电子科技大学

University of Electronic Science and Technology of China

本科生实验报告

实验课程：复杂数字集成电路设计（挑战性课程）

实验名称：基于 FPGA 的 UART 模块设计

实验地点：无

学生姓名：周岳恒

学　　号：2021340105016

指导教师：廖永波

实验时间：2023 年 11 月 1 日

一、 实验目的

了解 UART 的原理，尝试用 verilog 代码对 UART 模块实现仿真和综合，并配合异步 FIFO 使用下载到 FPGA 开发板上验证功能的正确。

二、 实验任务

1. 实现 UART 模块的发送和接收子模块的 verilog 代码，并可配置奇偶校验、停止位长度等功能，可配置不同波特率实现不同硬件条件下的 uart 收发功能
2. 借用之前的异步 FIFO 和此 UART 模块连接，搭建 UART 的实际工作环境并验证模块的正确性。
3. 将上述硬件代码在 FPGA 上实现，连接计算机串口作为上位机，实现不定长信息的收发功能，此操作可拓展为上位机对嵌入式单片机的指令传递。

三、 实验原理

UART(Universal Asynchronous Receiver/Transmitter)，是一种异步、串行、全双工、双设备的通信协议。

对于不同的通信协议，在时钟、数据线个数、工作模式等方面存在差异，也因此各个通信协议在不同领域有不同的优势和缺点。

时钟分为同步和异步：同步指通信的设备使用同一根时钟线实现时钟同步，信息的传递需要此时钟更新节拍，实现协议的各个功能。异步指的是通信设备之间不需要连接单独的时钟线，在通信之前双方约定好使用固定的通信速率实现收发信息比特位的区分。在这里，信息的组织以帧格式为单位，在一个信息帧中有多个 bit 信息，其中有一部分 bit 信息用来辅助确定通信的开始结束、奇偶校验等功能。而且由于是异步通信，双方时钟不同，长时间连续通信会造成时钟偏斜误差累积，所以一个信息帧不能太长。这导致异步通信需要大量辅助位标记确保时钟协调，所以传输效率较低，适合少量数据传输，一般用在字符串的传输，如上位机给下位机发送指令。

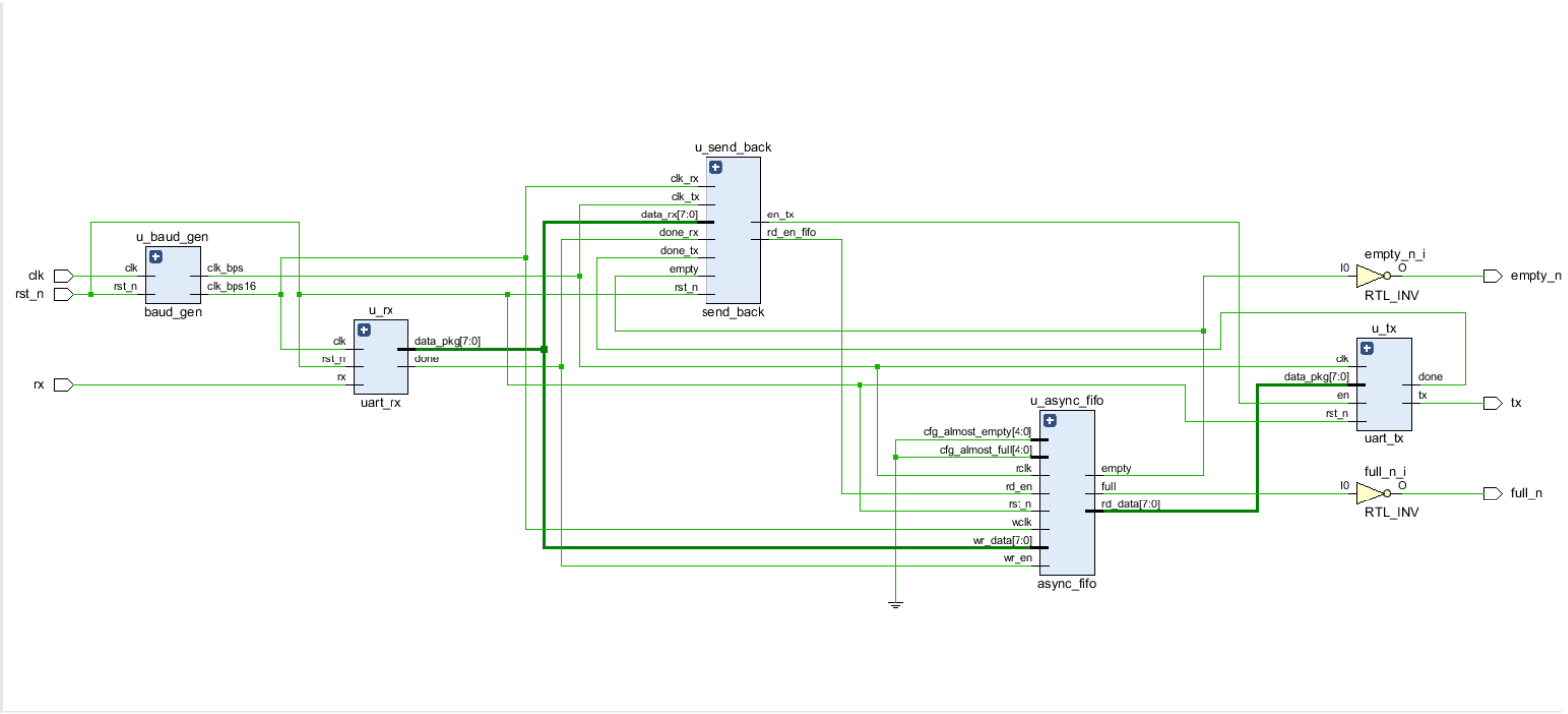
串行通信需要一条数据线时分复用传输多个数据 bit，并行通信根据在同一时间传输 bit 的个数确定数据线个数，利用空间换取时间。串行通信用时间换取空间，传输速率较低，但是节省了数据线的个数，适用于简单应用场景，对资源的要求比较低。

工作模式分为单工、半双工、全双工。单工表示通信只允许由设备 A 传输给另一个设备 B，方向固定不可改变。全双工表示可以在同一时间内 A 传输到 B 且 B 传输到 A，两者互不影响。半双工表示在同一时间内只能由 A 传输到 B 或 B 传输到 A，不可同时进行。相较于单工模式，传输的方向可以改变。UART 一般包含两根数据线 TX 和 RX，两条线的传输相互独立，参与传输的设备可以在任何时间内发送或接收。

双设备指通信线为一对一的结构，而不是如 AMBA、I^2C 一样使用总线连接多个设备使用仲裁等结构。所以 UART 不适用于多个设备的连接，否则会消耗较多的连线资源，每两个设备之间需要单独的连线。

