Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
Unidad Zacatenco

Departamento de Ingeniería Eléctrica
Sección de Computación

Implementación Eficiente de Algoritmos Criptográficos en Dispositivos de Hardware Reconfigurable †

Tesis que presenta:
Nazar Abbas Saqib

Para obtener el grado de:
Doctor en Ciencias

En la especialidad de:
Ingeniería Eléctrica

Opción:
Computación

Directores de tesis:
Dr. Arturo Díaz-Pérez
Dr. Francisco Rodríguez-Henriquez

Ciudad de México, México. 3 de Septiembre de 2004.

† Este trabajo fue parcialmente financiado mediante el proyecto CONACyT 31892-A: Algoritmos y arquitecturas de computadoras con dispositivos reconfigurables.

Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
Campus Zacatenco

Computer Science Section
Electrical Engineering Department

Efficient Implementation of Cryptographic Algorithms on Reconfigurable Hardware Devices †

By
Nazar Abbas Saqib

in partial fulfillment of the requirements for the degree of

Doctor of Science

Specialization in
Electrical Engineering

Option
Computer Sciences

Advisors:
Dr. Arturo Díaz-Pérez
Dr. Francisco Rodríguez-Henriquez

Mexico City, Mexico. September 3, 2004.

† This work was partially supported by CONACyT, project No. 31892-A: Computer algorithms and architectures with reconfigurable devices.

To my wife, **Afshan,**
for her devotions, sincerity, and solidarity for me.

To my daughter, **Fizza** (*7 years*) and **Ahmer** (*5 years*),
whose only presence give me a new sense to my existence.

To my **parents, brothers** and **sister**,
for their moral, financial, and everlasting support.

Acknowledgements

I thank to Ministry of Education, Islamabad, Pakistan for the award of Cultural Scholarship for pursuing doctorate in computer sciences at Mexico, a beautiful country in North America.

I thank to Mexican Government for a warm welcome and a regular financing during my doctorate. I would wish to thank all officials of Foreign Office Secretariat, Ministry of Foreign Affairs for their kind treatment and cooperation during my stay in Mexico.

I would acknowledge my advisor, Dr. Arturo Díaz-Pérez for his knowledge, his experience, and his guidance for this thesis work. I am thankful to him for arranging financial support in last months of my studies.

I would acknowledge my advisor, Dr. Francisco Rodríguez-Henriquez for his hard work and useful ideas for improving this dissertation work. I shall ever remember long discussions with him during days and nights for the clear conception and presentation of ideas.

I am personally thankful to my examiners for the revision of my thesis and for their valuable suggestions. I would like to thank Prof. Dr. Çetin Kaya Koç (Oregon State University, USA), Dr. Claudia Feregrino Uribe (INAOE, Mexcio), Dr. Guillermo Morales-Luna (CINVESTAV-IPN), Dr. Adriano de Luca Pennacchia (CINVESTAV-IPN) and my advisors who are also member of my examination committee.

I would like to thank all professors, officials, and colleagues of my department for their kind cooperation. I cannot forget Sofia Reza and Flor Córdova for their friendly talks and guidance for academic matters. I thank all officials in academic services and library officials for their nice treatment.

RESUMEN

Conforme nos movemos hacia una sociedad de información, la seguridad se ha convertido en un asunto crucial en áreas como la industria, los negocios y la administración. Las técnicas básicas que se requieren para proteger la información pertenecen al campo de la criptografía. La criptografía aplicada a la seguridad es una herramienta importante para asegurar confidencialidad (en la transmisión y almacenamiento de la información), integridad (no hay cambio que pueda no ser detectado), identificación de fuente (el que envía puede ser identificado), y no repudiación (el que envía no puede negar que envió un mensaje).

Los algoritmos criptográficos están clasificados en dos categorías: algoritmos de llave secreta y algoritmos de llave pública. Los algoritmos de llave secreta o simétrica utilizan una llave secreta para cifrar o descifrar los mensajes. Algunos ejemplos de este tipo de algoritmos son: DES, AES, Serpent, MARS y IDEA. En los algoritmos de llave pública o asimétrica, las llaves se organizan en pares: llave pública y llave privada. Cualquiera puede utilizar la llave pública para cifrar un mensaje, sin embargo, solo quien posee la llave privada puede descifrar este mensaje. RSA y Criptografía de Curvas Elípticas (ECC por sus siglas en inglés) son esquemas criptográficos populares de llave pública. Los algoritmos de llave secreta son rápidos y pueden ser usados para cifrar grandes cantidades de datos. Por otra parte, los algoritmos de llave pública son computacionalmente más demandantes que los algoritmos simétricos pero evitan la necesidad que exista previamente un secreto entre dos objetos que quieren comunicarse.

Los algoritmos criptográficos pueden ser implementados en plataformas de software y hardware. Las soluciones criptográficas basadas en software, pueden ser usadas para aplicaciones de seguridad donde el tráfico no es muy demandante y la velocidad de encriptación no es muy alta. Por otro lado, los métodos por hardware ofrecen soluciones veloces para aplicaciones donde el tráfico de datos es más intenso y la gran cantidad de datos requiere una encriptación en tiempo real. Los circuitos VLSI, y los dispositivos FPGAs (Field Programmable Gate Arrays) son dos alternativas para implementar algoritmos criptográficos en hardware. Los FPGAs ofrecen

grandes beneficios para la implementación de algoritmos criptográficos al compararlos con las soluciones VLSI por su alta flexibilidad. Debido a que los FPGAs tienen la propiedad de ser reconfigurables, las llaves se pueden cambiar rápidamente. Más aún, las primitivas básicas de la mayoría de algoritmos criptográficos pueden ser eficientemente implementadas en FPGAs.

El objetivo principal de esta tesis, es obtener implementaciones de algoritmos criptográficos basadas en dispositivos reconfigurables como los FPGAs que tengan un alto desempeño sin tener que utilizar altos requerimientos de hardware. Esto es, el objetivo es encontrar un balance adecuado entre el espacio requerido por los circuitos y la rapidez con que se pueden realizar las operaciones de cifrамiento y desciframiento. Para cumplir el objetivo de esta tesis se han elegido tres algoritmos de acuerdo a su importancia en aplicaciones de seguridad: Data Encryption Standard (DES) y Advanced Encryption Standard (AES) como algoritmos de llave simétrica, y Criptografía de Curva Elíptica como algoritmo de llave asimétrica.

En primer lugar, se presenta la aritmética sobre campos finitos GF(2^m) dado que constituye la base teórica para el desarrollo de los algoritmos criptográficos elegidos en este trabajo de tesis. Más adelante, se abordaron metas específicas para cada algoritmo.

Se desarrollaron guías generales para implementar cifradores de bloque de llave simétrica en plataformas reconfigurables. Se presenta la estructura general y los principios del diseño para cifradores de bloque y se identifican las primitivas básicas en este tipo de algoritmos. Se presentan algunas técnicas útiles de diseño para obtener implementaciones eficientes en dispositivos reconfigurables. Para aplicar tales guías y técnicas se utilizó a DES como caso de estudio y se obtuvo una implementación rápida y compacta para este algoritmo.

Por otra parte, se exploraron varias alternativas arquitecturales para la implementación de AES en dispositivos reconfigurables. Tales alternativas se presentan como opciones convenientes para diversas aplicaciones de seguridad. Las arquitecturas diversas desarrolladas para AES fueron optimizadas para obtener alto desempeño, bajo costo o soluciones altamente portables. La eficiencia de los diseños se obtuvo mediante la aplicación de técnicas optimizadas de diseño y aplicando algunas transformaciones a los algoritmos originalmente planteados.

Finalmente, se presenta una arquitectura genérica para realizar la multiplicación escalar en curvas elípticas, la operación más importante de ECC. Tal arquitectura fue optimizada para dispositivos reconfigurables (FPGAs) tomando ventaja del máximo nivel de paralelismo que se puede explotar en la multiplicación escalar y mediante un uso eficiente de loa recursos de hardware. Combinando las operaciones de la aritmética de campos finitos en GF(2^m) y de la aritmética de curvas elípticas, se formaron bloques básicos para realizar la multiplicación escalar en curvas elípticas.

La característica principal entre todos los diseños desarrollados para hardware reconfigurable es el uso de técnicas paralelas para realizar las operaciones básicas de los algoritmos y, de esta manera, reducir el retraso de la ruta crítica del circuito. El consumo bajo de recursos se obtuvo identificando las operaciones comunes en

diferentes pasos y reutilizando los bloques básicos. Por otra parte, se buscó hacer un mapeo adecuado a la estructura del dispositivo reconfigurable seleccionado para obtener diseños optimizados para FPGAs. Los resultados mostraron que se obtuvieron diseños para algoritmos criptográficos con un alto desempeño mediante un uso eficiente de los recursos de hardware los cuales son comparables con implementaciones reconfigurables similares que están reportadas en la literatura a la fecha.

ABSTRACT

As we move into an information society, information security has become a crucial issue in industry, business, and administration. The techniques needed to protect information data belong to the field of cryptography. It is an important tool in assuring confidentiality (in transmission or storage of information), integrity (no change can be made undetectably), source identification (the sender can be identified and all other than that sender can be excluded), and non-repudiation (the sender should not be able to deny sending the message).

Cryptographic algorithms fall into two categories: secret and public key algorithms. Secret or symmetric key algorithms use a secret key for encrypting or decrypting a message. DES, AES, Serpent, MARS, and IDEA are few examples of symmetric algorithms. Public or symmetric key algorithms involve a pair of keys: a public key and a private key. Anyone can use a public-key to encrypt a message, however, only a private key holder can decrypt that message. RSA and Elliptic Curve Cryptography (ECC) are popular public key cryptographic schemes. Secret key algorithms are computationally faster and can be used for encrypting large data. Public-key algorithms are computationally more intensive than symmetric algorithms.

Cryptographic algorithms can be implemented in software and hardware platforms. Cryptographic solutions using software methods can be used for those security applications where data traffic is not too large and low encryption rate is tolerable. On the other hand, hardware methods offer high-speed solutions making them highly suitable for applications where data traffic is fast and large data is required to be encrypted in real time. VLSI (also known as ASIC), and FPGAs (Field Programmable Gate Arrays) are two alternatives for implementing cryptographic algorithms in hardware. FPGAs offer several benefits for cryptographic algorithm implementations over VLSI as they offer high flexibility. Due to its reconfigurable property, keys can be changed rapidly. Moreover, basic primitives in most cryptographic algorithms can efficiently be implemented in FPGAs.

The main goal of this thesis is to achieve high-speed implementations of cryptographic algorithms on reconfigurable hardware devices without posing high re-

quirements for hardware resources. That is, to achieve a right balance between required space (hardware area) for circuits and high-speed for performing encryption/decryption operations. To complete this goal, three algorithms were chosen for their importance in security applications: Data Encryption Standard (DES), Advanced Encryption Standard (AES) from symmetric key and Elliptic Curve Cryptography from public key cryptography.

First, basic concepts of finite field arithmetic in GF(2^m) are presented constructing a theoretical base for understanding of selected cryptographic algorithms in this thesis work. Then, specific goals have been set for each algorithm.

A general guideline for implementing block ciphers in reconfigurable platform is provided. General structure and design principles for block ciphers are discussed. Basic primitives in block ciphers are identified and some useful design techniques were devised for efficient implementations on reconfigurable devices. For applying said guideline and techniques, DES was taken as a case of study producing a fast and compact reconfigurable architecture for this algorithm.

On the other side, multiple architectural options for AES reconfigurable implementations are explored as an effort to determine the suitable candidate for diverse security applications. Distinct AES architectures are optimized for high speed, low cost and highly portable solutions to cryptographic applications. High design performances were achieved by applying optimizing techniques for designing and by modifying standard transformations of the algorithms.

Finally, a generic architecture for implementation of elliptic curve scalar multiplication (a most important operation in ECC) is presented which is further optimized for reconfigurable hardware devices. Key feature in ECC implementation is to exploit maximum parallelism making an efficient use of hardware resources. This approach is adopted for finite field arithmetic in GF(2^m) and elliptic curve arithmetic, which form basic building blocks for elliptic curve scalar multiplication.

The key feature among all reconfigurable architectures is the use of parallel techniques for performing basic operations in cryptographic algorithms and in this way, reducing delays occurred in critical path of the circuit. Low hardware area utilization was obtained by identifying common operation in different steps and by reusing basic building blocks. On the other hand, an accurate matching to the structure of the selected device was made producing optimized designs for FPGAs. Our results show high performance architectures for cryptographic algorithms through an efficient use of hardware resources as compared to existing similar reconfigurable implementations reported in the literature to-date.

Table of Contents

List of Tables

List of Figures

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
BRAMs	Block Rams
BS	Byte Substitution
CLB	Configurable Logic Block
CPU	Central Processing Unit
DES	Data Encryption Standard
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
ECDLP	Elliptic Curve Discrete Logarithmic Problem
ETSI	European Telecommunications Standards Institute
FLT	Fermat's Little Theorem
FPGAs	Field Programmable Gate Arrays
GSM	Global System for Mobile Communications
HDLs	Hardware Description Languages
IAF	Inverse Affine Transformation
IARK	Inverse Add Round Key
IBS	Inverse Byte Substitution
IL	Iterative Looping
IMC	Inverse Mix Column
IOBs	Input/Output Blocks
IPSec	Internet Protocol Security
ISO	International Organization for Standardization
ISR	Inverse ShiftRow
ITMIA	Itoh-Tsujii Multiplicative Inverse Algorithm
ITU	International Telecommunication Union
LAN	Local Area Network
LC	Logic Cell

LUT	Look-Up Table
MAN	Metropolitan Area Network
MC	MixColumn
MI	Multiplicative Inverse
ModM	Modification for MixColumn
NIST	National Institute of Standard Technology
PDAs	Personal Digital Assistants
Soc	System-on-Chip
SR	Shift Row
VDSL	Very high speed Digital Scriber Lines
VHDL	Very high speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integrated circuits
WAP	Wireless Application Protocol

Chapter I

Introduction

This chapter presents a complete outline for this thesis. It introduces the basic concepts of cryptography, fundamental operations in cryptographic algorithms and some important cryptographic applications in the industry. Alternatives for the implementation of cryptographic algorithms on various software and hardware platforms are discussed. This chapter explains also the research goals for this thesis, the chosen methodology to achieve those goals, and a summary of our contributions on this thesis work. The dissertation presentation is provided at the end.

I.1 Introduction

In this information age, the need for protecting information is more pronounced than ever. Secure communication for the sensitive information is not only compelled to military or government institutions but also to the business sector and private individuals. Already the exchange of sensitive information, such as bank transactions, credit card numbers over the Internet and telecommunication services are now common practices. As the world becomes more connected, the dependency on the electronic services has been amplified. For protecting data in computer and communication systems from unauthorized disclosure and modification, non-interceptable means for data storage and transmission must be adopted. A cryptographic cipher system is one such possible system, which 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.

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication [56], which can be defined as follows:

- **Confidentiality:** It is a service used to keep the content of information from all but those authorized to have it. When two or more parties are involved

in a communication, the purpose of confidentiality is to guarantee that only those 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 an unauthorized 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 hides its contents from 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 recent cryptographic applications. For several crucial applications, a combination of both the secret and public key methods is indispensable.

I.1.1 Secret key cryptography

Secret or symmetric key cryptography makes use of the same key for encryption and decryption (or the decryption key is easily derived from the encryption key) as shown in Figure I.1.

Both encryption and decryption keys (or sometimes same keys) are kept secret and must be known at both ends to perform encryption or decryption. Symmetric algorithms are fast and are used for encrypting/decrypting high volume data. All of the classical (pre-1970) cryptosystems are symmetric. The most popular symmetric cryptosystem is DES (Data Encryption Standard) which is widely used as TripleDES (a variation of DES) in the world. On October 2000, a new symmetric cryptographic

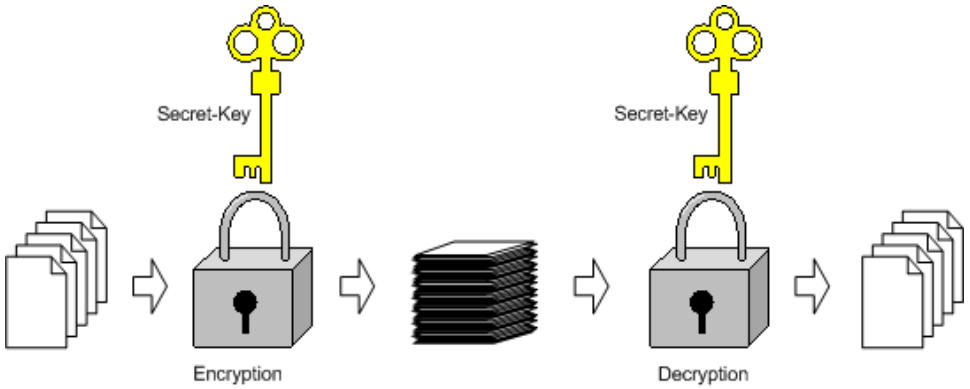


Figure I.1. Secret key cryptography

algorithm 'Rijndael' was chosen as a new Advanced Encryption Standard (AES) [24] by NIST (National Institute of Standards and Technology) [65]. Due to its enhanced security level, it is replacing DES and TripleDES in a wide range of applications. Symmetric algorithms can be divided into 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. The official block length for Rijndael is 128 bits however the algorithm can process block sizes 192 and 256 bits. Other famous block ciphers are Serpent, RC6, MARS, IDEA, Twofish etc.

I.1.2 Public key cryptography

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 the Figure I.2.

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 element to the 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 messages and only the corresponding private key can be used to decrypt them. The most popular public-key algorithms are RSA (based on factoring large numbers), ElGamal (based on

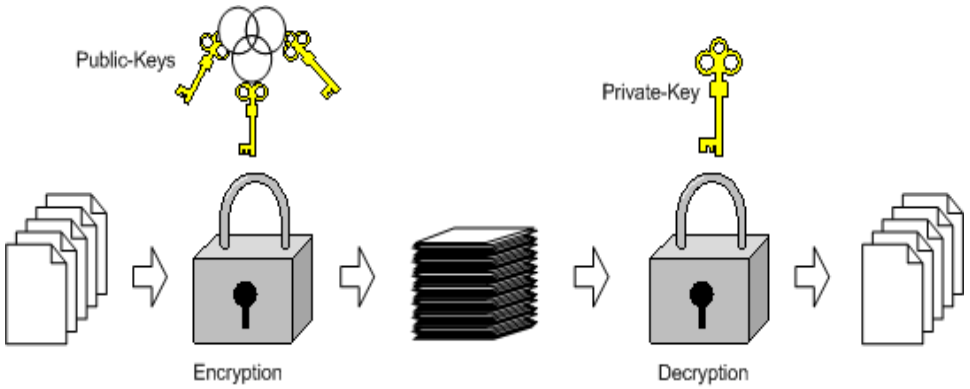


Figure I.2. Public key cryptography

discrete log problem) and McEliece (based on error correcting codes). 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.

I.2 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 type 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 to encrypt large data files and are applied to other important cryptographic applications like key exchange, signature, verification etc. A detail survey has been conducted by [21] to identify the basic operations involved in several cryptographic algorithms. That survey has been slightly updated as shown in Table I.1.

From Table I.1, it is clear that most cryptographic algorithms mainly include bit-wise operations like XOR, AND/OR etc. These operations provide simple logic in general on all the platforms. However they are well suited for their implementation on hardware platforms. Long word length for the cryptographic algorithms is another characteristic, which is recommended by various international standards to attain sufficient security against brute force attacks. Brute force attacks are based on exhaustive search for all possible key combinations to determine the original key. As a simple example, a 4-bit word can provide a maximum of $2^4 = 16$ combinations. The

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

Table I.1. Primitives of cryptographic algorithms (symmetric ciphers)

recommended word length for the modern block ciphers is no less than 80-bits [65]. The new Advance Encryption Standard (Rijndael) can support a word length of 128, 192 and 256 bits. For public key cryptographic algorithms like Elliptic curve cryptography, the minimum key length is 160 bits according to the recommendation by the international standards [14, 73]. For RSA a 1024-bit key length is normally used. The long key/world length of cryptographic algorithms restrict parallel flow of the data 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 either input or output allowing high parallelism of data. 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 a parallel way ensuing significant improvements in timings. Those features in cryptographic algorithms well suit for their implementations on hardware platform resulting high-speed architectures for them.

I.3 Potential cryptographic applications

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 busi-

ness groups. A wide range of applications including networking, e-commerce, and telecommunication, must be protected against security risks due to heavy losses of useful information. It is a great risk now to send valuable information on insecure/open channels. Big industrial organizations including Microsoft, IBM, Motorola and Hewlett Packard have been integrating security modules in their products. A few examples of the security products in use by the public sector are Secure telephones: *Cipher*TAC 2000 by Motorola [63], CSD 4100 by TCC [96], Video conferencing: Sony PCS-1, INTROFIGC-II [93], Polycom VSX 3000, V500 [1], Cellular phone: Nokia 6225, 6600 [2], Ericcson T310, T230, T610 [3]. Various international organizations have been working in developing standards for determining security and speed of those products. Examples include standards for video conferencing: H310, H323, H324 by ITU [4], for mobile communications: GSM by ETSI [5], for wireless LANs: 802.11a, 802.11b by IEEE LAN/MAN Committee [6]. Numerous useful activities for the strength of security using cryptographic algorithms can be observed in the world. Fast and strong cryptographic algorithms have been developed. The selection of the new Advance Encryption Standard (AES) 'Rijndael' and the inclusion of Elliptic curve cryptography (ECC) in international standards provide such examples.

Potential applications for cryptographic algorithms can be classified into two categories [64].

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, choice of specific item to order, order placement with signature, bank account information extraction, making e-payment, 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 I.2. These are so called 'highly efficient' application requiring high data rates. Table I.2 presents both the downstream and upstream data transfer ranges on VDSL (Very high speed Digital Subscriber Line) [7, 12]. 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 I.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 highly efficient applications ranges from 384Kbps to 24Mbits for upstream and 64Kbps to 3Mbps for the downstream traffic. From Table I.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

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

Table I.2. A few potential cryptographic applications

algorithms or using 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. High speeds above 400Kbps can easily be achieved on hardware platforms. The implementations of cryptographic algorithms on both, the classical (ASICs) and reconfigurable (FPGAs) types of hardware platforms can achieve a throughput factor in the order of Gbits/s.

I.4 Alternatives for implementing cryptographic algorithms

The implementation approaches for cryptographic algorithms are based on the question: what needs to be secured? Low speed or moderately high speed network where very small amount of bit traffic is to be processed unpredictable and in real time, can be implemented in software. Usually this is done by developing software packages for Pentium class PCs. Cryptographic products in software by Certicom [20] for the protection of e-mail (movianMail) and data storage for mobile devices (movianCrypt) are such examples. For small sized applications like wireless devices, the implementation can be made in the device's main CPU, usually via some single-on-chip (SoC) variation around an ARM (ARM1026EJ-S, ARM926EJ-S, etc from ARM Limited) or MIINTROFIG (MIINTROFIG64, MIINTROFIG32, etc. from MIINTROFIG Technologies, Inc.) processor. If the information needs to be encrypted by 3DES or by Rijndael, it can be done in software using embedded processors. The encryption routine will be used in a small part of the whole day; it will not affect the economics and speed of the design.

High-speed network where large amount of data traffic is to 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 line speeds and must be treated in real time. Examples of such situation include telephone conversation, video confer-

encing, streaming audio or encoded video transmissions and so forth. VLSI solutions might be a suitable choice if the architecture prevent buffer overflow performing full processing before the next incoming piece of data is presented at the input. VLSI implementations on the other hand take considerable time from high-level specifications to final prototype resulting high fabrication cost although cost per unit is low.

The big challenge is in the design domain where the data flow rate is significant, the available time to get the job done is short, and the available computational resources are limited. Software on general-purpose processors (CPU or DSP) would not solve the problem. If software is not the solution the alternative is hardware. 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. The design size becomes a bottleneck when it does not fit for small sized applications like handheld devices. However, a hardware platform, which overcomes the difficulties of VLSI solutions and accommodates high data flow in considerable short time, is reconfigurable hardware. Reconfigurable devices like FPGAs (Field Programmable Gate Arrays) provide fast solutions in short time with high degree of flexibility and low cost. Table I.3 presents a quick comparison of reconfigurable logic to software and VLSI based solutions. Software

	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	not too low
Testing/Verification	easy	difficult	easy
Run-time configuration	none	none	yes

Table I.3. Comparison between Software, VLSI, and FPGA platforms.

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 the rest of this subsection a short introduction to reconfigurable computing is presented followed by its advantages/disadvantages for implementing cryptographic algorithms.

I.4. ALTERNATIVES FOR IMPLEMENTING CRYPTOGRAPHIC ALGORITHMS

I.4.1 Reconfigurable computing

Reconfigurable computing (custom computing) denotes the use of reconfigurable hardware to speed-up execution of software. Field programmable gate arrays (FPGAs) are considered reconfigurable devices as they can be configured at run time. In other words, the design of the hardware may change in response to the demands placed upon the system while it is running. Faster programming times permit dynamic design swapping: a single FPGA performs a series of tasks in rapid succession, reconfiguring itself between each one. Such designs operate on a time sharing mode and swap between successive configurations so rapidly that it appears as FPGA is performing all its functions at once.

An FPGA is an integrated circuit that belongs to a class of programmable devices and consists of thousands of building blocks, known as *Configuration Logic Blocks*(CLB) connected through programmable interconnections as shown in Fig. I.3. These CLBs can be reconfigured by the designers themselves resulting a functionally new digital circuit. A CLB can be configured into two modes: logic mode and

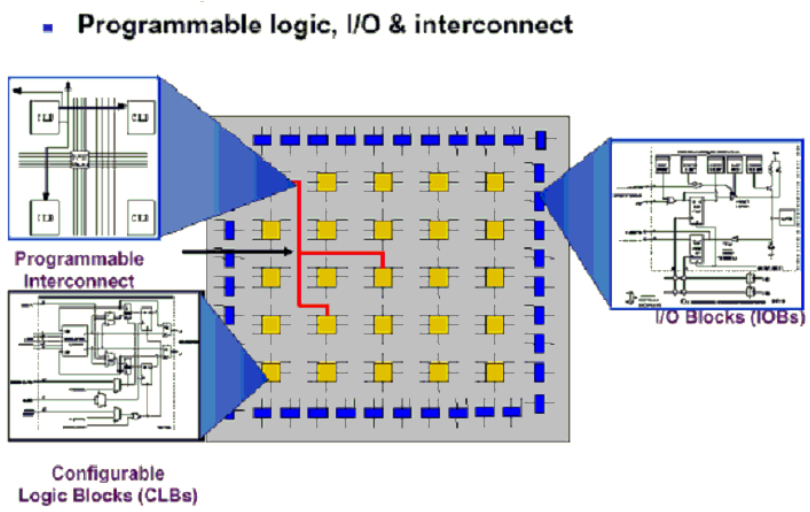


Figure I.3. Basic architecture of an FPGA

memory mode. As shown in Fig. I.4, in logic mode, each of these elementary units consists of a combinational logic block and a one bit register. Those combinational logic blocks can be reprogrammed to any combinational logic function like XORs, ANDs, etc. In memory mode, combinational logic is replaced with two small piece of memory blocks. Normally, each CLB contains two small functional units (slices) but strictly speaking its exact architecture depends upon the family of FPGA devices. Modern FPGAs offer four slices packed into one CLB. The functionality and quantity of a CLB are defined by the manufacturer specifications. There exist various families and manufacturers of FPGA's devices, the most famous are Virtex, Spartan (Xilinx), FLEX, APEX (Altera), ACT (Actel), pASIC (QuickLogic), LCA

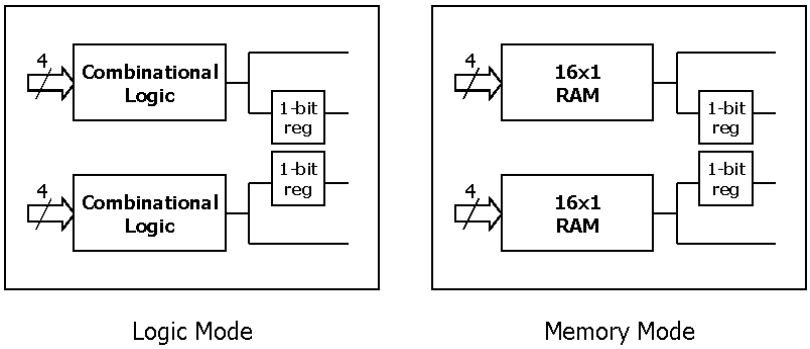


Figure I.4. CLB configuration modes

(Logic Cell Array) and ORCA (Lucent). In recent years, huge developments have been shown in FPGA technology. Modern FPGAs operate at more than 400 MHz internal clock with a gate complexity of over than 8 Million gates in a single chip FPGA (Virtex-II). The improvements in technology are not only limited to increase number of logic gates but also to the addition of many functional blocks like fast access memories, multipliers and even some FPGA's families include PowerPCs integrated with the single chip. The Virtex-II family of devices for example, include up to four IBM processors @125MHz, up to 8 Mbits embedded memory, and up to 168 (18×18) multipliers.

I.4.2 Advantages/disadvantages of reconfigurable computing

Using reconfigurable hardware for cryptographic algorithms is beneficial in several ways:

- Most of the cryptographic algorithms especially symmetric ciphers contain bit-wise logic operations well suited to FPGA CLB structure.
- In Section I.2, it was mentioned the iterative nature of most cryptographic algorithms. An iterative looping design (IL) implements only one round and 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 are replicated and registers are provided between the rounds to control the flow of data, called *loop unrolling* or *pipeline* (PP) approach. Reconfigurable FPGA logic is 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 make use of lot of memory resources. Using pipeline design strategies, the memory requirements become significant. Fortunately modern FPGA families like Virtex series device are equipped with more than 280 built-in memory blocks 4K each, called *BlockRams* (BRAM). A dual port BRAM

can be configured into two single port BRAMs with independent and fast data access, that is, read or write operations can be executed independently at each port. Only the simultaneous write operation at the same location is prohibited.

- At the same time, in several contexts, designers 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 is a practical solution to this problem.
- FPGA devices are ideal for debugging of a design, specially if the synthesized hardware description can be mapped by the design team from an FPGA domain to ASIC.
- The ease of integration into larger platform and the straightforward modification in the architecture are worthwhile advantages over FPGA platform, especially in government communications where versatility, flexibility and functionality in many cases are of primary importance, as opposed to cost or power consumption.

Reconfigurable logic offers numerous useful advantages, however use of FPGA devices in inexpensive consumer-oriented devices like wireless PDAs and handsets seems to be difficult at this moment. The physical size, high power consumption and expansive price tag are some of the distressing elements. Some recent models however can easily be integrated into very high seed communication systems. FPGA devices obviously can be contemplated on large wireless systems, big (transportable or not) transmitters and receivers, repeaters, spectrum scanning devices, and intelligent equipment.

I.5 Research goals

The choice of reconfigurable logic as a target platform for the implementations of cryptographic algorithms appears to be a practical solution for small sized and high-speed applications. One of the reasons could be the well suitability of the basic operations in cryptographic algorithms on FPGA structure. It was therefore planned to look for the possibilities of high-speed cryptographic solutions on reconfigurable hardware platform. Theoretically, it looks feasible. This thesis however investigates practical issues of cryptographic solutions. High-speed solutions on cryptographic applications where speed factor is indispensable and the cost is of secondary concern. Acceptable cryptographic solutions to those applications where high speed is desired but cost is also an important factor. Low cost cryptographic solutions where speed factor is not so strict. Both efficient and cost effective solutions of cryptographic algorithms are desired on reconfigurable logic platform. The term 'efficient' is normally

referred to 'high speed' solutions. In this thesis however it is not only meant a high speed but also a low area (hardware resources) solution. The problem is therefore to find high speed and low area implementations of cryptographic algorithms using reconfigurable logic devices. That implies careful considerations to cryptographic algorithms whether they are from secret or public key domain, which often will lead to suggest/modify the standard transformations of those algorithms. That also implies a deep 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 from its high level specifications to physical prototype is carefully examined.

Efficient implementations of cryptographic algorithms on reconfigurable hardware was the main goal of this research. A set of three of the most important cryptographic algorithms from both symmetric and asymmetric cryptography had been selected for their implementation on reconfigurable logic devices. Those algorithms are Data Encryption Standard (DES), Advance Encryption Standard (AES) from symmetric ciphers and Elliptic Curve Cryptography (ECC) from asymmetric cryptosystems. Several considerations were made to achieve high speed and economical (using minimum hardware resources) implementations of those algorithms on reconfigurable logic platform. One of them was to exploit high bit-level parallelism where and whenever it was possible. Similarly, good designing techniques well mapped to the structure of the target devices and the features of the design tools for optimizing the circuit for speed and area, were also addressed in this thesis. Some specific tasks for the three chosen cryptographic algorithms are described below.

DES was the first algorithm studied in this thesis. The basic primitives involved in block ciphers specifically for DES were analyzed for their implementations on reconfigurable logic platform. A high-speed one round FPGA implementation of DES was carried out exploiting high bit-level parallelism. The experiments were made for optimizing an FPGA architecture with respect to both hardware area and speed. Expertise in design tools and techniques were the other benefits for implementing other cryptographic algorithms on reconfigurable devices.

A more detailed study was planned regarding AES due to its importance for the current security needs in the industrial and non-industrial units. 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. The encryptor/decryptor cores using both the look-up table method and on-fly architecture schemes were optimized for the FPGA implementation of AES. It was attempted to reduce the critical paths for encryption/decryption by sharing common resources or optimizing the standard transformations of the algorithm. Several high speed and economical AES encryptor and encryptor/decryptor cores were the contributions for this thesis.

ECC is a public key cryptographic scheme, which is widely accepted by aca-

demic and industrial organizations for its shorter key length as compared to others contemporary algorithms providing the same security level. A three-layer model for ECC implementation is normally adopted. First layer deals with the implementations of finite field arithmetic in GF(2^m): squaring, multiplication, and reduction in GF(2^m). Second layer makes use of those building blocks to implement elliptic curve arithmetic in GF(2^m): 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. In this thesis, ECC implementation on reconfigurable logic platform was addressed using parallel techniques for all the three layers. Efficient building blocks were obtained for finite field arithmetic in GF(2^m) on FPGAs. Some parallel strategies were devised for the best use of those building blocks to execute point addition and doubling operations. A generic architecture for the elliptic curve scalar multiplication was proposed and implemented on FPGA platform. Achieved results improve previous FPGA implementations of elliptic curve scalar multiplications.

