IMPLEMENTING A SECURE ENCRYPTION ALGORITHM USING ELLIPTIC CURVES

Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Technology

By

Batch No.2 :

Prathesh Srinivas 21ECB0F24
Malavika Dilu 21ECB0F24
Kolla Sree Harshitha 21ECB0F24

Guided By:

Dr. P. Prithvi
Dr. V. Narendar

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL
2022-23

ABSTRACT

Elliptic Curve Cryptography is a public key cryptography technique based on the algebraic structure of elliptic curves over finite fields. To provide the same level of security as the other algorithm, it performs the process in less time, less memory, less computations and less power consumption. It bridges the gap between public and private details. Thus, it is suitable for environments with constraints and like computing power, memory and battery life. Typical examples of such environments are embedded devices, wireless devices and real-time applications. On the other hand, FPGA is considered as the modern-day technology for building prototypes where the chip can be configured by the end user to realize different designs by programming logic blocks and interconnects. The hardware implementation of ECC using FPGA enhances the system performance since FPGA technology is faster by nature and much more secure than software crypto-systems as they are vulnerable to attacks over the cloud.

TABLE OF CONTENTS

LIST OF FIGURES

<u>CERTIFICATE</u>

This is to certify that the dissertation work entitled **IMPLEMENTING A SECURE ENCRYPTION ALGORITHM USING ELLIPTIC CURVES** is a bonafide record of seminar work carried out by PRATHESH SRINIVAS (21ECB0F22), MALAVIKA DILU (21ECB0F23), SREE HARSHITHA KOLLA (21ECB0F24) submitted to the faculty of "Electronics and Communication Engineering Department", in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in "Electronics and Communication Engineering" at National Institute of Technology, Warangal during the academic year 2022-2023.

Dr. P. Prithvi

Assistant Professor

Department of Electronics and Communication Engineering

National Institute of Technology, Warangal

Dr. V. Narendar

Assistant Professor

Department of Electronics and Communication Engineering

National Institute of Technology, Warangal

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to our faculty in-charge Dr. P. Prithvi, Assistant Professor, Department of Electronics and Communication Engineering, National Institute of Technology, Warangal and Dr. V. Narender, Assistant Professor, Department of Electronics and Communication Engineering, National Institute of Technology, Warangal for their constant supervision, guidance, suggestions and encouragement during this seminar.

INTRODUCTION

As technology continues to evolve and data security becomes increasingly important, implementing strong encryption algorithms is more critical than ever before. FPGAs are versatile and powerful tools that can be used to implement complex algorithms efficiently. ECC is a public-key encryption technique that offers a high level of security with a relatively small key size compared to other encryption methods. Implementing ECC on an FPGA allows for fast and efficient encryption and decryption of data, making it ideal for applications that require real-time data processing and transmission.

The main objective of this project is to design and implement a secure ECC-based encryption system on an FPGA board, utilizing the Verilog hardware description language. We will also evaluate the performance of the system in terms of speed, area utilization, and power consumption, and compare it with existing software-based encryption systems.
The implementation of a secure encryption algorithm using ECC on an FPGA has practical applications in various fields, including secure communications, financial transactions, and data storage. By completing this project, we aim to demonstrate the effectiveness of FPGA-based implementations of encryption algorithms and contribute to the development of secure data transmission and storage systems.

Cryptography is the practice of securing communication from third-party interference by using mathematical algorithms and protocols. It is used to protect sensitive information such as financial transactions, personal data, and confidential communications.It works by encrypting the original data using a secret key or a password. This encrypted data, or ciphertext, is then transmitted securely to the intended recipient. The recipient can then use the same key or password to decrypt the ciphertext and recover the original message. There are various cryptographic algorithms and protocols available, such as symmetric-key cryptography, public-key cryptography, hash functions, and digital signatures. These techniques are used to ensure the confidentiality, integrity, and authenticity of data in communication and storage.

METHODS

Elliptic Curve Cryptography (ECC) is a type of public key cryptography that is based on the algebraic structure of elliptic curves over finite fields. In ECC, the public key consists of a point on an elliptic curve, and the private key is a randomly chosen number that is used to derive the public key.

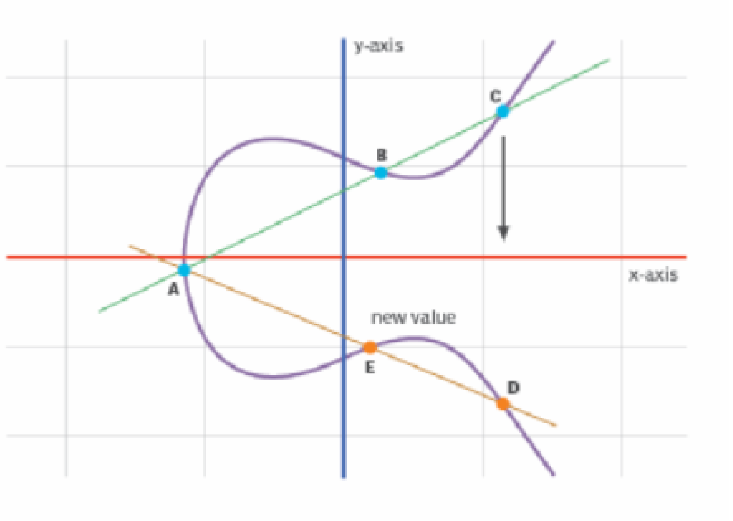


Fig 1. Elliptic curve showing three points of intersection

The elliptic curve for the ECC is based on the points satisfying the equation:

$$y^2 = x^3 + ax + b$$

$$\text{where, } 4a^3 + 27b^2 \neq 0$$

ECC is widely used in a variety of applications, including secure communication protocols, digital signatures, and key exchange protocols. It is particularly well-suited for use in mobile devices and other resource-constrained environments where smaller key sizes and faster operations are important. However, despite its advantages, ECC is also more complex to implement and can be vulnerable to certain types of attacks if not implemented correctly.

2. 1. ECC ALGORITHM

(Here, Pm is a (x,y) point encoded with the help of plaintext message 'm'. The Pm is the point used for encryption and decryption.)

Global Public Elements

Step I. Eq(a, b) elliptic curve with parameters a, b, and q, where q is a prime or integer of the form 2m. Step II. G point on elliptic curve whose order is large value n

User A Key Generation

Step I. Select private key nA; nA < n

Step II. Calculate public key PA

Step III. PA = nAG User

B Key Generation

Step I. Select private key nB; nB < n

Step II. Calculate public key PB

Step III. PB = nBG

Calculation of Secret Key by User A

Step I. K = nAPB

Calculation of Secret Key by User B

Step I. K = nBPA

Encryption by A using B's Public Key

Step I. A chooses message Pm and a random positive integer 'k'

Step II. Ciphertext: Cm = { kG, Pm + kPB }

Decryption by B using his own Private Key

Step I. Ciphertext: Cm

Step II. Plaintext: Pm = Pm + kPB - nB (kG) = Pm + k(nBG) - nB (kG)

Once the secret key has been generated, the message signal and secret key are entered into the AES algorithm for encryption and decryption. Adding this second layer of encryption enhances the security provided by the algorithm. AES on its own is a secure system, but in addition to that, when elliptic curves are used to generate a key for the algorithm, it generates more than a billion different possibilities hence decryption is not possible without having the correct key.

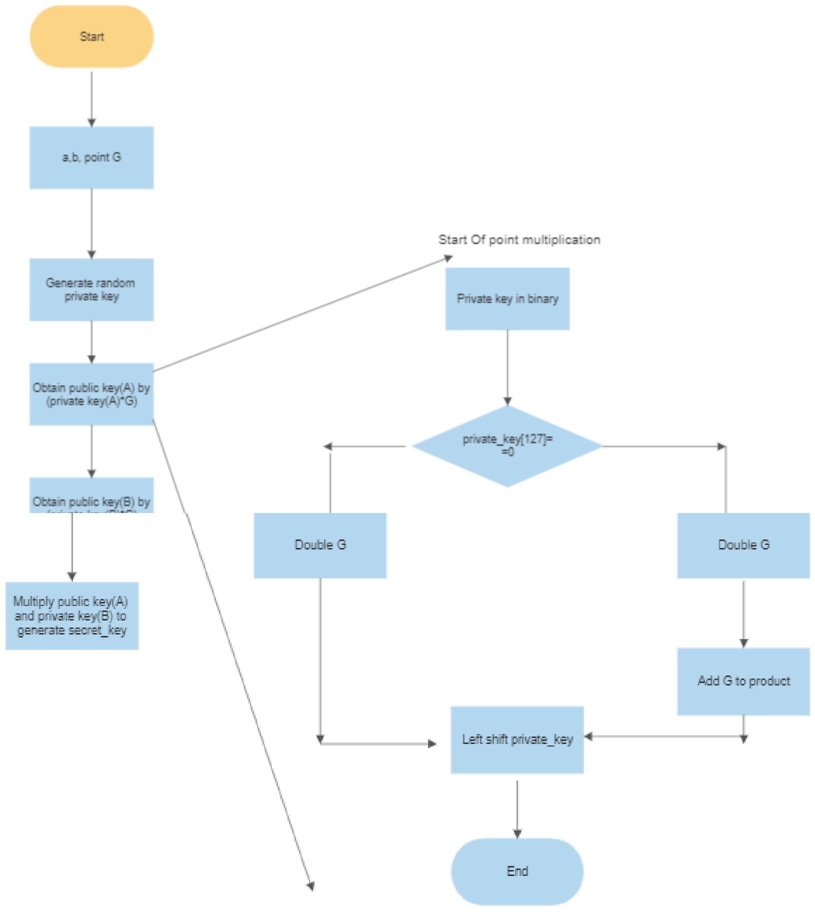

Start
a,b, point G
Generate random private key
Obtain public key(A) by (private key(A)*G)
Obtain public key(B) by
Multiply public key(A) and private key(B) to generate secret_key
Start Of point multiplication
Private key in binary
private_key[127]==0
Double G
Double G
Add G to product
Left shift private_key
End

Fig 2. ECC Algorithm for generating Secret Key

2.1.1. POINT ADDITION

It is a property of elliptic curves that upon adding any one point on the elliptic curve to another point on the elliptic curve, the result is also going to be on the elliptic curve itself.

We perform point addition using the following formula to calculate the slope of the curve at the given point:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

After calculating the slope of the curve at that point, the coordinates of the resultant point are obtained using the following formulae:

$$x_3 = m^2 - x_1 - x_2$$

$$y_3 = m(x_1 - x_3) - y_1$$

The resulting point satisfies the equation of the curve, and hence exists on the same curve.

