ABSTRACT

HIGH SPEED VLSI IMPLEMENTATION OF THE RIJNDAEL ENCRYPTION ALGORITHM

Sever, Refik

M.S., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Murat Aşkar

September 2003, 80 pages

This thesis study presents a high speed VLSI implementation of the Rijndael Encryption Algorithm, which is selected to be the new Advanced Encryption Standard (AES) Algorithm. Both the encryption and the decryption algorithms of Rijndael are implemented as a single ASIC. Although data size is fixed to 128 bits in the AES, our implementation supports all the data sizes of the original Rijndael Algorithm. The core is optimised for both area and speed. Using 149K gates in a 0.35-µm standard CMOS process, 132 MHz worst-case clock speed is achieved yielding 2.41 Gbit/s non-pipelined throughput in both encryption and decryption.

The design has a latency of 30 clock periods for key expansion that takes 228 ns for this implementation. A single encryption or decryption of a data block requires at most 44 clock periods. The area of the chip is 12.8 mm^2 including the pads. 0.35-µm Standard Cell Libraries of the AMI Semiconductor Company are used in the implementation. The literature survey revealed that this implementation is the fastest published non-pipelined implementation for both encryption and decryption algorithms.

Keywords: AES, Rijndael Algorithm, ASIC, Encryption.

ÖZ

RIJNDAEL SİFRELEME ALGORİTMASININ YÜKSEK HIZLI TÜMDEVRE GERÇEKLEŞTİRİMİ.

Sever, Refik

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Murat Aşkar

Eylül 2003, 80 sayfa

Bu tezde, İleri Şifreleme Standardı (AES) olarak seçilen Rijndael Şifreleme Algoritması'nın yüksek hızlı tümdevre gerçekleştirimi sunulmaktadır. Rijndael Algoritması'nın şifreleme ve deşifreleme kısımları, yarı-özel CMOS tasarım teknikleri kullanılarak tek bir yonga üzerinde Uygulamaya Özel Tümdevre olarak gerçekleştirilmiştir. Veri uzunluğu İleri Şifreleme Standardı'nda 128 olarak sabitlenmesine rağmen, bu gerçekleştirim orjinal Rijndael Algoritması'ndaki tüm veri uzunluklarını desteklemektedir. Tasarlanan yonga, alan ve hız için optimize edilmiştir. Tümdevre, 0.35-µm standard CMOS teknolojisinde 149000 kapı kullanılarak gerçekleştirilmiş ve en kötü çalışma koşullarında, 132 MHz saat hızına ve saniyede 2.41 Gbit şifreleme ve deşifreleme işlem yoğunluğuna, boru hattı mimarisi kullanılmadan ulaşılmıştır. Tasarımda anahtar açılımı 30 saat periyodu (228 ns) sürmektedir. Bir veri bloğunun şifrelenmesi ya da çözülmesi en fazla 44 saat periyodunda tamamlanmaktadır. Tümdevre, 12.8 mm^2 alan kaplamıştır. Gerçekleştirimde AMI Semiconductor Firması'nın 0.35-µm Standard Hücre Kütüphaneleri kullanılmıştır. Bilgimize göre, bu çalışmada sunulan gerçekleştirim şu ana kadar yayınlanmış, boru hattı mimarisi kullanılmadan yapılan en hızlı Rijndael gerçekleştirimidir.

Anahtar Kelimeler: AES, Rijndael Algoritmasi, Uygulamaya Özel Tümdevre, Şifreleme.

ACKNOWLEDGMENTS

I am very grateful to Assist. Prof. Dr. Y. Çağatay Tekmen for his endless support and encouragement throughout this study. He has always been patient and motivative at all the stages of this thesis.

I would like to express my appreciation to my supervisor Prof. Dr. Murat Aşkar for his guidance and helpful comments in the development of this thesis.

I would like to thank to Assoc. Prof. Dr. Melek Yücel for starting the RSA Project in TÜBİTAK-ODTÜ-BİLTEN and giving inspiration about my thesis subject.

I would also like to thank to all of my colleagues in TÜBİTAK-ODTÜ-BİLTEN; especially to my coordinator Neslin İsmailoğlu for sharing her deep experience about VLSI Design. I also wish to thank to my sincere friends Oğuz Benderli and Soner Yeşil for their helps and interests throughout my thesis.

And finally, I would like to express very special gratitudes to my dear wife Aslı, for her patience and continuous support, and my dear family for their love and encouragement during my studies.

To my dear wife, Aslı.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Cryptography is a Greek word that means to write secrets. Throughout history, hiding secrets has played an important role in people's lives. Especially the importance of encryption for military purposes has been a strong driving force for cryptography. Cryptanalysis is the art of breaking into secure communications and understanding their contents. The combination of the cryptography and the cryptanalysis is commonly referred to as cryptology. The ongoing competition between code writers and code breakers has resulted in many advances in the field of cryptology.

The growing of the Internet at the last decade has led to an increase in the importance of cryptography. Internet applications such as electronic commerce and electronic communication have made cryptography an essential tool. In fact, cryptography is used in a wide range of applications including digital signature, authentication and commanding purposes. In these applications, two forms of cryptography are commonly used. These are symmetric key algorithms and public key algorithms, also referred to as ciphers. In symmetric key ciphers, one secret key is used for both encryption and decryption. In public key ciphers, two keys are used. The first one, known as the public key, is not secret and openly available and is used for encryption. The second key, known as the private key is secret and is used for decryption.

Public key ciphers are generally used in media where key distribution is a problem. A very large number of users communicate with each other through the Internet. Each user has a public key and a private key. If a user wants to communicate with another in a secure way, he can encrypt his message with the public key of the receiving user. The user who receives the message decrypts it using his private key. No one other then the intended receiver can decrypt the message. Public key ciphers are slow and therefore are generally not used to encrypt whole messages. They are only used to encrypt secret keys, whereas the message is encrypted using much faster symmetric key ciphers.

Data Encryption Standard (DES) Algorithm [1] had been the standard symmetric key algorithm for the government of the United States since 1977. However, the enormous increase in computational power has now made the DES algorithm obsolete. DES cracker hardware [2] is now available to easily break the algorithm in nearly 20 hours. Therefore, to satisfy the security needs of the community at large, the National Institute of Standards and Technology (NIST) [3] held a competition to determine the new standard. In October 2000, after 3 years of long competition between 15 candidates, the NIST selected the Rijndael Algorithm [4] as the new Advanced Encryption Standard (AES) Algorithm to replace DES.

In high-speed data communication systems, fast data encryption and decryption is very important. In this thesis, a high speed VLSI implementation of the Rijndael Algorithm is presented. Both the encryption and the decryption algorithms of Rijndael are implemented as a single ASIC.

Chapter 2 describes the mathematical details of the Rijndael Algorithm. In Chapter 3, the hardware implementations of the Rijndael Algorithm reported in related literature are discussed. Chapter 4 describes the proposed architecture and presents the hardware implementation and the simulation results. In Chapter 5 a summary of the results and possibilities for future studies are given.

CHAPTER 2

RIJNDAEL ALGORITHM

2.1 Introduction

This section explains the Rijndael Algorithm. The first part discusses the mathematical background used in the algorithm. The encryption and decryption procedures are described in the second and third parts respectively.

2.2 Mathematical Preliminaries

In Rijndael Algorithm, most of the operations are done at byte level. These bytes can be considered as elements of Galois Field (2^8), which is an extension field of Galois field (2) having elements of {0,1}. A sequence of 8 bits from Galois Field (2) forms an element in Galois Field (2^8). These 8 bit elements could be represented in polynomial notation. A byte consisting of elements [b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0] is represented as $b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0 x^0$ in polynomial notation, where $b_i \in \{0,1\}$. There are two operations in Galois Field (2^8), namely addition and multiplication, which are different from conventional addition and multiplication. These operations are explained in the next two sections.

2.2.1 Addition Operation

The addition operation in Galois Field (2^8) can be considered as addition of the polynomials. That is, the coefficients having the same degree are added. The coefficients of the polynomials are the elements of Galois Field (2), which are {0,1} and the addition of these coefficients are defined as simple EXOR operation.

2.2.2 Multiplication Operation

In Galois Field (2^8) the multiplication operation is defined as polynomial multiplication. When two polynomials are multiplied, the result may have a degree greater then 7 so a modulus operation is required. A prime polynomial of degree 8 is used in this modulus operation. In the Rijndael Algorithm this prime polynomial is named m(x) and is chosen as:

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

For example, multiplying a polynomial with 'x', which is a frequent operation, is done as follows: A '0' is concatenated to the right. If the leftmost bit is '1', then the byte is EXOR'ed with "100011011"; else no EXOR operation is required. The rightmost eight bits become the result. Multiplication with other polynomials can be considered as a sequence of multiplications with 'x'.

2.2.3 Polynomials with coefficients in GF(2^8)

Polynomials with coefficients in GF(2^8) can also be formed. In the Rijndael Algorithm 4-byte vectors are used in some operations, that is we need 3rd degree polynomials having 8-bit coefficients to represent these vectors. The addition of these polynomials is again a simple bitwise EXOR operation. Multiplication of these polynomials is done modulo $M(x) = x^4 + 1$. For example, given

$$a(x) = a_3\, x^3 + a_2\, x^2 + a_1\, x^1 + a_0\, x^0 ,$$
$$b(x) = b_3\, x^3 + b_2\, x^2 + b_1\, x^1 + b_0\, x^0 ,$$
$$d(x) = a(x) \otimes b(x) \bmod (x^4 + 1) = d_3\, x^3 + d_2\, x^2 + d_1\, x^1 + d_0\, x^0 \Rightarrow$$

By using simple mathematics, it can be easily shown that

$$d_0 = a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 ,$$
$$d_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3 ,$$
$$d_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3 ,$$
$$d_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 .$$

Where "$\bullet$" operation represents multiplication of 8-bit elements modulo m(x), "$\oplus$" operation represents bit wise EXOR'ing of these 8-bit elements and "$\otimes$" operation represents multiplication of 3rd degree polynomials modulo M(x).

In matrix representation:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Here all the coefficients are in Galois Field (2^8), that is all of them are 8 bits. And the multiplications of these 8 bit coefficients are modulo $x^8 + x^4 + x^3 + x + 1$.

2.3 The Rijndael Cipher

Rijndael is a symmetric key block cipher. The term symmetric key means that there is only one secret key, which is used for both encryption and decryption. The term block cipher means that the data to be ciphered is processed in blocks of constant length. The output, named ciphered text, has same length as the input. The data before ciphering is called plaintext. The encryption algorithm has constant key and plaintext sizes that can be independently chosen as 128, 192 or 256 bits. These bits can be considered as an array consisting of 8 bits. There are 4 rows in this array and the number of columns, denoted by Nb, can be 4, 6 or 8 depending on the plaintext and the key length. Figure 2.1 shows an example array corresponding to a key size of 192 bits.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

Figure 2.1: Array of 192 bits.

This array is called the state, and the transformations are applied on this state. There is also a cipher key, which can be represented in a similar array. The number of columns in the array of the cipher key is denoted by Nk. Nb and Nk determine the number of rounds to encrypt a given block. A round consists of a sequence of operations which will be discussed in the next section.

2.3.1 Encryption Procedure

Rijndael Encryption Procedure is the combination of the transformations discussed in the previous sections. It is an iterative operation applied to the inner state. For how many rounds the encryption procedure will continue is determined by the plaintext and the key sizes. Table 2.2 shows the number of rounds, Nr, as a function of key and block length, Nk and Nb.

Table 2.1 Number of rounds as a function of key and data length.

Number of rounds	Nb=4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

The encryption starts with the addition of the initial key to the plaintext. Then the iteration continues for Nr – 1 rounds. In these rounds, different keys, which were obtained from the key expansion procedure, are used. Figure 2.13 shows the block diagram of the encryption procedure.

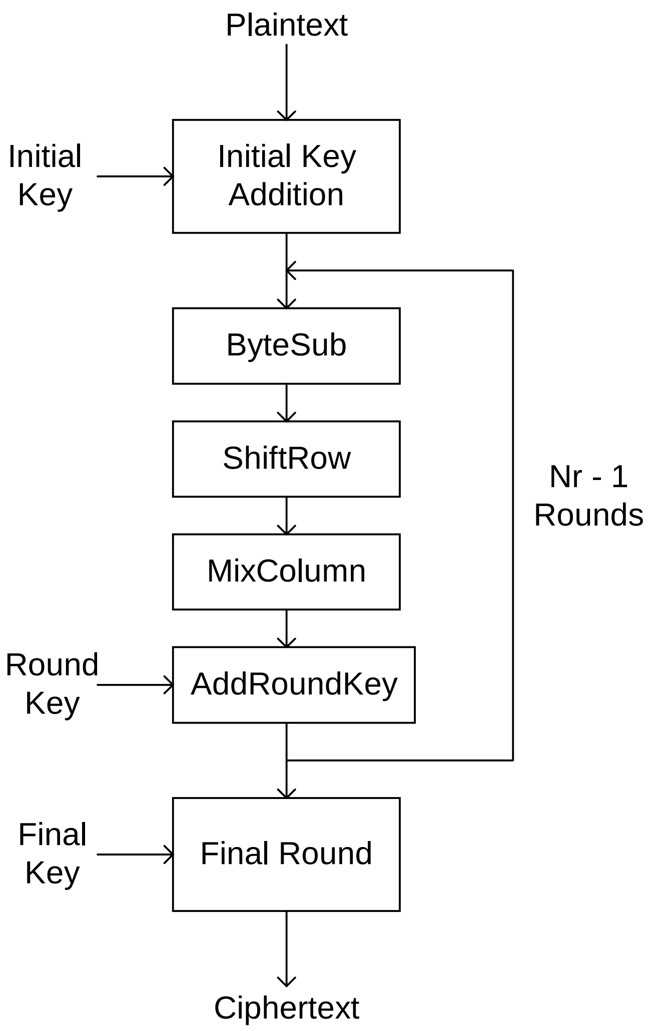


Figure 2.2: Encryption procedure.

2.3.2 Round Transformation

The round transformation in Rijndael Algorithm is composed of four transformations:

- Bytesub,
- ShiftRow,
- MixColumn,
- AddRoundKey.

These transformations are applied to the inner state consequently. Figure 2.2 shows the block diagram of a round.

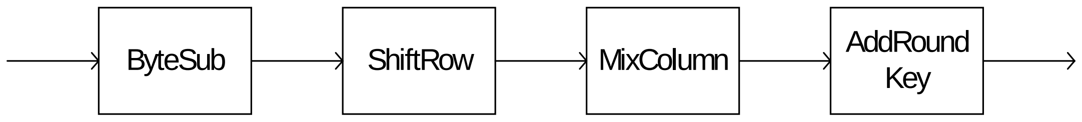


Figure 2.3: Rijndael round.

In the Rijndael Algorithm all rounds are identical, except for the final round, which does not contain MixColumn Operation.

2.3.2.1 The ByteSub Transformation

The ByteSub Transformation is a non-linear transformation that is applied to the individual bytes of the inner state. This is an invertible operation composed of two transformations. In the first operation, the multiplicative inverse of the byte in Galois Field (2^8) is calculated. The byte '00' does not have a multiplicative inverse so its inverse is taken as itself. As a second operation, an affine mapping over Galois Field (2) is applied. Figure 2.3 shows this mapping.

$$\begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \\ Y_5 \\ Y_6 \\ Y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 2.4: Affine mapping.

The ByteSub transformation can be considered as a substitution table, which has 256 elements consisting of 8 bits. This substitution table is referred to as an S-box. In the ByteSub Transformation all the bytes of a given block are passed through these S-boxes. Figure 2.4 shows this transformation. The corresponding substitution tables are given in the Appendix.

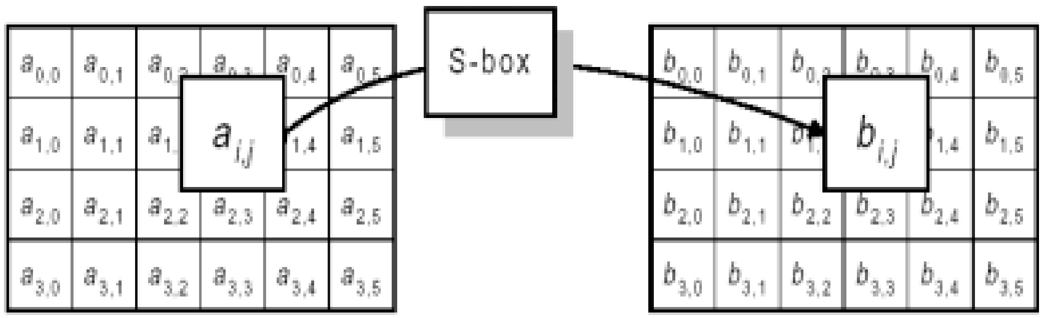

S-box
$a_{i,j}$
$b_{i,j}$
$a_{0,0}$ $a_{0,1}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$
$b_{0,0}$ $b_{0,1}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$ $b_{1,5}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$ $b_{2,5}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$ $b_{3,4}$ $b_{3,5}$

Figure 2.5: ByteSub Transformation.

2.3.2.2 The ShiftRow Transformation

The ShiftRow Transformation is a cyclical shift operation that is applied to the rows of the inner state and consists of the shifting of the rows by different offsets which depend on the data block length Nb. The first row is not shifted; the second row is shifted over 1 byte to the left. The third row is shifted over 2 bytes, if the block length is 128 or 192 bits, and it is shifted over 3 bytes, if the block length is 256 bits. The fourth row is shifted over 3 bytes if the block length is 128 or 192 bits, and it is shifted over 4 bytes if the block length is 256 bits. Table 2.1 shows the offset values as a function of the block length.

Table 2.2 ShiftRow offset values.

Block Length	Row 1 offset	Row 2 offset	Row 3 offset
128 bits	1	2	3
192 bits	1	2	3
256 bits	1	3	4

2.3.2.3 The MixColumn Transformation

The MixColumn Transformation is applied to the columns of the inner state. There are four bytes in a column, forming a vector. We can consider this vector as coefficients of a 3rd degree polynomial. This polynomial is multiplied with a fixed polynomial to complete MixColumn operation. In the Rijndael Algorithm this polynomial is named c(x) having 8-bit coefficients and is given by

$$c(x) = \text{‘03’}\, x^3 + \text{‘01’}\, x^2 + \text{‘01’}\, x + \text{‘02’}.$$

This polynomial multiplication is modulo $x^4 + 1$ and can be written as a matrix multiplication, as stated in Section 1.1.3. Let $b(x) = a(x) \otimes c(x)$, then by using simple mathematics the coefficients of the 3rd degree polynomial b(x) can be found as:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

'02' represents 'x' and '03' represents 'x + 1' in polynomial notation. Multiplication with 'x' was discussed in Section 1.1.2 and multiplication by 'x + 1' is obtained by EXOR'ing the result of 'x' multiplication to the coefficient itself.

In MixColumn Transformation all the columns are multiplied with c(x). Figure 2.5 shows the MixColumn operation applied to an inner state of 192 bits.

