A COMPARATIVE ANALYSIS OF DIFFERENT AES IMPLEMENTATIONS FOR 65NM TECHNOLOGIES

By

ANDREA LAPIANA
Bachelor of Science in Electrical Engineering
Binghamton University
Binghamton, NY, USA
2010

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Masters of Science
December, 2015

A COMPARATIVE ANALYSIS OF DIFFERENT AES IMPLEMENTATIONS FOR 65NM TECHNOLOGIES

Thesis Approved:

Dr.James Stine

Thesis Advisor

Dr. Daniel Grischkowsky

Dr. Jingtong Hu

ACKNOWLEDGMENTS

There are a number of people who have been instrumental to my personal, educational and professional success. Between family, friends, teachers and mentors, I feel very blessed to have such a strong support network in my life.

To my husband, Giuseppe LaPiana. You have played many roles since you have come into my life: friend, classmate, boyfriend, roommate, co-worker and husband, but perhaps the one that encompasses them all is partner. You have been by my side throughout the last 9 years, providing immeasurable support in every aspect of my life whether at school, work or home. After the past year, I can add two more roles to the list, mentor and editor. Thank you for putting up with my throughout the process of this thesis. More specifically, for allowing me to bounce ideas off you, for giving feedback on things I need to include or clarify and for helping me to edit my writing. Although this journey has been stressful at times, you have been by my side providing love and support, and late night coffee!

To my parents, Joann and Leonard DeSanto. I would not be the woman I am today if it wasn't for your love and guidance throughout my life. Thank you for fostering my interests in math and science and tolerating my inquisitive nature from a young age. You have always encouraged me to pursue my dreams regardless of what they were and motivated me to do my very best. You taught by example that hard work and doing the right thing pays off in the end.

To my advisor, Dr. James Stine. Thank you for advising me throughout my graduate studies here at Oklahoma State University. In particular, thank you for providing feedback, support and encouragement throughout this whole thesis process.

Your optimistic outlook on life is contagious, I am glad to have you as a friend and mentor.

To all my previous teachers and mentors, thank you for being a resource for my continued growth and development as I have ventured throughout my educational and professional career.

Name: Andrea LaPiana Date of Degree: December, 2015

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A COMPARATIVE ANALYSIS OF DIFFERENT AES IMPLEMENTATIONS FOR 65NM TECHNOLOGIES

Major Field: Electrical Engineering

Encryption has a strong presence in todays digital electronics with the frequent transmission and storage of sensitive data.NIST selected AES as the standard encrytpion algorithm and it is commonly used as a fast solution to secure data. When designing VLSI systems, the task of balancing the area, power, and speed is a challenge; hardware encryption is no different. System requirements drive certain performance parameters to the forefront, identifying how to alter a design implementations to meet performance requirements is not always apparent. Multiple resources in this research field have identified AES algorithm features of interest and discussed their impact a few of the design tradespaces, however, a single comparative analysis was lacking. This thesis explores six different AES features key size, mode specificity, round key storage, round unravelling, SBOX implementation, and pipelining. A summarized view of the resulting designs allow readers to quickly analyze how each of the six features impacts speed, power, area, latency and throughput using 65nm process.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

The demand for hardware encryption is growing at a fast pace as the evolution of technology and digital electronics has matured to allow society to stay connected and simplify daily life. These modern day conveniences that allow us to shop on-line, easily perform bank transactions, and surf the web from mobile hotspots also makes our personal data vulnerable to attack.

Over the last few years the consumer market has shown a stronger interest in protecting data due to the damaging effects it could have on a companies profits and reputation. Recently, Comcast decided to encrypt all the channels they are digitally transmitting, requiring customers to have a special set-top box to decode the incoming signal [1]. This was done to avoid non TV subscribers from getting these channels for free. In the gaming industry, Sony Playstation 3 uses encryption to only allow authorized games to be played only on their console. The security of financial data is also a major concern where Point of Sale terminals are a vital attack point, such as the 2013 Target Hack where unencrypted credit and debit card information was harvested from around 40 million customers [2]. Consequently, the need for efficient security and encryption is important in today's society.

Hardware encryption plays a vital role in the military's transportation, communication and surveillance needs. Unencrypted navigation and reconnaissance information can allow adversaries to intercept and spoof a plane or UAVs GPS coordinates or view of what it sees [3]. And, it is damaging when government agencies fail to properly use encryption to protect classified and sensitive data. For example, [4] dis-

cusses how secret government technology blueprints were stolen by Chinese hackers and a data breach possibly caused the striking resemblance of China's J-31 to the US-F-35. Worse yet, it was announced that hackers infiltrated government systems to obtain unencrypted personal data and social security information for up to one million federal employees[5].

Physical implementation is also an important consideration as it impacts security, speed, area, and power of devices. For systems dealing with national security and protection of classified data, designers may emphasize speed and security. On the other hand, the desire for high speed and high throughput can be the goal for networking applications that require data transmission of secured information. Portable electronics such as RFID cards, smartphones and activity trackers might choose to optimize power and area efficiency due to a finite battery power source and a small form factor.

Previously, Advanced Encryption Standard (AES) implementations for both Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) have been introduced. Flexibility was described in [10] [11] [12] [20] with the ability to allow multiple key lengths, and also in [13] where multiple modes of operation were supported in hardware. Round key generation and pipeline ability affecting the throughput of a design were analyzed in [20] [21] [23]. On the fly key generation was analyzed in [20] and compared to round key storage in [21], both of which were ASIC designs, however, [23] performed the same comparative analysis in an FPGA. The substitution box (SBOX) used in the SubBytes step proves to have the biggest impact on area, speed, and power which is why many previous designs focus on this implementation feature most [14] [15] [16] [17] [18] [19] [22]. In [22], it was expressed that a high performance implementation is achieved by using 16 copies of this SBOX to perform the byte substitution. In [11] [13] [14] the SBOX is implemented as a look up table, whereas, in [15] [16] [17] [18] a combinational logic

approach is taken, using composite field arithmetic to perform the SBOX substitution.

This thesis will leverage previous work and compare and contrast a wide range of implementations in a single source. These will be demonstrated in 65 nanometer technology and the impact of these features have on the trade space will be discussed.

1.1 Research Goals

The main goal of this thesis is to provide an organized collection of different AES algorithm features like key size, block cipher mode, round key generation and storage, loop unraveling and pipelining and summarize their impact on the design tradeoffs of speed, area, power, and throughput for others to use as a quick reference. Another goal of this thesis is to provide a good explanation of the the AES algorithm and identify the areas that are critical to optimize. Also, this thesis should provide a detailed description of the designs discussed as well as the scripts and test benches used for their evaluation.

1.2 Report Format

Chapter 2 provides an overview of the AES algorithm where the details of each step of the process is explained as well as the different modes of encryption that are commonly used when encrypting large blocks of data. An overview of the ASIC design flow process is also described to show of how the work performed in this thesis was executed and how it fits into the bigger picture of VLSI design.

In Chapter 3 details of the different design implementations are elaborated and the methodology for creating, testing and evaluating these designs is presented. The Computer Aided Design (CAD) tools used to facilitate this process are mentioned and the test benches used to verify the correctness of the designs are described as well as how scripts were used to aid in the CAD process.

Chapter 4 contains the results of each design described in Chapter 3. Finally,

the conclusion summarizes the results and findings of the work described in the previous chapters and discusses how this can benefit future AES hardware encryption designers. In addition, some concluding ideas are given for future work in this area.

CHAPTER 2

Background

Encryption allows data to be securely stored, authenticated, and transported from one place to another by encoding it in such a way that it is incomprehensible to an attacker. The encryption process uses a cipher and secret key to transform the original message, as shown in Figure 2.1

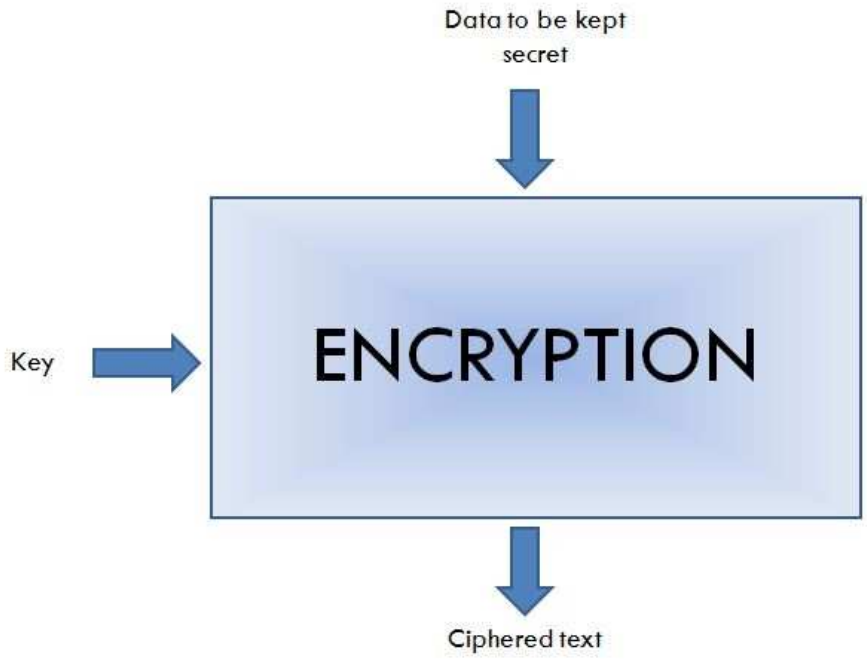


Figure 2.1: High Level Encryption

There are two types of encryption, asymmetric and symmetric key encryption. Asymmetric uses a different key, commonly known as a public private pair, to encrypt and decrypt data. On the other hand, symmetric key encryption uses the same key for both encryption and decryption, making the secrecy of the key vital to the

message security. Symmetric key encryption is commonly used because it is much faster compared with asymmetric and will be the focused form of encryption for this thesis. Although there are many different ciphers that can be used for encryption(AES, DES, Triple DES, Blowfish, Serpent), both the sender and receiver must agree on and negotiate a key prior to sharing data.

2.1 AES Algorithm Overview

The Advanced Encryption Standard(AES) is a symmetric key algorithm that is considered stronger than its predecessor Data Encryption Standard (DES) because it takes $5x10^{21}$ years for a brute force attack on an AES128 key as opposed to 400 days to crack a DES key [8]. The AES was proposed by Vincent Rijmen and Joan Daemen in 1999 in response to a request by the National Institute of Standards in Technology (NIST) for new a encryption method and has been chosen as the preferred standard of encryption [9].

The AES algorithm is structured to perform a series of four steps, SubBytes, ShiftRows, MixCols, AddRoundKey, each of which mathematically transform an input data block. Although it was proposed by Rijndael that the input data block sizes could be either 128 bit 192 bit or 256 bit, the AES standard defines a fixed input data block size of 128 bits [6]. The 128 bit input data block is conceptually arranged in a 4x4 matrix of bytes with each column of bytes representing a word. This is often referred to as a state matrix or state array and is shown in Figure 2.2

In [7] each of the data bytes of the state array represent elements in the GF(2^8) finite field. The byte, consisting of 8 bits, is represented as a polynomial in this finite field as:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \tag{2.1}$$

where the coefficients b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 represent each bit can take on the value 1

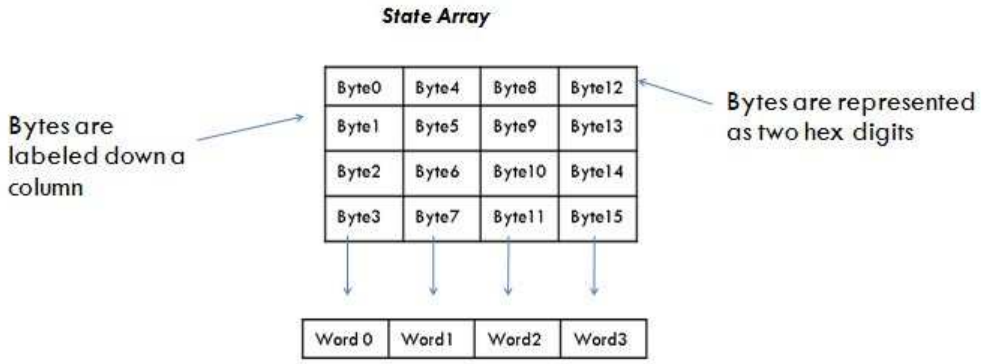


Figure 2.2: State Array

or 0. For example, hexadecimal '53' is represented by the polynomial $x^6 + x^4 + x + 1$.

The series of steps is usually referred to as a round and is iterated a specific number of times depending on the key length. There are three key lengths available, 128, 192, 256 bits. A table of key lengths and the associated number of rounds is shown in Figure 2.3

	Number of Rounds
AES-128	10
AES-192	12
AES-256	14

Figure 2.3: Number of Rounds for Various Key Lengths

A 128-bit key length allows for 3.4×10^{38} different key combinations, whereas, a 256-bit key length allows for 1.16×10^{77} different combinations. These numbers are important when discussing a brute force attack on encryption and are sometimes used as justification for using a 256 bit key instead of a 128 bit one. Since the key length impacts the number of rounds performed, the importance of security compared to the combined impact of area, speed, and power is often analyzed prior to choosing a key length. Figure 2.4 shows the full encryption process as a block diagram when

a 128 bit key is used.

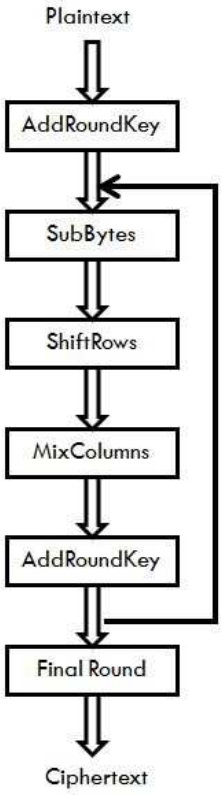


Figure 2.4: Encryption Process for 128 Bit Key

According to the AES standard [6], the initial block of data or plaintext is transformed using the original key in the AddRoundKey step prior to going through the designated number of rounds. Next, the SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations are performed sequentially and then that process is iterated for one less than the number of rounds required. The last round is shown outside the iteration loop because it is special and does not include the MixColumns transformation. The output of the final round is the encrypted data block or ciphertext.

2.2 SubBytes Transformation

The SubBytes transformation updates each byte in the state array with a corresponding byte in the Substitution Box (SBOX). The SBOX is the result of performing the multiplicative inverse followed by the affine transform of an element in the state array [7]. The details of this process are often obscured and a Look Up table is often used, since each 8 bit element will map to the same value after performing these two operations, as shown Figure 2.5.

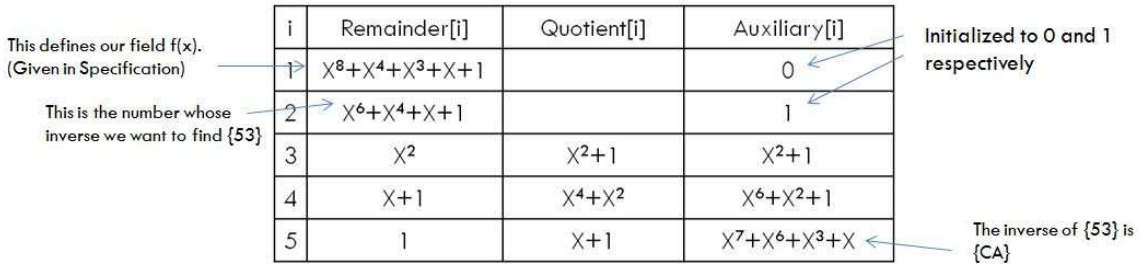
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	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 2.5: The SBOX

Unless this table is replicated 16 times, like the high performance implementation described in [22] suggests, the SubBytes step of the round can take 16 clock cycles. Normally, this is undesirable, so multiple copies of this table are made so that the look-up can occur within one clock cycle. Normally this will have a impact on the area consumption with the severity varying depending on the type of platform used. Therefore, it is important to understand the process used to generate the SBOX, so that the decision of whether or not to use a look up table can be made.