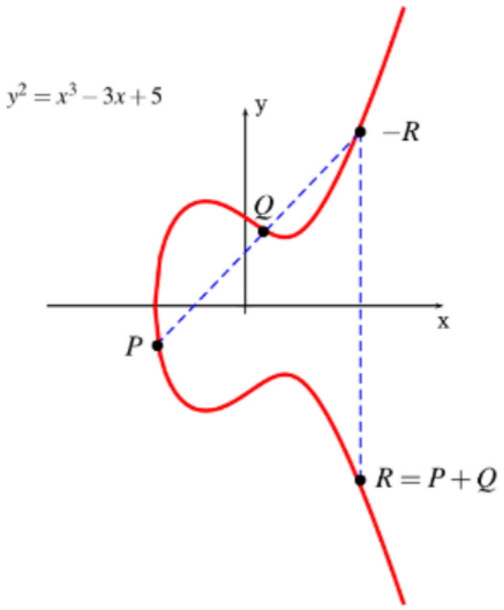


Fig 3. Point Addition and the Resultant Point

2.1.2. POINT DOUBLING

When the same point is added twice, it is called point doubling. This is very useful in calculating the product of a point and a number (used to generate the public key in the ECC algorithm). The following formulae are used to calculate the resulting point

$$m = \frac{3x_1^2 + A}{2y_1}$$

(here, A is the coefficient of x term)

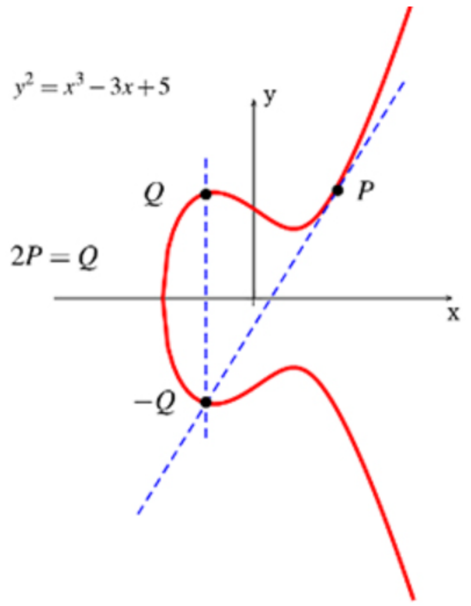


Fig 4. Point Doubling and the Resultant Point

2.1.3. POINT MULTIPLICATION

To multiply a given point (P) on the elliptic curve, first the number to multiply is converted to binary format. The next step is to iterate through the digits of the resulting binary number. The following steps are followed to obtain the product:

1. Ignore the MSB of the binary number
2. From the second number onwards, if it is a 0, then the point is doubled (2P), and if it is a 1, then the point is doubled and the original point is added to the doubled resultant (2P+P).

2.2. AES ALGORITHM:

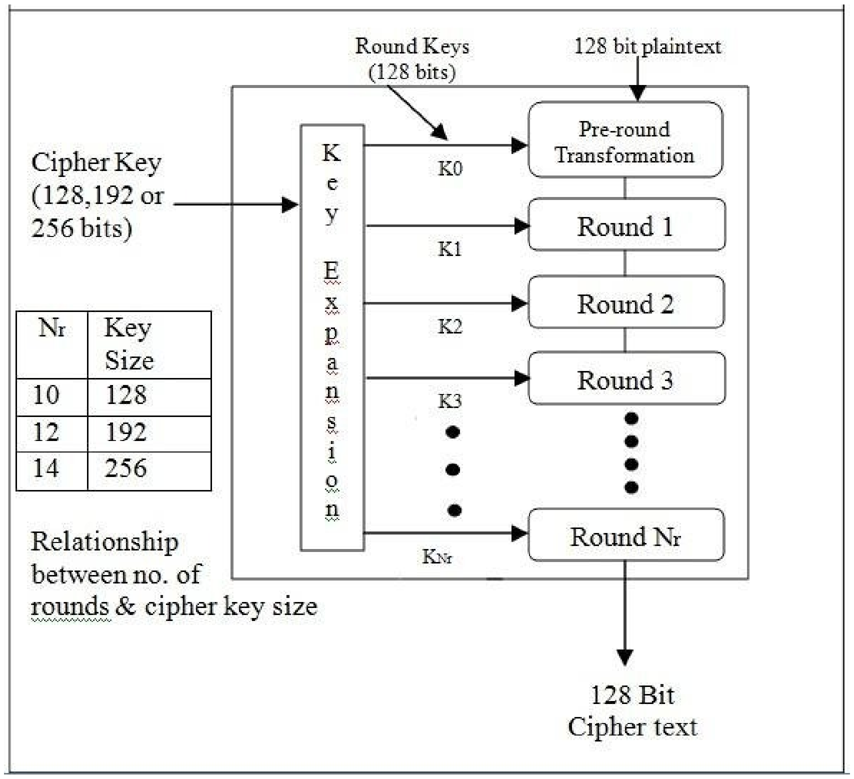
Advanced Encryption Standard is implemented in software and hardware throughout the world to encrypt sensitive data. It is essential for government computer security, cybersecurity and electronic data protection. It was released by the National Institute of Standards and Technology in December 2001 as a replacement for DES Algorithm.

It is a symmetric key-block encryption algorithm which has a fixed data block of 128 bits and variable key length of 128, 192 or 256 bits. The secret key is used to encrypt and decrypt the data and it works on the plain text. The length of the key chosen depends on the level of security required (highly sensitive data is encrypted using the 256 bit secret key). AES algorithm involves a number of substitutions and permutations to scramble the initial plain text data into a cipher text.

AES includes three block ciphers:

1. AES-128 uses a 128-bit key length to encrypt and decrypt a block of messages.
2. AES-192 uses a 192-bit key length to encrypt and decrypt a block of messages.
3. AES-256 uses a 256-bit key length to encrypt and decrypt a block of messages.

Symmetric, also known as secret key, ciphers use the same key for encrypting and decrypting. The sender and the receiver must both know -- and use -- the same secret key.



Nr	Key Size
10	128
12	192
14	256

Fig 5. Block diagram of AES Algorithm

2.2.1. KEY EXPANSION

It is the process by which cipher keys are generated. The key expansion process takes in the input original cipher key and produces a set of round keys that will be used in each round of encryption. The number of rounds in a particular encryption depends on the size or length of the cipher key. Therefore, there are majorly 3 versions of AES with 10, 12 and 14 rounds for key length of size 128, 192 or 256 bits, respectively.

The key expansion process involves several steps:

Key Expansion Constant: A set of fixed values called key expansion constants are used as the basis for generating the round keys.

Key Schedule: The original cipher key is divided into blocks of bytes, and the key schedule generates a set of words that will be used to produce the round keys.

Key Expansion: The key expansion algorithm uses a combination of substitution and permutation operations to generate a set of round keys that are derived from the key schedule.

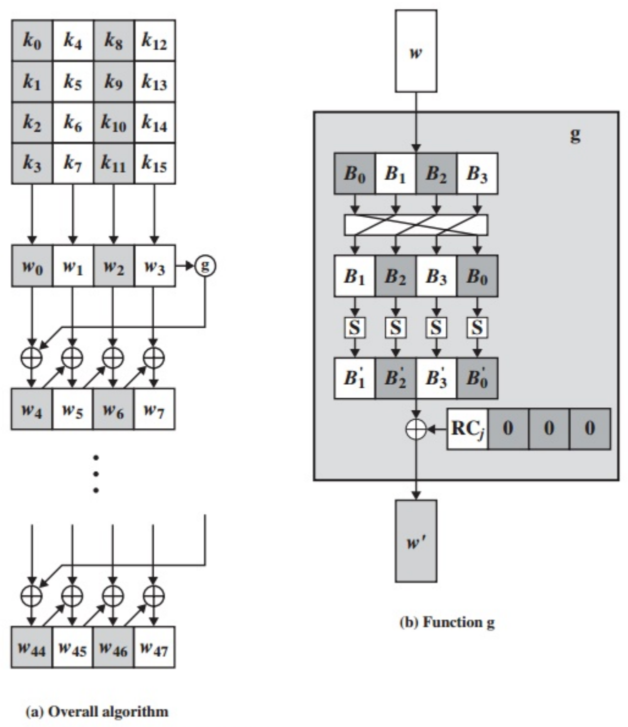

k_0 k_4 k_8 k_{12}
k_1 k_5 k_9 k_{13}
k_2 k_6 k_{10} k_{14}
k_3 k_7 k_{11} k_{15}
w_0 w_1 w_2 w_3
g
w_4 w_5 w_6 w_7
w_{44} w_{45} w_{46} w_{47}
(a) Overall algorithm
w
g
B_0 B_1 B_2 B_3
B_1 B_2 B_3 B_0
S S S S
B'_1 B'_2 B'_3 B'_0
RC_j 0 0 0
w'
(b) Function g

Figure 5.9 AES Key Expansion

Fig 6. AES Key Expansion

2.2.2 ROUND KEY GENERATION

The block data stored in the state array is XORed with the round 0 key produced and the resultant state is fed into the next step as input.

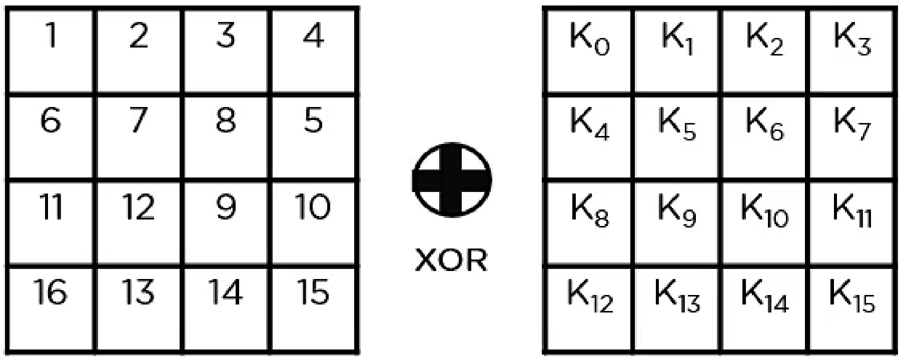


Fig 7. Generation of round key by XOR

2.2.3 ROUNDS AND FUNCTIONS WITHIN THEM

The number of rounds depends on the key length; 10, 12 or 14 rounds based on if the key is 128, 192 or 256 bits long, respectively. Each round ends with the addition of a round key. The state which is obtained after the round key addition acts as the input for the next process.

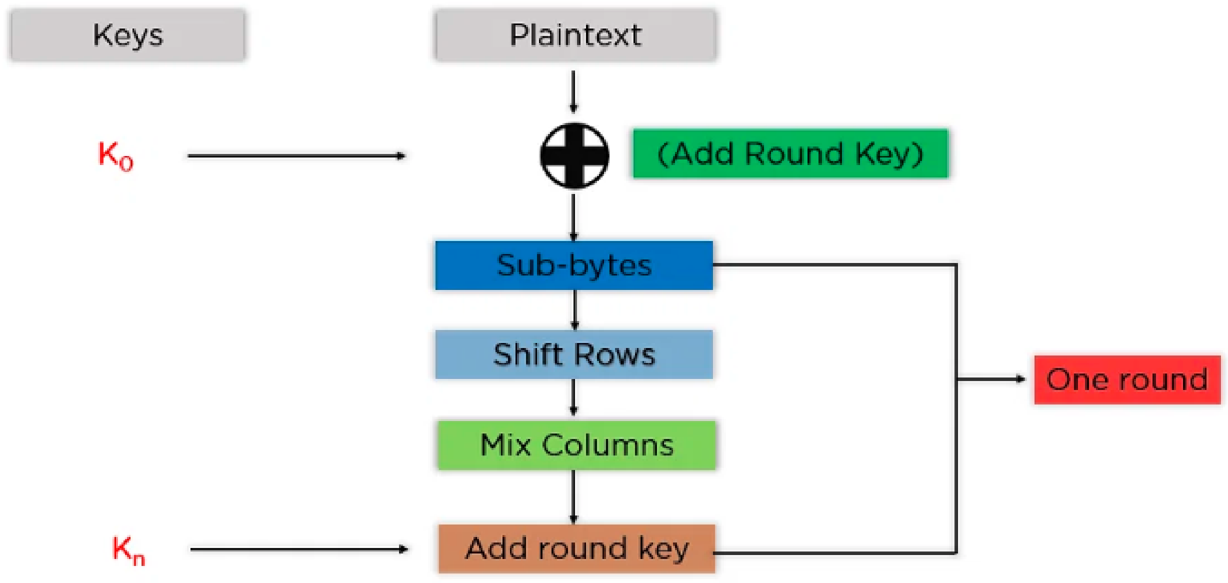


Fig 8. Flow chart with functions within the rounds

Under every round, there are a few processes that take place which help in increasing the security of the AES encryption algorithm. They are as follows:

2.2.4. S-BOX

. S-box is a process where substitution of values takes place by using a LUT. It converts input data into secret data using substitution which results in a more complex relation between the key and the cipher text.

