Lab 11

Rivest-Shamir-Adleman (RSA) Encryption

Name: Syed Asghar Abbas Zaidi	**Student ID: 07201**

11.1 Objective

The Objectives of this lab are:

- Generating an RSA encryption key pair
- Implementing encryption and decryption using RSA

Just like previously, this is how my Task answers will be formatted

Design
Simulation code
Actual simulation

Log files values are in decimal but simulation output is in hexa!

FOLLOWED BY

EXPLANATION

11.2 Background/Scenario

RSA encryption is a crucial part of digital security. It was created in the late 1970s by Ron Rivest, Adi Shamir, and Leonard Adleman. RSA uses two exponents, *e* (public) and *d* (private), along with a large modulus number, *n*, created during key generation.

Both encryption and decryption rely on modular exponentiation, a process made efficient by algorithms like fast exponentiation. It is crucial to emphasize that while performing modular exponentiation is an efficient process, its inverse operation, involving modular logarithm, is an exceptionally challenging task. This complexity arises because solving modular logarithms is as difficult as factoring the modulus, a problem for which no known polynomial-time algorithm exists.

To put it simply, anyone can easily encrypt data quickly when the public exponent (*e*) is known. Likewise, with the private exponent (*d*), decryption is a straightforward process. However, for any unauthorized third party, decrypting the information is nearly impossible due to the formidable computational hurdles involved. This inherent security feature of RSA ensures that sensitive data remains protected from prying eyes.

11.3 Procedure

Figure 1 shows the general idea behind the procedure used in RSA.

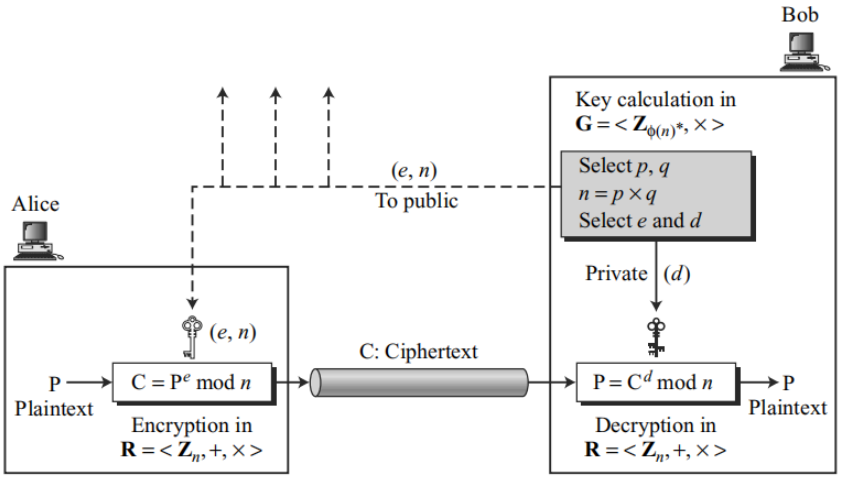


Figure 1: Encryption, Decryption and Key Generation

RSA uses two algebraic structures; a ring and a group.

Encryption/Decryption Ring

Encryption and decryption are done using the commutative ring $R = < Z_n, +, \times >$

with two arithmetic operations: addition and multiplication. In RSA, this ring is public because the modulus *n* is public. Anyone can send a message to Bob using this ring to do encryption.

Key-Generation Group

RSA uses a multiplicative group $G = < Z_{\varphi(n)}*, \times >$

for key generation. This group supports only multiplication and division (using multiplicative inverses), which are needed for generating public and private keys. This group is hidden from the public because its modulus, $\varphi(\pi)$

is hidden from the public.

Task 1: Generate an RSA encryption key pair

To generate both a public and a private key, you can follow the procedure outlined in Figure 2. Once the keys are generated, the public key is represented as the tuple *(e, n)*, and the private key is simply the integer *d.* It is important to note that the values of *p, q*, and $\varphi(\pi)$

can be safely discarded after key generation, as they will not be required unless there is a need to change the private key without altering the modulus – a practice usually discouraged.

For security, it is recommended that each prime, p or q, should have a size of 512 bits, equivalent to 154 decimal digits. This choice results in a modulus, n, that is 1024 bits in size, or 309 digits long.

RSA_Key_Generation

{

Select two large primes p and q such that $p \neq q$.

