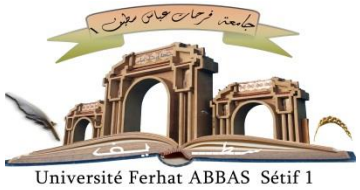
الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research



UNIVERSITY FERHAT ABBAS - SETIF1

FACULTY OF TECHNOLOGY

THESIS

Submitted to the Department of Electronics

In Fulfilment of the Requirements for the degree of

DOCTORATE

Domain: Sciences and Technologies

Field: Electronics **Specialty: Electronics of Embedded Systems**

By

Abdelaziz KERBOUCHE

THEME

Optimized FPGA Implementation of Round-Based Ciphers: A High-Performance and Area-Efficient AES Implementation

Defended on /02/2025 in front of Jury:

Hamida Abdelhak FERHAT	**Professor**	**Univ. Ferhat Abbas Sétif 1**	**President**
Mouloud AYED	**Professor**	**Univ. Ferhat Abbas Sétif 1**	**Thesis director**
Abdelhalim MAYOUF	**Professor**	**Univ. Ferhat Abbas Sétif 1**	**Examiner**
Abdelhamid DAAMOUCHE	**Professor**	**Univ. M'hamed Bougara Boumerdes**	**Examiner**
Abdellatif KHELIL	**Professor**	**Univ. Hamma Lakhder El Oued**	**Examiner**
Hamimi CHEMALI	**Professor**	**Univ. Ferhat Abbas Sétif 1**	**Guest**

الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de L'Enseignement Supérieur et de la Recherche Scientifique

Université Ferhat ABBAS Sétif 1

UNIVERSITÉ FERHAT ABBAS - SETIF1

FACULTÉ DE TECHNOLOGIE

THÈSE

Présentée au Département d'Électronique

Pour l'obtention du diplôme de

DOCTORAT

Domaine : Sciences et Technologie

Filière : Électronique **Option : Électronique des Systèmes Embarqués**

Par

KERBOUCHE Abdelaziz

THÈME

Implémentation optimisée sur FPGA de chiffrements à base de rondes : une architecture AES performante et à faible empreinte matérielle

Soutenue le /02/2025 devant le Jury :

FERHAT Hamida Abdelhak	**Professeur**	**Univ. Ferhat Abbas Sétif 1**	**Président**
AYED Mouloud	**Professeur**	**Univ. Ferhat Abbas Sétif 1**	**Directeur de thèse**
MAYOUF Abdelhalim	**Professeur**	**Univ. Ferhat Abbas Sétif 1**	**Examinateur**
DAAMOUCHE Abdelhamid	**Professeur**	**Univ. M'hamed Bougara Boumerdes**	**Examinateur**
KHELIL Abdellatif	**Professeur**	**Univ. Hamma Lakhder El Oued**	**Examinateur**
CHEMALI Hamimi	**Professeur**	**Univ. Ferhat Abbas Sétif 1**	**Invité**

Année : 2024-2025

Abstract

Modern life relies heavily on embedded systems, which, despite their compact and resource-constrained nature, serve as the core intelligence behind most devices. These smart systems collect user data to continuously improve decision-making processes. However, the gathered data often includes sensitive information, highlighting the critical need for robust security measures that ensure trustworthiness and reliability without compromising performance. This project aims to identify an optimal implementation technique for security algorithms in embedded systems, balancing resource utilization and timing performance. The primary challenge addressed is maintaining the integrity of the security algorithm while optimizing hardware architecture to maximize efficiency. The Rijndael algorithm, specifically the Advanced Encryption Standard (AES), is chosen as the security core and implemented using two distinct techniques. The proposed approach is designed for Field-Programmable Gate Arrays (FPGAs), with results benchmarked against similar projects in terms of timing and area performance.

Keywords: Advanced Encryption Standard (AES) algorithm, Field Programmable Gate Array (FPGA), Cryptography, embedded systems

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Pr Hamimi CHEMALI, and Pr Mouloud AYAD, for their invaluable guidance, continuous support, and encouragement throughout my research journey. Their expertise and insightful advice have greatly contributed to the completion of this work.

I extend my sincere thanks to the members of the examination committee, Pr Hamida Abdelhak FERHAT (University of Setif 1), Pr Abdelhalim MAYOUF (University of Setif 1), Pr Abdelhamid DAAMOUCHE (University of Boumerdes), and Pr Abdullatif KHELIL (University of El Oued), for their valuable time, constructive feedback, and thoughtful suggestions, which have significantly improved the quality of this thesis.

My appreciation also goes to Ferhat Abbas University of Setif1 and LCCNS laboratory for providing the necessary resources, facilities, and a stimulating research environment that have been instrumental in the successful completion of my PhD project.

Finally, I would like to express my heartfelt thanks to Dr Salaheddine LAIB and Mr Zinedine MENNANI for their generous help, insightful discussions, and encouragement, which have greatly enriched my research experience.

My gratitude to all those who have contributed to this work in various ways, including friends, colleagues, and family members, for their unwavering support, encouragement, and assistance throughout this journey.

Dedication

To my dear parents,
whose constant love, countless sacrifices, and endless wisdom have shaped who I am today. Your support has guided me, and your faith in me has been my greatest strength.

To my beloved wife,
my companion through every challenge and success. Your patience, support, and steady belief have been my comfort throughout this journey. Thank you for being by my side every step of the way.

To my wonderful children,
who fill my life with inspiration and happiness. Your bright smiles and endless curiosity remind me daily of the joy of learning.

To my whole family, this work is dedicated to you all, with heartfelt love and gratitude

General introduction

An embedded system is a microcontroller or microprocessor-based system, designed to perform a specific task, either independently or as part of a larger system. It typically consists of two main components: hardware and software, serving as the core intelligence of electronic systems and playing a crucial role in various aspects of daily life. With ongoing technological advancements, embedded systems are playing an increasingly vital role in decision-making. Their importance continues to grow, particularly in smart and cloud-based applications like IoT solutions, where real-time data processing and automation are essential [1]. Numerous areas rely on embedded systems, such as defense, banking, healthcare, and other critical fields, where even a slight security failure may lead to catastrophic consequences for individuals [2]. One example of security failure is the recent remote cyberattack targeting embedded systems in some wireless communication devices, causing widespread disruption, and physical harm to numerous individuals. A similar cyber operation was performed remotely to attack electronic systems that are embedded in a highly sensitive facility, causing significant damage to instruments, and leading to a major failure in the system. These incidents and others highlight the critical risks of an inadequate or insufficient security, where vulnerabilities can be exploited with devastating consequences, which underscores the urgent need for robust security measures to protect systems and safeguard lives [3]. Security measures in embedded systems must address diverse threats, including physical tampering, side-channel attacks, malware, and communication interception. However, the implementation of robust security mechanisms, which usually require high computational capabilities and significant energy within a constrained systems is a difficult operation [4]. Scientists and researchers have thoroughly focused on this challenge, leading to the creation of practical solutions that successfully balance the need for security with the inherent limitations of embedded systems.

This thesis focuses on data security as a critical aspect, emphasizing its role in preventing unauthorized access to sensitive information, or injecting unwanted data, either during a run-time, during data transfer, or a storage operation. Among all security levels, this thesis focuses on data encryption, with a general overview of other security aspects. The research studies the implementation of data security within embedded systems and the challenges associated to it. The practical side of this project aims to the identification of the optimal trade-off between embedding security and maintaining systems' performance through smooth data processing. In this project, an FPGA implementation of the well-known Advanced Encryption Standard (AES) algorithm is elaborated using different structures, to address embedded systems constraints; with a main challenge of preserving the original algorithm without modifications [6].

The development process includes several stages: design, optimization, unit testing, integrity testing of individual modules, and finally, simulation of the overall system. Resource utilization and timing performance form a monitoring parameter along with the efficiency ratio. This comparative analysis highlights the effectiveness of the proposed approaches in balancing between security and efficiency. The remainder of this thesis is structured as follows: Chapter 1 presents a comprehensive literature review of related works and proposed solutions in the field. Chapter 2 provides a foundational background on cryptography, security algorithms, and their practical use cases. Chapter 3 examines common optimization techniques for FPGA designs. Finally, Chapter 4 details the methodology of the proposed project, employing various implementation techniques, and includes an in-depth discussion and analysis of obtained results.

Chapter 1. Literature Review

Embedded systems are fundamental to modern technology, serving as critical components in a diverse array of applications, ranging from consumer electronics to essential infrastructure systems. However, as the complexity of these systems continues to grow, so too do the challenges associated with their design and implementation. Among the most pressing challenges is the need to ensure robust security without compromising performance or efficiency. This chapter presents a comprehensive literature review of the challenges related to integrating security mechanisms into embedded systems, exploring the inherent trade-offs between security, performance, and resource constraints. Additionally, it examines the various solutions proposed in this context, highlighting innovative approaches and methodologies that aim to address these challenges while maintaining the efficiency and reliability of embedded systems.

1.1. Limitations and challenge

Embedded systems face a major challenge when integrating additional safety measures, mainly due to their strict constraints on time, resources and performance. These systems often depend on precise process control with minimal latency. Accordingly, the introduction of addition security mechanisms, such as encryption or secure boot processes, leads usually to a latency in processing, which can be considerable in some timing-critical applications [1].

Closely related to timing, the challenge of resource management is equally crucial. Embedded systems generally operate in a limited resource environment, namely, constrained area on hardware chips, or computational power, which require some optimization strategies to bypass this limitation. Which is opposed to security cores that usually require dedicated resource allocation, with some high capabilities and an extended power need. This challenge has attracted the attention of researchers and developers to adopt effective resource management techniques, and ensure that the integration of security functionalities does not degrade overall performance [1]. While enhancing security is essential, the additional layers it introduces can slow down the system, particularly in real-time applications. For instance, implementing secure boot processes in a smart appliance may improve security but could also lead to longer boot times, negatively impacting user experience. This trade-off necessitates careful evaluation during the design phase, to ensure that security measures do not undermine the system's efficiency. Developers must strike a delicate balance between robust security and optimal performance to meet the demands of real-time applications [2].

Beyond these technical challenges, scalability and integration pose additional complexities, especially as embedded systems are deployed across diverse applications such as IoT devices. This requires adaptable and scalable solutions that can accommodate the unique demands of different environments without compromising security or functionality.

In summary, integrating security into embedded systems involves navigating a complex landscape of challenges, including timing constraints, resource limitations, performance trade-offs, and scalability issues. Addressing these challenges requires a complete approach that combines thoughtful design, optimization, and a deep understanding of the interplay between security and system constraints. By carefully considering these factors, developers can create embedded systems that are not only secure but also reliable and high-performing.

1.2. Implementation of cryptographic algorithms

Field Programmable Gate Arrays (FPGA) are an ideal platform for embedded system developers, offering the flexibility to redesign and describe desired architectures using a high-level language. FPGA is an integrated circuit or chip that enables the creation of fully customized digital logic. It consists of numerous logic cells that serve as the fundamental building blocks for designing digital circuits. These cells can be configured to operate in specific ways by interconnecting them and optimizing their functionality to form the core of any digital circuit. Additionally, FPGAs provide access to various resources, such as clock signals, RAM blocks, and interfaces for managing different types of inputs. Some advanced FPGAs include also peripherals like analog-to-digital converters and analog outputs. One of the key advantages of FPGAs is the ability to support parallel processing, allowing multiple operations to execute within a single clock cycle. A simple architecture of FPGA is represented in Fig. 1.1.

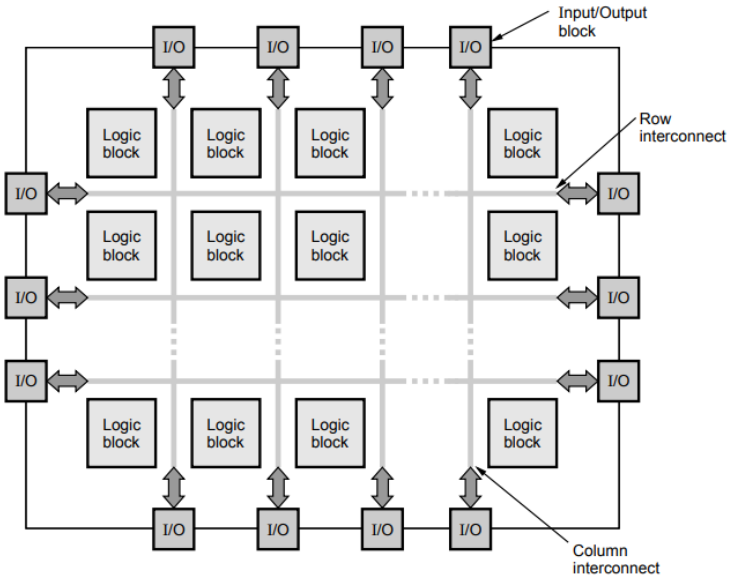


Fig. 1.1. Simple FPGA architecture.

Implementing cryptography and encryption algorithms in embedded systems presents significant challenges due to the complexity of the mathematical operations involved. These algorithms often require extended processing times because of their recursive nature, which can strain the limited computational resources of embedded systems. Compounding this challenge is the fact that data

encryption in such systems is typically a secondary task, operating quietly in the background without disrupting primary functions. To address this, the encryption unit is often integrated as a black box, designed to process input data and produce output as efficiently and seamlessly as possible.

A prime example of this complexity is the Advanced Encryption Standard (AES), a widely used encryption algorithm. The processing time of AES depends on rounds, which are iterative sets of operations executed sequentially. In this structure, the output of one function serves as the input to the next, creating a recursive sequence that is repeated N times based on the key length. This design introduces latency, as each processing unit remains idle while waiting for input from the previous operation. As illustrated in Fig. 1.2, the chronogram of an AES encryption process reveals that the holding time, which is defined by the period during which units wait for input, is considerable. When analyzing the rate of processing time relative to total time in an iterative AES design, the latency becomes even more apparent, highlighting the need for optimization in embedded systems where efficiency is critical.

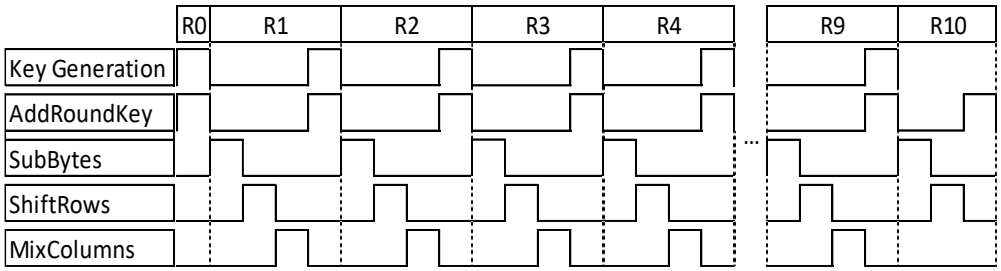


Fig. 1.2. AES process of a single plaintext.

The chronogram demonstrates that only one in four operations is executed each time, resulting to an operational rate of $R = 24\%$, which signifies a 76% of holding time as shown in Fig. 1.3.

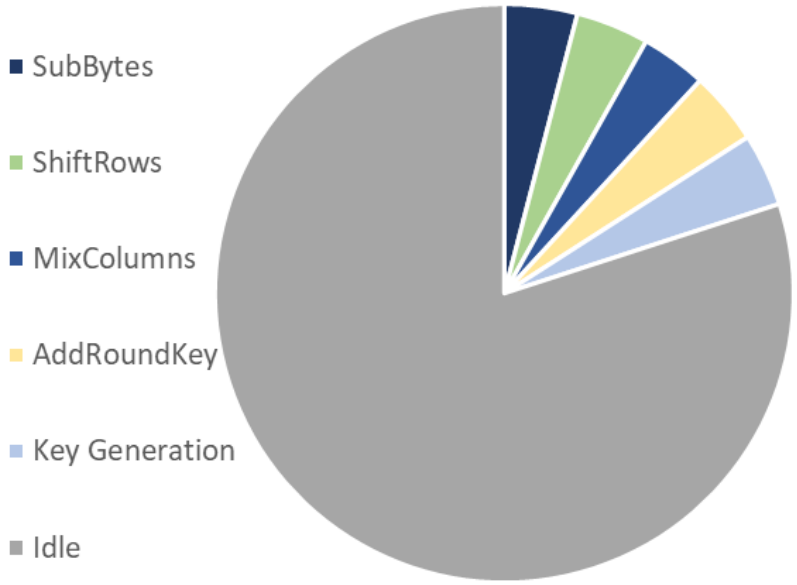


Fig. 1.3. Operational time rate in AES.

In contrast, the pipeline implementation of the AES divides the encryption process into multiple stages that operate concurrently. This parallelism allows each stage to process data independently, eliminating the idle waiting time and enabling a continuous flow of operations. As shown in Fig. 1.4, the chronogram of the pipeline approach demonstrates a high reduction in holding time, leading to a faster and more efficient encryption process. The throughput of the system increases significantly, as data moves seamlessly through the pipeline without the bottlenecks inherent in the traditional method.

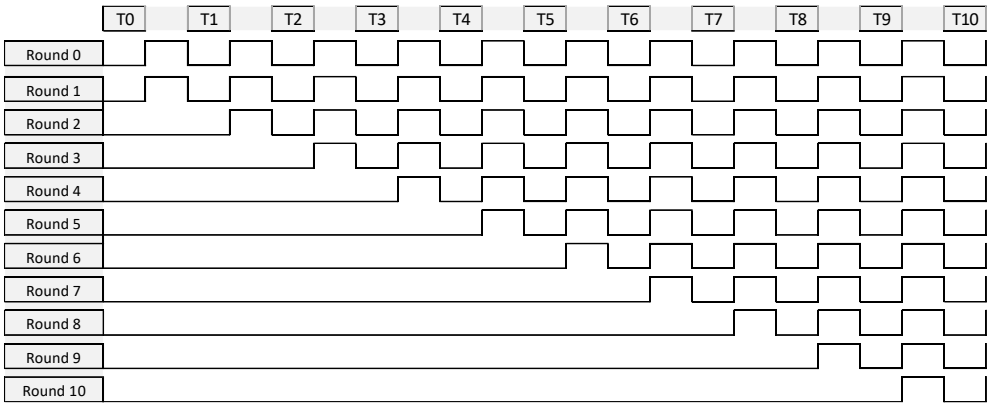


Fig. 1.4. Chronogram of pipeline implementation.

The chronogram in Fig. 1.4 clearly shows that the operational rate, defined as the ratio of active processing modules to holding modules, is approximately 50% for the first input. Since each module performs a single round, the first module becomes available to process the next plaintext immediately after completing the current one. Additionally, the operational rate reaches 100% after 10 ciphertexts with a steady data flow.

The key takeaway that we conclude from the comparison made above, is the profound impact that the choice of implementation method has on the overall performance of cryptographic algorithms in embedded systems. By carefully selecting and optimizing the implementation approach (whether iterative, pipeline, or another technique) designers can significantly enhance processing speed, reduce latency, and improve resource utilization. The implementation technique is not the only method to enhance performance, other techniques are thoroughly described in literature, addressing the efficiency of the implementation of cryptography algorithms in hardware and software. The following section enumerates some of recent works on this field.

1.3. Related works

The growing demand for secure and efficient embedded systems has driven researchers to explore various techniques for optimizing security cores, particularly in the context of cryptography and encryption. As embedded systems often operate under stringent constraints, such as limited computational power, memory, and energy resources, traditional cryptographic algorithms and their

implementations may not always be suitable. To address these challenges, researchers have proposed a wide range of optimization strategies, each targeting different aspects of the design and execution of security cores. This section provides a comprehensive overview of these optimization techniques, categorized based on the type of modification they introduce.

1.3.1. New lightweight block ciphers

Several approaches are proposed in literature to propose new lightweight block ciphers. Authors in [3], developed a lightweight block cipher, the proposed algorithm was examined to test its efficiency, where it acquires a good level of cryptographic characteristics, and a crypto analyses was performed to test security. The algorithm is derived from the WG stream cipher. The proposed algorithm was implemented in a low-power MCU to evaluate its performance and compared with other similar works. The contribution of this work was on the power consumption which represents a major limitation in embedded systems. This proposed cipher was designed for energy-constraint systems, especially those powered from an external source such as RFID tags, smart cards, wireless sensors etc.

In [4], authors designed a combination of Feistel and ARX structures to develop a new block cipher GFRX for resources-limited systems. The proposed algorithm intervenes in the round function by reducing complex operations, and take advantage of the flexible combination of Feistel and ARX to resolve the diffusion and confusion latency in traditional structures. This project shows better results compared to ARX and Feistel separated. It was tested against effective differential and linear attacks, and therefore is considered secure.

Yan et al in [5], suggested a new dynamic S-box based block cipher DBST, this new approach is based on bit-slice technology, where the s-box depends on the key. This new approach increases randomness, and enhance security against differential and linear attacks. The rounds operation is based on a new Feistel variant structure, and shows more infusibility compared to traditional structure. The authors compared their work to similar lightweight algorithms, and concluded that it is well adapted for 5G applications.

Authors in [6], investigated a new block cipher RAZOR designed for IoT application. The new approach is aimed to enhance security, but also to be resource-efficient for resource-limited systems. The diffusion layer is based on XOR operations and rotation, the project was implemented in software. Security analyses is proven against both differential and linear attacks. RAZOR was compared with other similar algorithms in terms of security.

A lightweight crypto algorithm named LBC-IoT is proposed in [7]. The objective of the project is to design a security-enhanced algorithm adapted to resource-limited systems. LBC-IoT uses a compact s-box. The advantage of the proposed algorithm is its simplicity, and the low GE it has compared to

similar works. Additionally, it is well adapted to resource-limited devices as it has a small software footprint.

Buja et al in [8] presented a survey on lightweight ciphers for mobile big data computing. The study addresses data security concerns on mobile, and concluded that existing lightweight algorithms have some issue that should be taken in consideration. The necessity to develop data security algorithms for resource-limited systems, and the drive to propose compact algorithms should never come at the expense of ignoring or downplaying security concerns. The advancement of computer calculation capacities accelerates the attacks technique and put current data security systems into danger, which requires finding new solution, or enhancing existing methods.

1.3.2. Lightweight hash functions

Hash functions have also been a significant focus of research, particularly in optimizing their efficiency and security for resource-constrained embedded systems. Andrey et al, proposed in [9] a lightweight Hash function named SPONGENT. The new function supports 88, 128, 160, 224, and 256 bits. This function has a small footprint, compared to similar functions, while for throughput and area requirement, they depend on the technology used.

In [10], Susila et al, presented an implementation of a lightweight hash functions LWCHF. The proposed implementation is done in hardware and software, and results were examined based on nine metrics. The cost of the final solution, and the security against different attacks, in addition to performance were considered in the development process.

Sevin and Çavuşoğlu in [11], proposed a new lightweight hash based on encryption algorithm. The efficiency of proposed system was examined, and security performance was carried out and detailed in the study. The algorithm shows a robustness against differential and linear attacks, and sensitivity of hash value analyses showed a high precision for all cases.

Jian, Thomas and Axel in [12], proposed PHOTON, based on a well-known AES design strategy, with the introduction of a new layer without affecting its size, to make fit small area devices. The proposed approach attained an excellent area / throughput trade-off.

1.3.3. Implementation optimization

Another optimization approach focuses on enhancing the implementation technique of the algorithm itself, without making adjustments to its core structure. A pipeline and parallel implementation of the AES is presented by Nabil et al in [13]. The aim of this research is the reduction of processing time. The idea was to use maximum resources available in the design, and eliminate idle time. Results are compared to traditional implementation method proposed by them, and also compared to software

implementation and similar hardware approaches, in terms of resources and timing performance. The parallel architecture achieved the highest throughput at the cost of increased resource utilization.

A hybrid pipeline architecture was proposed in [14] by Algredo-Badillo et al. The proposed architecture combines time redundancy and hardware redundancy, and results show an enhanced resource utilization compared to standard AES. The designed system is based on error detection to avoid cascade effect, and it is well adapted to IoT systems.

In [15], P Rajasekar and H. Mangalam proposed an area optimized and power efficient AES implementation in FPGA for IoT applications. This proposed approach addresses issues related to power consumption and area, to design and implement optimized functions for AES core. Simulation results show that the proposed architecture provides high security with low power, and reaches a maximum frequency of 190 MHz.

Authors in [16] propose a low area high-speed FPGA implementation of the AES architecture for cryptography applications. It introduces a modified positive polarity reed muller (MPPRM) architecture for the SubBytes and InvSubBytes transformations to reduce hardware requirements and improve speed.

In [17], S. J. H. Pirzada et al describe an optimization proposition of the AES algorithm for satellite applications. This proposed optimization is adapted to the space environment. Implementation results are presented in details, with a security analysis against recent attacks.

1.4. Conclusion

Advancements in cybersecurity demand the enhancement of security algorithms, resulting in more complex mathematical operations and arithmetic functions. This, in turn, requires greater energy, resources, and processing time; further emphasizing the challenges of constrained systems, and making the integration of security solutions into embedded systems a challenging task.

In summarize, it is noticeable when looking to proposed solution that there are two major axes of cryptography algorithms optimization for embedded systems. The first intervene on the algorithm itself, via proposing new block cipher algorithms, combining several methods or applying some adjustments on existing approaches. The second preserves the original algorithm without any adjustments, but focus on implementation technique, via introducing parallelism to the design, implement pipeline stages, or switching between implementation methods such as logic-based, memory-based or hybrid implementation. The distinction between these approaches lies in the validation phase. Where methods that involve modifying existing algorithms or proposing new ones require additional security analyses to demonstrate the robustness and reliability of their approach against various attacks and threats.

Chapter 2. Theoretical Background on Cryptography

2.1. Introduction

The written word stands as one of humanity's most transformative inventions, it enables the sharing and preservation of knowledge across generations. With the presence of the ability to communicate, came the equally the important need to protect sensitive information from unintended eyes. This dual necessity to share and to conceal, has driven the evolution of cryptography, a field dedicated to securing information through encoding and decoding techniques. From ancient ciphers to modern encryption algorithms, cryptography has played an important role in safeguarding communication, ensuring privacy, and enabling trust in an increasingly interconnected world. This chapter explores the foundational concepts and historical development of cryptography, providing the essential background for understanding its critical role in today's digital age.

2.2. Core Security Principles

The fundamental aspects of security provide a solid framework for protecting embedded systems and digital technologies. These principles ensure the protection of data and system integrity, enabling trusted operations in increasingly connected environments. By addressing confidentiality, integrity, authentication, they offer a comprehensive approach to mitigating security risks. Each system that deals with data transferring or data storage have this compromise of three characteristics to guarantee a good level of security [18]. The balance between these characteristics varies depending on the field of application of that system. Core Security principles and threats with corresponding solutions are illustrated in Fig. 2.1, as designed by Shoufan and Huss in [19]. The diagram shows the level of intervention for each security concern and the dedicated solution.

