



哈爾濱工業大學 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期: 2025 秋季

课程名称: 数字逻辑设计 (实验)

实验名称: 综合实验

实验性质: 综合设计型

实验学时: 6 地点: _____

学生班级:

学生学号: _____

学生姓名: _____

评阅教师: _____

报告成绩: _____

实验与创新实践教育中心制

2025 年 10 月

注：本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

设计的功能描述

基本功能：先使用三段式状态机描述法实现 UART 的接收，并结合前面实现的模块，以 9600 的波特率，实现如下的收发功能：

1. 接收：从电脑端串口软件发送数据，FPGA 解析收到的数据送到数码管上显示。

(1) 只测试 ASCII 字符 0-F，之外的可以任意显示或者不显示。

数码管显示用的编码方式不限，只要做到收发两端数据一致，比如字符 A，若用 STR 模式发送数码管显示 A，也可以显示 A 对应的十六进制 41，或者自定义编码。

(2) 显示最近接收到的 6 个字符，不足 6 个高位不显示，对应数码管完全关闭而不是显示 0。

(3) 数码管剩余 2 位显示累计接收到的字符个数，超过 2 位数只显示低 2 位。

2. 发送：

(1) 按 S3，发送“个人学号”，比如“20240101”。

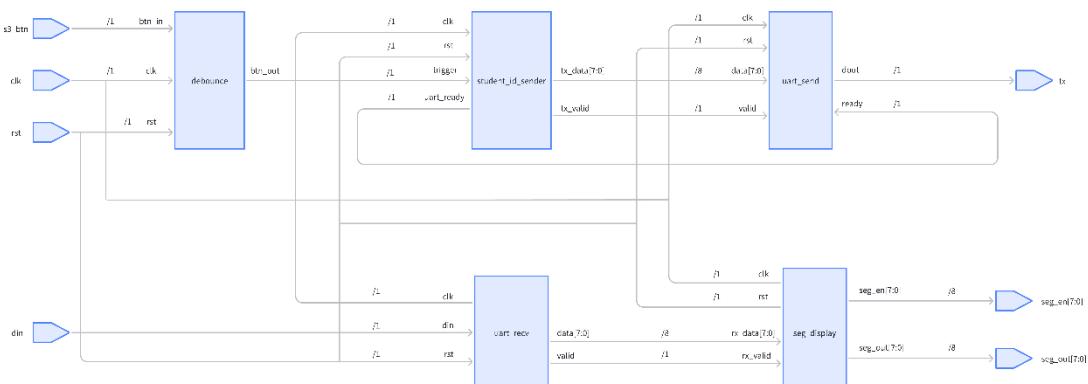
(2) 按键需要消抖，按一次发送一次。

(3) 按键开关 S1 作为异步复位信号。

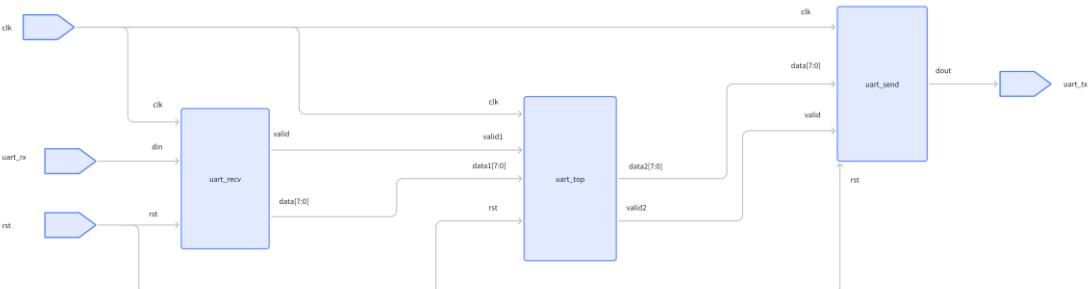
(4) 未明确要求的细节可根据自己的理解自行实现。

系统设计

用硬件框图描述系统主要功能及各模块之间的相互关系



附加题：



模块设计与实现

各子模块设计思路，及实现代码

(1) 设计思路：

采用自顶向下的结构化设计方法，将系统拆分为以下模块：

1. 顶层模块：实例化并连接所有子模块
2. UART 收发模块：处理串口通信协议
3. 数码管显示模块：控制 8 位数码管显示
4. 按键消抖模块：实现按键消抖和学号发送控制

拆分以后可以让功能变得专一，接口清晰，可测试性强，便于调试和修改。

(2) 本实验实现了基于 UART 串口通信的数字系统，主要功能包括：

1. UART 接收模块 (uart_recv.v)

接收功能：从电脑端串口软件接收数据，FPGA 解析后将数据显示在数码管上。采用三段式状态机设计。

关键逻辑：波特率：9600bps，基于 100MHz 时钟分频。状态机包含 IDLE、START、DATA、STOP 四个状态。在数据位中间时刻采样，提高稳定性。使用一个计数器，根据系统时钟频率和目标波特率(9600bps)生成接收时钟，并利用状态机在 DATA 状态，在每个位的采样点将串行输入数据移位到一个 8 位寄存器中实现接收解码代码：

```

reg din_sync1, din_sync2, din_sync3;

module uart_recv(
    input clk,
    input rst,
    input din,
    output reg valid,
    output reg [7:0] data
);

// -----
localparam IDLE = 3'b000;
localparam START = 3'b001;
localparam DATA = 3'b010;
localparam STOP = 3'b011;

// -----
localparam CLK_FREQ = 100_000_000; // 100MHz
localparam BAUD_RATE = 9600; // -----
localparam BIT_CYCLES = CLK_FREQ / BAUD_RATE;
localparam BIT_CYCLES_HALF = BIT_CYCLES / 2;

// -----
reg [2:0] current_state, next_state;
reg [13:0] bit_counter;
reg [2:0] bit_index;
reg [7:0] data_shift;

begin

    reg din_sync1, din_sync2, din_sync3;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            din_sync1 <= 1'b1;
            din_sync2 <= 1'b1;
            din_sync3 <= 1'b1;
        end else begin
            din_sync1 <= din;
            din_sync2 <= din_sync1;
            din_sync3 <= din_sync2;
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst)
            current_state <= IDLE;
        else
            current_state <= next_state;
    end

    always @(*) begin
        case (current_state)
            IDLE: next_state = (!din_sync2) ? START : IDLE;
            START: next_state = (bit_counter == BIT_CYCLES - 1) ? DATA : START;
            DATA: next_state = (bit_index == 3'd7 && bit_counter == BIT_CYCLES - 1) ? STOP : DATA;
            STOP: next_state = (bit_counter == BIT_CYCLES - 1) ? IDLE : STOP;
            default: next_state = IDLE;
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            bit_index <= 0;
        end else begin
            if (current_state == DATA) begin
                if (bit_counter == BIT_CYCLES - 1) begin
                    if (bit_index < 3'd7)
                        bit_index <= bit_index + 1;
                    else
                        bit_index <= 0;
                end
            end else begin
                bit_index <= 0;
            end
        end
    end

    always @(posedge clk or posedge rst) begin
        if (bit_index < BIT_CYCLES - 1)
            bit_index <= bit_index + 1;
        else
            bit_index <= 0;
    end

    always @(posedge clk or posedge rst) begin
        if (rst)
            data_shift <= 0;
        end else begin
            if (current_state == DATA) begin
                if (bit_counter == BIT_CYCLES_HALF - 1)
                    data_shift <= {din_sync3, data_shift[7:1]};
                end
            end else if (current_state == IDLE) begin
                data_shift <= 0;
            end
        end
    end
end
end

