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

实验报告

开课学期：2024 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：T2612

学生班级：7 班

学生学号：2023311709

学生姓名：宁中昊

评阅教师：

报告成绩：

实验与创新实践教育中心制

2024 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

概述基本功能

主要分为两个功能：

1. 数据发送（uart_send）

使用拨码开关 SW7-SW0，输入十六进制 ASCII 码，通过串口软件 Supercom 发送到电脑端。

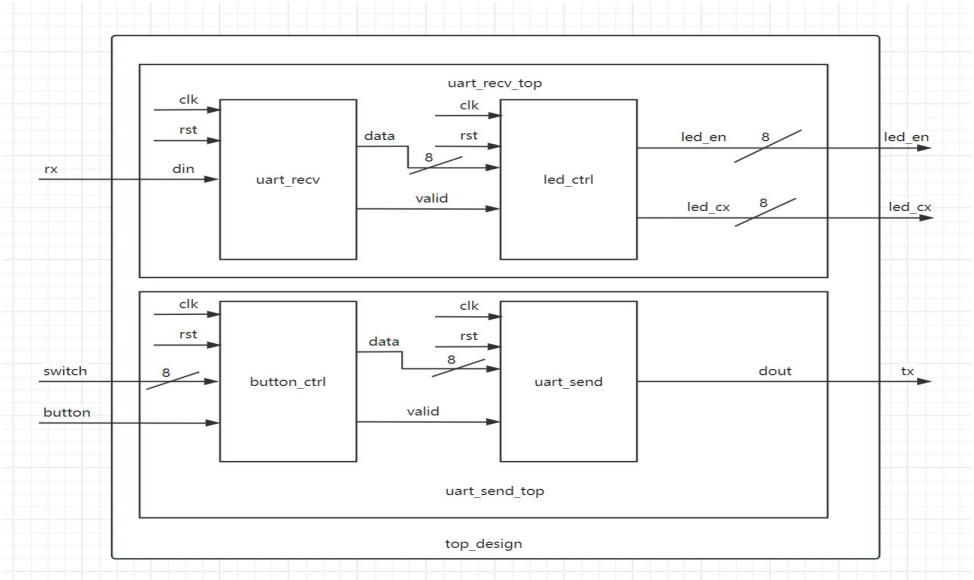
2. 数据接收（uart_recv）

使用串口软件 Supercom 发送十六进制数据（不带换行），用开发板接收，并将电脑端发送的数据显示在数码管上。

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

一、系统设计的硬件框图



二、文字说明

各模块功能：

（1）顶层模块

分为三个部分，分别对应于接收数据、发送数据与最顶层的功能连接。一开始的构思是两个功能共用一个顶层模块，但是由于接收与发送两个功能相对独立，且都其顶层设计都需要两个模块相连，因此将两者分开，先在对应的顶层模块连接好实现功能再一起连接到最顶层。

第一部分 top_design：最顶层模块，连接 uart_recv_top 模块与 uart_send_top 模块，将两部分功能的模块连接到一起。这两个部分是相对独立的，uart_top 向 top_design 传入 LED 控制的相关数据 led_en 与 led_cx 并由 top_desgin 连接到对应数码管的管脚，uart_send 通过 tx 向 top_desgin 传入待发送数据，由 top_design 连接到 UART 功能的发送管脚。

第二部分 uart_recv_top：接收功能的顶层模块，连接实现接收功能的两个模块 uart_recv 与 led_ctrl，将 UART 接收端口的信号 rx 接入 uart_recv 模块的输入信号 din，再将其产生的输出信号 data 与有效信号 valid 输入 led_ctrl 模块中，输出对应的 LED 控制信号 led_en 与 led_cx。

第三部分 uart_send_top：发送功能的顶层模块，将拨码开关对应的 8 位数据 switch 与按钮 S3 的信号 button 输入 uart_send，再将 button_ctrl 模块产生的输出信号 dout 与有效信号 valid 接到 uart_send。

