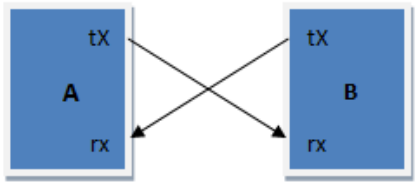
标题: FPGA学习–串口发送模块与验证

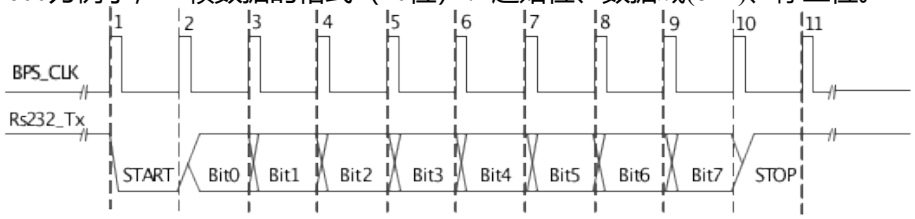
作者: 51hei人人 **时间:** 2016-3-12 22:12
标题: FPGA学习–串口发送模块与验证
本帖最后由 51hei人人 于 2016-3-12 22:14 编辑

一、Rs232串口协议
串口通信指串口按位(bit)发送和接受字节。虽然比并行通信要慢，但是其物理线路简单并且通信距离
长，可达到1200米。
物理连接：



A发送数据时通过tx将数据一位一位的传输给B的rx，表现出来的就是tx线的高低电平，B就可以通过rx来检测高低电平来确定数据1、0。
由于A、B之间并没有时钟线，不能像I2C那样，可以通过时钟为高时检测数据脚的电平状态来确定数
据，那么在串口协议中B应当如何确定何时采集rx端口的电平来作为数据呢？我们常常可以听到波特率为9600或者115200等这些数值，那么B就是通过这个数值来确定何时采集rx端口的电平。

以9600为例子，一帧数据的格式（10位）：起始位、数据域(8bit)、停止位。



9600波特率 --> 9600Hz --> 1/9600(周期) --> 0.00010416666666666667(秒) --> 约 104167(ns)
也就是说A的tx发送的每一个位的数据所保持的时间都必须在 104167ns 这个时间。而B也必须在这个时间内至少采集一次rx的电平状态来得到数据。即A、B双方都是以相同的速度去发送、采集数据。
启动发送时，先将Tx拉低作为启动信号，发送结束后则拉高Tx作为停止信号，空闲时Tx应为高电平状态。
串口发送模块所必须具备的两个部分：
1、波特率的产生
采用计数分频的使能时钟方式产生波特率，那么计数值应如何计算呢。
9600bps 约等于 104167ns，假如系统时钟为 50MHz，那么一个时钟周期为 （1/50）*1000 = 20ns。
104167ns / 20ns = 5208次，即数系统时钟数5208次即为104167ns。

baud_set	波特率	波特率周期	波特率分频计数值 System_clk_period = 20计数值 (从0开始计算所以-1)

0	9600	104167ns	104167/ System_clk_period	5208-1
1	19200	52083ns	52083/ System_clk_period	2604-1
2	38400	26041ns	26041/ System_clk_period	1302-1
3	57600	17361ns	17361/ System_clk_period	868-1
4	115200	8680ns	8680/ System_clk_period	434-1