```

```

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            valid <= 1'b0;
        end else begin
            valid <= 1'b0; |

            if (current_state == STOP) begin
                if (bit_counter == BIT_CYCLES_HALF - 1) begin
                    valid <= 1'b1;
                end
            end
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            data <= 8'b0;
        end else begin
            if (current_state == STOP) begin
                if (bit_counter == BIT_CYCLES_HALF - 1) begin
                    data <= data_shift;
                end
            end
        end
    end
endmodule

```

2. UART 发送模块 (uart_send.v)

发送功能：通过 S3 按键出发发送个人学号“2024311668”。

关键逻辑：采用四状态状态机：IDLE、START、DATA、STOP。提供 ready 信号指示发送就绪状态。使用状态机控制 IDLE：发送线保持高电平。当收到发送请求时，锁存待发送数据，进入起始位状态。START：发送一个比特时间的低电平，作为起始位。DATA：从最低位到最高位，依次将 8 位数据串行输出。STOP：发送一个比特时间的高电平，作为停止位，然后回到 IDLE 状态。在 DATA 状态，使用一个计数器控制，依次将 8 位并行数据的每一位输出。

```

`timescale 1ns / 1ps

module uart_send(
    input clk,
    input rst,
    input valid,
    input [7:0] data,
    output reg dout,
    output reg ready
);

localparam IDLE = 2'b00;
localparam START = 2'b01;
localparam DATA = 2'b10;
localparam STOP = 2'b11;

localparam CLOCK_FREQ = 100_000_000;
localparam BAUD_RATE = 9600;
localparam DIVIDER = CLOCK_FREQ / BAUD_RATE - 1;

reg [1:0] current_state;
reg [1:0] next_state;

reg [13:0] baud_counter;
reg [2:0] bit_counter;
reg [7:0] shift_reg;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        current_state <= IDLE;
    end else begin
        current_state <= next_state;
    end
end
endmodule

```

```

    always @(*) begin
        case (current_state)
            IDLE: begin
                if (valid)
                    next_state = START;
                else
                    next_state = IDLE;
            end
            START: begin
                if (baud_counter == DIVIDER)
                    next_state = DATA;
                else
                    next_state = START;
            end
            DATA: begin
                if ((baud_counter == DIVIDER) && (bit_counter == 3'd7))
                    next_state = STOP;
                else
                    next_state = DATA;
            end
            STOP: begin
                if (baud_counter == DIVIDER)
                    next_state = IDLE;
                else
                    next_state = STOP;
            end
        default: next_state = IDLE;
        endcase
    end
end

```

```

        ready <= 1'b1;
    end else begin
        ready <= 1'b0;
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        dout <= 1'b1;
    end else begin
        case (current_state)
            IDLE: begin
                dout <= 1'b1;
            end

            START: begin
                dout <= 1'b0;
            end

            DATA: begin
                dout <= shift_reg[0];
            end

            STOP: begin
                dout <= 1'b1;
            end

            default: begin
                dout <= 1'b1;
            end
        endcase
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        ready <= 1'b1;
    end else begin
        if (current_state == IDLE && !valid) begin
            ready <= 1'b1;
        end else begin
            if (baud_counter == DIVIDER) begin
                baud_counter <= 14'd0;
            end else begin
                baud_counter <= baud_counter + 14'd1;
            end
        end
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        bit_counter <= 3'd0;
    end else begin
        case (current_state)
            IDLE: begin
                bit_counter <= 3'd0;
            end

            START: begin
                if (baud_counter == DIVIDER) begin
                    bit_counter <= 3'd0;
                end
            end

            default: begin
                bit_counter <= 3'd0;
            end
        endcase
        if (baud_counter == DIVIDER) begin
            bit_counter <= 3'd0;
        end
    end
end

DATA: begin
    if (baud_counter == DIVIDER) begin
        bit_counter <= bit_counter + 3'd1;
    end
end

STOP: begin
    if (baud_counter == DIVIDER) begin
        bit_counter <= 3'd0;
    end
end

default: begin
    bit_counter <= 3'd0;
end
endcase
end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        shift_reg <= 8'd0;
    end else begin
        case (current_state)
            IDLE: begin
                if (valid) begin
                    shift_reg <= data;
                end
            end

            DATA: begin
                if (baud_counter == DIVIDER) begin
                    shift_reg <= {1'b0, shift_reg[7:1]};
                end
            end

            default: begin
                end
            end
        endcase
        if (valid) begin
            shift_reg <= data;
        end
    end
end
endmodule

```

3. 数码管显示模块 (seg_display.v)