$n \leftarrow p \times q$

$\phi(n) \leftarrow (p-1) \times (q-1)$

Select e such that $1 < e < \phi(n)$ and e is coprime to $\phi(n)$

$d \leftarrow e^{-1} \bmod \phi(n)$ // d is inverse of e modulo $\phi(n)$

Public_key ← (e, n) // To be announced publicly

Private_key ← d // To be kept secret

return Public_key and Private_key

}

Figure 2: RSA key generation algorithm

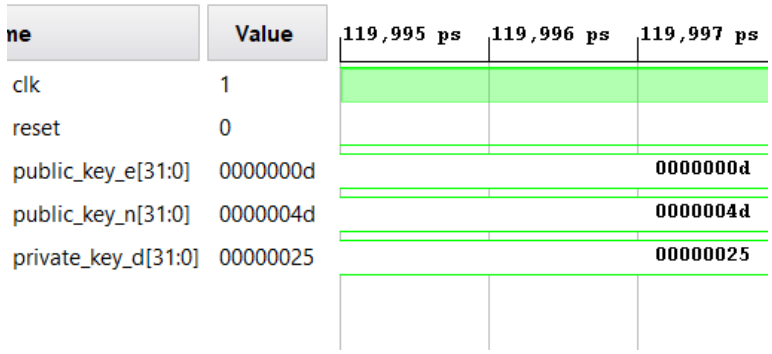
Create a module that implements the functionality shown in Figure 2.

```
// ASGHAR 07201
module RSA_Key_Generation(
        input wire clk,             // Clock signal for synchronization
        input wire reset,           // Reset signal to initialize the module
        output reg [31:0] public_key_e,     // 32-bit register for storing the public exponent
        output reg [31:0] public_key_n,     // 32-bit register for storing the modulus n
        output reg [31:0] private_key_d     // 32-bit register for storing the private key d
);
        // Registers to hold prime numbers, n, phi(n), e, and d
        reg [31:0] p, q, n, phi_n, e, d;
        // State variable for managing the key generation process
        reg [2:0] state;

        // Initial block to set example values for p, q, and e
```

```
        initial begin
        p = 7;  // Example prime number p
        q = 11; // Example prime number q
        e = 13; // Example public exponent e
        state = 0; // Initialize the state to 0
        end

        // Always block triggered on positive clock edge or reset
        always @(posedge clk or posedge reset) begin
        if (reset) begin
        // Reset logic: Initialize registers when reset is active
        p <= 7;
        q <= 11;
        n <= 0;
        phi_n <= 0;
        e <= 13;
        d <= 0;
        state <= 0;
        end else begin
        // State machine for RSA key generation
        case (state)
        0: begin
        // Calculate n and phi(n) in state 0
        n <= p * q;  // n is the product of p and q
        phi_n <= (p - 1) * (q - 1);  // phi(n) = (p-1)*(q-1)
        state <= 1;  // Move to next state
        end
        1: begin
        // Calculate modular inverse of e modulo phi(n) in state 1
        d <= modinv(e, phi_n);  // Compute the private key d
        state <= 2;  // Move to the next state
        end
        2: begin
        // Assign the public and private keys to output registers in state 2
        public_key_e <= e;          // Public exponent
        public_key_n <= n;          // Modulus
        private_key_d <= d;         // Private key
        state <= 3;  // Finish the key generation process
        end
        default: state <= 3;  // Final state
        endcase
        end
        end

        // Function to calculate the modular inverse using the Extended Euclidean Algorithm
  function [31:0] modinv(input [31:0] a, input [31:0] m);
        integer m0, t, q;
        integer x0, x1;
        begin
        m0 = m;  // Store the original modulus value
        t = 0;
        q = 0;
        x0 = 0;
        x1 = 1;

        // Extended Euclidean Algorithm to find the modular inverse
        while (a > 1) begin
        q = a / m;
        t = m;
        m = a % m;
        a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
        end

        // Ensure that x1 is positive
        if (x1 < 0)
        x1 = x1 + m0;

        modinv = x1;  // Return the modular inverse
        end
        endfunction
endmodule
```

\\ ASGHAR
`timescale 1ns / 1ps

```
module testbench_keygen();
        // Declare the signals for the clock, reset, and RSA key outputs
        reg clk;                // Clock signal for synchronization
        reg reset;              // Reset signal to initialize the design
        wire [31:0] public_key_e;       // Wire to capture the public exponent
        wire [31:0] public_key_n;       // Wire to capture the modulus n
        wire [31:0] private_key_d;      // Wire to capture the private key d

        // Instantiate the RSA_Key_Generation module
        RSA_Key_Generation uut (
        .clk(clk),              // Connect the clock signal to the module
        .reset(reset),          // Connect the reset signal to the module
        .public_key_e(public_key_e), // Connect the public exponent wire to the module
        .public_key_n(public_key_n), // Connect the modulus wire to the module
        .private_key_d(private_key_d) // Connect the private key wire to the module
        );

        // Initial block for setting up the simulation
        initial begin
        // Initialize the clock and reset signals
        clk = 0;                // Set clock to 0 initially
        reset = 0;              // Set reset to 0 initially

        // Apply reset to initialize the design
        reset = 1;              // Assert reset to initialize the design
        #10 reset = 0;          // Deassert reset after 10 time units

        // Wait for the key generation process to complete
        #100;                   // Wait for 100 time units (key generation time)

        // Display the generated public and private keys
        $display("Public Key (e, n): (%d, %d)", public_key_e, public_key_n); // Display public key
        $display("Private Key (d): %d", private_key_d);                      // Display private key

        // End the simulation after displaying the keys
        #10 $finish;            // Terminate the simulation after 10 time units
        end

        // Clock generation block
        always #5 clk = ~clk;           // Toggle the clock every 5 time units
endmodule
```




```
#     set_property needs_save false [current_wave_config]
#   } else {
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Wav
#   }
# }
# run 1000ns
Public Key (e, n): (        13,         77)
Private Key (d):         37
$finish called at time : 120 ns : File "C:/Users/DELL/OneDrive - Habib University/Pictures/work/University/Semester 7/Cryptography/Labs/Lab 11/project_1/project_1.sr
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_keygen_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:09 . Memory (MB): peak = 1016.117 ; gain = 0.000
```

