ALEXANDRIA UNIVERSITY
FACULTY OF ENGINEERING

HARDWARE IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD ON FIELD PROGRAMMABLE GATE ARRAY

A Thesis

Presented to the Graduate School,
Faculty of Engineering, Alexandria University,
In Partial Fulfillment of the
Requirements for the Degree

Of

Master of Science

In

Electrical Engineering

By

Mohammed Morsy Naeem Farag
B.Sc. In Electrical Engineering (Communications and Electronics), June 2003

2006

HARDWARE IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD ON FIELD PROGRAMMABLE GATE ARRAY

Presented by

Eng. Mohammed Morsy Naeem Farag

For the Degree of

Master of Science

In

Electrical Engineering

Examiners' Committee: Approved

Prof. Dr. Hasan Elkamshoshy ……..……
Faculty of Engineering, Alexandria University

Prof. Dr. Magdy Fekry Ragaey ……..
Faculty of Engineering, Cairo University

Prof. Dr. Mohammed Rizk …….
Faculty of Engineering, Alexandria University

Vice Dean for Graduate Studies and Research,

Prof. Dr. Hossam Mohammed Fahmy Ghanem ……
Faculty of Engineering, Alexandria University

Advisors' Committee: Approved

Dr. Mohammed Rizk **...…..……**
Faculty of Engineering, Alexandria University

Dr. Hanan Hosny **……..**
Faculty of Engineering, Alexandria University

Dr. Haniah Farag **……..**
Faculty of Engineering, Alexandria University

ACKNOWLEDGEMENTS

First, I start my book with thanking **GOD** for helping me in completing this thesis and presenting it in the following form. Then, I would like to express my deepest gratitude to my parents for supporting me during every moment in my life. Certainly, I would like to express my sincere gratitude to *Prof. Dr. Mohamed Rizk* for his advice & concern that was invaluable to this work. Also, I wish to express my thanks to the staff of faculty of engineering, Alexandria University for their friendly cooperation. Finally, thanks to everyone who helped me to make this work come to reality.

Mohammed Morsy
November, 2006

ABSTRACT

The Advanced Encryption Standard (AES) is the new standard for cryptography and has gained wide support as means to secure digital data. In this thesis, we explored design and implementation approaches of the AES on field programmable gate arrays (FPGAs). We introduced the heuristic design techniques of the AES substitution boxes and we suggested an AES substitution box with good cryptographic properties. Tradeoffs of speed vs. area that are inherent in the design of a security processor are explored. Two implementations of the AES on Xilinx Virtex 4 FPGA are introduced, the first design is called optimized area AES which is based on the basic architecture of the AES, the second one is called optimized speed AES which is based on the sub-pipelined architecture of the AES. An AES crypto processor with serial interface was implemented and it could be used with any of our designed encryptor or decryptor. Two applications of the AES algorithm in feedback mode of operation were implemented on Xilinx Virtex 4 FPGA, one of the applications is the AES key wrap algorithm which could be used in the key transfer in unsecured communication channel, and the other application is the AES block cipher based deterministic random bit generator (DRBG) which could be used as a pseudo random number generator (PRNG). Loop unrolled architecture is used in the implementation of the AES in feedback mode of operation. A complete simulations and implementation results are provided for all of our designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
CAD	Computer Aided Design
CBC	Cipher Block Chaining
CFB	Cipher Feedback
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CTR	Counter
DES	Data Encryption Standard
DLL	Delay Looked Loop
DRBG	Deterministic Random Bit Generator
ECB	Electronic Codebook
EEPROM	Electrically Erasable Programmable Read Only Memory
FB	Feedback
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Gate Array
GF	Galois Field
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property
IV	Initial Value
KED	Key Encryption Data
KEK	Key Encryption Key
LC	Logic Cell
LUT	Look Up Table
NDRBG	Non- Deterministic Random Bit Generator
NFB	Non Feedback
NIST	National Institute of Standards and Technology
Nr	Number of Rounds
OFB	Output Feedback
PKC	Public Key Cryptography
PKI	Public Key Infrastructure
PRNG	Pseudo Random Number Generator
RAM	Random Access Memory
RNG	Random Number Generator
ROM	Read Only Memory

S-Box	Substitution Box
SKC	Secret Key Cryptography
SOP	Sum of Products
SRAM	Static Random Access Memory
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VHDL	Very High Speed Hardware Description Language

Chapter 1

INTRODUCTION

Since privacy issues and network security are emerging due to the wide proliferation of Internet, the research in protecting information is increasing. Cryptographic algorithms, also known as ciphers, form the fundamental aspect within this research field. The most used and analyzed cryptographic algorithm of the last 20 years, is the Data Encryption Standard (DES) [1]. With the introduction of this cipher in the early 70s, there were several accusations concerning hidden back-doors, not transparent S-boxes and the length of the key. Despite all the criticism, DES became the encryption standard in 1977. In 1983 it was shown [1] that the cipher is vulnerable due to its short key length. Considering the fact that the computing capacity is always increasing, the vulnerability of DES was a thorn in the eye. Therefore, an enhanced version of the cipher was introduced. This enhanced version known as Triple-DES [1] performs DES three times sequentially and therefore it is more secure than DES. However, the speed performance of Triple-DES on software based platforms was not interesting for practical applications. Therefore in 1997, the National Institute of Standards and Technology (NIST) organized a contest in order to develop a new cryptographic algorithm standard which would replace both DES and Triple-DES. More precisely, the main objective was to develop an algorithm that would at least offer the same security level which was provided by Triple-DES, but that should have higher performance than the performance of Triple-DES. Fifteen new block cryptographic algorithms were submitted [2]. On November 26, 2001, the algorithm known as Rijndael (pronounced Rhine-dall) was chosen to be the replacement for DES and since then it is known as the Advanced Encryption Standard (AES). This algorithm satisfies the following National Institute of Standard and Technology (NIST) statement :"Assuming that one could build a machine that could recover a DES key in a second, then it would take that machine approximately 149 thousand-billion (149 trillion) years to crack a 128-bit AES key. To put that into perspective, the universe is believed to be less than 20 billion years old." The development of high speed networks, has directed the research framework of protecting information, to a broader aspect then that of solely developing ciphers. Cipher performance, key management, policies and reliability aspects are important topics nowadays. These days, a lot of network security services and systems are implemented, such as Public Key Infrastructure (PKI) systems, web appliances, high-speed routers and Firewalls, that are used for securing information. The communication in these systems is based on various protocols, for example the Secure Sockets Layer (SSL) protocol, the IP Security Protocol (IPSec) and the Transport Layer Security (TLS) protocol. Such protocols are not limited to one or two cryptographic algorithms, but they often use a combination of various cryptographic algorithms. The choice for a certain cipher within a communication process depends on several factors such as company policies on encryption strength and government restrictions on encryption export. Considering these facts and the fact that cryptographic algorithms are relative frequently upgraded, cryptographic flexibility and high speed performance are requirements in network systems.

The choice for a certain platform (e.g. software, ASICs or FPGAs) for implementing cryptographic applications is driven by several design aspects such as performance, costs, power and flexibility. The performance, costs and power aspects are expressed by well known metrics. However, these metrics do not completely characterize the designs that are implemented in reconfigurable hardware or software. For these designs, flexibility in redesign or hardware reconfiguration is also an important design issue. Flexibility is defined by IEEE as: "the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed" [4].

Since DES, and therefore also Triple-DES [3], was primarily designed for hardware based platforms, both cryptographic algorithms were often implemented in Application Specific Integrated Circuits (ASIC). These systems showed adequate speed performance.

Also the AES implemented in ASIC results in high speed performance [5]. Furthermore, since ASICS are often produced in large quantities, they have favorable costs. However, since ASICs are completely hard-wired they lack flexibility. Their redesign is a complex and expensive process. Every change of the IC design leads to new an IC process mask, which is one of the major cost factors. Moreover, a prediction for future semiconductor technologies is that the cost of the mask will grow exponentially and will soon dominate the total cost of the production process. In short, although ASICs based solutions show adequate speed performance they are not suitable for demanding cryptographic network systems.

In contrast to ASIC technology, a Field-Programmable Gate Array (FPGA) is fully reprogrammable. FPGAs provide reconfigurable hardware, flexible interconnect, and field-programmable ability without introducing extra costs. Therefore, substantial amount of work has been reported on various cryptographic algorithms implemented on FPGAs.

While FPGAs used to be selected for lower speed/complexity/volume designs in the past, today’s FPGAs easily push the 500 MHz performance barrier. With unprecedented logic density increases and a host of other features, such as embedded processors, DSP blocks, clocking, and high-speed serial at ever lower price points, FPGAs are a compelling proposition for almost any type of design. Although these solutions show adequate speed performance and provide flexibility, they are not interesting for mass productions, since the cost of FPGA devices are still a bottleneck. FPGA devices are expensive compared to ASIC and software based solutions, but at the same time the FPGA devices have other advantages such as simple design cycle, faster time to test and market and field reprogram ability. These advantages besides the high speed of recent FPGAs devices make the FPGAs are the best solutions for the research purposes.

Software based solutions, e.g. targeting applications on general-purpose microprocessors, digital signal processors or microcontrollers are fully reprogrammable. Beside the flexibility aspect, the cost aspect of such solutions is most favorable. However, the disadvantage of these solutions is that the speed performance is significantly lower than that based on ASICs and FPGAs. Therefore, even software based solutions are not suitable for some demanding cryptographic network systems.

When selecting the AES algorithm, both efficient software and hardware implementations were taken into consideration. This thesis addresses efficient hardware implementation approaches for the AES algorithm and introduces two implementation approaches (Optimized Area and Optimized speed) for the AES algorithm on FPGA (field programmable gate array).

The organization of this thesis will be as follows:

Chapter 2 provides an introduction to cryptography and its types (with focus on private key cryptography and its modes of operation) and provides a detailed explanation for the AES algorithm with a brief mathematical background for the algorithm.

Chapter 3 discusses the cryptographic properties of good substitution boxes (S-boxes) and the heuristic optimization techniques used to improve this properties for a single Boolean function and S-box. Also this chapter introduces a suggested S-box which could be used in the AES algorithm.

Chapter 4 addresses various approaches for efficient hardware implementation of the AES from the architectural and algorithmic point of view (area and time are the optimization goal in this chapter). Also this chapter discusses Resources sharing issues between encryptor and decryptor.

Chapter 5 addresses various implementation methods of SubBytes transformation which is considered the most area and time consuming transformation in the AES algorithm. In this chapter we will provide a detailed explanation for the implementation method which we will use in our design for the algorithm with a comparison between different used methods.

Chapter 6 introduces our hardware implementation and simulation results of the optimized area AES (AES encryptor and decryptor with minimum area resources) and optimized speed AES (AES encryptor and decryptor with minimum delay) using various design methods addressed in Chapter 4 and Chapter 5 with a comparison between the two implemented hardware and other previous implemented circuits. Finally we will introduce our implementation and simulation results for AES crypto processor with serial interface which could be used to make a practical test for any of our designed encryptors or decryptors.

Chapter 7 addresses two applications of the AES with there hardware implementation on FPGA . The first application is the AES key wrap/ unwrap algorithm. The second application is the deterministic random number generator (DRBG) based on the AES in the counter (CTR) and output feedback (OFB) modes of operation.

Chapter 2

ADVANCED ENCRYPTION STANDARD

2.1. Brief Introduction to Cryptography

Cryptography is the science of writing in secret code and is an ancient art; the first documented use of cryptography in writing dates back to circa 1900 B.C. when an Egyptian scribe used non-standard hieroglyphs in an inscription. Some experts argue that cryptography appeared spontaneously sometime after writing was invented, with applications ranging from diplomatic missives to war-time battle plans. It is no surprise, then, that new forms of cryptography came soon after the widespread development of computer communications. In data and telecommunications, cryptography is necessary when communicating over any un-trusted medium, which includes just about any network, particularly the Internet.

- Within the context of any application-to-application communication, there are some specific security requirements, including:
- Authentication: The process of proving one's identity. (The primary forms of host-to-host authentication on the Internet today are name-based or address-based, both of which are notoriously weak.).
- Privacy/confidentiality: Ensuring that no one can read the message except the intended receiver.
- Integrity: Assuring the receiver that the received message has not been altered in any way from the original.
- Non-repudiation: A mechanism to prove that the sender really sent this message.

Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication. There are, in general, three types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash functions, each of which is described below. In all cases, the initial unencrypted data is referred to as plaintext. It is encrypted into ciphertext, which will in turn (usually) be decrypted into usable plaintext.

2.2. Types of Cryptographic Algorithms

There are several ways of classifying cryptographic algorithms [6]. They will be categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use. The three types of algorithms that will be discussed are shown in Figure 2–1:

- Secret Key Cryptography (SKC): Uses a single key for both encryption and decryption.
- Public Key Cryptography (PKC): Uses one key for encryption and another for decryption.

- Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information.

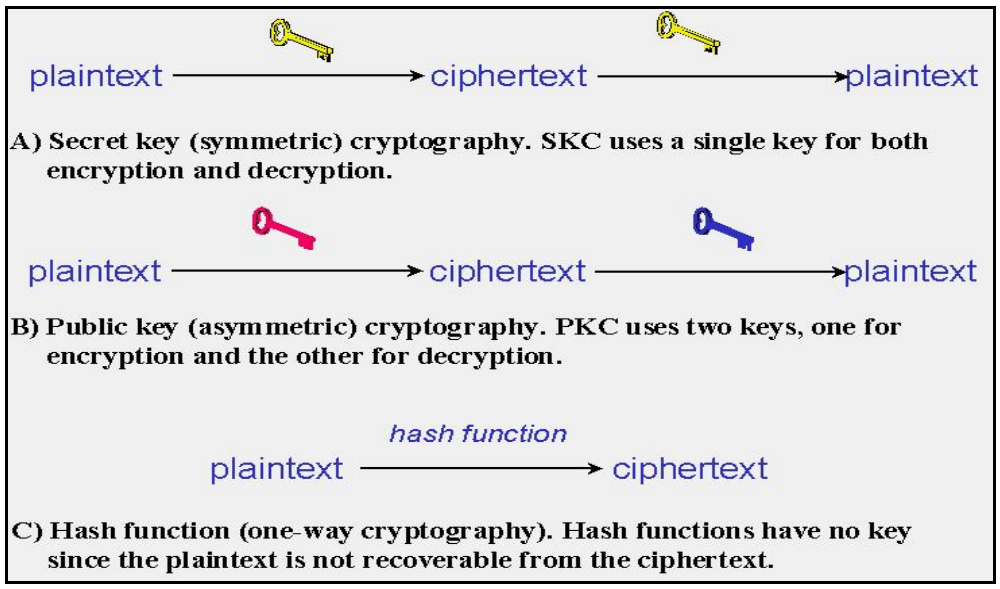


Figure 2–1: Three types of cryptography (Secret Key, Public Key and Hash Function)

2.2.1. Secret Key Cryptography

With secret key cryptography, a single key is used for both encryption and decryption. As shown in Figure 2–1-A, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or rule set) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called symmetric encryption.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver; that, in fact, is the secret. The biggest difficulty with this approach, of course, is the distribution of the key.

Secret key cryptography schemes are generally categorized as being either stream ciphers or block ciphers. Stream ciphers operate on a single bit (byte or computer word) at a time and implement some form of feedback mechanism so that the key is constantly changing. A block cipher is so-called because the scheme encrypts one block of data at a time using the same key on each block. In general, the same plaintext block will always encrypt to the same ciphertext when using the same key in a block cipher whereas the same plaintext will encrypt to different ciphertext in a stream cipher.

Stream ciphers come in several flavors but two are worth mentioning here. Self-synchronizing stream ciphers calculate each bit in the keystream as a function of the previous n bits in the keystream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the n-bit keystream it is. One problem is error propagation; a garbled bit in transmission will result in n garbled bits at the receiving side. Synchronous stream ciphers generate the keystream in a fashion independent of the message stream but by using the

same keystream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the keystream will eventually repeat.

Block ciphers can operate in one of several modes; the following five are the most important [7], [8]:

- **Electronic Codebook (ECB) mode** is the simplest, most obvious application: the secret key is used to encrypt the plaintext block to form a ciphertext block. Two identical plaintext blocks, then, will always generate the same ciphertext block as shown in Figure 2–2. Although this is the most common mode of block ciphers, it is susceptible to a variety of brute-force attacks.

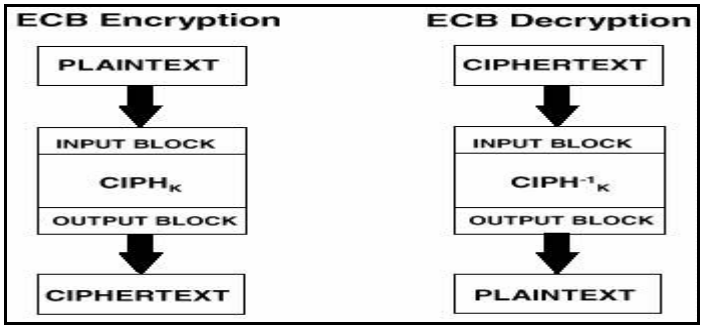


Figure 2–2: Electronic Code Book (ECB) Mode

- **Cipher Block Chaining (CBC) mode** adds a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption as shown in Figure 2–3. In this mode, two identical blocks of plaintext never encrypt to the same ciphertext.

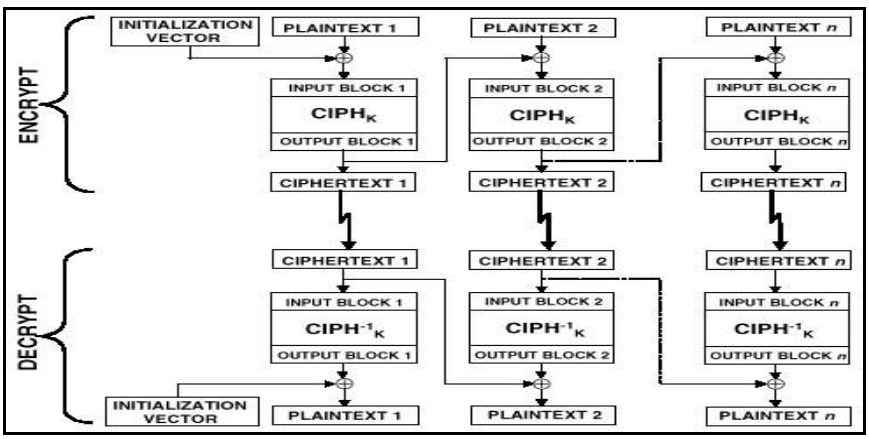


Figure 2–3: Cipher Bloch Chaining (CBC) Mode

- **Cipher Feedback (CFB) mode** is a block cipher implementation as a self-synchronizing stream cipher as shown in Figure 2–4. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting interactive terminal input. If we were using 1-byte CFB mode, for example, each incoming character is placed into a shift register the same size as the block, encrypted, and the block transmitted. At the

receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded.

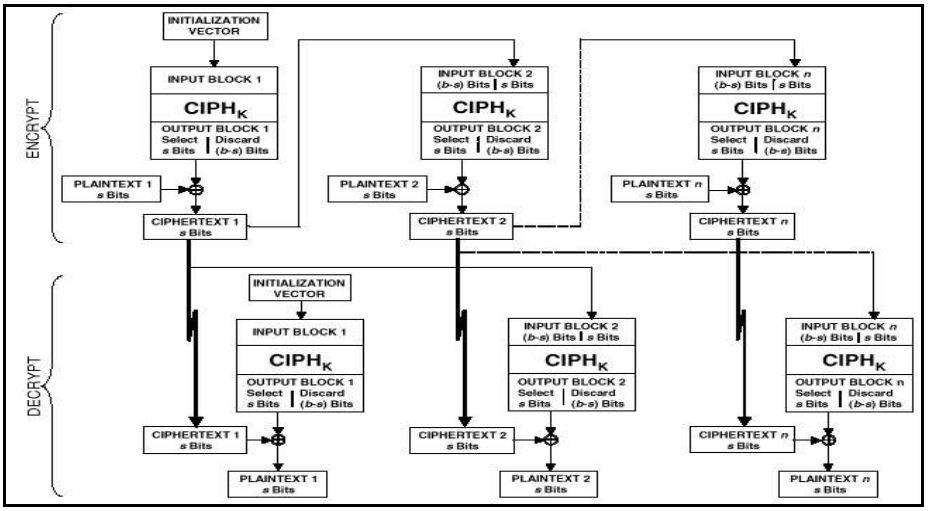


Figure 2–4: Cipher Feedback (CFB) Mode

- **Output Feedback (OFB) mode** is a block cipher implementation conceptually similar to a synchronous stream cipher as shown in Figure 2–5. OFB prevents the same plaintext block from generating the same ciphertext block by using an internal feedback mechanism that is independent of both the plaintext and ciphertext bitstreams.

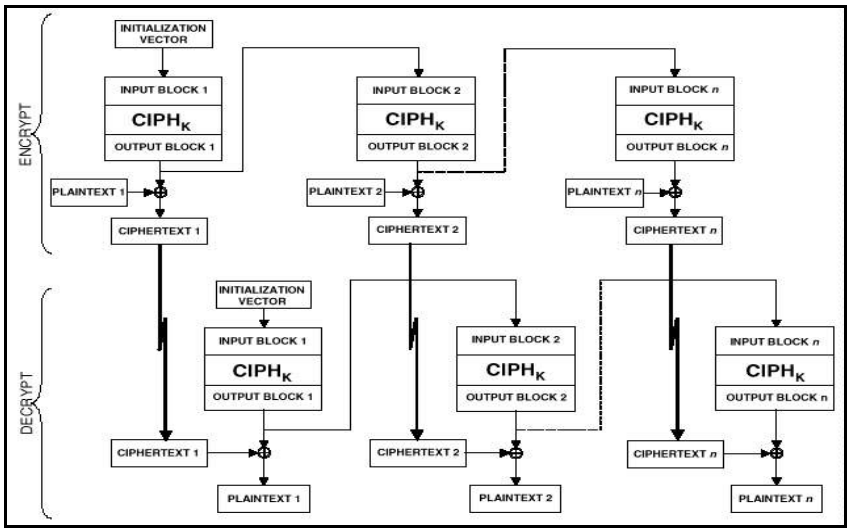


Figure 2–5: Output Feedback (OFB) Mode

- **The Counter Mode** is a confidentiality mode that features the application of the forward cipher to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa as shown in Figure 2–6. The sequence of counters must have the property that each block in the sequence is different from every other block. This condition is not restricted to a single message: across all of the

messages that are encrypted under the given key, all of the counters must be distinct.

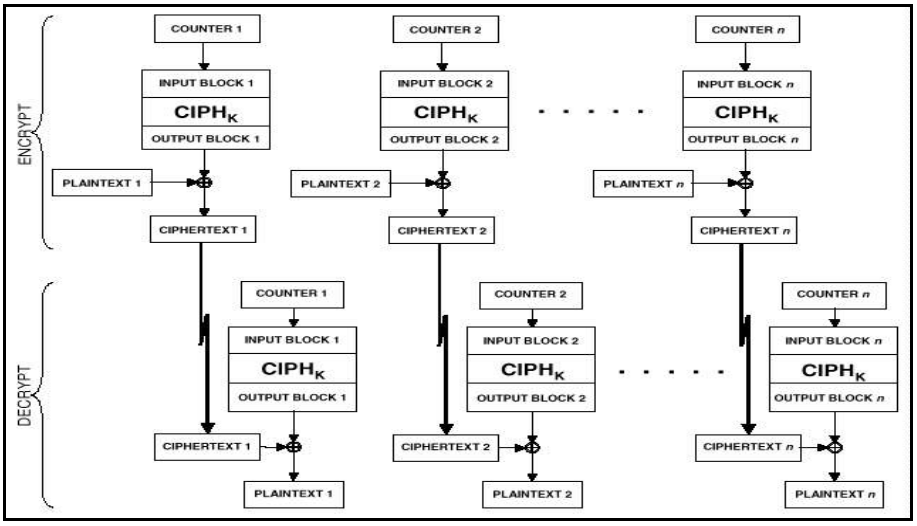


Figure 2–6: Counter (CTR) Mode

2.2.2. Public-Key Cryptography

Public-key cryptography has been said to be the most significant new development in cryptography in the last 300-400 years. Modern PKC was first described publicly by Stanford University professor Martin Hellman and graduate student Whitfield Diffie in 1976. Their paper described a two-key crypto system in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key.

PKC depends upon the existence of so-called one-way functions, or mathematical functions that are easy to computer whereas their inverse function is relatively difficult to compute. Let me give you two simple examples:

1. Multiplication vs. factorization: Suppose I tell you that I have two numbers, 9 and 16, and that I want to calculate the product; it should take almost no time to calculate the product, 144. Suppose instead that I tell you that I have a number, 144, and I need you tell me which pair of integers I multiplied together to obtain that number. You will eventually come up with the solution but whereas calculating the product took milliseconds, factoring will take longer because you first need to find the 8 pair of integer factors and then determine which one is the correct pair.
2. Exponentiation vs. logarithms: Suppose I tell you that I want to take the number 3 to the 6th power; again, it is easy to calculate 36=729. But if I tell you that I have the number 729 and want you to tell me the two integers that I used, x and y so that $\log_x 729 = y$, it will take you longer to find all possible solutions and select the pair that I used.

While the examples above are trivial, they do represent two of the functional pairs that are used with PKC; namely, the ease of multiplication and exponentiation versus the

relative difficulty of factoring and calculating logarithms, respectively. The mathematical "trick" in PKC is to find a trap door in the one-way function so that the inverse calculation becomes easy given knowledge of some item of information.

Generic PKC employs two keys that are mathematically related although knowledge of one key does not allow someone to easily determine the other key. One key is used to encrypt the plaintext and the other key is used to decrypt the ciphertext. The important point here is that it does not matter which key is applied first, but that both keys are required for the process to work (Figure 2–1-B). Because pair of keys is required, this approach is also called asymmetric cryptography.

In PKC, one of the keys is designated the public key and may be advertised as widely as the owner wants. The other key is designated the private key and is never revealed to another party.

2.2.3. Hash Functions

Hash functions (also called message digests and one-way encryption) are algorithms that in some sense use no key (Figure 2–1-C). Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered. Hash algorithms are typically used to provide a digital fingerprint of a file contents, often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Hash functions, then, provide a measure of the integrity of a file.

Figure 2–7 shows types of cryptography and some used algorithms in each type.

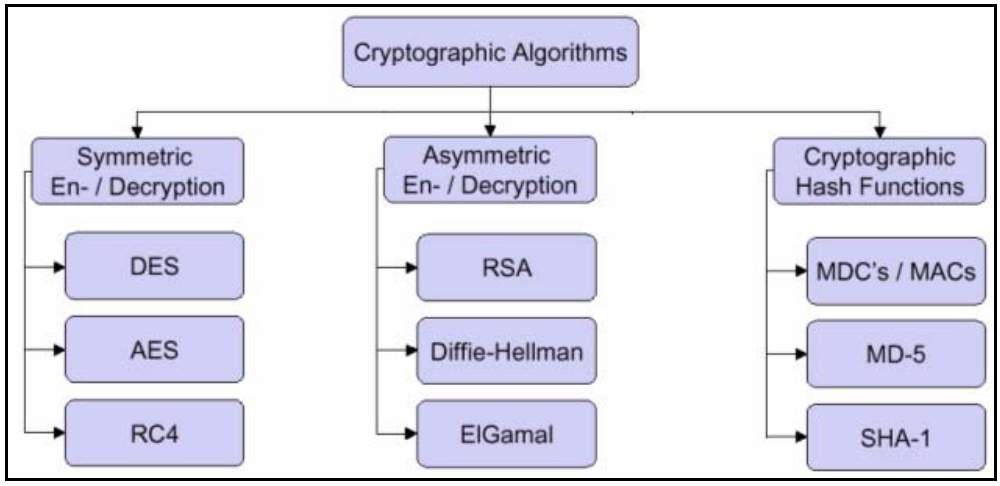


Figure 2–7: Types of cryptography and its examples

2.3. History of AES Algorithm

On January 2, 1997, the National Institute of Standards and Technology (NIST) invited proposals for new algorithms for the Advanced Encryption Standard (AES) to replace the old Data Encryption Standard (DES). Among the 15 preliminary candidates, MARS,

RC6, Rijndael [7], Serpent, and Twofish were announced as the finalist candidates on August 9, 1999 for further evaluation. After studying all available information and public comments on these finalist candidates, NIST announced in October 2000 that Rijndael was selected as the AES algorithm.

AES is a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. Rijndael was designed to handle additional block sizes and key lengths; however they are not adopted in this standard [9], [10].

Throughout the remainder of this standard, the algorithm specified herein will be referred to as "the AES algorithm." The algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".

2.4. Mathematical Background for the AES

The Rijndael Algorithm is based on mathematical concept of finite fields. Knowledge of finite fields and related terms will help in understanding structure of Rijndael and the motivation behind some of the optimizations. This section explains the mathematical concepts it presents the mathematical preliminaries in field theory and linear algebra [1]

1. *Groups* An Abelian group $<G, +>$ is defined as a set G and an operation + defined on the elements of G given by the following relationship:

$$+ : G \times G \rightarrow G : (a,b) \rightarrow a+b$$

In addition, the operation + must satisfy the following conditions:

1. Closed : $\forall a,b \; in \; G, a+b \; is \; also \; in \; G$
2. Associative : $\forall a,b,c \; in \; G, (a+b)+c = a+(b+c)$
3. Commutative : $\forall a,b \; in \; G, a+b = b+a$
4. Neutral element : $a+0 = a \; where \; a,0 \; are \; in \; G$
5. Inverse elements : $\forall a \; in \; G, \exists b \; in \; G \; such \; that \; a+b = 0$

2. *Ring* A ring $<R, +, \bullet>$ is defined as a set R and two operations + and $\bullet$ defined on the elements of R and which fulfill the following conditions:

1. $<R, +, \bullet>$ is an Abelian group
2. Closed : $\forall a,b \; in \; G, a+b \; is \; also \; in \; G$
3. Associative : $\forall a,b,c \; in \; G, (a+b)+c = a+(b+c)$
4. Distributive : $\forall a,b,c \; in \; G, \; (a+b) \bullet c = (a \bullet c)+(b \bullet c)$
5. Neutral element : $a \bullet 1 = a \; where \; a,1 \; belong \; to \; R$

If the operation $\bullet$ is commutative, $<R, +, \bullet>$ is called a commutative ring.

3. *Field* A field <*F, +, •*> is defined as the structure that fulfills the following conditions:

    1. <*F, +, •*> is a commutative ring.
    2. <*F, +*> and <*F0, •*> are Abelian groups
    3. distributive : $\forall a,b,c \; in \; F, (a+b) \bullet c = (a \bullet c) + (b \bullet c)$
    4. Neutral element : $a + 0 = a \; where \; a, 0 \; are \; in \; F$
    5. Neutral element : $\forall a \; in \; F, a \bullet b = 0$

    The number of elements in the field is called its order.

4. *Finite Fields* A field that has a finite order is called a finite field. A field of order m exists on iff m is a prime power, i.e. $m = p^n$ where p is a prime and n is an integer. Here, p is called the characteristic of the finite field.
5. *Galois Field* A finite field with p^n elements is called a Galois Field $GF(p^n)$. Galois Fields are named after the French mathematician Evariste Galois who did some early work on fields. Rijndael uses the Galois Field $GF(2^8)$.
6. *Polynomials over a field* A polynomial over a field is expressed as

$$b(x) = b_{n-1}x^{n-1} + b_{n-1}x^{n-1} + \cdots + b_1 x + b_0 \qquad (2.1)$$

    Here, x is called the indeterminate of the polynomial and $b_{n-1}, b_{n-2}, \cdots, b_1, b_0$ are called coefficients. The degree of the polynomial is the highest value of b_i which is non-zero.

All bytes in the AES algorithm are interpreted as finite field elements using the notation introduced in the above equation. Finite field elements can be added and multiplied, but these operations are different from those used for numbers. The following subsections introduce the basic mathematical concepts used in AES algorithm.

2.4.1. Polynomial Addition

The addition of two elements in a finite field is achieved by "adding" the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the XOR operation (denoted by $\oplus$) - i.e., modulo 2 - so that $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, *and* $0 \oplus 0 = 0$. Consequently, subtraction of polynomials is identical to addition of polynomials.

Alternatively, addition of finite field elements can be described as the modulo 2 addition of corresponding bits in the byte. For two bytes $\{a_7a_6a_5a_4a_3a_2a_1a_0\}$ and $\{b_7b_6b_5b_4b_3b_2b_1b_0\}$, the sum is $\{c_7c_6c_5c_4c_3c_2c_1c_0\}$, where each $c_i = a_i \oplus b_i$ (i.e., $c_7 = a_7 \oplus b_7$, $c_6 = a_6 \oplus b_6 \ldots$, $c_0 = a_0 \oplus b_0$).

2.4.2. Polynomial Multiplication

In the polynomial representation, multiplication in $GF(2^8)$ (denoted by $\bullet$) corresponds with the multiplication of polynomials modulo an irreducible polynomial of degree *8*. A polynomial is irreducible if its only divisors are one and itself. For the AES algorithm, this irreducible polynomial is

$$m(x) = x^8 + x^4 + x^3 + x + 1 \qquad (2.2)$$

Or *{01}{1b}* in hexadecimal notation.

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication.

The multiplication defined above is associative, and the element *{01}* is the multiplicative identity. For any non-zero binary polynomial $b(x)$ of degree less than *8*, the multiplicative inverse of $b(x)$, denoted $b^{-1}(x)$, can be found as follows: the extended Euclidean algorithm [5] is used to compute polynomials $a(x)$ and $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1 \qquad (2.3)$$

Hence, $a(x) \bullet b(x) \bmod m(x) = 1$, which means

$$b^{-1}(x) = a(x) \bmod m(x) \qquad (2.4)$$

Moreover, for any *a(*x*), b(*x*) and* c*(*x*)* in the field, it holds that

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

It follows that the set of *256* possible byte values, with XOR used as addition and the multiplication defined as above, has the structure of the finite field $GF(2^8)$.

2.4.3. Multiplication by *x*

Multiplying the binary polynomial defined in equation (2.1) with the polynomial x results in

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \qquad (2.5)$$

The result $x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$, as defined in equation (2.1).If $b_7 = 0$, the result is already in reduced form. If $b_7 = 1$, the reduction is accomplished by subtracting (i.e., XORing) the polynomial $m(x)$. It follows that multiplication by *x (i.e., {00000010} or {02})* can be implemented at the byte level as a left shift and a subsequent conditional bitwise XOR with *{1b}*.

This operation on bytes is denoted by `xtime ()`. Multiplication by higher powers of *x* can be implemented by repeated application of `xtime ()`.

By adding intermediate results, multiplication by any constant can be implemented.

2.4.4. Polynomials with Coefficients in $GF(2^8)$

Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \qquad (2.6)$$

This will be denoted as a word in the form $[a_0, a_1, a_2, a_3]$. Note that the polynomials in this section behave somewhat differently than the polynomials used in the definition of finite field elements, even though both types of polynomials use the same indeterminate, x. The coefficients in this section are themselves finite field elements, i.e., bytes, instead of bits; also, the multiplication of four-term polynomials uses a different reduction polynomial, defined below. The distinction should always be clear from the context.

To illustrate the addition and multiplication operations, let

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \qquad (2.7)$$

Define a second four-term polynomial. Addition is performed by adding the finite field coefficients of like powers of *x*. This addition corresponds to an XOR operation between the corresponding bytes in each of the words – in other words, the XOR of the complete word values.

Thus, using the equations of (2.6) and (2.7),

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0) \qquad (2.8)$$

Multiplication is achieved in two steps. In the first step, the polynomial product $c(x) = a(x) \bullet b(x)$ is algebraically expanded, and like powers are collected to give

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \qquad (2.9)$$

Where

$$\begin{aligned}
c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\
c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\
c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_6 &= a_3 \bullet b_3 \\
c_3 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3
\end{aligned} \qquad (2.10)$$

The result, *c(x)*, does not represent a four-byte word. Therefore, the second step of the multiplication is to reduce *c(x)* modulo a polynomial of degree 4; the result can be reduced to a polynomial of degree less than *4*. For the AES algorithm, this is accomplished with the polynomial $x^4 + 1$, so that

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4} \tag{2.11}$$

The modular product of $a(x)$ and $b(x)$, denoted by $a(x) \otimes b(x)$, is given by the four-term polynomial $d(x)$, defined as follows:

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \tag{2.12}$$

With

$$\begin{aligned} d_0 &= (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3) \\ d_1 &= (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3) \\ d_2 &= (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3) \\ d_3 &= (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3) \end{aligned} \tag{2.13}$$

When $a(x)$ is a fixed polynomial, the operation defined in equation (2.12) can be written in matrix form as:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.14}$$

Because x^4+1 is not an irreducible polynomial over $GF(2^8)$, multiplication by a fixed four-term polynomial is not necessarily invertible. However, the AES algorithm specifies a fixed four-term polynomial that *does* have an inverse (which is required for the decryption process):

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{2.15}$$

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \tag{2.16}$$

2.5. The AES Cipher/ Decipher Algorithm

The Rijndael proposal for AES defined a cipher in which the block length and the key length can be independently specified to be 128, 192, or 256 bits. The AES specification uses the same three key size alternatives but limits the block length to 128 bits. A number of AES parameters depend on the key length (Table 2–1). In the description of this section, we assume a key length of 128 bits, which is likely to be the one most commonly implemented [7], [10].

Table 2–1: AES Parameters

Key size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

Rijndael was designed to have the following characteristics:

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

Figure 2–8 shows the overall structure of AES. The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a square matrix of bytes. This block is copied into the State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix. These operations are depicted in Figure 2–8-a. Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is four bytes and the total key schedule is 44 words for the 128-bit key (Figure 2–8-b). Note that the ordering of bytes within a matrix is by column. So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the in matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the w matrix.

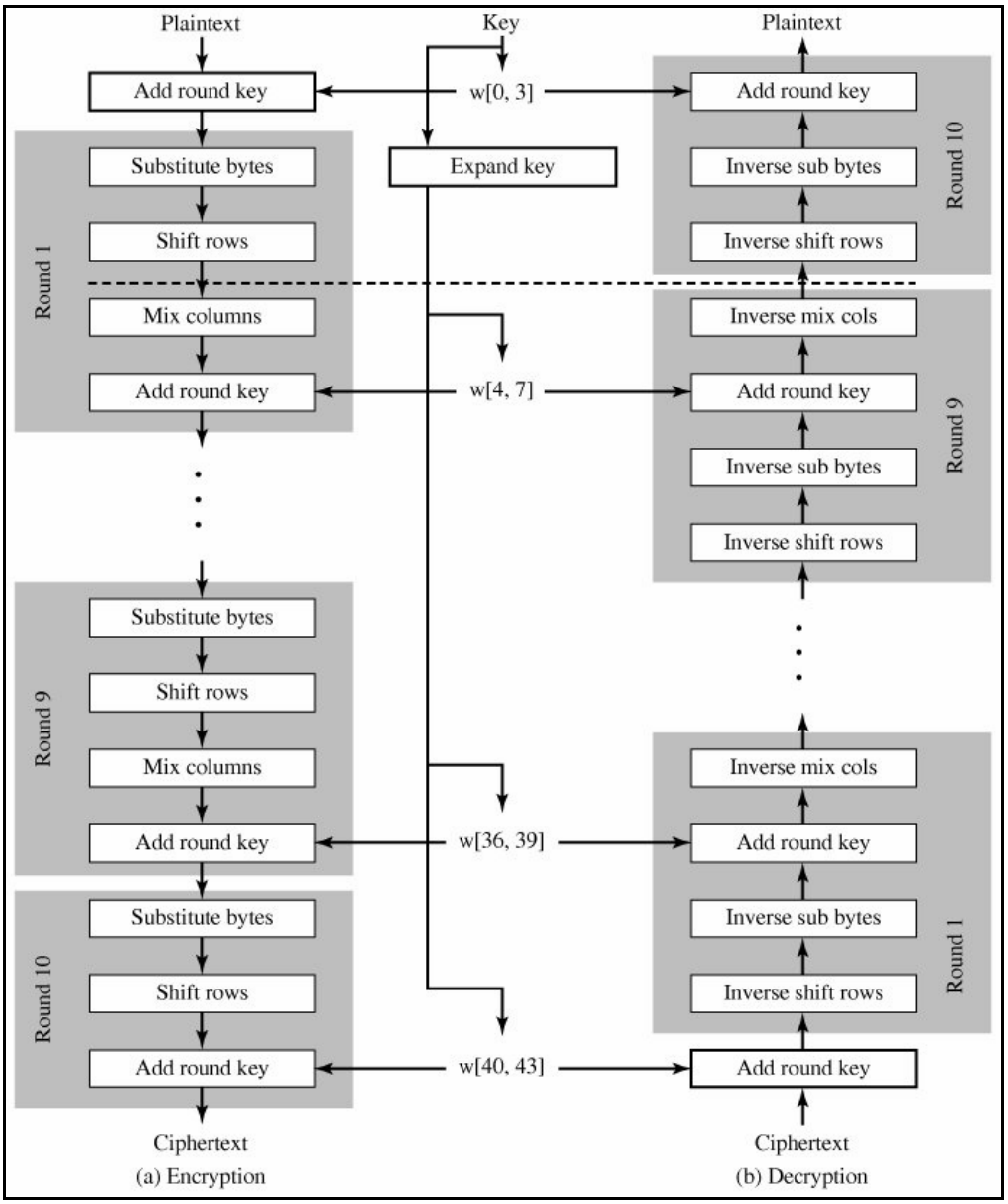


Figure 2–8: AES Encryption and Decryption

Before delving into details, we can make several comments about the overall AES structure:

1. One noteworthy feature of this structure is that it is not a Feistel structure. In the classic Feistel structure, half of the data block is used to modify the other half of the data block, and then the halves are swapped. Two of the AES finalists, including Rijndael, do not use a Feistel structure but process the entire data block in parallel during each round using substitutions and permutation.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, *w[i]*. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 2–8.
3. Four different stages are used, one of permutation and three of substitution:

- SubBytes(): Uses an S-box to perform a byte-by-byte substitution of the block
- ShiftRows(): A simple permutation
- MixColumns(): A substitution that makes use of arithmetic over GF(2^8)
- AddRoundKey( ): A simple bitwise XOR of the current block with a portion of the expanded key

4. The structure is quite simple. For both encryption and decryption, the cipher begins with an AddRoundKey stage, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages. Figure 2–9 depicts the structure of a full encryption round.