显示功能：控制 8 位数码管显示最近接收的 6 个字符和累计接收字符数。

关键逻辑：使用一个高速计数器循环产生位选信号 (seg_en)，依次点亮每一位

数码管。在点亮某一位的同时，向段选线 (seg_out) 发送该位对应的译码值。

只要扫描频率足够高，看起来就像是所有数码管同时稳定显示。

代码：

```

`timescale 1ns / 1ps

module seg_display(
    input clk,
    input rst,
    input [7:0] rx_data,
    input rx_valid,
    output reg [7:0] seg_en,
    output reg [7:0] seg_out
);

reg [7:0] display_buffer [7:0];
reg [7:0] char_buffer [5:0];
reg [7:0] char_count;
reg [19:0] scan_counter;
reg [2:0] scan_index;

reg rx_valid_prev;

function [7:0] seg_encoder;
    input [7:0] ascii_data;
begin
    case (ascii_data)
        8'h30: seg_encoder = 8'hC0; // '0'
        8'h31: seg_encoder = 8'hF9; // '1'
        8'h32: seg_encoder = 8'hA4; // '2'
        8'h33: seg_encoder = 8'hB0; // '3'
        8'h34: seg_encoder = 8'h99; // '4'
        8'h35: seg_encoder = 8'h92; // '5'
        8'h36: seg_encoder = 8'h82; // '6'
        8'h37: seg_encoder = 8'hF8; // '7'
        8'h38: seg_encoder = 8'h80; // '8'
        8'h39: seg_encoder = 8'h90; // '9'
    endcase
endfunction

always @(posedge clk or posedge rst) begin
    if (rst) begin
        rx_valid_prev <= 0;
    end else begin
        rx_valid_prev <= rx_valid;
    end
end

wire rx_posedge = rx_valid && !rx_valid_prev;
wire is_valid_char = ((rx_data >= 8'h30 && rx_data <= 8'h39) || // 0-9
                      (rx_data >= 8'h41 && rx_data <= 8'h46) || // A-F
                      (rx_data >= 8'h61 && rx_data <= 8'h66)); // a-f

always @(posedge clk or posedge rst) begin
    if (rst) begin
        char_count <= 0;
    end else begin
        if (rx_posedge && is_valid_char) begin
            char_count <= char_count + 1;
        end
    end
end

integer i;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < 6; i = i + 1)
            char_buffer[i] <= 8'h00;
    end else begin
        if (rx_posedge && is_valid_char) begin
            for (i = 0; i < 5; i = i + 1)
                char_buffer[i] <= char_buffer[i + 1];
            char_buffer[5] <= rx_data;
        end
    end
end

function [7:0] digit_to_seg;
    input [3:0] digit;
begin
    case (digit)
        4'h0: digit_to_seg = 8'hC0;
        4'h1: digit_to_seg = 8'hF9;
        4'h2: digit_to_seg = 8'hA4;
        4'h3: digit_to_seg = 8'hB0;
        4'h4: digit_to_seg = 8'h99;
        4'h5: digit_to_seg = 8'h92;
        4'h6: digit_to_seg = 8'h82;
        4'h7: digit_to_seg = 8'hF8;
        4'h8: digit_to_seg = 8'h80;
        4'h9: digit_to_seg = 8'h90;
    default: digit_to_seg = 8'hFF;
    endcase
endfunction

always @(posedge clk or posedge rst) begin
    char_buffer[5] <= 8'h00;
end else begin
    if (rx_posedge && is_valid_char) begin
        for (i = 0; i < 5; i = i + 1)
            char_buffer[i] <= char_buffer[i + 1];
        char_buffer[5] <= rx_data;
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < 8; i = i + 1)
            display_buffer[i] <= 8'hFF;
    end else begin
        for (i = 0; i < 6; i = i + 1) begin
            display_buffer[7-i] <= (char_buffer[i] != 8'h00) ? seg_encoder(char_buffer[i]) : 8'hFF;
        end
        display_buffer[1] <= digit_to_seg((char_count % 100) / 10);
        display_buffer[0] <= digit_to_seg(char_count % 10);
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        scan_counter <= 0;
    end else begin
        if (scan_counter < 100000) begin
            scan_counter <= scan_counter + 1;
        end else begin
            scan_counter <= 0;
        end
    end
end

```

```

        scan_counter <= 0;
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        scan_index <= 0;
    end else begin
        if (scan_counter == 100000) begin
            if (scan_index == 3'd7) begin
                scan_index <= 0;
            end else begin
                scan_index <= scan_index + 1;
            end
        end
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        seg_out <= 8'hFF;
    end else begin
        seg_out <= display_buffer[scan_index];
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        seg_en <= 8'b11111111;
    end else begin
        seg_en <= ~(8'b1 << scan_index);
    end
end
endmodule

```

4. 按键消抖模块 (debounce.v)

消抖功能：消除机械按键的抖动现象。

关键逻辑：将异步的按钮输入信号用三级寄存器同步到系统时钟域，检测到按钮电平变化后，启动一个计数器（延时 10ms）。在延时期间，如果按钮电平保持稳定，则认为是有效操作；如果发生抖动，则计数器清零并重新开始计时。在确认电平稳定后，检测其上升沿，产生一个单时钟周期的高电平脉冲作为有效按键信号。

代码：

```

`timescale 1ns / 1ps

module debounce(
    input clk,
    input rst,
    input btn_in,
    output reg btn_out
);

reg [19:0] counter;
reg btn_sync1, btn_sync2, btn_sync3;
reg btn_prev;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        btn_sync1 <= 0;
        btn_sync2 <= 0;
        btn_sync3 <= 0;
    end else begin
        btn_sync1 <= btn_in;
        btn_sync2 <= btn_sync1;
        btn_sync3 <= btn_sync2;
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        btn_prev <= 0;
    end else begin
        btn_prev <= btn_sync3;
    end
end

always @ (posedge clk or posedge rst) begin
    if (rst) begin
        counter <= 0;
    end else begin
        if (btn_prev != btn_sync3) begin
            counter <= 0;
        end else if (counter < 1000000) begin
            counter <= counter + 1;
        end
    end
end

if (counter == 1000000) begin
    btn_out <= 1;
end
endmodule

```

5. 学号发送控制模块 (student_id_sender.v)

学号发送功能：控制学号字符串的发送。

关键逻辑：检测按键触发，按顺序发送学号字符，与 UART 发送模块协调。

代码：

```

`timescale 1ns / 1ps

module student_id_sender(
    input clk,
    input rst,
    input trigger,
    input uart_ready,
    output reg [7:0] tx_data,
    output reg tx_valid
);
    localparam IDLE = 1'b0;
    localparam SENDING = 1'b1;

    reg state;
    initial begin
        student_id[0] = "2"; // ASCII '2'
        student_id[1] = "0"; // ASCII '0'
        student_id[2] = "2"; // ASCII '2'
        student_id[3] = "4"; // ASCII '4'
        student_id[4] = "3"; // ASCII '3'
        student_id[5] = "1"; // ASCII '1'
        student_id[6] = "1"; // ASCII '1'
        student_id[7] = "6"; // ASCII '6'
        student_id[8] = "6"; // ASCII '6'
        student_id[9] = "8"; // ASCII '8'
    end
    reg [3:0] send_index;
    reg sending;
    reg trigger_prev;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            trigger_prev <= 1'b0;
        end else begin
            trigger_prev <= trigger;
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            tx_valid <= 1'b0;
        end else begin
            case (state)
                IDLE: begin
                    if (!trigger_prev && trigger) begin
                        if (uart_ready) begin
                            tx_valid <= 1'b1;
                        end else begin
                            tx_valid <= 1'b0;
                        end
                    end else begin
                        tx_valid <= 1'b0;
                    end
                end
                SENDING: begin
                    if (tx_valid) begin
                        tx_valid <= 1'b0;
                    end else if (uart_ready) begin
                        if (send_index < 4'd9) begin
                            tx_valid <= 1'b1;
                        end else begin
                            tx_valid <= 1'b0;
                        end
                    end
                end
            endcase
        end
    end
endmodule

