**Voting Machine Authentication and Security with AES and SHA Algorithm using Verilog**

Submitted in partial fulfillment of the requirements

of the degree of

**Bachelor of Technology**

By

Sai Rohith - 21ECB0B58

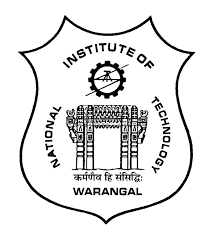
Anusha - 21ECB0B59

Bhavya - 21ECB0B60

Guided by:

**Dr. P. Prithvi Dr V. Narendar**

**Assistant Professor Assistant Professor**



**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

**NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL**

**2022-2023**

**CERTIFICATE**

This is to certify that the dissertation work entitled **VOTING MACHINE AUTHENTICATION AND SECURITY WITH AES AND SHA ALGORITHMS USING VERILOG** is a bonafied record of work carried out work by Sai Rohith (21ECB0B58) , Anusha (21ECB0B59) and Tanneru Bhavya (21ECB0B60) submitted to 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 academic year (2022-2023).

**Dr. P. Prithvi Dr. V. Narendar**

Assistant Professor Assistant Professor

Department of Electronics and Department of Electronics and

Communication Engineering Communication Engineering

National Institute of Technology National Institute of Technology

Warangal Warangal

**ACKNOWLEDGEMENT**

I would like to express my deepest gratitude to my faculty in-charge **Dr. P. Prithvi,** **Assistant Professor**, Department of Electronics and Communication Engineering and **Dr. V. Narendar**, **Assistant Professor**, Department of Electronics and Communication Engineering, National Institute of Technology, Warangal for their constant supervision, guidance, suggestion and encouragement during this semester.

**TABLE OF CONTENT:**

* Abstract
* Working Principle
* Block diagrams
* Introduction
  + - SHA-256 Algorithm
    - AES algorithm
* Verilog Codes
* Simulations
  + - Waveform
    - Schematic
    - FPGA Implementation
* Conclusion
* Further Works

**ABSTRACT:**

Elections is one such area where authentication and security is concerned. The AES and SHA algorithms can be used together to securely store and verify passwords in a voting machine. Each voter is assigned a unique password that is hashed using SHA-256 and encrypted using AES with a secret key. The encrypted password is stored in a secure database. When a voter approaches the voting machine, they are prompted to enter their password. The entered password is hashed using SHA-256 and encrypted using AES with the same secret key. The encrypted entered password is compared to the stored encrypted password. If they match, the voter is authorized to use the voting machine and cast their vote. This algorithm ensures that only authorized voters can use the voting machine, and their passwords are securely stored and verified. Using a single password increases the vulnerability of the system because with a one-password scheme, there is a high risk of passwords being tampered since they are exposed to many different parties. Cryptosystems are hence vital to be incorporated into a smart card to shield users against attackers.

**INTRODUCTION**

**Working principle**

**1.Voter Enrollment**:

During the voter enrollment process, the first step is for the voter to create their own password. This password should be unique and strong enough to protect against unauthorized access. The password should not be shared with anyone else. The password is entered by the voter on the registration form and stored securely in the database.

**2.Salting and Hashing with SHA256**:

The password is then salted and hashed using the SHA256 algorithm. Salting is a technique where a random value, called a salt (24 value bit is used here), is added to the password before hashing. This makes it harder for attackers to crack the password using precomputed rainbow tables. The salt is stored in the database along with the hashed password.

**3.Encryption with AES128**:

The hashed password and salt are then encrypted using the AES128 algorithm. Encryption is a process of converting plain text data into a secure format that can only be decrypted by authorized parties. The encrypted data is then stored securely in the database.

**4.Voter Verification**:

When the voter goes to the polling booth, they are asked to enter their password. The password is then hashed using the SHA256 algorithm and salted with the same salt used during the enrollment process.

**5.Decryption with AES128**:

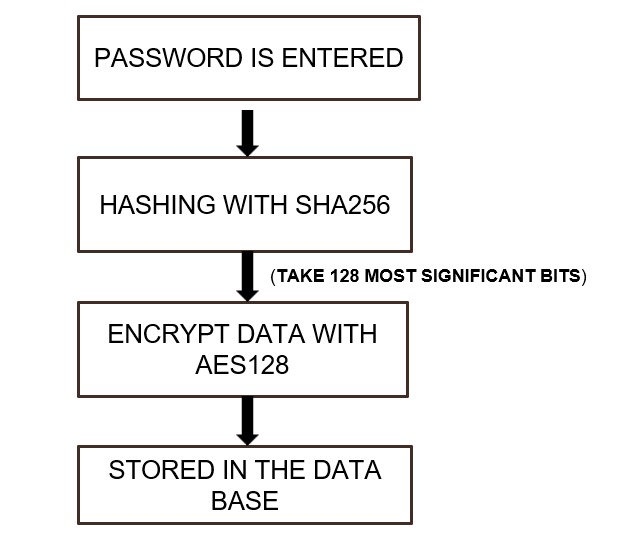
The hashed and salted password is then encrypted using the AES128 algorithm. The resulting encrypted data is compared to the encrypted password and salt stored in the database.

**6.Voting Access**:

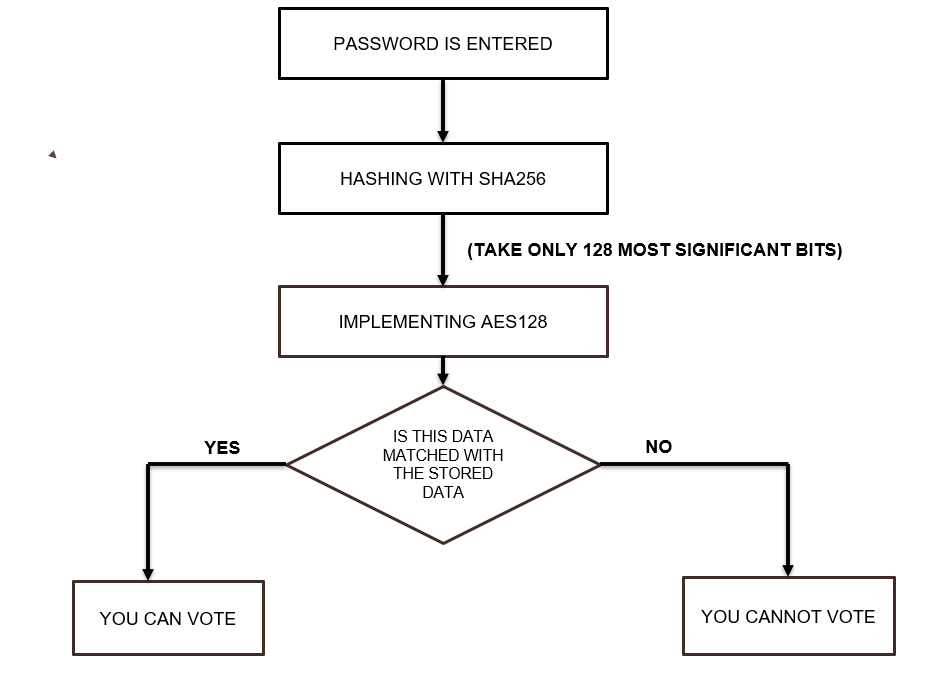
If the encrypted data matches, the voter is allowed to cast their vote. If the encrypted data does not match, the voter is denied access to vote.

During the voting process, the voter's password is hashed and salted with the same salt used during enrollment, then encrypted with AES128, and compared to the encrypted password stored in the database. If the encrypted data matches, the voter is allowed to cast their vote.

