Design and Implementation of Data Encryption Standard using VHDL

MAJOR PROJECT THESIS

Submitted in Partial Fulfillment of the Requirements for the Award of the Degree of

MASTER OF ENGINEERING

in

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

MAMTA RANI
Delhi University Roll No. 12681

Under the Guidance of

Ms. RAJESHWARI PANDEY
Department of Electronics and Communication

Department of Electronics and Communication Engineering
Delhi College of Engineering
Delhi-110042

CERTIFICATE

Department of Electronics and Communication Engineering
Delhi College of Engineering
University of Delhi
Delhi-110042

This is to certify that the Major project thesis entitled **“Design and Implementation of Data Encryption Standard VHDL”,** being submitted by **Mamta Rani** in the partial fulfillment of the requirement for the degree of Master of Engineering in Electronics and Communication in the Department of Electronics and Communication, Delhi College of Engineering, University of Delhi is a record of bonafide work done by her under my supervision and guidance. It is also certified that the dissertation has not been submitted elsewhere for any other degree.

She has worked under my supervision and guidance. She has fulfilled all the requirements for submission of the Major Project thesis, which has reached the requisite standard.

Ms. Rajeshwari Pandey
Assistant Professor
Delhi College of Engineering
Bawana Road, New Delhi

ACKNOWLEDGEMENT

I am highly indebted to my Project guide **Ms. Rajeshwari Pandey, Department of Electronics and Communication Engineering,** for giving me the opportunity to work under her invaluable supervision. She had encouraged and guided me to accomplish this research work.

I must acknowledge **Dr. Asok Bhattacharyya (Professor and H.O.D)** of Electronics and Communication Engineering Department, Delhi College of Engineering, for his invaluable guidance during the research work.

I must also acknowledge the staff of Department of Electronics and Communication Engineering Laboratories, Delhi College of Engineering, for their support and help during the research work.I would also like to express my gratitude to all my colleagues in particular to those at the Department of Electronics and Communication Engineering for their support, co-operation and fruitful discussions on diverse research topics.

I want to thank my family & friends for their sincere interest in my work and their moral support.

Mamta Rani
University Roll No. 12681
Delhi College of Engineering
University of Delhi, Delhi

ABSTRACT

Many people wish to communicate privately. To prevent unauthorized persons from extracting information from the communication channel or injecting misinformation into the communication channel, messages need to be disguised by encryption. At the transmitter, the plaintext is encrypted to produce the ciphertext. The ciphertext is transmitted over an insecure channel to the receiver. The receiver then decrypts the ciphertext to obtain the original plaintext.

DES, which stands for Data Encryption Standard is a block encryption algorithm adopted by the National Bureau of Standards. With this algorithm, a 64-bit plaintext and a 64-bit key are provided as input. By applying a sequence of initial permutation, switch, shift on the key and plaintext, the 64-bit ciphertext is generated at the output after 16 clock cycles.

A test bench for simulation is critically important for the final success of the whole work. This test bench provides a sequence of key and plaintext to the DES design.

With the test bench, the pre-synthesis simulation is then made using Active HDL 6.3. This is an RTL level simulation which verifies the logic functionality of the code without gatelevel information involved. After the successful pre-synthesis simulation, IDE Xyling Project Navigator is used to synthesize the DES design.

The main objective of the project is to design a synthesizable VHDL model for Data Encryption Standard Algorithm.The basic idea behind a synthesizable model is the need to implement the algorithm on FPGA.Implementing cryptographic algorithm on reconfigurable hardware provides major benefits over VLSI and software platforms since they offer high speed similar to VLSI and high flexibility similar to software.

CONTENTS

		Page No.
Chapter 1		**1**
1.1	Introduction	1
1.2	Cryptography Goals	2
Chapter 2		**4**
2.1	Block Ciphers and Stream Ciphers	4
2.2	Symmetric and Asymmetric Encryptions	5
2.3	Conventional Encryption	9
2.4	Cryptanalysis	10
Chapter 3		**12**
3.1	DES Overview	12
3.2	Key Scheduling	15
3.3	Plaintext Preparation	18
3.4	DES Core Function	19
3.5	Ciphertext Preparation	26
3.6	Encryption and Decryption	26
Chapter 4		**30**
4.1	FPGA Introduction	30
4.2	History	31
4.3	Architecture	32
4.4	Basic Process Techonology Types	35
Chapter 5		**36**
5.1	Implementation	36
5.2	Results	37
5.3	Conclusion and Future Scope	41
References		**42**
Appendix		**43**

LIST OF FIGURES

Page No.

Fig. 1.1: Encryption-Decryption Flow 2

Fig. 2.1: Block Cipher ECB Mode 5

Fig. 2.2: Stream Cipher (Simple Mode) 5

Fig. 2.3: Symmetric Encryption 6

Fig. 2.4: Asymmetric Encryption 7

Fig. 2.5: Encryption Model 10

Fig. 3.1: General Depiction of DES 15

Fig. 3.2: Single Round of DES 20

Fig. 4.1: FPGA Logic Block 32

Fig. 4.2: Logic Block Pin Locations 33

Fig. 4.3: Switch Box Topology 34

CHAPTER 1

1.1 Introduction

During this time when the Internet provides essential communication between tens of millions of people and is being increasingly used as a tool for commerce, security becomes a tremendously important issue to deal with.

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography, while cryptography is necessary for secure communications, it is not by itself sufficient

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

Cryptography is usually referred to as "the study of secret", while nowadays is most attached to the definition of encryption. Encryption is the process of converting plain text "unhidden" to a cryptic text "hidden" to secure it against data thieves. This process has another part where cryptic text needs to be decrypted on the other end to be understood.

Fig. 1.1 shows the simple flow of commonly used encryption algorithms

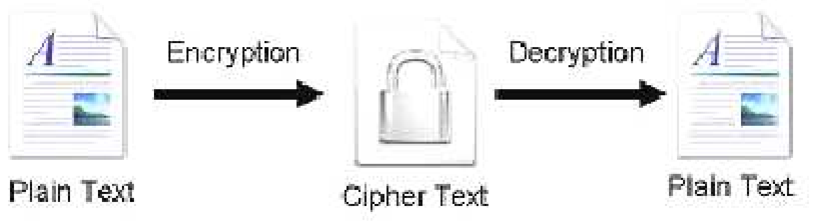


Fig. 1.1: Encryption-Decryption Flow

Cryptographic system is "a set of cryptographic algorithms together with the key management processes that support use of the algorithms in some application context."

This definition defines the whole mechanism that provides the necessary level of security comprised of network protocols and data encryption algorithms.

1.2 Cryptography Goals

This section explains the five main goals behind using Cryptography.

Every security system must provide a bundle of security functions that can assure the secrecy of the system.These functions are usually referred to as the goals of the security system. These goals can be listed under the following five main categories

Authentication: This means that before sending and receiving data using the system, the receiver and sender identity should be verified.

Secrecy or Confidentiality: Usually this function (feature) is how most people identify a secure system. It means that only the authenticated people are able to interpret the message (date) content and no one else.

Integrity: Integrity means that the content of the communicated data is assured to be free from any type of modification between the end points (sender and receiver).

Non-Repudiation: This function implies that neither the sender nor the receiver can falsely deny that they have sent a certain message.

Service Reliability and Availability: Since secure systems usually get attacked by intruders, which may affect their availability and type of service to their users. Such systems should provide a way to grant their users the quality of service they expect.

CHAPTER 2

2.1 Block Ciphers and Stream Ciphers

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

Stream ciphers come in several flavors but two are worth mentioning here. Self-synchronizing stream ciphers calculate each bit in the keystream as a function of the previous n bits in the keystream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the n-bit keystream it is. One problem is error propagation; a garbled bit in transmission will result in n garbled bits at the receiving side. Synchronous stream ciphers generate the keystream in a fashion independent of the message stream but by using the same keystream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the keystream will eventually repeat.

ECB (Electronic Codebook Mode) is the basic form of clock cipher where data blocks are encrypted directly to generate its correspondent ciphered blocks (shown in Fig. 2.1). More discussion about modes of operations will be discussed later.

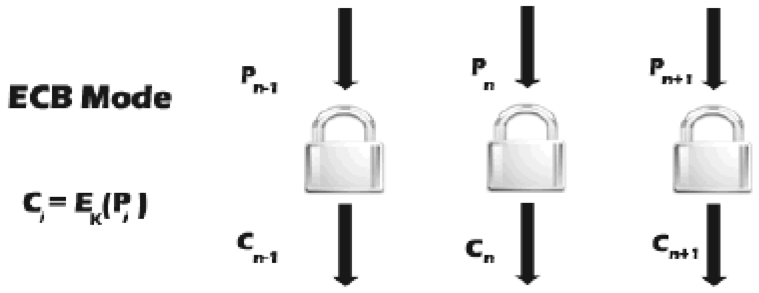


Fig. 2.1: Block Cipher ECB Mode

Stream cipher functions on a stream of data by operating on it bit by bit. Stream cipher consists of two major components: a key stream generator, and a mixing function.

Mixing function is usually just an XOR function, while key stream generator is the main unit in stream cipher encryption technique. For example, if the key stream generator produces a series of zeros, the outputted ciphered stream will be identical to the original plain text. Fig. 2.2 shows the operation of the simple mode in stream cipher.

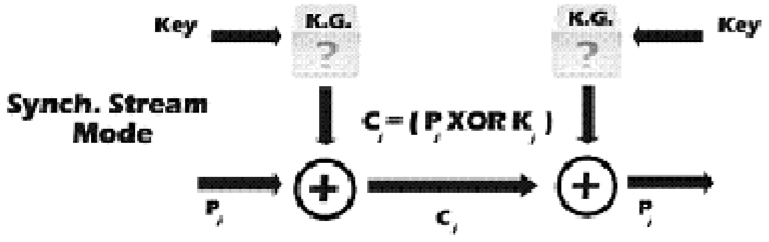


Fig. 2.2: Stream Cipher (Simple Mode)

2.2 Symmetric and Asymmetric Encryptions

Data encryption procedures are mainly categorized into two categories depending on the type of security keys used to encrypt/decrypt the secured data. These two categories are:

Asymmetric and Symmetric encryption techniques

2.2.1 Symmetric Encryption

In this type of encryption, the sender and the receiver agree on a secret (shared) key. Then they use this secret key to encrypt and decrypt their sent messages. Fig. 2.3 shows the process of symmetric cryptography. Node A and B first agree on the encryption technique to be used in encryption and decryption of communicated data.

Then they agree on the secret key that both of them will use in this connection. After the encryption setup finishes, node A starts sending its data encrypted with the shared key, on the other side node B uses the same key to decrypt the encrypted messages.

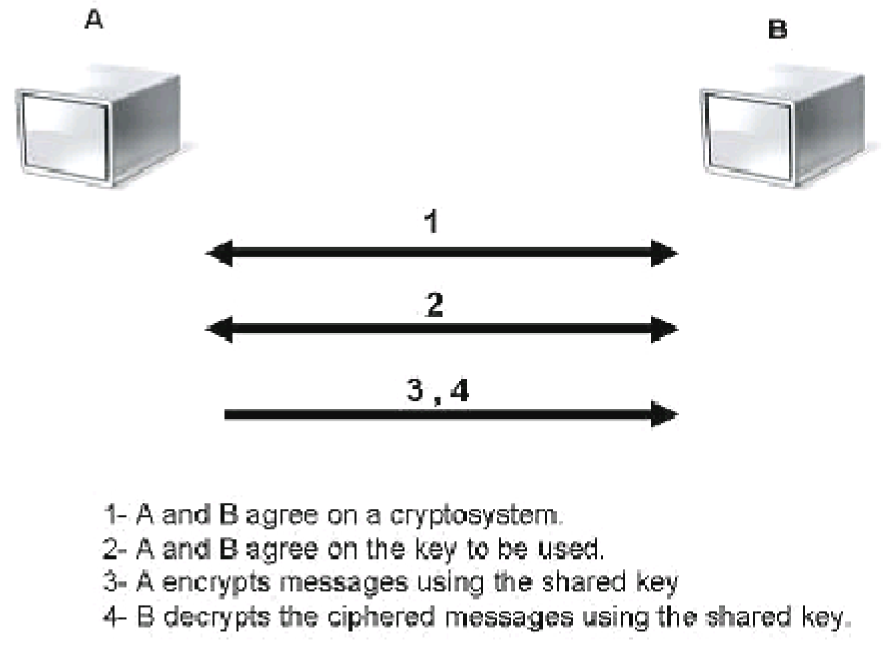

A
B
1
2
3 , 4
1- A and B agree on a cryptosystem.
2- A and B agree on the key to be used.
3- A encrypts messages using the shared key
4- B decrypts the ciphered messages using the shared key.

Fig. 2.3: Symmetric Encryption

The main concern behind symmetric encryption is how to share the secret key securely between the two peers. If the key gets known for any reason, the whole system collapses.

The key management for this type of encryption is troublesome, especially if a unique secret key is used for each peer-to-peer connection, then the total number of secret keys to be saved and managed for n-nodes will be n(n-1)/2.

2.2.2 Asymmetric Encryption

Asymmetric encryption is the other type of encryption where two keys are used. To explain more, what Key1 can encrypt only Key2 can decrypt, and vice versa. It is also known as Public Key Cryptography (PKC), because users tend to use two keys: public key, which is known to the public, and private key which is known only to the user.

Fig. 2.4 below illustrates the use of the two keys between node A and node B. After agreeing on the type of encryption to be used in the connection, node B sends its public key to node A. Node A uses the received public key to encrypt its messages. Then when the encrypted messages arrive, node B uses its private key to decrypt them.

