

从电脑端串口软件发送数据

设计的功能描述

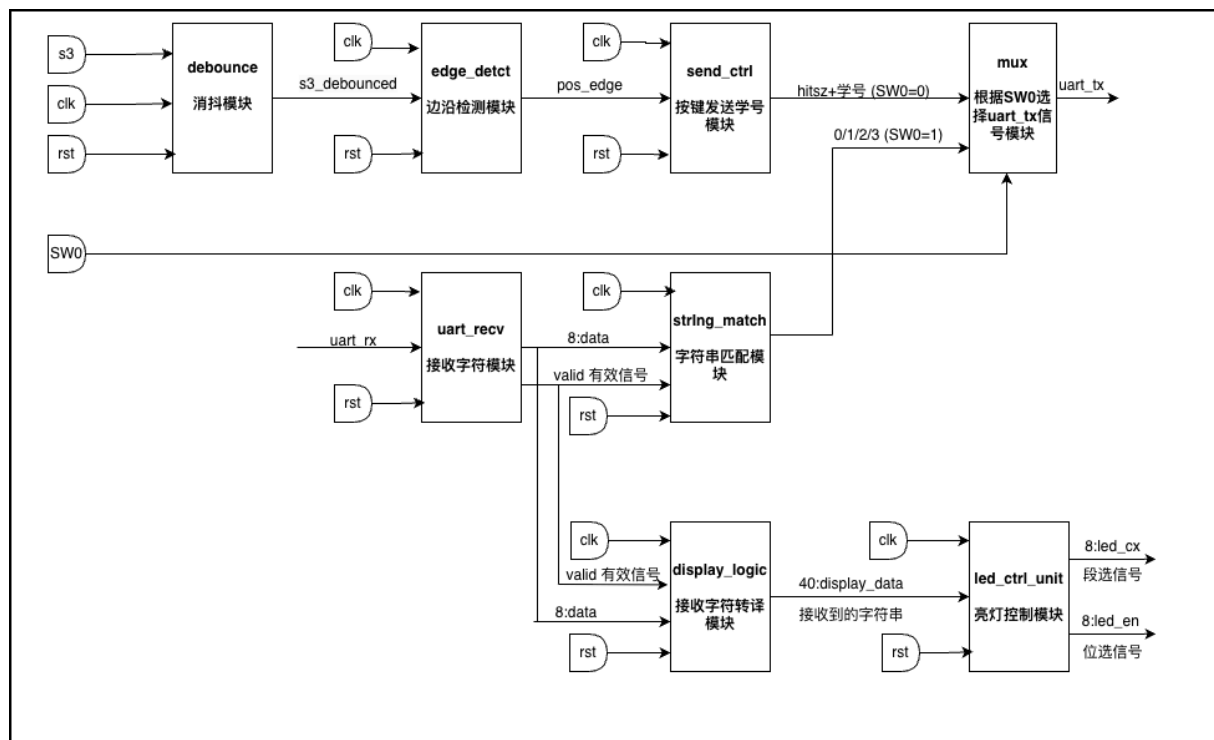
概述基本功能

本实验代码主要完成三部分:

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。
 - 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。
 - 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。
 - 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:
 - 按下一次 S3, 发送一次 hitsz+“个人学号”(hitsz2024311000)。
 - 按键需要消抖, 按一次发送一次。
3. 字符串检测: 从电脑端串口软件发送一串字符串, 检测其中的“start”, “stop”, “hitsz”, 分别回复 1,2,3, 如果最后不存在的匹配, 统一回复 0。

系统设计

系统设计硬件框图 (含附加题)



top_module

Figure 1: 系统设计硬件框图(含附加题)

模块设计与实现

模块功能设计

代码主要拆分成一个 top.v 顶层文件和 8 个子模块，分别是 uart_recv.v, display_logic.v, led_ctrl_unit.v, send_ctrl.v, uart_send.v, edge_detect.v, debounce.v, string_match.v。

uart_recv.v 负责接收电脑端发送的数据， display_logic.v 负责处理接收到的数据并传递给数码管显示模块 led_ctrl_unit.v, send_ctrl.v 负责处理按键 S3 的按下并触发数据发送。

uart_send.v 负责将数据发送到电脑端串口软件， edge_detect.v 负责检测按键 S3 的边沿检测， debounce.v 负责对按键 S3 进行消抖处理，

string_match.v 负责检测接收到的字符串中是否包含“start”、“stop”、“hitsz”，并根据匹配结果发送相应的回复数据给 uart_send.v。

• uart_recv.v: UART 接收模块

- 模块功能：主要负责接受电脑端发送的数据，并将接收到的数据传递给显示逻辑模块和字符串匹配模块。
- 关键逻辑：实现可 FSM 状态机的 UART 接收功能，实现 IDLE START RECV STOP 四种状态的转换，用 din 来作为接收的数据位，din 为 0 的时候说明接收到起始位，状态 IDLE 转换到 START 状态，然后在 RECV 状态下每隔 10416 个时钟周期接收一个数据位，其中在每次计时的中间加入 cyc_half 的检验逻辑，防止毛刺的产生。最后在 STOP 状态下收到停止位，valid 变为 1，表示数据接收完成，data 存储接收到的八位数据，然后通过 top.v 的例化，传向 string_match 和 display_logic 中。

// 状态转移

```
reg [1:0] current_state, next_state;
always @(posedge clk or posedge rst)
begin
    if (rst)
    begin
        current_state <= IDLE;
    end
    else
    begin
        current_state <= next_state;
    end
end
always @(*)
begin
    case (current_state)
        IDLE:
        begin
            if (din == 1'b0)
                next_state = START; // 检测到起始位
            else
                next_state = IDLE;
        end
        START:
        begin
            if (cyc_half) // 采样点
            begin
                if (din == 1'b0)
                    next_state = RECV; // 起始位
```

```

        else
            next_state = IDLE; // 毛刺
        end
    else
        begin
            next_state = START;
        end
    end
end
RECV:
begin
    if (cyc_flag)
        begin
            if (bit_flag)
                next_state = STOP; // 8 位数据接收完毕
            else
                next_state = RECV; // 继续接收下一位
            end
        end
    else
        begin
            next_state = RECV;
        end
    end
end
STOP:
begin
    if (cyc_flag)
        next_state = IDLE; // 停止位接收完毕
    else
        next_state = STOP;
    end
end
default:
begin
    next_state = IDLE;
end
endcase
end

```

• display_logic.v: 显示逻辑模块

- 模块功能: 负责将接收到的数据进行处理, 统计收到字符个数, 排除无效字符, 然后传送给数码管显示模块。
- 关键逻辑: 首先发送数据的十六位代码 (recv_data) 转换为对应的字符 (hex_out), 然后将有效字符存储在一个移位寄存器 (display_slots_chars) 中, 确保只保留最近接收到的 6 个字符。同时, 统计有效字符 (recv_count) 的总数, 并将其低两位传递给数码管显示模块。

```

// 移位寄存器
always @(posedge clk or posedge rst)
begin
    if (rst)
        begin
            display_slots_chars[0] <= empty_char;
            display_slots_chars[1] <= empty_char;
            display_slots_chars[2] <= empty_char;
            display_slots_chars[3] <= empty_char;
            display_slots_chars[4] <= empty_char;
            display_slots_chars[5] <= empty_char;
        end
    end
end

