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：T2506

学生班级：计算机与电子通信 3 班

学生学号：2023311323

学生姓名：咸浩洋

评阅教师：

报告成绩：

实验与创新实践教育中心制

2024 年 10 月

设计的功能描述
概述基本功能 **1. top.v** • **功能**：顶层模块，连接其他子模块。管理发送和接收模块的输入输出。 **2. led.v** • **功能**：控制 LED 状态，接收串行输入并更新显示数据。 **3. led_display.v** • **功能**：根据输入数据和 LED 使能信号控制 LED 显示。 **4. uart_recv.v** • **功能**：实现 UART 接收功能，从串行输入中接收数据并在接收完成后输出有效信号。 **5. sw_cnt.v** • **功能**：将 8 个开关的状态转换为输出，表示为 8 位数据。 **6. sw_display.v** • **功能**：处理开关输入并在检测到有效信号时将数据传输到 UART 发送模块。 **7. uart_send.v** • **功能**：实现 UART 发送功能，将 8 位数据通过串行方式发送。模块在状态机控制下运行，分为 IDLE、START、DATA 和 STOP 四个状态。

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

1. top

- **功能**：顶层模块，连接各个子模块。负责处理输入输出信号的管理
- **数据流动**：将输入信号传递给 led 和 sw_display 模块，将其输出连接到 LED 显示和数据发送

2. led

- **功能**：控制 LED 的状态，处理从 uart_recv 接收的数据并更新 LED 显示
- **数据流动**：接收 UART 数据，更新内部数据和 LED 计数，最终将数据传递给 display 模块以进行可视化显示

3. led_display

- **功能**：显示数据到 LED，通过将数据转换为 LED 显示格式
- **数据流动**：根据输入数据和 LED 使能信号，更新 LED 状态

4. uart_recv

- **功能**：实现 UART 串行数据接收。接收串行输入信号 din，并在接收到完整数据后将其输出为 8 位数据
- **数据流动**：通过状态机监测输入信号。接收到起始位后，计数时钟以采样数据位，最后触发 valid 信号，输出接收到的数据

