Advanced Encryption Standard Implementation on Field Programmable Gate Arrays

by

Maryam Behrouzinekoo

B.Eng., University of Guilan, 2011

A Report Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Engineering

in the Department of Electrical and Computer Engineering

University of Victoria

Advanced Encryption Standard Implementation on Field Programmable Gate Arrays

by

Maryam Behrouzinekoo

B.Eng., University of Guilan, 2011

Supervisory Committee

Dr. T. Aaron Gulliver, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Samer Moein, Departmental Member

(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. T. Aaron Gulliver, Supervisor

(Department of Electrical and Computer Engineering)

Dr. Samer Moein, Departmental Member

(Department of Electrical and Computer Engineering)

ABSTRACT

Cryptography provides users with secure communications and data transmission privacy and authenticity (Coron, 2006). Today the most widely used algorithm for private key encryption is the Advanced Encryption Standard (AES). It operates on 128 bit blocks of data in the form of a 4×4 matrix of bytes called the state matrix. The encryption/decryption process is performed on this matrix using key sizes of 128, 192 or 256 bits. The AES round operations include shift rows, mix columns, and sub bytes using finite field arithmetic. Numerous studies have been done on the AES cryptosystem focusing on design optimization in terms of the memory used in hardware implementation (Van Dyken & Delgado-Frias, 2010). The sub bytes operations dominates the hardware complexity of AES due to its non linearity. In this report, the AES hardware feasibility is improved by implementing the sub bytes operation using inversion in $GF(2^8)$. This inversion is decomposed into a network of logic gates which reduces the required read only memory (ROM) by 89% compared to using look up tables.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor, Dr. T. Aaron Gulliver, for his guidance, thoughtful insights, and support. He is always open and kind in communicating with his students.

My Parents, for moral support and the amazing chances they have provided me during my time in Victoria.

The University of Victoria, for an extraordinarily supportive, diverse, and peaceful environment.

Chapter 1

Introduction

With the technological evolution of computer systems, security and confidentiality have become important concerns to avoid theft and fraud. Cryptography plays a vital role in providing privacy, authentication and integrity protection (Gaj & Chodowiec, 2009). There are two types of cryptographic systems, symmetric and asymmetric. Asymmetric cryptography, or public key cryptography, uses a pair of keys consisting of a private key and a public key. In an asymmetric encryption system, a third party can encrypt a message using the public key. However, this message can only be decrypted using the private key. On the other hand, symmetric cryptography, or private key cryptography, uses the same cryptographic key for both encryption and decryption. In symmetric key encryption, stream ciphers or block ciphers can be used. The flow charts of encrypting a message of i bits using stream and block ciphers are shown in Figures 1.1 and 1.2, respectively. This shows that stream ciphers encrypt bits individually using the key stream and the message. On the other hand, block ciphers encrypt blocks of message bits at a

time using a key block. In these figures $x_0 x_1 \cdots x_i$ is the message, $y_0 y_1 \cdots y_i$ is the cipher text, and k is the cipher key.

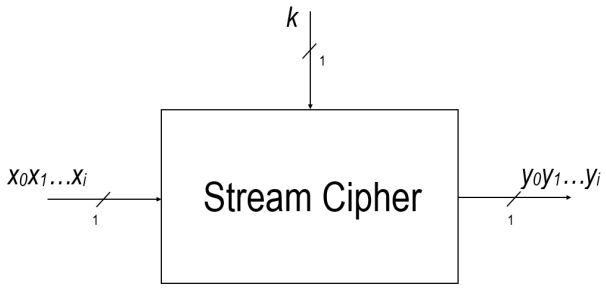


Figure 1.1: Flow chart of encrypting an i bit message using a stream cipher.

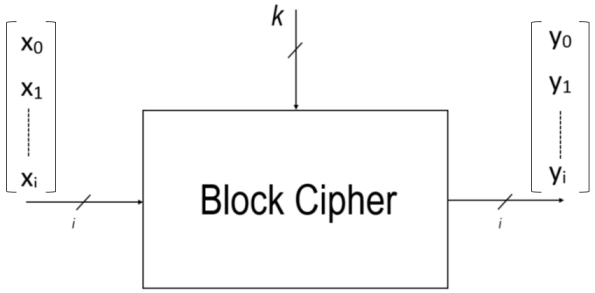


Figure 1.2: Flow chart of encrypting an i bit message using a block cipher.

An example of a symmetric cryptographic algorithm is the Data Encryption Standard (DES), which was developed by IBM in the early 1970s. This algorithm was based on a design by Feistel (Smid & Branstad, 1988). In 1999, the US National Institute of Standards and Technology (NIST) announced that triple DES (3DES) should be used instead of DES. Although 3DES provides higher security than DES, it has the following drawbacks (Paar & Pelzl, 2010).

- It is not very efficient in software implementations.

- 3DES is three times slower than DES.
- The DES key size is only 56 bits. Due to this small size, it is easily breakable and therefore too weak for many security applications.

As a result, DES was withdrawn as a standard by NIST (Gaj & Chodowiec, 2009) which created a need for a new and more secure design.

On January 2, 1997, NIST initiated the AES contest and formally called for algorithms on September 12, 1997. By June 1998, research groups from all over the world had submitted 15 algorithms to NIST. In August 1999, five of the submitted algorithms were chosen to be considered to become the standard. Security, efficiency in software and hardware, and flexibility were the main criteria used to evaluate the algorithms. The algorithms were required to be block ciphers with a 128 bit block size and have key lengths of 128, 192 and 256 bits (Paar & Pelzl, 2010). In October 2000, NIST announced Rijndael as the winner of the AES contest (Gaj & Chodowiec, 2009). The NIST evaluation of AES provided the first efficient hardware architecture. The first implementation of AES based on Field Programmable Gate Arrays (FPGAs) was provided in 2000 (Dandalis, Prasanna, & Rolim, 2000). The National Security Agency (NSA) developed the first implementation based on Application Specific Integrated Circuits (ASICs) (Ichikawa, Kasuya, & Matsui, 2000).

Since AES was chosen by NIST, researchers have focused on three major design directions.

- The development of high speed architectures resulting in AES implementations

operating at speeds of tens of gigabits per second (Schaumont, Kuo, & Verbauwhede, 2002).

- The development of compact architectures for AES optimized for minimum circuit area. This resulted in architectures with 64, 32, and 8 bit data paths (Satoh, Morioka, Takano, & Munetoh, 2001).

- The optimization of AES operations including logic only implementations of the sub bytes operation and optimization of the mix columns operation (Schaumont et al., 2002).

The simplicity of the AES algorithm facilitates implementation on a wide range of platforms under different constraints (Gaj & Chodowiec, 2009). Software implementations have been developed in programming languages such as C, C++, Java, and assembly. Hardware implementations have been developed in hardware description languages such as VHDL and Verilog HDL for realization using ASICs and FPGAs.

Table 1.1 compares the implementations of cryptographic operations based on ASICs and FPGAs (hardware) and microprocessors (software). This shows that the performance of ASICs and FPGAs is almost the same. ASICs and FPGAs can provide parallel processing and pipelining and can operate on variable word sizes. Parallel processing and piplining in general purpose microprocessors are limited by the number of processor functional units and their internal structures. Moreover, the processor functional units only operate on fixed word sizes. In terms of performance, FPGAs are slower than ASICs due to delays in the reconfiguration circuitry.

	ASICs	FPGAs	Microprocessors
Performance Characteristics			
Parallel Processing	Yes	Yes	Limited
Pipelining	Yes	Yes	Limited
Word Size	Variable	Variable	Fixed
Speed	Very Fast	Fast	Moderately fast
Functionality			
Algorithm Agility	No	Yes	Yes
Tamper Resistance	Strong	Limited	Weak
Access Control to Keys	Strong	Moderate	Weak
Development Process			
Description Languages	VHDL, Verilog HDL	VHDL, Verilog HDL	C, C++, Java, Assembly Language
Design Cycle Long	Long	Moderately long	Short
Design Tools	Very expensive	Moderately expensive	Inexpensive
Maintenance and Upgrades	Very Expensive	Inexpensive	Inexpensive

Table 1.1: Comparison of cryptographic operation implementations on different platforms (Gaj & Chodowiec, 2009).

FPGAs and ASICs are very similar in their development processes. Circuits are described using a hardware description language and then verified by a digital circuit simulator. Unlike ASICs, FPGAs do not require physical design (layout), fabrication, and testing for physical defects which results in a significantly shorter design cycle. ASICs cannot be reconfigured once they are in production. They are customized for a particular application rather than for general purpose use. Therefore, maintenance and upgrades can be costly. The programmable nature of FPGAs provides users with the ability to quickly reprogram them. This allows for fast changes in the function of each building block to obtain a new circuit.

FPGAs are now the most popular reconfigurable devices, and they are suitable for hardware implementation of cryptographic primitives (Gaj & Chodowiec, 2009). FPGAs

consist of Configurable Logic Blocks (CLBs), I/O blocks and reconfigurable interconnections. Figure 1.3 shows the generic architecture and the main building blocks of an FPGA. CLBs are used to implement logic designs. Each CLB consists of flip-flops and multiplexers. I/O blocks are used to control the flow of data between the I/O pins and the internal logic of the device. Reconfigurable interconnections are used to connect blocks on the chip.

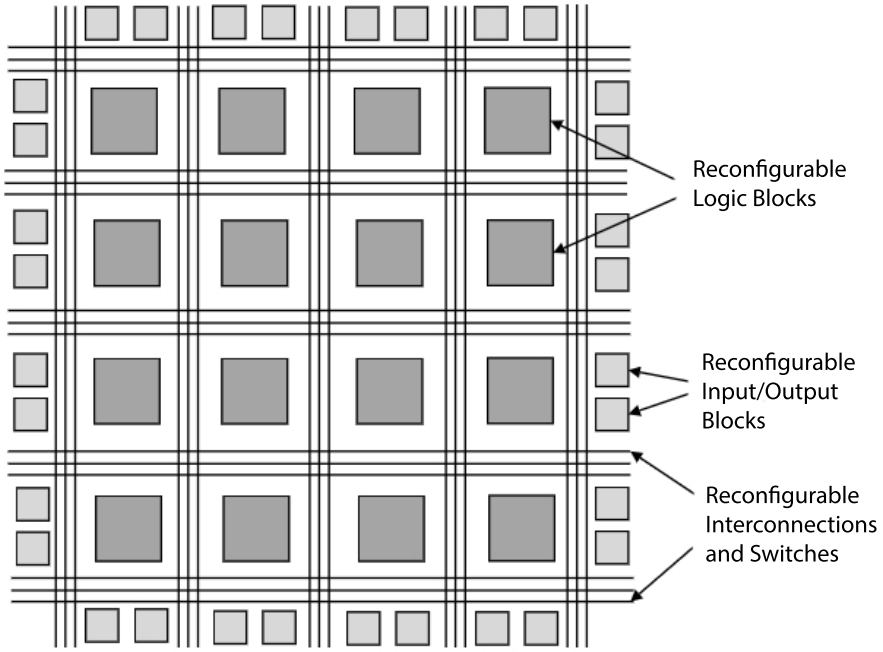


Figure 1.3: Generic FPGA architecture.