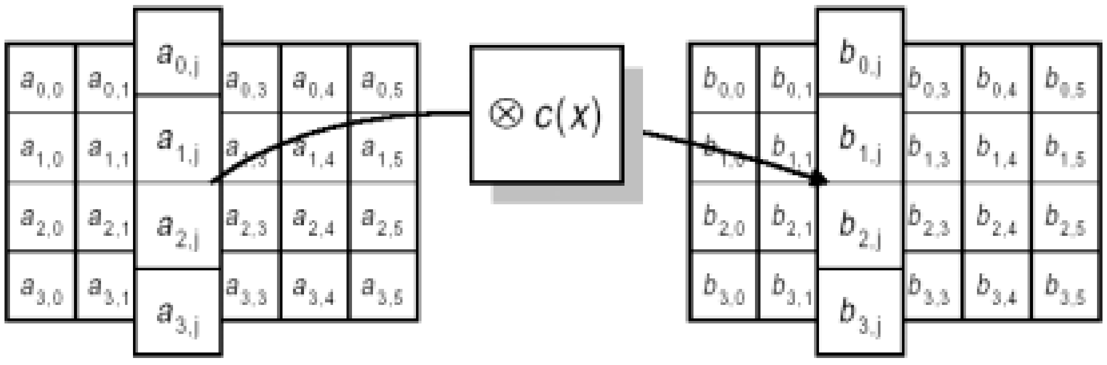

$a_{0,0}$ $a_{0,1}$ $a_{0,j}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,j}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,j}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,j}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$
$\otimes c(x)$
$b_{0,0}$ $b_{0,1}$ $b_{0,j}$ $b_{0,3}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$ $b_{1,j}$ $b_{1,3}$ $b_{1,4}$ $b_{1,5}$
$b_{2,0}$ $b_{2,1}$ $b_{2,j}$ $b_{2,3}$ $b_{2,4}$ $b_{2,5}$
$b_{3,0}$ $b_{3,1}$ $b_{3,j}$ $b_{3,3}$ $b_{3,4}$ $b_{3,5}$

Figure 2.6: MixColumn Operation.

2.3.2.4 AddRoundKey Transformation

In this operation, the inner state is bitwise EXOR'ed with a round key. Figure 2.6 shows this operation.

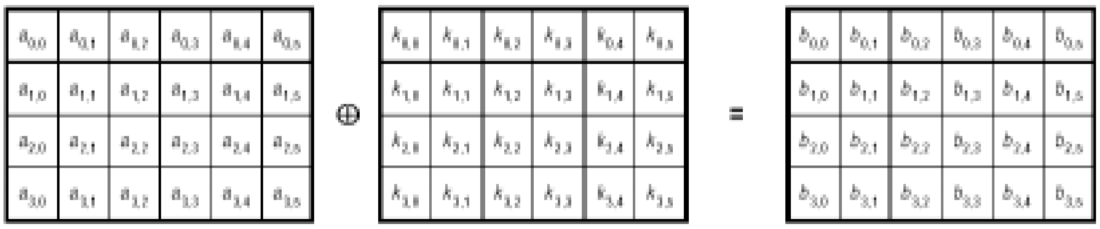

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$
$\oplus$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$ $k_{0,4}$ $k_{0,5}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$ $k_{1,4}$ $k_{1,5}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$ $k_{2,4}$ $k_{2,5}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$ $k_{3,4}$ $k_{3,5}$
=
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{1,4}$ $b_{1,5}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$ $b_{2,5}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$ $b_{3,4}$ $b_{3,5}$

Figure 2.7: Round Key Addition.

Every round has its own specific key which is obtained from the initial cipher key by using a procedure called the key schedule. This procedure is described in the next section.

2.3.3 Key Schedule

The round keys are obtained from the initial cipher key, which can be 128, 192 or 256 bits long. The length of the plaintext determines how many bits are needed at each round. For example, if Nk = 4, that is the cipher key is 128 bits long, and Nb = 6, that is the plaintext is 192 bits long, then there are 12 rounds specified by the algorithm. In each round, a 192-bit key is needed, and another 192 bits are needed for the initial key addition, so we need a total of 12 * 192 + 192 = 2496 bits to complete the encryption.

The procedure for obtaining these 2496 bits is called key expansion. Key expansion and round key selection are discussed in the next two subsections.

2.3.3.1 Key Expansion

In the key expansion procedure, the columns of the key can be considered as 4-byte vectors. These vectors form an array, called W. The first Nk vectors of this array contain the cipher key. The remaining elements for Nk = 4, 6 and 8 are calculated as:

i. Nk = 4:

The first 4 vectors contain the cipher key. The ith vector for i > 3 is calculated as:

$W[i] = W[i-1] \oplus W[i-4]$, if $(i \bmod 4) \neq 0$;

$W[i] = \text{Subbyte}(\text{Rotbyte}(W[i-1])) \oplus \text{Rcon}[i/4] \oplus W[i-4]$, if $(i \bmod 4) = 0$.

Here, Rotbyte is a permutation function operating on the column vector. Figure 2.7 shows this function.

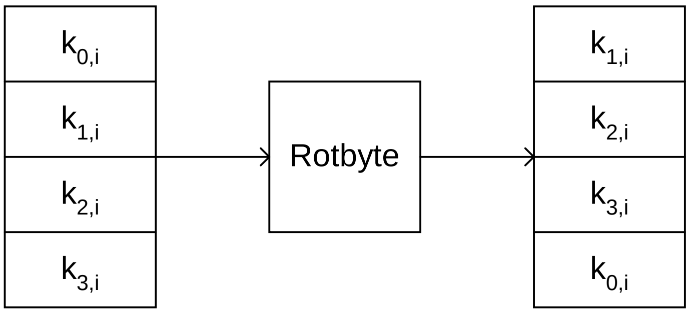


Figure 2.8: Rotbyte Function.

Subbyte is a function that returns 4-byte vector where each each byte is the result of S-box transformation operating on the corresponding byte in the input vector. Rcon is a 4-byte vector named as round constant. The round constants are independent from the key size. It is defined as:

$$\mathrm{Rcon}[i] = (\mathrm{RC}[i], \text{‘00’}, \text{‘00’}, \text{‘00’}).$$

RC[i] is an 8-bit vector updated when (i mod Nk) = 0 and it is calculated as:

$$\mathrm{RC}[i] = x * \mathrm{RC}[i-1] \text{ where } \mathrm{RC}[1] = \text{‘01’}.$$

Figure 2.8 shows the key expansion for Nk = 4.

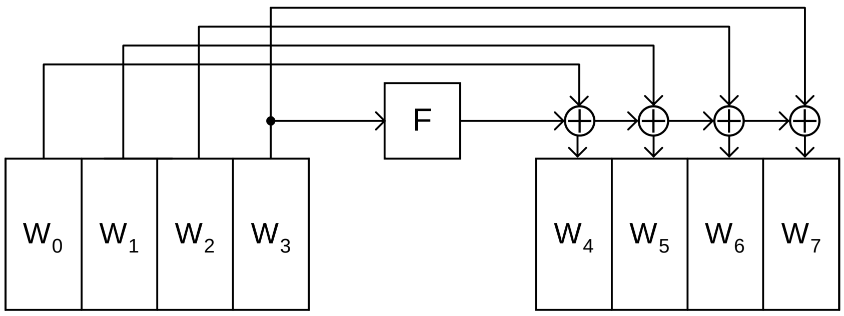


Figure 2.9: Key expansion for 128 bit cipher key.

F is the function Subbyte(Rotbyte(W[i – 1])) ⊕ Rcon[i / 4] represented in the figure below.

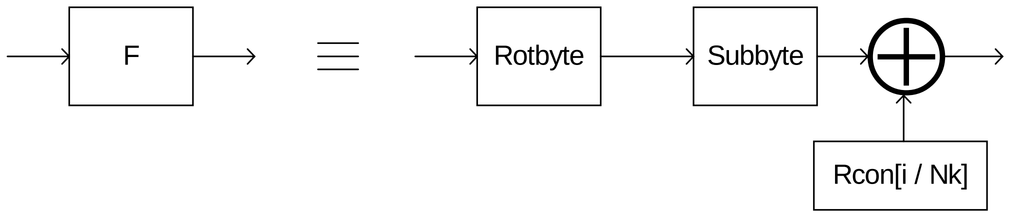


Figure 2.10: The function F.

ii. Nk = 6:

For Nk = 6, first 6 vectors contain the cipher key. Similar with Nk = 4, the other vectors for i > 5 are calculated as:

W[i] = W[i – 1] $\oplus$ W[i – 6] , if (i mod 6) $\neq$ 0;

W[i] = Subbyte(Rotbyte(W[i – 1])) $\oplus$ Rcon[i / 6] $\oplus$ W[i – 6], if (i mod 6) = 0.

Figure 2.10 shows the block diagram of this operation.

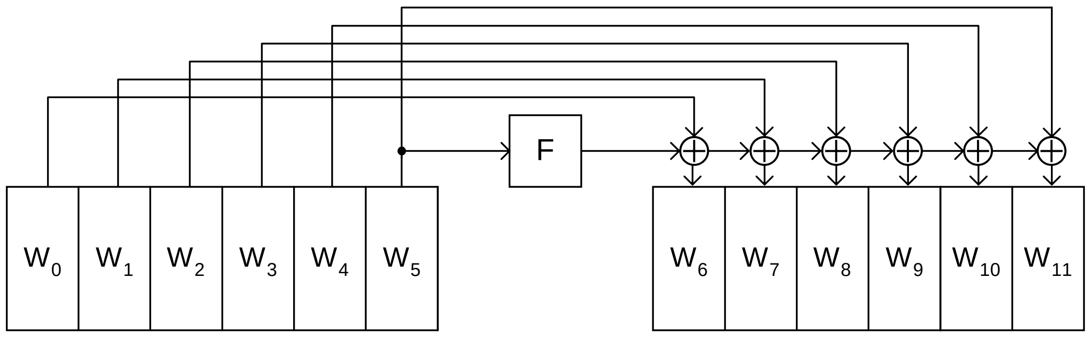


Figure 2.11: Key expansion for 192 bit key size.

iii. Nk = 8:

Similarly, first 8 vectors contain the cipher key. The other vectors for i > 7 are calculated as:

W [i] = Subbyte(Rotbyte(W [i – 1])) ⊕ Rcon [i / 8] ⊕ W [i – 8], if (i mod 8) = 0;

W [i] = Subbyte(W [i – 1]) ⊕ W [i – 8] , if (i mod 8) = 4;

W [i] = W [i – 1] ⊕ W [i – 8] , otherwise.

Figure 2.11 shows the block diagram for the key expansion of 256-bit cipher key.

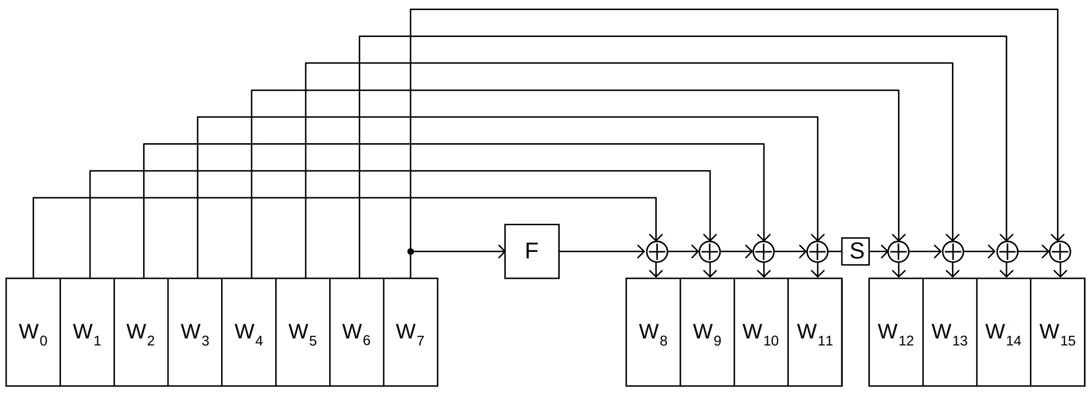


Figure 2.12: Key expansion for 256 bit key size.

2.3.3.2 Round Key Selection

In each round of the encryption, a different key is used. For the ith round, the round key contains the array elements from W[Nb * i] to W[Nb * (i + 1)]. Figure 2.12 shows an example key selection for Nb = 4.

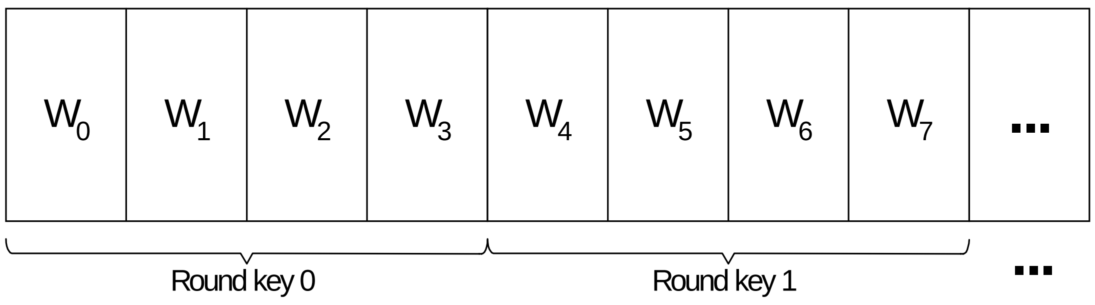


Figure 2.13: Key selection for 128-bit block size.

2.4 Rijndael Decipher

The decryption process is the exact inverse of encryption. The inverses of the transformations are done from the last to the first. In the Inverse ByteSub Transformation, inverse S-boxes are used. In the Inverse ShiftRow Transformation, cyclically shifting is done to the right with the same offsets of the ShiftRow Transformation. The Inverse MixColumn Transformation is more complicated then the MixColumn Transformation. In the MixColumn Transformation the column vector is multiplied c(x) = ‘03’ x^3 + ‘01’ x^2 + ‘01’ x + ‘02’ modulo $x^4 + 1$. As stated in Section 1.2.4 this modular multiplication can be written as a multiplication of matrices as

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

For the inverse transformation, column vector must be multiplied with a polynomial named d(x) where d(x) = ‘0B’ x^3 + ‘0D’ x^2 + ‘09’ x + ‘0E’. This multiplication can also be written in matrix form as

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The coefficients of d(x) are higher then the coefficients of c(x), making the Inverse MixColumn Transformation more complicated then the MixColumn Transformation. The Inverse AddRoundKey Transformation is the same as AddRoundKey Transformation since it is only bit-wise EXOR operation.

The decryption procedure has the same number of rounds as the encryption procedure. It is the exact inverse of the encryption procedure starting with the inverse of the final round. The final key is added to the ciphertext, followed by Inverse ShiftRow and Inverse ByteSub Transformations. Then, the inverse of the round is applied for Nr – 1 times. Finally, the initial key is added to the inner state giving the plaintext. Figure 2.14 shows the block diagram of this procedure.

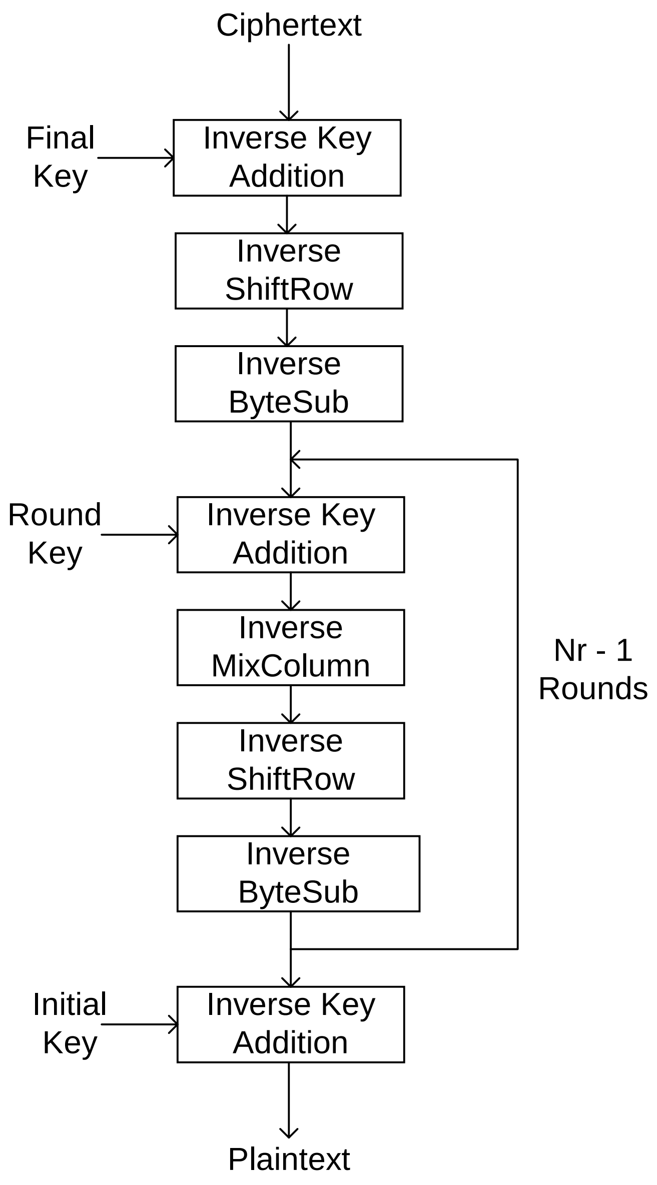


Figure 2.14: Decryption procedure.

In this chapter, the Rijndael Algorithm is explained. The mathematical background of the algorithm, the encryption-decryption flow and the operations used in the algorithm are explained in detail. Chapter 3 presents the different Rijndael implementations.

CHAPTER 3

DIFFERENT RIJNDAEL PROCESSOR IMPLEMENTATIONS

Both academia and industry have focused on the efficient implementation of the new AES algorithm. There are many publications involving FPGA, ASIC, and software implementations.

An ASIC implementation and its experimental results are presented by Henry Kuo, Ingrid Verbauwhede and Patrick Shaumont in [5], [6] and [7]. In this study, a Rijndael encryption core having a non-pipelined encryption data path is presented. This chip has on-the-fly key schedule data path. This means, the round keys necessary for the encryption are calculated at every round. No pre calculation and storage is needed for this case. However, for some data-key sizes, 2 round keys must be prepared in the same clock period. This significantly increases the critical length and decreases the overall speed of the chip.

The chip architecture of this Rijndael IC is shown in Figure 3.1 [5]. There is an encryption module, a processor for controlling the operations, two 256-bit data buffers for input and output data storage, two key scheduling modules to generate the required keys necessary for each round and two Finite State Machines (FSM's) controlling the input and output data interface.

