplication algorithm;
- The τ operator applied on Koblitz elliptic curves and;
- Point multiplication using halving

[1] Elliptic curve cryptosystem primitives, namely, Key generation, Digital Signature and Verification were studied in §2.5

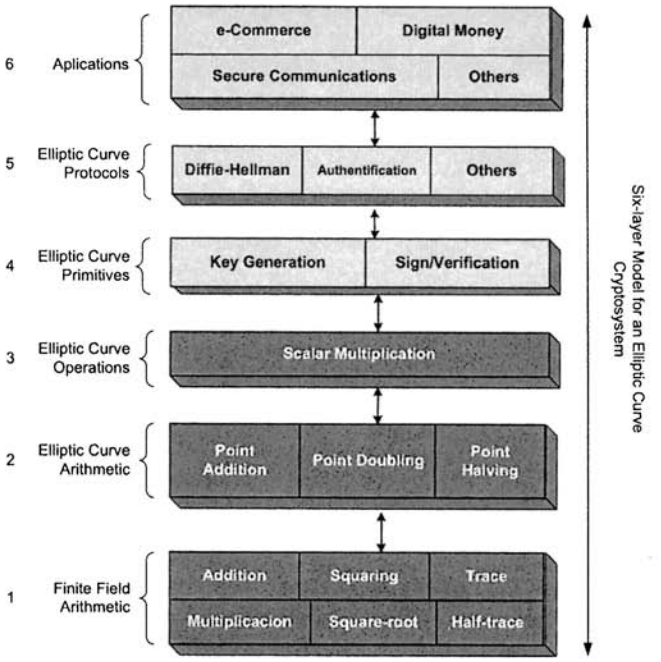

Fig. 10.1. Hierarchical Model for Elliptic Curve Cryptography

The rest of this Chapter is organized as follows. Section 10.2 briefly describe the Hessian form of an elliptic curve together with its corresponding group law. Then, in Section 10.3 we describe Weierstrass elliptic curve including a description of the Montgomery point multiplication algorithm. In Section 10.4 we present an analysis of how the ability of having more than one field multiplier unit can be exploited by designers for obtaining a high parallelism on the elliptic curve computations. Then, In Section 10.5 we describe the generic parallel architecture for elliptic curve scalar multiplication. Section 10.6 discusses some novels parallel formulations for the scalar multiplication on Koblitz curves. In Section 10.7 we give design details of a reconfigurable hardware architecture able to compute the scalar multiplication algorithm using halving. Section 10.8 includes a performance comparison of the design presented in this Chapter with other similar implementations previously reported. Finally, in Section 10.9 some concluding remarks are highlighted.

10.2 Hessian Form

Chudnvosky et al. presented in [53] a comprehensive study of formal group laws for reduced elliptic curves and Abelian varieties. In this section we discuss the Hessian form of elliptic curves and its corresponding group law followed by the Weierstrass elliptic curve form.

The original form for the law of addition on the general cubic was first developed by Cauchy and was later simplified by Sylvester-Desboves [316, 66]. Chudnovsky considered this particular elliptic curve form: "*By far the best and the prettiest*" [53]. In modern era, the Hessian form of Elliptic curves has been studied by Smart and Quisquater [335, 160].

Let $P(x)$ be a degree-m polynomial, irreducible over GF(2). Then $P(x)$ generates the finite field $\mathbb{F}_q = GF(2^m)$ of characteristic two. A Hessian elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y, z) \in GF(2^m) \times GF(2^m)$ that satisfy the canonical homogeneous equation,

$$x^3 + y^3 + z^3 = Dxyz \tag{10.1}$$

Together with the point at infinity denoted by $\mathcal{O}$ and given by $(1, 0, -1)$.

Let $P = (x_1, y_1, z_1)$ and $Q = (x_2, y_2, z_2)$ be two points that belong to the plane cubic curve of Eq. 10.1. Then we define $-P = (y_1, x_1, z_1)$ and $P + Q = (x_3, y_3, z_3)$ where,

$$\begin{aligned} x_3 &= {y_1}^2 x_2 z_2 - {y_2}^2 x_1 z_1 \\ y_3 &= {x_1}^2 y_2 z_2 - {x_2}^2 y_1 z_1 \\ z_3 &= {z_1}^2 y_2 x_2 - {z_2}^2 y_1 x_1 \end{aligned} \tag{10.2}$$

Provided that $P \neq Q$. The addition formulae of Eq. (10.2) might be parallelized using 12 field multiplications as follows [335],

$$\begin{array}{lll} \lambda_1 = y_1 x_2 & \lambda_2 = x_1 y_2 & \lambda_3 = x_1 z_2 \\ \lambda_4 = z_1 x_2 & \lambda_5 = z_1 y_2 & \lambda_6 = z_2 y_1 \\ s_1 = \lambda_1 \lambda_6 & s_2 = \lambda_2 \lambda_3 & s_3 = \lambda_5 \lambda_4 \\ t_1 = \lambda_2 \lambda_5 & t_2 = \lambda_1 \lambda_4 & t_3 = \lambda_6 \lambda_3 \\ x_3 = s_1 - t_1 & y_3 = s_2 - t_2 & z_3 = s_3 - t_3 \end{array} \tag{10.3}$$

Whereas the formulae for point doubling are giving by

$$\begin{aligned} x_3 &= y_1 \left({z_1}^3 - {x_1}^3 \right); \\ y_3 &= x_1 \left({y_1}^3 - {z_1}^3 \right); \\ z_3 &= z_1 \left({x_1}^3 - {y_1}^3 \right). \end{aligned} \tag{10.4}$$

Where $2P = (x_3, y_3, z_3)$. The doubling formulae of Eq. (10.4) can be also parallelized requiring 6 field multiplications plus three field squarings for their computation. The resulting arrangement can be rewritten as [335],

$$\begin{array}{lll} \lambda_1 = {x_1}^2 & \lambda_2 = {y_1}^2 & \lambda_3 = {z_1}^2; \\ \lambda_4 = x_1 \lambda_1 & \lambda_5 = y_1 \lambda_2 & \lambda_6 = z_1 \lambda_3; \\ \lambda_7 = \lambda_5 - \lambda_6 & \lambda_8 = \lambda_6 - \lambda_4 & \lambda_9 = \lambda_4 - \lambda_5; \\ x_2 = y_1 \lambda_8 & y_2 = x_1 \lambda_7 & z_2 = z_1 \lambda_9; \end{array} \tag{10.5}$$

Algorithm 10.1 Doubling & Add algorithm for Scalar Multiplication: MSB-First

Require: $k = (k_{m-1}, k_{m-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF(2^m))$
Ensure: $Q = kP$
1: $Q = P$;
2: **for** $i = m - 2$ downto 0 **do**
3: $Q = 2 \cdot Q$; /*point doubling*/
4: **if** $k_i = 1$ **then**
5: $Q = Q + P$; /*point addition*/
6: **end if**
7: **end for**
8: **Return** Q

By implementing Eqs. (10.3) and (10.5), one can obtain the two building blocks needed for the implementation of the second layer shown in Figure 10.1. Hence, provided that those two blocks are available, one can compute the third layer of Figure 10.1 by using the well-known doubling and add Algorithm 10.1. That sequential algorithm needs an average of $\frac{m-1}{2}$ point additions plus m point doublings in order to complete one scalar multiplication computation.

Alternatively, we can use the algorithm of Figure 10.2 that can potentially be implemented in parallel since in this case the point addition and doubling operations do not show any dependencies between them. Therefore, if we assume that the algorithm of Figure 10.2 is implemented in parallel, its execution time in average will be of that of approximately $\frac{m}{2}$ point additions plus $\frac{m}{2}$ point doublings[2].

In Subsection 10.4 we discuss how to obtain an efficient parallel-sequential implementation of the second and third layers of the model of Figure 10.1.

Algorithm 10.2 Doubling & Add algorithm for Scalar Multiplication: LSB-First

Require: $k = (k_{m-1}, k_{m-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF(2^m))$
Ensure: $Q = kP$
1: $Q = 1$; $R = P$;
2: **for** $i = 0$ to $m - 1$ **do**
3: **if** $k_i = 1$ **then**
4: $Q = Q + R$; /*point addition*/
5: **end if**
6: $R = 2 \cdot R$; /*point doubling*/
7: **end for**
8: **Return** Q

[2] Because of the inherent parallelism of this algorithm, $\frac{m}{2}$ point doublings computations can be overlapped with the execution of about $\frac{m}{2}$ point additions.

10.3 Weierstrass Non-Singular Form

As it was already studied in Section 4.3, a Weierstrass non-supersingular elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the affine equation,

$$y^2 + xy = x^3 + ax^2 + b, \tag{10.6}$$

Where a and $b \in \mathbb{F}_q, b \neq 0$, together with the point at infinity denoted by $\mathcal{O}$. The Weierstrass elliptic curve group law for affine coordinates is given as follows.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve 10.6 then $-P = (x_1, x_1 + y_1)$. For all P on the curve $P + \mathcal{O} = \mathcal{O} + \mathcal{P} = \mathcal{P}$. If $Q \neq -P$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a & P \neq Q \\ x_1^2 + \frac{b}{x_1^2} & \mathrm{P = Q} \end{cases} \tag{10.7}$$

$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1 & P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3 & \mathrm{P = Q} \end{cases} \tag{10.8}$$

From Eqns. (10.7) and (10.8) it can be seen that for both of them, point addition (when $P \neq -Q$) and point doubling (when $P = Q$), the computations for (x_3, y_3) require one field inversion and two field multiplications[3].

Notice also (a clever observation first made by Montgomery) that the x-coordinate of $2P$ does not involve the y-coordinate of P.

10.3.1 Projective Coordinates

Compared with field multiplication in affine coordinates, inversion is by far the most expensive basic arithmetic operation in $GF(2^m)$. Inversion can be avoided by means of projective coordinate representation. A point P in projective coordinates is represented using three coordinates X, Y, and Z. This representation greatly helps to reduce internal computational operations[4]. It is customary to convert the point P back from projective to affine coordinates in the final step. This is due to the fact that affine coordinate representation involves the usage of only two coordinates and therefore is more useful for external communication saving some valuable bandwidth.

In *standard* projective coordinates the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z$. The projective equation of the elliptic curve is given as:

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \tag{10.9}$$

[3] The computational costs of field additions and squarings are usually neglected.

[4] Projective Coordinates were studied in more detail in §4.5

10.3.2 The Montgomery Method

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve of Equation 10.6. Then $P + Q = (x_3, y_3)$ and $P - Q = (x_4, y_4)$, also belong to the curve and it can be shown that x_3 is given as [128],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2; \tag{10.10}$$

Hence we only need the x coordinates of P, Q and $P - Q$ to exactly determine the value of the x-coordinate of the point $P + Q$. Let the x coordinate of P be represented by X/Z. Then, when the point $2P = (X_2, -, Z_2)$ is converted to projective coordinate representation, it becomes [211],

$$\begin{aligned} X_2 &= X^4 + b \cdot Z^4; \\ Z_2 &= X^2 \cdot Z^2; \end{aligned} \tag{10.11}$$

The computation of Eq. 10.11 requires one general multiplication, one multiplication by the constant b, five squarings and one addition. Fig. 10.3 is the sequence of instructions needed to compute a single point doubling operation $Mdouble(X_1, Z_1)$ at a cost of two field multiplications.

Algorithm 10.3 Montgomery Point Doubling

Require: $P = (X_1, -, Z_1) \in E(GF(2^m)), c$ such that $c^2 = b$
Ensure: $P = 2 \cdot P$ /* Mdouble(X_1, Z_1)*/
1: $T = X_1^2$;
2: $M = c \cdot Z_1^2$;
3: $Z_2 = T \cdot Z_1^2$;
4: $M = M^2$;
5: $T = T^2$;
6: $X_2 = T + M$;
7: **Return** (X_2, Z_2)

In a similar way, the coordinates of $P + Q$ in projective coordinates can be computed as the fraction X_3/Z_3 and are given as:

$$\begin{aligned} Z_3 &= (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \\ X_3 &= x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1); \end{aligned} \tag{10.12}$$

The required field operations for point addition of Eq. 10.12 are three general multiplications, one multiplication by x, one squaring and two additions. This operation can be efficiently implemented as shown in Fig. 10.4.

Algorithm 10.4 Montgomery Point Addition

Require: $P = (X_1, -, Z_1), Q = (X_2, -, Z_2) \in E(GF2^m)$
Ensure: $P = P + Q$ /* Madd(X_1, Z_1, X_2, Z_2)*/
1: $M = (X_1 \cdot Z_2) + (Z_1 \cdot X_2)$;
2: $Z_3 = M^2$;
3: $N = (X_1 \cdot Z_2) \cdot (Z_1 \cdot X_2)$;
4: $M = x \cdot Z_3$;
5: $X_3 = M + N$;
6: **Return** (X_3, Z_3)

Montgomery Point Multiplication

A method based on the formulas for doubling (from Eq. 10.11) and for addition (from Eq. 10.12) is shown in Fig. 10.5 [211]. Notice that steps 2.2 and 2.3 are formulae for point doubling (*Mdouble*) and point addition (*Madd*) from Figs. 10.3 and 10.4 respectively. In fact both *Mdouble* and *Madd* operations are executed in each iteration of the algorithm. If the test bit k_i is '1', the manipulations are made for $Madd(X_1, Z_1, X_2, Z_2)$ and $Mdouble(X_2, Z_2)$ (steps 5-6) else $Madd(X_2, Z_2, X_1, Z_1)$ and $Mdouble(X_1, Z_1)$, i.e., *Mdouble* and *Madd* with reversed arguments (step 8-9).

The approximate running time of the algorithm shown in Fig. 10.5 is $6m$M + ($1I$ + 10M) where M represents a field multiplication operation, m stands for the number of bits and I corresponds to inversion. It is to be noted that the factor ($1I$ + 10M) represents time needed to convert from standard projective to affine coordinates. In the next Subsection we explain the conversion from SP to affine coordinates and then in Subsection 10.4, we discuss how to obtain an efficient parallel implementation of the above algorithm.

Conversion from Standard Projective (SP) to Affine Coordinates

Both, point addition and point doubling algorithms are presented in standard projective coordinates. A conversion process is therefore needed from SP to affine coordinates. Referring to the algorithm of Fig. 10.5, the corresponding affine x-coordinate is obtained in step 3:

$$x_3 = X_1/Z_1.$$

Whereas the affine representation for the y-coordinate is computed by step 4:

$$y_3 = (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y.$$

Notice also that both expressions for x_3 and y_3 in affine coordinates include one inversion operation. Although this conversion procedure must be performed only once in the final step, still it would be useful to minimize the number of inversion operations as much as possible. Fortunately it is possible to reduce one inversion operation by using the common operations from

Algorithm 10.5 Montgomery Point Multiplication

Require: $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$, $P(x, y, z) \in E(GF2^m)$
Ensure: $Q = kP$
1: $X_1 = x$;, $Z_1 = 1$;
2: $X_2 = x^4 + b$;, $Z_2 = x^2$;
3: **for** $i = n - 2$ downto 0 **do**
4: **if** $k_i = 1$ **then**
5: $Madd(X_1, Z_1, X_2, Z_2)$;
6: $Mdouble(X_2, Z_2)$;
7: **else**
8: $Madd(X_2, Z_2, X_1, Z_1)$;
9: $Mdouble(X_1, Z_1)$;
10: **end if**
11: **end for**
12: $x_3 = X_1/Z_1$;
13: $y_3 = (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$;
14: **Return** (x_3, y_3)

the conversion formulae for both x and y-coordinates. A possible sequence of the instructions from SP to affine coordinates is given by the algorithm in Fig. 10.6.

Algorithm 10.6 Standard Projective to Affine Coordinates

Require: $P = (X_1, Z_1)$, $Q = (X_2, Z_2)$, $P(x, y) \in E(GF2^m)$
Ensure: (x_3, y_3) /* affine coordinates */
1: $\lambda_1 = Z_1 \times Z_2$;
2: $\lambda_2 = Z_1 \times x$;
3: $\lambda_3 = \lambda_2 + X_1$;
4: $\lambda_4 = Z_2 \times x$;
5: $\lambda_5 = \lambda_4 + X_1$;
6: $\lambda_6 = \lambda_4 + X_2$;
7: $\lambda_7 = \lambda_3 \times \lambda_6$;
8: $\lambda_8 = x^2 + y$;
9: $\lambda_9 = \lambda_1 \times \lambda_8$;
10: $\lambda_{10} = \lambda_7 + \lambda_9$;
11: $\lambda_{11} = x \times \lambda_1$;
12: $\lambda_{12} = inverse(\lambda_{11})$;
13: $\lambda_{13} = \lambda_{12} \times \lambda_{10}$;
14: $x_3 = \lambda_{14} = \lambda_5 \times \lambda_{12}$;
15: $\lambda_{15} = \lambda_{14} + x$;
16: $\lambda_{16} = \lambda_{15} \times \lambda_{13}$;
17: $y_3 = \lambda_{16} + y$;
18: **Return** (x_3, y_3)

The coordinate conversion process makes use of 10 multiplications and only 1 inversion ignoring addition and squaring operations.

The algorithm in Fig. 10.6 includes one inversion operation which can be performed using Extended Euclidean Algorithm or Fermat's Little Theorem (FLT)[5].

10.4 Parallel Strategies for Scalar Point Multiplication

As it was mentioned in the introduction Section, parallel implementations of the three underlying layers depicted in Figure 10.1 constitutes the main interest of this Chapter. Several parallel techniques for performing field arithmetic, i.e. the first Layer of the model, were discussed in Chapter 5. However, hardware resource limitations restrict us from attempting a fully parallel implementation of second and third layers. Thus, a compromising strategy must be adopted to exploit parallelism at second and third layers.

Let us suppose that our hardware resources allow us to accommodate up to two field multiplier blocks. Under this scenario, the Hessian form point addition primitive $(x_3 : y_3 : z_3) = (x_1 : y_1 : z_1) + (x_2 : y_2 : z_2)$ studied in Section 10.2 can be accomplished in just six clock cycles as[6],

$$
\begin{array}{lll}
\textbf{Cycle 1}: & \lambda_1 = y_1 \cdot x_2; & \lambda_2 = x_1 \cdot y_2; \\
\textbf{Cycle 2}: & \lambda_3 = x_1 \cdot z_2; & \lambda_4 = z_1 \cdot x_2; \\
\textbf{Cycle 3}: & \lambda_5 = z_1 \cdot y_2; & \lambda_6 = z_2 \cdot y_1; \\
\textbf{Cycle 4}: & s_1 = \lambda_1 \cdot \lambda_6; & s_2 = \lambda_2 \cdot \lambda_3; \\
\textbf{Cycle 5}: & s_3 = \lambda_5 \cdot \lambda_4; & t_1 = \lambda_2 \cdot \lambda_5; \\
\textbf{Cycle 6}: & t_2 = \lambda_1 \cdot \lambda_4; & t_3 = \lambda_6 \cdot \lambda_3; \\
\textbf{Cycle 6.a}: & x_3 = s_1 - t_1; & y_3 = s_2 - t_2;\ z_3 = s_3 - t_3;
\end{array}
$$

Similarly, the Hessian point doubling primitive, namely, $2(x_1 : y_1 : z_1) = (x_2 : y_2 : z_2)$ can be performed in just 3 cycles as[7],

$$
\begin{array}{lll}
\textbf{Cycle 1}: & \lambda_1 = {x_1}^2;\ \lambda_2 = {y_1}^2; & \lambda_3 = {z_1}^2; \\
\textbf{Cycle 1.a}: & \lambda_4 = x_1 \cdot \lambda_1; & \lambda_5 = y_1 \cdot \lambda_2; \\
\textbf{Cycle 2}: & \lambda_6 = z_1 \cdot \lambda_3; & z_2 = z_1 \cdot (\lambda_4 - \lambda_5); \\
\textbf{Cycle 2.a}: & \lambda_7 = \lambda_5 - \lambda_6; & \lambda_8 = \lambda_6 - \lambda_4; \\
\textbf{Cycle 3}: & x_2 = y_1 \cdot \lambda_8; & y_2 = x_1 \cdot \lambda_7;
\end{array}
$$

The same analysis can be carried out for the Montgomery point multiplication primitives. The Montgomery point doubling primitive $2(X_1 : - : Z_1) =$

[5] Efficient multiplicative inverse algorithms were studied in §6.3.

[6] Because of their simplicity, the arithmetic operations of Cycle 6.a can be computed during the execution of Cycle 6.

[7] Due to the simplicity of the arithmetic operations included in cycles 1 and 2.a above, those operations can be merged with the operations performed in cycles 1.a and 2, respectively.

$(X_2 : - : Z_2)$ when using two multiplier blocks can be accomplished in just one clock cycle as,

$$\begin{array}{ll} \textbf{Cycle 1}: & T = X_1^2; \quad M = c \cdot Z_1^2; \; Z_2 = T \cdot Z_1^2; \\ \textbf{Cycle 1.a}: & X_2 = T^2 + M^2; \end{array} \tag{10.13}$$

Whereas, the Montgomery point addition primitive $(X_1 : - : Z_1) = (X_1 : - : Z_1) + (X_2 : - : Z_2)$ when using two multiplier blocks can be accomplished in just two clock cycles as,

$$\begin{array}{ll} \textbf{Cycle 1}: & t_1 = (X_1 \cdot Z_2); \; t_2 = (Z_1 \cdot X_2); \\ \textbf{Cycle 1.a}: & M = t_1 + t_2; \quad Z_1 = M^2; \\ \textbf{Cycle 2}: & N = t_1 \cdot t_2; \quad M = x \cdot Z_1; \\ \textbf{Cycle 2.a}: & X_1 = M + N; \end{array} \tag{10.14}$$

If two multiplier blocks are available, we can choose whether we want to parallelize the second or the third Layer of the model shown in Fig.10.1.

