Implementation of Lightweight Cryptography Algorithms in FPGAs

Oikonomou Asterios

Thesis Committee:

Associate Professor Sotirios Ioannidis (supervisor)
Professor Apostolos Dollas
Professor George N. Karystinos

School of Electrical and Computer Engineering

Technical University of Crete

October 11, 2022

Abstract

IoT applications typically involve a tree-like structure of devices, with the top levels occupied by a small number of highly complex powerful machines and a plethora of more restricted in computational and energy resources devices as one moves to the bottom levels. While cost and physical size restrictions mandate that those limited devices employ simplistic computing resources with small capacity batteries, the penetration of IoT applications in all forms of activities has multiplied the computational tasks (both in number and complexity) that have to be executed, increasing the pressure to use highly computationally and energy efficient devices and implementations of algorithms to be executed on their hardware. One of those tasks is data encryption, employed as one of the fundamental means to secure the exchange and storage of sensitive information. Lightweight encryption algorithms have been proposed as a good balance between data protection and computational complexity, enabling simple devices to handle those tasks. In this diploma thesis, three such algorithms are examined (Clefia, Simon and Present) focusing on devices that are placed at the edge of the IoT infrastructure. These devices communicate with a significant number of leaf-nodes and concentrate the traffic between the leaves and the higher levels of infrastructure. While they retain much of the limitations of the lower-levels of the IoT architecture (low cost, small size and limited energy resources), they have to provide significant computational performance and this work aims to explore the implementation of the encryption algorithm in custom reconfigurable hardware in order to achieve higher performance than common CPU-based systems and even higher energy efficiency. A comparison is therefore made between implementations of the three algorithms in software and hardware Specifically, the three algorithms were tested in software on four different processors(ARM Cortex-A9, ARM Cortex-A53, Intel Core I5-4200U and Intel Xeon E5-2630) and for the hardware at Xilinx KRIA KV260 FPGA. The results of the measurements show that Present block cipher is 1,473 times faster when applied on hardware compared to the Intel Xeon E5-2630. The same comparison is made with the others algorithms where Simon cipher appears to 29 times faster when applied to the hardware while Clefia block cipher is 252 times faster when applied to the hardware compared to the Intel Xeon processor E5-2630 .

Περίληψη

Οι εφαρμογές IoT περιλαμβάνουν συνήθως μια δομή συσκευών που μοιάζει με δέντρο, όπου τα υψηλότερα επίπεδα καταλαμβάνονται από ενα μικρό αριθμό εξαιρετικά πολύπλοκων ισχυρών μηχανών και απο πληθώρα συσκευών με περιορισμένους πόρους ως προς την κατανάλωση ενέργεια και την πολυπλεξία των υπολογισμών καθώς κινούμαστε στα χαμηλότερα επίπεδα. Ενώ το κόστος και οι περιορισμοί φυσικού μεγέθους απαιτούν αυτές οι συσκευές να χρησιμοποιούν απλούς υπολογιστικούς πόρους με μπαταρίες χαμηλής χωρητικότητας. Η διείσδυση των εφαρμογών του Διαδικτύου σε κάθε μορφή δραστηριότητας έχει πολλαπλασιάσει τις υπολογιστικές εργασίες(τόσο σε αριθμό όσο και σε πολυπλοκότητα) που πρέπει να εκτελεστούν, αυξάνοντας την πίεση για χρήση συσκευών με υψηλή ενεργειακή και υπολογιστική απόδοση αλλά και εφαρμογών, αλγορίθμων που πρέπει να εκτελεστούν στο δικό τους υλικό. Μία από αυτές τις εργασίες είναι η κρυπτογράφηση δεδομένων, που χρησιμοποιείται ως ένα από τα θεμελιώδη μέσα για την εξασφάλιση της ανταλλαγής και αποθήκευσης ευαίσθητων πληροφοριών. Οι αλγόριθμοι ελαφριάς κρυπτογράφησης έχουν προταθεί καθώς εξασφαλίζουν μια ισορροπία μεταξύ προστασίας δεδομένων και υπολογιστικής πολυπλοκότητας, επιτρέποντας απλές συσκευές για να χειριστούν τις εργασίες αυτές. Στην παρούσα διπλωματική εργασία τρεις τέτοιοι αλγόριθμοι εξετάζονται(Present, Simon και Clefia) εστιάζοντας σε συσκευές που βρίσκονται στην άκρη της υποδομής του IoT. Αυτές οι συσκευές επικοινωνούν με έναν σημαντικό αριθμό κόμβων φύλλων και συγκεντρώνουν την πληροφορία μεταξύ των φύλλων και των υψηλότερων επιπέδων υποδομής. Ενώ διατηρούν μεγάλο μέρος των περιορισμών που έχουν τα κατώτερα επίπεδα της αρχιτεκτονικής του IoT (χαμηλό κόστος μικρό μέγεθος και περιορισμένοι ενεργειακοί πόροι) παρέχουν σημαντική υπολογιστική απόδοση και αυτή η εργασία στοχεύει στην υλοποίηση των αλγορίθμων κρυπτογράφησης σε αναδιατασόμενη λογική με σκόπό την επίτευξη υψηλότερης απόδοσης σε σχέση με ένα σύστημα που βασίζεται σε CPU καθώς και ακόμη υψηλότερη ενεργειακή απόδοση. Γίνετε σύγκριση μεταξύ των υλοποιήσεων των τριών αλγορίθμων σε λογισμικό και σε υλικό. Συγκεκριμένα οι τρεις αλγόριθμοι εξετάστηκαν στο λογισμικό σε τέσσερις διαφορετικούς επεξεργαστές (ARM Cortex-A9, ARM Cortex-A53, Intel Core I5-4200U και Intel Xeon E5-2630) και για το υλικό υλοποιήθηκαν στην Xilinx KRIA KV260. Τα αποτελέσματα των μετρήσεων δείχνουν ότι ο αλγόριθμος Present είναι 1.473 φορές πιο γρήγορος όταν εφαρμόζεται στο υλικό από όταν εφαρμόζεται στον επεξεργαστή Intel Xeon E5-2630. Η ίδια σύγκριση γίνεται και με τους άλλους αλγόριθμους όπου ο Clefia εμφανίζεται να είναι 252 φορές πιο γρήγορος όταν εφαρμόζεται στο υλικό ενώ ο Simon είναι 29 φορές πιο γρήγορος όταν εφαρμόζεται στο υλικό σε σύγκριση με τον επεξεργαστή Intel Xeon E5-2630.

Acknowledgements

At this point I would like to express my gratitude and appreciation to Prof. Sotirios Ioannidis for being my supervisor and for giving me the opportunity to do my Research and to know so many knew things on the field of hardware. I would like to thank Andreas Brokalakis for his guidance and for all his support during my thesis. I would like also to thank Prof. Apostolos Dollas and Prof. George N. Karystinos for being members of the committee and for assessing my thesis. Finally, I am grateful for my family and my friends for trusting and supporting me over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet-of-Things (IoT) is a paradigm that connects various physical devices to the Internet through various wireless technologies. In recent decades, the IoT(smart environment) has taken a large share in the development of the technology. An attempt to transform machine-to-machine communication into the IoT has brought about the greatest revolution in the current human era. The vision of IoT is to create a heterogeneous network of millions of connected objects that communicate securely over the Internet. IoT has become an integral part of socioeconomically growth as it has various domain-specific applications in different fields. These include healthcare, surveillance, transportation, security, manufacturing, environmental monitoring, food processing as shown in Fig 1.1.

In IoT applications, the computing and power resources are limited especially at the edge where the sensors and other very simple devices are generally deployed. At the same time, as IoT applications are increasingly involved in areas where sensitive information is gathered, stored and exchanged, a higher level of security and data protection must be provided. Typical methods of securing data transmissions and distributed storage is to encrypt all sensitive data. There are a number of different cryptographic algorithms that serve different purposes: low-complexity algorithms for weak computing devices, low-latency, high-throughput solutions for performance-critical applications, enhanced attack-resistance algorithms and features for critical data, etc [6].

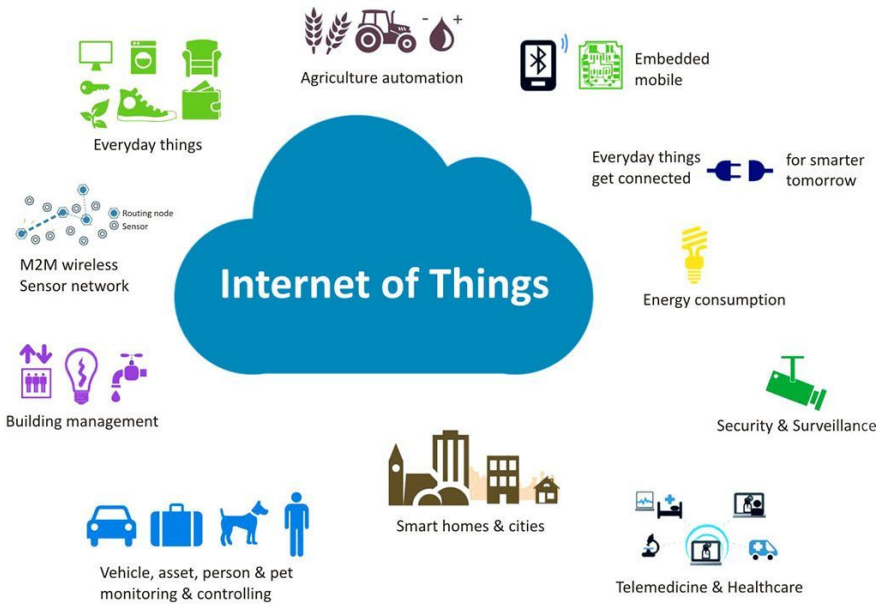


Figure 1.1: Applications of Internet-of-Things[1]

Considering the very broad scope of applications and devices used in IoT environments it is hard to select a "one-size-fits-all" solution. Additionally, having the ability to switch between different algorithms is beneficial for both security (commonly refereed to as Moving Target Defense or MTD) and efficiency. Two solutions are currently available. The first one sacrifices flexibility and adaptability by choosing to support a single algorithm (or family algorithms with common computational characteristics) in hardware in order to gain performance and achieve low energy consumption. The second retains all flexibility and security resilience by implementing all required algorithms in software with significant impact in performance and efficiency.

For this thesis, we adopt a middle ground solution that aims to retain the advantages of the aforementioned solutions (high performance, low energy consumption and flexibility) without their drawbacks by opting for an accelerated implementation of the cryptographic algorithms in re-configurable hardware. This way, the performance and low energy consumption of a hardwired implementation may be mostly retained, while the reconfigurability of the hardware allows for easy and fast adaptation to different algorithms and implementations. Three lightweight cryptographic algorithms are selected with different characteristics. More specifically, we focus on Clefia, Simon and Present ciphers. All ciphers are well-established and recognised and cover generally

distinct fields of application. We plan to create a common API and interface to communicate between the software and the hardware components and a software library that exposes these accelerators to the software application and manages beyond the software/hardware communication, all hardware reconfiguration tasks of the FPGA resources (abstracting them from the user). We intent to use also a creation of interface at the hardware level will be employed.

1.1 About Cryptography

The enormous advances in network technology have resulted in an amazing potential for changing the way we communicate and do business over the Internet. However, the cost effectiveness and globalism that the Internet offers for the transmission of confidential data is offset by the main disadvantage of public networks: security risks. The rapidly increasing growth in confidential data traffic over the Internet makes the issue of security a fundamental problem. As a result, applications such as electronic banking, electronic commerce, and Virtual Private Networks (VPNs) require an efficient and cost-effective way to address security threats over public networks [7].

Cryptography is a security technique that encode messages in an unreadable form. In simple terms, it is nothing but a technique used to protect data during transmission from sender to receiver and denies unauthorized access. Therefore, security and confidentiality is very required in this aspect.[7] Cryptography is also the fundamental component to secure Internet traffic. However, cryptographic algorithms place enormous demands on processing power, which can be a bottleneck in high-speed networks.

In general there are three types of cryptography:

- Symmetric Key Cryptography.
- Asymmetric Key Cryptography.
- Hash Functions.

The implementation of a cryptographic algorithm must achieve a high processing rate in order to fully utilize the available network bandwidth. To keep up with the diversity and rapid changes in algorithms and standards, a cryptographic implementation must also support different algorithms and be up gradable in the field. Otherwise, interoperability between different systems is prohibited and any upgrade will result in

excessive costs. The ultimate solution to the problem would be an adaptive processor that can offer software-like flexibility with hardware-like performance.[8]

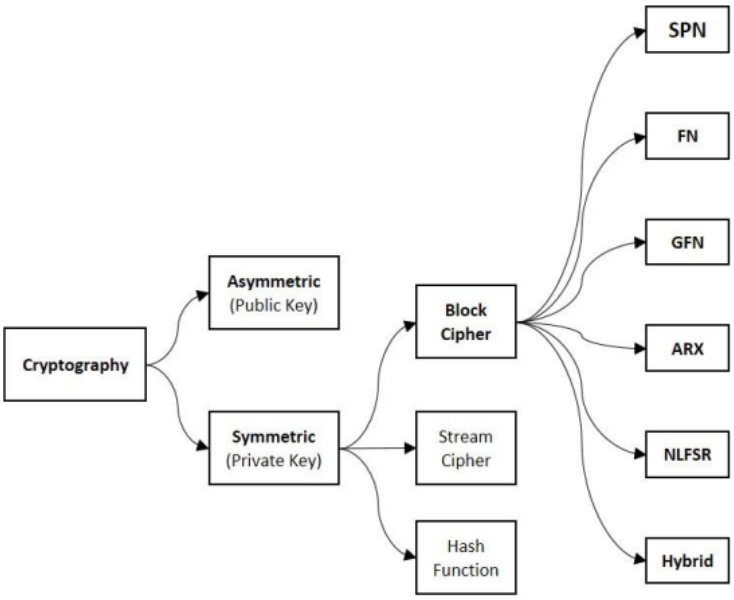


Figure 1.2: Cryptography Primitives

1.2 Symmetric Key Cryptography

Symmetric key cryptography is also known as secret key or shared key cryptography Fig 1.3. In this process, a sender and a receiver both share a common key through secret communication for both encryption and decryption. Symmetric cryptography is better suited for IoT applications due to its fast operations which are mainly XOR and permutations. The processing speed is faster and they don't consume many resources. Table 1.1 shows direct comparison between Asymmetric Key Cryptography and Symmetric Key Cryptography [2]. Mathematically, symmetric encryption may be considered as follows:

Definition 1. *A symmetric-key encryption scheme consists of a map*

$$E : K * M \rightarrow C$$

such that for each $k \in K$*, the map*

$$E_k : M \rightarrow C, m \rightarrow E(k, m)$$

is inevitable. The elements $m \in M$ are the plain-texts (also called messages). C is the set of cipher-texts or cryptograms, the elements $k \in K$ are the keys. E_k is called the encryption function with respect to the key k. The inverse function $D_k := E_k^{`1}$ is called the decryption function. It is assumed that efficient algorithms to compute E_k and D_k exist.

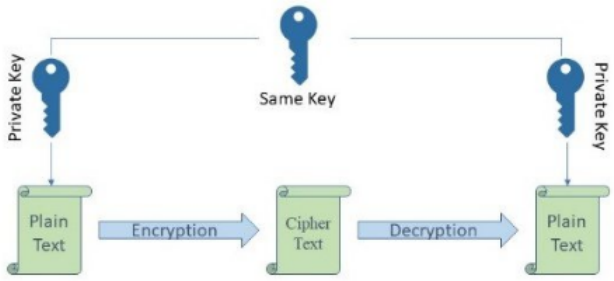


Figure 1.3: Symmetric Key Cryptography [2].

The key k is shared between the communication partners and kept secret. A basic security requirement for the encryption map E is that, without knowing the key k, it should be impossible to successfully perform the decryption function D_k. Among all encryption algorithms, symmetric-key encryption algorithms have the fastest hardware and software implementations. They are therefore very well-suited for encrypting large amounts of data. Public-key encryption methods are less efficient and therefore not suitable for large amounts of data. Thus, symmetric-key encryption and public-key encryption complement each other to provide practical cryptosystems [9]. A very important distinction in Symmetric Algorithms is among Stream and Block ciphers where analyzed at 1.6 subsection.

1.3 Asymmetric Key Cryptography

Asymmetric key cryptography is known as public key cryptography, because in this technique, a pair of public key and private key are needed Fig1.4. Recently the focus has shifted from lightweight cryptography to asymmetric key cryptography, but the results are not yet stable and fruitful like symmetric key cryptography. Lightweight asymmetric algorithms are complex to operate and not time efficient. The size of the operands and the relentless advance of attack models also make these algorithms vulnerable [10]. Some of the very important asymmetric algorithms are as follows:

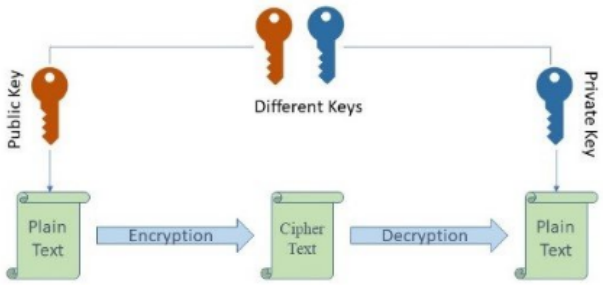


Figure 1.4: Asymmetric Key Cryptography [2].

Rivest-Shamir-Adleman (RSA), Diffie-Hellman, Digital Signature Algorithm (DSA), Elliptical Curve Cryptography (ECC).

- Rivest-Shamir Adleman (RSA) – The reverse procedure is very difficult for an attacker and it is also difficult to produce the private key from the public key. So, this method is highly secure, but key generation is complex, and the process is very slow.
- Diffie-Hellman – The private key is very short, as a result, the process is faster. Due to a short private key, it faces more attacks and the process is also vulnerable to man in the middle attacks.
- Digital Signature Algorithm (DSA) – This process is faster and more beneficial than other Asymmetric algorithms, but the digital signatures have short life span and the sharing is complicated [11].
- Elliptical Curve Cryptography (ECC) – Though it is more complex and difficult to implement, it consumes less power. Amongst the different types of Asymmetric algorithms ECC is most favorable for implementation in restricted devices [10].

ECC approach for IoT has become an important research topic, but mostly from software perspective. An implementation and evaluation of an open source ECC for the Contici OS for IoT has done by [12]. Their implementation is released under BSD license. ECC approach was applied by [13] and the paper presented an implementation of Zero Knowledge Protocol in an open source and generic programming library called Wiselib. A hardware approach was taken by [14] and they showed that an ECC computation can be efficiently protected against Side Channel Attacks. Their approach is well suited for lightweight implementations with a minimal security level and with a

limited hardware overhead. A comparative study was done by [15] among RSA, Diffie-Hellman and Elliptical Curve Cryptography with Diffie- Hellman (ECDH) and they have found ECDH is better than other algorithms in terms of power and area [2].

	Cryptography Methods	
	Asymmetric Key Cryptography	**Symmetric Key Cryptography**
Keys	A unique pair of private and public key	One shared private key
Number of keys	Linearly proportional to the number of users	Exponentially proportional to the number of users
Speed and Complexity	Because of different keys used, it requires more time to get the transmission done	Faster than asymmetric key cryptography
Hardware Complexity	More complex hardware implementation as it implements computational heavy algorithms which need more powerful hardware	Less complex hardware implementation as it implements algorithms with simple operations which need relatively inexpensive hardware
Use	Key Encryption and distributing keys, it provides Confidentiality	Bulk data encryption, encrypting files and communication paths, it provides Confidentiality and Authentication

Table 1.1: Comparison between asymmetric key cryptography and symmetric key cryptography [2]

1.4 Hash Functions

The term hash function has been used in computer science for a long time and refers to a function that compresses a string of any input into a fixed length string. However, if it meets some additional requirements, it can be used for cryptographic applications

and then called cryptographic hash functions. A cryptographic hash function (CHF) is a mathematical algorithm that maps data of an arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function for which it is practically infeasible to invert or reverse the computation [16]. Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Cryptography hash functions are a basic tool of modern cryptography.

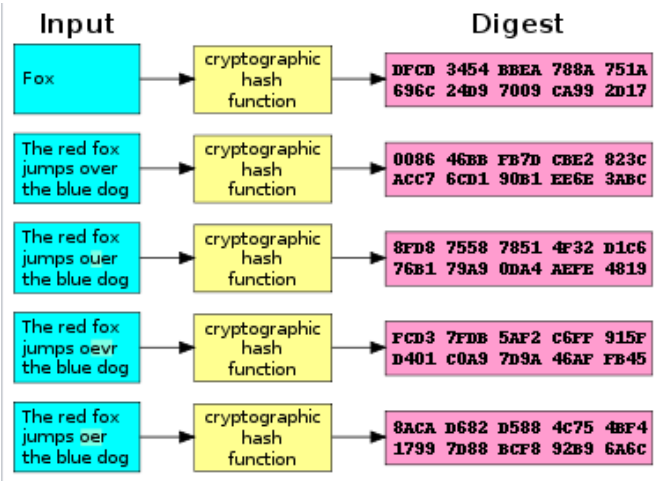


Figure 1.5: A cryptographic hash function at work [3].

A cryptographic hash function must be deterministic, meaning that the same message always results in the same hash. Ideally it should also have the following properties:

- It is quick to compute the hash value for any given message.
- It is infeasible to generate a message that yields a given hash value(i.e to reverse the process that generated the given hash value).
- It is infeasible to find two different messages with the same hash value.
- A small change to a message should change the hash value so extensively that a new hash value appears uncorrelated with the old hash value.

Cryptography hash functions have many information security applications, notably in digital signatures, message authentication codes, and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for

fingerprinting, to detect duplicate data or uniquely identify files, and as check sums to detect accidental data corruption. Indeed, in information-security contexts, cryptographic hash values are sometimes called (digital) fingerprints, check sums, or just hash values, even though all these terms stand for more general functions with rather different properties and purposes [17].

1.5 Lightweight Cryptography

Lightweight cryptography refers to cryptographic algorithms that are tailored for implementation in constrained devices (e.g. sensors, RFID cards, implantable devices etc). Lightweight cryptography contributes to the security of smart object networks and it is practical to use in these environments, offering a good balance between data protection and computational complexity (and subsequently low energy consumption). Furthermore, their implementation can be realised with few resources contributing to an overall reduction in cost compared to traditional cryptographic algorithms [18]. Lightweight cryptography can offer:

- Smaller block sizes(64-bit or less).
- Smaller key size(80-bit or less).
- Simple round logic based on simple computations.
- Simple key scheduling.
- Strong Structure(like SPN or FN)

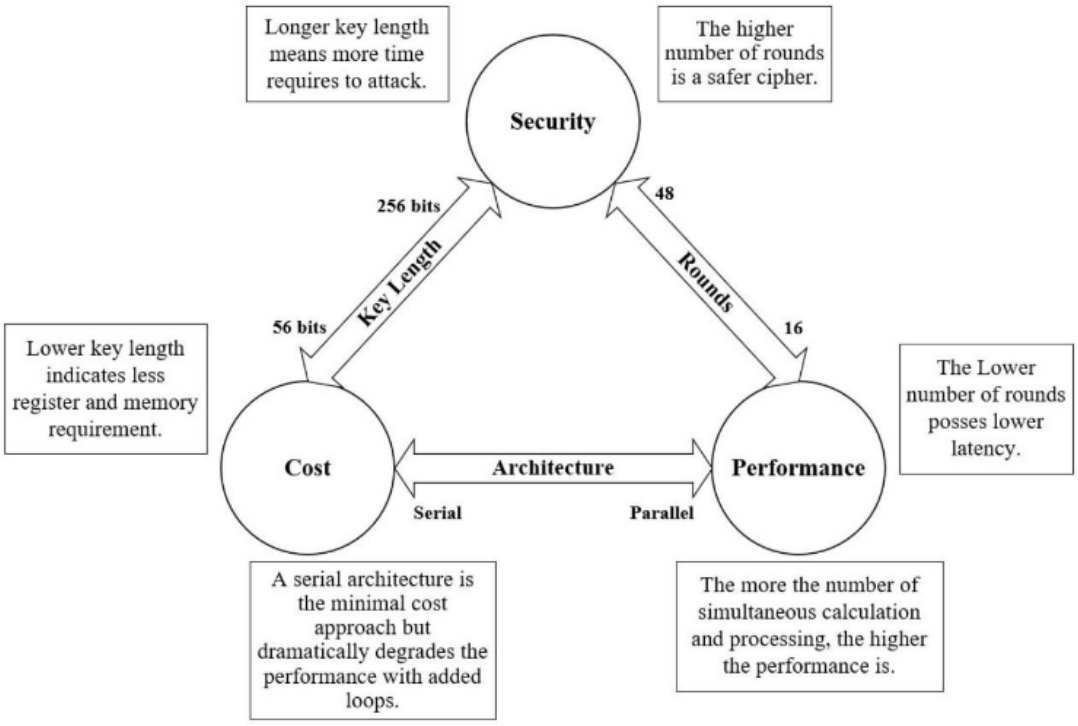


Figure 1.6: Lightweight Cryptography Performance [4]

When designing lightweight cryptography algorithms, a balance between cost, security and performance is sought. For example, in block encryption, the key length indicates the balance between security and cost, while in hardware implementation, number of rounds determines the balance between cost and performance. This problem is shown in Fig 1.6. Usually, any two of these three goals, such as security and cost, security and performance, or cost and performance can be easily achieved; however,it is very difficult to achieve all three goals at the same time. In order to fulfill the objectives mentioned in a lightweight cryptography algorithm, various methods have been proposed.[19].

Lightweight cryptography targets a wide variety of devices that can be implemented on a wide range of hardware and software. At the top end of the device spectrum are servers and desktop computers followed by tablets and smartphones. Traditional cryptographic algorithms can work well in these devices; therefore, these platforms may not require lightweight algorithms. Finally, at the lower end of the spectrum are devices such as embedded systems, RFID devices and sensor networks. Lightweight cryptography focuses primarily on the highly constrained devices found at the lower

end of this spectrum.[20]

The three main characteristics of lightweight cryptographic algorithms and their offerings are listed in Table 1.2.

