Abstract

The internet is an open and public system where information is send and received over shared wires and connections. Though we still exchange a lot of private data, things like credit card numbers, bank information, passwords, and emails. Especially during this pandemic, data exchange increased drastically due to the radical shift in the ways people communicate and engage with one another. Schools adopted online education, whereas employees are working from home. Virtual classrooms, webinars, and online meetings on video chat applications like zoom, google, and facebook have become the new normal of today's world. Protecting and securing this tremendous amount of data in transit requires strong cryptographic algorithms. One of the most popular and widely used data protection schemes nowadays across the globe is The Advanced Encryption Algorithm (AES). However, security alone does not fulfil the requirements of modern systems, where speed is considered being an integral part of every design.

The aim of our project is not to implement the AES-128 algorithm in VHDL but to maximize its performance while using it in a video recording encryption/decryption application within the constrained embedded hardware of the Terasic DE10 Standard FPGA Board. Therefore, we have started our experiment by a pure software implementation of the AES-128 in C language, and then we created multiple design versions of the hybrid system that make use of the SoC HPS and FPGA. Finally, we implemented the fastest designs in a video encryption/decryption application. The results showed that the hybrid system has the advantage of faster execution by a factor of over 4 times when compared to the pure C implementation, which makes it a favorable choice in this type of applications on this platform.

Acknowledgements

In the name of Allah, the Most Gracious and the Most Merciful.

All praises to Allah and His blessing for the successful completion of our project.

First and foremost, we would like to sincerely thank our supervisor **Dr. Ahmed MAACHE** for his guidance, understanding, patience and most importantly, he has provided us with the necessary equipment used in our experiment. It has been a great pleasure and honor to have him as our supervisor.

Our deepest gratitude goes to our families for their endless love, unconditional support, and encouragement throughout our lives. We also want to extend our thanks to our beloved friends who accompanied us throughout this journey.

Last but not least, we would like to express our gratitude to all IGEE teachers and staff for their kindness and great help during our studies.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Algorithm
AXI	Advanced eXtensible Interface
DC	Direct Current
DDR3	Double Data Rate 3
DES	Data Encryption Standard
DMA	Direct Memory Access
EDS	Embedded Design Suite
FFMPEG	Fast Forward Motion Picture Experts Group
FIPS PUBS	Federal Information Processing Standards Publications
FPGA	Field Programmable Gate Arrays
GCD	Greatest Common Divisor
HPS	Hard Processor System
IO	Input Output
IP	Intellectual Property
JTAG	Joint Test Action Group
MDS	Maximal Distance Separable
MPU	Microprocessor Unit
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PIO	Parallel Input Output
PLL	Phase Locked Loop
RTL	Register-transfer Level
S-Box	Substitution Box
SD Card	Secure Digital Card
SoC	System on a Chip
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver/Transmitter
ULPI	UTMI+LPI, Universal Transceiver Macrocell Interface+ Low Pin Interface
USB	Universal Serial Bus
VHDL	VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language

1 Introduction and Background

1.1. Introduction to cryptography

Cryptography is the practice and study of techniques for secure communication in the presence of third parties called adversaries. The term is derived from Ancient Greek, “kryptós” means hidden or secret whereas “graphein” means study or write.

Since the development of rotor cipher machines in World War I and the advent of computers in World War II, cryptography methods have become increasingly complex and its applications more varied [1].

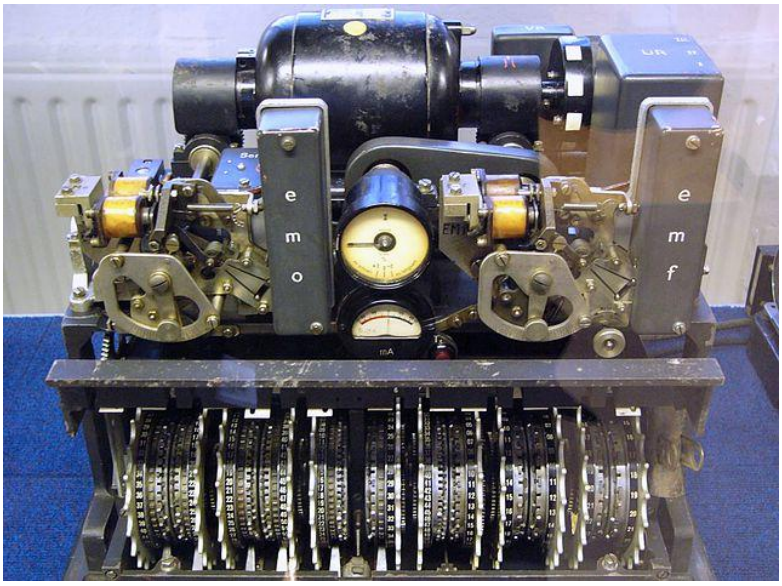


Figure 1- 1. German Lorenz cipher machine, used in World War II **[1]**.

1.1.1. Crypto-terminologies

Cryptography, cryptology, cryptanalysis, three terms often used interchangeably but what is the actual difference between them?

- **Cryptology:** is the most general term that represents the field to which cryptography and cryptanalysis belong.
- **Cryptography**: is the science whose purpose is hiding the meaning of a message in a secret writing context.

- **Cryptanalysis**: is the art and science of breaking cryptosystems, mainly this is needed in order to know whether a crypto system is secure enough or not.

Since in our work we are only interested in cryptosystems that implement the AES algorithm, let's focus on exploring more details about cryptography.

1.1.2. Functions of cryptography

There are five primary functions of cryptography [2]:

- **Privacy/confidentiality:** Ensuring that no one can read the message except the intended receiver.
- **Authentication:** The process of proving one's identity.
- **Integrity:** Assuring the receiver that the received message has not been altered in any way from the original.
- **Non-repudiation:** A mechanism to prove that the sender really sent this message.
- **Key exchange:** The method by which crypto keys are shared between sender and receiver.

1.1.3. Types of cryptographic algorithms

In general, cryptographic algorithms are categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use.

Thus, there are three classifications [2] [3]:

- **Symmetric key/secret key cryptography**: Uses a single common key for encryption and decryption, it is simple and fast. Primarily used for privacy and confidentiality.
- **Asymmetric key/public key cryptography**: Uses one key for encryption, public key that is known by everyone, and another key for decryption, private key that is only known by the intended receiver. Primarily used for authentication, non-repudiation, and key exchange.

- **Hash functions**: Uses no key but a mathematical function to encrypt the message, it is also called one-way encryption since the original text cannot be retrieved from the ciphered text. Primarily used for message integrity.

In practice, most cryptographic protocols will use asymmetric encryption to establish a connection and create a shared secret. Then, they will switch to symmetric encryption to benefit from the speed difference and make them more robust against security attacks. These systems are referred to as **hybrid schemes**.

1.1.4. Requirements of a strong crypto algorithm

According to the famous information theorist *Claude Shannon*, strong encryption algorithms must provide two main primitives [4]:

Confusion: achieved by a series of operations that shadow the relationship between the key used and the cipher text produced.

Diffusion: it is the spread of influence of an individual bit from the plain-text over the entire cipher text.

1.1.5. AES history

Let's start with a bit of background information on encryption standards. The Data Encryption Standard (DES) developed in the mid-1970s by IBM and officially adopted in 1977, was the most dominating symmetric encryption algorithm used in secure data manipulation. However, by late 1990s, DES had become outdated and increasingly susceptible to cyberattacks because of the 56-bit long keys used in it that nowadays became relatively easy to break thanks to the improvement of the hardware and through an exhaustive key search. This incident gave rise to a need in developing a safer and more advanced standard to protect secret data. In 1997, the US National Institute of Standards and Technology (NIST) called for proposals for a new Advanced Encryption Algorithm (AES) in an open selection process of 3 evaluation rounds.

By August 1998, 15 candidates from several countries submitted their work. One year after, this number narrowed down to 5 finalists, and by 2001 NIST declared the block cipher Rijndael as the new AES and published it as a final standard (FIPS PUB 197). It is also remarkable that in 2003 the US National Security Agency (NSA) announced that it allows AES to encrypt classified documents up to the level SECRET for all key lengths, and up to the TOP SECRET level for key lengths of either 192 or 256 bits. Prior to that date, only non-public algorithms had been used for the encryption of classified documents [4].

1.1.6. AES description and design rational

The AES is a symmetric block cipher, this means that the key that is used in the encryption is the same one used also in the decryption being symmetric, and the algorithm operates on a fixed-length group of bits, called block, namely a data block of 128, 192 or 256 bits in the case of AES with a key length of either 128, 192 or 256 bits [5].

By varying the key length from shortest to longest, we increase the complexity of computations hence the strength of the ciphering and the difficulty of breaking our cryptosystem. Though, this comes at the cost of increasing the time needed to encrypt or decrypt data. Until now, there exist no practical way to break it even with key length of 128 bits.

Key size (bits)	Number of alternative keys	Time required at 10^6 decryption/μs
32	$2^{32} = 4.3 \times 10^9$	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	10 hours
128	$2^{128} = 3.4 \times 10^{38}$	5.4×10^{18} years
256	$2^{256} = 3.7 \times 10^{50}$	5.9×10^{30} years

Table 1- 1. Average time required for exhaustive key search **[6]**.

The three criteria taken into account in the design of Rijndael are the following [5]:

- Resistance against all known attacks.
- Speed and code compactness on a wide range of platforms.
- Design simplicity.

1.2. Mathematics of AES

Mathematics is vital in all domains of our life in general, and in creating any crypto algorithm in particular. In order to fully understand how AES works, one must check the mathematics behind it.

1.2.1. Some needed perquisites

- A **group G** is a set of elements $G = \{a, b, c, ...\}$ and an operation $\oplus$ for which the following axioms hold [4]:
  - Closure: for any $a \in G$, $b \in G$, the element $a \oplus b \in G$.
  - Associative law: for any $a, b, c \in G$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
  - Identity: there is an identity element 0 in G for which $a \oplus 0 = 0 \oplus a = a$ for all $a \in G$.
  - Inverse: for each $a \in G$, there is an inverse $(-a)$ such that $a \oplus (-a) = 0$.
- A **field F** is a set of elements where [4]:
  - All elements of **F** form an additive group with the group operation "+" and the neutral element 0.
  - All elements of **F** except 0 form a multiplicative group with the group operation "×" and the neutral element 1.
  - When the two group operations are mixed, the distributive law holds, i.e., for all $a, b, c \in F$: $a(b + c) = (ab) + (ac)$.

Informally, a field is a set of elements where we can add, subtract, multiply and invert.

1.2.2. Finite fields

A finite field, also known as a ***Galois field***, named after the French mathematician *Évariste Galois*, is a field with a finite number of elements. This number is called the *order* or *cardinality* [4] . In cryptography, we are mostly interested in this type of fields since it is not possible in practice to deal with infinite quantities.

Theorem 1 *"A field with order m only exists if m is a prime power, i.e.,* $m = p^n$*, for some positive integer n and prime integer p".* [4]

Hence, we represent a Galois field by the notation GF (p^n).

1.2.3. Types of finite fields

- **Prime fields**: this is a special case where n = 1. Elements of the field GF (p) can be represented by integers 0,1,…,p-1.
- **Extension fields**: this is when n > 1. In AES, we are mostly concerned about the finite field whose number of elements is 256; this is $GF(2^8)$.

AES treats every byte of the internal data path as a one element from the $GF(2^8)$ and performs arithmetic to it [4].

1.2.4. Arithmetic of finite fields

In this section, we will cover the details of finite field's arithmetic, which includes the operations: addition, subtraction, multiplication, and inversion.

Prime filed arithmetic

For all $a,b \in GF(p)$ we have [4]:

- $a+b \equiv c \bmod p$
- $a-b \equiv d \bmod p$
- $a*b \equiv e \bmod p$
- The inverse a^{-1} must satisfy $a^{-1}.a \equiv 1 \bmod p$. Where a^{-1} can be computed using the extended Euclidean algorithm that will be seen after.

such that the result of any operation is always in $GF(p)$.

GF(p^m) arithmetic

Elements of GF(p^m) are polynomials over GF(p) [7]. In extension fields $GF(2^m)$ elements are represented as polynomials with coefficients in $GF(2) = \{0,1\}$ and not as integers. The polynomials have a maximum degree of m - 1, and a minimum degree of 0 [4].

In the field $GF(2^8)$, each element $A \in GF(2^8)$ is thus represented as:

$$A(x) = a_7 x^7 + ... + a_1 x + a_0$$

With $a_i \in GF(2) = \{0,1\}$. There are exactly $256 = 2^8$ such polynomials.

It is helpful to observe that every polynomial can be stored as an 8 bit vector in digital format as:

$$A = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$$

- **Addition and subtraction:** simply achieved by performing standard polynomial addition and subtraction; we add or subtract coefficients with equal powers of x then we apply modulo 2. Because of the latter, addition and subtraction are the same operation [4].

  Example: the addition of $x^7 + x^4 + x^3 + x^2 + x + 1$ with $x^4 + x^2 + 1$ results in:

  $(1 \bmod 2)x^7 + ((1+1) \bmod 2)x^4 + (1 \bmod 2)x^3 + ((1+1) \bmod 2)x^2 + (1 \bmod 2)x + ((1+1) \bmod 2)1$

  Since $(1 \bmod 2) = 1$ and $(2 \bmod 2) = 0$, the resulting polynomial is simply $x^7 + x^3 + x$.