Algorithm 10.5, i.e. the third Layer of Fig. 10.1, can be executed in parallel by assigning one of our two multiplier blocks to compute the Montgomery point addition of Algorithm 10.4, and the other to perform the Montgomery point doubling of Algorithm 10.3. Then, the corresponding computational cost of point addition and point doubling primitives become of four and two field multiplications, respectively. In exchange, steps 5-6 and 8-9 of Algorithm 10.5 can be performed in parallel. Since those steps can be performed concurrently their associated execution time reduces to about 4 field multiplications. Therefore, the execution time associated to Algorithm 10.5 would be equivalent to $4m$ field multiplications[8].

Alternatively, the second layer can be executed in parallel by using our two multiplier blocks for computing point addition and point doubling in just 2 and 1 cycles, as it was shown in Eqs.(10.14) and (10.13), respectively. However, this decision will force us to implement Algorithm 10.5 (corresponding to the third layer of Fig.10.1) in a sequential manner. Therefore, the execution time associated to Algorithm 10.5 would be equivalent to $3m$ field multiplications.

If our hardware resources allow us to implement up to four field multiplier blocks, then we can execute both, the second and third Layers of Fig.10.1 in parallel. In that case the execution time of Algorithm10.5 reduces to just $2m$ field multiplications.

It is noticed that this high parallelism achieved by the Montgomery point multiplication method cannot be achieved by the Hessian form of the Elliptic curve.

Table 10.1 presents four of the many options that we can follow in order to parallelize the computation of scalar point multiplication. The computational costs shown in Table 10.1 are normalized with respect to the required number

[8] Since we can execute concurrently the procedures *Mdouble* and *Madd* the execution time of the former is completely overlapped by the latter.

Table 10.1. $GF(2^m)$ Elliptic Curve Point Multiplication Computational Costs

Strategy		Req. No.	EC Operation Cost		Equivalent	EC Operation Cost		Equivalent
2nd	3rd	of Field	Hessian Form		Time	Montgomery Algorithm		Time
Layer	Layer	Mults.	Doubling	Addition	Costs	Doubling	Addition	Costs
Sequential	Sequential	1	$6M$	$12M$	$12mM$	$2M$	$4M$	$6mM$
Sequential	Parallel	2	$6M$	$12M$	$9mM$	$2M$	$4M$	$4mM$
Parallel	Sequential	2	$3M$	$6M$	$6mM$	$1M$	$2M$	$3mM$
Parallel	Parallel	4	$3M$	$6M$	$\frac{3}{2}mM$	M	$2M$	$2mM$

of field multiplication operations (since the computation time of squaring operations is usually neglected in arithmetic over GF(2^m)).

Notice that the computation times of the Hessian form has been estimated assuming that the scalar multiplication has been accomplished by executing Algorithm 10.2. For instance, the execution time of the Hessian form in the fourth row of Table 10.1 has been estimated as follows,

$$\text{Time Cost} = \frac{m}{2}PD + \frac{m}{2}PA = \frac{3m}{2}M + \frac{6m}{2}M = \frac{9m}{2}M.$$

Due to area restrictions we can afford to accommodate up to two fully parallel field multipliers in our design. Thus, we can afford both, second and third options of Table 10.1. However, third option is definitely more attractive as it demonstrates better timing performance at the same area cost. Therefore, and as it is indicated in the third row of Table 10.1, the estimated computational cost of our elliptic curve Point multiplication implementation will be of $6m$ field multiplications in Hessian form. It costs only $3m$ field multiplications using the Montgomery algorithm for the Weierstrass form.

In the next Section we discuss how this approach can be carried out on hardware platforms.

10.5 Implementing scalar multiplication on Reconfigurable Hardware

Figure 10.2 shows a generic structure for the implementation of elliptic curve scalar multiplication on hardware platforms. That structure is able to implement the parallel-sequential approach listed in the third row of Table 10.1, assuming the availability of two GF(2^m) multiplier blocks.

In the rest of this Section, it is presupposed that two fully-parallel GF(2^{191}) Karatsuba-Ofman field multipliers can be accommodated on the target FPGA device.

The architecture in Figure 10.2 is comprised of four classes of blocks: field multipliers, Combinational logic blocks and/or finite field arithmetic (i.e. squaring, etc.), Blocks for intermediate results storage and selection (i.e. registers, multiplexers, etc.), and a Control unit (CU).

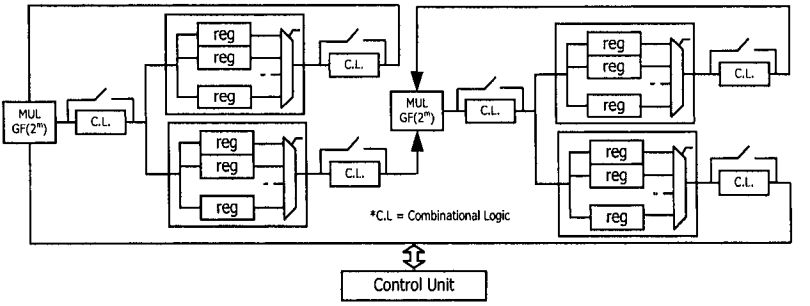

Fig. 10.2. Basic Organization of Elliptic Curve Scalar Implementation

A Control Unit is present in virtually every hardware design. Its main responsibility is to control the dataflow among the different design's modules. Design's main architecture, on the other hand, is responsible of computing all required arithmetic/logic operations. It is frequently called Arithmetic-Logic Unit (ALU).

10.5.1 Arithmetic-Logic Unit for Scalar Multiplication

Figure 10.3 shows the arithmetic-logic unit designed for computing the scalar multiplication algorithms discussed in the preceding Sections. It is a generic FPGA architecture based on the parallel-sequential approach for kP computations discussed before.

In order to implement the memory blocks of Figure 10.2, fast access FPGA's read/write memories BlockRAMs (BRAMs) were used. As it was studied in Chapter 3, a dual port BRAM can be configured as a two single port BRAMs with independent data access. This special feature allows us to save a considerable number of multiplexer operations as the required data is independently accessible from any of the two available input ports. Hence, two similar BRAMs blocks (each one composed by 12 BRAMs) provide four operands to the two multiplier blocks simultaneously. Since each BRAM contains 4k memory cells, two BRAM blocks are sufficient. The combination of 12 BRAMs provides access to a 191-bit bus length. All control signals (read/write, address signals to the BRAMs and multiplexer enable signals) are generated by the control unit (CU). A master clock is directly fed to the BRAM block which is afterwards divided by two, serving as a master clock for the rest of the circuitry. The external multiplexers apply pre and post computations (squaring, XOR, etc.) on the inputs of the multipliers whenever they are required.

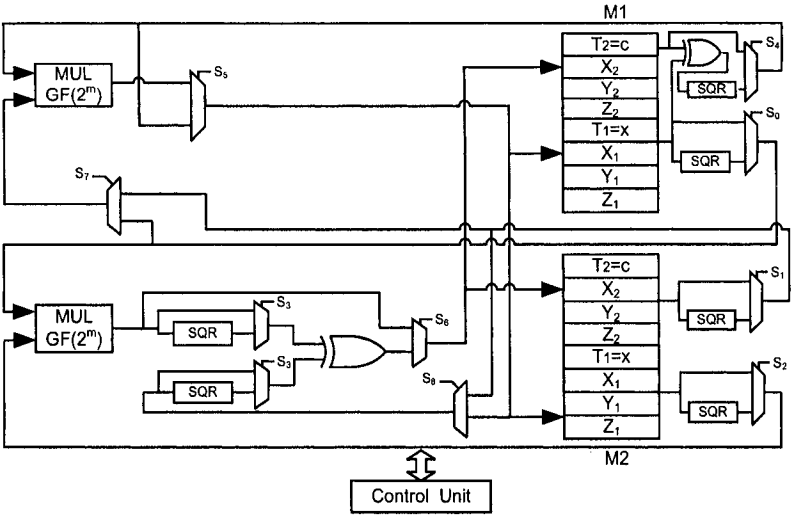

Fig. 10.3. Arithmetic-Logic Unit for Scalar Multiplication on FPGA Platforms

Let us recall that we need to perform an inversion operation in order to convert from standard projective coordinates to affine coordinates [9]. A squarer block "SqrInv" is especially included for the sole purpose of performing that inversion. As it was explained in Section 6.3.2, the Itoh-Tsujii multiplicative inverse algorithm requires the computation of m field squarings. This can be accomplished by cascading several squarer blocks so that several squaring operations can be executed within a single clock cycle (See Fig. 6.11 for more details).

In the next Subsection we discuss how the arithmetic logic unit of Figure 10.2 can be utilized for computing a Hessian scalar multiplication.

10.5.2 Scalar multiplication in Hessian Form

According to Eq. (10.3) of Section 10.2 we know that the addition of two points in Hessian form consists of 12 multiplications, 3 squarings and 3 addition operations. Implementing squaring over GF(2^m) is simple, so we can neglect it. Using the parallel architecture proposed in Figure 10.3, point addition can be performed in 6 clock cycles using two GF(2^{191}) multiplier blocks. The Hessian curve point addition sequence using two multiplier units is specified in Eq. (10.13). Table 10.2 shows that sequence in terms of read/write cycles.

[9] This conversion is required when executing a Montgomery point multiplication in Standard Projective coordinates

Referring to the architecture of Figure 10.3, $M1$ and $M2$ are two memory (BRAMs) blocks, each one composed of two independent ports $PT1$ and $PT2$.

It is noticed that the inputs/outputs of the multipliers are different from those read/write values at the memory blocks. This is due to pre or post computations required during the next clock cycle. Table 10.2 lists computed values during/after multiplications for both, the read and write cycles.

Table 10.2. Point addition in Hessian Form

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	Y_1	X_1	X_2	Y_2	λ_1	λ_2
2	X_1	Z_1	Z_2	X_2	λ_3	λ_4
3	Z_1	Z_2	Y_2	Y_1	λ_5	λ_6
4	λ_1	λ_2	λ_6	λ_5	x_3	–
5	λ_2	λ_1	λ_3	λ_4	y_3	–
6	λ_5	λ_6	λ_4	λ_3	z_3	–

Similarly, Hessian point doubling implementation of Eq. (10.13) consists of 6 multiplications, 3 squarings and 3 additions. Table 10.3 describes the algorithm flow implemented using the same architecture (Figure 10.3).

Table 10.3. Point doubling in Hessian Form

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_1	Y_1	X_1	Y_1	λ_4	λ_9
2	λ_9	λ_4	Z_1	Z_1	z_2	λ_8
3	$\lambda 8$	λ_9	Y_1	X_1	x_2	y_2

Let m represents the number of bits and M denotes a single finite field multiplication. Then the number of multiplications for one point addition and point doubling are $6M$ and $3M$, respectively. Referring to the algorithm in Figure 10.1, average of $(\frac{m}{2})6M$ and $3mM$ multiplications are needed for computing all m bits of the vector k. Thus, 6mM are the total multiplication operations required for computing kP scalar multiplication.

In the case of $m = 191$ bits, the total number of field multiplications required by the algorithm are 1146. Let T be the minimum clock period allowed by the synthesis tool. Then, $1146 \times T$ is the total time required for completing one Hessian elliptic curve scalar multiplication.

10.5.3 Montgomery Point Multiplication

Let us consider now Algorithm 10.5, where each bit of the scalar k are scanned from left to right (i.e., MSB-First).

At every iteration (regardless if the bit scanned is zero or one), both point addition ($Madd$) and point doubling ($Mdouble$) operations must be performed. However, notice that the order of the arguments is reversed: if the tested bit is '1', $Mdouble(X_2, Z_2)$, $Madd(X_1, Z_1, X_2, Z_2)$ are computed and $Mdouble(X_1, Z_1)$, $Madd(X_2, Z_2, X_1, Z_1)$ otherwise. Algorithms 10.4 and 10.3 describe the sequence of instructions for $Madd$ and $Mdouble$ operations, respectively, whereas Eqs. (10.14) and (10.13) specify how those primitives can be accomplished in 2 and 1 cycles, respectively[10].

Tables 10.4 and 10.5 describe the multiplications performed for both point addition and point doubling operations in three normal clock cycles when the scanned bit is '1' or '0' respectively. We kept the same notations used in algorithms 10.4 and 10.3 for point addition and point doubling, respectively. M1 and M2 represent two memory blocks (BRAMs) each one with two independent ports $PT1$ and $PT2$. Some required arithmetic operations (squaring etc.) need to be performed during read/write cycles at the memories before and after the multiplication operations.

Table 10.4. kP Computation, if Test-Bit is '1'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_1	Z_2	Z_1	X_2	P	Q
2	X_2	Z_2	Z_2	T_1	Z_2=Z_3	X_2=X_3
3	P	Q	Q	T_2	X_1=X'	Z_1=Z'

The resulting vectors X_1,Z_1,X_2,Z_2, are updated at the memories after the completion of point addition and doubling operations using 3 clock cycles per each bit. Therefore, the total time for the whole 191-bit scalar is $191 \times 3 \times T$, where T represents design's maximum allowed frequency.

10.5.4 Implementation Summary

All finite field arithmetic blocks and then the kP computational architecture were implemented on a VirtexE XCV3200e-8bg560 device by using Xilinx Foundation Tool F4.1i for design entry, synthesis, testing, implementation and verification of results. Table 10.6 lists timing performances and occupied resources by the said architectures.

[10] Provided that two multiplier units are available.

Table 10.5. kP Computation, If Test-Bit is '0'

Cycle	Read				Write	
	M1		M2		M1/M2	
	$PT1$	$PT2$	$PT1$	$PT2$	$PT1$	$PT2$
1	X_2	Z_1	Z_2	X_1	P	Q
2	X_1	Z_1	Z_1	T_1	$Z_1{=}Z_3$	$X_1{=}X_3$
3	P	Q	Q	T_2	$X_2{=}X'$	$Z_2{=}Z'$

Elliptic curve point addition and point doubling do not participate directly as a single computational unit in this design; however parallel computations for both point addition and point doubling are designed together as it was shown in Algorithm 10.1.

Both point addition and point doubling occupy 18300 (56.39 %) CLB slices and it takes $100.1\eta s$ (at a clock speed of 9.99 MHz) to complete one execution cycle. As it was mentioned in Section 10.2, when using two field multiplier units, six and three clock cycles are needed for computing point addition and point doubling in Hessian form, respectively.

The total consumed time for computing each iteration of the algorithm of Figure 10.1 is 900.9η if the corresponding bit is one and $300.3\eta s$ otherwise. Therefore, scalar point multiplication in Hessian form is the time needed to complete m/2 point additions (in average) and m point doublings. For our case m=191, the total time is therefore $(191/2) \cdot (600.6\eta) + 191 \cdot (300.3\eta) = 114.71\mu s$[11].

Similarly, two and one clock cycles are needed to perform Montgomery point addition and point doubling, respectively. The associated executing time is thus, $200.1\eta s$ and $100.2\eta s$ for point addition and point doubling respectively. Each iteration of the algorithm thus consumes $300.3\eta s$ for 3 clock cycles. In the case of $m = 191$, the total time needed for computing a scalar multiplication is $191(300.3) = 57\mu s$.

Inversion is performed at the end of the main loop of Algorithm 10.5. It takes 28 clock cycles to perform one inversion in $GF(2^{191})$ occupying 1312 CLB slices. The CLB slices for inversion in fact are the FPGA resources occupied for squaring operations only and the multiplier blocks are the same used for point addition and point doubling. The total conversion time (See Algorithm 10.6) is therefore $28 \cdot 100.1\eta + 10 \cdot 100.1\eta = 3.8\mu s$. Therefore, the execution time for algorithm 10.5 is given as the sum of the time for computing the scalar multiplication and the time to perform coordinate conversion namely,

$$57.36 + 3.8 = 61.16\mu s.$$

[11] It is noted that we did not include a conversion from projective to affine coordinates in the case of the Hessian form.

The architecture for elliptic curve scalar multiplication in both cases (Hessian form & Montgomery point multiplication) occupies 19626 (60 %) CLB slices, 24 (11%) BRAMs and performs at the rate of $100.1\eta s$ (9.99 MHz). The design for GF(2^{191}) Karatsuba-Ofman Multiplier occupies 8721 (26.87%) CLB slices, where one field multiplication is performed in $43.1\eta s$. Table 10.6 summarizes the design statistics.

Table 10.6. Design Implementation Summary

Design	Device (XCV)	CLB slices	Timings
Inversion in $GF(2^{191})$	3200E	1312	$2.8\eta s$
Binary Karatsuba Multiplier	3200E	8721	$43.1\eta s$
1 Field Multiplication			$100.1\eta s$
Point addition + Point doubling in Hessian Form	3200E	18300	$300.3\eta s$ (if bit = '0') $900.9\eta s$ (if bit = '1')
Point Multiplication in Hessian form	3200E	19626 & 24 BRAMs	$114.71\mu s$
Point addition + Point doubling (Montgomery Point Multiplication)	3200E	18300	$300.3\eta s$ (3 Multiplications)
Point Multiplication (Montgomery Point Multiplication)	3200E	19626 & 24 BRAMs	$61.16\mu s$

10.6 Koblitz Curves

First proposed in 1991 by N. Koblitz [180], *Koblitz Elliptic Curves* have been object of analysis and study since then, due to their superb usage of endomorphism via the Frobenius map for increasing the elliptic curve arithmetic computational performance [180, 133]. Across the years, several efforts for speeding up elliptic curve scalar multiplication on Koblitz curves have been reported both, in hardware and software platforms [13, 384, 216, 133, 132, 339, 340].

Let $P(x)$ be a degree-m polynomial, irreducible over $GF(2)$. Then $P(x)$ generates the finite field $\mathbb{F}_q = GF(2^m)$ of characteristic two. A Koblitz elliptic curve $E_a(\mathbb{F}_q)$, also known as Anomalous Binary Curve (ABC) [180], is defined as the set of points $(x, y) \in GF(2^m) \times GF(2^m)$, that satisfy the Koblitz equation,

$$E_a : y^2 + xy = x^3 + ax^2 + 1, \tag{10.15}$$

together with the point at infinity denoted by $\mathcal{O}$. It is customary to use the notation E_a where $a \in \{0, 1\}$. It is known that E_a forms an *addition Abelian group* with respect to the elliptic point addition operation[12].

[12] Notice that since Eq. (10.15) assumes $a \in \{0, 1\}$, then Koblitz curves are also defined over $GF(2)$.

So far, most works have strived for reducing the cost associated to the double-and-add method by following two main strategies: Reducing the computational complexity of both, point addition and point doubling primitives and; reducing the number of times that the point addition primitive is invoked during the algorithm execution. Recently, the idea of representing the scalar k in mixed base rather than the traditional binary form has been proposed. This way, point doublings can be partially substituted with advantage by tripling, quadrupling and even halving a point [171, 69, 12, 13, 385, 176].

In this Section we discuss yet another approach for speeding up the computational cost of scalar multiplication on Koblitz curves: the usage of parallel strategies. In concrete, we show that the usage of the τ^{-1} Frobenius operator can be successfully applied in the domain of Koblitz elliptic curves giving an extra flexibility and potential speedup to known elliptic curve scalar multiplication procedures.

The rest of this Section is organized as follows. In Subsection 10.6.1 some relevant mathematical concepts are briefly outlined. Then, in Subsection 10.6.2 several parallel formulations of the scalar multiplication on Koblitz curves are presented. Subsection 10.6.3 discusses relevant implementation aspects of the proposed parallel algorithms for hardware platforms.

10.6.1 The τ and τ^{-1} Frobenius Operators

In a field of characteristic two, the map between an element x and its square x^2 is called the Frobenius map. It can be defined on elliptic points as:

$$\tau(x, y) := (x^2, y^2).$$

Similarly, we can define the τ^{-1} Frobenius operator as,

$$\tau^{-1}(x, y) := (\sqrt{x}, \sqrt{y}).$$

In binary extension fields, the Lagrange theorem[13] dictates that $A^{2^m} = A$ for any arbitrary element $A \in GF(2^m)$, which in turn implies that for any $i \in \mathbb{Z}$, $A^{2^i} = A^{2^{i \bmod m}}$. Notice also that by applying the square root operator in both sides of Fermat little theorem identity, we obtain, $\sqrt{A} = A^{2^{-1}} = A^{2^{m-1}}$, which can be generalized as, $A^{2^{-i}} = A^{2^{m-i}}$ for $i = 0, 1, \ldots, m$.

Using above identities, it is easy to show that the Frobenius operator satisfies the properties enumerated in the next theorem.

