

设计的功能描述

基本功能概述

0. 初始态与复位:

初始时, 所有 LED 数码管均灭, 按键 S2 为复位按键, 一旦按下将回到初始态。

1. UART 数据接收与 LED 数码管显示:

电脑端串口软件发送字符, 接收到“0”~“9”以及“A”~“F”时, 将其 ASCII 值解析并显示到 LED 数码管上, 且数码管只显示最近接收到的 8 个字符, 不足 8 位时高位不显示。接收到新数据时, 新字符显示在数码管最低位, 其他位字符左移一位; 当接收到的字符不在显示范围内时, 数码管其他位字符仍然左移, 但新数据位不显示。

2. UART 数据发送:

(1) 数据自主发送:

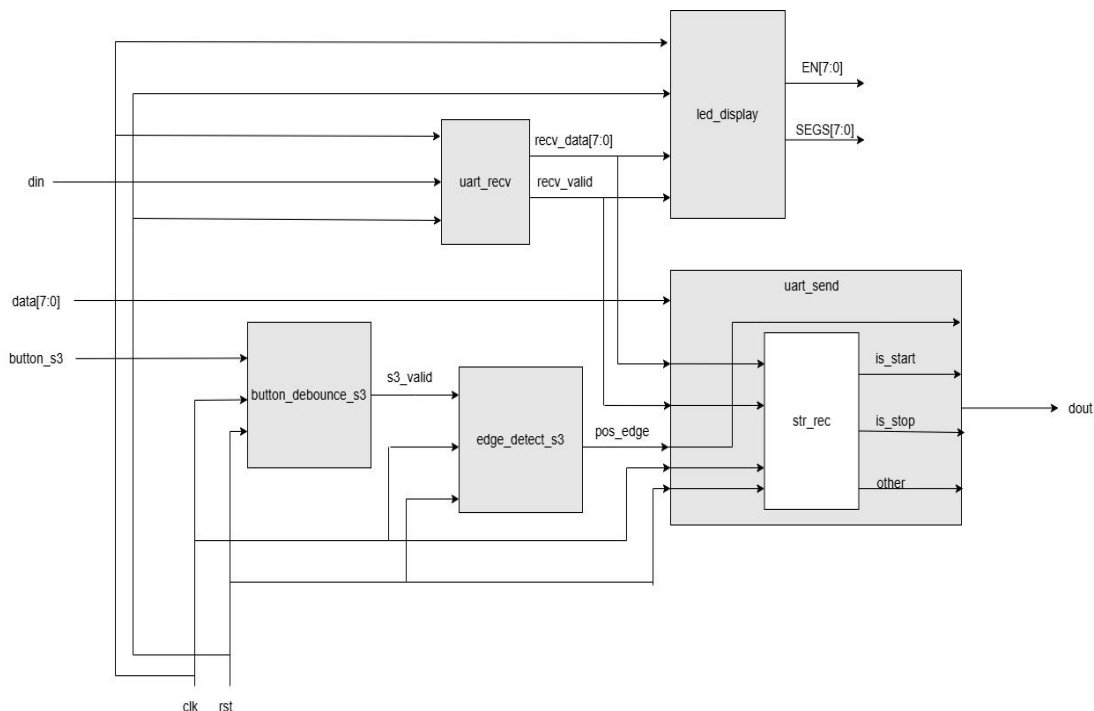
待发送的数据从拨码开关 SW7-SW0 输入, 一旦按键 S3 按下, 拨码开关的数据读取作为 ASCII 值发送到电脑端串口软件。S3 按住不放时数据只发送一次。

(2) 字符串检测:

对 UART 连续接收到的数据作检测, 若检测到字符串“start”(全为小写)时, 将字符“1”作为回复发送、检测到“stop”时发送“2”, 否则发送“0”, 支持连续检测。这里的字符串指包含指定字符串的字符串, 并以回车换行符为终结符。且对于形如“...start...start...”的字符串, 只回复“1”而非“11”。

系统设计

1. 系统整体设计硬件框图：



模块设计与实现

1. 各子模块的描述：

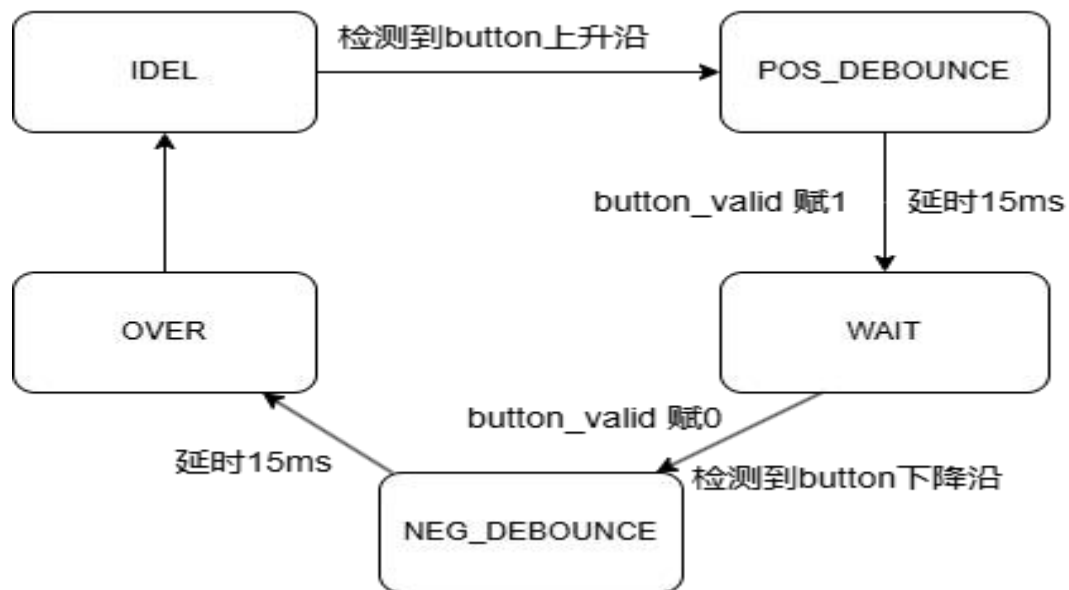
(0) 对计数器不作单独模块的说明：

虽然整个项目多处用到计数器，但计数时的条件各有差异。以 `uart_send` 以及 `uart_rcv` 中的波特率计数器为例，前者中计数器置零的条件为发送使能 `valid` 为 1，而后者中计数器置零的条件为 `current_state == IDLE`（状态机所处状态为初态），这样的差异对于模块化是不友好的。此外，一些小型的计数功能没有必要使用全部计数器代码（如 `uart_send` 中对发送数据位数的计数），故本项目中不考虑将计数器模块化。