. The process is done by converting each element of the entire array into hexadecimal format and dividing them into 2 parts, rows and columns. The LUT consists of 256 bit values and so the rows and columns are individually matched with the LUT and hence there is a generation of new bit values which makes up the final state array. The Figure related to this is given below.

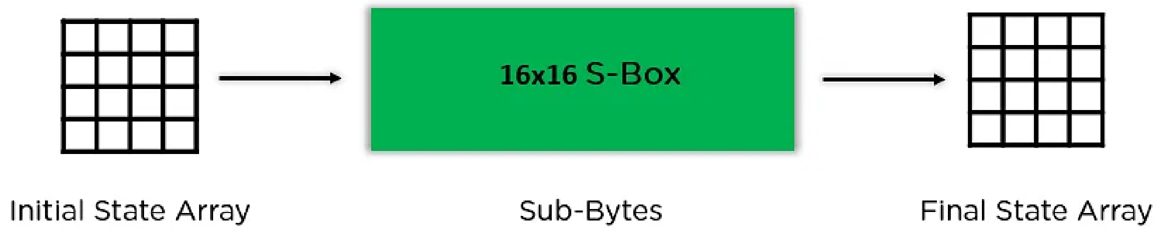
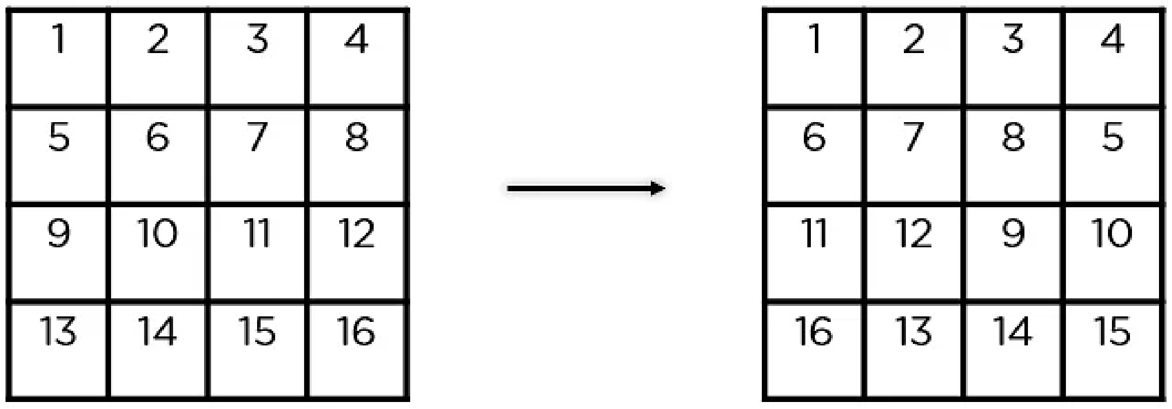


Fig 9. S-Box representation

2.2.5. P-BOX

. The p-box process also called the shift row process is where to increase the security even more, as the name suggests, shifting elements are done.

. Here, in the 4 x 4 array that we have, the first row does not undergo any changes. There is a left shift of one element in the second row, left shift of 2 elements in the 3rd row and so on. This helps in increasing the security. The figure is shown below



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
6	7	8	5
11	12	9	10
16	13	14	15

Fig 10. P-Box representation

2.2.6. Mix column

. In the mix column box, the security is further tightened by doing a multiplication process. Here, we take a fixed array of 4x1 matrix and we take our new state array. We then multiply each term of our array to the fixed array and hence the product obtained is our required new state array.

. This step is usually not carried out for the last round of the AES mechanism to maintain symmetry between the encryption and the decryption process. The figure is shown below.

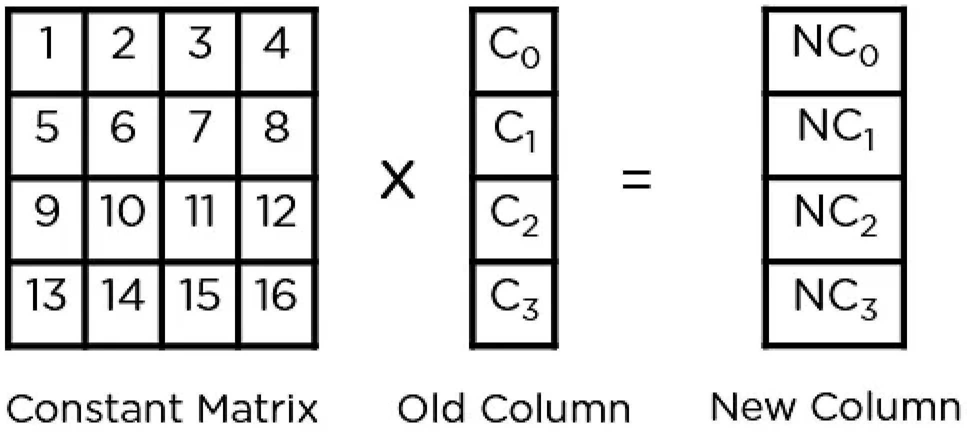


Fig 11. Mix Column Representation

2.2.7. Round Key Addition

. This step is carried out at the end of every round. First, the 16 byte value obtained from the state array is converted back to 128 bit value. Now the new cipher comes into picture and is XORed with the 128 bit value and an output is obtained.

. Now if this round is the last round of the AES block then the newly obtained 128 bit value is taken out as the cipher text and is our required value. Else if there are many more rounds after this, the 128 bit value is again converted to 16 byte value and goes into the S.BOX of the net round and the process continues. The figure is shown below

