实验四 综合电路设计实验报告

组长：左逸龙　　**学号**：1120231863

班级：07112303　　**手机**：18680517248

组员：陈墨霏　　**学号**：1120233329

班级：07112304　　**手机**：13126146305

1. 实验题目

短跑计时器设计与实现（难度系数：0.9）

- 短跑计时器描述如下：
  - 短跑计时器显示分、秒、毫秒；
  - “毫秒”用两位数码管显示：百位、十位；
  - “秒”用两位数码管显示：十位、个位；
  - “分”用一位 LED 灯显示，LED 灯“亮”为 1 分；
  - 最大计时为 1 分 59 秒 99，超限值时应可视或可闻报警；
  - 三个按键开关：计时开始/继续（A）、计时停止/暂停（B）、复位/清零（C）

  键控流程如下：

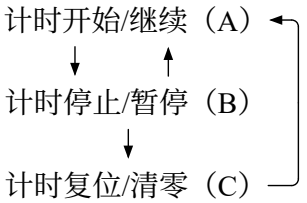


图 1　短跑计时器的键控流程

2. 电路设计

a) 输入输出设计

短跑计时器电路包含5个输入信号和4个输出信号。输入信号包括系统时钟(`sys_clk_pin`)、系统复位(`reset_n_pin`)以及三个按键控制信号(`key_a_pin`、`key_b_pin`、`key_c_pin`)。其中系统时钟提供 100MHz 的基准时钟，系统复位为低电平有效，用于将系统恢复到初始状态。三个按键分别用于控制计时的开始/继续、停止/暂停以及复位/清零操作。

输出信号包括七段数码管段选信号(seg_pins)、位选信号(an_pins)以及两个 LED 指示灯(led_minute_pin、led_alarm_pin)。七段数码管用于显示时间，其中段选信号控制各段点亮，位选信号控制显示位置。分钟 LED 用于指示是否达到 1 分钟，报警 LED 用于指示计时是否超过最大计时值(1 分 59 秒 99)。

表 1　短跑计时器的输入和输出变量

输入符号	名称	值为 1 的含义	值为 0 的含义
sys_clk_pin	系统时钟	时钟上升沿	时钟下降沿
reset_n_pin	系统复位	正常工作	系统复位
key_a_pin	按键 A	计时开始/继续	无操作
key_b_pin	按键 B	计时停止/暂停	无操作
key_c_pin	按键 C	计时复位/清零	无操作
输出符号	名称	值为 1 的含义	值为 0 的含义
seg_pins	七段数码管段选	段点亮	段点灭
an_pins	数码管位选	位选有效	位选无效
led_minute_pin	分钟 LED	分钟指示	无指示
led_alarm_pin	报警 LED	报警指示	无报警

b) 模块设计

按键消抖模块 button_interface_debounce

按键消抖模块是短跑计时器的基础输入处理模块，其主要功能是对原始按键输入进行消抖处理并产生单周期脉冲信号。该模块采用三级同步器结构，通过 key_state_q0、key_state_q1 和 key_state_q2 三个寄存器实现按键状态的稳定采样和边沿检测。模块内部包含一个 20 位的消抖计数器，在 100MHz 系统时钟下可实现 10ms 的消抖时间。当检测到按键状态发生变化时，计数器会立即复位；当按键状态保持稳定且计数器达到预设值时，才会更新稳定的按键状态。最后通过比较相邻两个稳定状态，在检测到上升沿时产生一个时钟周期的有效脉冲信号，确保每次按键操作只触发一次响应。

状态控制模块 state_controller

状态控制模块是整个计时器的核心控制单元，采用 Moore 型状态机实现，包含 IDLE（空闲）、RUNNING（运行）、PAUSED（暂停）和 ALARM（报警）四个状态。该模块根据按键输入和最大计时时间到达信号，控制计时器的运行、暂停、复位以及报警状态。

在 IDLE 状态下，系统等待开始信号；RUNNING 状态下，计时器正常运行，可响应暂停或复位命令；PAUSED 状态下，计时器暂停，等待继续或复位命令；ALARM 状态下，系统发出报警信号，

等待复位命令。模块输出包括计时器运行使能、复位命令和报警激活信号，这些信号直接控制其他模块的工作状态。

状态控制模块采用 Moore 型状态机实现，其输出仅取决于当前状态。在 IDLE 状态下，timer_run_en_out 为 0 表示计时器停止运行，timer_reset_cmd_out 为 1 表示持续请求复位计时器，alarm_active_out 为 0 表示无报警。在 RUNNING 状态下，timer_run_en_out 为 1 表示计时器正常运行，timer_reset_cmd_out 为 0 表示不复位，alarm_active_out 为 0 表示无报警。在 PAUSED 状态下，timer_run_en_out 为 0 表示计时器暂停，timer_reset_cmd_out 为 0 表示不复位，alarm_active_out 为 0 表示无报警。在 ALARM 状态下，timer_run_en_out 为 0 表示计时器停止运行，timer_reset_cmd_out 为 0 表示不复位，alarm_active_out 为 1 表示激活报警。

根据上述对于状态的说明和状态转移的描述，可以得到以下的短跑计时器的状态转移表。

表 2　短跑计时器的状态转移表

当前状态	当前状态状态码	转换条件	下一状态	下一状态状态码
IDLE	00	key_a_pulse	RUNNING	01
		其他情况	IDLE	00
RUNNING	01	key_b_pulse	PAUSED	10
		key_c_pulse	IDLE	00
		max_time_reached_in	ALARM	11
		其他情况且未超时	RUNNING	01
PAUSED	10	key_a_pulse	RUNNING	01
		key_c_pulse	IDLE	00
		其他情况	PAUSED	10
ALARM	11	key_c_pulse	IDLE	00
		其他情况	ALARM	11

根据状态转移表，可以绘制出以下的短跑计时器的状态机图。

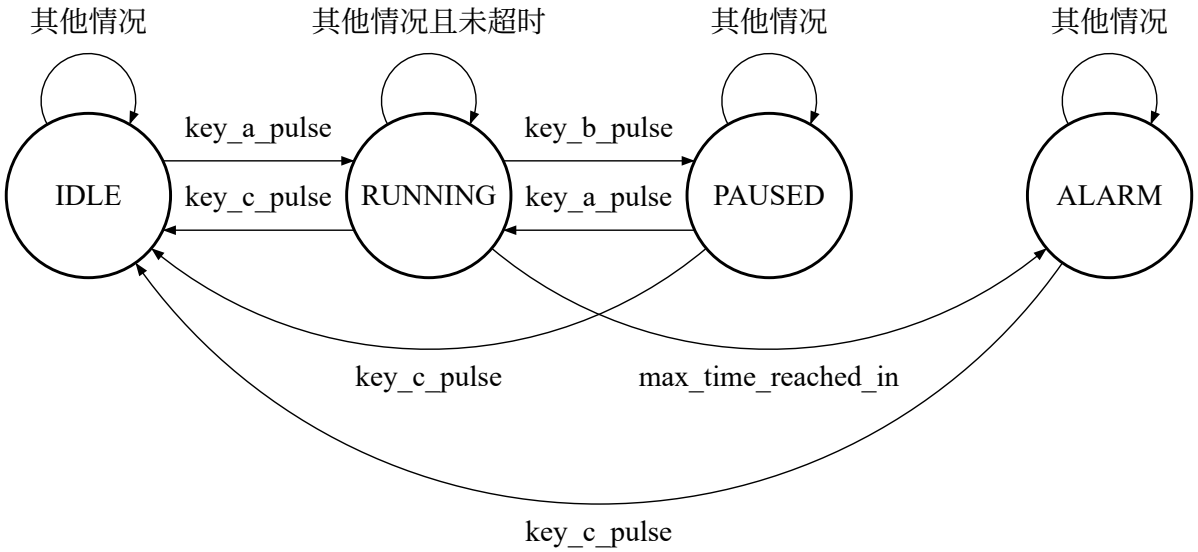


图 2　短跑计时器的状态机图

计数器模块 clk_divider_counter

计数器模块是计时器的核心计时单元，负责产生各种时钟信号并进行时间计数。该模块首先通过分频产生 10ms 的计时脉冲，同时生成数码管扫描时钟和报警 LED 闪烁时钟。在计时功能方面，模块采用 BCD 码计数方式，分别对毫秒（0-99）、秒（0-59）和分钟（0-1）进行计数。毫秒计数采用 7 位二进制计数器，通过组合逻辑转换为两位 BCD 码输出；秒计数采用 6 位二进制计数器，同样转换为两位 BCD 码；分钟则使用 1 位二进制计数器。模块还实现了最大计时时间（1 分 59 秒 990 毫秒）的检测功能，当达到最大时间时，会置位 `max_time_reached` 信号，触发报警状态。

输出显示模块 display_driver

输出显示模块负责将计时器的数值转换为可视化的显示信号。该模块通过七段数码管和 LED 灯两种显示器件来展示计时信息。七段数码管采用共阴极接法，其段选信号（a-g）高电平有效，分别控制数码管的 7 个发光段，从高位到低位依次为 a、b、c、d、e、f、g。同时，模块还通过 LED 灯来显示分钟和报警状态，其中报警 LED 采用闪烁方式提示。