**Enrollment Block Diagram**



**Password Verification Block Diagram**

****

**SHA-256 Algorithm**

The SHA-256 algorithm is a widely used cryptographic hash function that is designed to provide a high level of security and data integrity. It was developed by the US National Security Agency (NSA) and is a part of the SHA-2 family of hash functions, which includes other variants such as SHA-224, SHA-384, and SHA-512.

The term "hash" refers to a one-way function that takes an input (such as a message or a file) of arbitrary length and produces a fixed-length output, called a hash value or digest. The SHA-256 algorithm produces a 256-bit hash value, which makes it highly resistant to collisions and brute-force attacks.

The SHA-256 algorithm works by taking the input message and breaking it down into fixed-size chunks, which are then processed using a series of mathematical operations that scramble the data and produce a unique hash value. The algorithm also includes a message padding process to ensure that the input message is a multiple of a certain block size.

The security properties of the SHA-256 algorithm make it ideal for use in applications such as digital signatures, password storage, and data integrity verification. It provides confidentiality, integrity, and authenticity of the data, ensuring that the data cannot be tampered with or modified without detection.

Overall, the SHA-256 algorithm is a powerful and widely used cryptographic hash function that provides a high level of security and data integrity. Its strength and versatility make it an essential tool for ensuring the security of data in a wide range of applications.

**ALGORITHM DESCRIPTION:**

The **SHA-256 algorithm** is a cryptographic hash function that takes an input message of any length and produces a 256-bit hash value as output. It uses a series of steps, including message padding, message scheduling, and a compression function, to transform the input message into a fixed-length output.

When some plain text data is entered by the user,based on the ascii table the data is going to get converted into pure binary form because this is the only language the computer understands.

For example, if we take the word “FPGA”, by using the equivalent ascii values as follows

F = 01000110

P = 01010000

G = 01000111

A = 01000001

We get our binary form of the word “FPGA” as

01000110 01010000 01000111 01000001.

**Message Padding:**

The purpose of padding is to ensure that the padded message is a multiple of 512 bits. Padding can be inserted before hash computation begins on a message or at any other time during the hash computation prior to processing the block that will contain the padding.

Suppose that the length of the message M is l bits. Append the bit “1” to the end of the message followed by k zero bits, where k is the smallest non-negative solution to the equation

l+1+k=448mod512.Then append the 64-bit block that is equal to

the number l expressed using a binary representation. For example, the (8-bit ASCII) message “**abc**” has length 8\*3=24, so the message is padded with a one bit, then 448-(24+1)=423 zero bits and then the message length to become the 512-bit padded message.



The length of the padded message should now be a multiple of 512 bits.

**Parsing the message:**

The message and its padding must be parsed into N m-bit blocks.

For SHA-256, the message and its padding are parsed into N 512-bit blocks, M(1), M(2),…..,M(n).Since the 512 bits of the input block may be expressed as sixteen 32 bit words, the first 32 bits of message block i are denoted M0(i),the next 32 bits are M1(i),and so on up to M15(i).

**Setting the Initial Hash Value(H(0)):**

Before hash computation begins for each of the secure hash algorithms, the initial hash value, H(0),must be set. The size and number of words in H(0) depends on the message digest size.

For SHA-256, the initial hash value, H(0),shall consist of the following eight-32 bit words, in hex:

H0(0) = 6a09e667

H1(0) = bb67ae85

H2(0) = 3c6ef372

H3(0) = a54ff53a

H4(0) = 510e527f

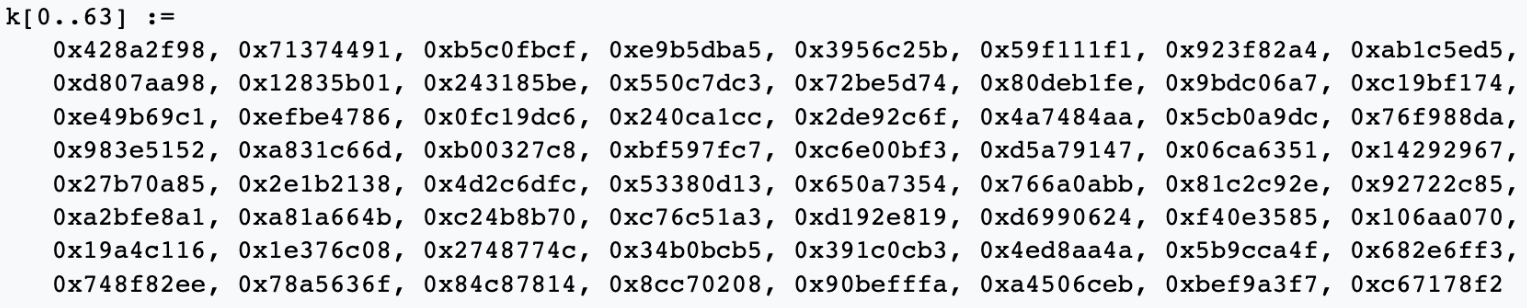
H5(0) = 9b05688c

H6(0) = 1f83d9ab

H7(0) = 5be0cd19

These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

SHA-256 use the sequence of sixty-four constant 32-bit words. These words represent the first 32 bits of the fractional parts of the cube roots of the first sixth-four prime numbers. In hex, these constant words are: -

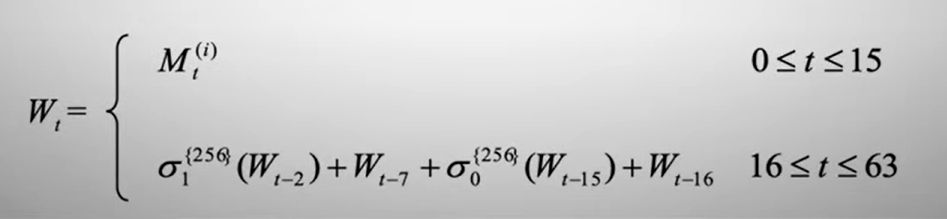


**SHA-256 Hash Computation:-**

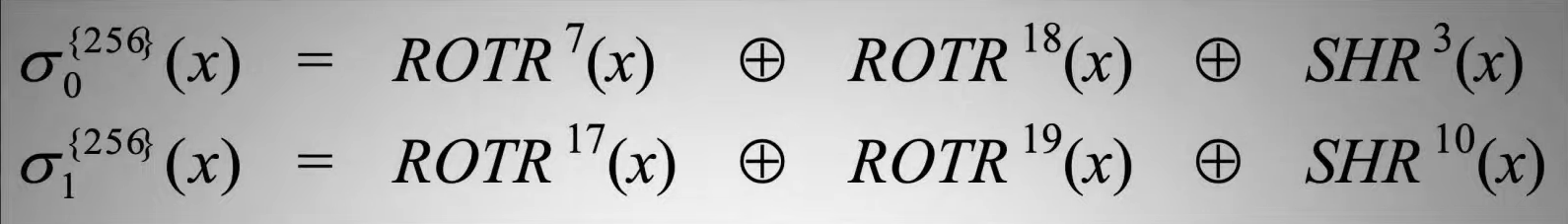
Each message block, M(1),M(2),……M(n),is processed in order, using the following steps:

For i=1 to N:

Prepare the message schedule, {Wt}:



Where sigma0 and sigma1 are calculated by the following equations:-



Where **ROTR** stands for rotation right by some number provided above which means we get each bit and we just move it over by the number of bits which is provided and so on.

And **SHR** stands for shift right operation, we have to shift the data by number of bits provided above the function.