2.2.1 Extended Euclidean Algorithm

One way to find the multiplicative inverse is to use the Extended Euclidean Algorithm [24]. An example of the algorithm is given to help explain the process and Figure 2.6 presents the process.



i	Remainder[i]	Quotient[i]	Auxiliary[i]
1	$X^8+X^4+X^3+X+1$		0
2	X^6+X^4+X+1		1
3	X^2	X^2+1	X^2+1
4	$X+1$	X^4+X^2	X^6+X^2+1
5	1	$X+1$	$X^7+X^6+X^3+X$

Figure 2.6: Extended Euclidean Algorithm Example

The first column in Figure 2.6 labeled i, is a variable to keep track of the iteration, the next three columns keep track of the, remainder, quotient and auxiliary. Remainder(1) is filled in with the polynomial that defines the finite field, as given in the AES specification as $x^8 + x^4 + x^3 + x + 1$ or hex '11B' [6]. The Remainder(2) is the number whose inverse needs to be determined. In this example, the multiplicative inverse of hex '53' is the goal, so '53' is represented in polynomial form. Quotient(1) and Quotient(2) are left blank because they are not assigned yet. In addition, auxiliary(1) and Auxiliary(2) are initialized to 0 and 1, respectively. Then, to calculate the polynomial for Remainder(3) the modulus operation is performed: Remainder(1) mod Remainder(2). To calculate the polynomial for Quotient(3) the division operation is performed: Remainder(1)/Remainder(2). To calculate the polynomial for Auxiliary(3) the operation is: Quotient(3)*Auxiliary(2) + Auxiliary(1). These calculations are shown in Figure 2.7

Normally, performing the Extended Euclidean algorithm is an iterative process that ends when a remainder of 1 is reached. The auxiliary of the ith iteration where a remainder of 1 was reached is the inverse of the original number. So, after going

Remainder [3]:

100011011
1010011
001010111
001010011
000000100 = X^2

Quotient [3]:

000000101 = X^2+1
1010011 | 100011011
1010011
001010111
001010011
000000100

Auxiliary [3]

$X^2+1 * 1 + 0 = X^2+1$

Figure 2.7: Extended Euclidean Algorithm Calculations

through the process in the example, the multiplicative inverse of hex '53' is hex 'CA'. For most implementations, this is difficult to implement in hardware, so another approach using composite field arithmetic is often used.

2.2.2 Composite Field Arithmetic

In [16] another approach for performing the multiplicative inverse is explored. This is shown in the block diagram in Figure 2.9

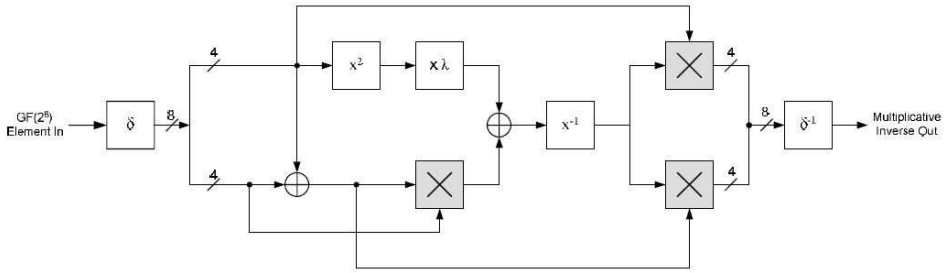


Figure 2.8: Multiplicative Inversion Using Composite Field Arithmetic [16]

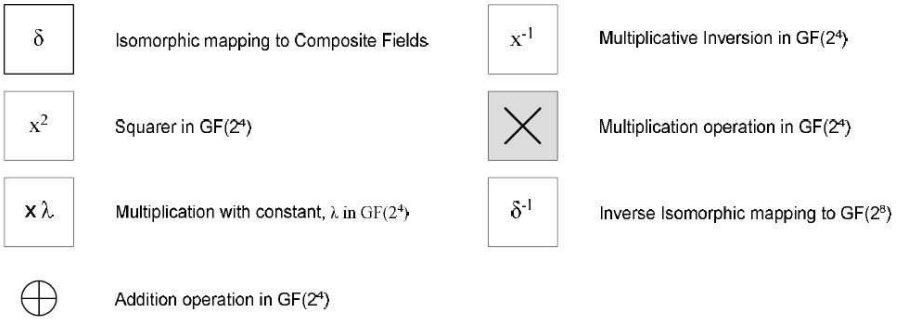


Figure 2.9: Legend of Multiplicative Inversion Boxes [16]

For this work, the multiplicative inverse calculation is decomposed in composite

field arithmetic to use lower order fields like GF(2^4), GF(2^2) and GF(2) as opposed to the more complex GF(2^8). First, the element to be substituted is translated to composite field representation via an isomorphic function. The isomorphic function uses matrix multiplication to combine the element to be substituted with a fixed matrix. Next, a number of different operations like squaring, multiplication, addition, and inversion are performed in the lower level fields, as shown in Figure 2.9. Once this is complete, the result is mapped back to GF(2^8) field via the inverse isomorphic function.

Once the multiplicative inverse is completed for each cell in the state matrix, the affine

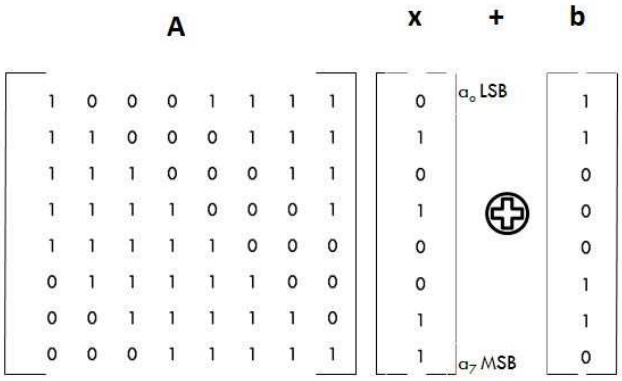


Figure 2.10: Affine Transform

transform is taken to mangle the bits. The affine transform is defined by the matrix A, as shown in Figure 2.10 where X is the new value after taking the multiplicative inverse. Multiplying A and X using matrix multiplication and XORing that result with B is shown as a series of XOR equations shown in Figure 2.11.

In Figure 2.11, the affine transform of the hex value 0xCA is shown to be 0xED. The goal of the SBOX is to provide non-linearity in the cipher, reducing the correlation between the input and output bits [7].

For the decryption process the InvSbox is used. This process works by generating the InvSbox, done in the opposite manor, where, first, the inverse affine transform of the input value is taken. This is defined by the matrix in Figure 2.12.

$$
\begin{aligned}
a'_0 &= a_0 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7 \oplus 1 \\
&= 0 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 1 \\
a'_1 &= 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0 \\
a'_2 &= 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
a'_3 &= 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1 \\
a'_4 &= 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 0 \\
a'_5 &= 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 = 1 \\
a'_6 &= 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1 \\
a'_7 &= 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1
\end{aligned}
$$

Figure 2.11: Affine Transform XOR Equations

$$
\begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix}
\oplus
\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
$$

Figure 2.12: Inverse Affine Transform

Then the multiplicative inverse is done by either using the Extended Euclidean Algorithm or composite field arithmetic discussed previously. The resulting InvSbox is shown in Figure 2.13

2.3 ShiftRows Transformation

The ShiftRows transformation shifts the rows of the state array by a certain amount. Row(0) is left alone, Row(1) is shifted to the left one byte, Row(2) is shifted to the left two bytes and Row(3) is shifted to the left three bytes. After the shift rows transformation the byte order of the block is scrambled and shown in Figure 2.14

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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 2.13: The Inverse SBOX

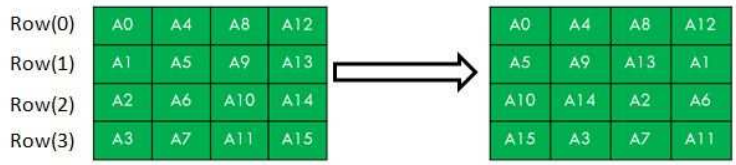


Figure 2.14: Shift Rows for Encryption

For decryption, bytes are shifted the opposite direction, to the right instead of to the left. So Row(0) is left alone, Row(1) is shifted to the right by one byte, Row(2) is shifted to the right by two bytes and Row(3) is shifted to the right by three bytes.

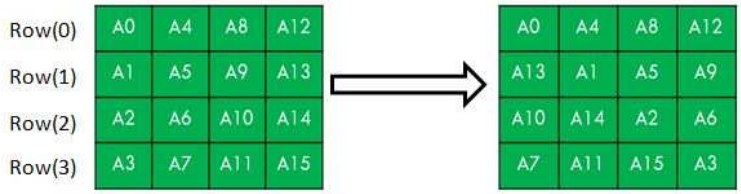


Figure 2.15: Inverse Shift Rows for Decryption

2.4 Mix Columns Transformation

In the MixColumns transformation the state array is multiplied by a circulant Maximum Distance Separable (MDS) matrix. The columns of the circulant matrix are

shifted to the right circularly. The first columns is left alone, the second is shifted by one byte, the third column is shifted by two bytes and the fourth column is shifted by three bytes. The circulant matrix used in the AES algorithm is defined by the polynomial $c(x) = 3x^3 + x^2 + x + 2$ where 2 represents c_0, 1 represents c_1 and c_2, and 3 represents c_3 shown in Figure 2.16. Each column of the state array is then multiplied by this entire circulant matrix to produce each column of the new state array after the transformation.

Then, the correct matrix multiplication is performed, as shown in Figure 2.17,

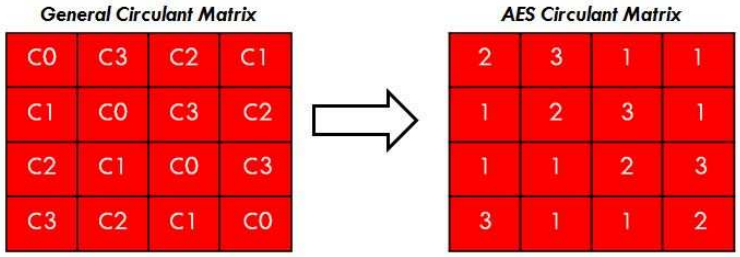


Figure 2.16: Circulant Matrix

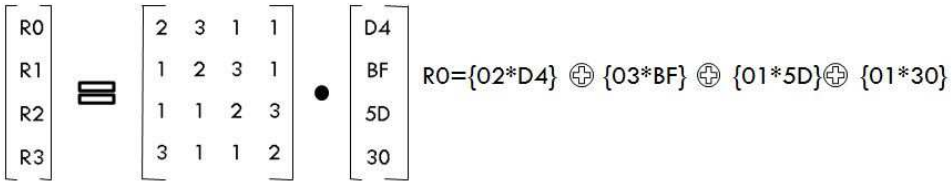


Figure 2.17: Circulant Matrix Column Multiplication

where the first row of the circulant matrix is multiplied by the first column of the state array. The first operation is multiplication by hex 0x2, which in finite field mathematics is a bit shift to the left of the original value. If the most significant bit is a 1, the result is XORed with hex 0x1B [7]. In performing multiplication by 0x3 it can be split into the XOR combination of multiplication by 0x2 and 0x1 to simplify the calculation. This is shown in Figure 2.18. The results of each of these multiplications are XORed together to obtain the first element of the new column. This process is iterated to determine each element in the new state array.

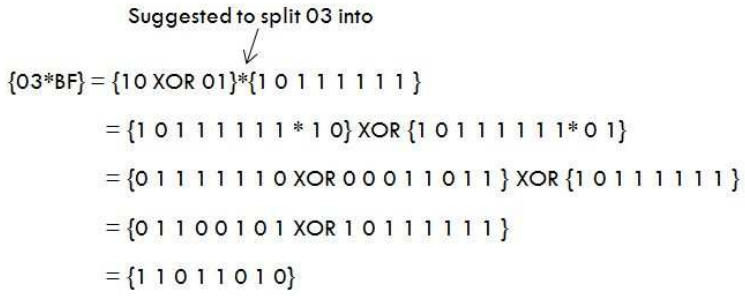


Figure 2.18: Multiplication by 3 in GF(2^8)

The purpose of the MixColumns step along with the ShiftRows step is to provide diffusion in the Rijndael cipher [7]. Diffusion in cryptography means the output bits depend on the input bits, but in a complex way. If one input bit is changed, the output bits should change completely, in an unpredictable manner [29]. For decryption, the InvMixColumns step is performed using the Inverse Circulant Matrix shown in Figure 2.19

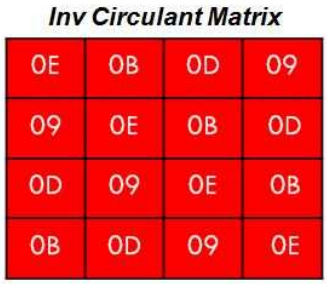


Figure 2.19: Inverse Circulant Matrix

The multiplication of each element by 0x9 0xB 0xD and 0xE are simplified when broken up into an XOR combination of simpler multiplications. For example, multiplication by 0x0B can be decomposed into multiplication by 0x8, which is a shift left by 3, XORed with multiplication by 0x2, shift left by 1 and XORed with multiplication by 0x1, which is just the original element.The MixColumns and InvMixColumns steps are not performed in the final round of the encryption and decryption process.

2.5 Key Expansion

The original key is expanded to create a key for each round which is used in the AddRoundKey step of the encryption process. A 128 bit key is arranged in the same state array that was previously described for the data. It contains 4 words, or columns in the state array representation, an additional 40 words are needed for the 10 rounds of the encryption process. To visualize the key expansion process, a key state array is shown in Figure 2.21 followed by blank round key state arrays. Each column is labeled to represent the word number, it can be seen that the first 4 words, $W_0W_1W_2W_3$ represent the original key. This model will be used to explain the key expansion process using a 128 bit key.

A round constant is needed to determine each round key and can be coded as a simple look up table, however, the derivation of this element will be explained briefly. The round constant is a word whose right most bytes are always zero: RCON[i] = (RC[i], '00','00','00') The initial round constant for encryption is RC[1] = 01 where the following ones can be determined by : RC[i]= 02 * RC[i-1], or a simple bit shift of the previous one. The 10 round constants needed for AES128 are shown in Figure 2.20

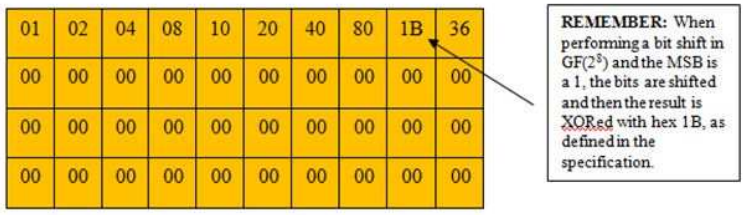


Figure 2.20: RCON Table

It is important to note that when calculating the round constant for the 9th round that the MSB before the shift is a '1' and, because of this, the resulting value after the shift is XORed with hex 0x1B. For decryption, the RCON table is the exact opposite. That is, the first RCON value is 36, then 1B, then 80, etc. To determine if the first

word of the round key is different and slightly more complicated than the other three words. As shown in Figure 2.21, the first word after the original key is labeled W_i, and the words of the original key are labeled in decrementing order. Then, a series of steps are performed to find W_i

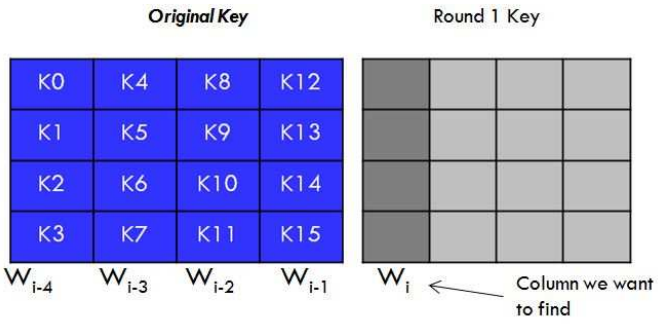


Figure 2.21: Key Expansion First Column Setup

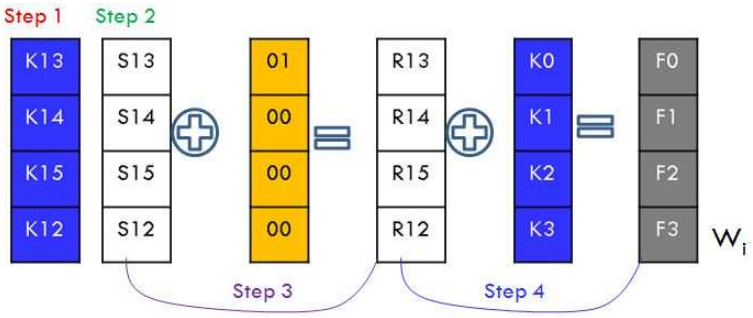


Figure 2.22: Key Expansion First Column

The first W_{i-1} is rotated to the left by 1 byte as shown in step 1 of Figure 2.22. Next, each byte is substituted in step 2 using the S-box look up table, which was previously explained. Then, that word is XORed with the first word of the RCON table, shown in step 3. The purpose of the round constant is to destroy any symmetries that may have been introduced in other steps of the key schedule [7]. Finally, the result of this is XORed with W_{i-4} in step4 and becomes W_i. To determine the remaining columns of the new round key the conceptual picture is updated and relabeled to place W_i where the new column to be determined, shown in Figure 2.23.