该模块采用动态扫描方式驱动 4 位七段数码管，显示顺序为秒十位、秒个位、百毫秒位和十毫秒位。模块内部包含一个 2 位计数器用于位选控制，通过 `scan_clk_enable` 信号控制扫描频率。在显示控制方面，模块将输入的 BCD 码转换为七段数码管的段选信号，同时控制位选信号 `an_out` 实现动态扫描。对于分钟显示，模块直接驱动 LED 灯；对于报警状态，模块通过 `blink_clk_enable` 信号控制报警 LED 的闪烁效果。整个显示过程采用共阴极数码管，段选信号高电平有效，位选信号低电平有效，确保了显示效果的清晰和稳定。

c) 模块连接

根据上述模块的功能描述，我们可以将各个模块连接起来，形成完整的短跑计时器系统。系统的顶层模块将状态机控制器、计数器模块和显示驱动模块连接在一起，实现计时、控制和显示功能。下图展示了各个模块之间的连接关系。

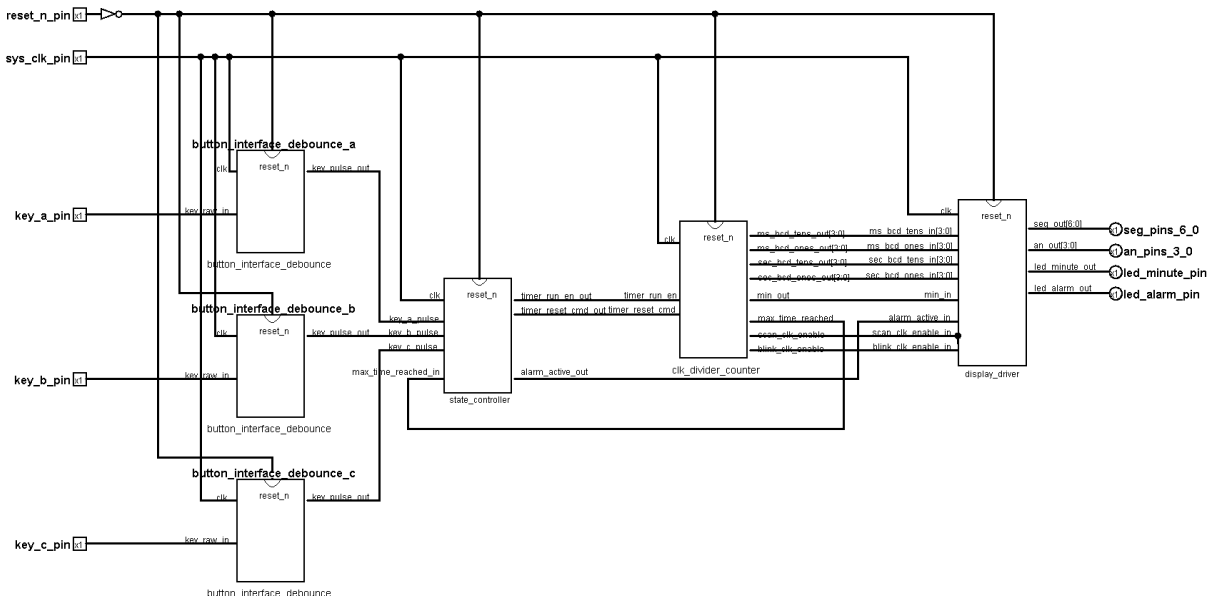


图 3　短跑计时器的电路模块图

3. 电路实现

模块：top_stopwatch

功能：短跑计时器顶层模块，实例化并连接所有子模块。

```
module top_stopwatch (
    // 输入端口
    input  wire        sys_clk_pin,      // 系统时钟输入
    input  wire        reset_n_pin,      // 系统复位按键输入（低电平有效）
    input  wire        key_a_pin,        // 按键A（开始/继续）（高电平有效）
    input  wire        key_b_pin,        // 按键B（暂停）     （高电平有效）
    input  wire        key_c_pin,        // 按键C（复位）     （高电平有效）

    // 输出端口
    output wire [6:0]  seg_pins,         // 七段数码管段选（A-G，G=[6]）
    output wire [3:0]  an_pins,          // 四位片选（低有效，[3]是最左边）
    output wire        led_minute_pin,   // 分钟LED（高有效）
    output wire        led_alarm_pin     // 报警LED（高有效）
);

    // 内部信号线（wires）
    wire key_a_pulse_w;
    wire key_b_pulse_w;
```

```
19      wire key_c_pulse_w;
20
21      wire timer_run_en_w;
22      wire timer_reset_cmd_w;
23      wire alarm_active_w;
24
25      wire [3:0] ms_bcd_tens_w;
26      wire [3:0] ms_bcd_ones_w;
27      wire [3:0] sec_bcd_tens_w;
28      wire [3:0] sec_bcd_ones_w;
29      wire       min_val_w;
30      wire       max_time_reached_w;
31      wire       scan_clk_enable_w;
32      wire       blink_clk_enable_w;
33
34      // 实例化按键消抖模块 (每个按键一个)
35      button_interface_debounce u_debounce_A (
36          .clk            (sys_clk_pin),
37          .reset_n        (reset_n_pin), // 所有模块共用一个主复位
38          .key_raw_in     (key_a_pin),
39          .key_pulse_out  (key_a_pulse_w)
40      );
41
42      button_interface_debounce u_debounce_B (
43          .clk            (sys_clk_pin),
44          .reset_n        (reset_n_pin),
45          .key_raw_in     (key_b_pin),
46          .key_pulse_out  (key_b_pulse_w)
47      );
48
49      button_interface_debounce u_debounce_C (
50          .clk            (sys_clk_pin),
51          .reset_n        (reset_n_pin),
52          .key_raw_in     (key_c_pin),
53          .key_pulse_out  (key_c_pulse_w)
54      );
55
56      // 实例化状态控制器模块
57      state_controller u_state_controller (
58          .clk                    (sys_clk_pin),
59          .reset_n                (reset_n_pin),
60          .key_a_pulse            (key_a_pulse_w),
61          .key_b_pulse            (key_b_pulse_w),
62          .key_c_pulse            (key_c_pulse_w),
63          .max_time_reached_in    (max_time_reached_w),
64
65          .timer_run_en_out       (timer_run_en_w),
66          .timer_reset_cmd_out    (timer_reset_cmd_w),
67          .alarm_active_out       (alarm_active_w)
68      );
```

```
69
70      // 实例化时钟分频与计时计数模块
71      clk_divider_counter u_clk_divider_counter (
72          .clk                (sys_clk_pin),
73          .reset_n            (reset_n_pin),
74          .timer_run_en       (timer_run_en_w),
75          .timer_reset_cmd    (timer_reset_cmd_w),
76
77          .ms_bcd_tens_out    (ms_bcd_tens_w),
78          .ms_bcd_ones_out    (ms_bcd_ones_w),
79          .sec_bcd_tens_out   (sec_bcd_tens_w),
80          .sec_bcd_ones_out   (sec_bcd_ones_w),
81          .min_out            (min_val_w),
82          .max_time_reached   (max_time_reached_w),
83          .scan_clk_enable    (scan_clk_enable_w),
84          .blink_clk_enable   (blink_clk_enable_w)
85      );
86
87      // 实例化显示驱动模块
88      display_driver u_display_driver (
89          .clk                  (sys_clk_pin),
90          .reset_n              (reset_n_pin),
91          .ms_bcd_tens_in       (ms_bcd_tens_w),
92          .ms_bcd_ones_in       (ms_bcd_ones_w),
93          .sec_bcd_tens_in      (sec_bcd_tens_w),
94          .sec_bcd_ones_in      (sec_bcd_ones_w),
95          .min_in               (min_val_w),
96          .alarm_active_in      (alarm_active_w),
97          .scan_clk_enable_in   (scan_clk_enable_w),
98          .blink_clk_enable_in  (blink_clk_enable_w),
99
100         .seg_out              (seg_pins),
101         .an_out               (an_pins),
102         .led_minute_out       (led_minute_pin),
103         .led_alarm_out        (led_alarm_pin)
104     );
105
106 endmodule
```

