XOR-Shift-Permutation (XSP) Encryption and Decryption in Verilog

Overview

The XOR-Shift-Permutation (XSP) encryption algorithm is designed to encrypt 8-bit plaintext blocks using an 8-bit key. This encryption scheme involves a series of bitwise operations, circular shifts, and bit permutations to transform the plaintext into ciphertext. The decryption process involves reversing these operations to retrieve the original plaintext. This documentation provides a detailed explanation of the algorithm, its implementation in Verilog, and a comprehensive testbench to verify its functionality.

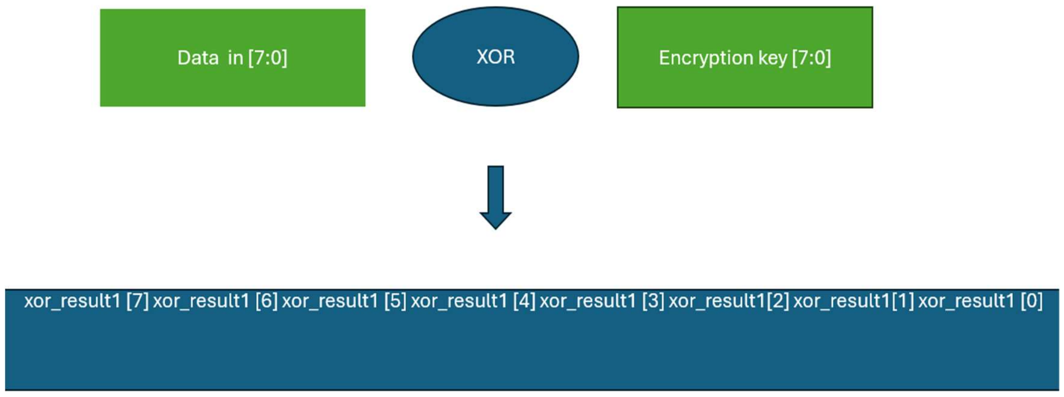
Encryption Algorithm

Initialization

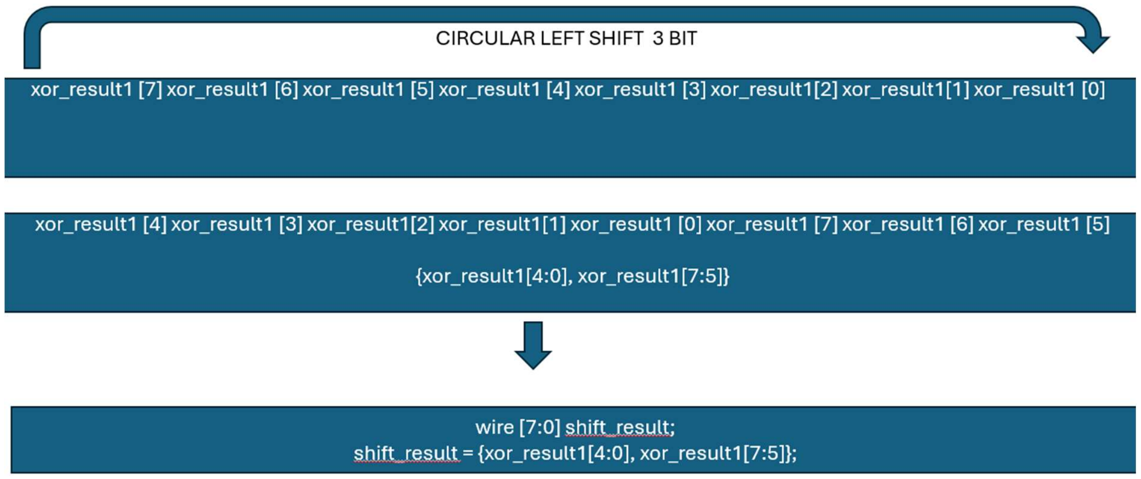
The encryption process begins with the initialization of an 8-bit plaintext block and an 8-bit key. These inputs are essential for the subsequent encryption steps.

Encryption Steps

1. **XOR Operation**: The first step in the encryption process is to XOR the plaintext block with the key. This operation provides the initial layer of obfuscation.