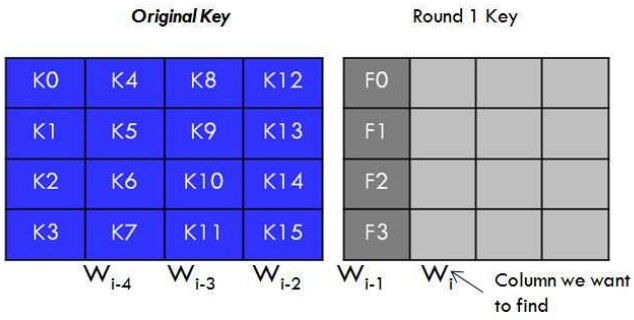


Figure 2.23: Key Expansion Other Columns Setup

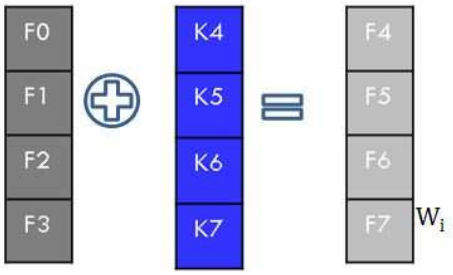


Figure 2.24: Key Expansion Other Column

When calculating any round key word other than the first one, W_{i-1} is XORed with w_{i-4}, it is i-4 because the key length is four words. This is repeated two more times to complete the last two words of the first round key. The key expansion is then performed 9 more times to complete the key schedule, resulting in the original key plus 10 round keys that are used in the encryption process.

It is important to note that the first word of a new round key happened to be the same word that required the extra steps to determine. In a more general case, the special set of steps is done to any word that is a multiple of the original key length. For AES128, this is W4, W8, W12, W16 ... W40.

For AES192, 12 round keys are needed, that requires a total of 52 round key words and includes the original key. The process for determining the key schedule is the same as AES128 except the special set of steps is done on W6, W12, W18...W48.

For AES256, 14 round keys are needed, that requires a total of 60 round key words and includes the original key. The process for determining the key schedule is the

same as AES128 except the special set of steps is done on W8, W16, W24...W46. Another difference when using a 256 bit key is that there is another transformation that is done on every 8th word starting at W12 and continuing with W20, W28, W36...W52. Normally, W_i is XORed with W_{i-8}, i-8 because the key length is 8, but instead each byte of W_i is first transformed using the SBOX substitution and then XORed with W_{i-8}.

The key expansion algorithm assures that there are no weak keys [7], which is important because a weak key reduces the security of the cipher in a predictable manner potentially helping attackers of the system.

2.6 AddRoundKey Transformation

In this step of the encryption process, the state array is XORed with a round key generated from the previously mentioned key expansion process.

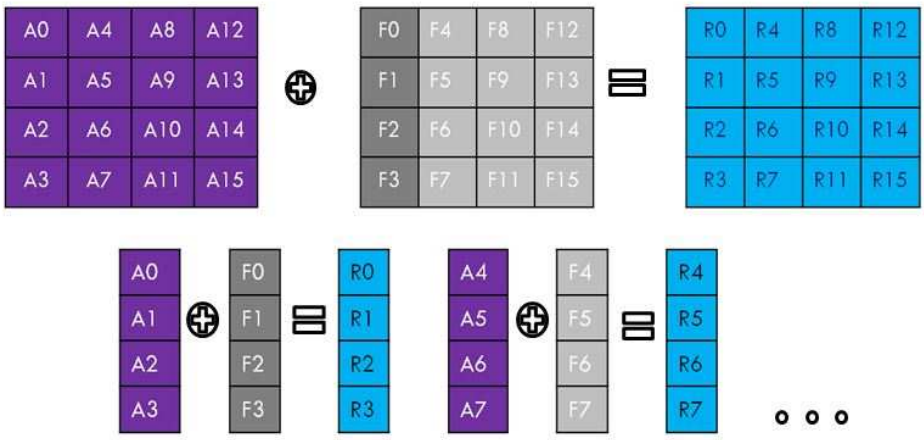


Figure 2.25: Add Round Key Step

The purple matrix in Figure 2.25 represents the current state array, the gray matrix represents the round key, and the resultant matrix is shown in blue. During this step, each word of the state array and round key matrices are XORed together to create a word in the resultant matrix.

2.7 Decryption Process Overview

The decryption process is similar to the the encryption process in that four transformations are performed iteratively for 9 rounds and a final special round which omits the MixColumns step. The difference is that the inverse transformations are performed for all but the AddRoundKey step. The flow diagram in Figure 2.26 depicts the decryption process.

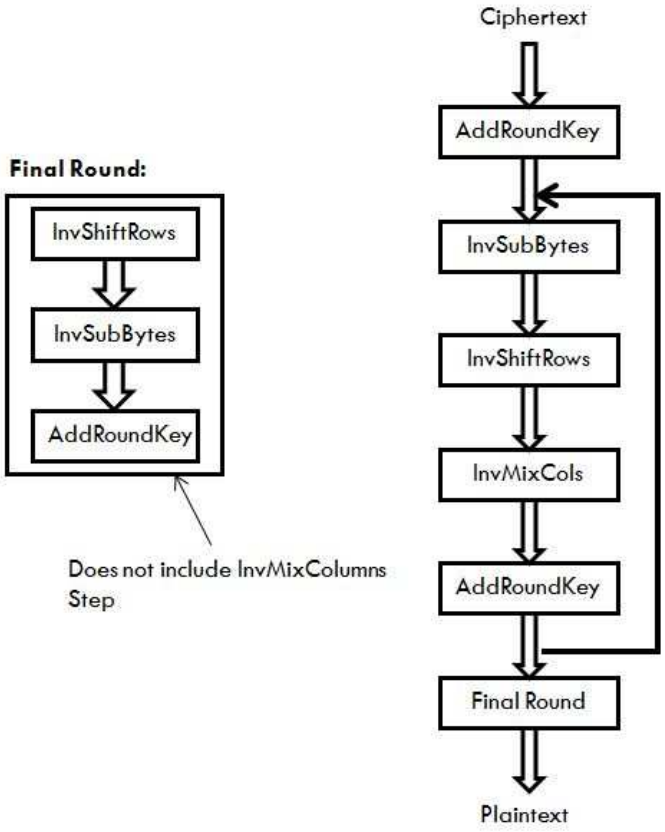


Figure 2.26: Decryption Process

2.8 AES MODES OF OPERATION

Practical applications of AES encryption usually work with data pieces larger than the simple 128 bits that the algorithm accepts as an input. Because of this, different modes of operation are used to allow large amounts of data to be encrypted under

the same key. For a better understanding of these modes the Forward Cipher will be the name used for the encryption process and the inverse Cipher will be the name used for the decryption process as described previously in the chapter.

2.8.1 ECB MODE

An Electronic Code Book or ECB mode is the simplest way to encrypt a large message. In this mode, the message is broken up into 128 bit blocks and the Forward Cipher is applied to each block, as shown in Figure 2.27. To decrypt, the ciphertext is broken up into 128 bit blocks and the Inverse Cipher is performed, as shown in Figure 2.28

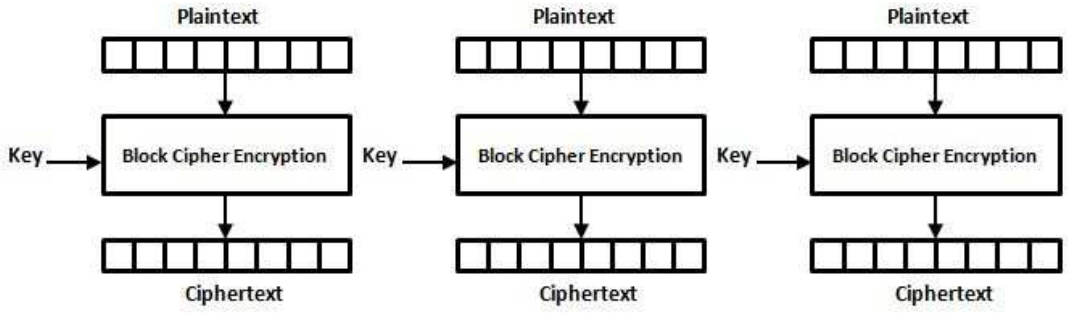


Figure 2.27: ECB Mode Encryption

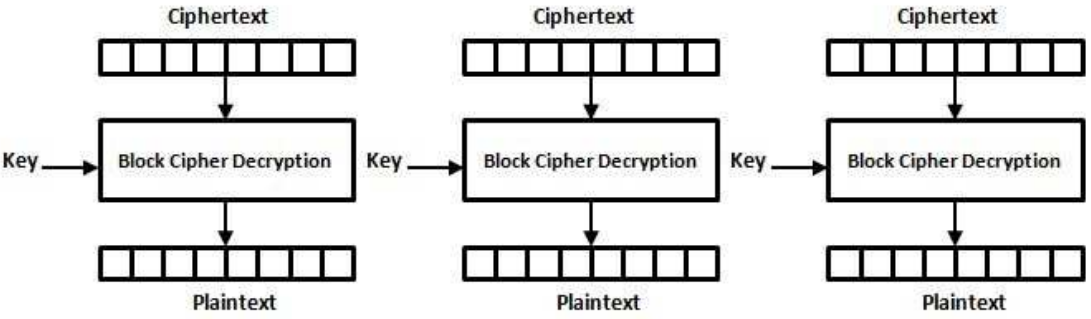


Figure 2.28: ECB Mode Decryption

One problem with this mode of operation is that if a message contains a large number of identical data blocks, they will all be mapped to the same cipher text blocks. An example of this is shown in Figure 2.29 [25]. That is, when the image

is encrypted using ECB mode it is not truly masked, because all of the pixels that were the same color were mapped to the same encrypted value. This is a flaw when encrypting large messages under the same key. The image on the right in Figure 2.29 shows the same original image encrypted using one of the other modes of encryption, discussed later in the section [26].

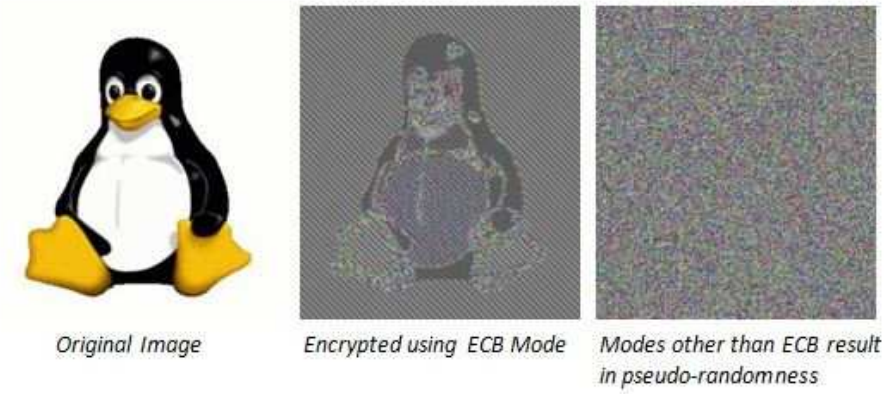


Figure 2.29: Linux Penguin Encrypted with Different Modes [25]

2.8.2 Initialization vector

An initialization vector is used in all other modes of operation to produce unique ciphertext even if the same data is encrypted multiple times with the same key. The initialization vector does not need to be kept secret, however, it is important not to reuse the same vector under the same key [26].

2.8.3 CBC Mode

In cipher block chaining or CBC mode, the initial plaintext block is XORed with an initialization vector, then the Forward Cipher is applied. Each of the subsequent plain text blocks are XORed with the previous cipher text and then put into the Forward Cipher for encryption, as shown in Figure 2.30. For decryption, the cipher text is run through the Inverse Cipher and then XORed with the same initialization vector

that was used for encryption. Each subsequent ciphertext is put through the Inverse Cipher and then XORed with the previous ciphertext, as shown in Figure 2.31.

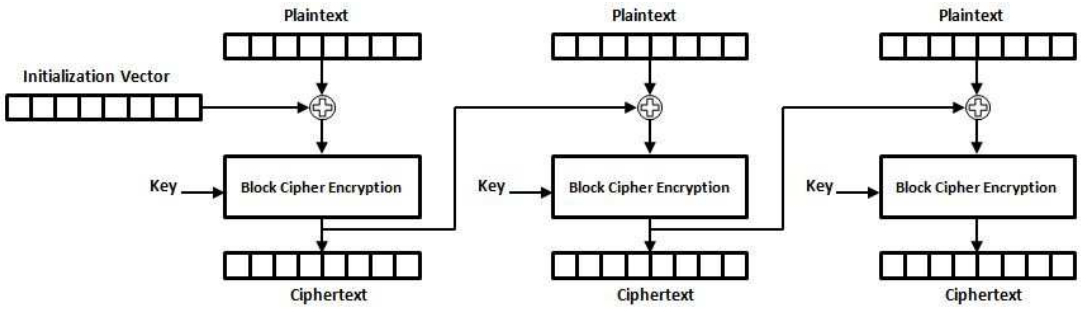


Figure 2.30: CBC Mode Encryption

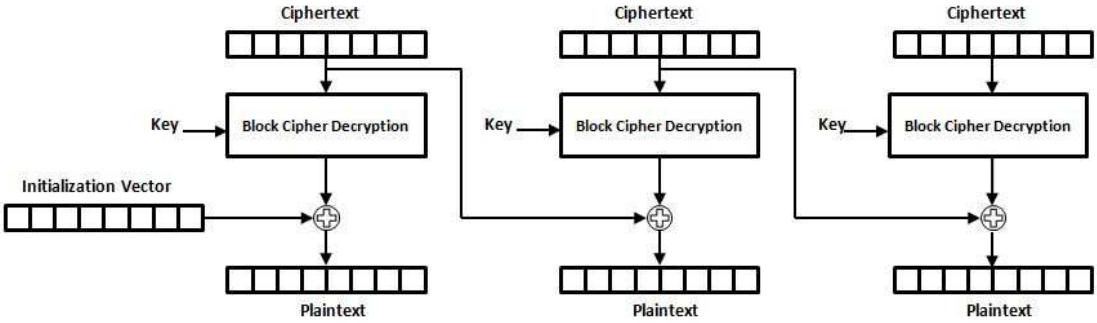


Figure 2.31: CBC Mode Decryption

With this mode of operation, encryption requires ciphertext from the block before that is not available until after the Forward Cipher is applied, therefore, it cannot be parallelized. The decryption process, however, can be parallelized because it uses the previous blocks ciphertext which is available at the start of decryption. The message must be a multiple of 128 bits or it must be padded to be a multiple of 128 bits. Another drawback with this mode is that a one bit change in the plaintext affects all subsequent ciphertexts during encryption. A one bit change in the ciphertext causes a complete corruption of that current blocks plaintext and inverts that same bit in the following blocks plaintext [26].

2.8.4 CFB Mode

For Cipher Feedback or CFB mode the initialization vector is encrypted at the start for both encryption and decryption and the result is XORed with either the plaintext block or the ciphertext block. For encryption, each of the subsequent blocks use the previous ciphertext as the input to the Forward Cipher and then the plaintext is XORed with the result, as shown in Figure 2.32. For decryption, each subsequent block uses the previous ciphertext as the input to the Forward Cipher and the result is XORed with the current ciphertext to produce the plaintext, as shown in Figure 2.33.

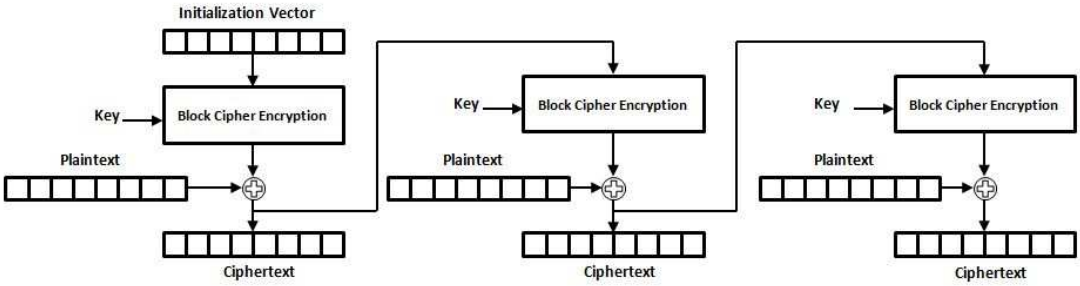


Figure 2.32: CFB Mode Encryption

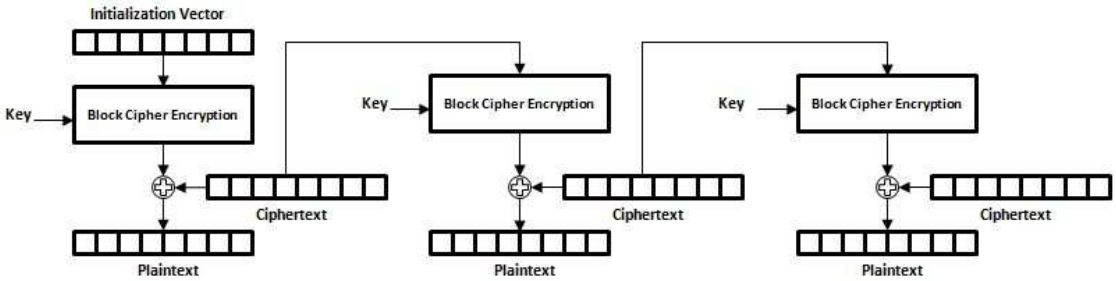


Figure 2.33: CFB Mode Decryption

One advantage of using CFB mode over the previously mentioned CBC mode is that both encryption and decryption use the Forward Cipher logic. Another, is that the message does not need to be padded to a multiple of 128 bits. Similarly to CBC,

the CFB mode of encryption cannot be parallelized because it uses the previous blocks ciphertext, which is not available until after the Forward Cipher and XOR operations are performed. However, the decryption process can be parallelized, because it uses the previous blocks ciphertext as the input to the Inverse Cipher, which is available at the start of decryption. Just like the CBC mode, a one bit change in the ciphertext causes an inverted bit in the current blocks plaintext and a complete corruption in the following blocks plaintext [26].