I.6 Methodology

Achieving high-speed implementations for cryptographic algorithms is an exciting job requiring deep considerations at every stage of the design. Our methodology for this investigation is therefore not only based on the best implementing techniques on reconfigurable platform but also in contributing on the theoretical side by improving the standard transformations of cryptographic algorithms. The key idea in this methodology is to adopt parallel approaches while designing or working on the algorithms occupying minimum possible hardware resources for the designs. Important considerations in our methodology to achieve high speed and low area architecture for cryptographic algorithms are:

I.6.1 Parallelism at algorithm level

Secret-key ciphers provide standard transformation for encryption/decryption. One of the crucial points in our methodology is to search for those operations, which can be executed in a parallel way. There is a possibility to modify the standard transformation of the algorithms. It is also possible to manipulate the standard transformations using different methods. The manipulation of AES S-Box [24] in GF$(2)^4$ instead of GF$(2)^8$ is one example. In public-key ciphers most common operations are squaring, doubling, multiplication or inversion. Those arithmetic operations are normally not bound to a particular algorithm. A general or fast algorithm could be adopted or modified by the designer.

I.6.2 Parallelism at design level

The design of a cryptographic algorithm could be more efficient when it is optimized to a special platform (VLSI or FPGA) or a special device (Virtex, Spartan devices,

etc.). Number of design tools in the market now offers multiple platform selection (VLSI or FPGA). The circuit designed using any method (hardware description language, etc.) can be mapped to VLSI or FPGAs (Synopsis 7.0). However the design optimization to a particular platform either VLSI or FPGA might produce good results. For example, each Virtex CLB consists of two functional units with 4-input/1-output configuration as shown in Fig. I.4. That shows an optimal use of CLB slices if the design logic is well mapped to 4-input/1-output configuration mode. The expression $Z = A \oplus B \oplus C \oplus D$ consists of four inputs A, B, C, D and one output, Z. The same expression can be manipulated by combination of two inputs i-e $X = A \oplus B$, $Y = X \oplus C$, and $Z = Y \oplus D$. The output is the same for both of them, however, first expression well maps to CLB structure and occupies just half slice as compared to second expression, which utilizes one and half slices. These peculiarities have no importance in VLSI as 2,3, or 4-input/1 output gates do not occupy equal resources.

I.6.3 Design strategies

Various design strategies are used for the implementation of a typical secret-key cipher. 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 I.5a. It utilizes less area but consumes more clock cycles resulting on a relatively low speed encryption. Architecture with loop unrolling is shown in Figure I.5b. 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 are too high. Figure I.6a shows an inner-

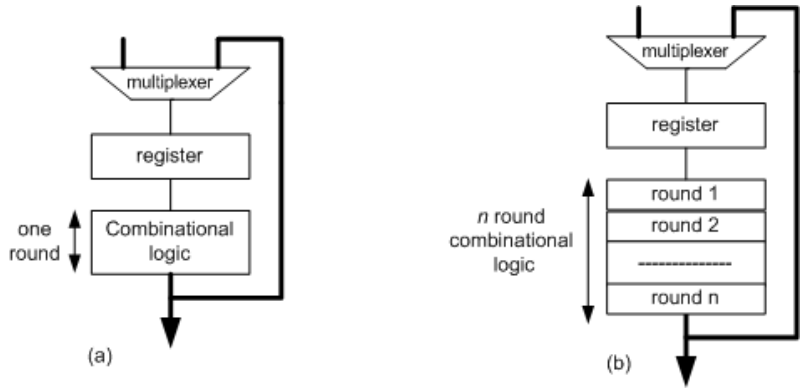


Figure I.5. Basic architectures for (a) iterative looping (b) loop unrolling

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. High speed circuit results at the cost of more hardware resources in the form of registers. Outer-round pipelining is created by loop

unrolling by adding extra registers at different stages of the same round as shown in Figure I.6b. It enables to directly trade circuit speed with the circuit area.

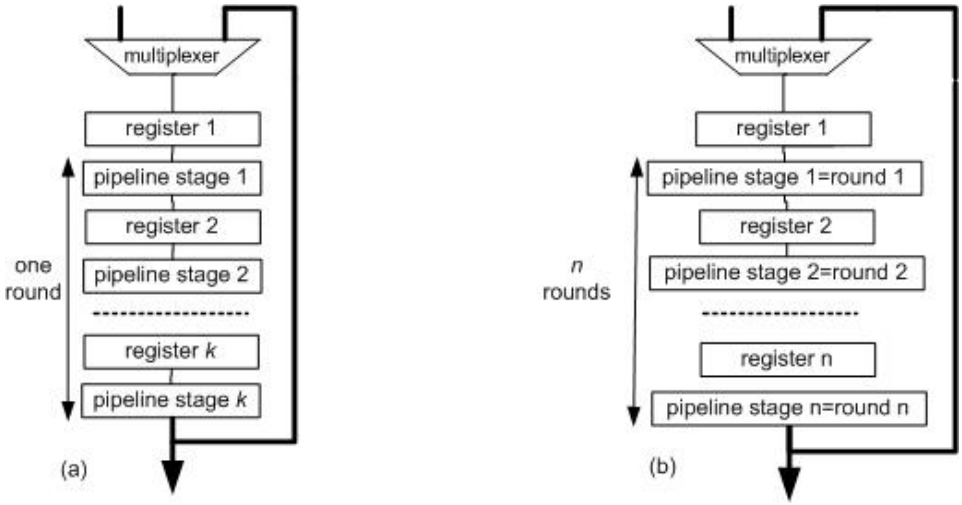


Figure I.6. Round-pipelining for (a) one round (b) n rounds

It was already pointed out that high-speed and economical circuits (less area) would be targeted to this investigation. 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 setup. It is possible to exploit parallelism at the different stages of the hierarchal model.

I.6.4 Design tools

An intelligent use of the design tools is considered useful in our methodology for better performances. The design tools (Xilinx Foundation Series, Synopsis, etc.) provide several useful features for the improvements of the design. Better placement of the components or better routing of the tracks can obviously be helpful in cutting net delays in the circuit. This is the reason that some of the design tools in the market claim to provide more efficient circuits.

I.7 Design statistics of an FPGA architecture

Although the basic structure of FPGAs is the same, FPGA manufacturers provide diverse features for their devices. Those features include 1, 2 or 4 slices within a CLB, built-in additional logic or memory modules. It will be therefore useful to understand the architectural description of the target device. In this thesis, Virtex series devices are mainly used. The architectural description is therefore provided for Virtex family of devices. That will help in defining the basic parameters used for describing the performance of an FPGA architecture.

I.7.1 Architectural description of the target device

FPGAs comprise two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs). CLBs provide the functional elements for constructing

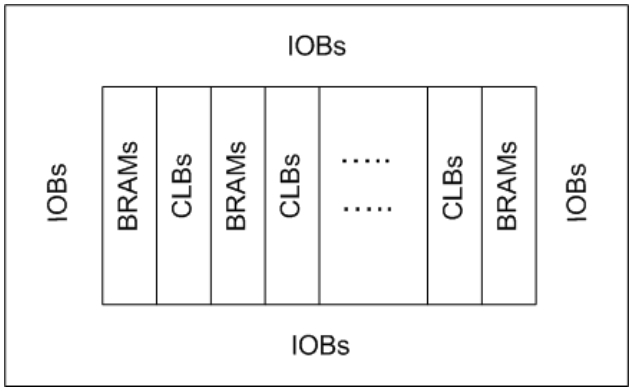


Figure I.7. VirtexE architecture overview.

Logic and IOBs provide the interface between the package pins and the CLBs. FPGAs come from different vendors but the basic CLB structure remains the same. High density FPGAs contains large number of CLBs and IOBs. Some families of FPGA devices offer additional logic such as built-in multipliers and memory modules and even some of them include power microprocessors. We used Virtex-E devices for our FPGA implementations in this thesis. A brief discussion is therefore provided explaining the internal structure for Virtex-E family of devices. It would be helpful for the understanding of design statistics for an FPGA architecture.

An overview of VirtexE architecture is shown in Figure I.7. As shown in Figure I.7, VirtexE FPGAs occupy a separate space for BRAMs in addition to the space occupied by CLBs and IOBs. BRAMs (Block Selected RAMs) are built-in memory modules in Virtex Series devices. Virtex series devices contain more than 280 BRAMs, each of 4096 bits.

The basic building block of the Virtex-E CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and a storage element (flip-flop) as shown in Figure I.8.

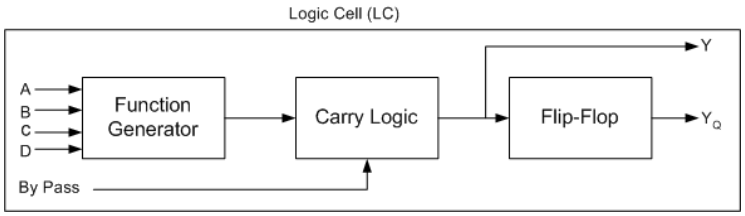


Figure I.8. VirtexE Logic Cell (LC).

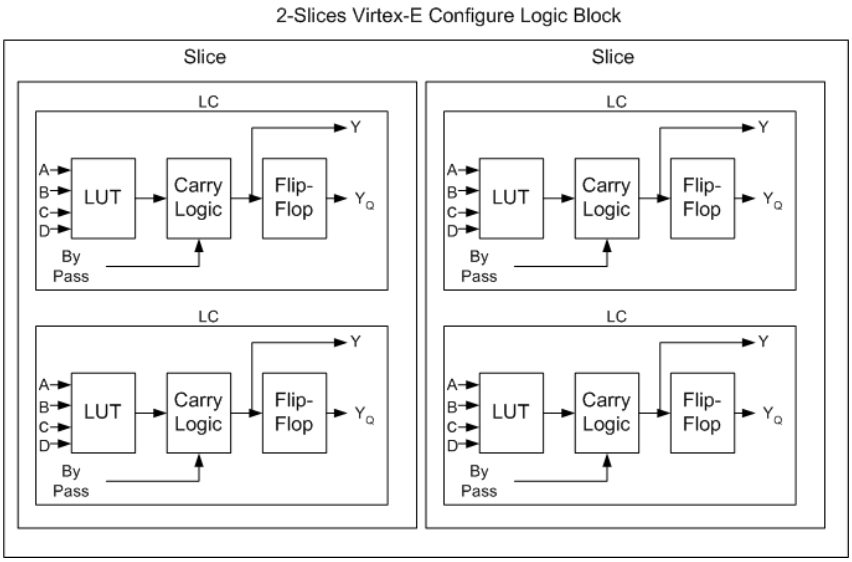


Figure I.9. 2-Slices VirtexE Configuration Logic Block (CLB).

Each Virtex-E CLB contains four LCs, organized in two similar slices. Virtex-E function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM. Figure I.9 shows the structure for a single VirtexE CLB.

Figure I.9 can help in understanding design statistics of an FPGA architecture. Each VirtexE CLB contains 2 slices and each slice contains 2 LUTs, 2 Flip-Flops, and some carry logic. As an example, Virtex XCV400e-8-bg560 contains 4800 slices, 9600 LUTs, and 9600 Flip-Flops in addition to 404 IOBs. It should be noted that slices are often termed as, CLB slices and Flip-Flops as, slice Flip-Flops. Sometimes, the specifications of an FPGA architecture are directly shown in terms of LUTs. To calculate number of slices from LUTs, is complicated as design tools can map a design occupying LUTs from different slices. Ideally, if all the LUTs of all the slices were occupied by a design, the number of LUTs would be exactly double of the number of slices.

I.7.2 Metrics to measure performance

Both area and throughput factors provide a measure for comparing different designs. Their definitions are supplied to understand implementation results presented in the next sections.

1. **Area:** The space occupied by the design is expressed in terms of CLB slices. Some FPGAs contains other resources like BRAMs, multipliers, etc. If these

resources are used in the design, it is important to mention those resources because those are dedicated resources and do not occupy CLB slices. In case they are not mentioned, it is difficult to justify the savings on space occupied by a given design. The numbers of inputs/outputs are also mentioned in some cases. The design is said to be economical if less space is occupied by it.

2. **Throughput:** Throughput is an important factor to measure timing performances of the design. 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:

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

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

3. **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 high in case of high throughput and less space.

I.8 Summary of contributions

This dissertation investigates the hardware (on reconfigurable logic devices) implementations of three important cryptographic algorithms: DES and AES (from secret key cryptography) and ECC (public key cryptography). Those are highly optimized implementations with respect to both area and time. However this thesis is not limited to the implementation work only but it contributes in the theoretical side too by introducing modifications in the standard transformations of the said algorithms. The thesis work therefore sums up optimizations for cryptographic algorithms due to both theoretical and practical considerations. As a result high speed and economical implementations were obtained. In the rest of this Section, the contributions of this investigation are explained with respect to chosen cryptographic algorithms.

- **DES Algorithm**

  DES served a case of study for implementing block ciphers on reconfigurable platform. The contributions of this work are as follows:

  1. FPGA implementation of DES was made on Xilinx VirtexE device XCV400e-8-bg. It is a one round implementation, which exploits high bit-level parallelism. The design occupies 165 (3%) CLB slices, 117 (1%) Slice Flip-Flops, and 129 (2%) IOBs. The system runs at a frequency of 68.5 MHz and achieves a throughput of 274 Mbits/s which is 10 times faster than the fastest one round FPGA implementation of DES reported in the literature [103]. It is also an economical design occupying less than half of the FPGA resources as compared to [103].

2. The basic primitives in block ciphers specifically for DES were deeply studied. Some possible options were provided for the efficient implementation of those primitives on reconfigurable devices. The design procedure for an FPGA architecture and some useful design tips were given for the implementation of other block ciphers. The discussion made in DES chapter would provide a general guideline for the implementation of other block ciphers on reconfigurable platform.

- **AES**

  Some efficient AES encryptor and encryptor/decryptor cores on FPGAs form part of this thesis contribution. Those FPGA implementations were optimized with respect to both area and time by introducing some modifications in the standard transformation of AES and by using better design techniques for FPGAs. A short description of our contributions about AES are presented below:

  1. **AES Encryptor Core-Sequential Approach:** It is one round single chip FPGA implementation of AES on VirtexE XCV812 device. It takes 10 clock cycles for 10 AES rounds to complete one encryption (block length = 128 bits). Also there is a 0 round which is simply an addition of the plain text and the user-key. The design occupies 2744 (28%) CLB slices and 385 (95%) IOBs. The system runs at 20.192 MHz and data is processed at the rate of 258.5 Mbits/s. The core implements an iterative looping approach by using the standard resources of an FPGA.
  2. **AES Encryptor Core-Pipeline Approach:** It is a pipeline architecture implementing all AES rounds on a single chip FPGA. The design targets VirtexE XCV812 device occupying 2136 CLB slices, 100 Block RAMs, and 385 (95%) IOBs. There is no initial delay as round keys are generated in parallel making possible the encryption process from the first clock cycle. The main characteristics of this design is a look up table method for AES S-Box where the pre-computed values of the S-Box were stored in the built-in memory modules of FPGA. The encryptor core structure occupies 2136 CLB slices (22%) and achieves a throughput of 5.2 Gbits/s at the rate of 40.575 MHz. Both AES encryptor cores using iterative looping (from the last step) and pipeline approach show a trade-off between hardware area and time. The former occupies less hardware resources consuming more time for one encryption. The later is a fast architecture but the cost has been paid in terms of hardware resources. Both AES encryptor cores produced promising results and were published in [83, 82].
  3. **AES Decryptor Core:** Encryption and decryption processes in AES are not symmetrical. A separate decryptor core is therefore implemented on FPGA to investigate the time differences for encryption and decryption.

It is a single chip FPGA implementation using a pipeline architecture. There is an initial delay of 11 clock cycles, as the round keys for decryption are required to be stored in the reverse order before the decryption starts. The design is implemented on VirtexE XCV812 device occupying 3216 CLB slices(34%), 100 BRAMs (35%), and 385 IOBs (95%). The decryptor core achieves a throughput of 4.95 Gbits/s at the rate of 38.67 MHz. The achieved throughput for the decryptor core is still less than the encryptor core (5.2 Gbits/s) from the previous step and it consumes more FPGA resources. In fact, it has been tried to minimize the decryption time by introducing some modifications in the standard transformations of the algorithm. The goal was achieved, as both the encryption and decryption timings are now very close. This work was published in [80].

4. 1^{st} **AES Encryptor/Decryptor Core:** This is a single chip FPGA implementation of an encryptor/decryptor core using VirtexE XCV2600 device. The architecture explores various options for optimizing AES S-Box for its implementation on FPGAs. AES S-Box can be implemented using look-up table method where the pre-computed values are stored in the memories. Some FPGA families like Virtex series devices contain more than 280 memory modules each one of 4K bits. Those are fast memories and can be used to speed up S-Box (substitution) operation. However most of FPGA families do not posses such a large number of memory modules. An alternative could be on-fly architecture scheme where the values for the S-Box are manipulated in each clock cycle. Both approaches for AES S-Box were implemented. Also it has been observed that there exist some common operations for an encryptor/decryptor core. Those operations were implemented once and were shared for both encryption and decryption. The encryptor/decryptor core using look up table method is fast and achieves a throughput factor of 3840 Mbits/s by consuming 5677 (22.3%) CLB slices and 80 (43%) Block RAMs. The encryptor/decryptor core using on-fly architecture scheme is relatively slow as it achieves throughput of 3136 Mbits/s and occupies 12270 (48%) CLB slices but no Block RAMs. The achieved results for both architectures improves similar FPGA implementations of AES and were published in [84].

5. 2^{nd} **AES Encrytpor/Decryptor Core:** This is an encryptor/decryptor core which uses the look-up table method for the AES S-Box and then focus on optimizing an other important operation known as MixColumn/Inverse MixColumn (see Chapter 3). Normally both the MixColumn for encryption and Inverse MixColumn for decryption are implemented separately consuming more FPGA resources and producing long critical paths. We introduced a modification for this architecture, which uses the MixColumn step for both encryption/decryption and only a small transformation is applied after MixColumn for decryption. That consumed less

FPGA resources and helped in reducing the critical paths for the circuit. 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. It was published in [75].

- **ECC:** Reconfigurable implementations of ECC conclude the third part of this thesis. It has been explained earlier in Section I.5 that most important operation in ECC is elliptic curve scalar multiplication or point multiplication, which was implemented using a three-layer model: finite field arithmetic in GF(2^m), elliptic curve arithmetic in GF(2^m), and point multiplication. The efficiency of point multiplication therefore directly depends on the efficiency of the bottom two layers. In this thesis, all three layers have been addressed.

  1. **Finite Field Arithmetic GF**(2^m). Finite field arithmetic in GF(2^m) in this thesis includes multiplication, squaring, inversion, and then reduction in GF(2^m). Multiplication in GF(2^m) is a costly operation which has been implemented on FPGAs using binary Karatsuba-Ofman approach (a variant of Karatsuba Multiplier). Some economical and efficient methods were devised for the FPGA implementation of squaring in GF(2^m) and for reduction. Inversion mainly includes a sequence of multiplication operations, which was performed by using same building block for the multiplier using binary Karatsuba-Ofman approach. The results of those high speed implementations plus the estimations for the second and third layers were published in [76].
  2. **Elliptic Curve Scalar Multiplication-Hessian form:** As it was earlier explained in Section I.5, elliptic curve scalar multiplications is the most dominant operation in elliptic curve cryptography. For the second and third layers, several parallel approaches were considered. An efficient architecture for point multiplication using Hessian form of elliptic curve was implemented. The group law (point addition & doubling) for the Hessian form of elliptic curve allows maximum parallelism subject to the availability of hardware resources. Two Karatsuba multipliers in GF(2^{191}) were used for the implementation of Hessian form of elliptic curve on FPGAs. The architecture was implemented on VirtexE XCV2600 which consumed 18314 (56%) CLB slices and 24 (11%) BRAMs and computes scalar multiplication in $114.71\mu S$. Achieved results improve time factor for point multiplication over the existing ECC FPGA implementations and were published in [87]. An improved version of this work has been already accepted as a book chapter published by Nova Science New York [86].
  3. **Elliptic Curve Scalar Multiplication-Montgomery Algorithm:** Implementation for the point multiplication is based on Montgomery

algorithm which was further modified in [50]. The algorithm offer less field operations as compared to the Hessian form of elliptic curves. A generic architecture for the implementation of point multiplication was presented which was further used for the implementation of Montgomery point multiplication on FPGAs. The FPGA implementation of this algorithms was made on VirtexE XCV2600 which computes point multiplication (GF(2^{191})) in just $56.44\mu S$. It costs 18314 (56%) CLB slices and 24 (11%) BRAMs. This work was published in [81]. An improved version of this work is already accepted for its publication in [88].

I.9 Dissertation organization

This dissertation starts with some mathematical concepts, which are necessary for the understanding of the basic operations involved in cryptographic algorithms. In the next chapters, it is presented a short introduction to the chosen cryptographic algorithms and our investigations about them. The result comparisons and conclusion remarks are presented at the end of each Chapter. Concluding remarks for the whole dissertation work were presented in the last Chapter. A short summary for each chapter is presented below.

In Chapter 1, some important mathematical concepts are presented. Those concepts are particularly helpful for the understanding of cryptographic operations for AES and ECC. Mathematical steps for some selected applications of ECC are also described at the end of this Chapter.

In Chapter 2, 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.

In Chapter 3, we explore multiple architectures for AES. Some novel techniques for the implementation of AES are described. Several efficient AES encryptor and encryptor/decryptor cores based on those novel techniques are presented on reconfigurable platforms. The benefits/drawbacks of all AES cores are examined. Achieved results are compared with previous state-of-the-art implementations of AES on FPGAs.

In Chapter 4, several issues for implementing Elliptic Curve Cryptography (ECC) are discussed. Theoretical description and FPGA implementation of finite field arithmetic in GF(2^m) are provided. Some algorithms for elliptic curve arithmetic (point addition & point doubling) are described and discussions are made for their fast implementations on FPGAs. Parallel strategies for elliptic curve scalar multiplication and some efficient parallel architectures for the implementation of elliptic curve scalar multiplication are described.

I.9. DISSERTATION ORGANIZATION

In Chapter 5, Conclusion remarks for the whole dissertation are drawn.

Chapter II

Mathematical Background

This chapter presents a brief introduction to the theory of finite fields necessary to be familiar with the basic operations involved in the specifications of Rijndael algorithm (new Advance Encryption Standard (AES)). The reader is then introduced to some of the most important mathematical concepts, fundamental for the understanding of elliptic curve public-key cryptosystems. For a more detailed treatment of these aspects, the reader is referred to Number theory books like [100, 53, 22, 78], and to elliptic Curve mathematical books like [57, 47, 56]. The material presented in this chapter was written based on [24, 20, 77].

II.1 Finite fields

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

II.1.1 Rings

A ring R is a set whose objects can be added and multiplied, satisfying the following conditions:

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

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 R, that is, if there is a unique $u \in R$ such that: $xu = ux = 1$. 1 is called the *unit element* of the ring.

II.1.2 Fields

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

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

II.1.3 Finite fields

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

In cryptography two cases are mostly used: $q = p$, with p a prime and $q = 2^m$. The former case, GF(p), is denoted as *prime field*, whereas the latter, GF(2^m), is known as finite field of characteristic two or simply *binary extension field*. Binary extension field is also denoted as F_{2^m}. Elements of F_{2^m} are m-bit strings. The rules for arithmetic in F_{2^m} can be determined by either polynomial representation or by optimal normal basis representation. Since F_{2^m} operates on bit strings, computers can perform arithmetic in this field very efficiently. The arithmetic performed in our work is based on binary extension fields using polynomials basis. A short description of polynomial basis and their arithmetic operations is provided in the next subsection.

II.1.4 Polynomials over a field

A polynomial over a field F is an expression of the form

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_2x^2 + b_1x + b_0 \qquad \text{(II.1)}$$

where,

- x is called *indeterminate* of the polynomial, n is a non-negative integer, and $b_{n-1}, ..., b_0$ ($b_i \in$ F) are fixed scalars, called the coefficients of the polynomial b.
- The highest occurring power of x ($n-1$, if the coefficient a_{n-1} is not zero) is called the *degree* of b.

II.1.5 Operations on polynomials

We define two operations addition and multiplication here.

Addition. Let $c(x)$ be the sum of two polynomials $a(x)$ and $b(x)$ then addition of polynomials consists of adding the coefficients with equal power of x, where the addition of the coefficients occur in the underlying field F:

$$c(x) = a(x) + b(x) \Leftrightarrow c_i = a_i + b_i, 0 \leq i < n \tag{II.2}$$

- In addition, neutral element 0 is the polynomial with all coefficients equal to 0.
- The inverse element is found by replacing each coefficient with its inverse in F.
- The degree of $c(x)$ is at most the maximum degree of $a(x)$ and $b(x)$.
- The addition and subtraction are the same if F=GF(2).

Example:
Let F be the field GF(2), the sum of polynomials denoted by 57 and 83 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.

Multiplication. If $m(x)$ is the reduction polynomial, then multiplication of two polynomials $a(x)$ and $b(x)$ is the algebric product of the polynomials modulo the polynomial $m(x)$:

$$c(x) = a(x).b(x) \Leftrightarrow c(x) = a(x) \times b(x) \qquad (\texttt{mod } m(x)) \tag{II.3}$$

Multiplication of polynomials is associative, commutative and distributive with respect to addition.

II.1.6 Polynomials and bytes

A byte can be considered as a polynomial with coefficients in GF(2):

$$\begin{aligned}&b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b(x) \\ &b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0\end{aligned}$$

The set of all byte values corresponds to the set of all polynomials with degree less than eight.

Addition: Addition of bytes can be defined as addition of corresponding polynomials.

Multiplication: In order to define byte multiplication, we need to select a reduction polynomial $m(x)$. Let us use the following irreducible polynomial as reduction polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{II.4}$$

Since this reduction polynomial is irreducible, we have constructed a representation for the field GF(2^8). Hence we can say that the bytes are considered as elements of GF(2^8). Operations on bytes are defined as operations in GF(2^8).

Example:
In our representation for GF(2^8), the product of the elements 57 (hex) and 83 (hex) is the element C1 (hex), since:

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

and

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

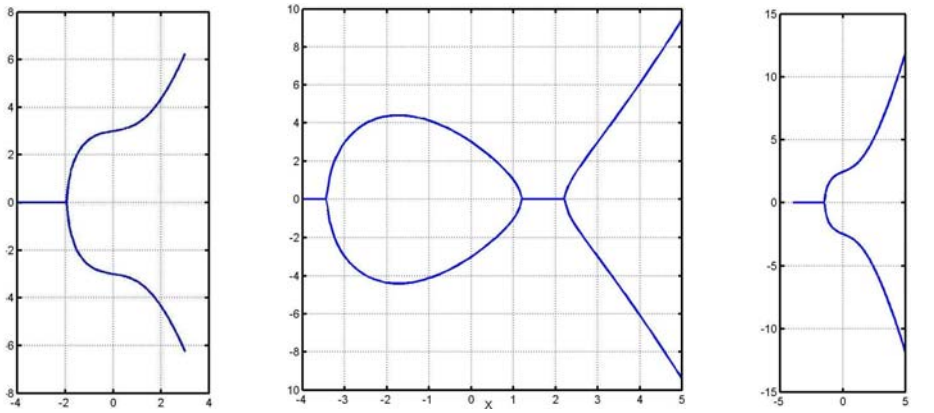
II.2 Elliptic curves

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 first time proposed by N. Koblitz [48] and V. Miller [60]. 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. To explain the geometric properties for elliptic curves they are examined first over real numbers. However, 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 our work.

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



$y^2 = x^3 + x + 9$ $y^2 = x^3 - 9x + 9$ $y^2 = x^3 + 2x + 6$

Figure II.1. Elliptic curve equation $y^2 = x^3 + ax + b$ for different a and b

II.2.1 Definition

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

$$y^2 = x^3 + ax + b \tag{II.5}$$

where a and b are real numbers. Each choice of a and b produces a different elliptic curve as shown in Figure II.1. The elliptic curve in Equation II.5 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 O called the point at infinity.

II.2.2 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 presented here. It is to be recalled that the negative of a point $P = (x, y)$ is 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 II.2.

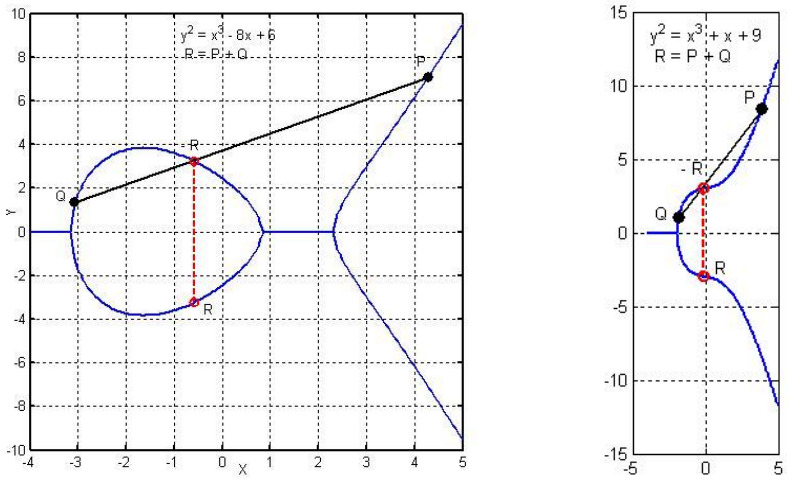


Figure II.2. Adding two distinct points on an Elliptic curve ($Q \neq -P$).

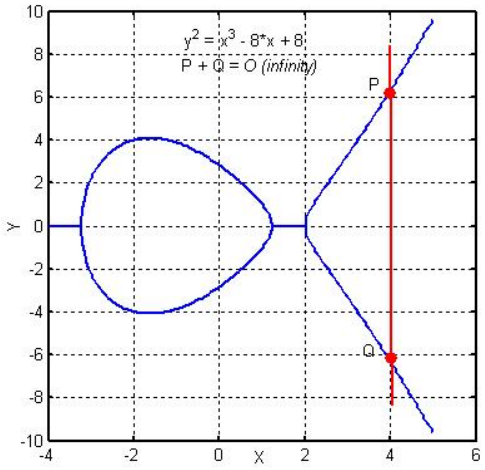


Figure II.3. Adding two points P and Q when $Q = -P$

- **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 II.3. This is the reason why the elliptic curve group includes the point at infinity O. By definition, $P + (-P) = O$. As a result of this equation, $P + O = P$ in the elliptic curve group. The point at infinity O is called the additive identity of the elliptic curve group. All elliptic curves have an additive identity.

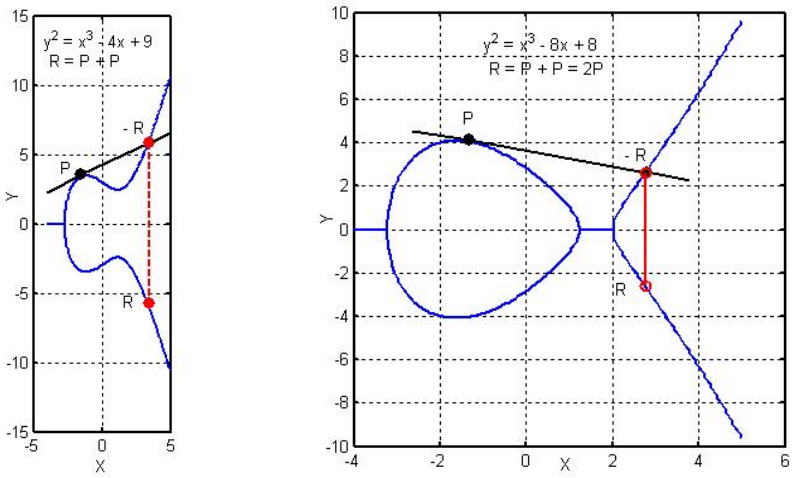


Figure II.4. Doubling a point P on an Elliptic curve.

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

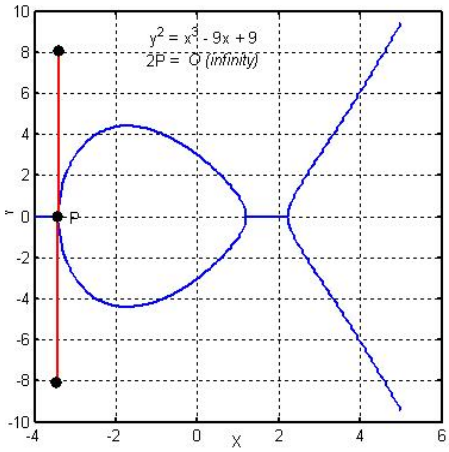


Figure II.5. Doubling $P(x, y)$ when $y = 0$

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 II.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, $2P = O$ for such a point P. If one wanted to find $3P$ in this situation, one can add $2P + P$. This becomes $P + O = P$. Thus $3P = P$, $4P = O$, $5P = P$, $6P = O$, $7P = P$, etc.

II.2.3 Elliptic curve scalar multiplication

There is no multiplication operation in elliptic curve groups. However, the scalar product nP can be obtained by adding n copies of the same point P by using the addition and doubling formulae as were explained in the last section. Thus the product $nP = P + P + ...P$ obtained in this way is referred to elliptic curve scalar multiplication. Figure II.6 shows the scalar multiplication process for obtaining 6 copies of the point P. However for the security of elliptic curve cryptosystems, higher and prime values of k are used. The values range from 160-521 bits provides enough security for elliptic curve cryptosystems against known attacks.

II.3 Elliptic curves over F_{2^m}

The equation for the elliptic curve with the underlying field F_{2^m} is slightly adjusted for binary representation because of its characteristic 2 as shown in Equation II.6. It is formed by choosing the elements a and b within F_{2^m} with $b \neq 0$.