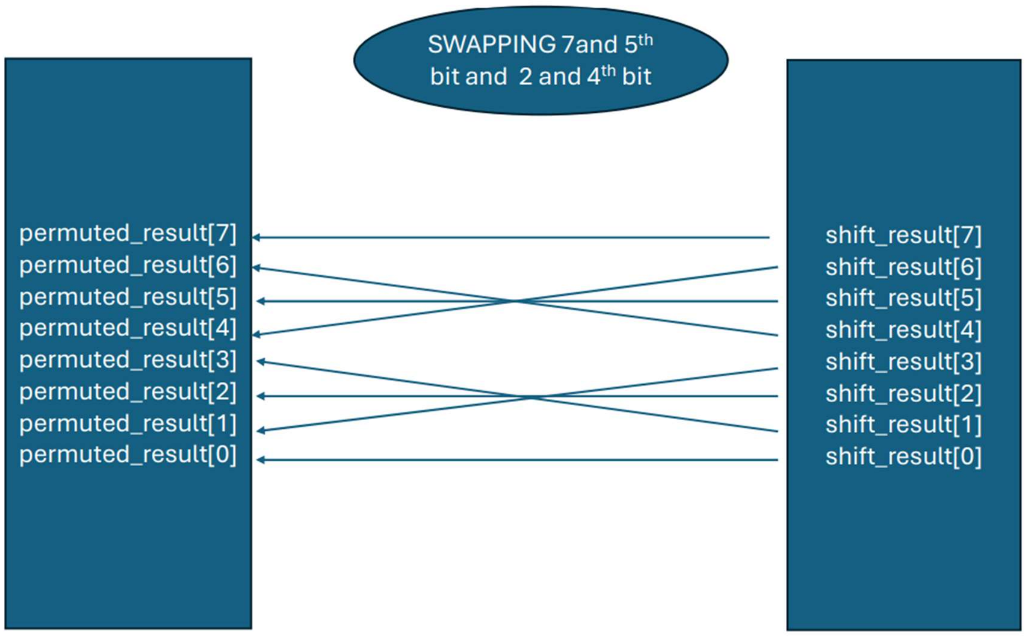

Data in [7:0]
XOR
Encryption key [7:0]
xor_result1 [7] xor_result1 [6] xor_result1 [5] xor_result1 [4] xor_result1 [3] xor_result1[2] xor_result1[1] xor_result1 [0]

2. **Circular Left Shift**: The result of the XOR operation is then circularly left-shifted by 3 bits. Circular shifting means that bits shifted out of the leftmost position are reintroduced at the rightmost position, ensuring no loss of information.

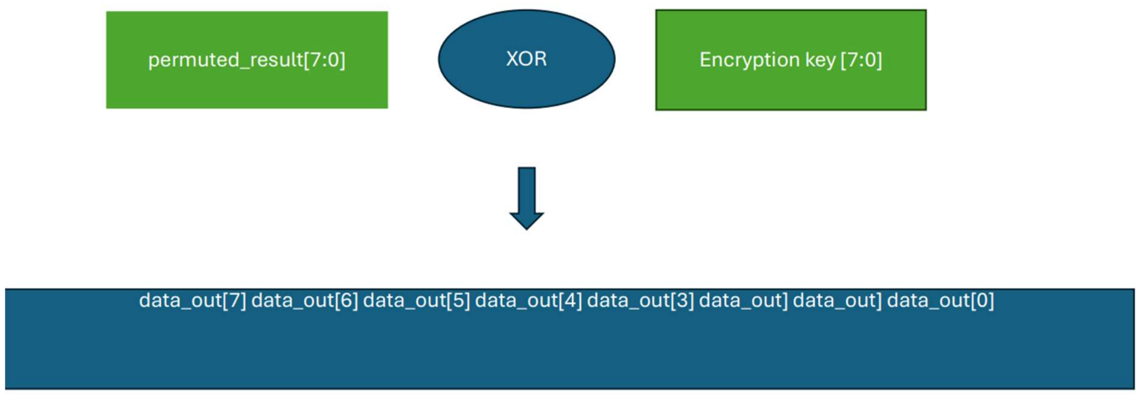

CIRCULAR LEFT SHIFT 3 BIT
xor_result1 [7] xor_result1 [6] xor_result1 [5] xor_result1 [4] xor_result1 [3] xor_result1[2] xor_result1[1] xor_result1 [0]
xor_result1 [4] xor_result1 [3] xor_result1[2] xor_result1[1] xor_result1 [0] xor_result1 [7] xor_result1 [6] xor_result1 [5]
{xor_result1[4:0], xor_result1[7:5]}

```
wire [7:0] shift_result;
shift_result = {xor_result1[4:0], xor_result1[7:5]};
```

