Lab 3 UART with Even Parity Bit
CSE 521 FPGA Design
Aaron Chamberlain, ID: 003392908

I Goals:

Serial communication, although by design is a very old communication protocol, is still widely used in many specific fields that are aided by computer design. Multi-meters, industrial tools, and even television broadcast equipment all continue to use the Serial communication method. Thus notably, it is a very common design paradigm that may need to be reproduced in a number of different chips and as a part of larger electrical designs. The goal of this lab is to learn the internal workings of the UART communication design by producing a working UART that uses at a minimum even or odd parity for error checking. In this design, many other smaller topics become important, such as syncing the bit transport stream to a clock so that all bits are signaled for a controlled amount of time.

II Design Process:

Given the obvious timing design constraints mentioned in the goals section above, the design will begin with a graph of the internal functional units. Given that we will be updating data registers over time, this design paradigm is most easily shown using an ASMD. Before we go about designing the mechanism for the creating our serial signals, we must first define the protocol communication standard. Most communication standards have very well defined signal bits and timing constraints, and the Serial protocol is one of them. So first, we will observe the packet standard as seen in (Figure 1).

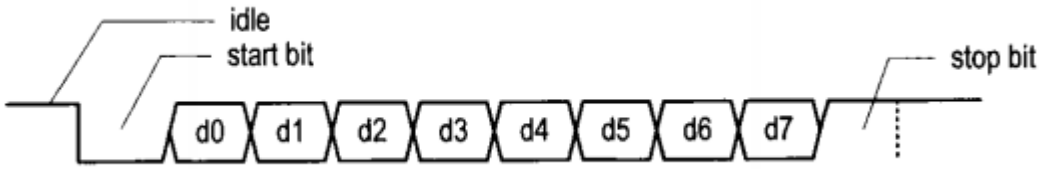

Figure 1. The packet design and timing of Serial Communication

When in the idle state, we continue to simply send a stable logic level of one. When a packet is ready to be transmitted or received, the signal is switched to zero. In this way we recognize the negative edge of the signal and trigger the rest of the transmission circuit. When the start bit is detected, we begin a counter that is used to control the data sampling period. We count up to 7, and then begin to count from 0 to 15 again. In this way, each time the counter reaches 15, the data signal will be immediately in the middle of it's transmission. This is a very effective way of compensating for skew, or the signaling latency present in the circuit.

Now we begin to reach the data section of the protocol. The data is an 8 bit representation of the value being sent, which means that we can only send values from 0 to 255 in decimal, but this is plenty sufficient if we handle the receipt of this data in software where combinations can be chained logically to derive more complex circuits. We simply take the value that we wish to transmit and send it bit by bit. As soon as the data has been completely sent, we signal a Logic level high to represent the stop bit, which will conveniently allow us to indefinitely remain in the idle state again until data transmission resumes.

Now that basics of the UART have been discussed without parity, we may introduce the subject of data parity and how it will be added to the previous circuit design. Data parity is one of many hardware methods of compensating for random signal degradation. This is sometimes in modern applications, but is extremely important in scientific or industrial applications. Data parity comes in two modes, odd or even parity and is a single additional bit added directly after the data. The parity bit

represents the number of ones present in the data section of the transmission. As the transmitter is sending its signal, it calculates the parity that it will send. The receiving computer will also calculate the expected parity bit, and then compare the expected value against the actual value that was sent. If the values do not match, then we may assume that the data was incorrectly transmitted.

In even parity mode, the parity bit will be set to zero if an even number of ones is present in the data signal. In odd parity, the parity bit will be set to zero if an odd number of ones is present. Note that this bit is active low, so that if added immediately after data it will allow us to switch to the regular active high stop bit without changing the rest protocol design. So for example, if we transmit the value 85 in decimal, or 01010101, then in even parity mode, the parity bit will be set to zero. In an odd parity mode, this same signal will set the parity bit to one, because an odd number of ones is not present.