Next, initialize the eight working variables **a, b, c, d, e, f, g and h** with the (i-1) st hash value:

a = H0(i-1)

b = H1(i-1)

c = H2(i-1)

d = H3(i-1)

e = H4(i-1)

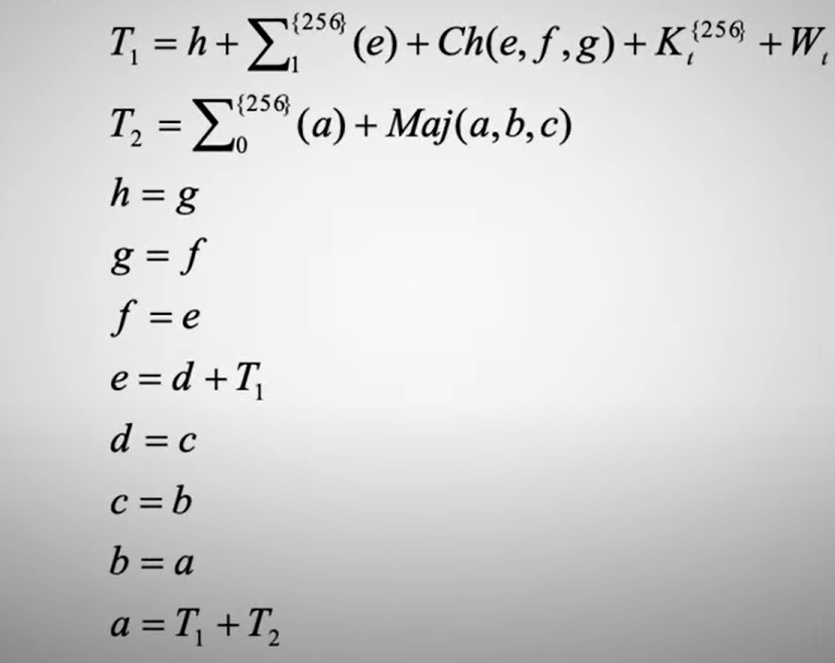
f = H5(i-1)

g = H6(i-1)

h = H7(i-1)

Next, we have to perform a for loop

For t=0 to 63:



Where sigma0 is calculated by ROTR2 + ROTR13+ROTR22

Without SHR operation performed whereas sigma1 is calculated by ROTR6+ROTR11+ROTR25.

(**NOTE:** The additions performed here are Bit wise 232 modulo)

**Ch** function stands for choose function. In Ch(e,f,g) by using the e bits as reference the final bits are generated.

If the bit is 0 in e, then the respective bit is chosen from g and if the bit is 1 in e, then it is chosen from f.

**Maj** function stands for majority function. In Maj(a,b,c) if the majority bits are 0 in the respective position then the final bit is taken as 0 and vice versa.

Next, Compute the ith intermediate hash value H(i):

H0(i) = a+H0(i-1)

H1(i) = b+H1(i-1)

H2(i) = c+H2(i-2)

H3(i) = d+H3(i-3)

H4(i) = e+H4(i-4)

H5(i) = f+H5(i-5)

H6(i) = g+H6(i-6)

H7(i) = h+H7(i-7)

After repeating steps one through four a total of N times,

THE RESULTING 256-BIT MESSAGE DIGEST OF THE MESSAGE M IS,

**H0(N)||H1(N)||H2(N)||H3(N)||H4(N)||H5(N)||H6(N)||H7(N)**

**ADVANCED ENCRYPTION STANDARD (AES) ALGORITHM**

AES (Advanced Encryption Standard) is a widely used symmetric encryption algorithm that operates on blocks of data. It employs a fixed block size of 128 bits and a key size of 128 bits. The AES algorithm operates on a block of data, which is divided into 128-bit blocks. AES has three different key sizes: 128 bits, 192 bits, and 256 bits. AES-128 uses a 128-bit key to encrypt and decrypt data.

The AES-128 encryption algorithm consists of several rounds of operations that transform the input plaintext into the final ciphertext. The number of rounds depends on the key size and the block size. The AES128 algorithm consists of multiple rounds of transformation applied to the input data. These rounds involve key expansion, byte substitution, row shifting, and column mixing operations, which collectively provide a high level of security. During encryption, the input data is divided into blocks, and each block undergoes these transformations. The key expansion process generates a set of round keys derived from the original encryption key.

**Advantages of 128-bit key size:**

1.It is faster compared to larger key sizes because it requires fewer rounds of encryption.

2.It requires less memory to store the key.

3.It is generally considered secure enough for most applications.

**Disadvantages of 128-bit key size:**

The key space is relatively small compared to larger key sizes, which makes it more vulnerable to brute-force attacks.

128 bits key performs 10 rounds of operation

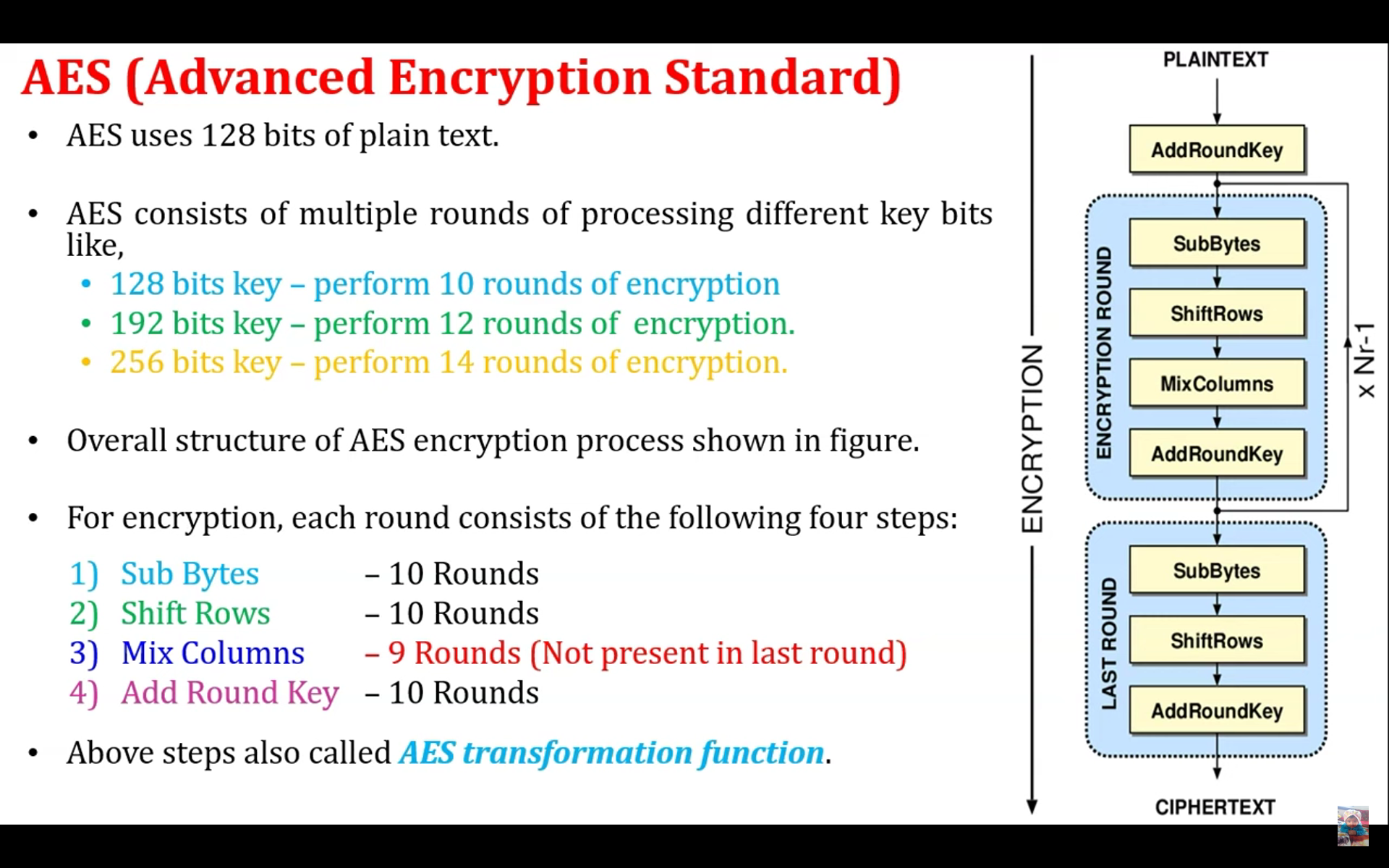
**Each Round consist of four steps**

1.Sub Bytes (Substitute Bytes Transformation Function)

