BERKE KUZEY ARDIÇ

22103340

SECTION-004

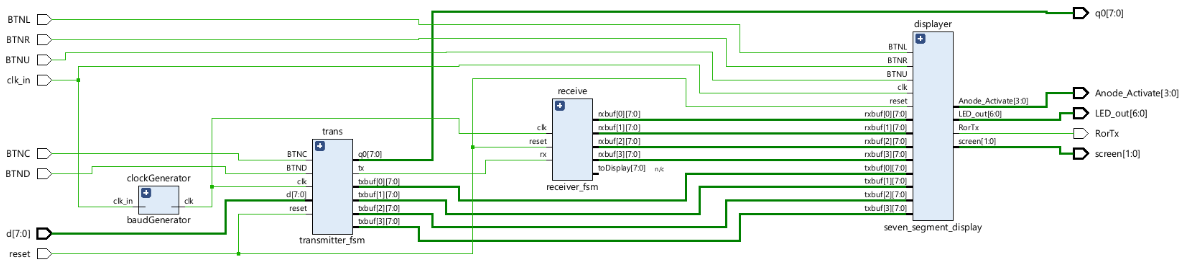
Detailed Explanation Of Design

In my implementation I first sketch the HLSM logic of the transmitter and receiver for the stage 1 of the project. Then I detailed the FSM and the datapath to recognize the logic and began to code. My transmitter FSM consisted of 8 states later in stage 3 ı made it 9 these were two stop states that send stop bits, an init state, a wait state for button d, a load state, another wait for button c , counter and register clear state and counting and shifting states. The Transmitter logic was initializing a register, shift register and counter first then loading the data into the register and by using the shift register and counting up to the 8 sending data bit to bit to the receiver module.Meanwhile ı was sending the necessary tx with a 4 to 1 mux to indicate receiver the process. The receiver FSM consisted of 5 states, logic was waiting for rx to be 0 and when rx is 0 take the data into a shift register and in the last stage load the data into the output register. In the coding stage first ı designed the side modules that ı will use into the main logic like eight-bit register, shift register, multiplexers, counter, baud rate generator. Then ı implement two main modules: transmitter_fsm and receiver_fsm. In transmitter_fsm it creates a 32 bit register that consists of 4 eight-bit data called txbuf a register called t register the handle the load to the shift register which is connected to the least significant 8 bits of the txbuf register. In state 0 I reset all the registers and counters. In State1 ı wait for the button D and if button d is pressed ı move on to the nextstate which is S2. In S2 after button D I load the data into the txbuf register of the transmitter module by taking the data into the most significant eight-bits and I shift the and move on to the nextstate to wait for the Button C. In S3 ı wait for the Button C to be pressed and if button C is pressed I move on to the preparation state for the shifting stage then ı shift the data and send it with the tx. In the receiver logic is very simple I have an another array which will be used for the storing the send data when tx is 0 for the first time it moves to the receive state takes the data bit to bit into the shift register than connects it to the another eight-bit register and this eightbit register load the data to the most significant bits of the 32 bit register and while loading it shifts the register data to display the oldest 4 in a fifo structure.

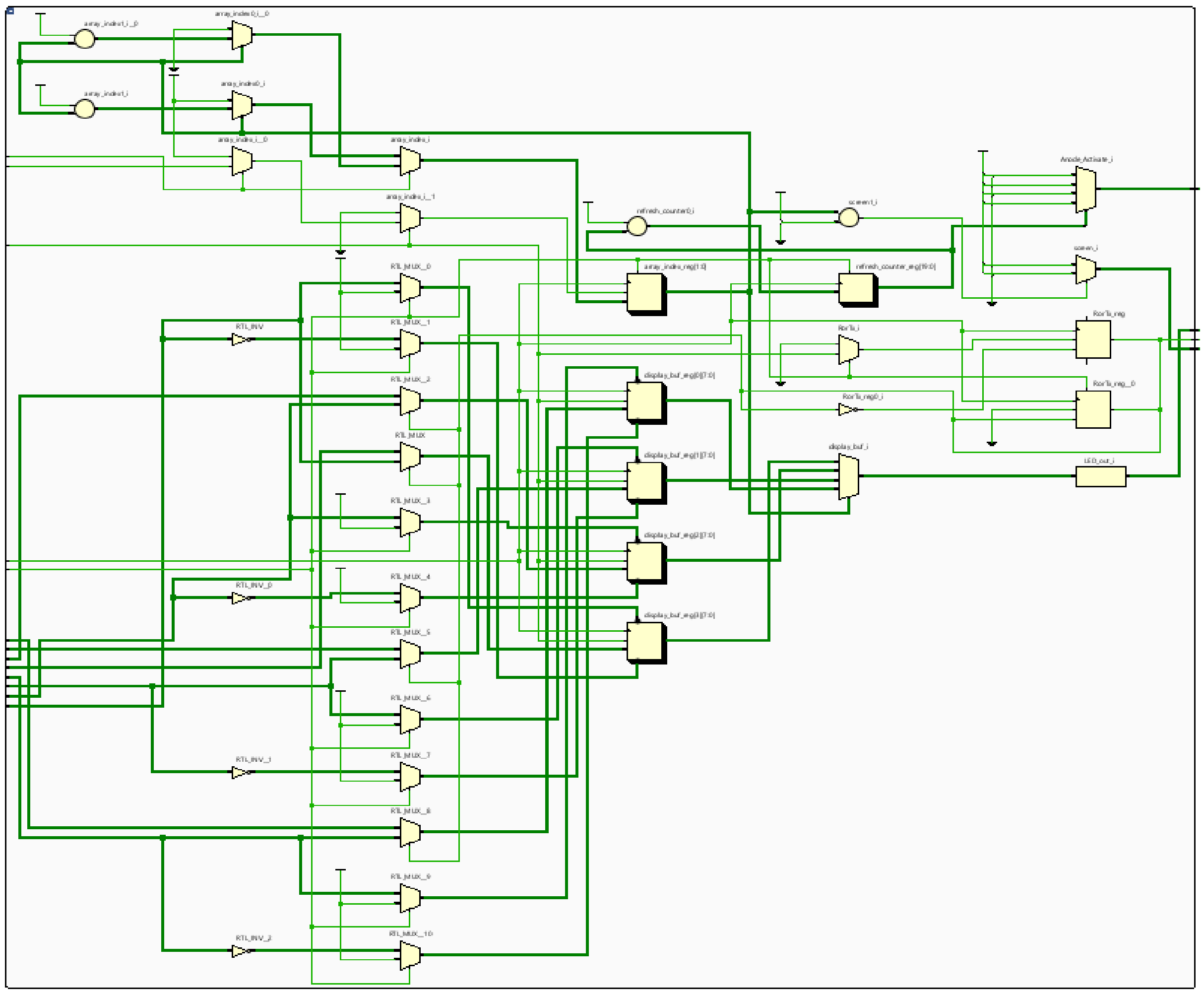
In an another module called seven segment display I handle the seven segment display logic I take the rxbuf and the txbuf arrays as inputs for this module and create a 32 bit register and load it with rxbuf or txbuf by the input of button U and display the necessary bits regarding the button left and right. If the input is positive RorTx output in the module indicates and it also acts a decessor to understand which to display ı handle this logic by simple if cases and ı designed a digit for each anothe and kathode values to show the necessary digits.

