DESIGN AND SYSTEMC IMPLEMENTATION OF A CRYPTO PROCESSOR FOR AES AND DES ALGORITHMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TUFAN EGEMEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2007

Approval of the thesis:

DESIGN AND SYSTEMC IMPLEMENTATION OF A CRYPTO PROCESSOR FOR AES AND DES ALGORITHMS

submitted by Tufan Egemen in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Electronics Engineering, Middle East Technical University by,

Date: 05.12. 2007

ABSTRACT

DESIGN AND SYSTEMC IMPLEMENTATION OF A CRYPTO PROCESSOR FOR AES AND DES ALGORITHMS

Egemen, Tufan

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Prof. Dr. Murat Aşkar

December 2007, 129 pages

This thesis study presents design and SystemC implementation of a Crypto Processor for Advanced Encryption Standard (AES), Data Encryption Standard (DES) and Triple DES (TDES) algorithms. All of the algorithms are implemented in single architecture instead of using separate architectures for each of the algorithm. There is an Instruction Set Architecture (ISA) implemented for this Crypto Processor and the encryption and decryption of algorithms can be performed by using the proper instructions in the ISA.

A permutation module is added to perform bit permutation operations, in addition to some basic structures of general purpose micro processors. Also the Arithmetic Logic Unit (ALU) structure is modified to process some crypto algorithm-specific operations.

The design of the proposed architecture is studied using SystemC. The architecture is implemented in modules by using the advantages of SystemC in modular structures. The simulation results from SystemC are analyzed to verify the proposed design. The instruction sets to implement the crypto algorithms are presented and a detailed hardware synthesis study has been carried out using the tool called SystemCrafter.

Keywords: AES, DES, TDES, Crypto Processor, Encryption, Bit Permutation

ÖZ

AES VE DES ALGORİTMALARI İÇİN BİR KRİPTO İŞLEMCİSİ TASARIMI VE SYSTEMC İLE GERÇEKLENMESİ

Egemen, Tufan

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Murat Aşkar

Aralık 2007, 129 sayfa

Bu tezde, İleri Şifreleme Standardı (AES), Veri Şifreleme Standardı (DES) ve Üçlü Veri Şifreleme Standardı (TDES) algoritmaları için bir Kripto İşlemcisi tasarımı ve SystemC gerçekleştirimi sunulmaktadır. Her bir algoritma için ayrı bir yapı kullanmak yerine, üç algoritma da tek bir yapı içerisinde gerçekleştirilmiştir. Kripto işlemcisi için ayrı bir Komut Küme Yapısı (ISA) oluşturulmuştur; şifreleme ve çözme algoritma işlemleri bu Komut Küme Yapısındaki uygun komutların kullanımı ile yapılabilir.

Genel amaçlı mikro işlemcilerdeki bazı temel yapılara ek olarak, bit permütasyon işlemlerini gerçekleştirmek üzere bir permütasyon modülü eklenmiştir. Bunun yanında Aritmetik Mantık Birimi (ALU) yapısı da kullanılan bazı kripto algoritmalarına has fonksiyonları işlemek için değiştirilmiştir.

Önerilen yapının tasarımı SystemC kullanılarak çalışılmıştır. Bu yapı SystemC'nin modüler yapılardaki avantajlarını kullanan modüller halinde gerçeklenmiştir. SystemC'den elde edilen simülasyon sonuçları, önerilen tasarımın doğruluğunu kontrol etmek için analiz edilmiştir. Kripto algoritmalarını gerçeklemek için Komut seti sunulmuş ve SystemCrafter adlı program kullanılarak detaylı bir donanım sentez çalışması yapılmıştır.

Keywords: AES, DES, TDES, Kripto İşlemci, Şifreleme, Bit Permütasyon

To My Family

ACKNOWLEDGEMENTS

I would like to express my special thanks to my supervisor Prof. Dr. Murat Aşkar for his guidance and great support in the development of this thesis work.

I would also like to thank my dear family for their support, understanding and encouragement during this thesis work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Cryptography is the science of encryption and decryption of data. With the help of cryptography, people aim to hide some important information as secret. Generally, cryptography is used for the privacy of information, while it facilitates communication between two points. This requirement can be realized by encrypting the plaintext data with a key to a ciphertext, and then decrypting the ciphertext back to its original form on the other side of the communication channel. Nowadays authentication, digital signatures, and secure computation are other important application areas of cryptography.

The most commonly used crypto algorithms are the Advanced Encryption Algorithm (AES) [1] - [3], which is the standard announced for block ciphers, the previous Data Encryption Standard (DES) [4], and Triple Data Encryption Algorithm (TDEA), also known as Triple DES (TDES) [4] algorithm.

The designs for crypto systems are generally implemented using a specific algorithm and using special hardware architecture which is dedicated to that algorithm. With such architecture, it is much easier to configure the hardware according to the desired specification; hence the crypto algorithm process is much faster.

There are several strategies to make the design of architecture specific to the algorithm. The area and the throughput of the chip are the main parameters while determining the structure of the design according to the desired specification.

To maximize the throughput of the selected algorithm when there is no area constraint in the design, all the iterated rounds of the algorithm can be implemented in the chip layout. For example as given in [9] [10], for AES, the throughput can be increased with inner-round and outer-round pipeline structure. The data path of the structure is also an important parameter in the algorithm-specific design. The data path can be set to the input plaintext length for fast applications, or it can be set to smaller data lengths for area limited operations. There are many categories in the market, which have different data path characteristics. In [15], the data path and bit length discussions are presented.

Another parameter for the algorithm-specific designs is the key schedule part. The key schedule can be arranged as on-the-fly key generation method, which produces the keys in each clock, simultaneously with the round process. Therefore, it is not necessary to use internal registers for the round keys, as explained for AES in [6] [13]. The second key schedule method produces all the round keys before encryption or decryption process and then performs the algorithm’s round operations. This method requires storage registers for the round keys.

Basic crypto operations of algorithms can be executed on using general purpose micro processors. But in general purpose processors, there are no special instructions, or any special block to perform cryptographic operations, making it difficult to process a crypto algorithm in a general purpose processor. Besides general purpose processors, there are crypto processors, which are designed for crypto operations and have crypto specific blocks. Most of these crypto processors are designed to process only a single algorithm with configurable parameters. For example such a structure is discussed in [11] [14] for AES algorithm. Some of the crypto processors can perform several algorithms in a single design. Most of the time there is one disjoint block for each included crypto algorithm. These kinds of structures are not area efficient and they are used mainly for high throughput applications.

There are programmable crypto processors, which are able to process more than one crypto algorithm in a single architecture, such as the joint implementations of AES, DES and TDEA as presented in [7] [8]. The most important property of these kinds of processors is their programmable architectures. The processors can be programmed according to the applied crypto algorithm.

The objective of this work is the implementation of a programmable Crypto processor architecture using the SystemC tool. The Advanced Encryption Algorithm (AES), which is the standard announced for block ciphers, the previous Data Encryption Standard (DES), and Triple Data Encryption Algorithm (TDEA), also known as Triple DES (TDES) algorithms are chosen for the implementation of the architecture. The Crypto architecture is implemented in the SystemC [24] environment. SystemC is based on C++, with some additional class libraries to model the hardware based features like clock, signals, logic and delay elements. SystemC allows modeling from the system level to Register Transfer Level (RTL). This modeling structure provides higher productivity than other modeling environments due to its easier and faster implementation. In the SystemC approach the design is implemented in modular structures. With this property of SystemC, the design can be modified to add new hardware blocks without changing the general structure.

In this thesis, instead of implementing two different blocks for each algorithm, the architecture is implemented as a common unit, which can perform operations of the chosen crypto algorithms. The implemented architecture is fully programmable and all the algorithms' operations are performed according to the instructions. The architecture is similar to general microcontroller's structure, but there are some differences for crypto operations. The internal structure of the implemented architecture is based on 32-bit data length and all crypto operations are performed in 32-bit arithmetic.

The operations are controlled by a Control Unit module and performed in Arithmetic Logic Unit (ALU) module or Permutation module according to Control Unit signals. The main operations are performed in the ALU. ALU is responsible for performing the crypto specific instructions as well as general purpose instructions. There is an internal memory block implemented inside the ALU for Substitution Table (SBox) operations. The SBox values for both of the AES and DES algorithms are stored in this memory unit.

Besides ALU, which performs the logic and arithmetic operations, a Permutation module is added into the design. In general applications, the bit permutation operation is implemented as a memory based structure or as a hardware routing structure. But in this implemented architecture, all of the bit permutation operations are performed in a single permutation module block. The bit permutation operations are used in Data Encryption Standard (DES) algorithm; therefore, the main purpose of this permutation module is performing DES permutations. But it can also perform other bit permutations depending on the applications.

The characteristics of Crypto Algorithms are described in Chapter 2. The types of the crypto algorithms are described in the first section of this chapter. Then the transformations of AES and DES algorithms and their basic process structures are explained in the following two sections. In Chapter 3, different implementations of Crypto processors in literature are discussed. In the first part of this chapter, the structures dedicates to a single algorithm and in the second part, crypto processors, which are capable of performing several algorithms, are discussed. The implemented architecture and its module structures are given in Chapter 4, where each module in the architecture is explained in detail. Also, the implemented Instruction Set Architecture and the instruction descriptions are given in this chapter. Finally, a conclusion for this work and proposed future works are presented in Chapter 5.

CHAPTER 2

CHARACTERISTICS OF CRYPTO ALGORITHMS

2.1 Introduction

This chapter explains the general description of the cipher algorithms and detailed structure of Advanced Encryption Standard and Data Encryption Standard algorithms. In the first section, the types and properties of the cryptographic algorithms are described. The AES algorithm, the DES algorithm and their operations are discussed in the second and third sections, respectively.

2.2 Types of Cryptographic Algorithms

Cryptography becomes a more important parameter with today's increasing security issues on communication area. There are lots of activities over communication networks of different applications and the security of the data in these applications are provided by using different cryptographic algorithms. These algorithms can be divided into three groups, as symmetric-key algorithms, public-key algorithms and hash algorithms.

2.2.1 Symmetric-key Algorithms

The encryption and decryption processes in the symmetric-key algorithms are performed with one key. There is only one secret key between the two sides of communication. The plaintext is encrypted by using the secret key and transmitted.

Then this ciphered data is decrypted by using the same secret key, which is used in the encryption part [5] [20]. This communication structure can be seen in Figure 2.1.

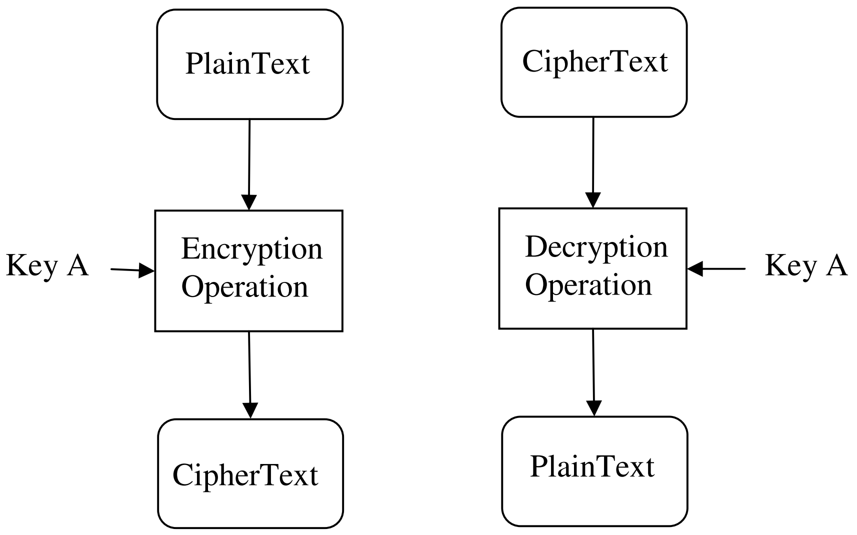


Figure 2.1: Symmetric Key Algorithms

The power of the symmetric algorithm is directly dependent to the key length. Because of the decryption process can be performed with trying all possible key combinations. Therefore the resistance of the symmetric algorithm against possible key trials is much higher with the increasing key length.

Symmetric algorithms can be divided into two groups as stream ciphers and block ciphers. The difference between these two groups is, the block ciphers use always the same sized data chunks in the encryption or decryption operations, but stream ciphers use different sized data in encryption or decryption operation.

2.2.1.1 Block Ciphers

Encryption and decryption operations are performed over blocks of data in the Block ciphers. Each block is used sequentially in the cipher operations. More clearly, a set of Boolean operations are performed on a definite length of bit vectors in a block cipher [1] [5].

There are normally two main techniques used in the Block ciphers. These are confusion and diffusion techniques. The aim of the confusion is making the output of the encryption as much as different from the input plaintext. Therefore the relation between input and output of the encryption will be more unpredictable. The substitution operation is mainly used in confusion technique.

On the other hand the diffusion technique is used to distribute the redundancy of the plaintext as much as possible into the cipher text. The main operation used for diffusion technique is permutation operation.

2.2.1.2 Stream Ciphers

Unlike block ciphers, stream ciphers operate on data context, with different bit lengths. Encryption or decryption is processed over these different sized data [5].

There are keys for each stream, which are generated by a key stream generator. The lengths of the key data is depends on the length of the data stream. Therefore the sequential key stream's length may show differences. In the encryption these keys and the plain data streams are XORed to get the ciphered data. Also in the decryption the same operation is performed. The same key stream data is XORed with the ciphered data, in this case to get the plaintext back.

In the stream cipher operation, the power of the operation is directly related to the key stream generator performance.

2.2.2 Asymmetric (Public-Key) Algorithms

Unlike symmetric algorithms, the asymmetric algorithms use different keys for encryption and decryption algorithms. There are two types of keys in the asymmetric algorithms. One of them is called private key and this key is known only by its owner. The other key type is called public key and this is known by all users in the communication [5].

In the asymmetric algorithms the relation between encryption side and decryption side is given in Figure 2.2. The encryption operation is processed by using the public key. Unlike encryption, decryption operation is processed only with the private key. The important point in the decryption is the private key’s owner issue. The private key should belong to the unit, which encrypted data with its public key, for a correct decryption.

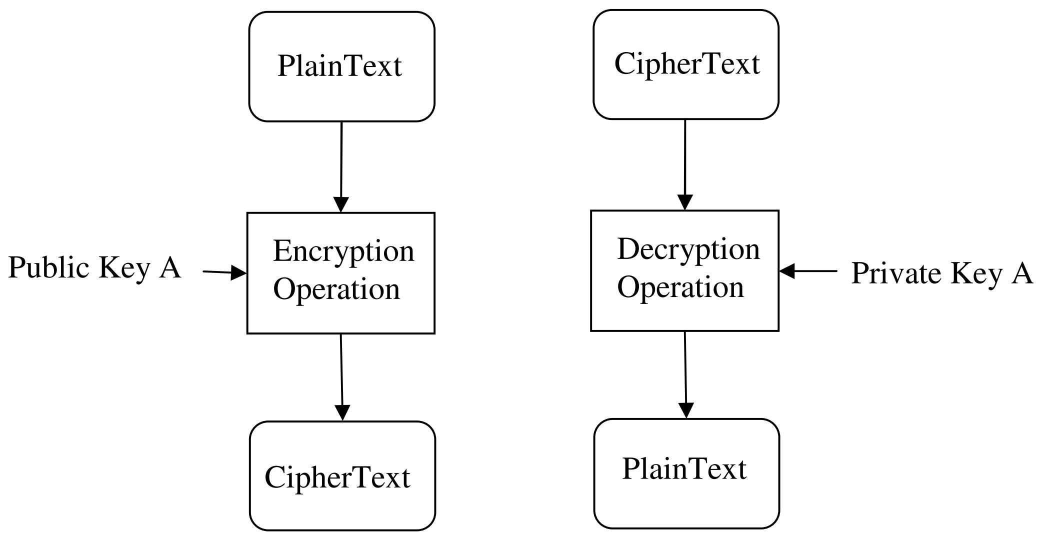


Figure 2.2: Asymmetric Key Algorithms

2.2.3 Hash Algorithms

Hash algorithms are a kind of pseudo random number generators in cryptography. There is no any formal description of Hash algorithms, but there are some general properties for it.

- For a given input message, there should be not any second input message, which gives the same hash output as the first input message. This property is known as collision resistance.
- For a given hash algorithm output, it should be hard to compute the input message. This property depends on the one-way function characteristic of the hash algorithms.

In the Hash algorithms, the input plaintext length is not fixed and can have a variety of lengths. But the output ciphered data of the Hash algorithm has a fixed data length. This property is achieved generally by processing the input data in equal-sized blocks and performed a one-way compression on the blocks. Therefore a very small change at the input side can create a very big change at the output side [1].

2.3 AES Algorithm

The Advanced Encryption Standard (AES) is a new Federal Information Processing Standard (FIPS) which was announced after an encryption algorithm standard competition by National Institute of Standards and Technology [5]. AES is also known as Rijndael [1] [2], but there are some small differences between AES and original Rijndael. The input data length is fixed to 128-bit in AES, while it can be 128, 192 or 256 bits in Rijndael.

The AES algorithm is a symmetric key algorithm and operates the encryption and decryption processes in blocks. The input data and key data of AES can be considered as one-dimensional array [1]. Each element of the array consists of 8-bit data. The one dimensional array of the incoming plaintext data (P) can be denoted by

$$P = p_0 p_1 p_2 p_3 \cdots p_{4*Nb-1},$$

where p_0 is the first byte and p_{4*Nb-1} is the last byte of plaintext. The incoming plaintext data is then mapped into a two dimensional matrix, which is called State [1]. All the AES operations are performed on the State matrix. The State matrix has a variable column number for different data and key lengths, with four rows. The column numbers are denoted by N_b for data state matrix and defined as;

$$N_b = \text{input data length} / 32.$$

The elements of the two dimensional State matrix can be defined as;

$$a_{i,j} = p_{i+4j},\ 0 \le i < 4,\ 0 \le j < N_b,$$

where $a_{i,j}$ denotes the byte in row i and column j.

Similarly, the input key is also mapped into a two dimensional matrix. The row number of key matrix is also four like in state matrix, and the column number is denoted by N_k, which is defined as below;

$$N_k = \text{input key length} / 32.$$

If we denote the one dimensional array of the key data (Z) by

$$Z = z_0 z_1 z_2 z_3 \cdots z_{4*Nk-1},$$

where z_0 is the first byte and z_{4*Nk-1} is the last byte of key, then the two dimensional matrix elements can be defined as below;

$$k_{i,j} = z_{i+4j},\ 0 \le i < 4,\ 0 \le j < N_k.$$

The input key bytes are mapped onto key state matrix in the order $k_{0,0}$, $k_{1,0}$, $k_{2,0}$, $k_{3,0}$, $k_{0,1}$, $k_{1,1}$, $k_{2,1}$,… [1]. The Data State matrix for 128-bit data is shown in Figure 2.3 and the Key State matrix for 192-bit key is shown in Figure2.4. The N_b value is 4 for AES, because the data input is fixed at 128-bit. N_k can have the values of 4, 6 and 8 for 128-bit, 192-bit and 256-bit, respectively.

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

Figure 2.3: Data State (for 128-bit data $N_b = 4$)

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

Figure 2.4: Key State (for 192-bit key data $N_k = 6$)

The rounds have sequential operations to perform encryption or decryption. The round numbers (N_r) are also depending on the N_b and N_k values. The Table 2.1 gives the round numbers for different data and key lengths for Rijndael.

Table 2.1: Round numbers (N_r) for different data and key lengths

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

In the AES algorithm, most of the operations are based on mathematical operations in Galois Field (2^8). Therefore, a brief explanation of the Galois Field (2^8) is discussed in the next part.

2.3.1 Galois Field (2^8)

The byte level operations in the AES algorithm are defined in the finite field (or Galois Field) GF (2^8) [1]. There are only a finite number of elements in a finite field and this number of elements is given as p^n, where p is a prime number and n is a positive integer.

The Galois Field (2^8) is an extension field of Galois Field (2) and it is represented by the coefficients of {0, 1}. A finite field can be represented as polynomials of degree smaller than the degree of the irreducible, reduction polynomial. A byte polynomial representation is given below;

$$b(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0 x^0 .$$

The arithmetic operations in the finite field are different from standard arithmetic and they will be explained in the following part. When the elements are represented as polynomials, then the arithmetic operations are performed modulo m. m is an irreducible polynomial over the Galois field with the same degree. For AES algorithm this irreducible polynomial is given by;

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

2.3.1.1 Addition and Subtraction

The addition and subtraction of the polynomials in a finite field is a simple EXOR operation and same for both of addition and subtraction.

2.3.1.2 Multiplication

In the finite field the multiplication operation can be expressed as multiplication of the polynomials with using an irreducible reducing polynomial for a modulus operation [1]. The irreducible polynomial for AES is given in m(x). The multiplication operation between a polynomial b(x) and “x” can be expressed in Figure 2.5.

First of all the polynomial is shifted to the left with a concatenated “0” on the leftmost bit. If the leftmost bit of the b(x) is ”1”, an EXOR operation is performed between the polynomial b(x) and the irreducible polynomial m(x), else EXOR operation is not performed. The result polynomial is the rightmost eight bits. The multiplication with “x” can be assumed as a fundamental operation in multiplication. Because of other polynomial multiplications can be considered as a sequence of multiplication with “x” [6].

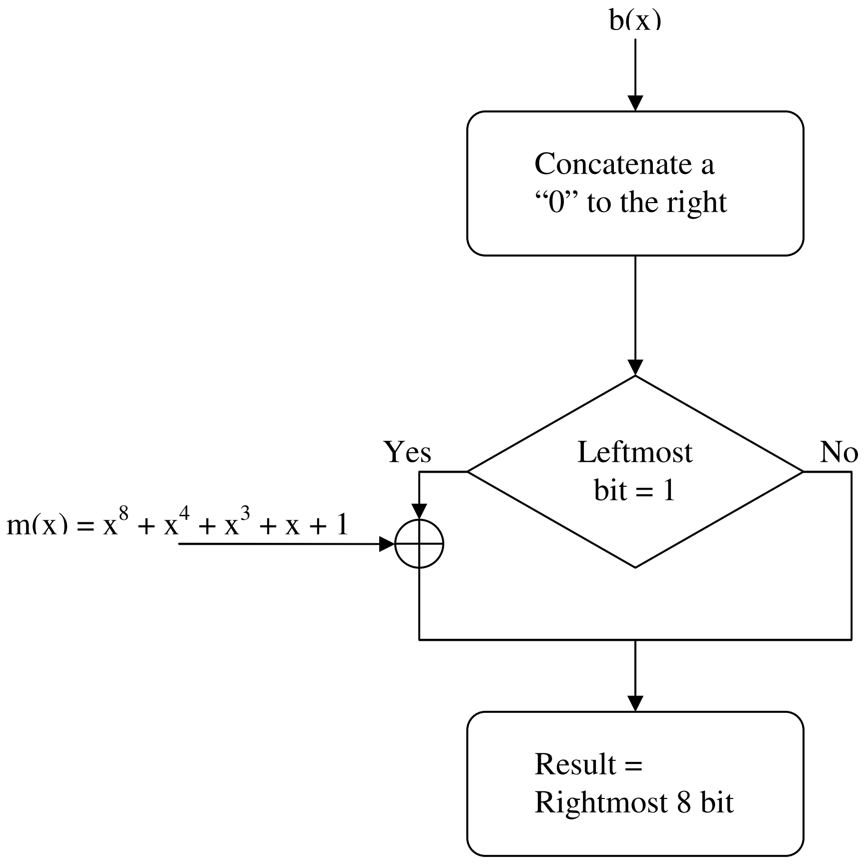

b(x)
Concatenate a “0” to the right
Yes
Leftmost bit = 1
No
m(x) = x^8 + x^4 + x^3 + x + 1
Result = Rightmost 8 bit

Figure 2.5: Multiplication of b(x) and x

2.3.2 Encryption Process of AES

The encryption process performed the inner state transformations over the plaintext data and as a result of these transformations the ciphertext data is given as output. The encryption diagram of the AES is given in Figure 2.6. There are four different transformations operation in the encryption process of AES algorithm. These are;

- SubBytes operation
- ShiftRows operation
- MixColumns operation
- AddRoundKey operation

The encryption process starts with an EXOR operation of plaintext and initial key data. Then the main iterated block, which consist of SubBytes, ShiftRows, MixColumns and AddRoundKey operations respectively. This main block repeats itself $N_r - 1$ times. In the final round only MixColumns operation is missing as a difference of main iterative block. The output of the final round is called as ciphertext data.

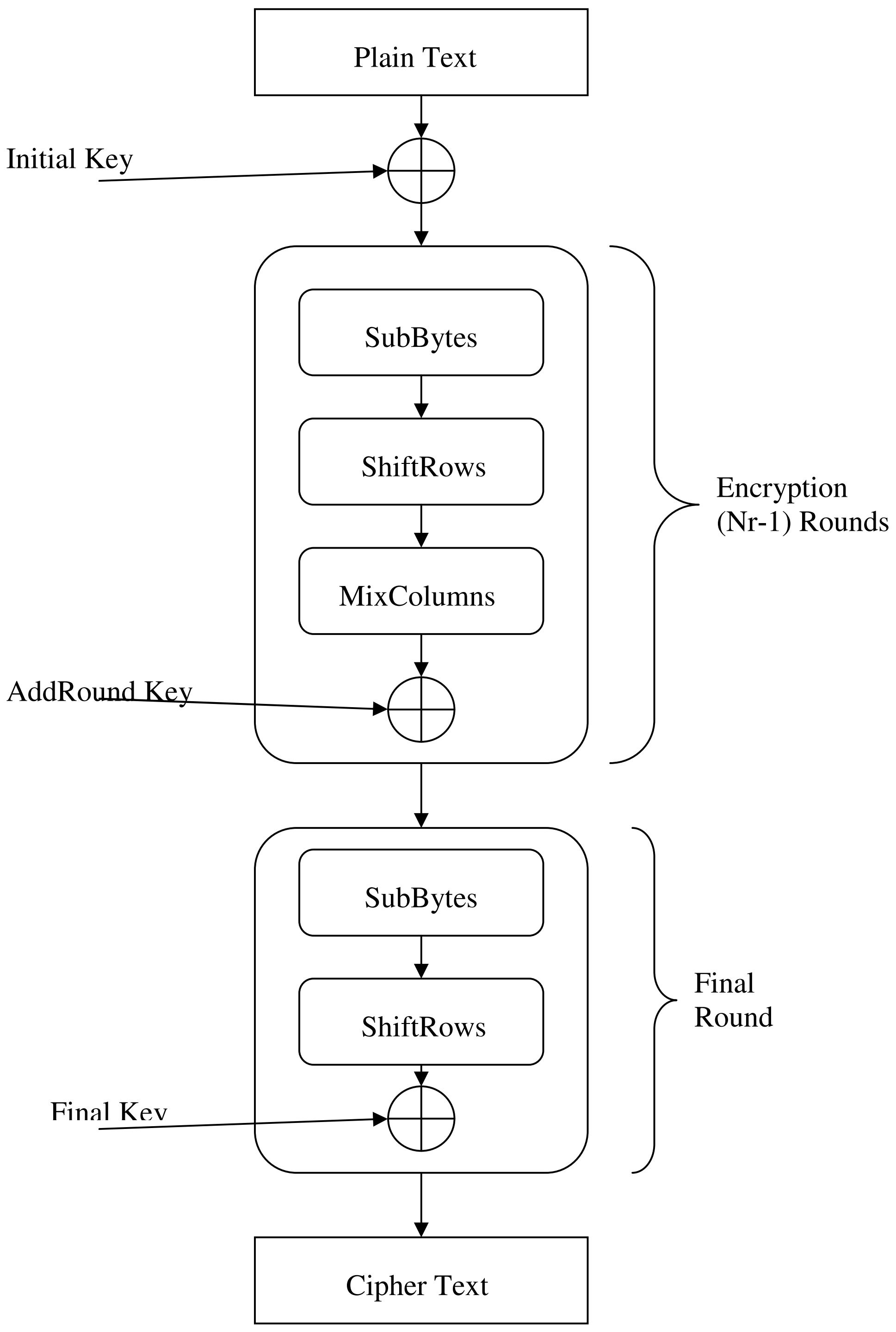


Figure 2.6: AES Encryption

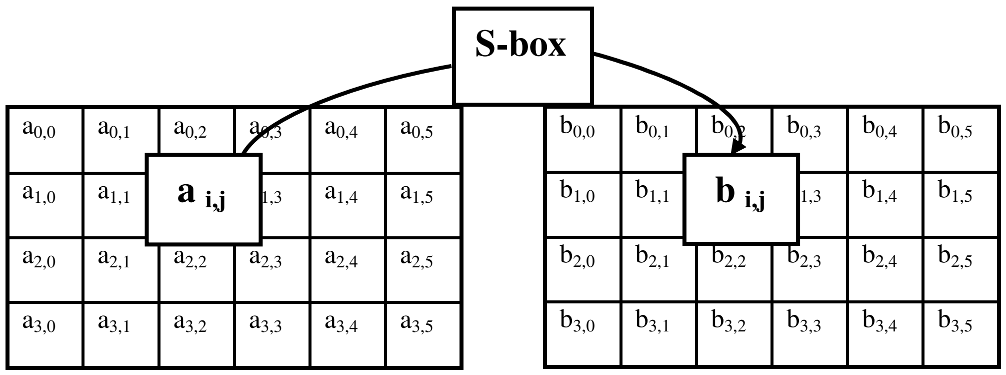
2.3.2.1 The SubBytes Transformation

In the SubBytes operation each State byte is replaced with the related substitution table element, which is determined according the State byte's value. The Substitution operation is the only nonlinear operation and the table is invertible.

In the construction of the SBox table, there are two operations. Firstly, the multiplicative inverse of the State byte is calculated in GF (2^8) .Then an affine transformation is applied, which is given in below Figure.

$$\begin{bmatrix} b_o \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_o \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 2.7: Affine Transformation



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

Figure 2.8: SubBytes Transformation on State

2.3.2.2 The ShiftRows Transformation

In this operation the rows of the State matrix are shifted to the right cyclically. For each data length and for each State matrix row, there is a different shift offset. The offset values are given for data length and row numbers in Table 2.2.

Table 2.2: The ShiftRows operation offset values for different data lengths

Row number /Data length	***128***	***192***	***256***
Row0	0	0	0
Row1	1	1	1
Row2	2	2	3
Row3	3	3	4

2.3.2.3 MixColumns Transformation

The MixColumns Transformation is a polynomial multiplication operation over GF (2^8). Each column of the State is considered as a unique polynomial and multiplied with a constant and invertible polynomial c(x), which is co prime to x^4+1.

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

The multiplication of the State column a(x) with the constant polynomial c(x) and the result State column b(x) can be written in a matrix form as given in Figure 2.9.

$$b(x) = c(x) * a(x) \qquad (\text{mod } x^4+1)$$

$$\begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \end{bmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix}$$

Figure 2.9: The multiplication of State Column and c(x)

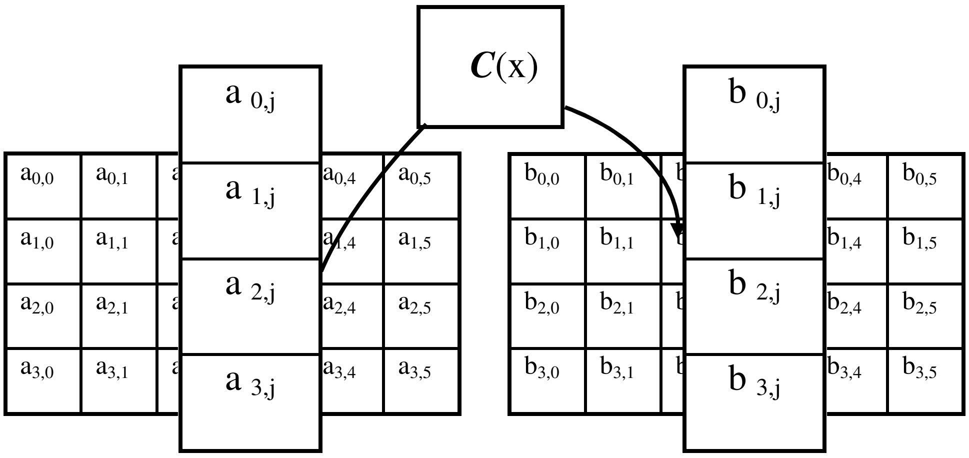

$C(x)$
$a_{0,j}$
$a_{1,j}$
$a_{2,j}$
$a_{3,j}$
$b_{0,j}$
$b_{1,j}$
$b_{2,j}$
$b_{3,j}$
$a_{0,0}$ $a_{0,1}$ $a_{0,4}$ $a_{0,5}$
$a_{1,0}$ $a_{1,1}$ $a_{1,4}$ $a_{1,5}$
$a_{2,0}$ $a_{2,1}$ $a_{2,4}$ $a_{2,5}$
$a_{3,0}$ $a_{3,1}$ $a_{3,4}$ $a_{3,5}$
$b_{0,0}$ $b_{0,1}$ $b_{0,4}$ $b_{0,5}$
$b_{1,0}$ $b_{1,1}$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

Figure 2.10: MixColumns operation

2.3.2.4 AddRoundKey Transformation

In AddRoundKey Addition operation the round data and AddRoundKey data is subjected to an EXOR operation.

$$\begin{array}{|c|c|c|c|}\hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \oplus \begin{array}{|c|c|c|c|}\hline k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ \hline k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ \hline k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ \hline k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \\ \hline \end{array} = \begin{array}{|c|c|c|c|}\hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array}$$

Figure 2.11: AddRoundKey Addition

2.3.3 Decryption Process of AES

The decryption process is the inverse operation of the encryption process. The transformations in the encryption round are reversed in the mean of the sequence. The decryption diagram of the AES is given in Figure 2.12.