模块：clk_divider_counter

功能：

1. 产生 10ms 的计时脉冲 (tick_10ms)
2. 进行分钟、秒、毫秒(百毫秒、十毫秒)的 BCD 计数
3. 判断是否达到最大计时时间 (1 分 59 秒 990 毫秒)
4. 产生数码管动态扫描使能信号 (scan_clk_enable)
5. 产生报警 LED 闪烁使能信号 (blink_clk_enable)

```v
`define SIMULATION_MODE // 定义仿真模式，上板烧录时需要注释掉

module clk_divider_counter (
    input  wire        clk,              // 系统时钟 (100MHz)
    input  wire        reset_n,          // 异步复位信号 (低电平有效)
    input  wire        timer_run_en,     // 计时运行使能 (来自FSM)
    input  wire        timer_reset_cmd,  // 计时复位命令 (来自FSM，高电平有效)

    output reg [3:0]   ms_bcd_tens_out,  // 毫秒显示的十位 (代表百毫秒，0-9)
    output reg [3:0]   ms_bcd_ones_out,  // 毫秒显示的个位 (代表十毫秒，0-9)
    output reg [3:0]   sec_bcd_tens_out, // 秒显示的十位 (0-5)
    output reg [3:0]   sec_bcd_ones_out, // 秒显示的个位 (0-9)
    output reg         min_out,          // 分钟显示 (0-1，使用单独的LED显示)

    output reg         max_time_reached, // 到达最大计时时间标志
    output wire        scan_clk_enable,  // 数码管扫描时钟使能
    output wire        blink_clk_enable  // 报警LED闪烁时钟使能
);

    // 参数定义
    `ifdef SIMULATION_MODE // 仿真模式加速1000倍
        localparam CNT_10MS_MAX = 1000 - 1;
        localparam CNT_SCAN_MAX = 100 - 1;
        localparam CNT_BLINK_MAX = 10000 - 1;
    `else // 上板烧录模式正常运行
        // 100MHz 时钟: 1 周期 = 10ns
        // 1. 10ms (100Hz) 计时脉冲计数器参数
        localparam CNT_10MS_MAX = 100_000_0 - 1; // 100MHz / 100Hz - 1 = 1,000,000 - 1

        // 2. 数码管扫描时钟使能 (扫描频率约 1kHz -> 4位整体刷新率250Hz)
        localparam CNT_SCAN_MAX = 100_000 - 1;   // 100MHz / 1kHz - 1

        // 3. 报警LED闪烁时钟使能 (闪烁频率约 5Hz)
        localparam CNT_BLINK_MAX = 10_000_000 - 1; // 100MHz / (2*5Hz) - 1 (产生约5Hz方波的半周期)
    `endif

    // 内部计数器
    reg [$clog2(CNT_10MS_MAX+1)-1:0] cnt_10ms;
    reg [$clog2(CNT_SCAN_MAX+1)-1:0] cnt_scan;
    reg [$clog2(CNT_BLINK_MAX+1)-1:0] cnt_blink;

    // 计时脉冲信号
    wire tick_10ms;

    // 内部计时值 (二进制)
    reg [6:0] ms_counter;  // 0-99
    reg [5:0] sec_counter; // 0-59
    reg       min_counter; // 0-1
```

```
49
50      // 1. 产生各种时钟使能/脉冲
51      assign tick_10ms = (cnt_10ms == CNT_10MS_MAX);
52      assign scan_clk_enable = (cnt_scan == CNT_SCAN_MAX);
53      assign blink_clk_enable = (cnt_blink == CNT_BLINK_MAX);
54
55      always @(posedge clk or negedge reset_n) begin
56          if (!reset_n) begin
57              cnt_10ms  <= 0;
58              cnt_scan  <= 0;
59              cnt_blink <= 0;
60          end else begin
61              // 10ms 脉冲计数器
62              if (cnt_10ms == CNT_10MS_MAX) begin
63                  cnt_10ms <= 0;
64              end else begin
65                  cnt_10ms <= cnt_10ms + 1'b1;
66              end
67
68              // 数码管扫描使能计数器
69              if (cnt_scan == CNT_SCAN_MAX) begin
70                  cnt_scan <= 0;
71              end else begin
72                  cnt_scan <= cnt_scan + 1'b1;
73              end
74
75              // 报警闪烁使能计数器
76              if (cnt_blink == CNT_BLINK_MAX) begin
77                  cnt_blink <= 0;
78              end else begin
79                  cnt_blink <= cnt_blink + 1'b1;
80              end
81          end
82      end
83
84      // 2. 计时逻辑 (分、秒、毫秒)
85      always @(posedge clk or negedge reset_n) begin
86          if (!reset_n) begin
87              ms_counter  <= 7'd0;
88              sec_counter <= 6'd0;
89              min_counter <= 1'b0;
90              max_time_reached <= 1'b0;
91          end else if (timer_reset_cmd) begin // FSM发出的复位命令
92              ms_counter  <= 7'd0;
93              sec_counter <= 6'd0;
94              min_counter <= 1'b0;
95              max_time_reached <= 1'b0;
96          end else if (timer_run_en && tick_10ms) begin // 运行且10ms到
97              if (max_time_reached) begin
```

```
                // 如果已达最大时间，计时值清零，并且FSM会转到ALARM状态并设置timer_run_en为0
            end else begin
                // 毫秒计数 (0-99)
                if (ms_counter == 7'd99) begin
                    ms_counter <= 7'd0;
                    // 秒计数 (0-59)
                    if (sec_counter == 6'd59) begin
                        sec_counter <= 6'd0;
                        // 分钟计数 (0-1)
                        if (min_counter == 1'b1) begin // 原来是1分59秒990毫秒
                            min_counter <= 1'b1; // 保持在1，并置位max_time_reached
                            max_time_reached <= 1'b1;
                        end else begin // 原来是0分59秒990毫秒
                            min_counter <= min_counter + 1'b1; // 0 -> 1
                        end
                    end else begin // 秒不进位
                        sec_counter <= sec_counter + 1'b1;
                    end
                end else begin // 毫秒不进位
                    ms_counter <= ms_counter + 1'b1;
                end

                // 更新最大时间到达标志
                // (1分59秒990毫秒)
                if (min_counter == 1'b1 && sec_counter == 6'd59 && ms_counter == 7'd99) begin
                    max_time_reached <= 1'b1;
                end else begin
                    max_time_reached <= 1'b0; // 如果中途复位，确保标志也被复位
                end
            end
        end else if (!timer_run_en && !timer_reset_cmd) begin // 暂停状态，且不是复位命令
             max_time_reached <= max_time_reached; // 保持状态
        end
    end

    // 3. 将二进制计数值转换为BCD码输出
    // 毫秒 (0-99)
    always @(*) begin
        ms_bcd_tens_out = ms_counter / 10;
        ms_bcd_ones_out = ms_counter % 10;
    end

    // 秒 (0-59)
    always @(*) begin
        sec_bcd_tens_out = sec_counter / 10;
        sec_bcd_ones_out = sec_counter % 10;
    end

```

```
146     // 分 (0-1)
147     always @(*) begin
148         min_out = min_counter;
149     end
150 
151 endmodule
```

模块：button_interface_debounce

功能：对按键输入进行消抖处理，并产生单周期脉冲信号。
按键按下（输入高电平）时，输出一个时钟周期的有效脉冲。

```
1  `define SIMULATION_MODE // 定义仿真模式，上板烧录时需要注释掉
2  
3  module button_interface_debounce (
4      input  wire clk,             // 系统时钟 (100MHz)
5      input  wire reset_n,         // 异步复位信号 (低电平有效)
6      input  wire key_raw_in,      // 原始按键输入 (高电平有效)
7      output reg  key_pulse_out    // 消抖后的单周期按键脉冲 (高电平有效)
8  );
9  
10     // 参数定义
11     parameter DEBOUNCE_TIME_MS = 10; // 消抖时间，单位ms
12 
13     `ifdef SIMULATION_MODE
14         // 仿真模式下，设置一个很短的延时
15         localparam DEBOUNCE_COUNT_MAX = 4;
16     `else
17         // 上板烧录模式下，计算实际的延时
18         // 100MHz时钟下，10ms 对应的计数值：DEBOUNCE_TIME_MS * 100_000
19         localparam DEBOUNCE_COUNT_MAX = DEBOUNCE_TIME_MS * 100_000 - 1; // 计数器从0数
   到MAX
20     `endif
21 
22     // 内部信号定义
23     reg [$clog2(DEBOUNCE_COUNT_MAX + 1)-1:0] debounce_counter;   // 消抖计数器
24     reg key_state_q0;           // 按键状态寄存器0 (用于同步和边沿检测)
25     reg key_state_q1;           // 按键状态寄存器1 (稳定后的按键状态)
26     reg key_state_q2;           // 按键状态寄存器2 (用于产生脉冲)
27 
28     // 主逻辑
29     always @(posedge clk or negedge reset_n) begin
30         if (!reset_n) begin
31             debounce_counter <= 20'd0;
32             key_state_q0     <= 1'b0; // 假设按键未按下时为低电平
33             key_state_q1     <= 1'b0;
34             key_state_q2     <= 1'b0;
35             key_pulse_out    <= 1'b0;
```

```
        end else begin
            // 第一级同步，并检测原始输入变化
            key_state_q0 <= key_raw_in;

            if (key_state_q0 != key_raw_in) begin // 如果原始输入有变化，或者和第一级同步器不同:
                debounce_counter <= 20'd0; // 则复位消抖计数器
            end else if (debounce_counter < DEBOUNCE_COUNT_MAX) begin
                debounce_counter <= debounce_counter + 1'b1; // 如果稳定，则计数
            end else begin // 如果计数器达到最大值，则认为状态稳定
                key_state_q1 <= key_raw_in; // 因此我们更新稳定后的按键状态
            end

            // 使用稳定后的状态产生单周期脉冲
            // 其中 key_state_q1 是当前稳定状态，key_state_q2 是上一拍的稳定状态
            key_state_q2  <= key_state_q1;
            if (key_state_q1 == 1'b1 && key_state_q2 == 1'b0) begin // 检测到上升沿（按键按下）
                key_pulse_out <= 1'b1; // 则输出一个时钟周期的有效脉冲
            end else begin
                key_pulse_out <= 1'b0;
            end
        end
    end

endmodule
```