四、 实验过程

模块总体框架：



UART 模块包括 TX、RX、波特率生成器；此外还有异步 FIFO 存储数据，send_back 用来决定回传数据的条件并生成控制信号。

UART 的 RX 模块：

verilog 代码：

```
module uart_rx #(
```

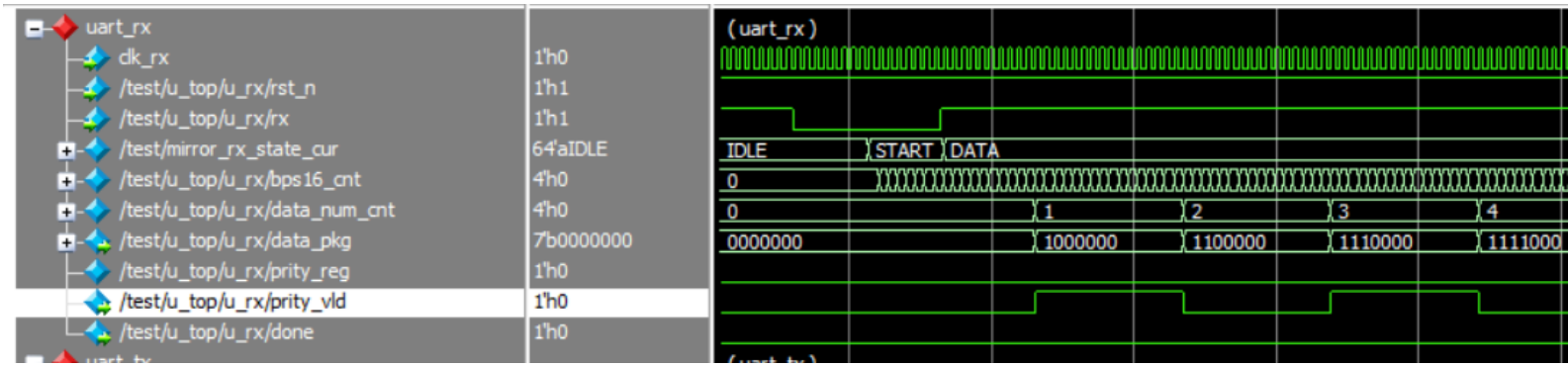
```
    parameter DATA_WIDTH = 8,
    parameter PARITY_ON = 1,
    parameter PARITY_ODD = 1,
    parameter STOP_BIT = 1  // choose 1 / 2 / 3=1.5
)(
    input clk,  // (9600 * 16)Hz
    input rst_n,
    input rx,
    output [DATA_WIDTH-1:0] data_pkg,
    output prity_vld,
    output done
);
    reg [15:0] rx_reg;
    reg [3:0] bps16_cnt;
    reg [3:0] data_num_cnt;
    reg stop_fin;

    reg [2:0] state_cur,state_nxt;
    parameter IDLE_S = 3'b000,
            START_S  = 3'b001,
            DATA_S   = 3'b011,
            PRITY_S  = 3'b010,
            STOP_S   = 3'b110;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            state_cur <= IDLE_S;
        else
            state_cur <= state_nxt;
    end
    always @(*) begin
        case (state_cur)
            IDLE_S:  state_nxt = ~(|rx_reg[7:0]) ? START_S : IDLE_S;
            START_S: state_nxt = (bps16_cnt == 7) ? DATA_S : START_S;
            DATA_S:  state_nxt = (data_num_cnt == DATA_WIDTH &&
bps16_cnt == 15) ?
                (PARITY_ON ? PRITY_S : STOP_S) : DATA_S;
            PRITY_S: state_nxt = (bps16_cnt == 15) ? STOP_S : PRITY_S;
            STOP_S:  state_nxt = (stop_fin) ? IDLE_S : STOP_S;
            default: state_nxt = IDLE_S;
        endcase
    end

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

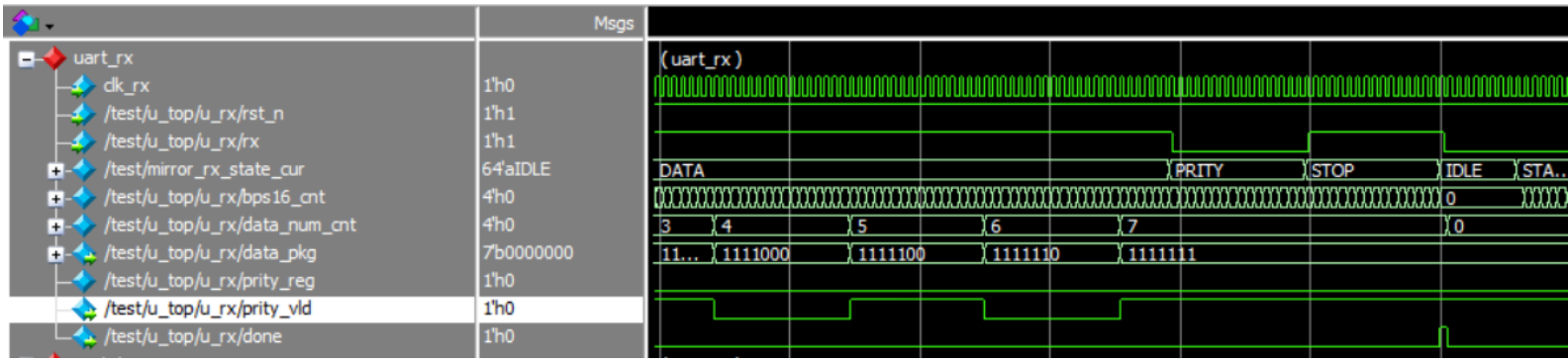
```
        if(~rst_n)
            rx_reg <= 16'hFFFF;
        else
            rx_reg <= {rx_reg[14:0], rx};   // left shift
    end

    // bps16_cnt
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            bps16_cnt <= 0;
        else begin
            case (state_cur)
                IDLE_S:  bps16_cnt <= 0;
                START_S: bps16_cnt <= (bps16_cnt == 7) ? 0 : bps16_cnt +
1;
                default: bps16_cnt <= bps16_cnt + 1;
            endcase
        end
    end

    // data_num_cnt
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            data_num_cnt <= 0;
        else if(state_cur == DATA_S && bps16_cnt == 9)
            data_num_cnt <= data_num_cnt + 1;
        else if(state_cur == IDLE_S)
            data_num_cnt <= 0;
    end

    // data_pkg_reg
    reg [DATA_WIDTH-1:0] data_pkg_reg;
    wire rx_ave;
    assign rx_ave = (rx_reg[0] & rx_reg[1]) | (rx_reg[0] & rx_reg[2]) |
(rx_reg[1] & rx_reg[2]);
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            data_pkg_reg <= 0;
        else if(state_cur == DATA_S && bps16_cnt == 9)   // rx_reg[2:0]
--> the 7/8/9 data of 1~16 samples
            data_pkg_reg <= {rx_ave, data_pkg_reg[DATA_WIDTH-1:1]}; //
right shift (LSB first)
    end
    assign data_pkg = data_pkg_reg;
```

