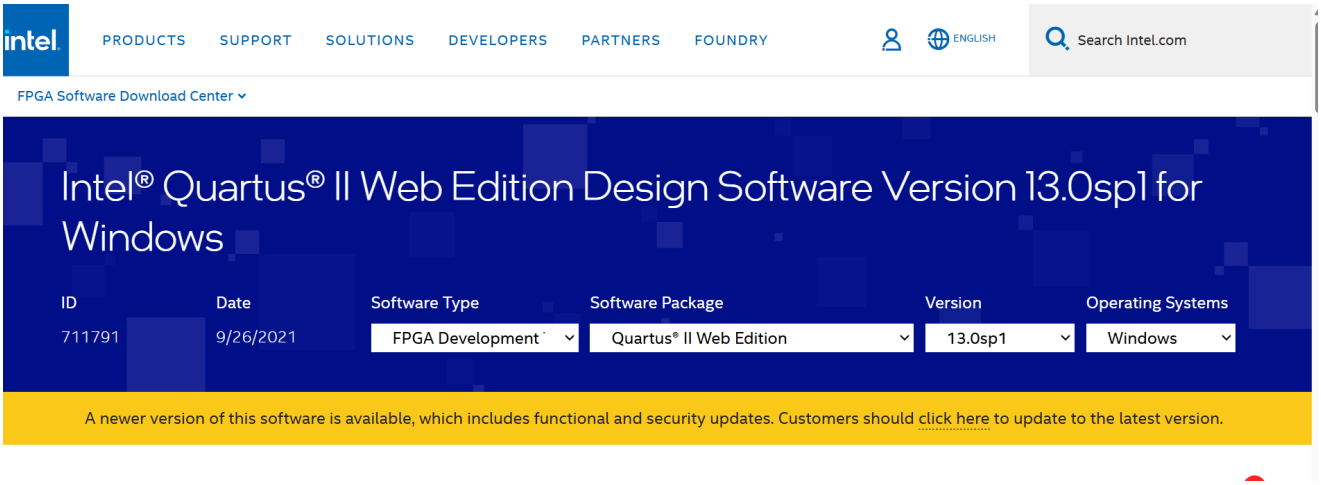
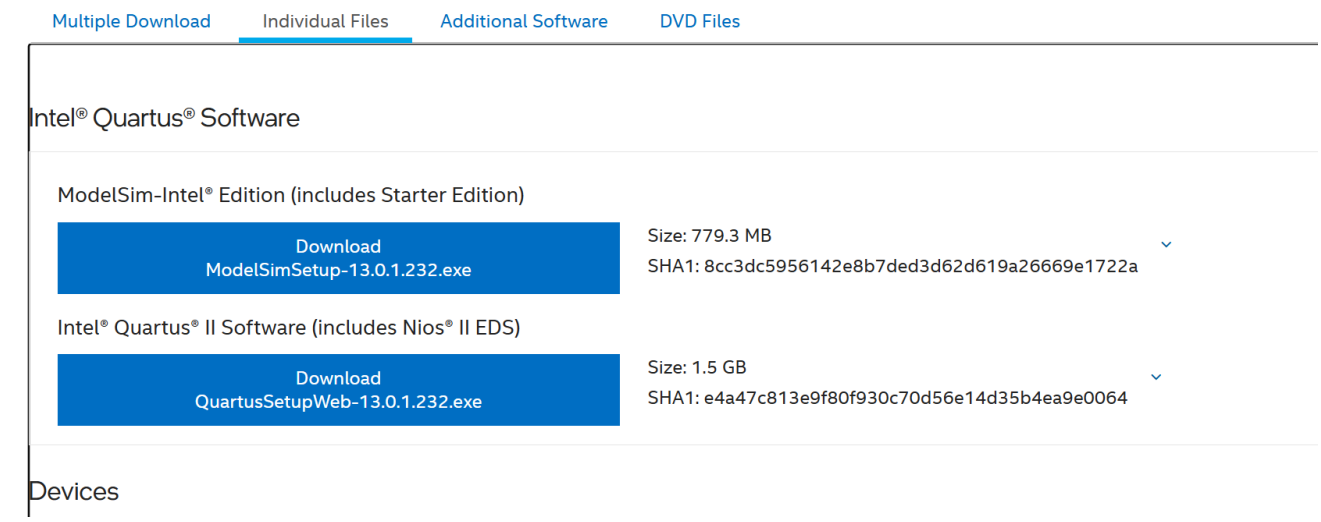
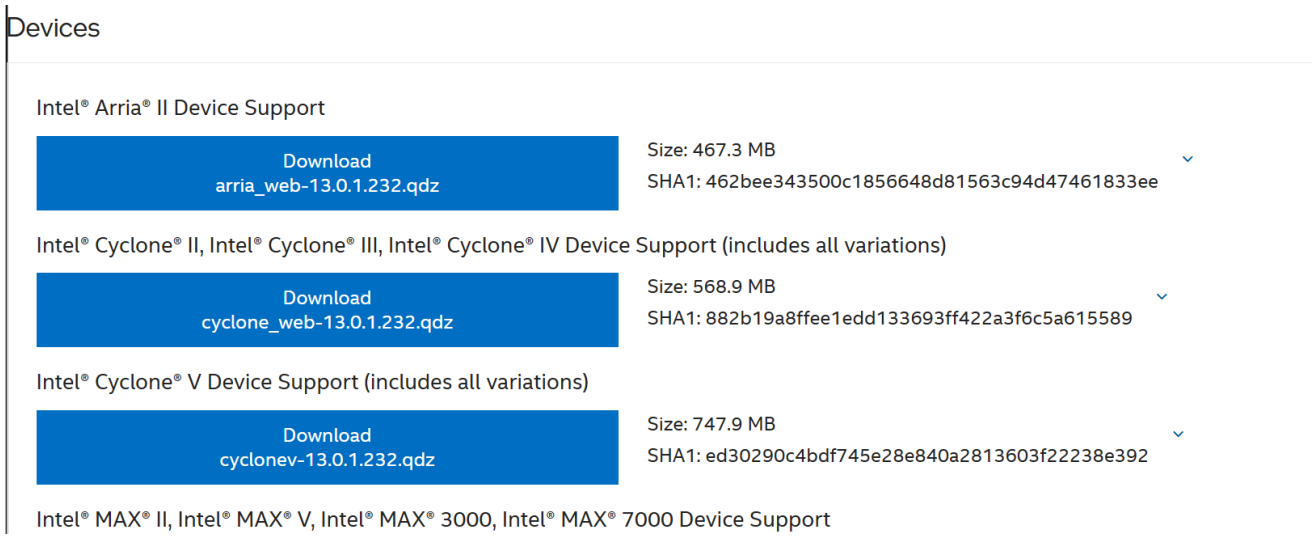
# Quartus II 13.0 SP1 & ModelSim 电子钟文档