```

```

        end else begin
            tx_valid <= 1'b0;
        end
    end
endcase
end
default: begin
    tx_valid <= 1'b0;
end
endcase
end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        tx_data <= 8'b0;
    end else begin
        case (state)
            IDLE: begin
                if (!trigger_prev && trigger) begin
                    if (uart_ready) begin
                        tx_data <= student_id[0];
                    end
                end
            end
        end
    end
end

SENDING: begin
    if (!tx_valid && uart_ready) begin
        if (send_index < 4'd9) begin
            tx_data <= student_id[send_index + 1];
        end
    end
end
endcase
end
case (state)
    IDLE: begin
        if (!trigger_prev && trigger) begin
            if (uart_ready) begin
                sending <= 1'b1;
            end
        end
    end
    end
always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;
    end else begin
        case (state)
            IDLE: begin
                if (!trigger_prev && trigger) begin
                    if (uart_ready) begin
                        state <= SENDING;
                    end
                end
            end
        end
    end
end

SENDING: begin
    if (!tx_valid && uart_ready) begin
        if (send_index >= 4'd9) begin
            sending <= 1'b0;
        end
    end
end
endcase
end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        state <= IDLE;
    end else begin
        case (state)
            IDLE: begin
                if (!trigger_prev && trigger) begin
                    if (uart_ready) begin
                        state <= SENDING;
                    end
                end
            end
        end
    end
end

SENDING: begin
    if (!tx_valid && uart_ready) begin
        if (send_index >= 4'd9) begin
            state <= IDLE;
        end
    end
end
endcase
end
endmodule

```

6. 顶层模块 (top_uart.v)

模块功能：实例化并连接所有子模块。

```

`timescale 1ns / 1ps

module top_uart(
    input clk,
    input rst,
    input din,
    input s3_btn,
    output tx,
    output [7:0] seg_en,
    output [7:0] seg_out
);

    wire [7:0] uart_rx_data;
    wire uart_rx_valid;
    wire [7:0] student_id_data;
    wire student_id_valid;
    wire btn_debounced;
    wire uart_send_ready;

    uart_recv u_uart_recv(
        .clk(clk),
        .rst(rst),
        .din(din),
        .valid(uart_rx_valid),
        .data(uart_rx_data)
    );
    debounce u_debounce(
        .clk(clk),
        .rst(rst),
        .btn_in(s3_btn),
        .btn_out(btn_debounced)
    );
    uart_send u_uart_send(
        .clk(clk),
        .rst(rst),
        .valid(student_id_valid),
        .data(student_id_data),
        .dout(tx),
        .ready(uart_send_ready)
    );
    student_id_sender u_student_id(
        .clk(clk),
        .rst(rst),
        .trigger(btn_debounced),
        .uart_ready(uart_send_ready),
        .tx_data(student_id_data),
        .tx_valid(student_id_valid)
    );
    seg_display u_seg_display(
        .clk(clk),
        .rst(rst),
        .rx_data(uart_rx_data),
        .rx_valid(uart_rx_valid),
        .seg_en(seg_en),
        .seg_out(seg_out)
    );
endmodule