Finally, now that we discussed the details of the UART system, we can split the design of the transmission and receiving circuits into two smaller modules. The first of these modules is broken down in an ASMD below (see Figure 2).

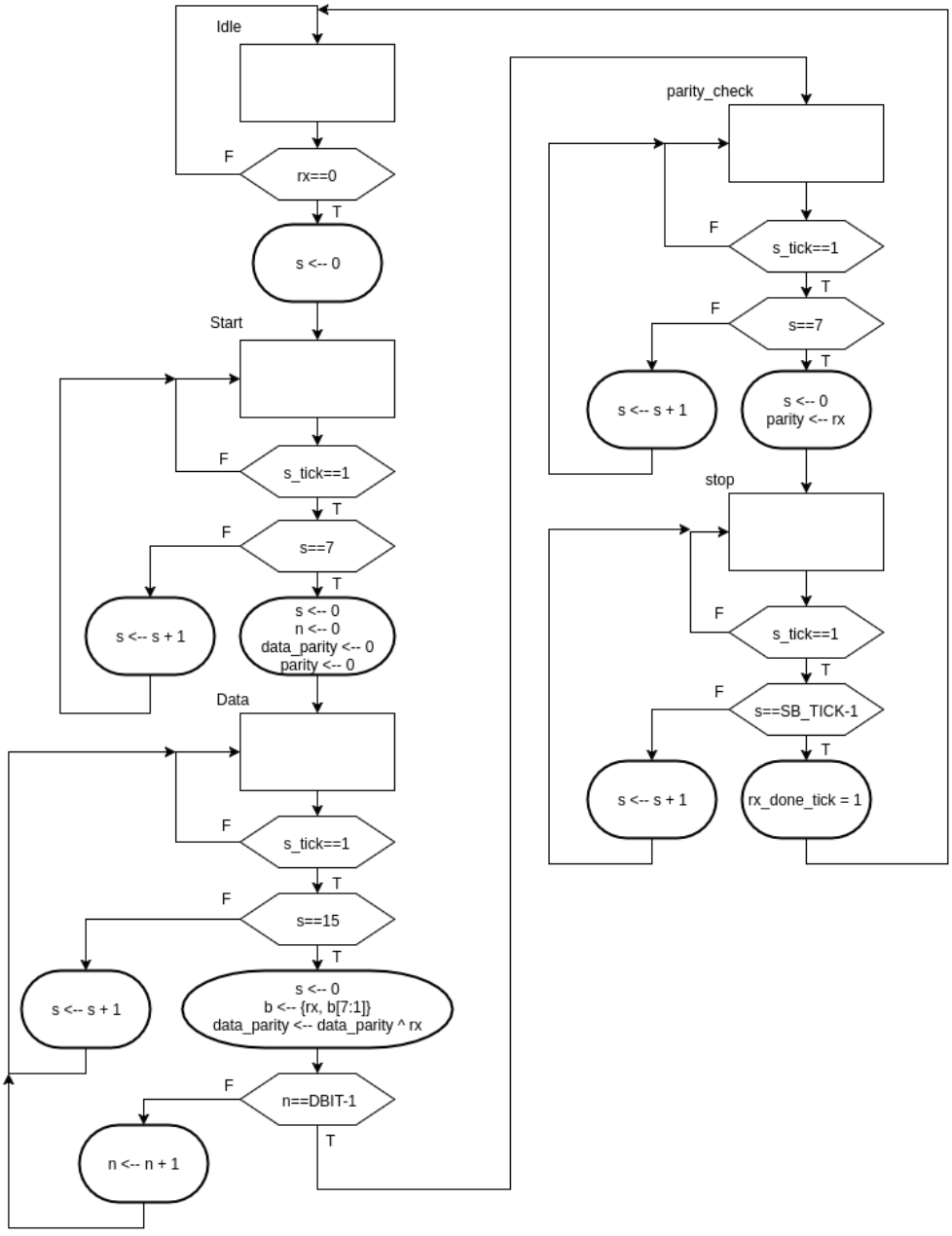

Figure 2. ASMD of UART receiver with Parity Bit

With the diagram in place, we can begin to validate the design and verify that it will perform the functions that we desire of the UART. We start in the idle state that sends out a constant HIGH logic level, when the rx signal of 0 is received, it means that we have received the start bit, so we begin by setting the s state to zero. The s state controls the bit timing counter.