模块：state_controller

功能：控制计时器的启停、复位以及报警状态，实现短跑计时器的状态机控制逻辑。
状态: IDLE, RUNNING, PAUSED, ALARM

```
module state_controller (
    input  wire clk,                  // 系统时钟
    input  wire reset_n,              // 异步复位（低有效）
    input  wire key_a_pulse,          // 按键A脉冲（开始/继续）
    input  wire key_b_pulse,          // 按键B脉冲（停止/暂停）
    input  wire key_c_pulse,          // 按键C脉冲（复位/清零）
    input  wire max_time_reached_in,  // 最大计时时间到达信号

    output reg  timer_run_en_out,     // 计时器运行使能
    output reg  timer_reset_cmd_out,  // 计时器复位命令（高有效）
    output reg  alarm_active_out      // 报警激活信号
);

    // 状态定义
    localparam S_IDLE    = 2'b00; // 空闲/复位状态
    localparam S_RUNNING = 2'b01; // 计时运行状态
```

```
    localparam S_PAUSED  = 2'b10; // 计时暂停状态
    localparam S_ALARM   = 2'b11; // 报警状态

    // 状态寄存器
    reg [1:0] current_state;
    reg [1:0] next_state;

    // 状态转移逻辑 (时序逻辑: 更新当前状态)
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            current_state <= S_IDLE;
        end else begin
            current_state <= next_state;
        end
    end

    // 次态逻辑 (组合逻辑: 根据当前状态和输入决定下一状态)
    always @(*) begin
        next_state = current_state; // 默认保持当前状态
        case (current_state)
            S_IDLE: begin
                if (key_a_pulse) begin
                    next_state = S_RUNNING;
                end
                // 按下C在IDLE状态下保持IDLE, 复位命令将在输出逻辑中处理
            end
            S_RUNNING: begin
                if (key_b_pulse) begin
                    next_state = S_PAUSED;
                end else if (key_c_pulse) begin
                    next_state = S_IDLE;
                end else if (max_time_reached_in) begin
                    next_state = S_ALARM;
                end
            end
            S_PAUSED: begin
                if (key_a_pulse) begin
                    next_state = S_RUNNING;
                end else if (key_c_pulse) begin
                    next_state = S_IDLE;
                end
            end
            S_ALARM: begin
                if (key_c_pulse) begin
                    next_state = S_IDLE;
                end
            end
            default: begin
                next_state = S_IDLE;
            end
```

```
        endcase
    end

    // 输出逻辑 (组合逻辑: 根据当前状态确定输出)
    always @(*) begin
        // 默认输出
        timer_run_en_out    = 1'b0;
        timer_reset_cmd_out = 1'b0;
        alarm_active_out    = 1'b0;

        case (current_state)
            S_IDLE: begin
                timer_run_en_out    = 1'b0;
                timer_reset_cmd_out = 1'b1; // 在IDLE状态, 持续请求复位计时器
                alarm_active_out    = 1'b0;
            end
            S_RUNNING: begin
                timer_run_en_out    = 1'b1;
                timer_reset_cmd_out = 1'b0;
                alarm_active_out    = 1'b0;
            end
            S_PAUSED: begin
                timer_run_en_out    = 1'b0;
                timer_reset_cmd_out = 1'b0;
                alarm_active_out    = 1'b0;
            end
            S_ALARM: begin
                timer_run_en_out    = 1'b0; // 报警时停止计时
                timer_reset_cmd_out = 1'b0;
                alarm_active_out    = 1'b1;
            end
            default: begin // 安全起见, 默认行为同IDLE
                timer_run_en_out    = 1'b0;
                timer_reset_cmd_out = 1'b1;
                alarm_active_out    = 1'b0;
            end
        endcase
    end

endmodule
```

模块：display_driver

功能：

1. 将输入的 BCD 码时间值译码成七段数码管段选信号。
2. 实现 4 位七段数码管的动态扫描。

3. 驱动分钟 LED 和报警 LED。
显示顺序 (an_out[3] → an_out[0]): 秒十位, 秒个位, 百毫秒位, 十毫秒位

```
module display_driver (
    input  wire        clk,                // 系统时钟
    input  wire        reset_n,            // 异步复位 (低有效)

    input  wire [3:0]  ms_bcd_tens_in,     // 百毫秒位 (0-9 BCD)
    input  wire [3:0]  ms_bcd_ones_in,     // 十毫秒位 (0-9 BCD)
    input  wire [3:0]  sec_bcd_tens_in,    // 秒的十位 (0-5 BCD)
    input  wire [3:0]  sec_bcd_ones_in,    // 秒的个位 (0-9 BCD)
    input  wire        min_in,             // 分钟 (0-1)

    input  wire        alarm_active_in,    // 报警激活信号
    input  wire        scan_clk_enable_in, // 数码管扫描时钟使能
    input  wire        blink_clk_enable_in, // LED闪烁时钟使能

    output reg [6:0]   seg_out,            // 七段数码管段选信号 (A-G, G是seg_out[6])
    output reg [3:0]   an_out,             // 四位片选信号 (低电平有效)
    output wire        led_minute_out,     // 分钟LED输出
    output reg         led_alarm_out       // 报警LED输出
);

    // 内部信号
    reg [1:0] digit_sel_counter; // 数码管位选计数器 (00, 01, 10, 11)
    reg [3:0] current_bcd_val;   // 当前选通位要显示的BCD值
    reg blink_toggle_q;          // 用于LED闪烁的触发器

    // 分钟LED直接输出
    assign led_minute_out = min_in;

    // 报警LED闪烁逻辑
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            blink_toggle_q <= 1'b0;
            led_alarm_out  <= 1'b0;
        end else begin
            if (blink_clk_enable_in) begin
                blink_toggle_q <= ~blink_toggle_q; // 闪烁使能到来时翻转
            end

            if (alarm_active_in) begin
                led_alarm_out <= blink_toggle_q; // 报警时根据翻转信号输出
            end else begin
                led_alarm_out <= 1'b0; // 非报警状态, LED灭
            end
        end
    end

    // 数码管动态扫描 - 位选计数器
```

```
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            digit_sel_counter <= 2'b00;
        end else if (scan_clk_enable_in) begin // 扫描时钟使能到来时
            digit_sel_counter <= digit_sel_counter + 1'b1; // 00->01->10->11->00
        end
    end

    // 数码管动态扫描 - 根据位选选择要显示的BCD值和更新位选an_out
    // an_out: [3]秒十位, [2]秒个位, [1]百毫秒, [0]十毫秒
    always @(*) begin
        case (digit_sel_counter)
            2'b00: begin // 显示十毫秒位 (最右边的数码管)
                current_bcd_val = ms_bcd_ones_in;
                an_out          = 4'b0001; // 选中第0位
            end
            2'b01: begin // 显示百毫秒位
                current_bcd_val = ms_bcd_tens_in;
                an_out          = 4'b0010; // 选中第1位
            end
            2'b10: begin // 显示秒的个位
                current_bcd_val = sec_bcd_ones_in;
                an_out          = 4'b0100; // 选中第2位
            end
            2'b11: begin // 显示秒的十位 (最左边的数码管)
                current_bcd_val = sec_bcd_tens_in;
                an_out          = 4'b1000; // 选中第3位
            end
            default: begin // 理论上不会到这里
                current_bcd_val = 4'b0000; // 显示0
                an_out          = 4'b0000; // 全不选
            end
        endcase
    end

    // BCD码到七段数码管段选译码 (共阴极, 高电平点亮)
    // seg_out: [6]=G, [5]=F, [4]=E, [3]=D, [2]=C, [1]=B, [0]=A
    always @(*) begin
        case (current_bcd_val)
            4'h0: seg_out = 7'b0111111; // 0: A,B,C,D,E,F
            4'h1: seg_out = 7'b0000110; // 1: B,C
            4'h2: seg_out = 7'b1011011; // 2: A,B,D,E,G
            4'h3: seg_out = 7'b1001111; // 3: A,B,C,D,G
            4'h4: seg_out = 7'b1100110; // 4: B,C,F,G
            4'h5: seg_out = 7'b1101101; // 5: A,C,D,F,G
            4'h6: seg_out = 7'b1111101; // 6: A,C,D,E,F,G
            4'h7: seg_out = 7'b0000111; // 7: A,B,C
            4'h8: seg_out = 7'b1111111; // 8: A,B,C,D,E,F,G
            4'h9: seg_out = 7'b1101111; // 9: A,B,C,D,F,G
            default: seg_out = 7'b1000000; // G (代表错误或无效输入, 显示'-')
```

```
        endcase
    end

endmodule
```

