John Peterson
Tim Zodrow
Manji Pal
ECE 554
Lab 1
February 3, 2015

Mini-Project 1: SPART RS-232

This project was to design and implement a Special Purpose Asynchronous Receiver and Transmitter module. This consists of multiple parts: On the top level this is a processor (driver) and RS232 SPART. The driver controls the reception and transmission of data, and interfaces the data received/data to be sent with the FPGA board. The SPART consists of four parts: a baud rate generator, a transmitter, a receiver, and a small controller. The baud rate generator produces an enable signal for one clock cycle at the desired baud rate (frequency), which can vary to four different speeds. The transmitter handles transmission over RS232 of characters, including a buffer to load in the data and a shift register to shift out one bit at a time. The receiver handles the reception of RS232 data, including a buffer to read in data, and a shift register to shift it in as well. These are controlled and handled by the bus interface, or controller. This device ensures the inout signal is either driven for output or not driven for input, and ensures data is sent to the TX module or read from the RX module based on the signals it receives.

These modules were all tested and verified with various testbenches. For the individual parts, these testbenches were hand-crafted, raising certain signals at different times and reading the result manually. For example, to test RX module individually, we set up the signals, received 01010101 and then followed with 10101010. Once this worked, we were able to hook it up to the top level module and test more thoroughly. We followed a similar process for the TX module, baud rate generator and the bus line interface.

Following the individual testbenches, we used a larger testbench with the baud rate generator, TX and RX modules integrated. This testbench would run to completion if all parts worked correctly. Finally, we tested the driver using a testbench and tested the SPART using a testbench. These appeared to be working correctly.

Finally, once the code was tested in testbenches it was ready to be loaded on the Virtex-5 FPGA. The process for testing the characters proceeded as follows: we tested each individual character, and recorded the results. We encountered a few errors along the way, such as a couple characters are not displaying correctly. As the time was running out, we left it functioning as is.

The following are problems we encountered along the way. First of all, getting used to the Xilinx software provided a few hurdles. We referenced the slides and the tutorial to get over this. We also had some hurdles getting used to verilog basics, different types of style like blocking and non-blocking. Now that we figured out all the basics we will be ready to tackle more difficult challenges. Next, we encountered challenges figuring out how each module would work at a lower level stage, and what information we would need to share between modules. Examples and tutorials found on the internet of UART and RS232 helped us out. The baud rate generator also posed some problems, as we needed to figure out what enable signal and how often we would put it out. Confusion about the different signals was cleared up when we asked the TA to clarify. Finally, in our testbenches, challenges that we encountered included considering edge cases, so that we could guarantee performance for the demonstration.

For the final testbench, we designed a self-testing top_level_tb.v of the top_level.v that created its own RX, TX, and Baud Generator for testing the entire design. It would loop through initialization and data processing for each baud rate and send the same data. If the data was received correctly, no flag would be set. If there was an issue with the test, the bit corresponding to the baud rate would be set in the flags variable. Using these, we could verify that each Baud rate was working as intended and that our design could be implemented.

Following is the commented code for the different modules.

top_level.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:  Tim Zodrow, Manjot S Pal, Jack Peterson
//
// Create Date:
// Design Name:
// Module Name:    top_level
// Project Name:
// Target Devices:
// Tool versions:
// Description:
```

```
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module top_level(
    input clk,          // 100mhz clock
    input rst,          // Asynchronous reset, tied to dip switch 0
    output txd,         // RS232 Transmit Data
    input rxd,          // RS232 Recieve Data
    input [1:0] br_cfg // Baud Rate Configuration, Tied to dip switches 2 and 3
    );

       wire iocs;
       wire iorw;
       wire rda;
       wire tbr;
       wire [1:0] ioaddr;
       wire [7:0] databus;

       // Instantiate your SPART here
       spart spart0( .clk(clk),
                 .rst(rst),
                                    .iocs(iocs),
                                    .iorw(iorw),
                                    .rda(rda),
                                    .tbr(tbr),
                                    .ioaddr(ioaddr),
                                    .databus(databus),
                                    .txd(txd),
                                    .rxd(rxd)
                                   );

