```verilog
`timescale 1ns / 1ps
/****************************************************************************
 * File Name: receiveEngine.v
 * Project:  FULL UART
 * Designer: Marc Dominic Cabote
 * Email: marcdominic011@gmail.com
 * Rev. Date: 5 May, 2018
 *
 * Purpose:The receive engine has the logic needed to communicate with a
 *         convert bits of signals received into data. Using a shift register,
 *         The register is filled with data throu the Rx pin which either
 *         goes high or low. Once the data has been received, It spits out
 *         the data to the tramelblaze to be output in RealTerm.
 *
 * Notes:   -  This module has a asynchronous reset input.
 *
 ****************************************************************************/
module receiveEngine(input clk, rst, Rx, eight, pen, reads0, even,
                     input [18:0] k,
                     output reg RxRdy, perr, ferr, ovf,
                     output [7:0] data);

   wire btu, done, shift, overflow, stop_bit, parity;

   reg start, doit, parity_gen, parity_gen_sel, parity_bit_sel,
       stop_bit_sel;
   reg [1:0] pstate, nstate;
   reg [3:0] bit_count, bit_counter, bits; // bit counter
   reg [9:0] shift_out, combo_out;
   reg [18:0] bit_time_count, bit_time_counter, bit_time;//bit-time counter

   //==================================================================
   // State Machine
   //==================================================================
   always @ (posedge clk, posedge rst)
      begin
         if (rst)
            pstate <= 2'b00;
         else
            pstate <= nstate;
      end

   always @ (*)
      begin
         case (pstate)
            //state 0
            0: begin
               {start, doit} = 2'b00;
               //nstate = Rx ? 2'b00 : 2'b01; //if ~Rx go next
               if(Rx)
                  nstate = 0;
               else
                  nstate = 1;
               end

            //state 1
            1: begin
```

```
                  {start, doit} = 2'b11;
                  //nstate = Rx ? 2'b00: btu ? 2'b10 : 2'b01; //if ~Rx & Btu go next
                  if (Rx)
                     nstate = 0;else
                  if (~Rx && btu)
                     nstate = 1;
                  else
                     nstate = 2;
                  end

               //state 2
               2: begin
                  {start, doit} = 2'b01;
                  //nstate = done ? 2'b00 : 2'b01; //if done go next
                  if (done)
                     nstate = 0;
                  else
                     nstate = 2;
                  end

               //default state 0
               default: begin
                     {start, doit} = 2'b00;
                     nstate = 2'b00;
                     end
            endcase
         end

      //==================================================================
      //Bit Time Counter k select
      //==================================================================

      always @ (*)
         begin
            if (start)
               bit_time = k >> 1;//divide k in half
            else
               bit_time = k;     //else default baud
         end

      //==================================================================
      //Bit Time Counter
      //==================================================================
      assign btu = (bit_time_count == bit_time);
      assign shift = btu & ~start;

      always @(posedge clk, posedge rst)
         begin
            if(rst)
               bit_time_count <= 0;
            else
               bit_time_count <= bit_time_counter;
         end

      always @(*)
         begin
            case ({doit,btu})
```

```
               0: bit_time_counter = 0;
               1: bit_time_counter = 0;
               2: bit_time_counter = bit_time_count + 1;
               3: bit_time_counter = 0;
            endcase
         end

      //=====================================================================
      //Bit Counter Number of Bits
      //=====================================================================
      always @ (*)
         begin
            case ({eight,pen})
               0: bits = 9;
               1: bits = 10;
               2: bits = 10;
               3: bits = 11;
               default: bits = 9;
            endcase
         end

      //=====================================================================
      //Bit Counter
      //=====================================================================
      assign done = (bit_count == bits);

      always @(posedge clk, posedge rst)
         begin
            if(rst)
               bit_count <= 0;
            else
               bit_count <= bit_counter;
         end

      always @(*)
         begin
            case ({doit,btu})
               0: bit_counter = 0;
               1: bit_counter = 0;
               2: bit_counter = bit_count;
               3: bit_counter = bit_count + 1;
            endcase
         end

      //=====================================================================
      // Shift Register
      //=====================================================================
      always @(posedge clk, posedge rst)
         begin
            if (rst)
               shift_out <= 10'b0; else
            if (shift)
               shift_out <= {Rx, shift_out[9:1]};
            else
               shift_out <= shift_out;
         end

```

```
    //==================================================================
    // Remap Combo
    //==================================================================
    always @(*)
       begin
          case ({eight, pen})
             0: combo_out = shift_out >> 2;         //7N1
             1: combo_out = shift_out >> 1;         //7P1
             2: combo_out = shift_out >> 1;         //8N1
             3: combo_out = shift_out;              //8P1
          endcase
       end

    assign data = combo_out [6:0]; //output to the TB

    //==================================================================
    // Parity Gen Select
    //==================================================================
    always @(*)
       begin
          if (eight)
                parity_gen_sel = combo_out[7];
          else
                parity_gen_sel = 0;
       end
    //==================================================================
    // Parity Bit Select
    //==================================================================
    always @(*)
       begin
          if (eight)
             parity_bit_sel = combo_out[8];
          else
             parity_bit_sel = combo_out[7];
       end

    //==================================================================
    // Stop Bit Select
    //==================================================================
    always @(*)
       begin
          case ({eight, pen})
             0: stop_bit_sel = combo_out[7];
             1: stop_bit_sel = combo_out[8];
             2: stop_bit_sel = combo_out[8];
             3: stop_bit_sel = combo_out[9];
          endcase
       end

    //==================================================================
    // RxRdy RS Flop
    //==================================================================
    always @(posedge clk, posedge rst)
        begin
          if(rst)
             RxRdy <= 0;else
          if(done)
```

```
               RxRdy <= 1;else
            if(reads0)
               RxRdy <= 0;
            else
               RxRdy <= RxRdy;
         end

      //==================================================================
      // Parity Error RS Flop
      //==================================================================
      always @(*)
         begin
            if (even)
               parity_gen = parity_gen_sel;
            else
               parity_gen = ~parity_gen_sel;
         end

      assign parity = (parity_gen ^ parity_bit_sel) & pen & done;

      always @ (posedge clk, posedge rst)
         begin
            if (rst)
               perr <= 0; else
            if (parity)
               perr <= 1; else
            if (reads0)
               perr <= 0;
            else
               perr <= perr;
         end

      //==================================================================
      // Framing Error RS Flop
      //==================================================================
      assign stop_bit = done & ~stop_bit_sel;

      always @ (posedge clk, posedge rst)
         begin
            if (rst)
               ferr <= 0; else
            if (stop_bit)
               ferr <= 1; else
            if (reads0)
               ferr <= 0;
            else
               ferr <= ferr;
         end

      //==================================================================
      // Overflow Error RS Flop
      //==================================================================
      assign overflow = RxRdy & done;

      always @ (posedge clk, posedge rst)
         begin
            if (rst)
            ovf <= 0; else
        if (overflow)
            ovf <= 1; else
        if (reads0)
            ovf <= 0;
        else
            ovf <= ovf;
    end

endmodule

```