The transformations used in encryption operation are also inversed in the decryption process. The InvSubBytes transformation is the inverse operation of the SubBytes. The InvSubBytes transformation uses the inverse table of the normal SBox table.

The inverse SBox table is obtained by applying the inverse of the affine transformation followed by taking the multiplicative inverse in GF(2^8) [1]. For example the SBox value of the input 0x81 is 0x0c. And in the inverse SBox table the output of the 0x0c is 0x81.

The inverse SBox table is given in Table A.10 in Appendix A. In the Inverse ShiftRows transformation the shift operation is performed to the right instead of the left side in the encryption process. The offset values in the both shift transformations are same.

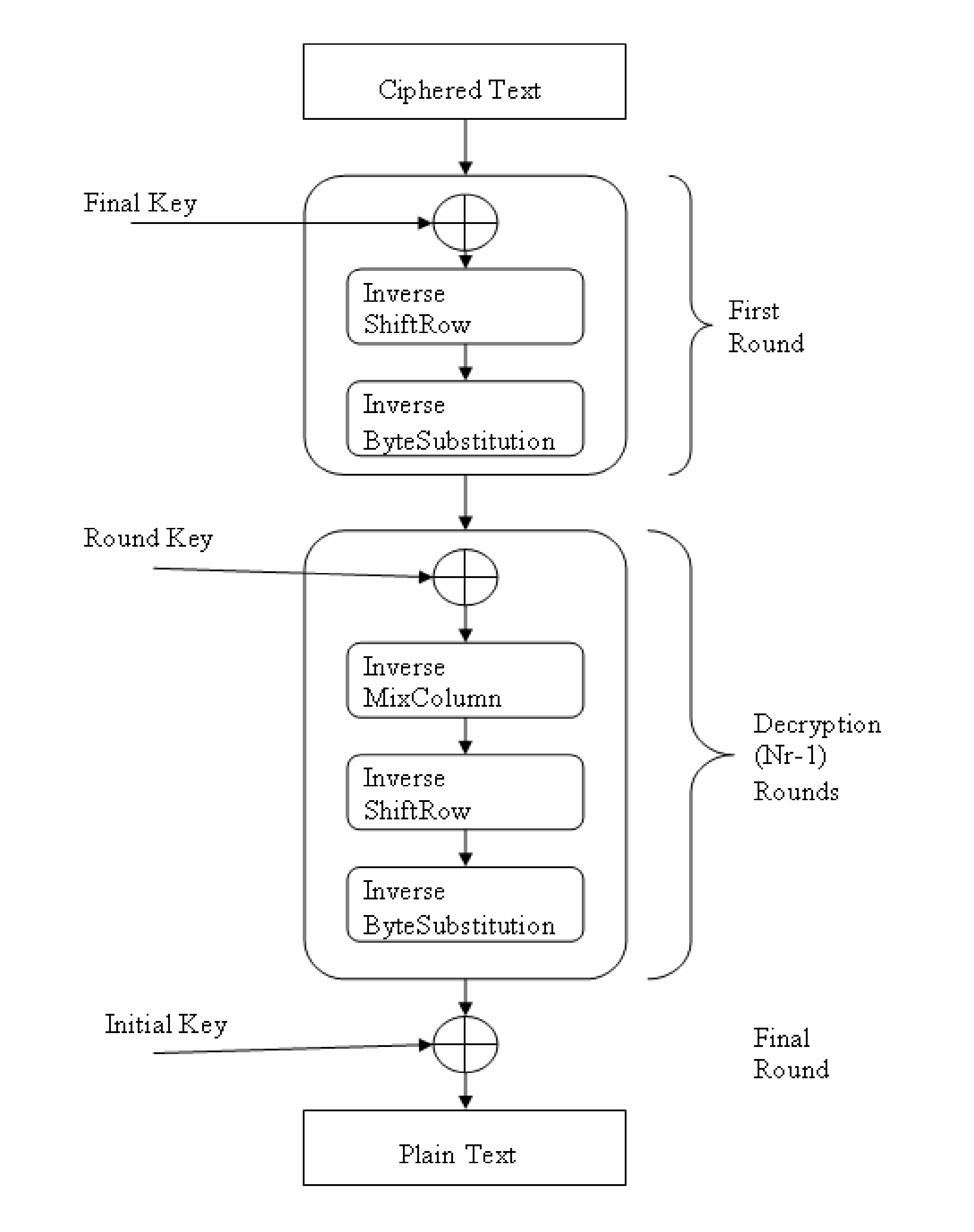


Figure 2.12: AES Decryption

The Inverse MixColumns transformation is similar to MixColumns in encryption. But the coefficients of the constant polynomial are changed. The constant polynomial for the Inverse MixColumns transformation is named as d(x), where

$$d(x) = \text{'0B'}\ x^3 + \text{'0D'}\ x^2 + \text{'09'}\ x + \text{'0E'},$$

and $c(x) \cdot d(x) \equiv 01 \pmod{x^4+1}$.

$$\begin{pmatrix} b0 \\ b1 \\ b2 \\ b3 \end{pmatrix} = \begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} \bullet \begin{pmatrix} a0 \\ a1 \\ a2 \\ a3 \end{pmatrix}$$

Figure 2.13: The multiplication of State Column and d(x)

The round number is same for decryption process. But the AddRoundKey is applied in reverse order. The first operation of the decryption is EXOR operation between the final round key and the ciphered data, which is the input of the decryption process.

Then inverse ShiftRows and Inverse SubBytes are performed sequentially. The iterated rounds start with the AddRoundKey transformation and then continue with Inverse MixColumns, Inverse ShiftRows and Inverse SubBytes transformations. As a last operation the EXOR operation with the first round key is performed to get the plaintext.

2.3.4 Key Expansion and Round Key Selection

The Key Expansion part is responsible to provide the round keys for relevant rounds of cipher operation. While the round number can be different for different key lengths, the operation of the Key Expansion can show differences. The operation is same for 128-bit and 192-bit key length but it is different for 256-bit key length.

The operations in the Key Expansion is made over 32 bits, named as word “W”. The input key is assigned as the first N_k words of the Key Expansion. All of the other words are obtained recursively of these words. The expansion operation of the remaining words is given in Figure 2.14 for 128 and 192 bits and in Figure 2.15 for 256 bits.

The recursive operation for obtaining the following words after first N_k word uses the previous words, the N_k positions earlier words and round constants. The recursive function is directly related to the position of the word. If the current position “i” is not a multiple of the N_k, then a simple XOR operation between previous word (W[i-1]) and N_k earlier word (W[i – N_k]) gives the current word value (W[i]). In the other situation, if “i” is a multiple of the N_k, the current word W[i] is the result of the EXOR operation of N_k earlier word and the nonlinear function of the previous word W[i – 1]. This nonlinear function consists of a cyclically rotation operation to right by one byte, which is called RotByte, a nonlinear byte substitution operation for each byte in the word element, which is called SubByte, and addition of a round constant value. The round constants are independent of the N_k value, and defined by a recursion rule in GF (2^8) as shown below.

Rcon [1] = x^0 (i.e. 01)

Rcon [2] = x^1 (i.e. 02)

Rcon [k] = x * Rcon [k-1] = x^{k-1} , k > 2.

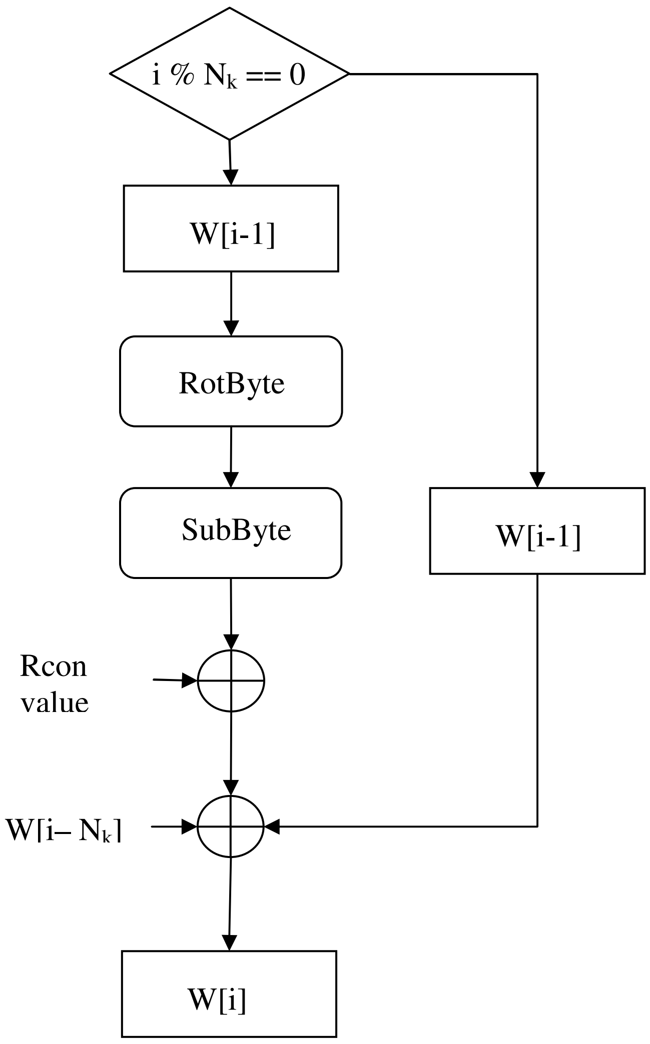


Figure 2.14: Key Expansion for 128 and 192 bits

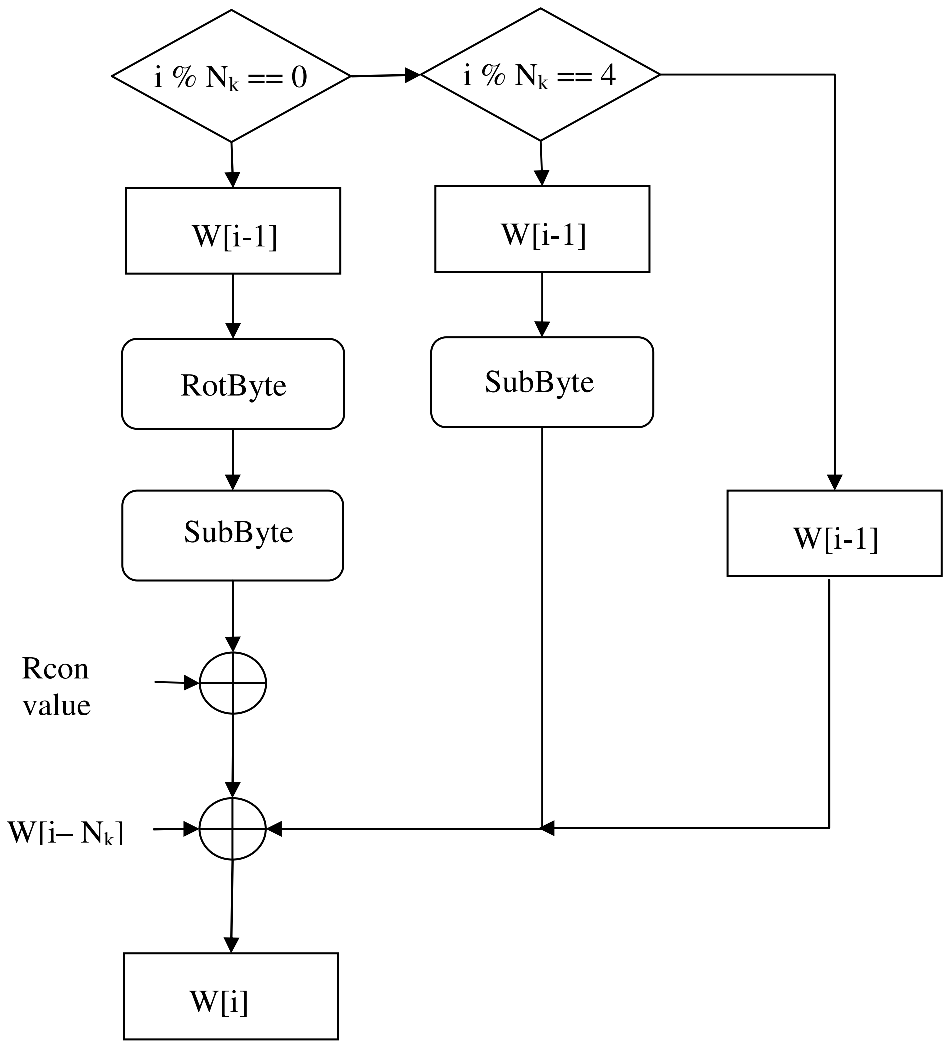


Figure 2.15: Key Expansion for 256 bits

The round keys are chosen from the word array of Key Expansion part. The round keys' length should be equal to the input plaintext length. Hence the round key consists of array elements from word W [N_b*i] to word W [N_b * (i+1)].

The round key selection is illustrated in Figure 2.16.

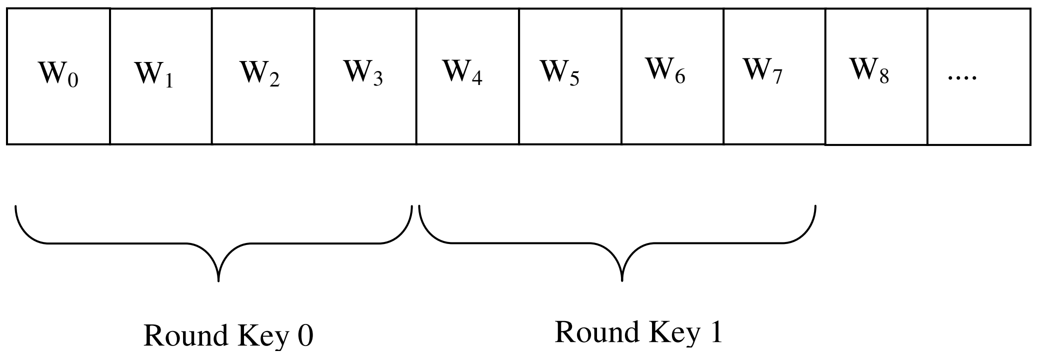


Figure 2.16: Round Key Selection

2.4 DES Algorithm

The Data Encryption Standard (DES) was developed by IBM in 1970s and then approved as a standardized crypto algorithm by Federal Information Processing Standard (FIPS) [4] in 1977. DES is a symmetric crypto algorithm, which operates on 64-bit block size within 16 rounds. The input plaintext and the output ciphered text are 64-bit. The encryption or decryption operation is achieved by a 64-bit key data. But only the 56bits of the whole key data is effective. The remaining 8 bits have no effect on the encryption/decryption process of the DES. The encryption and decryption processes use the same key due to symmetric nature of the algorithm. Also the ciphering flow is same for both the encryption and decryption.

The only difference is the order of the round keys. The round keys are in reverse order for the decryption process. The block diagram of the DES encryption algorithm is given in Figure 2.17. The DES algorithm can be analyzed in two parts. The first part is the Key Expansion part, which generates the necessary round keys. And the second part is the encryption part. In the second part the encryption or decryption process is operated with the contribution of the round keys. Also the encryption part can be divided into two group of operation. First one is the permutation operations, which are the first and last operations of the cipher part, and the second group consists of rounds operation between these permutations.

2.4.1 DES Rounds

2.4.1.1 Initial Permutation

The Initial Permutation is the first operation in the DES encryption algorithm. The incoming 64-bit plaintext data is subjected to initial permutation table, which is given in Table2.3. According to the table the first bit of the output data is the 58. bit of the input data, the second bit of the output is the 50. bit of the input data and so on.

Table 2.3: Initial Permutation Table

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

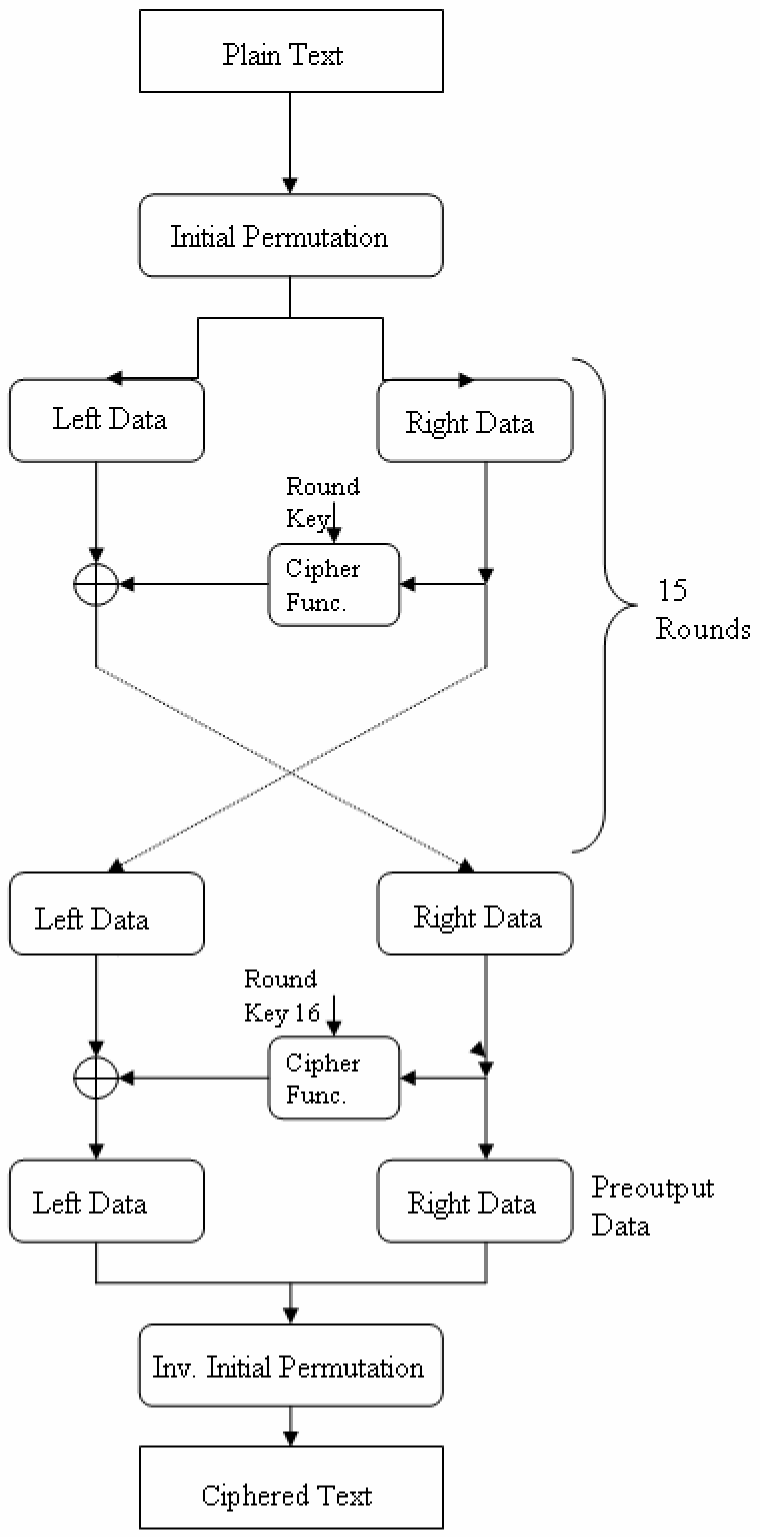


Figure 2.17: DES Algorithm

The main round processes start after the Initial permutation. The data is split into two groups of 32 bits as shown in Figure 2.17. These groups of data are named as “R” right half and “L” left half. The Right half is joined to the encryption or decryption process with the round key data. The key-dependent operation, substitution tables operations are processed in a function, called cipher function.

2.4.1.2 Cipher Function

The operations in the Cipher function are given in Figure 2.18. There are two permutation operations, which are E Table permutation and P permutation, a Substitution operation and an EXOR operation with the round key data.

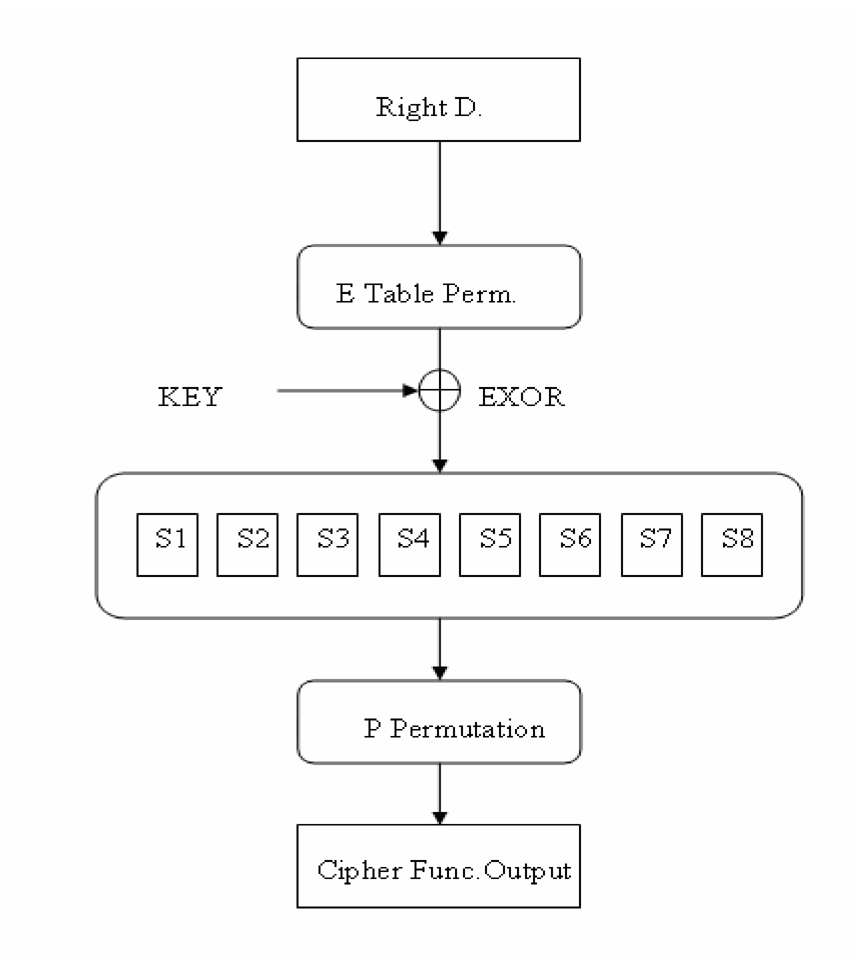


Figure 2.18: DES Cipher Function

Table 2.4: E Bit Selection Table

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

The E Bit Selection table is the first operation in the Cipher function. The round keys in the DES algorithm are 48 bits, while the round data from group R is 32 bits. The E Bit Selection table matches the number of bits of the round data to the round key data, as duplicating some of the bits, which is given in Table 2.4.

After this operation an EXOR operation performed between the key data and round data. The output of the EXOR operation is fed into a SBox array. Each one of the eight SBox units takes 6-bit data as input and gives 4-bit data as output.
The SBox Table S1 is given below. The whole SBox tables from S1 to S8 appear in Appendix A.

Table 2.5: SBox S1 Table

Row Num	Column Number															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Each of the SBox units has 64 memory elements. The input 6-bit data is replaced with one of the SBox memory elements according to its value. The first and last bits of the incoming data to the SBox unit determines the row number and the middle 4 bits represent the column number of the output data in the SBox unit. The output data of the SBox units is 32-bit data again. Hence the expanded round data is reduced again to its normal bit length with this operation. The output data of the SBox undergoes to another permutation, P permutation, which is defined in Table 2.6.

Table 2.6: P Permutation Table

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

The definition of the table is same with other permutation tables. The first bit of the output is the 16. bit, the second bit is the seventh bit of the input and so on.

The P permutation is the last operation of the Cipher function. Then the output of the Cipher function and the Left part of the round data is XORed. The result of the XOR operation will be the Right part of the next round data. And the Right data of the current round becomes the Left part of the next round data. This swap operation between Left part and Right part is not performed in the 16.th round.

2.4.1.3 Inverse Initial Permutation

The Left and Right data of the 16.th round are concatenated and named as preoutput block. This preoutput block data is subjected to the Inverse Initial permutation. This permutation is the last operation of the DES encryption/decryption process and it is the inverse operation of the Initial permutation.

Table 2.7: Inverse Initial Permutation Table

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

2.4.2 Key Expansion Part

The Key Expansion part takes the 64-bit key data as input and prepares the 16 round key data for encryption process. However the input data is 64-bit length, only 56-bit data is used for the round keys preparation. The eight bits of the each byte is dropped. In some cases the 8th bits can be used as parity bit for error detection in key generation. There are three parts of the Key Expansion part. In the first part the input data is subjected to the PC-1 permutation. The permutation table of the PC-1 is given in Table 2.8. Then the output data are split into two parts like in encryption process, but here the divided parts are 28-bit long. The second part of the Key Expansion is cyclic left shift operation applied each of these two 28-bit parts individually. The two parts are shifted to left with predefined offset values before calculating the round key. The offset values for each round are given in Table 2.10. The last process in the Key Expansion is PC-2 permutation, which permutation table is given in Table 2.9. The input of the PC-2 permutation is 56-bit data and the output is 48-bit data. There is compressing process applied into the key data with the PC-2 permutation. After each left shift operation the data is subjected to the PC-2 permutation and the result of this operation is the round key data.

Table 2.8: PC-1 Permutation Table

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Table 2.9: PC-2 Permutation Table

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Table 2.10: Left Shift Offset Value Table

Round Number	Left Shift Offset
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

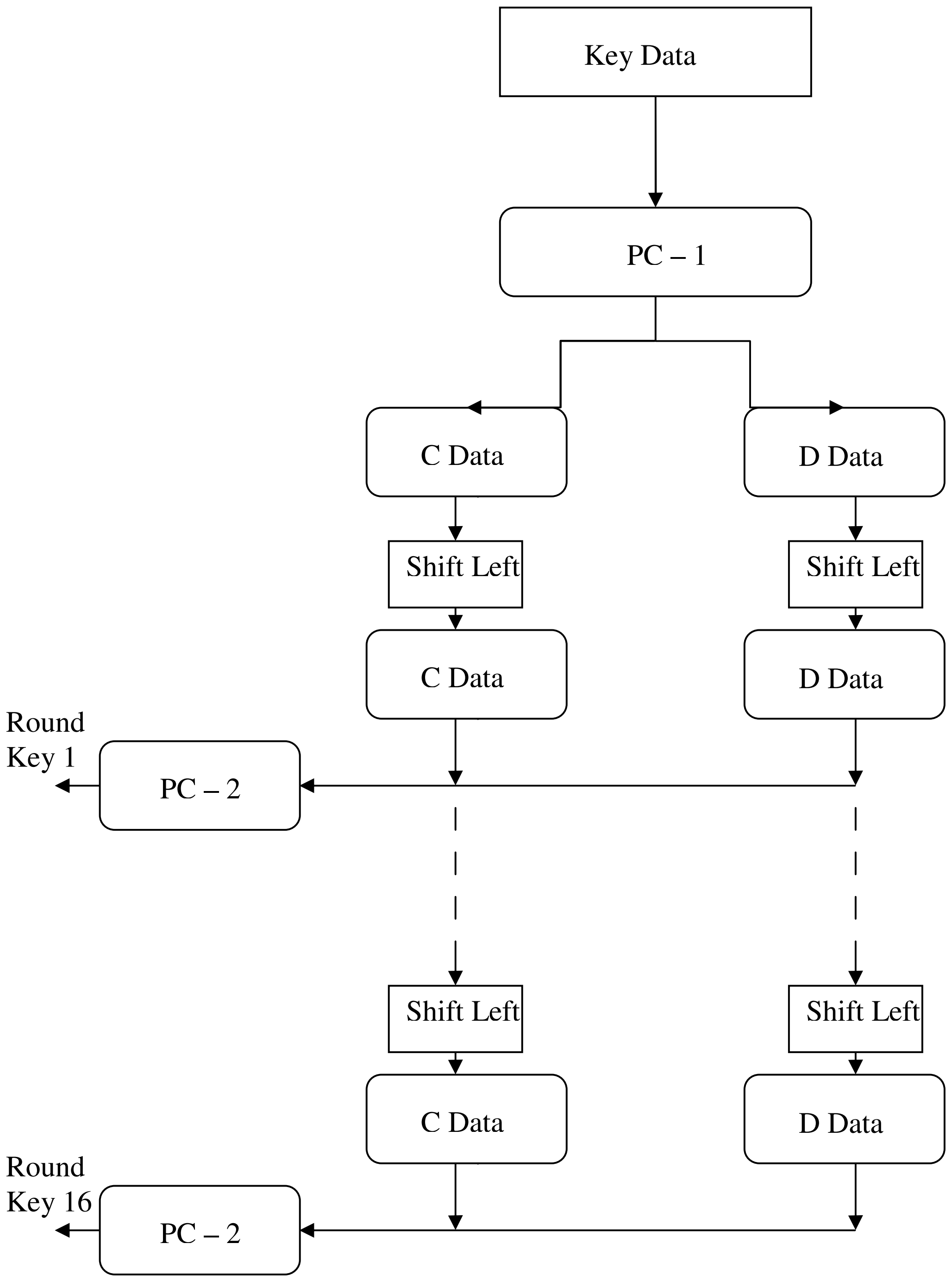


Figure 2.19: DES Key Expansion

2.5 Triple DES (TDES) Algorithm

The Triple Data Encryption Algorithm (TDEA), more commonly Triple DES is an approved cryptographic algorithm, which is enlarge the key space of the DES algorithm. There are three DES keys in the TDES operation, which are called as Key_1, Key_2, Key_3 and referred to as a key bundle (KEY). These three keys are used in two allowed option to form the key bundle. In the first option, all three keys are mutually independent (i.e. Key_1, Key_2 and Key_3, where $Key_1 \neq Key_2 \neq Key_3 \neq Key_1$). And in the second option, there are mutually independent keys and a third key that is the same as the first key (i.e. Key_1, Key_2 and Key_3, where $Key_1 \neq Key_2$ and $Key_3 = Key_1$). The simple encryption and decryption operations of the TDES are given in Figure 2.20 and Figure 2.21 respectively. In the TDES encryption operation, the algorithm process begins with the DES encryption by using Key_1 , then continue with DES decryption operation by using Key_2 and it is finished with DES encryption operation by using Key_3.

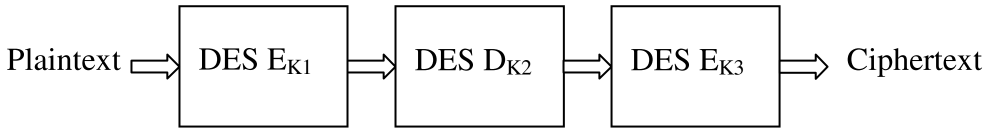


Figure 2.20: TDES Encryption Operation

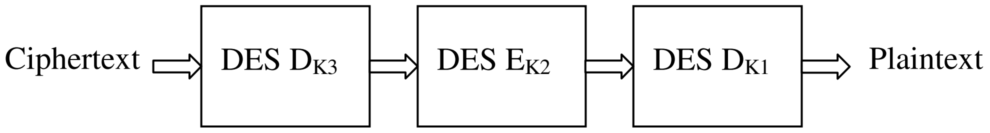


Figure 2.21: TDES Decryption Operation

In this chapter the AES, DES and TDES algorithms have been discussed. The crypto algorithm types, the encryption and decryption structures of the AES, DES and TDES algorithm and the transformations used in algorithms are explained. The next chapter presents the different implementations of these algorithms.

CHAPTER 3

CRYPTO PROCESSOR ARCHITECTURES

3.1 Introduction

In Section 3.2, different architectures and implementations of these architectures are discussed. In Section 3.3 bit permutation instructions in the literature, are described.

3.2 Different Processor Implementations

There are many studies about the crypto processors in the literature. Some of these studies are focused on only a single crypto algorithm, and some others are designed to support several algorithms in a single architecture. The different architectures are discussed in this chapter.

Crypto processor architecture, called Cryptonite, is presented by Rainer Buchty, Nevin Heintze and Dino Oliva in [7]. This study is about a programmable architecture for the cryptographic applications. DES, TDES, AES, IDEA, RC6, MD5, and SHA-1 algorithms are supported by this architecture.

This architecture has a different instruction set for cryptographic processing such as parallel 8-way permutation lookups, parameterized 64-bit/32-bit rotation, and XOR-based fold operations.

These instructions are used for the core functions of different crypto algorithms and show differences than general purpose instructions. All instructions are executed in a single cycle. 64-bit and 32-bit computations are supported in this study. The main architecture of the Cryptonite is given in Figure 3.1.