In this project, the AES encryption algorithm is implemented on the Spartan 3E device XC3S1200E. This family of FPGAs is specifically designed for cost sensitive applications. The logic per I/O block is increased in Spartan 3E compared to Spartan 3, which reduces the cost per logic cell (Gaj & Chodowiec, 2009).

Chapter 2

Finite Fields

Finite field arithmetic is used in the sub bytes and mix columns AES operations. Therefore, an introduction to finite fields is provided in this chapter.

Definition 2.0.1. Group

A group, G, is a set of elements with an operation $\circ$ which combines two elements of G. A group has the following properties.

1. The group operation $\circ$ is closed. For all $a, b \in G$

   $a \circ b = c \in G$.

2. The group operation $\circ$ is associative. For all $a, b, c \in G$

   $a \circ (b \circ c) = (a \circ b) \circ c$.

3. There is an identity element $e \in G$ such that

   $a \circ e = e \circ a = a$, for all $a \in G$.

4. For each $a \in G$ there is an inverse element $b \in G$ such that

   $a \circ b = b \circ a = e$.

5. The group G is commutative if for all $a, c \in G$

   $a \circ c = c \circ a$.

For example, the set of integers $\mathbb{Z}_m = \{0, 1, \ldots, m-1\}$ with addition modulo m forms a group with identity 0. Note that this set does not form a group with multiplication modulo m, since Condition 4 in Definition 2.0.1 is not met.

Definition 2.0.2. Field

A field, F, is a set of elements with the following properties.

1. All elements of F form an additive group with group operation + and identity 0.

2. All elements of F except 0 form a multiplicative group with group operation $\times$ and identity 1.

3. The distributively law holds so that for all $a, b, c \in F$

   $a \times (b + c) = a \times b + a \times c$.

For example the set of real numbers $\mathbb{R}$ is a field. For all $a \in \mathbb{R}$, there is an additive inverse $-a$, and for all $a \neq 0 \in \mathbb{R}$, there is a multiplicative inverse $1/a = a^{-1}$.

In cryptography, fields with a finite number of elements are considered, which are called finite fields or Galois fields. The number of elements in a field is called the order

of the field. A field with order m only exists if m is a prime power

$$m = p^n,$$

where p is a prime integer called the characteristic of the field and n is a positive integer. $GF(2)$ is a prime field and is the smallest finite field. The two field elements are 0 and 1. The multiplication and addition tables for this field are shown in Tables 2.1 and 2.2, respectively. In Table 2.1 multiplication by 0 is defined as 0. These tables show that addition is equivalent to an XOR gate, and multiplication corresponds to an AND gate.

×	0	1
0	0	0
1	0	1

Table 2.1: $GF(2)$ multiplication.

+	0	1
0	0	1
1	1	0

Table 2.2: $GF(2)$ addition.

2.1 Extension Fields $GF(2^m)$

In AES, the sub bytes table contains 256 elements, so they can be denoted as elements of $GF(2^8)$. The order of this finite field in 256, which is not a prime number. Therefore, the addition and multiplication operations cannot be represented by addition and multiplication of integers modulo 256. Such non prime fields are called extension fields. In

order to perform arithmetic operations in $GF(2^8)$, a polynomial representation of the elements and polynomial arithmetic are used.

The finite field $GF(2^m)$ is an m dimensional vector space over $GF(2)$ and

$$\{x^{m-1}, x^{m-2}, \cdots, x^2, x, 1\}$$

is a basis for $GF(2^m)$ over $GF(2)$. For example, the field $GF(2^8)$ can be represented as polynomials with coefficients in $GF(2)$. These polynomials have degree 7 and can be written as

$$A(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0,$$

where $a_i \in 0, 1$.

2.1.1 Addition in $GF(2^m)$

Addition in $GF(2^m)$ is achieved using polynomial addition. The addition of $A(x), B(x) \in GF(2^m)$ is

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1} c_i x^i, \qquad c_i = a_i + b_i \mod 2$$

As an example, the addition of $A(x) = x^7 + x^6 + x^4 + 1$ and $B(x) = x^4 + x^2 + 1$ is $C(x) = x^7 + x^6 + x^2$ as shown below

$$\begin{array}{rccccccccc} & x^7 & + & x^6 & + & x^4 & + & & & 1 \\ + & & & & & x^4 & + & x^2 & + & 1 \\ \hline & x^7 & + & x^6 & + & & & x^2 & & \end{array}$$

2.1.2 Multiplication in $GF(2^m)$

In the mix columns operation multiplication in $GF(2^8)$ is used. Multiplication in $GF(2^m)$ is similar to multiplication in prime fields. In a prime field two integers are multiplied and the result is taken modulo p. In an extension field, the product of two polynomials is taken modulo an irreducible polynomial. Thus, irreducible polynomials perform the same function as prime numbers in prime fields.

Let $A(x), B(x), P(x) \in GF(2^m)$, where $P(x)$ is an irreducible polynomial. The multiplication of $A(x)$ and $B(x)$ is

$$C(x) = A(x)B(x) \mod P(x).$$

As an example, let $A(x) = x^3 + x^2 + 1$, $B(x) = x^2 + x$, $P(x) = x^4 + x + 1 \in GF(2^4)$. Then the product of $A(x)$ and $B(x)$ is

$$\begin{aligned} C(x) &= A(x)B(x) \mod P(x) \\ &= (x^5 + x^3 + x^2 + x) \mod (x^4 + x + 1) \\ &= x^3. \end{aligned}$$

2.1.3 Inversion in $GF(2^m)$

In AES, the sub bytes operation is based on inversion in $GF(2^8)$. The inverse $A^{-1}(x)$ of a nonzero element $A(x) \in GF(2^m)$ is defined according to

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

where $p(x)$ is an irreducible polynomial. Table 2.3 gives the inverses of the elements in $GF(2^8)$ where $P(x) = x^8 + x^4 + x^3 + x + 1$. The polynomial coefficients are given in hexadecimal notation. For example, $D(x) = x^7 + x^6 + x^4 + x^2 + x + 1$ is $D7$ in hexadecimal. As mentioned previously the element 0 does not have an inverse. However, the sub bytes table requires an inverse for all 256 field values so the inverse of 0 is defined as 0.

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

Table 2.3: $GF(2^8)$ multiplicative inverse table.

Chapter 3

Advanced Encryption Standard

The Advanced Encryption Standard (AES) is the encryption algorithm chosen by the National Institute of Standards and Technology (NIST). Thus, AES is one of the most popular and widely used algorithms for symmetric key cryptography which means that the same key is used for both encryption and decryption (Gaj & Chodowiec, 2009). It is currently used in applications such as e-commerce and financial transactions. The AES block cipher is used in industry standards and commercial systems such as the internet security standard IPsec, the Wi-Fi encryption standard IEEE 802.11i, and the internet phone Skype (Paar & Pelzl, 2010).

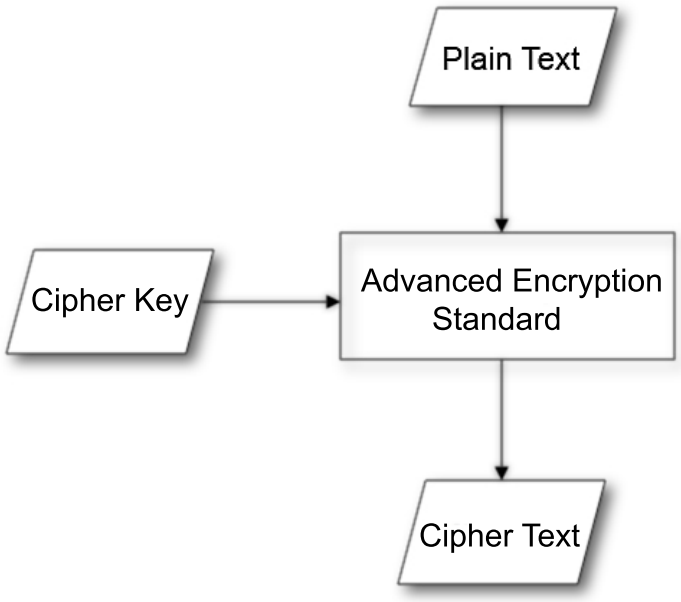


Figure 3.1: The Advanced Encryption Standard (AES) block diagram.

The AES block diagram is shown in Figure 3.1. Plain text is the message before encryption. Encryption transforms the plain text to cipher text using a cipher key. AES is an iterative cipher which operates on 128 bit plain text as input with key sizes of 128,192 or 256 bits. The number of iterations depends on the key size. Before the encryption process begins the plain text is arranged in a state matrix. This representation of the 128 bit plain text is shown in Figure 3.2, where the $S_{i,j}$ denote bytes.