EXPLANATION

In this testbench, the primary goal is to simulate the behavior of the RSA key generation module. The testbench starts by defining the clock, reset, and key output signals. The clock signal is used to synchronize the operations, and the reset signal is used to initialize the RSA key generation module.

First, the clock and reset signals are initialized. The reset is asserted to ensure the design starts from a known state. After a short delay, the reset is deasserted, and the design is allowed to begin generating keys.

The testbench then waits for a period to allow enough time for the key generation process to complete. Once the keys are generated, the testbench outputs the generated public and private keys to the console. This helps verify that the key generation module is functioning correctly.

Finally, the simulation ends after a brief delay. The clock signal is toggled continuously at regular intervals, ensuring that the RSA key generation module receives the necessary clock pulses throughout the simulation. This setup tests whether the module correctly generates and outputs the public and private keys.

Task 2: RSA Encryption

Messages can be sent by anyone using this public key. In RSA, the encryption process can be executed using an algorithm that operates with polynomial time complexity, as demonstrated in Figure 3. It is worth noting that the size of the plaintext must be smaller than n. If the plaintext exceeds this limit, it should be divided into blocks for proper encryption.

```
RSA_Encryption (P, e, n)                  // P is the plaintext in Z_n and P < n
{
   C ← Fast_Exponentiation (P, e, n)      // Calculation of (P^e mod n)
   return C
}
```

Figure 3: RSA encryption

Create a module that implements the functionality shown in Figure 3.

EXPLANATION FOR WHAT WE ARE DOING

The fast exponentiation and RSA encryption modules are designed to perform modular exponentiation and RSA encryption, respectively, both crucial in cryptography. Fast exponentiation is particularly efficient because it reduces the number of multiplications needed to compute large powers, making it ideal for RSA, where encryption often involves raising large numbers to high powers under a modulus.

The fast exponentiation module takes three inputs: the base, exponent, and modulus. Its goal is to calculate (base^exp) % mod using an iterative method, sometimes called "exponentiation by squaring." The base is first reduced modulo the modulus to ensure all calculations are within a limited range. The algorithm then iteratively squares the base and, depending on the bits of the exponent, multiplies the result by the current base when a bit is set to 1. This efficient method avoids the direct computation of high powers and keeps intermediate values small due to the modular reduction in each step, ensuring performance within typical constraints of digital circuits.

The RSA encryption module uses the fast exponentiation module to perform encryption by computing ciphertext = (plaintext^e) % n, where plaintext is the original message, e is the public exponent, and n is the modulus. The RSA encryption module acts as a wrapper, passing the plaintext, exponent, and modulus to the fast exponentiation module and then assigning the resulting value to the ciphertext output.

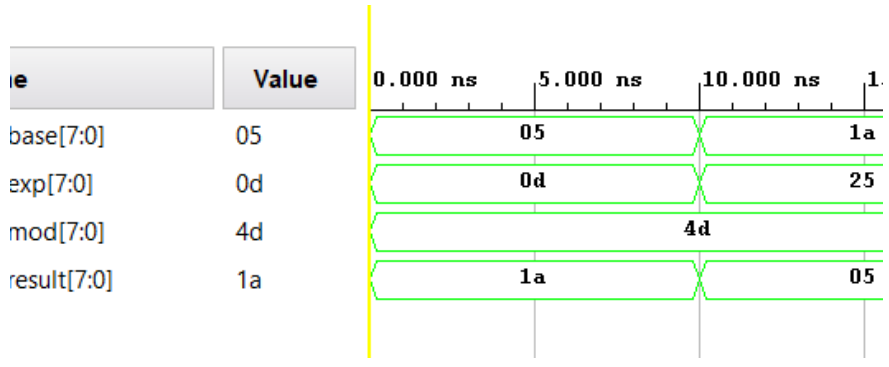
In the simulation, a testbench runs two test cases to verify the encryption process. The testbench sets different values for plaintext, exponent, and modulus, and after a short delay, it displays the resulting ciphertext for each case. This output allows verification that the RSA encryption module is correctly calculating the ciphertext by raising the plaintext to the power of e modulo n. Through the test cases, the testbench confirms that the system accurately encrypts data according to the RSA algorithm, utilizing efficient modular exponentiation for fast calculations. The results can be used to verify the encryption's integrity and correctness in each case.