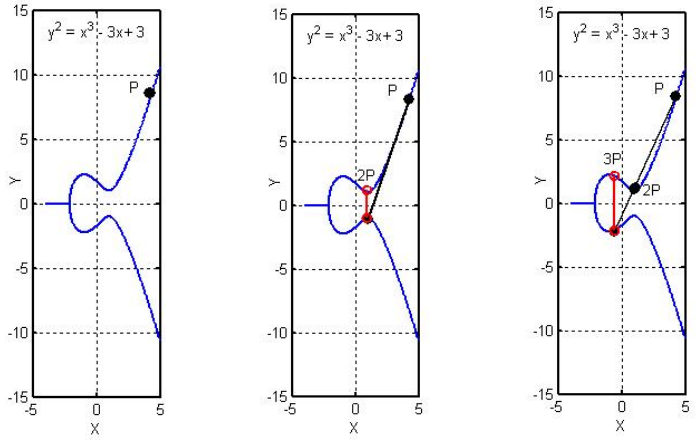
$$y^2 + xy = x^3 + ax^2 + b \tag{II.6}$$

The elliptic curve includes all points (x, y) which satisfy the elliptic curve equation over F_{2^m} (where x and $y \in F_{2^m}$). An elliptic curve group over F_{2^m} consists of the points on the corresponding elliptic curve, together with a point at infinity, O.

The points on an Elliptic curve can be represented using either two or three coordinates. In affine-coordinate representation, a finite point on $E(F_q)$ is specified by two coordinates x; $y \in F_q$ satisfying Equation II.6. The point at infinity has no affine coordinates. We can make use of the concept of a projective plane over the field F_q [59]. In this way, one can represent a number 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,

$$P(x; y) = P(X; Y; Z)$$

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



(a) P (b) $2P$ (c) $3P$

(d) $4P$ (e) $5P$ (f) $6P$

Figure II.6. Elliptic curve scalar multiplication kP, for $k = 6$ and for the elliptic curve $y^2 = x^3 - 3x + 3$

Affine-to-Projective:	$X = x;$	$Y = y;$	Z=1
Projective-to-Affine:	$x = X/Z^2;$	$y = Y/Z^3$	

The algebraic formulae for the group law in an underlying field are different for affine and projective coordinates. In the next subsections the group law over F_{2^m} is explained using affine coordinates representation. The group laws for the different projective coordinates representations can be seen at [40].

II.3.1 Point addition

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

$$\begin{aligned} m &= \tfrac{(y_2+y_1)}{(x_2+x_1)} \\ x_3 &= m^2 + m + x_1 + x_2 + a \\ y_3 &= m(x_1 + x_3) + x_3 + y_1 \end{aligned} \tag{II.7}$$

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

II.3.2 Point doubling

Let $P(x_1, y_1)$ is the point on the curve. If $x_1 = 0$, then $2P = O$. If $x_1 \neq 0$ then $R = 2P$, and $R(x_2, y_2)$ is:

$$\begin{aligned} m &= x_1 + \tfrac{y_1}{x_1} \\ x_2 &= m^2 + m + a \\ y_2 &= (x_1)^2 + (m + 1) \times x_2 \end{aligned} \tag{II.8}$$

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.

II.3.3 Order of an elliptic curve

Recall that all the points in F_q that satisfy the Equation II.6 plus the point at infinity O forms the elliptic curve group and is denoted as $E(\mathrm{F}_q)$. Each group consists of finite number of elements. 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 $E(\mathrm{F}_q)$ 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 O, is called the order of the curve. The order is written $\#E(\mathrm{F}_q)$. A renowned result discovered by Hasse gives the lower and the upper bounds for this number.

Theorem[57]: Let $\#E(\mathrm{F}_q)$ be the number of points in $E(\mathrm{F}_q)$. Then,

$$q + 1 - 2\sqrt{q} \leq \#E(F_q) \leq q + 1 + 2\sqrt{q} \tag{II.9}$$

The interval $[q + 1 - 2\sqrt{q}, q + 1 + 2\sqrt{q}]$ is called the *Hasse interval*. An alternate form of this theorem is as: if E is defined over F_q, then $\#E(\mathrm{F}_q) \leq q + 1 - t$ where $t \leq 2\sqrt{q}$; t is the called the *trace* of E over F_q. Since $2\sqrt{q}$ is small relative to q, we have $\#E(\mathrm{F}_q) \approx q$.

The order of an element in elliptic curves can also be described as follows: The order of a point P on $E(\mathrm{F}_q)$ is the smallest integer k such that $k\mathrm{P} = O$. The order of any point always exists and divides the order of the curve $\#E(\mathrm{F}_q)$. This guarantees that if r and l are integers, then $rP = lP$ if and only if $r \equiv l \pmod{k}$.

II.3.4 Elliptic curve groups and the discrete logarithm problem

Every cryptosystem is based on 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 get sum of any number of copies of a point ($2P$, $3P$, kP, etc). The determination of a point nP in this manner is referred to as *Scalar Multiplication* of a point. The ECDLP is based upon the intractability of scalar multiplication products.

II.3.5 An Example

Consider the field F_{2^4} with the irreducible polynomial $p(x)$, defined as:

$$p(x) = x^4 + x + 1 \tag{II.10}$$

Then if g is the root of $p(x)$, we have $p(x)= 0$ that implies:

$$p(g) = g^4 + g + 1 = 0 \tag{II.11}$$

Addition and subtraction in binary field arithmetic are the same, hence the Equation II.11 can be written as:

$$g^4 = g + 1 \tag{II.12}$$

The powers of g give all the elements in the field F_{2^4}. We can define the 15 non-zero elements for F_{2^4} by using Equations II.10 and II.12 as shown in Table II.1. Notice that any element in F_{2^4} can be defined using only four coordinates. Also it is to be noted that each element in F_{2^4} can be described in any of the three representation as given in Table II.1: coordinate representation, polynomial representation and powers of the primitive root g.

Let us now consider the curve in Equation II.6 defined as the set of points $(x, y) \in F \times F$ that satisfy:

$$y^2 + xy = x^3 + g^4x^2 + 1 \tag{II.13}$$

We have selected the values of $a = g^4$ and $b = g^0 = 1$ and substituted in Equation II.6 for the above equation. Using Table II.1, the point (g^5, g^3) satisfies this equation over F_{2^m}, since:

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

Table II.1. Elements of the field F_{2^4} using irreducible polynomial $p(x) = x^4+x+1$

$$\begin{aligned}(g^3)^2+(g^3)(g^5) &= (g^5)^3+g^4(g^3)^2+1\\ g^6+g^8 &= g^{15}+g^{10}+1\\ (1100)+(0101) &= (0001)+(1001)+(0001)\\ (1001) &= (1001)\end{aligned}$$

There exists a total of fifteen points that satisfy this equation. The graphical representation of those points is shown in Figure II.7. The points are:

$(1, g^{13})$, (g^3, g^{13}), (g^5, g^{11}), (g^6, g^{14}), (g^9, g^{13}), (g^{10}, g^8)
(g^{12}, g^{12}), $(1, g^6)$, (g^3, g^8), (g^5, g^3), (g^6, g^8), (g^9, g^{10}),
(g^{10}, g), $(g^{12}, 0)$, $(0, 1)$

Let us now add any two points on the curve: say P = $(1, g^{13})$ and Q = $(g^{12}, 0)$. Then by using the point addition formulae from Equation II.7 and the Table II.1,

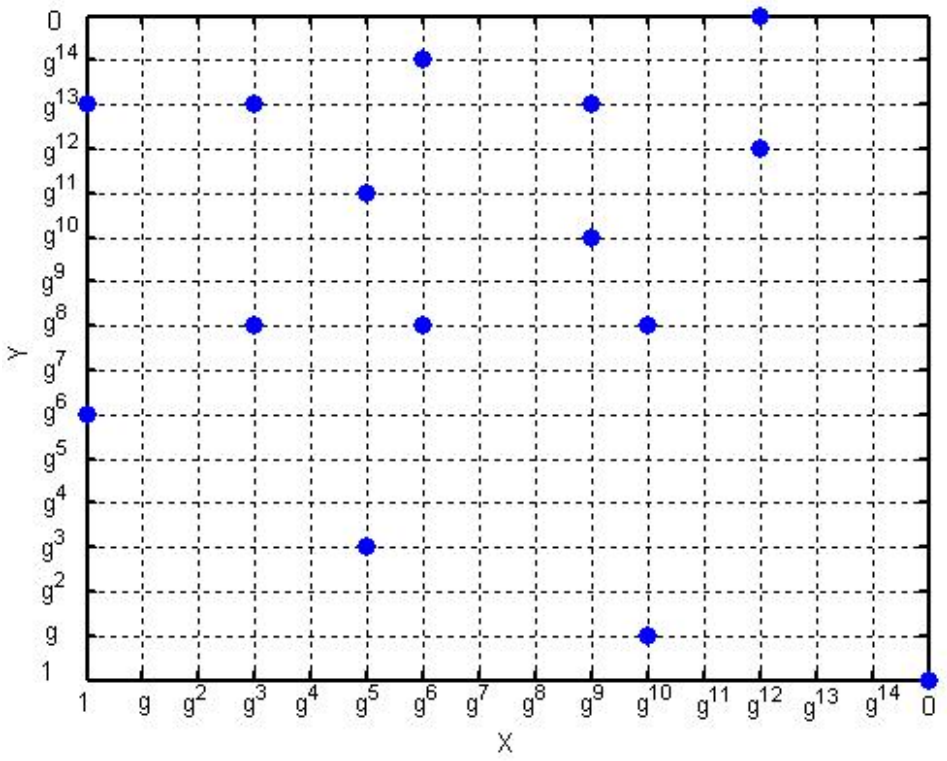


Figure II.7. Points for the elliptic curve $y^2 + xy = x^3 + g^4x^2 + 1$ over GF(2^4)

we have:

$$\begin{aligned}
m &= \tfrac{(y_2+y_1)}{(x_2+x_1)} \\
&= \tfrac{(g^{13})}{(1+g^{12})} \\
&= \tfrac{(g^{13})}{(1+g^3+g^2+g+1)} \\
&= \tfrac{(g^{13})}{(g^{11})} = g^{13}.g^{-11} \\
&= g^{13}.g^4 = g^2 \\
x_3 &= m^2 + m + x_1 + x_2 + a \\
&= g^4 + g^2 + 1 + g^{12} + g^4 \\
&= g^4 + g^2 + 1 + g^3 + g^2 + g + 1 + g^4 \\
&= g^3 + g = g^9 \\
y_3 &= g^2(1 + g^9) + g^9 + g^{13} \\
&= g^2 + g^{11} + g^9 + g^{13} \\
&= g^2 + g^3 + g^2 + g + g^3 + g + g^3 + g^2 + 1 \\
&= g^{13}
\end{aligned} \tag{II.14}$$

The new point $R(x_3, y_3) = (g^9, g^{13})$ is one of the fifteen points and hence it lies on the elliptic curve of Equation II.13. Similarly the resultant point obtained by doubling any point on the curve must lie on the same curve.

P	$2P$	$3P$	$4P$	$5P$	$6P$
(g^3,g^8)	(g^5,g^3)	(g^9,g^{10})	$(1,g^{13})$	(g^6,g^8)	(g^{10},g)

Table II.2. Scalar multiplication for the point P of Equation II.13

It has been mentioned earlier that there is no multiplication operation in Elliptic curve groups. Instead the product kP is obtained by adding k copies of the same point P by using the addition and doubling formulas. This was called as elliptic curve scalar multiplication. An example of getting $6P$ for the same curve in Equation II.13 and by using Table II.1 is shown in Table II.2.

II.4 Elliptic curve cryptography

In the previous sections, we described elliptic curves, their curve operations and defined elliptic curve scalar multiplication as well as elliptic curve discrete logarithmic problem. With this mathematical background we can now build a public-key cryptosystem based on the theory of elliptic curves. These cryptosystems are mainly applicable for establishing secret keys for further use in symmetrical key cryptosystems and the creation of digital signature as well as their digital verification. In the remaining part of this chapter, we will briefly discuss some of the most Relevant aspects in the construction and design of elliptic curve cryptosystems.

II.4.1 Elliptic curve cryptosystem parameters

Suppose that we select the elliptic curve $E(\mathrm{F}_q)$ as defined in the Equation II.6 and that the underlying finite field F_q, its coefficients a, b and the order $\#E(\mathrm{F}_q)$ of the selected curve are given. Let us select the base point $P \in E(\mathrm{F}_q)$ with a prime order k then the private/public key pair can be defined by computing elliptic curve discrete logarithm $W = sP$ where,

- The private key s is integer modulo k.
- The corresponding public key W is a point on the curve $E(\mathrm{F}_q)$.

The calculations for the public key W are directly based on the elliptic curve discrete logarithm problem. It is therefore said that the security of this cryptosystem relies in the difficulty of discrete logarithm problem.

II.4.2 Key pair generation

Let P is the point on the elliptic curve $E(\mathrm{F}_q)$ of prime order k then for the computation of a private/public key pair we choose first a random integer $d \in [1, k-1]$ which is a private key. After that, a public key Q is computed as:

$$Q = (x_Q, y_Q) = dP \tag{II.15}$$

II.4.3 Key exchange

In secret key cryptography, it is necessary that both parties at the sending and receiving ends agree on a secret key for the transfer of messages. Fortunately key agreement protocols exist which help in establishing a key by exchanging information between two parties. It turns out that key protocols are best done using asymmetric (public key) cryptography. We present a famous protocol due to Diffie-Hellman, which provides the key establishment of a key with two message transfers. Then it is explained how the same process can be executed using elliptic curve discrete logarithmic algorithms (ECDSA).

Diffie-Hellman key exchange: Diffie-Hellman key exchange was invented in 1976 during collaboration between Whitfield Diffie, Martin Hellman and Ralph Merkle and was the first practical method for establishing a shared secret over an unprotected communications channel. Let A and B are already agree on a group G (most commonly used group G is the group of integers modulo p) and an element g in G (g mod p). Then the protocol is as follows (Figure II.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=$(g^a)^b$.

g and p are the public keys and K is the private key for the session which from this moment will be used for the communication between A and B. The protocol is considered secure if G and g are chosen properly: the eavesdropper has difficulty to compute the element g^{ab}, because it is needed to solve the Diffie-Hellman problem related to discrete logarithms in order to deduce a from g^a or b from g^b.

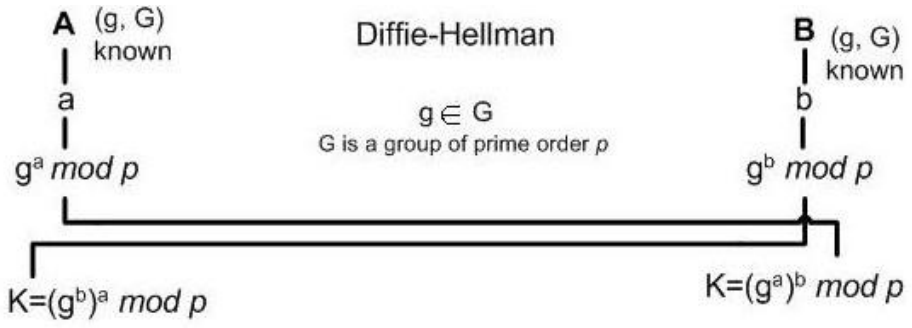


Figure II.8. Diffie-Hellman protocol for key exchange

Elliptic Curve Diffie-Hellman Key Exchange: Let A and B agree on an elliptic curve E over a large finite field F and a point P on that curve. Then following are the steps to exchange a secret key by using elliptic curve discrete logarithmic algorithm as shown in Figure II.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: A by multiplying the received value of r_2P by his/her secret number r_1, and B vice-versa.

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

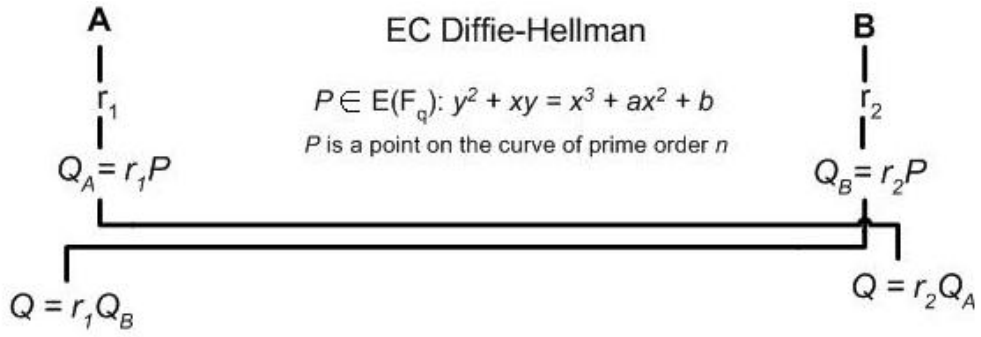


Figure II.9. Variant of Diffie-Hellamn for Elliptic Curves.

II.4.4 Digital signature scheme

Digital signatures confer three qualities on an electronic document. These are data integrity, authentication and non-repudiation. Successful verification of a digital signature ensures the recipient that the 'document received' is identical to the 'document sent' (data integrity) and confirms the identity of the sender (authentication). It also prevents any subsequent denial by the sender that the document originated with them (non-repudiation).

In cryptography, digital signature schemes relay on public key encryption where data is signed by encrypting with private key of the sender and verified by decrypting it with the public key. A general digital signature scheme is shown in Figure II.10.

As shown in Figure II.10 a hash function (a hash function h maps bitstrings of arbitrary finite length to strings of fixed length, say n bits. MD5 and SHA-1 are two examples of hash functions) is applied to the message to add an extra information to the document which is then signed by encrypting it with the private key of the sender: the signature part of the message. The signature is annexed to the original message and sent to the destination. On the receiving end, both parts: the message

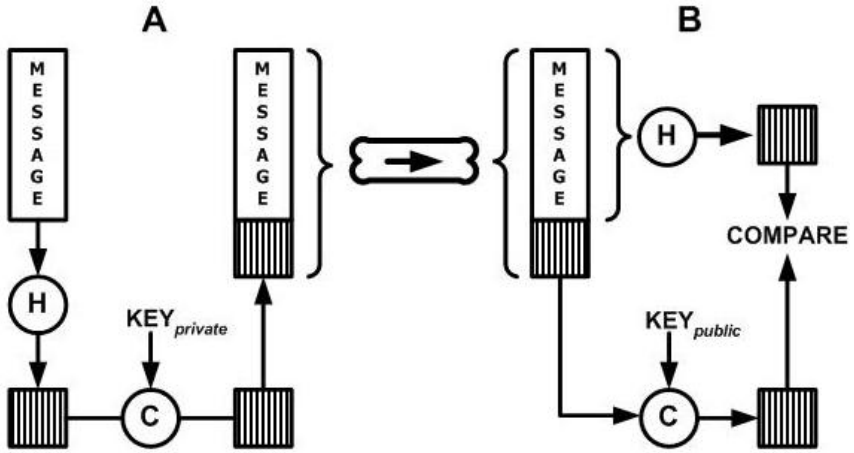


Figure II.10. A general digital signature scheme

and the signature are separated. The same hash function is applied to the received message to get λ_1 and the signature is verified by performing the decryption using the public key of the signer obtaining λ_2. If $\lambda_1 = \lambda_2$, then is said that message has not changed during transmission and the origin of the message is also confirmed.

Digital signature with ECDSA: Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the Digital Signature Algorithm (DSA). It was accepted in 1999 as an ANSI standard, and was accepted in 2000 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.

The following is the procedure to digitally sign a message with ECDSA. Only a private key holder can perform digital signature.

1. Let M be the message then a compressed version of M is obtained using Hash function, $e = H(M)$.
2. A random integer $n \in [1, k-1]$ is selected which is secret and valid for the specific message only.
3. A point on the curve is obtained using n, $(x_1, y_1) = nP$.
4. Using the field element x_1 from the last step, generate,

$$r = x_1 \pmod{k} \tag{II.16}$$

and

$$s = n^{-1}(e + dr) \pmod{k} \tag{II.17}$$

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 no one can substitute a different message for the same signature.

Verification: On the receiving end, the recipient can verify the signature using the received signature' s values and the public key Q of the singer. Let us call the received pair (r', s'). The signature has been verified if the received pair (r', s') is equal to the pair (r, s). The following is the procedure for the signature' s verification.

1. Verify that r' and s' are between $[1, k-1]$, if not, the signature is rejected.
2. Using hash function, compute $e' = H(M')$.
3. Compute

$$\begin{aligned} c &= (s')^{-1} \quad (mod\ k) \\ u &= (e')c \quad (mod\ k) \\ v &= (r')c \quad (mod\ k) \end{aligned} \tag{II.18}$$

4. Compute the point $(x, y) = uP + vQ$. If this point is the point at infinity, the signature is rejected.
5. Compute $w = x \pmod k$.
6. If $r' = w$, the signature is valid and the process of verification ends.

II.5 Symmetric vs asymmetric cryptography

In the last section, it has been explained the various applications (signature, verification, key exchange) of elliptic curve cryptography which is an asymmetric or public key version of cryptography. The other variants of the same protocols using other asymmetric ciphers like RSA have been used in the industry since long time. Those algorithms are based on complicated mathematical problems, which are infeasible to solve in a considerable time period. On the other hand, symmetric ciphers are based on well understandable mathematical problems, which contribute to the algorithmic strength and can efficiently be implemented on both software and hardware platforms. That especial characteristic of symmetric ciphers makes them suitable for the encryption of large amount of data at high speed. Both the symmetric and asymmetric cryptography therefore have their unique applications in worldwide industries. In fact, for a large number of applications where symmetric cryptography plays a vital role for rapid encryption, asymmetric encryption works at the same time to decide the session keys and to provide other features like authentication. In short, both symmetric and asymmetric cryptosystems have their specific applications and also they are used together for the safe communications on insecure networks. This thesis therefore addresses rapid implementations of cryptographic algorithms from both symmetric and asymmetric cryptography.

Chapter III

General Guidelines for Implementing Block Ciphers in FPGAs

This chapter provides general guidelines for the implementation of block ciphers in reconfigurable logic platform. The general structure and the design principles for block ciphers are discussed. Basic primitives in block ciphers are identified and some useful design techniques are devised for efficient implementations on reconfigurable devices. The chapter presents Data Encryption Standard (DES) as a case of study. A reconfigurable implementation of DES has been carried out using the same guidelines in this chapter achieving a throughput of 274 Mbits/s occupying just 165 CLB slices for DES one round implementation.

III.1 Introduction

Block ciphers are based on well-understood mathematical problems using non-linear functions and linear modular algebra [56].

Most of the block ciphers have regular structure: same building block is repeated multiple times. Generally, block ciphers are of symmetric nature: the encryption and decryption only differ in the way of using the key so that the same device can be used for both encryption and decryption.

Implementation of block ciphers mainly use bit-level operations and table look-ups. Bit-wise operators (XORs, AND/OR, etc.), substitutions, logical shifts and permutations are quite common operations. Such operations are well suited for fast execution in reconfigurable hardware platform (FPGAs). Furthermore, currently abundant memory resources in FPGAs enhance encryption speed for the operations

like substitution. Highly parallel architectures can be designed in reconfigurable hardware achieving higher performance compared to software implementations.

In this chapter, we analyze basic characteristics of block ciphers and we explore about general strategies for implementing them in reconfigurable logic platform. We search for the most frequent operations involved in block ciphers and we develop strategies for their implementations in reconfigurable devices. It is also explained how the bit level parallelism is exploited for block ciphers using either iterative or pipeline approaches. We present a case of study for Data Encryption Standard (DES), which can be extended to the other similar cryptosystems.

DES is the most popular example in the field of block ciphers [26, 32], which 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 FPGAs, DES implementations can be found on all platforms: software [26, 32, 46, 19, 18], VLSI [29, 28, 102] and reconfigurable hardware using FPGA devices [49, 103, 45, 13, 55, 102, 71]. 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 III.2 describes the general structure and design principles for block ciphers. Some useful properties for the implementation of block ciphers in reconfigurable platform are also discussed in this Section. An introduction to Data Encryption Standard (DES) is presented in Section III.3. A description of encryption and key scheduling processes using DES is also provided in this Section. In Section III.4, design procedure and the design techniques for an FPGA architecture are described first, followed by DES implementation in FPGAs. Finally, conclusions are made in Section III.5.

III.2 Block ciphers

In cryptography, a block cipher is a type of symmetric key cipher which operates on groups of bits of a fixed length, called blocks. Block sizes are typically 64 or 128 bits, though some block ciphers have a variable block size. DES is a typical example of block ciphers, which operates on 64-bit block. Modern block ciphers operate on block length of 128 bits or more. Rijndael (new Advanced Encryption Standard) allows block lengths of 128, 192, or 256 bits. A block cipher makes use of a key for both encryption and decryption. The key length not necessary to matches the block size of the input data. In 3DES (a variant of DES), a 64-bit block length is allowed using a key of 192-bit (three keys of 64-bit each) for encryption and decryption. Rijndael allows the various combinations of 128, 192, and 256 bits. Block length of less than 80 bits is not recommended for current security applications [65].

In the rest of this Section, general structure and the design principles for the block ciphers are explained. Basic primitives in block ciphers are discussed. Finally, the implementation considerations are made in general for the hardware platforms specifically for reconfigurable type of hardware.

III.2.1 General structure of a block cipher

As shown in Figure III.1, there are three main building blocks for a block cipher: block cipher encryption, block cipher decryption, and key schedule. The input is plaintext for a block cipher encryption and the output is ciphertext. Block cipher decryption takes ciphertext as input and converts back to plaintexts. A number of rounds are performed for a single encryption/decryption of a block. Each round uses a round key which is derived from cipher key through a process called *key scheduling*. Key schedule is occupied for both block cipher encryption and decryption. The general structure of a block cipher involves three main building blocks:

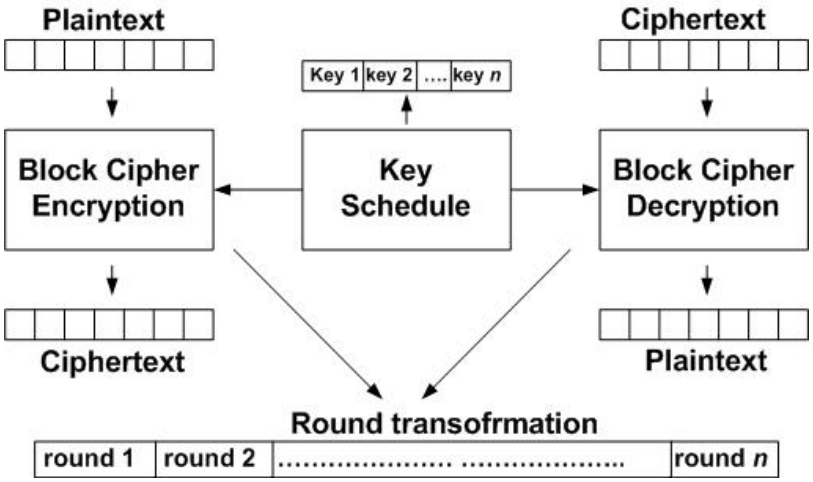


Figure III.1. General structure for a block cipher

1. **Block cipher encryption:** Most of the block ciphers are Fiestel [94] ciphers. Fiestel ciphers divide input into two halves and then two halves of the data pass through n rounds of processing. In the final round, the two halves are combined to produce the ciphertext block. All rounds have the same structure. Each round uses a round key, which is derived from the cipher key. In general all the round keys are different from each other and from the cipher key. Most of the block ciphers partial or completely follow the same Fiestel structure. DES is a perfect Fiestel cipher. Modern block ciphers also repeat n rounds of the algorithm but they do not divide the input block into two halves. All the rounds of the algorithm are generally similar. The round transformation normally includes the basic operations of substitution and permutation, which apply non-linear transformations to strength the algorithm against the current cryptanalytic attacks.

2. **Block cipher decryption:** Fiestel ciphers use the same structure for both encryption and decryption. Modern block ciphers however maintain symmetric nature of the algorithm but use different critical path for decryption, that is,

for each operation there is a reverse operation but the number of operations remains the same.

3. **Key schedule:** The round keys are derived from the user key through a process called *key scheduling*. Generally, block ciphers occupy the same steps for the derivation of round keys for both encryption and decryption or the round keys for encryption can be used for decryption in reverse order.

III.2.2 Design principles for a block cipher

As knowledge of cryptanalysis has been evolved and as the need for both software and hardware encryption has emerged, advances have been shown in the designing new block ciphers to comply with the current security requirements. In this subsection, we analyze some key features in designing of a block cipher.

1. **Key size:** If the block cipher is designed to be highly resistant against brute force attack, then its strength is determined by its key length: the longer the key, the longer its takes against brute force search. Modern block ciphers employ key lengths of 128 bits or more. A key length of less than 80 bits is not sufficient for the current security requirements [65].

2. **Variable key length:** The encryption speed may be reduced by increasing key length. Modern block ciphers offer variable key lengths to tradeoff the security and encryption speed. Blowfish, CAST, Mars, Rijndael provide a variable key length.

3. **Mixed operations:** The complication against cryptanalysis arise with the use of more than one arithmetic and/or Boolean operator in a block cipher. This approach provides non-linearity as an alternative to S-boxes and also to introduce non-linearity in S-Boxes. S-boxes substitute or transform input bits into output bits. Almost all modern block ciphers use mixed operators.

4. **Variable number of rounds:** Increasing the number of rounds increases the cryptanalytic strength, but also increases encryption/decryption time. Modern block ciphers provide variable number of rounds to allow the user to make a tradeoff between security and execution speed. Considering number of rounds for a block cipher, the strength of an algorithm is interlinked with the other design parameters. For example, AES with 10 rounds provides higher security level as compared to DES with 16 rounds.

5. **Variable plaintext/ciphertext block length:** A longer block length yields greater cryptographic strength. Also, a variable block length can provide a measure of convenience, allowing the algorithm to be tailored to the application. RC5 adopts this strategy.

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-wise operations	√	√	√	√	√	√	√	√
Permutation	√		×	×	×	×	√	√
S-Box	√	√	×	√	√	×	√	√
Shift or rotation	√		√	√	√	√	√	√
Fast Key setup	√	×	√	√	√	√	√	√

Table III.1. Key features for some famous block ciphers

6. **Fast key setup:** In Blowfish, a lengthy key schedule algorithm was used. The generation of round keys takes much longer than a single encryption/decryption. As a result, an effort to brute force attacks is greatly magnified. However, fast key setup is strongly required where the keys are required to be changed rapidly. For example, the encryption of Internet protocol (IP) packets in Internet Protocol Security, the overhead due to key setup may become quite relevant. Modern block ciphers offer simple and fast key schedule algorithm. The key schedule algorithm in Rijndael is such an example.

7. **Software/Hardware implementations:** Modern block ciphers are designed to be implemented for both the software and hardware platforms. That implies the use of those arithmetic and logical operations likely to be implemented on both the platforms offering high encryption speed. It is in fact a need for the current security applications.

8. **Simple arithmetic/logical operations:** Perhaps, this characteristic can be seen in most block ciphers. Modern block ciphers also maintain this feature as they include simple bit-wise operations. The algorithms are simple but strong enough against known crypto attacks.

Table III.1 describes key features for some famous block ciphers including five finalists (AES, MARS, RC6, Serpent, Twofish) for the competition of 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, they are fixed for some block ciphers but they can vary depending on the chosen block and key lengths. Most block ciphers can efficiently be implemented in software and hardware platforms. All block ciphers generally include bit-wise (XOR, AND) and shift or rotate operations. Excluding few block ciphers, most of them use S-boxes for substitution. Fast key set-up is an important feature among modern block ciphers. They are not symmetric, that is, same building blocks used for encryption cannot be used for decryption. Normally, each step for encryption has its own inverse for decryption.

III.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 and some properties of symmetric-key cryptographic algorithms well match their implementation on reconfigurable devices like FPGAs.

1. **Bit-wise operations:** Almost all the block ciphers include bit-level operations: XOR, AND, OR, etc. The abundance of bit-level operations in cryptographic algorithms makes their execution faster on FPGAs and additionally they occupy relatively less hardware resources. The basic unit in FPGAs is a logic cell (LC) which provides a 4-input/1-ouput configuration for many families of FPGA devices. This useful feature of FPGAs allow to build 2,3, or 4 input and 1 output Boolean logic using the same hardware resources as shown in Figure III.2.

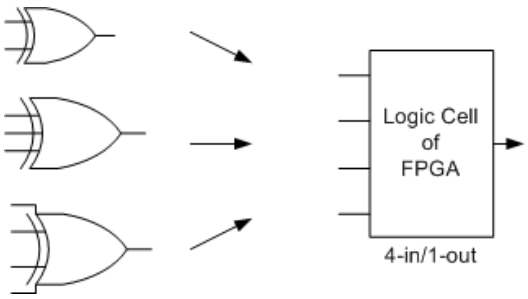


Figure III.2. Same resources for 2,3,4-in/1-out Boolean logic in FPGAs

2. **Substitution:** Substitution is an important operation in symmetric block ciphers to add maximum non-linearity. Substitution is usually built as a look-up table called S-Box. The strength of DES is mainly considered due to its substitution boxes (S-Boxes). AES S-Box is used for both encryption/decryption

and key schedule algorithms. FPGAs offer three solutions for the implementation of substitution operation as shown in Figure III.3. a) The logic cell (LC) of an FPGA can be configured into memory mode. In fact, a function generator (FG) in an LC is implemented as a 4-input Look Up Table (LUT) (see Section I.7.1). A 4-in/1-out LUT provides 4×1 memory. Large number of LUTs can be combined into a big memory. This is a fast approach because the pre-computed values for the S-Box can be stored saving the computational time for S-Box manipulation. b) The Boolean logic for substitution operation can be implemented by configuring LCs in logic mode. It would be slow due to large routing overheads among a large number of LCs. 3) Some FPGA devices contain built-in memory modules, BRAMs (Block RAMs). Those are fast access memories which do not make use of LCs and consequently with minor routing overheads. This approach is similar but fast as compared to the option (a). Only the Virtex series devices contain BRAMs. Virtex series devices contain more than 280 BRAMs of 4K each [8].