```
    // prity_reg
    reg prity_reg;
    wire odd_check;
    assign odd_check = ^{data_pkg, prity_reg};
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            prity_reg <= 0;
        else if(state_cur == PRITY_S && bps16_cnt == 9)
            prity_reg <= rx_ave;
    end
    assign prity_vld = PARITY_ON ? (PARITY_ODD ? odd_check :
~odd_check) : 1'b0;

    // stop_fin
    reg [1:0] stop_bit_cnt;
    always @(posedge clk,negedge rst_n) begin
        if(~rst_n)
            stop_bit_cnt <= 0;
        else if(state_cur == STOP_S && bps16_cnt == 15)
            stop_bit_cnt <= stop_bit_cnt + 1;
        else if(state_cur == IDLE_S)
            stop_bit_cnt <= 0;
    end
    always @(*) begin
        case (STOP_BIT)
            3:   stop_fin = (stop_bit_cnt == 1 && bps16_cnt == 7);
            2:   stop_fin = (stop_bit_cnt == 1 && bps16_cnt == 15);
            default: stop_fin = (stop_bit_cnt == 0 && bps16_cnt == 15);
        endcase
    end

    // done
    reg done_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            done_reg <= 0;
        else if(state_cur == STOP_S && stop_fin)
            done_reg <= 1;
        else if(state_cur == IDLE_S)
            done_reg <= 0;
    end
    assign done = done_reg;
```

```
endmodule
```

波形解释：





本模块使用有限状态机实现。使用了 5 个状态：idle 空闲、start 起始位、data 数据位、prity 校验位、stop 停止位。在每个状态都会启动波特率计数器计数，控制状态机在每个状态的停留时间。

示例使用奇校验，波特率 9600bps，数据位 7 位（一个 ASCII 码要 7bit 数据），停止位 1 位。

波特率计数器在大部分状态位计数 16 次，因为 rx 的时钟就是波特率的 16 倍频，所以计数 16 次相当于 16 分频，得到的状态时钟和波特率一致。使用 16 倍频是为了满足接收的超采样要求：使用多次采样得到的值可以有效减少信号线上干扰造成数据位的错误，尽可能保证数据传输的准确性。可以在每一位的中间做 3 次采样，选择重复次数多的位记录值，使用真值表列解卡诺图得到逻辑表达式。

在这里，我选择在数据起始位记录前 8 个时钟全部采样到 0 则进入接收数据

模式，进入 start 状态并计数 8 次进入 data 状态。从此开始采样，每当波特率计数器到 7、8、9 时选择有效值存入存数据的移位寄存器中作为 rx 输入。并且采样到一个 bit 数据后对 data_num 数据计数器递增计数，确保帧格式和通讯设备相同。

记录完数据后进入校验位检测，校验位采样和数据位采样相同，将校验位采样的值拼接记录数据的移位寄存器进行异或运算，如果结果为 1 则奇校验正确，如果位 0 则偶校验正确，反之不正确。最后使用计数器计数停止位的个数，结束后进入 IDLE 等待下一个周期的输入。除此之外没有设计帧格式错误的检测，只有奇偶校验的检测，所以不会检测停止位是否发生异常，不会采样停止位的值。

done 信号在 stop 状态结束后产生一个周期的脉冲，说明本次数据已经接收好，需要被取出。

重新进入 IDLE 状态后所有寄存器都清零，确保及时准备好下一次数据接收。

UART 的 TX 模块：

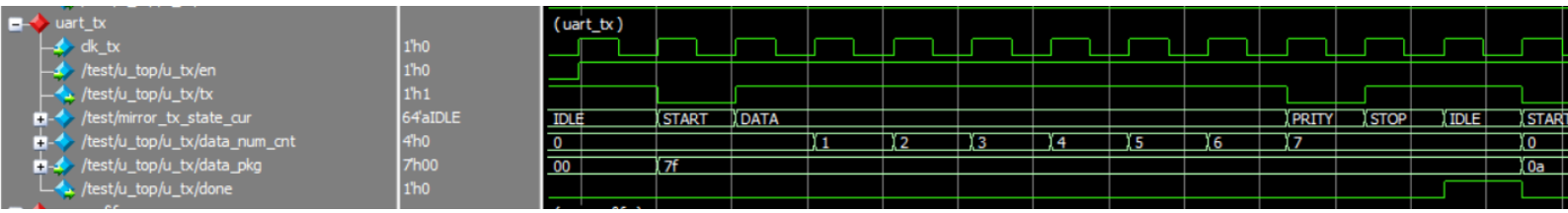
verilog 代码：

```
module uart_tx #(
    parameter DATA_WIDTH = 8,
    parameter PARITY_ON = 1,
    parameter PARITY_ODD = 1,
    parameter STOP_BIT = 1  // choose 1 / 2
)(
    input clk,  // 9600Hz
    input rst_n,
    input [DATA_WIDTH-1:0] data_pkg,
    input en,   // only hold for one clock
    output tx,
    output done
);
    reg [3:0] data_num_cnt;
    wire stop_fin;

    reg [2:0] state_cur,state_nxt;
    parameter IDLE_S = 3'b000,
            START_S  = 3'b001,
            DATA_S   = 3'b011,
            PRITY_S  = 3'b010,
            STOP_S   = 3'b110;
    always @(posedge clk, negedge rst_n) begin
```

```
        if(~rst_n)
            state_cur <= IDLE_S;
        else
            state_cur <= state_nxt;
    end
    always @(*) begin
        case (state_cur)
            IDLE_S:  state_nxt = (en) ? START_S : IDLE_S;
            START_S: state_nxt = DATA_S;
            DATA_S:  state_nxt = (data_num_cnt == DATA_WIDTH - 1) ?
                (PARITY_ON ? PRITY_S : STOP_S) : DATA_S;
            PRITY_S: state_nxt = STOP_S;
            STOP_S:  state_nxt = (stop_fin) ? IDLE_S : STOP_S;
            default: state_nxt = IDLE_S;
        endcase
    end

    // data_num_cnt
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            data_num_cnt <= 0;
        else if(state_cur == DATA_S)
            data_num_cnt <= data_num_cnt + 1;
        else if(state_cur == IDLE_S)
            data_num_cnt <= 0;
    end

    // tx_reg
    reg tx_reg;
    always @(*) begin
        case (state_cur)
            START_S: tx_reg <= 1'b0;
            DATA_S:  tx_reg <= data_pkg[data_num_cnt];
            PRITY_S: tx_reg <= PARITY_ODD ? (~(^data_pkg)) :
(^data_pkg);   // already ensure PRITY_ON = 1
            default: tx_reg <= 1'b1;
        endcase
    end
    assign tx = tx_reg;

    // stop_fin
    reg stop_bit_cnt;
    always @(posedge clk,negedge rst_n) begin
        if(~rst_n)
```