  By performing the subtraction of the two previous polynomials, we get:

  $(1 \bmod 2)x^7 + ((1-1) \bmod 2)x^4 + (1 \bmod 2)x^3 + ((1-1) \bmod 2)x^2 + (1 \bmod 2)x + ((1-1) \bmod 2)1$

  Ending up with in the same result: $x^7 + x^3 + x$

  Important Note: using byte representation, the polynomial $x^7 + x^4 + x^3 + x^2 + x + 1$ corresponds to 10011111 and the polynomial $x^4 + x^2 + 1$ corresponds to 00010101, so that the addition 10011111 + 00010101 = 10001010 corresponds to bitwise XOR.

- **Multiplication:** two elements of $GF(2^m)$ (represented by their polynomials) A(x) and $B(x)$ are multiplied using the standard polynomial multiplication rule.

  In general, the product polynomial $C(x)$ will have a degree higher than $m-1$ and has to be reduced to fit in $GF(2^m)$. In a similar approach to prime field multiplication, the product of the multiplication is divided not by a prime integer p where the remainder is taken but by a certain polynomial, called irreducible polynomial, and we consider only the remainder of the division. A Polynomial is irreducible if it accepts no divisors but 1 and itself. Hence, if we call $P(x)$ the irreducible polynomial, we can write [4]:

  $$C(x) \equiv A(x).\mathrm{B}(x) \mod P(x) \qquad (1.1)$$

  Note that $P(x)$ is not unique.

As part of AES specifications, the irreducible polynomial used is:

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

Example: again, multiplying the two previous polynomials we get:

$$x^7.x^4 + x^7.x^2 + x^7.1 + x^4.x^4 + x^4.x^2 + x^4.1 + x^3.x^4 + x^3.x^2 + x^3.1$$
$$+ x^2.x^4 + x^2.x^2 + x^2.1 + x.x^4 + x.x^2 + x.1 + 1.x^4 + 1.x^2 + 1.1$$

Which simplifies to:

$$x^{11} + x^9 + x^7 + x^8 + x^6 + x^4 + x^7 + x^5 + x^3 + x^6 + x^4 + x^2 + x^5 + x^3 + x + x^4 + x^2 + 1$$

And further to:

$$x^{11} + x^9 + x^8 + x^4 + x + 1$$

Now if we consider doing the previous multiplication in $GF(2^8)$,we would need to apply modulo a reducible polynomial to the result of multiplication. By doing the polynomial division, using long division, of $x^{11} + x^9 + x^8 + x^4 + x + 1$ over the irreducible polynomial $P(x) = x^8 + x^4 + x^3 + x + 1$ used in AES, the result is $x^3 + x + 1$ and the remainder is $x^7 + x^6 + x^5 + x^2 + x \in GF(2^8)$ which is the desired result. To sum up what we had:

$$((x^7 + x^4 + x^3 + x^2 + x + 1).(x^4 + x^2 + 1)) \mod (x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + x^5 + x^2 + x$$

- **Inversion:** the inverse $A^{-1}(x)$ of an element $A(x) \in GF(2^m)$ must satisfy $A.A^{-1} \equiv 1 \mod P(x)$ where A^{-1} can also be calculated using the extended Euclidean algorithm, that is by keeping track of the coefficients.

The Extended Euclidean Algorithm

The Euclidean algorithm is a method of computing the GCD (greatest common divisor) of two integers *a* and *b*, denoted by $\gcd(a,b)$,through dividing the divisor by the remainder until the remainder becomes 0. The extended Euclidean algorithm does not only allow us to find the GCD but also write it in terms of β and γ where:

$$a.\beta + b\gamma = \gcd(a,b) \qquad (1.2)$$

The idea is to use the Euclidean algorithm to find the GCD and recursively work our way back to find the coefficients of *a* and *b* [4].

Example: let $a = 81$ and $b = 57$

$$81 = 57 \times 1 + 24$$

$$57 = 24 \times 2 + 9$$

$$24 = 9 \times 2 + 6$$

$$9 = 6 \times 1 + 3$$

$$6 = 3 \times 2 + 0$$

So *gcd*(81, 57) = 3, because 3 is the last non-zero remainder.

Now to get the coefficients β and γ we reverse the steps above and write the equations in the form of:

$$r = a - q \times b \tag{1.3}$$

Where *r* is last non-zero remainder (or the *gcd*), *a*, *q* and *b* are integers.

Applying this to our example, we have:

$$3 = 9 - 6 \times 1 = 9 + 6 \times (-1)$$

We replace 6 by $6 = 24 - 9 \times 2 = 24 + 9 \times (-2)$ that we took from the previous line and we get:

$$3 = 9 + (24 + 9 \times (-2)) \times (-1) = 9 \times 3 + 24 \times (-1)$$

And so on until we reach the first equation:

$$3 = 57 \times 10 + 81 \times (-7)$$

So $\beta = 10$ and $\gamma = -7$.

In our case. We want to use this algorithm to find the inverse $A^{-1}(x) \in GF(2^m)$ of $A(x) \in GF(2^m)$ using the irreducible polynomial $P(x)$. We have:

$$A(x).\beta(x) + P(x).\gamma(x) = \gcd(A(x), P(x)) \tag{1.4}$$

Since $P(x)$ and $A(x)$ are relatively prime, because $P(x)$ is irreducible and has a higher degree than $A(x)$, their *gcd* will be 1, so we can write:

$$A(x).\beta(x) + P(x).\gamma(x) = 1$$

If we reduce both sides to mod $P(x)$ we find:

$$A(x).\beta(x) \mod P(x) = 1 \mod P(x)$$

And we have:

$$A.A^{-1} \equiv 1 \mod P(x)$$

Then we can deduce that:

$$A^{-1}(x) = \beta(x) \mod P(x) \tag{1.5}$$

Example [4]: let $A(x) = x^2$ and $P(x) = x^3 + x + 1$

$$x^3 + x + 1 = [x].x^2 + [x + 1].....(1)$$

$$x^2 = [x].(x + 1) - x = [x].(x + 1) + x .....(2)$$

$$x + 1 = [1].x + 1.....(3)$$

$$x = [x].1 + 0.....(4)$$

The coefficients are computed In $GF(2)$ meaning that addition and subtraction are the same so we can change a negative sign with a positive one.

From (3) we have:

$$1 = (x + 1) - x.1.....(5)$$

And from (2):

$$x = x^2 - [x].(x + 1).....(6)$$

We substitute (6) in (5):

$$1 = (x + 1).(x + 1) + 1.x^2.....(7)$$

From (1):

$$x + 1 = x^3 + x + 1 - [x].x^2.....(8)$$

We substitute (8) in (7):

$$1 = (x^3 + x + 1).(x+1) + x^2.(x^2 + x + 1)$$

Therefore $A^{-1} = x^2 + x + 1$.

1.2.5. Polynomials with coefficients in GF(2^8)

As we will mention later in the next section, our data including the input plain text, the key and the cipher text will be organized as 4 by 4 matrix form during AES encryption and decryption. Because of the latter, sometimes we will refer to individual columns of our matrix as 4-byte vectors. In this case, we will need to use polynomials whose coefficients are not elements of $GF(2)$, but are elements of $GF(2^8)$.Recall that $GF(2^8) = \{0,1,...,255\}$.

When polynomials have coefficients in $GF(2^8)$, we can say that a polynomial of degree 4 corresponds to a vector of 4 bytes. The product of two such polynomials generally cannot be represented by a four-byte vector; in this case, we reduce the result by a polynomial of degree 4 [5].

In Rijndael, the polynomial x^4+1 is used. Hence, for all $a(x),b(x)\in GF(2^8)$, such that:

$$a(x)=a_0+a_1.x+a_2.x^2+a_3.x^3$$

And:

$$b(x)=b_0+b_1.x+b_2.x^2+b_3.x^3$$

The product:

$$c(x)=c_0+c_1.x+c_2.x^2+c_3.x^3+c_4.x^4+c_5.x^5+c_6.x^6$$

Where:

$$c_0=a_0.\mathrm{b}_0$$

$$c_1=a_1.\mathrm{b}_0+a_0.\mathrm{b}_1$$

$$c_2=a_2.\mathrm{b}_0+a_1.\mathrm{b}_1+a_0.\mathrm{b}_2$$

$$c_3=a_2.\mathrm{b}_0+a_2.\mathrm{b}_1+a_1.\mathrm{b}_2+a_0.\mathrm{b}_3$$

$$c_4=a_3.\mathrm{b}_1+a_2.\mathrm{b}_2+a_1.\mathrm{b}_3$$

$$c_5=a_3.\mathrm{b}_2+a_2.\mathrm{b}_3$$

$$c_6=a_3.\mathrm{b}_3$$

Reducing $c(x)$ by x^4+1 results in the modular product:

$$d(x)=d_0+d_1.x+d_2.x^2+d_3.x^3$$

With:

$$d_0=a_0b_0+a_3b_1+a_2b_2+a_1b_3$$

$$d_1=a_1b_0+a_0b_1+a_3b_2+a_2b_3$$

$$d_2=a_2b_0+a_1b_1+a_0b_2+a_3b_3$$

$$d_3=a_3b_0+a_2b_1+a_1b_2+a_0b_3$$

Or in matrix representation we can write [5]:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_2 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1.6}$$

Note: $x^4 + 1$ has a multiplicative inverse even though it's not an irreducible polynomial over $GF(2^8)$ [5].

1.3. AES Algorithm

The AES algorithm is a block cipher, which means that data is encrypted by a fixed number of bits, called blocks, and not by single bits as is the case in stream ciphers. In our case, the data block is128-bit long. The block of plain-text, along with that of the key and the produced cipher text are divided into 16 bytes and put in a matrix form as a 4 by 4 grid, column-wise ordered, each column containing four bytes.

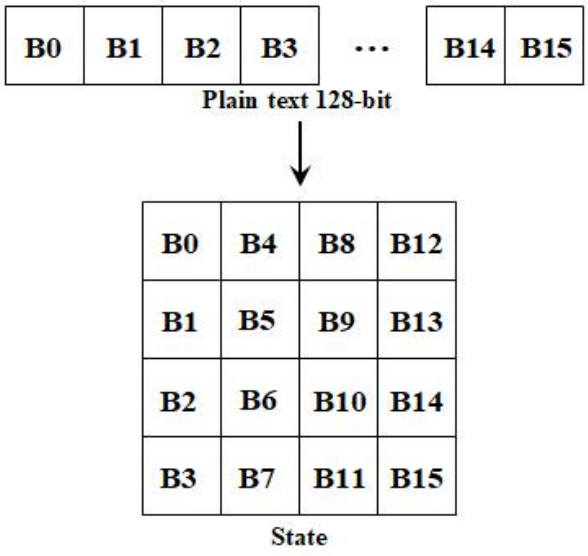


Figure 1- 2. The sate array

In order to organize and facilitate the operations done on the plain-text and the key in the ciphering process. The AES algorithm, either in encryption or decryption, consists of four transformations that act each time to the immediate cipher result, called state. Each of these four transformations applies in its corresponding layer, and all of them make up a "round", which is then iterated N times: 10 times for a 128-bit key, 12 times for a 192-bit key and 14 times for a 256-bit key (plus the initial round) [5], [8].

In our work, we chose to work on AES with key length of 128, seeking simplicity, since using a longer key only requires more iterations of the same algorithm and results in more robustness of the system. In what follows, we explain how the key schedule of AES-128 is obtained, the encryption and decryption process of the algorithm as well as the different transformations applied in every layer of it.

1.3.1. Encryption

- **Description:** the cipher consists of an initial round of adding round key followed by 9 main rounds and finally a final round, slightly different that the main rounds.
- **Flowchart:**

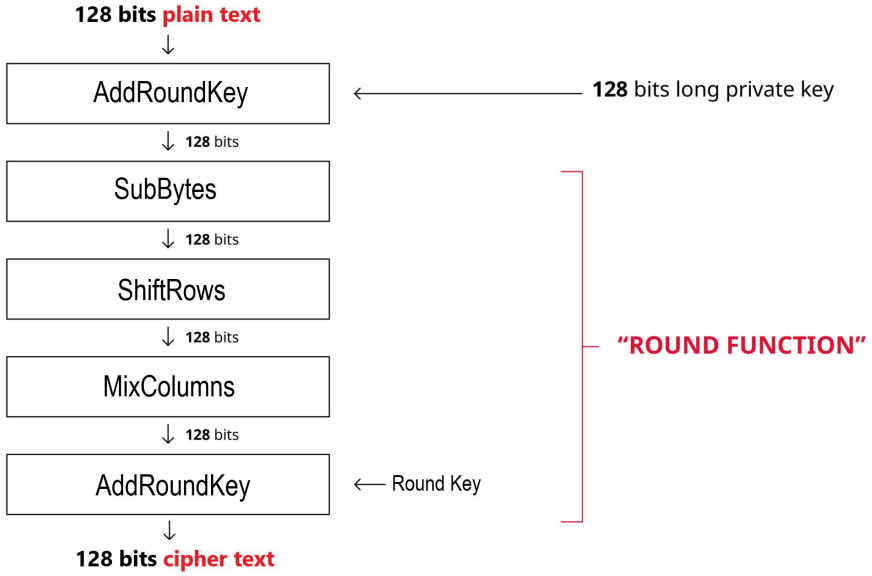


Figure 1- 3. Encryption flowchart.

Bytes Substitution Transformation

It is the only nonlinear operation in the algorithm that substitutes every byte into its correspondent one from a lookup table called “The S-Box” that is invertible.

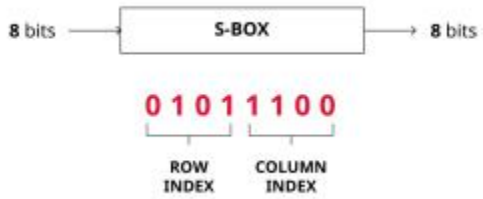


Figure 1- 4. Bytes Substitution **[9]**.

The first nibble of the data determines the row index and the second nibble determines the column row in the S-box.