2.8.5 OFB Mode

For the Output Feedback or OFB mode the initialization vector is encrypted at the start and the result is XORed with either the plaintext block or the ciphertext block. For encryption, each of the subsequent blocks uses the previous Forward Cipher blocks as the input to the current Forward Cipher. The current Forward Cipher result is XORed with the plaintext to generate the ciphertext as shown in Figure 2.34. For decryption, each subsequent block uses the previous Forward Cipher blocks as the input to the current Forward cipher. The current Forward Cipher result is XORed with the ciphertext to produce the plaintext as shown in Figure 2.35.

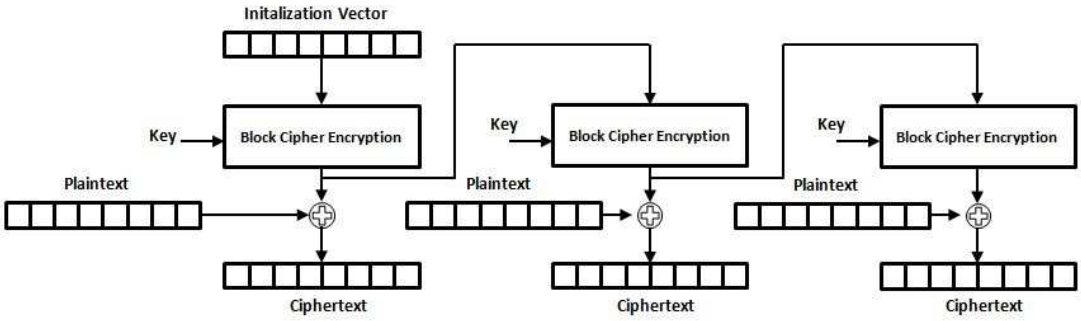


Figure 2.34: OFB Mode Encryption

Similarly to CFB, an advantage to using OFB mode is that both encryption and decryption only use the Forward Cipher logic. Neither the encryption nor decryption process can be parallelized, because the next input relies on the previous output from

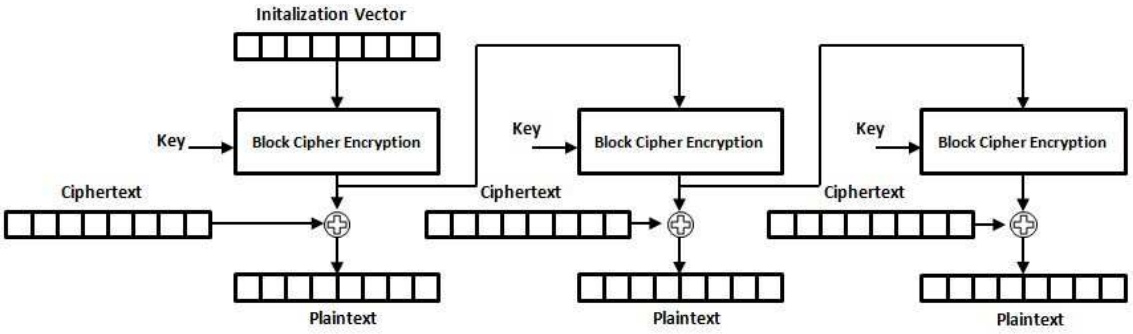


Figure 2.35: OFB Mode Decryption

the Forward Cipher. However, since the initialization vector, not the data, is fed into the forward cipher, it is possible to perform the forward cipher calculation on the initialization vector up front and then parallelize the XOR calculation with the data. The general rule of using a unique initialization vector with each message under the same key applies. Flipping a bit in the ciphertext produces the same flipped bit in the plaintext. This does not need to be padded to a multiple of 128 bits [26].

2.8.6 CTR Mode

The Counter or CTR mode of operation uses a counting vector for encryption and decryption as the input to the Forward Cipher instead of an initialization vector. The counting vector must not repeat for any blocks encrypted under the same key and can consist of a nonce concatenated with a count value or simply just a count value. For encryption, the result from the Forward Cipher is XORed with the plaintext, whereas, for decryption the result is XORed with the ciphertext as shown in Figures 2.36 and 2.37, respectively.

Both encryption and decryption use only the Forward Cipher and the message does not need to be a multiple of 128 bits, just like CFB and OFB. Since this mode does not rely on calculations of the previous block of data for encryption or decryption both can be parallelized. Flipping a bit in the ciphertext produces the same flipped

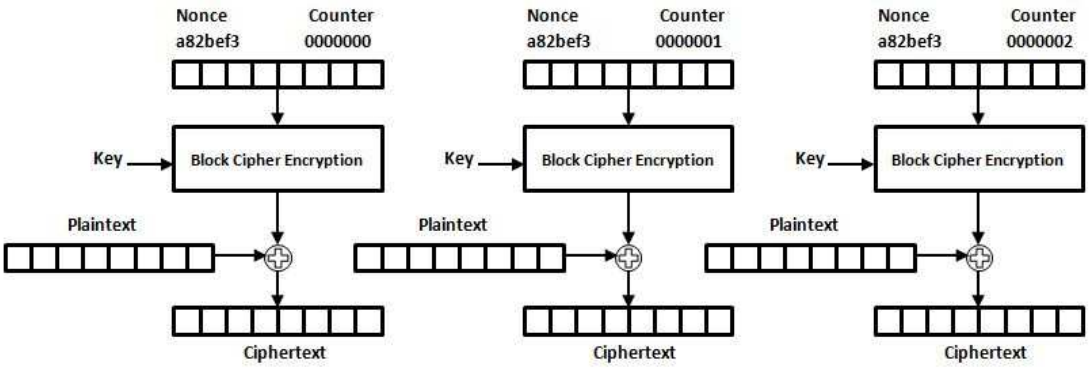


Figure 2.36: CTR Mode Encryption

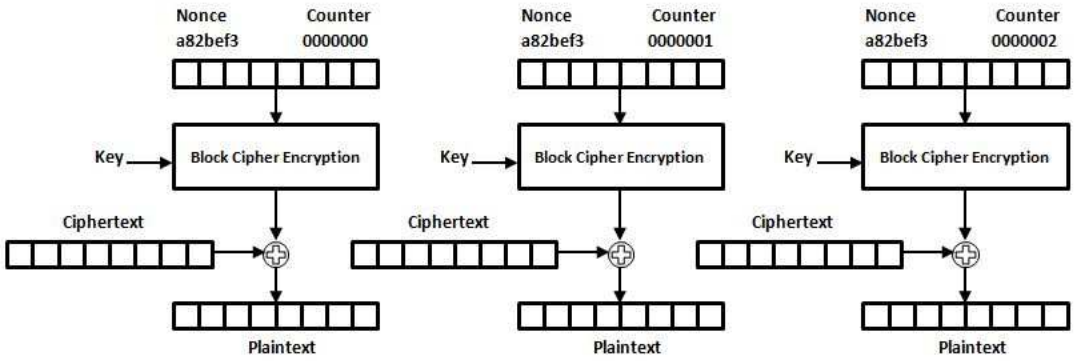


Figure 2.37: CTR Mode Decryption

bit in the plaintext and vice versa [26].

It is a common misconception that counting mode is suseptable to differential cryptanalysis due to the fact that counting vectors for successive blocks only differ by a small amount. However, if conclusions can be made about the result of the cipher because of the knowledge of the input, this would be an inherent flaw with the Forward Cipher and not a result of the Counter Mode [27].

	ECB		CBC		CFB		OFB		CTR	
	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC
Forward/Inverse Cipher	For	Inv	For	Inv	For	For	For	For	For	For
Pipeline?	Yes	Yes	No	Yes	No	No	No	No	Yes	Yes
XOR before/after	None	None	Before	After	After	After	After	After	After	After

Figure 2.38: CTR Mode Decryption

Figure 2.38 shows a comparison of each of the modes of operation described in this chapter. All modes are compared based on hardware simplicity, whether both the forward and inverse cipher are needed, and can the encryption and decryption process be pipelined or does it depend on the previous blocks cipher results. After analyzing the table, the counting mode seems to be optimal as it is just as robust as the others modes while only needing the Forward Cipher logic and also has the ability to be pipelined.

2.9 ASIC Design Flow

The process of going from design concepts to a finished product is represented in a simplified design, as shown in Figure 2.39. This is an iterative process and is in some cases can be hierarchical. The process has four main phases, VHDL Design Entry, Synthesis, Physical Layout and Fabrication. At the end of each phase, simulation is performed to verify the design which is important to the process.

Beginning with a design concept the details of the hardware are written using a Hardware Description Language (HDL). As the hardware is being developed, tests are written to exercise the features of the hardware and detect correct operation. If the block does not function as intended it must be altered and/or redesigned. Once a correct Register Transfer Level (RTL) simulation is working, the design is then synthesized, where the described logical operations are mapped to actual technology specific gates. It is important during the design phase that the hardware is written explicitly to make it easier for the synthesizer to translate the logic correctly. As the design is being mapped to gates the tool takes into consideration timing, power, and area and balances these using constraints. After the most optimal design is achieved, reports are generated summarizing each trade space. If there are any violations in this report, where the tool was unable to meet the clock speed, further investigation is needed. In addition, the resulting synthesized netlist must be free of preliminary

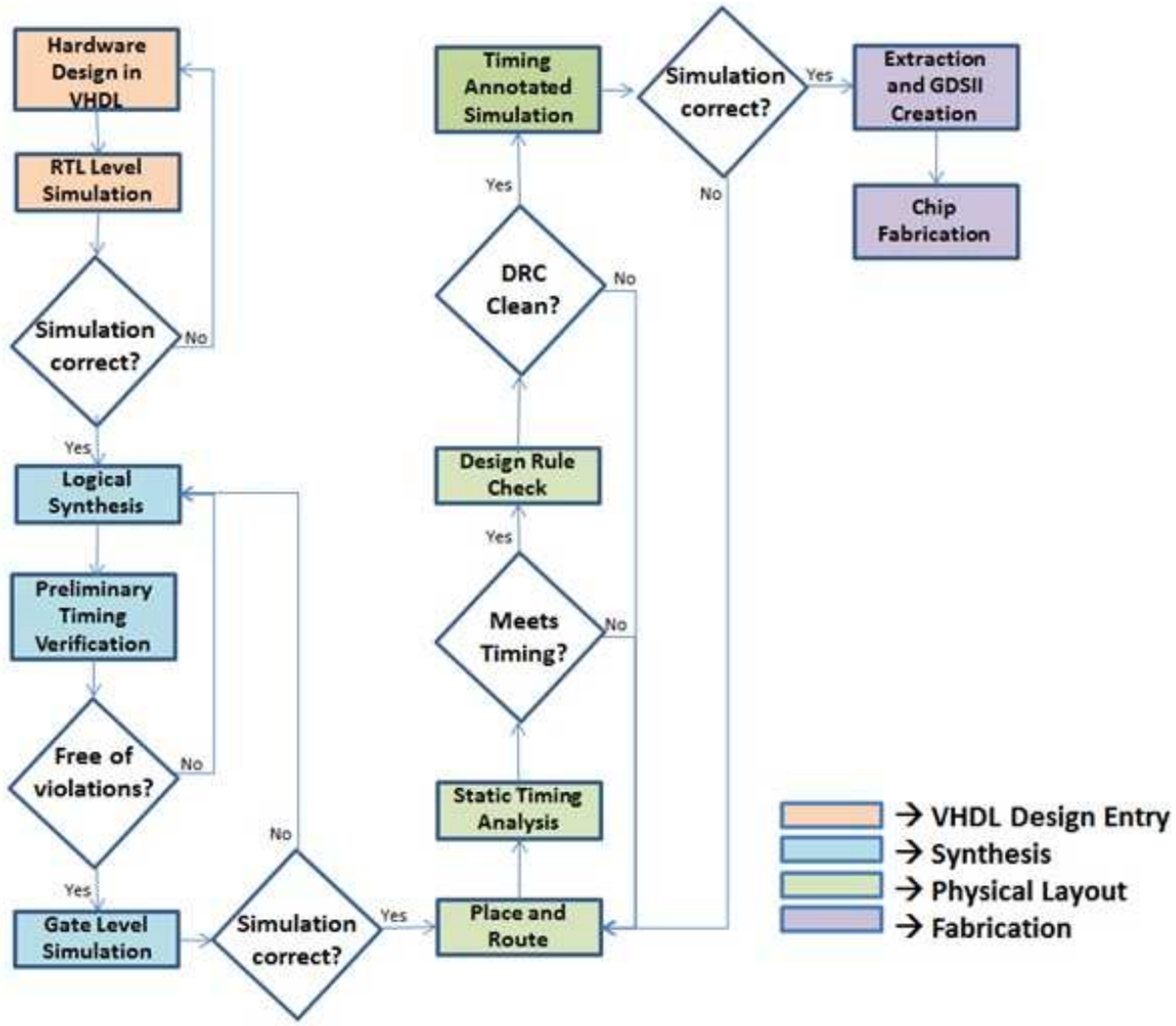


Figure 2.39: ASIC Design Flow

timing violations before it can be simulated.

Once the mapped design meets the constraints, the new netlist generated from synthesizer is used to run a gate level simulation to ensure the logic was translated properly. If there are any errors in simulation, the netlist must be analyzed to see where the translation error occurred and the VHDL or constraint file must be altered

Next a working, mapped design is then entered into a place and route design tool. Here, a floorplan is created based on the placement requirements defined for the level of hierarchy; that is, at the top level this will be the footprint of the chip. The placement of the gates and routing is done using a series of scripts containing specific

detailed commands that direct the tool throughout the process. During place and route, the tool optimizes the design for the new characteristics that was not taken into account during the synthesis phase. For example, if a logic path is spaced out across the length of the chip, the timing delay that is now introduced by the wire connecting them must be taken into account to still meet the clock speed requirements. This timing analysis along with design optimization is done during this phase and a resulting report is generated. If there are any path violations at this point, the series of scripts used may need to be altered to guide the tool further.

Once a violation free design is achieved, a check called a Design Rule Check (DRC) is done to ensure that all of the design rules defined by the fabrication facility are met. This includes rules like minimum spacing between wires and minimum wire width. Once these are all resolved, a new netlist is generated and is used in a timing annotated simulation to verify that any alterations to the design to help with timing did not impact its functionality. Finally, a DRC clean and timing accurate design is extracted using another CAD tool. This creates a GDSII mask file which is then sent off to the fabrication facility to construct the resulting design, completing the hardware design process.

2.10 Summary

In this chapter a detailed explanation of AES encryption described each step of the process, different modes of operation for large blocks of data, and also gave a brief overview of the VLSI design process. The next section will discuss a number of implementations considerations used in a tradeoff analysis to determine the best implementation for users motivated in a couple of these areas.

CHAPTER 3

METHODOLOGY

The starting design for this thesis was chosen due to availability. The design uses a 128 bit key that is expanded upfront and stored in two sets of key registers to allow for two rounds of logic to be calculated at once. The design implements ECB mode with the ability to support other modes with software. The design also uses a VHDL coded look up table for the SBOX substitution.

The desired control design for this experiment is an AES unit with a 256 bit key using the counting mode operation. Applications for a particular need of robust security often desire AES256 because it is the least susceptible to brute force attack when compared to other encryption methods; thus, NSA and government agencies use AES256 to protect top secret data. For this reason it was chosen as the key size of the control experiment. The ECB design provides a good starting design due to its ability to support all the modes, however, it is usually not chosen unless flexibility is a design requirement. As discussed in Chapter 2, the counting mode of encryption provides a nice balance between robustness, pipeline ability and area. For these reasons this specific mode was chosen as the control for the experiment.