3. **Permutation**: The next step involves permuting specific bits of the shifted result. Specifically, the 2nd and 4th bits are swapped, and the 5th and 7th bits are swapped. This permutation step further scrambles the data to enhance security.



4. **Final XOR Operation**: The permuted result is then XORed with the original key again to produce the final encrypted output. This second XOR operation ensures that even small changes in the key or plaintext produce significantly different ciphertexts.


permuted_result[7:0]
XOR
Encryption key [7:0]
data_out[7] data_out[6] data_out[5] data_out[4] data_out[3] data_out] data_out] data_out[0]

Decryption Algorithm

Initialization

The decryption process starts with the encrypted 8-bit ciphertext block and the same 8-bit key used for encryption.

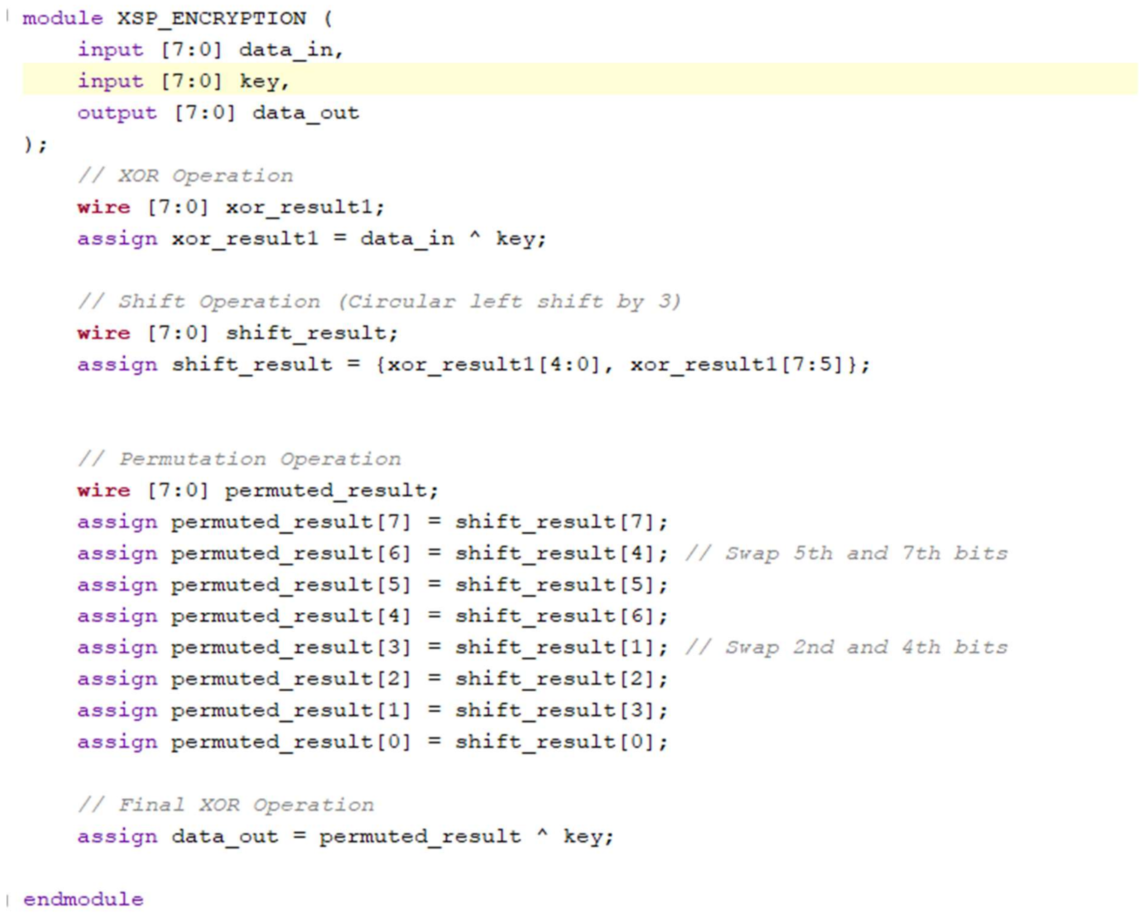
Decryption Steps

1. **Final XOR Operation**: The decryption process begins by XORing the ciphertext block with the key. This operation reverses the final XOR operation performed during encryption.
2. **Reverse Permutation**: The result of the XOR operation is then subjected to a reverse permutation. The bits that were swapped during encryption are swapped back to their original positions. Specifically, the 2nd and 4th bits, as well as the 5th and 7th bits, are swapped back.
3. **Circular Right Shift**: The permuted result is then circularly right-shifted by 3 bits. This operation reverses the circular left shift performed during encryption, restoring the bit positions to their original order.