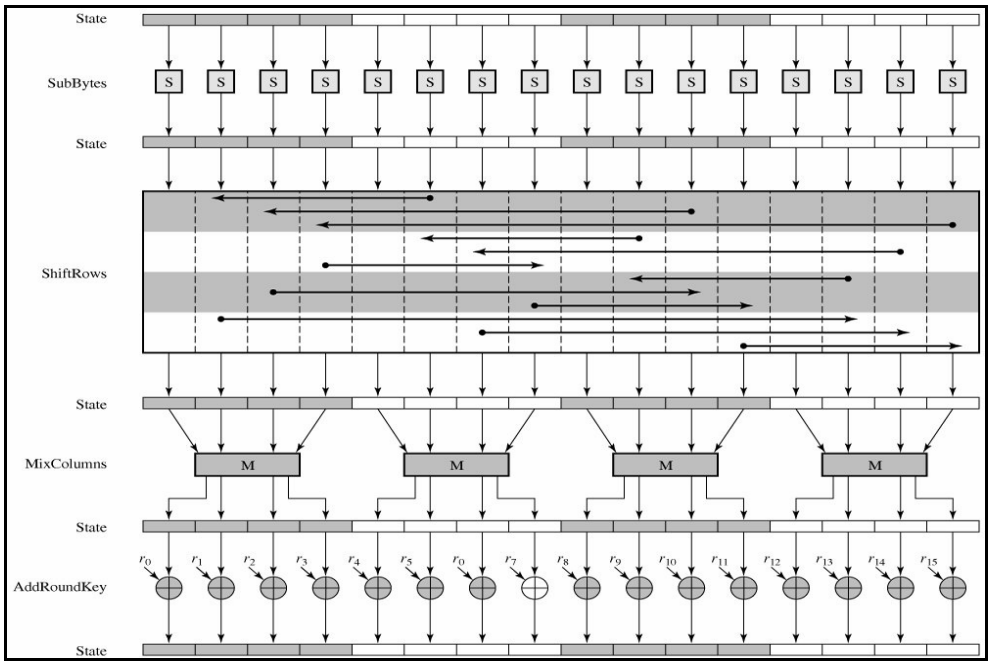


Figure 2–9: AES Encryption Round

5. Only the AddRoundKey stage makes use of the key. For this reason, the cipher begins and ends with an AddRoundKey stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.
6. The AddRoundKey stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. We can view the cipher as alternating operations of XOR encryption (AddRoundKey) of a block, followed by scrambling of the block (the other three stages), and followed by XOR encryption, and so on. This scheme is both efficient and highly secure.
7. Each stage is easily reversible. For the Substitute Byte, ShiftRows, and MixColumns stages, an inverse function is used in the decryption algorithm. For the AddRoundKey stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus A \oplus B = B$.

8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.
9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 2–9 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), State is the same for both encryption and decryption.
10. The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

We now turn to a discussion of each of the four stages used in AES. For each stage, we describe the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rationale for the stage. This is followed by a discussion of key expansion. As was mentioned in Sec. 2.4, AES uses arithmetic in the finite field $GF(2^8)$, with the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

2.5.1. SubBytes Transformation (Forward and Inverse Transformations)

The forward substitute byte transformation, called SubBytes(), is a simple table lookup (Figure 2–10-a). AES defines a 16 x 16 matrix of byte values, called an S-box (Table 2–2-a), that contains a permutation of all possible 256 8-bit values. Each individual byte of State is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value.

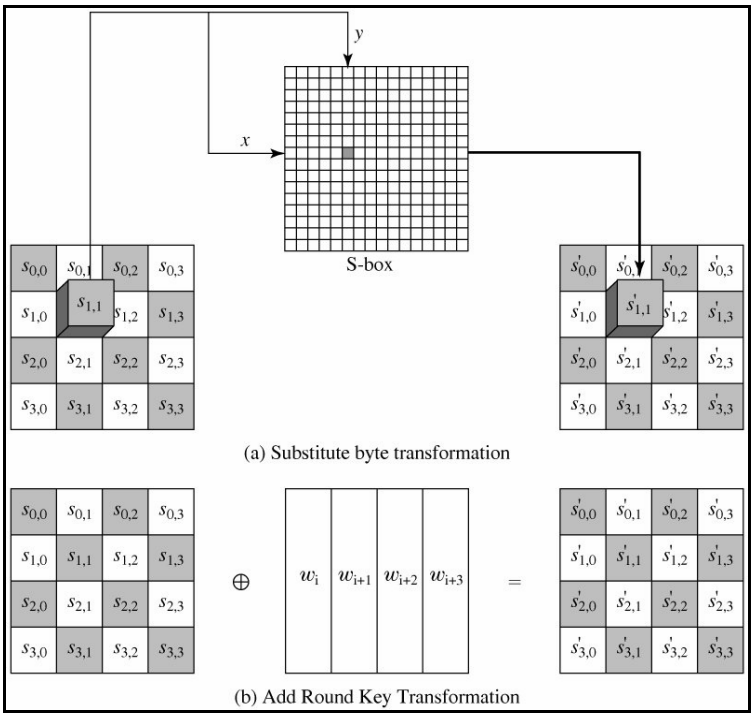


Figure 2–10: AES Byte level operations

Table 2–2: AES S-Boxes

(a) S-box

x \ y	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

(b) Inverse S-box

x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

The S-box is constructed in the following fashion:

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains {00}, {01}, {02} …, {0F}; the second row contains {10}, {11}, etc.; and so on. Thus, the value of the byte at row x, column y is {xy}.
2. Map each byte in the S-box to its multiplicative inverse in the finite field $GF(2^8)$; the value {00} is mapped to itself.
3. Consider that each byte in the S-box consists of 8 bits labeled $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$. Apply the following transformation to each bit of each byte in the S-box:

$$b_i' = b_i + b_{(i+4)\bmod 8} + b_{(i+5)\bmod 8} + b_{(i+6)\bmod 8} + b_{(i+7)\bmod 8} + c_i \qquad (2.17)$$

Where c_i is the i^{th} bit of byte c with the value $\{63\}$; that is, $(c_7c_6c_5c_4c_3c_2c_1c_0) = (01100011)$. The prime (') indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad (2.18)$$

Equation (2.18) has to be interpreted carefully. In ordinary matrix multiplication, each element in the product matrix is the sum of products of the elements or one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Further, the final addition shown in Equation (2.18) is a bitwise XOR.

The inverse substitute byte transformation, called InvSubBytes(), makes use of the inverse S-box shown in Table 2–2-b. Note, for example, that the input {2A} produces the output {95} and the input {95} to the S-box produces {2A}. The inverse S-box is constructed by applying the inverse of the transformation in equation (2.17) followed by taking the multiplicative inverse in $GF(2^8)$. The inverse transformation is:

$$b_i' = b_i + b_{(i+2) \bmod 8} + b_{(i+5) \bmod 8} + b_{(i+7) \bmod 8} + d_i \qquad (2.19)$$

Where byte $d = \{05\}$, or 00000101. We can depict this transformation as follows:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad (2.20)$$

2.5.2. ShiftRows Transformation (Forward and Inverse Transformations)

The forward shift row transformation, called ShiftRows(), is depicted in Figure 2–11-a. The first row of State is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed.

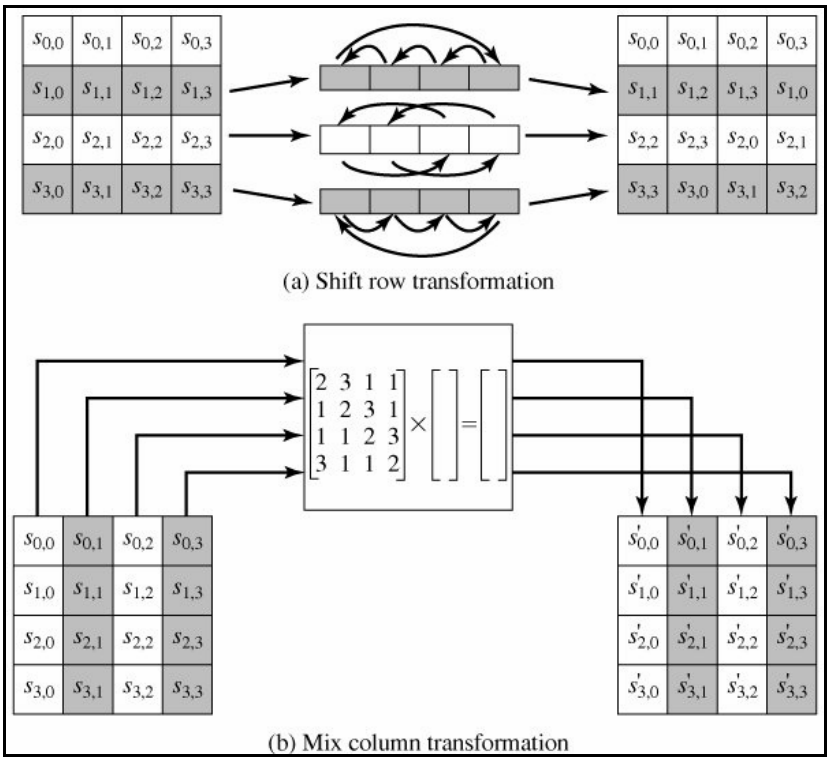


Figure 2–11: AES Row and Column Properties

The inverse shift row transformation, called InvShiftRows(), performs the circular shifts in the opposite direction for each of the last three rows, with a one-byte circular right shift for the second row, and so on.

2.5.3. MixColumns Transformation (Forward and Inverse Transformations)

The forward mix column transformation, called MixColumns() transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 2.4.4. The columns are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x)$, given by equation (2.15)

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

As described in Sec. 2.4.4, this can be written as a matrix multiplication. Let:

$$s'(x) = a(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 02 & 02 & 01 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad for \ \ 0 \le c < Nb \qquad (2.21)$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}) \end{aligned} \qquad (2.22)$$

The inverse mix column transformation, called InvMixColumns() is the inverse of the `MixColumns()` transformation. InvMixColumns() operates on the State column-by-column, treating each column as a four- term polynomial as described in Sec. 2.4.4. The columns are considered as polynomials over $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by equation (2.16)

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

As described in Sec. 4.3, this can be written as a matrix multiplication. Let

$$s'(x) = a^{-1}(x) \otimes s(x)$$

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad for \;\; 0 \le c < Nb \qquad (2.23)$$

As a result of this multiplication, the four bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c}) \\ s'_{1,c} &= (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c}) \\ s'_{2,c} &= (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c}) \end{aligned} \qquad (2.24)$$

2.5.4. AddRoundKey Transformation (Forward and Inverse Transformations)

In the forward add round key transformation, called AddRoundKey, the 128 bits of State are bitwise XORed with the 128 bits of the round key. As shown in Figure 2–10-b, the operation is viewed as a column wise operation between the 4 bytes of a State column and one word of the round key; it can also be viewed as a byte-level operation.

2.6. AES Key Expansion Algorithm

The AES key expansion algorithm takes as input a 4-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a 4-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The Figure below shows pseudo-code describes the expansion:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

Note that Nk=4, 6, and 8 do not all have to be implemented;
they are all included in the conditional statement above for
conciseness.   Specific implementation requirements for the
Cipher Key are presented in Sec. 6.1.
```

Figure 2–12: Key Expansion Pseudo-code

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word *w[i]* depends on the immediately preceding word, *w[i 1]*, and the word four positions back, *w[i 4]*. In three out of four cases, a simple XOR is used. For a word whose position in the w array is a multiple of 4, a more complex function is used. Figure 2–13 illustrates the generation of the first eight words of the expanded key, using the symbol g to represent that complex function. The function g consists of the following sub-functions:

1. RotWord performs a one-byte circular left shift on a word. This means that an input word $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.
2. SubWord performs a byte substitution on each byte of its input word, using the S-box (Table 2–2-a).
3. The result of steps 1 and 2 is XORed with a round constant, *Rcon[j]*.

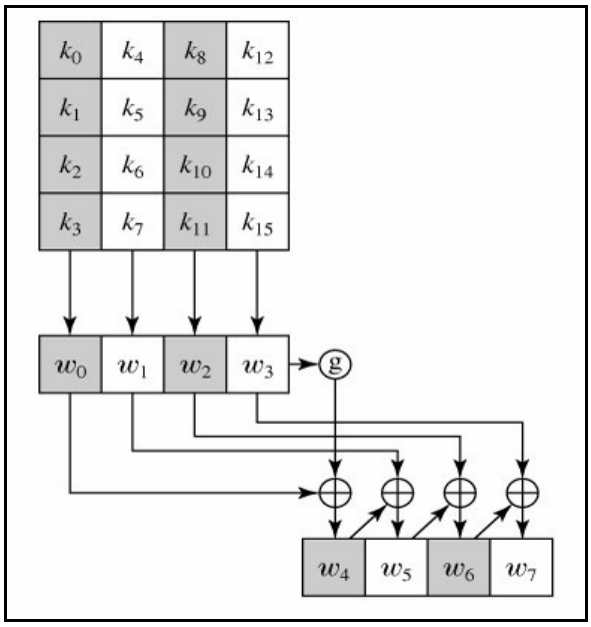


Figure 2–13: AES Key Expansion

The round constant is a word in which the three rightmost bytes are always 0. Thus the effect of an XOR of a word with *Rcon* is to only perform an XOR on the leftmost byte of the word. The round constant is different for each round and is defined as *Rcon[j] = (RC[j], 0, 0, 0)*, with *RC[1] = 1, RC[j] = 2 · RC[j - 1]* and with multiplication defined over the field $GF(2^8)$. The values of RC[j] in hexadecimal are shown in Table 2–3

Table 2–3: Rcon *[j]* Values

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

2.7. Equivalent Inverse Cipher

As was mentioned, the AES decryption cipher is not identical to the encryption cipher (Figure 2–8). That is, the sequence of transformations for decryption differs from that for encryption, although the form of the key schedules for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm. The equivalent version has the same sequence of transformations as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

Two separate changes are needed to bring the decryption structure in line with the encryption structure. An encryption round has the structure SubBytes, ShiftRows, MixColumns and AddRoundKey. The standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey and InvMixColumns, Thus, the first two stages of the decryption round need to be interchanged, and the second two stages of the decryption round need to be interchanged.

2.7.1. Interchanging InvShiftRows and InvSubBytes

InvShiftRows affects the sequence of bytes in State but does not alter byte contents and does not depend on byte contents to perform its transformation. InvSubBytes affects the contents of bytes in State but does not alter byte sequence and does not depend on byte sequence to perform its transformation. Thus, these two operations commute and can be interchanged. For a given State S_i,

$$InvShiftRows\ [InvSubBytes\ (S_i)] = InvSubBytes\ [InvShiftRows\ (S_i)]$$

2.7.2. Interchanging AddRoundKey and InvMixColumns

The transformations AddRoundKey and InvMixColumns do not alter the sequence of bytes in State. If we view the key as a sequence of words, then both AddRoundKey and InvMixColumns operate on State one column at a time. These two operations are linear with respect to the column input. That is, for a given State S_i and a given round key w_j:

$$InvMixColumns\ (S_i \oplus w_j) = [InvMixColumns\ (S_i)] \oplus [InvMixColumns\ (w_j)]$$

Figure 2–14 illustrates the equivalent decryption algorithm.

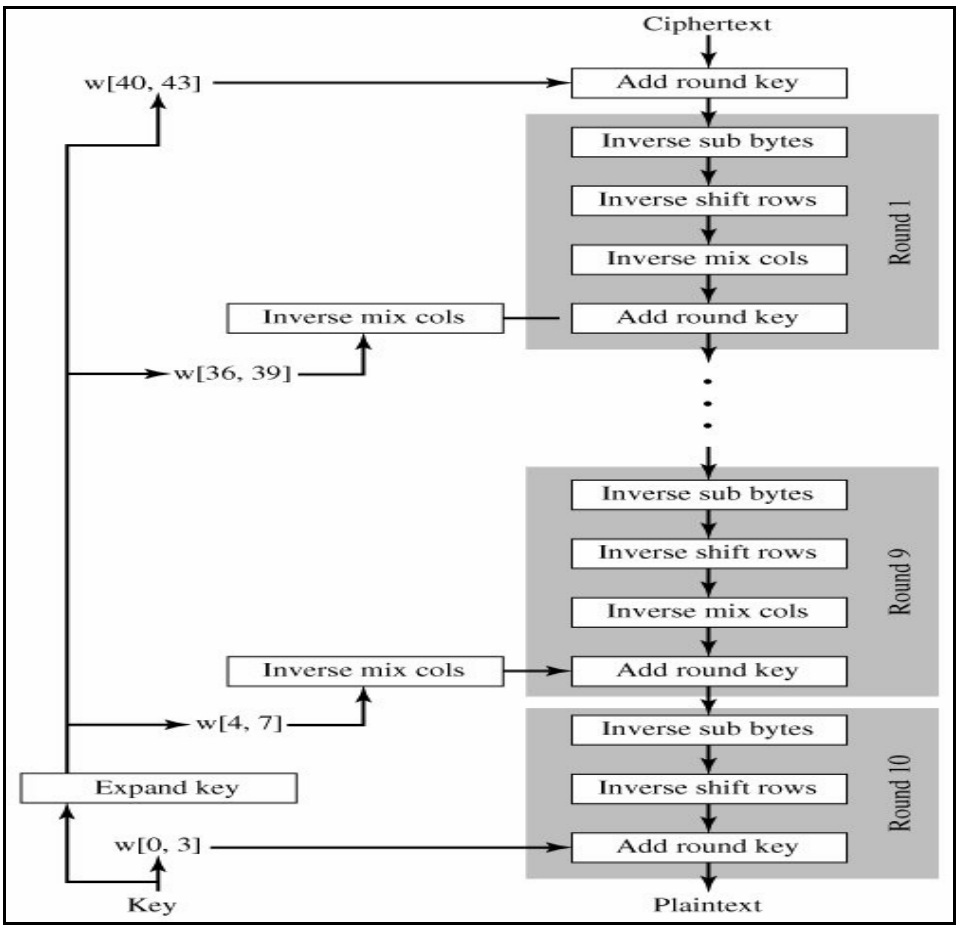


Figure 2–14: Equivalent Inverse Cipher

Chapter 3

HEURISTIC DESIGN OF RIJNDAEL S-BOX

Cipher systems are a prime target for an attacker wishing to compromise the information being protected by a security system. In line with the three forms of protection mentioned above, the typical motives of an attacker include seeking to reveal confidential information, to illicitly and surreptitiously modify information, and to falsely claim an identity. In addition, an attacker may seek to remove evidence, or even insert false evidence, that a particular event or transaction has occurred. Compromising a cipher system which endeavors to protect this information can either directly enable these actions to occur, or indirectly weaken another part of the system to enable these actions to later occur. Powerful existing cryptanalytic attacks against cipher systems have proved to be successful under the right conditions [11].

The overall strength of a security system is dependent on the strength of the individual components, such as the authentication system, the key management system, the data storage system, the cipher system and the policies and procedures, to name a few. Similarly, the overall strength of a cipher system is dependent on the strength of its individual components. A weakness in any of the individual components may lead to a catastrophic failure in the whole cipher.

Boolean functions and substitution boxes (s-boxes) are two of the most common and critical components of cryptographic cipher systems. These components are directly related by function quantity. That is, a substitution box is typically comprised of multiple single output Boolean functions, but if it maps to only one bit, is identical to a Boolean function.

Boolean functions are often utilized in the keystream generation process of stream ciphers as they are highly suitable for receiving bits of linear feedback shift registers as input in order to combine them as securely as possible to produce the single keystream. Further, Boolean functions are capable of exhibiting the combination of cryptographic properties necessary to resist the typical types of attacks which seek to reveal part or all of the keystream.

The most common type of cipher system which employs s-boxes are block ciphers. As a block cipher system encrypts its data in fixed length blocks, s-boxes are a natural component of such a system. They provide a means of substituting multiple bits (part of or a whole block) of data for a completely different set of output bits. More importantly is the use of strong s-boxes (those which possess good cryptographic properties) so the substitution signifies a complex relationship between input and output bits of the s-box. The typical use of s-boxes in the cipher's iterative round function serves to increase the effort needed to exploit any statistical structure in the data.

Boolean functions and s-boxes will only be able to contribute to the security of a cipher by possessing good measures of desirable cryptographic properties. Obtaining strong

Boolean functions and s-boxes for incorporation into cryptographic cipher systems to enhance their security is an ongoing research problem. This is particularly so as cryptanalytic techniques become more sophisticated, and with the advancement of computing technology which works both for and against cryptographic security. The size of Boolean functions and the dimension of s-boxes have a significant bearing on security, though larger functions generally require more computational effort in order to exploit weaknesses, so too is the computational effort increased when attempting to obtain large functions with exceptionally good measures of desirable cryptographic properties. This adds an extra element of difficulty to the research problem.

In this chapter we propose a new AES S-box with good cryptographic properties such as high nonlinearity and low autocorrelation.

3.1. Boolean Function and S-box Theory

This section provides some definitions of relevance to Boolean functions with cryptographic application. We denote the substitution table of an n-input k-output Boolean function by $f : B^n \to B^k$, mapping each combination of n Boolean input values to some combination of k Boolean output values [12].

For single-output functions if the number of combinations mapping to 0 is the same as the number mapping to 1 then the function is said to be *balanced*. For the multiple-output case, if each k-bit output value appears the same number of times, we say that the function is *regular*.

For the single-output case the substitution table is generally referred to as a 'truth table'. The *polarity truth table* is a particularly useful representation for our purposes. It is defined by $\hat{f}(x) = (-1)^{f(x)}$. Two functions f and g are said to be uncorrelated when $\sum_{x \in B_n} \hat{f}(x)\hat{g}(x) = 0$. If so, if you try to approximate f by using g, you will be right half the time and wrong half the time.

An area of particular importance for cryptanalysts is the ability to approximate a function f by a simple linear function. One of the cryptosystem designer's tasks is to make such approximation as difficult as possible (by making the function f suitably *nonlinear*). Linearity is a form of structure crypto designers clearly strive to avoid. One form of attack that exploits linearity is known as linear cryptanalysis, introduced by Matsui [13]. It has attracted a great deal of attention. Another form of structure that is to be avoided is *differential* structure. Essentially, particular differences in input words (difference defined by simple bitwise XOR) may be associated with particular differences of output words (again defined by bitwise XOR) with some strong bias (i.e. the output difference is not uniform for a particular input difference). This can often be exploited by a form of attack known as differential cryptanalysis, introduced by Biham and Shamir [14], [15].

Substitution boxes are essentially n-input k-output functions. These can be viewed as a combination of k individual single-output Boolean functions. Several important security criteria are actually defined in terms of single-output function criteria and so it is essential

to understand first the basic Boolean function definitions and concepts. We then extend these to cater for the multiple-output case.

3.2. Cryptographic Criteria for Single-output Functions and S-boxes

Two formal criteria have been defined for the single-output case to capture some aspects of resilience to the sorts of attacks indicated above [11, 12]. These are high nonlinearity and low autocorrelation and are defined below together with other terminology used in this chapter.

Linear Boolean Function: A linear Boolean function *f*, selected by $\omega \in B^n$, is denoted by $L_\omega(x) = \omega_1 x_1 \oplus \omega_1 x_1 \ldots \oplus \omega_n x_n$, Where $\omega_i x_i$ denotes the bitwise AND of the i^{th} bits of ω and x, and $\oplus$ denotes bitwise XOR.

Affine Boolean Function: The set of affine functions is the set of linear functions and their complements $A_{\omega,c}(x) = L_\omega(x) \oplus c, \quad c \in B$

Walsh Hadamard Transform: For a Boolean function f the Walsh Hadamard Transform $\hat{F}_f$ is defined by $\hat{F}_f(\omega) = \sum_{x \in B^n} \hat{f}(x)\hat{l}_\omega(x)$. We denote the maximum absolute value taken by the transform by $WH_{\max}(f) = \max_{\omega \in B^n} \left|\hat{F}_f(\omega)\right|$. It is related to the nonlinearity of *f*.

Nonlinearity: The nonlinearity N_f of a Boolean function *f* is its minimum distance to any affine function. It is given by $N_f = \frac{1}{2}(2^n - WH_{\max}(f))$.

Parseval's Theorem: This states that $\sum_{\omega \in B^n} (\hat{F}(w))^2 = 2^n$. A consequence of this result is that $WH_{\max}(f) \geq 2^{n/2}$.

Autocorrelation Transform: The autocorrelation transform of a Boolean function *f* is given by $\hat{r}_f(s) = \sum_x \hat{f}(x)\hat{f}(x \oplus s)$. We denote the maximum absolute value in the autocorrelation spectra of a function *f* by AC_f, *i.e.*, $AC_f = \max_s \sum_x \hat{f}(x)\hat{f}(x \oplus s)$. Here x and s range over B^n.

Extensions to S-boxes: For each k-output S-box, we can extract a single-output Boolean function by simply *XORing* some subset of the output bits together. If $f(x): B^n \to B^k$ is an *n*-input *k*-output S-box then each $\beta \in B^k$ defines a function that is a linear combination $f_\beta(x)$ of the *m* outputs of *f*. This is given by

$$f_\beta(x) = \beta_1 f_1(x) \oplus \beta_2 f_2(x) \ldots \oplus \beta_k f_k(x)$$

For each such function f_β the Walsh-Hadamard values $\hat{F}_\beta(\omega)$ and autocorrelation values $r_\beta(s)$ are defined in the usual way. (Each such function is now a single-output function defined over the *n* inputs.) There are 2^k-1 non-trivial functions obtainable in this way. The notions of non-linearity and autocorrelation are readily extended to the multiple output case. For the *k*-output case the non-linearity is the worst (lowest) non-linearity of all the 2^k-1 non-trivial single output functions obtained as indicated above. Similarly, the autocorrelation is the worst (highest) over all such derived single-output functions.

3.3. Cost Functions

3.3.1. Traditional Cost Functions

In virtually all work done so far existing optimization based work aimed at producing highly nonlinear functions has generally used nonlinearity itself as the fitness function, i.e. the fitness of a function f on n input variables is given by

$$fitness(f) = N_f = \frac{1}{2}(2^N - WH_{\max}(f)) \tag{3.1}$$

Or, when viewed as a minimization problem, the cost function is given by

$$cost(f) = WH_{\max}(f) = \max_\omega \left|\hat{F}(\omega)\right| \tag{3.2}$$

Similarly, with low autocorrelation as the target, the autocorrelation itself has been used as the cost function, i.e. the cost function is given by

$$cost(f) = \max_{s\neq 0}\left|\sum_x \hat{f}(x)\hat{f}(x+s)\right| = \max_{s\neq 0}\left|\hat{r}(s)\right| \tag{3.3}$$

Previous optimization approaches to evolving Boolean functions with desirable cryptographic properties have been generalized to the multiple-output case. Millan has compared random generation and hill-climbing as means of evolving highly nonlinear bijective S-boxes [16]. Burnett *et al.* have investigated the use of genetic algorithms and hill-climbing to evolve regular S-boxes [17]. Both high nonlinearity and low autocorrelation were targets. The fitness and cost measures for an S-box were the nonlinearity and autocorrelation values of that S-box. For the S-box case, the researchers above have used extensions of the basic definitions as cost functions. For non-linearity the cost function was:

$$cost(f) = \max_{\beta\in B^k,\omega\in B^n}\left|\hat{F}_\beta(\omega)\right| \tag{3.4}$$

For autocorrelation the cost function was:

$$cost(f) = \max_{\beta \in B^k \setminus \{0^k\}, s \in B^n \setminus \{0^n\}} \left| \hat{r}_\beta(s) \right| \qquad (3.5)$$

3.3.2. Spectrum Based Cost Functions

Traditional optimization work in non-linearity attempts to improve the non-linearity directly. Equivalently (see the definition of N_f in Section 3.3.1), it seeks to minimize the cost function

$$cost(f) = WH_{\max}(f)$$

Essentially, the search considers the effect of a move only on those extreme (or near extreme) values of the Walsh Hadamard Transforms $\hat{F}(\omega)$ for the current solution. A more indirect approach can be derived by considering Parseval's theorem below.

$$\sum_{\omega \in B^n} (\hat{F}(\omega))^2 = 2^{2n}$$

This constrains $WH_{\max}(f) = \max_{\omega \in B^n} |\hat{F}(\omega)|$ to be at least $2^{\frac{n}{2}}$. It would achieve this bound when for each ω, $\hat{F}(\omega) = 2^{\frac{n}{2}}$. In practice this bound may be impossible. When some $|\hat{F}(\omega)|$ are greater than this ideal bound, Parseval's theorem ensures that some $|\hat{F}(\omega)|$ must be smaller than it. Thus, it would appear that attempting to restrict the spread of absolute values achieved is well-motivated. This suggests a cost function of the following form:

$$cost(\hat{f}) = \sum_{\omega \in B^n} \left| \left| \hat{F}(\omega) \right| - 2^{\frac{n}{2}} \right|^R \qquad (3.6)$$

The value *R* is positive and can be varied. Note that it does not *necessarily* follow that a reduction in our cost function gives rise to an increase in non-linearity but if the range of absolute values is small, then the maximum value will be small too.

The above cost function could be written as:

$$cost(\hat{f}) = \sum_{\omega \in B^n} \left| \left| \hat{F}(\omega) \right| - X \right|^R \qquad (3.7)$$

Where X and R are real-valued parameters. It is difficult to predict what the best such parameter values should be and considerable experimentation is needed. However, as indicated above, they have produced some exceptional results (effectively equaling the

best results of theoreticians for functions of 8-inputs or less). A similar cost function obtained by substituting $\hat{r}_f(s)$ for $\hat{F}_f(\omega)$ was later used to similar effect [18], [19].

Since spectrum-based approaches generated interesting results for the single-output case an obvious question to pose is 'Can the spectrum-based approaches be generalized to allow S-boxes to be evolved with desirable properties?' Two cost functions can now be defined for use in S-box evolution. A cost function based on Walsh-Hadamard spectra is given by

$$\cos t(\hat{f}) = \sum_{\beta \in B^k} \sum_{\omega \in B^n} \left| \left| \hat{F}_\beta(\omega) \right| - X \right|^R \tag{3.8}$$

And a similar cost function based on autocorrelation spectra is given by

$$\cos t(\hat{f}) = \sum_{\beta \in B^k} \sum_{s \in B^n} \left| \left| \hat{r}_\beta(s) \right| - X \right|^R \tag{3.9}$$

The single output cost functions have been applied to each function defined as a linear combination of the outputs and the results summed over all such combinations.

3.4. Optimization Algorithms of a Single Boolean Function

The two main techniques which have been used for this purpose by researchers in the field are [11]:

1. Heuristic techniques; and
2. Algebraic constructions.

Heuristic techniques are driven by a directed search algorithm typically searching in a localized area from a specified starting point. Their use is more frequent for searching in large spaces in order to find a large number of solutions which are satisfactory, but generally not optimal. For this reason, heuristic techniques are often applied to difficult combinatorial problems. Well known heuristic techniques include Simulated Annealing [43], Tabu Search [31], Genetic Algorithms [37] and Hill Climbing techniques [70].

Algebraic constructions rely on proven mathematical relationships holding for a generalized construction of functions. Whilst algebraic constructions have been shown to generally produce functions with the most optimum combinations of properties, they are not typically designed to produce a great number of such functions. Further, the existence of inherent weaknesses in functions produced by algebraic construction is a valid concern. In contrast, the vast amount of experimentation so far performed using heuristic techniques has shown that, for large input spaces, these techniques are generally unable to generate optimal functions. This is due to the nature of the technique as simply being a way to non-deterministically search through a search space in a directed fashion. Thus, as the number of input variables increases by one, the number of functions in the space increases by a factor of 2^{2N} and the probability of discovering optimal functions decreases.

However, because heuristic techniques involve directed search methods, they have been shown to produce consistent results in finding functions with good properties, and unlike algebraic constructions, are able to produce a large number of such functions. For this reason, the approach taken in this section has been primarily focused on the application of heuristic techniques.

3.4.1. Hill Climbing

The basic hill climbing technique involves searching, at each iteration, for elements of a function to modify which will result in an improvement in the results already obtained. At the end of the process, it is expected that the final output will represent the best solution obtainable.

For cryptographic applications used in this research, hill climbing is referred to as being the process whereby one or more distinct elements in the truth table of a function are complemented in order to make iterative improvements to the cryptographic properties or fitness of the function. The fitness of a function is the measure of a particular cryptographic property or properties exhibited by the function. In [20], the authors categorize the fitness function into either weak or strong acceptance. A weak acceptance condition will accept an incremental change in the truth table even if such a change produces no increase in the fitness of the new function, provided that there is no decrease in the fitness. A strong acceptance condition, on the other hand, will only accept an incremental change in the truth table when such a change produces an increase in the fitness of the new function. Thus, the only time an increase in the fitness is forced is when a strong acceptance condition is imposed. In addition to relying on this measure as a criterion for deciding whether to accept or reject functions to be input into the next iteration of the process, hill climbing requires the formation of improvement sets. Improvement sets are defined according to the fitness function which is utilized in the hill climbing process.

The hill climbing approach to Boolean function design was introduced in [21] as a mean of improving the nonlinearity of a given Boolean function by making well chosen alterations of one or two places of the truth table. It is easy to show that any single truth table change causes $\Delta_{WHT}(\omega) \in \{-2, 2\}$ for all ω. Any two changes cause $\Delta_{WHT}(\omega) \in \{-4, 0, 4\}$. When the two function values satisfy $f(x_1) \neq f(x_2)$ then the Hamming weight will not change. By starting with a balanced function, we can hill climb to a more nonlinear balanced function.

Paper [21] has introduced the requirements for improvement of the WHT for one and two changes to the truth table. Here we briefly give a more general derivation of the rules for the two change case (to keep function balancing).

Consider a given Boolean function *f(x)* in polarity truth table form $\hat{f}(x)$. Now let the truth table output be complemented for two distinct inputs x_1 and x_2. We have $\hat{g}(x_i) = -\hat{f}(x_i)$ *for* $i \in \{1, 2\}$, and $\hat{g}(x) = \hat{f}(x)$ for other *x*. Now consider the WHT of *g(x)*.

$$\begin{aligned}\hat{G}(\omega) &= \sum_x \hat{g}(x)\hat{L}_\omega(x) \\ &= \hat{g}(x_1)\hat{L}_\omega(x_1) + \hat{g}(x_2)\hat{L}_\omega(x_2) + \sum_{x \notin \{x_1, x_2\}} \hat{g}(x)\hat{L}_\omega(x) \\ &= -(\hat{f}(x_1)\hat{L}_\omega(x_1) + \hat{f}(x_2)\hat{L}_\omega(x_2)) + \sum_{x \notin \{x_1, x_2\}} \hat{f}(x)\hat{L}_\omega(x)\end{aligned}$$

We will naturally define the change in the WHT value for all ω as

$$\Delta_{WHT}(\omega) = \hat{G}(\omega) - \hat{F}(\omega).$$

It follows directly that

$$\Delta_{WHT}(\omega) = -2\hat{f}(x_1)\hat{L}_\omega(x_1) - 2\hat{f}(x_2)L_\omega(x_2).$$

This result can be used directly to quickly update the WHT each iteration of a 2-step hill climbing program. It is now a straightforward matter to determine the conditions required for the choice of *(x1, x2)* to complement so that the WHT values change as required. It is clear that two changes ensure $\Delta_{WHT}(\omega) \in \{-4, 0, 4\}$. As in all hill climbing methods we assume $f(x_1) \neq f(x_2)$ has been fixed, so that Hamming weight doesn't change. We have

$$\begin{aligned}&\Delta_{WHT}(\omega) = -4 \Leftrightarrow both\, f(x_i) = L_\omega(x_i)\, for\, i \in \{1,2\}, \\ &\Delta_{WHT}(\omega) = +4 \Leftrightarrow both\, f(x_i) \neq L_\omega(x_i)\, for\, i \in \{1,2\}\ and \\ &\Delta_{WHT}(\omega) = 0 \quad \Leftrightarrow one\ f(x_i) = L_\omega(x_i)\ and\ another \\ &\qquad\qquad\qquad\qquad f(x_i) \neq L_\omega(x_i)\, for\, i \in \{1,2\}.\end{aligned}$$

This specifies the tests for all conditions of interest in 2-step hill climbing. When we require definite improvement of the WHT and wish to maintain Hamming weight, then we may complement the truth table output for any pair (*x1, x2*) that satisfies all the following conditions:

$$\begin{aligned}&(i)\, f(x_1) \neq f(x_2) \\ &(ii)\, both\, f(x_i) = L_\omega(x_i)\ for\ i \in \{1,2\}, for\ all\ \{\omega : \hat{F}(\omega) = WH_{\max}\} \\ &(iii)\, both\, f(x_i) \neq L_\omega(x_i)\ for\ i \in \{1,2\}, for\ all\ \{\omega : \hat{F}(\omega) = -WH_{\max}\} \\ &(iv)\, not\ both\, f(x_i) \neq L_\omega(x_i)\ for\ i \in \{1,2\}, for\ all\ \{\omega : \hat{F}(\omega) = (WH_{\max} - 4)\} \\ &(v)\, not\ both\, f(x_i) = L_\omega(x_i)\ for\ i \in \{1,2\}, for\ all\ \{\omega : \hat{F}(\omega) = -(WH_{\max} - 4)\}\end{aligned}$$

Figure 3–1 shows the Boolean function hill climbing algorithm.

1. Generate a random Boolean function f and calculate its Walsh Hadamard transform.
2. By parsing the WHT find the values of ω which belong to the groups $WH_{\max}, -WH_{\max}, (WH_{\max}-4), -(WH_{\max}-4)$.
3. For i =1 to 2^n-1
   For $j = i$+1 to 2^n
   If $f(i) \neq f(j)$
   $g=f;$
   $g(i)=f(j);$
   $g(j)=f(i);$
   if conditions ii, iii, iv, v satisfied
   $f=g;$
   Goto 2;
4. Go to step 1 until no improvement occurs.

Figure 3–1: Boolean function Hill Climbing Algorithm

A similar hill climbing algorithm [20] could be used to improve the autocorrelation of Boolean function.

Figure 3–2 shows the nonlinearity achieved by the hill climbing algorithm for 100 iterations. One can see that the maximum achievable nonlinearity achieved using this algorithm is 112.

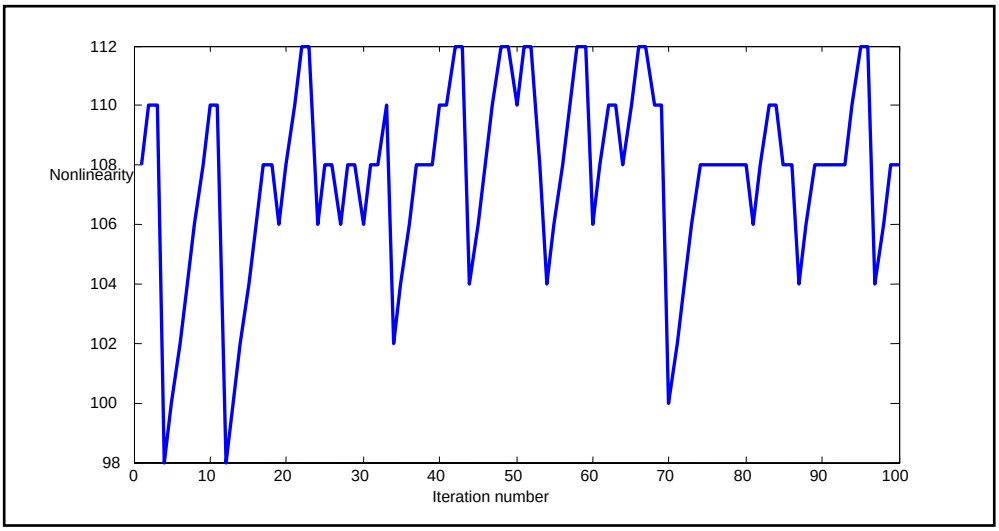


Figure 3–2: Hill Climbing Algorithm output (Nonlinearity vs Iteration number)

3.4.2. Simulated Annealing

In 1983 Kirkpatrick et al. [22] proposed a new search technique based on the cooling processes of molten metals. The technique was *simulated annealing*. It has proved to be an extraordinarily simple, yet powerful, heuristic search technique. It merges hill-climbing with the probabilistic acceptance of non-improving moves. The basic algorithm is shown in Figure 3–1.