To go from the starting design to the desired control design, one feature will be changed at a time. First, the starting design will be implemented, tested to verify correctness and then synthesized. Then alterations will be made to accommodate a 256 bit key and the design will be retested and synthesized. Finally, the 256 bit design is modified to specify the counting mode operation and again tested and synthesized.

From the control design, different features of the design will be varied: the SBOX,

Key Storage, number of rounds per clock cycle and pipleining, and they will be tested, synthesized and documented. This will allow for an analysis of how these features affect the design with regard to area speed and power. A summary of the features associated with each of the design implementations that will be discussed in this chapter is shown in Table 3.1.

Table 3.1: Summary of Implementations and Features

Design	Key	Mode	Rounds/clk	SBOX	Key Gen	Pipelined
AES128	128	ECB	2	LUT	Stored	No
AES256	256	ECB	2	LUT	Stored	No
AES256_CTR	256	CTR	2	LUT	Stored	No
AES128_CTR_logicsbox	256	CTR	2	Logic	Stored	No
AES256_CTR_OTF	256	CTR	2	LUT	OTF	No
AES256_CTR_4r	256	CTR	4	LUT	Stored	No
AES256_CTR_pipelined	256	CTR	2	LUT	Stored	Yes

The next sections will describe these implementations in more detail.

3.1 AES-128

3.1.1 Design

The starting design uses a single 128 bit key that is loaded in and stored in a flip flop based register. A top level block diagram for this design can be seen in Figure 3.1. The top level input signals are critical to the operation of the lower level AES128 Core Logic units' functionality and are coordinated via a simple finite state machine. The cloud of logic represents both combinatorial and sequential logic used to register the incoming signals and manage a counter to keep track of the AES core logic behavior.

An expanded view of the state finite machine is shown in Figure 3.2.

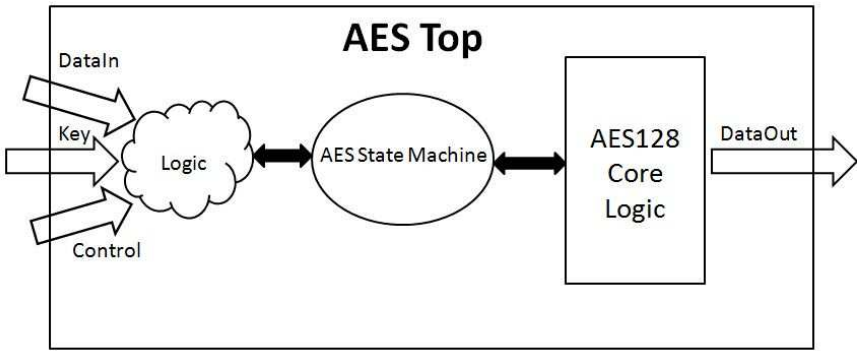


Figure 3.1: AES Top

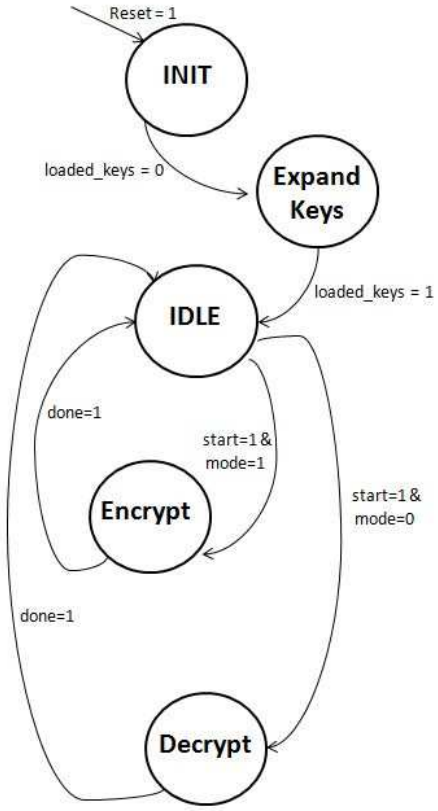


Figure 3.2: AES State Machine

The finite state machine is initialized to the INIT state upon reset and immediately goes into the Expand Keys state, because the loaded_keys signal is initialized to 0. Once expanded, the state machine moves to the IDLE state where it remains there

until a start signal is received and the mode bit is set to either a 1, for encryption, or 0 for decryption. Upon completion of either the encrypt or decrypt function, the done signal is asserted returning the state machine back to the IDLE state. The AES128 Core Logic block diagram is shown in Figure 3.3.

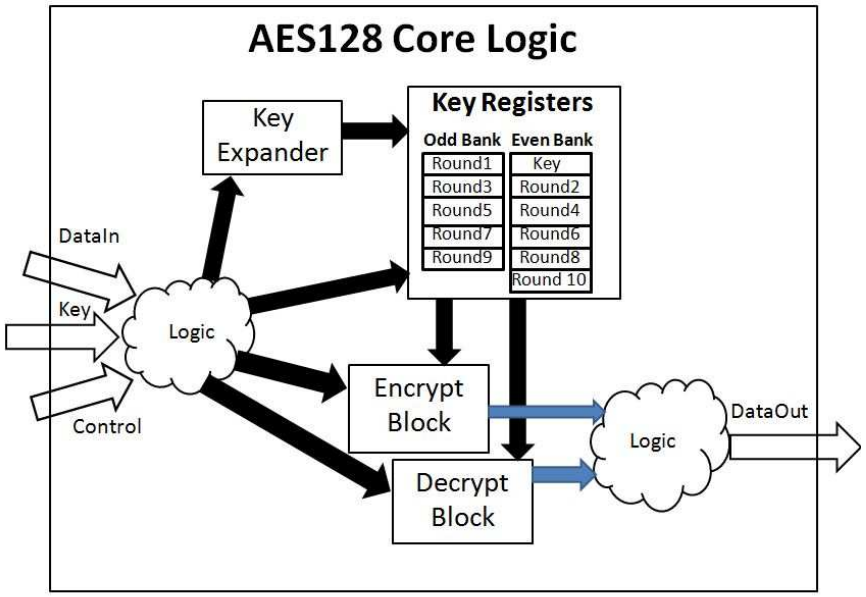


Figure 3.3: AES128 Core Logic

As in the top level block, signals pertaining to the data, key and control enter the block and some logic determines how the signals are propagated to either the Key expander, encrypt block, decrypt block and key registers. Since there is no desire to have both the encryption and decryption units working at the same time, the data input for the unit not being used is masked off so that they are not toggling at same time, this will save on power. The key expander unit performs the key expansion process discussed in Chapter 2 and is also shown in the Figure 3.4

All 10 rounds of the key are calculated and stored along with the original key into two sets of register banks, an odd and an even, both of which have a read and write port. Having access to both the odd and even key at once allows the ability to perform two rounds per clock cycle. The round logic in the encrypt and decrypt

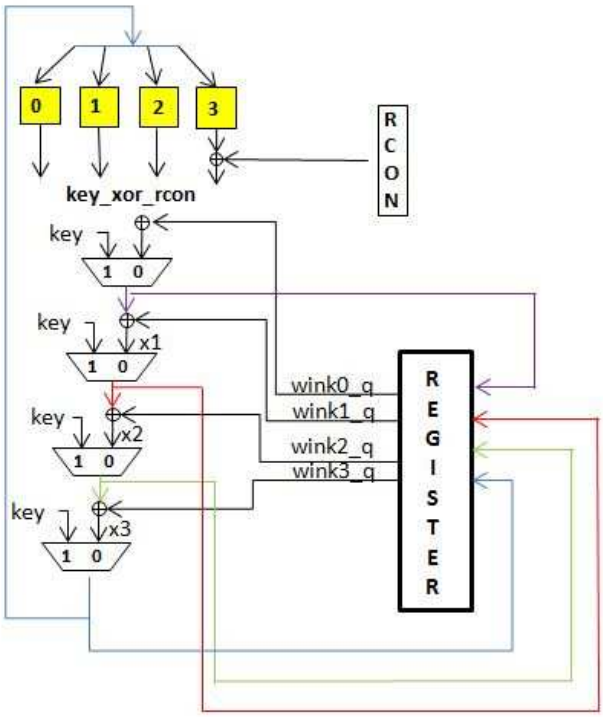


Figure 3.4: Key Expansion Unit for 128b Key

blocks are unraveled to facilitate this. The encrypt and decrypt block data flows are shown in Figures 3.5 and 3.6, respectively.

The SBOX substitution was implemented using a VHDL coded look up table, a direct mapping of the inputs and outputs of the table shown in Figure 2.5. In coding the SBOX like this, as opposed to a more defined look up table structure, allows the synthesizer to optimize the logic as best as it can. Also, the SBOX block shown in both the encrypt and decrypt data flow pictures is actually a collection of 16 copies of the SBOX stamped out to allow for each byte lookup to be done in parallel, as shown in Figure 3.7.

While this suggested approach requires more area to accommodate the SBOX copies [22], it is done so that this step can be completed in one cycle, as opposed to 16 cycles using a single SBOX table. Because two rounds are done at a time for both the encrypt and decrypt block and the SBOX is also used in the key expander,

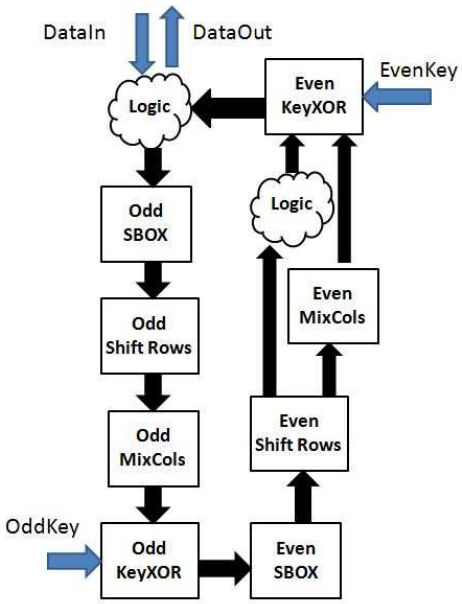


Figure 3.5: Encrypt Data Flow

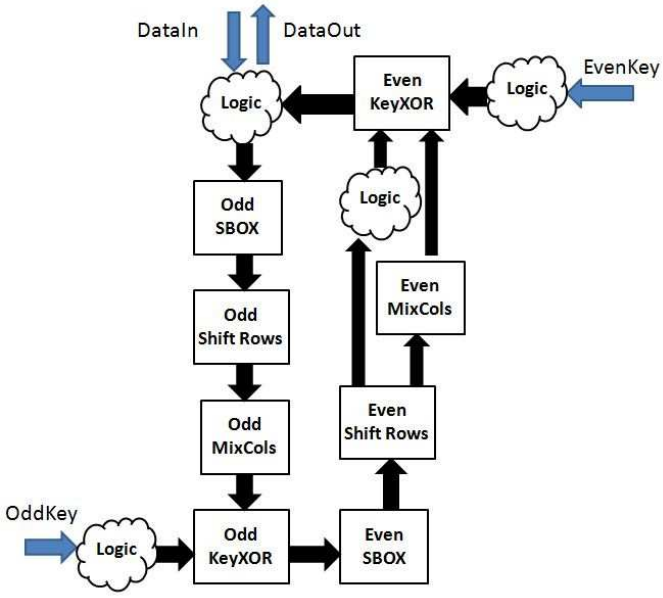


Figure 3.6: Decrypt Data Flow

the area increase is even greater than in [22]. This implementation performs basic ECB mode encryption where additional modes of operation can be supported with external logic or software.

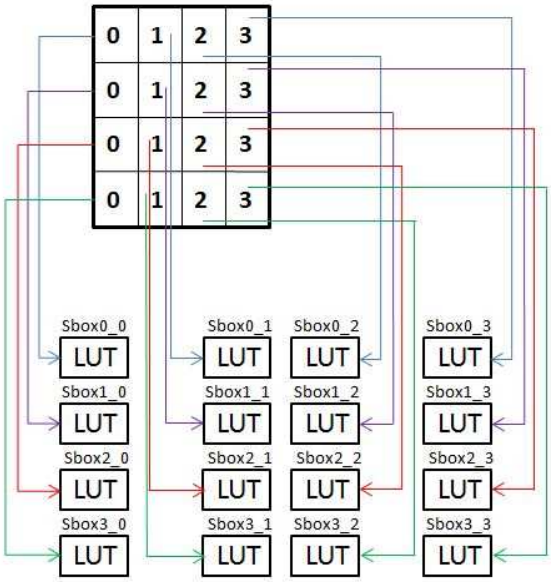


Figure 3.7: SBOX Replication

3.1.2 Simulation and Verification

A hardware test bench was written in VHDL to verify the design of this unit. The hardware test bench uses file IO to read in a text file of NIST test vectors and then simulates the units behavior using Mentor Graphics® ModelSim™. The file contains 128 different plaintext options to be encrypted with the same key, all 0's, and the corresponding ciphertexts. These vectors were obtained from the suite of AES Known Answer Test Vectors in [28] sample vectors shown in Appendix A.

The test bench first clears the important control signals and while holding the block in reset. After toggling the reset signal the test bench waits a sufficient amount of time so that the keys can be expanded and stored in the registers banks. This is referred to as the initialization time and is used as a metric for evaluating the design implementations. This value is measured in clock cycles from the time the unit comes out of reset until the time that the loaded_keys signal is high. Next, the test bench reads a line of the file, which contains the plaintext, and sets this to the data in signal and simultaneously puts a '1' on the mode and start bit. After waiting more than

7 clock cycles, the next line of the file is read in and set to a variable ciphertext so that it can be compared to the data out signal. If the two vectors are not equal, an error signal is triggered. To potentially detect multiple errors, the error signal is set to '0' after a few clock cycles. This process is repeated 127 more times to test all of the different plaintext values. A count value keeps track of how many plaintexts have been tested, when the last vector is tested a done signal is flagged to indicate the completion of the test bench.

Latency and throughput are two other metrics that are used to evaluate these implementations. Latency is defined as the time before the first encryption is available on the output; like initialization time, it is also measured in cycles. This is measured from when the first data_in vector is dropped off to when the first data_out is valid. Throughput is the rate at which data can be encrypted, number of bits per second, this refers to peak throughput assuming a full pipeline, if applicable. This is measured in bits per second and is calculated by dividing the number of bits that are being processed at a time by the product of the number of cycles the process takes and the clock rate, as shown the Equation 3.1.
In ModelSim™the critical signals are brought into the wave file after the test is run to detect if the design could properly encrypt the set of test vectors.

$$Throughput = \frac{num\_bits}{num\_cycles * clk\_freq} \tag{3.1}$$

3.1.3 Synthesis

Synthesis was done using Synopsys® Design Compiler™. A set of Tcl scripts containing the vital commands and constraints are used to facilitate the synthesis process, for this thesis, two scripts are used. The first, is an ACS setup file which defines the design library that will be used and also creates a link to the technology specific files that contain critical timing and area information about the standard cells. The

other script is used to automate the commands for pulling in the VHDL design design files, defining timing constraints, elaborating the design and reporting the necessary information about timing area and power. The area taken from the synthesis area report and is measured in square microns (um). The synthesizer uses the library files for the standard cells, which contain sizing and timing information, to aid in this calculation. This area is broken out into area of registers and area of combinational logic. The speed of the design is calculated by inverting the clock period to get the clock frequency, as shown in Equation 3.2.

$$Frequency = \frac{1}{clock\_per} \tag{3.2}$$

The timing report from the synthesis tool is used to determine if the constrained clock period was met. This report is broken into three types of paths, input to register, register to register and register to output. The critical path is calculated using the timing information for the cells along with the input, output and uncertainty timing constraints to determine the timing for each path [30].

$$clock\_per \geq input\_delay + uncertainty + critical\_path + setup\_time \tag{3.3}$$

$$clock\_per \geq uncertainty + critical\_path + setup\_time + clk\_to\_q \tag{3.4}$$

$$clock\_per \geq output\_delay + uncertainty + critical\_path + clk\_to\_q \tag{3.5}$$

After providing a clock speed timing constraint along with clock uncertainty and input and output constraints, the timing report will indicate whether the equations for input to register, register to register, and output to register, defined in Equations 3.3, 3.4, 3.5, respectively, were met and if the design could function at the assigned clock rate.

A rough power estimate of the design is also reported from the synthesizer tool which is a sum of the cell leakage power and dynamic power and is indicated in mW. The dynamic power is broken up into cell internal power and net switching power. The library files for the standard cells indicate the internal and leakage power consumption of each cell, and the net switching power is based on a user defined, or defaulted, switching activity value.