Characteristics		**What LWC can offer?**
Physical(Cost)	Physical Area(GEs, logic blocks)	Smaller block sizes (64-bit or less)
	Memory (registers, RAM, ROM)	Smaller key size (80-bit or less)
	Battery power (energy consumption)	Simple round logic based on simple
Performance	Computing Power (latency, throughput)	computations
Security	Minimum security strength (bits)	Simple key scheduling
	Attack models (related key, multi- keys)	Strong Structure (like SPN or FNS)
	Side channel attack	

Table 1.2: LWC Characteristics

1.6 Block and Stream Cipher

Block cipher performs both encryption and decryption on a fixed-size block (64 bits or more) at the same time, while stream cipher processes the entire message byte by byte (8 bits at a time). Furthermore, the stream cipher uses only the confusion property (to make the relationship as complex as possible by using a substitution between the ciphertext and the key) while the block cipher uses both confusion and diffusion (to dissipate statistical structure of plaintext over bulk of ciphertext using permutation), two fundamental properties of cryptography introduced by Claude Shannon[21],[22], to strengthen the simple design cipher compared to the stream cipher. Reversing ciphertext is difficult in the block cipher while the stream cipher uses XOR for encryption which is easily into the plaintext.

For the above reasons, block cipher is preferred in resource-constrained IoT devices over stream cipher. A symmetric lightweight block cipher uses one of the following structure:

- Substitution-Permutation Network (SPN).
- Feistel Network (FN).
- General Feistel Network (GFN).

- Add-Rotate-XOR (ARX).
- NonLinear-Feedback Shift Register (NLFSR).
- Hybrid.

Substitution-Permutation network (SPN) processes data through a series of substitution (S-box) and permutation (table) by altering the data and finally formulating them for the next round. A **Feistel network (FN)** is a multi-round cipher that divides the input block into two parts and operates only on a half (diffusion) in each round of encryption or decryption. Between rounds, the left and right halves of the block are swapped. The **generalized Feistel network (GFN)** is a generalized form of the classical Feistel cipher. In GFN, input block is split into two or more sub-blocks and applies a (classical) Feistel transformation for every two sub blocks, and then performs a cyclic shift relevant to number of sub blocks [23]. **ARX** performs encryption-decryption using addition, rotation and XOR functions without making any use of S-box. Implementation of ARX is fast and compact but limits in security properties compared to SPN and Feistel ciphers. **Nonlinear feedback shift register (NLFSR)** can be applied in both stream cipher and block cipher designs. It utilizes the building blocks of stream ciphers whose current state is a nonlinear feedback function of its previous state. **Hybrid** cipher combine the any three types (SPN, FN, GFN, ARX, NLFSR) to improve specific characteristic (for example, throughput, energy, GE, etc.) based on its application requirements or even could mix of block and stream cipher.

Out of these structures, SPN and FN are the most popular choice due to their flexibility to implement the structure based on application requirements . Feistel structures can be implemented in low average power hardware, as a round function is applied to only one half of the state . On contrary, Feistel structures apply non-linearity in just one half of the state in each round, maintaining safety margins usually requires more round function compared to SPN structures. When there is choice between fewer SPN function rounds and higher Feistel function rounds with the same level of security and similar energy costs, SPN function could be a smarter choice [24].

1.7 Thesis Outline

- **Chapter 2 - Algorithms Description :** The three algorithms that are examined (Clefia, Present and Simon) are presented along with the main aspects of the computations involved in each one.
- **Chapter 3 - Platforms :** The platforms that have been used to implement these algorithms are presented.
- **Chapter 4 - FPGA Implementation :** Describes the hardware architectures that have been designed and their respective implementations.
- **Chapter 5 - Results :** In this chapter presents the results of our work and provide also existing works.
- **Chapter 6 - Conclusions :** The last chapter concludes the outcomes of this thesis.

Chapter 2

Algorithms Description

Several researches on performance and security of algorithm implementations of lightweight encryption in hardware and software are currently being published. For this work three encryption lightweight algorithms were implemented in hardware and software. More specifically, Present, Simon and Clefia were selected because these algorithms offer different characteristics as for the security, throughput and latency. All ciphers are well-established and recognised and cover generally distinct fields of application. This section, provides a description of the algorithms.

2.1 Present algorithm

2.1.1 Introduction of Present

Present is a lightweight cipher based on SP-network consisting of 31 rounds. The block length is 64 bits and there are two key lengths(one 80 bits and the second 128 bits). Each round passes through an S-box(Substitution Box)Layer and a P-Layer(Permutation Layer). The current block cipher is one of the most popular and leanest lightweight algorithms. The algorithm was proposed in 2007 and is now approved by the ISO/IEC standard for security reasons along with accepted performance and cost [25]. At the Table 2.1 we can see the characteristics of present block cipher.

block size	key size	rounds T
64 bit	80 bit	31
64 bit	128 bit	31

Table 2.1: Present block cipher operating parameters

When designing a block cipher suitable for extremely constrained environments, it

is important to realize that the PRESENT block cipher was not designed to be suitable for wide-spread use; we already have AES for that. Instead PRESENT targets some very specific applications for which AES is unsuitable. These generally correspond to the following properties[25].

- The cipher is to implemented in hardware.
- Applications will only require moderate security levels. Consequently, 80-bit security will be adequate.
- Applications are unlikely to require the encryption of large amounts of data.Implementations might therefore be optimised for performance or for space without too much practical impact.
- In some applications it is possible that the key will be fixed at the time of device manufacture. In such cases there would be no need to re-key a device (which would incidentally rule out a range of key manipulation attacks).
- – After security, the physical space required for an implementation will be the primary consideration. This is closely followed by peak and average power consumption, with the timing requirements being a third important metric.
- In applications that demand the most efficient use of space, the block cipher will often only be implemented as encryption-only. In this way it can be used within challenge-response authentication protocols and, with some careful state management, it could be used for both encryption and decryption of communications to and from the device by using the counter mode

Hardware implementation of **PRESENT** has much higher throughput and requires a half of gates compared to the implementation of the **AES** with similar key size[26],[25].

2.1.2 Present Encryption Block Cipher

PRESENT algorithm has 2 inputs: the plaintext(64-bits) and the key(80 or 128-bits) and one output the ciphertext as shown at Fig 2.1.

```
generateRoundKeys()
for i = 1 to 31 do
    addRoundKey(STATE,K_i)
    sBoxLayer(STATE)
    pLayer(STATE)
end for
addRoundKey(STATE,K_32)
```

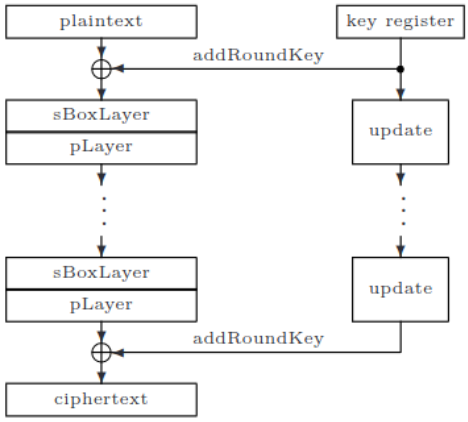


Figure 2.1: A top-level algorithmic description of present

Also the algorithm has 31 regular rounds and a final round that only consists of the key mixing step.One regular round consists of 3 basic processes:

- Key Scheduling
- AddRoundKey
- SPN

First, the encryption process applies AddRoundKey to the input block be encrypted. Plaintext values are xored with 64 MSB of the key.Then sBoxLayer is applied and each value of AddRoundKey is replaced with S-Box value. Then pLayer(Permutation Layer) to permutate the data. The output of this process is fed into the next round and the key values are calculated from the Key Scheduling part. The reverse process is performed for decryption of the data.

2.1.3 addRoundKey

Given round key $K_i = k^i_{63}...k^i_0$ for $1 \leq i \leq 32$ and current STATE $b_{63}...b_0$ addRoundKey consists of the operation for $0 \leq j \leq 63$

$$b_j \rightarrow b_j \oplus k^k_j$$

2.1.4 sBoxLayer

The S-box used in PRESENT is a 4-bit to 4-bit S-box $S : F^4_2 \rightarrow F^4_2$. The action of this box is hexadecimal notation is given by the following Table 2.2.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2.2: 4-bit S-Box Mapping.

For sBoxLayer the current STATE $b_63...b_0$ is considered as sixteen 4-bit words $w_15...w_0$ where $w_i = b_{4*i+3}||b_{4*i+2}||b_{4*i+1}||b_{4*i}$ for $0 \leq i \leq 15$ and the output nibble $S[w_i]$ provides the updated state values in the obvious way.

2.1.5 pLayer

The bit permutation used in **PRESENT** is given in the Table 2.3.
Bit i of STATE is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Table 2.3: Present Permutate

2.1.6 Key Schedule

Present's key schedule consists of a 61-bit left rotation, an S-Box and an XOR with round counter. Note that PRESENT uses the same S-Box for the datapath and the key schedule, allowing for resource sharing. The user-supplied key is stored in a key register and its 64 most significant bits serve as the round key. The key register is rotated left by 61 bit positions, the leftmost four bits are passed through the PRESENT S-Box, and the round counter value i is exclusive-ored with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of K with the least significant bit of round counter on the right. After extracting the round key K_i, the key register $K = k_{79}k_{78}...k_0$ is updated as follows 2.2.

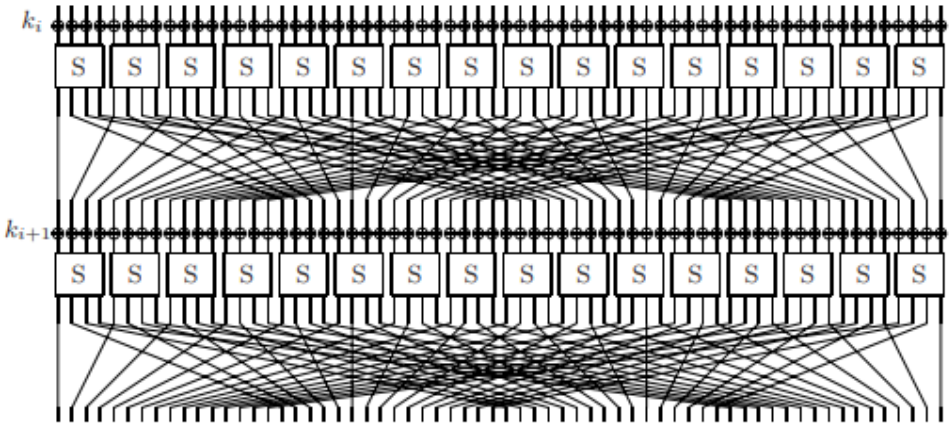


Figure 2.2: The S/P network of present

$$1.[k_{79}k_{78}....k_0] = [k_{18}k_{17}....k_{20}k_{19}]$$

$$2.[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$$

$$3.[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus roundcounter$$

Thus, the key register is rotated by 61 bit positions to the left, the left-most four bits are passed through the present S-box, and the round counter value i is exclusive-ored with bits $k_{19}k_{18}k_{17}k_{16}k_{15}$ of K with the least significant bit of round counter on the right[25].

2.2 Simon algorithm

2.2.1 Introduction of Simon

SIMON is a family of lightweight block ciphers developed by the NSA with the goal of providing a cipher with optimal hardware performance. The Simon block cipher with an n-bit word(and therefore a 2n-bit block) is denoted Simon2n, where n must be 16,24,32,48, or 64.The 32-bit block size versions are intended for the most restricted applications, where a minimal level of security is required. The 48 and 96 bit variants are primarily intended for EPC applications. The 64-bit versions are intended for a variety of lightweight applications, and the 128-bit instantiations are for applications where the highest security is required but where AES is unsuitable due to hardware

or software constraints. Simon2n with an m-word (mn-bit) key will be referred to as Simon2n/mn. The algorithm is engineered to be extremely small in hardware and easy to serialize at various levels. The characteristics of the SIMON block cipher are given at Table 2.4.

block size 2n	key size mn	word size n	key words m	const seq.	rounds T
32 bit	64 bit	16	4	z_0	32
48 bit	72 bit	24	3	z_0	36
48 bit	96 bit	24	4	z_1	36
64 bit	96 bit	32	3	z_2	42
64 bit	128 bit	32	4	z_3	44
96 bit	96 bit	48	2	z_2	52
96 bit	144 bit	48	3	z_3	54
128 bit	128 bit	64	2	z_2	68
128 bit	192 bit	64	3	z_3	69
128 bit	256 bit	64	4	z_4	72

Table 2.4: Simon block cipher operating parameters

The aim of Simon was to fill the need for secure, flexible, and analysable lightweight block ciphers. The algorithm offers excellent performance on hardware and software platforms, is flexible enough to admit a variety of implementations on a given platform, and is amenable to analysis using existing techniques. Perform very well across the full spectrum of lightweight applications . The reason why the algorithm work so well on each platform is because of very simple constructed. So it is very easy to find efficient implementations. For algorithms such as AES it required longer time of research to find near-optimal implementations [27],[28].

2.2.2 Simon Round Function

The Simon2n encryption maps make use of the following operations on n-bit words:

- bitwise XOR, $\oplus$
- bitwise AND, &
- left circular shift. S^j,by j bits

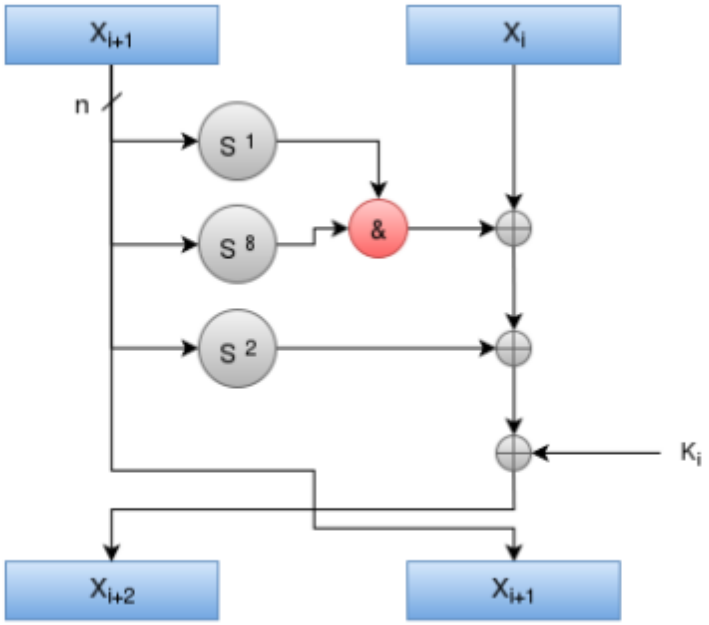


Figure 2.3: The Simon Round Function

Simon is a Feistel network with two branches using the following Fig 2.3 round function:

At the stage of **encryption** we have the data input (plaintext) where splits into two same size words. We can see the encryption progress in the following function which is the Feistel network. We denote as x the MSB digits of the data with y the LSB digits and with k the key of the algorithm. At the end of each round output reversed with input. So in the next round as MSB digits input of the function R will be the output of the previous round and as input of LSB digits will be the input of the previous round [29].

$$R_k(x, y) = (y \oplus f(x) \oplus k, x)$$

where $f(x) = (Sx \& S^8 X) \oplus S^2 x$ and k is the round key.The inverse of the round function, used for decryption, is:

$$R_k^{-1}(x, y) = (y, x \oplus f(y) \oplus k)$$

As for the **decryption** progress the only changes have to do with input data, output data and the keys. More specific at the decryption progress MSB digits change with LSB digits before become input in the first round. At the last round we make the same

change with MSB and LSB digits. One more difference is that keys are reading from the opposite for the entire decryption progress from the last to the first. For that reason need to end first the key production and then to begin the decryption progress.

The round functions for Simon2n take as input an n-bit round key k, together with two n-bit intermediate ciphertext words. The round function is the 2-stage Feistel map. Fig 2.3 shows the effect of the round function Rk_i on the two words of sub-cipher (x_{i+1}, x_i) at the ith step of this process.

2.2.3 Simon Key Schedule

The key schedule of SIMON is described as a function that will operate on two, three or four n-bit word registers, depending on the size of the master key. It performs two rotations to the right by $x \gg 3$ and $x \gg 1$ and XOR the results together with a fixed constant c and five constant. The value c is a constant equal to $(2^n - 4)$. For Simon2n with m key words $k_{m-1}, ..., k_1, k_0$ and constant sequence z_j , round keys are generated by:

$$K_{i+m} = \begin{cases} c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+1}, & if m = 2 \\ c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1})S^{-}3k_{i+2}, & if m = 3 \\ c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+3} \oplus k_{i+1}, & if m = 4 \end{cases}$$

At this Fig 2.4 we see the Simon key schedule for $m\epsilon\{2, 3, 4\}$. The computation of round key k_i depends on k_{i-1} and k_{i-m} and also $ki - m + 1$ in the case of m=4.

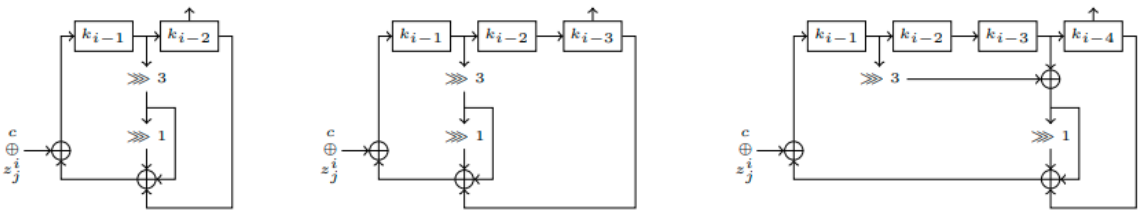


Figure 2.4: For m = 2,3 and 4 key words

Because the key schedule is needed to turn a key into a sequence of round keys the Simon algorithm for the key schedules employ a sequence of 1-bit round constants specifically for the purpose of eliminating slide properties and circular shift symmetries. The designers provide some cryptographic separation between different versions of Simon having the same block size by defining five such sequences: z_0, z_1, z_2, z_3, z_4. At Table 2.5 we can see the values of z_j vectors [30],[31],[32].

j	z_j
0	11111010001001010110000111001101111101000100101011000011100110
1	10001110111110010011000010110101000111011111001001100001011010
2	10101111011100000011010010011000101000010001111110010110110011
3	11011011101011000110010111100000010010001010011100110100001111
4	11010001111001101011011000100000010111000011001010010011101111

Table 2.5: The z_j vectors used in the SIMON key schedule

2.3 Clefia algorithm

2.3.1 Introduction of Clefia

CLEFIA cipher is a 128-bit symmetrical block ciphering algorithm supports 128, 192, and 256-bit keys as we can see at Table 2.6 and provides improved cryptographic security through the use of Diffusion Switch Mechanisms and whitening keys among others, in order to ensure immunity against differential and linear attacks. This algorithm is based on the well known and commonly used Feistel network structure. As in most block ciphers, the input data is processed over several rounds, adding confusion and diffusion with the input key. In this particular algorithm the data and key are processed over 18, 22, or 26 rounds depending on the key sizes. The round computation is exactly the same for each iteration [33].

block size	key size	rounds T
128 bit	128 bit	18
128 bit	192 bit	22
128 bit	256 bit	26

Table 2.6: Clefia block cipher operating parameters

Also clefia has good performance profile both in hardware and software, also can provide a high security level along with good hardware and software implementation capabilities, such as high-speed performance on a wide range of processors. We can see that CLEFIA is an well studied algorithm about the security and implementation is ready to use in practical system, we can use the algorithm for time sensitive applications, smartcards, RFID tags sensor networks, medical devices[33],[34].

2.3.2 Clefia Encryption Block Cipher

The encryption process takes a 128-bit input data block $P = P0|P1|P2|P3$, four 32-bit whitening keys $WK = WK0|WK1|WK2|WK3$, and several 32-bit round keys RKi as data inputs. The resulting outputted ciphertext is a 128-bit cryptogram. At Fig 2.5 we see the data processing part of clefia block cipher.

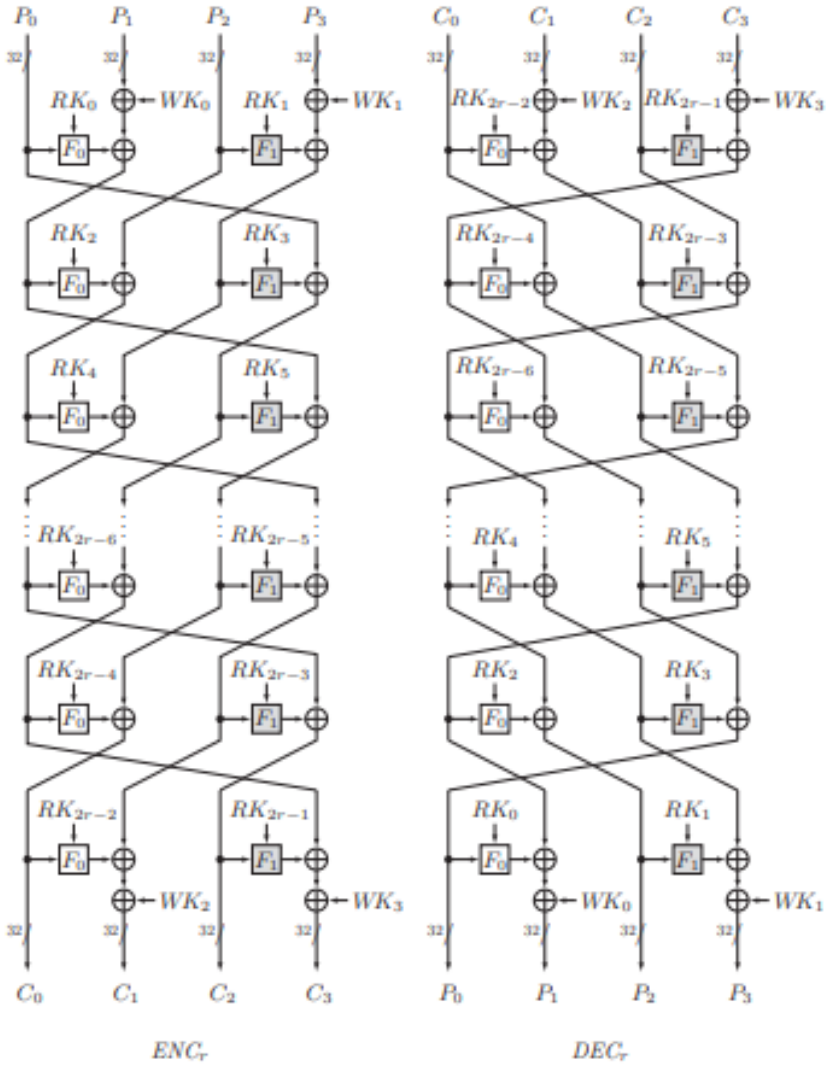


Figure 2.5: Structures of Data Processing Part

The first step of the encryption process is to XOR the second and fourth words of the plaintext (P1 and P3) with the first and second 32-bits of the original key (WK0 and WK1), performing the first key whitening procedure. After this operation the rounds are executed. Each round is computed by a 4-branch Feistel structure, defined by $GFN_{4,n}$, where n is the number of rounds to be computed. The round computation

contains two parallel non-linear F functions per round, where a copy of the first and third words, and two round keys, are their respective inputs[35].

In the final round the second and fourth final words are XORed with the last two whitening keys. Besides the round keys addition, the F0 and F1 functions employ two different types of 8-bit S-Boxes (S0 and S1) and two distinct diffusion matrices (M0 and M1)[36].

For the input 128-bit plaintext ($P = P_0|P_1|P_2|P_3$ the output ciphertext ($C = C_0|C_1|C_2|C_3$ calculated as follows:

$$1. \quad C_0^0 = P_0, C_1^0 = P_1 \oplus WK_0, C_2^0 = P_2, C_3^0 = P_3 \oplus WK_1.$$

$$2. \quad For \quad i = 1 \quad to \quad r-1$$

$$C_0^i = C_1^{i-1} \oplus F_0(C_0^{i-1}, RK_{2i-2}), C_1^i = C_2^{i-1},$$

$$C_2^i = C_3^{i-1} \oplus F_1(C_2^{i-1}, RK_{2i-1}), C_3^i = C_0^{i-1}.$$

$$3. \quad C_0^r = C_0^{r-1}, C_1^r = C_1^{r-1} \oplus F_0(C_0^{r-1}, RK_{2r-2}) \oplus WK_2,$$

$$C_2^r = C_2^{r-1}, C_3^r = C_3^{r-1} \oplus F_1(C_2^{r-1}, RK_{2r-1}) \oplus WK_3.$$

The structure of F_0, F_1 functions are shown in Figs **??**, **??**.

2.3.3 Key Scheduling

Since each round uses two 32-bit round keys a total of 36, 44, or 52 round keys (depending on the number of rounds) are needed, plus 4 additional whitening keys. These round keys are obtained using the specified key schedule algorithm.The whitening key (WK) generation is accomplished according to the key size [37]. For a 128-bit input key, the four 32-bit whitening keys are obtained directly from the input key, by:

$$WK_0|WK_1|WK_2|WK_3 \leftarrow K.$$

For the 192 or 256-bit input keys, the value is divided into two 128-bit blocks, KL and KR, as shown by:

$$KL||KR \leftarrow K_0|K_1|K_2|K_3 \;||\; K_4|K_5|\overline{K_0}|\overline{K_1} : K^{192}$$

$$KL||KR \leftarrow K_0|K_1|K_2|K_3 \;||\; K_4|K_5|K_6|K_7 : K^{256}$$

The corresponding whitening key is then computed by:

$$WK = KL \oplus KR$$

The key expansion of a 128-bit key uses the same 4- branched GFN network used for the CLEFIA main encryption process. The differences in the 128-bit key expansion is that the input data of the GFN structure is now the input key itself, and the round keys are replaced with predefined constants.When considering the key schedule for the 192 and 256- bit keys, the GFN network becomes an 8-branch structure ($GFN_{8,n}$). In this case, the input value is a combination of $K = KL||KR$. The 8-branch Feistel structure uses the same two non-linear F functions, twice per round and processes eight input words on each round.Instead of a ciphered text, the output of the GFN structure, in the key expansion process, is either a 128- bit block (L), for 128-bit input keys, or two 128-bit blocks (LL and LR) for the remaining key sizes. After the GFN computation is completed, the result (L or LL and LR) is expanded in an iterative way using a double swap (Σ) function, as:

$$L = \Sigma(L)$$

$$LL = \Sigma(LL)$$

$$LR = \Sigma(LR)$$

The Σ function swaps several bits of its 128-bit input and returns another equally sized output, specified by:

$$\Sigma(X) = X[7-63]|X[121-127]|X[0-6]|X[64-120]$$

With this, the 32-bit round keys are obtained by adding alternately the L, K, and $\Sigma(X)$ values with another predefined set of constants.

2.3.4 Functions F_0, F_1

The functions uses two different S-box with 8-bit input and output and two different tables M_0, M_1 Figures 2.6, 2.7.