Theorem 10.6.1 *The Frobenius operator satisfies the following properties,*

1. $\tau\tau^{-1} = \tau^{-1}\tau = 1$
2. $\tau^i = \tau^{i \bmod m}$, *for* $i \in \mathbb{Z}$
3. $\tau^{-i} = \tau^{m-i}$, *for* $i = 1, 2, \cdots, m-1$
4. $\tau^i = \tau^{-(m-i)}$, *for* $i = 1, 2, \cdots, m-1$

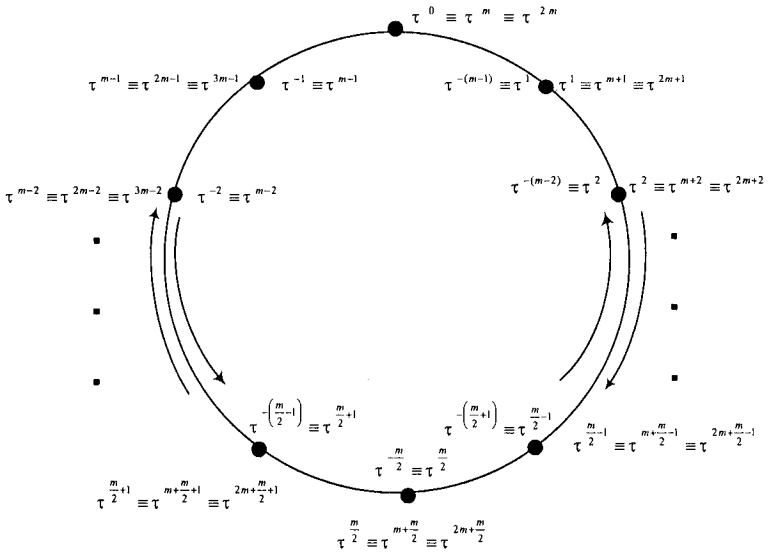

Fig. 10.4. An illustration of the τ and τ^{-1} Abelian Groups (with m an Even Number)

In other words, the τ and the τ^{-1} operators generate an *Abelian group* of order m as is depicted in Fig. 10.4. Considering an arbitrary element $A \in GF(2^m)$, with m even, Fig. 10.4 illustrates, in the clockwise direction, all the m elliptic curve points that can be generated by repeatedly computing the τ operator, i.e., $\tau^i P$ for $i = 0, 1, \cdots, m-1$. On the other hand, in the counter-clockwise direction, Fig. 10.4 illustrates all the m points that can be generated by repeatedly computing the τ^{-1} operator, i.e., $\tau^{-i} P$ for $i = 0, 1, \cdots, m-1$.

Frobenius Operator Applied on Koblitz Curves

Koblitz curves exhibit the property that, if $P = (x, y)$ is a point in E_a then so is the point (x^2, y^2) [338]. Moreover, it has been shown that, $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for every (x, y) on E_a, where $\mu = (-1)^{1-a}$. Therefore, using the Frobenius notation, we can write the relation,

$$\tau(\tau P) + 2P = (\tau^2 + 2)P = \mu\tau P. \tag{10.16}$$

Notice that last equation implies that a point doubling can be computed by applying twice the τ Frobenius operator to the point P followed by a point

[13] Lagrange theorem can be used to prove the Fermat's little theorem and its generalization Euler's theorem studied in Chapter 4

addition of the points $\mu\tau P$ and $\tau^2 P$. Let us recall that the Frobenius operator is an inexpensive operation since field squaring is a linear operation in binary extension fields.

By solving the quadratic Eq. 10.16 for τ, we can find an equivalence between a squaring map and the scalar multiplication with the complex number $\tau = \frac{-1+\sqrt{-7}}{2}$. It can be shown that any positive integer k can be reduced modulo $\tau^m - 1$. Hence, a τ-adic non-adjacent form (τNAF) of the scalar k can be produced as,

$$k = \sum_{i=0}^{l-1} u_i \tau^I,$$

where each $u_i \in \{0, \pm 1\}$ and l is the expansion's length. The scalar multiplication kP can then be computed with an equivalent non-adjacent form (NAF) addition-subtraction method.

Standard (NAF) addition-subtraction method computes a scalar multiplication in about m doubles and $m/3$ additions [129]. Likewise, the τNAF method implies the computation of l τ mappings (field squarings) and $l/3$ additions.

On the other hand, it is possible to process ω digits of the scalar k at a time. Let $\omega \geq 2$ be a positive integer. Let us define $\alpha_i = i \bmod \tau^\omega$ for $i \in [1, 3, 5, \ldots, 2^{\omega-1} - 1]$. A width-$\omega$ τNAF of a nonzero element k is an expression $k = \sum_{i=0}^{l-1} u_i \tau^i$ where each $u_i \in [0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{\omega-1}-1}]$ and $u_{l-1} \neq 0$. It is also guaranteed that at most one of any consecutive ω coefficients is nonzero. Therefore, the $\omega\tau$NAF expansion of k represents an equivalence relation between the scalar multiplication kP and the expression,

$$u_0 P + \tau u_1 P + \tau^2 u_2 P + \ldots + \tau^{l-1} u_{l-1} P \tag{10.17}$$

In [338, 337, 26] it was proved that for a Koblitz elliptic curve $E_a[GF(2^m)]$, the length l of a τNAF expansion, is always less or equal than $m + a + 3$,

$$\ell_{NAF} \leq m + a + 3$$

Using the properties enounced in Theorem 10.6.1, Equation (10.17) can be *reduced* even further whenever $l \geq m$.

Indeed, given the fact that $\tau^{m+i} = \tau^i$ for $i = 0, 1, \cdots, m-1$, we can reduce all the expansion coefficients u_i greater than m as follows,

$$k = \sum_{i=0}^{m+a+2} u_i \tau^i = \sum_{i=0}^{m-1} u_i \tau^i + \sum_{i=m}^{m+a+2} u_i \tau^i = \sum_{i=0}^{a+2} (u_i + u_{m+i})\, \tau^i + \sum_{i=a+3}^{m-1} u_i \tau^i \tag{10.18}$$

Furthermore, using property 4 of Theorem 10.6.1, it is always possible to express a length m $\omega\tau$NAF expansion in terms of the τ^{-1} operator as follows,

$$k = \sum_{i=0}^{m-1} u_i \tau^i = \left(u_0 + u_1\tau^1 + u_2\tau^2 + \ldots + u_{m-1}\tau^{m-1}\right) \tag{10.19}$$

$$= \left(u_0 + u_1\tau^{-(m-1)} + u_2\tau^{-(m-2)} + \ldots + u_{m-1}\tau^{-1}\right) = \sum_{i=0}^{m-1} u_i \tau^{-(m-i)}$$

Summarizing, Koblitz elliptic curve scalar multiplication can be accomplished by processing elliptic point additions and τ and/or τ^{-1} mappings. Hence, a Koblitz multiplication algorithm is usually divided into two main phases: a ω-TNAF expansion of the scalar k; and the scalar multiplication itself based on the τ Frobenius operator and elliptic curve addition sequences.

10.6.2 $\omega\tau$NAF Scalar Multiplication in Two Phases

Algorithm 10.7 $\omega\tau$NAF Expansion[133, 132]

Require: *Curve Parameters; representative elements:* $\alpha_u = \beta_u + \gamma_u\tau$ *for* $u = 1, 3, \ldots, 2^{w-1} - 1$;$\delta$; *Scalar* k.
Ensure: $\omega\tau NAF(k)$
1: Compute $(r_0, r_1) \leftarrow k \bmod \delta$;
2: **for** $\{i = 0;\ (r_0 \neq 0)$ OR $(r_1 \neq 0);\ i = i + 1\}$ **do**
3: **if** r_0 is odd **then**
4: $u \leftarrow r_0 + r_1 t_w \bmod 2^w$;
5: **if** $u > 0$ **then**
6: $\xi \leftarrow 1$;
7: **else**
8: $\xi \leftarrow -1$; $u \leftarrow -u$;
9: **end if**
10: $r_0 \leftarrow r_0 - \xi\beta_u$; $r_1 \leftarrow r_1 - \xi\gamma_u$; $u_i \leftarrow \xi\alpha_u$;
11: **else**
12: $u_i \leftarrow 0$;
13: **end if**
14: $(r_0, r_1) \leftarrow (r_1 + \frac{\mu r_0}{2}, \frac{-r_0}{2})$;
15: **end for**
16: $l = i$;
17: **Return** $l, (u_{l-1}, u_{l-2}, \cdots, u_1, u_0)$;

Algorithms 10.7 and 10.8 show the adaptations of Solinas procedures as they were reported in [132, 133].

It should be noticed that Algorithm 10.7 produces the $\omega\tau$NAF expansion coefficients from right to left, i.e., the least significant coefficient u_0 is first produced, then u_1 and so on, until the most significant coefficient, namely, u_{l-1}, is obtained. Algorithm 10.8 on the contrary, computes the expression 10.17 from left to right, i.e., it starts processing u_{l-1} first, then u_{l-2} until it ends with the coefficient u_0.

Algorithm 10.8 $\omega\tau$NAF Scalar Multiplication [133, 132]

Require: $\omega\tau NAF(k) = \sum_{i=0}^{l-1} u_i \tau^i$, $P \in E_a(F_{2^m})$.
Ensure: kP
1: Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
2: $Q \leftarrow \mathcal{O}$;
3: **for** i from $l - 1$ downto 0 **do**
4: $Q \leftarrow \tau Q$;
5: **if** $u_i \neq 0$ **then**
6: Find u such that $\alpha_u = \pm u_i$;
7: **if** $u > 0$ **then**
8: $Q \leftarrow Q + P_u$;
9: **else**
10: $Q \leftarrow Q - P_{-u}$;
11: **end if**
12: **end if**
13: **end for**
14: **Return** Q;

The combination of those two characteristics is unfortunate as it forces us to work in a strictly sequential manner: First Algorithm 10.7 must be executed and only when it finishes, Algorithm 10.8 can start the computation of the Koblitz curve scalar multiplication operation. However, invoking Eq. (10.19), we can formulate a parallel version of Algorithm 10.8 as is shown in Algorithm 10.9. If two separated point addition units are available, the expected computational speedup of the parallel version in Algorithm 10.9 is of about 50 % when compared with its sequential version.

10.6.3 Hardware Implementation Considerations

In an effort to minimize the number of clock cycles required by Algorithm 10.8 when implemented in a hardware platform, we first proceed to pre-process the width-$\omega\tau$NAF expansion of coefficient k as described below.

Firstly, without loss of generality we will assume that the length of the expansion is m[14]. Secondly, let us recall that it is guaranteed that at most one of any consecutive ω coefficients of an $\omega\tau$NAF expansion is nonzero. Let $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ denote each one of the up to $N_w = \lceil \frac{m}{\omega+1} \rceil$ nonzero $\omega\tau NAF$ expansion coefficients. Then, the expansion would have the following structure:

$$w_0, 0 \ldots 0, w_1, 0 \ldots 0, w_2, 0, \ldots, 0, w_{i-1}, 0 \ldots 0, w_{N_w - 1}$$

Above runs of up to $2w - 2$ consecutive *zeroes* [340], can be counted and stored. Let $z_i \in [\omega - 1, 2\omega - 2]$ denote the length of each of the at most

[14] Otherwise, if $l > m$, we can use Eq. (10.18) in order to reduce the expansion length back to m.

Algorithm 10.9 $\omega\tau$NAF Scalar Multiplication: Parallel Version

Require: $\omega\tau NAF(k) = \sum_{i=0}^{m-1} u_i \tau^i$, $P \in E_a(F_{2^m})$.
Ensure: kP

PreCompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
$Q = R = \mathcal{O}$;
$N = \lfloor \frac{m}{2} \rfloor$; $u_m = 0$;
for i from N downto 0 **do**
  $Q \leftarrow \tau Q$;
  if $u_i \neq 0$ **then**
    Find u such that $\alpha_{\pm u} = \pm u_i$;
    if $u > 0$ **then**
      $Q \leftarrow Q + P_u$;
    else
      $Q \leftarrow Q - P_{-u}$;
    end if
  end if
end for
$Q \leftarrow Q + R$;
Return Q;

for $j = N + 1$ **to** m **do**
  $R \leftarrow \tau^{-1} R$;
  if $u_j \neq 0$ **then**
    Find u such that $\alpha_{\pm u} = \pm u_j$;
    if $u > 0$ **then**
      $R \leftarrow R + P_u$;
    else
      $R \leftarrow R - P_{-u}$;
    end if
  end if
end for

Algorithm 10.10 $\omega\tau$NAF Scalar Multiplication: Hardware Version

Require: $\tau NAF_\omega(k)$ in the format: $w_0, z_1, w_2, z_3, \ldots, z_{N_w-2}, w_{N_w-1}$, $N_w = 2\lceil \frac{m}{w+1} \rceil$. Where $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ and $z_i \in [w - 1, 2w - 2]$
Ensure: kP

Precompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
$Q \leftarrow \mathcal{O}$
for i from $N - 1$ downto 0 **do**
  if i is odd **then** {/*processing a zero coefficient z_i*/}
    $Q \leftarrow \tau^{w-1} Q$
    $z_i \leftarrow z_i - (w - 1)$
    if $z_i \neq 0$ **then**
      $Q \leftarrow \tau^{z_i} Q$
    end if
  else {/*processing a nonzero coefficient w_i*/}
    Find u such that $\alpha_u = \pm \omega_i$;
    if $u > 0$ **then**
      $Q \leftarrow Q + P_u$;
    else
      $Q \leftarrow Q - P_{-u}$;
    end if
  end if
end for
Return Q;

$N_w = \lfloor \frac{m}{\omega+1} \rfloor$ *zero* runs. Then, the proposed compact version of the expansion has the following form,

$$w_0, z_0, w_1, z_2, \ldots, z_{N_w-1}, w_{N_w-1} \tag{10.20}$$

In this new format we just need to store in memory at most $2\lceil \frac{(m)}{\omega+1} \rceil$ expansion coefficients. Algorithm 10.10 shows how to take advantage of the compact representation just described. Given the relatively cheap cost of the field squaring operation, steps 5-8 of Algorithm 10.10 can compute up to $\omega-1$ applications of the τ Frobenius operator[15]. This will render a valuable saving of system clock cycles. Moreover, using the same idea already employed in Algorithm 10.9, we can parallelize Algorithm 10.10 using the τ and τ^{-1} operators concurrently. The resulting procedure is shown in Algorithm 10.11.

Algorithm 10.11 $\omega\tau$NAF Scalar Multiplication: Parallel HW Version

Require: $\tau NAF_\omega(k)$ in the format: $w_0, z_1, w_2, z_3, \ldots, z_{N_w-2}, w_{N_w-1}$, $N_w = 2\lceil \frac{m}{w+1} \rceil$. Where $w_i \in [1, 3, 5, \ldots, 2^{w-1} - 1]$ and $z_i \in [w-1, 2w-2]$
Ensure: kP

1: PreCompute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, ..., 2^{w-1} - 1\}$ where $\alpha_i = i \bmod \tau^w$ for $i \in \{1, 3, ..., 2^{w-1} - 1\}$;
2: $Q = R = \mathcal{O}$;
3: $N = \lfloor \frac{N_w}{2} \rfloor$;
4: **for** i from N downto 0 **do**
5: **if** i is odd **then**
6: $Q \leftarrow \tau^{\omega-1} Q$;
7: $z_i \leftarrow z_i - (w-1)$;
8: **if** $z_i \neq 0$ **then**
9: $Q \leftarrow \tau^{z_i} Q$;
10: **end if**
11: **else**
12: Find u such that $\alpha_{\pm u} = \pm u_i$;
13: **if** $u > 0$ **then**
14: $Q \leftarrow Q + P_u$;
15: **else**
16: $Q \leftarrow Q - P_{-u}$;
17: **end if**
18: **end if**
19: **end for**
20: $Q \leftarrow Q + R$;
21: **Return** Q;

for $j = N + 1$ **to** m **do**
if i is odd **then**
$R \leftarrow \tau^{-(\omega-1)} R$;
$z_j \leftarrow z_j - (w-1)$;
if $z_i \neq 0$ **then**
$R \leftarrow \tau^{z_j} R$;
end if
else
Find u such that $\alpha_{\pm u} = \pm u_j$;
if $u > 0$ **then**
$R \leftarrow R + P_u$;
else
$R \leftarrow R - P_{-u}$;
end if
end if
end for

[15] Let us recall that applying i times the τ Frobenius operator over an elliptic point Q consists of squaring each coordinate of Q i times. See §6.2 for details about how to compute efficiently squaring and other field arithmetic operations

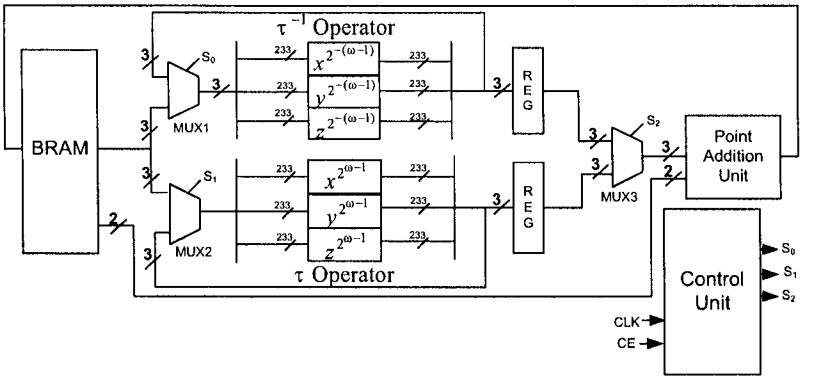

Fig. 10.5. A Hardware Architecture for Scalar Multiplication on the NIST Koblitz Curve K-233

Proposed Hardware Architecture

According to Algorithm 10.11, one can accomplish a scalar multiplication operation by computing two sequences, namely, τ operator-then-add and; τ^{-1} operator-then-add. Both sequences are independent and therefore, they can be processed concurrently provided that hardware resources meet up design requirements. An aggressive approach would be to use two point addition units with τ and τ^{-1} blocks operating separately. That, however, could be unaffordable as the point addition block consumes a vast amount of hardware resources. A more conservative approach consisting of a single point addition unit is shown in Fig. 10.5. The main idea used there is to keep the τ and τ^{-1} computations in parallel while a multiplexer block allows the control unit to decide which result will be processed next by the point addition unit. Intermediate results required for next stages of the algorithm are read/written in a Block select RAM (BRAM).

The inputs/output of the point addition unit read/write data from/to the BRAM block according to an address scheme orchestrated by the control unit. Data paths for the τ and τ^{-1} operators and then point addition are adjusted by providing selection bits for the three multiplexers MUX1, MUX2, and MUX3. Notice that all three multiplexers handle three 233-bit inputs/outputs. This is the required size for a three-coordinate LD projective point as it was described in Subsection 4.5.2. The τ and τ^{-1} operators were designed using the formulae described in §6.2. The Point Addition Unit (PAU) performs the point addition operation using the LD-affine mixed coordinates algorithm to be explained in the next Section. PAU has two inputs. One input comes from (via MUX3) the output of either τ or τ^{-1} blocks in the form of a three-coordinate LD projective point. The other input comes directly from the BRAM block and corresponds to one of the pre-computed multiples of P, namely, $P_{u_i} =$

$\alpha_{u_i} P$. Those multiples have been pre-computed in affine coordinates. A 4- bit counter and a ROM constitute the control unit block. The ROM block is filled with *control words*, which are used at each clock cycle for the orchestration and synchronization of algorithm's dataflow. The ROM block address bits are timely incremented by a 4-bit counter. A total of 11 bits (8 bits for each port of the BRAM, 1 bit for MUX1, 1 bit for MUX2 and 1 bit for MUX3) are used for controlling and synchronizing the whole circuitry. The 11-bit control word for each clock cycle is filled in the BRAM block, and then they are extracted at the rising edge of each clock cycle.

The expected performance of the architecture shown in Fig. 10.5 can be estimated as follows. As it has been mentioned, in a $\omega\tau$ NAF expansion there exists a total of $N_w = \lceil \frac{m}{\omega+1} \rceil$ nonzero coefficients. Let ξ be the number of cycles required for computing an elliptic point addition operation. Knowing that the Frobenius operators depicted in Fig. 10.5 are each able to compute $\omega - 1$ τ or τ^{-1} operators in one cycle, it seems fair to say that our architecture can process a coefficient zero in $\frac{1}{\omega-1}$ cycles. Therefore, the total number of system clock cycles required by Algorithm 10.10 for computing a scalar multiplication can be estimated as,

$$\#\text{Number of Clock Cycles} = \xi \frac{m}{\omega+1} + \frac{1}{\omega-1}\frac{\omega m}{\omega+1}. \tag{10.21}$$

In the case of Algorithm 10.11 since the τ and τ^{-1} operations are computed at the same time that the point addition processing is taking place, the total number of clock cycles can be estimated as just,

$$\#\text{Number of Clock Cycles} = \xi \frac{m}{\omega+1}. \tag{10.22}$$

As a way of illustration, let us assume that the architecture shown in Fig. 10.5 has been implemented using the arithmetic building blocks for the NIST recommended K-233 Koblitz curve. Then using $m = 233$ and $\xi = 8$ and equations (10.21) and (10.22), a saving of 14.28%,13.51% and 13.04% can be obtained when using $\omega = 4, 5, 6$, respectively.

10.7 Half-and-Add Algorithm for Scalar Multiplication

Schroeppel [322] and Knudsen [176] independently proposed in 1999 a method to speedup scalar multiplication on elliptic curves defined over binary extension fields. Their method is based on a novel elliptic curve primitive called *point halving*, which can be defined as follows.

Given a point Q of odd order, compute P such that $Q = 2P$. The point P is denoted as $\frac{1}{2}Q$. Since theoretically, point halving is up to three times as fast as point doubling, it is possible to improve the performance of scalar multiplication computation $Q = nP$ by replacing the double-and-add algorithm

with a half-and-add method based on an expansion of the scalar n in terms of negative powers of 2.

