IMPLEMENTATION OF ENIGMA MACHINE ON FPGA

Submitted in partial fulfillment of the requirements

of the degree

By

Taras Rajan 21ECB0B61

Veera Kumar 21ECB0B62

Sai Sharanya 21ECB0B63

Guided by:

Dr. P. Prithvi

Assistant Professor

Dr V. Narendar

Assistant Professor

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL

2022-2023

CERTIFICATE

This is to certify that the dissertation work entitled **IMPLEMENTATION OF ENIGMA MACHINE ON FPGA** is a bonafide record of work carried out work by Taras Rajan (21ECB0B61) Veera Kumar(21ECB0B62) and Sai Sharanya (21ECB0B63) submitted to faculty of "Electronics and Communication Engineering Department" , in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in "Electronics and Communication Engineering" at National Institute of Technology, Warangal during academic year (2022-2023).

Dr. P. Prithvi

Assistant professor

Department of Electronics and Communication Engineering

National Institute of Technology

Dr. V. Narendar

Assistant professor

Department of Electronics and Communication Engineering

National Institute of Technology

ACKNOWLEDGEMENT

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

ABSTRACT:

The Enigma machine was a cipher machine used extensively by the military to encrypt and decrypt secret messages. Military personnel frequently employed the Enigma machine, a cipher device, to encrypt and decrypt top-secret communications. The machine has a bad reputation for being complicated. The Enigma machine would be implemented in this project using a Field-Programmable Gate Array (FPGA) architecture.

The Enigma machine will be implemented on an FPGA by creating the hardware elements required to reproduce the machine's functioning. The original Enigma machine's electrical connections and logical functions will be replicated by the FPGA through programming.

Verilog is utilized as the hardware description language (HDL) for the implementation and used the Xilinx Nexys 4 DDR Artix-7 FPGA Board.

We will benefit greatly from the opportunity to practice digital hardware design and cryptography through the implementation of the Enigma machine on an FPGA. Additionally, it will show off the adaptability of FPGA platforms and their potential for practical use.

CONTENTS

INTRODUCTION:

The German military predominantly used the Enigma machine, an electro-mechanical encryption tool, throughout World War II. It was developed by German engineer Arthur Scherbius in the early 1920s, and the German military forces made considerable use of it to send covert messages throughout the war.

Only someone with another Enigma machine and the exact same rotor settings could decipher the messages thanks to the machine's intricate system of rotors and electrical contacts. German soldiers on the front lines communicated with each other and with command centers using the portable, user-friendly Enigma machine

Allied cryptographers eventually succeeded in deciphering the Enigma code using a combination of mathematical analysis and human intellect, despite the complexity of the machine's encryption scheme. The deciphering of the Enigma code, largely regarded as one of the 20th century's greatest cryptanalysis accomplishments, was crucial to the Allies' final triumph in World War II.

FIG 1. ELECTRO-MECHANICAL ENIGMA

FPGAs provide a flexible and adaptable foundation for designing and implementing complex digital systems, such as cryptographic algorithms. By running the Enigma on an FPGA, one can examine the device's internal operations, understand its benefits and drawbacks, and discover more about modern encryption techniques.

A fantastic way to research the history of cryptography and how it has evolved through time is to place the Enigma on an FPGA. Learning about the design, programming, and debugging of digital logic may be an interesting project for students majoring in computer engineering. Running the Enigma on an FPGA like the Nexys 4 DDR Artix 7 FPGA has the general purpose of teaching and practicing digital systems design and cryptography.

WORKING

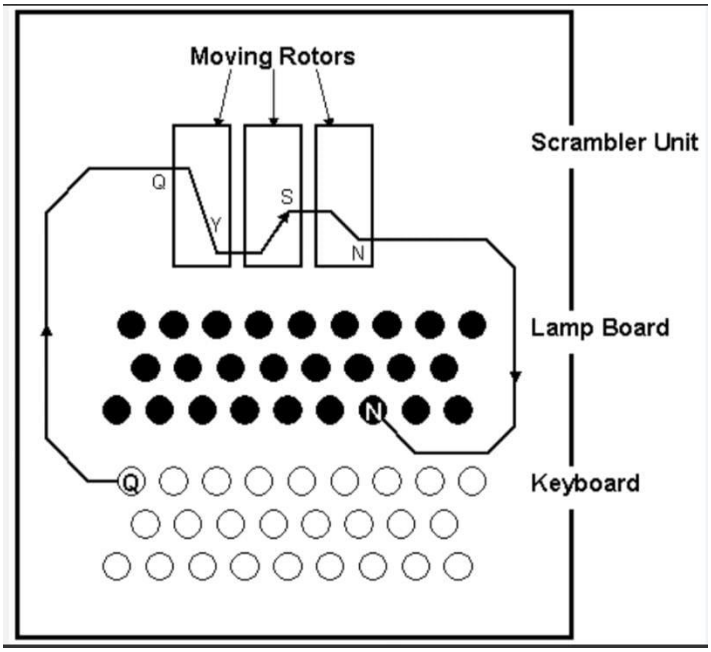


FIG 2. WORKING PRINCIPLE OF ENIGMA

Keyboard:

The keyboard consists of all 26 alphabets it is used for entering the text. When a character is entered an electrical circuit having the key pressed as switch starts to conduct the current which makes some other key(encrypted character) on the lamp board to glow.

Rotors:

There are three rotors in total which contains 26 electrical contacts on both sides representing 26 alphabets. Each input is coupled to a different contact on the back. This is done with the help of wires present inside them. The wires connected to the input are randomly assigned to outputs. This provides an electrical path for the signal from keyboard. Each rotor

have a unique connection wires inside them. The speciality of rotors that the electrical path changes every time we rotate. This enables us to obtain different encrypted outputs for the same character entered twice with different rotor settings.

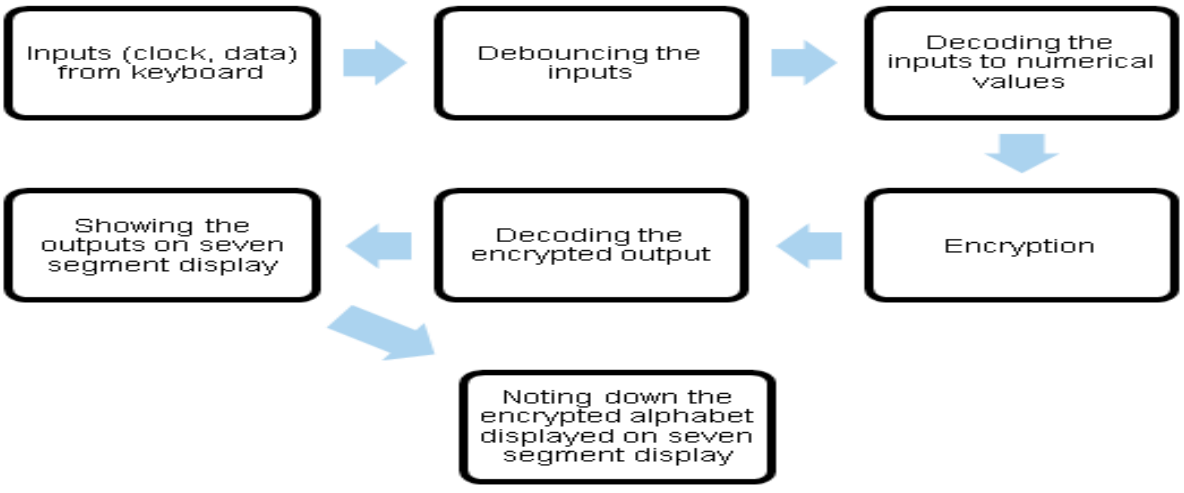
Reflector:

The reflector does the same job of rotors i.e., assigning the input to a different output by it can't be rotated so the electrical paths are fixed, it just reflects the input to a different output hence called reflector.