```
            stop_bit_cnt <= 0;
        else if(state_cur == STOP_S)
            stop_bit_cnt <= stop_bit_cnt + 1;
        else if(state_cur == IDLE_S)
            stop_bit_cnt <= 0;
    end
    assign stop_fin = (stop_bit_cnt == STOP_BIT - 1);

    // done
    reg done_reg;
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n)
            done_reg <= 0;
        else if(state_cur == STOP_S && stop_fin)
            done_reg <= 1;
        else if(state_cur == IDLE_S)
            done_reg <= 0;
    end
    assign done = done_reg;

endmodule
```

波形解释：



TX 发送端的时钟按照波特率更新，不需要倍频，除非停止位为 1.5 位则需要 2 倍频。当使能信号有效后状态寄存器从 IDLE 到 START 输出起始位，随后进入数据位依次输出数据寄存器数据，同时 data_num 计数器递增计数到协议规定的数据宽度。数据相位结束后进如奇偶校验位，奇校验输出结果为数据寄存器异或的非，偶校验结果为数据寄存器异或。随后进入停止位输出高电平。停止位结束后重新回到 IDLE 状态等待下次输入。

done 信号在 stop 状态结束后产生一个周期的脉冲，表明数据已经写入，如果需要下次发送就将使能信号置 1。

波特率产生模块：

verilog 代码：

```
module baud_gen #(
    parameter CLK_FREQ = 50_000_000,
    parameter BAUD_RATE = 9600
)(
    input clk,
    input rst_n,
    output clk_bps,
    output clk_bps16
);
    localparam BPS_DIV = CLK_FREQ / (BAUD_RATE * 2);
    localparam BPS16_DIV = CLK_FREQ / (BAUD_RATE * 16 * 2);

    reg [31:0] bps_cnt;
    reg [31:0] bps16_cnt;
    reg clk_bps_reg;
    reg clk_bps16_reg;

    always @(posedge clk, negedge rst_n) begin
        if(~rst_n) begin
            bps_cnt <= 0;
            clk_bps_reg <= 1'b0;
        end
        else begin
            if(bps_cnt == BPS_DIV - 1) begin
                bps_cnt <= 0;
                clk_bps_reg <= ~clk_bps_reg;
            end
            else
                bps_cnt <= bps_cnt + 1;
        end
    end
    always @(posedge clk, negedge rst_n) begin
        if(~rst_n) begin
            bps16_cnt <= 0;
            clk_bps16_reg <= 1'b0;
        end
        else begin
            if(bps16_cnt == BPS16_DIV - 1) begin
                bps16_cnt <= 0;
                clk_bps16_reg <= ~clk_bps16_reg;
```

```
            end
            else
                bps16_cnt <= bps16_cnt + 1;
        end
    end

    assign clk_bps = clk_bps_reg;
    assign clk_bps16 = clk_bps16_reg;

endmodule
```

两个分频器。根据总时钟和波特率分出波特率时钟给 TX，分出波特率的 16 倍时钟给 RX 超采样。

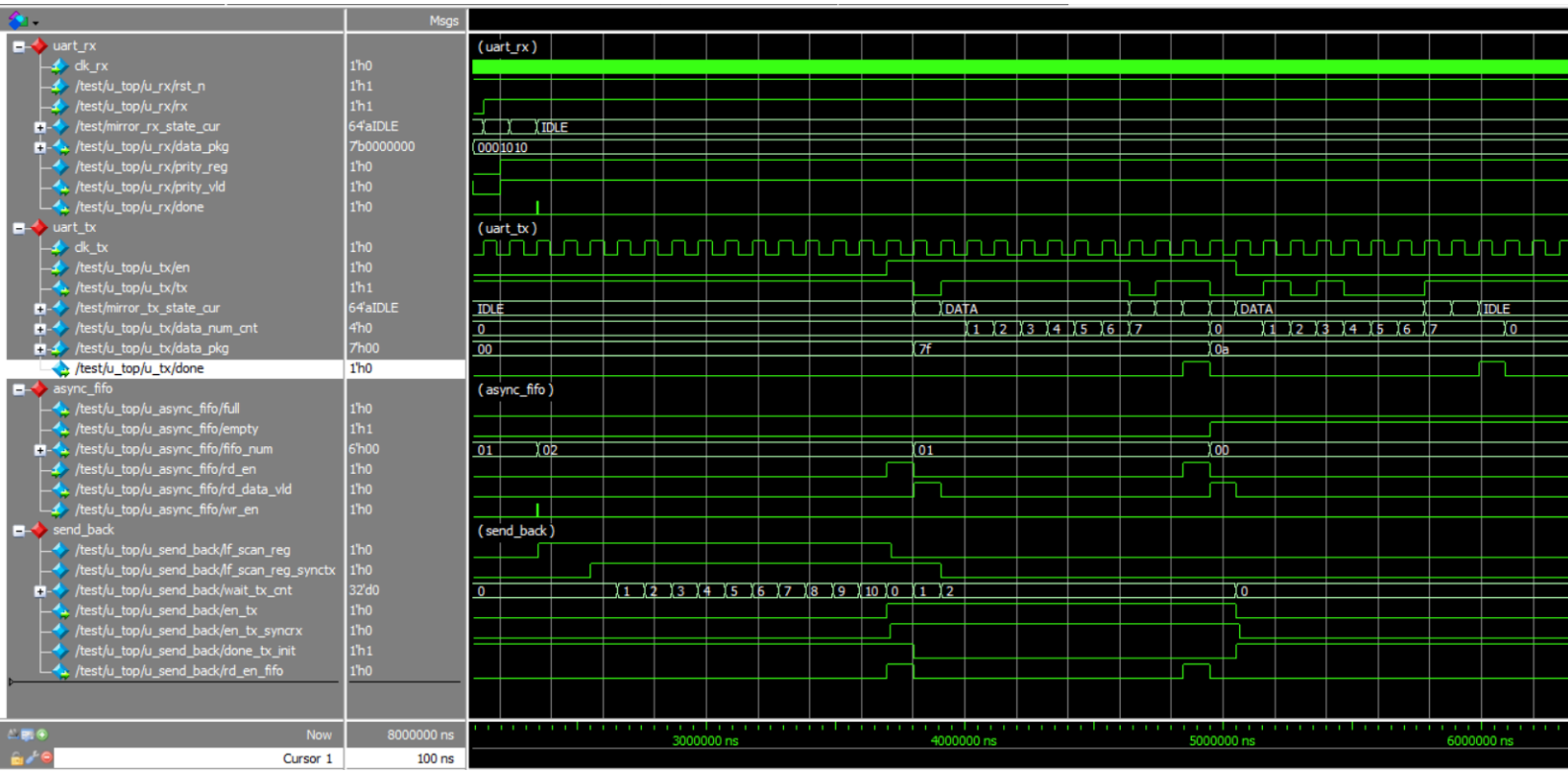
数据回传模块：

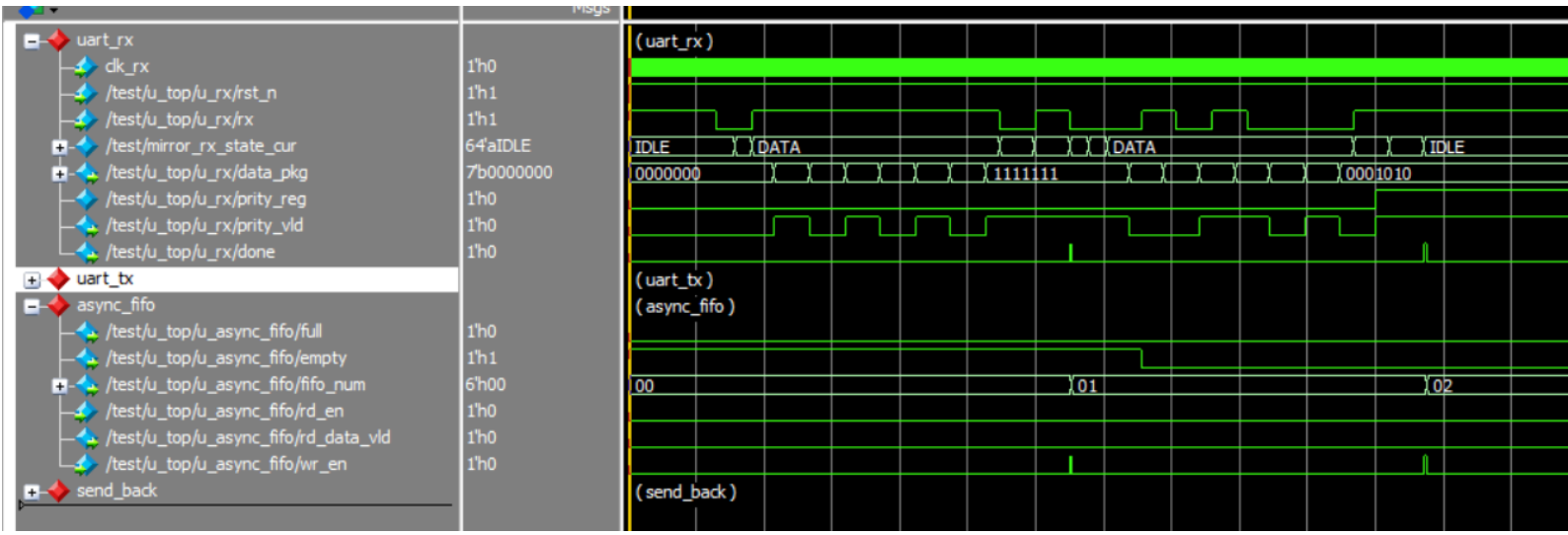
verilog 代码：