2、数据发送模块

二、FPGA 程序框图

串口发送模块的端口框图：

输入：

Send_En：发送使能

Data_Byte[7:0]：要发送的数据

Baud_Set[2:0]：波特率选择

Clk：系统时钟

Rst_n：复位信号

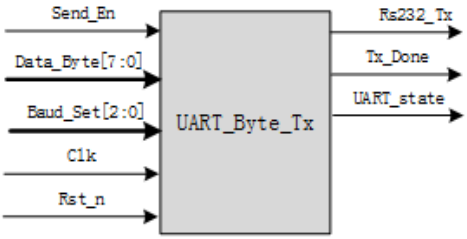
输出：

Rs232_Tx：数据发送引脚

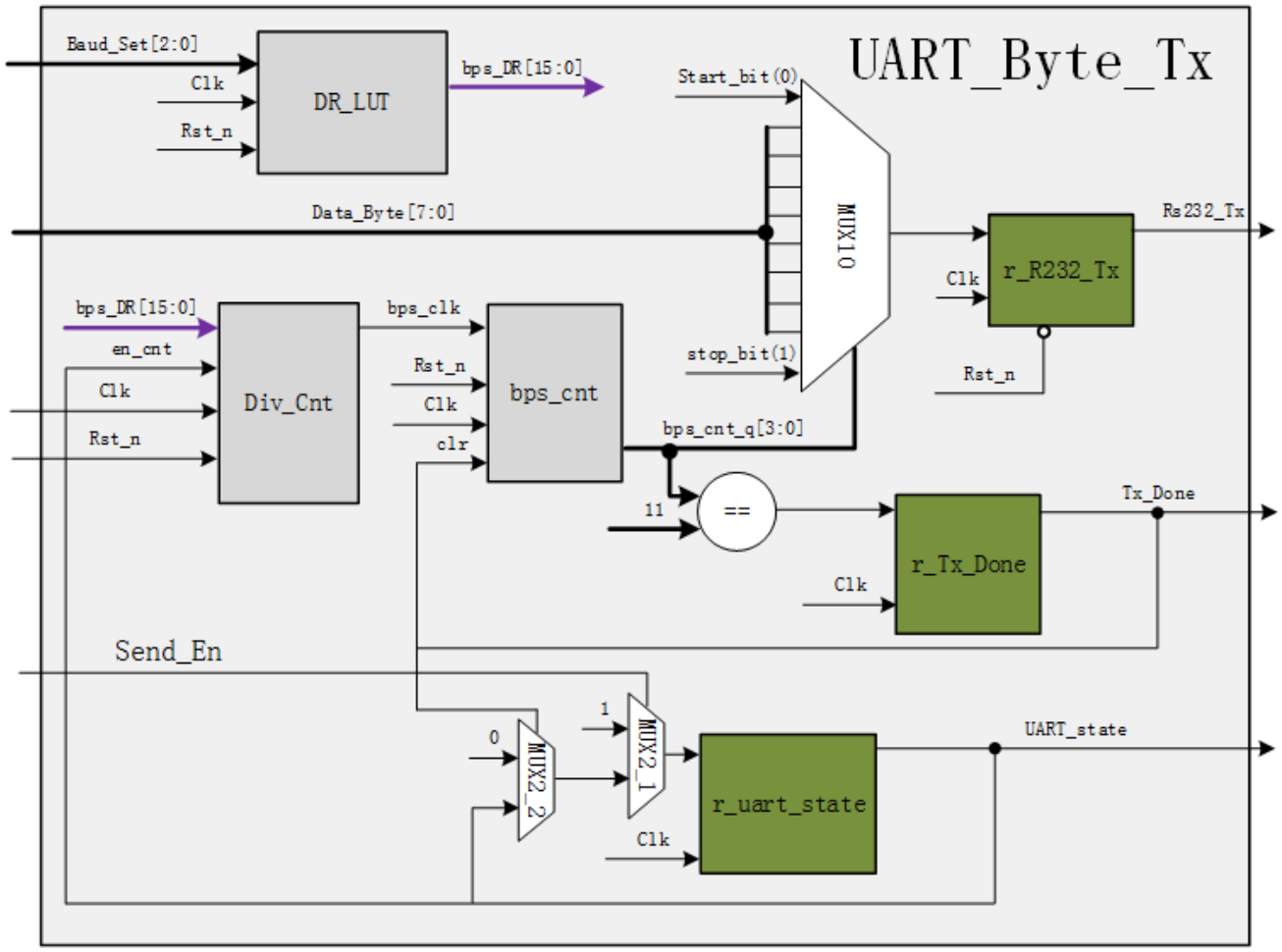
Tx_Done：发送完成通知信号（1：表示发送完成）

UART_state：模块工作状态（1：正在发送数据 0：发送完成或空闲状态）

串口发送模块详细结构图：



功能模块描述:



DR_LUT：查表模块，根据 Baud_Set[2:0] 选择的波特率去查表得到计数分频所需要的计数值即bps_DR[15:0]。

Div_Cnt： 计数分频模块，根据bps_DR[15:0] 来产生bps_clk作为tx发送数据位的节拍，即来一个bps_clk就发送一个数据位。该受en_cnt信号控制，en_cnt 为 0 时Div_Cnt模块不计数，也就不会产生 bps_clk，也就不会发送数据。

bps_cnt：数据位计数模块，对bps_clk进行计数，输出bps_cnt_q[3:0]，用于控制发送的数据位数，完成一帧数据长度的控制，当数到第11个bps_clk时会置高Tx_Done信号。

MUX10：10选1多路器，根据bps_cnt_q[3:0] 来输出起始位、8位数据、停止位来设置 Rs232_Tx信号。其实这里应该是11选1多路器，第0个为输出停止位信号。

MUX2_1、MUX2_2：二选一多路器，用于形成具有优先级的状态控制机制。当Send_En信号为1时，那么MUX2_1就会直接忽略MUX2_2 ，当Send_En信号为0时才会根据MUX2_2的选择来控制UART_state.

整个逻辑控制流程：

1、当Send_En置高一个时钟周期时 [MUX2_1] 输出1到UART_state和en_cnt，此时 UART_state和en_cnt均为1。

下一个时钟来临之后，[MUX2_1] 取 [MUX2_2] 的状态，由于 [MUX2_2] 取自Tx_Done信号，而Tx_Done为0，所以 [MUX2_2] 取的是en_cnt的信号，即UART_state == en_cnt == [MUX2_2] == 1。

只要Tx_Done信号为1，则 [MUX2_2] 就会选择输出0，从而改变 UART_state、en_cnt信号，注意[bsp_cnt] 模块的clr信号也受Tx_Done控制。

2、en_cnt为1触发 [Div_Cnt] 模块工作，[Div_Cnt] 开始以bps_DR[15:0] 所设置的计数间隔输出bps_clk信号。

3、bsp_cnt模块检测到bps_clk，开始数bps_clk个数，并输出bps_cnt_q[3:0] 给 [MUX10] 多路器。bps由于clr信号来自Tx_Done信号，所以clr为0，不会清0计数。若bps_cnt_1[3:0]等于11，即bps_clk的个数为11，则输出1给Tx_Done，出现连锁反应：

1、clr信号变为1：bsp_cnt模块计数清零
2、[MUX2_2] 输出0到 [MUX2_1] 再到UART_start 再到 en_cnt 导致 [Div_Cnt] 停止输出bsp_clk。
3、整个发送模块也就停止发送数据。
4、MUX10：通过视频中所写的代码来看，这里应该是11选1多路器，0为Tx空闲时的状态，即为高电平，1为起始位，2~9为要发送的数据，即Data_Byte[7:0]。10则是停止位。根据bps_cnt_q[3:0]来确定要选择数据帧的哪一个位输出到r_R232_Tx。
5、至此，整个逻辑部分完成。
三、代码实现
代码1：（代码与视频所写的有点不太一样，修改了几句代码是为了尽量符合上面的框图设计）

