ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 242E
DIGITAL CIRCUITS LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	8
EXPERIMENT DATE	:	21.05.2021
LAB SESSION	:	FRIDAY - 10.30
GROUP NO	:	G17

GROUP MEMBERS:

150180024	:	ŞULE BEYZA KARADAĞ
150190710	:	SENİHA SERRA BOZKURT
150190024	:	AHMET FURKAN KAVRAZ

SPRING 2021

Contents

1 INTRODUCTION [10 points]

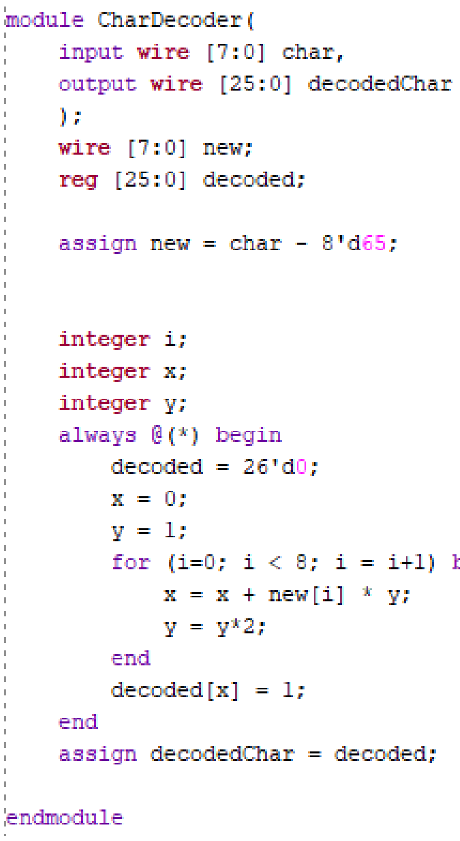
In the following experiment, we are going to implement three important cryptography applications in the history of cryptography as digital designs in Verilog. We will learn:
- How to implement a Char Decoder Module and a Char Encoder Module,
- How to implement an Circular Right Shift Module and Circular Left Shift Module,
- Implementation of Caesar Cipher,
- Implementation of the Vigenère Cipher,
- How to implement an Enigma machine.

2 MATERIALS AND METHODS [40 points]

2.1 Experiment

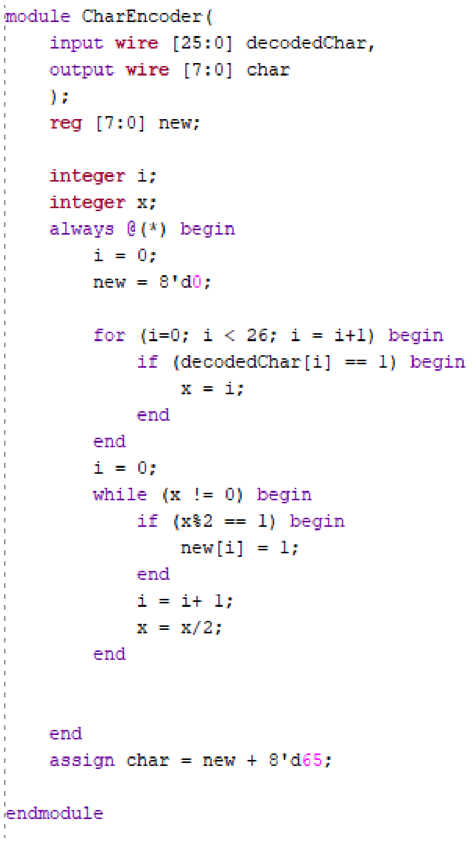
2.1.1 Part 1

In this part, we implemented the designs of 4 helper modules.



```
module CharDecoder(
    input wire [7:0] char,
    output wire [25:0] decodedChar
    );
    wire [7:0] new;
    reg [25:0] decoded;

    assign new = char - 8'd65;


    integer i;
    integer x;
    integer y;
    always @(*) begin
        decoded = 26'd0;
        x = 0;
        y = 1;
        for (i=0; i < 8; i = i+1) begin
            x = x + new[i] * y;
            y = y*2;
        end
        decoded[x] = 1;
    end
    assign decodedChar = decoded;

endmodule
```

(a) Char Decoder Module

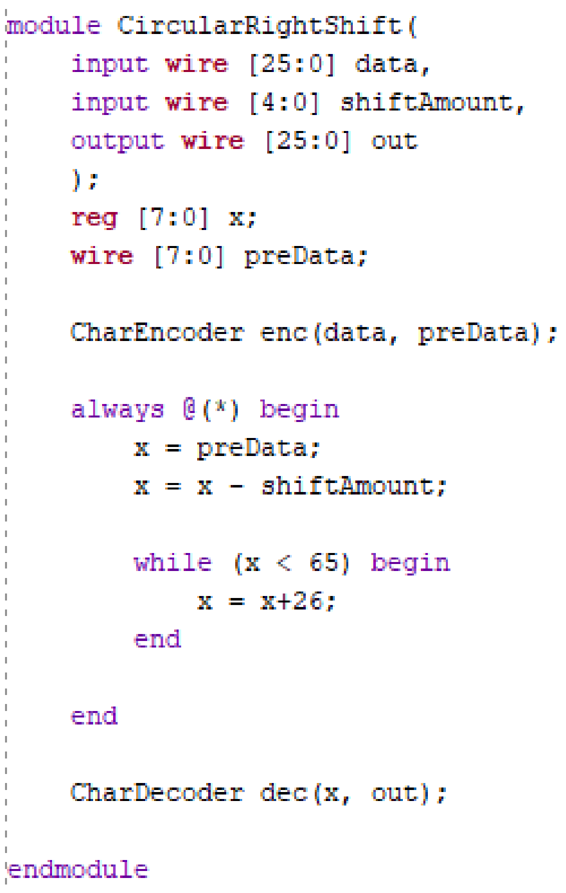


```
module CharEncoder(
    input wire [25:0] decodedChar,
    output wire [7:0] char
    );
    reg [7:0] new;

    integer i;
    integer x;
    always @(*) begin
        i = 0;
        new = 8'd0;

        for (i=0; i < 26; i = i+1) begin
            if (decodedChar[i] == 1) begin
                x = i;
            end
        end
        i = 0;
        while (x != 0) begin
            if (x%2 == 1) begin
                new[i] = 1;
            end
            i = i+ 1;
            x = x/2;
        end


    end
    assign char = new + 8'd65;

endmodule
```

(b) Char Encoder Module

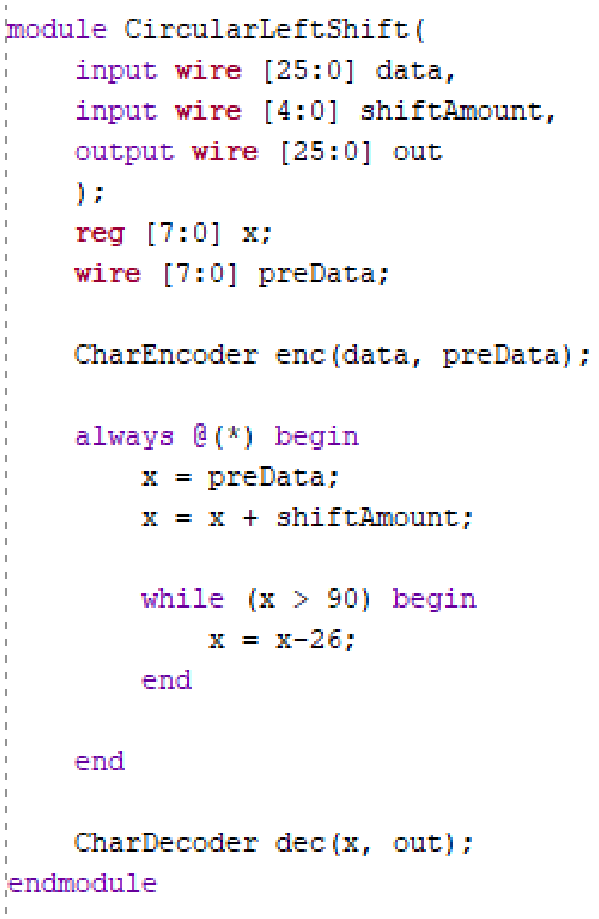
Figure 1

- There are *char* and *decodedChar* wires.(see in Figure 1) They are used as input in Figure 1.a and Figure 1.b, respectively. Also they are used as output in Figure 1.b and Figure 1.a, respectively.
- *new* is used for determining the order of the letter, for example A is the first (0^{th}) letter. In order to do this assignment, we did subtract *char* by 65 (see Figure 1.a); add 65 to *new* (see Figure 1.b) because the ASCII code of A is 065.
- In order to show which of the 26 letters the letter is, we converted *new* to *integer x* and made the x^{th} digit of the decoded, which consists of 26 digits, as 1.(see Figure 1.a)
- We found the digit 1 in *decodedChar*, which represents what letter it is, and assigned it to *integer x*. Then we convert that *integer x* to *new* so that we can do addition.(see Figure 1.b)



```
module CircularRightShift(
    input wire [25:0] data,
    input wire [4:0] shiftAmount,
    output wire [25:0] out
    );
    reg [7:0] x;
    wire [7:0] preData;

    CharEncoder enc(data, preData);

    always @(*) begin
        x = preData;
        x = x - shiftAmount;

        while (x < 65) begin
            x = x+26;
        end

    end

    CharDecoder dec(x, out);

endmodule
```

(a) Circular Right Shift Module



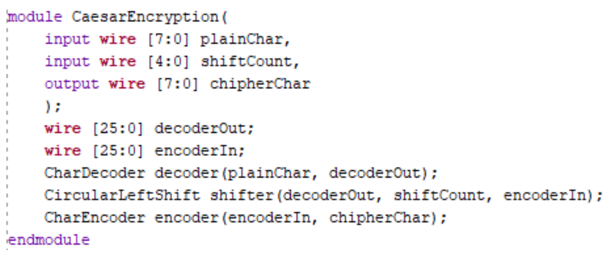
```
module CircularLeftShift(
    input wire [25:0] data,
    input wire [4:0] shiftAmount,
    output wire [25:0] out
    );
    reg [7:0] x;
    wire [7:0] preData;

    CharEncoder enc(data, preData);

    always @(*) begin
        x = preData;
        x = x + shiftAmount;

        while (x > 90) begin
            x = x-26;
        end

    end

    CharDecoder dec(x, out);
endmodule
```

(b) Circular Left Shift Module

Figure 2

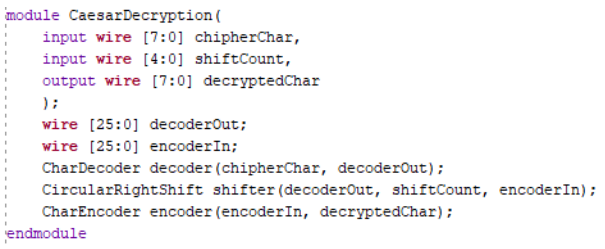
2.1.2 Part 2

In this part, we implemented Caesar Cipher Encryption and Decryption modules, and created a Caesar Module to show encrypted and decrypted messages.



```
module CaesarEncryption(
    input wire [7:0] plainChar,
    input wire [4:0] shiftCount,
    output wire [7:0] chipherChar
    );
    wire [25:0] decoderOut;
    wire [25:0] encoderIn;
    CharDecoder decoder(plainChar, decoderOut);
    CircularLeftShift shifter(decoderOut, shiftCount, encoderIn);
    CharEncoder encoder(encoderIn, chipherChar);
endmodule
```

(a) Caesar Cipher Encryption Module

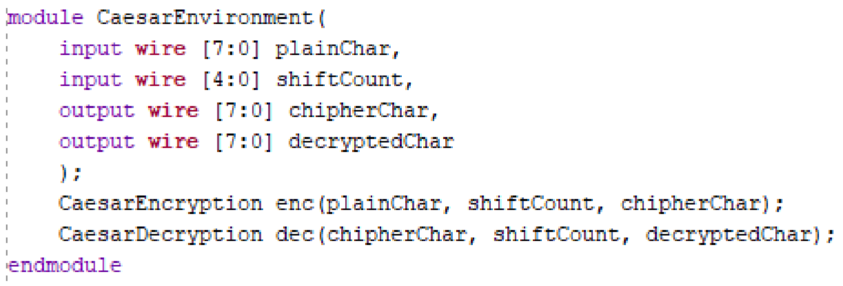


```
module CaesarDecryption(
    input wire [7:0] chipherChar,
    input wire [4:0] shiftCount,
    output wire [7:0] decryptedChar
    );
    wire [25:0] decoderOut;
    wire [25:0] encoderIn;
    CharDecoder decoder(chipherChar, decoderOut);
    CircularRightShift shifter(decoderOut, shiftCount, encoderIn);
    CharEncoder encoder(encoderIn, decryptedChar);
endmodule
```

(b) Caesar Cipher Decryption Module

Figure 3

- There are *plainChar*, *shiftCount* and *chipherChar* wires. They are used as ASCII code of the input character, shift amount of the cipher and the output character, respectively.(see Figure 3.a)
- There are *chipherChar*, *shiftCount* and *decryptedChar* wires. They are used as encrypted char input, shift amount of the cipher and the output character, respectively.(see Figure 3.b)
- In both encryption and decryption, first the inputs are decoded using the decoder module, after shifting operation, they are re-encoded using the encoder module.



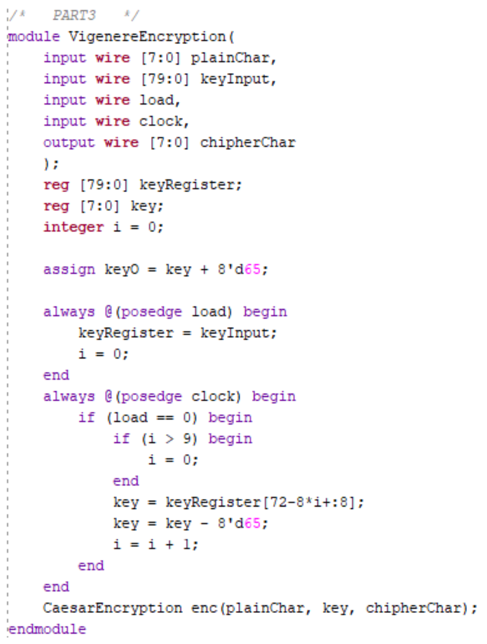
```
module CaesarEnvironment(
    input wire [7:0] plainChar,
    input wire [4:0] shiftCount,
    output wire [7:0] chipherChar,
    output wire [7:0] decryptedChar
    );
    CaesarEncryption enc(plainChar, shiftCount, chipherChar);
    CaesarDecryption dec(chipherChar, shiftCount, decryptedChar);
endmodule
```

Figure 4: Caesar Environment Module

- There are *plainChar*, *shiftCount*, *chipherChar* and *decryptedChar* wires. They are used as ASCII code of the input character, shift amount of the cipher, the output character which is encrypted using Caesar Cipher with *shiftCount* and the output character which is decrypted using Caesar Cipher with *shiftCount*, respectively.
- Caesar Cipher Encryption Module and Caesar Cipher Decryption Module used in order, *chipherChar* output of the encryption is given as input for decryption.(see Figure 4)

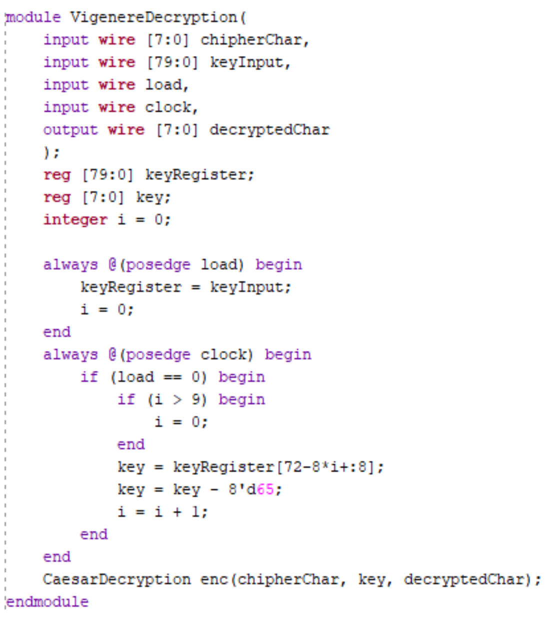
2.1.3 Part 3

In this part, we implemented the Vigenère Cipher Encryption and Decryption modules and a Vigenèere Module to show encrypted and decrypted messages.



```
/*   PART3   */
module VigenereEncryption(
    input wire [7:0] plainChar,
    input wire [79:0] keyInput,
    input wire load,
    input wire clock,
    output wire [7:0] chipherChar
    );
    reg [79:0] keyRegister;
    reg [7:0] key;
    integer i = 0;

    assign key0 = key + 8'd65;

    always @(posedge load) begin
        keyRegister = keyInput;
        i = 0;
    end
    always @(posedge clock) begin
        if (load == 0) begin
            if (i > 9) begin
                i = 0;
            end
            key = keyRegister[72-8*i+:8];
            key = key - 8'd65;
            i = i + 1;
        end
    end
    CaesarEncryption enc(plainChar, key, chipherChar);
endmodule
```

(a) Vigenère Cipher Encryption Module

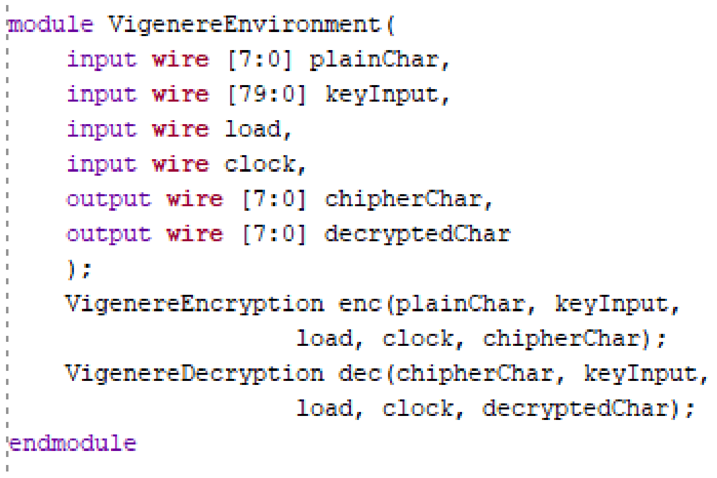


```
module VigenereDecryption(
    input wire [7:0] chipherChar,
    input wire [79:0] keyInput,
    input wire load,
    input wire clock,
    output wire [7:0] decryptedChar
    );
    reg [79:0] keyRegister;
    reg [7:0] key;
    integer i = 0;

    always @(posedge load) begin
        keyRegister = keyInput;
        i = 0;
    end
    always @(posedge clock) begin
        if (load == 0) begin
            if (i > 9) begin
                i = 0;
            end
            key = keyRegister[72-8*i+:8];
            key = key - 8'd65;
            i = i + 1;
        end
    end
    CaesarDecryption enc(chipherChar, key, decryptedChar);
endmodule
```