5. sw_cnt

- **功能**：计数开关输入，输出当前开关状态

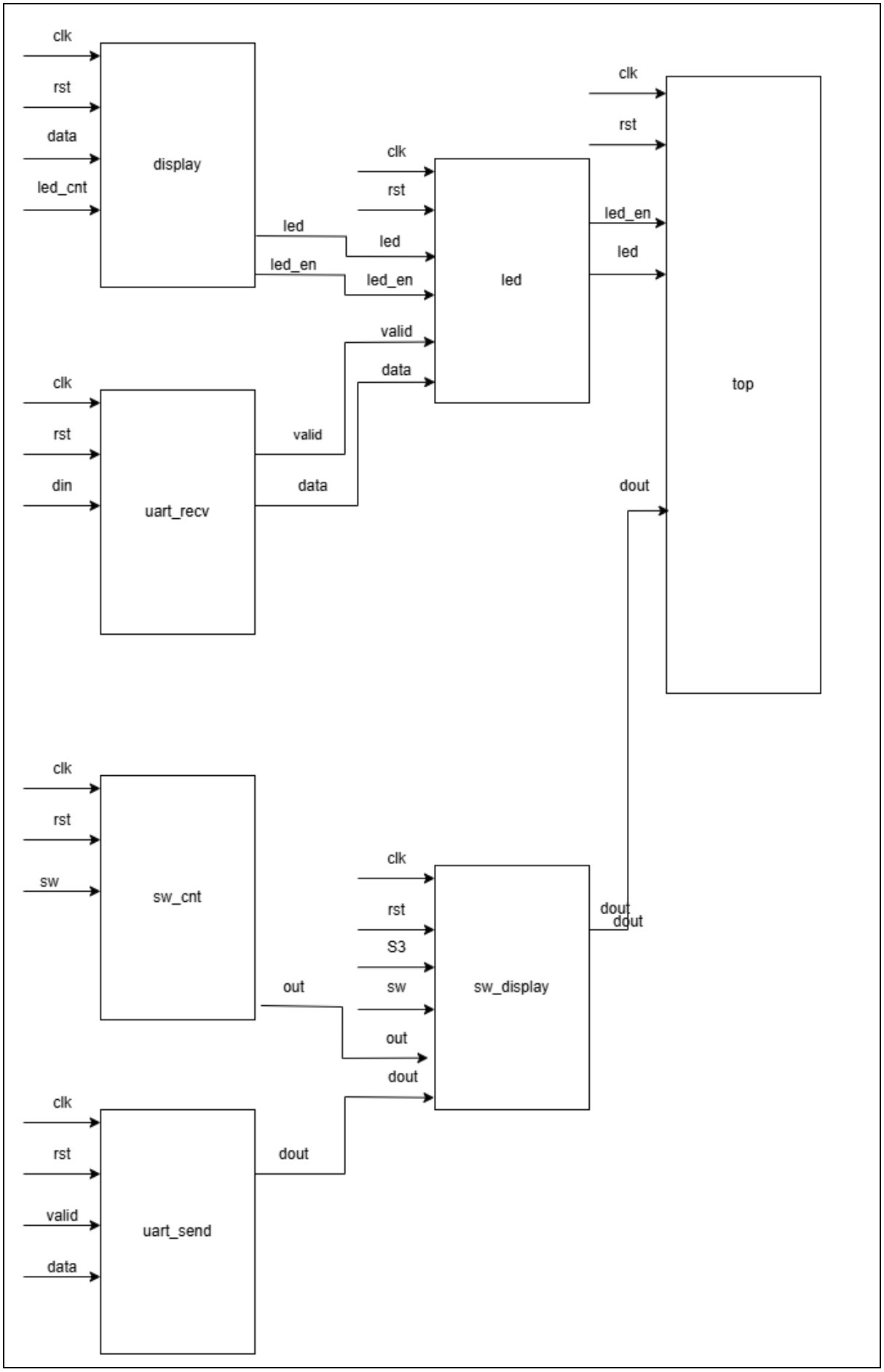
- **数据流动**：每次时钟上升沿检查开关状态，更新输出 out

6. sw_display

- **功能**：处理开关输入，并在按下 S3 时将开关状态发送到 uart_send 模块
- **数据流动**：将 8 位开关输入 sw 中的状态传递给 uart_send 模块，valid 信号指示数据有效性

7. uart_send

- **功能**：实现 UART 串行数据发送。接收时钟、复位、有效信号和待发送数据，将数据转换为串行形式输出
- **数据流动**：当 valid 信号为高时，模块开始发送 data。数据通过状态机控制发送过程，包括起始位、数据位（8 位）和停止位。输出为 dout


clk
rst
data
led_cnt
display
led
led_en
clk
rst
led
led_en
led
valid
data
clk
rst
din
uart_recv
valid
data
clk
rst
led_en
led
top
dout
clk
rst
sw
sw_cnt
out
clk
rst
S3
sw
out
dout
sw_display
dout
dout
clk
rst
valid
data
uart_send
dout

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

1. 顶层模块 top

功能

top 模块是整个设计的顶层模块，负责连接各个子模块并处理输入输出信号。它接收时钟信号 clk 和复位信号 rst，并将输入信号传递给相应的子模块。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - S3: 开关信号
  - sw[7:0]: 8 位开关输入
  - din: 数据输入信号
- **输出端口**:
  - dout: 数据输出信号
  - led_en[7:0]: LED 使能信号
  - led[7:0]: LED 状态信号

关键代码

```
23  module top(
24
25      input wire clk,
26      input wire rst,
27      //send
28      input wire S3,
29      input [7:0] sw,
30      output wire dout,
31      //recv
32      input din,
33      output [7:0] led_en,
34      output [7:0] led
35    );
36
37    led u_led(
38          .clk(clk),
39          .rst(rst),
40          .din(din),
41          .led_en(led_en),
42          .led(led)
43        );
44
45    sw_display u_sw_display(
46                  .clk(clk),
47                  .rst(rst),
48                  .sw(sw),
49                  .S3(S3),
50                  .dout(dout)
51                );
52  endmodule
53
```

2. 模块 led

功能

led 模块负责接收数据输入 din，处理数据并控制 LED 使能信号和状态信号。它将输入数据转换为 LED 显示的内容，并通过状态信号控制 LED 的亮灭。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - din: 数据输入信号
- **输出端口**:
  - led_en[7:0]: LED 使能信号
  - led[7:0]: LED 状态信号

关键代码

```
23  module led(
24      input clk,
25      input rst,
26      input din,
27      output reg [7:0] led_en,
28      output reg [7:0] led
29    );
30
31    wire [7:0] data;
32    wire valid;
33    reg [31:0] data_to_send;
34    wire [7:0] _led_en;
35    wire [7:0] _led;
36    reg[7:0] led_cnt;
37
38    always@(posedge clk or posedge rst)
39    begin
40      if(rst)
41      begin
42        data_to_send <= 32'b0;
43        led_cnt <= 8'b0;
44      end
45      else
46      begin
47        if(valid)
48        begin
49          case (data)
50            8'h30:
51            begin
52              data_to_send <= {data_to_send[27:0],4'b0000};
53              led_cnt <= led_cnt << 1 | 1;
54            end
55            8'h31:
56            begin
57              data_to_send <= {data_to_send[27:0],4'b0001};
58              led_cnt <= led_cnt << 1 | 1;
59            end
```

```
160             // 报告中省略其他case
161             default:
162               led_cnt <= led_cnt << 1 | 1;
163           endcase
164         end
165       end
166     end
167
168     always @(*)
169     begin
170       led_en = _led_en;
171       led = _led;
172     end
173
174     display u_display(
175               .clk(clk),
176               .rst(rst),
177               .data(data_to_send),
178               .led_cnt(led_cnt),
179               .led(_led),
180               .led_en(_led_en)
181             );
182     uart_recv u_uart_recv(
183                 .clk(clk),
184                 .rst(rst),
185                 .valid(valid),
186                 .data(data),
187                 .din(din)
188               );
189   endmodule
190
```