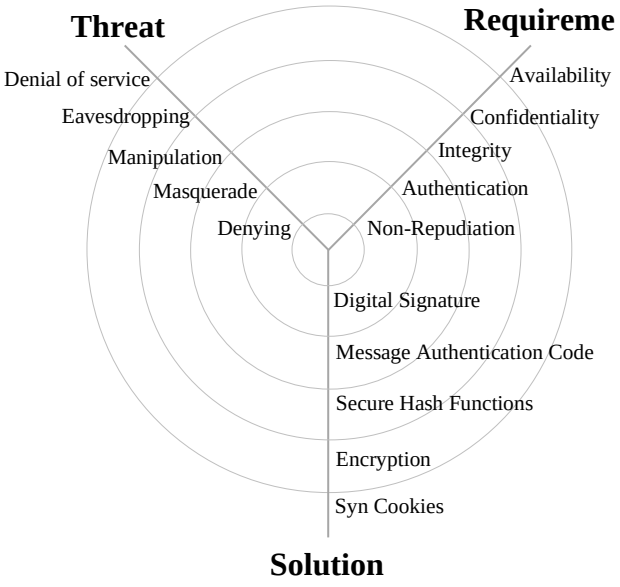


Fig. 2.1. Security Y-Diagram.

2.2.1. Confidentiality

Confidentiality is the principle of ensuring that sensitive information is accessible only to authorized individuals or systems. For embedded systems, this may involve encrypting data stored in memory, transmitted over networks, or processed within the device. Confidentiality is crucial in applications like IoT, where devices usually handle personal or private information. For example, a smart home system must ensure that communication between sensors and the central controller remains private to prevent unauthorized access.

Embedded systems achieve confidentiality through cryptographic methods such as symmetric encryption (e.g. AES) and asymmetric encryption (e.g. RSA). However, implementing these methods in resource-constrained devices can be challenging. Hardware accelerators, including FPGAs, are often used to optimize encryption processes while maintaining high performance [20].

2.2.2. Integrity

Integrity ensures that data remains unaltered during storage, processing, and transmission, barring any unauthorized changes. In embedded systems, this principle is critical for maintaining the reliability and accuracy of operations. For instance, in automotive systems, ensuring the integrity of control signals between the vehicle's sensors and actuators is essential for safety.

To safeguard integrity, embedded systems employ techniques like checksums, cryptographic hash functions, and digital signatures. These mechanisms verify that data has not been tampered with, whether due to accidental corruption or malicious attacks. Secure firmware updates are another critical aspect, ensuring that only authentic and verified software runs on the device [21].

2.2.3. Availability

Availability refers to ensuring that a system remains operational and accessible to authorized users whenever needed. This principle is especially vital for embedded systems in critical applications, such as medical devices or industrial control systems, where downtime could lead to severe consequences [22].

Threats to availability include denial-of-service (DoS) attacks, hardware failures, and resource exhaustion. Embedded systems mitigate these risks through redundancy, fault-tolerant designs, and real-time monitoring. For instance, a pacemaker must remain available under all circumstances to sustain a patient’s life. In such cases, embedded systems incorporate backup components and failover mechanisms to ensure uninterrupted operation.

2.2.4. Authentication

Authentication is the process of verifying the identity of users, devices, or systems before granting access to resources. In embedded systems, authentication is a cornerstone of secure communication and operation. For example, in IoT networks, devices must authenticate themselves to a central hub to establish trusted communication channels.

Authentication mechanisms include passwords, digital certificates, and biometric verification. Embedded systems often use lightweight authentication protocols, such as HMAC or public-key infrastructure (PKI), tailored to their resource constraints. Secure boot processes also rely on authentication to ensure that only trusted firmware is executed, protecting the system from malicious code [23].

2.3. Cryptography

The word cryptography comes from the Greek meaning “hidden writing”. Some of the earliest forms of secret writing come from ancient Greece as well, where military used a covered letters that are written in a wooden board on what is known as “steganography” which means a covered writing. This approach has some issues, the main one is that once the message is discovered, its contents will be easily revealed. This dilemma gave rise to “cryptography” which doesn't hide the existence of a message but instead hides its meaning.

Cryptography can be broken down into two subtypes: transposition and substitution. Transposition is when a document is rearranged creating an anagram, which is the earlier form of cryptography. An early historical example is once again from ancient Greece, where they used military method of communications, consists of wrapping the message around a wooden rod, in a way that it can’t be read unless it is wrapped again around an identical rod, otherwise the message would seem like jumble of letter. This method has its limitations, especially when dealing with large messages.

Because of the limitations in the transposition cryptography, the substitution cryptography was created. Substitution replaces the letters of a message rather than rearranging them. The most famous substitution cipher is the Caesar cipher, also known as the Caesar shift cipher as illustrated in Fig. 2.2, where it replaces any given letter with another letter from the alphabet (shift over a given offset) [24].

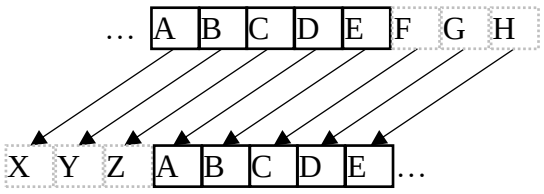
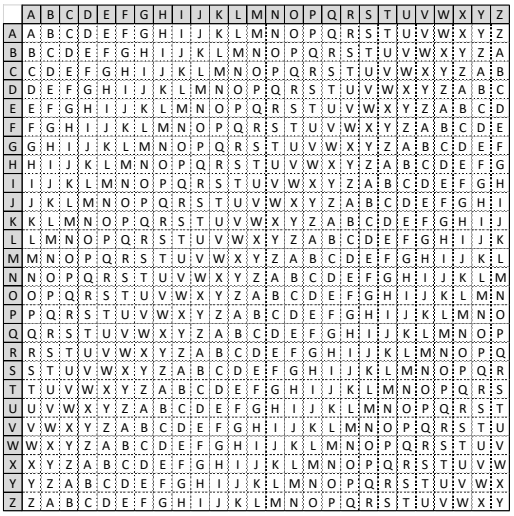


Fig. 2.2 Caesar shift cipher.

The only information that keeps the original message secure in the case of shift cipher is the shifting offset value, which is not enough to prevent unauthorized decryption in most cases. Cryptography progressed steadily over time, and hacking techniques were also advancing. Nowadays it includes combinations of complex operations such as characters substitution, data shifting, and mixing with keywords. Fig. 2.3, shows an example of an obsolete substitution cipher (Polyalphabetic Substitution Ciphers).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Fig. 2.3 Polyalphabetic Substitution Cipher.

The process of securing data using this set of operation is called Encryption, which is defined as the process of converting plaintext into secure, unreadable data (ciphertext) using specific algorithms.

2.4. Hashing and Hashing Algorithms

Hashing or hashing algorithms are mathematical formula, that transform messages into a deterministic fixed length representation of the original string. The result of a hashing algorithm is called the “digest”, and this digest is the representational string that represents the original message. A simple example of a hashing algorithm, is the summation of the alphabetic orders of the original message letters, an example is presented in Fig. 2.4.

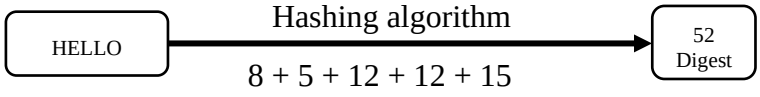


Fig. 2.4 Example of Hashing.

The purpose of a hashing algorithms, is to determine if the original message has changed since it was last hashed. If word “hello” in the previous example is changed to the word “cello”, the same hashing algorithm will result 47 instead of 52. We can easily then tell that the message has been changed by simply comparing the resulting digest. It is impossible to reverse engineer Hash unless the number of

characters of the original message is known. In this case a brute force attack can be applied to get all the possible combinations [25]. Some commonly used hash algorithms are presented in Table 2.1.

Table 2.1. Commonly used hash algorithms

Legacy	md5, sha1
Modern	SHA224, SHA256, SHA384, SHA512
Future	SHA3-224, SHA3-256, SHA3-384, SHA3-512

2.5. Symmetric cryptography

Symmetric cryptography is mathematical operations that are performed with a secret key, and either verified or undone with the same secret key. Symmetric cryptography includes three operations: encryption, Message Authentication Code (MAC), and pseudo random function [26].

2.5.1. Symmetric encryption

Encryption is a subset of cryptography and involves the process of converting readable data into unreadable. It is needed to protect important information from being accessed by unauthorized parties before transmitting or storing. Encryption uses a secret key, and a specific algorithm to process data, so that it appears as random and incomprehensible as possible.

The word symmetric signifies that the same key is used for encryption and decryption. The simplest example. The symmetric encryption process is shown in Fig. 2.5.

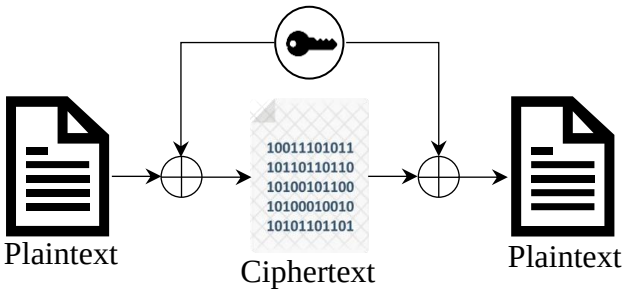


Fig. 2.5 Symmetric encryption.

There are many encryption algorithms, Table 2.2 lists some common algorithms. AES is one of the most famous algorithms.

Table 2.2. Some commonly used algorithms

Legacy	DES, RC4, 3DES
Modern	AES-128, AES-192, AES-256
Future	AES (is considered Quantum safe)

2.5.1.1. Data Encryption Standard algorithm (DES)

Data encryption standard (DES) is one of popular symmetric encryption algorithms. DES is developed between 1972-1977. It is a block cipher of 64-bit block, and a key length of 56-bit, which means that it processes 64-bits of data at a time. The full Encryption and Decryption process of the DES algorithm is illustrated in Fig. 2.6. The small size of DES key makes it fragile against brute force, a reason that made the algorithm obsolete, and no longer supported. DES is separated into two processes, the key generation known also by Key schedule, and the encryption process.

The key generation process, as the name suggests, involves creating subkeys for each round of the encryption process. [27]. The first step in this process is the key selection, where 56-bits are selected from the initial key through the Permuted Choice, which is a table that rearranges the key, and reduces it from 64 bits to 56-bits by discarding the rest (8-bit), which are used as parity bits. the selected bits are split after that into two parts: left and right (C_0 and D_0). Each part is shifted cyclically to the left by 1 or 2 positions depending on the round number. Both halves are compressed in the final operation to form a 48-bit key that serves the encryption process as a subkey. The operations described above are repeated for all the 16 rounds. Fig. 2.7 presents the Key generation process of the DES algorithm.

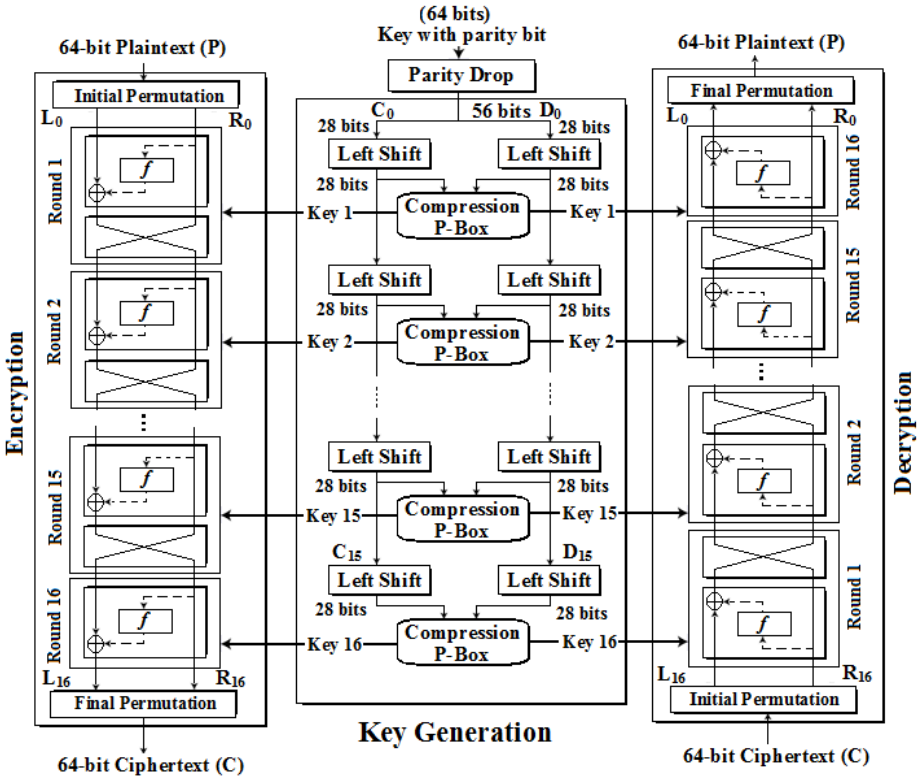


Fig. 2.6 DES Encryption algorithm.

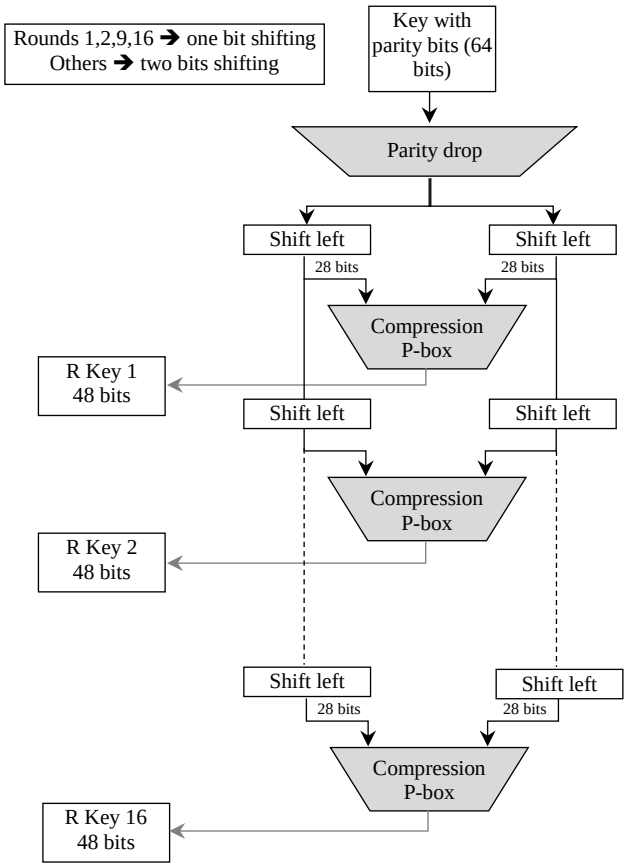


Fig. 2.7 Key generation process of DES algorithm.

The encryption process on the other hand is a set of operations applied to the plaintext to generate the ciphertext using the provided key. The plaintext passes by the Initial permutation (IP) to reorder all the bits according to a predefined table. The output of the IP is then divided into two parts (L_0 and R_0), the size of each is 32 bits. Each half goes through a Feistel Cipher function to perform the 16 rounds as follows:

Input: L_{i-1} and R_{i-1} (the left and right 32-bit halves from the previous round).

Process: For each round i (where $i \in [1, 16]$)

- Compute the new left half: $L_i = R_{i-1}$
- Compute the new right half: $R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$

Where:

- F is the Feistel function.
- K_i is the subkey from the key generation process

The Feistel function $F(R_{i-1}, K_i)$:

- Expansion (E): The expansion of the 32-bits to 48-bits.

$$E(R_{i-1}) = 48\ bit$$

- Key Mixing: The expended 48-bits block is XORed with the 48-bits subkey k_i

$$A = E(R_{i-1}) \oplus k_i$$

- Substitution (S-boxes): The 48-bit result A is divided into 8 groups of 6 bits each. Each group is processed by a corresponding S-box (Substitution box), which reduces the 6 bits to 4 bits. This results in a 32-bit output.

$$B = S_1(A_1) \parallel S_2(A_2) \parallel \cdots \parallel S_8(A_8)$$

- Permutation (P): The 32-bit output B is permuted using the Permutation Table (P-table)

$$\text{a}F(R_{i-1}, K_i) = P(B)$$

The Feistel function is represented in Fig. 2.8.

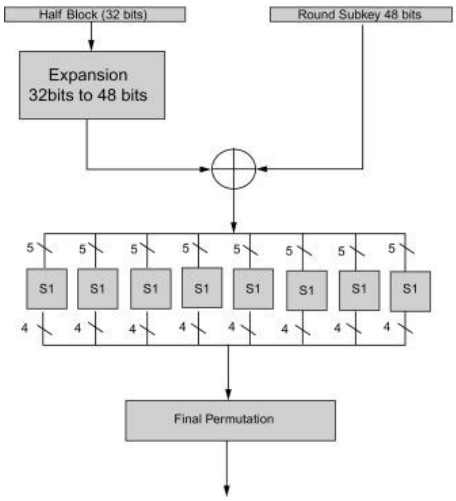


Fig. 2.8 The Feistel function in DES algorithm.

2.5.1.2. Triple DES Encryption algorithm

As its name indicates, the Triple DES algorithm is an extended version of the DES algorithm. 3DES uses three different keys to be able to perform its encryption process. It actually performs three separate passes through the data, and that's the signification of number 3 in its name. The first pass encrypts the plaintext with the first key, the second pass decrypts the result with the second key, and the third pass performs an encryption of result with the third key [28]. A presentation of the 3DES algorithm is shown in Fig. 2.9.

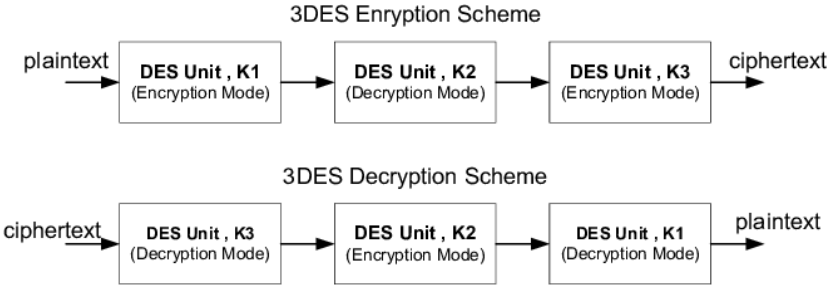


Fig. 2.9 Triple DES Algorithm.

2.5.1.3. RC4 Algorithm

RC4 is a stream cipher algorithm, it was created by Ron Rivest as a part of the original web stand, that is no longer used in today wireless networks. it was also a part of the SSL standard, but when replaced by TLS, RC4 was also replaced. One of the major problems in RC4 is the biased output, that means if the third byte of the original state is zero, and the second byte is not equal to two, then the second output byte is always zero. This little quirk that caused the deprecate of use of RC4 making it not a very common symmetric encryption today [29]. The RC4 algorithm is illustrated in Figure 9.

Here's a breakdown of the mathematics behind RC4

The Key Scheduling Algorithm (KSA) initializes the permutation of an array S, based on the input key:

- Initialization of *S:* The array *S* is initialized with values from 0 to 255:

  $S[i] = i, for\ i = 0,1, \dots ,255$

- Mixing *S* using the Key *K:*

  $j = 0$

  $For\ i = 0\ to\ 255:$

  $j = (j\ +\ S[i]\ +\ K\ [i\ mod\ len(k)])\ mod\ 256$

  Swap $[i] \leftrightarrow S[j]$

  Where:

  - K is (a variable-length key, typically 5 to 256 bytes).
  - *len(k)* is the length of *K*.

The result is a scrambled array *S* that depends on the input key.

Pseudo-Random Generation Algorithm (PRGA) uses the scrambled *S* array to generate a stream of pseudorandom bytes:

- Initialization

  $i = 0, j = 0$

- Generate pseudo-random byte: For each iteration

  - Increment *i* $\quad i = (i + 1)\ mod\ 256$
  - Update j $\quad j = (j + S[i])\ mod\ 256$
  - Swap $\quad S[i] \leftrightarrow S[j]$

- Generate pseudorandom byte (Output):

  $S[(S[i] + S[j])\ mod\ 256]$

RC4 encryption and decryption are performed by XORing each byte of the plaintext or ciphertext with the corresponding byte of the keystream:

- Encryption

  $C[n] = P[n] \oplus K[n]$

Where:

- $C[n]$ is the n^{th} byte of the ciphertext.
- $P[n]$ is the n^{th} byte of the plaintext.
- $K[n]$ is the n^{th} byte of the keystream.

- Decryption

$$P[n] = C[n] \oplus K[n]$$

Where:

- $P[n]$ is the n^{th} byte of the recovered plaintext.
- $K[n]$ is the n^{th} byte of the keystream.
- $C[n]$ is the n^{th} byte of the ciphertext.

RC4 is mathematically simple and efficient, it has several cryptographic weaknesses, such as non-random initial bytes in the keystream (susceptible to bias), and the vulnerability to key-reuse attacks (as in WEP). These weaknesses make RC4 unsuitable for modern cryptographic applications

2.5.1.4. The Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) was established as the standard encryption algorithm by the National Institute of Standards and Technology (NIST) in the United States. It was officially adopted as a Federal Information Processing Standard (FIPS 197) in November 2001. AES was selected as the successor to the Data Encryption Standard (DES) after a rigorous evaluation process, which included public submissions and extensive analysis of security, performance, and efficiency [30]. AES is a symmetric block cipher that encrypts 128-bits in a single pass, and it supports 128, 192 and 256-bits key size. It was developed by Joan Daemen and Vincent Rijmen, and is commonly used algorithm especially in wireless communication. The encryption and Decryption process of AES-128 is presented in Fig. 2.10. The AES-128 means using 128-bits key length [31]. As illustrated in Fig. 2.10, the AES consists of 10 rounds in case of 128 key length, but the number of rounds is 12 and 14 in case of 192-bits and 256-bits respectively. The round in AES represents an iteration, and each iteration consists of a set of operations: AddRoundKey, SubBytes, ShiftRows, and MixColumns. These operations are performed in every round except the final round, where the MixColumns operation is omitted. The AddRoundKey operation is a XOR of the output from the previous operation and the subkey of the active round. The subkeys are generated through the key expansion process, which derives a unique subkey for each round using a series of defined operations [31].

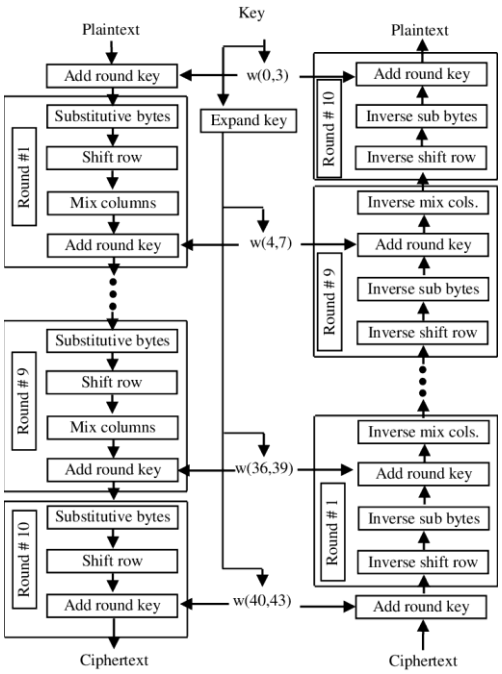


Fig. 2.10 AES-128 Algorithm process.

The AddRoundKeyis considered the most important operation despite its simplicity, because it hides the relationship between the ciphertext and the plaintext. Here is the description of each operation in AES algorithm:

a- The SubBytes

The Byte substitution operation replaces each byte in the State, a 4x4 byte matrix, with a corresponding byte from the S-Box. The S-Box is a pre-calculated substitution table, where each State byte is used to index and retrieve the new byte. In this context, the S-Box provides the nonlinear substitution for each byte of the State, and is represented by the following formula:

$$S: \{0,1\}^n \rightarrow \{0,1\}^m$$

Where n is number of input bits, and m is the number of output bits.

Each byte in the AES state is treated as an element of the finite field $GF(2^8)$, defined by the irreducible polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

For each non-zero-byte b in $GF(2^8)$, the multiplicative inverse b^{-1} in the field is calculated using the following formula:

$$b.b^{-1} \, mod \, m(x) = 1$$

Note: if $b=0$, its inverse is defined as 0.

The multiplicative inverse b^{-1} undergoes an affine transformation in $GF(2^8)$ to further increase non-linearity. The transformation is defined as

$$S(b) = A.b^{-1} \oplus c$$

Where:

- b^{-1} is treated as an 8-bit column vector.
- c is a constant binary vector:

$$c = [1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0]^T$$

- A is 8x8 binary matrix, defined by:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The SubBytes operation is illustrated in Fig. 2.11. the SubBytes operation relays on the s-box table to replace bytes.

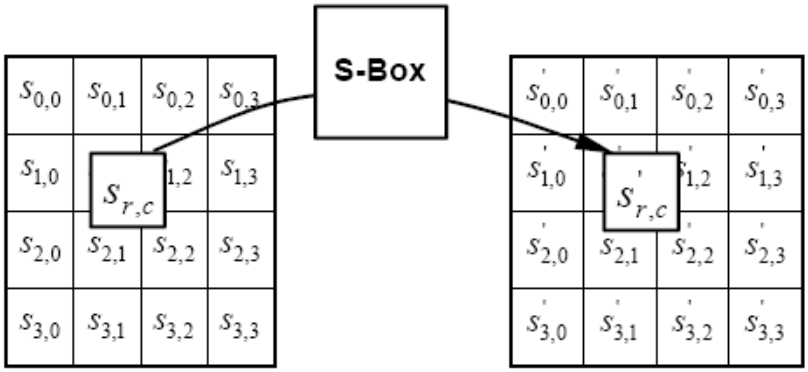


Fig. 2.11 SubBytes operation in AES algorithm.

b- ShiftRows

ShiftRows is permutation step that operates on the state matrix to introduce diffusion to the AES algorithm. It consists on re-arranging the bytes in each row of the state by shifting them to the left [32]. The state matrix is defined as follows:

$$S = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

Where each element $s_{i,j}$ represents a byte.

The first row remains unchanged, the second row is shifted left by 1 byte, the third row and the fourth row are shifted by 2 and 3 bytes respectively, the mathematical description of this operation is defined as follows:

$$s_{i,j} \rightarrow s_{i,(j-i)\ mod\ 4}$$

According to this definition, the state presented gives the following state after applying the ShiftRows operation:

$$S' = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

The ShiftRows operation ensures that the bytes in each column depend on bytes from multiple rows after the subsequent MixColumns step that will be described in the following section. It also prevents having independent columns during the encryption which render security weak.

c- MixColumns

This operation is a linear transformation of data applied to the state columns. It mixes the bytes of each column of the state matrix to provide diffusion across the ciphertext which ensures that changes in each bytes affects other bytes [32]. Each column is multiplied by a fixed 4x4 matrix in $GF(2^8)$ as follows:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Where:

- The coefficients 1,2 and 3 are constants in $GF(2^8)$
- The multiplication is performed modulo $x^8 + x^4 + x^3 + x + 1$

The multiplication and addition performed in $GF(2^8)$ using XOR for addition and finitr field multiplication for products. The output bytes are:

$$s'_{0,c} = (2.\ s_{0,c}) \oplus (3.s_{1,c}) \oplus (1.s_{2,c}) \oplus (1.s_{3,c})$$

$$s'_{1,c} = (1.\ s_{0,c}) \oplus (2.s_{1,c}) \oplus (3.s_{2,c}) \oplus (1.s_{3,c})$$

$$s'_{2,c} = (1.\ s_{0,c}) \oplus (1.s_{1,c}) \oplus (2.s_{2,c}) \oplus (3.s_{3,c})$$

$$s'_{3,c} = (3.\ s_{0,c}) \oplus (1.s_{1,c}) \oplus (1.s_{2,c}) \oplus (2.s_{3,c})$$

Multiplication in $GF(2^8)$:

$$1 . x = x$$

$$2 . x = left\ shift\ (x \ll 1), reduced\ modulo\ (x^8 + x^4 + x^3 + x + 1)$$

$$3 . x = (2 . x) \oplus x$$

The transformation matrix for decryption is:

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 19 & 14 \end{bmatrix}$$

The MixColumns operation is presented in Fig. 2.12.