Upon entering into the Start state we check if ‘s_tick’ is equal to one, which is signaled by the mod-m counter, wherein the signal is always 1 except for when the counter has reached it’s maximum value. We then check the value of ‘s’ to ensure that it has reached half way through our defined data period. If it has not, then we increment and go again. If not, we reset all the datapath registers in preparation for entering into that state. The use of ‘s’ to count the data signal duration compensates for the effect of skew, as discussed above.

In the data state, we perform the same logical checks to see if the counter has reached the middle of the data signal, in which case, we reset the signal counter, and make the value of ‘b’ the bit we have received concatenated with the data we already have. We then check to see how many data bits we have sent out using the value ‘n’, in our case, note the calculation of the parity bit, where the data parity value is changed based off of the current bit received. At this point we are ready to move to Parity Check state. In this state, we perform similar logic check to see if we are in the middle of the receipt signal, at which point we set the parity register to the value that was transmitted to us. At another location in the application we can then compare the values of ‘data_parity’ and ‘parity’ to verify that the data was transmitted correctly. Finally, we transfer to the Stop state, which simply checks to see if we are at the correct time cycle of our bit stream. If that is the case, then we can simply jump to the idle state, otherwise, we stay to ensure proper timing remains before moving on.

After this design was performed, we could move on to the very similar transmitting ASMD design, which is seen below in (Figure 3). Only the differences between it and the receiver will be noted for brevity.