3. 模块 display

功能

display 模块负责将输入的数据转换为适合于 LED 显示的格式。它根据 LED 计数决定哪些 LED 需要显示，并根据输入数据生成相应的 LED 状态信号。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - data[31:0]: 输入数据，包含要显示的内容
  - led_cnt[7:0]: LED 计数，指示哪些 LED 需要被激活
- **输出端口**:
  - led_en[7:0]: LED 使能信号，指示哪个 LED 被使能
  - led[7:0]: LED 状态信号，表示当前显示的内容

主要实现逻辑

1. **数据转换**: 使用 case 语句将 4 位输入数据映射到对应的 LED 显示格式。
2. **LED 使能**: 通过 led_cnt 控制哪些 LED 被激活。
3. **时间控制**: 通过计数器 time_cnt 和信号 time_end 来控制 LED 的轮换显示。
4. **状态更新**: 在时钟上升沿更新 LED 显示和使能信号。

关键代码

```
22  module display(
23      input wire       clk,
24      input wire       rst,
25      input [31:0] data,
26      input [7:0] led_cnt,
27      output reg [7:0] led_en,
28      output reg [7:0] led
29    );
30
31    reg  time_step;
32    reg  time_end;
33    reg [24:0] time_cnt;
34    reg [7:0] tdata[7:0];
35    integer i;
36
37    always @(*)
38    begin
39      for(i=0;i<8;i = i + 1)
40      begin
41        if(led_cnt[i])
42        begin
43          case(data[(4*i)+:4])
44            4'b0000:
45              tdata[i] = 8'b1100_0000;   // 0
46            4'b0001:
47              tdata[i] = 8'b1111_1001;   // 1
```

```
76              // 报告中省略其他case
77              default:
78                tdata[i] = 8'b1111_1111;
79            endcase
80          end
81          else
82          begin
83            tdata[i] = 8'b1111_1111;
84          end
85        end
86      end
87      always @(*)
88      begin
89        time_end = (time_cnt == 20'd100000) & time_step;
90      end
91
92      always @(posedge clk or posedge rst)
93      begin
94        if (rst)
95          time_step <= 1'b1;
96        else
97          time_step <= 1'b1;
98      end
99
100     always @(posedge clk or posedge rst)
101     begin
102       if (rst)
103         time_cnt <= 1'b0;
104       else if(time_end)
105         time_cnt <= 1'b0;
106       else
107         time_cnt <= time_cnt + time_step;
108     end
109
110     always @(posedge clk)
111     begin
112       if (rst)
113         led_en <= 8'b1111_1111;
114       if (led_en == 8'b1111_1111 && ~rst)
115         led_en <= 8'b0111_1111;
116       if (time_end)
117       begin
118         led_en <= {led_en[6:0], led_en[7]};
```

```
119       end
120     end
121
122     always @(posedge clk)
123     begin
124       if (rst)
125         led <= 8'b1111_1111;
126       case (led_en)
127         8'b0111_1111:
128           led <= tdata[0];
129         8'b1011_1111:
130           led <= tdata[1];
131         8'b1101_1111:
132           led <= tdata[2];
133         8'b1110_1111:
134           led <= tdata[3];
135         8'b1111_0111:
136           led <= tdata[4];
137         8'b1111_1011:
138           led <= tdata[5];
139         8'b1111_1101:
140           led <= tdata[6];
141         8'b1111_1110:
142           led <= tdata[7];
143         default:
144           led <= 8'b1111_1111;
145       endcase
146     end
147
148   endmodule
149
150
```

4. 模块 uart_recv

UART 接收模块

功能

uart_recv 模块负责接收 UART 数据。它通过检测输入信号 din 的变化，识别起始位、数据位和停止位。该模块在接收到完整的数据字节后，输出有效信号 valid 和接收的数据 data。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - din: UART 数据输入信号
- **输出端口**:
  - valid: 数据有效信号，指示接收到的数据有效
  - data[7:0]: 接收到的数据字节