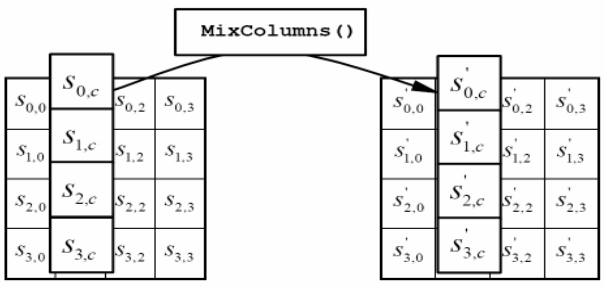


Fig. 2.12 MixColumns operation.

d- AddRoundKey

At this step the encryption process meets the Key generation process. The State in AddRoundKey is XORed with the round Subkey [32]. This operation is defined mathematically as follows:

$$s'_{i,j} = s_{i,j} \oplus k_{i,j}$$

Where:

- $s_{i,j}$: Byte in the state matrix.
- $k_{i,j}$: Corresponding byte in the round key matrix.
- $s'_{i,j}$: Resultant byte in the transformed state matrix.

The inputs to the AddRoundKey is:

The state $$S = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

The Key $$k = \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

The output of the AddRoundKey is:

The output $$S' = \begin{bmatrix} s_{0,0} \oplus k_{0,0} & s_{0,1} \oplus k_{0,1} & s_{0,2} \oplus k_{0,2} & s_{0,3} \oplus k_{0,3} \\ s_{1,0} \oplus k_{1,0} & s_{1,1} \oplus k_{1,1} & s_{1,2} \oplus k_{1,2} & s_{1,3} \oplus k_{1,3} \\ s_{2,0} \oplus k_{2,0} & s_{2,1} \oplus k_{2,1} & s_{2,2} \oplus k_{2,2} & s_{2,3} \oplus k_{2,3} \\ s_{3,0} \oplus k_{3,0} & s_{3,1} \oplus k_{3,1} & s_{3,2} \oplus k_{3,2} & s_{3,3} \oplus k_{3,3} \end{bmatrix}$$

e- Key Generation

The key generation is the process of generating subkeys for each round starting from the initial key. Number of rounds as mentioned before is 10, 12 or 14 depending on the length of the key, 128, 192 or 256 respectively. The initial key is divided into 4-byte words ($W[i]$). Number of words N_K is 16,

24 or 32 according the key length 128, 192 or 256. The process of key generation is presented in Fig. 2.13.

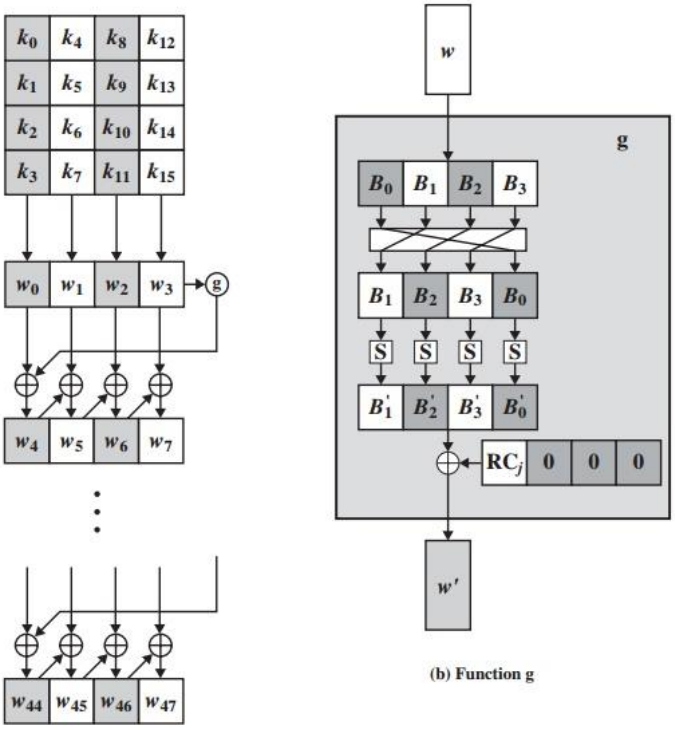


Figure 5.9 AES Key Expansion

Fig. 2.13 Key generation process in AES algorithm.

The total number of words in the expended key is:

$$N_b \; x \; (N_r + 1)$$

Where:

- $N_b = 4$: Number of columns in the state matrix
- N_r: Number of rounds (10, 12 or 14)

The round constant (RCON) is a table of constants used in the key expansion to add a nonlinearity to the output. $RCON\ [i]$ is divided from power of 2 in the finite field $GF(2^8)$:

$$RCON\ [1] = 0x01, RCON\ [2] = 0x02, RCON\ [3] = 0x04, \ldots$$

The steps to expend the key are:

For $i < N_k$:

The initial key word $W[0]\ to\ W[N_k - 1]$

are directly copied from the original key

For $i \geq N_k$:

$W[i] = W[i - N_k] \oplus T(W[i - 1])$

If $i\ mod\ N_k = 0$ Or $W[i] = W[i - N_k] \oplus W[i - 1]$

T function (For $i \; mod \; N_k = 0$):

Rotate $W[i-1]$: Circularly rotate the bytes by one position.

Substitute bytes from the S-box as described in the SubByte section.

XOR the first byte of the word with $RCON[i/N_K]$.

The following step is additional for the 256-bits key length:

If $i \; mod \; N_k = 4$, the $[i-1]$ word is substituted using S-box table.

2.5.2. Message authentication codes (MAC)

Message authentication codes is the concept of combining a message with a secret key before hashing. The purpose is to detect alterations of the message or the digest. While this may appear similar to the definition and purpose of hashing outlined in the preceding sections, it is, in fact, distinct. Hashing alone is not enough when sending packets across a communication channel, the reason is that the digest (result of hashing) is sent alongside the original message to enable the receiver to verify if the message has been altered or damaged. However, in a scenario where the message is intercepted and modified, with calculation of the new digest, the receiver would generate the same altered digest. As a result, the receiver would remain unaware that the message has been tampered. To address this issue, and given that the hash function ensures data integrity rather than message confidentiality, the Message authentication codes is the optimal solution for this issue, the process of MAC is illustrated in the Fig. 2.14.

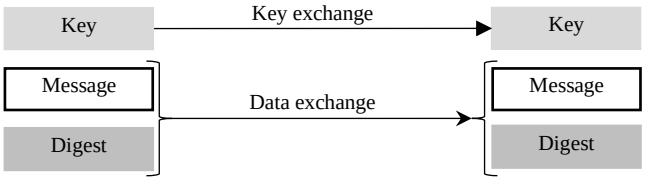


Fig. 2.14 Message Authentication Code.

An important characteristic on hashing is that the order is important, which means combining Message + Key is different that combining Key + Message. So, even if the key is the same, and the message is the same but the order is different, the receiver will not get the same digest, that's where a Hash-Based Message Authentication Code (HMAC) comes into play. HMAC is merely a standard way of combining a message and a key so if transmitter and receiver want to speak in a way that guarantees the integrity of the message, they not only have to agree upon using the concept of combining a message and a key, but they also have to agree on combining that message and a key in a specific way. HMAC is defined and detailed in RFC 2104 [33]. it includes all the instructions for exactly how you can combine a message and a key, in order to guarantee integrity of that message.

Table 2.3 lists some standard way of combining a message and a secret key to guarantee message integrity.

Table 2.3. Commonly used standards for message integrity

Legacy	-
Modern	HMAC, Poly1305
Future	GCM, CCM, AEAD Ciphers

2.5.3. Pseudo Random Functions (PRF)

Pseudo random function is similar as hashing function, except for output which is arbitrary length, meaning that the length is controllable. It's like a hashing algorithm that feeds back in on itself. The purpose of a PRF is to use a single key to generate an unlimited number of keys. In secure communication, a PRF enables the transmitter and receiver to derive multiple secret keys from a shared secret key, which can then be used for data encryption (confidentiality) and MAC (authentication and integrity). This approach eliminates the need for multiple key exchanges, which is a complex and risky operation. The diagram in Fig. 2.15 represent the PRF function.

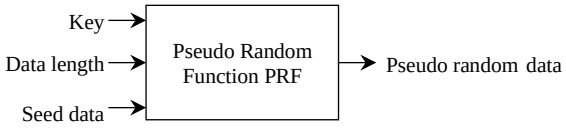


Fig. 2.15 Pseudo Random Function diagram.

If a PRF is running using only the key and data length input, the same output will be generated each time, because the starting information (seed data) hasn't changed, the seed data allows to differentiate the pseudo random data [34]. The PRF function is defined as follows:

$$F: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^m$$

Input Key	$k \in \{0,1\}$ (secret and fixed)
Seed data	$x \in \{0,1\}$
Output	Pseudo random $m \in \{0,1\}$

2.5.4. Key Derivation Function (KDF)

KDF is similar to a PRF but requires also an additional random data that is mixed into the initial data, and a Slowdown mechanism. The purpose of this extra data, which is known also by the name “Salt” is to guarantee a minimum level of entropy, while the Slowdown mechanism can be any one of a number of strategies, like running the KDF for a certain number of iterations instead of going straight from input to Output by recalculating the output several times through the same KDF, or through using a memory intensive match, or even prevent parallelization intestinally.

The purpose of a KDF is to make brute forcing the output infeasible. A good example of this is password storage, it is commonly known that it's better to use longer passwords, but there's probably a lot of people that are still using eight-character passwords or even less, that's where salt comes into play, salt adds some level of randomness to the password so that even in case of an insecure five-character password, salt might add another 40 characters to it, making it more secure . Additionally, passwords to online application are not stored in clear text on servers because it is insecure, but instead they are stored on a sort of hash of that password combined with the salt to enhance security. Furthermore, hashing and PRF are built for Speed, whereas KDFs are intentionally built to slow down the process, that's because modern cracking array can run through calculations of hashes on PRFs at a rate of billions or even trillions per second, but if each of those calculations is slightly slow down, it takes years to do the same number of guesses [36]. The KDF function is described as follows:

$$KDF(S, P) \rightarrow k$$

Where:

- S is the secrete input (e.g., a password)
- P is optional contextual information (e.g., salt)
- K is the derived key(s) of the desired length

2.6. Asymmetric cryptography

Asymmetric cryptography relies on a pair of keys to perform mathematical operations: one key is used to encrypt or sign data, while the other is used to decrypt or verify it. The first key, known as the private key, is kept secret and securely held by the owner. The second key, called the public key, is shared openly and can be used by others to verify signatures or encrypt messages intended for the private key holder [37]. The mathematics underlying asymmetric cryptography, which enables encryption with one key and decryption with another, is integrally complex. These mathematical operations involve multiple calculations and values within a single process. When referring to the private key or public key, it is important to note that these are not necessarily single values. Instead, a public key contains all the values required to perform an asymmetric cryptographic operation (such as encryption or signature verification), while a private key consists of the values needed to reverse or verify that operation (such as decryption or signing). In summary, the public and private keys represent the sets of values necessary to carry out their respective roles in asymmetric cryptography. Another important thing about asymmetric cryptography is that the keys used to perform an operation, and the keys used to verify that operation can sometimes be switched [44]. The keys used in asymmetric cryptography and their utilization is illustrated in Fig. 2.16.

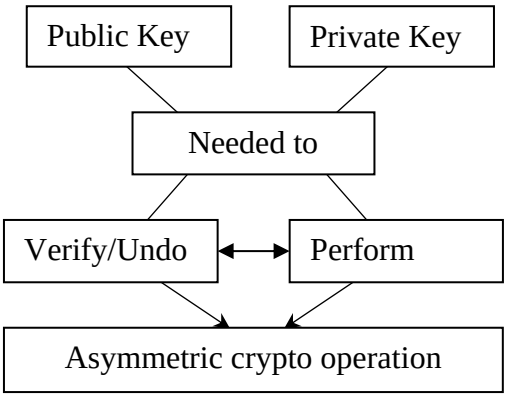


Fig. 2.16 Keys use in asymmetric cryptography.

Asymmetric cryptography allows to make three operations, encryption, signatures, and key exchange. The asymmetric cryptography process is illustrated in Fig. 2.17.

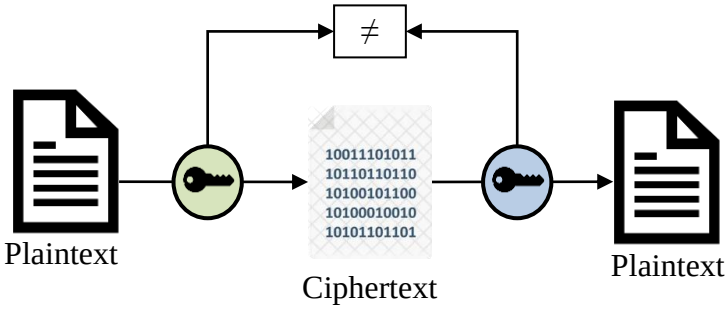


Fig. 2.17 Overall asymmetric algorithm process.

To perform these operations, we need specific algorithms that are listed in the Table 2.4.

Table 2.4. Asymmetric cryptography algorithms

RSA	Rivest, Shamir, Adleman
DSA	Digital Signature Algorithm
DH	Diffie-Hellman Key Exchange

RSA is the only algorithm that can do all the operations, other algorithms are designed to do specific operations. The following sections describes asymmetric cryptography operations.

2.6.1. Asymmetric encryption

The key difference between Symmetric and Asymmetric Encryption is the used keys, the first uses a single key to lock and decrypt data, while the second uses two different keys. The purpose of asymmetric encryption is to provide a cryptographic property known as confidentiality this is provided because only whoever has the private key can reverse the cipher text back into the original plaintext and therefore read the original data [45]. Encryption and decryption are a set of mathematical

operations defined to encrypt with one value “Public key” and decrypt with another value “Private key”.

2.6.1.1. Rivest, Shamir, Adleman (RSA) encryption algorithm

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm that ensures secure data transmission by leveraging the mathematical difficulty of factoring large integers. It uses a public key for encryption and a private key for decryption. The overall encryption and decryption algorithm is illustrated in Fig. 2.18. The mathematical definition of the RSA algorithm is given as follow:

Key setup:

Choosing two large prime numbers p and q Where $p \neq q$.

The security of RSA relies on the difficulty of factoring the product $n = p \times q$.

Compute Euler's totient function: $\phi(n) = (p-1)(q-1)$

Choose a public exponent e where $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$.

- e and (n) are coprime.
- e is often a small value like 3, 17, or 65537 (2^{16}+1) as they make encryption efficient.

Compute the private exponent $d \equiv e^{-1} \, mod \, \phi(n)$ which means $d \times e \equiv 1 \, (mod \, \phi(n))$

Resulting keys:

- Public Key: (e, n) used for encryption or verifying signatures.
- Private Key: (d, n) used for encryption or verifying signatures.

Encryption:

Given a plaintext message M represented as an integer where $0 \leq M < n$.

And a public key (e, n) where e is the public exponent and n is the modulus.

The ciphertext C is composed as $C \equiv M^e \, (mod \, n)$

Decryption:

Given a ciphertext C.

And a private key (d, n), where d is the private exponent and n is the modulus.

The original plaintext M is recovered as $M = C^d \, mod \, n$

The correctness of RSA relies on the mathematical relationship between the public exponent e, the private exponent d and Euler's totient function $\phi(n)$.

This works because: $M \equiv (M^e)^d \, mod \, n \equiv M^{(e.d)} \, mod \, n$

During the key generation, d is chosen such that $e.d \equiv 1 \, mod \, \phi(n)$

Which means $e.d = k.\phi(n) + 1$ for some integer k.

Starting from the ciphertext $C \equiv M^e \, (mod \, n)$ raising C to the power of d gives:

$$C^d = (M^e)^d \, (mod \, n)$$

Substitute $e.d = k.\phi(n) + 1$:

$$C^d \equiv M^{k.\phi(n)+1} \ (mod\ n)$$

Using Euler's theorem, which states that if $\gcd(M, n) = 1$, then:

$$M^{\phi(n)} \equiv 1^k \equiv 1 \ (mod\ n)$$

Therefore:

$$M^{k.\phi(n)} \equiv 1^k \equiv 1 \ (mod\ n)$$

Substitution back, Thus, recovering the original plaintext *M*:

$$C^d \equiv M^{k.\phi(n)+1} \equiv M.M^{k.\phi(n)} \equiv M.1 \equiv M \ (mod\ n)$$

Breaking RSA requires factoring n into p and q, which is computationally infeasible for large primes. Modern implementations use n with at least 2048 bits for strong security [45]. The RSA algorithm flowchart is shown in Fig. 2.18.

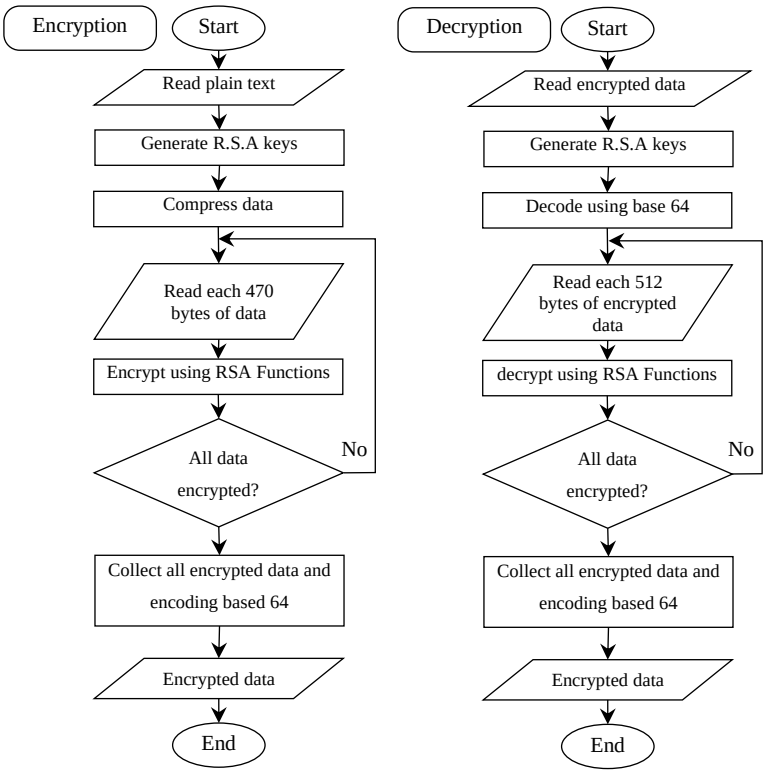


Fig. 2.18 RSA algorithm flowchart.

2.6.2. Signatures

Digital signatures provide two important cryptographic principles: integrity, and authentication. signatures are one of the operations that exist within asymmetric cryptography. It is an operation that guarantees that data has not changed since it was signed, so if some sort of message run through a signature operation, the output can be used to verify that this data has not changed since the signature was created. Signatures are an asymmetric crypto operation, which means two different keys are involved: one key to do the signature and the other key to do the verification [41]. The private key in

signatures is the one that is used to create the signature, while the public key is used to verify the signature.

A practical example to understand digital signatures is the passport or the driving license, where the owner is the only person that should be able to sign the document, but anybody can verify the signature on the passport or driving license. Similarly, the only person that should be able to create a digital signature is the one that has the private key, which in theory has never been shared with anybody else, but the public key allows the verification of a digital signature by anyone.

Not only messages can be signed, but nearly anything can be signed, whether it is a document, an email, a file, a software, or even certificates. However, and since signatures are an asymmetric crypto operation, the math is somewhat computationally expensive. Therefore it wouldn't be used for every file or every message between two people, but occasionally for selected files it might make sense to use it [42].

The purpose of signatures is providing both integrity and authentication to data, integrity because if anything changes in the message this signature -which applies only to this message- will no longer verify with the new changed-message, and authentication because the only person that can create this signature is whoever has the private key. Another fact is, when verifying a signature with a particular public key, the only person that could have created that signature is whoever had the matching private key, that's what gives authentication.

There are two cryptographic operations that provide integrity and authentication, MAC as described in symmetric cryptography section, and Signatures. The major difference between signatures and MAC is that signatures are an asymmetric operation and MAC are a symmetric operation, which means MAC are more efficient to calculate on bulk data, whereas signatures are more limited to smaller data sets. When doing something with one value and then verifying it with another value, as in asymmetric operations, it understandably going to be more complicated than doing and undoing something with the same value. This fact brings us to the second major difference between signatures and MAC, signatures can be verified by anyone, the public key is shared publicly, which means anybody can attain the public key and therefore verify the signature, whereas with MACs, only the peer that has the secret key can verify the MAC and ensure the integrity and authentication of data.

2.6.2.1. Signatures using RSA algorithm

The process of signature creation is shown in Fig. 2.19. It all starts with some sort of data that needs to be signed, this data can be of any kind, file, message, or any other data. To create a signature on this data using RSA algorithm we first need to run it through a hashing algorithm which is going to create a digest. Then that digest will be encrypted using the RSA private key to have the final signature. The hashing step is very important since the RSA signatures is computationally expensive, we can't expect to do the math on large amount of data, but if it is run through a hashing algorithm,

any file of any size is compressed into a smaller representational value, and then it's much more feasible to do the complex math of RSA signatures on that smaller value. Once created, the signature can then be attached to the original data before sending that it across the wire or simply including it in the metadata of the file itself on a hard drive [41].

Fig. 2.19 Signature creation using RSA.

The signature verification process consists on separating data from signature, the data is then run through the same hashing algorithm that was used when creating the signature to generate a digest. The signature on the other hand is decrypted using RSA public key that matches the private key used in the encryption. data is considered unchanged if, and only if the decrypted signature matches the value of calculated digest, as presented in Fig. 2.20.

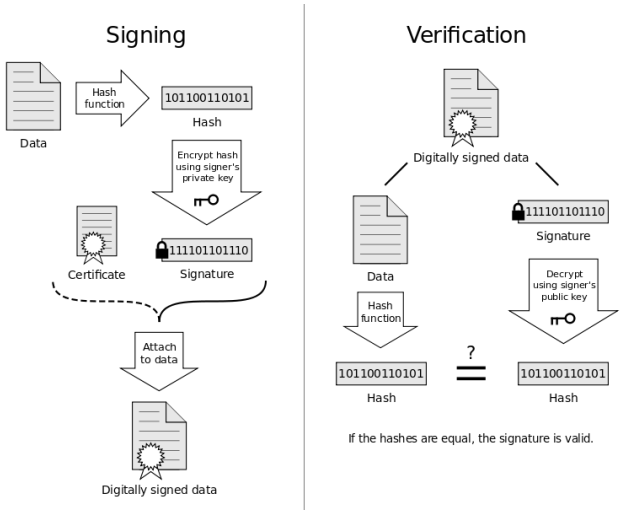


Fig. 2.20 RSA signature creation and verification process.

2.6.2.2. Signatures using DSA algorithm

The Digital Signature Algorithm (DSA) is a widely adopted public-key cryptographic algorithm specifically designed for generating digital signatures. It is formally defined in FIPS 186, a standard published by the National Institute of Standards and Technology (NIST). DSA ensures the integrity and authenticity of a message by enabling the recipient to verify that the message originated from the legitimate sender and has not been altered. Unlike RSA, which supports encryption, decryption, and key exchange, DSA is designed for digital signatures and cannot be used for other purposes. [43]. It involves two formulas, the first generates the signature, and the second verifies it. the formula for Signature generation requires the original data as input along with the private key. The DSA signature generation and verification is illustrated in Fig. 2.21.

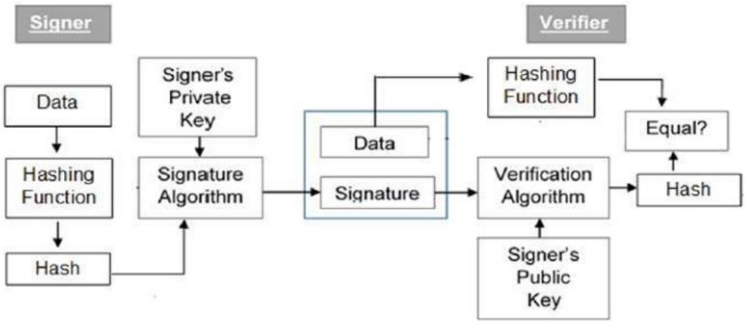


Fig. 2.21 DSA Signature creation and verification.

Verification process uses a combination of the received data and signature with the public key, to get one or a zero indicating true or false whether the signature checked out or not. The difference between DSA and RSA is the absence of encryption and decryption in DSA.

2.6.3. Key exchange

Key exchange addresses a critical challenge in data cryptography: securely sharing the key between the encryption and decryption parties (the transmitter and receiver). For example, when data is shared over the internet, two essential elements are required to ensure secure transmission. The first is confidentiality, which guarantees that the transmitted information is only accessible to the intended users. The second is integrity, which ensures that the data cannot be altered or tampered with during transit. Confidentiality can be achieved by using symmetric encryption and this is going to require a secret key, and Integrity can be guaranteed by using a message authentication code, which requires also a secret key. The issue here is how to get these secret keys across the internet to the other side, it can't be just sent because anybody is listening, will then get a copy of those keys and will therefore be able to intercept and read, or even change anything sent, that is what is known as the key exchange problem.

Key exchange enables the transmitter and receiver to establish a shared secret key over an unsecured medium, such as the internet. The key exchange itself will have both of these endpoints exchanging certain pieces of information with each other, and the result is known as a "shared secret", which is a string of ones and zeros known only by the intended parties, it is used as the seed value from which to generate as many symmetric secret keys as needed using a pseudo random function (PRF) [44].

2.6.3.1. RSA key exchange

The goal of key exchanges is to establish a mutual shared secret on either side of the wire, and RSA is well adapted to this operation. Doing a key exchange with RSA simply takes advantage of the algorithm's ability to encrypt and decrypt, to perform the operation efficiently. One side of communication (receiver), must have an RSA asymmetric key pair (public and private). The other

side (transmitter) randomly generates a seed (ones and zeros), and encrypt it using the public key to generate the Ciphertext. this Ciphertext is sent across the communication channel, even if it is intercepted it needs to be decrypted using the private key, so it is safe. The receiver uses its private key to decrypt that seed value, and the result will be the exact identical value that transmitter randomly generated in the first step, and now both parties have the same mutual shared secret, which can be used to generate any amount of symmetric secret keys to do symmetric encryption. Since only the transmitter and the receiver have the same seed value, they're the only users that could have generated the exact set of secret keys, and therefore anything shared between them that is protected by these keys is only readable by them. so that is how RSA algorithm is used for key exchange [45].