（2）接收功能模块

总体上分为两部分：接收实现 uart_recv 与数码管控制 led_ctrl，而在 led_ctrl 中，对数码管显示数字 0-9 与字母 A-F 对应的 led_cx 赋值的操作经常重复使用，且涉及到大量常量的赋值，多次重复书写没有意义，因此单独将其提取出来成为一个新的模块 led_display，由于其未涉及功能实现没有将其纳入框图之中。

uart_recv：接收信号，并将其转换为 8 位信号 data，每一次接收 8 位信号完毕时拉高有效信号 valid。

led_ctrl：将 data 中表示的数据显示在数码管上，只显示最近接收的 8 个字符。不足 8 个字符高位不显示。

led_display：根据输入其中的信号产生对应数据为显示在数码管上的管脚控制信号。

（3）发送功能模块

分为两个部分：发送实现 uart_send 与按钮控制 button_ctrl，其中 uart_send 模块与实验 4 相同，button_ctrl 中的大部分功能与实验 3 中相同。

uart_send：button_ctrl 每给到一次高位有效信号，就将拨码开关表示的十六进制 ASCII 码对应的数据发送出去。

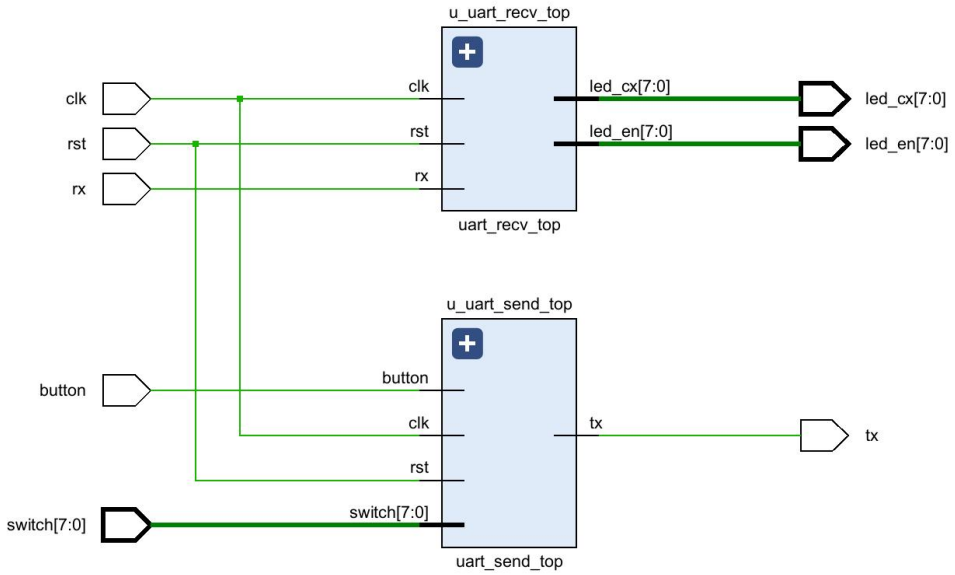
button_ctrl：每次按下按钮 S3 就拉高一次有效信号 valid。

模块设计与实现

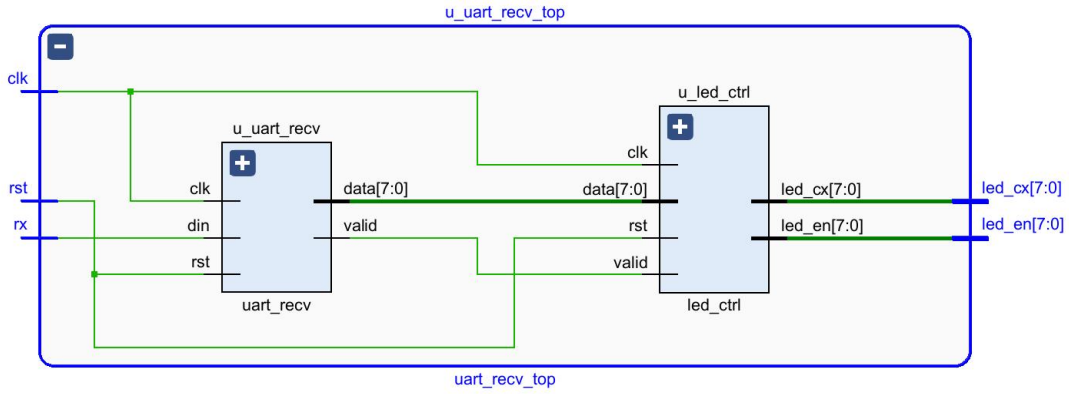
包括各子模块设计思路，输入、输出端口及关键代码

一、顶层模块 RTL 分析图

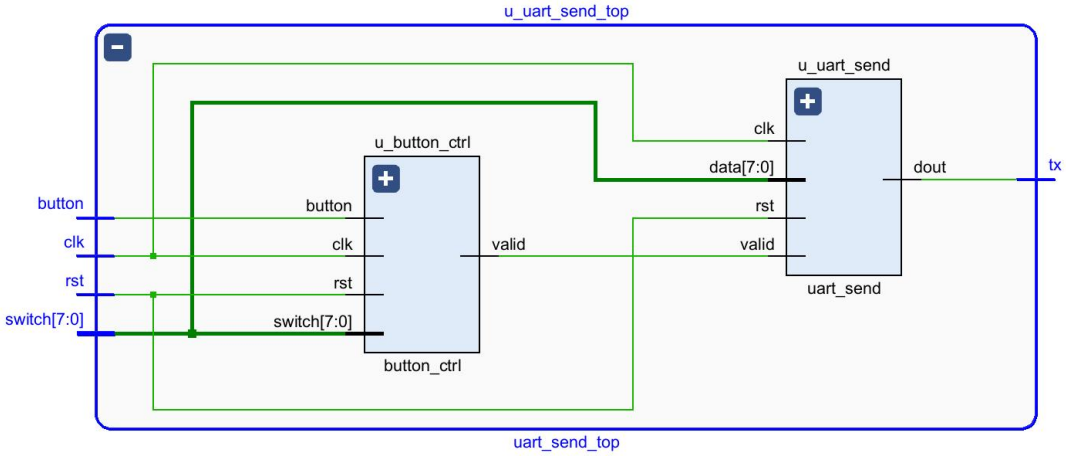
（1）top_design



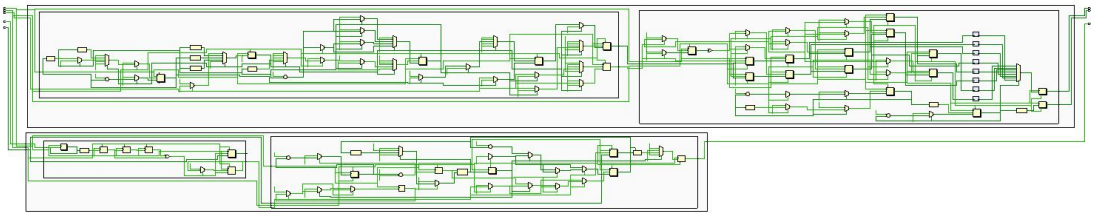
（2）uart_recv_top



（3）uart_send_top



总体显示：



二、子模块设计思路、输入输出端口与关键代码

（1）uart_recv

设计思路：三段式实现状态机，重难点在于描述好状态转移条件，以控制采样时间在信号中段。

使用一个计数器来控制波特率，在起始状态达到计数到波特率周期的一半的时候进入数据状态，接着以一个波特率周期为时间间隔进行采样，如此则可以控制所有采样点在信号中段。

输入端口：clk、rst、din

输出端口：valid、data

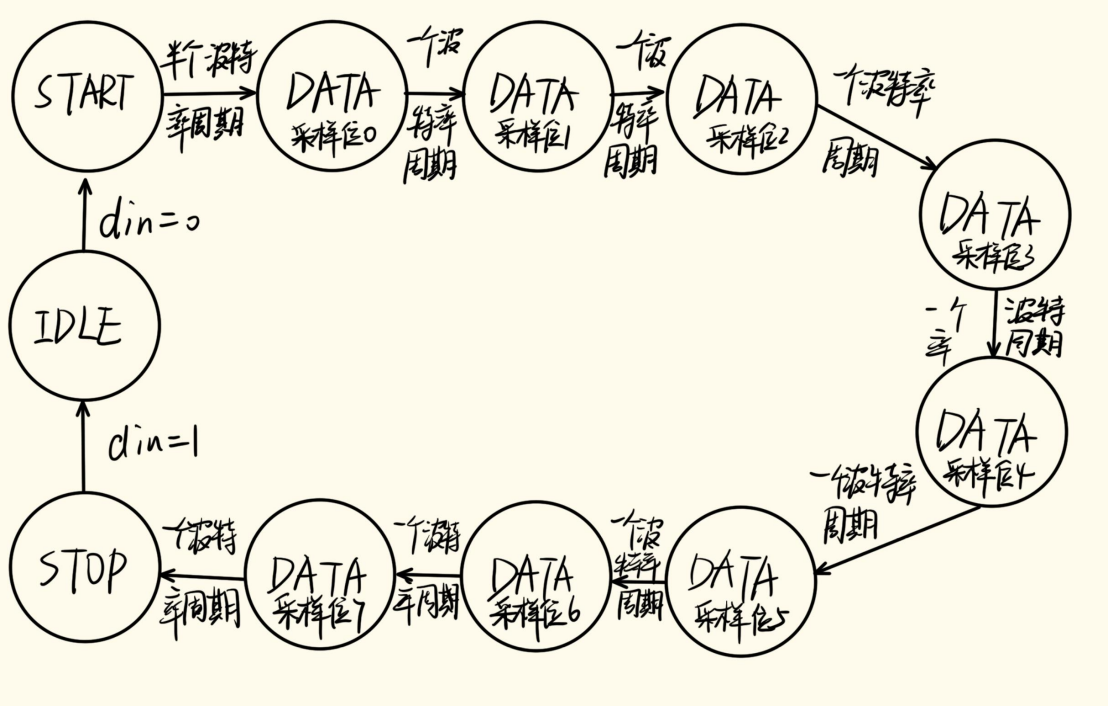
关键代码：

```
//* 三段式实现状态机
// 第 1 个 always 块，描述次态迁移到现态
always @(posedge clk or posedge rst) begin
    if (rst) current_state <= IDLE;
    else     current_state <= next_state;
end

// 第 2 个 always 块，描述状态转移条件判断
always @(*) begin
    case (current_state)
        // 空闲状态：接收到起始信号 din == 0 后进入起始状态