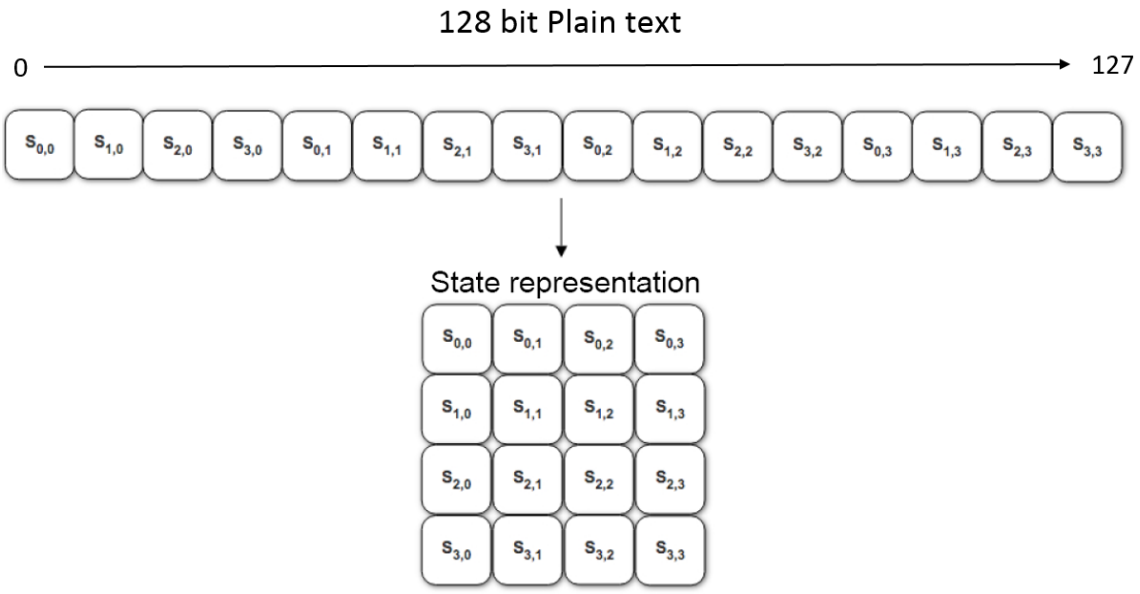


Figure 3.2: State matrix representation of 128 bit plain text.

The AES encryption algorithm uses rounds composed of the following byte operations.

- sub bytes
- shift rows
- mix columns
- add round key

AES employs the inverse of these operations for decryption. These inverse operations are called inverse sub bytes, inverse shift rows, inverse mix columns, and inverse add round key. Add round key is equivalent to bitwise XOR and so is the same as its inverse. Depending on the key size, a specified number of rounds are executed. For key sizes of 128, 192, and 256 bits, 10, 12, and 14 rounds are executed, respectively. Figure 3.3 presents the AES flow chart.

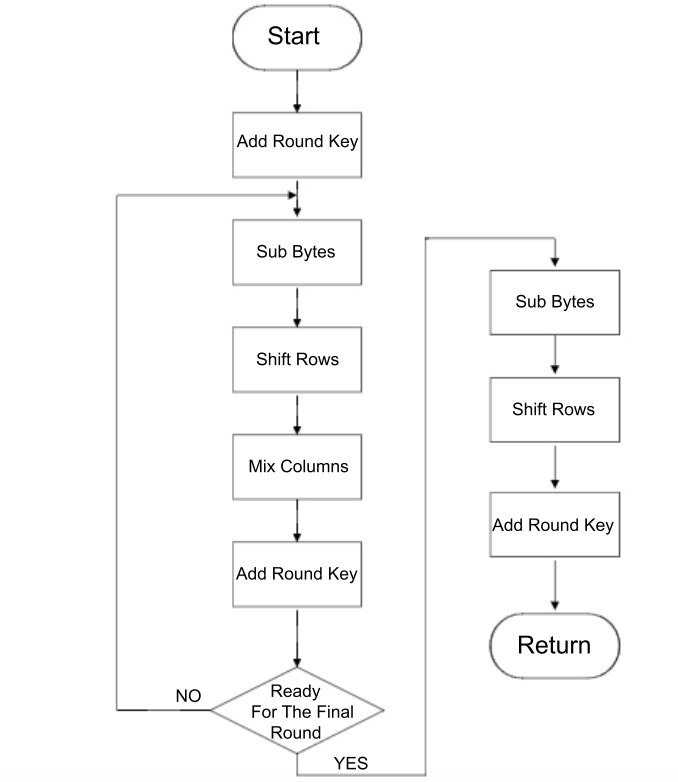


Figure 3.3: The AES algorithm flow chart.

In this project, a key size of 128 bits was chosen since it provides sufficient security for most applications (*128-Bit Versus 256-Bit AES Encryption.*, 2017) and the operations are the same for all key sizes. Furthermore, only the encryption is implemented because AES is a symmetric block cipher so decryption is just the inverse of encryption. As mentioned above, 10 rounds of byte operations need to be executed. According to the AES flow chart, the initial round includes add round key, followed by 9 main rounds of sub bytes, shift rows, mix columns, and add round key. The final round includes all the round operations except mix columns. Excluding the mix columns operation from the final round makes the encryption and decryption symmetric. In the following sections, each operation is discussed in detail.

3.1 Sub Bytes

The sub bytes operation determines the hardware complexity of AES. Due to its non-linearity, this structure is considered to be the most complex and expensive part of the algorithm. The sub bytes operation used in AES is derived from the multiplicative inverse operation over $GF(2^8)$, which is known to have good non-linearity (Van Dyken & Delgado-Frias, 2010). Figure 3.4 shows this operation. In this operation, each byte from the state matrix is substituted by a byte from the AES substitution box given in Table 3.1, which contains all 256 possible values in hexadecimal. As an example, $3d$ from the state matrix is substituted by the element at the intersection of row 3 and column d which is 27.

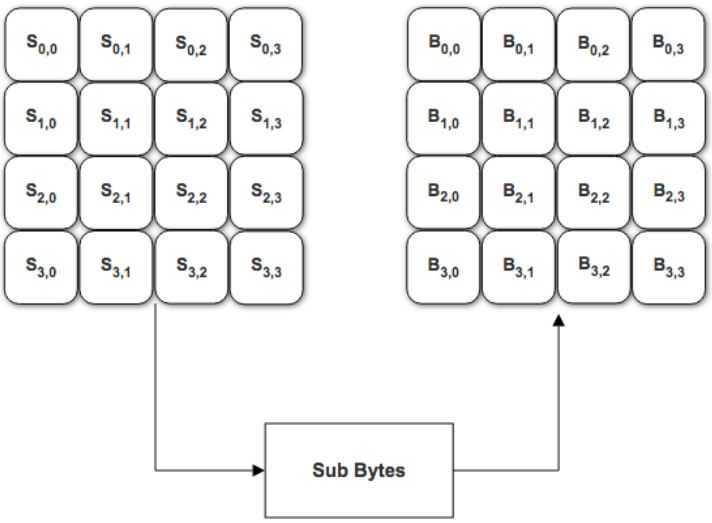


Figure 3.4: The AES sub bytes operation.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
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	2b	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

Table 3.1: AES sub bytes hexadecimal table.

3.2 Shift Rows

Shift rows is an operation in which the blocks in the rows of the state matrix are shifted by an offset as shown Figure 3.5. The first row is unchanged, and each block in the second row is shifted 1 byte to the left. The bytes in the third and fourth rows are shifted 2 and 3 bytes to the left, respectively.

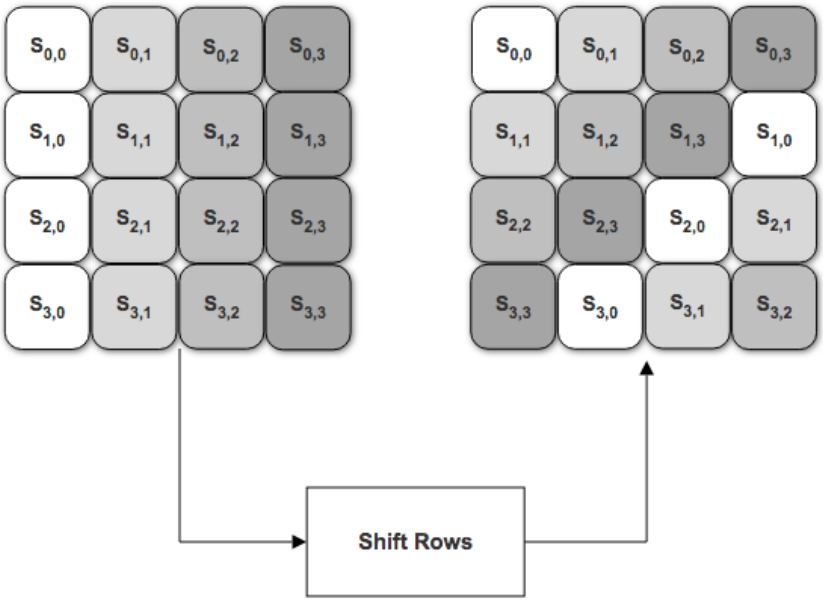


Figure 3.5: The AES shift rows operation.

3.3 Mix Columns

Mix columns operates on each column of the state matrix individually. Each column is mixed using Galois field multiplication. Figure 3.6 shows the mix columns operation.

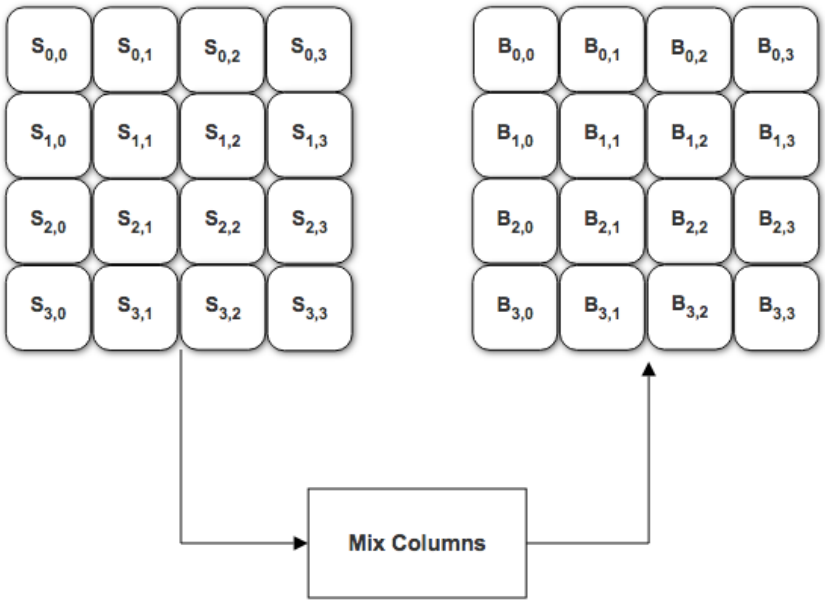


Figure 3.6: The AES mix columns operation.

