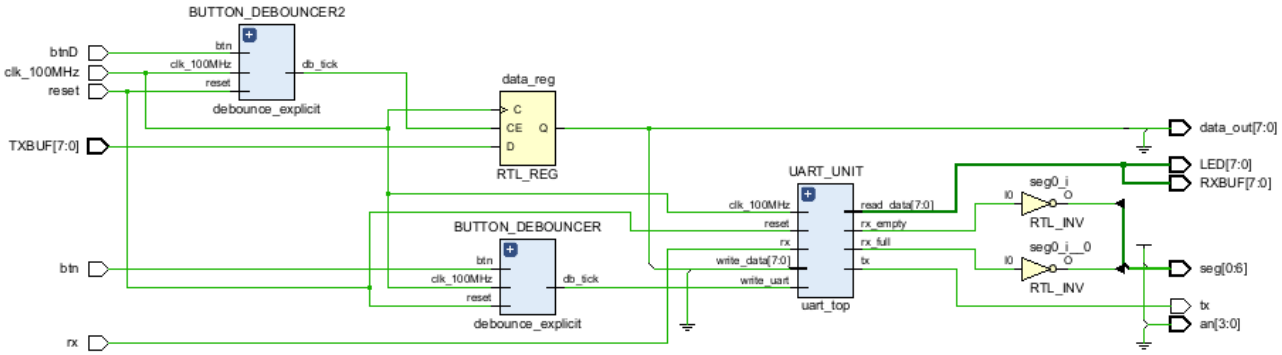
CS223 Lab Project

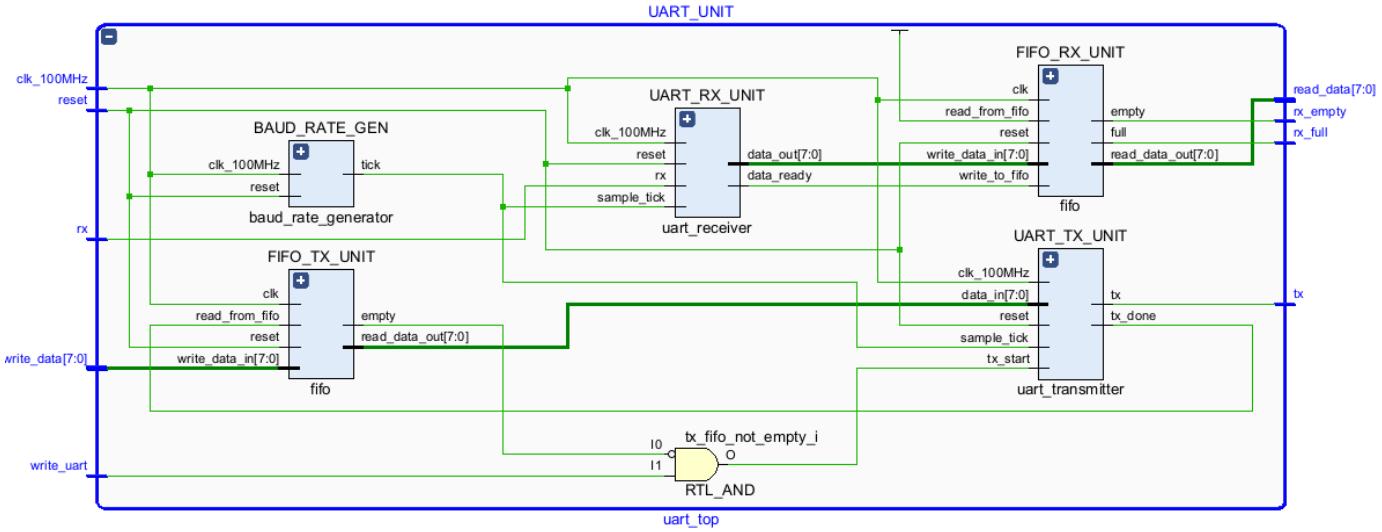
Altay İlker Yiğitel
CS223 Sec: 01
22203024

I have implemented the uart using main modules and connecting modules which are baud controller uart receiver transmitter debouncers seg7 displays and fifos connecting rx's and tx's. UART allows me to send and receive data from and to another device. It sends and receives data in bytes. There are 1 stop bit, 1 start bit and 1 parity bit.