(1) button_debounce:

①功能：对按键 S3 进行信号消抖。

②实现逻辑：模块采用状态机对按键信号进行消抖。其状态转换图如下：



如此设计的前提是 button 不会在 POS_DEBOUNCE 时有效信号就结束，现实中确实是不会违背这个前提的。

此外，这里以原始信号的上升沿、下降沿作为条件，是因为真实的抖动信号一般都是长于一个时钟周期的。其实实现见 pos_detect 模块。

③输入输出信号：

input: clk, rst, button, 其中 button 为原始按键信号；

output: button_valid, 即经过消抖后输出的有效标准按键信号。

④关键代码：（消抖代码，篇幅排版原因，next_state 的赋值用状态对应的值代替）

```

always @(posedge clk or negedge rst) begin
    if (rst) current_state <= 0;
    else     current_state <= next_state;
end
  
```

```

end

always @(posedge clk or negedge rst) begin
    if (rst) next_state <= 0;
    else
        case (next_state)
            IDLE:      if (pos_edge)      next_state <= 1;
                       else                next_state <= 0;
            POS_DEBOUNCE: if (button_cnt == DELAY) next_state <= 2;
                       else                next_state <= 1;
            WAIT:      if (neg_edge)      next_state <= 3;
                       else                next_state <= 2;
            NEG_DEBOUNCE: if (button_cnt == DELAY) next_state <= 4;
                       else                next_state <= 3;
            OVER:      next_state <= 0;
            default:    next_state <= 0;
        endcase
    end

always @(posedge clk or negedge rst) begin
    if (rst) button_valid <= 0;
    else
        case (current_state)
            WAIT:      button_valid <= 1;
            NEG_DEBOUNCE: button_valid <= 0;
            default:    button_valid <= button_valid;
        endcase
    end
end

```

(2) edge_detect:

①功能：检测信号的上升沿和下降沿。

②实现逻辑：采用类似打拍的检测方式，但是模块的面对对象是标准信号（无抖动信号）或消抖时的抖动信号，因此没有必要进行多次打拍，只进行一次打拍即可。

③输入输出信号：

input: clk, rst, signal;

output: pos_edge, neg_edge.

④关键代码

```
reg signal_prev;

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

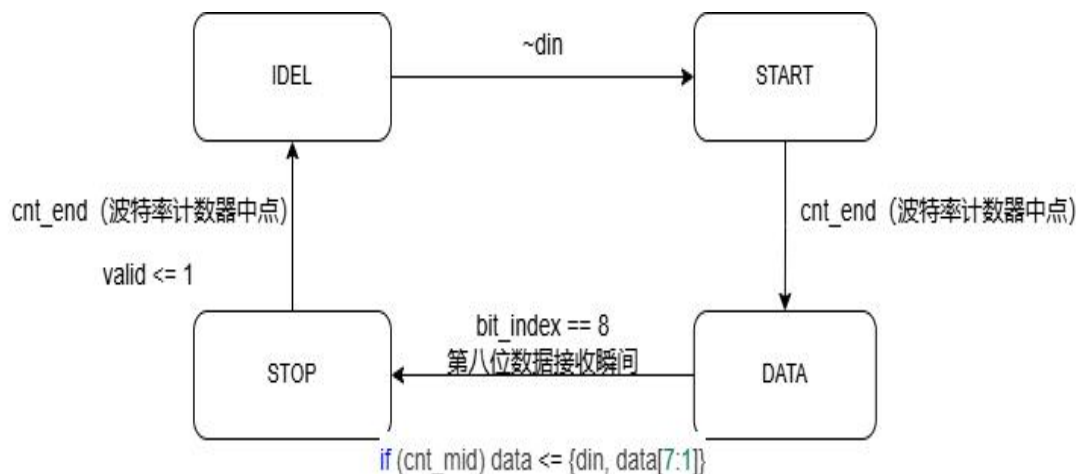
always @ (posedge clk or posedge rst) begin
    if (rst) pos_edge <= 1'b0;
    else     pos_edge <= ~signal_prev & signal;
end

always @ (posedge clk or posedge rst) begin
    if (rst) neg_edge <= 1'b0;
    else     neg_edge <= signal_prev & ~signal;
end
```

(3) uart_recv:

①功能：接收从电脑端口软件发送的数据。

②实现逻辑：使用状态机，状态转换图如下



③输入输出信号：

input: clk, rst, din

output: valid, [7:0]data

④关键代码（状态机三段式代码及说明）：

a. 状态迁移描述：

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

b. 状态转移条件判断描述：

```
always @(*) begin
    case (current_state)
        IDLE: begin
            if (~din) next_state = START;
            else      next_state = IDLE;
        end
        START: begin
            if (cnt_mid) next_state = DATA;
            else          next_state = START;
        end
        DATA: begin
            if (bit_index < 8) next_state = DATA; // 发送 8 位数据
            else                next_state = STOP; // 8 位数据发送完进入 STOP
        end
        STOP: begin
            if (cnt_mid) next_state = IDLE;
            else          next_state = STOP;
        end
        default: next_state = IDLE;
    endcase
end
```

类似于 $\text{cnt_end} = (\text{cnt} == \text{BAUD_DEVIDER})$ ，这里的 cnt_mid 为 $\text{cnt_mid} = (\text{cnt} == \text{BAUD_MID})$ ，其中 $\text{BAUD_DIVIDER} = 10416 - 1$ ， $\text{BAUD_MID} = 5208 - 1$ 。

c. 输出描述：

```
// data
always @(posedge clk or posedge rst) begin
    if (rst) data <= 8'h00;
```

```

else begin
    case (current_state)
        DATA: begin
            if (cnt_mid) data <= {din, data[7:1]};
            else      data <= data;
        end
        default: data <= data;
    endcase
end
end
end