2.Shift Rows

3.Mix Columns

4.Add Round key



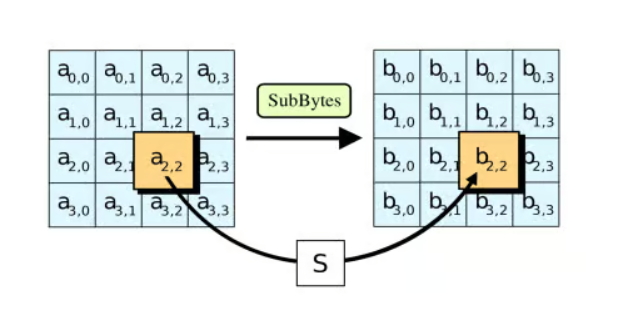
**1.Sub Bytes:**

The SubBytes step is the first operation performed in each round of the AES-128 encryption algorithm. In this step, each byte of the input matrix is replaced with a corresponding byte from a lookup table called the S-box.

The S-box is a fixed table of 256 bytes, each representing a substitution value. The substitution value for each byte is determined using a mathematical function that is designed to be resistant to cryptanalysis. The S-box is created using a combination of a fixed permutation of the input bits and a mathematical transformation called an affine transformation.

The substitution is performed by dividing the input matrix into individual bytes, and replacing each byte with the corresponding substitution value from the S-box. For example, if the input matrix contains the byte value 0x53, the corresponding substitution value from the S-box would be 0xED.

The SubBytes step is an important part of the AES-128 encryption algorithm because it provides confusion by changing the values of the input data. The S-box substitution is a nonlinear operation that helps to prevent patterns in the input data from being preserved in the output ciphertext. This makes it more difficult for an attacker to perform cryptanalysis on the encrypted data.

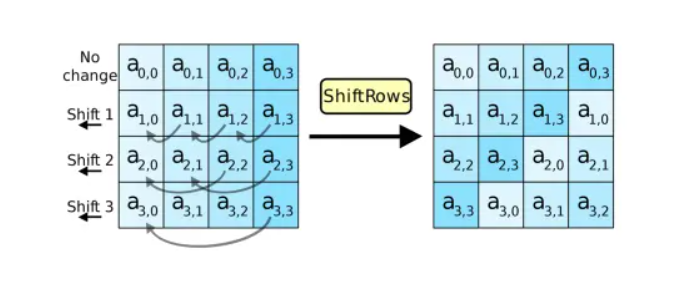


**2.Shift Rows:**

The ShiftRows step is the second operation performed in each round of the AES-128 encryption algorithm. In this step, the rows of the input matrix are shifted to the left.

The ShiftRows step is designed to provide diffusion, which is the process of spreading out the input data so that small changes in the input result in significant changes in the output ciphertext. This helps to make it more difficult for an attacker to find patterns in the data and perform cryptanalysis.

In the ShiftRows step, each row of the input matrix is shifted to the left by a certain number of bytes. The first row is not shifted at all, the second row is shifted one byte to the left, the third row is shifted two bytes to the left, and the fourth row is shifted three bytes to the left.

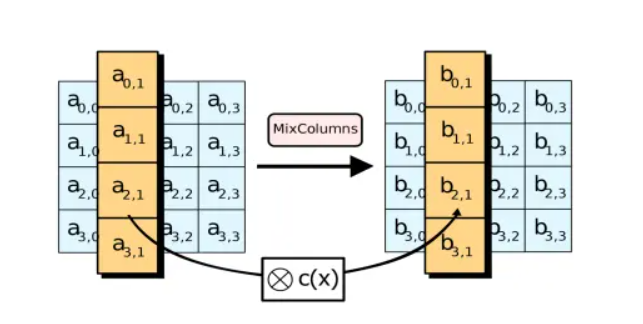


**3.Mix Columns:**

The MixColumns step is the third operation performed in each round of the AES-128 encryption algorithm. In this step, each column of the input matrix is mixed using a fixed matrix multiplication.

The purpose of the Mix Columns step is to provide confusion and diffusion, which are both important properties of a secure encryption algorithm. Confusion refers to the process of making the relationship between the input and output data as complex as possible, while diffusion refers to the process of spreading out the input data so that small changes in the input result in significant changes in the output.

In the MixColumns step, each column of the input matrix is multiplied by a fixed matrix. This matrix multiplication is done in a special way that ensures that each byte in the input matrix affects multiple bytes in the output matrix. The multiplication is done using the Galois field arithmetic, which is a special kind of arithmetic that is used in cryptography.

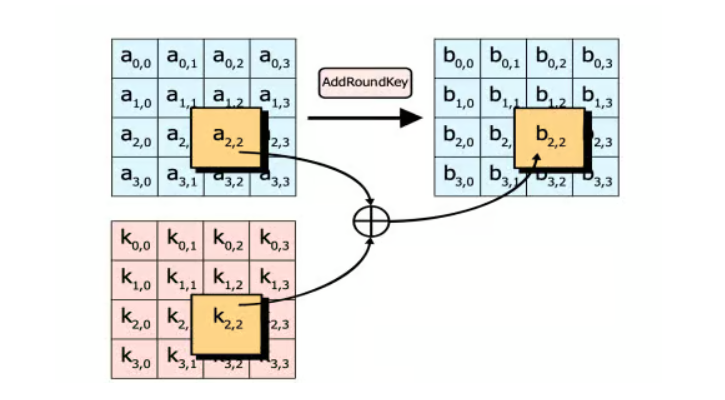
****

**4.Add Round Key:**

The AddRoundKey step is the fourth and final operation performed in each round of the AES-128 encryption algorithm. In this step, each byte of the input matrix is combined with a byte of the round key using an XOR operation.

The purpose of the AddRoundKey step is to add a layer of security to the encryption process. The round key is a key derived from the original encryption key and is unique for each round of the algorithm. By combining each byte of the input matrix with a byte of the round key, the AddRoundKey step helps to ensure that the output ciphertext is different for each round of the algorithm.

In the AddRoundKey step, each byte of the input matrix is XORed with a corresponding byte of the round key.