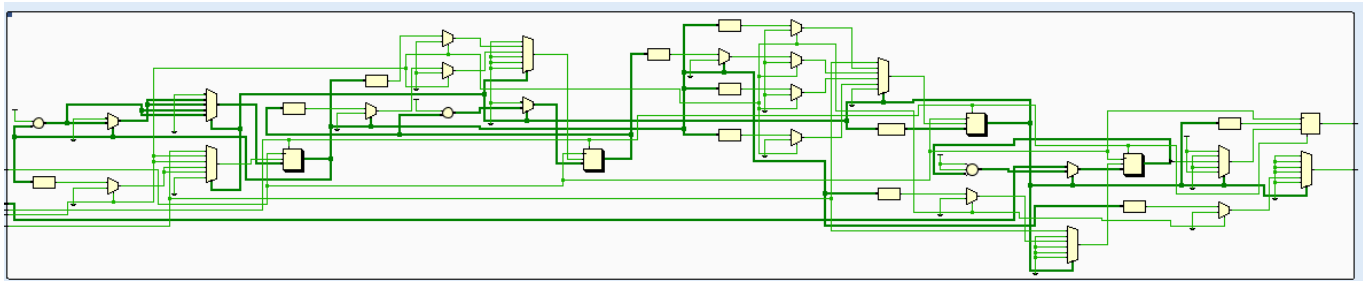
Uart top



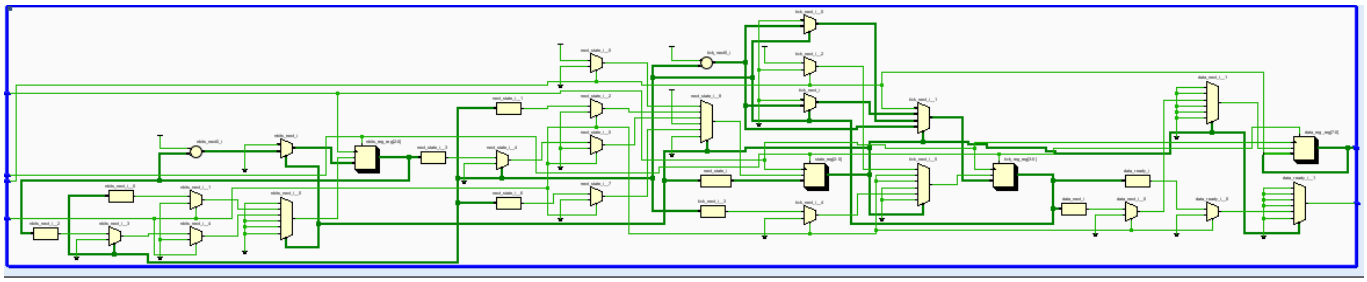
Uart

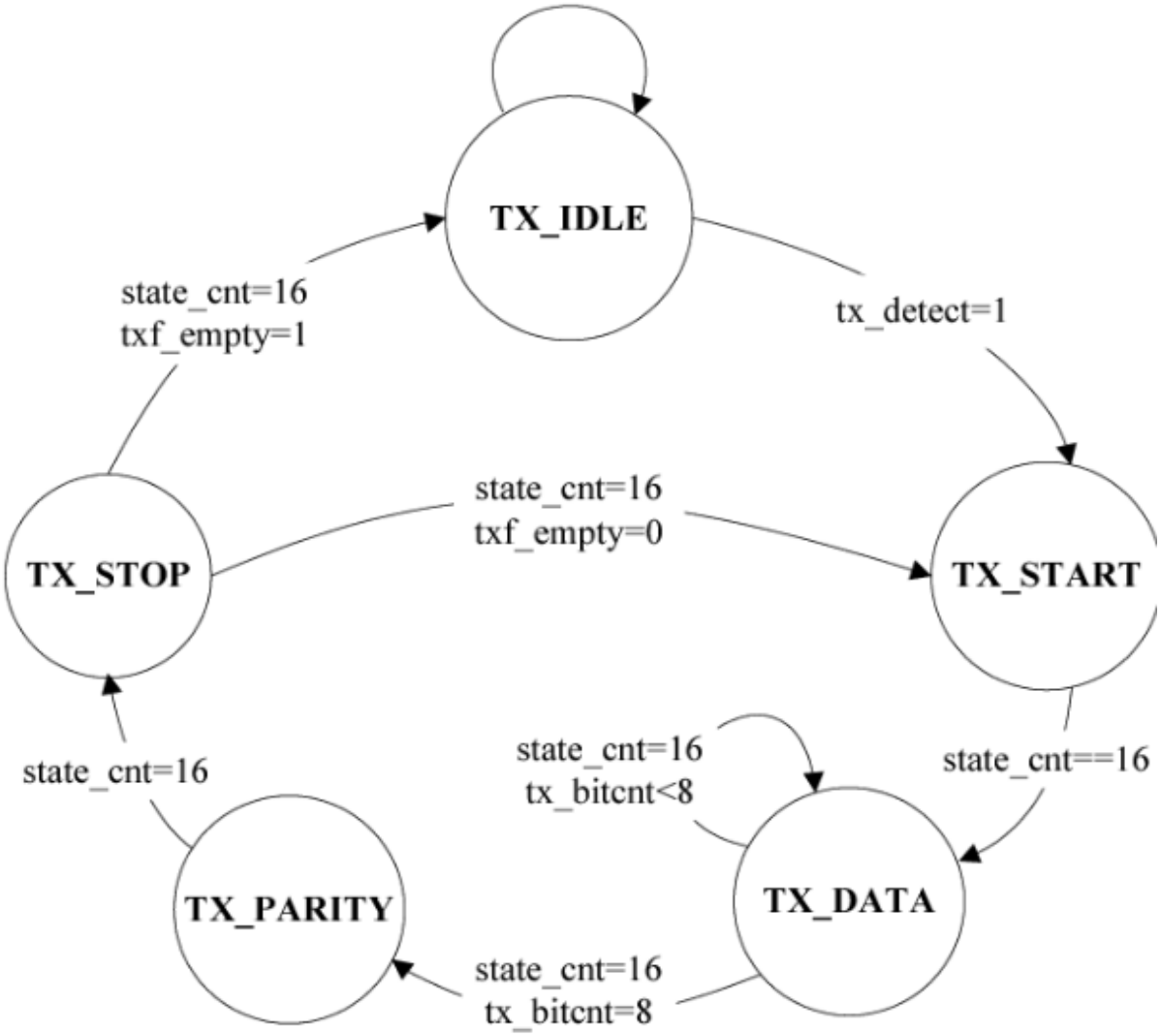


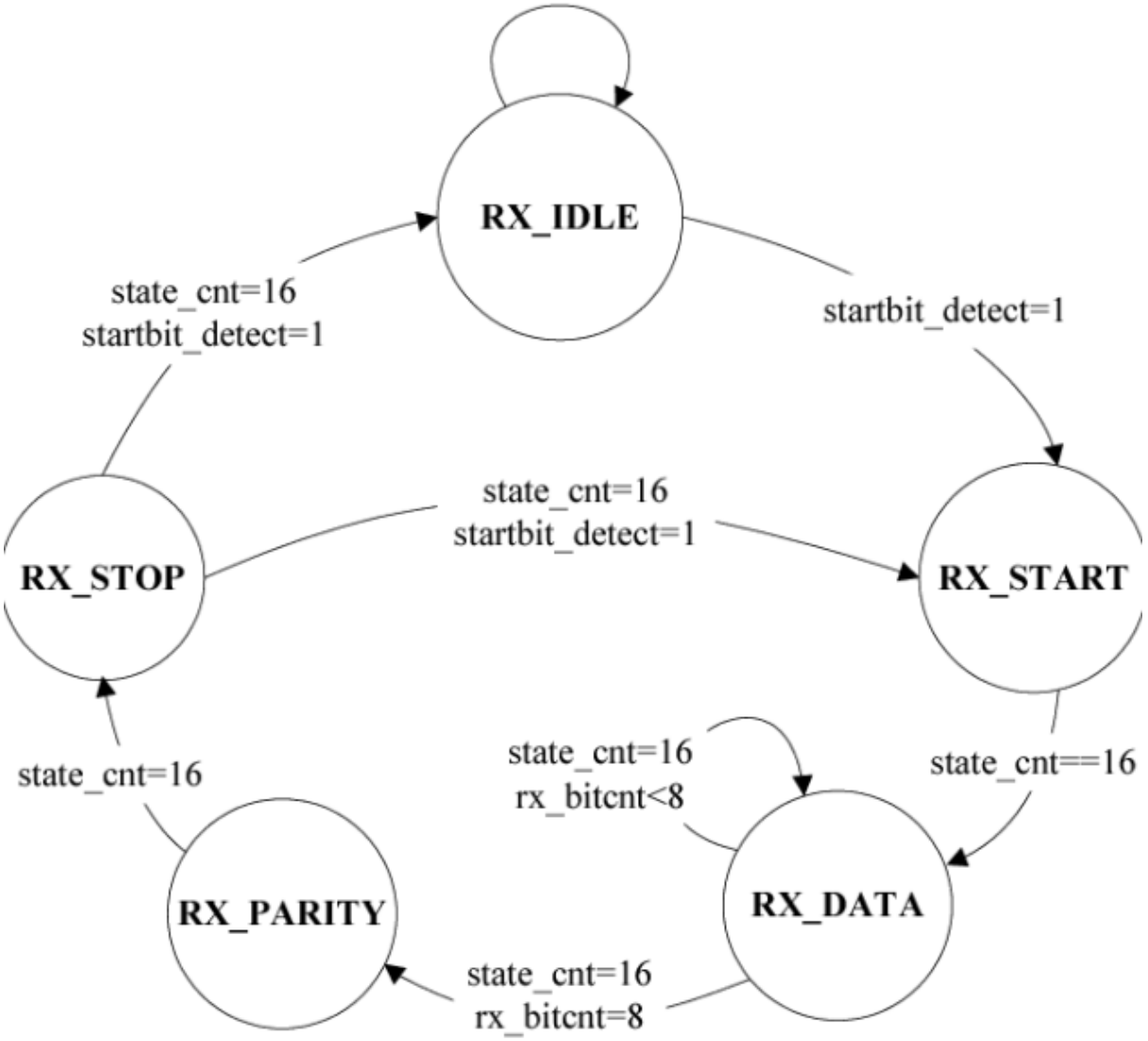
TX

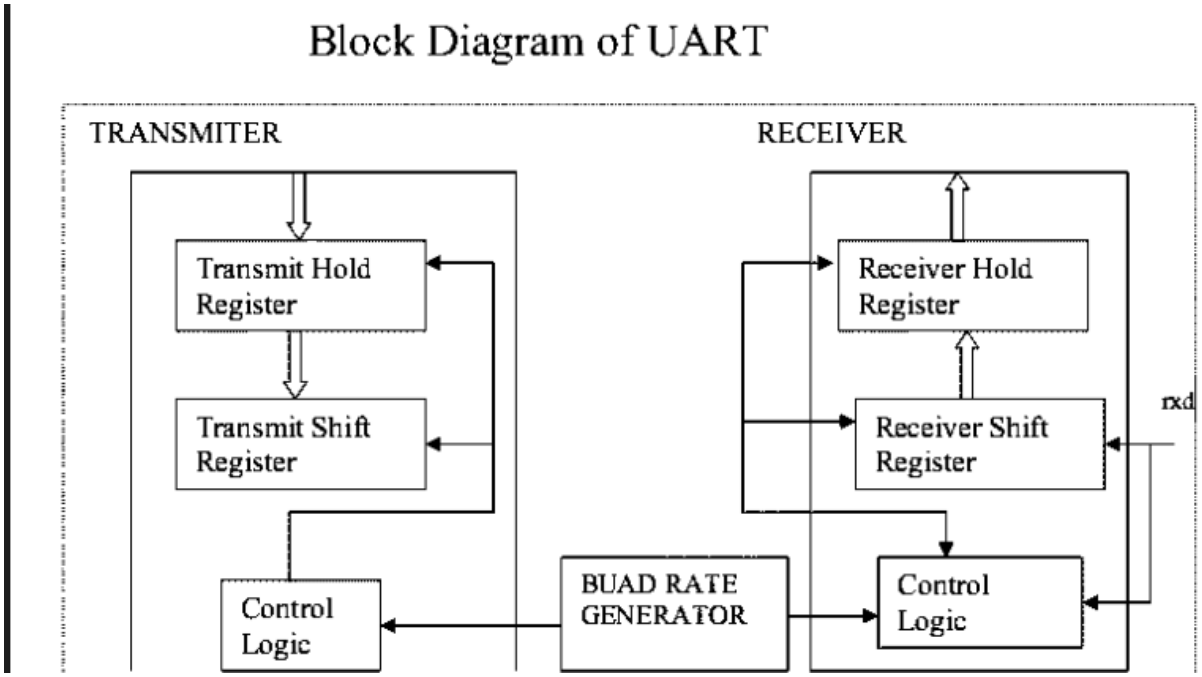


RX




TX_IDLE
state_cnt=16
txf_empty=1
tx_detect=1
TX_STOP
state_cnt=16
txf_empty=0
TX_START
state_cnt=16
state_cnt==16
state_cnt=16
tx_bitcnt<8
TX_PARITY
TX_DATA
state_cnt=16
tx_bitcnt=8


RX_IDLE
startbit_detect=1
RX_START
state_cnt==16
RX_DATA
state_cnt=16
rx_bitcnt=8
RX_PARITY
state_cnt=16
RX_STOP
state_cnt=16
startbit_detect=1
state_cnt=16
rx_bitcnt<8




```
module uart_test(
    input clk_100MHz,       // basys 3 FPGA clock signal
    input reset,            // btnR
    input rx,               // USB-RS232 Rx
    input btn,              // btnL (read and write FIFO operation)
    input btnD,
    input btnU,
    input btnL,
    input [7:0] TXBUF,
    output tx,              // USB-RS232 Tx
    output [3:0] an,        // 7 segment display digits
    output [0:6] seg,       // 7 segment display segments
    output [7:0] LED,       // data byte display
    output [7:0] data_out,
    output reg [7:0] RXBUF
    );
    bit [7:0] hxxt[4];
    bit [7:0] hxxr[4];
    reg  lastr = 0;


    reg data;
```

```
reg [3:0] sec_ones = 0;
reg [3:0] sec_tens = 0;
reg [3:0] sec_hundereds = 0;
reg [3:0] sec_thousands = 0;

// Connection Signals
wire rx_full, rx_empty, btn_tick,btn_tick2,btn_tick3,btn_tick4;
wire [7:0] rec_data, rec_data1;

initial begin
    for(int i = 0;i<$size(hxxt);i++)begin
        hxxt[i] = 0;
        hxxr[i] = 0;
    end
end

always @(posedge clk_100MHz)
    if(btn_tick2)
        data = TXBUF;


always@(posedge clk_100MHz)
    if(hxxt[0] == 0 && ~rx)begin
        for(int i = $size(hxxt)-2;i>=0;i--)begin
        hxxt[i] = hxxt[i+1];
        end
        hxxt[0] = TXBUF;

    end



always@(posedge clk_100MHz)
    if(hxxr[0] == 0 && btn_tick2)begin
        for(int i = $size(hxxt)-2;i>=0;i--)begin
        hxxr[i] = hxxr[i+1];
        end
        hxxr[0] = RXBUF;
        if(lastr<3)
        lastr++;
    end
```

```
always@(posedge clk_100MHz)
    if(btn_tick3) begin
        if(sec_thousands<1)
            sec_thousands++;
        else
            sec_thousands = 0;

    end

always@(posedge clk_100MHz)
    if(btn_tick4) begin
        if(sec_hundereds<3)
            sec_hundereds++;
        else
            sec_hundereds = 0;

    end

always@(posedge clk_100MHz)
    if(sec_thousands == 0)begin
        sec_tens = hxxt[sec_hundereds] [3:0];
        sec_ones = hxxt[sec_hundereds] [7:4];
    end
    else begin
        sec_tens = hxxr[sec_hundereds] [3:0];
        sec_ones = hxxr[sec_hundereds] [7:4];
    end









// Complete UART Core
uart_top UART_UNIT
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        //.read_uart(btn_tick),
        .write_uart(btn_tick),
```

```
        .rx(rx),
        .write_data(data), //rec_data1
        .rx_full(rx_full),