主要实现逻辑

1. **状态机**: 模块采用状态机实现，主要有四个状态：IDLE、START、DATA 和 STOP。根据输入信号和时钟计数器的值进行状态转换。
2. **时钟计数**: 使用 clk_cnt 变量来计数时钟周期，以确保在正确的时钟边缘读取数据位。
3. **数据接收**: 在 DATA 状态下，接收每个数据位，并在 STOP 状态下发出有效信号。

关键代码

```
22  `define bit_clk_cnt 10416
23  `define mid_bit_clk_cnt 5208
24
25  module uart_recv(
26      input clk,
27      input rst,
28      input din,
29      output reg valid,
30      output reg[7:0] data
31  );
32
33  localparam IDLE  = 2'b00;
34  localparam START = 2'b01;
35  localparam DATA  = 2'b10;
36  localparam STOP  = 2'b11;
37
38  reg [31:0] clk_cnt;
39  reg [3:0] bit_pos;
40  reg [1:0] state;
41
42  always @(posedge clk or posedge rst) begin
43      if(rst) begin
44          clk_cnt <= 0;
45          bit_pos <= 0;
46          state <= 0;
47          valid <= 0;
48          data <= 0;
49      end else begin
50          case (state)
51              IDLE: begin
52                  if (!din) begin
53                      clk_cnt <= 0;
54                      state <= START;
55                  end
56              end
57              START: begin
58                  if (clk_cnt == `bit_clk_cnt) begin
59                      clk_cnt <= 0;
60                      bit_pos <= 0;
61                      state <= DATA;
62                  end else begin
63                      clk_cnt <= clk_cnt + 1;
64                  end
65              end
66              DATA: begin
67                  if (clk_cnt == `mid_bit_clk_cnt) begin
68                      data[bit_pos] <= din;
```

```
69                  end
70                  if (clk_cnt == `bit_clk_cnt) begin
71                      clk_cnt <= 0;
72                      if (bit_pos == 7) begin
73                          state <= STOP;
74                      end else begin
75                          bit_pos <= bit_pos + 1;
76                      end
77                  end else begin
78                      clk_cnt <= clk_cnt + 1;
79                  end
80              end
81              default: begin
82                  if (clk_cnt == `mid_bit_clk_cnt) begin
83                      valid <= 1;
84                  end else if (clk_cnt == `mid_bit_clk_cnt + 1) begin
85                      valid <= 0;
86                  end
87                  if (clk_cnt == `bit_clk_cnt) begin
88                      clk_cnt <= 0;
89                      state <= IDLE;
90                  end else begin
91                      clk_cnt <= clk_cnt + 1;
92                  end
93              end
94          endcase
95      end
96  end
97
98  endmodule
99
```

5. 模块 sw_cnt

功能

sw_cnt 模块的主要功能是读取 8 个开关的状态，并根据这些状态生成一个 8 位的输出信号。每个输出位对应一个开关的状态，若开关被按下（为 1），则输出相应位为 1；否则输出为 0。

输入输出端口

- **输入端口**:
    - clk: 时钟信号

- rst: 复位信号
- sw[7:0]: 8 个开关的状态输入

- **输出端口**:
  - out[7:0]: 开关状态的输出信号

```
23  module sw_cnt(
24      input wire clk,
25      input wire rst,
26      input wire[7:0] sw,
27      output reg[7:0] out
28  );
29
30      integer i;
31
32      always @(posedge clk or posedge rst)
33      begin
34          if(rst)
35              out = 8'b0000_0000;
36          else
37          begin
38              out = 8'b0000_0000;
39              for(i = 0;i < 8;i = i + 1)
40                  if(sw[i])
41                      out[i] = 1;
42                  else
43                      out[i] = 0;
44          end
45      end
46
47  endmodule
48
```

6. 模块 sw_display

功能

sw_display 模块的主要功能是读取开关状态，并通过 UART 发送这些状态。在此过程中，模块还实现了对按键 S3 的去抖动处理，以确保可

靠的数据采集。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - S3: 按键输入信号
  - sw[7:0]: 8 个开关的状态输入
- **输出端口**:
  - dout: UART 发送的数据输出

主要实现逻辑

1. **按键去抖动**: 使用定时器 timer_div_10ms 来去抖动按键 S3，避免因开关抖动导致的误触发。定时器计数到 10 毫秒后，更新 limit_div_10ms 使其有效。
2. **状态更新**: 在 limit_div_10ms 触发时，如果 S3 按下，则将当前开关状态 sw_in 赋值，并设置 valid 为 1，指示数据有效。
3. **开关状态读取**: 使用 sw_cnt 模块读取开关状态，将结果传递给 UART 发送模块 uart_send。