RTL SCHEMATİCS

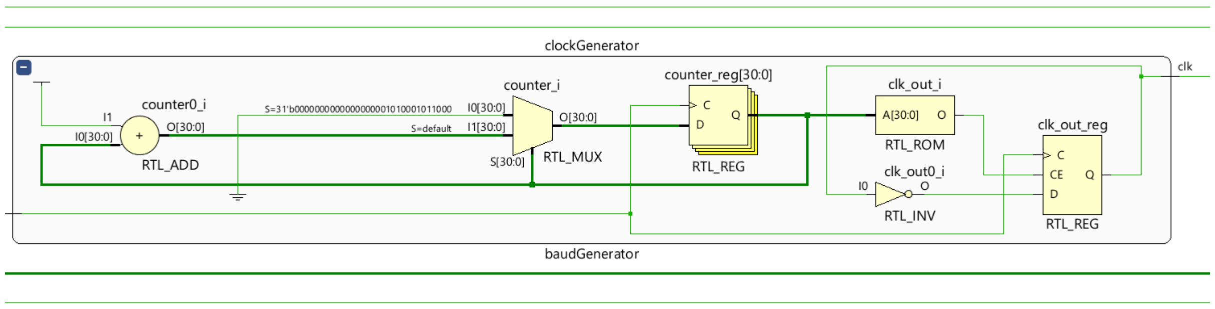
UART



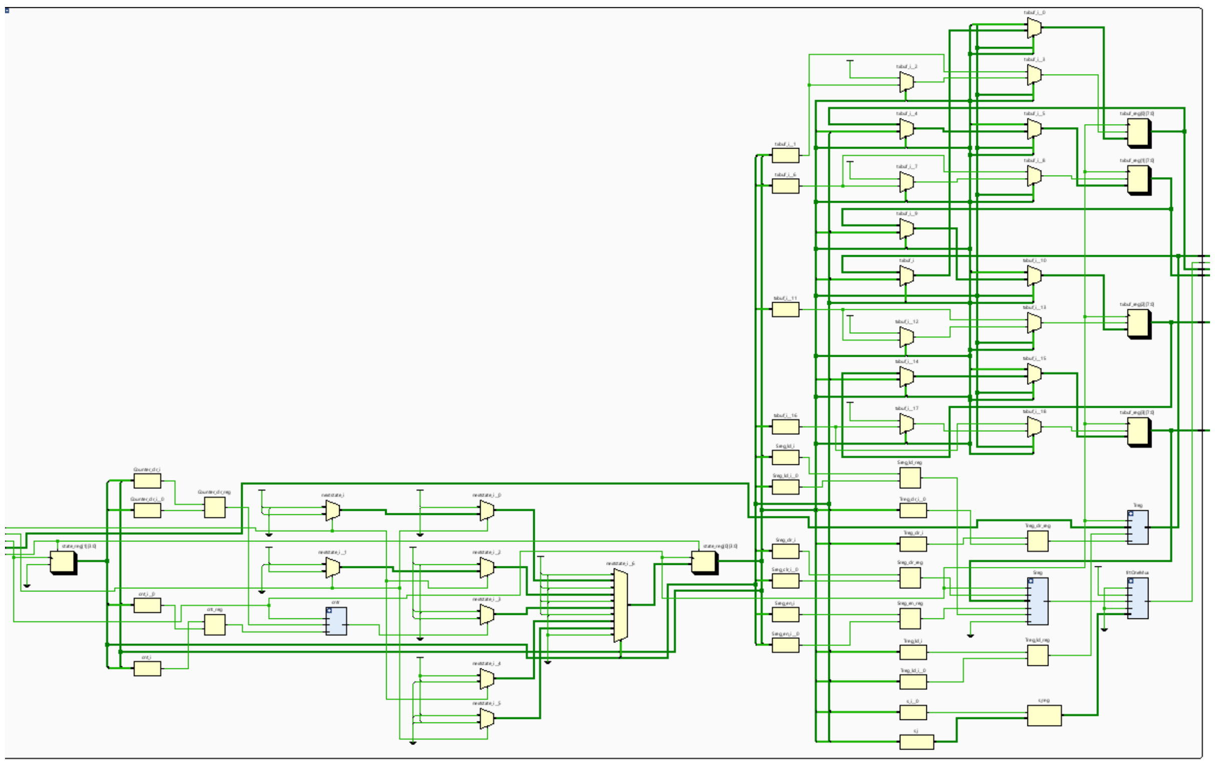
Seven Segment Displayer



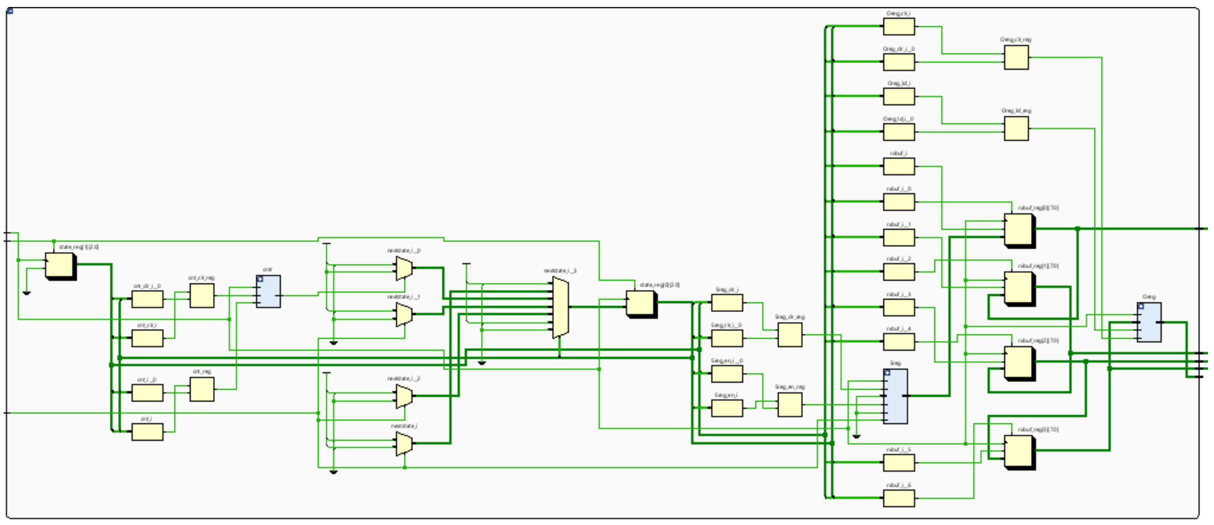
Clock Generator



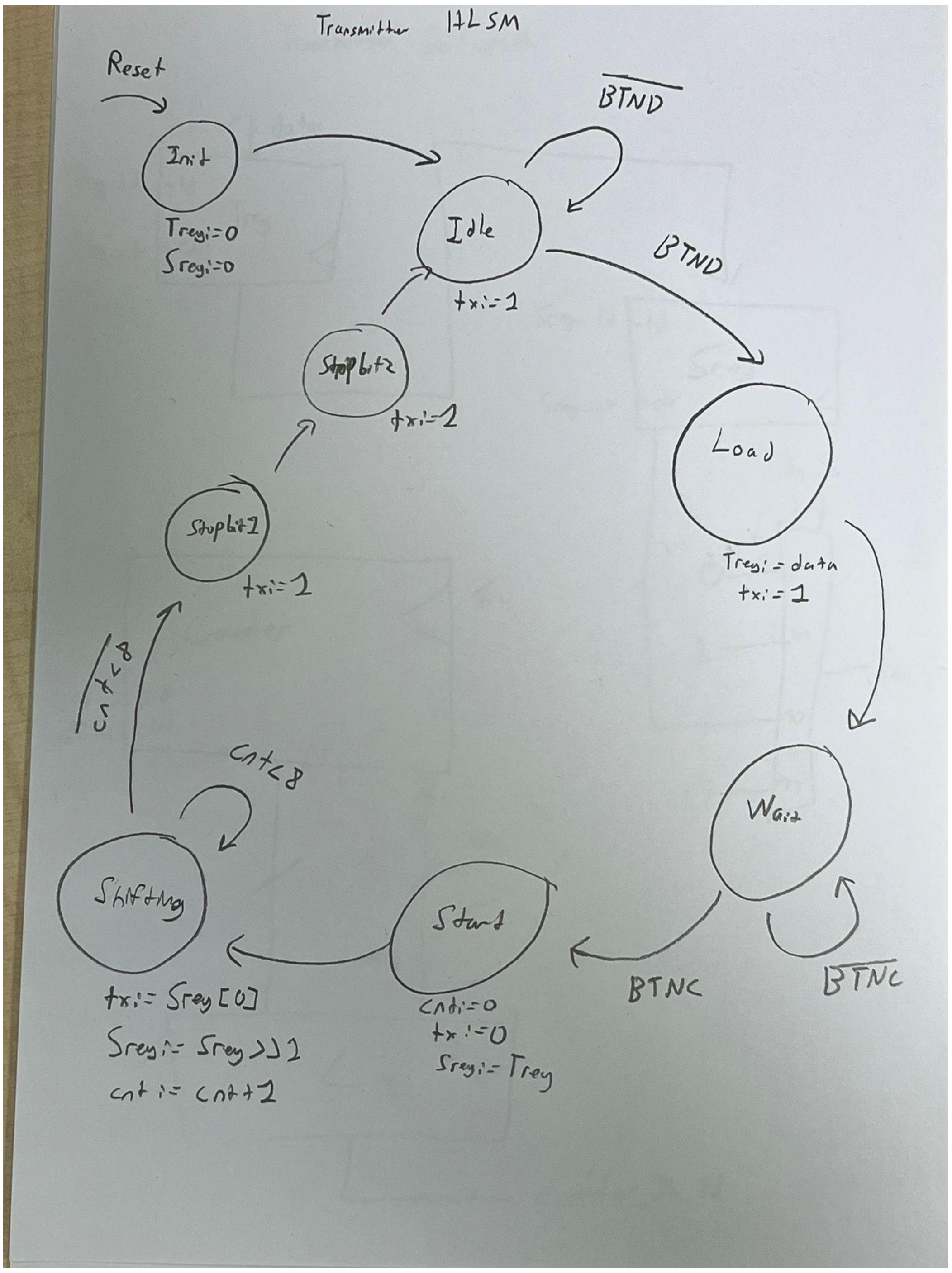
Transmitter



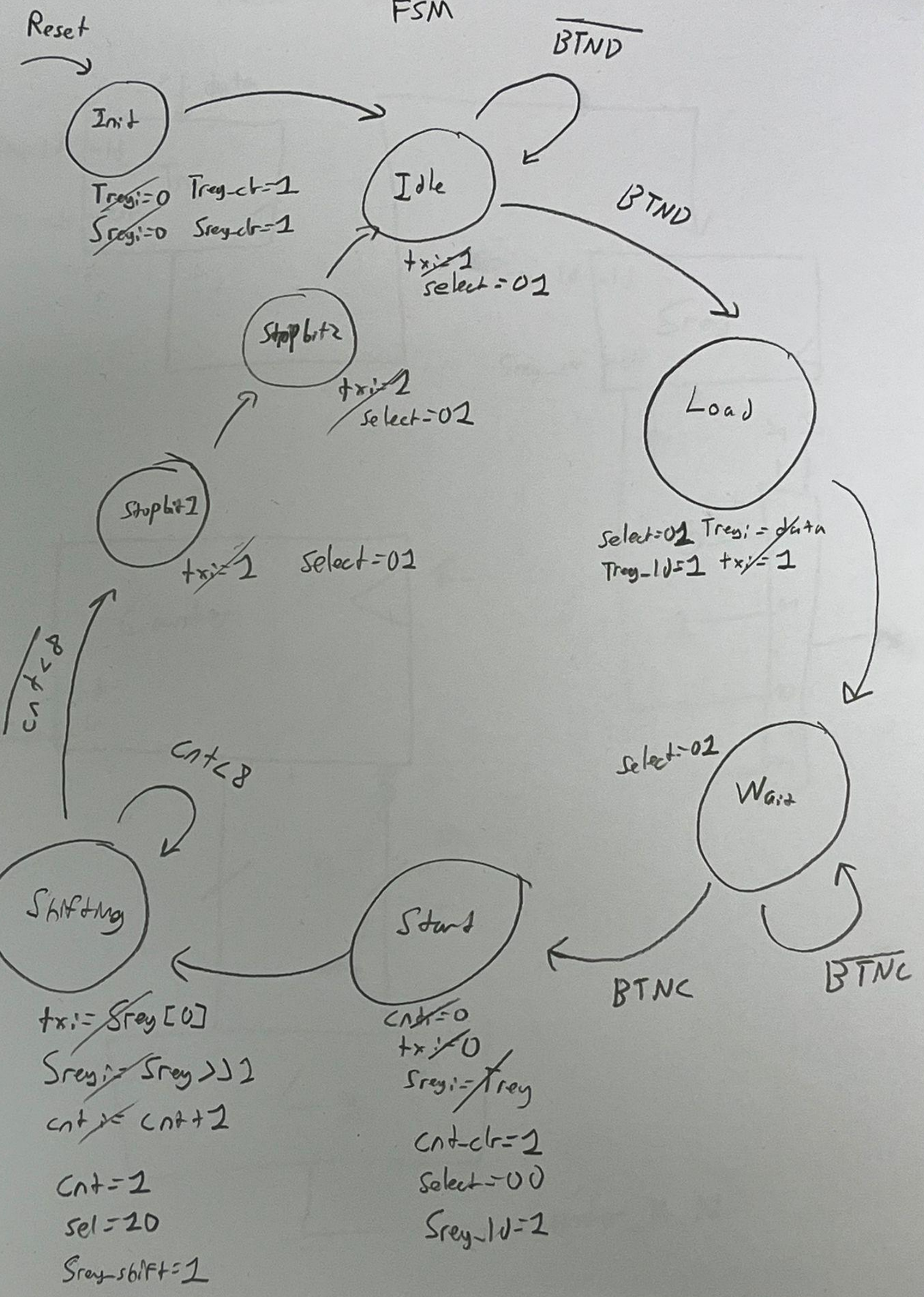
Receiver


state_reg[1][2:0]
cnt_clr_i__0
cnt_clk_reg
cnt
cnt_clk_i
cnt_i__0
cnt_reg
cnt_i
nextstate_i__0
nextstate_i__1
nextstate_i__2
nextstate_i
nextstate_i__3
state_reg[0][2:0]
Sreg_clr_i
Sreg_clr_reg
Sreg_clk_i__0
Sreg_en_i__0
Sreg_en_reg
Sreg_en_i
Oreg_clk_i
Oreg_clk_reg
Oreg_clr_i__0
Oreg_ld_i
Oreg_ld_reg
Oreg_ld_i__0
rxbuf_i
rxbuf_i__0
rxbuf_reg[0][7:0]
rxbuf_i__1
rxbuf_i__2
rxbuf_reg[1][7:0]
rxbuf_i__3
rxbuf_i__4
Oreg
rxbuf_reg[2][7:0]
Sreg
rxbuf_reg[3][7:0]
rxbuf_i__5
rxbuf_i__6