        .rx_empty(rx_empty),
        .read_data(rec_data),
        .tx(tx)
    );

// Button Debouncer
debounce_explicit BUTTON_DEBOUNCER
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .btn(btn),
        .db_level(),
        .db_tick(btn_tick)
    );

debounce_explicit BUTTON_DEBOUNCER2
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .btn(btnD),
        .db_level(),
        .db_tick(btn_tick2)
    );
debounce_explicit BUTTON_DEBOUNCER3
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .btn(btnU),
        .db_level(),
        .db_tick(btn_tick3)
    );
debounce_explicit BUTTON_DEBOUNCER4
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .btn(btnL),
        .db_level(),
        .db_tick(btn_tick4)
    );
```

```
   seg7_control seg7(.clk_100MHz(clk_100MHz), .reset(res), .ones(sec_ones),
.tens(sec_tens), .hundereds(sec_hundereds),
                    .thousands(sec_thousands), .seg(seg), .an(an));


   // Signal Logic
   assign rec_data1 = rec_data + 1;    // add 1 to ascii value of received data (to
transmit)


   assign RXBUF = rec_data;


   // Output Logic
   assign LED = hxxr[lastr];             // data byte received displayed on LEDs

   assign data_out = hxxt[0];
Endmodule




module uart_top
   #(
      parameter   DBITS = 8,          // number of data bits in a word
                  SB_TICK = 16,       // number of stop bit / oversampling ticks
                  BR_LIMIT = 52,     // baud rate generator counter limit
                  BR_BITS = 6,       // number of baud rate generator counter bits
                  FIFO_EXP = 2        // exponent for number of FIFO addresses (2^2 = 4)
   )
   (
      input clk_100MHz,               // FPGA clock
      input reset,                   // reset
      //input read_uart,                // button
      input write_uart,               // button
      input rx,                      // serial data in
      input [DBITS-1:0] write_data,   // data from Tx FIFO
      output rx_full,                 // do not write data to FIFO
      output rx_empty,                // no data to read from FIFO
      output tx,                     // serial data out
      output [DBITS-1:0] read_data    // data to Rx FIFO
   );
```

```
// Connection Signals
wire tick;                      // sample tick from baud rate generator
wire rx_done_tick;              // data word received
wire tx_done_tick;              // data transmission complete
wire tx_empty;                  // Tx FIFO has no data to transmit
wire tx_fifo_not_empty;         // Tx FIFO contains data to transmit
wire [DBITS-1:0] tx_fifo_out;   // from Tx FIFO to UART transmitter
wire [DBITS-1:0] rx_data_out;   // from UART receiver to Rx FIFO




// Instantiate Modules for UART Core
baud_rate_generator
   #(
      .M(BR_LIMIT),
      .N(BR_BITS)
    )
   BAUD_RATE_GEN
   (
      .clk_100MHz(clk_100MHz),
      .reset(reset),
      .tick(tick)
    );

uart_receiver
   #(
      .DBITS(DBITS),