4. **Initial XOR Operation**: The final step is to XOR the shifted result with the key again. This operation reverses the initial XOR operation from the encryption process, producing the original plaintext.

Verilog Implementation

Encryption Module (`XSP_ENCRYPTION.v`)

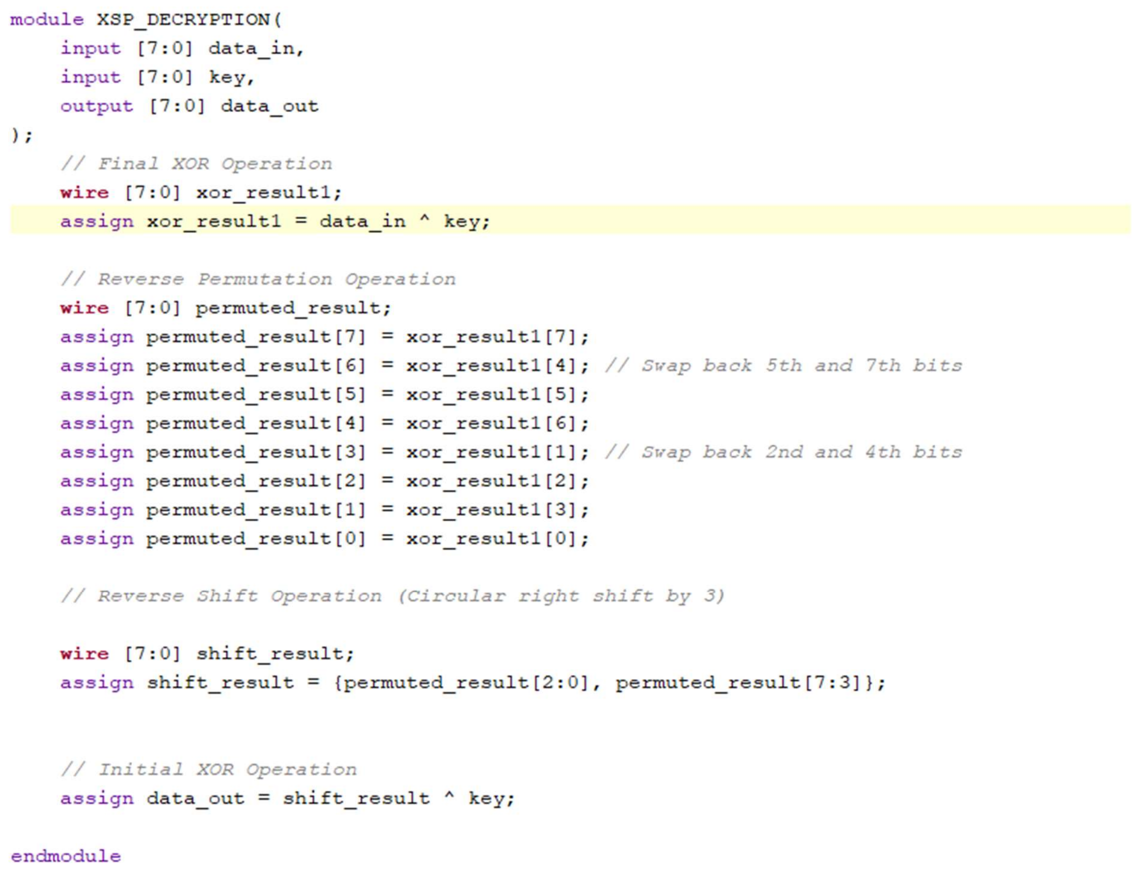
The encryption module implements the XSP encryption algorithm. It takes an 8-bit plaintext block and an 8-bit key as inputs and produces an 8-bit ciphertext block as output.



```verilog
module XSP_ENCRYPTION (
    input [7:0] data_in,
    input [7:0] key,
    output [7:0] data_out
);
    // XOR Operation
    wire [7:0] xor_result1;
    assign xor_result1 = data_in ^ key;

    // Shift Operation (Circular left shift by 3)
    wire [7:0] shift_result;
    assign shift_result = {xor_result1[4:0], xor_result1[7:5]};


    // Permutation Operation
    wire [7:0] permuted_result;
    assign permuted_result[7] = shift_result[7];
    assign permuted_result[6] = shift_result[4]; // Swap 5th and 7th bits
    assign permuted_result[5] = shift_result[5];
    assign permuted_result[4] = shift_result[6];
    assign permuted_result[3] = shift_result[1]; // Swap 2nd and 4th bits
    assign permuted_result[2] = shift_result[2];
    assign permuted_result[1] = shift_result[3];
    assign permuted_result[0] = shift_result[0];

    // Final XOR Operation
    assign data_out = permuted_result ^ key;

endmodule
```