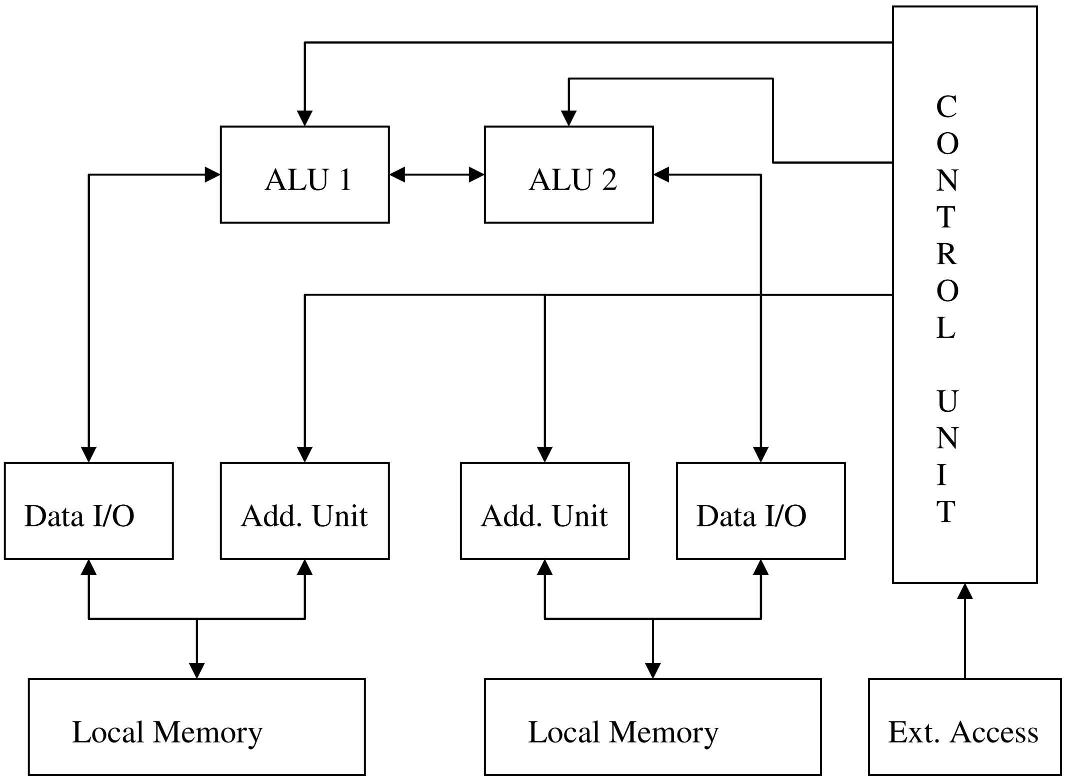


Figure 3.1: Cryptonite architecture

The Control unit controls the system according to the instructions. There is a two-cluster architecture presented in the study. There is an ALU and its accompanying data I/O unit for each of cluster. The data unit of the ALU is responsible of the data access between local data memory and ALU. There is an interlink between the ALUs to enable the data change in complex computations.

Furthermore the new XOR unit implementation into the data path, a parameterizable permutation engine, a DES specific unit and some AES supporting functions are implemented in the architecture. The DES specific unit is implemented into the memory unit instead of the ALU.

The XOR unit of this architecture has 6 input. These inputs come from ALU registers, memory unit and as immediate value. The aim of this 6-input XOR unit is to avoid the sequential operations between multi input XOR operations.

The other new unit is the parameterizable permutation engine. The permutation operations are performed with a lookup table, which can be used up to 8 parallel lookups. The vector memory unit, which is used as reconfigurable permutation engine, receives a vector of indexes and a scalar base address to address the memories of a vector. The collections of addressed memories form the result data vector. The structure of the vectored memory access is given in Figure 3.2.

The results of the Cryptonite architecture are given in Table 3.1.

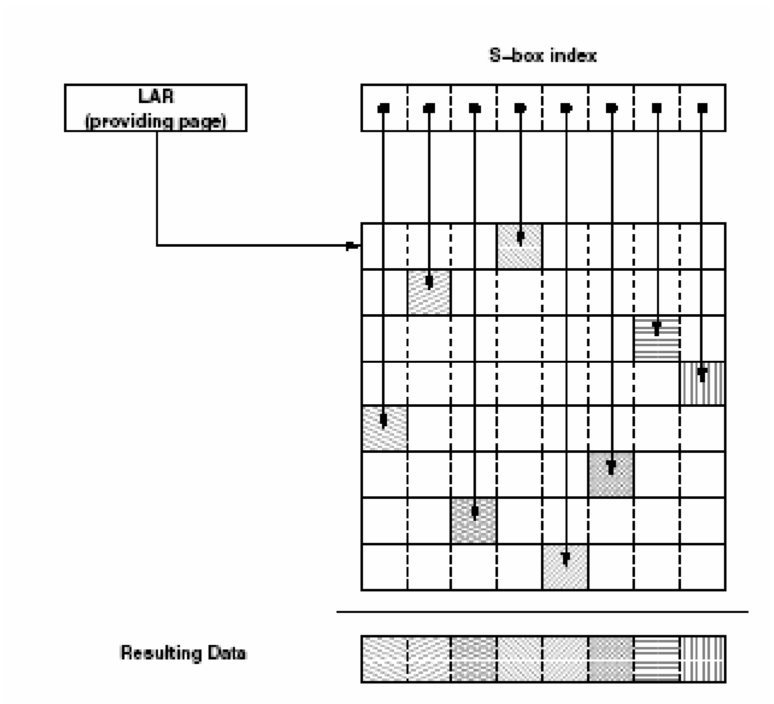


Figure 3.2: Vectored Memory Access

Table 3.1: Cryptonite architecture results

Algorithm	***Throughput (Mbit/s)***	***Cycle count***	***Speed (MHz)***
DES	732	35	400
TDES	244	105	400
AES	731	70	400
MD5	421	504	400

Another architecture for the programmable processor is presented by Lisa Wu, Chris Weaver and Todd Austin [8]. The presented architecture, called CryptoManiac, is a 4-wide, 4-stage 32-bit VLIW processor with a three input operand ISA. There is a simple branch predicter in the processor, but it does not have a cache. The code and data is stored in a static RAM. The branch predicter is used to make predictions about the next target address when there are more than one branch instructions in an instruction word.

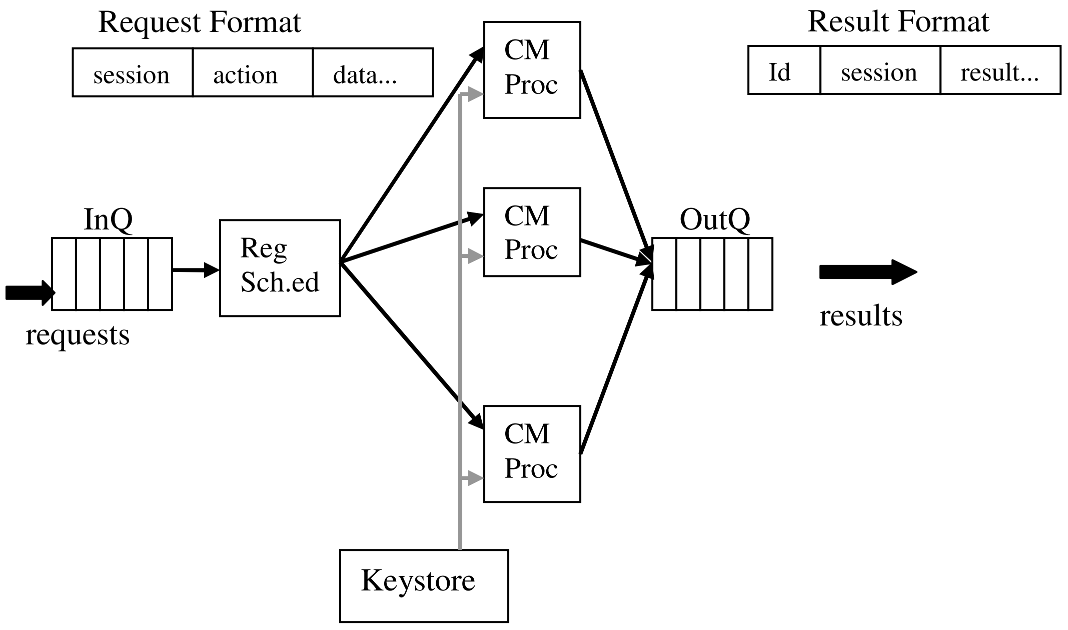


Figure 3.3: Schematic of CryptoManiac Architecture

The interface between a host processor and CryptoManiac is provided by input and output request queues. A request scheduler distributes the requests of host processor to CryptoManiac processor in the order of receive. The Keystore part is a high-density storage element for storing key data and substitution tables. Simultaneous session processing on the same processor is available by storing key-specific data in the shared keystore.

This data includes substitution data, permutation counters, and other internal algorithm state data. This part is only used for multisession applications and not necessary for single session applications. There are four parallel functional units in the CryptoManiac architecture. The process in the architecture is started with fetching a single VLIW instruction word that contains four independent instructions.

The instruction set consists of 32-bit instructions and enhanced for the cryptographic processes by combining general arithmetic instructions with logical instructions, substitutions with logical instructions, and rotate operations with logical instructions. Each instruction has three operands as input and one operand for the output, again to combine some instructions.

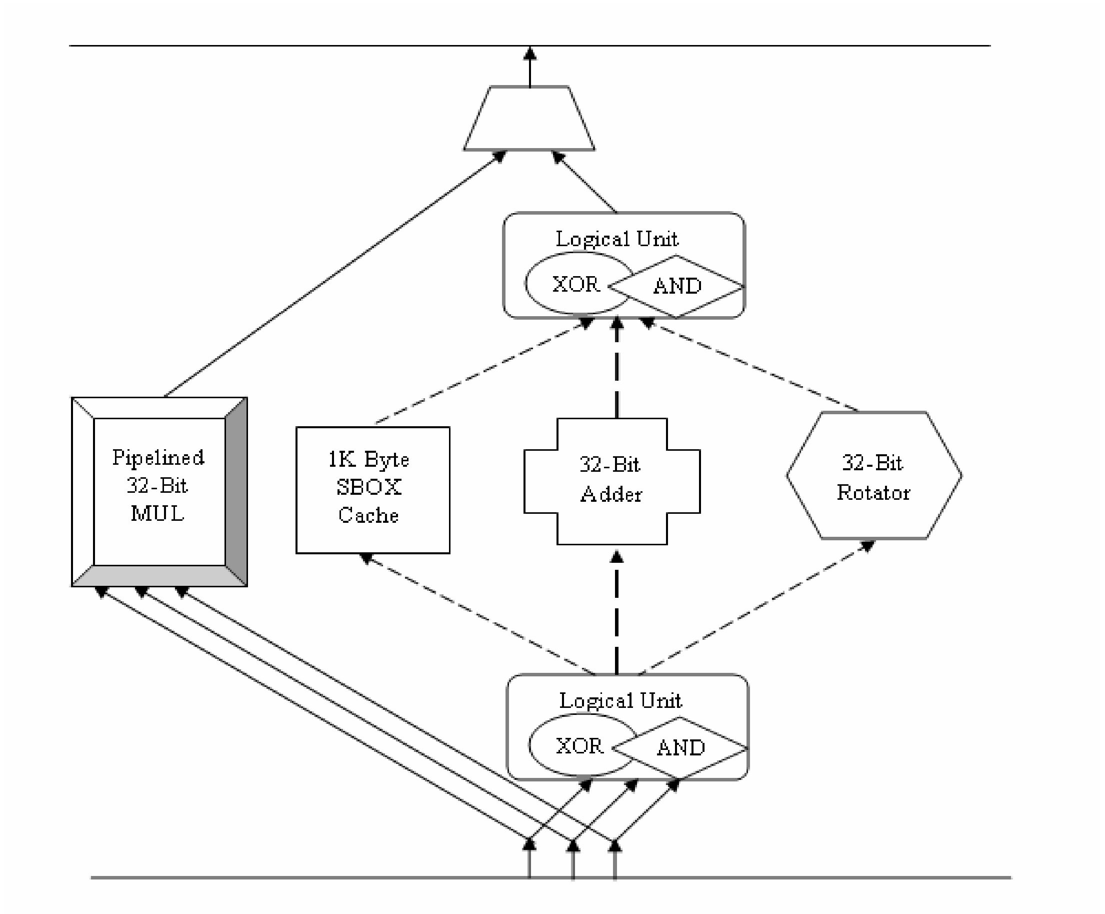


Figure 3.4: Schematic of a single functional unit

The Figure 3.4 shows the internal structure of a functional unit. Each functional unit consists of two logical units, one adder, one 1k-byte SBox cache, and one rotator. The multiplier block is added to only two blocks. The XOR, AND operations are processed in the logical units. SBox cache is responsible for holding all the data, key and SBox parameters instead of using a memory. The estimated result of the CryptoManiac is given in Table 3.2.

Table 3.2: Estimated results of CryptoManiac architecture

Algorithm	***Throughput (Mbit/s)***	***Cycle count***	***Speed (MHz)***
TDES	68	336	360
TDES corr.	59	392	360
AES 128/128	511	90	360
AES 128/128 corr.	353	130	360

In another study by Ricardo Chaves, Georgi Kuzmanov, Stamatis Vassiliadis and Leonel Sousa [9] the AES encryption/decryption algorithm with a memory based hardware design is proposed. In the memory based design both the SubBytesand the polynomial multiplication are implemented in internal memories of the FPGA (BRAM).

There are two AES encryption/ decryption cores presented. One of them is a completely unrolled loop structure capable of achieving a throughput above 34 Gbits/s, with an implementation cost of 3513 slices and 80 BRAMs; and the other one is a fully folded structure, requiring only 515 slices and 12 BRAMs, capable of a throughput above 2 Gbits/s.

The first structure has not any area constraints and it is designed for the higher throughput requirements. The second structure is designed for the area constraints. In the general AES block implementation they use dual port memory blocks such as BRAM's in the FPGA's for the SBox and MixColumns processes. In the BRAM's 2 SubBytesstitutions and 2 full multiplications can be mapped in a single memory block.

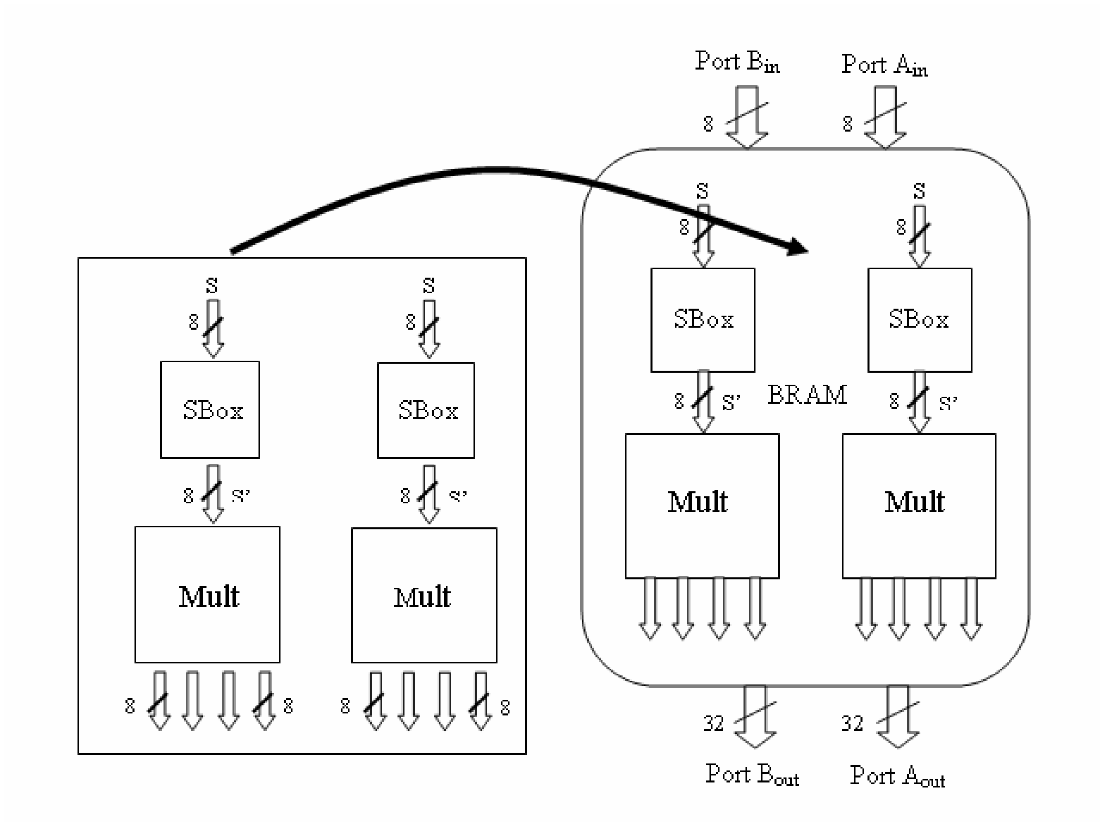


Figure 3.5: SBox and MixColumns computation using BRAM

After implementing the BRAM structure, the paper proposes two architectures for the AES operation. AES unfolded core is designed with adding sequentially all the rounds and AES folded core is designed with only one core which repeats the rounds.

In a study by Alireza Hodjat, Ingrid Verbauwhede [10] an area-throughput trade-off for an ASIC implementation of the Advanced Encryption Standard is presented. The paper presents throughputs of 30 Gbits/s to 70 Gbits/s with loop unrolling and inner-round and outer-round pipelining techniques, using a 0,18 μm CMOS technology. Also, the possibility of achieving a throughput of over 30 Gbits/s encryption using the AES algorithm with minimum area cost is explored in this paper. The main goal of the paper is combining the pipelining with a composite field implementation. The paper calculates the SBox values using the Galois Field operations. The input byte (element of GF (2^8)) is mapped to two elements of GF (2^4). Then, the multiplicative inverse is calculated using GF (2^4) operators. Then, the two GF (2^4) elements are inverse mapped to one element in GF (2^8). In the end, the affine transformation is performed. There is also used pipelined structure in the SBox calculation structure to avoid the high latency in the Galois field operations.By using a pipelined structure in the SBox process, the area is reduced up to 35 percent and by designing an offline key scheduling unit for the high speed AES processor, an area reduction of an extra 28 percent is achieved according to the paper.

In another study by Oscar Perez, Yves Berviller, Camel Tanougast and Serge Weber [11] the experimental results of different strategies of implementation of AES encryption algorithm is presented. There is given a comparison between different techniques at the beginning of the study.

These techniques are Inner-Round pipelining, Outer-Round pipelining, Full Loop Unrolling, Iterative looping and reconfiguration. They divided the algorithm into two parts, which are Key Expansion part and Cipher Part. The above strategies are used in the implementation of these two parts and then a comparison is made between the cost and the performance of the implemented techniques. The paper offers three strategies to compare the performances.

1. Unrolling the loop and reconfiguration techniques
2. Iterative looping and the reconfiguration techniques
3. Pipelined technique

The performance tests are implemented on the FPGA Xilinx XC2V6000. According to the results the best throughput is achieved by config1. But the weak side of this technique is reconfiguration time. By contrast, in config2 who uses reconfiguration and the reusing of operators, the throughput is very low, but it offers two advantages: the use of few resources and a density of calculation quite near the other implementations. The performance, surface is interesting because these values are close to the best implementation. On the other hand, they are penalized in terms of latency by the time used for the reconfiguration. The pipelined technique also has good throughput results, but it uses higher BRAM capacity.

A reconfigurable processor implementation is proposed by Yongzhi Fu, Lin Hao and Xuejie Zhang [12]. This study is about the implementation of a counter mode AES based on the Xilinx Virtex2 FPGA platform. In the AES design there is loop unrolling, inner and outer round and mixed pipelining. The clock frequency of the fully mixed inner and outer round pipelined architecture has achieved 212.5MHz and that translate to throughput of 27.1Gb/s. The difference of this article is using a switch between MixColumns operation and AddRoundKey operation.

For the SBox operation Look Up Tables are used and the ShiftRows is implemented by configuring the routing resources. In the MixColumns operation they use shift and accumulation method, which is shift the incoming data 0 bit left when the polynomial constant is '02' and then XOR the results. The AddRoundKeys are computed before the encryption process for a pipelined structure. According to the several implementation tests the best result is achieved using the mixed structure, which includes inner and outer pipelining, and loop unrolling.

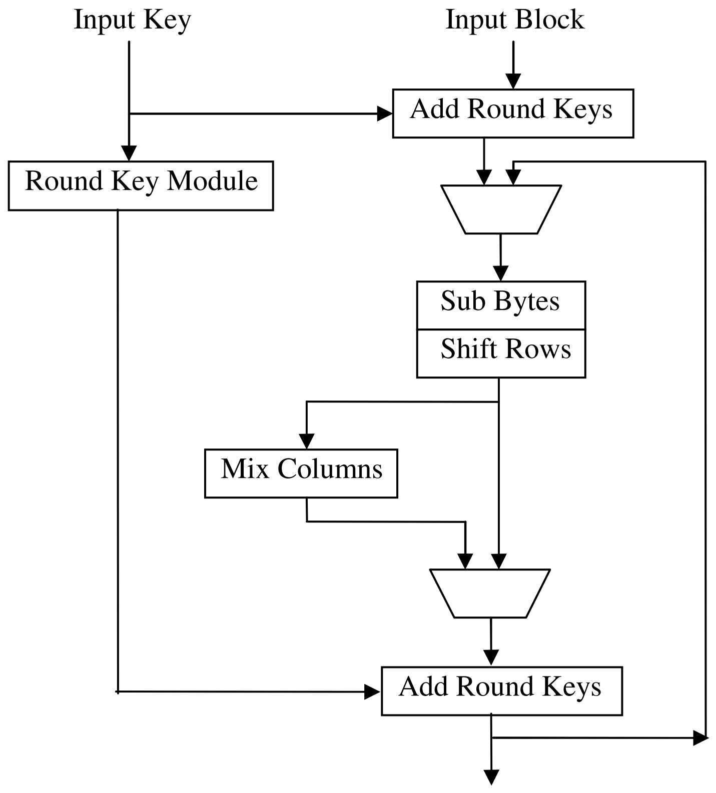


Figure 3.6: The switch structure

In another study by Alireza Hodjat, David D. Hwang, Bocheng Lai, Kris Tiri and Ingrid Verbauwhede [13] an AES crypto processor, which can handle both feedback and non feedback modes of operation is presented. It is reported that this implementation can achieve a throughput of 3.84 Gbps at a 330 MHz clock frequency. For the implementation of the non-feedback modes of the operation the design has a non-pipelined structure. In this design all implementation is based on the single clock cycle. All rounds are designed for this purpose. In the SBox operation the LUT are used and in the MixColumns operation there are used a chain of XORs.

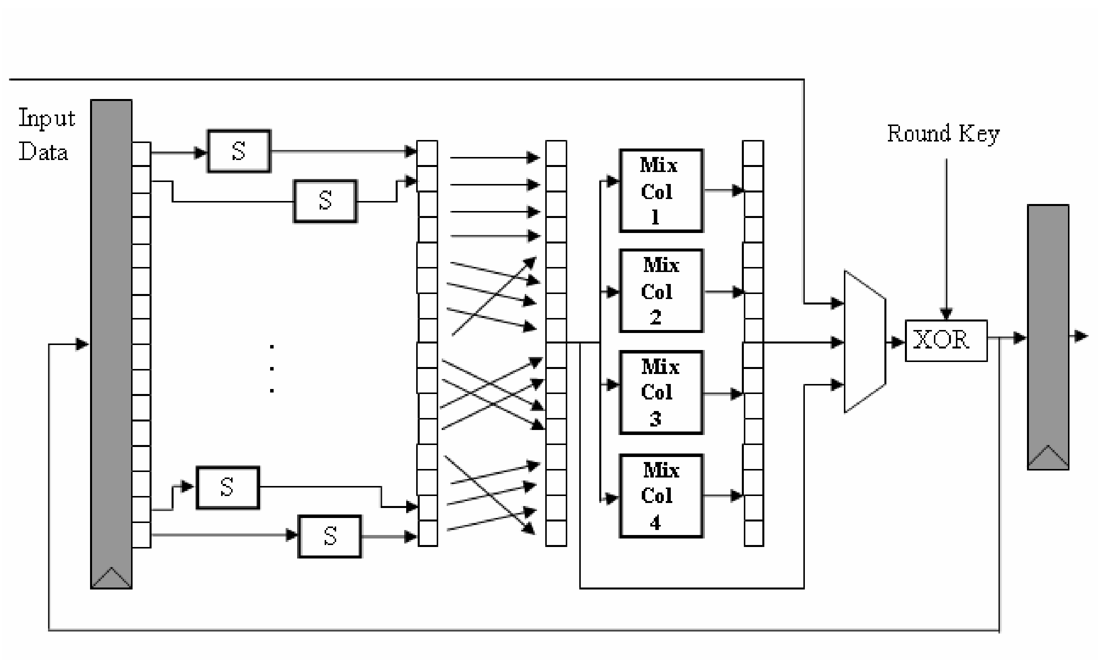


Figure 3.7: The architecture of AES Core

The proposed crypto coprocessor can be programmed through the memory-mapped interface of an embedded CPU core. The embedded CPU core can read or write to the registers by accessing different memory locations. The memory-mapped interface decodes the memory addresses and updates the registers' values.

Another configurable AES processor and its experimental results are presented by Chih-Pin Su, Chia-Lung Horng, Chih-Tsun Huang and Cheng-Wen Wu [14]. This study proposes a configurable AES processor, which can run both the original AES and the extended AES algorithm. The extended AES algorithm has some additional properties like providing some flexibility to the configuring to parameters of each transform defined in AES. They provide the flexibility by configuring the parameters given below;

1. Irreducible polynomial in SBox
2. Fixed polynomial values in MixColumns
3. Affine transformation in SBox

The AES core is called AESTHETIC. The original AES algorithm and the extended AES algorithm are reconfigured depending on the application.

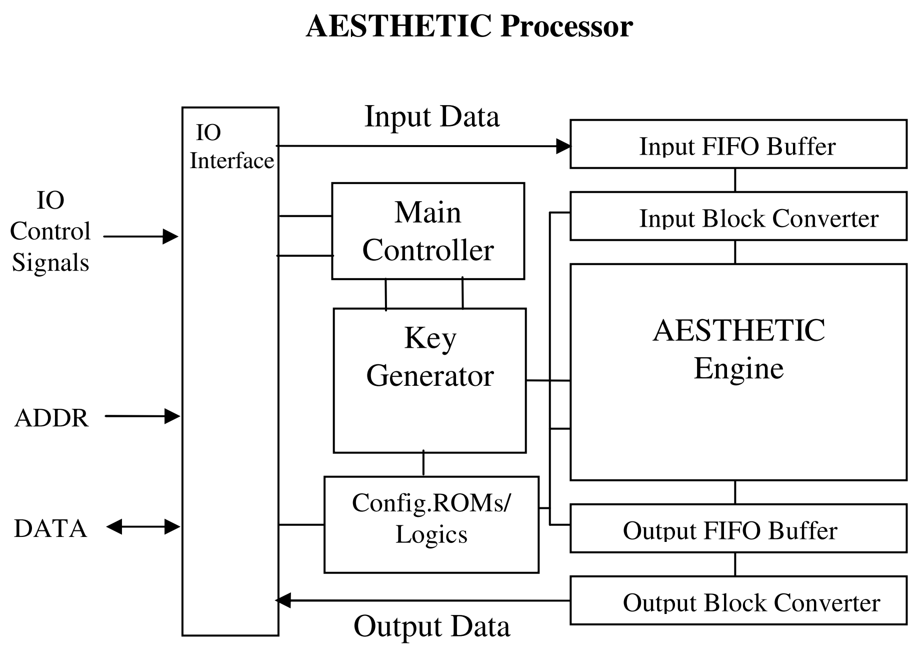


Figure 3.8: Block diagram of the AESTHETIC processor

The AESTHETIC core is similar like the original AES operations. In the design the SBox operation is implemented as using Galois Field arithmetic operations. And for the MixColumns operation there are 64 GF $((2^4)^2)$ multipliers to process the data block in parallel. The design generates the AddRoundKeys on the fly method.
The implementation results of this design are;

844.8 Mbps for 128-bit keys @66MHz clock frequency
704.0 Mbps for 192-bit keys @66MHz clock frequency
603.4 Mbps for 256-bit keys @66MHz clock frequency

In another study by Refik Sever, A. Neslin İsmailoğlu, Yusuf C. Tekmen and Murat Aşkar [6] the VLSI design and implementation of Rijndael algorithm is presented. In this study, both of the encryption and decryption algorithms are implemented for all data and key sizes on a single ASIC, with a non pipelined structure. The main diagram of the implemented architecture is given in Figure 3.9.

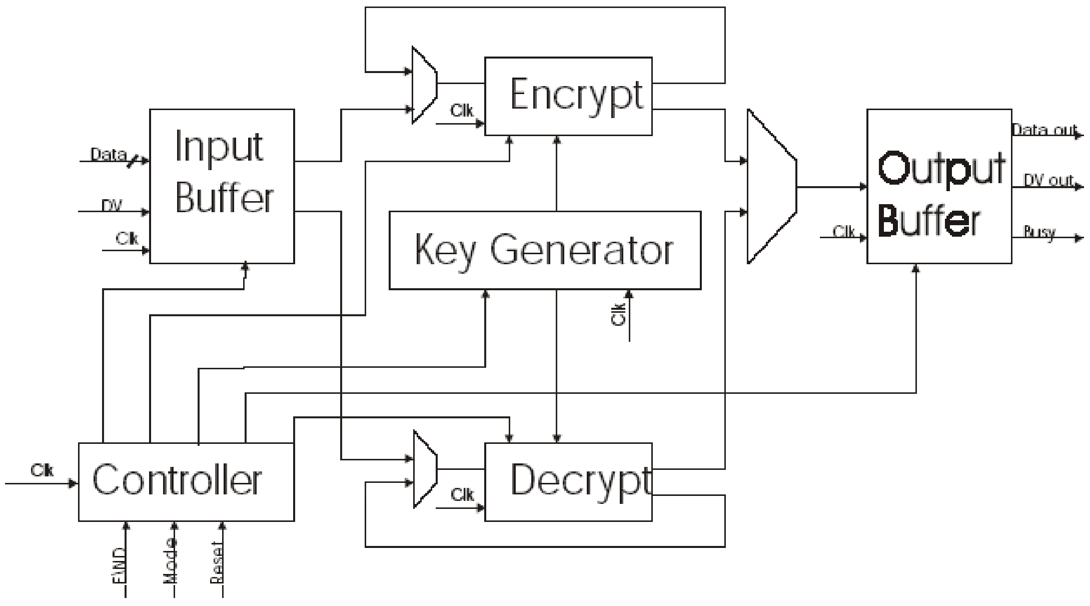


Figure 3.9: Block diagram of the implemented architecture

A single round of the algorithm is completed in one clock cycle. There are 32-S box to complete one round of the algorithm in one clock cycle.

The SBox part is implemented using combinatorial logic instead of using Look up Table. There are two separate EXOR blocks. The last round of the encryption of the current block and the first round of the encryption of the next block are processed at the same time. This two separate EXOR blocks are necessary for not loosing one clock cycle. Key Generator module consists of three sub modules: Key Expansion module, Key Storage module and Key selection module. All the keys needed for encryption and decryption processes are produced by Key Expansion module and stored by Key Storage module.

All the keys are generated and stored before encryption or decryption starts. The implementation results are given below,

- 0.35 μm CMOS technology
- Modules are described using Verilog HDL, and then synthesized with Synopsys Design Analyzer
- The chip area is 12.8 mm^2
- There are 149K gates
- The worst case clock frequency is 132 Mhz
- The maximum throughput is 2.41 Gbps

In a study by Toby Schaffer, Alan Glaser and Paul D. Franzon [18] the design and implementation of a DES processor is presented. The processor has three separate circuit, each can operate on an individual data stream to perform DES algorithm, or three can operate together to perform TDES algorithm. The block diagram of one block is given in Figure 3.10.