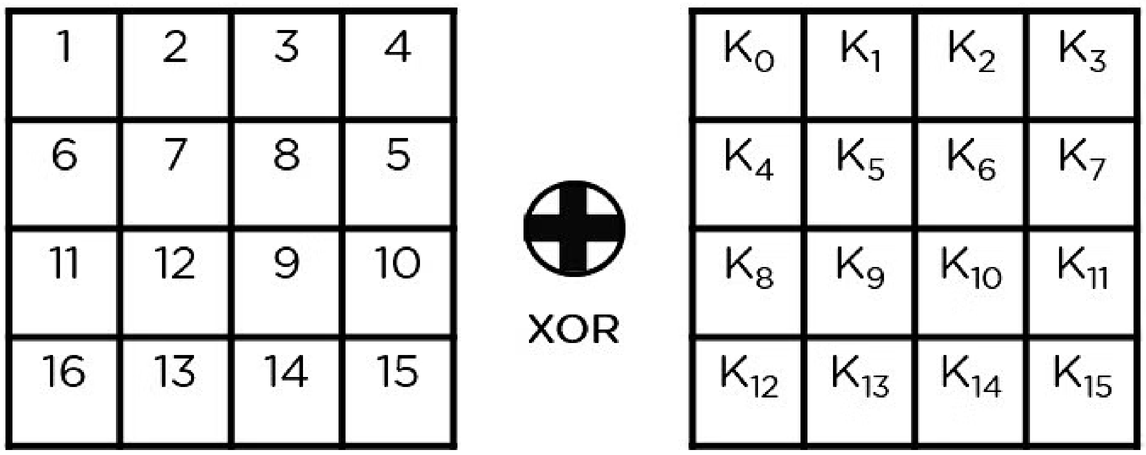


Fig 12. Round key addition Representation

CODE

```
module ecc_encrypt(
   input [127:0] plaintext,      // 128-bit plaintext input
   input [8:0]a;
   input [8:0]b;
   input [8:0]prime_field;
   input [63:0]G1;
   input [63:0]G2;
   input [127:0] public_key,   // 128-bit public key
   output reg [127:0] ciphertext // 128-bit ciphertext output);

// Point addition operation for elliptic curve cryptography
function [127:0] ecc_add
( input [127:0] P1,  // First point on curve
    input [127:0] P2   // Second point on curve);
  reg [127:0] R;
  reg [127:0] X1, Y1, X2, Y2;
  reg [127:0] lambda, temp, inv_temp;

  // Separate the X and Y values of the points
  assign X1 = P1[127:64];
  assign Y1 = P1[63:0];
  assign X2 = P2[127:64];
  assign Y2 = P2[63:0];

  if (P1 == P2) begin       // Check if points are equal
    lambda = (3 * X1 * X1 + a) * inv_mod(2 * Y1, p);
  end else begin
    temp = X2 - X1;
    inv_temp = inv_mod(temp, p);
    lambda = (Y2 - Y1) * inv_temp;
  end
  R[127:64] = (lambda * lambda - X1 - X2) % p;    // Compute the X and Y values of the result
point
  R[63:0] = (lambda * (X1 - R[255:128]) - Y1) % p;
  ecc_add = R;
endfunction
```

```
// Modular inverse operation for elliptic curve cryptography
function [127:0] inv_mod(
    input [127:0] a,   // Input value
    input [127:0] p    // Modulus);
  reg [127:0] t, new_t;
  reg [127:0] r, new_r;

  // Initialize variables
  t = 0; new_t = 1;
r = p; new_r = a;

  // Calculate inverse using extended Euclidean algorithm
  integer i;
  for (i = 0; i < 128; i = i + 1) begin
    if (new_r == 0) begin
      inv_mod = 0;
      return;
    end
    if (new_r == 1) begin
      inv_mod = t;
      return;
    end
    temp = r / new_r;
    t = new_t;
    new_t = t - temp * new_t;
    r = new_r;
    new_r = r - temp * new_r;
  end
inv_mod = t;
endfunction

// Generate a random private key for ECC
function [127:0] ecc_gen_privkey;
  ecc_gen_privkey = $random % p;
endfunction

// Compute the public key for ECC
function [127:0] ecc_gen_pubkey(
    input [127:0] privkey   // Private key);
  ecc_gen_pubkey = ecc_mult(G, privkey);
endfunction

// Point multiplication operation for elliptical curves
function [127:0] ecc_mult(input [127:0] privkey, input [12:0] R);

integer i;
for (i=0; i<128; i=i+1)
begin
    privkey<<1;
    if(privkey==0)
        ecc_add(R, R);
    else
        begin
        assign 2R = ecc_add(R,R);
        acc_add(2R, R);
        end
end
endfunction

function secret_key_gen(input [127:0]privkey, input [127:0] pubkey);
assign secret_key_gen = ecc_mult(privkey, pubkey);
endfunction

endmodule
```

```
`timescale 1ns / 1ps

module encrypt(enable,in,secret_key, e128, d128, e192, d192, e256,
d256,encrypted128,encrypted192,encrypted256,decrypted128,decrypted192,decrypted256);

output wire e128;

output wire d128;

output wire e192;

output wire d192;

output wire e256;

output wire d256;

input enable;

input [127:0]in;

input [127:0]secret_key;

// The plain text used as input

wire[127:0] in = 128'h_00112233445566778899aabbccddeeff;

// The different keys used for testing (one of each type)

wire[127:0] key128 = 128'h_000102030405060708090a0b0c0d0e0f;

wire[191:0] key192 = 192'h_000102030405060708090a0b0c0d0e0f1011121314151617;

wire[255:0] key256 =
256'h_000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f;


// The expected outputs from the encryption module

wire[127:0] expected128 = 128'h_69c4e0d86a7b0430d8cdb78070b4c55a;

wire[127:0] expected192 = 128'h_dda97ca4864cdfe06eaf70a0ec0d7191;

wire[127:0] expected256 = 128'h_8ea2b7ca516745bfeafc49904b496089;


// The result of the encryption module for every type

output [127:0] encrypted128;

output [127:0] encrypted192;

output [127:0] encrypted256;
```

```
assign e128 = (encrypted128 == expected128 && enable) ? 1'b1 : 1'b0;

assign e192 = (encrypted192 == expected192 && enable) ? 1'b1 : 1'b0;

assign e256 = (encrypted256 == expected256 && enable) ? 1'b1 : 1'b0;


// The result of the decryption module for every type

output [127:0] decrypted128;

output [127:0] decrypted192;

output [127:0] decrypted256;


AES_Encrypt a(in,key128,encrypted128);

AES_Encrypt #(192,12,6) b(in,key192,encrypted192);

AES_Encrypt #(256,14,8) c(in,key256,encrypted256);


AES_Decrypt a2(encrypted128,key128,decrypted128);

AES_Decrypt #(192,12,6) b2(encrypted192,key192,decrypted192);

AES_Decrypt #(256,14,8) c2(encrypted256,key256,decrypted256);


assign d128 = (decrypted128 == in && enable) ? 1'b1 : 1'b0;

assign d192 = (decrypted192 == in && enable) ? 1'b1 : 1'b0;

assign d256 = (decrypted256 == in && enable) ? 1'b1 : 1'b0;

endmodule


module AES_Decrypt#(parameter N=128,parameter Nr=10,parameter Nk=4)(in,key,out);

input [127:0] in;

input [N-1:0] key;

output [127:0] out;

wire [(128*(Nr+1))-1 :0] fullkeys;

wire [127:0] states [Nr+1:0] ;
wire [127:0] afterSubBytes;

wire [127:0] afterShiftRows;


keyExpansion #(Nk,Nr) ke (key,fullkeys);

addRoundKey addrk1 (in,states[0],fullkeys[127:0]);

genvar i;

generate

        for(i=1; i<Nr ;i=i+1)begin : loop

                decryptRound dr(states[i-1],fullkeys[i*128+:128],states[i]);

                end

                inverseShiftRows sr(states[Nr-1],afterShiftRows);

                inverseSubBytes sb(afterShiftRows,afterSubBytes);

                addRoundKey addrk2(afterSubBytes,states[Nr],fullkeys[((128*(Nr+1))-1)-:128]);

                        assign out=states[Nr];


endgenerate

endmodule


module AES_Encrypt#(parameter N=128,parameter Nr=10,parameter Nk=4)(in,key,out);

input [127:0] in;

input [N-1:0] key;

output [127:0] out;

wire [(128*(Nr+1))-1 :0] fullkeys;

wire [127:0] states [Nr+1:0] ;

wire [127:0] afterSubBytes;

wire [127:0] afterShiftRows;


keyExpansion #(Nk,Nr) ke (key,fullkeys);

addRoundKey addrk1 (in,states[0],fullkeys[((128*(Nr+1))-1)-:128]);
```

```
genvar i;

generate

        for(i=1; i<Nr ;i=i+1)begin : loop

                encryptRound er(states[i-1],fullkeys[(((128*(Nr+1))-1)-128*i)-:128],states[i]);

                end

                subBytes sb(states[Nr-1],afterSubBytes);

                shiftRows sr(afterSubBytes,afterShiftRows);

                addRoundKey addrk2(afterShiftRows,states[Nr],fullkeys[127:0]);

                        assign out=states[Nr];


endgenerate

endmodule


// generates round key

module addRoundKey(data, out, key);


input [127:0] data;

input [127:0] key;

output [127:0] out;


assign out = key ^ data;

endmodule


//for decryption

module decryptRound(in,key,out);

input [127:0] in;

output [127:0] out;

input [127:0] key;

wire [127:0] afterSubBytes;
wire [127:0] afterShiftRows;

wire [127:0] afterMixColumns;

wire [127:0] afterAddroundKey;


inverseShiftRows r(in,afterShiftRows);

inverseSubBytes s(afterShiftRows,afterSubBytes);

addRoundKey b(afterSubBytes,afterAddroundKey,key);

inverseMixColumns m(afterAddroundKey,out);


endmodule


// used for encryption

module encryptRound(in,key,out);

input [127:0] in;

output [127:0] out;

input [127:0] key;

wire [127:0] afterSubBytes;

wire [127:0] afterShiftRows;

wire [127:0] afterMixColumns;

wire [127:0] afterAddroundKey;


subBytes s(in,afterSubBytes);

shiftRows r(afterSubBytes,afterShiftRows);

mixColumns m(afterShiftRows,afterMixColumns);

addRoundKey b(afterMixColumns,out,key);


endmodule


module inverseMixColumns(state_in,state_out);
```

```
input [127:0] state_in;

output [127:0] state_out;


//This function multiply by {02} n-times

function[7:0] multiply(input [7:0]x,input integer n);

integer i;

begin

        for(i=0;i<n;i=i+1)begin

                if(x[7] == 1) x = ((x << 1) ^ 8'h1b);

                else x = x << 1;

        end

        multiply=x;

end

endfunction


function [7:0] mb0e; //multiply by {0e}

input [7:0] x;

begin

        mb0e=multiply(x,3) ^ multiply(x,2)^ multiply(x,1);

end

endfunction


function [7:0] mb0d; //multiply by {0d}

input [7:0] x;

begin

        mb0d=multiply(x,3) ^ multiply(x,2)^ x;

end

endfunction
```

```
endmodule


//used for inverse subtitution

module inverseSbox(selector,sbout);

input  [7:0] selector;

output reg [7:0] sbout;


 always@(*)

 begin

   case(selector)

                                                    8'h00:sbout =8'h52;

                                                    8'h01:sbout =8'h09;

                                                    8'h02:sbout =8'h6a;

                                                    8'h03:sbout =8'hd5;

                                                    8'h04:sbout =8'h30;

                                                    8'h05:sbout =8'h36;