According to (Gaj & Chodowiec, 2009), the mix columns operation is defined as

$$\begin{bmatrix} B_{0,0} \\ B_{1,0} \\ B_{2,0} \\ B_{3,0} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_{0,0} \\ S_{1,0} \\ S_{2,0} \\ S_{3,0} \end{bmatrix} \tag{3.1}$$

The other columns of output bytes are computed by multiplying the input bytes by the matrix (3.1). From a cryptographic point of view, the mix columns operation mixes bytes across the state matrix and creates a strong dependence between the input bytes and output bytes (Gaj & Chodowiec, 2009). If $[S_{0,0}, S_{1,0}, S_{2,0}, S_{3,0}]$ is $[d4, 25, 5d, 30]$, the first output column is given by

$$\begin{bmatrix} B_{0,0} \\ B_{1,0} \\ B_{2,0} \\ B_{3,0} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} d4 \\ 25 \\ 5d \\ 30 \end{bmatrix}$$

For example $B_{0,0}$ is obtained as follows

$$\begin{aligned}
(02)(d4) &= x(x^7 + x^6 + x^4 + x^2) \\
&= (x^8 + x^7 + x^5 + x^3) \mod (x^8 + x^4 + x^3 + x + 1) \\
&= x^7 + x^5 + x^4 + x + 1 \\
(03)(25) &= (x+1)(x^5 + x^2 + 1) \\
&= (x^6 + x^3 + x) + (x^5 + x^2 + 1) \\
&= x^6 + x^5 + x^3 + x^2 + x + 1 \\
(01)(5d) &= 1(x^6 + x^4 + x^3 + x^2 + 1) \\
&= x^6 + x^4 + x^3 + x^2 + 1 \\
(01)(30) &= 1(x^5 + x^4) \\
&= x^5 + x^4
\end{aligned}$$

Then these are added as shown in Table 3.2. Multiplication by 01, 02 and 03 is as follows. Multiplication by 01 is multiplication by the identity. Multiplication by 02 is multiplication by x, which is a left shift by one bit, followed by a modular reduction with $P(x) = x^8 + x^4 + x^3 + x + 1$. Multiplication by 03 is multiplication by $x + 1$ followed by

$(01)(30) =$			x^5+	x^4+				
$(01)(5d) =$		x^6+		x^4+	x^3+	x^2+		1
$(02)(d4) =$	x^7+		x^5+	x^4+			$x+$	1
$(03)(25) =$		x^6+	x^5+		x^3+	x^2+	$x+$	1
$B_{0,0} =$	x^7+		x^5+	x^4+				1

Table 3.2: Calculation of $B_{0,0}$.

modular reduction with $P(x)$.

3.4 Add Round Key

The output of mix columns is XORed with the round key derived from the user key. This operation is symmetric for encryption and decryption. AES executes the operations discussed in the previous sections as follows.

- Initial round of add round key using round key 0.
- Nine rounds of sub bytes, shift rows, mix columns, and add round key using round keys 1 to 9.
- A final round of sub bytes, shift rows, and add round key using round key 10.

3.5 Summary

Numerous studies have been done on the hardware design of AES cryptosystems in order to achieve an optimized structure in terms of the chip area, memory and delay (Van Dyken & Delgado-Frias, 2010). In the next chapter, the sub bytes operation is implemented using two different approaches. The first approach uses a look up table

which requires a large amount of memory (Gaj & Chodowiec, 2009). Then this memory is reduced by implementing sub bytes using the logic structure of inversion in $GF(2^8)$. Shift rows, mix columns, and add round key are implemented in Sections 4.3, 4.4, and 4.5, respectively.

Chapter 4

AES Implementation on FPGAs

In this project, two different AES designs are implemented on FPGAs and discussed in detail. The first design employs a single ROM containing 16 identical 8 bit look up tables to perform the 128 bit sub bytes operation in parallel. In the second design, sub bytes is implemented using logic gates. As shown in Figure 4.1, the initial round of the process employs add round key with the state matrix XORed with key 0. The following nine rounds employ sub bytes, shift rows, mix columns, and add round key using keys 1 to 9. The final round executes all of the operations except mix columns and the output of this round is the final cipher text. In the following sections, the implementation of each operation is discussed.

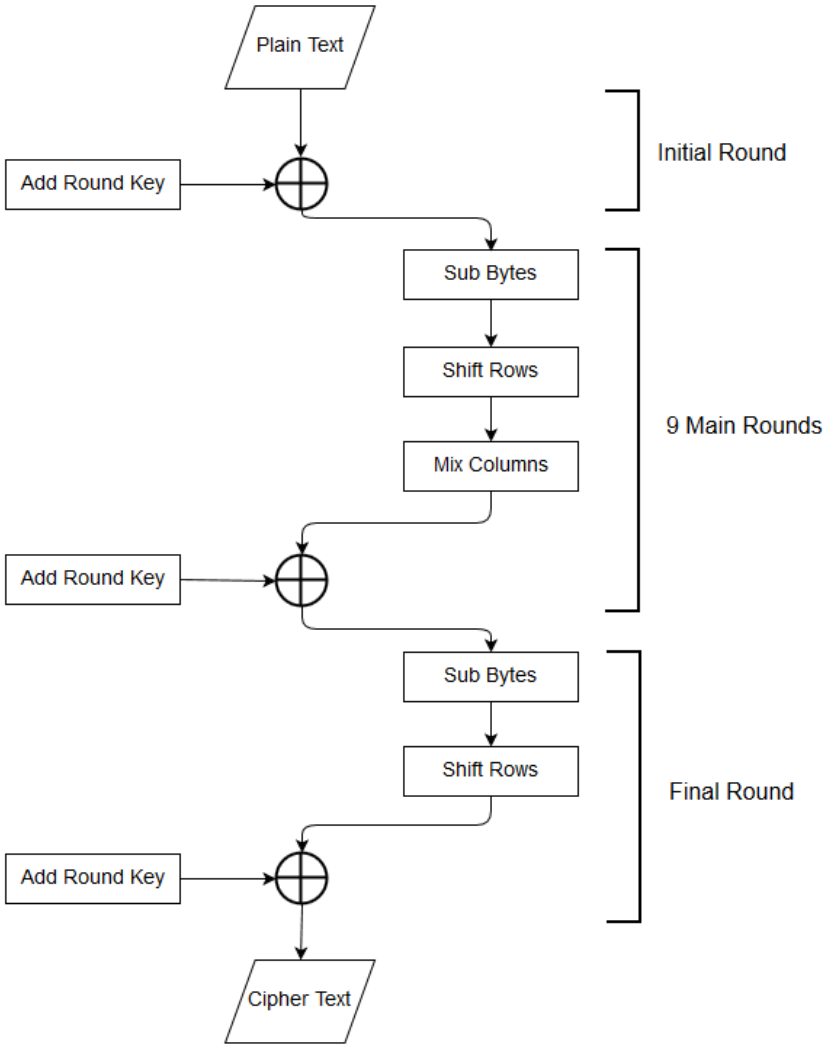


Figure 4.1: The AES encryption flowchart.

4.1 Sub Bytes

Sub bytes is the only nonlinear AES operation. It is a bijective mapping in which each byte of the state matrix is mapped to a byte of the sub byte table. Therefore, each of the $2^8 = 256$ possible input elements is one-to-one mapped to an output element so the inverse of sub bytes required for decryption can be obtained.

4.1.1 Look Up Table

In this design, sub bytes is a single ROM including a 128 bit look up table (LUT) as shown in Figure 4.2. This LUT is implemented using a ROM. Each input byte X_i is connected to an 8 bit LUT, and Y_i is obtained at the output. This architecture includes 16 identical 8×8 look up tables in parallel that can be implemented using a 256×8 look up table containing all possible values for the sub bytes operation. Implementing the sub bytes operation using look up tables leads to a smaller delay compared to logic only implementation. This also reduces the hardware complexity, but requires a large amount of memory. The logic structure of sub bytes can be used to reduce the memory required to implement this operation.

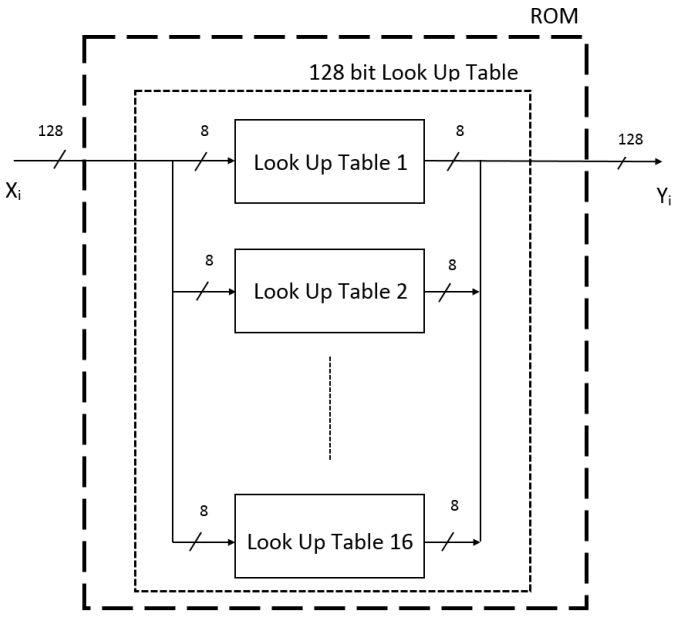


Figure 4.2: Look up table sub bytes architecture.

4.1.2 Logic

The logic only design of sub bytes is based on inversion in $GF(2^8)$ as mentioned in Section 2.1.3. The architecture of 128 bit logic only sub bytes includes 16 identical 8 bit sub bytes operations as shown in Figure 4.4. Each consists of the following steps.

- $GF(2^8)$ to $GF(((2^2)^2)^2)$ converter.
- $GF(((2^2)^2)^2)$ inversion.
- $GF(((2^2)^2)^2)$ to $GF(2^8)$ converter.
- Affine transformation.