```
S = S_0
T = T_0
Repeat
{
        for(int i = 0; i < MIL; i++)
        {
                Select Y ∈ N(S)
                δ = f(Y) − f(S)
                if (δ < 0) then
                        S = Y
                else
                        Generate U = U(0, 1)
                        if (U < exp(−δ/T)) then S = Y
        }
        T = T × α
}
Until stopping criterion is met
```

Figure 3–3: Basic Simulated Annealing for Minimization Problems

The search starts at some initial state $S=S_0$. There is a control parameter T known as the temperature. This starts 'high' at T_0 and is gradually lowered. At each temperature, a number *MIL* (Moves in Inner Loop) of moves to new states are attempted. A candidate state Y is randomly selected from the neighborhood $N(S)$ of the current state. The change in value, δ, of f is calculated. If it improves the value of $f(s)$ (i.e. if the $\delta<0$ for a minimization problem) then a move to that state is taken ($S=Y$); if not, then it is taken with some probability. The worse a move is, the less likely it is to be accepted. The lower the temperature T the less likely is a worsening moves to be accepted. Probabilistic acceptance is determined by generating a random value in the range (0...1) and performing the indicated comparison. Initially the temperature is high and virtually any move is accepted. As the temperature is lowered it becomes ever more difficult to accept worsening moves. Eventually, only improving moves are allowed and the process becomes 'frozen'. The algorithm terminates when the stopping criterion is met. Common stopping criteria, and the ones used for the work in this thesis, are to stop the search after a fixed number *MaxIL* of inner loops have been executed, or else when some maximum number *MUL* of consecutive unproductive inner loops have been executed (an inner loop is termed unproductive if no move is accepted within it). Generally the best state achieved so far will also be recorded (since the search may actually move out of it and subsequently be unable to find a state of similar quality). At the end of each inner loop the temperature is lowered. The simplest way of lowering the temperature is shown. This is known as geometric cooling. The basic simulated annealing algorithm has proven remarkably effective over a range of problems. This technique will be used (with hill-climbing) to improve the AES S-box.

Figure 3–4 shows the Boolean function optimization results for (*MaxIL=100)* and *(MUL=100)* and *(T=10)* and *(α=0.9)*. It is clear that the maximum achievable Nonlinearity using this method is 114.

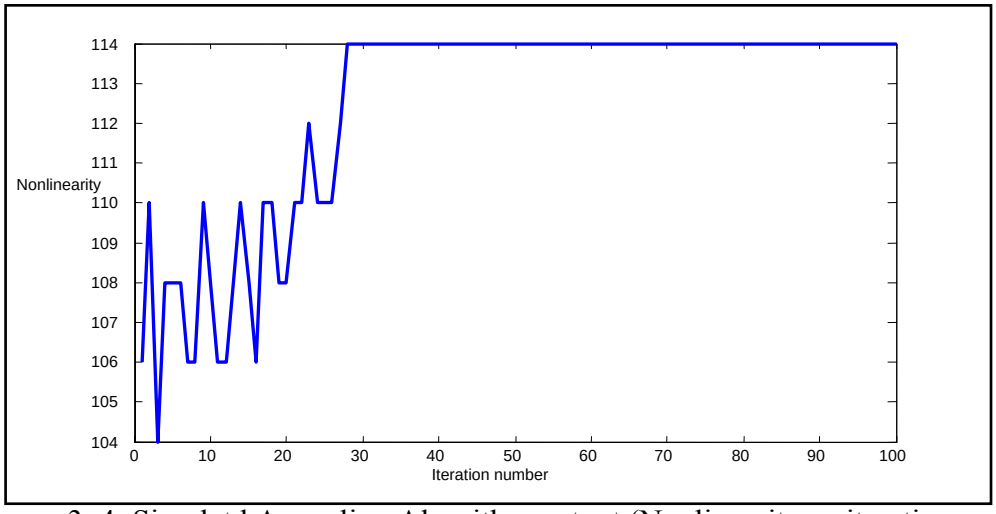


Figure 3–4: Simulatd Annealing Algorithm output (Nonlinearity vs iteration number)

Figure 3–5 shows the Simulated annealing results when the optimization objectives are both high nonlinearity and low autocorrelation. It is clear that the maximum achievable nonlinearity is 112 and the minimum achievable autocorrelation is 48.

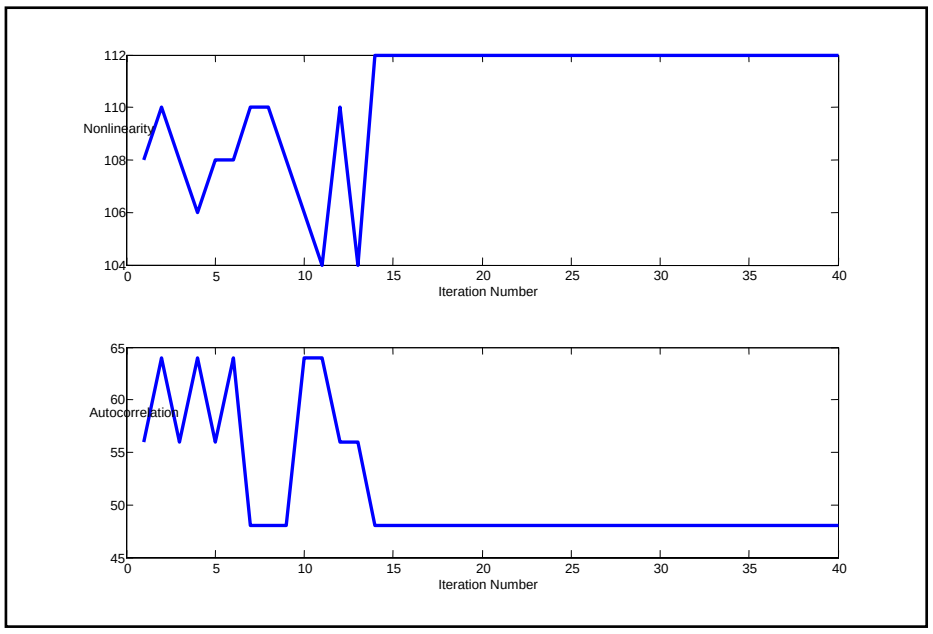


Figure 3–5: Simulatd Annealing Algorithm output (Nonlinearity & Autocorrelation vs iteration number)

3.4.3. Tabu Search

Tabu search is a widely used modern local search technique. The next move to take is decided using cost function values but also historical information (i.e. it uses memory of some form). This allows the search to escape from local optima and also to explore the search space in a productive fashion. Tabu search generally adopts a best improvement local search but moderates this policy using historical information.

If a particular solution *S* is reached then it becomes 'tabu' for some number T_s of transitions, generally referred to as the solution's tabu tenure. If a solution is tabu, the search is normally prevented from moving to that solution, i.e. the local neighborhood from which the next solution is chosen excludes those solutions that are currently tabu. Conceptually, the currently tabu solutions together with their remaining tabu tenures form a 'tabu list'. In its simplest form, with common tabu tenure of *T*, the list becomes a FIFO queue. The most recently visited solution is added and the solution visited *T* moves ago is removed. The tabu list implements what is generally referred to as a *recency* criterion. It prevents the search revisiting solutions in the short term (and so short cycles are prevented). The higher the tabu tenure the more the search is forced to explore the solution space. The tabu tenure may be varied during the search. Figure 2.2 outlines a basic tabu search procedure (taken from [23], which provides an interesting consideration of metaheuristic techniques more generally).

$S = GenerateInitialSolution()$
$InitializeTabuLists(TL_1, \ldots, TL_r)$
$k = 0$
while termination conditions not met **do**
{
$AllowedSet(S, k) = \{z \in N(s)|$ no tabu condition is violated or at least one aspiration criterion is satisfied $\}$
$S = BestImprovement(S, AllowedSet(S, K))$
$UpdateTabuListsAndAspirationConditions()$
k=k+1
}
end while

Figure 3–6: Basic Tabu Search Procedure

In practice maintaining lists of *solutions* is very inefficient. Much more common is to keep lists of solution attributes or moves. Consider an object permutation problem, i.e. where objects $O_1, O_2 \ldots O_n$ must be arranged in some order (and there is a cost associated with each such order). If a move *(i,j)* (with $i<j$) is taken that swaps the positions of objects O_i and O_j then this could be made tabu for a period. A more stringent tabu criterion would make any move involving object O_i or object O_j tabu. Thus, taking move (1, 4) would render tabu any move of the form (a,b) where either a or b is equal to 1 or 4. Other features may be taken into account. For example, the actual cost associated with a solution could be made tabu. The search would be prevented from visiting solutions with the same cost function value for the tabu tenure.

The tabu status of a move can be relaxed if taking that move would give rise to a particularly good solution, most typically a solution better than any reached so far (this is generally referred to as the *aspiration criterion*). Other aspects of history can also be taken into account, such as long-term frequencies of particular move types. The notion of *influence* is also used to guide the search; a move that causes greater change (measured in some fashion) is deemed to be more influential. Thus, influence criteria can be created and applied to diversify the search. For an excellent discussion of tabu search details the reader is referred to the chapter on tabu search by Glover in [24].

Figure 3–7 shows the Boolean function optimization results for Tabu search with long term memory. It is clear that the maximum achievable Nonlinearity using this method is 114.

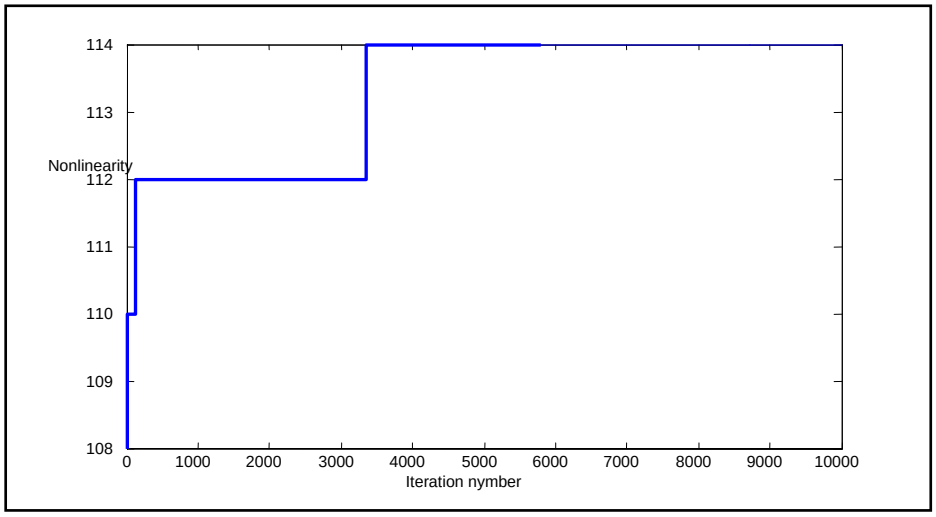


Figure 3–7: Tabu search Algorithm output (Nonlinearity & Autocorrelation vs iteration number)

Figure 3–8 shows the Simulated annealing results when the optimization objectives are both high nonlinearity and low autocorrelation. It is clear that the maximum achievable nonlinearity is 112 and the minimum achievable autocorrelation is 48.

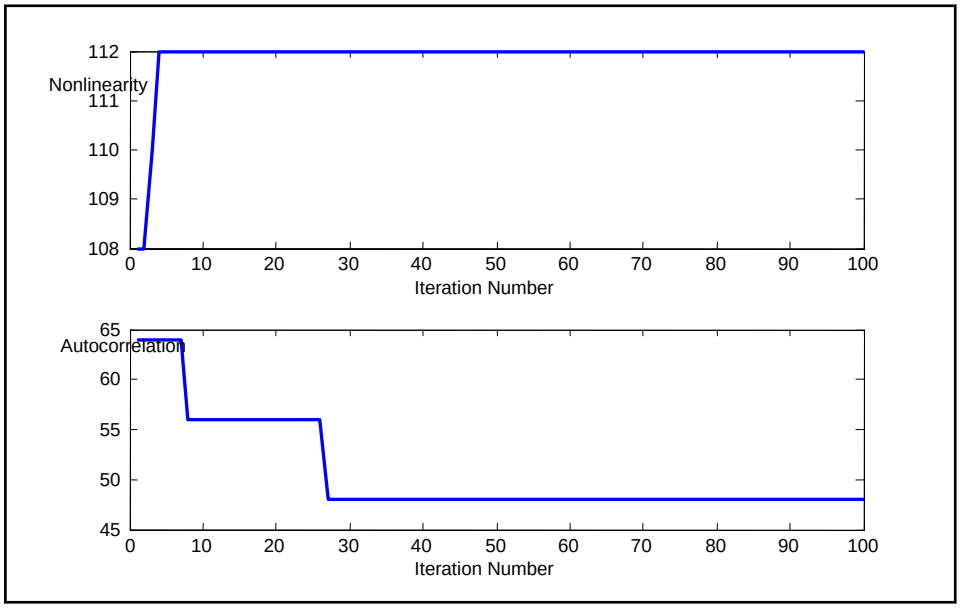


Figure 3–8: Tabu Search Algorithm output (Nonlinearity & Autocorrelation vs iteration number)

3.4.4. Genetic Algorithms

Genetic algorithms are part of a class of what is known as Evolutionary algorithms. Evolutionary algorithms are computational models that solve a given problem by

maintaining a changing population of individuals, each with its own level of "fitness". The change in the population is achieved by the selection, reproduction and mutation procedures within the method. The operation of these three procedures is dependent upon the fitness of the individuals concerned [11].

We begin by defining some terminology relating to natural selection:

- ***Parent pool:*** contains the current set of candidate solutions.
- ***Parents:*** the pair of individuals in the parent pool chosen for breeding.
- ***Children:*** offspring resulting from the breeding of two parents.
- ***Breed:*** the process whereby two parents are combined or mated to produce a child.
- ***Fitness:*** the measure taken in order to ascertain which individuals will survive to the next generation.

Genetic algorithms are characterized by the fact that all the information for any individual in the population is encoded using some linear encoding system. This (usually binary) encoding is intended to be analogous to natural DNA consisting of a string of four kinds of chromosomes.

Initially, a pool of P solutions is selected randomly and the fitness of each solution in the pool is calculated. Here, the pool consists of truth tables corresponding to (initially random) balanced Boolean functions. From this pool pairs of parents are chosen to act as the parents of the next generation. Parents may be chosen randomly, based on their fitness or exhaustively (all possible paring are tried). The breading process requires some mating function for combining parent solutions. We illustrate the general process of breeding by describing below two common schemes for breeding functions in a population pool.

- ***Roulette Wheel:*** This scheme arises from the idea that the parents in the population pool occupy a particular percentage angle on a roulette wheel, the size of which is determined proportionately by their fitness measure. Thus, fitter individuals occupy greater angles on the wheel and have a higher chance of selection with the spinning of the wheel [25].
- ***Crossover:*** This breeding scheme is based on the genetic mechanism of crossover which occurs in sexual reproduction. In this natural process, genetic variation results from the breaking and recombination of linked genes in homologous chromosomes, thus producing offspring with combined attributes of two parents. Function breeding schemes based on crossover extend this idea by choosing a random position in the two parent functions at which the crossover of elements will begin and subsequently interchanging the elements in the parent functions from this point. Thus, the resulting offspring will take the elements of the first parent up to and including the element at the crossover point and the elements of the second parent for the remaining positions [26].

Here we use a merging operation which combines two parents to produce a single offspring. The offspring will be a balanced function which is similar to each of its parents (the merge operation described in detail below). Typically, each of the offspring undergoes some mutation. As will be seen below, the merging operation used incorporates

a random mutation so a separate mutation operation is not required. At this stage the survivors for the next iteration are chosen. This involves combining the parents and offspring pools and selecting the most fit as the new solution pool for the next iteration.

The merging (or mating) operation is now described [18]. This operation takes two balanced Boolean functions as input and produces a single balanced Boolean function as offspring. Consider two Boolean functions of n inputs. The truth tables corresponding to these functions will contain 2^n bits. Call the two parent functions p_1 and p_2, and let $p_k[i]$ denote the i^{th} in the truth table of parent k. Also, n_1 denotes the number of 1's which have been placed in the child in positions where the parents differ, and dist (p_1, p_2) is the Hamming distance between the two truth tables, p_1 and p_2. The objective of the algorithm is to ensure that a child is balanced. The offspring c is determined as shown in Figure 3–9.

1. Let $n_1 = 0$ and $k = 0$.
2. If $\text{dist}(p_1, p_2) > 2^n/2$ complement p_1 or p_2.
3. For $i = 1$ to 2^n do:
   (a) If $p_1[i] = p_2[i]$ then $c[i] = p_1[i]$ $(= p_2[i])$;
   (b) Otherwise (if $p_1[i] \neq p_2[i]$)
      i. If $n_1 = \text{dist}(p_1, p_2)/2$ then $c[i] = 0$;
      ii. Else if $n_1 + \text{dist}(p_1, p_2) - k = \text{dist}(p_1, p_2)/2$ then $c[i] = 1$;
      iii. Else $c[i] =$ a random bit.
      iv. Increment k $(k = k + 1)$.
      v. If $c[i] = 1$ then $n_1 = n_1 + 1$.

Figure 3–9: Breeding Scheme of the Genetic Algorithm

The check in step 2 is to ensure that only parents which are close to each other are allowed to breed. It should be noted that complementing a Boolean function's truth table doesn't alter its nonlinearity. The checks in step 3(b)i and 3(b)ii are used to force offspring to be balanced. The overall genetic algorithm is shown in

1. Generate a pool of P random, balanced Boolean functions (represented by their truth table) and calculate the fitness of each. Call this pool S_0.
2. For $i = 1$ to MAXITER do:
   (a) For all $P(P-1)/2$ pairings of the pool S_{i-1} perform the merging operation (as described above) to produce $P(P-1)/2$ offspring.
   (b) (Optional hill climbing.) Apply the hill climbing procedure to each of the offspring.
   (c) Determine the fitness of each of the offspring.
   (d) Combine the pool of offspring with the current pool, S_{i-1}, and select the P best as the new pool, S_i. Give precedence to offspring with fitness equal to solutions already in the pool. Duplicate solutions should be removed.
   (e) (Optional *resetting* step.) If there has been no improvement in the fitness of the best solution for a number of iterations, then keep the best solution and generate $P - 1$ random, balanced functions as the remainder of the pool.
3. Output the best solution from the current pool.

Figure 3–10: Genetic Algorithm to Improve Nonlinearity of Boolean function

Figure 3–11 shows the Boolean function optimization results for Genetic algorithm for initial population (p=10) and 10 iterations. It is clear that the maximum achievable Nonlinearity using this method is 114.

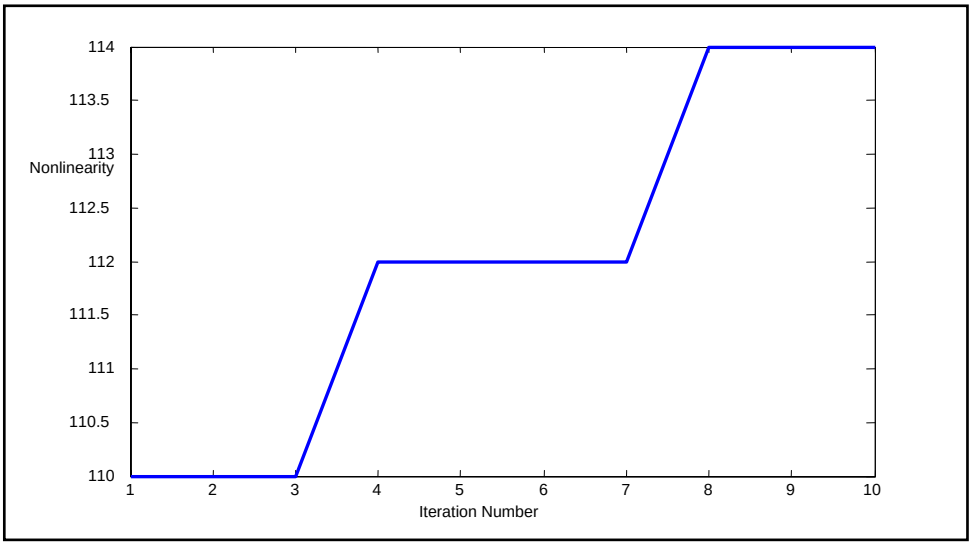


Figure 3–11: Genetic Algorithm output (Nonlinearity vs iteration number)

Figure 3–12 shows the Genetic algorithm results when the optimization objectives are both high nonlinearity and low autocorrelation (p=5 and number of iterations=100). It is clear that the maximum achievable nonlinearity is 114 and the minimum achievable autocorrelation is 40.

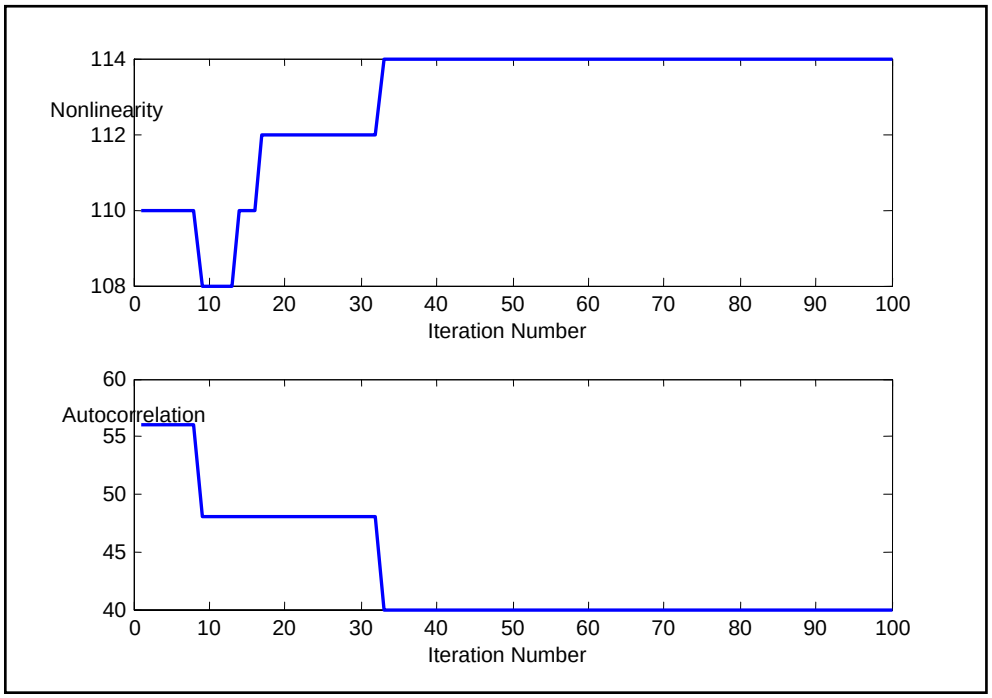


Figure 3–12: Genetic Algorithm output (Nonlinearity & Autocorrelation vs iteration number)

3.4.5. Comparison between Different Optimization Results

Table 3–1 shows our optimization results for the nonlinearity and autocorrelation of 8 inputs balanced Boolean function. The best achievable result for nonlinearity as cited in [18] is 116 and the upper bound of the nonlinearity is 118. The best achievable results for autocorrelation as cited in [12] is 40 and the lower bound of the nonlinearity is 32.

Table 3–1: Comparison between nonlinearity and autocorrelation for different optimization algorithms

	Hill Climbing	Simulated Annealing	Tabu Search	Genetic Algorithm
Nonlinearity	112	114	114	114
Autocorrelation	--	48	48	40

3.5. Optimization Algorithms of S-box

A substitution box (or S-box) is a mapping from n binary inputs to m binary outputs. Any S-box may be described by the set of m single output Boolean functions. The main cryptographic interest has been with reversible, or bijective, S-boxes. For an S-box to be bijective $n=m$, and all possible output vectors appear exactly one each [16]. A bijective S-box implements a permutation of the input vectors. From this it is easy to show that every linear combination of the output is a balanced function.

We will use the same optimization algorithms used with a single Boolean function to optimize n x n S-box. For the n-output case the non-linearity is the worst (lowest) non-linearity of all the linear combination of the n Boolean function. Similarly, the autocorrelation is the worst (highest) over all such derived single-output functions.

Figure 3–13 shows the genetic algorithm optimization results of *8x8* s-box nonlinearity for initial population ($p=5$) and ten iterations. The maximum nonlinearity achieved by this algorithm is 100.

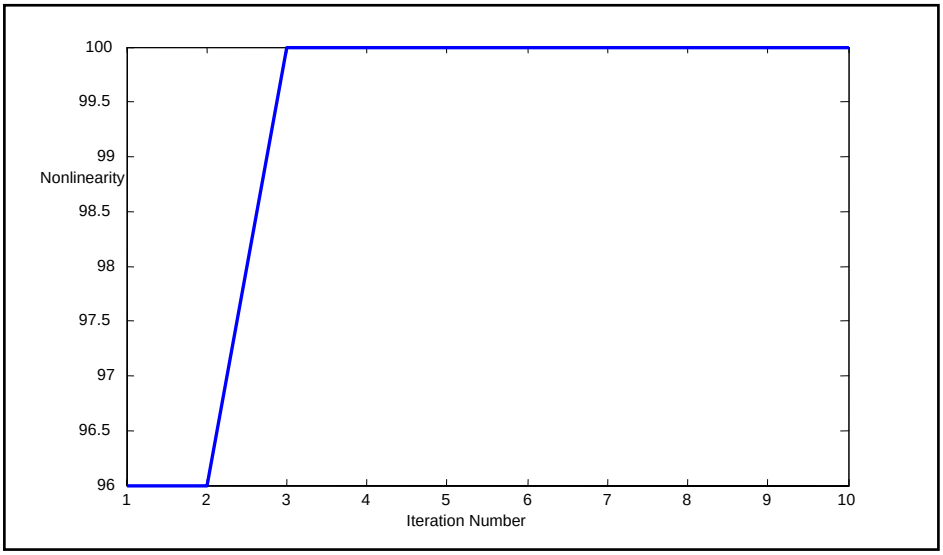


Figure 3–13: S-box Genetic Algorithm output (nonlinearity vs iteration number)

Figure 3–14 shows the simulated annealing optimization results of *8x8* s-box nonlinearity for (*MIL=20*), (*T=100*), (*α=0.9*) and 40 iterations. The maximum nonlinearity achieved by this algorithm is 98.

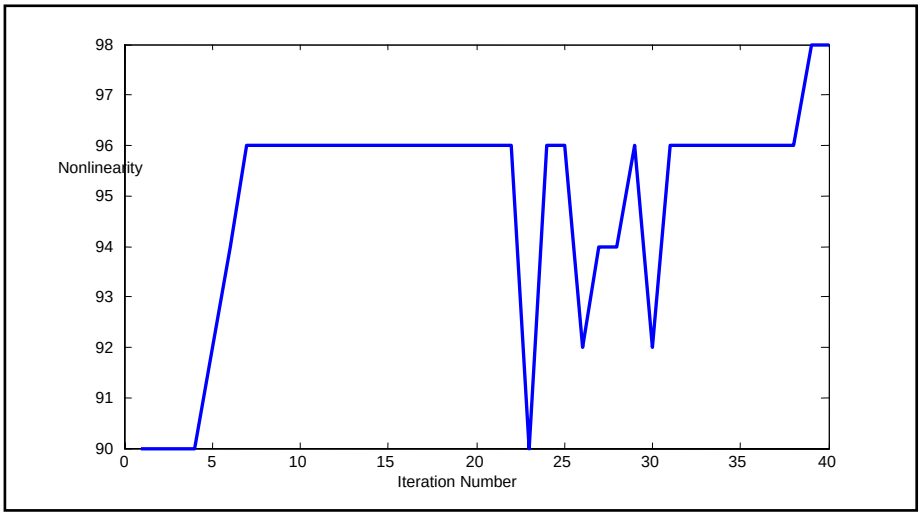


Figure 3–14: S-box simulated annealing output (nonlinearity vs iteration number)

Figure 3–15 shows the Tabu search with long term memory optimization results of *8x8* s-box nonlinearity for 100 iterations. The maximum nonlinearity achieved by this algorithm is 96.

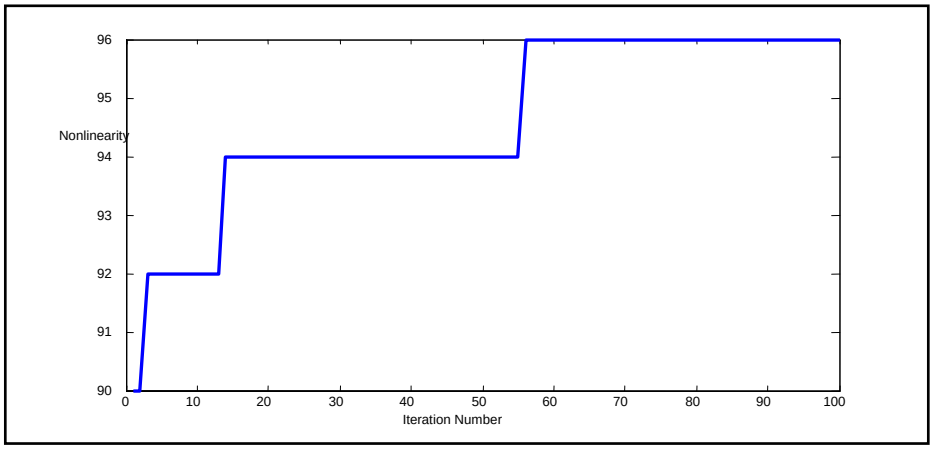


Figure 3–15: Tabu search S-box output (nonlinearity vs iteration number)

Figure 3–16 shows the Genetic algorithm results when the optimization objectives are both high nonlinearity and low autocorrelation (p=5 and number of iterations=100). It is clear that the maximum achievable nonlinearity is 114 and the minimum achievable autocorrelation is 40.

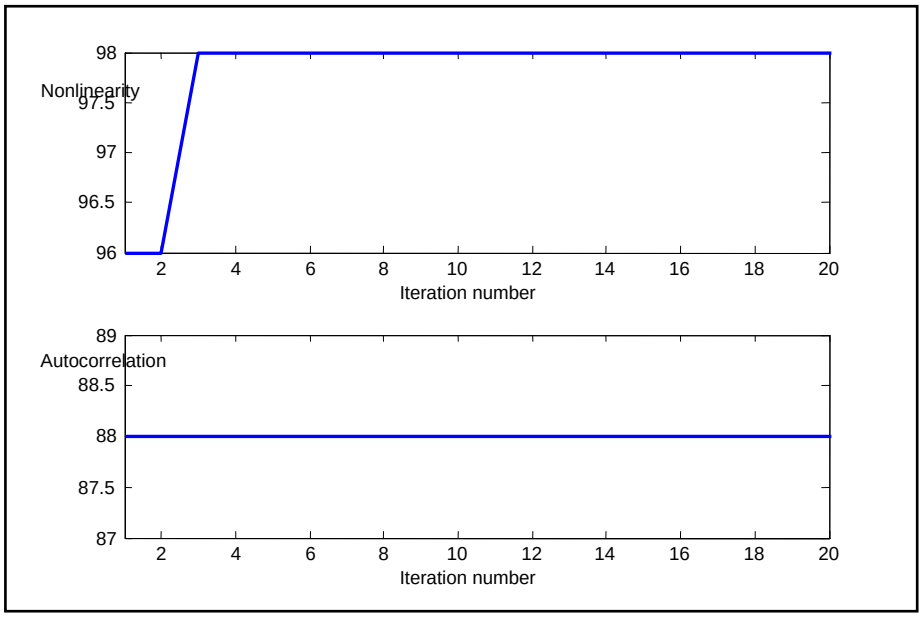


Figure 3–16: S-box Genetic Algorithm output (Nonlinearity & Autocorrelation vs iteration number)

It is clear that the best results are achieved using genetic algorithm and our new suggested Rijndael like s-box will be that one obtained using genetic algorithm and its objectives are high nonlinearity and low autocorrelation (98,88). Table 3–2 shows our suggested Rijndael like s-box.

Table 3–2: Rijndeal Like S-Box

		Y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
X	0	D6	81	89	12	86	2A	2D	C4	7E	DF	1D	9E	A2	97	94	21
	1	37	53	1B	C8	C5	52	6F	8F	77	3A	EA	E1	8D	19	9F	F8
	2	8B	CD	43	36	7C	26	32	F7	59	BA	F4	EF	61	AF	82	6E
	3	C7	DC	85	3C	ED	5	15	46	0B	35	8A	B5	B0	7	65	DD
	4	9B	1C	9	5C	4	A5	2B	1E	64	FA	F5	EE	22	C0	58	49
	5	E9	27	B8	79	40	E3	3F	8	BE	3D	33	5B	B1	90	34	B2
	6	5A	CB	AB	1A	F2	3E	4C	29	67	6	CF	DB	AA	D7	A1	47
	7	83	F9	E5	80	16	57	4A	4F	F0	B7	BC	C2	84	0C	99	5E
	8	A4	FF	48	55	0E	B9	17	CC	93	D4	13	0	FE	3B	9C	8C
	9	6B	0A	44	10	D0	50	C9	D5	E8	69	91	B6	88	1F	25	DE
	A	B3	7B	2F	0F	98	BF	D8	A6	C1	39	41	18	75	60	24	1
	B	0D	4D	6C	14	73	D3	7F	20	E6	74	7A	63	E7	66	A7	2C
	C	A8	E4	EC	D9	76	6D	30	31	BB	CE	D1	92	2	42	BD	7D
	D	68	62	E2	9D	E0	DA	D2	28	95	38	5D	2E	F6	CA	71	A9
	E	6A	96	54	9A	FC	23	8E	51	5F	F3	A3	AC	FB	C6	87	78
	F	4B	B4	A0	56	72	3	AE	AD	45	F1	11	EB	C3	4E	70	FD

Chapter 4

IMPLEMENTATION APPROACHES FOR THE AES

In this chapter, we introduce the architectural optimization approaches for the AES algorithm; algorithmic optimizations for each round unit in the AES algorithm are described. The last section explores resource sharing between encryptor and decryptor.

4.1. Architectural Optimization

A block cipher encrypts plain text in fixed-size n-bit blocks (n = 128 for AES). Messages longer than n bits are divided into n-bit blocks, and each block is encrypted separately. Basically, there are five modes of operation: electronic codebook (ECB), cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB) and counter (CTR) mode. Non-feedback (NFB) modes such as the ECB mode offer less security, but can achieve great speedup by processing multiple blocks simultaneously. The other three basic modes belong to feedback (FB) mode, which can offer a higher level of security but can hardly achieve any speedup by multi-block processing due to the existence of feedback the processing of the next block cannot begin until the current block is finished.

4.1.1. Architectures of AES Encryptor/ Decryptor

Three types of architectures can be used to increase the speed of encryptor/ decryptor by duplicating hardware for implementing each round, which is also called round unit in this paper. These architectures are based on pipelining, sub-pipelining, and loop unrolling [27]. They are illustrated in Figure 4–1 together with basic reference architecture.

4.1.1.1. Pipelining

The pipelined architecture can increase the speed of encryption/ decryption by processing multiple blocks of data simultaneously. It is realized by inserting rows of registers among combinational logic. Parts of logic between two consecutive registers form pipeline stages. Each pipeline stage is one round unit in this case. During each clock cycle, the partially processed data moves to the next stage and its place is taken by the subsequent data block. The number of round units in each loop, k, is usually chosen as a divisor of Nr and the maximum value of k is Nr, in which case it becomes a fully pipelined architecture. For a k-round pipelined architecture, when a partially processed block reaches the k^{th} round, it will be fed back to the first round until all the Nr rounds are performed on this block. After the pipeline reaches its full depth, that is after the first block reaches the k^{th} stage, k blocks of data are processed simultaneously in different stages and k blocks of data are processed every Nr cycles. The area of the pipelined architecture is proportional to k.

4.1.1.2. Sub-Pipelining

Similar to the pipelining, sub-pipelining also inserts rows of registers among combinational logic, but in this case, registers are inserted both between and inside each round unit. If each round unit can be divided into r stages with equal delay, a k-round sub-pipelined architecture can achieve approximately r times the speed of a k-round pipelined architecture with a slight increase of area caused by additional registers and control logic. However, dividing each round unit into an arbitrary number of stages does not always bring speedup. Since the minimum clock period is decided by the indivisible combinational element with the longest delay, dividing the rest of the round unit into more stages with shorter delay does not reduce the minimum clock period. Although more blocks of data are being processed simultaneously, the average number of clock cycles to process one block of data is increased by the same proportion. Therefore the overall speed does not improve despite increased area caused by the additional registers.