Decryption Module (`XSP_DECRYPTION.v`)

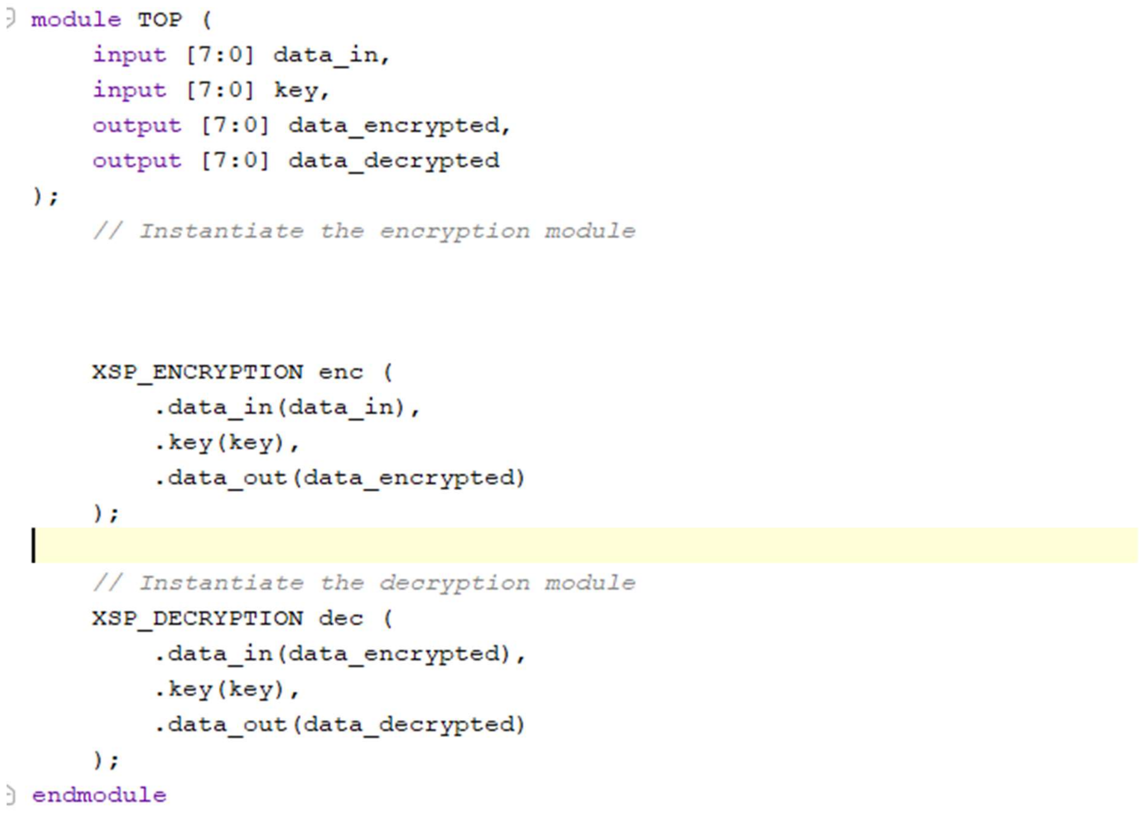
The decryption module implements the XSP decryption algorithm. It takes an 8-bit ciphertext block and an 8-bit key as inputs and produces an 8-bit plaintext block as output.



```
module XSP_DECRYPTION(
    input [7:0] data_in,
    input [7:0] key,
    output [7:0] data_out
);
    // Final XOR Operation
    wire [7:0] xor_result1;
    assign xor_result1 = data_in ^ key;

    // Reverse Permutation Operation
    wire [7:0] permuted_result;
    assign permuted_result[7] = xor_result1[7];
    assign permuted_result[6] = xor_result1[4]; // Swap back 5th and 7th bits
    assign permuted_result[5] = xor_result1[5];
    assign permuted_result[4] = xor_result1[6];
    assign permuted_result[3] = xor_result1[1]; // Swap back 2nd and 4th bits
    assign permuted_result[2] = xor_result1[2];
    assign permuted_result[1] = xor_result1[3];
    assign permuted_result[0] = xor_result1[0];

    // Reverse Shift Operation (Circular right shift by 3)

    wire [7:0] shift_result;
    assign shift_result = {permuted_result[2:0], permuted_result[7:3]};


    // Initial XOR Operation
    assign data_out = shift_result ^ key;

endmodule
```