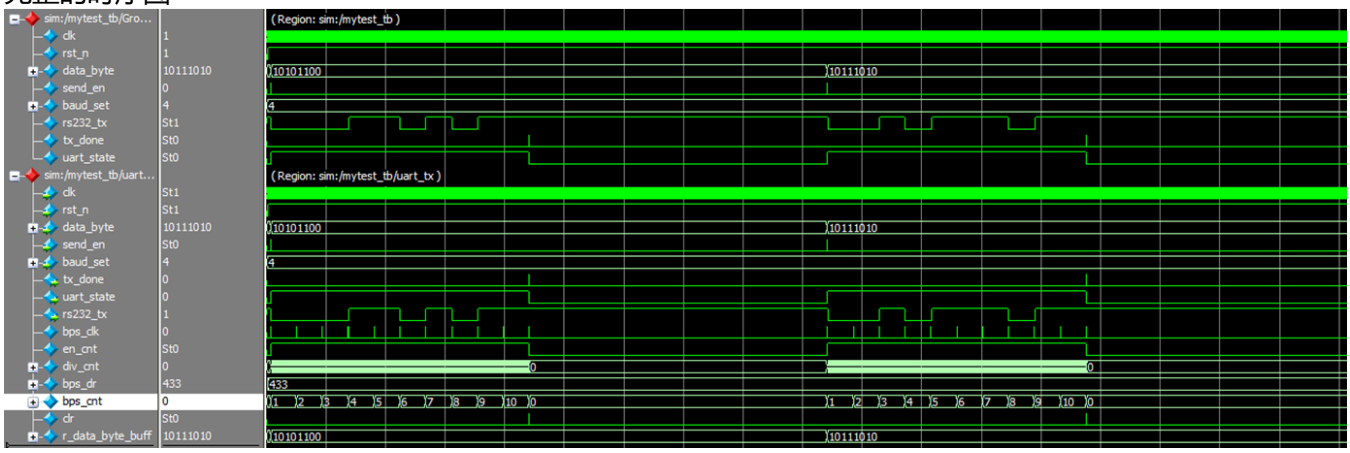
```
module mytest(clk, rst_n, data_byte, send_en, baud_set, rs232_tx, tx_done, uart_state);

    input clk;                          // 系统时钟
    input rst_n;                        // 复位
    input[7:0] data_byte;               // 要发送的数据
    input send_en;                      // 启动发送
    input[2:0] baud_set;                // 波特率选择

    output reg rs232_tx;
    output reg tx_done;                 // 发送完毕通知     1:发送完毕 0:正在发送
    output reg uart_state;              // 发送状态 1:正在发送数据 0:空闲状态

    reg bps_clk;                        // 波特率时钟
    wire en_cnt;                        // 计数使能 1:使能 0:失能
    reg[15:0] div_cnt;                  // 分频计数器
    reg[15:0] bps_dr;                   // 分频计数最大值
    reg[3:0]    bps_cnt;                // 波特率时钟计数器
    wire clr;                           // 清零信号
    reg[7:0] r_data_byte_buff;          // 缓冲区，用于存储需要发送的数据，避免在发送过程中数据突然改变


    localparam START_BIT    = 1'b0;
    localparam STOP_BIN     = 1'b1;


    // 串口工作状态
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            uart_state <= 1'b0;
        else if(send_en)
            uart_state <= 1'b1;
        else if(tx_done)                // 发送完毕
            uart_state <= 1'b0;
        else
            uart_state <= uart_state;

    end

    assign en_cnt = uart_state;

    // 用于启动发送时锁存即将要发送的数据
    // 这样就可以避免在发送的过程中数据突然改变导致发送的数据不正确。
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            r_data_byte_buff <= 8'd0;
        else if(send_en)
            r_data_byte_buff <= data_byte;     // 启动发送则锁存最新的数据
        else
            r_data_byte_buff <= r_data_byte_buff;

    end

    // 【DR_LUT】 通过查表的方式将波特率转换为对应的分频计数最大值
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
```

```
            bps_dr <= 16'd5207;                 // 9600bps
        else begin
            case(baud_set)          // 查找表
                0:bps_dr <= 16'd5207;             // 9600bps
                1:bps_dr <= 16'd2603;           // 19200bps
                2:bps_dr <= 16'd1301;           // 38400bps
                3:bps_dr <= 16'd0867;             // 57600bps
                4:bps_dr <= 16'd0433;             // 115200bps
                default:bps_dr <= 16'd5207;     // 9600bps
            endcase
        end

    end

    // 【Div_Cnt】 计数功能
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            div_cnt <= 16'd0;
        else if(en_cnt)    begin
            if(div_cnt == bps_dr)
                div_cnt <= 16'd0;
            else
                div_cnt <= div_cnt + 1'b1;
        end else
            div_cnt <= 16'd0;

    end

    // 【Div_Cnt】 bps_clk 时钟产生
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            bps_clk <= 1'b0;
        else if(div_cnt == 16'd1)         // 当计数器刚开始计数时就产生一个时钟
            bps_clk <= 1'b1;                 // 这样就相当于启动发送时就立即开始发送数据
        else
            bps_clk <= 1'b0;

    end

     // 【bps_cnt】 bps 计数（即发送的数据位数计数）
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            bps_cnt <= 4'd0;
        else if(clr)
            bps_cnt <= 4'd0;
        else if(bps_clk)
            bps_cnt <= bps_cnt + 1'b1;
        else
            bps_cnt <= bps_cnt;

    end

    // 【MUX10】、【r_R232_Tx】 尽量避免组合逻辑直接输出，输出是有毛刺的可能会出现不太稳定的情况
    // 发送数据模块
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            rs232_tx <= STOP_BIN;             // 起始位为低电平，所以空闲时为高电平即停止位
        else begin

            case(bps_cnt)
                0:rs232_tx <= STOP_BIN;         // 空闲时 bps_cnt 会一直为 0
                1:rs232_tx <= START_BIT;         // 起始位
                2:rs232_tx <= r_data_byte_buff[0];
                3:rs232_tx <= r_data_byte_buff[1];
                4:rs232_tx <= r_data_byte_buff[2];
                5:rs232_tx <= r_data_byte_buff[3];
                6:rs232_tx <= r_data_byte_buff[4];
                7:rs232_tx <= r_data_byte_buff[5];
                8:rs232_tx <= r_data_byte_buff[6];
```