```

```

else if (valid && valid_char)
begin
    display_slots_chars[5] <= display_slots_chars[4];
    display_slots_chars[4] <= display_slots_chars[3];
    display_slots_chars[3] <= display_slots_chars[2];
    display_slots_chars[2] <= display_slots_chars[1];
    display_slots_chars[1] <= display_slots_chars[0];
    display_slots_chars[0] <= {1'b0, hex_out};
end
end
// 计数器
assign count_tens = (recv_count / 10) % 10; // 十位
assign count_ones = recv_count % 10;      // 个位
// display_data 拼接, 然后送给 led_ctrl_unit处理
assign display_data = {
    display_slots_chars[5],
    display_slots_chars[4],
    display_slots_chars[3],
    display_slots_chars[2],
    display_slots_chars[1],
    display_slots_chars[0],
    {1'b0, count_tens},
    {1'b0, count_ones}
};

```

• led_ctrl_unit.v: LED 控制模块

- 模块功能: 负责控制数码管的显示, 包括显示接收到的字符和累计字符个数。
- 关键逻辑: 首先控制数码管显示的位选信号 (led_en), 同时控制刷新速度 (refresh_cnt), 然后根据 display_logic.v 模块传递的数据 (display), 控制段选信号 (led_cx), 使得数码管显示对应的字符。

```

always @(*)
begin
    case (anode_select)
    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;
    default:
        led_en = 8'b11111111; // 全灭
    endcase
end

```

```

reg [4:0] datadigit;
always @(*)
begin
    case (anode_select)
        3'd0:
            datadigit = display[ 4: 0];
        3'd1:
            datadigit = display[ 9: 5];
        3'd2:
            datadigit = display[14:10];
        3'd3:
            datadigit = display[19:15];
        3'd4:
            datadigit = display[24:20];
        3'd5:
            datadigit = display[29:25];
        3'd6:
            datadigit = display[34:30];
        3'd7:
            datadigit = display[39:35];
        default:
            datadigit = empty_char;
    endcase
end
always @(*)
begin
    case (datadigit)
        5'h00:
            led_cx = 8'h03; //8'b1100_0011
        5'h01:
            led_cx = 8'h9F; //8'b1001_1111
        5'h02:
            led_cx = 8'h25; //8'b0010_0101
        5'h03:
            led_cx = 8'h0D; //8'b0000_1101
        5'h04:
            led_cx = 8'h99; //8'b1001_1001
        5'h05:
            led_cx = 8'h49; //8'b0100_1001
        5'h06:
            led_cx = 8'h41; //8'b0100_0001
        5'h07:
            led_cx = 8'h1F; //8'b0001_1111
        5'h08:
            led_cx = 8'h01; //8'b0000_0001
        5'h09:
            led_cx = 8'h09; //8'b0000_1001
        5'h0A:
            led_cx = 8'h11; // 8'b0001_0001;
        5'h0B:
            led_cx = 8'hC1; // 8'b0100_0000;
        5'h0C:
            led_cx = 8'h63; // 8'b1100_0110;
        5'h0D:
            led_cx = 8'h85; // 8'b0010_0000;
        5'h0E:

```

```

        led_cx = 8'h61; // 8'b0110_0001;
5'h0F:
        led_cx = 8'h71; // 8'b0111_0001;
default:
        led_cx = 8'hFF; // 全灭
    endcase
end

```

- send_ctrl.v: 发送控制模块

- 模块功能: 负责处理按键 S3 的按下的时候, 实现 idle send_char wait_char 三个状态的转换, uart_valid 作为握手触发脉冲传送给 uart_send.v, char_wait_cnt 控制字符间发送间隔, pointer 控制要发送的数据位 string_rom。

```

// 状态转移
always @(posedge clk or posedge rst)
begin
    if (rst)
        current_state <= idle;
    else
        current_state <= next_state;
    end
always @(*)
begin
    case (current_state)
    idle:
        begin
            if (s3)
                next_state = send_char;
            else
                next_state = idle;
            end
    send_char:
        next_state = wait_char;
    wait_char:
        begin
            if (char_wait_cnt == char_wait_max - 1)
            begin
                if (pointer == 14)
                    next_state = idle;
                else
                    next_state = send_char;
                end
            else
                next_state = wait_char; // 保持在等待状态
            end
        default:
            next_state = idle;
        endcase
    end
// uart_valid
always @(posedge clk or posedge rst)
begin
    if (rst)
        uart_valid <= 1'b0;
    else
        uart_valid <= (current_state == send_char);
    end

```

```

end
// 字符等待计数
always @(posedge clk or posedge rst)
begin
    if (rst)
        char_wait_cnt <= 19'd0;
    else if (current_state == send_char)
        char_wait_cnt <= 19'd0;
    else if (current_state == wait_char)
        begin
            if (char_wait_cnt == char_wait_max - 1)
                char_wait_cnt <= 19'd0;
            else
                char_wait_cnt <= char_wait_cnt + 1'b1;
        end
    end
end

// 发送字符计数
always @(posedge clk or posedge rst)
begin
    if (rst)
        pointer <= 4'd0;
    else if (current_state == wait_char && char_wait_cnt == char_wait_max - 1)
        begin
            if (pointer == 14)
                pointer <= 4'd0;
            else
                pointer <= pointer + 1'b1;
        end
    end
end
end

```

- uart_send.v: UART 发送模块

- 模块功能: 负责将数据发送到电脑端串口软件。
- 关键逻辑: 实现 UART 的发送功能, 代码在四个状态 IDLE START DATA STOP 之间转移, 在 9600 波特率下, 每隔 10416 个时钟周期发送一个数据位, 同时用 bit_cnt 计算发送的位数, dout 存储要发送的数据位。

```

// 状态转换
always @(posedge clk or posedge rst)
begin
    if (rst)
        begin
            current_state <= IDLE;
        end
    else
        begin
            current_state <= next_state;
        end
    end
end
always @(*)
begin
    case (current_state)
        IDLE:
            begin
                if (valid)

```

```

        begin
            next_state = START;
        end
    else
        begin
            next_state = IDLE;
        end
    end
end
START:
begin
    if (baud_tick)
        begin
            next_state = DATA;
        end
    else
        begin
            next_state = START;
        end
    end
end
DATA:
begin
    if (baud_tick)
        begin
            if (bit_cnt == 3'd7)
                begin
                    next_state = STOP;
                end
            else
                begin
                    next_state = DATA;
                end
            end
        end
    else
        begin
            next_state = DATA;
        end
    end
end
STOP:
begin
    if (baud_tick)
        begin
            next_state = IDLE;
        end
    else
        begin
            next_state = STOP;
        end
    end
end
default:
begin
    next_state = IDLE;
end
endcase
end

```


- edge_detect.v: 边缘检测模块

- 模块功能: 负责检测按键 S3 的边沿检测。
- 关键逻辑: 通过二级信号级联检测按键 S3 的上升沿和下降沿。

```
reg sig_r0, sig_r1;
always @(posedge clk or posedge rst)
begin
    if (rst)
        sig_r0 <= 1'b0;
    else
        sig_r0 <= signal;
    end
always @(posedge clk or posedge rst)
begin
    if (rst)
        sig_r1 <= 1'b0;
    else
        sig_r1 <= sig_r0;
    end
assign pos_edge = sig_r0 & !sig_r1;
```