      .SB_TICK(SB_TICK)
    )
    UART_RX_UNIT
    (
      .clk_100MHz(clk_100MHz),
      .reset(reset),
      .rx(rx),
      .sample_tick(tick),
      .data_ready(rx_done_tick),
      .data_out(rx_data_out)
    );

uart_transmitter
   #(
      .DBITS(DBITS),
```

```
        .SB_TICK(SB_TICK)
    )
    UART_TX_UNIT
    (
        .clk_100MHz(clk_100MHz),
        .reset(reset),
        .tx_start(tx_fifo_not_empty),
        .sample_tick(tick),
        .data_in(tx_fifo_out),
        .tx_done(tx_done_tick),
        .tx(tx)
    );

fifo
   #(
        .DATA_SIZE(DBITS),
        .ADDR_SPACE_EXP(FIFO_EXP)
    )
    FIFO_RX_UNIT
    (
        .clk(clk_100MHz),
        .reset(reset),
        .write_to_fifo(rx_done_tick),
   .read_from_fifo(1'b1),   //read_uart
   .write_data_in(rx_data_out),
   .read_data_out(read_data),
   .empty(rx_empty),
   .full(rx_full)
  );

fifo
   #(
        .DATA_SIZE(DBITS),
        .ADDR_SPACE_EXP(FIFO_EXP)
    )
    FIFO_TX_UNIT
    (
        .clk(clk_100MHz),
        .reset(reset),
        .write_to_fifo(write_uart2),
   .read_from_fifo(tx_done_tick),
   .write_data_in(write_data),
   .read_data_out(tx_fifo_out),
```

```
        .empty(tx_empty),
        .full()               // intentionally disconnected
      );
//        debounce_explicit debounce1(
//              .clk_100MHz(clk_100MHz),
//              .reset(reset),
//              .btn(write_uart),
//              .db_level(),
//              .db_tick(write_uart2)
//              );


    // Signal Logic
    assign tx_fifo_not_empty = ~tx_empty && write_uart;

Endmodule




module baud_rate_generator
    #(              // 115200 baud
        parameter   N = 6,     // number of counter bits
                    M = 52     // counter limit value
    )
    (
        input clk_100MHz,       // basys 3 clock
        input reset,            // reset
        output tick             // sample tick
    );

    // Counter Register
    reg [N-1:0] counter;        // counter value
    wire [N-1:0] next;          // next counter value

    // Register Logic
    always @(posedge clk_100MHz, posedge reset)
        if(reset)
            counter <= 0;
        else
            counter <= next;

    // Next Counter Value Logic
```

```
    assign next = (counter == (M-1)) ? 0 : counter + 1;

    // Output Logic
    assign tick = (counter == (M-1)) ? 1'b1 : 1'b0;

Endmodule




module uart_receiver
    #(
        parameter   DBITS = 8,          // number of data bits in a data word
                    SB_TICK = 16        // number of stop bit / oversampling ticks (1 stop bit)
    )
    (
        input clk_100MHz,               // basys 3 FPGA
        input reset,                    // reset
        input rx,                       // receiver data line
        input sample_tick,              // sample tick from baud rate generator
        output reg data_ready,          // signal when new data word is complete
(received)