                                                    8'h06:sbout =8'ha5;

                                                    8'h07:sbout =8'h38;

                                                    8'h08:sbout =8'hbf;

                                                    8'h09:sbout =8'h40;

                                                    8'h0a:sbout =8'ha3;

                                                    8'h0b:sbout =8'h9e;

                                                    8'h0c:sbout =8'h81;

                                                    8'h0d:sbout =8'hf3;

                                                    8'h0e:sbout =8'hd7;

                                                    8'h0f:sbout =8'hfb;

                                                    8'h10:sbout =8'h7c;

                                                    8'h11:sbout =8'he3;

                                                    8'h12:sbout =8'h39;
```

```
8'h13:sbout =8'h82;

8'h14:sbout =8'h9b;

8'h15:sbout =8'h2f;

8'h16:sbout =8'hff;

8'h17:sbout =8'h87;

8'h18:sbout =8'h34;

8'h19:sbout =8'h8e;

8'h1a:sbout =8'h43;

8'h1b:sbout =8'h44;

8'h1c:sbout =8'hc4;

8'h1d:sbout =8'hde;

8'h1e:sbout =8'he9;

8'h1f:sbout =8'hcb;

8'h20:sbout =8'h54;

8'h21:sbout =8'h7b;

8'h22:sbout =8'h94;

8'h23:sbout =8'h32;

8'h24:sbout =8'ha6;

8'h25:sbout =8'hc2;

8'h26:sbout =8'h23;

8'h27:sbout =8'h3d;

8'h28:sbout =8'hee;

8'h29:sbout =8'h4c;

8'h2a:sbout =8'h95;

8'h2b:sbout =8'h0b;

8'h2c:sbout =8'h42;

8'h2d:sbout =8'hfa;

8'h2e:sbout =8'hc3;

8'h2f:sbout =8'h4e;
```

```
8'h30:sbout =8'h08;

8'h31:sbout =8'h2e;

8'h32:sbout =8'ha1;

8'h33:sbout =8'h66;

8'h34:sbout =8'h28;

8'h35:sbout =8'hd9;

8'h36:sbout =8'h24;

8'h37:sbout =8'hb2;

8'h38:sbout =8'h76;

8'h39:sbout =8'h5b;

8'h3a:sbout =8'ha2;

8'h3b:sbout =8'h49;

8'h3c:sbout =8'h6d;

8'h3d:sbout =8'h8b;

8'h3e:sbout =8'hd1;

8'h3f:sbout =8'h25;

8'h40:sbout =8'h72;

8'h41:sbout =8'hf8;

8'h42:sbout =8'hf6;

8'h43:sbout =8'h64;

8'h44:sbout =8'h86;

8'h45:sbout =8'h68;

8'h46:sbout =8'h98;

8'h47:sbout =8'h16;

8'h48:sbout =8'hd4;

8'h49:sbout =8'ha4;

8'h4a:sbout =8'h5c;

8'h4b:sbout =8'hcc;

8'h4c:sbout =8'h5d;
```

```
8'h4d:sbout =8'h65;
8'h4e:sbout =8'hb6;
8'h4f:sbout =8'h92;
8'h50:sbout =8'h6c;
8'h51:sbout =8'h70;
8'h52:sbout =8'h48;
8'h53:sbout =8'h50;
8'h54:sbout =8'hfd;
8'h55:sbout =8'hed;
8'h56:sbout =8'hb9;
8'h57:sbout =8'hda;
8'h58:sbout =8'h5e;
8'h59:sbout =8'h15;
8'h5a:sbout =8'h46;
8'h5b:sbout =8'h57;
8'h5c:sbout =8'ha7;
8'h5d:sbout =8'h8d;
8'h5e:sbout =8'h9d;
8'h5f:sbout =8'h84;
8'h60:sbout =8'h90;
8'h61:sbout =8'hd8;
8'h62:sbout =8'hab;
8'h63:sbout =8'h00;
8'h64:sbout =8'h8c;
8'h65:sbout =8'hbc;
8'h66:sbout =8'hd3;
8'h67:sbout =8'h0a;
8'h68:sbout =8'hf7;
8'h69:sbout =8'he4;
```

```
8'h6a:sbout =8'h58;

8'h6b:sbout =8'h05;

8'h6c:sbout =8'hb8;

8'h6d:sbout =8'hb3;

8'h6e:sbout =8'h45;

8'h6f:sbout =8'h06;

8'h70:sbout =8'hd0;

8'h71:sbout =8'h2c;

8'h72:sbout =8'h1e;

8'h73:sbout =8'h8f;

8'h74:sbout =8'hca;

8'h75:sbout =8'h3f;

8'h76:sbout =8'h0f;

8'h77:sbout =8'h02;

8'h78:sbout =8'hc1;

8'h79:sbout =8'haf;

8'h7a:sbout =8'hbd;

8'h7b:sbout =8'h03;

8'h7c:sbout =8'h01;

8'h7d:sbout =8'h13;

8'h7e:sbout =8'h8a;

8'h7f:sbout =8'h6b;

8'h80:sbout =8'h3a;

8'h81:sbout =8'h91;

8'h82:sbout =8'h11;

8'h83:sbout =8'h41;

8'h84:sbout =8'h4f;

8'h85:sbout =8'h67;

8'h86:sbout =8'hdc;
```

```
8'h87:sbout =8'hea;

8'h88:sbout =8'h97;

8'h89:sbout =8'hf2;

8'h8a:sbout =8'hcf;

8'h8b:sbout =8'hce;

8'h8c:sbout =8'hf0;

8'h8d:sbout =8'hb4;

8'h8e:sbout =8'he6;

8'h8f:sbout =8'h73;

8'h90:sbout =8'h96;

8'h91:sbout =8'hac;

8'h92:sbout =8'h74;

8'h93:sbout =8'h22;

8'h94:sbout =8'he7;

8'h95:sbout =8'had;

8'h96:sbout =8'h35;

8'h97:sbout =8'h85;

8'h98:sbout =8'he2;

8'h99:sbout =8'hf9;

8'h9a:sbout =8'h37;

8'h9b:sbout =8'he8;

8'h9c:sbout =8'h1c;

8'h9d:sbout =8'h75;

8'h9e:sbout =8'hdf;

8'h9f:sbout =8'h6e;

8'ha0:sbout =8'h47;

8'ha1:sbout =8'hf1;

8'ha2:sbout =8'h1a;

8'ha3:sbout =8'h71;
```

```
8'hc1:sbout =8'hdd;

8'hc2:sbout =8'ha8;

8'hc3:sbout =8'h33;

8'hc4:sbout =8'h88;

8'hc5:sbout =8'h07;

8'hc6:sbout =8'hc7;

8'hc7:sbout =8'h31;

8'hc8:sbout =8'hb1;

8'hc9:sbout =8'h12;

8'hca:sbout =8'h10;

8'hcb:sbout =8'h59;

8'hcc:sbout =8'h27;

8'hcd:sbout =8'h80;

8'hce:sbout =8'hec;

8'hcf:sbout =8'h5f;

8'hd0:sbout =8'h60;

8'hd1:sbout =8'h51;

8'hd2:sbout =8'h7f;

8'hd3:sbout =8'ha9;

8'hd4:sbout =8'h19;

8'hd5:sbout =8'hb5;

8'hd6:sbout =8'h4a;

8'hd7:sbout =8'h0d;

8'hd8:sbout =8'h2d;

8'hd9:sbout =8'he5;

8'hda:sbout =8'h7a;

8'hdb:sbout =8'h9f;

8'hdc:sbout =8'h93;

8'hdd:sbout =8'hc9;
```

```
8'hde:sbout =8'h9c;

8'hdf:sbout =8'hef;

8'he0:sbout =8'ha0;

8'he1:sbout =8'he0;

8'he2:sbout =8'h3b;

8'he3:sbout =8'h4d;

8'he4:sbout =8'hae;

8'he5:sbout =8'h2a;

8'he6:sbout =8'hf5;

8'he7:sbout =8'hb0;

8'he8:sbout =8'hc8;

8'he9:sbout =8'heb;

8'hea:sbout =8'hbb;

8'heb:sbout =8'h3c;

8'hec:sbout =8'h83;

8'hed:sbout =8'h53;

8'hee:sbout =8'h99;

8'hef:sbout =8'h61;

8'hf0:sbout =8'h17;

8'hf1:sbout =8'h2b;

8'hf2:sbout =8'h04;

8'hf3:sbout =8'h7e;

8'hf4:sbout =8'hba;

8'hf5:sbout =8'h77;

8'hf6:sbout =8'hd6;

8'hf7:sbout =8'h26;

8'hf8:sbout =8'he1;

8'hf9:sbout =8'h69;

8'hfa:sbout =8'h14;
```

```
  assign shifted[80+:8] = in[16+:8];

  assign shifted[112+:8] = in[48+:8];


            // Fourth row (r = 3) is cyclically right shifted by 3 offsets

  assign shifted[24+:8] = in[56+:8];

  assign shifted[56+:8] = in[88+:8];

  assign shifted[88+:8] = in[120+:8];

  assign shifted[120+:8] = in[24+:8];


endmodule


// calls inverse sbox for reverse subtitution

module inverseSubBytes(in,out);

input [127:0] in;

output [127:0] out;


genvar i;

generate

for(i=0;i<128;i=i+8) begin :sub_Bytes

            inverseSbox s(in[i +:8],out[i +:8]);

            end

endgenerate

endmodule


module keyExpansion #(parameter nk=4,parameter nr=10)(key,w);

input [0 : (nk * 32) - 1] key;

output reg [0 : (128 * (nr + 1)) - 1] w;

reg [0:31] temp;

reg [0:31] r;
```

```
                                        8'hfb:sbout =8'h63;
                                        8'hfc:sbout =8'h55;
                                        8'hfd:sbout =8'h21;
                                        8'hfe:sbout =8'h0c;
                                        8'hff:sbout =8'h7d;
                                        endcase
end
endmodule

// inverse shift row
module inverseShiftRows (in, shifted);
          input [0:127] in;
          output [0:127] shifted;

          // First row (r = 0) is not shifted
          assign shifted[0+:8] = in[0+:8];
          assign shifted[32+:8] = in[32+:8];
          assign shifted[64+:8] = in[64+:8];
  assign shifted[96+:8] = in[96+:8];

          // Second row (r = 1) is cyclically right shifted by 1 offset
  assign shifted[8+:8] = in[104+:8];
  assign shifted[40+:8] = in[8+:8];
  assign shifted[72+:8] = in[40+:8];
  assign shifted[104+:8] = in[72+:8];

          // Third row (r = 2) is cyclically right shifted by 2 offsets
  assign shifted[16+:8] = in[80+:8];
  assign shifted[48+:8] = in[112+:8];
```

```
function [0:31] rotword;
input [0:31] x;
begin
                rotword={x[8:31],x[0:7]};
end
endfunction

function [0:31] subwordx;
input [0:31] a;
begin
subwordx[0:7]=c(a[0:7]);
subwordx[8:15]=c(a[8:15]);
subwordx[16:23]=c(a[16:23]);
subwordx[24:31]=c(a[24:31]);
end
endfunction

function [7:0] c(input [7:0] a);
begin
  case (a)
   8'h00: c=8'h63;
         8'h01: c=8'h7c;
         8'h02: c=8'h77;
         8'h03: c=8'h7b;
         8'h04: c=8'hf2;
         8'h05: c=8'h6b;
         8'h06: c=8'h6f;
         8'h07: c=8'hc5;
         8'h08: c=8'h30;
```

```
reg [0:31] rot; // It stores the returned value from the function rotword().

reg [0:31] x;      //It stores the returned value from the function subwordx().

reg [0:31] rconv; //It stores the returned value from the function rconx().

reg [0:31]new;


integer i;

always@* begin

//The first [(nk*32)-1 ]-bit key is stored in W.

        w = key;

        for(i = nk; i < 4*(nr + 1); i = i + 1) begin

        temp = w[(128 * (nr + 1) - 32) +: 32];

        if(i % nk == 0) begin

                rot = rotword(temp); // A call to the function rotword() is done and the returned value is
stored in rot.

                x = subwordx (rot);      //A call to the function subwordx() is done and the returned
value is stored in x.

                rconv = rconx (i/nk); //A call to the function rconx() is done and the returned value is
stored in rconv.

                temp = x ^ rconv;

        end

        else if(nk >6 && i % nk == 4) begin