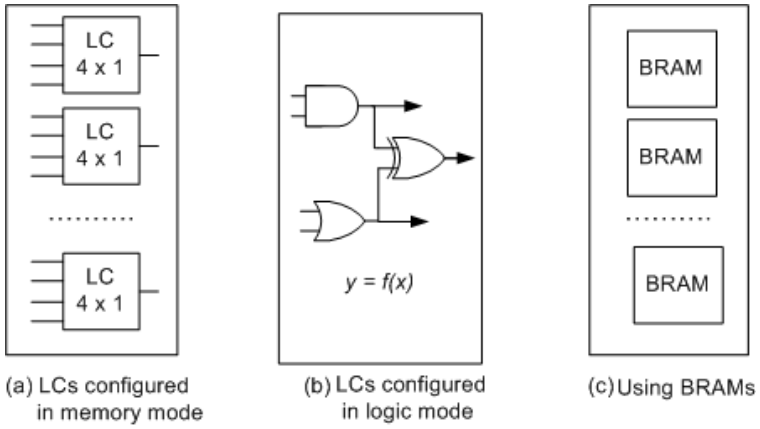


Figure III.3. 3 approaches for the implementation of S-Box in FPGAs.

3. **Permutation operation:** Permutation is a common operation among block ciphers. Fortunately, there is no cost associated with it since it does not make use of FPGA resources and therefore consume no time for its executions. Permutation is just rewiring and the bits are rearranged (concatenated) according to the required order. Figure III.4 demonstrates a simple example of permutation operation for 6 bits only. The same strategy is extended for the permutation operation over longer blocks.

4. **Shift & Rotation operations:** Shift is simpler than the permutation operation. Shift operation is normally executed to achieve some particular data bits or bytes, for example, retrieving 6-bit shorter blocks from 48-bit block to pass through substitution operation in DES. In this case, shorter blocks are made by dividing longer bus to a number of shorter buses as shown in Figure III.5a.

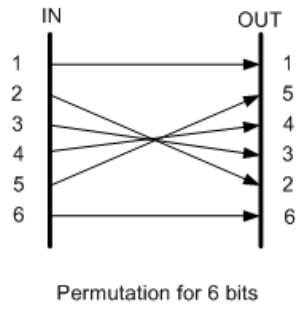


Figure III.4. Permutation operation in FPGAs.

In some cases, the input data is shifted n bits and n zeroes are added, called as zero padding. In FPGAS, zero padding for n bits is achieved by simply connecting n bits to the ground as shown in Figure III.5b. Most block ciphers (AES, RC6, DEAL, etc.) use rotation operation. It is similar to shift operation but no zero padding is used. Instead, bits are moved according to a defined setup. For example, for a 4-bit buffer, shifting left $a_0a_1a_2a_3$ by 1-bit is $a_1a_2a_3$0 while rotating left by 1-bit is $a_1a_2a_3a_0$. 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.

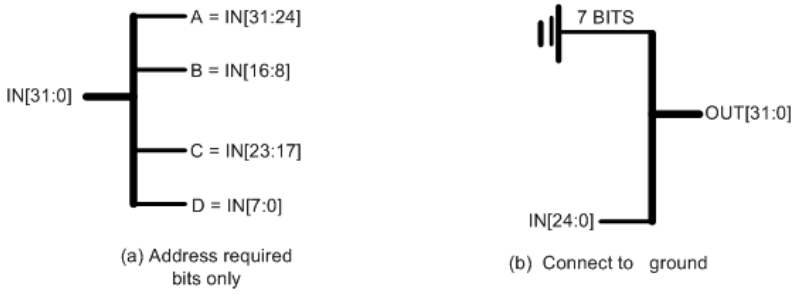


Figure III.5. Shift operation in FPGAs.

5. **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 which implements the algorithm and executes n iterations of it by consuming n clock cycles for a single encryption/decryption as shown in Figure III.6. Obviously, 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 no more critical.

6. **Pipeline design strategy:** If reconfigurable platform is the choice for the

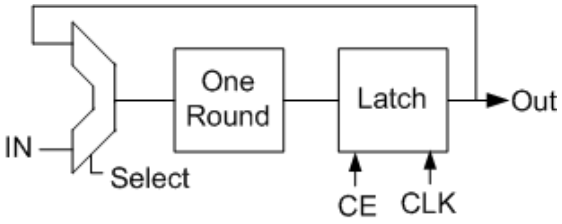


Figure III.6. Iterative design strategy.

implementation of a block cipher, a high speed architecture would result by implementing n rounds of the algorithm as modern FPGAs are bigger enough to accommodate massive circuits. This is called loop unrolling or pipeline approach as shown in Figure III.7. Registers are provided between two consecutive rounds, which operate with the same clock cycle. Once the pipeline is filled, the output blocks therefore appear at each successive clock cycle. This is a fast approach but costly in terms of hardware area.

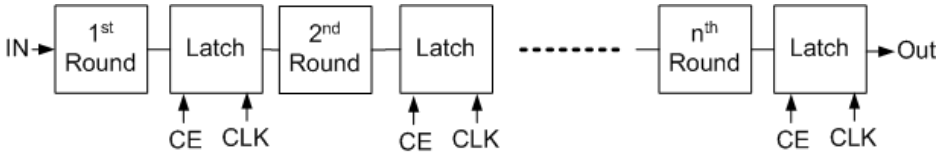


Figure III.7. Pipeline design strategy.

7. **Sub-pipelining design strategy:** 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 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 III.8 represents a sub-pipeline design strategy.

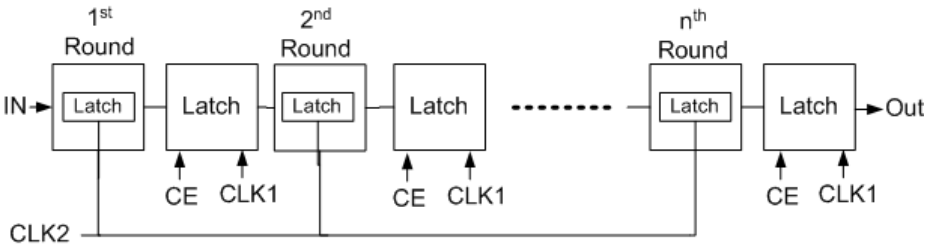


Figure III.8. Sub-pipeline design strategy.

8. **Managing block size:** Modern block ciphers operate on data blocks of 128 bits or more. Unlike software implementations on general-purpose microprocessors, FPGAs permit parallel execution of the whole data block provided there is no data dependency in the algorithm. Our experience shows that the manual search for operations, which can be executed in parallel, is always useful. Moreover, 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.

9. **Key scheduling:** Fast key setup is one of the characteristics of modern block ciphers. The keys are required to be changed rapidly in some cryptographic applications e.g. the encryption of Internet protocol (IP) packets in Internet Protocol Security (IPSec). It is possible to exploit the reconfigurable feature of the FPGAs to configure for various keys rapidly.

10. **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.

III.3 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)[65], to protect data during transmission and storage. NBS launched an evaluation process with the help of National Security Agency (NSA) and finally adopted a modification of LUCIFER algorithm as the new Data Encryption Standard (DES) on July 15, 1977. The Data Encryption Standard [105], known as Data Encryption Algorithm (DEA) by the ANSI [105] and the DEA-1 by the ISO [43] remained a worldwide standard for a long time until it was replaced by the new Advanced Encryption Standard (AES) on October 2000. However, it is expected that DES will remain in the public domain for a number of years. It provides a basis for comparison for new algorithms and it is still used in IPSec protocols, ATM encryption, the secure socket layer (SSL) protocol and in TripleDES. A detail description of DES algorithm can be seen at [90, 56, 97]. A brief introduction about DES in this chapter is mainly based on [90].

DES uses a 64-bit block for the key; however, 8 of these bits are used for odd parity and are, therefore, not counted in the key length. The effective key length is then calculated as 56 bits, giving 256 possible keys. DES is a block cipher: It encrypts/decrypts data in 64-bit blocks using a 64-bit key. DES is a symmetric algorithm: The same algorithm and 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 subkey is derived from the original key through a process called *key scheduling*. Key

scheduling algorithm for encryption and decryption is the same however round keys for decryption are used in reverse order. Figure III.9 demonstrate the basic algorithm flow for both the encryption and key schedule processes.

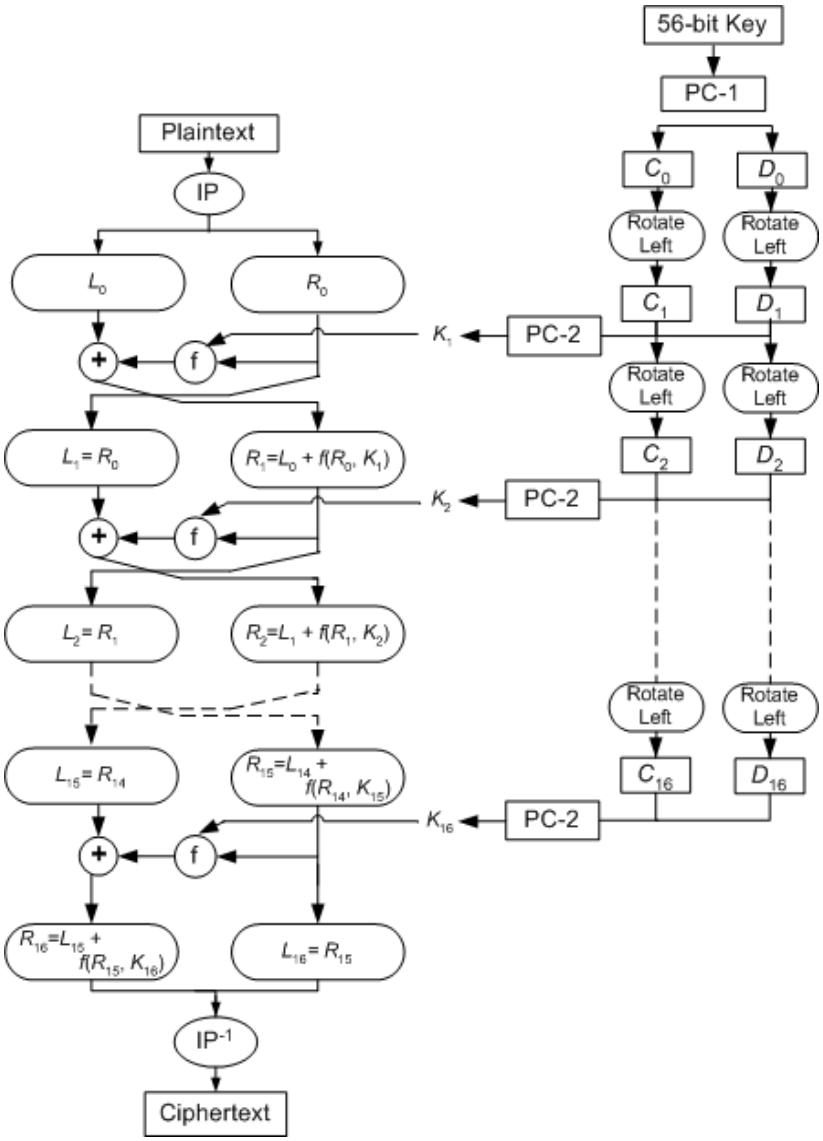


Figure III.9. DES Algorithm

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 succeeding 16 iterations of

the function f_k which is given by ($n = 0$ to 15):

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} + f(R_{n-1}, K_n) \end{aligned} \tag{III.1}$$

After 16 iterations, the contents of the right and left half registers are subjected to a 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 followed by the process of generating sub-keys from the original key.

III.3.1 The initial permutation (IP^{-1})

The initial permutation is the first operation applied to the input block before iteration 1. It transposes the input block as described in Table III.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. The initial permutation has no effect on DES security and its primary purpose is to make it easier to load plain-text into a DES chip in a byte-sized pieces. Implementing it in hardware is trivial but cumbersome in software.

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

Table III.2. Initial permutation for 64-bit input block

III.3.2 Structure of the function f_k

The 64-bit output from the initial permutation is divided into two halves L_0 of 32 bits and R_0 of 32 bits, as shown in Figure III.9. Both halves go through the 16 iterations of the function f_k (Eq. III.1) which is described as follows:

For the first iteration, R_0 of 32 bits 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).

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 III.3. E-bit selection

The Expansion Permutation (Permutation E)

This operation expands right half 32-bit to 48 bits. Some bits are therefore repeated and the order of the bits is also changed as shown in Table III.3.

Table III.3 shows the position relationship between inputs and outputs. For example, the bit in position 3 of input block moves to position 4 of the output block, and the bit in position 21 of the input block moves to position 30 and 32 of the output block. The repeated bits and their positions in output block can easily be seen as they are outside the squares in boldface letter as shown Table III.3.

This operation has two purposes. First, it makes the size of right half register equal to key size for XOR operation. Second, it provides longer result that can be compressed during substitution operation.

The output 48-bit is XORed with the 48-bit round key. We now have 48 bits, or eight groups of six bits. The six bits of eight groups are used as addresses in tables called, S-Boxes. Each group of six bits is substituted to a group of 4 bits.

The S-Box substitution

DES S-box is 64-entry table arranged into four rows and sixteen columns. The input is 6-bit address and output of each S-box is 4-bit long. This way, 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. Hence the S-box will substitute 101011, the entity located at row 4th (11) and column 6th (0101). To substitute 48-bit input, DES contains eight S-boxes each of size $64 \times 4 = 256$ bits occupying a total of 2k memory as shown in Table III.4

The 32-bit S-Box output undergoes through another permutation, which is called P-Box Permutation.

The P-Box permutation

This is called *straight permutation* or just permutation. The 32-bit output of the S-box substitution is permutated according to P-box as shown in Table III.5. Table III.5 shows the position to which each bit moves. For example, bit 21 moves to bit 4, while bit 4 moves to bit 31. No bits are used twice and no bits are ignored.

The output 32-bit after this permutation is XORed with the 32-bit of L_0, achieving R_1. This is the end of first iteration. Let us remember that all the operations were executed on the contents of 32-bit register R_0 and we achieve R_1. In the next

Row	Column 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	S-Box
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7	
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8	S_1
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	
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5	S_2
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	
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1	S_3
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	
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9	S_4
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	
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6	S_5
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	
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8	S_6
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	
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6	S_7
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	
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2	S_8
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	

Table III.4. DES S-boxes

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

Table III.5. Permutation P

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 was used for R_0 and so on for 16 iterations. At the end of the 16^{th} iterations 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 $R_{16}L_{16}$. A final permutation IP^{-1} is then applied.

The Final permutation

This permutation is exactly inverse of the initial permutation and is described in Table III.6. Referring to Fig. III.9, left and right halves are not changed after the

last round of DES, instead the concatenated block R16L16 is used as input to the final permutation. Exchanging halves and shifting around the permutation would yield the same results. This way, the same algorithm can be used for both encryption and decryption.

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

Table III.6. Inverse permutation

This completes the encryption process for a single block. Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the sub-keys are applied.

III.3.3 Key schedule

The sub-keys for all the 16 rounds are derived from the original key as shown in Figure III.9. First the 64-bit DES key is reduced by ignoring every 8th bit governed by the Table III.7 (Permuted Choice One PC-1). After 56-bit key is extracted, the

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

Table III.7. Permuted choice one PC-1

48-bit sub-keys for all rounds are generated as follows:

The 56-bit key is divided into two halves C_0 and D_0. At each round, the two halves are subjected to a circular left shift or rotation by either one or two bits, depending on the round as given in Table III.8.

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

Table III.8. Number of key bits shifted per round

The shifted values serve as input to the next round and also serve as input to the Permutated Choice Two (PC-2) as given in Table III.9. The resulting 48 bits is the required round key.

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

Table III.9. Permutated choice two (PC-2)

III.4 FPGA implementation of DES algorithm

In this section DES implementation is described on reconfigurable hardware platform. All the design steps for the development of FPGA architecture are explained. Some useful design techniques for the improvement of design performance are illustrated. Performance results and comparison with the previous FPGA implementations of DES are presented at the end of this Section.

III.4.1 Design steps

In general, most of the design tools follow the basic six steps for an FPGA design as shown in Fig. III.10. Those steps are not executed in a specific order but they can be repeated to improve design's performance. A short description of each step is provided below.

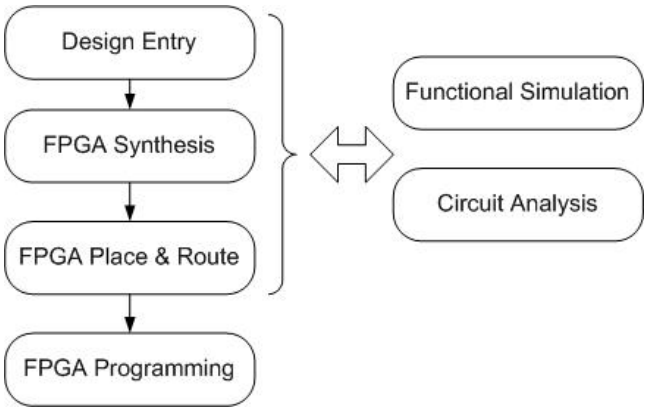


Figure III.10. Design flow

1. **Design Entry** : There are two standard ways to specify a design on FPGA.

- 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. All the existing software management methods well hold for those languages as any change or modification is being treated like in other traditional languages. However, this really depends on the quality of the tools used for synthesis.
- 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:** It checks the logical correctness of an FPGA design. After the design has been represented by using HDLs or schematic, it is necessary to verify if such description meets the design specifications.

3. **FPGA synthesis:** Synthesis converts design entry 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 netlist into FPGA real resources. This is an important step based on design entry. An FPGA designer must keep in mind the basic structure of the target device while writing HDL code or using schematic libraries of the device for design entry.

4. **FPGA place and route:** Place and route selects the optimal position for elementary design blocks and minimal interconnection distance between them. Place and route tools normally use device vendor specifications. Usually they provide hand-placement and also automatic features for optimizing critical paths for speed or area.

5. **Circuit analysis:** Circuit analysis determines different features of the circuit related to efficiency. 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, power analysis provides power consumption by the circuit. There exist special tools to determine and to improve such type of circuit features.

6. **Programming FPGA device:** Programming FPGA implies downloading bit stream codes from the last 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.

III.4.2 Design techniques

Here we discuss few design techniques, in general for symmetric-key block ciphers and in particular for DES on FPGAs. It has been observed that better design techniques for both design entry and design implementation play an important role for optimizing circuits. A short description of those optimizing techniques is provided here.

- **Design strategy:** Design strategy is application dependent. Time critical applications care of timings and other factors like hardware resources or cost of the device are of secondary importance. Cryptographic algorithms (especially block ciphers) have iterative nature and n iterations of the same algorithm are made. It is therefore possible to implement just one round and consume n cycles (iterative looping) or implement n rounds of the algorithm (pipelining) to achieve high timing performances. DES has 16 rounds.

- **Choice of target device:** The choice of the target device (FPGA) is based on the design strategy. A large number of FPGAs are available in the market by various manufacturers. The basic structure of all FPGAs is the same however some of them offer additional features like built-in-memories, built-in-arithmetic functions, etc. FPGAs are of different sizes depending on the prices. For DES implementation, the chosen target device XCV400fg456 belongs to VirtexE family of devices. Virtex and VirtexE family of devices contains more than 280 built-in memory modules called as BlockRAMs (BRAMs). Those memories are fast access memories and can be occupied for DES SP-boxes. However, due to small memory requirements, BRAMs were not used. Instead, FPGA CLBs were 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 in reducing design size and path delays in the circuit. It may not be a good effort to implement a fast software code in hardware. Software codes are often optimized for 8, 16 or 32 bit general-purpose microprocessors. In hardware, such types of limitations do not exist. A bit-level parallelism can freely be exploited unless one encounters data dependencies or resource limitations. Let us consider an instruction from a software code optimized for 32-bit general-purpose microprocessor:

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

which requires 16 right shifts, logical XOR and then logical AND with $0 \times 0000FFFF$. That instruction in software is an effort to execute an XOR operation for the 16 most significant bits of 32-bit 'left' and 'right' registers. In

hardware description languages, the same instruction can be implemented just caring for language notations. One of the best option is to eliminate the AND and 16 logical Shifts by executing XOR operation directly to the 16 most significant bits of left and right registers, that is, work = left [31:16] XOR right [31:16].

- **Selecting FPGA resources:** An FPGA designer can opt for multiple options for performing a function. For example, two choices for implementing a 2-bit multiplexer are shown in Figure III.11.

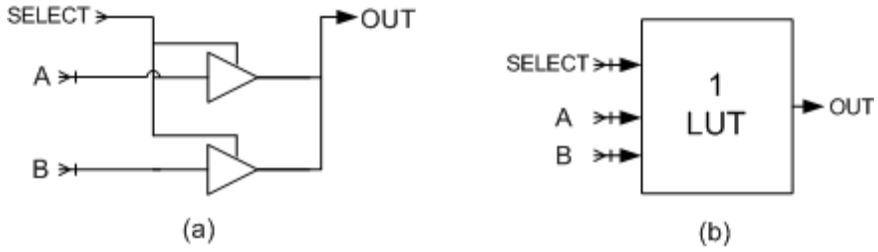


Figure III.11. 2-bit multiplixer using (a) Tristate Buffer. (b) LUT

Figure III.11a 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, our experience shows that, using large number of tri-state buffers slows down the circuit. It is due to 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 III.11b. Using adjacent LUTs for an n to 1 multiplexer would be useful when a circuit is required to 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:** Hardware Description Languages (HDLs) are analogous to other high level languages (C, C++, etc.) with some subtle differences. Both types of language are processed by a compiler. Writing style is the same. However they differ in executing compiled code. 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. Compilers compile an HDL code and provide a list of electronic components in the circuit and also give details of how they are connected. We will focus here in writing an HDL code for an FPGA design.

  An FPGA designer can know about the components (flip-flops, tri-state buffers, LUTs) resulted by an HDL instruction. For example, an *if statement* in HDL

describes a multiplexer or a flip-flop. Now, the frequent use of this statement would insert large number of multiplexers or flip-flops resulting a circuit, which is functionally correct but may or may not be efficient. Here, hardware approach, therefore, defines in thinking about the component first and then writing an HDL code accordingly. In other words if a programmer requires the functionality of a flip-flop than he can search for 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 complete control to the designer over the used and available resources in an FPGA.

III.4.3 DES implementation on FPGAs

Figure III.12 represents a block diagram for DES implementation on FPGA. As it has been earlier discussed that permutation is a simple operation on hardware devices. It can be implemented by changing bit positions for the outgoing bus (change of wires), hence it is free of cost. DES includes lot of permutation operations (initial, final, permutation E, permutation P). The building blocks for those operations in Figure III.12 are therefore symbolic representations having no logic inside. Each S-Box in DES occupy 64×4=256-bit memory, a total of 2048 bits for eight S-Boxes. The implementation for S-Boxes consumes only 32 CLB slices configured in memory mode. Temporary registers and XOR blocks are the other operations, which occupy FPGA resources.

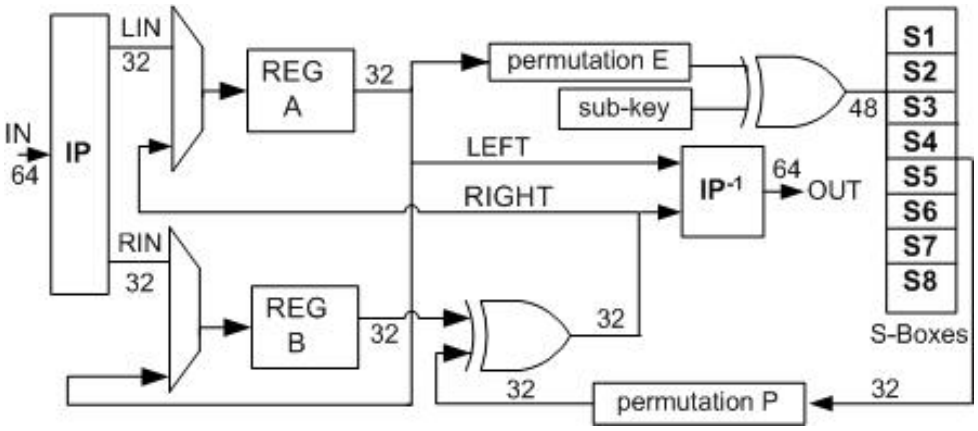


Figure III.12. DES implementation on FPGA

Three inputs: Chip Enable (CE), Clock (CLK), input data (IN) and the only output (OUT) are the four pins of DES chip. Chip enable (CE) activates the timing logic as well as the rest of the circuitry, in its low state (when it is '0'). The external clock CLK is the master clock for the whole circuit that is used to generate all the control signals to synchronize the data flow. When CE enables the circuit,

The 64-bit at the input are permuted and divided into two halves RIN and LIN. At the first rising edge of the clock both halves are being transferred to the output of the two registers REGA and REGB. The right halves (REGA output) go through a number of operations: Permutation E; addition with sub-key; substitution (through S-Boxes); Permutation P and; addition with the original left half (REGB output). Before the next clock comes, the old right half (RIGHT) is the input of the register REGB and the new left half (LEFT) is the input of the register REGA. The sixteen iterations are then executed. After 16th clock cycles the two halves RIGHT and LEFT are concatenated and the resulting block goes through the inverse permutation (IP^{-1}) resulting one encryption for a 64-bit input block. Notice that the usage of an eight DES S-Boxes parallel structure, results in a significant reduction of the critical path for encryption/decryption.

III.4.4 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. That is useful feature by the design tool to verify circuit performance taking into account all circuit overheads. Both functional and timing verifications must be performed for a successful design implementation. For both cases, test vectors are used for the verification of the results. Table III.10 shows the test vectors used in our DES implementations.

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

Table III.10. Test vectors

Figure III.13 and Figure III.14 show the results for the functional simulation and the timing verification for DES implementation on FPGA. Notice that the difference between Figure III.13 and III.14. Time delays in Figure III.14 are more clear. Timings (clock cycles) presented in Figure III.14 are the real timings for the circuit.

III.4.5 Performance results and comparison

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

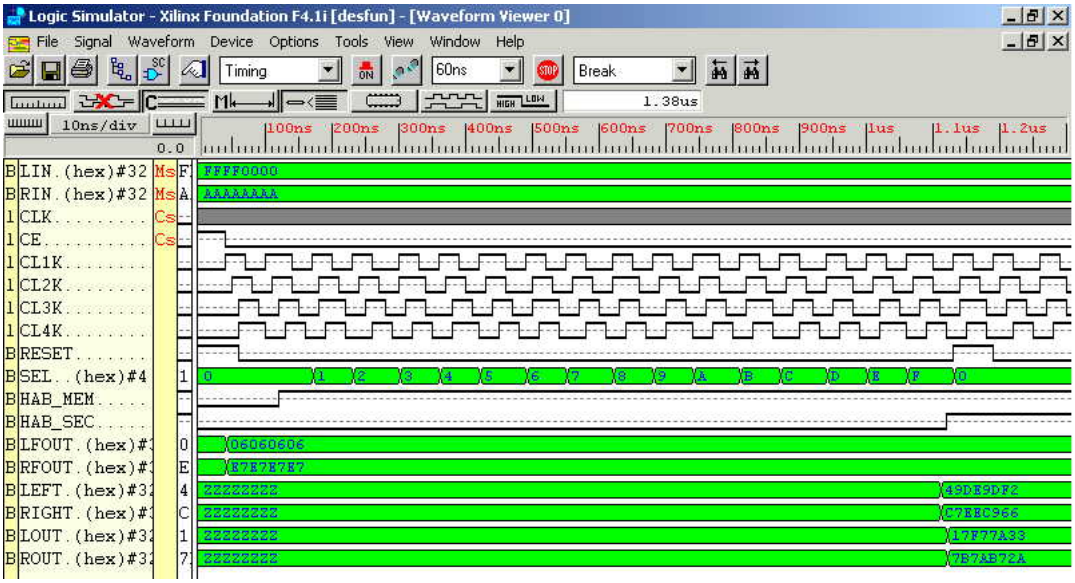


Figure III.13. Functional simulation

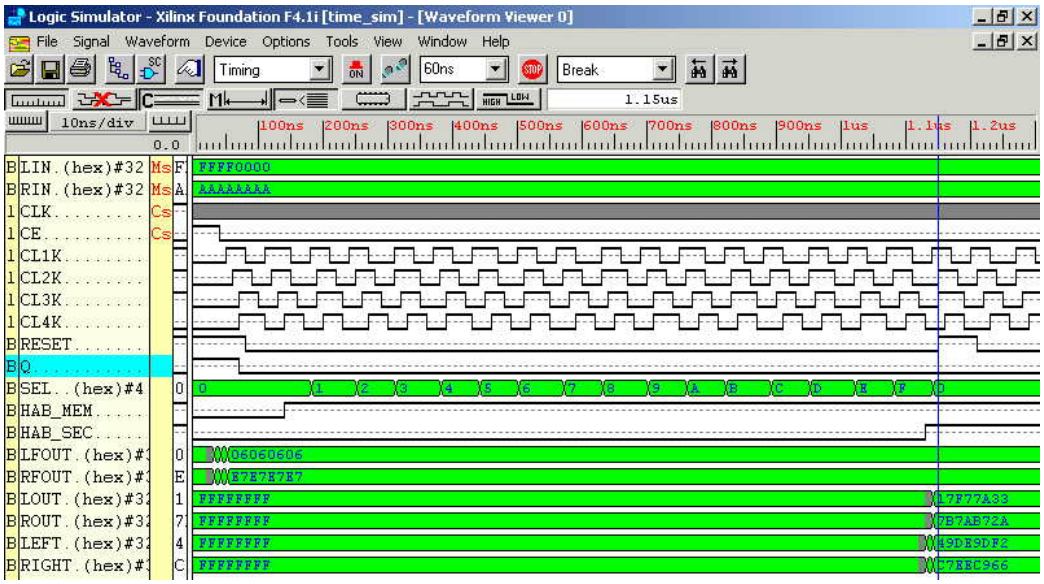


Figure III.14. Timing verification

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.

Table III.11 shows the performance figures for some representative DES hardware

implementations. Notice that the achieved results are competitive with the existing implementations.

Author	Device	Design Strategy	Area (A) CLB slices	Freq. (MHz)	Throughput (Mbits/s) (T)	T/A
Kaps and Paar [45]	XC4028EX	pipeline (4-round)	741	25.18	402.7	0.54
Free-DES [13]	XCV400	Pipeline	5263	47.7	3052	0.57
McLoony et al. [55]	XCV1000	Pipeline	6446	59.5	3808	0.59
Patterson (Jbits) [71]	XCV150	pipeline SW/HW	1584	168	10752	6.78
Wong et al. [103]	XC4020E	Iterative	438	10	26.7	0.06
This work	XCV400e	Iterative	117	68.05	274	2.34

Table III.11. Recent DES reconfigurable hardware implementations

Several FPGA implementations of DES have been reported in the literature achieving throughput ranges from 26 to 10752 Mbits/s. A DES implementation in [13] is a free DES core which achieves a data rate of 3052 Mbits/s using pipeline approach in ECB mode. A java-based (Jbits) DES implementation in [71] achieves the fastest encryption rate of 10752 Mbits/s. It implements all DES primitives in FPGA while key schedule in software. The communication between the two operations is made through a Java-based Application Programming Interface (API) which is used for the runtime creation and modification of the configuration bit-stream. FPGA implementation of DES in [45] implement both 2-stage and 4-stage pipeline approaches obtaining throughput of 183.8 Mbits/s and 402.7 Mbits/s respectively. Almost all FPGA architectures for DES implement partial or fully pipeline approaches. Only the design in [103] is a one round DES implementation on a single-chip FPGA. A fair comparison is possible with this design only. The design was implemented on XC4020E occupying 438 CLB slices. It takes 24 cycles to complete encryption for one single data block achieving a throughput of 26.7 Mbits/s. Hence the Throughput/Area factor is 0.06. Our DES implementation improves both the area and throughput factors consuming only 165 CLB slices on XCV400 showing a throughput of 274 Mbits/s. The Throughput/Area factor for our design is 2.34. Comparing our architecture with the design in [103], we get a speedup improvement of almost 10 times in throughput occupying four times less CLB slices. In fact our design ranks second considering as a figure of merit the Throughput/Area Factor is really convincing.

III.5 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. Some useful properties for implementing block ciphers in FPGAs were discussed. The design steps and some design techniques for an FPGA implementation were described. A general guideline, therefore, developed for the implementation of block ciphers in reconfigurable devices. Same guideline was then applied for DES implementation resulting an efficient and compact DES core on reconfigurable hardware platform. From Table III.11, it can be seen that our design showed improvements both in time and area when compared with other reported reconfigurable hardware implementations of DES.

Our architecture can be improved to offer even better results in terms of achieved throughput. The most obvious extension is to design a fully pipelined architecture in order to obtain a higher throughput at the price of area.

Chapter IV

Architectural Designs For Advanced Encryption Standard

This chapter investigate the significance of FPGA implementations for new Advanced Encryption Standard (AES). Multiple architectural implementation options are explored for AES. The results of each AES implementation are evaluated in an effort to determine the most suitable candidate architecture for variety of different cryptographic applications restricted to area or time. The performance results are well compared with the existing FPGA implementations of AES algorithm showing significant improvements both in space and time.

IV.1 Introduction

On October 2000, a new Advanced Encryption Standard (AES) 'Rijndael' (based on the names of two inventors 'Rijmen' and 'Daemen') was selected by NIST [65] replacing Data Encryption Standard DES. From its evaluation process to post selection period the algorithm has been implemented on all kind of hardware and software platforms. Gladman [34] and Guido et al [17], 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 [42, 99, 51, 79]. Performance of VLSI implementations ranges from 2 to 7.5 Gbits/s.

Similarly, various AES FPGA implementations have been reported at [33, 25, 30, 52]. Those are one round (*iterative*) or n rounds (*pipeline*) FPGA implementations

optimized for encryption or encryption/decryption processes. Since FPGA devices used in previous works are rather different, performance results are broadly variable; they range from 300 Mbps to 6 Gbit/s, approximately.