```
module send_back #(
    parameter DATA_WIDTH = 8
)(
    input clk_tx,
    input clk_rx,
    input rst_n,
    input done_tx,
    input done_rx,
    input [DATA_WIDTH-1:0] data_rx,
    input empty,
    output en_tx,
    output rd_en_fifo
);
    reg en_tx_reg;
    assign en_tx = en_tx_reg;

    reg lf_scan_reg;    // clk_rx
    wire en_tx_syncrx;
    always @(posedge clk_rx, negedge rst_n) begin
        if(~rst_n)
            lf_scan_reg <= 0;
        else if(done_rx && data_rx == 8'b00001010)  // get LF(line feed)
for ascii=0xA
            lf_scan_reg <= 1;
        else if(en_tx_syncrx)
            lf_scan_reg <= 0;
```

```
    end
    sig_sync u_sync_en_tx(
        .clk(clk_rx),
        .rst_n(rst_n),
        .sig_async(en_tx_reg),
        .sig_sync(en_tx_syncrx)
    );

    reg [31:0] wait_tx_cnt;
    wire lf_scan_reg_synctx;
    sig_sync u_sync_lf_scan_reg(
        .clk(clk_tx),
        .rst_n(rst_n),
        .sig_async(lf_scan_reg),
        .sig_sync(lf_scan_reg_synctx)
    );
    always @(posedge clk_tx, negedge rst_n) begin
        if(~rst_n) begin
            wait_tx_cnt <= 0;
            en_tx_reg <= 0;
        end
        else if(empty) begin
            wait_tx_cnt <= 0;
            en_tx_reg <= 0;
        end
        else if(lf_scan_reg_synctx) begin
            wait_tx_cnt <= (wait_tx_cnt == 10) ? 0 : wait_tx_cnt + 1;
            if(wait_tx_cnt == 10)
                en_tx_reg <= 1;
        end
    end

    reg done_tx_init;
    always @(posedge clk_tx, negedge rst_n) begin
        if(~rst_n)
            done_tx_init <= 1;
        else if(empty)
            done_tx_init <= 1;
        else if(en_tx_reg)
            done_tx_init <= 0;
    end
    assign rd_en_fifo = en_tx_reg & (done_tx_init | done_tx);