```

当进入到 DATA 状态后，每当计数到计时中点时，对 data 进行一次右移，使得新数据位于 data 最左端。

```

// valid
always @(posedge clk or posedge rst) begin
    if (rst) valid <= 0;
    else begin
        case (current_state)
            STOP: begin
                if (cnt_mid) valid <= 1;
                else      valid <= 0;
            end
            default: valid <= 0;
        endcase
    end
end
end

```

d. bit_index 的更新:

```

always @(posedge clk or posedge rst) begin
    if (rst) bit_index <= 0;
    else begin
        case (current_state)
            START: bit_index <= 0;
            DATA: begin
                if (bit_index < 8 && cnt_mid) bit_index <= bit_index + 1'b1;
                else      bit_index <= bit_index;
            end
            default: bit_index <= 0;
        endcase
    end
end
end

```

bit_index 每次在 START 状态时重置为 0，在 DATA 状态时，一

旦在 `cnt_mid` 处接收数据, `bit_index` 进行一次加一更新, 直至接收到第八个数据时, `bit_index` 保持为 8, 这时状态机状态将会在下一个时钟周期的上升沿转换到 `STOP`, `data` 不再更新。

e. 计数器 `cnt` 何时归零:

在计数器的 `always` 块中, 加以 `if` 判断

```
else if (current_state == IDLE) baud_cnt <= 0;
```

(4) `led_display`:

①功能: 实现基本功能概述中“LED 数码管显示”的功能, 这里不再赘述。

②实现逻辑:

a. 预备模块:

基于数码管的显示逻辑, 需要时钟分频信号 `freq_div` 输出一个 2ms 的时钟对 `EN` 使能信号不断进行移位。而移位是在分频信号的上升沿进行, 这也就需要再次使用 `edge_detect` 模块以检测分频信号的上升沿。

该模块中还存在一个 `led_list` 子模块, 对于传入的 `num`, 输出其对应的 LED 数码管显示信号 `SEGS` (使用 `case` 即可)。如传入 `num = 1`, 则输出 `8'b10011111`, 若传入的 `num` 不再“0”~“9”或“A”~“F”范围内, 则输出 `8'b11111111`

上述 `freq_div`、`led_list` 模块并没有单独列为功能模块体现在功能概述的模块框图中, 主要原因是它们的逻辑是简单的, 没有必要使用

大量篇幅描述它们。

b. 逻辑:

由于要求接收信号不足八位时，高位不显示，因此需要对接收到的信号进行计数。如新开一个变量 `read`，`uart_recv` 接收到一个完整八位信号时，会输出一个时钟周期的 `valid` 有效信号，可以此为依据对接收到的信号计数。

新开一个 `data_flag` 变量，根据 `read` 的情况对其赋值，用于判断数码管的显示情况。如当 `read = 2`，证明已经读入两个完整数据，则 `data_flag` 被赋为 `8'b00000011`。在数码管显示时，如果 `EN == 8'b11111110`，即第 0 位数码管使能有效，判断 `data_flag[0]` 是否有效，如果有效则显示，如果无效则 `SEGS <= 8'b11111111`，不显示。

③输入输出信号:

input: `clk`, `rst`, `[7:0]recv_data`, `recv_valid`

output: `[7:0]EN`, `[7:0]SEGS`

④关键代码:

a. 接收数据读取:

```
reg [7:0] seg_data [7:0];
always @(posedge clk or posedge rst) begin
    if (rst)
        seg_data[0] <= 8'b0;
    else if (recv_valid)
        seg_data[0] <= recv_data;

    always @(posedge clk or posedge rst) begin
        if (rst)
            seg_data[1] <= 8'b0;
        else if (recv_valid)
            seg_data[1] <= seg_data[0];
```

2-7 位类推，这里的 `recv_valid` 即 `uart_recv` 输出的 `valid`。这样的代码看起来会比较冗杂，但是 `seg_data[i]` 与 `seg_data[j]` 本质上不是一

个变量，采用不同 `always` 块赋值是必要的。

b. read 与 flag_data:

```
always @(posedge clk or posedge rst) begin
    if (rst)
        read <= 0;
    else if (recv_valid && read < 7) read <= read + 1'b1;
    else if (read == 7)
        read <= read;
end

always @(posedge clk or posedge rst) begin
    if (rst)
        data_flag <= 8'b00000000;
    else if (recv_valid) begin
        case (read)
            0: data_flag <= 8'b00000001;
            1: data_flag <= 8'b00000011;
            ..... (略)
            default: data_flag <= 8'b0;
        endcase
    end
end
```

c. 数据显示信号

```
wire [7:0] SEGS_index [7:0];
led_list led0(
    .num(seg_data[0]),
    .SEGS(SEGS_index[0])
);
```

1-7 类推。

d. 综合逻辑实现:

```
always @(posedge clk or posedge rst) begin
    if (rst) SEGS <= 8'b11111111;
    else begin
        case(EN)
            8'b11111110: if (data_flag[0]) SEGS <= SEGS_index[0];
                        else
                        SEGS <= 8'b11111111;
            8'b11111101: if (data_flag[1]) SEGS <= SEGS_index[1];
                        else
                        SEGS <= 8'b11111111;
            ..... (略)
            default: SEGS <= 8'b11111111;
        endcase
    end
end
```

end

⑤重难点:

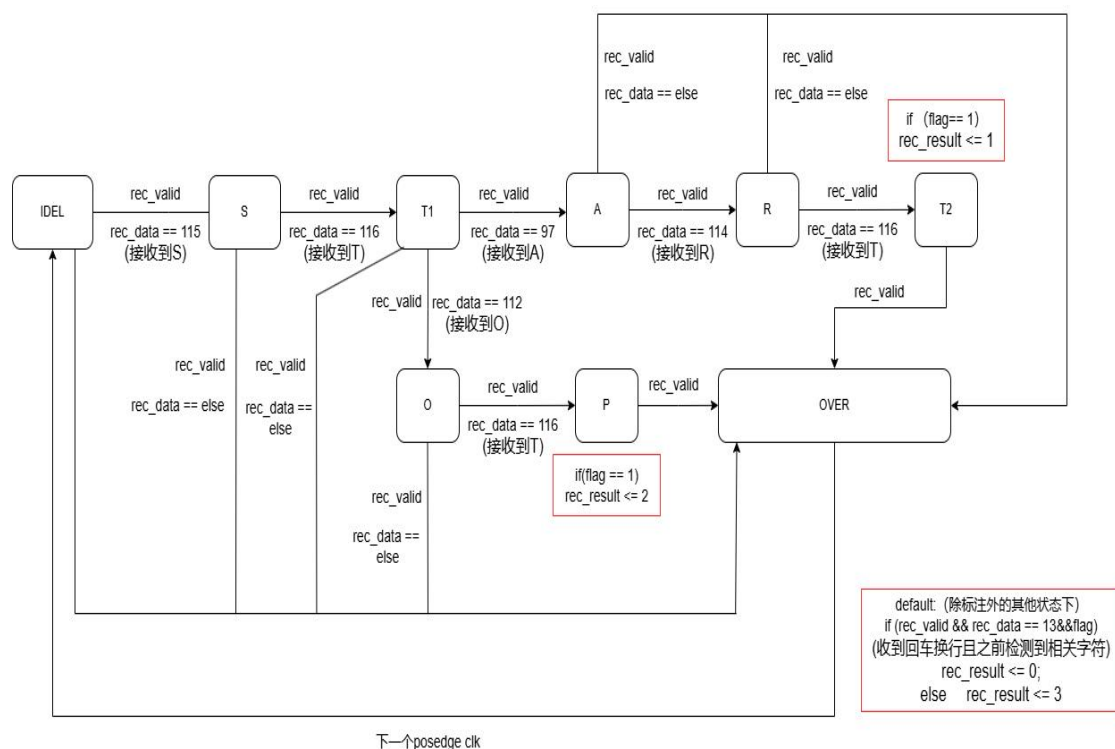
怎样满足接收未满八位时高位不显示、如何在显示新数据时旧数据左移。

(5) str_rec:

①功能：完成基本功能概述中“字符串检测”的功能，这里不再赘述。

②实现逻辑:

使用状态机，状态转换图及其说明如下:



各状态下，只有接收到特定字符才能进入到下一状态，只要接收到的字符不对立马进入到 **OVER**，并在下一个时钟上升沿回到 **IDEL**。

图中的 **rec_result** 代表检测了的结果，为 0 时代表没有检测到“start”与“stop”之中的任意一个，为 1 代表检测到了“start”，为

2 时代表检测到了“stop”，为 3 的情况下一段说明。图中的 flag 代表了之前是否已经检测到相关字符，如果检测到过，则 $\text{flag} > 1$ 。只有在 T2 或 P 状态下且 flag 为 1 时才会使 rec_result 为 1 或 2，这可以避免当需要检测的字符串为“...start...start...”时回复“11”的情况，但这也决定了如检测“start123stop”时，将回复“1”而非“2”或“12”。这就需要在第一次检测到指定字符时 flag 被赋值为 1，且只有在 $\text{recv_valid} \ \&\& \ \text{recv_data} == 13$ （回车）、即进行下一轮检测时才被赋 0。

flag 的具体实现纯靠文字难以说明，请跳转到下一部分的仿真分析中，参考波形更好理解。

由于回复的“0”“1”“2”需要通过 uart_send 发送，这就意味着需要有一个“valid”，由上面的说明可知，rec_result 为 1、2、0 的时长只会为一个时钟周期，故可以使用三个线网型变量

```
assign is_start = (rec_result == 1);
assign is_stop  = (rec_result == 2);
assign other    = (rec_result == 0);
```

即可。而当 rec_result 为 3 即为没有既没有检测到指定字符串、但检测还没有结束的状态，不会向 uart_send 传递任何信号。

③输入输出信号：

input: clk, rst, recv_valid, [7:0]recv_data

output: is_start, is_stop, other

④关键代码：

a. 状态机三段式：

```
always @(posedge clk or posedge rst) begin
```

```
    if (rst) current_state <= IDLE;
    else     current_state <= next_state;
end

always @(posedge clk or posedge rst) begin
    if (rst) next_state <= IDLE;
    else begin case (current_state)
        IDLE: begin
            if (recv_valid)
                if (recv_data == 115) next_state <= S;
                else
                    next_state <= OVER;
            else next_state <= next_state;
        end
        S: begin
            if (recv_valid)
                if (recv_data == 116) next_state <= T1;
                else
                    next_state <= OVER;
            else next_state <= next_state;
        end
        T1: begin
            if (recv_valid)
                if (recv_data == 97)      next_state <= A;
                else if (recv_data == 111) next_state <= 0;
                else
                    next_state <= OVER;
            else next_state <= next_state;
        end
        A: begin
            if (recv_valid)
                if (recv_data == 114) next_state <= R;
                else
                    next_state <= OVER;
            else next_state <= next_state;
        end
        R: begin
            if (recv_valid)
                if (recv_data == 116) next_state <= T2;
                else
                    next_state <= OVER;
            else next_state <= next_state;
        end
        T2: begin
            if (recv_valid) next_state <= OVER;
            else
                next_state <= next_state;
        end
        0: begin
            if (recv_valid)
```

```

        if (recv_data == 112) next_state <= P;
        else
            next_state <= OVER;
        else next_state <= next_state;
    end
P: begin
    if (recv_valid) next_state <= OVER;
    else
        next_state <= next_state;
    end
OVER: next_state <= IDLE;
default: ;
endcase
end
end