        IDLE: begin
            if(din == 0) begin
                next_state = START;
            end else next_state = IDLE;
        end
        // 起始状态：持续半个波特率，后进入数据状态
        // 保证之后采样间隔为一个波特率的同时，在信号的中段采样
        START: begin
            if(baud_counter == BAUD_HALF) begin
                next_state = DATA;
            end else next_state = START;
        end
        // 数据状态：传入 8 位数据后进入停止状态
        DATA: begin
            if (bit_index == 8) begin
                next_state = STOP;
            end else next_state = DATA;
        end
        // 停止状态：继续采样，接受到停止信号 din == 1 后转入
空闲状态
        STOP: begin
            if(baud_counter == BAUD_END) begin
                next_state = IDLE;
            end else next_state = STOP;
        end
        default: next_state = IDLE;
    endcase
end

// 第 3 个 always 块，描述输出逻辑
always @(posedge clk or posedge rst) begin
```

```
        if(rst) begin
            data <= 0;
            data_saved <= 0;
            valid <= 0;
            baud_counter <= 0;
        end else begin
            case(current_state)
                // 空闲状态：复位计数器与有效信号
                IDLE: begin
                    valid <= 0;
                    baud_counter <= 0;
                end
                // 起始状态：计数半个波特率，完成后复位波特率计数
器与位计数器
                START: begin
                    if(baud_counter == BAUD_HALF) begin
                        baud_counter <= 0;
                        bit_index <= 0;
                    end else begin
                        baud_counter <= baud_counter + 1;
                    end
                end
                // 数据状态：以波特率为间隔采样，逐位传入数据
                DATA: begin
                    if(baud_counter == BAUD_END) begin
                        data_saved[bit_index] <= din;
                        baud_counter <= 0;
                        if(bit_index == 8) begin
                            baud_counter <= 0;
                        end else begin
                            bit_index <= bit_index + 1;
                        end
                    end else begin
                        baud_counter <= baud_counter + 1;
                    end
                end
                // 停止状态：继续采样，接收到停止信号 din == 1 后
将 valid 拉高，并将暂存的数据赋值给输出端口
                STOP: begin
                    if(baud_counter == BAUD_END) begin
                        if(din == 1) begin
                            baud_counter <= 0;
                            data <= data_saved;
                            valid <= 1;
```

```
                    end else baud_counter <= 0;
                end else baud_counter = baud_counter + 1;
            end
            default: begin
                valid <= 0;
                data <= 0;
            end
        endcase
    end
end
```