       // Instantiate your driver here
       driver driver0( .clk(clk),
                       .rst(rst),
                                           .br_cfg(br_cfg),
                                           .iocs(iocs),
                                           .iorw(iorw),
                                           .rda(rda),
                                           .tbr(tbr),
                                           .ioaddr(ioaddr),
                                           .databus(databus)
                                        );

endmodule
```

spart.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
// Company: UW Madison
// Engineers: Tim Zodrow, Manjot S Pal, Jack Peterson
```

```
//
// Create Date: 2/2/2015
// Design Name: mini-spart
// Module Name: spart
// Project Name: miniproject1
// Target Devices: FPGA
// Description: The spart is the main controller that brings all the modules together and
// performs the intreactions between the serial input (rx) and output (tx) as well as the
// databus and signals from the driver. It connects the bus interface, tx, rx, and baud
generator
// together

//////////////////////////////////////////////////////////////////////////////////
//
module spart(
    input clk,
    input rst,
    input iocs,
    input iorw,
    output rda,
    output tbr,
    input [1:0] ioaddr,
    inout [7:0] databus,
    output txd,
    input rxd
    );

    ///////////////////////////////////
    // Interconnects between modules //
    ///////////////////////////////////
       wire [7:0] databus_out, bus_interface_out, rx_data_out;
       wire sel, wrt_db_low, wrt_db_high, wrt_tx, tx_rx_en, rd_rx;

       // Tri-state buffer used to receive and send data via the databuse
    // Sel high = output, Sel low = input
       assign databus = sel ? databus_out : 8'bzz;

    // Bus Interface Module
    bus_interface bus0( .iocs(iocs),
                        .iorw(iorw),
                        .ioaddr(ioaddr),
                        .rda(rda),
                        .tbr(tbr),
                        .databus_in(databus),
                                        .databus_out(databus_out),
                        .data_in(rx_data_out),
                        .data_out(bus_interface_out),
                        .wrt_db_low(wrt_db_low),
                        .wrt_db_high(wrt_db_high),
                        .wrt_tx(wrt_tx),
                                        .rd_rx(rd_rx),
                                        .databus_sel(sel)
                        );

       // Baud Rate Generator Module
       baud_rate_gen baud0(  .clk(clk),
                                                .rst(rst),
                                                .en(tx_rx_en),
                                                .data(bus_interface_out),
                                                .sel_low(wrt_db_low),
```

```
                                                    .sel_high(wrt_db_high)
                                                    );

        // TX Module (Sends data out)
        tx tx0( .clk(clk),
                        .rst(rst),
                        .data(bus_interface_out),
                        .en(tx_rx_en),
                        .en_tx(wrt_tx),
                        .tbr(tbr),
                        .TxD(txd)
                        );

    // RX Module (Recieves data in)
        rx rx0( .clk(clk),
                        .rst(rst),
                        .RxD(rxd),
                        .Baud(tx_rx_en),
                        .RxD_data(rx_data_out),
                        .RDA(rda),
                        .rd_rx(rd_rx)
                );

endmodule
```

driver.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: UW Madison
// Engineers: Tim Zodrow, Manjot S Pal, Jack Peterson
//
// Create Date: 2/2/2015
// Design Name: mini-spart
// Module Name: driver
// Project Name: miniproject1
// Target Devices: FPGA
// Description: The driver starts initializes and drives the sparts transactions and receives
// data from the databus. It sets the baud rate from the br_cfg and sends out signals to
// capture the data from the spart and send data back.

//////////////////////////////////////////////////////////////////////////////////
module driver(
    input clk,
    input rst,
    input [1:0] br_cfg,
    output reg iocs,
    output reg iorw,
    input rda,
    input tbr,
```

```
    output reg [1:0] ioaddr,
    inout [7:0] databus
    );

    ////////////////////////////
    // States for State Machine //
    ////////////////////////////
    localparam INIT_LOW_DB = 2'b00;
    localparam INIT_HIGH_DB = 2'b01;
    localparam RECEIVE = 2'b11;
    localparam RECEIVE_WAIT = 2'b10;