2.7. Block cipher operation modes

The block cipher processes data in fixed-length blocks. If more data needs to be encrypted, adjustments are required to handle the additional input, which is why different operation modes exist. These modes vary based on how they manage the relationship between blocks during encryption.

2.7.1. Electronic Code Book (ECB)

Electronic code book is a mode where each block of data is processed separately, and then concatenated after encryption. If the same input plaintext is encrypted using this mode repeatedly, the ciphertext is going always to be the same. ECB is considered weak against some known attacks. Fig. 2.22 illustrates the ECB mode [46].

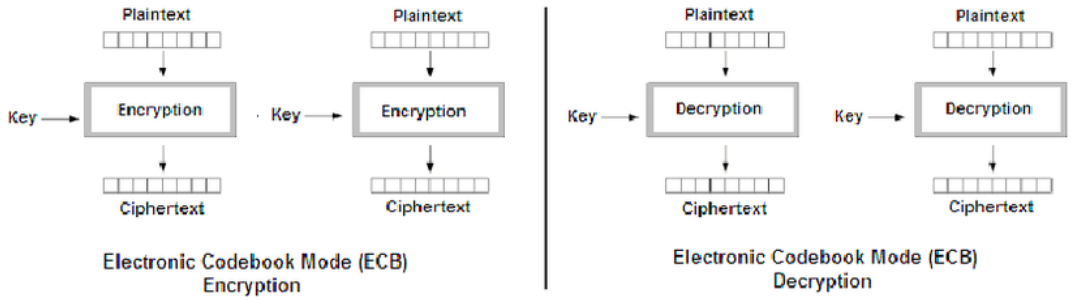


Fig. 2.22 Electronic Code Book mode (ECB).

2.7.2. Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) differs from ECB. In CBC, the first plaintext block is XORed with an initialization vector (IV) before encryption. The resulting ciphertext is then used as the IV for the next block. This process continues, with each ciphertext block becoming the IV for the subsequent block, until the end of the message. This chaining mechanism introduces greater randomness and obscures the relationship between the input and output, enhancing security [46]. Fig. 2.23 illustrates the CBC encryption mode.

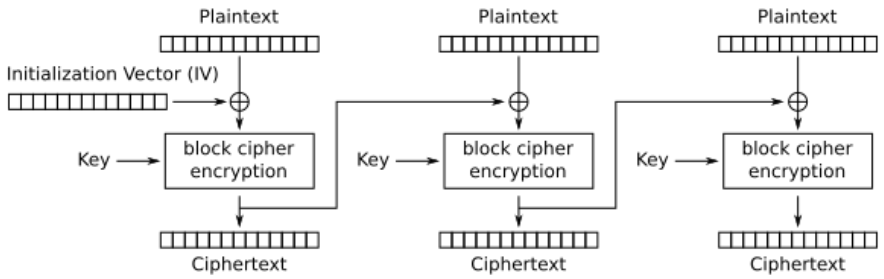


Fig. 2.23 Cipher Block Chaining mode (CBC).

The encryption of the Cipher Block Chaining formula is presented in the following formula

$$C_i = E_k(P_i \oplus C_{i-1})$$

Where $C_0 = IV$ (Initial vector)

And for the decryption, the formula is presented as follows:

$$P_i = D_k(C_i) \oplus C_{i-1}$$

Where $C_0 = IV$ (Initial vector)

The advantage of this chaining mode is the randomness introduced to the vectors, which obscures patterns in the plaintext even when blocks are identical. However, the process is sequential, making it less suitable for parallel processing.

2.7.3. Cipher FeedBack (CFB)

Unlike ECB and CBC, CFB does not directly encrypt the plaintext, but instead, A first initialization vector is encrypted with an algorithm such as the advanced encryption standard after being randomly generated or created, and then XORed with the plaintext. The result is considered to be the first ciphertext, and is also driven to be used as the initialization vector for the next block, and so on. This process goes on forever as presented in the block diagram of Fig. 2.24. The important thing about Cipher FeedBack is that it's a self-synchronizing stream [46].

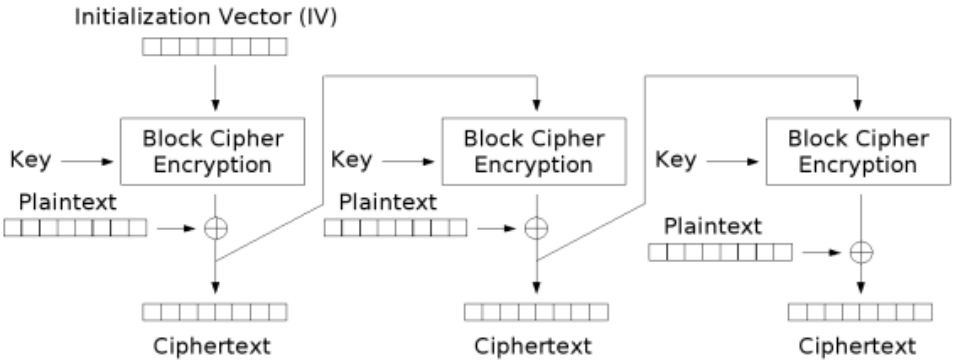


Fig. 2.24 Cipher FeedBack mode (CFB).

The encryption formula of the CFB is defined as follows:

$$C_i = P_i \oplus E_k(C_{i-1})$$

Where:

- C_i = ciphertext block at position *i*
- P_i = plaintext block at position *i*
- $C_0 = IV$ (Initial vector)
- C_{i-1} =ciphertext block at position *i-1*
- E_k = encryption function using key *k*
- $i \geq 1$

The decryption is defined by the formula:

$$P_i = C_i \oplus E_k(C_{i-1})$$

Where:

- C_i = ciphertext block at position *i*
- P_i = plaintext block at position *i*
- $C_0 = IV$ (Initial vector)
- E_k = encryption function using key *k*

One of the disadvantages of this mode is the sequential nature of the algorithm, which limits the parallelization.

2.7.4. Output feedback (OFB)

Output Feedback is a block cipher mode of operation that converts a block cipher into a synchronous stream cipher. The block cipher encrypts an Initialization Vector (IV), and the result is used as a keystream. The keystream is XORed with the plaintext to produce the ciphertext [46]. Unlike other modes, neither the plaintext nor the ciphertext are fed back into the encryption process. The blocks diagram of the OFB is presented in Fig. 2.25.

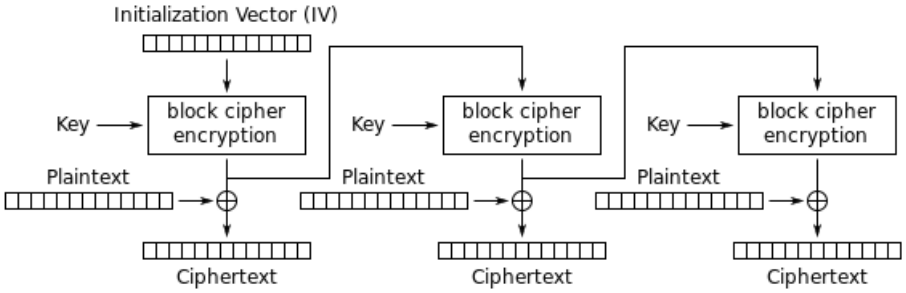


Fig. 2.25 Output feedback mode (OFB).

Formula for encryption in OFB mode:

$$C_i = P_i \oplus O_i$$

Where:

- C_i = ciphertext block at position *i*
- P_i = plaintext block at position *i*
- *Oi* = keystream block at position *i* generated as:
  - $O_i = E_k(O_{i-1})$
- $O_0 = IV$ (Initial vector)
- E_k = encryption function using key *k*
- $i \geq 1$

The decryption mode is presented by the formula:

$$C_i = P_i \oplus O_i$$

Where:

- C_i = ciphertext block at position *i*
- P_i = plaintext block at position *i*
- *Oi* = keystream block at position *i* generated as:
  - $O_i = E_k(O_{i-1})$
- $O_0 = IV$ (Initial vector)
- E_k = encryption function using key *k*
- $i \geq 1$

OFB mode is ideal for scenarios where error tolerance is crucial, such as satellite communications and real-time audio or video streams.

2.7.5. Counter mode (CTR)

The counter mode (CTR) is another common type of block cipher. Counter mode uses an incremental counter to be able to add randomization to the encryption process. The incremental counter starts with an initial value, for the first encryption operation, and is incremented for next encryption operations. The difference of CTR compared to other modes, is that the plaintext is not encrypted, instead, the counter value is encrypted and then XORed with the plaintext to create the ciphertext. Another thing about CTR, is that the ciphertext is not mixed with the following plaintext, alternatively, the counter is incremented and mixed to the plaintext to repeat the same process. These modes of operation can not only provide encryption but can also provide authentication. A good example is the Galois Counter Mode (GCM), which combines counter mode with Galois authentication. This mode provides a way to not only encrypt data very quickly but make sure that we can authenticate where the data came from. This is commonly used in wireless connectivity. The CTR mode is illustrated in Fig. 2.26.

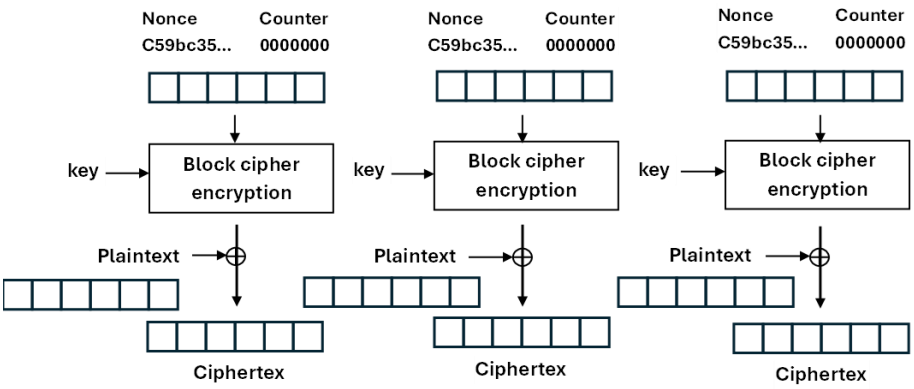


Fig. 2.26 Counter mode (CTR)

The encryption formula of the CTR mode is:

$$C_i = P_i \oplus E_k(T_i)$$

Where:

- C_i = ciphertext block at position i
- P_i = plaintext block at position i
- T_i = counter value at position i
- E_k = encryption function using key k

The encryption formula of the CTR mode is:

$$P_i = C_i \oplus E_k(T_i)$$

Where:

- C_i = ciphertext block at position i
- P_i = plaintext block at position i
- T_i = counter value at position i
- E_k = encryption function using key k
- $i \geq 1$

The counter value T_i is defined as follows:

$$T_i = \text{Nonce} \parallel \text{Counter}$$

Where:

- Nonce: A unique value used once, usually random
- Counter: A block-specific value that increments for each block

Chapter 3. Hardware design optimization strategies

Field-Programmable Gate Arrays (FPGA) offer flexibility in electronic design, which provides the ability to achieve an optimal performance. This chapter provides some common strategies for designs optimization, focusing on resource utilization and timing efficiency. Design optimization aims to reduce logic elements, memory, and power consumption. Timing optimization on the other hand, ensures meeting circuit requirements, via using several techniques, such as pipelining, and clock domain management. The following sections highlight the impact of these strategies on design efficiency, and maximizing FPGA performance, to enable a faster, more reliable, and resource-efficient implementations for a wide range of applications.

3.1. Timing Constraints

One of the first things to do when creating an FPGA application, is specifying the timing constraints of the design. The synthesis tool then tries to generate the designed circuit that meets those timing constraints, using something called static timing analysis. if any of the constraints aren't met after static timing analysis, this means timing optimization must be performed until all constraints are met also known as timing closure, which is a difficult operation that requires a long development time. This section explains the background material and underlying challenges followed by practical acts to perform a timing optimization [47].

The definition of timing constraints depends on different factors and situations. The first situation is when the FPGA itself has a fixed clock frequency; in this case, the timing constraint is based on that specific clock frequency. Essentially, the constraints in this case tell the synthesis tool that the design, after placement and routing, should run at least at this specific clock frequency. The second situation is working with an application that has real-time requirements, such as a signal processing circuit that must produce outputs every 1000 cycles while processing 44.1 kHz audio. This means the design needs to have a clock frequency 1000 times faster than the processing requirements (44.1 MHz), which corresponds to the timing constraint. Another situation is when having a design with specific bandwidth requirements, for example a circuit with a 32-bit bus (4 bytes) and requires 1.6 GB/s of input bandwidth to achieve the desired functionality. the clock in this case must be at least 400 MHz in order to achieve this amount of bandwidth, which corresponds to the timing constraint. The fourth situation is a common case where the objective is maximizing performance of design as much as possible. In this case there is no specific timing constraints, so the development process starts with an aggressive constraint, and then a timing optimization is performed, until that constraint is met, or the process is repeated until the clock frequency can’t be enhanced anymore. alternatively, it's also reasonably common to pick an aggressive constraint that is not expected to be met actually, and then fall back to whatever clock frequency is actually reported as safe by the timing analyzer.

Before proceeding to the global timing constraints and the delay paths, it's very important to have a background on the possible path endpoints that are taken in consideration during synthesis of a design. While global timing constraints are very simple, understanding possible path endpoints is useful when learning about path-specific timing constraints, to properly define the design timing objectives to the implementation tools. Simply put, path endpoints are IO pads, and synchronous elements. Synchronous elements include flip-flops, latches, RAMs, DSP slices, and shift register. Path endpoints do not include LUTs, nets, or any other purely asynchronous elements, so basically anything without a clock port. To be noticed that, while the LUT itself is not an endpoint, but reconfiguring it as a RAM or shift register makes it becomes synchronous, and thus, it can be a path endpoint.

3.2. Clock Frequency

The first thing to take in consideration before performing any kind of timing optimization, is how clock frequencies and periods are determined. For any set of connected flip-flops for example, data must arrive at each flip-flop before the next clock cycle. This means that the clock period essentially acts as a deadline for the delay that can occur between flip flops, this clock period (T_{clk}) is called the deadline ($T_{Deadline}$) and the delay between flip flops ($T_{FF\text{-}to\text{-}FF}$). So, what should be ensured during timing optimization is that the delay between FF is less than the deadline, $T_{FF-to-FF} \leq T_{Deadline}$ where $T_{Deadline} = T_{Clk}$. Fig. 3.1 shows a simple example of this situation. If $T_{FF-to-FF} = 5\ ns$ in Fig. 3.1, then $T_{Clk} \geq 5ns$.

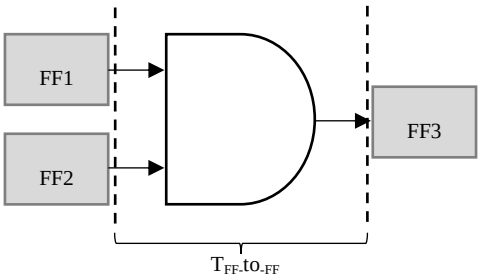


Fig. 3.1 FF to FF timing.

The definition of $T_{Deadline}$ is updated according to the design size, and the parameters involved accordingly, which is discussed in the next sections.

Delays consist of the summation of two different components, cell delays (T_C), and interconnect delays (T_{IC}). A cell is a generic term used to refer to any non-interconnect FPGA resource, like Configurable Logic Blocks (CLBs,) lookup tables, Random Access Memories (RAM), Digital Signal Processing (DSP), etc. Even a simple flip flop contributes to cell delays, where they don't immediately output a value after a rising clock edge, instead there is a small amount of time often referred to as a

clk-to-Q delay, that passes before the output appears. Interconnect delays are the delays of all the connections between cells, which includes the FPGA routing tracks, the connection boxes, the switch boxes, basically all the interconnect components of the FPGA [47]. An example of delays is illustrated in Fig. 3.2.

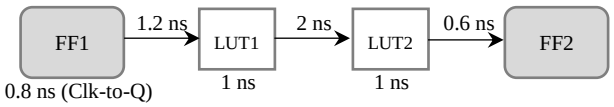


Fig. 3.2 Interconnect delay, and FF delay.

Cell delays and interconnect delays in Fig. 3.2 are calculated as follows:

$T_C = 0.8ns + 1ns + 1ns = 2.8ns$

$T_{IC} = 1.2ns + 2ns + 0.6ns = 3.8ns$

$T_{FF\text{-}to\text{-}FF} = T_C + T_{IC} = 2.8ns + 3.8ns = 6.6ns$

$Frequency = 1/ T_{FF\text{-}to\text{-}FF} = 1/ 6.6ns = 151\ MHz$

One of the common situations, is having multiple paths between the same set of flip flops with different delays. Even when using the same set of resources, result may give different delays due to differences in the routing of each path. The critical path in this case is the one with the maximum delay. Fig. 3.3 illustrates an example of this situation where there are two paths between the source and destination flip flops.

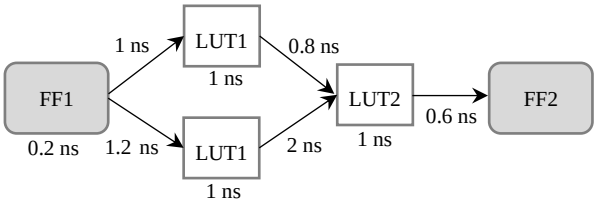


Fig. 3.3 Multiple paths for the same destination and source.

The cell delays of each path in Fig. 3.3, are identical since both pass through two lookup tables; however, the interconnect delays of each path are different.

T_C of both paths = 0.2 ns + 1 ns + 1 ns = 2.8 ns

T_{IC} of bottom path = 1.2 ns + 2 ns + 0.6 ns = 3.8 ns

Max $T_{FF\text{-}to\text{-}FF} = T_C + T_{IC}$ = 2.8 ns + 3.8 ns = 6.6 ns

$Frequency = 1/ T_{FF\text{-}to\text{-}FF} = 1/ 6.6\ ns = 151\ MHz$

If the top path is considered to calculate the frequency, result would be higher, but it does not work, because the bottom path would not yet have its data available by the next rising clock edge.

3.3. Maximum frequency and timing optimization

Timing optimization is ultimately looking for the maximum clock frequency that can be used for the entire design. This frequency is often referred to as Fmax, and is determined by the timing analyzer through performing the analysis to identify the longest FF-to-FF delay in the design, which is referred to as the critical path. Fig. 3.4 shows an example of circuit with three pairs of FF, with the propagation delays between each pair. The considered critical path for this example is the path between FF 3 and 4, which has the longest FF-to-FF propagation delay in the entire design. this path requires a Clk ≥ 10ns, which signifies a maximum frequency of a 100 MHz. Even though the other FF pairs can run faster than 100 MHz, but they have to synchronize with the critical path at a lower frequency.

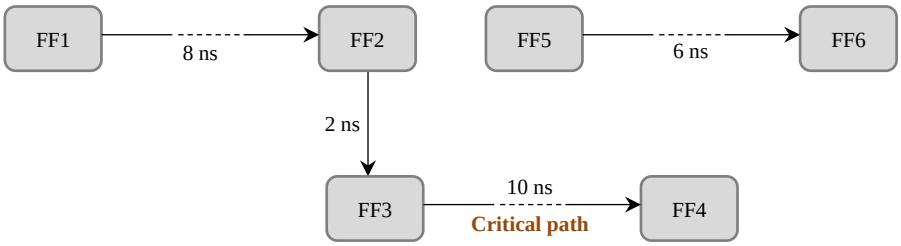


Fig. 3.4 Critical path of a design.

3.4. Cells-Clock synchronization

3.4.1. Setup times

Setup time (T_{setup}) is a small window of time before a rising clock edge, where the input to the cell (FF) cannot change. if the inputs change during the setup window, the output of the FF becomes metastable, which is an issue. So, data should arrive to the destination FF not only before the next clock edge, but before the setup window of the next clock edge [47] [48]. Fig. 3.5 illustrates Setup time window.

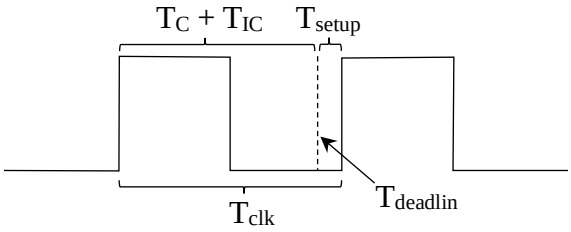


Fig. 3.5 Setup time.

From the previous definition and, the new definition of deadline time ($T_{deadline}$) as illustrated in Fig. 3.5, is the T_{setup} event instead the rising clk edge. Basically, the summation of the cell delays and interconnect delays along each path, must be less than or equal to the clock period minus the setup time.

$$T_C + T_{ic} \leq T_{clk} - T_{Setup}$$

If there exist paths that violates this condition, it's called a setup violation, and must be optimized to ensure correct functionality.

$$T_C + T_{ic} > T_{clk} - T_{Setup} \text{ (Setup violation).}$$

3.4.2. Clock Skew

In most timing diagrams, the clock signals instantly propagate to all destinations, however, this is not possible physically, in addition, clocks are usually extremely high fanout signals, which makes it impossible to deliver the clock to every cell at the same time, so the inevitable differences in clock arrival times between cells is known as the clock skew [49]. This skew is important because it affects the available time before a setup violation as shown in the Fig. 3.6.

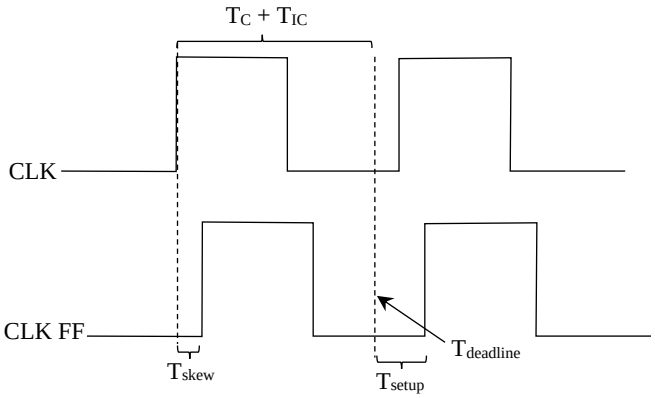


Fig. 3.6 clock skew in FPGA circuit.

A positive skew increases deadline, which can help avoiding setup violations. Positive skew means that the destination FF receives its clock signal after the source FF, so in this case the allowable time for the FF-to-FF delay, which is the sum of this cell and interconnect delays, is actually increased as the skew shifts the rising clock edge of the destination clock like shown in Fig. 3.6 [50]. When taking skew into consideration the summation of the cell delays and interconnect delays verifies the following equation.

$$T_C + T_{IC} > T_{clk} + T_{Skew} - T_{Setup}$$

This is still the same goal of ensuring that the delay between FF is less than or equal to the deadline, with an update of the deadline to include both the setup time and the clock skew. While positive skew allows avoiding setup violations because it increases the maximum allowable delay between FF, a negative skew has the opposite effect which can be counter-intuitive when performing timing optimization.

3.4.3. Setup Slack

One important parameter during timing optimization is setup slack (S_{setup}), it is defined by the difference in time between the setup deadline and the time where data arrives to the destination FF. When the setup slack is negative for a given path, it means that the path has a setup violation, and must be optimized. A positive slack on the other hand, means that the path meets timing constraints [47]. The setup slack is illustrated in Fig. 3.7, and is represented by the following equation:

$$S_{Setup} = \left(T_{clk} + T_{Skew} - T_{Setup}\right) - \left(T_C - T_{IC}\right) = T_{Deadline} - T_{FF-to-FF}$$

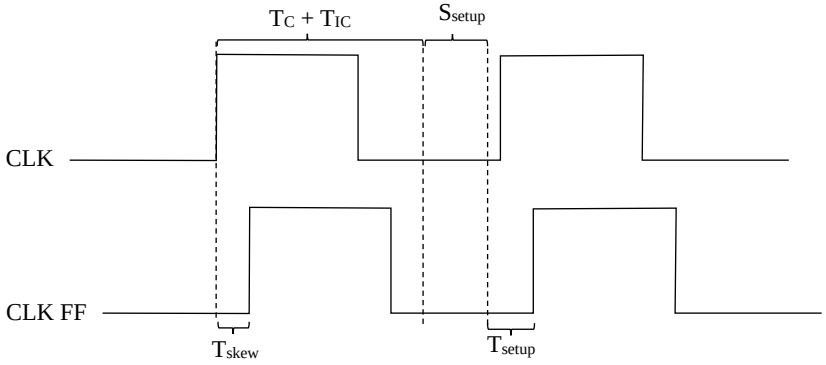


Fig. 3.7 Setup slack in FPGA design.

3.4.4. Hold Times

Hold times are the opposite of setup times; an FF hold time is a small window after the rising clock edge, where the input to the FF must not change, otherwise the output is meta-stable [51]. This means that new data must not arrive at the FF during this hold window.

$$T_C - T_{IC} \geq T_{Hold} + T_{skew}$$

Mathematically, the sum of the cell and interconnect delays must exceed both the clock skew and hold time as shown in Fig. 3.8.

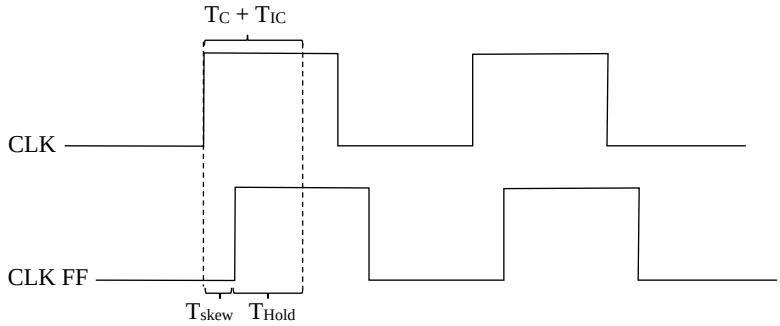


Fig. 3.8 Hold time in FPGA circuit.

3.4.5. Hold Slack

A hold slack concept is defined by the time between the arrival of data at the destination FF and the hold violation

$$S_{Hold} = (T_C + T_{IC}) - (T_{skew} + T_{Hold})$$

Like in the setup slack, a negative hold slack corresponds to a hold violation, which tend to be much less common than setup violations, but both are usually caused by the clock skew, because an increased skew decreases the slack [48]. FPGAs usually have specialized clock networks to help minimize this skew. Hold Slack in FPGA design is presented in Fig. 3.9

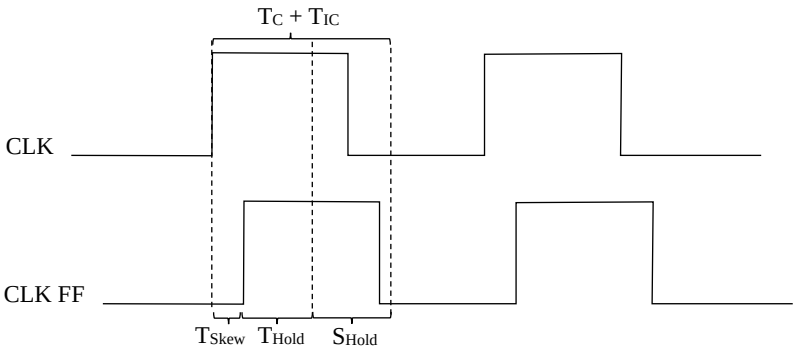


Fig. 3.9 Hold Slack in FPGA design.