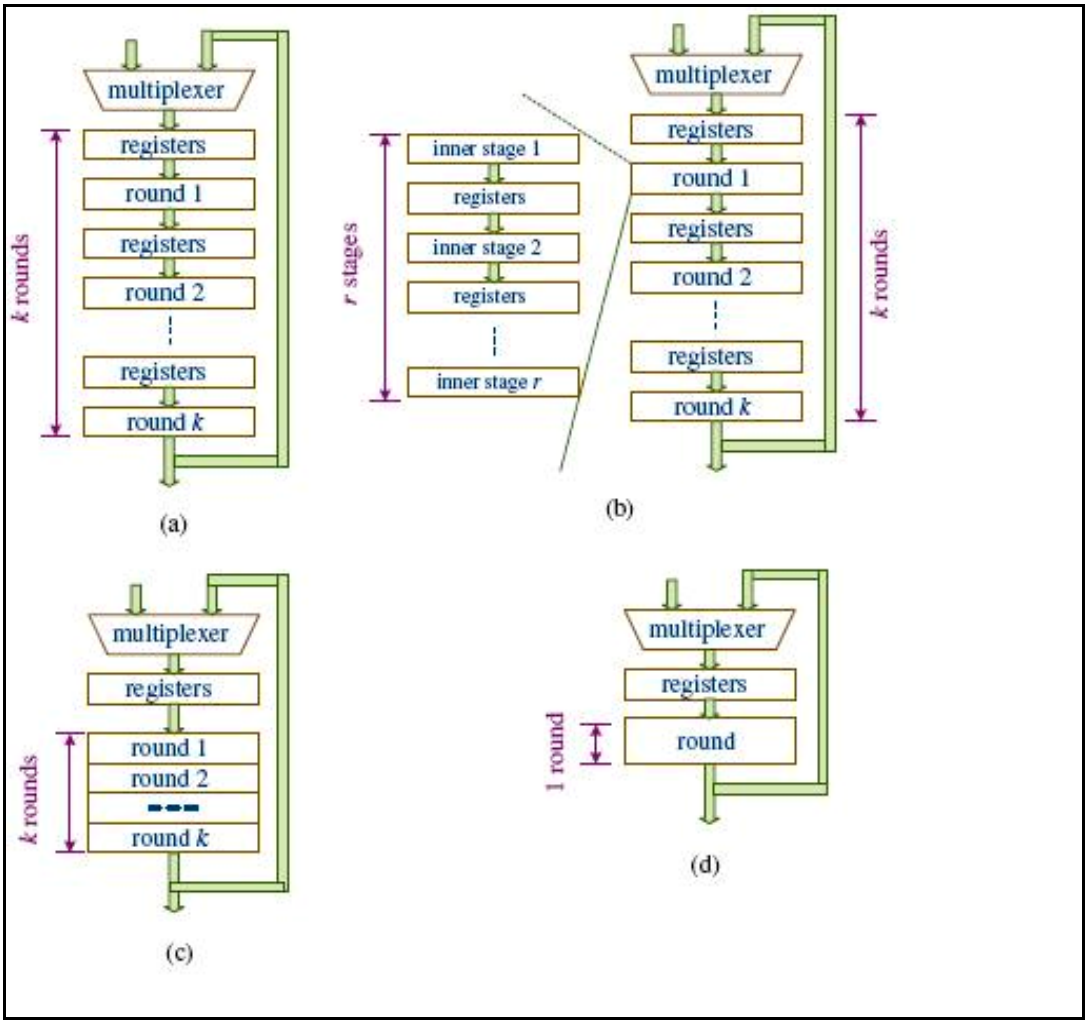


Figure 4–1: Three types of architecture of encryptor/decryptor with a basic reference architecture: (a) pipelined architecture, (b) sub-pipelined architecture, (c) loop unrolled architecture, (d) basic reference architecture

4.1.1.3. Loop Unrolling

Loop unrolled or unfolded architectures can process only one block of data at a time, but multiple rounds are performed in each clock cycle. The unrolling or unfolding factor, k, is usually chosen as a divisor of Nr and the maximum value of k is Nr. The number of cycles to process one block of data is Nr / k in this case. Meanwhile, the clock period of a k-round loop unrolled architecture is increased to slightly smaller than k times the clock period of a pipelined architecture because of the setup time and propagation delay of registers. The area of this architecture is also proportional to the number of rounds in each loop.

Most of the proposed implementations can be classified into one of the above three architectures. Detailed studies of all these architectures were carried out in [28] and [29]. In this section, we separately address the speedup factor of these three architectures for FB and NFB modes compared to the basic reference architecture.

4.1.2. Architectural Optimization for Non-Feedback Modes

The speed of a system can be measured by throughput, which is given by

$$Throughput = average\ \ number\ \ of\ \ bits\ \ processed / \text{second}$$

In the case of the AES algorithm, it can also be expressed as

$$Throughput = 128 / (average\ \ number\ \ of\ \ clock\ \ cycles\ \ to\ \ process\ \ one\ \ block\ x\ \ clock\ \ period) \qquad (4.1)$$

Maximum achievable throughput for each architecture is compared in this section. In the basic architecture in Figure 4–1-d, only one round is performed in each clock cycle, so Nr clock cycles are needed to finish processing one block of data. The minimum clock period t_{basic} can be expressed as

$$t_{basic} = t_{round} + t_{setup} + t_{prop} + t_{mux} \qquad (4.2)$$

In the above equation, t_{round} is the delay of the combinational logic in each round unit; t_{mux} denotes the delay of the multiplexer, whereas t_{setup} and t_{prop} stands for the setup time and propagation delay of the registers, respectively. From equation (4.1), the maximum achievable throughput of the basic architecture is given by

$$throughput_{basic} = 128\ /\ (Nr\ x\ t_{basic}).$$

In the pipelined architecture in Figure 4–1-a, assuming k is a divisor of Nr, after the initial k clock cycles, k blocks of data are processed every Nr cycles. Meanwhile, the minimum clock period is the same as that of the basic architecture. The speedup of pipelined architecture over the basic architecture is $throughput_{pipe} / throughput_{basic} = k$

The area of this architecture is proportional to the number of pipeline stages, *k*. Tradeoffs between area and speed can be easily made by changing *k*. In the sub-pipelined architecture of the AES algorithm, each of the round units should be divided into no more than two stages according to the former discussion in this section.

SubBytes/InvSubBytes is usually implemented by look-up tables. ShiftRows/InvShiftRows does not need any logic to implement, MixColumns/InvMixColumns can be implemented by XOR gates, and AddRoundKey is only one step of XOR operation. Hence each round unit is usually divided into $r = 2$ stages, one for SubBytes/InvSubBytes transformation and another for the rest of the transformations. Assuming the two stages in each round have equal delay, $2k$ blocks of data will be processed every $2Nr$ cycles after the pipeline reaches its full depth. Let $\tau = (t_{setup} + t_{prop} + t_{mux}) / t_{round}$. The minimum clock period of sub-pipelining is $(0.5 + \tau)/(1 + \tau)$ times that of the basic architecture. The speed up of a *k*-round sub-pipelined architecture with $r = 2$ is given by

$$throughput_{sub\text{-}pipe} / throughput_{basic} = k(1 + \tau)/(0.5 + \tau) \qquad (4.3)$$

The area of sub-pipelined architecture is also proportional to the parameter *k* but does not change much with *r*. Increasing number of inner round stages only introduces more registers, whose area is small compared to the total area of implementation.

The throughput of this architecture is $(1 + \tau)/(0.5 + \tau)$ times that of a pipelined architecture with the same *k*. Usually τ is small, so there is almost twice speedup over pipelining at the cost of $r - 1$ additional rows of registers.

In the loop unrolled architecture in Figure 4–1-c, assuming *k* is a divisor of *Nr*, one block of data is processed every *Nr* / *k* cycles. However, the minimum clock period is increased to

$$t_{lu} = k \times t_{round} + t_{setup} + t_{prop} + t_{mux} \qquad (4.4)$$

Which is $(k + \tau)/(1 + \tau)$ times the minimum clock period of the basic architecture. Hence the speedup of a *k*- round loop unrolled architecture can be expressed as

$$throughput_{lu} / throughput_{basic} = (1 + \tau)/(1 + \tau / k) \qquad (4.5)$$

The area of loop unrolled architecture is also proportional to the number of rounds per loop; *k*. Compared to the *k*-stage pipelined architecture, the speedup is much lower at roughly the same area.

Depending on different optimization criteria, different architectures can be employed. Optimization for maximum speed can be realized by a fully sub-pipelined architecture. In the application requiring minimum area, the basic architecture is desired. In the case of optimum speed/area ratio, sub- pipelining seems to be the best choice. Numerous implementations of these architectures on different technologies have been studied. The

reported fastest FPGA implementation can reach 12 Gbit/sec on a Xilinx Virtex-E XCV812E-8BG560 device for a fully pipelined 128-bit key encryptor in NFB modes [30].

4.1.3. Architectural Optimization for Feedback Mode

In feedback modes, the encryption/ decryption of the next block cannot start until the current block is finished. In this case, pipelining does not lead to any speedup, because only one stage is processing one block of data in each cycle, while the other stages are idle. Meanwhile, the area increases proportionally to *k*. Therefore pipelined architecture is not suitable for feedback applications. Loop-unrolled architecture, however, can bring some speedup at the cost of significantly increased area. The speedup which can be achieved is the same as that in non-FB modes given by equation (4.5). Sub-pipelining can even deteriorate the performance; *Nr x r* cycles are needed to encrypt/decrypt one block of data, but even in the optimum case when each inner stage has equal delay, the clock period is longer than t_{basic} / r because of the setup and propagation delay of the registers. The fastest implementation for FB modes reported so far employed a fully loop unrolled architecture, and achieved a throughput of 1950.03 Mbits/sec based on Mitsubishi Electric's 0.35 micron CMOS technology [31].

4.2. Algorithmic Optimization

A complete AES system can be divided into three major blocks: Key Expand, Control, and EnDecrypt, as illustrated in Figure 4–2. The Key Expand block loads keys, performs Key Expansion transformation, and generates proper roundkeys under the control signals from the Control block. Control block takes 'start' signal, 'reset' signal, 'enc' signal, and 'key_length' signal from outside and generates all the control signals for the whole system. The 'enc' signal and the 'key_length' signal are optional. The 'enc' signal is the control signal for encryption/decryption; it is needed when the system can perform both encryption and decryption. The 'key_length' signal gives the key length information; it is needed when the system can perform multiple key length encryption/decryption. The EnDecrypt block gets roundkeys from the Key-Expand block and encrypts/ decrypts 'data_in' according to the AES algorithm. Each of the architectures pipelining, loop unrolling, and sub-pipelining covered in the last section can be used in the EnDecrypt block. The speed and area trade-offs of the AES algorithm can not only be made by changing the overall architecture of the EnDecrypt block, but also by exploiting the implementation of each round unit. A variety of methods have been brought up to implement individual round unit [31], [33–37]. They are discussed in detail in this section.

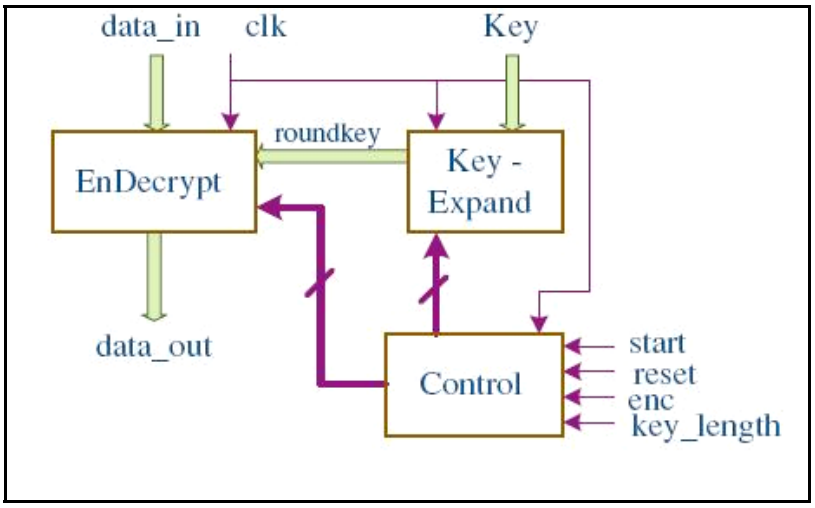


Figure 4–2: Block diagram of the AES system

4.2.1. Implementation of Separate Transformations

No optimization can be performed on ShiftRows/ InvShiftRows and AddRoundKey transformations, since no logic gates are needed for the former transformation and only one step of XOR operation is needed for the latter. However, different methods can be used to implement the SubBytes/ InvSubBytes and MixColumns/InvMixColumns transformations.

4.2.2. Implementation of SubBytes/ InvSubBytes

SubBytes/InvSubBytes is usually implemented by look-up tables. Each S-box/S^{-1}-box needs a look-up table of 256 x 8 = 2k-bits and each round needs 16 S-boxes/S^{-1}-boxes, so the area for look-up tables becomes huge when multiple round units are implemented. For area critical applications, a better choice is to map the arithmetic operations on $GF(2^8)$ to isomorphic field $GF((2^4)^2)$. This implementation requires smaller area for look-up tables, but has longer delay [20, 21].

4.2.3. Implementation of MixColumns/ InvMixColumns

In the MixColumns transformation, we need to implement constant multiplication of {02} and {03} in $GF(2^8)$. Assuming X is a byte in the State, $\{02\}X$ can be implemented by shifting and bit-wise XOR operations, and $\{03\}X$ can be computed by $(\{02\}X) \oplus X$. If X is expressed in binary form as $\{x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0\}$, $\{02\}X$ can be calculated by

$$\{02\}X = \{x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0, 0\} \oplus \{0, 0, 0, x_7, x_7, 0, x_7, x_7\} \qquad (4.6)$$

Since $0 \oplus x_i = x_i$, equation (4.6) only needs 4 XOR gates to implement. The block diagram in Figure 4–3 shows the straightforward way to calculate $S'_{0,c}$ ($0 \le c < 4$) in the MixColumns transformation [8]. Since $\{01\}X = X$, X instead of $\{01\}X$ is used in this and the following figures. Calculation of $S'_{1,c}$, $S'_{2,c}$, and $S'_{3,c}$ can be done by connecting

appropriate $\{02\}X$, $\{03\}X$, or X of $S'_{1,c}$, $S'_{2,c}$, and $S'_{3,c}$ to the last row of XOR gates in Figure 4–3 according to equation (2.22). As shown in the figure, the critical path has 4 XOR gates and a total of (4 x 8 + 4) x 4 =144 2-input XOR gates is needed to implement the MixColumns transformation for one column of the State.

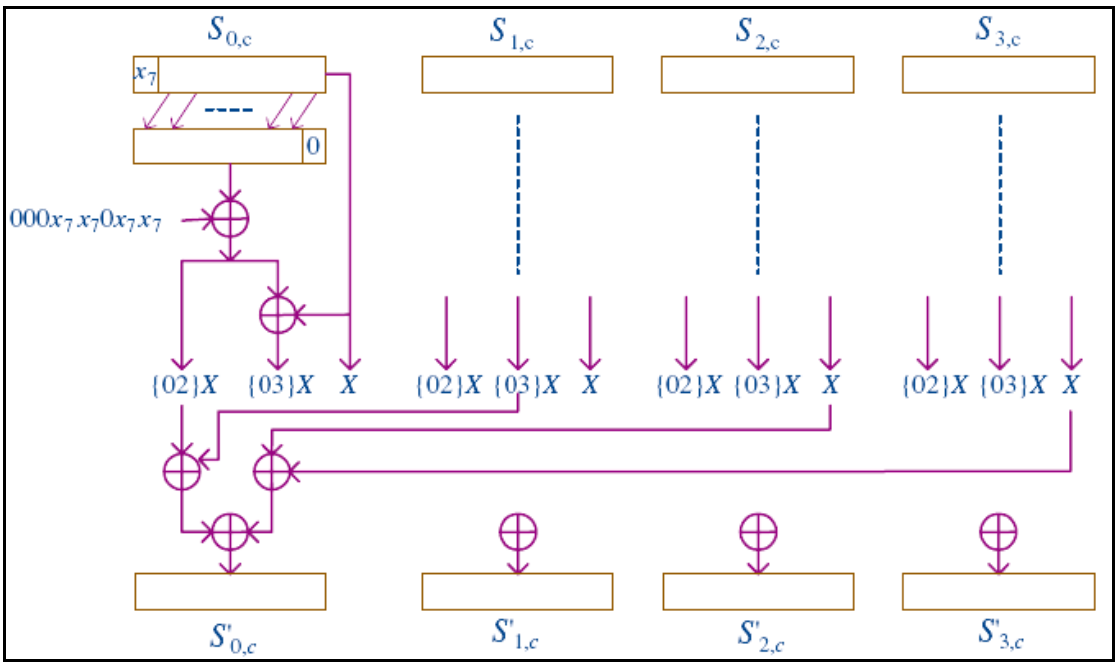


Figure 4–3: Block diagram for straightforward implementation of the MixColumns transformation

The InvMixColumns transformation is more complicated. Constant multiplications used in the InvMixColumns transformation can be expressed as

$$\{0b\}X = \{08\}X \oplus \{02\}X \oplus X, \qquad \{0d\}X = \{08\}X \oplus \{04\}X \oplus X,$$
$$\{09\}X = \{08\}X \oplus X, \qquad \{0e\}X = \{08\ \}X\{ \oplus \{04\}X \oplus \{02\}X$$

A straightforward way to calculate $S'_{0,c}$ $(0 \le c < 4)$ of the State in the InvMixColumns transformation is illustrated in Figure 4–4. In order to simplify the diagram, an XTime block is introduced as shown in Figure 4–5. XTime block implements the constant multiplication by $\{02\}$ in $GF(2^8)$ [9], each XTime block consists of 4 XOR gates and the critical path includes only one XOR gate.

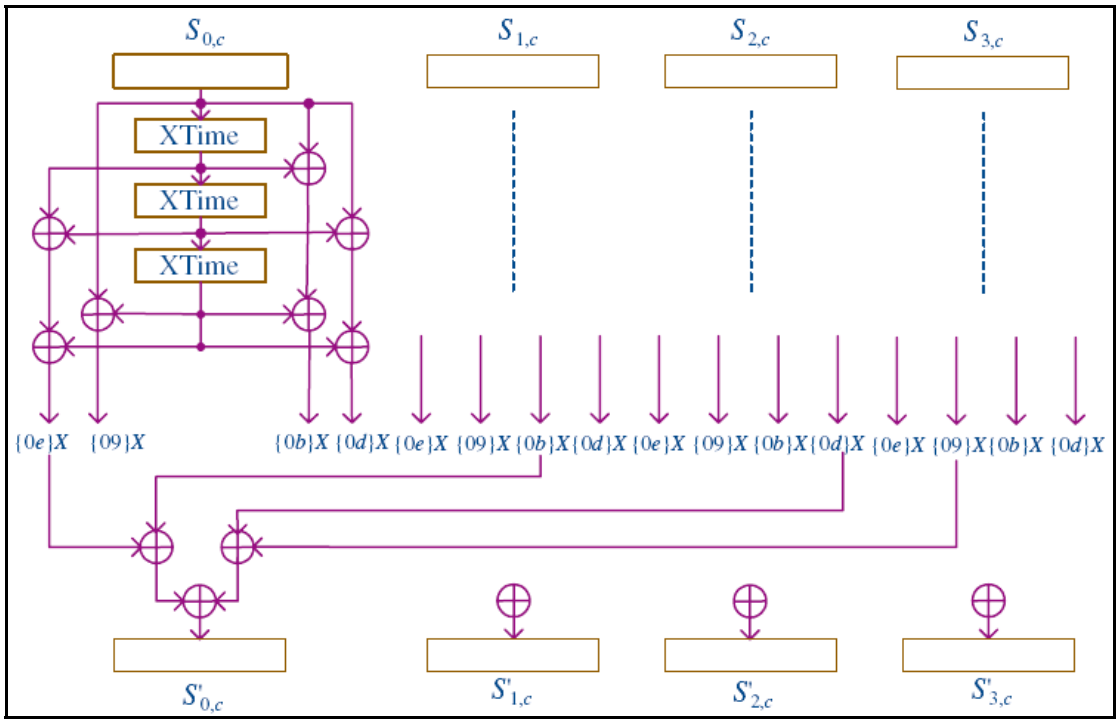


Figure 4–4: Block diagram for straight forward implementation of the InvMixColumns transformation

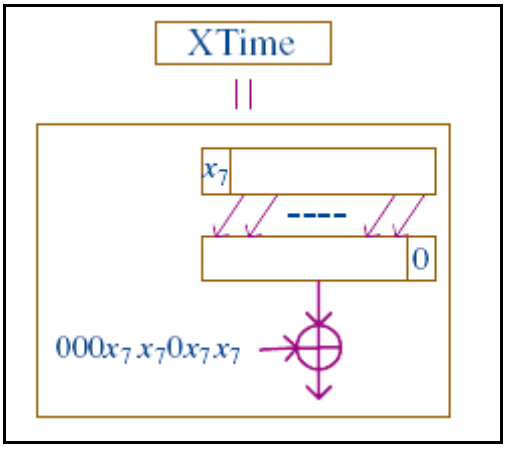


Figure 4–5: Block diagram of Xtime

In the InvMixColumns transformation, the calculation for the other bytes can be carried out similarly according to equation (2.24). As shown in Figure 4–4, the critical path is 6 XOR gates, and a total of (10 x 8 + 3 x 4) x 4 = 368 XOR gates is needed to implement the InvMixColumns transformation for one column of the State.

Studies in [33, 42, 36] have proposed alternative ways to implement the MixColumns/InvMixColumns transformation. Both the studies in [33, 36] exploited the idea of substructure sharing. In [33]'s study, taking the bytes in the first row of the State for example,

$$s'_{0,c} = [\{02\}\ \{03\}\ \{01\}\ \{01\}] \times [s_{0,c}\ s_{1,c}\ s_{2,c}\ s_{3,c}]^T \qquad (4.7)$$

This can be rewritten as

$$S'_{0,c} = \{02\}(S_{0,c} + S_{1,c}) + S_{1,c} + (S_{2,c} + S_{3,c}) \quad (4.8)$$

The above equation can be realized as shown in Figure 3–6.

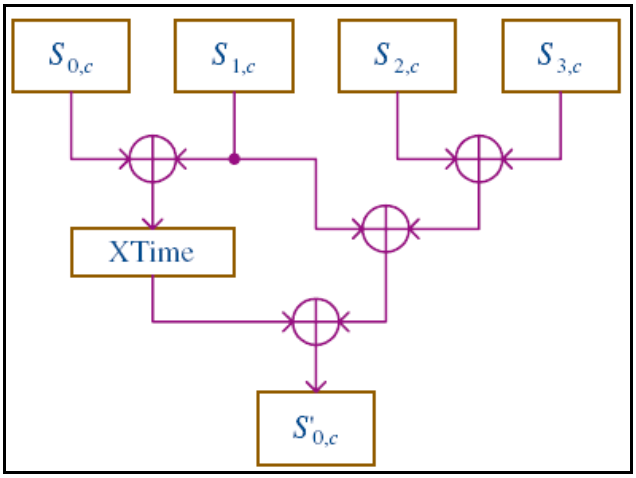


Figure 4–6: Block diagram for substructure sharing implementation of MixColumns transformation

The computation of the other bytes in the State can be implemented by similar structure and the same number of XOR gates. Compared to Figure 4–4, the total number of XOR gates for computing one column of the State remains 144, but the critical path has been reduced to 3 XOR gates.

The same substructure sharing idea can be used in the InvMixColumns transformation. For example, the bytes in row one of the State are calculated

$$s'_{0,c} = [\{0e\}\{0b\}\{09\}\{0d\}] \times [s_{0,c}\ s_{1,c}\ s_{2,c}\ s_{3,c}]^T \quad (4.9)$$

This can be rewritten as

$$\begin{aligned} S'_{0,c} = \{04\}(\{02\}(S_{0,c} + S_{1,c}) + \{02\}(S_{2,c} + S_{3,c}) + (S_{0,c} + S_{2,c})) \\ + \{02\}(S_{0,c} + S_{1,c}) + S_{1,c} + (S_{2,c} + S_{3,c}) \end{aligned} \quad (4.10)$$

Equation (4.10) can be implemented as shown in Figure 4–7. Compared to the straightforward implementation in Figure 4–4 , the number of XOR gates is reduced to (8 x 8 + 4 x 4) x 4 = 320 for computing one column of the State, but the critical path is increased to 7 XOR gates.

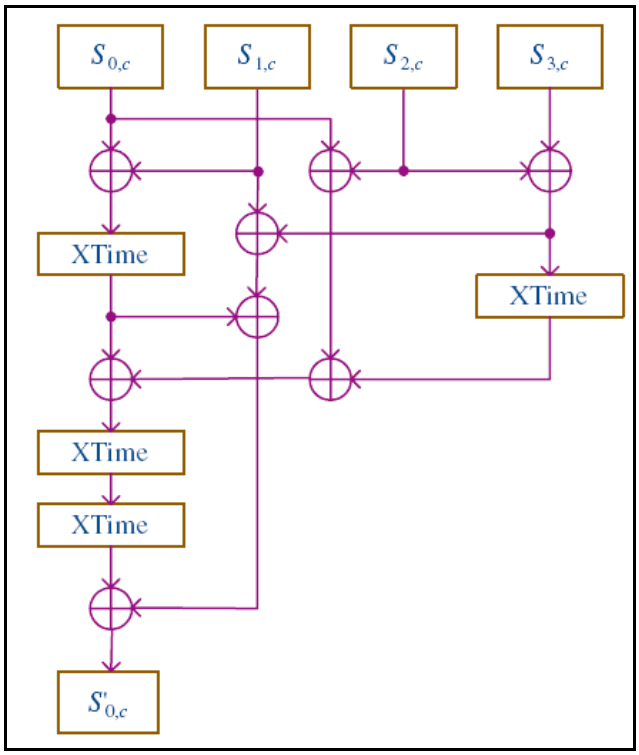


Figure 4–7: Block diagram for substructure sharing implementation of the InvMixColumns transformation

Substructure sharing can be used in another way as in [36]. Since

$$\{0b\}X = (\{08\}X \oplus X) \oplus \{02\}X,$$
$$\{0d\}X = (\{08\}X \oplus X) \oplus \{04\}X,$$
$$\{09\}X = \{08\}X \oplus X,$$
$$\{0e\}X = \{08\}X \oplus \{04\}X \oplus \{02\}X.$$

Three of the above equations have the common factor $\{08\}X \oplus X$. Therefore, hardware usage can be reduced by first calculating the common factor and then using it in the calculation of the other equations. This approach leads to the implementation shown in Figure 4–8. In this figure, the critical path still has 7 XOR gates, but the total number of XOR gates has been reduced further to (8 x 8 + 3 x 4) 4= 304 for calculation of one column of the State.

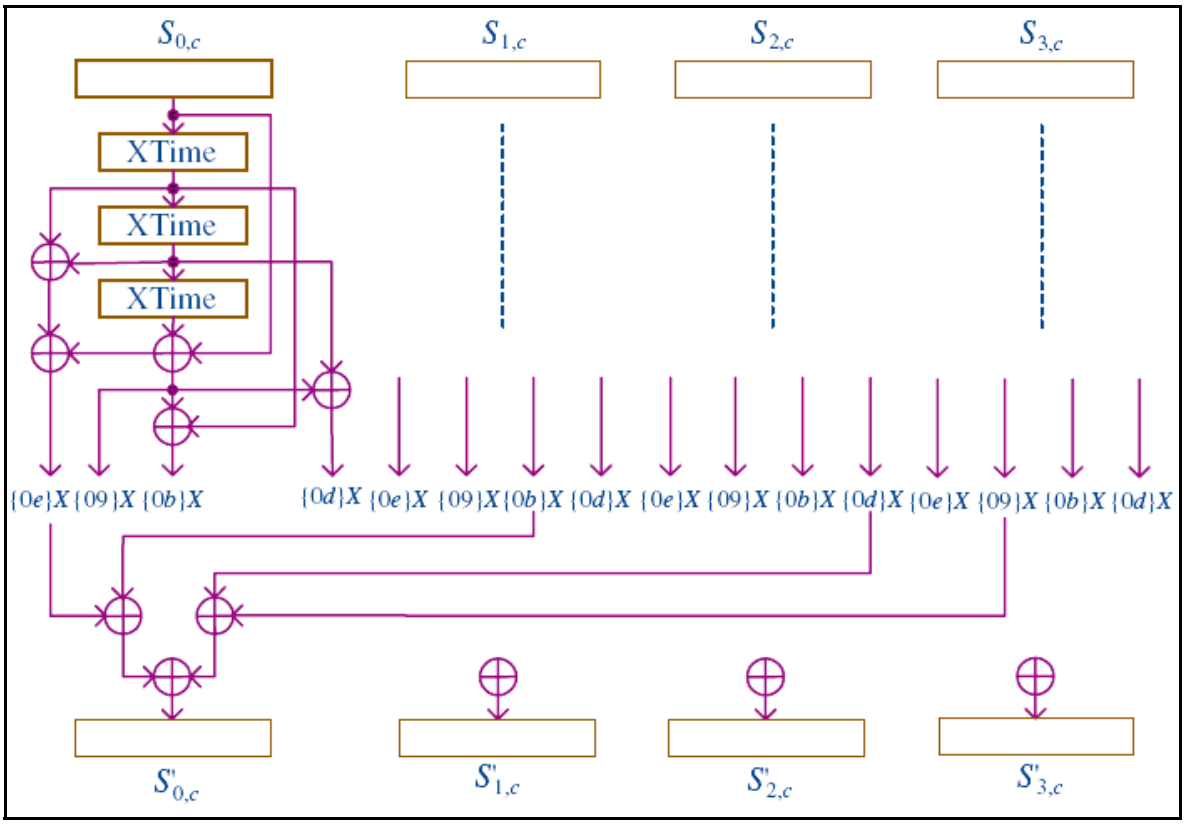


Figure 4–8: Block diagram for alternative substructure sharing implementation of the InvMixColumns transformation.

Another approach in [35], [36] makes use of the polynomial notation of elements in $GF(2^8)$ with irreducible polynomial $m(\alpha)=\alpha^8+\alpha^4+\alpha^3+\alpha+1$. For example, *{03}* can be expressed as polynomial $\alpha+1$,

$$Y = \{03\}X = (\alpha+1)\,(x_7\alpha^7 + x_6\alpha^6 + x_5\alpha^5 + x_4\alpha^4 + x_3\alpha^3 + x_2\alpha^2 + x_1\alpha + x_0) \bmod m(\alpha)$$

Then each bit of Y can be expressed as

$$y_7 = x_7 \oplus x_6, \quad y_6 = x_6 \oplus x_5, \quad y_5 = x_5 \oplus x_4, \quad y_4 = x_7 \oplus x_4 \oplus x_3,$$
$$y_3 = x_7 \oplus x_3 \oplus x_2, \quad y_2 = x_2 \oplus x_1, \quad y_1 = x_7 \oplus x_1 \oplus x_0, \quad y_0 = x_7 \oplus x_0.$$

Similarly, all the constant multiplication used in MixColumns and InvMixColumns transformations can be calculated by the equations in Table 4–1.

Table 4–1: Individual Bit Expression for Constant Multiplications

	{02}X	{03}X	{09}X	{0b}X	{0d}X	{0e}X
y_7	x_6	$x_6 \oplus x_7$	$x_4 \oplus x_7$	$x_4 \oplus x_6 \oplus x_7$	$x_4 \oplus x_5 \oplus x_7$	$x_4 \oplus x_5 \oplus x_6$
y_6	x_5	$x_5 \oplus x_6$	$x_3 \oplus x_6 \oplus x_7$	$x_3 \oplus x_5 \oplus x_6 \oplus x_7$	$x_3 \oplus x_4 \oplus x_6 \oplus x_7$	$x_3 \oplus x_4 \oplus x_5 \oplus x_7$
y_5	x_4	$x_4 \oplus x_5$	$x_2 \oplus x_5 \oplus x_6 \oplus x_7$	$x_2 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7$	$x_2 \oplus x_3 \oplus x_5 \oplus x_6$	$x_2 \oplus x_3 \oplus x_4 \oplus x_6$
y_4	$x_3 \oplus x_7$	$x_3 \oplus x_4 \oplus x_7$	$x_1 \oplus x_4 \oplus x_5 \oplus x_6$	$x_1 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7$	$x_1 \oplus x_2 \oplus x_4 \oplus x_5 \oplus x_7$	$x_1 \oplus x_2 \oplus x_3 \oplus x_5$
y_3	$x_2 \oplus x_7$	$x_2 \oplus x_3 \oplus x_7$	$x_0 \oplus x_3 \oplus x_5 \oplus x_7$	$x_0 \oplus x_2 \oplus x_3 \oplus x_5$	$x_0 \oplus x_1 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_7$	$x_0 \oplus x_1 \oplus x_2 \oplus x_5 \oplus x_6$
y_2	x_1	$x_1 \oplus x_2$	$x_2 \oplus x_6 \oplus x_7$	$x_1 \oplus x_2 \oplus x_6 \oplus x_7$	$x_0 \oplus x_2 \oplus x_6$	$x_0 \oplus x_1 \oplus x_6$
y_1	$x_0 \oplus x_7$	$x_0 \oplus x_1 \oplus x_7$	$x_1 \oplus x_5 \oplus x_6$	$x_0 \oplus x_1 \oplus x_5 \oplus x_6 \oplus x_7$	$x_1 \oplus x_5 \oplus x_7$	$x_0 \oplus x_5$
y_0	x_7	$x_0 \oplus x_7$	$x_0 \oplus x_5$	$x_0 \oplus x_5 \oplus x_7$	$x_0 \oplus x_5 \oplus x_6$	$x_5 \oplus x_6 \oplus x_7$

Studies in [35], [36] did not further investigate the individual bit calculation of constant multiplication. Direct implementation of the equations in Table 4–1 does not bring any area or critical path reduction. However, the idea of substructure sharing can be also applied to the calculation of individual bits. After making modifications to the algorithm in [39], the following algorithm is derived to find the substructures that can be shared in the constant multiplications.

1. round = 0.
2. For i, j = 0 to 7 + round, count the number of times $x_i \oplus x_j$ appears in all the equations, denote the number by $N(i, j)$. Find the biggest number $N(m, n)$. If there is a tie, pick one at random.
3. If $N(m, n) > 1$, then replace $x_m \oplus x_n$ in all those equations with $x_{7 + \text{round}}$, otherwise STOP.
4. round = round + 1, go to step 2.

For example, in the MixColumns transformation, we need to calculate the sixteen equations in the second and third columns in Table 4–1 for each byte in the State, the biggest number of third columns in Table 4–1 for each byte in the State. The biggest number of times $x_i \oplus x_j$ appears is three when (i =0, j = 7) or (i = 3, j = 7). We pick $x_0 \oplus x_7$ randomly and replace $x_0 \oplus x_7$ with x_8 in all equations, and then the second and third columns in Table 4–1 become Table 4–2. In the next round, the biggest number of $x_i \oplus x_j$ in common in all the equations in Table 4–2 is three, when (i = 3, j = 7). Replacing $x_3 \oplus x_7$ with x_9 in Table 4–2, Table 4–3 is derived. In the third round, the biggest $N(i, j)$ is 1, so the algorithm stops and Table 4–3 is the final table. MixColumns can be then using them in the computing of other equations according to Table 4–3.

Figure 4–9 illustrates this implementation.

Table 4–2: Substructure Sharing in Individual Bit Calculation for the MixColumns Transformation after the First Round

	{02}X	{03}X
y_7	x_6	$x_6 \oplus x_7$
y_6	x_5	$x_5 \oplus x_6$
y_5	x_4	$x_4 \oplus x_5$
y_4	$x_3 \oplus x_7$	$x_3 \oplus x_4 \oplus x_7$
y_3	$x_2 \oplus x_7$	$x_2 \oplus x_3 \oplus x_7$
y_2	x_1	$x_1 \oplus x_2$
y_1	x_8	$x_1 \oplus x_8$
y_0	x_7	x_8

Table 4–3: Substructure Sharing in Individual Bit Calculation for the MixColumns Transformation after the Second Round

	{02}X	{03}X
y_7	x_6	$x_6 \oplus x_7$
y_6	x_5	$x_5 \oplus x_6$
y_5	x_4	$x_4 \oplus x_5$
y_4	X_9	$x_4 \oplus x_9$
y_3	$x_2 \oplus x_7$	$x_2 \oplus x_9$
y_2	x_1	$x_1 \oplus x_2$
y_1	x_8	$x_1 \oplus x_8$
y_0	x_7	x_8

Table 4–4: Substructure Sharing in Individual Bit Calculation for the InvMixColumns Transformation

	{09}X	{0b}X	{0d}X	{0e}X
y_7	x_9	$x_6 \oplus x_9$	$x_5 \oplus x_9$	$x_4 \oplus x_8$
y_6	$x_6 \oplus x_{13}$	$x_8 \oplus x_{13}$	$x_6 \oplus x_{17}$	$x_5 \oplus x_{17}$
y_5	$x_7 \oplus x_{18}$	$x_9 \oplus x_{18}$	$x_8 \oplus x_{12}$	$x_4 \oplus x_6 \oplus x_{12}$
y_4	$x_4 \oplus x_{10}$	$x_{10} \oplus x_{17}$	$x_2 \oplus x_9 \oplus x_{15}$	$x_{12} \oplus x_{15}$
y_3	$x_{11} \oplus x_{13}$	$x_{11} \oplus x_{12}$	$x_{13} \oplus x_{14}$	$x_2 \oplus x_{14}$
y_2	x_{19}	$x_1 \oplus x_{19}$	$x_0 \oplus x_{16}$	$x_0 \oplus x_1 \oplus x_6$
y_1	x_{10}	$x_7 \oplus x_{14}$	$x_7 \oplus x_{15}$	x_{11}
y_0	x_{11}	$x_7 \oplus x_{11}$	$x_0 \oplus x_8$	$x_7 \oplus x_8$

Where

$x_8 = x_5 \oplus x_6$	$x_9 = x_4 \oplus x_7$
$x_{10} = x_1 \oplus x_8$	$x_{11} = x_0 \oplus x_5$
$x_12 = x_2 \oplus x_3$	$x_{13} = x_3 \oplus x_7$
$x_{14} = x_0 \oplus x_{10}$	$x_{15} = x_1 \oplus x_5$
$x_{16} = x_2 \oplus x_6$	$x_{17} = x_3 \oplus x_9$
$x_{18} = x_2 \oplus x_8$	$x_{19} = x_7 \oplus x_{16}$

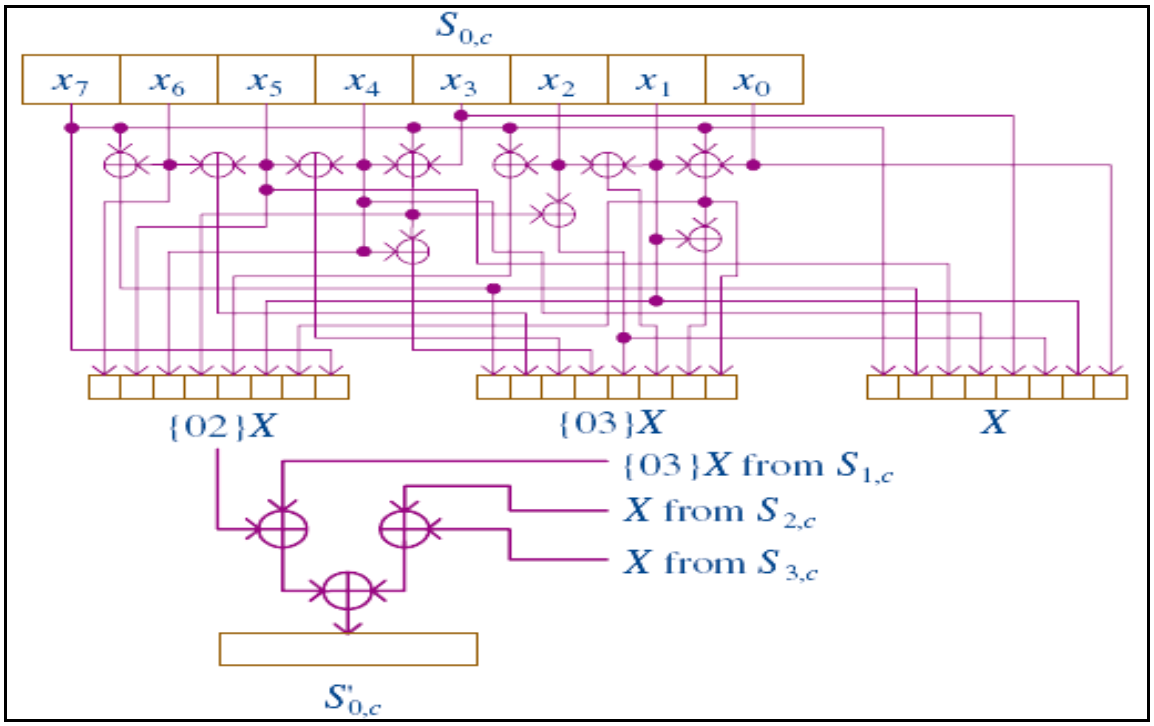


Figure 4–9: Block diagram for bit-wise implementation of the MixColumns transformation

The critical path remains 4 XOR gates as in Figure 3–3, but the total number of XOR gates to calculate on a column of the State has been reduced to 4 x (10 + 3 x 8) = 136.

Applying the same algorithm to the equations in the last four columns in Table 4–1, we get Table 4–4 as the final table for substructure sharing in the InvMixColumns transformation. According to this table, the critical path of the InvMixColumns transformation can only have 6 XOR gates if tree adders are used. At the same time, the total number of XOR gates to calculate one column of the State has been reduced to (30 + 12 + 24) x 4 = 264.

4.2.4. Look-Up Table Implementation of the Whole Round Unit