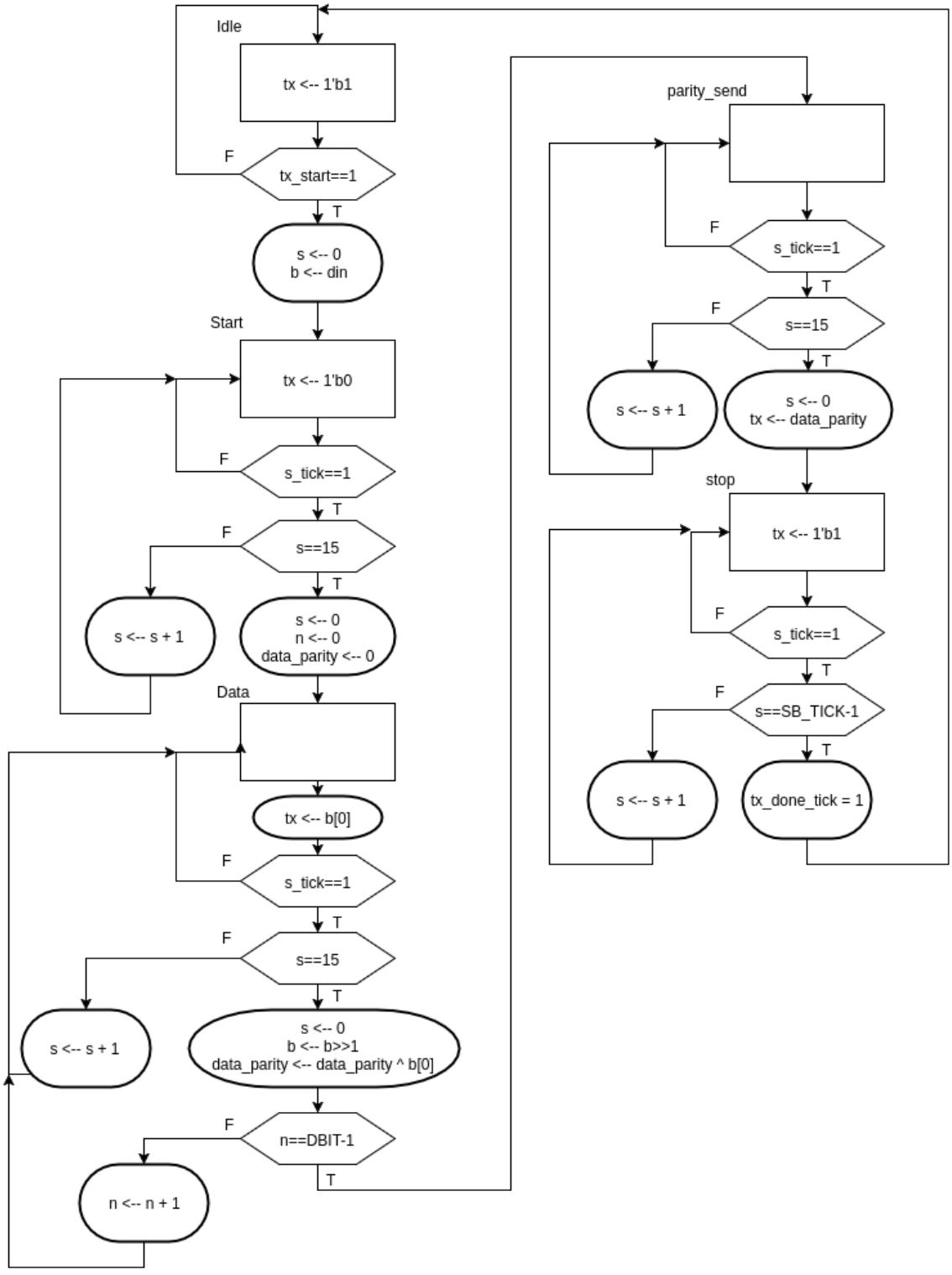

Figure 3. ASMD for the Transmitting Circuit with Even Parity Bit.

The first thing that may be noted is that in the transmitter circuit, we have several states where the output is based off of the current state. In the idle state, we simply continue to send a logic level HIGH to indicate we are still in idle to the receiver. When the start flag is finally asserted for the transmitter, we reset out state data counter to zero, and set ‘b’ to the value that is to be transmitted.

After this is done, we move into the Start state, wherein we perform the regular checks to ensure we are in the middle of the data flag, we reset ‘s’ and ‘n’ again, and also reset out ‘data_parity’ bit and immediately move into the Data state. In the data state, we move the Least Significant Bit into the transmitting bit, check to ensure we are in the middle of the data clock period, in which case we reset ‘s’, shift ‘b’ to the left in preparation for sending the next bit, and calculate the next value of ‘data_parity’. Once all bits have been sent, then we move onto the Parity Send state.

In the Parity Send state, we simply check to see if the data clock is at the middle of the data, reset ‘s’ and then set the transmitting bit to the value of ‘data_parity’ previously calculated. After this we simply move onto the Stop state where we assert that we have reached the correct point for us to switch back to the idle state, and send a flag to indicate we are done.

One final design element involved in this UART is the use of a FIFO buffer to allow the entire system to send values quicker. The FIFO buffer is implemented suck that we have 4 slots that will accept data. When a value is sent on din, the value is not immediately sent, but rather gets placed inside of the buffer. Once the buffer has had all slots filled, we will not fill it any more, and a signal is asserted so that we can ensure that it is functioning correctly. Only at the push of one of the designated buttons, will the first value be popped from the buffer and sent to the UART transmitter itself, wherein the regular functions of the circuit as described above will be carried out.