As it was discussed in Chapter 2, the efficiency of ECDSA depends on the arithmetic involving the points of the curve. For this reason it becomes necessary to implement efficient curve operations in order to obtain high performances. In this Section we describe an architecture that employs a parallelized version of the half-and-add method and its associated building blocks.

The rest of this Section is organized as follows. Subsection 10.7.1, describes the algorithms utilized for implementing elliptic curve arithmetic. In Subsection 10.7.2, the proposed hardware architecture is explained in detail.

10.7.1 Efficient Elliptic Curve Arithmetic

With the help of the arithmetic operators described in Chapter 6, we can efficiently construct the three main elliptic curve operations, namely, point addition, point doubling and point halving.

As a means of avoiding the expensive field inversion operation, it results convenient to work with *López-Dahab (LD)* projective coordinates[16]. For convenience, here we will repeat some of the main characteristics of those coordinates.

In LD projective coordinates, the projective point (X:Y:Z) with Z$\neq$ 0 corresponds to the affine coordinates $x = X/Z$ and $y = Y/Z^2$. The elliptic curve Equation (10.6) mapped to LD projective coordinates is given as,

$$Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4 \tag{10.23}$$

The point at infinity is represented as $\mathcal{O} = (1 : 0 : 0)$. Let $P = (X_1 : Y_1 : Z_1)$ and $Q = (X_2 : Y_2 : 1)$ be an arbitrary point belonging to the curve 4.19. Then the point $-P = (X_1 : X_1 + Y_1 : Z)$ is the addition inverse of the point P.

Point Doubling

The point doubling primitive $2(X_1 : Y_1 : Z_1) = (X_3 : Y_3 : Z_3)$ can be performed as,

$$\begin{aligned} &Z_3 = {X_1}^2 \cdot {Z_1}^2; X_3 = {X_1}^4 + b \cdot {Z_1}^4; \\ &Y_3 = b{Z_1}^4 Z_3 + X_3 \cdot (aZ_3 + {Y_1}^2 + b{Z_1}^4) \end{aligned} \tag{10.24}$$

Assuming that only one field multiplier block is available, it is possible to compute above Equations in just three clock cycles as shown in Table 10.7.

[16] LD projective coordinates were already studied in Section 4.5.

Table 10.7. Parallel López-Dahab Point Doubling Algorithm

A Parallel approach of point doubling, LD-affine coordinates.		
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates		
on $E/K : y^2 + xy = x^3 + ax^2 + b, a \in \{0, 1\}$.		
Output: $2P = (X_3 : Y_3 : Z_3)$ in LD coordinates		
# cycle	C_0	C_1
1. cycle:	$Z_3 = X_1^2 \cdot Z_1^2$	$T_1 = b \cdot Z_1^4$
2. cycle:	$T_2 = (X_1^4 + T_1) \cdot (Z_3 + Y_1^2 + T_1)$	$X_3 = X_1^4 + T_1$
3: cycle:	$Y_3 = T_1 \cdot Z_3 + T_2$	

Point Addition

If $Q \neq -P$, the point addition primitive $(X_1 : Y_1 : Z_1) + (X_2 : Y_2) = (X_3 : Y_3 : Z_3)$ can be performed at a computational cost of 8 field multiplications as,

$$\begin{array}{ll} A = Y_2 \cdot Z_1^2 + Y_1; & B = X_2 \cdot Z_1 + X_1; \\ C = Z_1 \cdot B; & D = B^2 \cdot (C + aZ_1^2); \\ Z_3 = C^2; & E = A \cdot C; \\ X_3 = A^2 + D + E; & F = X_3 + X_2 \cdot Z_3; \\ G = (X_2 + Y_2) \cdot Z_3^2; & Y_3 = (E + Z_3) \cdot F + G \end{array} \tag{10.25}$$

Table 10.8. Parallel López-Dahab Point Addition Algorithm

A parallel approach of point addition, LD-affine coordinates.		
Input: $P = (X_1 : Y_1 : Z_1)$ in LD coordinates,		
$Q = (x_2, y_2)$ in affine coordinates		
on $E/K : y^2 + xy = x^3 + ax^2 + b$.		
Output: $P + Q = (X_3 : Y_3 : Z_3)$ in LD coordinates		
# cycle	C_0	C_1
1. cycle:	$Y_3 = y_2 \cdot Z_1^2 + Y_1$	
2. cycle:	$X_3 = x_2 \cdot Z_1 + X_1$	
3. cycle:	$T_1 = X_3 \cdot Z_1$	
4. cycle:	$X_3 = X_3^2 \cdot (a \cdot Z_1^2 + T_1)$	$Z_3 = T_1^2$
5. cycle:	$X_3 = Y_3 \cdot T_1 + X_3 + Y_3^2$	$T_1 = Y_3 \cdot T1$
6. cycle:	$T_1 = x_2 \cdot Z_3 + X_3$	
7. cycle:	$Y_3 = (x_2 + y_2) \cdot Z_3^2$	$T_2 = T_3$
8. cycle:	$Y_3 = (T_2 + Z_3) \cdot T_1 + Y_3$	

Once again, we point out that field multiplication is by far the most time consuming arithmetic operation. Field addition can be time neglected in a hardware implementation.

Therefore we can parallelize some operations in such a way that we can perform two operations at a time. As it is shown in Table 10.8, by rearranging the set of Equations 10.25 we can manage for computing a point addition operation in LD projective coordinates in just eight clock cycles.

Point Halving

Point halving can be seen as the reverse operation of point doubling [96]. We can define the elliptic curve point halving as follows. Let $Q = (x_2, y_2)$ be an arbitrary point that belongs to the curve of Eq. (10.6). Our problem in hand is to find a second point $P = (x_1, y_1)$, such that $Q = 2P$: This can be accomplished by solving the following set of equations,

$$\begin{aligned} \lambda^2 + \lambda &= x_2 + a \\ x_1 &= \sqrt{y_2 + x_2(\lambda + 1)} \\ y_1 &= \lambda x_1 + x_1^2 \end{aligned}$$

Algorithm 10.12 Point Halving Algorithm

Require: $2P = (x_2, y_2)$
Ensure: $P = (x_1, y_1)$
1: Solve $\lambda^2 + \lambda = x_2 + a$ for λ.
2: $t = y_2 + x_2 \cdot \lambda$;
3: **if** $Tr(t) = 0$ **then**
4: $x_1 = \sqrt{t + x_2}$;
5: **else**
6: $\lambda = \lambda + 1; x_1 = \sqrt{t}$;
7: **end if**
8: $y_1 = \lambda \cdot x_1 + x_1^2$;
9: **Return** (x_1, y_1)

Algorithm 10.12 was proposed in [96] for computing an elliptic point halving. However, it results more convenient in practice to define the *λ-representation* of a point as follows. Given $Q = (x, y) \in E(GF(2^m))$, let us define (x, λ_Q), where

$$\lambda_Q = x + \frac{y}{x}$$

Given the λ-representation of Q, we may compute a point halving without converting back to affine coordinates. In this way, repeated halvings can be performed directly on λ-representation.

Half-and-Add Scalar Multiplication Algorithm

In Chapter 6 several algorithms addressing the problem of how to perform efficient finite field arithmetic were studied. Notice that Algorithm 10.12 requires the following $GF(2^m)$ arithmetic main building blocks,

1. Computing field square root (studied in §6.2).
2. Computing the trace (studied in §6.4.1).
3. Solving quadratic equations (studied in §6.4.2).

Above operations constitute the building blocks for performing elliptic curve scalar multiplication using the half-and-add method shown in Algorithms 10.12 and 10.13.

Algorithm 10.13 Half-and-Add LSB-First Point Multiplication Algorithm

Require: $P \in E(GF(2^m))$, $k = k'_0/2^{m-1} + \cdots + k'_{m-1} + 2k'_m \bmod n$, with $k_i \in \{-1, 0, 1\}$ for $i =, 1, \ldots, m$.
Ensure: kP
1: $Q = \mathcal{O}$;
2: **if** $k'_m = 1$ **then**
3: $Q = 2P$;
4: **end if**
5: **for** i from $m - 1$ downto 0 **do**
6: **if** $k'_i > 0$ **then**
7: $Q = Q + P$;
8: **else if** $k'_i < 0$ **then**
9: $Q = Q - P$;
10: **end if**
11: $P = P/2$;
12: **end for**
13: **Return** (Q)

10.7.2 Implementation

The proposed architecture for achieving elliptic curve scalar multiplication is shown in Figure 10.6. The architecture consists of two main units, namely, an Arithmetic Logic Unit (ALU) block (responsible of performing field arithmetic and elliptic curve arithmetic), and a control unit (that manages and controls the dataflow of the whole circuit).

Control Unit

Table 10.9 shows the operations that can be performed by the circuit per clock cycle. In the first column the operations that the ALU can perform are listed. The first eight rows specify the sequence of operations needed for computing an elliptic curve point addition. The next three rows specify the operations needed for computing a point doubling primitive. The last three rows show the necessary operations for computing a point halving (either in λ-representation or in affine coordinates).

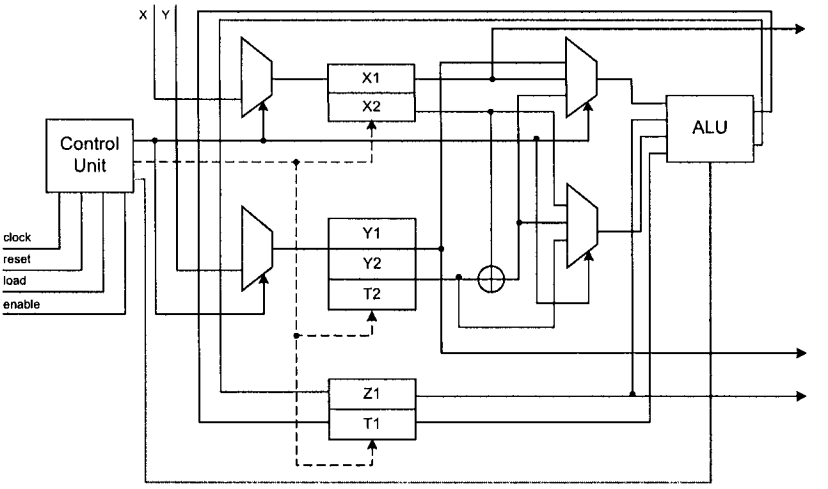

Fig. 10.6. Point Halving Scalar Multiplication Architecture

The second column represents the inputs given to the ALU circuit, whereas the fourth column shows the ALU circuit output being written to memory.

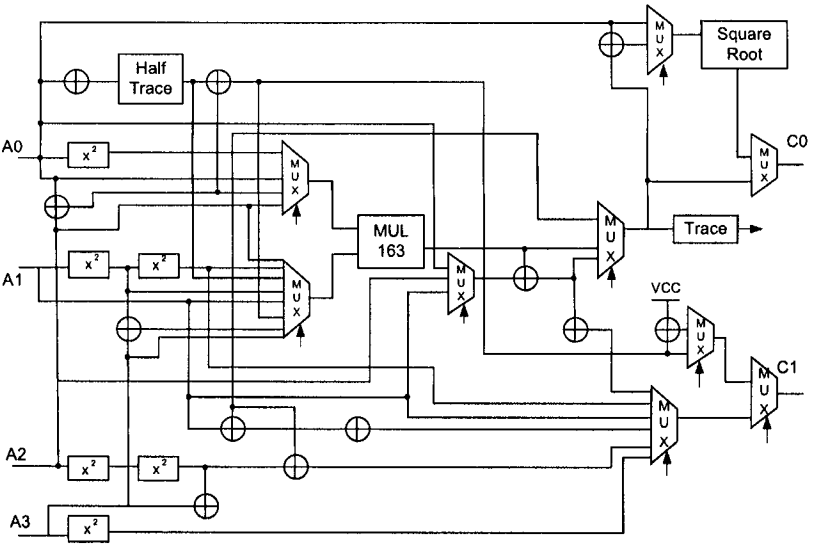

Fig. 10.7. Point Halving Arithmetic Logic Unit

Finally, the third column includes a twenty-six bit control word that stipulates which parts of the Arithmetic Logic Unit must be activated by the Control Unit. The control word format is explained below.

Table 10.9. Operations Supported by the ALU Module

operation	input	control word	output
	$a_0a_1a_2a_3$	$s_{25}\cdots s_0$	c_0c_1
$Y_1 = y_2 \cdot Z_1^2 + Y_1$	$y_2Z_1Y_1-$	1xx01000xx11010000110xxx1x	Y_1x
$X_1 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	X_1x
$T_1 = X_1 \cdot Z_1$	X_1Z_1--	10xxxxx0x0xx01001xx00xxx1x	T_1x
$X_1 = X_1^2 \cdot (Z_1^2 + T_1)$	$X_1Z_1-T_1$	00xxxxx010xx00100xx0000111	X_1Z_1
$X_1 = Y_1 \cdot T_1 + X_1 + Y_1^2$	$y_2Z_1Y_1-$	0xx01000xx11010000110xxx1x	T_1X_1
$T_2 = x_2 \cdot Z_1 + X_1$	$x_2Z_1X_1-$	110xxxx0xx00010010110xxx1x	T_2x
$Y_1 = (x_2 + y_2) \cdot Z_1^2$	$x_2Z_1y_2-$	01xxx010xx0111000xx00xxx1x	Y_1x
$Y_1 = (T_1 + Z_1) \cdot T_2 + Y_1$	$Y_1T_1T_2Z_1$	0xx0010x1011100110010xxx1x	Y_1x
$Z_1 = X_1^2 \cdot Z_1^2$	X_1Z_1--	00xxxxx0x0xx00000xx0000011	Z_1T_2
$X_1 = (X_1^4 + T_1) \cdot (Y_1^2 + Z_1 + T1)$	$Y_1Z_1X_1T_1$	0x010xxxx10xxxxxxxx0101011	T_2X_1
$Y_1 = Z_1 \cdot T_1 + T_2$	$T_2Z_1-T_1$	00xxx101xx01010010110xxx1x	Y_1x
Point Halving (affines)	x_2-y_2-	101xxx01xx01011010110xxx00	x_2y_2
Point Halving (λ-representation)	x_2-y_2-	101xxx01xx0101110xx00xxx00	$x_2\lambda$
$y_2 = \lambda x_2 + x_2^2$	x_2-y_2-	101xxx01xx01010011010xxx1x	$-y_2$

Each control word consists of a string of 26 bits organized as follows:

$$\underbrace{XX001010}_{direction}\ \underbrace{1100}_{MUX}\ \underbrace{100110010XXX1X}_{ALU}$$

The first eight bits designate the addresses to be read by the memory block, the next four bits designate which operand will be loaded to the ALU unit, and finally the last fourteen bits designate which operations will be performed by the ALU unit according to the list of supported operations shown in Table 10.9.

As an example, consider point halving computation in affine coordinates of Algorithm 10.12. The datapath for this computation is illustrated in Fig. 10.8. First, it is necessary to load x_2, y_2 into the input registers A_0, A_2, respectively. Additionally, a copy of x_2 is stored in A_1. Then, the operations for loading $HT(A_0 + 1)$ and A_1 on the finite field multiplier are commanded by the Control Unit. Next, we multiply $A_1 \cdot HT(A_0 + 1)$ and immediately after A_2 is added to that product obtaining $A_2 + A_1 \cdot HT(A_0 + 1)$. Thereafter, the result obtained by the multiplication operation is computed into the trace unit, in order to choose the appropriate operand for the square-root unit, and to send the corresponding outputs C_0, C_1. The dataflow just described is highlighted in Figure 10.8.

As mentioned previously, our architecture allows us to perform three main elliptic curve operations, namely, point addition, point doubling and point

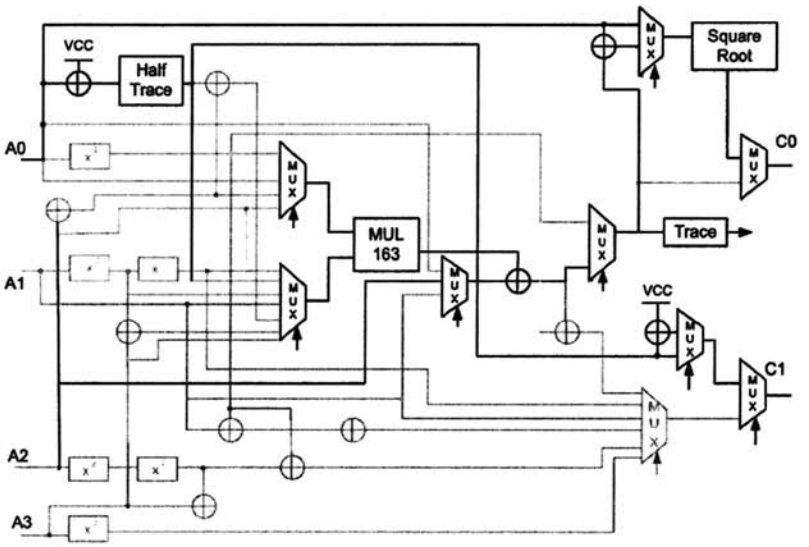

Fig. 10.8. Point Halving Execution

halving, Table 10.10 lists the number of cycles required in order to perform such operations. Furthermore, Figures 10.9 and 10.10 show the time diagram corresponding to the execution of the point addition and point doubling primitives, respectively.

Table 10.10. Cycles per Operation

Elliptic curve operations	# cycles
Point Halving (affine coordinates)	1
Point Halving (λ-representation)	2
Point Doubling	3
Point Addition	8

10.7.3 Performance Estimation

We estimate the running time of the circuit of Fig. 10.6 as follows. We need eight cycles and one cycle for performing a Point Addition (PA) in mixed LD coordinates and a Point Halving (PH) operation, respectively. On the other hand, the computational cost of Algorithm 10.13 is approximately,

$$\frac{m}{3}PA + mPH.$$

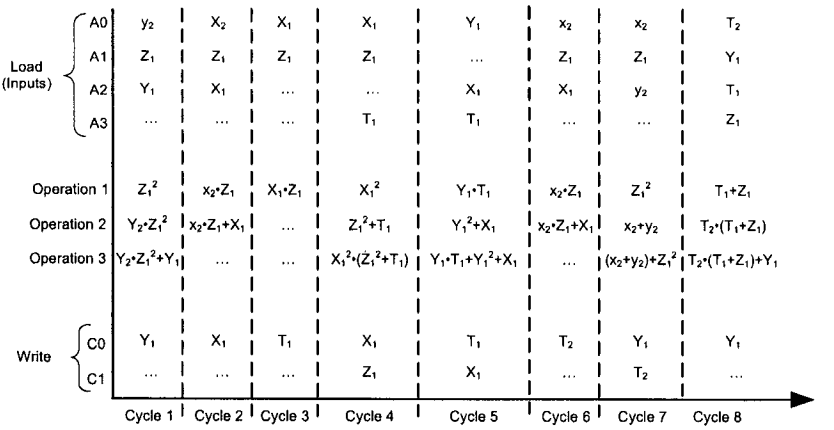

Fig. 10.9. Point Addition Execution

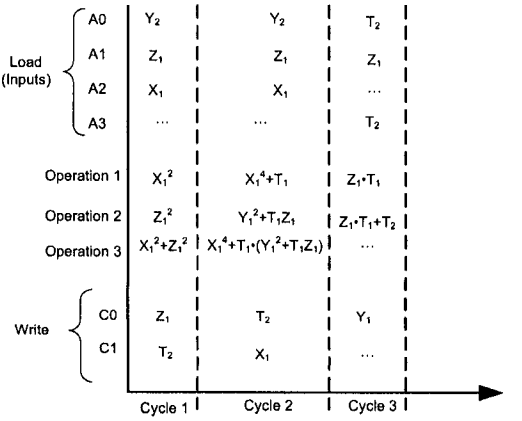

Fig. 10.10. Point Doubling Execution

Translating above equation to clock cycles, we get,

$$\frac{m}{3}(8) + mPH(1) = \frac{11}{3}m \text{ Clock Cycles.}$$

In other words, the architecture presented in this Section (see Figures 10.6 and 10.7) needs approximately $\frac{11}{3}m$ clock cycles for performing an elliptic curve point multiplication using the Half-and-Add Algorithm 10.13.

Table 10.11. Fastest Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (μS)	Cost LUTs	$\frac{m}{T \cdot LUT}$
Cruz-A. et al.[54]	2006	Virtex II	233	27.58	17.64	39762(11)	332.19
Hernández-R et al.[137]	2005	Virtex II	163	23.94	25.0	22665	287.67
Cheung et al. [50]	2005	Virtex 4	113	65	30	13922 (est)	270.55
Shu et al.[329]	2005	Virtex II	163	68.9	48	25763	131.81
Saqib et al.[310]	2006	Virtex II	191	9.99	61.16	39252(24)	79.56
Lutz [216]	2004	Virtex II	163	66.0	75	10017	216.95
Jarvinen et al.[155]	2004	Virtex II	163	90.2	106	36158(est)	42.53
Gura et al. [125]	2002	Virtex II	163	66.4	143	22665	36.14
Satoh et al. [313]	2003	0.13μm CMOS	160	510.2	190	–	–
Orlando et al.[261]	2000	Virtex	167	76.7	210	3002	265.03
Bednara et al. [20]	2002	Virtex	191	50	270	–	–
Sozzani et al. [341]	2005	0.13μm CMOS	163	417	270	–	–
Ernst et al. [313]	2002	Atmel	113	12	1400	–	–
Schroeppel et al. [322]	2003	0.13μm CMOS	178	227	4400	143K gates	–

10.8 Performance Comparison

In this Section we compare some of the most representative elliptic curve designs reported during this decade. In our survey we considered three metrics: speed, compactness and efficiency. Our study tries to sum up the state-of-the-art of scalar multiplication hardware implementations.