Look-up tables not only can be used to implement the SubBytes/ InvSubBytes transformation, they can also be used to incorporate MixColumns/InvMixColumns transformation [9, 30, 35, 40]. The T-box approach implements the combination of SubBytes, ShiftRows and MixColumns transformations by look-up tables. Beginning from the SubBytes transformation, the updated State after the MixColumns transformation can be expressed as, for $0 \le c < Nb$,

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} Subbytes(s_{0,c}) \\ Subbytes(s_{1,c+1}) \\ Subbytes(s_{2,c+2}) \\ Subbytes(s_{3,c+3}) \end{bmatrix} \qquad (4.11)$$

Instead of storing only the value of SubBytes($S_{i,j}$) in the *S*-box approach, the *T*-box approach stores values of SubBytes($S_{i,j}$), {02}SubBytes($S_{i,j}$) and {03}SubBytes($S_{i,j}$). Each *T*-box has three 8-bits outputs and can be expressed as

$$T(s_{i,j}) = \begin{bmatrix} T_1(s_{i,j}) \\ T_2(s_{i,j}) \\ T_3(s_{i,j}) \end{bmatrix} = \begin{bmatrix} SubBytes(s_{i,j}) \\ \{02\}SubBytes(s_{i,j}) \\ \{03\}SubBytes(s_{i,j}) \end{bmatrix}. \qquad for\ 0 \le i, j < 4. \qquad (4.12)$$

Now equation (4.11) can be rewritten as

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} T_2(s_{0,c}) \oplus T_3(s_{1,c+1}) \oplus T_1(s_{2,c+2}) \oplus T_1(s_{3,c+3}) \\ T_1(s_{0,c}) \oplus T_2(s_{1,c+1}) \oplus T_3(s_{2,c+2}) \oplus T_1(s_{3,c+3}) \\ T_1(s_{0,c}) \oplus T_1(s_{1,c+1}) \oplus T_2(s_{2,c+2}) \oplus T_3(s_{3,c+3}) \\ T_3(s_{0,c}) \oplus T_1(s_{1,c+1}) \oplus T_1(s_{2,c+2}) \oplus T_2(s_{3,c+3}) \end{bmatrix} \qquad 0 \le c < Nb \qquad (4.13)$$

The combination of SubBytes, ShiftRows and MixColumns transformations can be implemented by XORing the outputs of *T*-boxes. In the final round of encryption, there is no MixColumns transformation, so S-box instead of *T*-box should be used. In the fully pipelined or fully loop unrolled architecture, this will not be a problem, since each round uses separate hard- ware. However, for other architectures in which one round unit is used to perform different rounds of encryption in different clock cycles, *T*-box cannot be simply replaced by *S*-box. Adding an additional *S*-box is a solution, but this will lead to extra area for look-up tables. Another solution is to extract *S*-box from *T*-box; *S*-box is exactly the T_1 output of a *T*-box [35]. The *T*-box implementation has shorter delay than the S-box approach. The delay of MixColumns is eliminated by adding a delay of 2 XOR gates if a tree adder is used to add up the four items in each row of the matrix on the right side of [40]. Based on the same technology and assumptions, the *T*-box approach improves the speed of an encryptor from 7 Gbit/sec in [41] to 12 Gbit/sec in [31]. However, the price paid for shorter delay is the three-times-bigger look-up tables.

Correspondingly, T^{-1}-box can be used to implement the combination of InvSubBytes, InvShiftRows and InvMixColumns transformations. From the beginning of InvSubBytes transformation, the updated State after the InvMixColumns transformation can be expressed as

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} InvSubbytes(s_{0,c}) \\ InvSubbytes(s_{1,c+1}) \\ InvSubbytes(s_{2,c+2}) \\ InvSubbytes(s_{3,c+3}) \end{bmatrix} \qquad (4.14)$$

For $0 \le c < Nb$. Each T^{-1}-box stores four sets of values:

$$\{09\}InvSubbytes(s_{i,j}), \qquad \{0b\}InvSubBytes(s_{i,j}),$$
$$\{0d\}InvSubbytes(s_{i,j}), \qquad \{0e\}InvSubBytes(s_{i,j}),$$

Unlike T-box, each T^{-1}-box has four 8-bit outputs and is four times the size of an S-box, and can be expressed as

$$T^{-1}(s_{i,j}) = \begin{bmatrix} T_0^{-1}(s_{i,j}) \\ T_1^{-1}(s_{i,j}) \\ T_2^{-1}(s_{i,j}) \\ T_3^{-1}(s_{i,j}) \end{bmatrix} = \begin{bmatrix} \{09\}InvSubBytes(s_{i,j}) \\ \{0b\}InvSubBytes(s_{i,j}) \\ \{0d\}InvSubBytes(s_{i,j}) \\ \{0e\}InvSubBytes(s_{i,j}) \end{bmatrix}. \qquad for\ 0 \le i,j < 4. \qquad (4.15)$$

Now equation (4.14) can be rewritten as

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} T^{-1}{}_3(s_{0,c}) \oplus T_1^{-1}(s_{1,c+1}) \oplus T_2^{-1}(s_{2,c+2}) \oplus T_0^{-1}(s_{3,c+3}) \\ T^{-1}{}_0(s_{0,c}) \oplus T_3^{-1}(s_{1,c+1}) \oplus T_1^{-1}(s_{2,c+2}) \oplus T_2^{-1}(s_{3,c+3}) \\ T^{-1}{}_2(s_{0,c}) \oplus T_0^{-1}(s_{1,c+1}) \oplus T_3^{-1}(s_{2,c+2}) \oplus T_1^{-1}(s_{3,c+3}) \\ T^{-1}{}_1(s_{0,c}) \oplus T_2^{-1}(s_{1,c+1}) \oplus T_0^{-1}(s_{2,c+2}) \oplus T_3^{-1}(s_{3,c+3}) \end{bmatrix} \quad 0 \le c < Nb \qquad (4.16)$$

Similar to the encryption case, S^{-1}-box needs to be used in the final round, but none of the outputs of T^{-1}-box is InvSubBytes($S_{i,j}$) this time. [35] brought up an efficient method to calculate InvSubBytes($S_{i,j}$) from theoutput of T^{-1}-box. For any specific $S_{i,j}$, each output byte of a T^{-1}-box can be expressed in binary form as $T^{-1}(S\ ) = [t_{m7}, t_{m6}, t_{m5}, t_{m4}, t_{m3}, t_{m2}, t_{m1}, t_{m0}]$ (m = 0,1, 2, 3), and InvSubBytes($S_{i,j}$) can be expressed in binary form as InvSubBytes($S_{i,j}$) = $[s_7^{-1}, s_6^{-1}, s_5^{-1}, s_4^{-1}, s_3^{-1}, s_2^{-1}, s_1^{-1}, s_0^{-1}]$.

Since $\{09\}^{-1} = \{4f\}, \{0b\}^{-1} = \{c0\}$, $\{0d\}^{-1} = \{e1\}$, and $\{0e\}^{-1} = \{e5\}$,

$$\begin{aligned} InvSubBytes(s_{i,j}) &= \{4f\}T_0^{-1}(s_{i,j}) = \{c0\}T_1^{-1}(s_{i,j}) \\ &= \{e1\}T_2^{-1}(s_{i,j}) = \{e5\}T_3^{-1}(s_{i,j}) \end{aligned} \qquad (4.17)$$

Each of the 8 bits of InvSubBytes ($S_{i,j}$) can be computed as functions of individual bits in $T^{-1}(S)$. Four sets of expressions of s_n ($0 \le n < 8$) can be derived from equation (4.17). Expressions with the shortest delay are chosen for each s_n^{-1} from the four sets as shown in Table 4–5. From Table V, at most two XOR gates are needed to compute each bit of InvSubBytes($S_{i,j}$) from T^{-1}-box. The critical path of the final round in T^{-1}-box approach consists of a look-up table, 2 XOR gates to extract the value of InvSubBytes($S_{i,j}$) and another XOR gate to add up the roundkey. The critical path of other round units includes a look-up table, 2 XOR gates to add up four outputs from different T^1 -

boxes according to equation (4.14), using adder tree structure, and another XOR gate to add the roundkeys. We can observe that the delays of each round in a T^{-1}-box approach are the same, and equal the total delay of a look-up table and 3 XOR gates. Compared to the total delay of a look-up table and at least 6 XOR gates in the S^{-1}-box approaches, this approach has shorter delay but the price paid for that is the requirement of 4- times-bigger look-up tables of an S^{-1}-box approach, which makes pipelining or loop unrolling more expensive.

Table 4–5: Extraction of S^{-1}-box from T^{-1}-box

$s_7^{-1} = t_{07} \oplus t_{04} \oplus t_{01}$	$s_6^{-1} = t_{06} \oplus t_{03} \oplus t_{00}$
$s_5^{-1} = t_{05} \oplus t_{02}$	$s_4^{-1} = t_{04} \oplus t_{01}$
$s_3^{-1} = t_{13} \oplus t_{12} \oplus t_{11}$	$s_2^{-1} = t_{36} \oplus t_{35} \oplus t_{30}$
$s_1^{-1} = t_{37} \oplus t_{35} \oplus t_{34}$	$s_0^{-1} = t_{15} \oplus t_{12} \oplus t_{10}$

4.2.5. Implementation of Key Expansion

Roundkeys can either be generated beforehand and stored in memory or be generated on the fly. The former case is suitable for the applications which do not change keys constantly and can afford large area for memory.

During encryption/decryption, roundkeys can be read out from memory by appropriate address, and there is no extra delay for decryption. In this case, reducing the critical path of Key Expansion can reduce the overhead, but will not speed up the whole system. While in the applications which need to change keys constantly, expanding keys on the fly is preferred. From Figure 2–12, we can observe that the critical path of Key Expansion consists of one multiplexer, one *S*-box, and one XOR gate. Since the critical path of Key Expansion is shorter than that of a round unit, reducing the critical path of Key Expansion will not increase the speed of the whole system. Generating roundkeys on the fly eliminates the requirement for key storage, but brings overhead for decryption since decryption can only begin after the last roundkey is generated.

4.3. Joint Implementation Issues of Encryptor/ Decryptor

Source sharing becomes important when only small area is available for implementing both encryptor and decryptor, as in smart cards and cellular phones. While the algorithmic strength in the last section can be exploited to reduce area, the design could be improved further by sharing the resources between encryptor and decryptor.

4.3.1. Joint Implementation of SubBytes and InvSubBytes

In [10], each *S*-box/S^{-1}-box requires a 2k-bit look-up table, and each round unit needs 32 such look-up tables to implement both encryption and decryption. However, studies in [9, 16] proposed that the SubBytes and InvSubBytes transformation can share a 2k-

bit look-up table for each byte in the State. The SubBytes transformation can be expressed as

$$S' = M\ S^{-1} + c \tag{4.18}$$

Where

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

And $c = [0\ 1\ 1\ 0\ 0\ 0\ 1\ 1]$.

The inverse of (18) is given by

$$S_{i,j} = (M^{-1}(S'_{i,j} + c))^{-1} \tag{4.19}$$

From equations (4.18) and (4.19), the SubBytes and InvSubBytes transformations can share look-up tables which only implement multiplicative-inverse in $GF(2^8)$. Figure 4–10 illustrates the block diagram for a joint SubBytes and InvSubBytes transformation [33]. The Joint *S*-box block is a look-up table which stores the value of multiplicative inverse, while the two rectangular blocks in the top implement the corresponding matrix multiplication and addition. The signal 'enc' is '1' when it's in encryption mode, and is '0' otherwise.

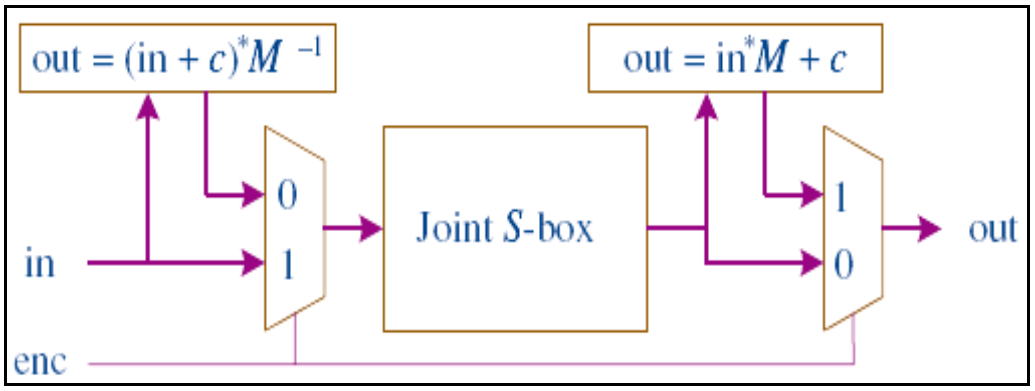


Figure 4–10: Joint implementation of the SubBytes and the InvSubBytes transformations

Since both M and M^{-1} are binary matrices, the matrix multiplication block can be implemented simply by XOR gates. Another approach is to store the value of *S*- box and S^{-1}-box in two separate ROMs, and read the initial values into RAMs at the beginning of encryption/decryption [31]. This approach eliminates the duplicated

memory by 2 additional ROMs, but introduces an over- head of 256 clock cycles to read in the initial values.

4.3.2. Resource Sharing in MixColumns and InvMixColumns

Although equation (4.10) leads to an InvMixColumns implementation with neither the shortest delay nor the smallest area, combined with equation (4.8), it leads to the hardware implementation with least area of joint MixColumns/ InvMixColumns transformation. Figure 4–11 illustrates the diagram according to equations (4.8) and (4.10). The four inputs, '*a*', '*b*', '*c*', '*d*', and two outputs, 'mix' and 'invmix', all represent single bytes. '*a*', '*b*', '*c*', '*d*' are the four bytes in a column of the State with ascending row numbers. 'Mix' and 'invmix' are the outcomes of applying MixColumns and InvMixColumns transformation to the inputs, respectively. The block diagram for applying MixColumns and InvMixColumns to the bytes in other rows can be obtained by exchanging the position of the input bytes according to equations (2.22) and (2.24).

Figure 4–12 shows the diagram of applying MixColumns and InvMixColumns transformations to one column of the State. Each of the JointMix blocks consists of the diagram in Figure 4–11. 'Mixword' is the output of applying the MixColumns transformation to the 'inword', and 'invword' is the output of applying the InvMixColumns transformation to the 'inword'. The 'outword' gets the value of 'mixword' when 'enc' = '1', which indicates encryption mode, and gets the value of 'invword' otherwise.

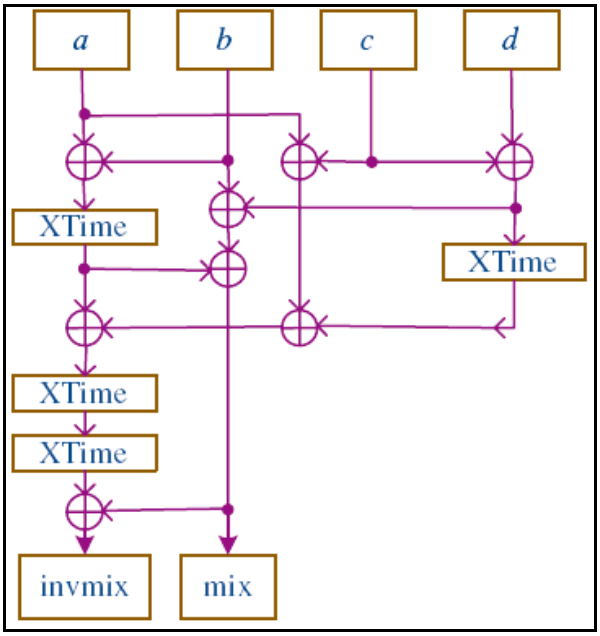


Figure 4–11: Joint implementation of the MixColumns and the InvMixColumns transformations (bytes in the first row of the State)

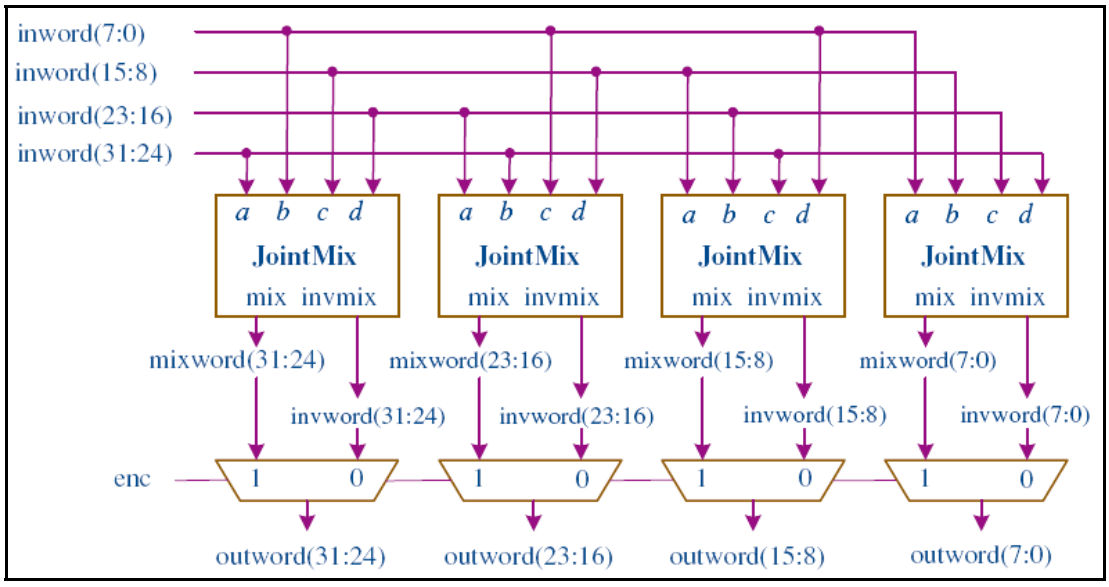


Figure 4–12: Joint implementation of the MixColumns and the InvMixColumns transformations (one column in the State)

4.3.3. Resource Sharing of Generating Roundkeys in Encryption and Decryption

In the applications with limited area, generating roundkeys on the fly is a better choice. Paper [33] proposed an efficient architecture which can generate roundkey(i + 1) from roundkey(i) and *vice versa*. This architecture is illustrated in Figure 4–13.

In Figure 4–13, each of the 'roundkey' and the 'next-roundkey' consist of four words. Assuming the 'roundkey' is expressed by four words as (w_{4i}, w_{4i+1}, w_{4i+2}, w_{4i+3}), the 4 sets of XOR gates from left to right get w_{4i}, w_{4i+1}, w_{4i+2}, and w_{4i+3} as one of the inputs from the 'roundkey' bus, respectively. The RotSub block in Figure 4–13, which performs RotWord followed by SubBytes transformation, is made up of 4 *S*-boxes. Since all the lower bytes of the round constant Rcon are zeros, the step of adding round constant only needs to be performed on the most significant byte. Meanwhile, *Rcon(i+1)* equals {02}Rcon(i) *($1 \le i < Nr - 1$).*

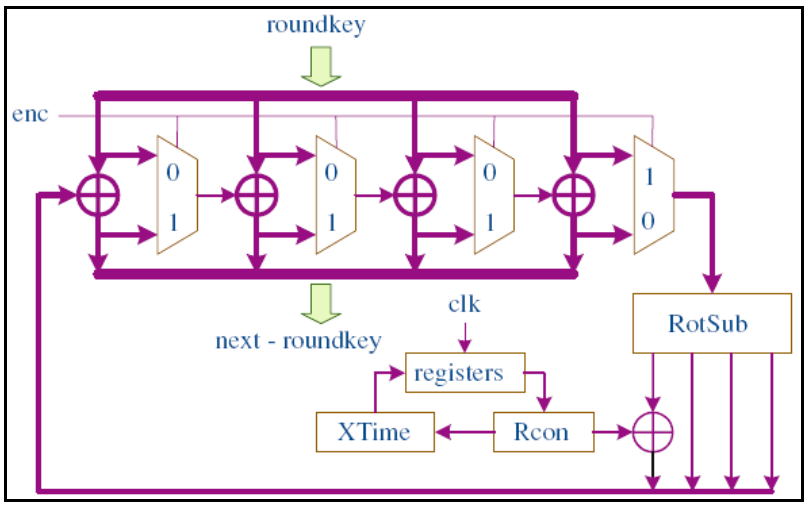


Figure 4–13: Joint implementation of Key Expansion in encryptor and decryptor.

Rcon(i+1) can also be generated on the fly from the stored *Rcon(1) = {01}*. When 'enc' is '1', which stands for encryption mode, w_{4i+3} is loaded into the RotSub block, after the output of RotSub block was XORed with *Rcon(i)* and w_{4i}, $w_{4(i+1)}$ is generated at the output of the first XOR gate on the left. Consequently, $w_{4(i+1)+1}$, $w_{4(i+1)+2}$, and $w_{4(i+1)+3}$ are generated one by one as the updated data propagates through each multiplexer from left to right. When 'enc' is '0', $w_{4i+3} \oplus w_{4i+2} = w_{4(i-1)+3}$ $(0 < i \leq Nr)$ is loaded into the RotSub block, after the output of RotSub is XORed with Rcon(i),a temporary value temp = Sub- Word(RotWord($w_{4(i-1)+3}$)) $\oplus$ Rcon(i) is fed back to the leftmost XOR gate in Figure 4–13. $w_{4(i-1)} = w_{4i} \oplus$ temp is generated after an XOR gate delay. As the updated data propagates through each of the multiplexers, $w_{4(i-1)+1}$, $w_{4(i-1)+2}$, and $w_{4(i-1)+3}$ are generated in a sequence.

Chapter 5

SUBBYTES TRANSFORMATION OPTIM IZATION METHODS

Modern symmetric ciphers require non-linear functions in order to defend against linear cryptanalysis. Substitution is a popular function for introducing non-linearity. A substitution function is commonly referred to as S-box and can be defined on basis of arithmetic operations or as an arbitrary mapping. Different cipher algorithms also use different numbers of S-boxes, e.g. DES uses eight S-boxes which map six to four bits, while AES uses a single S-box which is a bijective mapping from eight to eight bits.

The AES algorithm makes use of its S-box in the SubBytes round transformation as well as in the key expansion. From a mathematical point of view, the AES S-box is defined as an inversion in the finite field $GF(2^8)$ with a specific reduction polynomial, followed by an affine transformation. The inverse S-box, which is required for the InvSubBytes round transformation for decryption, is simply the inverse of the affine transformation, followed by an inversion in $GF(2^8)$ as described in section 2.5.1. The finite field inversion is the only non-linear operation of the AES algorithm. Since there are many design options for the S-box in hardware, it is challenging to find an optimal implementation for a particular purpose. On the one hand, the main criterion for high-speed implementations is a short critical path, which allows reaching high clock frequencies. On the other hand, S-box implementations for embedded devices call for small silicon area and low power consumption. Several hardware implementations of the nonlinear step are possible, and there is no evident best general solution. We could exploit the particular features of the ASIC technology library or FPGA platform to limit area occupation for the S-box component, which is critical to the success of the implementation.

Because the S-box is based on an operation of inversion in the finite field $GF(2^8)$, we can propose different architectures. A broad classification divides all the possible implementations in two main categories: serial architectures and parallel architectures. While it is true that serial architectures could lead to compact circuits, in the following, we will focus on parallel ones, assuming that the dedicated round logic computes one round of the algorithm in one clock cycle with 16 S-boxes instantiated, one per each State byte. Even if this is not the case, we can pipeline a completely combinatorial implementation by inserting registers to obtain an enhanced throughput, multi-cycle architecture. Section 4.1 shows that such pipelined combinatorial architectures tend to be very efficient in terms of both area requirements and delay when mapped to commercial FPGA units.

In many AES implementations two sub-steps required in the SubBytes transformation are typically combined into a single lookup table. The table size is 16 by 16 with the content 8 bits in length. The ROM size of 256×8 bit is not big for current technology and can be implemented in a fairly simple manner with modern design tools. The ROM design will achieve a high speed S-box and we will use it in the design of the optimized speed

AES. However when the area is restricted or a ROM cannot be incorporated, the inversion hardware becomes necessary. Within this scenario, the efficient S-box implementation is the major concern. The affine transformation however requires small number of gates and introduces small delays.

Several techniques for S-box computation have been developed. These are, for instances:

1) The mentioned above table look up where step 2 is usually combined to be a single table.
2) Synthesis and optimized logic function of S-box using CAD tools.
3) Compute the inversion of element in $GF(2^8)$ and optimize the logic functions.

In the computation of element inversion in $GF(2^k)$ one can use either extended Euclid algorithm [43, 44] or composite field technique [38, 45, 46, 47, 48]. The use of composite field in the S-box computation has been reported in literatures [38, 47, 48]. Rudra *et al.* [47, 49] mapped all the operation (except ShiftRows) into the composite field of $GF(2^4)^2$. Multiplication, squaring and inversion are borrowed form those detailed in [50]. Morioka and Satoh [48] also have exploited the used of composite field in the design of a low power S-box transform. Elements in $GF(2^8)$ are mapped to those defined in $GF((2^2)^2)^2$. Multiplication and inversion are optimized in the ground field. Rijnmen [38] also mentioned the computation of inversion in $GF(2^4)$.

In our design for optimized speed AES we will use the ROM design for the S-box which is considered the most efficient approach and achieves the least delay with comparison with other designs. In this chapter we will explain an area efficient approach which we will use in the design of the S-box for optimized area AES. This approach is cited in the paper An Architecture for S-Box Computation in the AES [51]. This approach depends on the mapping of the Elements in $GF(2^8)$ to those defined in $GF((2^2)^2)^2$. Multiplication and inversion are optimized in the ground field.

5.1. An Efficient S-box Computation

In the case of Rijndael, the field polynomial $m(x)= x^8+x^4+x^3+x+1$ has been chosen. It is an irreducible polynomial in $GF(2^8)$ but not a primitive one. We first have to map elements in $GF(2^k)$ into $GF(2^n)^m$ where $k=mn$. A method elaborated in [47] is again summarized here.

1. Let α be a primitive element of $GF(2^m)^n$, and γ be a primitive element of $GF(2^k)$, such that field isomorphism holds.
2. Map α^i to γ^i for $i \in \{0,...,(2^k-1)\}$.
3. Check whether $\forall\, i \in \{0,...,(2^k-1)\}$, if $\alpha^r=\alpha^i+1$ then $\gamma^r=\gamma^i+1$. If so, we then have the required mapping, otherwise we have to search for the next primitive element.
4. The inverse mapping can be easily found by matrix inversion, i.e., if [T] is a mapping matrix, $[T]^{-1}$ is an inverse mapping matrix.

With above procedure, we select the polynomial $p(x)= x^2+x+\beta^{14}$ where β^{14} denotes the element in $GF(2^4)$ of which $I(x)=x^4+x+1$ is the primitive irreducible polynomial. As a

result, the transform matrices that map an element in $GF(2^8)$ to the corresponding element in the composite field $GF(2^4)^2$ or vice versa are obtained as follow.

$$T = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (5.1)$$

And

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad (5.2)$$

The upper–left element in the above matrices denotes the least significant bit. An Advantage of mapping elements form $GF(2^8)$ to $GF((2^4)^2)$ is the simpler multiplicative inverse computation since inversion is performed in $GF(2^4)$. For such a small field size, inversion using either the direct truth-table mapping or table look up consumes small area. Moreover, in Rijndael system data are treated naturally in byte format. Let data (byte) be expressed as

$A = \{bc\} = bx + c$, the inversion of A , say

$B = A^{-1} = \{pq\} = px + q$. For the field polynomial

$p(x) = x^2 + Cx + D$, One can have

$$p = b\Delta^{-1} \quad (5.3)$$

$$q = (Cb \oplus c)\Delta^{-1} \quad (5.4)$$

Where

$$\Delta = c(Cb \oplus c) \oplus b^2 D \quad (5.5)$$

Or

$$\Delta = bcC \oplus c^2 \oplus b^2 D \quad (5.6)$$

For $GF((2^n)^2)$, the polynomial in the form of $p(x) = x^2 + x + \lambda$ always exists [45]. As such, C and D can be set to {1} and {9} (in $GF(2^4)$) respectively. Fixed-coefficient multiplication (i.e., b^2D) as well as squaring units is relatively simple according to their small field size. The multiplications required in computing equations (5.3), (5.4) and (5.5) can be done straight away in $GF(2^4)$ or can be further simplified by making use of composite field $GF((2^2)^2)$ [48]. Borrowed form [45], in our implementation, we use