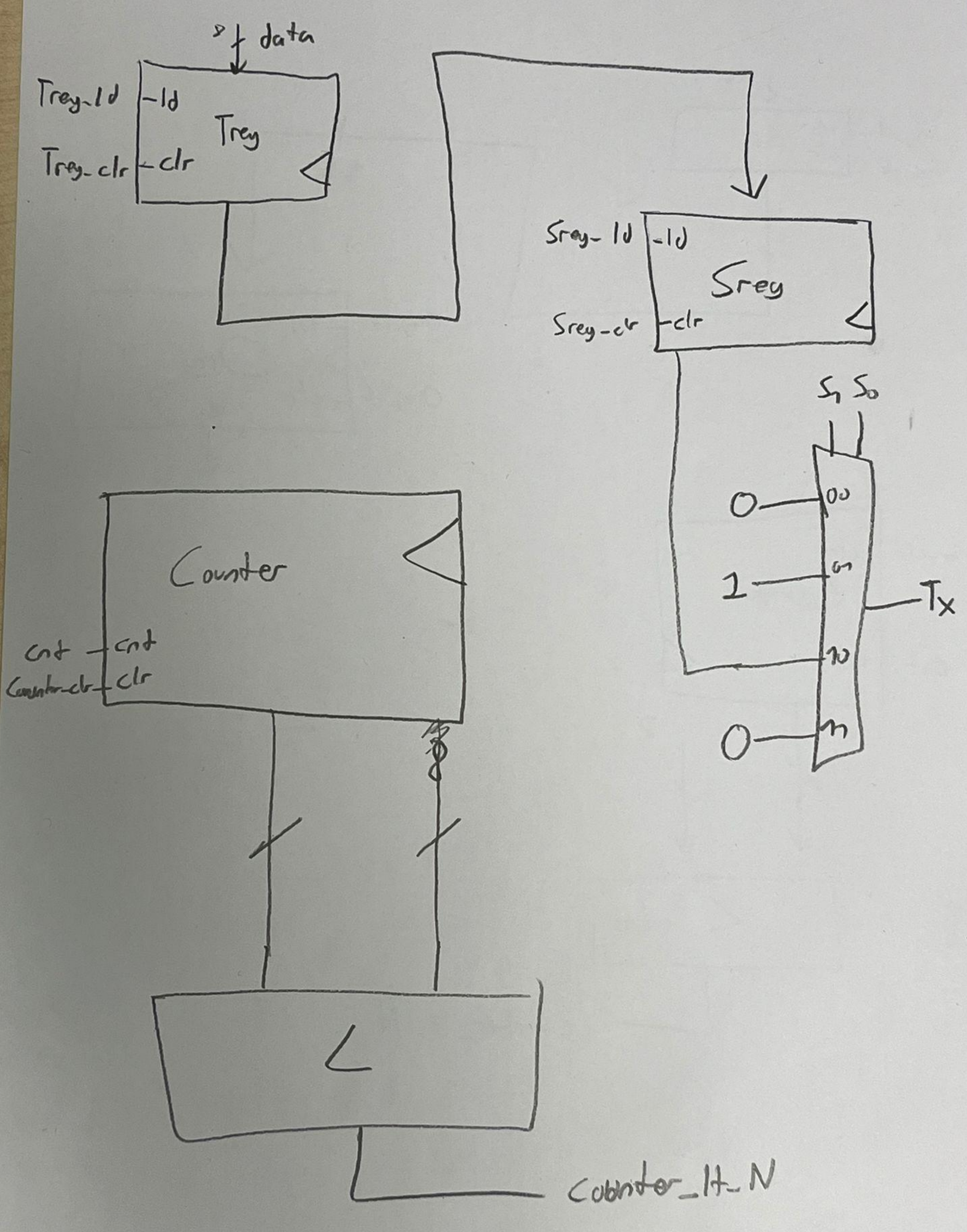
State Diagrams



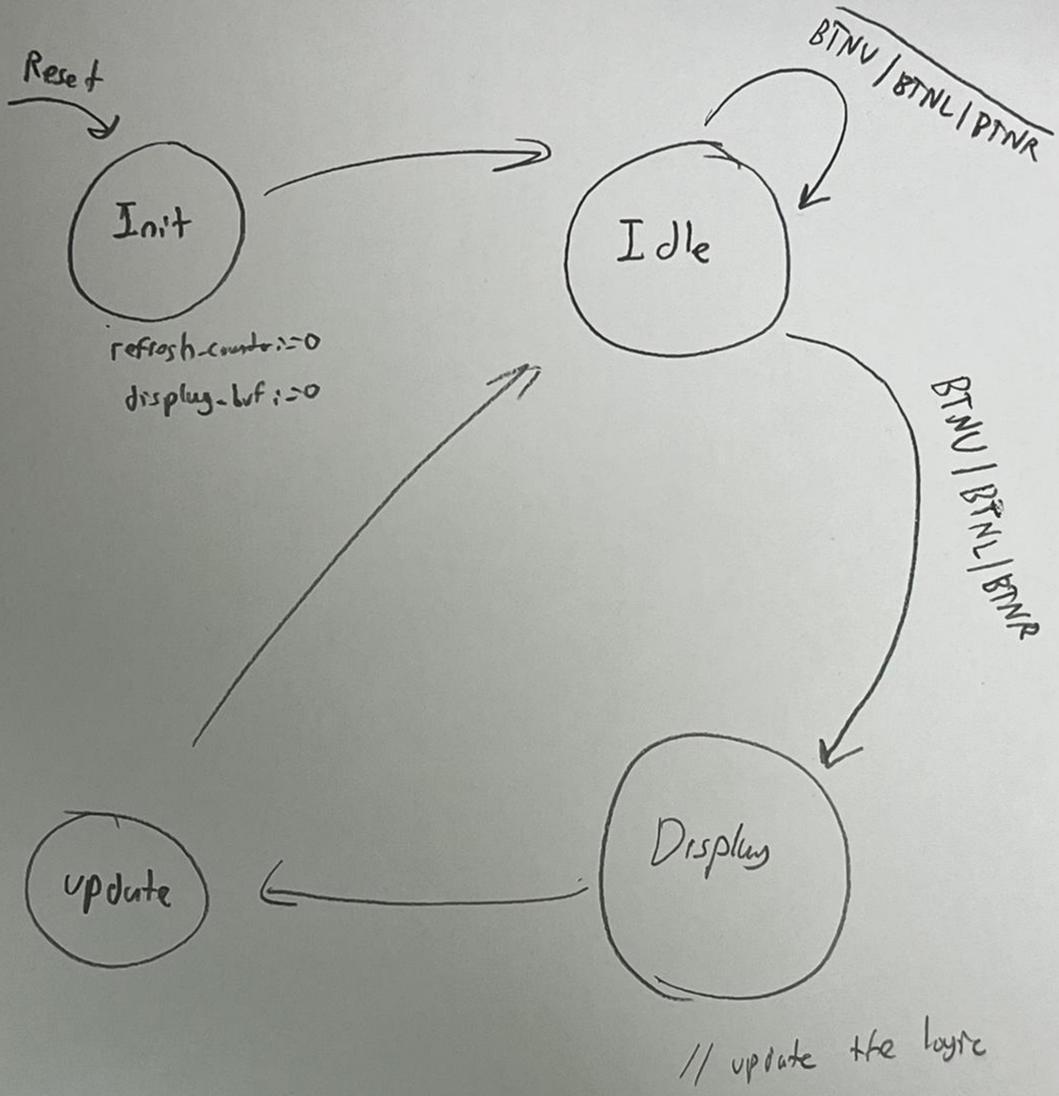
Transmitter ~~HLSM~~ FSM



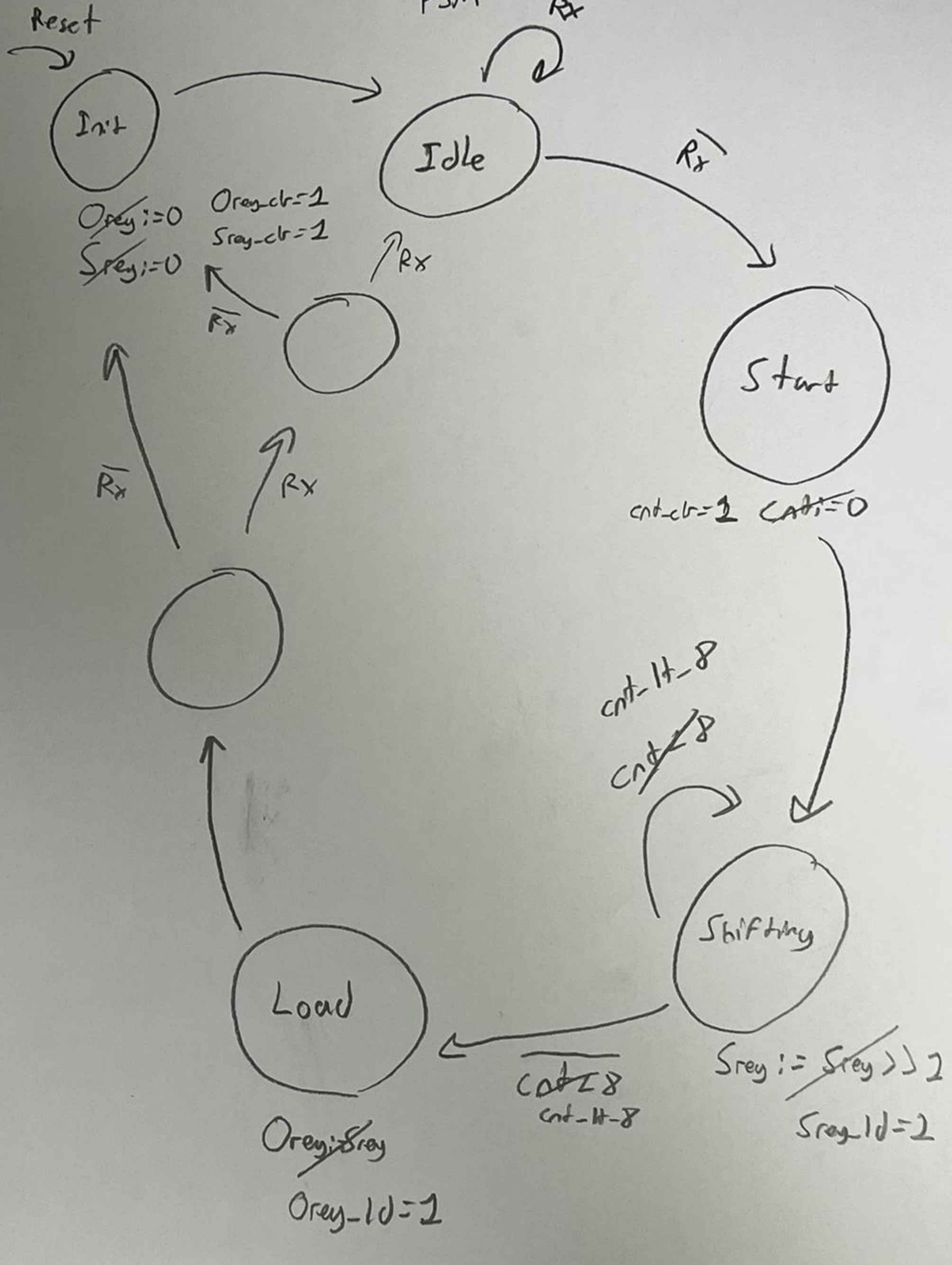
Transmitter Datapath



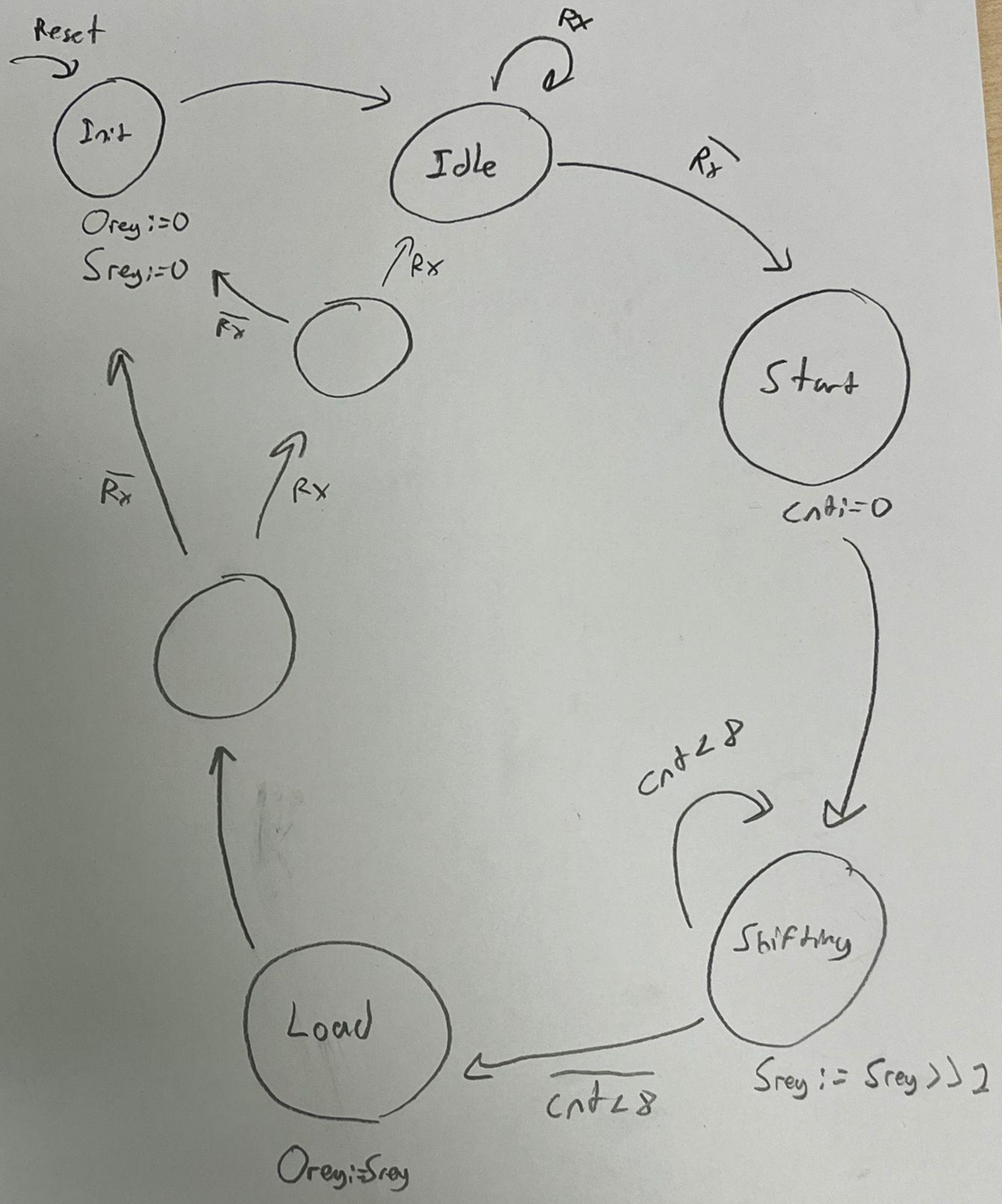
Seven Segment Display HLSM



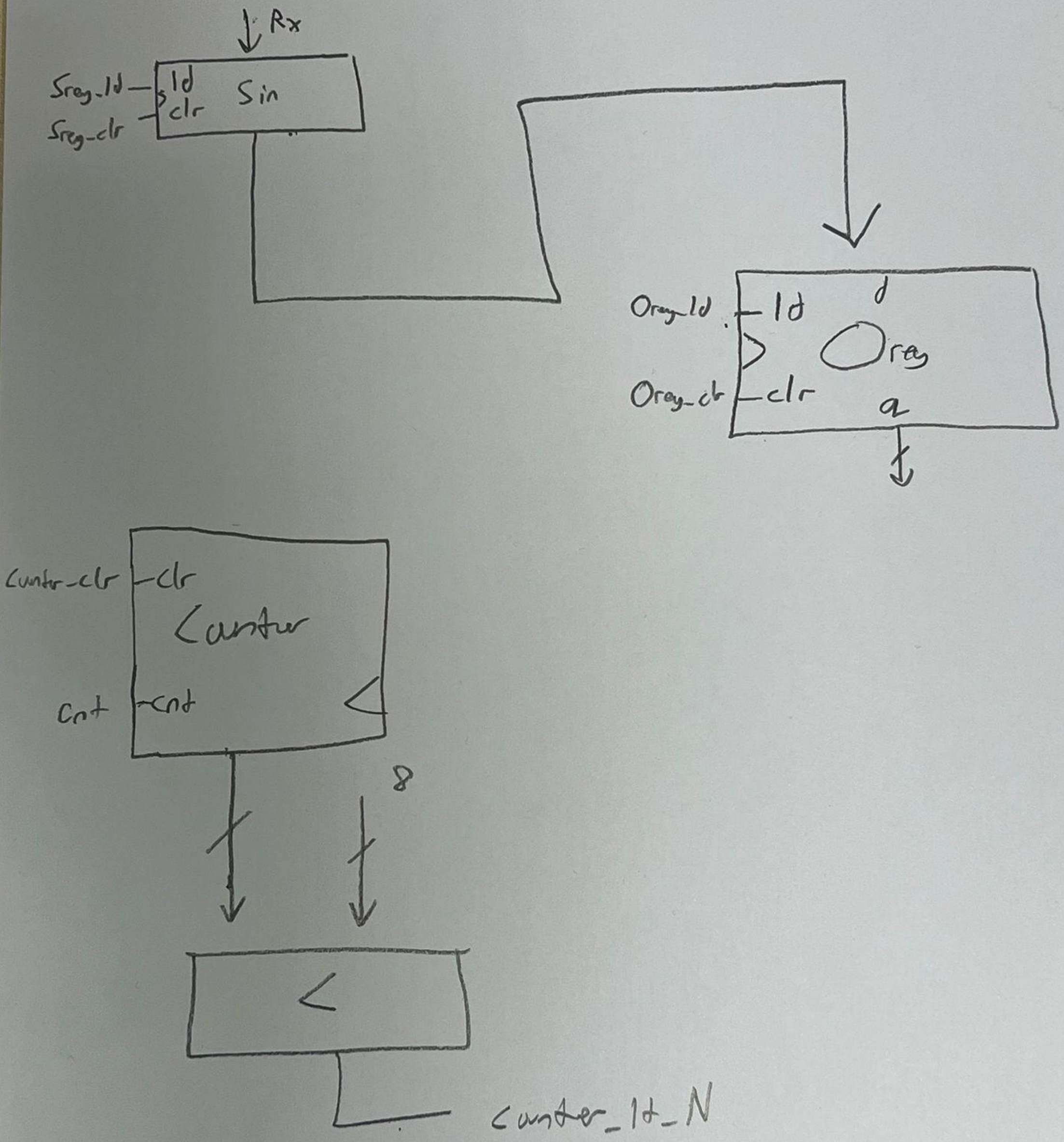
Receiver ~~HLSM~~ FSM



Receiver HLSM



Receiver Datapath



Appendix

```
module Uart_st1(
 input logic BTNU, BTNL, BTNR, BTND, BTNC, clk_in, reset,
 input logic [7:0] d,
 output logic [7:0] q0,
 output logic [1:0] screen,
 output logic RorTx,
 output logic [3:0] Anode_Activate,
 output logic [6:0] LED_out
    );

    reg [7:0] rxbuf [3:0];
    reg [7:0] txbuf [3:0];

    logic clk,tx;
    logic [7:0] q1;

    baudGenerator  clockGenerator(clk_in,clk);
    transmitter_fsm  trans(BTND,BTNC,clk,reset,d,q0,tx,txbuf);
    receiver_fsm  receive(tx,clk,reset,q1,rxbuf);
    seven_segment_display
displayer(BTNU,BTNL,BTNR,clk_in,reset,rxbuf,txbuf,screen,RorTx,Anode_Activate,L
ED_out);
```

```
endmodule

module transmitter_fsm( input logic BTND, BTNC, clk,reset,

input logic [7:0] d ,  output logic [7:0] q0,

output logic tx, output reg [7:0] txbuf [3:0]);


   logic Treg_clr, Sreg_clr,q0, Counter_clr, Sreg_ld, Treg_ld, cnt, sout, counter_lt_8,

Sreg_en;

   logic [1:0] s;

   logic [7:0] q1;



   eightBitRegister Treg(clk,Treg_clr,Treg_ld,d,q0);

   eightBitShiftRegister Sreg(clk,Sreg_clr,Sreg_ld,0,Sreg_en,txbuf[3],q1,sout);

   fourToOneMux frtOneMux(0,1,sout,0,s,tx);

   counter cntr(clk,Counter_clr,cnt,counter_lt_8);




    typedef enum logic [3:0] {S0,S1,S2,S3,S4,S5,S6,S7,S8} statetype;