Lampboard:

The lamp board contains a panel of lights that indicated the encrypted output of the Enigma machine. Each time a key is pressed on the Enigma keyboard, the lamp board would display the corresponding encrypted letter. The user would then write down the encrypted letter and continue typing the message. It consists of 26 lights, one for each letter of the alphabet. The lights were arranged in two rows, with the top row displaying the letters A-M and the bottom row displaying the letters N-Z.

FLOWCHART: FIG 3. FLOW CHART OF IMPLEMENTATION



CODES:

```
`timescale 1ns / 1ps
module top(
    input         clk,
    input         USBData,
    input         USBClk,
    input [14:0] switch,
    output [6:0] segment,
    output [7:0] anode,
    output [15:0] LED
);
    reg         CLK50MHZ=0;
    reg  [15:0] keycodev=0;
    wire [15:0] keycode;
    wire        flag;
    wire        debugflag;
    reg         control=0;
always @(posedge(clk))begin
        CLK50MHZ<=~CLK50MHZ;
    end

    USBReceiver uut (
        .clk(CLK50MHZ),
        .keyclk(PS2Clk),


     .keyclk(PS2Clk),
     .keydata(USBData),
     .keycode(keycode),
     .oflag(flag),
     .myflag(debugflag)
 );


 always@(keycode)
     if (keycode[7:0] == 8'hf0) begin
         control <= 1'b0;
     end else if (keycode[15:8] == 8'hf0) begin
         control <= keycode != keycodev;
     end else begin
         control <= keycode[7:0] != keycodev[7:0] || keycodev[15:8] == 8'hf0;
     end

 always@(posedge clk)
     if (flag == 1'b1 && control == 1'b1) begin
         keycodev <= keycode;
     end
 wire [6:0] tens, ones;


     wire [4:0] character, encrypted_char;

     B2D converter(keycodev[7:0], character);
     Encrypt encr(encrypted_char, character, switch[4:0], switch[9:5], switch[14:10]);
     wire mod_clk1, mod_clk2;
     wire [31:0] reg_out;

     Alphabet decoder(tens, ones, encrypted_char);
     assign LED[7:0] = keycodev[7:0];
     // Debugging signals
     assign LED[8] = USBData;
     assign LED[9] = USBClk;
     assign LED[10] = debugflag;
     // high frequency clock
     clock_divider #(5000) cd_fast(clk, mod_clk1);

     mux2v #(8) m0(segment, ones, tens, mod_clk1);
     assign anode[7:2] = 6'b111111;
     assign anode[1] = mod_clk1;
     assign anode[0] = ~mod_clk1;

endmodule
```

```
module mux2v(out, A, B, sel);
   parameter
     width = 32;
   output [width-1:0] out;
   input  [width-1:0] A, B;
   input sel;
   wire [width-1:0] temp1 = ({width{(!sel)}} & A);
   wire [width-1:0] temp2 = ({width{(sel)}} & B);
   assign out = temp1 | temp2;
endmodule // mux2v
```

Explanation: This is top module which takes inputs form a USB receiver, and outputs on a 7-segment display. For processing the USB input data and giving out encrypted data six modules are used, The time units used in the code are determined by the timeframe directive at the beginning. The first module which takes input is USB receiver the outputs of this module is given to the converter module which converts the hexadecimal input to decimal number which can be given to encryption module. The outputs of the encryption module are given to alphabet decoder module

which will convert the output into a format which can be displayed on seven segment display.

```
module USBReceiver(
    input clk,
    input keyclk,
    input keydata,
    output reg [15:0] keycode=0,
    output reg oflag,
    output myflag
    );

    wire keyclkf, keydataf;
    reg [7:0]currentdata=0;
    reg [7:0]dataprev=0;
    reg [3:0]count=0;
    reg flag=0;

    Debounce #(
        .COUNT_MAX(19),
        .COUNT_WIDTH(5)
    ) db_clk(
        .clk(clk),
        .in(keyclk),
        .op(keyclkf)
     );
     Debounce #(
        .COUNT_MAX(19),
        .COUNT_WIDTH(5)
     ) db_data(
         .clk(clk),
         .ip(keydata),
         .op(keydataf)
     );
     assign myflag = keydataf;
     always@(negedge(keyclkf))begin
         case(count)
         0:;//Start bit
         1:currentdata[0]<=keydataf;
         2:currentdata[1]<=keydataf;
         3:currentdata[2]<=keydataf;
         4:currentdata[3]<=keydataf;
         5:currentdata[4]<=keydataf;
         6:currentdata[5]<=keydataf;
         7:currentdata[6]<=keydataf;
         8:currentdata[7]<=keydataf;


         9:flag<=1'b1;//parity bit
         10:flag<=1'b0;
         endcase
             if(count<=9) count<=count+1;
             else if(count==10) count<=0;
     end

     reg pflag;
     always@(posedge clk) begin
         if (flag == 1'b1 && pflag == 1'b0) begin
             keycode <= {dataprev, currentdata};
             oflag <= 1'b1;
             dataprev <= currentdata;
         end else
             oflag <= 'b0;
         pflag <= flag;
     end

endmodule
```

```
module Debounce(
    input clk,
    input in,
    output reg op
    );
    parameter COUNT_MAX=255, COUNT_WIDTH=8;
    reg [COUNT_WIDTH-1:0] count;
    reg inv=0;
    always@(posedge clk)
        if (in == inv) begin
            if (count == COUNT_MAX)
                op <= in;
            else
                count <= count + 1'b1;
        end else begin
            count <= 'b0;
            inv <= in;
        end

endmodule
```