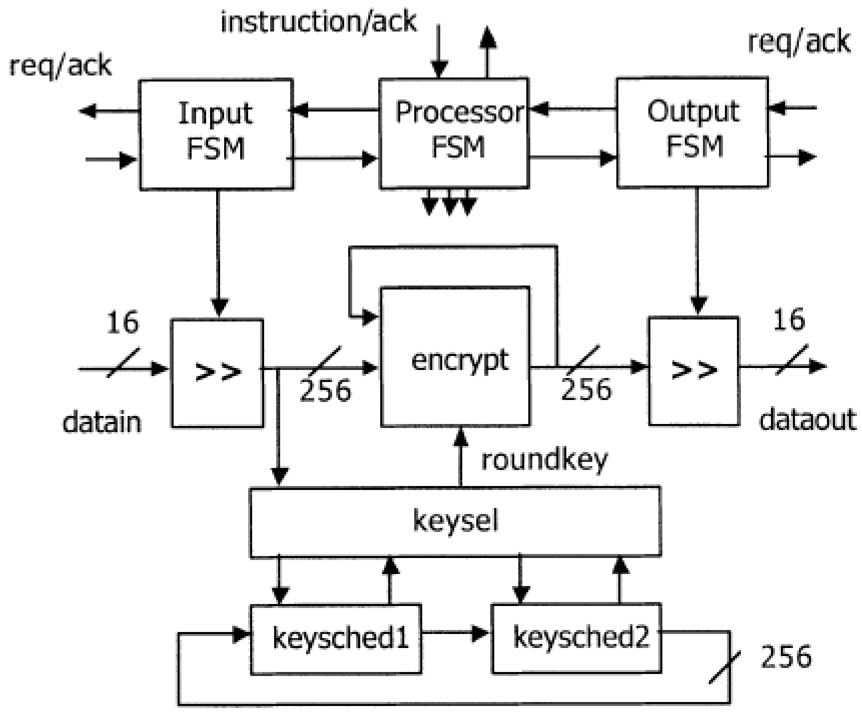


Figure 3.1: Block diagram of the Rijndael IC in [5].

This IC has a 16-bit asynchronous input and output interface which allows the chip to operate at higher frequencies independent from the speed of input and output interface. However, if the input and output interface is slow, then the fast operation of the chip has no use, because the operation is limited by the input and output interface.

It is reported that this implementation can run at 125 MHz by having a critical path of 8 ns. However, this is the speed of the encryption module. The key schedule module has a critical path of 10 ns decreasing the overall speed of the IC to 100 MHz. The chip has a gate count of 173,000. It is implemented by using 0.18 µm CMOS process. The chip occupies silicon area of 3.96 mm^2. Table 3.1 shows the throughput values for different data-key combinations. These values are calculated by considering a speed of 125 MHz. That is the speed of the encryption module and must be modified by a factor of 100/125 to get the overall throughput.

Table 3.1 Throughput values for different data-key lengths.

Throughput	Key Length = 128	Key Length = 192	Key Length = 256
Data Length = 128	1.6 Gbit/s	1.33 Gbit/s	1.14 Gbit/s
Data Length = 192	2.0 Gbit/s	2.0 Gbit/s	1.71 Gbit/s
Data Length = 256	2.29 Gbit/s	2.29 Gbit/s	2.29 Gbit/s

In another study by Pawel Chodowiec, Po Khuon and Kris Gaj [8], a different design methodology for secret key block ciphers including Rijndael, TDES and some other AES finalists is proposed. In this methodology, some pipeline stages are inserted inside the rounds of the ciphers. Most of the pipeline implementations [9] [10] use pipeline registers between rounds of the ciphers. However, this implementation uses both inner round registers and outer round registers to achieve very high speeds. The ciphers are implemented targeting a Xilinx [11] Virtex FPGA device, XCV1000BG560-6, which has about one million equivalent logic gates and fabricated using 0.22 µm CMOS process. For Rijndael implementation, a throughput of 414 Mb/sec is achieved by using basic iterative approach. By using inner round pipelining, a throughput of 1.265 Gbit/sec and by using full mixed inner and outer round pipelining a throughput of 12.160 Gbit/sec is achieved. This FPGA implementation runs at 32 MHz for iterative approach, 99 MHz for inner round pipelining and 95 MHz for mixed inner and outer round pipelining. In the FPGA implementations, the total number of the basic building blocks used by the design determines the area. These basic building blocks are named as Configurable Logic Block (CLB) for Xilinx FPGA's. For Rijndael implementation, 2507 CLBs are used in iterative approach, 2057 CLBs and 8 Embedded Array Blocks (EABs) are used in inner round pipelining, and 12600 CLBs and 80 RAMs are used in mixed inner and outer round pipelining. Pipeline registers are used in such a way that the delay between any two register is equal to the delay of a single CLB. However, after some point the routing delay becomes dominant. This decreases the area-speed efficiency. The mixed inner and outer pipeline implementation does not fit into a single XCV1000 FPGA. So for this implementation 3 different FPGA devices are used.

In a study by N. Sklavos and O. Koufopavlou [9] two different architectures and their FPGA implementations are presented. Both the encryption and decryption algorithms are implemented. The first architecture uses feedback logic and has a throughput of 259 Mbit/sec. The second architecture is optimized for pipeline approach and reaches to a throughput of 3.65 Gbit/sec. In the first design, the key expansion unit is integrated with the encryption-decryption core. In the second design, a RAM is used for key storage and loading. These two architectures are designed for data size of 128 bits. They do not support 192 bit and 256 bit data sizes specified in the original Rijndael algorithm. The pipeline architecture uses outer round pipelining. There are 10 separate round blocks and between these blocks, there are pipeline registers. No pipeline registers inside the rounds are used. In the Subbyte transformation, they used LUT for the calculation of the multiplicative inverses and operate a separate affine mapping. These two operations can be mapped on a single module. The first architecture runs at 22 MHz and has 2358 CLBs. The second architecture operates at 28.5 MHz and occupies 17314 CLBs.

In another study by Maire McLoone and John V. McCanny [10], a single-chip FPGA implementation of the AES is presented. Only 128-bit data size is supported. Pipeline implementation technique is used and a throughput of 7 Gbit/sec is reached on a Xilinx Virtex-E XCV812E-8-BG560 FPGA device. This implementation has ten pipeline stages and all stages are composed of a single Rijndael round. 2 ROMs are used for key scheduling procedure and 80 ROMs are used for 10 separate rounds. For encryption and decryption processes, different ROMs are needed. One method is doubling the block RAMs, which are used as ROMs. This increases the area requirement considerably. A good solution is proposed to overcome this problem. Two further ROMs are included, one of them containing the initialization values for the Look-up Tables (LUTs) required for the encryption and the other one containing the initialization values required by the decryption procedure. Before the encryption or decryption process starts, the block RAMs are initialized with the values specified for encryption or decryption. However, the latency of this initialization procedure can be important especially when the encryption-decryption modes are changed frequently. The design is implemented using Xilinx Foundation Series 3.1.i. software and Synplify Pro V6.0 [12]. The encryption implementation utilizes 2679 CLB slices and 82 block RAMs. 385 IO pads are used for the interface. The system clock is 54.35 MHz achieving 7 Gbit/sec throughput. The decryption implementation utilizes 4304 CLB slices and 82 block RAMs. Although the area of the chip increases, the system clock speed decreases to 49.9 MHz achieving 6.38 Gbit/sec throughput.

An FPGA implementation for both the encryption and decryption is presented by Joon Hyoung Shim, Dae Won Kim, Young Kyu Kang, Taek Won Kwon and Jun Rim Choi [13]. On-the-fly key scheduler is implemented performing forward key generation for encryption and reverse key generation for decryption. Only 128 bit data and key size is implemented, reducing the complexity of the forward and especially reverse key scheduling a lot. The Rijndael cryptoprocessor is implemented using Verilog-HDL and Xilinx XCV1000E FPGA device is targeted. The operating frequency is 38.8 MHz giving a throughput of 451.5 Mbps for encryption and decryption. It has 32 bit data input and 32 bit data output. 2580 CLB slices are used for this implementation.

Another ASIC implementation is presented by Tetsuya Ichikawa, Tomomi Kasuya and Mitsuru Matsui [14]. In this study, the critical paths of the AES finalists are analyzed. Fully loop-unrolled designs without any pipeline registers are implemented. This means, all rounds of the algorithm are cascaded for having a circuit that implements encryption of one block at one clock cycle. Only 128-bit key versions of the AES finalists are implemented. Mitsubishi Electric's 0.35-micron CMOS ASIC design libraries are used. The design is implemented using Verilog-HDL. Synopsys Design Compiler version 1998.08 [15] is used for logic synthesis. In the implementations, it is assumed that all the keys are calculated beforehand so no key setup time is required. The main concern in this study is to analyze the critical paths of the AES finalists so they do not concentrate on reducing the size of the hardware. The Rijndael chip utilizes 612,834 gates having a critical path of 65.64 ns and a throughput of 1.95 Gbit/sec.

In another study by Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim [16], concurrent error detection (CED) architectures for symmetric key block ciphers are investigated. Some techniques for these CED architectures are proposed and validated on FPGA implementations of AES finalists. (The proposed CED techniques are out of the scope of this text. The main concern of this thesis is about the 128-bit Rijndael FPGA implementation.) The round keys are stored in a register file for use of encryption and decryption. Each of the ByteSub, ShiftRow, MixColumn, and AddRoundKey operations are applied at separate clock cycles so 11 round operation costs 4 x 11 = 44 clock cycles. This decreases the throughput. When no CED architecture is used, the frequency of the chip reaches to 46.93 MHz giving a throughput of 136.53 Mbit/sec. When algorithm level CED is implemented, the frequency of the chip decreases to 36.44 MHz and the throughput becomes 53.04 Mbit/sec. For the round level and operation level CED implementations, the frequency becomes 37.60 and 36.69 MHz giving a throughput of 100.27 Mbit/sec and 104.36 Mbit/sec respectively. The chip with no CED architecture utilizes 3973 CLB slices. For the algorithm level, round level, and operation level CED architectures, the area increases to 4806, 4724, and 5486 CLB slices respectively.

Table 3.2 summarizes the hardware implementations of the Rijndael Algorithm.

Table 3.2: Different hardware implementations.

Architect.	Process	Design	Tech.	Number of Gates	Frequency	Throughput
H. Kuo & et. al. [5]	Enc.	ASIC	0.18 µm	173k	125 MHz	2.29 Gbit/s
K. Gaj & et. al. 1 [8]	Enc.	FPGA	_	2507 CLB	32 MHz	414 Mbit/s
K. Gaj & et. al. 2 [8]	Enc.	FPGA	_	12.6k CLB	95 MHz	12.2 Gbit/s Pipelined
N. Sklavos & et. al. 1 [9]	Enc/Dec.	FPGA	_	2358 CLB	22 MHz	259 Mbit/s
N. Sklavos & et. al. 2 [9]	Enc/Dec.	FPGA	_	17.3k CLB	28.5 MHz	3.65 Gbit/s Pipelined
McLoone & et. al. 1 [10]	Enc.	FPGA	_	2.7kCLB+ 82 RAM	54.4 MHz.	7 Gbit/s Pipelined
McLoone & et. al. 2 [10]	Dec.	FPGA	_	4.3k CLB + 82 RAM	49.9 MHz	6.38 Gbit/s Pipelined
H. Shim & et. al. [13]	Enc/Dec	FPGA	_	2580 CLB	38.8 Hz	452 Mbit/s
Ichikawa & et. al. [14]	Enc	ASIC	0.35 µm	613k	15.2 MHz	1.95 Gbit/s
R. Karri & et. al. [16]	Enc	FPGA	_	3973 CLB	47 MHz	137 Mbit/s

This chapter presented the different published Rijndael implementations. Chapter 4 explains the implementation details of the Rijndael Processor.

CHAPTER 4

IMPLEMENTATION DETAILS OF RIJNDAEL PROCESSOR

4.1 Introduction

In this work, the Rijndael Processor is designed and implemented as an ASIC using 0.35 µm Standard CMOS Technology. The processor supports all the key and data lengths of the Rijndael Algorithm and it can perform both the encryption and decryption. Also an FPGA implementation is completed for hardware verification of the design.

In Section 4.2 the design of the Rijndael Processor is described. Section 4.3 and Section 4.4 explain the ASIC and the FPGA implementations respectively. The simulation results are described in Section 4.5.

4.2 Design of Rijndael Processor

Rijndael is a symmetric block cipher having variable key and data length. Although both the key and data length can be independently chosen as 128, 192 or 256 bits, the NIST has fixed the data length to 128 bits in the AES. This implementation, however, supports all the key and data length combinations. Figure 4.1 shows the block diagram of the Rijndael Processor.

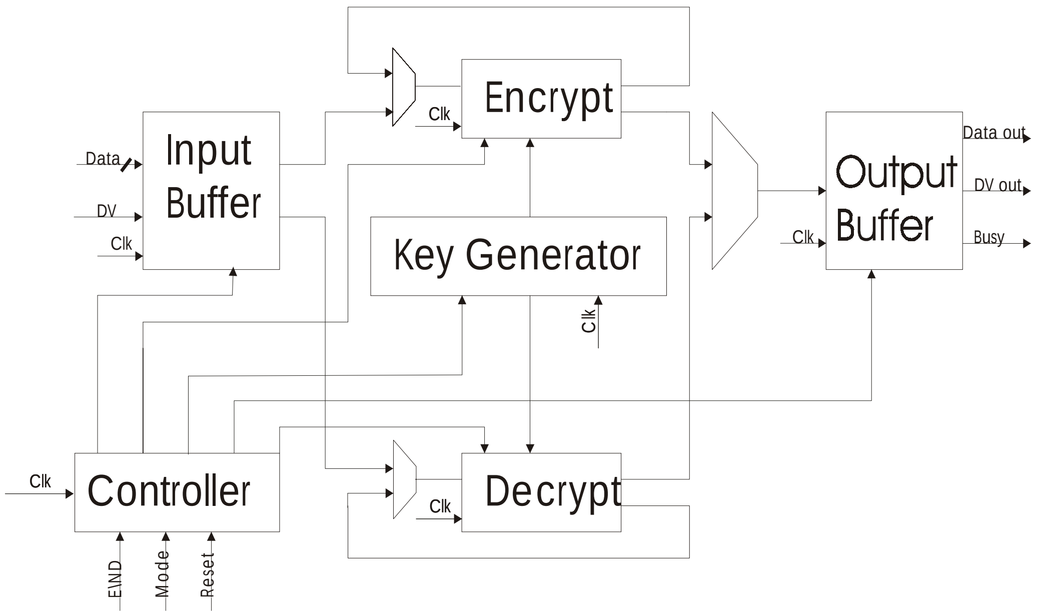


Figure 4.1: Block diagram of the Rijndael Processor.

There is an Encryption Module, a Decryption Module, a Key Generator Module and a Controller Module. There are also input and output buffers for data storage. These modules are explained in the next sections.

4.2.1 Encryption Module

Figure 4.2 shows the Encryption Module. It includes three sub modules, which are S_box_logic_32, ShiftRow and Mixcolumn_256. Details of these modules are explained in the following sections. As it is seen from the figure, the Encryption Module completes one round of the Rijndael algorithm in one clock cycle. There is a register called Encrypt Register for storing the inner state of the encryption process. It is a 256-bit register because the plaintext can be at most 256 bits long. When data size is 128 bits, the remaining 128 bits of Encrypt Register are not used. Similarly for 192-bit data processing, the remaining 64 bits are not used.

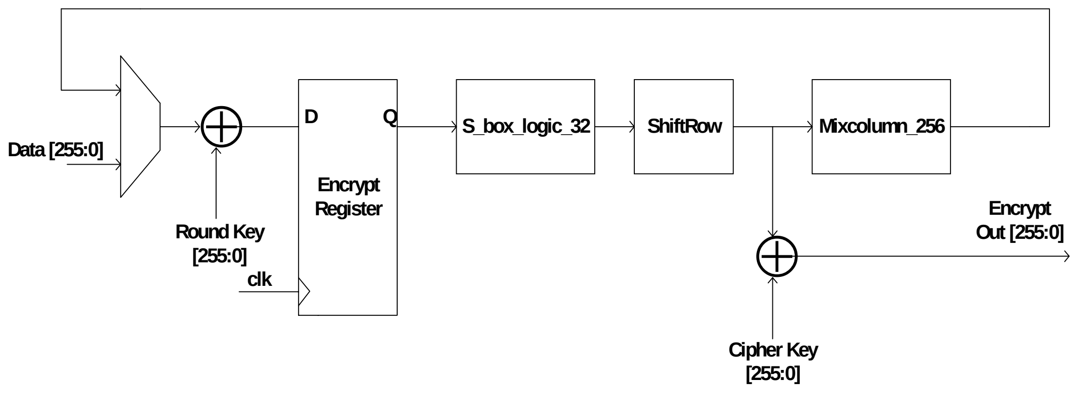


Figure 4.2: Block diagram of the Encryption Module.

In the Encryption Module, there is a multiplexer for choosing one of the two different 256 bit groups. When encryption of a block ends, then the encryption of the next block starts. At this beginning state, the multiplexer lets the new data to enter the Encrypt Register after an EXOR operation with the round key. While the encryption process continues, the multiplexer lets the output of the Mixcolumn_256 Module to enter.

At the last round of the Rijndael algorithm, the output of the Shift Row Module is EXOR'ed with the cipher key. Cipher key is the last key of the encryption procedure and for different data and key sizes it gets different values. The Key Generator Module, which will be described in Section 4.2.3, supplies the cipher and round keys to the Encryption Module.

There are two separate EXOR implementations in the Encryption Module because the last round of the encryption of the current block and the first round of the encryption of the next block are processed at the same time. This prevents loosing one clock cycle.

4.2.1.1 S-Box Implementation

S-box implementation is very important in the design of the Rijndael Processor. These S-boxes must be very fast to achieve high throughput. However, at the same time the gate count of these S-boxes must be optimum to achieve an area efficient design. In one clock cycle, one round of the algorithm is completed. To pass all 256 bits of the inner state through S-boxes in one clock cycle needs parallel processing. In fact, 32 S-Boxes are needed for this parallel processing.

S boxes can be implemented as a ROM having 8-bit address and 8-bit data. However, ROM's are not so fast. In this work, the S-boxes are implemented using combinational logic.

4.2.1.2 MixColumn Implementation

In the MixColumn Transformation, a column of the inner state is multiplied with a matrix and this operation was discussed in Section 2.3.4. For this transformation, multiplication with 'x' in Galois Field (2^8) is needed. This multiplication is called 'xtime operation' [4]. Block diagram of the xtime operation is shown in Figure 4.3.