关键代码

```
23  module sw_display(
24      input clk,
25      input rst,
26      input S3,
27      input [7:0] sw,
28      output wire dout
29    );
30
31    reg [7:0] data;
32    reg valid;
33    wire [7:0]sw_out;
34    reg [7:0] sw_in;
35
36    reg [31:0] timer_div_10ms;
37    reg [1:0] limit_div_10ms = 0;
38
39    reg S3_before;
40    wire button_posedge;
41
42    always @(posedge clk or posedge rst or posedge button_posedge)
43    begin //10ms 按键去抖动
44      if (rst)
45      begin
46        S3_before <= 0;
47        timer_div_10ms <= 0;
48        limit_div_10ms <= 0;
49      end
50      else
51      begin
52        S3_before <= S3;
53        if (button_posedge)
54        begin
55          timer_div_10ms <= 0;
56          limit_div_10ms <= 0;
57        end
58        else
59        begin
60          timer_div_10ms <= timer_div_10ms + 1;
61          if (timer_div_10ms == 10000000)
62            //if (timer_div_10ms == 10000000)
63          begin
64            limit_div_10ms <= 1;
```

```
65          end
66        end
67      end
68    end
69    assign button_posedge = (S3 & ~S3_before);
70
71    always@ (posedge limit_div_10ms or posedge rst)
72    begin
73      if(rst)
74      begin
75        valid <= 0;
76        data <= 0;
77      end
78      else
79      begin
80        if (S3)
81        begin
82          valid <= 1;
83          data <= sw_in;
84        end
85        else
86        begin
87          valid <= 0;
88        end
89      end
90    end
91
92
93    always@(*)
94    begin
95      sw_in = sw_out;
96    end
97
98
99    sw_cnt u_sw_cnt(
100             .clk(clk),
101             .rst(rst),
102             .out(sw_out),
103             .sw(sw)
104           );
105
```

```
106     uart_send u_uart_send(
107                 .clk(clk),
108                 .rst(rst),
109                 .valid(valid),
110                 .data(data),
111                 .dout(dout)
112               );
113   endmodule
114
```

7. 模块 uart_send

功能

uart_send 模块负责将数据通过 UART 协议发送。它实现了从 IDLE 状态到 START、DATA 和 STOP 状态的状态机，并根据设定的波特率控制发送的时序。

输入输出端口

- **输入端口**:
  - clk: 时钟信号
  - rst: 复位信号
  - valid: 数据有效信号，指示何时发送数据
  - data[7:0]: 要发送的 8 位数据
- **输出端口**:
  - dout: UART 数据输出信号

主要实现逻辑

1. **状态机**: 模块使用一个状态机来管理数据发送的各个阶段，包括 IDLE、START、DATA 和 STOP。根据 valid 信号和波特率计数器的状态进行状态转移。

2. **波特率控制**: 通过 baud_counter 控制发送速率。BAUD_TICKS 根据系统时钟频率和设定的波特率计算得到。
3. **数据发送**: 在 DATA 状态下，从 data_to_send 中逐位发送数据。每次发送一位后，更新 bit_count，直到发送完 8 位数据后进入 STOP 状态。
4. **输出控制**: 控制 dout 信号的状态，确保在不同的状态下输出正确的 UART 信号。