(b) Vigenère Cipher Decryption Module

Figure 5

- There are *plainChar*, *keyInput* and *chipherChar* wires. They are used as ASCII code of the input character, key message of the cipher and the output character, respectively.(see Figure 5.a)
- There are *chipherChar*, *keyInput* and *decryptedChar* wires. They are used as Encrypted char input, key message of the cipher and the output character, respectively.(see Figure 5.b)
- *load* is used for representing the loading of *keyInput* to *key register*, *clock* is used for determining the shifting time of *key register*.(see Figure 5)
- In both encryption and decryption, *keyInput* is loaded to *key register* at the rising edge of the load signal and *key register* is shifted only one character at the rising edge of the clock signal when load = 0.(see Figure 5)



```
module VigenereEnvironment(
    input wire [7:0] plainChar,
    input wire [79:0] keyInput,
    input wire load,
    input wire clock,
    output wire [7:0] chipherChar,
    output wire [7:0] decryptedChar
    );
    VigenereEncryption enc(plainChar, keyInput,
                    load, clock, chipherChar);
    VigenereDecryption dec(chipherChar, keyInput,
                    load, clock, decryptedChar);
endmodule
```

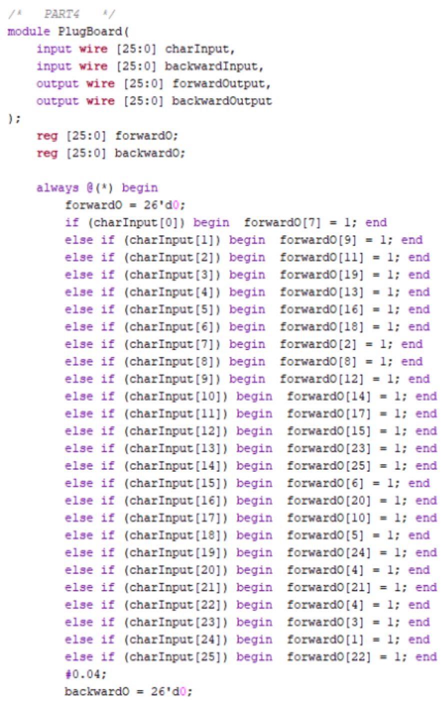
Figure 6: Vigenère Environment Module

- There are *plainChar*, *keyInput*, *chipherChar* and *decryptedChar* wires. They are used as ASCII code of the input character, key message of the cipher, the output character which is encrypted and the output character which is decrypted, respectively.

- Vigenère Cipher Encryption Module and Vigenère Cipher Decryption Module used in order, *chipherChar* output of the encryption is given as input for decryption.(see Figure 6)

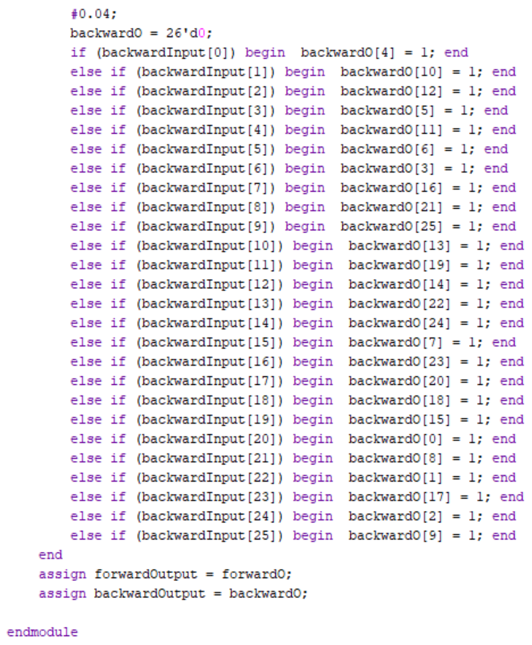
2.1.4 Part 4

In this part, we tried to implement the Enigma machine.



```
/*   PART4   */
module PlugBoard(
    input wire [25:0] charInput,
    input wire [25:0] backwardInput,
    output wire [25:0] forwardOutput,
    output wire [25:0] backwardOutput
);
    reg [25:0] forward0;
    reg [25:0] backward0;

    always @(*) begin
        forward0 = 26'd0;
        if (charInput[0]) begin  forward0[7] = 1; end
        else if (charInput[1]) begin  forward0[9] = 1; end
        else if (charInput[2]) begin  forward0[11] = 1; end
        else if (charInput[3]) begin  forward0[19] = 1; end
        else if (charInput[4]) begin  forward0[13] = 1; end
        else if (charInput[5]) begin  forward0[16] = 1; end
        else if (charInput[6]) begin  forward0[18] = 1; end
        else if (charInput[7]) begin  forward0[2] = 1; end
        else if (charInput[8]) begin  forward0[8] = 1; end
        else if (charInput[9]) begin  forward0[12] = 1; end
        else if (charInput[10]) begin  forward0[14] = 1; end
        else if (charInput[11]) begin  forward0[17] = 1; end
        else if (charInput[12]) begin  forward0[15] = 1; end
        else if (charInput[13]) begin  forward0[23] = 1; end
        else if (charInput[14]) begin  forward0[25] = 1; end
        else if (charInput[15]) begin  forward0[6] = 1; end
        else if (charInput[16]) begin  forward0[20] = 1; end
        else if (charInput[17]) begin  forward0[10] = 1; end
        else if (charInput[18]) begin  forward0[5] = 1; end
        else if (charInput[19]) begin  forward0[24] = 1; end
        else if (charInput[20]) begin  forward0[4] = 1; end
        else if (charInput[21]) begin  forward0[21] = 1; end
        else if (charInput[22]) begin  forward0[4] = 1; end
        else if (charInput[23]) begin  forward0[3] = 1; end
        else if (charInput[24]) begin  forward0[1] = 1; end
        else if (charInput[25]) begin  forward0[22] = 1; end
        #0.04;
        backward0 = 26'd0;
```

(a)

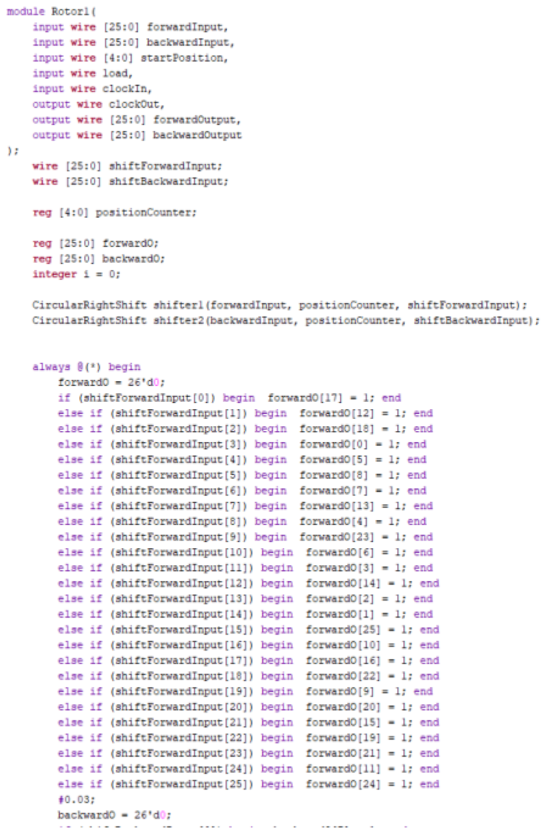


```
        #0.04;
        backward0 = 26'd0;
        if (backwardInput[0]) begin  backward0[4] = 1; end
        else if (backwardInput[1]) begin  backward0[10] = 1; end
        else if (backwardInput[2]) begin  backward0[12] = 1; end
        else if (backwardInput[3]) begin  backward0[5] = 1; end
        else if (backwardInput[4]) begin  backward0[11] = 1; end
        else if (backwardInput[5]) begin  backward0[6] = 1; end
        else if (backwardInput[6]) begin  backward0[3] = 1; end
        else if (backwardInput[7]) begin  backward0[16] = 1; end
        else if (backwardInput[8]) begin  backward0[21] = 1; end
        else if (backwardInput[9]) begin  backward0[25] = 1; end
        else if (backwardInput[10]) begin  backward0[13] = 1; end
        else if (backwardInput[11]) begin  backward0[19] = 1; end
        else if (backwardInput[12]) begin  backward0[14] = 1; end
        else if (backwardInput[13]) begin  backward0[22] = 1; end
        else if (backwardInput[14]) begin  backward0[24] = 1; end
        else if (backwardInput[15]) begin  backward0[7] = 1; end
        else if (backwardInput[16]) begin  backward0[23] = 1; end
        else if (backwardInput[17]) begin  backward0[20] = 1; end
        else if (backwardInput[18]) begin  backward0[18] = 1; end
        else if (backwardInput[19]) begin  backward0[15] = 1; end
        else if (backwardInput[20]) begin  backward0[0] = 1; end
        else if (backwardInput[21]) begin  backward0[8] = 1; end
        else if (backwardInput[22]) begin  backward0[1] = 1; end
        else if (backwardInput[23]) begin  backward0[17] = 1; end
        else if (backwardInput[24]) begin  backward0[2] = 1; end
        else if (backwardInput[25]) begin  backward0[9] = 1; end
    end
    assign forwardOutput = forward0;
    assign backwardOutput = backward0;

endmodule
```

(b)

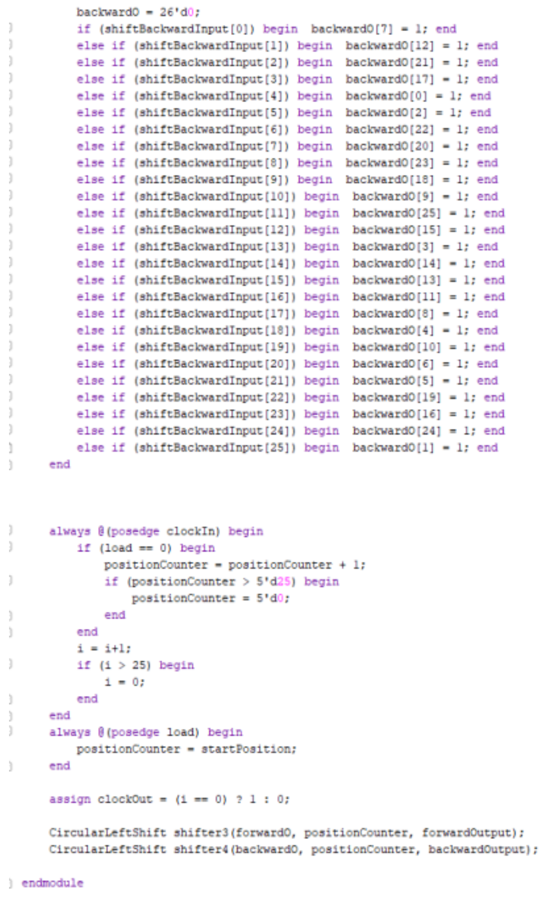
Figure 7: Plugboard Module

- There are *charInput*, *backwardInput*, *forwardOutput* and *backwardOutput* wires. They are used as forward direction input of the character, backward direction input of the character, forward direction output of the character and backward direction output of the character, respectively.
- In the forward direction, 1 assignment to the *forward0* was made according to the *charInput*'s equivalents in Table 2: Plugboard Bit Connections in experiment pdf.
- In the backward direction, 1 assignment to the *backward0* was made according to the *backwardInput*'s equivalents in Table 2: Plugboard Bit Connections in experiment pdf.
- At the end, *forward0* and *backward0* are assigned to *forwardOutput* and *backward-Output*, respectively.



```verilog
module Rotor1(
    input wire [25:0] forwardInput,
    input wire [25:0] backwardInput,
    input wire [4:0] startPosition,
    input wire load,
    input wire clockIn,
    output wire clockOut,
    output wire [25:0] forwardOutput,
    output wire [25:0] backwardOutput
);
    wire [25:0] shiftForwardInput;
    wire [25:0] shiftBackwardInput;

    reg [4:0] positionCounter;

    reg [25:0] forward0;
    reg [25:0] backward0;
    integer i = 0;

    CircularRightShift shifter1(forwardInput, positionCounter, shiftForwardInput);
    CircularRightShift shifter2(backwardInput, positionCounter, shiftBackwardInput);


    always @(*) begin
        forward0 = 26'd0;
        if (shiftForwardInput[0]) begin  forward0[17] = 1; end
        else if (shiftForwardInput[1]) begin  forward0[12] = 1; end
        else if (shiftForwardInput[2]) begin  forward0[18] = 1; end
        else if (shiftForwardInput[3]) begin  forward0[0] = 1; end
        else if (shiftForwardInput[4]) begin  forward0[5] = 1; end
        else if (shiftForwardInput[5]) begin  forward0[8] = 1; end
        else if (shiftForwardInput[6]) begin  forward0[7] = 1; end
        else if (shiftForwardInput[7]) begin  forward0[13] = 1; end
        else if (shiftForwardInput[8]) begin  forward0[4] = 1; end
        else if (shiftForwardInput[9]) begin  forward0[23] = 1; end
        else if (shiftForwardInput[10]) begin  forward0[6] = 1; end
        else if (shiftForwardInput[11]) begin  forward0[3] = 1; end
        else if (shiftForwardInput[12]) begin  forward0[14] = 1; end
        else if (shiftForwardInput[13]) begin  forward0[2] = 1; end
        else if (shiftForwardInput[14]) begin  forward0[1] = 1; end
        else if (shiftForwardInput[15]) begin  forward0[25] = 1; end
        else if (shiftForwardInput[16]) begin  forward0[10] = 1; end
        else if (shiftForwardInput[17]) begin  forward0[16] = 1; end
        else if (shiftForwardInput[18]) begin  forward0[22] = 1; end
        else if (shiftForwardInput[19]) begin  forward0[9] = 1; end
        else if (shiftForwardInput[20]) begin  forward0[20] = 1; end
        else if (shiftForwardInput[21]) begin  forward0[15] = 1; end
        else if (shiftForwardInput[22]) begin  forward0[19] = 1; end
        else if (shiftForwardInput[23]) begin  forward0[21] = 1; end
        else if (shiftForwardInput[24]) begin  forward0[11] = 1; end
        else if (shiftForwardInput[25]) begin  forward0[24] = 1; end
        #0.03;
        backward0 = 26'd0;
```

(a)

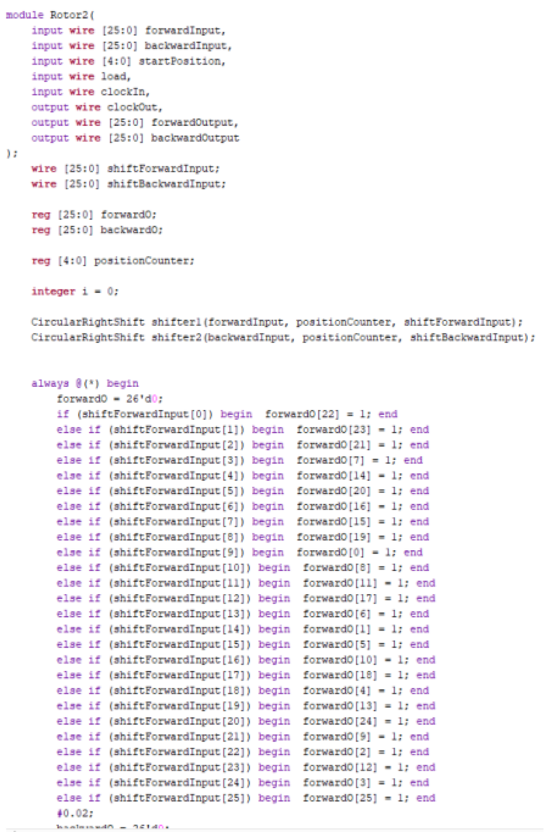


```verilog
        backward0 = 26'd0;
        if (shiftBackwardInput[0]) begin  backward0[7] = 1; end
        else if (shiftBackwardInput[1]) begin  backward0[12] = 1; end
        else if (shiftBackwardInput[2]) begin  backward0[21] = 1; end
        else if (shiftBackwardInput[3]) begin  backward0[17] = 1; end
        else if (shiftBackwardInput[4]) begin  backward0[0] = 1; end
        else if (shiftBackwardInput[5]) begin  backward0[2] = 1; end
        else if (shiftBackwardInput[6]) begin  backward0[22] = 1; end
        else if (shiftBackwardInput[7]) begin  backward0[20] = 1; end
        else if (shiftBackwardInput[8]) begin  backward0[23] = 1; end
        else if (shiftBackwardInput[9]) begin  backward0[18] = 1; end
        else if (shiftBackwardInput[10]) begin  backward0[9] = 1; end
        else if (shiftBackwardInput[11]) begin  backward0[25] = 1; end
        else if (shiftBackwardInput[12]) begin  backward0[15] = 1; end
        else if (shiftBackwardInput[13]) begin  backward0[3] = 1; end
        else if (shiftBackwardInput[14]) begin  backward0[14] = 1; end
        else if (shiftBackwardInput[15]) begin  backward0[13] = 1; end
        else if (shiftBackwardInput[16]) begin  backward0[11] = 1; end
        else if (shiftBackwardInput[17]) begin  backward0[8] = 1; end
        else if (shiftBackwardInput[18]) begin  backward0[4] = 1; end
        else if (shiftBackwardInput[19]) begin  backward0[10] = 1; end
        else if (shiftBackwardInput[20]) begin  backward0[6] = 1; end
        else if (shiftBackwardInput[21]) begin  backward0[5] = 1; end
        else if (shiftBackwardInput[22]) begin  backward0[19] = 1; end
        else if (shiftBackwardInput[23]) begin  backward0[16] = 1; end
        else if (shiftBackwardInput[24]) begin  backward0[24] = 1; end
        else if (shiftBackwardInput[25]) begin  backward0[1] = 1; end
    end



    always @(posedge clockIn) begin
        if (load == 0) begin
            positionCounter = positionCounter + 1;
            if (positionCounter > 5'd25) begin
                positionCounter = 5'd0;
            end
        end
        i = i+1;
        if (i > 25) begin
            i = 0;
        end
    end
    always @(posedge load) begin
        positionCounter = startPosition;
    end

    assign clockOut = (i == 0) ? 1 : 0;

    CircularLeftShift shifter3(forward0, positionCounter, forwardOutput);
    CircularLeftShift shifter4(backward0, positionCounter, backwardOutput);

endmodule
```