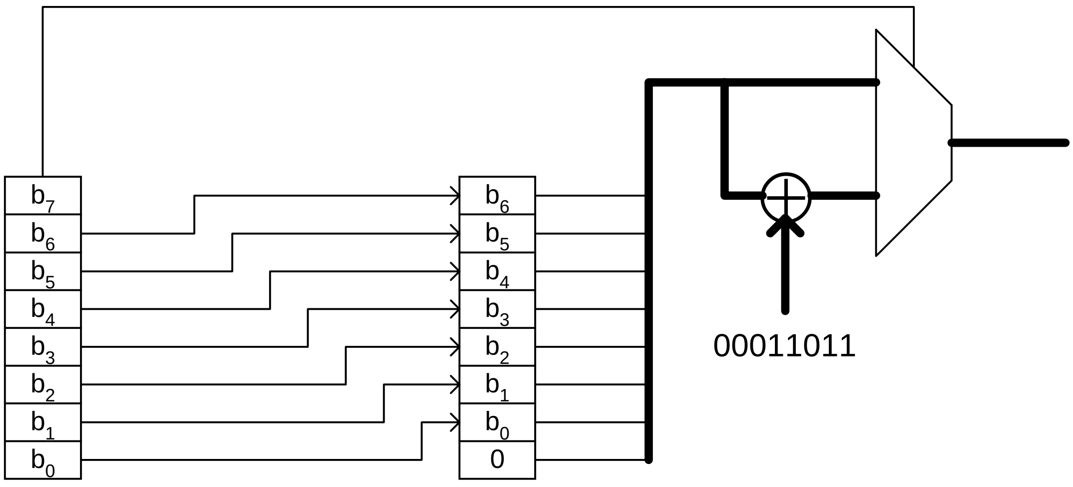


Figure 4.3: ‘xtime’ operation.

In the MixColumn Module, 4 xtime modules are used. Figure 4.4 shows the design of the MixColumn Module.

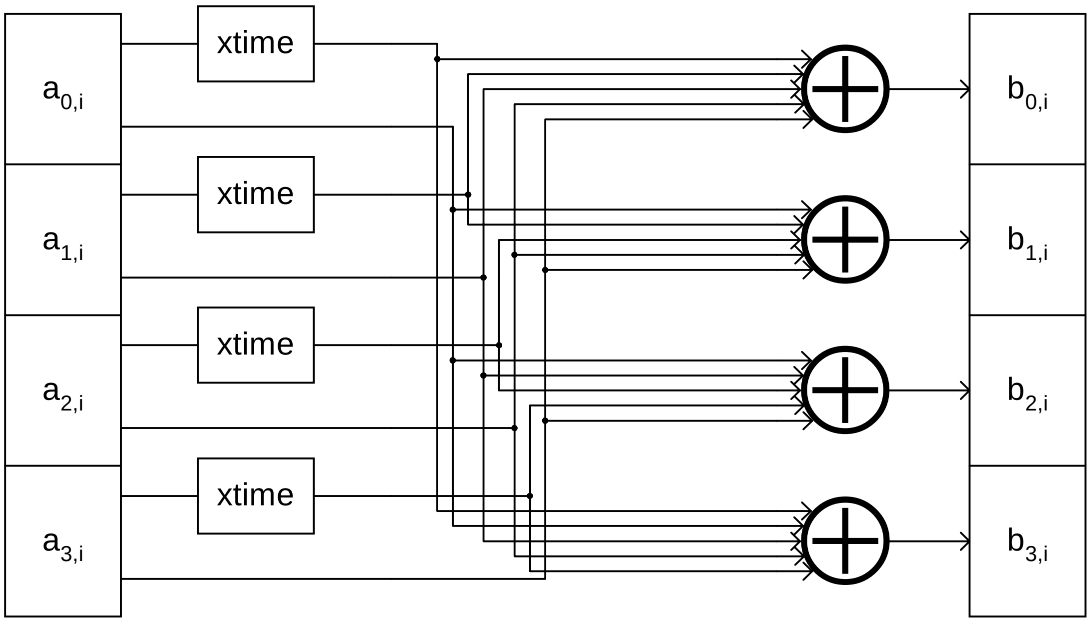


Figure 4.4: Block diagram of the Mixcolumn Module.

The MixColumn Module operates on only one column of the inner state. To achieve parallel operation for completing the round in one clock cycle, 8 MixColumn Module's must be used in parallel. This new module is called as Mixcolumn_256 because it operates on 256 bits at the same time. Figure 4.5 shows the block diagram of this module.

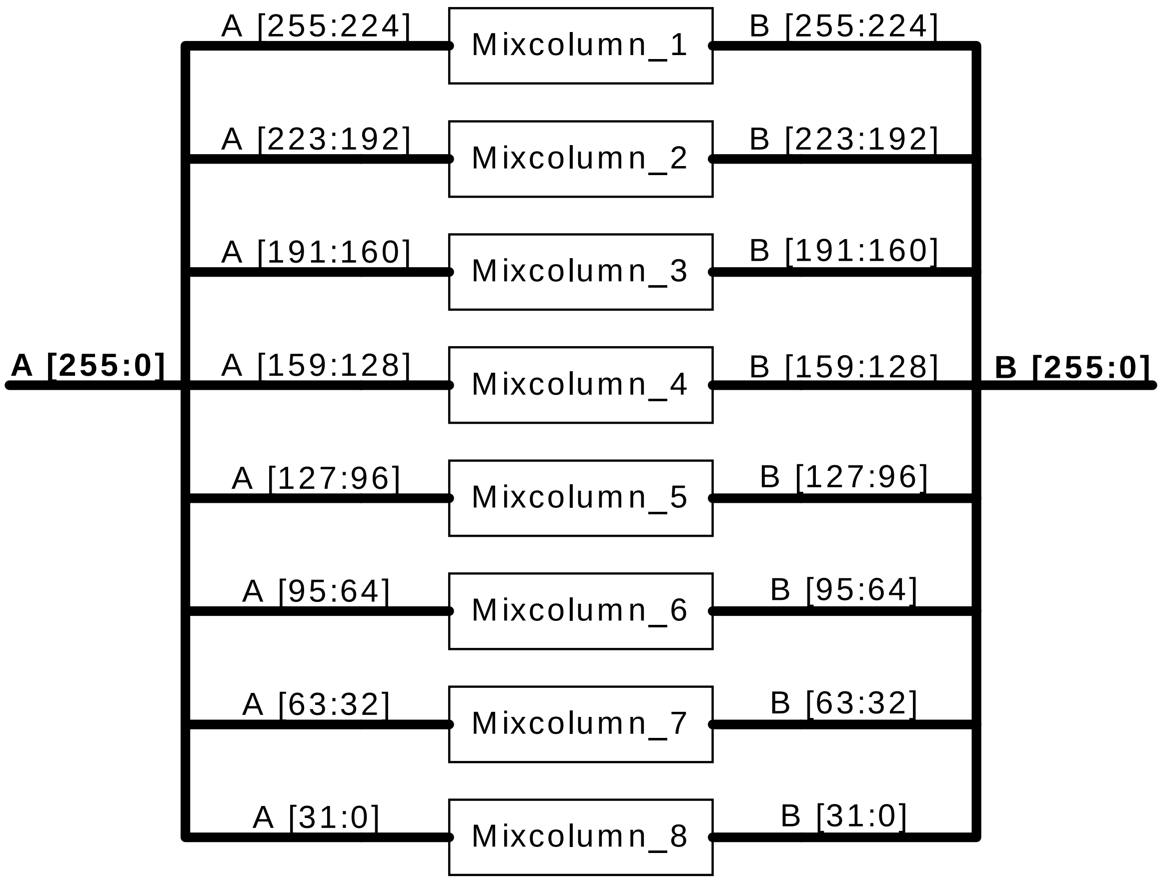


Figure 4.5: Block diagram of the Mixcolumn_256 Module.

4.2.1.3 ShiftRow Module

ShiftRow Transformation is a permutation operation. The rows of the inner state are cyclically shifted to the left by constant offsets. These offsets are determined by the size of the plaintext. In the implementation of the ShiftRow Module, multiplexers are used to provide this shifting. First row is not shifted, so no multiplexers are needed for this row. The second row is shifted by 1 for all the block sizes, so again no multiplexing is needed. A permutation with only wiring is enough for that row. The third row is shifted by either 2 or 3, and the last row is shifted by either 3 or 4 so 2x1 multiplexers are needed for the shifting of these rows. There are totally 128 bits in these rows so 128 multiplexers (2x1) are needed. Figure 4.6 shows the ShiftRow Transformation. Only the first column output is shown.

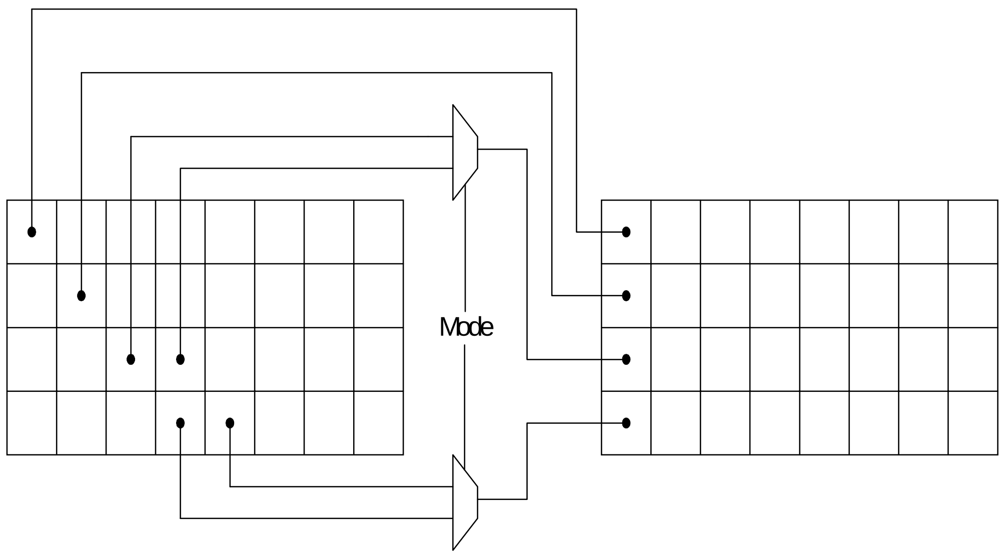


Figure 4.6: ShiftRow Transformation

4.2.1.4 AddRoundKey Implementation

Round key addition is simply an EXOR operation. The key is supplied from the outside of the Encrypt Module. The length of the key and the data can be at most 256 bits so 256 EXOR gates having 2 inputs are used.

4.2.2 Decryption Module

Decryption Module has a similar structure with the Encryption Module. Figure 4.7 shows the block diagram of this module. It has inverse sub modules, which are Inverse Mixcolumn_256, Inverse ShiftRow and Inverse S_Box_logic_256. These modules are explained in the following sections.

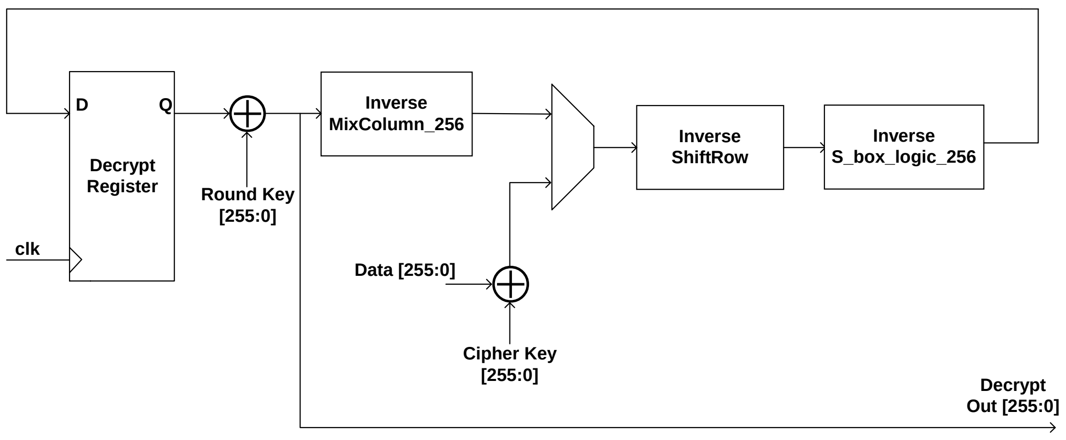


Figure 4.7: Block diagram of the Decryption Module.

The Decryption Module is completely a separate module. Although it has some overlapping operations with the Encryption Module, operational blocks are not shared to achieve high-speed operation. This implementation style decreases the metal routing of the chip, which is an important problem for especially high-density chips. Many multiplexers are also avoided by completely separating the Encryption and Decryption Modules.

The order of the operations for decryption is the inverse of the order for the encryption procedure. The decryption procedure starts with the round key addition. Therefore, EXOR's for the key addition are implemented at the output of the Decrypt Register. The last round of the process does not contain MixColumn operation, so the place of the multiplexer is different then the Encryption Module.

The multiplexer is prior to the Inverse Shift Row Module. When the decryption procedure continues, the multiplexer lets the output of the Inverse Mixcolumn_256 Module to enter the Inverse Shift Row Module. When it is just the first round, then the multiplexer chooses the EXOR output of the new data block and the cipher key. Since cipher key is the last key in the encryption procedure, it is applied at the first step in the decryption procedure.

Similar to the Encryption Module, there are two separate EXOR implementations for processing the last round of the current block and the first round of the next block at the same time.

4.2.2.1 Inverse S-Box Logic Implementation

Inverse S-Box logic is also a critical implementation. There are 32 Inverse S-boxes in the design to achieve parallelism for completing the Inverse ByteSub transformation in one clock cycle. This situation makes both the area and the timing of these Inverse S-boxes critical. Similar to the S-box implementation, the Inverse S-boxes are implemented by using combinational logic to achieve fast operation.

4.2.2.2 Inverse MixColumn Implementation

Inverse MixColumn operation is the multiplication of the column vector with the polynomial $d(x) = \text{‘0B’}\ x^3 + \text{‘0D’}\ x^2 + \text{‘09’}\ x + \text{‘14’}$ modulo $x^4 + 1$. In Section 2.4 it was described that this modular multiplication can be written in a matrix form of:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figure 4.8 shows the block diagram of the Inverse MixColumn operation.

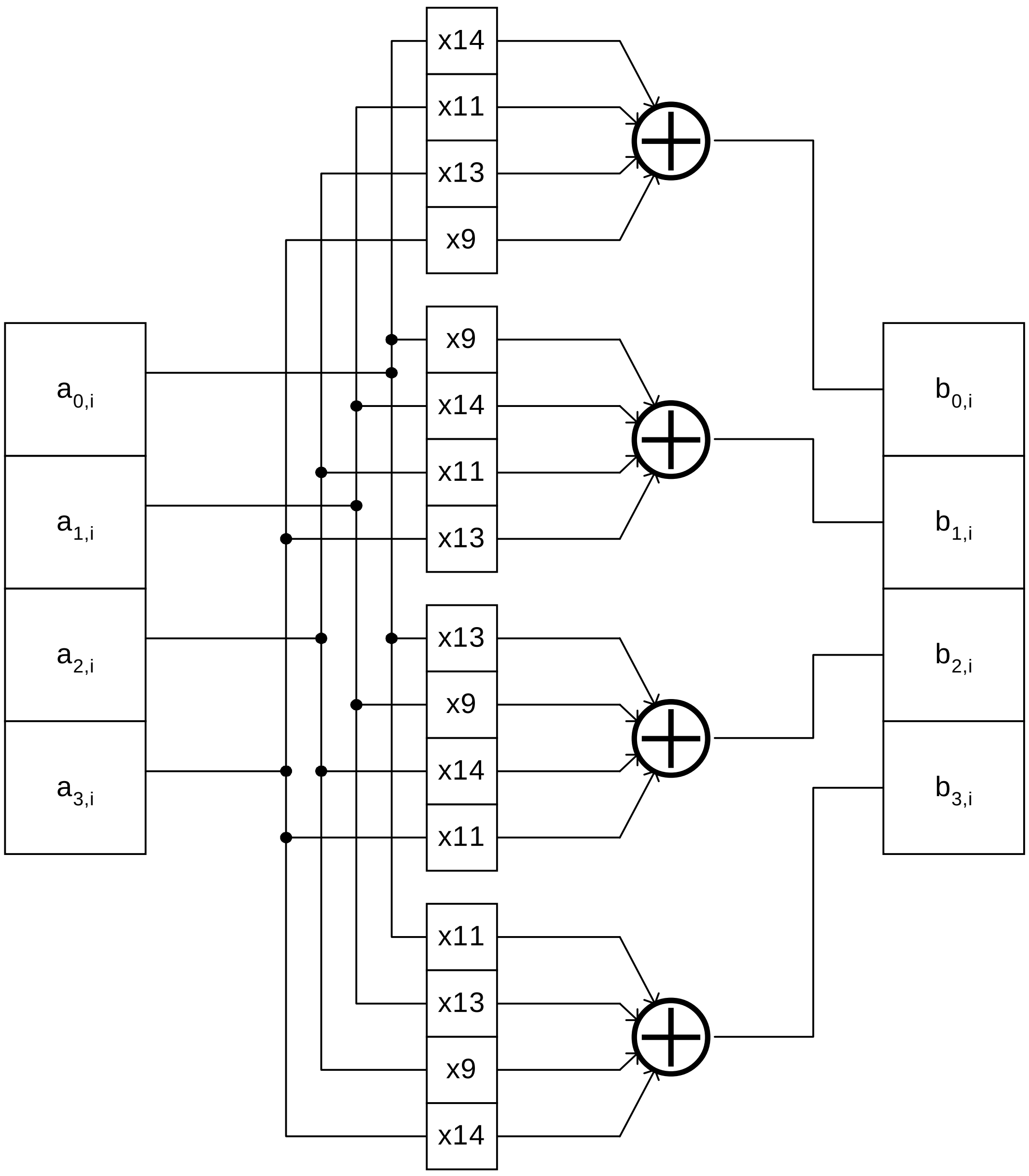


Figure 4.8: Block diagram of Inverse Mixcolumn Module.

The coefficients of the Inverse MixColumn operation are bigger than the coefficients of the MixColumn operation. This increases the complexity of the design. The polynomial notations of these coefficients are:

‘09’ = $x^3 + 1$;

‘11’ = $x^3 + x + 1$;

‘13’ = $x^3 + x^2 + 1$;

‘14’ = $x^3 + x^2 + x$.

The multiplications with these coefficients are combinations of multiplications with x^3, x^2, x and 1.

4.2.2.2.1 Multiplication with x^3

Multiplication with ‘x^{3}’ can be implementation as three subsequent multiplications with ‘x’ that is shown in the figure below.

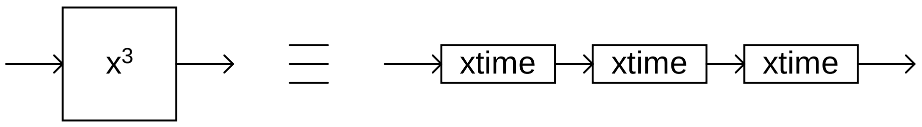


Figure 4.9: Cascaded multiplications.

However, in this method of implementation, the internal delays of xtime modules are added, producing 3 times more delay then a single xtime component. In our implementation, the speed of the design is critical so the parallelism of the design is increased.