关键代码

```
23  module uart_send(
24      input        clk,
25      input        rst,
26      input        valid,
27      input [7:0]  data,
28      output reg   dout
29    );
30
31
32    localparam IDLE  = 2'b00;
33    localparam START = 2'b01;
34    localparam DATA  = 2'b10;
35    localparam STOP  = 2'b11;
36
37    localparam BAUD_RATE = 9600;
38    localparam SYS_CLK_FREQ = 100_000_000;
39    localparam BAUD_TICKS = SYS_CLK_FREQ / BAUD_RATE;
40
41    reg [1:0] state, next_state;
42    reg [3:0] bit_count;
43    reg [31:0] baud_counter;
44    reg [7:0] data_to_send;
45
46
47    always @(*)
48    begin
49      case (state)
50        IDLE:
51        begin
52          if (valid)
53          begin
54            next_state = START;
55          end
56          else
57          begin
58            next_state = IDLE;
59          end
60        end
61        START:
62        begin
63          if(baud_counter== 0)
64          begin
65            next_state = DATA;
66          end
67          else
68          begin
```

```
69              next_state = START;
70            end
71          end
72          DATA:
73          begin
74            if (bit_count == 8)
75            begin
76              next_state = STOP;
77            end
78            else
79            begin
80              next_state = DATA;
81            end
82          end
83          STOP:
84          begin
85            if(baud_counter==0)
86            begin
87              next_state = IDLE;
88            end
89            else
90            begin
91              next_state = STOP;
92            end
93          end
94          default:
95          begin
96            if(valid)
97            begin
98              next_state = START;
99            end
100         end
101       endcase
102     end
103
104
```

```
105   always @(posedge clk or posedge rst)
106   begin
107     if (rst)
108     begin
109       state <= IDLE;
110       bit_count <= 4'd0;
111       baud_counter <= 32'd0;
112       data_to_send <= 8'd0;
113     end
114     else
115     begin
116       if (baud_counter == 0)
117       begin
118         if (state == DATA)
119         begin
120           bit_count <= bit_count + 1;
121           data_to_send <= {1'b0,data_to_send[7:1]};
122           baud_counter <= BAUD_TICKS - 1;
123         end
124         else if (state == START)
125         begin
126           baud_counter <= BAUD_TICKS - 1;
127           bit_count <= 4'd0;
128         end
129         else if(state==IDLE)
130         begin
131           if(valid)
132           begin
133             data_to_send <= data;
134             baud_counter = BAUD_TICKS - 1;
135           end
136         end
137       end
138       else
139       begin
140         baud_counter <= baud_counter - 1;
141       end
142       state <= next_state;
143     end
144   end
145
```

```
147     always @(posedge clk or posedge rst)
148     begin
149       if (rst)
150       begin
151         dout <= 1'b1;
152       end
153       else
154       begin
155         case (state)
156           IDLE:
157           begin
158             dout <= 1'b1;
159           end
160           START:
161           begin
162             dout <= 1'b0;
163           end
164           DATA:
165           begin
166             dout <= data_to_send[0];
167           end
168           STOP:
169           begin
170             dout <= 1'b1;
171           end
172         endcase
173       end
174     end
175
176   endmodule
177
```

UART 接收模块：

1. **IDLE**: 等待接收数据的状态，监测输入信号 din 是否为低电平（表示开始位）。
2. **START**: 检测到开始位后，进入此状态，并开始计时，准备接收数据位。
3. **DATA**: 逐位接收数据，直到接收完整的 8 位数据。
4. **STOP**: 接收完数据后，等待停止位，并确认接收完成