The clock period was set to 2.5ns to meet the desired operating frequency of 400 MHz. The input and output constraints were defined to be 20 percent of the clock period, or 0.5ns, and the uncertainty was estimated to be 50ps. The tool optimizes the logic accordingly to meet these timing constraints. With that said, the Synopsys® Design Compiler™tool then optimizes for area once the timing constraints are met, so the fastest design isn't always achieved unless the clock period is dialed down until failure. The reports are used to analyze the resulting netlist, to get an estimated standard cell size area or estimated power, and to see whether or not the design meets timing. After synthesis, the design is retested to insure the translation did not change the behavior or misinterpret the intended behavior of the circuitry.

3.2 AES 256BIT KEY

For AES256, the top level was altered to accommodate an input key of 256 bits. According to Figure 2.3, AES256 performs 14 rounds which requires a change to the round counter logic. These changes to the top level block diagram are highlighted in yellow in Figure 3.8. The state machine from Figure 3.2 remains unchanged. Since 14 total rounds are performed, this requires 7 iterations through the odd/even data path, as opposed to 5 iterations with the 128bit key. Also, the size of the register key banks are increased from 5 and 6 to 7 and 8 to accommodate the extra round keys. These changes are shown in yellow in Figure 3.9.

The key difference between this design and the starting one is the key expansion

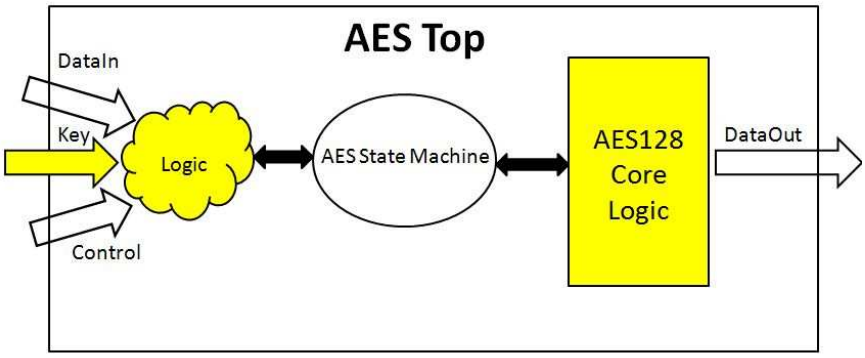


Figure 3.8: AES-256 Top

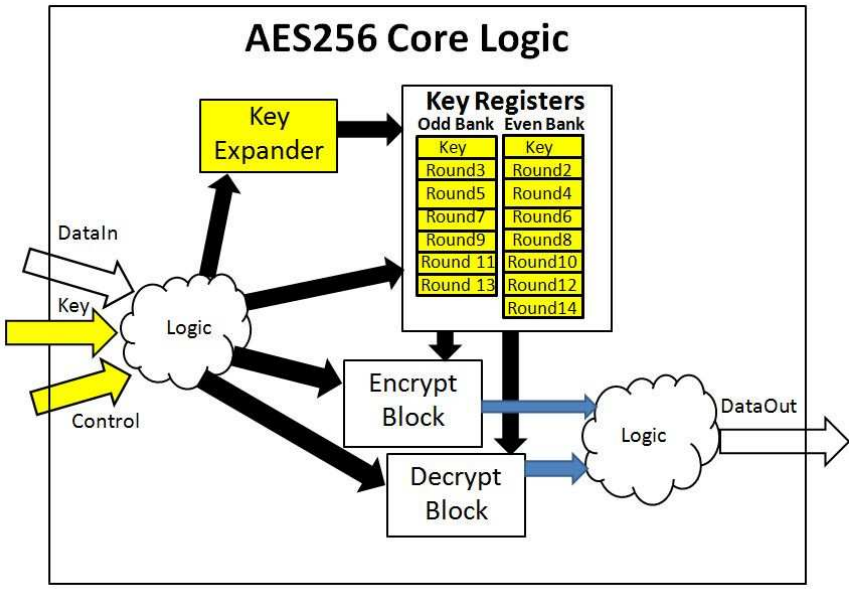


Figure 3.9: AES-256 Core Logic

block. The key expansion process different for AES256 than it was for the AES128 and is shown in Figure 3.10.

3.2.1 Simulation and Verification

For simulation, there are only slight changes to the test bench from the starting test bench. A longer key is inserted into the test bench and the time to wait to expand the key and to wait for the encryption to complete is increased to accommodate the larger key and extra rounds. The same synthesis script from the AES128 implementation

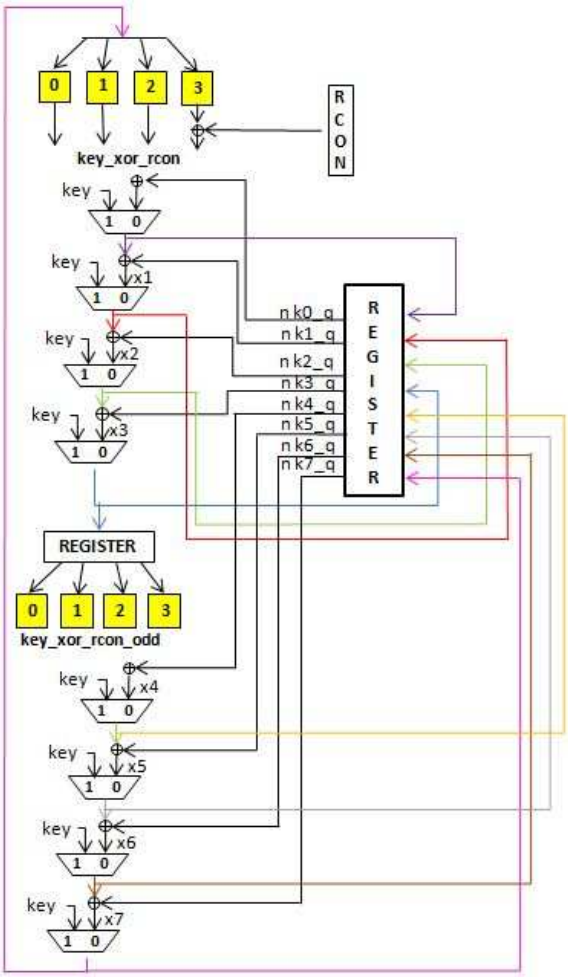


Figure 3.10: Key Expansion Unit for 256 bit

was used for this implementation.

3.3 AES256 Counting Mode

The counting mode encryption implementation is an addition to the AES256 design. Recall, the CTR mode encryption encrypts the initialization vector and not the data, therefore, the Forward Cipher logic is utilized for both the encryption and decryption process. The top level of the AES 256 counting mode block is shown in Figure 3.11. The changes from the AES256 design are highlighted in yellow. The finite state machine is simplified without the decrypt logic and is shown in Figure 3.12. The new AES256 Counting mode core level logic is shown in Figure 3.13.

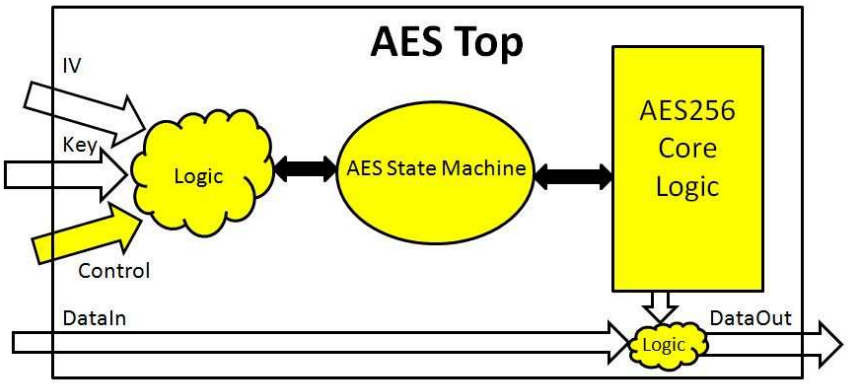


Figure 3.11: AES 256 Top Specified for Counter Mode

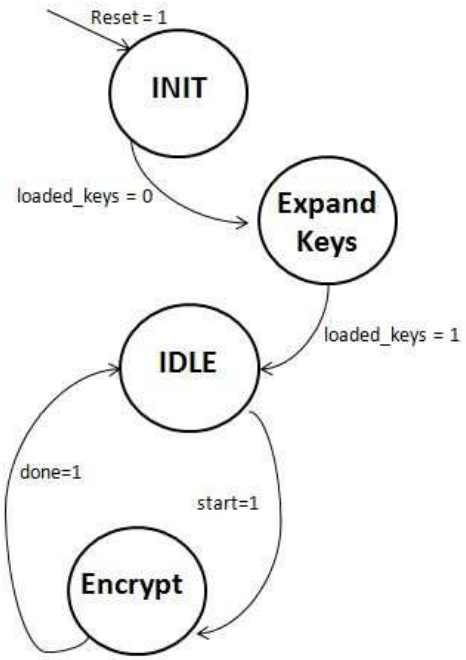


Figure 3.12: AES 256 State Machine Specified for Counter Mode

3.3.1 Simulation and Verification

The test bench used for simulation and verification remains the same as the one discussed before for AES256, because there were no CTR mode specific test vectors provided in the Known Answer Test test suite. Since the data out is a result of an XOR of the plaintext with the output of the Forward cipher, the data in vectors

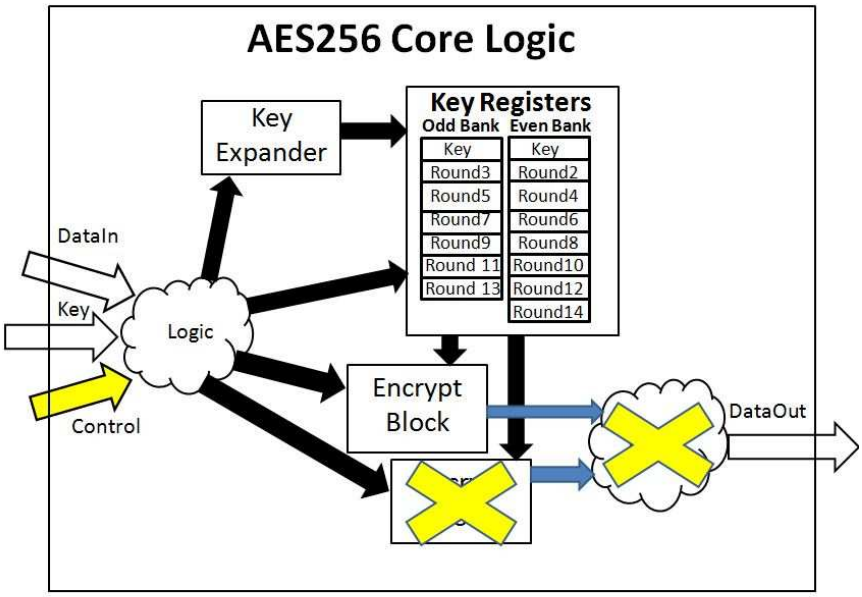


Figure 3.13: AES 256 Core Specified for Counter Mode

used in the previous simulations were fed in to the initialization vector port and the data_in vector was set to all 0s so that the data out would result in the same expected ciphertext values. Again, the same synthesis script from the AES256 implementation was used for this implementation.

3.4 Logical SBOX

Since the biggest area impact of an AES unit is typically related to its SBOX, it is important to explore the SBOX implementation. Utilizing 16 copies of the SBOX allows the entire look up for all 16 bytes to be done in a single cycle. Previously, this was coded in VHDL as a lookup table, however, this implementation explores computing the multiplicative inverse and affine transform to perform the byte substitution, as proposed in [15], [16], and [17].

In Chapter 2, Figure 2.9 shows the high level functions required to perform the multiplicative inverse using composite field arithmetic. Each of the blocks are implemented as a separate entity and the system is then build up from those sub blocks.

There are no changes made to the top level aes block diagram, or state machine, they are the same as shown in Figures 3.11 and 3.12, respectively. In Figure 3.14, the changes to the encrypt data flow are shown in yellow. The odd and even labeled SBOXes are replaced with this hierarchical multiplicative inverse and the affine transform instead of a VHDL coded look up table.

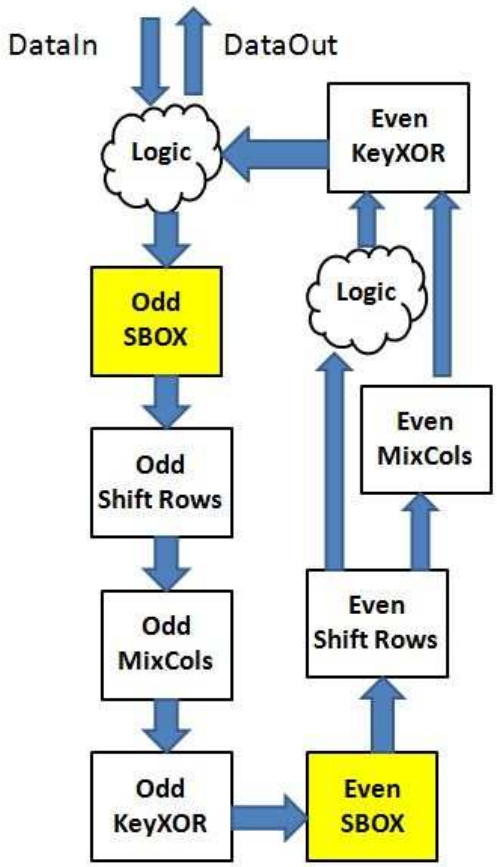


Figure 3.14: Encrypt Data Flow for Logical SBOX Implementation

3.4.1 Simulation and Verification

The same test bench used for the control AES256 Counting Mode implementation was used to simulate and verify this logical SBOX implementation. In addition, the same synthesis script for the control, AES256 Counter Mode implementation, was

used for this implementation. Due to initial results of synthesis not meeting timing, a synthesis directive (ungroup -all -flatten) was used to flatten the design.

3.5 On the Fly Key Expansion

Storing the expanded key in odd and even key bank registers for immediate accessibility during round calculation is convenient and was originally performed for this convenience at the expense of area. In [20], the key can be expanded on the fly and each round key is available as it is needed. The control design was altered to remove the expand key port as well as both register banks in order to explore this method. The top level of the AES256 Counting Mode Block with on the fly Key Expansion is shown in Figure 3.15. The finite state machine is simplified without the state used to expand the keys and is shown in Figure 3.16.Since the key expansion is happening on the fly, the key expansion block is removed from the core logic block. This is shown in Figure 3.17.

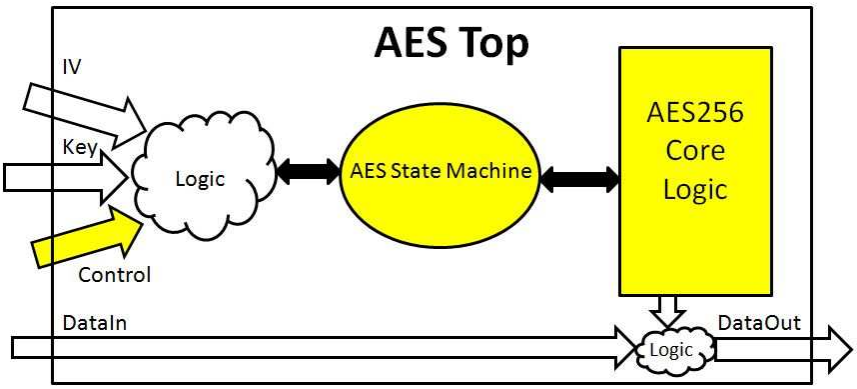


Figure 3.15: AES 256 Top with On the Fly key Expansion

The key expansion logic is is now computing the round key alongside the encrypt data flow logic.This requires two sets of the 16 SBOX's tables for the algorithm, like the previous designs, but also two sets of the four SBOX's required for the special first columns in the key expansion process.