**CODES (VERILOG HDL):**

**Password verification module**

`timescale 1ns / 1ps

module Password\_Verification (

input clk,rst,

input [7:0] password,

// input last\_word,

output reg success,

output reg access

);

wire [255:0] hash\_data;

wire output\_valid;

wire [127:0] encrypted\_output;

wire [127:0] hashed\_output\_1;

wire [127:0] hashed\_output\_2;

wire [127:0] hashed\_output\_3;

reg count\_1=0;

reg count\_2=0;

reg count\_3=0;

Vote\_Enrollment V1(

.clk(clk),

.rst(rst),

.hashed\_output\_1(hashed\_output\_1),

.hashed\_output\_2(hashed\_output\_2),

.hashed\_output\_3(hashed\_output\_3)

);

// Instantiate the SHA256 module

sha256 sha(

.input\_data({password, 24'b000100110111101000011111}),

.input\_valid(1'b1),

.input\_ready(),

.last\_word(1'b1),

.clk(clk),

.rst(rst),

.output\_valid(output\_valid),

.hash\_data(hash\_data)

);

// Instantiate the AES module

aescipher aes(

.clk(clk),

.datain(hash\_data[255:128]),

//.key(key),

.dataout(encrypted\_output)

);

always@(posedge clk)

begin

if(encrypted\_output == (hashed\_output\_1))

if(count\_1==0)

begin

success=1'b1;

access=1;

count\_1=1;

end

else

begin

access=0;

end

else if( encrypted\_output == hashed\_output\_2 )

if(count\_2==0)

begin

success=1'b1;

access=1;

count\_2=1;

end

else

begin

access=0;

end

else if(encrypted\_output == hashed\_output\_3)

if(count\_3==0)

begin

success=1'b1;

access=1;

count\_3=1;

end

else

begin

access=0;

end

else

success=1'b0;

end

endmodule

**Vote Enrollment**

`timescale 1ns / 1ps

module Vote\_Enrollment(

input clk,

input rst,

output reg [127:0] hashed\_output\_1,

output reg [127:0] hashed\_output\_2,

output reg [127:0] hashed\_output\_3

);

wire [255:0] hash\_data1;

wire output\_valid1;

wire [127:0] encrypted\_output1;

wire [255:0] hash\_data2;

wire output\_valid2;

wire [127:0] encrypted\_output2;

wire [255:0] hash\_data3;

wire output\_valid3;

wire [127:0] encrypted\_output3;

sha256 s1(

.input\_data({8'b01110010 ,24'b000100110111101000011111}),//password(r)

.input\_valid(1'b1),

.input\_ready(),

.last\_word(1'b1),

.clk(clk),

.rst(rst),

.output\_valid(output\_valid1),

.hash\_data(hash\_data1)

);

// Instantiate the AES module

aescipher a1(

.clk(clk),

.datain(hash\_data1[255:128]),

//.key(key),

.dataout(encrypted\_output1)

);

// 2nd password

sha256 s2(

.input\_data({8'b11010101,24'b000100110111101000011111}),// Õ([password2)

.input\_valid(1'b1),

.input\_ready(),

.last\_word(1'b1),

.clk(clk),

.rst(rst),

.output\_valid(output\_valid2),

.hash\_data(hash\_data2)

);

// Instantiate the AES module

aescipher a2(

.clk(clk),

.datain(hash\_data2[255:128]),

//.key(key),

.dataout(encrypted\_output2)

);

sha256 s3(

.input\_data({8'b00101111, 24'b000100110111101000011111}),//(-password3)

.input\_valid(1'b1),

.input\_ready(),

.last\_word(1'b1),

.clk(clk),

.rst(rst),

.output\_valid(output\_valid3),

.hash\_data(hash\_data3)

);

// Instantiate the AES module

aescipher a3(

.clk(clk),

.datain(hash\_data3[255:128]),

//.key(key),

.dataout(encrypted\_output3)

);

always@(posedge clk) begin

hashed\_output\_1 = encrypted\_output1;

hashed\_output\_2 = encrypted\_output2;

hashed\_output\_3 = encrypted\_output3;

end

endmodule

**SHA-modules**

Module

sha256(input\_data, input\_valid, input\_ready, last\_word, clk, rst, output\_valid, hash\_data);

input [31:0]input\_data;

input input\_valid;

input last\_word;

input clk,rst;

output reg input\_ready;

output wire [255:0]hash\_data;

output wire output\_valid;

wire [6:0]block\_counter;

reg [4:0] length\_counter;

reg [63:0]size;

reg [1:0]special;

reg [31:0]output\_data;

reg D;

reg last\_word\_delayed;

reg [1:0]last\_next;

always @(negedge rst)

begin

last\_next = 2'b00;

input\_ready <= 1'b1;

length\_counter <= 5'd31;

special <= 2'd0;

size <= 64'hffffffffffffffe0;

D <= 1'b1;

last\_word\_delayed <= 1'b0;

end

sha\_engine Eng(output\_data,clk,block\_counter,rst,output\_valid,hash\_data,last\_word,last\_next);

always @(posedge clk)

begin

if(input\_valid == 1'b1 && input\_ready == 1'b1)

begin

if(length\_counter == 5'd14)

input\_ready <= 1'b0;

length\_counter = length\_counter + 1;

end

end

always @(posedge clk)

begin

if(last\_word == 1'b1)

last\_word\_delayed <= 1'b1;

end

always @(posedge last\_word\_delayed)

begin

if(length\_counter<=13)

special <= 2'b01;

else

special <= 2'b11;

end

always @(posedge clk)

begin

case(special)

2'b00 : begin

output\_data <= input\_data;

last\_next <= 2'b00;

end

2'b01 : begin

last\_next <= 2'b01;

if(length\_counter<15)

begin

if(length\_counter == 1 && size == 512)

output\_data <= {1'b1,{31{1'b0}}};

else

begin

D <= 1'b0;

output\_data <= {D,{31{1'b0}}};

end

end

else if(length\_counter == 15)

begin

output\_data <= size[63:32];

length\_counter <= length\_counter + 1;

end

else

output\_data <= size[31:0];

end

2'b11 : begin

last\_next <= 2'b10;

D <= 1'b0;

output\_data <= {D,{31{1'b0}}};

end

endcase

end

always @(posedge clk)

begin

if(block\_counter == 64)

begin

length\_counter <= 5'd31;

input\_ready <= 1'b1;

end

if(block\_counter == 65)

begin

if(special == 3)

special <= 2'b01;

else

special <= 2'd0;

end

end

always @(posedge clk)

begin

if(input\_ready == 1 && special == 0)

size <= size + 32;

end

always @(posedge last\_word)

begin

if(length\_counter == 15)

size <= size + 32;

end

endmodule

module sha\_engine(word,clk,index,rst,output\_valid,hash\_data,last\_word,last\_next);

input [31:0]word;

input clk,rst,last\_word;

input [1:0]last\_next;

reg [31:0]W[63:0];

output reg [6:0] index;

output [255:0]hash\_data;

output reg output\_valid;

reg [31:0]p,q,r,x,a,b,c,d,e,f,g,h,sum0,Ch,sum1,Maj,T1,T2;

reg [31:0]H[7:0];

reg [31:0]K[63:0];

reg D;

always @(negedge rst)

begin

D = 1'b0;

index = 7'd126;

output\_valid = 1'b0;

H[0] = 32'h6a09e667;

H[1] = 32'hbb67ae85;

H[2] = 32'h3c6ef372;

H[3] = 32'ha54ff53a;

H[4] = 32'h510e527f;

H[5] = 32'h9b05688c;

H[6] = 32'h1f83d9ab;

H[7] = 32'h5be0cd19;

K[0] = 32'h428a2f98;

K[1] = 32'h71374491;

K[2] = 32'hb5c0fbcf;

K[3] = 32'he9b5dba5;

K[4] = 32'h3956c25b;

K[5] = 32'h59f111f1;

K[6] = 32'h923f82a4;

K[7] = 32'hab1c5ed5;

K[8] = 32'hd807aa98;

K[9] = 32'h12835b01;

K[10] = 32'h243185be;