FAST EXPONENTIATION

```
// Asghar
// Set the time unit and time precision for simulation (1 nanosecond time unit, 1 picosecond time precision)
`timescale 1ns / 1ps

// Define the fast_exponentiation module with a parameterizable bit-width, default width is 7 bits
module fast_exponentiation #(parameter width = 7) (
        input [width:0] base,       // Input base value for exponentiation
        input [width:0] exp,        // Input exponent value
        input [width:0] mod,        // Input modulus for modular arithmetic
        output reg [width:0] result             // Output result of the exponentiation
);

        // Internal 16-bit registers for intermediate calculations
        reg [15:0] temp_base;     // Temporary base used in calculations
        reg [15:0] temp_result;   // Temporary result to hold intermediate results
        integer i;                // Loop counter

        // Define combinational logic for fast exponentiation
        always @(*) begin
        // Initialize temp_base as base % mod for modular arithmetic
        temp_base = base % mod;
        // Initialize temp_result to 1 (identity element for multiplication)
        temp_result = 1;

        // Perform fast exponentiation using the "square-and-multiply" method
        for (i = 0; i < 7; i = i + 1) begin
        // If the current bit of exponent is 1, multiply temp_result by temp_base modulo mod
        if (exp[i] == 1) begin
        temp_result = (temp_result * temp_base) % mod;
        end
        // Square temp_base modulo mod for the next iteration
        temp_base = (temp_base * temp_base) % mod;
        end
```

```
                // Assign the final result after loop execution
                result = temp_result;
                end
endmodule
```

```
// Set the time unit and precision for simulation (1 nanosecond time unit, 1 picosecond precision)
`timescale 1ns / 1ps

// Define the fast_exp_test module to test the fast_exponentiation module
module fast_exp_test();

                // Declare 8-bit registers for inputs to the fast exponentiation module
                reg [7:0] base;         // Base value for exponentiation
                reg [7:0] exp;          // Exponent value
                reg [7:0] mod;          // Modulus for modular arithmetic
                wire [7:0] result; // Output wire for the computed result

                // Instantiate the fast_exponentiation module (assumed to be defined elsewhere)
                // Connect input and output ports to the declared testbench signals
                fast_exponentiation x (
                .base(base),
                .exp(exp),
                .mod(mod),
                .result(result)
                );

                // Begin an initial block for simulation
                initial begin
                // Test case 1
                base = 8'd5; // Set base to 5
                exp = 8'd13;            // Set exponent to 13
                mod = 8'd77;            // Set modulus to 77
                #10;            // Wait for 10 time units
                // Display the test case results on the console
                $display("Test case 1: base=%d, exp=%d, mod=%d, result=%d", base, exp, mod, result);

                // Test case 2
                base = 8'd26;           // Set base to 26
                exp = 8'd37;            // Set exponent to 37
                mod = 8'd77;            // Set modulus to 77
                #10;            // Wait for 10 time units
                // Display the test case results on the console
                $display("Test case 2: base=%d, exp=%d, mod=%d, result=%d", base, exp, mod, result);

                // End the simulation
                $finish;
                end

endmodule
```