        output [DBITS-1:0] data_out     // data to FIFO
    );

    // State Machine States
    localparam [2:0] idle  = 3'b000,
                     start = 3'b001,
                     data  = 3'b010,
                     stop  = 3'b011,
                     parity = 3'b100;

    // Registers
    reg [2:0] state, next_state;        // state registers
    reg [3:0] tick_reg, tick_next;      // number of ticks received from baud rate
generator
    reg [2:0] nbits_reg, nbits_next;    // number of bits received in data state
    reg [7:0] data_reg, data_next;      // reassembled data word

    // Register Logic
    always @(posedge clk_100MHz, posedge reset)
        if(reset) begin
            state <= idle;
```

```
            tick_reg <= 0;
            nbits_reg <= 0;
            data_reg <= 0;
        end
        else begin
            state <= next_state;
            tick_reg <= tick_next;
            nbits_reg <= nbits_next;
            data_reg <= data_next;
        end

    // State Machine Logic
    always @* begin
        next_state = state;
        data_ready = 1'b0;
        tick_next = tick_reg;
        nbits_next = nbits_reg;
        data_next = data_reg;

        case(state)
            idle:
                if(~rx) begin              // when data line goes LOW (start condition)
                    next_state = start;
                    tick_next = 0;
                end
            start:
                if(sample_tick)
                    if(tick_reg == 7) begin
                        next_state = data;
                        tick_next = 0;
                        nbits_next = 0;
                    end
                    else
                        tick_next = tick_reg + 1;
            data:
                if(sample_tick)
                    if(tick_reg == 15) begin
                        tick_next = 0;
                        data_next = {rx, data_reg[7:1]};
                        if(nbits_reg == (DBITS-1))
                            next_state = stop;
                        else
                            nbits_next = nbits_reg + 1;
```

```
                end
                else
                    tick_next = tick_reg + 1;

            parity: begin
                ;
                next_state = stop;
            end


            stop:
                if(sample_tick)
                    if(tick_reg == (SB_TICK-1)) begin
                        next_state = idle;
                        data_ready = 1'b1;
                    end
                    else
                        tick_next = tick_reg + 1;
        endcase
    end

    // Output Logic
    assign data_out = data_reg;

Endmodule



module uart_transmitter
    #(
        parameter   DBITS = 8,          // number of data bits
                    SB_TICK = 16        // number of stop bit / oversampling ticks (1 stop bit)
    )
    (
        input clk_100MHz,               // basys 3 FPGA
        input reset,                    // reset
