Introduction :

# **3.1** Introduction

The RSA algorithm is a widely used public-key encryption technique that has been employed to secure online transactions, communication networks, and digital signatures for over four decades. The algorithm's security relies heavily on the generation of secure keys, which are used for encryption and decryption. Key generation is a critical component of the RSA algorithm, as it directly affects the security of the data being transmitted.

In this project, we aim to implement the key generation function in Verilog for hardware FPGA implementation, demonstrating the feasibility of using FPGAs for RSA key generation. This approach offers several advantages, including high-speed and low-power consumption, making it suitable for a wide range of applications, including secure communication networks and digital signatures.

The RSA algorithm was first introduced in 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman, and it has since become a widely accepted standard for secure data transmission. The algorithm's security is based on the difficulty of factoring large composite numbers, which are used to generate the public and private keys.

The key generation process involves selecting two large prime numbers, p and q, and computing the modulus, n, and the Euler's totient function, φ(n). The public exponent, e, is then chosen, and the private exponent, d, is computed using the extended Euclidean algorithm. Finally, the public and private keys are generated using the computed values.

The security of the RSA algorithm relies on the difficulty of factoring the modulus, n, which is a product of the two prime numbers, p and q. If an attacker can factor n, they can compute the private key, d, and decrypt the encrypted data. Therefore, it is essential to choose large prime numbers, p and q, to ensure the security of the algorithm.

In addition to the security benefits, the RSA algorithm also offers several other advantages, including:

- Key exchange: The RSA algorithm allows for secure key exchange between two parties over an insecure communication channel.

- Digital signatures: The RSA algorithm can be used to create digital signatures, which are used to authenticate the sender of a message and ensure the integrity of the data.

- Authentication: The RSA algorithm can be used for authentication purposes, such as verifying the identity of a user or device.

Yahaflowchart lga do or thodijgahachhod do us diagram ko explain krne le liye

# Type of keys :

Key generation is a critical component of cryptography, as it enables secure communication over an insecure channel.

Types of Keys:

There are two types of keys used in key generation:

1. Public Key (PK): The public key is used for encryption and is made publicly available. It consists of the modulus, n, and the public exponent, e.

2. Private Key (SK): The private key is used for decryption and is kept secret. It consists of the modulus, n, and the private exponent, d.

How Keys Help to Encrypt and Decrypt Messages:

The public and private keys play a crucial role in encrypting and decrypting messages. Here's how it works:

Encryption:

1. Alice wants to send a message, M; to Bob.

2. Alice uses Bob's public key, PK, to encrypt the message, M.

3. The encrypted message, C, is computed as C = M^e (mod n).

4. Alice sends the encrypted message, C, to Bob.

Decryption:

1. Bob receives the encrypted message, C.

2. Bob uses his private key, SK, to decrypt the message, C.

3. The decrypted message, M, is computed as M = C^d (mod n).

4. Bob recovers the original message, M.

How Keys Help to Decrease the Message:

The RSA algorithm uses large key sizes to ensure security, which can result in increased computational overhead and memory requirements. However, the keys can be used to decrease the message size, making it more efficient for transmission. Here's how:

1. Message compression: The message, M, can be compressed using a compression algorithm, such as gzip or zlib.

2. Encryption: The compressed message, M', is encrypted using the public key, PK.

3. Transmission: The encrypted message, C, is transmitted over the insecure channel.

4. Decryption: The encrypted message, C, is decrypted using the private key, SK.

5. Decompression: The decrypted message, M', is decompressed using the compression algorithm.

By using the keys to encrypt and decrypt messages, the RSA algorithm provides a secure and efficient way to transmit data over an insecure channel. The keys help to decrease the message size, making it more efficient for transmission, while ensuring the security and integrity of the data.