```

```

always @(posedge clk or posedge rst) begin
    if (rst)
        rec_result <= 3;
    else if (~recv_valid)
        rec_result <= 3;
    else begin case (current_state) // recv_valid = 1
        T2: begin
            if (flag == 1)
                rec_result <= 1;
            else
                rec_result <= 3;
        end
        P: begin
            if (flag == 1)
                rec_result <= 2;
            else
                rec_result <= 3;
        end
        default: begin
            if (recv_data == 13) begin
                if (flag)
                    rec_result <= 3;
                else
                    rec_result <= 0;
            end
            else
                rec_result <= 3;
        end
    endcase
end
end

```

b. flag 信号

```

reg [31:0] flag;
wire o,r;
assign {o,r} = {current_state == 0 && recv_data == 112, current_state ==
R && recv_data == 116};

```

```

always @(posedge clk or posedge rst) begin
    if (rst)                                flag <= 0;
    else if (recv_valid && recv_data == 13) flag <= 0;
    else if (recv_valid && (o || r))         flag <= flag + 1;
    else                                    flag <= flag;
end

```

⑤重难点:

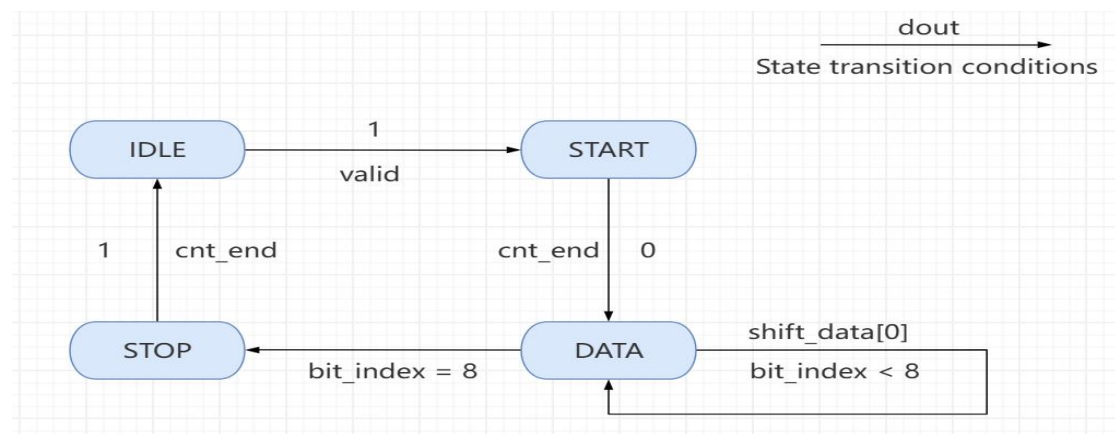
- 对于连续序列的检测，是否能够联想到状态机为一难点。
- 对于待检测字符串中包含多个目标字符串时，如何避免重复发送或冲突是一难点。
- 如何将检测的结果作为“valid”传达给发送模块是一难点，也是重点。

(6) uart_send:

①功能：当按键 S3 按下时，将拨码开关 SW7-SW0 的八位数据发送出去，且 S3 按下不动时只发送一次；对 rst_rec 模块中检测的结果进行发送。

②实现逻辑:

发送逻辑使用状态机，状态转换图如下：



这里的 valid 是经过消抖后的 S3 标准信号的上升沿 pos_edge_s3 与 str_rec 模块输出的 is_start, is_stop, other 共四个信号的或。这样做的依据是上述四个信号都只有一个时钟周期的有效信号,且现实中几乎不会出现按键的同时刚好要发送字符串检测结果。

其他的逻辑已在实验四实验报告中阐述过,这里不再赘述。

③输入输出信号:

input: clk, rst, valid, [7:0]data, recv_valid, [7:0]recv_data

output: dout

输入信号有 uart_recv 输出的 recv_valid, [7:0]recv_data, 因为 str_rec 模块是内置在 uart_send 模块中的。

④关键代码:

a. 发送使能信号:

```
assign imp_valid = (is_start | is_stop | other | valid);
```

b. 要发送的数据:

```
always @(posedge clk or posedge rst) begin
    if (rst) shift_data <= 0; // 复位时, 数据清零
    else begin
        case (current_state)
            IDLE: begin
                if (valid)          shift_data <= data;
                else if (is_start) shift_data <= 8'b00110001;
                else if (is_stop)  shift_data <= 8'b00110010;
                else if (other)    shift_data <= 8'b00110000;
                else                shift_data <= shift_data;
            end
            DATA: begin
                if (bit_index < 7 && cnt_end) shift_data <= {1'b0,
shift_data[7:1]}; // 移一位, 准备发送下一个数据位, bit_index = 7 时不移
                else shift_data <= shift_data;
            end
            default: ;
        endcase
    end
end
```



```

        endcase
    end
end

```

需要说明的是，`bit_index = 7` 时 `shift_data` 不移位，否则若接收数据的最后一位数据为 1，会出现一个时钟周期的低电平毛刺。

c. 状态机：

```

// 第 1 个 always 块，描述状态迁移
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)
        IDLE: begin
            if (imp_valid) next_state = START;
            else next_state = IDLE;
        end
        START: begin
            if (cnt_end) next_state = DATA;
            else         next_state = START;
        end
        DATA: begin
            if (bit_index < 8) next_state = DATA; // 发送 8 位数据
            else               next_state = STOP; // 8 位数据发送完后，进入
STOP
        end
        STOP: begin
            if (cnt_end) next_state = IDLE;
            else         next_state = STOP;
        end
        default: next_state = IDLE;
    endcase
end

// 第 3 个 always 块，描述输出逻辑 (dout)
always @(posedge clk or posedge rst) begin
    if (rst) dout <= 1'b1; // 复位时，dout 保持高电平
    else begin
        case (current_state)

```

```
IDLE: dout <= 1'b1;
START: dout <= 1'b0;
DATA: dout <= shift_data[0];
STOP: dout <= 1'b1;
default: dout <= 1'b1; // 默认状态
endcase
end
end
```

⑤重难点:

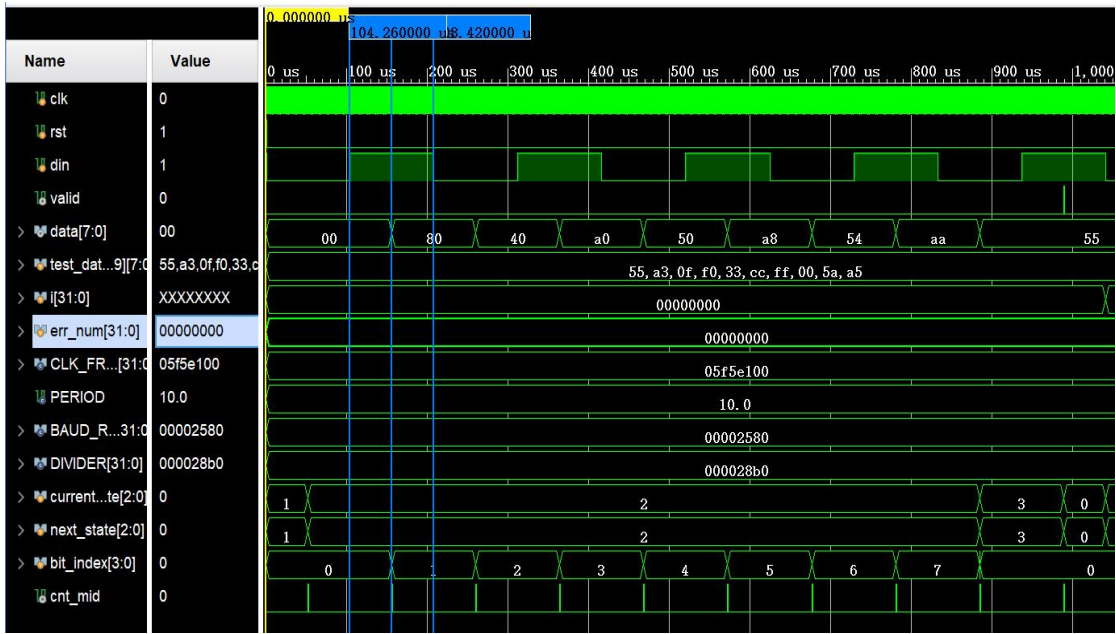
如何将 `str_rec` 合理的嵌入到发送模块中，以及如何合理地布置发送使能信号 `imp_valid`、如何合理的布置要发送的信号 `shift_data` 是难点，也是重点。

调试报告

仿真波形截图及仿真分析

(1) UART 接收核心功能的仿真分析:

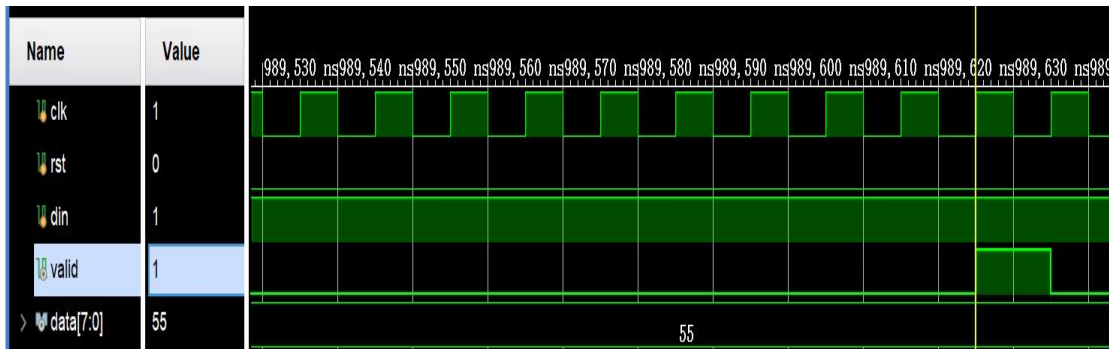
下图为一个完整数据帧的接收仿真图:



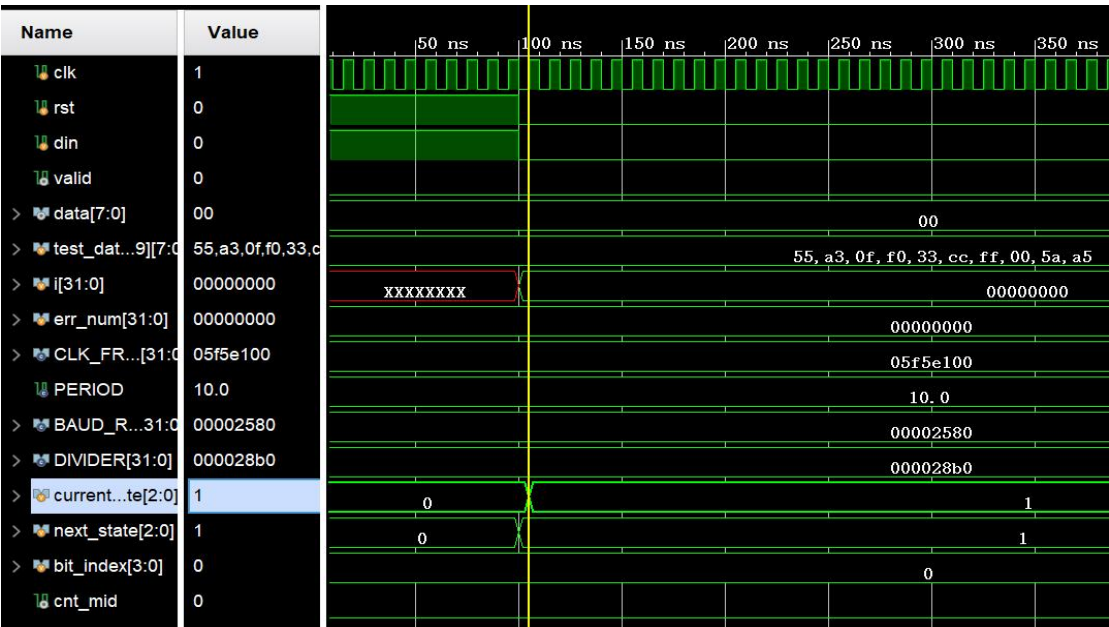
可见 `cnt_mid` 出现在每个数据位的中点。

在启动信号的 cnt_mid 处，状态转为 2（DATA）；在 DATA 状态下的 cnt_mid 处，bit_index 加了 1，除了最后一个数据位处回到 0，之后不再改变，此时看接收到的数据 data，已经完整接收到数据，并且之后不再改变。