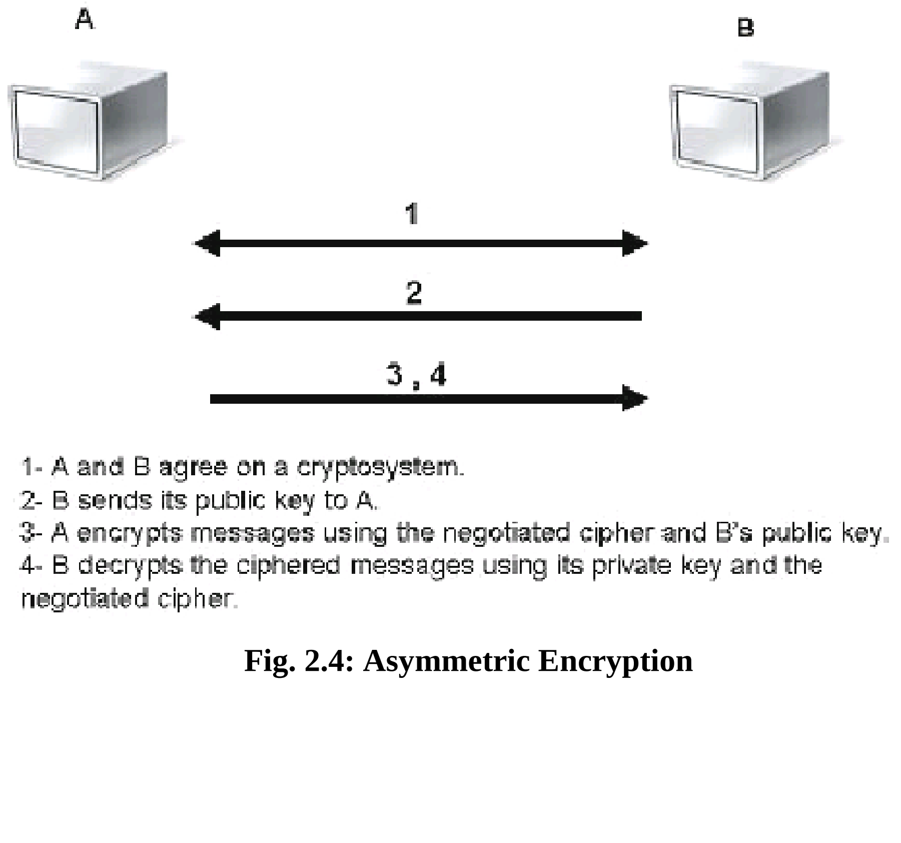

A
B
1
2
3 , 4
1- A and B agree on a cryptosystem.
2- B sends its public key to A.
3- A encrypts messages using the negotiated cipher and B's public key.
4- B decrypts the ciphered messages using its private key and the negotiated cipher.

Fig. 2.4: Asymmetric Encryption

This capability surmounts the symmetric encryption problem of managing secret keys. But on the other hand, this unique feature of public key encryption makes it Mathematically more prone to attacks. Moreover, asymmetric encryption techniues are almost 1000 times slower than symmetric techniques, because they require more computational processing power .To get the benefits of both methods, a hybrid technique is usually used. In this technique, asymmetric encryption is used to exchange the secret key, symmetric encryption is then used to transfer data between sender and receiver.

Advantages of Public Key Cryptography

- Only the private keys must be kept secret.
- The administration of keys on a n/w requires the presence of only a functionally trusted TTP.
- A private key/public key pair may remain unchanged for considerable periods of time e.g. many sessions or even many years.
- In a large n/w, the number of keys necessary may be considerably smaller than in the symmetric key scenario.

Disadvantages of Public Key Cryptography

- Throughput rates are several orders slower than symmetric key schemes.
- Key sizes are typically much larger than those required for symmetric key encryption.
- No public key scheme has been proven to be secure.

Advantages of Symmetric Key Cryptography

- High rates of data throughput
- Key length is relatively short
- Produce stronger ciphers

Disadvantages of Symmetric Key Cryptography

- In a two party communication, the key must remain secret at both ends
- In a large n/w, there are many key pairs to be managed.
- Keys are to be changed frequently, mostly for each communication session.

2.3 Conventional Encryption

A symmetric encryption scheme has five ingredients

Plaintext: This is the original intelligible message or data that is fed into the algorithm as input.

Encryption algorithm: The encryption algorithm performs various substitution and transformation on the plain text.

Secret key: The secret key is also input to the encryption algorithm.The key is value independentof the plain text.The algorithm will produce a different output depending on the specific key used at that time

Ciphertext: This is the scrambled message produced at the output.It depends on the plaintext and the secret key.For a given message,two different keys will produce two different ciphertext.The ciphertext is an apparently random stream of data and is unintelligible.

Decryption algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plain text.

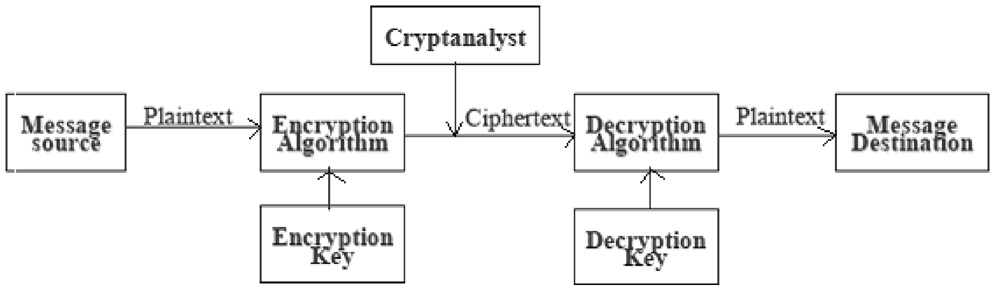


Fig. 2.5: Encryption Model

2.4 Cryptanalysis

There are two general approaches to attacking a conventional encryption scheme:

Cryptanalysis: Cryptanalysis attacks rely on the nature of the algorithm plus perhaps some knowledge of the general characteristics of the plaintext or even some sample plaintext-ciphertext pairs. This type of attack exploits the characteristics of the algorithm to attempt to deduce a specific plaintext or to deduce the key being used. If the attack succeeds in deducing the key, the effect is catastrophic: All future and past messages encrypted with that key are compromised.

Brute-force attack: The attacker tries possible tries every possible key on a piece of ciphertext until an intelligence translation into plaintext is obtained. On average, half of all possible keys must be tried to achieve success.

2.4.1 Types of Attack on Encrypted Messages

Types of Attack	**Known to Cryptanalyst**
Ciphertext only	Encryption algorithm Ciphertext to be decoded
Known plaintext	Encryption algorithm Ciphertext to be decoded One or more plaintext-cipher text pairs formed with the secret key
Chosen plaintext	Encryption algorithm Ciphertext to be decoded Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key
Chosen ciphertext	Encryption algorithm Ciphertext to be decoded Purported ciphertext chosen by cryptanalyst together with its corresponding plaintext generated with the secret key
Chosen text	Encryption algorithm Ciphertext to be decoded Plaintext message chosen by cryptanalyst, together with its corresponding ciphertext generated with the secret key Purported ciphertext chosen by cryptanalyst together with its corresponding plaintext generated with the secret key

CHAPTER 3

3.1 DES Overview

In 1972, the National Institute of Standards and Technology (called the National Bureau of Standards at the time) decided that a strong cryptographic algorithm was needed to protect non-classified information. The algorithm was required to be cheap, widely available, and very secure. NIST envisioned something that would be available to the general public and could be used in a wide variety of applications. So they asked for public proposals for such an algorithm. In 1974 IBM submitted the Lucifer algorithm, which appeared to meet most of NIST's design requirements.

NIST enlisted the help of the National Security Agency to evaluate the security of Lucifer. At the time many people distrusted the NSA due to their extremely secretive activities, so there was initially a certain degree of skepticism regarding the analysis of Lucifer. One of the greatest worries was that the key length, originally 128 bits, was reduced to just 56 bits, weakening it significantly. The NSA was also accused of changing the algorithm to plant a "back door" in it that would allow agents to decrypt any information without having to know the encryption key. But these fears proved unjustified and no such back door has ever been found.

The modified Lucifer algorithm was adopted by NIST as a federal standard on November 23, 1976. Its name was changed to the Data Encryption Standard (DES). The algorithm specification was published in January 1977, and with the official backing of the government it became a very widely employed algorithm in a short amount of time.

Unfortunately, over time various shortcut attacks were found that could significantly reduce the amount of time needed to find a DES key by brute force. And as computers became progressively faster and more powerful, it was recognized that a 56-bit key was simply not large enough for high security applications. As a result of these serious flaws, NIST abandoned their official endorsement of DES in 1997 and began work on a replacement, to be called the Advanced Encryption Standard (AES). Despite the growing concerns about its vulnerability, DES is still widely used by financial services and other industries worldwide to protect sensitive on-line applications.

To highlight the need for stronger security than a 56-bit key can offer, RSA Data Security has been sponsoring a series of DES cracking contests since early 1997. In 1998 the Electronic Frontier Foundation won the RSA DES Challenge II-2 contest by breaking DES in less than 3 days. EFF used a specially developed computer called the DES Cracker, which was developed for under $250,000. The encryption chip that powered the DES Cracker was capable of processing 88 billion keys per second. More recently, in early 1999, Distributed. Net used the DES Cracker and a worldwide network of nearly 100,000 PCs to win the RSA DES Challenge III in a record breaking 22 hours and 15 minutes. The DES Cracker and PCs combined were testing 245 billion keys per second when the correct key was found. In addition, it has been shown that for a cost of one million dollars a dedicated hardware device can be built that can search all possible DES keys in about 3.5 hours. This just serves to illustrate that any organization with moderate resources can break through DES with very little effort these days.

In Depth

DES encrypts and decrypts data in 64-bit blocks, using a 64-bit key (although the effective key strength is only 56 bits). It takes a 64-bit block of plaintext as input and outputs a 64-bit block of ciphertext. Since it always operates on blocks of equal size and it uses both permutations and substitutions in the algorithm, DES is both a block cipher and a product cipher.

DES has 16 rounds, meaning the main algorithm is repeated 16 times to produce the ciphertext. It has been found that the number of rounds is exponentially proportional to the amount of time required to find a key using a brute-force attack. So as the number of rounds increases, the security of the algorithm increases exponentially.

The processing of the plain text proceeds in three phrases.First the 64 bit plaintext passes through an initial permutation that rearranges the bits to produce the permuted input.This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last round consist of 64 bits that are a function of the input plaintext and the key.The left and the right halves of the output are swapped to produce the preoutput. Finally the preoutput is passed through a permutation that is the inverse of the initial permutation function, to produce the 64 bit cipher text.

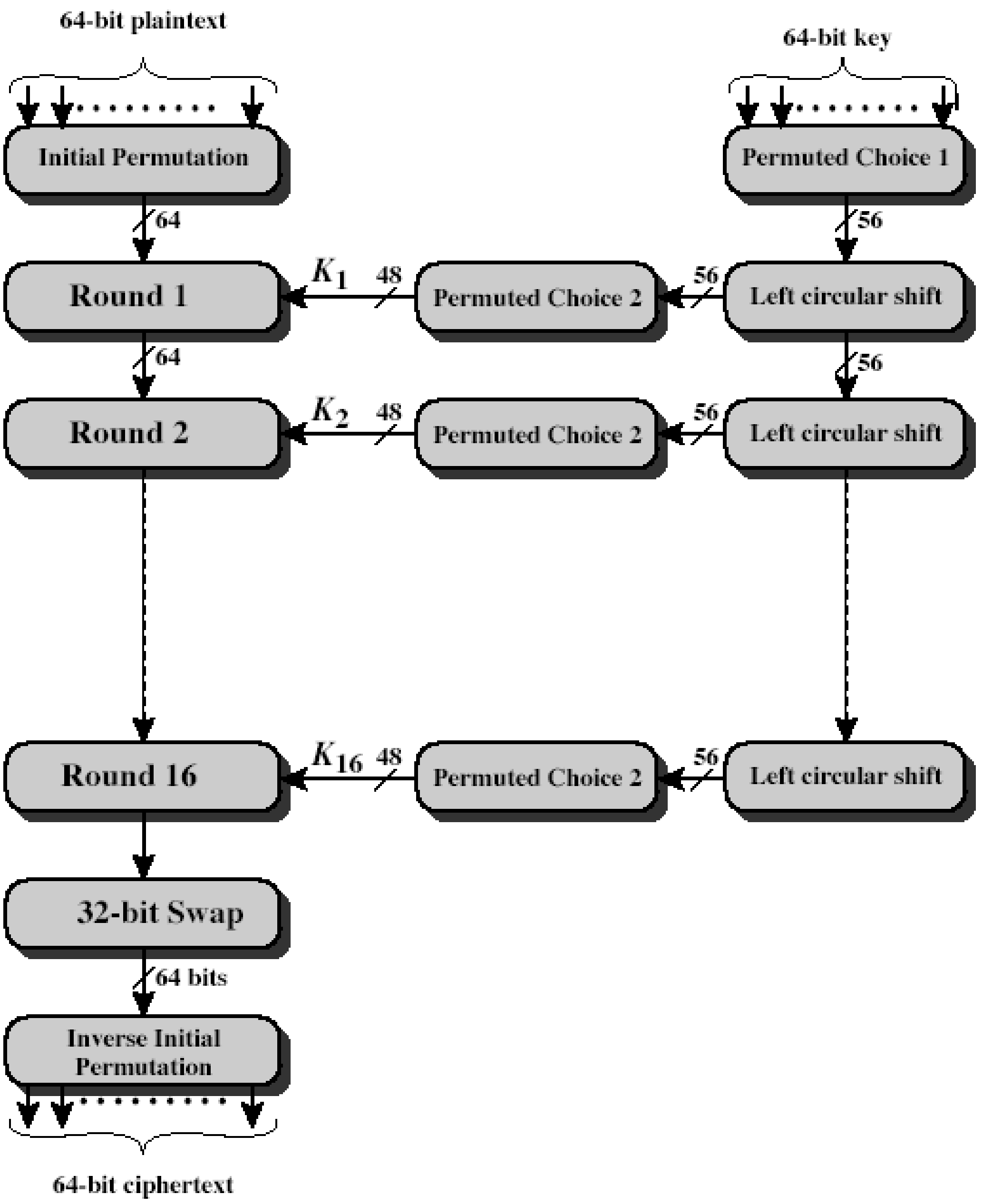


Fig. 3.1: General Depiction of DES

3.2 Key Scheduling

Although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length.

The first step is to pass the 64-bit key through a permutation called Permuted Choice 1, or PC-1 for short. The table for this is given below. Note that in all subsequent descriptions of bit numbers, 1 is the left-most bit in the number, and n is the rightmost bit.

PC-1: Permuted Choice 1							
Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**
1	57	49	41	33	25	17	9
8	1	58	50	42	34	26	18
15	10	2	59	51	43	35	27
22	19	11	3	60	52	44	36
29	63	55	47	39	31	23	15
36	7	62	54	46	38	30	22
43	14	6	61	53	45	37	29
50	21	13	5	28	20	12	4

For example, we can use the PC-1 table to figure out how bit 30 of the original 64-bit key transforms to a bit in the new 56-bit key. Find the number 30 in the table, and notice that it belongs to the column labeled 5 and the row labeled 36. Add up the value of the row and column to find the new position of the bit within the key. For bit 30, 36 + 5 = 41, so bit 30 becomes bit 41 of the new 56-bit key. Note that bits 8, 16, 24, 32, 40, 48, 56 and 64 of the original key are not in the table. These are the unused parity bits that are discarded when the final 56-bit key is created.