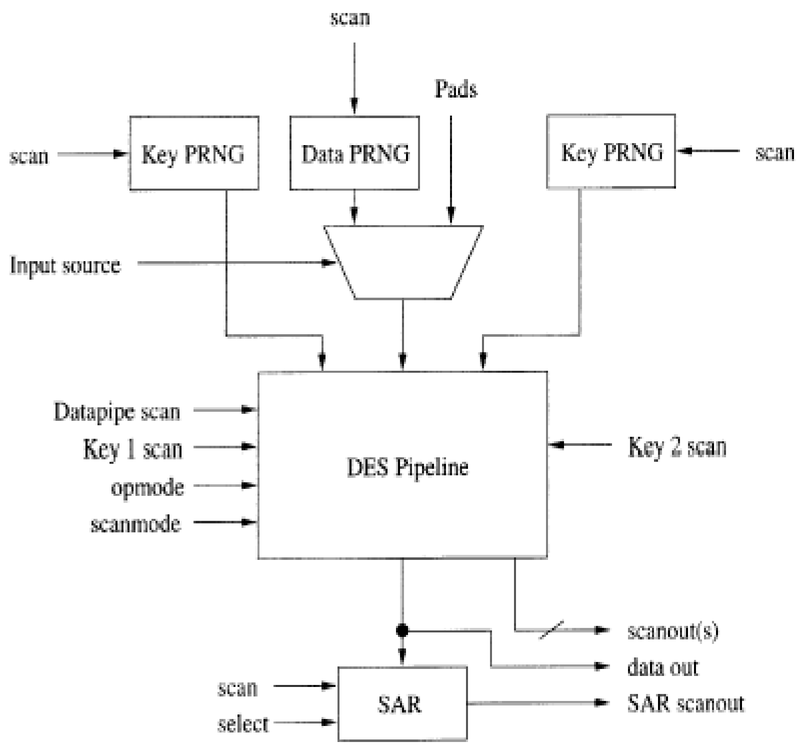


Figure 3.10: Block diagram of the one DES circuit

The iterated rounds of DES algorithm are implemented in a 16 pipelined stage structure. There are two different pseudo-random number generators (PRNG), one for key generation and one for the cipher functions. The encryption or decryption operations are chosen according to the opmode signal. This opmode signal controls the shifting of the key values to right or left side. In encryption the keys are shifted to left in a round sequence and in decryption they shifted to right to satisfy the inverse structure.

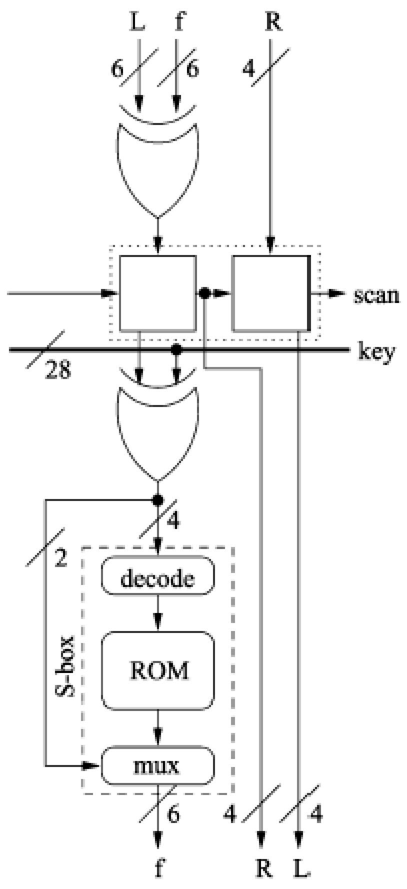


Figure 3.11: Pipelined cell structure

Figure 3.11 shows the structure of a pipelined cell structure. There are eight cells in one stage of the 16- stage pipelined structure. The operation of a single cell consist of the EXOR operations between the cipher function output data and left half data of previous stage, the EXOR operation of the key data and the result of previous EXOR operation and the SBox operation. At the output of each pipelined cell structure, the P permutation and E Table permutation are implemented separately. TDES operation throughput is reported over 7 Gb/s at 110 Mhz clock frequency, as a result of this study.

In the study by P. Kitsos, S. Goudevenos and O. Koufopavlou [19] three different hardware implementations of TDES algorithm are presented. Two of the proposed structures have pipelined structure and the third proposed structure consists of sequential iterations. In the first proposed architecture there are used 48 pipeline registers between each round to improve the throughput. The keys are shifted to the reverse direction of the encryption operation to perform decryption. In the proposed architecture initial permutations of the second and third DES and the inverse initial permutations of the first and the second DES are not implemented. As a result of this property it is reported that a gain in time delay is achieved. The key expansion is performed with using on the fly technique.

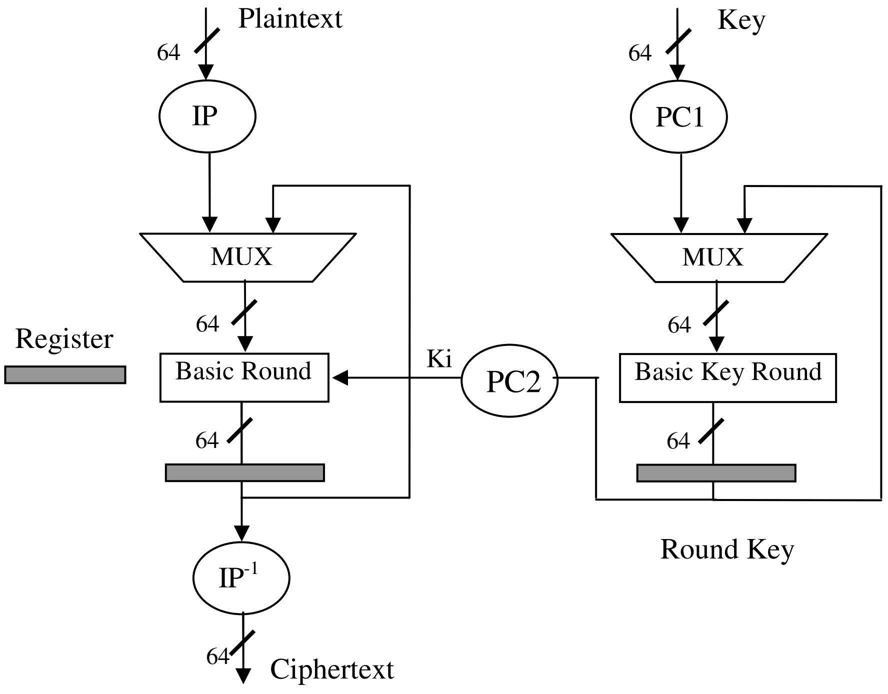


Figure 3.12: The third proposed architecture

In the second architecture 16 registers are used between the rounds. The key expansion is similar with the first architecture. This architecture has the capability of processing 16 independent data blocks simultaneously for higher throughput.

The structure of the third architecture is given in Figure 3.12. There is only one round implemented and its output of this round is registered and routed to a multiplexer. The multiplexer determines whether the output of the round is used as input or the data from permutation module is used. This architecture is proposed for are restricted applications.
The three architectures are implemented in two different Xilinx devices with using Look Up tables in one device and ROMs in the second device. The maximum throughput values are achieved on the device with ROMs and they are 7.36 Gbps for the first one, 2.45 Gbps for the second one and 121 Mbps for the third one.

3.3 Bit Permutation Instructions

Generally bit permutation operations can be performed with common instructions like “and”, “or” and “rotate”. But with these instructions the bit permutation operation in any cryptography algorithm cannot be made very efficient. Every bit of the source register is extracted from source, then placed to its new position in destination register and finally combined with other bits to make the result register. Because the complex bit permutation operations may use the common instructions for several times to form the result destination register and this operation will take to much time for a permutation operation [16]. Another way of implementing the bit permutation is using look up tables. In this type of operation there should be only one table with 2^n elements, each element is n bits, or m look up tables, with $2^{(n/m)}$ elements in each table. For example to permute 16 bits data one table can be used with 2^{16} elements, each element is 16 bits. Or the look up table number can be chosen as two and in this situation each table should have 2^8 elements, where each element is 16-bit wide [16] [20].

In addition of the conventional methods there are some bit permutation instructions implemented. These instructions aimed to solve the problems of the current microprocessor. Some of the most popular bit permutation instructions, like GRP, CROSS, OMFLIP, PPERM and SWPERM are discussed in the following part.

3.3.1 GRP instruction

GRP instruction [22] is very similar to the current microprocessor instructions structure with two operands and one result. The GRP instruction is defined as below;

GRP R3, R1, R2
R3: destination register
R1: source register
R2: source register

The data bits are divided into two groups, a left group and a right group, according to the value of the control bits. If the bit "i" in control bits is 0 the bit "i" in data goes to the left group, and it goes to the right group otherwise.
During this process, the relative positions of bits within the same group do not change. The Figure 3.13 gives an 8-bit GRP operation. In this operation, since the control bits of b, c, e, and h are 0, these four bits are placed in the left group in result register. a, d, f, and g are placed in the right group in result register because their control bit is 1.

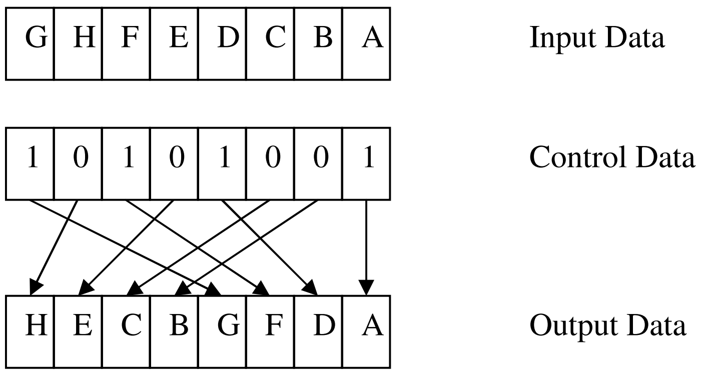


Figure 3.13: An 8-bit GRP operation

3.3.2 PPERM3R and PPERM instructions

PPERM3R and the new version PPERM [16] instructions explicitly specify the original position of each bit in the destination register. There is a control register to specify the destination positions. There should be nlg(n) bits in the control registers to permute n bit data. The PPERM3R instruction does not specify all nlg(n) bits in a single instruction. Instead, it specifies the original position only for a subset of bits, and a sequence of PPERM3R instructions specify the original position for all the bits in the destination.

The PPERM3R instruction is defined as:

PPERM3R, x Rd, Rs1, Rs2

Rs1: source register (data bits)
Rs2: source register (control bits)

Rd: destination register

x: specifies which subset in Rd will be updated with the bits extracted from Rs1.

The bits to be updated are consecutive. Except the bits in the subset, other bits in Rd remain unchanged.

3.3.3 CROSS instruction

The CROSS instruction [21] is formed by concatenating a butterfly network and an inverse butterfly network. As a property of the butterfly network a bit at any position in input, can be directed to any position in output with proper connections in network stages.

An n-bit butterfly network consists of lg(n) stages. In each stage, n bits are divided into n/2 pairs. Two bits are controlled by one control bit. Each one of the two bits are located in different pairs, and the control bit determines, whether these bits are kept their positions in the next stage or exchange their bit positions with the other bit. So n/2 control bits are needed in each stage to specify the path for n/2 data pairs. The stages in a butterfly network are differentiated by how bits are paired. In the first stage of butterfly network the distance between paired bits is n/2 bits. In the each following stage this distance is reduced by a factor two. For example for an 8-bit permutation the distance between paired bits is 4 bits. And in the second stage the distance is reduced by two to 2 bits and so on.

The inverse butterfly network can be constructed by reversing the stages in a butterfly network. The last stage in the butterfly network, for example, becomes the first stage in the inverse butterfly network.

The CROSS instruction is defined as:

CROSS, m1, m2 Rd, Rs, Rc

Rs: source register (data bits)

Rc: source register (control bits)

Rd: destination register

m1: the lower n/2 bits of the control bits

m2: the higher n/2 bits of the control bits

Figure 3.14 shows the combination of the Butterfly network and Inverse Butterfly network for 8 bits permutation.

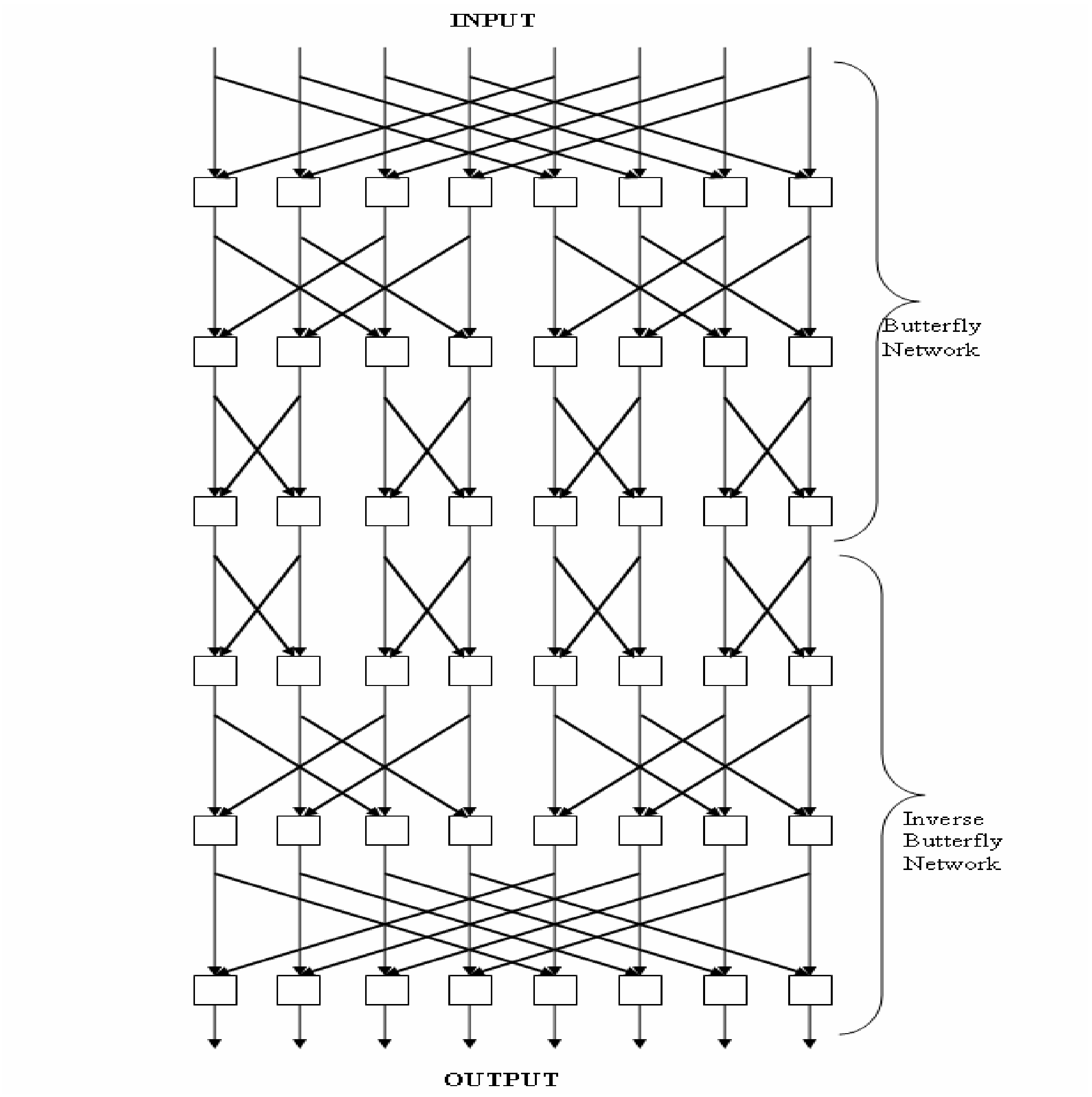


Figure 3.14: An 8-bit Benes network for CROSS instruction

3.3.4 OMFLIP instruction

The OMFLIP instruction [21] structure is similar to CROSS instruction structure. The OMFLIP is constructed by combination of an omega network and a flip network. A flip network is the mirror image of a network. For n bits input data both networks have lg(n) identical stages. An OMFLIP instruction permutes bits with two stages of an omega-flip network, and lg(n) instructions can perform an arbitrary n-bit permutation. This looks similar to the CROSS instruction.

But the difference is the omega flip network has only two distinct stages, because all omega stages and flip stages are identical.

Hence, only two omega stages and two flip stages are enough to do the OMFLIP instructions. Unlike CROSS, the number of stages does not depend on the number of bits to be permuted; only four stages are sufficient to implement an OMFLIP instruction for any word size.

3.3.5 SWPERM and SIEVE instruction

The SWPERM instruction [23] is similar to the PPERM instruction. The difference from PPERM is the fixed subword size in the SWPERM. The subword size in the SWPERM instructions is fixed at four bits.

The SWPERM instruction is given in the Figure 3.15. The positions of the bits in the destination are exactly specified with the control bits of SWPERM instruction.

For a 64-bit data permutation there are sixteen 4-bit subwords. Four bits are used to identify one of the sixteen 4-bit subwords in the source register. In total, sixteen 4-bit subwords in the destination register need exactly 64 bits, which can be put in the second operand.

The instruction SWPERM is defined as:

SWPERM Rd, Rs, Rc

Rs: source register (data bits)
Rc: source register (control bits)
Rd: destination register

A single SWPERM instruction can perform arbitrary permutation of subwords of size four bits or greater.

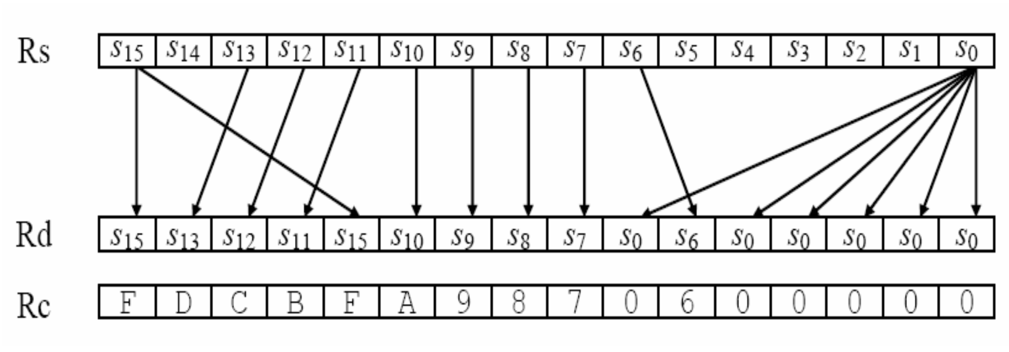

Rs
s_{15} s_{14} s_{13} s_{12} s_{11} s_{10} s_9 s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
Rd
s_{15} s_{13} s_{12} s_{11} s_{15} s_{10} s_9 s_8 s_7 s_0 s_6 s_0 s_0 s_0 s_0 s_0
Rc
F D C B F A 9 8 7 0 6 0 0 0 0 0

Figure 3.15: SWPERM instruction

CHAPTER 4

IMPLEMENTATION OF THE CRYPTO PROCESSOR

4.1 Introduction

In this chapter, the architecture and implementation of the Crypto Processor for the Advanced Encryption Standard (AES) and Data Encryption Standard (DES) algorithms are explained. The main blocks of the architecture are described in the following section. After that the Instruction Set Architecture is presented for this implemented architecture. Finally the simulation results are given in the last section.

4.2 Architecture of the Crypto Processor

The implemented architecture is based on a combination of different modules. The main architecture and its modules are given in Figure 4.1. The modules are;

- Control Unit Module
- Data Input Output Module
- Memory Module
- Arithmetic Logic Unit (ALU) Module
- Permutation Module

The properties of each module are discussed in the following sections.

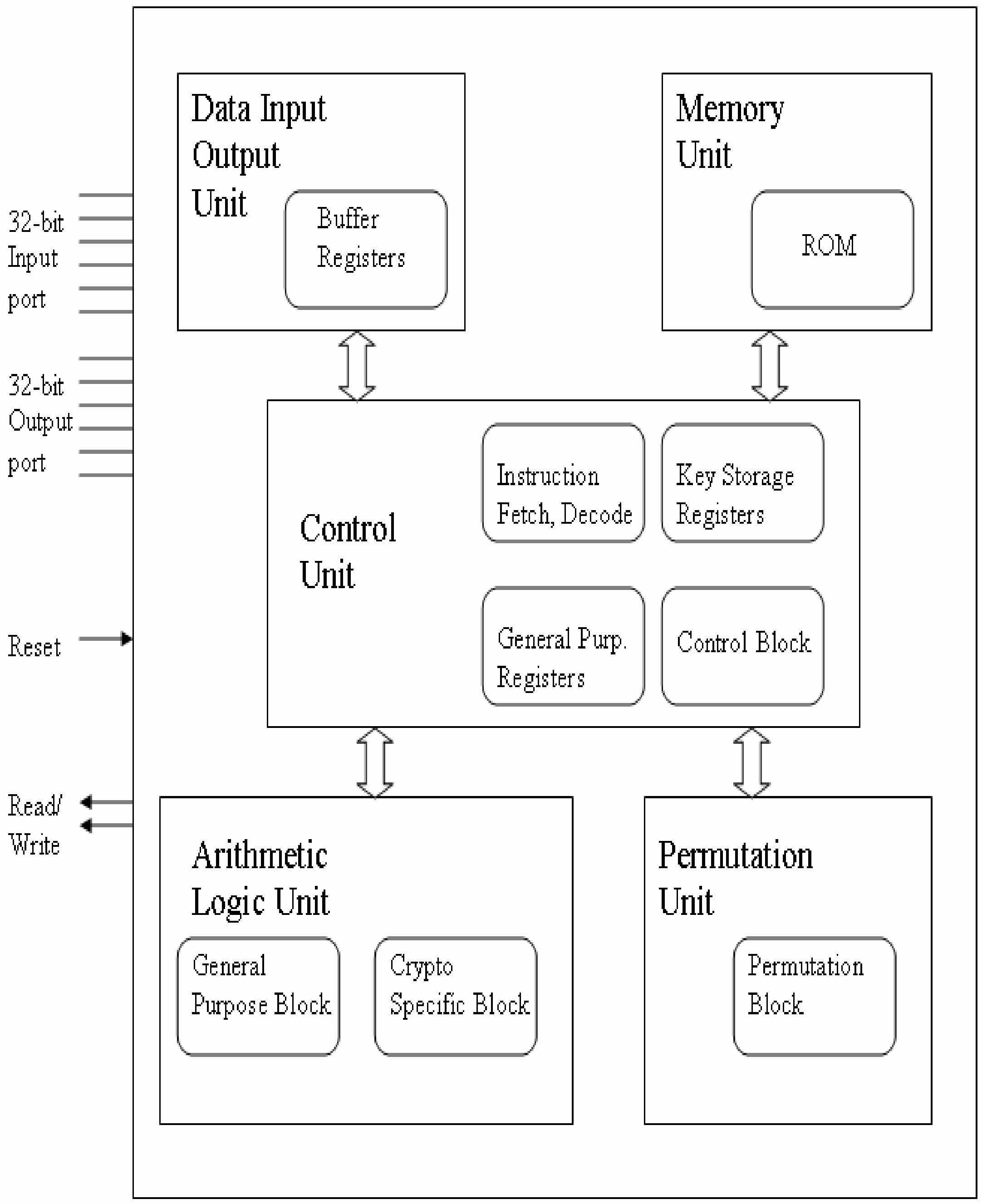


Figure 4.1: The main architecture of the implemented Crypto Processor

4.3 Control Unit Module

Control Unit is the main module of the architecture. The main function of this module is to control the others modules activities according to the instructions. The instructions are fetched from the Memory module, and then the fetched instruction is decoded and copied to the internal registers.

After the decode process of the instruction the execution process is activated. In this process the Control Unit sends the proper control signal to the related modules with the proper operand data. Then the result data is copied back to related registers to finish one instruction operation.

One machine cycle in the implemented architecture consists of 4 clock cycles. During a machine cycle, fetch, decode and execute operations are performed. The most of the instructions are one machine cycle instructions, but some of the instructions are processed in three or four cycles. The Instruction Set Architecture for this implemented Crypto Processor is given in the last section of this chapter.

There are sixteen 32-bit internal data registers in the Control Unit module [15]. Half of these sixteen registers are used as general purpose registers. The number of the general purpose registers is chosen as eight, because of holding all the state data info once in a register set. And the maximum data length in AES and DES algorithm is 256 bits, which is used in AES algorithm.

The remaining eight data registers are used just for data storing of internal state values of general purpose registers. These are necessary because in some operations the next state value is calculated with the operations between the previous state value and some intermediate values obtained by previous state value.

There is also a different register block in the Control Unit module. This register block is used to store the round key values for the performed crypto algorithm. The number of the register block is determined in respect of the maximum necessary round key value. Therefore to supply all the round key numbers the register block consists of 120 32-bit registers.

4.4 Data Input/Output Module

The external access to the implemented Crypto processor is provided by the Data Input/Output Module. The Data I/O module has two different 32 bits external interface. One of the interfaces is assigned as input to the processor and the other one is assigned as output.

The input and output processes are performed according to the I/O commands, which are sent by Control Unit. There are eight different operations in the Data I/O module. These operations are categorized in respect of the crypto algorithms data lengths. The data lengths can have four different data length as 64-bit, 128-bit, 192-bit and 256-bit. The I/O commands specify the data length and the data are processed in sequential clock cycles. For the 64-bit input data the first 32-bit is stored in the input buffer register of the I/O module in the first clock cycle. In the second cycle this data is fed to the Control Unit internal registers, while the second 32-bit data is taken from input interface and stored in the buffer register. In the third clock cycle this 32-bit data is also transferred to the Control Unit internal registers. The input and output operations are performed in a similar way for the other data lengths.

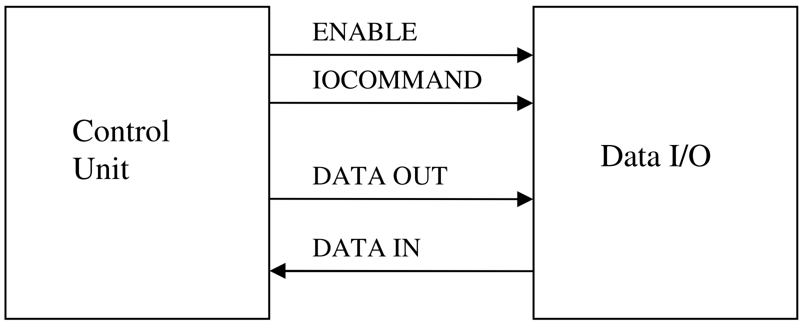


Figure 4.2: Control Unit – Data Input/Output Module Interface

4.5 Memory Module

The memory unit consists of a ROM block. The instructions are stored in the ROM block and they are subjected to the Control Unit with an 8-bit wide data link between Control Unit and Memory Module.

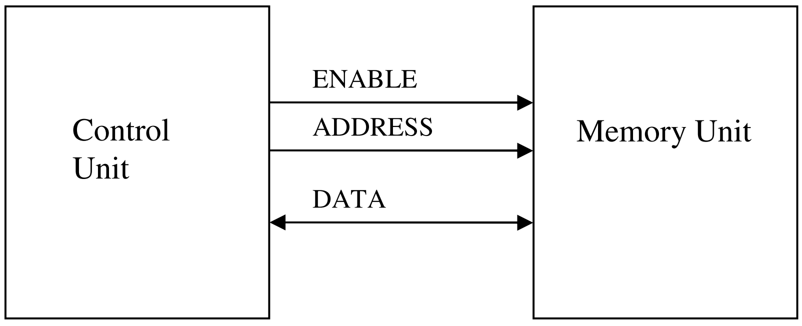


Figure 4.3: Control Unit – Memory Module Interface

The control of the read operation is provided by the enable signal. The ROM block is activated only when there is a low enable signal. The instructions are called from memory according to the Address data. The Control Unit assigns the next address data to the Address link between Control Unit and Memory Unit or determines the next address data according to some control signals, created by the last instruction.

4.6 Arithmetic Logic Unit (ALU) Module

ALU module process the incoming data according to the commands from Control Unit. The basic operations are performed in the implemented ALU module, like Boolean functions, addition – subtraction operations, shift operations. Further to that some AES and DES specific operations can also handled in the ALU module.

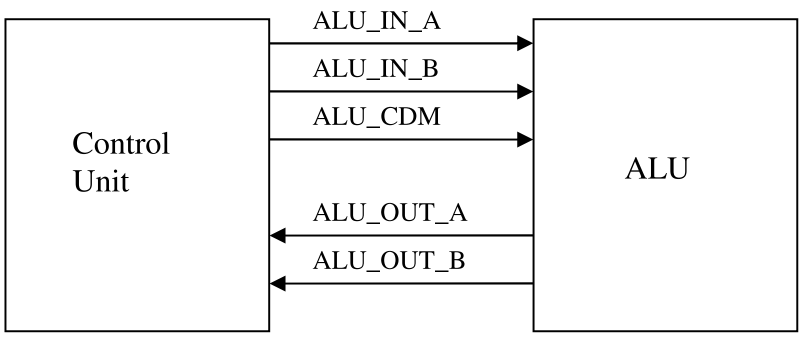


Figure 4.4: Control Unit – ALU Interface

There are four 32-bit data links between Control Unit and ALU. The current data registers values are directed from Control Unit to the ALU by two of these four 32-bit data registers (AluInA and AluInB). And the remaining two data links are from ALU to Control Unit for the output of the processed data in the ALU (AluOutA and AluOutB).

The alucmd link is for the ALU commands, created by the Control Unit module according to the decoded instruction. ALU module process the data according to these incoming commands.

The main operations are performed over 32 bits data, but there are some exceptions for both of AES and DES algorithms. There are some operations performed over bytes. In this case the incoming 32 bits data is divided into suitable data chunks and then the operations are performed.

In addition of general logic operations, the SBox operations are performed also in the ALU module. For this purpose ALU module has a memory unit, which stores the SBox values for both of AES and DES algorithms. The structure of this memory unit will be discussed in detail in the next section. The ALU command set is given in Table 4.1.

Table 4.1: ALU Commands

Code	Command
0x00	Alu_DES_ROR
0x01	Alu_DES_ROL
0x02	Alu_DES_SBOX1
0x03	Alu_SBOX
0x04	Alu_XTIME
0x05	Alu_MIX
0x06	ALU_BIT_MAP
0x07	Alu_SWAP
0x08	Alu_AES_SBOX
0x09	Alu_EXOR
0x0a	Alu_ROR_BYTE
0x0b	Alu_ROL_BYTE
0x0c	Alu_STORE0
0x0d	Alu_STORE1
0x0e	Alu_STORE2
0x0f	Alu_SHIFT128_0
0x10	Alu_SHIFT128_1
0x11	Alu_SHIFT192_0
0x12	Alu_SHIFT192_1
0x13	Alu_SHIFT192_2
0x14	Alu_SHIFT256_0
0x15	Alu_SHIFT256_1
0x16	Alu_SHIFT256_2
0x17	Alu_SHIFT256_3
0x18	Alu_NOP

4.6.1 SBox Memory Unit

The SBox operation plays an important role for both of AES and DES algorithms. The main structure of the SBox operation is different for these algorithms. In AES algorithm the SBox operation is performed over bytes. Each byte in the State matrix is replaced with a SBox table element. And the address of the table element is given directly the input byte data itself. On the other hand in the DES algorithm the SBox operation is performed with 6-bit data input and 4-bit data output. The address of the 4-bit output data is calculated according to some rules on the input 6-bit data. But when the address calculation operations of the DES are handled in a way, the next operation for both algorithms can be performed with using Look up Tables. The implemented Look up Table structure is given in Figure 4.5.

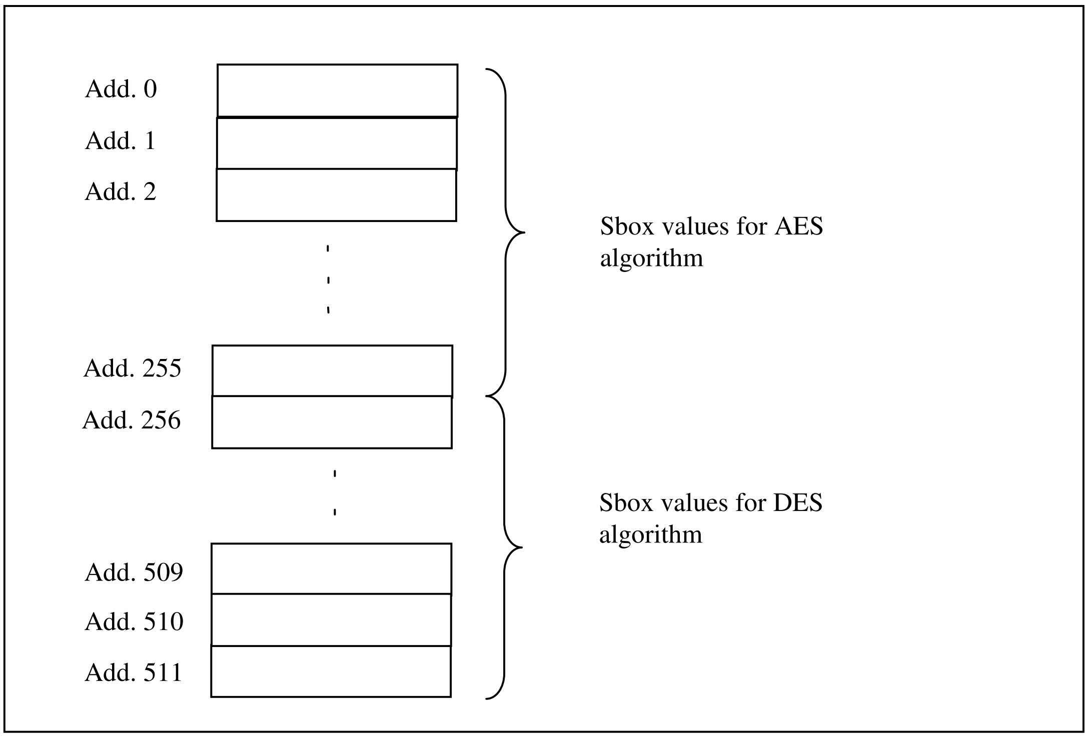


Figure 4.5: SBox memory unit

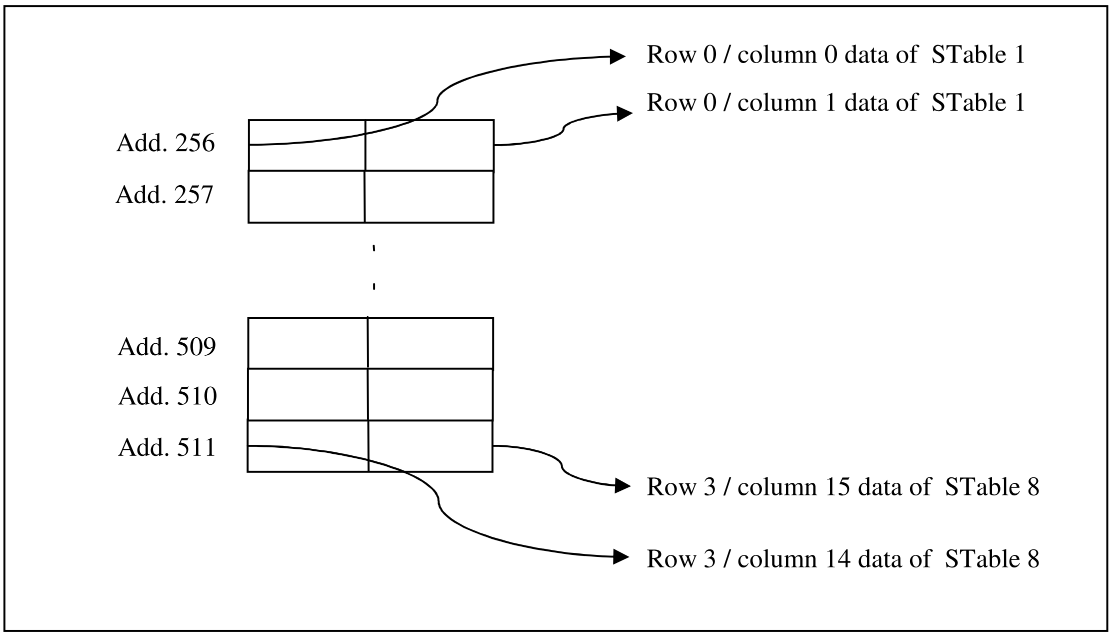
The Look up Table consists of 512 memory elements; each one is 8-bit wide. There are 256 memory elements for the AES algorithm [1]. These values for AES algorithm are stored in memory from address 0 to address 255.