K[11] = 32'h550c7dc3;

K[12] = 32'h72be5d74;

K[13] = 32'h80deb1fe;

K[14] = 32'h9bdc06a7;

K[15] = 32'hc19bf174;

K[16] = 32'he49b69c1;

K[17] = 32'hefbe4786;

K[18] = 32'h0fc19dc6;

K[19] = 32'h240ca1cc;

K[20] = 32'h2de92c6f;

K[21] = 32'h4a7484aa;

K[22] = 32'h5cb0a9dc;

K[23] = 32'h76f988da;

K[24] = 32'h983e5152;

K[25] = 32'ha831c66d;

K[26] = 32'hb00327c8;

K[27] = 32'hbf597fc7;

K[28] = 32'hc6e00bf3;

K[29] = 32'hd5a79147;

K[30] = 32'h06ca6351;

K[31] = 32'h14292967;

K[32] = 32'h27b70a85;

K[33] = 32'h2e1b2138;

K[34] = 32'h4d2c6dfc;

K[35] = 32'h53380d13;

K[36] = 32'h650a7354;

K[37] = 32'h766a0abb;

K[38] = 32'h81c2c92e;

K[39] = 32'h92722c85;

K[40] = 32'ha2bfe8a1;

K[41] = 32'ha81a664b;

K[42] = 32'hc24b8b70;

K[43] = 32'hc76c51a3;

K[44] = 32'hd192e819;

K[45] = 32'hd6990624;

K[46] = 32'hf40e3585;

K[47] = 32'h106aa070;

K[48] = 32'h19a4c116;

K[49] = 32'h1e376c08;

K[50] = 32'h2748774c;

K[51] = 32'h34b0bcb5;

K[52] = 32'h391c0cb3;

K[53] = 32'h4ed8aa4a;

K[54] = 32'h5b9cca4f;

K[55] = 32'h682e6ff3;

K[56] = 32'h748f82ee;

K[57] = 32'h78a5636f;

K[58] = 32'h84c87814;

K[59] = 32'h8cc70208;

K[60] = 32'h90befffa;

K[61] = 32'ha4506ceb;

K[62] = 32'hbef9a3f7;

K[63] = 32'hc67178f2;

end

always @(posedge clk)

begin

if(index == 127)

begin

a <= H[0];

b <= H[1];

c <= H[2];

d <= H[3];

e <= H[4];

f <= H[5];

g <= H[6];

h <= H[7];

end

if(index<16)

W[index] <= word;

else if(index<64)

begin

p = {W[index-2][16:0],W[index-2][31:17]}^{W[index-2][18:0],W[index-2][31:19]}^(W[index-2]>>10);

q = W[index-7];

r = {W[index-15][6:0],W[index-15][31:7]}^{W[index-15][17:0],W[index-15][31:18]}^(W[index-15]>>3);

x = W[index-16];

W[index] <= p + q + r +x;

end

index <= index + 1;

end

always @(posedge clk)

begin

if(index>0 && index <65)

begin

sum1 = {e[5:0],e[31:6]}^{e[10:0],e[31:11]}^{e[24:0],e[31:25]};

sum0 = {a[1:0],a[31:2]}^{a[12:0],a[31:13]}^{a[21:0],a[31:22]};

Ch = (e&f)^((~e)&g);

Maj = (a&b)^(b&c)^(c&a);

T1 = h + sum1 + Ch +K[index-1] + W[index-1];

T2 = sum0 + Maj;

h <= g;

g <= f;

f <= e;

e <= d + h + sum1 + Ch + K[index-1]+ W[index-1];

d <= c;

c <= b;

b <= a;

a <= T1 + T2;

end

if(index == 65)

begin

H[0] <= a + H[0];

H[1] <= b + H[1];

H[2] <= c + H[2];

H[3] <= d + H[3];

H[4] <= e + H[4];

H[5] <= f + H[5];

H[6] <= g + H[6];

H[7] <= h + H[7];

index <= 127;

if(last\_word == 1 && (last\_next[0] == 1'b1 || D == 1'b1))

begin

output\_valid = 1'b1;

end

if(last\_next[1] == 1'b1)

begin

D <= 1'b1;

end

end

end

assign hash\_data = {H[0],H[1],H[2],H[3],H[4],H[5],H[6],H[7]};

endmodule

**AES\_based modules**

`timescale 1ns / 1ps

module aescipher(clk,datain,dataout);

input clk;

input [127:0] datain;

//input [127:0] key;

output[127:0] dataout;

wire [127:0] r0\_out;

wire [127:0] r1\_out,r2\_out,r3\_out,r4\_out,r5\_out,r6\_out,r7\_out,r8\_out,r9\_out;

wire [127:0] keyout1,keyout2,keyout3,keyout4,keyout5,keyout6,keyout7,keyout8,keyout9;

assign r0\_out = datain^(128'h00112233445566778899AABBCCDDEEFF);

rounds r1(.clk(clk),.rc(4'b0000),.data(r0\_out),.keyin(128'h00112233445566778899AABBCCDDEEFF),.keyout(keyout1),.rndout(r1\_out));

rounds r2(.clk(clk),.rc(4'b0001),.data(r1\_out),.keyin(keyout1),.keyout(keyout2),.rndout(r2\_out));

rounds r3(.clk(clk),.rc(4'b0010),.data(r2\_out),.keyin(keyout2),.keyout(keyout3),.rndout(r3\_out));

rounds r4(.clk(clk),.rc(4'b0011),.data(r3\_out),.keyin(keyout3),.keyout(keyout4),.rndout(r4\_out));

rounds r5(.clk(clk),.rc(4'b0100),.data(r4\_out),.keyin(keyout4),.keyout(keyout5),.rndout(r5\_out));

rounds r6(.clk(clk),.rc(4'b0101),.data(r5\_out),.keyin(keyout5),.keyout(keyout6),.rndout(r6\_out));

rounds r7(.clk(clk),.rc(4'b0110),.data(r6\_out),.keyin(keyout6),.keyout(keyout7),.rndout(r7\_out));

rounds r8(.clk(clk),.rc(4'b0111),.data(r7\_out),.keyin(keyout7),.keyout(keyout8),.rndout(r8\_out));

rounds r9(.clk(clk),.rc(4'b1000),.data(r8\_out),.keyin(keyout8),.keyout(keyout9),.rndout(r9\_out));

rounndlast r10(.clk(clk),.rc(4'b1001),.rin(r9\_out),.keylastin(keyout9),.fout(dataout));

endmodule

**rounds**

`timescale 1ns / 1ps

module rounds(clk,rc,data,keyin,keyout,rndout);

input clk;

input [3:0]rc;

input [127:0]data;

input [127:0]keyin;

output [127:0]keyout;

output [127:0]rndout;

wire [127:0] sb,sr,mcl;

KeyGeneration t0(rc,keyin,keyout);

subbytes t1(data,sb);

shiftrow t2(sb,sr);

mixcolumn t3(sr,mcl);

assign rndout= keyout^mcl;

endmodule

**KeyGeneration**

`timescale 1ns / 1ps

module KeyGeneration(rc,key,keyout);

input [3:0] rc;

input [127:0]key;

output [127:0] keyout;

wire [31:0] w0,w1,w2,w3,tem;

assign w0 = key[127:96];

assign w1 = key[95:64];

assign w2 = key[63:32];

assign w3 = key[31:0];

assign keyout[127:96]= w0 ^ tem ^ rcon(rc);

assign keyout[95:64] = w0 ^ tem ^ rcon(rc)^ w1;

assign keyout[63:32] = w0 ^ tem ^ rcon(rc)^ w1 ^ w2;

assign keyout[31:0] = w0 ^ tem ^ rcon(rc)^ w1 ^ w2 ^ w3;

sbox a1(.a(w3[23:16]),.c(tem[31:24]));

sbox a2(.a(w3[15:8]),.c(tem[23:16]));

sbox a3(.a(w3[7:0]),.c(tem[15:8]));

sbox a4(.a(w3[31:24]),.c(tem[7:0]));

function [31:0] rcon;

input [3:0] rc;

case(rc)