    ////////////////////////////////////
    // Registers used for control signals //
    ////////////////////////////////////
    reg sel,wrt_rx_data;

    // State Registers
    reg [2:0] state, nxt_state;
    // Data Registers
    reg [7:0] data_out, rx_data;

    // Tri-state buffer used to receive and send data via the databuse
    // Sel high = output, Sel low = input
        assign databus = sel ? data_out : 8'bz;

    // RX Received Data Flop
        always  @ (posedge clk, posedge rst) begin
                if(rst)
                        rx_data <= 8'h00;
                else if (wrt_rx_data)
                        rx_data <= databus;
        end

    // State Flop
        always @ (posedge clk, posedge rst) begin
                if(rst)
                        state <= 2'b00;
                else
                        state <= nxt_state;
        end

        ////////////////////
    // State Machine //
    ////////////////////
    always@(*) begin
         // Initializations
                ioaddr = 2'b00;
                sel = 0;
```

```
                iocs = 1;
                iorw = 1;
                nxt_state = INIT_LOW_DB;
                data_out = 8'h00;
                wrt_rx_data = 0;

                case(state)
              // Write the lower byte to Baud Gen
                        INIT_LOW_DB: begin
                                        ioaddr = 2'b10;
                                        sel = 1;
                                        nxt_state = INIT_HIGH_DB;
                                        case(br_cfg)
                                                2'b00:
                                                data_out = 8'hc0;            //baud rate to 4800
                                                2'b01:
                                                data_out = 8'h80;            //baud rate to 9600

                                                2'b10:
                                                data_out = 8'h00;            //baud rate to
19200
                                                2'b11:
                                                data_out = 8'h00;            //baud rate to
38400
                                        endcase
                        end
                        // Write the higher byte to Baud Gen
                        INIT_HIGH_DB: begin
                                        ioaddr = 2'b11;
                                        sel = 1;
                                        nxt_state = RECEIVE_WAIT;
                                        case(br_cfg)
                                                2'b00:
                                                data_out = 8'h12;            //baud rate to 4800
                                                2'b01:
                                                data_out = 8'h25;            //baud rate to 9600

                                                2'b10:
                                                data_out = 8'h4b;            //baud rate to
19200
                                                2'b11:
                                                data_out = 8'h96;            //baud rate to
38400
                                        endcase
                        end
                        // Wait for receive data to be read
                        RECEIVE_WAIT: begin
                                        if(~rda) begin
                                                nxt_state = RECEIVE_WAIT;
```

```
                                        iocs = 0;
                                end
                                else begin
                                        nxt_state = RECEIVE;
                                        wrt_rx_data = 1;
                                        ioaddr = 2'b00;
                                end
                        end
                        // Send receive data to TX when TX is ready for data
                        RECEIVE: begin
                                if(tbr) begin
                                        nxt_state = RECEIVE_WAIT;

                                        ioaddr = 2'b00;
                                        iorw = 0;
                                        data_out = rx_data;
                                        sel = 1;
                                end
                                else begin
                                        nxt_state = RECEIVE;
                                end
                        end
                endcase
        end
endmodule
```

baud_rate_gen.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Tim Zodrow, Manjot S Pal, Jack Peterson
//
// Create Date:    18:06:17 01/29/2015
// Design Name:
// Module Name:    baud_rate_gen
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module baud_rate_gen(
        input clk,
        input rst,
```

```
        output reg en,
        input [7:0] data,
        input sel_low,
        input sel_high
        );

reg [15:0] divisor_buffer;
reg state, nxt_state;
reg load_counter, en_counter;
reg [11:0] counter, counter_start;



localparam SEND_SIGNAL = 0;
localparam WAIT = 1;

always @ (posedge clk, posedge rst)
        if(rst)
                divisor_buffer <= 16'h0000;
        else if(sel_low)
                divisor_buffer <= {divisor_buffer[15:8], data};
        else if(sel_high)
                divisor_buffer <= {data, divisor_buffer[7:0]};

always @ (posedge clk, posedge rst)
        if(rst)
                state <= SEND_SIGNAL;
        else
                state <= nxt_state;

always @ (posedge clk, posedge rst)
        if(rst)
                counter <= 12'h000;
        else if(load_counter)
                counter <= counter_start;
        else if(en_counter)
                counter <= counter - 1;

always @ (*) begin
        counter_start = 0;
        case(divisor_buffer[15:12])
                4'h1 : counter_start = 12'h515;
                4'h2 : counter_start = 12'h28B;
                4'h4 : counter_start = 12'h145;
                4'h9 : counter_start = 12'h0A3;
        endcase
end


always @ (clk, rst, state, counter) begin
        en = 0;
        load_counter = 0;
        en_counter = 0;
        nxt_state = SEND_SIGNAL;
        case(state)
                SEND_SIGNAL : begin
                        en = 1;
                        load_counter = 1;
                        nxt_state = WAIT;
                end
```

```
                WAIT : begin
                        if(~(|counter))
                                nxt_state = SEND_SIGNAL;
                        else begin
                                en_counter = 1;
                                nxt_state = WAIT;
                        end
                end
        endcase
end
endmodule
```