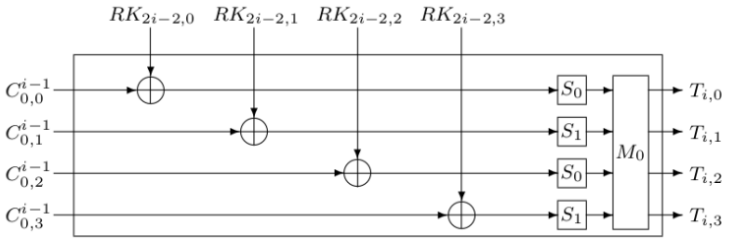


Figure 2.6: F_0 Function

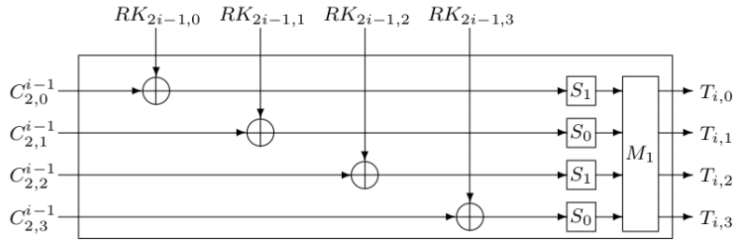


Figure 2.7: F_1 Function

The 32-bit output $(T_i, T_i \exists \{0,1\}^{32})$ calculated as follows:

$$1. \quad T_i = C_0^{i-1} \oplus RK_{2i-2}$$

$$2. \quad Let \quad T_i = T_{i,0}|T_{i,1}|T_{i,2}|T_{i,3},\ T_{i,j} \exists \{0,1\}^8 \quad (j = 0,1,2,3),$$

$$T_{i,0} = S_0(T_{i,0}), T_{i,1} = S_1(T_{i,1}), T_{i,2} = S_0(T_{i,2}), T_{i,3} = S_1(T_{i,3}).$$

$$(T_{i,0}, T_{i,1}, T_{i,2}, T_{i,3})^T = M0(T_{i,0}, T_{i,1}, T_{i,2}, T_{i,3})^T$$

2.3.5 Tables M_0, M_1

The multiplications of a matrix and a vector are performed in $GF(2^8)$ defined by the lexicographically first primitive polynomial $z^8+z^4+z^3+z^2+z^1+1$. Tables M_0, M_1 are defined at Figure 2.8.

$$M_0 = \begin{pmatrix} \texttt{0x01} & \texttt{0x02} & \texttt{0x04} & \texttt{0x06} \\ \texttt{0x02} & \texttt{0x01} & \texttt{0x06} & \texttt{0x04} \\ \texttt{0x04} & \texttt{0x06} & \texttt{0x01} & \texttt{0x02} \\ \texttt{0x06} & \texttt{0x04} & \texttt{0x02} & \texttt{0x01} \end{pmatrix}, \qquad M_1 = \begin{pmatrix} \texttt{0x01} & \texttt{0x08} & \texttt{0x02} & \texttt{0x0a} \\ \texttt{0x08} & \texttt{0x01} & \texttt{0x0a} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x0a} & \texttt{0x01} & \texttt{0x08} \\ \texttt{0x0a} & \texttt{0x02} & \texttt{0x08} & \texttt{0x01} \end{pmatrix}$$

Figure 2.8: Tables M_0, M_1

2.3.6 Clefia Sboxes

Tables S_0, S_1 are defined at Fig 2.9

	S_0																S_1															
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	57	49	d1	c6	2f	33	74	fb	95	6d	82	ea	0e	b0	a8	1c	6c	da	c3	e9	4e	9d	0a	3d	b8	36	b4	38	13	34	0c	d9
1.	28	d0	4b	92	5c	ee	85	b1	c4	0a	76	3d	63	f9	17	af	bf	74	94	8f	b7	9c	e5	dc	9e	07	49	4f	98	2c	b0	93
2.	bf	a1	19	65	f7	7a	32	20	06	ce	e4	83	9d	5b	4c	d8	12	eb	cd	b3	92	e7	41	60	e3	21	27	3b	e6	19	d2	0e
3.	42	5d	2e	e8	d4	9b	0f	13	3c	89	67	c0	71	aa	b6	f5	91	11	c7	3f	2a	8e	a1	bc	2b	c8	c5	0f	5b	f3	87	8b
4.	a4	be	fd	8c	12	00	97	da	78	e1	cf	6b	39	43	55	26	fb	f5	de	20	c6	a7	84	ce	d8	65	51	c9	a4	ef	43	53
5.	30	98	cc	dd	eb	54	b3	8f	4e	16	fa	22	a5	77	09	61	25	5d	9b	31	e8	3e	0d	d7	80	ff	69	8a	ba	0b	73	5c
6.	d6	2a	53	37	45	c1	6c	ae	ef	70	08	99	8b	1d	f2	b4	6e	54	15	62	f6	35	30	52	a3	16	d3	28	32	fa	aa	5e
7.	e9	c7	9f	4a	31	25	fe	7c	d3	a2	bd	56	14	88	60	0b	cf	ea	ed	78	33	58	09	7b	63	c0	c1	46	1e	df	a9	99
8.	cd	e2	34	50	9e	dc	11	05	2b	b7	a9	48	ff	66	8a	73	55	04	c4	86	39	77	82	ec	40	18	90	97	59	dd	83	1f
9.	03	75	86	f1	6a	a7	40	c2	b9	2c	db	1f	58	94	3e	ed	9a	37	06	24	64	7c	a5	56	48	08	85	d0	61	26	ca	6f
a.	fc	1b	a0	04	b8	8d	e6	59	62	93	35	7e	ca	21	df	47	7e	6a	b6	71	a0	70	05	d1	45	8c	23	1c	f0	ee	89	ad
b.	15	f3	ba	7f	a6	69	c8	4d	87	3b	9c	01	e0	de	24	52	7a	4b	c2	2f	db	5a	4d	76	67	17	2d	f4	cb	b1	4a	a8
c.	7b	0c	68	1e	80	b2	5a	e7	ad	d5	23	f4	46	3f	91	c9	b5	22	47	3a	d5	10	4c	72	cc	00	f9	e0	fd	e2	fe	ae
d.	6e	84	72	bb	0d	18	d9	96	f0	5f	41	ac	27	c5	e3	3a	f8	5f	ab	f1	1b	42	81	d6	be	44	29	a6	57	b9	af	f2
e.	81	6f	07	a3	79	f6	2d	38	1a	44	5e	b5	d2	ec	cb	90	d4	75	66	bb	68	9f	50	02	01	3c	7f	8d	1a	88	bd	ac
f.	9a	36	e5	29	c3	4f	ab	64	51	f8	10	d7	bc	02	7d	8e	f7	e4	79	96	a2	fc	6d	b2	6b	03	e1	2e	7d	14	95	1d

Figure 2.9: Tables S_0, S_1

2.3.7 DoubleSwap Function

The DoubleSwap function $\Sigma : \{0,1\}^{128} \rightarrow \{0,1\}^{128}$ is defined as follows:

$$X_{128} \rightarrow Y_{128}$$

$$Y = X[7-63]|X[121-127]|X[0-6]|X[64-120]$$

Where $X[a-b]$ denotes a bit string cut from the a-th bit to the b-th bit of X.

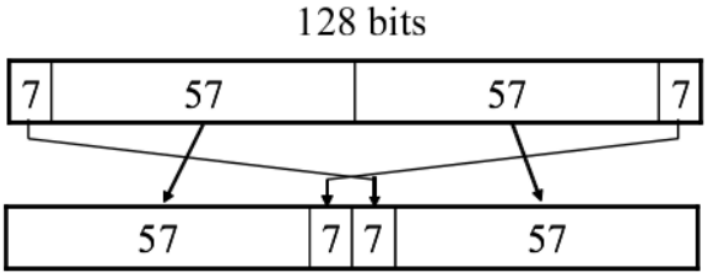


Figure 2.10: DoubleSwap Function

Chapter 3

Platforms

3.1 The structure of FPGAs

FPGA (Field programmable Gate Array) provide programmable logic resources that enable users to implement custom functions at the hardware level. These logic resources can be programmed at the "field" several times (they can be "reconfigured") to realise different functions over time as opposed to fixed hardware. They are built around an array of programmable logic blocks embedded in a sea of programmable interconnects. This array is often referred to as the programmable logic fabric or just the fabric . At the edges are programmable I/O blocks designed to interface the fabric signals to the external world. It was this set of innovations that sparked the FPGA industry. Fig 3.1 shows a basic architecture of an FPGA [38].

Interestingly, nearly all the other special FPGA features such as carry chains, block RAM, or DSP blocks can also be implemented in programmable logic. This is in fact the approach the initial FPGAs took and users did implement these functions in LUTs. However, as the FPGA markets developed, it became clear that these special functions would be more cost effective as dedicated functions built from hard gates and later FPGA families such as the Xilinx 4 K series and Virtex began to harden these special functions. This hardening improved not only cost but also improved frequency substantially [38].

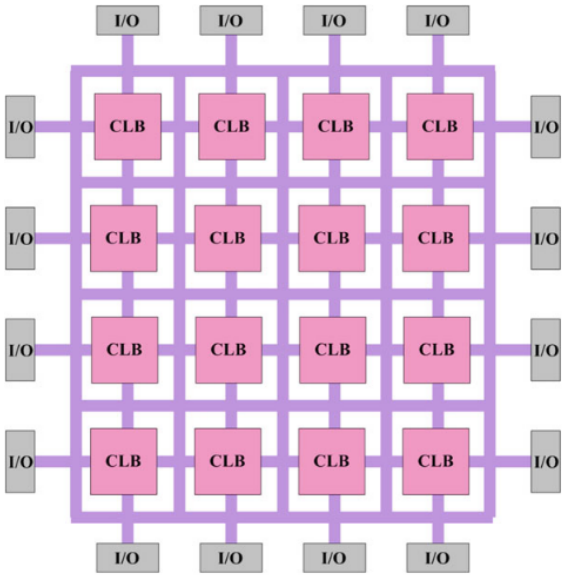


Figure 3.1: Overview of FPGA architecture [5].

Within any FPGA family, all devices will share a common fabric architecture, but each device will contain a different amount of programmable logic. This enables the user to match their logic requirements to the right-sized FPGA device. FPGAs are also available in two or more package sizes which allow the user to match the application I/O requirements to the device package. FPGA devices are also available in multiple speed grades and multiple temperature grades as well as multiple voltage levels. The highest speed devices are typically 25 the lower speed devices. By designing to the lowest speed devices, users can save on cost, but the higher performance of the faster devices may minimize system level cost [38].

Modern FPGAs commonly operate at 100–500 MHz. In general, most logic designs which are not targeted at FPGA architectures will run at the lower frequency range, and designs targeted at FPGAs will run in the mid-frequency range. The highest frequency designs are typically DSP designs constructed specifically to take advantage of FPGA DSP and BRAM blocks [38].

Also for varying requirements, a portion of an FPGA can be partially reconfigured while the rest of an FPGA is still running. Any future updates in the final product can be easily upgraded by simply downloading a new application bit-stream. However, the main advantage of FPGAs i.e. flexibility is also the major cause of its drawback. Flexible nature of FPGAs makes them significantly larger, slower, and more power consuming than their ASIC counterparts. These disadvantages arise largely because of

the programmable routing interconnect of FPGAs which comprises of almost 90% of total area of FPGAs. But despite these disadvantages, FPGAs present a compelling alternative for digital system implementation due to their less time to market and low volume cost [38].

3.2 Platforms

During this thesis experiments on software and on hardware were made. The equipment used for the software tests was a generic Laptop CPU(Intel Core I5-4200U) a server(Intel Xeon E5-2630 v4) and also a CPU from Zedboard(ARM Cortex-A9) and a CPU from Kria Kv260(ARM Cortex-A53). For the hardware implementation, tests were made on Kria Kv260 FPGA.

3.2.1 Kria KV260 Vision AI Starter Kit

The Xilinx Kria KV260 Vision AI Starter Kit is comprised of a non-production version of the K26 system-on- module (SOM), carrier card, and thermal solution. The SOM is very compact and only includes key components such as a Zynq UltraScale+ MPSoC based silicon device, memory, boot, and security module. The carrier card allows various interfacing options and includes a power solution and network connectors for camera, display, and microSD card. The thermal solution has a heat sink, heat sink cover, and fan. The Kria KV260 Vision AI Starter Kit is designed to provide customers a platform to evaluate their target applications and ultimately design their own carrier card with Xilinx K26 SOMs. While the SOM itself has broad AI/ML applicability across markets and applications, target applications for the Kria KV260 Vision AI Starter Kit include smart city and machine vision, security cameras, retail analytic, and other industrial applications.

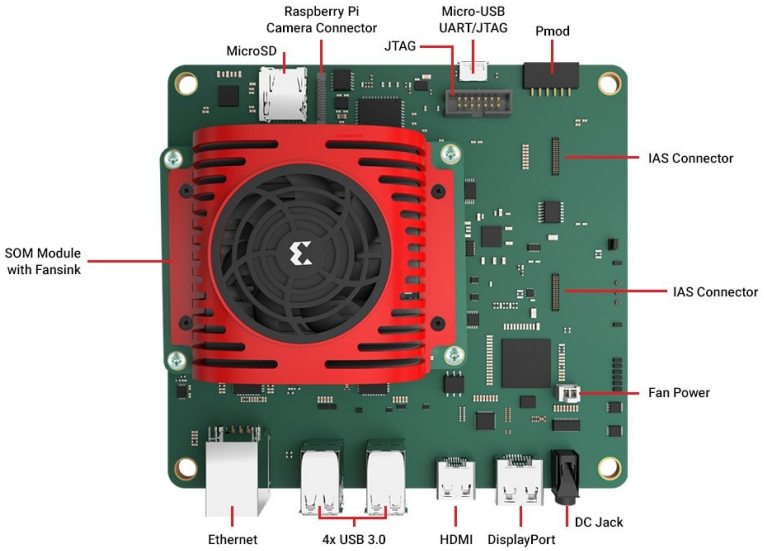


Figure 3.2: Kria Platform

The Kria KV260 Vision AI Starter Kit is an evaluation platform for the K26 SOM focused on machine learning acceleration in vision applications. The kit brings together a Zynq UltraScale+ MPSoC based SOM with user selectable, vision focused peripherals and a set of pre-built accelerated applications. It offers flexibility in both hardware and software applications. The following table shows the most important PL features.

Parameter	**KV260**
System logic cells	256,000
Block Ram blocks	144
DSP slices	1,200
Ethernet interface	One 10/100/1000 Mb/s
DDR memory	4GB(4*512Mb*16 bit)
UltraRAM blocks	64

Table 3.1: KRIA KV260 Specifications

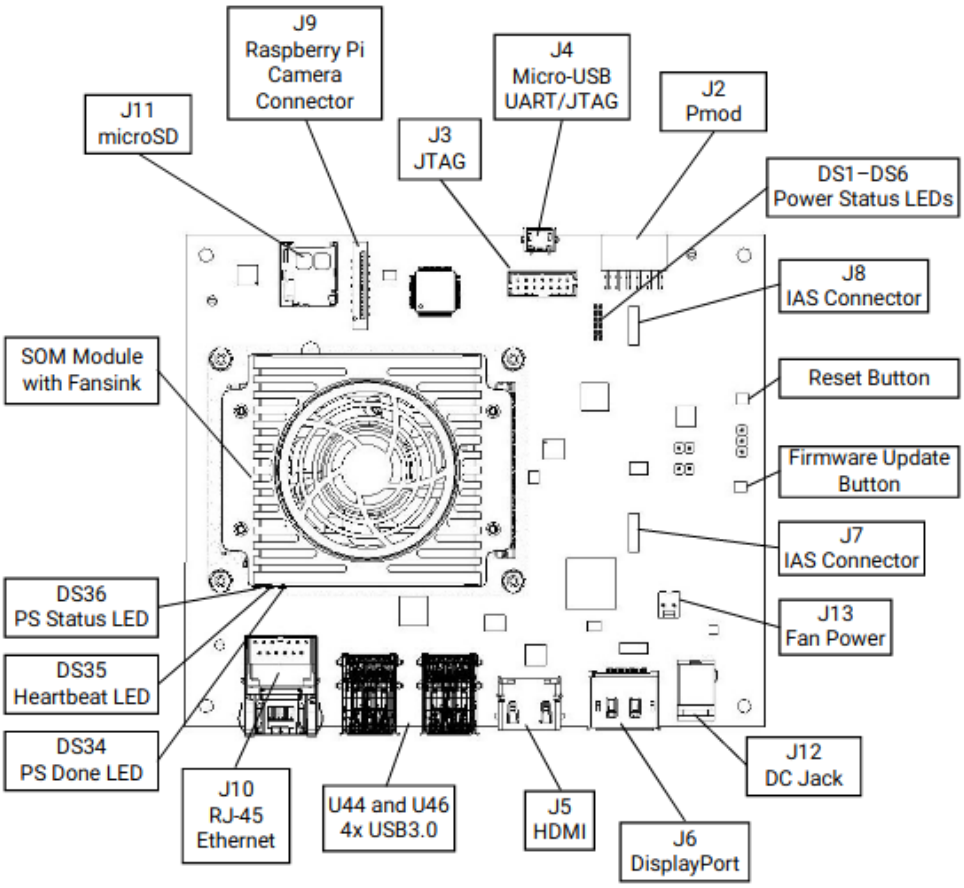


Figure 3.3: Interfaces and Connectors

Chapter 4

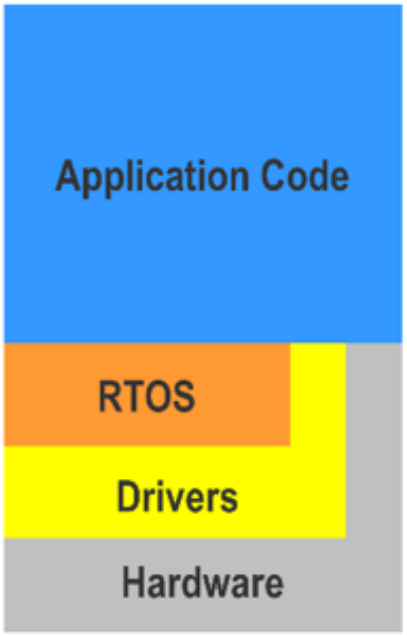
FPGA Implementation

4.1 Architecture Analysis

This section explains the actual hardware accelerator developed in this work. As described previously, the design is made on a Xilinx Kria kv260 FPGA. Next subsection first shows the general architecture of the accelerator. Each algorithm is divided into several logic parts and that will be explained at the next sections in more details.

4.1.1 System Architecture

The entire hardware accelerator is divided into multiple separate blocks, each of which has its own control and can do the tasks for which it was created. It is simpler to build a large system and ensure that every block completes its own mission without errors when a system is divided into smaller units. Figure 4.1 the main building blocks and connections of the acceleration system. The Zynq Ultrascale+ MPSoC core is the first block in the left. This core serves as the Zynq Processing system CPU, making it the block in charge of managing the entire system. Direct signals or data can be sent to the FPGA hardware accelerator by a C program running on an ARM core of the Zynq Programmable System with bare metal method. With bare metal programming we meant the programming without various abstraction layers, or as some experts put it without an operating system backing it. When interacting with a system, bare-metal programming takes into account the hardware's unique construction.



In many cases, bare-metal programming concentrates on the operation of the CPU and other system components, interacting with the BIOS and boot sequence, and writing straightforward code modules to produce particular outputs based on the hardware configuration. Instead than relying on tools like complicated compilers, with that method we can work directly with the hardware by using languages like C and C++. The Vitis Bare-Metal app runs on the board's processing system. The purpose is to enable and control UART communication between the user and the design. The serial connection allows the user to send instructions to the PS and receive output. Users can perform a encryption with one of the three lightweight algorithms(Simon, Present, Clefia) by entering the data they want to encrypt.

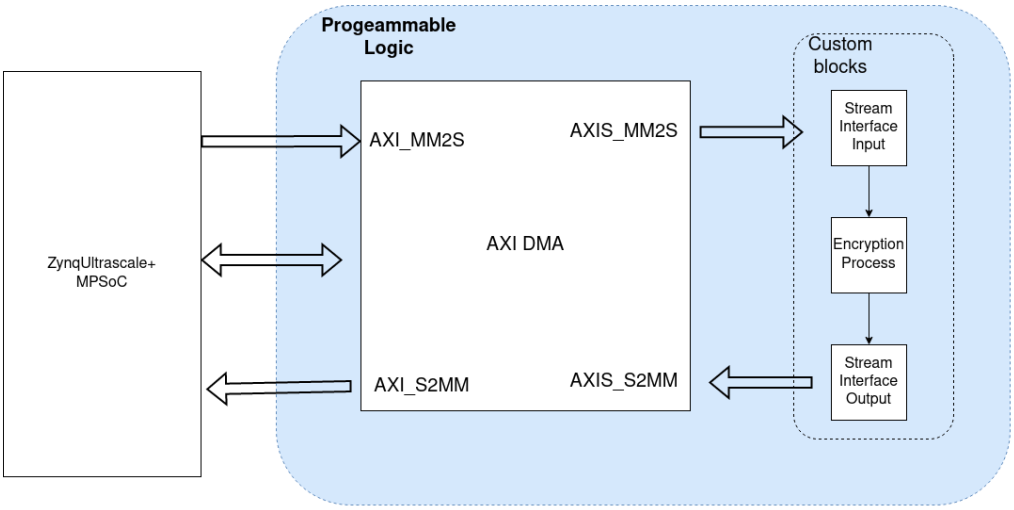


Figure 4.1: A condensed representation of the hardware accelerator system, including all significant components.

The custom blocks that implement each of the three algorithms receive the input data for the encryption process from the Zynq Processing Core via a Direct Memory Access (DMA) block. Without using the CPU, a DMA block moves data from one area to another(Processing System to Programmable Logic and vice versa). The DMA only needs to be configured once, at the beginning of the transfer, after which the CPU can carry on with other tasks while the DMA sends the data. Data stored at a specified position in the memory is to be sent directly to the custom blocks, as instructed by the Zynq core to the DMA block. The DMA block then transfers the data after waiting for the other blocks to finish processing it. The data transfer between Processing System and Programmable Logic is done with the AXI4 Stream protocol, the details about this protocol are provided in subsection 4.5.2. The first block on custom blocks (Stream interface input) is the initial block that receives the data from the DMA block. This block is in charge of obtaining the streaming data from DMA block, rearranging it, and then transmitting it over a 64-bit connection to the encryption process block. After the encryption process is done the encrypted data goes back to the DMA block as a stream data.

4.1.2 Data transfer with DMA and configuration

An AXI DMA core (provided by Xilinx) is utilized to transmit data from the Zynq Processing Core to the lightweight algorithm block and back to the Processing Core. This section will describe the connections between the DMA block and the other blocks as well as the controls and configuration of the DMA will be covered in further detail.

The Advanced eXtensible Interface (AXI) is used by the DMA to connect with other blocks. This protocol is a component of the advanced microcontroller bus architecture (AMBA) standard. AMBA bus architecture is very useful for DMA operations also for connecting blocks in System on Chip (SoC). As we can see in figure 4.1 the DMA has five channels for transferring data. The AXI4 Read Master to AXI4 memory-mapped to stream (MM2S) Master and AXI stream to memory-mapped (S2MM) Slave to AXI4 Write Master are the primary high-speed DMA data transfer components between system memory and the stream target.

The MM2S channel and S2MM channel operate independently. The AXI DMA offers automatic burst mapping, 4 KB address boundary protection (when configured in non-Micro DMA), the ability to queue multiple transfer requests, and practically the whole bandwidth of the AXI4-Stream buses. To communicate user application data to the target IP, the MM2S channel provides an AXI Control stream. An AXI Status stream is made available for the S2MM channel to receive user application data from the target IP.

The Vivado Design Suite settings can be used to adapt the DMA block for a particular application. First the required connections are established and both MM2S and S2MM channels are enabled. Another setting is width of buffer length register of 20 bits and the address width is set to 32 bits. Also for the channels where DMA reads and writes from and to the stream interface (S2MM and MM2S) is set to 64 bits because the AXI Stream interface designed to be 64-bit. After finish the implementation and the synthesis in the Vivado we export the hardware for having access on Vitis where a program will run for controlling the PS side. After exporting the hardware Xilinx provide some standard functions and libraries that we need to configure and initialize($XAxiDma\_LookupConfigBaseAddr, AxiDma\_CfgInitialize$) the DMA for the data transfer. As previously mentioned, the Zynq Processing System manages the DMA using a memory-mapped AXI Lite interface. This indicates that a fixed address

in the system's memory is directly mapped to the status and control registers. The mapping for the various DMA status and control register is contained in a single, continuous block of memory, and it can be accessed using a fixed base address as offset. The DMA's data-sheet [39] contains the offset value of each register. Every channel uses five 32-bit registers so they can run independently of one other. At the program that we run on PS the data that is to be processed first is written to a specific position in the PS memory by the software. Secondly, a signal that resets the DMA control registers is transmitted. The address is then written to the lower address register as the source for the MM2S or the destination for S2MM channels. After we decide the length of the transfer packet that we want to sent, bytes is written to the appropriate register. Data is then sent through the AXI Stream interface by the DMA which has now begun reading data from the memory. In the opposite direction in order to write data back to memory, DMA waiting for the respective lightweight algorithm to finish the encryption process to load the results into the memory and display them back to Processing System.

4.1.3 Custom Blocks

Data from the DMA is sent to the lightweight cryptography algorithm block through an AXI Stream connection. In this connection the DMA's port is the master and the algorithm' s block port is the slave because data is flowing from the DMA to the algorithm block. The DMA master informs the slave that is prepared to begin data transfer after receiving the data to be processed from the PS. The slave responds by letting the master know that is prepared to begin receiving the data. Every cycle a new 64-bit word is sent from the master to slave as long as both sides are available. The slave only needs to deassert the ready signal if it is unable to follow the master for some reason. When the master recognizes this, it will stop transmitting new data until the slave is once more prepared. For controlling the signals when the DMA is ready or when the Stream interface is ready to take or sent some data a State Machine is made. From the opposite side when the Stream interface sent data to DMA, the AXI Stream port is the master and the Dma port is the slave. The 64-bit word where is the result of the processing process is placed on the Dma after this packet has been read and the ready signal is asserted. Every cycle a new word is transferred once the slave is prepared as well. The lightweight algorithm is looking for new data after the the last

word of the packet is sending.

4.2 Architecture of Present Encryption

In this work, the cryptographic hardware of the present algorithm was based on the model from its developers, Bogdanov rt al. (2007) for the version with block size of 64 bit and 80 bit key. Present's architecture is modular with each specific operation performed by a module or component managed by the control module, which is properly connected to the Present top encryption module. The functions of each module that make up the architecture are described below.

4.2.1 Top Module

Figure 4.2 presents the main diagram of the algorithm. All the components are described below.