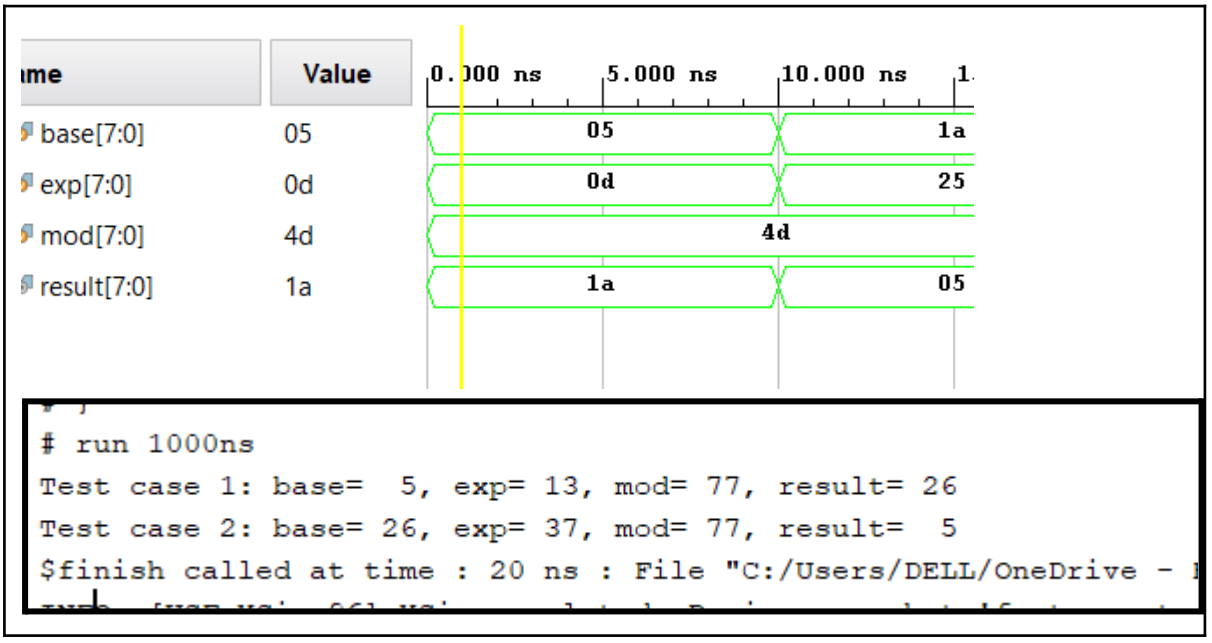


RSA ENCRYPTION

```
// Set the time unit and precision for simulation (1 nanosecond time unit, 1 picosecond precision)
`timescale 1ns / 1ps

// ASGHAR 07201
// Define the RSA_Encryption module, which performs RSA encryption
```

```
module RSA_Encryption(
        input [7:0] plaintext,  // 8-bit plaintext to be encrypted
        input [7:0] e,          // 8-bit public exponent in RSA encryption
        input [7:0] n,          // 8-bit modulus used in RSA encryption
        output [7:0] ciphertext // 8-bit encrypted ciphertext output
);

        // Intermediate wire to hold the result from the fast exponentiation module
        wire [7:0] result;

        // Instantiate the fast_exponentiation module to perform modular exponentiation
        // The plaintext, exponent `e`, and modulus `n` are passed as inputs
        // The result is the computed (plaintext^e) % n
        fast_exponentiation exp (
        .base(plaintext), // Base is the plaintext message
        .exp(e),        // Exponent is the public exponent `e`
        .mod(n),        // Modulus is `n`
        .result(result)   // Result of exponentiation is stored in `result`
        );

        // Assign the result of the modular exponentiation to the ciphertext output
        assign ciphertext = result;

endmodule
```

```
`timescale 1ns / 1ps

// ASGHAR 07201
// Define the rsa_encrypt_test module to test the RSA_Encryption module
module rsa_encrypt_test();

        // Declare 8-bit registers for inputs to the RSA encryption module
        reg [7:0] plaintext;  // Plaintext message to be encrypted
        reg [7:0] e;          // Public exponent in RSA encryption
        reg [7:0] n;          // Modulus used in RSA encryption
        wire [7:0] ciphertext; // Output wire for the encrypted ciphertext

        // Instantiate the RSA_Encryption module
        // Connect input and output ports to the declared testbench signals
        RSA_Encryption x (
        .plaintext(plaintext),
        .e(e),
        .n(n),
        .ciphertext(ciphertext)
        );

        // Begin an initial block for running the test cases in simulation
        initial begin
        // Test case 1
        plaintext = 8'd26;  // Set plaintext to 26
        e = 8'd37;          // Set public exponent e to 37
        n = 8'd77;          // Set modulus n to 77
        #10;                // Wait for 10 time units
        // Display the result of test case 1 in the console
        $display("Test case 1: plaintext=%d, e=%d, n=%d, ciphertext=%d", plaintext, e, n, ciphertext);

        // Test case 2
        plaintext = 8'd5;   // Set plaintext to 5
        e = 8'd13;          // Set public exponent e to 13
        n = 8'd77;          // Set modulus n to 77
        #10;                // Wait for 10 time units
        // Display the result of test case 2 in the console
        $display("Test case 2: plaintext=%d, e=%d, n=%d, ciphertext=%d", plaintext, e, n, ciphertext);

        // End the simulation
        $finish;
        end

endmodule
```



Task 3: RSA Decryption

The receiver can decrypt the received ciphertext message using the algorithm in figure 4. The RSA decryption process employs an algorithm with polynomial time complexity. It is important to highlight that the size of the ciphertext is always smaller than the modulus, n.

```
RSA_Decryption (C, d, n)                    //C is the ciphertext in Z_n
{
   P ← Fast_Exponentiation (C, d, n)        // Calculation of (C^d mod n)
   return P
}
```