bus_interface.v

```
//////////////////////////////////////////////////////////////////////////////////
// Company: UW Madison
// Engineers: Tim Zodrow, Manjot S Pal, Jack Peterson
//
// Create Date: 2/2/2015
// Design Name: mini-spart
// Module Name:    bus_interface
// Project Name: miniproject1
// Target Devices: FPGA
// Description: The Bus Interface constitutes a 3 state buffer which connects the SPART to
//the DATABUS. It contains combinational logic for the selecting the lower or higher bits.
//It utilizes IOCS and IOR/W signals to make sure that 3 state drivers don't in conflict
with
//other drivers on DATABUS.

//////////////////////////////////////////////////////////////////////////////////
//


//////////////////////
// Inputs, outputs //
//////////////////////
module bus_interface(
        input iocs,
        input iorw,
        input [1:0] ioaddr,
        input rda,
        input tbr,
        input [7:0] databus_in,
        output reg [7:0] databus_out,
        input [7:0] data_in,
        output reg [7:0] data_out,
        output reg wrt_db_low,
        output reg wrt_db_high,
        output reg wrt_tx,
        output reg rd_rx,
        output reg databus_sel
        );

//defaulting/initializing the signals
        always @ (*) begin
                data_out = 8'h00;
                wrt_db_low = 0;
                wrt_db_high = 0;
```

```
                wrt_tx = 0;
                databus_sel = 0;
                databus_out = 8'h00;
                rd_rx = 0;

//getting into different cases only when chip select is high
                if(iocs) begin
                        case(ioaddr)
                                2'b00 : begin
                                        // Recieve Buffer
                                        if(iorw) begin
                                                databus_sel = 1;
                                                databus_out = data_in;
                                                rd_rx = 1;
                                        end
                                        // Transmit Buffer
                                        else begin
                                                data_out = databus_in;
                                                wrt_tx = 1;
                                        end
                                end
                                2'b01 : begin
                                        //Status register
                                        if(iorw) begin
                                                //setting the databus select high
                                                databus_sel = 1;
                                                databus_out = {6'b000000, rda, tbr};
                                        end
                                end
                                2'b10 : begin
                                        //low division buffer
                                        data_out = databus_in;
                                        wrt_db_low = 1;                 //setting the low bits
signal high to indicate to fill out the lower bits
                                end
                                2'b11 : begin
                                        //high division buffer
                                        data_out = databus_in;
                                        wrt_db_high = 1;        //setting the high bits signal
high to indicate to fill out the higher bits
                                end
                        //ending the case statement
                        endcase
                end
        end
endmodule
```