(b)

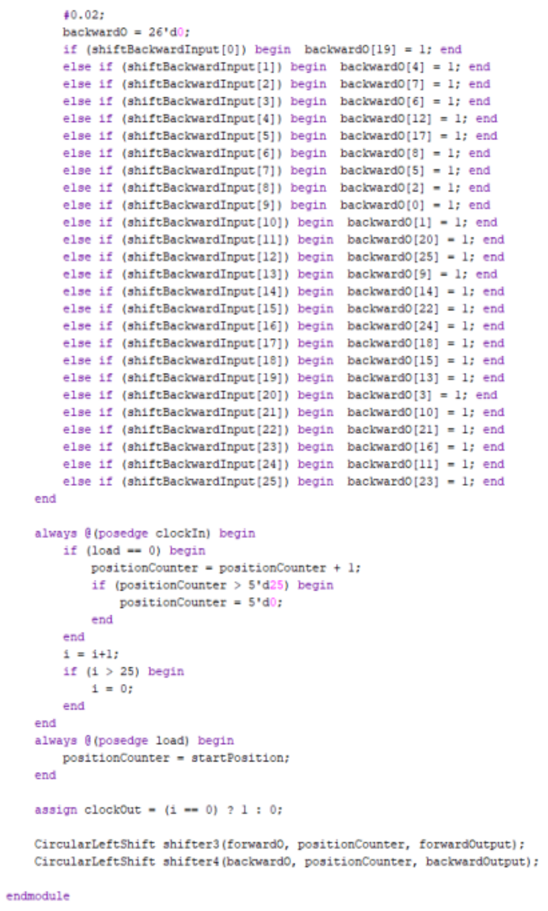
Figure 8: Rotor1 Module

- There are *forwardInput*, *backwardInput*, *startPosition*, *forwardOutput* and *backwardOutput* wires. They are used as forward direction input of the character, backward direction input of the character, the starting position of the rotor, forward direction output of the character and backward direction output of the character, respectively.
- Right shifting from *forwardInput* to *shiftForwardInput*, *backwardInput* to *shiftBackwardInput*, with respect to *positionCounter* has been done.
- In the forward direction, 1 assignment to the *forward0* was made according to the *shiftForwardInput*'s equivalents in Table 3: Rotor1 Bit Connections in experiment pdf.
- In the backward direction, 1 assignment to the *backward0* was made according to the *shiftBackwardInput*'s equivalents in Table 3: Rotor1 Bit Connections in experiment pdf.
- At the end, left shifting from *forward0* to *forwardOnput*, *backward0* to *backwardOutput*, with respect to *positionCounter* has been done.



```verilog
module Rotor2(
    input wire [25:0] forwardInput,
    input wire [25:0] backwardInput,
    input wire [4:0] startPosition,
    input wire load,
    input wire clockIn,
    output wire clockOut,
    output wire [25:0] forwardOutput,
    output wire [25:0] backwardOutput
);
    wire [25:0] shiftForwardInput;
    wire [25:0] shiftBackwardInput;

    reg [25:0] forward0;
    reg [25:0] backward0;

    reg [4:0] positionCounter;

    integer i = 0;

    CircularRightShift shifter1(forwardInput, positionCounter, shiftForwardInput);
    CircularRightShift shifter2(backwardInput, positionCounter, shiftBackwardInput);


    always @(*) begin
        forward0 = 26'd0;
        if (shiftForwardInput[0]) begin  forward0[22] = 1; end
        else if (shiftForwardInput[1]) begin  forward0[23] = 1; end
        else if (shiftForwardInput[2]) begin  forward0[21] = 1; end
        else if (shiftForwardInput[3]) begin  forward0[7] = 1; end
        else if (shiftForwardInput[4]) begin  forward0[14] = 1; end
        else if (shiftForwardInput[5]) begin  forward0[20] = 1; end
        else if (shiftForwardInput[6]) begin  forward0[16] = 1; end
        else if (shiftForwardInput[7]) begin  forward0[15] = 1; end
        else if (shiftForwardInput[8]) begin  forward0[19] = 1; end
        else if (shiftForwardInput[9]) begin  forward0[0] = 1; end
        else if (shiftForwardInput[10]) begin  forward0[8] = 1; end
        else if (shiftForwardInput[11]) begin  forward0[11] = 1; end
        else if (shiftForwardInput[12]) begin  forward0[17] = 1; end
        else if (shiftForwardInput[13]) begin  forward0[6] = 1; end
        else if (shiftForwardInput[14]) begin  forward0[1] = 1; end
        else if (shiftForwardInput[15]) begin  forward0[5] = 1; end
        else if (shiftForwardInput[16]) begin  forward0[10] = 1; end
        else if (shiftForwardInput[17]) begin  forward0[18] = 1; end
        else if (shiftForwardInput[18]) begin  forward0[4] = 1; end
        else if (shiftForwardInput[19]) begin  forward0[13] = 1; end
        else if (shiftForwardInput[20]) begin  forward0[24] = 1; end
        else if (shiftForwardInput[21]) begin  forward0[9] = 1; end
        else if (shiftForwardInput[22]) begin  forward0[2] = 1; end
        else if (shiftForwardInput[23]) begin  forward0[12] = 1; end
        else if (shiftForwardInput[24]) begin  forward0[3] = 1; end
        else if (shiftForwardInput[25]) begin  forward0[25] = 1; end
        #0.02;
```

(a)

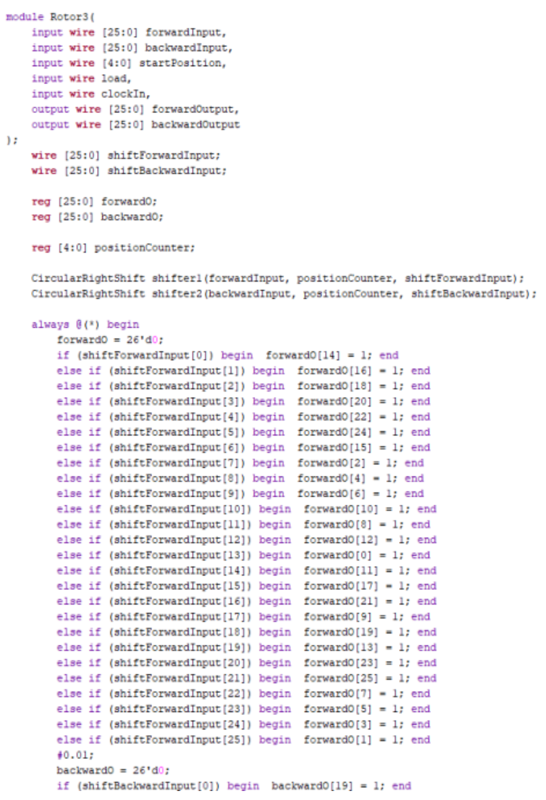


```verilog
        #0.02;
        backward0 = 26'd0;
        if (shiftBackwardInput[0]) begin  backward0[19] = 1; end
        else if (shiftBackwardInput[1]) begin  backward0[4] = 1; end
        else if (shiftBackwardInput[2]) begin  backward0[7] = 1; end
        else if (shiftBackwardInput[3]) begin  backward0[6] = 1; end
        else if (shiftBackwardInput[4]) begin  backward0[12] = 1; end
        else if (shiftBackwardInput[5]) begin  backward0[17] = 1; end
        else if (shiftBackwardInput[6]) begin  backward0[8] = 1; end
        else if (shiftBackwardInput[7]) begin  backward0[5] = 1; end
        else if (shiftBackwardInput[8]) begin  backward0[2] = 1; end
        else if (shiftBackwardInput[9]) begin  backward0[0] = 1; end
        else if (shiftBackwardInput[10]) begin  backward0[1] = 1; end
        else if (shiftBackwardInput[11]) begin  backward0[20] = 1; end
        else if (shiftBackwardInput[12]) begin  backward0[25] = 1; end
        else if (shiftBackwardInput[13]) begin  backward0[9] = 1; end
        else if (shiftBackwardInput[14]) begin  backward0[14] = 1; end
        else if (shiftBackwardInput[15]) begin  backward0[22] = 1; end
        else if (shiftBackwardInput[16]) begin  backward0[24] = 1; end
        else if (shiftBackwardInput[17]) begin  backward0[18] = 1; end
        else if (shiftBackwardInput[18]) begin  backward0[15] = 1; end
        else if (shiftBackwardInput[19]) begin  backward0[13] = 1; end
        else if (shiftBackwardInput[20]) begin  backward0[3] = 1; end
        else if (shiftBackwardInput[21]) begin  backward0[10] = 1; end
        else if (shiftBackwardInput[22]) begin  backward0[21] = 1; end
        else if (shiftBackwardInput[23]) begin  backward0[16] = 1; end
        else if (shiftBackwardInput[24]) begin  backward0[11] = 1; end
        else if (shiftBackwardInput[25]) begin  backward0[23] = 1; end
    end

    always @(posedge clockIn) begin
        if (load == 0) begin
            positionCounter = positionCounter + 1;
            if (positionCounter > 5'd25) begin
                positionCounter = 5'd0;
            end
        end
        i = i+1;
        if (i > 25) begin
            i = 0;
        end
    end
    always @(posedge load) begin
        positionCounter = startPosition;
    end

    assign clockOut = (i == 0) ? 1 : 0;

    CircularLeftShift shifter3(forward0, positionCounter, forwardOutput);
    CircularLeftShift shifter4(backward0, positionCounter, backwardOutput);

endmodule
```

(b)

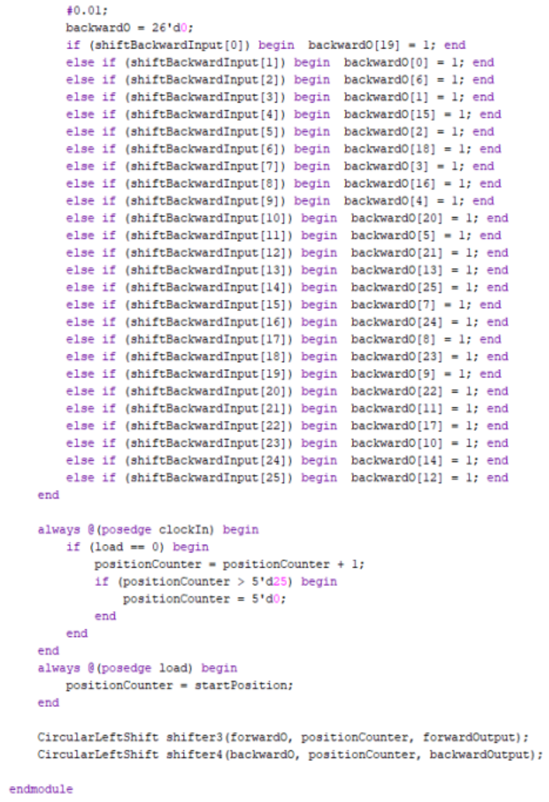
Figure 9: Rotor2 Module

- There are *forwardInput*, *backwardInput*, *startPosition*, *forwardOutput* and *backwardOutput* wires. They are used as forward direction input of the character, backward direction input of the character, the starting position of the rotor, forward direction output of the character and backward direction output of the character, respectively.
- Right shifting from *forwardInput* to *shiftForwardInput*, *backwardInput* to *shiftBackwardInput*, with respect to *positionCounter* has been done.
- In the forward direction, 1 assignment to the *forward0* was made according to the *shiftForwardInput*'s equivalents in Table 4: Rotor2 Bit Connections in experiment pdf.
- In the backward direction, 1 assignment to the *backward0* was made according to the *shiftBackwardInput*'s equivalents in Table 4: Rotor2 Bit Connections in experiment pdf.
- At the end, left shifting from *forward0* to *forwardOnput*, *backward0* to *backwardOutput*, with respect to *positionCounter* has been done.



```
module Rotor3(
    input wire [25:0] forwardInput,
    input wire [25:0] backwardInput,
    input wire [4:0] startPosition,
    input wire load,
    input wire clockIn,
    output wire [25:0] forwardOutput,
    output wire [25:0] backwardOutput
);
    wire [25:0] shiftForwardInput;
    wire [25:0] shiftBackwardInput;

    reg [25:0] forward0;
    reg [25:0] backward0;

    reg [4:0] positionCounter;

    CircularRightShift shifter1(forwardInput, positionCounter, shiftForwardInput);
    CircularRightShift shifter2(backwardInput, positionCounter, shiftBackwardInput);

    always @(*) begin
        forward0 = 26'd0;
        if (shiftForwardInput[0]) begin  forward0[14] = 1; end
        else if (shiftForwardInput[1]) begin  forward0[16] = 1; end
        else if (shiftForwardInput[2]) begin  forward0[18] = 1; end
        else if (shiftForwardInput[3]) begin  forward0[20] = 1; end
        else if (shiftForwardInput[4]) begin  forward0[22] = 1; end
        else if (shiftForwardInput[5]) begin  forward0[24] = 1; end
        else if (shiftForwardInput[6]) begin  forward0[15] = 1; end
        else if (shiftForwardInput[7]) begin  forward0[2] = 1; end
        else if (shiftForwardInput[8]) begin  forward0[4] = 1; end
        else if (shiftForwardInput[9]) begin  forward0[6] = 1; end
        else if (shiftForwardInput[10]) begin  forward0[10] = 1; end
        else if (shiftForwardInput[11]) begin  forward0[8] = 1; end
        else if (shiftForwardInput[12]) begin  forward0[12] = 1; end
        else if (shiftForwardInput[13]) begin  forward0[0] = 1; end
        else if (shiftForwardInput[14]) begin  forward0[11] = 1; end
        else if (shiftForwardInput[15]) begin  forward0[17] = 1; end
        else if (shiftForwardInput[16]) begin  forward0[21] = 1; end
        else if (shiftForwardInput[17]) begin  forward0[9] = 1; end
        else if (shiftForwardInput[18]) begin  forward0[19] = 1; end
        else if (shiftForwardInput[19]) begin  forward0[13] = 1; end
        else if (shiftForwardInput[20]) begin  forward0[23] = 1; end
        else if (shiftForwardInput[21]) begin  forward0[25] = 1; end
        else if (shiftForwardInput[22]) begin  forward0[7] = 1; end
        else if (shiftForwardInput[23]) begin  forward0[5] = 1; end
        else if (shiftForwardInput[24]) begin  forward0[3] = 1; end
        else if (shiftForwardInput[25]) begin  forward0[1] = 1; end
        #0.01;
        backward0 = 26'd0;
        if (shiftBackwardInput[0]) begin  backward0[19] = 1; end
```

(a)



```
        #0.01;
        backward0 = 26'd0;
        if (shiftBackwardInput[0]) begin  backward0[19] = 1; end
        else if (shiftBackwardInput[1]) begin  backward0[0] = 1; end
        else if (shiftBackwardInput[2]) begin  backward0[6] = 1; end
        else if (shiftBackwardInput[3]) begin  backward0[1] = 1; end
        else if (shiftBackwardInput[4]) begin  backward0[15] = 1; end
        else if (shiftBackwardInput[5]) begin  backward0[2] = 1; end
        else if (shiftBackwardInput[6]) begin  backward0[18] = 1; end
        else if (shiftBackwardInput[7]) begin  backward0[3] = 1; end
        else if (shiftBackwardInput[8]) begin  backward0[16] = 1; end
        else if (shiftBackwardInput[9]) begin  backward0[4] = 1; end
        else if (shiftBackwardInput[10]) begin  backward0[20] = 1; end
        else if (shiftBackwardInput[11]) begin  backward0[5] = 1; end
        else if (shiftBackwardInput[12]) begin  backward0[21] = 1; end
        else if (shiftBackwardInput[13]) begin  backward0[13] = 1; end
        else if (shiftBackwardInput[14]) begin  backward0[25] = 1; end
        else if (shiftBackwardInput[15]) begin  backward0[7] = 1; end
        else if (shiftBackwardInput[16]) begin  backward0[24] = 1; end
        else if (shiftBackwardInput[17]) begin  backward0[8] = 1; end
        else if (shiftBackwardInput[18]) begin  backward0[23] = 1; end
        else if (shiftBackwardInput[19]) begin  backward0[9] = 1; end
        else if (shiftBackwardInput[20]) begin  backward0[22] = 1; end
        else if (shiftBackwardInput[21]) begin  backward0[11] = 1; end
        else if (shiftBackwardInput[22]) begin  backward0[17] = 1; end
        else if (shiftBackwardInput[23]) begin  backward0[10] = 1; end
        else if (shiftBackwardInput[24]) begin  backward0[14] = 1; end
        else if (shiftBackwardInput[25]) begin  backward0[12] = 1; end
    end

    always @(posedge clockIn) begin
        if (load == 0) begin
            positionCounter = positionCounter + 1;
            if (positionCounter > 5'd25) begin
                positionCounter = 5'd0;
            end
        end
    end
    always @(posedge load) begin
        positionCounter = startPosition;
    end

    CircularLeftShift shifter3(forward0, positionCounter, forwardOutput);
    CircularLeftShift shifter4(backward0, positionCounter, backwardOutput);

endmodule
```

(b)