From (Satoh et al., 2001), inversion in $GF(2^8)$ can be implemented in $GF(((2^2)^2)^2)$, which is a field extension of degree 2 over $GF((2^2)^2)$. Similarly, $GF((2^2)^2)$ is a field extension of degree 2 over $GF(2^2)$, and finally $GF(2^2)$ is a field extension of degree 2 over $GF(2)$. Therefore, inversion in $GF(2^8)$ is decomposed into operations in $GF(2)$. The computation of inversion in $GF(2^8)$ is shown in Figure 4.3. As shown in Figure 4.4, the first step

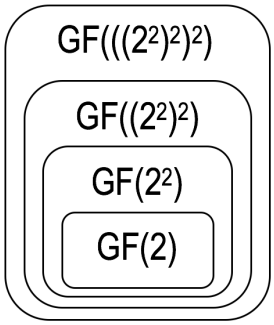


Figure 4.3: Inversion computation in $GF(2^8)$.

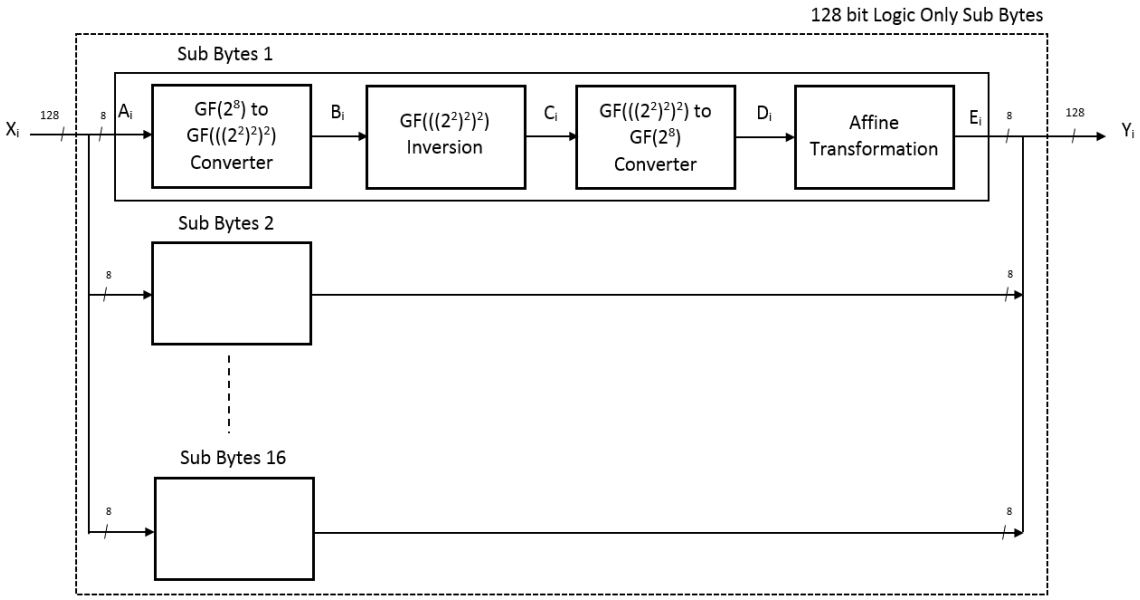


Figure 4.4: Logic only sub bytes architecture.

is conversion from $GF(2^8)$ to $GF(((2^2)^2)^2)$. This transforms a representation in $GF(2^8)$ to a representation in $GF(((2^2)^2)^2)$. In (Satoh et al., 2001), this transformation is given by

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \\ B_4 \\ B_5 \\ B_6 \\ B_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \end{bmatrix}$$

where A_0 to A_7 are the inputs of the $GF(2^8)$ to $GF(((2^2)^2)^2)$ converter and B_0 to B_7 are the outputs. After this transformation, $GF(((2^2)^2)^2)$ inversion is done. As mentioned above, this inversion is decomposed into operations in $GF(2)$. This inversion algorithm is based on Fermat's Little Theorem (FLT) (Satoh et al., 2001).

Inversion in $GF(((2^2)^2)^2)$ can be decomposed into a sequence of operations in $GF((2^2)^2)$ as shown in Figure 4.5(a). This inversion includes three $GF((2^2)^2)$ multipliers, one $GF((2^2)^2)$ squarer and scaler, two adders (XOR gates), and one $GF((2^2)^2)$ inversion. Figure 4.5(b) shows inversion in $GF((2^2)^2)$ which can be expressed in terms of operations in $GF(2^2)$. The components include three $GF(2^2)$ multipliers, one $GF(2^2)$ squarer and scaler, two adders (XOR gates), and one $GF(2^2)$ inversion. Inversion in $GF(2^2)$ is shown in Fig-

ure 4.5(c) and does not involve any logic.

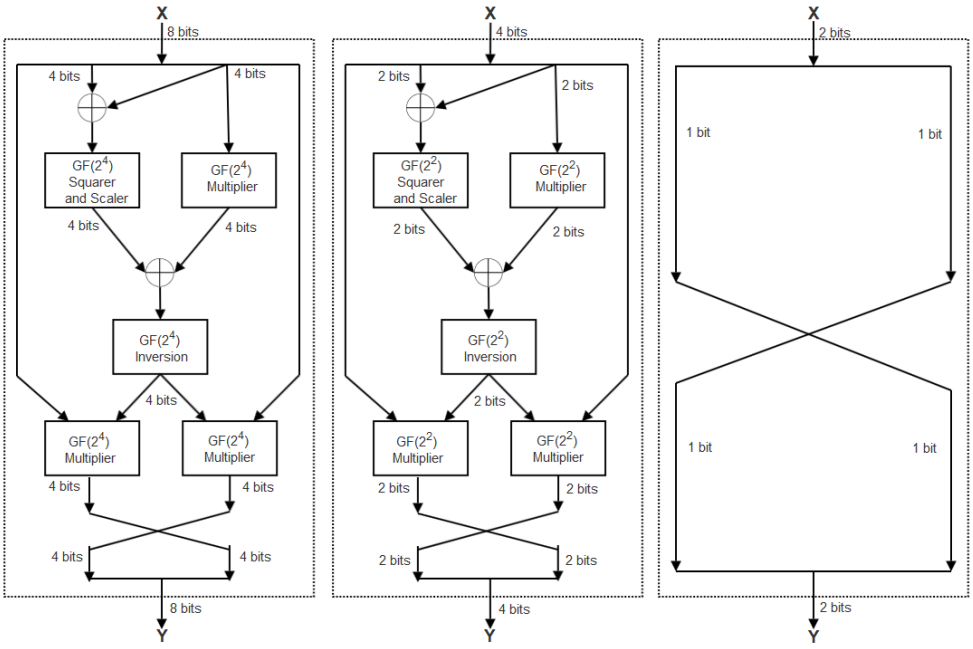


Figure 4.5: (a) $GF(2^8)$ inversion, (b) $GF(2^4)$ inversion, and (c) $GF(2^2)$ inversion.

Figure 4.6(a) shows the $GF((2^2)^2)$ squarer and scaler, which includes two $GF(2^2)$ squarers, one $GF(2^2)$ scaler, and one adder (XOR gate). Figure 4.6(b) shows the $GF((2^2)^2)$ multiplier, which includes two $GF(2^2)$ multipliers, one $GF(2^2)$ multiplier and scaler, and four $GF(2^2)$ adders (XOR gates). The $GF(2^2)$ squarer, $GF(2^2)$ squarer and scaler, and $GF(2^2)$ scaler are shown in Figures 4.7(a), 4.7(b), and 4.7(c), respectively.

Figure 4.8(a) shows the $GF(2^2)$ multiplier, which has the same structure as the $GF((2^2)^2)$ multiplier, but the two multipliers and the multiplier and scaler in $GF(2^2)$ are replaced with AND gates. Figure 4.8(b) shows the $GF(2^2)$ multiplier and scaler, which includes three $GF(2)$ multipliers and four $GF(2)$ adders. The operations in $GF(2)$ can be implemented using XOR gates (addition) and AND gates (multiplication). Thus, inversion in $GF(((2^2)^2)^2)$ can be decomposed into a logic circuit composed of only XOR and AND

gates. As shown in Figure 4.4, inversion in $GF(((2^2)^2)^2)$ is followed by a $GF(((2^2)^2)^2)$ to

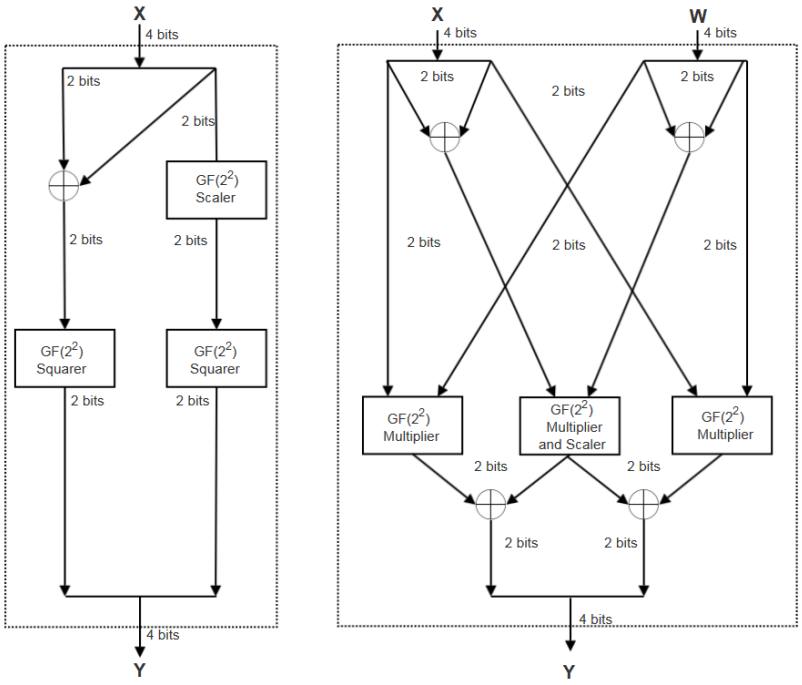


Figure 4.6: (a) $GF(2^4)$ squarer and scaler, and (b) $GF(2^4)$ multiplier.