```

附加题：图1：状态定义。图二-图四：状态转移逻辑。图五：输出逻辑。

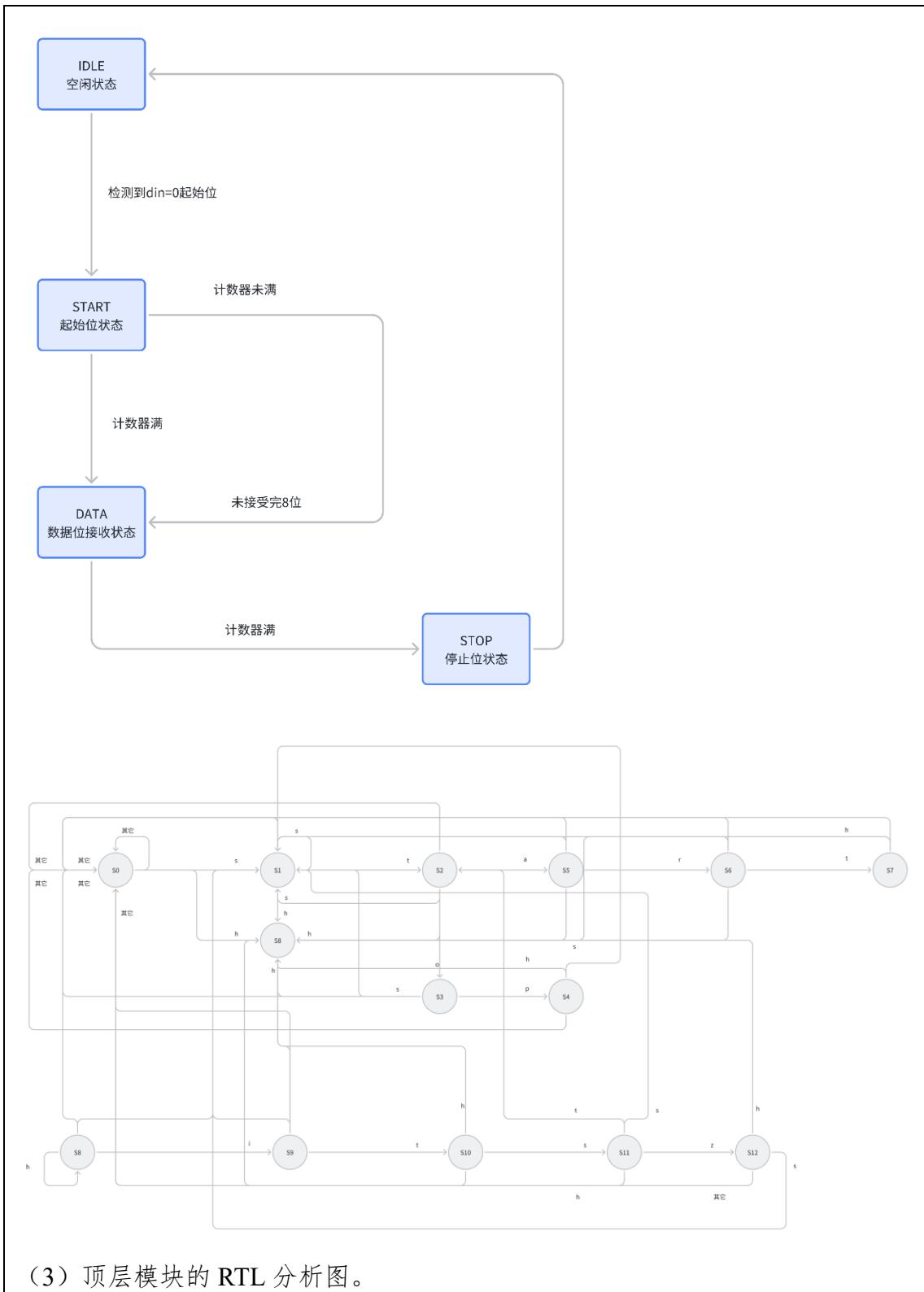
```

localparam S0 = 4'b0000;
localparam S1 = 4'b0010;
localparam S2 = 4'b0010;
localparam S3 = 4'b0011;
localparam S4 = 4'b0100;
localparam S5 = 4'b0101;
localparam S6 = 4'b0110;
localparam S7 = 4'b0111;
localparam S8 = 4'b1000;
localparam S9 = 4'b1001;
localparam S10 = 4'b1010;
localparam S11 = 4'b1011;
localparam S12 = 4'b1100;
localparam S = 8'h73; // 's' - 115 (0x73)
localparam T = 8'h74; // 't' - 116 (0x74)
localparam O = 8'h6F; // 'o' - 111 (0x6F)
localparam P = 8'h70; // 'p' - 112 (0x70)
localparam A = 8'h61; // 'a' - 97 (0x61)
localparam R = 8'h72; // 'r' - 114 (0x72)
localparam H = 8'h68; // 'h' - 104 (0x68)
localparam I = 8'h69; // 'i' - 105 (0x69)
localparam Z = 8'h7A; // 'z' - 122 (0x7A)

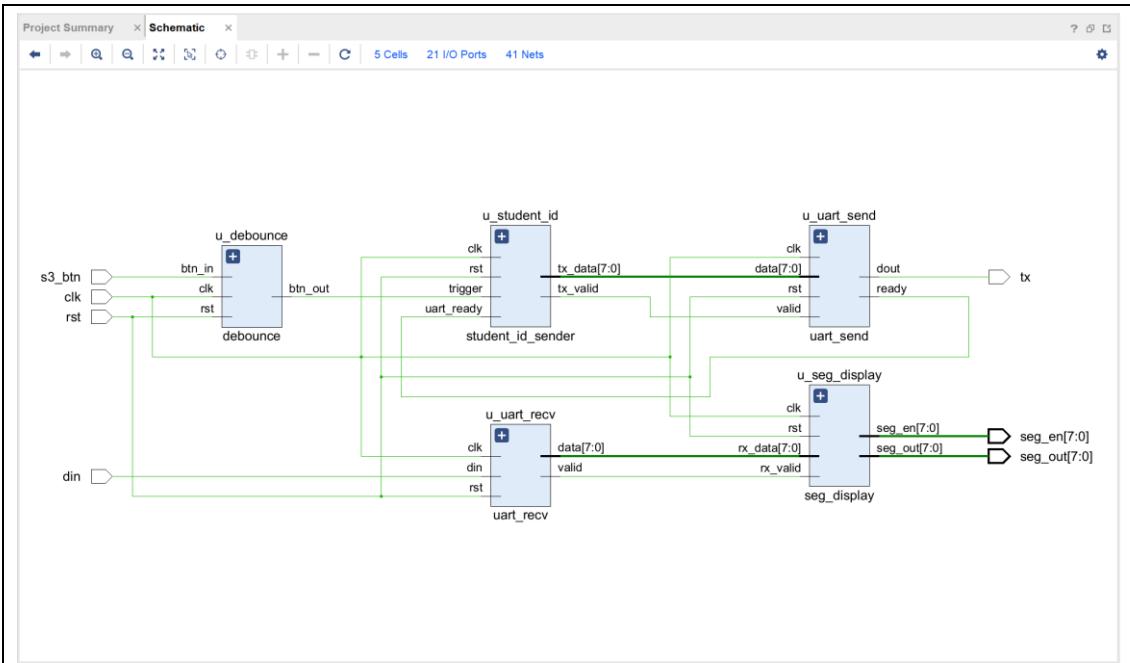
always @(*) begin
    if(!valid1)next_state = current_state;
    else if(baud_done)next_state = S0;
    else begin
        case (current_state)
            S0: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else next_state = S0;
            end
            S1: begin
                if (data1==T)next_state = S2;
                else if(data1==S)next_state = S8;
                else if(data1==H)next_state = S8;
                else next_state = S0;
            end
            S2: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==O)next_state = S3;
                else if(data1==A)next_state = S5;
                else next_state = S0;
            end
            S3: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==P)next_state = S4;
                else next_state = S0;
            end
            S4: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==R)next_state = S6;
                else next_state = S0;
            end
            S5: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==T)next_state = S7;
                else next_state = S0;
            end
            S6: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==T)next_state = S7;
                else next_state = S0;
            end
            S7: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else next_state = S0;
            end
            S8: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==I)next_state = S9;
                else next_state = S0;
            end
            S9: begin
                if (data1==S)next_state = S1;
                else if(data1==H)next_state = S8;
                else if(data1==T)next_state = S10;
                else next_state = S0;
            end
        endcase
    end
    if (data1==S)next_state = S1;
    else if(data1==H)next_state = S8;
    else next_state = S0;
end
endcase
end