There are 8 different SBox table in the DES algorithm. There are 64 memory elements; each one is 4-bit data in each table. With a proper organization these SBox data can be arranged as 32 memory element, and each element as 8-bit data. And for eight SBox tables in DES there should be 256 memory elements, which are 8-bit data. The DES algorithm SBox values are stored in memory from address 256 to address 511.

As discussed before the SBox operation for AES algorithm is performed directly with a single instruction. The data is replaced with a SBox memory element, that the address of the result data is the incoming data itself.

The SBox operation in the DES algorithm is a bit more complex than AES algorithm. The input of a SBox table is 6-bit data. The memory address of the output element is obtained by another instruction, because of providing a common use to the SBox instruction.

The output data of the SBox table is 4-bit data. But in the memory unit the data is stored as 8-bit data. Therefore the two sequential SBox table output data is stored in the memory unit in the same address. For example the output data of row 0 / column 0 and row 0 / column 1 are stored in the same memory address. The high part of the memory data is the output data of row 0 / column 0 and the low part is the output data of row 0 / column 1. After replacing the memory element with the input data of the SBox table, the high 4-bit or the low 4-bit is chosen according to some control signals.



Row 0 / column 0 data of STable 1
Row 0 / column 1 data of STable 1
Add. 256
Add. 257
Add. 509
Add. 510
Add. 511
Row 3 / column 15 data of STable 8
Row 3 / column 14 data of STable 8

Figure 4.6: SBox memory unit organization for DES

4.7 Permutation Module

Bit Permutation is an important operation in the Block ciphers. In the bit permutation operations, the incoming data is subjected to the some bit position changes according to the permutation type. The using aim of the bit permutation is mainly for the diffusing objective. With diffusion the redundancy of the plaintext data is spread over a large part of the cipher text.

The bit permutation operations have a big process part in DES and TDES algorithms. In many other solutions for DES algorithm these blocks are mainly implemented as look up tables or implemented as hardware routing for only DES unique processors or implemented with current microprocessor instructions like “and”, “rotate” and “or”. But these kind of solutions have some disadvantages like slow process time and area inefficiency.

Therefore a separate permutation module is implemented in the architecture. The main purpose of this module is directly dedicated to bit permutation operations and the main structure is based on Butterfly network structure.

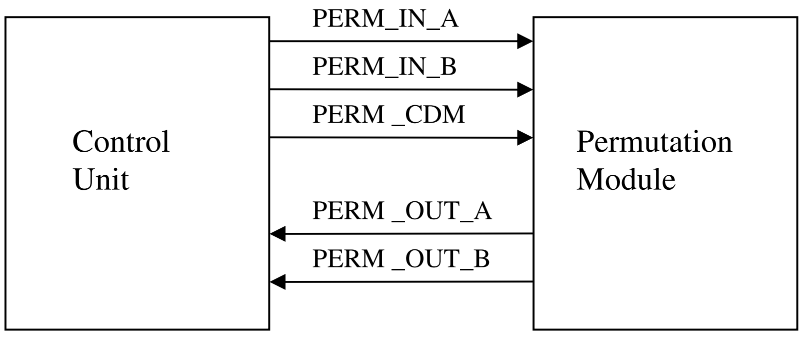
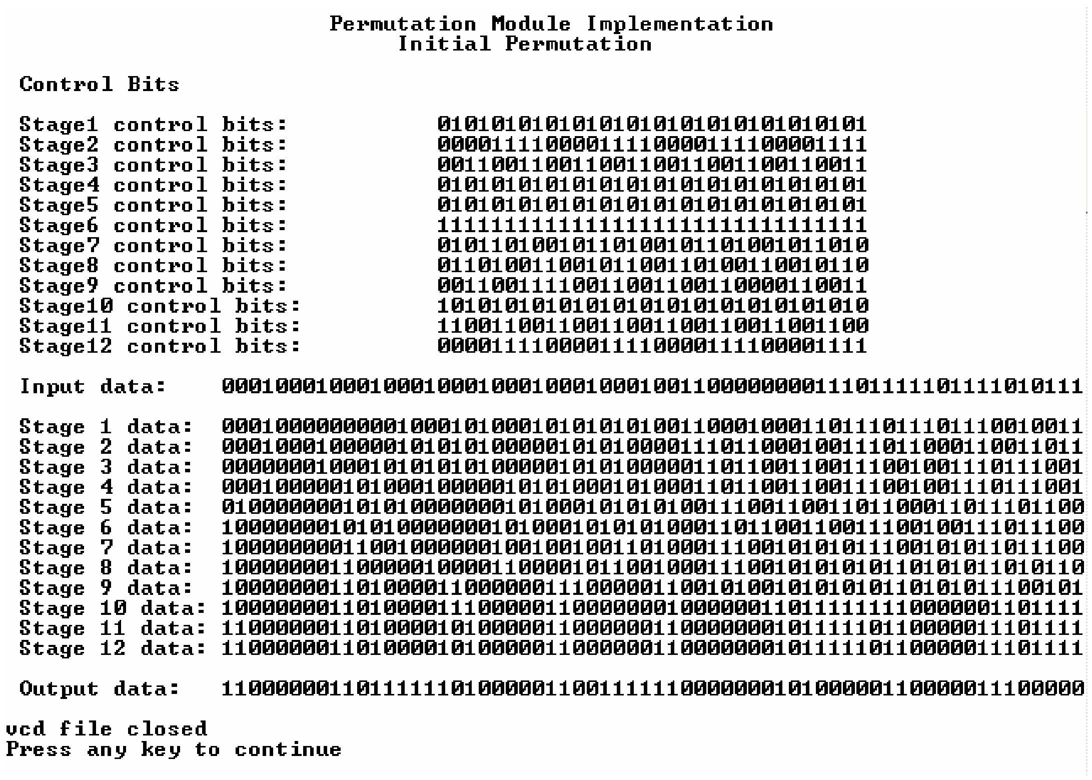


Figure 4.7: Control Unit – Permutation Module Interface

The Permutation Module interface is similar to the ALU interface. There are four 32-bit data links between the Control Unit and Permutation Module. The operation type is same with ALU. The Permutation module is activated by the permcmd signal. Besides of the activation function permcmd determines that which permutation operation is performed in the module. The interface between Control Unit and Permutation module is given in Figure 4.7. The implemented permutation module is a combination of a Butterfly network and Inverse Butterfly network [16]. This module is designed for permutations of 64 bits data. Therefore there are 12 stages in the module, 6 stages belong to Butterfly network and the remaining 6 stages belong to Inverse Butterfly network. The transition between stages is controlled by the dedicated control registers for each stage, so there are 12 control bit registers. Each control register is 32-bit wide and each control bit determines the next stage position' of two different bits in current stage.

The necessary stage control bits and the bit positions between sequential stages are given in below figures for each permutation operation of DES algorithm. In the simulation of permutation operations same input is applied to the permutation module for each permutation operation of DES. From the simulation figures of DES permutation operations, it can be seen easily that, there are a different bit transitions map between sequential network stages according to the relevant permutation operation control bits.



```
                        Permutation Module Implementation
                             Initial Permutation

 Control Bits

 Stage1 control bits:          01010101010101010101010101010101
 Stage2 control bits:          00001111000011110000111100001111
 Stage3 control bits:          00110011001100110011001100110011
 Stage4 control bits:          01010101010101010101010101010101
 Stage5 control bits:          01010101010101010101010101010101
 Stage6 control bits:          11111111111111111111111111111111
 Stage7 control bits:          01011010010110100101101001011010
 Stage8 control bits:          01101001100101100110100110010110
 Stage9 control bits:          00110011110011001100110000110011
 Stage10 control bits:         10101010101010101010101010101010
 Stage11 control bits:         11001100110011001100110011001100
 Stage12 control bits:         00001111000011110000111100001111

 Input data:    0001000100010001000100010001000100110000000011101111101111010111

 Stage 1 data:  0001000000000100010100010101010100110001000110111011101110010011
 Stage 2 data:  0001000100000101010100000101010000111011000100111011000110011011
 Stage 3 data:  0000000100010101010100000101010000011011001100111001001110111001
 Stage 4 data:  0001000001010001000001010100010100011011001100111001001110111001
 Stage 5 data:  0100000001010100000001010001010101001110011001101100011011101100
 Stage 6 data:  1000000010101000000010100010101010001101100110011100100111011100
 Stage 7 data:  1000000001100100000010010010011010001110010101011100101011011100
 Stage 8 data:  1000000011000001000011000010110010001110010101010110101011010110
 Stage 9 data:  1000000011010000110000001110000011001010010101010110101011100101
 Stage 10 data: 1000000011010000111000001100000001000000110111111110000001101111
 Stage 11 data: 1100000011010000101000001100000011000000010111110110000011101111
 Stage 12 data: 1100000011010000101000001100000011000000010111110110000011101111

 Output data:   1100000011011111101000001100111111000000010100000110000011100000

vcd file closed
Press any key to continue
```

Figure 4.8: DES Initial Permutation

The stage control bits for Initial Permutation in the permutation module are;

Initial_Perm_Control[0] = 0x55555555;

Initial_Perm_Control[1] = 0x0f0f0f0f;

Initial_Perm_Control[2] = 0x33333333;

Initial_Perm_Control[3] = 0x55555555;

Initial_Perm_Control[4] = 0x55555555;

Initial_Perm_Control[5] = 0xffffffff;

Initial_Perm_Control[6] = 0x5a5a5a5a;

Initial_Perm_Control[7] = 0x69966996;

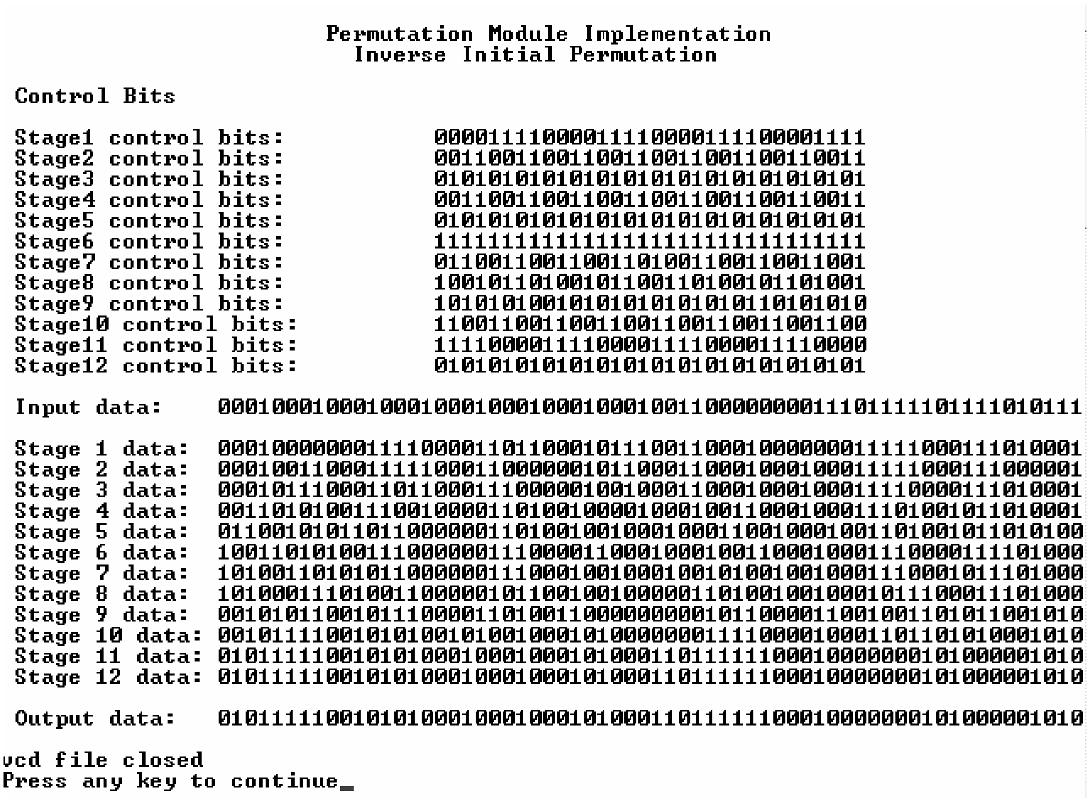
Initial_Perm_Control[8] = 0x33cccc33;

Initial_Perm_Control[9] = 0xaaaaaaaa;

Initial_Perm_Control[10] = 0xcccccccc;

Initial_Perm_Control[11] = 0x0f0f0f0f;

The Initial Permutation is performed, when the permute command is 0x01 and then these values are transferred to the stage control bits to process the data correctly.



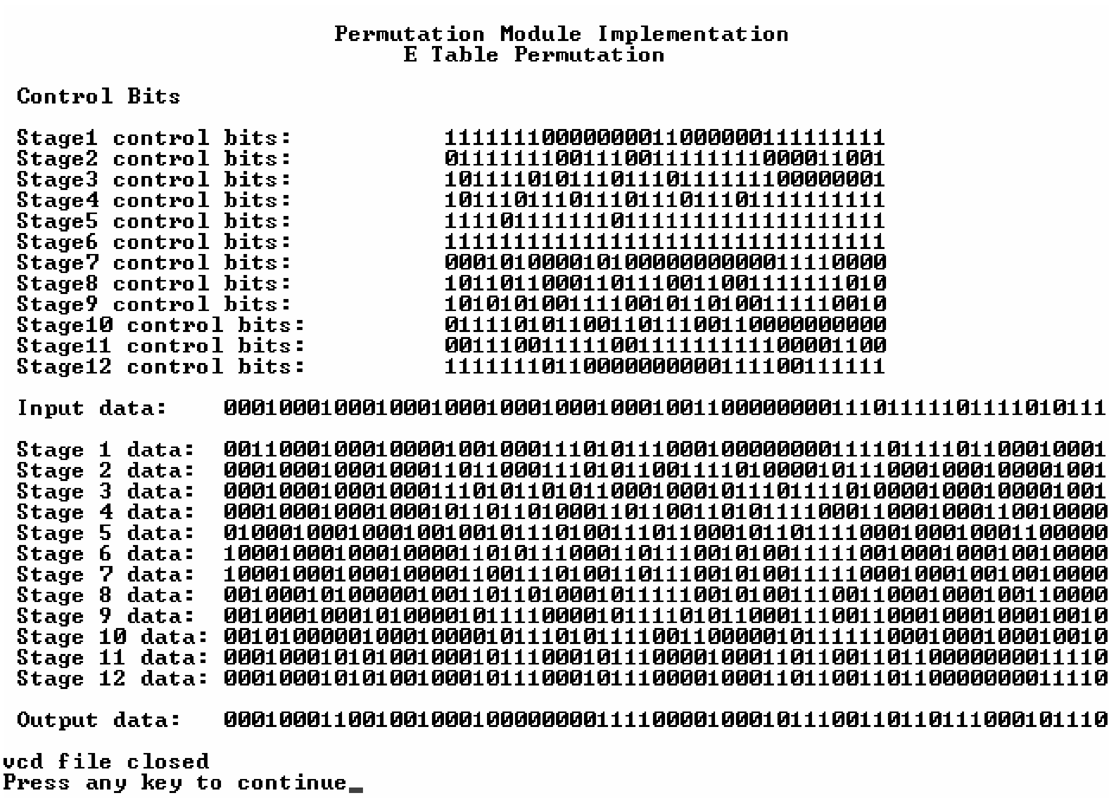
```
                         Permutation Module Implementation
                           Inverse Initial Permutation

 Control Bits

 Stage1 control bits:           00001111000011110000111100001111
 Stage2 control bits:           00110011001100110011001100110011
 Stage3 control bits:           01010101010101010101010101010101
 Stage4 control bits:           00110011001100110011001100110011
 Stage5 control bits:           01010101010101010101010101010101
 Stage6 control bits:           11111111111111111111111111111111
 Stage7 control bits:           01100110011001101001100110011001
 Stage8 control bits:           10010110100101100110100101101001
 Stage9 control bits:           10101010010101010101010110101010
 Stage10 control bits:          11001100110011001100110011001100
 Stage11 control bits:          11110000111100001111000011110000
 Stage12 control bits:          01010101010101010101010101010101

 Input data:    00010001000100010001000100010001001100000000111011111101111010111

 Stage 1 data:  000100000000111100001101100010111001100010000000011111000111010001
 Stage 2 data:  0001001100011111000110000001011000110001000100011111000111000001
 Stage 3 data:  00010111000110110001110000001001000110001000100011110000111010001
 Stage 4 data:  0011010100111001000011010010000100010011000100011101001011010001
 Stage 5 data:  0110010101101100000011010010010001000110010001001101001011010100
 Stage 6 data:  1001101010011100000011100001100010001001100010001110000111101000
 Stage 7 data:  1010011010101100000011100010010001001010010010001110001011101000
 Stage 8 data:  10100011101001100000010110010010000011010010010001011100011101000
 Stage 9 data:  00101011001011100001101001100000000001011000011001001101011001010
 Stage 10 data: 0010111100101010010100100010100000001111000010001101101010001010
 Stage 11 data: 0101111100101010001000100010100011011111100010000000101000001010
 Stage 12 data: 0101111100101010001000100010100011011111100010000000101000001010

 Output data:   0101111100101010001000100010100011011111100010000000101000001010

vcd file closed
Press any key to continue_
```

Figure 4.9: DES Inverse Initial Permutation

The stage control bits for Inverse Initial Permutation in the permutation module are;

```
Inverse_Initial_Perm_Control[0]  = 0x0f0f0f0f;
Inverse_Initial_Perm_Control[1]  = 0x33333333;
Inverse_Initial_Perm_Control[2]  = 0x55555555;
Inverse_Initial_Perm_Control[3]  = 0x33333333;
Inverse_Initial_Perm_Control[4]  = 0x55555555;
Inverse_Initial_Perm_Control[5]  = 0xffffffff;
Inverse_Initial_Perm_Control[6]  = 0x66669999;
Inverse_Initial_Perm_Control[7]  = 0x96966969;
Inverse_Initial_Perm_Control[8]  = 0xaa5555aa;
Inverse_Initial_Perm_Control[9]  = 0xcccccccc;
Inverse_Initial_Perm_Control[10] = 0xf0f0f0f0;
Inverse_Initial_Perm_Control[11] = 0x55555555;
```

The permute command should be 0x03 for the Inverse Initial Permutation operation.



```
                         Permutation Module Implementation
                              E Table Permutation

 Control Bits

 Stage1 control bits:           11111100000000011000000111111111
 Stage2 control bits:           01111110011100111111111000011001
 Stage3 control bits:           10111101011101110111111100000001
 Stage4 control bits:           10111011101110111011101111111111
 Stage5 control bits:           11110111111101111111111111111111
 Stage6 control bits:           11111111111111111111111111111111
 Stage7 control bits:           00010100000101000000000001110000
 Stage8 control bits:           10110110001101110011001111111010
 Stage9 control bits:           10101010011110010110100111110010
 Stage10 control bits:          01111010110011011100110000000000
 Stage11 control bits:          00111001111100111111111100001100
 Stage12 control bits:          11111110110000000000111100111111

 Input data:    000100010001000100010001000100010001001100000000111011111011111010111

 Stage 1 data:  001100010001000010010001110101110001000000001111011110110001000 1
 Stage 2 data:  000100010001000110110001110101100111101000001011100010001000010 01
 Stage 3 data:  000100010001000111010110101100010001011101111010000100010000100 1
 Stage 4 data:  000100010001000101101101000110110011010111100011000100011001000 0
 Stage 5 data:  010001000100010010010111010011101100010110111100010001000110000 0
 Stage 6 data:  100010001000100001101011100011011100101001111100100010001001000 0
 Stage 7 data:  100010001000100001100111010011011100101001111100010001001001000 0
 Stage 8 data:  001000101000001001101101000101111100101001110011000100010011000 0
 Stage 9 data:  001000100010100001011110000101111010110001110011000100010001001 0
 Stage 10 data: 001010000010001000010111010111100110000010111111000100010001001 0
 Stage 11 data: 000100010101001000101110001011100001000110110011011000000000111 0
 Stage 12 data: 000100010101001000101110001011100001000110110011011000000000111 0

 Output data:   000100011001001000100000000111100001000101110011011011100010111 0

vcd file closed
Press any key to continue_
```

Figure 4.10: DES E Table Permutation

The stage control bits for E Table Permutation in the permutation module are;

EBit_Selection_Control[0] = 0xfe0181ff;

EBit_Selection_Control[1] = 0x7f39fe19;

EBit_Selection_Control[2] = 0xbd777f01;

EBit_Selection_Control[3] = 0xbbbbbbff;

EBit_Selection_Control[4] = 0xf7f7ffff;

EBit_Selection_Control[5] = 0xffffffff;

EBit_Selection_Control[6] = 0x142800f0;

EBit_Selection_Control[7] = 0xb63733fa;

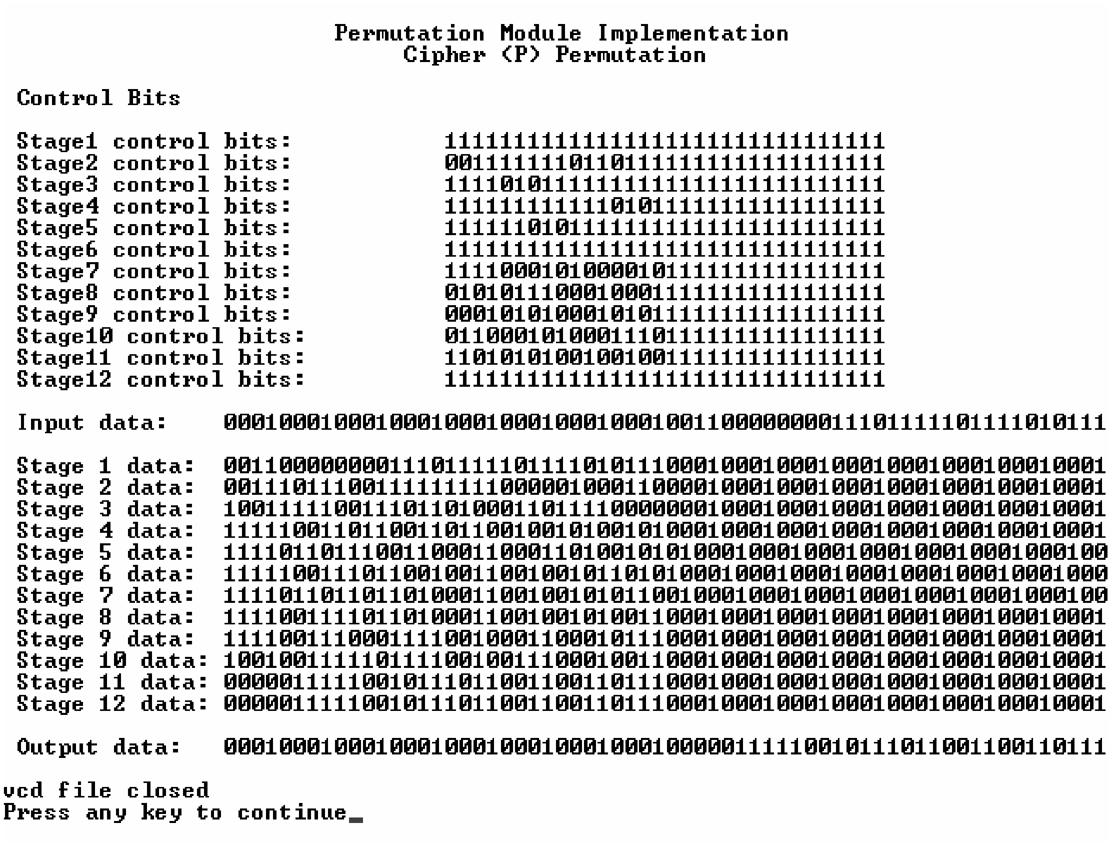
EBit_Selection_Control[8] = 0xaa7969f2;

EBit_Selection_Control[9] = 0x7acdcc00;

EBit_Selection_Control[10] = 0x39f3ff0c;

EBit_Selection_Control[11] = 0xfec00f3f;

The permute command should be 0x0b for the E Table Permutation operation.



```
                    Permutation Module Implementation
                         Cipher (P) Permutation

 Control Bits

 Stage1 control bits:          11111111111111111111111111111111
 Stage2 control bits:          00111111101101111111111111111111
 Stage3 control bits:          11110101111111111111111111111111
 Stage4 control bits:          11111111111101011111111111111111
 Stage5 control bits:          11111101011111111111111111111111
 Stage6 control bits:          11111111111111111111111111111111
 Stage7 control bits:          11110001010000101111111111111111
 Stage8 control bits:          01010111000100011111111111111111
 Stage9 control bits:          00010101000101011111111111111111
 Stage10 control bits:         01100010100011101111111111111111
 Stage11 control bits:         11010101001001001111111111111111
 Stage12 control bits:         11111111111111111111111111111111

 Input data:    000100010001000100010001000100110000000011101111101111010111

 Stage 1 data:  0011000000001110111110111101011100010001000100010001000100010001
 Stage 2 data:  0011101110011111111100000100011000010001000100010001000100010001
 Stage 3 data:  1001111100111011010001101111000000001000100010001000100010001
 Stage 4 data:  1111100110110011011001001010010100010001000100010001000100010001
 Stage 5 data:  11110110111001100011000110100101010001000100010001000100010001000100
 Stage 6 data:  1111100111011001001100100101101010001000100010001000100010001000
 Stage 7 data:  11110110110110100011001001010110010001000100010001000100010001000100
 Stage 8 data:  1111001111011010001100100101001100010001000100010001000100010001
 Stage 9 data:  1111001110001111001000110001011100010001000100010001000100010001
 Stage 10 data: 1001001111101111001001110001001100010001000100010001000100010001
 Stage 11 data: 0000011111001011101100110011011100010001000100010001000100010001
 Stage 12 data: 0000011111001011101100110011011100010001000100010001000100010001

 Output data:   0001000100010001000100010001000100000111110010111011001100110111

vcd file closed
Press any key to continue_
```

Figure 4.11: DES Cipher (P) Permutation

The stage control bits for Cipher (P) Permutation in the permutation module are;

CipPer_Perm_Control[0] = 0xffffffff;

CipPer_Perm_Control[1] = 0x3fb7ffff;

CipPer_Perm_Control[2] = 0xf5ffffff;

CipPer_Perm_Control[3] = 0xfff5ffff;

CipPer_Perm_Control[4] = 0xfd7fffff;

CipPer_Perm_Control[5] = 0xffffffff;

CipPer_Perm_Control[6] = 0xf142ffff;

CipPer_Perm_Control[7] = 0x5711ffff;

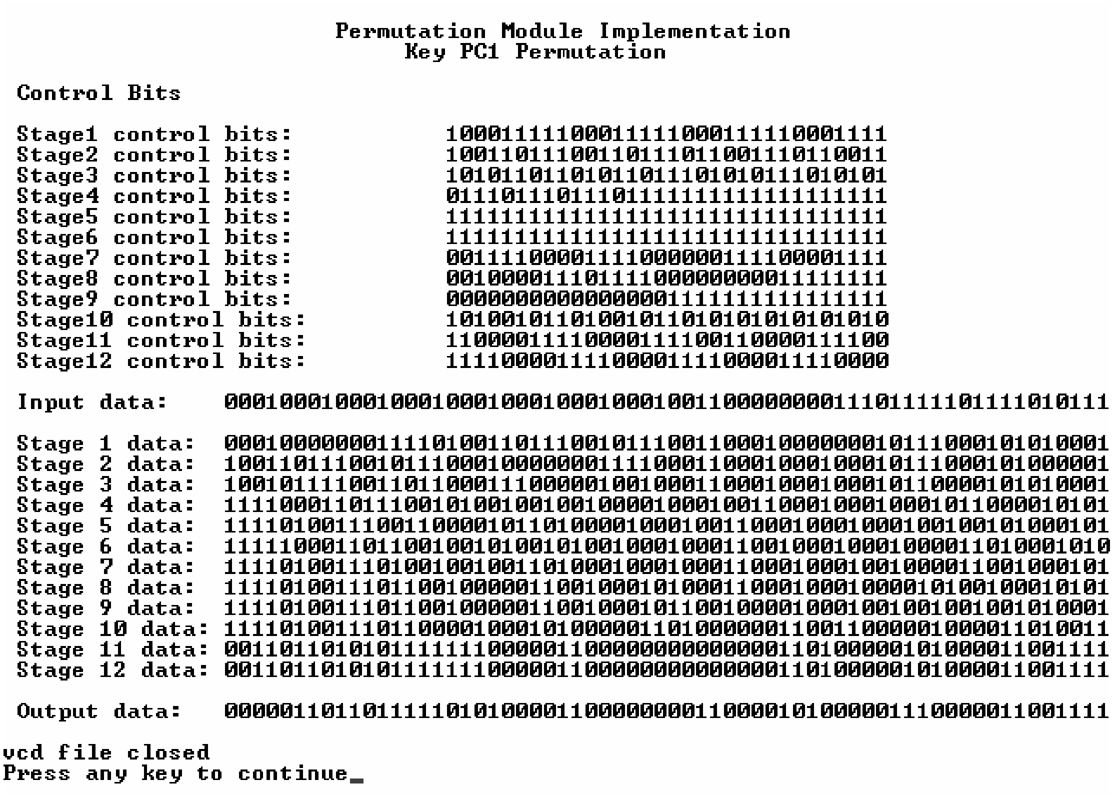
CipPer_Perm_Control[8] = 0x1515ffff;

CipPer_Perm_Control[9] = 0x628effff;

CipPer_Perm_Control[10] = 0xd524ffff;

CipPer_Perm_Control[11] = 0xffffffff;

The permute command should be 0x05 for the Cipher (P) Permutation operation.



```
                         Permutation Module Implementation
                              Key PC1 Permutation

 Control Bits

 Stage1 control bits:           10001111100011111000111110001111
 Stage2 control bits:           10011011100110111011001110110011
 Stage3 control bits:           10101101101011011101010111010101
 Stage4 control bits:           01110111011101111111111111111111
 Stage5 control bits:           11111111111111111111111111111111
 Stage6 control bits:           11111111111111111111111111111111
 Stage7 control bits:           00111100001111000000111100001111
 Stage8 control bits:           00100001110111100000000011111111
 Stage9 control bits:           00000000000000001111111111111111
 Stage10 control bits:          10100101101001011010101010101010
 Stage11 control bits:          11000011110000111100110000111100
 Stage12 control bits:          11110000111100001111000011110000

 Input data:     0001000100010001000100010001000100110000000011101111110111101011 1

 Stage 1 data:  00010000000011110100110111001011100110001000000001011100010101 0001
 Stage 2 data:  10011011100101110001000000001111000110001000100010111000101000001
 Stage 3 data:  10010111100110110001110000010010001100010001000101100001010 10001
 Stage 4 data:  11110001101110010100100100100001000100110001000100010110000 10101
 Stage 5 data:  111101001110011000010110100000100010011000100010001001001010 00101
 Stage 6 data:  11111000110110010010100101001000100011001000100010000110100 01010
 Stage 7 data:  1111010011101001001001101000100010001100010001001000011001000101
 Stage 8 data:  11110100111011001000000110010001010001100010001000010100100010101
 Stage 9 data:  1111010011101100100000011001000101100100001000100100100100101 0001
 Stage 10 data: 1111010011101100001000101000000110100000001100110000010000110 10011
 Stage 11 data: 0011011010101111111000000110000000000000001101000000101000011001111
 Stage 12 data: 0011011010101111111000000110000000000000001101000000101000011001111

 Output data:   0000011011011111010100001100000000011000010100000011100000011001111

vcd file closed
Press any key to continue_
```