3.5. Timing optimization methodology

Once a clock period is defined in a design constraint, the only possible control after synthesis is the delay between FF, which leaves two optimization options: reducing cell delays, or reducing interconnect delays. Through analyzing the timing statistics report which is generated by the timing analyzer after the design synthesis, the total negative slack (TNS) is one of the most important parameters that should be noticed. This total negative slack is the summation of slack on all failing paths that have setup violations. If the TNS is positive, it basically means there are no timing violations; However, if its value is negative, then the magnitude of that negative number provides an estimate of the effort required for timing optimization. For example, if the total negative slack is (-0.01 ns), that will likely only require very small optimizations on one or several paths. If the total negative slack is less than (– 1000 ns) on the other hand, the design requires a major correction.

Path analyzing, means looking for cells and interconnect delays (or possibly both) that are violating timing setup, and causing a bottleneck to the design which limits the maximum frequency. After identifying those bottlenecks, the design must be modified by applying one of the timing optimization techniques to eliminate that bottleneck [51]. The operation of bottlenecks correction consists of identifying the sources of the timing violation within the most critical path, and correcting it if possible. This process is repeated until meeting the timing requirements. Timing characteristics of

the entire circuit may change significantly after performing some optimizations, leading to resolving some of the previous bottlenecks, but also new bottlenecks may appear.

3.5.1. Logic delays optimization

3.5.1.1. Reducing number of logic inputs

One of the common strategies used to reduce lookup table delays, focuses on reducing the number of logic inputs, which allows fitting the same function into fewer lookup tables, and therefore reducing the depth of lookup table hierarchy. A Lookup Table (LUT) in an FPGA is a small memory block that stores pre-defined outputs for every possible input combination, effectively implementing a logic function. A LUT has N inputs and one (or two) outputs. This implies, that any function with more than N inputs requires more than one LUT as shown in Fig. 3.10 and Fig. 3.11.

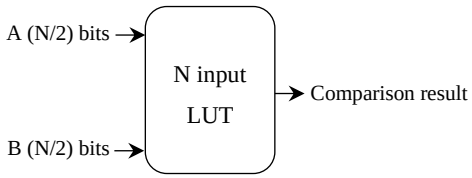


Fig. 3.10 Comparison of N/2-bit vectors.

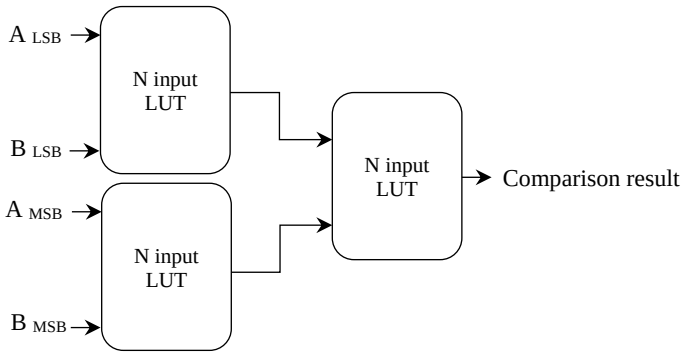


Fig. 3.11 Comparison of N bit vectors.

A counter implementation is a good example to illustrate resource optimization, as there are two possible methods; either counting up from zero to the target or counting down from the target to zero. The target in both cases is defined as input [52]. However, Counting down is more resource-efficient, where the counter is initialized with the target value and decremented until reaching zero, therefore there is no need for additional register to store the target value for comparison purposes, which saves 6 bits register in case of 6 bits counter [48]. Furthermore, it trims down the comparator logic, reducing the number of LUTs getting used in the design, as it needs to compare with all bits at zero. Fig. 3.12 and Fig. 3.13 shows the two methods for implementing a counter in FPGA.

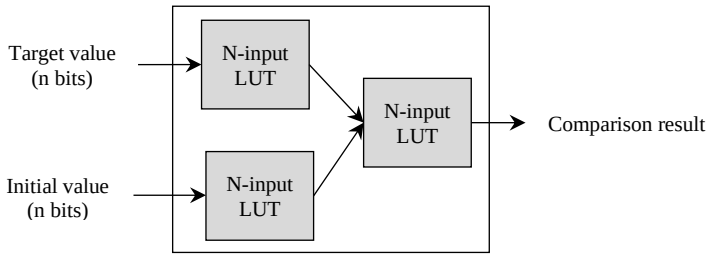


Fig. 3.12 Counter design before optimization (counting up).

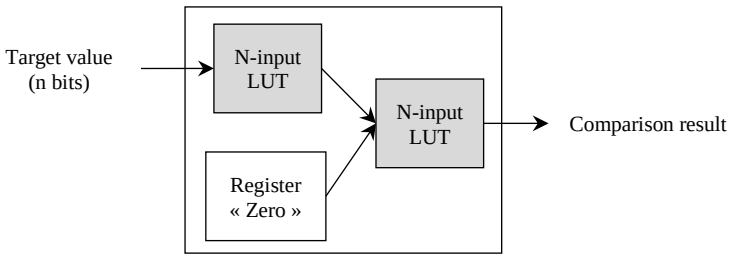


Fig. 3.13 Counter design after optimization (counting down).

3.5.1.2. Pipelining

A second strategy for minimizing lookup table (LUT) delays is reducing the longest LUT path by incorporating additional registers, this technique also referred to as pipelining. Pipelining is a frequently employed timing optimization methods. The principle of pipelining is increasing latency for exchange of a reduction in logic delays, which leads to higher the clock frequency. Taking the context of a -input comparator as example, several scenarios where the number of inputs cannot be reduced, resulting in a fixed LUT hierarchy that is two levels deep. Pipelining in such cases, can be applied by introducing additional registers at the outputs of each LUT, which effectively result into trading extra cycles of latency for a remarkable reduction in LUT delays, as illustrated in the Fig. 3.14. Despite its effectiveness, pipelining has also some limitations [48]. Namely, modifying latency by adding registers to a component can disturb the functionality of the overall circuit. This is why register-transfer level (RTL) synthesis tools cannot automatically implement such changes, necessitating manual adjustments to other parts of the circuit, which may, in turn, introduce new bottlenecks that require further pipelining [52]. Additionally, implementing pipelining in RTL code can be challenging. For example, while a purely combinational comparator can be easily implemented using the equality operator, introducing registers into the logic may require a counterintuitive design approach. This could involve explicitly partitioning the logic based on LUT placement and manually inserting registers at the outputs of each LUT, a process that is neither intuitive nor efficient. Fortunately, automated solutions exist in many cases to streamline this process, mitigating some of the practical challenges associated with manual implementation.

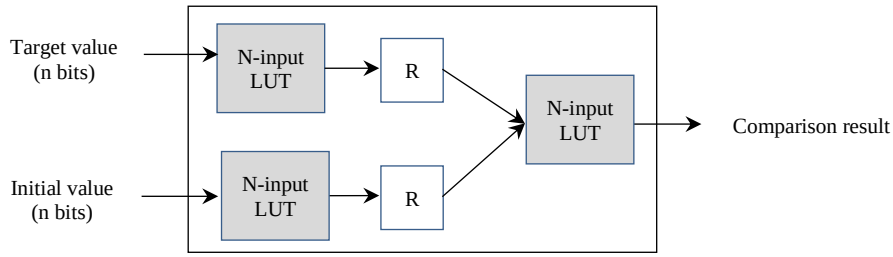


Fig. 3.14 Pipeline structure in FPGA design.

3.5.1.3. Balancing Logic and Register Placement

The third strategy for reducing lookup table delays is re-timing, which is the process of moving registers either forwards or backwards to help balancing logic delays. Basically, re-timing tries to move logic from a stage with low slack into a stage with higher slack. For example, as illustrated in Fig. 3.15, the original circuit has two adders with no registers in between them but with a register on the output, retiming can take the output register, and move it backwards onto both inputs of the second adder, which reduces the maximum cell delay from two adders to one adder, at the cost of one extra register [52]. Alternatively, forward retiming could be applied by moving the two input registers of the first adder to a single register at the output, which improves cell delays and reduces the number of required registers by one [51]. However, the feasibility of such adjustments depends on the broader context of the circuit, as the impact of retiming cannot be fully assessed without considering the preceding logic.

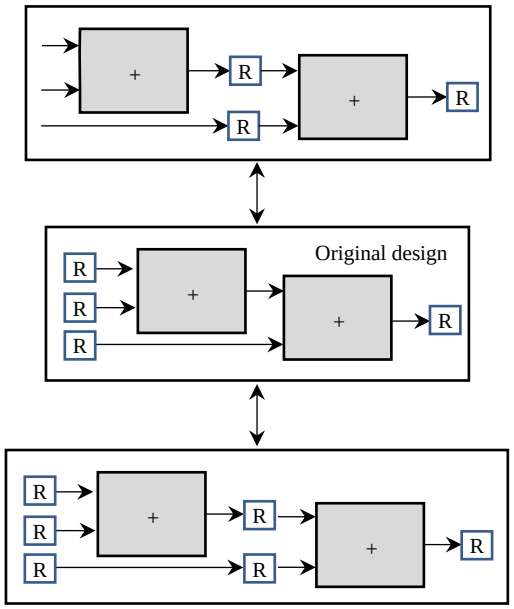


Fig. 3.15 FPGA design retiming example.

Retiming can be done manually through placing registers in their exact position, or automatically via synthesis tools. Automatic retiming requires the existence of the registers, as not all RTL synthesis can add extra registers or take away existing registers, but it moves around existing registers.

Another critical consideration is that automatic retiming is constrained by the need to preserve the initial states. This means that upon reset, there can be parts of the logic with slight harmless differences between the original circuit and the re-time circuit, but the synthesis tool identifies them as differences anyway, regardless if they impact or not the functionality of the design [52]. Fig. 3.16 highlights the ability of automatic retiming to streamline the implementation of pipeline designs, such as the comparator example discussed earlier.

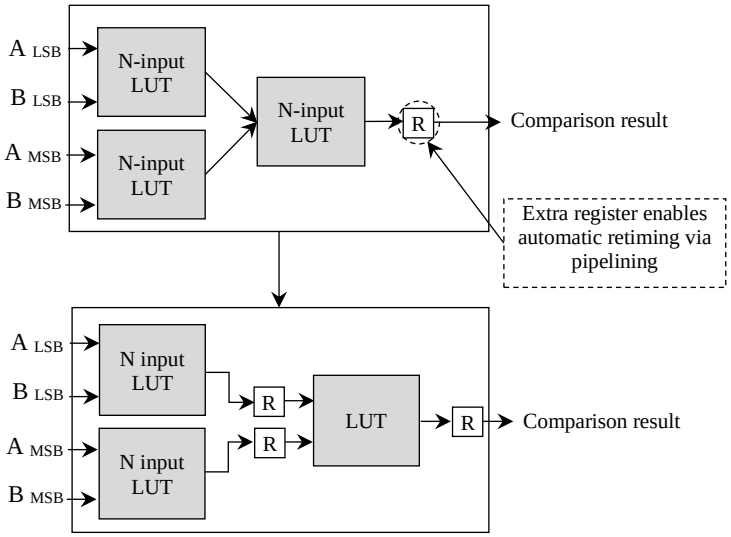


Fig. 3.16 Simplified pipeline design using automatic retiming: comparator example.

By simply adding an extra flip-flop at the output and enabling retiming in the synthesis tool, the tool can automatically reposition the flip-flop between the lookup tables, achieving the desired pipeline structure without manual intervention. However, the effectiveness of automatic retiming can vary across different synthesis tools, and comprehensive guidelines for its use compared to manual retiming are not universally established. Therefore, it is advisable to evaluate automatic retiming in a modular manner, testing its performance on individual components of the design before scaling up [53]. This approach ensures that the synthesis tool can achieve the desired retiming outcomes without necessitating extensive manual adjustments, thereby avoiding potential inefficiencies in the design process.

3.5.1.4. Precision optimization for reducing Lookup Table Delays

A fourth common strategy for reducing lookup table delays is to optimize precision. For instance, the previous comparator example, the two N-bit inputs can be replaced with three bits each, which enables using a single six-bit LUT instead of two. The obvious limitation in this strategy is that the optimized precision, this comparator situation is for explanation purposes, as it does not fit every situation, and the circuit functionality may be impacted by such modification. However, there are many other realistic situations where the precision optimization can be applied. Machine learning and

signal processing for example, are widely known to perform well with reduced precision compared to other applications. Neural networks are a good example, where the 32-bit floating point could potentially be replaced by 16-bit fixed point, or in many cases even by 10-bit fixed point. So basically, the idea is to identify an application specific optimized precision, which then enables a ton of logic optimization that will also reduce lookup table delays. One important remark, is that optimizing the precision does not guarantee a reduction in the number of lookup tables, or the maximum lookup table delay, a 4-bit comparator for example, would have the same maximum lookup table delay as the 6-bit comparator, despite the fact that the inputs used 2 fewer bits, so in this example, the reduction in delay cannot be noticed until the number of bits is reduced from 6 to 3 [54].

3.5.2. Interconnect delays

During the placement step of compilation, the placer tries to solve interconnect delay problem, which results from distantly placed resources, by minimizing the distances of connections between resources. This interconnect delays cannot be done manually especially in high resource utilization designs. For instance, Fig. 3.17 illustrates a simple example of this problem, the circuit shown in the figure contains a 4-to-1 MUX with flip flops on the inputs as shown in the left of the figure. The ideal placement of this MUX is represented in the middle of the figure, where, if we consider the fundamental building block in the target FPGA as a single six input lookup table, all of the connected components in this case are as close together as possible on the FPGA, which minimizes interconnect delays. Unfortunately, such a placement is not feasible in most cases, as FPGA designs typically require high resources, and when considering the design as a whole, the result would closely resemble the circuit shown on the right of the figure, where other components represented with “X” interfere with the placement of the MUX and corresponding FF, this, in turn, pushes some input further away from the MUX, leading to interconnect delay increases [52].

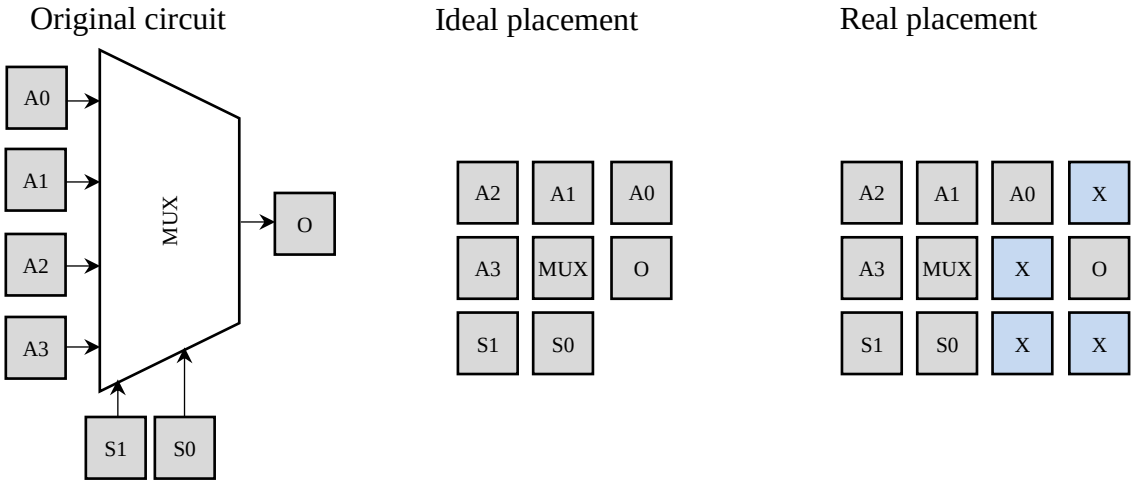


Fig. 3.17 Interconnect delays: ideal and suboptimal placement.

3.5.2.1. Interconnect delay optimization through resource utilization reduction

One common technique for optimizing interconnect delays is to reduce resource usage. In practice, this involves modifying resource types, specifically those causing bottlenecks in the design. Delaying a signal by many cycles for example, could use a large number of FF resources, however, it could also be implemented in any type of embedded memory. Similarly, DSPs often have low level optimizations that allows performing multiple operations within a single DSP. Another method that helps optimizing resources involves sacrificing performance, assuming this is possible, within design constraints by time multiplexing operations across fewer shared resources. Fig. 3.18 illustrates how resource reduction can help placement tool taking distant resources and move them closer together.

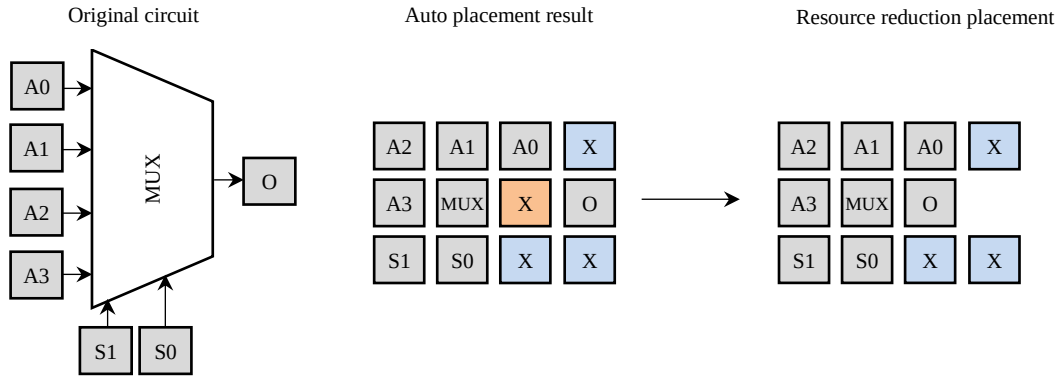


Fig. 3.18 Resource reduction for a reduced interconnect delay.

3.5.2.2. Pipelining interconnect paths to reduce routing delay

Placing additional flip-flops along long routing paths is another strategy to reduce the distance between resources, a technique known as pipelining. This technique impacts interconnect delays, where is sacrificing latency by adding extra registers, to break up a long delay into several shorter pipeline stages. Pipelining wires in this manner can be an effective technique for improving performance; however, its implementation can be complex. To be able to benefit from pipelining, a timing analyze should be used to identify distant connections first, subsequently the design must be modified to incorporate additional FF for the corresponding wires. The challenge of pipelining operation is that placement and routing aren't guaranteed to produce the same results on every run, where a lengthy connection in one compilation, might not be lengthy in another compile, but it may lead to the appearance of new lengthy connections, it all depends on how the components are placed. Large designs therefore face an additional challenge; besides potentially iterating multiple times to add registers to different wires -each time shifting the bottleneck- there is also an increased flip-flop usage, which can lead to routing congestion. Fortunately, these limitations are not always present and depend on the targeted FPGA. Fig. 3.19 illustrates a design pipelining example.

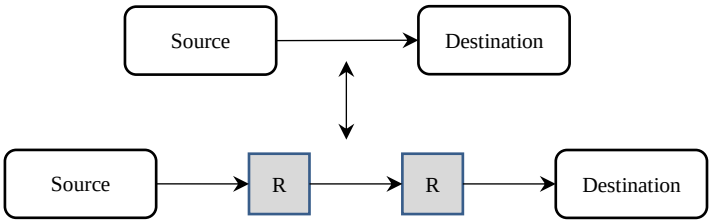


Fig. 3.19 Pipelining long interconnect paths for delay optimization.

3.5.3. Routing Congestion in FPGA Designs

After improving the placement of distant resources, the developed design may be face to a high routing congestion, which is a frequent issue in FPGA designs. Routing congestion usually occurs when many signals attempt to use the same routing resources in a specific region of the FPGA, leading to a high density of interconnects and components in that area. an example of this problem is shown in Fig. 3.20, where the red areas signify a high concentration of signals passing through that part of the design, resulting in heavy congestion. The main concern with routing congestion is not only the congestion itself, but the longer paths it creates, as the router attempts to reroute signals through alternative paths -often much longer- to avoid congested areas, interconnect delays increase.

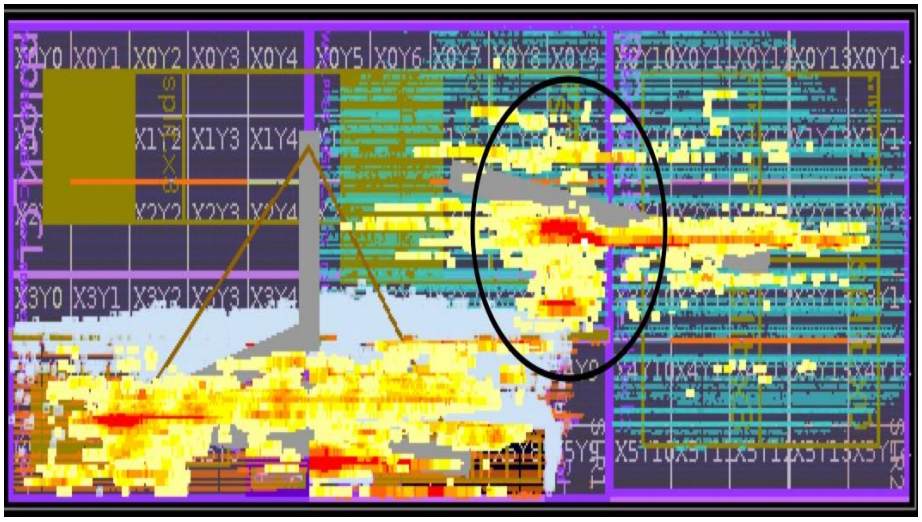


Fig. 3.20 Examples of routing congestion and optimization in FPGA Designs.

While numerous application-specific strategies exist to mitigate routing congestion, several general approaches can be highlighted. Resources reduction is one possible solution for routing congestion, with fewer resources or fewer connections between resources, routing congestion will potentially be reduced. Pipelining can also be a solution for long paths caused by congestion, but must be conducted carefully, as it has similar drawbacks as previously mentioned: long paths might change between compilations, and adding more FF to an already crowded design could make congestion worse, possibly canceling out the advantages of pipelining.

3.5.4. Optimizing Fanout

3.5.4.1. Register Duplication

Routing congestion is frequently caused by high fan-out signals, particularly wide ones, making this a common bottleneck in FPGA designs. When analyzing interconnect delays using a timing analyzer, it is crucial to identify nets with high fan-out, as these are often primary contributors to timing issues. One effective strategy for addressing fan-out bottlenecks is register duplication. High fan-out signals typically result in distant resource placements and routing congestion, as illustrated in the left part of Fig. 3.21, where the sinks (flip-flops 2 through 7) cannot all be placed close to the source (flip-flop 0). Register duplication mitigates this issue by replicating the source register, as shown in the right part of the figure. Here, a copy of flip-flop 0 is created, which reduced fan-out of three instead of six, enabling closer placement of some sinks.

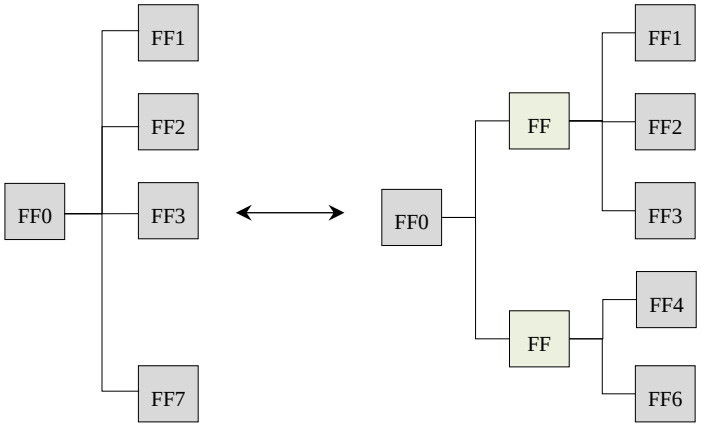


Fig. 3.21 Register Duplication for Reducing Fan-Out and Routing Congestion.

This optimization comes at a cost, in addition to the increased flip-flop usage, register duplication also raises the fan-out of the preceding signal (flip-flop 0 in this case). If the path between flip-flop 0 and flip-flop 1 meets timing constraints, the duplication is an effective optimization. However, register duplication essentially redistributes slack from the preceding path to the fan-out paths, which can sometimes shift timing violations from the original fan-out to the preceding logic. This trade-off makes it challenging to determine the optimal amount of duplication. While synthesis tools can automate this process, manual exploration of replication amounts is often necessary to achieve the best results [49].

3.5.4.2. Pipelined Fan-Out

A second strategy for addressing high fan-out signals is the use of pipelined fan-out trees. This approach distributes a high fan-out signal across multiple cycles, each with a lower fan-out, as illustrated in Fig. 3.22. The original circuit, with a fan-out of 8 from flip-flop 0, is transformed into the structure of a tree of flip-flops, added to connect flip-flop 0 to all sinks over two cycles, ensuring that no stage exceeds a fan-out of 3. Like other pipelining techniques, pipelined fan-out trees increase

flip-flop usage and latency, which may introduce limitations discussed earlier. Additionally, manual implementation can be cumbersome, as the optimal number of cycles and fan-out distribution for each instance may vary [49].

To simplify this process, some synthesis tools offer automated features, such as Hierarchical Proximity Register Chains, which allow designers to add registers before a fan-out source. The tool then retimes the design to automatically generate the desired tree structure [48]. However, this feature may impose coding restrictions that necessitate significant design modifications, potentially making manual implementation a more practical option in some cases.

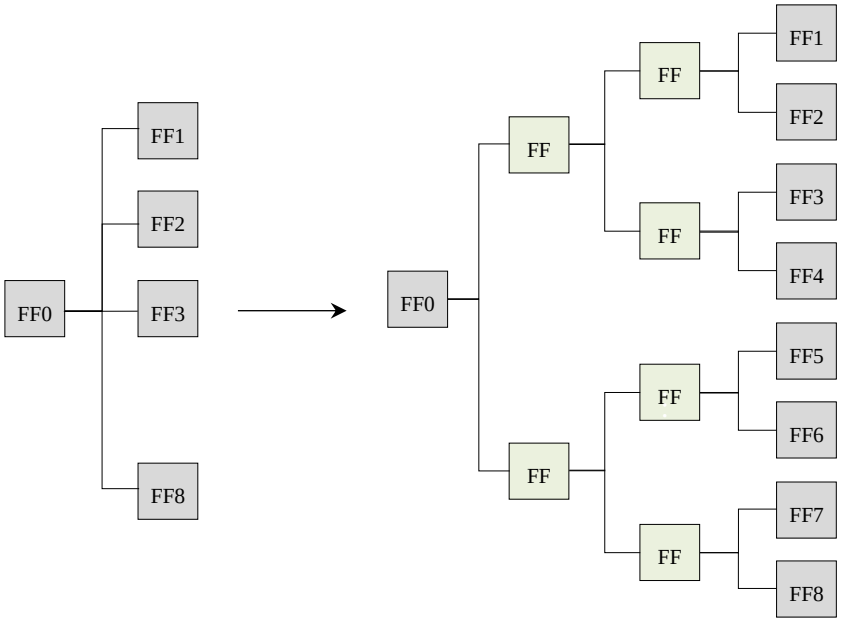


Fig. 3.22 Pipelining strategy for high fan-out

Chapter 4. Efficient AES implementation using FPGA

4.1. Introduction

The Advanced Encryption Standard is an iterative algorithm with recursive operations as described earlier, where the input to each operation depends on the output from the previous operation. Similarly, the input to each round depends on the output from the previous round, which indeed, is a point of strength in terms of security as it guaranties diffusion and breaks the relation between the plaintext and the ciphertext, but also a big challenge for resources-constrained systems. This challenging situation drew researchers' attention and motivated scientists to propose effective solutions to bypass the extended processing time because of the large number of rounds, and the complexity of its integration into resource-constrained systems. Among suggested solution we distinguish two categories:

The first solution involves modifications to the algorithm to reduce processing time and resources requirement. The resulting algorithm after this modification is a lightweight version of the original algorithm. Results in this case are satisfying in term of resource utilization and operational timing, however, it usually requires additional security analyses as it does not guarantee the same level of security as the original algorithm.