Asymmetric characteristics of AES encryption and decryption processes limit the implementation of high-performance AES cores. Each step in AES for encryption has its inverse (in GF(2^m)) 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 even impossible or difficult to meet in several FPGA families of devices. This is one of the reasons why designs reported [33, 25, 30], have considered only the encryption part of AES. Both designs at [33, 25] are one round implementations with [25] or without [33] key scheduling. The design at [30] implements all AES rounds (pipeline) without key scheduling. Only the design at [52] has reported an FPGA implementation of a fully pipeline AES encryptor/decryptor core.

In this Chapter, various FPGA implementations of AES are presented. Those implementations cover all three basic processes: key scheduling, encryption and decryption, on a single-chip FPGA. Different design architectures are considered by implementing AES encryptor, decryptor and encryptor/decryptor cores separately. Both iterative and pipeline techniques are applied showing time-area tradeoffs. All AES implementations are optimized for low cost, high efficiency or high portability. The rest of this Chapter is organized as follows.

An introduction to AES algorithm is presented in Section IV.2. The difference between AES and Rijndael is clarified. The basic transformations of the algorithm and their effects on the algorithm cryptographic strength are also explained in this Section. Section IV.3 describes various novel techniques for implementing AES on FPGAs. Those techniques help to improve overall algorithm performance by modifying the most costly operations of the algorithm. Section IV.4 deals with AES implementation on FPGAs by taking into account the modifications in AES transformations discussed in the previous Sections. Section IV.5 compares achieved results followed by conclusion remarks in Section IV.6.

IV.2 The Rijndael algorithm

Rijndael algorithm was announced as the winner of the contest for new Advanced Encryption Standard(AES) [65, 24] by NIST [65] on October, 2000. This Section explains the behavior the algorithm. First it is explained the difference between AES and Rijndael. Secondly, the basic structure and building blocks are described. Third, it is specified the round transformation of the algorithm. Then, key schedule is described.

IV.2.1 Difference between AES and Rijndael

Rijndael is a block cipher algorithm with variable block and key lengths. It can process blocks of 128, 192, and 256 bits. All combinations of block and key lengths

are possible. The only difference between Rijndael and AES is the supported range of block length and cipher key length.

key length *(bits)*	Block length *(bits)*		
	128	192	256
128	10	12	14
192	12	12	14
256	14	14	14

Table IV.1. Selection of Rijndael rounds

The AES fixes the block length to 128 bits and supports key lengths of 128, 192 or 256 bits only. The numbers of rounds depend upon the combination of the selected block and key lengths as shown in Table IV.1. From Table IV.1, the allowed number of rounds for this combination are 10. All over this document whenever we use AES, it means a block and key lengths of 128 bits.

IV.2.2 Structure of the AES algorithm

The basic structure of AES algorithm is shown in Figure IV.1.

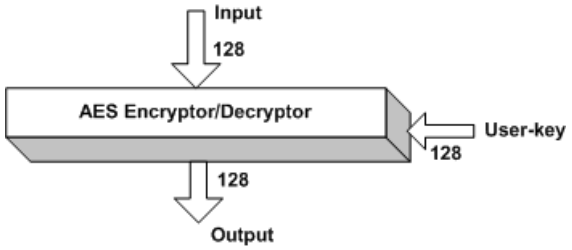


Figure IV.1. Basic structure of Rijndael algorithm.

For encryption the input is a plaintext block and a key block, and the output is ciphertext block. For decryption the input is ciphertext block and a key, the output is plaintext. The basic algorithm flow for encrypting one block of data is shown in Figure IV.2.

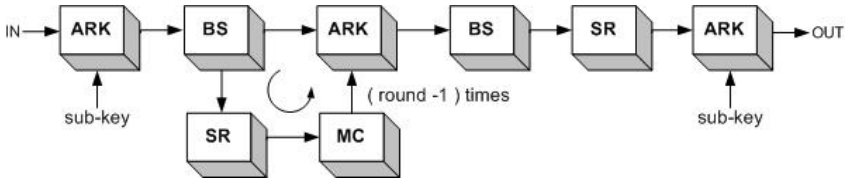


Figure IV.2. Basic algorithm flow.

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 transformation, followed by a main loop where nine iterations called *rounds* are executed. Each *round transformation* is composed of a sequence of four transformations: Byte-Substitution (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*. Finally, a last round consisting of three transformations BS, SR and ARK is executed. The AES decryption algorithm operates similarly by applying the inverse of all the transformations described above in the reverse order.

IV.2.3 The round transformation

The round transformation [24] is a sequence of four transformations BS, SR, MC and ARK. All four transformations contribute in AES strength by inducing confusion and diffusion optimalities. The *confusion* and *diffusion* are two important properties that give some indication about the strength of a cryptographic system. 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. Those, roughly correspond to substitution and permutation. Symmetric ciphers need to be complex, so they cannot be analyzed easily. Also, they need to be simple, enough to be implemented efficiently achieving high encryption rates. For AES, the general criteria for round transformation was invertibility and simplicity besides the step-specific criteria. This Section describes all four steps of round transformation with step-specific criteria to provide an idea how effective AES is at mixing up bits.

ByteSubstitution (BS)

It is mainly a non-linear transformation where each input byte of the State matrix is independently replaced by another byte from a look-up table called *S-Box* as shown in Figure IV.3.

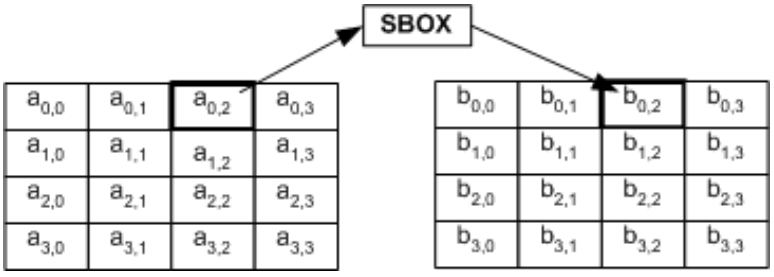


Figure IV.3. BS operates at each individual byte of the state matrix

The *non-linearity* and *algebraic complexity* are the general criteria for designing AES S-Box [24]. The AES S-Box is a 256-entry table composed of 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; followed by an affine transformation over GF(2).

Multiplicative inverse can be found by means of extended Euclidean algorithm [56]. Let MI of the polynomial $a(x)$ is desired then extended Euclidean algorithm can be used to find two polynomials $b(x)$ and $c(x)$ such that:

$$a(x) \times b(x) + m(x) \times c(x) = \gcd(a(x), m(x)) \tag{IV.1}$$

where $\gcd(a(x), m(x))$ represents greatest common divisor of the polynomials $a(x)$ and $m(x)$ which is always 1 iff $m(x)$ is irreducible. Applying modular reduction to Equation IV.1, we get:

$$a(x) \times b(x) = 1 \bmod m(x) \tag{IV.2}$$

which means that $b(x)$ is the inverse element of $a(x)$ (by the definition of multiplication).

The affine transformation f is defined as follows.

$$y = f(x)$$

$$\begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{IV.3}$$

The non-linearity is introduced by applying multiplicative inverse in GF(2^8). The affine transformation has no impact on the non-linearity but contribute in algebraic complexity.

Inverse operation: The inverse S-Box is obtained by applying inverse affine transformations followed by the multiplicative inverse in GF(2^8). The inverse of affine transformation is defined as follows.

$$x = f^{-1}(y)$$

$$\begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} y_7 \\ y_6 \\ y_5 \\ y_4 \\ y_3 \\ y_2 \\ y_1 \\ y_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{IV.4}$$

For both affine and inverse affine transformations, multiplicative inverse is taken in GF(2^8) with irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

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 IV.4.

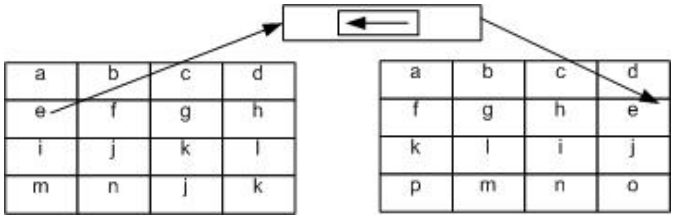


Figure IV.4. ShiftRows operates at rows of the state matrix

The *diffusion optimality* is the design criteria for selecting the offsets. The diffusion optimality require the four offsets have to be different.

Inverse Operation: The inverse operation of ShiftRows is called Inverse ShiftRows (ISR). It is a cyclic shift operation where each row is rotated cyclically to the right using 0,1,2 and 3-byte offset for decryption.

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:

$$c(x) = 03.x^3 + 01.x^2 + 01.x + 02$$

Let $b(x) = c(x).a(x) \bmod x^4 + 1$, then the modular multiplication with a fixed polynomial can be written as shown in Equation IV.5.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{IV.5}$$

MixColumns operates on the columns of the state matrix as shown in Figure IV.5. The design criteria for MixColumns step includes *dimensions*, *linearity*, *diffusion* and *performance on 8-bit processor*. The *dimension* criteria defines transformation operation on 4-byte columns.

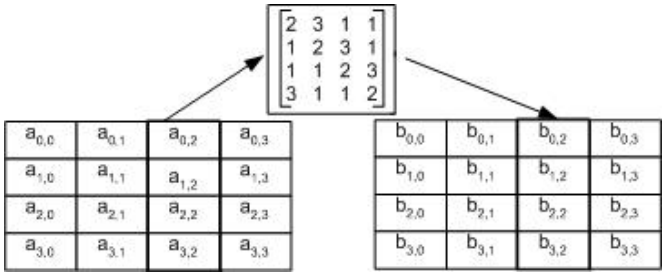


Figure IV.5. MixColumns operates at columns of the state matrix

Inverse operation: The inverse of MixColumns is called (IMC). The constant polynomial $c(x)$ is co-prime to $x^4 + 1$ and therefore invertible denoted by $d(x)$ written as follows.

$$(03.x^3 + 01.x^2 + 01.x + 02).d(x) \equiv 01(\mathrm{mod} x^4 + 1)$$

The $d(x)$ is given by:

$$d(x) = 0B.x^3 + 0D.x^2 + 09.x + 0E$$

In IMC, each column is transformed by multiplying with constant polynomial $d(x)$ written as a matrix multiplication as shown in Equation IV.6.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{IV.6}$$

AddRoundKey (ARK)

In the last step, the output of MC is XOR-ed with the corresponding round key derived from

the user's key. This step is denoted as ARK, is essentially the same for encryption and decryption. Figure IV.6 illustrates the effect of key addition on the state matrix.

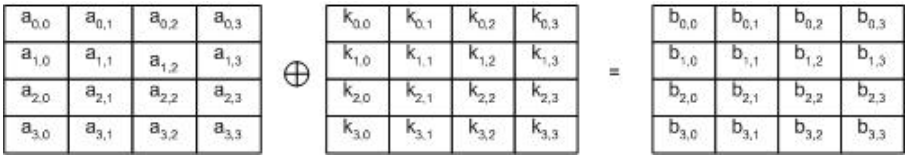


Figure IV.6. ARK operates at bits of the state matrix

IV.2.4 Key schedule

Both encryption and decryption, each require the generation of round keys, called collectively *key schedule*. The round keys are obtained through the expansion of secret user key by attaching recursively the 4-byte word $k_i = (k_{0,i}, k_{1,i}, k_{2,i}, k_{3,i})$ to the user key. The original user key consists of 128 bits 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 as follows:

Let the columns up to $w[i-1]$ have been defined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \textit{otherwise} \end{cases} \tag{IV.7}$$

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 S-Box $S(z)$, $S(w)$, $S(x)$ 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, the columns from w[4] to w[43] are generated from the first four columns. The round key for the ith round consists of the columns

$$(w(4i), w(4i+1), w(4i+2), w(4i+3))$$

IV.3 Novel techniques for efficient implementation of AES round transformation on FPGAs

Section IV.2.3 describes round transformation and impact of four basic transformations (BS, SR, MC, and ARK) on the state matrix. The most important operations for all the 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 resources. The addition is a simple XOR operation. Perhaps the most costly operation is polynomial multiplication in GF(2^8) for BS/IBS. The polynomial multiplication in GF(2^8) for MC/IMC takes advantage of constant multiplication and is relatively less costly however occupies considerable FPGA's resources. Therefore, both BS/IBS and MC/IMC are good candidates for improving overall performance of the round transformation. On the other hand, AES hardware implementation poses a challenge since encryption and decryption

processes are not symmetrical. Designing separated architectures for encryption and decryption processes would imply the allocation of a large amount of FPGA resources. Various techniques can be derived for efficient implementation of BS/IBS and MC/IMC.

In the rest of this Section, two approaches for implementing BS/IBS are described. In the first approach, pre-computed values are stored in FPGA's built-in memory modules to save computational time. The second approach provides an alternative to reduce memory requirements and is based on the on-fly architecture strategy. Similarly, two approaches for MC/IMC implementations are presented. First approach called *classical approach*, deals with the structural 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 algorithm performance by cutting path delays.

IV.3.1 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. This requires memory modules with fast memory access. In FPGAs, there are two ways to organize memory: by using flip-flops and CLBs, and by using FPGAs with built-in memory modules called BRAMs (BlockRAMs). To increase performance several BS operations should be performed in parallel which translates in high memory requirements. A different approach manipulates BS in $GF(2^4)$ and $GF(2^2)^2$ instead of $GF(2^8)$. In this approach, memory requirements are reduced. Clearly, it will occupy more area and does not have fast access time like BRAMs.

S-Box and inverse S-Box based on look-up table

To avoid utilization of considerable amount of FPGA resources, the implementation of S-Box and inverse S-Box can be made using look-up table method. Both may be computed by implementing affine (AF) and inverse affine (IAF) transformations together with a look-up table for multiplicative inverse (MI). In this way, the combination MI + AF provides S-Box for encryption while IAF + MI computes the inverse S-Box for decryption. To use only one MI module for both encryption and decryption, a multiplexer is used to switch the data path for encryption/decryption, as shown in Figure IV.7

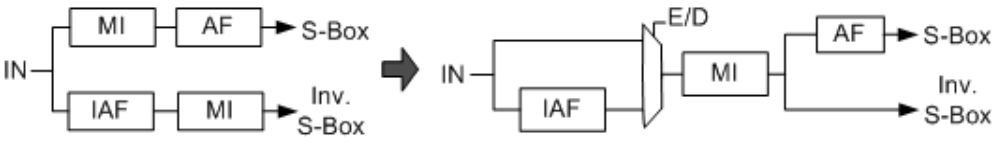


Figure IV.7. S-Box and Inv. S-Box using same look-up table

II. S-Box and inverse S-Box based on composite field techniques

Several authors [69, 62, 79] have designed AES S-Box based on the composite field techniques reported first in [69]. Those techniques use a three-stage strategy:

1. Map the element $A \in \mathrm{GF}(2^8)$ to a 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 [62], 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 *[69, 36] The multiplicative inverse of an element A of the composite field GF$(2^m)^n$, $A \neq 0$, can be computed by,*

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma-1} \bmod P(x) \tag{IV.8}$$

$$\texttt{where} \qquad A^{\gamma} \in \texttt{GF}(2^n) \quad \& \quad \gamma = \frac{2^{nm}-1}{2^m-1}$$

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,

$$A^{-1} = (A^{\gamma})^{-1} A^{\gamma-1} = (A^{17})^{-1} A^{16} \tag{IV.9}$$

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 .

$$\begin{array}{ll} F_1 = \texttt{GF}(2^2) & P_0(x) = x^2 + x + 1 \\ F_2 = \texttt{GF}((2^2)^2 & P_1(y) = y^2 + y + \phi \\ F_3 = \texttt{GF}(((2^2)^2)^2 & P_2(z) = z^2 + z + \lambda \end{array} \tag{IV.10}$$

$$\texttt{where} \quad \phi = \{10\}_2, \quad \lambda = \{1100\}_2$$

The inverse multiplicative over the composite field F_2 defined in the Equation IV.9, can be found as follows:

Let $A \in F_2 = \mathtt{GF}(2^2)^2$ 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 IV.10, then it can be shown that,

$$\begin{aligned} A^{16} &= A_H y + (A_H + A_L) \\ A^{17} &= A^{16}.A = 0.y + (\lambda(A_H)^{16} A_H + (A_L)^{16} A_L) \\ &= \lambda(A_H)^2 + (A_L)^{16} A_L \end{aligned} \tag{IV.11}$$

Figure IV.19 depicts block diagram to three-stage inverse multiplier represented by Equations IV.9 and IV.11. The circuit shown in Figure IV.20 and Figure IV.21 present a gate level implementation of the aforementioned strategy.

As we explained above, 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 $F_2 = \mathtt{GF}(2^2)^2$ 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 the said element A, by the isomorphic functions shown in Equation IV.12 given by [62]:

$$\delta = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \quad \delta^{-1} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{IV.12}$$

The isomorphism function δ and δ^{-1}) can be constructed as follows: Let α and β are roots of a same primitive irreducible polynomial (we use $m(x) = x^8+x^4+x^3+x^2+1$), first search for primitive element α in the field A and a primitive element β in the field B. Once we find δ and δ^{-1}), the definition table for both of them can be determined, where α^k is mapped to β^k or vice versa.

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 IV.9 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).

IV.3.2 MC/IMC implementations on FPGA

For an encryptor/decryptor core MC/IMC steps are implemented separately and can be realized in a small series of instructions. These instructions can be realized

by keeping in mind the basic structure of instructions of CLB (4 input/1 output) to limit path delays and to save space. Let us call this approach MC/IMC classical 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 transformations classic approach

Encryption' MC can be efficiently computed by using only 3 steps [17, 24]: a sum step, a doubling step and a final sum step. Let the element of *State* matrix's column one be $a[0]$, $a[1]$, $a[2]$, 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 IV.13.

$$\begin{array}{c} t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; \\ \begin{array}{lll} v = a[0] \oplus a[1]; & v = xtime(v); & a'[0] = a[0] \oplus v \oplus t; \\ v = a[1] \oplus a[2]; & v = xtime(v); & a'[1] = a[1] \oplus v \oplus t; \\ v = a[2] \oplus a[3]; & v = xtime(v); & a'[2] = a[2] \oplus v \oplus t; \\ v = a[3] \oplus a[0]; & v = xtime(v); & a'[3] = a[3] \oplus v \oplus t; \end{array} \end{array} \tag{IV.13}$$

Here $xtime(v)$ represents the field multiplication of $02 \times v$, where 02 stands for the constant polynomial x in GF(2^8). To enhance time performance, the above computation can be re-structured. Further optimization consists on embedding ARK step to fully exploit 4-input FPGA slice resources as shown in Equation IV.14.

$$\begin{array}{lll} v = a[1] \oplus a[2] \oplus a[3]; & xt_0 = xtime(a[0]); & a'[0] = k[0] \oplus v \oplus xt_0 \oplus xt_1; \\ v = a[0] \oplus a[2] \oplus a[3]; & xt_1 = xtime(a[1]); & a'[1] = k[1] \oplus v \oplus xt_1 \oplus xt_2; \\ v = a[0] \oplus a[1] \oplus a[3]; & xt_2 = xtime(a[2]); & a'[2] = k[2] \oplus v \oplus xt_2 \oplus xt_3; \\ v = a[0] \oplus a[1] \oplus a[2]; & xt_3 = xtime(a[3]); & a'[3] = k[3] \oplus v \oplus xt_3 \oplus xt_0; \end{array} \tag{IV.14}$$

The same strategy applied above for MC would yield up to seven steps to compute IMC: four sum steps and three doubling steps. The difference is due to the fact that coefficients in Equation IV.6 have a higher Hamming weight than the ones in Equation IV.5. To overcome this drawback, we use the strategy depicted in Equation IV.15 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 separate implementation of ARK is made.

$$
\begin{array}{rll}
 & \textit{Step 1} & \textit{Step 2} \\
t & = a[0] \oplus a[1] \oplus a[3] & \\
s_0 & = xtime(a[0]); & s'_0 = xtime(s_0); \\
s_1 & = xtime(a[1]); & s'_1 = xtime(s_1); \\
s_2 & = xtime(a[2]); & s'_2 = xtime(s_2); \\
s_3 & = xtime(a[3]); & s'_3 = xtime(s_3);
\end{array}
$$

$$
\begin{array}{l}
\textit{Step 3} \\
u = s'_0 \oplus s'_1 \oplus s'_2 \oplus s'_3; \\
v = s_0 \oplus s_1 \oplus s'_0 \oplus s'_2; \\
v = s_1 \oplus s_2 \oplus s'_1 \oplus s'_3; \\
v = s_2 \oplus s_3 \oplus s'_0 \oplus s'_2; \\
v = s_3 \oplus s_0 \oplus s'_1 \oplus s'_3;
\end{array}
$$

$$
\begin{array}{lll}
\textit{Step 4} & \textit{Step 5} & \textit{Step 6} \\
u = xtime(u); & t' = t \oplus u; & a'[0] = a[0] \oplus t' \oplus v \oplus k[0]; \\
 & & a'[1] = a[1] \oplus t' \oplus v \oplus k[1]; \\
 & & a'[2] = a[2] \oplus t' \oplus v \oplus k[2]; \\
 & & a'[3] = a[3] \oplus t' \oplus v \oplus k[3];
\end{array} \tag{IV.15}
$$

MC and IMC transformations 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 this complexity difference, we use the following strategy.

We observe that it should exist a 4×4 byte matrix $D(x)$ in GF(2^8) such that the constant MC matrix of Equation IV.5 can be related to the constant matrix of Equation IV.6 as,

$$
\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} D(x) \tag{IV.16}
$$

Using the fact that both constant matrices in Equation IV.16 are the inverse of each other in the finite field $F = \text{GF}(2^8)$. We can solve Equation IV.16 using irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ [24] for the first column of $D(x)$ as shown in Equation IV.17.

$$
\begin{bmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 0E \\ 09 \\ 0D \\ 0B \end{bmatrix} \tag{IV.17}
$$

Where $d_{i,0}$, $i = 0, 1, 2, 3$ represent the four coefficients of the first column of $D(x)$. It follows that Equation IV.17 has a unique solution in the finite field F as given in Equation IV.18,

$$d_{0,0} = 5 \quad d_{1,0} = 0 \quad d_{2,0} = 4 \quad d_{3,0} = 0 \tag{IV.18}$$

Hence Equation IV.16 can be re-written as shown in Equation IV.19.

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} 05 & 00 & 04 & 00 \\ 00 & 05 & 00 & 04 \\ 04 & 00 & 05 & 00 \\ 00 & 04 & 00 & 05 \end{bmatrix} \tag{IV.19}$$

Equation IV.19 suggests an efficient way to compute IMC by re-using the MC transformation to obtain IMC constant matrix. This is useful since the second matrix in the right side of Equation IV.19 is very light from the computational point of view as compared to the original constant matrix for IMC.

IV.3.3 Key schedule optimization

The original user key consists of 128 bits arranged as a 4 x 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 as follows:

Let the columns up to $w[i-1]$ have been defined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if i mod } 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \textit{otherwise} \end{cases} \tag{IV.20}$$

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 [24].

Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be represented as:

$$w[0] = \begin{bmatrix} k_0 \\ k_4 \\ k_8 \\ k_{12} \end{bmatrix} \qquad w[1] = \begin{bmatrix} k_1 \\ k_5 \\ k_9 \\ k_{13} \end{bmatrix}$$
$$w[2] = \begin{bmatrix} k_2 \\ k_6 \\ k_{10} \\ k_{14} \end{bmatrix} \qquad w[3] = \begin{bmatrix} k_3 \\ k_7 \\ k_{11} \\ k_{15} \end{bmatrix} \tag{IV.21}$$

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 IV.22.

$$\begin{array}{llll} \textit{Step 1} & \textit{Step 2} & \textit{Step 3} & \textit{Step 4} \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_4 = k_4 \oplus k'_0; & k'_8 = k_8 \oplus k'_4; & k'_{12} = k_{12} \oplus k'_8; \\ k'_1 = k_0 \oplus SBox(k_{14}); & k'_5 = k_5 \oplus k'_1; & k'_9 = k_9 \oplus k'_5; & k'_{13} = k_{13} \oplus k'_9; \\ k'_2 = k_0 \oplus SBox(k_{15}); & k'_6 = k_6 \oplus k'_2; & k'_{10} = k_{10} \oplus k'_6; & k'_{14} = k_{14} \oplus k'_{10}; \\ k'_3 = k_0 \oplus SBox(k_{12}); & k'_7 = k_7 \oplus k'_3; & k'_{11} = k_{11} \oplus k'_7; & k'_{15} = k_{15} \oplus k'_{11}; \end{array} \quad \text{(IV.22)}$$

But as we have mentioned before, in a typical FPGA device, 4 input bits 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 IV.22 can be rewritten as it is shown in Equation IV.23 for the first row. Parallel computations are applied to obtain k'_4, k'_8, and k'_{12}.

$$\begin{array}{ll} \textit{Step 1} & \textit{Step 2} \\ & k'_4 = k_4 \oplus k'_0; \\ k'_0 = k_0 \oplus SBox(k_{13}) \oplus rcon; & k'_8 = k_4 \oplus k_8 \oplus k'_0; \\ & k'_{12} = k_4 \oplus k_8 \oplus k_{12} \oplus k'_0; \end{array} \quad \text{(IV.23)}$$

IV.4 AES implementations on FPGAs

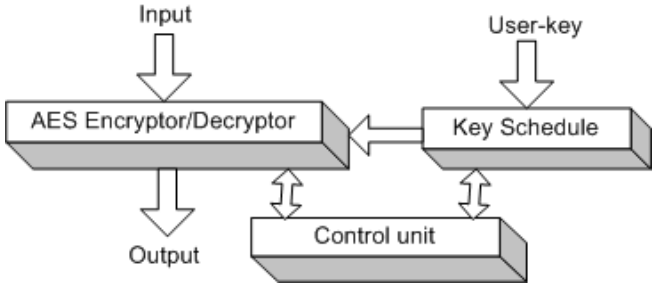


Figure IV.8. Basic organization of a block cipher

The basic organization of the hardware implementation of an AES algorithm is shown in Figure IV.8 which represents three blocks encryptor/decryptor unit, key scheduling unit, and a control unit for synchronizing the flow of data between them. The control unit is the default unit of each application however, three main processes, which participate for AES, are:

- Key Schedule
- Encryption.
- Decryption

For AES implementation, the above three processes can be implemented separately or totally on a single-chip FPGA as shown in Table IV.2. Table IV.2 details all implemented AES designs in this Section. It provides four parameters: design (Sec.IV.4), based on Section (Sec. IV.3), E/D/K module (encryption/decryption/key schedule) and architecture (encryptor, decryptor or encryptor/decryptor core).

The variety of different AES architectures presents area-time tradeoffs. A separate implementation of AES encryptor or decryptor core would be less complex and efficient. However, a single-chip FPGA implementation of AES encryptor/decryptor will give out a complete solution. Moreover, by implementing a single copy of common operations between encryption and decryption, both complexity and efficiency factors could be improved. For AES, the key schedule process is quite different for an encryptor, decryptor or encryptor/decryptor cores. The use 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 to select encryption/decryption key of his own choice.

Table IV.2. A roadmap to implemented AES designs.

Design	Based on the Section	E/D/K Module	Architecture
Sec. IV.4.1	Sec. IV.3.3	(Key schedule)	For iterative & pipeline encryptor cores only
Sec. IV.4.1	Sec. IV.3.3	(Key schedule)	For Pipeline encryptor/decryptor cores
Sec. IV.4.2	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table MC classic	Encryptor core (Iterative)
Sec. IV.4.2	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. IV.4.3	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table MC classic	Encryptor/decryptor core (Pipeline)
Sec. IV.4.3	Sec. IV.3.1 Sec. IV.3.2	S-box Composite field MC classic	Encryptor/decryptor core (Pipeline)
Sec. IV.4.4	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table Modified MC/IMC	Encryptor/decryptor core (Pipeline)
Sec. IV.4.4	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table MC classic	Encryptor core (Pipeline)
Sec. IV.4.4	Sec. IV.3.1 Sec. IV.3.2	S-box Look-up table Modified IMC	Decryptor core (Pipeline)

The efficiency is another important factor for AES implementations on FPGAs. Several approaches can be followed to implement AES on hardware to achieve variable performance results. An iterative looping design (IL), implements only one round and n iterations of the algorithm are carried out by feeding back previous round results. The design utilizes less area but the achieved throughput would be too low. Sub-pipelining (SP) is achieved by placing buffers between different stages of a

round, hence reducing the pipeline's delay but at the same time increases the number of clock cycles needed to perform an encryption and by adding an area penalty for buffers' implementation. In a pipeline design (PP), rounds are replicated and registers are provided between the rounds to control the flow of data. This design offers high throughput but area requirements might be too high for some devices.

In the rest of this section some promising AES implementations on a single-chip FPGA are presented. Those AES implementations consist of 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 IV.2 provides a roadmap to all implemented AES designs. The key schedule implementations for encryptor, decryptor and encryptor/decryptor cores are also presented.

The Xilinx Foundation Tool F4.1i is used for implementing, testing and verificating the results. All designs are either coded in VHDL or by using libraries of the target devices. The Coregenerator is another tool used for design entry.

IV.4.1 Key schedule algorithm implementations

Let the user key consisting of 16 bytes be arranged as:

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_{10} & k_{11} \\ k_{12} & k_{13} & k_{14} & k_{15} \end{bmatrix} \tag{IV.24}$$

Then the process of generating next round key is optimized as discussed in Section IV.3.3 and is shown in Figure IV.9. 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 IV.9, 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 XOR_ed 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, keys must be available to the corresponding rounds, therefore, the management for latching round keys is different for different implementation's strategies. 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 fed back to KGEN block to

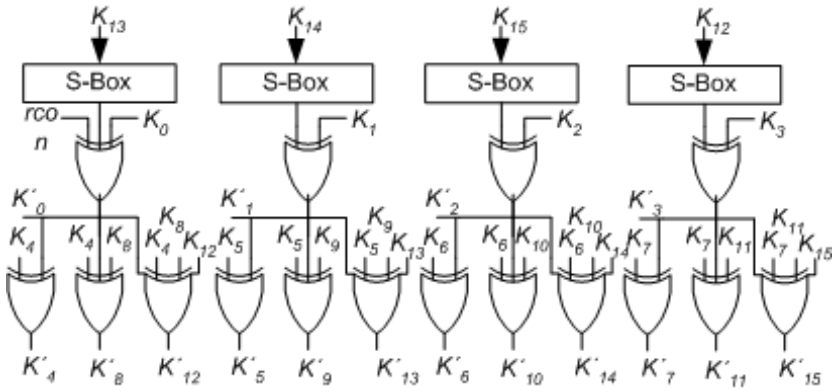


Figure IV.9. KGEN architecture

prepare for next round key as shown in Figure IV.10. The 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.

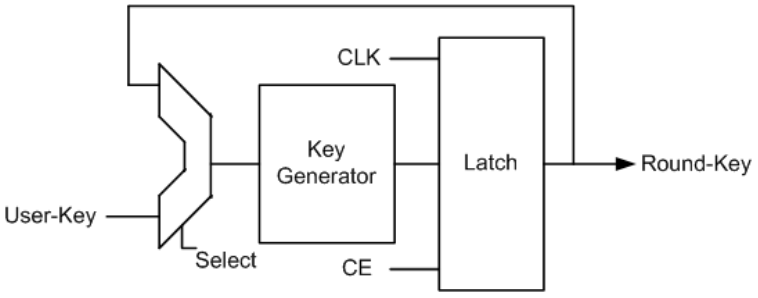


Figure IV.10. Key schedule for an encryptor core in iterative mode

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 IV.11. The internal structure of each block is the same as shown in Figure IV.10, 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.

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 because they are required in reverse order for decryption. The process of key generation is the same as explained

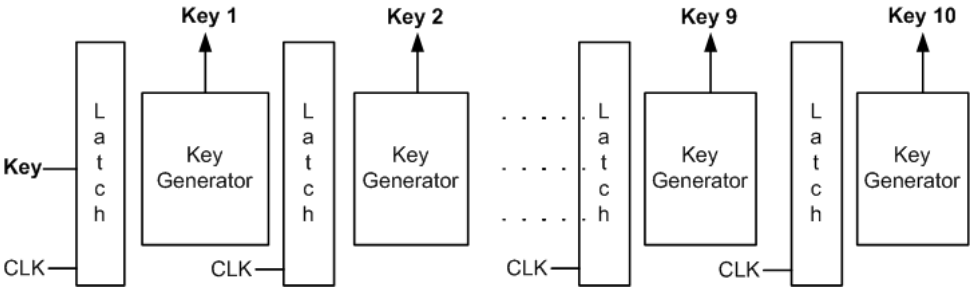


Figure IV.11. Key schedule for a fully pipeline encryptor core

above, however, round keys are stored in the registers for encryption and decryption in ascending or descending order respectively as shown in Figure IV.12. Besides this difference, the same blocks can be used for encryption and decryption processes.

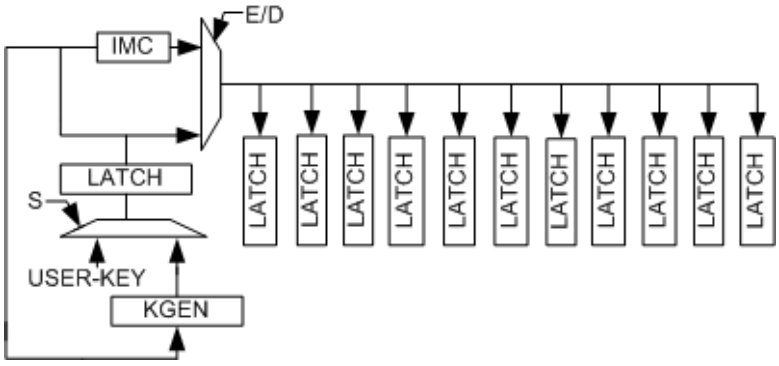


Figure IV.12. Key schedule for a fully pipeline encryptor/decryptor core

As shown in Figure IV.12, round keys are generated by KGEN block as explained above with two modifications. The first one is the generation of select signals (s_i) for latches through an up/down counter. Those select signals provide a complete control for storing round keys in the corresponding registers depending upon encryption/decryption processes. The second modification is the addition of IMC step. The multiplexer allows to pass round keys directly for encryption. For decryption, IMC operation is performed before all the round keys are stored in their registers except the first and last round keys according to the theoretical descriptions for AES decryption process.