Now that we have the 56-bit key, the next step is to use this key to generate 16 48-bit subkeys, called K[1]-K[16], which are used in the 16 rounds of DES for encryption and decryption. The procedure for generating the subkeys - known as key scheduling - is fairly simple:

1. Set the round number R to 1.
2. Split the current 56-bit key, K, up into two 28-bit blocks, L (the left-hand half) and R (the right-hand half).
3. Rotate L left by the number of bits specified in the table below, and rotate R left by the same number of bits as well.
4. Join L and R together to get the new K.
5. Apply Permuted Choice 2 (PC-2) to K to get the final K[R], where R is the round number we are on.
6. Increment R by 1 and repeat the procedure until we have all 16 subkeys K[1]-K[16].

Here are the tables involved in these operations:

Subkey Rotation Table																
Round Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Number of bits to rotate	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

PC-2: Permuted Choice 2						
Bit	**0**	**1**	**2**	**3**	**4**	**5**
1	14	17	11	24	1	5
7	3	28	15	6	21	10
13	23	19	12	4	26	8
19	16	7	27	20	13	2
25	41	52	31	37	47	55
31	30	40	51	45	33	48
37	44	49	39	56	34	53
43	46	42	50	36	29	32

3.3 Plaintext Preparation

Once the key scheduling has been performed, the next step is to prepare the plaintext for the actual encryption. This is done by passing the plaintext through a permutation called the Initial Permutation, or IP for short. This table also has an inverse, called the Inverse Initial Permutation, or IP^(-1). Sometimes IP^(-1) is also called the Final Permutation.

Both of these tables are shown below.

IP: Initial Permutation								
Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	58	50	42	34	26	18	10	2
9	60	52	44	36	28	20	12	4
17	62	54	46	38	30	22	14	6
25	64	56	48	40	32	24	16	8
33	57	49	41	33	25	17	9	1
41	59	51	43	35	27	19	11	3
49	61	53	45	37	29	21	13	5
57	63	55	47	39	31	23	15	7

IP^(-1): Inverse Initial Permutation								
Bit	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	40	8	48	16	56	24	64	32
9	39	7	47	15	55	23	63	31
17	38	6	46	14	54	22	62	30
25	37	5	45	13	53	21	61	29
33	36	4	44	12	52	20	60	28
41	35	3	43	11	51	19	59	27
49	34	2	42	10	50	18	58	26
57	33	1	41	9	49	17	57	25

These tables are used just like PC-1 and PC-2 were for the key scheduling. By looking at the table is becomes apparent why one permutation is called the inverse of the other. For example, let's examine how bit 32 is transformed under IP. In the table, bit 32 is located at the intersection of the column labeled 4 and the row labeled 25. So this bit becomes bit 29 of the 64-bit block after the permutation. Now let's apply IP^(-1). In IP^(-1), bit 29 is located at the intersection of the column labeled 7 and the row labeled 25. So this bit becomes bit 32 after the permutation. And this is the bit position that we started with before the first permutation. So IP^(-1) really is the inverse of IP. It does the exact opposite of IP. On running a block of plaintext through IP and then pass the resulting block through IP^(-1),resultant is the original block.

3.4 DES Core Function

Once the key scheduling and plaintext preparation have been completed, the actual encryption or decryption is performed by the main DES algorithm. The 64-bit block of input data is first split into two halves, L and R. L is the left-most 32 bits, and R is the right-most 32 bits. The following process is repeated 16 times, making up the 16 rounds of standard DES. The 16 sets of halves are L[0]-L[15] and R[0]-R[15].

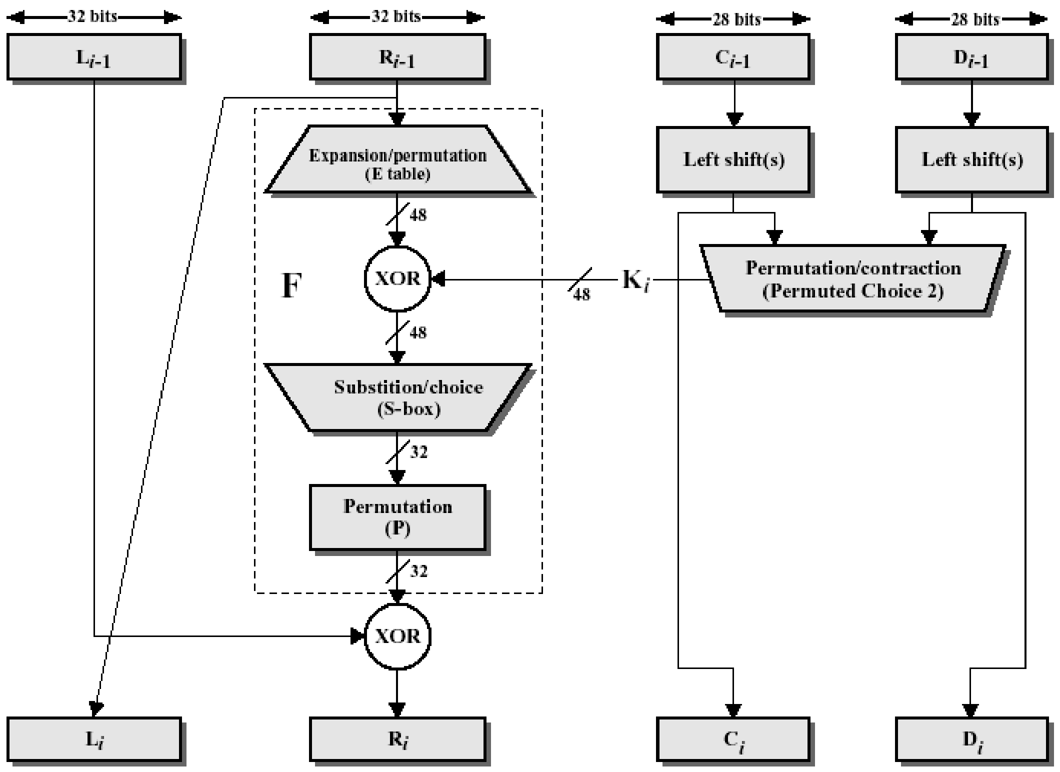


Fig. 3.2: Single Round of DES

1. R[I-1] - where I is the round number, starting at 1 - is taken and fed into the E-Bit Selection Table, which is like a permutation, except that some of the bits are used more than once. This expands the number R[I-1] from 32 to 48 bits to prepare for the next step.
2. The 48-bit R[I-1] is XORed with K[I] and stored in a temporary buffer so that R[I-1] is not modified.
3. The result from the previous step is now split into 8 segments of 6 bits each. The left-most 6 bits are B[1], and the right-most 6 bits are B[8]. These blocks form the index into the S-boxes, which are used in the next step. The Substitution boxes, known as S-boxes, are a set of 8 two-dimensional arrays, each with 4 rows and 16 columns. The numbers in the boxes are always 4 bits in length, so their values range from 0-15. The S-boxes are numbered S[1]-S[8].

4. Starting with B[1], the first and last bits of the 6-bit block are taken and used as an index into the row number of S[1], which can range from 0 to 3, and the middle four bits are used as an index into the column number, which can range from 0 to 15. The number from this position in the S-box is retrieved and stored away. This is repeated with B[2] and S[2], B[3] and S[3], and the others up to B[8] and S[8]. At this point, 8 4-bit numbers, which when strung together one after the other in the order of retrieval, give a 32-bit result.
5. The result from the previous stage is now passed into the P Permutation.
6. This number is now XORed with L[I-1], and moved into R[I]. R[I-1] is moved into L[I].
7. At this point we have a new L[I] and R[I]. Here, we increment I and repeat the core function until I = 17, which means that 16 rounds have been executed and keys K[1]-K[16] have all been used.

When L[16] and R[16] have been obtained, they are joined back together in the same fashion they were split apart (L[16] is the left-hand half, R[16] is the right-hand half), then the two halves are swapped, R[16] becomes the left-most 32 bits and L[16] becomes the right-most 32 bits of the pre-output block and the resultant 64-bit number is called the pre-output.

Tables used in the DES Core Function

E-Bit Selection Table						
Bit	**0**	**1**	**2**	**3**	**4**	**5**
1	32	1	2	3	4	5
7	4	5	6	7	8	9
13	8	9	10	11	12	13
19	12	13	14	15	16	17
25	16	17	18	19	20	21
31	20	21	22	23	24	25
37	24	25	26	27	28	29
43	28	29	30	31	32	1

P Permutation				
Bit	**0**	**1**	**2**	**3**
1	16	7	20	21
5	29	12	28	17
9	1	15	23	26
13	5	18	31	10
17	2	8	24	14
21	32	27	3	9
25	19	13	30	6
29	22	11	4	25

S-Box 1: Substitution Box 1																
Row / Column	**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

S-Box 2: Substitution Box 2																
Row / Column	**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

S-Box 3: Substitution Box 3																
Row / Column	**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

S-Box 4: Substitution Box 4																
Row / Column	**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

S-Box 5: Substitution Box 5																
Row / Column	**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

S-Box 6: Substitution Box 6																
Row / Column	**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

S-Box 7: Substitution Box 7																
Row / Column	**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

S-Box 8: Substitution Box 8																
Row / Column	**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

How to use the S-Boxes

The purpose of this example is to clarify how the S-boxes work. Consider the following 48-bit binary number:

011101000101110101000111101000011100101101011101

In order to pass this through steps 3 and 4 of the Core Function as outlined above, the number is split up into 8 6-bit blocks, labeled B[1] to B[8] from left to right:

011101 000101 110101 000111 101000 011100 101101 011101

Now, eight numbers are extracted from the S-boxes - one from each box:

B[1] = S[1](01,1110) = S[1][1][14] = 3 = 0011

B[2] = S[2](01,0010) = S[2][1][2] = 4 = 0100

B[3] = S[3](11,1010) = S[3][3][10] = 14 = 1110

B[4] = S[4](01,0011) = S[4][1][3] = 5 = 0101

B[5] = S[5](10,0100) = S[5][2][4] = 10 = 1010

B[6] = S[6](00,1110) = S[6][0][14] = 5 = 0101

B[7] = S[7](11,0110) = S[7][3][6] = 10 = 1010

B[8] = S[8](01, 1110) = S[8][1][14] = 9 = 1001

In each case of S[n][row][column], the first and last bits of the current B[n] are used as the row index, and the middle four bits as the column index.

The results are now joined together to form a 32-bit number which serves as the input to stage 5 of the Core Function (the P Permutation):

00110100111001011010010110101001

3.5 Ciphertext Preparation

The final step is to apply the permutation IP^(-1) to the pre-output. The result is the completely encrypted ciphertext.

3.6 Encryption and Decryption

The same algorithm can be used for encryption or decryption. The method described above will encrypt a block of plaintext and return a block of ciphertext. In order to decrypt the ciphertext and get the original plaintext again, the procedure is simply repeated but the subkeys are applied in reverse order, from K[16]-K[1]. That is, stage 2 of the Core Function as outlined above changes from R[I-1] XOR K[I] to R[I-1] XOR K[17-I]. Other than that, decryption is performed exactly the same as encryption

3.7 Modes of Operation

This section explains the two most common modes of operations in Block Cipher encryption-ECB and CBCwith a quick visit to other modes.

There are many variances of block cipher, where different techniques are used to strengthen the security of the system. The most common methods are: ECB (Electronic Codebook Mode), CBC (Chain Block Chaining Mode), and OFB (Output Feedback Mode). ECB mode is the CBC mode uses the cipher block from the previous step of encryption in the current one, which forms a chain-like encryption process. OFB operates on plain text in away similar to stream cipher that will be described below, where the encryption key used in every step depends on the encryption key from the previous step.

There are many other modes like CTR (counter), CFB (Cipher Feedback), or 3DES specific modes that are not discussed in this paper due to the fact that in this paper the main concentration will be on ECB and CBC modes.

3.7.1 ECB (Electronic Code Book)

This is the regular DES algorithm, exactly as described above. Data is divided into 64-bit blocks and each block is encrypted one at a time. Separate encryptions with different blocks are totally independent of each other. This means that if data is transmitted over a network or phone line, transmission errors will only affect the block containing the error.

It also means, however, that the blocks can be rearranged, thus scrambling a file beyond recognition, and this action would go undetected. ECB is the weakest of the various modes because no additional security measures are implemented besides the basic DES algorithm. However, ECB is the fastest and easiest to implement, making it the most common mode of DES seen in commercial applications. This is the mode of operation used by Private Encryptor.

3.7.2 CBC (Cipher Block Chaining)

In this mode of operation, each block of ECB encrypted ciphertext is XORed with the next plaintext block to be encrypted, thus making all the blocks dependent on all the previous blocks. This means that in order to find the plaintext of a particular block, you need to know the ciphertext, the key, and the ciphertext for the previous block. The first block to be encrypted has no previous ciphertext, so the plaintext is XORed with a 64-bit number called the Initialization Vector, or IV for short. So if data is transmitted over a network or phone line and there is a transmission error (adding or deleting bits), the error will be carried forward to all subsequent blocks since each block is dependent upon the last. If the bits are just modified in transit (as is the more common case) the error will only affect all of the bits in the changed block, and the corresponding bits in the following block. The error doesn't propagate any further.

This mode of operation is more secure than ECB because the extra XOR step adds one more layer to the encryption process.

3.7.3 CFB (Cipher Feedback)

In this mode, blocks of plaintext that are less than 64 bits long can be Encrypted.Normally, special processing has to be used to handle files whose size is not a perfect multiple of 8 bytes, but this mode removes that necessity (Private Encryptor handles this case by adding several dummy bytes to the end of a file before encrypting it). The plaintext itself is not actually passed through the DES algorithm, but merely XORed with an output block from it, in the following manner: A 64-bit block called the Shift Register is used as the input plaintext to DES. This is initially set to some arbitrary value, and encrypted with the DES algorithm. The ciphertext is then passed through an extra component called the M-box, which simply selects the left-most M bits of the ciphertext, where M is the number of bits in the block we wish to encrypt. This value is XORed with the real plaintext, and the output of that is the final ciphertext. Finally, the ciphertext is encrypted. As with CBC mode, an error in one block affects all subsequent blocks during data transmission. This mode of operation is similar to CBC and is very secure, but it is slower than ECB due to the added complexity.