Explanation: This is the USB receiver module that receives keycodes from a USB keyboard. The key clock and key data signals are debounced by the Debounce module. On the falling edge of the key clock, the module captures the key data and saves it in the current data register. Additionally, the module examines the keycode's parity bit and adjusts the flag appropriately. After a predetermined number of samples have been seen, the debounce module counts how many successive samples of the input signal are equal before producing a single output signal.

```
module B2D(ip,op);
    input [7:0] ip;
    output [4:0] op;

    wire alph [26:1];

    assign alph[1] = ip==8'h1c;
    assign alph[2] = ip==8'h32;
    assign alph[3] = ip==8'h21;
    assign alph[4] = ip==8'h23;
    assign alph[5] = ip==8'h24;
    assign alph[6] = ip==8'h2b;
    assign alph[7] = ip==8'h34;
    assign alph[8] = ip==8'h33;
    assign alph[9] = ip==8'h43;
    assign alph[10] = ip==8'h3b;
    assign alph[11] = ip==8'h42;
    assign alph[12] = ip==8'h4b;
    assign alph[13] = ip==8'h3a;
    assign alph[14] = ip==8'h31;
    assign alph[15] = ip==8'h44;
    assign alph[16] = ip==8'h4d;
```

```
    assign alph[17] = ip==8'h15;
    assign alph[18] = ip==8'h2d;
    assign alph[19] = ip==8'h1b;
    assign alph[20] = ip==8'h2c;
    assign alph[21] = ip==8'h3c;
    assign alph[22] = ip==8'h2a;
    assign alph[23] = ip==8'h1d;
    assign alph[24] = ip==8'h22;
    assign alph[25] = ip==8'h35;
    assign alph[26] = ip==8'h1a;

    assign op[0]=alph[1]|alph[3]|alph[5]|alph[7]|alph[9]|alph[11]|alph[13]|alph[15]|alph[17]|alph[19]|alph[21]|alph[23]|alph[25
    assign op[1]=alph[2]|alph[3]|alph[6]|alph[7]|alph[10]|alph[11]|alph[14]|alph[15]|alph[18]|alph[19]|alph[23]|alph[22]|alph[2
    assign op[2]=alph[4]|alph[5]|alph[6]|alph[7]|alph[12]|alph[13]|alph[14]|alph[15]|alph[20]|alph[21]|alph[22]|alph[23];
    assign op[3]=alph[8]|alph[9]|alph[10]|alph[11]|alph[12]|alph[13]|alph[14]|alph[15]|alph[24]|alph[25]|alph[26];
    assign op[4]=alph[16]|alph[17]|alph[18]|alph[19]|alph[20]|alph[21]|alph[22]|alph[23]|alph[24]|alph[25]|alph[26];

} endmodule
```

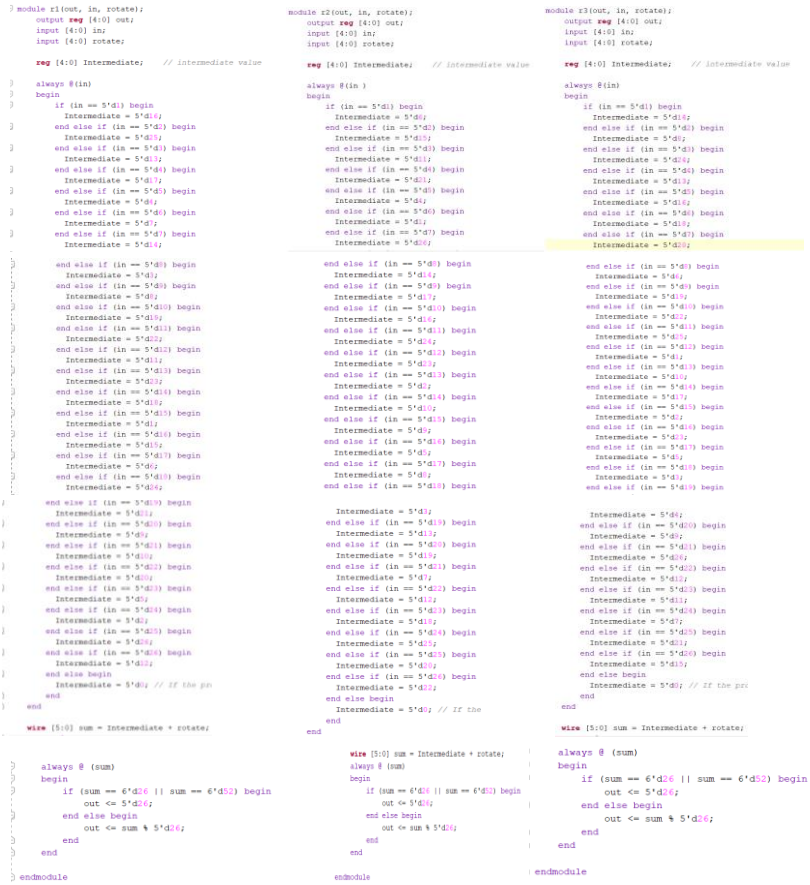
Explanation: The provided code is a Verilog module that changes an input of 8 bits to an output of 5 bits of decimal which can be given to encryption module for encryption. Each decimal digit's input is compared to a corresponding hexadecimal value using a lookup table to encode the associated decimal output.