Table 10.11 shows the fastest designs reported to date for elliptic scalar multiplication over $GF(2^m)$[17]. It can be observed that the design of [54] which features a specialized design on Koblitz curves shows the highest speed of all designs considered.

Table 10.12. Most Compact Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (mS)	Cost	$\frac{m}{T \cdot Gates}$
Kim et al. [172]	2002	0.35μm CMOS	192 binary	10	36.2 (est)	16.84K gates	0.315
Öztürk et al. [265]	2004	0.13μm CMOS	167 prime	20	31.9	30.3K gates	0.1727
			167 prime	200	3.1	34.4K gates	1.56
Aigner et al. [2]	2004	0.13μm CMOS	191 binary	10	46.9	25K NANDs	0.163
Schroeppel et al. [322]	2003	0.13μm CMOS	178 binary	227	4.4	143K gates	0.283
Shuhua et al. [330]	2005	Virtex II	192 prime	50	6	4729 LUTs	–

[17] Whenever the number of LUTs utilized by the design is not available, an estimation based on the reported number of CLBs has been made. The number in parenthesis in the seventh column represents the total number of BRAMs.

In Table 6.4 we show a selection of some of the most compact reconfigurable hardware elliptic curve designs reported to date. It is noted that this category is dominated by those designs implemented in VLSI working with elliptic curves defined over $GF(2^m)$. Indeed, the most compact $GF(P)$ elliptic curve design in [265] has a hardware cost 1.8 times greater than that of the smallest $GF(2^m)$ elliptic curve design in [172].

We measure efficiency by taking the ratio of number of bits processed over slices multiplied by the time delay achieved by the design, namely,

$$\frac{bits}{Slices \times timings}$$

For instance, consider the Koblitz design presented in [54]. As is shown in Table 10.11, working over $GF(2^{233})$, that design achieved a time delay of just 17.64μS at a cost of 39762 Look Up Tables (LUTs) and 11 Block RAMs. Therefore its efficiency is calculated as,

$$\frac{bits}{Slices \times timings} = \frac{233}{39762 \times 17.64\mu} = 332.19$$

When comparing the designs featured in Tables 10.11 and 10.13, it is noticed that the fastest and most efficient multiplier designs are the Koblitz elliptic curve designs as well as the half-and-add scalar multiplication design studied in this Chapter.

Table 10.13. Most Efficient Elliptic Curve Scalar Multiplication Hardware Designs

Author	year	platform	m	clock MHz	time (μS)	Cost LUTs	$\frac{m}{T \cdot LUT}$
Cruz-A. et al.[54]	2006	Virtex II	233	27.58	17.64	39762(11)	332.19
Hernández-R et al.[137]	2005	Virtex II	163	23.94	25.0	22665	287.67
Cheung et al. [50]	2005	Virtex 4	113	65	30	13922 (est)	270.55
			163	35	50	20047 (est)	162.61
Orlando et al.[261]	2000	Virtex	167	76.7	210	3002	265.03
Lutz [216]	2004	Virtex II	163	66.0	75	10017	216.95
Shu et al.[329]	2005	Virtex II	163	68.9	48	25763	131.81
			233	67.9	89	35800	73.13
Saqib et al.[310]	2006	Virtex II	191	9.99	61.16	39252(24)	79.56
			191	9.99	114.71	39252(24)	42.41
Jarvinen et al.[155]	2004	Virtex II	163	90.2	106	36158(est)	42.53
			193	90.2	139	38500(est)	36.06
			233	73.6	227	46040(est)	22.29
Gura et al. [125]	2002	Virtex II	163	66.4	143	22665	36.14
Leung et al. [205]	2002	Virtex	113	31	750	17506	8.61

10.9 Conclusions

Two major factors contribute for achieving high performances in the architectures presented throughout this chapter. Firstly, the usage of parallel strategies applied at every stage of the design. Secondly, efficient elliptic curve algorithms such as the Montgomery point multiplication, scalar multiplication on Koblitz curves, the half-and-add method, etc, along with their efficient implementations on reconfigurable hardware. Furthermore, it resulted also crucial to take advantage of the lower-grained characteristic of reconfigurable hardware devices and their associated functionality (in the form of BRAMs and other resources).

In §10.5 we studied a generic architecture able to compute the scalar multiplication in Hessian form as well as the Montgomery point multiplication algorithm. It is noticed that theoretically (see Table 10.1), the Weierstrass form utilizing the Montgomery point multiplication formulation can be computed in about half the execution time consumed by the Hessian form. This prediction was confirmed in practice in [310] for elliptic curves defined over $GF(2^{191})$, as is shown in Table 10.13.

Then, we presented in §10.6 parallel formulations of the scalar multiplication operation on Koblitz curves. The main idea proposed in that Section consisted on the concurrent usage of the τ and τ^{-1} Frobenius operators, which allowed us to parallelize the computation of scalar multiplication on elliptic curves. On the other hand, we described a compact format of the $\omega\tau$NAF expansion which was especially tailored for hardware implementations. In this new format at most $2\lceil \frac{(m)}{\omega+1} \rceil$ expansion coefficients need to be stored and processed, provided that the arithmetic unit can compute up to $\omega - 1$ subsequent applications of the τ Frobenius operator in one single clock cycle. Furthermore, it was shown that by using as building blocks the τ and τ^{-1} Frobenius operators along with a single point addition unit, a parallel version of the classical double-and-add scalar multiplication algorithm can be obtained, with an estimated speedup of up to 14% percent when compared with the traditional sequential version.

In §10.7 we presented an architecture that is able to compute the elliptic curve scalar multiplication using the half-and-add method. Additionally, we presented optimizations strategies for computing a point addition and a point doubling using LD projective coordinates in just eight and three clock cycles, respectively.

Finally, in §10.8 we compared some of the most representative elliptic curve designs reported during this decade. In our survey we considered three metrics: speed, compactness and efficiency. Our study tries to sum up the state-of-the-art of scalar multiplication hardware implementations.

References

1. S. Adam, J. Ioannidis, and A. D. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. Technical report, ATT Labs TD-4ZCPZZ, Available at: http://www.cs.rice.edu/~astubble/wep., August 2001.
2. H. Aigner, H. Bock, M. Hütter, and J. Wolkerstorfer. A Low-Cost ECC Coprocessor for Smartcards. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2004.
3. Altera. Design Software, 2006.
   URL: http://www.altera.com/products/software/sfw-index.jsp.
4. Altera. Device Family Overview, 2006.
   http://www.altera.com/products/devices/common/dev-family_overview.html.
5. Altera. The Nios II Processor, 2006.
   url: http://www.altera.com/literature/lit-nio2.jsp.
6. D. N. Amanor, V. Bunimov, C. Paar, J. Pelzl, and M. Schimmler. Efficient Hardware Architectures for Modular Multiplication on FPGAs. In T. Rissa, S. J. E. Wilton, and P. H. W. Leong, editors, *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL), Tampere, Finland, August 24-26, 2005*, pages 539–542. IEEE, 2005.
7. Amphion Semiconductor. *CS5210-40: High Performance AES Encryption Cores*, 2003.
8. R. J. Anderson and E. Biham. TIGER: A Fast New Hash Function. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 89–97, London, UK, 1996. Springer-Verlag.
9. B. Ansari and H. Wu. Parallel Scalar Multiplication for Elliptic Curve Cryptosystems. In *International Conference on Communications, Circuits and Systems, 2005*, volume I, pages 71–73. IEEE Computer Society, May 2005.
10. F. Argüello. Lehmer-Based Algorithm for Computing Inverses in Galois Fields gf(2^m). *IEE Electronic Letters*, 42(5):270–271, March 1997.
11. P. J. Ashenden. *Circuit Design with VHDL*. Morgan Kaufmann Publishers, second edition, 2002.
12. R. M. Avanzi, C. Heuberger, and H. Prodinger. Minimality of the Hamming Weight of the τ-NAF for Koblitz Curves and Improved Combination

with Point Halving. Cryptology ePrint Archive, Report 2005/225, 2005. http://eprint.iacr.org/.
13. R. M. Avanzi and F. Sica. Scalar Multiplication on Koblitz Curves using Double Bases. Cryptology ePrint Archive, Report 2006/067, 2006. http://eprint.iacr.org/.
14. E. Bach and J. Shallit. *Algorithmic Number Theory, Volume I: Efficient Algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.
15. D. Bae, G. Kim, J. Kim, S. Park, and O. Song. An Efficient Design of CCMP for Robust Security Network. In *International Conference on Information Security and Cryptology*, volume 3935, pages 337–346, Seoul, Korea, December 2005. Springer-Verlag.
16. J. C. Bajard, L. Imbert, and G. A. Jullien. Parallel Montgomery Multiplication in GF(2^k) Using Trinomial Residue Arithmetic. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17 2005), 27-29 June 2005, Cape Cod, MA, USA*, pages 164–171. IEEE Computer Society, 2005.
17. P. Barreto. The Hash Functions Lounge. Available at: http://paginas.terra.com.br/informatica/paulobarreto/hflounge.html#BC04.
18. L. Batina, N. Mentens, S.B. Ors, and B. Preneel. Serial Multiplier Architectures over GF(2^n) for Elliptic Curve Cryptosystems. In *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference MELECON 2004*, volume 2, pages 779–782. IEEE Computer Society, May 2004.
19. F. Bauspiess and F. Damm. Requirements for Cryptographic Hash Functions. *Computers and Security*, 11(5):427–437, September 1992.
20. M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, and J. von zur Gathen. Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. In *9th Reconfigurable Architectures Workshop (RAW-02)*, pages 157–164, Fort Lauderdale, Florida, U.S.A., April 2002.
21. G. Bertoni, L. Breveglieri, P. Fragneto, M. Macchetti, and S. Marchesin. Efficient Software Implementation of AES on 32-bits Platforms. In *Proceedings of the CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 159–171. Springer, 2002.
22. E. Biham. A Fast New DES Implementation in Software. In *FSE '97: Proceedings of the 4th International Workshop on Fast Software Encryption*, pages 260–272, London, UK, 1997. Springer-Verlag.
23. E. Biham. A Fast New DES Implementation in Software. In *4th Int. Workshop on Fast Software Encryption, FSE97*, pages 260–271, Haifa, Israel, January 1997. Springer-Verlag, 1997.
24. E. Biham and R. Chen. Near-Collisions of SHA-0. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2004.
25. M. Bishop. An Application of a Fast Data Encryption Standard Implementation. In *Computing Systems, 1(3)*, pages 221–254, Summer 1988.
26. I. F. Blake, V. K. Murty, and G. Xu. A Note on Window τ-NAF Algorithm. *Inf. Process. Lett.*, 95(5):496–502, 2005.
27. G. R. Blakley. A Computer Algorithm for the Product AB modulo M. *IEEE Transactions on Computers*, 32(5):497–500, May 1983.
28. A. Blasius. Generating a Rotation Reduction Perfect Hashing Function. *Mathematics Magazine*, 68(1):35–41, Feb 1995.

29. T. Blum and C. Paar. High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. *IEEE Trans. Computers*, 50(7):759–764, 2001.
30. J. Bos and M. Coster. Addition Chain Heuristics. *In G. Brassard, (editor)* Advances in Cryptology —CRYPTO 89 *Lecture Notes in Computer Science*, 435:400–407, 1989.
31. A. Bosselaers, R. Govaerts, and J. Vandewalle. Fast Hashing on the Pentium. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 298–312, London, UK, 1996. Springer-Verlag.
32. R. P. Brent and H. T. Kung. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*, 31(3):260–264, March 1982.
33. E. F. Brickell. A Fast Modular Multiplication Algorithm with Application to Two Key Cryptography. In *Advances in Cryptology, Proceedings of Crypto 86*, pages 51–60, New York, NY, 1982. Plenum Press.
34. E. F. Brickell. A Survey of Hardware Implementation of RSA (abstract). In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, Lecture Notes in Computer Science, pages 368–370. Springer, 1989.
35. E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast Exponentiation with Precomputation. *In R. A. Rueppel, (editor)* Advances in Cryptology —EUROCRYPT 92 *Lecture Notes in Computer Science*, 658:200–207, 1992.
36. M. Brown, D. Hankerson, J. López, and A. Menezes. Software Implementation of the NIST Elliptic Curves over Prime Fields. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 250–265, London, UK, 2001. Springer-Verlag.
37. G. J. Calderon, J. Velasco-Medina, and J. López-Hernández. Implementación en Hardware del Algoritmo Rijndael [in spanish]. In *X Workshop IBERCHIP*, page 113, 2004.
38. D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.
39. Celoxica. Agility compiler, version 1.2, 2006.
40. CERTICOM. Certicom challenge: Eccp-109 solved. Available at: http://www.certicom.com/index.php, 2002.
41. CERTICOM. Certicom challenge: Ecc2-109 solved. Available at: http://www.certicom.com/index.php, 2004.
42. Certicom™. ECC Tutorial. http://www.certicom.com/index.php?action=ecc_tutorial,home.
43. N. S. Chang, C. H. Kim, Y. H. Park, and J. Lim. A Non-Redundant and Efficient Architecture for Karatsuba-Ofman Algorithm. In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, volume 3650 of *Lecture Notes in Computer Science*, pages 288–299. Springer, 2005.
44. S. Charlwood and P. James-Roxby. Evaluation of the XC6200-Series Architecture for Cryptographic Application. In *FPL 98, Lecture Notes in Computer Science 1482*, pages 218–227. Springer-Verlag Berlin Heidelberg 2003, August/September 1998.

45. F. Charot, E. Yahya, and C. Wagner. Efficient Modular-Pipelined AES Implementation in Counter Mode on ALTERA FPGA. In *Field-Programable Logic and Applications*, pages 282–291, 2003.
46. R. C. C. Cheung, N. J. Telle, W. Luk, and P. Y. K. Cheung. Customizable Elliptic Curve Cryptosystems. *IEEE Trans. Computers on Very Large Scale Integration (VLSI) Systems*, 13(9):1048–1059, September 2005.
47. L. Childs. *A Concrete Introduction to Higher Algebra*. Springer-Verlag Berlin Heidelberg, Germany, 1995.
48. P. Chodowiec and K. Gaj. Very Compact FPGA Implementation of the AES Algorithm. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.
49. D. V. Chudnovsky and G. V. Chudnovsky. Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests. *Advances in Applied Math.*, 7:385–434, 1986.
50. J. Cruz-Alcaraz and F. Rodríguez-Henríquez. Multiplicación Escalar en Curvas de Koblitz: Arquitectura en Hardware Reconfigurable (in spanish). In *XII-IBERCHIP Workshop, IWS-2006*, pages 1–10. Iberoamerican Development Program of Science and Technology (CYTED), March 2006.
51. J. Daemen. *Cipher and Hash Function Design, Strategies Based on Linear and Differential Cryptanalysis*. PhD thesis, Katholieke Universiteit Leuven, 1995.
52. J. Daemen and C. S. K. Clapp. Fast Hashing and Stream Encryption with PANAMA. In *FSE '98: Proceedings of the 5th International Workshop on Fast Software Encryption*, pages 60–74, London, UK, 1998. Springer-Verlag.
53. J. Daemen, R. G., and J. Vandewalle. A Hardware Design Model for Cryptographic Algorithms. In *ESORICS '92: Proceedings of the Second European Symposium on Research in Computer Security*, pages 419–434, London, UK, 1992. Springer-Verlag.
54. J. Daemen, R. Govaerts, and J. Vandewalle. Fast Hashing Both in Hardware and Software. *ESAT-COSIC Report 92-2*, Department of Electrical Engineering, Katholieke Universiteit Leuven, April 1992.
55. J. Daemen, R. Govaerts, and J. Vandewalle. A Framework for the Design of One-Way Hash Functions including Cryptanalysis of Damgård's One-Way Function based on a Cellular Automaton. In *ASIACRYPT*, pages 82–96, 1991.
56. J. Daemen and V. Rijmen. *The Design of Rijndael, AES-The Advance Encryption Standard*. Springer-Verlag Berlin Heidelberg, New York, 2002.
57. W. M. Dal and R. G. Kammer. FIPS 180-1: Secure Hash Standard SHA1, January 2000. Available at: http://www.nist.org.
58. I. Damgard. A Design Principle for Hash Functions. In *CRYPTO '89: Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology*, pages 416–427, London, UK, 1990. Springer-Verlag.
59. A. Dandalis, V. K. Prasanna, and J. D. P. Rolim. A Comparitive Study of Performance of AES Candidates Using FPGAs. In *The Third AES3 Candidate Conference*, New York, April 2000.
60. M. Davio, Y. Desmedt, J. Goubert, F. Hoornaert, and J. J. Quisquater. Efficient Hardware and Software Implementations for the DES. In *Proc. of Crypto' 83*, pages 144–146, August 1984.

61. J. Deepakumara, H. Heys, and R. Venkatesan. FPGA Implementation of MD5 Hash Algorithm. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 919–924, Toronto, Canada, May 2001.
62. A. Desboves. Résolution, en nombres entiers et sous sa forme la plus générale, de l'équation cubique, homogène, à trois inconnues. *Nouvelles Annales de Mathématiques 3-éme série*, 5:545–579, 1886.
63. J.M. Diez, S. Bojanic, Lj. Stanimirovicc, C. Carreras, and O. Nieto-Taladriz. Hash Algorithms for Cryptographic Protocols: FPGA Implementations. In *Proceedings of the* 10^{th} *Telecommunications Forum, TELFOR2002*, Belgrade, Yugoslavia, May 26 -28, 2002.
64. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.
65. V. S. Dimitrov, L. Imbert, and P. K. Mishra. Fast Elliptic Curve Point Multiplication using Double-Base Chains. Cryptology ePrint Archive, Report 2005/069, 2005. Available at: http://eprint.iacr.org/.
66. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 71–82, London, UK, 1996. Springer-Verlag.
67. S. Dominikus. A Hardware Implementation of MD4-Family Hash Algorithms. In *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2002*, Dubrovnik, Croatia, Sep. 15–18 2002.
68. S. R. Dussé and B. S. Kaliski, Jr. A Cryptographic Library for the Motorola DSP56000. In *EUROCRYPT '90: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*, pages 230–244, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
69. M. Dworkin. NIST Special Publication 800-58C: Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, May 2004. Available at: http://csrc.nist.gov/CryptoToolkit/modes/.
70. M. Dworkin. NIST Special Publication 800-58B: Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, May 2005. Available at: http://csrc.nist.gov/CryptoToolkit/modes/.
71. Morris Dworkin. NIST Special Publication 800-58A: Recommendation for Block Cipher Modes of Operation, December 2001. Available at: http://csrc.nist.gov/CryptoToolkit/modes/.
72. H. Eberle. A High Speed DES Implementation for Network Applications. In *Advances in Cryptology-CRYPTO'92, Lecture Notes in Computer Science*, pages 521–539, Berlin, Germany, September 1992. Springer-Verlag.
73. H. Eberle, N. Gura, S. C. Shantz, and V. Gupta. A Cryptographic Processor for Arbitrary Elliptic Curves over $GF(2^m)$. Technical Report TR-2003-123, Sun Microsystem Laboratories, Available at: http://research.sun.com/, May 2003.
74. H. Eberle and C. P. Thacker. A 1 Gbit/Second GaAs DES Chip. In *IEEE 1992 Custom Integrated Circuits Conference*, pages 19.7/1–4, New York,USA, 1992. Springer-Verlag.
75. E. E. Swartzlander (editor). *Computer Arithmetic, volume I and II.* IEEE Computer Society Press, Los Alamitos, CA, 1990.
76. Ö. Eğecioğlu and Ç. K. Koç. Fast Modular Exponentiation. In E. Arıkan, editor, *Communication, Control, and Signal Processing: Proceedings of 1990 Bilkent International Conference on New Trends in Communication, Control, and Signal Processing*, pages 188–194. Elsevier, 1990.