3.7.4 OFB (Output Feedback)

This is similar to CFB mode, except that the ciphertext output of DES is fed back into the Shift Register, rather than the actual final ciphertext. The Shift Register is set to an arbitrary initial value, and passed through the DES algorithm. The output from DES is passed through the M-box and then fed back into the Shift Register to prepare for the next block. This value is then XORed with the real plaintext (which may be less than 64 bits in length, like CFB mode), and the result is the final ciphertext. Note that unlike CFB and CBC, a transmission error in one block will not affect subsequent blocks because once the recipient has the initial Shift Register value, it will continue to generate new Shift Register plaintext inputs without any further data input. However, this mode of operation is less secure than CFB mode because only the real ciphertext and DES ciphertext output is needed to find the plaintext of the most recent block. Knowledge of the key is not required.

CHAPTER 4

4.1 FPGA INTRODUCTION

A field programmable gate array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks, in FPGA parlance) also include memory elements, which may be simple flip-flops or more complete blocks of memories.

A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. These logic blocks and interconnects can be programmed after the manufacturing process by the customer/designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed.

Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. A typical integrated circuit performs a particular function defined at the time of manufacture. In contrast, the FPGA's function is defined by a program written by someone other than the device manufacturer. Depending on the particular device, the program is either 'burned' in permanently or semi-permanently as device is powered up. This user programmability gives the user access to complex integrated designs without the high engineering costs associated with application specific integrated circuits

4.2 HISTORY

The historical roots of FPGAs are in complex programmable logic devices (CPLDs) of the early to mid 1980s. Ross Freeman, Xilinx co-founder, invented the field programmable gate array in 1984. CPLDs and FPGAs include a relatively large number of programmable logic elements. CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several million.

The primary differences between CPLDs and FPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also far more complex to design for.

Another notable difference between CPLDs and FPGAs is the presence in most FPGAs of higher-level embedded functions (such as adders and multipliers) and embedded memories. A related, important difference is that many modern FPGAs support full or partial in-system reconfiguration, allowing their designs to be changed "on the fly" either for system upgrades or for dynamic reconfiguration as a normal part of system operation.

Some FPGAs have the capability of partial re-configuration that lets one portion of the device be re-programmed while other portions continue running.

4.3 ARCHITECTURE

The typical basic architecture consists of an array of configurable logic blocks (CLBs) and routing channels. Multiple I/O pads may fit into the height of one row or the width of one column in the array. Generally, all the routing channels have the same width (number of wires).

An application circuit must be mapped into an FPGA with adequate resources.

The typical FPGA logic block consists of a 4-input lookup table (LUT), and a flip-flop, as shown below.

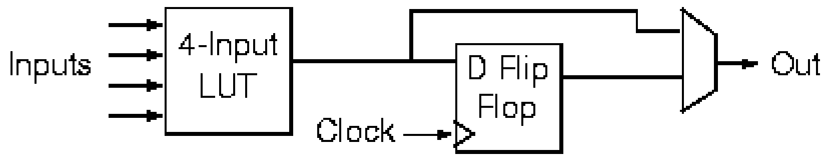


Fig. 4.1: FPGA Logic Block

There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they and other signals are separately managed.

For this example architecture, the locations of the FPGA logic block pins are shown below.

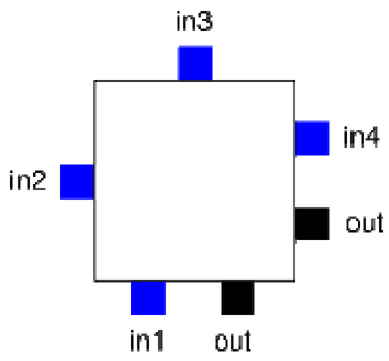


Fig. 4.2: Logic Block Pin Locations

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block.

Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O pad can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O pad at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect there is a switch box. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.

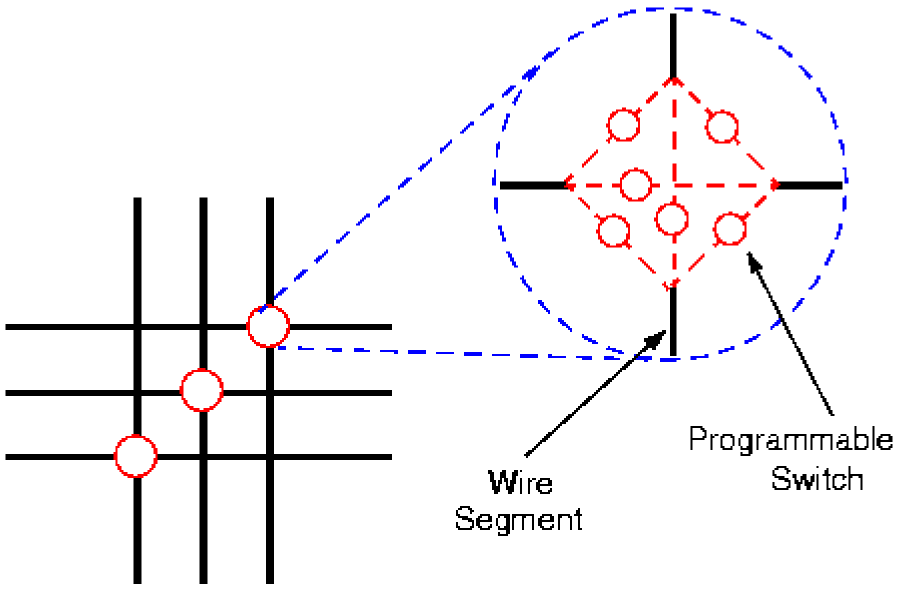


Fig. 4.3: Switch Box Topology

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

FPGAs are also widely used for systems validation including pre-silicon validation, post-silicon validation, and firmware development. This allows chip companies to validate their design before the chip is produced in the factory, reducing the time to market.

4.4 Basic Process Techonology Types

There are three basic approaches in providing programmability of FPGAs:

1. On chip control latches that are set with bit pattern to define the chip configuration. This type is called SRAM FPGA because the set of control latches can be considered as a static random access memory. These FPGAs are volatile that is the programming information is not preserved after the chip is powered down.
2. Antifuse programmed devices that are programmed electrically to provide connections that define the chip configuration. The programming is done by permanently closing some of the antifuse switches. Thus unlike static RAM FPGAs these devices cannot be reprogrammed. However these nonvolatile FPGAs are ffaster than the SRAM type devices. One important advantage of antifuses they are very small size allowing a large no. interconnections on a chip.
3. Using several electrically programmable devices(EPROMs and EEPROMs) and a shared interconnect mechanism on a single chip. In contrast to SRAM based FPGA EEPROM and EEPROM FPGAs technologies donot requie external permanent memory to preserve chip configuration. On the other hand they requie more complex chip fabrication process and use larger cells.

CHAPTER 5

5.1 Implementation

The DES algorithm in all used mainly following major components as desenc-The top module which structurally implements DES encryption.It comprises four components:

keysched- This is the key scheduling part.It includes two components pc1 and pc2.pc1 and pc2 both are permuting bits components.pc1 discards 8 bits from the key.pc2 also discards some bits to reduce the number of bits from 56 to 48.It generates the required keys at each of the sixteen stages.

IP- Performs initial permutation of the input bits before delivering to the the round function block.

roundfunc-Round function actually implements the DES algorithm by implementing all the logical operations and transformation needed.It is the structural design which connects the following components together

.xp

.s1,s2,s3,s4,s5,s6,s7,s8(s-boxes)

.desxor1

.pp

.desxor2

xp stands for expansion,since its behaviour is to expand the number of bitsfrom 38 to 48 bits desxor1 is a giant 48 bits xor gate which xors the sub key and the expanded input of the round function.The 8 s-boxes are the look-up table.pp is permutation ie. bits swapping.Finally another xor gate(desxor2) is responsible to xor the result of the permutation with the left part of the preceding round.

FP-Final permutation is the inverse of the initial permutation.

The only components that will use logical resources:

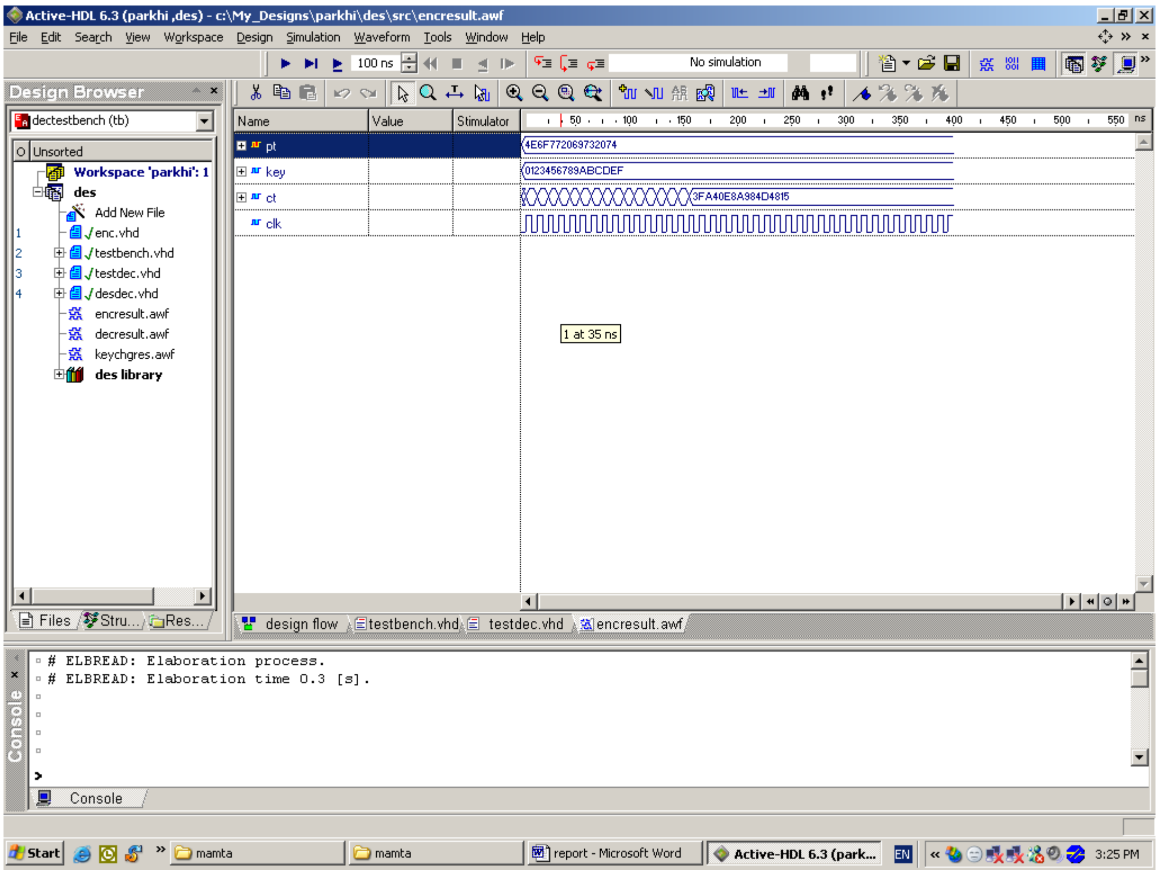
.desxor1

.desxor2

.s-boxes

5.2 Results

5.2.1 Output window showing encryption



Name	Value	Stimulator	Waveform
pt			4E6F772069732074
key			0123456789ABCDEF
ct			3FA40E8A984D4815
clk			

1 at 35 ns

```
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.3 [s].
>
```

DES test vector used for encryption is

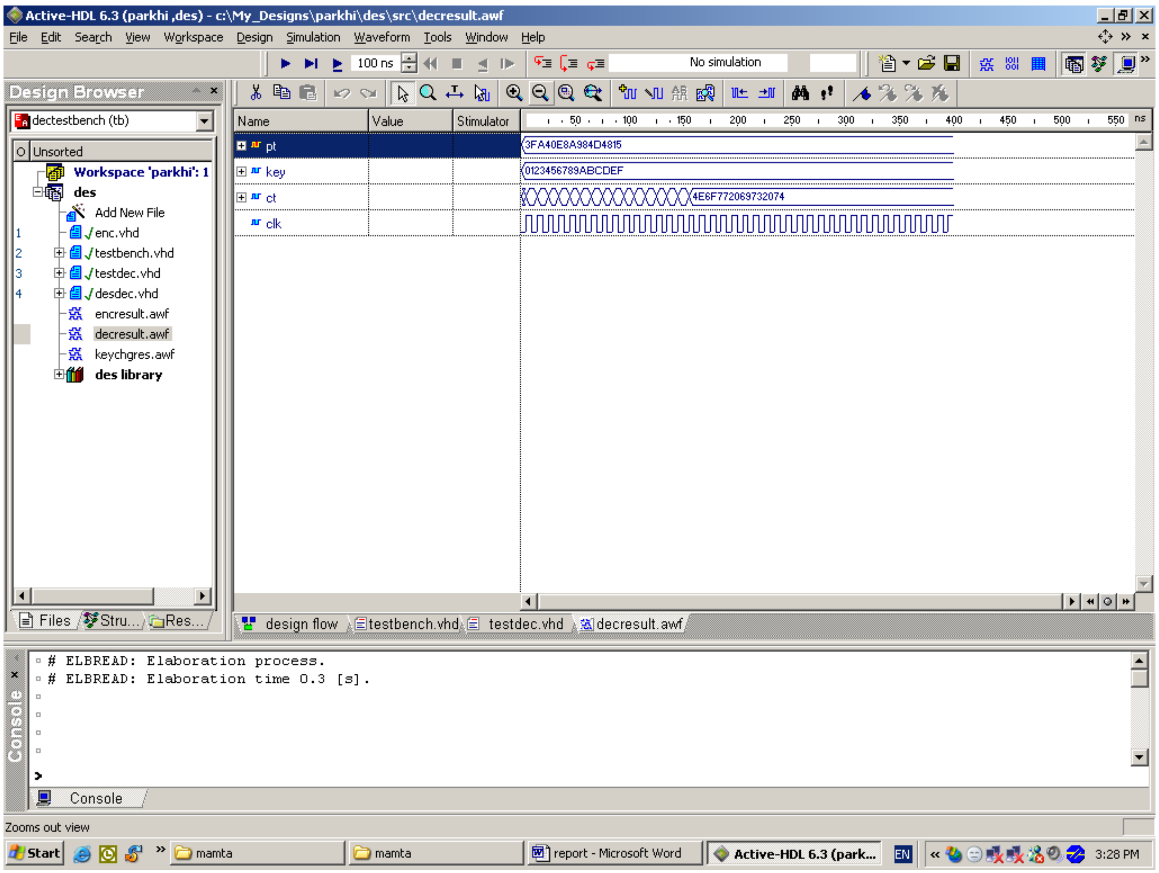
P=0100111001101111011101110010000001101001011100110010000001110100

K=0000000100100011010001010110011110001001101010111100110111101111

 The cipher text produce is

C=0011111110100100000011101000101010011000010011010100100000010101

5.2.2 Output window showing decryption



```
pt   3FA40E8A984D4815
key  0123456789ABCDEF
ct   4E6F772069732074
clk
```

```
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.3 [s].
```

DES test vector used for decryption is

C=0011111110100100000011101000101010011000010011010100100000010101

K=0000000100100011010001010110011110001001101010111100110111101111

The plain text produce is

P=0100111001101111011101110010000001101001011100110010000001110100

A desirable property of any encryption algorithm is that a small change in either the plain text or the key should produce a significant change in the cipher text.In particular, a change in one bit of the plain text or one bit of the key should produce a change in many bits of the cipher text.