- **Rijndael S-box:** stands for Rijndael substitution box, is a two-step mathematical transformation used in the Byte Substitution stage of Rijndael cipher that serves as a one-to-one mapping of every byte of the input data to a byte in the output of the Byte Substitution layer.

  Note: the byte substitution transformation provides great confusion in Rijndael algorithm.

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

Figure 1- 5. Rijnhael Forward S-box lookup table **[8]**.

To compute the individual entries of the S-box, two operations on the input byte are needed: $GF(2^8)$ inversion followed by an affine transformation.

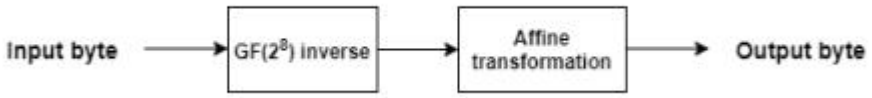


Figure 1- 6. Computation of the S-box.

$GF(2^8)$ *Inversion*: for A in $GF(2^8)$, A^{-1} is also in $GF(2^8)$ such that $A.A^{-1} = 1 \mod P(x)$.

Affine transformation: the inverse of the input byte is multiplied by a predefined constant 8x8 matrix and added to a predefined constant vector of length 8 over $GF(2)$ .The result is then stored in the S-box.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 1- 7. Affine transformation **[8]**.

Since obtaining the S-box requires lots of computations, in our work we preferred to use a ready lookup table representing it rather than doing all the calculations, for the sake of increasing the speed of our algorithm.
Note: a similar lookup table representing the inverse S-box was used in the decryption process**.**

Shift Rows Transformation

The rows are cyclically shifted to the left. The first row is not shifted (shifted by zero), the second is shifted by one, the third by two and the fourth by three.

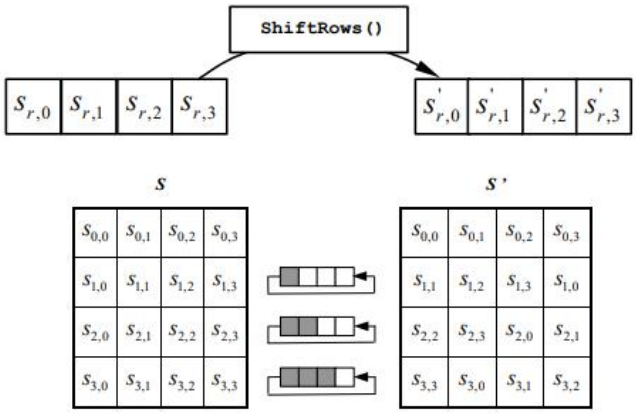


Figure 1- 8. Shift Rows Transformation **[8]**.

Mixing of Columns Transformation

This operation is based on Galois Field multiplication. The columns of the state are considered as polynomials of degree 4 of $GF(2^8)$ with coefficients that are also from $GF(2^8)$. These polynomials are next modulo x^4+1 multiplied with a fixed predefined polynomial $(03)_{16}.x^3+(01)_{16}.x^2+(01)_{16}.x+(02)_{16}$. Thus, in matrix representation as seen in last chapter, each of the columns of state is replaced by the modulo multiplication of that column with the Maximal Distance Separable(MDS) matrix **[5]**.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix} = \begin{bmatrix} S'_0 \\ S'_1 \\ S'_2 \\ S'_3 \end{bmatrix}$$

Figure 1- 9. Mixing of Columns Transformation **[9]**.

$$S'_0 = 2.S_0 \oplus 3.S_1 \oplus 1.S_2 \oplus 1.S_3$$

$$S'_1 = 1.S_0 \oplus 2.S_1 \oplus 3.S_2 \oplus 1.S_3$$

$$S'_2 = 1.S_0 \oplus 1.S_1 \oplus 2.S_2 \oplus 3.S_3$$

$$S'_3 = 3.S_0 \oplus 1.S_1 \oplus 1.S_2 \oplus 2.S_3$$

Note: the Mix Columns transformation together with the Shift Rows transformation represent the primary source of diffusion in Rijndael algorithm.

Addition of Round Key Transformation

The Round key generated in each round is $GF(2)$ added to our resulted matrix after the mix columns transformation by a bitwise XOR operation.

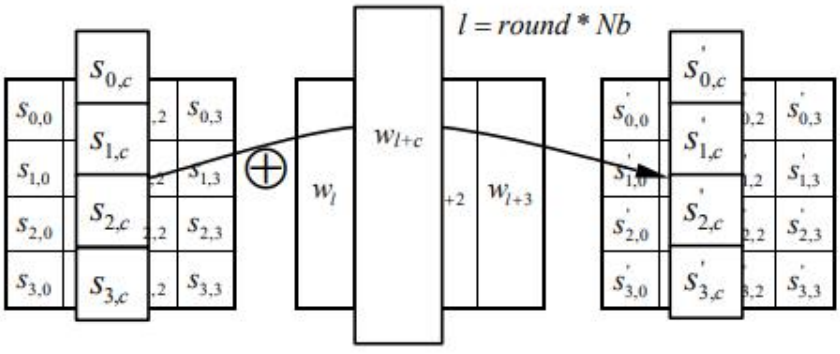


Figure 1- 10. Round key addition **[8]**.

Note: these four transformations are applied to the state for 9 rounds, the last round does not include the Mix Columns transformation.

1.3.2. Decryption

- **Description:** Because the decryption is the inverse operation of encryption in AES, each transformation in the decryption process is nothing but the inverse of the corresponding transformation in the decryption process. This means that the Byte Substitution transformation, the Shift Rows transformation and the Mix Columns transformation in encryption are equivalent to the Inverse Byte Substitution Transformation, the Inverse Shift Rows transformation and the Inverse Mix Columns transformation in decryption, respectively. For Add Round Key transformation, it is the same one used in both encryption and decryption since it is based on the bitwise XOR operation that is considered to be the inverse of itself. Moreover, the key schedule is reversed in a way that the last round key used in encryption represents the key that is used in the first round of decryption.

- **Flowchart:**

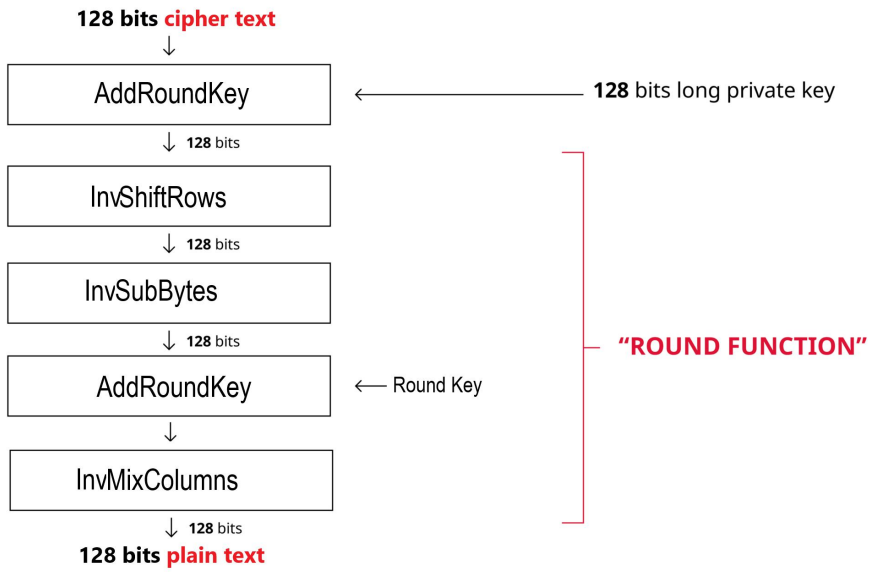


Figure 1- 11. Decryption flowchart.

Inverse Mix Columns Transformation

As we recall in the encryption process, the MDS matrix was used in the modulo multiplication. In order to have the inverse of this transformation, the inverse of this matrix is used instead, which is generated after the x^4+1 modulo multiplication of each column of the state with the polynomial $(0B)_{16}.x^3+(0D)_{16}.x^2+(09)_{16}.x+(0E)_{16}$ being the inverse of the one used in Mix Columns transformation in encryption.

0e	0b	0d	09		S_0	S'_0
09	0e	0b	0d	X	S_1	S'_1
0d	09	0e	0b		S_2	S'_2
0b	0d	09	0e		S_3	S'_3

Figure 1- 12. Inverse Mix Columns Transformation **[9]**.

$$S'_0 = 0e.S_0 \oplus 0b.S_1 \oplus 0d.S_2 \oplus 09.S_3$$

$$S'_1 = 09.S_0 \oplus 0e.S_1 \oplus 0b.S_2 \oplus 0d.S_3$$

$$S'_2 = 0d.S_0 \oplus 09.S_1 \oplus 0e.S_2 \oplus 0b.S_3$$

$$S'_3 = 0b.S_0 \oplus 0d.S_1 \oplus 09.S_2 \oplus 0e.S_3$$

Inverse Shift Rows Transformation

In the shift Rows transformation of the encryption process, a circular shift to the left was applied. To inverse this transformation, the same amount of shift is applied to the rows of the matrix in the opposite direction, meaning that the first row is not shifted, the second is shifted over one byte to the right, the third row is shifted over two bytes to the right and the fourth row is shifted over three bytes to the right.

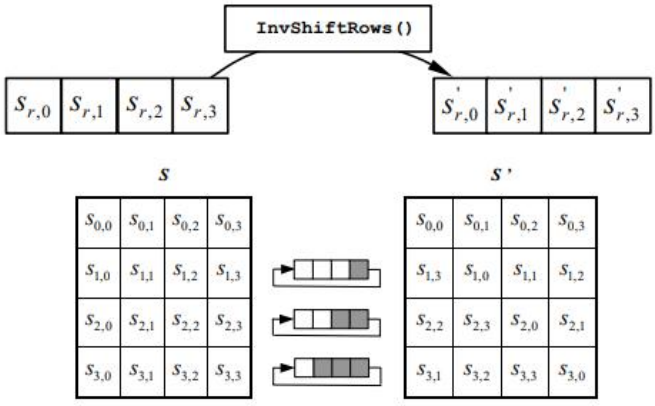


Figure 1- 13. Inverse Shift Rows Transformation **[8]**.

Inverse Byte Substitution Transformation

Similar to the Byte Substitution transformation that uses the (forward) S-box to substitute every byte from the plain-text, the Inverse Byte substitution transformation uses the inverse S-box to substitute every byte of the cipher text. The entries of the inverse S-box are obtained by taking the inverse of the affine mapping referred to previously, followed by taking the inverse in $GF(2^8)$.

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

Figure 1- 14. Rijndael Inverse S-box lookup table **[8]**.

As a demonstration example, if the byte from the cipher text is $A = (28)_{16}$, then its substitute in the Inverse Byte Substitution transformation is $B = S(A) = (ee)_{16}$.

1.3.3. Key schedule generation

Starting with a single secret key, a round key is generated after each round iteration, resulting in 11 partial keys used in the initial round, the 9 main rounds and the final round. The collection of the expansion keys can be seen as a 4 by 44 matrix where each column is numbered from 0 to 43 and contains 4 bytes. The first four columns are filled with the initial secret key, as illustrated below:

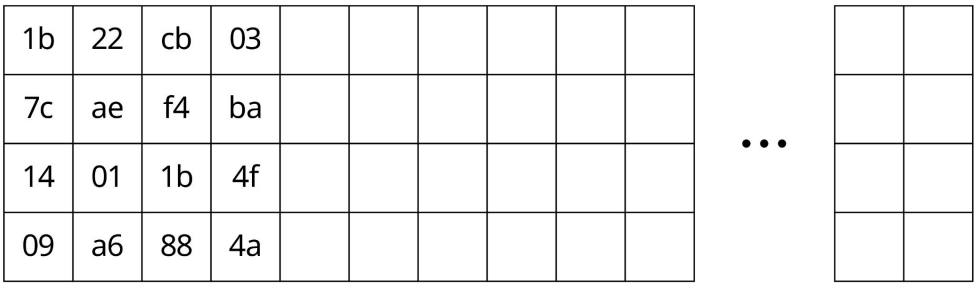


Figure 1- 15. Initial Secret Key **[9]**.

To fill in the rest of the columns, we follow the below procedure:

- Columns having a position *i* that is a multiple of four (i.e. columns: 4, 8, 12, . . . , 40) are calculated in the following manner:

1- Apply a Rotation Word transformation to the previous column, that is, a column is shifted up by one step, and then apply byte substitution transformation explained previously.

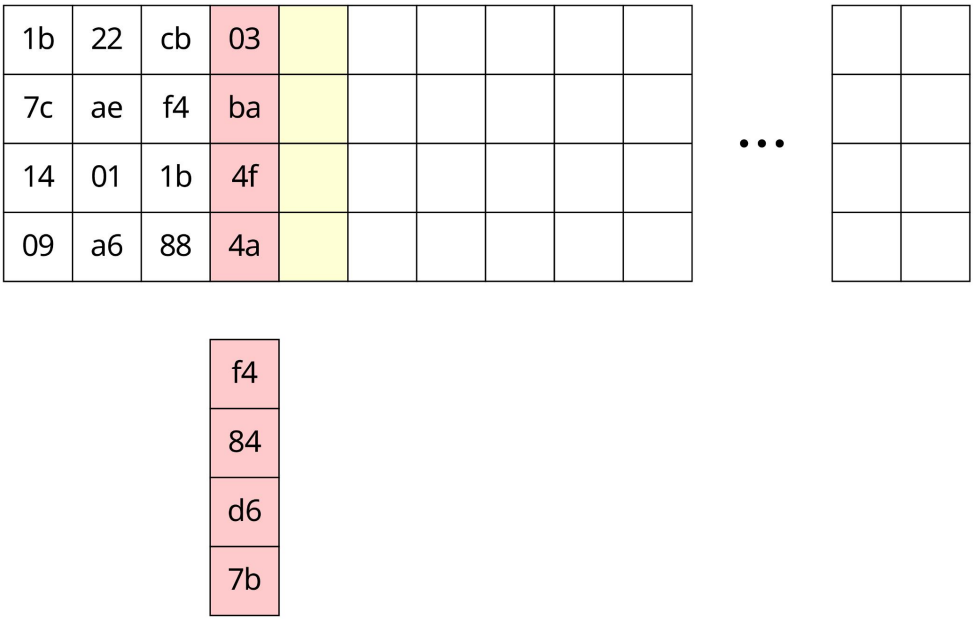


Figure 1- 16. The Key After The Word Rotation **[9]**.