# steps to generate a key:

The steps to generate a key pair for the RSA algorithm are:

Step 1: Choose two large prime numbers, p and q

- These prime numbers should be secret and randomly chosen

- They should be of similar size, e.g. both 1024-bit or both 2048-bit

Step 2: Compute the modulus, n = p \* q

- This is the public modulus, which is used to encrypt and decrypt data

Step 3: Compute the Euler's totient function, φ(n) = (p - 1) \* (q - 1)

- This is a value that is used to compute the private exponent, d

Step 4: Choose the public exponent, e

- This should be a small prime number, e.g. 3 or 65537

- It should be coprime with φ(n), meaning that gcd(e, φ(n)) = 1

Step 5: Compute the private exponent, d

- This is computed using the extended Euclidean algorithm

- d is the modular multiplicative inverse of e modulo φ(n), i.e. d \* e ≡ 1 (mod φ(n))

Step 6: Generate the public key, PK = (e, n)

- This is the public key that is used to encrypt data

Step 7: Generate the private key, SK = (d, n)

- This is the private key that is used to decrypt data

Note: These steps are for generating a basic RSA key pair. In practice, additional steps may be taken to ensure security, such as:

- Generating a random seed for the prime numbers

- Using a secure random number generator

- Testing the prime numbers for primality

- Using a secure key storage and management system

It's also important to note that the key size should be sufficient for security, e.g. at least 2048-bit.

However, the RSA algorithm also has some limitations, including:

Key size: The RSA algorithm requires large key sizes to ensure security, which can result in increased computational overhead and memory requirements.

Computation time: The RSA algorithm can be computationally intensive, particularly for large key sizes, which can result in increased computation time.

Code:

MAIN.v

`timescale 1ns / 1ps

Module Main(

// input is what you want to encrypt or decrypt

Input [15:0] Input,

Input [7:0]firstPrimeNumber,

Input [7:0]secondPrimeNumber,

Input clk,

Input start,

Input start1,

Input start2,

Output [7:0]encryptionKey,

Output [15:0]n,

Output [15:0]Output,

Output [15:0] decryptionKey,

Output finish,

Output fin1

);

Wire [15:0] MPower;

Parameter InstructionSelector = 0;

encryptionKeyGenerator k1 (firstPrimeNumber,secondPrimeNumber,start,clk,encryptionKey,finish);

decryptionKeyGenerator kd1 (firstPrimeNumber,secondPrimeNumber,encryptionKey,clk,start1,n,decryptionKey,fin1);

generate

if(InstructionSelector)

begin

modularMultiplicatorencryptor (Input,{8’b00000000,encryptionKey},n,start2,clk,finished,Mpower,Output);

assign MPowerOutput = Mpower;

end

else

begin

modularMultiplicatordecryptor (Input,decryptionKey,n,start2,clk,finished,Mpower,Output);

assign MPowerOutput = Mpower;

end

endgenerate

endmodule

Encryption ki generation ( public key )

`timescale 1ns / 1ps

Module encryptionKeyGenerator(

Input [7:0]p,

Input [7:0]q,

Input start,

Input clk,

Output [7:0]e1,

Output reg finish

);

Reg [15:0]e;

Assign e1=e[7:0];

Reg fin;

Wire [15:0]phin;

Assign phin=(p-1)\*(q-1); //calculating phi(n)

Reg [15:0]x,y,random,gcd;

Wire [15:0]r,x1,y1;

Division16 d2(x1,y1,outResult,r);

Assign y1=y,x1=x;

Always @(posedge clk)

Begin

If(start)

Begin

X<=phin;

Random<=3; //start checking gcd from random number=3

Y<=3;

Gcd<=0;

Fin<=0;

Finish<=0;

E<=0;

End

If((fin==1) & (gcd==1)) //output when gcd is 1

Begin

E<=random;

Finish<=1;

End

If (r==0) //gcd is found when remainder is 0(euclidean)

Begin

Gcd<=y;

Fin<=1;

End

If( fin==0) //finding gcd

Begin

X<=y;

Y<=r;