DES exhibits a strong avalanche effect.

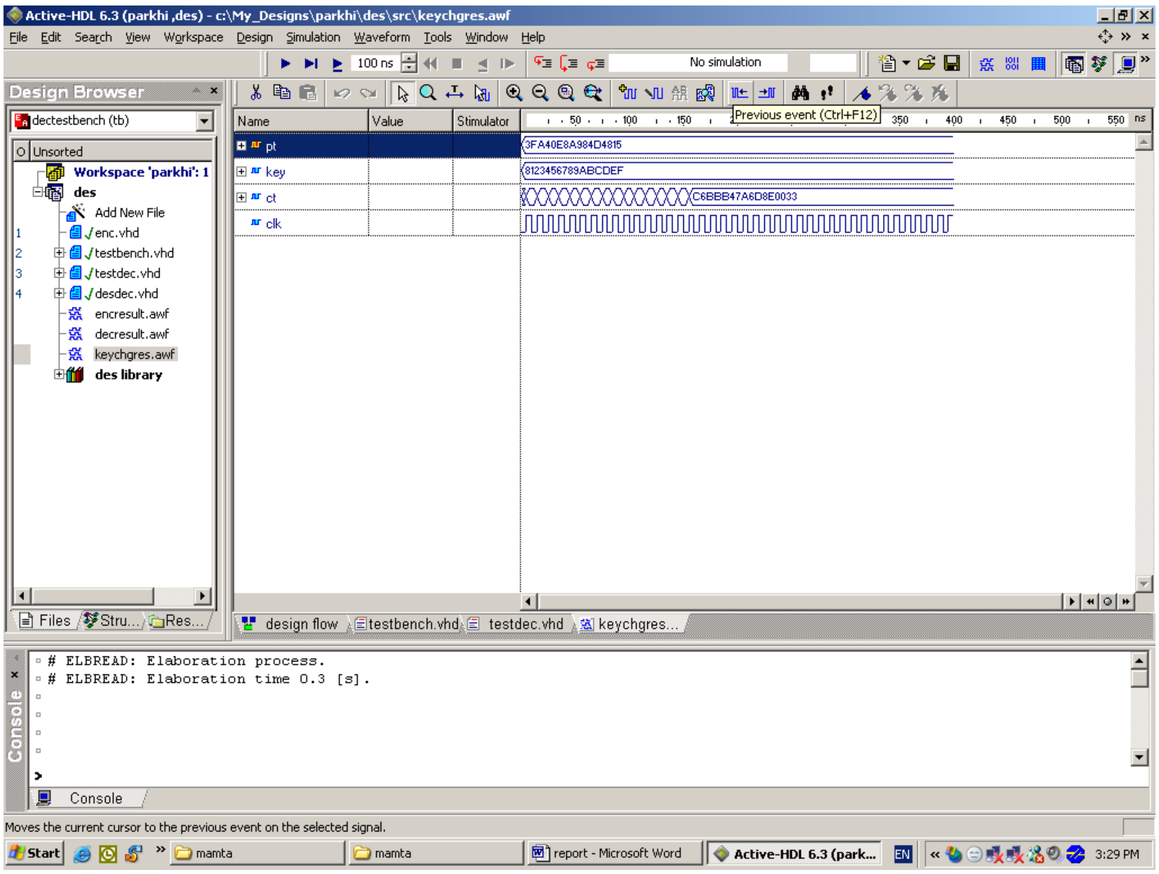
P=0100111001101111011101110010000001101001011100110010000001110100

K=0000000100100011010001010110011110001001101010111100110111101111

The cipher text produce is

C=1100011010111011101101000111101001101101100011100000000000110011

On changing a single bit of the key,the cipher text changes by 35 bits.

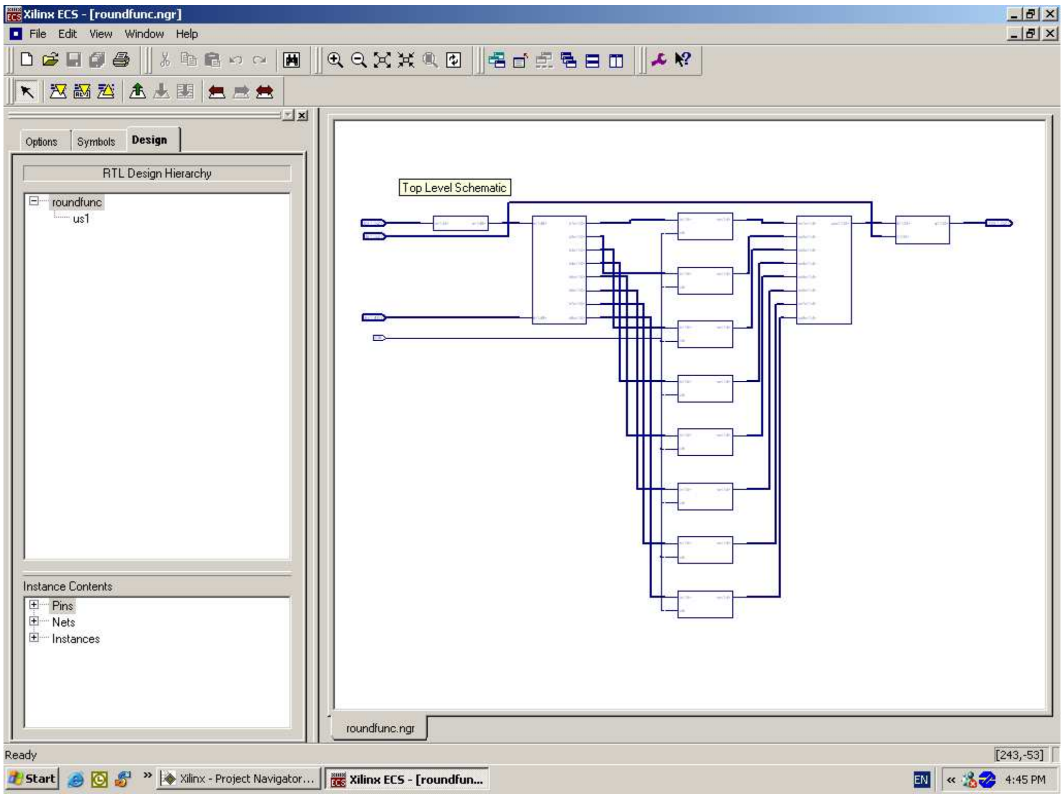

Active-HDL 6.3 (parkhi ,des) - c:\My_Designs\parkhi\des\src\keychgres.awf

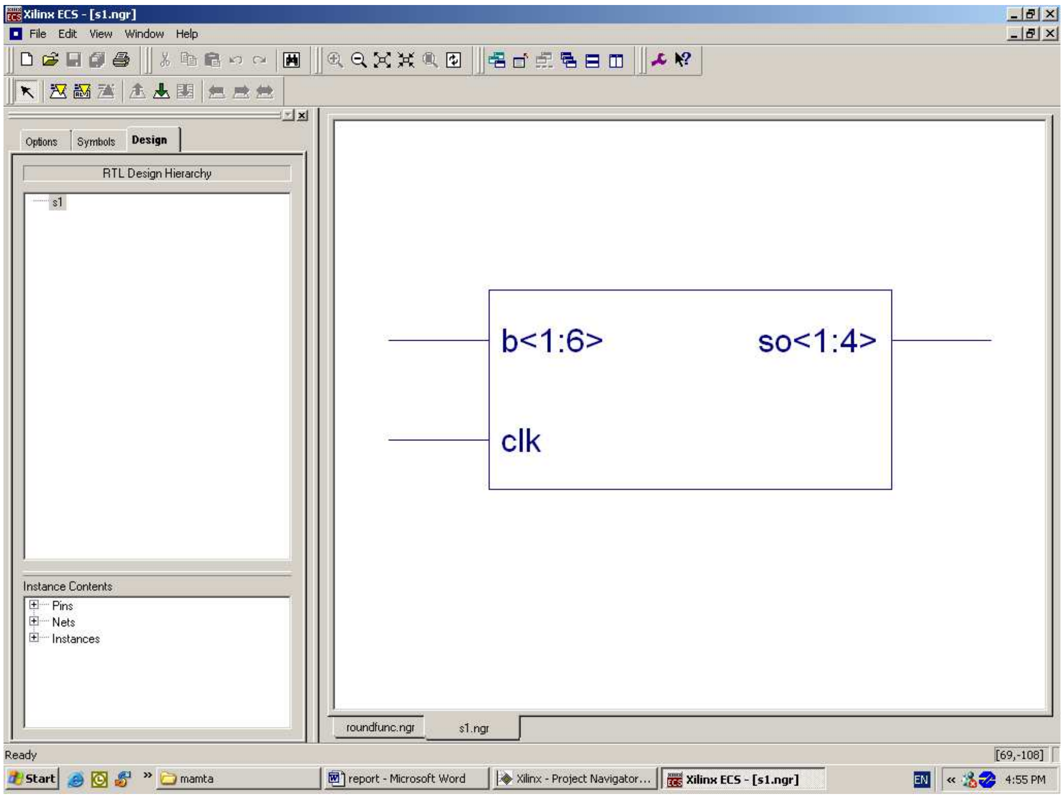
Name	Value	Stimulator	Waveform
pt			3FA40E8A984D4815
key			8123456789ABCDEF
ct			C6BBB47A6D8E0033
clk			

```
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.3 [s].
```

Moves the current cursor to the previous event on the selected signal.

Below is shown the RTL view of the complete roundfunc with all the other blocks that it uses.





5.3 Conclusion and Future Scope

To protect people’s privacy, cryptography technology is becoming more and more important in the communication area. The rapid progress of VLSI technology benefits the hardware realization of encryption and decryption a lot, making the devices smaller, faster, and less power-consuming.In this thesis, the complete synthesizable unit of Data Encryption Standard is designed.It is functionally simulated and synthesized showing the corresponding RTL views.

DES is not in use where high level of security is required, it can be used in the form of Triple DES and can also be replaced by stronger algorithm like AES.But it is still widely used if a high level of security is not required

The original description of DES is not optimized for FPGA implementation regarding the speed performance and the number of LUTs used. In the the future,all the implementations can be optimized further to their optimizing goals.Various implementations of DES can be integrated to the real application environment to test all the parameters.

REFERENCES

[1] Gael Rouvroy, Francois-Xaviet Standaert, Jean-Jacques Quisquater, Jean-Didier Legat, Efficient Uses of FPGAs for Implementations of DES and Its Experimental Linear Cryptanalysis, IEEE Transactions on Computers, Vol. 52, April 2003.

[2] K.Wong, M.Wark, E.Dawson, A single chip FPGA implementation of the Data Encryption Standard Algorithm, 1998 IEEE.

[3] Seung-Jo Han, Heang-Soo Oh, Jongan Park, The improved Data Encryption Standard Algorithm, 1996 IEEE.

[4] Touria Arich, Mohssine Eleuldj, Hardware Implementations of the Data Encryption Standard, 2002 IEEE.

[5] Ibrahim E.Ziedan, Mohammed M.Fouad, Doaa H. Salem, Application of Data Encryption Standard to Bitmap and JPEG Images, Twenteith National Radio Science Conference, 2003.

[6] A.Menezes, P.Van Ooschot, S.Vanstone, Handbook of Cryptography, CRC Press, 1996.

[7] William Stallings, Cryptography and Network Security, Pearson Education, 2003.

[8] Andrew S.Tanenbaum, Computer Networks, Printice Hall, 2007.

[9] B.Schneier, Applied Cryptography, John Wiley & Sons, 1996.