Figure 10: Rotor3 Module

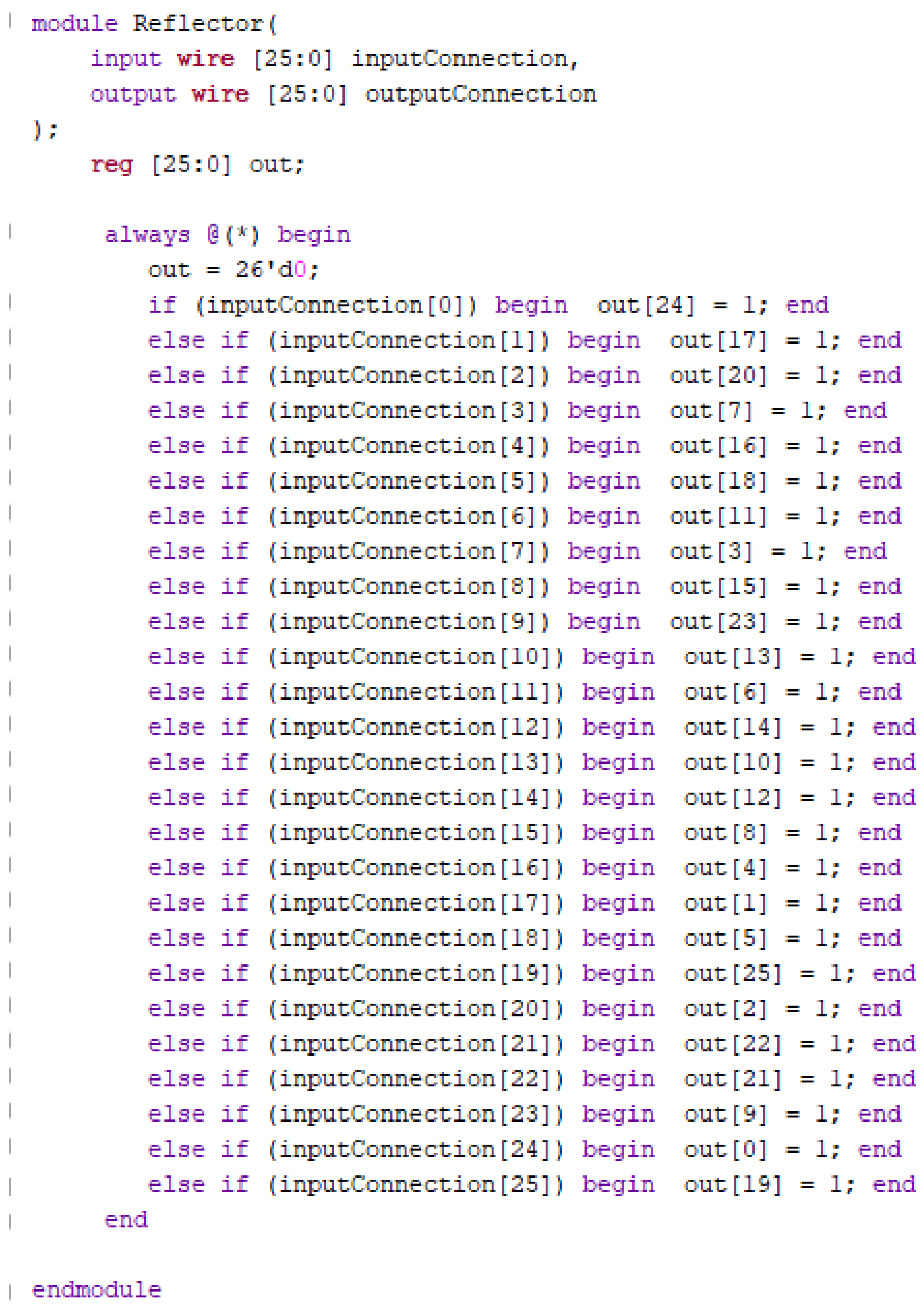
- There are *forwardInput*, *backwardInput*, *startPosition*, *forwardOutput* and *backwardOutput* wires. They are used as forward direction input of the character, backward direction input of the character, the starting position of the rotor, forward direction output of the character and backward direction output of the character, respectively.

- Right shifting from *forwardInput* to *shiftForwardInput*, *backwardInput* to *shiftBackwardInput*, with respect to *positionCounter* has been done.

- In the forward direction, 1 assignment to the *forward0* was made according to the *shiftForwardInput*'s equivalents in Table 5: Rotor3 Bit Connections in experiment pdf.

- In the backward direction, 1 assignment to the *backward0* was made according to the *shiftBackwardInput*'s equivalents in Table 5: Rotor3 Bit Connections in experiment pdf.

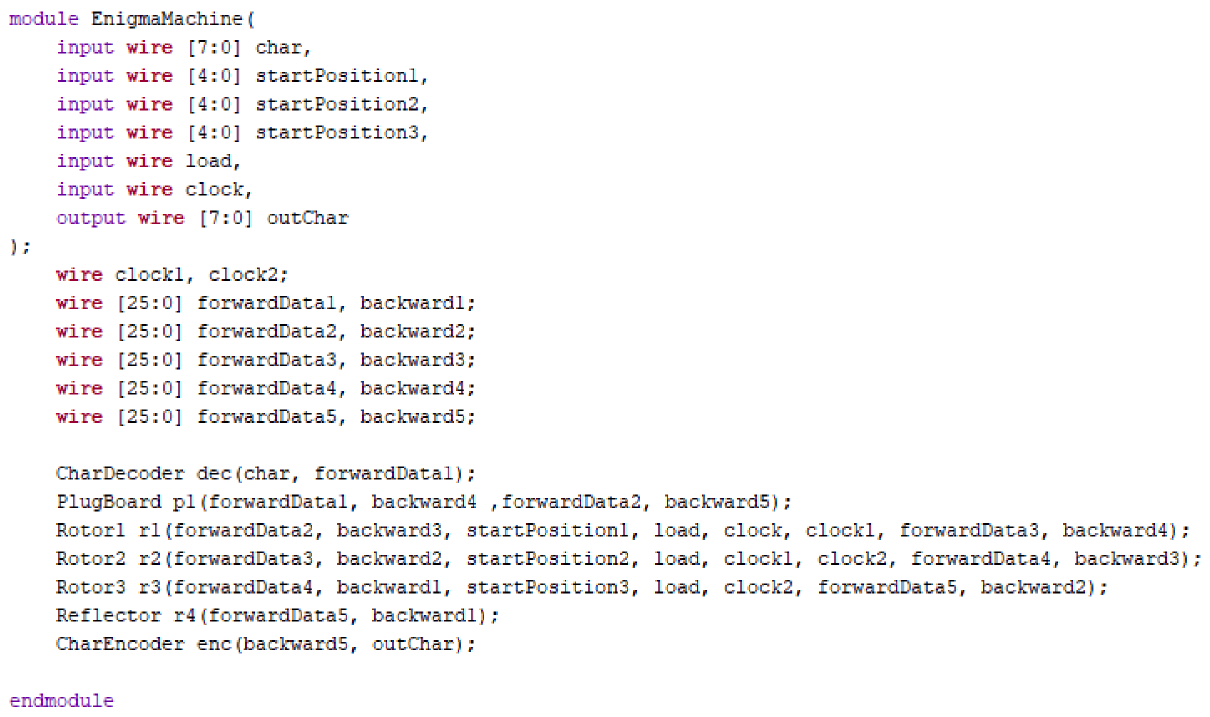
- At the end, left shifting from *forward0* to *forwardOnput*, *backward0* to *backwardOutput*, with respect to *positionCounter* has been done.



```
module Reflector(
    input wire [25:0] inputConnection,
    output wire [25:0] outputConnection
);
    reg [25:0] out;

     always @(*) begin
        out = 26'd0;
        if (inputConnection[0]) begin  out[24] = 1; end
        else if (inputConnection[1]) begin  out[17] = 1; end
        else if (inputConnection[2]) begin  out[20] = 1; end
        else if (inputConnection[3]) begin  out[7] = 1; end
        else if (inputConnection[4]) begin  out[16] = 1; end
        else if (inputConnection[5]) begin  out[18] = 1; end
        else if (inputConnection[6]) begin  out[11] = 1; end
        else if (inputConnection[7]) begin  out[3] = 1; end
        else if (inputConnection[8]) begin  out[15] = 1; end
        else if (inputConnection[9]) begin  out[23] = 1; end
        else if (inputConnection[10]) begin  out[13] = 1; end
        else if (inputConnection[11]) begin  out[6] = 1; end
        else if (inputConnection[12]) begin  out[14] = 1; end
        else if (inputConnection[13]) begin  out[10] = 1; end
        else if (inputConnection[14]) begin  out[12] = 1; end
        else if (inputConnection[15]) begin  out[8] = 1; end
        else if (inputConnection[16]) begin  out[4] = 1; end
        else if (inputConnection[17]) begin  out[1] = 1; end
        else if (inputConnection[18]) begin  out[5] = 1; end
        else if (inputConnection[19]) begin  out[25] = 1; end
        else if (inputConnection[20]) begin  out[2] = 1; end
        else if (inputConnection[21]) begin  out[22] = 1; end
        else if (inputConnection[22]) begin  out[21] = 1; end
        else if (inputConnection[23]) begin  out[9] = 1; end
        else if (inputConnection[24]) begin  out[0] = 1; end
        else if (inputConnection[25]) begin  out[19] = 1; end
     end

endmodule
```

Figure 11: Reflector Module

- There are *inputConnection* and *outputConnection* wires. They are used as forward direction input of the character and backward direction output of the character, respectively.
- Reflector module is basicly used for turning the forward direction of the character to the backward direction. (see Figure 11)
- 1 assignment to the *out* was made according to the *inputConnection*'s equivalents in Table 6: Reflector Bit Connections in experiment pdf.



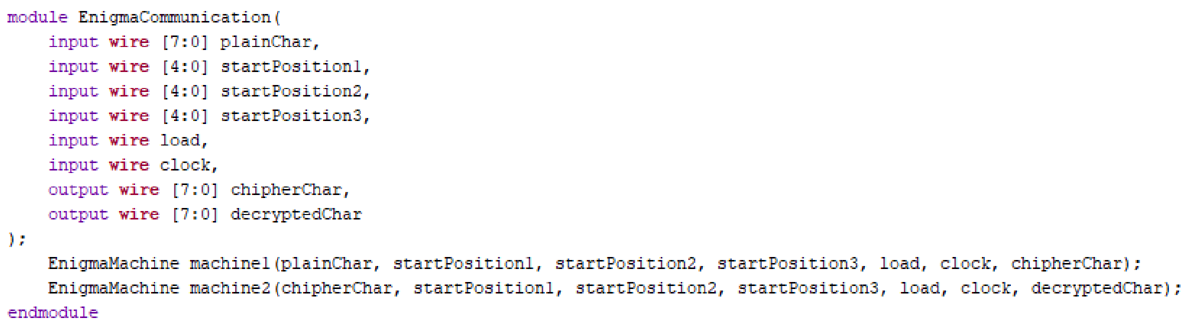
```
module EnigmaMachine(
    input wire [7:0] char,
    input wire [4:0] startPosition1,
    input wire [4:0] startPosition2,
    input wire [4:0] startPosition3,
    input wire load,
    input wire clock,
    output wire [7:0] outChar
);
    wire clock1, clock2;
    wire [25:0] forwardData1, backward1;
    wire [25:0] forwardData2, backward2;
    wire [25:0] forwardData3, backward3;
    wire [25:0] forwardData4, backward4;
    wire [25:0] forwardData5, backward5;

    CharDecoder dec(char, forwardData1);
    PlugBoard p1(forwardData1, backward4 ,forwardData2, backward5);
    Rotor1 r1(forwardData2, backward3, startPosition1, load, clock, clock1, forwardData3, backward4);
    Rotor2 r2(forwardData3, backward2, startPosition2, load, clock1, clock2, forwardData4, backward3);
    Rotor3 r3(forwardData4, backward1, startPosition3, load, clock2, forwardData5, backward2);
    Reflector r4(forwardData5, backward1);
    CharEncoder enc(backward5, outChar);

endmodule
```

Figure 12: Enigma Machine Module

- There are *char*, *startPosition1*, *startPosition2*, *startPosition3* and *outChar* wires. They are used as input of the *charDecoder*, StartPosition information for Rotor1, StartPosition information for Rotor2, StartPosition information for Rotor3 and output of the Enigma machine, respectively.

- *PlugBoard*, *Rotor1*, *Rotor2*, *Rotor3* and *Reflector* are connected to each other.(see Figure 12)

- *charDecoder* decodes *char* into *forwardData1* and it is send to *PlugBoard*. *charEncoder* encodes *backward5*, which is backward output of the *PlugBoard*, into *outChar*.



```
module EnigmaCommunication(
    input wire [7:0] plainChar,
    input wire [4:0] startPosition1,
    input wire [4:0] startPosition2,
    input wire [4:0] startPosition3,
    input wire load,
    input wire clock,
    output wire [7:0] chipherChar,
    output wire [7:0] decryptedChar
);
    EnigmaMachine machine1(plainChar, startPosition1, startPosition2, startPosition3, load, clock, chipherChar);
    EnigmaMachine machine2(chipherChar, startPosition1, startPosition2, startPosition3, load, clock, decryptedChar);
endmodule
```

Figure 13: Enigma Communication Module

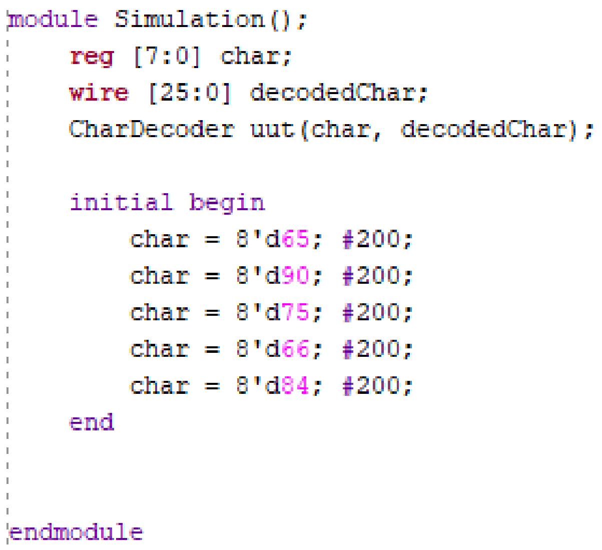
- There are *plainChar*, *startPosition1*, *startPosition2*, *startPosition3*, *chipherChar* and *decryptedChar* wires. They are used as the character to be transferred, StartPosition information for Rotor1, StartPosition information for Rotor2, StartPosition information for Rotor3, encrypted character which is the output of the first Enigma machine and also the input of the second Enigma machine and decrypted character which is the output of the second Enigma machine respectively.
- *plainChar* is the input of the first Enigma machine.
- At each part, except Reflector and PlugBoard Module, *load* is used to load the starting position data.
- *clockIn* is used for increasing the position counter at the rising edge.
- *clockOut* is used for providing clock signal for the next part. There is no *clockOut* for Rotor3 because it does not affect any Rotor in terms of clock signals.
- *clock* is used for providing clock signal to Rotor1 in both Enigma Machine and Enigma Communication Module.

3 RESULTS [15 points]

You can see the simulation codes we used for Experiment parts and their simulation results here. To carry out each different implementation of the experiment, we arranged inputs of each simulation according to part it belongs. The simulation results we obtained after completing each step of the experiments resulted in the way we expected. This concludes the fact that our experiment is coherent for each individual part.

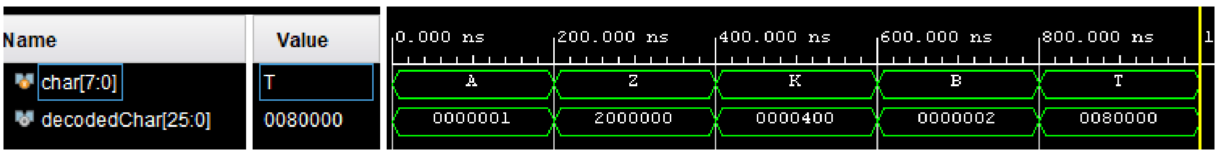
About part-1, see **Part 1** section above. In this part, we implemented 4 helper modules, which are:

- Char Decoder Module
- Char Encoder Module
- Circular Right Shift Module
- Circular Left Shift Module



```
module Simulation();
    reg [7:0] char;
    wire [25:0] decodedChar;
    CharDecoder uut(char, decodedChar);

    initial begin
        char = 8'd65; #200;
        char = 8'd90; #200;
        char = 8'd75; #200;
        char = 8'd66; #200;
        char = 8'd84; #200;
    end


endmodule
```

Figure 14: Simulation code of *Figure 1.a*

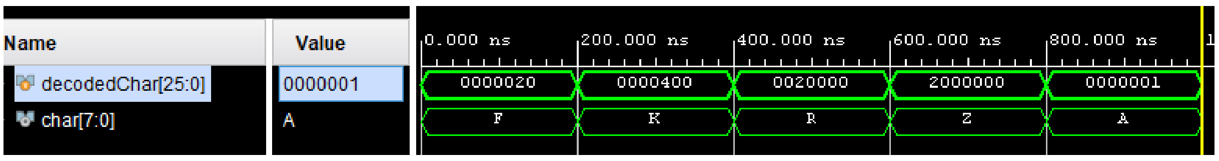


Name	Value	0.000 ns	200.000 ns	400.000 ns	600.000 ns	800.000 ns
char[7:0]	T	A	Z	K	B	T
decodedChar[25:0]	0080000	0000001	2000000	0000400	0000002	0080000

Figure 15: Simulation result of the Char Decoder Module

```
/* CharEncoder */
reg [25:0] decodedChar;
wire [7:0] char;
CharEncoder uut(decodedChar, char);

initial begin
    decodedChar = 26'h0000020; #200;
    decodedChar = 26'h0000400; #200;
    decodedChar = 26'h0020000; #200;
    decodedChar = 26'h2000000; #200;
    decodedChar = 26'h0000001; #200;
end
```