Figure 4.12: DES Key PC1 Permutation

The stage control bits for Key PC1 Permutation in the permutation module are;

Key_PC1_Control[0] = 0x8f8f8f8f;

Key_PC1_Control[1] = 0x9b9bb3b3;

Key_PC1_Control[2] = 0xadadd5d5;

Key_PC1_Control[3] = 0x7777ffff;

Key_PC1_Control[4] = 0xffffffff;

Key_PC1_Control[5] = 0xffffffff;

Key_PC1_Control[6] = 0x3c3c0f0f;

Key_PC1_Control[7] = 0x21de00ff;

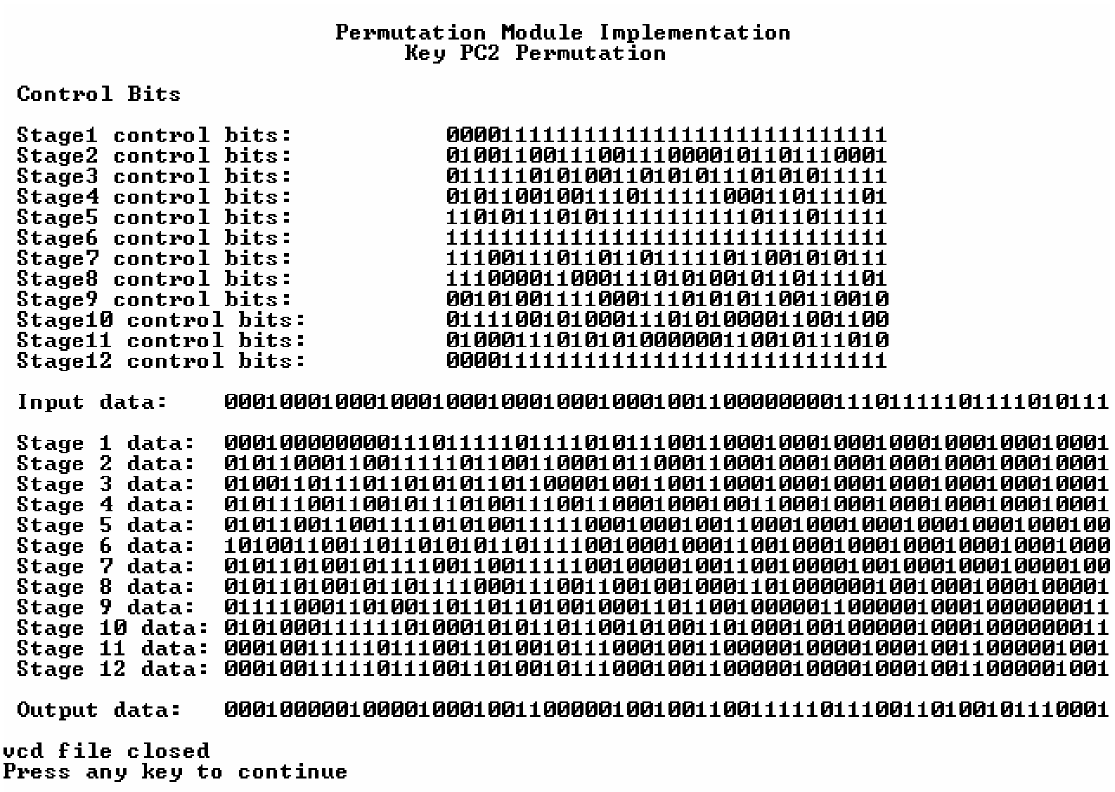
Key_PC1_Control[8] = 0x0000ffff;

Key_PC1_Control[9] = 0xa5a5aaaa;

Key_PC1_Control[10] = 0xc3c3cc3c;

Key_PC1_Control[11] = 0xf0f0f0f0;

The permute command should be 0x07 for the Key PC1 Permutation operation.



```
                      Permutation Module Implementation
                           Key PC2 Permutation
 Control Bits

 Stage1 control bits:          00001111111111111111111111111111
 Stage2 control bits:          01001100111001110000101101110001
 Stage3 control bits:          01111101010011010101110101011111
 Stage4 control bits:          01011001001110111111000110111101
 Stage5 control bits:          11010111010111111111110111011111
 Stage6 control bits:          11111111111111111111111111111111
 Stage7 control bits:          11100111011011011111011001010111
 Stage8 control bits:          11100000110001110101001011011101
 Stage9 control bits:          00101001111000111010101100110010
 Stage10 control bits:         01111001010001110101000011001100
 Stage11 control bits:         01000111010101000000110010111010
 Stage12 control bits:         00001111111111111111111111111111

 Input data:    0001000100010001000100010001000100110000000011101111101111010111

 Stage 1 data:  0001000000001110111110111101011100110001000100010001000100010001
 Stage 2 data:  0101100011001111101100110001011000110001000100010001000100010001
 Stage 3 data:  0100110111011010101101100001001100110001000100010001000100010001
 Stage 4 data:  0101110011001011101001110011000100010011000100010001000100010001
 Stage 5 data:  0101100110011110101001111100010001001100010001000100010001000100
 Stage 6 data:  1010011001101101010110111100100010001100100010001000100010001000
 Stage 7 data:  0101101001011110011001111100100001001100100001001000100010000100
 Stage 8 data:  0101101001011011110001110011001001000110100000010010001000100001
 Stage 9 data:  0111100011010011011011010010001101100100000110000010001000000011
 Stage 10 data: 0101000111111010001010110110010100110100010010000010001000000011
 Stage 11 data: 0001001111101110011010010111000100110000010000100010011000001001
 Stage 12 data: 0001001111101110011010010111000100110000010000100010011000001001

 Output data:   0001000000100001000100110000010010011001111101110011010010111000 1

vcd file closed
Press any key to continue
```

Figure 4.13: DES Key PC2 Permutation

The stage control bits for Key PC2 Permutation in the permutation module are;

Key_PC2_Control[0] = 0x0fffffff;

Key_PC2_Control[1] = 0x4ce70b71;

Key_PC2_Control[2] = 0x7d4d5d5f;

Key_PC2_Control[3] = 0x593bf1bd;

```
Key_PC2_Control[4]  = 0xd75ffddf;
Key_PC2_Control[5]  = 0xffffffff;
Key_PC2_Control[6]  = 0xe76df657;
Key_PC2_Control[7]  = 0xe18ea5bd;
Key_PC2_Control[8]  = 0x29e3ab32;
Key_PC2_Control[9]  = 0x794750cc;
Key_PC2_Control[10] = 0x47540cba;
Key_PC2_Control[11] = 0x0fffffff;
```

The permute command should be 0x09 for the Key PC2 Permutation operation.

4.8 Instruction Set Architecture

The Instruction Set Architecture is implemented to execute the basic parts of the crypto algorithms easily. Firstly, DES and AES algorithms are analyzed carefully to obtain the common properties for both of algorithms and some instructions are assigned to perform these common operations in DES and AES. The SBox, round key addition, round key store operations are some examples for the common blocks in DES and AES algorithms. After that the basic blocks of the algorithms are studied, and the instructions in the ISA are implemented according to these basic parts of algorithms. Therefore each instruction in the ISA performs one simple operation in the crypto algorithms. The purpose of this implementation is making the hardware design simpler.

The TDES algorithm is also performed with this ISA. Because of the TDES algorithm is an extension of DES algorithm, the instructions used in TDES are the same instructions, which are used in DES algorithm. But to execute the TDES algorithm correctly, there is only an additional control operation implemented in the Control Unit module.

XTME

Opcode: 0x40 to 0x47

Operation: xtime operation on MixColumns operation of AES algorithm

Syntax: *xtme   register*

Description: This instruction is used for the xtime operations [1] [6] for AES algorithm. The multiplication with the “x” coefficient in the Galois field is called as “xtime” and this instruction calculates the xtime value of the data in register and result data is stored again in the initial register.

MIX

Opcode: 0x48 to 0x4f

Operation: mix operation on MixColumns of AES algorithm

Syntax: *mix registera , registerb*

Description: mix instruction is used for the exor operations after the xtime instruction [6]. The MixColumns transformation in AES algorithm is a combination of the xtime and mix instruction. The data is taken from registera and registerb and the result data is written to the registera back.

SBOX

Opcode: 0x30 to 0x37

Operation: SBox operation for both of AES and DES algorithms

Syntax: *SBox   register*

Description: This instruction is used for the SBox operations. The data in register is replaced with the relevant memory data and result data is stored in the initial register.

SHIFT

Opcode: 0x38

Operation: AES 128-bit ShiftRows operation

Syntax: *shft   128*

Description: shift 128 instruction is used for the AES ShiftRows operation. The ShiftRows operation is operated over the rows in original AES state matrix. In the implemented architecture the data is stored in registers as given in Figure 4.14. The row elements of the original matrix are placed in the same byte of sequential registers. If we assume the sequential registers as a matrix, the original state rows are the column elements of the registers. Therefore the ShiftRows operation is performed in a different way in the architecture. This instruction is 3 machine cycle instructions. In first cycle it sends two 32-bit data (reg1 and reg2 values) to the ALU. In the second cycle two more 32-bit data (reg3 and reg4 values) is sent, also it takes two 32-bit result data (oreg1 and oreg2 values) and stored them in the relevant registers (reg1 and reg2) back. In the last cycle last two 32-bit data (oreg3 and oreg4 values) is taken from ALU and stored in registers (reg3 and reg4). The result register matrix is given in Figure 4.15. The incoming data is placed into the 32-bit data registers as given below.

Col3	Col2	Col1	Col0	
Byte3	Byte2	Byte1	Byte0	Reg1
Byte7	Byte6	Byte5	Byte4	Reg2
Byte11	Byte10	Byte9	Byte8	Reg3
Byte15	Byte14	Byte13	Byte12	Reg4

Figure 4.14: Register State values before shift operation

And the expected output data should be placed as given below.

Byte15	Byte10	Byte5	Byte0	OReg1
Byte3	Byte14	Byte9	Byte4	OReg2
Byte7	Byte2	Byte13	Byte8	OReg3
Byte11	Byte6	Byte1	Byte12	OReg4

Figure 4.15: Register State values after shift operation

SHIFT

Opcode: 0x39
Operation: AES 192-bit ShiftRows operation
Syntax: *shft 192*

Description: shift 192 instruction is used for the AES ShiftRows operation. This instruction is 4-cycle instruction. This instruction's structure is same like shift 128 instruction. But this needs one more cycle due to its increased data size.

SHIFT

Opcode: 0x3a
Operation: AES 256-bit ShiftRows operation
Syntax: *shft 256*

Description: shift 256 instruction is used for the AES ShiftRows operation. This instruction is 5-cycle instruction. Also this instruction is same like shift 128 and shift 192 instruction.

EXOR

Opcode: 0x24 to 0x2c

Operation: Bitwise EXOR operation

Syntax: *exor registera, registerb*

Description: EXOR instruction does a bitwise "EXCLUSIVE OR" operation between registera and register*b*, leaving the resulting value in registera. The value of registerb is not changed.

MOV1

Opcode: 0x50

Operation: Store accumulator group1 values

Syntax: *mov1*

Description: This instruction is used to store the accumulator registers group1 values in a different group of registers. This is necessary because of the accumulator values were used in some inner stage operations to calculate intermediate results and then another calculations with these intermediate results.

MOV2

Opcode: 0x51

Operation: Store accumulator group2 values

Syntax: *mov2*

Description: This instruction is used to store the accumulator registers group2 values in a different group of registers.

MOV

Opcode: 0x52 to 0x54
Operation: Move a value to a constant
Syntax: *mov constant , #value*

Description: These instructions are necessary for predetermining some internal values in intermediate operations of the algorithms.

MOV

Opcode: 0x55 to 0x59
Operation: Copy a register value to another register
Syntax: *mov registera , registerb*

Description: This instruction copies the registerb value to the registera.

EXK0

Opcode: 0x60 to 0x6f
Operation: Key exor operation
Syntax: *exk0 #number*

Description: The round data and key data EXOR operation is provided with this instruction. This instruction is continued three clock cycles. In the first clock the first half of the round data and its corresponding key data is forwarded to the ALU for the EXOR operation. In the second clock the output of the first operation is written back to the first accumulator register reg0 and also the second half of the round data and key data is processed. And in the third clock the result data is stored in the reg1.

EXK1

Opcode: 0x70 to 0x7f

Operation: Key exor operation

Syntax: *exk1 #number*

Description: Same as exk0 structure. The necessity of this instruction is due to the increased data and key length. The EXK0instruction has only a limited capacity for EXOR operation.

EXK2

Opcode: 0x80 to 0x8f

Operation: Key exor operation

Syntax: *exk2 #number*

Description: Same as EXK0

EXK3

Opcode: 0x90 to 0x9f

Operation: Key exor operation

Syntax: *exk3 #number*

Description: Same as EXK0

MVK0

Opcode: 0xd0 to 0xdf

Operation: Store AddRoundKey values

Syntax: *mvk0 #number*

Description: This instruction is used to store the AddRoundKey values in the predefined registers for further AddRoundKey operations. There is a specific portion in the register module, dedicated to the AddRoundKey data.

MVK1

Opcode: 0xe0 to 0xef
Operation: Store AddRoundKey values
Syntax: *mvk1 #number*

Description: Same as MVK0

RCON

Opcode: 0xf0 to 0xff
Operation: Load Rcon values to reg0
Syntax: *rcon #number*

Description: The RCON values are predefined constants which are used in key expansion of AES. Therefore this instruction is AES specific and used for load the Rcon values to accumulator register reg0.

PC1P

Opcode: 0xb0
Operation: DES Permute Key PC1
Syntax: *pc1p registera , registerb*

Description: PC1P instruction is used for Key PC1 permutation in the Key Expansion of the DES algorithm.

*In all the permutation operations the registera as left half data and registerb as right half data and writes the resulting value into that registers back.

PC2P

Opcode: 0xb1
Operation: DES Permute Key PC2
Syntax: *pc2p registera , registerb*

Description: PC2P instruction is used in the PC2 permutation of DES Key Expansion.

INIP

Opcode: 0xb2
Operation: DES Initial Permutation
Syntax: *inip registera, registerb*

Description: INIP is used for the Initial Permutation of DES. This is the first operation in the DES algorithm.

ETBP

Opcode: 0xb3
Operation: DES E Table Permutation
Syntax: *etbp registera, registerb*

Description: This permutation is used for bit permutations in the DES cipher operation. In this permutation bit mapping is used. The number of the incoming data is increased by using some bits more than once.

CIPP

Opcode: 0xb4

Operation: DES Cipher Permutation

Syntax: *cipp registera , registerb*

Description: Cipher permutation is used at the end of the cipher operation.

INVP

Opcode: 0xb5

Operation: DES Inverse Initial Permutation

Syntax: *invp registera , registerb*

Description: INVP is used for the Inverse Initial Permutation of DES. This is the last operation in the DES algorithm.

DIFP

Opcode: 0xb6

Operation: Any 64 bits Permutation

Syntax: *difp registera , registerb*

Description: DIFP instruction is used to permute any 64-bit data. The permutation block is capable of permute 64-bit data according to the control bits. The internal control bits are only for the DES permutation operations. This instruction use the control bits, which are loaded to the internal control registers with the LDPM instruction

RORB

Opcode: 0xa0

Operation: Rotate byte to right

Syntax: *rorb register*

Description: RORB instruction is used for rotating the register data to the right by 8-bit. The rightmost 8 bit is loaded to the leftmost 8-bit part of the result data.

RORD

Opcode: 0xa1

Operation: Rotate bit to right in DES

Syntax: *rord registera , registerb*

Description: RORD instruction is used for the DES key expansion and it is specific to the DES algorithm.

The round key data are rotated to left according to different offset values for each round. Also the rotated data is divided into two 28-bit units and each unit is rotated independently. There is not a unique process to divide the data, because it is already separated by storing the data in 32-bit registers.

The leftmost 4 bit in the registers is not important for this operation. Because of in each register there is 28-bit data. These data are called C-data and D-data in the DES algorithm specification. The result data is stored back to the same registers.

BMAP

Opcode: 0xa2

Operation: DES Bit Mapping

Syntax: *bmap registera , registerb*

Description: This instruction is used for the bit repetitions on the permutation operations. The data in registera is mapped according to the control data in registerb and the result data is stored in registerb. The data register registera is not affected. In the DES algorithm the E Table Selection operation is include some bit repetitions. The input of the E Table Selection permutation is 32-bit data and the output is 48-bit. The bit repetitions cannot be solved with the permute instruction.

Therefore a specific instruction for this purpose is implemented. The data and control data are 32-bit. This instruction takes two operands. One of them is the data, which will be permuted and the other one is the control data for determining the bit repetitions. The output of this instruction is 32-bit repetition data. Therefore at one time only 32-bit data mapping operation can be made.

The input data bits are mapped to the output data according to the control data bits. If the control data bit is “1” the respective data bit is mapped to the output data. The mapping instruction figure is given below.

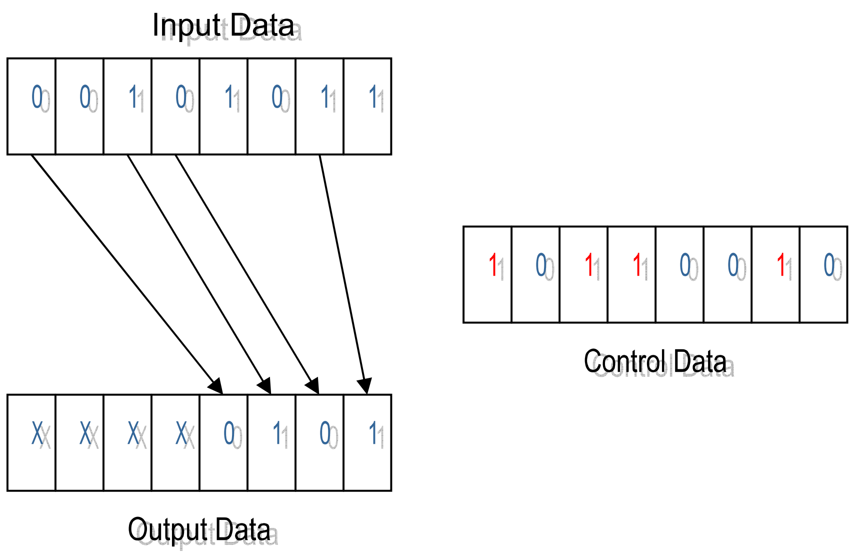


Figure 4.16: BMAP function

SBAD

Opcode: 0xa3

Operation: DES SBox address calculation

Syntax: *sbad registera , registerb*

Description: des SBox address instruction is used for the DES SBox operation. The address information is computed for memory access operation. The operands are taken from registers and the result data is written back to the registers. In the DES algorithm the SBox operation is performed in a different way compared to AES algorithm. The result value position in the SBox table is calculated according to some rules of the input data. And this instruction calculates the memory address data for the SBox operation.

SWAP

Opcode: 0xa5
Operation: Swap register values
Syntax: *swap registera , registerb*

Description: SWAP instruction is used to change the registera and registerb values.

DREX

Opcode: 0xa6
Operation: Des round data exor
Syntax: *drex registera , registerb*

Description: This instruction is used in the end of the each round of the DES. In each end of the round the half of data swapped with other half. In this swap operation on half of the data is EXORed with the output of the cipher operation. To avoid a sequential of exor and move operations this single instruction exor the data and store it to correct place.

COPY

Opcode: 0x11 , 0x15
Operation: copy data from IO Module or to IO Module
Syntax: *copy ibuffer or copy obuffer*

Description: This instruction is used to copy the input buffer registers of the IO Module to internal registers of the Control Unit and the state register values of Control Unit to the output buffers of IO Module.

READ

Opcode: 0x18

Operation: Read data from external interface

Syntax: *read iport*

Description: This instruction is used to read data from external interface.

WRTE

Opcode: 0x19

Operation: Write data to external interface

Syntax: *wrte oport*

Description: This instruction is used to write the output data to external interface.

INSX

Opcode: 0xc0 to 0xc7

Operation: inverse SBox operation for AES algorithm

Syntax: *insx   register*

Description: This instruction is used for the inverse SBox operations. The data in register is replaced with the relevant memory data and result data is stored in the initial register for Inverse SBox operation.

INMX

Opcode: 0xc8 to 0xcf
Operation: inverse MixColumns operation for AES algorithm
Syntax: *inmx register*

Description: This instruction is used for the inverse MixColumns operations. The Inverse MixColumns operation in the AES algorithm is more complex than the normal MixColumns operation. There are the x^3, x^2 parameters in the Inverse operation. These operations can be performed as cascaded the xtime block several times. With this instruction the xtime block is used several times and Inverse MixColumns operation is performed.

ISFT

Opcode: 0x3b
Operation: AES 128-bit Inverse ShiftRows operation
Syntax: *isft 128*

Description: This instruction is used in Inverse ShiftRows operation of AES algorithm. The principle of this instruction is same with normal shift operation. Only the structure is modified according to shift direction.

ISFT

Opcode: 0x3c
Operation: AES 192-bit Inverse ShiftRows operation
Syntax: *isft 192*

Description: same as ISFT 128

ISFT

Opcode: 0x3d

Operation: AES 256-bit Inverse ShiftRows operation

Syntax: *isft 256*

Description: same as ISFT 128

LDSB

Opcode: 0x02

Operation: Load SBox memory elements

Syntax: *ldsb*

Description: The internal data of the SBox memory can be reloaded to perform other applications. The first 256 memory element is allowed to reload and reuse in different applications.

LDPM

Opcode: 0x03

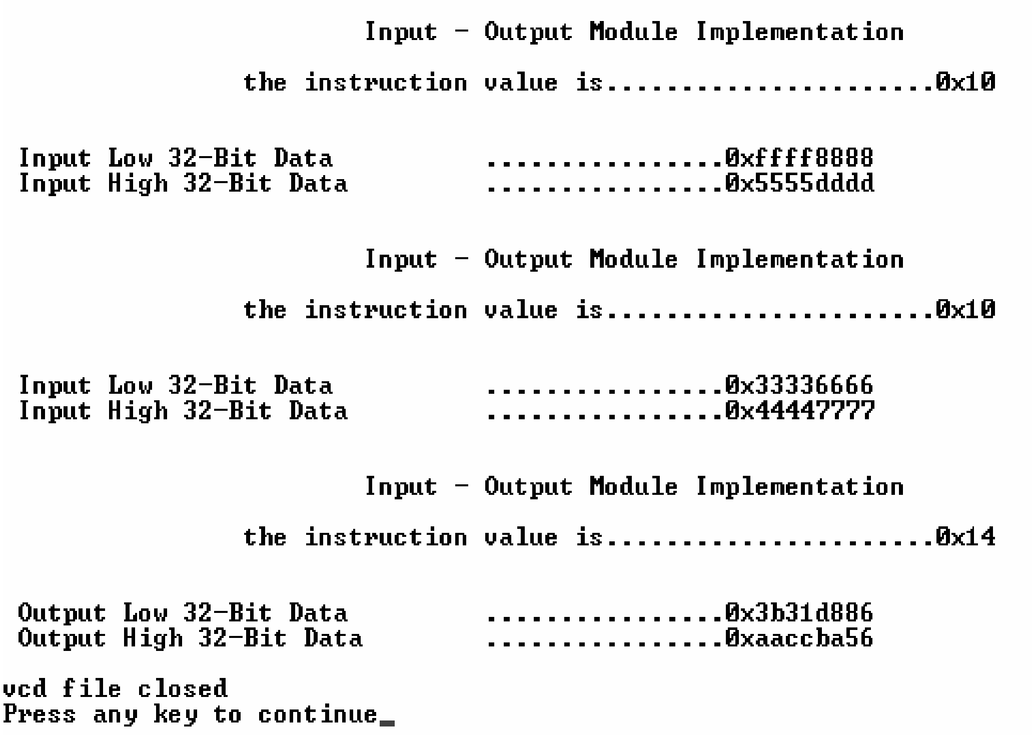
Operation: Load Permutation Control Bits

Syntax: *ldpm*

Description: There are a predefined control bits in the permutation bit, which are dedicated for the any other 64-bit permutation independently of DES permutation control bits. These control bits can be used only with the DIFP instruction.

4.9 Simulations and Implementation Results

The simulations are performed to verify the implemented design. Firstly the encryption operations of DES, TDES and AES algorithms are simulated. In the console output figures the first input data are input key data to the simulator. Then a random plaintext data is applied to the simulator for encryption. After that for decryption operation the ciphertext output of the encryption part is applied as input to simulator. Then a comparison between the input of encryption and the output of decryption is made.



```
                        Input - Output Module Implementation

                the instruction value is......................0x10

 Input Low 32-Bit Data          ................0xffff8888
 Input High 32-Bit Data         ................0x5555dddd

                        Input - Output Module Implementation

                the instruction value is......................0x10

 Input Low 32-Bit Data          ................0x33336666
 Input High 32-Bit Data         ................0x44447777

                        Input - Output Module Implementation

                the instruction value is......................0x14

 Output Low 32-Bit Data         ................0x3b31d886
 Output High 32-Bit Data        ................0xaaccba56

vcd file closed
Press any key to continue_
```

Figure 4.17: DES Encryption Console Output

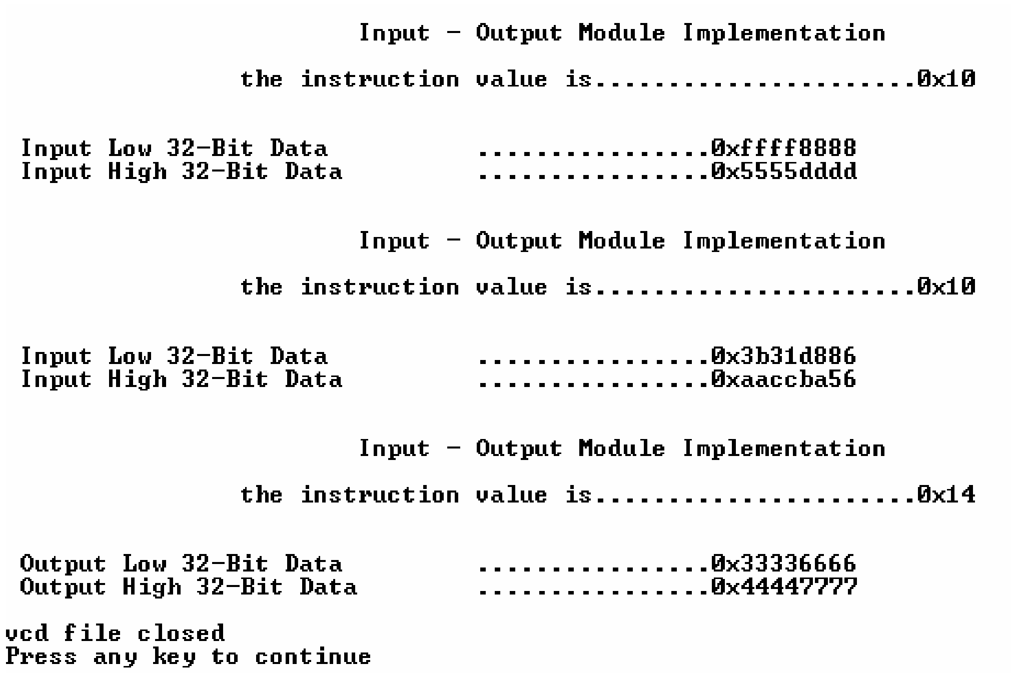

```
                     Input - Output Module Implementation

             the instruction value is......................0x10


 Input Low 32-Bit Data          ................0xffff8888
 Input High 32-Bit Data         ................0x5555dddd

                     Input - Output Module Implementation

             the instruction value is......................0x10


 Input Low 32-Bit Data          ................0x3b31d886
 Input High 32-Bit Data         ................0xaaccba56

                     Input - Output Module Implementation

             the instruction value is......................0x14


 Output Low 32-Bit Data         ................0x33336666
 Output High 32-Bit Data        ................0x44447777

vcd file closed
Press any key to continue
```

Figure 4.18: DES Decryption Console Output

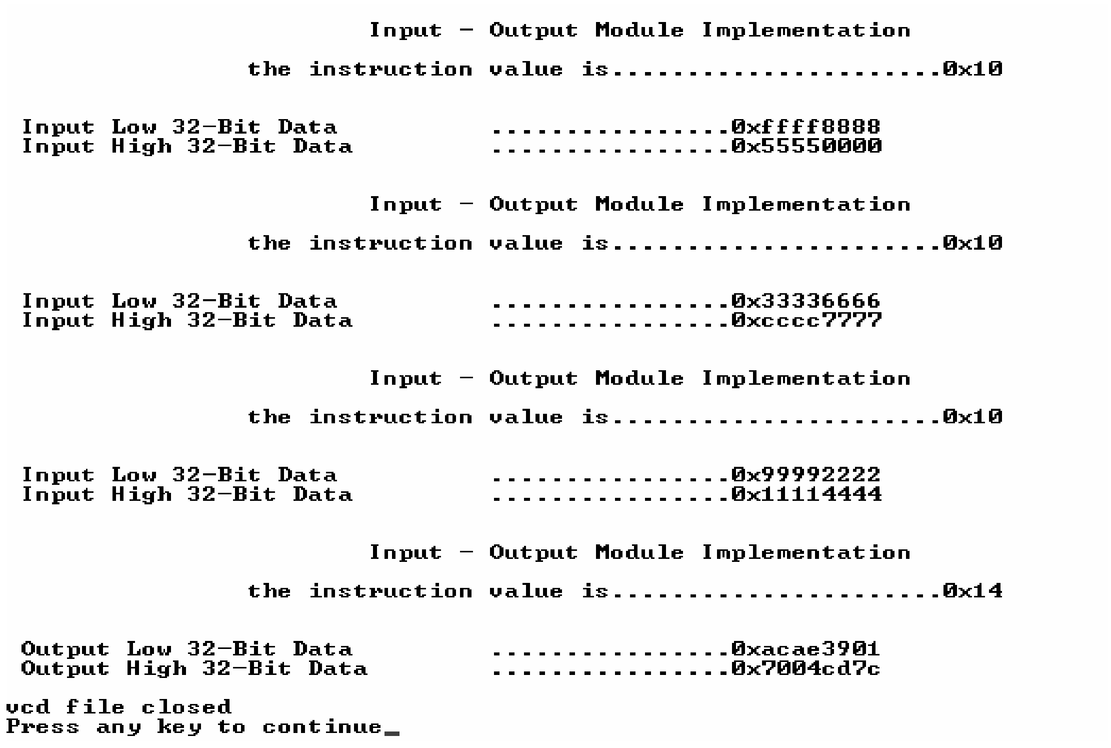
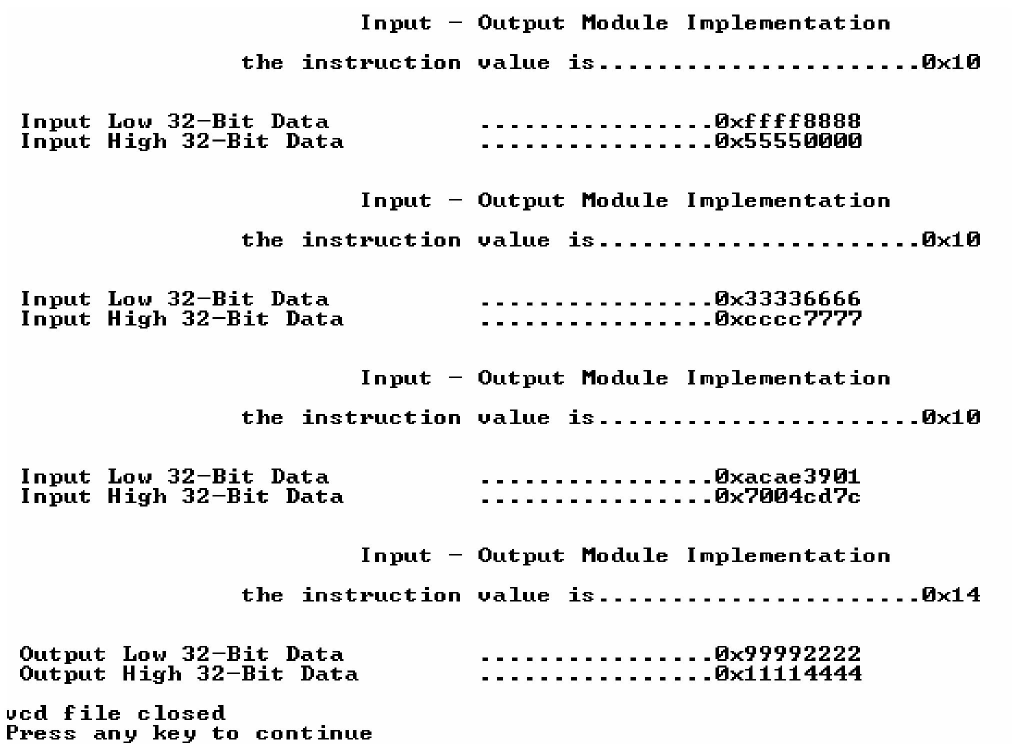
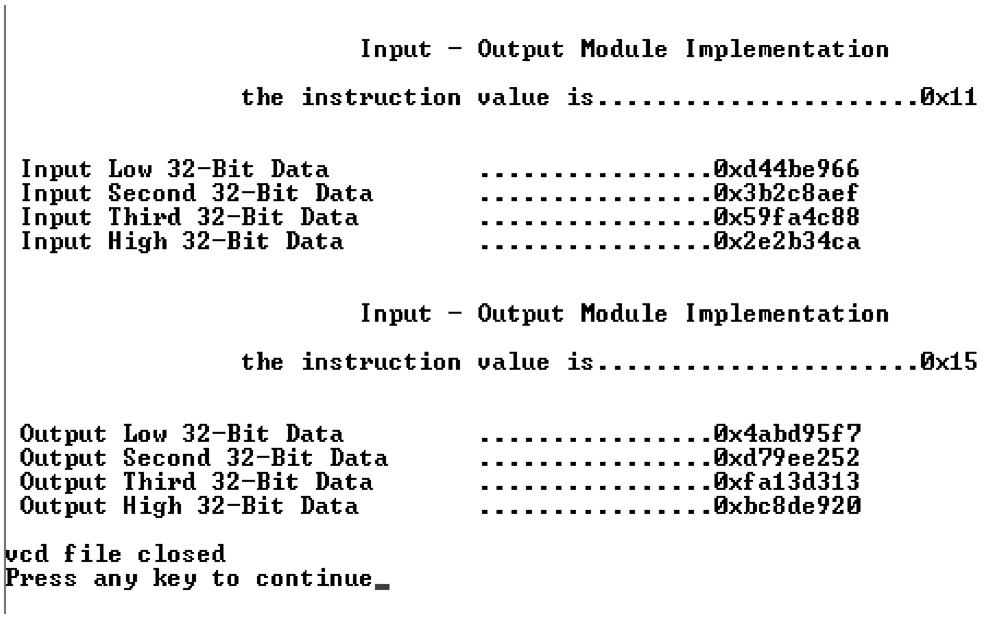

```
                        Input - Output Module Implementation

               the instruction value is......................0x10

 Input Low 32-Bit Data          ................0xffff8888
 Input High 32-Bit Data         ................0x55550000

                        Input - Output Module Implementation

               the instruction value is......................0x10

 Input Low 32-Bit Data          ................0x33336666
 Input High 32-Bit Data         ................0xcccc7777

                        Input - Output Module Implementation

               the instruction value is......................0x10

 Input Low 32-Bit Data          ................0x99992222
 Input High 32-Bit Data         ................0x11114444

                        Input - Output Module Implementation

               the instruction value is......................0x14

 Output Low 32-Bit Data         ................0xacae3901
 Output High 32-Bit Data        ................0x7004cd7c

vcd file closed
Press any key to continue_
```