- debounce.v: 消抖模块

- 模块功能: 负责对按键 S3 进行消抖处理, 确保按键按下时不会产生抖动信号。
- 关键逻辑: 使用计数法进行消抖, 只有在按键状态稳定一段时间后, 才认为按键状态发生了变化。

```
reg [23:0] count;
reg state;
always @(posedge clk or posedge rst)
begin
    if (rst) // 复位
    begin
        state <= 0;
        button_out <= 0;
        count <= 0;
    end
    else
    begin
        if (button_in != state)
        begin
            state <= button_in;
            count <= 0;
        end
        else if (count < cnt_max)
        begin
            count <= count + 1;
        end
        else
        begin
            button_out <= state;
        end
    end
end
```

- string_match.v: 字符串匹配模块

- 模块功能: 负责检测接收到的字符串中是否包含“start”、“stop”、“hitsz”, 并根据匹配结果发送相应的回复数据给 uart_send.v。

```
// mealy
always @(*)
begin
    uart_valid = 1'b0;
    string_rom = 8'h00;
    if (valid)
    begin
        case (current_state)
            star_state:
                if (recv_data == t || recv_data == T)
                begin
                    uart_valid = 1'b1;
                    string_rom = 8'h31;
                end
            sto_state:
                if (recv_data == p || recv_data == P)
                begin
                    uart_valid = 1'b1;
                    string_rom = 8'h32;
                end
            hits_state:
                if (recv_data == z || recv_data == Z)
                begin
                    uart_valid = 1'b1;
                    string_rom = 8'h33;
                end
            default;;
        endcase
        if (uart_valid == 1'b0 && (recv_data == CR || recv_data == LF))
        begin
            if (flag == 1'b0)
            begin
                uart_valid = 1'b1;
                string_rom = 8'h30;
            end
        end
    end
end
else
begin
    uart_valid = 1'b0;
    string_rom = 8'h00;
end
end

always @(posedge clk or posedge rst)
begin
    if (rst)
        flag <= 1'b0;
    else if (valid && (recv_data == CR || recv_data == LF))
        flag <= 1'b0;
    else if (uart_valid && string_rom != 8'h30)
```

```

    flag <= 1'b1;
end

```

- mux.v : uart_tx 选择模块

- 模块功能: 负责在 send_ctrl.v 和 string_match.v 之间选择 uart_tx 的信号源。
- 关键逻辑: 通过 SW0 选择是 send_ctrl.v 还是 string_match.v 的输出作为 uart_tx 的输入。

```

module mux (
    input wire SW0,
    input wire in0,
    input wire in1,
    output wire uart_tx
);
    assign uart_tx = (SW0 == 1'b0) ? in0 : in1;
endmodule