        input tx_start,                 // begin data transmission (FIFO NOT empty)
        input sample_tick,              // from baud rate generator
        input [DBITS-1:0] data_in,      // data word from FIFO
        output reg tx_done,             // end of transmission
        output tx                       // transmitter data line
    );
```

```
    // State Machine States
    localparam [2:0]    idle  = 3'b000,
                        start = 3'b001,
                        data  = 3'b010,
                        stop  = 3'b011,
                        parity = 3'b100;

    // Registers
    reg [2:0] state, next_state;            // state registers
    reg [3:0] tick_reg, tick_next;          // number of ticks received from baud rate
generator
    reg [2:0] nbits_reg, nbits_next;        // number of bits transmitted in data state
    reg [DBITS-1:0] data_reg, data_next;    // assembled data word to transmit serially
    reg tx_reg, tx_next;                    // data filter for potential glitches

    // Register Logic
    always @(posedge clk_100MHz, posedge reset)
        if(reset) begin
            state <= idle;
            tick_reg <= 0;
            nbits_reg <= 0;
            data_reg <= 0;
            tx_reg <= 1'b1;
        end
        else begin
            state <= next_state;
            tick_reg <= tick_next;
            nbits_reg <= nbits_next;
            data_reg <= data_next;
            tx_reg <= tx_next;
        end

    // State Machine Logic
    always @* begin
        next_state = state;
        tx_done = 1'b0;
        tick_next = tick_reg;
        nbits_next = nbits_reg;
        data_next = data_reg;
        tx_next = tx_reg;

        case(state)
            idle: begin                 // no data in FIFO
```

```
    tx_next = 1'b1;             // transmit idle
    if(tx_start) begin          // when FIFO is NOT empty
        next_state = start;
        tick_next = 0;
        data_next = data_in;
    end
end

start: begin
    tx_next = 1'b0;             // start bit
    if(sample_tick)
        if(tick_reg == 15) begin
            next_state = data;
            tick_next = 0;
            nbits_next = 0;
        end
        else
            tick_next = tick_reg + 1;
end

data: begin
    tx_next = data_reg[0];
    if(sample_tick)
        if(tick_reg == 15) begin
            tick_next = 0;
            data_next = data_reg >> 1;
            if(nbits_reg == (DBITS-1))
                next_state = parity;
            else
                nbits_next = nbits_reg + 1;
        end
        else
            tick_next = tick_reg + 1;
end

parity: begin
    tx_next = 1'b0;
    if(sample_tick)
        if(tick_reg == (SB_TICK-1)) begin
            next_state = stop;
        end
        else
            tick_next = tick_reg + 1;
```

```
            end


            stop: begin
                tx_next = 1'b1;         // back to idle
                if(sample_tick)