rx.v

```
// RX module for RS232 communication
// Authors: John Peterson, Tim Zodrow

`timescale 1ns / 1ps
module rx(
                clk,
                rst,
                RxD,
                Baud,           // Input for Baud Rate, 16 ticks per bit
```

```
        RxD_data,      // Output for data received
        RDA,           // Data ready to be read
        rd_rx          // Input to clear ready signal, prepare for RX
        );
////////////
// Inputs //
////////////
input clk, rst, RxD, Baud, rd_rx;

/////////////
// Outputs //
/////////////
output [7:0] RxD_data;
output reg RDA;

////////////
// States //
////////////
parameter IDLE = 2'b00;
parameter STRTBIT = 2'b01;
parameter RCV = 2'b10;
parameter DONE = 2'b11;
reg [1:0] state, nxt_state;

///////////////
// Registers //
///////////////
reg RxD_ff1, RxD_ff2;
reg shift, set_RDA;
reg [7:0] RxD_shift;
reg [3:0] bit_cnt;
reg [4:0] baud_cnt;

/////////////
// Signals //
/////////////
wire negedgeRxD;
reg rst_bit_cnt, rst_baud_cnt;

// shift when baud cnt = 16;
// buffer for shift is 4 baud ticks (1/4 of period)
assign strt_shift = (baud_cnt == 5'b01000);
// Detect negative edge of RX for start signal
assign negedgeRxD = (~RxD_ff1 && RxD_ff2);
// Set output data
assign RxD_data = RxD_shift;

////////////////////
// Double flop RX //
////////////////////
always@(posedge clk, posedge rst) begin
        if(rst) begin
                RxD_ff1 <= 1'b0;
                RxD_ff2 <= 1'b0;
        end
        else begin
                RxD_ff1 <= RxD;
                RxD_ff2 <= RxD_ff1;
        end
end
```

```
//////////////////////
// RX shift register //
//////////////////////
always@(posedge clk, posedge rst) begin
	if(rst) begin
		RxD_shift <= 8'h00;
	end
	else if(shift) begin
		RxD_shift <= {RxD_ff2, RxD_shift[7:1]};
	end
end

////////////////////
// Bit counter FF //
////////////////////
always@(posedge clk, posedge rst) begin
	if(rst)
		bit_cnt <= 4'h0;
	else if(rst_bit_cnt) begin
		bit_cnt <= 4'h0;
	end
	else if(shift) begin
		bit_cnt <= bit_cnt + 1;
	end
end

//////////////////////////
// Baud tick counter FF //
//////////////////////////
always@(posedge clk, posedge rst) begin
	if(rst)
		baud_cnt <= 5'h0;
	else if(rst_baud_cnt) begin
		baud_cnt <= 5'h0;
	end
	else if(Baud) begin
		baud_cnt <= baud_cnt + 1;
	end
end

//////////////
// State FF //
//////////////
always@(posedge clk, posedge rst) begin
	if(rst) begin
		state <= IDLE;
	end
	else begin
		state <= nxt_state;
	end
end

always @ (posedge clk, posedge rst) begin
	if(rst)
		RDA <= 0;
	else
		RDA <= set_RDA;
end
```

```
	// Combinational logic for state machine
	always@(*) begin
		rst_bit_cnt = 0;
		rst_baud_cnt = 0;
		set_RDA = 0;
		nxt_state = IDLE;
		shift = 0;

		case(state)
			IDLE: begin    // Waiting for reception. Data not ready
				if(negedgeRxD) begin
					nxt_state = STRTBIT;
					rst_baud_cnt = 1;
				end
			end
			STRTBIT: begin // Receive start bit, transition to receive
				if(strt_shift) begin
					rst_baud_cnt = 1;
					shift = 1;
					rst_bit_cnt = 1;
					nxt_state = RCV;
				end
				else nxt_state = STRTBIT;
			end
			RCV: begin     // Receive data, wait until all eight received
				if(baud_cnt == 5'b10000) begin
					shift = 1;
					rst_baud_cnt = 1;
					if(bit_cnt == 4'h7) begin
						nxt_state = DONE;
						set_RDA = 1;
					end
					else begin
						nxt_state = RCV;
					end
				end
				else nxt_state = RCV;
			end
			DONE : begin   // Receive successful, output acknowledge
				if(rd_rx)
					nxt_state = IDLE;
				else begin
					nxt_state = DONE;
					set_RDA = 1;
				end
			end
		endcase
	end

endmodule
```