    statetype [1:0] state, nextstate;


     always @(posedge clk, posedge reset)
   if(reset) state <= S0;
    else  state <= nextstate;

 //kuzeyyyyyy
always_comb
case (state)
S0: nextstate=S1;
S1: begin
if(BTND) nextstate=S2;
else if(BTNC) nextstate=S4;
else nextstate=S1;
end
S2:  nextstate=S3;
S3:
begin
if(BTNC) nextstate=S4;
else if(BTND) nextstate = S2;
else nextstate=S3;
end
S4: nextstate=S5;
S5: if(counter_lt_8) nextstate=S5;
else nextstate=S6;
S6: nextstate=S7;
S7: if(BTNC) nextstate=S7;
else nextstate = S1;
```

```
    S8: if(BTND) nextstate=S8;
    else nextstate = S1;
    default: nextstate=S0;
    endcase;

    always @(state)
    case (state)
    S0: begin s= 2'b01; Treg_clr=1; Treg_ld =0; Sreg_clr=1; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=1; end
    S1: begin s= 2'b01; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
    S2: begin s= 2'b01; Treg_clr=0; Treg_ld =1; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
    S3: begin s= 2'b01; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
    S4: begin s= 2'b00; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=1; Sreg_en=0;
cnt=0; Counter_clr=1; end
    S5: begin s= 2'b10; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=1;
cnt=1; Counter_clr=0; end
    S6: begin s= 2'b01; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
    S7: begin s= 2'b01; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
    S8: begin s= 2'b01; Treg_clr=0; Treg_ld =0; Sreg_clr=0; Sreg_ld=0; Sreg_en=0;
cnt=0; Counter_clr=0; end
```

```
    endcase