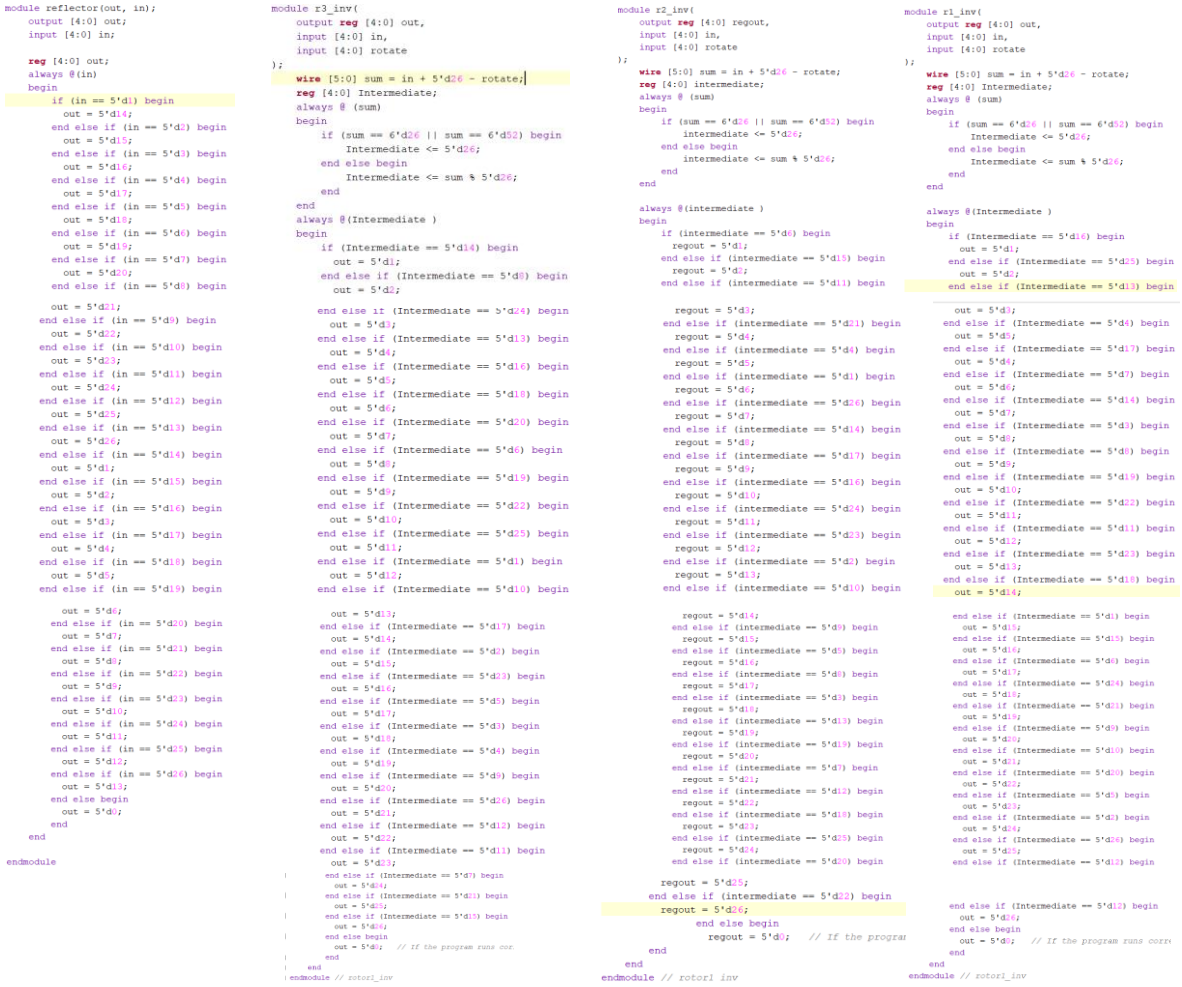
```
module Encrypt(out, in, rotate1, rotate2, rotate3);
    output [4:0] out;
    input [4:0] in;
    input [4:0] rotate1, rotate2, rotate3;

    wire [4:0] w1, w2, w3, w4, w5, w6;

    r1 ro1(w1, in, rotate1);
    r2 ro2(w2, w1, rotate2);
    r3 ro3(w3, w2, rotate3);
    reflector ro4(w4, w3);
    r3_inv ro5(w5, w4, rotate3);
    r2_inv ro6(w6, w5, rotate2);
    r1_inv ro7(out, w6, rotate1);

endmodule // encryption
```





```
module reflector(out, in);
    output [4:0] out;
    input [4:0] in;

    reg [4:0] out;
    always @(in)
    begin
        if (in == 5'd1) begin
          out = 5'd14;
        end else if (in == 5'd2) begin
          out = 5'd15;
        end else if (in == 5'd3) begin
          out = 5'd16;
        end else if (in == 5'd4) begin
          out = 5'd17;
        end else if (in == 5'd5) begin
          out = 5'd18;
        end else if (in == 5'd6) begin
          out = 5'd19;
        end else if (in == 5'd7) begin
          out = 5'd20;
        end else if (in == 5'd8) begin

        out = 5'd21;
      end else if (in == 5'd9) begin
        out = 5'd22;
      end else if (in == 5'd10) begin
        out = 5'd23;
      end else if (in == 5'd11) begin
        out = 5'd24;
      end else if (in == 5'd12) begin
        out = 5'd25;
      end else if (in == 5'd13) begin
        out = 5'd26;
      end else if (in == 5'd14) begin
        out = 5'd1;
      end else if (in == 5'd15) begin
        out = 5'd2;
      end else if (in == 5'd16) begin
        out = 5'd3;
      end else if (in == 5'd17) begin
        out = 5'd4;
      end else if (in == 5'd18) begin
        out = 5'd5;
      end else if (in == 5'd19) begin

          out = 5'd6;
        end else if (in == 5'd20) begin
          out = 5'd7;
        end else if (in == 5'd21) begin
          out = 5'd8;
        end else if (in == 5'd22) begin
          out = 5'd9;
        end else if (in == 5'd23) begin
          out = 5'd10;
        end else if (in == 5'd24) begin
          out = 5'd11;
        end else if (in == 5'd25) begin
          out = 5'd12;
        end else if (in == 5'd26) begin
          out = 5'd13;
        end else begin
          out = 5'd0;
        end
    end

endmodule
```

```
module r3_inv(
    output reg [4:0] out,
    input [4:0] in,
    input [4:0] rotate
);
    wire [5:0] sum = in + 5'd26 - rotate;
    reg [4:0] Intermediate;
    always @ (sum)
    begin
        if (sum == 6'd26 || sum == 6'd52) begin
            Intermediate <= 5'd26;
        end else begin
            Intermediate <= sum % 5'd26;
        end
    end
    always @(Intermediate )
    begin
        if (Intermediate == 5'd14) begin
          out = 5'd1;
        end else if (Intermediate == 5'd8) begin
          out = 5'd2;

        end else if (Intermediate == 5'd24) begin
          out = 5'd3;
        end else if (Intermediate == 5'd13) begin
          out = 5'd4;
        end else if (Intermediate == 5'd16) begin
          out = 5'd5;
        end else if (Intermediate == 5'd18) begin
          out = 5'd6;
        end else if (Intermediate == 5'd20) begin
          out = 5'd7;
        end else if (Intermediate == 5'd6) begin
          out = 5'd8;
        end else if (Intermediate == 5'd19) begin
          out = 5'd9;
        end else if (Intermediate == 5'd22) begin
          out = 5'd10;
        end else if (Intermediate == 5'd25) begin
          out = 5'd11;
        end else if (Intermediate == 5'd1) begin
          out = 5'd12;
        end else if (Intermediate == 5'd10) begin

          out = 5'd13;
        end else if (Intermediate == 5'd17) begin
          out = 5'd14;
        end else if (Intermediate == 5'd2) begin
          out = 5'd15;
        end else if (Intermediate == 5'd23) begin
          out = 5'd16;
        end else if (Intermediate == 5'd5) begin
          out = 5'd17;
        end else if (Intermediate == 5'd3) begin
          out = 5'd18;
        end else if (Intermediate == 5'd4) begin
          out = 5'd19;
        end else if (Intermediate == 5'd9) begin
          out = 5'd20;
        end else if (Intermediate == 5'd26) begin
          out = 5'd21;
        end else if (Intermediate == 5'd12) begin
          out = 5'd22;
        end else if (Intermediate == 5'd11) begin
          out = 5'd23;
        end else if (Intermediate == 5'd7) begin
          out = 5'd24;
        end else if (Intermediate == 5'd21) begin
          out = 5'd25;
        end else if (Intermediate == 5'd15) begin
          out = 5'd26;
        end else begin
          out = 5'd0;   // If the program runs cor
        end
    end
endmodule // rotor1_inv
```

```
module r2_inv(
    output reg [4:0] regout,
    input [4:0] in,
    input [4:0] rotate
);
    wire [5:0] sum = in + 5'd26 - rotate;
    reg [4:0] intermediate;
    always @ (sum)
    begin
        if (sum == 6'd26 || sum == 6'd52) begin
            intermediate <= 5'd26;
        end else begin
            intermediate <= sum % 5'd26;
        end
    end

    always @(intermediate )
    begin
        if (intermediate == 5'd6) begin
          regout = 5'd1;
        end else if (intermediate == 5'd15) begin
          regout = 5'd2;
        end else if (intermediate == 5'd11) begin

          regout = 5'd3;
        end else if (intermediate == 5'd21) begin
          regout = 5'd4;
        end else if (intermediate == 5'd4) begin
          regout = 5'd5;
        end else if (intermediate == 5'd1) begin
          regout = 5'd6;
        end else if (intermediate == 5'd26) begin
          regout = 5'd7;
        end else if (intermediate == 5'd14) begin
          regout = 5'd8;
        end else if (intermediate == 5'd17) begin
          regout = 5'd9;
        end else if (intermediate == 5'd16) begin
          regout = 5'd10;
        end else if (intermediate == 5'd24) begin
          regout = 5'd11;
        end else if (intermediate == 5'd23) begin
          regout = 5'd12;
        end else if (intermediate == 5'd2) begin
          regout = 5'd13;
        end else if (intermediate == 5'd10) begin

          regout = 5'd14;
        end else if (intermediate == 5'd9) begin
          regout = 5'd15;
        end else if (intermediate == 5'd5) begin
          regout = 5'd16;
        end else if (intermediate == 5'd8) begin
          regout = 5'd17;
        end else if (intermediate == 5'd3) begin
          regout = 5'd18;
        end else if (intermediate == 5'd13) begin
          regout = 5'd19;
        end else if (intermediate == 5'd19) begin
          regout = 5'd20;
        end else if (intermediate == 5'd7) begin
          regout = 5'd21;
        end else if (intermediate == 5'd12) begin
          regout = 5'd22;
        end else if (intermediate == 5'd18) begin
          regout = 5'd23;
        end else if (intermediate == 5'd25) begin
          regout = 5'd24;
        end else if (intermediate == 5'd20) begin

          regout = 5'd25;
        end else if (intermediate == 5'd22) begin
          regout = 5'd26;
              end else begin
                regout = 5'd0;   // If the prograa
        end
    end
endmodule // rotor1 inv
```

```
module r1_inv(
    output reg [4:0] out,
    input [4:0] in,
    input [4:0] rotate
);
    wire [5:0] sum = in + 5'd26 - rotate;
    reg [4:0] Intermediate;
    always @ (sum)
    begin
        if (sum == 6'd26 || sum == 6'd52) begin
            Intermediate <= 5'd26;
        end else begin
            Intermediate <= sum % 5'd26;
        end
    end

    always @(Intermediate )
    begin
        if (Intermediate == 5'd16) begin
          out = 5'd1;
        end else if (Intermediate == 5'd25) begin
          out = 5'd2;
        end else if (Intermediate == 5'd13) begin

          out = 5'd3;
        end else if (Intermediate == 5'd6) begin
          out = 5'd4;
        end else if (Intermediate == 5'd17) begin
          out = 5'd5;
        end else if (Intermediate == 5'd7) begin
          out = 5'd6;
        end else if (Intermediate == 5'd14) begin
          out = 5'd7;
        end else if (Intermediate == 5'd3) begin
          out = 5'd8;
        end else if (Intermediate == 5'd8) begin
          out = 5'd9;
        end else if (Intermediate == 5'd19) begin
          out = 5'd10;
        end else if (Intermediate == 5'd22) begin
          out = 5'd11;
        end else if (Intermediate == 5'd11) begin
          out = 5'd12;
        end else if (Intermediate == 5'd23) begin
          out = 5'd13;
        end else if (Intermediate == 5'd18) begin
          out = 5'd14;

        end else if (Intermediate == 5'd1) begin
          out = 5'd15;
        end else if (Intermediate == 5'd15) begin
          out = 5'd16;
        end else if (Intermediate == 5'd6) begin
          out = 5'd17;
        end else if (Intermediate == 5'd24) begin
          out = 5'd18;
        end else if (Intermediate == 5'd21) begin
          out = 5'd19;
        end else if (Intermediate == 5'd9) begin
          out = 5'd20;
        end else if (Intermediate == 5'd10) begin
          out = 5'd21;
        end else if (Intermediate == 5'd20) begin
          out = 5'd22;
        end else if (Intermediate == 5'd5) begin
          out = 5'd23;
        end else if (Intermediate == 5'd2) begin
          out = 5'd24;
        end else if (Intermediate == 5'd26) begin
          out = 5'd25;
        end else if (Intermediate == 5'd12) begin


        end else if (Intermediate == 5'd12) begin
          out = 5'd26;
        end else begin
          out = 5'd0;   // If the program runs corre
        end
    end
endmodule // rotor1_inv
```