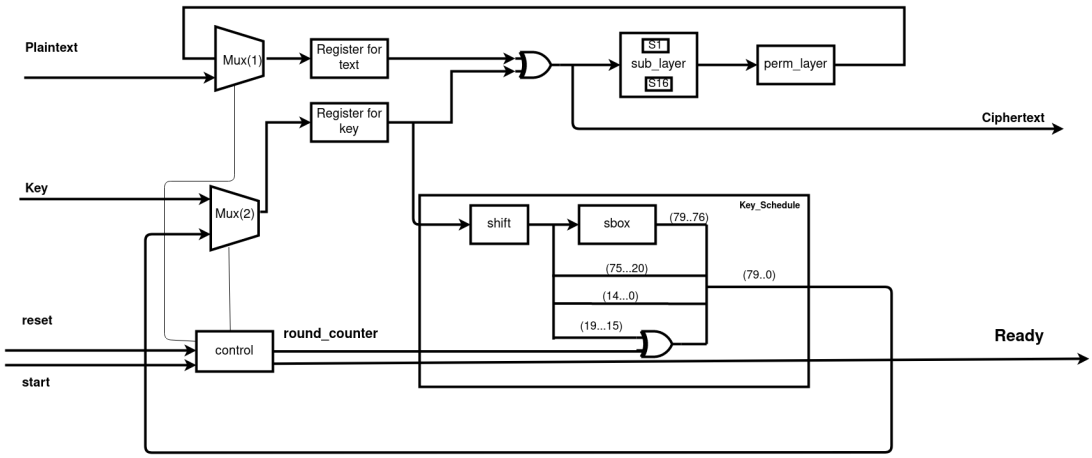


Figure 4.2: Present Top Module.

The add on modules of top level described below:

- **Register for text**:Register for the Plaintext and the encrypted at each round of the algorithm text(64 bits).
- **Register for key**:Register for the original key and the generated keys at each round of the algorithm with size(80 bits).
- **MUX(1)**:This mux is responsible for the selection of the Plaintext or the Ciphertext at each round of encryption process.

- **MUX(2)**:This mux is responsible for the selection of the Original Key or the generated round Keys at every round of encryption process.

4.2.2 Module sBoxLayer

This module aims to perform substitution operations based on the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 4.1: 4-bit S-Box Mapping.

For sBoxLayer the current STATE $b_{63}....b0$ is consider as sixteen 4-bit words $w_{15}...w0$ where $w_i = b_{4*i+3}||b_{4*i+2}||b_{4*i+1}||b_{4*i}$ for $0 \leq i \leq 15$ and the output nibble $S[w_i]$ provides the updated state values in the obvious way.
The VHDL algorithm for the substitution uses CASE and WHEN operations. Part of the sBoxLayer operation algorithm is displayed in the table 4.1, where in order to meet the need to display the 64-bit vector of text for encryption round, 16 modules where instantiated to perform the substitution operation.

4.2.3 Module pLayer

This module performs simple (bitwise) permutation operations of an array of 64-bits with the data coming from sBoxLayer.
The bit permutation used in PRESENT is given by the following table.
Bit i of STATE is moved to bit position $P(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Table 4.2: Present Permutate

It is worth mentioning that for the permutate layer operation, the most significant bit ordering (MSB) to the least significant bit (LSB) of the 64-bit vector, occurs from

left to right. on the right, as shown in table 4.2.

4.2.4 Module Key Schedule

PRESENT can take keys of either 80 or 128 bits. However we focus on the version with 80-bit keys. The user-supplied key is stored in a key register K and represented as $k_{79}k78...k0$. At round i the 64-bit round key $K_i = k_{63}k_{62}...k_0$ consists of the 64 leftmost bits of the current contents of register K.This module performs original key update operations, aiming to attend the operation of adding the round key, which is performed along with the text at each round of encryption. The key schedule updated every round according to the following steps:

$$1.[k_{79}k_{78}....k_0] = [k_{18}k_{17}....k_{20}k_{19}]$$

$$2.[k_{79}k_{78}k_{77}k_{76}] = S[k_{79}k_{78}k_{77}k_{76}]$$

$$3.[k_{19}k_{18}k_{17}k_{16}k_{15}] = [k_{19}k_{18}k_{17}k_{16}k_{15}] \oplus roundcounter$$

In vhdl code, first the 19 LSB of the key are permuted and become the MSB of the key. Then the leftmost 4 bits goes into sBoxLayer operation and finally the bits from 19 to 15 of the key that is being updated are xored with the bits of the round counter variable that represent in which round the algorithm is.Need to say also that for the operation of the key schedule the round key is performed through an xor operation of the bits of the text block (64 bits), with the 64 bits of the key from key schedule that updated each round of encryption.

4.2.5 Module State Machine

This module manages modules and circuits aiming at enabling and correct selection of data at each round of encryption, as well as proper signaling at the end of the plaintext encryption process(64 bits).This control module has 4 stages:

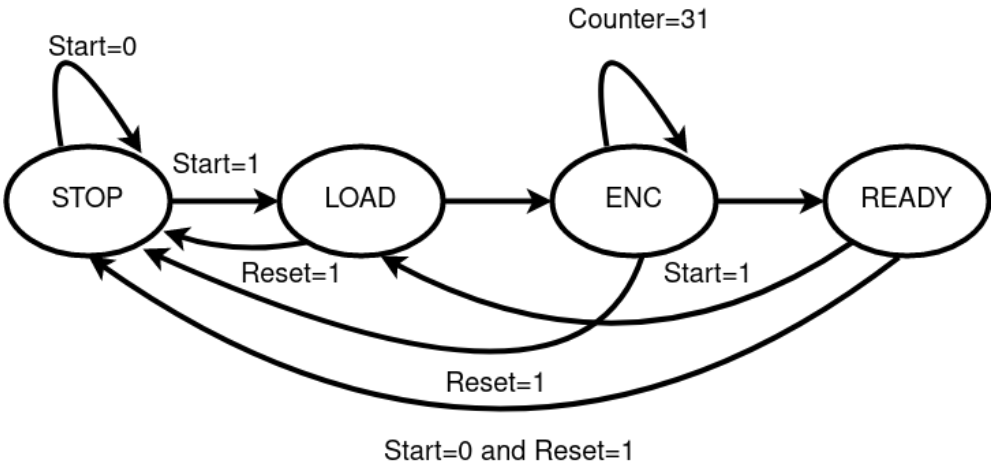


Figure 4.3: State Machine Diagram.

- **STOP**:Stop the process or initial the parameters.
- **LOAD**:Load the register with Plaintext(64 bits) and Key register(80 bits) and perform the round key addition operation.
- **ENC**:Run the necessary rounds of encryption. In this implementation are performed 31 rounds of encryption.
- **READY**:Signals for reading the ciphertext after the encryption process.

4.3 Architecture of Simon Encryption

The cryptographic hardware of the simon algorithm was based on the model from the paper [28] for the version with 64 bit block size and 128 bit key. The simon block cipher implemented in VHDL language for the FPGA usage.

4.3.1 Top Level Module

Simon block cipher implemented as a full-loop unrolling architecture. The number of clock cycles required to encrypt a block of data or decrypt a block of data is reduced by a factor of K (K - rounds), and the minimum clock period is reduced by a factor slightly smaller than K giving an overall increase in throughput and decrease in latency while simultaneously resulting in an increase the area more or less proportional to K due to unrolling of combinational logic of round and key expansion functionality as well

as the number of simultaneously stored round keys. A simple diagram with inputs and outputs from simon block cipher are shown in figure 4.4.

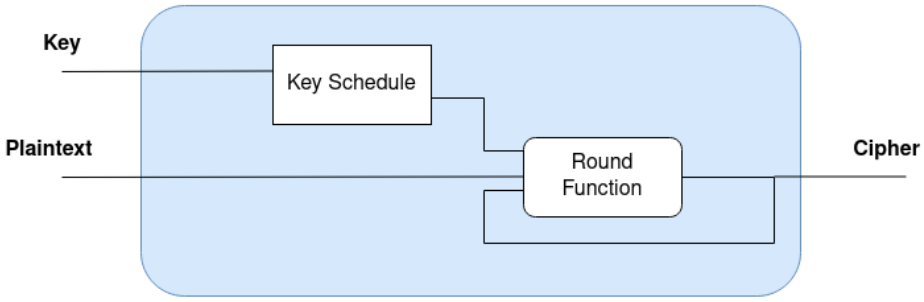


Figure 4.4: Top Level Diagram.

In that top level module an FSM machine have been created at VHDL language. The machine is responsible for taking the data from outside and send the data out when it's ready. The FSM include 4 states:

- **IDLE**
- **STABILIZE**
- **WRITE CIPHERTEXT**
- **LAST STATE**

Firstly, the algorithm can accept data from the DMA controller when the state is idle. In that state 64 bit of plaintext data are written. After that fsm wait for the algorithm for the encryption process of the data. At write ciphertext state the simon algorithm has finished with the data encryption and is ready to send the data back to dma. Figure 4.5 shows the diagram of the fsm.

4.3.2 Module Key Schedule

Key schedule for SIMON block cipher provides key expansion capabilities by subsequently generating all round keys from the master. For the implementation of SIMON with 64 bit block and 128 bit key size the key schedule generates 44 round keys with 32 bit size where they provided from master key with size 128 bit. For the key schedule round i and constant c is needed to generate the round keys. The key expansion function utilizes the following operations:

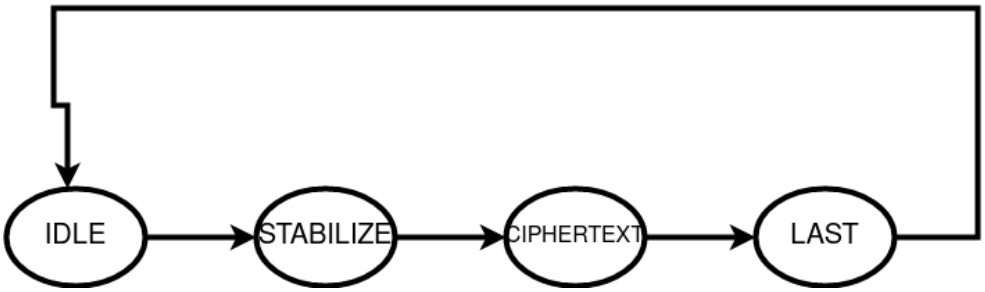


Figure 4.5: Simon FSM.

- Bitwise XOR, denoted as $x \oplus y$.
- Right bitwise rotation ROR, denoted as $S^y(x)$ where y is the rotation count.

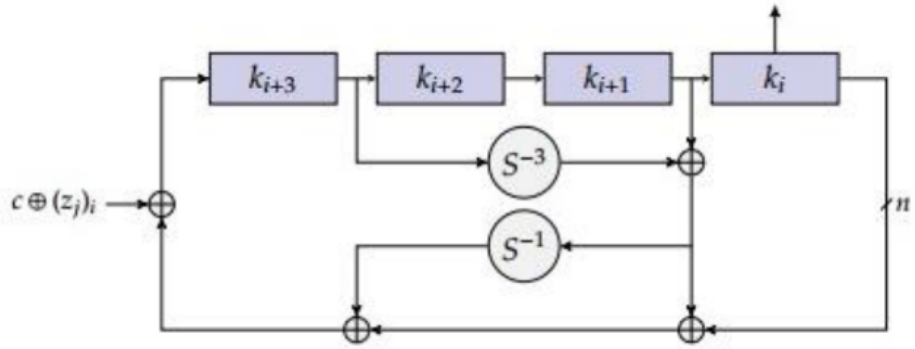


Figure 4.6: Simon 4-word key expansion.

The key expansion is expressed as:

$$K_i(k, c, z_i) = F(k_{i+3}, k_{i+1}) \oplus S^{-1}(F(k_{i+3}, k_{i+1})) \oplus k_i \oplus c \oplus (z_j)_i$$

Where:

$$F(x, y) = S^{-3}(x) \oplus y.$$

After the expansion operation, the cached round keys are rotated to the right, discarding the first and replacing the last with the newly generated round key. $k_i = k_{i+j}$ for $j = 0, 1, 2$ and $k_{i+3} = K_i(k, c, z_j)$

The Key Schedule is implemented as a combinational circuit were the round keys are generated from master key, also two constants c and z_3 are combined into a single constant $C = c \oplus z_3$ for efficiency purposes.

4.3.3 Round Function

The round function module for simon block cipher is implemented in vhdl language as combinational circuit as expressed at chapter 2.2.2. For that purpose shift right, left operations and logic gates xor, and are used to help the implementation of that module.

To implement full unrolling we repeated the round K = 44 times, interconnecting each round with the following one and connecting each round to the appropriate signal supplying the round key from the unrolled key scheduling circuit. In this way, 44 round function operates are performed inside a single clock cycle to encrypt the plaintext, converting it into ciphertext. The architecture of round function module is shown in figure 4.7.

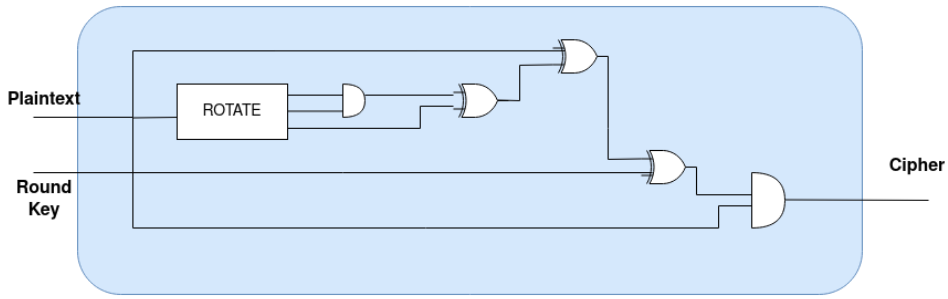


Figure 4.7: Simon Round Function.

For the Encryption process of simon block cipher no RAM is required.

4.4 Architecture of Clefia Encryption

For use with FPGAs, the Clefia block cipher has also been implemented with VHDL language. The algorithm used in this work was created using the original specs and a verilog version made available by Sony Corporation (2010). The version of this method that was put into into practice has a block size of 128 bits and a key size of 128 bits. As already mentioned in the section 2.3.1, clefia employs a four-branch Generalized Feistel structure (GF4N) with two functions F(f_0 and f_1). Below is a description of each module that makes up the architecture.

4.4.1 Top Module

At previous sections described the modules for key scheduling and encryption process of the algorithm, here the top level block diagram of the algorithm is shown at Fig 4.8.

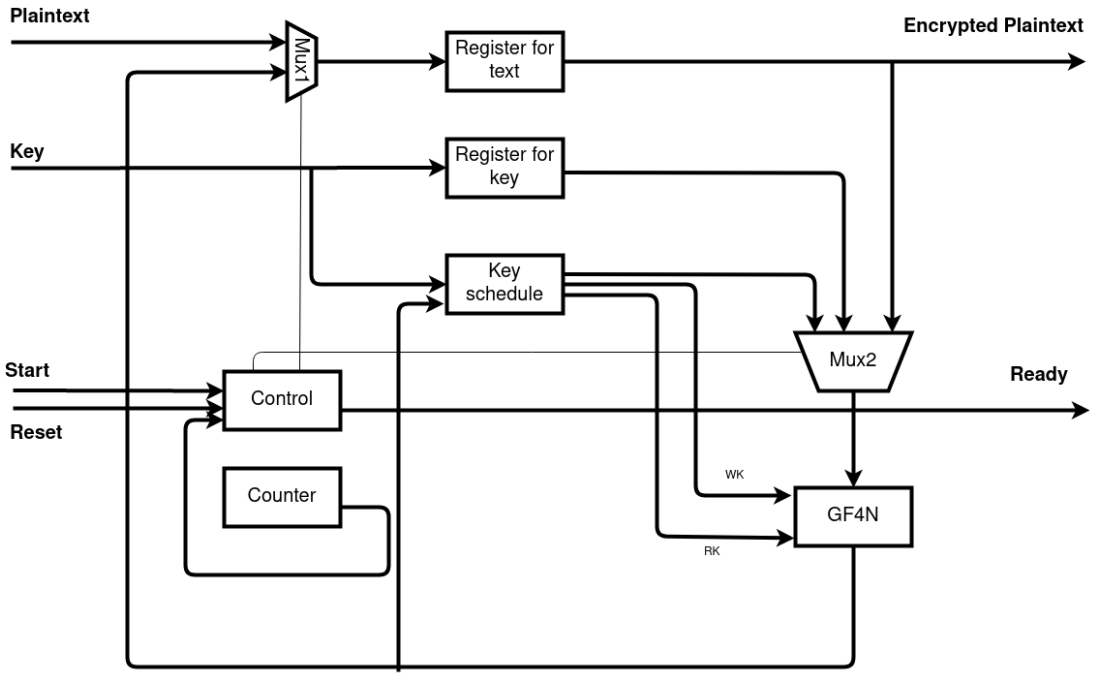


Figure 4.8: Clefia Top Level

The add on modules of top level described below:

- **MUX1**:This mux has 2 inputs, the plaintext and the encrypted text of each round of encryption process, so the mux is responsible to choose the plaintext or the ciphertext at each round.
- **Register for text**: Register for Plaintext and encrypted text for every rounds of encryption process.
- **Register for key**:Register for the original key and the round keys that generated at each round of the algorithm.
- **Counter**: A 5-bit counter that counts the rounds for key generation and encryption process of the algorithm.
- **MUX2**:This mux has 3 inputs, the original key, the data derived from key sched-

ule module for generating the intermediate key L and the text derived from register text for the encryption process of the algorithm.

4.4.2 Modules S0,S1

S0 and S1 are nonlinear 8-bit S-boxes. In these tables all values are expressed in hexadecimal form, suffixes '0x' are omitted. For an 8-bit input of an S-box the upper 4-bit indicates a row and the lower 4-bit indicates a column. The S0 and S1 modules, described in VHDL and use operations like with, select, when, for its construction.

	S_0																S_1															
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	57	49	d1	c6	2f	33	74	fb	95	6d	82	ea	0e	b0	a8	1c	6c	da	c3	e9	4e	9d	0a	3d	b8	36	b4	38	13	34	0c	d9
1.	28	d0	4b	92	5c	ee	85	b1	c4	0a	76	3d	63	f9	17	af	bf	74	94	8f	b7	9c	e5	dc	9e	07	49	4f	98	2c	b0	93
2.	bf	a1	19	65	f7	7a	32	20	06	ce	e4	83	9d	5b	4c	d8	12	eb	cd	b3	92	e7	41	60	e3	21	27	3b	e6	19	d2	0e
3.	42	5d	2e	e8	d4	9b	0f	13	3c	89	67	c0	71	aa	b6	f5	91	11	c7	3f	2a	8e	a1	bc	2b	c8	c5	0f	5b	f3	87	8b
4.	a4	be	fd	8c	12	00	97	da	78	e1	cf	6b	39	43	55	26	fb	f5	de	20	c6	a7	84	ce	d8	65	51	c9	a4	ef	43	53
5.	30	98	cc	dd	eb	54	b3	8f	4e	16	fa	22	a5	77	09	61	25	5d	9b	31	e8	3e	0d	d7	80	ff	69	8a	ba	0b	73	5c
6.	d6	2a	53	37	45	c1	6c	ae	ef	70	08	99	8b	1d	f2	b4	6e	54	15	62	f6	35	30	52	a3	16	d3	28	32	fa	aa	5e
7.	e9	c7	9f	4a	31	25	fe	7c	d3	a2	bd	56	14	88	60	0b	cf	ea	ed	78	33	58	09	7b	63	c0	c1	46	1e	df	a9	99
8.	cd	e2	34	50	9e	dc	11	05	2b	b7	a9	48	ff	66	8a	73	55	04	c4	86	39	77	82	ec	40	18	90	97	59	dd	83	1f
9.	03	75	86	f1	6a	a7	40	c2	b9	2c	db	1f	58	94	3e	ed	9a	37	06	24	64	7c	a5	56	48	08	85	d0	61	26	ca	6f
a.	fc	1b	a0	04	b8	8d	e6	59	62	93	35	7e	ca	21	df	47	7e	6a	b6	71	a0	70	05	d1	45	8c	23	1c	f0	ee	89	ad
b.	15	f3	ba	7f	a6	69	c8	4d	87	3b	9c	01	e0	de	24	52	7a	4b	c2	2f	db	5a	4d	76	67	17	2d	f4	cb	b1	4a	a8
c.	7b	0c	68	1e	80	b2	5a	e7	ad	d5	23	f4	46	3f	91	c9	b5	22	47	3a	d5	10	4c	72	cc	00	f9	e0	fd	e2	fe	ae
d.	6e	84	72	bb	0d	18	d9	96	f0	5f	41	ac	27	c5	e3	3a	f8	5f	ab	f1	1b	42	81	d6	be	44	29	a6	57	b9	af	f2
e.	81	6f	07	a3	79	f6	2d	38	1a	44	5e	b5	d2	ec	cb	90	d4	75	66	bb	68	9f	50	02	01	3c	7f	8d	1a	88	bd	ac
f.	9a	36	e5	29	c3	4f	ab	64	51	f8	10	d7	bc	02	7d	8e	f7	e4	79	96	a2	fc	6d	b2	6b	03	e1	2e	7d	14	95	1d

Figure 4.9: Tables S_0, S_1

4.4.3 Module M0

These two modules M0 and M1 performs two matrices with multiplication operations on $GF(2^8)$ (Galois Field) which is defined by the primitive polynomial $z^8 + z^4 + z^2 + 1$ to ensure an adequate level of security and resistance to attacks. Modules M0 and M1 takes the data from S0 and S1 respectively. At next figure we can see the matrix M0.

$$\begin{pmatrix} 0x01 & 0x02 & 0x04 & 0x06 \\ 0x02 & 0x01 & 0x06 & 0x04 \\ 0x04 & 0x06 & 0x01 & 0x02 \\ 0x06 & 0x04 & 0x02 & 0x01 \end{pmatrix}$$

Figure 4.10: Matrix M0.

The VHDL algorithm implementation of matrix M0 uses the following steps. M0 ma-

trix decomposed into three matrices.The result is the sum of multiplication in $GF(2^8)$ where the variables X_0, X_1, X_2, X_3 represent the inputs and the variables Y_0, Y_1, Y_2, Y_3 represent the output of the multiplication process. Module M0 implemented as follow.

$$\begin{pmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} 01 & 02 & 04 & 06 \\ 02 & 01 & 06 & 04 \\ 04 & 06 & 01 & 02 \\ 06 & 04 & 02 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 00 & 02 & 00 & 02 \\ 02 & 00 & 02 & 00 \\ 00 & 02 & 00 & 02 \\ 02 & 00 & 02 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

$$+ \begin{pmatrix} 00 & 00 & 04 & 04 \\ 00 & 00 & 04 & 04 \\ 04 & 04 & 00 & 00 \\ 04 & 04 & 00 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

After that step, some variables that help the multiplication process are shown.

$$\begin{cases} A_0 = X_0 \oplus X_1 \\ A_1 = X_2 \oplus X_3 \\ B_0 = X_0 \oplus X_2 \\ B_1 = X_1 \oplus X_3 \end{cases} \quad \begin{cases} C_0 = \{02\} \times B_0 \\ C_1 = \{02\} \times B_1 \\ D_0 = \{04\} \times A_0 \\ D_1 = \{04\} \times A_1 \end{cases} \quad \begin{cases} Y_0 = C_1 \oplus D_1 \oplus X_0 \\ Y_1 = C_0 \oplus D_1 \oplus X_1 \\ Y_2 = C_1 \oplus D_0 \oplus X_2 \\ Y_3 = C_0 \oplus D_0 \oplus X_3 \end{cases}$$

For the multiplication with 0x02 and 0x04 two functions (F2 and F4) were constructed.

4.4.4 Module M1

For that module same logic with module M0 were. Here there is two functions (F2 and F8) that represent the multiplication by 0x02 and 0x08.

$$\begin{pmatrix} \texttt{0x01} & \texttt{0x08} & \texttt{0x02} & \texttt{0x0a} \\ \texttt{0x08} & \texttt{0x01} & \texttt{0x0a} & \texttt{0x02} \\ \texttt{0x02} & \texttt{0x0a} & \texttt{0x01} & \texttt{0x08} \\ \texttt{0x0a} & \texttt{0x02} & \texttt{0x08} & \texttt{0x01} \end{pmatrix}$$

Figure 4.11: Matrix M1.

M1 matrix decomposed into three matrices.The result is the sum of multiplication

in $GF(2^8)$ where the variables X_0, X_1, X_2, X_3 represent the inputs and the variables Z_0, Z_1, Z_2, Z_3 represent the output of the multiplication process. Module M1 implemented as follow.

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{pmatrix} = \begin{pmatrix} 01 & 08 & 02 & 0a \\ 08 & 01 & 0a & 02 \\ 02 & 0a & 01 & 08 \\ 0a & 02 & 08 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} + \begin{pmatrix} 00 & 00 & 02 & 02 \\ 00 & 00 & 02 & 02 \\ 02 & 02 & 00 & 00 \\ 02 & 02 & 00 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

$$+ \begin{pmatrix} 00 & 08 & 00 & 08 \\ 08 & 00 & 08 & 00 \\ 00 & 08 & 00 & 08 \\ 08 & 00 & 08 & 00 \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix}$$

After that some variables that helps the multiplication process are shown similarly with MO construction.

$$\begin{cases} A_0 = X_0 \oplus X_1 \\ A_1 = X_2 \oplus X_3 \end{cases} \quad \begin{cases} C_0 = \{02\} \times A_0 \\ C_1 = \{02\} \times A_1 \end{cases} \quad \begin{cases} Z_0 = C_1 \oplus D_1 \oplus X_0 \\ Z_1 = C_1 \oplus D_0 \oplus X_1 \end{cases}$$

$$\begin{cases} B_0 = X_0 \oplus X_2 \\ B_1 = X_1 \oplus X_3 \end{cases} \quad \begin{cases} D_0 = \{08\} \times B_0 \\ D_1 = \{08\} \times B_1 \end{cases} \quad \begin{cases} Z_2 = C_0 \oplus D_1 \oplus X_2 \\ Z_3 = C_0 \oplus D_0 \oplus X_3 \end{cases}$$

4.4.5 Module F0

F0 module has structural implementation with components of M0, S0 and S1 modules.This module has as input 32-bit with text and 32-bit with round keys. The text input of 32-bit is divided into 4 inputs of 8-bit(x_0, x_1, x_2, x_3) which are xored first with 8-bit round keys inputs(k_0, k_1, k_2, k_3). The result of that operations comes as input in 4 S-boxes (two S0 and two S1) and the output of that modules being connected to the M0 module, ultimately generating 32-bit output Y from the module F0.