     always_ff@(posedge clk)
     if(state == S0) begin
     txbuf[0] = 0;
     txbuf[1] = 0;
     txbuf[2] = 0;
     txbuf[3] = 0;
     end
     else if (state == S2) begin
     txbuf[0] <= q0;
     txbuf[1] <=  txbuf[0];
     txbuf[2] <=  txbuf[1];
     txbuf[3] <=  txbuf[2];
     end


endmodule
module receiver_fsm( input logic rx,clk,reset,
          output logic [7:0] toDisplay,
          output reg [7:0] rxbuf [3:0]
    );
       logic Oreg_clr, Sreg_clr, cnt, cnt_clr, Sreg_ld, Oreg_ld, counter_lt_8, Sreg_en,
sout;


    logic [7:0] q0;
```

```
eightBitShiftRegister Sreg(clk,Sreg_clr,0,rx,Sreg_en,0,q0,sout);

eightBitRegister Oreg(clk,Oreg_clr,Oreg_ld,rxbuf[3],toDisplay);

counter cntr(clk,cnt_clr,cnt,counter_lt_8);


 typedef enum logic [2:0] {S0,S1,S2,S3,S4,S5} statetype;

 statetype [1:0] state, nextstate;


 always @(posedge clk, posedge reset)

   if(reset) state <= S0;

    else  state <= nextstate;




always_comb

case (state)

S0: nextstate = S1;

S1: if(rx) nextstate=S1;

else nextstate=S2;

S2:  if(counter_lt_8) nextstate=S2;

else nextstate=S3;

S3: if(rx) nextstate=S4;

else nextstate=S0;

S4: if(rx) nextstate=S5;

else nextstate=S0;
```

```
   S5: nextstate=S1;
   default: nextstate=S0;
   endcase



   always @(state)
   case (state)
   S0: begin  Oreg_clr = 1; Oreg_ld = 0; Sreg_clr = 1; Sreg_ld = 0; Sreg_en = 0;
cnt_clr = 1; cnt = 0;

    end
   S1: begin  Oreg_clr = 0; Oreg_ld = 0; Sreg_clr = 1; Sreg_ld = 0; Sreg_en = 0;
cnt_clr = 1; cnt = 0;  end
   S2: begin  Oreg_clr = 0; Oreg_ld = 0; Sreg_clr = 0; Sreg_ld = 0; Sreg_en = 1;