In multiplication with x^3, the complicated thing is the modulus operation. It is enough to look at the first three bits of the coefficient to decide if any modulus operation is needed or not. If the first 3 bits are ‘000’, then no modulus operation is needed and multiplication with x^3 becomes just shifting the byte 3 bits to the left and adding 3 zeros to the right. If the first three bits are ‘101’, then the multiplication is done in this way: First, the byte is shifted 1 place to the left and a zero is added to the rightmost bit. Since the leftmost bit is 1, then a modulus operation is needed, that is the byte is EXOR’ed with ‘00011011’. After this operation, the second bit, which is zero, becomes the first bit so in the second shift operation, no modulus operation is needed. Now the third bit, which is one, becomes the first bit, so after the third shifting, a modulus operation is again needed. After these operations, the byte changes as:

$$a_7\, a_6\, a_5\, a_4\, a_3\, a_2\, a_1\, a_0 \longrightarrow a_4\, \overline{a_3}\, \overline{a_2}\, \overline{a_1}\, a_0\, 1\, 1\, 1$$

For the remaining values of the first three bits, the multiplication results with x^3 can be found. Table 4.1 shows the results:

Table 4.1: Multiplication with x^3.

(a_7,a_6,a_5)	Multiplication with x^3
‘000’	$a_4\ a_3\ a_2\ a_1\ a_0\ 000$
‘001’	$a_4\ a_3\ a_2\ \overline{a_1}\ \overline{a_0}\ 011$
‘010’	$a_4\ a_3\ \overline{a_2}\ \overline{a_1}\ a_0\ 110$
‘011’	$a_4\ a_3\ \overline{a_2}\ a_1\ \overline{a_0}\ 101$
‘100’	$a_4\ \overline{a_3}\ \overline{a_2}\ a_1\ \overline{a_0}\ 100$
‘101’	$a_4\ \overline{a_3}\ \overline{a_2}\ \overline{a_1}\ a_0\ 111$
‘110’	$a_4\ \overline{a_3}\ a_2\ \overline{a_1}\ \overline{a_0}\ 010$
‘111’	$a_4\ \overline{a_3}\ a_2\ a_1\ a_0\ 001$

4.2.2.2.2 Multiplication with x^2

Multiplication with ‘x^2’ is implemented in a similar way with ‘x^3’. In this case, first 2 bits are important to decide on whether to operate a modulus operation or not. Table 4.2 shows the results of modular multiplication with x^2.

Table 4.2: Multiplication with x^2.

(a_7,a_6)	Multiplication with x^2
‘00’	$a_5\ a_4\ a_3\ a_2\ a_1\ a_0\ 0\,0$
‘01’	$a_5\ a_4\ a_3\ \overline{a_2}\ \overline{a_1}\ a_0\ 1\,1$
‘10’	$a_5\ a_4\ \overline{a_3}\ \overline{a_2}\ a_1\ \overline{a_0}\ 1\,0$
‘11’	$a_5\ a_4\ \overline{a_3}\ a_2\ \overline{a_1}\ \overline{a_0}\ 0\,1$

4.2.2.2.3 Multiplication with x

Multiplication with ‘x’ was already implemented and it was named as ‘xtime’. Table 4.3 shows the result of multiplication with ‘x’.

Table 4.3: Multiplication with x.

(a_7)	Multiplication with ‘x’
‘0’	$a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0\ 0$
‘1’	$a_6\ a_5\ a_4\ \overline{a_3}\ \overline{a_2}\ a_1\ \overline{a_0}\ 1$

4.2.2.2.4 Multiplication with ‘09’

Multiplying with ‘$x^3 + 1$’ is achieved by the addition of the result coming from multiplication by ‘x^3’ and the byte itself. The addition operation is EXOR operation, so when two inverses are EXOR’ed, then the inverse signs cancel out. Let

- $k_7 = a_7 \oplus a_4$;
- $k_6 = a_6 \oplus a_3$;
- $k_5 = a_5 \oplus a_2$;
- $k_4 = a_4 \oplus a_1$;
- $k_3 = a_3 \oplus a_0$;
- $k_2 = a_2$;
- $k_1 = a_1$;
- $k_0 = a_0$.

Then the multiplication table of ‘09’ is:

Table 4.4: Multiplication with ‘09’.

(a_7,a_6,a_5)	Multiplication with $x^3 + 1$
‘000’	$k_7\ k_6\ k_5\ k_4\ k_3\ k_2\ k_1\ k_0$
‘001’	$k_7\ k_6\ k_5\ \overline{k_4}\ \overline{k_3}\ k_2\ \overline{k_1}\ \overline{k_0}$
‘010’	$k_7\ k_6\ \overline{k_5}\ \overline{k_4}\ k_3\ \overline{k_2}\ \overline{k_1}\ k_0$
‘011’	$k_7\ k_6\ \overline{k_5}\ k_4\ \overline{k_3}\ \overline{k_2}\ k_1\ \overline{k_0}$
‘100’	$k_7\ \overline{k_6}\ \overline{k_5}\ k_4\ \overline{k_3}\ \overline{k_2}\ k_1\ k_0$
‘101’	$k_7\ \overline{k_6}\ \overline{k_5}\ \overline{k_4}\ k_3\ \overline{k_2}\ \overline{k_1}\ \overline{k_0}$
‘110’	$k_7\ \overline{k_6}\ k_5\ \overline{k_4}\ k_3\ k_2\ \overline{k_1}\ k_0$
‘111’	$k_7\ \overline{k_6}\ k_5\ k_4\ k_3\ k_2\ k_1\ \overline{k_0}$

4.2.2.2.5 Multiplication with ‘11’

Multiplying with ‘$x^3 + x + 1$’ is achieved by the addition of the results coming from the multiplication by x^3, multiplication by ‘x’ and the byte itself. Let

- $p_7 = a_7 \oplus a_6 \oplus a_4$;
- $p_6 = a_6 \oplus a_5 \oplus a_3$;
- $p_5 = a_5 \oplus a_4 \oplus a_2$;
- $p_4 = a_4 \oplus a_3 \oplus a_1$;
- $p_3 = a_3 \oplus a_2 \oplus a_0$;
- $p_2 = a_2 \oplus a_1$;
- $p_1 = a_1 \oplus a0$;
- $p_0 = a_0$.

Then the multiplication table of ‘11’ is:

Table 4.5: Multiplication with ‘11’.

(a_7,a_6,a_5)	Multiplication with $x^3 + x + 1$
‘000’	$p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$
‘001’	$p_7\ p_6\ p_5\ \overline{p_4}\ \overline{p_3}\ p_2\ \overline{p_1}\ \overline{p_0}$
‘010’	$p_7\ p_6\ \overline{p_5}\ \overline{p_4}\ p_3\ \overline{p_2}\ \overline{p_1}\ p_0$
‘011’	$p_7\ p_6\ \overline{p_5}\ p_4\ \overline{p_3}\ \overline{p_2}\ p_1\ \overline{p_0}$
‘100’	$p_7\ \overline{p_6}\ \overline{p_5}\ \overline{p_4}\ p_3\ \overline{p_2}\ \overline{p_1}\ \overline{p_0}$
‘101’	$p_7\ \overline{p_6}\ \overline{p_5}\ p_4\ \overline{p_3}\ \overline{p_2}\ p_1\ p_0$
‘110’	$p_7\ \overline{p_6}\ p_5\ p_4\ p_3\ p_2\ p_1\ \overline{p_0}$
‘111’	$p_7\ \overline{p_6}\ p_5\ \overline{p_4}\ \overline{p_3}\ p_2\ \overline{p_1}\ p_0$

4.2.2.2.6 Multiplication with ‘13’

Multiplying with ‘$x^3 + x^2 + 1$’ is achieved by the addition of the results coming from the multiplication by x^3, multiplication by x^2 and the byte itself. Let

- $l_7 = a_7 \oplus a_5 \oplus a_4$;
- $l_6 = a_6 \oplus a_4 \oplus a_3$;
- $l_5 = a_5 \oplus a_3 \oplus a_2$;
- $l_4 = a_4 \oplus a_2 \oplus a_1$;
- $l_3 = a_3 \oplus a_1 \oplus a_0$;
- $l_2 = a_2 \oplus a_0$;
- $l_1 = a_1$;
- $l_0 = a_0$.

Then the multiplication table of ‘13’ is:

Table 4.6: Multiplication with ‘13’.

(a_7,a_6,a_5)	Multiplication with $x^3 + x^2 + 1$
‘000’	$l_7\ l_6\ l_5\ l_4\ l_3\ l_2\ l_1\ l_0$
‘001’	$l_7\ l_6\ l_5\ \overline{l_4}\ \overline{l_3}\ l_2\ \overline{l_1}\ \overline{l_0}$
‘010’	$l_7\ l_6\ \overline{l_5}\ l_4\ \overline{l_3}\ \overline{l_2}\ l_1\ \overline{l_0}$
‘011’	$l_7\ l_6\ \overline{l_5}\ \overline{l_4}\ l_3\ \overline{l_2}\ \overline{l_1}\ l_0$
‘100’	$l_7\ \overline{l_6}\ l_5\ \overline{l_4}\ \overline{l_3}\ l_2\ \overline{l_1}\ l_0$
‘101’	$l_7\ \overline{l_6}\ l_5\ l_4\ l_3\ l_2\ l_1\ \overline{l_0}$
‘110’	$l_7\ \overline{l_6}\ \overline{l_5}\ \overline{l_4}\ l_3\ \overline{l_2}\ \overline{l_1}\ \overline{l_0}$
‘111’	$l_7\ \overline{l_6}\ \overline{l_5}\ l_4\ \overline{l_3}\ \overline{l_2}\ l_1\ l_0$

4.2.2.2.7 Multiplication with ‘14’

Multiplying with ‘$x^3 + x^2 + x$’ is achieved by the addition of the results coming from the multiplication by x^3, multiplication by x^2 and the multiplication by x. Let

- $t_7 = a_6 \oplus a_5 \oplus a_4$;
- $t_6 = a_5 \oplus a_4 \oplus a_3$;
- $t_5 = a_4 \oplus a_3 \oplus a_2$;
- $t_4 = a_3 \oplus a_2 \oplus a_1$;
- $t_3 = a_2 \oplus a_1 \oplus a_0$;
- $t_2 = a_1 \oplus a_0$;
- $t_1 = a_1 \oplus a0$.

Then the multiplication table of ‘14’ is:

Table 4.7: Multiplication with ‘14’.

(a_7,a_6,a_5)	Multiplication with $x^3 + x^2 + x$
‘000’	$t_7\ t_6\ t_5\ t_4\ t_3\ t_2\ t_1\ 0$
‘001’	$t_7\ t_6\ t_5\ \overline{t_4}\ \overline{t_3}\ t_2\ \overline{t_1}\ 1$
‘010’	$t_7\ t_6\ \overline{t_5}\ t_4\ \overline{t_3}\ \overline{t_2}\ t_1\ 1$
‘011’	$t_7\ t_6\ \overline{t_5}\ \overline{t_4}\ t_3\ \overline{t_2}\ \overline{t_1}\ 0$
‘100’	$t_7\ \overline{t_6}\ t_5\ t_4\ t_3\ t_2\ t_1\ 1$
‘101’	$t_7\ \overline{t_6}\ t_5\ \overline{t_4}\ \overline{t_3}\ t_2\ \overline{t_1}\ 0$
‘110’	$t_7\ \overline{t_6}\ \overline{t_5}\ t_4\ \overline{t_3}\ \overline{t_2}\ t_1\ 0$
‘111’	$t_7\ \overline{t_6}\ \overline{t_5}\ \overline{t_4}\ t_3\ \overline{t_2}\ \overline{t_1}\ 1$

4.2.2.3 Inverse ShiftRow Implementation

This implementation is very similar to the Shift Row implementation. The only difference is the direction of the shifting. Instead of shifting to the left, the rows are shifted to the right.

4.2.2.4 Inverse AddRoundKey Implementation

This implementation is the same with the AddRoundKey implementation. These two modules are implemented separately to achieve high speed. Therefore, another set of EXOR gates is implemented for the Inverse AddRoundKey operation.

4.2.3 Key Generator Module

The Key Generator Module is responsible for generating the round keys and supplying these keys to the Decryption and Encryption Modules. The Key Generator Module is composed of 3 sub-modules. They are the Key Expansion Module, the Key Storage Module and the Key Selection Module. All the keys needed for encryption and decryption are generated by Key Expansion Module and they are stored in the Key Storage Module.

On the fly key generation is a method, which produces the keys needed for a round at every clock and does not store all the keys in a register. This decreases the number of the registers needed for the key generation. However, it is not practical for implementations of encryption and decryption modules supporting all key and data sizes of the Rijndael Algorithm. It may be good for implementations that make encryption only. For decryption implementations, the round keys are needed from the last to the first, requiring the key generation for both forward and reverse direction. So key generation becomes more complex, increasing the area and decreasing the speed of the chip. If all the key and data sizes are implemented, which is the case in our chip, then for some data-key sizes two key generations are needed at the same clock. This nearly doubles the timing of the critical path so the critical path of the overall chip is determined by the Key Generator Module. For implementing a very high speed Rijndael Processor, on-the-fly key generation is avoided. All the keys are generated and stored before encryption or decryption starts.

4.2.3.1 Key Expansion

The key expansion algorithm, which was described in Section 2.3.6.1, is different for key sizes of 128 bits, 192 bits and 256 bits. When the key size is 128 or 192 bits, then one ByteSub Transformation is enough for key generation of one round. If the key size is 256 bits, then two ByteSub Transformations are required and two consequent S-box operations increase the delay significantly. Therefore, it is preferred to implement half of the round key generation at one clock cycle for key size of 256 bits.

The expansion of all the keys needs at most 30 clock cycles and for our ASIC implementation, this costs a latency of only 228 ns, which is very low. And while the key expansion continues, the data, which will be ciphered, can be taken inside to overlap with the key generation. In this implementation, it is not preferred to overlap these two operations because of simplicity. Data is sequenced in after the key expansion ends.

4.2.3.2 Key Storage

All round keys are stored in a shift register of 3840 bits. Shift register is used to avoid addressing in writing to the register. The Key Expansion Module generates round keys, and they are fed to the shift register. When Nk = 4 or Nk = 8, then at every clock 128-bit keys are generated. When Nk = 6, at every clock 192-bit keys are generated. Our shift register can be thought as a parallel shift register having 20 stages and all the stages are 192-bits long. So if Nk = 4 or Nk = 8, then at every 3 clock cycle 3x128 = 384 bits are produced so the shift register shifts 2 times. When Nk = 6, at every clock 192-bit keys are produced so the shift register always shifts. When all the keys are generated, the shift register ends shifting.

4.2.3.3 Key Selection

At every round, proper keys must be fed to the Encrypt and Decrypt Modules. When Nk = 4, the keys are selected in chunks of 128 bits. When Nk = 6, chunks of 192 bits and when Nk = 8, chunks of 256 bits are needed. For encryption, these keys are selected starting from the first place. If the mode is decryption, then these keys are selected from the last to the first. This module contains many multiplexers to achieve selection. The delay for the data to pass through these multiplexers is high. If the Encryption or Decryption Module waits for the incoming key, then the delays of Encryption-Decryption Modules and the Key Selection Module will be are added. To prevent this situation, the round keys are registered at every clock so Encryption or Decryption Modules does not be affected by the latency of the round keys.

4.2.4 Data Interface

The chip has 24 inputs and 18 outputs. Figure 4.10 shows the data interface.

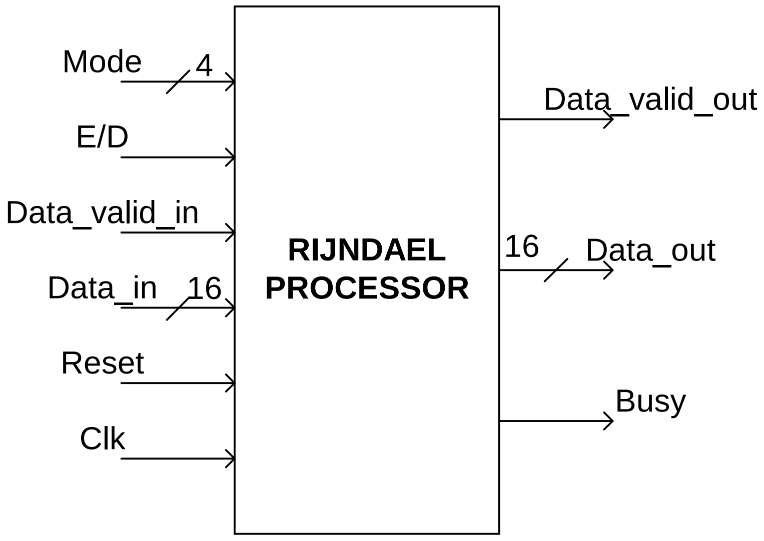


Figure 4.10: Data interface of the Rijndael Processor.

There are 16-bit parallel data input and data output, which are synchronized to the system clock. Data input and output are separated so while new data is being taken inside, the processed data can be send to the output. At the same time the processing of the data continues. This increases the throughput, so no clock cycles are wasted for the data interface. There is a data valid input indicating when the input data should be valid. In addition, a separate data valid output indicates whether the output data is valid or not.

There is an input named 'E/ND', to select between encrypt or decrypt modes of the chip. If this input is high, the chip operates as an encryption processor and if it is low, the chip makes decryption. In addition, there is a 4-bit input named 'mode' to choose the data and key sizes of the procedure. There are nine different combinations of data and plaintext so 4 bits are needed to separate them. Table below shows the modes for all the data-key length combinations.

Table 4.8: Operation modes.

Mode	Data length	Key length
0000	128	128
0001	192	128
0010	256	128
0011	128	192
0100	192	192
0101	256	192
0110	128	256
0111	192	256
1000	256	256

There is an output named 'BUSY' to show that the chip will not accept any data. In some data-key combinations, processing the data is longer then taking the data. For example, when data is 128-bits long and key is 256-bits long, then the encryption or decryption process is completed at 14 clock cycles. However, data reception takes only eight clock cycles so BUSY output is activated for 6 clock cycles.

4.2.5 Controller Module

The Rijndael Processor has a Controller Module that controls the encryption-decryption flow and the data interface. There are four different state registers in this module for controlling the operations.

The first state register is a 5-bit register. Four bits hold the mode of the chip that is they hold the sizes of the plaintext and the key. There are nine different combinations of key and plaintext size, so a 4-bit register is required to hold the mode. The last bit is used to determine if the chip operates as an encryption chip or a decryption chip.

The second state register is used to hold the round number. There are at most 14 rounds so a 4-bit register is enough to count the rounds. The third and fourth state registers are for the input and the output interfaces. They count the number of the plaintext data taken inside and the ciphertext data sent to the output. 4-bit state registers are enough for these operations.