4'h0: rcon=32'h01\_00\_00\_00;

4'h1: rcon=32'h02\_00\_00\_00;

4'h2: rcon=32'h04\_00\_00\_00;

4'h3: rcon=32'h08\_00\_00\_00;

4'h4: rcon=32'h10\_00\_00\_00;

4'h5: rcon=32'h20\_00\_00\_00;

4'h6: rcon=32'h40\_00\_00\_00;

4'h7: rcon=32'h80\_00\_00\_00;

4'h8: rcon=32'h1b\_00\_00\_00;

4'h9: rcon=32'h36\_00\_00\_00;

default: rcon=32'h00\_00\_00\_00;

endcase

endfunction

endmodule

**Sbox**

`timescale 1ns / 1ps

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

**Subbytes**

`timescale 1ns / 1ps

module subbytes(data,sb);

input [127:0] data;

output [127:0] sb;

sbox q0( .a(data[127:120]),.c(sb[127:120]) );

sbox q1( .a(data[119:112]),.c(sb[119:112]) );

sbox q2( .a(data[111:104]),.c(sb[111:104]) );

sbox q3( .a(data[103:96]),.c(sb[103:96]) );

sbox q4( .a(data[95:88]),.c(sb[95:88]) );

sbox q5( .a(data[87:80]),.c(sb[87:80]) );

sbox q6( .a(data[79:72]),.c(sb[79:72]) );

sbox q7( .a(data[71:64]),.c(sb[71:64]) );

sbox q8( .a(data[63:56]),.c(sb[63:56]) );

sbox q9( .a(data[55:48]),.c(sb[55:48]) );

sbox q10(.a(data[47:40]),.c(sb[47:40]) );

sbox q11(.a(data[39:32]),.c(sb[39:32]) );

sbox q12(.a(data[31:24]),.c(sb[31:24]) );

sbox q13(.a(data[23:16]),.c(sb[23:16]) );

sbox q14(.a(data[15:8]),.c(sb[15:8]) );

sbox q16(.a(data[7:0]),.c(sb[7:0]) );

endmodule

**Shift\_row**

`timescale 1ns / 1ps

module shiftrow(sb,sr);

input [127:0] sb;

output [127:0] sr;

assign sr[127:120] = sb[127:120];

assign sr[119:112] = sb[87:80];

assign sr[111:104] = sb[47:40];

assign sr[103:96] = sb[7:0];

assign sr[95:88] = sb[95:88];

assign sr[87:80] = sb[55:48];

assign sr[79:72] = sb[15:8];

assign sr[71:64] = sb[103:96];

assign sr[63:56] = sb[63:56];

assign sr[55:48] = sb[23:16];

assign sr[47:40] = sb[111:104];

assign sr[39:32] = sb[71:64];

assign sr[31:24] = sb[31:24];

assign sr[23:16] = sb[119:112];

assign sr[15:8] = sb[79:72];

assign sr[7:0] = sb[39:32];

endmodule

**Mix column**

`timescale 1ns / 1ps

module mixcolumn(a,mcl);

input [127:0] a;

output [127:0] mcl;

assign mcl[127:120]= mixcolumn32 (a[127:120],a[119:112],a[111:104],a[103:96]);

assign mcl[119:112]= mixcolumn32 (a[119:112],a[111:104],a[103:96],a[127:120]);

assign mcl[111:104]= mixcolumn32 (a[111:104],a[103:96],a[127:120],a[119:112]);

assign mcl[103:96]= mixcolumn32 (a[103:96],a[127:120],a[119:112],a[111:104]);

assign mcl[95:88]= mixcolumn32 (a[95:88],a[87:80],a[79:72],a[71:64]);

assign mcl[87:80]= mixcolumn32 (a[87:80],a[79:72],a[71:64],a[95:88]);

assign mcl[79:72]= mixcolumn32 (a[79:72],a[71:64],a[95:88],a[87:80]);

assign mcl[71:64]= mixcolumn32 (a[71:64],a[95:88],a[87:80],a[79:72]);

assign mcl[63:56]= mixcolumn32 (a[63:56],a[55:48],a[47:40],a[39:32]);

assign mcl[55:48]= mixcolumn32 (a[55:48],a[47:40],a[39:32],a[63:56]);

assign mcl[47:40]= mixcolumn32 (a[47:40],a[39:32],a[63:56],a[55:48]);

assign mcl[39:32]= mixcolumn32 (a[39:32],a[63:56],a[55:48],a[47:40]);

assign mcl[31:24]= mixcolumn32 (a[31:24],a[23:16],a[15:8],a[7:0]);

assign mcl[23:16]= mixcolumn32 (a[23:16],a[15:8],a[7:0],a[31:24]);

assign mcl[15:8]= mixcolumn32 (a[15:8],a[7:0],a[31:24],a[23:16]);

assign mcl[7:0]= mixcolumn32 (a[7:0],a[31:24],a[23:16],a[15:8]);

function [7:0] mixcolumn32;

input [7:0] i1,i2,i3,i4;

begin

mixcolumn32[7]=i1[6] ^ i2[6] ^ i2[7] ^ i3[7] ^ i4[7];

mixcolumn32[6]=i1[5] ^ i2[5] ^ i2[6] ^ i3[6] ^ i4[6];

mixcolumn32[5]=i1[4] ^ i2[4] ^ i2[5] ^ i3[5] ^ i4[5];

mixcolumn32[4]=i1[3] ^ i1[7] ^ i2[3] ^ i2[4] ^ i2[7] ^ i3[4] ^ i4[4];

mixcolumn32[3]=i1[2] ^ i1[7] ^ i2[2] ^ i2[3] ^ i2[7] ^ i3[3] ^ i4[3];

mixcolumn32[2]=i1[1] ^ i2[1] ^ i2[2] ^ i3[2] ^ i4[2];

mixcolumn32[1]=i1[0] ^ i1[7] ^ i2[0] ^ i2[1] ^ i2[7] ^ i3[1] ^ i4[1];

mixcolumn32[0]=i1[7] ^ i2[7] ^ i2[0] ^ i3[0] ^ i4[0];

end

endfunction

endmodule

Remaining 8 rounds have same modules and the **Last round module** is

No mix column module

**Test Bench**

`timescale 1ns / 1ps

module tb\_Password\_Verification;

// Inputs

reg [7:0] password;

//reg [7:0] enterpassword;

reg clk;

reg rst;

wire success;

Password\_Verification dut (

.password(password),

//.enterpassword(enterpassword),

//.key(key),

.clk(clk),

.rst(rst),

.success(success),

.access(access)

);

initial begin

clk = 0;

forever #1 clk = ~clk;

end

initial begin

rst = 1; //not edited

#50;

rst = 0; //edited

#100;

// password = 8'b11010101;

password = 8'b11111111;

#100;

rst = 1; //not edited

#50;

rst = 0; //edited

#100;

//

password = 8'b11011011;

#100;

rst = 1; //not edited

#50;

rst = 0; //edited

#100;

//

//password=8'b01110010;

password=8'b00000000;

#100;