```

状态机转移图

- uart_recv.v 状态机转移图

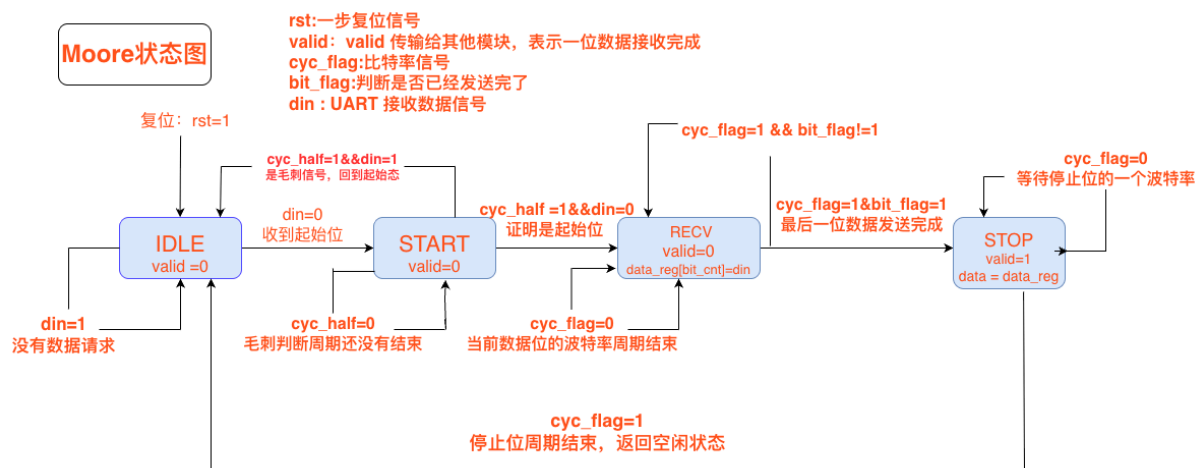


Figure 2: uart_recv.v 状态机转移图

• string_match.v 状态机转移图

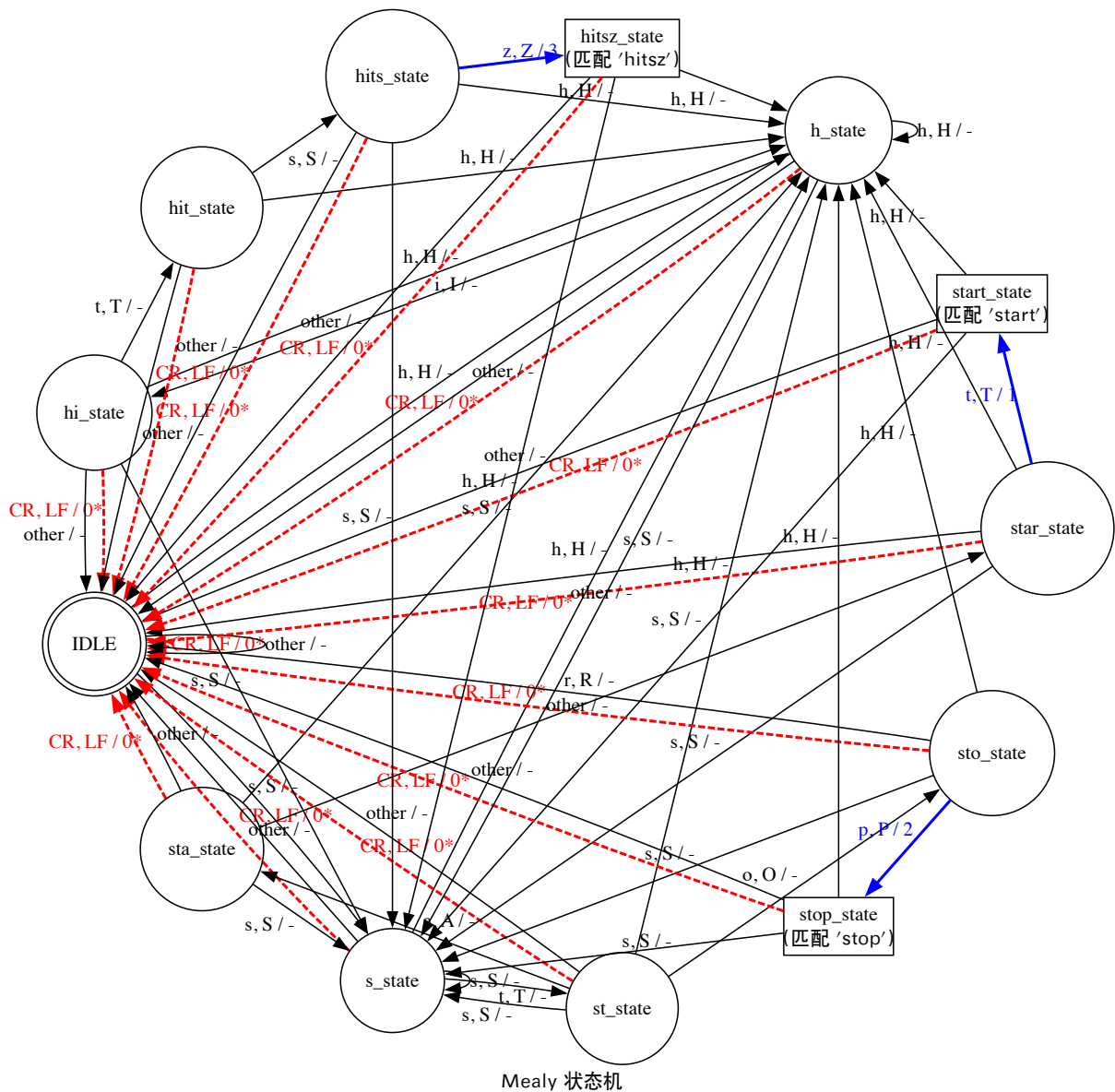


Figure 3: string_match.v 状态机转移图

状态转移图说明:蓝色线表示最终的输出, 比如 star_state&&recv_data-t-> start_state 最终输出 1
 红色线表示收到换行符, 强制回到 IDLE 的情况, 如果前面存在匹配, 什么都不发送, 不存在匹配则发送 0。同时除了某些特殊情况(收到 s, 返回 s_state), 不匹配的字符会使得状态机回到 IDLE 状态。

顶层模块的 RTL 分析图

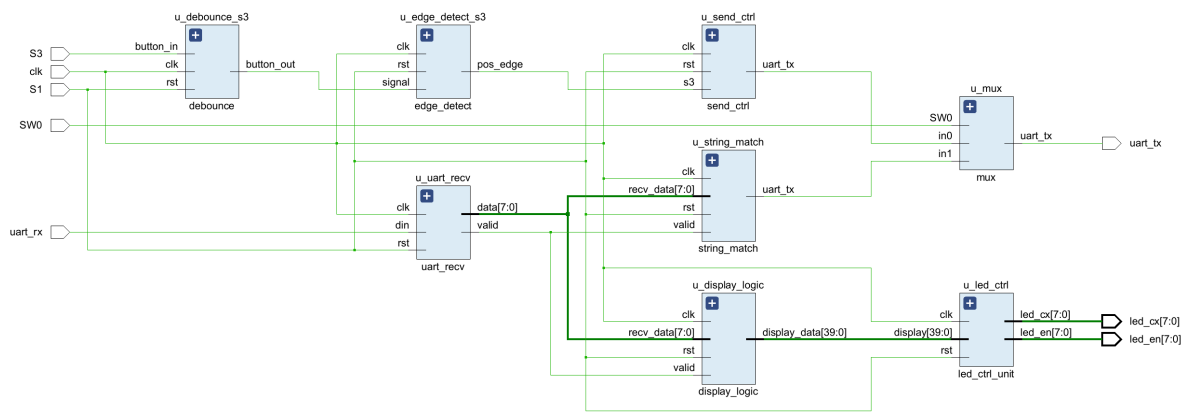


Figure 4: 顶层模块的 RTL 分析图

仿真调试

仿真波形调试

```
Tcl Console x Messages x Log
[Icons] [Z] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z] [0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [Space] [Tab] [Enter] [Backspace] [Delete] [Insert] [Home] [End] [Page Up] [Page Down] [Print] [F1] [F2] [F3] [F4] [F5] [F6] [F7] [F8] [F9] [F10] [F11] [F12] [Esc] [Win] [Alt] [Ctrl] [Shift] [Cmd] [Fn] [Num Lock] [Caps Lock] [F13] [F14] [F15] [F16] [F17] [F18] [F19] [F20] [F21] [F22] [F23] [F24] [F25] [F26] [F27] [F28] [F29] [F30] [F31] [F32] [F33] [F34] [F35] [F36] [F37] [F38] [F39] [F40] [F41] [F42] [F43] [F44] [F45] [F46] [F47] [F48] [F49] [F50] [F51] [F52] [F53] [F54] [F55] [F56] [F57] [F58] [F59] [F60] [F61] [F62] [F63] [F64] [F65] [F66] [F67] [F68] [F69] [F70] [F71] [F72] [F73] [F74] [F75] [F76] [F77] [F78] [F79] [F80] [F81] [F82] [F83] [F84] [F85] [F86] [F87] [F88] [F89] [F90] [F91] [F92] [F93] [F94] [F95] [F96] [F97] [F98] [F99] [F100] [F101] [F102] [F103] [F104] [F105] [F106] [F107] [F108] [F109] [F110] [F111] [F112] [F113] [F114] [F115] [F116] [F117] [F118] [F119] [F120] [F121] [F122] [F123] [F124] [F125] [F126] [F127] [F128] [F129] [F130] [F131] [F132] [F133] [F134] [F135] [F136] [F137] [F138] [F139] [F140] [F141] [F142] [F143] [F144] [F145] [F146] [F147] [F148] [F149] [F150] [F151] [F152] [F153] [F154] [F155] [F156] [F157] [F158] [F159] [F160] [F161] [F162] [F163] [F164] [F165] [F166] [F167] [F168] [F169] [F170] [F171] [F172] [F173] [F174] [F175] [F176] [F177] [F178] [F179] [F180] [F181] [F182] [F183] [F184] [F185] [F186] [F187] [F188] [F189] [F190] [F191] [F192] [F193] [F194] [F195] [F196] [F197] [F198] [F199] [F200] [F201] [F202] [F203] [F204] [F205] [F206] [F207] [F208] [F209] [F210] [F211] [F212] [F213] [F214] [F215] [F216] [F217] [F218] [F219] [F220] [F221] [F222] [F223] [F224] [F225] [F226] [F227] [F228] [F229] [F230] [F231] [F232] [F233] [F234] [F235] [F236] [F237] [F238] [F239] [F240] [F241] [F242] [F243] [F244] [F245] [F246] [F247] [F248] [F249] [F250] [F251] [F252] [F253] [F254] [F255] [F256] [F257] [F258] [F259] [F260] [F261] [F262] [F263] [F264] [F265] [F266] [F267] [F268] [F269] [F270] [F271] [F272] [F273] [F274] [F275] [F276] [F277] [F278] [F279] [F280] [F281] [F282] [F283] [F284] [F285] [F286] [F287] [F288] [F289] [F290] [F291] [F292] [F293] [F294] [F295] [F296] [F297] [F298] [F299] [F300] [F301] [F302] [F303] [F304] [F305] [F306] [F307] [F308] [F309] [F310] [F311] [F312] [F313] [F314] [F315] [F316] [F317] [F318] [F319] [F320] [F321] [F322] [F323] [F324] [F325] [F326] [F327] [F328] [F329] [F330] [F331] [F332] [F333] [F334] [F335] [F336] [F337] [F338] [F339] [F340] [F341] [F342] [F343] [F344] [F345] [F346] [F347] [F348] [F349] [F350] [F351] [F352] [F353] [F354] [F355] [F356] [F357] [F358] [F359] [F360] [F361] [F362] [F363] [F364] [F365] [F366] [F367] [F368] [F369] [F370] [F371] [F372] [F373] [F374] [F375] [F376] [F377] [F378] [F379] [F380] [F381] [F382] [F383] [F384] [F385] [F386] [F387] [F388] [F389] [F390] [F391] [F392] [F393] [F394] [F395] [F396] [F397] [F398] [F399] [F400] [F401] [F402] [F403] [F404] [F405] [F406] [F407] [F408] [F409] [F410] [F411] [F412] [F413] [F414] [F415] [F416] [F417] [F418] [F419] [F420] [F421] [F422] [F423] [F424] [F425] [F426] [F427] [F428] [F429] [F430] [F431] [F432] [F433] [F434] [F435] [F436] [F437] [F438] [F439] [F440] [F441] [F442] [F443] [F444] [F445] [F446] [F447] [F448] [F449] [F450] [F451] [F452] [F453] [F454] [F455] [F456] [F457] [F458] [F459] [F460] [F461] [F462] [F463] [F464] [F465] [F466] [F467] [F468] [F469] [F470] [F471] [F472] [F473] [F474] [F475] [F476] [F477] [F478] [F479] [F480] [F481] [F482] [F483] [F484] [F485] [F486] [F487] [F488] [F489] [F490] [F491] [F492] [F493] [F494] [F495] [F496] [F497] [F498] [F499] [F500] [F501] [F502] [F503] [F504] [F505] [F506] [F507] [F508] [F509] [F510] [F511] [F512] [F513] [F514] [F515] [F516] [F517] [F518] [F519] [F520] [F521] [F522] [F523] [F524] [F525] [F526] [F527] [F528] [F529] [F530] [F531] [F532] [F533] [F534] [F535] [F536] [F537] [F538] [F539] [F540] [F541] [F542] [F543] [F544] [F545] [F546] [F547] [F548] [F549] [F550] [F551] [F552] [F553] [F554] [F555] [F556] [F557] [F558] [F559] [F560] [F561] [F562] [F563] [F564] [F565] [F566] [F567] [F568] [F569] [F570] [F571] [F572] [F573] [F574] [F575] [F576] [F577] [F578] [F579] [F580] [F581] [F582] [F583] [F584] [F585] [F586] [F587] [F588] [F589] [F590] [F591] [F592] [F593] [F594] [F595] [F596] [F597] [F598] [F599] [F600] [F601] [F602] [F603] [F604] [F605] [F606] [F607] [F608] [F609] [F610] [F611] [F612] [F613] [F614] [F615] [F616] [F617] [F618] [F619] [F620] [F621] [F622] [F623] [F624] [F625] [F626] [F627] [F628] [F629] [F630] [F631] [F632] [F633] [F634] [F635] [F636] [F637] [F638] [F639] [F640] [F641] [F642] [F643] [F644] [F645] [F646] [F647] [F648] [F649] [F650] [F651] [F652] [F653] [F654] [F655] [F656] [F657] [F658] [F659] [F660] [F661] [F662] [F663] [F664] [F665
```