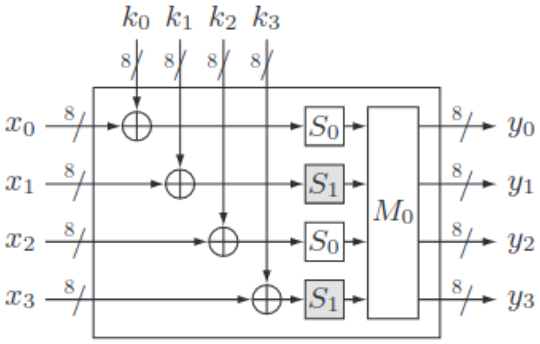


Figure 4.12: Function F0 of Clefia.

For VHDL implementation of F0 module the input is 64-bit (32-bit text and 32-bit key) and the output is 32-bit. The inputs are subdivided into 8 bytes (4 text bytes and 4 key bytes) in order to perform F0-specific calculations.

4.4.6 Module F1

The F1 module has a similar structure with F0 module were the 8 bytes input are being 4 bytes for the text (X_0, X_1, X_2, X_3) and xor operations with 4 bytes key (k_0, k_1, k_2, k_3), the outputs are connected to the inputs of the S modules (two S0 modules and two S1), and also M1 matrix exist instead of M0 from function F0.

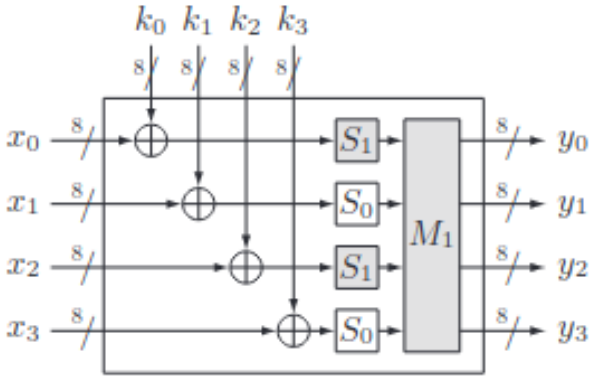


Figure 4.13: Function F1.

For VHDL implementation of F1 module the input is 64-bit (32-bit text and 32-bit key) and the output is 32-bit. The inputs are subdivided into 8 bytes (4 text bytes and 4 key bytes) in order to perform F1-specific calculations.

4.4.7 Module GF4N

This module includes the so-called Generalized Feistel Network which is composed of modules such as F0 and F1 and a few more modules and circuits aiming to meet the specifies of the algorithm, such as:

- **Module WK**: Perform Whitening Key operations in which two original keywords are added through an xor operation with two words of text, at the initial and the final round of encryption process.
- **Module Shift**: Perform permutation (shift the data lines to the left) before the next round except the last.
- **MUX1**: This mux is responsible for the selection of data from the input of GF4N or the WK module that will be submitted to modules F0,F1 and shift.
- **MUX2**: This mux is responsible for selecting the data to be submitted to the WK module. Either input plaintext, either the output from functions F0,F1
- **MUX3**: This mux is responsible to select at the last round the data which are not shifted.
- **MUX4**: This mux is responsible for the selections of the output data in which, in the last round of encryption the selected data is derived from WK module.

The GF4N module described in VHDL is formed by the instances of modules F0,F1,WK,SHIFT and multiplexer circuits.

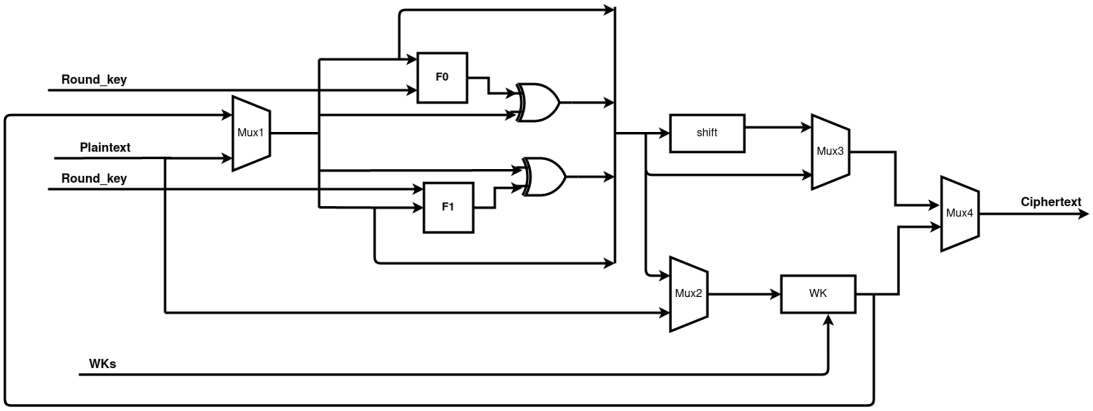


Figure 4.14: GF4N Diagram.

4.4.8 Module Key Schedule

This module aims to meet the key generation operations Round Keys, which are used each round during the encryption process.The version of CLEFIA implemented at this thesis uses an original key with a size of 128 bits and for this architecture a total of 36 32-bit keys are generated, two for each round of encryption, of a total of 18 rounds of encryption.

For generation of Round Keys an intermediate key of 128 bits is first required. This intermediate key is generated from the original key, which is subjected to GF4N for 12 rounds (without WKs) and for the L generation process the round keys are predefined constants which in this implementation comes from a ROM memory module. At the end of 12 rounds the K is stored in a register and will be used to generate the round keys for the encryption process.

During the encryption process, round keys are generated from xor operations and permutations that used from the intermediate key L, the original key and predefined constants.WK which is used in the first and the last round of encryption, is generated by splitting the original key of 128 bits in four words(32 bits each).

(Generating L from K)
Step 1. $L \leftarrow GFN_{4,12}(CON_0^{(128)}, \ldots, CON_{23}^{(128)}, K_0, \ldots, K_3)$

(Expanding K and L)
Step 2. $WK_0|WK_1|WK_2|WK_3 \leftarrow K$
Step 3. For $i = 0$ to 8 do the following:
$T \leftarrow L \oplus (CON_{24+4i}^{(128)} \mid CON_{24+4i+1}^{(128)} \mid CON_{24+4i+2}^{(128)} \mid CON_{24+4i+3}^{(128)})$
$L \leftarrow \Sigma(L)$
if i is odd: $T \leftarrow T \oplus K$
$RK_{4i}|RK_{4i+1}|RK_{4i+2}|RK_{4i+3} \leftarrow T$

Figure 4.15: Generating Keys.

The Key schedule module is composed of the following modules and circuits:

- **Module Constant**: A memory module that stores the predefined constants defined by the CLEFIA developers being 60 constants of 32 bits, which are used in the generation of L and round keys.
- **Register L key**: Register for intermediate key L.

- **Module T key**: This module performs xor operations at each encryption round with part of the intermediate key L (64-bit), part of the original key K (64 bit) and two constants (64 bit).
- **Double Swap**: This module performs permutation operations with the data of the output of the L key register.
- **MUX1**: This mux is responsible for selecting the input of the L register, and during the process of generating the intermediate key L(12 rounds), the selection is provided from the output of the GF4N module or the selected input comes from the output of the Double Swap permutation module during the generation process of the Round Keys.
- **MUX2**: This mux is responsible for selecting the output of the round keys, which during the generation of L comes from the constant module and during the generation of Round Keys output comes from the T key module.

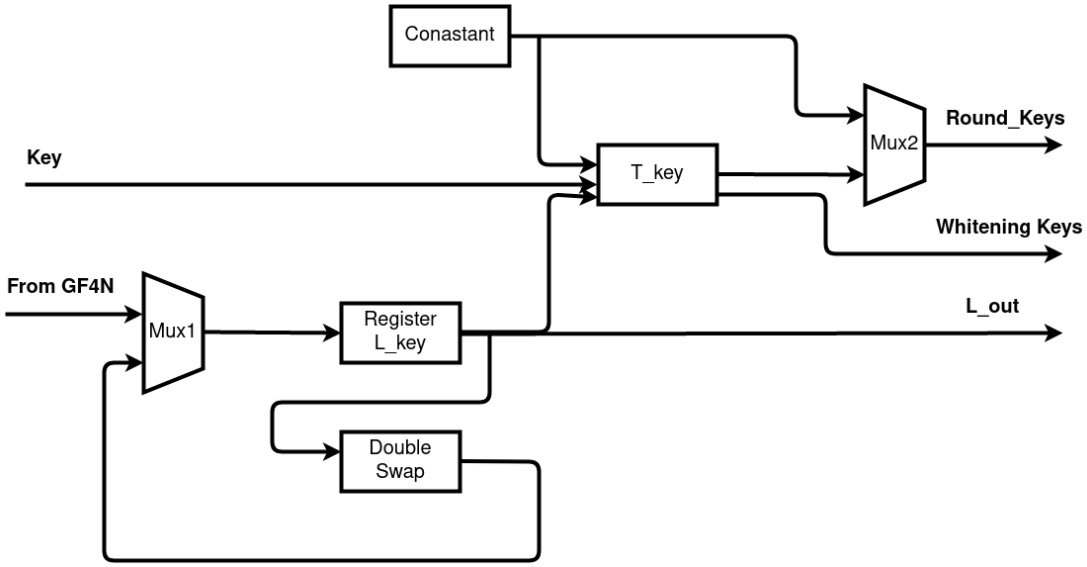


Figure 4.16: Module key schedule

4.4.9 Module Control

The control of modules and selectors for multiplexers for taking the correct signals for data lines at each round either for key generation or for encryption process is performed by this control module.

This module is a state machine module with 5 States with the description below:

- **SM STOP**: Stop the process or initial parameters.
- **SM LOAD**: Load the register with Plaintext(128 bit) and Key register(128 bit).
- **SM L KEY**:At this state the intermediate key L is generated, which is used in the process of generating keys for the rounds. 12 rounds (iterations) are executed in this state (counter 1 to 12).
- **SM ENC**:Run the necessary rounds of encryption. In this implementation are 18 rounds of encryption performed (counter 13 to 30).
- **SM READY**:At this state signals for reading the ciphertext after the encryption process are implemented.

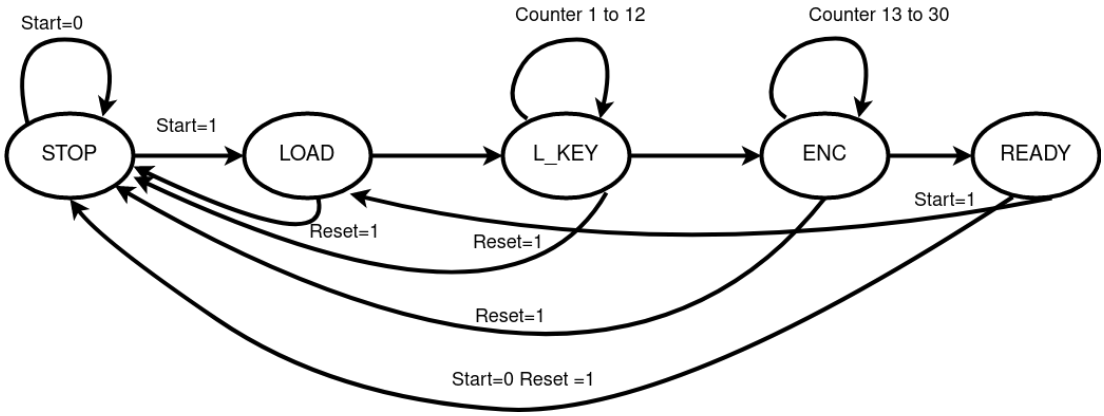


Figure 4.17: State Machine

4.5 Tools

For this thesis, the Vivado design tool version 2021.1 was used to implement the algorithms in the hardware and the Vitis tool version 2021.1 to test the algorithms in the software.

4.5.1 Communication between Software and Hardware

The use of FPGA and and a processor combines the benefits of a programmable logic (PL) component and microprocessor. This solution is actually adopted to validate the functionality of such hardware applications that require important complexity and large data to deal with. Furthermore, heterogeneous on-chip designs consume less power and have lower cost and higher reliability than multi chip systems. Inside the Zynq

architecture, the software is programmed into the processing system (PS) and hardware implementation is located in the PL. Xilinx adopted the Advanced extensible Interface AXI4 of the Advanced Micro controller Bus Architecture (AMBA) protocols it is used to exchange data between PS and PL in an efficient and flexible way.

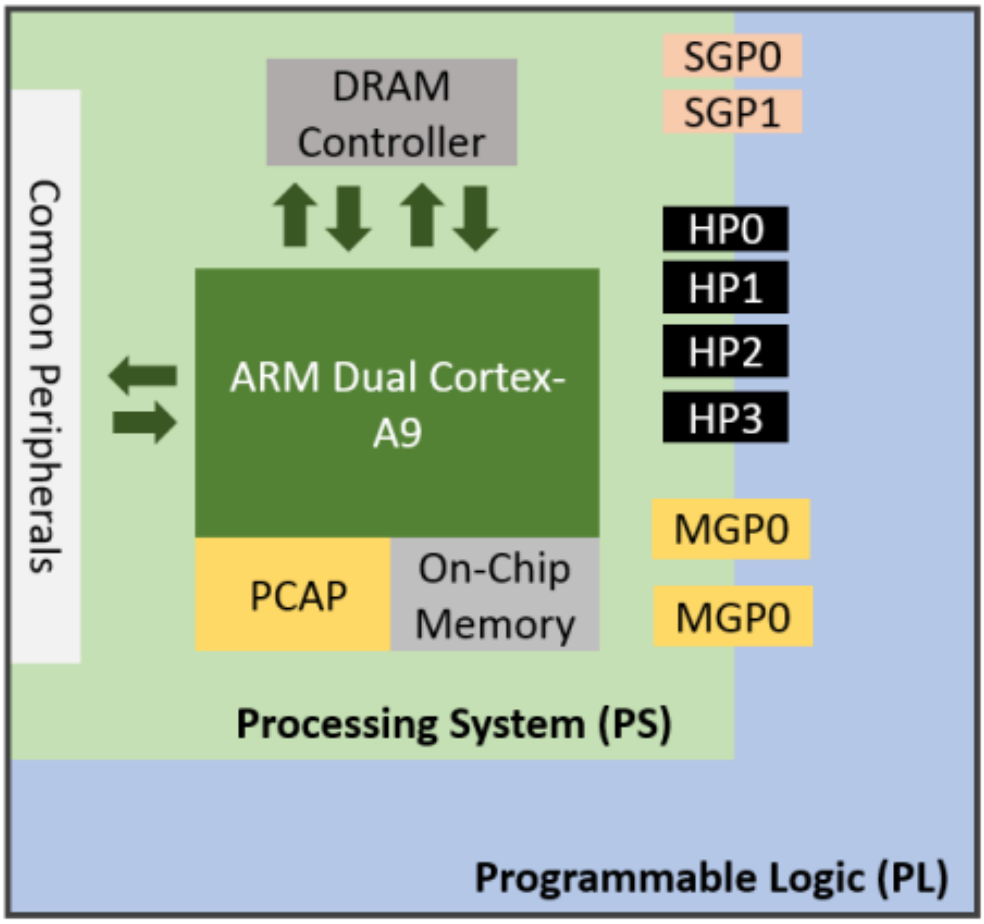


Figure 4.18: Communication between PS and PL

4.5.2 AXI Protocol

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second major version of AXI, AXI4.

There are three types of AXI4 interfaces:

- **AXI4:** For high-performance memory-mapped requirements.
- **AXI4-Lite:** For simple, low throughput memory-mapped communication (for example, to and from control and status registers).

- **AXI4-Stream:** For high-speed streaming data.

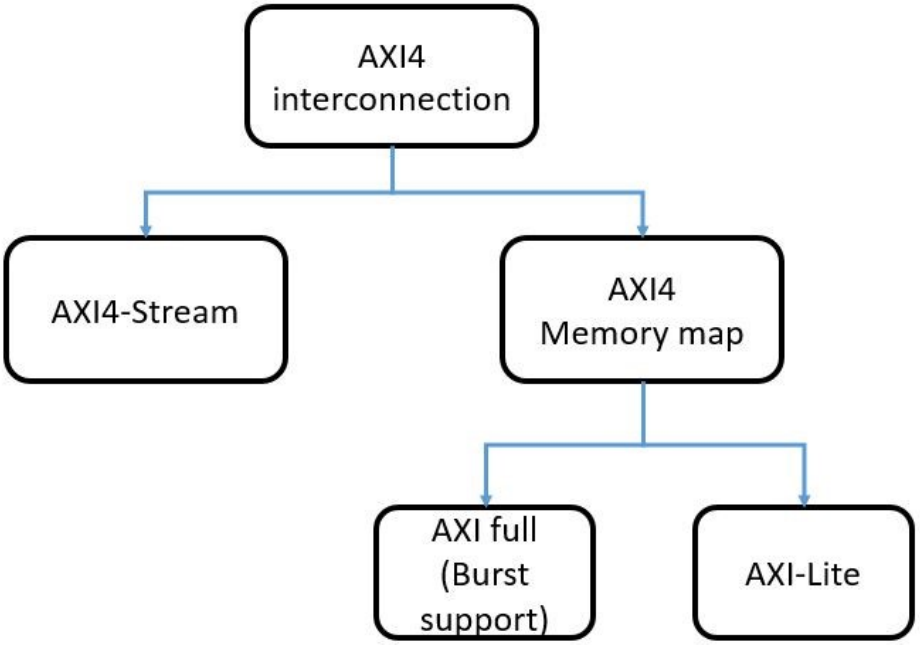


Figure 4.19: AXI interconnection flowchart.

4.5.3 How AXI Works

The AXI specifications describe an interface between a single AXI master and AXI slave, representing IP cores that exchange information with each other. Multiple memory-mapped AXI masters and slaves can be connected together using AXI infrastructure IP blocks. The Xilinx AXI Interconnect IP and the newer AXI Smart Connect IP contain a configurable number of AXI-compliant master and slave interfaces, and can be used to route transactions between one or more AXI masters and slaves. The AXI Interconnect architecture use a traditional, monolithic crossbar approach;

Both AXI4 and AXI4-Lite interfaces consist of five different channels:

- Read Address Channel.
- Write Address Channel.
- Read Data Channel.
- Write Data Channel.

- Write Response Channel.

Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. The limit in AXI4 is a burst transaction of up to 256 data transfers. AXI4-Lite allows only one data transfer per transaction.

The following Fig 4.20 shows how an AXI4 read transaction uses the read address and read data channels.

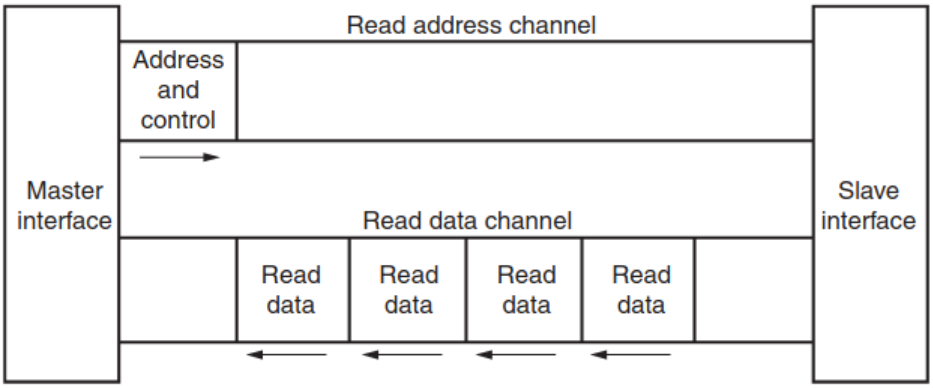


Figure 4.20: Channel Architecture of Reads

Fig 4.21 shows how a write transaction uses the write address, write data, and write response channels.

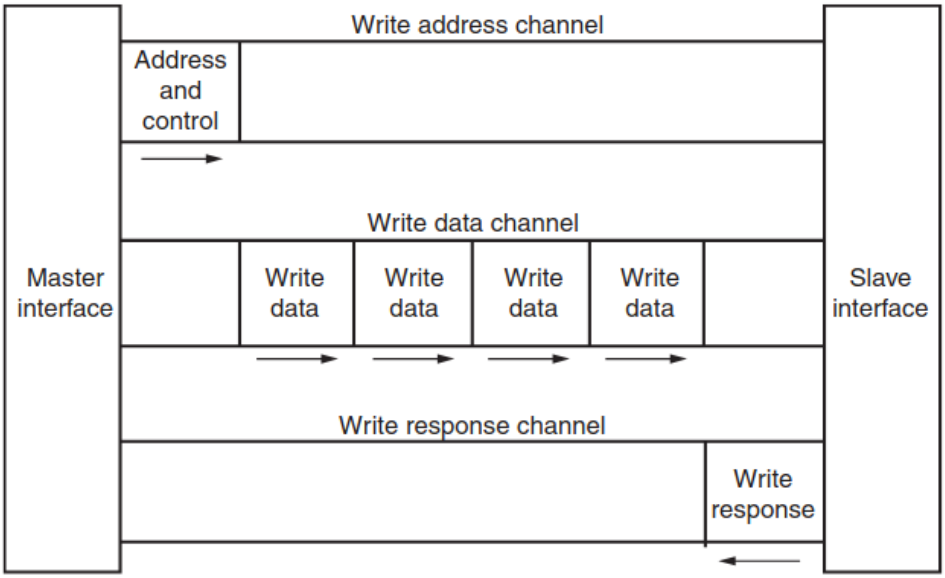


Figure 4.21: Channel Architecture of Writes

Fig 4.22 shows a transaction for AXI4 Stream interface.

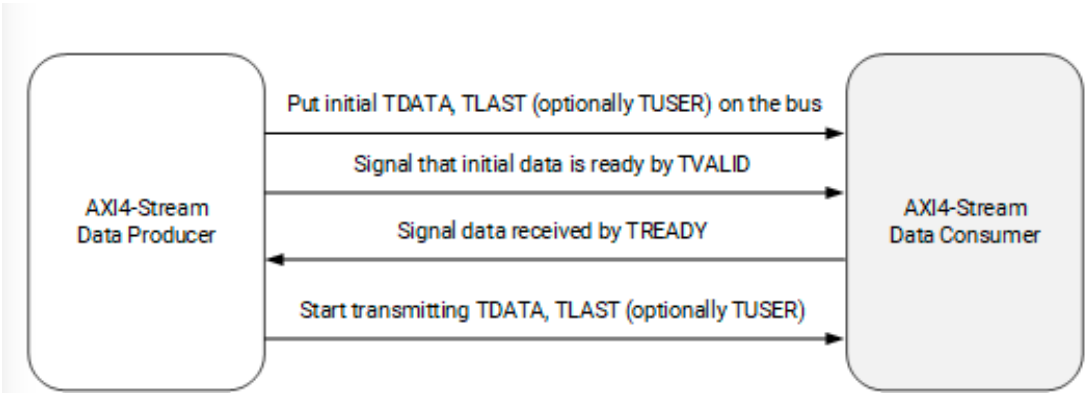


Figure 4.22: AXI4-Stream Handshake

As shown in the preceding figures, AXI4:

- Provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional data transfer.
- Requires a single address and then bursts up to 256 words of data.

The AXI4 protocol describes options that allow AXI4-compliant systems to achieve very high data throughput. Some of these features, in addition to bursting, are: data upsizing and downsizing, multiple outstanding addresses, and out-of-order transaction processing.

Data must typically transit through the processor's DDR before being sent over PL in one of three methods (AXI Lite, AXI Memory Mapped, AXI4 Stream).

- **Memory-Mapped Protocol:** In memory-mapped protocols (AXI3, AXI4, and AXI4-Lite), all transactions involve the concept of transferring a target address within a system memory space and data.
- **AXI4-Stream Protocol:** The AXI4-Stream protocol is used as a standard interface to connect components that wish to exchange data. The interface can be used to connect a single master, that generates data, to a single slave, that receives data. The protocol can also be used when connecting larger numbers of master and slave components.The protocol supports multiple data streams using the same set of shared wires, allowing a generic interconnect to be constructed that can perform upsizing, downsizing and routing operations.

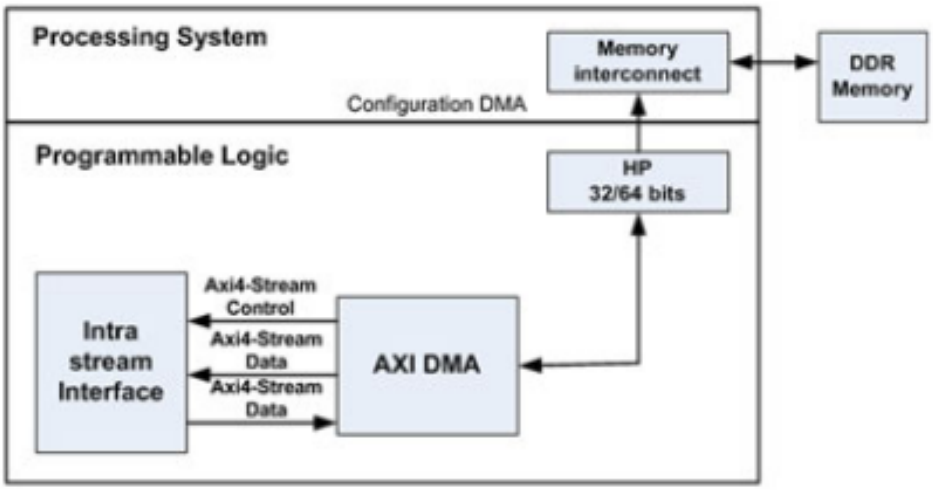


Figure 4.23: Communication between software and hardware using AXI-stream interface.

- **AXI Direct Memory Access (AXI DMA)**: The AXI DMA provides high-bandwidth direct memory access between the AXI4 memory mapped and AXI4 memory stream interfaces. Its optional scatter gather capabilities also of load data movement tasks from Central Processing Unit (CPU) in processor-based systems. Initialization, status, and management registers are accessed through AXI4-Lite slave interface. Figure 4.24 illustrates the functional composition of the core.