Some modifications are introduced in IMC as discussed in Section IV.3.2. Those modifications (ModM) are also introduced in IMC step for key scheduling and serve for the second AES encryptor/decryptor design explained in the next Section. For that design, key schedule procedure will remain the same as explained above for an encryptor/decryptor core except the change in IMC as shown in Figure IV.13.

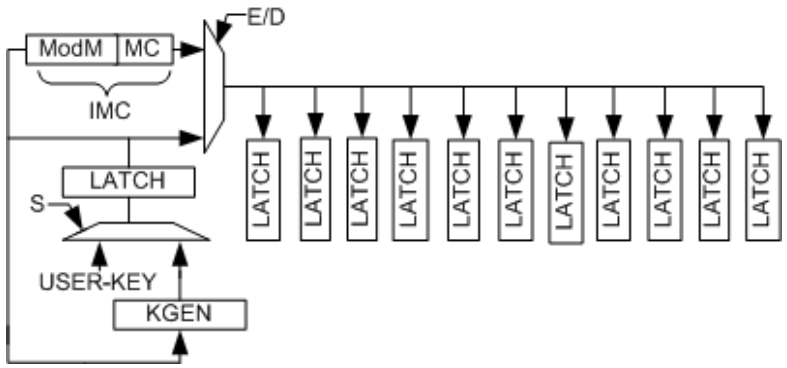


Figure IV.13. Key schedule for a fully pipeline encryptor/decryptor core with modified IMC

IV.4.2 AES encryptor cores - iterative and pipeline approaches

For an encryptor core, the AES implementations on FPGAs are carried out in two modes: iterative and pipeline. The area requirements are reduced for an iterative architecture but low timing performances are achieved. On the other hand, high performances are obtained for a pipeline architecture at the price of high area requirements.

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 iterative approach for AES is shown in Figure IV.14.

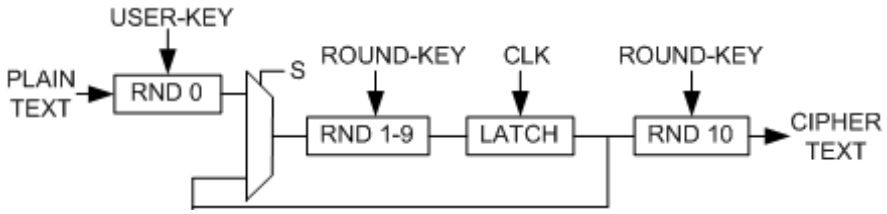


Figure IV.14. Iterative approach for AES encryptor core

An encryption process is presented in Figure IV.14 where RND0 is a simple ARK step: the user-key and plain-text are added. In the next nine cycle, RND1-9 is performed that includes four AES steps (BS,SR,MC,ARK). All nine iterations are made by using new round keys. Each time, an encryption round is executed, the output is stored in the register and provide a feedback for the next encryption round. The multiplexer switch RND0 output at the first time and the RND1-9 results for the next nine cycles. Finally, RND10 is executed which is similar to RND1-9 but excludes MC step.

16 ROMs (256×8) are configured by using CLB in memory mode to perform BS step for RND1-9. RND10 also includes BS step, 16 ROMs are occupied there making a total of 32 ROMs for encryption part only. The key scheduling algorithm also includes BS step for the last four bytes of each round key (IV.4.1) as shown in Figure IV.9 occupying 4 ROMs. The SR step is just a change of wires and is combined with BS step. The MC and ARK steps are joined to reduce area requirements as discussed in Section IV.3.2.

The design is implemented on Xilinx VirtexE FPGA devices (XCV812BEG) and occupies 36 ROMs, 385 I/O Blocks (95%) and 2744 slices (28%). Data blocks are accepted after each 10 clock cycle and similarly output blocks appear after each 10 clock cycles. An allowed frequency for this architecture is 20.192 MHz and the data is processed at a rate of 258.5 Mbits/sec. It does not make use of FPGA dedicated resources (BRAMs, etc.), hence it has a high portability as it can be implemented in virtually every commercial FPGA device available.

Fully pipeline AES encryptor core

For a pipeline architecture, all AES rounds are unrolled. That is achieved by repeating one AES round in iterative mode 11 times as shown in Figure IV.15.

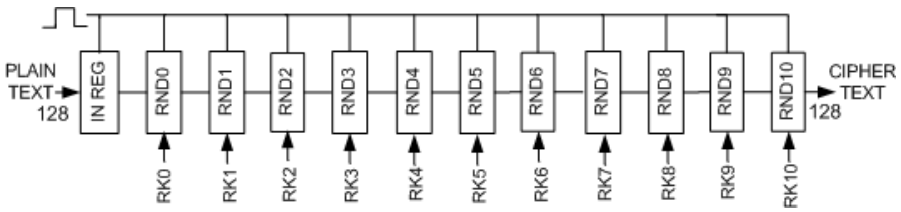


Figure IV.15. Fully pipeline AES encryptor core

Similar to the iterative architecture, RND0 is just ARK step. RND 1-9 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. Critical data path in a pipeline architecture is bigger which causes the design to run at lower speeds. However, by using dedicated memory modules BRAMs, as explained in the introduction Section, it is possible to reduce delays. The Virtex and VirtexE FPGA devices [9] contain more than 280 BRAMs each of 4K. Each dual port BRAM can be configured to two single port BRAMs which reduces half of the memory requirements, a total of 80 BRAMs are 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 one encrypted block starts appearing at each consecutive clock cycle. This is why, pipeline architecture is preferred when high speed is required.

By comparing iterative and pipeline architectures, number of CLB slices occupied by the pipeline architecture seem 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 use of dedicated memory resources (BRAMs) makes the pipeline design importable as it can be targeted to only those FPGA family of devices which already contain similar memory modules. However, it is a highly economical and an efficient architecture.

IV.4.3 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 is 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 rest of three steps SR, MC, and ARK is same as explained in Section IV.4.2. In the following, BS/IBS implementation strategies are discussed.

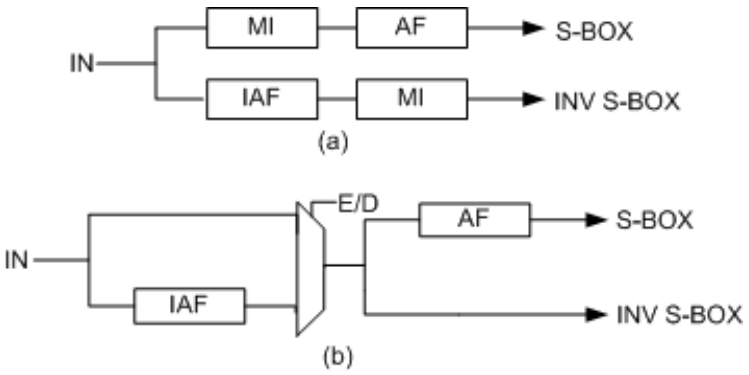


Figure IV.16. S-Box and Inv S-Box using (a) different MI (b) same MI

For encryption, BS implementation can be made by taking Multiplicative Inverse (MI) of input byte in GF(2^8) followed by affine 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 IV.16 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 for MI. By configuring a dual port BRAM into two single port BRAMs, 8 BRAMs are required for a 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 E/D signal. A complete set-up is shown in Figure IV.17

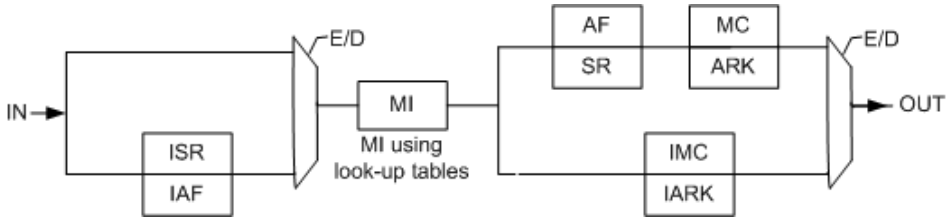


Figure IV.17. Data path for encryption/decryption

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. 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 expedient for those cryptographic applications where time factor really matters.

MI with composite field calculation

This is composite field approach that deals MI manipulation in GF(2^2) and GF(2^4) instead of GF(2^8) as explained in Section IV.3.1. It is 3-stage strategy as shown in Figure IV.18. 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 middle stage with two initial and final transformations is represented in

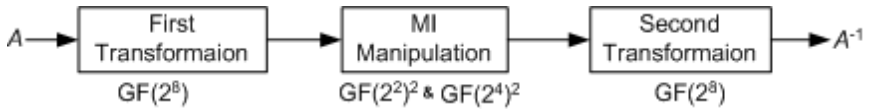


Figure IV.18. Block diagrm for 3-stage MI manipulation

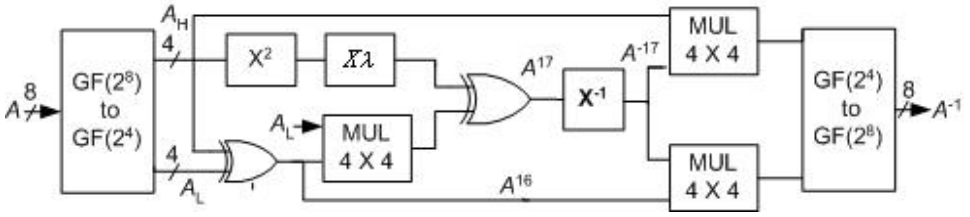


Figure IV.19. Three-stage to compute multiplicative inverse in composite fields.

Figure IV.19 which depicts block diagram to three-stage inverse multiplier represented by Equations IV.9 and IV.11. Data path for encryption/decryption for this approach remains same as the change is introduced in MI manipulation.

The circuit shown in Figure IV.20 and Figure IV.21 present a gate level implementation of the aforementioned strategy.

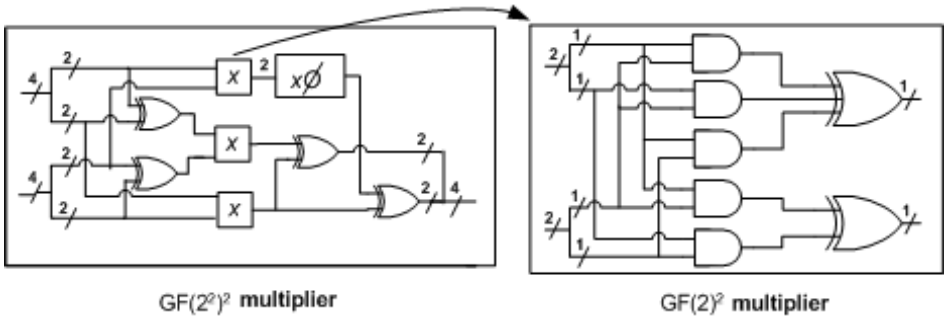


Figure IV.20. $GF(2^2)^2$ and $GF(2^2)$ multipliers.

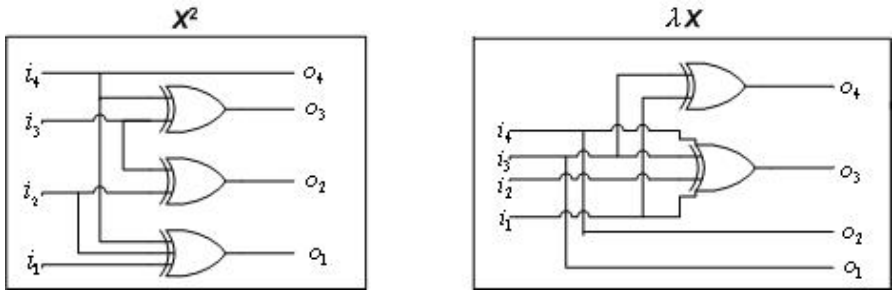


Figure IV.21. Gate level implementation for x^2 and λx.

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 increase in CLB slices for this design is due to the manipulation for MI instead of using BRAMs. The increase in design complexity causes the throughput to decrease as compared to first design. However it is a fully pipeline encryptor/decryptor core with relatively less throughput but it is a portable and a cost-effective solution.

IV.4.4 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 IV.3.2 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, all reported AES implementations are either encryptor cores or encryptor/decryptor cores and there is no known decryptor core. In the rest of this Section, the implementation aspects of all three cores are described.

Encryptor/decryptor core

This architecture reduces the large difference between the encryption/decryption time by exploiting the ideas explained in Section IV.3.2 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 IV.4.3. 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 IV.22. Two multiplexers are used to switch the data path for encryption and decryption.

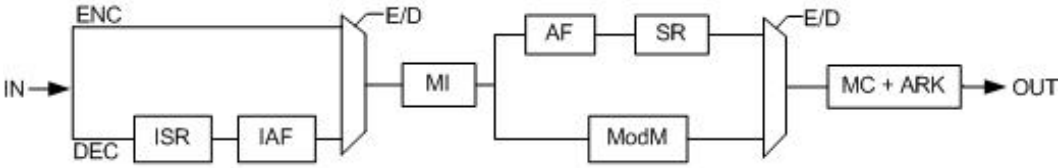


Figure IV.22. AES algorithm encryptor/decryptor implementation

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. It is useful design for applications where time factor dominates all other factors like cost, portability etc.

Encryptor core

It is a fully pipeline AES encryptor core. As it is 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 IV.23.

Figure IV.23. The data path for encryptor core implementation

For BS step, pre-computed values for S-Box are directly stored in the memories (BRAMs), therefore, AF transformation is embedded into BS. For symmetry purposes, 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 time delays for the circuit. The encryption process starts from the first clock cycle as the round-keys are generated in parallel as described in Section IV.4.1. The encrypted blocks appear at the output after 11 clock cycles after 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 separate 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 in the same Section. The critical path for this decryptor core is taken from Figure IV.22 and then modified for IBS implementations is shown in Figure IV.24.

The computations for IBS step are made by using look-up tables and pre-computed values of inverse S-Box are directly stored in the memories (BRAMs). The IAF step is embedded into IBS step for symmetric reasons which is obtained by just changing wires in FPGA implementations. The IMC step implementation is a major change in this design, which is implemented by performing a small modification ModM before

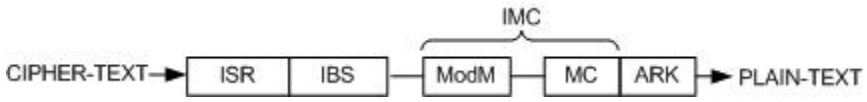


Figure IV.24. The data path for decryptor core implementation

MC step as discussed in Section IV.3.2. The MC and ARK steps are merged for the same reasons of targeting 4-input/1-output CLB structure.

The decryption process requires 11 cycles to generate the entire round Keys, then 11 cycles are consumed to fill the pipeline. Once the pipeline is filled the output plain-texts appear at the output after each consecutive clock cycle. On the other hand, 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).

By comparing encryptor and decryptor cores, there is no big difference in the number of CLB slices occupied by both encryptor and decryptor cores. Also the achieved throughput for both designs is very close. Both parameters indicate strong effect by modified IMC transformation which resulted reduced data path for decryption. The performance comparison of the separate implementations of encryptor and decryptor cores with combined encryptor/decryptor implementation is significant. It provides another option to the end-user to select a big chip for combined implementation or to choose two small chips for separate implementations of encryptor and decryptor cores to accomplish high gains in throughput.

IV.5 Performance comparison

IV.5.1 Previous work

Although the selection of new advanced encryption standard was finalized on October, 2000, still very few AES implementations are reported on FPGAs. Three main features are observed for the previous AES implementations on FPGAs.

1. **Algorithm's selection:** Not all of the AES architectures implemented the whole process: encryption, decryption and key schedule algorithms. Most of them implement the encryption part only. The key schedule algorithm is often ignored and it is supposed that keys are stored in the internal memory of FPGAs or can be provided through an external interface. The FPGA's implementations at [33, 30, 25] are encryptor cores and the key schedule algorithm is implemented for the design at [25] only. The AES cores at [52, 10, 11] implements both encryption and decryption with key schedule algorithm.

2. **Design's strategy:** This is an important factor that decides design's strategy as area/time tradeoffs. The reported AES cores adopted various implementation's strategies. Some of them are iterative looping (IL) [33], sub-pipline

(SP) [30], one-round implementation [25]. Few fully pipeline (PP) architectures have been also reported at [52, 10, 11].

3. **Selection of FPGA:** The high performance of AES cores is also subjected to the selection of FPGAs. High performance FPGAs can be efficiently used to achieve high gains in throughput. Most of the reported AES cores used Virtex series devices (XCV812, XCV1000, XCV3200). Those are single chip FPGA implementations. Some AES cores achieved ultra high throughput but at the cost of multi-chip FPGA architectures [10, 11].

IV.5.2 Results comparison

The results comparison for FPGA's implementations is not simple, since the three features mentioned strongly affect the algorithm performance. It would be a fair comparison if it were done under the same environment for all implementations. The performance of an encryptor core must be compared with the performances of the encryptor cores using the same FPGA, same design's strategies and same design specifications. Table IV.3 provides a quick comparison of the results with existing FPGA implementations. The comparison of encryptor/decryptor cores is made first followed by the comparison of encryptor cores with existing FPGA implementations. Finally, the comments are provided for our decryptor core implementation.

Our encryptor/decryptor cores (Sec. IV.4.3) are compared with the only known encryptor/decryptor core architecture at [52]. The first encryptor/decryptor core (Sec. IV.4.3) improves design at [52] both in speed and area. There is an increase of 15% in throughput at 15% less area utilization. The second encryptor/core (Sec.IV.4.3) is based on on-fly architecture scheme for BS/IBS computations in GF(2^4) and GF(2^2)^2 and does not occupy BRAMs. In fact, memory requirements for the first encryptor/decryptor core (Sec. IV.4.3) were overcomed by introducing this idea. The penalty paid was an increment in CLB slices however throughput still proves to be competitive. Our encryptor/decryptor core (Sec. IV.4.4) is another highly optimized core. The results are highly appreciable as it requires up to 27.23% higher throughput, 25.06% less CLB slices and 21.56% less BRAMs than the fastest AES single-chip FPGA implementation at [52]. That high performance 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. It should be noted that some highly efficient AES encryptor/decryptor cores are available at [10, 11]. Those cores are for commercial use and high throughput is achieved by targeting multi-chip FPGA architectures.

According current results, our encryptor core design (Sec. IV.4.2) can be compared with two existing cores at [33, 25], since they are encryptor cores using the same IL architecture. Those cores were implemented to compare performance results for AES candidates algorithms and are designed to achieve high timing performances by using some primary assumptions. The design at [33] does not implement key schedule and use external interfaces to load input data and to store output data,

the feedback paths are not observed. The design at [25] implements key schedule algorithm too and achieves high throughput by using 5673 CLB slices. Both implementations are nice for comparing AES candidates and not for practical uses. However, our encryptor core is optimized for both area/time parameters and includes a complete set-up for encryption process. The user-key is accepted and round-keys are generated. The results of each round are latched for next rounds and a final output appears at the output after 10 rounds. The design complexity therefore, increases which causes the decrease in throughput however our encryptor occupies 2744 CLB slices, less CLB slices for any encryptor core reported to-date. Similarly, our fully pipeline encryptor cores (Sec. IV.4.2 and IV.4.4) well compare with the similar design in [52]. Due to the optimization work for reducing design area, our fully pipeline architecture consumes only 2136 CLB slices plus 100 BRAMs. Throughput obtained for this architecture is 5.2 Gbits/s. The timing optimizations are pending and there exist strong chances to improve throughput in the range of 6 to 7 Gbits/s.

Finally, our decryptor core (Sec. IV.4.4) achieves a throughput of 4.9 Gbits/s by consuming 3216 CLB slices. The author does not know any reported decryptor core implementation on FPGAs for the purpose of comparison. Decryptor core implementation was considered important for our investigation work about AES to estimate performances for combined and separated implementations of the encryptor and decryptor cores. In addition, it is convenient to provide an option to the end-user for the selection of encryptor/decryptor core on a single chip or encryptor and decryptor cores on two small chips with high gains in throughput.

Table IV.3. Specifications of AES FPGA implementations.

	Core	Type	Device (XCV)	BRAMs	CLB(S) Slices	Throughput Mbits/s (T)	T/S
Gaj et al [33]	E	IL	1000		2902	331.5	0.11
Dandalis et al [25]	E	IL	1000		5673	353	0.06
Elbirt et al [30]	E	SP	1000		9004	1940	0.26
McLoone et al [52]	E	P	812E	100	2222	6956	3.1
McLoone et al [52]	E/D	P	3200E	102	7576	3239	0.43
Sec. IV.4.3 [84]	E/D	P	2600E	80	6676	3840	0.58
Sec. IV.4.3 [84]	E/D	P	2600E		13416	3136	0.24
Sec. IV.4.4 [75]	E/D	P	2600E	100	5677	4121	1.73
Sec. IV.4.2 [82]	E	IL	812E		2744	258.5	0.09
Sec. IV.4.2 [82]	E	P	812E	100	2136	5193	2.43
Sec. IV.4.4 [83]	E	P	812E	100	2136	5193	2.43
Sec. IV.4.4 [80]	D	P	812E	100	3216	4949	1.54

IV.6 Conclusions

A variety of different encryptor, decryptor and encryptor/decryptor AES cores are presented. The encryptor cores are implemented both in iterative and pipeline

modes. Some novel implementation techniques are presented for the implementations of encryptor/decryptor cores which include: composite field approach for BS/IBS, look-up table method for BS/IBS, and modified MC/IMC approach. All the architectures are highly optimized AES cores that tradeoff between time and area. For each architecture, we focus not only time performances but space is also a major concern. Three main factors were considered for implementing diverse AES cores.

- High performance : High performances require 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 and consequently low cost FPGAs.
- Portable architecture: A portable architecture can be migrated to most of the FPGAs 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 targets to the standard resources available in most FPGAs. A general methodology for achieving a portable architecture, in some cases, costs timing decrease in performance.

For AES encryptor cores, both iterative and fully pipeline architectures are implemented. The high timing performances are achieved by consuming the minimum space occupied by any AES core reported to-date. The iterative encryptor core design is implemented using VirtexE device XCV812 by occupying the standard resources of the FPGAs and therefore can be migrated to all other family of FPGA devices. The fully pipeline encryptor core is implemented on the same device (VirtexE device XCV812). That core is optimized for high timing performances by using dedicated memory resources (BRAMs); it is a fully pipeline architecture useful for the applications where time is crucial and other factors like portability and cost are of secondary concern. The same factors can be considered to AES encryptor/decryptor cores. The AES encryptor/decryptor cores based on the BS/IBS implementation using look-up table method are efficient architectures as compared to BS/IBS implementation using composite fields which is a portable and low cost solution. The AES encryptor/decryptor core based on the modified MC/IMC is a good example to achieve high performance by using both efficient design and algorithmic techniques. It is a single-chip FPGA implementation that exhibits high performance at low area consumption. In short, time/area tradeoffs are always present, however by using efficient techniques at both design and algorithm level, the gap between area and time can be significantly reduced. A fully pipeline decryptor core for FPGAs was presented and there is no other known decryptor core implementation reported in the literature. FPGA implementations presented in this Section are competitive or in some cases even outperforms existing FPGA implementations reported in the literature to-date.

Chapter V

Elliptic Curve Cryptography

In this chapter we present a generic parallel architecture for the computation of the scalar multiplication over binary fields for two elliptic curve forms: and Weierstrass non-singular form. The architecture was designed as general as possible trying to make no assumptions about the specific hardware platform to be used by the designers. The idea of using parallel strategies was considered in every design stage and implemented as much as our hardware resources allowed us to do it so. The design results reported in this work allow us to compute GF(2^{191}) elliptic curve scalar multiplication operations for the Hessian and the Weierstrass non-singular form in about 114.7μ Secs and 59.26μ Secs, respectively.

V.1 Introduction

Over the past 17 years, many mathematical evidences have consistently shown that Elliptic Curve Cryptography (ECC) offers more security by key length than any other major public key cryptosystem. The most important operation for elliptic curve cryptosystems is the so-called *Scalar multiplication* operation. Let n be a positive integer and P a point on an elliptic curve. Then the scalar multiple $Q = nP$ is the point resulting of adding $n - 1$ copies of P to itself. Scalar multiplication is the main building block used in all the three fundamental ECC primitives: *Key Generation*, *Signature* and *Verification* schemes.

The security of elliptic curve systems 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 F_q and two points Q and P that belong to the curve, where P has order r, find a positive scalar $n \in [0, r-1]$ such that the equation $Q = nP$ 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.

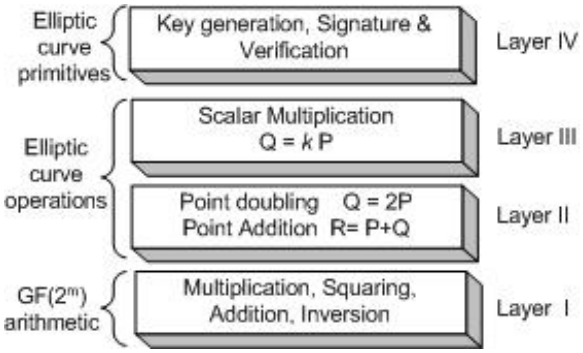


Figure V.1. Hierarchical Model for Elliptic Curve Cryptography

Several implementations have been reported so far [39, 92, 67, 91, 16, 89], and most of them utilize a four-layer hierarchical scheme such as the one depicted in Figure V.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 V.1 can be implemented using parallel strategies. Although parallel architectures offer an interesting potential for obtaining a high timing performance at the price of area, only in [91, 16] authors have explicitly attempted a parallel strategy to compute elliptic curve scalar multiplication. The contribution of this Chapter is the design of a generic parallel architecture especially tailored to obtain fast computation of the elliptic curves scalar multiplication operation. The architecture proposed here exploits the inherent parallelism of two elliptic curves forms defined over GF(2^m): The Hessian form and the Weierstrass non-supersingular form.

The rest of this Chapter is organized as follows. Section V.2 explains $GF(2^m)$ finite field arithmetic. In Section V.3 we briefly describe the Hessian and Weierstrass Forms of an elliptic curve together with their corresponding group laws. We also include some comments about the different design options that one can take in order to obtain a parallel version of the point addition and doubling operators. In Section V.4 we describe the generic parallel architecture for scalar multiplication that constitutes the main contribution of this chapter. Then in Section V.5 we give all the design details accomplished to implement the proposed architecture on an FPGA platform. Section V.6 includes a performance comparison of our design with other similar implementations previously reported. Finally, in Section V.7 some conclusions remarks as well as future work are drawn.

V.2 $GF(2^m)$ Finite field arithmetic

Arithmetic over $GF(2^m)$ has many important applications, in particular in the theory of error control coding and in cryptography [54, 57, 101]. Finite field's arithmetic

operations include addition, subtraction, multiplication, and division. 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.

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 $P(x)$, $C'(x) = A(x)B(x) \bmod P(x)$. In order to obtain $C'(x)$, we can first obtain the product polynomial $C(x)$ of degree at most $2m-2$, as

$$C(x) = A(x)B(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} b_i x^i) \tag{V.1}$$

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

$$C'(x) = C(x) \bmod P(x) \tag{V.2}$$

Notice that once the irreducible polynomial $P(x)$ has been selected, the reduction step can be accomplished by using XOR gates only.

Hardware implementation efficiency of finite field arithmetic is measured in terms of the associated space and time complexities. Space complexity is defined as the total amount of hardware resources needed to implement the circuit, whereas time complexity is the total delay of the circuit.

In the rest of this section different implementation aspects and several efficient methods to compute $GF(2^m)$ finite field arithmetic are discussed. In § V.2.1 and § V.2.2 we study the problem of how to compute Equation V.1 efficiently, considering two separate cases. First, in section § V.2.1 a variation of the classical Karatsuba-Ofman algorithm is analyzed as one of the most efficient techniques to find the polynomial product of Equation V.1. In subsection § V.2.2 we describe an efficient method to compute polynomial squaring in hardware, at a complexity cost of only $O(1)$. In § V.2.3 we describe in detail a highly efficient hardware implementation that carries on the reduction step of Equation V.2. Finally, in § V.2.4, it is explained of how the inversion in $GF(2^m)$ can be computed in less number of clock cycles.

V.2.1 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 [41, 95, 61, 104, 16]. 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 [70]. Discovered in 1962, it was the first algorithm to accomplish polynomial multiplication in under

$O(m^2)$ operations [15]. 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 [37] was presented a Karatsuba 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 [68] 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 [74]. 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

$$C = x^m A^H B^H + (A^H B^L + A^L B^H) x^{\frac{m}{2}} + A^L B^L. \tag{V.3}$$

Karatsuba-Ofman algorithm is based on the idea that the product of last equation can be equivalently written as,

$$\begin{aligned} C &= x^m A^H B^H + A^L B^L + \\ &\quad (A^H B^H + A^L B^L + (A^H + A^L)(B^L + B^H)) x^{\frac{m}{2}} \\ &= x^m C^H + C^L. \end{aligned} \tag{V.4}$$

Let us define

$$\begin{aligned} M_A &:= A^H + A^L; \\ M_B &:= B^L + B^H; \\ M &:= M_A M_B. \end{aligned} \tag{V.5}$$

Using Equation V.4, and taking into account that the polynomial product C has at most $2m-1$ coordinates, we can classify its coordinates as,

$$\begin{array}{rcl} C^H & = & [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m]; \\ C^L & = & [c_{m-1}, c_{m-2}, \ldots, c_1, c_0]. \end{array} \tag{V.6}$$

Although (V.4) seems to be more complicated than (V.3), it is easy to see that Equation (V.4) can be used to compute the product at a cost of four polynomial additions and three polynomial multiplications. In contrast, when using equation (V.3), 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 (V.4) is computationally simpler than the classic algorithm.

Input: Two elements $A, B \in GF(2^m)$ with $m = rn = 2^k n$, and
where A, B can be expressed as,
$A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m-1$ coordinates,
where $C = x^m C^H + C^L$.
Procedure Kmul2^k(C, A, B)
0. begin
1. if $(r == 1)$ then
2. $C = mul\_n(A, B)$;
3. return;
4. for i from 0 to $\frac{r}{2} - 1$ do
5. $M_{Ai} = A_i^L + A_i^H$;
6. $M_{Bi} = B_i^L + B_i^H$;
7. end
8. $mul2^k(C^L, A^L, B^L)$;
9. $mul2^k(M, M_A, M_B)$;
10. $mul2^k(C^H, A^H, B^H)$;
11. for i from 0 to $r-1$ do
12. $M_i = M_i + C_i^L + C_i^H$;
13. end
14. for i from 0 to $r-1$ do
15. $C_{\frac{r}{2}+i} = C_{\frac{r}{2}+i} + M_i$;
16. end
17. end

Figure V.2. $m = 2^k n$-bit Karatsuba-Ofman multiplier.

Karatsuba-Ofman's algorithm can be applied recursively to the three polynomial multiplications in (V.4). 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 (classic algorithm or other techniques).

The algorithm presented in Figure V.2 implements the Karatsuba-Ofman strategy for polynomial multiplication. By Combining the Karatsuba-Ofman algorithm with the classic algorithm, it can be shown [74] 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

$$\begin{aligned} \#\ \text{XORs} \leq &\quad \left(\tfrac{m}{n}\right)^{\log_2 3} (n^2 + 6n - 1) - 8m + 2\ ; \\ \#\ \text{ANDs} \leq &\quad 3^{\log_2 r} M_{and2^n} \ = \ \left(\tfrac{m}{n}\right)^{\log_2 3} n^2; \\ \text{Delay} \leq &\quad T_{AND} + T_X(\log_2 n + 4\log_2 r)\ . \end{aligned} \tag{V.7}$$

Where T_X and T_{AND} correspond to an XOR gate delay and an AND gate delay, respectively. Table V.1 shows the space and time complexities of Karatsuba-Ofman multipliers for the optimal case when m is a power of two. The values of m presented in Table V.1 are the first eight powers of two. Various factors including target devices and design techniques influence the space complexity in terms of number of CLBs. The space complexity provided in the last column of Table V.1 is based on our experimental results optimized for VirtexE devices only.

Table V.1. 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 CLBs)
1	1	1	1	0	T_A	–
2	1	2	4	1	$T_X + T_A$	–
4	1	4	16	9	$2T_X + T_A$	8
8	2	4	48	55	$6T_X + T_A$	32
16	4	4	144	225	$10T_X + T_A$	115
32	8	4	432	799	$14T_X + T_A$	368
64	16	4	1296	2649	$18T_X + T_A$	1171
128	32	4	3888	8455	$22T_X + T_A$	3379

To generalize the Karatsuba-Ofman algorithm of Figure V.2 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 $m = 2^k + d$. As a very first approach, we could pretend that both operands have 2^{k+1} coordinates

$$A = [\overbrace{\underbrace{0, \ldots, 0, 0}, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}}^{A^H}, \overbrace{a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0}^{A^L}];$$

$$A^H = [0, \ldots, 0, 0, a_{2^k+d-1}, \ldots, a_{2^k+1}, a_{2^k}];$$
$$A^L = [a_{2^k-1}, a_{2^k-2}, \ldots, a_2, a_1, a_0];$$

Figure V.3. Binary Karatsuba-Ofman strategy

each, where their respective $2^{k+1}-d$ most significant bits are all equal to zero. Figure V.3 shows how the subpolynomials A^H and A^L will be redefined according with this approach. If we partition the operands A and B as shown in Figure V.3, then, in order to compute their polynomial multiplication, we can use the regular Karatsuba-Ofman algorithm using $m = 2^{k+1}$. Although this approach is a valid one, 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 shown in Figure V.4, which has been adapted from the one presented in previous Figure V.2.