Explanation: This are the rotor modules present in the encryption module The top-level module "Encrypt" creates an output "out" from an input "in" and three rotation values ("rotate1", "rotate2", and "rotate3"). The "Encrypt" module encrypts data using the "r1", "r2", "r3", "reflector", "r3_inv", and "r2_inv" modules, which are six additional modules.

```
module Alphabet(seg0,seg1,in);
    input [4:0] in;
    output [6:0] seg0,seg1;
    wire dec0,dec1,dec2;
    wire w1,w2,w3,w4,w5,w6,w7,w8,w9,w0;

    assign dec0= in[4:3]==2'b00|in==5'd9|in==5'd8;
    assign dec2= in==5'd20|in==5'd21|in==5'd22|in==5'd23|in==5'd24|in==5'd25|in==5'd26;
    assign dec1= in==5'd10|in==5'd11|in[4:2]==3'b011|in[4:2]==3'b100|in==5'd19;

    assign w0=in==5'd0|in==5'd10|in==5'd20;
    assign w1=in==5'd1|in==5'd11|in==5'd21;
    assign w2=in==5'd2|in==5'd12|in==5'd22;
    assign w3=in==5'd3|in==5'd13|in==5'd23;
    assign w4=in==5'd4|in==5'd14|in==5'd24;
    assign w5=in==5'd5|in==5'd15|in==5'd25;
    assign w6=in==5'd6|in==5'd16|in==5'd26;
    assign w7=in==5'd7|in==5'd17;
    assign w8=in==5'd8|in==5'd18;
    assign w9=in==5'd9|in==5'd19;

    assign seg0[0]=w1|w4;
    assign seg0[1]=w5|w6;
    assign seg0[2]=w2;
    assign seg0[3]=w1|w4|w7;
    assign seg0[4]=w1|w3|w4|w5|w7|w9;
    assign seg0[5]=w1|w2|w3|w7;
    assign seg0[6]=w0|w1|w7;
    assign seg1[0]=dec1;
    assign seg1[1]=0;
    assign seg1[2]=dec2;
    assign seg1[3]=dec1;
    assign seg1[4]=dec1;
    assign seg1[5]=dec1|dec2;
    assign seg1[6]=dec0|dec1;
endmodule
```

Explanation: This module which has two seg0 and seg1 output signals and accepts a 5-bit input signal, is defined to decode the input decimal number .Based on the value of the input in, the module employs conditional assignments to determine the values of intermediate signals dec0, dec1, dec2, and w0 through w9.

The intermediate signals w0 through w9, dec0, dec1, and dec2 are utilized to calculate seg0 values. The output seg0 represents a seven-segment display that displays alphanumeric letters, while seg1 is used to display a decimal point or other supplementary information.

```
module clock_divider (input clkin, output clkout);
    parameter
        period = 5000;
    reg [31:0] counter = 1;
    reg temp_clk = 0;
    always @ (posedge(clkin)) begin
        if (counter == period)
        begin
            counter <= 1;
            temp_clk <= ~temp_clk;
        end
        else
            counter <= counter + 1;
    end
    assign clkout = temp_clk;
endmodule
```

Explanation: clock divider circuit that converts an input clock signal ("clkin") into an output clock signal ("clkout") with a specified period after reaching the predetermined interval, the circuit toggles the output clock signal by using a counter to count the amount of clock cycles. Using an always block with a posedge(clkin) sensitivity list and an assign statement to attach the toggling output signal to the "clkout" output port.