endmodule
```

波形解释：





系统启动后使用 UART 的 RX 端接收数据并传输到 FIFO 中记录数据。RX 和 FIFO 的 write 使用一个时钟，TX 和 FIFO 的 read 使用一个时钟。使用 RX 端

的 done 信号作为 FIFO 的写使能信号，把数据写入异步 FIFO。可以看到，empty 信号延迟置 0，因为它在相对较慢的读时钟。

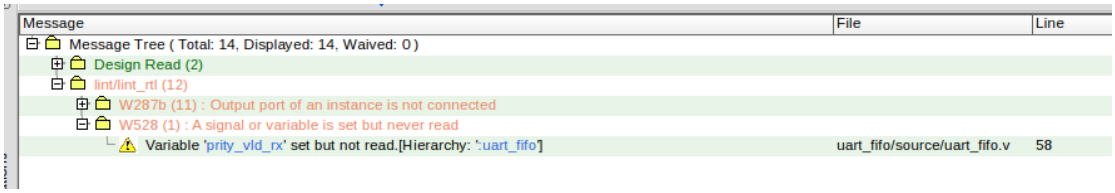
数据回传的条件是 UART 的 RX 端接收到 LF 换行符且接收完成。此时激活 lf_scan_reg 信号，开启计数器计时到特定值后开始回传数据。这里 LF 的接收在 RX 的时钟，计数器计数以及回传数据在 TX 时钟，需要 CDC 打拍寄存器延时。计数器定时结束后控制 FIFO 的写使能信号在需要发数据时有效，依据是 TX 端 done 信号产生的脉冲。使能信号会在检测到 FIFO 的 empty 信号有效时置 0，并把回传模块的所有寄存器复位，准备好下一次传输。此时 FIFO 已空，等待下一接收数据。本次不定长数据传输结束。

当 FIFO 已经 full 时在写入数据会通过模块顶层端口输出信息，且之后输入的信息都无效，不会存入 FIFO 中。直到接收到 LF 再把之前已经存入 FIFO 的数据依次读出。

done_tx_init 是保证在刚开始 TX 端没发送数据时 done 无效时能够给一个开始信号作为引线，帮助第一次使能读出 FIFO，之后依赖 TX 的 done 信号就不需要 init 信号了。

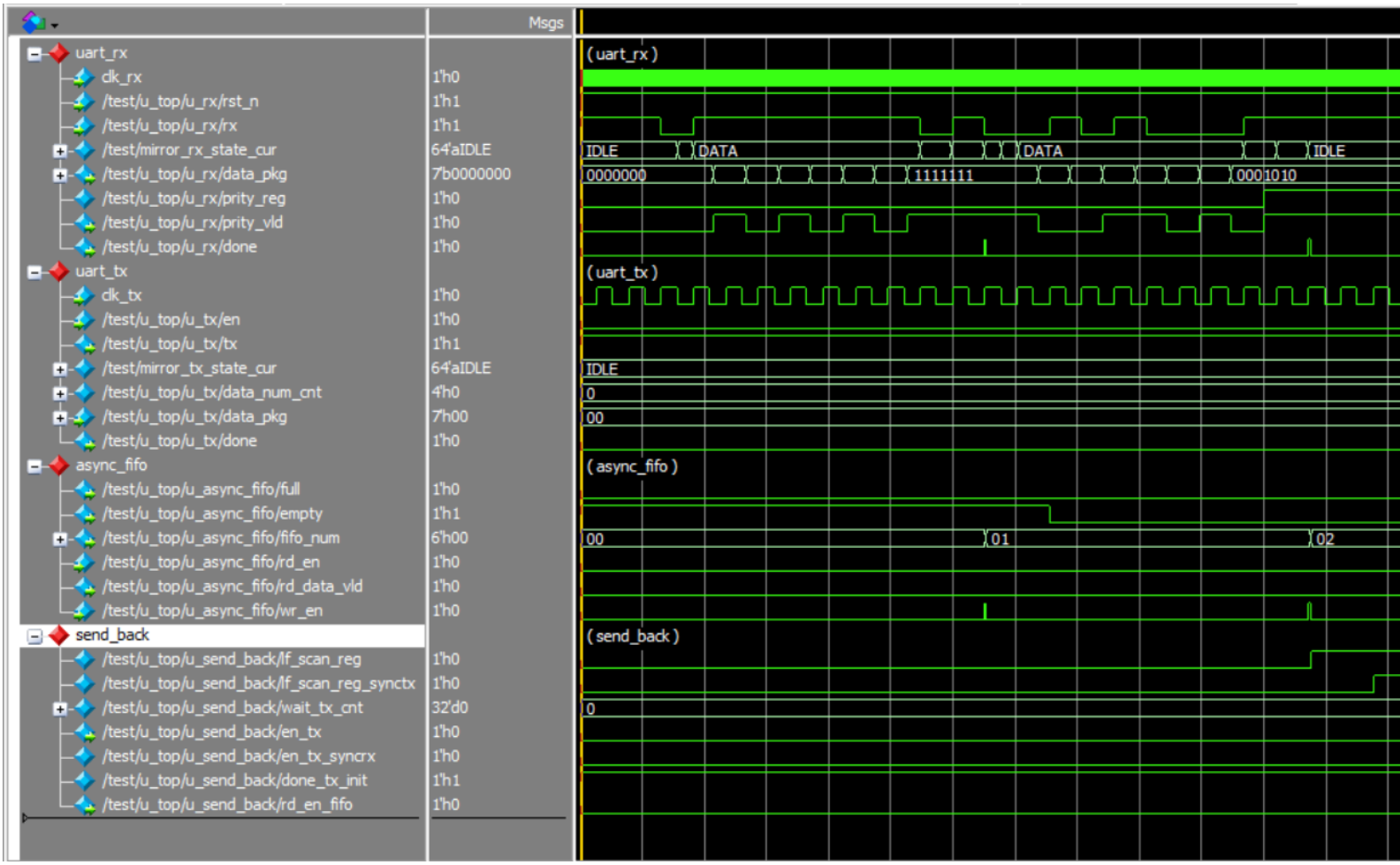
异步 FIFO 的具体信息见实验 8。

spyglass 报告：

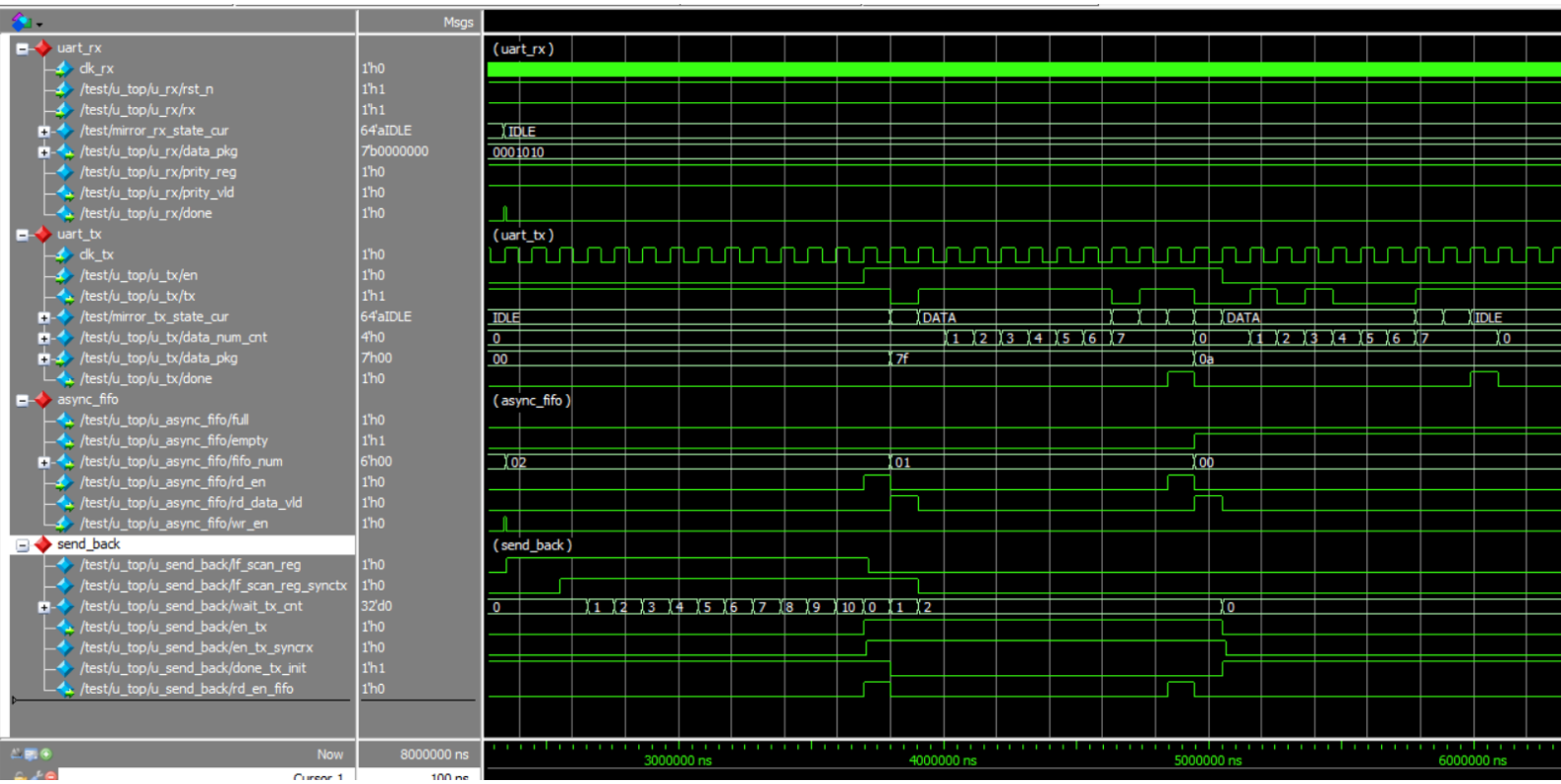


部分输出端口浮空。prity_vld_rx 没有映射到具体功能，但是在 modelsim 仿真时需要方便观察。确认没有其他问题。

五、 仿真结果



具体信号见上文。在这里外界发送 7’b1111111 和 7’b0001010（LF）两个帧的数据并倍 FIFO 接收。由于检测到了 LF 信号，send_back 回传模块将开始工作。



在检测到 LF 后计数器计数，到规定值后使能 FIFO 的读使能信号，依赖 done 信号把电平转变为脉冲从 FIFO 中读出数据（FIFO 读和 UART 的 TX 一个时钟，FIFO 的读使能信号不能一直有效，UART 发送需要好几个时钟周期）。当 FIFO 读到 empty 后回传模块寄存器复位，完成输出输出，准备下一次数据输入。

testbench 代码：