                    if(tick_reg == (SB_TICK-1)) begin
                        next_state = idle;
                        tx_done = 1'b1;
                    end
                    else
                        tick_next = tick_reg + 1;
            end
        endcase
    end

    // Output Logic
    assign tx = tx_reg;

Endmodule




module fifo
#(
   parameter  DATA_SIZE     = 8,        // number of bits in a data word
    ADDR_SPACE_EXP = 4                  // number of address bits (2^4 = 16 addresses)
)
(
   input clk,                           // FPGA clock
   input reset,                          // reset button
   input write_to_fifo,                     // signal start writing to FIFO
   input read_from_fifo,                    // signal start reading from FIFO
   input [DATA_SIZE-1:0] write_data_in,    // data word into FIFO
   output [DATA_SIZE-1:0] read_data_out,   // data word out of FIFO
   output empty,                           // FIFO is empty (no read)
   output full                            // FIFO is full (no write)
);

// signal declaration
reg [DATA_SIZE-1:0] memory [2**ADDR_SPACE_EXP-1:0];                  // memory
array register
```

```
reg [ADDR_SPACE_EXP-1:0] current_write_addr, current_write_addr_buff,
next_write_addr;
reg [ADDR_SPACE_EXP-1:0] current_read_addr, current_read_addr_buff,
next_read_addr;
reg fifo_full, fifo_empty, full_buff, empty_buff;
wire write_enabled;

// register file (memory) write operation
always @(posedge clk)
if(write_enabled)
memory[current_write_addr] <= write_data_in;

// register file (memory) read operation
assign read_data_out = memory[current_read_addr];

// only allow write operation when FIFO is NOT full
assign write_enabled = write_to_fifo & ~fifo_full;

// FIFO control logic
// register logic
always @(posedge clk or posedge reset)
if(reset) begin
current_write_addr  <= 0;
current_read_addr  <= 0;
fifo_full                   <= 1'b0;
fifo_empty                  <= 1'b1;       // FIFO is empty after reset
end
else begin
current_write_addr  <= current_write_addr_buff;
current_read_addr   <= current_read_addr_buff;
fifo_full                   <= full_buff;
fifo_empty                  <= empty_buff;
end

// next state logic for read and write address pointers
always @* begin
// successive pointer values
next_write_addr = current_write_addr + 1;
next_read_addr  = current_read_addr + 1;

// default: keep old values
current_write_addr_buff = current_write_addr;
current_read_addr_buff  = current_read_addr;
```

```
full_buff  = fifo_full;
empty_buff = fifo_empty;

// Button press logic
case({write_to_fifo, 1'b1})     // check both buttons
// 2'b00: neither buttons pressed, do nothing

2'b01: // read button pressed?
if(~fifo_empty) begin   // FIFO not empty
current_read_addr_buff = next_read_addr;
full_buff = 1'b0;   // after read, FIFO not full anymore
if(next_read_addr == current_write_addr)
empty_buff = 1'b1;
end

2'b10: // write button pressed?