cnt_clr = 0; cnt = 1;  end
   S3: begin  Oreg_clr = 0; Oreg_ld = 0; Sreg_clr = 0; Sreg_ld = 0; Sreg_en = 0;
cnt_clr = 0; cnt = 0;  end
   S4: begin  Oreg_clr = 0; Oreg_ld = 0; Sreg_clr = 0; Sreg_ld = 0; Sreg_en = 0;
cnt_clr = 0; cnt = 0;  end
   S5: begin  Oreg_clr = 0; Oreg_ld = 1; Sreg_clr = 0; Sreg_ld = 0; Sreg_en = 0;
cnt_clr = 0; cnt = 0;  end
   endcase

module seven_segment_display(
   input logic BTNU, BTNL, BTNR, clk, reset,
    input logic [7:0] rxbuf [3:0],
    input logic [7:0] txbuf [3:0],
    output logic [1:0] screen ,
    output logic RorTx,
    output logic [3:0] Anode_Activate,
    output logic [6:0] LED_out
);

    reg [7:0] display_buf [3:0];
    reg [1:0] current_digit;
    reg [1:0] array_index;
    reg [3:0] BCD;
    reg [19:0] refresh_counter;
    wire [1:0] LED_activating_counter;
    reg display_rxbuf;

    // Refresh counter for multiplexing display digits
    always @(posedge clk, posedge reset) begin
        if(reset) begin
            refresh_counter <= 0;
            RorTx <= 0;
        end else begin
            refresh_counter <= refresh_counter + 1;
        end
    end
```

```
assign LED_activating_counter = refresh_counter[19:18];


always_ff@(posedge clk, posedge reset) begin
   if(reset) begin
      display_buf <= txbuf;
      current_digit <= 0;
      array_index <= 0;
   end
   else if(BTNU) begin
      RorTx <= ~RorTx;
      display_buf <= RorTx ? rxbuf : txbuf;
   end
   else if(BTNL) begin
      array_index <= (array_index == 0) ? 3 : array_index - 1;
   end
   else if(BTNR) begin
      array_index <= (array_index == 3) ? 0 : array_index + 1;