2- XORing the resultant column with the column in position *i* - 4, and XORing that with a predefined round constant column.

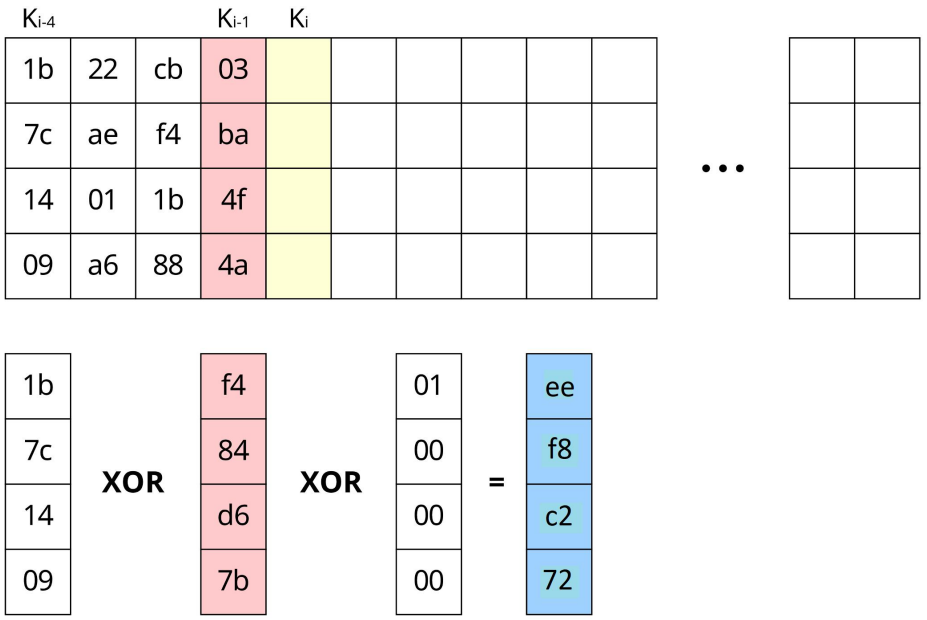


Figure 1- 17. The Key After The XOR Operation **[9].**

- The remaining columns are calculated by XORing the previous word to them with the word four positions before them.

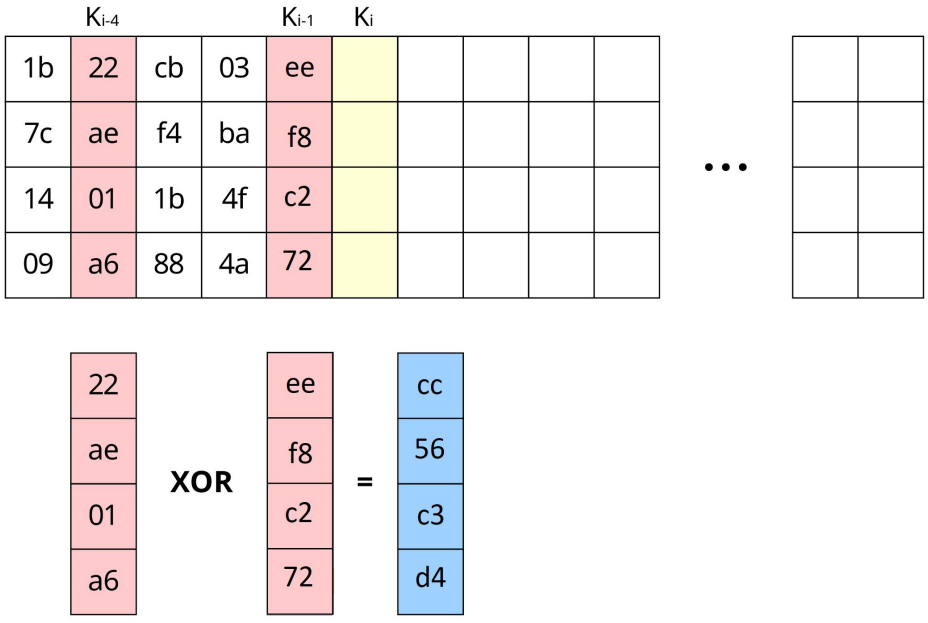


Figure 1- 18. The Key After The XOR Operation with previous columns **[9]**.

And the procedure repeats until the last round key.

1b	22	cb	03	ee	cc	07	04				
7c	ae	f4	ba	f8	56	a2	18				
14	01	1b	4f	c2	c3	d8	97				
09	a6	88	4a	72	d4	5c	16				

Figure 1- 19. The Round Key **[9]**.

1.4. DE10 standard SoC FPGA board

1.4.1. What is an FPGA?

A FPGA, which stands for Field Programmable Gate Array, is an integrated circuit that contains an array of configurable logic blocks with programmable interconnect hierarchy in order to wire and reprogram these blocks together to perform desired combinational and sequential applications using its large resources of logic gates and memory elements [10] . FPGAs allow flexible reprogramming after manufacturing, and their configuration is specified using a hardware description language (HDL), typically Verilog and VHDL [11]. In our project, we have used the DE10 standard board containing the Cyclone V SoC FPGA device.

1.4.2. DE10 standard

The DE10 standard board represents a robust hardware design platform. It combines the dual-core Cortex-A9 embedded cores with industry leading programmable logic. Intel's SoC integrates an ARM-based hard processor system (HPS) consisting of a processor, peripherals and memory interfaces tied seamlessly with the FPGA fabric. The DE10 standard is equipped with a high speed DDR3 memory, video and audio capabilities, Ethernet networking and more. The DE10 runs with a computer that has Microsoft Windows XP or later. [12], [13].

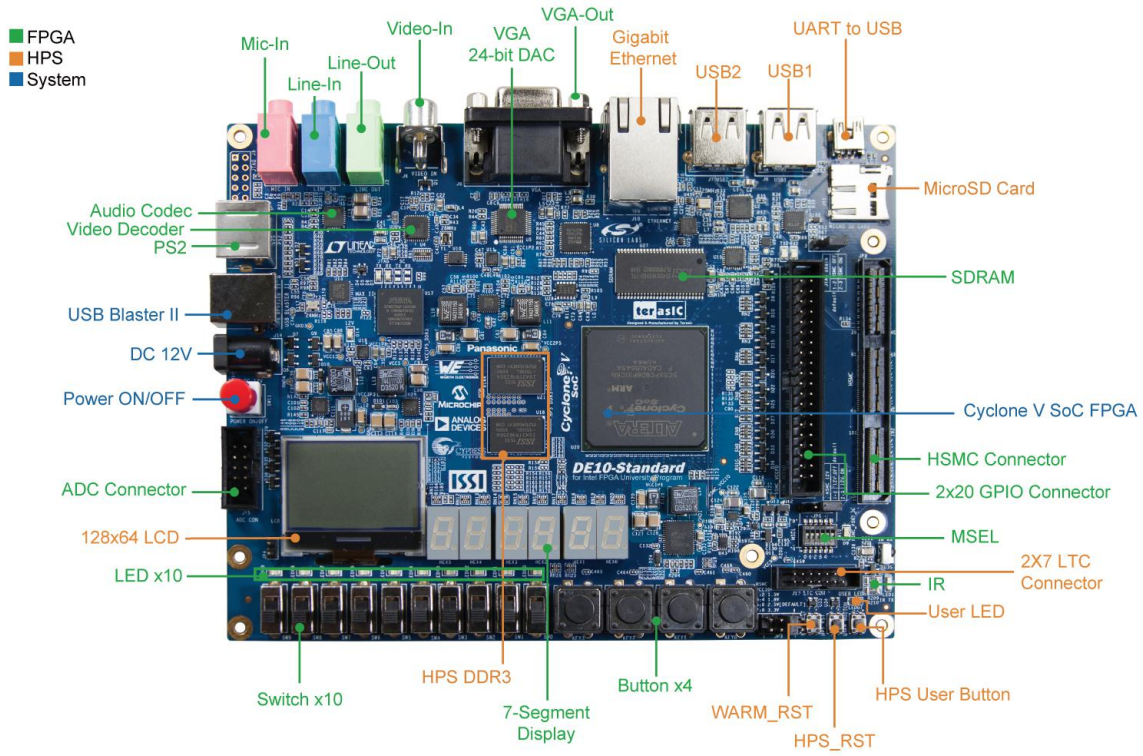


Figure 1- 20. Top View of the DE10 Board **[13]**.

1.4.3. Specifications and components

DE10 standard board has the following specifications [14]:

System

Communication

• Two port USB 2.0 Host (ULPI interface with USB type A connector)

• UART to USB (USB Mini-B connector)

• 10/100/1000 Ethernet

Power

• 12V DC input

FPGA

• Intel Cyclone V SE 5CSXFC6D6F31C6N device

• Serial configuration device – EPCS128

• USB-Blaster II on-board for programming; JTAG Mode

• 64MB SDRAM (16-bit data bus)

• Four 50MHz clock sources from the clock generator

• TV decoder (NTSC/PAL/SECAM) and TV-in connector

HPS (Hard Processor System)

• 925MHz Dual-core ARM Cortex-A9 MPCore processor

• 1GB DDR3 SDRAM (32-bit data bus)

• Micro SD card socket

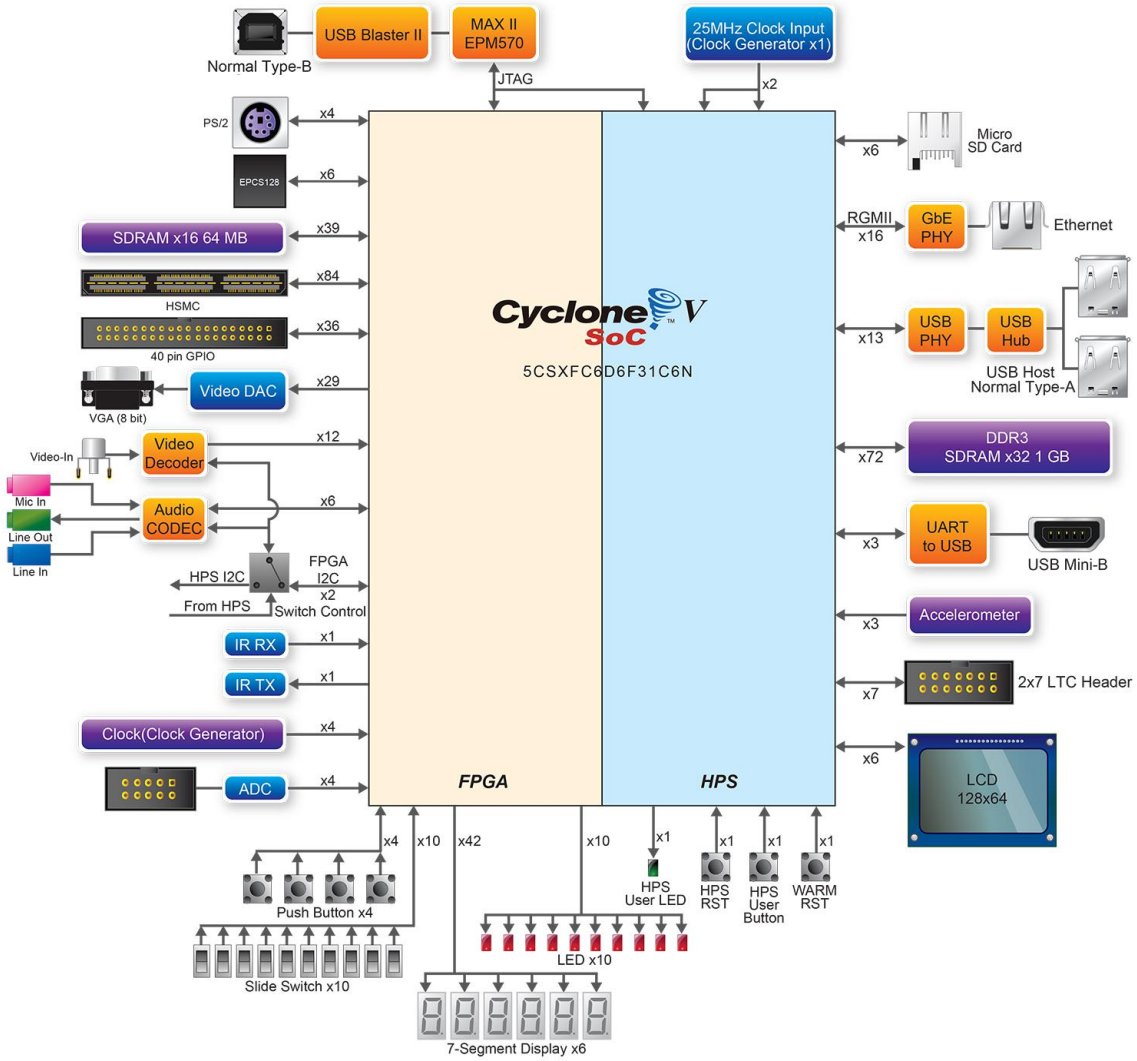


Figure 1- 21. Block Diagram of the DE10 Board **[13]**.