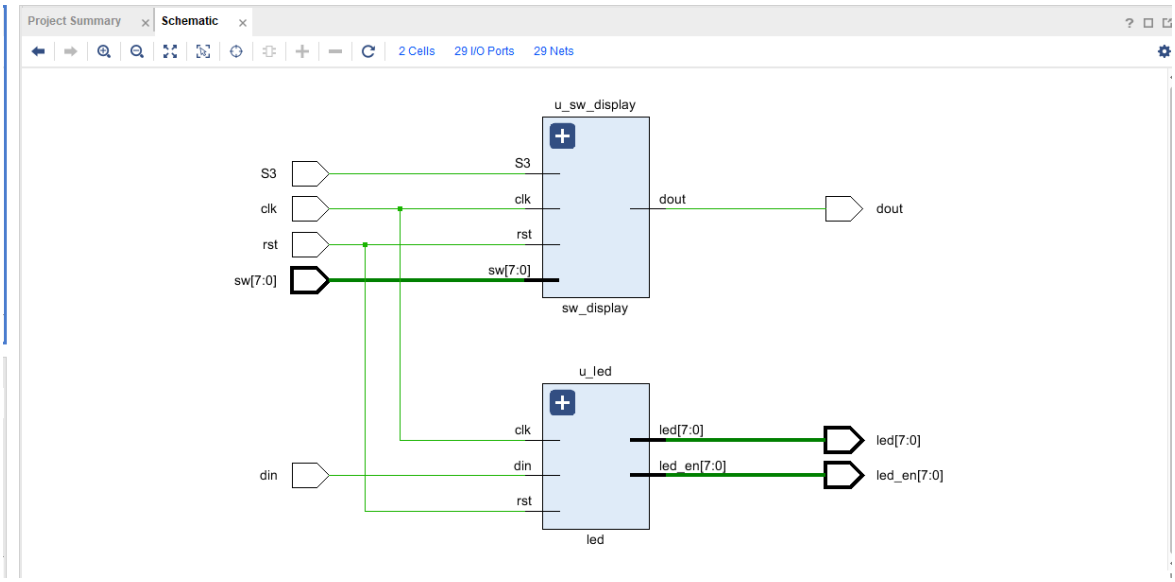
复位逻辑: 当 rst 信号为高时，所有寄存器和状态被重置。

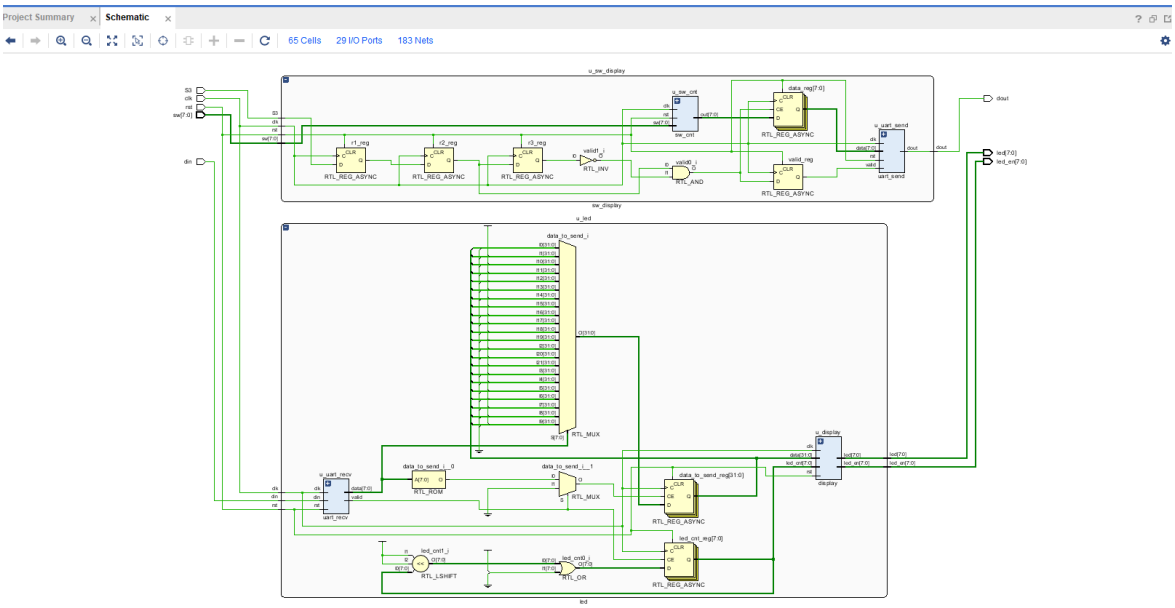
状态转移:

- **1. IDLE 状态**
- **输入**: din = 1（表示线路空闲）
- **现态**: state = IDLE
- **次态**: 若 din 变为 0，则转移到 START 状态。
- **输出**: valid = 0，data = 0
- **2. START 状态**
- **输入**: din = 0（检测到起始位）
- **现态**: state = START
- **次态**: 当 clk_cnt 达到 bit_clk_cnt，转移到 DATA 状态。
- **输出**: 在此状态没有数据输出，bit_pos 计数器重置为 0。
- **3. DATA 状态**

- **输入**: 连续接收数据位
- **现态**: state = DATA
- **次态**:
- 每接收一个数据位，bit_pos 加 1；
- 若 bit_pos 达到 7（接收完 8 位数据），则转移到 STOP 状态。
- **输出**:
- 在 clk_cnt 达到 mid_bit_clk_cnt 时，data[bit_pos] <= din，接收数据位。
- valid 在数据接收完成后会在 STOP 状态中被设置为 1。
- **4. STOP 状态**
- **输入**: 接收到停止位
- **现态**: state = STOP
- **次态**:
- 当 clk_cnt 达到 mid_bit_clk_cnt 后，将 valid 设置为 1；
- clk_cnt 达到 bit_clk_cnt 后，转移回 IDLE 状态。
- **输出**: valid = 1（表示接收到有效数据）。