Top-Level Module (`TOP.v`)

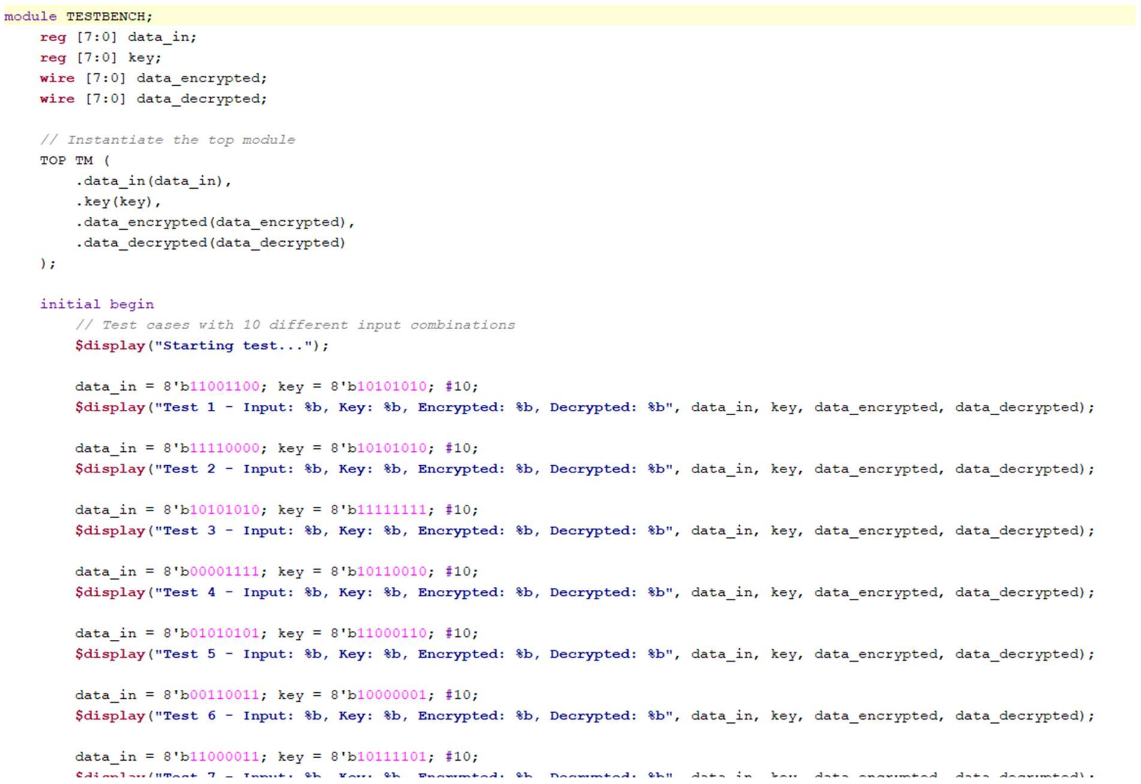
The top-level module integrates both the encryption and decryption modules. It facilitates the testing of the encryption-decryption process by providing a single interface for both operations.