在接收结束信号的 cnt_mid 处，状态回到 0（IDEL），并且接收有效信号 valid 出现，放大可见 valid 信号为一个时钟周期。



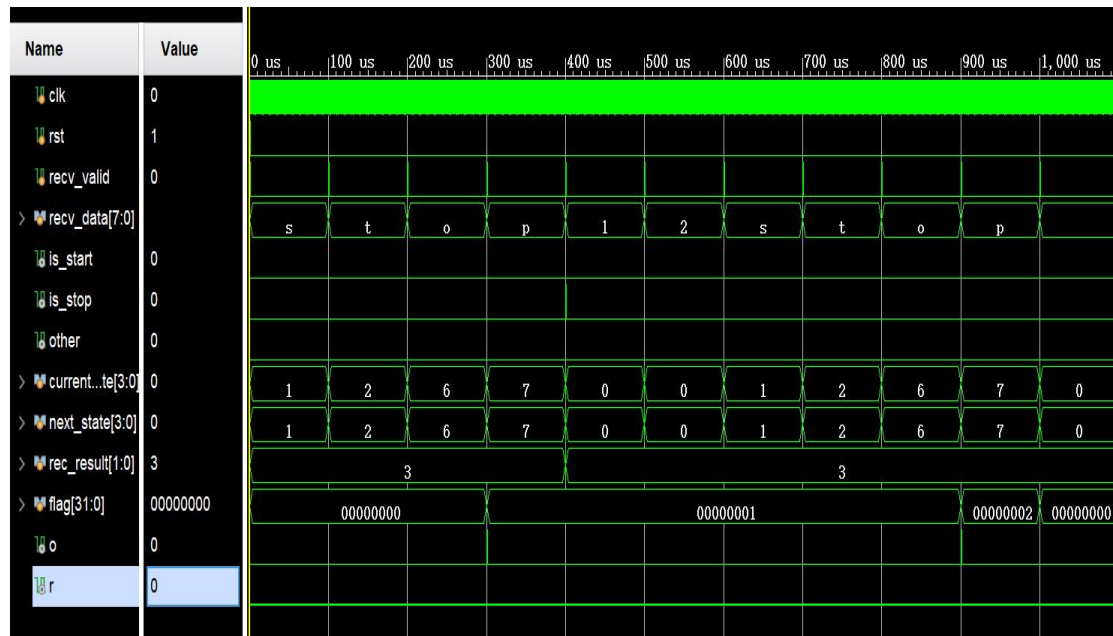
不同的是，状态由 IDEL 到 START 的转变并不看 cnt_mid，只要检测到 ~din，即实现转变，如下图所示。



所有状态的转换与状态转换图完全一致。

(2) 附加题 str_rec 的主要功能的仿真分析：

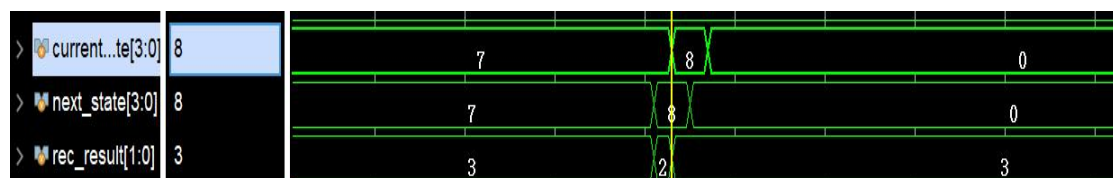
对 “stop12stop\n” 的整体仿真图如下：



将 flag 的代码迁移下来以作参考：

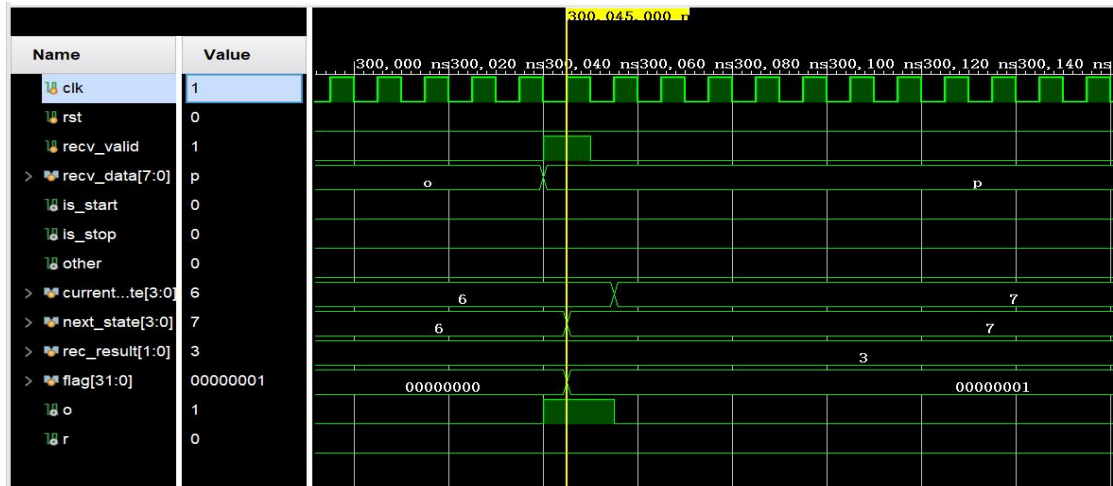
```
reg [31:0] flag;
wire o,r;
assign {o,r} = {current_state == 0 && recv_data == 112, current_state ==
R && recv_data == 116};
```

```
always @(posedge clk or posedge rst) begin
    if (rst) flag <= 0;
    else if (recv_valid && recv_data == 13) flag <= 0;
    else if (recv_valid && (o || r)) flag <= flag + 1;
    else flag <= flag;
end
```