[10] http:www.ciphersbyritter, com/LEARNING.HTM

APPENDIX

Code for data encryption standard algorithm

```
library ieee;

use ieee.std_logic_1164.all;

entity pc1 is port

( key : in std_logic_vector(1 TO 64);

c0x,d0x : out std_logic_vector(1 TO 28));

end pc1;

architecture behavior of pc1 is

signal XX : std_logic_vector(1 to 56);

begin

XX(1)<=key(57); XX(2)<=key(49); XX(3)<=key(41); XX(4)<=key(33);

XX(5)<=key(25); XX(6)<=key(17); XX(7)<=key(9);

XX(8)<=key(1); XX(9)<=key(58); XX(10)<=key(50); XX(11)<=key(42);

XX(12)<=key(34); XX(13)<=key(26); XX(14)<=key(18);

XX(15)<=key(10); XX(16)<=key(2); XX(17)<=key(59); XX(18)<=key(51);

XX(19)<=key(43); XX(20)<=key(35); XX(21)<=key(27);

XX(22)<=key(19); XX(23)<=key(11); XX(24)<=key(3); XX(25)<=key(60);

XX(26)<=key(52); XX(27)<=key(44); XX(28)<=key(36);

XX(29)<=key(63); XX(30)<=key(55); XX(31)<=key(47); XX(32)<=key(39);

XX(33)<=key(31); XX(34)<=key(23); XX(35)<=key(15);

XX(36)<=key(7); XX(37)<=key(62); XX(38)<=key(54); XX(39)<=key(46);

XX(40)<=key(38); XX(41)<=key(30); XX(42)<=key(22);
XX(43)<=key(14); XX(44)<=key(6); XX(45)<=key(61); XX(46)<=key(53);
XX(47)<=key(45); XX(48)<=key(37); XX(49)<=key(29);
XX(50)<=key(21); XX(51)<=key(13); XX(52)<=key(5); XX(53)<=key(28);
XX(54)<=key(20); XX(55)<=key(12); XX(56)<=key(4);
c0x<=XX(1 to 28); d0x<=XX(29 to 56);
end behavior;


LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity pc2 is port
(c,d : in std_logic_vector(1 TO 28);
k : out std_logic_vector(1 TO 48));
end pc2;
architecture behavior of pc2 is
signal YY : std_logic_vector(1 to 56);
begin
YY(1 to 28)<=c; YY(29 to 56)<=d;
k(1)<=YY(14); k(2)<=YY(17); k(3)<=YY(11); k(4)<=YY(24); k(5)<=YY(1);
k(6)<=YY(5);
k(7)<=YY(3); k(8)<=YY(28); k(9)<=YY(15); k(10)<=YY(6); k(11)<=YY(21);
k(12)<=YY(10);
k(13)<=YY(23); k(14)<=YY(19); k(15)<=YY(12); k(16)<=YY(4); k(17)<=YY(26);
k(18)<=YY(8);
k(19)<=YY(16); k(20)<=YY(7); k(21)<=YY(27); k(22)<=YY(20); k(23)<=YY(13);
k(24)<=YY(2);
```

```
k(25)<=YY(41); k(26)<=YY(52); k(27)<=YY(31); k(28)<=YY(37); k(29)<=YY(47);
k(30)<=YY(55);
k(31)<=YY(30); k(32)<=YY(40); k(33)<=YY(51); k(34)<=YY(45); k(35)<=YY(33);
k(36)<=YY(48);
k(37)<=YY(44); k(38)<=YY(49); k(39)<=YY(39); k(40)<=YY(56); k(41)<=YY(34);
k(42)<=YY(53);
k(43)<=YY(46); k(44)<=YY(42); k(45)<=YY(50); k(46)<=YY(36); k(47)<=YY(29);
k(48)<=YY(32);
end behavior;

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity keysched is port
(key : in std_logic_vector(1 to 64);
k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
: out std_logic_vector(1 to 48));
end keysched;
architecture behaviour of keysched is
COMPONENT pc1 port
(key : in std_logic_vector(1 TO 64);
c0x,d0x : out std_logic_vector(1 TO 28));
end COMPONENT;
COMPONENT pc2 port
(c,d : in std_logic_vector(1 TO 28);
k : out std_logic_vector(1 TO 48));
```

```
end COMPONENT;
signal
c0x,c1x,c2x,c3x,c4x,c5x,c6x,c7x,c8x,c9x,c10x,c11x,c12x,c13x,c14x,c15x,c16x :
std_logic_vector(1 to 28);
signal
d0x,d1x,d2x,d3x,d4x,d5x,d6x,d7x,d8x,d9x,d10x,d11x,d12x,d13x,d14x,d15x,d16x :
std_logic_vector(1 to 28);
begin
upc1: pc1 port map ( key=>key, c0x=>c0x, d0x=>d0x );
c1x <= c0x(2 to 28) & c0x(1); d1x <= d0x(2 to 28) & d0x(1);
c2x <= c1x(2 to 28) & c1x(1); d2x <= d1x(2 to 28) & d1x(1);
c3x <= c2x(3 to 28) & c2x(1 to 2); d3x <= d2x(3 to 28) & d2x(1 to 2);
c4x <= c3x(3 to 28) & c3x(1 to 2); d4x <= d3x(3 to 28) & d3x(1 to 2);
c5x <= c4x(3 to 28) & c4x(1 to 2); d5x <= d4x(3 to 28) & d4x(1 to 2);
c6x <= c5x(3 to 28) & c5x(1 to 2); d6x <= d5x(3 to 28) & d5x(1 to 2);
c7x <= c6x(3 to 28) & c6x(1 to 2); d7x <= d6x(3 to 28) & d6x(1 to 2);
c8x <= c7x(3 to 28) & c7x(1 to 2); d8x <= d7x(3 to 28) & d7x(1 to 2);
c9x <= c8x(2 to 28) & c8x(1); d9x <= d8x(2 to 28) & d8x(1);
c10x <= c9x(3 to 28) & c9x(1 to 2); d10x <= d9x(3 to 28) & d9x(1 to 2);
c11x <= c10x(3 to 28) & c10x(1 to 2); d11x <= d10x(3 to 28) & d10x(1 to 2);
c12x <= c11x(3 to 28) & c11x(1 to 2); d12x <= d11x(3 to 28) & d11x(1 to 2);
c13x <= c12x(3 to 28) & c12x(1 to 2); d13x <= d12x(3 to 28) & d12x(1 to 2);
c14x <= c13x(3 to 28) & c13x(1 to 2); d14x <= d13x(3 to 28) & d13x(1 to 2);
c15x <= c14x(3 to 28) & c14x(1 to 2); d15x <= d14x(3 to 28) & d14x(1 to 2);
c16x <= c15x(2 to 28) & c15x(1); d16x <= d15x(2 to 28) & d15x(1);
```

```
pc2x1: pc2 port map ( c=>c1x, d=>d1x, k=>k1x );

pc2x2: pc2 port map ( c=>c2x, d=>d2x, k=>k2x );

pc2x3: pc2 port map ( c=>c3x, d=>d3x, k=>k3x );

pc2x4: pc2 port map ( c=>c4x, d=>d4x, k=>k4x );

pc2x5: pc2 port map ( c=>c5x, d=>d5x, k=>k5x );

pc2x6: pc2 port map ( c=>c6x, d=>d6x, k=>k6x );

pc2x7: pc2 port map ( c=>c7x, d=>d7x, k=>k7x );

pc2x8: pc2 port map ( c=>c8x, d=>d8x, k=>k8x );

pc2x9: pc2 port map ( c=>c9x, d=>d9x, k=>k9x );

pc2x10: pc2 port map ( c=>c10x, d=>d10x, k=>k10x );

pc2x11: pc2 port map ( c=>c11x, d=>d11x, k=>k11x );

pc2x12: pc2 port map ( c=>c12x, d=>d12x, k=>k12x );

pc2x13: pc2 port map ( c=>c13x, d=>d13x, k=>k13x );

pc2x14: pc2 port map ( c=>c14x, d=>d14x, k=>k14x );

pc2x15: pc2 port map ( c=>c15x, d=>d15x, k=>k15x );

pc2x16: pc2 port map ( c=>c16x, d=>d16x, k=>k16x );

end;


library ieee;

use ieee.std_logic_1164.all;

entity ip is port

(pt : in std_logic_vector(1 TO 64);

l0x : out std_logic_vector(1 TO 32);

r0x : out std_logic_vector(1 TO 32));

end ip;
```

```
architecture behavior of ip is
begin
l0x(1)<=pt(58); l0x(2)<=pt(50); l0x(3)<=pt(42); l0x(4)<=pt(34);
l0x(5)<=pt(26); l0x(6)<=pt(18); l0x(7)<=pt(10); l0x(8)<=pt(2);
l0x(9)<=pt(60); l0x(10)<=pt(52); l0x(11)<=pt(44); l0x(12)<=pt(36);
l0x(13)<=pt(28); l0x(14)<=pt(20); l0x(15)<=pt(12); l0x(16)<=pt(4);
l0x(17)<=pt(62); l0x(18)<=pt(54); l0x(19)<=pt(46); l0x(20)<=pt(38);
l0x(21)<=pt(30); l0x(22)<=pt(22); l0x(23)<=pt(14); l0x(24)<=pt(6);
l0x(25)<=pt(64); l0x(26)<=pt(56); l0x(27)<=pt(48); l0x(28)<=pt(40);
l0x(29)<=pt(32); l0x(30)<=pt(24); l0x(31)<=pt(16); l0x(32)<=pt(8);
r0x(1)<=pt(57); r0x(2)<=pt(49); r0x(3)<=pt(41); r0x(4)<=pt(33);
r0x(5)<=pt(25); r0x(6)<=pt(17); r0x(7)<=pt(9); r0x(8)<=pt(1);
r0x(9)<=pt(59); r0x(10)<=pt(51); r0x(11)<=pt(43); r0x(12)<=pt(35);
r0x(13)<=pt(27); r0x(14)<=pt(19); r0x(15)<=pt(11); r0x(16)<=pt(3);
r0x(17)<=pt(61); r0x(18)<=pt(53); r0x(19)<=pt(45); r0x(20)<=pt(37);
r0x(21)<=pt(29); r0x(22)<=pt(21); r0x(23)<=pt(13); r0x(24)<=pt(5);
r0x(25)<=pt(63); r0x(26)<=pt(55); r0x(27)<=pt(47); r0x(28)<=pt(39);
r0x(29)<=pt(31); r0x(30)<=pt(23); r0x(31)<=pt(15); r0x(32)<=pt(7);
end behavior;


library ieee;
use ieee.std_logic_1164.all;
entity xp is port
(ri : in std_logic_vector(1 TO 32);
e : out std_logic_vector(1 TO 48));
```

```
end xp;

architecture behavior of xp is

begin

e(1)<=ri(32); e(2)<=ri(1); e(3)<=ri(2); e(4)<=ri(3); e(5)<=ri(4); e(6)<=ri(5);

e(7)<=ri(4); e(8)<=ri(5);e(9)<=ri(6); e(10)<=ri(7); e(11)<=ri(8); e(12)<=ri(9);

e(13)<=ri(8); e(14)<=ri(9); e(15)<=ri(10); e(16)<=ri(11);e(17)<=ri(12);

e(18)<=ri(13); e(19)<=ri(12); e(20)<=ri(13); e(21)<=ri(14); e(22)<=ri(15);

e(23)<=ri(16); e(24)<=ri(17);

e(25)<=ri(16); e(26)<=ri(17); e(27)<=ri(18); e(28)<=ri(19); e(29)<=ri(20);

e(30)<=ri(21); e(31)<=ri(20); e(32)<=ri(21);e(33)<=ri(22); e(34)<=ri(23);

e(35)<=ri(24); e(36)<=ri(25); e(37)<=ri(24); e(38)<=ri(25); e(39)<=ri(26);

e(40)<=ri(27);

e(41)<=ri(28); e(42)<=ri(29); e(43)<=ri(28); e(44)<=ri(29); e(45)<=ri(30);

e(46)<=ri(31); e(47)<=ri(32); e(48)<=ri(1);

end behavior;


library ieee;

use ieee.std_logic_1164.all;

entity desxor1 is port

(e : in std_logic_vector(1 TO 48);

b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x

: out std_logic_vector (1 TO 6);

k : in std_logic_vector (1 TO 48));

 end desxor1;

architecture behavior of desxor1 is
```

```
signal XX : std_logic_vector( 1 to 48);
begin
XX<=k xor e;
b1x<=XX(1 to 6);b2x<=XX(7 to 12);b3x<=XX(13 to 18);b4x<=XX(19 to 24);
b5x<=XX(25 to 30); b6x<=XX(31 to 36);b7x<=XX(37 to 42);b8x<=XX(43 to 48);
end behavior;


library ieee;
use ieee.std_logic_1164.all;
entity s1 is port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end s1;
architecture behaviour of s1 is
begin
process(clk)
begin
if(clk'event and clk='1') then
case b is
when b"000000"=> so<=x"e";
when b"000010"=> so<=x"4";
when b"000100"=> so<=x"d";
when b"000110"=> so<=x"1";
when b"001000"=> so<=x"2";
when b"001010"=> so<=x"f";
when b"001100"=> so<=x"b";
when b"001110"=> so<=x"8";
when b"010000"=> so<=x"3";
when b"010010"=> so<=x"a";
when b"010100"=> so<=x"6";
when b"010110"=> so<=x"c";
when b"011000"=> so<=x"5";
when b"011010"=> so<=x"9";
when b"011100"=> so<=x"0";
when b"011110"=> so<=x"7";
when b"000001"=> so<=x"0";
when b"000011"=> so<=x"f";
when b"000101"=> so<=x"7";
when b"000111"=> so<=x"4";
when b"001001"=> so<=x"e";
when b"001011"=> so<=x"2";
when b"001101"=> so<=x"d";
when b"001111"=> so<=x"1";
when b"010001"=> so<=x"a";
when b"010011"=> so<=x"6";
when b"010101"=> so<=x"c";
when b"010111"=> so<=x"b";
when b"011001"=> so<=x"9";
when b"011011"=> so<=x"5";
when b"011101"=> so<=x"3";

when b"011111"=> so<=x"8";

when b"100000"=> so<=x"4";

when b"100010"=> so<=x"1";

when b"100100"=> so<=x"e";

when b"100110"=> so<=x"8";

when b"101000"=> so<=x"d";

when b"101010"=> so<=x"6";

when b"101100"=> so<=x"2";

when b"101110"=> so<=x"b";

when b"110000"=> so<=x"f";

when b"110010"=> so<=x"c";

when b"110100"=> so<=x"9";

when b"110110"=> so<=x"7";

when b"111000"=> so<=x"3";

when b"111010"=> so<=x"a";

when b"111100"=> so<=x"5";

when b"111110"=> so<=x"0";

when b"100001"=> so<=x"f";

when b"100011"=> so<=x"c";

when b"100101"=> so<=x"8";

when b"100111"=> so<=x"2";

when b"101001"=> so<=x"4";

when b"101011"=> so<=x"9";

when b"101101"=> so<=x"1";
when b"101111"=> so<=x"7";

when b"110001"=> so<=x"5";

when b"110011"=> so<=x"b";

when b"110101"=> so<=x"3";

when b"110111"=> so<=x"e";

when b"111001"=> so<=x"a";

when b"111011"=> so<=x"0";

when b"111101"=> so<=x"6";

when others=> so<=x"d";

end case;

end if;

end process;

end;


LIBRARY ieee ;

use ieee.std_logic_1164.all;

entity s2 is port

(clk : in std_logic;

b : in std_logic_vector(1 to 6);

so : out std_logic_vector(1 to 4));

end s2;

architecture behaviour of s2 is

begin

process(clk)

begin
```

```
if(clk'event and clk='1') then

case b is

when b"000000"=> so<=x"f";

when b"000010"=> so<=x"1";

when b"000100"=> so<=x"8";

when b"000110"=> so<=x"e";

when b"001000"=> so<=x"6";

when b"001010"=> so<=x"b";

when b"001100"=> so<=x"3";

when b"001110"=> so<=x"4";

when b"010000"=> so<=x"9";

when b"010010"=> so<=x"7";

when b"010100"=> so<=x"2";

when b"010110"=> so<=x"d";

when b"011000"=> so<=x"c";

when b"011010"=> so<=x"0";

when b"011100"=> so<=x"5";

when b"011110"=> so<=x"a";

when b"000001"=> so<=x"3";

when b"000011"=> so<=x"d";

when b"000101"=> so<=x"4";

when b"000111"=> so<=x"7";

when b"001001"=> so<=x"f";

when b"001011"=> so<=x"2";

when b"001101"=> so<=x"8";
when b"001111"=> so<=x"e";
when b"010001"=> so<=x"c";
when b"010011"=> so<=x"0";
when b"010101"=> so<=x"1";
when b"010111"=> so<=x"a";
when b"011001"=> so<=x"6";
when b"011011"=> so<=x"9";
when b"011101"=> so<=x"b";
when b"011111"=> so<=x"5";
when b"100000"=> so<=x"0";
when b"100010"=> so<=x"e";
when b"100100"=> so<=x"7";
when b"100110"=> so<=x"b";
when b"101000"=> so<=x"a";
when b"101010"=> so<=x"4";
when b"101100"=> so<=x"d";
when b"101110"=> so<=x"1";
when b"110000"=> so<=x"5";
when b"110010"=> so<=x"8";
when b"110100"=> so<=x"c";
when b"110110"=> so<=x"6";
when b"111000"=> so<=x"9";
when b"111010"=> so<=x"3";
when b"111100"=> so<=x"2";
when b"111110"=> so<=x"f";
when b"100001"=> so<=x"d";
when b"100011"=> so<=x"8";
when b"100101"=> so<=x"a";
when b"100111"=> so<=x"1";
when b"101001"=> so<=x"3";
when b"101011"=> so<=x"f";
when b"101101"=> so<=x"4";
when b"101111"=> so<=x"2";
when b"110001"=> so<=x"b";
when b"110011"=> so<=x"6";
when b"110101"=> so<=x"7";
when b"110111"=> so<=x"c";
when b"111001"=> so<=x"0";
when b"111011"=> so<=x"5";
when b"111101"=> so<=x"e";
when others=> so<=x"9";
end case;
end if;
end process;
end;

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity s3 is port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);

so : out std_logic_vector(1 to 4));

end s3;

architecture behaviour of s3 is

begin

process(clk)

begin

if(clk'event and clk='1') then

case b is

when b"000000"=> so<=x"a";

when b"000010"=> so<=x"0";

when b"000100"=> so<=x"9";

when b"000110"=> so<=x"e";

when b"001000"=> so<=x"6";

when b"001010"=> so<=x"3";

when b"001100"=> so<=x"f";

when b"001110"=> so<=x"5";

when b"010000"=> so<=x"1";

when b"010010"=> so<=x"d";

when b"010100"=> so<=x"c";

when b"010110"=> so<=x"7";

when b"011000"=> so<=x"b";

when b"011010"=> so<=x"4";

when b"011100"=> so<=x"2";

when b"011110"=> so<=x"8";
when b"000001"=> so<=x"d";
when b"000011"=> so<=x"7";
when b"000101"=> so<=x"0";
when b"000111"=> so<=x"9";
when b"001001"=> so<=x"3";
when b"001011"=> so<=x"4";
when b"001101"=> so<=x"6";
when b"001111"=> so<=x"a";
when b"010001"=> so<=x"2";
when b"010011"=> so<=x"8";
when b"010101"=> so<=x"5";
when b"010111"=> so<=x"e";
when b"011001"=> so<=x"c";
when b"011011"=> so<=x"b";
when b"011101"=> so<=x"f";
when b"011111"=> so<=x"1";
when b"100000"=> so<=x"d";
when b"100010"=> so<=x"6";
when b"100100"=> so<=x"4";
when b"100110"=> so<=x"9";
when b"101000"=> so<=x"8";
when b"101010"=> so<=x"f";
when b"101100"=> so<=x"3";
when b"101110"=> so<=x"0";
when b"110000"=> so<=x"b";
when b"110010"=> so<=x"1";

when b"110100"=> so<=x"2";

when b"110110"=> so<=x"c";

when b"111000"=> so<=x"5";

when b"111010"=> so<=x"a";

when b"111100"=> so<=x"e";

when b"111110"=> so<=x"7";

when b"100001"=> so<=x"1";

when b"100011"=> so<=x"a";

when b"100101"=> so<=x"d";

when b"100111"=> so<=x"0";

when b"101001"=> so<=x"6";

when b"101011"=> so<=x"9";

when b"101101"=> so<=x"8";

when b"101111"=> so<=x"7";

when b"110001"=> so<=x"4";

when b"110011"=> so<=x"f";

when b"110101"=> so<=x"e";

when b"110111"=> so<=x"3";

when b"111001"=> so<=x"b";

when b"111011"=> so<=x"5";

when b"111101"=> so<=x"2";

when others=> so<=x"c";

end case;

end if;
end process;
end;