状态转移图：



（2）led_ctrl

设计思路：基本思路是 LED 轮询工作，但是实现过程中还是有一些重难点。

重点：串口软件 Supercom 只能发送十六位数据，而十六进制包含了两个八位数据，因此需要两个数据两个数据地保存更新。解决方

案：将接收到的数据拆分为两段，分别保存。

难点：会出现大量重复且没有意义的赋值代码，很麻烦，且会导致按钮按下两次才能正确地在数码管上显示。解决方案：使用循环，将重复的赋值语句拆分为新的 led_display 模块。

输入端口：clk、rst、data、valid

输出端口：led_en、led_cx

关键代码：

```
// 串口软件只能发送十六进制信号，也就是一次会发送两个信号，所以需要将一次接收两个信号
always @(posedge clk or posedge rst) begin
    if (rst) begin
        flag <= 0;
        data_former <= 5'h1f;
        data_latter <= 5'h1f;
        buffer_cnt <= 0;
        for(i=0; i<8; i=i+1) begin
            buffer[i] <= 5'h1f;
        end
    end else if (valid) begin
        data_former <= data[7:4];
        data_latter <= data[3:0];
        flag <= 1;
    end else if (flag) begin
        for (n=7; n>1; n=n-1) begin
            buffer[n] <= buffer[n-2];
        end
        buffer_cnt <= buffer_cnt +1;
        buffer[1] <= data_former;
        buffer[0] <= data_latter;
        flag <= 0;
    end
end

// 循环实例化 led_display 模块，将数据转化为 led_cx
genvar q;
generate
    for(q=0; q<8; q=q+1) begin: u_led_display
```

```
            led_display u_led_display(
                .data      (buffer  [q]),
                .led_ctrl_cx(data_to_led[q])
        );
      end
    endgenerate

    // LED 灯轮询工作
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            led_cnt <= 0;
            led_index <= 0;
        end else begin
            if (led_cnt == REFRESH_RATE) begin
                led_cnt <= 0;
                if(led_index == 7) begin
                    led_index <= 0;
                end else led_index <= led_index + 1;
            end else begin
                led_cnt = led_cnt + 1;
          end
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            led_cx <= 8'b11111111;
        end else begin
            led_cx <= data_to_led[led_index];
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            led_en <= 8'b11111111;
        end else begin
            case(led_index)
                3'd0: led_en <= 8'b11111110;
                3'd1: led_en <= 8'b11111101;
                3'd2: led_en <= 8'b11111011;
                3'd3: led_en <= 8'b11110111;
                3'd4: led_en <= 8'b11101111;
                3'd5: led_en <= 8'b11011111;
                3'd6: led_en <= 8'b10111111;
```

```
                3'd7: led_en <= 8'b01111111;
            endcase
        end
    end
```