                temp = subwordx(temp);

        end

        new = (w[(128*(nr+1)-(nk*32))+:32] ^ temp);

        // We would shift W by 32 bit to the left to add the new generated key word (new) at its end.

        w = w << 32;

        w = {w[0 : (128 * (nr + 1) - 32) - 1], new};

end

end
```

```
8'h09: c=8'h01;
8'h0a: c=8'h67;
8'h0b: c=8'h2b;
8'h0c: c=8'hfe;
8'h0d: c=8'hd7;
8'h0e: c=8'hab;
8'h0f: c=8'h76;
8'h10: c=8'hca;
8'h11: c=8'h82;
8'h12: c=8'hc9;
8'h13: c=8'h7d;
8'h14: c=8'hfa;
8'h15: c=8'h59;
8'h16: c=8'h47;
8'h17: c=8'hf0;
8'h18: c=8'had;
8'h19: c=8'hd4;
8'h1a: c=8'ha2;
8'h1b: c=8'haf;
8'h1c: c=8'h9c;
8'h1d: c=8'ha4;
8'h1e: c=8'h72;
8'h1f: c=8'hc0;
8'h20: c=8'hb7;
8'h21: c=8'hfd;
8'h22: c=8'h93;
8'h23: c=8'h26;
8'h24: c=8'h36;
8'h25: c=8'h3f;
```

```
8'h26: c=8'hf7;

8'h27: c=8'hcc;

8'h28: c=8'h34;

8'h29: c=8'ha5;

8'h2a: c=8'he5;

8'h2b: c=8'hf1;

8'h2c: c=8'h71;

8'h2d: c=8'hd8;

8'h2e: c=8'h31;

8'h2f: c=8'h15;

8'h30: c=8'h04;

8'h31: c=8'hc7;

8'h32: c=8'h23;

8'h33: c=8'hc3;

8'h34: c=8'h18;

8'h35: c=8'h96;

8'h36: c=8'h05;

8'h37: c=8'h9a;

8'h38: c=8'h07;

8'h39: c=8'h12;

8'h3a: c=8'h80;

8'h3b: c=8'he2;

8'h3c: c=8'heb;

8'h3d: c=8'h27;

8'h3e: c=8'hb2;

8'h3f: c=8'h75;

8'h40: c=8'h09;

8'h41: c=8'h83;

8'h42: c=8'h2c;
```

```
8'h43: c=8'h1a;

8'h44: c=8'h1b;

8'h45: c=8'h6e;

8'h46: c=8'h5a;

8'h47: c=8'ha0;

8'h48: c=8'h52;

8'h49: c=8'h3b;

8'h4a: c=8'hd6;

8'h4b: c=8'hb3;

8'h4c: c=8'h29;

8'h4d: c=8'he3;

8'h4e: c=8'h2f;

8'h4f: c=8'h84;

8'h50: c=8'h53;

8'h51: c=8'hd1;

8'h52: c=8'h00;

8'h53: c=8'hed;

8'h54: c=8'h20;

8'h55: c=8'hfc;

8'h56: c=8'hb1;

8'h57: c=8'h5b;

8'h58: c=8'h6a;

8'h59: c=8'hcb;

8'h5a: c=8'hbe;

8'h5b: c=8'h39;

8'h5c: c=8'h4a;

8'h5d: c=8'h4c;

8'h5e: c=8'h58;

8'h5f: c=8'hcf;
```

```
8'h60: c=8'hd0;

8'h61: c=8'hef;

8'h62: c=8'haa;

8'h63: c=8'hfb;

8'h64: c=8'h43;

8'h65: c=8'h4d;

8'h66: c=8'h33;

8'h67: c=8'h85;

8'h68: c=8'h45;

8'h69: c=8'hf9;

8'h6a: c=8'h02;

8'h6b: c=8'h7f;

8'h6c: c=8'h50;

8'h6d: c=8'h3c;

8'h6e: c=8'h9f;

8'h6f: c=8'ha8;

8'h70: c=8'h51;

8'h71: c=8'ha3;

8'h72: c=8'h40;

8'h73: c=8'h8f;

8'h74: c=8'h92;

8'h75: c=8'h9d;

8'h76: c=8'h38;

8'h77: c=8'hf5;

8'h78: c=8'hbc;

8'h79: c=8'hb6;

8'h7a: c=8'hda;

8'h7b: c=8'h21;

8'h7c: c=8'h10;
```

```
8'h7d: c=8'hff;

8'h7e: c=8'hf3;

8'h7f: c=8'hd2;

8'h80: c=8'hcd;

8'h81: c=8'h0c;

8'h82: c=8'h13;

8'h83: c=8'hec;

8'h84: c=8'h5f;

8'h85: c=8'h97;

8'h86: c=8'h44;

8'h87: c=8'h17;

8'h88: c=8'hc4;

8'h89: c=8'ha7;

8'h8a: c=8'h7e;

8'h8b: c=8'h3d;

8'h8c: c=8'h64;

8'h8d: c=8'h5d;

8'h8e: c=8'h19;

8'h8f: c=8'h73;

8'h90: c=8'h60;

8'h91: c=8'h81;

8'h92: c=8'h4f;

8'h93: c=8'hdc;

8'h94: c=8'h22;

8'h95: c=8'h2a;

8'h96: c=8'h90;

8'h97: c=8'h88;

8'h98: c=8'h46;

8'h99: c=8'hee;
```

```
8'h9a: c=8'hb8;

8'h9b: c=8'h14;

8'h9c: c=8'hde;

8'h9d: c=8'h5e;

8'h9e: c=8'h0b;

8'h9f: c=8'hdb;

8'ha0: c=8'he0;

8'ha1: c=8'h32;

8'ha2: c=8'h3a;

8'ha3: c=8'h0a;

8'ha4: c=8'h49;

8'ha5: c=8'h06;

8'ha6: c=8'h24;

8'ha7: c=8'h5c;

8'ha8: c=8'hc2;

8'ha9: c=8'hd3;

8'haa: c=8'hac;

8'hab: c=8'h62;

8'hac: c=8'h91;

8'had: c=8'h95;

8'hae: c=8'he4;

8'haf: c=8'h79;

8'hb0: c=8'he7;

8'hb1: c=8'hc8;

8'hb2: c=8'h37;

8'hb3: c=8'h6d;

8'hb4: c=8'h8d;

8'hb5: c=8'hd5;

8'hb6: c=8'h4e;
```

```
8'hb7: c=8'ha9;

8'hb8: c=8'h6c;

8'hb9: c=8'h56;

8'hba: c=8'hf4;

8'hbb: c=8'hea;

8'hbc: c=8'h65;

8'hbd: c=8'h7a;

8'hbe: c=8'hae;

8'hbf: c=8'h08;

8'hc0: c=8'hba;

8'hc1: c=8'h78;

8'hc2: c=8'h25;

8'hc3: c=8'h2e;

8'hc4: c=8'h1c;

8'hc5: c=8'ha6;

8'hc6: c=8'hb4;

8'hc7: c=8'hc6;

8'hc8: c=8'he8;

8'hc9: c=8'hdd;

8'hca: c=8'h74;

8'hcb: c=8'h1f;

8'hcc: c=8'h4b;

8'hcd: c=8'hbd;

8'hce: c=8'h8b;

8'hcf: c=8'h8a;

8'hd0: c=8'h70;

8'hd1: c=8'h3e;

8'hd2: c=8'hb5;

8'hd3: c=8'h66;
```

```
8'hd4: c=8'h48;

8'hd5: c=8'h03;

8'hd6: c=8'hf6;

8'hd7: c=8'h0e;

8'hd8: c=8'h61;

8'hd9: c=8'h35;

8'hda: c=8'h57;

8'hdb: c=8'hb9;

8'hdc: c=8'h86;

8'hdd: c=8'hc1;

8'hde: c=8'h1d;

8'hdf: c=8'h9e;

8'he0: c=8'he1;

8'he1: c=8'hf8;

8'he2: c=8'h98;

8'he3: c=8'h11;

8'he4: c=8'h69;

8'he5: c=8'hd9;

8'he6: c=8'h8e;

8'he7: c=8'h94;

8'he8: c=8'h9b;

8'he9: c=8'h1e;

8'hea: c=8'h87;

8'heb: c=8'he9;

8'hec: c=8'hce;

8'hed: c=8'h55;

8'hee: c=8'h28;

8'hef: c=8'hdf;

8'hf0: c=8'h8c;
```

```
                8'hf1: c=8'ha1;

                8'hf2: c=8'h89;

                8'hf3: c=8'h0d;

                8'hf4: c=8'hbf;

                8'hf5: c=8'he6;

                8'hf6: c=8'h42;

                8'hf7: c=8'h68;

                8'hf8: c=8'h41;

                8'hf9: c=8'h99;

                8'hfa: c=8'h2d;

                8'hfb: c=8'h0f;

                8'hfc: c=8'hb0;

                8'hfd: c=8'h54;

                8'hfe: c=8'hbb;

                8'hff: c=8'h16;

            endcase

end

endfunction


function[0:31] rconx;

input [0:31] r;

begin

 case(r)

   4'h1: rconx=32'h01000000;

   4'h2: rconx=32'h02000000;

   4'h3: rconx=32'h04000000;

   4'h4: rconx=32'h08000000;

   4'h5: rconx=32'h10000000;

   4'h6: rconx=32'h20000000;
```

```
  4'h7: rconx=32'h40000000;
  4'h8: rconx=32'h80000000;
  4'h9: rconx=32'h1b000000;
  4'ha: rconx=32'h36000000;
  default: rconx=32'h00000000;
 endcase
 end
endfunction
endmodule

//used for mixcolumn
module mixColumns(state_in,state_out);
input [127:0] state_in;
output[127:0] state_out;
function [7:0] mb2; //multiply by 2
	input [7:0] x;
	begin
			if(x[7] == 1) mb2 = ((x << 1) ^ 8'h1b);
			else mb2 = x << 1;
	end
endfunction

function [7:0] mb3; //multiply by 3
	input [7:0] x;
	begin
			mb3 = mb2(x) ^ x;
	end
endfunction
```

```
genvar i;

generate

for(i=0;i< 4;i=i+1) begin : m_col

            assign state_out[(i*32 + 24)+:8]= mb2(state_in[(i*32 + 24)+:8]) ^ mb3(state_in[(i*32 + 16)+:8]) ^
state_in[(i*32 + 8)+:8] ^ state_in[i*32+:8];

            assign state_out[(i*32 + 16)+:8]= state_in[(i*32 + 24)+:8] ^ mb2(state_in[(i*32 + 16)+:8]) ^
mb3(state_in[(i*32 + 8)+:8]) ^ state_in[i*32+:8];

            assign state_out[(i*32 + 8)+:8]= state_in[(i*32 + 24)+:8] ^ state_in[(i*32 + 16)+:8] ^
mb2(state_in[(i*32 + 8)+:8]) ^ mb3(state_in[i*32+:8]);

   assign state_out[i*32+:8]= mb3(state_in[(i*32 + 24)+:8]) ^ state_in[(i*32 + 16)+:8] ^ state_in[(i*32 +
8)+:8] ^ mb2(state_in[i*32+:8]);

end

endgenerate

endmodule


// used for subtituting the elements from sbox

module sbox(a,c);


input  [7:0] a;

output [7:0] c;

reg [7:0] c;

    always @(a)

   case (a)

     8'h00: c=8'h63;

            8'h01: c=8'h7c;

            8'h02: c=8'h77;

            8'h03: c=8'h7b;

            8'h04: c=8'hf2;

            8'h05: c=8'h6b;

            8'h06: c=8'h6f;
```

```
8'h07: c=8'hc5;

8'h08: c=8'h30;

8'h09: c=8'h01;

8'h0a: c=8'h67;

8'h0b: c=8'h2b;

8'h0c: c=8'hfe;

8'h0d: c=8'hd7;

8'h0e: c=8'hab;

8'h0f: c=8'h76;

8'h10: c=8'hca;

8'h11: c=8'h82;

8'h12: c=8'hc9;

8'h13: c=8'h7d;

8'h14: c=8'hfa;

8'h15: c=8'h59;

8'h16: c=8'h47;

8'h17: c=8'hf0;

8'h18: c=8'had;

8'h19: c=8'hd4;

8'h1a: c=8'ha2;

8'h1b: c=8'haf;

8'h1c: c=8'h9c;

8'h1d: c=8'ha4;

8'h1e: c=8'h72;

8'h1f: c=8'hc0;

8'h20: c=8'hb7;

8'h21: c=8'hfd;

8'h22: c=8'h93;

8'h23: c=8'h26;
```

```
8'h24: c=8'h36;

8'h25: c=8'h3f;

8'h26: c=8'hf7;

8'h27: c=8'hcc;

8'h28: c=8'h34;

8'h29: c=8'ha5;

8'h2a: c=8'he5;

8'h2b: c=8'hf1;

8'h2c: c=8'h71;

8'h2d: c=8'hd8;

8'h2e: c=8'h31;

8'h2f: c=8'h15;

8'h30: c=8'h04;

8'h31: c=8'hc7;

8'h32: c=8'h23;

8'h33: c=8'hc3;

8'h34: c=8'h18;

8'h35: c=8'h96;

8'h36: c=8'h05;

8'h37: c=8'h9a;

8'h38: c=8'h07;

8'h39: c=8'h12;

8'h3a: c=8'h80;

8'h3b: c=8'he2;

8'h3c: c=8'heb;

8'h3d: c=8'h27;

8'h3e: c=8'hb2;

8'h3f: c=8'h75;

8'h40: c=8'h09;
```

```
8'h41: c=8'h83;
8'h42: c=8'h2c;
8'h43: c=8'h1a;
8'h44: c=8'h1b;
8'h45: c=8'h6e;
8'h46: c=8'h5a;
8'h47: c=8'ha0;
8'h48: c=8'h52;
8'h49: c=8'h3b;
8'h4a: c=8'hd6;
8'h4b: c=8'hb3;
8'h4c: c=8'h29;
8'h4d: c=8'he3;
8'h4e: c=8'h2f;
8'h4f: c=8'h84;
8'h50: c=8'h53;
8'h51: c=8'hd1;
8'h52: c=8'h00;
8'h53: c=8'hed;
8'h54: c=8'h20;
8'h55: c=8'hfc;
8'h56: c=8'hb1;
8'h57: c=8'h5b;
8'h58: c=8'h6a;
8'h59: c=8'hcb;
8'h5a: c=8'hbe;
8'h5b: c=8'h39;
8'h5c: c=8'h4a;
8'h5d: c=8'h4c;
```

```
8'h5e: c=8'h58;

8'h5f: c=8'hcf;

8'h60: c=8'hd0;

8'h61: c=8'hef;

8'h62: c=8'haa;

8'h63: c=8'hfb;

8'h64: c=8'h43;

8'h65: c=8'h4d;

8'h66: c=8'h33;

8'h67: c=8'h85;

8'h68: c=8'h45;

8'h69: c=8'hf9;

8'h6a: c=8'h02;

8'h6b: c=8'h7f;

8'h6c: c=8'h50;

8'h6d: c=8'h3c;

8'h6e: c=8'h9f;

8'h6f: c=8'ha8;

8'h70: c=8'h51;

8'h71: c=8'ha3;

8'h72: c=8'h40;

8'h73: c=8'h8f;

8'h74: c=8'h92;

8'h75: c=8'h9d;

8'h76: c=8'h38;

8'h77: c=8'hf5;

8'h78: c=8'hbc;

8'h79: c=8'hb6;

8'h7a: c=8'hda;
```