77. A. Elbirt and C. Paar. Efficient Implementation of Galois Field Fixed Field Constant Multiplication. In *Third International Conference on Information Technology: New Generations, ITNG 2006*, pages 172–177. IEEE Computer Society, April 2006.
78. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(4):545–557, 2001.
79. J. Elbirt, W. Yip, B. Chetwyned, and C. Paar. A FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalist. In *The Third AES3 Candidate Conference*, New York, April 2000.
80. T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.
81. S. S. Erdem and Ç. K. Koç. A Less Recursive Variant of Karatsuba-Ofman Algorithm for Multiplying Operands of Size a Power of Two. In *16th IEEE Symposium on Computer Arithmetic (Arith-16 2003), 15-18 June 2003, Santiago de Compostela, Spain*, pages 28–35. IEEE Computer Society, 2003.
82. M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $GF(2^n)$. In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 381–399. Springer-Verlag, 2003.
83. ETSI. European Telecommunications Standards Institute. URL: http://www.etsi.//org/.
84. ETSI. ETSI Technical Specification. Access Transmission Systems on Metallic Access Cables; Very High Speed Digital Subscriber Line (VDSL); Part 1: Functional requirements.
85. H. Fan and Y. Dai. Low Complexity Bit-Parallel Normal Bases Multipliers for $GF(2^n)$. *IEE Electronics Letters*, 40(1):24–26, 2004.
86. H. Fan and Y. Dai. Fast Bit-Parallel $GF(2^n)$ Multiplier for All Trinomials. *IEEE Trans. Computers*, 54(4):485–490, 2005.
87. H. Fan and M. Anwar Hasan. A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields. Centre for Applied Cryptographic Research (CACR) Technical Report CACR 2006-02, 2006. available at: http://www.cacr.math.uwaterloo.ca/.
88. D. C. Feldmeier. A High Speed Crypt Program, April 1989. Technical Memo TM-ARH-013711.
89. G. L. Feng. A VLSI Architecture for Fast Inversion in $GF(2^m)$. *IEEE Transactions on Computers*, 38(10):1383–1386, October 1989.
90. FIPS. Data Encryption Standard. Federal Information Standards Publication, Dec. 1993. Federal Information Processing Standards Publication 46-2.
91. FIPS (Federal Information Processing Standards Publication). *Secure Hash Standard: FIPS PUB 180.* Federal Information Processing Standards Publication, May 1993. Available at: http://www.nist.org.
92. K. Fong, D. Hankerson, J. López, and A. Menezes. Field Inversion and Point Halving Revisited. *IEEE Trans. Computers*, 53(8):1047–1059, 2004.
93. A. P. Fournaris and O. Koufopavlou. $GF(2^k)$ Multipliers Based on Montgomery Multiplication Algorithm. In *Proceedings of the 2004 International Symposium on Circuits and Systems ISCAS'04*, volume 2, pages 849–852, May 2004.

94. M. K. Franklin, editor. *Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*. Springer, 2004.
95. Free-DES. Free-DES Core (2000), March 2000. URL: http://www.free-ip.com/DES/.
96. Y. Fu, L. Hao, and X. Zhang. Design of an Extremely High Performance Counter Mode AES Reconfigurable Processor. In *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05)*, pages 262–268. IEEE Computer Society, 2005.
97. G. Estrin. Organization of Computer Systems – the Fixed Plus Variable Structure Computer. In *Western Joint Computer Conference*, volume 3, pages 33–40, 1960.
98. K. Gaj and P. Chodowiec. Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. In *The Third AES3 Candidate Conference*, pages 40–54, New York, April 2000.
99. K. Gaj and P. Chodowiec. Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 84–99, London, UK, 2001. Springer-Verlag.
100. M. García-Martínez, R. Posada-Gamez, G. Morales-Luna, and F. Rodríguez-Henríquez. FPGA Implementation of an Efficient Multiplier over Finite Fields $GF(2^m)$. In *International Conference on Reconfigurable Computing and FPGAs ReConFig05, Puebla City, Mexico*, pages 1–4, September 2005.
101. H. L. Garner. The Residue Number Systems. *IRE Transactions on Electronic Computers*, 8(6):140–147, June 1959.
102. J. Gathen and J. Shokrollahi. Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F_2. In *Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers*, volume 3897 of *Lecture Notes in Computer Science*, pages 359–369. Springer-Verlag, 2006.
103. P. Gauravaram, W. Millan, and J. Gonzalez-Nieto. Some Thoughts on Collision Attacks in the Hash Functions MD5, SHA-0 and SHA-1. Cryptology ePrint Archive, Report 2005/391, 2005. Available at: http://eprint.iacr.org/.
104. B. Gilchrist, J. H. Pomerene, and S. Y. Wong. Fast Carry Logic for Digital Computers. *IRE Transactions on Electronic Computers*, 4:133–136, 1955.
105. B. Gladman. The AES Algorithm (Rijndael) in C and C++. Available at: http://fp.gladman.plus.com/cryptography_technology/rijndael/.
106. O. Goldreich. *Foundations of Cryptography Volume 1, Basic Tools*. Cambridge University Press, 2003. Reprinted with corrections.
107. O. Goldreich. *Foundations of Cryptography Volume 2, Basic Applications*. Cambridge University Press, 2004.
108. D. Gollmann. Equally Spaced Polynomials, Dual Bases, and Multiplication in F_{2^n}. *IEEE Trans. Computers*, 51(5):588–591, 2002.
109. T. Good and M. Benaissa. AES on FPGA from the Fastest to the Smallest. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2005.

110. J. Goodman and A. P. Chandrakasan. An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, Nov. 2001.
111. D. Gordon. Discrete Logaritms in $GF(P)$ Using the Number Field Sieve. *SIAM Journal on Discrete Mathematics*, 6:124–138, 1993.
112. D. M. Gordon. A Survey of Fast Exponentiation Methods. *Journal of Algorithms*, 27(1):129–146, April 1998.
113. C. Grabbe, M. B., J. Gathen, J. Shokrollahi, and J. Teich. A High Performance VLIW Processor for Finite Field Arithmetic. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 189. IEEE Computer Society, 2003.
114. C. Grabbe, M. Bednara, J. Teich, J. Gathen, and J. Shokrollahi. FPGA Designs of Parallel High Performance GF(2^{233}) Multipliers. In *ISCAS (2)*, pages 268–271, 2003.
115. X. Gregg. Hashing Forth: It's a Topic Discussed so Nonchalantly that Neophytes Hesitate to Ask How it Works. *Forth Dimensions*, 17(4), 1995.
116. T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott. Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512. In *ISC '02: Proceedings of the 5th International Conference on Information Security*, pages 75–89, London, UK, 2002. Springer-Verlag.
117. J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology-CRYPTO 97*, volume 1294 of *Lecture Notes in Computer Science*, pages 342–356, Berlin, Germany, 1997. Springer-Verlag.
118. J. Guajardo and C. Paar. Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes. *Designs, Codes and Cryptography*, 25:207–216, 2002.
119. Z. Guo, B. Buyukkurt, W. Najjar, and K. Vissers. Optimized Generation of Data-Path from C Codes for FPGAs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.
120. Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 162–170, New York, NY, USA, 2004. ACM Press.
121. N. Gura, S. Shantz, and H. Eberle et. al. An End-to-End Systems Approach to Elliptic Curve Cryptography. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, 2523:349–365, August 2003.
122. A. A. A. Gutub, M. K. Ibrahim, and A. Kayali. Pipelining GF(P) Elliptic Curve Cryptography Computation. In *International Conference on Communications, Circuits and Systems, 2005*, pages 93–99. IEEE Computer Society, March 2006.
123. A. A. A. Gutub, A. F. Tenca, E. Savas, and Ç. K. Koç. Scalable and Unified Hardware to Compute Montgomery Inverse in GF(P) and GF(2^n). *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA*, 2523:484–499, August 20002.
124. A. Halbutogullari and Ç. K. Koç. Mastrovito Multiplier for General Irreducible Polynomials. *IEEE Transactions on Computers*, 49(5):503–518, 2000.

125. A. Halbutogullari and Ç. K. Koç. Parallel Multiplication in using Polynomial Residue Arithmetic. *Des. Codes Cryptography*, 20(2):155–173, 2000.
126. T. R. Halfhill. MIPS Embraces Configurable Technology: Pro Series Processors with Corextend Compete with ARC and Tensilica, March 2003. Available at: http://www.altera.com/literature/lit-nio2.jsp.
127. P. Hämäläinen, M. Hännikäinen, and J. Saarinen. Configurable Hardware Implementation of Triple-DES Encryption Algorithm for Wireless Local Network. In *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2001)*, volume II, pages 1221–1224, Salt Lake City, USA, May 2001. IEEE.
128. D. Hankerson, J. López-Hernández, and A. Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, 1965:1–24, August 2000.
129. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Cryptography*. Springer-Verlag, New York, 2004.
130. D. Harris, R. Krishnamurthy, M. Anders, S. Mathew, and S. Hsu. An Improved Unified Scalable Radix-2 Montgomery Multiplier. In *17th IEEE Symposium on Computer Arithmetic (ARITH-17 2005), 27-29 June 2005, Cape Cod, MA, USA*, pages 172–178. IEEE Computer Society, 2005.
131. M. A. Hasan. Efficient Computation of Multiplicative Inverses for Cryptographic Applications. In *15th IEEE Symposium on Computer Arithmetic*, Vail, Colorado, U.S.A., June 2001.
132. M. A. Hasan, M. Z. Wang, and V. K. Bhargava. A Modified Massey-Omura Parallel Multiplier for a Class of Finite Fields. *IEEE Transactions on Computers*, 42(10):1278–1280, November 1993.
133. S. M. Hernández-Rodríguez and F. Rodríguez-Henríquez. An FPGA Arithmetic Logic Unit for Computing Scalar Multiplication Using the Half-and-Add Method. In *IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig05)*, pages 1–7. IEEE Computer Society Press, September 2005.
134. Y. Hirano, T. Satoh, and F. Miura. Improved Extendible Hashing with High Concurrency. *Systems and Computers in Japan*, 26(13):1–11, 1995.
135. F. Hoornaert, M. Decroos, J. Vandewalle, and R. Govaerts. Fast RSA-Hardware: Dream or Reality? In *Advances in Cryptology — EUROCRYPT 88*, volume 330 of *Lecture Notes in Computer Science*, pages 257–264. Springer, 1988.
136. S. F. Hsiao and M. C. Chen. Efficient Substructure Sharing Methods for Optimising the Inner-Product Operations in Rijndael Advanced Encryption Standard. *IEE Proceedings on Computer and Digital Technology*, 152(5):653–665, September 2005.
137. M. Hutton, J. Rabaey, G. Delp, R. Vasishta, V. Betz, and S. Knapp. Will Power Kill FPGAs?, 2006. Session Chair-Mike Hutton.
138. K. Hwang. *Computer Arithmetic, Principles, Architecture, and Design*. John Wiley & Sons, New York, NY, 1979.
139. T. Ichikawa, T. Kasuya, and M. Matsui. Hardware Evaluation of the AES Finalists. In *The Third AES3 Candidate Conference*, pages 279–285, New York, April 2000.
140. IEEE. IEEE 802 LAN/MAN Standards Committee. URL: http://grouper.ieee.org/groups/802/index.html.

141. IEEE standards documents. *IEEE P1363: Standard Specifications for Public Key Cryptography. Draft Version D18.* IEEE, November 2004. http://grouper.ieee.org/groups/1363/.
142. J. L. Imana, J. M. Sanchez, and F. Tirado. Bit-Parallel Finite Field Multipliers for Irreducible Trinomials. *IEEE Transactions on Computers*, 55(5):520–533, 2006.
143. CAST Inc. DES Encryption Core. available from URL: http://www.cast-inc.com.
144. Xilinx Inc., V. Pasham, and S. Triemberger. High-speed DES and TripleDES Encryptor/Decryptor, August 2001. URL: http://www.xilinx.com/xapp/xapp270.pdf.
145. Y. Inoguchi. Outline of the Ultra Fine Grained Parallel Processing by FPGA. In *Seventh International COnference on High Performance Computing and Grid in Asia Pacific Region HPCAsia'04*, pages 434–441. IEEE Computer Society Press, July 2004.
146. ISO. ISO standard 8731-2, 1988. Available at: http://www.iso.org/.
147. ISO. ISO N179 AR Fingerprint Function. Working document, ISO-IEC/JTC1/SC27 /WG2, International Organization for Standardization, 1992.
148. ISO/IEC 15946. Information Technology - Security Techniques - Cryptographic techniques based on Elliptic Curve. *Committee Draft (CD),*, 1999. URL: http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CS NUMBER=31077.
149. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Basis. *Information and Computing*, 78:171–177, 1988.
150. ITU. International Telecommunication Union. URL: http://www.itu.int/home/index.html.
151. K. Jarvinen, M. Tommiska, and J. Skytta. A Scalable Architecture for Elliptic Curve Point Multiplication. In *IEEE International Conference on Field-Programmable Technology, FPT2004*, pages 303–306. IEEE Computer Society Press, December 2004.
152. K. Jarvinen, M. Tommiska, and J. Skytta. Hardware Implementation Analysis of the MD5 Hash Algorithm. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9*, page 298.1, Washington, DC, USA, 2005. IEEE Computer Society.
153. K. U. Jarvinen, M. T. Tommiska, and J. O. Skytta. A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor. In *Proc. of Int. Symp. Field-Programmable Gate-Arrays (FPGA2003*, pages 207–215, Monterey, CA, Feb. 2003.
154. J. Jedwab and C. J. Mitchell. Minimum Weight Modified Signed-Digit Representations and Fast Exponentiation. *IEE Electronics Letters*, 25(17):1171–1172, August 1989.
155. A. Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.
156. M. Joye and J. Quisquater. Hessian Elliptic Curves and Side-Channel Attacks. *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, 2162:402–410, May 2001.

157. M. Joye and J. J. Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004.
158. B. S. Kaliski Jr. *RFC 1319: The MD2 Message-Digest Algorithm*. Internet Activities Board, April 1992.
159. B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*. Springer, 2003.
160. M. Juliato, G. Araujo, J. López, and R. Dahab. A Custom Instruction Approach for Hardware and Software Implementations of Finite Field Arithmetic over F_2^{163} using Gaussian Normal Bases. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005, 11-14 December 2005, Singagore*, pages 5–12. IEEE Computer Society, 2005.
161. A. Kahate. *Cryptography and Network Security*. Tata McGraw-Hill, 2003.
162. Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi. An Efficient Implementation of Hash Function Processor for IPSEC. In *Proceedings of 2002 IEEE Asia-Pacific Conference on ASIC*, pages 93–96, Taipei, Taiwan, Aug 2002.
163. J. P. Kaps and C. Paar. Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine. In *Proc. 5th Annual Workshop on selected areas in cryptography-Sac' 98*, pages 234–247, Ontario, Canada, August 1998. Springer-Verlag, 1998.
164. A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Phys. Doklady (English Translation)*, 7(7):595–596, January 1963.
165. P. R. Karn. Karns DES implementation source code.
166. K. Kelley and D. Harris. Very High Radix Scalable Montgomery Multipliers. In *Proceedings of the 5th IEEE International Workshop on System-on-Chip for Real-Time Applications (IWSOC 2005), 20-24 July 2004, Banff, Alberta, Canada*, pages 400–404. IEEE Computer Society, 2005.
167. M. Khabbazian and T.A. Gulliver. A New Minimal Average Weight Representation for Left-to-Right Point Multiplication Methods. Cryptology ePrint Archive, Report 2004/266, 2004. Available at: http://eprint.iacr.org/.
168. J. H. Kim and D. H. Lee. A Compact Finite Field Processor over $GF(2^m)$ for Elliptic Curve Cryptography. In *IEEE International Conference on Communications, Circuits and Systems, ICCCAS 2002*, volume II, pages 340–342. IEEE Computer Society Press, May 2002.
169. P. Kitsos and O. Koufopavlou. Efficient Architecture and Hardware Implementation of the Whirlpool Hash Function. *IEEE Transactions on Consumer Electronics*, 50(1):208–214, February 2004.
170. V. Klima. Finding MD5 Collisions a Toy for a Notebook. Cryptology ePrint Archive, Report 2005/075, 2005. Available at: http://eprint.iacr.org/.
171. V. Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105, 2006. Available at: http://eprint.iacr.org/.
172. E. W. Knudsen. Elliptic Scalar Multiplication Using Point Halving. In K. Y. Lam, E. Okamoto, and C. Xing, editors, *Advances in Cryptology - ASIACRYPT '99*, volume 1716 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.

173. L. R. Knudsen. SMASH A Cryptographic Hash Function. In *FSE*, pages 228–242, 2005. to appear.
174. D. E. Knuth. *The Art of Computer Programming 3rd. ed.* Addison-Wesley, Reading, Massachusetts, 1997.
175. N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of Computation*, 48(177):203–209, Janury 1987.
176. N. Koblitz. CM-Curves with Good Cryptographic Properties. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, 1991.
177. Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, 71 pages, RSA Laboratories, Redwood City, CA, 1994.
178. Ç. K. Koç and T. Acar. Montgomery Multiplication in GF(2^k). *Designs, Codes and Cryptography*, 14(1):57–69, 1998.
179. Ç. K. Koç and C. Y. Hung. Carry Save Adders for Computing the Product AB modulo N. *IEE Electronics Letters*, 26(13):899–900, June 1990.
180. Ç. K. Koç and C. Y. Hung. Multi-Operand Modulo Addition Using Carry Save Adders. *IEE Electronics Letters*, 26(6):361–363, March 1990.
181. Ç. K. Koç and C. Y. Hung. Bit-Level Systolic Arrays for Modular Multiplication. *Journal of VLSI Signal Processing*, 3(3):215–223, 1991.
182. Ç. K. Koç, D. Naccache, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*. Springer, 2001.
183. Ç. K. Koç and C. Paar, editors. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*. Springer, 1999.
184. Ç. K. Koç and C. Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*. Springer, 2000.
185. M. Kochanski. Developing an RSA Chip. In *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 350–357. Springer, 1985.
186. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.
187. I. Koren. *Computer Arithmetic Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
188. D. C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, NY, 1992.
189. D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi. Compile-time Area Estimation for LUT-based FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):104–122, 2006.
190. N. Kunihiro and H. Yamamoto. New Methods for Generating Short Addition Chains. *IEICE Trans. Fundamentals*, E83-A(1):60–67, January 2000.
191. I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. In *FPGA'06: Proceedings of the internation symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM Press.

192. A. Labbé and A. Pérez. AES Implementations on FPGA: Time Flexibility Tradeoff. In *Proceedings of FPL02*, pages 836–844, 2002.
193. RSA Laboratories. The Public-Key Cryptography Standards (PKCS), June 2002. Available at: http://www.rsasecurity.com/rsalabs/node.asp?id=2124.
194. RSA Laboratories. RSA Challenge. Available at: http://www.rsasecurity.com/rsalabs/node.asp?id=2092, November 2005.
195. RSA Laboratories. RSA Security, 2005. http://www.rsasecurity.com/rsalabs/.
196. R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, 1980.
197. S. Lakshmivarahan and S. K. Dhall. *Parallelism in the Prefix Problem.* Oxford University Press, Oxford, London, 1994.
198. J. Lamoureux and S. J. E. Wilton. FPGA Clock Network Architecture: Flexibility vs. Area and Power. In *FPGA'06: Proceedings of the international symposium on Field programmable gate arrays*, pages 101–108, New York, NY, USA, 2006. ACM Press.
199. D. Laurichesse and L. Blain. Optimized Implementation of RSA Cryptosystem. *Computers & Security*, 10(3):263–267, May 1991.
200. S. O. Lee, S. W. Jung, C. H. Kim, J. Yoon, J. Y. Koh, and D. Kim. Design of Bit Parallel Multiplier with Lower Time Complexity. In *Information Security and Cryptology - ICISC 2003, 6th International Conference, Seoul, Korea, November 27-28, 2003, Revised Papers*, volume 2971 of *Lecture Notes in Computer Science*, pages 127–139. Springer-Verlag, 2004.
201. H. Leitold, W. Mayerwieser, U. Payer, K. C. Posch, R. Posch, and J. Wolkerstorfer. A 155 Mbps Triple-DES Network Encryptor. In *CHESS 2000*, pages 164–174, LNCS 1965, 2000. Springer-Verlag.
202. A. Lenstra and H. Lenstra, editors. *The Development of the Number Field Sieve, Lecture Notes in Mathematics 1554.* Springer-Verlag, 1993.
203. J. Leonard and W. H. Magione-Smith. A Case Study of Partially Evaluated Hardware Circuits: Key Specific DES. In *Field-Programmable Logic and Applications, FPL' 97*, pages 234–247, London, UK, September 1997. Springer-Verlag, 1997.
204. I. K. H. Leung and P. H. W. Leong. A Microcoded Elliptic Curve Processor using FPGA Technology. *IEEE Transactions on VLSI Systems*, 10(5):550–559, 2002.
205. S. Levy. The Open Secret. *Wired Magazine*, 7(04):1–6, April 1999. Available at: http://www.wired.com/wired/archive/7.04/crypto.html.
206. D. Lewis, E. Ahmed, G. Baeckler, V. Betz, and et al. The Stratix II Logic and Routing Architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, New York, NY, USA, 2005. ACM Press.
207. D. Lewis, V. Betz, D. Jefferson, A. Lee, C. Lane, P. Leventis, and et al. The Stratix 960; Routing and Logic Architecture. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20, New York, NY, USA, 2003. ACM Press.
208. J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley, Reading, MA, 1981.
209. Q. Liu, D. Tong, and X. Cheng. Non-Interleaving Architecture for Hardware Implementation of Modular Multiplication. In *IEEE International Symposium on Circuits and Systems, 2005. ISCAS 2005*, volume 1, pages 660–663. IEEE, May 2005.