4. 电路验证

a) TestBench

TestBench: tb_top_stopwatch

功能: 用于仿真验证 top_stopwatch 模块，测试 3 项核心功能。
仿真速度已加速 1000 倍。

```
`timescale 1ns / 1ps

module tb_top_stopwatch;

    // Testbench 内部信号
    reg tb_sys_clk_pin;
    reg tb_reset_n_pin;
    reg tb_key_a_pin;
    reg tb_key_b_pin;
    reg tb_key_c_pin;

    // 从被测模块 (DUT - Design Under Test) 输出的信号
    wire [6:0] tb_seg_pins;
    wire [3:0] tb_an_pins;
    wire       tb_led_minute_pin;
    wire       tb_led_alarm_pin;

    // 实例化被测模块 (DUT)
    top_stopwatch uut (
        .sys_clk_pin    (tb_sys_clk_pin),
        .reset_n_pin    (tb_reset_n_pin),
        .key_a_pin      (tb_key_a_pin),
        .key_b_pin      (tb_key_b_pin),
        .key_c_pin      (tb_key_c_pin),

        .seg_pins       (tb_seg_pins),
        .an_pins        (tb_an_pins),
        .led_minute_pin (tb_led_minute_pin),
        .led_alarm_pin  (tb_led_alarm_pin)
    );

    // 时钟生成 (100MHz -> 周期 10ns)
    localparam CLK_PERIOD = 10; // ns
    always begin
        tb_sys_clk_pin = 1'b0;
```

```
        #(CLK_PERIOD / 2);
        tb_sys_clk_pin = 1'b1;
        #(CLK_PERIOD / 2);
    end

    // 仿真激励序列
    initial begin
        // 0. 初始化
        tb_reset_n_pin = 1'b1;
        tb_key_a_pin   = 1'b0;
        tb_key_b_pin   = 1'b0;
        tb_key_c_pin   = 1'b0;
        $display("SIM_INFO: 仿真开始");

        // 确保进入稳定IDLE状态
        tb_reset_n_pin = 1'b0;
        #(200);
        tb_reset_n_pin = 1'b1;
        repeat(10) @(posedge tb_sys_clk_pin);

        // 第一部分：测试IDLE状态下的按键C
        $display("SIM_INFO: 测试IDLE状态下的按键C");
        $display("SIM_INFO: [Test 1] 在IDLE状态下测试按键C (应无效果)");
        tb_key_c_pin = 1'b1; #(200); tb_key_c_pin = 1'b0;
        #(100 * 1000 * CLK_PERIOD); // 等待100ms，观察计时器是否仍为0

        // 第二部分：测试正常的运行与暂停功能
        $display("SIM_INFO: 测试正常的运行与暂停功能");
        $display("SIM_INFO: [Test 2] 按下按键A开始计时 (IDLE -> RUNNING)");
        tb_key_a_pin = 1'b1; #(200); tb_key_a_pin = 1'b0;
        #(25 * 1000 * CLK_PERIOD); // 运行250ms

        $display("SIM_INFO: [Test 3] 按下按键B暂停计时 (RUNNING -> PAUSED)");
        tb_key_b_pin = 1'b1; #(200); tb_key_b_pin = 1'b0;
        #(100 * 1000 * CLK_PERIOD); // 暂停100ms，观察时间是否不变

        $display("SIM_INFO: [Test 4] 再按下按键A继续计时 (PAUSED -> RUNNING)");
        tb_key_a_pin = 1'b1; #(200); tb_key_a_pin = 1'b0;
        #(30 * 1000 * CLK_PERIOD); // 再运行300ms，总计时约550ms

        // 第三部分：测试暂停状态下的复位功能
        $display("SIM_INFO: 测试暂停状态下的复位功能");
        $display("SIM_INFO: [Test 5] 先按下按键B暂停计时 (RUNNING -> PAUSED)");
        tb_key_b_pin = 1'b1; #(200); tb_key_b_pin = 1'b0;
        #(50 * 1000 * CLK_PERIOD); // 短暂暂停

        $display("SIM_INFO: [Test 6] 然后按下按键C复位 (PAUSED -> IDLE)");
        tb_key_c_pin = 1'b1; #(200); tb_key_c_pin = 1'b0;
        #(100 * 1000 * CLK_PERIOD); // 等待观察复位效果

```

```
        // 第四部分：测试运行状态下的复位功能
        $display("SIM_INFO: 测试运行状态下的复位功能");
        $display("SIM_INFO: [Test 7] 再次按下按键A开始计时 (IDLE -> RUNNING)");
        tb_key_a_pin = 1'b1; #(200); tb_key_a_pin = 1'b0;
        #(50 * 1000 * CLK_PERIOD); // 运行500ms

        $display("SIM_INFO: [Test 8] 然后按下按键C复位 (RUNNING -> IDLE)");
        tb_key_c_pin = 1'b1; #(200); tb_key_c_pin = 1'b0;
        #(100 * 1000 * CLK_PERIOD); // 等待观察复位效果

        // 第五部分：测试报警功能与该状态下的复位
        $display("SIM_INFO: 测试报警功能与该状态下的复位");
        $display("SIM_INFO: [Test 9 & 10] 启动计时器至最大时间 (IDLE -> RUNNING -> ALARM)");
        tb_key_a_pin = 1'b1; #(200); tb_key_a_pin = 1'b0; // [Test 9]

        // [Test 10] 等待到达最大时间 1:59:99
        #(11998 * 1000 * CLK_PERIOD); // 先走到最大时间前一个tick
        $display("SIM_INFO: 接近最大时间");
        #(1 * 1000 * CLK_PERIOD);   // 最后一个tick，此时应该达到最大时间

        #(200 * 1000 * CLK_PERIOD); // 维持报警状态200ms
        $display("SIM_INFO: 维持报警状态200ms");

        $display("SIM_INFO: [Test 11] 按下按键C复位 (ALARM -> IDLE)");
        tb_key_c_pin = 1'b1; #(200); tb_key_c_pin = 1'b0;
        #(100 * 1000 * CLK_PERIOD);

        $display("SIM_INFO: 仿真结束");
        $finish; // 结束仿真
    end

endmodule
```

b) 仿真结果

在 Testbench 当中，我们希望验证短跑计时器的 3 项核心功能：

- 正常计时功能
- 四种状态下的复位功能
- 报警功能

通过合理的编排，我们将这 3 项核心功能的测试融入到了依次进行的 6 个阶段当中，以下是各个阶段的说明以及对应的仿真结果：

1) IDLE 状态下的复位功能

按下按键 C 复位，验证 IDLE 状态下的复位功能。预期结果是计时仍为 0，实际结果与预期相符：

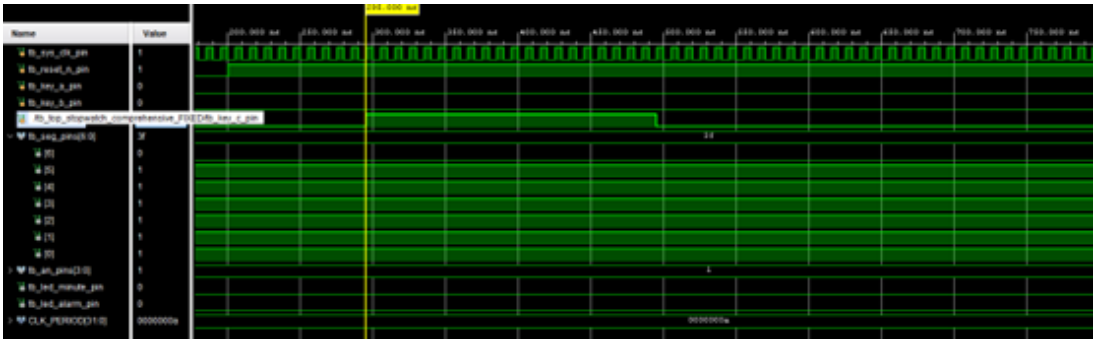


图 4　IDLE 状态下的复位功能

2) 正常计时功能当中的运行与暂停功能

- **Step.1)** 按下按键 A 开始计时，验证 IDLE->RUNNING。预期结果为计时由零开始逐渐增加，实际结果与预期相符，限于篇幅无法在此处展现具体数值，但可以看出数字正在变化：