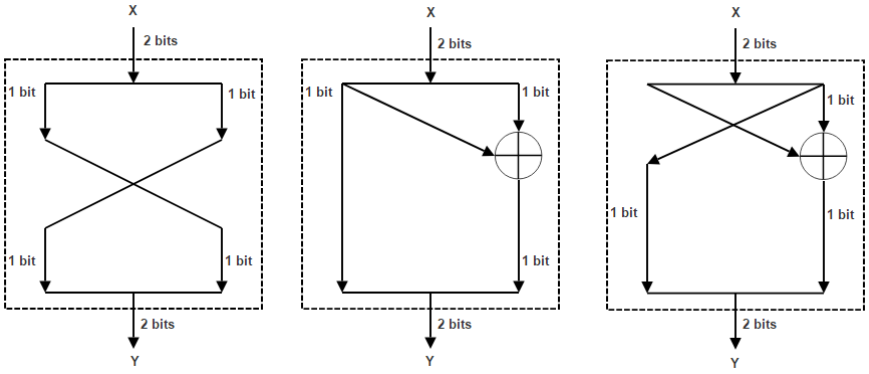


Figure 4.7: (a) $GF(2^2)$ squarer, (b) $GF(2^2)$ squarer and scaler, and (c) $GF(2^2)$ scaler.

$GF(2^8)$ converter. This converter can be written as

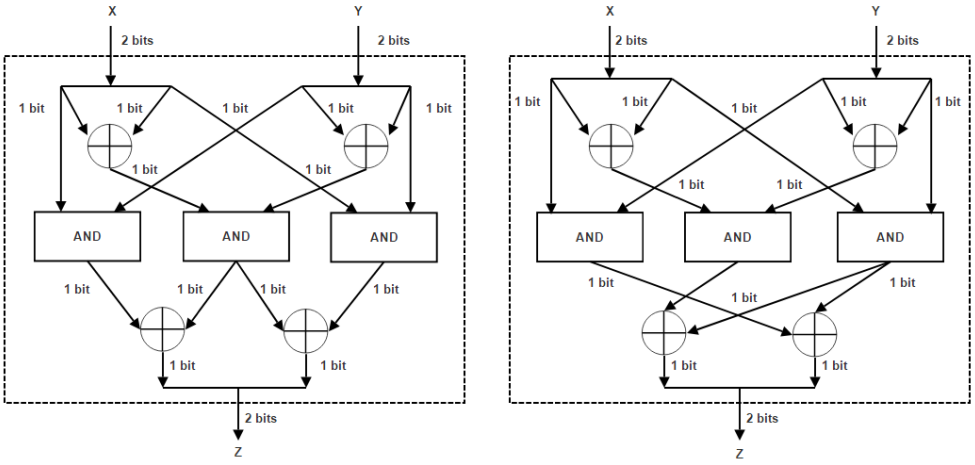


Figure 4.8: (a) $GF(2^2)$ multiplier, and (b) $GF(2^2)$ multiplier and scaler.

$$\begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \\ D_6 \\ D_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \end{bmatrix}$$

where C_0 to C_7 are the inputs and D_0 to D_7 are the outputs. According to (Daemen & Rijmen, 2002), the last step of the 8 bit sub bytes operation is an affine transformation, which can be used to thwart attacks. This transformation does not change the non linear property of the sub bytes operation. Although inversion in $GF(2^8)$ is a simple algebraic operation, sub bytes is a complex algebraic operation when inversion in $GF(2^8)$ is combined with the affine transformation. In (Daemen & Rijmen, 2002), this transformation is given by

$$\begin{bmatrix} E_0 \\ E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \\ E_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \\ D_6 \\ D_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

where D_0 to D_7 are the inputs and E_0 to E_7 are the outputs.

4.2 Mix Columns

Mix columns operates on each column individually. Figure 4.9 shows the implementation of the 128 bit mix columns operation, which includes four 32 bit mix columns operations in parallel.

4.3 Add Round Key

In each round, the mix columns output is XORed with the corresponding key. For example in round 5, key 5 is XORed with the output of the previous round. This step is implemented using XOR gates.

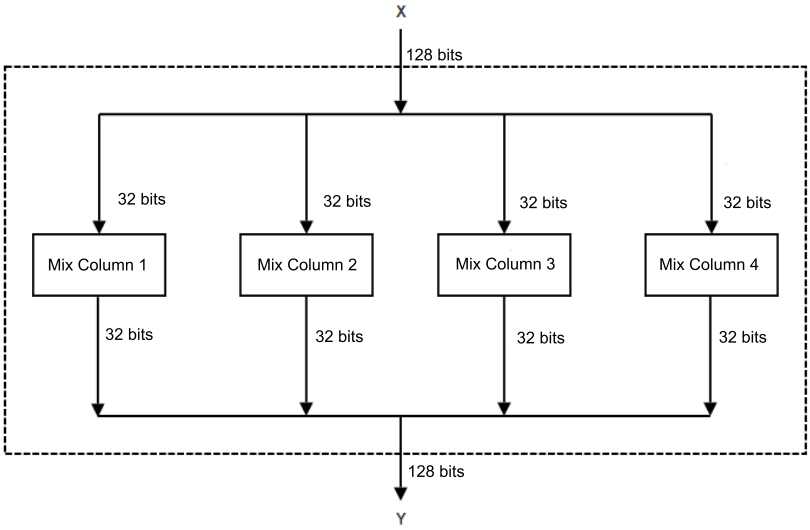


Figure 4.9: The mix columns architecture for 128 bits.

4.4 Key Scheduling

For a 128 bit plain text, 10 keys are required. Table 4.1 shows the keys used in the algorithm with their addresses. Note that keys 1 to 9 are used for the nine main rounds (including sub bytes, shift rows, mix columns, and add round key). Key 0 is used for the initial round, and key 10 is used for the final round (including sub bytes, shift rows, and add round key) . A 4 bit address is used to retrieve the keys when required by the encryption process. These keys are stored in a Read only Memory (ROM), and then read when called by the control circuit.

Round	Key	Address
2	b692cf0b643dbdf1be9bc5006830b3fe	0000
3	b6ff744ed2c2c9bf6c590cbf0469bf41	0001
4	47f7f7bc95353e03f96c32bcfd058dfd	0010
5	3caaa3e8a99f9deb50f3af57adf622aa	0011
6	5e390f7df7a69296a7553dc10aa31f6b	0100
7	14f9701ae35fe28c440adf4d4ea9c026	0101
8	47438735a41c65b9e016baf4aebf7ad2	0110
9	549932d1f08557681093ed9cbe2c974e	0111
10	13111d7fe3944a17f307a78b4d2b30c5	0111
Not Used	00000000000000000000000000000000	1001
Not Used	00000000000000000000000000000000	1010
Not Used	00000000000000000000000000000000	1011
Not Used	00000000000000000000000000000000	1100
Not Used	00000000000000000000000000000000	1101
1	d6aa74fdd2af72fadaa678f1d6ab76fe	1110
0	000102030405060708090a0b0c0d0e0f	1111

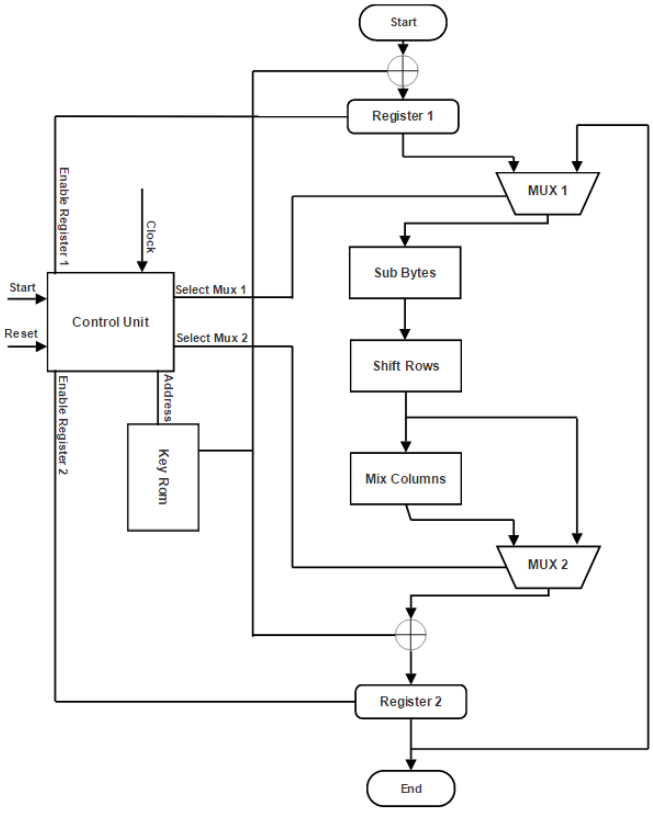
Table 4.1: AES key ROM.

4.5 AES Data Path and Control Circuit

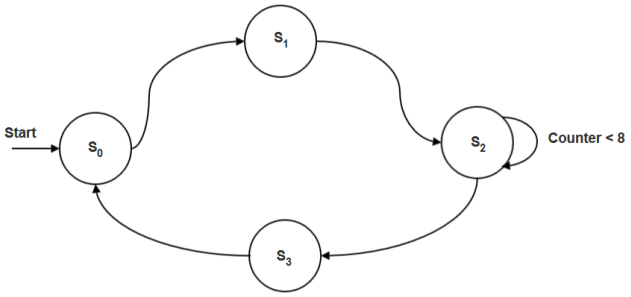
The AES data path includes all the steps of the encryption process in addition to the control components such as multiplexers and registers. The control circuit synchronizes the encryption process by controlling the information flow. The data path and control circuit for AES encryption are shown in Figure 4.10. The control circuit in Figure 4.10(b) is a finite state machine (FSM). This FSM has four states, which are specified in Table 4.2. The output from each state is the input to the next state. In the initial state, S_0, the state matrix is added to the initial key 0 using the first add round key and the output is stored in register 1. In state S_1 the multiplexer (MUX 1) passes the value in register 1 to be used in the first round using key 1. In state S_2 eight main rounds of operations are performed using keys 2 to 9. In this state, a counter is used to count the eight main rounds of operations and it remains off for the rest of the process. In state S_3 the multiplexer (MUX 2) passes the shift rows output instead of the mix column output in order to only execute sub bytes, shift rows, and add round key using key 10. The final output is the cipher text. Note that in each iteration the output of the second add round key is stored in register 2 to be used in the next iteration.

State	Enable Register 1	Enable Register 2	Select MUX 1	Select MUX 2	Key	Counter
S_0	1	0	0	0	0	Off
S_1	0	1	0	0	1	Off
S_2	0	1	1	0	2,3,4,5,6,7,8,9	0-7
S_3	0	1	1	1	10	Off

Table 4.2: Control unit state specifications.