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity s4 is port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end s4;
architecture behaviour of s4 is
begin
process(clk)
begin
if(clk'event and clk='1') then
case b is
when b"000000"=> so<=x"7";
when b"000010"=> so<=x"d";
when b"000100"=> so<=x"e";
when b"000110"=> so<=x"3";
when b"001000"=> so<=x"0";
when b"001010"=> so<=x"6";
when b"001100"=> so<=x"9";
when b"001110"=> so<=x"a";
when b"010000"=> so<=x"1";
when b"010010"=> so<=x"2";

when b"010100"=> so<=x"8";

when b"010110"=> so<=x"5";

when b"011000"=> so<=x"b";

when b"011010"=> so<=x"c";

when b"011100"=> so<=x"4";

when b"011110"=> so<=x"f";

when b"000001"=> so<=x"d";

when b"000011"=> so<=x"8";

when b"000101"=> so<=x"b";

when b"000111"=> so<=x"5";

when b"001001"=> so<=x"6";

when b"001011"=> so<=x"f";

when b"001101"=> so<=x"0";

when b"001111"=> so<=x"3";

when b"010001"=> so<=x"4";

when b"010011"=> so<=x"7";

when b"010101"=> so<=x"2";

when b"010111"=> so<=x"c";

when b"011001"=> so<=x"1";

when b"011011"=> so<=x"a";

when b"011101"=> so<=x"e";

when b"011111"=> so<=x"9";

when b"100000"=> so<=x"a";

when b"100010"=> so<=x"6";
when b"100100"=> so<=x"9";
when b"100110"=> so<=x"0";
when b"101000"=> so<=x"c";
when b"101010"=> so<=x"b";
when b"101100"=> so<=x"7";
when b"101110"=> so<=x"d";
when b"110000"=> so<=x"f";
when b"110010"=> so<=x"1";
when b"110100"=> so<=x"3";
when b"110110"=> so<=x"e";
when b"111000"=> so<=x"5";
when b"111010"=> so<=x"2";
when b"111100"=> so<=x"8";
when b"111110"=> so<=x"4";
when b"100001"=> so<=x"3";
when b"100011"=> so<=x"f";
when b"100101"=> so<=x"0";
when b"100111"=> so<=x"6";
when b"101001"=> so<=x"a";
when b"101011"=> so<=x"1";
when b"101101"=> so<=x"d";
when b"101111"=> so<=x"8";
when b"110001"=> so<=x"9";
when b"110011"=> so<=x"4";
when b"110101"=> so<=x"5";
when b"110111"=> so<=x"b";

when b"111001"=> so<=x"c";

when b"111011"=> so<=x"7";

when b"111101"=> so<=x"2";

when others=> so<=x"e";

end case;

end if;

end process;

end;


LIBRARY ieee ;

use ieee.std_logic_1164.all;

entity s5 is port

(clk : in std_logic;

b : in std_logic_vector(1 to 6);

so : out std_logic_vector(1 to 4));

end s5;

architecture behaviour of s5 is

begin

process(clk)

begin

if(clk'event and clk='1') then

case b is

when b"000000"=> so<=x"2";

when b"000010"=> so<=x"c";
when b"000100"=> so<=x"4";
when b"000110"=> so<=x"1";
when b"001000"=> so<=x"7";
when b"001010"=> so<=x"a";
when b"001100"=> so<=x"b";
when b"001110"=> so<=x"6";
when b"010000"=> so<=x"8";
when b"010010"=> so<=x"5";
when b"010100"=> so<=x"3";
when b"010110"=> so<=x"f";
when b"011000"=> so<=x"d";
when b"011010"=> so<=x"0";
when b"011100"=> so<=x"e";
when b"011110"=> so<=x"9";
when b"000001"=> so<=x"e";
when b"000011"=> so<=x"b";
when b"000101"=> so<=x"2";
when b"000111"=> so<=x"c";
when b"001001"=> so<=x"4";
when b"001011"=> so<=x"7";
when b"001101"=> so<=x"d";
when b"001111"=> so<=x"1";
when b"010001"=> so<=x"5";
when b"010011"=> so<=x"0";
when b"010101"=> so<=x"f";
when b"010111"=> so<=x"a";
when b"011001"=> so<=x"3";
when b"011011"=> so<=x"9";
when b"011101"=> so<=x"8";
when b"011111"=> so<=x"6";
when b"100000"=> so<=x"4";
when b"100010"=> so<=x"2";
when b"100100"=> so<=x"1";
when b"100110"=> so<=x"b";
when b"101000"=> so<=x"a";
when b"101010"=> so<=x"d";
when b"101100"=> so<=x"7";
when b"101110"=> so<=x"8";
when b"110000"=> so<=x"f";
when b"110010"=> so<=x"9";
when b"110100"=> so<=x"c";
when b"110110"=> so<=x"5";
when b"111000"=> so<=x"6";
when b"111010"=> so<=x"3";
when b"111100"=> so<=x"0";
when b"111110"=> so<=x"e";
when b"100001"=> so<=x"b";
when b"100011"=> so<=x"8";
when b"100101"=> so<=x"c";
when b"100111"=> so<=x"7";
when b"101001"=> so<=x"1";
when b"101011"=> so<=x"e";
when b"101101"=> so<=x"2";
when b"101111"=> so<=x"d";
when b"110001"=> so<=x"6";
when b"110011"=> so<=x"f";
when b"110101"=> so<=x"0";
when b"110111"=> so<=x"9";
when b"111001"=> so<=x"a";
when b"111011"=> so<=x"4";
when b"111101"=> so<=x"5";
when others=> so<=x"3";
end case;
end if;
end process;
end;


LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity s6 is port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end s6;
architecture behaviour of s6 is
```

```
begin
process(clk)
begin
if(clk'event and clk='1') then
case b is
when b"000000"=> so<=x"c";
when b"000010"=> so<=x"1";
when b"000100"=> so<=x"a";
when b"000110"=> so<=x"f";
when b"001000"=> so<=x"9";
when b"001010"=> so<=x"2";
when b"001100"=> so<=x"6";
when b"001110"=> so<=x"8";
when b"010000"=> so<=x"0";
when b"010010"=> so<=x"d";
when b"010100"=> so<=x"3";
when b"010110"=> so<=x"4";
when b"011000"=> so<=x"e";
when b"011010"=> so<=x"7";
when b"011100"=> so<=x"5";
when b"011110"=> so<=x"b";
when b"000001"=> so<=x"a";
when b"000011"=> so<=x"f";
when b"000101"=> so<=x"4";
when b"000111"=> so<=x"2";
when b"001001"=> so<=x"7";
when b"001011"=> so<=x"c";
when b"001101"=> so<=x"9";
when b"001111"=> so<=x"5";
when b"010001"=> so<=x"6";
when b"010011"=> so<=x"1";
when b"010101"=> so<=x"d";
when b"010111"=> so<=x"e";
when b"011001"=> so<=x"0";
when b"011011"=> so<=x"b";
when b"011101"=> so<=x"3";
when b"011111"=> so<=x"8";
when b"100000"=> so<=x"9";
when b"100010"=> so<=x"e";
when b"100100"=> so<=x"f";
when b"100110"=> so<=x"5";
when b"101000"=> so<=x"2";
when b"101010"=> so<=x"8";
when b"101100"=> so<=x"c";
when b"101110"=> so<=x"3";
when b"110000"=> so<=x"7";
when b"110010"=> so<=x"0";
when b"110100"=> so<=x"4";
when b"110110"=> so<=x"a";
when b"111000"=> so<=x"1";
when b"111010"=> so<=x"d";

when b"111100"=> so<=x"b";

when b"111110"=> so<=x"6";

when b"100001"=> so<=x"4";

when b"100011"=> so<=x"3";

when b"100101"=> so<=x"2";

when b"100111"=> so<=x"c";

when b"101001"=> so<=x"9";

when b"101011"=> so<=x"5";

when b"101101"=> so<=x"f";

when b"101111"=> so<=x"a";

when b"110001"=> so<=x"b";

when b"110011"=> so<=x"e";

when b"110101"=> so<=x"1";

when b"110111"=> so<=x"7";

when b"111001"=> so<=x"6";

when b"111011"=> so<=x"0";

when b"111101"=> so<=x"8";

when others=> so<=x"d";

end case;

end if;

end process;

end;
```

```
LIBRARY ieee ;

use ieee.std_logic_1164.all;

entity s7 is port

(clk : in std_logic;

b : in std_logic_vector(1 to 6);

so : out std_logic_vector(1 to 4));

end s7;

architecture behaviour of s7 is

begin

process(clk)

begin

if(clk'event and clk='1') then

case b is

when b"000000"=> so<=x"4";

when b"000010"=> so<=x"b";

when b"000100"=> so<=x"2";

when b"000110"=> so<=x"e";

when b"001000"=> so<=x"f";

when b"001010"=> so<=x"0";

when b"001100"=> so<=x"8";

when b"001110"=> so<=x"d";

when b"010000"=> so<=x"3";

when b"010010"=> so<=x"c";

when b"010100"=> so<=x"9";

when b"010110"=> so<=x"7";
when b"011000"=> so<=x"5";

when b"011010"=> so<=x"a";

when b"011100"=> so<=x"6";

when b"011110"=> so<=x"1";

when b"000001"=> so<=x"d";

when b"000011"=> so<=x"0";

when b"000101"=> so<=x"b";

when b"000111"=> so<=x"7";

when b"001001"=> so<=x"4";

when b"001011"=> so<=x"9";

when b"001101"=> so<=x"1";

when b"001111"=> so<=x"a";

when b"010001"=> so<=x"e";

when b"010011"=> so<=x"3";

when b"010101"=> so<=x"5";

when b"010111"=> so<=x"c";

when b"011001"=> so<=x"2";

when b"011011"=> so<=x"f";

when b"011101"=> so<=x"8";

when b"011111"=> so<=x"6";

when b"100000"=> so<=x"1";

when b"100010"=> so<=x"4";

when b"100100"=> so<=x"b";

when b"100110"=> so<=x"d";

when b"101000"=> so<=x"c";
when b"101010"=> so<=x"3";
when b"101100"=> so<=x"7";
when b"101110"=> so<=x"e";
when b"110000"=> so<=x"a";
when b"110010"=> so<=x"f";
when b"110100"=> so<=x"6";
when b"110110"=> so<=x"8";
when b"111000"=> so<=x"0";
when b"111010"=> so<=x"5";
when b"111100"=> so<=x"9";
when b"111110"=> so<=x"2";
when b"100001"=> so<=x"6";
when b"100011"=> so<=x"b";
when b"100101"=> so<=x"d";
when b"100111"=> so<=x"8";
when b"101001"=> so<=x"1";
when b"101011"=> so<=x"4";
when b"101101"=> so<=x"a";
when b"101111"=> so<=x"7";
when b"110001"=> so<=x"9";
when b"110011"=> so<=x"5";
when b"110101"=> so<=x"0";
when b"110111"=> so<=x"f";
when b"111001"=> so<=x"e";
when b"111011"=> so<=x"2";
when b"111101"=> so<=x"3";
when others=> so<=x"c";
end case;
end if;
end process;
end;

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity s8 is port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end;
architecture behaviour of s8 is
begin
process(clk)
begin
if(clk'event and clk='1') then
case b is
when b"000000"=> so<=x"d";
when b"000010"=> so<=x"2";
when b"000100"=> so<=x"8";
when b"000110"=> so<=x"4";
when b"001000"=> so<=x"6";
when b"001010"=> so<=x"f";

when b"001100"=> so<=x"b";

when b"001110"=> so<=x"1";

when b"010000"=> so<=x"a";

when b"010010"=> so<=x"9";

when b"010100"=> so<=x"3";

when b"010110"=> so<=x"e";

when b"011000"=> so<=x"5";

when b"011010"=> so<=x"0";

when b"011100"=> so<=x"c";

when b"011110"=> so<=x"7";

when b"000001"=> so<=x"1";

when b"000011"=> so<=x"f";

when b"000101"=> so<=x"d";

when b"000111"=> so<=x"8";

when b"001001"=> so<=x"a";

when b"001011"=> so<=x"3";

when b"001101"=> so<=x"7";

when b"001111"=> so<=x"4";

when b"010001"=> so<=x"c";

when b"010011"=> so<=x"5";

when b"010101"=> so<=x"6";

when b"010111"=> so<=x"b";

when b"011001"=> so<=x"0";

when b"011011"=> so<=x"e";
when b"011101"=> so<=x"9";
when b"011111"=> so<=x"2";
when b"100000"=> so<=x"7";
when b"100010"=> so<=x"b";
when b"100100"=> so<=x"4";
when b"100110"=> so<=x"1";
when b"101000"=> so<=x"9";
when b"101010"=> so<=x"c";
when b"101100"=> so<=x"e";
when b"101110"=> so<=x"2";
when b"110000"=> so<=x"0";
when b"110010"=> so<=x"6";
when b"110100"=> so<=x"a";
when b"110110"=> so<=x"d";
when b"111000"=> so<=x"f";
when b"111010"=> so<=x"3";
when b"111100"=> so<=x"5";
when b"111110"=> so<=x"8";
when b"100001"=> so<=x"2";
when b"100011"=> so<=x"1";
when b"100101"=> so<=x"e";
when b"100111"=> so<=x"7";
when b"101001"=> so<=x"4";
when b"101011"=> so<=x"a";
when b"101101"=> so<=x"8";
when b"101111"=> so<=x"d";

when b"110001"=> so<=x"f";

when b"110011"=> so<=x"c";

when b"110101"=> so<=x"9";

when b"110111"=> so<=x"0";

when b"111001"=> so<=x"3";

when b"111011"=> so<=x"5";

when b"111101"=> so<=x"6";

when others=> so<=x"b";

end case;

end if;

end process;

end;


LIBRARY ieee ;

use ieee.std_logic_1164.all;

entity pp is

port(so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x: in std_logic_vector(1 to 4);

ppo : out std_logic_vector(1 to 32));

end pp;

architecture behaviour of pp is

signal XX : std_logic_vector(1 to 32);

begin

XX(1 to 4)<=so1x; XX(5 to 8)<=so2x; XX(9 to 12)<=so3x; XX(13 to 16)<=so4x;
XX(17 to 20)<=so5x; XX(21 to 24)<=so6x; XX(25 to 28)<=so7x; XX(29 to
32)<=so8x;
ppo(1)<=XX(16); ppo(2)<=XX(7); ppo(3)<=XX(20); ppo(4)<=XX(21);
ppo(5)<=XX(29); ppo(6)<=XX(12); ppo(7)<=XX(28); ppo(8)<=XX(17);
ppo(9)<=XX(1); ppo(10)<=XX(15); ppo(11)<=XX(23); ppo(12)<=XX(26);
ppo(13)<=XX(5); ppo(14)<=XX(18); ppo(15)<=XX(31); ppo(16)<=XX(10);
ppo(17)<=XX(2); ppo(18)<=XX(8); ppo(19)<=XX(24); ppo(20)<=XX(14);
ppo(21)<=XX(32); ppo(22)<=XX(27); ppo(23)<=XX(3); ppo(24)<=XX(9);
ppo(25)<=XX(19); ppo(26)<=XX(13); ppo(27)<=XX(30); ppo(28)<=XX(6);
ppo(29)<=XX(22); ppo(30)<=XX(11); ppo(31)<=XX(4); ppo(32)<=XX(25);
end;

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity desxor2 is port
(d,l : in std_logic_vector(1 to 32);
q : out std_logic_vector(1 to 32));
end desxor2;
architecture behaviour of desxor2 is
begin
q<=d xor l;
end;
```

```
LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity roundfunc is port
(clk : in std_logic;
li,ri : in std_logic_vector(1 to 32);
k : in std_logic_vector(1 to 48);
lo,ro : out std_logic_vector(1 to 32));
end roundfunc;
architecture behaviour of roundfunc is
component s1 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s2 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s3 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s4 port
```

```
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s5 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s6 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s7 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component s8 port
(clk : in std_logic;
b : in std_logic_vector(1 to 6);
so : out std_logic_vector(1 to 4));
end component;
component pp port
```

```
(so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x
: in std_logic_vector(1 to 4);
ppo : out std_logic_vector(1 to 32));
end component;
component desxor2 port
(d,l : in std_logic_vector(1 to 32);
q : out std_logic_vector(1 to 32));
end component;
component desxor1 port
(e : in std_logic_vector(1 TO 48);
b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x
: out std_logic_vector (1 TO 6);
k : in std_logic_vector (1 TO 48));
end component;
component xp port
(ri : in std_logic_vector(1 TO 32);
e : out std_logic_vector(1 TO 48));
end component;
signal e : std_logic_vector(1 to 48);
signal b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x
: std_logic_vector(1 to 6);
signal so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x
: std_logic_vector(1 to 4);
signal ppo : std_logic_vector(1 to 32);
begin
```

```
uxp: xp port map ( ri=>ri, e=>e );

udesxor1: desxor1 port map ( e=>e, k=>k, b1x=>b1x, b2x=>b2x, b3x=>b3x,

b4x=>b4x, b5x=>b5x,b6x=>b6x, b7x=>b7x, b8x=>b8x );

us1: s1 port map ( clk=>clk, b=>b1x, so=>so1x );

us2: s2 port map ( clk=>clk, b=>b2x, so=>so2x );

us3: s3 port map ( clk=>clk, b=>b3x, so=>so3x );

us4: s4 port map ( clk=>clk, b=>b4x, so=>so4x );

us5: s5 port map ( clk=>clk, b=>b5x, so=>so5x );

us6: s6 port map ( clk=>clk, b=>b6x, so=>so6x );

us7: s7 port map ( clk=>clk, b=>b7x, so=>so7x );

us8: s8 port map ( clk=>clk, b=>b8x, so=>so8x );

upp: pp port map ( so1x=>so1x, so2x=>so2x, so3x=>so3x, so4x=>so4x, so5x=>so5x,

so6x=>so6x,so7x=>so7x, so8x=>so8x, ppo=>ppo );

udesxor2: desxor2 port map ( d=>ppo, l=>li, q=>ro );

lo<=ri;

end;


LIBRARY ieee ;

use ieee.std_logic_1164.all;

entity fp is port

(l,r : in std_logic_vector(1 to 32);

ct : out std_logic_vector(1 to 64));

end fp;

architecture behaviour of fp is

begin
```