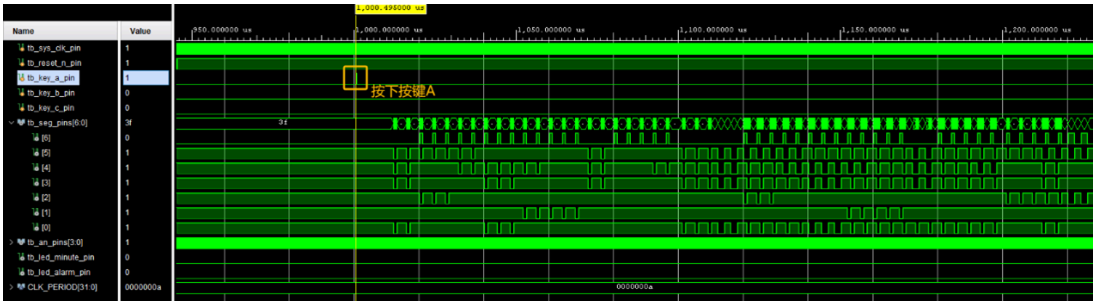


图 5　正常计时功能当中的运行与暂停功能

- **Step.2)** 再按下按键 B 暂停计时，验证 RUNNING->PAUSED。预期结果为计时开始保持不变，实际结果与预期相符：

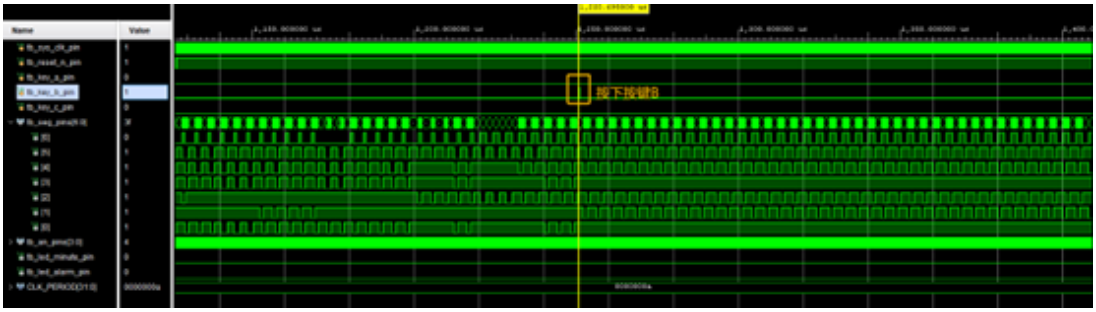


图 6　正常计时功能当中的运行与暂停功能

- **Step.3)** 此时再按下按键 A 开始计时，验证 PAUSED->RUNNING。预期结果为计时继续增加，实际结果与预期相符：

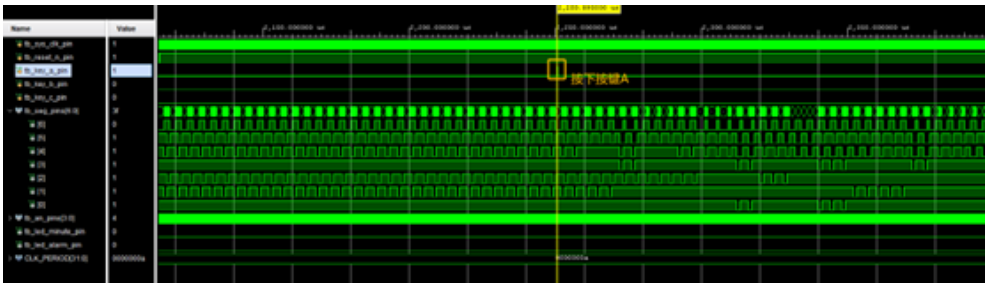


图 7　正常计时功能当中的运行与暂停功能

3) 暂停状态下的复位功能

- **Step.1)** 首先按下按键 B 暂停计时。该操作已得到验证，不再过多赘述。
- **Step.2)** 随后按下按键 C 复位，验证 PAUSED->IDLE。预期结果为计时清零，实际结果与预期相符：

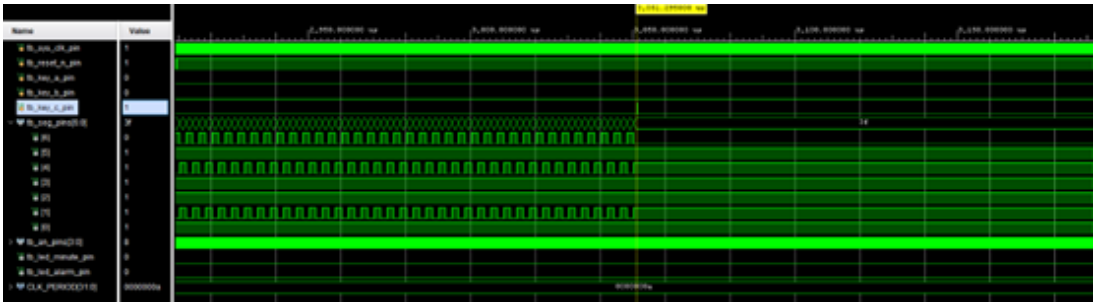


图 8　暂停状态下的复位功能

4) 运行状态下的复位功能

- **Step.1)** 首先按下按键 A 开始计时。该操作已得到验证，不再过多赘述。
- **Step.2)** 随后按下按键 C 复位，验证 RUNNING->IDLE。预期结果为计时清零，实际结果与预期相符：

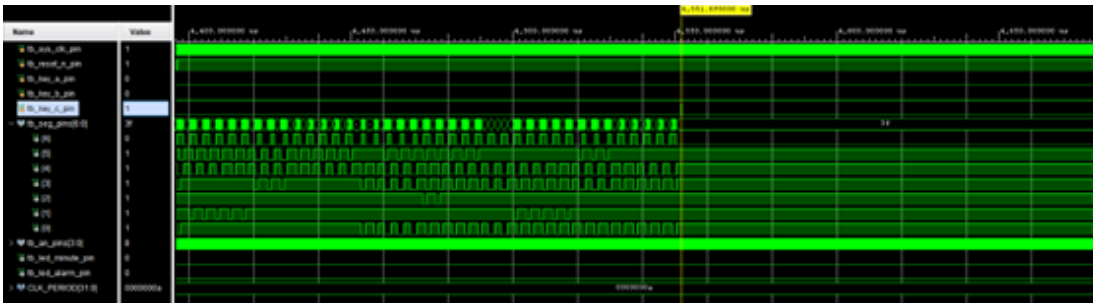


图 9　运行状态下的复位功能

5) 正常计时功能当中的“进位”功能

由于短跑计时器对于分钟的处理与秒、毫秒有所不同，因此此处单独验证。按下按键 A 开始计时，并等待时间到达一分钟。预期结果为 `tb_led_minute_pin` 跳变为 1，实际结果与预期相符：

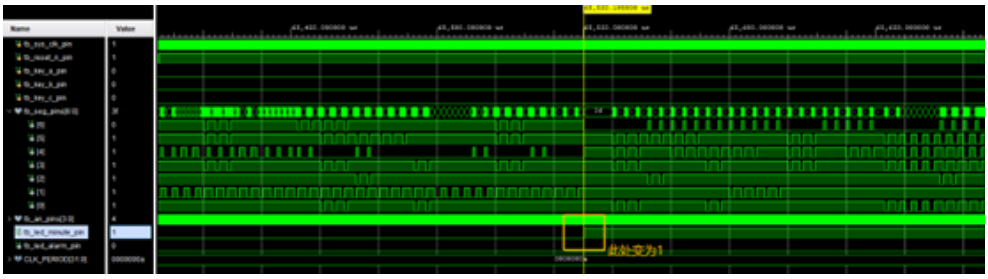


图 10　正常计时功能当中的“进位”功能

6) 报警功能与该状态下的复位

- **Step.1)** 等待计时到达两分钟，验证 RUNNING->ALARM。预期结果为计时清零，同时 `tb_led_alarm_pin` 数值在 0、1 之间反复跳变(即 LED 灯闪烁报警)，实际结果与预期相符：

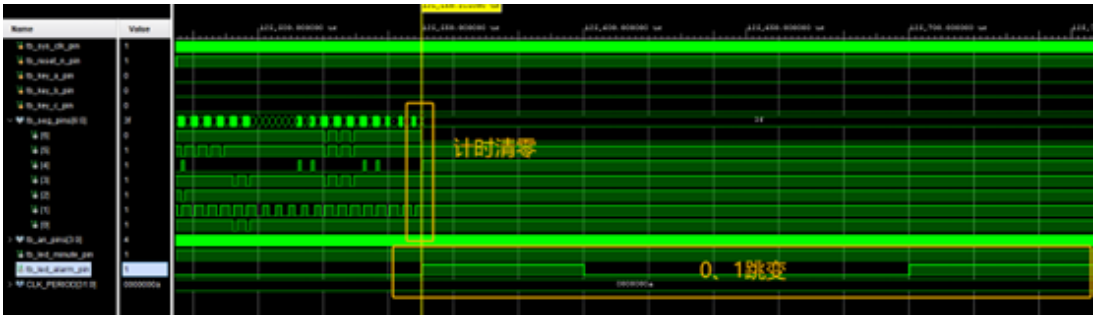


图 11　报警功能与该状态下的复位

- **Step.2)** 按下按键 C 复位，验证 ALARM->IDLE。预期结果为计时、 `tb_led_minute_pin` 、`tb_led_alarm_pin` 均清零，计时器回归初始状态，实际结果与预期相符：