Figure 16: Simulation code of *Figure 1.b*

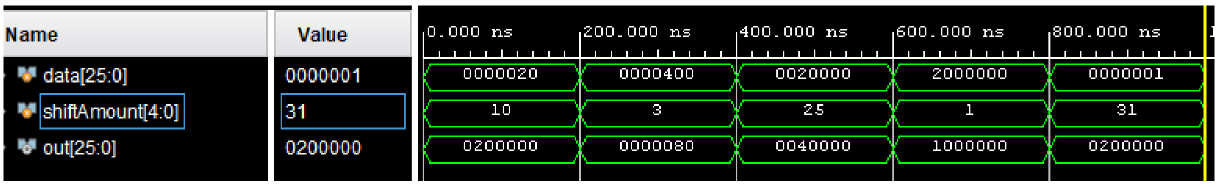


```
Name                 Value     |0.000 ns  |200.000 ns |400.000 ns |600.000 ns |800.000 ns
decodedChar[25:0]    0000001   | 0000020  | 0000400   | 0020000   | 2000000   | 0000001
char[7:0]            A         | F        | K         | R         | Z         | A
```

Figure 17: Simulation result of the Char Encoder Module

```
/*  CircularRightShift  */
reg [25:0] data;
reg [4:0] shiftAmount;
wire [25:0] out;
CircularRightShift uut(data, shiftAmount, out);

initial begin
    data = 26'h0000020; shiftAmount = 5'd10; #200;
    data = 26'h0000400; shiftAmount = 5'd3;  #200;
    data = 26'h0020000; shiftAmount = 5'd25; #200;
    data = 26'h2000000; shiftAmount = 5'd1;  #200;
    data = 26'h0000001; shiftAmount = 5'd31; #200;
end
```

Figure 18: Simulation code of *Figure 2.a*

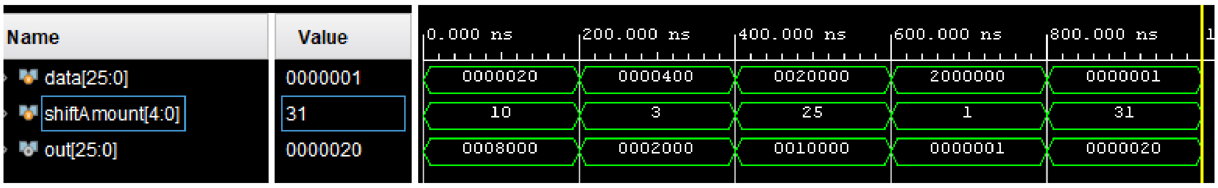


```
Name                 Value     |0.000 ns  |200.000 ns |400.000 ns |600.000 ns |800.000 ns
data[25:0]           0000001   | 0000020  | 0000400   | 0020000   | 2000000   | 0000001
shiftAmount[4:0]     31        | 10       | 3         | 25        | 1         | 31
out[25:0]            0200000   | 0200000  | 0000080   | 0040000   | 1000000   | 0200000
```

Figure 19: Simulation result of the Circular Right Shift Module

```
/*  CircularLeftShift */
reg [25:0] data;
reg [4:0] shiftAmount;
wire [25:0] out;
CircularLeftShift uut(data, shiftAmount, out);

initial begin
    data = 26'h0000020; shiftAmount = 5'd10; #200;
    data = 26'h0000400; shiftAmount = 5'd3;  #200;
    data = 26'h0020000; shiftAmount = 5'd25; #200;
    data = 26'h2000000; shiftAmount = 5'd1;  #200;
    data = 26'h0000001; shiftAmount = 5'd31; #200;
end
```

Figure 20: Simulation code of *Figure 2.b*



```
Name              Value      |0.000 ns  |200.000 ns |400.000 ns |600.000 ns |800.000 ns |1
data[25:0]        0000001    | 0000020  | 0000400   | 0020000   | 2000000   | 0000001
shiftAmount[4:0]  31         | 10       | 3         | 25        | 1         | 31
out[25:0]         0000020    | 0008000  | 0002000   | 0010000   | 0000001   | 0000020
```

Figure 21: Simulation result of the Circular Left Shift Module

About part-2, see **Part 2** section above. In this part, we implemented a Caesar Cipher Encryption Module, a Caesar Cipher Decryption Module and by using these two, we also implemented a Caesar Environment Module. For simulation codes and results, see figures below:

```
/*  CaesarEncryption  */
reg [7:0] plainChar;
reg [4:0] shiftCount;
wire [7:0] chipherChar;
CaesarEncryption uut(plainChar, shiftCount, chipherChar);

initial begin
    plainChar = 8'd65; shiftCount = 3; #200;
    plainChar = 8'd90; shiftCount = 10; #200;
    plainChar = 8'd75; shiftCount = 1; #200;
    plainChar = 8'd66; shiftCount = 15; #200;
    plainChar = 8'd84; shiftCount = 25; #200;
end
```

Figure 22: Simulation code of *Figure 3.a*

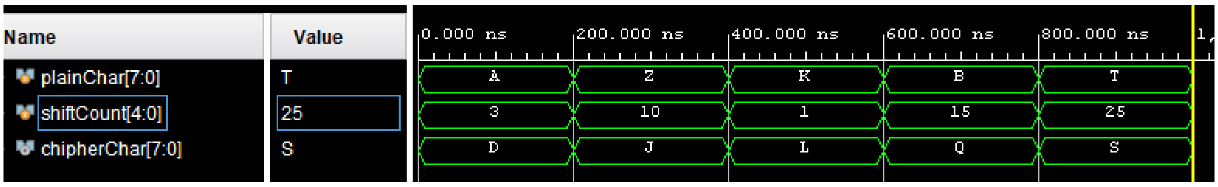
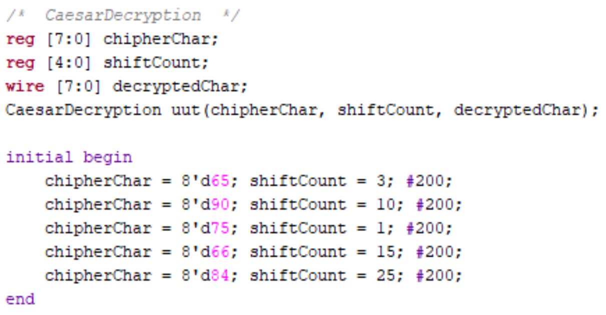


Figure 23: Simulation result of Caesar Cipher Encryption Module



```
/*  CaesarDecryption  */
reg [7:0] chipherChar;
reg [4:0] shiftCount;
wire [7:0] decryptedChar;
CaesarDecryption uut(chipherChar, shiftCount, decryptedChar);

initial begin
    chipherChar = 8'd65; shiftCount = 3; #200;
    chipherChar = 8'd90; shiftCount = 10; #200;
    chipherChar = 8'd75; shiftCount = 1; #200;
    chipherChar = 8'd66; shiftCount = 15; #200;
    chipherChar = 8'd84; shiftCount = 25; #200;
end
```

Figure 24: Simulation code of *Figure 3.b*

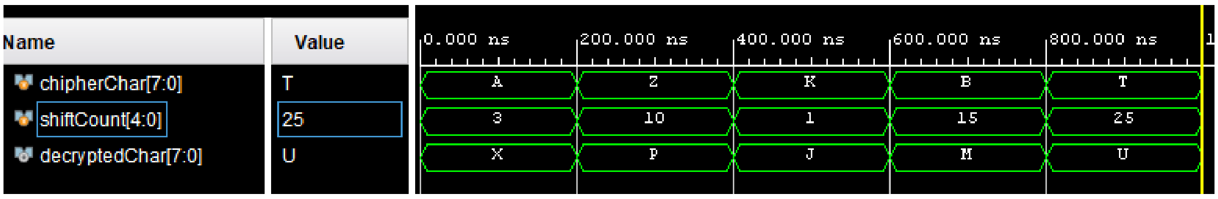
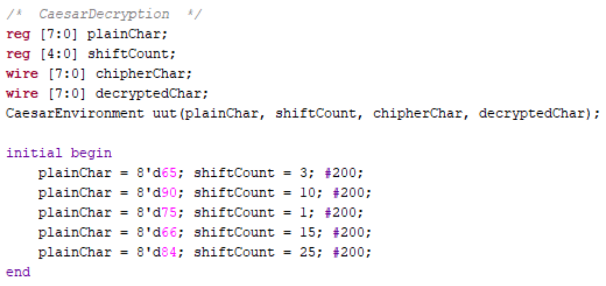


Figure 25: Simulation result of Caesar Cipher Decryption Module



```
/*  CaesarDecryption  */
reg [7:0] plainChar;
reg [4:0] shiftCount;
wire [7:0] chipherChar;
wire [7:0] decryptedChar;
CaesarEnvironment uut(plainChar, shiftCount, chipherChar, decryptedChar);

initial begin
    plainChar = 8'd65; shiftCount = 3; #200;
    plainChar = 8'd90; shiftCount = 10; #200;
    plainChar = 8'd75; shiftCount = 1; #200;
    plainChar = 8'd66; shiftCount = 15; #200;
    plainChar = 8'd84; shiftCount = 25; #200;
end
```

Figure 26: Simulation code of *Figure 4*

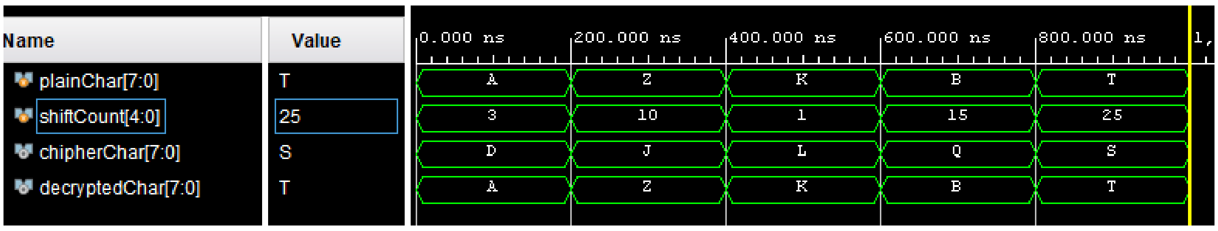
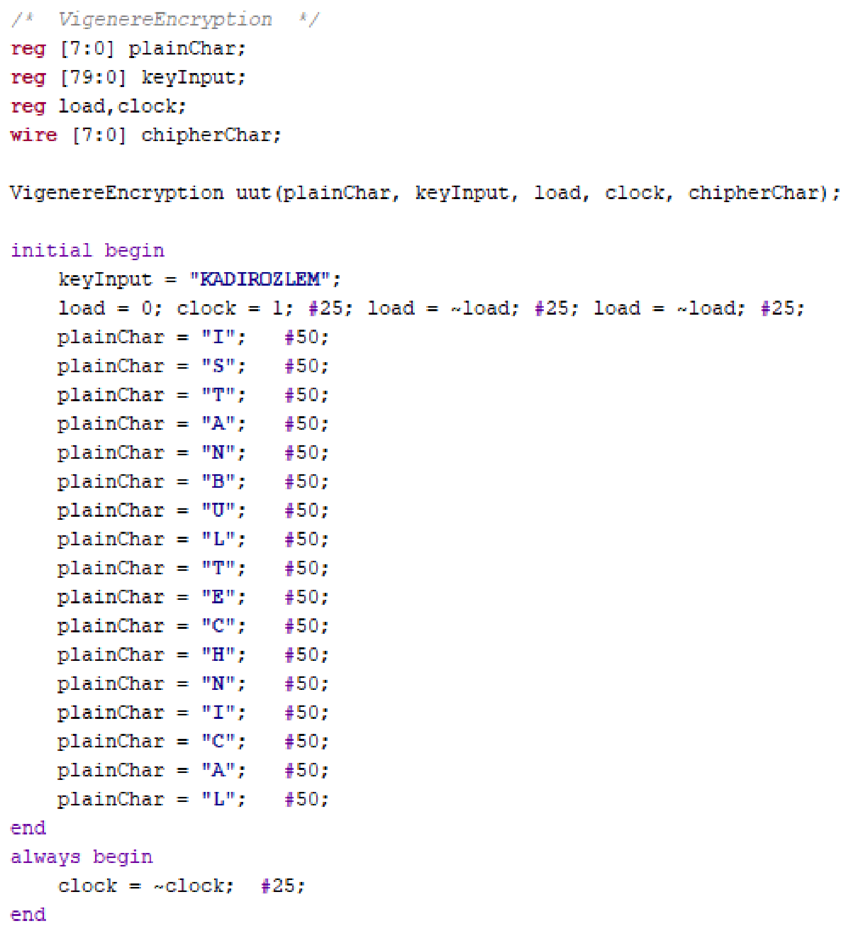


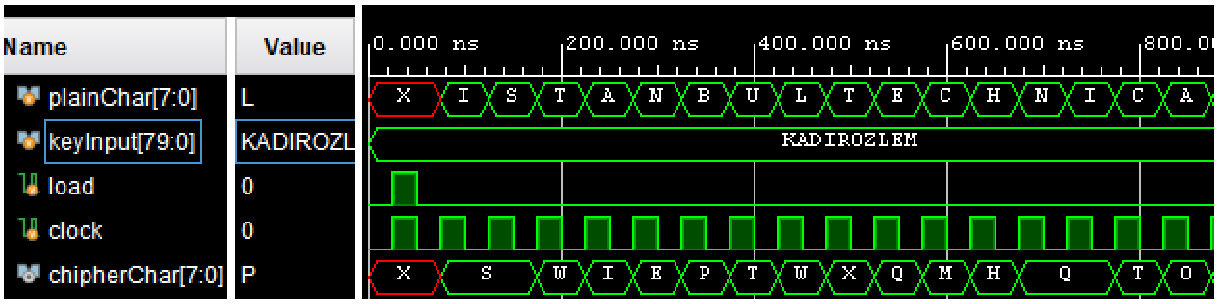
Figure 27: Simulation result of Caesar Environment Module

About part-3, see **Part 3** section above. In this part, we implemented a Vigenère Cipher Encryption Module, a Vigenère Cipher Decryption Module and by using these two, we also implemented a Vigenère Environment Module. For simulation codes and results, see figures below:



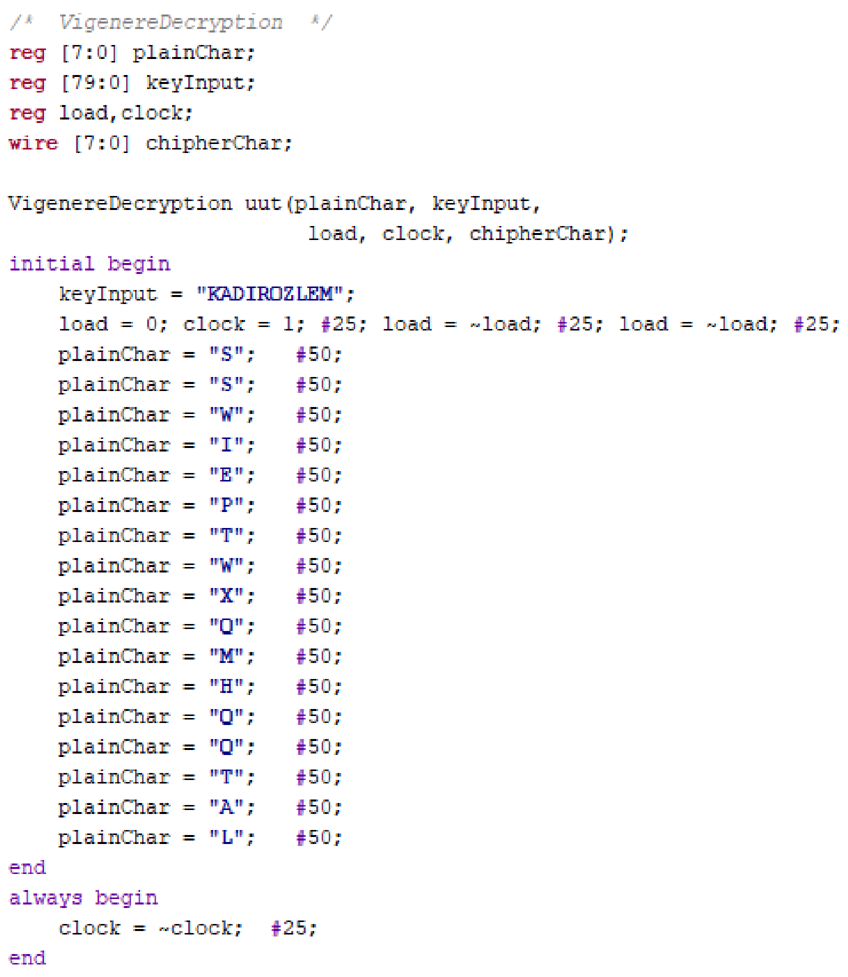
```
/*  VigenereEncryption  */
reg [7:0] plainChar;
reg [79:0] keyInput;
reg load,clock;
wire [7:0] chipherChar;

VigenereEncryption uut(plainChar, keyInput, load, clock, chipherChar);

initial begin
    keyInput = "KADIROZLEM";
    load = 0; clock = 1; #25; load = ~load; #25; load = ~load; #25;
    plainChar = "I";   #50;
    plainChar = "S";   #50;
    plainChar = "T";   #50;
    plainChar = "A";   #50;
    plainChar = "N";   #50;
    plainChar = "B";   #50;
    plainChar = "U";   #50;
    plainChar = "L";   #50;
    plainChar = "T";   #50;
    plainChar = "E";   #50;
    plainChar = "C";   #50;
    plainChar = "H";   #50;
    plainChar = "N";   #50;
    plainChar = "I";   #50;
    plainChar = "C";   #50;
    plainChar = "A";   #50;
    plainChar = "L";   #50;
end
always begin
    clock = ~clock;  #25;
end
```

Figure 28: Simulation code of *Figure 5.a*



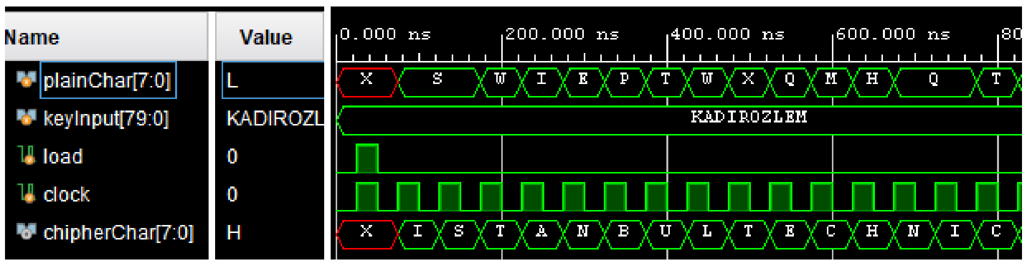
```
Name               Value       0.000 ns   200.000 ns   400.000 ns   600.000 ns   800.0
plainChar[7:0]     L           X I S T A N B U L T E C H N I C A
keyInput[79:0]     KADIROZL    KADIROZLEM
load               0
clock              0
chipherChar[7:0]   P           X S W I E P T W X Q M H Q T O
```