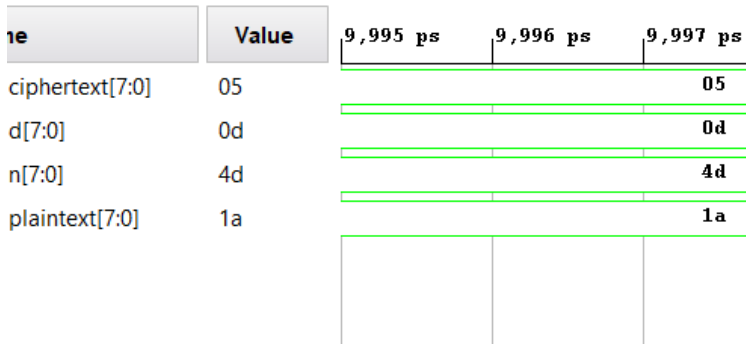
Figure 4: RSA decryption

Create a module that implements the functionality shown in Figure 4.

```
// ASGHAR 07201
// Set the time unit and precision for simulation (1 nanosecond time unit, 1 picosecond precision)
`timescale 1ns / 1ps

// Define the RSA_Decryption module, which performs RSA decryption
module RSA_Decryption(
        input [7:0] ciphertext,  // 8-bit ciphertext input, representing the encrypted message
        input [7:0] d,           // 8-bit private exponent used in RSA decryption
        input [7:0] n,           // 8-bit modulus for the decryption operation
        output [7:0] plaintext   // 8-bit plaintext output, representing the decrypted message
);

        // Intermediate wire to hold the result from the fast_exponentiation module
        wire [7:0] result;
```

```
        // Instantiate the fast_exponentiation module to perform modular exponentiation
        // In RSA decryption, this computes (ciphertext^d) % n to retrieve the original plaintext
        fast_exponentiation exp1 (
        .base(ciphertext), // Base is the encrypted message (ciphertext)
        .exp(d),           // Exponent is the private exponent `d`
        .mod(n),           // Modulus is `n`
        .result(result)    // Result of exponentiation is stored in `result`
        );

        // Assign the result of the modular exponentiation to the plaintext output
        assign plaintext = result;

endmodule
```

```
// ASGHAR 07201
// Set the time unit and precision for simulation (1 nanosecond time unit, 1 picosecond precision)
`timescale 1ns / 1ps

// Testbench for the RSA_Decryption module
module rsa_decrypt_test();
        reg [7:0] ciphertext;  // 8-bit register to store the ciphertext input
        reg [7:0] d;           // 8-bit register for the private exponent used in decryption
        reg [7:0] n;           // 8-bit register for the modulus
        wire [7:0] plaintext;  // 8-bit wire to observe the plaintext output after decryption

        // Instantiate the RSA_Decryption module
        RSA_Decryption x (
        .ciphertext(ciphertext), // Connect ciphertext input to RSA_Decryption
        .d(d),                   // Connect private exponent `d` to RSA_Decryption
        .n(n),                   // Connect modulus `n` to RSA_Decryption
        .plaintext(plaintext)    // Connect plaintext output to RSA_Decryption
        );

        initial begin
        // Test case 1: Test decryption with given ciphertext, private exponent `d`, and modulus `n`
        ciphertext = 8'd5;       // Example ciphertext value
        d = 8'd13;               // Private exponent `d` (inverse of e mod phi(n)), derived as d = e^-1 mod 60
        n = 8'd77;               // Modulus value used in both encryption and decryption
        #10;                     // Delay for 10 time units to allow decryption operation to complete

        // Display the result for verification
        $display("Test case 1: ciphertext=%d, d=%d, n=%d, plaintext=%d", ciphertext, d, n, plaintext);

        // End the simulation after displaying results
        $finish;
        end
endmodule
```



```
#    }
# }
# run 1000ns
Test case 1: ciphertext=  5, d= 13, n= 77, plaintext= 26
$finish called at time : 10 ns : File "C:/Users/DELL/OneDrive -
INFO: [USF-XSim-96] XSim completed. Design snapshot 'rsa decrypt
```

Explanation

The RSA decryption logic in this module leverages modular exponentiation to reverse the encryption process and retrieve the original plaintext. In RSA encryption, a message (plaintext) is raised to the power of a public exponent e and taken modulo a large number n, which is the product of two prime numbers. The resulting value is the ciphertext. To decrypt, the ciphertext must be raised to the power of the private exponent d, which is derived from e and φ(n), where φ(n) is the Euler's totient of n. This decryption operation computes the original message as (ciphertext^d) mod n.

In this module, the RSA_Decryption component receives the ciphertext, private exponent d, and modulus n as inputs. The decryption is performed by passing these inputs to a fast exponentiation module, which is efficient at computing large powers under a modulus. The result of this modular exponentiation is assigned to the plaintext output, recovering the original message.