（3）led_display

设计思路：将循环的赋值语句拆分出来。

输入端口：data

输出端口：led_ctrl_cx

关键代码：

```
    always @(*) begin
        if (data == 5'h1f) led_ctrl_cx = 8'b11111111; // 不显示
        else begin
            case (data)
                4'h0:    led_ctrl_cx = 8'b00000011; // 0
                4'h1:    led_ctrl_cx = 8'b10011111; // 1
                4'h2:    led_ctrl_cx = 8'b00100101; // 2
                4'h3:    led_ctrl_cx = 8'b00001101; // 3
                4'h4:    led_ctrl_cx = 8'b10011001; // 4
                4'h5:    led_ctrl_cx = 8'b01001001; // 5
                4'h6:    led_ctrl_cx = 8'b01000001; // 6
                4'h7:    led_ctrl_cx = 8'b00011111; // 7
                4'h8:    led_ctrl_cx = 8'b00000001; // 8
                4'h9:    led_ctrl_cx = 8'b00001001; // 9
                4'ha:    led_ctrl_cx = 8'b00010001; // A
                4'hb:    led_ctrl_cx = 8'b11000001; // B
                4'hc:    led_ctrl_cx = 8'b11100101; // C
                4'hd:    led_ctrl_cx = 8'b10000101; // D
                4'he:    led_ctrl_cx = 8'b01100001; // E
                4'hf:    led_ctrl_cx = 8'b01110001; // F
                default: led_ctrl_cx = 8'b11111111; // 不显示
            endcase
        end
    end
```

（4）uart_send

设计思路：三段式实现状态机，与实验 4 相同。

输入端口：clk、rst、valid、data

输出端口：dout

关键代码：

```
// * 三段式实现状态机
// 第 1 个 always 块，描述次态迁移到现态
always @(posedge clk or posedge rst) begin
    if (rst) current_state <= IDLE;
    else current_state <= next_state;
end

// 第 2 个 always 块，描述状态转移条件判断
always @(*) begin
    if (baud_check) begin
        if (start_flag) next_state = START;
        else begin
            case (next_state)
            IDLE: begin
                next_state = IDLE;                          //
空闲状态 => 空闲状态
            end
            START: begin
                next_state = DATA;                          //
起始状态 => 数据状态
            end
            DATA: begin
                if (bit_index == 7) next_state = STOP;  //
数据状态 => 停止状态
                else next_state = DATA;                     //
否则继续发送数据位
            end
            STOP: begin
                next_state = IDLE;                          //
停止状态 => 空闲状态
            end
            default: next_state = IDLE;                     //
默认：空闲
            endcase
```

```
            end
        end
    end

    // 逐位输出数据
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            bit_index <= 3'b0;
        end else if (valid) bit_index =0;
        else if (baud_check) begin
            if (current_state == DATA)
                bit_index <= bit_index + 1;                    //
数据位计数器加一
        end
    end

    // 第 3 个 always 块，描述输出逻辑
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            dout <= 1'b1;
        end else begin
            case (current_state)
            IDLE:    dout <= 1'b1;                             //
空闲状态发送持续的高电平
            START:   dout <= 1'b0;                             //
起始状态发送一位低电平
            DATA:    dout <=
data_saved[bit_index];         // 数据状态逐位发送数据
            STOP:    dout <= 1'b1;                             //
停止状态发送一位高电平
            default: dout <= 1'b1;                             //
默认：空闲状态
            endcase
        end
    end
```