Figure 29: Simulation result of Vigenère Cipher Encryption Module



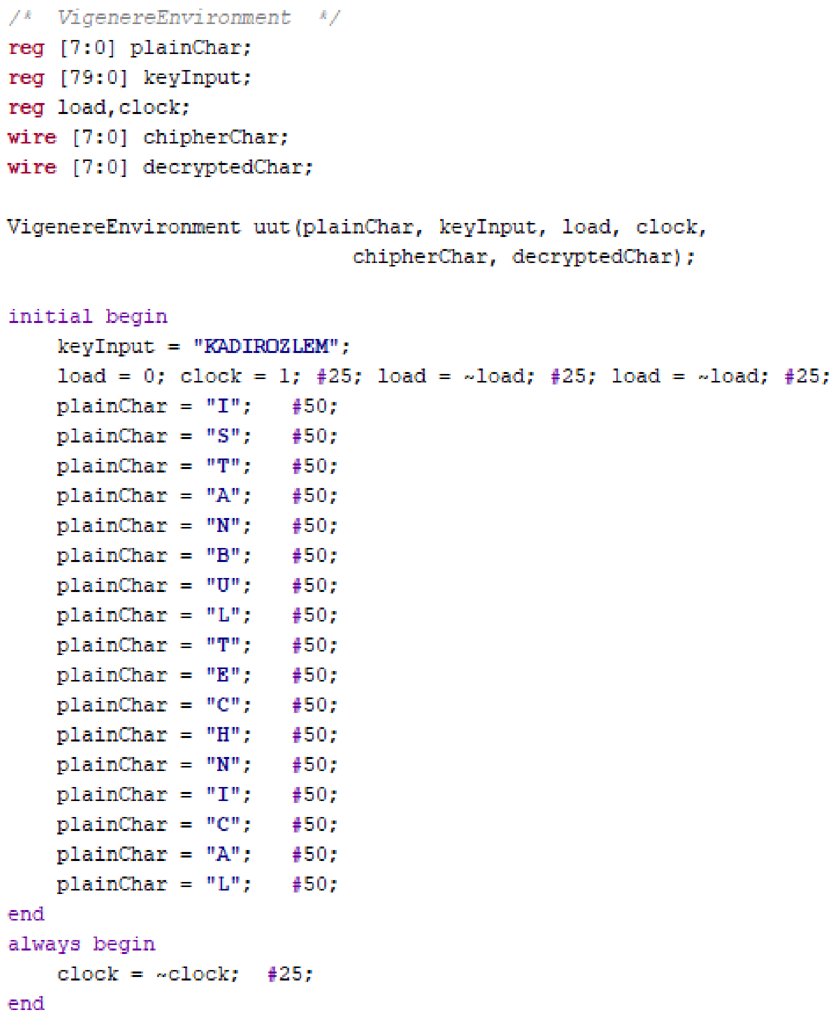
```
/*  VigenereDecryption  */
reg [7:0] plainChar;
reg [79:0] keyInput;
reg load,clock;
wire [7:0] chipherChar;

VigenereDecryption uut(plainChar, keyInput,
                        load, clock, chipherChar);
initial begin
    keyInput = "KADIROZLEM";
    load = 0; clock = 1; #25; load = ~load; #25; load = ~load; #25;
    plainChar = "S";   #50;
    plainChar = "S";   #50;
    plainChar = "W";   #50;
    plainChar = "I";   #50;
    plainChar = "E";   #50;
    plainChar = "P";   #50;
    plainChar = "T";   #50;
    plainChar = "W";   #50;
    plainChar = "X";   #50;
    plainChar = "Q";   #50;
    plainChar = "M";   #50;
    plainChar = "H";   #50;
    plainChar = "Q";   #50;
    plainChar = "Q";   #50;
    plainChar = "T";   #50;
    plainChar = "A";   #50;
    plainChar = "L";   #50;
end
always begin
    clock = ~clock;  #25;
end
```

Figure 30: Simulation code of *Figure 5.b*



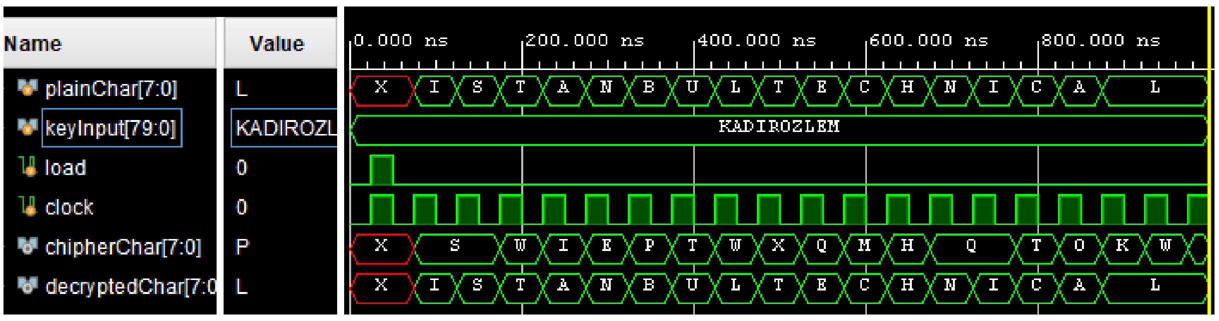
Name	Value
plainChar[7:0]	L
keyInput[79:0]	KADIROZL
load	0
clock	0
chipherChar[7:0]	H

Figure 31: Simulation result of Vigenère Cipher Decryption Module



```
/*  VigenereEnvironment  */
reg [7:0] plainChar;
reg [79:0] keyInput;
reg load,clock;
wire [7:0] chipherChar;
wire [7:0] decryptedChar;

VigenereEnvironment uut(plainChar, keyInput, load, clock,
                            chipherChar, decryptedChar);

initial begin
    keyInput = "KADIROZLEM";
    load = 0; clock = 1; #25; load = ~load; #25; load = ~load; #25;
    plainChar = "I";   #50;
    plainChar = "S";   #50;
    plainChar = "T";   #50;
    plainChar = "A";   #50;
    plainChar = "N";   #50;
    plainChar = "B";   #50;
    plainChar = "U";   #50;
    plainChar = "L";   #50;
    plainChar = "T";   #50;
    plainChar = "E";   #50;
    plainChar = "C";   #50;
    plainChar = "H";   #50;
    plainChar = "N";   #50;
    plainChar = "I";   #50;
    plainChar = "C";   #50;
    plainChar = "A";   #50;
    plainChar = "L";   #50;
end
always begin
    clock = ~clock;  #25;
end
```

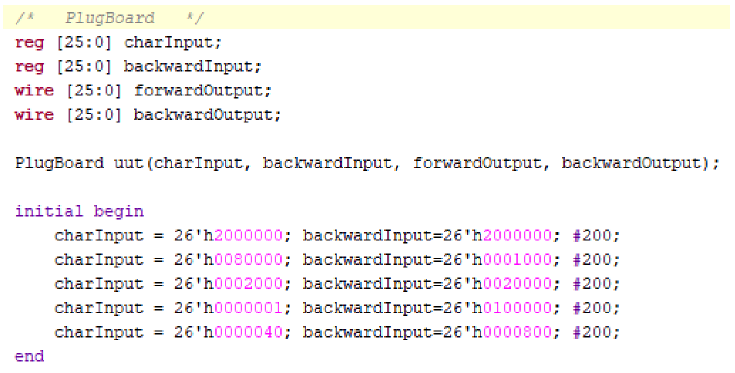
Figure 32: Simulation code of *Figure 6*



```
Name                Value   0.000 ns   200.000 ns   400.000 ns   600.000 ns   800.000 ns
plainChar[7:0]      L       X I S T A N B U L T E C H N I C A L
keyInput[79:0]      KADIROZL  KADIROZLEM
load                0
clock               0
chipherChar[7:0]    P       X S W I E P T W X Q M H Q T O K W
decryptedChar[7:0]  L       X I S T A N B U L T E C H N I C A L
```

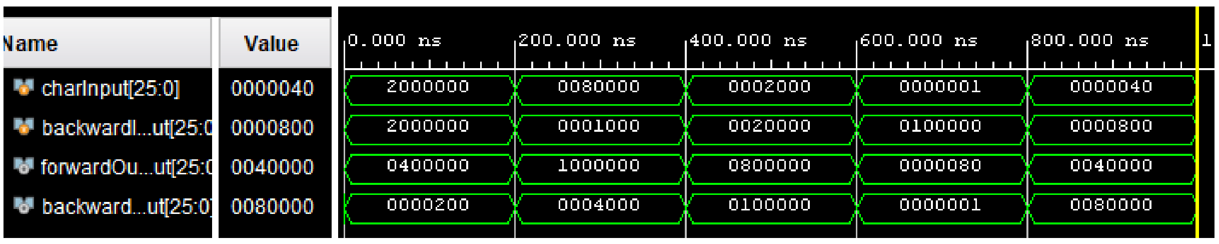
Figure 33: Simulation result of Vigenère Environment Module

About part-4, see **Part 4** section above. In this part, we implemented a Plugboard Module, 3 Rotor modules, a Reflector Module and by using these we also implemented a Enigma Machine Module. Also we implemented a Enigma Communication Module by using 2 Enigma Machine Module. For simulation codes and results, see figures below:



```
/*   PlugBoard   */
reg [25:0] charInput;
reg [25:0] backwardInput;
wire [25:0] forwardOutput;
wire [25:0] backwardOutput;

PlugBoard uut(charInput, backwardInput, forwardOutput, backwardOutput);

initial begin
    charInput = 26'h2000000; backwardInput=26'h2000000; #200;
    charInput = 26'h0080000; backwardInput=26'h0001000; #200;
    charInput = 26'h0002000; backwardInput=26'h0020000; #200;
    charInput = 26'h0000001; backwardInput=26'h0100000; #200;
    charInput = 26'h0000040; backwardInput=26'h0000800; #200;
end
```

Figure 34: Simulation code of Plugboard Module



```
Name                 Value     |0.000 ns   |200.000 ns  |400.000 ns  |600.000 ns  |800.000 ns  |1
charInput[25:0]      0000040     2000000     0080000      0002000      0000001      0000040
backwardI...ut[25:0  0000800     2000000     0001000      0020000      0100000      0000800
forwardOu...ut[25:0  0040000     0400000     1000000      0800000      0000080      0040000
backward...ut[25:0]  0080000     0000200     0004000      0100000      0000001      0080000
```

Figure 35: Simulation result of Plugboard Module

```
/*    Rotor1    */
reg [25:0] charInput;
reg [25:0] backwardInput;
reg [4:0] startPosition1;
reg load, clock;
wire clock1;
wire [25:0] forwardOutput;
wire [25:0] backwardOutput;
Rotor1 uut(charInput, backwardInput, startPosition1, load, clock,
                              clock1, forwardOutput, backwardOutput);

initial begin
    clock = 0; charInput = 26'h2000000; backwardInput = 26'h0000200;
    load = 1;  startPosition1 = 5'b00000; #200;
    load = 0;
    charInput = 26'h2000000; backwardInput = 26'h0000200;load = 0; #200;
    charInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    charInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    charInput = 26'h2000000; backwardInput = 26'h0000200; #200;
end
always begin
    clock = ~clock; #100;
end
```

Figure 36: Simulation code of Rotor1 Module

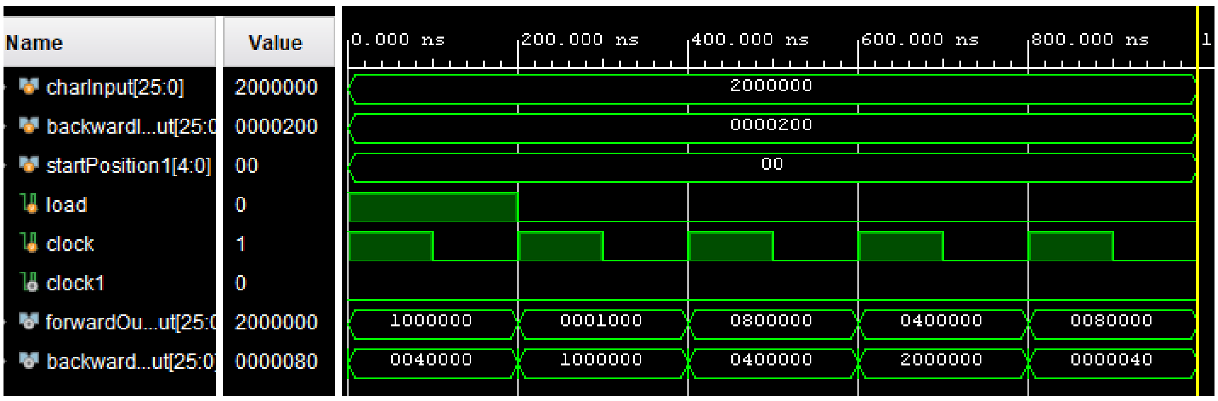

```
Name                 Value      0.000 ns  200.000 ns  400.000 ns  600.000 ns  800.000 ns
charInput[25:0]      2000000    2000000
backwardl...ut[25:0  0000200    0000200
startPosition1[4:0]  00         00
load                 0
clock                1
clock1               0
forwardOu...ut[25:0  2000000    1000000   0001000   0800000   0400000   0080000
backward...ut[25:0]  0000080    0040000   1000000   0400000   2000000   0000040
```

Figure 37: Simulation result of Rotor1 Module

```
/*    Rotor2    */
reg [25:0] forwardInput;
reg [25:0] backwardInput;
reg [4:0] startPosition1;
reg load, clock;
wire clock1;
wire [25:0] forwardOutput;
wire [25:0] backwardOutput;
Rotor2 uut(forwardInput, backwardInput, startPosition1, load, clock,
                              clock1, forwardOutput, backwardOutput);

initial begin
    clock = 0; forwardInput = 26'h2000000; backwardInput = 26'h0000200;
    load = 1;  startPosition1 = 5'b00000; #200;
    load = 0;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200;load = 0; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
end
always begin
    clock = ~clock; #100;
end
```

Figure 38: Simulation code of Rotor2 Module

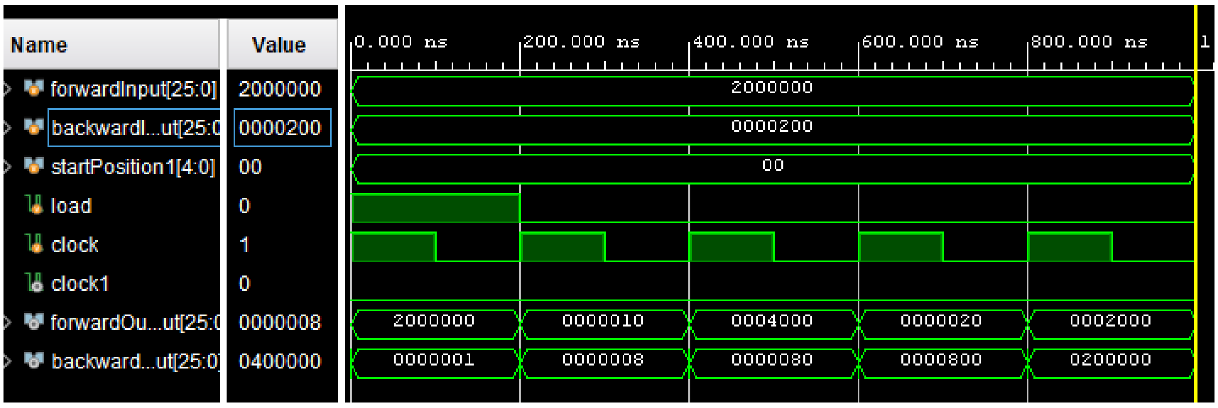


Figure 39: Simulation result of Rotor2 Module

```
/*    Rotor3    */
reg [25:0] forwardInput;
reg [25:0] backwardInput;
reg [4:0] startPosition3;
reg load, clock;
wire [25:0] forwardOutput;
wire [25:0] backwardOutput;
Rotor3 uut(forwardInput, backwardInput, startPosition3, load, clock,
                                   forwardOutput, backwardOutput);

initial begin
    clock = 0; forwardInput = 26'h2000000; backwardInput = 26'h0000200;
    load = 1;  startPosition3 = 5'b00000; #200;
    load = 0;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200;load = 0; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
    forwardInput = 26'h2000000; backwardInput = 26'h0000200; #200;
end
always begin
    clock = ~clock; #100;
end
```