   end
end


// Determine which digit to activate and display
always @(*) begin
   case(LED_activating_counter)
      2'b00: Anode_Activate <= 4'b1110;
      2'b01: Anode_Activate <= 4'b1101;
```

```
        2'b10: Anode_Activate <= 4'b1011;
        2'b11: Anode_Activate <= 4'b0111;
    endcase
    BCD <= display_buf[array_index][3:0];
end

always @(*) begin
    case(BCD)
    4'b0000: LED_out = 7'b0000001; // Example pattern for '0'
    4'b0001: LED_out = 7'b1001111; // "1"
    4'b0010: LED_out = 7'b0010010; // "2"
    4'b0011: LED_out = 7'b0000110; // "3"
    4'b0100: LED_out = 7'b1001100; // "4"
    4'b0101: LED_out = 7'b0100100; // "5"
    4'b0110: LED_out = 7'b0100000; // "6"
    4'b0111: LED_out = 7'b0001111; // "7"
    4'b1000: LED_out = 7'b0000000; // "8"
    4'b1001: LED_out = 7'b0000100; // "9"
    4'b1010: LED_out = 7'b0001000; // "A"
    4'b1011: LED_out = 7'b0000011; // "B"
    4'b1100: LED_out = 7'b1000110; // "C"
    4'b1101: LED_out = 7'b0100001; // "D"
    4'b1110: LED_out = 7'b0000110; // "E"
    4'b1111: LED_out = 7'b0001110; // "F"
    default: LED_out = 7'b1111111; // Blank
```

```
        endcase
    end



    assign screen = (array_index < 2) ? 2'b01 : 2'b10;
endmodule




module baudGenerator#(parameter N = 112500)(input logic clk_in,
                     output logic clk);
reg [32:0] counter;
reg clk_out = 0;
    always@(posedge clk_in)
    begin
    counter <= counter+1;
    if(counter == 150000000/N)
    begin
    counter <= 0;
    clk_out = ~clk_out;
    end
    end

assign clk = clk_out;
```

```
endmodule
```