```

(2) UART 接收模块要有状态转移图及说明，要求参考前面 UART 发送。如附加题的功能也用到了状态机，也给出对应的状态转移图、代码截图和说明。



(3) 顶层模块的 RTL 分析图。



仿真调试

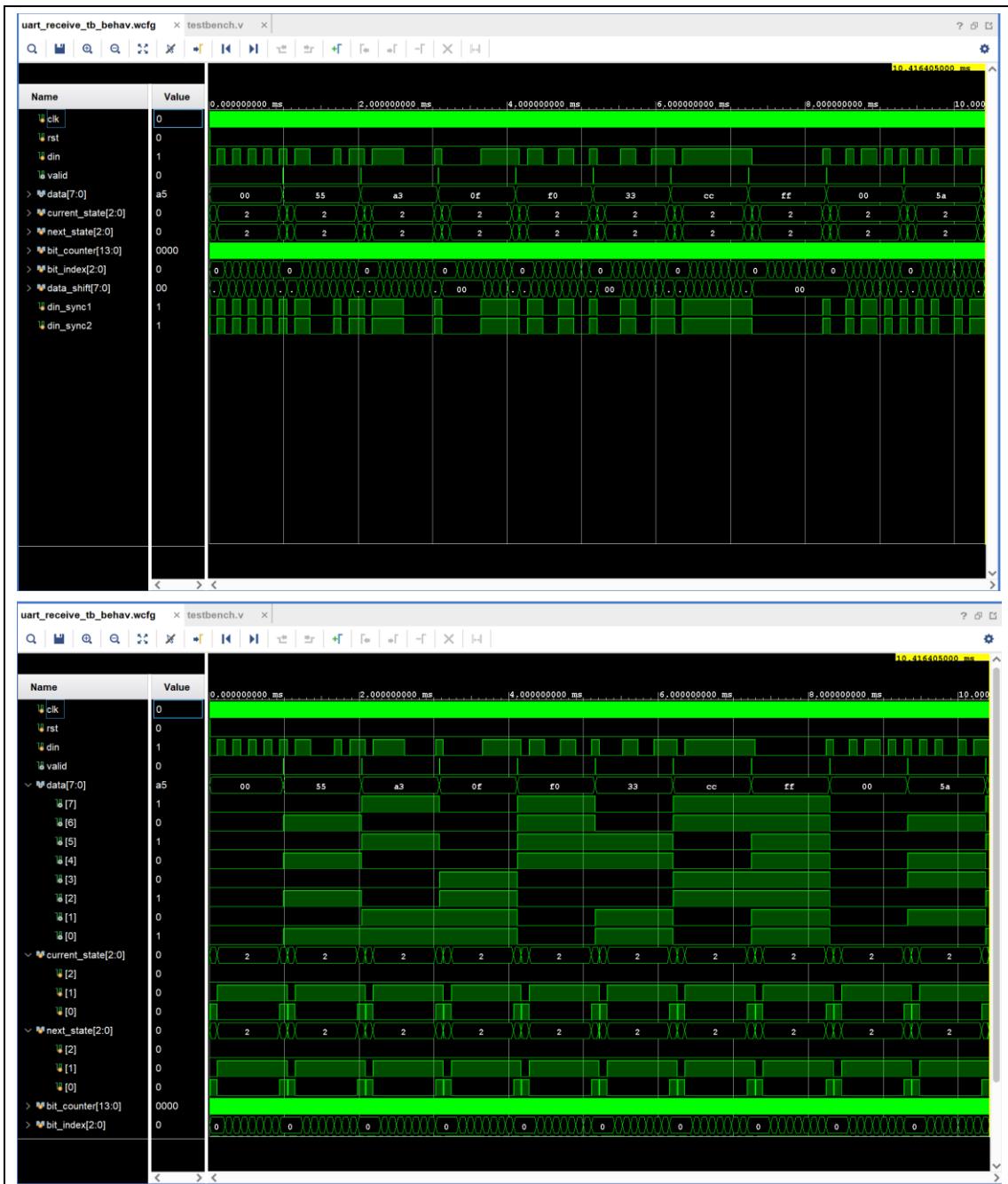
(1) Tcl Console, 仿真波形截图及仿真分析

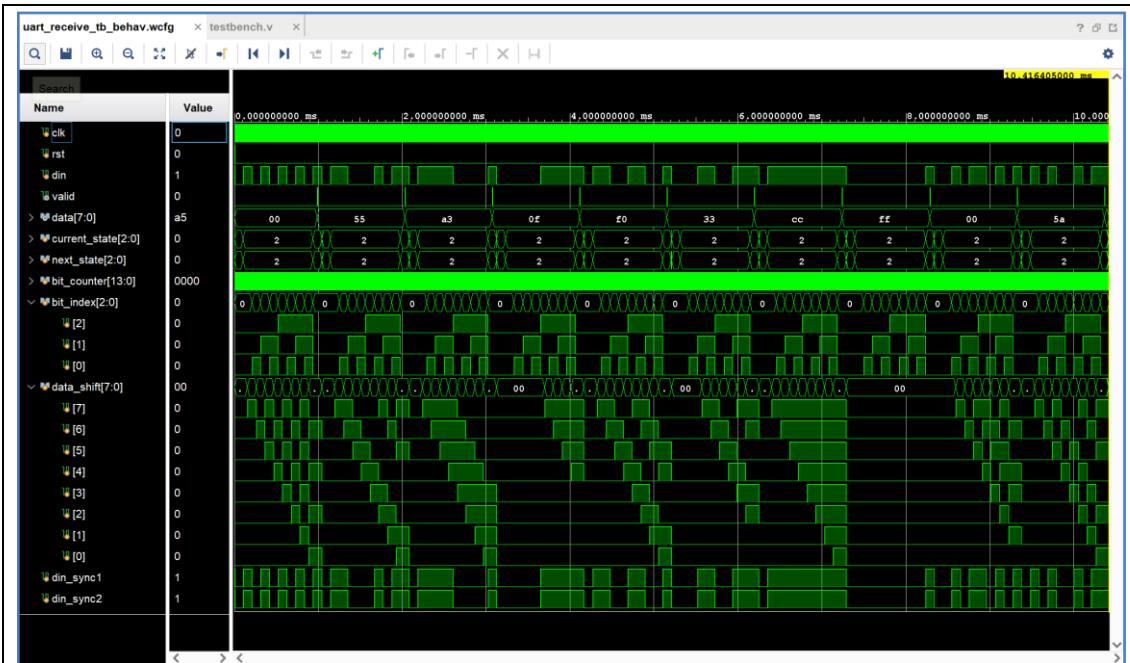
```

# set curr_wave [current_wave_config]
# if ( [string length $curr_wave] == 0 ) {
#   if ( [llength [get_objects]] > 0 ) {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a wave window go to 'File->New Waveform' or press F4"
#   }
# }

# run 1000ns
Uart baud rate =      9600, freq divider =      10416
INFO: [USF-XSim-96] XSim completed. Design snapshot 'uart_receive_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cput = 00:00:02 ; elapsed = 00:00:05 . Memory (MB): peak = 1663.809 ; gain = 17.664
update_compile_order -fileset sources_1
run all
Test case: 0 success, Received expected data = 0x55
Test case: 1 success, Received expected data = 0xa3
Test case: 2 success, Received expected data = 0xf
Test case: 3 success, Received expected data = 0xf0
Test case: 4 success, Received expected data = 0x33
Test case: 5 success, Received expected data = 0xcc
Test case: 6 success, Received expected data = 0xff
Test case: 7 success, Received expected data = 0x00
Test case: 8 success, Received expected data = 0x5a
Test case: 9 success, Received expected data = 0xa5
All test cases passed.
Finish called at time : 10416405 ns : File "C:/Users/gcgst/Desktop/exercise6/exercise6.sim_1/new/testbench.v" Line 73

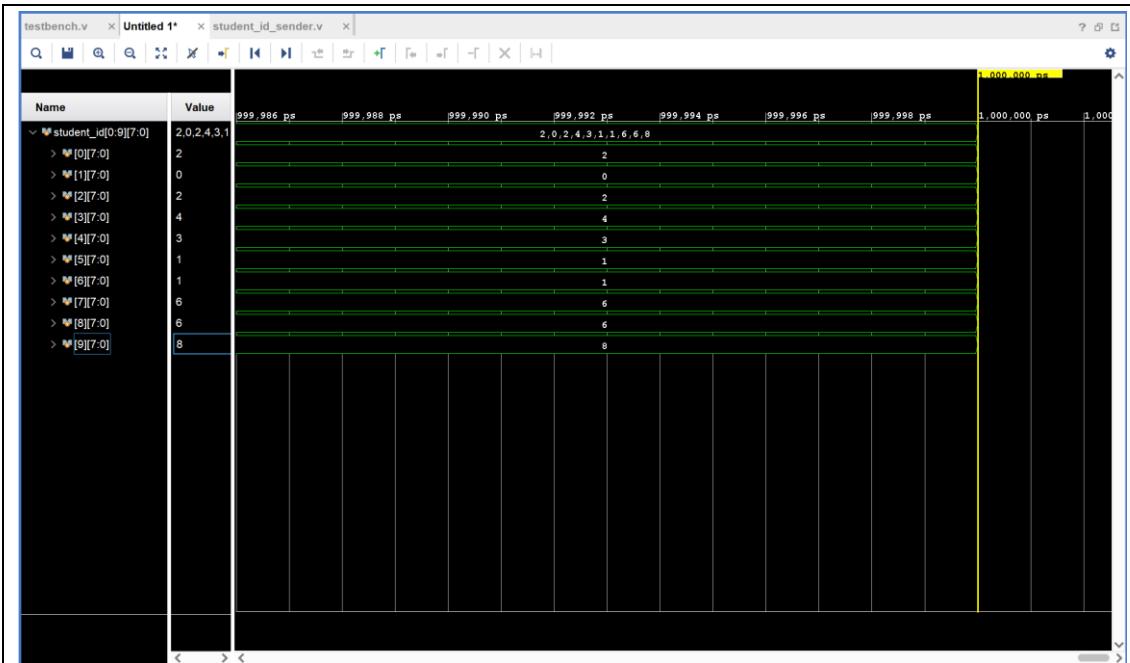
```