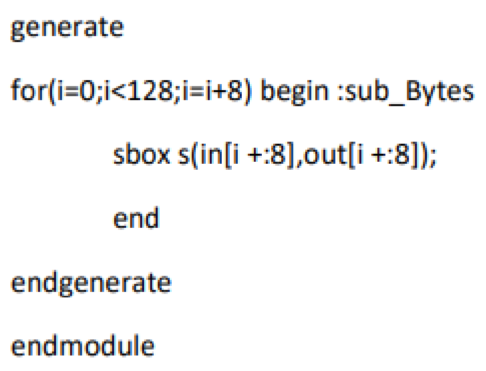
```
8'h7b: c=8'h21;
8'h7c: c=8'h10;
8'h7d: c=8'hff;
8'h7e: c=8'hf3;
8'h7f: c=8'hd2;
8'h80: c=8'hcd;
8'h81: c=8'h0c;
8'h82: c=8'h13;
8'h83: c=8'hec;
8'h84: c=8'h5f;
8'h85: c=8'h97;
8'h86: c=8'h44;
8'h87: c=8'h17;
8'h88: c=8'hc4;
8'h89: c=8'ha7;
8'h8a: c=8'h7e;
8'h8b: c=8'h3d;
8'h8c: c=8'h64;
8'h8d: c=8'h5d;
8'h8e: c=8'h19;
8'h8f: c=8'h73;
8'h90: c=8'h60;
8'h91: c=8'h81;
8'h92: c=8'h4f;
8'h93: c=8'hdc;
8'h94: c=8'h22;
8'h95: c=8'h2a;
8'h96: c=8'h90;
8'h97: c=8'h88;
```

```
8'h98: c=8'h46;
8'h99: c=8'hee;
8'h9a: c=8'hb8;
8'h9b: c=8'h14;
8'h9c: c=8'hde;
8'h9d: c=8'h5e;
8'h9e: c=8'h0b;
8'h9f: c=8'hdb;
8'ha0: c=8'he0;
8'ha1: c=8'h32;
8'ha2: c=8'h3a;
8'ha3: c=8'h0a;
8'ha4: c=8'h49;
8'ha5: c=8'h06;
8'ha6: c=8'h24;
8'ha7: c=8'h5c;
8'ha8: c=8'hc2;
8'ha9: c=8'hd3;
8'haa: c=8'hac;
8'hab: c=8'h62;
8'hac: c=8'h91;
8'had: c=8'h95;
8'hae: c=8'he4;
8'haf: c=8'h79;
8'hb0: c=8'he7;
8'hb1: c=8'hc8;
8'hb2: c=8'h37;
8'hb3: c=8'h6d;
8'hb4: c=8'h8d;
```

```
8'hb5: c=8'hd5;
8'hb6: c=8'h4e;
8'hb7: c=8'ha9;
8'hb8: c=8'h6c;
8'hb9: c=8'h56;
8'hba: c=8'hf4;
8'hbb: c=8'hea;
8'hbc: c=8'h65;
8'hbd: c=8'h7a;
8'hbe: c=8'hae;
8'hbf: c=8'h08;
8'hc0: c=8'hba;
8'hc1: c=8'h78;
8'hc2: c=8'h25;
8'hc3: c=8'h2e;
8'hc4: c=8'h1c;
8'hc5: c=8'ha6;
8'hc6: c=8'hb4;
8'hc7: c=8'hc6;
8'hc8: c=8'he8;
8'hc9: c=8'hdd;
8'hca: c=8'h74;
8'hcb: c=8'h1f;
8'hcc: c=8'h4b;
8'hcd: c=8'hbd;
8'hce: c=8'h8b;
8'hcf: c=8'h8a;
8'hd0: c=8'h70;
8'hd1: c=8'h3e;
```

```
8'hd2: c=8'hb5;

8'hd3: c=8'h66;

8'hd4: c=8'h48;

8'hd5: c=8'h03;

8'hd6: c=8'hf6;

8'hd7: c=8'h0e;

8'hd8: c=8'h61;

8'hd9: c=8'h35;

8'hda: c=8'h57;

8'hdb: c=8'hb9;

8'hdc: c=8'h86;

8'hdd: c=8'hc1;

8'hde: c=8'h1d;

8'hdf: c=8'h9e;

8'he0: c=8'he1;

8'he1: c=8'hf8;

8'he2: c=8'h98;

8'he3: c=8'h11;

8'he4: c=8'h69;

8'he5: c=8'hd9;

8'he6: c=8'h8e;

8'he7: c=8'h94;

8'he8: c=8'h9b;

8'he9: c=8'h1e;

8'hea: c=8'h87;

8'heb: c=8'he9;

8'hec: c=8'hce;

8'hed: c=8'h55;

8'hee: c=8'h28;
```

```
                8'hef: c=8'hdf;
                8'hf0: c=8'h8c;
                8'hf1: c=8'ha1;
                8'hf2: c=8'h89;
                8'hf3: c=8'h0d;
                8'hf4: c=8'hbf;
                8'hf5: c=8'he6;
                8'hf6: c=8'h42;
                8'hf7: c=8'h68;
                8'hf8: c=8'h41;
                8'hf9: c=8'h99;
                8'hfa: c=8'h2d;
                8'hfb: c=8'h0f;
                8'hfc: c=8'hb0;
                8'hfd: c=8'h54;
                8'hfe: c=8'hbb;
                8'hff: c=8'h16;
            endcase

endmodule


// shifting the rows
module shiftRows (in, shifted);
            input [0:127] in;
            output [0:127] shifted;

            // First row (r = 0) is not shifted
            assign shifted[0+:8] = in[0+:8];
            assign shifted[32+:8] = in[32+:8];
            assign shifted[64+:8] = in[64+:8];
   assign shifted[96+:8] = in[96+:8];

            // Second row (r = 1) is cyclically left shifted by 1 offset
   assign shifted[8+:8] = in[40+:8];
   assign shifted[40+:8] = in[72+:8];
   assign shifted[72+:8] = in[104+:8];
   assign shifted[104+:8] = in[8+:8];

            // Third row (r = 2) is cyclically left shifted by 2 offsets
   assign shifted[16+:8] = in[80+:8];
   assign shifted[48+:8] = in[112+:8];
   assign shifted[80+:8] = in[16+:8];
   assign shifted[112+:8] = in[48+:8];

            // Fourth row (r = 3) is cyclically left shifted by 3 offsets
   assign shifted[24+:8] = in[120+:8];
   assign shifted[56+:8] = in[24+:8];
   assign shifted[88+:8] = in[56+:8];
   assign shifted[120+:8] = in[88+:8];

endmodule

// taking 128 byte number and calling sbox for subtitution
module subBytes(in,out);
input [127:0] in;
output [127:0] out;

genvar i;
```



```
generate
for(i=0;i<128;i=i+8) begin :sub_Bytes
        sbox s(in[i +:8],out[i +:8]);
        end
endgenerate
endmodule
```