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 of Figure V.4 reverts to the specialized algorithm in Figure V.2 presented in the previous section. If that is not the case, the algorithm of Figure V.4 uses the constants k and d to prevent us to compute unnecessary arithmetic operations. In lines 6-9, the d least significant bits of M_A and M_B of equation (V.5) 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 Figure V.2 in lines 10 and 11. 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 12, 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. As a design example, consider the binary Karatsuba-Ofman multiplier shown in Figure V.5. 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

Input: Two elements $A, B \in GF(2^m)$ with m an arbitrary number, and where A, B can be expressed as $A = x^{\frac{m}{2}} A^H + A^L, B = x^{\frac{m}{2}} B^H + B^L$.
Output: A polynomial $C = AB$ with up to $2m - 1$ coordinates, where $C = x^m C^H + C^L$.
Procedure mulgen_m(C, A, B)
0. begin
1. $k = \lfloor \log_2 m \rfloor$;
2. $d = m - 2^k$;
3. if $(d == 0)$ then
4. $C = Kmul2^k(A, B)$;
5. return;
6. for i from 0 to $d - 1$ do
7. $M_{Ai} = A_i^L + A_i^H$;
8. $M_{Bi} = B_i^L + B_i^H$;
9. end
10. $mul2^k(C^L, A^L, B^L)$;
11. $mul2^k(M, M_A, M_B)$;
12. $mulgen\_d(C^H, A^H, B^H)$;
13. for i from 0 to $2^k - 2$ do
14. $M_i = M_i + C_i^L + C_i^H$;
15. end
16. for i from 0 to $2^k - 2$ do
17. $C_{k+i} = C_{k+i} + M_i$;
18. end
19. end

Figure V.4. m-bit binary Karatsuba-Ofman multiplier.

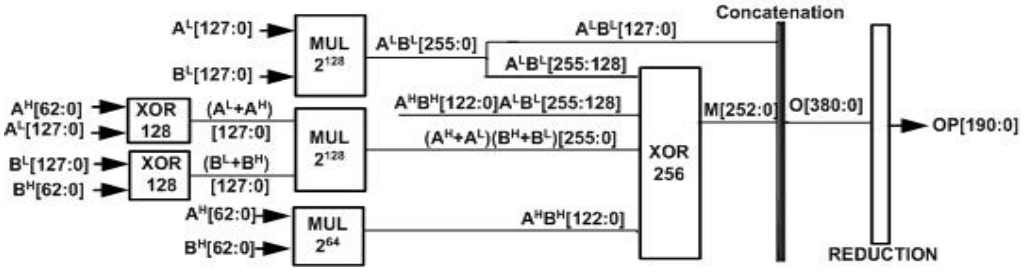


Figure V.5. Karatsuba Multiplier $GF(2^{191})$

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 V.4 can finish the computation in only two iterations, as shown in Figure V.5. By using the complexity Figures listed in Table V.1, we can estimate the space and time complexities of the generalized 191-bit binary Karatsuba-Ofman multiplier as,

$$\begin{aligned} \text{Number of CLBs} &= 2MUL_X(128) + MUL_X(64) + C \\ &= 2 \cdot 3379 + 1171 + C \\ &= 7929 + C \\ \text{Delay} &= MUL_{delay}(2^{\lfloor \log_2 m \rfloor}) + O \\ &= MUL_{delay}(2^{\lfloor \log_2 191 \rfloor}) + O \\ &= MUL_{delay}(2^7) + O \end{aligned} \tag{V.8}$$

Where C and O represent the overhead in space and time, respectively, associated with the extra circuitry shown in Figure V.5. 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.

V.2.2 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

$$C(x) = A(x)A(x) = A^2(x) = (\sum_{i=0}^{m-1} a_i x^i)(\sum_{i=0}^{m-1} a_i x^i) = \sum_{i=0}^{m-1} a_i x^{2i}. \tag{V.9}$$

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 V.6 simply interleaves a zero value between each one of the original bits of A yielding the required squaring computation. The implementation shown in Figure V.6 has a computational complexity O(1), and hence its cost can be basically neglected.

V.2.3 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 (V.1), by using any one of the methods described in the previous two subsections, we want to obtain the modular product C' of

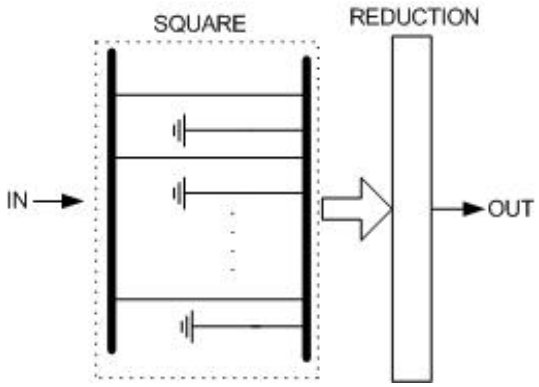


Figure V.6. Squaring Circuit

Equation (V.2). Recall that the polynomial product C and the modular product C', have $2m-1$ and m, coordinates, respectively, i.e.,

$$\begin{array}{rcl} C & = & [c_{2m-2}, c_{2m-3}, \ldots, c_{m+1}, c_m, \ldots, c_1, c_0]; \\ C' & = & [c'_{m-1}, c'_{m-2}, \ldots, c'_1, c'_0]. \end{array} \tag{V.10}$$

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. 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 (V.1), we use the property $P(\alpha) = 0$, and write

$$\begin{array}{rcl} \alpha^m & = & 1 + \alpha^n \ ; \\ \alpha^{m+1} & = & \alpha + \alpha^{n+1} \ ; \\ & \vdots & \\ \alpha^{2m-3} & = & \alpha^{m-3} + \alpha^{m+n-3} \ ; \\ \alpha^{2m-2} & = & \alpha^{m-2} + \alpha^{m+n-2} \ . \end{array} \tag{V.11}$$

The above $m-1$ set of identities suggests a method to obtain the m-coordinates of the modular product C' of Equation (V.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 (V.11). 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,

$$
\begin{array}{lcllll}
c'_0 & = & c_0 & + & c_m \ ; & \\
c'_1 & = & c_1 & + & c_{m+1} \ ; & \\
 & \vdots & & & & \\
c'_{n-1} & = & c_{n-1} & + & c_{m+n-1} \ ; & \\
c'_n & = & c_n & + & c_{m+n} & + \ c_m \ ; \\
c'_{n+1} & = & c_{n+1} & + & c_{m+n+1} & + \ c_{m+1} \ ; \\
 & \vdots & & & & \\
c'_{m-2} & = & c_{m-2} & + & c_{2m-2} & + \ c_{2m-n-2} \ ; \\
c'_{m-1} & = & c_{m-1} & & & + \ c_{2m-n-1} \ ; \\
c'_m & = & c_m & & & + \ c_{2m-n} \ ; \\
 & \vdots & & & & \\
c'_{m+n-3} & = & c_{m+n-3} & & & + \ c_{2m-3} \ ; \\
c'_{m+n-2} & = & c_{m+n-2} & & & + \ c_{2m-2} \ .
\end{array}
\tag{V.12}
$$

Notice that in the reduction process of (V.12), 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 (V.12) in the range $[c'_n, c'_{m+n-2}]$. Notice also that there is an overlap

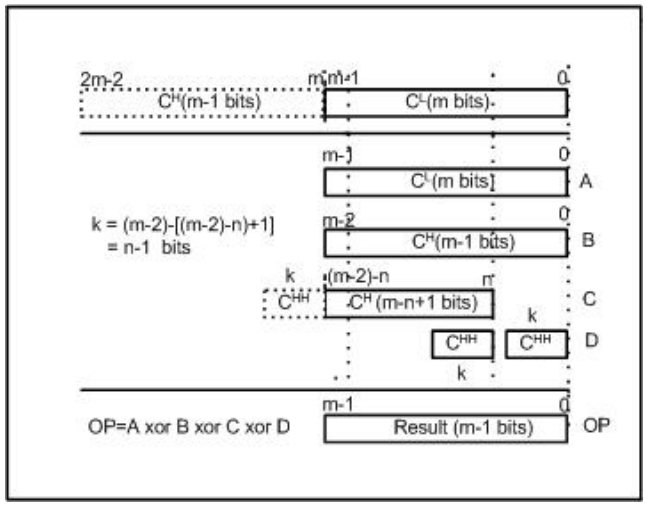


Figure V.7. Reduction Diagram

in the range $[c'_n, c'_{m-2}]$, where both the constant and the middle coefficients of $P(x)$ affect the partially reduced coordinates.

We say that the coefficients in (V.12) have been partially reduced because in general, if $n > 1$, we still need to reduce the $n-2$ most significant reduced coordinates

of (V.12). However, this same idea can be used repeatedly until the $m-1$ modular coordinates of (V.10) are obtained. Each time that this strategy is applied we reduce $m-n$ coordinates.

Figure V.7 shows how to implement on reconfigurable hardware the reduction strategy just outlined. As it was mentioned before, the reduction step involves XOR and overlap operations only.

Although the strategy shown in Figure V.7 works for arbitrary irreducible trinomials, for the purposes of this research work we utilized a fixed irreducible generating trinomial, namely, $P(x) = x^{191} + x^9 + 1$.

V.2.4 Inversion

The Euclidean algorithm required $O(n^2)$ bit operations to perform inversion. Fermat 's Little Theorem (FLT) establishes that for any nonzero element $\alpha \in GF(2^m)$, the identity $\alpha^{-1} \equiv \alpha^{2^m-2}$ holds. Therefore, multiplicative inversion can be performed by computing,

$$\alpha^{2^m-2} \;=\; \alpha^{2^1} \times \alpha^{2^1} \times \cdots \alpha^{2^{m-1}} \tag{V.13}$$

A straightforward implementation of Eq. V.13 can be carried out using the binary exponentiation method, which requires $m-1$ field squarings and $m-2$ field multiplications. The Itoh-Tsujii Multiplicative Inverse Algorithm (ITMIA) algorithm [44] on the other hand, reduces the required number of multiplications to $k+hw(m-1)-2$, where $k \;=\; \lfloor log_2(m-1) \rfloor$ and $hw(m-1)$ are the number of bits and the Hamming weight of the binary representation of $m-1$, respectively. This remarkably saving on the number of multiplications is based on the observation that since $2^m - 2 = (2^{m-1}-1)\cdot 2$, then the identity from Fermat's little theorem can be rewritten as $\alpha^{-1} \equiv \alpha^{2^m-2} \equiv \alpha^{(2^{m-1}-1)^2}$. Then ITMIA computes the field element $(2^{m-1}-1)$ using a recursive re-arrangement of the finite field operations.

Using the ITMIA algorithm one can find an addition chain for an arbitrary m. We used this algorithm to obtain an addition chain for m = 191 (our design example). In this case $e = m-1 = 190 = 2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$. Therefore,

$$\begin{aligned} U = 1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow \\ 160 \rightarrow 176 \rightarrow 184 \rightarrow 188 \rightarrow 190 \end{aligned} \tag{V.14}$$

The addition chain in Eq. V.14 can be accomplished using 12 multiplications and $m-1$ squarings in accordance to the algorithm. However it can be seen that various addition chains of shortest length for the same m exist. Let l be the shortest length of any valid addition chain for a given positive integer $e \;=\; m-1$. Then the problem of finding an addition chain for e with length l is an **NP**-hard problem [58]. Let us first formally define an addition chain in order to describe a methodology to achieve field multiplicative inverses.

Addition Chains

An *addition chain* U for a positive integer $e = m-1$ of length t is a sequence of positive integers $U = \{u_0, u_1, \cdots, u_t\}$, and an associated sequence of r pairs $V = \{(v_1, v_2 \cdots, v_t\}$ with $v_i = (i_1, i_2), 0 \leq i_2 \leq i_1 < i$, such that:

- $u_0 = 1$ and $u_t = m-1$;
- for each $u_i, 1 \leq i \leq t, u_i = u_{i_1} + u_{i_2}$.

Consider the same design example for $m = 191$, $e = m-1 = 190-1$. Then, one of possible addition chain with length $t = 10$ is,

$$U = 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 80 \rightarrow 160 \rightarrow 180 \rightarrow 190 \quad \text{(V.15)}$$

Each term of this chain u_i is obtained either by doubling the immediate preceding term $u_{i-1} + u_{i-1}$ or by adding any two previous terms $u_j + u_k$. That provides us two indices i_1 and i_2. In the former case, the same term was doubled so $i_1 = i_2 = i-1$ while for later case two different terms u_j and u_k were added provided $i_1 = j$ and $i_2 = k$.

Input: An element $\alpha \in GF(2^m)$, an addition chain U of length t for $m-1$ and its associated sequence V.
Output: $\alpha^{-1} \in GF(2^m)$
Procedure MultiplicativeInversion(α, U)
1. $\beta_0 = \alpha$
2. for i from 1 to t do:
3. $\beta_i = (\beta_{i_1})^{2^{u_{i_2}}} \cdot \beta_{i_2}$
4. return $(\beta_t{}^2)$;

Figure V.8. An Algorithm for multiplicative inversion using addition chains

Consider the algorithm shown in Fig. V.8. That algorithm iteratively computes the β_i coefficients in the exact order stipulated by the addition chain U. Indeed, starting from $\beta_0 = (\alpha)^{2^{u_0}-1} = (\alpha)^{2^{u_0}-1} = \alpha^{2^1-1}$, the algorithm computes the other t β_i coefficients. In the final iteration, after having computed the coefficient $\beta_t = (\alpha)^{2^{m-1}-1}$, the algorithm returns the required multiplicative inversion by performing a regular field squaring, namely, $\beta_t^2 = (\alpha^{2^m-2}) = \alpha^{-1}$.

Let us assume that the binary extension field $GF(2^{191})$ has been generated with the trinomial $P(x) = x^{191} + x^9 + 1$, irreducible over $GF(2)$. Let $\alpha \in GF(2^{191})$ be an arbitrary nonzero field element. Then, using the addition chain of Example V.15, the algorithm of Fig. V.8 will compute the sequence of β coefficients as shown in

Table V.2. Algorithm of Fig. V.8: β_i Coefficient Generation

i	u_i	i_1	i_2	u_{i2}	rule	$\beta_{i_1}^{2^{u_{i_2}}} \cdot \beta_{i_2}$	$\beta_i = \alpha^{2_i^u - 1}$
0	1	0	0	1	–	–	$\beta_0 = \alpha^{2^1-1}$
1	2	0	0	1	$u_0 + u_0$	$\beta_0^{2^1} \cdot \beta_0$	$\beta_1 = \alpha^{2^2-1}$
2	3	1	0	1	$u_1 + u_0$	$\beta_1^{2^1} \cdot \beta_0$	$\beta_2 = \alpha^{2^3-1}$
3	5	2	1	2	$u_2 + u_1$	$\beta_2^{2^2} \cdot \beta_1$	$\beta_3 = \alpha^{2^5-1}$
4	10	3	3	5	$u_3 + u_3$	$\beta_3^{2^5} \cdot \beta_3$	$\beta_4 = \alpha^{2^{10}-1}$
5	20	4	4	10	$u_4 + u_4$	$\beta_4^{2^{10}} \cdot \beta_4$	$\beta_5 = \alpha^{2^{20}-1}$
6	40	5	5	20	$u_5 + u_5$	$\beta_5^{2^{20}} \cdot \beta_5$	$\beta_6 = \alpha^{2^{40}-1}$
7	80	6	6	40	$u_6 + u_6$	$\beta_6^{2^{40}} \cdot \beta_6$	$\beta_7 = \alpha^{2^{80}-1}$
8	160	7	7	80	$u_7 + u_7$	$\beta_7^{2^{80}} \cdot \beta_7$	$\beta_8 = \alpha^{2^{160}-1}$
9	180	8	5	20	$u_8 + u_5$	$\beta_8^{2^{20}} \cdot \beta_5$	$\beta_9 = \alpha^{2^{180}-1}$
10	190	9	4	10	$u_9 + u_4$	$\beta_9^{2^{10}} \cdot \beta_4$	$\beta_{10} = \alpha^{2^{190}-1}$

Table V.2.4. Once again, notice that after having computed the coefficient β_{10} the only remaining step is to obtain α^{-1} as, $\alpha^{-1} = \beta_{10}^2$

Let us now assess the computational complexity of the algorithm shown in Fig. V.8. 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-esime iteration is $u_i - 1$ [98]. 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. V.8 can find the inverse of any nonzero element of the field using exactly,

$$\begin{aligned} \#Multiplications &= t; \\ \#Squarings &= m - 1. \end{aligned} \tag{V.16}$$

Squarer and Multiplier are the two main building blocks for performing inversion in GF(2^m). We have already discussed the structure of our multiplier in Section V.2.1. Squaring in GF(2^m) is a simple operation however, as it was stated in Eq. V.16, it devours $m-1$ clock cycles performing $m-1$ squarings for m-bit inversion. However, much less field multiplications are needed for computing the multiplicative

inversion, which is valuable for the design since field multiplication in GF(2^m) is a costly and extensive time consuming operation. Figure V.9.a shows our strategy for

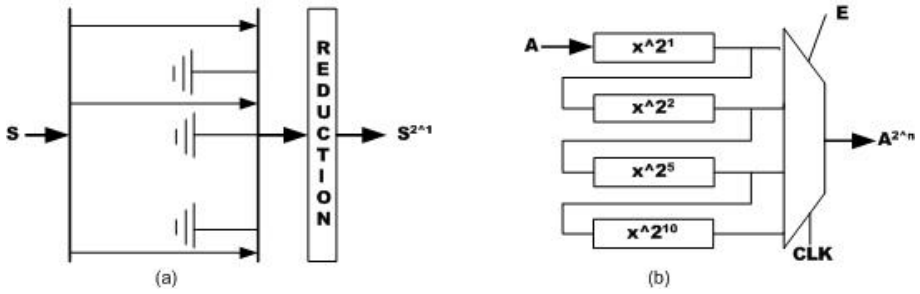


Figure V.9. Squarer GF(2^{193}) (a) for x^{2^1} (b) for x^{2^n} implementation

implementing field squaring trying to use as few clock cycles as possible. Polynomial squaring is free of cost and is obtained by connecting each alternate even bit to zero (ground). That operation is followed by a second step where reduction is performed by using few XOR gates. Figure V.9.b shows the $GF(2^{191})$ field squarer implementation used in this work. Referring to the addition chain described in Table V.2.4, frequent squaring operations in $GF(2^{191})$ are ${\beta_i}^{2^1}$ (2 times), $\beta_i 2^2$ (1 time), $\beta_i 2^5$ (1 time), and $\beta_i 2^{10}$ (10 times).

That is why we preferred to cascade 10 field squarer blocks back to back and then by the appropriate use of multiplexers obtain the corresponding outputs after 1, 2, 5, and 10 squarer blocks as shown in Figure V.9.b. As an example, the $x^{2^{20}}$ field operation can be performed in just two clock cycles by taking the output after the last squarer block (10 squarers) in the first clock cycle and then after a second clock cycle we will get the required 20 field squarings.

V.3 Elliptic curve scalar multiplication

In this section we discuss the Hessian form of elliptic curves and its group law followed by Montgomery point multiplication algorithm. The group law for Montgomery point multiplication is discussed both in affine and projective coordinates. The discussions are also made for the conversion from projective to affine coordinates.

V.3.1 Hessian form

Let $P(x)$ be a degree-m polynomial, irreducible over GF(2). Then $P(x)$ generates the finite field $F_q = GF(2^m)$ of characteristic two. A Hessian elliptic curve $E(F_q)$ 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,

$$x^3 + y^3 + z^3 = Dxyz \tag{V.17}$$

Together with the point at infinity denoted by $\mathcal{O}$ and given by $(1, 0, -1)$.

The original form of the group law on curves in Hessian form belongs to Cauchy and was later simplified by Sylvester-Desboves [23].

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. V.22. Then we define $-P = (y_1, x_1, z_1)$ and $P + Q = (x_3, y_3, z_3)$ where,

$$\begin{array}{rcl} x_3 & = & {y_1}^2 x_2 z_2 - {y_2}^2 x_1 z_1 \\ y_3 & = & {x_1}^2 y_2 z_2 - {x_2}^2 y_1 z_1 \\ z_3 & = & {z_1}^2 y_2 x_2 - {z_2}^2 y_1 x_1 \end{array} \tag{V.18}$$

Provided that $P \neq Q$. The addition formulae of Eq. (V.18) might be parallelized using 12 field multiplications as follows [91],

$$\begin{array}{lll} \lambda_1 = y_1 x_2 & \lambda_2 = x_1 y_2 & \lambda_3 = x_1 z_2 \\ \lambda_4 = z_1 x_2 & \lambda_5 = z_1 y_2 & \lambda_6 = z_2 y_1 \\ s_1 = \lambda_1 \lambda_6 & s_2 = \lambda_2 \lambda_3 & s_3 = \lambda_5 \lambda_4 \\ t_1 = \lambda_2 \lambda_5 & t_2 = \lambda_1 \lambda_4 & t_3 = \lambda_6 \lambda_3 \\ x_3 = s_1 - t_1 & y_3 = s_2 - t_2 & z_3 = s_3 - t_3 \end{array} \tag{V.19}$$

Whereas the formulae for point doubling are giving by

$$\begin{array}{rcl} x_3 & = & y_1 \left({z_1}^3 - {x_1}^3 \right); \\ y_3 & = & x_1 \left({y_1}^3 - {z_1}^3 \right); \\ z_3 & = & z_1 \left({x_1}^3 - {y_1}^3 \right). \end{array} \tag{V.20}$$

Where $2P = (x_3, y_3, z_3)$. The doubling formulae of Eq. (V.20) can be also parallelized requiring 6 field multiplications plus three field squarings for their computation. The resulting arrangement can be rewritten as [91],

$$\begin{array}{lll} \lambda_1 = {x_1}^2 & \lambda_2 = {y_1}^2 & \lambda_3 = {z_1}^2; \\ \lambda_4 = x_1 \lambda_1 & \lambda_5 = y_1 \lambda_2 & \lambda_6 = z_1 \lambda_3; \\ \lambda_7 = \lambda_5 - \lambda_6 & \lambda_8 = \lambda_6 - \lambda_4 & \lambda_9 = \lambda_4 - \lambda_5; \\ x_2 = y_1 \lambda_8 & y_2 = x_1 \lambda_7 & z_2 = z_1 \lambda_9; \end{array} \tag{V.21}$$

By implementing Eqs. (V.19) and (V.21), one can obtain the two building blocks needed for the implementation of the second layer shown in Figure V.1. Hence, provided that those two blocks are available, one can compute the third layer of Figure V.1 by using the well-known doubling and add algorithm of Figure V.10. That sequential algorithm needs an average of $\frac{m}{2}$ point additions plus m point doublings in order to complete one scalar multiplication computation.

Alternatively, we can use the algorithm of Figure V.11 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 V.11 is implemented in parallel, one needs an average of just $\frac{m}{2}$ point additions plus $\frac{m}{2}$ point doublings in order to complete one scalar multiplication computation.

In Subsection V.3.3 we discuss how to obtain an efficient parallel-sequential implementation of the second and third layers of the model of Figure V.1.

Input: $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$,
$P(x, y, z) \in E(F_{2^m})$
Output: $Q = kP$

1. Set $Q = P$
2. For i from $n - 2$ downto 0 do
3. $\quad Q = 2.Q$ (point doubling)
4. $\quad$ if $(k_i = 1)$ then
5. $\quad\quad Q = Q + P$ (point addition)
6. $\quad$ end if;
7. end for;

Figure V.10. Doubling & Add algorithm for Scalar Multiplication: MSB-First

Input: $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$,
$P(x, y, z) \in E(F_{2^m})$
Output: $Q = kP$

1. Set $Q = 1$; $R = P$
2. For i from 0 to $n - 1$ do
3. $\quad$ if $(k_i = 1)$ then
4. $\quad\quad Q = Q + R$ (point addition)
5. $\quad R = 2.R$ (point doubling)
6. $\quad$ end if;
7. end for;

Figure V.11. Doubling & Add algorithm for Scalar Multiplication: LSB-First

V.3.2 Weierstrass Non-Singular form and Montgomery Point Multiplication Algorithm

Let $P(x)$ be a degree-m polynomial, irreducible over $GF(2)$. Then $P(x)$ generates the finite field $F_q = GF(2^m)$ of characteristic two. A non-supersingular elliptic curve $E(F_q)$ is defined to be the set of points $(x, y) \in GF(2^m) \times GF(2^m)$ that satisfy the affine equation,

$$y^2 + xy \;=\; x^3 + ax^2 + b, \tag{V.22}$$

Where a and $b \in F_q, b \neq 0$, together with the point at infinity denoted by O. The elliptic curve group law in affine coordinates is given by:

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve V.22 then $-P = (x_1, x_1 + y_1)$. For all P on the curve $P + O = O + P = P$. If $Q \neq -P$,

then $P + Q = (x_3, y_3)$, where

$$x_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})^2 + \frac{y_1+y_2}{x_1+x_2} + x_1 + x_2 + a & P \neq Q \\ x_1^2 + \frac{b}{x_1^2} & \text{P = Q} \end{cases} \tag{V.23}$$

$$y_3 = \begin{cases} (\frac{y_1+y_2}{x_1+x_2})(x_1 + x_3) + x_3 + y_1 & P \neq Q \\ x_1^2 + (x_1 + \frac{y_1}{x_1})x_3 + x_3 & \text{P = Q} \end{cases} \tag{V.24}$$

It can be seen from Eqns. V.23 and V.24 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 neglecting the costs of field additions and squarings. An important observation first made by Montgomery is that the x-coordinate of $2P$ does not involve the y-coordinate of P.

I. 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. 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:

$$Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \tag{V.25}$$

II. Montgomery Group Law

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two points that belong to the curve of Equation V.22. 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 [39],

$$x_3 = x_4 + \frac{x_1}{x_1 + x_2} + \left(\frac{x_1}{x_1 + x_2}\right)^2; \tag{V.26}$$

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 $2P = (X_{2P}, Y_{2P}, Z_{2P})$ is converted to projective coordinate representation, it becomes [50],

$$\begin{aligned} x_{2P} &= X^4 + b \cdot Z^4; \\ z_{2P} &= X^2 \cdot Z^2; \end{aligned} \tag{V.27}$$

The computation of Eq. V.27 requires one general multiplication, one multiplication by the constant b, five squarings and one addition. Fig. V.12 is the sequence of instructions needed to compute a single point doubling operation $Mdouble(X_1, Z_1)$ efficiently.

Input: $P = (X_1, -, Z_1) \in E(F_{2^m}), c$ such that $c^2 = b$
Output: $P = 2 \cdot P$
Procedure: Mdouble(X_1, Z_1)

1. $T = X_1^2$
2. $M = c \cdot Z_1^2$
3. $Z_1 = T \cdot Z_1^2$
4. $M = M^2$
5. $T = T^2$
6. $X_1 = T + M$

Figure V.12. Montgomery point doubling

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:

$$\begin{array}{rcl} Z_3 & = & (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2; \\ X_3 & = & x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1); \end{array} \tag{V.28}$$

Input: $P = (X_1, -, Z_1), Q = (X_2, -, Z_2) \in E(F_{2^m})$
Output: $P = P + Q$
Procedure: Madd(X_1, Z_1, X_2, Z_2)

1. $M = (X_1 \cdot Z_2) + (Z_1 \cdot X_2)$
2. $Z_1 = M^2$
3. $N = (X_1 \cdot Z_2) \cdot (Z_1 \cdot X_2)$
4. $M = x \cdot Z_3$
5. $X_1 = M + N$

Figure V.13. Montgomery point addition

The required field operations for point addition of Eq. V.28 are three general multiplications, one multiplication by x, one squaring and two additions. This operation can be efficiently implemented as shown in Fig. V.13.

III. Montgomery Point Multiplication

A method based on the formulas for doubling (from Eq. V.27) and for addition (from Eq. V.28) is shown in Fig. V.14 [50]. Notice that steps 2.2 and 2.3 are formulae for point doubling (Mdouble) and point addition (Madd) from Figs. V.12 and V.13 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) (step 2.2) 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 2.3).

The approximate running time of the algorithm shown in Fig. V.14 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 V.3.3, we discuss how to obtain an efficient parallel implementation of the above algorithm quite especially for step 2.

Input: $k = (k_{n-1}, k_{n-2}....., k_1, k_0)_2$ with $k_{n-1} = 1$,
$P(x, y) \in E(F_{2^m})$
Output: $Q = kP$

Procedure: MontPointMult(P, k)
1. Set $X_1 \leftarrow x$, $Z_1 \leftarrow 1$, $X_2 \leftarrow x^4 + b$, $Z_2 \leftarrow x^2$
2. For i from $n-2$ downto 0 do
   - 2.1 if $(k_i = 1)$ then
     $Madd(X_1, Z_1, X_2, Z_2)$;
     $Mdouble(X_2, Z_2)$;
   - 2.2 else
     $Madd(X_2, Z_2, X_1, Z_1)$;
     $Mdouble(X_1, Z_1)$;
3. $x_3 \leftarrow X_1/Z_1$
4. $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2)+$
   $(x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$
5. Return (x_3, y_3)

Figure V.14. Montgomery point multiplication

IV. 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. V.14, 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: $y_3 \leftarrow (x + X_1/Z_1)[(X_1 + xZ_1)(X_2 + xZ_2) + (x^2 + y)(Z_1Z_2)](xZ_1Z_2)^{-1} + y$. 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 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. V.15.

Input: $P = (X_1, Z_1)$, $Q = (X_2, Z_2) \in E(F_{2^m})$
$P(x, y) \in E(F_{2^m})$
Output:(x_3, y_3) affine coordinates
Procedure: $SPtoAffine(X_1, Z_1, X_2, Z_2)$

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. $\lambda_9 = \lambda_1 \times \lambda_8$
11. $\lambda_{10} = \lambda_7 + \lambda_9$
12. $\lambda_{11} = x \times \lambda_1$
13. $\lambda_{12} = inverse(\lambda_{11})$
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. $y_3 = \lambda_{16} + y$

Figure V.15. Standard projective to affine coordinate

The coordinate conversion process makes use of 10 multiplications and only 1 inversion ignoring addition and squaring operations.

The algorithm in Fig. V.15 includes one inversion operation which can be performed using Extended Euclidean Algorithm or Fermat's Little Theorem (FLT).

V.3.3 Parallel strategies for scalar point multiplication

As it was mentioned in the introduction Section, parallel implementations of the three underlying layers depicted in Figure V.1 constitutes the main interest of this chapter. We briefly described how to do so in the case of the first layer in Subsection §V.2. 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. Several options to do so are shown in Table V.3.

Table V.3. $GF(2^m)$ Elliptic Curve Point Multiplication Computational Costs

Strategy		Req. No.	EC Operation Cost		T. N0. of	EC Operation Cost		T. No. of
			Hessian form			Montgomery Algorithm		
2nd Layer	3rd Layer	of Field Mults.	Doubling	Addition	Field Mults.	Doubling	Addition	Field Mults.
S	S	1	$6M$	$12M$	$18mM$	$2M$	$4M$	$6mM$
S	P	2	$6M$	$12M$	$12mM$	$2M$	$4M$	$4mM$
P	S	2	$3M$	$6M$	$9mM$	$1M$	$2M$	$3mM$
P	P	4	$6M$	$6M$	$6mM$	M	$2M$	$2mM$

Table V.3 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 V.3 are normalized with respect to the required number of field multiplication operations (since the computation time of squaring operations can be neglected in arithmetic over GF(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 V.3. 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 V.3, the estimated computational cost of our elliptic curve Point multiplication implementation will be of $9m$ 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.

V.4 Generic architecture for scalar point multiplication

Figure V.16 shows a generic structure for parallel 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 V.3, assuming the availability of two GF(2^m) multiplier blocks.

As shown in Figure V.16, the basic organization for the computation of elliptic curve scalar multiplication is comprised of four classes of blocks: GF(2^m) multipliers, Combinational logic blocks and/or finite field arithmetic (i.e. squaring, etc.), Blocks

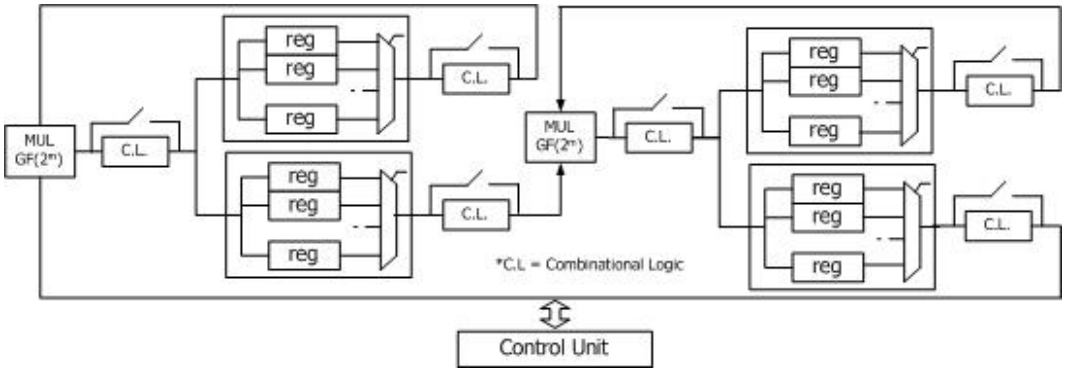


Figure V.16. Basic organization of elliptic curve scalar implementation

for intermediate results storage and selection (i.e. registers, multiplexers, etc.), and a Control unit (CU). The CU is the default part of every hardware circuit to control the flow of data between different stages of the design. Registers are provided to save the intermediate results. The results are further multiplexed to provide correct operands to the multipliers for next stages. The CL blocks perform pre or post computations at the multipliers to obtain final results within pre-defined clock cycles.

Figure V.16 presents a generic architecture that can be used for elliptic curve scalar multiplication with minor modifications on different hardware platforms like VLSI and FPGAs. Although the selection of the components is totally platform dependent, the selection of some special devices might be helpful to reduce design complexity. For example, using read/write memories (RAMs) instead of several registers and multiplexers might provide more design modularity and efficiency. On the other hand, one cannot ignore the fact that the most costly operation in kP computations by far is finite field multiplication over GF(2^m). Therefore, efficient GF(2^m) field multiplier blocks become indispensable in order to obtain fast and low-area implementations of elliptic curve scalar multiplication.