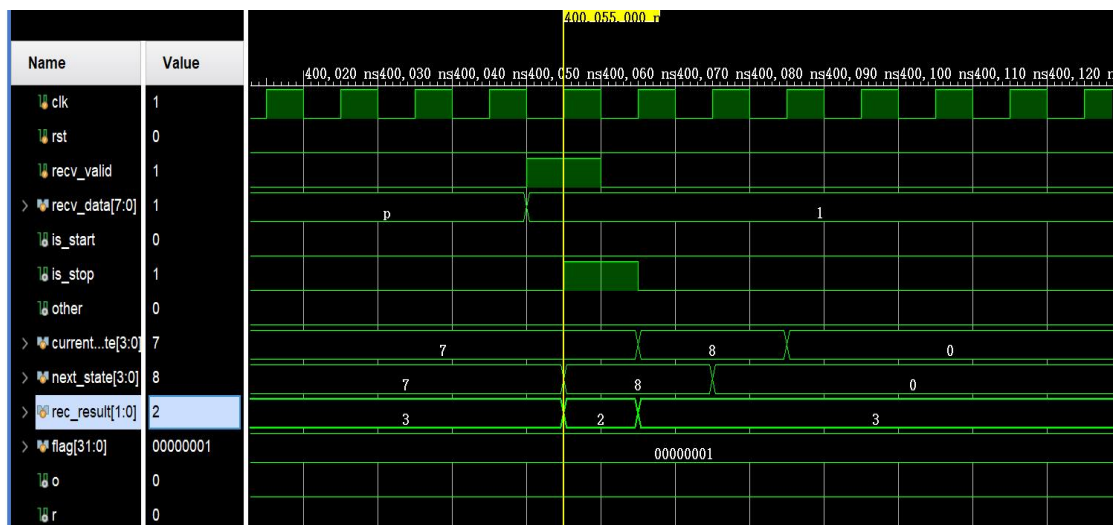


由波形可见，每次接收到新字符时状态都会转变，当字符不符合自身期望时，状态立马转到 over 并在下一个时钟周期的上升沿回到 IDLE，

具体如上图所示。



如上图所示，当 `recv_valid` 来临，`recv_data == "p"` 时，状态仍然停留在 O 状态，此时线网型变量 `o` 为 1，满足 `recv_valid && (o || r)` 的条件，因此 `flag` 由 0 加到 1。



如上图所示，在状态为 P 状态时，下一个 `recv_valid` 到来，此时 `flag == 1`，因此 `rec_result` 变为 2，进而 `is_stop` 变为 1，向 `uart_send` 模块发送了使能信号。可以看到整体仿真波形中，第二次 `o` 为 1 时，`flag` 加为 2，但是下一次 `rev_valid` 到来时 `is_stop` 并没有变为 1。

综上所述，模块做到了检测，但不重复检测的功能。

设计过程中遇到的问题及解决方法

以上述仿真中避免重复检测为例。

为了避免重复检测，出现了原来的检测功能无法正常实现的问题，通过写仿真代码，锁定到 bug 如下：

最初的 flag 的赋值代码为

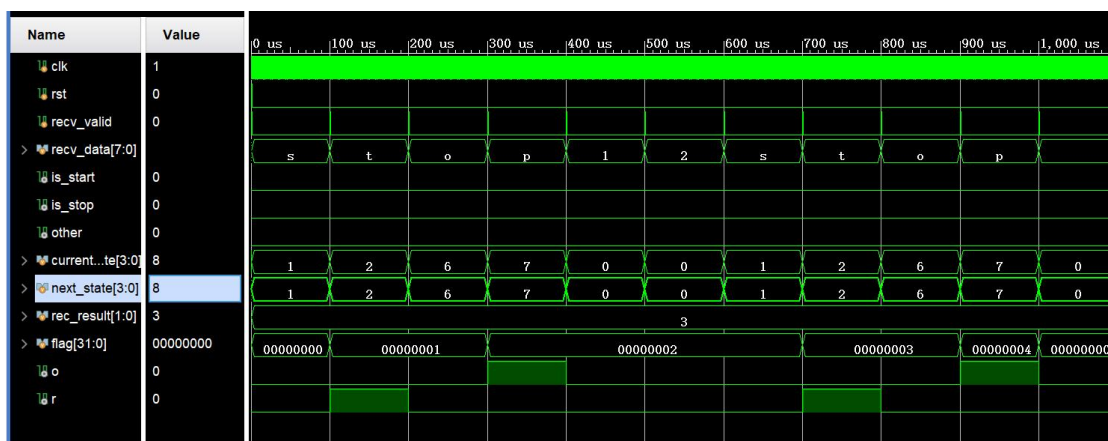
```
assign {o,r} = {recv_data == 112, recv_data == 116};
always @(posedge clk or posedge rst) begin
    if (rst)                    flag <= 0;
    else if (recv_valid && recv_data == 13) flag <= 0;
    else if (recv_valid && (o || r))      flag <= flag + 1;
    else                             flag <= flag;
end
```

这段代码存在的问题为以下两点：

1. 最初的代码

```
assign {o,r} = {recv_data == 112, recv_data == 116};
```

导致在检测“stop”时，由于其中也包含了“t”，因此在 o 被赋值为 1 之前，r 先被赋值为 1，如下图所示



这导致 flag 提前变为 1，并在真正检测到“p”时 flag 变为了 2，使得检测失败。

改正的方法就是加以状态上的限制，通过放大可知，在“p”的

rec_valid 到来时，状态其实是停留在 O 上的，因此需要加以限制

current_state = O，对于变量 r 同理。因此代码改正为

```
assign {o,r} = {current_state == O && recv_data == 112, current_state == R &&  
recv_data == 116};
```

即可完成修正。

课程设计总结

(1) 本实验写代码（不包括思考时间，只包括写代码与 debug）共花费约 5h，撰写报告约花费 4h.

(2) 课程收获、总结：

个人 Verilog 代码水平以及 debug 有了极大提升，对数字逻辑系统的模块化设计理解有了极大提升，心态得到磨合提高，能够耐心地写 testbench 代码进行精准的 debug。