SIMULATIONS:

Waveform-

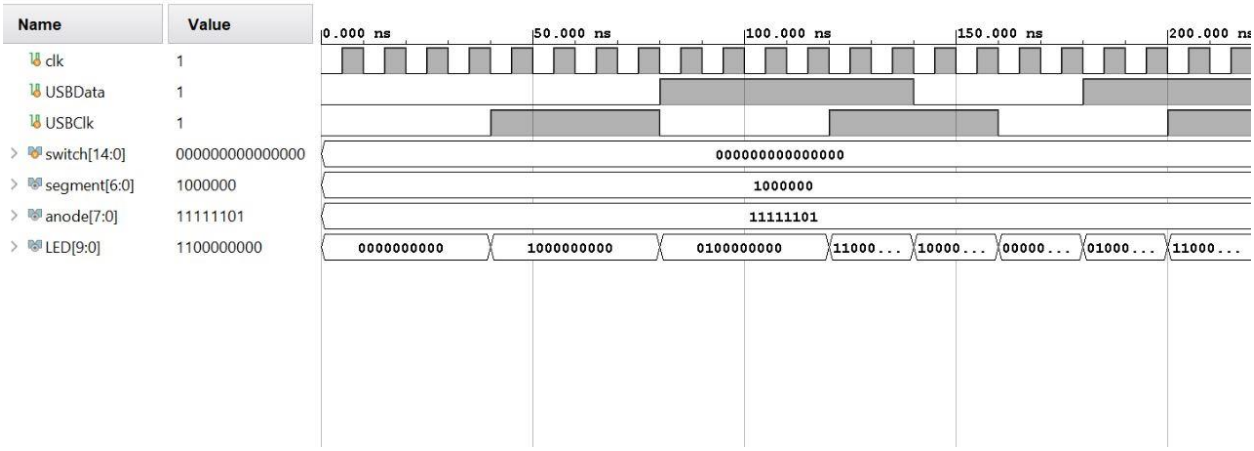
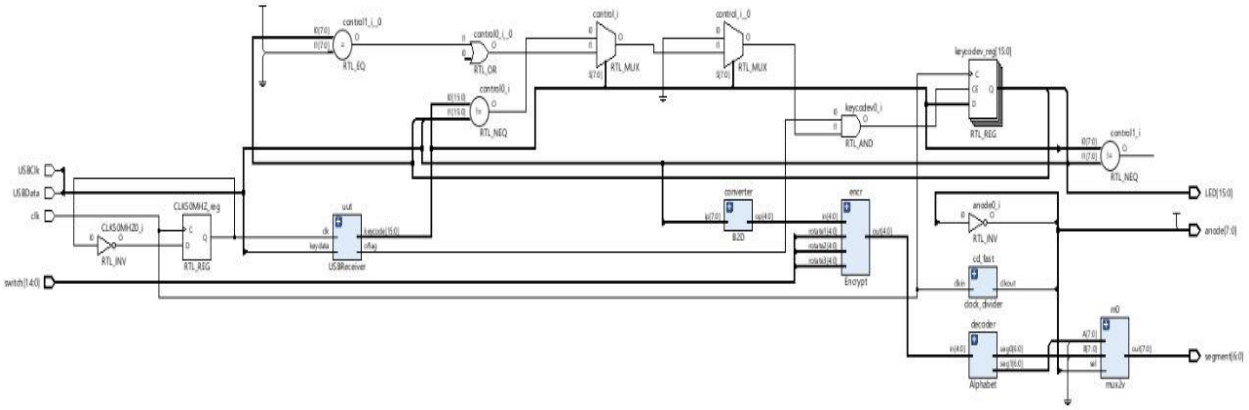


FIG 4. OUTPUT WAVEFORM

RTL Schematic-

FIG 5. TOP MODULE SCHEMATIC



USB Receiver:

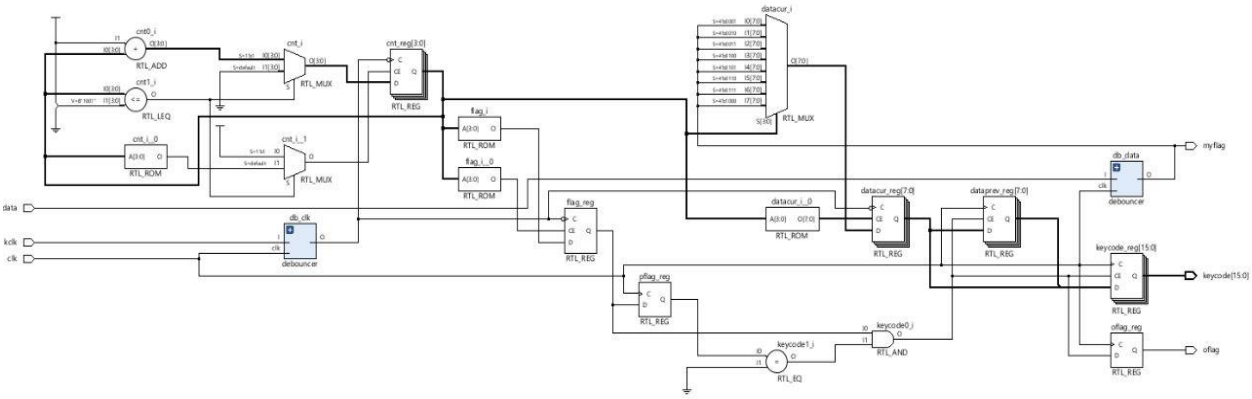
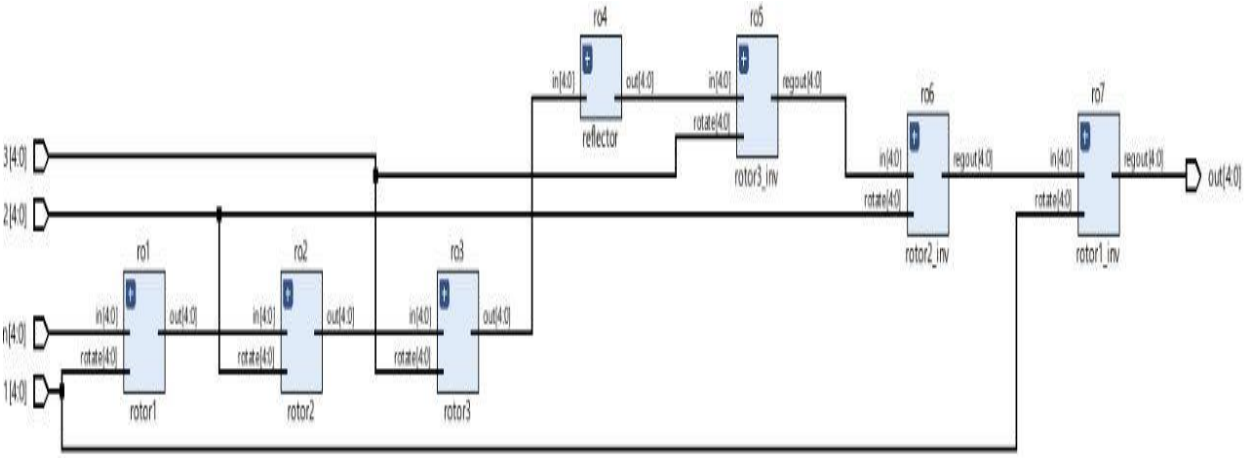


FIG 6. USB RECEIVER SCHEMATIC

Encryption:

FIG 7. ENCRYPTION SCHEMATIC



Debouncer:

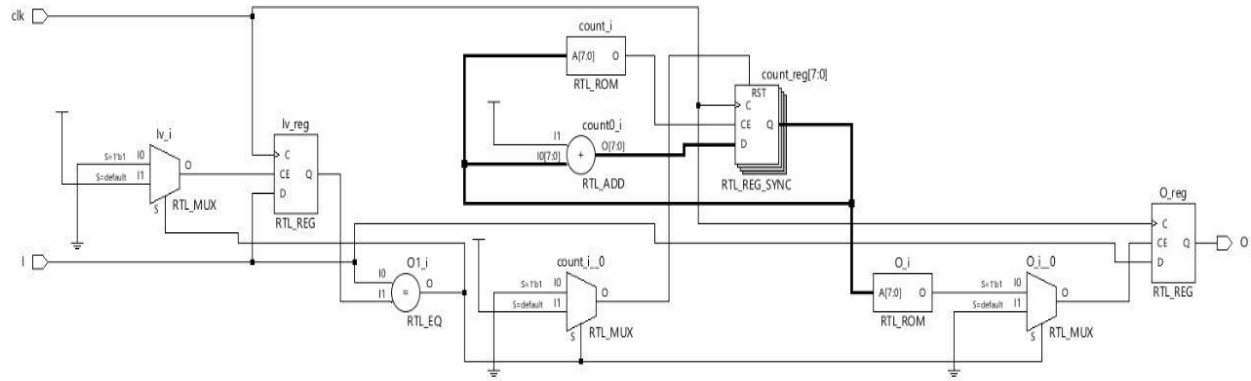


FIG 8. DEBOUNCER SCHEMATIC

Alphabet Decoder:

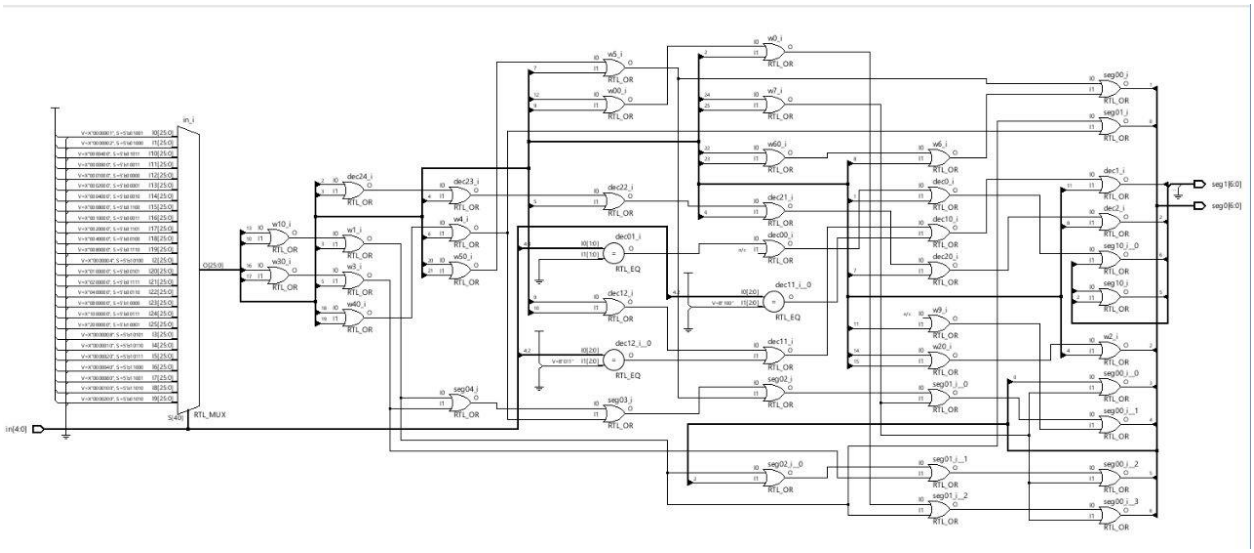
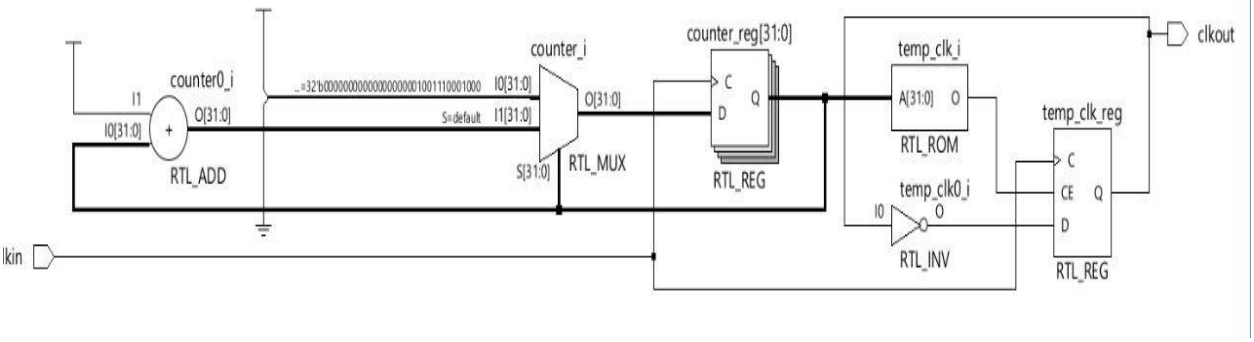


FIG 9. ALPHABET DECODER SCHEMATIC

Clock Divider:

FIG 10. CLOCK DIVIDER SCHEMATIC



Nexys 4 DDR (XC7A100TCSG324-1) FPGA Implementation:

Entered A with rotor 1 used:

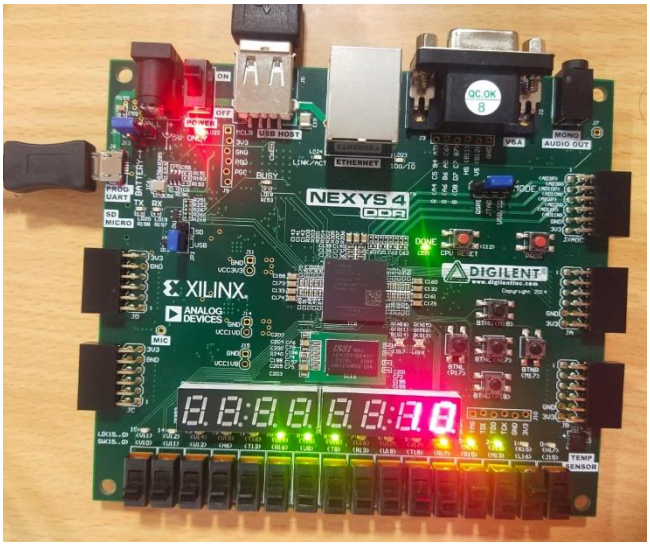


FIG 11. OUTPUT EXAMPLE 1A

Entered J(10) to get A(01) with same rotor setting:

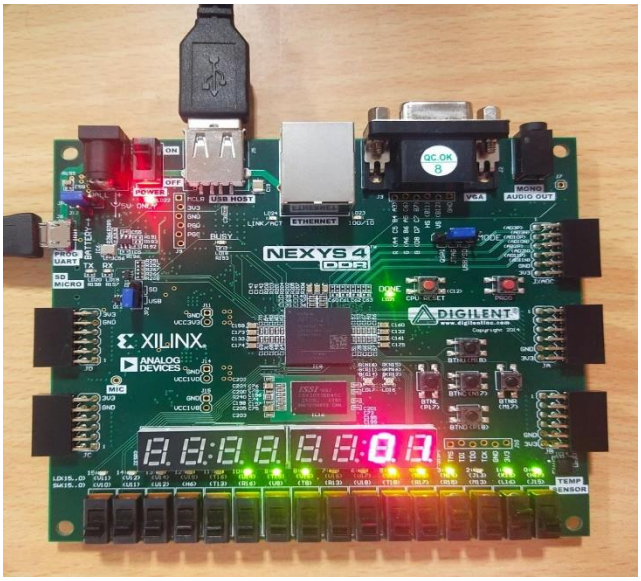


FIG 12. OUTPUT EXAMPLE 1B

Entered A with rotor 1,2 used :