End

If ((fin==1) & (gcd!=1)) // check for another random number if gcd is not 1

Begin

Random<=random+2;

Y<=random+2;

X<=phin;

Gcd<=0;

Fin<=0;

End

End

Endmodule

Decryption ki generation (private key)

`timescale 1ns / 1ps

Module decryptionKeyGenerator(input [7:0] p,

Input [7:0] q,

Input [7:0] e1,

Input clk,

Input start,

Output reg [15:0] n,

Output [15:0] d,

Output finished

);

Reg [47:0] A,B,C;

Reg [15:0] G;

Reg [15:0]e;

Wire [15:0]outResult,Q;

Division16 d2(A[15:0],B[15:0],outResult,remainder);

Assign Q=outResult;

always@(posedgeclk)

begin

if(start)

begin

e={8’b00000000,e1};

n=p\*q;

G=(p-1)\*(q-1);

A={16’h001,16’h000,G};

B={16’h000,16’h001,e};

End

Else if(B[15:0]!=1)

Begin

C[47:32]=A[47:32]-Q\*B[47:32];

C[31:16]=A[31:16]-Q\*B[31:16];

C[15:0]=A[15:0]-Q\*B[15:0];

A=B;

B=C;

End

End

Assign d=B[31:16];

Assign finished=B[15:0]==1;

Endmodule

SUB-MODULES

1. Divider 16bit.v

`timescale 1ns / 1ps

Module Division16(A,B,Res,remainder);

Parameter WIDTH = 16;

//input and output ports.

Input [WIDTH-1:0] A;

Input [WIDTH-1:0] B;

Output [WIDTH-1:0] Res;

Output reg [WIDTH-1:0] remainder;

//internal variables

Reg [WIDTH-1:0] Res = 0;

Reg [WIDTH-1:0] a1,b1;

Reg [WIDTH:0] p1;

Reg [7:0]remainderL;

Integer I;

always@ (A or B)

begin

a1 = A;

b1 = B;

p1= 0;

for(i=0;I <WIDTH;i=i+1) begin

p1 = {p1[WIDTH-2:0],a1[WIDTH-1]};

a1[WIDTH-1:1] = a1[WIDTH-2:0];

p1 = p1-b1;

if(p1[WIDTH-1] == 1) begin

a1[0] = 0;

p1 = p1 + b1; end

else

a1[0] = 1;

end

Res = a1;

{remainderL,remainder} = p1;

End

Endmodule

1. Divider 32bit.v

`timescale 1ns / 1ps

Module Division32(A,B,Res,remainder);

Parameter WIDTH = 32;

Input [WIDTH-1:0] A;

Input [WIDTH-1:0] B;

Output [WIDTH-1:0] Res;

Output reg [WIDTH-1:0] remainder;

Reg [WIDTH-1:0] Res = 0;

Reg [WIDTH-1:0] a1,b1;

Reg [WIDTH:0] p1;

Reg [7:0]remainderL;

Integer I;

always@ (A or B)

begin

a1 = A;

b1 = B;

p1= 0;

for(i=0;I <WIDTH;i=i+1) begin

p1 = {p1[WIDTH-2:0],a1[WIDTH-1]};

a1[WIDTH-1:1] = a1[WIDTH-2:0];

p1 = p1-b1;

if(p1[WIDTH-1] == 1) begin

a1[0] = 0;

p1 = p1 + b1; end

else

a1[0] = 1;

end

Res = a1;

{remainderL,remainder} = p1;

End

Endmodule

1. Multiplayer.v

`timescale 1ns / 1ps

Module modularMultiplicator(input [15:0]M,

Input [15:0]e,

Input [15:0]n,

Input start,

Input clk,

Output finished,

Output reg[31:0]Mpower,

Output [15:0] remainder

);

Reg [15:0] ncount;

Reg [31:0]x,n1;

Division32 d1(x,n1,outResult,remainder);

Always @(posedge clk)

Begin

// $display(“start: %d”, start);