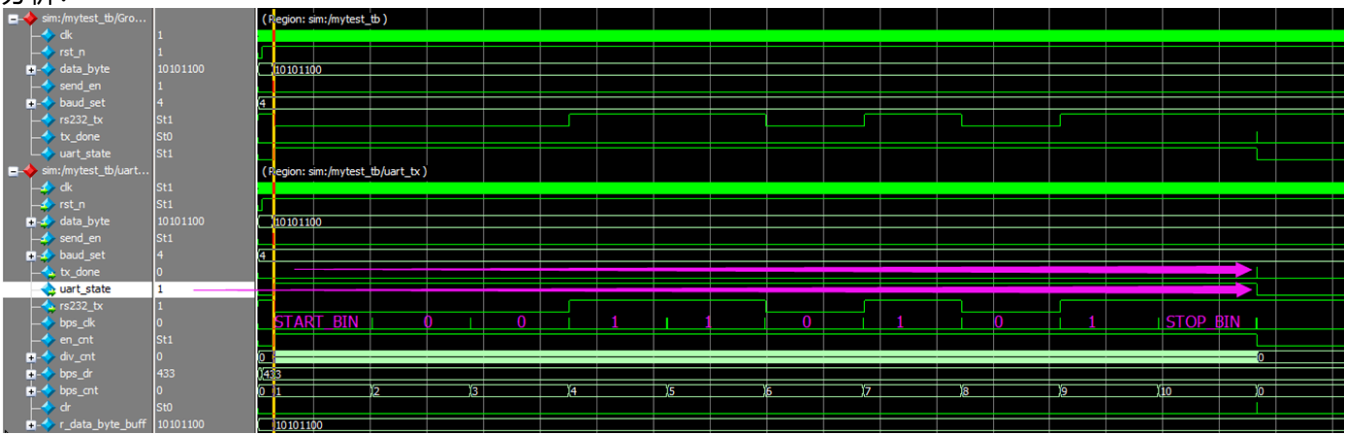
```
                9:rs232_tx <= r_data_byte_buff[7];
                10:rs232_tx <= STOP_BIN;          // 停止位
                default:rs232_tx <= STOP_BIN;
            endcase

        end

    end

    // 检测一帧数据是否发送完成
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            tx_done <= 1'b0;
        else if(bps_cnt == 4'd11)
            tx_done <= 1'b1;
        else
            tx_done <= 1'b0;
    end

    assign clr = tx_done;              // 当完成一帧数据发送之后清除 bps 计数器
```