Figure 4.19: TDES Encryption Console Output



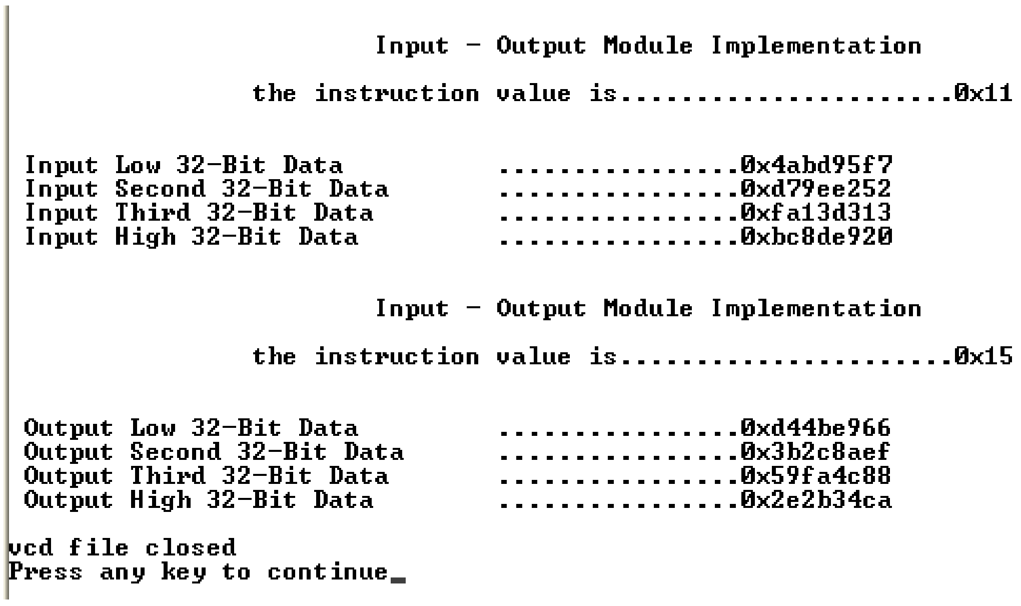
```
                    Input - Output Module Implementation

            the instruction value is.......................0x10

 Input Low 32-Bit Data          .................0xffff8888
 Input High 32-Bit Data         .................0x55550000

                    Input - Output Module Implementation

            the instruction value is.......................0x10

 Input Low 32-Bit Data          .................0x33336666
 Input High 32-Bit Data         .................0xcccc7777

                    Input - Output Module Implementation

            the instruction value is.......................0x10

 Input Low 32-Bit Data          .................0xacae3901
 Input High 32-Bit Data         .................0x7004cd7c

                    Input - Output Module Implementation

            the instruction value is.......................0x14

 Output Low 32-Bit Data         .................0x99992222
 Output High 32-Bit Data        .................0x11114444

vcd file closed
Press any key to continue
```

Figure 4.20: TDES Decryption Console Output



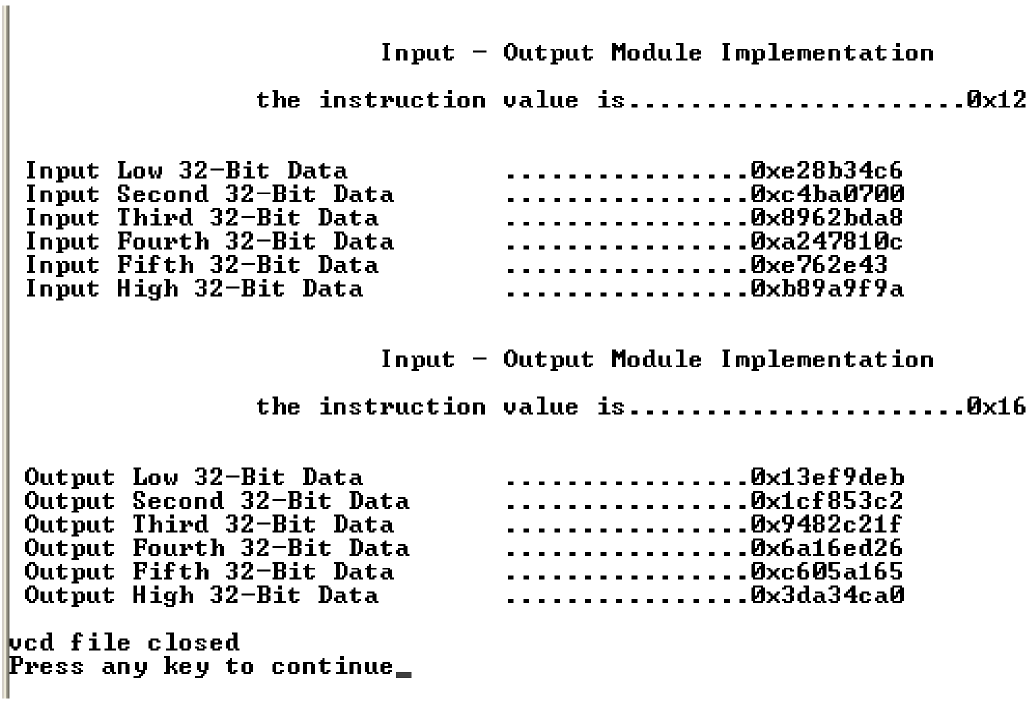
```
                    Input - Output Module Implementation

            the instruction value is.......................0x11


 Input Low 32-Bit Data          .................0xd44be966
 Input Second 32-Bit Data       .................0x3b2c8aef
 Input Third 32-Bit Data        .................0x59fa4c88
 Input High 32-Bit Data         .................0x2e2b34ca

                    Input - Output Module Implementation

            the instruction value is.......................0x15


 Output Low 32-Bit Data         .................0x4abd95f7
 Output Second 32-Bit Data      .................0xd79ee252
 Output Third 32-Bit Data       .................0xfa13d313
 Output High 32-Bit Data        .................0xbc8de920

vcd file closed
Press any key to continue_
```

Figure 4.21: AES-128 Encryption Console Output



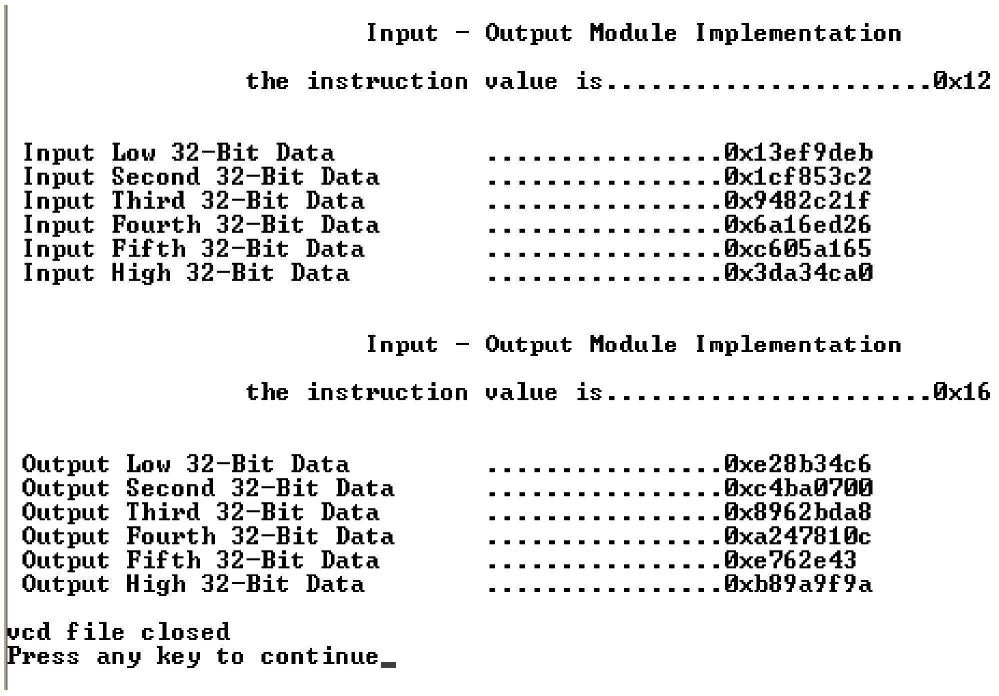
```
                    Input - Output Module Implementation

            the instruction value is......................0x11


 Input Low 32-Bit Data          .................0x4abd95f7
 Input Second 32-Bit Data       .................0xd79ee252
 Input Third 32-Bit Data        .................0xfa13d313
 Input High 32-Bit Data         .................0xbc8de920

                    Input - Output Module Implementation

            the instruction value is......................0x15


 Output Low 32-Bit Data         .................0xd44be966
 Output Second 32-Bit Data      .................0x3b2c8aef
 Output Third 32-Bit Data       .................0x59fa4c88
 Output High 32-Bit Data        .................0x2e2b34ca

vcd file closed
Press any key to continue_
```

Figure 4.22: AES-128 Decryption Console Output



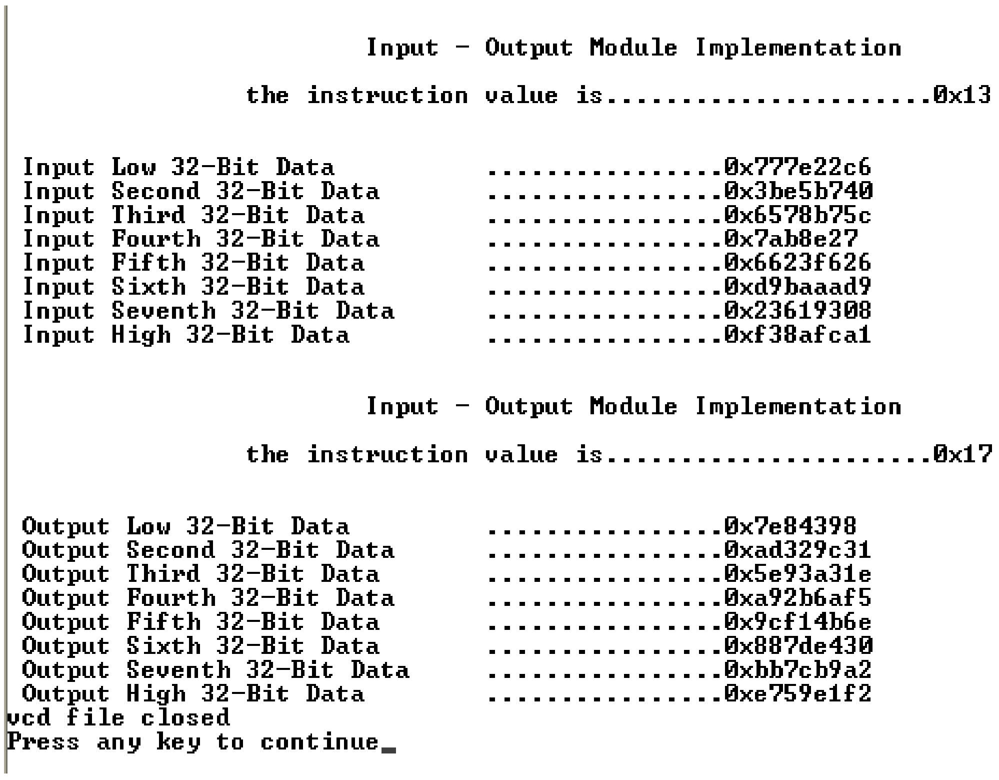
```
                    Input - Output Module Implementation

            the instruction value is......................0x12


 Input Low 32-Bit Data          .................0xe28b34c6
 Input Second 32-Bit Data       .................0xc4ba0700
 Input Third 32-Bit Data        .................0x8962bda8
 Input Fourth 32-Bit Data       .................0xa247810c
 Input Fifth 32-Bit Data        .................0xe762e43
 Input High 32-Bit Data         .................0xb89a9f9a

                    Input - Output Module Implementation

            the instruction value is......................0x16


 Output Low 32-Bit Data         .................0x13ef9deb
 Output Second 32-Bit Data      .................0x1cf853c2
 Output Third 32-Bit Data       .................0x9482c21f
 Output Fourth 32-Bit Data      .................0x6a16ed26
 Output Fifth 32-Bit Data       .................0xc605a165
 Output High 32-Bit Data        .................0x3da34ca0

vcd file closed
Press any key to continue_
```

Figure 4.23: AES-192 Encryption Console Output



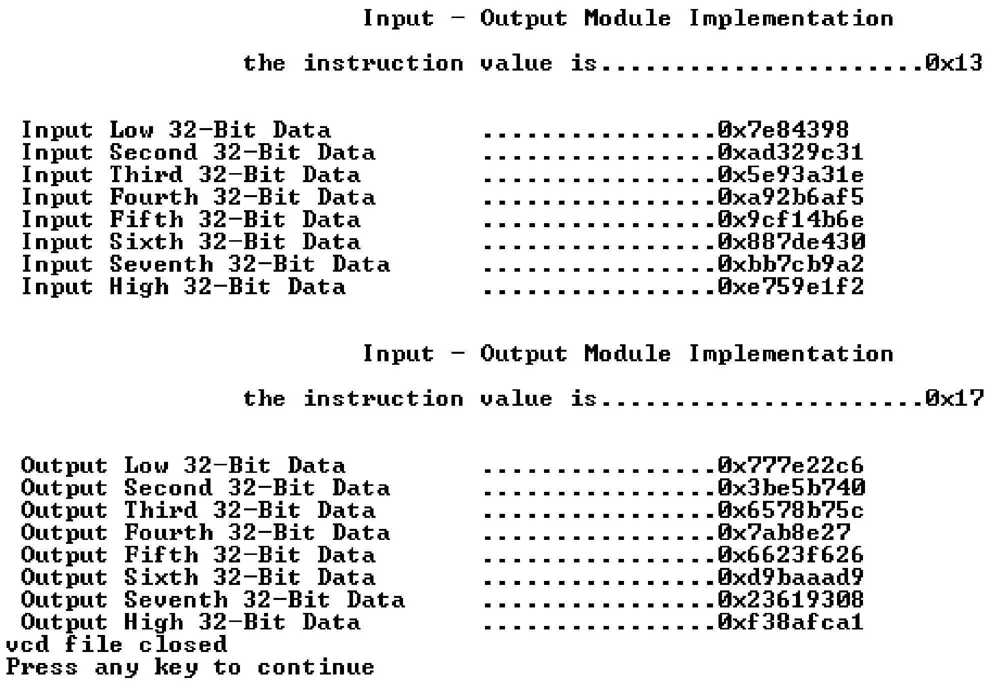
```
                        Input - Output Module Implementation

                the instruction value is......................0x12


 Input Low 32-Bit Data          .................0x13ef9deb
 Input Second 32-Bit Data       .................0x1cf853c2
 Input Third 32-Bit Data        .................0x9482c21f
 Input Fourth 32-Bit Data       .................0x6a16ed26
 Input Fifth 32-Bit Data        .................0xc605a165
 Input High 32-Bit Data         .................0x3da34ca0

                        Input - Output Module Implementation

                the instruction value is......................0x16


 Output Low 32-Bit Data         .................0xe28b34c6
 Output Second 32-Bit Data      .................0xc4ba0700
 Output Third 32-Bit Data       .................0x8962bda8
 Output Fourth 32-Bit Data      .................0xa247810c
 Output Fifth 32-Bit Data       .................0xe762e43
 Output High 32-Bit Data        .................0xb89a9f9a

vcd file closed
Press any key to continue_
```

Figure 4.24: AES-192 Decryption Console Output



```
                        Input - Output Module Implementation

                the instruction value is......................0x13


 Input Low 32-Bit Data          .................0x777e22c6
 Input Second 32-Bit Data       .................0x3be5b740
 Input Third 32-Bit Data        .................0x6578b75c
 Input Fourth 32-Bit Data       .................0x7ab8e27
 Input Fifth 32-Bit Data        .................0x6623f626
 Input Sixth 32-Bit Data        .................0xd9baaad9
 Input Seventh 32-Bit Data      .................0x23619308
 Input High 32-Bit Data         .................0xf38afca1

                        Input - Output Module Implementation

                the instruction value is......................0x17


 Output Low 32-Bit Data         .................0x7e84398
 Output Second 32-Bit Data      .................0xad329c31
 Output Third 32-Bit Data       .................0x5e93a31e
 Output Fourth 32-Bit Data      .................0xa92b6af5
 Output Fifth 32-Bit Data       .................0x9cf14b6e
 Output Sixth 32-Bit Data       .................0x887de430
 Output Seventh 32-Bit Data     .................0xbb7cb9a2
 Output High 32-Bit Data        .................0xe759e1f2
vcd file closed
Press any key to continue_
```

Figure 4.25: AES-256 Encryption Console Output



```
                    Input - Output Module Implementation

            the instruction value is......................0x13

 Input Low 32-Bit Data          .................0x7e84398
 Input Second 32-Bit Data       .................0xad329c31
 Input Third 32-Bit Data        .................0x5e93a31e
 Input Fourth 32-Bit Data       .................0xa92b6af5
 Input Fifth 32-Bit Data        .................0x9cf14b6e
 Input Sixth 32-Bit Data        .................0x887de430
 Input Seventh 32-Bit Data      .................0xbb7cb9a2
 Input High 32-Bit Data         .................0xe759e1f2

                    Input - Output Module Implementation

            the instruction value is......................0x17

 Output Low 32-Bit Data         .................0x777e22c6
 Output Second 32-Bit Data      .................0x3be5b740
 Output Third 32-Bit Data       .................0x6578b75c
 Output Fourth 32-Bit Data      .................0x7ab8e27
 Output Fifth 32-Bit Data       .................0x6623f626
 Output Sixth 32-Bit Data       .................0xd9baaad9
 Output Seventh 32-Bit Data     .................0x23619308
 Output High 32-Bit Data        .................0xf38afca1
vcd file closed
Press any key to continue
```

Figure 4.26: AES-256 Decryption Console Output

For the implementation results, main parts of ALU module and Permutation module in SystemC descriptions are compiled into hardware using the SystemCrafter tool. And then the outputs of the SystemCrafter tool is used in synthesis process together with Xilinx tool into Spartan3AXC3S200A device. The results of this process are given in Table 4.2. The SBox Table and Permutation module are also compiled into hardware using the SystemCrafter tool. But due to compiler limit problems of SystemCrafter tool, the basic parts of the SBox and Permutation module are compiled into hardware and synthesized with Xilinx tools. Then some assumptions are made to get an idea about the SBox and Permutation blocks areas. The results of these assumptions are given in Table 4.3.

Table 4.2: Slices values for some crypto specific blocks

	SWAP	**XTIME**	**MIX**	**SHIFT 256**
Logic Utilization				
Number of Slice Flip Flops	142	82	139	222
Number of 4 input LUTs	129	65	265	360
Logic Distribution				
Number of occupied slices	107	63	178	227
Only related logic	107	63	178	227
Unrelated logic	0	0	0	0
Total Number of 4 input LUTs				
Number of bonded IOBs	129	65	97	129
IOB Flip Flops	64	32	64	64
Total equivalent gate count for design	2425	1341	3361	4417
	SHIFT 192	**SHIFT 128**	**DESS ADD**	**EXOR**
Logic Utilization				
Number of Slice Flip Flops	211	198	28	112
Number of 4 input LUTs	334	275	19	65
Logic Distribution				
Number of occupied slices	215	197	18	77
Only related logic	215	197	18	77
Unrelated logic	0	0	0	0
Total Number of 4 input LUTs				
Number of bonded IOBs	129	129	39	97
IOB Flip Flops	64	64	6	64
Total equivalent gate count for design	4328	4013	389	1897
	DES ROR	**ROR BYTE**	**DESS DATA**	**XTIME 2**
Logic Utilization				
Number of Slice Flip Flops	123	83	58	138
Number of 4 input LUTs	109	73	49	257
Logic Distribution				
Number of occupied slices	94	60	39	182
Only related logic	94	60	39	182
Unrelated logic	0	0	0	0
Total Number of 4 input LUTs				
Number of bonded IOBs	119	65	49	65
IOB Flip Flops	54	32	16	32
Total equivalent gate count for design	2073	1361	889	3097

Table 4.3: Approximately Slices values for SBox and Permutation blocks

Crypto Specific Block	Number of occupied Slices
SBox	4960
Permutation	1672

Table 4.4 gives the machine cycle values of the implemented Crypto processor for the related algorithms, and Table 4.5 gives a machine cycles comparison for the performed crypto algorithms by the implemented crypto processor and other programmable crypto processors.

Table 4.4: Machine Cycles for performed Crypto Algorithms

Crypto Algorithm	Machine Cycle
128 AES	213
192 AES	397
256 AES	517
DES	196
TDES	596

Table 4.5 Comparison between Machine Cycles of Programmable Crypto Processors

	128-bit AES	DES	TDES	Expected Area	Structure
Cryptonite	70	35	105	3A	Complex
CryptoManiac	90	130	392	4A	Complex
Impl. Processor	213	196	596	A	Simple

Below figures are the simulation outputs of some instructions used in the crypto algorithms.