1.5. Video file formats

Video file Formats are standardized rules for storing the containers, codecs, meta data and sometimes even folder structure of video files, so that it's easier to support them across a large number of different devices and players.

Since the very first digital video formats became popular, new formats have been developed every year in an attempt to take their place - aiming for improvements in image quality, file size, video playback, and the addition of special features.

For the purposes of our work, we're mainly covering two formats and a command line tool to convert between them.

1.5.1. Y'CbCr format (.yuv)

Y'CbCr is a digital uncompressed video and image file format based on the YUV model that specifies the ratio between the luma component Y' (brightness) and the chroma components C_B (blue projection) and C_R (red projection) per pixel in a color pipeline.

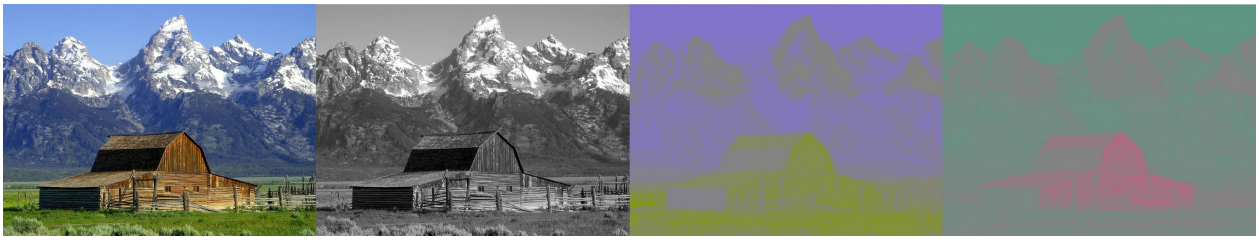


Figure 1- 22. A color image and its Y′, C_B and C_R components **[15]**.

YUV model was invented back in the 1950s when engineers were given the task to compress video for color television without exceeding the transmission bandwidth of black-and white infrastructure. The solution was a system were luma(brightness) information was saved for every pixel of an image but the chroma(color) information was shared between 2 or more pixels since the human eye sees differences in black and white better than in color. This approach is called chroma subsampling, and it has the following schemes:

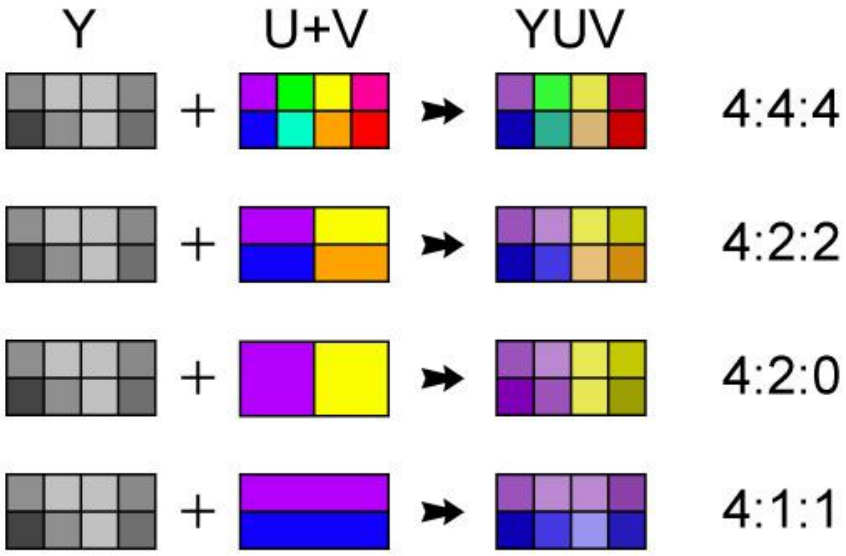


Figure 1- 23. Chroma subsampling schemes **[16]**.

Our system uses 4:2:2 format which means each two pixels occupy 4 bytes of space [17]. So a resolution of 640x360 will occupy 640*360/2*4 = 460800 bytes for a single frame.

1.5.2. MPEG-4 Part 14 format (.mp4)

MPEG-4 Part 14, MP4 as an abbreviation, is a compressed file format created by Motion Picture Experts Group (MPEG) and is based on Apple QuickTime MOV format finalized in 2003. It's a multimedia container format that includes not only videos, but also audio, images and subtitles. Usually, MP4 files with video and audio are save with the official .mp4 extension [18].

1.5.3. FFMPEG Command line tool

FFMPEG is a very fast video and audio converter that can also grab from a live source. It can convert between arbitrary sample rates and resize video on the fly with a high quality polyphase filter. FFMPEG calls the *libavformat* library to read input files and get packets containing encoded data from them. When there are multiple input files, FFMPEG tries to keep

them synchronized by tracking lowest timestamp on any active input stream. Encoded packets are then passed to the decoder which produces uncompressed frames (raw video/PCM audio/...) which can be processed further by filtering. After filtering, the frames are passed to the encoder, which encodes them and outputs encoded packets. Finally those are passed to the muxer, which writes the encoded packets to the output file [19].

```
ffmpeg [global_options] {[input_file_options] -i input_url} ... {[output_file_options] output_url} ...
```

Figure 1- 24. FFMPEG command synopsis.

FFMPEG provides some global generic Codecs, which are standards of video content compression. Video encoding refers to the process of converting raw video into a digital format that is compatible with many devices. When it comes to streaming, videos are often compressed from gigabytes of data down to megabytes of data. H.264, also known as AVC, is the most common video codec [20].

1.6. Chapter summary

In this chapter, we have seen the basic concepts of cryptography and its different types. Then, we introduced the AES algorithm along with the mathematics behind it, including finite fields. After that, we discussed each step of the algorithm individually: encryption, decryption and round key generation, followed by a description of DE10 standard FPGA board. At the end, we looked the video file formats used in our work and the FFMPEG tool. In the next chapter, we will move to the implementation part of our project.

2 Design and Implementation

Test vectors

In order to test the different Designs used in our experiment, we referred to the test vectors provided by the Federal Information Processing Standards Publications (FIPS PUBS), issued by the US National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce [8], [21].

Software

- Quartus Prime 18.1
- ARM SoC Embedded Design Suite (EDS)
- PuTTy
- Visual Studio Code
- FileZilla

2.1. Software implementation of AES algorithm using C language

As a start, we implemented the AES algorithm from scratch using C, this strengthen our algorithmic theoretical knowledge. C was the language of choice because we wanted to get the most performance from the limited platform we are working on, moreover we are more familiar with C debug tools.

Since we are expecting all sizes of files, we had to take care of the instances where the files size is not a multiple of 128 bit by counting the number of bytes of the last 128bit that are our actual data rather than some added garbage data, and storing the count as a header in the encrypted file. This header will be used in the decryption process to discard the additional bytes that were not present in the original file. Also, we were more concerned about speed and not executable size or RAM size so we used look-up tables for multiplication in Galois Field. In addition, we made the code load the whole data to be processed to RAM, this way we avoid disk access time.

2.2. VHDL implementation of AES algorithm

For VHDL, an online open-source implementation was used [22] which is based on NIST standard. It has an encryption and decryption speed of 13 cycles with no pipeline as seen in Figure 2- 1. Throughout this document, we will refer to this core as **`aes_core`**.

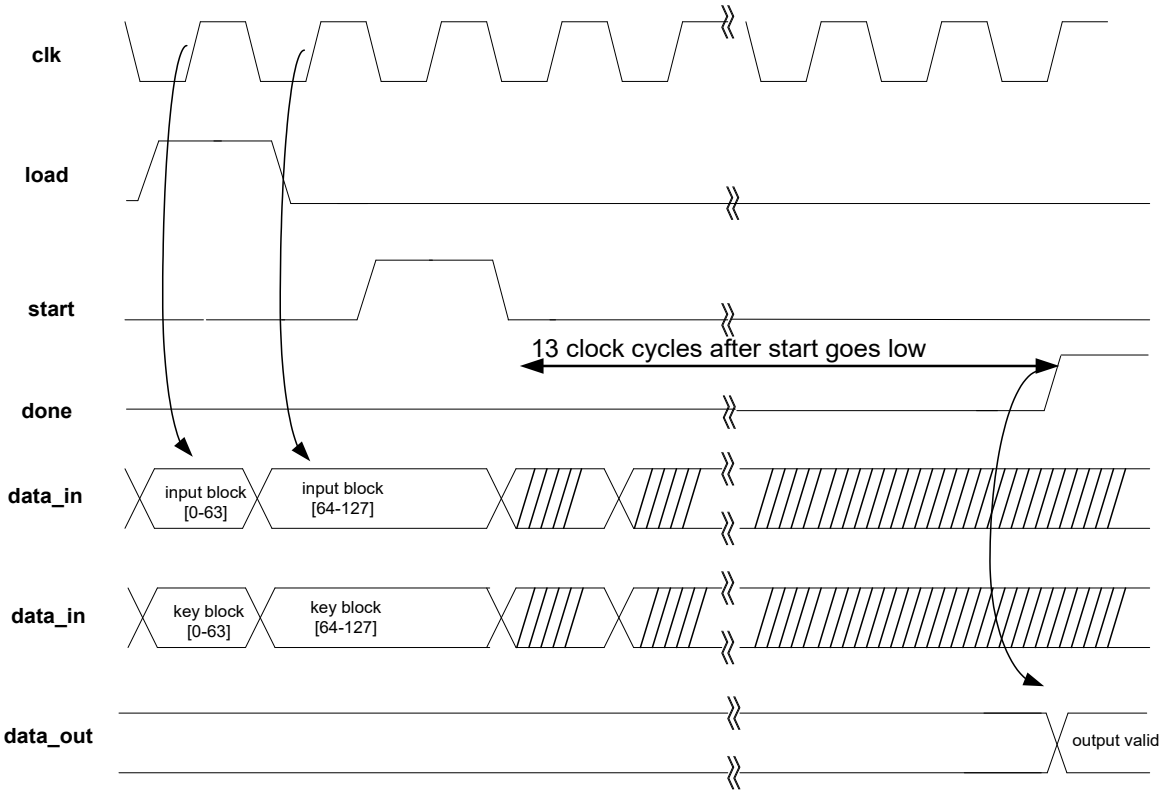


Figure 2- 1. Process sequence for encryption/decryption **[22]**.

We note that there is a bug in the VHDL code, the decryption process requires the use of the last round key instead of the key, that beats the purpose of the symmetry of AES. We wanted to fix it, but unfortunately, we did not have the time. So we just used it as it is since it did not impact performance. We also note that we had modified it to load 128bit of data at a time instead of 64bit, also changed the signal assignment of the ports of the design to fit the Little Endianness of our system and accommodate for different C memory write statements (64bit and 32bit).

2.3. Hybrid implementation of AES algorithm

Having a purely hardware implementation to do the cryptography is best, However, that is not very flexible and increases the hardware size. For that purpose, we created C program to control the **`aes_core`**. We made multiple versions while designing; we started by the simple ones and moved to the faster more complex designs, as we will discuss now.

2.3.1. Lightweight AXI and PIOs

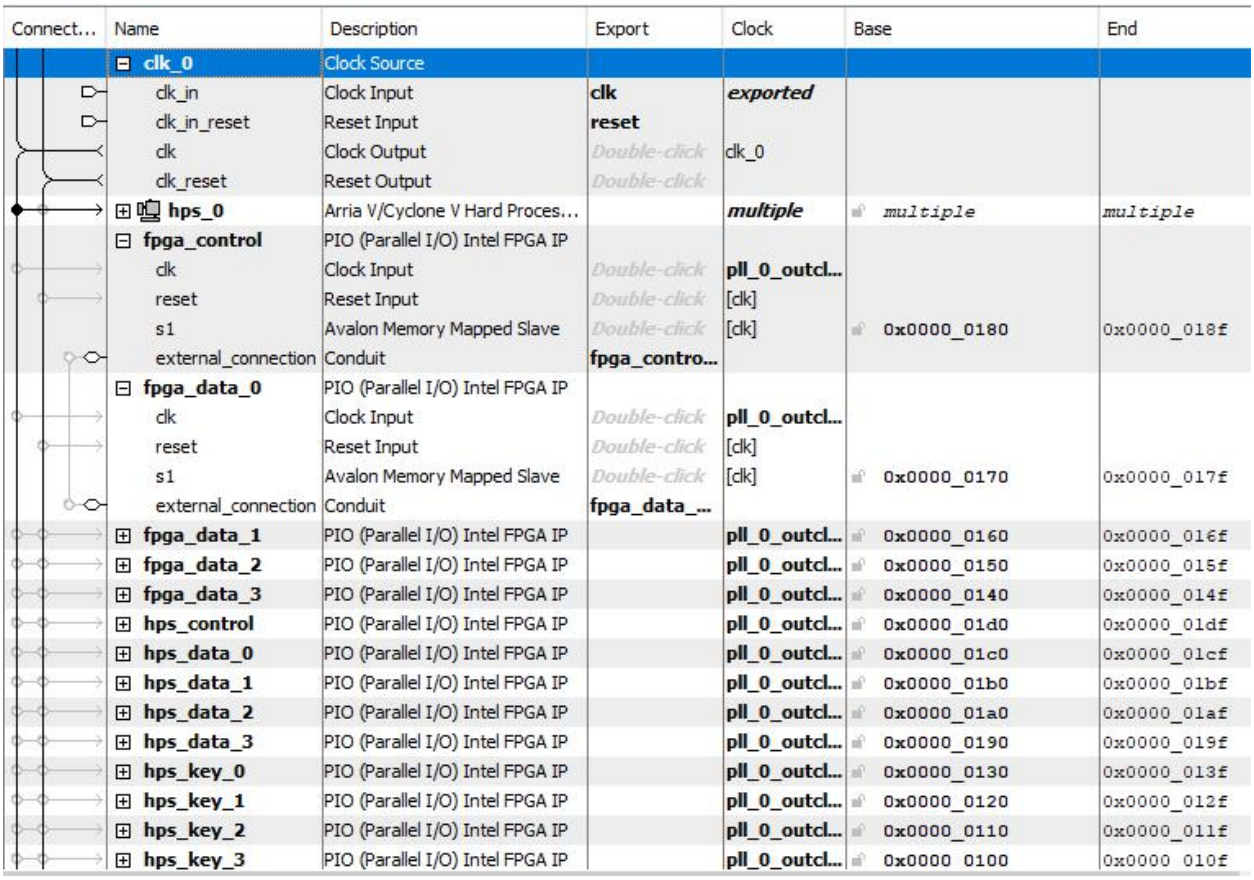
The HPS logic and FPGA fabric are connected through the Advanced eXtensible Interface (AXI) bridge. The HPS contains the following HPS-FPGA AXI bridges [14]:

- FPGA-to-HPS Bridge: 32, 64 or 128 bits.