V.5 Implementing scalar multiplication on reconfigurable hardware

Figure V.17 proposes a parallel structure for implementing the point multiplication algorithm discussed in Section V.3. It is a generic FPGA architecture based on the parallel-sequential approach for kP computations discussed before. To implement the memory blocks of Figure V.16, fast access FPGA's read/write memories Block-RAMs (BRAMs) are used. BRAMs are built-in memory blocks available in modern FPGAs. A dual port BRAM can be configured as a two single port BRAMs with independent data access. This special feature allows the saving of a considerable number of multiplexer operations as the required data is accessible independently from any of the two available input ports. Hence, two similar BRAMs blocks (each one of 12 BRAMs) provide four operands to the two multiplier blocks simultane-

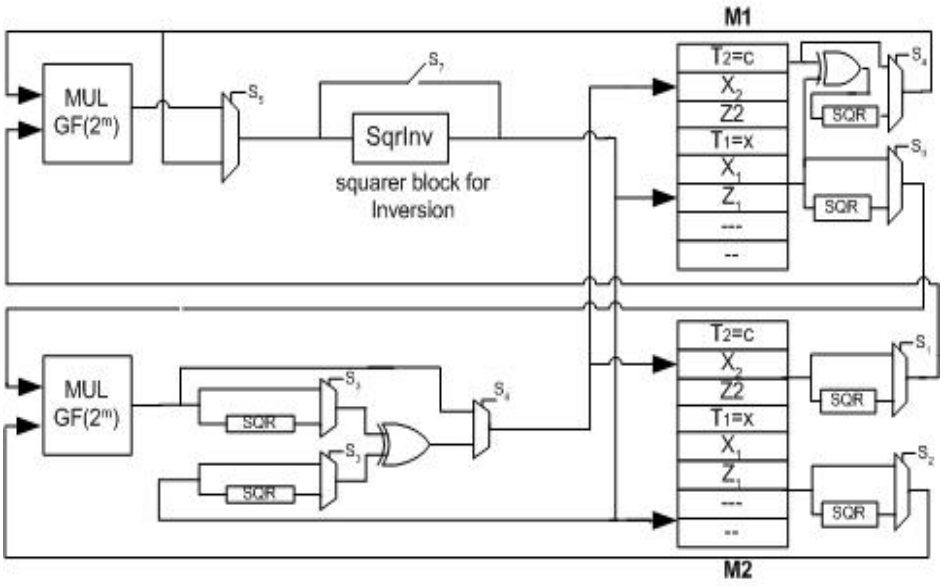


Figure V.17. Implementation of Scalar Multiplication on FPGA platforms

ously. Since each BRAM contains 4k memory cells, two BRAM blocks are sufficient. The combination of 12 BRAMs however provide an access to 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 do apply pre and post computations (squaring, XOR, etc.) on the inputs of the multipliers when they are required.

An inversion operation is performed once at the end for the conversion of SP to affine coordinates (in case of Montgomery point multiplication using Standard Projective coordinates). That operation uses building blocks for multiplication and squaring in $GF(2^m)$. One of the multiplier block MUL $GF(2^m)$ is used for the multiplication. A squarer block 'SqrInv' is especially added for the inversion only. As it was earlier explained that $m-1$ squarings are performed to complete one inversion in $GF(2^m)$. It takes $m-1$ clock cycles by using a single squarer block. However several squarer blocks can be cascaded to perform more than one squaring operation in the same clock cycle. That would be a useful approach for performing all squaring operations using a few clock cycles as was explained in Section V.2.4.

In the next Subsection we discuss how the architecture of Figure V.16 can be used to compute Hessian scalar multiplication.

V.5.1 Scalar multiplication in Hessian form

According to Eq. (V.19) of Section V.3.1 we know that addition of two points in Hessian form consists of 12 multiplications, 3 squarings and 3 addition operations.

Implementing squaring over GF(2^m) is a trivial operation, so we can neglect it. Using the parallel architecture proposed in Figure V.17, point addition can be performed in 6 clock cycles using two GF(2^{191}) multiplier blocks. For computing Hessian point addition according to Eq. (V.19), the sequence of the field multiplications must be followed as shown in Table V.4. In the architecture of Figure V.17, $M1$ and $M2$ are two memory (BRAMs) blocks, each one comprising of two independent ports $PT1$ and $PT2$. It should be noticed that the inputs/outputs of the multipliers are different from those read/write values at the memory blocks. It is due to pre or post computations required during the next clock cycle. Table V.4 lists values for the multiplications during read cycle and after the multiplications during write cycle.

Table V.4. 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. (V.21) consists of 6 multiplications, 3 squarings and 3 additions. Table V.5 describes the algorithm flow implemented using the same architecture (Figure V.17).

Table V.5. 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 V.10, 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 our case, m = 191 bits, the total number of field multiplications required by the algorithm are 1146. Let T be the allowed clock period then $1146 \times T$ is the total time to complete elliptic curve scalar multiplication.

V.5.2 Montgomery point multiplication

Referring to the algorithm of Figure V.14, each bit of vector k is tested from left to right (in descending order).

For each test bit (zero or one), both point addition (Madd) and point doubling (Mdouble) operations are performed. However, order of the arguments is reversed: if the test 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. The algorithms in Figures V.13 and V.12 describe the sequence of the instructions for Madd and Mdouble operations respectively.

Tables V.6 and V.7 describe the multiplications performed for both point addition and point doubling operations in three normal clock cycles when test bit is '1' or '0' respectively. The notations used for the algorithms in Figures V.13 and V.12 for point addition and point doubling were kept the same. 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 V.6. 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 resultant 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 for each bit. Total time for whole 191-bit test vector is therefore $191 \times 3 \times T$, T represents allowed frequency.

Table V.7. 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'

V.5.3 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

Table V.8. Design Implementation Summary

Design	Device (XCV)	CLB slices	Timings
Inversion in $GF(2^{191})$	3200E	1312	$1.9\eta s$
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	$59.26\mu s$

Tool F4.1i for design entry, synthesis, testing, implementation and verification of results. Table V.8 lists timing performances and occupied resources by the said architectures.

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 Figure V.10. 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 V.3.1, six and three cycles are needed for computing point addition and point doubling in Hessian form, respectively. Thus the total consumed time for computing each iteration of the algorithm of Figure V.10 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 to complete m/2 point additions (in average) and m point doublings. For our case m=191, the total time is therefore $(191/2)(600.6) + 191(300.3) = 114.71\mu s$. Point multiplication in Hessian form uses affine coordinates and therefore no time has been consumed in coordinates conversion.

Similarly, two and one multiplications are needed to perform Montgomery point multiplication, thus consuming $100.1\eta s$ and $200.2\eta s$ for point doubling and point addition respectively. Each iteration of the algorithm thus consumes $300.3\eta s$ for 3 multiplications. For our case $m = 191$, total time is therefore $191(300.3) = 57\mu s$ for elliptic curve scalar multiplication. Inversion is performed at the end. It takes 19 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 time is the sum of the time for the scalar multiplication and the time to perform inversion for coordinate conversion i-e $57.36 + 1.9 = 59.26$ μs.

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 Karatsuba Multiplier in GF(2^{191}) occupies 8721 (26.87%) CLB slices and one field multiplication is performed in $43.1\eta s$.

V.6 Performance comparison

Table V.9 provides a quick comparison of the existing FPGA's implementations of elliptic curve scalar multiplication over GF(2^m). That table sums up the last three years state of the art implementations, where most of the works featured have been published this same year. A microcoded EC processor in [72] is implemented on Annapolis Microsystems Wildstar board. For this design, EC multiplication is executed in $4300\mu s$, $8300\mu s$, and $11100\mu s$ for GF(2^{113}), GF(2^{155}), and GF(2^{173}) respectively. An efficient VLSI EC processor in [89] support EC scalar

multiplication both in GF(p) and GF(2^n). Achieved results for a 160-bit EC scalar multiplication are $1210\mu s$ and $190\mu s$ for GF(p) and GF(2^n) respectively. A generic VLSI architecture in [35] implements cryptographic primitives over various fields. It consumes $6950\mu s$ to compute point multiplication using a repeated double-and-add algorithm. Another reconfigurable system on chip ECC implementation is reported on a special architecture AT94K40 from Atmel that integrates various components including an AVR 8-bit RISC micro-controller, several peripheral devices and up to 36K bytes SRAM within a single chip. That design execute EC operation in just $1400\mu s$. All other designs [66, 92, 91, 16] implements EC scalar multiplication on single chip FPGA.

Table V.9 also includes a speedup factor that measures how much our design is faster than the others. It is obtained by dividing kP computational time of a given design over the time taken by our design. As it can be seen, our design shows an improvement ranging from 2.55 times up to 316 times of speedup.

The design presented in [89] can handle arbitrary fields and elliptic curves without changing its hardware configuration, while those parameters have been fixed in our implementation. However the design in [89] was implemented on a traditional ASIC chip, where the flexibility for design changes is quite limited or many times even inexistent. In our approach on the other hand, taking advantage of the reconfigurability feature of the platform selected, we preferred to optimize the performance of our design for a given field while the possibility to reconfigure the design for other parameters can still be instrumented.

Table V.9 provides a list of reported designs for elliptic curve scalar multiplication over GF(2^m) implemented on FPGAs or VLSI. From that list, our designs obtained the fastest computation time needing only $114.71\mu s$ in Hessian form and $59.26\mu s$ using Montgomery point multiplication algorithm.

Table V.9. GF(2^m) Elliptic Curve Point Multiplication Hardware Performance Comparison

EC scalar multiplication	Field	Platform	Freq. MHz	kP (μs)	Speed up	Multiplier Block Utilized
[72]	GF(2^{113}) GF(2^{155}) GF(2^{173})	Annapolis Micro Sys. Wildstar board		4300 8300 11100	76 148 198	Serial multiplier
[89]	GF(2^{160})	0.13μ CMOS ASIC	510.2	190	3.39	64-bit multiplier
[35]	GF(2^{176})	0.25μ CMOS	50	6950	124	1024-bit adder
[66]	GF(2^{167})	XCV400E	76.7	210	3.75	16-bit multiplier 167-bit squarer
[91]	GF(2^{191})	XCV4000XL		11820 17710	211 316	Not specified
[38]	GF(2^{163}) GF(2^{193})	XCV2000E		143 187	2.55 3.34	64-bit shift and add multiplier
[31]	GF(2^{113})	AT94K40	12	1400	25	24-bit Karatsuba
[16]	GF(2^{191})	XCV1000BG	50 36	270 2270	4.82 40	4 2-Bit level LFSR 1 Massey-Omura
Hessian Form	GF(2^{191})	XCV3200E	9.99	114.71		Two GF(2^{191}) Binary karatsuba multiplier
Montgomery Point Mult.	GF(2^{191})	XCV3200E	9.99	59.26		Two GF(2^{191}) Binary karatsuba multiplier

V.7 Conclusions

In this chapter, a generic architecture for elliptic curve point multiplication was presented. The proposed architecture is based on parallel implementations of each stage of the kP computation process. Although the architecture was optimized for reconfigurable devices, it can be applied to other hardware platforms (such as VLSI) with minor modifications.

The structure presented in this chapter constitutes a generic architecture for the scalar multiplication in Hessian form as well as for Montgomery point multiplication defined over GF(2^{191}). The resulting performance time for the scalar point multiplication operation in Hessian form is of 114.71μs, while it takes just 59.26μs to complete Montgomery point multiplication. Therefore the Weierstrass form utilizing the Montgomery group formulation can be computed in about half the execution time consumed by the time needed in the case of the Hessian form.

Two major factors contribute in achieving high performances for our architecture. First, parallel strategies that were applied throughout the design ranging from finite field arithmetic to scalar point multiplication. The efficient field units (multipliers, squarers, etc) were designed and optimized for shortest possible data critical paths. Second, we investigated for the best use of those basic building blocks that includes the structural arrangements related to the computation of point addition and point doubling as described in Table V.3. As a whole, implementation of elliptic

curve scalar multiplications on FPGA devices yields efficient architectures, showing a good balance between speed and time.

Future work includes the search for faster algorithms for elliptic curve scalar multiplication, the implementation of other design strategies and comparison between them.

Chapter VI

Conclusions

The main focus of this thesis was the study and analysis of cryptographic algorithms (CA) for their compact and high-speed implementations on reconfigurable logic platform by using optimized techniques with respect to both time and area. Three main goals were formulated to make a comprehensive study of cryptographic algorithms: (1) a study of general symmetric block-cipher cryptographic algorithms to develop general guidelines for implementing them in FPGAs which were applied to Data Encryption Standard (DES), (2) a study of architectural alternatives for implementing a modern and standard symmetric block-cipher (AES) in FPGAs, and (3) a study of basic operations required by public-key cryptography based on elliptic curves (ECC).

To achieve our first goal, we searched for the most common and fundamental operations in symmetric block ciphers. It was observed that most block ciphers were generally constructed using bit-wise logic, permutation and substitution operations. Another important feature was their iterative nature where a single encryption was concluded by repeating n iterations of the algorithm. We presented some novel techniques for implementing such operations on reconfigurable devices. We also explained whole design procedure for an FPGA architecture. Some design recommendations were provided for the optimization of an FPGA design leading to high performance circuits. A case of study about DES is presented. Therefore, our contributions obtained from this goal are: we develop general guidelines for the reconfigurable implementation of block ciphers, and we present a compact and efficient implementation of DES. Those contributions were reported in [85, 27].

To achieve our second goal, We deeply revised all steps in AES algorithm. Various options for its implementations were discussed considering performance, cost and area limitations on reconfigurable platform. Consequently, several AES designs for encryptor, encryptor/decryptor, and decryptor cores using iterative and pipelined design strategies were developed which were reported in [83, 82, 84, 75, 80].

In the first part of this study, an encryptor core was developed. Iterative and pipelined architectures were devised for it resulting a pipeline architecture requiring more area but achieving n times better throughput as compared to the iterative design.

For a full encryptor/decryptor core, we found that one of the most costly transformations in terms of area are byte substitution (BS) and its corresponding inverse (IBS) for AES. They are typically implemented as a substitution box (S-Box). Both transformations require a multiplicative inverse (MI) step plus an affine transformation (AF). Applying AF before MI produces S-Box for encryption. For decryption, inverse of AF (IAF) is applied after MI.

Since cost of BS and IBS is mainly due to MI step, in fully pipelined encryptor/decryptor core, we devised a technique for sharing a single MI module for both processes. Two implementations for MI were considered: by using pre-computed values stored in look-up tables (built-in memory modules) or by calculating values on-the-fly using composite field. AES encryptor/decryptor core using look-up table method for AES S-Box was faster but with high memory requirements. On the other hand, the composite field strategy yielded a less efficient but memory free architecture which can be suitable for cheap and portable FPGAs. In any case, our designs reported in [84] are competitive, or in some cases, even better compared to previous results.

Another consideration in developing a fully pipelined encryptor/decryptor core was made on MixColumn and Inverse MixColumn transformation (MC and IMC). A separated implementation for MC and IMC steps is a straightforward approach. However, we observed that IMC can be implemented by applying a small modification followed by MC. We evaluated both approaches whose results were reported in [75, 80]. We observed that FPGA resources were saved and the critical path for decryption was reduced resulting high throughput for the AES encryptor/decryptor core with a modified MC transformation.

Regarding AES, our last architectural consideration was focused on designing a decryptor core only. It was firstly motivated to show the difference between encryption and decryption timing for AES. In fact, due to asymmetric nature of AES, different steps are used for encryption/decryption and steps involved in decryption process are more intensive resulting higher timing for decryption. AES decryptor core was realized by isolating the data-path for decryption from the most efficient encryptor/decryptor core. Further, some steps were combined exploiting the CLB four-input/one-output configuration. S-Box/inverse S-Box were implemented using look-up table method. Inverse MixColumn was performed using MixColumn plus a small modification. Using this strategy, decryption timing was very close to that of encryption.

For all seven AES architectures, key schedule implementations were also made and all AES architectures were optimized with respect to both area and time. As a result, high speed and economical (occupying less hardware resources) AES architectures were achieved. In summary, our results for AES FPGA implementations

were either competitive or better than previous state-of-the-art AES FPGA implementations reported in the literature to-date.

The third goal of this dissertation was dedicated to public-key cryptosystem based on Elliptic Curve Cryptography (ECC). Achieved results for this work were reported in [76, 87, 81, 86, 88]. We developed FPGA implementations for performing the most important operation in ECC: elliptic curve scalar multiplication (or point multiplication). We adopted a three-stage model in which basic blocks from the lower stages were used to implement operations required in higher stages.

In the first stage, we considered the implementation of finite field Arithmetic: addition/subtraction, squaring, multiplication, and inversion in GF(2^m). Implementing addition/subtraction, squaring and then reduction for all of them is trivial. However, multiplication in GF(2^m) is complicated. We introduced binary Karatsuba multiplier, a variant of Karatsuba-Ofman multiplier which is a useful approach with respect to both time and area. Inversion in GF(2^m) is a sequence of multiplication and squaring operations. The common feature for implementing finite field arithmetic operations was to exploit maximum parallelism and to reduce critical paths producing less execution time.

Second stage for performing elliptic curve scalar multiplication is elliptic curve arithmetic: point addition and doubling. Both operations need building blocks for multiplication and squaring from first stage. Since, multiplication in GF(2^m) occupies large space, a study was made for maximum multipliers allowed by target FPGA that can produce maximum throughput at this stage. It was found that we are limited to two multipliers in parallel on VirtexE device XCV3200.

In the third stage of our model, the elliptic curve scalar multiplication was finally implemented by using building blocks from the two previous stages. A generic architecture was proposed for the implementation of elliptic curve scalar multiplication. That architecture was further used for FPGA implementations of Hessian form of Elliptic curve and Montgomery point multiplication. Due to careful considerations at all stages of the design, our results improve existing FPGA implementations of Elliptic curve scalar multiplication reported in the literature to-date.

Concluding Remarks

In this thesis work, we have analyzed cryptographic algorithms for high-speed implementation on reconfigurable logic platform. Concluding remarks for this work are drawn as follows:

1. We focus on both symmetric and asymmetric cryptographic algorithms. The study about symmetric algorithms is mainly dedicated to block ciphers. Basic structure, design principles and most frequent operations in block ciphers are explained. It has been observed that modern block ciphers are more efficient and resistive but they exhibit similar structure: same operations such as transposition and substitution (in improved form), which are repeated for

number of times. Asymmetric algorithms involve complex mathematical operations however some useful efforts have been observed for making them simple and proficient.

2. The concept of parallelism can be effectively used for symmetric block ciphers. Parallel operations in a block cipher can easily be detected. Basic building blocks in a block cipher can be opted for several combinations in a parallel way. In asymmetric ciphers, parallel operations are not common. However, an effort has been in this thesis work for developing parallel algorithms for them.

3. Reconfigurable logic platform proves to be a suitable platform for implementing cryptographic algorithms. Basic structure of reconfigurable hardware devices is found to be suitable for implementing basic primitives in cryptographic algorithms. Moreover, high density FPGAs can accommodate big circuits due to parallel approaches for cryptographic algorithms. Reconfigurable logic platform offers several useful benefits for cryptographic algorithm implementations but it may not be a right choice for all kinds of security applications or all kinds of cryptographic algorithms.

4. Parallelism is a key feature of this thesis work for achieving high design performances equally useful for other similar applications. It requires a deep analysis at algorithm level in searching parallel operations among cryptographic algorithms. It works at design level in designing involved operations according to the structure of target devices. It is useful when hardware resources allow multiple building blocks for a single operation. However, the concept of parallelism is not based on occupying unlimited hardware resources for achieving high performances but on the quality work exploiting parallelism where it has a worth.

5. Optimization is an essential task for an FPGA architecture in improving design performance with respect to time and space. It requires manual work in eliminating overheads in a design: using schematic components for an FPGA design is cumbersome as compared to HDLs but it facilitates a synthesis tool while placing and routing design components. Similarly, Synthesis tools allow a designer to put several time and area constraints according to design specifications. However, synthesis tools takes long time (in hours) in achieving these goals and sometimes also fail.

Future Work

We believe that on the basis of this dissertation work, there are strong chances for more contributions in the field of cryptography. As a future work, a short description of some of them is provided below:

1. A hardware-software approach for implementing cryptographic algorithms can be helpful in achieving high performances. Some parts in a cryptographic

algorithm can efficiently be implemented in software and some of them in hardware. Modern FPGAs include microcontrollers besides their standard resources and also establish an interface for communication between them. Such type of FPGAs can help in reducing communication overheads between software-hardware methods.

2. Faster algorithms for the basic transformations of cryptographic algorithms are always investigated for improving overall performances of cryptographic schemes. New techniques need to be explored. As an example, ECC implementations in this thesis are based on addition-doubling method for EC arithmetic. A new technique based on addition-halving [40] for EC could be an option for improving design performance.

3. Similar cryptographic schemes can be implemented for comparison between timing and occupied hardware resources by them. ECC is considered for providing same security at shorter key length as compared to cotemporary algorithms such as RSA. A comparison between ECC and RSA implementations would be interesting. Similarly, Koblitz curves for ECC are believed to be more secure. A comparison of Koblitz curve implementations with other curves can provide cost estimations for achieving higher security.

4. Cryptographic solutions for smart cards pose a challenge since security is desired under limited hardware resources. Compact and fast cryptographic solutions for smart cards can be a useful and challenging task.

5. Hardware implementations provide physical security. But it is still an open question whether they are secure against other possible attacks such as side channel attacks.

6. In this thesis work, reconfigurable logic platform was chosen for implementing cryptographic algorithms. However, other possible options (software methods using embedded processors, VLSI) need to be explored for resolving time and space issues.

7. Several design strategies can be adopted achieving high data rates for cryptographic algorithms. A comparison for iterative and pipeline approaches was included in this thesis work. Use of other design strategies such as inner-round or outer-round looping could be an effort to determine cost for the design in terms of hardware area for high speed.

8. In recent years, security applications using cryptographic algorithms have been increased. To combat current security threats, it is required to provide security solutions for large number of security applications within their own framework. This thesis work is required to be extended to other cryptographic algorithms such as streams ciphers.

Bibliography

[1] Polycom. URL: http://www.polycom.com/common/pw_item_show_doc/0,1276,3076,00.pdf.

[2] Nokia 6225 Phone. URL: http://www.nokiausa.com/phones/6225.

[3] Ericcson. URL: http://www.erricson.com/.

[4] International Telecommunication Union. URL: http://www.itu.int/home/index.html.

[5] European Telecommunications Standards Institute. URL: http://www.etsi.org/.

[6] IEEE 802 LAN/MAN Standards Committee. URL: http://grouper.ieee.org/groups/802/index.html.

[7] ETSI Technical Specification. Access transmission systems on metallic access cables; Very High Speed Digital Subscriber Line (VDSL); Part 1: Functional requirements.

[8] Xilinx Virtex TM-E 1.8V Field Programmable Gate Arrays. URL: http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp.

[9] Xilinx Virtex TM-E 1.8V Field Programmable Gate Arrays. URL: www.xilinx.com.

[10] Hardware IP Cores of Advanced Encryption Standard AES-Rijndael. URL: http://ece.gmu.edu/crypto/rijndael.htm.

[11] High Performance Solution in Silicon: AES (Rijndael) Cores. URL: http://www.heliontech.com/core2.htm.

[12] ANSI T1E1.4, Sep. 1 1999. Draft Technical Document, Revision16, Very High Speed Digital Subscriber Lines; System requirements.

[13] Free-DES Core(2000), March 2000. URL: http://www.free-ip.com/DES/.

[14] ANSI X9.63. Public Key Cryptography for the Financial Services Industry: *Elliptic Curve Key Agreement and Key Transport Protocols,* working draft., August 1999.

[15] E. Bach and J. Shallit. *Algorithmic number theory, Volume I: efficient algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.

[16] M. Bednara, M. Daldrup, J. Shokrollahi, J. Teich, and J. von zur Gathen. Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. In *Proc. of The 9th Reconfigurable Architectures Workshop (RAW-02)*, Fort Lauderdale, Florida, U.S.A., April 2002.

[17] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. Efficient Software Implementation of AES on 32-bits platforms. In *Proceedings of the CHESS 2002*, pages 159–171, LNCS 2523, 2002.

[18] 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.

[19] Matt Bishop. An application of a fast Data Encryption Standard implementation. In *Computing Systems, 1(3)*, pages 221–254, Summer 1988.

[20] certicom^{TM}. ECC Tutorial. http://www.certicom.com/index.php?action=ecc_tutorial,home.

[21] Stephen Charlwood and Philip 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.

[22] L. Childs. *A concrete introduction to higher algebra*. Springer-Verlag Berlin Heidelberg, Germany, 1995.

[23] 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.

[24] Joan Daemen and Vincent Rijmen. *The design of Rijndael, AES-The Advance Encryption Standard*. Springer-Verlag Berlin Heidelberg, New York, 2002.

[25] 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.

[26] Marc Davio, Yvo Desmedt, Jo Goubert, Frank Hoornaert, and Jean Jacques Quisquater. Efficient hardware and software implementations for the DES. In *Proc. of Crypto' 83*, pages 144–146, August 1984.

[27] Arturo Díaz-Pérez, Nazar A. Saqib, and Francisco Rodríguez-Henriquez. Some Guidelines for Implementing Symmetric-Key Cryptosystems on Reconfigurable-Hardware. In *(*Accepted for*) IV Jornadas de Computación Reconfigurable y Aplicaciones*, Barcelona, Spain, September.

[28] 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, 1992.

[29] H. Eberle and C.P. Thacker. A 1 Gbit/second GaAs DES chip. In *Proc. IEEE 1992 Custom Integrated Circuits Conference*, pages 19.7/1–4, New York,USA, 1992. Springer-Verlag, 1992.

[30] 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.

[31] M. Ernst, M. Jung, and F. Madlener et. al. A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $GF(2^n)$. *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, 2523:381–399, August 2003.

[32] David C. Feldmeier. A high speed crypt program, April 1989. Technical Memo TM-ARH-013711.

[33] Kris Gaj and Pawel Chodowiec. Comparison of the Hardware Performance of the AES Candidates using Reconfigurable Hardware. In *The Third AES3 Candidate Conference*, New York, April 2000.

[34] Brian Gladman. The AES Algorithm (Rijndael) in C and C++. URL: http://fp.gladman.plus.com/`cryptography_technology`/rijndael/.

[35] J. Goodman and A.P. Chandrakasan. An Energy-Efficient Reconfigurability Public-Key Cryptography Processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[36] J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology-CRYPTO 97, LNCS 1294*, pages 342–356, Springer-Verlag, Berlin, Germany, 1997.

[37] J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. *In B. S. Kaliski Jr. (editor)* Advances in Cryptology —CRYPTO 97 *Lecture Notes in Computer Science*, 1294:342–356, 1997.

[38] 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.

[39] D. Hankerson, J. Lopez-Hernandez, 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.

[40] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Cryptography*. Springer-Verlag Newyork, Inc., 2004.

[41] 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.

[42] T. Ichikawa, T. Kasuya, and M. Matsui. Hardware Evaluation of the Aes Finalists. In *The Third AES3 Candidate Conference*, New York, 2000.

[43] 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.

[44] 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.

[45] 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.

[46] Philip R. Karn. Karns DES implementation source code.

[47] N. Koblitz. *Introduction to elliptic curves and modular forms*. Springer-Verlag Berlin Heidelberg, Germany, 1984.

[48] N. Koblitz. CM-curve with good cryptographic properties. In *Advances in Cryptology, Crypto'91*, pages 279–287. Springer-Verlag, 1992.

[49] J. Leonard and W.H. Magione-Smith. A case study of partially evaluated hardware circuits: key specific DES. In *Proc. Field-Programmable Logic and Applications, FPL' 97*, pages 234–247, London, UK, September 1997. Springer-Verlag, 1997.

[50] 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.

[51] 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 CHESS 2002*, pages 171–184, LNCS 2523, 2002.

[52] J.V. McCanny Maire McLoone. High Performance FPGA Rijndael Algorithm Implementation. In *Proceedings of the CHESS 2001*, pages 68–80, LNCS 2162, 2001.

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

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

[55] 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.

[56] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.

[57] 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.

[58] A. J. Menezes, Paul C. van Oorschot, and Scott A.Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1996.

[59] A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[60] 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.

[61] 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.

[62] S. Morioka and A. Satoh. An Optimized S-Box Circuit Architecture for Low Power AES Design. In *Proceesings of the CHESS 2002*, pages 172–183, LNCS 2523, 2002.

[63] Motorola. URL: http://www.motorola.com/LMPS/RNSG/federal/products/secure_telecom/CipherTAC_2000.html.

[64] Randall K. Nichols and Panos C. Lekkas. *Wireless Security: Models, Threats, and Solutions*. McGraw Hill, 2000.

[65] NIST. Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Standards Publication, Nov. 2001. URL: http://csrc.nist.gov/CryptoToolkit/aes/index.html.

[66] 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.

[67] 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.

[68] IEEE P1363. *Standard specifications for public-key cryptography.* IEEE standards documents, "http://grouper.ieee.org/groups/1363/", draft version 7 edition, September 1998.

[69] C. Paar. Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields: PhD thesis: Universitat GH Essen,1994. VDI Verlag.

[70] 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.

[71] 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.

[72] philip:des. Philip R. Karn. Technical Memo TM-ARH-013711.

[73] ANSI X9.17 (Revised):. National Standards for financial institution key management (wholesale), American Bankers Association, 1986.

[74] 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)*, Cancun, Mexico, May 2003.

[75] F. Rodríguez-Henriquez, N. A. Saqib, and A. Díaz-Pérez. 4.2 Gbit/s Single-Chip FPGA Implementation of AES Algorithm. In *ELECTRONICS LETTERS*, volume 39, pages 1115–1116. Springer-Verlag Berlin Heidelberg 2003, July 2003.

[76] F. Rodríguez-Henriquez, N. A. Saqib, and A. Díaz-Pérez. A Fast Parallel Implementation of Elliptic Curve Point Multiplication over GF(2^m). *Microprocessor and Microsystems*, 28:329–339, August 2004.

[77] Francisco Rodríguez-Henriquez. New Algorithms and Architectures for Arithmetic in GF(2^m) suitable for Elliptic Curve Cryptography, PhD thesis: Oregon State University, 2000.

[78] K. Rosen. *Elementary Number Theory and its Applications*. Addison-Wesley, Reading, MA, 1992.

[79] Atri Rudra, Pardeep K. Dubey, Charanjit S. julta, Vijay Kumar, Jodyuls R.Rao, and Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. In *Proceedings of the CHESS 2001*, pages 171–184, LNCS 2162, 2001.

[80] N. A. Saqib, A. Díaz-Pérez, and F. Rodríguez-Henriquez. Highly Optimized Single-Chip FPGA Implementations of AES Encryption and Decryption Cores. In *X Workshop Iberchip*, pages 117–118, Cartagena-Colombia, March 2004.

[81] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF(2^m). In *Proceedings of 18th International Parallel & Distributed Processing Symposium (RAW 2004)*, page 144, Santa Fé, New Mexico, April. IEEE Computer Society Press.

[82] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo 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.

[83] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo 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.

[84] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. Two Approaches for a Single-chip FPGA Implementation of an Encryptor/Decryptor AES Core. In *FPL 2003, Lecture Notes in Computer Science 2778*, pages 303–312. Springer-Verlag Berlin Heidelberg 2003, 2003.

[85] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. A Compact and Efficient FPGA Implementation of the DES Algorithm. In *(*Accepted for*) International Conference on Reconfigurable Computing and FPGAs (ReConFig04)*, Colima, Mexico, September 2004. Mexican Society for Computer Sciences.

[86] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. *A Generic Coprocessor For Elliptic Curve Scalar Multiplication on Hardware*. Nova Science Publishers, New York, 2004. *In press.*

[87] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. A Parallel Architecture for Computing Scalar Multiplication on Hessian Elliptic Curves. In *International Symposium on Information Technology (ITCC 2004)*, pages 546–552, Las Vegas (NV), USA, April 2004. IEEE Computer Society Press.

[88] Nazar A. Saqib, Francisco Rodríguez-Henriquez, and Arturo Díaz-Pérez. A Reconfigurable Processor for High Speed Point Multiplication in Elliptic Curves. *International Journal of Embedded Systems, (*In press for 2004 *)*, 2004.

[89] A. Satoh and K. Takano. A Scalable Dual-Field Elliptic Curve Cryptographic Processor. *IEEE Transactions on Computers*, 52(4):449–460, April 2003.

[90] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, 1996.

[91] 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.

[92] 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.

[93] Sony. URL: http://www.sony.com/.

[94] William Stallings. *CRYPTOGRAPHY AND NETWROK SECURITY:* Pprinciples and Practice. Prentice Hall, Upper Saddle River, New Jersey 07458, 1999.

[95] B. Sunar and Ç. K. Koç. Mastrovito multiplier for all trinomials. *IEEE Transactions on Computers*, 48(5):522–527, May 1999.

[96] TCC. URL: http://www.tccsecure.com/csd4100.htm.

[97] W. Trappe and L.C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, Inc., Upper Saddle River, NJ 07458, 2002.

[98] Joachim von zur Gathen and Michael Nöcker. Computing special powers in finite fields: extended abstract. In *Proceedings of the 1999 international symposium on Symbolic and algebraic computation*, pages 83–90. ACM Press, 1999.

[99] 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.

[100] S. Wicker. *Error control systems for digital communication and storage*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[101] S. B. Wicker and V. K. Bhargava (editors). *Reed-Solomon Codes and Their Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[102] 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 *CHESS 99*, pages 37–48, LNCS 1717, August 1999.

[103] 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.

[104] H. Wu and M. A. Hasan. Low complexity bit-parallel multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(8):883–887, August 1998.

[105] ANSI X9.62. Federal Information Processing Standard (FIPS) 46, National Bureau Standards, January 1977.

GLOSSARY

Area (hardware) Hardware resources occupied by the design. In terms of FPGAs, hardware area includes number of CLBs, memory blocks, IOBs, etc.

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.

Confusion Confusion makes the output dependent on the key. Ideally every key bit influences every output bit.

Decryption The process of retrieving plaintext from ciphertext is called decryption.

Plaintext Diffusion makes the output dependent on the previous input (plaintext/ciphertext). Ideally each output bit is influenced by every input bit.

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 Given an elliptic curve E and a point P on E, the scalar product is then Q = nP. The elliptic curve discrete logarithm problem (ECDLP) is then to determine the integer n, given points P and Q, and given that nP = Q.

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.

FPGA A field-programmable gate array or FPGA is a gate array that can be reprogrammed, after it is manufactured.

Iterative Looping It implements only one round and n iterations of the algorithm are carried out by feeding back previous round results.

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 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.

Plaintext In cryptographic terminology, message is called plaintext

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.

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