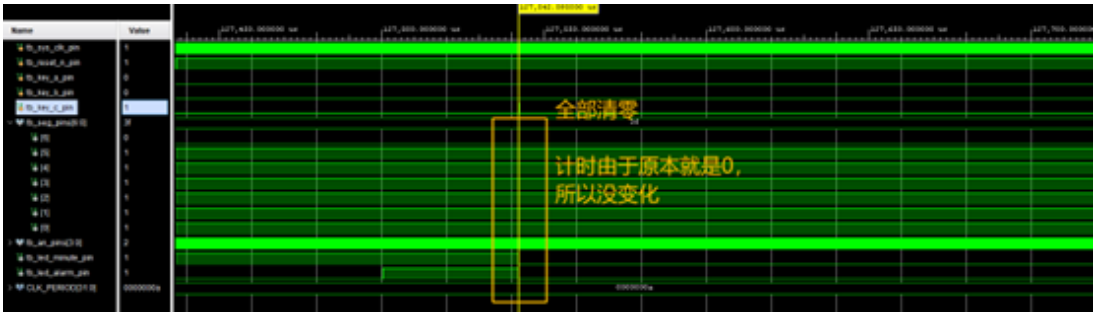


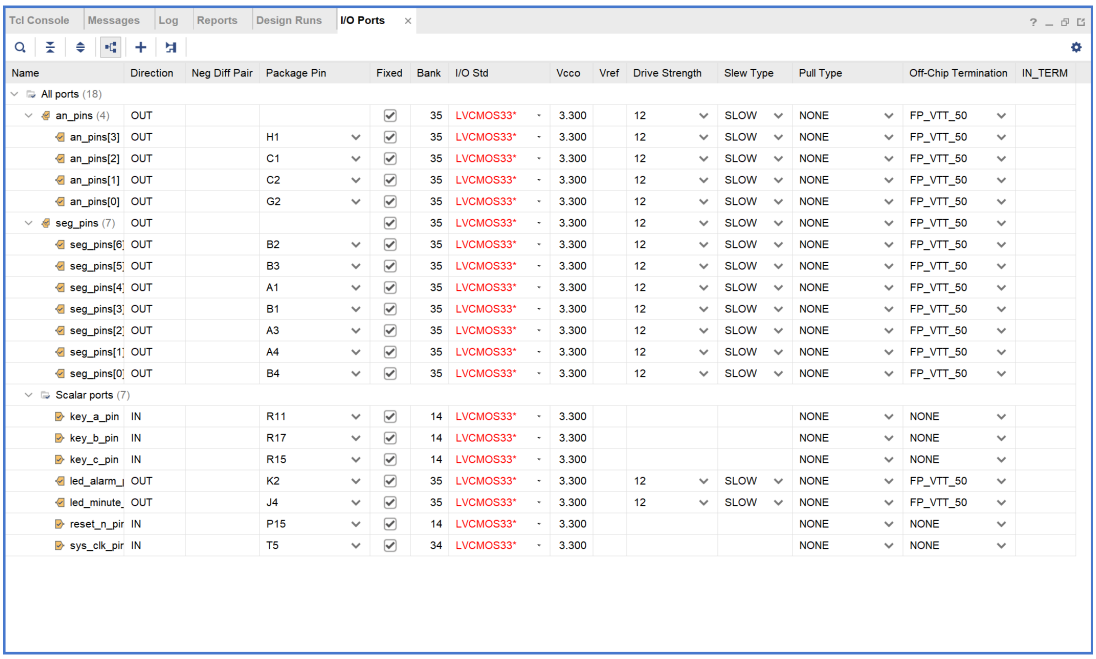
图 12　报警功能与该状态下的复位

总之，我们设计的短跑计时器的 3 项核心功能在 Testbench 当中得到了很好的验证，证明了设计思路与 Verilog 代码的有效性。

5. 电路上板

a) 管脚配置

结合 EES-338 开发板的管脚分配说明书，以及源代码输入输出变量设置，可以得到如下图中表格的管脚配置：



Tcl Console | Messages | Log | Reports | Design Runs | I/O Ports

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
All ports (18)													
an_pins (4)	OUT			☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
an_pins[3]	OUT		H1	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
an_pins[2]	OUT		C1	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
an_pins[1]	OUT		C2	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
an_pins[0]	OUT		G2	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins (7)	OUT			☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[6]	OUT		B2	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[5]	OUT		B3	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[4]	OUT		A1	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[3]	OUT		B1	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[2]	OUT		A3	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[1]	OUT		A4	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
seg_pins[0]	OUT		B4	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
Scalar ports (7)													
key_a_pin	IN		R11	☑	14	LVCMOS33*	3.300				NONE	NONE	
key_b_pin	IN		R17	☑	14	LVCMOS33*	3.300				NONE	NONE	
key_c_pin	IN		R15	☑	14	LVCMOS33*	3.300				NONE	NONE	
led_alarm_	OUT		K2	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
led_minute_	OUT		J4	☑	35	LVCMOS33*	3.300		12	SLOW	NONE	FP_VTT_50	
reset_n_pir	IN		P15	☑	14	LVCMOS33*	3.300				NONE	NONE	
sys_clk_pir	IN		T5	☑	34	LVCMOS33*	3.300				NONE	NONE	

图 13　短跑计时器的管脚配置

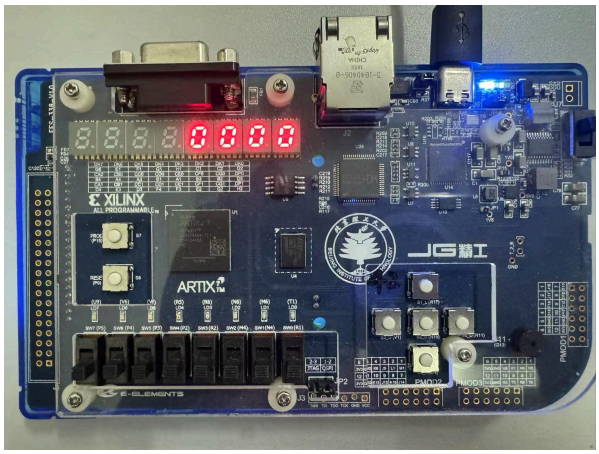
b) 上板情况

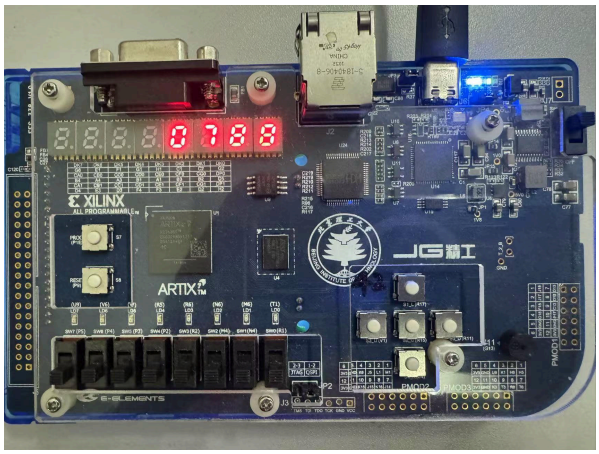
在 Vivado 中依次完成管脚配置、生成比特流文件、开发板连接和代码上板。

上板之后，可以观察到初始状态下，右侧四位数码管亮起，显示“0000”。

当按下开始键时，计时器进入计时状态，数码管开始显示实时计时的秒数和毫秒数。随着时间推移，当计时达到 1 分钟时，分钟指示灯自动点亮，提示已经计时超过 1 分钟。当计时达到 1 分 59 秒 99 毫秒时，系统触发报警状态，报警指示灯开始闪烁，同时计时器停止计时。

在计时过程中，用户可以随时通过暂停键控制计时器的运行状态。按下暂停键后，计时器暂停计时，数码管显示的数字保持不变。此时再次按下开始键，计时器将继续从暂停时刻继续计时。无论是在暂停状态还是计时状态，按下复位键都会使计时器回到初始状态：分钟指示灯和报警指示灯熄灭，数码管显示归零为“0000”。