III Program:

mod_m_counter:

```
module mod_m_counter
  #(
   parameter N=4, // number of bits in counter
             M=10 // mod-M
  )
  (
   input wire clk, reset,
   output wire max_tick,
   output wire [N-1:0] q
  );

  //signal declaration
  reg [N-1:0] r_reg;
  wire [N-1:0] r_next;

  // body
  // register
  always @(posedge clk, posedge reset)
    if (reset)
      r_reg <= 0;
    else
      r_reg <= r_next;

  // next-state logic
  assign r_next = (r_reg==(M-1)) ? 0 : r_reg + 1;
  // output logic
  assign q = r_reg;
  assign max_tick = (r_reg==(M-1)) ? 1'b1 : 1'b0;
```

```
endmodule
```

UART_rx:

```
module uart_rx
  #(
    parameter DBIT = 8,     // # data bits
              SB_TICK = 16  // # ticks for stop bits
  )
  (
   input wire clk, reset,
   input wire rx, s_tick,
   output reg rx_done_tick,
   output wire [7:0] dout,
           output reg dataParity_reg,
           output wire parity
  );

  // symbolic state declaration
  localparam [2:0]
     idle  = 3'b000,
     start = 3'b001,
     data  = 3'b010,
     parityCheck  = 3'b011,
                    stop = 3'b100;

  // signal declaration
  reg [2:0] state_reg, state_next;
  reg [3:0] s_reg, s_next;
  reg [2:0] n_reg, n_next;
  reg [7:0] b_reg, b_next;
          reg dataParity_next;
          reg parity_reg, parity_next;

  // body
  // FSMD state & data registers
  always @(posedge clk, posedge reset)
     if (reset)
        begin
          state_reg <= idle;
          s_reg <= 0;
          n_reg <= 0;
          b_reg <= 0;
                                    dataParity_reg <= 0;
                                    parity_reg <= 0;
        end
     else
        begin
          state_reg <= state_next;
          s_reg <= s_next;
```

```
        n_reg <= n_next;
        b_reg <= b_next;
                                dataParity_reg <= dataParity_next;
                                parity_reg <= parity_next;
    end

// FSMD next-state logic
always @*
begin
  state_next = state_reg;
  rx_done_tick = 1'b0;
  s_next = s_reg;
  n_next = n_reg;
  b_next = b_reg;
                dataParity_next = dataParity_reg;//needed?
                parity_next = parity_reg;
  case (state_reg)
    idle:
      if (~rx)
        begin
          state_next = start;
          s_next = 0;
                                                dataParity_next = 0;
        end
    start:
      if (s_tick)
        if (s_reg==7)
          begin
            state_next = data;
            s_next = 0;
            n_next = 0;
          end
        else
          s_next = s_reg + 1;
    data:
      if (s_tick)
        if (s_reg==15)
          begin
            s_next = 0;
            b_next = {rx, b_reg[7:1]};
                                                    dataParity_next = dataParity_reg ^ rx;
            if (n_reg==(DBIT-1))
              state_next = parityCheck;
             else
              n_next = n_reg + 1;
           end
        else
          s_next = s_reg + 1;
                    parityCheck:
```

```
                        if (s_tick)
                            if (s_reg==15)
                                begin
                                    s_next = 0;
                                    parity_next = rx;
                                    state_next = stop;
                                end
                            else
                                s_next = s_reg + 1;
        stop:
          if (s_tick)
            if (s_reg==(SB_TICK-1))
              begin
                state_next = idle;
                rx_done_tick =1'b1;
              end
            else
              s_next = s_reg + 1;
      endcase
    end
    // output
    assign dout = b_reg;
          assign parity = parity_reg;
endmodule
```