Figure 5: tcl 图片

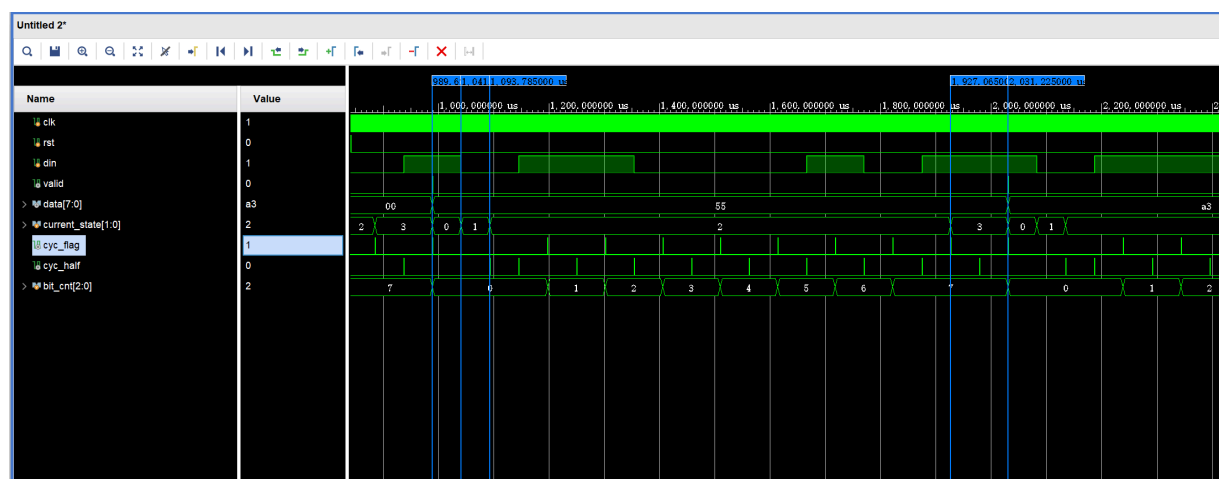


Figure 6: uart_recv.v 仿真波形图 1

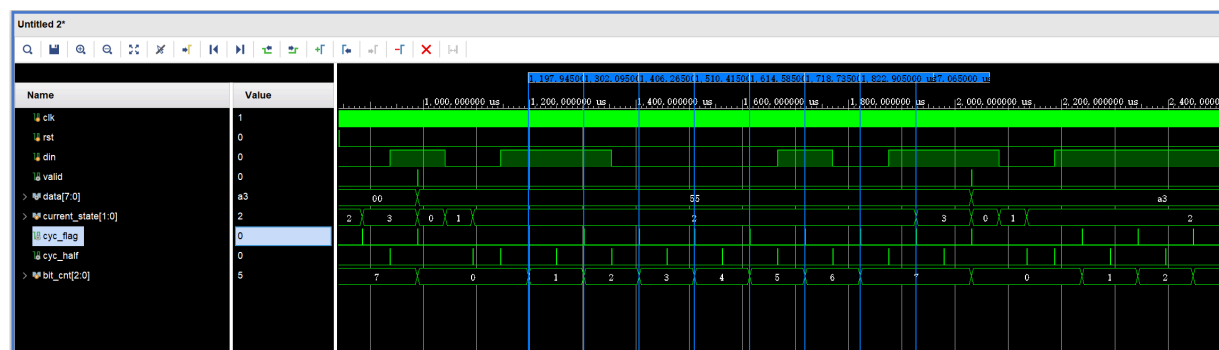


Figure 7: string_match.v 仿真波形图 2

这两张图体现了 `uart_recv.v` 模块的一个完整数据帧的发送过程，首先在 IDLE 状态下等待起始位的到来，第二个 marker, `din` 变为 0，代表起始位的到来，`current_state` 变为 START，然后从 START 状态进入 RECV 状态，开始接收数据，从第二个 marker 到第三个 marker, `current_state`