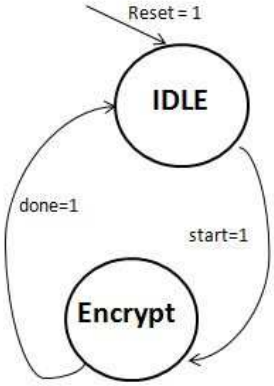


Figure 3.16: AES 256 State Machine with On the Fly key Expansion

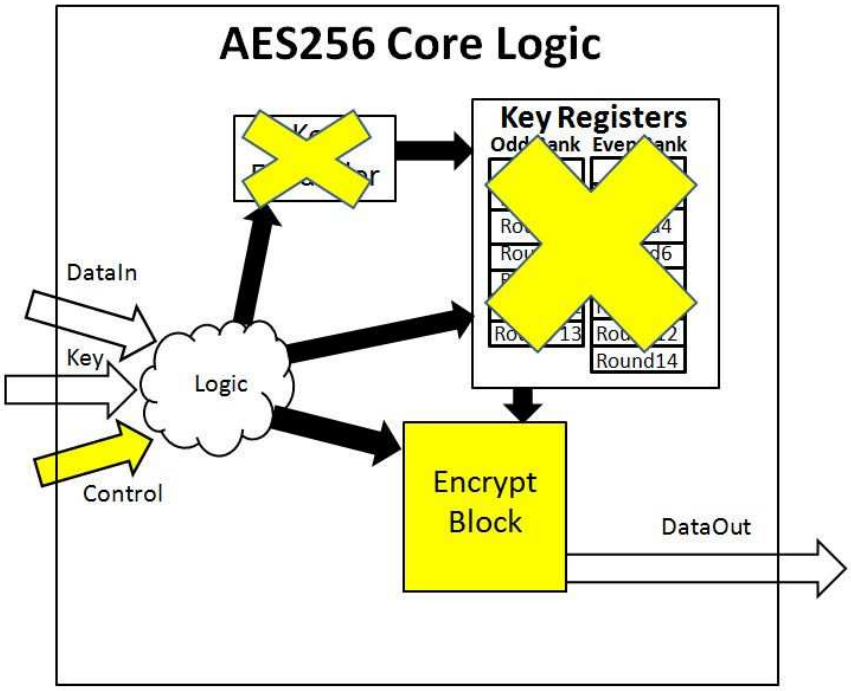


Figure 3.17: AES 256 with On the Fly key Expansion

The same test bench used for AES256 counting mode implementation was used to simulate and verify this on the fly key expansion implementation. The same synthesis script from AES256 counting mode implementation was used for this implementation.

3.6 Four Rounds Per Clock

3.6.1 Design

The control design performed two rounds per clock cycle, and required one clock cycle to load the data and 7 additional clocks for the 14 rounds of the algorithm. By combining four rounds in one clock cycle, a single 128bit AES operation can be completed in 5 total cycles. The AES top level block diagram for this implementation is shown in Figure 3.18

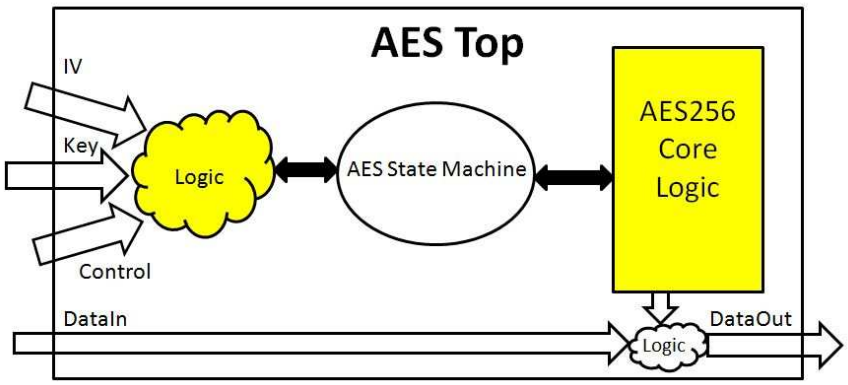


Figure 3.18: AES 256 CTR with 4 Rounds/clock Top

The finite state machine itself remains unchanged, however, the counter that manages the state machine is updated. Previously, 7 iterations through the two round logic was needed, now, only four iterations are required with the four round logic. The core logic block is also updated to contain four separate register banks for each of the four stages of round logic. This along with other changes to the AES256 core logic from the control design are highlighted in yellow in Figure 3.19. The data flow of this design is changed to unravel more round logic and can be seen in Figure 3.20.

The same test bench used from the control design implementation was used to test this implementation. The test bench could have been altered to pick up the data earlier and compare it against the results before moving on to the next one, however, since this serves the purpose of verifying the design not exercising it at peak speed,

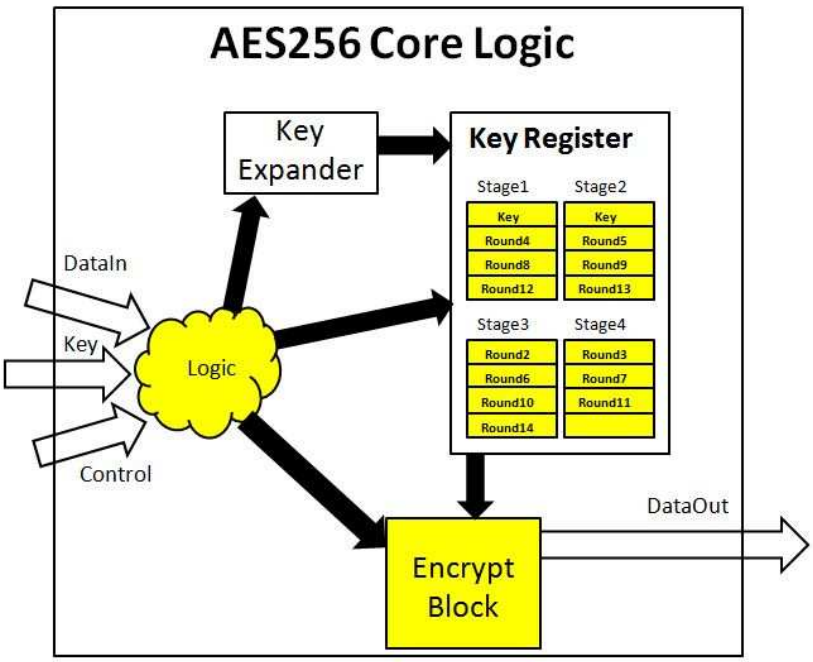


Figure 3.19: AES 256 CTR with 4 Rounds/clock Core Logic

the test bench from the AES256 counting mode implementation was left unaltered and used. The same synthesis script from control AES256 Counter mode design was used, however, the clock constraint was loosened to 3ns to fit the larger critical path of the unraveled four rounds.

3.7 Pipelined

3.7.1 Design

In the pipelined implementation, the control design of AES25 counting mode is altered to contain 14 copies of the round logic and an expanded key memory to allow access to all 14 round keys at once. The original one read one write port register bank was modified to use a one write 14 read port register bank. There is a register stage in between each round so that another initialization vector can immediately be inserted.The data_in signal was also registered to keep the right data_in value with

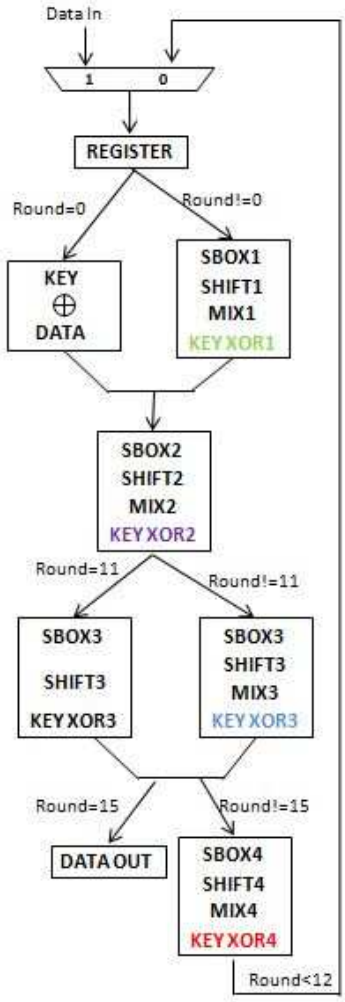


Figure 3.20: AES 256 CTR with 4 Rounds/clock Data Flow

the corresponding initialization value. A data_valid signal was also added to indicate whether the data_out was valid.These top level block diagram for the pipelined design is shown in Figure 3.21. The changes to the core logic can be shown in yellow in Figure 3.22.

There is an initial delay of 14 clock cycles before the first result is seen on the data out port, but each data out after that is seen on every clock cycle. A data valid signal was added to the design to indicate when the data_out signal has valid data. A figure depicting the data flow of this implementation is shown in Figure 3.23.

The test bench for this implementation is split into two separate operations, one

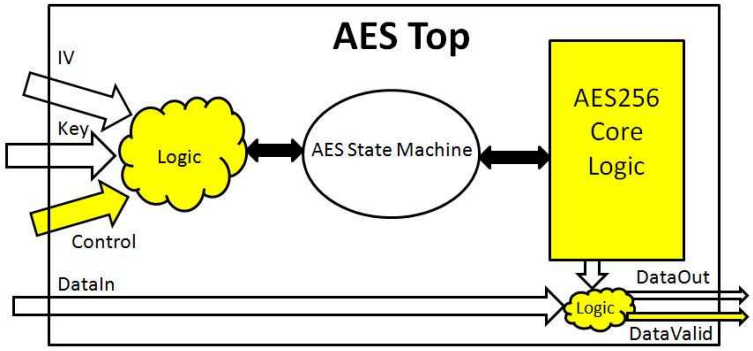


Figure 3.21: AES 256 Pipelined Top

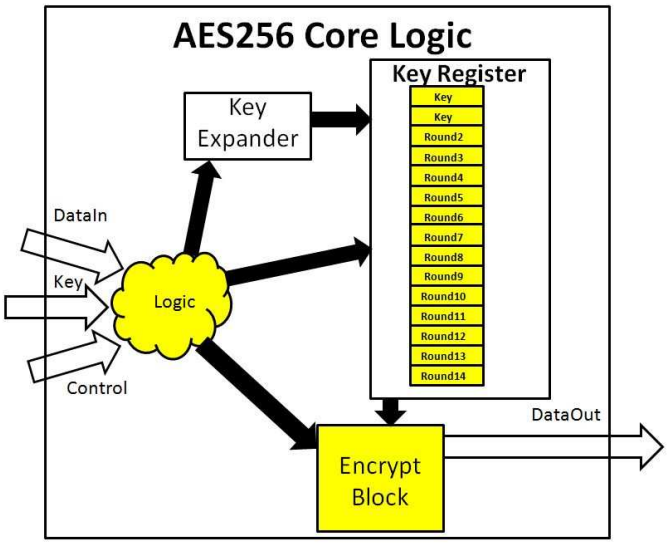


Figure 3.22: AES 256 Pipelined Core Logic

to feed the test vectors into the the AES unit and another to collect the data on the output and compare it against the known ciphertext values. This was done to test the pipeline structure fully and see that once the pipeline was filled, data could be received every clock cycle. If the data out of the AES unit did not match the expected ciphertext from the file, an error signal was flagged.

The same synthesis script from the AES256 Counter mode implementation was used however, the clock period timing constraint was reduced because the register to register path is much shorter than it was for the control design. The input and output constraints were also tightened to the same percentage of the clock period that was

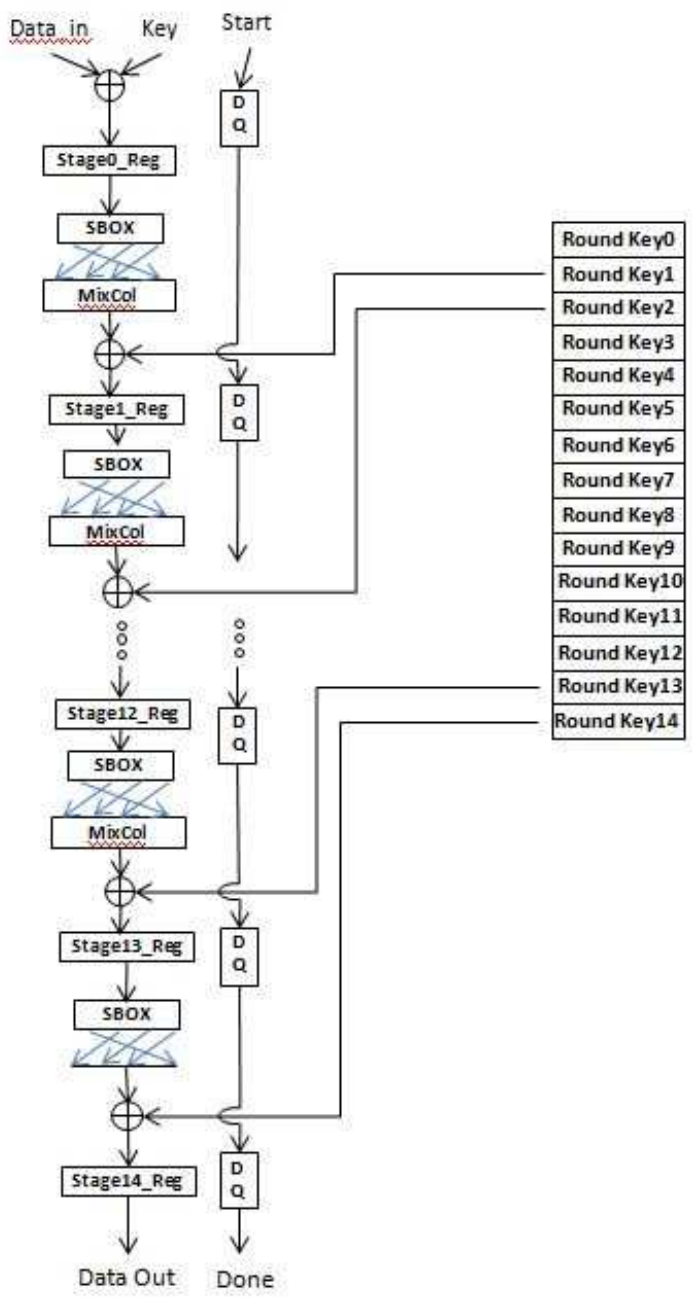


Figure 3.23: AES 256 Pipelined Data Flow

used for the control design.

3.8 Summary

In this chapter the details of the seven different AES implementations were described. The methodology for verifying the design correctness and also the procedure for collecting the report data pertaining to each implementation was explained. The next section the results of these implementations will be discussed.

CHAPTER 4

RESULTS

In this chapter, the results of each design implementation that was discussed in the methodology section are addressed. The designs are evaluated according to the metrics discussed in the previous chapter and a summary of each designs' resulting information will be given at the end of each section. The last section of this chapter will contain a table summarizing the results of all the designs and discuss the tradespace.

4.1 AES128

The AES128 implementation was described in VHDL and tested in ModelSim^{TM}using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 12 cycles and the latency was 6 cycles. The area of the design was 143,085um, registers took up 21,379um/143,085um, or 15 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 2.5ns clock. Therefore, the speed of this design was calcuated to be 1/2.5ns or 400MHz. The throughput was calculated to be 8.5Gb/s, 128 bits every 6 cycles at a clockperiod of 2.5ns 128/(6*2.5ns). The power was reported as 5.54mw cell internal power, 4.26mW net switching power, and 60uW of cell leakage power for a total of 9.87mW.

Table 4.1: AES128 Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES128	143,085	400	9.87	12	6	8.5

4.2 AES256

The AES256 implementation was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 16 cycles and the latency was 8 cycles. The area of the design was 158,950um, registers took up 29,049um/158,950um, or 18 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 2.5ns clock. Therefore the speed of this design was calcuated to be 400MHz. The throughput was calculated to be 6.4Gb/s, 128bits every 8 cycles at a clockperiod of 2.5ns 128/(8*2.5ns). The power was reported as 6.45mw cell internal power, 4.43mW net switching power, and 66uW of cell leakage power for a total of 10.95mW.

Table 4.2: AES256 Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256	158,950	400	10.95	16	8	6.4

4.3 AES256 Counting Mode

The AES256 counting mode implementation was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The

testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 16 cycles and the latency was 8 cycles. The area of the design was 94,622um, registers took up 27,816um/94,622um, or 29 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 2.5ns clock. Therefore the speed of this design was calcuated to be 400MHz. The throughput was calculated to be 6.4Gb/s, 128bits every 8 cycles at a clockperiod of 2.5ns 128/(8*2.5ns). The power was reported as 5.45mw cell internal power, 3.53mW net switching power, and 34uW of cell leakage power for a total of 9.01mW.

Table 4.3: AES256 Counting Mode Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256_CTR	94,622	400	9.01	16	8	6.4

4.4 AES256 Counting Mode Logical SBOX

The AES256 Counting Mode implementation with a logical SBOX was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 16 cycles and the latency was 8 cycles. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was not able to meet timing with a 2.5ns clock, it violated by 0.19ns which means it could only run at a period of 2.61ns. Therefore the speed of this design was calcuated to be 1/2.61ns or 380MHz. A large difference between the logical and look up table SBOX implementation is the large amount of hierarchy in the logical approach, as described in the background section. The

synthesizer does not optimize across hierarchical boundaries very well; because of this a synthesis option was used to flatten the design (ungroup -all -flatten).

After this flatten option was inserted, the area of the design was 118,109um, registers took up 27,507um/118,109um, or 23 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 2.5ns clock. Therefore the speed of this design was calcuated to be 400MHz. The throughput was calcuated to be 6.4Gb/s, 128bits every 8 cycles at a clockperiod of 2.5ns 128/(8*2.5ns). The power was reported as 7.88mw cell internal power, 4.37mW net switching power, and 84uW of cell leakage power for a total of 12.33mW.