FIFO:

```
module fifo
  #(
   parameter B=8, // number of bits in a word
            W=4  // number of address bits
  )
  (
   input wire clk, reset,
   input wire rd, wr,
   input wire [B-1:0] w_data,
   output wire empty, full,
   output wire [B-1:0] r_data
  );

  //signal declaration
  reg [B-1:0] array_reg [2**W-1:0];  // register array
  reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
  reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
  reg full_reg, empty_reg, full_next, empty_next;
  wire wr_en;

  // body
  // register file write operation
  always @(posedge clk)
```

```
  if (wr_en)
    array_reg[w_ptr_reg] <= w_data;
// register file read operation
assign r_data = array_reg[r_ptr_reg];
// write enabled only when FIFO is not full
assign wr_en = wr & ~full_reg;

// fifo control logic
// register for read and write pointers
always @(posedge clk, posedge reset)
  if (reset)
    begin
      w_ptr_reg <= 0;
      r_ptr_reg <= 0;
      full_reg <= 1'b0;
      empty_reg <= 1'b1;
    end
  else
    begin
      w_ptr_reg <= w_ptr_next;
      r_ptr_reg <= r_ptr_next;
      full_reg <= full_next;
      empty_reg <= empty_next;
    end

// next-state logic for read and write pointers
always @*
begin
  // successive pointer values
  w_ptr_succ = w_ptr_reg + 1;
  r_ptr_succ = r_ptr_reg + 1;
  // default: keep old values
  w_ptr_next = w_ptr_reg;
  r_ptr_next = r_ptr_reg;
  full_next = full_reg;
  empty_next = empty_reg;
  case ({wr, rd})
    // 2'b00:  no op
    2'b01: // read
      if (~empty_reg) // not empty
        begin
          r_ptr_next = r_ptr_succ;
          full_next = 1'b0;
          if (r_ptr_succ==w_ptr_reg)
            empty_next = 1'b1;
        end
    2'b10: // write
      if (~full_reg) // not full
        begin
```

```
            w_ptr_next = w_ptr_succ;
            empty_next = 1'b0;
            if (w_ptr_succ==r_ptr_reg)
              full_next = 1'b1;
          end
        2'b11: // write and read
          begin
            w_ptr_next = w_ptr_succ;
            r_ptr_next = r_ptr_succ;
          end
      endcase
    end

    // output
    assign full = full_reg;
    assign empty = empty_reg;

endmodule
```

UART_tx:

```
module uart_tx
  #(
    parameter DBIT = 8,     // # data bits
              SB_TICK = 16  // # ticks for stop bits
  )
  (
   input wire clk, reset,
   input wire tx_start, s_tick,
   input wire [7:0] din,
   output reg tx_done_tick,
   output wire tx
  );

  // symbolic state declaration
  localparam [2:0]
     idle  = 3'b000,
     start = 3'b001,
     data  = 3'b010,
     paritySend = 3'b011,
                  stop = 3'b100;

  // signal declaration
  reg [2:0] state_reg, state_next;
  reg [3:0] s_reg, s_next;
  reg [2:0] n_reg, n_next;
  reg [7:0] b_reg, b_next;
  reg tx_reg, tx_next;
        reg dataParity_reg, dataParity_next;
```

```
// body
// FSMD state & data registers
always @(posedge clk, posedge reset)
  if (reset)
    begin
      state_reg <= idle;
      s_reg <= 0;
      n_reg <= 0;
      b_reg <= 0;
      tx_reg <= 1'b1;
                                dataParity_reg <= 0;
    end
  else
    begin
      state_reg <= state_next;
      s_reg <= s_next;
      n_reg <= n_next;
      b_reg <= b_next;
      tx_reg <= tx_next;
                                dataParity_reg <= dataParity_next;
    end

// FSMD next-state logic & functional units
always @*
begin
  state_next = state_reg;
  tx_done_tick = 1'b0;
  s_next = s_reg;
  n_next = n_reg;
  b_next = b_reg;
  tx_next = tx_reg;
              dataParity_next = dataParity_reg;
  case (state_reg)
    idle:
      begin
        tx_next = 1'b1;
        if (tx_start)
          begin
            state_next = start;
            s_next = 0;
            b_next = din;
          end
      end
    start:
      begin
        tx_next = 1'b0;
        if (s_tick)
          if (s_reg==15)
            begin
```

```
            state_next = data;
            s_next = 0;
            n_next = 0;
          end
        else
          s_next = s_reg + 1;
    end
  data:
    begin
      tx_next = b_reg[0];
      if (s_tick)
        if (s_reg==15)
          begin
            s_next = 0;
                                                    dataParity_next = dataParity_reg ^
b_reg[0]; //changed
            b_next = b_reg >> 1;
            if (n_reg==(DBIT-1))
              state_next = paritySend;
            else
              n_next = n_reg + 1;
          end
        else
          s_next = s_reg + 1;
    end
                paritySend:
                        begin
                                if (s_tick)
                                        if(s_reg==15)
                                                begin
                                                        s_next = 0;
                                                        tx_next = dataParity_reg;
                                                        state_next = stop;
                                                end
                                        else
                                                s_next = s_reg + 1;
                                end
  stop:
    begin
      tx_next = 1'b1;
      if (s_tick)
        if (s_reg==(SB_TICK-1))
          begin
            state_next = idle;
            tx_done_tick = 1'b1;
          end
        else
          s_next = s_reg + 1;
    end
```

```
    endcase
  end
  // output
  assign tx = tx_reg;

endmodule
```