tx.v

```
//////////////////////////////////////////////////////////////////////////////////
// Company: UW Madison
// Engineers: Tim Zodrow, Manjot S Pal, Jack
//
// Create Date: 2/2/2015
```

```
// Design Name: mini-spart
// Module Name:    tx
// Project Name: miniproject1
// Target Devices: FPGA
// Description: This is a simple module which transmit the data. This is done with the
//help of some select signals which when asserted uses the sequential logic to count the
//number of enable signals and the correct number of shifts.

//////////////////////////////////////////////////////////////////////////////////
//


//////////////////////
// Inputs, outputs //
//////////////////////
`timescale 1ns / 1ps
module tx(
	input clk,
	input rst,
	input [7:0] data,
	input en,
	input en_tx,
	output reg tbr,
	output TxD
	);

//creating two states in order to carryout transmit
localparam IDLE = 1'b0;
localparam TRANS = 1'b1;

//10 bit receive buffer (including start and stop bit)
reg [9:0] receive_buffer;
//4 bit counter for counting enable signals and number of shifts
reg [3:0] en_counter, shft_counter;

//registers for different signals for sequential logic on which transmit takes place
reg state, nxt_state;
reg shft_start, shft_tick;
reg en_start, en_tick;
reg load, shft;

//////////////
// State FF //
//////////////
//always block for current state and next state set on posedge clk and rst
always @ (posedge clk, posedge rst)
	if(rst)
		state <= IDLE;
	else
		state <= nxt_state;

////////////////////
// Receive Buffer //
////////////////////
//always block for receive buffer for the data it gets depending on which signal is
//set set on posedge clk and rst
always @ (posedge clk, posedge rst)
	if(rst)
		receive_buffer <= 10'h001;
	else if (load)
```

```
                receive_buffer <= {1'b1,data,1'b0};
        else if (shft)
                receive_buffer <= {1'b1, receive_buffer[9:1]};

////////////////////////////////
// Enable (baud tick) counter //
////////////////////////////////

//always block for downcounting the enable signals set on posedge clk and rst
always @ (posedge clk, posedge rst)
        if(rst)
                en_counter <= 4'h0;
        else if(en_start)
                en_counter <= 4'hF;
        else if(en_tick)
                en_counter <= en_counter - 1;

//////////////////////
// Shift reg counter //
//////////////////////
//always block for counting the number of shifts set on posedge clk and rst
always @ (posedge clk, posedge rst)
        if(rst)
                shft_counter <= 4'h0;
        else if(shft_start)
                shft_counter <= 4'h9;
        else if(shft_tick)
                shft_counter <= shft_counter - 1;

//TxD always gets LSB of receive_buffer
assign TxD = receive_buffer[0];

//setting up defaults
always @ (clk, rst, data, en) begin
        nxt_state = IDLE;
        load = 0;
        en_start = 0;
        en_tick = 0;
        shft_start = 0;
        shft_tick = 0;
        shft = 0;
        tbr = 0;
        case(state)
                IDLE : begin
                        tbr = 1;
                        //setting different signals when en_tx is asserted and transmitting to
TRANS state
                        if(en_tx) begin
                                load = 1;
                                en_start = 1;
                                shft_start = 1;
                                nxt_state = TRANS;
                        end
                end
                TRANS : begin
                        tbr = 0;
                        if(en) begin                                            // Enable signal
detected
                                if(~(|en_counter)) begin                        // All enable
signals detected (16)
```

```
                                    if(~(|shft_counter)) begin          // All bits
transfered so transmission to IDLE
                                            nxt_state = IDLE;
                                    end
                                    else begin                          //if all bits not
transfered then stay in TRANS
                                            en_start = 1;               //state to transfer
rest of the bits
                                            shft_tick = 1;
                                            shft = 1;
                                            nxt_state = TRANS;
                                    end
                            end
                            else begin
                                    en_tick = 1;                        //if enable counter
is not full, stay in trans to
                                    nxt_state = TRANS;                  //count the rest of
the enable signals
                            end
                    end
                    else begin
                            nxt_state = TRANS;                          //if enable signal
is not asserted, stay in TRANS
                    end
            end
        endcase
end

endmodule
```