保持为 RECV,cyc_flag 有 8 次变为 1(cyc_half 用于检测毛刺), 代表收到了 8 位数据, 同时观察第二张图中的 8 个 marker,din 发送的编码正好是 1010_0011,对应十六进制 8'hA3, bit_cnt 也由 0 变到 7, 最后 current_state 变为 STOP, 表示数据接收完成, 在等待 cyc_flag 变为高电平之后, current_state 从 STOP 变为 IDLE, 开始接收下一位数据, 同时 data 也变为 8'hA3。

• send_ctrl.v 模块仿真波形图分析

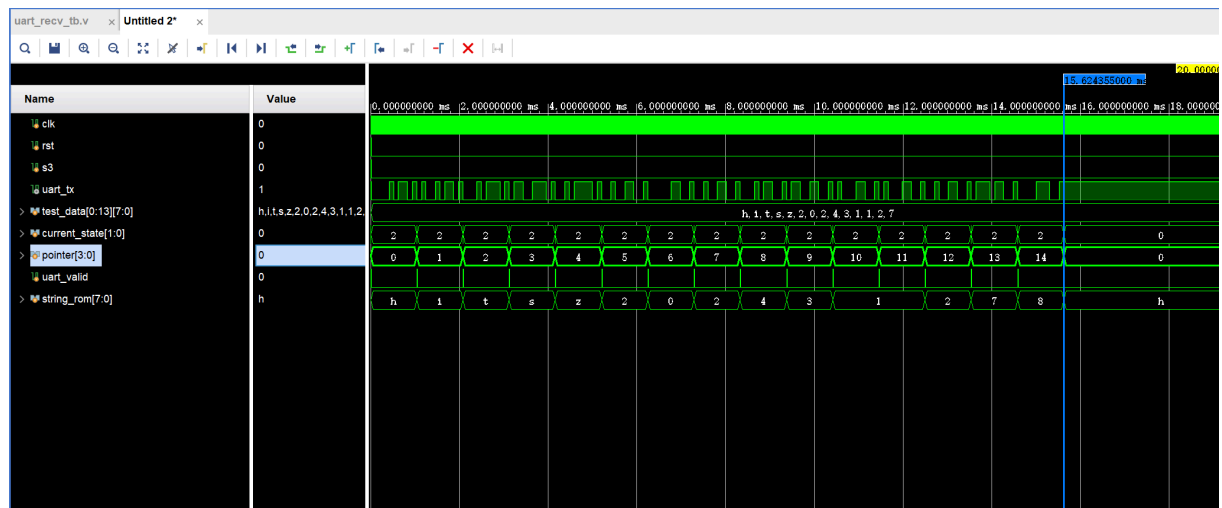


Figure 8: send_ctrl.v 模块仿真波形图

这张图体现了 send_ctrl.v 模块的一个完整发送过程, 首先在 idle 状态下等待按键信号 s3 的到来, 然后 current_state 变为 send_char,send_char 再变为 wait_char, 整体 current_state 0 变 1 再变 2 (由于变化时间周期过短, 图上无法体现), 在进入状态 wait_char 后, current_state 在 send_char 和 wait_char 之间来回切换。在一个 wait_char 周期内, string_rom 跟随 pointer 变化 (例如:pointer=0->string_rom= h), 同时每一个 wait_char 周期对应一个 uart_valid 的高电平脉冲, 表示成功发送一个有效字符。最终, 经过十四个 wait_char 周期后, pointer 变为 14, 表示发送完成。r 然后, current_state 重新变为 idle, pointer 被清零, 等待下一次按键触发。