endmodule

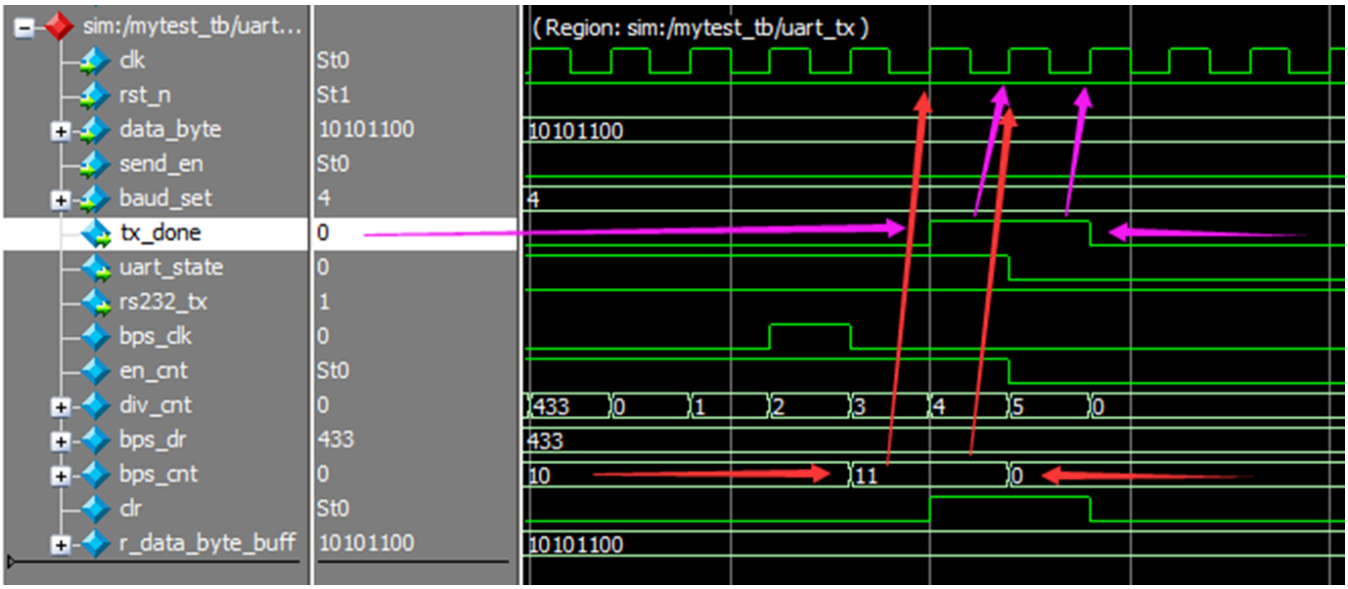
完整的时序图



分析：



问题：tx_done、bps_cnt 会分别维持两个时钟周期的 1 和 11



原因：

因为当 bps_cnt 变为 11 的时候，需要等第2个时钟周期才会被采样到。当采样到之后 tx_done = 1，因为 assign clr 也立即变为 1 ，而 clr 为 1 的时候也需要等第3个时钟周期才能被 bps_cnt 采样到变为 0，而 bps_cnt 为 0 时，需要等到第4个时钟周期才能被 tx_done 采样，才会变为 0 。