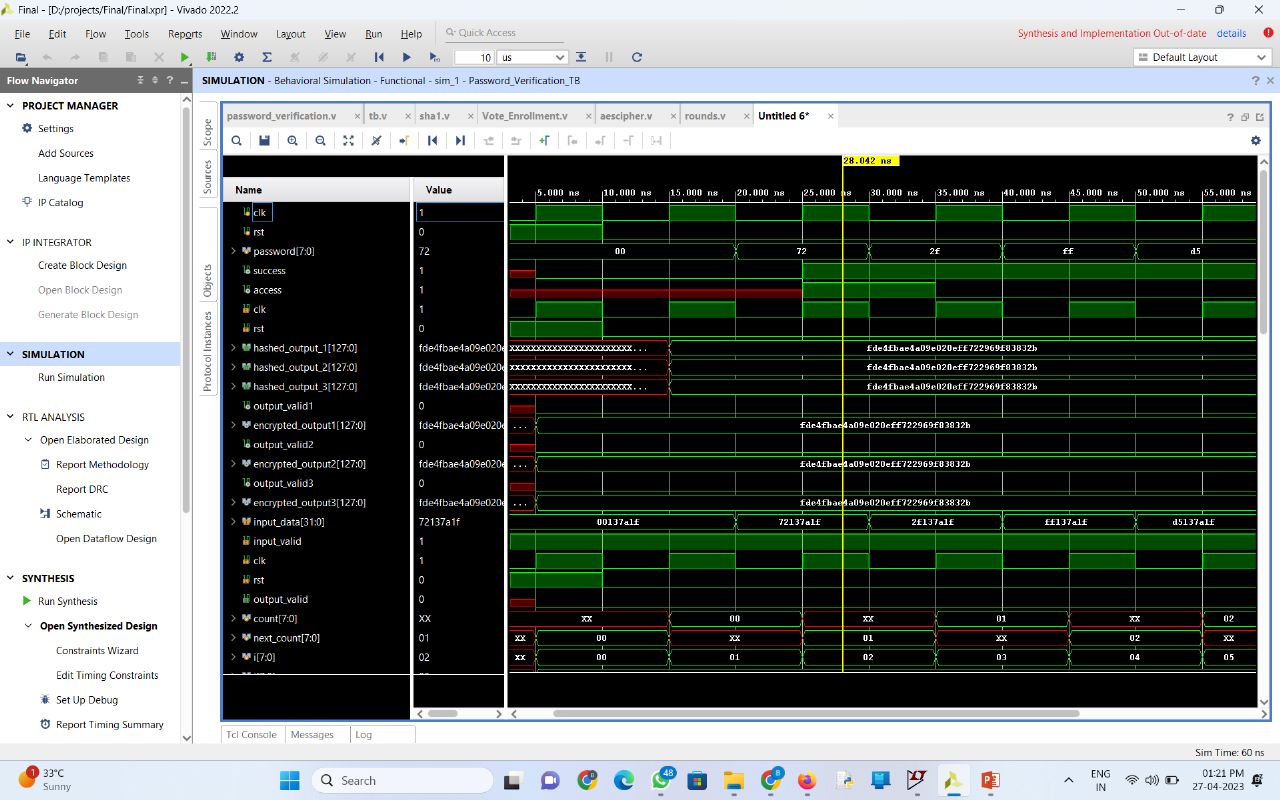
$finish;

End

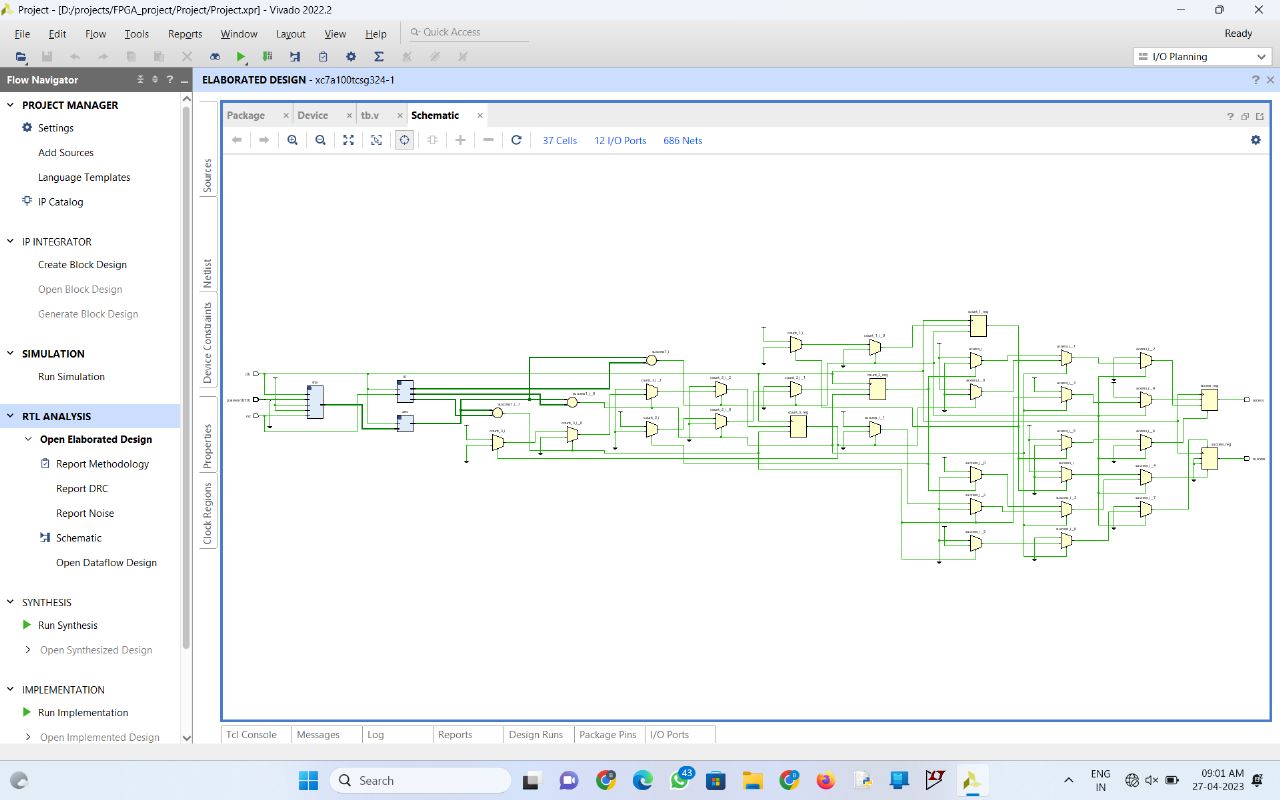
Endmodule

**SIMULATIONS:**

**Waveform:**

****

**RTL SCHEMATIC**:



**Nexys 4DDR Artix-7 (XC7A100TCSG324-1) FPGA Implementation-**



**CONCLUSION**

In this project, we implemented a secure password-based voter enrollment and verification system using cryptographic techniques. The system ensures the protection of voter’s passwords and prevents unauthorized access to the voting process.

**FURTHER WORKS**

We can use Finger print sensor instead of password for Voter verification.

**References**

* [**https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm**](https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm)
* [**https://www.ijert.org/research/verilog-implementation-of-a-secure-system-for-smart-card-transactions-with-advanced-encryption-standard-and-secure-hash-algorithm-IJERTV4IS050326.pdf**](https://www.ijert.org/research/verilog-implementation-of-a-secure-system-for-smart-card-transactions-with-advanced-encryption-standard-and-secure-hash-algorithm-IJERTV4IS050326.pdf)