• string_match.v 模块仿真波形图分析

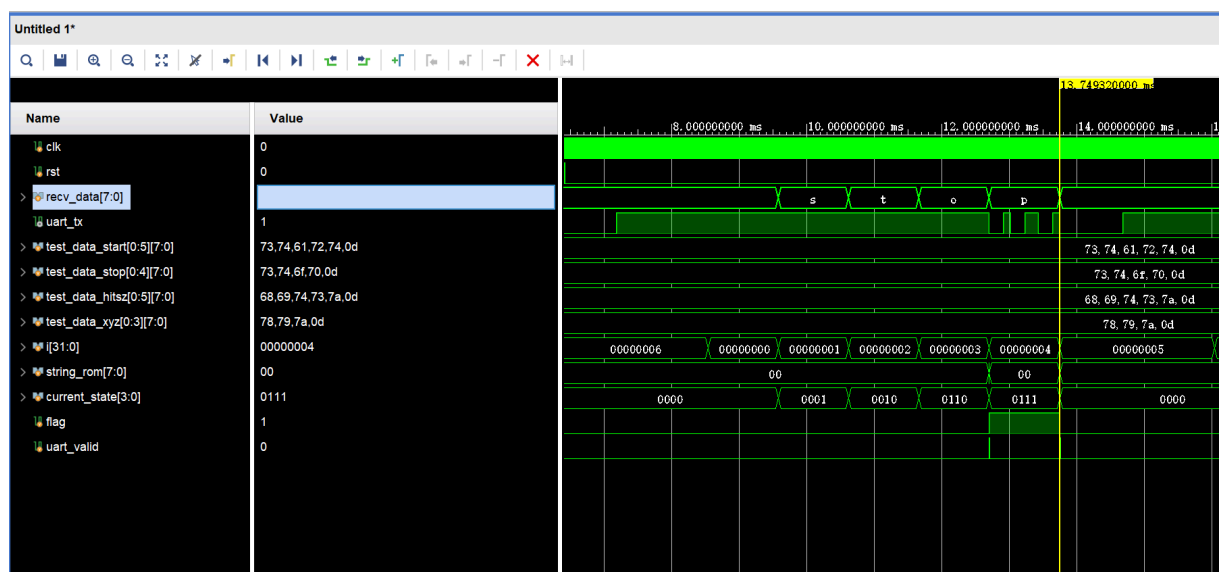


Figure 9: string_match.v 模块仿真波形图 1

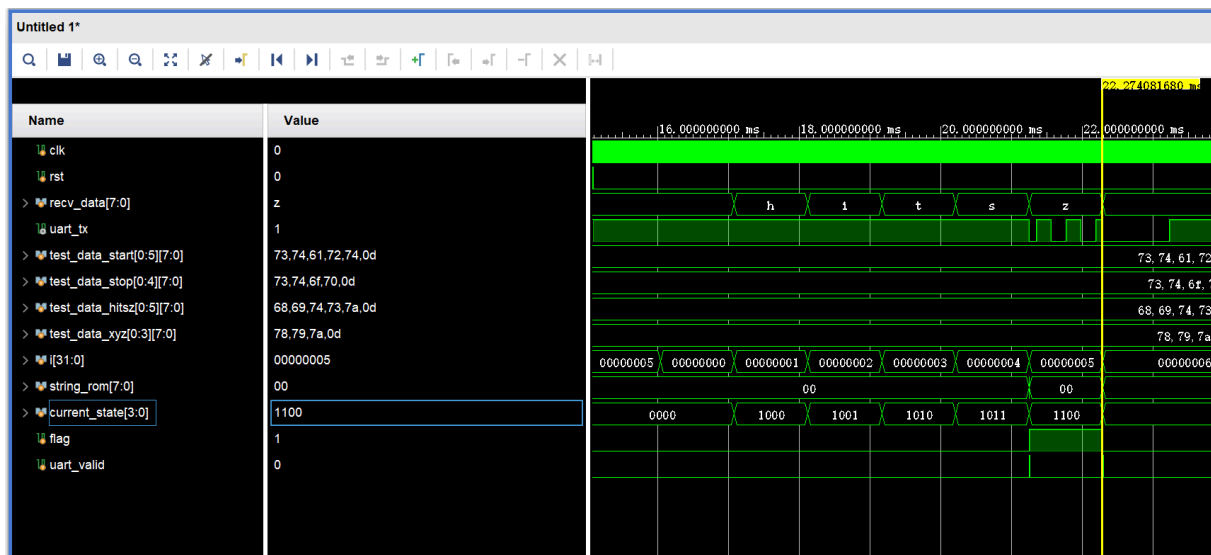


Figure 10: string_match.v 模块仿真波形图 2

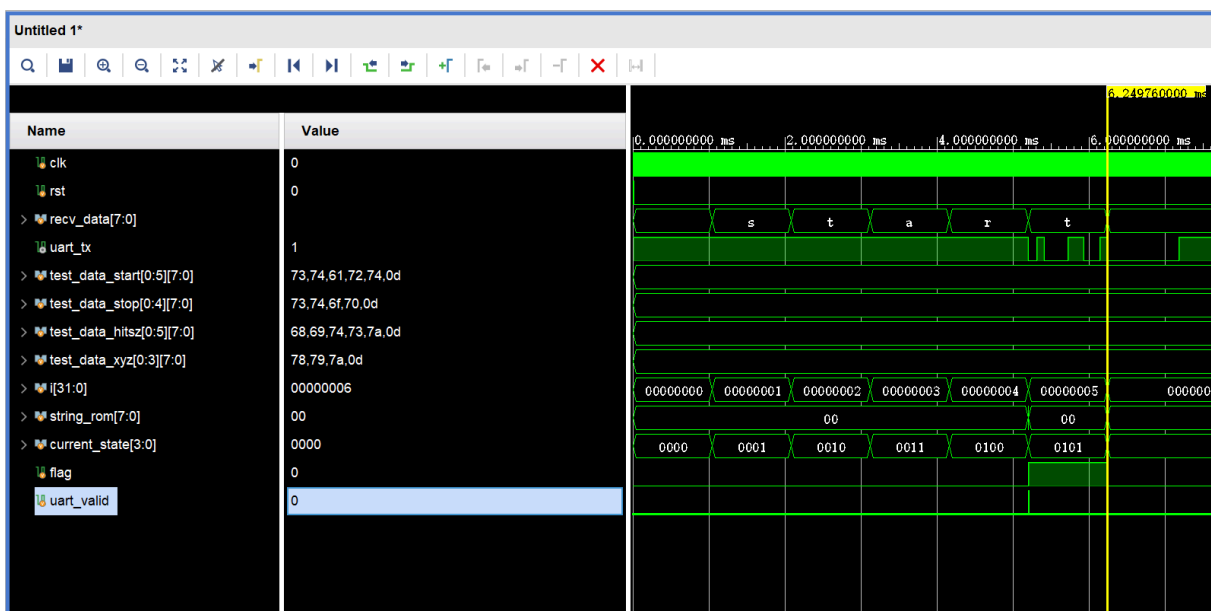


Figure 11: string_match.v 模块仿真波形图 3

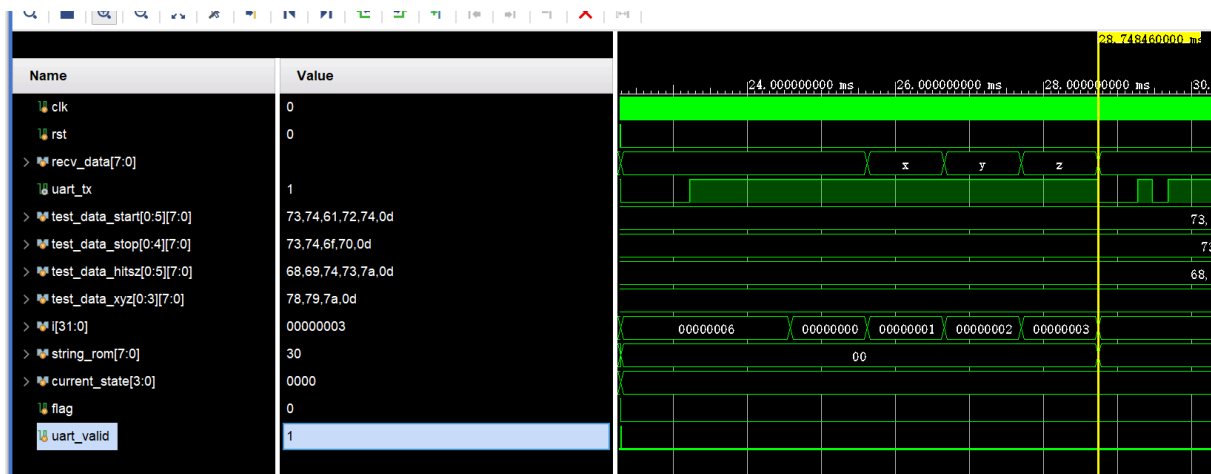


Figure 12: string_match.v 模块仿真波形图 4

前三张波形图分别展现了“stop”、“hitsz”、“start”三个字符串的匹配过程。以 start 字符串发送图 Figure 11 为例，current_state 初始为 IDLE(0000)，当接收到字符 ‘s’ 时，current_state 变为 s_state(0001)，接着接收到 ‘t’ 时，current_state 变为 st_state(0010)，依次类推，知道 current_state == star_state 同时 recv_data == t 的时候(mealy 状态机)，uart_valid 变为高电平脉冲，表示匹配成功，发送 string_rom == 8’h31 (ascii 字符等于 1)，与此同时,flag 变为 1，表示存在匹配，不再发送 0，接收完成之后，current_state 重新变为 IDLE。

最后一张图 Figure 12 展现了字符串未匹配的情况，可以见到 current_state 一直维持在 IDLE 状态，最终 uart_valid 变为高电平脉冲，发送 string_rom == 8’h30 (ascii 字符等于 0)，表示未匹配成功。