Figure 40: Simulation code of Rotor3 Module

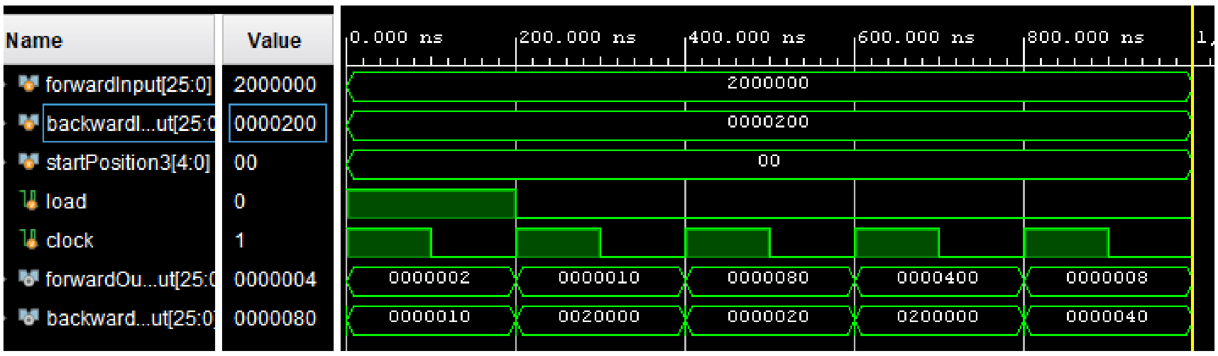
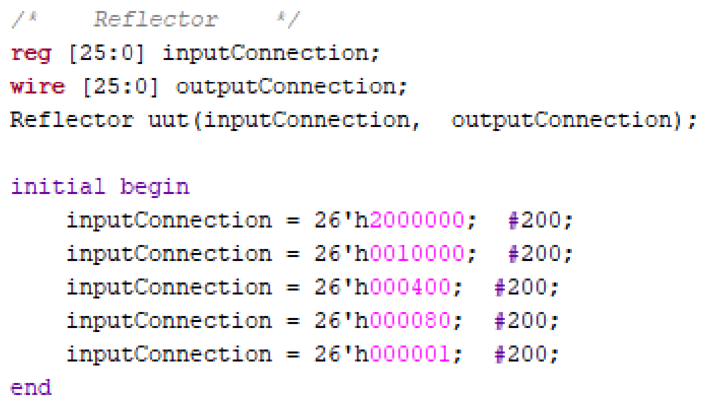
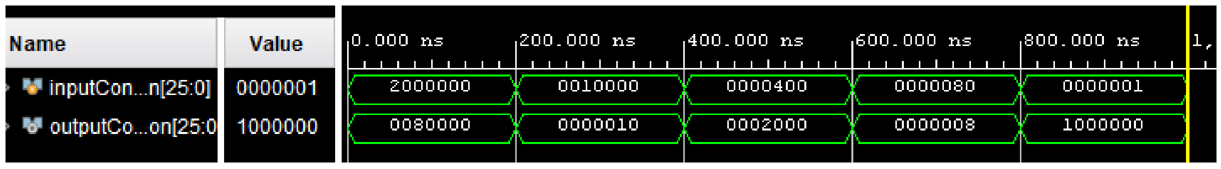


Figure 41: Simulation result of Rotor3 Module



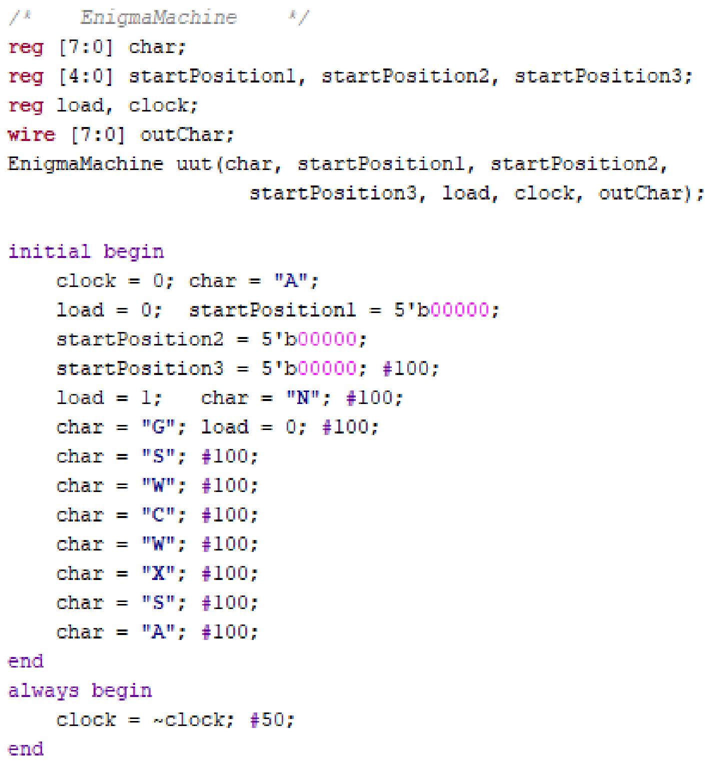
```verilog
/*    Reflector    */
reg [25:0] inputConnection;
wire [25:0] outputConnection;
Reflector uut(inputConnection,  outputConnection);

initial begin
    inputConnection = 26'h2000000;  #200;
    inputConnection = 26'h0010000;  #200;
    inputConnection = 26'h000400;  #200;
    inputConnection = 26'h000080;  #200;
    inputConnection = 26'h000001;  #200;
end
```

Figure 42: Simulation code of Reflector Module



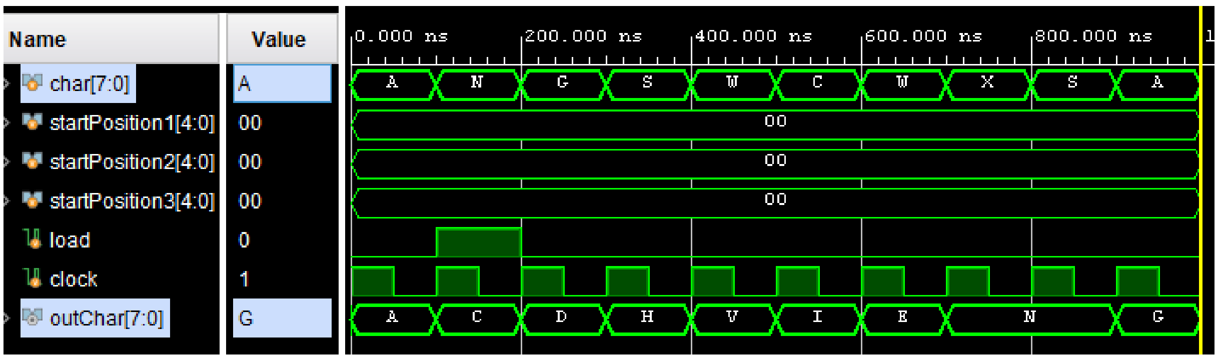
```
Name                 Value     |0.000 ns  |200.000 ns |400.000 ns |600.000 ns |800.000 ns |1,
inputCon...n[25:0]   0000001   | 2000000  | 0010000   | 0000400   | 0000080   | 0000001   |
outputCo...on[25:0   1000000   | 0080000  | 0000010   | 0002000   | 0000008   | 1000000   |
```

Figure 43: Simulation result of Reflector Module



```verilog
/*    EnigmaMachine    */
reg [7:0] char;
reg [4:0] startPosition1, startPosition2, startPosition3;
reg load, clock;
wire [7:0] outChar;
EnigmaMachine uut(char, startPosition1, startPosition2,
                    startPosition3, load, clock, outChar);

initial begin
    clock = 0; char = "A";
    load = 0;  startPosition1 = 5'b00000;
    startPosition2 = 5'b00000;
    startPosition3 = 5'b00000; #100;
    load = 1;   char = "N"; #100;
    char = "G"; load = 0; #100;
    char = "S"; #100;
    char = "W"; #100;
    char = "C"; #100;
    char = "W"; #100;
    char = "X"; #100;
    char = "S"; #100;
    char = "A"; #100;
end
always begin
    clock = ~clock; #50;
end
```

Figure 44: Simulation code of Enigma Machine Module

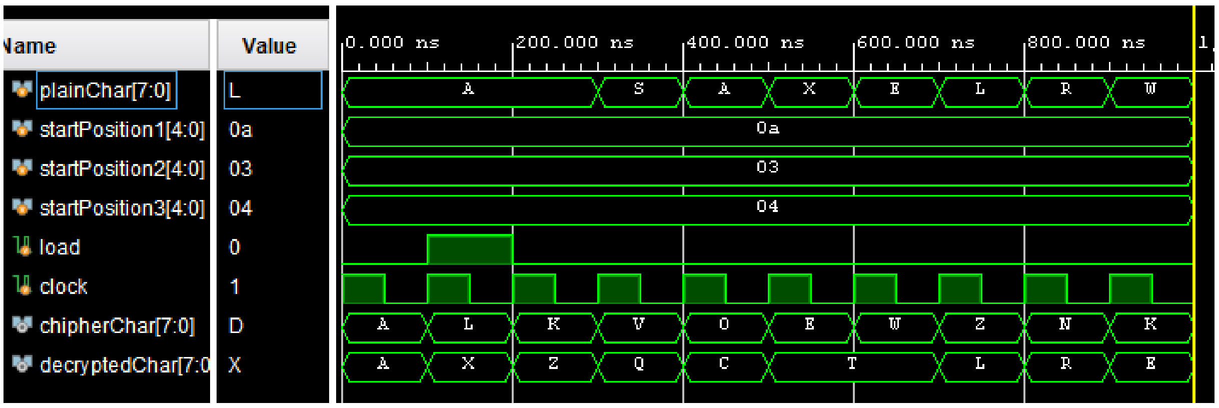


```
Name                 Value   0.000 ns   200.000 ns   400.000 ns   600.000 ns   800.000 ns
char[7:0]            A       A  N  G  S  W  C  W  X  S  A
startPosition1[4:0]  00      00
startPosition2[4:0]  00      00
startPosition3[4:0]  00      00
load                 0
clock                1
outChar[7:0]         G       A  C  D  H  V  I  E  N  G
```

Figure 45: Simulation result of Enigma Machine Module

```
/*    EnigmaCommunication    */
reg [7:0] plainChar;
reg [4:0] startPosition1, startPosition2, startPosition3;
reg load, clock;
wire [7:0] chipherChar, decryptedChar;
EnigmaCommunication uut(plainChar, startPosition1, startPosition2,
            startPosition3, load, clock, chipherChar, decryptedChar);

initial begin
    clock = 0; plainChar = "A";
    load = 0;  startPosition1 = 5'b01010;
    startPosition2 = 5'b00011; startPosition3 = 5'b00100; #100;
    load = 1; #100;
    load = 0; #100;
    plainChar = "S"; #100;
    plainChar = "A"; #100;
    plainChar = "X"; #100;
    plainChar = "E"; #100;
    plainChar = "L"; #100;
    plainChar = "R"; #100;
    plainChar = "W"; #100;
    plainChar = "L"; #100;
end
always begin
    clock = ~clock; #50;
end
```

Figure 46: Simulation code of Enigma Communication Module



```
Name                 Value   0.000 ns   200.000 ns   400.000 ns   600.000 ns   800.000 ns
plainChar[7:0]       L       A  S  A  X  E  L  R  W
startPosition1[4:0]  0a      0a
startPosition2[4:0]  03      03
startPosition3[4:0]  04      04
load                 0
clock                1
chipherChar[7:0]     D       A  L  K  V  O  E  W  Z  N  K
decryptedChar[7:0    X       A  X  Z  Q  C  T  L  R  E
```

Figure 47: Simulation result of Enigma Communication Module

4 DISCUSSION [25 points]

First, we implemented 4 helper modules given below.

- CharDecoder Module to transform A-Z chars to binary decoded version. It takes char, an 8-bit input (ASCII code of the character) as input and gives decodedChar, 26-bit output (Decoded output of the character).

  Ex: 'A' → h0000001, 'Z' → h2000000, 'T' → h0080000

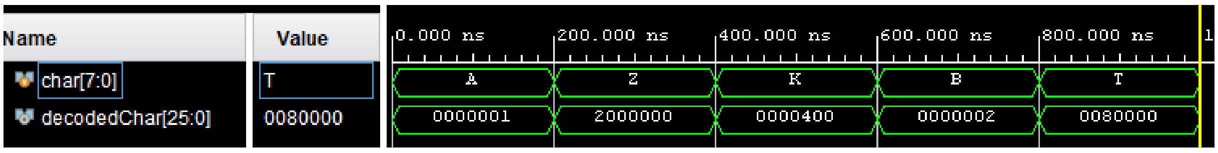


Figure 48: Simulation result of the Char Decoder Module

- CharEncoder Module to transform binary decoded version of the char to ASCII code for A-Z char. It takes a decodedChar, which is a 26-bit input (Decoded input of the character), and gives a char, which is an 8-bit output (ASCII code of the character)

  Ex: h0000020 → 'F', h0000400 → 'K', h0020000 → 'R'

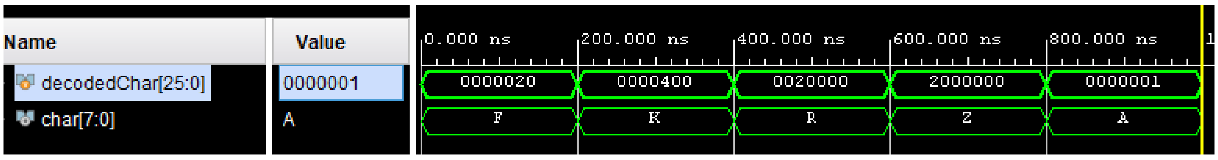


Figure 49: Simulation result of the Char Encoder Module

- CircularRightShift Module to obtain a right Shift Module with shift count. This gets a data, which is a 26-bit input (Input data) and a shiftAmount, which is a 5-bit input (Shift amount), and returns out, which is a 26-bit output (Data shifted to the right 'shiftAmount' times)

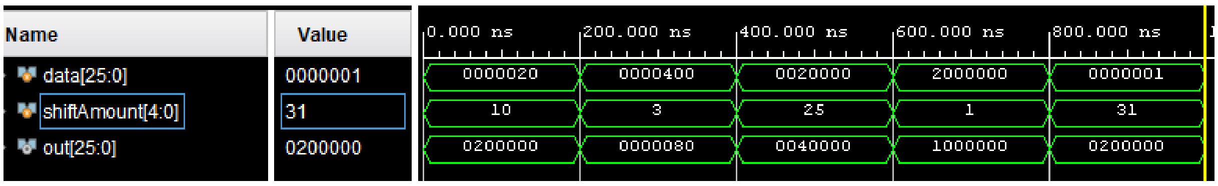


Figure 50: Simulation result of the Circular Right Shift Module

- CircularLeftShift Module to obtain left Shift Module with shift count. This also gets 2 inputs, first is data, a 26-bit input (Input Data) and second one is shiftAmount, a 5-bit input (Shift amount) and returns 1 output, which is out, a 26-bit output (Data shifted to the left ‘shiftAmount’ times)

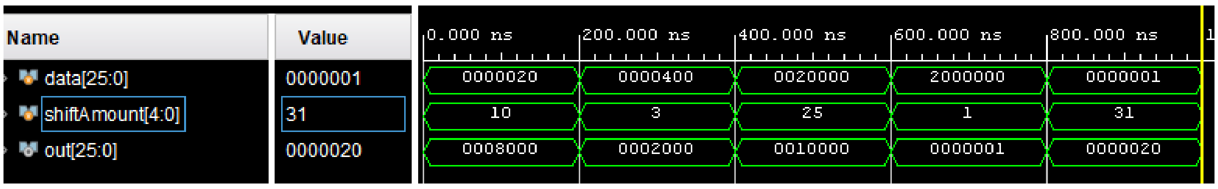


Figure 51: Simulation result of the Circular Left Shift Module

Secondly on Part 2, we first implemented a Caesar Cipher Encryption module, and then Caesar Cipher Decryption module.

- CaesarEncryption Module to encrypt the char using Caesar Cipher technique, this gets 2 inputs, a plainChar which is an 8-bit input (ASCII code of the input character) and shiftCount which is a 5-bit input: (Shift amount of the cipher), and returns an 8-bit output, namely chipherChar which is the encrypted character using Caesar Cipher with shiftCount.

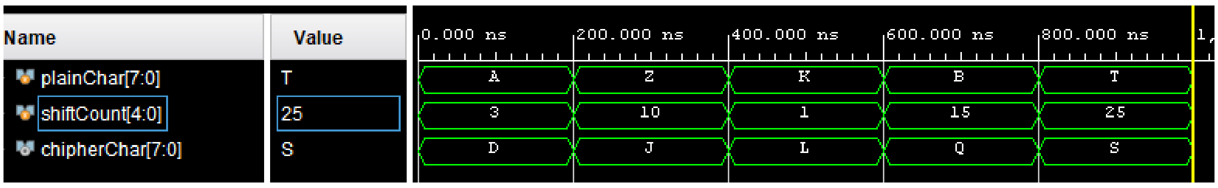


Figure 52: Simulation result of Caesar Cipher Encryption Module

- CaesarDecryption Module to decrypt the encrypted char using Caesar Cipher technique, this gets 2 inputs, a chipherChar which is an 8-bit input (Encrypted char input) and shiftCount which is a 5-bit input (Shift amount of the cipher), and returns an 8-bit output, namely chipherChar which is the decrypted character using Caesar Cipher with shiftCount.

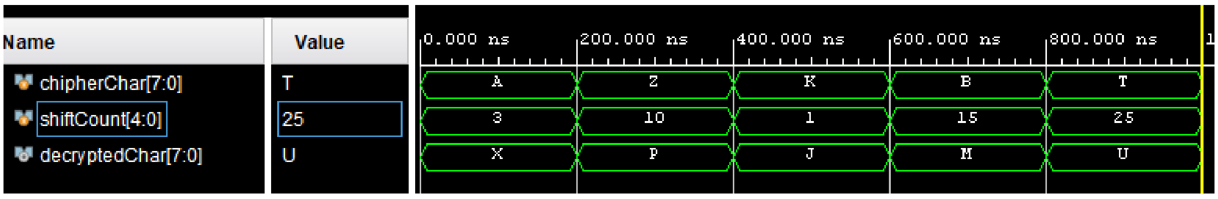


Figure 53: Simulation result of Caesar Cipher Decryption Module

Then we created a Caesar Module to show encrypted and decrypted messages.

- CaesarEnvironment Module: Encryption and decryption processes of a plain char. This takes 2 inputs and returns 2 outputs. Input-1 is plainChar (8-bit input) which is ASCII code of the input character. Input-2 is shiftCount (5-bit input) which is Shift amount of the cipher. Output-1 is chipherChar (8-bit output) which is The character which is encrypted using Caesar Cipher with shiftCount. And finally output-2 is decryptedChar (8-bit output) which is The character which is decrypted using Caesar Cipher with shiftCount.