if(~fifo_full) begin     // FIFO not full
current_write_addr_buff = next_write_addr;
empty_buff = 1'b0;  // after write, FIFO not empty anymore
if(next_write_addr == current_read_addr)
full_buff = 1'b1;
end

2'b11: begin  // write and read
current_write_addr_buff = next_write_addr;
current_read_addr_buff  = next_read_addr;
end
endcase
end

// output
assign full = fifo_full;
assign empty = fifo_empty;

Endmodule



module debounce_explicit(
    input clk_100MHz,
    input reset,
    input btn,              // button input
    output reg db_level,    // for switches
```

```
output reg db_tick      // for buttons
);

// state declarations
parameter [1:0] zero  = 2'b00,
                wait0 = 2'b01,
                one   = 2'b10,
                wait1 = 2'b11;

// Artix-7 has a 100MHz clk with a period of 10ns
// number of counter bits (2^N * 10ns = ~40ms)
parameter N = 22;

// signal declaration
reg [1:0] state_reg, next_state;
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
wire q_zero;
reg q_load, q_dec;

// body
// FSMD state and data registers
always @(posedge clk_100MHz or posedge reset)
   if(reset) begin
      state_reg <= zero;
      q_reg <= 0;
   end
   else begin
      state_reg <= next_state;
      q_reg <= q_next;
   end

// FSMD data path (counter) next state logic
assign q_next = (q_load) ? {N{1'b1}} :      // load all 1s
                (q_dec) ? q_reg - 1  :      // decrement
                 q_reg;                     // no change in q

// status signal
assign q_zero = (q_next == 0);

// FSMD control path next state logic
always @* begin
   next_state = state_reg;
```

```
        q_load = 1'b0;
        q_dec = 1'b0;
        db_tick = 1'b0;

        case(state_reg)
            zero    : begin
                        db_level = 1'b0;
                        if(btn) begin
                            next_state = wait1;
                            q_load = 1'b1;
                        end
            end

            wait1   : begin
                        db_level = 1'b0;
                        if(btn) begin
                            q_dec = 1'b1;
                            if(q_zero) begin
                                next_state = one;
                                db_tick = 1'b1;
                            end
                        end
                        else
                            next_state = zero;
            end

            one     : begin
                        db_level = 1'b1;
                        if(~btn) begin
                            q_dec = 1'b1;
                            if(q_zero)
                                next_state = zero;
                        end
                        else
                            next_state = one;
            end

            default : next_state = zero;
        endcase
    end

Endmodule
```

```
module seg7_control(
    input clk_100MHz,
    input reset,
    input [3:0] ones,
    input [3:0] tens,
    input [3:0] hundereds,
    input [3:0] thousands,
    output reg [0:6] seg,
    output reg [3:0] an
    );

    // Parameters for segment values
    parameter NULL  = 7'b111_1111;  // Turn off all segments
    parameter ZERO  = 7'b000_0001;  // 0
    parameter ONE   = 7'b100_1111;  // 1
    parameter TWO   = 7'b001_0010;  // 2
    parameter THREE = 7'b000_0110;  // 3
    parameter FOUR  = 7'b100_1100;  // 4
    parameter FIVE  = 7'b010_0100;  // 5
    parameter SIX   = 7'b010_0000;  // 6
    parameter SEVEN = 7'b000_1111;  // 7
    parameter EIGHT = 7'b000_0000;  // 8
    parameter NINE  = 7'b000_0100;  // 9
    parameter F = 7'b011_1000; //F;
    parameter E = 7'b011_0000; //E;
    parameter d = 7'b100_0010; //d;
    parameter c = 7'b111_0010; //c;
    parameter b = 7'b110_0000; //b;
    parameter A = 7'b000_1000; //A;
    parameter t = 7'b111_1000; //t;
    parameter r = 7'b111_1010; //r;


    // To select each anode in turn
        reg [1:0] anode_select;
        reg [16:0] anode_timer;

        always @(posedge clk_100MHz or posedge reset) begin
            if(reset) begin
                anode_select <= 0;
```

```
            anode_timer <= 0;
        end
        else
            if(anode_timer == 99_999) begin
                anode_timer <= 0;
                anode_select <=  anode_select + 1;
            end
            else
                anode_timer <=  anode_timer + 1;
    end

    always @(anode_select) begin
        case(anode_select)
            2'b00 : an = 4'b0111;
            2'b01 : an = 4'b1011;
            2'b10 : an = 4'b1101;
            2'b11 : an = 4'b1110;
        endcase
    end

// To drive the segments
always @*
    case(anode_select)
        2'b00 : begin
                    case(thousands)
                        4'b0000 : seg = t;
                        4'b0001 : seg = r;


                    endcase
                end

        2'b01 : begin
                    case(hundereds)
                        4'b0000 : seg = ZERO;
                        4'b0001 : seg = ONE;
                        4'b0010 : seg = TWO;
                        4'b0011 : seg = THREE;
                    endcase
                end

        2'b10 : begin
                    case(tens)
```

```
                4'b0000 : seg = ZERO;
                4'b0001 : seg = ONE;
                4'b0010 : seg = TWO;
                4'b0011 : seg = THREE;
                4'b0100 : seg = FOUR;
                4'b0101 : seg = FIVE;
                4'b0110 : seg = SIX;
                4'b0111 : seg = SEVEN;
                4'b1000 : seg = EIGHT;
                4'b1001 : seg = NINE;
                4'b1010 : seg = F;
                4'b1011 : seg = E;
                4'b1100 : seg = d;
                4'b1101 : seg = c;
                4'b1110 : seg = b;
                4'b1111 : seg = A;
            endcase
        end

    2'b11 : begin
            case(ones)
                4'b0000 : seg = ZERO;
                4'b0001 : seg = ONE;
                4'b0010 : seg = TWO;
                4'b0011 : seg = THREE;
                4'b0100 : seg = FOUR;
                4'b0101 : seg = FIVE;
                4'b0110 : seg = SIX;
                4'b0111 : seg = SEVEN;
                4'b1000 : seg = EIGHT;
                4'b1001 : seg = NINE;
                4'b1010 : seg = F;
                4'b1011 : seg = E;
                4'b1100 : seg = d;
                4'b1101 : seg = c;
                4'b1110 : seg = b;
                4'b1111 : seg = A;
            endcase
        end
endcase

Endmodule
```

References:

Github FGPADude, FGPA Projects. https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/UART