The second solution suggests maintaining the original algorithm without any adjustments to preserve its high level of security, but intervenes in the implementation technique instead to reduce processing time and resources utilization.

This section describes the designed system that takes advantage of the AES algorithm modularity to synthesize two efficient implementations, namely Pipeline and Iterative architectures. The iterative system is the classic implementation technique of AES, and the pipeline is one of famous architectures. observing timing and area occupation as monitoring parameters along with the efficiency ratio. obtained results are validated and compared to similar works.

4.2. Motivation for the Proposed Approach

The choice of the AES algorithm is based on its proven robustness, making it the most widely used encryption method. Furthermore, it is often integrated into MCUs and FPGAs as a security engine or core to safeguard configurations and firmware from reverse engineering aiming to trustworthiness. These factors have driven researchers to develop lightweight and high-speed encryption cores to ensure an efficient encryption mechanism with a steady dataflow. Thomas in [55] has listed the security measures taken by some FPGA vendors to protect configurations and prevent them from cloning and overbuilding, this list is presented in Table 4.1.

Table 4.1. Protection Mechanisms for some FPGA vendors.

Manufacturer	Device	Bitstream Encryption	Key storage
Achronix	Speedster22i HD	AES-256 (CBC)	eFuse
Altera	Stratix II/II GX	AES-128	NVM
	Stratix III/IV/V	AES-128	Volatile/NVM
	Cyclone III LS [Alt11a]	AES-128	Volatile
Latice	ECP2/M SS-Series	AES-128	eFuse
	ECP3	AES-128	eFuse
	ECP4	AES-128	eFuse
	XP2	AES-128	eFuse
MicroSemi	IGLOO	AES-128	NVM
	ProASIC3	AES-128	NVM
	SmartFusion	AES-128	NVM
	SmartFusion2	AES-128	PUF
Xilinx	Spartan3-AN	-	NVM
	Virtex-II	DES / triple-DES	Volatile
	Virtex-4	AES-256	Volatile
	Virtex-5	AES-256	Volatile
	Spartan-6	AES-256	eFuse/volatile
	Virtex-6	AES-256	eFuse/volatile
	7 Series FPGAs	AES-256	eFuse/volatile
	Zynq-7000	AES-256	eFuse/volatile

As presented in Table 4.1, most of FPGA vendors use AES as the encryption core of their configurations. Similarly, several other devices like MCU consider AES as the security system.

The Encryption process of AES consists of repeated iterations known by Rounds [56], each round is composed of sequential operations called SubBytes, ShiftRows, MixColumns, AddRoundKey, and key generation. The overall AES algorithm is presented in Fig. 4.1.

Iterative implementation is a closed loop structure of the main operations interconnected as point-to-point blocks in a feedback loop, the same blocks are therefore used during all the rounds.

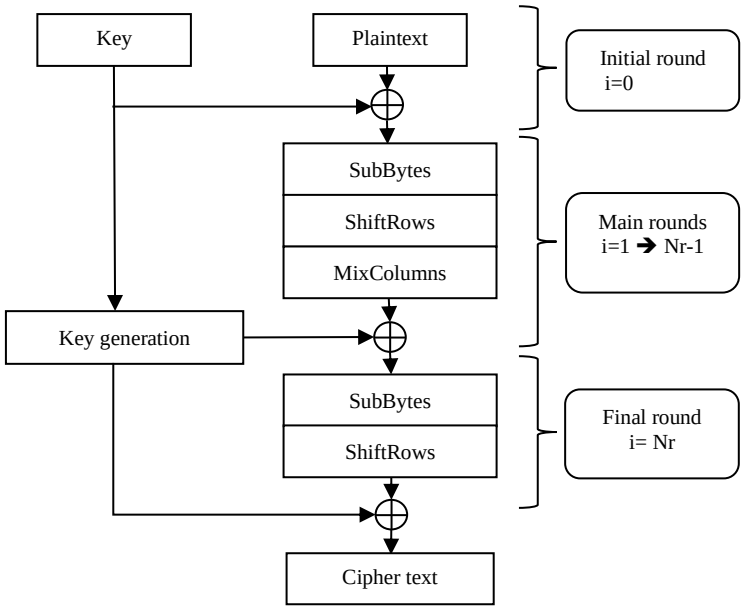


Fig. 4.1 Encryption process flowchart of the AES algorithm.

The major limitation of this method is the idle time caused by the iterations effect, but it requires less resources compared to other techniques, which makes it more compatible with resource-limited systems [57] . The iterative system process is illustrated in Fig. 4.2.

.

	Add round-key	Sub bytes	Shift rows	Mix columns
Round 0	Processing	IDLE	IDLE	IDLE
Round 1	IDLE	Processing	IDLE	IDLE
	IDLE	IDLE	Processing	IDLE
	IDLE	IDLE	IDLE	Processing
	Processing	IDLE	IDLE	IDLE
Round 2	IDLE	Processing	IDLE	IDLE
	IDLE	IDLE	Processing	IDLE
	IDLE	IDLE	IDLE	Processing
	Processing	IDLE	IDLE	IDLE
	. . .			
Round 10	IDLE	Processing	IDLE	IDLE
	IDLE	IDLE	Processing	IDLE
	Processing	IDLE	IDLE	IDLE

Fig. 4.2 Iterative architecture process.

Pipeline structure consists of multiple AES modules interconnected sequentially [58]. Each module processes one round, hence, it operates one time during the encryption process and becomes free for the next input. As pipeline architecture allows processing several inputs for reducing idle time and critical path, it increases the operating frequency, but consumes more resources. Fig. 4.3 illustrates the AES pipeline architecture, and Fig. 4.4 shows the pipeline process. This technique is well adapted to systems that require a steady stream of data.

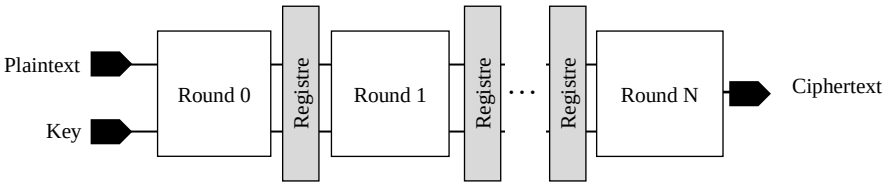


Fig. 4.3 Pipeline Architecture of AES algorithm.

SubBytes is a nonlinear operation, in which each byte in the input is transformed as a polynomial in the Galois Field $GF(2^8)$ using the affine transformation and matrix multiplication [59]. The same results can be obtained using a precalculated substitution table called S-box.

ShiftRows and MixColumns are linear transformations to the plaintext, where rows are shifted and columns are mixed in an invertible manner. Rows shifting operation consists of sliding rows to the left by a given offset, the first row is unchanged, while the second row is shifted over one byte, similarly the third and the last rows are shifted by two and three bytes respectively. In the same way, MixColumns changes the order of columns by multiplying the plaintext by a given matrix.

AddRoundKey is the operation where the main process meets the key generation process. It is considered the most important step since it mixes the plaintext with the secret key to hides the relationship between the original data and the ciphertext [60].

The set of operations described earlier are repeated *N* times in a feedback loop, where *N* is the number of rounds. The input to each round during the encryption process is driven from the output of the previous round in a closed loop structure, which is a point of strength in terms of security [61], but also a big challenge for resources-limited and interactive systems. Numerous methodologies of AES implementation have been elaborated, and this work focuses on iterative and pipeline techniques.

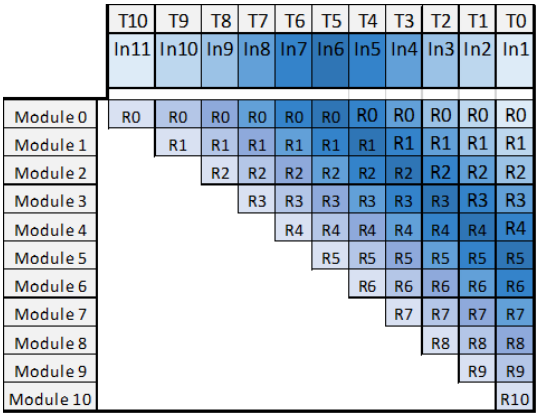


Fig. 4.4 Pipeline process of AES algorithm.

Proposed architectures are designed using a Hardware Description Language (HDL). Synthetization and simulation results are compared to similar works in terms of resource utilization, throughput, and efficiency ratio

4.3. Development process

The Synthesis and simulation proposed architectures is performed using Vivado HLx. Developed approach is built following steps illustrated in Fig. 4.5. The design is developed hierarchically, beginning with the simplest components and progressing towards the top-level structure.

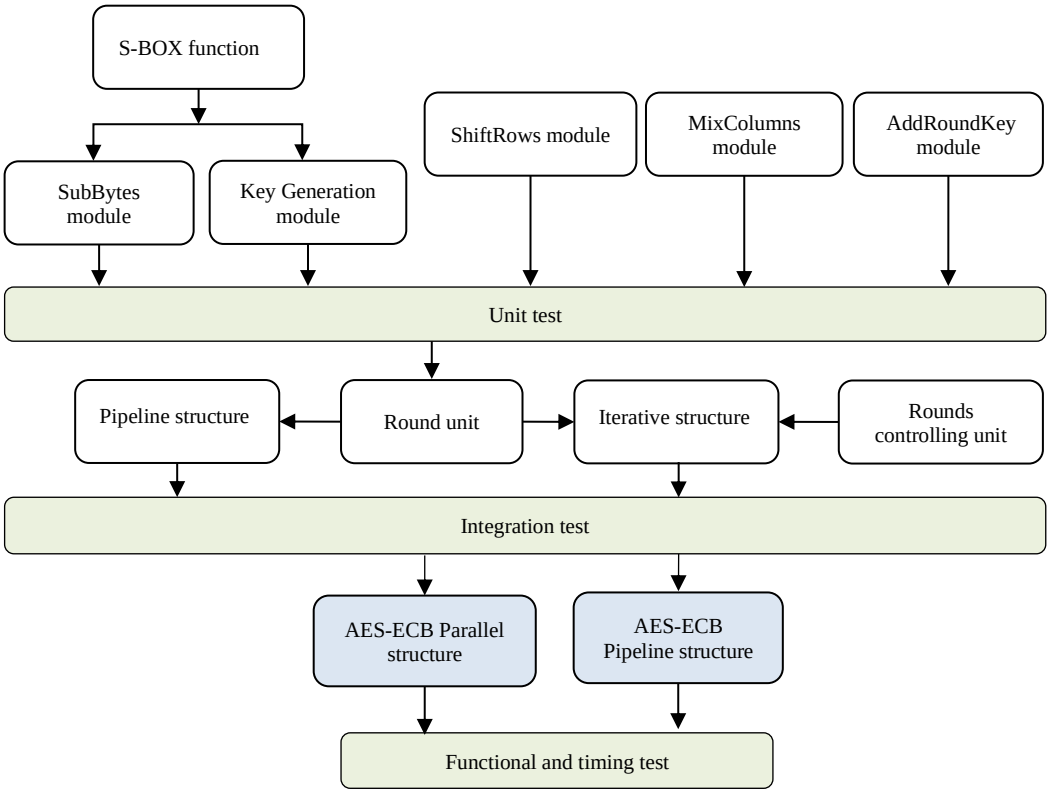


Fig. 4.5 Development process of the proposed designs.

4.4. Development of different modules

As described in previous sections, AES is a set of operations grouped into a Round, which is repeated a fixed number of times depending on the key size (10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys). The hardware design of this algorithm starts with basic elements, such as S-boxes and XOR gates, and builds up hierarchically. Since HDL (Hardware Description Language) supports modular design, the implementation is constructed step by step, from simple components to the full AES module.

4.4.1. SubBytes unit

SubBytes is the part of AES that guarantees the non-linearity of output to input data. This module is based on the S-box, as it uses the state bytes as the s-box coordinates to permutate them by the precalculated bytes. The S-box has two possible implementations techniques, namely Memory based and-Logic-based implementations [62]. The S-box is set to memory-based implementation as it is

static, and does not require updates during the process. The choice of memory-based implementation of S-box is done based on experiments, as it results to a reduced latency compared to logical implementation. The schematic diagram of the S-Box module as generated form the design IDE is illustrated in Fig. 4.6, and the SubByte schematic in Fig. 4.7.

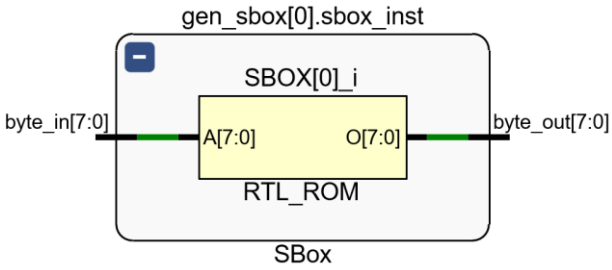


Fig. 4.6 S-box byte-cell schematic diagram.

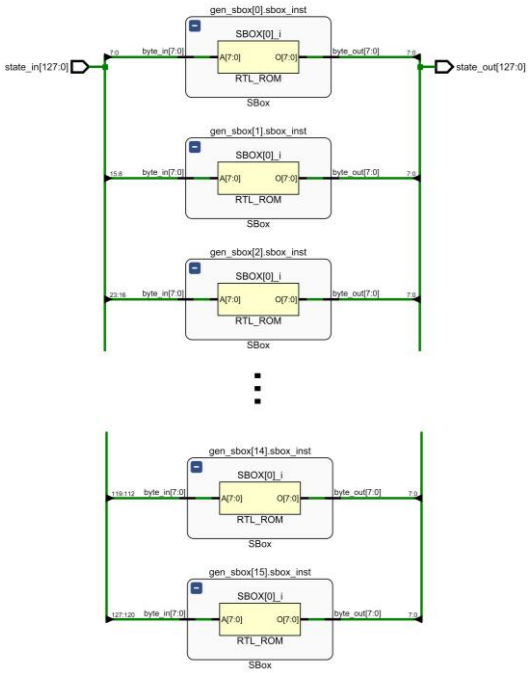


Fig. 4.7 SubBytes schematic diagram.

4.4.2. Key Generation (Key expansion)

Key Generation uses the same s-box architecture as in the SubBytes module, combined with logic operations to generate a sub-key for each round of the algorithm. The schematic diagram of the Key Generation is illustrated in Fig. 4.8.

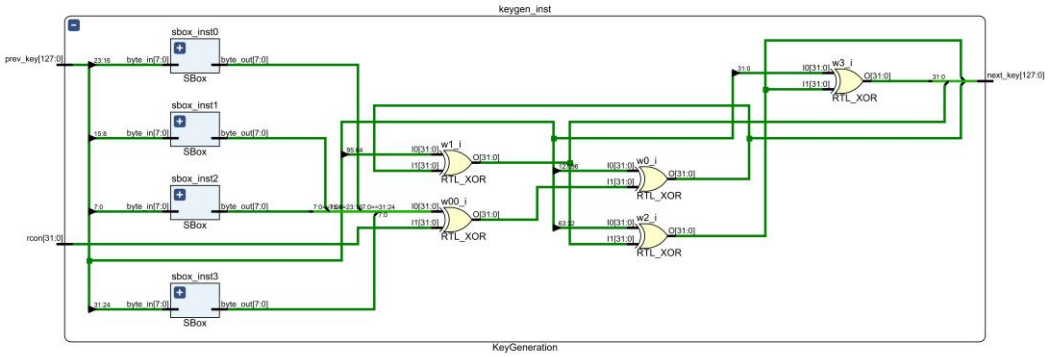


Fig. 4.8 Key Expansion module schematic diagram.

4.4.3. ShiftRows

ShiftRows is a linear operation, it consists on driving input bits in a different order to the output, this module does not include complex operations.

4.4.4. MixColumns

MixColumns is another linear operation, but it involves some operation as it modulo-multiply each column from the input in Galois field by a predefined matrix. The overall Mixcolums schematic diagram is presented in Fig. 4.9, and the MixColumn sub-module in Fig. 4.10.

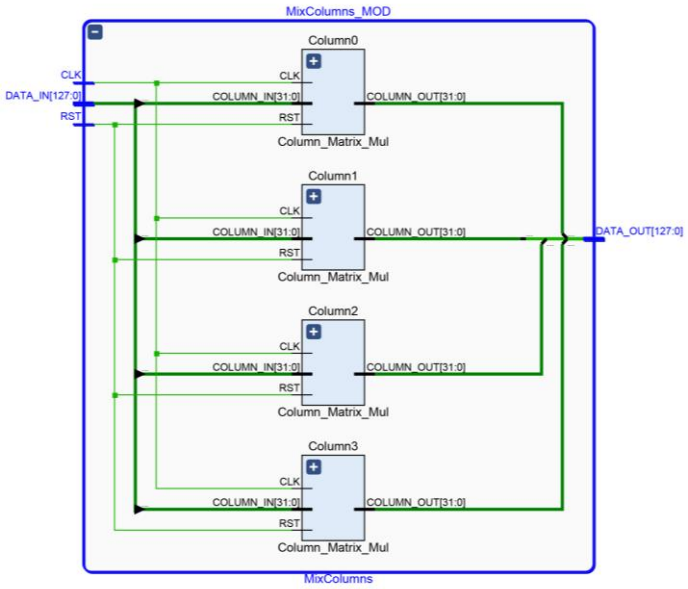


Fig. 4.9 MixColumns schematic diagram.

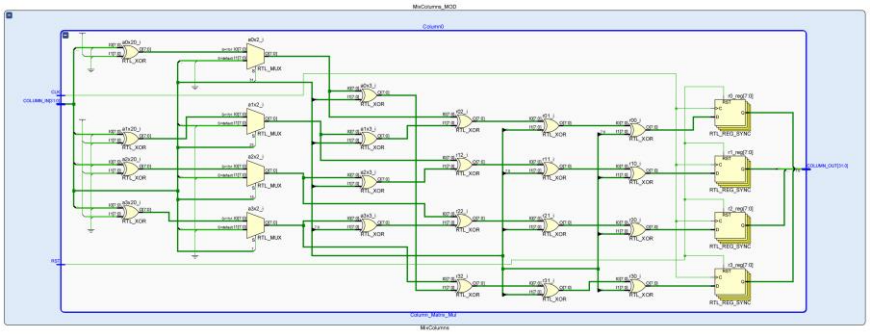


Fig. 4.10 MixColumns Sub-Module.

4.4.5. AddRoundKey

The encryption main process meets the key Generation process at this level to perform a XOR operation between the subkey of the active round and the state. The output of this module is the ciphertext itself in case the active round is the last round, else it is driven to the SubBytes input of the next round. The schematic diagram of the AddRoundKey module is illustrated in Fig. 4.11.

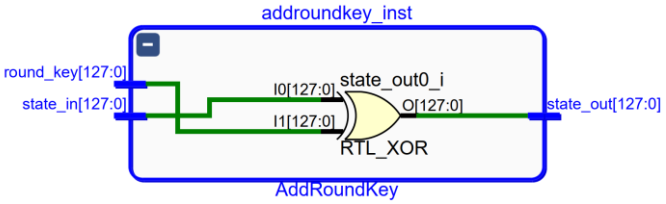


Fig. 4.11 AddRoundKey schematic diagram.

4.4.6. Round units

The AES-round unit envelops the units: SubBytes, ShiftRows, MixColumns and AddRoundkey, in addition to the Key Generation operations. At this level we notice the pipeline effect in the developed system. A synchronization registers are placed in order to avoid latency cumulation that may be causes by the cascade effect in the case of pipeline architecture. This module is considered the fundamental component for both iterative and pipeline architectures. Using the same round modules gives a better observation of the architecture impact on the design performance. The schematic diagram of the round unit is presented in Fig. 4.12.

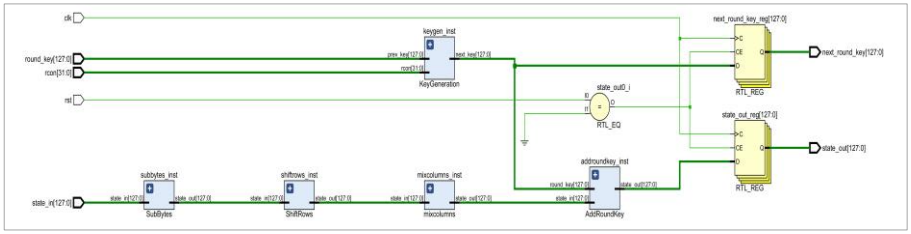


Fig. 4.12 Schematic diagram of the round unit.

4.4.7. Iterations control unit

The rounds controlling unit plays a key role in timing performance and reducing latency through managing rounds counting, dataflow, and synchronize between modules. This module is designed for the iterative system. It allows using one single round to run all the encryption process recursively. This module controls all dataflow, it drives inputs according to the active operation and the active round. It also puts the plaintext in hold while processing a data, and can be updated to generate a status output along with operations flags to manage dataflow. This feature makes the designed architecture scalable as it is modular and include a centralized control unit. The schematic diagram of the control unit is illustrated in Fig. 4.12.

4.4.8. Design of the iterative system

The Iterative system at this level is obtained by assembling the round unit with the controlling unit in a closed loop structure. The schematic of the iterative system is presented in Fig. 4.13.

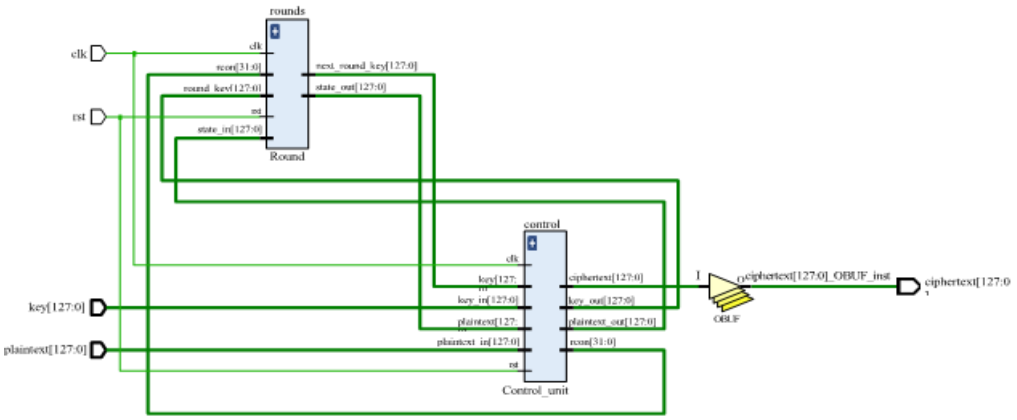


Fig. 4.13 The iterative system schematic diagram.

4.4.9. Design of the pipeline system

Pipeline implementation is designed through placing 10 round units in serial, each round unit operates one single time during the same encryption process, as a result, the system processes a new input each clock cycle. Synchronization registers are placed in each round output to reduce data latency cumulation. These registers are essential in pipeline designs. Fig. 4.14 shows the schematic diagram of pipeline implementation.

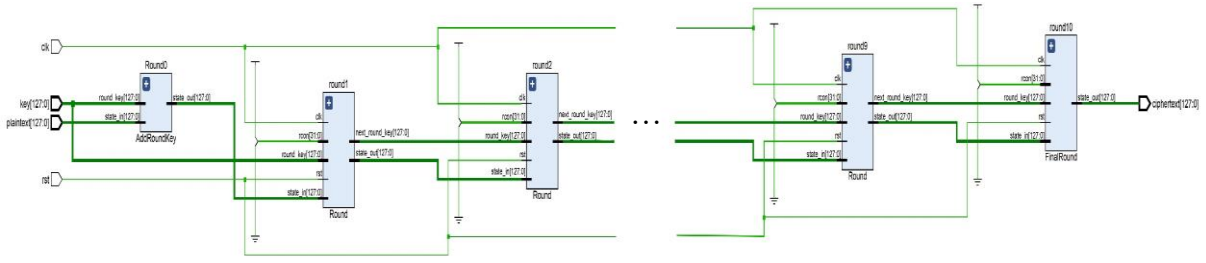


Fig. 4.14 Pipeline system schematic.

4.5. Results and discussion

The proposed architectures are implemented in Zynq 7000, and Virtex-7 FPGA. Fig. 4.15 and Fig. 4.16 illustrate a graphical representation of resource utilization, throughput, and efficiency ratio of proposed architecture in Virtex-7 and Zynq 7000 FPGAs respectively.

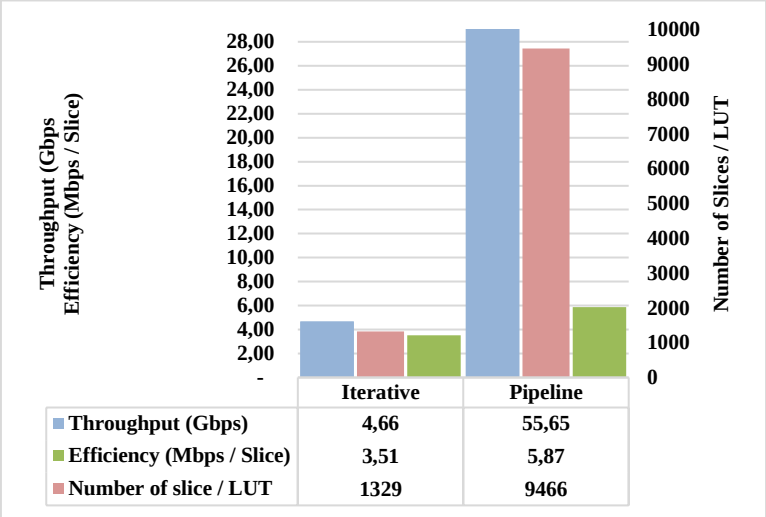


Fig. 4.15 Graphical representation of implementation results in Virtex-7 FPGA.