（5）button_ctrl

设计思路：进行按键消抖，与上升沿检测。

输入端口：clk、rst、button、switch

输出端口：valid、data

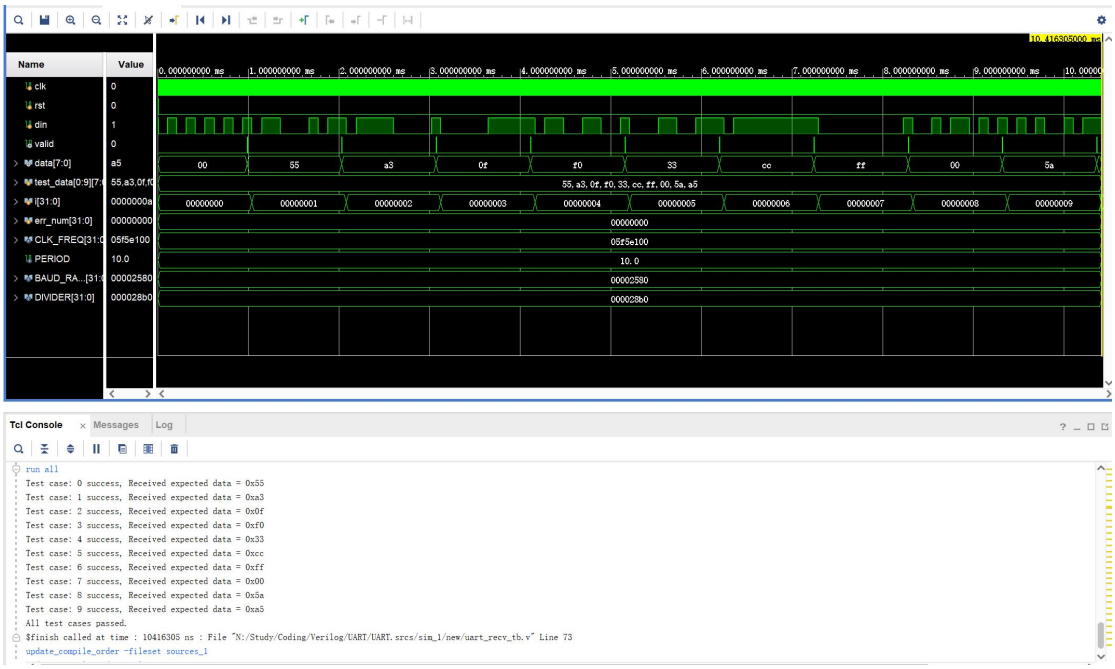
关键代码：

```
// button 按键去抖
always @(posedge clk) begin
  debounce_shift_reg <= {debounce_shift_reg[14:0],
button};
end

assign button_status = (debounce_shift_reg ==
16'hFFFF) ? 1'b1 : 1'b0;

// button 上升沿检测
always @ (posedge clk or posedge rst) begin
    if(rst) sig_r0 <= 1'b0;
    else    sig_r0 <= button_status;
end

always @ (posedge clk or posedge rst) begin
    if(rst) sig_r1 <= 1'b0;
    else    sig_r1 <= sig_r0;
end

always @ (posedge clk or posedge rst) begin
    if(rst) sig_r2 <= 1'b0;
    else    sig_r2 <= sig_r1;
end

assign button_edge_detect = ~sig_r2 & sig_r0;
```

调试报告

仿真波形截图及仿真分析

仿真波形截图：



如图，通过全部的十组仿真数据测试。

仿真分析：

如图，在每个数据测试中，接收模块在 IDLE 状态直接进入 START 状态，在这里经过半个波特率周期采样低电平信号 `din==0`，接着进入 DATA 状态，每隔一个波特率周期采样一次，刚好在信号中段采样。经过 8 位数据的采样后，进入 STOP 状态，停留一个波特率周期采样到高电平信号 `din==1`，同时拉高有效信号 `valid` 两个周期。重新进入 IDLE 状态，进入下一次数据的测试。

设计过程中遇到的问题及解决方法

现象：

原本正常通过测试，但是在修改了代码格式、添加注释与改动了由现态转移到次态的条件之后，第一个数据正常，第二个数据以后由于 valid 信号没有正常拉高，不能正常无法通过仿真测试。

分析过程：

正常通过测试表明基本逻辑没有问题，应当是转移条件有误。由于查看仿真波形图 valid 没有正常拉高，且第一个数据没有问题，首先猜测是进入 STOP 状态的条件有误，检查无果。