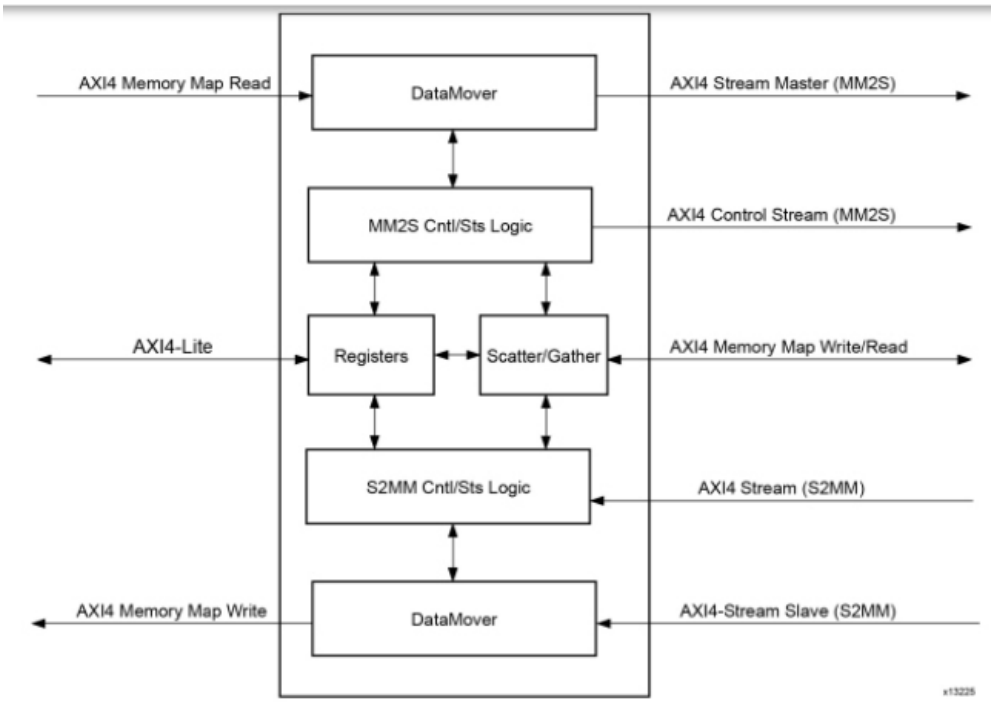


Figure 4.24: DMA Core.

Chapter 5

Results

This section presents the design results of the hardware accelerators for the CLEFIA, PRESENT and SIMON algorithms focusing on the performance (in terms of throughput), power and area (FPGA resource usage). The power and performance results are compared with purely software implementations of the algorithms in different platforms.

The first subsection presents the four different software platforms that have been used to deploy the algorithms. These platforms represent a wide range of systems varying from small footprint, low-power ARM-based systems up to high performance data-center servers. As such, the proposed hardware accelerated solution can be evaluated against a wide range of systems that may be considered to be used at the edge environment.

As mentioned both performance and power consumption (and efficiency) results are presented. For the former, the time required to encrypt a block of data at the unit of time (throughput) is used in order to determine the performance of each platform. For the latter, power consumption estimations are used in order to measure the power required during the encryption of data in each case. More details about the estimation will be provided in the next subsection.

The results demonstrate that the proposed hardware accelerated solution manages to outperform the software-based deployments by significant margins. In terms of throughput, the FPGA-based solution provides at least one order of magnitude higher throughput than the highest performance offered by a software solution. This is achieved for significantly less power consumption (ranging from 3 to 10 times lower), resulting in a very favourable performance per watt performance.

5.1 Software Implementations and Performance/Power Comparison Methodology

Before proceeding with the presentation of the results of the proposed hardware-accelerated solutions, it is important to present the baseline against which they will be evaluated, as well as the methodology that has been followed in order to measure performance and power and perform comparisons.

5.1.1 Software Platforms

As mentioned in the beginning of this thesis, the proposed hardware accelerated solution aims to enhance the performance and power efficiency of CPU-based solutions. A number of platforms with different characteristics in terms of cost, performance, power consumption and physical size has been chosen to execute reference software implementations of the three encryption algorithms. The following table describes these platforms ordering them from the simplest ones to the most complex.

Platform	**Processor**	**Memory**	**Form Factor**	**Cost**	**Reference**
Digilent/AVNET ZedBoard	ARM Cortex A9 (32-bit) on Zynq 7020 SoC running @667MHz	512 MB DDR3	Development Board	~$500	[40]
Xilinx Kria KV260 AI Starter Kit	ARM Cortex A53 (64-bit) on Zynq UltraScale+ MPSoC running @1.5GHz	4 GB DDR4	SOM + Carrier Card (similar to RPi / nVIdia Jetson Nano form factor)	~$199	[41]
Generic Laptop	Intel Core i5-4200U (64-bit) running @1.6GHz	16 GB DDR3L	Laptop	~$1000	[42]
Dell PowerEdge R530 (kronos.mhl.tuc.gr)	2x IntelXeon E5-2630 v4 CPUs running @2.2GHz (up to 3.1 GHz Turbo)	256 GB DDR4	Rack Server	~$8000	[43],[44]

Table 5.1: Description of the platforms

5.1.2 Software Implementations

For each of the three encryption algorithms, reference software implementations have been chosen. These codes are publicly available, well-documented and broadly accepted in relevant literature. It cannot be claimed that they represent the highest

performance software implementations for the three algorithms, however to the best of our knowledge they represent the best openly available solutions that can be accessed by anyone and the results presented in this thesis can be verified by any third party. In particular, the following open-source codes have been used:

1. Reference CLEFIA implementation published by Sony Corporation [45].
2. D. Klose's (University of Bochum) reference implementation of PRESENT cipher [46].
3. Crypto Library's implementation of the SIMON algorithm available in [47].

For all three ciphers, their source code has been compiled and executed on the four available platforms. For the Zynq-based platforms (Zedboard and Kria), the gcc compiler included with the Vitis tools (v2021.1) has been employed and the software was compiled targeting bare metal execution on the ARM Cortex processors (no OS). For the laptop platform, the OS used was Ubuntu Linux 20.04 and the gcc compiler (v9.4.0) has been employed. Lastly, the server system was running CentOS Linux v.7.6 and the compiler employed was gcc v4.8.5.

5.1.3 Performance and Power Comparison Methodology

In order to measure performance on all platforms, the effective throughput defined as number of encrypted bits per second has been calculated. It should be noted that the time required to encrypt a single plaintext block even in the simplest platforms is quite small. As such, to have statistically important information and in order to reduce the effect of random variations from execution to execution (e.g. from an interrupt or a system call), a plaintext in the order of several megabytes has been randomly created and this same plaintext has been used as input for the encryption processes. The encryption process for each algorithm has been repeated several times and the average throughput of all executions is reported.

A special note has to be made concerning the hardware accelerated solutions. The reported performance numbers cover the entire process and not only the execution time of the hardware kernel. This means that the times used include also the data transfers between the processor (software components) and the hardware kernel and therefore the comparisons between the software platforms and the hardware accelerated solution are equivalent.

To perform power measurements and consequently examine the efficiency (performance per watt) of each platform when executing each encryption algorithm, we relied on power estimations.

To compare the power consumption of the 5.3, 5.17, 5.11 . These results present the power consumption estimation on the Zynq / Zynq US+ chips covering both the Programmable System (ARM processors and hardwired blocks) and the Programmable Logic (FPGA resources) parts. As such, the estimations cover both the power consumption of the software components as well as the power consumption of the hardware kernels.

To estimate the power consumption in the laptop and server platforms when the encryption algorithms are executed, the powerstat [48] tool has been employed. Powerstat measures the power consumption of a machine using the battery stats (ACPI) or the Intel RAPL interface (Running Average Power Limit). The output of the tool is like vmstat [49] but also shows power consumption statistics. At the end of a run, powerstat will calculate the average, standard deviation and min/max of the gathered data, covering the processor and direct attached memory power domains.

The Powerstat tool was executed in parallel with the execution of the encryption algorithms to monitor system behavior during the runs. The specific flags used during execution were the "-R -c -z" ones, specifying a readout from the RAPL interface and a monitoring of the processor's C-states (for more information about these flags, please refer in the tool's documentation: [48]. Each execution of powerstat runs for 60 seconds and then displays the average power consumption value. At least 15 minutes of the encryption processes were used and therefore multiple readouts from the powerstat tool were gathered. At the end of this process, the average value was calculated and this value was used in the comparisons with the power estimations from the vivado tool presented in Figures 5.7, 5.13, 5.19.

It should be noted that the powerstat tool nor any other similar tool could be used in the case of the Zedboard and Kria ARM platforms when executing only software versions of the encryption algorithms. This was due to the fact that no OS was used (bare metal execution) and not any RAPL or similar interfaces were provided by these platforms and associated development tools. As a result, in order to have an estimation of the power consumption of the the programmable system in these platforms (that is the ARM processor complex), a dummy hardware kernel was created that performed

practically no computation (inverse of a small number of bits) and the overall software/hardware application was passed through the vivado tools Figures 5.4, 5.5. . This way a power consumption estimation was provided by the tools, according to what is described in the beginning of this section.

To calculate the performance per watts on a specific platform for each of the three algorithms we use the following method. Given the platform and an algorithm: running the algorithm on the platform takes T seconds while consuming W watts. The formula that links energy and power is:

$$\textbf{Energy} = \textbf{Power} \times \textbf{Time}$$

The unit of energy is the joule, the unit of power is the watt, and the unit of time is the second. So the performance per watt value is calculated as follow:

$$\textbf{PPW} = \frac{1}{\textbf{Energy}}$$

Tables 5.5, 5.13, 5.9 present the results of the 3 algorithms about performance, energy efficiency and power consumption while they are implemented on different platforms.

5.2 Results of Present Algorithm

In this subsection the results of cryptography algorithm implemented on the FPGA and the software are presented and discussed.

5.2.1 Area, Latency and Power performance

The Present algorithm implemented in this work performs the encryption process with 64-bit plaintext size and 80 bit key size. For the encryption process this algorithm needs 32 rounds. Figure 5.1 demonstrates the entire hardware-accelerated solution for the Present algorithm (see section 4 for a description of the different parts of their system and their function). Among the system components the custom IP is displayed with the name axi_stream_wrapper_0.

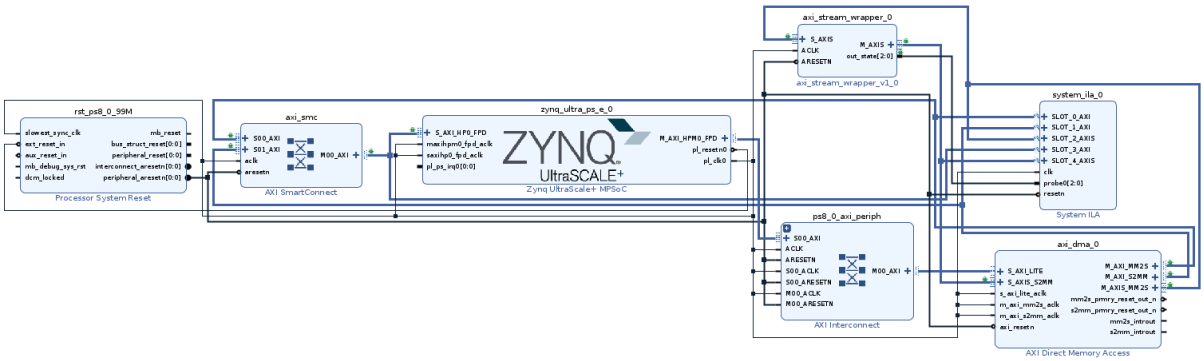


Figure 5.1: Present System.

The following table shows the utilization of the present core when the synthesis process is done.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
Present64/80	179	151	0	0	0	300	32

Table 5.2: Utilization of Present core Post-Synthesis stage.

The utilization of the entire system when the synthesis process is done is shown on the table below.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
System	5887	8188	31	7	9	300	32

Table 5.3: Utilization of System Post-Synthesis stage.

After the placement and the routing table 5.4 shows the results in terms of area of the final system including, the present core, the DMA core, the axi stream interface, axi interconnect and axi smart connect in the post-implementation stage.

Resource	Utilization	Available	Utilization %
LUT	5017	117120	4.28
LUTRAM	816	57600	1.42
FF	7611	234240	3.25
BRAM	9	144	6.25
BUFG	1	352	0.28

Table 5.4: System Present Utilization Post-Implementation stage.

This figure 5.2 shows the PRESENT algorithm mapped to the FPGA.

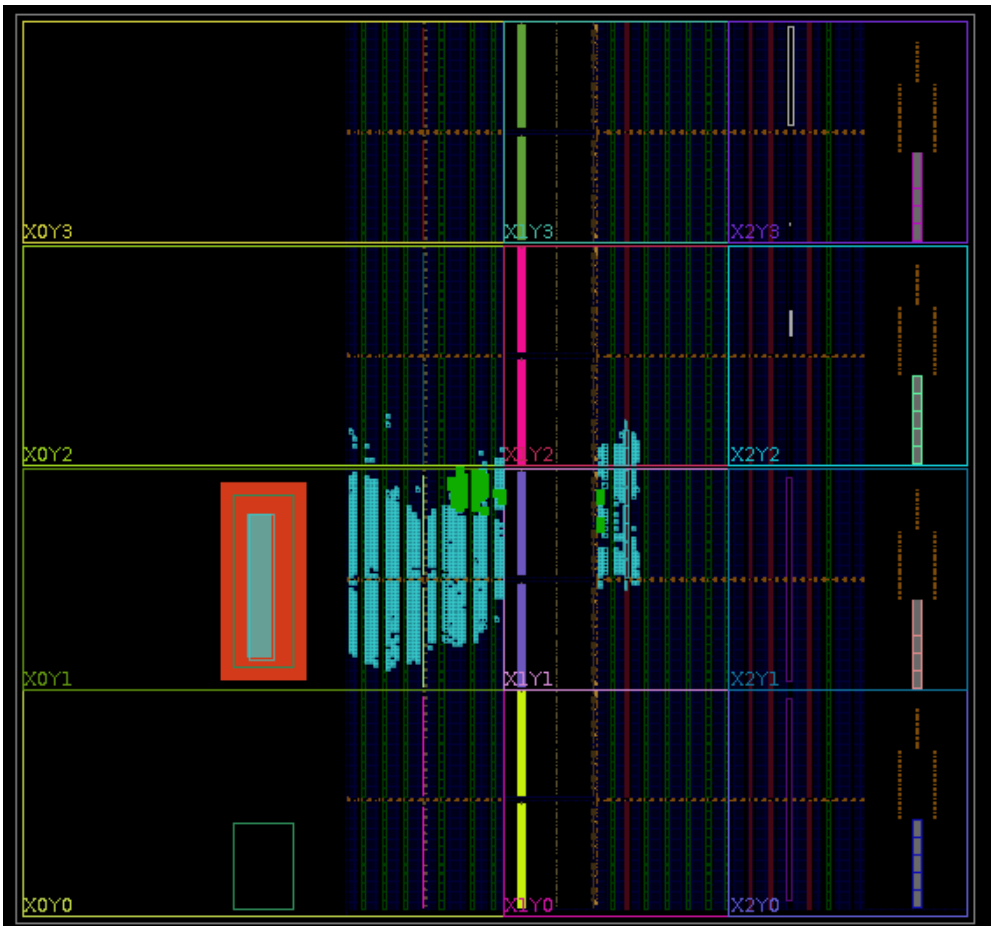


Figure 5.2: Present64/80 mapped on FPGA.

The On-Chip power consumption in the Static and Dynamic scenarios of the algorithm PRESENT is displayed in Figure 5.3.

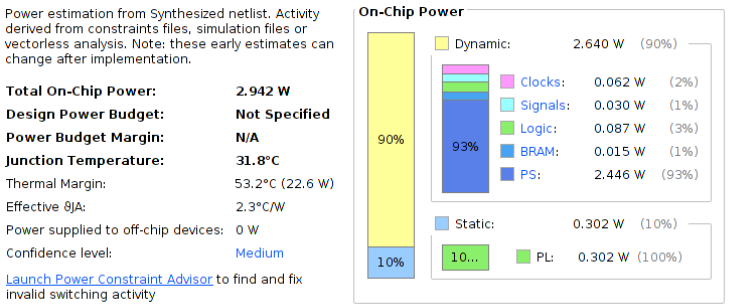


Figure 5.3: Power on chip of present 64/80 algorithm on the fpga.

Figure 5.3 shows the power consumption of present algorithm on different parts of the FPGA. It is observed that in Static state there is almost 0.301 W in the other hand at Dynamic state the power is 2.578 W.

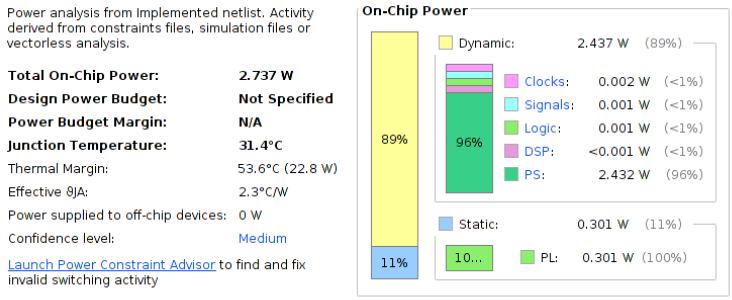


Figure 5.4: Power of kria platform on PS.

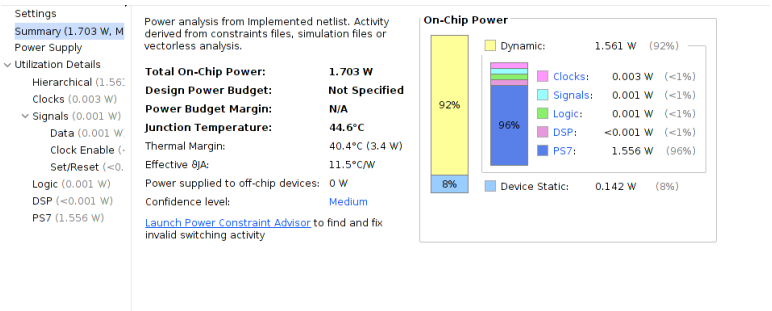


Figure 5.5: Power of zedboard platform on PS.

At figures 5.4 and 5.5 we see the power consumption on zedboard and kria platforms when a simple program is running. About kria the PS part consumes 2.432 W while on zedboard platform PS part consumes 1.556 W.

5.2.2 Present Software and Hardware comparison

In this subsection the results of the lightweight cryptography algorithm Present will be presented and discussed. These results simply comes from executing the Present algorithm numerous times. Due to Vivado AXI DMA IP restrictions, a straightforward transaction can only be completed with transfers of up to 64MB each. As a result, our design can return up to one million results at once with each initialization (1,000,000 * 64 = 61.0352MB).

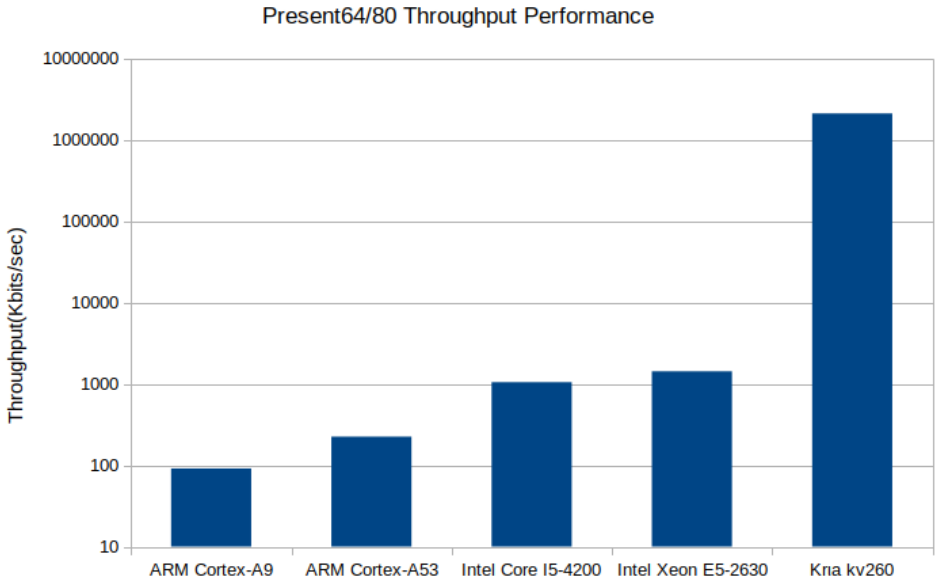


Figure 5.6: Present block cipher software performance.

Figure 5.6 shows the throughput measurements for the software implementation of the Present block cipher in the four available platforms as well as in the proposed hardware accelerated implementation on the KRIA KV260 system. A log scale is used for the Y axis of the graph. The best throughput at software design was achieved when the algorithm ran on the server kronos(Intel Xeon E5-2630) as expected. This approach gives an average throughput value of 1425Kbits/sec. On the other hand the zedboard platform (ARM Cortex-A9) which perform an average throughput of 91.095Kbits/sec shows the lowest performance. The hardware implementation on Kria KV260 FPGA achieves the best performance compared to all software implementations with an average throughput value of 2104Mbits/sec. An average throughput Speedup of 1.473x is obtained when comparing the the hardware accelerated implementation on Kria with the one on CPU Intel Xeon E5-2630.

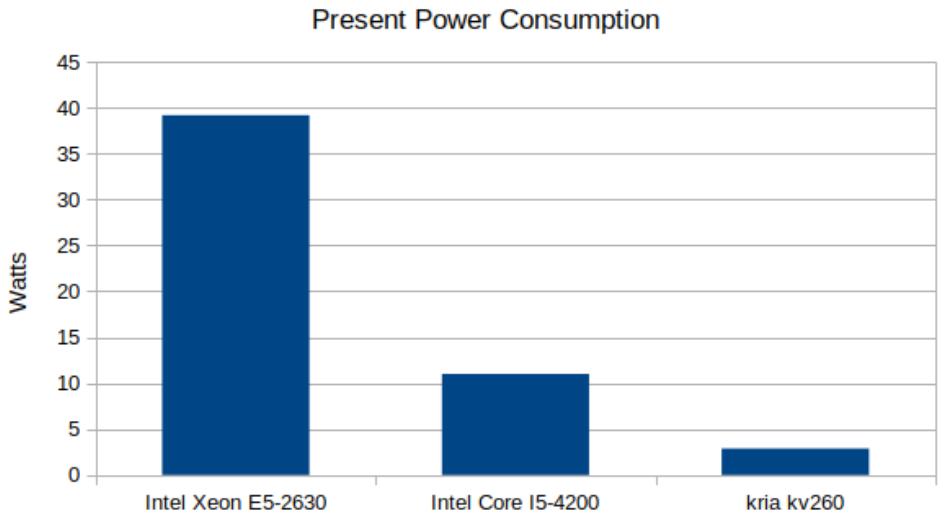


Figure 5.7: Comparison of power consumption of Present algorithm.

Making a comparison between the hardware and the software implementations of the algorithm regarding to the power consumption we notice that the hardware implementation of Present algorithm requires 2.942W on chip power while the software implementations on Intel Core I5-4200 and Intel Xeon E5-2630 require 11W and 39.17W respectively.

Platform	**Performance (1/Exec Time)**	**Power Consumption (Watts)**	**Energy efficiency (Performance per Watts)**
Kria KV260 (Software only)	0.0084	2.432	0,0034
ZedBoard (Software only)	0.0034	1.556	0,0022
Generic Laptop	0.045	11	0,0040
Server IntelXeon E5-2630 (kronos.mhl.tuc.gr)	0.053	39.17	0,0014
Kria KV260 (Hardware)	232.56	2.942	47,30

Table 5.5: Comparison between different platforms of Present algorithm

About the energy efficiency and performance as we can see in table 5.5 the implementation of the algorithm on hardware in both cases shows better results. After the calculation of performance per watt ratio we observe that FPGA implementation is

11825x more energy efficient from CPU implementation on generic laptop. The results of the energy efficiency are also depicted at the following graph. A log scale is used for the Y axis of the graph.

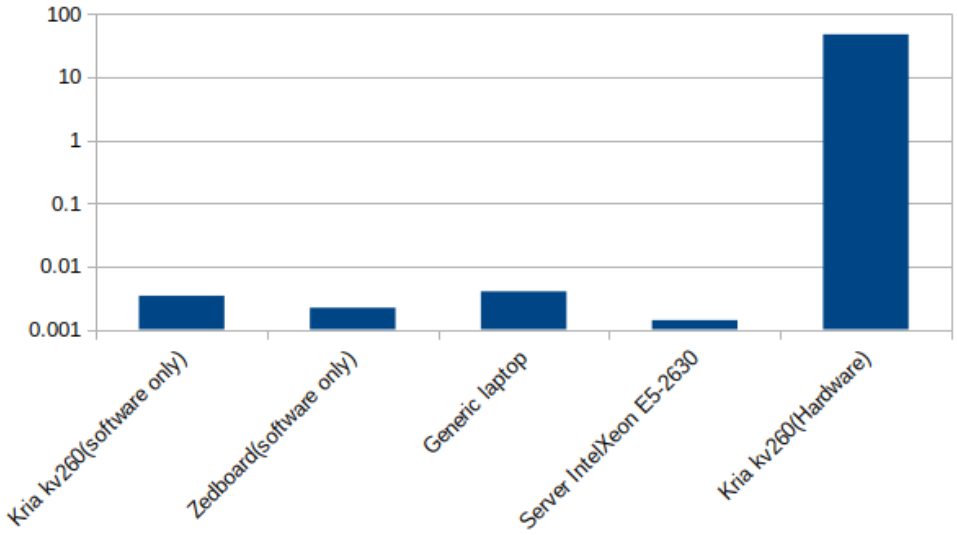


Figure 5.8: Energy efficiency of Present algorithm.

5.2.3 Results of Simon Algorithm

In this subsection the results of lightweight cryptography algorithm Simon for the implementations on FPGA and software are presented and discussed.

5.2.4 Area, Latency and Power performance

The Simon algorithm implemented in this work performs the encryption process with 64-bit plaintext size and 128 bit key size. For the encryption process this algorithm needs 44 rounds. Figure 5.9 demonstrates the entire hardware-accelerated solution for the Present algorithm (see section 4 for a description of the different parts of their system and their function). Among the system components the custom IP is displayed with the name (axi_stream_wrapper_0).

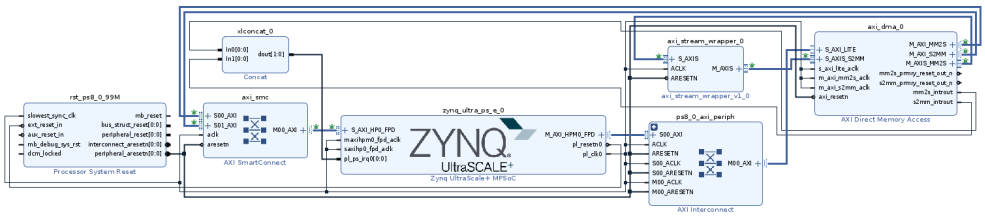


Figure 5.9: Simon System.

The following table shows the utilization of the Simon core when the synthesis process is done.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
Simon64/128	1410	132	0	0	0	50	1

Table 5.6: Utilization of Simon core Post-Synthesis stage.