210. J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In *SAC'98*, volume 1556 of *Lecture Notes in Computer Science*, pages 201–212, 1998.
211. J. Lopez and R. Dahab. Fast Multiplication on Elliptic Curves over $GF(2^m)$ without Precomputation. *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, 1717:316–327, August 1999.
212. J. López-Hernández. Personal communication with J. López-Hernández, 2006.
213. E. López-Trejo, F. Rodríguez Henríquez, and A. Díaz-Pérez. An Efficient FPGA Implementation of CCM Mode Using AES. In *International Conference on Information Security and Cryptology*, volume 3935 of *Lecture Notes in Computer Science*, pages 208–215, Seoul, Korea, December 2005. Springer-Verlag.
214. A. K. Lutz, J. Treichler, F. K. Gurkaynak, H. Kaeslin, G. Basler, A. Erni, S. Reichmuth, P. Rommens, S. Oetiker, and W. Fitchtner. 2 Gbits/s Hardware Realization of RIJNDAEL and SERPENT-A Comparative Analysis. In *Proceedings of the CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2002.
215. J. Lutz. High Performance Elliptic Curve Cryptographic Co-processor. Master's thesis, University of Waterloo, 2004.
216. R. Lysecky and F. Vahid. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 18–23. IEEE Computer Society, 2005.
217. S. Mangard. A High Regular and Scalable AES Hardware Architecture. *IEEE Transactions on Computers*, 52(4):483–491, April 2003.
218. G. Martínez-Silva, F. Rodríguez-Henríquez, N. Cruz-Cortés, and L. G. De la Fraga. On the Generation of X.509v3 Certificates with Biometric Information. Technical report, CINVESTAV-IPN, April 2006. Available at: http://delta.cs.cinvestav.mx/ francisco/.
219. E. D. Mastrovito. VLSI Designs for Multiplication over Finite Fields GF (2^m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 6th International Conference, AAECC-6, Rome, Italy, July 4-8, 1988, Proceedings*, volume 357 of *Lecture Notes in Computer Science*, pages 297–309. Springer-Verlag, 1989.
220. R. J. McEliece. *Finite Fields for Computer Scientists and Engineers.* Kluwer Academic Publishers, Boston, MA, 1987.
221. R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. Optimisation of the SHA-2 Family of Hash Functions on FPGAs. *ISVLSI 2006*, pages 317–322, 2006.
222. M. McLoone and J. V. McCanny. High Performance FPGA Rijndael Algorithm Implementation. In *Proceedings of the CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 2001.
223. M. McLoone and J.V. McCanny. Efficient Single-Chip Implementation of SHA-384 and SHA-512. In *Proceedings. 2002 IEEE International Conference on Field- Programmable Technology, FPT02*, volume 5, pages 311–314, Hong Kong, December 16-18, 2002.
224. M. McLoone and J.V. McCanny. High-performance FPGA Implementation of DES Using a Novel Method for Implementing the Key Schedule. *IEE Proc.: Circuits, Devices & Systems*, 150(5):373–378, October 2003.

225. M. McLoone, C. McIvor, and A. Savage. High-Speed Hardware Architectures of the Whirlpool Hash Function. In *FPT'05*, pages 147–162. IEEE Computer Society Press, 2005.
226. A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.
227. A. J. Menezes, P. C. van Oorschot, and S. A.Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, 1996.
228. A.J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, 1993.
229. Mentor Graphics. Catapult C, 2005.
230. Mentor Graphics, http://www.model.com/. *ModelSim*, 2005.
231. MentorGraphics, http://www.mentor.com/products/fpga_pld/synthesis/. *LeonardoSpectrum*, 2003.
232. R. Merkle. Secrecy, Authentication, and Public Key Systems. Stanford University, 1979.
233. R. C. Merkle. One Way Hash Functions and DES. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 428–446, New York, NY, USA, 1989. Springer-Verlag New York, Inc.
234. R. C. Merkle. A Fast Software One-Way Hash Function. *Journal of Cryptology*, 3:43–58, 1990.
235. V. Miller. Uses of Elliptic Curves in Cryptography. *In H. C. Williams (editor)* Advances in Cryptology — CRYPTO 85 Proceedings, *Lecture Notes in Computer Science*, 218:417–426, January 1985.
236. S. Miyaguchi, K. Ohta, and M. Iwata. 128-bit Hash Function (N-Hash). In *SECURICOM '90*, pages 123–137, 1990.
237. P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.
238. P. L. Montgomery. Five, Six, and Seven-Term Karatsuba-Like Formulae. *IEEE Trans. Comput.*, 54(3):362–369, 2005.
239. F. Morain and J. Olivos. Speeding Up the Computations on an Elliptic Curve Using Addition-Subtraction Chains. Rapport de Recherche 983, INRIA, March 1989.
240. M. Morii, M. Kasahara, and D. L. Whiting. Efficient Bit-Serial Multiplication and the Discrete-Time Wiener-Hopf Equation over Finite Fields. *IEEE Transactions on Information Theory*, 35(6):1177–1183, 1989.
241. S. Morioka and A. Satoh. An Optimized S-Box Circuit Architecture for Low Power AES Design. In *Proceesings of the CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2002.
242. K. Mukaida, M. Takenaka, N. Torii, and S. Masui. Design of High-Speed and Area-Efficient Montgomery Modular Multiplier for RSA Algorithm. In *IEEE Symposium on VLSI Circuits, 2004*, pages 320–323. IEEE Computer Society, 2004.
243. R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. *Logic Synthesis for Field-Programmable Gate Arrays.* Kluwer Academic Publishers, Norwell, MA, USA, 1995.
244. M. Naor and M. Yung. Universal One-way Hash Functions and their Cryptographic Applications. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 33–43, New York, NY, USA, 1989. ACM Press.

245. J. Nechvatal. Public Key Cryptography. In *In G. Simmons ed. Contemporary Cryptology: The Science of Information Integrity*, Piseataway, NJ, 1992. IEEE Press.
246. C. Nègre. Quadrinomial Modular Arithmetic using Modified Polynomial Basis. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA*, pages 550–555. IEEE Computer Society, 2005.
247. M. Negrete-Cervantes, K. Gómez-Avila, and F. Rodríguez-Henríquez. Investigating Modular Inversion in Binary Finite Fields (in spanish). Technical Report CINVESTAV_COMP 2006-1, 29 pages, Computer Science Department CINVESTAV-IPN, Mexico, May 2006.
248. C. W. Ng, T. S. Ng, and K. W. Yip. A Unified Architecture of MD5 and RIPEMD-160 Hash Algorithms. In *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2004*, volume 2, pages II–889– II–892, Vancouver, Canada, 2004.
249. R. K. Nichols and P. C. Lekkas. *Wireless Security: Models, Threats, and Solutions.* McGraw Hill, 2000.
250. NIST. FIPS 46-3: Data Encryption Standard DES. Federal Information Processing Standards Publication 46-3, 1999. Available at:http://csrc.nist.gov/publications/fips/.
251. NIST. ANSI T1E1.4, Sep. 1 1999. Draft Technical Document, Revision16, Very High Speed Digital Subscriber Lines; System requirements.
252. NIST. Announcing the Advanced Encryption Standard AES. Federal Information Standards Publication, November 2001. Available at: http://csrc.nist.gov/CryptoToolkit/aes/index.html.
253. NIST. FIPS 186-2: Digital Signature Standard DSS. Federal Information Processing Standards Publication 186-2, October 2001. Available at:http://csrc.nist.gov/publications/fips/.
254. NIST. Secure Hash Signature Standard (SHS). Technical Report FIPS PUB 180-2, NIST, August 1 2002.
255. NIST. FIPS 186-3: Digital Signature Standard DSS. Federal Information Processing Standards Publication 186-3, march 2006. Available at: http://csrc.nist.gov/publications/drafts/.
256. Government Committee of Russia for Standards. Information Technology. Cryptographic Data Security. Hashing function, 1994. Gosudarstvennyi Standard of Russian Federation.
257. National Institute of Standards and Technology. NIST Special Publication 800-57: Recommendation for Key Management Part 1: General, August 2005.
258. J. V. Oldfield and R. C. Dorf. *Field Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementations of Digital Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.
259. J. K. Omura. A Public Key Cell Design for Smart Card Chips. In *International Symposium on Information Theory and its Applications*, pages 27–30, November 1990.
260. G. Orlando and C. Paar. A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, 1965:41–56, August 2000.

261. G. Orlando and C. Paar. A Scalable $GF(P)$ Elliptic Curve Processor Architecture for Programmable Hardware. *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, 2162:348–363, May 2001.
262. S. B. Örs, E. Oswald, and B. Preneel. Power-Analysis Attacks on an FPGA - First Experimental Results. In *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2003.
263. E. Öztürk, B. Sunar, and E. Savas. Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2004.
264. G. Theodoridis P. Kitsos and O. Koufopavlou. An Efficient Reconfigurable Multiplier for Galois Field $GF(2^m)$. *Elsevier Microelectronics Journal*, 34(10):975–980, October 2003.
265. C. Paar. *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*. PhD thesis, Universität GH Essen, 1994.
266. C. Paar. A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.
267. C. Paar, P. Fleischmann, and P. Roelse. Efficient Multiplier Architectures for Galois Fields GF(2^{4n}). *IEEE Trans. Computers*, 47(2):162–170, 1998.
268. C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents. *IEEE Trans. Computers*, 48(10):1025–1034, 1999.
269. C. Patterson. High Performance DES Encryption in Virtex FPGAs using Jbits. In *Field-programmable custom computing machines, FCCM' 00*, pages 113–121, Napa Valley, CA, USA, January 2000. IEEE Comput. Soc., CA, USA, 2000.
270. V. A. Pedroni. *Circuit Design with VHDL*. The MIT Press, August 2004.
271. J. Pollard. Montecarlo Methods for Index Computacion (mod p). *Mathematics of Computation*, 13:918–924, 1978.
272. N. Pramstaller, C. Rechberger, and V. Rijmen. A Compact FPGA Implementation of the Hash Function Whirlpool. In *FPGA'06: Proceedings of the international symposium on Field Programmable Gate Arrays*, pages 159–166, New York, NY, USA, 2006. ACM Press.
273. B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.
274. B. Preneel. Cryptographic Hash Functions. *European Transactions on Telecommunications*, 5(4):431–448, 1994.
275. B. Preneel. Design Principles for Dedicated Hash Functions. In *Fast Software Encryption, FSE 1993*, volume 809 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1994.
276. B. Preneel, R. Govaerts, and J. Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1994.

277. J. J. Quisquater and C. Couvreur. Fast Decipherment Algorithm for RSA Public-Key Cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.
278. J. R. Rao and B. Sunar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*. Springer, 2005.
279. A. Reyhani-Masoleh. Efficient Algorithms and Architectures for Field Multiplication Using Gaussian Normal Bases. *IEEE Trans. Comput.*, 55(1):34–47, 2006.
280. A. Reyhani-Masoleh and M. A. Hasan. A New Construction of Massey-Omura Parallel Multiplier over GF(2). *IEEE Trans. Computers*, 51(5):511–520, 2002.
281. A. Reyhani-Masoleh and M. A. Hasan. Efficient Multiplication Beyond Optimal Normal Bases. *IEEE Trans. Computers*, 52(4):428–439, 2003.
282. A. Reyhani-Masoleh and M. A. Hasan. Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over GF(2^m). *IEEE Trans. Computers*, 53(8):945–959, 2004.
283. A. Reyhani-Masoleh and M. Anwar Hasan. Low Complexity Word-Level Sequential Normal Basis Multipliers. *IEEE Trans. Comput.*, 54(2):98–110, 2005.
284. V. Rijmen and P. S. L. M. Barreto. The Whirlpool Hash Function. First open NESSIE Workshop, Nov. 13–14 2000.
285. RIPE. RIPE Integrity Primitives: Final Report of RACE Integrity Primitives Evaluation (R1040). Technical report, Research and Development in Advanced Communication Technologies in Europe, June 1992.
286. R. Rivest. The Md4 Message Digest Algorithm. In *Advances in Cryptology – CRYPTO '90 Proceedings*, pages 303–311, 1991.
287. R. Rivest. The MD5 Message-Digest Algorithm. Technical Report Internet RFC-1321, IETF, 1992. http://www.ietf.org/rfc/rfc1321.txt.
288. Ronald L. Rivest. RSA Chips (Past/Present/Future). In *Advances in Cryptology, Proceedings of EUROCRYPT 84*, volume 209 of *Lecture Notes in Computer Science*, pages 159–165, 1984.
289. F. Rodríguez-Henríquez. New Algorithms and Architectures for Arithmetic in GF(2^m) Suitable for Elliptic Curve Cryptography, PhD thesis: Oregon State University, 2000.
290. F. Rodríguez-Henríquez and Ç. K. Koç. On Fully Parallel Karatsuba Multipliers for $GF(2^m)$. In *International Conference on Computer Science and Technology (CST 2003)*, pages 405–410, Cancun, Mexico, May 2003.
291. F. Rodríguez-Henríquez and Ç. K. KoÇ. Parallel Multipliers Based on Special Irreducible Pentanomials. *IEEE Trans. Computers*, 52(12):1535–1542, 2003.
292. F. Rodríguez-Henríquez, C.E. López-Peza, and M.A León-Chávez. Comparative Performance Analysis of Public-Key Cryptographic Operations in the WTLS Handshake Protocol. In *1st International Conference on Electrical and Electronics Engineering ICEEE 2004*, pages 124–129. IEEE Computer Society, 2004.
293. F. Rodríguez-Henríquez, G. Morales-Luna, N. Saqib, and N. Cruz-Cortés. Parallel Itoh-Tsujii Multiplicative Inversion Algorithm for a Special Class of Trinomials. Cryptology ePrint Archive, Report 2006/035, 2006. http://eprint.iacr.org/.
294. F. Rodríguez-Henríquez, N. A. Saqib, and N. Cruz-Cortés. A Fast Implementation of Multiplicative Inversion over GF(2^m). In *International Symposium*

on Information Technology (ITCC 2005), volume 1, pages 574–579, Las Vegas, Nevada, U.S.A., April 2005.

295. F. Rodríguez-Henríquez, N. A. Saqib, and A. Díaz-Pérez. 4.2 Gbit/s Single-Chip FPGA Implementation of AES Algorithm. *IEE Electronics Letters*, 39(15):1115–1116, July 2003.
296. F. Rodríguez-Henríquez, N. A. Saqib, and A. Díaz-Pérez. A Fast Parallel Implementation of Elliptic Curve Point Multiplication over GF(2^m). *Microprocessor and Microsystems*, 28(5-6):329–339, August 2004.
297. K. Rosen. *Elementary Number Theory and its Applications*. Addison-Wesley, Reading, MA, 1992.
298. G. Rouvroy, F. X. Standaert, J. J. Quisquater, and J. D. Legat. Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES. In *FPL 2003*, volume 2778 of *Lecture Notes in Computer Science*, pages 181–193. Springer-Verlag Berlin Heidelberg 2003, 2003.
299. G. Rouvroy, F. X. Standaert, J. J. Quisquater, and J. D. Legat. Eficcient Uses of FPGAs for Implementations of DES and its Experimental Linear Cryptoanalysis. *IEEE Transactions on Computers*, 52(4):473–482, 2003.
300. G. Rouvroy, F. X. Standaert, J. J. Quisquater, and J. D. Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of AES Rijndael Very Well Suited for Embedded Applications. In *International Conference on Information Technology: Coding and Computing 2004 (ITCC2004)*, volume 2, pages 538–587, 2004.
301. A. Rudra, P. K. Dubey, C. S. Julta, V. Kumar, J. R. Rao, and P. Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In *Proceedings of the CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.
302. A. Rushton. *VHDL for Logic Synthesis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.
303. G. P. Saggese, A. Mazzeo, N. Mazzocca, and A. G. M. Strollo. An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm. In *Field-Programable Logic and Applications FPL03, Lecture Notes in Computer Science 2778*, pages 292–302, 2003.
304. N. A. Saqib, A. Díaz-Pérez, and F. Rodríguez-Henríquez. Highly Optimized Single-Chip FPGA Implementations of AES Encryption and Decryption Cores. In *X Workshop Iberchip*, pages 117–118, Cartagena-Colombia, March 2004.
305. N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. Sequential and Pipelined Architecures for AES Implementation. In *Proceedings of the IASTED International Conference on Computer Science and Technology*, pages 159–163, Cancun, Mexico, May 2003. IASTED/ACTA Press.
306. N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. Two Approaches for a Single-Chip FPGA Implementation of an Encryptor/Decryptor AES Core. In *FPL 2003*, volume 2778 of *Lecture Notes in Computer Science*, pages 303–312. Springer-Verlag Berlin Heidelberg 2003, 2003.
307. N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. A Compact and Efficient FPGA Implementation of the DES Algorithm. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig04)*, pages 12–18, Colima, Mexico, September 2004. Mexican Society for Computer Sciences.

308. N. A. Saqib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. A Reconfigurable Processor for High Speed Point Multiplication in Elliptic Curves. *International Journal of Embedded Systems, (*In press *)*, 2006.
309. N. A. Saquib, F. Rodríguez-Henríquez, and A. Díaz-Pérez. AES Algorithm Implementation - An Efficient Approach for Sequential and Pipeline Architecures. In *Fourth Mexican International Conference on Computer Science*, pages 126–130, Tlaxcala-Mexico, September 2003. IEEE Computer Society Press.
310. A. Satoh and T. Inoue. ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 532–537, Washington, DC, USA, 2005. IEEE Computer Society.
311. A. Satoh and K. Takano. A Scalable Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Transactions on Computers*, 52(4):449–460, April 2003.
312. E. Savas, M. Naseer, A. Gutub A.A, and Ç. K. Koç. Efficient Unified Montgomery Inversion with Multibit Shifting. *IEE Proceedings-Computers and Digital Techniques*, 152(4):489–498, July 2005.
313. E. Savas, A. F. Tenca, and Ç. K. Koç. A Scalable and Unified Multiplier Architecture for Finite Fields GF() and GF(2^m). In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 277–292. Springer-Verlag, 2000.
314. N. Schappacher. Développement de la loi de groupe sur une cubique. *Progress in Mathematics-Birkhäuser*, pages 159–184, 1991. available at:http://www-irma.u-strasbg.fr/ schappa/Publications.html.
315. B. Schneier. *Applied Cryptography*. John Wiley and Sons, New York, second edition edition, 1998.
316. C. P. Schnorr. FFT-Hashing, An Efficient Cryptographic Hash Function, 1991. Crypto'91 rump session, unpublished manuscript.
317. C. P. Schnorr. FFT-hash II, Efficient Cryptographic Hashing. *Lecture Notes in Computer Sciences*, 658:45–54, 1993.
318. C. P. Schnorr and S. Vaudenay. Parallel FFT-Hashing. In *Fast Software Encryption, Cambridge Security Workshop*, pages 149–156, London, UK, 1994. Springer-Verlag.
319. A. Schönhage. A Lower Bound for the Length of Addition Chains. *Theoretical Computer Science*, 1:1–12, 1975.
320. R. Schroeppel, C. Beaver, R. Gonzales, R. Miller, and T. Draelos. A low-power Design for an Elliptic Curve Digital Signature Chip. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, 2523:366–380, August 2003.
321. R. Schroeppel, H. Orman, S. W. O'Malley, and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In *CRYPTO '95: Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 43–56, London, UK, 1995. Springer-Verlag.
322. H. Sedlak. The RSA Cryptography Processor. In *Advances in Cryptology — EUROCRYPT 87*, volume 304 of *Lecture Notes in Computer Science*, pages 95–105, 1987.
323. A. Segredoas, E. Zabala, and G. Bello. Diseño de un Procesador Criptográfico Rijndael en FPGA [in spanish]. In *X Workshop IBERCHIP*, page 64, 2004.

324. V. Serrano-Hernández and F. Rodríguez-Henríquez. An FPGA Evaluation of Karatusba-Ofman Multiplier Variants (in spanish). Technical Report CINVESTAV_COMP 2006-2, 12 pages, Computer Science Department CINVESTAV-IPN, Mexico, May 2006.
325. A. Shamir. Turing Lecture on Cryptology: A Status Report. Available at: http://www.acm.org/awards/turing_citations/rivest-shamir-adleman.html, 2002.
326. M. B. Sherigar, A. S. Mahadevan, K. S. Kumar, and S. David. A Pipelined Parallel Processor to Implement MD4 Message Digest Algorithm on Xilinx FPGA. In *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, page 394, Washington, DC, USA, 1998. IEEE Computer Society.
327. C. Shu, K. Gaj, and T. A. El-Ghazawi. Low Latency Elliptic Curve Cryptography Accelerators for NIST Curves Over Binary Fields. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005, 11-14 December 2005, Singagore*, pages 309–310. IEEE, 2005.
328. W. Shuhua and Z. Yuefei. A Timing-and-Area Tradeoff GF(P) Elliptic Curve Processor Architecture for FPGA. In *IEEE International Conference on Communications, Circuits and Systems, ICCCAS 2005*, pages 1308–1312. IEEE Computer Society Press, June 2005.
329. K. Siozios, G. Koutroumpezis, K. Tatas, D. Soudris, and A. Thanailakis. DAGGER: A Novel Generic Methodology for FPGA Bitstream Generation and its Software Tool Implementation. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CA, USA*. IEEE Computer Society, 2005.
330. N. Sklavos, P. Kitsos, K. Papadomanolakis, and O. Koufopavlou. Random Number Generator Architecture and VLSI Implementation. In *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2002*, pages IV–854– IV–857, Scottsdale, Arizona, May 2002.
331. N. Sklavos and O. Koufopavlou. On the Hardware Implementations of the SHA-2 (256, 384, 512) Hash Functions. In *Proceedings of IEEE International Symposium on Circuits and Systems, ISCAS 2003*, volume 5, pages V–153–V–156, Bangkok, Thailand, 2003.
332. K. R. Sloan, Jr. Comments on "A Computer Algorithm for the Product AB modulo M". *IEEE Transactions on Computers*, 34(3):290–292, March 1985.
333. N. Smart. The Hessian Form of an Elliptic Curve. *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, 2162:118–125, May 2001.
334. N. Smart and E. Westwood. Point Multiplication on Ordinary Elliptic Curves over Fields of Characteristic Three. *Applicable Algebra in Engineering, Communication and Computing*, 13:485–497, 2003.
335. M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and editors F. J. Taylor. *Residue Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Press, New York, NY, 1986.
336. J. Solinas. Generalized Mersenne Numbers. Technical Report CORR 1999-39, Dept. of Combinatorics and Optimization, Univ. of Waterloo, Canada, 1999.
337. J. A. Solinas. An Improved Algorithm for Arithmetic on a Family of Elliptic Curves. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 357–371, London, UK, 1997. Springer-Verlag.