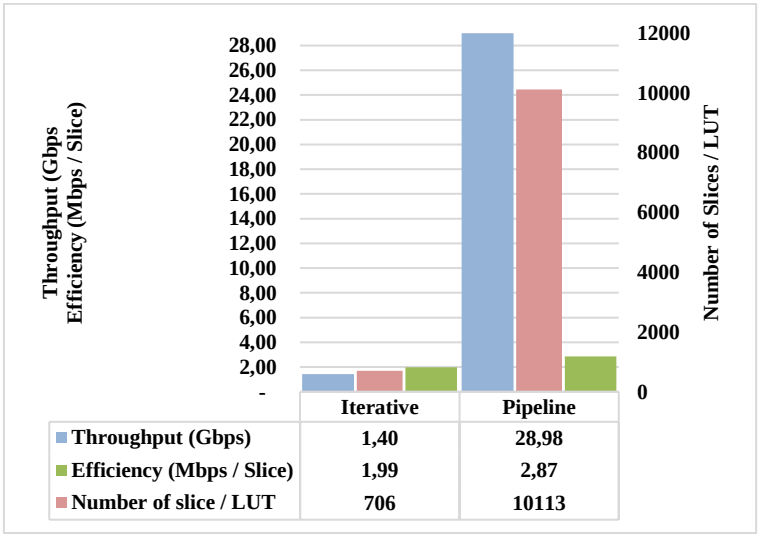


Fig. 4.16 Graphical representation of implementation results in Zynq7000 FPGA.

We notice that pipeline architecture processes data faster at the expense of using more resources than the iterative system. This remark is applicable for both platforms. The progression of efficiency ratio when transitioning from iterative to pipeline systems provides a distinct indication of the optimal trade-off between resource utilization and operational time.

Table 4.2. Comparison of the proposed architectures results with published works summarizes timing results of the methods proposed in this work, and a comparison with similar works elaborated in [13], [15], [16], [17] and [63] in terms of resource utilization and efficiency ratio.

The maximum frequency Fmax of a design is computed through running the synthesis and implementation and increasing the timing constraint until getting the smallest Slack violation (WNS <0) in the timing analyses report. The Fmax is calculated using the following equation:

$$Fmax\ (MHz) = \max\left(\frac{1000}{T_i - WNS}\right)$$

Where:

- T_i is the target clock period (*ns*).
- WNS is the worst negative slack (*ns*) of the target clock.

Table 4.2. Comparison of the proposed architectures results with published works

Architecture	Device	Number of clock cycles	Frequency (MHz)	Slices	Throughput (Gbps)	Efficiency (Mbps/Slice)
Pipeline system [13]	Spartan-3	12	-	46745	0.75 (Cycles/Byte)	-
Pipeline system [15]	Virtex-6	-	190.65	1551	0.56	0.361
AES- MPPRM [16]	Virtex-6	1	433.28	8342	-	-
Pipeline system (proposed)	Virtex-7	1	434.78	9466	55.65	5.87
Pipeline system (proposed)	Zynq 7000	1	226.44	10113	28.98	2.87
AES-CTR [64]	Virtex-6	11	318.67	7758 (area)	20.3	2.85
Non-Pipelined [65]	Virtex-7	11	456	2444	5.30	-
Iterative system (proposed)	Virtex-7	10	364.17	1329	4.66	3.51
Iterative system (proposed)	Zynq 7000	10	109.56	706	1.402	1.99

Throughput is calculated using the Fmax and the number of clock cycles, Throughput is defined by the following equation:

$$S\ (Mbps)\ = \left(\frac{Fmax\ \times\ len(d)}{n}\right)$$

Where:

- *Fmax* is the maximum frequency of the design (*MHz*).
- *len(d)* = 128 is the block size (*bits*).
- *n* Number of clock cycles.

Efficiency is the ratio of area to speed, and is calculated as follows:

$$E\ =\ \left(\frac{S}{N}\right)$$

Where:

- *S* is throughput (Mbps)
- *N* in number of slices

4.6. Simulation results

Simulation results of iterative and pipeline implementations are shown in Fig. 4.17 and Fig. 4.18 respectively

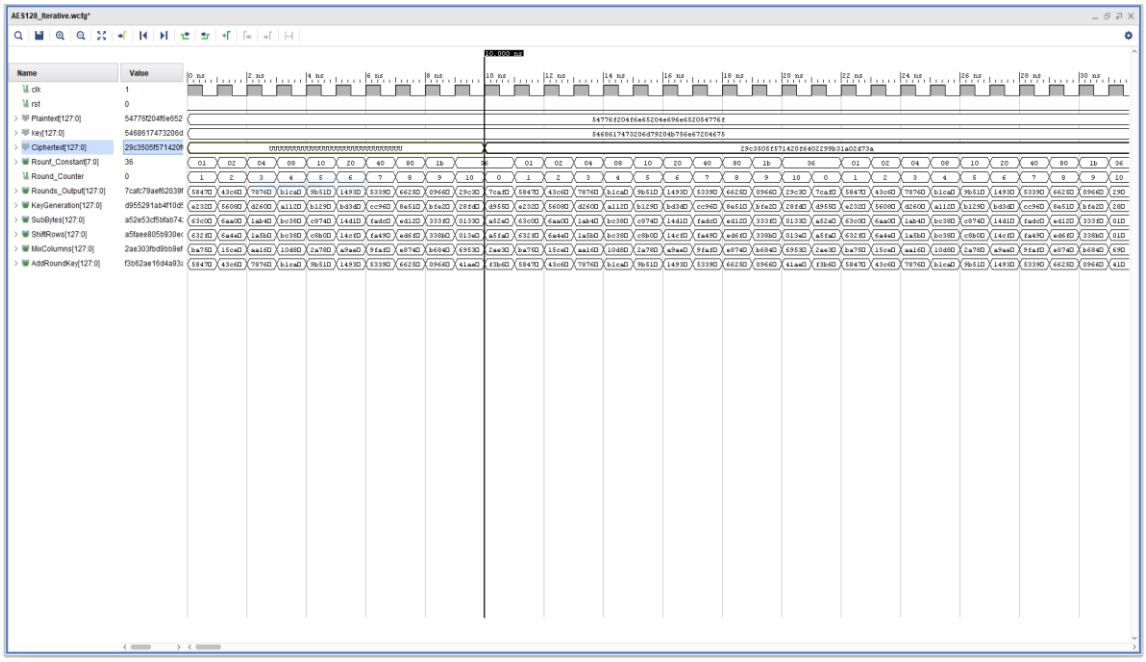


Fig. 4.17 Chronogram result of the iterative system.

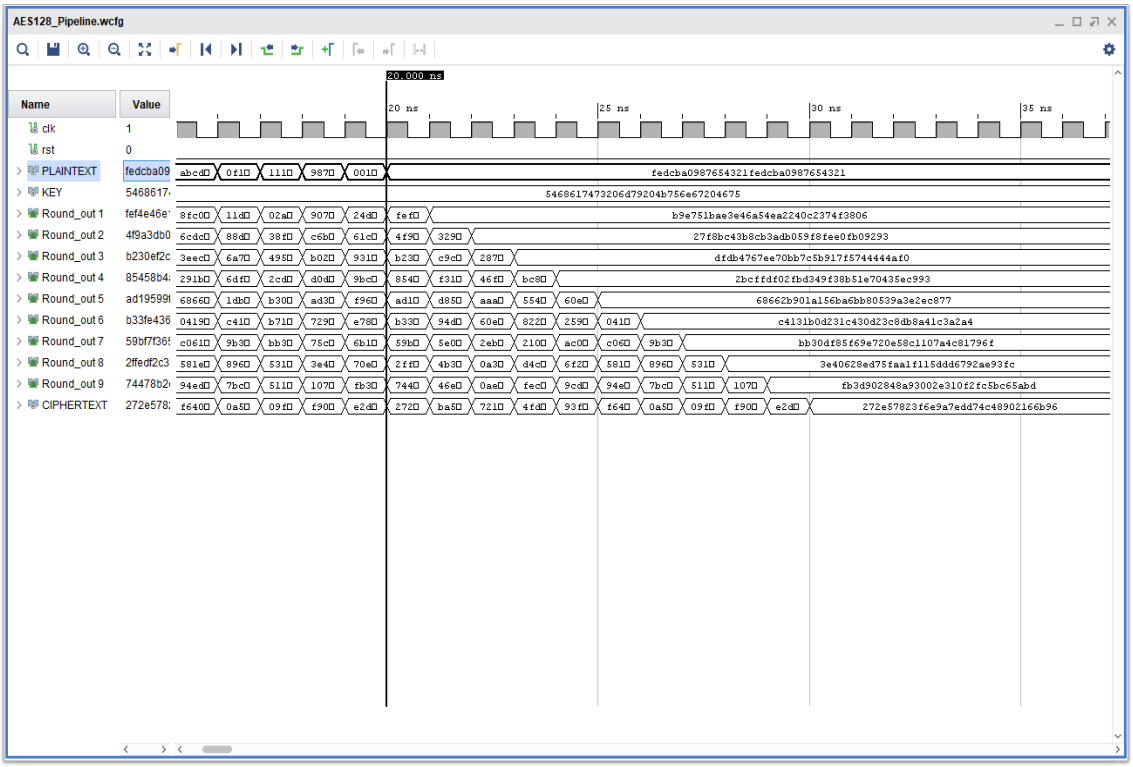


Fig. 4.18 Chronogram result of the pipeline system.

4.7. Implementation results

Both iterative and pipeline architectures are implemented in Virtex-7 and Zynq7000 FPGA families of Xilinx. Resource utilization is measured post optimization and implementation of the design. Implementation footprints of iterative architecture in Virtex-7 and Zynq7000 devices are illustrated in Fig. 4.19, and Fig. 4.20 respectively. Implementation footprints of pipeline architecture in Virtex-7 and Zynq7000 devices are illustrated in Fig. 4.21 and Fig. 4.22 respectively.

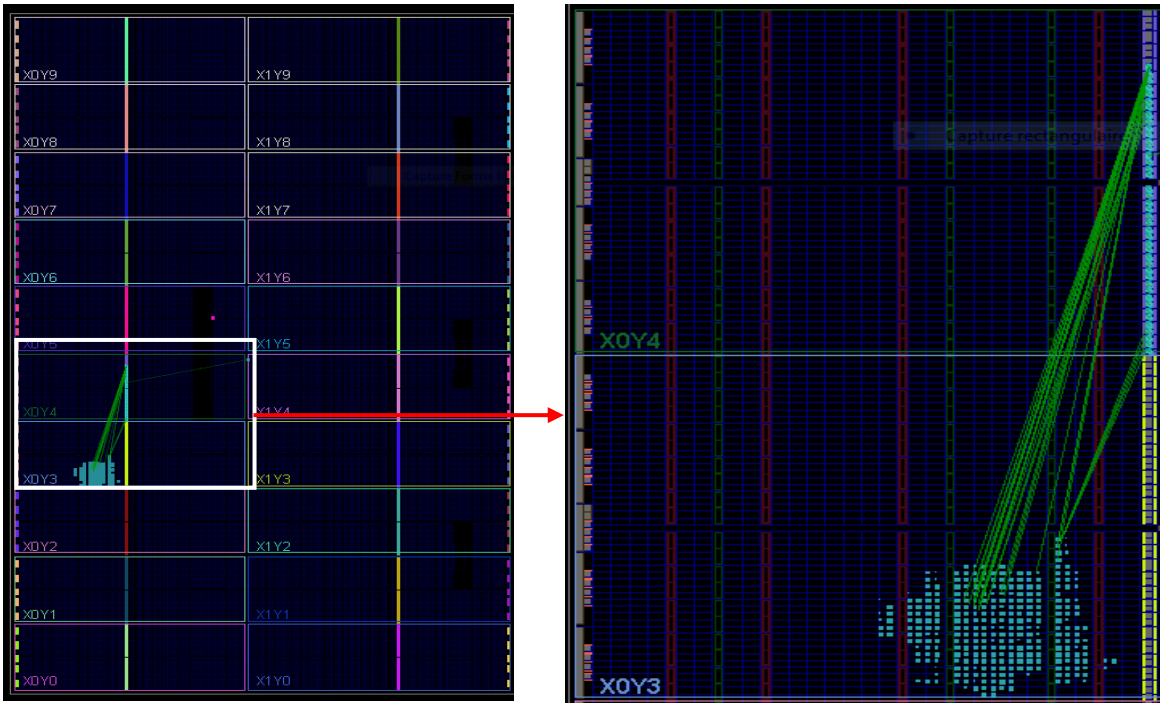


Fig. 4.19 Implementation footprint of iterative system on Virtex-7 FPGA.

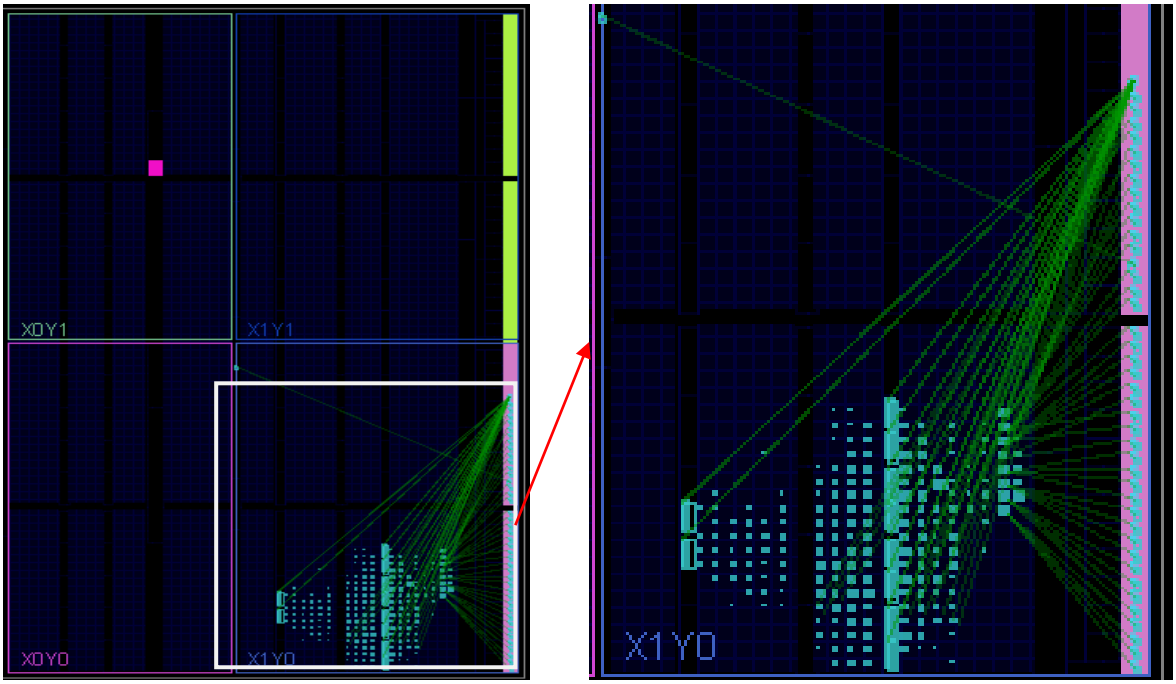


Fig. 4.20 iterative structure footprint on Zynq7000.

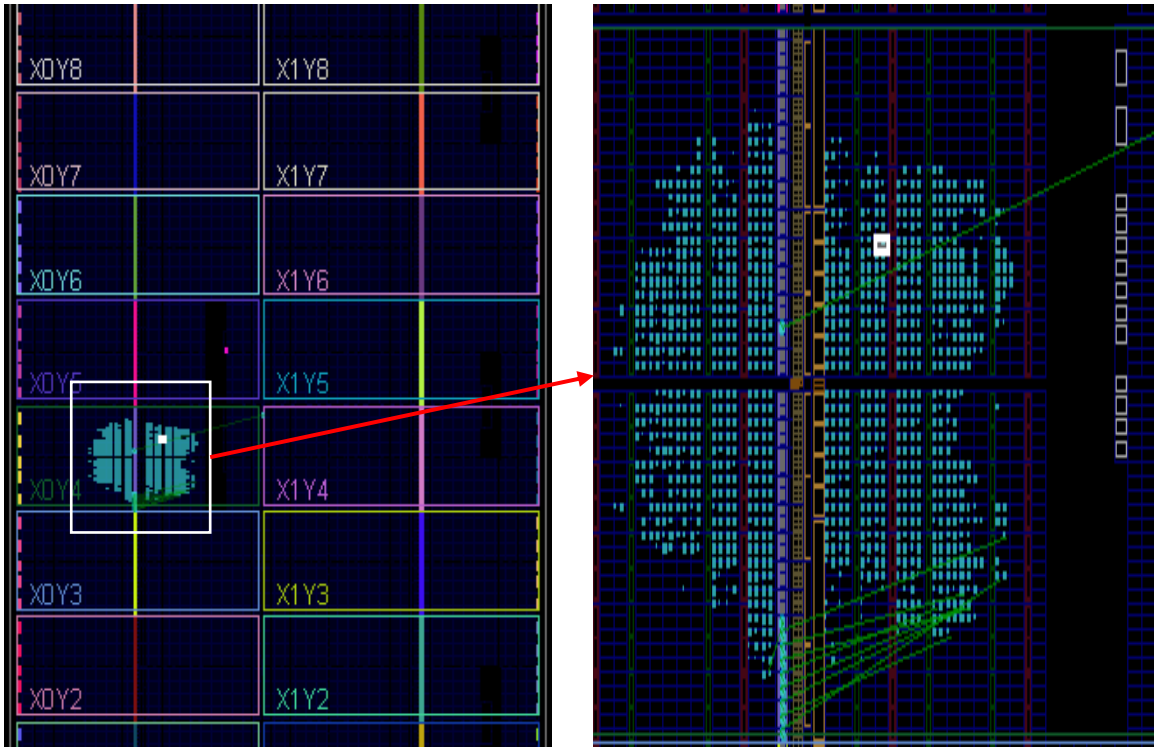


Fig. 4.21 Pipeline structure footprint on Virtex-7.

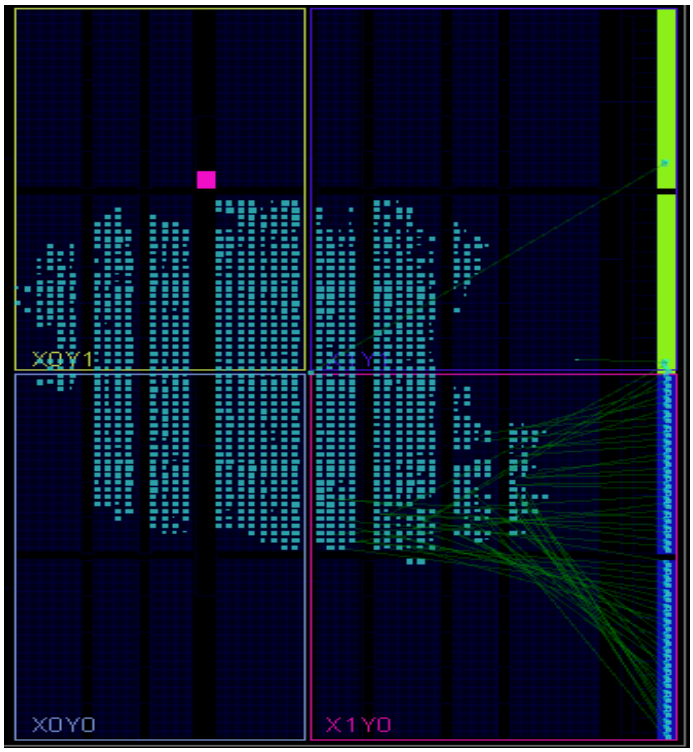


Fig. 4.22 Pipeline structure footprint on Zynq-7000.

The iterative architecture processes each plaintext in 10 clock cycles, as it has a single core of round unit controlled by a central controller.

In the pipeline simulation results we notice the presence of the 10 modules that represent the 10 rounds, where each module processes a single round. The pipeline dataflow is also illustrated in the chronogram. The first input is processed in 10 clock cycles, and the following plaintexts are processed each clock cycle. The pipeline design succeeded to work on ECB mode with a steady dataflow, where it can be loaded with a new input each clock cycle, and does the implementation separately.

4.8. Design of Electronic codebook mode

The pipeline structure is tested in ECB mode to encrypt larger datasets, as it has been proven to be the most efficient approach as mentioned earlier and supported by related works. In this step, the AES - ECB is tested using two distinct methods. The first method uses a single AES unit with an extra module at the input to divide original data into vectors, and another module in the output to concatenate ciphertext to generate encrypted data. The overall structure of the first method uses an inner pipelining system, while the AES unit itself is an outer-round pipelining. This combination allows achieving a maximum throughput with an optimized area. The structure of this method is shown in Fig. 4.24.

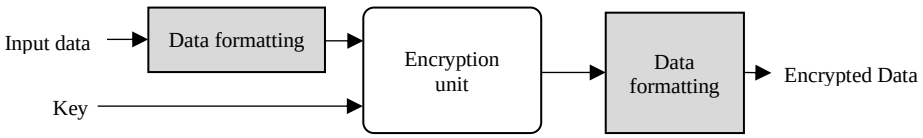


Fig. 4.23 Pipeline structure of AES-ECB mode.

The second method uses a similar inner-round pipelining system, but defer in the overall structure, where it uses a parallel data processing instead via integrating multiple encryption units. The generated ciphertexts are concatenated using a buffer positioned in the output as illustrated in Fig. 4.24.

The simulation results of parallel and pipeline structures are illustrated in Fig. 4.25, and Fig. 4.26 respectively. The pipeline structure processes the first input in 10 clock cycles, while one plaintext is loaded every clock cycle. The parallel structure processes several inputs simultaneously; therefore, it achieves a high throughput at the cost of high resource utilization.

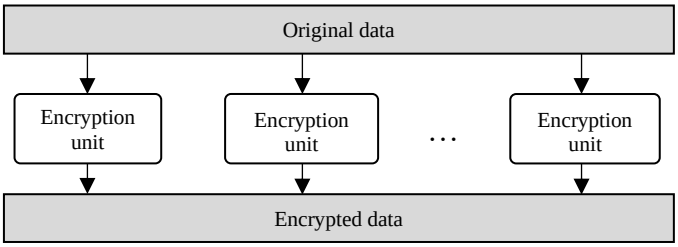


Fig. 4.24 Parallel structure of AES-ECB mode.

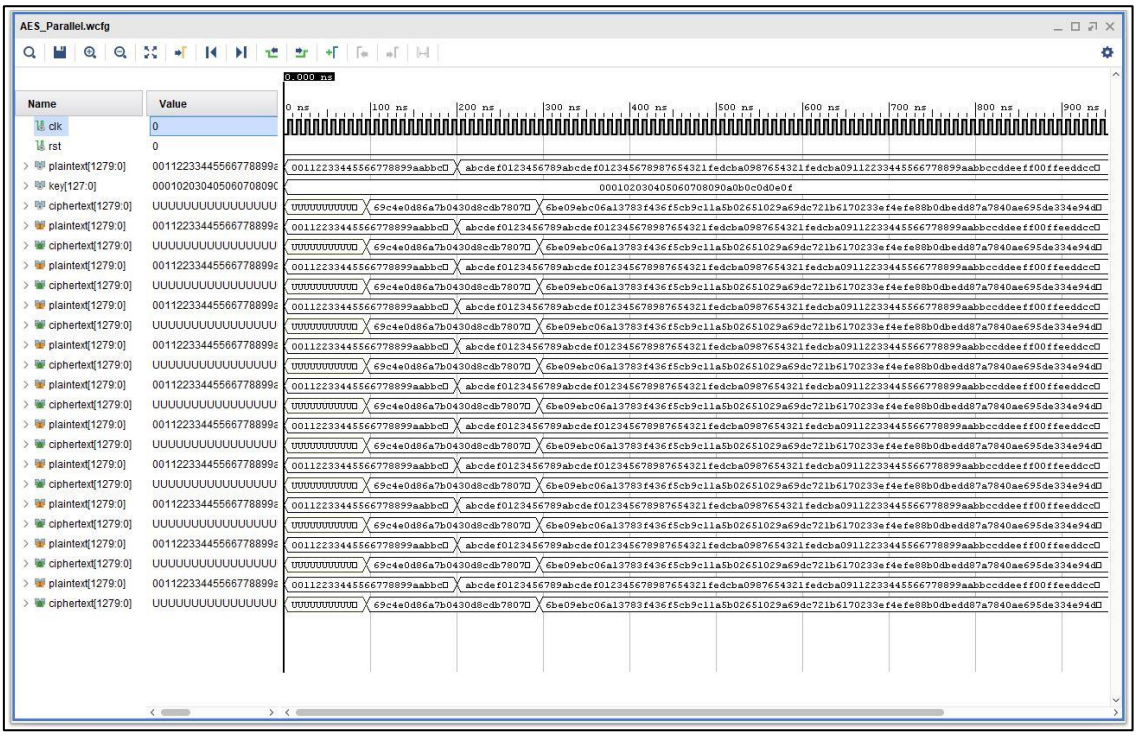


Fig. 4.25 Simulation results of parallel structure.

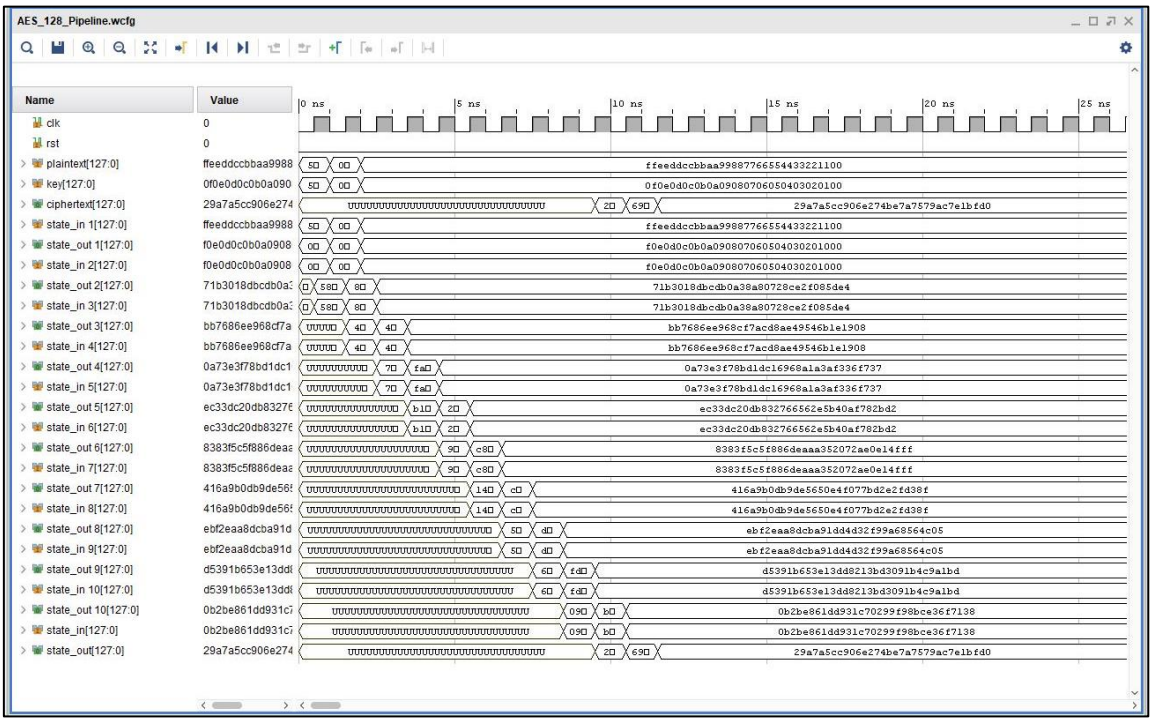


Fig. 4.26 Simulation results of pipeline structure.