- HPS-to-FPGA Bridge: 32, 64 or 128 bits.

- Lightweight HPS-to-FPGA Bridge: 32 bits.

For this design, we used the lightweight bridge with 14 PIOs with maximum width of 32bits. We need 4 PIOs to hold 128bit data; so each of data in, key and data out had 4 PIOs and an additional 2 PIOs of width 8bits to control **`aes_core`**. This requires the C code to handle loading the key and data properly. The Qsys SoC system is shown in Figure 2-2 below.



Connect...	Name	Description	Export	Clock	Base	End
	clk_0	Clock Source				
	clk_in	Clock Input	clk	exported		
	clk_in_reset	Reset Input	reset			
	clk	Clock Output	Double-click	clk_0		
	clk_reset	Reset Output	Double-click			
	hps_0	Arria V/Cyclone V Hard Proces...		multiple	multiple	multiple
	fpga_control	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click	pll_0_outcl...		
	reset	Reset Input	Double-click	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click	[clk]	0x0000_0180	0x0000_018f
	external_connection	Conduit	fpga_contro...			
	fpga_data_0	PIO (Parallel I/O) Intel FPGA IP				
	clk	Clock Input	Double-click	pll_0_outcl...		
	reset	Reset Input	Double-click	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click	[clk]	0x0000_0170	0x0000_017f
	external_connection	Conduit	fpga_data_...			
	fpga_data_1	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0160	0x0000_016f
	fpga_data_2	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0150	0x0000_015f
	fpga_data_3	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0140	0x0000_014f
	hps_control	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_01d0	0x0000_01df
	hps_data_0	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_01c0	0x0000_01cf
	hps_data_1	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_01b0	0x0000_01bf
	hps_data_2	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_01a0	0x0000_01af
	hps_data_3	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0190	0x0000_019f
	hps_key_0	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0130	0x0000_013f
	hps_key_1	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0120	0x0000_012f
	hps_key_2	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0110	0x0000_011f
	hps_key_3	PIO (Parallel I/O) Intel FPGA IP		pll_0_outcl...	0x0000_0100	0x0000_010f

Figure 2- 2. Screenshot of Qsys soc system (Lightweight AXI and PIOs).

2.3.2. AXI and on-chip memory with an interfacing module

To get better performance, we moved to using the AXI bus instead of the Lightweight AXI, with on-chip memory to act as buffer. To expose the on-chip memory to FPGA, we used several External Bus to Avalon bridge; they connect an Avalon memory mapped component to the asynchronous part in the FPGA. With this, we built an interfacing module (FSM) to sit between the **`aes_core`** and the on-chip memory. The main role of this interfacing module is to provide an abstraction layer to the C code in HPS, where only setting one bit will trigger it to do the rest of operation of loading the keys data and starting the encryption/decryption process and also resetting the start bit in HPS side. Another benefit of this design is not using any PIOs, and no memory mapped bridge, thus reducing the latency. For the on-chip memory settings, we have used dual-port access allowing for simultaneous access from HPS and FPGA. The port width is

128bits with memory space of 64 bytes. The Qsys SoC and the block diagram of this design are shown in Figure 2-3 and Figure 2-4. You may notice a PPL component in Figure 2-3 this was used to increase frequency, as we will see in Results and Discussion chapter.

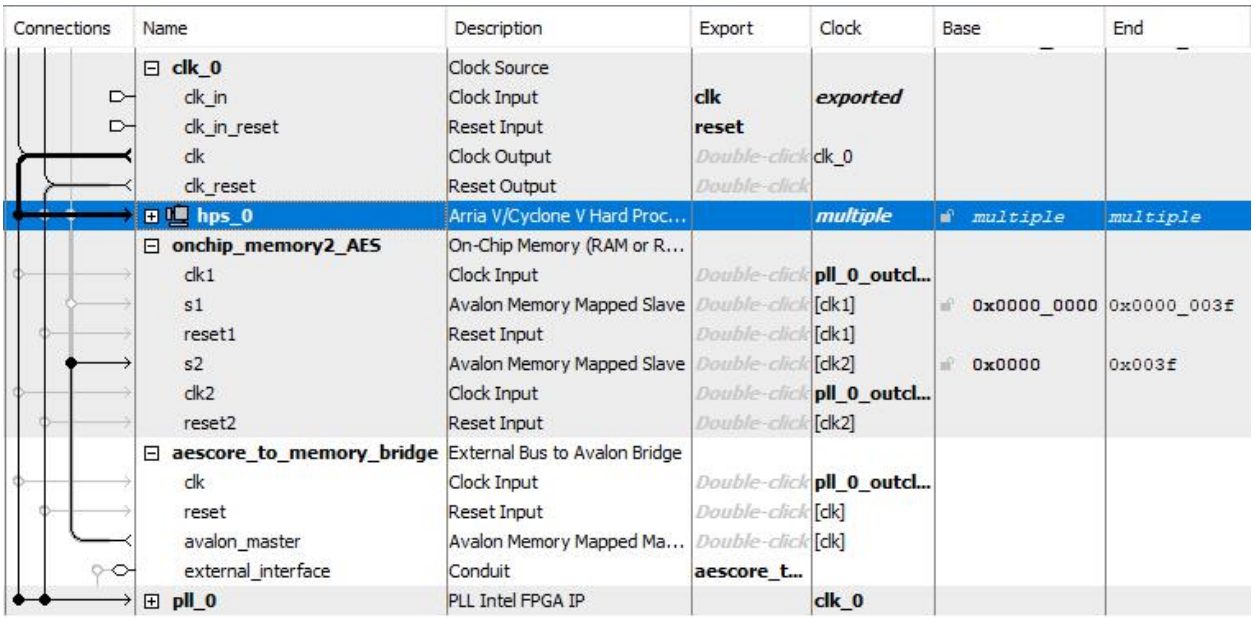


Figure 2-3. Screenshot of Qsys soc system(AXI and on-chip memory).

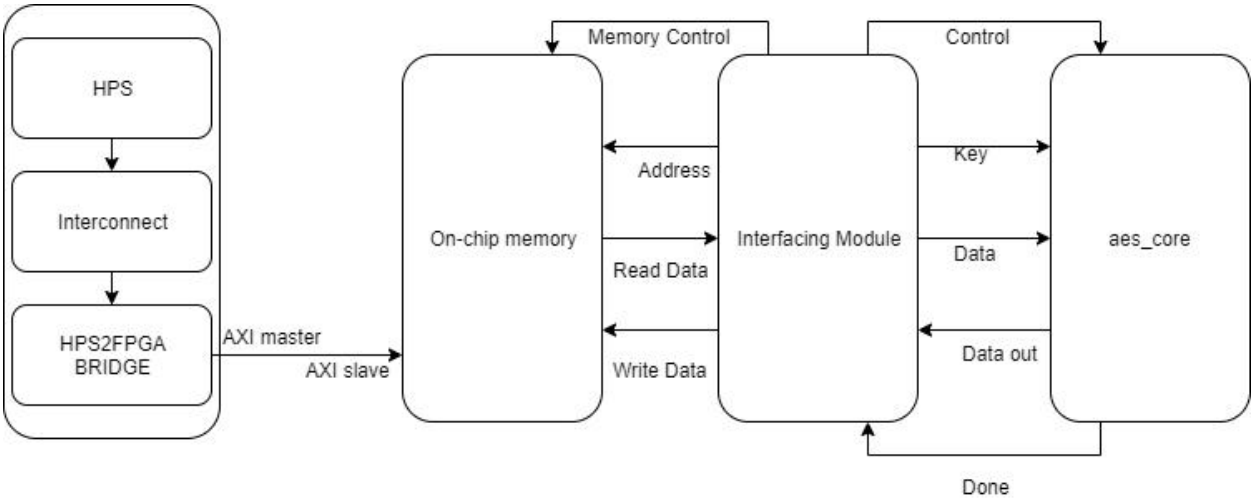


Figure 2-4. Block diagram of our hybrid system using the Interfacing Module.

2.3.3. AES Core as Quartus Intellectual Property component ‘qip’

This implementation uses a modified version of the template found online [23], written in VHDL instead of the original SystemVerilog. Where we imported the VHDL code of **`aes_core`** added some logic for the qip to interface with it, hence it has the same abstraction layer to C code as the previous design. With this approach, we got an Avalon memory mapped slave connected directly to the AXI bus of the HPS which should provide less latency and easier interfacing, as shown in Figure 2- 5 and Figure 2- 6 below.

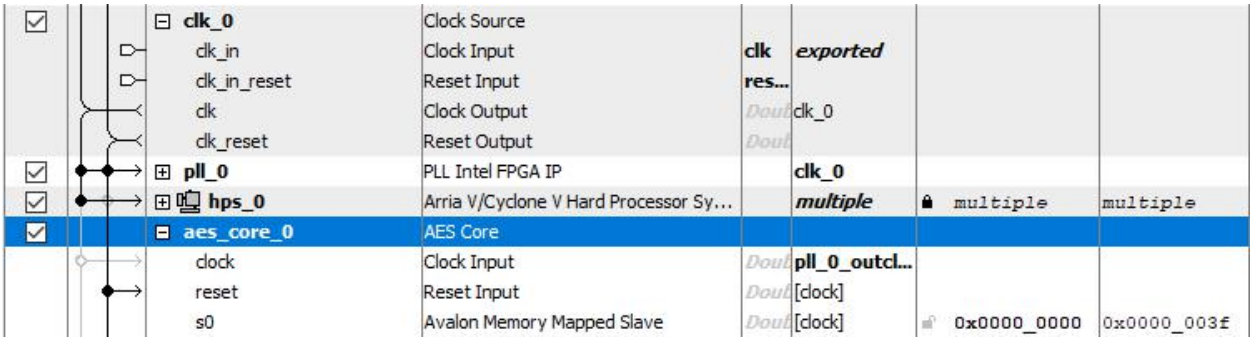


Figure 2- 5. Screenshot of Qsys soc system(AES Core as Quartus IP component).

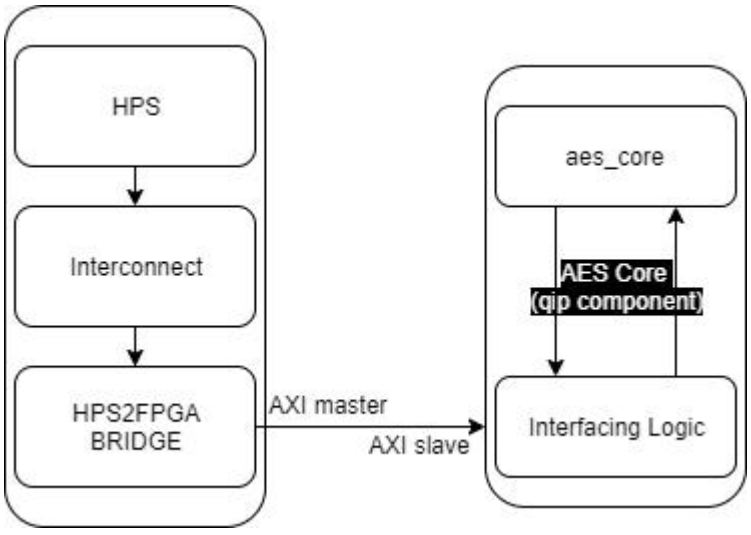


Figure 2- 6. Block diagram of our hybrid system using aes_core as Quartus IP component.

2.3.4. C programs for hybrid designs

The function of the C programs for hybrid design is to load the data from the source into memory then send to **`aes_core`** in 32bit or 64bit at a time depending on the design, for processing and finally reading back the results and storing them. We loaded all the data into memory before processing because of the combination of disk access being slow and how OS handles file writes, this way benchmarking will not include the time it takes for disk access (read/writes).

2.4. Video Recording

2.4.1. Altera hardware reference design

Altera provides demonstration designs to facilitate getting started with the DE-10 Standard board. There are multiple demo projects provided that deal with capturing video through video-in port and doing many operations with it. We choose "DE10-Standard_VIP_TV" demo project [24] as it was easier to get hold of video frames with it. We used about half the qip components from that project but with different setting (see Figure 2- 7), these are:

– Clocked video input II with default settings

– Color plane sequencer II width default settings

– Deinterlacer with default settings

– Clipper II where we changed resolution from 720x480 to 640x360

– Instead of Frame Buffer we used a direct memory access controller with an on-chip memory where we specified the resolution, and set the size of on-chip memory to hold exactly one frame that is 460800 bytes, plus two additional bytes at the beginning which are unused by our C program but are needed by the direct memory access controller, they are not mentioned in the documentation, it is only by experimentation and knowing the exact frame size to expect that we noticed the first two bytes having a value of zero using a hex editor program. The resolution of 640x360 was chosen because any higher resolution will produce more data than can be fitted in the on-chip memory.