TEST BENCH

```
module AES_tb();
wire e128, d128, e192, d192, e256, d256;
wire [127:0] encrypted128, encrypted192, encrypted256;
wire [127:0] decrypted128, decrypted192, decrypted256;
reg enable;
reg [127:0]in;
reg [127:0]secret_key;

encrypt a(enable, in,secret_key,e128, d128, e192, d192, e256,
d256,encrypted128,encrypted192,encrypted256,decrypted128,decrypted192,decrypted256);
initial begin
$monitor("Encrypt128 = %b, Decrypt128 = %b, Encrypt192 = %b, Decrypt192 = %b, Encrypt256
= %b, Decrypt256 = %b",
e128, d128, e192, d192, e256, d256);
// Turning on enable to check that all tests passed
enable = 1;
#10;
// Turning off enable to check if the leds turn off
```

```
enable = 0;
#10;
// Turning on enable to check if the leds turn on again
enable = 1;
in = 128'h_00112233445566778899aabbccddeeff;
secret_key=128'h_000102030405060708090a0b0c0d0e0f;
#10;
end
endmodule
```

SCHEMATIC REPRESENTATION

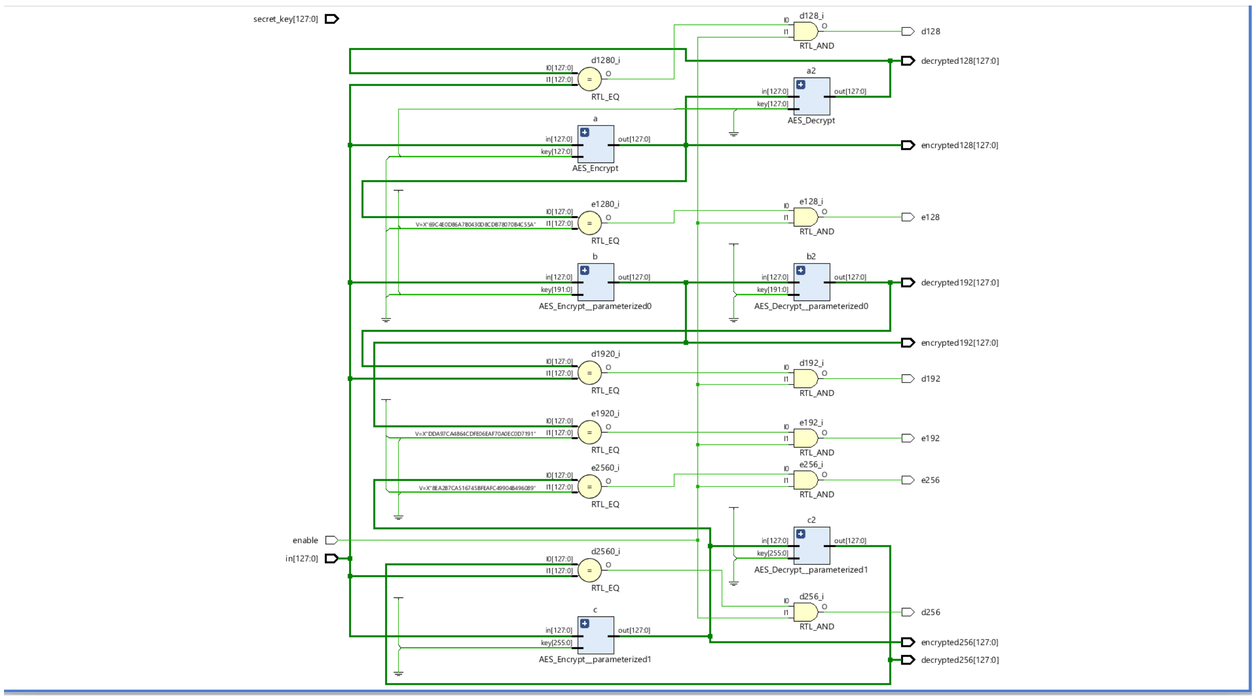


Fig 13. Schematic of AES Algorithm

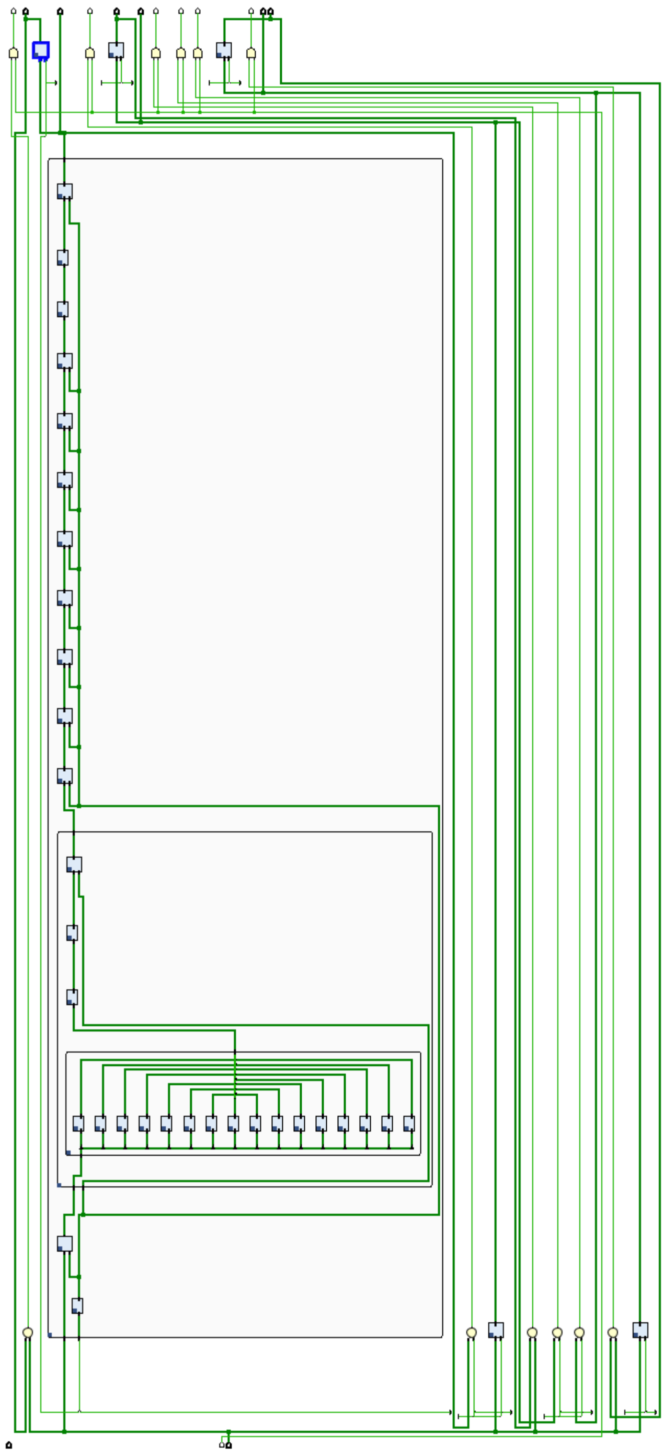


Fig 14. Expansion of the AES Algorithm Schematic

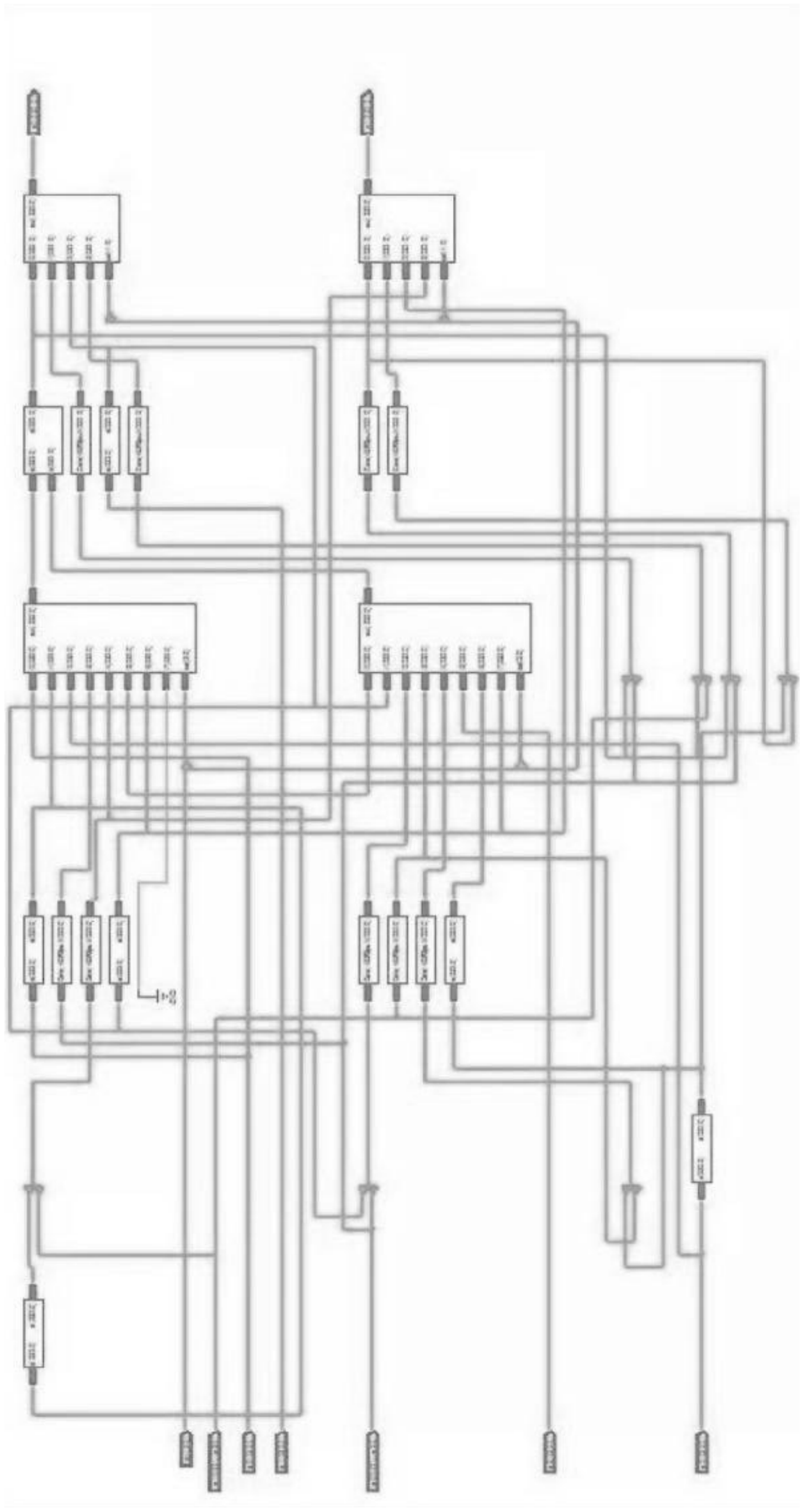
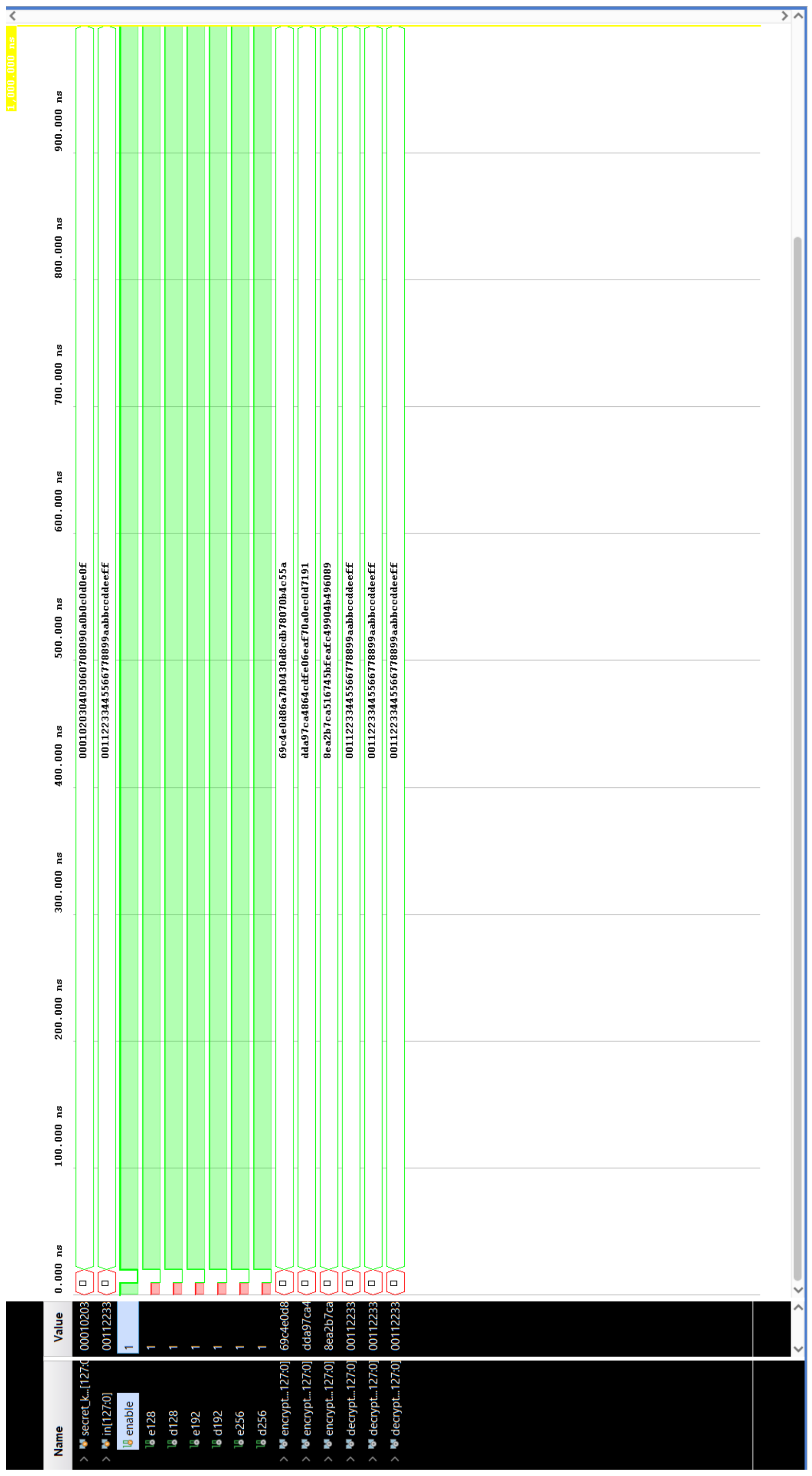


Fig 15. Schematic of ECC Algorithm

SIMULATION



```
Name                Value
secret_k...[127:0]  00010203   000102030405060708090a0b0c0d0e0f
in[127:0]           00112233   00112233445566778899aabbccddeeff
enable              1
e128                1
d128                1
e192                1
d192                1
e256                1
d256                1
encrypt...127:0]    69c4e0d8   69c4e0d86a7b0430d8cdb78070b4c55a
encrypt...127:0]    dda97ca4   dda97ca4864cdfe06eaf70a0ec0d7191
encrypt...127:0]    8ea2b7ca   8ea2b7ca516745bfeafc49904b496089
decrypt...127:0]    00112233   00112233445566778899aabbccddeeff
decrypt...127:0]    00112233   00112233445566778899aabbccddeeff
decrypt...127:0]    00112233   00112233445566778899aabbccddeeff
```

Fig 16. Simulation

APPLICATIONS

Elliptic Curve Cryptography is very useful in implementing highly secure encryptions. The information database of the Government offices or bank details such as credit and debit card information, netbanking, account transactions, etc., can be encrypted using ECC.
Secure communication: ECC is commonly used in secure communication protocols, such as SSL/TLS, SSH, and IPsec. It provides efficient and robust encryption and authentication mechanisms for secure communication over networks.
Digital signatures: ECC is used to create digital signatures, which are used to verify the authenticity and integrity of digital documents and transactions. Digital signatures are widely used in e-commerce, online banking, and other online applications.
Cryptocurrency: ECC is used in several cryptocurrency systems, such as Bitcoin, to provide secure transactions and to protect against attacks such as double-spending.

FUTURE SCOPE OF ELLIPTICAL CURVE CRYPTOGRAPHY

Elliptic curve cryptography (ECC) is a type of public-key cryptography that is based on the mathematical properties of elliptic curves. It has several advantages over other forms of cryptography, including smaller key sizes, faster computations, and stronger security.
In the future, ECC is likely to see increased use in a variety of applications. Here are some potential areas of growth for elliptic curve cryptography:
Internet of Things (IoT) Security: As the number of IoT devices grows, the need for secure communication between them will become more critical. ECC's ability to provide strong encryption with smaller key sizes makes it an attractive choice for securing IoT devices.
Blockchain and Cryptocurrency: Many cryptocurrencies, such as Bitcoin and Ethereum, already use ECC for key generation and signing transactions. As the use of cryptocurrencies and blockchain technology continues to grow, so too will the demand for ECC.
Post-quantum Cryptography: One of the biggest threats to current cryptographic systems is the development of quantum computers. ECC is one of the few cryptographic systems that is

believed to be resistant to attacks by quantum computers. As such, it is likely to play a prominent role in post-quantum cryptography.
Cloud Computing: As more businesses move their operations to the cloud, the need for secure communication and data storage becomes increasingly important. ECC's smaller key sizes and faster computations make it a good choice for secure cloud computing.
Overall, the future looks bright for elliptic curve cryptography, with many potential applications and areas for growth.

CONCLUSION

Elliptic curve cryptography (ECC) is a powerful cryptographic technique that offers several advantages over traditional cryptographic methods. In this project, we explored the key concepts of ECC and how it can be used for secure communication. We implemented a basic ECC algorithm in software and tested it using different inputs.
Our experiments showed that ECC offers several benefits over other cryptographic techniques, including smaller key sizes, faster computations, and better resistance to attacks such as brute-force and side-channel attacks. We also discussed some of the challenges of implementing ECC in hardware and how these challenges can be addressed.
Overall, our project demonstrates the effectiveness of ECC for secure communication and highlights the potential of this technology for future applications in the fields of IoT, mobile devices, and cloud computing. Further research and development are needed to address some of the remaining challenges of ECC and to improve its scalability and efficiency.

REFERENCES

https://avinetworks.com/glossary/elliptic-curve-cryptography/
https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/
https://youtu.be/pZ1qD0n8NC4
Learn Public Key Cryptography in just 18 Minutes - Cryptography Tutorial - YouTube

Elliptic Curve #3: Example of Point Doubling and Point Addition - YouTube

Elliptic curve #4: Double-and-Add Algorithm - YouTube

https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard#:~:text=The%20Advanced%20Encryption%20Standard%20(AES)%20is%20a%20symmetric%20block%20cipher,cybersecurity%20and%20electronic%20data%20protection

https://www.ripublication.com/ijaer17/ijaerv12n19_140.pdf