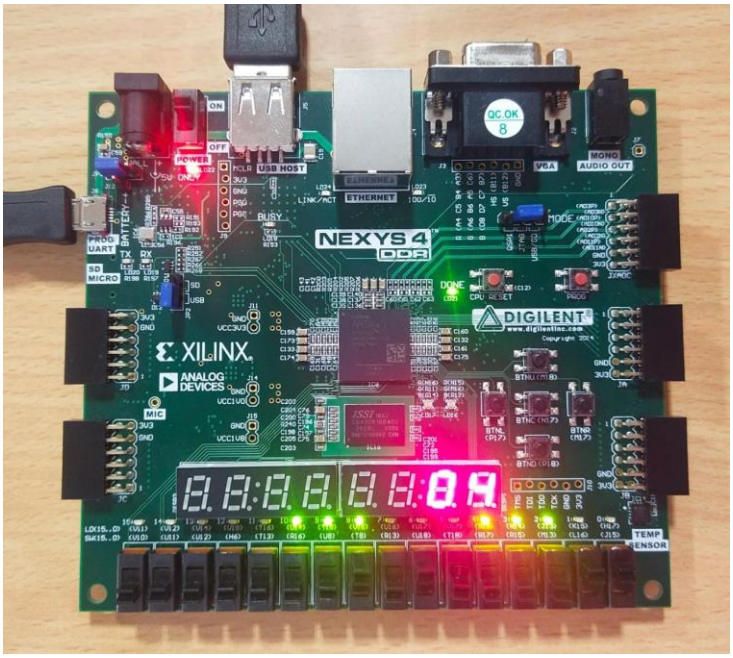


FIG 13. OUTPUT EXAMPLE 2A

Entered D(4) to get A(01) with same setting:

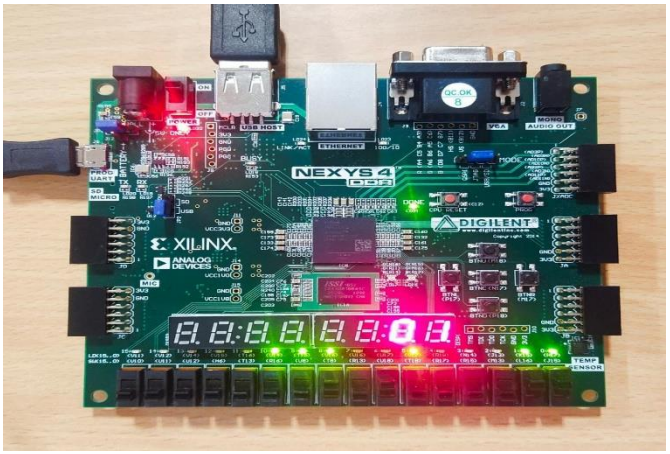


FIG 14. OUTPUT EXAMPLE 2B

Entered A with rotors 1,2 & 3 used:

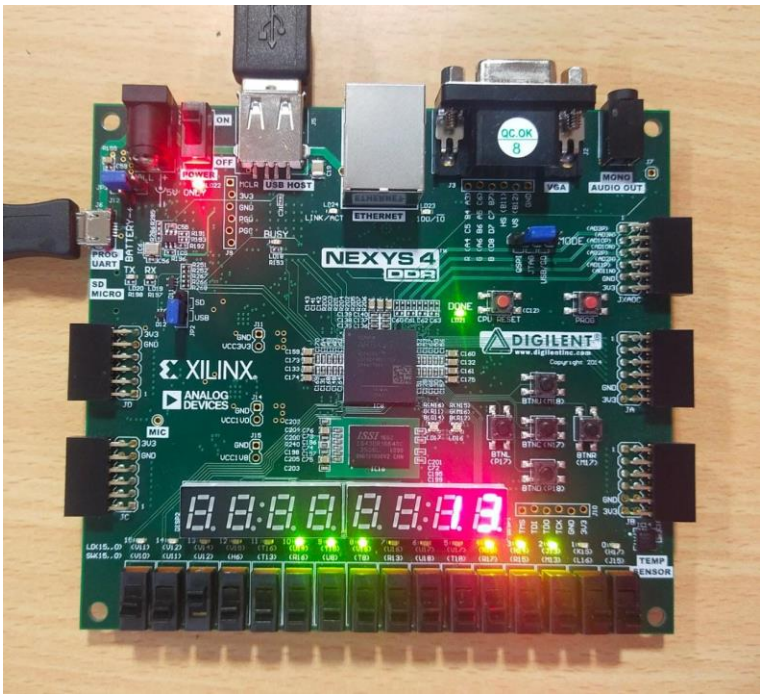


FIG 15. OUTPUT EXAMPLE 3A

Entered M(13) to get A(01) with same setting of rotors 1,2 & 3:

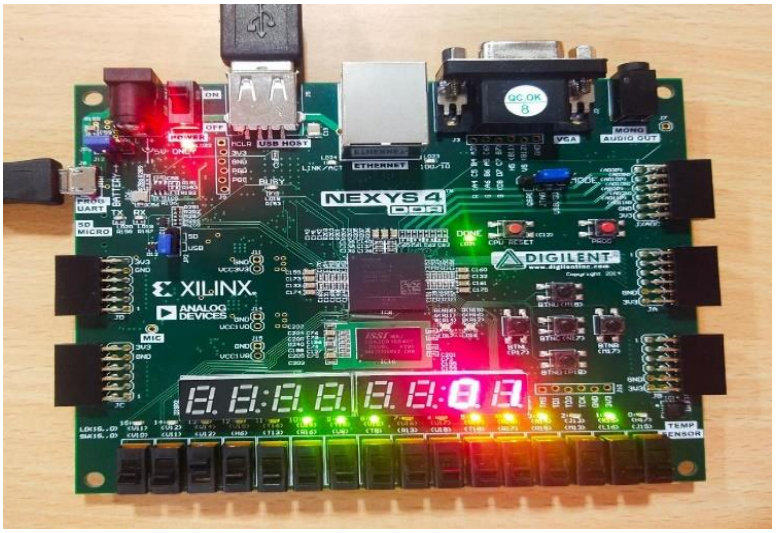


FIG 16. OUTPUT EXAMPLE 3B

APPLICATION:

The FPGA implementation of the Enigma machine has numerous potential applications. One is in the field of cybersecurity, where the machine could be used to develop and test new encryption algorithms or to analyze the effectiveness of existing ones.

Another application is in education, where the FPGA implementation could be used as a teaching tool to help students understand the principles of cryptography and computer engineering. Finally, the FPGA implementation could also be used as a hobby project for enthusiasts interested in recreating historical technology.

CONCLUSION:

The FPGA implementation of the Enigma machine represents an exciting intersection of historical technology and modern computing. While there are challenges involved in accurately emulating the behavior of the original machine and ensuring security, the potential applications of this technology are numerous and varied.

Whether used in cybersecurity research, education, or as a hobby project, the FPGA implementation of the Enigma machine offers a fascinating glimpse into the history and future of cryptography and computing.

REFERENCES:

- https://ieeexplore.ieee.org/document/7394608
- https://www.pantechsolutions.net/ps-2-usb-keyboard-interface-with-fpga
- https://www.101computing.net/enigma-machine-emulator/