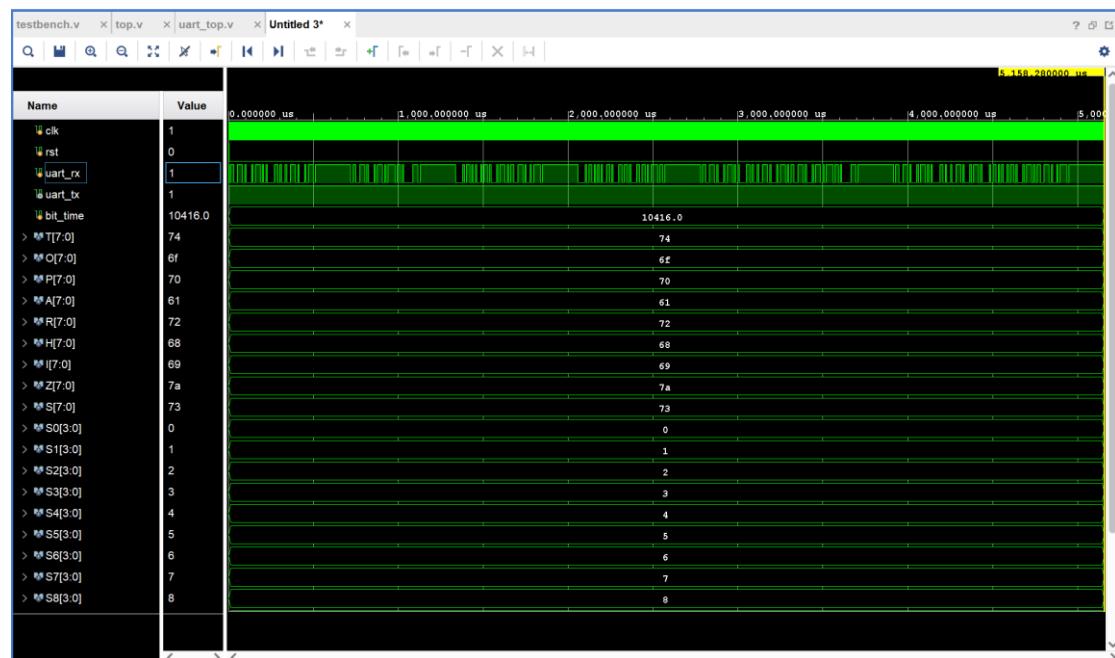
状态转移时序：初始状态：current_state 处于 IDLE(3'b000)状态，等待起始位。
 起始位检测：当 din 出现起始位（低电平）时，状态立即从 IDLE 跳转为 START(3'b001)。valid 信号在此时保持为 0，表示数据尚未有效。
 数据接收阶段：在 START 状态，当 bit_counter 计数到中间位置（5207）时，确认起始位有效。状态从 START 转移到 DATA(3'b010)，开始接收 8 位数据。bit_index 从 0 开始计数，标识当前接收的数据位位置。
 数据位采样：在 DATA 状态的每个位周期中间时刻（bit_counter=5207）。采样 din 信号并移入 data_shift 寄存器。bit_index 在每个数据位接收完成后递增。
 停止位处理：当 bit_index 计数到 7 且 bit_counter 计数满时，状态转移到 STOP(3'b011)。在 STOP 状态的中间时刻，设置 valid 信号为 1，表示数据有效。将 data_shift 内容输出到 data 总线。
 返回空闲：STOP 状态结束后，状态返回 IDLE，准备接收下一个数据帧。valid 信号恢复为 0，所有计数器复位。
 计数器工作分析：bit_counter 计数器。在非 IDLE 状态中，bit_counter 从 0 递增到 10415 为每个位周期提供精确的时间基准在状态转换时自动复位。中间点（5207）用于数据采样，结束点（10415）用于状态转换。仅在 DATA 状态中工作，范围 0-7，每个数据位接收完成后递增。
 标识当前接收的数据位在字节中的位置。当计数到 7 时表示 8 位数据接收完成。