接着猜测进入 STOP 状态后没有正常进入 IDLE 状态，故意做出一些错误修改以作为调试，发现错误的修改对仿真结果没有影响，排除。

然后由于第一个信号正常，此时想到可能是 DATA 状态采样后没有将 bit_index 复位，发现的确没有，但是仍然无法解决。

最后发现 DATA 数据在 `bit_index==7` 的时候就进行了复位，导致数据一直不足 8 位因而无法进入 STOP 状态拉高 valid，问题解决。

错误原因：

1. DATA 状态采样后没有将 bit_index 复位；

2. DATA 数据在 `bit_index==7` 的时候就进行了复位，导致数据一直不足 8 位因而无法进入 STOP 状态拉高 valid。

```
//* 三段式实现状态机
// 第 1 个 always 块，描述次态迁移到现态
always @(posedge clk or posedge rst) begin
    if (rst) current_state <= IDLE;
```

```
        else    current_state <= next_state;
    end

    // 第 2 个 always 块，描述状态转移条件判断
    always @(*) begin
        case (current_state)
            // 空闲状态：接收到起始信号 din == 0 后进入起始状态
            IDLE: begin
                if(din == 0) begin
                    next_state = START;
                end else next_state = IDLE;
            end
            // 起始状态：持续半个波特率，后进入数据状态
            // 保证之后采样间隔为一个波特率的同时，在信号的中段采样
            START: begin
                if(baud_counter == BAUD_HALF) begin
                    next_state = DATA;
                end else next_state = START;
            end
            // 数据状态：传入 8 位数据后进入停止状态
            DATA: begin
                if (bit_index == 8) begin
                    next_state = STOP;
                end else next_state = DATA;
            end
            // 停止状态：继续采样，接受到停止信号 din == 1 后转入
空闲状态
            STOP: begin
                if(baud_counter == BAUD_END) begin
                    next_state = IDLE;
                end else next_state = STOP;
            end
            default: next_state = IDLE;
        endcase
    end

    // 第 3 个 always 块，描述输出逻辑
    always @(posedge clk or posedge rst) begin
        if(rst) begin
            data <= 0;
            data_saved <= 0;
            valid <= 0;
            baud_counter <= 0;
        end else begin
```

```
            case(current_state)
                // 空闲状态：复位计数器与有效信号
                IDLE: begin
                    valid <= 0;
                    baud_counter <= 0;
                end
                // 起始状态：计数半个波特率，完成后复位波特率计数器与位计数器
                START: begin
                    if(baud_counter == BAUD_HALF) begin
                        baud_counter <= 0;
                    end else begin
                        baud_counter <= baud_counter + 1;
                    end
                end
                // 数据状态：以波特率为间隔采样，逐位传入数据
                DATA: begin
                    if(baud_counter == BAUD_END) begin
                        data_saved[bit_index] <= din;
                        baud_counter <= 0;
                        if(bit_index == 7) begin
                            baud_counter <= 0;
                        end else begin
                            bit_index <= bit_index + 1;
                        end
                    end else begin
                        baud_counter <= baud_counter + 1;
                    end
                end
                // 停止状态：继续采样，接收到停止信号 din == 1 后将 valid 拉高，并将暂存的数据赋值给输出端口
                STOP: begin
                    if(baud_counter == BAUD_END) begin
                        if(din == 1) begin
                            baud_counter <= 0;
                            data <= data_saved;
                            valid <= 1;
                        end else baud_counter <= 0;
                    end else baud_counter = baud_counter + 1;
                end
                default: begin
                    valid <= 0;
                    data <= 0;
                end
```

```
            endcase
        end
    end
```

解决方案：

1. 在 START 状态复位 bit_index；
2. 将条件 bit_index==7 改为 bit_index==8。

课程设计总结

实验共用 12 小时，期中 8 小时写代码，4 小时写报告。

课程收获：

学会了使用 Verilog 来进行时序逻辑电路与组合逻辑电路的电路，理解了模块化设计电路的

总结：在写代码之前应该首先进行模块设计，保证思路清晰。

建议：实验开课时间和考试周重复了，重心一定会优先考试。