The process of encryption or decryption starts with resetting the chip. The mode of operation can be changed only when the chip is reset. After reset, the first valid data will be the key. The key is read by using the 16-bit data input so it takes 8 clock cycles for 128-bit key, 12 clock cycles for 192-bit key and 16 clock cycles for 256-bit key. When the key is completely read, the BUSY output goes HIGH indicating that the chip will not accept data anymore. After key expansion is completed, the BUSY output is asserted LOW so the chip can accept data to encrypt or decrypt. When a block of data is read, the operation starts. The chip can accept new data while the previous data processing continues.

There are two 256-bit registers for data buffering. The first buffer is used to store the read data, and the second one is used to store the encrypted or decrypted data. Controller Module is responsible for the control of these buffers. When the input buffer is full, the Controller Module activates the BUSY output and does not accept any more data. When the processing of the current data ends, the Controller Module lets the data in the input buffer to enter the Encryption or Decryption Module. After this, the input buffer becomes empty so the Controller Module deactivates the BUSY output so the chip can accept new data. The processed data is taken to the output buffer and the Controller Module immediately starts sending the processed data to the output. At every clock it sends 16-bit data to the output. The operations of taking new data, processing current data and sending the processed data are overlapped to increase the throughput of the chip.

4.3 ASIC Implementation

The Rijndael Processor is implemented as an ASIC using 0.35 μm standard CMOS technology. 0.35 μm is the smallest length of a transistor that can be produced using this technology. The implementation of the chip comprises of two steps. The first step is the synthesis of the design. The second step is the implementation of the synthesized logic as an ASIC. These procedures are described in the next sections.

4.3.1 Synthesis of the Design

The Rijndael Processor is designed using Verilog-HDL. Verilog-HDL is a Hardware Description Language and it is used to describe combinational and sequential circuits. Modular Design Methodology is used in the design. The design is implemented from bottom to top.

The Verilog-HDL codes of the modules are synthesized using Synopsys Design Analyzer. Synthesizing is the process of generating logic from the code and mapping this logic to a specified library. 0.35 μm standard CMOS technology is targeted for the ASIC implementation. 0.35 μm standard cell libraries MTC 45000 and MTC 45200, which are libraries of AMI Semiconductor [17], are used. Worst Case Operating Conditions are assumed during the synthesis of the modules.

The synthesis constraints are given in a way to increase the speed of the chip for a limited area. The area of the chip can be expressed as the number of standard cells in the chip. Usually single NAND gate is considered as a unit and the area of the chip is expressed as equivalent number of NAND gates in the chip.

The critical thing for the synthesis procedure is the selection of proper wire load models for the nets. Wire load models are used for estimating the routing delays prior to layout phase of the ASIC implementation. If the outputs of a module should go to long distances in the chip, than a single gate cannot drive all the wire capacitances added up to the fanout, hence the speed of the signals decrease. Selecting higher wire load models makes the Synthesis Tool to insert some buffers to the nets. If the wire load model is selected lower then required, than the buffers cannot drive the net fast enough. If the wire load model is selected higher than required, than the extra useless buffer delays are added to the delay of that net. While synthesizing a module, if the outputs of that module do not travel long distances, small wire load models are selected for that module. If the signals in a module distribute to the whole chip and travel long distances, then higher wire load models are selected for that module. The size of the module is also important for selecting the proper wire load models. For smaller modules, smaller wire load models are selected. To understand whether the proper wire load model is selected or not, the whole implementation must be completed and the delays must be back annotated. Therefore, many iterations are done to decide on the proper wire load models.

The S-boxes and the Inverse S-boxes are the most critical components in the design. They are critical for both area and the speed of the chip. There are 32 S-boxes and 32 Inverse S-boxes in the design. Therefore, the area of the single S-box and the single Inverse S-box are multiplied with 32. Having a fast logic with small area is critical. After several iterations, an optimum area-speed combination is found for these S-boxes. In our design an S-box utilizes 813 gates and its input-output delay is 3 ns.

Inverse S-box implementation is slightly faster than the S-box implementation. Inverse S-box utilizes 1100 gate and its input-output delay is 2.7 ns. Inverse S-boxes are made faster than S-boxes to compensate the time difference between Encryption and Decryption Implementations. Decryption operations are more complex decreasing the speed of the module. Balancing the speeds of all the paths is important to obtain an efficient implementation. The slowest path determines the speed of the chip, and in our implementation, the slowest path is in the Decryption Module. Therefore, for increasing the speed of that module and balancing the critical paths of all the modules, Inverse S-boxes are made slightly faster than S-boxes. Table 4.9 shows the synthesis results of the main modules.

Table 4.9: Synthesis results.

Encryption Module	33k
Decryption Module	49k
Key Generator Module	50k
Controller Module and Buffers	17k
Total	149k

4.3.2 Placement and Routing

Placement is the process of placing the standard logic cells on the silicon wafer and routing is the process of connecting these standard cells with each other to complete the circuit. Placement and routing of small circuits can be done manually. However, this chip is too big for manual placement and routing. Cadence's Silicon Ensemble Tool [18] is used for automatic placement and routing.

The Rijndael Processor has 24 inputs and 18 outputs. Therefore, 68-pin PLCC packet is chosen for this chip. 42 pins are used as input-output pins, 13 pins are used as VDD and the remaining 13 pins are used as GND input.

The first step of the placement is to import the synthesized netlist to the Silicon Ensemble tool. The synthesized netlist contains only the input-output pins so we need to import the VDD and GND pins separately. In addition, corner cells are inserted. The placement is a timing-driven placement for achieving a fast circuit so the third step is to import the system constraints, which determines the speed constraint of the chip.

After these steps, the floorplan must be initialised. The placement tool places the standard logic cells in rows. The row utilization constraint determines how many percent of the rows will be filled with the standard cells. If the rows are filled too tightly, the clock distribution gets difficult. There must be some spaces for inserting clock buffers during clock tree generation. In addition, the routing of the signals cannot be satisfied if the standard logic cells are placed too tightly. If row utilization decreases, the area of the chip becomes higher and the cost of the chip increases. Finding the optimum row utilization is important to achieve a compact and low-cost implementation. After several iterations, 85% row utilization is found to be optimum.

Another critical thing for floorplan initialisation is to place the input-output pads at a proper distance from the core of the chip. This is required for the placement of power rings between IO pads and the core. 70 μm spacing is considered to be enough for the insertion of two power rings, which are VDD and GND. Figure 4.11 shows the floorplan initialisation for 85% row utilization and 70 μm pad spacing.

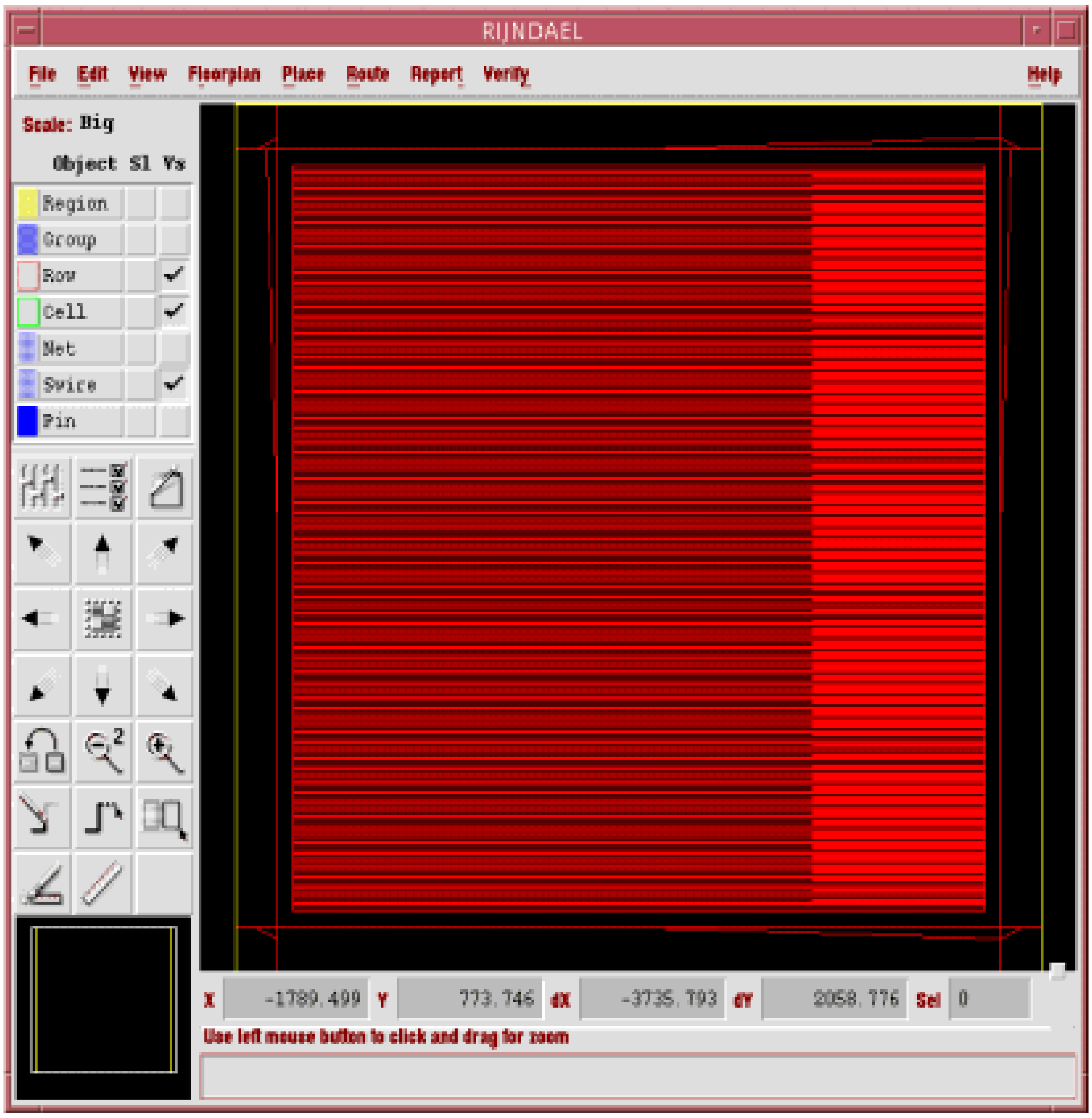


Figure 4.11: Floorplan initialisation.

The next step is the placement of the input-output pads. They are placed center-abutted. Figure 4.12 shows the chip after IO placement.

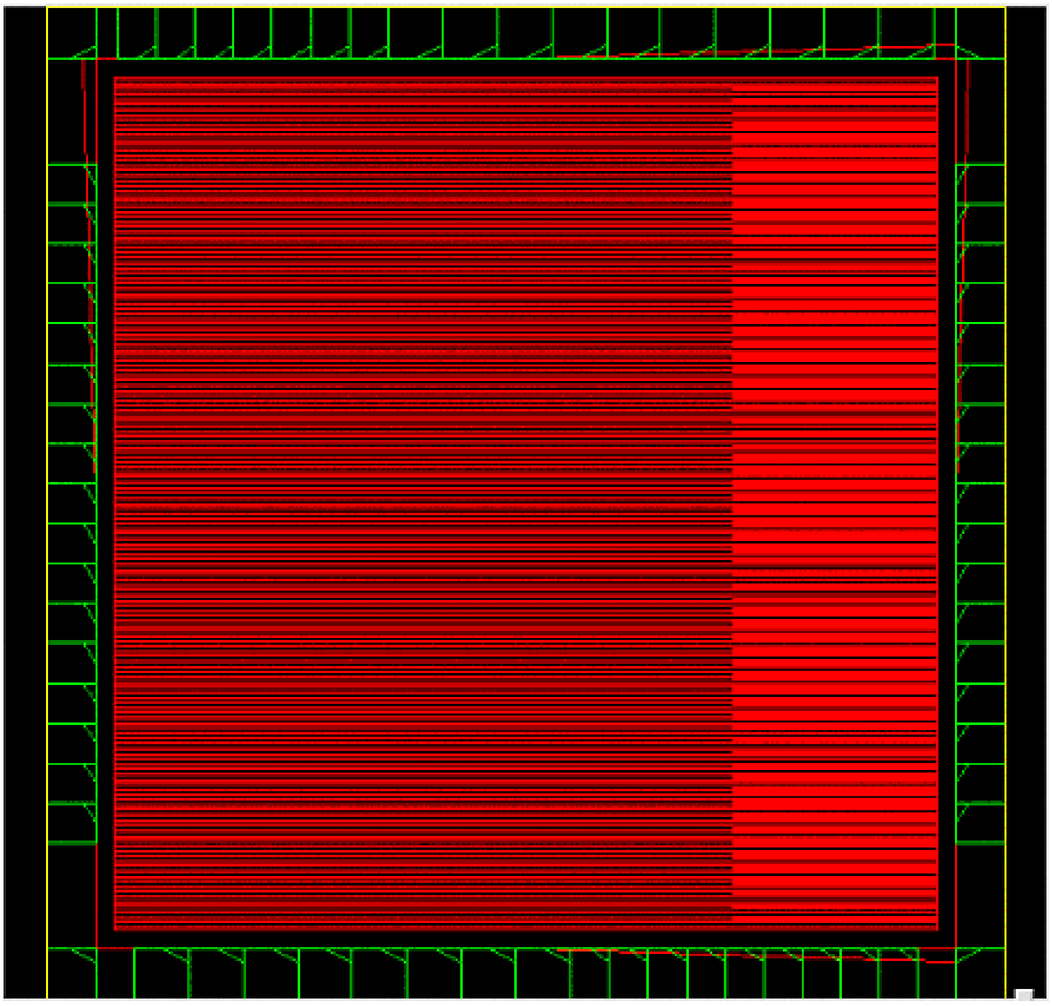


Figure 4.12: IO placement completed.

After the placement of the input-output pads, the logic cells are placed. It is a timing-driven placement for achieving a fast operating chip. Figure 4.13 shows the chip after placement of the standard logic cells.

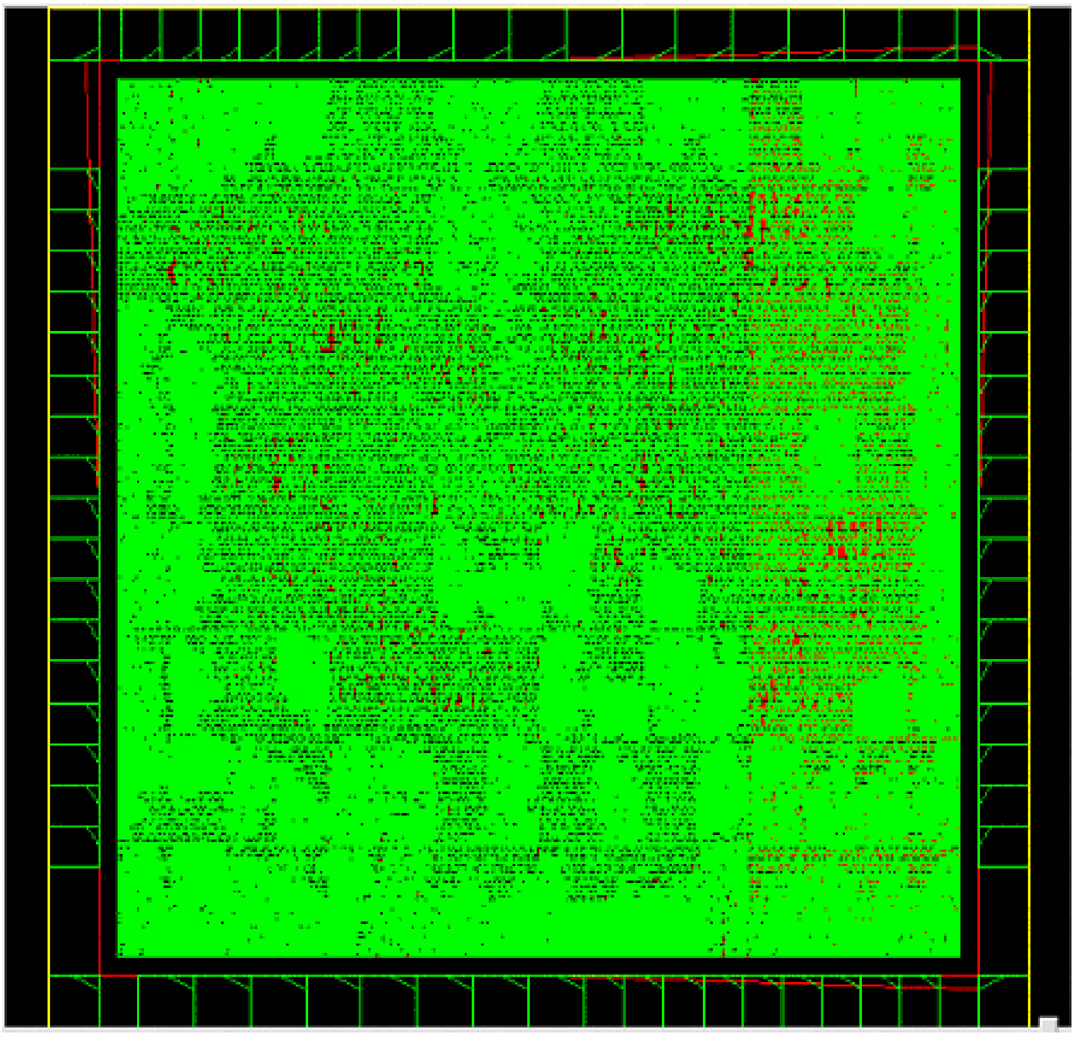


Figure 4.13: Cell placement completed.

Next step is the clock tree generation. Clock distribution is a problem for especially high-density chips. Clock skew must be minimized for proper operation of the chip. The maximum clock skew constraint is 0.1 ns in our implementation. After the clock tree generation, the actual clock skew becomes 0.09 ns.

For clock tree generation, some buffers are added to the rows. However, some spaces still remain in the chip. Core filler cells are added to fill these spaces. These cells do not have any input-output pins. They provide the connection of VDD and

GND lines between the adjacent cells. In addition, some IO filler cells are added to surround the chip with the IO pads. Connection of the VDD and GND lines of the IO pads are satisfied with the addition of these IO filler cells. Figure 4.14 shows the chip after clock tree generation and filler cell addition.