代码2：（代码与视频所修改的方式不太一样，修改了几句代码是为了尽量符合上面的框图设计）

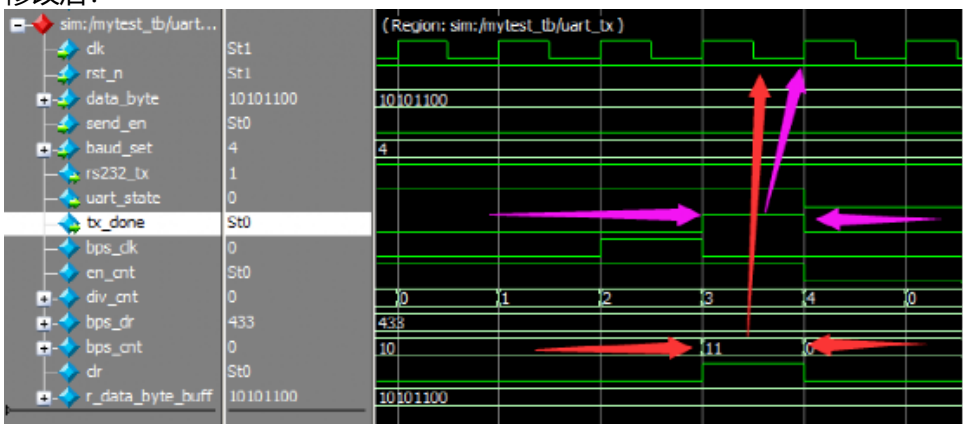
```
module mytest(clk, rst_n, data_byte, send_en, baud_set, rs232_tx, tx_done, uart_state);

    input clk;                          // 系统时钟
    input rst_n;                        // 复位
    input[7:0] data_byte;               // 要发送的数据
    input send_en;                      // 启动发送
    input[2:0] baud_set;                // 波特率选择

    output reg rs232_tx;
    output wire tx_done;                // 发送完毕通知    1:发送完毕 0:正在发送
    output reg uart_state;              // 发送状态 1:正在发送数据 0:空闲状态

    reg bps_clk;                        // 波特率时钟
    wire en_cnt;                        // 计数使能 1:使能 0:失能
    reg[15:0] div_cnt;                  // 分频计数器
    reg[15:0] bps_dr;                   // 分频计数最大值
    reg[3:0]    bps_cnt;                // 波特率时钟计数器
    wire clr;                           // 清零信号
    reg[7:0] r_data_byte_buff;          // 缓冲区，用于存储需要发送的数据，避免在发送过程中数据突然改变


    localparam START_BIT    = 1'b0;
    localparam STOP_BIN     = 1'b1;


    // 串口工作状态
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            uart_state <= 1'b0;
        else if(send_en)
            uart_state <= 1'b1;
        else if(tx_done)                // 发送完毕
            uart_state <= 1'b0;
        else
            uart_state <= uart_state;

    end

    assign en_cnt = uart_state;

    // 用于启动发送时锁存即将要发送的数据
```

```
// 这样就可以避免在发送的过程中数据突然改变导致发送的数据不正确。
always@(posedge clk, negedge rst_n) begin

    if(!rst_n)
        r_data_byte_buff <= 8'd0;
    else if(send_en)
        r_data_byte_buff <= data_byte;    // 启动发送则锁存最新的数据
    else
        r_data_byte_buff <= r_data_byte_buff;

end

// 【DR_LUT】 通过查表的方式将波特率转换为对应的分频计数最大值
always@(posedge clk, negedge rst_n) begin

    if(!rst_n)
        bps_dr <= 16'd5207;                // 9600bps
    else begin
        case(baud_set)          // 查找表
            0:bps_dr <= 16'd5207;            // 9600bps
            1:bps_dr <= 16'd2603;          // 19200bps
            2:bps_dr <= 16'd1301;          // 38400bps
            3:bps_dr <= 16'd0867;            // 57600bps
            4:bps_dr <= 16'd0433;            // 115200bps
            default:bps_dr <= 16'd5207;    // 9600bps
        endcase
    end

end

// 【Div_Cnt】 计数功能
always@(posedge clk, negedge rst_n) begin

    if(!rst_n)
        div_cnt <= 16'd0;