The testbench, rsa_decrypt_test, initializes with a set of inputs to test the decryption logic. In this simulation, we set ciphertext, d, and n values for one test case. The ciphertext value is the encrypted message, d is the private exponent (calculated as the modular inverse of e), and n is the modulus. After a brief delay, the testbench displays the plaintext output, which should match the original message if decryption is successful. The simulation is set to end immediately after displaying the output, allowing us to verify that the decryption operation correctly restores the original plaintext from the ciphertext based on the given inputs. This setup confirms that the decryption logic works as intended by successfully reversing the RSA encryption process.

Task 4: Verifying Implementation

Test your implementation by performing the following steps

1. Create an RSA key.
2. Encrypt a message with this key.
3. Decrypt the resulting ciphertext.
4. Compare the decrypted message with the original message.

For submission, please include all your Verilog files and a PDF document containing modules, results, and explanations where necessary

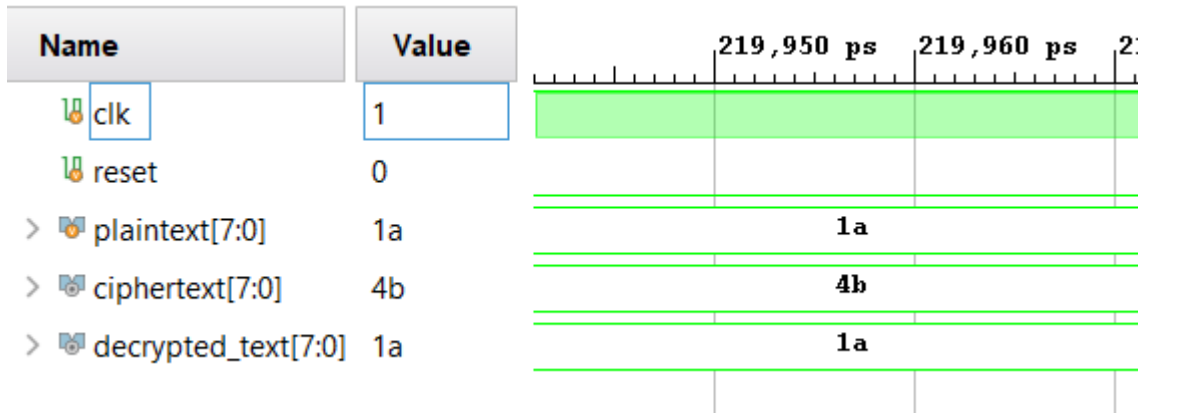
```
`timescale 1ns / 1ps
module RSA(
        input wire clk,
        input wire reset,
```

```
    input wire [7:0] plaintext,
    output wire [7:0] ciphertext,
    output wire [7:0] decrypted_text
);

    // Wires to connect the modules
    wire [31:0] public_key_e;
    wire [31:0] public_key_n;
    wire [31:0] private_key_d;
    wire [7:0] encrypted_text;

    // Instantiate the RSA Key Generation module
    RSA_Key_Generation rsa_key_gen (
    .clk(clk),
    .reset(reset),
    .public_key_e(public_key_e),
    .public_key_n(public_key_n),
    .private_key_d(private_key_d)
    );

    // Instantiate the RSA Encryption module
    RSA_Encryption rsa_encryption (
    .plaintext(plaintext),
    .e(public_key_e[7:0]),      // Only use the lower 8 bits
    .n(public_key_n[7:0]),      // Only use the lower 8 bits
    .ciphertext(encrypted_text)
    );

    // Instantiate the RSA Decryption module
    RSA_Decryption rsa_decryption (
    .ciphertext(encrypted_text),
    .d(private_key_d[7:0]),     // Only use the lower 8 bits
    .n(public_key_n[7:0]),      // Only use the lower 8 bits
    .plaintext(decrypted_text)
    );
    assign ciphertext = encrypted_text;
endmodule
```

```
`timescale 1ns / 1ps

module RSA_Test();
    // Declare registers and wires for the testbench
    reg clk;                    // Clock signal
    reg reset;                  // Reset signal
    reg [7:0] plaintext;        // 8-bit plaintext input
    wire [7:0] ciphertext;      // 8-bit ciphertext output
    wire [7:0] decrypted_text;          // 8-bit decrypted text output

    // Instantiate the RSA module (top-level)
    RSA rsa_top (
    .clk(clk),                  // Connect the clock signal
    .reset(reset),              // Connect the reset signal
    .plaintext(plaintext),      // Connect the plaintext input
    .ciphertext(ciphertext),    // Connect the ciphertext output
    .decrypted_text(decrypted_text) // Connect the decrypted text output
    );

    // Initial block for setting up the testbench
    initial begin
    // Initialize signals
    clk = 0;                    // Set the clock to 0 initially
    reset = 0;                  // Set reset to 0 initially
    plaintext = 8'd26;          // Set plaintext to 26 (arbitrary value for testing)

    // Assert reset signal for 10 time units
    reset = 1;
    #10 reset = 0;              // Deassert reset after 10 time units

    // Wait for key generation and processing (RSA computation)
    #100;

    // Display public and private keys used for encryption and decryption
    $display("Public Key (e, n): (%d, %d)", rsa_top.public_key_e, rsa_top.public_key_n);
    $display("Private Key (d): %d", rsa_top.private_key_d);

    // Wait for encryption and decryption operations to complete
    #100;

    // Display the results of encryption and decryption
    $display("Plaintext: %d", plaintext);           // Display original plaintext
    $display("Ciphertext: %d", ciphertext);         // Display encrypted ciphertext
    $display("Decrypted Text: %d", decrypted_text); // Display the decrypted text (should match plaintext)
```

```
        // End simulation
        #10 $finish;
        end

        // Clock generation: Toggle clock every 5 time units
        always #5 clk = ~clk;

endmodule
```

```
# run 1000ns
Public Key (e, n): (         13,          77)
Private Key (d):          37
Plaintext:  26
Ciphertext:  75
Decrypted Text:  26
$finish called at time : 220 ns : File "C:/Users/DELL/OneDrive - Habib University/Pictures/work/University/Semester 7/Cryptography/Labs/Lab 11/project_1/project_1.srcs/sim
xsim: Time (s): cpu = 00:00:00 ; elapsed = 00:00:05 . Memory (MB): peak = 1016.117 ; gain = 0.000
INFO: [USF-XSim-96] XSim completed. Design snapshot 'RSA_Test_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:00 ; elapsed = 00:00:08 . Memory (MB): peak = 1016.117 ; gain = 0.000
```