(a)



(b)

Figure 4.10: (a) Data path and control circuit of the AES algorithm, and (b) AES state diagram.

4.6 Results

The sub bytes operation determines the hardware complexity of AES. This structure is nonlinear and therefore it is the most complex and expensive part of the system (Van Dyken & Delgado-Frias, 2010). In this project, two structures were used to implement sub bytes. The first structure uses a single ROM which includes all 256 possible values for the sub bytes operation. It has sixteen identical 8 bit look up tables which work in parallel. Using look up tables for implementing this operation requires a large amount of memory. However, this can be reduced considerably by implementing the operation using a logic structure of inversion in $GF(2^8)$. With this approach, a network of logic gates is used to obtain a compact architecture for the sub bytes operation. As mentioned in Section 4.1.2, a $GF(2^8)$ to $GF(((2^2)^2)^2)$ converter is used to perform inversion is $GF(2^8)$ as shown in Figure 4.3.

The two sub bytes structures were used to implement AES with a 128 bit key on an FPGA. The Spartan-3E family was chosen for the implementation as this family of FPGAs offers a lower cost per logic than other FPGA families. The implementation was done using Xilinx ISE, which is a software tool produced by Xilinx for the synthesis and analysis of HDL designs. In this project, the plain text of 00112233445566778899aabbccddeeff was used as the input. Key 0 is 000102030405060708090a0b0c0d0e0f and the corresponding cipher text is 69c4e0d86a7b0430d8cdb78070b4c55a.

Figure 4.11 shows the outputs after each round operation. Figure 4.11(a) is the initial round operation. In this round, the plain text is XORed with key 0. This operation is

executed in state S_0. As shown in Figure 4.11(b), the next round operation include sub bytes, shift rows, mix columns, and add round key using key 1. This round is executed in state S_1. Rounds 3 to 9 are executed in state S_2 as shown in Figures 4.11(c), (d), (e), (f), (g), (h), (i), and (j). Figure 4.11(k) shows round 10, which executes all round operations except mix columns in state S_3. The output from this round is the cipher text.

Table 4.3 presents the required resources for 128 bit AES look up table and 128 bit AES $GF(2^8)$ inversion implementations. This table shows that both designs have the same number of flip flops, four input LUTs and IOBs, which are 258, 134, and 134, respectively. The number of slices used to implement the AES look up table is 1594. This number is reduced to 1186 with AES $GF(2^8)$ inversion. One counter is used in both designs to count the eight main rounds in state S_2. The AES look up table design requires 146 XOR gates, whereas AES $GF(2^8)$ inversion design employs 1666 of XOR gates. Thus, implementing the sub bytes operation based on inversion in $GF(2^8)$ increases the number of XOR gates significantly.

	Look Up Table	$GF(2^8)$ Inversion
Number of slices	1594	1186
Number of IOBs	134	134
Number of 4 input LUTs	134	134
Number of flip flops	258	258
Number counters	1	1
Number of XOR gates	146	1666

Table 4.3: Resources required for AES implementation.

Table 4.4 shows the number of ROMs required to implement AES look up table and AES $GF(2^8)$ inversion. A single 16×128-bit ROM is used in both designs. This ROM is used to store the plain text. The look up table design for the sub bytes operation employs

Initial round

Plain text			
00	44	88	cc
11	55	99	dd
22	66	aa	ee
33	77	bb	ff

+

Key 0			
00	04	08	0c
01	05	09	0d
02	06	0a	0e
03	07	0b	0f

=

After add round key			
00	40	80	c0
10	50	90	d0
20	60	a0	e0
30	70	b0	f0

(a) The initial round operations at state S_0.

Round 1

After sub bytes			
63	09	cd	ba
ca	53	60	70
b7	d0	e0	e1
04	51	e7	8c

After shift rows			
63	09	cd	ba
53	60	70	ca
e0	e1	b7	d0
8c	04	51	e7

After mix columns			
5f	55	2f	91
72	7f	7b	Db
64	5b	e3	9f
41	c9	b2	91

+

Key 1			
d6	d2	da	d6
aa	af	a6	ab
74	72	78	76
fd	fa	f1	fe

=

After add round key			
89	85	2d	cb
d8	5a	18	12
10	ce	43	8f
e8	68	d8	e4

(b) Round 1 operations at state S_1.

Round 2

After sub bytes			
a7	97	d8	1f
61	be	ad	c9
ca	8b	1a	73
9b	45	61	69

After shift rows			
a7	97	d8	1f
be	ad	c9	61
1a	73	ca	8b
69	9b	45	61

After mix columns			
ff	31	64	77
87	d8	51	3a
96	6a	51	d0
84	51	fa	09

+

Key 2			
b6	64	be	68
92	3d	9b	30
cf	bd	c5	b3
0b	f1	00	fe

=

After add round key			
49	55	da	1f
15	e5	ca	0a
59	d7	94	63
8f	a0	fa	f7

(c) Round 2 operations at state S_2.

Round 3

After sub bytes			
3b	fc	57	c0
59	d9	74	67
cb	0e	22	fb
73	e0	2d	68

After shift rows			
3b	fc	57	c0
d9	74	67	59
22	fb	cb	0e
68	73	e0	2d

After mix columns			
4c	f7	2c	53
9c	71	3f	4d
1e	f0	86	f2
66	76	8e	56

+

Key 3			
b6	d2	6c	04
ff	c2	59	69
74	c9	0c	bf
4e	bf	bf	41

=

After add round key			
fa	25	40	57
63	b3	66	24
6a	39	8a	4d
28	c9	31	17

(d) Round 3 operations at state S_2.

Round 4

After sub bytes				After shift rows				After mix columns				Key 4				After add round key			
2d	3f	09	5b	2d	3f	09	5b	63	fc	97	75	47	95	f9	fd	24	69	6e	88
fb	6d	33	36	6d	33	36	fb	85	53	be	47	f7	35	6c	05	72	66	d2	42
02	12	7e	e3	7e	e3	02	12	b7	8d	47	d6	f7	3e	32	8d	40	b3	75	5b
34	dd	c7	f0	f0	34	dd	c7	9f	f9	8e	91	bc	03	bc	fd	23	fa	32	6c

(e) Round 4 operations at state S_2.

Round 5

After sub bytes				After shift rows				After mix columns				Key 5				After add round key			
36	f9	9f	c4	36	f9	9f	c4	f4	32	75	1d	3c	a9	50	ad	c8	9b	25	b0
40	33	b5	2c	33	b5	2c	40	bc	e5	f1	d0	aa	9f	f3	f6	16	7a	02	26
09	6d	9d	39	9d	39	09	6d	d4	54	d6	3b	a3	9d	af	22	77	c9	79	19
26	2d	23	50	50	26	2d	23	54	d0	c5	3c	e8	eb	57	aa	bc	3b	92	96

(f) Round 5 operations at state S_2.

Round 6

After sub bytes				After shift rows				After mix columns				Key 6				After add round key			
e8	14	3f	e7	e8	14	3f	e7	98	00	6b	8e	5e	f7	a7	0a	c6	f7	cc	84
47	da	77	f7	da	77	f7	47	16	f8	2c	5a	39	a6	55	a3	2f	5e	79	f9
f5	dd	b6	d4	b6	d4	f5	dd	ee	7f	04	d0	0f	92	3d	1f	e1	ed	39	cf
65	e2	4f	90	90	65	e2	4f	74	55	9c	36	7d	96	c1	6b	09	c3	5d	5d

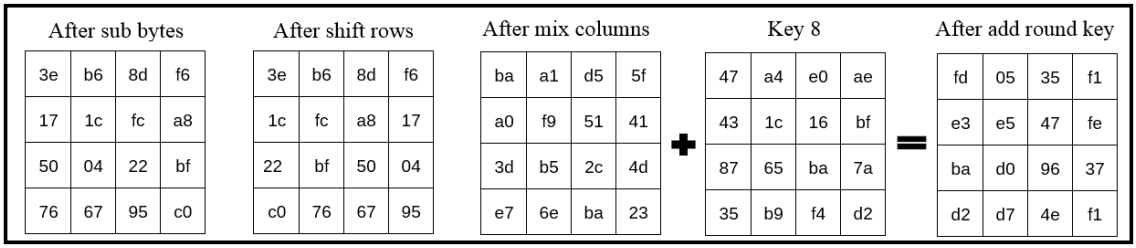
(g) Round 6 operations at state S_2.

Round 7

After sub bytes				After shift rows				After mix columns				Key 7				After add round key			
b4	68	4b	5f	b4	68	4b	5f	c5	9a	f0	98	14	e3	44	4e	d1	79	b4	d6
15	58	b6	99	58	b6	99	15	7e	9b	5f	c6	f9	5f	0a	a9	87	c4	55	6f
f8	55	12	8a	12	8a	f8	55	1c	d2	4b	34	70	e2	df	c0	6c	30	94	f4
01	2e	4c	4c	4c	01	2e	4c	15	86	e0	39	1a	8c	4d	26	0f	0a	ad	1f

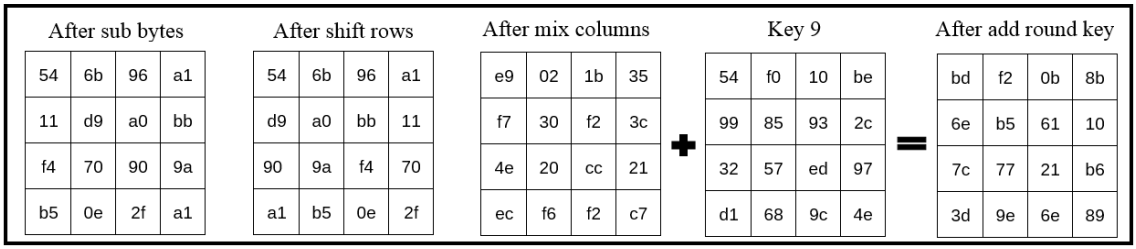
(h) Round 7 operations at state S_2.