Debouncer:

```
module debounce
  (
   input wire clk, reset,
   input wire sw,
   output reg db_level, db_tick
  );

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

  // number of counter bits (2^N * 20ns = 40ms)
  localparam N=21;

  // signal declaration
  reg [N-1:0] q_reg, q_next;
  reg [1:0] state_reg, state_next;

  // body
  // fsmd state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
        begin
          state_reg <= zero;
          q_reg <= 0;
        end
      else
        begin
          state_reg <= state_next;
          q_reg <= q_next;
        end

  // next-state logic & data path functional units/routing
  always @*
  begin
    state_next = state_reg;   // default state: the same
    q_next = q_reg;           // default q: unchnaged
    db_tick = 1'b0;           // default output: 0
```

```
case (state_reg)
  zero:
    begin
      db_level = 1'b0;
      if (sw)
        begin
          state_next = wait1;
          q_next = {N{1'b1}}; // load 1..1
        end
    end
  wait1:
    begin
      db_level = 1'b0;
      if (sw)
        begin
          q_next = q_reg - 1;
          if (q_next==0)
            begin
              state_next = one;
              db_tick = 1'b1;
            end
        end
      else // sw==0
        state_next = zero;
    end
  one:
    begin
      db_level = 1'b1;
      if (~sw)
        begin
          state_next = wait0;
          q_next = {N{1'b1}}; // load 1..1
        end
    end
  wait0:
    begin
      db_level = 1'b1;
      if (~sw)
        begin
          q_next = q_reg - 1;
          if (q_next==0)
            state_next = zero;
        end
      else // sw==1
        state_next = one;

    end
  default: state_next = zero;
endcase
```

```
  end

endmodule
```

UART:

```
module uart
  #( // Default setting:
    // 19,200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
    parameter DBIT = 8,     // # data bits
              SB_TICK = 16, // # ticks for stop bits, 16/24/32
                            // for 1/1.5/2 stop bits
              DVSR = 163,   // baud rate divisor
                            // DVSR = 50M/(16*baud rate)
              DVSR_BIT = 8, // # bits of DVSR
              FIFO_W = 2    // # addr bits of FIFO
                            // # words in FIFO=2^FIFO_W
  )
  (
   input wire clk, reset,
   input wire rd_uart, wr_uart, rx,
   input wire [7:0] w_data,
   output wire tx_full, rx_empty, tx,
   output wire [7:0] r_data,
          output wire rx_dataParity,
          output wire rx_parity
  );

  // signal declaration
  wire tick, rx_done_tick, tx_done_tick;
  wire tx_empty, tx_fifo_not_empty;
  wire [7:0] tx_fifo_out, rx_data_out;

  //body
  mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
    (.clk(clk), .reset(reset), .q(), .max_tick(tick));

  uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
    (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
     .rx_done_tick(rx_done_tick), .dout(rx_data_out),
                  .dataParity_reg(rx_dataParity), .parity(rx_parity));

  fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
    (.clk(clk), .reset(reset), .rd(rd_uart),
     .wr(rx_done_tick), .w_data(rx_data_out),
     .empty(rx_empty), .full(), .r_data(r_data));

  fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
    (.clk(clk), .reset(reset), .rd(tx_done_tick),
     .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
```

```
      .full(tx_full), .r_data(tx_fifo_out));

   uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
     (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
      .s_tick(tick), .din(tx_fifo_out),
      .tx_done_tick(tx_done_tick), .tx(tx));

   assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

UART_Test:

```
module uart_test
   (
    input wire clk, reset,
    input wire rx,
    input wire [2:0] btn,
    output wire tx,
    output wire [3:0] an,
    output wire [7:0] sseg, led,
           output wire dataParity,
           output wire parity
   );

   // signal declaration
   wire tx_full, rx_empty, btn_tick;
   wire [7:0] rec_data, rec_data1;

   // body
   // instantiate uart
   uart uart_unit
     (.clk(clk), .reset(reset), .rd_uart(btn_tick),
      .wr_uart(btn_tick), .rx(rx), .w_data(rec_data1),
      .tx_full(tx_full), .rx_empty(rx_empty),
      .r_data(rec_data), .tx(tx), .rx_dataParity(dataParity), .rx_parity(parity));
   // instantiate debounce circuit
   debounce btn_db_unit
     (.clk(clk), .reset(reset), .sw(btn[0]),
      .db_level(), .db_tick(btn_tick));
   // incremented data loops back
   assign rec_data1 = rec_data + 1;
   // LED display
   assign led = rec_data;
   assign an = 4'b1110;
   assign sseg = {1'b1, ~tx_full,dataParity,1'b1, ~rx_empty, 1'b1,parity ,1'b1}; //top bar is
dataParityMatch signal

endmodule
```

IV Results:

The design of the circuit was implemented and found to be correct, in that we were able to successfully communicate with the lab computers over using Putty, a Serial Terminal. The buffer successfully filled up to maximum and was able to be emptied completely, with LEDs on the seven segment indicating the state of the FIFO buffer itself. We also bubbled up two of the signals found in the receiver, ‘data_parity’ and ‘parity’ so that we could determine whether the values were in agreement. This functionality was demoed during Lab time to the instructor.

V Problem:

This lab did a good job at combining earlier topics that were introduced in the class with newer topics in order to create a larger and more functional product. The hardest parts of this lab were simply going over the ASMD design and the previously created Verilog code. I spent a good amount of time looking at each piece, particularly noting each internal signal and the logic that controlled it’s signaling. Although I understood the idea of the two different parity modes, I was not exactly sure how that would get placed within the overall design of the system. Once I understood that in the transmitter I only needed one logical parity value that needed to be transmitted whereas in the receiver I actually wanted two states so that I could evaluate what was sent, then I was able to go forward with the design and reprogramming of the circuit with ease.

VI What You Learned:

The things that were learned in this lab go hand in hand with some of the problems that occurred in the lab. First I learned how to better read pre-existing code. This is a useful skill to have, because upon entering a company, that will be one of the first and primary tasks of a job, to familiarize yourself with code that you in no way took part in previously. In this case the documentation was significant, but otherwise I learned to simply start at the top and get a look at how things are wired together, then move all the way down to the bottom. In this way you can find out what signals exist at the lowest levels, which ones get bubbled up, and how each signal level is controlled.

The next most import thing that I learned is that when dealing with a clock signal and Flip Flops that are storing the state of the circuit, we need to set a ‘next’ signal, which is essentially the input of the Flip Flop and allow that value to bubble across on the next cycle. This is a fundamental part of any design with memory because manipulating the signal itself would result in a very unstable signal where the value is being changed as it is being read. This mechanism allows the data or signaling state to be controlled for a known period of time.

Lastly, I added a new method or item to check off when debugging an application, that is to ensure that all values are of the correct width to ensure proper control. Although it was required to add the new states and thus increase the width of the local parameter variable, I forgot to increase the width of the state that would be absorbing this signal. In this case, that meant that the signal levels were simply truncated and that the Most Significant Bit (MSB) was disregarded. Given that the stop state now required a bit to be set in this third bit, that meant that the receiver and transmitter never reached the point in which they could stop, and thus remained in an infinite loop. Thus, checking the widths of all internal signals will now become a regular habit in debugging my future HDL code.