EXPLANATION:

In the final top module design, we created a system that implements the full RSA encryption and decryption process. The module is structured to integrate multiple submodules, including key generation, encryption, and decryption. The RSA testbench controls the overall simulation flow by providing inputs such as plaintext and generating corresponding outputs for ciphertext and decrypted text. The RSA module itself is connected to the top-level testbench through signals like clock, reset, and plaintext. The testbench also generates the clock signal that drives the module and handles reset functionality. In the simulation code, the reset is asserted for a brief period to initialize the system, after which key generation occurs. The testbench waits for the RSA key generation, encryption, and decryption processes to complete before displaying the public key, private key, ciphertext, and the decrypted text. The decrypted text is expected to match the original plaintext, verifying that the encryption and decryption processes work as intended. Finally, the simulation ends, providing a comprehensive test of the RSA functionality.

Verilog files are uploaded on this drive link:

Lab11 Verilog

Assessment Rubric

Lab 11

Rivest-Shamir-Adleman (RSA) Encryption

Name: Syed Asghar Abbas Zaidi	**Student ID: 07201**

Points Distribution

Task No.	**LR 2 Code**	**LR5 Results**	**LR9 Report**
Task 1	30		
Task 2	20		
Task 3	20		
Task 4		20	
Total	/70	/20	/10
CLO Mapped	CLO 2	CLO 2	CLO2

CLO	**Total Points**	**Points Obtained**
2	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer the provided Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics.

Lab Evaluation Assessment Rubric

#	Assessment Elements	Level 1: Unsatisfactory Points 0-1	Level 2: Developing Points 2	Level 3:Good Points 3	Level 4:Exemplary Points 4
LR2	**Program/Code / Simulation Model/ Network Model**	Program/code/simulation model/network model does not implement the required functionality and has several errors. The student is not able to utilize even the basic tools of the software.	Program/code/simulation model/network model has some errors and does not produce completely accurate results. Student has limited command on the basic tools of the software.	Program/code/simulation model/network model gives correct output but not efficiently implemented or implemented by computationally complex routine.	Program/code/simulation /network model is efficiently implemented and gives correct output. Student has full command on the basic tools of the software.
LR5	**Results & Plots**	Figures/ graphs / tables are not developed or are poorly constructed with erroneous results. Titles, captions, units are not mentioned. Data is presented in an obscure manner.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is not too clear.	All figures, graphs, tables are correctly drawn but contain minor errors or some of the details are missing.	Figures / graphs / tables are correctly drawn and appropriate titles/captions and proper units are mentioned. Data presentation is systematic.
LR9	**Report**	All the in-lab tasks are not included in report and / or the report is submitted too late.	Most of the tasks are included in report but are not well explained. All the necessary figures / plots are not included. Report is submitted after due date.	Good summary of most the in-lab tasks is included in report. The work is supported by figures and plots with explanations. The report is submitted timely.	Detailed summary of the in-lab tasks is provided. All tasks are included and explained well. Data is presented clearly including all the necessary figures, plots and tables.