Benjamin Hunt
815841735
#2 = Wednesday

Lab5 – UART Tx/Rx

Progress: I have demonstrated the Tx transmitter on the scope. I have also demonstrated the Tx and Rx to the 8 LEDs, but with a slight hiccup. I couldn't figure out why, but my last LED was not turning on or off. The dip switches corresponded to the correct LEDs without a problem, but the last one IO_LED[7] would never turn on.

Conclusion: At first, I started the Rx portion trying to be clever, by sending my data to a different pin and using as little case statements as possible to make the code compact**. I found come code on the internet and tried to model my code off of it. I ended up getting confused, because the example utilized a bunch of different timers, so I moved to a simpler method. I still used a different pin for Rx, but I just tried a bunch of case statements. This still wasn't working, so some peers in lab suggested I just use the same pin, which really simplifies the Rx code, by taking the output bits, and just flipping them into the Rx always block, which changes the LEDs on/off. After all this, I found my fatal flaw. In my always block to set the new Rx timer ('rx_new_clk'), I had set my counter to 'rx_new_clk', instead of 'rx_counter'. So, my 'rx_new_clk' was just counting to one everytime, which is obviously too fast to do anything.

If I'd had more time, I would have tried to get it working with my original idea; the compact code with just a few case statements. I think I could probably get it to work now, that my clock is actually correct.