Round 8



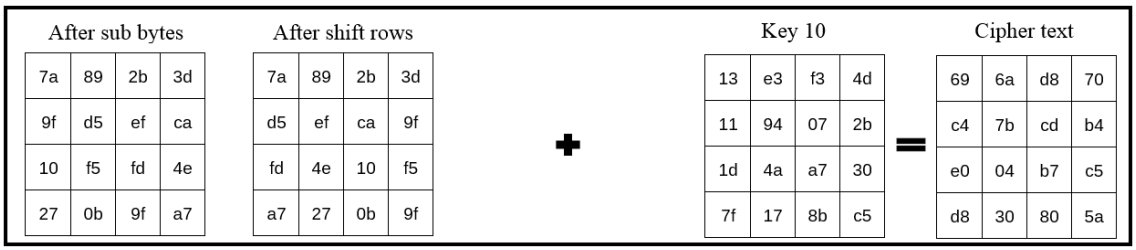
(i) Round 8 operations at state S_2.

Round 9



(j) Round 9 operations at state S_2.

Round 10



(k) Round 10 operations at state S_3.

Figure 4.11: The 128 bit AES round operations.

sixteen 256 × 8-bit ROMs, whereas no ROMs are required for inversion in $GF(2^8)$. One 4 × 128-bit ROM is used in both designs. This ROM is used to store the keys as discussed in Section 4.5. The total number of ROMs used in the look up table and inversion designs is 18 and 2, respectively.

Table 4.5 shows that the look up table design has a total delay of 8.588 ns of which 5.139 ns is logic delay and 3.449 ns is route delay. The inversion design has a total delay of 15.510 ns of which 8.137 ns is logic delay and 7.383 ns is route delay. The look up table design has a maximum frequency of 116.437 MHz, which means that the minimum period for the clock cycle is 8.588 ns. The inversion design has a maximum frequency of 64.437 MHz, which means that the minimum period for the clock cycle is 15.510 ns.

	Look Up Table	$GF(2^8)$ Inversion
16 × 128-bit ROMs	1	1
256 × 8-bit ROMs	16	0
4 × 128-bit ROMs	1	1
Total number of ROMs	18	2

Table 4.4: Number of ROMs required for AES look up table and AES $GF(2^8)$ inversion.

	AES (Look Up Table)	AES ($GF(2^8)$ Inversion)
Logic delay	5.139 ns	8.137 ns
Route delay	3.449 ns	7.383 ns
Total delay	8.588 ns	15.510 ns

Table 4.5: Delay for AES look up table and AES $GF(2^8)$ inversion.

Chapter 5

Conclusion and Future Work

The implementation of an AES hardware design depends on optimization factors and the semiconductor technology. These factors include chip area and power consumption. The chip area required for implementing AES is important as it is the main factor that determines the cost of an integrated circuit. Moreover, factors such as cost, fabrication technology, and power consumption can limit this area. For example, in FPGAs the chip area is limited by the available fabrication technology and the cost of the device. In this project, two 128 bit AES designs were implemented. The sub bytes operation was implemented using look up tables and logic gates as discussed in Sections 4.1.1 and 4.1.2, respectively. Implementing the sub bytes operation using look up tables requires more memory. Figure 5.1 shows the number of ROMs used to implement the look up table and inversion designs.

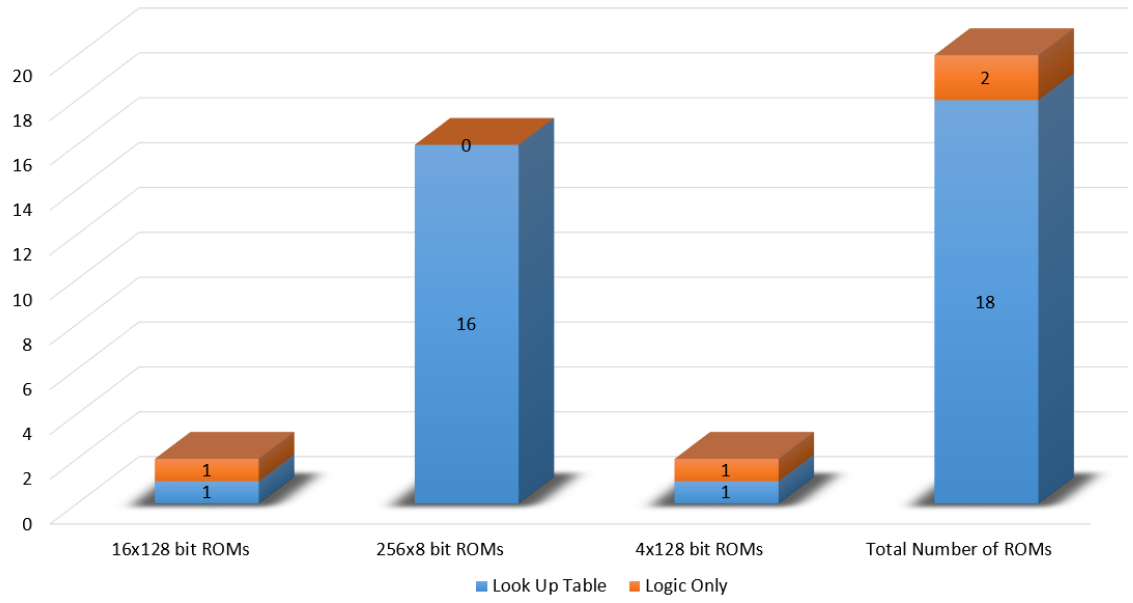
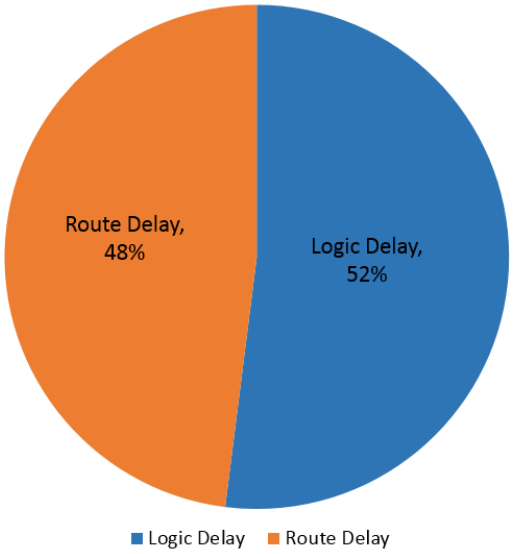


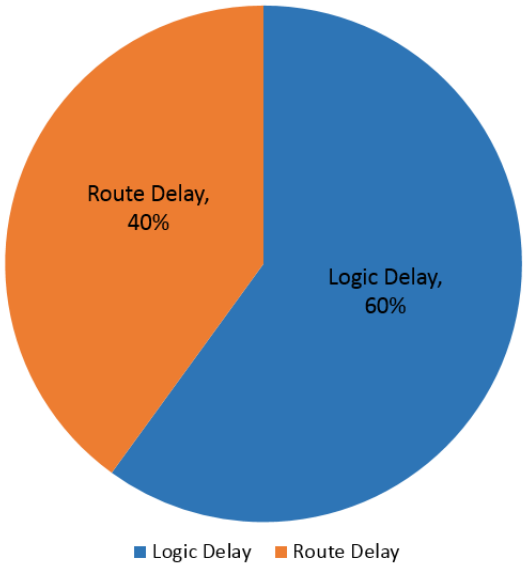
Figure 5.1: Number of ROMs required for the look up table and inversion designs.

This shows that no 256×8 bit ROMs are required for the inversion design whereas sixteen 256×8 bit ROMs are employed in the look up table design. The implementation of sub bytes based on the inversion in $GF(2^8)$ is preferred in ASICs since memory is costly.

Figures 5.2(a) and 5.2(b) show the delay for the inversion and look up table designs, respectively. The total delay when using logic gates for implementing the inversion design is 15.510 ns, 52% logic delay and 48% route delay. The total delay using look up tables to implement sub bytes in the design is 7.93 ns. This indicates that the look up table design should be considered for faster applications, since it leads to a higher clock frequency of 116 MHz compared to the inversion design.



(a) The delay of the inversion design.



(b) The delay of the look up table design.

Figure 5.2: 128 bit AES delay.

Future research can consider developing a faster and smaller hardware design for AES. AES encryption and decryption can be designed with a more compact data path where the hardware resources are efficiently shared. For example, generating the keys required for each round can employ the same look up table used in the sub bytes oper-

ation. In order to minimize the hardware size, this table can be shared between the sub bytes operation and the key scheduler. Security and efficiency in power consumption and chip area are now being considered by cipher designers. In some designs, efficiency needs to be sacrificed in order to achieve higher security. Therefore, the challenge is to design a cipher which provides reasonable security while maintaining the efficiency.

References

128-bit versus 256-bit AES encryption. (2017). Seagate Technology. Retrieved from `www.axantum.com/AxCrypt/etc/seagate128vs256.pdf`

Coron, J. S. (2006). What is cryptography? *IEEE Security & Privacy Magazine*, *4*(1), 70-73.

Daemen, J., & Rijmen, V. (2002). *The design of Rijndael.* Springer, Berlin.

Dandalis, A., Prasanna, V. K., & Rolim, J. D. (2000). A comparative study of performance of AES final candidates using FPGAs. In C. Paar & C. K. Koc (Eds.), *International workshop on cryptographic hardware and embedded systems* (p. 125-140). Springer, Berlin.

Gaj, K., & Chodowiec, P. (2009). FPGA and ASIC implementations of AES. In C. K. Koc (Ed.), *Cryptographic engineering* (p. 235-294). Springer, Boston.

Ichikawa, T., Kasuya, T., & Matsui, M. (2000). Hardware evaluation of the AES finalists. In *In the third advanced encryption standard candidate conference* (p. 279-285).

Paar, C., & Pelzl, J. (2010). *Understanding cryptography.* Springer, Berlin.

Satoh, A., Morioka, S., Takano, K., & Munetoh, S. (2001). A compact Rijndael hardware architecture with S-box optimization. In C. Boyd (Ed.), *Advances in cryptology* (p. 239-254). Springer, Berlin.

Schaumont, P. R., Kuo, H., & Verbauwhede, I. M. (2002). Unlocking the design secrets of a 2.29 Gb/s Rijndael processor. In *Proceedings of the ACM design automation conference* (p. 634-639).

Smid, M. E., & Branstad, D. K. (1988). Data encryption standard: past and future. *Proceedings of the IEEE, 76*(5), 550-559.

Van Dyken, J., & Delgado-Frias, J. G. (2010). FPGA schemes for minimizing the power-throughput trade-off in executing the advanced encryption standard algorithm. *Journal of Systems Architecture, 56*(2-3), 116-123.