    else if(en_cnt)    begin
        if(div_cnt == bps_dr)
            div_cnt <= 16'd0;
        else
            div_cnt <= div_cnt + 1'b1;
    end else
        div_cnt <= 16'd0;

end

// 【Div_Cnt】 bps_clk 时钟产生
always@(posedge clk, negedge rst_n) begin

    if(!rst_n)
        bps_clk <= 1'b0;
    else if(div_cnt == 16'd1)         // 当计数器刚开始计数时就产生一个时钟
        bps_clk <= 1'b1;                // 这样就相当于启动发送时就立即开始发送数据
    else
        bps_clk <= 1'b0;

end

 // 【bps_cnt】 bps 计数（即发送的数据位数计数）
always@(posedge clk, negedge rst_n) begin

    if(!rst_n)
        bps_cnt <= 4'd0;
    else if(clr)
        bps_cnt <= 4'd0;
    else if(bps_clk)
        bps_cnt <= bps_cnt + 1'b1;
    else
        bps_cnt <= bps_cnt;

end

// 【MUX10】、【r_R232_Tx】 尽量避免组合逻辑直接输出，输出是有毛刺的可能会出现不太稳定的情况
// 发送数据模块
```

```
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            rs232_tx <= STOP_BIN;            // 起始位为低电平，所以空闲时为高电平即停止位
        else begin

            case(bps_cnt)
                0:rs232_tx <= STOP_BIN;        // 空闲时 bps_cnt 会一直为 0
                1:rs232_tx <= START_BIT;        // 起始位
                2:rs232_tx <= r_data_byte_buff[0];
                3:rs232_tx <= r_data_byte_buff[1];
                4:rs232_tx <= r_data_byte_buff[2];
                5:rs232_tx <= r_data_byte_buff[3];
                6:rs232_tx <= r_data_byte_buff[4];
                7:rs232_tx <= r_data_byte_buff[5];
                8:rs232_tx <= r_data_byte_buff[6];
                9:rs232_tx <= r_data_byte_buff[7];
                10:rs232_tx <= STOP_BIN;        // 停止位
               default:rs232_tx <= STOP_BIN;
            endcase

        end

    end
    /*
    // 检测一帧数据是否发送完成 // 采用此种方式会导致 tx_done、bps_cnt 会分别维持两个时钟周期的 1
和 11
    // 因为当 bps_cnt 变为 11 的时候，需要等第2个时钟周期才会被采样到。当采样到之后 tx_done = 1,
而 clr 也立即变为 1 ,
    // 而 clr 为 1 的时候也需要等第3个时钟周期才能被 bps_cnt 采样到变为 0
    // 而 bps_cnt 为 0 时，需要等到第4个时钟周期才能被 tx_done 采样，才会变为 0
    always@(posedge clk, negedge rst_n) begin

        if(!rst_n)
            tx_done <= 1'b0;
        else if(bps_cnt == 4'd11)
            tx_done <= 1'b1;
        else
            tx_done <= 1'b0;

    end
    */
    // 【r_Tx_Done】 为了避免 tx_done 这里采用直接赋值的方式来避免出现延迟一个时钟的现象
    assign tx_done = bps_cnt == 4'd11 ? 1'b1 : 1'b0;

    assign clr = tx_done;            // 当完成一帧数据发送之后清除 bps 计数器
```

endmodule

修改后：



板级验证：

时间太晚，就不做板级实验了。