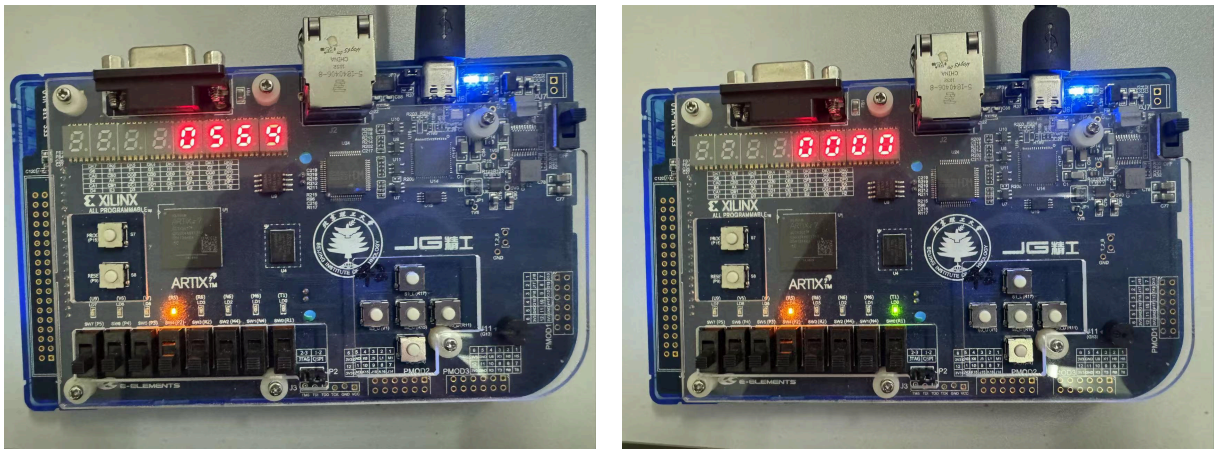


图 14　上板情况

具体操作演示请参考录制的视频。

6. 实验心得

左逸龙

本次短跑计时器实验是一次完整的数字电路设计与验证实践。我主要负责了电路的 Verilog 代码实现与仿真验证工作，而陈墨霏同学则负责了电路的概要设计和最终的板级烧录与验证。通过这次合作，我对数字逻辑设计的全流程有了更深刻的理解。

在电路实现阶段，我首先面临的挑战是数码管的显示机制。我初次了解到数码管并非静态显示，而是通过动态扫描方式进行分时点亮。这要求我在 `display_driver` 模块中精确控制位选和段选信号的时序。分模块设计理念在这一阶段展现了其优势。当我发现数码管显示异常时，能够迅速锁定 `display_driver` 模块进行排查，而无需检查整个系统代码。

代码编写过程中，我对时序逻辑和并行处理的理解得到了深化。例如，对非阻塞赋值（`<=`）的认识从语法层面提升到了其背后硬件寄存器并行更新的特性。这帮助我更好地预判信号在仿真中的传播行为。在设计 `state_controller` 模块时，确保状态机所有可能的跳转路径都得到覆盖是一大难点。我初期曾遗漏了从运行状态直接复位到空闲状态的路径，导致仿真中计时器无法立即重置，这强调了状态转移完整性的重要性。此外，在 `clk_divider_counter` 模块中，我通过设计多级分频器，将 100 MHz 的系统时钟转换为 10 ms 的计时脉冲以及数码管扫描和报警所需的辅助时钟。这个过程使我直观地体会到了数字系统中不同时间尺度的生成与转换。同时，在编写诸如 `reg counter` 这样的计数器时，我开始尝试将其映射为实际的触发器阵列，这种硬件思维有助于写出更符合可综合要求的代码。

仿真验证是本次实验中我投入精力最多的部分。起初，我将 Testbench 编写视为一项单纯的作业要求，但很快，它的实际价值就得以体现。在初次仿真时，我发现按键激励未能在波形图中正确显示。经过排查，我首先定位到 Testbench 中 task 参数传递的作用域问题，理解了直接通过 inout reg 参数修改信号电平的局限性。随后，尽管修正了激励方式，按键脉冲在宏观波形图中依然不可见，这促使我认识到 Vivado 仿真器波形优化机制的存在，并通过延长按键脉冲的持续时间来解决显示问题。最关键的一次调试经历是，我在 clk_divider_counter 模块中启用了仿真加速模式，却遗漏了在 button_interface_debounce 模块中同步调整消抖计时参数。这意味着，加速后的仿真按键脉冲持续时间远不足以满足原始的消抖计数要求，导致按键事件始终无法被识别。这个问题的发现让我深刻理解了跨模块参数同步的重要性，以及调试时需要全面检查所有相关时序参数的必要性。通过这些具体的调试过程，我的思维模式从单纯的代码编写转向了“先怀疑测试方法，再深入设计逻辑”的分析思路。同时，我对波形图的“阅读”能力也得到了锻炼，学会了如何将波形上的高低电平变化与代码中的逻辑状态和变量值准确对应。

在与组员的协作中，我负责的仿真验证与组员负责的板级烧录形成了闭环。当组员在板级测试中反馈数码管显示异常时，我基于仿真中的经验，迅速检查了 display_driver 模块中段选和位选信号的极性设置，最终发现存在一处高电平有效与低电平有效的混淆，修正后板载显示恢复正常。此外，我们也共同认识到，即使仿真结果完美，实际烧录到板上仍可能遇到问题，这其中，引脚约束的准确性尤为关键。我们初期在进行引脚约束时，由于未充分对照开发板手册，导致频繁出现接线错误，这凸显了在设计初期就进行细致引脚规划的重要性。

总的来说，本次实验最大的收获在于，我掌握了“仿真驱动开发”的流程。即在编写核心功能代码之前，优先构建并完善验证环境，用仿真验证设计的正确性，这一实践显著提升了开发效率，并减少了后期调试的复杂性。这次完整的项目经历，也为我未来学习《计算机组成原理》中控制单元的状态机设计、以及《操作系统》中进程调度算法提供了具象的硬件实践基础。对严谨的硬件调试思维的培养，也将对未来可能从事的芯片设计或嵌入式开发岗位产生积极影响。如果让我重新开始这个实验，我将从一开始就投入更多精力完善仿真测试用例，特别是覆盖各种边界条件和异常路径。我相信，这种前期的验证投入能够极大程度地减少后续百分之八十的调试时间，进一步提高开发效率。

陈墨霏

在本次短跑计时器实验中，我主要负责了电路的概要设计和最终的板级验证工作。通过与左逸龙同学的密切配合，我不仅深入理解了数字电路的设计流程，更在实践中体会到了硬件设计从理论到实现的完整过程。

在电路设计过程中，我深刻体会到了模块化设计理念的重要性。整个短跑计时器被合理地划分为 4 个功能明确的子模块和 1 个顶层综合模块（`top_stopwatch`），这种模块化的设计不仅使代码结构清晰，也大大提高了开发效率和代码的可维护性。通过这次实践，我对时序逻辑电路的设计有了更深层次的理解。在理论学习阶段，我们主要关注状态机的状态转移和输出逻辑，但在实际下板开发时，我发现这仅仅是整个设计过程中的一小部分。例如，在实现 `state_controller` 模块时，我们不仅要考虑状态机的正确性，还要考虑如何将其与实际的硬件设备进行交互。这让我意识到，硬件设计不仅仅是逻辑设计的实现，更是对硬件特性的深入理解和应用。在开发过程中，我们遇到了一个典型的硬件特性问题：数码管显示异常。通过和左逸龙同学的细致排查，我们发现问题的根源在于对位选信号有效电平的错误理解。我们错误地将位选信号认定为负有效，导致本该点亮的一位数码管不被点亮，而另外三位却被点亮。这个问题的解决过程让我深刻认识到，在实际硬件开发中，对硬件特性的准确理解是多么重要。此外，通过这次实验，我也深入理解了状态机设计中的关键概念，包括状态转移表、状态机图等工具的使用。这些工具不仅帮助我们清晰地表达设计意图，也为后续的代码实现和调试提供了重要参考。总的来说，这次实验让我对数字电路设计有了更全面的认识，从理论到实践，从逻辑设计到硬件实现，每一个环节都让我受益匪浅。

在本次实验中，我还负责了电路的上板验证工作。通过实践，我系统地掌握了 Vivado 开发环境下的完整上板流程，包括引脚约束配置、约束文件生成、综合实现、比特流文件生成以及最终的开发板烧录等环节。虽然这些操作步骤相对固定，投入的精力不及电路设计阶段，但这个过程对细节的把控要求极高。在初期上板过程中，由于对引脚约束的疏忽，我曾多次出现接线错误，导致系统无法正常工作。例如，在一次调试中，我将七段数码管的位选信号高低位顺序颠倒，直接导致了显示异常。这些经历让我深刻认识到，在硬件开发中，严谨细致的工作态度与扎实的技术功底同样重要。这种对细节的重视不仅适用于实验环境，更是实际工程开发中不可或缺的职业素养。

通过与左逸龙同学的密切合作，我们不仅顺利完成了实验本身的设计与实现，还共同完成了后续的视频拍摄、报告撰写等收尾工作。这次合作经历让我受益匪浅，特别是在技术层面，我从

队友那里学习到了许多宝贵的开发思路和实用工具。这些收获不仅对本次实验大有裨益，也将对我未来的学习和工作产生积极影响。在此，我向我的队友表示由衷的感谢。

通过本次短跑计时器的设计与实现，我不仅深化了对《数字逻辑》课程中状态机设计、时序逻辑等核心概念的理解，更重要的是建立起了对数字系统设计的整体认知框架。这些实践经验为我后续学习《计算机组成原理》中的控制单元设计、《操作系统》中的进程调度等课程奠定了坚实的硬件基础。同时，在实验过程中培养的严谨的硬件调试思维和模块化设计理念，也将对我未来可能从事的芯片设计、嵌入式开发等职业领域产生深远影响。