Table 4.4: AES256 Counting Mode Logical SBOX Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256_CTR_logicsbox	118,109	400	12.33	16	8	6.4

4.5 AES256 Counting Mode On the Fly Key Expansion

The AES256 counting mode on the fly key expansion implementation was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 0 cycles and the latency was 8 cycles. The area of the design was 64,309um, registers took up 8,054um/64,309um or 12.5 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 2.5ns clock. Therefore the speed of this design was calcuated to be 400MHz. The throughput

was calculated to be 6.4Gb/s, 128bits every 8 cycles at a clock period of 2.5ns or 128/(8*2.5ns). The power was reported as 8.24mW cell internal power, 6.86mW net switching power, and 21uW of cell leakage power for a total of 15.12mW.

Table 4.5: AES 256 Counting Mode On the Fly Key Expansion Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256_CTR_OTF	64,309	400	15.12	0	8	6.4

4.6 AES256 Counting Mode 4 Rounds per Clock

The AES256 counting mode 4 Rounds per Clock implementation was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 8 cycles and the latency was 5 cycles. The area of the design was 223,743um, registers took up 29,066um/223,743um, or 13 percent. With an input constraint of 0.5ns, output constraint of 0.5ns, and a clock uncertainty of 0.05ns the design was unable to meet timing with a 2.5ns clock. Therefore the clock period was increased to 3ns and the input and output constrants were adjusted to 0.6ns. The design was able to meet timing at this clock rate so the speed of this design was calcuated to be 333MHz. The throughput was calculated to be 8.5Gb/s, 128bits every 5 cycles at a clock period of 3ns or 128/(5*3ns). The power was reported as 9.69mW cell internal power, 7.3mW net switching power, and 149uW of cell leakage power for a total of 17.13mW.

Table 4.6: AES256 Counting Mode 4 Rounds per Clock Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256_CTR_4r	223,743	333	17.13	8	5	8.5

4.7 AES256 Pipelined Counting Mode

The AES256 pipelined counting mode implementation was described in VHDL and tested in ModelSim using the test bench structure described in the methodology section. The testbench ran through all 128 vectors and passed. The initialization time was seen in the simulation to be 9 cycles and the latency was 14 cycles. The area of the design was 471,002um, registers took up 43,131um/471,002um, or 9 percent. With an input constraint of 0.16ns, output constraint of 0.16ns, and a clock uncertainty of 0.05ns the design was able to meet timing with a 0.82ns clock. Therefore the speed of this design was calcuated to be 1.22GHz. The throughput was calculated to be 156Gb/s, 128bits every cycle at a clock period of 0.82ns or 128/(1*0.82ns). This assumes a full pipeline so that the encrypted data out is seen every cycle. The power was reported as 163.57mW cell internal power, 134.30mW net switching power, and 271uW of cell leakage power for a total of 298.1mW.

Table 4.7: AES256 Pipelined Counting Mode Results

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES256_CTR_pipelined	471,002	1220	298.1	9	14	156

4.8 Summary

A summary of the resulting speed area power and throughput for each implementation is shown in Table 4.8. The results show that the AES256CTROTF design was the smallest of all the implementations at 64,302um which was 32 percent smaller than the control design at 94,622um. The largest design was the pipelined implementation which was 470,808um or 398 percent larger. The 4round per clock cycle design is approximately double the size of the control, which was expected since twice the logic was unraveled in the 4 round implemntation. Surprisingly, the logicsbox implementation was larger than the control which was unexpected due to the research presented in [15] and [16], which suggested a smaller area for the combinational SBOX approach.

Table 4.8: Summary of AES Implentation Comparison of Speed Area Power

	Area [um]	Speed [MHz]	Power [mW]	Init [cycles]	Latency [cycles]	Throughput [Gb/s]
AES128	143,085	400	9.87	12	6	8.5
AES256	158,950	400	10.95	16	8	6.4
AES256_CTR	94,622	400	9.01	16	8	6.4
AES256_CTR_logicsbox	118,109	400	12.33	16	8	6.4
AES256_CTR_OTF	63,420	400	17.28	0	8	6.4
AES256_CTR_4r	223,743	333	17.13	8	5	8.5
AES256_CTR_pipelined	471,002	1220	298.1	9	14	156

The fastest design in terms of throughput and speed is the pipelined implementation which was able to run at 1.22GHz and can provide a throughput of 156Gb/s. The high data rate comes at a price of a 14 clock cycle inital latency. Next the 4

round per clock cycle design was able to achieve a 8.5Gb/s which was a higher rate than the control through of 6.4Gb/s while running at 333MHz which was slower than the control clock speed of 400MHz. This of course is due to the reduced number of clock cycles that it takes to complete an AES operation when compared with the control design. The on the fly key expansion implementation which runs at the same clock rate and has the same throughput as the control design however there is no initialization time required to expand the key.

The control design burned the least amount of power out of all the design implementations. It is no surprise that the pipelined implementation which was the largest and fastest also burned the most power at 298.1mw, this is significantly more than the control design power of 9.01mw. The on the fly key expansion implementation burned 17.28mW, which was more power than the control design that used stored round keys. This makes sense because more gates are toggling to expand the key each time it is used.

CHAPTER 5

CONCLUSIONS

The aim of this thesis has been to understand the AES algorithm and possible end applications in order to explore the impacts that optimizing different features has on the overall system performance parameters. Multiple resources in this research field have identified features of interest and discussed their impact on one or two of the design trade spaces, however, a single comparative analysis was lacking.

After providing a thorough background on the algorithm, different modes of operation a brief overview of the VLSI design flow process, this thesis has explored six different AES features; key size, mode specificity, round key storage, round unraveling, SBOX implementation, and pipelining. Using computer aided design tools, designs were created according to the AES specification, verified using NIST test vectors, and the appropriate data was collected to further investigate the speed area and power implications.The detailed explanation of the design and methodology allows fellow designers to replicate and further the research discussed in this thesis. The summarized view of the resulting design metrics allow readers to quickly analyze how each of the six features impacts speed power and area on the 65nm process.

After reviewing the results designers may lean toward inserting some features in their AES hardware implementation while avoiding others. Networking applications concerned with high data transmission rates may look at Table 4.8 and choose to adopt the AES256_CTR_pipelined design approach for the high speed and throughput capability, if they don't mind area overhead or paying initial latency. Portable electronics concerned with battery life like wearable monitors and cell phones are op-

timizing for power, in this case the control design, AES256_CTR, would be of interest due to its low power operation. Other portable electronics that are emphasize small form factor or designs where a large number of AES units are needed will be concerned with the compactness of the implementation. Here the AES256_CTR_OTF design would be appealing since it resulted in the smallest area. Military systems dealing with national security and protection of classified data will want to focus on the designs described in this thesis that utilize a 256bit key and counting mode specificity. This is suggested to maximize protection against brute force attacks and mitigate the image encryption vulnerabilities of the basic ECB mode. Perhaps systems concerned with response time would desire encryption encryption operations quickly under different keys, this may require a combination of approaches to achieve both a low latency and initialization time. By combining AES256_CTR_OTF with AES256_CTR_4r a new design could result in a 0 cycle initialization time and a low cycle latency.

Many results followed the initial intuition. For example the AES256_CTR_4r was approximately twice as big and burned twice the power compared to the control design AES256_CTR, this is because twice the logic was unraveled. The AES256_OTF design is smaller and has half the number of register elements than the control because the all the expanded keys are not stored in registers. The AES256_CTR_pipelined design would be much faster than the control once the pipeline is full and the area would be much larger makes sense because all 14 rounds of logic are expanded. However, the results of logical SBOX design did not align with the initial expectations. In [15] [16] this combinatorial approach to the SBOX suggested smaller area than typical ROM based design. When analyzing the difference between the control design and this one, it was surprising to learn that the area was bigger. The reasoning for difference in results could be that [16] was using an FPGA where ROM structure is different. Another reason could be that the coding style of the design in this thesis did not

build a traditional ROM based Look up table and instead the hard coded table of values was compressed and the synthesizer did a better job optimizing logic.

Encryption is vital to the electronics community where sensitive data is constantly being transmitted and stored. Careful focus must be placed on efficiently performing this task to meet the demanding system requirements. This thesis has provided a hardware encryption designer a reference for evaluating design feature effects on system performance parameters to assist them in their task.

5.1 Future Work

This thesis provides a good comparative analysis of these implementation features, however, the analysis is done post synthesis. It may be valuable to continue these designs through the physical layout and complete place and route to get a more accurate view of how these features affect area speed and power when things like long wires and buffers are inserted. The speed of large designs like the AES256_CTR_pipelined may decrease as routing challenges are introduced.

This work investigated a combinatorial logic SBOX approach by implementing the multiplicative inverse using composite field arithmetic according to the designs described in [15][16][17]. Future work could investigate utilizing other techniques to implement multipliers lower GF fields like the Mastrovito multiplier.

BIBLIOGRAPHY

[1] C. Denison, *Comcast Encrypts All Cable Channels, Requires All Customers to Use a Set-Top Box.* Digital Trends , 16 April 2013. http://www.digitaltrends.com/home-theater/comcast-encrypting-all-cable-channels/

[2] K. Zetter, *Target Got Hacked Hard in 2005. Here's Why They Let It Happen Again.* Wired , 17 January 2014. http://www.wired.com/2014/01/target-hack/

[3] K. Moskvitch, *Are drones the next target for hackers?.* BBC , 6 February 2014. http://www.bbc.com/future/story/20140206-can-drones-be-hacked

[4] S. Goldstein, *Chinese hackers stole F-35 fighter jet blueprints in Pentagon hack, Edward Snowden documents claim.* New York Daily News, 20 January 2015. http://www.nydailynews.com/news/national/snowden-chinese-hackers-stole-f-35-fighter-jet-blueprints-article-1.2084888

[5] D. Sanger, N. Perlroth, M. Shear *Attack Gave Chinese Hackers Privileged Access to U.S. Systems.* New York Times, 20 June 2015. http://www.nytimes.com/2015/06/21/us/attack-gave-chinese-hackers-privileged-access-to-us-systems.html?_r=0

[6] FIPS, *Specification for the Advanced Encryption Standard (AES).* Federal Information Processing Standards Publication 197, 2001.

[7] J. Daemen, V. Rijmen, *The design of Rijndael: AES the Advanced Encryption Standard.* Springer-Verlag, 2002.

[8] H.O Alanazi, B.B Zaidan, A.A Zaiden, H.A Jalab, M Shabbir Y.Al-Nabhani, *New Comparative Study Between DES, 3DES and AES within Nine Factors.* Journal of Computing, Volume2 Issue3, March2010, ISSN2151-9617. https://sites.google.com/site/journalofcomputing/

[9] P. Bulman, *Commerce Department Announces Winner of Global Information Security Competition.* NIST , 2 October 2000. http://www.nist.gov/public_affairs/releases/g00-176.cfm

[10] F.K. Gurkaynak, A.Burg,N.Felber, W.Fichtner, *A 2Gb/s Balanced AES Crypto-Chip Implementation.* GLSVLSI 2004, 26-28 April 2004.

[11] P.Saravanan, N. RenukaDevi, G. Swathi, Dr.PKalpana, *A High- Throughput ASIC Implenmentation of Configurable Advanced Encryption Standard(AES) Processor.* IJCA Special Issue on "Network Security and Cryptography" NSC, 2011.

[12] I.Verbauwhede, P. Shaumont, H. Kuo, *Design and Performance Testing of a 2.29-GB/s Rijndael Processor.* IEEE Journal of Solid State Circuits, Vol.28 No.3, March 2003.

[13] S.Lin, C.Huang, *A High-Throughput Low-Power AES Cipher for Network Applications.* IEEE, 2007.

[14] A.Page, P.V. Sriniwas Shastry, *AES-128 Key Expansion with LUT and OTF S-Box.* IJCTEE Volume 4 Issue 3 , June 2014.

[15] P.V. Sriniwas Shastry, A.Agnihotri, D.Kachhwaha, J, Singh *A Combintational Logic Implementation of S-Box of AES.* IEEE, 2011.

[16] E.Mui, *Practical Implementation of Rijndael S-Box Using Combinational Logic.* Accessed: October 2015, http://www.xess.com/static/media/projects/Rijndael_SBox.pdf

[17] J.Wolkerstorfer, E. Oswald, M.Lamberger *An Asic Implementation of the AES SBoxes*. B.Preneel(Ed.):CT-RSA 2002, LNCS 2271, pp 67-78, 2002 Springer-Verlag Berlin Heidelberg, 2002.

[18] A.Satoh, S. Morioka, K.Takano, S. Munetoh *A Compact Rijndael Hardware Architecture with S-Box Optimization*. C.Boyd(Ed.):ASIACRYPT 2002, LNCS 2248, pp 239-254, 2001 Springer-Verlag Berlin Heidelberg, 2001.

[19] Y.C.Mei, S. Z. M. Naziri, *The FPGA Implementation of Multiplicative Inverse Value of GF(2^8) Generator using Extended Euclidean Algorithm(EEA) Method for Advanced Encryption Standard (AES) Algorithm*. IEEE, 2011.

[20] P.C.Liu, H.C. Chang, C.Y.Lee, *A 1.69 Gb/s/ Area-Efficient AES Crypto Core with Compact On-the-fly Key Expansion Unit*. IEEE, 2009.

[21] A.Hodjat, I.Verbauwhede, *Area-Throughput Trade-Offs for Fully Pipelined 30 to 70 Gbits/s AES Processors*. IEEE, 2006.

[22] S.Mangard, M.Aigner, *A Highly Regular and Scalable AES Hardware Architecture*. IEEE, 2003.

[23] N. Sklavos, O.Koufopavlou, *Architectures and VLSI Implementations of the AES-Proposal Rijndael*. IEEE, 2002.

[24] C.Benvenuto, *Galois Field in Cryptography*. May 2012, https://www.math.washington.edu/ morrow/336_12/papers/juan.pdf

[25] Wikipedia, *Block Cipher Mode of Operation*. Wikipedia, Accessed: October 2015, https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation .

[26] M. Dworkin, *Recommendation for Block Cipher Modes of Operation: Methods and Technigues*. NIST Special Publication 800-38A, 2001 .

[27] H. Lipmaa, P.Rogoway, D.Wagner, *CTR-Mode Encryption.* Comments to NIST concerning AES Modes of Operations, 2000 .

[28] L.E. Bassham, *The Advanced Encryption Standard Algorithm Validation Sutie (AESAVS).* NIST Information Technology Laboratory Computer Security Division, November2002 .

[29] N. Daras *Applications of Mathematics and Informatics in Military Science Volume71 of Springer Optimization and Its Applications.* Springer Science and Business Media, Page 207, 2012 .

[30] M.Mano, M.Ciletti, *Digital Design With an Introduction to the Verilog HDL.* Prentice Hall, 2013.

APPENDIX A

KAT Vectors from NIST

```
80000000000000000000000000000000
ddc6bf790c15760d8d9aeb6f9a75fd4e

c0000000000000000000000000000000
0a6bdc6d4c1e6280301fd8e97ddbe601

e0000000000000000000000000000000
9b80eefb7ebe2d2b16247aa0efc72f5d

f0000000000000000000000000000000
7f2c5ece07a98d8bee13c51177395ff7

f8000000000000000000000000000000
7818d800dcf6f4be1e0e94f403d1e4c2

fc000000000000000000000000000000
e74cd1c92f0919c35a0324123d6177d3

fe000000000000000000000000000000
8092a4dcf2da7e77e93bdd371dfed82e

ff000000000000000000000000000000
49af6b372135acef10132e548f217b17

ff800000000000000000000000000000
8bcd40f94ebb63b9f7909676e667f1e7

ffc00000000000000000000000000000
fe1cffb83f45dcfb38b29be438dbd3ab

ffe00000000000000000000000000000
0dc58a8d886623705aec15cb1e70dc0e

fff00000000000000000000000000000
c218faa16056bd0774c3e8d79c35a5e4

fff80000000000000000000000000000
047bba83f7aa841731504e012208fc9e

fffc0000000000000000000000000000
dc8f0e4915fd81ba70a331310882f6da

fffe0000000000000000000000000000
1569859ea6b7206c30bf4fd0cbfac33c
/
--(DOS)---  AES_KAT_Vectors_key0.txt   Top L45    (Text)----------
```

Figure A.1: Sample of 128 KAT Vectors

VITA

Andrea LaPiana

Candidate for the Degree of

Masters of Science

Dissertation: A COMPARATIVE ANALYSIS OF DIFFERENT AES IMPLEMENTATIONS FOR 65NM TECHNOLOGIES

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Smithtown, NY, USA on August 20, 1988.

Education:

Received the B.S. degree from Binghamton University, Binghamton, NY, USA, 2010, in Electrical Engineering
Completed the requirements for the degree of Masters of Science with a major in Electrical Engineering Oklahoma State University in December, 2015.

Experience:

2010-2011 Field Applications Engineer, Intersil, Hauppauge, NY
2011-2015 Systems Engineer, Harris Corporation, Rome,NY