(2) 个人学号发送的功能仿真



波形分析：数据位 0-7 分别显示学号的八位“2024311668”。已改为 ASCII 码显示。

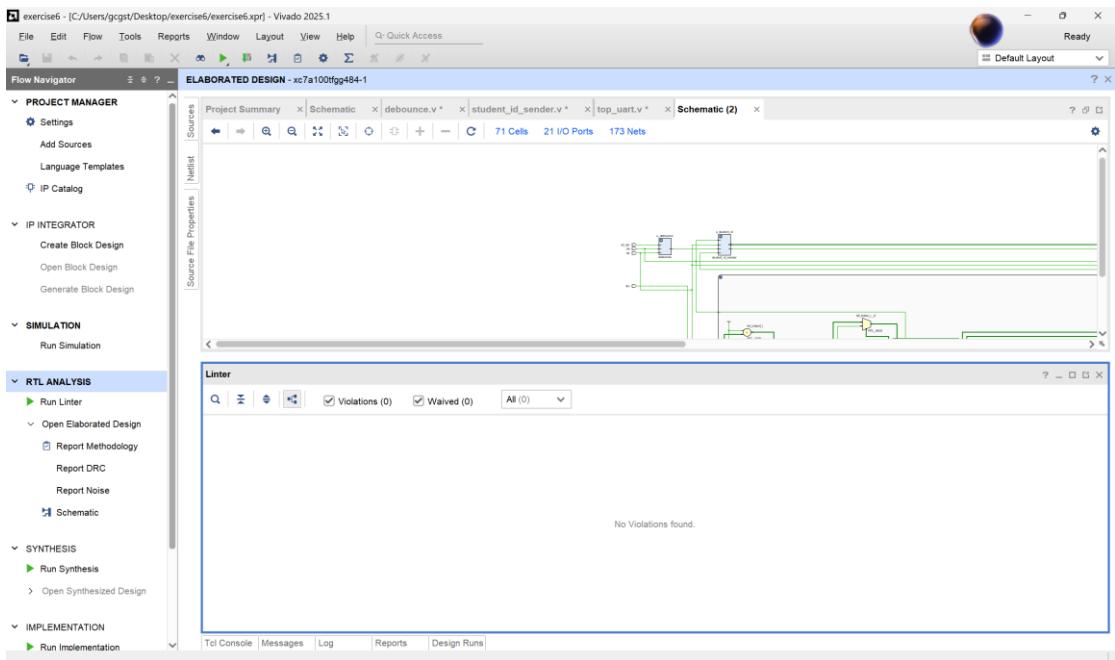
附加题仿真波形：



- [测试 1] 发送: start 收到回复: 1 (匹配 start)
- [测试 2] 发送: stop 收到回复: 2 (匹配 stop)
- [测试 3] 发送: hitsz 收到回复: 3 (匹配 hitsz)
- [测试 4] 发送: hello 收到回复: 0 (未匹配)
- [测试 5] 连续发送: startstop 收到回复: 1 (匹配 start) 收到回复: 2 (匹配 stop)
- [测试 6] 发送: starthitsz 收到回复: 1 (匹配 start) 收到回复: 3 (匹配 hitsz)

设计过程中遇到的问题及解决方法、Lint 分析

(1) Run Linter 报告截图



(2) 实验中碰到的代码逻辑或者规范性问题。

之前会在一个 always 块里面放多个变量，看似简洁但是很不符合硬件代码规范，比如说 always @(posedge clk or posedge rst) begin

```

if (rst) begin
    tx_valid <= 1'b0;
    tx_data <= 8'b0;
    send_index <= 4'b0;
    sending <= 1'b0;
    trigger_prev <= 1'b0;
    state <= IDLE;
end else begin
    trigger_prev <= trigger;
    tx_valid <= 1'b0; // 默认无效

    case (state)
        IDLE: begin
            // 检测触发信号的上升沿
            if (!trigger_prev && trigger) begin
                if (uart_ready) begin // 确保 UART 发送模块就绪
                    state <= SENDING;
                    send_index <= 4'b0;
                    tx_valid <= 1'b1;
                    tx_data <= student_id[0];
                end
            end
        end
    end

```

```

SENDING: begin
    if (tx_valid) begin
        // 当前字符已发送，等待下一个机会
        tx_valid <= 1'b0;
    end else if (uart_ready) begin
        // UART 就绪，发送下一个字符
        if (send_index < 4'd9) begin
            send_index <= send_index + 1;
            tx_valid <= 1'b1;
            tx_data <= student_id[send_index + 1];
        end else begin
            // 所有字符发送完成
            state <= IDLE;
            sending <= 1'b0;
        end
    end
    // 如果 uart_ready 为 0，则等待
end
endcase
end
end

```

这时候应该把他们拆分成多个 always 块，每个 always 块中只输出一个变量。比如：always @ (posedge clk or posedge rst) begin

```

if (rst) begin
    tx_valid <= 1'b0;
    tx_data <= 8'b0;
    send_index <= 4'b0;
    sending <= 1'b0;
    trigger_prev <= 1'b0;
    state <= IDLE;
end else begin
    trigger_prev <= trigger;
    tx_valid <= 1'b0; // 默认无效
end

case (state)
    IDLE: begin
        // 检测触发信号的上升沿
        if (!trigger_prev && trigger) begin
            if (uart_ready) begin // 确保 UART 发送模块就绪
                state <= SENDING;
                send_index <= 4'b0;
                tx_valid <= 1'b1;
                tx_data <= student_id[0];
            end
        end
    end
endcase

```

```

        end
    end
end

SENDING: begin
    if (tx_valid) begin
        // 当前字符已发送，等待下一个机会
        tx_valid <= 1'b0;
    end else if (uart_ready) begin
        // UART 就绪，发送下一个字符
        if (send_index < 4'd9) begin
            send_index <= send_index + 1;
            tx_valid <= 1'b1;
            tx_data <= student_id[send_index + 1];
        end else begin
            // 所有字符发送完成
            state <= IDLE;
            sending <= 1'b0;
        end
    end
    // 如果 uart_ready 为 0，则等待
end
endcase
end
end

```

AI 工具使用总结

(1) 实验课程中是否有用 HiAgent 的实验助手？有哪些体会和建议？

否

(2) 其他还用了哪些 AI 工具？

豆包, deepseek, kimi

(3) 所有 AI 工具使用的频率怎么样？比如每天对话 20 次以上、每天偶尔用等。

平均每个任务需要对话 10 到 15 次左右

(4) 是如何使用的？比如以下的情况，也可另行说明。

对于一些不太熟悉的任务会先让 AI 生成一些代码去学习，对照着课程报告里的实验原理搞明白后自己再去用自己的理解编写代码，彻底编写完后在之后的综合上板等遇到问题会再尝试用 AI 进行调试

(5) 总结：使用 AI 工具的体会、收获、建议等。

AI 确实能够帮助我们去更好的理解一些新的知识，但是容易让我们忽略一些比较现实的问题，实际上板子之后出现的问题 AI 是解决不了的，还是需要自己去思考并发现问题进行修改

课程设计总结

接触到了 verilog 语言以及 vivado 软件，以前写软件是一步一步执行，学了 Verilog 才明白硬件是“并行干活”的。比如做加法器，不是一个数一个数加，而是所有位同时算，这种并行思维得慢慢适应。我学会了利用仿真工具分析波形、定位逻辑错误，提升了硬件调试能力，能发现并解决状态机跳转异常、信号赋值不完整等常见问题。也知道了代码写得规范多重要，不然硬件调试起来没头没脑。也记住了设计时得考虑资源和性能，不能光顾着功能对。这些经验我认为对我以后的学习习惯和思路上会有很大的帮助，是非常有价值的一次实验经历。

希望课容量稍微小一点。