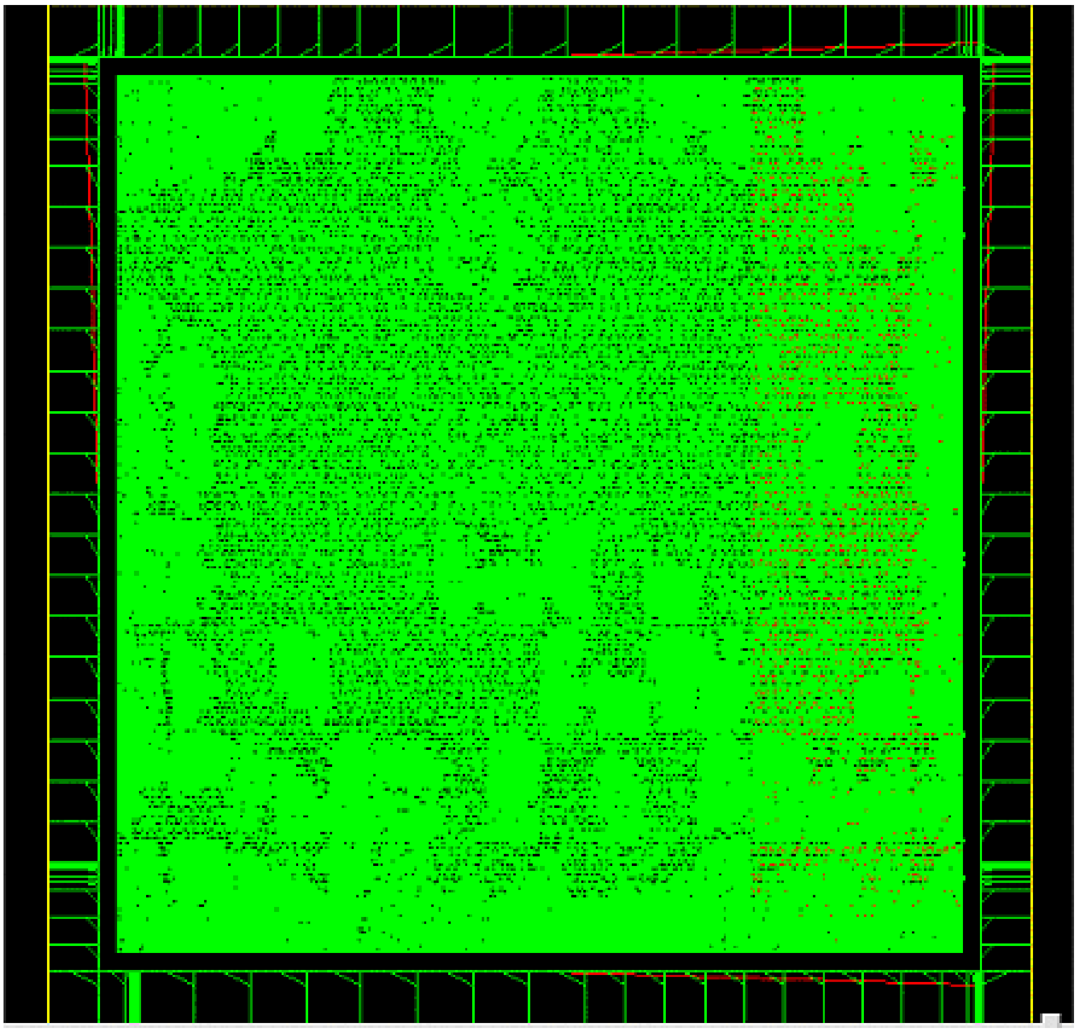


Figure 4.14: Filler cell addition and clock tree generation completed.

There are 13 GND and 13 VDD pads in this design. For the connection of the power lines of the rows, power rings are added to the design. GND and VDD pads are connected to these rings. Figure 4.15 shows the chip after the insertion of the power rings and connection of these rings to the power pins. The input-output pads are shown and this is the final view of the chip.

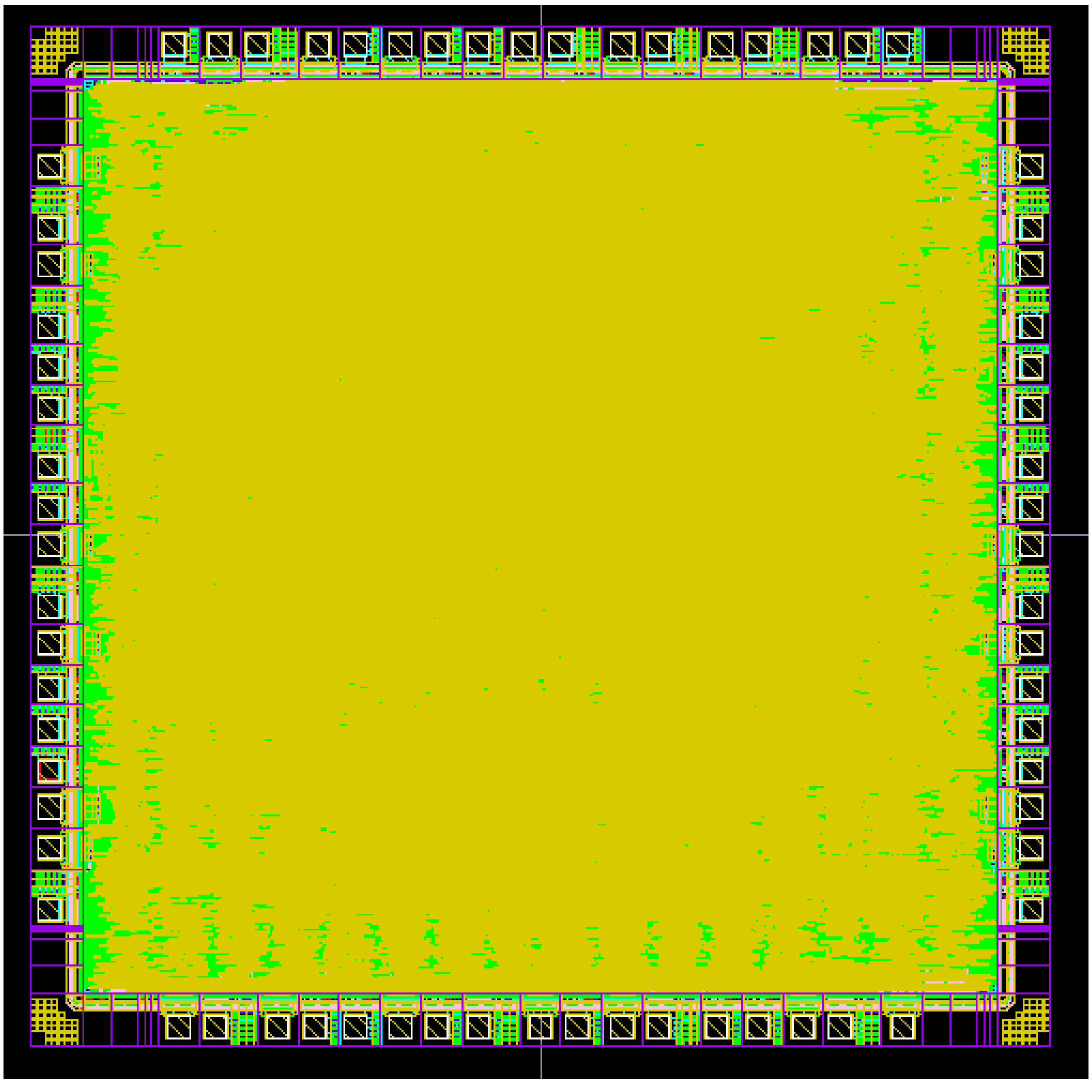


Figure 4.15: Power rings and final view of the chip.

The next step is to connect all the nets between standard logic cells. This process is named as WROUTE. There are five metal layers available in the 0.35 µm CMOS technology for routing. In some places of the chip, the routing density is very high and the auto-router cannot manage to route all the nets by using five metal layers.

Therefore, some short circuits between nets occur. These short circuits must be repaired using Search and Repair tool of the Silicon Ensemble. This tool makes several iterations about connecting these shorted nets. Another important thing in the routing process is the Antenna Effect. Long metal wires may store electrical charge during the fabrication of the chip. This excess charge may burn out the gate of the logic. Therefore, the metal wires must be short enough to limit this excess charge storage. When a wire in a metal layer becomes long, the auto-router passes to another layer to control this Antenna Effect. Controlling the Antenna Effect increases the yield of the fabrication process.

This step concludes the implementation of the chip as an ASIC. Design Rule Check (DRC) and Electrical Rule Check (ERC) tests are completed using Cadence environment. These tests are also done by the AMI Semiconductor before fabrication of the chip.

4.4 FPGA Implementation

The Rijndael Processor is also implemented as a Field Programmable Gate Array (FPGA) design. The Rijndael Processor design is optimised for ASIC implementation, not for FPGA implementation. The purpose of implementing the design in an FPGA is for fast verification of the design in a prototyping hardware. TÜBİTAK-ODTÜ-BİLTEN [19] provides the facilities for design, programming and testing of the FPGA's. Celoxica [20] RC1000 environment of TÜBİTAK-ODTÜ-BİLTEN is used for hardware tests of the Rijndael Processor. Test results are given in Section 4.5.

FPGA's are programmable devices. They are composed of basic programmable building blocks. Xilinx Virtex 2000E SRAM based, reconfigurable FPGA's are used for this implementation. Synopsys FPGA Express tool is used for synthesis and Design Manager release 3.3.08.i is used for placement and routing of the design. Target device for implementation is Xilinx XCV2000EBG560-6 which is mounted on Celoxica RC1000 test card XCV2000EBG560 device is used.

Since the FPGA implementation effects only the physical design phase, same Verilog codes are used for synthesis. No design change is done to optimise the design for FPGA implementation. In fact, the ROM and RAM facilities of Xilinx FPGA's are very useful especially for the S-Box implementation and the key storage, but using them requires design modifications so it is not preferred for this verification. The FPGA implementation has a speed of 31 MHz. Table 4.10 shows the elements used.

Table 4.10: Components utilized.

Number of Slices	13,309 out of 19,200	69%
Number of Flip-Flops	5,481 out of 38,400	14%
Number of 4 input LUTs	22,495 out of 38,400	58%
Number of bonded IOBs	40 out of 404	9%
Number of GCLKs	2 out of 4	50%

4.5 Experimental Results and Simulations

Several simulations have been done throughout the design of the Rijndael Processor. Basically simulations are performed at:

i. Behavioural Simulations
ii. Post-Synthesis Simulations
iii. Post-Layout Simulations

Behavioural simulations are done at the logical design phase of the Rijndael Processor. In the behavioural simulations, source codes are functionally simulated. The logic gate delays are not included in this kind of simulations.

After the synthesis of the design, post-synthesis simulations are done. There are libraries showing the single gate delays for all components. These delays are measured for specific temperatures by the foundry. The total gate delays are calculated from the synthesized logic and these delays are included for post-synthesis simulations.

For post-layout simulations, the routing capacitances and the delays coming from these capacitances must be added to the net delays. Routing capacitances are exported to the simulator by back annotation. A delay report containing the wire delays is generated and this report is used during post-layout simulation of the chip. This is the last simulation step and it is the closest one to the real operation.

The speed of the chip is calculated by static timing analysis. To make static timing analysis, these wire delays and the netlist are read by Synopsys Design Analyzer tool and the speed of the chip is calculated with this tool. Our implementation operates at a minimum clock speed of 7.6 ns, which is equal to 132 MHz. Throughput values for this implementation are shown in Table 4.11. The chip has an initial latency of 30 clock cycles for key expansion. A single data block is processed in at most 14 clock periods. The comparisons of this implementation with other implementations are explained in Section 4.6. The operating power is estimated as 200 mW.

Table 4.11: Throughput values for different data-key lengths.

Throughput	Key Length = 128	Key Length = 192	Key Length = 256
Data Length = 128	1.69 Gbit/s	1.41 Gbit/s	1.21 Gbit/s
Data Length = 192	2.11 Gbit/s	2.11 Gbit/s	1.81 Gbit/s
Data Length = 256	2.41 Gbit/s	2.41 Gbit/s	2.41 Gbit/s

The FPGA implementation is done for quick verification of the design in a prototyping hardware. For this purpose, a test setup consisting of a logic analyser (Tektronix TLA 715) [21], a PC and an FPGA test card (Celoxica RC1000) is used. Tektronix logic analyser has a pattern generator and the test vectors are produced by this pattern generator. The FPGA in the Celoxica RC1000 test card is programmed to operate as Rijndael Processor. PC is used to load the program file to the FPGA. Test vectors are sent to the inputs of the FPGA and the outputs are observed using the logic analyser. Figure 4.16 shows the test setup.

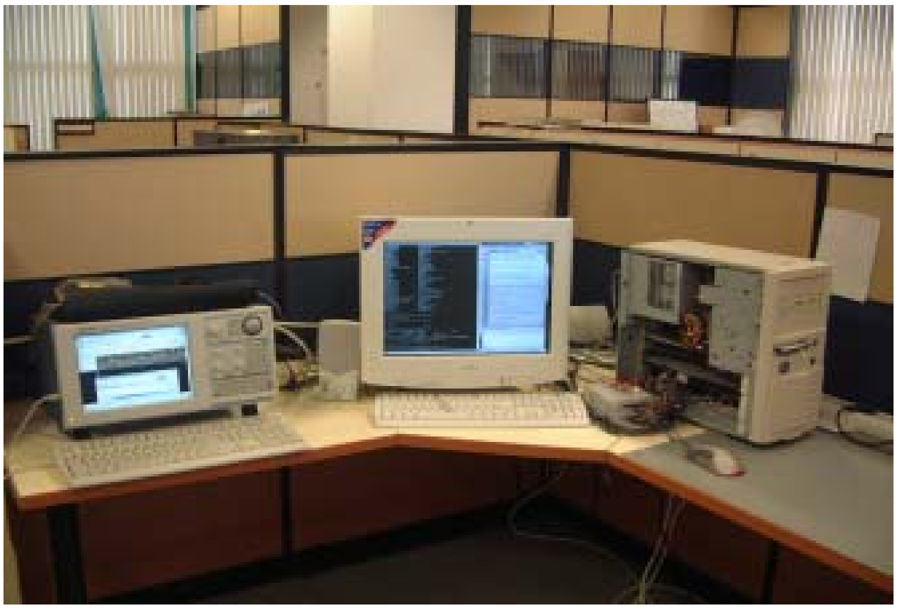


Figure 4.16: Test setup.

The Celoxica RC1000 test card is shown in the figure below.

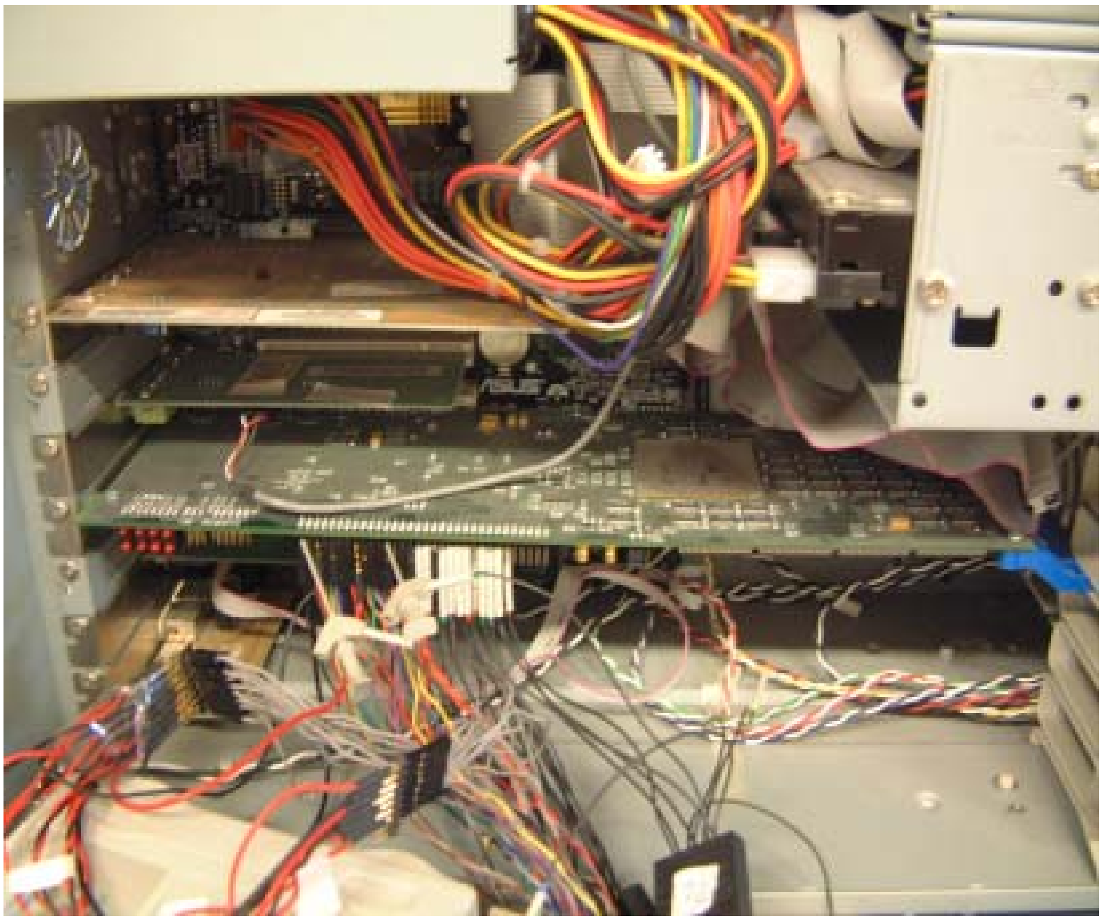
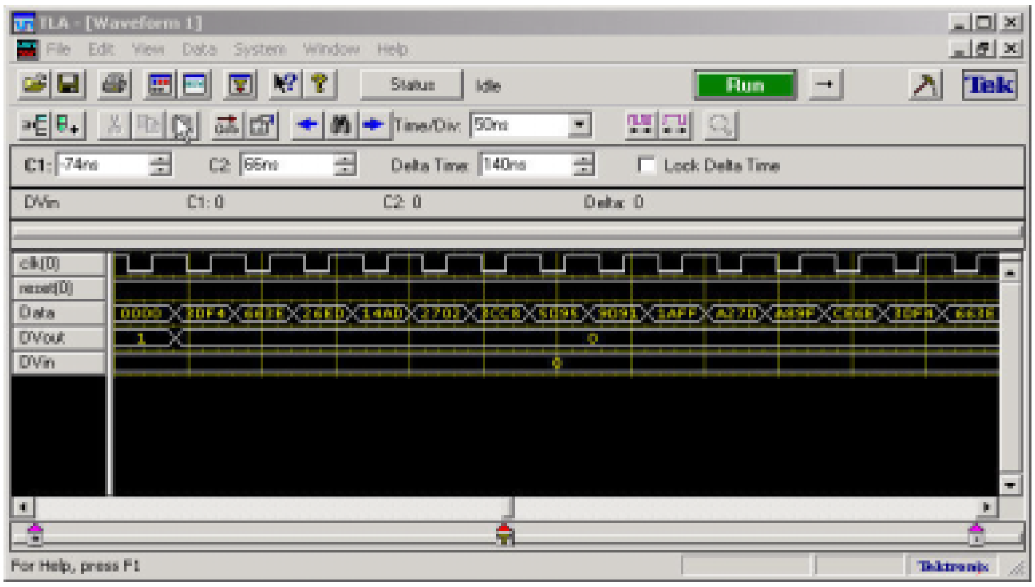


Figure 4.17: Celoxica RC1000 card.

A sample output for an encryption of a 192-bit data block with a 128-bit key is shown in the figure below. Here the input data and key is selected to be all ones.