FPGA-in-the-Loop (FiL) is a common and efficient technique to validate the functionality of a developed system. This technique involves connecting the FPGA to an external system to exchange data via standard interfacing protocols such as Universal Asynchronous Receiver-Transmitter (UART), Ethernet, or PCIe. The external system takes in charge data formatting, metadata separation,

concatenation after encryption, and data decryption. Based on this definition, Fig. 4.27 illustrates a simple FiL schematic of the AES core.

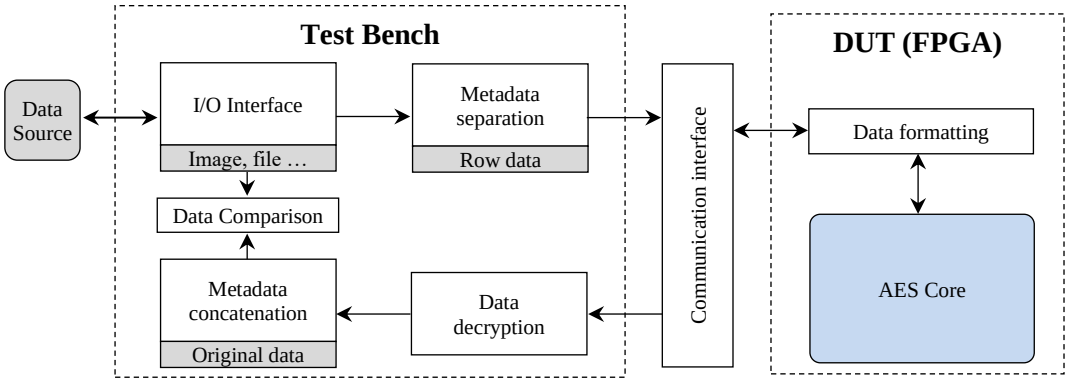


Fig. 4.27 FPGA In The Loop diagram

The major advantage of this validation method is that it isolates the Design Under Test (DUT), allowing it to focus on its primary task and necessary processing, while offloading validation to the test bench. The test bench includes dedicated applications for visualization, real-time verification, and comparison mechanisms. By reducing the need for full prototyping, this method helps catch design flaws early, saving time and costs. FiL is widely used in environments like MATLAB/Simulink, making it easier to integrate with control algorithms, DSP applications, and AI accelerators.

4.9. System Integration

The pipeline structure is designed for general purpose utilization, it accepts any format of data after converting it to binary. For example, an image can be encrypted after passing through a data separation process to separate metadata from raw data. The raw data goes then to the encryption system through ECB stream mode, and get concatenated again with the metadata. Similarly, any kind of information can be encrypted. Fig. 4.28 shows an illustration of the integration of encryption core within embedded system.

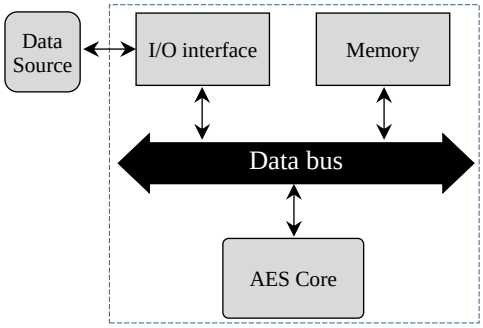


Fig. 4.28 System Integration of the proposed security core.

The AES core processes input data and provides output ciphertext regardless the type of original information, making it well adapted to general purpose embedded systems.

4.10. Conclusion

In this chapter, efficient techniques implementing AES in FPGA are developed to mainly address resource-limitation and timing-performance constraints in embedded systems. Iterative and pipeline architectures are built and then compared in terms of throughput-occupied area ratio. The considered configurations differ in the global structures, but use the same round module design aiming to perform a valid comparison between them.

Although the iterative system has led to the lowest efficiency ratio compared the pipeline configurations, it stands for its lightest implementation, making it well suitable for resource constrained systems. Simulation results have shown, in steady data flow, a superiority of the pipeline structure making suitable for systems in which timing is of paramount importance.

General conclusion

This thesis explores the key challenges of implementing security algorithms within resource-constrained embedded systems, with an in-depth discussion of various security techniques. The research includes a background study on cryptographic algorithms, and a literature review of related works on optimizing security algorithms. Previous optimizations projects of security algorithms are categorized into two types: the first focuses on modifying the algorithm by reducing complex operations and altering or removing certain stages, whereas the second approach centers on implementation techniques without modifying the algorithm itself. The proposed project addresses the challenge of preserving the integrity of the original AES algorithm, while developing efficient FPGA architectures. To this end, the final section presents two innovative implementation techniques, designed specifically to tackle resource limitations and timing performance constraints in embedded systems. Following the project presentation, the construction and comparison of iterative and pipeline architectures in terms of throughput, occupied area, and efficiency ratio are discussed, where the configurations differ in overall structure while utilizing the same round module design, enabling a fair and valid comparison between them. The observed metrics include area utilization, timing performance, and efficiency ratio. The pipeline architecture proved to be the most efficient, which aligns with the FPGA optimization strategies presented in the third section of this thesis. This section identifies pipelining as a key optimization strategy for addressing common design issues. The robustness of the proposed pipeline architecture is proven as capable of encrypting all types of data regardless of the final application, thus making it adaptable to various applications, whether as a cipher block or in chaining modes.

Reference List

[1] P. Marwedel, “Embedded System Hardware,” in *Embedded System Design*, in Embedded Systems. , Cham: Springer International Publishing, 2018, pp. 125–196. doi: 10.1007/978-3-319-56045-8_3.

[2] P. Marwedel, “Optimization,” in *Embedded System Design*, in Embedded Systems. , Cham: Springer International Publishing, 2018, pp. 337–366. doi: 10.1007/978-3-319-56045-8_7.

[3] X. Fan, K. Mandal, and G. Gong, “WG-8: A Lightweight Stream Cipher for Resource-Constrained Smart Devices,” in *Quality, Reliability, Security and Robustness in Heterogeneous Networks*, vol. 115, K. Singh and A. K. Awasthi, Eds., in Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 115. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 617–632. doi: 10.1007/978-3-642-37949-9_54.

[4] X. Zhang, S. Tang, T. Li, X. Li, and C. Wang, “GFRX: A New Lightweight Block Cipher for Resource-Constrained IoT Nodes,” *Electronics*, vol. 12, no. 2, p. 405, Jan. 2023, doi: 10.3390/electronics12020405.

[5] L. Yan, L. Li, and Y. Guo, “DBST: a lightweight block cipher based on dynamic S-box,” *Front. Comput. Sci.*, vol. 17, no. 3, p. 173805, Jun. 2023, doi: 10.1007/s11704-022-1677-5.

[6] D. Singh, M. Kumar, and T. Yadav, “RAZOR A Lightweight Block Cipher for Security in IoT,” *Def. Sc. J.*, vol. 74, no. 01, pp. 46–52, Jan. 2024, doi: 10.14429/dsj.74.18421.

[7] R. A. Ramadan, B. W. Aboshosha, K. Yadav, I. M. Alseadoon, M. J. Kashout, and M. Elhoseny, “LBC-IoT: Lightweight Block Cipher for IoT Constraint Devices,” *Computers, Materials & Continua*, vol. 67, no. 3, pp. 3563–3579, 2021, doi: 10.32604/cmc.2021.015519.

[8] A. G. Buja and S. F. A. Latip, “The Direction of Lightweight Ciphers in Mobile Big Data Computing,” *Procedia Computer Science*, vol. 72, pp. 469–476, 2015, doi: 10.1016/j.procs.2015.12.128.

[9] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, “spongent: A Lightweight Hash Function,” in *Cryptographic Hardware and Embedded Systems – CHES 2011*, vol. 6917, B. Preneel and T. Takagi, Eds., in Lecture Notes in Computer Science, vol. 6917. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 312–325. doi: 10.1007/978-3-642-23951-9_21.

[10] S. Windarta, S. Suryadi, K. Ramli, B. Pranggono, and T. S. Gunawan, “Lightweight Cryptographic Hash Functions: Design Trends, Comparative Study, and Future Directions,” *IEEE Access*, vol. 10, pp. 82272–82294, 2022, doi: 10.1109/ACCESS.2022.3195572.

[11] A. Sevin and Ü. Çavuşoğlu, “Design and Performance Analysis of a SPECK-Based Lightweight Hash Function,” *Electronics*, vol. 13, no. 23, p. 4767, Dec. 2024, doi: 10.3390/electronics13234767.

[12] J. Guo, T. Peyrin, and A. Poschmann, “The PHOTON Family of Lightweight Hash Functions,” in *Advances in Cryptology – CRYPTO 2011*, vol. 6841, P. Rogaway, Ed., in Lecture Notes in Computer Science, vol. 6841. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–239. doi: 10.1007/978-3-642-22792-9_13.

[13] M. Nabil, A. A. M. Khalaf, and S. M. Hassan, “Design and implementation of pipelined and parallel AES encryption systems using FPGA,” *IJEECS*, vol. 20, no. 1, p. 287, Oct. 2020, doi: 10.11591/ijeecs.v20.i1.pp287-299.

[14] I. Algredo-Badillo, K. A. Ramírez-Gutiérrez, L. A. Morales-Rosales, D. Pacheco Bautista, and C. Feregrino-Uribe, “Hybrid Pipeline Hardware Architecture Based on Error Detection and Correction for AES,” *Sensors*, vol. 21, no. 16, p. 5655, Aug. 2021, doi: 10.3390/s21165655.

[15] R. P. and M. H., “Design and implementation of power and area optimized AES architecture on FPGA for IoT application,” *CW*, vol. 47, no. 2, pp. 153–163, Jun. 2021, doi: 10.1108/CW-04-2019-0039.

[16] T. Kumar, K. Reddy, S. Rinaldi, B. Parameshachari, and K. Arunachalam, “A Low Area High Speed FPGA Implementation of AES Architecture for Cryptography Application,” *Electronics*, vol. 10, no. 16, p. 2023, Aug. 2021, doi: 10.3390/electronics10162023.
[17] S. J. H. Pirzada, M. N. Hasan, Z. W. Memon, M. Haris, T. Xu, and L. Jianwei, “High-Throughput Optimizations of AES Algorithm for Satellites,” in *2020 International Symposium on Recent Advances in Electrical Engineering & Computer Sciences (RAEE & CS)*, Islamabad, Pakistan: IEEE, Oct. 2020, pp. 1–6. doi: 10.1109/RAEECS50817.2020.9265688.
[18] A. Li and Y. Pan, “A Theory of Network Security: Principles of Natural Selection and Combinatorics,” *Internet Mathematics*, vol. 12, no. 3, pp. 145–204, May 2016, doi: 10.1080/15427951.2015.1098755.
[19] A. Shoufan and S. A. Huss, “High-Performance Rekeying Processor Architecture for Group Key Management,” *IEEE Trans. Comput.*, vol. 58, no. 10, pp. 1421–1434, Oct. 2009, doi: 10.1109/TC.2009.88.
[20] G. Savva, K. Manousakis, and G. Ellinas, “Providing Confidentiality in Optical Networks: Metaheuristic Techniques for the Joint Network Coding-Routing and Spectrum Allocation Problem,” in *2020 22nd International Conference on Transparent Optical Networks (ICTON)*, Bari, Italy: IEEE, Jul. 2020, pp. 1–4. doi: 10.1109/ICTON51198.2020.9203018.
[21] T. O. Oladoyinbo, S. O. Olabanji, O. O. Olaniyi, O. O. Adebiyi, O. J. Okunleye, and A. I. Alao, “Exploring the Challenges of Artificial Intelligence in Data Integrity and its Influence on Social Dynamics,” *AJARR*, vol. 18, no. 2, pp. 1–23, Jan. 2024, doi: 10.9734/ajarr/2024/v18i2601.
[22] I. Verbauwhede and P. Schaumont, “Design methods for Security and Trust,” in *2007 Design, Automation & Test in Europe Conference & Exhibition*, Nice, France: IEEE, Apr. 2007, pp. 1–6. doi: 10.1109/DATE.2007.364671.
[23] Z. Cui *et al.*, “A Hybrid BlockChain-Based Identity Authentication Scheme for Multi-WSN,” *IEEE Trans. Serv. Comput.*, pp. 1–1, 2020, doi: 10.1109/TSC.2020.2964537.
[24] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 1st ed. CRC Press, 2018. doi: 10.1201/9781439821916.
[25] A. Sadeghi-Nasab and V. Rafe, “A comprehensive review of the security flaws of hashing algorithms,” *J Comput Virol Hack Tech*, vol. 19, no. 2, pp. 287–302, Oct. 2022, doi: 10.1007/s11416-022-00447-w.
[26] H. N. Noura, A. Chehab, and R. Couturier, “Overview of Efficient Symmetric Cryptography: Dynamic vs Static Approaches,” in *2020 8th International Symposium on Digital Forensics and Security (ISDFS)*, Beirut, Lebanon: IEEE, Jun. 2020, pp. 1–6. doi: 10.1109/ISDFS49300.2020.9116441.
[27] S. Oukili and S. Bri, “High throughput FPGA Implementation of Data Encryption Standard with time variable sub-keys,” *IJECE*, vol. 6, no. 1, p. 298, Feb. 2016, doi: 10.11591/ijece.v6i1.pp298-306.
[28] C. Atika Sari, E. H. Rachmawanto, and C. A. Haryanto, “Cryptography Triple Data Encryption Standard (3DES) for Digital Image Security,” *SJI*, vol. 5, no. 2, pp. 105–117, Nov. 2018, doi: 10.15294/sji.v5i2.14844.
[29] G. Paul and S. Maitra, *RC4 Stream Cipher and Its Variants*, 0 ed. CRC Press, 2011. doi: 10.1201/b11310.
[30] J. Daemen and V. Rijmen, “The Block Cipher Rijndael,” in *Smart Card Research and Applications*, vol. 1820, J.-J. Quisquater and B. Schneier, Eds., in Lecture Notes in Computer Science, vol. 1820. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 277–284. doi: 10.1007/10721064_26.
[31] J. M. Koshy, “Introduction Advanced Encryption Standard (AES),” ScienceOpen. Accessed: Jan. 04, 2025. [Online]. Available: https://scienceopen.com/hosted-document?doi=10.14293/S2199-1006.1.SOR-.PPBWB9Z.v1
[32] G. Singh and S. Supriya, “A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security,” *IJCA*, vol. 67, no. 19, pp. 33–38, Apr. 2013, doi: 10.5120/11507-7224.

[33] M. Fischlin, C. Janson, and S. Mazaheri, “Backdoored Hash Functions: Immunizing HMAC and HKDF,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, Oxford: IEEE, Jul. 2018, pp. 105–118. doi: 10.1109/CSF.2018.00015.
[34] F. Bourse, D. Pointcheval, and O. Sanders, “Divisible E-Cash from Constrained Pseudo-Random Functions,” in *Advances in Cryptology – ASIACRYPT 2019*, vol. 11921, S. D. Galbraith and S. Moriai, Eds., in Lecture Notes in Computer Science, vol. 11921. , Cham: Springer International Publishing, 2019, pp. 679–708. doi: 10.1007/978-3-030-34578-5_24.
[35] F. Chen and J. Yuan, “Enhanced Key Derivation Function of HMAC-SHA-256 Algorithm in LTE Network,” in *2012 Fourth International Conference on Multimedia Information Networking and Security*, Nanjing, China: IEEE, Nov. 2012, pp. 15–18. doi: 10.1109/MINES.2012.106.
[36] C. C. Wen, E. Dawson, and L. Simpson, “Stream cipher based key derivation function,” *IJSN*, vol. 12, no. 2, p. 70, 2017, doi: 10.1504/IJSN.2017.083813.
[37] B. Sankhyan, “Review on Symmetric and Asymmetric Cryptography,” *IJRASET*, vol. 12, no. 3, pp. 2934–2940, Mar. 2024, doi: 10.22214/ijraset.2024.59538.
[38] M. S. A. Mohamad, R. Din, and J. I. Ahmad, “Research trends review on RSA scheme of asymmetric cryptography techniques,” *Bulletin EEI*, vol. 10, no. 1, pp. 487–492, Feb. 2021, doi: 10.11591/eei.v10i1.2493.
[39] Y. Cheng, Y. Liu, Z. Zhang, and Y. Li, “An Asymmetric Encryption-Based Key Distribution Method for Wireless Sensor Networks,” *Sensors*, vol. 23, no. 14, p. 6460, Jul. 2023, doi: 10.3390/s23146460.
[40] P. K. Panda and S. Chattopadhyay, “A hybrid security algorithm for RSA cryptosystem,” in *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, India: IEEE, Jan. 2017, pp. 1–6. doi: 10.1109/ICACCS.2017.8014644.
[41] J. H. Seo, “Efficient digital signatures from RSA without random oracles,” *Information Sciences*, vol. 512, pp. 471–480, Feb. 2020, doi: 10.1016/j.ins.2019.09.084.
[42] X.-Q. Cai, T.-Y. Wang, C.-Y. Wei, and F. Gao, “Cryptanalysis of multiparty quantum digital signatures,” *Quantum Inf Process*, vol. 18, no. 8, p. 252, Aug. 2019, doi: 10.1007/s11128-019-2365-8.
[43] A. Saepulrohman and A. Ismangil, “Data integrity and security of digital signatures on electronic systems using the digital signature algorithm (DSA),” *IJECS*, vol. 1, no. 1, pp. 11–15, Jun. 2021, doi: 10.24042/ijecs.v1i1.7923.
[44] C.-M. Chen, Y. Huang, K.-H. Wang, S. Kumari, and M.-E. Wu, “A secure authenticated and key exchange scheme for fog computing,” *Enterprise Information Systems*, vol. 15, no. 9, pp. 1200–1215, Oct. 2021, doi: 10.1080/17517575.2020.1712746.
[45] C. Gupta and N. V. Subba Reddy, “Enhancement of Security of Diffie-Hellman Key Exchange Protocol using RSA Cryptography.,” *J. Phys.: Conf. Ser.*, vol. 2161, no. 1, p. 012014, Jan. 2022, doi: 10.1088/1742-6596/2161/1/012014.
[46] X. Hou and J. Breier, *Cryptography and embedded systems security*. Cham: Springer, 2024.
[47] P. Simpson, “Timing Closure,” in *FPGA Design*, New York, NY: Springer New York, 2010, pp. 107–132. doi: 10.1007/978-1-4419-6339-0_12.
[48] G. Stitt, R. Lysecky, and F. Vahid, “Dynamic hardware/software partitioning: a first approach,” in *Proceedings of the 40th annual Design Automation Conference*, Anaheim CA USA: ACM, Jun. 2003, pp. 250–255.
[49] K. Fujiwara, K. Kawamura, M. Yanagisawa, and N. Togawa, “Clock skew estimate modeling for FPGA high-level synthesis and its application,” in *2015 IEEE 11th International Conference on ASIC (ASICON)*, Chengdu, China: IEEE, Nov. 2015, pp. 1–4.
[50] G. Stitt, W. Piard, and C. Crary, “Low-Latency, Line-Rate Variable-Length Field Parsing for 100+ Gb/s Ethernet,” in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey CA USA: ACM, Apr. 2024, pp. 12–21.
[51] C. Crary, W. Piard, G. Stitt, C. Bean, and B. Hicks, “Using FPGA Devices to Accelerate Tree-Based Genetic Programming: A Preliminary Exploration with Recent Technologies,” in *Genetic*

Programming, vol. 13986, G. Pappa, M. Giacobini, and Z. Vasicek, Eds., in Lecture Notes in Computer Science, vol. 13986. , Cham: Springer Nature Switzerland, 2023, pp. 182–197.
[52] P. Simpson, "RTL Design," in *FPGA Design*, New York, NY: Springer New York, 2010, pp. 51–78.
[53] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, Boston, MA, USA: IEEE, May 2014, pp. 36–43. doi: 10.1109/FCCM.2014.23.
[54] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*, 1st ed. Wiley, 2007. doi: 10.1002/9780470127896.
[55] T. Feller, "Design Security and Cyber-Physical Threats," in *Trustworthy Reconfigurable Systems*, Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 61–84. doi: 10.1007/978-3-658-07005-2_4.
[56] A. U. - and A. B. K. -, "Implementation of AES Algorithm," *IJFMR*, vol. 5, no. 2, p. 1766, Mar. 2023, doi: 10.36948/ijfmr.2023.v05i02.1766.
[57] M. A. Rabbi Emon *et al.*, "Advanced Encryption Standard for embedded applications: An FPGA-based implementation using VHDL," in *2021 3rd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, Agadir, Morocco: IEEE, Dec. 2021, pp. 120–124. doi: 10.1109/MENACOMM50742.2021.9678241.
[58] Department of Electronics and Communications, Faculty of Engineering, Minia University, Minia, Egypt., M. Nabil*, A. A. M. Khalaf, Department of Electronics and Communications, Faculty of Engineering, Minia University, Minia, Egypt., S. M. Hassan, and Department of Electronics and Communications, Faculty of Engineering, Modern Academy, Cairo, Egypt., "Design and Implementation of Pipelined AES Encryption System using FPGA," *IJRTE*, vol. 8, no. 5, pp. 2565–2571, Jan. 2020, doi: 10.35940/ijrte.E6475.018520.
[59] G. Y. Yen, S. Z. M. Naziri, R. C. Ismail, M. N. M. Isa, and R. Hussin, "Design of Multiplicative Inverse Value Generator using Logarithm Method for AES Algorithm," in *2020 32nd International Conference on Microelectronics (ICM)*, Aqaba, Jordan: IEEE, Dec. 2020, pp. 1–5. doi: 10.1109/ICM50269.2020.9331497.
[60] S. Liu, Y. Li, and Z. Jin, "Research on Enhanced AES Algorithm Based on Key Operations," in *2023 IEEE 5th International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, Dali, China: IEEE, Oct. 2023, pp. 318–322. doi: 10.1109/ICCASIT58768.2023.10351719.
[61] A. Tripathy and B. Singh, "A Study of AES Software Implementation for IoT Systems," in *2022 3rd International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Ghaziabad, India: IEEE, Nov. 2022, pp. 1–4. doi: 10.1109/ICICT55121.2022.10064507.
[62] P. Bulens, F.-X. Standaert, J.-J. Quisquater, P. Pellegrin, and G. Rouvroy, "Implementation of the AES-128 on Virtex-5 FPGAs," in *Progress in Cryptology – AFRICACRYPT 2008*, vol. 5023, S. Vaudenay, Ed., in Lecture Notes in Computer Science, vol. 5023. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 16–26. doi: 10.1007/978-3-540-68164-9_2.
[63] U. Hussain and H. Jamal, "An Efficient High Throughput FPGA Implementation of AES for Multi-gigabit Protocols," in *2012 10th International Conference on Frontiers of Information Technology*, Islamabad, Pakistan: IEEE, Dec. 2012, pp. 215–218. doi: 10.1109/FIT.2012.45.
[64] S. J. H. Pirzada, M. N. Hasan, Z. W. Memon, M. Haris, T. Xu, and L. Jianwei, "High-Throughput Optimizations of AES Algorithm for Satellites," in *2020 International Symposium on Recent Advances in Electrical Engineering & Computer Sciences (RAEE & CS)*, Islamabad, Pakistan: IEEE, Oct. 2020, pp. 1–6. doi: 10.1109/RAEECS50817.2020.9265688.
[65] U. Hussain and H. Jamal, "An Efficient High Throughput FPGA Implementation of AES for Multi-gigabit Protocols," in *2012 10th International Conference on Frontiers of Information Technology*, Islamabad, Pakistan: IEEE, Dec. 2012, pp. 215–218. doi: 10.1109/FIT.2012.45.

Résumé

La vie moderne dépend fortement des systèmes embarqués, qui, malgré leur nature compacte et à ressources limitées, constituent le noyau intelligent derrière la plupart des appareils. Ces systèmes intelligents collectent des données utilisateur pour améliorer continuellement les processus de prise de décision. Cependant, les données collectées incluent souvent des informations sensibles, soulignant le besoin crucial de mesures de sécurité robustes pour garantir la fiabilité et la confiance sans compromettre les performances. Ce projet vise à identifier une technique d'implémentation optimale pour les algorithmes de sécurité dans les systèmes embarqués, en équilibrant l'utilisation des ressources et les performances temporelles. Le défi principal abordé est de maintenir l'intégrité de l'algorithme de sécurité tout en optimisant l'architecture matérielle pour maximiser l'efficacité. L'algorithme Rijndael, plus précisément la norme de chiffrement avancé (AES), est choisi comme noyau de sécurité et est implémenté en utilisant deux techniques distinctes. L'approche proposée est conçue pour les réseaux de portes programmables (FPGA), avec des résultats comparés à des projets similaires en termes de performances temporelles et de surface.
Mots-clés : Algorithme Advanced Encryption Standard (AES), FPGA (Field Programmable Gate Array), cryptographie, systèmes embarqués,

ملخص

تعتمد الحياة الحديثة بشكل كبير على الأنظمة المدمجة، والتي، على الرغم من طبيعتها الصغيرة والمحدودة الموارد، تُعد بمثابة العقل المدبر وراء معظم الأجهزة. تقوم هذه الأنظمة الذكية بجمع بيانات المستخدم لتحسين عمليات اتخاذ القرار بشكل مستمر. ومع ذلك، فإن البيانات المجمعة غالبًا ما تتضمن معلومات حساسة، مما يبرز الحاجة الماسة إلى إجراءات أمنية قوية لضمان الموثوقية والثقة دون المساس بالأداء. يهدف هذا المشروع إلى تحديد تقنية تنفيذ مثالية لخوارزميات الأمان في الأنظمة المدمجة، مع تحقيق التوازن بين استخدام الموارد وأداء التوقيت. التحدي الرئيسي الذي يتم معالجته في هذا المشروع هو الحفاظ على سلامة خوارزمية الأمان مع تحسين بنية العتاد لتعظيم الكفاءة. تم اختيار خوارزمية Rijndael، وتحديدًا Advanced Encryption Standard (AES)، كنواة أمنية يتم تنفيذها باستخدام تقنيتين مختلفتين. تم تصميم النهج المقترح ليعمل على مصفوفات البوابات القابلة للبرمجة (FPGAs)، مع مقارنة النتائج بمشاريع مماثلة من حيث أداء التوقيت والمساحة.
الكلمات المفتاحية: خوارزمية معيار التشفير المتقدم (AES) ، مصفوفة البوابة الميدانية القابلة للبرمجة (FPGA) ، التشفير، الأنظمة المدمجة.

Abstract

Modern life relies heavily on embedded systems, which, despite their compact and resource-constrained nature, serve as the core intelligence behind most devices. These smart systems collect user data to continuously improve decision-making processes. However, the gathered data often includes sensitive information, highlighting the critical need for robust security measures that ensure trustworthiness and reliability without compromising performance. This project aims to identify an optimal implementation technique for security algorithms in embedded systems, balancing resource utilization and timing performance. The primary challenge addressed is maintaining the integrity of the security algorithm while optimizing hardware architecture to maximize efficiency. The Rijndael algorithm, specifically the Advanced Encryption Standard (AES), is chosen as the security core and implemented using two distinct techniques. The proposed approach is designed for Field-Programmable Gate Arrays (FPGAs), with results benchmarked against similar projects in terms of timing and area performance
Keywords: Advanced Encryption Standard (AES) algorithm, Field Programmable Gate Array (FPGA), Cryptography, embedded systems