The utilization of the entire system when the synthesis process is done is shown on the table below.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
System	7087	7967	31	7	9	50	1

Table 5.7: Utilization of System Post-Synthesis stage.

After the placement and the routing table 5.8 shows the results in terms of area of the final system including, the SIMON core, the DMA core, the axi stream interface, axi interconnect and axi smart connect in the post-implementation stage.

Resource	Utilization	Available	Utilization%
LUT	6227	117120	5.32
LUTRAM	816	57600	1.42
FF	7390	234240	3.15
BRAM	9	144	6.25
BUFG	1	352	0.28

Table 5.8: System Simon Utilization Post-Implementation stage.

- **Total On-Chip Power:** Total on-chip power is the power consumed internally

within the device, equal to the sum of device static power and design power. Its is also known as thermal power.

Figure 5.10 shows the Simon algorithm mapped on the FPGA.

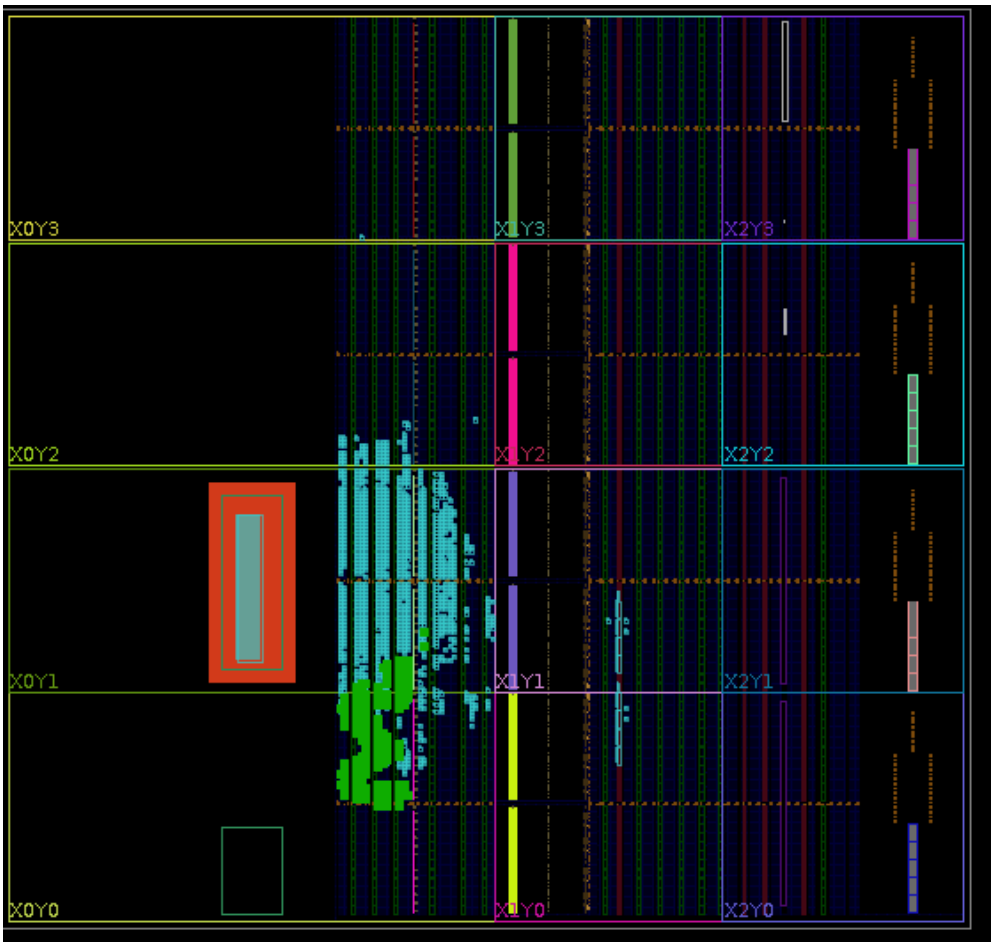


Figure 5.10: Simon64/128 mapped on FPGA .

The On-Chip power consumption in the Static and Dynamic scenarios of the algorithm Simon is displayed in Figure 5.11

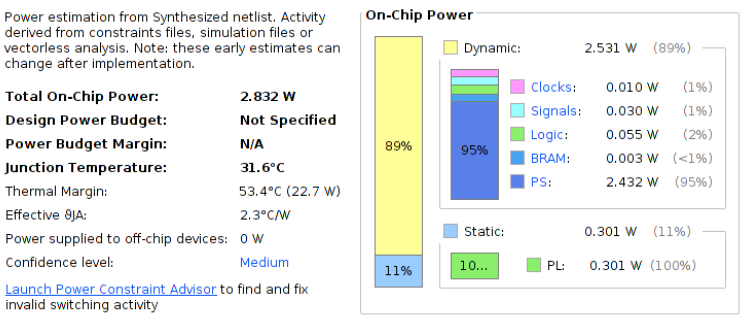


Figure 5.11: Power on chip of simon64/128 algorithm on the fpga. .

Figure 5.11 shows the power consumption of Simon algorithm on different parts of the FPGA. It is observed that in Static state there is almost 0.301 W, in the other hand at Dynamic state the power is 2.531 W.

5.2.5 Simon Software and Hardware comparison

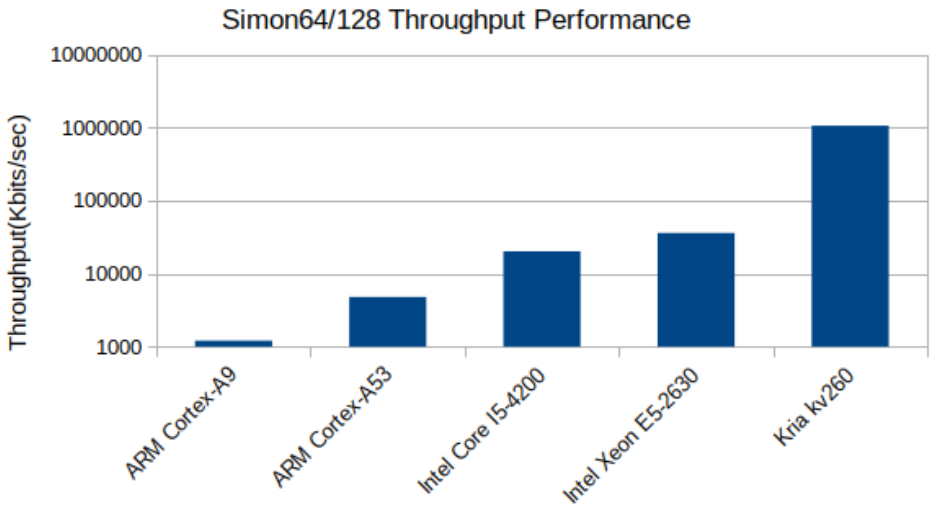


Figure 5.12: Simon64/128 Software Performance.

Figure 5.12 shows the throughput measurements for the software implementation of the Present block cipher in the four available platforms as well as in the proposed hardware-accelerated implementation on the Kria KV260 system. A log scale is used for the Y axis of the graph. The best software result was achieved when the algorithm was executed on the server kronos(Intel Xeon E5-2630) as expected. This approach gives an average throughput value of 36098Kbits/sec. On the other hand the zedboard platform (ARM Cortex-A9) which perform an average throughput of 1208Kbits/sec shows the lowest performance. The hardware implementation on Kria KV260 FPGA achieves the best performance compared to all software implementations with an average throughput value of 1057Mbits/sec. An average throughput Speedup of 29.36x is obtained when comparing the hardware accelerated implementation on Kria with the one on CPU Intel Xeon E5-2630.

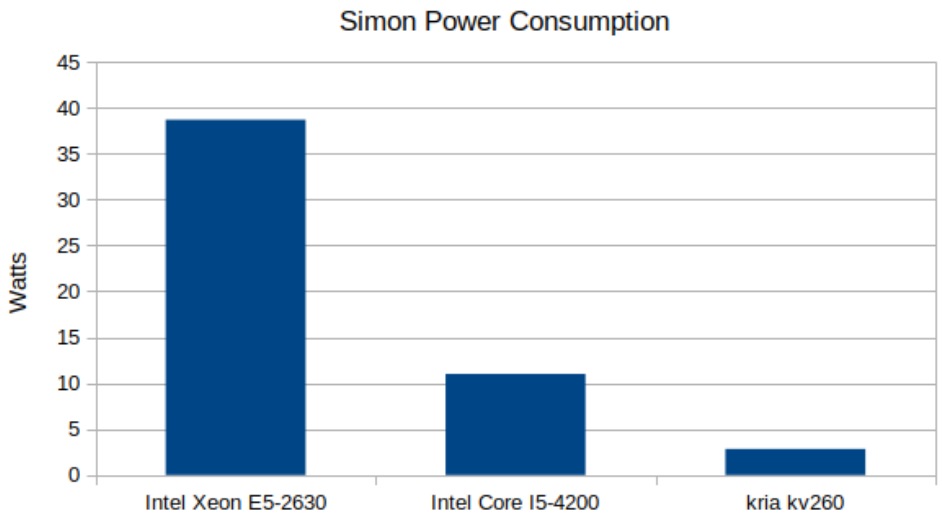


Figure 5.13: Comparison of power consumption of Simon algorithm.

Making a comparison between the hardware and the software implementations of the algorithm regarding to the power consumption as we can see on figure 5.13 we notice that the hardware implementation of Simon algorithm require 2.832W on chip power while the software implementations on Intel Core I5-4200 and Intel Xeon E5-2630 require 11W and 38.69W respectively.

Platform	**Performance (1/Exec Time)**	**Power Consumption (Watts)**	**Energy efficiency (Performance per Watts)**
Kria KV260 (Software only)	0.1197	2.432	0,0392
ZedBoard (Software only)	0.0302	1.556	0,0194
Generic Laptop	0.4975	11	0,0452
Server IntelXeon E5-2630 (kronos.mhl.tuc.gr)	0.9090	38.69	0,0142
Kria KV260 (Hardware)	86.967	2.828	30,850

Table 5.9: Comparison between different platforms of Simon algorithm

About the energy efficiency and performance as we can see in table 5.9 the implementation of the algorithm on hardware in both cases shows better results. After the calculation of performance per watt ratio we observe that FPGA implementation is

682.5x more energy efficient from CPU implementation on generic laptop. The results of the energy efficiency are also depicted at the following graph. A log scale is used for the Y axis of the graph.

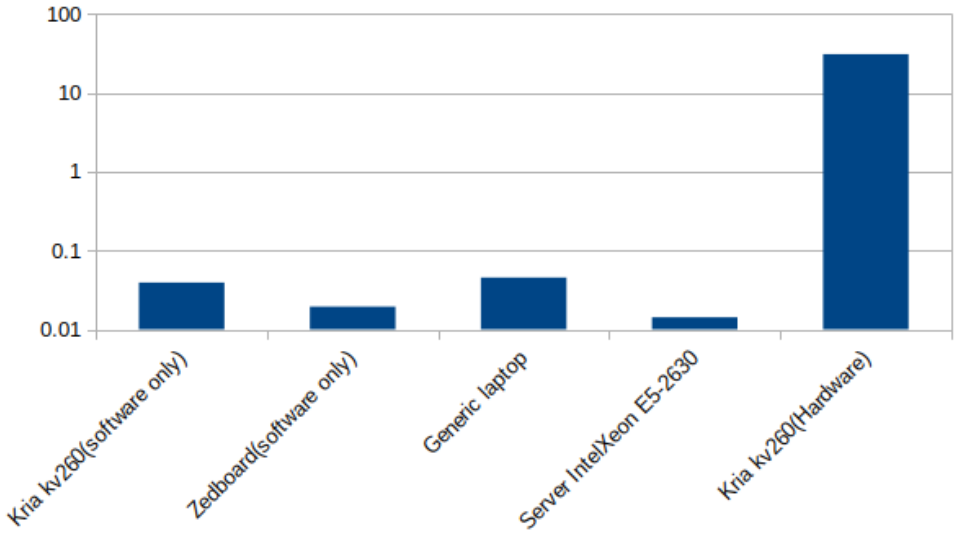


Figure 5.14: Energy efficiency of Simon algorithm.

5.2.6 Results of Clefia Algorithm

In this subsection the results of lightweight cryptography algorithm Clefia for the implementation on FPGA and software are presented and discussed.

5.2.7 Area, Latency and Power performance

The Clefia algorithm implemented in this work performs the encryption process with 128-bit plaintext size and 128 bit key size. For the encryption process this algorithm needs 18 rounds. Figure 5.15 demonstrates the entire hardware-accelerated solution for the Present algorithm (see section 4 for a description of the different parts of their system and their function). Among the system components the custom IP is displayed with the name (axi_stream_wrapper_0).

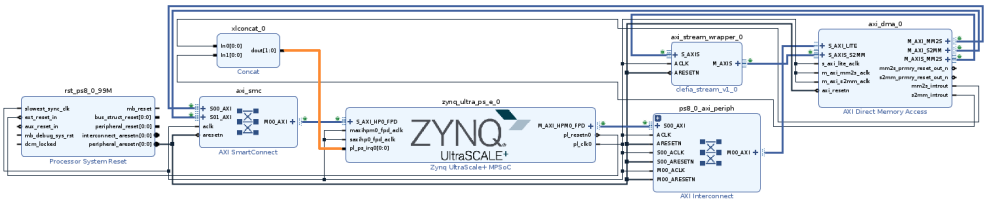


Figure 5.15: Clefia System.

The following table shows the utilization of the Clefia core when the synthesis process is done.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
Clefia128/128	1055	323	0	0	0	300	31

Table 5.10: Utilization of Clefia core Post-Synthesis stage.

The utilization of the entire system when the synthesis process is done is shown on the table below.

	CLB LUTs	CLB Reg	CARRY8	F7MUXES	Block Ram	Freq.Max(MHz)	Latency
System	6872	8580	36	7	9	300	31

Table 5.11: Utilization of System Post-Synthesis stage.

After the placement and the routing table 5.12 shows the results in terms of area of the final system including the clefia core, the DMA core, the axi stream interface, axi interconnect and axi smart connect in the post-implementation stage.

Resource	Utilization	Available	Utilization%
LUT	6080	117120	5.19
LUTRAM	834	57600	1.45
FF	8002	234240	3.42
BRAM	9	144	6.25
BUFG	1	352	0.28

Table 5.12: System Clefia Utilization Post-Implementation stage.

Figure 5.16 shows the Clefia algorithm mapped on FPGA.

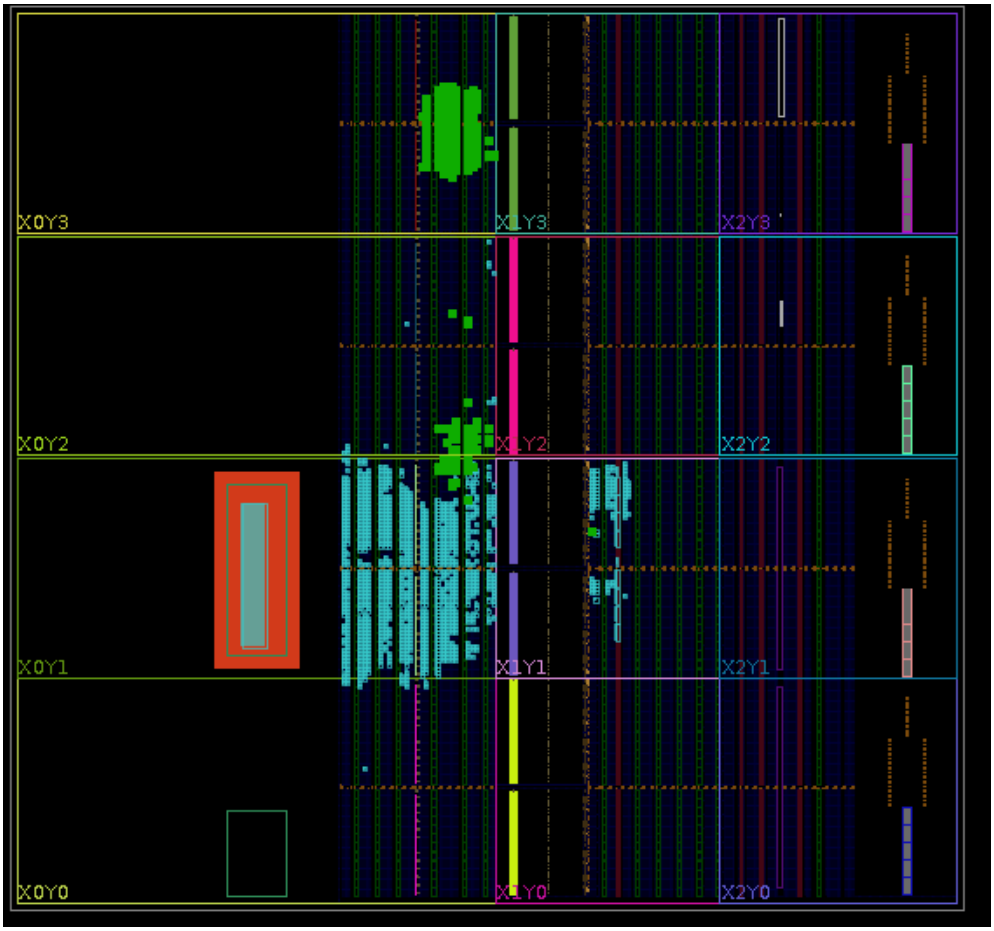


Figure 5.16: Clefia128/128 mapped on FPGA.

The On-Chip power consumption in the Static and Dynamic scenarios of the algorithm Clefia is displayed in the next Figure

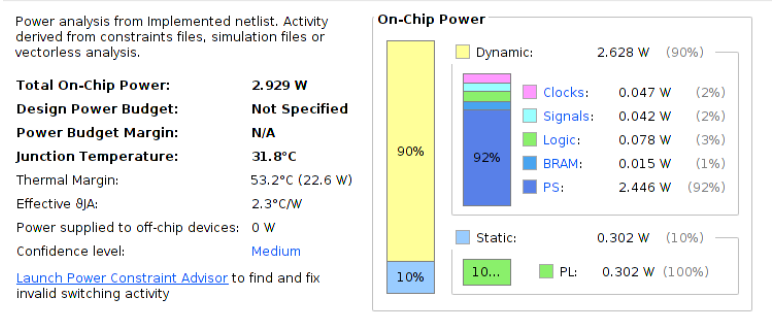


Figure 5.17: Power on chip of clefia128/128 on the fpga.

Figure 5.17 shows the power consumption of clefia algorithm on different parts of the FPGA. It is observed that in Static state there is almost 0.302 W, in the other hand at Dynamic state the power is 2.628 W.

5.2.8 Clefia Software and Hardware comparison

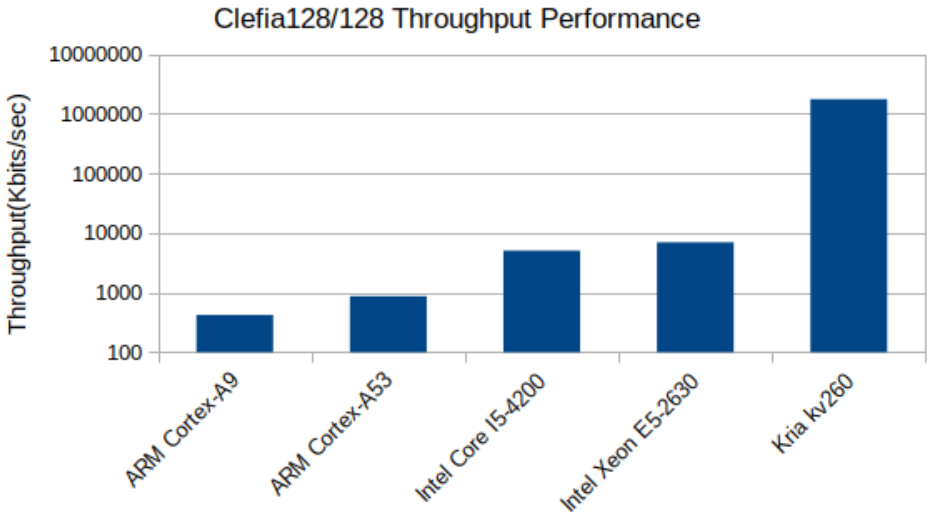


Figure 5.18: Clefia128/128 Software Performance.

Figure 5.18 shows the throughput measurements for the software implementation of the Present block cipher in the four available platforms as well as in the proposed hardware-accelerated implementation on the Kria KV260 system. A log scale is used for the Y axis of the graph. The best software result was achieved when the algorithm was executed on the server kronos(Intel Xeon E5-2630) as expected. This approach gives an average throughput value of 6939Kbits/sec. On the other hand the zedboard platform (ARM Cortex-A9) which perform an average throughput of 419Kbits/sec shows the lowest performance. The hardware implementation on Kria KV260 FPGA achieves the best performance compared to all software implementations with an average throughput value of 1742Mbits/sec. An average throughput Speedup of 252.46x is obtained when comparing the implementation on Kria with the one on CPU Intel Xeon E5-2630.

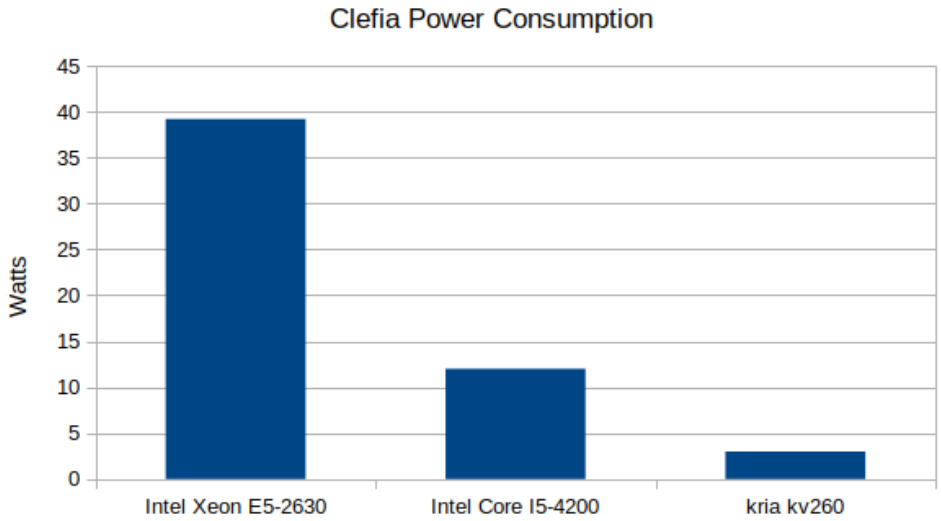


Figure 5.19: Comparison of power consumption of Clefia algorithm.

Making a comparison between the hardware and the software implementations of the algorithm regarding to the power consumption as we can see on figure 5.19 we notice that the hardware implementation of Clefia algorithm require 2.991W on chip power while the software implementations on Intel Core I5-4200 and Intel Xeon E5-2630 require 12W and 39.2W respectively.

Platform	**Performance (1/Exec Time)**	**Power Consumption (Watts)**	**Energy efficiency (Performance per Watts)**
Kria KV260 (Software only)	0.0176	2.432	0,0150
ZedBoard (Software only)	0.036	1.556	0,0113
Generic Laptop	0.2127	12	0,0177
Server IntelXeon E5-2630 (kronos.mhl.tuc.gr)	0.5617	39.2	0,0142
Kria KV260 (Hardware)	187.76	2.991	62,776

Table 5.13: Comparison between different platforms of Clefia algorithm

About the energy efficiency and performance as we can see in table 5.13 the implementation of the algorithm on hardware in both cases shows better results. After the calculation of performance per watt ratio we observe that FPGA implementation is

3547x more energy efficient from CPU implementation on generic laptop. The results of the energy efficiency are also depicted at the following graph. A log scale is used for the Y axis of the graph.

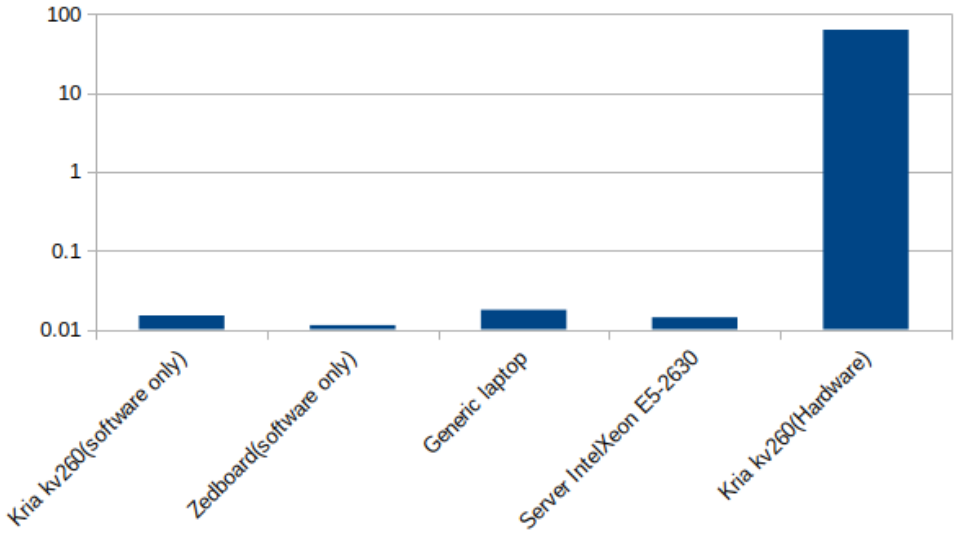


Figure 5.20: Energy efficiency of Clefia algorithm.

5.2.9 Related Work

In this subsection, the proposed implementations of this thesis are going to be compared with related works reported in the literature. The focus is on implementations that provide hardware accelerated versions of the three encryption algorithms (Simon, Clefia and Present) on FPGAs. Implementations of these algorithms on ASICs (Application Specific Integrated Circuits) will also be reported, as they are of interest from an architectural perspective, however direct comparisons in terms of performance and energy efficiency results are not considered. ASICs can provide the maximum degrees of performance and energy efficiency among all solutions but since they cannot be reprogrammed and modified post implementation, they cannot meet the flexibility and adaptability requirements considered for the targeted applications. Several of the implementations in the literature use pipeline architecture as well as unroll architecture.

Unrolled structures frequently have higher throughput, but they can have very high requirements and little flexibility. Additionally, because each unrolled structure can only target a specific key size, the occupancy overhead in systems that target many key sizes is increased. With the pipeline architecture the performance in terms of throughput is better but consumes more area for its implementation.