```
`timescale 1ns/1ns
module test ();
    reg clk;
    reg rst_n;
    reg rx;
    wire tx;
    wire full_n;
    wire empty_n;

    top u_top(
        .clk(clk),
        .rst_n(rst_n),
        .rx(rx),
        .tx(tx),
```

```
        .full_n(full_n),
        .empty_n(empty_n)
    );

    reg [63:0] mirror_rx_state_cur,mirror_tx_state_cur;
    always @(u_top.u_rx.state_cur) begin
        case (u_top.u_rx.state_cur)
            u_top.u_rx.IDLE_S: mirror_rx_state_cur = "IDLE";
            u_top.u_rx.START_S: mirror_rx_state_cur = "START";
            u_top.u_rx.DATA_S: mirror_rx_state_cur = "DATA";
            u_top.u_rx.PRITY_S: mirror_rx_state_cur = "PRITY";
            u_top.u_rx.STOP_S: mirror_rx_state_cur = "STOP";
        endcase
    end
    always @(u_top.u_tx.state_cur) begin
        case (u_top.u_tx.state_cur)
            u_top.u_tx.IDLE_S: mirror_tx_state_cur = "IDLE";
            u_top.u_tx.START_S: mirror_tx_state_cur = "START";
            u_top.u_tx.DATA_S: mirror_tx_state_cur = "DATA";
            u_top.u_tx.PRITY_S: mirror_tx_state_cur = "PRITY";
            u_top.u_tx.STOP_S: mirror_tx_state_cur = "STOP";
        endcase
    end

    initial begin
        clk = 1'b0;
        forever #10 clk = ~clk;
    end

    integer i;
    initial begin
        rst_n = 1'b0;
        rx = 1'b1;
        #11 rst_n = 1'b1;
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        #1 rx = 0;
        @(posedge u_top.clk_tx);
        #1 rx = 1;
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
```

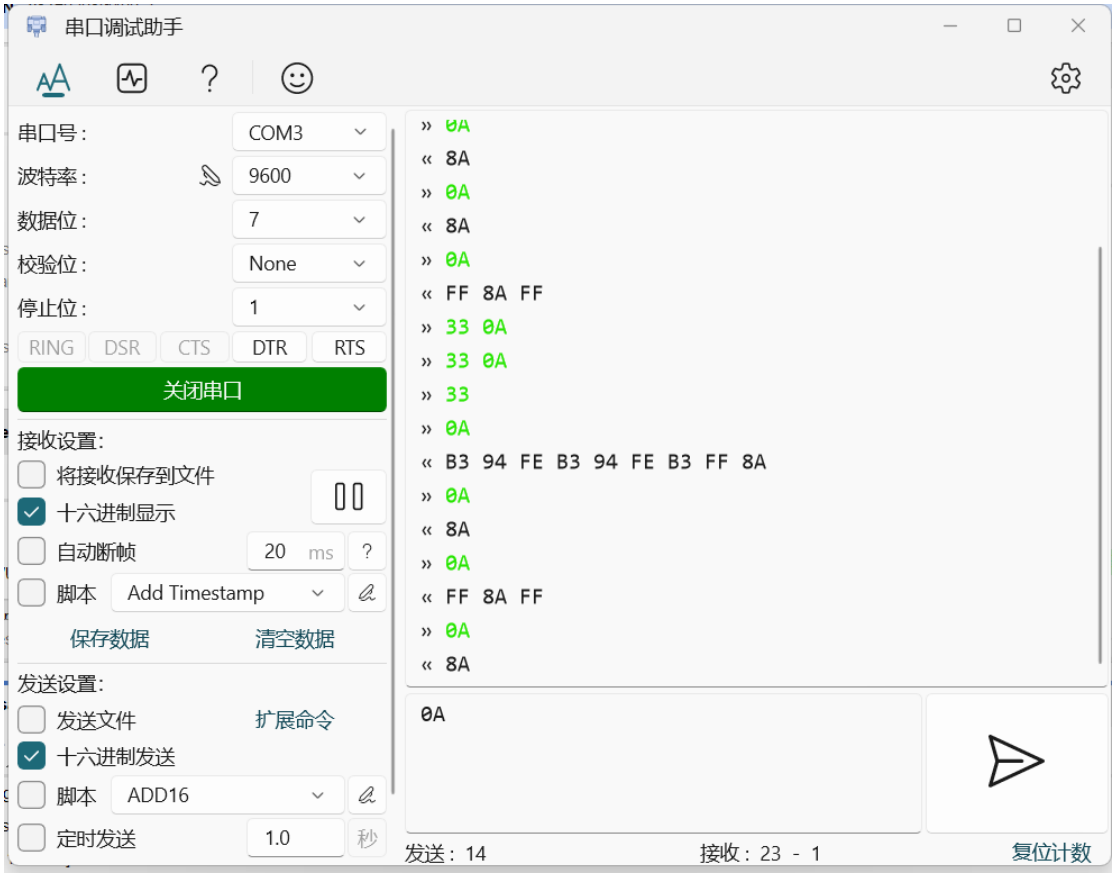
```
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        #1 rx = 0;
        @(posedge u_top.clk_tx);
        #1 rx = 1;
        @(posedge u_top.clk_tx);
        #1 rx = 0;
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        #1 rx = 1;
        @(posedge u_top.clk_tx);
        #1 rx = 0;
        @(posedge u_top.clk_tx);
        #1 rx = 1;
        @(posedge u_top.clk_tx);
        #1 rx = 0;
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        @(posedge u_top.clk_tx);
        #1 rx = 1;
    end