polynomials $p(x)=x^2+x+\sigma^2$ and $I(x)= x^2 +x+1$ for the computations in $GF((2^2)^2)$ and $GF(2^2)$ respectively.

In $GF((2^2)^2)$, let $U(x)=u_0+u_1x$ and $V(x)=v_0+v_1x$, then

$$Z(x)=U(x)V(x)_{\bmod p(x)}=z_0+z_1x \tag{5.7}$$

Where

$$z_0=u_0v_0\oplus\sigma^2\ u_1v_1 \tag{5.8}$$

$$z_1=u_0v_1\oplus u_1v_0\oplus u_1v_1 \tag{5.9}$$

Where $u_i, v_i, \sigma^2 \in GF(2^2)$.

In the (forward) SubBytes transformation, the inversion is followed by the affine transformation given previously in Section 2.5.1. This step can be combined with the inverse mapping and a single logic block is obtained. The resulted matrix is noted in equation (5.10). Regardless of the hardware reusable, the resulted matrix cannot be shared by the inverse SubBytes transformation.

$$T^{-1}D=\begin{bmatrix}0&1&0&0&1&0&0&1\\1&0&0&1&1&0&1&1\\0&1&1&1&1&1&0&1\\0&1&0&1&1&1&0&0\\1&1&1&1&0&0&0&0\\1&0&0&0&0&0&0&0\\1&0&0&0&1&0&0&0\\0&0&0&1&1&1&1&1\end{bmatrix} \tag{5.10}$$

In the decryption process, the inverse affine transform can be expressed as:

$$b(x)=\{4A\}d(x)_{\bmod(x^8+1)}\oplus c(x) \tag{5.11}$$

Where $c(x)=\{05\}=x^2+1$. This process has to be performed in prior to the (inverse) SubBytes transformation. Similarly, the affine transformation can be merged with the (forward) mapping. The resulted matrix noted in equation (5.12) is obtained. The combined matrices given in equations (5.10) and (5.12) are individual. The combined scheme can result in a slightly compact hardware but not applicable to the restricted hardware size application (such as in smart card) where a single inversion circuit is utilized by the ciphering and the deciphering procedures.

$$TB=\begin{bmatrix}0&1&0&0&0&1&0&0\\0&0&1&1&0&1&1&0\\0&1&0&1&0&1&0&0\\0&0&0&0&0&1&0&1\\1&1&1&0&1&1&1&1\\0&0&0&1&1&1&1&0\\1&0&0&0&1&1&1&0\\0&1&1&0&0&0&1&1\end{bmatrix} \tag{5.12}$$

5.2. Bit-parallel architecture of standard and composite field operations

In this section we will illustrate the bit architecture for the operations described in Section 5.1 [51].

5.2.1. GF(2^8) Computations

1. Inversion in $GF((2^4)^2)$; $p(x)=x^2+x+\beta^{14}$ b,c,p,q are elements in $GF(2^4)$
   $A(x)=bx+c$; $B(x)=A^{-1}(x)=px+q$
   $p=b\Delta^{-1}$, $q=(b \oplus c)\Delta^{-1}$; $\Delta=\beta^{14}b^2 \oplus bc \oplus c^2$.

5.2.2. GF(2^4) Computations

1. Multiplication in $GF(2^4)$, $I(x)=x^4+x+1$ To compute $C(x)= A(x)B(x)_{mod\ I(x)}$ where
   $a_i, b_i, c_i \in GF(2^4)$
   $c_0 = a_0b_0 \oplus a_3b_1 \oplus a_2b_2 \oplus a_1b_3$
   $c_1 = a_1b_0 \oplus a_0b_1 \oplus a_3b_1 \oplus a_3b_2 \oplus a_2b_2 \oplus a_2b_3 \oplus a_1b_3$
   $= a_1b_0 \oplus (a_0 \oplus a_3)b_1 \oplus (a_2 \oplus a_3)b_2 \oplus (a_1 \oplus a_2)b_3$
   $c_2 = a_2b_0 \oplus a_1b_1 \oplus a_0b_2 \oplus a_3b_2 \oplus a_3b_3 \oplus a_2b_3$
   $= a_2b_0 \oplus a_1b_1 \oplus (a_0 \oplus a_3)b_2 \oplus (a_2 \oplus a_3)b_3$
   $c_3 = a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3 \oplus a_3b_3$
   $= a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus (a_0 \oplus a_3)b_3$
2. Multiplication in the composite field $GF((2^2)^2)$,where $p(x)=x^2+x+\sigma^2$
   $U(x) = u_0+ u_1x$, $V(x) = v_0+ v_1x$ and $Z(x) =U(x)V(x)_{mod\ p(x)} = z_0+z_1x$, where $z_i, u_i, v_i \in GF(2^2)$.
   $z_0=u_0v_0 \oplus \sigma^2 u_1v_1$ *and* $z_1=u_0v_1 \oplus u_1v_0 \oplus u_1v_1$
3. Squaring; $B(x) = A(x)A(x)_{mod\ I(x)}$, $I(x)=x4+x+1$
   $b0=(a0 \oplus a2)$, $b1=a2, b2=(a1 \oplus a3), b3=a3$
4. Fixed-coefficient multiplication; To compute $C(x) = \beta^{14}B(x)_{mod\ I(x)}$ where
   $I(x) =x^4+ x +1$ $c0=(b0 \oplus b1), c1=b2, c2=b3, c3=b0$
5. To compute $C(x) = A^2(x)\beta^{14} mod\ I(x)$; Combine 3) and 4) above
   c0=$a0$, $c1=(a1 \oplus a3), c2=a3, c3=(a0 \oplus a2)$
6. Inversion; $C(x) = A^{-1}(x)$, $I(x)=x^4+x+1$
   $c_0 = a_2 \oplus a_3 \oplus \bar{a}_2(a_0 \oplus a_1) \oplus a_1a_2(a_0 \oplus a_3)$
   $c_1 = a_0(a_1 \oplus a_2) \oplus a_1(a_2 \oplus a_3) \oplus a_3(\bar{a}_0 \oplus \bar{a}_1)$
   $c_2 = a_0a_2a_3 \oplus a_0a_1 \oplus \bar{a}_0(a_2 \oplus a_3)$
   $c_3 = a_1 \oplus \bar{a}_0(a_2 \oplus a_3) \oplus a_1(a_0a_2 \oplus a_3)$

5.2.3. GF(2^2) Computations, I(x) = x^2 + x + 1

1. Multiplication in $GF(2^2)$ To compute $F(x) = g(x)h(x)_{mod\ I(x)}$

$$f_0 = g_0h_0 \oplus g_1h_1$$
$$f_1 = g_0h_1 \oplus g_1h_0 \oplus g_1h_1$$

2. Fixed-coefficient multiplication;
   $Z(x) = \sigma^2 U(x)_{mod\ I(x)}$ $\quad z_0=(u_0 \oplus u_1), z_1=u_0$
3. Squaring; $C(x) = A^2(x)_{mod\ I(x)}$
   $c0=(a0 \oplus a1), c1=a1$
4. Inversion; $C(x) = A^{-1}(x)$
   $c0=(a0 \oplus a1), c1=a1$

5.3. IMPLEMENTATIONS

The implementation of the SubBytes transformation formulated earlier in section 5.1 is detailed in this section.

Gate count and delay time of the designs with different architectures are investigated. Many related mathematic equations are given as appendix for convenience of referring. The inversion of elements in the S-box operation has been reported of its most power consuming compared to others (i.e., 75% [48]). We thus look at this operation in quite details. The S-box computation can be divided into 3 blocks as shown in Figure 5–1-a. The affine transform and the inverse mapping could be combined as mentioned earlier.

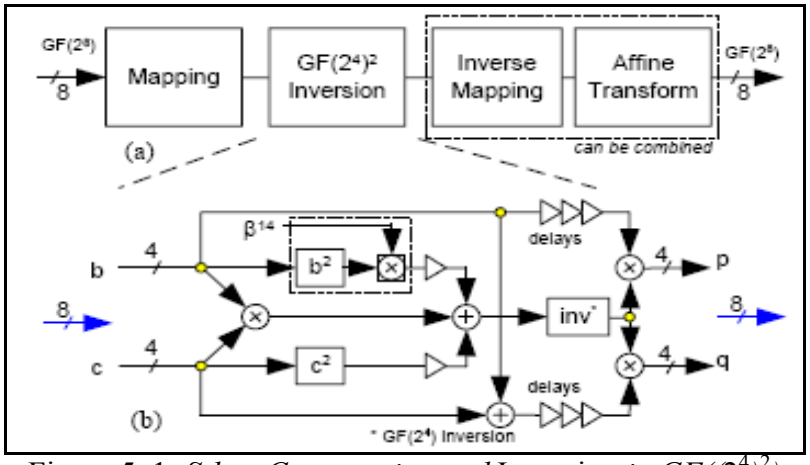


Figure 5–1: *S-box Computation and* Inversion *in* $GF((2^4)^2)$

Shown in Figure 5–1-b, the inversion in $GF(2^8)$ are performed in the composite field, $GF((2^4)^2)$. Three GP multiplications, two squaring, a fixed-coefficient multiplication and an inversion are involved. There are slight differences of the implementation of Figure 5–1. One can choose either equation (5.5) or (5.6). The differences are gate count, wiring complexity and critical data path. We chose (5.6) because of its smaller delays compared to that offered by (5.5). It also should be noted that the operation $b^2\beta^{14}$ could be combined into a single logic block (see Section 5.2.2).

A direct implementation of a general purpose multiplier (see Section 5.2.2) can result in a complexity of 16 AND and 10 OR, and with the delay time of 3τ.

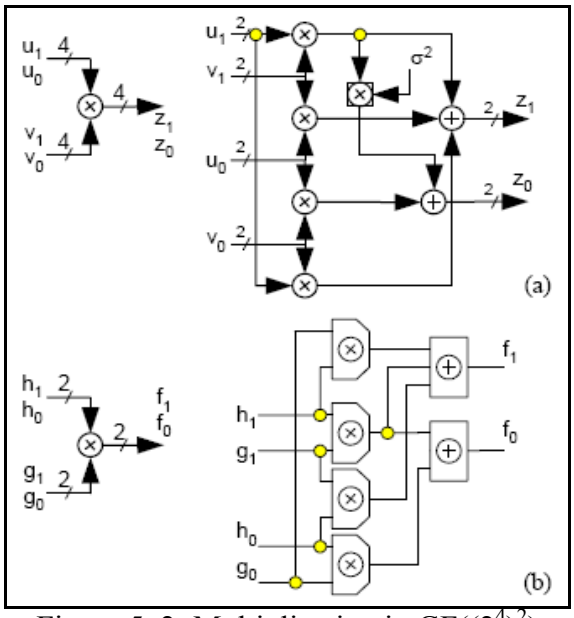


Figure 5–2: Multiplication in GF($(2^4)^2$)

The same multiplier can be also implemented using the composite field technique. This is shown in Figure 5–2 (see also Section 5.2.2). The complexity is 16 AND, 14 OR, and with the delay time of 3τ. Similarly the inversion in *GF(2^4)* can be implemented directly as given in (see Section 5.2.2)) [29]. The complexity is 11 AND, 11 OR, 5 INV, and with the delay time of 3τ. This inversion can be implemented in the composite field *GF((2^2)^2)* . The similar scheme as that shown in Figure 5–1-b is reapplied. The complexity is 12 AND, 9 OR and with the delay time of 4τ. Regardless the inserted delays, the above discussed inversion in *GF(2^8)* is summarized in Table 5–1 below.

Table 5–1: Complexity of GF($(2^4)^2$) inversion

Inversion GF($(2^4)^2$	Complexity*				Unit ** required
	AND	XOR	INV	τ	
Squaring	-	2	-	1	2
Fixed-Coeff. multiplication	-	1	-	1	1
Inversion (GF(2^4))	11	11	5	3	1
Inversion (GF($(2^4)^2$)	12	9	-	4	(1)
Multiplication (GF(2^4))	16	10	-	3	3
Multiplication (GF($(2^4)^2$)	16	14	-	3	(3)

The field mapping denoted by equations (5.1) and (5.2) above can be easily implemented with about 16 XORs as shown in Figure 5–3. If one needs the combined mapping noted by equations (5.10) and (5.12) can be implemented similarly.

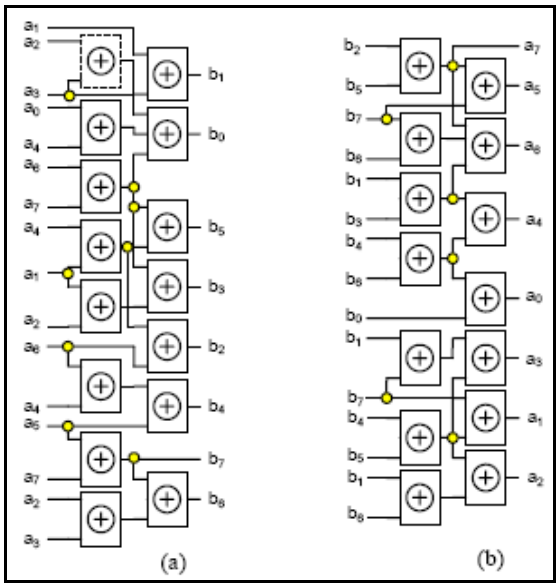


Figure 5–3: Field Mapping (a) and Inverse Mapping (b)

5.4. Comparison between difference S-box Implementations

Table 5–2 shows a performance comparison of various S-Boxes in 0.18_m ASIC libraries including our used method [48].

Table 5–2: **Comparison of various S-Box architectures (0.18 _m 1.8 V CMOS standard cell, 1 gate = 2 way-NAND)**

	Delay (ns)	Size (gate)
Itoh and Tsujii [14]	4.11	1,540
PPRM (1-stage)	1.32	2,242
Twisted-BDD [17]	0.66	1,977
BDD	0.96	857
Table look-up	0.91	1,706
Composite Field	3.01	305
SOP (1-stage)	0.97	1,142
Satoh Morioka	1.86	701

As shown in the above table the S-box designed with composite field architecture has the minimum number of gates, but it also has a large delay when compared with other methods, so we will choose it in the design of the optimized area AES.

Chapter 6

Hardware Implementation of AES

Hardware implementations of cryptographic algorithms have a long history. Traditionally, algorithms were implemented in hardware to achieve a higher speed than with implementations in software. The requirements of contemporary and future Applications however, demand often other properties of hardware implementations.

Today we can identify two application scenarios where hardware implementations are advantageous over software implementations. Firstly, these are high-speed applications where a cryptographic co-processor performs the cryptographic operations in order to relieve the rest of the system. Secondly, these are applications where low power and low area requirements are stringent. In both application scenarios, the secure storage of keys is important.

The AES has been the topic of much research to find suitable architectures for its hardware implementation. Architectural choices for a given application are driven by the system requirements in terms of speed and the resources consumed. This can simply be viewed as throughput and area; however, latency may also be important as may the cipher's mode of operation. The FIPS-197 specification details a number of modes of operation for the cipher, for example, the simplest is the Electronic Code Book (ECB). Additional resilience to attack can be gained by using one of the feedback modes, for example, Output Feed Back (OFB) mode unfortunately such modes also limit the effectiveness of pipelining.

6.1. Introduction to Digital VLSI Design on FPGAs

In this section we will give a brief introduction to Digital VLSI design cycle using HDL (Hardware Description Languages) and FPGAs (Field Programmable Gate Arrays)

6.1.1. Introduction to Digital VLSI Design

Traditionally, digital design was a manual process of designing and capturing circuits using schematic entry tools. This process has many disadvantages and is rapidly replaced by new methods [52].

System designers are always competing to build cost-effective products as fast as possible in highly competitive environment. In order to achieve this, they are turning to using top-down design methodologies that include using hardware description languages and synthesis, in addition to just the more traditional process of simulation. A product in this instance is any electronic equipment containing ASICs or FPGAs.

In recent years, designers have increasingly adopted top down design methodologies even though it takes them away from logic and transistor level design to abstract programming. The introduction of industry standard HDLs and commercially available

synthesis tools have helped establish this revolutionary design methodology. Some of the advantages are:

- Increased productivity yields shorter development cycles with more product features and reduced time to market.
- Reduced non-recurring engineering costs.
- Design reuse is enabled.
- Increased flexibility to design changes.
- Faster exploration of alternative architectures and technology libraries.
- Enables use of synthesis to rapidly sweep the design space of area and timing, and to automatically generate testable circuits.
- Better and easier design auditing and verification.

6.1.2. Top down design methodology

In an ideal word, a true top-down system level design methodology would mean describing a complete system at an abstract level using a HDL and the use of automated tools, for example, partitioners and synthesizers. This would drive the abstract level description to implement on ASICs or FPGAs.

A top down design methodology takes the HDL model of hardware, written at a high level of behavioral abstraction (system or algorithmic) Down through intermediate levels, to a low (gate or transistor) level as shown in Figure 6–1.

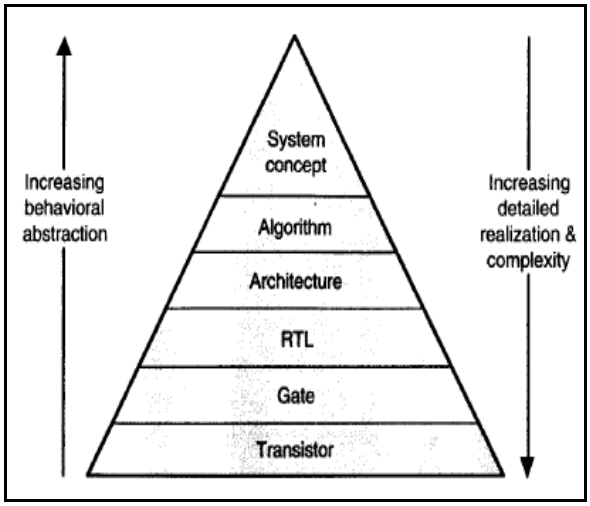


Figure 6–1: Behavioral level of abstraction pyramid

The term behavior represents the behavior of intended hardware and its independent of the level of abstraction by which it is modeled. A design represented at the gate level still represents the behavior of hardware intent. As hardware models are translated to progressively lower levels they become more complex and contain more structural detail. The benefit of modeling hardware at higher levels of behavioral abstraction is that designers are not overwhelmed with large amount of unnecessary details and complexity of design task is reduced. Figure 6–2 shows how the different behavioral levels of

abstraction overlap between the design-domains of pure abstraction, structural decomposition and physical implementation.

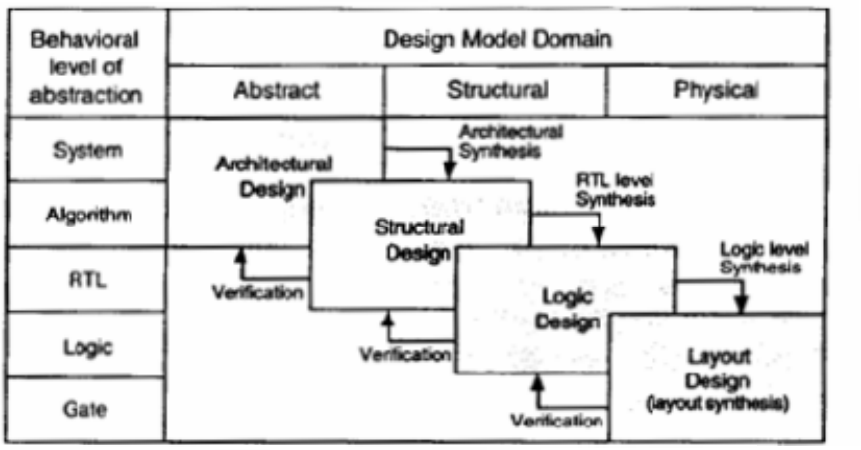


Figure 6–2: Design domain for different levels of design abstraction

6.1.3. Introduction to FPGA technology

The use of FPGA has been expanding from its traditional role in prototyping to mainstream production. This change is being driven by commercial pressures to reduce design cost, risk and achieve a faster time to market. Advances in technology have resulted in mask programmed mass produced versions of FPGA fabrics being offered by the leading manufacturers which, for some applications, remove the necessity to move prototype designs from FPGA to ASIC whilst still achieving a low unit cost [53].

The Xilinx Virtex family is the most used FPGA series in academia concerning cryptographic implementations. This section will give some more detailed description, about FPGA in general and the chips used in the cited contributions.

The original 2.5-Volt Virtex family was introduced in 1998 offering features, like Block RAM, Distributed RAM and High-speed external memory interfaces, Delay-Locked Loops (DLLs), and SelectI/O. The Virtex-E family, introduced in 1999, delivers more RAM, more DLLs, the SelectLink technology and high speed differential signaling. Virtex-4 and Virtex-5 FPGAs are the high end chips offered by Xilinx.

One of the biggest architectural differences between FPGAs and CPLDs is that FPGAs have an array of many small logic blocks with vast interconnection networks, while Complex Logic Device (CPLDs) have a few large logic block based on PALs, with smaller interconnection networks. Hence, FPGAs exist of three main components: Configurable Logic Blocks (CLBs), interconnections, and I/O blocks (see Figure 6–3).

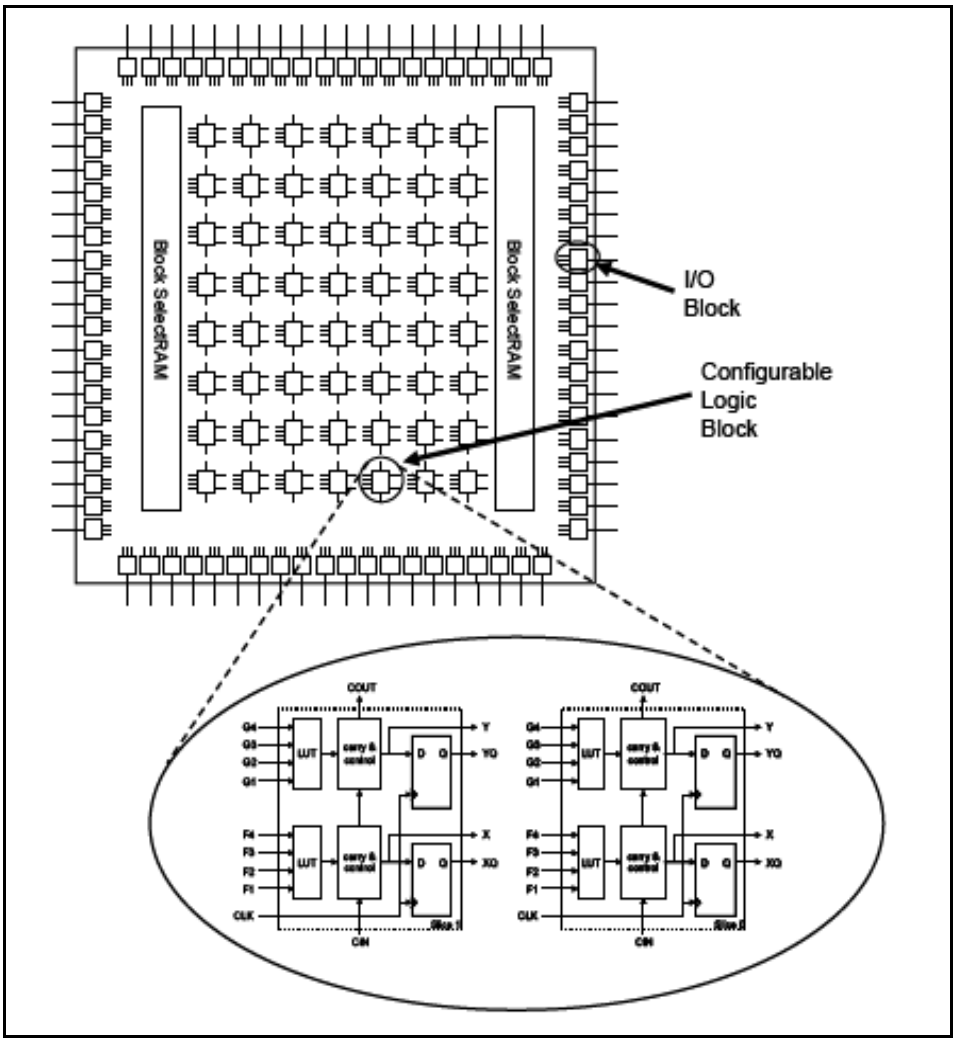


Figure 6–3: Structure of the Virtex FPGA

FPGA technology is usually based on SRAM, flash, EEPROM or anti-fuse interconnections. The Virtex family is based on SRAM technology. The I/O blocks of FPGAs are very similar to the I/O pads in an ASIC and act as buffers to the outside world. The CLBs are the core logic element in an FPGA. The main block in a Virtex CLBs is the logic cell (LC). Each Virtex CLB contains of four LCs, organized in two similar slices. An LC includes a 4-input function generator, carry logic, and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. The slice includes 4-input look-up tables (LUTs), which are the function generators of the CLB. Each LUT can provide a 16 x 1-bit synchronous RAM and the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM. The F5 multiplexer provides the ability to combine the function generator outputs, either to a function generator (implementing any 5 input function), to a 4:1 multiplexer, or to a selected functions of up to nine inputs. F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5 multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs. The XOR gate provides the possibility to implement a 1-bit full adder in one LC and the AND gate allows a efficient multiplier implementation.

Moreover, large block of RAM memories which are organized in columns are provided. Virtex devices have two columns that extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Virtex device 64 CLBs high contains 16 memory blocks per column, and a total of 32 blocks.

6.2. Hardware Basic Decisions and Considerations

AES algorithm has many architectures, key sizes and modes of operation. In our implementation of the algorithm we have make some decisions and considerations based on the optimization goal (Area/ Speed). In this section we state the decisions and considerations which we will use in our implementation for Optimized Area/ Speed AES:

- We only consider 128-bits key size, which means that we will have only ten Enc/ Dec rounds.
- We make separate implementations for both encryption and decryption modules based on that many applications have a separate transmitter and receiver.
- In our Implementation of the AES algorithm with a small area we will select the basic reference architecture (see Section 4.1.1) which needs the implementation of one round only and re-use it to complete the ten encryption rounds, we also designed the Encryptor/ Decryptor to complete one encryption round in one clock cycle so the output of the Encryptor/ Decryptor will be valid after ten clock cycles from the data entrance.
- In our Implementation of the AES algorithm with a high speed we will select the pipelined architecture with ($K=Nr=10$) ten rounds (see Section 4.1.1). This design will allow to us to update the input data each clock cycle but it will increase the area about ten times larger than optimized area AES.
- Another basic architecture decision we had make was the key schedule architecture. There are two ways for generating the round keys for encryption, either by generating all the sub-keys beforehand and storing them in a buffer, or generating all the sub-keys on the fly in parallel with the encryption module. Since buffer storage could take up substantial amount of space, we decided to generate the sub-keys on the fly during encryption. For the encryptor we implement the hardware required to generate one set of sub-key and re-use in the calculation of other the sub-keys, and at the same time also use one clock cycle for one sub-key generation.

  For the decryptor we must generate the last sub-key first to use it in the first decipher round, so we couldn't use the same key expansion architecture used with cipher and we must select one of the other architectures either by generation of all sub-keys beforehand and storing them in a RAM, or by generation of all sub-keys using pipelined architecture.
- Another decision we had make was about the mode of operation. There are many modes of operation of the AES block cipher, and this modes are classified into two major classes: feedback and non-feedback modes (see Section 2.2.1), in our design we will concern in non feedback mode of operation ECB (Electronic code book)
- Our hardware designs have been encoded in VHDL'93, and targeted on a Xilinx Virtex 4 FPGA. We use Xilinx ISE 7.1i and modelsim programs for simulation, synthesis, place and route for my designs.

6.3. Optimized Area AES Encryptor/ Decryptor

For optimized area AES our goal is to implement AES encryptor/ decryptor with a small area which it could be used in smart cards and other applications which required a high security with minimum resources. In this section we will illustrate the hardware architecture of cipher, decipher and key expansion units and then we will present my simulation, synthesis and implementation results.

6.3.1. Hardware Architecture

6.3.1.1. Cipher/ decipher Hardware Architecture

As shown in Figure 6–4 we use the basic architecture for the Cipher/ Decipher Module which consists of:

1. A multiplexer which is used to choose between the plaintext and the data output from the cipher round. In our design the multiplexer selector is controlled by a counter to allow plaintext to pass each ten clock cycles and allows the output data of cipher round to pass during this ten clock cycles.
2. 128-bits register which is used to hold the output data of the multiplexer to use it as input to the cipher round.
3. Cipher round which is used to make one round of encryption and output cipher data after ten clock cycles.

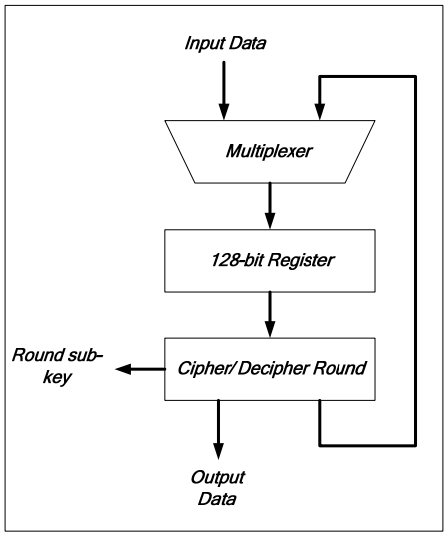


Figure 6–4: Optimized area cipher/ decipher Architecture

6.3.1.2. Key Expansion Hardware Architecture

The cipher key expansion module shown in Figure 6–5 has the same architecture as cipher module, and this architecture is used to generate four words (only one sub-key) each clock cycle in parallel with cipher. This architecture of key expansion module has the ability of changing the encryption key each ten clock cycles (which means that we can encrypt each Plaintext with a different key) and this is consider an advantage when compared with the other architecture which generates all sub-keys and stores them in RAM.

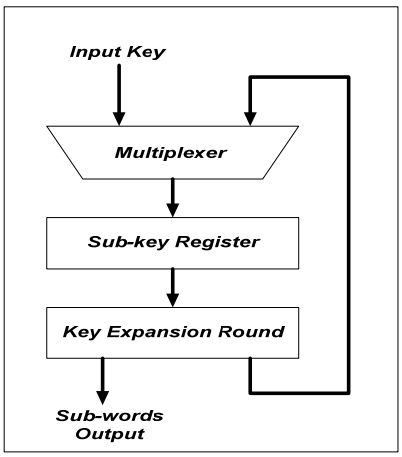


Figure 6–5: Cipher key expansion architecture

The decipher key expansion module shown in Figure 6–6 uses one key expansion round and Nr 128 bits registers (RAM), and it could be used for both encryption and decryption modules if we want to make resources sharing in case of design the same chip for encryption and decryption. Also this architecture could be used for other architectures of the AES such as pipelined or loop unrolled architecture.

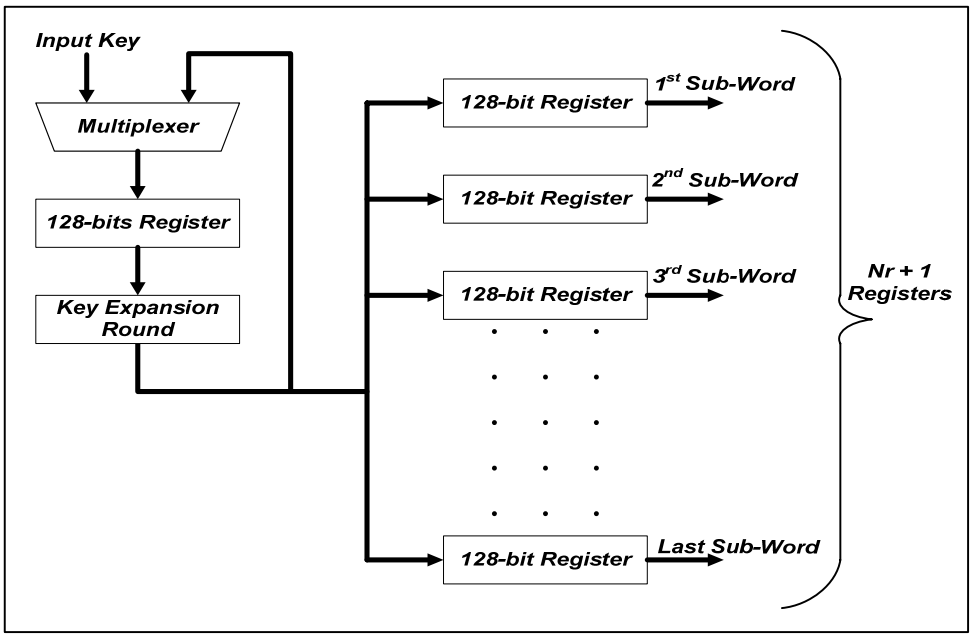


Figure 6–6: Decipher key expansion architecture

6.3.1.3. Cipher/ Decipher Round Architecture

AES cipher round can be divided into four basic operation blocks which operates on array of bytes, organized as a 4×4 matrix called the state as mentioned before. Four basic steps, called layers consist of the SubBytes transformation, the ShiftRows transformation, the MixColumns transformation, and AddRoundKey (see Section 2.5) as shown in Figure 6–7-a.