338. J. A. Solinas. Efficient Arithmetic on Koblitz Curves. *Des. Codes Cryptography*, 19(2-3):195–249, 2000.
339. F. Sozzani, G. Bertoni, S. Turcato, and L. Breveglieri. A Parallelized Design for an Elliptic Curve Cryptosystem Coprocessor. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 626–630, Washington, DC, USA, 2005. IEEE Computer Society.
340. W. Stallings. *Cryptography and Network Security:* Principles and Practice. Prentice Hall, Upper Saddle River, New Jersey 07458, 1999.
341. F. X. Standaert, L. O. T. Oldenzeel, D. Samyde, and J. J. Quisquater. Power Analysis of FPGAs: How Practical is the Attack? In *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 701–711. Springer, 2003.
342. F. X. Standaert, S. B. Örs, and B. Preneel. Power Analysis of an FPGA: Implementation of Rijndael: Is Pipelining a DPA Countermeasure? In M. Joye and J. J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 30–44. Springer, 2004.
343. F. X. Standaert, S. B. Örs, J. J. Quisquater, and B. Preneel. Power Analysis Attacks Against FPGA Implementations of the DES. In *Field Programmable Logic and Application, 14th International Conference , FPL 2004, Leuven, Belgium, August 30-September 1, 2004, Proceedings*, volume 3203 of *Lecture Notes in Computer Science*, pages 84–94. Springer, 2004.
344. F. X. Standaert, G. Rouvroy, J. J. Quisquater, and J. D. Legat. Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 334–350. Springer, 2003.
345. D. R. Stinson. Combinatorial Techniques for Universal Hashing. *Computer and System Sciences*, 48(2):337–346, April 1994.
346. D. R. Stinson. Universal Hashing and Authentication Codes. *Designs, Codes and Cryptography*, 4(4):369–380, 1994.
347. B. Sunar. A Generalized Method for Constructing Subquadratic Complexity GF(2^k) Multipliers. *IEEE Trans. Computers*, 53(9):1097–1105, 2004.
348. B. Sunar and Ç. K. Koç. Mastrovito Multiplier for All Trinomials. *IEEE Transactions on Computers*, 48(5):522–527, May 1999.
349. B. Sunar and Ç. K. Koç. An Efficient Optimal Normal Basis Type II Multiplier. *IEEE Trans. Computers*, 50(1):83–87, 2001.
350. E. J. Swankowski, R. R. Brooks, V. Narayanan, M. Kandemir, and M. J. Irwin. A Parallel Architecture for Secure FPGA Symmetric Encryption. In *18th International Parallel and Distributed Symposium IPDPS'04*, page 132. IEEE Computer Society, 2004.
351. Synopsys, http://www.synopsys.com/products/. *Galaxy Design Platform*, 2006.
352. N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and its Applications to Computer Technology.* McGraw-Hill, New York, NY, 1967.

353. N. Takagi, J. Yoshiki, and K. Tagaki. A Fast Algorithm for Multiplicative Inversion in GF(2^m) Using Normal Basis. *IEEE Transactions on Computers*, 50(5):394–398, May 2001.
354. Helion Tech. High Performance Solution in Silicon: AES (Rijndael) Cores. Available at: http://www.heliontech.com/core2.htm.
355. Helion Technology. Datasheet - High Performance MD5 Hash Core for Xilinx FPGA. url: http://www.heliontech.com/downloads/md5_xilinx_helioncore.pdf.
356. A. F. Tenca and Ç. K. Koç. A Scalable Architecture for Modular Multiplication Based on Montgomery's Algorithm. *IEEE Trans. Comput.*, 52(9):1215–1221, 2003.
357. J. P. Tillich and G. Zémor. Group-Theoretic Hash Functions. In *Algebraic Coding, First French-Israeli Workshop, Paris, France, July 19-21, 1993, Proceedings*, volume 781 of *Lecture Notes in Computer Science*, pages 90–110. Springer, 1993.
358. G. Todorov. ASIC Design, Implementation and Analysis of a Scalable High-Radix Montgomery Multiplier. Master's thesis, Oregon State University, December 2000.
359. W. Trappe and L.C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, Inc., Upper Saddle River, NJ 07458, 2002.
360. S. Trimberger, R. Pang, and A. Singh. A 12 Gbps DES Encryptor/Decryptor Core in an FPGA. In *CHESS 2000*, pages 156–163, LNCS 1965, 2000. Springer-Verlag.
361. T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger. A 90nm Low-power FPGA for Battery-Powered Applications. In *FPGA'06: Proceedings of the internation symposium on Field programmable gate arrays*, pages 3–11, New York, NY, USA, 2006. ACM Press.
362. K. Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM Press.
363. George Mason University. Hardware IP Cores of Advanced Encryption Standard AES-Rijndael. Available at: http://ece.gmu.edu/crypto/rijndael.htm.
364. VASG. VHDL Analysis and Standardization Group, March 2003.
365. C. D. Walter. Systolic Modular Multiplication. *IEEE Transactions on Computers*, 42(3):376–378, March 1993.
366. C. D. Walter, Ç. K. Koç, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.
367. X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. RUmp Session, Crypto 2004, Cryptology ePrint Archive, Report 2004/199, 2004. Available at: http://eprint.iacr.org/.
368. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

369. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.
370. S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital System Designers*. Holt, Rinehart and Winston, New York, NY, 1982.
371. P. Wayner. British Document Outlines Early Encryption Discovery, 1997. http://www.nytimes.com/library/cyber/week/122497encrypt.html.
372. N. Weaver and J. Wawrzynek. High Performance, Compact AES Implementations in Xilinx FPGAs. Technical report, U.C. Berkeley BRASS group, available at http://www.cs.berkeley.edu/ nnweaver/sfra/rijndael.pdf, 2002.
373. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware Performance of Round 2 Advanced Encryption Standard Algorithms. In *The Third AES3 Candidate Conference*, New York, April 2000.
374. A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Ruhr-Universität-Bochum, Germany. Technical Report, 2003. available at: http://www.crypto.ruhr-uni-bochum.de/en_publications.html.
375. D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). In *Submission to NIST*, 2002.
376. S. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
377. S. B. Wicker and V. K. Bhargava (editors). *Reed-Solomon Codes and Their Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1994.
378. D. C. Wilcox, L. G. Pierson, P. J. Robertson, E. L. Witzke, and K. Gass. A DES ASIC Suitable for Network Encryption at 10 Gbs and Beyond. In *CHES 99*, pages 37–48, LNCS 1717, August 1999.
379. T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art Implementations and Attacks. *Trans. on Embedded Computing Sys.*, 3(3):534–574, 2004.
380. T. J. Wollinger and C. Paar. How Secure Are FPGAs in Cryptographic Applications? In *Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003, Proceedings*, volume 2778 of *Lecture Notes in Computer Science*, pages 91–100. Springer, 2003.
381. K. Wong, M. Wark, and E. Dawson. A Single-Chip FPGA Implementation of the Data Encryption Standard (DES) Algorithm. In *IEEE Globecom Communication Conf.*, pages 827–832, Sydney, Australia, Nov. 1998.
382. K. W. Wong, E. C. W. Lee, L. M. Cheng, and X. Liao. Fast Elliptic Scalar Multiplication using New Double-base Chain and Point Halving. Cryptology ePrint Archive, Report 2006/124, 2006. Available at: http://eprint.iacr.org/.
383. H. Wu. Low Complexity Bit-Parallel Finite Field Arithmetic using Polynomial Basis. In Ç. K. Koç and C. Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES 99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 280–291. Springer-Verlag, August 1999.
384. H. Wu. On Complexity of Squaring Using Polynomial Basis in GF(2^m). In S. Tavares D. Stinson, editor, *Workshop on Selected Areas in Cryptography (SAC 2000)*, volume LNCS 2012, pages 118–129. Springer-Verlag, September 2000.

385. H. Wu. Montgomery Multiplier and Squarer for a Class of Finite Fields. *IEEE Trans. Computers*, 51(5):521–529, 2002.
386. H. Wu and M. A. Hasan. Low Complexity Bit-Parallel Multipliers for a Class of Finite Fields. *IEEE Trans. Computers*, 47(8):883–887, 1998.
387. H. Wu, M. A. Hasan, and I. F. Blake. New Low-Complexity Bit-Parallel Finite Field Multipliers Using Weakly Dual Bases. *IEEE Trans. Computers*, 47(11):1223–1234, 1998.
388. H. Wu, M. A. Hasan, I. F. Blake, and S. Gao. Finite Field Multiplier Using Redundant Representation. *IEEE Trans. Computers*, 51(11):1306–1316, 2002.
389. ANSI X9.62. Federal Information Processing Standard (FIPS) 46, National Bureau Standards, January 1977.
390. Xilinx, http://www.xilinx.com/support/techsup/tutorials/index.htm. *ISE 7 In-Depth Tutorial*, 2005.
391. Xilinx. MicroBlaze Soft Processor Core, 2005. Available at: http://www.xilinx.com/.
392. Xilinx, http://www.xilinx.com/bvdocs/publications/ds099.pdf. *Spartan-3 FPGA Family: Complete Data Sheet*, January 2005.
393. Xilinx. Virtex-4 Multi-Platform FPGA, 2005. Available at: http://www.xilinx.com/.
394. Xilinx. Virtex-II platform FPGAs: Complete Data Sheet, 2005. Available at: http://www.xilinx.com/.
395. Xilinx. Virtex-5 Multi-Platform FPGA, May 2006. Available at: http://www.xilinx.com/.
396. S. M. Yen. Improved Normal Basis Inversion in GF(2^m). *IEE Electronic Letters*, 33(3):196–197, January 1997.
397. J. Zambreno, D. Nguyen, and A. Choudhary. Exploring Area/Delay Trade-offs in an AES FPGA Implementation. In *Proc. of Field Programmable Logic and Applications (FPL*, volume 3203 of *Lecture Notes in Computer Science*, pages 575–585. Springer-Verlag, 2004.
398. T. Zhang and K. K. Parhi. Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible Polynomials. *IEEE Transactions on Computers*, 50(7):734–749, 2001.
399. Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL A One-Way Hashing Algorithm with Variable Length of Output. In *ASIACRYPT '92: Proceedings of the Workshop on the Theory and Application of Cryptographic Techniques*, pages 83–104, London, UK, 1993. Springer-Verlag.
400. J. Y. Zhou, X. G. Jiang, and H. H. Chen. An Efficient Architecture for Computing Division over GF(2^m) in Elliptic Curve Cryptography. In *Proceedings of the 6th International Conference On ASIC, ASICON 2005*, volume 1, pages 274–277. IEEE Computer Society, October 2005.
401. D. Zibin and Z. Ning. FPGA Implementation of SHA-1 Algorithm. In *Proceedings of the* 5^{th} *International Conference on ASIC*, pages 1321–1324, Oct 2003.
402. J. zur Gathen and M. Nöcker. Polynomial and Normal Bases for Finite Fields. *J. Cryptology*, 18(4):337–355, 2005.

Glossary

Adittion Chains An *addition chain* for an integer $m-1$ consists of a finite sequence of integers $U = (u_0, u_1, \ldots, u_t)$, and a sequence of integer pairs $V = ((k_1, j_1), \ldots, (k_t, j_t))$ such that $u_0 = 1$, $u_t = m - 1$, and whenever $1 \leq i \leq t$, $u_i = u_{k_i} + u_{j_i}$. Addition chains are particularly useful for performing field exponentiation.

Area (hardware) Hardware resources occupied by the design. In terms of FPGAs, hardware area includes number of CLBs, memory blocks, IOBs, etc.

Authentication It is a security service related to identification. This function applies to both entities and information itself.

Block cipher A type of symmetric key cipher which operates on groups of bits of a fixed length, termed blocks.

BlockRAMs Built-in memory modules in FPGAs.

Brute force attack A brute force attack is brute force search for key space: trying all possible keys to recover plaintext from ciphertext.

Cipher A cipher is an algorithm for performing encryption and decryption.

Ciphertext An encrypted message is called ciphertext.

CLB Configurable logic block (CLB) is a programmable unit in FPGAs. A CLB can be reconfigured by the designer resulting a functionally new digital circuit.

Confidentiality It guarantees that sensitive information can only be accessed by those users/entities authorized to unveil it.

Configurable Soc (CSoC) CSoc integrates reconfigurable hardware, one or more processor and memory blocks on a single chip.

Confusion Confusion makes the output dependent on the key. Ideally every key bit influences every output bit.

Cryptographic Security Strength the Security strength of a given cryptographic algorithm is determined by the quality of the algorithm itself, the key size used and the block size handled by the algorithm.

Data Integrity It is a service which addresses the unauthorized alteration of data. This property refers to data that has not been changed, destroyed, or lost in a malicious or accidental manner.

Decryption The process of retrieving plaintext from ciphertext is called decryption.

Diffie-Hellman Key Exchange Protocol Invented in 1976 by Whitfield Diffie, Martin Hellman and Ralph Merkle, the Diffie-Hellman key exchange protocol was the first practical method for establishing a shared secret over an unprotected communication channel.

Difussion Diffusion makes the output dependent on the previous input (plaintext/ciphertext). Ideally each output bit is influenced by every input bit.

Discrete Logarithm Problem Given a number p, a generator $g \in \mathbb{Z}_p{}^*$ and an arbitrary element $a \in \mathbb{Z}_p{}^*$, find the unique number i, $0 \leq i < p-1$, such that $a \equiv g^i (mod\, p)$.

Downstream It defines the transmission from line terminal to network terminal (from customer to network premise).

Elliptic curve In mathematics, elliptic curves are defined by certain cubic (third degree) equations. They find applications in cryptography.

Elliptic curve cryptography Elliptic curve cryptography (ECC) is an approach to public-key cryptography based on the mathematics of elliptic curves.

Elliptic Curve Discrete logarithmic problem Let E_{F_q} be an elliptic curve defined over the finite field $F_q$and let P be a point $P \in E_{F_q}$ with primer order n. Consider the k-multiple of the point P, $Q = kP$ defined as the elliptic curve point resulting of adding P, $k-1$ times with itself, where k is a positive scalar in $[\![1, n-1]\!]$. The elliptic curve discrete logarithm problem consists on finding the scalar k that satisfies the equation $Q = kP$.

Elliptic curve scalar multiplication Let P be a point on Elliptic curve then the scalar product nP can be obtained by adding n copies of the same point P. The product nP = P + P+.....+ P obtained in this way is referred as elliptic curve scalar multiplication.

Encryption Encoding the contents of the message in such a way that it hides its contents from outsiders is called Encryption.

Extended Euclidean Algorithm In order to obtain the modular inverse of a number a we may use the extended Euclidean algorithm, with which it is possible to find the two unique integer numbers x, y that satisfy the equation, $ax + my = 1$.

FPGA A field-programmable gate array or FPGA is a gate array that can be reprogrammed, after it is manufactured.

Full Adder A full-adder is a combinational circuit with 3 input and 2 outputs. The inputs A_i, B_i, C_i and the outputs S_i and C_{i+1} are boolean variables. It is assumed that A_i and B_i are the ith bits of the integers A and B, respectively, and C_i is the carry bit received by the ith position.

The FA cell computes the sum bit S_i and the carry-out bit C_{i+1} which is to be received by the next cell.

Fundamental Theorem of Arithmetic Any natural number $n > 1$ is either a prime number, or it can be factored as a product of powers of prime numbers p_i. Furthermore, except for the order of the factors, this factorization is unique.

Granularity Granularity of the reconfigurable logic is defined as the size of the smallest functional unit that can be addressed by device programming tools.

Greatest common divisor Given two integers a and b different than 0, we say that the integer $d > 1$ is the greatest common divisor, or gcd, of a and b if $d|a$, $d|b$ and for any other integer c such that $c|a$ and $c|b$ then $c|d$. In other words, d is the greatest positive number that divides both, a and b.

HDL Hardware Description Languages (HDLs) are used for formal description of electronic circuits. They describe circuit's operation, its design, and tests to verify its operation by means of simulation. Typical HDL compilers tools, verify, compile and synthesize an HDL code, providing a list of electronic components that represent the circuit and also giving details of how they are connected.

Integer Factorization Problem Given an integer number n, obtain its prime factorization, i.e., find $n = {p_1}^{e_1}{p_2}^{e_2}{p_3}^{e_3} \cdots {p_k}^{e_k}$, where p_i is a prime number and $e_i \geq 1$.

Iterative Looping It implements only one round and n iterations of the algorithm are carried out by feeding back previous round results.

JTAG The Joint Test Action Group (JTAG) is the common name for the IEEE 1149.1 standard that defines the interface protocol between programmable devices and high-end computers.

Key schedule In cryptography, the algorithm for computing the sub-keys for each round in a block cipher from the encryption (or decryption) key is called the key schedule."

Logic Cell A logic cell is a very basic unit in FPGA which includes a 4-input function generator, carry logic, and a storage element (flip-flop).

Look Up Table A function generator in a logic cell is implemented as a look-up table which can be programmed to a desired Boolean logic, in addition, each look up table acts as a memory unit.

Loop unrolling It implements n rounds of the algorithm, thus after an initial delay, output appears at each clock cycle.

Message Digest A cryptograph hash function takes a message of an arbitrary length and outputs a fixed length string, referred to as message digest or hash of that message. The purpose of message digest is to provide fingerprint of that message.

Montgomery Multiplier In 1985, P. L. Montgomery introduced an efficient algorithm for computing $R = A \cdot B \bmod n$ where A, B, and n are k-bit binary numbers. The Montgomery reduction algorithm computes the resulting k-bit number R without performing a division by the modu-

lus n. Via an ingenious representation of the residue class modulo n, this algorithm replaces division by n operation with division by a power of 2.

Non-Repudiation It is a security service which prevents an entity from denying previous commitments or actions.

One Way Function Is an injective function $f(x)$, such that $f(x)$ can be computed efficiently, but the computation of $f^{-1}(y)$ is computational intractable, even when using the most advanced algorithms along with the most sophisticated computer systems.

One-way Trapdoor Function We say that a one-way function is a One-way trapdoor function if is feasible to compute $f^{-1}(y)$ if and only if a supplementary information (usually the secret key) is provided.

Permutation Permutation refers to the rearrangement of an element. In cryptography, elements (bit strings) are generally permuted in according to some fixed permutation tables provided by the algorithm.

Plaintext In cryptographic terminology, message is called plaintext.

Portable Digital Assistants(PDAs) PDAs are handheld small computers that were originally designed as personal organizers. PDAs usually contain note pad, address book, task list, clock and calculator, etc. Modern PDAs are even more versatile. Most of them are equipped with an Intel XScale μProcessor running at 400 MHz with up to 128MB of RAM memory.

Reconfigurable computing Denotes the use of reconfigurable hardware, also called custom computing.

Reconfigurable hardware Hardware devices in which the functionality of the logic gates is customizable at run-time. FPGAs is a type of reconfigurable hardware.

Stream cipher Stream ciphers encrypt each bit of the plaintext individually before moving on to the next.

Substitution Substitution refers to the replacement of an element with a new element. In cryptography, substitution operation is mainly used in block ciphers where an element is replaced with the elements from the substitution boxes called as S-boxes. The substituted values in some block ciphers can also be calculated.

System-on-Chip (SoC) SoC is a programmable platform which integrates many functions into a single chip. It may include analog as well digital components. A typical SoC includes one or more processing element (microcontroller/microprocessor or DSP), memory blocks, oscillators, analog to digital or digital to analog or both and other peripherals (counter timers, USB, Ethernet, power supply).

Throughput It is a measure for timing performance of a design and is calculated as: Throughput= (Allowed Frequency x Number of bits)/ Number of rounds (bits/s).

Upstream It defines the transmission from network terminal to line terminal (from network to customer premise).

Index

SIGNALS AND COMMUNICATION TECHNOLOGY

(continued from page ii)

Information Measures
Information and its Description in Science and Engineering
C. Arndt ISBN 3-540-40855-X

Processing of SAR Data
Fundamentals, Signal Processing, Interferometry
A. Hein ISBN 3-540-05043-4

Chaos-Based Digital Communication Systems
Operating Principles, Analysis Methods, and Performance Evalutation
F.C.M. Lau and C.K. Tse
ISBN 3-540-00602-8

Adaptive Signal Processing
Application to Real-World Problems
J. Benesty and Y. Huang (Eds.)
ISBN 3-540-00051-8

Multimedia Information Retrieval and Management Technological
Fundamentals and Applications D. Feng, W.C. Siu, and H.J. Zhang (Eds.)
ISBN 3-540-00244-8

Structured Cable Systems
A.B. Semenov, S.K. Strizhakov,and I.R. Suncheley
ISBN 3-540-43000-8

UMTS
The Physical Layer of the Universal Mobile Telecommunications System
A. Springer and R. Weigel
ISBN 3-540-42162-9

Advanced Theory of Signal Detection
Weak Signal Detection in Generalized Obeservations
I. Song, J. Bae, and S.Y. Kim
ISBN 3-540-43064-4

Wireless Internet Access over GSMand UMTS
M. Taferner and E. Bonek
ISBN 3-540-42551-9

Printed in the United States of America.