endmodule
```

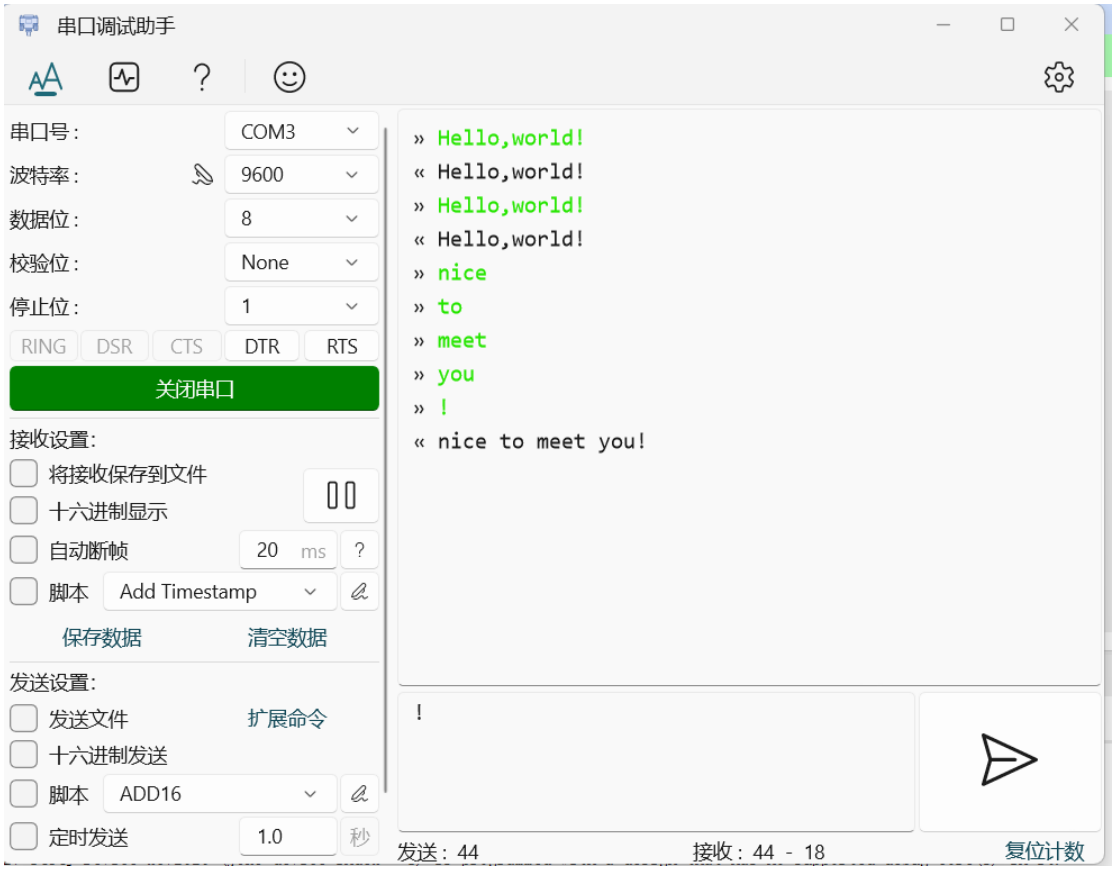
mirror_state_cur 方便在仿真波形中看到字符串表示的状态位。

FPGA 实验：


串口调试助手
波特率：
9600
数据位：
8
校验位：
None
停止位：
1
RING
DSR
CTS
DTR
RTS
关闭串口
接收设置：
将接收保存到文件
十六进制显示
自动断帧
20
ms
脚本
Add Timestamp
保存数据
清空数据
发送设置：
发送文件
扩展命令
十六进制发送
脚本
ADD16
定时发送
1.0
秒
显示发送字符串
« 0A
» 0A
« 0A
» 22
» 23
» 0A
« 22 23 0A
» 54 54 28 59 E6 0A
« 54 54 28 59 E6 0A
» Hello,world!
» Hello,world!
« Hello,world!Hello,world!
» Hello,world!
« Hello,world!
» Hello,world!
« Hello,world!
Hello,world!
发送：63
接收：63 - 13
复位计数


串口调试助手
波特率：
9600
数据位：
8
校验位：
None
停止位：
1
RING
DSR
CTS
DTR
RTS
关闭串口
接收设置：
将接收保存到文件
十六进制显示
自动断帧
20
ms
脚本
Add Timestamp
保存数据
清空数据
发送设置：
发送文件
扩展命令
十六进制发送
脚本
ADD16
定时发送
1.0
秒
显示发送字符串
» 0A
« 0A
» 0A
« 0A
» 0A
« 0A
» 22
» 23
» 0A
« 22 23 0A
0A
发送：6
接收：6 - 3
复位计数


串口调试助手
串口号：COM3
波特率：9600
数据位：7
校验位：None
停止位：1
RING DSR CTS DTR RTS
关闭串口
接收设置：
将接收保存到文件
十六进制显示
自动断帧 20 ms
脚本 Add Timestamp
保存数据 清空数据
发送设置：
发送文件 扩展命令
十六进制发送
脚本 ADD16
定时发送 1.0 秒
» 0A
« 8A
» 0A
« 8A
» 0A
« FF 8A FF
» 33 0A
» 33 0A
» 33
» 0A
« B3 94 FE B3 94 FE B3 FF 8A
» 0A
« 8A
» 0A
« FF 8A FF
» 0A
« 8A
0A
发送：14
接收：23 - 1
复位计数


串口调试助手
串口号：COM3
波特率：9600
数据位：8
校验位：None
停止位：1
RING DSR CTS DTR RTS
关闭串口
接收设置：
将接收保存到文件
十六进制显示
自动断帧 20 ms
脚本 Add Timestamp
保存数据 清空数据
发送设置：
发送文件 扩展命令
十六进制发送
脚本 ADD16
定时发送 1.0 秒
» Hello,world!
« Hello,world!
» Hello,world!
« Hello,world!
» nice
» to
» meet
» you
» !
« nice to meet you!
!
发送：44
接收：44 - 18
复位计数

六、 实验总结

本次实验实现了 UART 的 TX、RX 模块设计，实现 UART 波特率生成器的时钟分频。为了和异步 FIFO 联合使用实现不定长数据的收发，又实现了数据回传模块。结合异步 FIFO 的设计，完成了功能。本次实验我在学习 UART 的基础上又复习和验证了异步 FIFO，对模块设计的通用性的理解更深刻，可以设计更通用的接口。