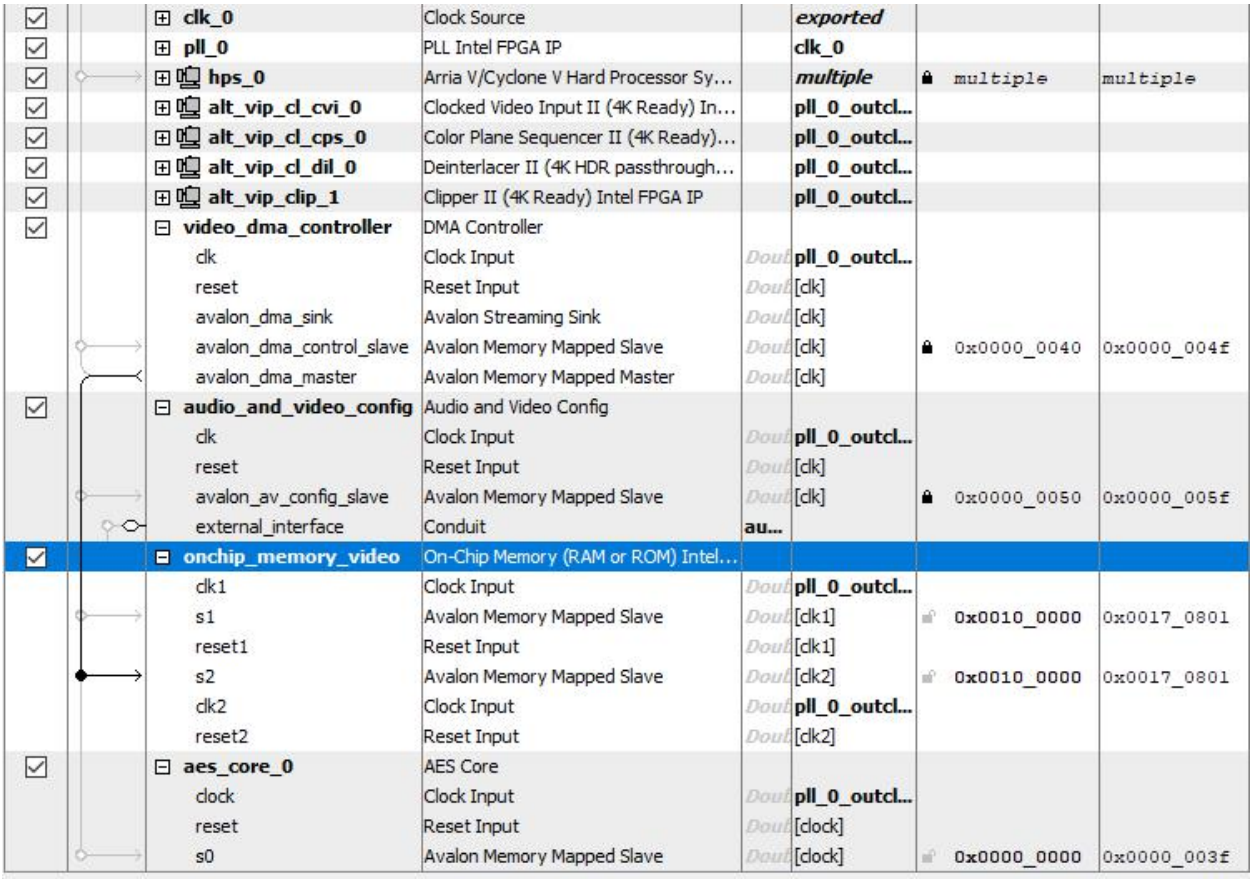


Figure 2- 7. Screenshot of Qsys soc system (capturing video through video-in port).

2.4.2. Software design

2.4.2.1 Getting hold of raw frames

Using the generated video on-chip memory address, and by adding 2 to move the pointer two bytes ahead as discussed in **2.4.1** we access the frame using *fwrite()* C function, "The *fwrite()* function writes binary and text data from an array to a given data stream [25]", where we specify the size as discussed above in section **1.5.1** to be 460800. This leads us to the next section where we discuss why we used *frwite()* in the first place.

2.4.2.2 Passing the raw frames to FFMPEG and setting its flags

One obvious problem is the fact that the video frames are in raw format specifically the planar *uyvy422*. This means a size of 460800 bytes for each frame in 640x360 resolution. Taking the standard film frame rate of 24 frames per second and a video length of 30 seconds, it will result in 460800x24x30 = 317 MB which is a large amount of data given our constrained system. Hence, FFMPEG is used to encode the raw frames into a more manageable MP4 file. Encoding uses a set of techniques to optimize, compress and reduce the data needed to reproduce the original video in an acceptable quality that meets human eye perception. FFMPEG accepts raw frame data either from a file stored in disk specified using input arguments, or using the standard input pipe. In our project, we wanted to process data in real-time so we used the latter. Again to achieve this we considered multiple approaches. The first is the *system()* function which is used to run another program by passing a command line, pointed to by a string, to the operating system's command processor that will then be executed. We quickly disposed of this approach since it was limited to only reading from a file. Next, we thought of creating a new thread and then use *execv()* which replaces the current process image with a new process image specified by a path and create a pipe, "a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process [26] ", using *pipe()* function to run FFMPEG which gives the user a lot of control, but there was another method that we will discuss next which does all of this while also handling errors properly, for this reason we decided the current approach is very error prone and unnecessary overkill. So after some research, we decided to use the *popen()* function which is described as: "The *popen()* function shall execute the command specified by the string command. It shall create a pipe between the

calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe" [27]. This was the easiest and safest way to send data to FFMPEG by writing to the pipe created by *popen()* between our process and the FFMPEG process. This leads to the next point which is the commands specified in *popen()* to run FFMPEG with a set of flags, below is a list of the most important ones:

***-y*:** overwrite files without asking the user.

***-f rawvideo*:** force the format to be raw video.

-framerate 60: sample at a rate of 60 frames per second.

-use_wallclock_as_timestamps 1: this enables the stamping of raw frames as they are received using the system clock, effectively placing the frames in appropriate timeline position in the encoded output file.

-pix_fmt uyvy422: specify what the raw frames format is.

-preset ultrafast: sets FFMPEG to be as lightweight as possible on hardware resources.

-pipe 1 –blocksize 1 | name_of_receiving_process: this explicitly says write the output data to the process named "name_of_receiving_process" in standard output pipe one byte at a time.

Now using *fwrite()* and passing the address of the on-chip memory, the size and the files pointer returned by *popen()* we can encode the raw frames into *matroska* format.

2.4.2.3 Encrypting the generated video

As indicated in the last section, we have implemented four C programs to be executed using the flag 'name_of_receiving_process'. These are used for testing the effect of hybrid system on this application. The first C program receives data and stores it in RAM. When the recording session is timed up, the program then encrypts it using the hybrid implementation. After all processing is done, it writes the data to disk. The second one, however, uses software implementation to encrypt. The third and fourth programs are basically a copy of the first two, but the encryption happens in real-time. Again, as in **section 2.3.4,** we process everything in memory first before writing to disk, because otherwise more CPU time will be taken to disk access rather than being spent on encoding as a consequence the resulting video will contain a lot of lags. Our system flowchart is shown in Figure 2- 8.

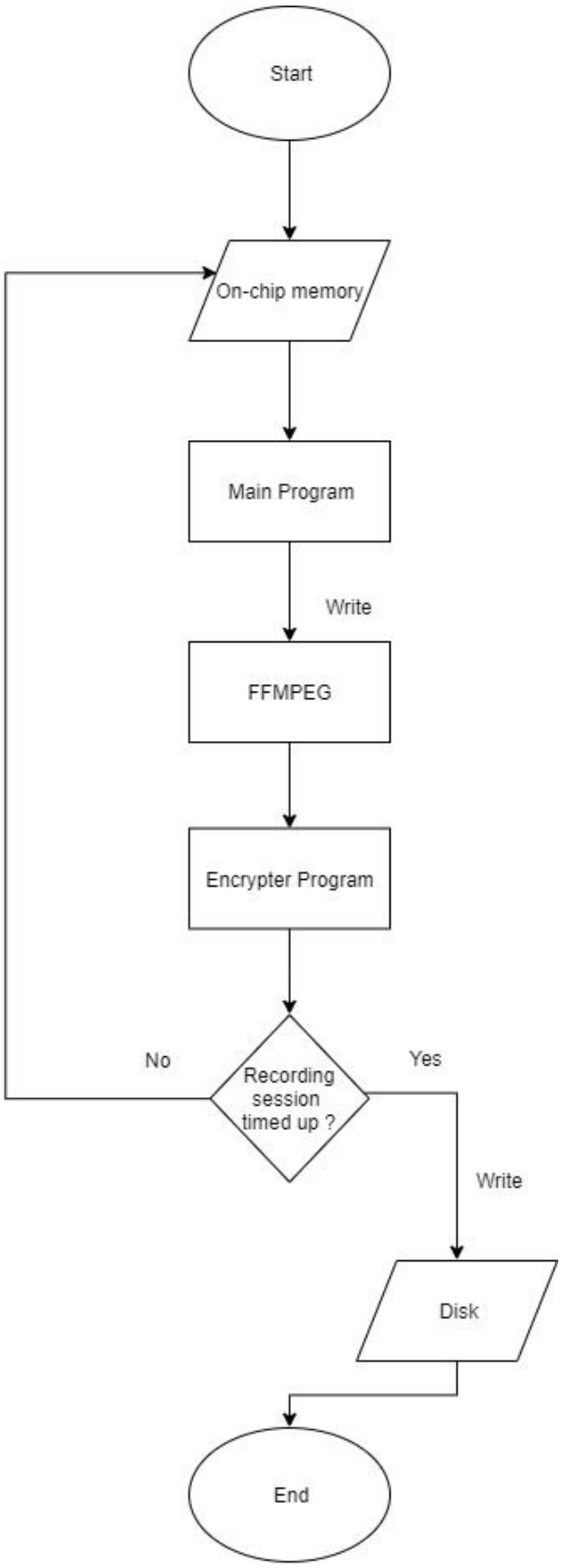


Figure 2- 8. Flowchart of our video encrypting system.

One thing we note here is that this design sometimes introduces a similar effect to screen tearing. “Screen tearing is a visual artifact in video display where a display shows information from multiple frames in a single screen draw” [28]. However, instead of a “display” we have FFMPEG and “single screen draw” as a single frame in the output video. We think this occurs due to HPS reading from Frame Buffer not being synchronized with the video chip writing to it. Either HPS is slow at reading hence video chip would have written more than one frame in the time it takes for HPS to finish reading the current frame. So we end up with an upper portion of frame 1 with lower portion of frame 2, or it could also be the other way around where HPS is faster than video chip. One way to fix this effect is by using interrupts. For hardware part, it will require creating a video DMA qip component from scratch having the ability to stop the video chip from writing to Frame Buffer then it interrupts the HPS to read (calls an ISR containing *fwrite()*). However, we were squeezed in time so we could not do it.

2.5. Chapter summary

In this chapter, we explored the different software and hardware methods that we used in our design and implementation, starting by pure software systems in C language, and ending by our hybrid system that uses the **`aes_core`** as Quartus IP component. After that, we explained the methods used to record then encrypt the generated video. In the next chapter, we will discuss and explain the results obtained by every implementation.

3 Results and Discussion

In order to compare the performance of each design, we used the *clock_gettime()* function defined in <*time.h*> header file which supports up to nanosecond accuracy [29]. The comparison is done taking the pure software implementation as our reference design, and assessing the performance of each hybrid implementation by calculating the acceleration ratio where:

$$\text{Acceleration ration} = \frac{t_{\text{Software System}}}{t_{\text{Hybrid System}}}$$

3.1. Results Comparison and Interpretation

3.1.1. AES 128bit Acceleration

The first test was done using the C implementation and the design of section **2.3.1** with 50MHz clock, and to our surprise this design performed worst by a speed of 0.67 of that of the software implementation. So we increased the frequency of the hybrid design to try and make it faster, we got a little bit faster but could not even match the software implementation speed and only managed to get as close as 0.87 of reference speed. Benchmarking results are summarized in Table 3- 1 below as well as the compilation summary of this design in Figure 3- 1.

Design		1 Mbytes of data	10 Mbytes of data	100 Mbytes of data
Pure Software	Encr	0.254296s	2.532373s	25.121665s
	Decr	0.282071s	2.809717s	27.870060s
LW AXI and PIOs 50MHz clock	Encr	0.400538s	3.990248s	39.660046s
	Decr	0.400782s	3.987869s	39.620633s
LW AXI and PIOs 100MHz clock	Encr	0.311569s	3.054902s	30.303658s
	Decr	0.306395s	3.052416s	30.252971s

Table 3- 1. Execution time for running the AES-128 algorithm using both the pure software and the hybrid Lightweight AXI and PIOs.

Flow Status	Successful - Fri Jun 18 11:41:26 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	5,826 / 41,910 (14 %)
Total registers	6725
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,816 / 5,662,720 (< 1 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	1 / 4 (25 %)

Figure 3- 1. Compilation summary of Lightweight AXI and PIOs design.

After some research online, we found out that lightweight hps to fpga bridge in combination with PIOs is very slow, so we moved to the faster hps to fpga bridge. Using design of section **2.3.2** we got more reasonable results with average acceleration ratio of 3.3. Results and compilation summary are shown respectively in Table 3- 2 and Figure 3- 2.