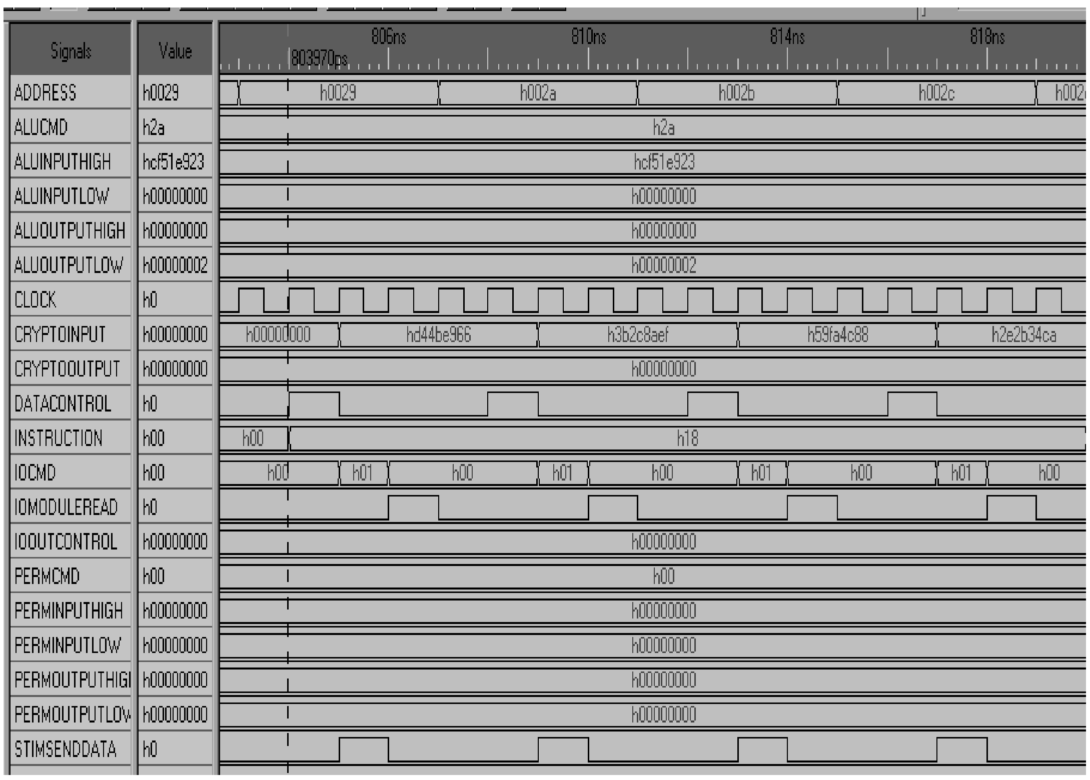


Figure 4.27: Instruction *read*

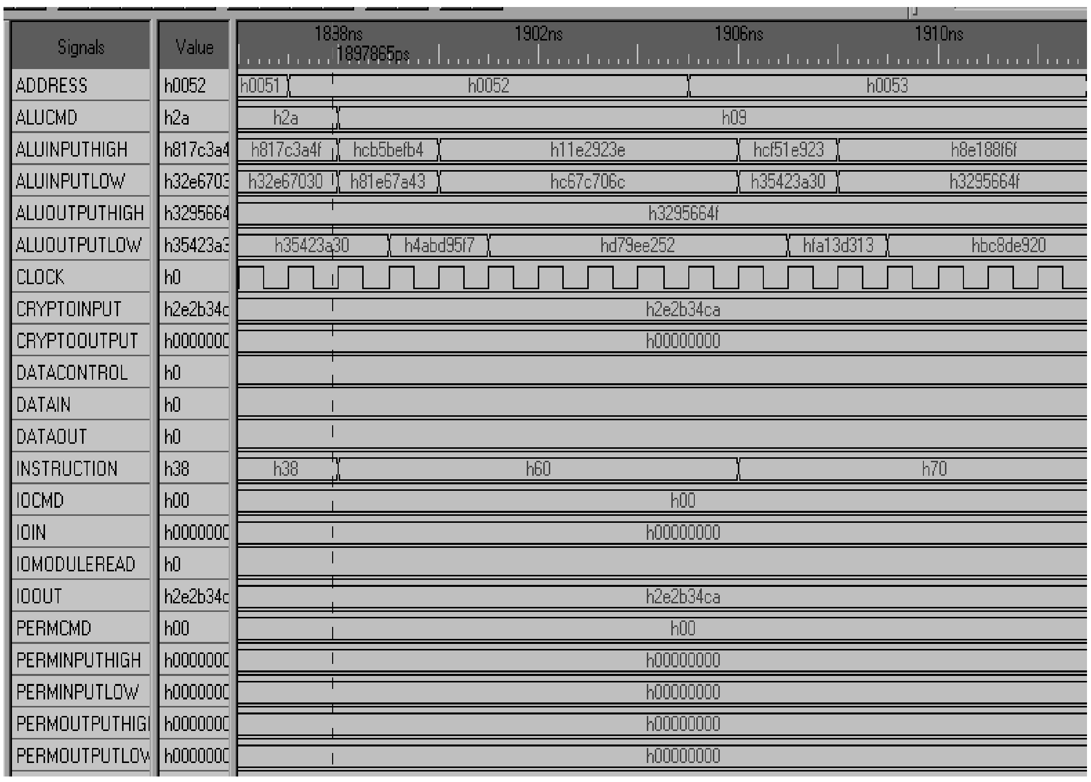


Figure 4.28: Instruction *exk0*

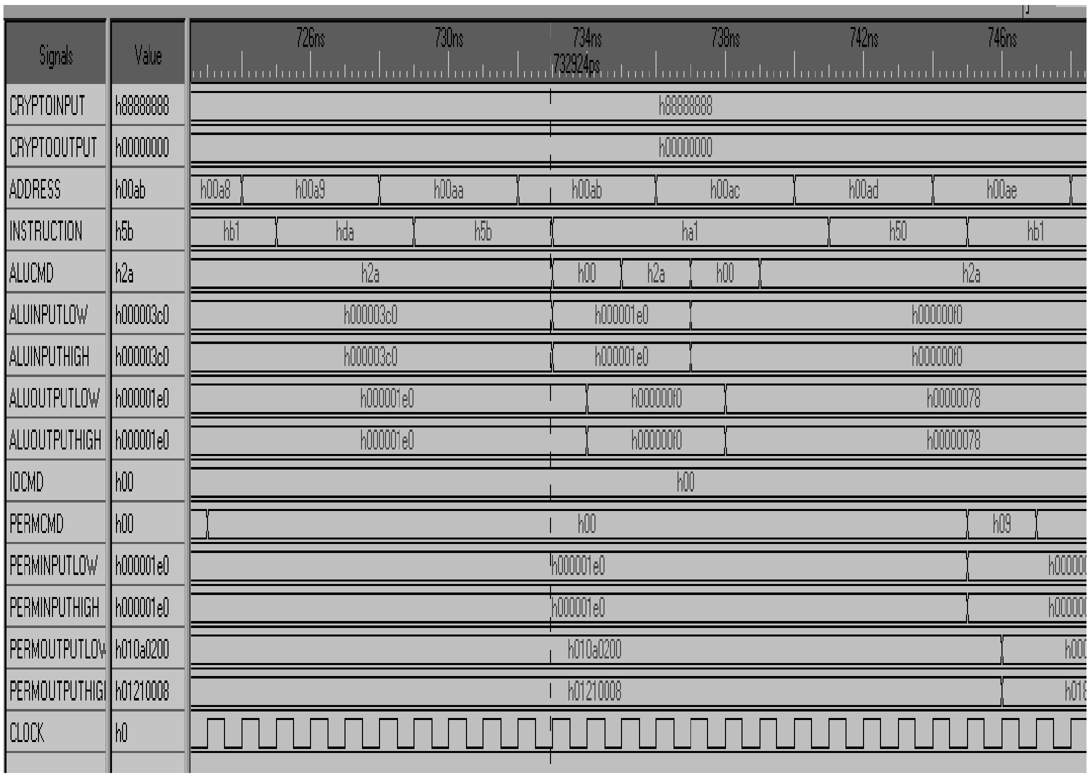


Figure 4.29: Instruction *rord*

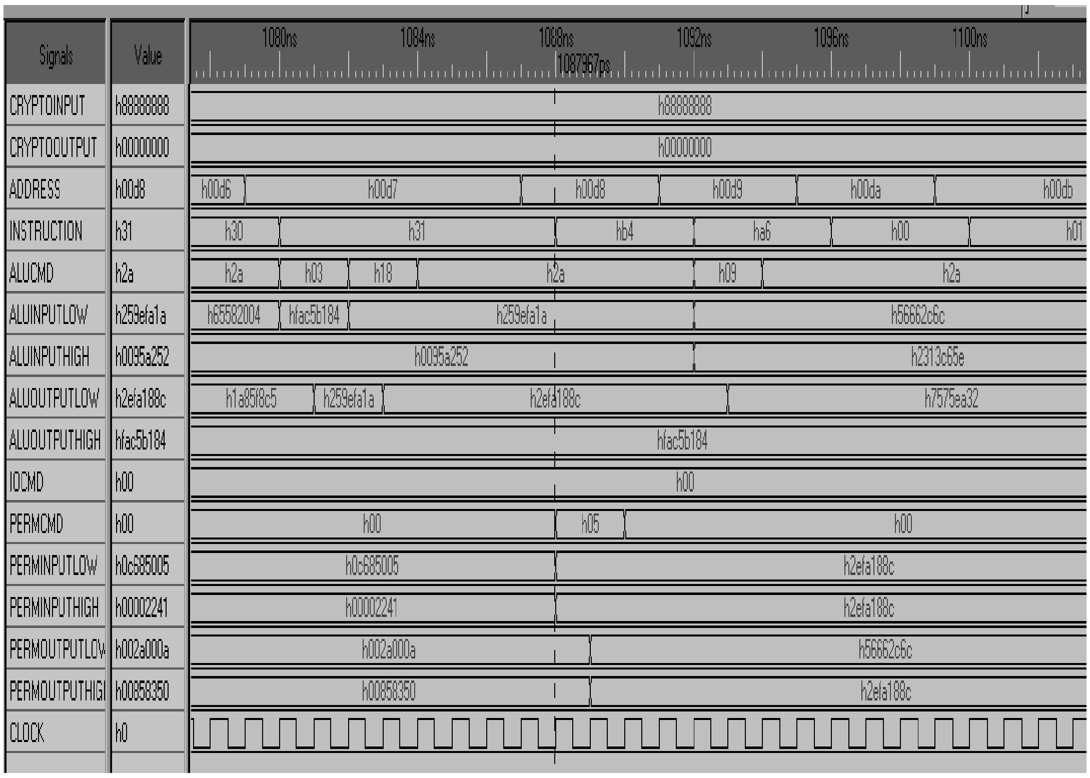
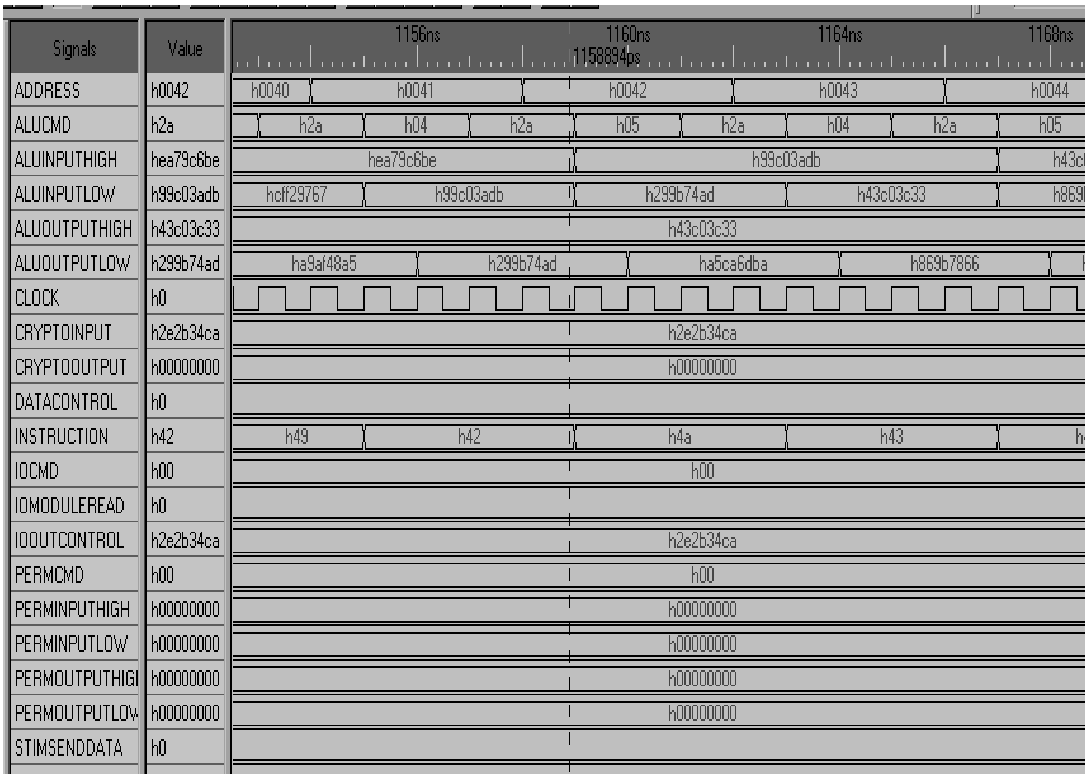
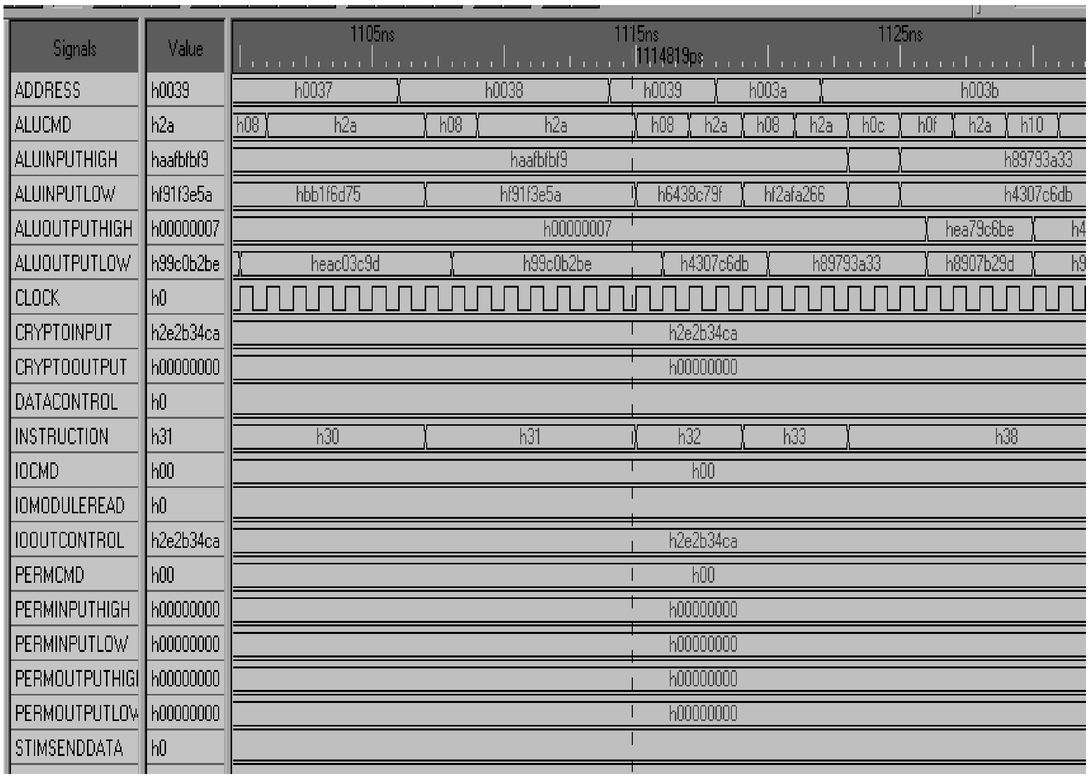


Figure 4.30: Instruction *cipp*



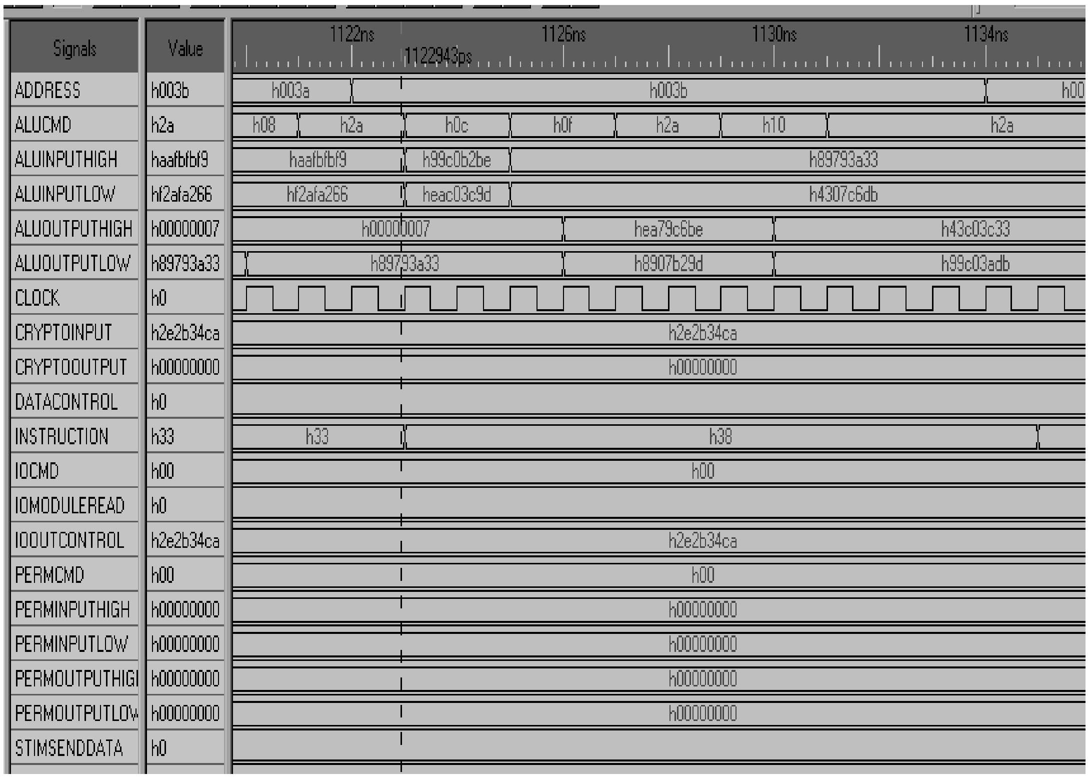
Signals	Value
ADDRESS	h0042
ALUCMD	h2a
ALUINPUTHIGH	hea79c6be
ALUINPUTLOW	h99c03adb
ALUOUTPUTHIGH	h43c03c33
ALUOUTPUTLOW	h299b74ad
CLOCK	h0
CRYPTOINPUT	h2e2b34ca
CRYPTOOUTPUT	h00000000
DATACONTROL	h0
INSTRUCTION	h42
IOCMD	h00
IOMODULEREAD	h0
IOOUTCONTROL	h2e2b34ca
PERMCMD	h00
PERMINPUTHIGH	h00000000
PERMINPUTLOW	h00000000
PERMOUTPUTHIGI	h00000000
PERMOUTPUTLOV	h00000000
STIMSENDDATA	h0

Figure 4.31: Instruction *mix*



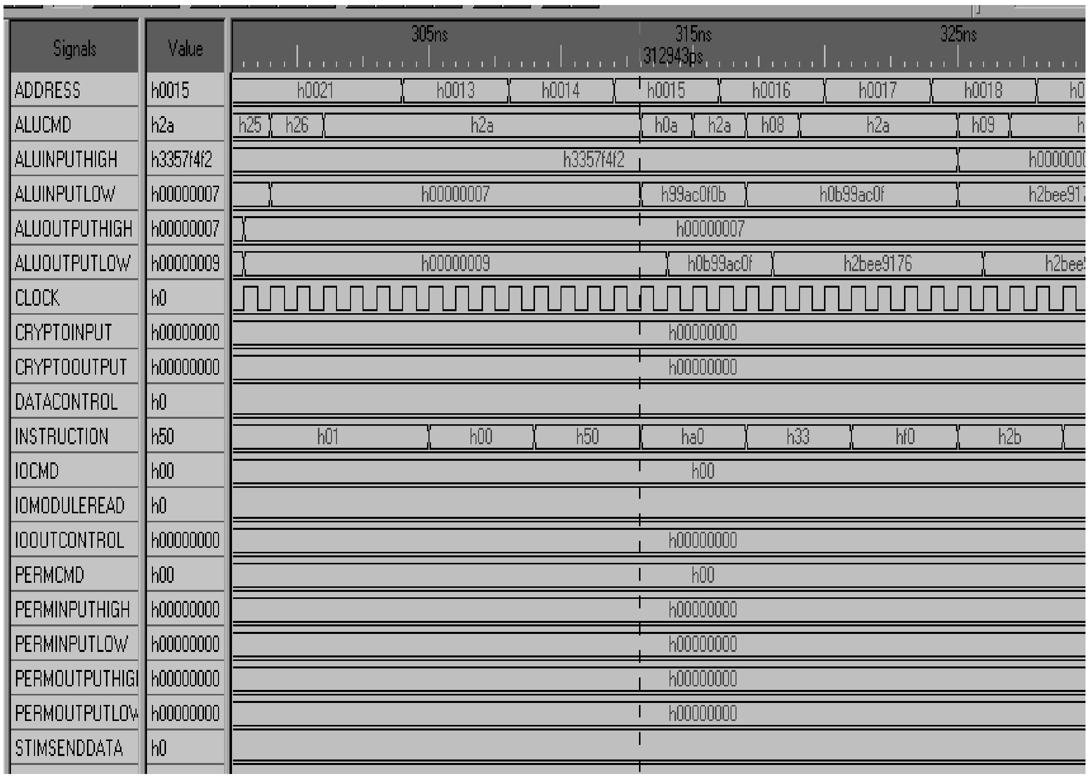
Signals	Value
ADDRESS	h0039
ALUCMD	h2a
ALUINPUTHIGH	haafbfbf9
ALUINPUTLOW	hf91f3e5a
ALUOUTPUTHIGH	h00000007
ALUOUTPUTLOW	h99c0b2be
CLOCK	h0
CRYPTOINPUT	h2e2b34ca
CRYPTOOUTPUT	h00000000
DATACONTROL	h0
INSTRUCTION	h31
IOCMD	h00
IOMODULEREAD	h0
IOOUTCONTROL	h2e2b34ca
PERMCMD	h00
PERMINPUTHIGH	h00000000
PERMINPUTLOW	h00000000
PERMOUTPUTHIGI	h00000000
PERMOUTPUTLOV	h00000000
STIMSENDDATA	h0

Figure 4.32: Instruction *sbox*



Signals	Value
ADDRESS	h003b
ALUCMD	h2a
ALUINPUTHIGH	haafbfbf9
ALUINPUTLOW	hf2afa266
ALUOUTPUTHIGH	h00000007
ALUOUTPUTLOW	h89793a33
CLOCK	h0
CRYPTOINPUT	h2e2b34ca
CRYPTOOUTPUT	h00000000
DATACONTROL	h0
INSTRUCTION	h33
IOCMD	h00
IOMODULEREAD	h0
IOOUTCONTROL	h2e2b34ca
PERMCMD	h00
PERMINPUTHIGH	h00000000
PERMINPUTLOW	h00000000
PERMOUTPUTHIGI	h00000000
PERMOUTPUTLOV	h00000000
STIMSENDDATA	h0

Figure 4.33: Instruction *shift 128*



Signals	Value
ADDRESS	h0015
ALUCMD	h2a
ALUINPUTHIGH	h3357f4f2
ALUINPUTLOW	h00000007
ALUOUTPUTHIGH	h00000007
ALUOUTPUTLOW	h00000009
CLOCK	h0
CRYPTOINPUT	h00000000
CRYPTOOUTPUT	h00000000
DATACONTROL	h0
INSTRUCTION	h50
IOCMD	h00
IOMODULEREAD	h0
IOOUTCONTROL	h00000000
PERMCMD	h00
PERMINPUTHIGH	h00000000
PERMINPUTLOW	h00000000
PERMOUTPUTHIGI	h00000000
PERMOUTPUTLOV	h00000000
STIMSENDDATA	h0

Figure 4.34: Instruction *rorb*

CHAPTER 5

CONCLUSION

In this thesis study, a programmable Crypto Processor is implemented for AES, DES and TDES algorithms, containing both encryption and decryption in the same design for all data and key lengths. A new Instruction Set Architecture is suggested and implemented to process all different modes easily.

The objective of this implementation is to combine the features of the AES and DES algorithms in single architecture and to utilize the reuse capability of the processor's instructions. Since the bit permutation operations are not so easy to be implemented with general ALU operations like "shift", "and", "or", "rotate", a special permutation module is added into the architecture to perform bit permutation operations. In several applications, the bit permutation operation is implemented in memory based structures or in hardware routing structure, which are dedicated to only single permutation.

Due to its architecture, the permutation module is used to do all of the bit permutation operations. All the DES permutation operations are performed in this permutation module, as well as other permutation operations. This permutation module is capable of doing any other 64-bit permutations. By loading the proper control bits, any 64-bit permutation can be performed in a single structure with the implemented Permutation Module.

The Data Substitution operation is performed using Look-up-tables. The Look-up-table is unified for AES and DES algorithms. In AES algorithm, the substitution operation is performed over bytes, but in DES algorithm the output of this operation is 4-bit data. Therefore, with a proper addressing scheme for DES substitution outputs, a single SBox memory is used for both of algorithms. The same instruction is used for data substitution operation within AES and DES algorithms.

Data stored in the SBox memory is easily modified to adapt this structure to new algorithms. In normal cases, the internal memory data is used for standard AES and DES applications. But if it is necessary, the first 256 memory element of the SBox memory can be reconfigured. With the proper instruction in the ISA, the memory elements are reloaded according to desired application.

The DES algorithm consists of an SBox block, an EXOR block and six different permutation blocks, which are Initial Permutation, Inverse Initial Permutation, E-Table Permutation, Cipher (P) Permutation, PC1 Permutation and PC2 Permutation. The AES algorithm consists of an SBox block, a ShiftRows block, a MixColumns block and an EXOR block, and there are some additional blocks that are necessary for the key expansion of AES.

In this architecture, all the permutation blocks are combined in a single permutation module. Some parts of the SBox operations are implemented in a common structure. Of course, there is an additional work for the DES SBox operation, due to the preference for the correct part of the SBox module. All EXOR operations are performed in the same block of the ALU module with a single instruction.

The shift operations for AES and DES operations are different, because of the difference in algorithms' structure. Therefore, there are two different shift blocks in the architecture; one for bit-based shift operations and the other one is for the byte-based shift operations. DES uses bit-based shift operations, whereas AES uses byte-based shift operations.

The MixColumns block is based on the xtime function. This function describes the multiplication the data with “x” in the Galois Field (2^8). All the multiplication with “x” in Galois Field (2^8) can be performed with the xtime instruction in the implemented architecture, including the Inverse MixColumns operation in the decryption algorithm of AES. But this function is decreasing the throughput in the decryption operation due to multiple uses for the different coefficients in Inverse MixColumns operation.

As a result of common blocks and different blocks of the implemented architecture, DES algorithm can be performed using 21 different instructions with the proposed ISA. On the other hand, AES-128 algorithm can be performed using 32 different instructions. There are 9 common instructions like SBOX, EXOR and MVK0 (store round key values) in the ISA, which are used for both of the DES and AES algorithms. Therefore, it is clear that implementing AES and DES algorithms in a single design is an efficient way to decrease the area.

The hardware architecture of this design is implemented using SystemC. The main architecture is divided into modules and each module is implemented separately. The advantage of using modules is, changing one of the modules’ internal parameters without affecting the other modules’ parameters. Therefore, the module parameters of the implemented architecture can be changed to satisfy different algorithm's specifications for future work of this study. The simulation results are analyzed to verify the implemented architecture. The encryption and decryption algorithms for AES, DES and TDES are simulated for different data and key lengths.

There is always a tradeoff between area and speed parameters of the implemented design. In this design, the area is considered to be optimized and the design is implemented so as to minimize the total area.

Most of the instructions in the ISA are implemented as single cycle instructions. The purpose of this structure is making the hardware design simpler and as a result, the implemented instructions are the basic parts of the algorithms. Each instruction performs one simple operation in the crypto algorithms. However, this property brings the disadvantage of low throughput capability, because of the long processing time compared with other programmable crypto architectures, Cryptonite and CryptoManiac as given in [7] and [8] respectively. The main advantages of Cryptonite and CryptoManiac are their complex hardware architectures. In those architectures, there are special hardware blocks, which can perform several instructions faster, in less machine cycles. Consequently, total machine cycles of those architectures are relatively small, however they occupy comparatively large area.

The designed ISA structure performs AES encryption/decryption in 213 cycles, excluding the key expansion operations. DES encryption/decryption is performed in about 200 cycles. The performance results for Cryptonite processor are 70 cycles for AES and 35 cycles for DES. For CryptoManiac processor, the results are 90 cycles for AES and 130 cycles for DES. These results have been achieved at the expense of area. Cryptonite processor uses two different ALU modules and CryptoManiac uses 4 different functional units. Besides they have dedicated memory units to ALU and address generation. The area of these processors is expected to be at least three times larger than the area of the structure suggested in this thesis.

The ISA structures in this thesis can be modified and two or more instructions may be combined into one instruction to perform a specific block of the algorithm to increase the throughput as a future work. Another important issue for a future work may be using a reduced SBox memory structure for further area minimization. In the implemented design, the SBox memory occupies more than 40% of the total area. Therefore, reducing the SBox memory to half will be very efficient for small area applications. In that case there should be only one 256 byte memory block, and according to the application the necessary SBox elements should be loaded to the memory block before algorithm operations.

REFERENCES

[1] The Design of Rijndael AES – The Advanced Encryption Standard, John Daemen and Vincent Rijmen, Springer-Verlag, 2002.

[2] AES Proposal: Rijndael, John Daemen and Vincent Rijmen, September 3, 1999.

[3] Advanced Encryption Standard (AES), available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[4] Data Encryption Standard (DES), available: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[5] National Institute of Standards and Technology Computer Security Resource Center: http://csrc.nist.gov/

[6] Refik Sever, A. Neslin İsmailoğlu, Yusuf C. Tekmen and Murat Aşkar, *A High Speed ASIC Implementation of The Rijndael Algorithm*, IEEE International Symposium on Circuits and Systems, 2004.

[7] Rainer Buchty, Nevin Heintze, and Dino Oliva, *Cryptonite – A Programmable Crypto Processor Architecture for High-Bandwidth Applications*, ARCS 2004, LNCS 2981, pp. 184–198, 2004.

[8] Lisa Wu, Chris Weaver, and Todd Austin, *CryptoManiac: A Fast Flexible Architecture for Secure Communication*, in 28th Annual International Symposium on Computer Architecture, June 2001.

[9] Ricardo Chaves, Georgi Kuzmanov, Stamatis Vassiliadis and Leonel Sousa, *Reconfigurable Memory Based AES Co-Processor*, 20th International Parallel and Distributed Processing Symposium, 2006.

[10] Alireza Hodjat and Ingrid Verbauwhede, *Area-Throughput Trade-Offs for Fully Pipelined 30 to 70 Gbits/s AES Processors*, IEEE Transactions on Computers, April 2006.

[11] Oscar Perez, Yves Berviller, Camel Tanougast and Serge Weber, *Comparison of various strategies of implementation of the algorithm of encryption AES on FPGA*, IEEE International Symposium on Industrial Electronics, 2006.

[12] Yongzhi Fu, Lin Hao and Xuejie Zhang, *Design of An Extremely High Performance Counter Mode AES Reconfigurable Processor*, IEEE Computer Society, 2005.

[13] Alireza Hodjat, David D. Hwang, Bocheng Lai, Kris Tiri and Ingrid Verbauwhede, *A 3.84 Gbits/s AES Crypto Coprocessor with Modes of Operation in a 0.18-µm CMOS Technology*, GLSVLSI 2005.

[14] Chih-Pin Su, Chia-Lung Horng, Chih-Tsun Huang and Cheng-Wen Wu, *A Configurable AES Processor for Enhanced Security*, Asia and South Pacific Design Automation Conference, 2005.

[15] S. Pongyupinpanich, S. Phathumvanh, and S. Choomchuay, *A 32 Bits Architecture For an AES System*, International Symposium on Communications and Information Technologies, 2004.

[16] R. B. Lee, Z. Shi, and X. Yang, *Efficient permutation instructions for fast software cryptography*, IEEE Micro, vol. 21, pp. 56–69, December 2001.

[17] Nazar A. Saqib, Francisco Rodriguez-Henriquez and Arturo Diaz-Perez, *AES Algorithm Implementation - An efficient approach for Sequential and Pipeline Architectures*, Proceedings of the Fourth Mexican International Conference on Computer Science, 2003.

[18] Toby Schaffer, Alan Glaser and Paul D. Franzon, *Chip-Package Co-Implementation of a Triple DES Processor*, IEEE Transactions on Advanced Packaging, vol. 27, no. 1, February 2004.

[19] P. Kitsos, S. Goudevenos and O. Koufopavlou, *VLSI Implementations of The Triple-DES Block Cipher*, IEEE International Conference on Electronics, Circuits and Systems, 2003.

[20] Zhijie Shi, Xiao Yang and Ruby B. Lee, *Arbitrary Bit Permutations in One or Two Cycles*, IEEE 14th International Conference on Application-Specific Systems, Architectures and Processors, June 2003.

[21] Zhijie Jerry Shi and Ruby B. Lee, *Implementation Complexity of Bit Permutation Instructions*, 37th Annual Asilomar Conference on Signals, Systems and Computers, 2003.

[22] Yedidya Hilewitz, Zhijie Jerry Shi and Ruby B. Lee, *Comparing Fast Implementations of Bit Permutation Instructions*, 38th Annual Asilomar Conference on Signals, Systems and Computers, November 2004.

[23] John P. McGregor and Ruby B. Lee, *Architectural Techniques for Accelerating Subword Permutations With Repetitions*, IEEE Transactions on VLSI Systems, vol. 11, no. 3, June 2003.

[24] SystemC 2.0.1 Language Reference Manual, Open SystemC Initiative, 2003.

[25] SystemCrafter Sc, available: http://www.sytemcrafter.com/

Appendix A: SBox Tables

Table A.1: DES SBox S1

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Table A.2: DES SBox S2

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

Table A.3: DES SBox S3

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

Table A.4: DES SBox S4

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Table A.5: DES SBox S5

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

Table A.6: DES SBox S6

Row	Column Number															
Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

Table A.7: DES SBox S7

Row Num	Column Number															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

Table A.8: DES SBox S8

Row Num	Column Number															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Table A.9: AES SBox

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

Table A.10: AES Inverse SBox

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

Appendix B: Sample Programming Codes

B.1 128-bit AES Programming Code

Programming Code	Opcode	Machine Cycle
move enkey , 0x1	0x05	1
read iport	0x18	4
copy ibuffer	0x11	3
move key_value , 0x4	0x52	1
mvk0	0xd0	1
move reg_djnz , 0x0a	0x04	1
keyexp: mov1 state_reg0	0x50	1
rorb reg3	0xa0	1
sbox reg3	0x33	1
rcon	0xf0	1
exor reg3 , reg0	0x2b	1
move reg0 , reg8	0x55	1
exor reg0 , reg3	0x2c	1
move reg3 , reg11	0x59	1
exor reg1 , reg0	0x24	1
exor reg2 , reg1	0x25	1
exor reg3 , reg2	0x26	1
mvk0	0xd0	1
djnz reg_djnz , keyexp	0x01	2
move enkey , 0x0	0x06	1
read iport	0x18	4
copy ibuffer	0x11	3
exk0	0x60	2
exk1	0x70	2
move reg_djnz , 0x09	0x04	1
round: sbox reg0	0x30	1
sbox reg1	0x31	1
sbox reg2	0x32	1
sbox reg3	0x33	1
shft 128	0x38	3
mov1 state_reg0	0x50	1
xtme reg0	0x40	1
mix reg0 , reg8	0x48	1
xtme reg1	0x41	1
mix reg1 , reg9	0x49	1

xtme reg2	0x42	1
mix reg2 , reg10	0x4a	1
xtme reg3	0x43	1
mix reg3 , reg11	0x4b	1
exk0	0x61	2
exk1	0x71	2
djnz reg_djnz , round	0x01	2
sbox reg0	0x30	1
sbox reg1	0x31	1
sbox reg2	0x32	1
sbox reg3	0x33	1
shft 128	0x38	3
exk0	0x60	2
exk1	0x70	2
copy obuffer	0x15	3
wrte oport	0x19	4

total machine cycle for key expansion and encryption process : 364 cycles

B.2 TDES Programming Code

Programming Code	Opcode	Machine Cycle
move enkey , 0x1	0x05	1
read iport	0x18	4
copy ibuffer	0x11	3
move key_value , 0x2	0x5a	1
pc1p reg0 , reg1	0xb0	1
rord reg0 , reg1	0xa1	1
mov1 state_reg0	0x50	1
pc2p reg0 , reg1	0xb1	1
mvk0	0xd0	1
mov1 back_state_reg0	0x5b	1
rord reg0 , reg1	0xa1	1
mov1 state_reg0	0x50	1
pc2p reg0 , reg1	0xb1	1
mvk0	0xd0	1
move reg_djnz , 0x06	0x04	1
keypart1: mov1 back_state_reg0	0x5b	1
rord reg0 , reg1	0xa1	1
rord reg0 , reg1	0xa1	1
mov1 state_reg0	0x50	1

```
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
djnz    reg_djnz , keypart1              0x01                  2


mov1 back_state_reg0                     0x5b                  1
rord reg0 , reg1                         0xa1                  1
mov1 state_reg0                          0x50                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
move   reg_djnz , 0x06                   0x04                  1
keypart2:  mov1 back_state_reg0          0x5b                  1
rord reg0 , reg1                         0xa1                  1
rord reg0 , reg1                         0xa1                  1
mov1 state_reg0                          0x50                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
djnz    reg_djnz , keypart2              0x01                  2
mov1 back_state_reg0                     0x5b                  1
rord reg0 , reg1                         0xa1                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1

move enkey 1                             0x05                  1
read iport                               0x18                  4
copy ibuffer                             0x11                  3
move key_value , 0x2                     0x5a                  1
pc1p  reg0 , reg1                        0xb0                  1
rord reg0 , reg1                         0xa1                  1
mov1 state_reg0                          0x50                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
mov1 back_state_reg0                     0x5b                  1
rord reg0 , reg1                         0xa1                  1
mov1 state_reg0                          0x50                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
move   reg_djnz , 0x06                   0x04                  1
keypart1:  mov1 back_state_reg0          0x5b                  1
rord reg0 , reg1                         0xa1                  1
rord reg0 , reg1                         0xa1                  1
mov1 state_reg0                          0x50                  1
pc2p  reg0 , reg1                        0xb1                  1
mvk0                                     0xd0                  1
djnz    reg_djnz , keypart1              0x01                  2
mov1 back_state_reg0                     0x5b                  1
rord reg0 , reg1                         0xa1                  1
```

```
mov1 state_reg0                     0x50          1
pc2p  reg0 , reg1                   0xb1          1
mvk0                                0xd0          1
move   reg_djnz , 0x6               0x04          1


keypart2:  mov1 back_state_reg0     0x5b          1
rord reg0 , reg1                    0xa1          1
rord reg0 , reg1                    0xa1          1
mov1 state_reg0                     0x50          1
pc2p  reg0 , reg1                   0xb1          1
mvk0                                0xd0          1
djnz    reg_djnz , keypart2         0x01          2
mov1 back_state_reg0                0x5b          1
rord reg0 , reg1                    0xa1          1
pc2p  reg0 , reg1                   0xb1          1
mvk0                                0xd0          1

move enkey , 0x0                    0x06          1
read iport                          0x18          4
copy ibuffer                        0x11          3

inip  reg0 , reg1                   0xb2          1
move reg_djnz , 0x10                0x04          1
round1:   mov1 state_reg0           0x50          1
bmap  reg0 , reg1                   0xa2          1
etbp  reg0 , reg1                   0xb3          1
exk0                                0x60          2
sbad reg0 , reg1                    0xa3          1
sbox reg0                           0x30          1
sbox reg1                           0x31          1
cipp  reg0 , reg1                   0xb4          1
drex reg0 , reg1                    0xa6          1
djnz    reg_djnz , round1           0x01          2
swap reg0 , reg1                    0xa5          1
invp  reg0 , reg1                   0xb5          1

inip  reg0 , reg1                   0xb2          1
move reg_djnz , 0x10                0x04          1
round1:   mov1 state_reg0           0x50          1
bmap  reg0 , reg1                   0xa2          1
etbp  reg0 , reg1                   0xb3          1
exk0                                0x60          2
sbad reg0 , reg1                    0xa3          1
sbox reg0                           0x30          1
sbox reg1                           0x31          1
```

cipp reg0 , reg1	0xb4	1
drex reg0 , reg1	0xa6	1
djnz reg_djnz , round1	0x01	2
swap reg0 , reg1	0xa5	1
invp reg0 , reg1	0xb5	1

inip reg0 , reg1	0xb2	1
move reg_djnz , 0x10	0x04	1
round1: mov1 state_reg0	0x50	1
bmap reg0 , reg1	0xa2	1
etbp reg0 , reg1	0xb3	1
exk1	0x70	2
sbad reg0 , reg1	0xa3	1
sbox reg0	0x30	1
sbox reg1	0x31	1
cipp reg0 , reg1	0xb4	1
drex reg0 , reg1	0xa6	1
djnz reg_djnz , round1	0x01	2
swap reg0 , reg1	0xa5	1
invp reg0 , reg1	0xb5	1

copy obuffer	0x15	3
wrte oport	0x19	4

total machine cycle for key expansion and encryption process : 847 cycles