Also several FPGA-based implementations of the Simon encryption algorithm have been presented. Wetzel and Bokslag [50] describe various hardware architecture designs for Simon with 64-bit plaintext and 128-bit key on Xilinx Spartan-6 FPGAs. Their highest performance designs use a mixed inner/outer round pipeline architecture. To increase the throughput they employ pipelining within each cipher round and also the round function is divided into n independent sub-functions. The performance of this implementation provides a 33Gbit/sec throughput occupying 1149 slices, 2752 slice registers and 4096 slice LUTs. Lower performance versions are reported that employ an iterative architecture instead of an unrolled one, that present far lower area requirements and also significantly lower throughput (269Mbps).

Our implementation of the Simon cipher (actually all three ciphers) follows the iterative design paradigm. Highly pipelined and unrolled architectures provide maximum throughput at the expense of significantly higher resource usage (more than 10x the resources), reduced flexibility and increased latency. Our effort has been to keep the resource usage levels low (thus enabling us to use the available FPGA fabric to host multiple encryption functions at the same time or employ entirely different accelerators that may be required in an IoT edge node), retain maximum flexibility (an iterative architecture easily adapts to different key lengths that require different number of rounds) and minimum latency (satisfying IoT application requirements and the being more suitable to support encryption modes that depend on the outcomes of previously encrypted blocks). Therefore, compared to the iterative designs in [50], our work has comparable requirements (222 slices vs 105 to 148 slices), but presents significantly higher performance (1057Mbps vs 206-269Mbps).

The bit-serial data-path used by the systems described in [51] and [52] within 3 different implementations of Simon algorithm on Xilinx Spartan-3 and Xilinx Spartan-6. Our work can be compared better with the one on Spartan-6. Their implementation provides a throughput of 392.4 Mbps while use 766 slices with 135 Max frequency. Comparing these results with our work we can observe that our implementation needs less slices (222 than 766) and achieve better throughput(1057Mbps instead of 392.4Mbps). On both the Virtex-7 and Zynq-7000, the fault attack resistant architectures proposed in [53] occupy 73 slices. These architectures are more space-efficient than our Simon implementation, but at the cost of a throughput our design have better results.

A Present implementation measuring efficiency as a trade-off between performance

and area was published by Sbeiti et al [54] in 2009. Their findings showed that Present was a good cipher for both high performance applications and those requiring inexpensive hardware. The cost of implementing that task using LUT-4 technology on a Spartan-III FPGA was 202 slices with max frequency of 254MHz and a throughput of 508Mbps. The authors design the algorithm in order to achieve a minimal hardware footprint targeted for low cost devices such as RFIDs. Comparing these results with our work we achieve more throughput (2107Mbps instead of 508Mbps) but our implementation needs more area (353 slices instead of 202) with higher frequency. The following work [55] proposed resource-efficient and high-performance architecture for Present block cipher. The implementation of this work have been done on LUT-6 technology based on Xilinx Virtex-5 XC5VFX70T-1-FF1136 FPGA device. This architectures have a maximum clock frequency of 306.84 MHz, a delay of 33 clock cycles, and a throughput of 595.08 Mbps. The required S-box(S) is realized by an area-optimized combinational logic datapath. In comparison with this thesis work were have 350 MHz maximum clock frequency and we achieve 3,5x times better throughput performance. Another work proposed on [56]. The author of this work Bahram Rashidi describes the hardware implementation of Present block cipher based on Virtex-5 XC5VLX50 and Spartan-3 XC3S200 FPGAs. The purpose of this work is to achieve high-throughput. To reduce the latency and increase throughput, the loop unrolling technique is applied in the structures. Multiple implementations with different unroll factors have been made on this work. On Spartan-3 FPGA with the unroll factor = 4 achieve a throughput performance of 982Mbps with 576 slices while with same unroll factor on Virtex-5 the throughput performance that he achieve is 2359Mbps using 257 slices. Our implementation without using loop unroll technique needs more slices (353 vs 257) with a throughput of 2107Mbps.

A pipelined unrolled architecture with three separate key expansion units for all potential key sizes made up the first FPGA-based Clefia implementation that was described in [57]. The authors of this work want to show that with a small area cost and with no performance impact, full key expansion can be supported. This is achieved by using addressable shift registers, allowing to compute the 4 and 8 branch CLEFIA Feistel network within the same structure. Their implementation needs 200 slices with 375 max frequency and achieve 1.3 Gbps throughput. These results are not very different from the results of our implementation. We achieve better throughput performance

(1,7Gbps instead of 1,3Gbps) with bigger hardware footprint (250 slices instead of 200). Also a pipeline implementation of Clefia block cipher is described in [58]. The FPGA for this work was a Alpha-Data ADM-XP platform. The authors of this implementation achieve a throughput performance of 2,4 GB/s with 200 MHz clock frequency. In comparison with our implementation were we achieve a throughout of 1,7 GB/s at 350 MHz clock frequency without a pipeline architecture. Another compact implementation of clefia is proposed in [59]. This work achieves a throughput above 1Gbit/s with a resource usage as low as 86 LUTs and 3 BRAMs on a VIRTEX 5 FPGA.

At table 5.14 displays information on the performance of the algorithms in this work as well as others existing implementations from the literature.

Algorithm	**Published**	**Board**	**Slices**	**Max.Freq (MHz)**	**Throughput (Mbps)**
Clefia	**-**	**Xilinx Kria kv260**	**250**	**350**	**1742**
Clefia	Proenca [59]	xc5vlx30	170	240	1700
Clefia	Kryjak [58]	Alpha-Data ADM-XP		200	2400
Clefia	Chaves [57]	xc5vlx30	200	375	1333
Present	**-**	**Xilinx Kria kv260**	**353**	**350**	**2107**
Present	Bahram Rashidi [56]	xc5vlx50	282	341	2359
Present	Pandey [55]	Virtex 5	-	306.84	595.0
Present	Sbeiti et. al [54]	Spartan-III	202	254	508
Simon	**-**	**Xilinx Kria kv260**	**222**	**50**	**1057**
Simon	Josh Wetzels Wouter Bokslag [50]	Spartan 6	113	-	269
Simon	PRASHANT AHIR [53]	Virtex 7	73	-	302
Simon	Aydin Aysu [51]	Spartan 6	766	135	392

Table 5.14: Compare implementations of PRESENT, CLEFIA and SIMON block ciphers.

In the following, implementations of the algorithms from the literature on ASIC technology are presented.

ASIC flow hardware implementation was the subject of several investigations. Different ASIC implementations for Simon including bit-serial, iterated, and partially and fully pipelined, were shown by Beaulieu et al [29]. The findings show that, in com-

parison to other ciphers, Simon had the highest efficiency. Simon implementations on ASIC hardware and 8-bit microcontroller software platforms were covered by Beaulieu et al [29]. According to the ASIC results Simon64/96 obtain higher throughput with less space(838 and 984 GE. respectively) but Present-80 produced a throughput of 12.4 Kbps at 100 kHz inside 1030 GE.

One of the first Clefia implementations, targeting ASIC technology, was proposed in [60]. It included Clefia encryption/decryption and a 128-bit key expansion but did not support 192 and 256-bit keys. Two hardware structures are suggested by the authors, and they are examined from the prospective of non-linear F function implementation. This structure achieves the best throughput/area efficiency measure. This implementation achieves on a ASIC 90 nm device 21.07 Slices with 746.24MHz max. Frequency and 5306Mbps throughput. Another pipeline implementations of clefia algorithm is shown in [58], this work is able to operate at a lower frequency, it still manages to attain a throughput of 21.376 Gbps with a pipeline implementation of the algorithm. This is done having a cost of 2479 slices.

An ASIC implementation of Present block cipher is reported in [61] this implementation was performed using Cadence software with 45nm technology. The total area is 10.99 mm^2 and total power is 2.26 mW. Another ASIC implementations of Present algorithm can be found at [62] . This architecture is based on 8-bit datapath consisting a 64-bit register for the encryption process and an 80-bit register for the key scheduling. This work is done in SCL 180 nm technology. Area of the chip layout is 1.55 mm^2 , with 1608 gate equivalent (GE). At 100 MHz operating frequency, total power consumption of the chip is 0.228 mW. A throughput of 130.612 Mbps, energy 112.15 nJ, energy/bit 14.018 nJ/bit, and 0.813 efficiency is obtained.

Chapter 6

Conclusions

In this thesis, three lightweight encryption algorithms were examined (Clefia, Simon and Present), focusing on their deployment on devices that are placed at the edge of the IoT infrastructure. Software implementations of these algorithms on a variety of CPU-based systems were tested covering a range of simple 32-bit ARM processors up to high-performance x86-64 server-grade ones. Against those platforms, FPGA-accelerated versions of the encryption processes running on simple Zynq US+ devices (Kria) were proposed as low-cost, high-performance and energy efficient alternatives.

Our results in terms of performance demonstrate that the proposed architectures can provide significant speedups compared to the best software platform (Intel Xeon-based server) ranging from 30x up to 1400x depending on the algorithm. This is achieved by devices that have the form factor and cost of the simplest software platforms (against which the performance difference is even more pronounced). A similar trend is observed when energy/power efficiency metrics are considered. The proposed hardware accelerated architectures typically consume as little power as the simple ARM Cortex A9 processor in the Zedboard, but since they provide orders of magnitude higher performance, the performance per watt difference is again up to three orders of magnitude higher.

Comparing the proposed architectures with other related works reported in the literature, it is shown that the performance achieved is lower than fully pipelined / completely unrolled architectures. This is to be expected and it is by design, since our stated goal is to provide area efficiency (so that multiple algorithms may be implemented on the available FPGA resources) and minimize latency issues as our target devices aim to serve multiple connected clients with stricter latency than throughput requirements. Compared to reported works that follow the same architectural princi-

ples and design goals, the proposed architectures typically perform better. It should be noted that the proposed architectures also include software components for greater flexibility and programmability compared to the related FPGA implementations reported in the literature, thus making our systems more easy to deploy in the field in IoT applications.

Bibliography

[1] "The Internet of Things kernel description," https://www.projectcubicle.com/internet-of-things-applications-in-industry/.

[2] I. Dutta, B. Ghosh, and M. Bayoumi, "Lightweight cryptography for internet of insecure things: A survey," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Aug. 2019, p. 0475–0481.

[3] "Wikipedia kernel description," https://en.wikipedia.org/wiki/Cryptographic_hash_function.

[4] M. Rana, Q. Mamun, and R. Islam, "Current lightweight cryptography protocols in smart city iot networks: A survey," 10 2020.

[5] "FPGA Architecture kernel description," https://coqube.com/fpga-design-flow/.

[6] V. Rao and K. V. Prema, "A review on lightweight cryptography for internet-of-things based applications," *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, p. 8835–8857.

[7] B. Chaitra, V. K. Kumar, and C. R. Shantharama, "A survey on various lightweight cryptographic algorithms on fpga," *IOSR Journal of Electronics and Communication Engineering (IOSR-JECE)*, vol. 12, p. 54–59, 2017.

[8] V. K. Prasanna, A. Dandalis, and D. S. J. D. Prasanna, "Fpga-based cryptography for internet security ," 2000.

[9] H. Delfs and H. Knebl, *Introduction to cryptography: principles and applications*, 2nd ed. *Springer Berlin Heidelberg*, Dec. 2007, ch. 2, p. 11–48.

[10] C. Lara-Nino, A. Díaz-Pérez, and M. Morales-Sandoval, "Elliptic curve lightweight cryptography: A survey," *IEEE Access*, vol. 4, p. 1–37, 2018.

[11] S. Chandra, S. Paira, S. Alam, and S. Bhattacharyya, "A comparative survey of symmetric and asymmetric key cryptography," in *IEEE 2014 International Conference on Electronics,Communication and Computational Engineering (ICECCE)*, Nov. 2014, p. 83–93.

[12] O. P. Piñol, S. Raza, J. Eriksson, and T. Voigt, "Bsd-based elliptic curve cryptography for the open internet of things," in *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, Jul. 2015, p. 1–5.

[13] I. Chatzigiannakis, A. Pyrgelis, P. G. Spirakis, and Y. C. Stamatiou, "Elliptic curve based zero knowledge proofs and their applicability on resource constrained devices," in *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, Nov. 2011, p. 715–720.

[14] T. Backenstrass, M. Blot, S. Pontie, and R. Leveugle, "Protection of ecc computations against side-channel attacks for lightweight implementations," in *2016 1st IEEE International Verification and Security Workshop (IVSW)*, Jul. 2016, p. 1–6.

[15] T. K. Goyal and V. Sahula, "Lightweight security algorithm for low power iot devices," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2016, p. 1725–1729.

[16] S. Al-Kuwari, J. Davenport, and R. Bradford, "Cryptographic hash functions: recent design trends and security notions," in *Short Paper Proceedings of 6th China International Conference on Information Security and Cryptology (Inscrypt '10)*, Jan. 2010, p. 133–150.

[17] S. Al-Kuwari, J. H. Davenport, and R. J. Bradford, "Cryptographic hash functions: recent design trends and security notions," in *The 6th China International Conference on Information Security and Cryptology*, Jan. 2010, p. 133–150.

[18] V. Thakor, M. A. Razzaque, and M. Khandaker, "Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities," *IEEE Access*, vol. 9, p. 28 177–28 193, 2021.

[19] M. Pourghasem, E. Sheikhloo, and R. Ebrahimi Atani, "Light weight implementation of stream ciphers for m-commerce applications," *arXiv*, 2014.

[20] K. Shankar and M. Elhoseny, *Secure image transmission in wireless sensor network (WSN) applications*, 1st ed. *Springer Cham*, 2019, ch. 1, p. 7–12.

[21] C. Thorat and V. Inamdar, *Applied Computing and Informatics*, 2018, ch. 10, p. 1–6.

[22] G. Hatzivasilis, K. Fysarakis, I. Papaefstathiou, and H. Manifavas, "A review of lightweight block ciphers," *Journal of Cryptographic Engineering*, vol. 8, p. 1–44, 2017.

[23] T. Suzaki and K. Minematsu, *Fast Software Encryption. Springer Berlin Heidelberg*, 2010, ch. 1, p. 19–39.

[24] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, "Lightweight cryptography for iot: A state-of-the-art," *IEEE Acess*, vol. arXiv:2006, p. 1–19, 2020.

[25] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, vol. 4727. *Springer Berlin Heidelberg*, 2007, p. 450–466.

[26] R. Chatterjee and R. Chakraborty, "A modified lightweight present cipher for iot security," in *2020 International Conference on Computer Science, Engineering and Applications (ICCSEA)*, 2020, p. 1–6.

[27] S. Feizi, A. Ahmadi, and A. Nemati, "A hardware implementation of simon cryptography algorithm," in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, 2014, p. 245–250.

[28] J. Wetzels and W. Bokslag, "Simple simon: Fpga implementations of the simon 64/128 block cipher," *ArXiv*, vol. abs/1507.06368, 2016.

[29] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "Implementation and performance of the simon and speck lightweight block ciphers on asics," *Mathematics, Computer Science*, 2016.

[30] H. AlKhzaimi and M. M. Lauridsen, "Cryptanalysis of the simon family of block ciphers," *IACR Cryptol. ePrint Arch.*, vol. 2013, p. 543, 2013.

[31] A. Singh, N. Chawla, M. Kar, and S. Mukhopadhyay, "Energy efficient and side-channel secure hardware architecture for lightweight cipher simon," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 4 2018, p. 159–162.

[32] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "Simon and speck: Block ciphers for the internet of things," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 585, 2015.

[33] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit block-cipher clefia (extended abstract)," vol. 4593, p. 181–195, 2007.

[34] B. Sun, R. Li, M. Wang, P. Li, and C. Li, "Impossible differential cryptanalysis of clefia." *IACR Cryptology ePrint Archive*, vol. 2008, p. 151, 01 2008.

[35] J. Takahashi and T. Fukunaga, "Improved differential fault analysis on clefia," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, Aug. 2008, p. 25–34.

[36] S. Rekha and S. Paramasivam, *Threshold Implementation of a Low-Cost CLEFIA-128 Cipher for Power Analysis Attack Resistance*, Aug. 2019, p. 272–285.

[37] M. Katagi, "The 128-Bit Blockcipher CLEFIA," p. 33, 2011.

[38] S. Churiwala, *Designing with xilinx® FPGAs: using vivado*, 1st ed. *Springer Cham*, Jan. 2017, ch. 1, p. 5–7.

[39] Xilinx, *LogiCORE IP DMA Product Guide (AXI)*.

[40] "Zedboard — avnet boards = https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/zedboard-board-family/!ut/p/z1/vvrdu6mwfp0r-objjif8hcdqkrzlp0xbvoykknqsbsrn4tzfv7g66zhty-5olwdi7px77j1nmamznefw8kbecsv1fc7cb5hv9mxobyvrehtopkkp2_chlhlhelsh9kkfdincfdtrw4ggyqvxr2nexy5kf9oppngoauu_gqyyrfqbtyjz3prcgvlvkleqyy0xna9352hfxlwfoeebusp_zgr7wgslite5vuwimv7oxr1uw2nbfbr6jpj98e300gcwvjdr3fpotszzpzgqf_hlanasfgdjzabcmqxwougm6xpl4-jv6h8twzjfaowpadqku72oz7vbxfaqy2a_bwzdiedumt5qmglwz_onogb3-y0olhku0mvwrmv6h-qwty5gikeyllvd6d9s-o8udlhbboete1rovvpsj4boawe_jtotqas2_pzwwqipxoqr_kjj735mod8b10knutdsuvkcwywrodmm3rzowxnqb-0m4tellnsfnuzaabz9sanr2aec!/dz/d5/l2dbisevz0fbis9nqseh/?urile=wcm%3apath%3a%2favnet%2bcontent%2blibrary%2favnethome%2fproducts%2favnet-boards%2fdev%2bboards%2bkits%2bsoms%2fzedboard%2fzedboard-board-family,."

[41] "Kria = https://www.xilinx.com/products/som/kria/kv260-vision-starter-kit.html,."

[42] "Intel core i5-4200u processor = https://ark.intel.com/content/www/us/en/ark/products/75459/intel-core-i54200u-processor-3m-cache-up-to-2-60-ghz.html,."

[43] "Intel xeon processor e5-2630 v4 = https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e52630-v4-25m-cache-2-20-ghz.html,."

[44] "Poweredge r530 rack server = https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e52630-v4-25m-cache-2-20-ghz.html,."

[45] "Sony corporation clefia software code = https://www.dell.com/en-uk/shop/productdetailstxn/poweredge-r530,."

[46] "Present implementation in c code = http://www.lightweightcrypto.org/implementations.php,."

[47] "Simon software implementation cryptolibrary = https://www.cryptopp.com/docs/ref/simon_8h.html#details, ."

[48] "Powerstat = https://www.hecticgeek.com/powerstat-power-calculator-ubuntu-linux/."

[49] "Vmstat tool = https://man7.org/linux/man-pages/man8/vmstat.8.html."

[50] J. Wetzels and W. Bokslag, "Simple simon: Fpga implementations of the simon 64/128 block cipher," *Cryptology ePrint Archive, Paper 2016/029*, 2016.

[51] A. Aysu, E. Gulcan, and P. Schaumont, "Simon says: Break area records of block ciphers on fpgas." *IEEE Embedded Systems Letters*, vol. 6, p. 37–40, 2014.

[52] E. Gulcan, A. Aysu, and P. Schaumont, "A flexible and compact hardware architecture for the simon block cipher," *International Workshop on Lightweight Cryptography for Security and Privacy*, vol. 6, p. 34–50, 2014.

[53] P. Ahir, M. Mozafari-Kermani, and R. Azarderakhsh, "Lightweight architectures for reliable and fault detection simon and speck cryptographic algorithms on fpga." *ACM Transactions on Embedded Computing Systems*, vol. 16, p. 1–17, 2017.

[54] M. Sbeiti, M. Silbermann, A. Poschmann, and C. Paar, "Design space exploration of present implementation for fpgas," in *5th Southern Conference on Programmable Logic*, April 2009.

[55] J. G. Pandey, T. Goel, and A. Karmakar, "An efficient vlsi architecture for present block cipher and its fpga implementation," in *VLSI Design and Test*, B. K. Kaushik, S. Dasgupta, and V. Singh, Eds. Singapore: Springer Singapore, 2017, p. 270–278.

[56] B. Rashidi, "High-throughput and lightweight hardware structures of hight and present block ciphers," *Microelectronics Journal*, 2019.

[57] J. C. Bittencout, J. Resende, W. Oliveira, and R. Chaves, "Clefia implementation with full key expansion," Aug. 2015.

[58] T. Kryjak and M. Gorgon, "Pipeline implementation of the 128-bit block cipher clefia in fpga," in *Field Programmable Logic and Application(FPL)International Conference on. IEEE*, 2009, p. 373–378.

[59] P. Proenca and R. Chaves, "Compact clefia implementation on fpgas," in *21st International Conference on Field Programmable Logic and Applications*, Sep. 2011, p. 512–517.

[60] T. Sugaware, N. Homma, T. Aoki, and A. Satoh, "Ieee international symposium on circuits and systems," in *5th Southern Conference on Programmable Logic*, 2008, p. 2925–2928.

[61] S. Anitha, Kumari and M. Mahalinga, "Asic implementation of present cipher for iot application," *Journal of VLSI Design and Signal Processing*, vol. 5, p. 12–18, 2019.

[62] J. G. Pandey, T. Goel, M. Nayak, C. Mitharwal, S. Khan, S. K. Vishvakarma, A. Karmakar, and R. Singh, "A vlsi architecture for the present block cipher with fpga and asic implementations," in *VLSI Design and Test*, S. Rajaram, N. Balamurugan, D. Gracia Nirmala Rani, and V. Singh, Eds. Singapore: Springer Singapore, 2019, p. 210–220.

Appendices

Appendix A

Appendix

A.1 Present Algorithm runs on Software

Present64/80 ARM Cortex-A9	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2.25mb	1646612621	24.7	666644785.91	91.09
4.5mb	32929075436	49.39	666715437.05	91.11
9mb	65867234274	98.8	666672411.68	91.09
18mb	131717354252	197.8	666653275.89	91.10
36mb	263496787042	395.25	666658537.74	91.08
90mb	658659349614	987.99	666666008.37	91.09

Table A.1: Present64/80 runs on Zedboard

Present64/80 ARM Cortex-A53	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2.25mb	2016778564	10	201677856.4	225
4.5mb	4033578468	20	201678923.4	225
9mb	8067178524	40	201679463.1	225
18mb	16134378486	80	201679730.07	225
36mb	32268778510	161	200427195.71	223.60
90mb	80671978652	403	200178607.07	223.35

Table A.2: Present64/80 runs on Kria KV260

Present64/80 Intel(R) Core(TM) I5-4200U	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2.25mb	3650338	1.82	2000000	1232.76
4.5mb	7781318	3.89	2000000	1156.61
9mb	15177084	7.58	2000000	1185.99
18mb	29906160	14.95	2000000	1203.76
36mb	61171204	30.58	2000000	1177.02
90mb	150905448	75.45	2000000	1192.79

Table A.3: Present 64/80 runs on Intel(R) Core(TM) I5-4200U

Present64/80 Intel(R) Xeon(R) E5-2630	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2.25mb	2980000	1.57	1898089.17	1433.12
4.5mb	7660000	3.13	2447284.34	1437.69
9mb	12520000	6.36	1968553.45	1415.09
18mb	23360000	12.68	1842271.29	1419.55
36mb	46720000	25.36	1842271.31	1419.62
90mb	118340000	63	1878412.69	1428.57

Table A.4: Present64/80 runs on Intel(R) Xeon(R) E5-2630

A.2 Clefia Algorithm runs on Software

Clefia128/128 ARM Cortex-A9	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2mb	3161389114	4.74	666959728.69	421.89
4mb	6404764954	9.61	666468777.73	416.23
8mb	12638951964	18.96	666611390.50	421.94
16mb	25499076418	38.25	666642520.73	418.30
32mb	51073576630	76.61	666669842.44	417.70
80mb	128051326706	192.08	666656219.83	416.49

Table A.5: Clefia128/128 runs on Zedboard

Clefia128/128 ARM Cortex-A53	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2mb	461026166	2.31	199578426.83	865.80
4mb	922055406	4.61	200012018.65	867.67
8mb	1844092842	9.22	200010069.63	867.67
16mb	3688181816	18.44	200009859.86	867.67
32mb	7376328934	36.88	200008919.03	867.67
80mb	18453210158	92.27	199991439.88	867.02

Table A.6: Clefia128/128 runs on Kria KV260

Clefia128/128 Intel(R) Core(TM) I5-4200U	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2mb	779280	0.39	2000000	5128.20
4mb	1570306	0.78	2000000	5094.54
8mb	3168166	1.58	2000000	5050.24
16mb	6838040	3.18	2134872.71	5031.44
32mb	12740248	6.37	2000000	5023.45
80mb	30817942	15.90	2000000	5031.44

Table A.7: Clefia128/128 runs on Intel(R) Core(TM) I5-4200U

Clefia128/128 Intel(R) Xeon(R) E5-2630	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
2mb	560000	0.28	1962846.12	7010.164
4mb	1140000	0.57	2000000	7017.54
8mb	2280000	1.14	2000000	7017.54
16mb	4640000	2.32	2000000	6896.55
32mb	9260000	4.63	2000000	6911.44
80mb	22920000	11.46	2000000	6980.80

Table A.8: Clefia128/128 runs on Intel(R) Xeon(R) E5-2630

A.3 Simon Algorithm runs on Software

Simon64/128 ARM Cortex-A53	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
40mb	22064811274	33.1	666610612.50	1208.45
100mb	55154812932	82.73	666684551.33	1208.75
500mb	275754810880	413.63	666670238.81	1208.80
1000mb	551504812892	827.26	666664425.80	1208.80

Table A.9: Sion64/128 runs on Zedboard

Simon64/128 ARM Cortex-A53	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
40mb	1670574850	8.35	200068844.31	4790.41
100mb	4174212370	20.87	200010175.85	4791.56
500mb	20865120416	104.33	199991569.21	4792.48
1000mb	41728757500	208.65	199994045.05	4792.71

Table A.10: Simon64/128 runs on Kria KV260

Simon64/128 Intel(R) Core(TM) I5-4200U	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
40mb	4032212	2.01	2000000	19840.22
100mb	9895050	4.94	2000000	20212.12
500mb	48421678	24.21	2000000	20651.90
1000mb	98721878	49.36	2000000	20258.933

Table A.11: Simon64/128 runs on Intel(R) Core(TM) I5-4200U

Simon64/128 Intel(R) Xeon(R) E5-2630	**clock cycles**	**seconds**	**cycles/sec**	**kbits/sec**
40mb	2200000	1.1	2000000	36363.63
100mb	5540000	2.77	2000000	36101.08
500mb	27700000	13.85	2000000	36101.08
1000mb	55820000	27.91	2000000	35829.45

Table A.12: Simon64/128 runs on Intel(R) Xeon(R) E5-2630