1. The SubBytes transformation: Non-linear byte substitution which is composed of multiplicative inverse and affine transformation. We use the composite field method described in Section 5.1 in our implementation of this transformation which will save the area but it will increase the delay.
2. The ShiftRows transformation: Linear diffusion process, operating on individual rows. Depending on the row location, offset of left shift varies from zero to three bytes.
3. The MixColumns transformation: Matrix multiplication over *GF* (2^8). Column vector is multiplied with a fixed matrix where the bytes are treated as a polynomials rather than numbers. We use the method of substructure sharing described in Section 4.2.3 in our implementation for this transformation which has the advantages of low area and high speed.
4. AddRoundKey: Simple byte XOR operation with the round key.

These four layer steps describe one round of AES. A 128 bit round sub-key, used in AddRoundKey operation, is generated by the key schedule.

Excluding the first and the last round, AES encryption round executes nine iterations. First round of the encryption step performs XOR with the original key and the last round skips MixColumns layer.

All four layers described above have corresponding inverse operations such that the decryption is simply the reverse order operations of these inverse transformations as shown in Figure 6–7-b.

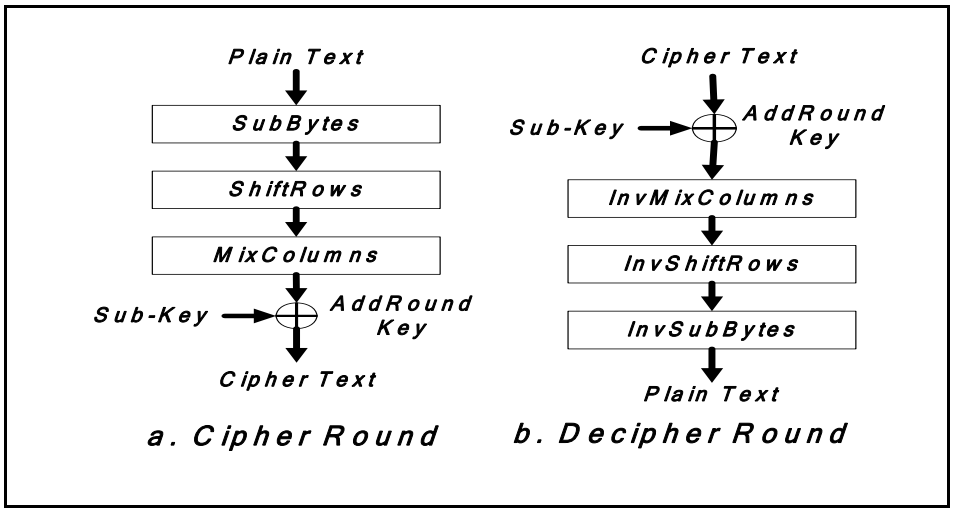


Figure 6–7: a. Cipher Round, b. Decipher Round

6.3.1.4. Key Expansion Round Architecture

Key expansion Round can be divided into the following operations which operates on array of four words (see section 2.6) as shown in Figure 6–8.

1. RotWord transformation: A simple rotate operation which rotates the word one byte to the right.
2. SubWord transformation: Each byte in the word is substituted by the SubBytes transformation.
3. XOR: Simple word XOR operation.

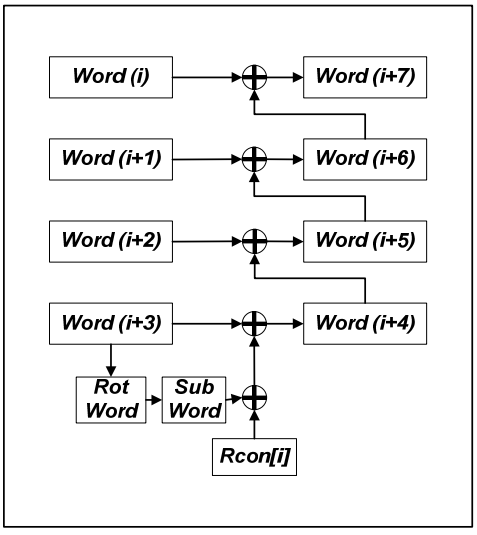


Figure 6–8: Key Expansion Round Architecture

Note that Rcon[i] are constants depend on the round number.

6.3.2. Hardware Implementation Results

6.3.2.1. Encryptor/ decryptor Circuit

Figure 6–9 and Figure 6–10 shows the encryptor and decryptor top level entities and Table 6–1 shows names, modes and functions of signals used in the design of optimized area AES encryptor and decryptor

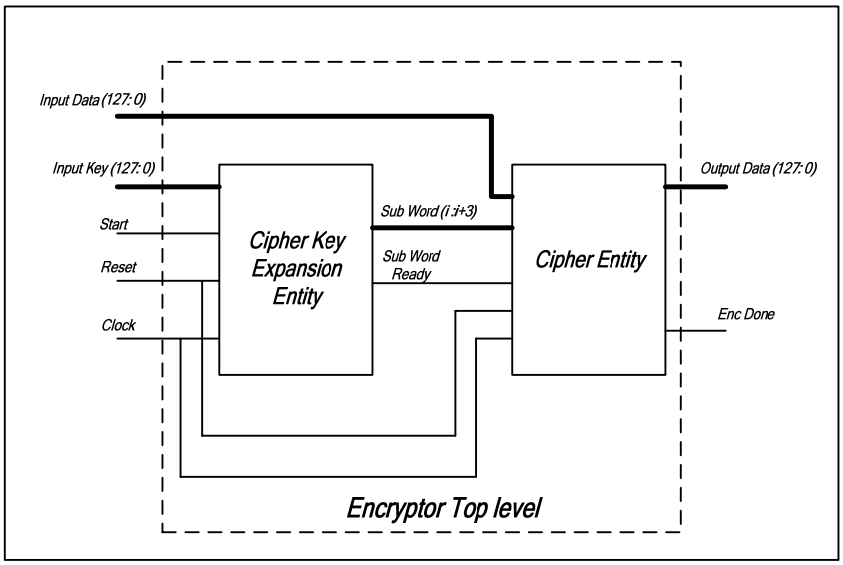


Figure 6–9: Encryptor Top level Entity

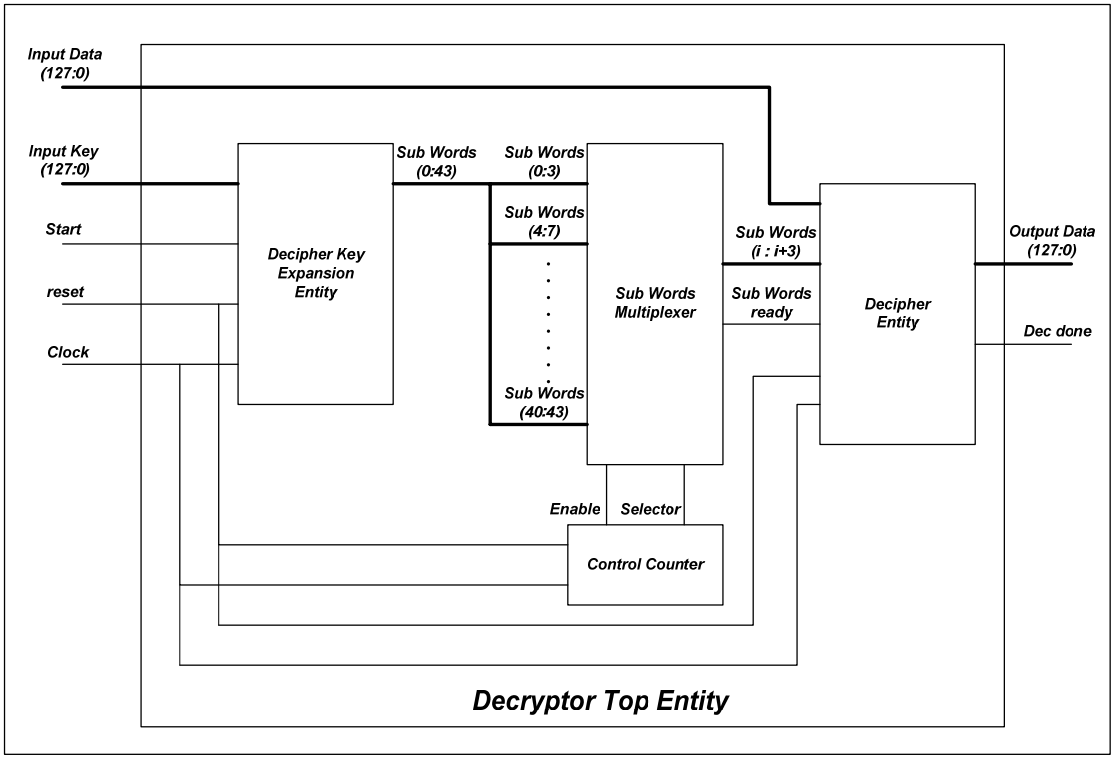


Figure 6–10: Decryptor Top level Entity

Table 6–1: Encryptor/ decryptor signals names and functions

Signal Name	***Type***	***Signal Function***
a. Encryptor/ Decryptor		
Input Data (127:0)	*Input*	128-bits data input to cipher (plaintext) or to decipher (ciphertext)
Input Key (127: 0)	*Input*	128-bits input key used in generation of sub key words
Reset	*Input*	Active high asynchronous reset signal
Start	*Input*	Active high asynchronous start signal used to start the cipher/ decipher key expansion unit which will be used in the interface of the encryptor/ decryptor and external peripheral
Clock	*Input*	Positive edge clock for the encryptor/ decryptor
Output Data (127:0)	*Output*	128-bits data output of cipher (ciphertext) or of decipher (plaintext)
Enc Done, Dec done	*Output*	Active high output signal when the output of the encryptor or the decryptor is ready which will be used in the interface of the encryptor/ decryptor and external peripheral
SubWord (i:i+3)	*Internal*	Four sub words used in encryptor/ decryptor round which changes each clock cycle where *i* depends on the present round number
Sub Word Ready	*Internal*	Active high signal acts as a start signal for the cipher/ decipher unit
b. Decryptor		
Sub Word (0:43)	*Internal*	All the 44 sub words output of the decipher key expansion unit
Sub Words Ready	*Internal*	Active high signal indicates that all key sub words are ready which will be used as a start signal to the control counter
Enable	*Internal*	An enable signal to the sub words Multiplexer which will be used to start output the four sub words which will be used in the decipher unit.

6.3.2.2. Functional Simulation Results

Figure 6–11and Figure 6–12 shows the functional simulation results for the optimized area AES encryptor and decryptor. As shown in Figure 6–11 we initially apply the AES test vector data to the input and key, we got the correct output data (as test vector output) [] after *Nr (10)* clock cycles as mentioned in section 6.2. Then we apply a random input data changes each ten clock cycles and we found that the output data responds to the input change which means that the encryption and decryption process is done one time each *Nr (10)* clock cycles. The decryptor test shown in Figure 6–12 is similar to the encryptor test with one difference in the delay between input and output (*2Nr* (20) clock cycles) which consists of the combination of the sub key words generation delay (*Nr (10)* clock cycles) and the decryption required time (*Nr (10)* clock cycles).

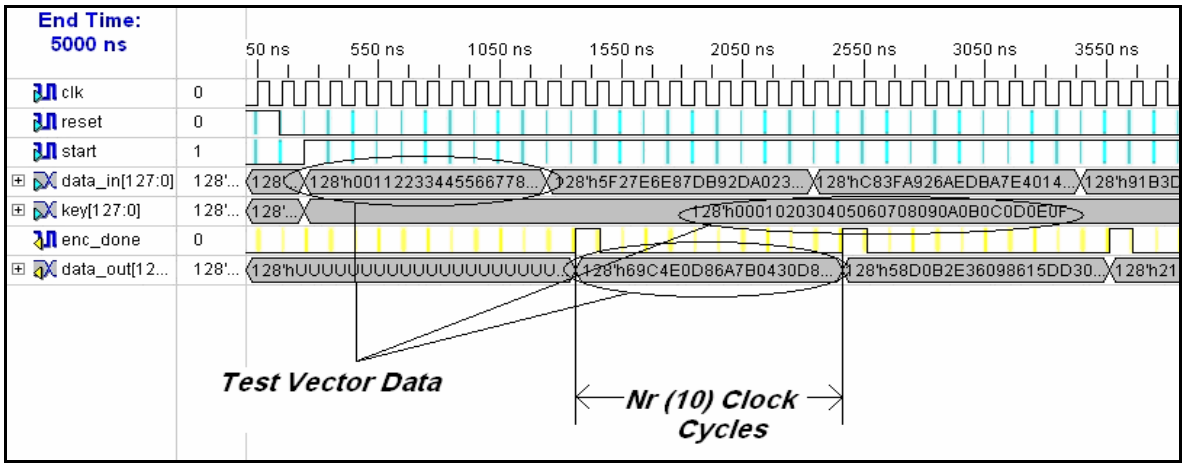


Figure 6–11: Behavioral simulation of optimized area AES Encryptor

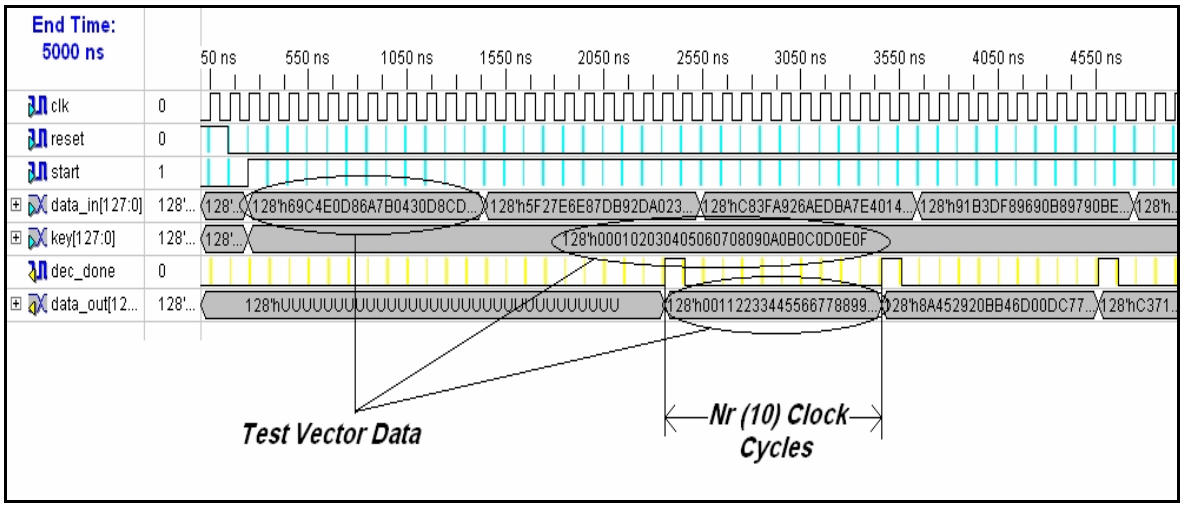


Figure 6–12: Behavioral simulation of optimized area AES Decryptor

6.3.2.3. Hardware Implementation Results

Table 6–2 shows the hardware implementation results of the optimized area AES cipher and decipher internal transformations. It is clear that SubBytes transformation and its inverse consume about (75%-80%) of the area and also they have the largest delay. Also we can see that ShiftRows transformation and it inverse have no area or delay. One can see that the routing delay increase as the levels of logic increase (number of gates in the critical path) and this delay may becomes larger than the logic gates delay.

Table 6–2: Implementation results for separate cipher/ decipher transformations

Transformation	*Number of gates*	*Maximum Delay*
a. Cipher Transformations		
SubBytes (State)	# XORs : 1192 1-bit xor2 : 666 1-bit xor3 : 264 1-bit xor4 : 185 1-bit xor5 : 68 1-bit xor6 : 9	8.580ns (2.086ns logic, 6.494ns route) (24.3% logic, 75.7% route) (Levels of Logic = 13)
ShiftRows (state)	0	0

MixColumns (state)	# XOR : 176 1-bit xor2 : 160 8-bit xor4 : 16	0.998ns (0.469ns logic, 0.529ns route) (47.0% logic, 53.0% route) (Levels of Logic = 2)
AddRoundKey (state)	# XORs : 16 8-bit xor2 : 16	0.322ns (0.322ns logic, 0.000ns route) (100.0% logic, 0.0% route) (Levels of Logic = 1)
b. Decipher Transformations		
InvSubBytes (state)	# XORs : 1130 1-bit xor2 : 658 1-bit xor3 : 209 1-bit xor4 : 188 1-bit xor5 : 65 1-bit xor6 : 10	7.255ns (1.792ns logic, 5.463ns route) (24.7% logic, 75.3% route) (Levels of Logic = 11)
InvShiftRows (state)	0	0
InvMixColumns (state)	# XORs : 715 1-bit xor2 : 619 1-bit xor3 : 79 1-bit xor4 : 1 8-bit xor4 : 16	2.703ns (0.910ns logic, 1.793ns route) (33.7% logic, 66.3% route) (Levels of Logic = 5)
AddRoundKey (state)	# XORs : 16 8-bit xor2 : 16	0.322ns (0.322ns logic, 0.000ns route) (100.0% logic, 0.0% route) (Levels of Logic = 1)

Table 6–3 shows the FPGA device utilization and timing characteristics of the optimized area AES Encryptor and Decryptor. It is clear that decryptor area is larger than Encryptor which is mainly due to that Decipher Key Expansion unit have a large area when compared to the Cipher Key Expansion unit and also that the decipher area is larger than the cipher area.

Table 6–3: FPGA (4vlx60ff668-12) device utilization and timing characteristics of optimized area AES

a. Encryptor			
	Cipher Key Expansion	***Cipher***	***Encryptor***
Number of Slices	452 out of 26624 1%	1151 out of 26624 4%	1468 out of 26624 5%
Number of Slice Flip-Flops	170 out of 53248 0%	290 out of 53248 0%	450 out of 53248 0%
Number of 4 inputs LUTs	856 out of 53248 1%	2194 out of 53248 4%	2799 out of 53248 5%
Minimum Period (Maximum Frequency)	6.824ns (146.547MHz)	7.774ns (128.636MHz)	7.693ns (129.989MHz)
Minimum input arrival time before clock	3.251ns	3.694ns	3.697ns
Maximum output required time after clock	4.163ns	3.921ns	3.921ns
b. Decryptor			
	Decipher Key Expansion	***Decipher***	***Decryptor***

Number of Slices	1175 out of 26624 4%	2366 out of 26624 8%	2752 out of 26624 10%
Number of Slice Flip-Flops	1651 out of 53248 3%	443 out of 53248 0%	2055 out of 53248 3%
Number of 4 inputs LUTs	815 out of 53248 1%	4459 out of 53248 8%	4801 out of 53248 9%
Minimum Period (Maximum Frequency)	7.007ns (142.709MHz)	7.972ns (125.433MHz)	8.009ns (124.863MHz)
Minimum input arrival time before clock	3.851ns	3.654ns	3.750ns
Maximum output required time after clock	4.007ns	3.921ns	3.921ns

We can calculate the throughput of the Encryptor/ Decryptor From equation (4.1)

$$Encryptor\,Throughput = \frac{128 \times 129.989\,MHz}{10} = 1.664\,Gbps$$

$$Decryptor\,Throughput = \frac{128 \times 124.863\,MHz}{10} = 1.598\,Gbps$$

6.4. Optimized Speed AES Encryptor/ Decryptor

For Optimized Speed AES our goal is to implement AES Encryptor/ Decryptor with a high speed which it could be used in network routers and other applications which required a high security with a high speed.

In this section i will illustrate the hardware architecture of cipher, decipher and key expansion unit which i used in my design and then i will present my simulation, synthesis and implementation results.

6.4.1. Hardware Architecture

6.4.1.1. Cipher/ decipher Hardware Architecture

We use the pipelined architecture with *k=Nr=10* for the Cipher/ Decipher Module as shown in Figure 6–13. Note that there is no multiplexer because we will not need for feedback because we implement all the Cipher/ Decipher rounds.

6.4.1.2. Key Expansion Hardware Architecture

For both of encryptor and decryptor we will use the architecture shown in Figure 6–14 which is called on the fly architecture. This architecture consumes a large area but it has the advantage of the possibility of changing the input key each clock cycle, which means that we could update the key for each input data.

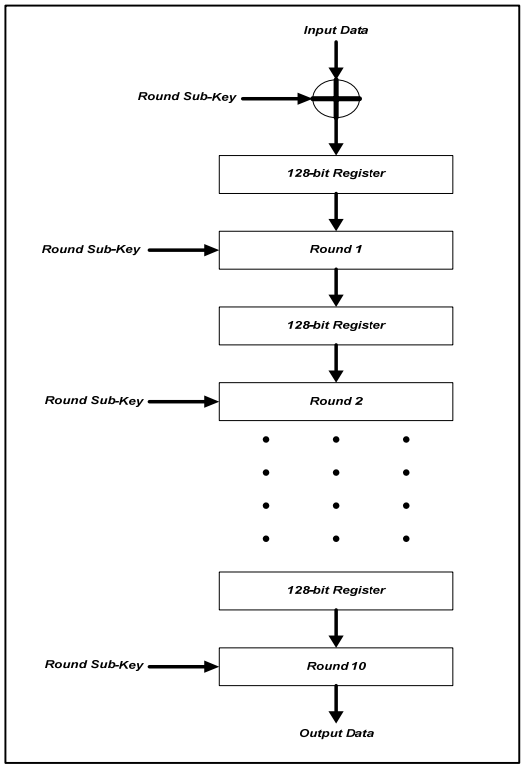


Figure 6–13: Optimized Speed Cipher/ Decipher Architecture

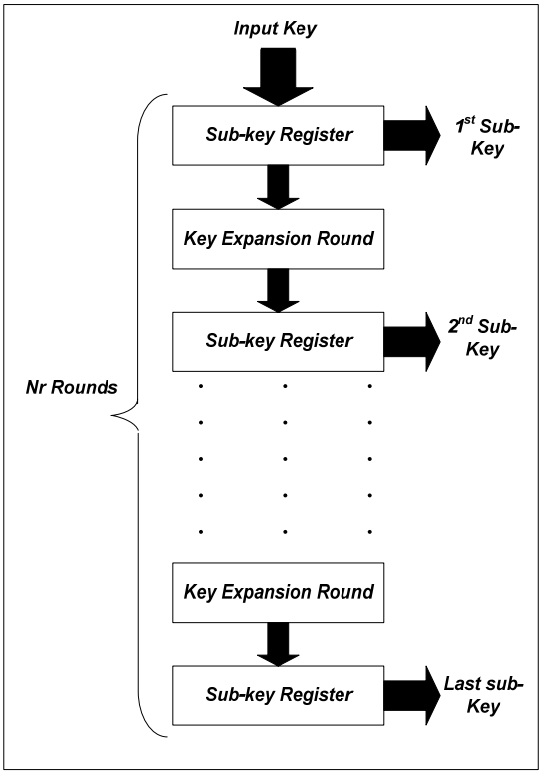


Figure 6–14: Optimized speed Key Expansion Architecture

6.4.1.3. Cipher/ Decipher Round Architecture

The Cipher/ Decipher round architecture will not differ from that used in the optimized area AES (see Figure 6–7). The difference will be in the transformations implementation methods. We will implement all transformations in the same methods like optimized area AES except the SubBytes/ InvSubBytes Transformation which will be implemented using the Look Up Table (ROM) method to decrease the delay.

6.4.1.4. Key Expansion Round Architecture

We will use the same architecture shown Figure 6–8 which is used in the optimized area AES. We will implement the SubWord Transformation using the ROM method.

6.4.2. Hardware Implementation Results

6.4.2.1. Encryptor/ decryptor Circuit

Figure 6–15 shows the encryptor and decryptor top level entities which is similar to the optimized area AES top level entity with some differences in internal signals(Sub Key Words are 44 words and Sub Key Ready are ten signals one for each round). The signals names and functions are shown in Table 6–1.

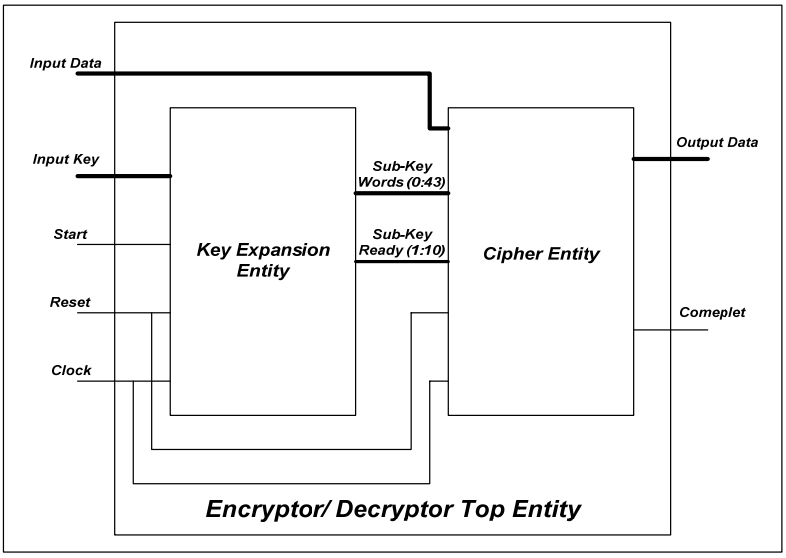


Figure 6–15: Encryptor/ Decryptor top level entity

6.4.2.2. Functional Simulation Results

Figure 6–16 and Figure 6–17 shows the behavioral simulation results of the optimized speed AES encryptor and decryptor. As shown in Figure 6–16 we initially apply the test vector data and ensure that the output is correct, also we can see that we have *Nr (10)* clock cycles delay between input and output data. After this we apply a random data each clock cycle and we found that the output responds to the input data change. The decryptor test shown in Figure 6–17 is similar to the encryptor test with one difference in the delay between input and output (*2Nr* (20) clock cycles) which consists of the combination of the sub key words generation delay (*Nr (10)* clock cycles) and the decryption required time (*Nr (10)* clock cycles).

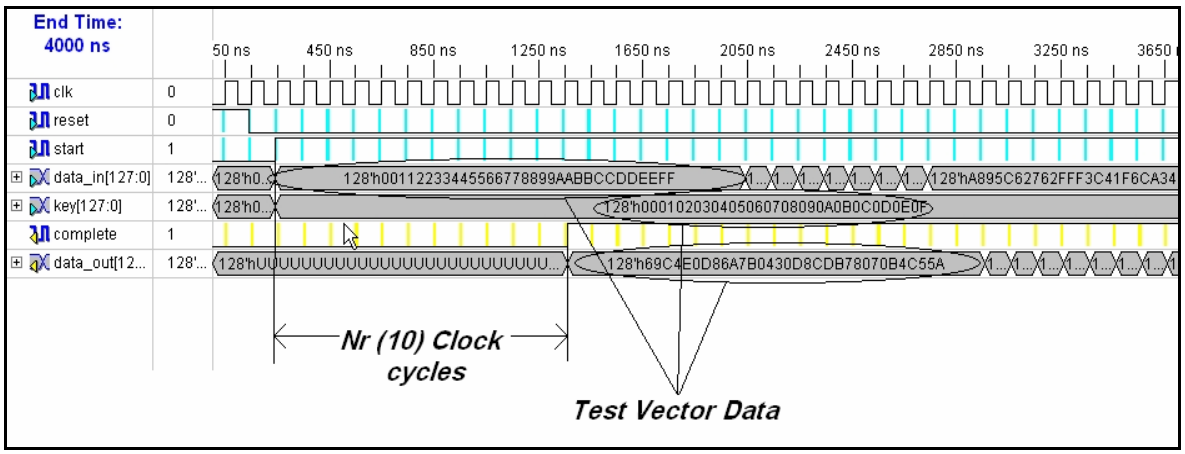


Figure 6–16: Behavioral simulation of optimized speed AES Encryptor

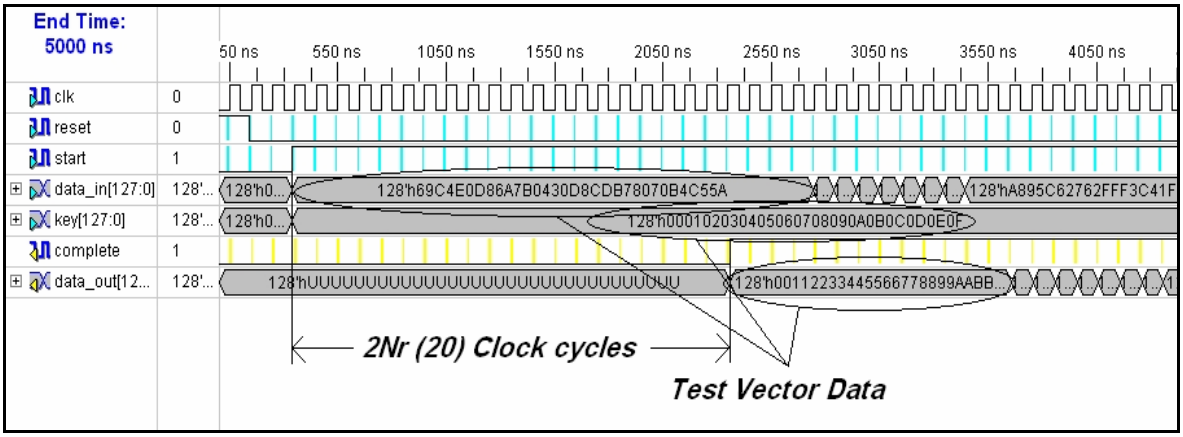


Figure 6–17: Behavioral simulation of optimized time AES Decryptor

6.4.2.3. Hardware Implementation Results

As we mentioned before, all cipher/ decipher transformations will be same to the optimized area AES (see Table 6–2) except the SubBytes/ InvSubBytes transformation. Table 6–4 shows the implementation results of the optimized speed SubBytes and InvSubBytes transformations.

Table 6–4: Optimized speed SubBytes and InvSubBytes implementation results

Transformation	*Number of gates*	*Maximum Delay*
SubBytes (State)	#ROMs :16 #256x8-bit ROM :16	1.274ns (1.274ns logic, 0.000ns route) (100.0% logic, 0.0% route) (Levels of Logic = 5)
InvSubBytes (state)	#ROMs :16 #256x8-bit ROM :16	1.274ns (1.274ns logic, 0.000ns route) (100.0% logic, 0.0% route) (Levels of Logic = 5)

Table 6–5 shows the FPGA device utilization and timing characteristics of the optimized speed AES encryptor and decryptor. It is clear that the decryptor has a larger area and delay than the encryptor because of the difference between the MixColumns and InvMixColumns transformation

Table 6–5: FPGA (4vlx60ff668-12) device utilization and timing characteristics of optimized speed AES

a. Encryptor			
	Cipher Key Expansion	***Cipher***	***Encryptor***
Number of Slices	4877 out of 26624 18%	15625 out of 26624 58%	18855 out of 26624 70%
Number of Slice Flip-Flops	4002 out of 53248 7%	8753 out of 53248 16%	9814 out of 53248 18%
Number of 4 inputs LUTs	7769 out of 53248 14%	23925 out of 53248 44%	31682 out of 53248 59%
Minimum Period	2.752ns	3.689ns	4.490ns

(Maximum Frequency)	(363.346MHz)	(271.076MHz)	(222.700MHz)
Minimum input arrival time before clock	2.882ns	5.299ns	5.991ns
Maximum output required time after clock	0.272ns	0.272ns	3.921ns
b. Decryptor			
	Decipher Key Expansion	***Decipher***	***Decryptor***
Number of Slices	4877 out of 26624 18%	16947 out of 26624 63%	20155 out of 26624 75%
Number of Slice Flip-Flops	4002 out of 53248 7%	8790 out of 53248 16%	9565 out of 53248 17%
Number of 4 inputs LUTs	7769 out of 53248 14%	29313 out of 53248 55%	36369 out of 53248 68%
Minimum Period (Maximum Frequency)	2.752ns (363.346MHz)	4.748ns (210.604MHz)	5.543ns (180.414MHz)
Minimum input arrival time before clock	2.882ns	6.682ns	6.364ns
Maximum output required time after clock	0.272ns	0.272ns	3.921ns

We can calculate the throughput of the Encryptor/ Decryptor From equation (3.1)

$$Encryptor\,Throughput = \frac{128 \times 222.7\,MHz}{1} = 28.51 Gbps$$

$$Decryptor\,Throughput = \frac{128 \times 180.414\,MHz}{1} = 23.09 Gbps$$

6.5. Comparison between Hardware Implementations of AES

In this section we will first introduce a comparison between our optimized area and optimized speed AES followed by a comparison between our hardware implementation and other previous implementations for the AES on FPGAs.

6.5.1. Comparison between Optimized Area and Optimized Speed AES

We will compare between the optimized area and optimized speed from the area and delay points of view. Taking into consideration that we synthesis both of optimized area and optimized speed on the same FPGA, from Table 6–3 and Table 6–5 we can summarize the differences in the following points:

- Area: the ratio between the device utilization in the optimized area *(5%, 10%)* and optimized speed *(70%, 75%)* AES is *(14, 8)* for the encryptor and decryptor consequently. This large difference in area is mainly due to the difference in the used hardware architectures, and secondly is due to the difference between algorithm implementation methods. The difference between the encryptor and decryptor for each approach is mainly due to the difference in the used key

expansion hardware architecture, and secondly is due to the difference between direct and inverse transformations.

- Clock Frequency: The optimized area maximum frequency is *(129.989, 124.863 MHz)* for the encryptor and decryptor consequently, while the optimized speed maximum frequency is *(222.7, 180.414 MHz)* for the encryptor and decryptor consequently. The difference between the maximum frequencies of the two approaches is due to difference between algorithm implementation methods. The difference between the encryptor and decryptor maximum frequencies for each approach is due to the difference between direct and inverse transformations.
- Throughput: The optimized area throughput is *(1.664, 1.598 Gbps)* for the encryptor and decryptor consequently, while the optimized speed throughput is *(28.51, 23.09 Gbps)* for the encryptor and decryptor consequently. The difference between the two approaches throughput is due to difference between the used hardware architectures. The difference between the encryptor and decryptor throughput for each approach is due to the difference between direct and inverse transformations.

6.5.2. Comparison of some Related Work for FPGAs

The architecture of an AES implementation mainly defines the required hardware resources on an FPGA. Additionally, the used synthesis tool and the target device influence this result.

Table 6–6 gives an overview of existing FPGA solutions. Because of the different FPGAs, most of the use Xilinx FPGAs, the values have to be seen as a relative comparison of resource requirements and data throughput [54].

Table 6–6: Comparison between difference FPGA implementations of AES

Authors	***LUTs***	***Block RAMs***	***Throughput [Gbps]***
Chodowiec	*222*	*3*	*0.166*
Chodowiec	*12,600*	*80*	*12.16*
Chodowiec	*2,057*	*8*	*1.265*
Chodowiec	*2,507*	*0*	*0.414*
Hodjat	*9,446*	*0*	*21.64*
Hodjat	*5,177*	*84*	*21.54*
McLoone	*2,222*	*100*	*7.0*
Pramstaller	*1,125*	*0*	*0.215*
Rouvroy	*146*	*3*	*0.358*
Saggese	*446*	*10*	*1.0*
Saggese	*648*	*10*	*1.82*
Saggese	*2,778*	*100*	*8.9*
Saggese	*5,810*	*100*	*20.3*
Standaert	*1,769*	*0*	*2.085*
Standaert	*15,112*	*0*	*18.560*

Wang	*1,857*	*0*	*1.604*
Zambreno	*387*	*10*	*1.41*
Zambreno	*1,254*	*20*	*4.44*
Zambreno	*2,206*	*50*	*10.88*
Zambreno	*3,766*	*100*	*22.93*
Zambreno]	*16,938*	*0*	*23.57*
Zhang	*9,406*	*0*	*11.965*
Zhang]	*11,022*	*0*	*21.556*
Our optimized area encryptor	*1468*	*0*	1.664
Our optimized area decryptor	*2752*	*0*	1.598
Our optimized speed encryptor	*18855*	*200*	28.51
Our optimized speed decryptor	*20155*	*200*	23.09

6.6. AES Crypto Processor

In this section we will introduce a simple processor that could be used to make the interface between the implemented AES encryptor/ decryptor datapaths and other external peripheral under the control of an operator. We introduce two modes of operation in the AES crypto processor. We called the first mode of operation discrete mode in which all the data operations (input, output and processing) could be done by orders from the operator. The second mode of operation is called the continuous mode in which the operator will only start to get the key and all the consequent operation will be done sequentially. We will use the first mode of operation to make the timing simulations (post synthesis, post map and post place and route simulations) and practical tests to the implemented hardware.

6.6.1. Crypto Processor Hardware Circuit

The crypto processor is mainly consists of the following components (Figure 6–18):

1. Encryptor/ decryptor unit: Any one of the previous implemented encryptors and decryptors could be used in the crypto processor.
2. Input interface unit: It is a serial interface with handshaking between the processor and external peripheral which is used to get the 128-bit data input (plaintext/ ciphertext) to the encryptor/ decryptor and it is mainly consists of serial to parallel shift register which takes data each clock cycle (it will takes 128 clock cycles to complete the data input to encryptor/ decryptor) and it has the start, complete and reset as asynchronous control signals.
3. Key interface unit: This component is similar to the input interface unit and it is used to input the 128-bits key used in encryption/ decryption unit.
4. Output interface unit: It is a 128-bit parallel to serial converter which is used to output the data (ciphertext/ plaintext) serially from the encryptor/ decryptor unit. Similar to the input and key interface units, the output interface unit takes

128 clock cycles to output the data and it has the start, complete and reset as asynchronous control signals.

5. Control Unit: It is a Moore finite state machine FSM (see Figure 6–19) which forms the interface between the operator and all another units in the processor. The control unit is used to generate all asynchronous control signals for all units in the design. From the Figure 6–19, it is clear that we will use the same states in the FSM for both of the two modes (continuous mode with the dotted arrows and discrete mode with solid arrows).

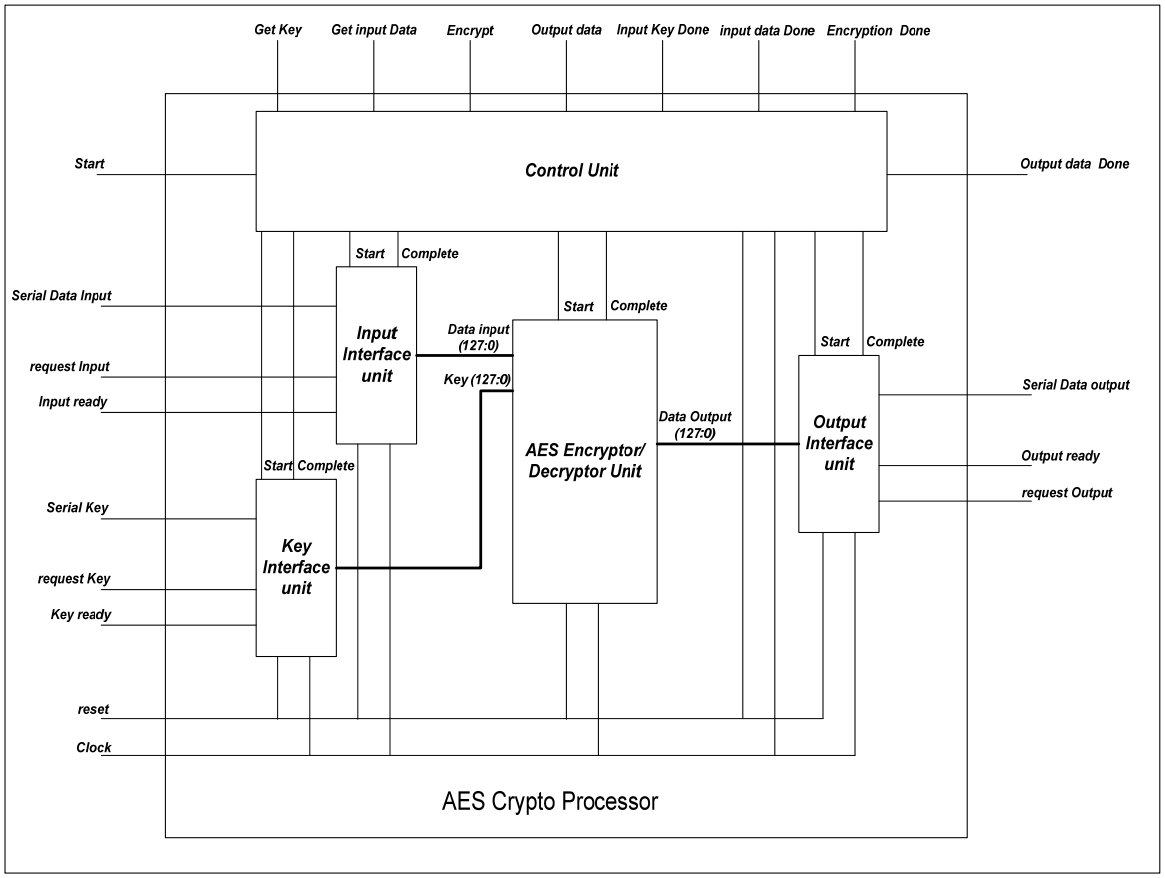
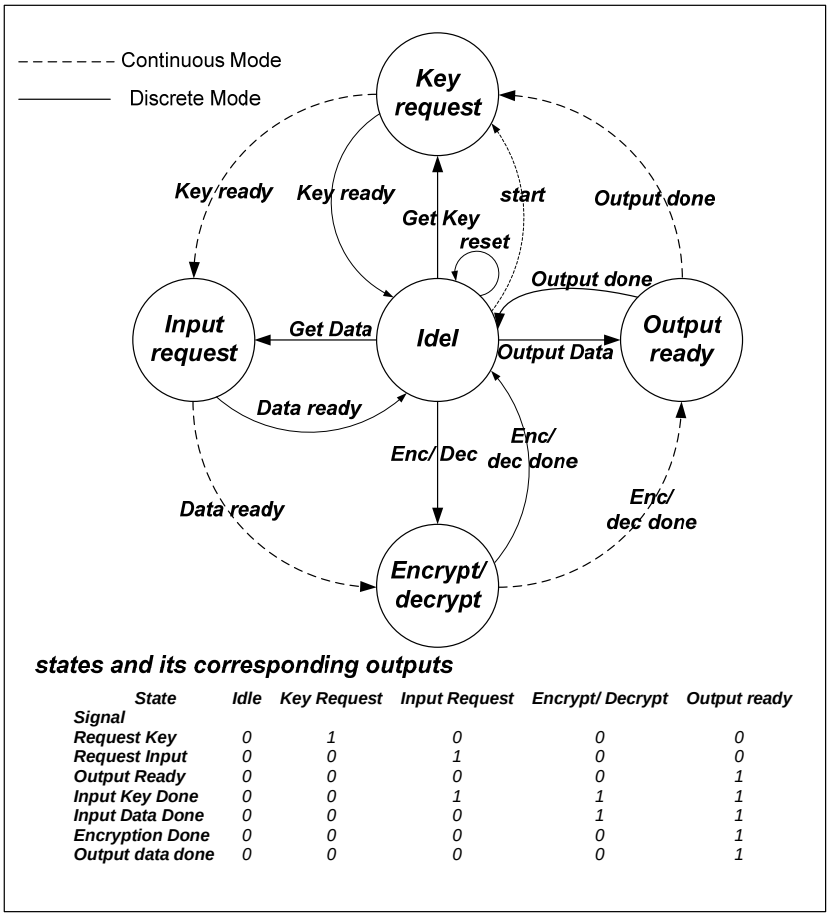


Figure 6–18: AES crypto processor



State Signal	Idle	Key Request	Input Request	Encrypt/ Decrypt	Output ready
Request Key	0	1	0	0	0
Request Input	0	0	1	0	0
Output Ready	0	0	0	0	1
Input Key Done	0	0	1	1	1
Input Data Done	0	0	0	1	1
Encryption Done	0	0	0	0	1
Output data done	0	0	0	0	1

Figure 6–19: Control Unit FSM

6.6.2. Crypto Processor Functional Simulation Results

Figure 6–20 shows the functional simulation for the AES crypto processor with the optimized area AES encryptor in the discrete mode of operation.

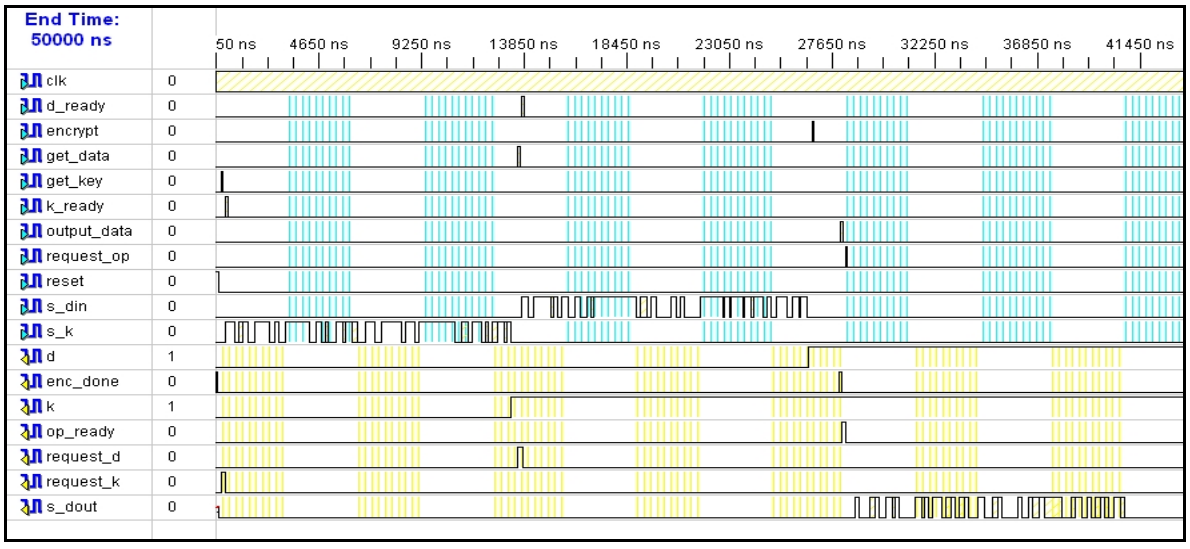


Figure 6–20: Behavioral simulation of AES crypto processor

6.6.3. Crypto Processor Timing Simulation Results

Figure 6–21 shows the post place and route simulation results (at 10 MHz Clock) which agrees with the functional simulation results and the following message appears on the modelsim simulator screen:

```
** Warning: /X_LATCHE RECOVERY  Low VIOLATION ON SET WITH RESPECT TO CLK;
#   Expected := 0.606 ns; Observed := 0.072 ns; At : 1.686 ns
#    Time: 1686 ps  Iteration: 5  Instance: /proc_test/uut/c7_dout_ok_4027
# ** Failure: Simulation successful (not a failure).  No problems detected.
#    Time: 50100 ns  Iteration: 0  Process: /proc_test/line__115 File: proc_test.timesim_vhw
```

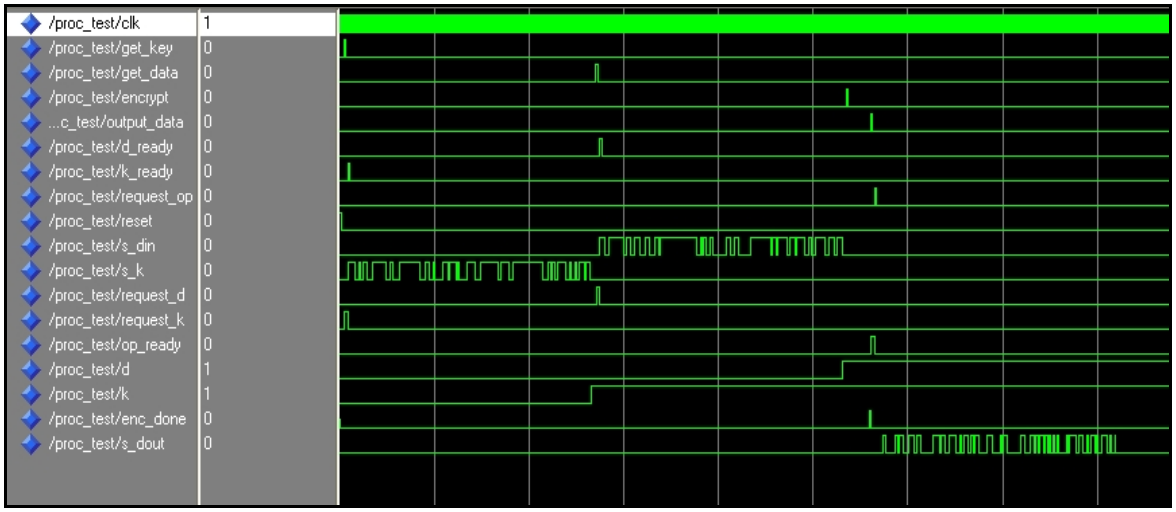


Figure 6–21: post place and route simulation of AES crypto processor

Similarly we made the timing simulation for the optimized area decryptor and optimized speed encryptor and decryptor and we got results as same as the above simulation.

Chapter 7

APPLICATIONS OF AES

In this chapter we present two applications of the AES. The first application is AES key wrap standard. The second application is deterministic random bit generator (DRBG) based on the AES block cipher.

7.1. AES Key Wrap

7.1.1. Introduction

The rapid growth of information technology that has resulted in significant advances in cryptography to protect the integrity and confidentiality of data is astounding. New algorithms have been introduced such as the Advanced Encryption Standard (AES) as defined in the Federal Information Processing Standard (FIPS) 197 to offer three security strengths: 128 bits, 192 bits and 256 bits. The use of AES requires the establishment/ wrap of shared keying material in advance. Manual distribution methods such as trusted couriers are inefficient and complex. They simply do not scale as the system grows. Key establishment/ wrap schemes are required to distribute keys in today's communication systems. Protocols such as S/MIME, SSL and IPSec all use key establishment/ wrap schemes. Key establishment/ wrap are fundamental to security that the American National Standards Institute (ANSI) and the National Institute of Standards and Technology (NIST) are producing standards and recommendations for key establishment/ wrap.

AES key wrap specification is intended to satisfy the NIST Key Wrap requirement to: Design a cryptographic algorithm called a Key Wrap that uses the Advanced Encryption Standard (AES) as a primitive to securely encrypt a plaintext key(s) with any associated integrity information and data, such that the combination could be longer than the width of the AES block size (128-bits). Each ciphertext bit should be a highly non-linear function of each plaintext bit and (when unwrapping) each plaintext bit should be a highly nonlinear function of each ciphertext bit. It is sufficient to approximate an ideal pseudorandom permutation to the degree that exploitation of undesirable phenomena is as unlikely as guessing the AES engine key. This key wrap algorithm needs to provide ample security to protect keys in the context of prudently designed key management architecture.

Throughout this section, any data being wrapped will be referred to as the key data. It makes no difference to the algorithm whether the data being wrapped is a key; in fact there is often good reason to include other data with the key, to wrap multiple keys together, or to wrap data that isn't strictly a key. So, the term "key data" is used broadly to mean any data being wrapped, but particularly keys, since this is primarily a key wrap algorithm. The key used to do the wrapping will be referred to as the key encryption key (KEK). In this document a KEK can be any valid key supported by the AES codebook. That is, a KEK can be a 128-bit key, a 192-bit key, or a 256-bit key [55].

7.1.2. Overview

Symmetric key algorithms may be used to wrap (i.e., encrypt) keying material using a key-wrapping key (also known as a key encrypting key). The wrapped keying material can then be stored or transmitted securely. Unwrapping the keying material requires the use of the same key-wrapping key that was used during the original wrapping process. Key wrapping differs from simple encryption in that the wrapping process includes an integrity feature. During the unwrapping process, this integrity feature detects accidental or intentional modifications to the wrapped keying material. The AES key wrap is designed to wrap or encrypt key data. The key wrap operates on blocks of 64 bits. Before being wrapped, the key data is parsed into *n* blocks of 64 bits.

The only restriction the key wrap algorithm places on *n* is that *n* is at least two. (For key data with length less than or equal to 64 bits, the constant field used in this specification and the key data form a single 128-bit codebook input making this key wrap unnecessary.) It is recognized that $n \leq 4$ will accommodate all supported AES key sizes. However, other cryptographic values often need to be wrapped. One such value is the seed of the random number generator for DSS. This seed value requires $n > 4$. Undoubtedly other values require this type of protection. Therefore, no upper bound is imposed on *n*. The AES key wrap can be configured to use any of the three key sizes supported by the AES codebook. The choice of a key size affects the overall security provided by the key wrap, but it does not alter the description of the key wrap algorithm. Therefore, in the description that follows, the key wrap will be described generically; i.e. no key size will be specified for the KEK.

7.1.3. Key Wrapping Algorithm

The specification of the key wrap algorithm requires the use of the AES codebook. The next three sections will describe the key wrap algorithm, the key unwrap algorithm, and the inherent data integrity check.

7.1.3.1. Key Wrap

The inputs to the key wrapping process are the KEK and the plaintext to be wrapped. The plaintext consists of *n* 64-bit blocks, containing the key data being wrapped. The key wrapping process is described below.

Inputs: Plaintext, n 64-bit values $\{P_1, P_2, \ldots, P_n\}$,
Key, K (the KEK).
Outputs: Ciphertext, (n+1) 64-bit values $\{C_0, C_1, \ldots, C_n\}$.

1) Initialize variables
   Set $A^0 = IV$, an initial value
   For $i = 1, \ldots, n$
   $R_i^0 = P_i$
2) Calculate intermediate values
   For $t = 1, \ldots, s$, where $s = 6n$
   $A^t = \mathbf{MSB}_{64}\left(\mathbf{AES}_{\mathrm{K}}\left(A^{t-1} \mid R_1^{t-1}\right)\right) \oplus t$
   For $i = 1, \ldots, n-1$
   $R_i^t = R_{i+1}^{t-1}$
   $R_n^t = \mathbf{LSB}_{64}\left(\mathbf{AES}_{\mathrm{K}}\left(A^{t-1} \mid R_1^{t-1}\right)\right)$
3) Output the results
   Set $C_0 = A^t$
   For $i = 1, \ldots, n$
   $C_i = R_i^t$

Figure 7–1 shows the motion of key wrap algorithm.

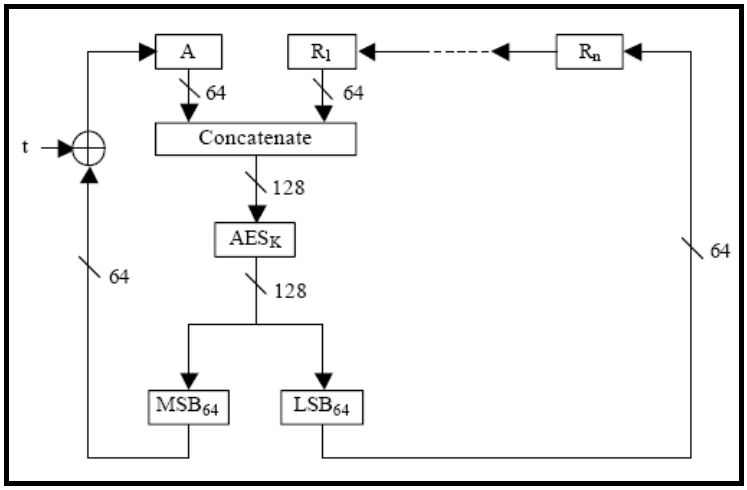


Figure 7–1: Motion of key wrap algorithm

An alternative description of the key wrap involves indexing rather than shifting. This approach allows you to calculate the wrapped key in place, avoiding the rotation in the previous description. This produces identical results and is more easily implemented in software.

Inputs: Plaintext, n 64-bit values $\{P_1, P_2, \ldots, P_n\}$
Key, K (the KEK).
Outputs: Ciphertext, (n+1) 64-bit values $\{C_0, C_1, \ldots, C_n\}$

1) Initialize variables
   Set $A = IV$, an initial value
   For $i = 1, \ldots, n$
   $R_i = P_i$
2) Calculate intermediate values
   For $j = 0, 1, \ldots, 5$
   For $i = 1, 2, \ldots, n$
   $B = \mathbf{AES}_{\mathrm{K}}(A \mid R_i)$
   $A = \mathbf{MSB}_{64}(B) \oplus t$ where $t = (n \cdot j) + i$
   $R_i = \mathbf{LSB}_{64}(B)$
3) Output the results
   Set $C_0 = A$
   For $i = 1, \ldots, n$
   $C_i = R_i$

7.1.3.2. Key Unwrap

The inputs to the unwrap process are the KEK and (n + 1) 64-bit blocks of ciphertext consisting of previously wrapped key. It returns n blocks of plaintext consisting of the n 64-bit blocks of the decrypted key data.

Inputs: Ciphertext $(n+1)$ 64-bit values $\{C_0, C_1, \ldots, C_n\}$,
Key, K (the KEK)
Outputs: Plaintext n 64-bit values $\{P_1, P_2, \ldots, P_n\}$

1) Initialize variables
   Set $A^s = C_0$ where $s = 6n$
   For $i = 1, \ldots, n$
   $R_i^s = C_i$
2) Calculate the intermediate values
   For $t = s, \ldots, 1$
   $A^{t-1} = \mathbf{MSB}_{64}\left(\mathbf{AES}_{\mathrm{K}}^{-1}\left(\left(A^t \oplus t\right) \mid R_n^t\right)\right)$
   $R_1^{t-1} = \mathbf{LSB}_{64}\left(\mathbf{AES}_{\mathrm{K}}^{-1}\left(\left(A^t \oplus t\right) \mid R_n^t\right)\right)$
   For $i = 2, \ldots, n$
3) Output the results
   If A_0 is an appropriate initial value
   Then
   For $i = 1, \ldots, n$
   $P_i = R_i^0$
   Else
   Return an error

Figure 7–2 shows the Motion of key unwrap algorithm

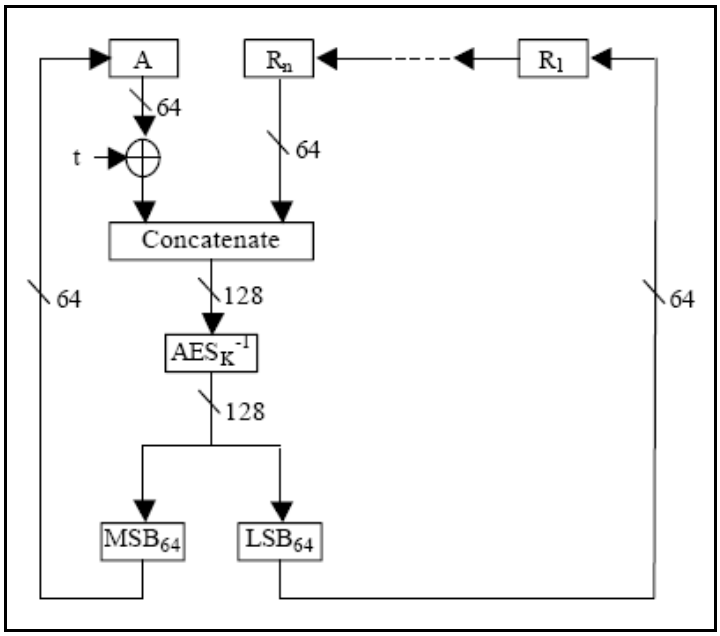


Figure 7–2: Motion of key unwrap algorithm

The key unwrap algorithm can also be specified as an index based operation, allowing the calculations to be carried out in place. Again, this produces the same results as the register shifting approach

Inputs: Ciphertext $(n+1)$ 64-bit values $\{C_0, C_1, \ldots, C_n\}$,
Key, K (the KEK)
Outputs: Plaintext n 64-bit values $\{P_1, P_2, \ldots, P_n\}$

1) Initialize variables
Set $A = C_0$
For $i = 1, \ldots, n$
$R_i = C_i$
2) Compute intermediate values
For $j = 5, \ldots, 0$
For $i = n, n-1, \ldots, 1$
$B = \mathbf{AES}_K^{-1}((A \oplus t) \mid R_i)$, where $t = (n \cdot j) + i$
$A = \mathbf{MSB}_{64}(B)$
$R_i = \mathbf{LSB}_{64}(B)$
3) Output results
If A is an appropriate initial value
Then
For $i = 1, \cdots, n$
$P_i = R_i$
Else
Return an error

7.1.3.3. Key Data Integrity—the Initial Value

The initial value (*IV*) refers to the value assigned to A_0 in the first step of the wrapping process. This value is used to obtain an integrity check on the key data. In the final step of

the unwrapping process, the recovered value of $_0 A$ is compared to the expected value of A_0. If there is a match, the key is accepted as valid, and it is returned by the unwrapping algorithm. If there is not a match, then the key is not accepted as valid, and the unwrapping algorithm returns an error.

The exact properties achieved by this integrity check depend on the definition of the initial value. Different applications may call for somewhat different properties; for example, whether there is need to determine the integrity of key data throughout its lifecycle or just when it is unwrapped. This specification defines a default initial value that supports integrity of the key data during the period it is wrapped. Provision is also made to support alternative initial values, if called for in other NIST publications on key management.

The default initial value (*IV*) is defined to be the hexadecimal constant, $A = IV$ = A6A6A6A6A6A6A6A6. The use of a constant as the *IV* supports a strong integrity check on the key data during the period that it is wrapped. If unwrapping produces A_0= A6A6A6A6A6A6A6A6, then the chance that the key data is corrupt is 2^{-64}. If unwrapping produces $A_0 \neq$ A6A6A6A6A6A6A6A6, then the key unwrap algorithm must return an error and not return any key data.

When the key wrap is used as part of a larger key management protocol or system, the desired scope for data integrity may be more than just the key data or the desired duration for more than just the period that it is wrapped. Also, if the key data is not just an AES key, it may not always be a multiple of 64 bits. Alternative definitions of the initial value can be used to address such problems. NIST will define alternative initial values in future key management publications as needed. In order to accommodate a set of alternatives that may evolve over time, key wrap implementations that are not application-specific will require some flexibility in the way that the initial value is set and tested.

7.2. Hardware Implementation of AES Key Wrap

7.2.1. Hardware Architecture

The key wrap/ unwrap algorithm shown in Figure 7–1 and Figure 7–2 uses the 128 bit AES encryptor/ decryptor with feedback. In our design of the AES key wrap/ unwrap algorithm we will use the loop unrolled architecture for both cipher/ decipher and key expansion (shown in Figure 7–3, Figure 7–4) which achieves the largest throughput in feedback modes. The key wrap/ unwrap top entity is shown in Figure 7–5.

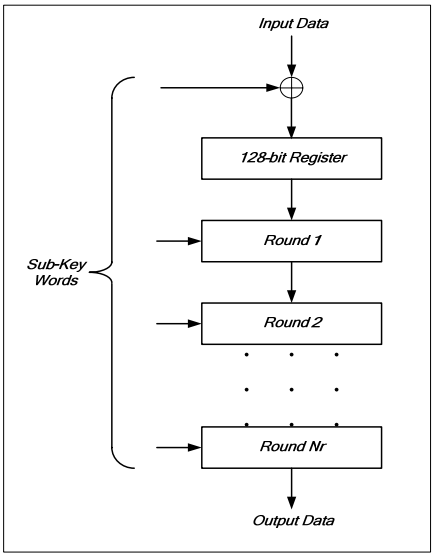


Figure 7–3 : Loop Unrolled Architecture

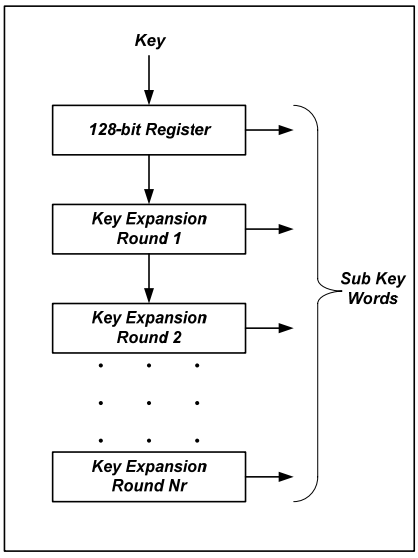


Figure 7–4: Key Expansion Architectur

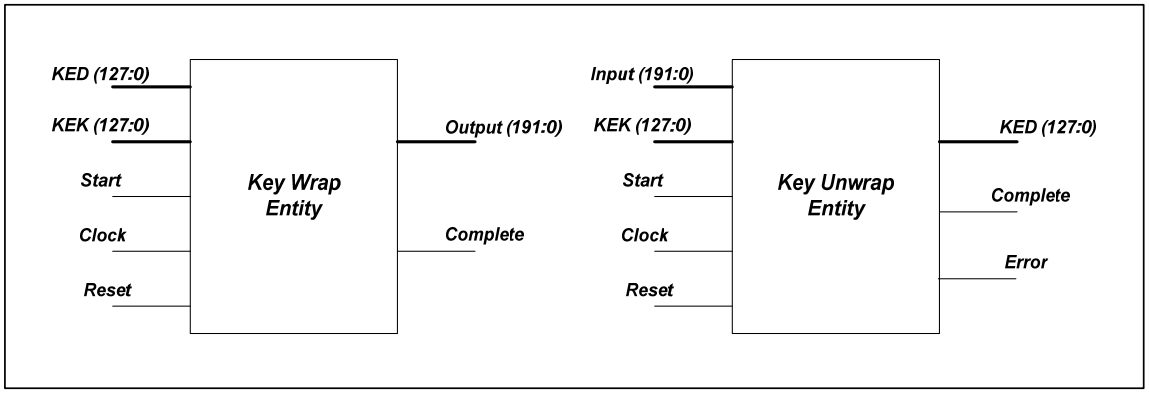


Figure 7–5: Key wrap/ unwrap top level entity

7.2.2. Functional Simulation Results

Figure 7–6 and Figure 7–7 shows the functional simulation results of the AES key wrap/ unwrap algorithm for the test vector data given in the NIST specification [55].

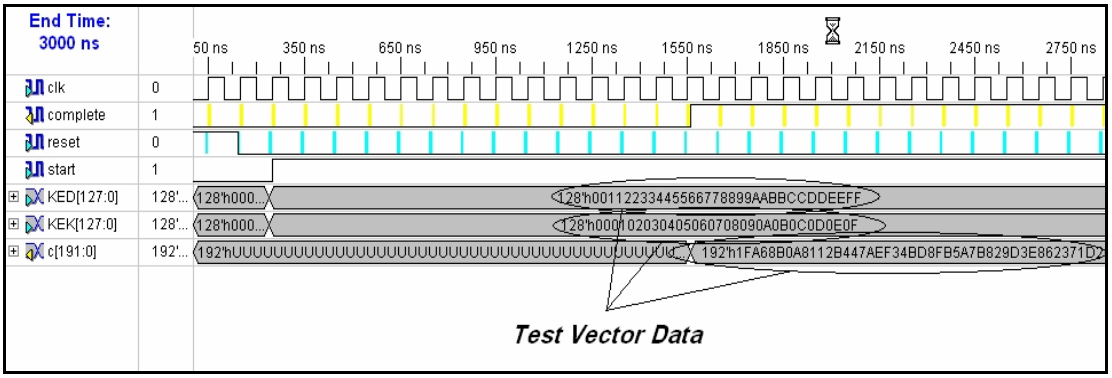


Figure 7–6: Functional simulation results of key wrap algorithm

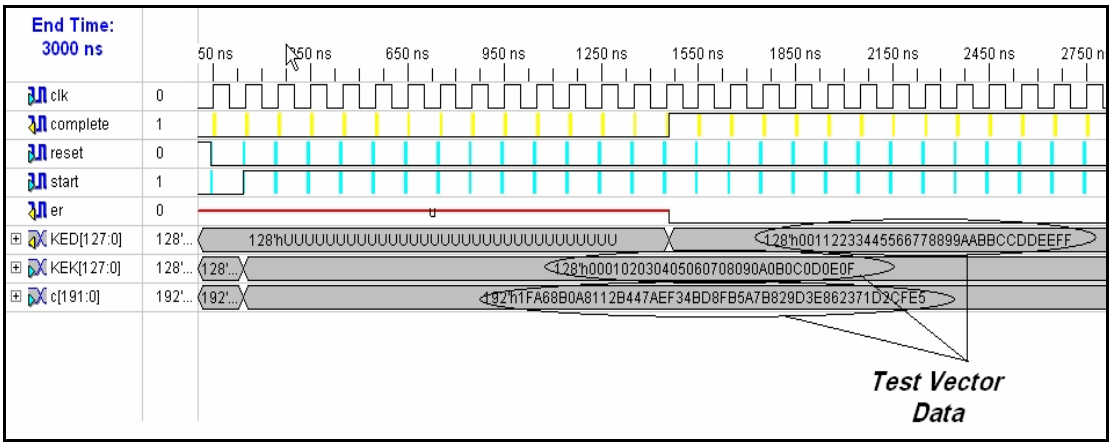


Figure 7–7: Functional simulation of key unwrap algorithm

7.2.3. Hardware Implementation Results

Table 7–1 shows the Implementation results of the key wrap/ unwrap algorithm. It is clear that loop unrolled architecture consumes area less than optimized speed (pipelined architecture) and it yields a throughput more than optimized area (basic architecture). The key wrap area and throughput differs from the key unwrap area and throughput due to the difference in the algorithms of the encryptor and decryptor.

Table 7–1: Implementation results of the key wrap/ unwrap algorithm

	Key wrap	***Key unwrap***
Number of Slices	17897 out of 26624 67%	17414 out of 26624 65%
Number of Slice Flip-Flops	1919 out of 53248 3%	1794 out of 53248 3%
Number of 4 inputs LUTs	34767 out of 53248 65%	33724 out of 53248 63%
Minimum Period (Maximum Frequency)	40.080ns (24.950MHz)	48.351ns (20.682MHz)
Minimum input arrival time before clock	36.873ns	34.630ns
Maximum output required time after clock	3.921ns	3.921ns

$$Key\ Wrap\ Throughput = \frac{128 \times 24.95\,MHz}{1} = 3.194\,Gbps$$

$$Key\ Unwrap\ Throughput = \frac{128 \times 20.682\,MHz}{1} = 2.65\,Gbps$$

7.3. DRBGs Based on AES Block Cipher

Random number generators (RNGs) are required for the generation of keying material (e.g., keys and initialization vectors (IVs)) [58]. There are two classes of RNGs: deterministic and non-deterministic. Deterministic Random bit Generators (DRBGs), sometimes called deterministic random number generators or pseudorandom number

generators, use cryptographic algorithms to generate random numbers. Non-Deterministic Random Bit Generators (NDRBGs), sometimes called true RNGs, produce output that is dependent on some unpredictable physical source that is outside human control, for example, radioactive decay or a true noise hardware randomizer.

There are two fundamentally different strategies for generating random bits. One strategy is to produce bits non-deterministically, where every bit of output is based on a physical process that is unpredictable; this class of random bit generators (RBGs) is commonly known as non deterministic random bit generators (NRBGs). The other strategy is to compute bits deterministically using an algorithm; this class of RBGs is known as Deterministic Random Bit Generators (DRBGs). A block cipher DRBG is based on a block cipher algorithm. The block cipher DRBG mechanism specified in this Recommendation has been designed to use any Approved block cipher algorithm and may be used by consuming applications requiring various security strengths, providing that the appropriate block cipher algorithm and key length are used, and sufficient entropy is obtained for the seed.

This section describes two classes of DRBGs based on block ciphers [56]: One class uses the block cipher in OFB mode; the other class uses the CTR mode. There are no practical security differences between these two DRBGs; CTR mode guarantees that short cycles cannot occur in a single output request, while OFB- mode guarantees that short cycles will have an extremely low probability. OFB mode makes slightly less demanding assumptions on the block cipher, but the security of both DRBGs relates in a very simple and clean way to the security of the block cipher in its intended applications. This is a fundamental difference between these DRBGs and the DRBGs based on hash functions, where the DRBG's security is ultimately based on pseudo randomness properties that don't form a normal part of the requirements for hash functions. An attack on any of the hash based DRBGs does not necessarily represent a weakness in the hash function; however, for these block cipher-based constructions, a weakness in the DRBG is directly related to a weakness in the block cipher .

Specifically, suppose that there is an algorithm for distinguishing the outputs of either DRBG from random with some advantage. If that algorithm exists, it can be used to build a new algorithm for distinguishing the block cipher from a random permutation, with the same time and memory requirements and advantage.

Because there is no practical security difference between the two classes of block-cipher based DRBGs, the choice between the two constructions is entirely a matter of implementation convenience and performance. An implementation that uses a block cipher in OFB, CBC, or full-block CFB mode can easily be used to implement the OFB based DRBG construction; an implementation that already supports counter mode can reuse that hardware or software to implement the counter-mode DRBG. In terms of performance, the CTR-mode construction is more amenable to pipelining and parallelism, while the OFB- mode construction seems to require slightly less supporting hardware.

In this section we will introduce the two DRBGs based on the 128 bit AES block cipher (CTR DRBG and OFB DRBG) [57].

7.3.1. DRBG Based on AES in CTR Mode

Figure 7–8 shows the DRBG based on the AES block cipher in CTR mode [57]. The initial 128-bit seed is loaded into the seed register. This seed forms the initial value of the counter register. Every clock cycle the counter value is incremented. The counter value is loaded into the AES unit and it is encrypted. The encrypted value is the generated random number that is written out. Starting form a secure non-repeating initial seed, 2^{128}sequences of 128-bit random numbers are generated. Moreover, the throughput rate of the random number generation is equal to maximum throughput of the AES algorithm.

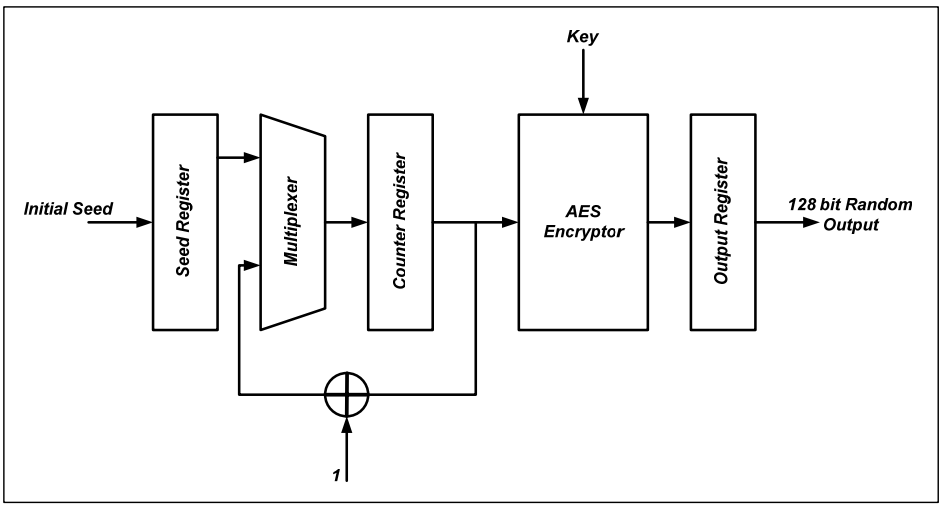


Figure 7–8: Counter mode AES DRBG

7.3.2. DRBG Based on AES in OFB Mode

Figure 7–9 shows the DRBG based on the AES block cipher in OFB mode [57]. The initial 128-bit seed is loaded into the seed register. This seed forms the initial value of the encryptor. Every clock cycle the encryptor output is feedback to the input register. The input value is loaded into the AES unit and it is encrypted. The encrypted value is the generated random number that is written out. Starting form a secure non-repeating initial seed, 2^{128} sequences of 128-bit random numbers are generated. Moreover, the throughput rate of the random number generation is equal to maximum throughput of the AES algorithm.

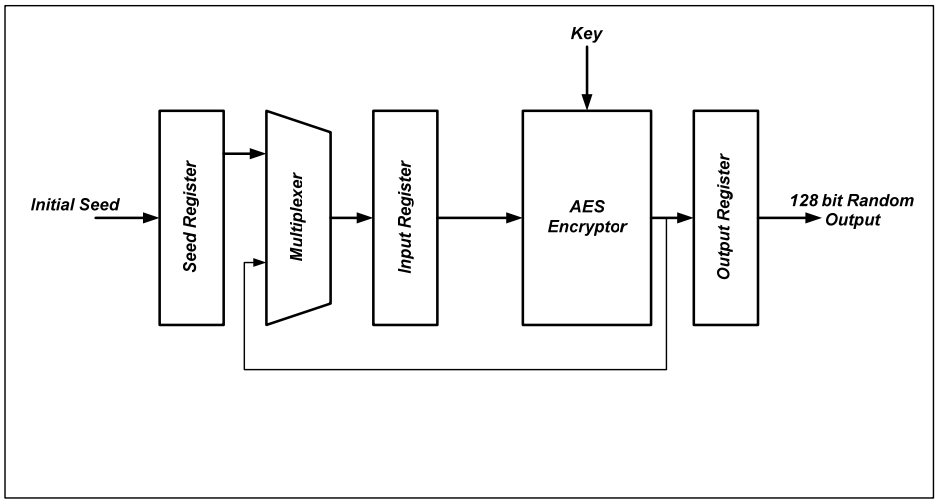


Figure 7–9: OFB mode AES DRBG

7.4. Hardware Implementation of the AES DRBG

7.4.1. Hardware Architecture

Any architecture of the AES hardware architectures mentioned in section 4.1.1 could be used to implement the DRBG based on the AES CTR mode. We will implement it based on the pipelined architecture (Figure 6–13-a) to achieve a high throughput.

For the DRBG based on the AES in the OFB mode the most suitable architecture is the loop unrolled architecture shown in Figure 7–3 which compromise between relatively high throughput (higher than basic architecture) and relatively low area (smaller than pipelined architecture).

Figure 7–10 shows the AES based DRBG top level entity for both CTR and OFB modes.

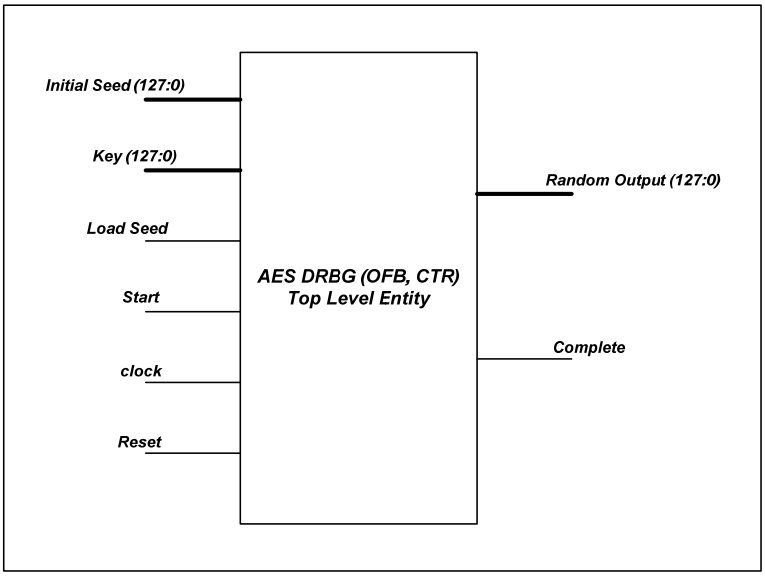


Figure 7–10: AES DRBG top level entity

7.4.2. Functional Simulation Results

Figure 7–11 and Figure 7–12 shows the functional simulation results of the AES CTR and OFB based DRBGs for the test vector key and initial seed [10].

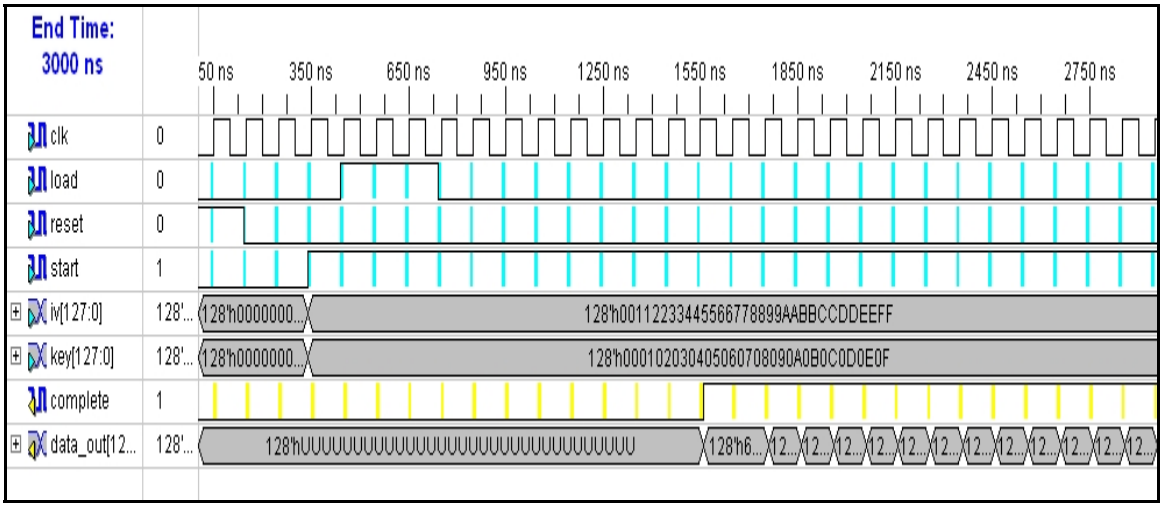


Figure 7–11: Functional simulation of the AES CTR DRBG

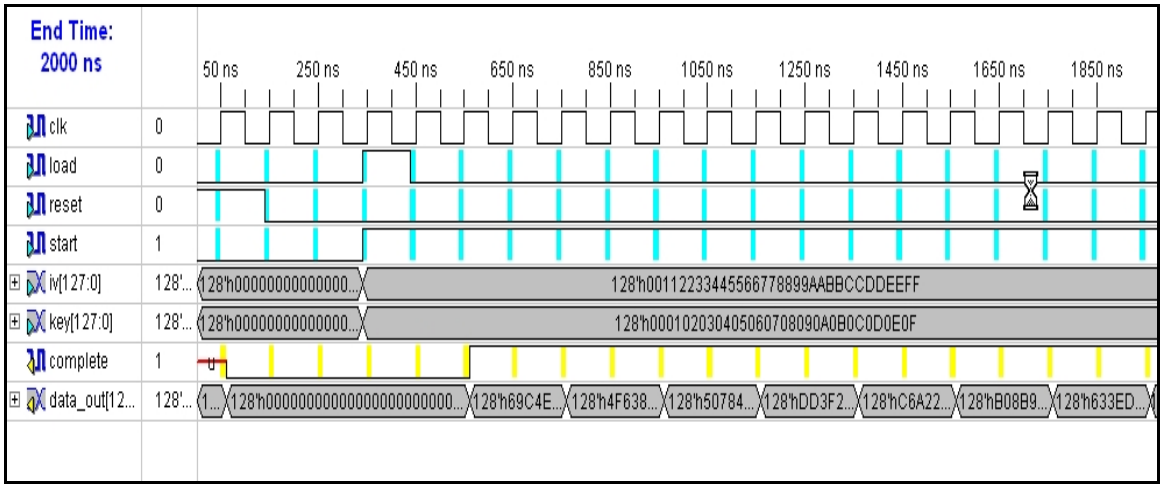


Figure 7–12: Functional simulation of the AES OFB DRBG

7.4.3. Hardware Implementation Results

Table 7–2 shows the implementation results of the DRBG based on the AES in CTR and OFB modes of operation. It is clear that the CTR DRBG has a high throughput and area and the OFB DRBG has area and throughput smaller than the CTR DRBG. We can use on of the two DRBGs in the applications required high speed or low area. The difference between the two DRBGs is mainly due to the difference of the number of registers between pipelined architecture and loop unrolled architecture.

Table 7–2: Implementation results of the DRBG based on the AES in CTR and OFB modes

	CTR DRBG	***OFB DRBG***
Number of Slices	16231 out of 26624 60%	17914 out of 26624 67%
Number of Slice Flip-Flops	5281 out of 53248 9%	1668 out of 53248 3%
Number of 4 inputs LUTs	31231 out of 53248 58%	34764 out of 53248 65%
Minimum Period (Maximum Frequency)	6.732ns (148.536MHz)	39.599ns (25.253MHz)
Minimum input arrival time before clock	5.111ns	36.867ns
Maximum output required time after clock	3.921ns	3.935ns

$$CTR\ DRBG\ Throughput = \frac{128 \times 148.536\,MHz}{1} = 19.012\,Gbps$$

$$OFB\ DRBG\ Throughput = \frac{128 \times 25.253\,MHz}{1} = 3.232\,Gbps$$

Chapter 8

CONCLUSION AND FUTURE WORK

Cryptography plays an important role in the security of data transmission. Different applications of the AES algorithm may require different speed/area trade-offs. Some applications, such as smart cards and cellular phones, require small area. Other applications, such as WWW servers and ATMs, are speed critical. Some other applications, such as digital video recorders, require an optimization of speed/area ratio. Various optimizations for implementation are developed to suit the different demands of applications. Architectural optimizations make use of duplicating the round units, while algorithmic optimizations explore algorithm simplification inside each encryption/ decryption round unit.

In chapter 2 we give a brief introduction to the cryptography and its types. We focus on the private key cryptography and its modes of operation. Also, we have introduced a brief mathematical background about the finite fields which is used in the AES. After this, a complete explanation of the AES algorithm (cipher, decipher and key expansion algorithms) has been introduced.

In chapter 3 we have introduced the heuristic techniques used in the design of the cryptographic substitution boxes (S-boxes). In this chapter, we introduce the cryptographic properties of a good Boolean function and S-box (high nonlinearity and low autocorrelation). We use heuristic optimization algorithms such as hill climbing, simulated annealing, tabu search and genetic algorithms to find a solution for the S-box problems and we introduce S-box with good cryptographic properties (compared with the recent research results cited in this thesis) which could be used as AES S-box.

Chapter 4 has explored various Architectural and algorithmic optimization approaches for efficient hardware implementations of the AES algorithm. In this chapter, we introduce various AES hardware architectures which could be used to meet the various implementation goals (area and speed). Also, a full description of the MixColumns transformation has been introduced. The implementation of the whole round unit as S-box has been covered by this chapter. In addition, the joint implementation issues of the encryptor and decryptor are also discussed and compared.

In Chapter 5, various methods used for efficient hardware implementation of the AES S-box have been introduced. In this chapter, we select the look up table method to implement an AES S-box with high speed and the finite field arithmetic (mapping from $GF(2^8)$ to $GF(2^4)$) to implement an AES S-box with low area.

The goals of this thesis were to implement a low area and a high speed AES encryptor and decryptor using various optimization techniques and to implement AES crypto processor with serial interface with external peripherals on FPGA. These goals have been met. In chapter 6, optimized area and optimized speed AES encryptor and decryptor and AES crypto processor are completed, simulated and verified. The code was written in

VHDL'93 and synthesized and verified using the Xilinx ISE 7.1 program and simulated using the Modelsim program.

Optimized area AES (encryptor, decryptor) have been implemented based on the basic architecture and it consumes *(1468, 2752 Xilinx slices)* and operates at *(1.664, 1.558 Gbps)*. Optimized speed AES (encryptor, decryptor) have been implemented based on the basic architecture and it consumes *(18855, 20155 Xilinx slices)* and operates at *(28.51, 23.09 Gbps*), which was greater than other works cited in this thesis.

In chapter 7 we have been introduced two applications of the AES. The first application was the AES key wrap/ unwrap algorithms based on the loop unrolled architecture and it consumes (*17897, 17414 Xilinx slices)* and operates at (*3.19, 2.65 Gbps*). The second application was the deterministic random bit generator (DRBG) based on the AES in counter (CTR) and output feedback mode (OFB). The CTR DRBG has been implemented based on the pipelined architecture and it consumes (*16231 Xilinx slices*) and operates at (*19 Gbps*).The OFB DRBG has been implemented based on the loop unrolled architecture and it consumes *(17914 Xilinx slices*) and operates at (*3.23 Gbps*).

There are several opportunities for future work as a result of this thesis:

1. Improve the cryptographic properties of the S-boxes is an urgent issue. Until now all the heuristic optimization methods didn't reach the desired cryptographic properties which are achieved using algebraic methods. Other optimization methods and cost functions could be used to improve the cryptographic properties of the S-boxes.
2. For both optimized area and speed AES on FPGA we found that the FPGA routing delay plays an important role in the maximum operating frequency, which could be reduced in ASIC design.
3. Another work that could be done is the study of optimization approaches for the implementations supporting multiple key lengths and modes of operation.
4. For the optimized speed AES, the look up implementation of the whole round unit could be applied to reduce the delay caused by the internal transformations.
5. For the optimized area AES, The joint hardware implementation of AES encryptor and decryptor to save the area consumed by separate implementations of encryptor and decryptor.
6. Also we can study other implementation approaches for the SubBytes and MixColumns transformations (which consume most of the chip area) to reduce the area.

REFRENCES

[1] P.C. Van Oorschot A.J. Menezes and S. A. Vanstone, 'Handbook of applied cryptography", CRC Press, Waterloo, Ontario, Canada, 2001.

[2] K. Aoki1 and H. Lipmaa, "Fast implementations of AES candidates, Third AES Candidate Conference" (New York City, USA), April 2000

[3] R.Ashruf, "The AES targeted on MOLEN processor", Ms thesis Delft university of technology , 2004.

[4] A.J. Elbirt W. Yip B. Chetwynd and C. Paar, "An FPGA implementation and performance evaluation of the aes block cipher candidate algorithm finalists", Third AES Candidate Conference (AES3), April 2000.

[5] N. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations", Asiacrypt 2002, December 2002.

[6] C. Kessler Gary, "An Overview of Cryptography", Handbook on Local Area Networks, Auerbach, 1998.

[7] S. William , "Cryptography and Network Security Principles and Practices", Fourth Edition, Prentice Hall, November 16, 2005

[8] D. Morris, "Recommendation for Block Cipher Modes of Operation Methods and Techniques", NIST Special Publication 800-38A 2001 Edition http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[9] J. Daemen and R.Rijmen, "AES Proposal: Rijndael", NIST AES proposal version 2, 1999. http://www.esat.kuleuven.ac.be/~rijmen/ rijndael.

[10] "Advanced Encryption Standard(AES)", Federal Information Processing Standards Publication 197, November 26, 2001

[11] Linda B, "Heuristic Design of Boolean Functions and Substitution Boxes for Cryptography", Ph.D Queensland University of Technology, 2005.

[12] A. C. John and L. J. Jeremy and Suzan S, "The Design of S-Box by Simulated Annealing",In CEC 2004: International Conference on Evolutionary Computation, Portland OR, USA, June 2004, pages 1533.1537. IEEE, 2004.

[13] M. Matsui. "Linear Cryptanalysis Method for DES Cipher". In Tor Helleseth, editor, Advances in Cryptology - EuroCrypt '93, pages 386– 397, Berlin, 1993. Springer-Verlag. Lecture Notes in Computer Science Volume 765.

[14] E. Biham and A. Shamir. "Differential Cryptanalysis of DES-like Cryptosystems" (Extended Abstract). In Alfred J. Menezes and Scott A. Vanstone, editors, Advances in Cryptology - Crypto '90, pages 2–21, Berlin, 1990. Springer-Verlag. Lecture Notes in Computer Science Volume 537.

[15] H. Heys. "A tutorial on linear and differential cryptanalysis". Technical report, Electrical and Computer Engineering, University of Newfoundland, St. John's, Newfoundland, Canada.

[16] W. Millan. "How to Improve the Non-linearity of Bijective S-boxes". In C. Boyd and E. Dawson, editors, 3rd Australian Conference on Information Security and Privacy, pages 181–192. Springer-Verlag, April 1998. Lecture Notes in Computer Science Volume 1438.

[17] W. Millan, L. Burnett, G. Carter, A. Clark, and E. Dawson. "Evolutionary Heuristics for Finding Cryptographically Strong S-Boxes". In ICICS 99, 1999.

[18] J. A. Clark, J. L. Jacob, Susan Stepney, S. Maitra, and W. Millan. "Evolving Boolean Functions Satisfying Multiple Criteria". In Progress in Cryptology - INDOCRYPT 2002, pages 246–259. Springer Verlag LNCS 2551, 2002.

[19] S. Maitra J. A. Clark, J. L. Jacob and P. Stanica. “Almost Boolean functions: the Design of Boolean Functions by Spectral Inversion.” In Conference on Evolutionary Computation —CEC-03, December 2003.
[20] W. Millan, A. Clark, and E. Dawson, “Boolean function design using hill climbing methods”. In Information Security and Privacy, ACISP '99, volume 1587 of Lecture Notes in Computer Science, pages 1{ 11. Springer Verlag, 1999.
[21] A. Canteaut, C. Carlet, P. Charpin, and Caroline Fontaine. Propagation characteristics and correlation-immunity of highly nonlinear Boolean functions. In Advances in Cryptology - EUROCRYPT 2000, volume 1807 of Lecture Notes in Computer Science, pages 507{522. Springer Verlag,2000).
[22] P. Chardaire, J. C. Lutton, and A. Sutter.” Thermostatistical Persistency: A Powerful Improving Concept for Simulated Annealing”. European Journal of Operations Research, 86:565–579, 1995.
[23] C. Blum and A. Roli. “Metheuristics in Combinatorial Optimization: Overview and Conceptual Comparison”. Research Report TR/IRIDIA/2001-13, Institut de Recherches Interdisciplinaires et de Developpements en Intelligence Artificielle (IRIDIA), Universit´e Libre de Bruxelles, 2001.
[24] C. R. Reeves, editor. “Modern Heuristic Techniques for Cominatorial Problems”. McGraw Hill, 1995.
[25] J. H. Holland. “Adaptation in Natural and Artificial Systems”. University of Michigan Press, 1975.
[26] L. Davis. “Handbook of Genetic Algorithms”. Van Nostrand Reinhold, January 1991.
[27] Z. Xinamiao and K. Parhi, “Implementation Approaches for The Advanced Encryption Standard”, Circuit and System Magazine, Volume 2, Number 4, Fourth Quarter 2002.
[28] J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, “An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalist”, the Third AES Conference (AES3), New York, April 2000.
http:// csrc.nist.gov/encryption/aes/round2/ conf3/aes3papers.html.
[29] K. Gaj and P. Chodowiec, “Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware”, The Third AES Conference (AES3), New York, April 2000.
http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.
[30] M. McLoone and J. V. McCanny, “Rijndael FPGA Implementation Utilizing Look-Up Tables”, IEEE Workshop on Signal Processing Systems, pp. 349–360, September 2001.
[31] M. McLoone and J. V. McCanny, “Rijndael FPGA Implementation Utilizing Look-Up Tables”, IEEE Workshop on Signal Processing Systems, pp. 349–360, September 2001.
[32] T. Ichikawa, T. Kasuya, and M. Matsui, “Hardware Evaluation of the AES Finalists”, The Third AES Conference (AES3), New York, April 2000. http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.
[33] C. C. Lu and S. Y. Tseng, “Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter”, IEEE Transactions on Information Theory, vol. 37, no. 5, pp. 1241–1260, September 1991.
[34] H. Kuo and I. Verbauwhede, “Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm”, Proceedings CHES 2001, pp.

51–64, Paris, France, May 2001.
[35] V. Fischer and M. Drutarovsky, “Two Methods of Rijndael Implementation in Reconfigurable Hardware”, Proceedings CHES 2001, pp. 77–92, Paris, France, May 2001.
[36] V. Fischer, “Realization of the Round 2 Candidates Using Altera FPGA”, The Third AES Conference (AES3), New York, Apr. 2000. http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html.
[37] A.Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, “Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic”, Proceedings CHES 2001, pp. 171–184, Paris, France, May 2001.
[38] V. Rijmen, “Efficient Implementation of the Rijndael S-box”, http:// www.esat.kuleuven.ac.be/~rijmen/rijndael.
[39] K. K. Parhi, “VLSI Digital Signal Processing Systems Design and Application”, John Wiley & Sons, pp. 559–562, 1999.
[40] J. Daemen and R.Rijmen, “Rijndael: The Advanced Encryption Standard”, Dr. Dobb’s Journal, pp. 137–139, March 2001.
[41] M. McLoone and J.V.McCanny, “High Performance Single-Chip FPGA Rijndael Algorithm Implementation”, Proceedings CHES 2001, pp. 65–76, Paris, France, May 2001.
[42] N. Weaver and J. Wawrzynek, “A Comparison of the AES Candidates Amenability to FPGA Implementation”, The Third AES Conference (AES3), New York, April 2000. http://csrc.nist.gov/ encryption/aes/round2/conf3/aes3papers.html.
[43] K. Araki, I. Fujita and M. Morisue, "Fast Inverter Over Finite Fields Based on Euclid's Algorithm," Trans. IEICE, Vol. E-72, No. 11, pp. 1230–1234, Nov. 1989.
[44] H. Brunner, A. Curiger, and M. Hofstetter, "On Computing Multiplicative Inverse in GF(2m) ," IEEE Trans. on Computer, Vol. 42., No. 8, pp. 1010–1015,Aug. 1993.
[45] C. Paar, "A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields," IEEE Trans. on Comp., Vol. 45, No. 7, pp 856–861, 1996.
[46] C. Paar, "Fast Arithmetic Architecture for Public Key Algorithms over Galois Fields GF((2n)m)," Proc. EUROCRYPT'97, LNCS Vol. 1233, Springer-Verlag, pp. 363-378, 1997.
[47] A. Rudra et. al., "Efficient Implementation of Rijndael Encryption with Composite Field Arithmetic," Proc. CHES 2001, LNCS Vol. 2162, pp. 175-188, 2001.
[48] S. Morioka and A. Satoh, "An Optimized S-box Circuit Architecture for Low Power AES Design," Proc. CHES 2002, LNCS Vol. 2523, pp. 172-186, 2003.
[49] C. Jutla, V. Kumar and A. Rudra, "On the Complexity of Isomorphic Galois Field Transformations," IBM Research Report, Vol. RC22652 (W0211-243), November 2002.
[50] E.D. Mastrovito, "VLSI Architecture for Computations in Galois Fields," Dept. Electrical Engineering, Linkoping University, Sweden, 1991.
[51] S. Chantarawong, P. Noo-intara, and S. Choomchuay,” An Architecture for S-Box Computation in the AES”, Proc of Information and Computer Engineering Workshop 2004 (ICEP2004), Prince of Songkla University (Phuket Campus), January 2004, pp.157-162.
[52] D. J. Smith, ”HDL Chip Design”, second edition, Doone Pubns (March 1998).

[53] P. R. Chodowiec, “Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware”, Master Thesis George Mason University, 2002.
[54] E. Oswald, “State of the Art in the Hardware Architecture”, European Network of Excellence in Cryptology, 2005.
[55] “AES Key Wrap Specification”, National Institute of Standards and Technology, November 2001.
[56] “Deterministic Random Bit Generator Based on Block Ciphers”, American National Standards Institute, ANSI X9.82, Part 3, July 2004.
[57] J. Kelsey, “Five DRBG Algorithms Based on Hash Functions and Block Ciphers”, National Institute of Standards and Technology, July 2004.
[58] E. Barker and J. Kelsey, “Recommendation for Random Number Generating Using Deterministic Random Bit Generators”, National Institute of Standards and Technology, Special Publication 800-90, June 2006.

ملخص عربي

نظام التتشفير المتقدم AES هو المعيار الجديد للتشفير و قد كسب تأييداً واسعاً ليصبح الوسيلة المناسبة لتأمين و حماية البيانات الرقمية. في هذه الرسالة قمنا باستعراض طرق و تقنيات تنفيذ نظام التشفير المتقدم على مصفوفة البوابات القابلة للبرمجة FPGA. أيضاً قمنا بتقديم طرق الأمثلة المستخدمة في تطوير صناديق التعويض S-box المستخدمة في نظام التشفير المتقدم و قمنا باقتراح صندوق تعويض جديد ذو خصائص تشفير جيدة. الخيارات المتاحة بين السرعة و المساحة الملازمة لتصميم معالج آمن قمنا أيضاً باستعراضها. قمنا في هذا البحث بتنفيذ تصميمين مختلفين لنظام التشفير المتقدم التصميم الأول معتمد على البنية الأساسية لنظام التشفير المتقدم و التصميم الثاني معتمد على النظام المعماري المنقول لنظام التشقير المتقدم وقمنا بتصميم و تنفيذ معالج امني ذو وصلة تسلسلية من الممكن استخدامه مع أي تصميم قمنا بتقديمه لنظام التشفير المتقدم. أيضا قمنا بتنفيذ تطبيقين مختلفين لنظام التشفير المتقدم في وضع التغذية الخلفية. قمنا في هذه الرسالة بإعطاء نتائج محاكاة و نتائج تنفيذ كاملة لكلٍ تصميم من التصميمات الواردة في الرسالة و قد قمنا بترتيب هذا البحث على النحو التالي:

في الفصل الأول مقدمة مختصرة عن نظام التشفير المتقدم و التطبيقات التي يستخدم فيها و مستلزماتها من سرعة أو مساحة و طرق تنفيذه على كل من البرمجيات و الشرائح الإلكترونية المعدة لغرض محدد و مصفوفات البوابات القابلة للبرمجة.

الفصل الثاني يوفر مدخلاً عاماً إلى علم التشفير و أنواعه مع التركيز على التشفير بمفتاح خاص و أوضاع عمله و يوفر أيضاً شرحاً مفصلاً لنظام التشفير المتقدم مع خلفية رياضية مختصرة عن النظام.

الفصل الثالث يناقش الخواص الأمنية الجيدة التي يجب توافرها في صناديق التعويض المستخدمة في نظام التشفير المتقدم و طرق الأمثلة التجريبية المستخدمة لتطوير هذه الخواص لدالة ثنائية واحدة و لصناديق التعويض.

الفصل الرابع يناقش الطرق المختلفة المستخدمة لتنفيذ نظام تشفير متقدم ذو كفاءة عالية من وجهة النظر المعمارية و الوظيفية حيث السرعة و المساحة هي أهداف الأمثلة في هذا الفصل. أيضاً قمنا بمناقشة مسألة مشاركة الموارد بين آلة التشفير و آلة حل الشفرة.

الفصل الخامس يستعرض طرق التصميم المختلفة لصناديق التعويض و التي تعتبر أكبر وظيفة تستهلك الوقت و المساحة في نظام التشفير المتقدم. هذا الفصل يعرض شرحاً مفصلاً لطريقة التنفيذ التي قمنا باستخدامها في تصميمنا لنظام التشفير المتقدم مع عرض مقارنة بين الطرق المختلفة المستخدمة.

الفصل السادس يقدم التنفيذ العملي و نتائج المحاكاة لنظام التشفير المتقدم ذو المساحة المثلى و نظام التشفير المتقدم ذو السرعة المثلى مستخدمين ما تم عرضه من طرق التصميم الواردة في الفصلين الرابع و الخامس و قمنا بتقديم مقارنة بين تنفيذنا لنظام التشفير المتقدم و تنفيذات سابقة لنفس النظام. أخيراً قمنا بتقديم نتائج المحاكاة و التصميم العملي للمعالج الأمني القائم على نظام التشفير المتقدم ذو الوصلة التسلسلية و الذي يمكن استخدامه في الاختبار العملي لأي من التصميمات التي قمنا بتنفيذها.

الفصل السابع يستعرض تطبيقين لنظام التشفير المتقدم و تصميمهم العملي على مصفوفة البوابات القابلة للبرمجة. التطبيق الأول هو الطريقة المستخدمة لتغطية مفتاح نظام التشفير المتقدم والتي يمكن استخدامها لنقل مفتاح الشفرة في قناة اتصال غير مؤمنة و التطبيق الثاني هو مولد الأرقام الثنائية العشوائي المحدد. نظام الحلقة المبسطة استخدم في تنفيذ نظام التشفير المتقدم في وضع التغذية الخلفية.

الفصل الثامن يستعرض ما تم تقديمه في هذه الرسالة مع تصور مستقبلي لما يمكن أن يتم عمله بناءً على هذه الرسالة.

لجنة الإشراف|	موافقون
د| محمد رزق محمد رزق كلية الهندسة , جامعة الإسكندرية	.....
د| حنان حسني كلية الهندسة , جامعة الإسكندرية	.....
د| هانية فرج كلية الهندسة , جامعة الإسكندرية	.....

التنفيذ العملي لنظام التشفير المتقدم
على مصفوفة البوابات القابلة للبرمجة

مقدمة من

محمد مرسي نعيم فرج

للحصول على درجة الماجستير في

الهندسة الكهربية

لجنة المناقشة و الحكم على الرسالة: **موافقون**

أ. د. حسن محمد الكمشوشي .....
كلية الهندسة , جامعة الإسكندرية

أ. د. مجدي فكري رجائي .....
كلية الهندسة , جامعة القاهرة

د. محمد رزق محمد رزق .....
كلية الهندسة , جامعة الإسكندرية

وكيل الكلية لشئون الدراسات العليا و البحوث

أ. د. حسام محمد فهمي غانم .....

جامعة الإسكندرية
كلية الهندسة

التنفيذ العملي لنظام التشفير المتقدم على مصفوفة البوابات القابلة للبرمجة

رسالة علمية
مقدمة إلى الدراسات العليا بكلية الهندسة- جامعة الإسكندرية
استيفاءاً جزئياً للدراسات المقررة للحصول على درجة

الماجستير

في
الهندسة الكهربية

مقدمة من

المهندس/ محمد مرسي نعيم فرج

2006