If(start) begin

Ncount = e-1;

// $display(“in start: ncount: %d; x: %d”,ncount, x);

Mpower = M;

X=0;

N1={16’b0000000000000000,n};

End

Else if(!finished)

Begin

Mpower = remainder \* M;

Ncount = ncount – 1;

// $display(“in not finished: ncount: %d; x: %d”,ncount, x);

End

X=Mpower;

// $display(“out of condition: ncount: %d; x: %d”,ncount, x);

End

Assign finished = (ncount == 0)?1:0;

Endmodule

*Yecheeze last m daalsktebhai but abikeliye ye sbrehenedena okay*

Encryption test bench

`timescale 1ns / 1ps

Module encryptor\_test\_bench;

// Inputs

Reg [15:0] Input;

Reg [7:0] firstPrimeNumber;

Reg [7:0] secondPrimeNumber;

Regclk;

Reg start;

Reg start1;

Reg start2;

// Outputs

Wire [7:0] encryptionKey;

Wire [15:0] n;

Wire [15:0] Output;

Wire [15:0] decryptionKey;

Wire finish;

Wire fin1;

Main #(.InstructionSelector(1)) uut (

.Input(Input),

.firstPrimeNumber(firstPrimeNumber),

.secondPrimeNumber(secondPrimeNumber),

.clk(clk),

.start(start),

.start1(start1),

.start2(start2),

.encryptionKey(encryptionKey),

.n(n),

.Output(Output),

.decryptionKey(decryptionKey),

.finish(finish),

.fin1(fin1)

);

Initial begin

// Initialize Inputs

Input = 0;

firstPrimeNumber = 0;

secondPrimeNumber = 0;

clk = 0;

start = 0;

start1 = 0;

start2 = 0;

// Wait 100 ns for global reset to finish

#10;

Input = 11;

firstPrimeNumber = 67;

secondPrimeNumber = 53;

#10;

Start = 1;

#5;

Start=0;

#40;

Start1=1;

#10;

Start1=0;

#30;

Start2=1;

#10;start2=0;

// Add stimulus here

End

Always #0.1 clk=!clk;

Endmodule

Description test bench

`timescale 1ns / 1ps

*Module decryptor\_test\_bench;*

*// Inputs*

*Reg [15:0] Input;*

*//regInstructionSelector;*

*Reg [7:0] firstPrimeNumber;*

*Reg [7:0] secondPrimeNumber;*

*Regclk;*

*Reg start;*

*Reg start1;*

*Reg start2;*

*// Outputs*

*Wire [7:0] encryptionKey;*

*Wire [15:0] n;*

*Wire [15:0] Output;*

*Wire [15:0] decryptionKey;*

*Wire finish;*

*Wire fin1;*

*// Instantiate the Unit Under Test (UUT)*

*Main #(.InstructionSelector(0)) uut (*

*.Input(Input),*

*.firstPrimeNumber(firstPrimeNumber),*

*.secondPrimeNumber(secondPrimeNumber),*

*.clk(clk),*

*.start(start),*

*.start1(start1),*

*.start2(start2),*

*.encryptionKey(encryptionKey),*

*.n(n),*

*.Output(Output),*

*.decryptionKey(decryptionKey),*

*.finish(finish),*

*.fin1(fin1)*

*);*

*Initial begin*

*// Initialize Inputs*

*Input = 0;*

*firstPrimeNumber = 0;*

*secondPrimeNumber = 0;*

*clk = 0;*

*start = 0;*

*start1 = 0;*

*start2 = 0;*

*#10;*

*Input = 1256;*

*firstPrimeNumber = 67;*

*secondPrimeNumber = 53;*

*#10;*

*Start = 1;*

*#5;*

*Start=0;*

*#40;*

*Start1=1;*

*#10;*

*Start1=0;*

*#30;*

*Start2=1;*

*#10;start2=0;*

*End*

*Always #0.1 clk=!clk;*

*Endmodule*