Design		1 Mbytes of data	10 Mbytes of data	100 Mbytes of data
Pure Software	Encr	0.254296s	2.532373s	25.121665s
	Decr	0.282071s	2.809717s	27.870060s
Interfacing module 100MHz clock	Encr	0.080443s	0.801840s	7.933851s
	Decr	0.080671s	0.801763s	7.990084s

Table 3- 2. Execution time for running the AES-128 algorithm using both the pure software and the hybrid AXI and on-chip memory with interfacing module.

Flow Status	Successful - Fri Jun 18 14:33:27 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	5,361 / 41,910 (13 %)
Total registers	5677
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	3,328 / 5,662,720 (< 1 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	1 / 4 (25 %)

Figure 3-2. Compilation summary of hybrid system using the Interfacing Module.

Finally, using the design of section **2.3.3** we got the best result of 4.2 over the software implementation. According to this result we can say that this design definitely makes sense in this type of application on this platform. Results and compilation summary are shown in Table 3-3 and Figure 3-3.

Design		1 Mbytes of data	10 Mbytes of data	100 Mbytes of data
Pure Software	Encr	0.254296s	2.532373s	25.121665s
	Decr	0.282071s	2.809717s	27.870060s
IP component	Encr	0.072736s	0.617851s	6.142173s
100MHz clock	Decr	0.064004s	0.633835s	6.315983s

Table 3-3. Execution time for running the AES-128 algorithm using both the pure software and the hybrid system using aes_core as a Quartus IP component.

Flow Status	Successful - Fri Jun 18 17:45:32 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	5,387 / 41,910 (13 %)
Total registers	5718
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,816 / 5,662,720 (< 1 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	1 / 4 (25 %)

Figure 3- 3. Compilation summary of hybrid system using aes_core as Quartus IP component.

For a more clear representation of the obtained results, we drew a chart summarizing the performance of each implementation (Figure 3- 4).

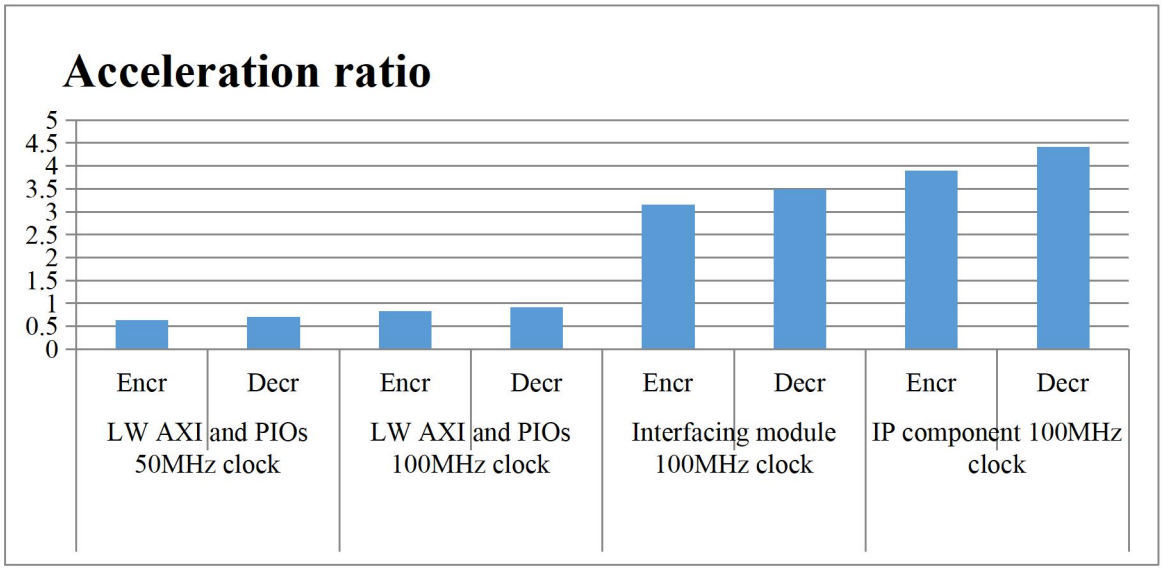


Figure 3- 4. Chart for the acceleration ratio of each design.

It might be noticed that the acceleration ratio for decryption is higher than encryption. This is because of the software implementation being slightly slower in decryption while hardware implementation does encryption and decryption at the same speed. Decryption is slower than encryption in software implementation because of inverse matrix multiplication requiring much more accesses to the multiplication look up table compared to matrix multiplication in encryption.

3.1.2. Video recording and encrypting

The video recording and with the designs that encrypt the video at the end of the recording session we get the same speed as shown in the table above. That was expected since we are just encrypting like any other file. However, we were surprised by the results with the design of a real-time encryption where the hybrid system had little to no effect on the end results. As it only managed to increase one frame per second in hybrid design compared to software design (see Figure 3-5 and Figure 3-6).

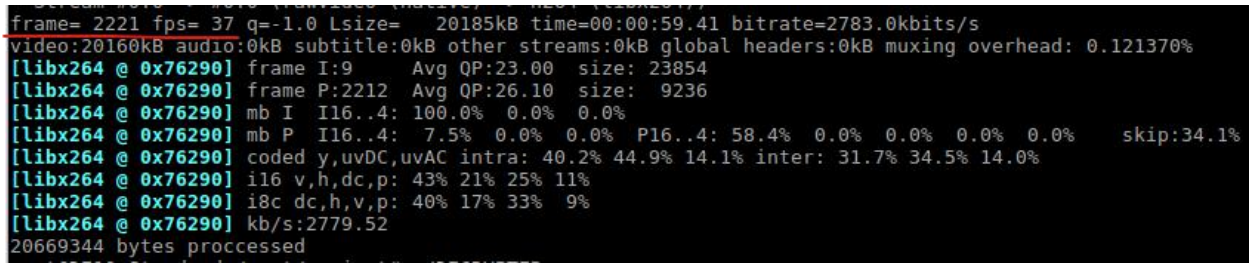


Figure 3-5. Recording from video-port and encrypting in real-time using hybrid implementation.

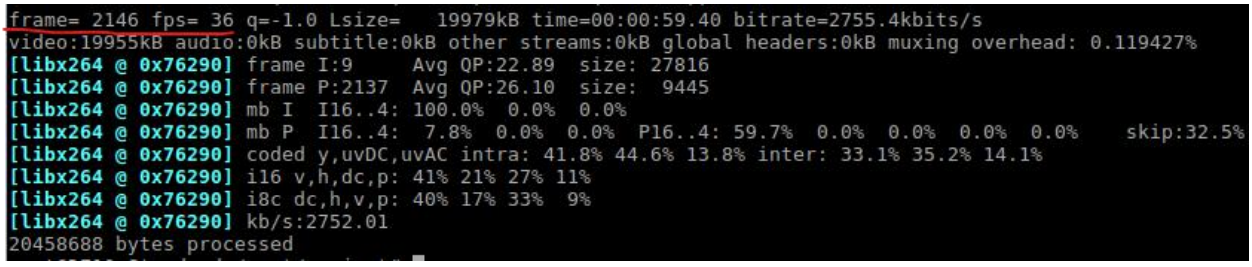


Figure 3-6. Recording from video-port and encrypting in real-time using software implementation.

After some testing, we figured that our quality settings for FFMPEG results in file sizes of around 5MB for 60s video. The results make more sense, because it means we are encrypting 5MB in a time span of 60s which can be done comfortably using software implementation. We tried to increase the video quality but due to the HPS being very limited in performance, the resulting video has way fewer frames per second and lot of stuttering. What we can say is that the hybrid design will have a more noticeable effect on another platform where the HPS is more capable. The compilation summary of this design is shown in Figure 3- 7.

Flow Status	Successful - Fri Jun 18 16:40:45 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	17,818 / 41,910 (43 %)
Total registers	15482
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	3,745,636 / 5,662,720 (66 %)
Total DSP Blocks	2 / 112 (2 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	1 / 4 (25 %)

Figure 3- 7. Compilation summary of video capturing system.

3.2. Chapter summary

In this chapter we have reviewed the results generated by the different implementations of our experiment, where we found that changing the AXI bridge used between HPS and FPGA leads to a significant improvement in the performance of our hybrid system. We also found that implementing the aes_core as a Quartus IP component leads to the best performance. Finally, recording from video-port and encrypting in real-time does not favor the hybrid system by a huge margin since we cannot increase the quality setting of FFMPEG due to the limited performance of HPS.

4 Conclusion and Recommendations

The purpose of this project was to explore, design, and optimize a cryptographic system implementing the AES-128 algorithm using the Terasic DE10 Standard Board. Where our major concern as engineers was minimizing the execution time that the AES crypto algorithm takes to correctly encrypt or decrypt data. In addition, implement a real-time video recording and encrypting application.

First, we dived into the theoretical aspect of cryptography and explored how the AES algorithm works. Next, to further understand every detail about this algorithm, we built it into a program using C language.

Second, we designed different hybrid implementations of the AES algorithm that uses SoC HPS and FPGA. Where we experimented and learned that changing the AXI Bridge from Light-weight to AXI 128 enhanced the performance of our system. Furthermore, introducing the crypto core as a Quartus Intellectual Property Component minimized latency which resulted in the best acceleration of over 4 times faster than the software implementation. This experiment has clearly shown that hybrid systems that merge hardware and software methods gain advantages of simplicity of control and speed of execution. However, the video quality of real-time recording and encrypting/decrypting did not benefit much from the hybrid system compared to the software system.

At the end, this experiment showed very interesting results regarding hardware acceleration of video encryption and our recommendation for future work would be to use our implementation in a web video chat application that allows secure and fast communication over the internet.

Bibliography

[1] "Cryptography," last edited on 28 May 2021. [Online]. Available: https://en.wikipedia.org/wiki/Cryptography. [Accessed june 2021].

[2] G. C. Kessler, "An Overview of Cryptography," [Online]. Available: https://www.garykessler.net/library/crypto.html. [Accessed june 2021].

[3] K. Jash, "Cryptography and its Types," Last Updated : 08 Jan, 2020. [Online]. Available: https://www.geeksforgeeks.org/cryptography-and-its-types/. [Accessed June,2021].

[4] C. Paar and J. Pelzl, Understanding Cryptography: A Textbook for Students and Practitioners, Springer Science & Business Media, 2009.

[5] J. Daemen and V. Rijmen, "AES proposal: The Rijndael Block Cipher," First candidate conference (AeS1), 1999.

[6] J. Ahmad and F. Ahmed, "Efficiency Analysis and Security Evaluation of," *International Journal of Video & Image Processing and Network Security,* vol. 12, no. 04, 2010.

[7] A. Kak, "Finite Fields,Purdue University online course," February 2021. [Online]. Available: https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture7.pdf. [Accessed june 2021].

[8] FIPS-197, "Announcing the advanced encryption standard (AES)," November 2001. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf. [Accessed june 2021].

[9] "What is AES? — Step by Step," medium, February 2019. [Online]. Available: https://zerofruit.medium.com/what-is-aes-step-by-step-fcb2ba41bb20. [Accessed june 2021].

[10] Xilinx, "Field Programmable Gate Array (FPGA)," [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed june 2021].

[11] J. P. Nicolle, "What are FPGAs?," [Online]. Available: https://www.fpga4fun.com/FPGAinfo1.html. [Accessed june 2021].

[12] Terasic.Inc, "DE10-Standard," [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=1081&PartNo=1. [Accessed April 2021].

[13] RocketBoards, "Terasic DE10-Standard Development Kit," January 2019. [Online]. Available: https://rocketboards.org/foswiki/Documentation/DE10Standard. [Accessed April 2021].

[14] Terasic.Inc, "DE10 Standard User Manual," 2018.

[15] Wikipedia, "YCbCr," May 2021. [Online]. Available: https://en.wikipedia.org/wiki/YCbCr. [Accessed June 2021].

[16] "A Primer on Chroma Subsampling and What It Means In Practical Terms," [Online]. Available: https://www.ravepubs.com/chroma-subsampling/. [Accessed May 2021].

[17] "YUV," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/YUV.

[18] AppGeeker, "What is MP4 (MPEG-4 Part 14)," [Online]. Available: https://www.appgeeker.com/how-to/what-is-mp4.html. [Accessed June 2021].

[19] "ffmpeg Documentation," [Online]. Available: https://www.ffmpeg.org/ffmpeg.html#Description. [Accessed June 2021].

[20] "Video Codecs and Encoding," [Online]. Available: https://www.wowza.com/blog/video-codecs-encoding. [Accessed May 2021].

[21] L. E. Bassham III, The Advanced Encryption Standard, National Institute of Standards and Technology, 2002.

[22] "AES128," OpenCores, March 2014. [Online]. Available: https://opencores.org/projects/aes_crypto_core. [Accessed April 2021].

[23] RocketBoards, "Embedded Linux Beginners Guide," [Online]. Available: https://rocketboards.org/foswiki/Documentation/EmbeddedLinuxBeginnerSGuide. [Accessed April 2021].

[24] Terasic.Inc, "DE10-Standard CD-ROM," 2019.

[25] "What is fwrite?," educative, [Online]. Available: https://www.educative.io/edpresso/what-is-fwrite. [Accessed May 2021].

[26] "pipe() System call," geeksforgeeks, [Online]. Available: https://www.geeksforgeeks.org/pipe-system-call/. [Accessed May 2021].

[27] "popen - initiate pipe streams to or from a process," The IEEE and The Open Group, 2004. [Online]. Available: https://pubs.opengroup.org/onlinepubs/009695399/functions/popen.html. [Accessed April 2021].

[28] "Screen tearing," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Screen_tearing. [Accessed June 2021].

[29] "Find execution time of a C program," Techie Delight, [Online]. Available: https://www.techiedelight.com/find-execution-time-c-program.