top_level_tb.v

```
`timescale 1ns / 1ps
module top_level_tb();

    reg stm_clk, stm_rst, stm_wrt_tx, stm_rd_rx, stm_sel_low, stm_sel_high;
    reg [1:0] stm_br_cfg;
    reg [7:0] stm_tx_data, stm_baud_data_in;
    reg [15:0] baud_rate_data;

    wire tbr_mon, txd_mon, rx_in, tx_rx_en, tb_rda_mon;
    wire [7:0] final_data_mon;

    reg [2:0] i;
    reg [3:0] flags;

       top_level top0(
           .clk(stm_clk),         // 100mhz clock
           .rst(stm_rst),         // Asynchronous reset, tied to dip switch 0
           .txd(txd_mon),        // RS232 Transmit Data
           .rxd(rx_in),         // RS232 Recieve Data
           .br_cfg(stm_br_cfg) // Baud Rate Configuration, Tied to dip switches 2 and 3
           );

       baud_rate_gen baud_tb0(
              .clk(stm_clk),
              .rst(stm_rst),
              .en(tx_rx_en),
```

```
                .data(stm_baud_data_in),
                .sel_low(stm_sel_low),
                .sel_high(stm_sel_high)
                );

        tx tx_tb0(
                .clk(stm_clk),
                .rst(stm_rst),
                .data(stm_tx_data),
                .en(tx_rx_en),
                .en_tx(stm_wrt_tx),
                .tbr(tbr_mon),
                .TxD(rx_in)
                );

        rx rx_tb0(
                .clk(stm_clk),
                .rst(stm_rst),
                .RxD(txd_mon),
                .Baud(tx_rx_en),
                .RxD_data(final_data_mon),
                .RDA(tb_rda_mon),
                .rd_rx(stm_rd_rx)
                );

/*
always
        case(i[1:0])
                2'b00 : baud_rate_data = 16'h12c0;
                2'b01 : baud_rate_data = 16'h2580;
                2'b10 : baud_rate_data = 16'h4b00;
                2'b11 : baud_rate_data = 16'h9600;
        endcase
*/

always
        #5 stm_clk <= ~stm_clk;

initial begin
        flags = 4'hf;
        for(i = 0; i < 4; i = i + 1) begin
                stm_clk = 0;
                stm_rst = 1;
                stm_br_cfg = i[1:0];

                // Baud Init
                stm_baud_data_in =    (i[1:0] == 2'b00) ? 8'hc0 :
                                      (i[1:0] == 2'b01) ? 8'h80 : 8'h00;

                stm_sel_low = 1;
                stm_sel_high = 0;

                // TX Init
                stm_tx_data = 8'h40;
                stm_wrt_tx = 0;

                // RX Init
                stm_rd_rx = 0;

                // Load Low Buffer
```

```
                @(posedge stm_clk);
                stm_rst = 0;
                $display("Low Buffer Loading...");

                // Load High Buffer
                @(posedge stm_clk);
                $display("High Buffer Loading...");
                stm_sel_low = 0;
                stm_sel_high = 1;
                stm_baud_data_in =    (i[1:0] == 2'b00) ? 8'h12 :
                                      (i[1:0] == 2'b01) ? 8'h25 :
                                      (i[1:0] == 2'b10) ? 8'h4b : 8'h96;

                // Send Data (Ready to Receive)
                @(posedge stm_clk);
                stm_sel_high = 0;
                $display("Sending data...");
                // Start sending spart data
                stm_wrt_tx = 1;
                @(posedge stm_clk);
                stm_wrt_tx = 0;

                $display("Waiting for receive signal...");
                @(posedge tb_rda_mon);
                @(posedge stm_clk);

                if(final_data_mon != stm_tx_data) begin
                        flags = (i[1:0] == 2'b00) ? flags & 4'b1110 :
                                (i[1:0] == 2'b01) ? flags & 4'b1101 :
                                (i[1:0] == 2'b10) ? flags & 4'b1011 : flags & 4'b0111;
                end
        end

        if(&flags)
                $display("All tests passed!");
        else
                $display("Some tests failed, check flags: %h", flags);

        $stop();
/*
        if(final_data_mon != stm_tx_data)
                $display("An error occured");
        else
                $display("Test passed!");
        $finish();

        */

end
endmodule
```