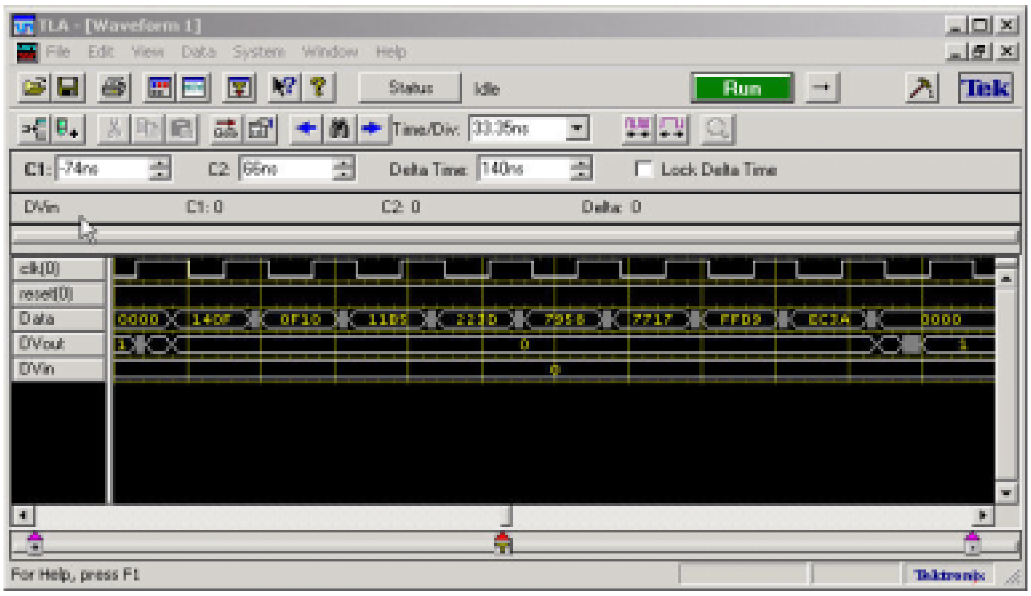
Time/Div: 50ns

C1: -74ns C2: 66ns Delta Time: 140ns Lock Delta Time

DVin C1: 0 C2: 0 Delta: 0

Figure 4.18: Encryption output from logic analyser.

A sample output for a decryption of a 128-bit data with 128-bit key is shown in figure 4.19. For simplicity, the data and key block is selected as all zeros.



Time/Div: 33.35ns

C1: -74ns C2: 66ns Delta Time: 140ns Lock Delta Time

DVin C1: 0 C2: 0 Delta: 0

Figure 4.19: Decryption output from logic analyser.

4.6 Comparison of Different Implementations

There are many published studies about the implementation of the new AES Standard, Rijndael Algorithm. Some of these studies were explained in Section 3. In this section, the comparison between these implementations and our implementation is presented.

Within our knowledge, the ASIC processor presented in this study is the fastest published processor to date for both encryption and decryption. It has 2.41 Gbit/s throughput at a worst-case clock speed of 132 MHz. Table 4.11 shows the throughput values for all the data-key length combinations. The implementation utilizes 149k gates to achieve this speed. It includes both the encryption and the decryption modules. The design is implemented using 0.35-μm CMOS technology.

The second fastest non-feedback implementation after our implementation is another ASIC implementation [5]. This ASIC implementation is implemented using 0.18-μm standard CMOS technology, which is smaller and faster than 0.35-μm technology. This implementation utilizes 173k gates and includes only the encryption implementation. Although this implementation uses smaller, hence faster technology and implements only the encryption block by using 173k gates, they reach to 2.29 Gbit/sec throughput. Our implementation, however, utilizes 149k gates and using older technology, we reached to 2.41 Gbit/sec throughput for both encryption and decryption. Table 4.12 shows the comparison between this implementation and our implementation.

Table 4.12: Comparison of our implementation with implementation in [5].

Architecture	Process	Technology	Number of Gates	Frequency	Throughput
ASIC in [5]	Enc.	0.18-μm	173k gates	125 MHz	2.29 Gbit/s
Rijndael Processor	Enc/Dec	0.35- μm	149k gates	132 MHz	2.41 Gbit/s

In an FPGA study [8], both pipelined and non-pipelined implementations are presented. Two different pipeline architectures, one of them using only inner round pipelining and the other one using mixed inner and outer round pipelining, are presented. Without using pipelining, 414 Mbit/s throughput is achieved. By using inner round pipelining, they achieve 1.265 Gbit/sec and by using full mixed inner and outer round pipelining they reach to 12.160 Gbit/sec.

There are feedback modes specified by NIST for encryption purposes. Figure 4.20 shows these feedback operation modes.

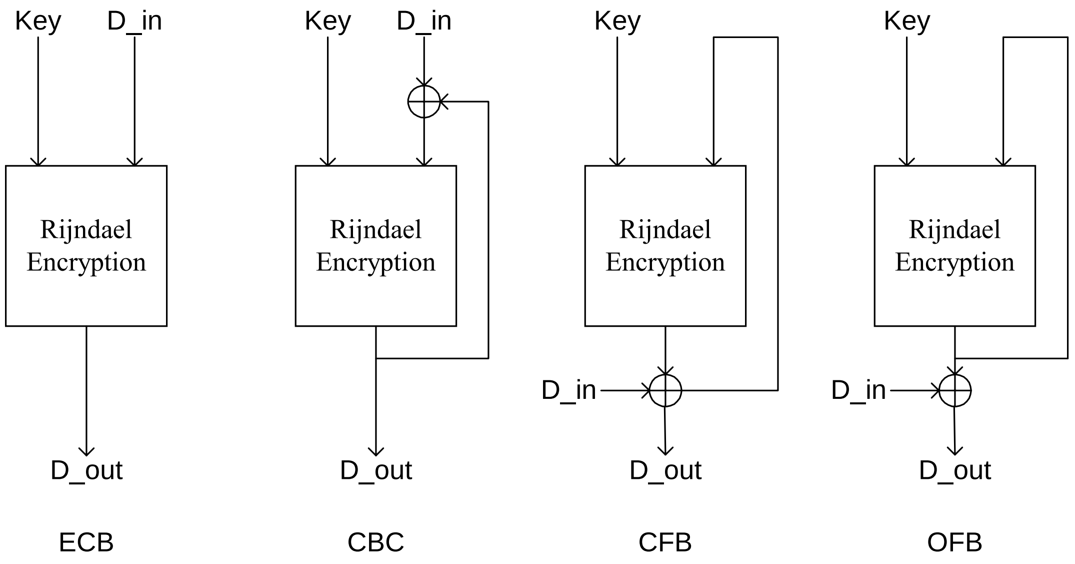


Figure 4.20: Feedback modes of operation.

When the encryption processor is used in the feedback modes, then pipelining does not give any benefit. In this case, the total throughput must be divided with at least 10, giving smaller throughput values. Also area-throughput efficiency decreases for these pipeline implementations.

Table 4.13 shows the comparison of our Rijndael Processor implementation with the implementations described in Section 3. The areas of the FPGA implementations are explained as the total number of Configurable Logic Blocks (CLBs) utilized. The areas of the ASIC implementations are expressed as the total number of ‘nand’ equivalent logic gates.

Table 4.13: Results comparisons.

Architect.	Process	Design	Tech.	Number of Gates	Frequency	Throughput
H. Kuo & et. al. [5]	Enc.	ASIC	0.18 µm	173k	125 MHz	2.29 Gbit/s
K. Gaj & et. al. 1 [8]	Enc.	FPGA	_	2507 CLB	32 MHz	414 Mbit/s
K. Gaj & et. al. 2 [8]	Enc.	FPGA	_	12.6k CLB	95 MHz	12.2 Gbit/s Pipelined
N. Sklavos & et. al. 1 [9]	Enc/Dec.	FPGA	_	2358 CLB	22 MHz	259 Mbit/s
N. Sklavos & et. al. 2 [9]	Enc/Dec.	FPGA	_	17.3k CLB	28.5 MHz	3.65 Gbit/s Pipelined
McLoone & et. al. 1 [10]	Enc.	FPGA	_	2.7kCLB+ 82 RAM	54.4 MHz.	7 Gbit/s Pipelined
McLoone & et. al. 2 [10]	Dec.	FPGA	_	4.3k CLB + 82 RAM	49.9 MHz	6.38 Gbit/s Pipelined
H. Shim & et. al. [13]	Enc/Dec	FPGA	_	2580 CLB	38.8 MHz	452 Mbit/s
Ichikawa & et. al. [14]	Enc	ASIC	0.35 µm	613k	15.2 MHz	1.95 Gbit/s
R. Karri & et. al. [16]	Enc	FPGA	_	3973 CLB	47 MHz	137 Mbit/s
Rijndael Processor	Enc/Dec	ASIC	0.35 µm	149k	132 MHz	2.41 Gbit/s

Chapter 4 presented the implementation details of the Rijndael Processor including the synthesis, place-route and simulation steps of the design. Chapter 5 gives a conclusion of the study.

CHAPTER 5

CONCLUSION

In this thesis study, Rijndael Algorithm is implemented as an ASIC and published in [22]. Both the encryption and decryption algorithms are implemented as a single chip. Although the data size is fixed to 128 bits in the Advanced Encryption Standard, this implementation supports all the key and data length combinations of the original Rijndael Algorithm.

Semi-custom design techniques are used for the implementation. The hardware is described using Verilog-HDL. The VLSI implementation targets 0.35-μm standard CMOS technology of the AMI Semiconductor Company. The area of the chip is 12.8 mm^2 including the input-output pads. The chip was sent to fabrication in June 2003 and it is expected to arrive on November 2003.

The aim for this implementation is to achieve the fastest non-pipelined Rijndael ASIC implementation. The implementation utilizes 149k nand equivalent gates and it has a worst-case operating frequency of 132 MHz giving a throughput of 2.41 Gbit/s, which is the fastest non-pipelined result for both encryption and decryption.

Many optimizations are done to achieve these results. The critical paths of all the modules are balanced to decrease the gate count of the design. Many iterations are done in both synthesis and placement-routing steps for finding the optimum area-speed combination. Another important issue for achieving these results is pre-calculating the round keys. On-the-fly key generation, as used in the previous implementations, requires two cascaded S-box operations that nearly double the critical path of the design.

As a future work of this study, data size can be fixed to 128 bits. This nearly decreases the area of the chip to half. The gain from the area can be used to implement a faster design.

REFERENCES

[1] "Data Encryption Standard," available from: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[2] www.rsasecurity.com

[3] National Institute of Standards and Technology: http://csrc.nist.gov/

[4] J.Daemen and V.Rijmen, "AES Proposal: Rijndael, AES algorithm submission," September 3, 1999, available: http://csrc.nist.gov/CryptoToolkit

[5] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and Performance Testing of a 2.29-GB/s Rijndael Processor," *IEEE Journal of Solid State Circuits*, vol. 38, No. 3, pp. 569-572, March 2003.

[6] H. Kuo, I. Verbauwhede, and P. Schaumont, "A 2.29 Gbits/sec, 56 mW Non-Pipelined Rijndael AES Encryption IC in a 1.8V, 0.18 μm CMOS Technology," *IEEE Custom Integrated Circuits Conference*, May 2002.

[7] P. Schaumont, H. Kuo, and I. Verbauwhede, "Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor," *39th Design Automation Conference*, June 2002.

[8] P. Chodowiec, P. Khuon and K. Gaj, "Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining," *Proc.*

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA'01, Monterey, CA, February 2001.

[9] N. Sklavos and O. Koufopavlou, "Architectures and VLSI Implementations of the AES-Proposal Rijndael," *IEEE Transactions on Computers,* vol. 51, Issue 12, pp. 1454-1459, 2002.

[10] M. McLoone and J. McCanny, "Single-chip FPGA Implementation of the Advanced Encryption Standard Algorithm," in *Proc. 11th Int. Conf. Field-Programmable Logic and Applications (FPL 2001),* LNSC 2147, pp. 152-161, 2001.

[11] Xilinx web page: www.xilinx.com

[12] Synplicity web page: www.synplicity.com

[13] J. H. Shim, D. W. Kim, Y. K. Kang, T.W. Kwon and J. R. Choi, "A Rijndael Cryptoprocessor Using Shared On-the-fly Key Scheduler," available:

www.ap-asic.org/2002/proceedings/2B/2B-3.PDF

[14] T. Ichikawa, T. Kasuya, and M. Matsui, "Hardware Evaluation of the AES Finalists," in *Proc. 3rd AES Candidate Conference*, pp. 279-285, New York, April 2000.

[15] Synopsys web page: www.synopsys.com

[16] R. Karri, K. Wu, P. Mishra, and Y. Kim, "Concurrent Error Detection Schemes for Fault-Based Side-Channel Cryptanalysis of Symmetric Block Ciphers," *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, No. 12, December 2002.

[17] AMI Semiconductor web page: www.amis.com

[18] CADENCE web page: www.cadence.com

[19] TÜBİTAK-ODTÜ-BİLTEN web page: www.bilten.metu.edu.tr

[20] Celoxica web page: www.celoxica.com

[21] Tektronix web page: www.tektronix.com

[22] Refik Sever, A. Neslin İsmailoğlu, and Yusuf C. Tekmen, "A 2.41 Gb/s ASIC Implementation of the Rijndael Encryption Algorithm," *Proceedings of the Work in Progress, Euromicro Symposium on Digital System Design, DSD2003*, Belek, Turkey, September 2003.

APPENDIX A

A.1 AES S-Box

	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

APPENDIX B

CELOXICA RC1000 HARDWARE

B.1. Overview

The RC1000-PP hardware platform is a standard PCI bus card equipped with a XILINX® Virtex TM family BG560 part with up to 1,000,000 system gates . It has 8Mb of SRAM directly connected to the FPGA in four 32 bit wide memory banks. The memory is also visible to the host CPU across the PCI bus as if it were normal memory. Each of the 4 banks may be granted to either the host CPU or the FPGA at any one time. Data can therefore be shared between the FPGA and host CPU by placing it in the SRAM on the board. It is then accessible to the FPGA directly and to the host CPU either by DMA transfers across the PCI bus or simply as a virtual address. The board is equipped with two industry standard PMC connectors for directly connecting other processors and I/O devices to the FPGA; a PCI-PCI bridge chip also connects these interfaces to the host PCI bus, thereby protecting the available bandwidth from the PMC to the FPGA from host PCI bus traffic. A 50 pin unassigned header is provided for either inter-board communication, allowing multiple RC1000-PPs to be connected in parallel or for connecting custom interfaces. The support software provides Linux(Intel), Windows®98 and NT®4.0+ drivers for the board, together with application examples written in Handel-C, or the board may be programmed using the XILINX® Alliance Series and Foundation.

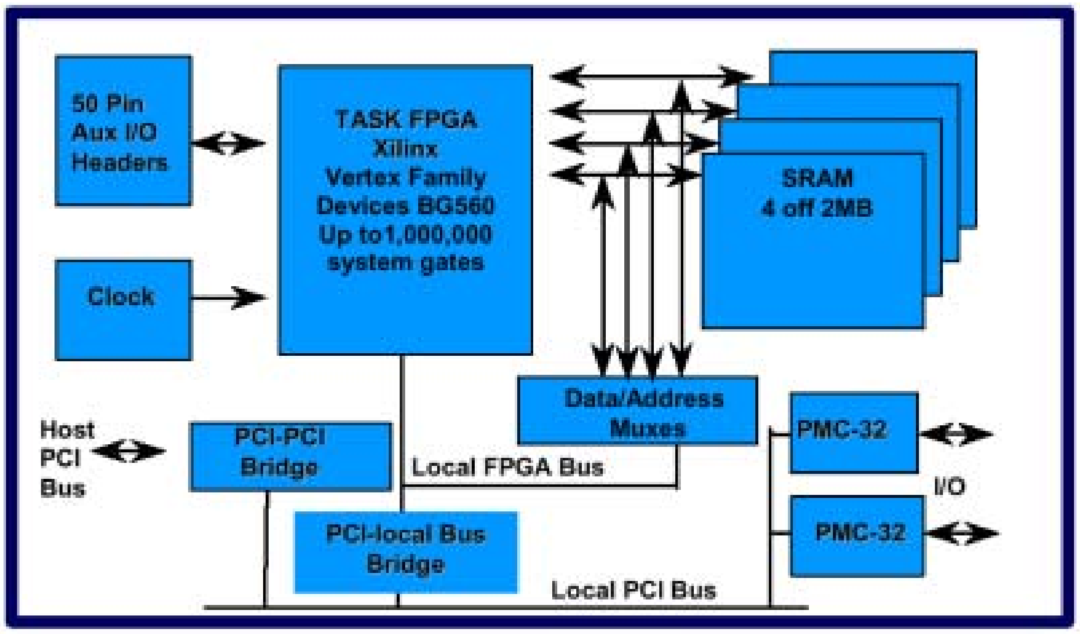


Figure B.1: Block Diagram of RC1000 Hardware.

APPENDIX C

AMI SEMICONDUCTOR 0.35μm TECHNOLOGY

C.1. Mixed A/D Technology

The 0.35 μm CMOS technology is a mixed Analog/Digital process. It is derived from the fully digital 0.35μ CMOS process and extended with analog capabilities

C.2. General Characteristics

- 0.35 μm, up to 5 metal layers
- Self-aligned twin tub N- and P Poly gates
- W-plug filling of stackable contacts and vias
- Nitride based passivation
- 2.0V to 3.6V Supply
- Protection :
    - Latchup resistance > +/- 200mA
    - ESD > +/- 2000V
- 6 Inch epi wafers

C.3. Layout Rules

- Drawn minimum gate length : 0.35μm for both PMOS and NMOS

- Polysilicon pitch : 0.9µm
- Metal 1 pitch : 1.1µm
- Metal 2 pitch : 1.4µm
- Metal 3 pitch : 1.4µm
- Metal 4 pitch : 1.4µm
- Metal 5 pitch : 2.8µm

C.4. Standard Cell Libraries

Following libraries are available for the AMI Semiconductor 0.35 µm CMOS technology :

AMI Semiconductor libraries supporting the ADS Asic Design Framework

- High Speed and Low Power Library (MTC 45000)
    - 393 core cells (gates, latches, flipflops,..)
    - 101 I/O cells (with slew rate controlled outputs and spike suppression)
    - ROM Density up to 240 Kbits/mm2
    - RAM Density (Static, single port): 25 Kbits/mm2
    - Gate density: 15000 NAND equiv. gates/mm2
    - Temp. range : -55 ... + 125deg.C
    - Typical gate delay(3,3V)
      - Unloaded invertor delay of 50ps
      - 2-input NAND delay of 610ps (typ) with fanout=2
    - Power : 0.5 µW/gate/MHz at 3 V

  - Additional analog modules

    - High ohmic polysilicon resistors (1kOhm/sq)

    - High value double poly capacitors (1.1 nF/mm2)

- Versatile I/O Library : PAD limited I/O cells (MTC45100)
- ROM and RAM compilation