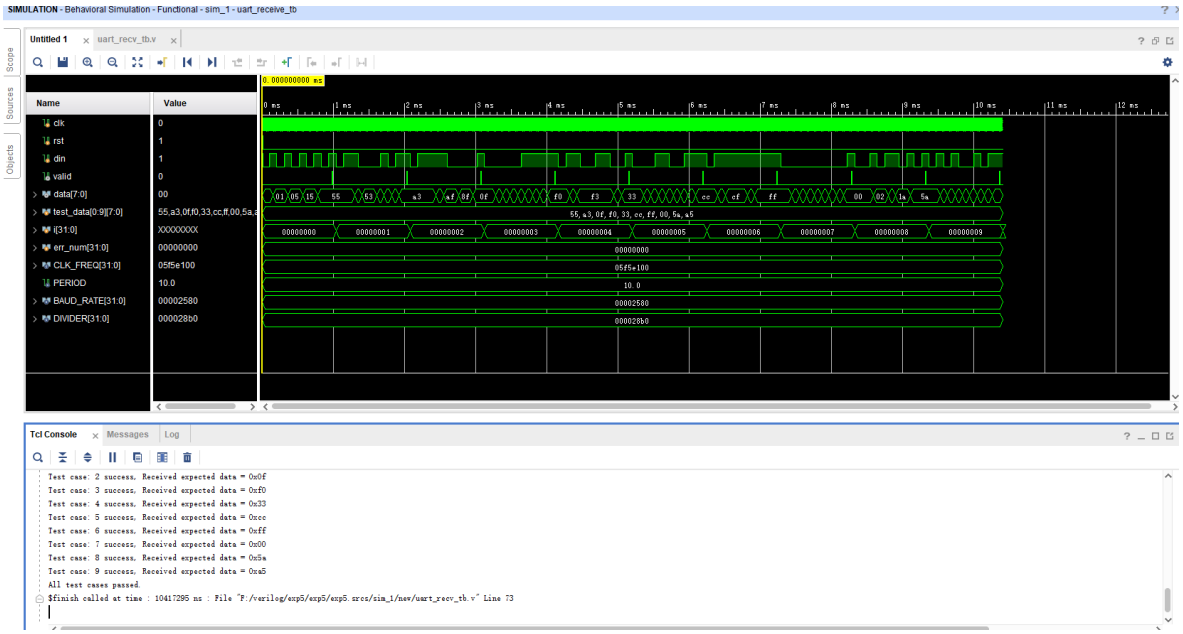
顶层模块的 RTL 分析原理图截图





调试报告

仿真波形截图及仿真分析



设计过程中遇到的问题及解决方法

实验三 数码管高频轮询显示问题分析

现象

在使用数码管进行高频轮询显示时，显示出现闪烁，数码管的显示内容无法稳定，时而显示正确的值，时而出现显示错误

分析过程

1. 发现控制信号频繁变动，且显示的数值并不稳定。
2. 检查了控制数码管的代码显示更新和时间计数的部分。
3. 通过调整时钟频率和轮询逻辑，发现闪烁问题是段选和位选更新逻辑错误导致的

错误原因

led_en 和 led_cx 的更新逻辑在复位后没有及时清除，导致在复位状态下也进行闪烁显示

解决方法

1.优化时间计数逻辑，确保时间计数器的更新与显示更新逻辑一致。

2.设置一个 data 数组存储待显示的数据。

3.修正 led_en 的初始化和更新，每个循环或者是在复位后，led_en 和 led_cx 应立即还原，并在每次时间结束后稳健地轮询

修改后的代码

```
always @ (posedge clk or posedge rst)
begin
  if(rst)
  begin
    p<=0;
    count_1<=1'd0;
    led_en<=8'b11111110;
    data[7] <= 4'd2;
    data[6] <= 4'd3;
  end
  else if(count_1>= 200000)
  begin
    count_1<= 0;
    p<=(p+1)%8;
    led_en<={led_en[6:0],led_en[7]};
  end
  else
    count_1<=count_1+1;
end

//开关打开计数
always@*
begin
  sum = sw[0]+sw[1]+sw[2]+sw[3]+sw[4]+sw[5]+sw[6]+sw[7];
  data[7] <= 4'd2;
  data[6] <= 4'd3;
  data[5]<=sum/10;
  data[4]<=sum%10;
  data[3]<=button_counter1;
  data[2]<=button_counter0;
  data[1]<=dec_counter1;
  data[0]<=dec_counter0;
end
```

```
154     always@*
155     begin
156       case(data[p])
157         4'd0:
158           led_cx<=8'b00000011;
159         4'd1:
160           led_cx<=8'b10011111;
161         4'd2:
162           led_cx<=8'b00100101;
163         4'd3:
164           led_cx<=8'b00001101;
165         4'd4:
166           led_cx<=8'b10011001;
167         4'd5:
168           led_cx<=8'b01001001;
169         4'd6:
170           led_cx<=8'b01000001;
171         4'd7:
172           led_cx<=8'b00011111;
173         4'd8:
174           led_cx<=8'b00000001;
175         4'd9:
176           led_cx<=8'b00011001;
177         default:
178           led_cx<=8'b00000011;
179       endcase
180     end
181
182   endmodule
183
```

课程设计总结

完成本实验所用小时数：约 12 h

其中写代码的小时数：约 9 h

写报告的小时数；约 3 h

课程收获

1. 硬件描述语言 理解 Verilog 语言的基本语法和结构，能够编写模块化的代码

2. 项目设计与实现 通过实际项目（计数器，数码管显示、UART 发送与接收等），提升了代码设计能力

3. 调试能力 学习了如何使用仿真工具对设计进行验证，能够有效地找出并修复代码中的错误，通过实际测试，增强了对硬件行为的理解

课程建议

好

指导书的引导感觉不够，代码没有框架完全是从零开始手搓，课下耗时很多

（但是因为我已经做完了，所以建议给下届的同学再多加内容）