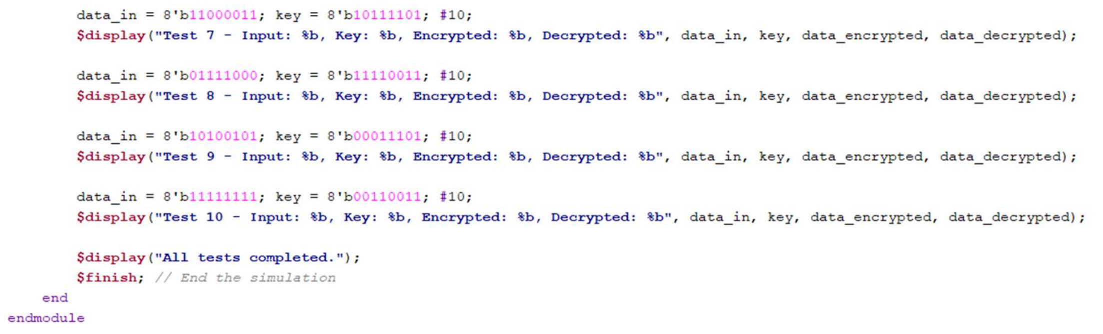
```
module TOP (
    input [7:0] data_in,
    input [7:0] key,
    output [7:0] data_encrypted,
    output [7:0] data_decrypted
);
    // Instantiate the encryption module



    XSP_ENCRYPTION enc (
        .data_in(data_in),
        .key(key),
        .data_out(data_encrypted)
    );

    // Instantiate the decryption module
    XSP_DECRYPTION dec (
        .data_in(data_encrypted),
        .key(key),
        .data_out(data_decrypted)
    );
endmodule
```

Testbench for Top-Level Module (`tb_xsp_top.v`)

The testbench verifies the functionality of the top-level module by testing it with multiple input combinations. The expected behavior is that the decrypted output matches the original plaintext for all test cases.



```verilog
module TESTBENCH;
    reg [7:0] data_in;
    reg [7:0] key;
    wire [7:0] data_encrypted;
    wire [7:0] data_decrypted;

    // Instantiate the top module
    TOP TM (
        .data_in(data_in),
        .key(key),
        .data_encrypted(data_encrypted),
        .data_decrypted(data_decrypted)
    );

    initial begin
        // Test cases with 10 different input combinations
        $display("Starting test...");

        data_in = 8'b11001100; key = 8'b10101010; #10;
        $display("Test 1 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b11110000; key = 8'b10101010; #10;
        $display("Test 2 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b10101010; key = 8'b11111111; #10;
        $display("Test 3 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b00001111; key = 8'b10110010; #10;
        $display("Test 4 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b01010101; key = 8'b11000110; #10;
        $display("Test 5 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b00110011; key = 8'b10000001; #10;
        $display("Test 6 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b11000011; key = 8'b10111101; #10;
```



```verilog
        data_in = 8'b11000011; key = 8'b10111101; #10;
        $display("Test 7 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b01111000; key = 8'b11110011; #10;
        $display("Test 8 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b10100101; key = 8'b00011101; #10;
        $display("Test 9 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        data_in = 8'b11111111; key = 8'b00110011; #10;
        $display("Test 10 - Input: %b, Key: %b, Encrypted: %b, Decrypted: %b", data_in, key, data_encrypted, data_decrypted);

        $display("All tests completed.");
        $finish; // End the simulation
    end
endmodule
```