设计过程中遇到的问题及解决方法、Lint 分析

linter 截图

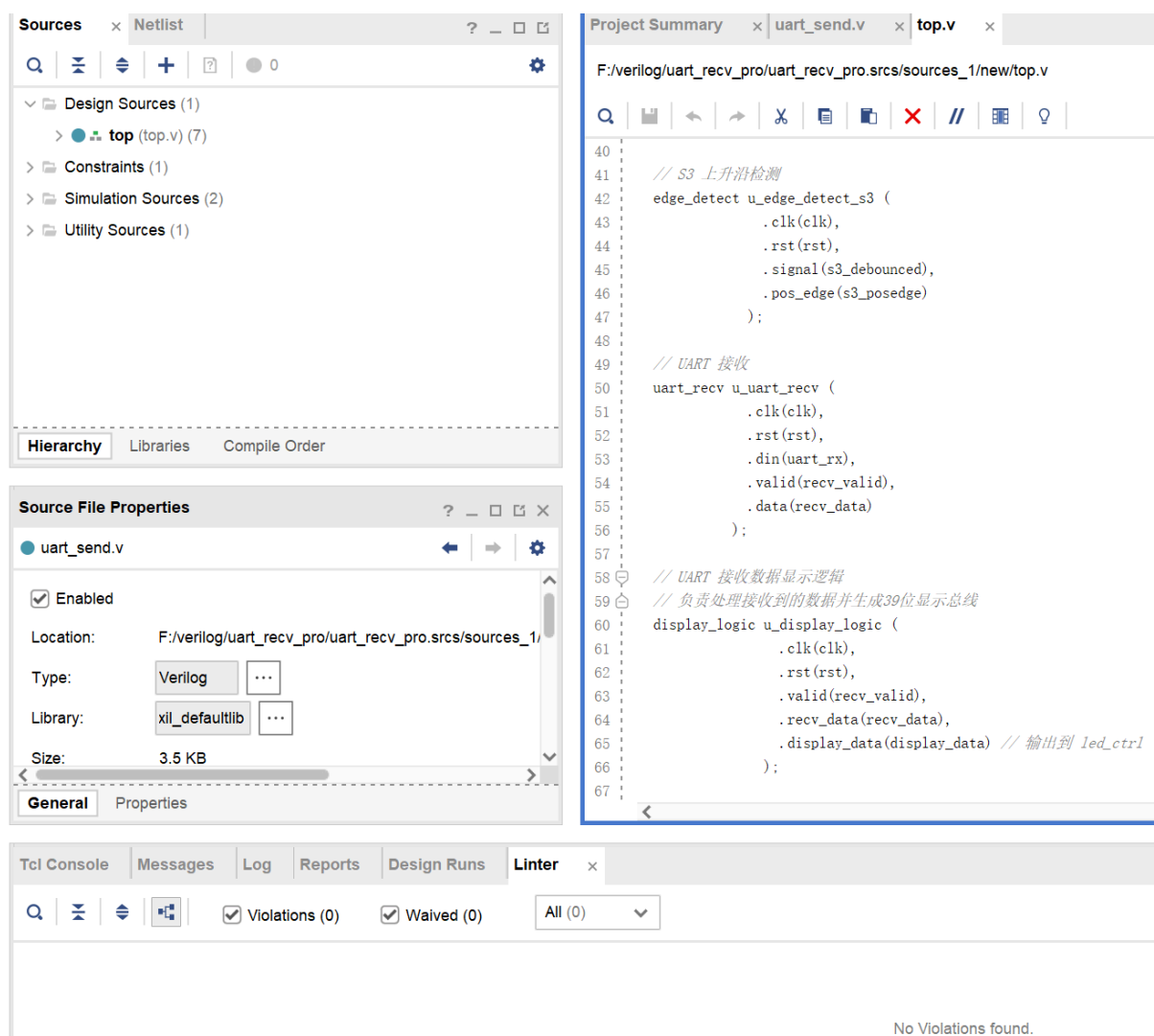


Figure 13: linter 截图

印象深刻的代码逻辑有关问题

让我印象最深刻的代码逻辑是附加题 1 中关于时序逻辑的处理，在设计 string_match.v 模块的时候

- 有问题的代码

```
always @(posedge clk or posedge rst)
begin
    if (rst)
        flag <= 1'b0;
    else if (valid && current_state == end_state)
        flag <= 1'b0;
    else if (uart_valid && string_rom != 8'h30)
        flag <= 1'b1;
    end
always @(posedge clk or posedge rst)
begin
    if (rst)
        uart_valid <= 1'b0;
    else
        uart_valid <= valid &&
            (current_state==start_state ||
             current_state==stop_state ||
             current_state==hitsz_state ||
             (current_state==end_state && !flag));
    end
end
```

- 上板异常与原因

我原来设计的是 moore 型状态机，定义了当收到换行符的时候 current_state 变为 end_state，然后在 end_state 状态下根据 flag 的值决定是否发送 0。例如前面存在匹配就会使得 flag=1 不会触发发送 0。但这段代码逻辑的问题在于：每个字符串在收到换行符的下一个时钟上升沿会自动清零，同时 uart_valid 也在下一个时钟上升沿赋值，导致每个字符串在末尾都会自动触发输出 0。如果修改成 flag <= flag 会导致无法连续匹配的问题

- 排查过程和思路修改后的代码

在收到上板的问题后，我选择重构 uart_valid (uart_send 的发送信号)， string_rom (发送的字符)， flag (是否存在匹配的判断逻辑) 的赋值逻辑，删除了 end_state, 将状态机修改为 mealy 状态机，同时用一个时间差巧妙避开了， flag 清零的问题。

uart_valid 和 string_rom 都是阻塞赋值， flag 是非阻塞赋值，当换行符到达的时候，触发下面代码

```
if (uart_valid == 1'b0 && (recv_data == CR || recv_data == LF))
begin
    if (flag == 1'b0)
begin
    uart_valid = 1'b1;
    string_rom = 8'h30;
    end
end
always @(posedge clk or posedge rst)
begin
    if (rst)
        flag <= 1'b0;
    else if (valid && (recv_data == CR || recv_data == LF))
        flag <= 1'b0;
    else if (uart_valid && string_rom != 8'h30)
```

```
flag <= 1'b1;  
end
```

判断逻辑，用当前 flag 的值去判断是否发送，可以准确判断这个字符串是否存在匹配，同时因为 flag 是非阻塞赋值，在下一个时钟上升沿才会被清零，也不会影响到下一个字符串识别。相当于打了一个时间差，抓住了非阻塞赋值要在下一个时钟上升沿才会赋值的特点，从而正确发出了命令

AI 工具使用总结

1. 实验课程中未使用 HiAgent 的实验助手
2. 其他使用了 Google 的 Gemini 2.5 pro
3. 前期对话次数较多，后期偶尔用用
4. 前面的实验较为简单，通过 AI 的编写代码去学习 verilog 的代码逻辑和基础知识，后期实验难度加大，AI 辅助作用较小，主要是用来检查代码逻辑和语法错误。
5. 总结：在前期不熟悉 verilog 语言的时候，通过前三个实验简单的项目熟悉 verilog 语言的语法和代码逻辑，AI 辅助作用较大，通过阅读 AI 的代码以及提问学习提升了很多，感觉 AI 更像一个可以随时随地提问的好老师，后期实验中，可以自主组织代码逻辑和完成实验，但 AI 依然可以检查很多逻辑问题与语法错误，节省了很多调试时间。

课程设计总结

收获：前面几次试验熟悉了 verilog 的基础语法和代码逻辑，以及 vivado 基本调试方法，同时也复习了很多数字逻辑设计的知识，例如时序逻辑，组合逻辑，状态机等等。而且最后一次实验让我了解到自顶向下的设计思路，用 top.v 作为顶层模块，分割成多个子模块，让我感觉这种设计思路很像函数式编程，顶层模块通过 wire 将需要操作的变量传入子模块，每个子模块相当于函数，可以完成独立的功能，同时顶层模块负责调用各个子模块或者子模块之间相互调用，完成整体功能。