## 一、软件安装(windows环境)

https://www.intel.com/content/www/us/en/software-kit/711791/intel-quartus-ii-web-edition-design-software-version-13-0sp1-for-windows.html






解压到自己选择的同一文件夹下（建议不要装到C盘），运行exe安装软件，一路next。

cd F:/FPGA/quartus/bin64/simulation/modelsim

# 二、仿真设计

## 1.功能实现：

### *1.Time & Clocking*

FPGA开发板提供的是一个非常高的频率时钟（50MHz），但我们的秒钟需要每秒跳动一次，因此需要一个分频器，分频器的原理是对高速时钟的跳动次数计数，例如数到50M次的时候（现实中刚好过去了一秒），产生一个单周期的脉冲信号，这个脉冲就是驱动整个时钟系统前进的心跳

实现模块：`clk_divider.v`

```
// src/clk_divider.v
// 功能：不生成一个独立时钟，而是生成一个1Hz的单周期脉冲使能信号

module clk_divider (
    input   wire    clk_in,
    input   wire    rst,
    output  reg     clk_1hz_en // 输出端口命名为 clk_1hz_en
);
    // 使用参数方便在仿真和硬件之间切换
    parameter SIMULATION = 1; // 仿真设为1，硬件设为0

    // 根据模式定义计数器最大值
    localparam CNT_MAX = (SIMULATION == 1) ? 50 : 50_000_000;

    reg [$clog2(CNT_MAX)-1:0] counter;

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

```
        if (rst) begin
            counter <= 0;
            clk_1hz_en <= 1'b0;
        end else begin
            if (counter == CNT_MAX - 1) begin
                counter <= 0;
                clk_1hz_en <= 1'b1; // 在计数达到最大值时，产生一个单周期的使能脉
冲
            end else begin
                counter <= counter + 1;
                clk_1hz_en <= 1'b0; // 在其他时间，使能信号保持为低电平
            end
        end
    end
endmodule
```

- parameter：是一个“**可配置的**”常量。它的值可以在模块实例化时被外部修改，或者在编译时通过 defparam 语句覆盖。它就像一个模块的“设置选项”。
- localparam：是一个“**不可更改的**”常量。它的值在模块内部定义后就固定了，不能被外部以任何方式修改。它就像模块内部为了方便计算而定义的私有常量。

`localparam CNT_MAX = (SIMULATION == 1) ? 50 : 50_000_000;`语句中包含一个三元操作符，首先计算括号里的条件是否成立，若成立则将问号后的50赋值给CNT_MAX，不成立则用50 000 000赋值。

---

reg 是Verilog中两种主要的数据类型之一（另一种是 wire）。它用于声明一个寄存器类型的变量。

核心特性:

- 状态保持: reg 型变量具有记忆能力。一旦被赋值，它会保持这个值不变，直到下一次被显式地赋予新值。
- 赋值上下文: reg 型变量只能在过程块 (Procedural Blocks) 中被赋值，即 always 块或 initial 块。
- 硬件映射: 在可综合的代码中，如果一个 reg 在时序逻辑块 (always @(posedge clk)) 中被赋值，它通常会被综合成一个D触发器 (D-Flip-Flop) 或一组触发器，这是构成所有时序电路（如计数器、状态机）的基础物理单元。