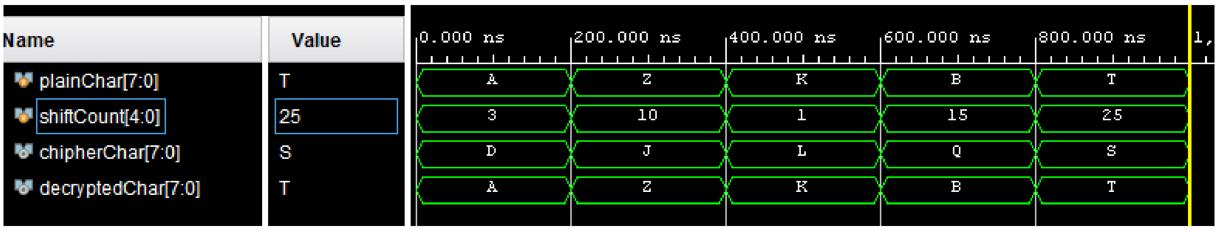


Figure 54: Simulation result of Caesar Environment Module

In part-3, we implemented a Vigenère Cipher Encryption Module, a Vigenère Cipher Decryption Module and by using these two, we also implemented a Vigenère Environment Module, just like we did in the above part-2 but instead of Caesar, we had Vigenère this time. So we used the given equations while doing that.

- VigenereEncryption Module to encrypt the char using Vigenère Cipher technique, this gets 4 inputs, a plainChar which is an 8-bit input (ASCII code of the input character), a keyInput which is an 80-bit input (Key message of the cipher (10 characters)), a load which is a 1-bit input (Load the keyInput to key register at the rising edge of the load signal) and a clock which is a 1-bit input (Shift key register only one character at the rising edge of the clock signal (when load = 0)), and returns an 8-bit output, namely chipherChar which is the encrypted character using Vigenère Cipher.

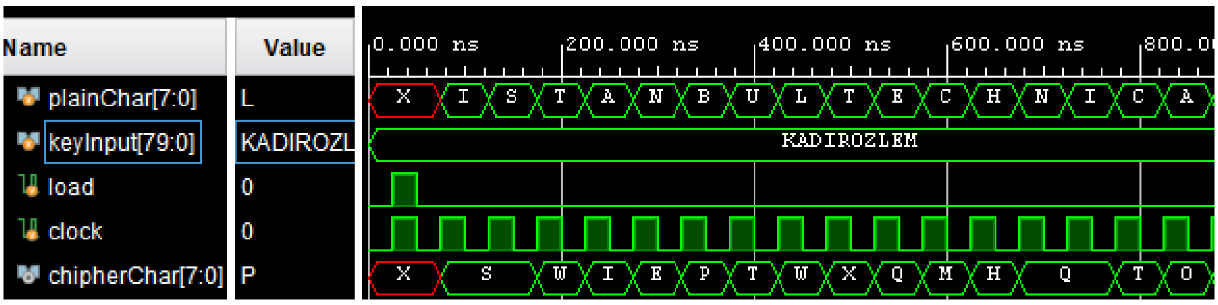


Figure 55: Simulation result of Vigenère Cipher Encryption Module

- VigenereDecryption Module to Decrypt the encrypted char using Vigenère Cipher technique, this gets 4 inputs, a chipherChar which is an 8-bit input (ASCII code of the input character), a keyInput which is an 80-bit input (Key message of the cipher (10 characters)), a load which is a 1-bit input (Load the keyInput to key register at the rising edge of the load signal) and a clock which is a 1-bit input (Shift key register only one character at the rising edge of the clock signal (when load = 0)), and returns an 8-bit output, namely decryptedChar which is the decrypted character using Vigenère Cipher.

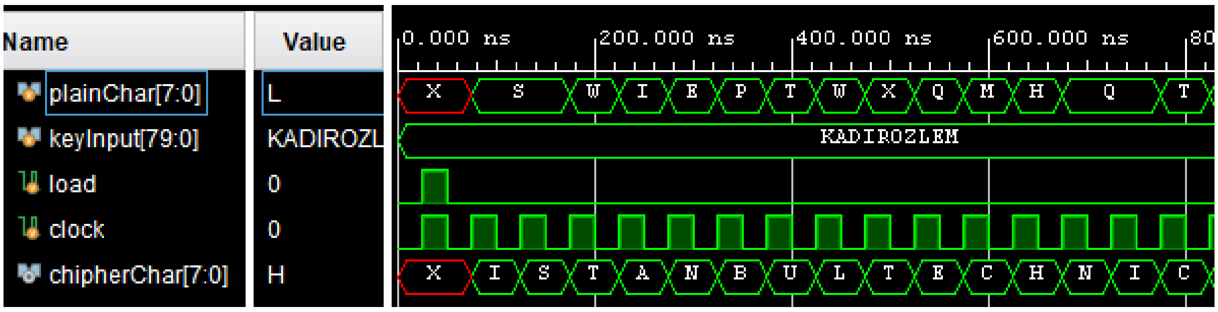


Figure 56: Simulation result of Vigenère Cipher Decryption Module

- VigenereEnvironment Module for encryption and decryption processes of the plain char, this gets 4 inputs, a plainChar which is an 8-bit input (ASCII code of the input character), a keyInput which is an 80-bit input (Key message of the cipher (10 characters)), a load which is a 1-bit input (Load the keyInput to key register at the rising edge of the load signal) and a clock which is a 1-bit input (Shift key register only one character at the rising edge of the clock signal (when load = 0)), and returns 2 outputs, an 8-bit output, namely chipherChar which is the encrypted character using Vigenère Cipher, and an 8-bit output, namely decryptedChar which is the decrypted character using Vigenère Cipher.

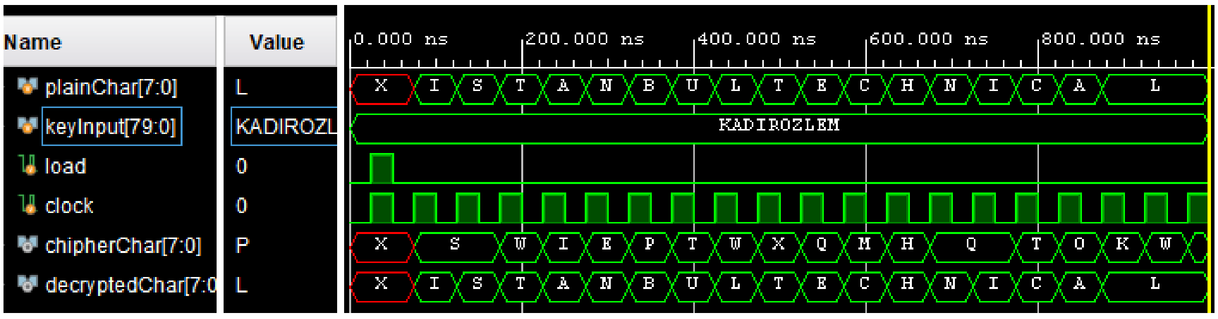


Figure 57: Simulation result of Vigenère Environment Module

In part **four**, we implement the Enigma machine which can both encrypt and decrypt a message. You can see the simulation results below:

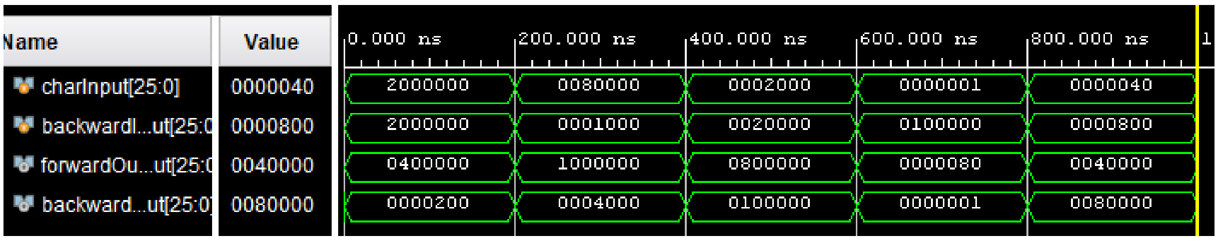

Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
charInput[25:0] 0000040 2000000 0080000 0002000 0000001 0000040
backwardI...ut[25:0 0000800 2000000 0001000 0020000 0100000 0000800
forwardOu...ut[25:0 0400000 0400000 1000000 0800000 0000080 0040000
backward...ut[25:0] 0080000 0000200 0004000 0100000 0000001 0080000

Figure 58: Simulation result of Plugboard Module

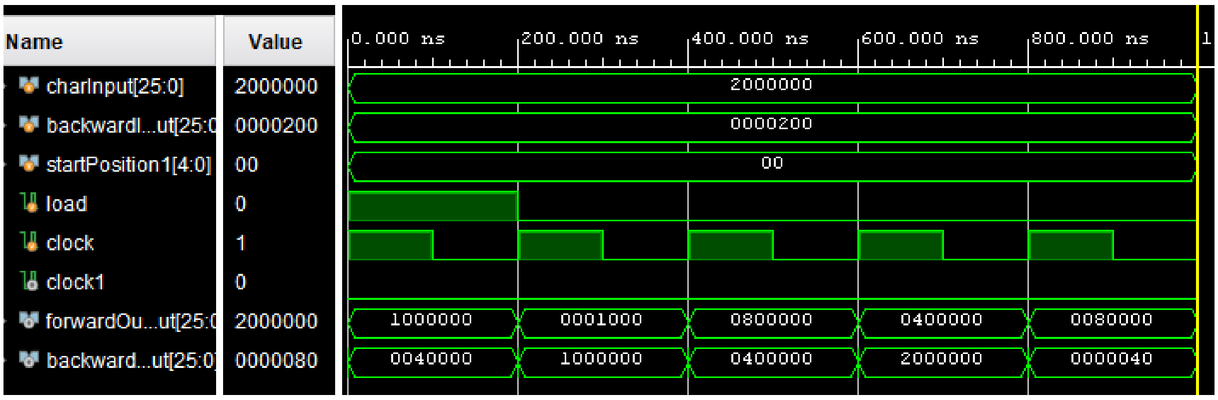

Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
charInput[25:0] 2000000 2000000
backwardI...ut[25:0 0000200 0000200
startPosition1[4:0] 00 00
load 0
clock 1
clock1 0
forwardOu...ut[25:0 2000000 1000000 0001000 0800000 0400000 0080000
backward...ut[25:0] 0000080 0040000 1000000 0400000 2000000 0000040

Figure 59: Simulation result of Rotor1 Module

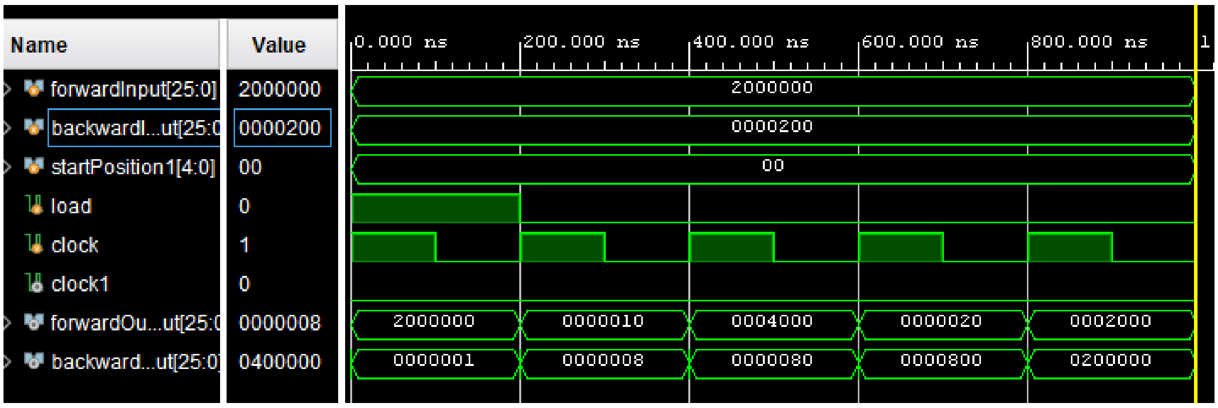

Name
Value
0.000 ns
200.000 ns
400.000 ns
600.000 ns
800.000 ns
forwardInput[25:0] 2000000 2000000
backwardI...ut[25:0 0000200 0000200
startPosition1[4:0] 00 00
load 0
clock 1
clock1 0
forwardOu...ut[25:0 0000008 2000000 0000010 0004000 0000020 0002000
backward...ut[25:0] 0400000 0000001 0000008 0000080 0000800 0200000

Figure 60: Simulation result of Rotor2 Module

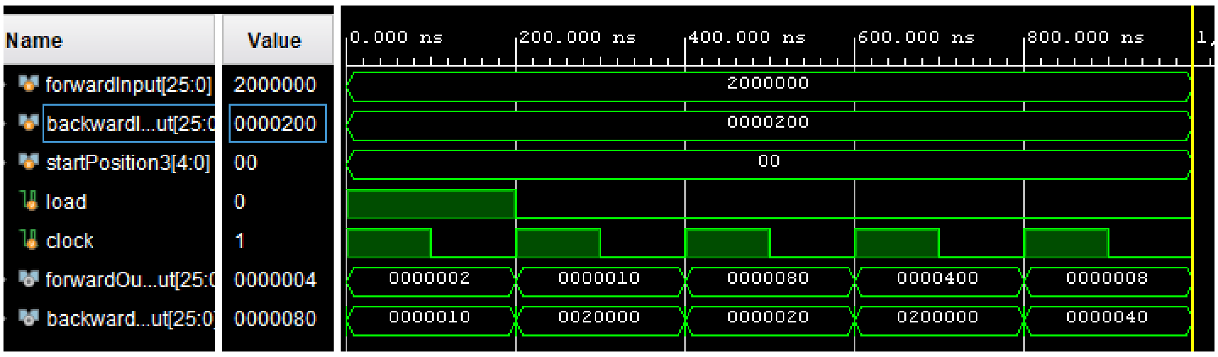


Figure 61: Simulation result of Rotor3 Module

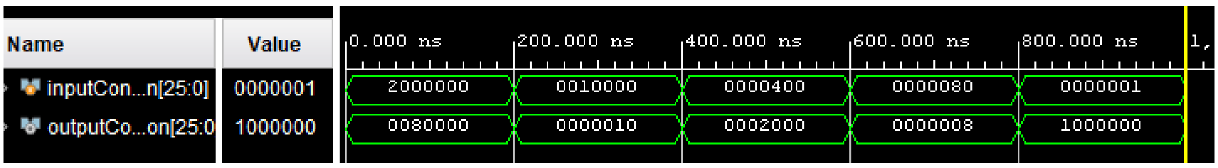


Figure 62: Simulation result of Reflector Module

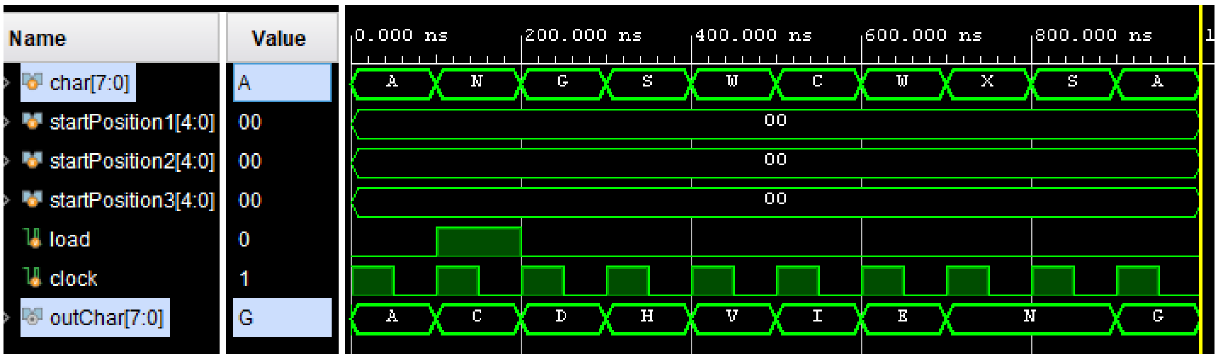


Figure 63: Simulation result of Enigma Machine Module

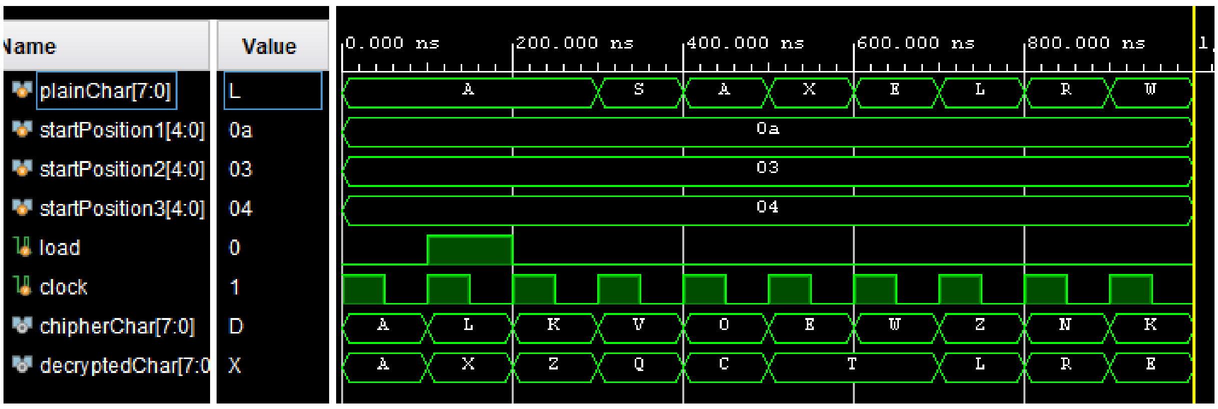


Figure 64: Simulation result of Enigma Communication Module

5 CONCLUSION [10 points]

In Part 1, we designed the helper modules which we use in the other parts. The modules that we implement are for decoding or encoding the characters and also, shifting the characters.

Then in Part 2, we implemented Ceaser Cipher. The encryption and Decryption modules are very easy according to the other parts. We only shift the characters and then we simulate parts.

In Part 3, we implemented Vigenere Cipher. It was more complicate than Ceaser Cipher but it was also simple. Basically, we just sum the character with the key characters. And, we have a cipher message which can only decryptable with the key input. After that, we encrypt and decrypt a message with using a key properly.

Finally, we tried to implement an Enigma Machine. We understood how it is actually working. Also, as we simulate its modules are performing properly. However, when we try to combine all parts and implement a Enigma Machine, it is encrypting and decrypting messages however, not in properly. It is just running. We can not understand what is the problem.

In conclusion, we tried to implement different cryptology methods. We learn lots of thing but we could not reach complete success.