** I have kept all 3 different methods in the Verilog file

```
module ioTest(input  M_CLOCK,
                                    input  [3:0] IO_PB,       // IO Board Pushbutton Switches
                                input  [7:0] IO_DSW,      // IO Board Dip Switchs
                                    input RX,
                                    output [3:0] F_LED,       // FPGA LEDs
                                output reg [7:0] IO_LED,  // IO Board LEDs
                                    output [3:0] IO_SSEGD,    // IO Board Seven Segment
Digits
                                    output [7:0] IO_SSEG,     //7=dp, 6=g, 5=f,4=e,
3=d,2=c,1=b, 0=a
                                    output IO_SSEG_COL,       // Seven segment column
                                    output DEC_POINT,
                                    output reg POINT);

        //==================================
        //Variable declaration

assign IO_SSEG_COL = 1; // deactivate the column displays
assign DEC_POINT = 1;   // deactivate the the decimal point of the seven segment
assign IO_SSEG = 8'b11111111; // deactivate the seven segment display
//assign IO_LED = 8'b00000000; // deactivate the IO board leds
assign IO_SSEGD = 4'b1111; // deactivate the decimal points
assign F_LED = 4'b0000; // deactivate the fpga board leds

/////TX
reg [1:0] tx_state = 0; // state to change --> 0,1,2
reg [2:0] count = 0; //counts 0-7, used for sending [7:0]data
reg [7:0] data = 0; // data bits given by the dip switches

/////RX
reg [2:0] state = 0; // state to change --> 0,1,2,3,4
reg [7:0] r_Clock_Count = 0;
reg [2:0] counter = 0; //8 bits total
reg [7:0] LED_bits = 0;



        // generate new clock: TX
reg [12:0] clk_count = 0; //counter up to 8192, enough to cover the new 50Mhz/9600 clock
reg new_clk = 0; //clock starts at 0

always @(posedge M_CLOCK) begin //count each posedge of the fpga clock
        if (clk_count < 5208) begin //count until M_CLOCK reaches 5208
                clk_count <= clk_count + 1;
```

```
        end
        else begin //once the counter value is reached, reset counter and toggle new_clk
                clk_count <= 0;
                new_clk <= ~new_clk; //toggles from 0 to 1, or 1 to 0
                //used in the always loops as the new clock speed
  end
end

// generate new clock: RX
reg [12:0] rx_clk_count = 0; //counter up to 8192, enough to cover the new 50Mhz/9600 clock
reg rx_new_clk = 0; //clock starts at 0

always @(posedge M_CLOCK) begin //count each posedge of the fpga clock
        if (rx_clk_count < 651) begin //count until M_CLOCK reaches 5208
                rx_clk_count <= rx_clk_count + 1;
        end
        else begin //once the counter value is reached, reset counter and toggle new_clk
                rx_clk_count <= 0;
                rx_new_clk <= ~rx_new_clk; //toggles from 0 to 1, or 1 to 0
                //used in the always loops as the new clock speed
  end
end

//I'm still not sure why this didn't work, reading the push button in its own always block
//always @(posedge new_clk) begin
//        if (IO_PB[0] == 0) begin //if the pushbutton is down
//                data <= IO_DSW; //the 8 dip switch positions are loaded into the data
//                state <= 0; //the state is changed from original to 0, or 2 to 0 if the case statements
have already been executed
//        end
//end


//////TX/RX
always @(posedge new_clk) begin //using the generated clock speed
        if (IO_PB[0] == 0) begin //if the pushbutton is down
                data <= IO_DSW; //the 8 dip switch positions are loaded into the data
                tx_state <= 0; //the state is changed from original to 0, or 2 to 0 if the case
statements have already been executed
        end
        else begin
                case (tx_state)
                        0 : begin //state zero
```

```
                                        POINT <= 0; //send the start bit of 0 first, before sending
data
                                        count <= 0; //restart count here to that it doesnt reset before
the LEDs are on
                                        tx_state <= 1; //change state

                                end
                        1 : begin //state 1, sends the data to the pin 1 bit at a time, using the
counter
                                        POINT <= data[count]; //send the bits

                                        if (count == 7) begin //check to see if all 8 bits have been
sent, 0-7
                                                count <= count + 1; //increment to 8, then subract 1
in Rx loop
                                                tx_state <= 2; //if transfer complete, change to next
state
                                        end
                                        else begin //if the transfer is not complete, increment
counter to send next bit
                                                count <= count + 1;
                                                tx_state <= 1; //keep current state
                                        end
                                end

                        2 : begin
                                        POINT <= 1; //send end bit

                                end
                        default : POINT <= 1; //keep the output high
                endcase
        end
end

always@(posedge rx_new_clk) begin //turn the LEDs on
        IO_LED[count - 1] <= POINT;  // ******still not getting the last LED on******
end

/////RX
//always @(posedge rx_new_clk) begin //using the generated clock speed
//                case (state)
//                        0 : begin //state zero
```

```
//                                      if (RX == 0) begin //send the start bit of 0 first, before
sending data
//                                              state <= 1; //change state
//                                      end
//                                      else begin
//                                              state <= 0;
//                                      end
//                                  end
//                          1 : begin //state 1, sends the data to the pin 1 bit at a time, using the
counter
//                                      IO_LED[counter - 1] <= RX; //send the bits
//
//                                      if (counter == 7) begin //check to see if all 8 bits have been
sent, 0-7
//                                              state <= 2; //if transfer complete, change to next
state
//                                      end
//                                      else begin //if the transfer is not complete, increment
counter to send next bit
//                                              counter <= counter + 1;
//                                              state <= 1; //keep current state
//                                      end
//                                  end
//
//                          2 : begin
//                                      counter <= 0; //reset counter
//                                      state <= 0;
//                                  end
//                          default : state <= 0; //keep the output high
//                  endcase
//end
//
/////RX
//always @(posedge rx_new_clk)
//  begin
//
//    case (state)
//      0 :
//        begin
//          r_Clock_Count <= 0;
//          counter   <= 0;
//
//          if (RX == 1'b0) // Start bit detected
```

```
//           state <= 1;
//         else
//           state <= 0;
//       end
//
//     // Check middle of start bit to make sure it's still low
//     1 :
//       begin
//         if (r_Clock_Count == (CLKS_PER_BIT-1)/2)
//         begin
//           if (RX == 1'b0)
//           begin
//             r_Clock_Count <= 0;  // reset counter, found the middle
//             state     <= 2;
//           end
//           else
//             state <= 0;
//         end
//         else
//         begin
//           r_Clock_Count <= r_Clock_Count + 1;
//           state     <= 1;
//         end
//       end // case: RX_START_BIT
//
//
//     // Wait CLKS_PER_BIT-1 clock cycles to sample serial data
//     2 :
//       begin
//         if (r_Clock_Count < CLKS_PER_BIT-1)
//         begin
//           r_Clock_Count <= r_Clock_Count + 1;
//           state <= 2;
//         end
//         else
//         begin
//           r_Clock_Count         <= 0;
//           LED_bits[counter] <= RX;
//
//           // Check if we have received all bits
//           if (counter < 7)
//           begin
//             counter <= counter + 1;
```

```
//             state   <= 2;
//           end
//           else
//           begin
//             counter <= 0;
//             state   <= 3;
//           end
//         end
//       end // case: RX_DATA_BITS
//
//
//     // Receive Stop bit.  Stop bit = 1
//     3 :
//       begin
//         // Wait CLKS_PER_BIT-1 clock cycles for Stop bit to finish
//         if (r_Clock_Count < CLKS_PER_BIT-1)
//         begin
//           r_Clock_Count <= r_Clock_Count + 1;
//              state     <= 3;
//         end
//         else
//         begin
//           r_Clock_Count <= 0;
//           state     <= 0;
//         end
//       end // case: RX_STOP_BIT
//
//     default :
//       state <= 0;
//
//   endcase
//  end

//always@(posedge rx_new_clk) begin
//         case(state)
//                  0 : begin
//                                      if (RX == 0) begin
//                                              state <= 1;
//                                      end
//                                      else begin
//                                              state <= 0;
//                                      end
//                          end
```

```
//                1 : begin
//                                LED_bits[0] <= RX;
//                                state <= 2;
//                                 end
//                2 : begin
//                                LED_bits[1] <= RX;
//                                state <= 3;
//                                end
//                3 : begin
//                                LED_bits[2] <= RX;
//                                state <= 4;
//                                end
//                4 : begin
//                                LED_bits[3] <= RX;
//                                state <= 5;
//                                end
//                5 : begin
//                                LED_bits[4] <= RX;
//                                state <= 6;
//                                end
//                6 : begin
//                                LED_bits[5] <= RX;
//                                state <= 7;
//                                end
//                7 : begin
//                                LED_bits[6] <= RX;
//                                state <= 8;
//                                end
//                8 : begin
//                                LED_bits[7] <= RX;
//                                state <= 9;
//                                end
//                9 : begin
//                                state <= 0;
//                                end
//                default:
//                                state <= 0;
//                endcase
//        end


//always@(posedge M_CLOCK)begin
//        IO_LED <= LED_bits;
```

```
//end
//



endmodule
```