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

LIBRARY ieee ;
use ieee.std_logic_1164.all;
entity desenc is port
(pt : in std_logic_vector(1 TO 64);
key : in std_logic_vector(1 TO 64);
ct : out std_logic_vector(1 TO 64);
clk : in std_logic
);
end desenc;
architecture behavior of desenc is
```

```
component keysched port

(key : in std_logic_vector(1 to 64);

k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x

: out std_logic_vector(1 to 48));

end component;

component roundfunc port

(clk : in std_logic;

li,ri : in std_logic_vector(1 to 32);

k : in std_logic_vector(1 to 48);

lo,ro : out std_logic_vector(1 to 32));

end component;

component ip port

(pt : in std_logic_vector(1 TO 64);

l0x : out std_logic_vector(1 TO 32);

r0x : out std_logic_vector(1 TO 32));

end component;

component fp port

(l,r : in std_logic_vector(1 to 32);

ct : out std_logic_vector(1 to 64));

end component;

signal k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x :

std_logic_vector(1 to 48);

signal l0x,l1x,l2x,l3x,l4x,l5x,l6x,l7x,l8x,l9x,l10x,l11x,l12x,l13x,l14x,l15x,l16x :

std_logic_vector(1 to 32);
```

```
signal r0x,r1x,r2x,r3x,r4x,r5x,r6x,r7x,r8x,r9x,r10x,r11x,r12x,r13x,r14x,r15x,r16x :
std_logic_vector(1 to 32);
begin
ukeysched: keysched port map ( key=>key, k1x=>k1x, k2x=>k2x, k3x=>k3x,
k4x=>k4x, k5x=>k5x, k6x=>k6x,k7x=>k7x, k8x=>k8x, k9x=>k9x, k10x=>k10x,
k11x=>k11x, k12x=>k12x, k13x=>k13x,k14x=>k14x, k15x=>k15x, k16x=>k16x );
uip: ip port map ( pt=>pt, l0x=>l0x, r0x=>r0x );
round1: roundfunc port map ( clk=>clk, li=>l0x, ri=>r0x, lo=>l1x, ro=>r1x, k=>k1x );
round2: roundfunc port map ( clk=>clk, li=>l1x, ri=>r1x, lo=>l2x, ro=>r2x, k=>k2x );
round3: roundfunc port map ( clk=>clk, li=>l2x, ri=>r2x, lo=>l3x, ro=>r3x, k=>k3x );
round4: roundfunc port map ( clk=>clk, li=>l3x, ri=>r3x, lo=>l4x, ro=>r4x, k=>k4x );
round5: roundfunc port map ( clk=>clk, li=>l4x, ri=>r4x, lo=>l5x, ro=>r5x, k=>k5x );
round6: roundfunc port map ( clk=>clk, li=>l5x, ri=>r5x, lo=>l6x, ro=>r6x, k=>k6x );
round7: roundfunc port map ( clk=>clk, li=>l6x, ri=>r6x, lo=>l7x, ro=>r7x, k=>k7x );
round8: roundfunc port map ( clk=>clk, li=>l7x, ri=>r7x, lo=>l8x, ro=>r8x, k=>k8x );
round9: roundfunc port map ( clk=>clk, li=>l8x, ri=>r8x, lo=>l9x, ro=>r9x, k=>k9x );
round10: roundfunc port map ( clk=>clk, li=>l9x, ri=>r9x,
lo=>l10x,ro=>r10x,k=>k10x);
round11: roundfunc port map ( clk=>clk, li=>l10x,ri=>r10x,lo=>l11x,ro=>r11x,
k=>k11x );
round12: roundfunc port map ( clk=>clk, li=>l11x, ri=>r11x, lo=>l12x, ro=>r12x,
k=>k12x );
round13: roundfunc port map ( clk=>clk, li=>l12x, ri=>r12x, lo=>l13x, ro=>r13x,
k=>k13x );
```

```
round14: roundfunc port map ( clk=>clk, li=>l13x, ri=>r13x, lo=>l14x, ro=>r14x,
k=>k14x );
round15: roundfunc port map ( clk=>clk, li=>l14x, ri=>r14x, lo=>l15x, ro=>r15x,
k=>k15x );
round16: roundfunc port map ( clk=>clk, li=>l15x, ri=>r15x, lo=>l16x, ro=>r16x,
k=>k16x );
ufp: fp port map ( l=>r16x, r=>l16x, ct=>ct );
end behavior;
```

Testbench for encryption

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use     std.textio.all;
use ieee.std_logic_textio.all;
entity enctestbench is end;
architecture tb of  enctestbench is
        component desenc
                port (pt: in std_logic_vector(1 to 64);
               key:in std_logic_vector(1 to 64);
              ct:out std_logic_vector(1 to 64);
               clk:in std_logic);
       end component ;
signal  pt:  std_logic_vector(1 to 64);
signal  key: std_logic_vector(1 to 64);
signal  ct: std_logic_vector(1 to 64);
signal  clk: std_logic;
begin
        UUT:desenc port map(pt=>pt,key=>key,ct=>ct,clk=>clk);
        process
        begin
       clk <='0';
        wait for 5ns;
        clk     <='1';
        wait for 5ns;
end process;
pt<="0100111001101111011101110010000001101001011100110010000001110100";
key<="0000000100100011010001010110011110001001101010111100110111101111";
end tb;
```

Testbench for decryption

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use     std.textio.all;
use ieee.std_logic_textio.all;
entity dectestbench is end;
        architecture tb of  dectestbench is
        component desdec
                port (pt: in std_logic_vector(1 to 64);
                    key:in std_logic_vector(1 to 64);
                    ct:out std_logic_vector(1 to 64);
                    clk:in std_logic);
          end component ;
signal   pt:  std_logic_vector(1 to 64);
signal  key: std_logic_vector(1 to 64);
signal  ct: std_logic_vector(1 to 64);
signal   clk: std_logic;
begin
                    UUT:desdec port map(pt=>pt,key=>key,ct=>ct,clk=>clk);
                    process
                    begin
                    clk <='0';
                    wait for 5ns;
                    clk      <='1';
                    wait for 5ns;
end process;
pt<="0011111110100100000011101000101010011000010011010100100000010101";
key<="0000000100100011010001010110011110001001101010111100110111101111";
end tb;
```