[...] (方括号)是向量范围声明符 (Vector Range Specifier)

方括号用于声明一个多位的向量 (Vector)，而不是一个单比特的标量 (Scalar)。它定义了这个向量的位宽和索引范围。
标准格式: [ <MSB_index> : <LSB_index> ]

<MSB_index>: 最高有效位 (Most Significant Bit) 的索引号。
<LSB_index>: 最低有效位 (Least Significant Bit) 的索引号。

索引号可以是任何整数常量或在编译时可确定的表达式。

在本句中的作用: 告诉编译器，counter 不是一个1位的 reg，而是一个由多位 reg 组成的数组，其位宽由方括号内的表达式动态决定。***注意：位宽=索引的最高位减最低位的绝对值+1***

$clog2()是一个系统函数 (System Function)

功能: 计算并返回一个整数的**“以2为底的天花板对数 (Ceiling of Logarithm Base 2)”。

通俗地讲，它回答了这样一个问题：“要无符号地表示从0到 N-1 的所有整数，***注意：位宽=索引的最高位减最低位的绝对值+1***至少需要多少个比特位？” $clog2(N) 就能给出这个位数。

举例:
要表示0-7（共8个数），需要 log2(8) = 3 位。
$clog2(8)$返回3。要表示$0-9$（共10个数），需要$log2(10) \approx 3.32$，向上取整为$4$位。clog2(10) 返回 4。
在本句中的作用: 自动计算为了能让计数器数到 CNT_MAX - 1 所需的最小位宽。如果 CNT_MAX 是 50，那么计数器需要表示0-49共50个数字，$clog2(50) 返回 6，即需要6位。

---

> ***将所有语法元素组合起来，`reg [$clog2(CNT_MAX)-1:0] counter;` 这句话的完整含义是：***
>
> ***“声明 (Declare) 一个名为 counter 的寄存器 (reg) 类型的向量 (vector)。这个向量的位宽是动态计算出来的：它等于向上取整的log2(CNT_MAX)。其最高位的索引是 ($clog2(CNT_MAX) - 1)，最低位的索引是 0。”***

---

`1'b0` 是一种Verilog的常量数值表示法，它能精确地告诉编译器一个数字的位宽 (width)、基数 (base) 和 值 (value)

`1'b0` 的完整、精确的含义是：“这是一个1位的、二进制的常量，其值为0。”在电路中，它通常就代表了逻辑低电平 (Logic Low)。

`always @(posedge clk_in or posedge rst)` 这句的含义是“永远监听 clk_in 信号的上升瞬间，或者监听 rst 信号的上升瞬间。”

- always @(...): 这是Verilog中定义时序逻辑的语句。always 表示这个逻辑块会不断地被评估，@ 符号表示“在...事件发生时”。
- posedge: 关键字，意思是“上升沿 (Positive Edge)”，也就是信号从0变到1的那个精确瞬间。
- or: 关键字，表示敏感列表中的任何一个事件发生，都会触发代码块的执行。

语句内部的条件判断逻辑比较易懂，在这里不用解释。

---

***2.Storage & Counting：***

有了“滴答”信号，我们现在需要一个能够记住当前时间并根据“滴答”声更新自己的模块。
功能：实现时、分、秒的存储和进位

我们需要三个独立的寄存器分别存储小时、分钟和秒的数值。所有这些寄存器都由同一个高速主时钟 `clk` 驱动，但它们只在接收到来自第一层的 `clk_1hz_en` “滴答”信号时，才执行加一操作。

- 秒寄存器 (sec)：每个“滴答”都加一。当它从59变为0时，需要向上层（分钟）发出一个进位信号。
- 分寄存器 (min)：只在接收到秒的进位信号时才加一。当它从59变为0时，向上层（小时）发出进位信号。
- 时寄存器 (hour)：只在接收到分的进位信号时才加一。当它从23变为0时，完成一天循环。

此外，这个模块还需要一个“加载”功能，以便在用户调整时间时，能用外部传入的值覆盖当前的计数值。

实现模块: `time_counter.v`:

```
// src/time_counter.v
// 功能：在统一的主时钟域下工作，通过时钟使能信号控制计数

module time_counter (
    input   wire        clk,            // 使用主时钟 clk
    input   wire        clk_1hz_en,     // 使用1Hz使能信号
    input   wire        rst,
    input   wire        time_count_en,
    input   wire        load_en,
    input   wire [4:0]  hour_in,
    input   wire [5:0]  min_in,
    output  reg [5:0]   sec,
    output  reg [5:0]   min,
    output  reg [4:0]   hour
);
    //  【改动】将所有逻辑合并到一个由主时钟 clk 驱动的 always 块中
    // 这样可以确保 load_en 信号能被正确采样
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sec <= 6'd0;
            min <= 6'd0;
            hour <= 5'd0; //小时只需要数到23所以五位数就够用
        end
```

```
        // 加载操作具有高优先级，只要 load_en 有效就执行
        else if (load_en) begin
            sec <= 6'd0;        // 调整时间时，秒数清零
            min <= min_in;
            hour <= hour_in;
        end
        // 只有在1Hz使能信号有效且计数使能时，才进行计数
        else if (time_count_en && clk_1hz_en) begin
            if (sec == 6'd59) begin
                sec <= 6'd0;
                if (min == 6'd59) begin
                    min <= 6'd0;
                    if (hour == 5'd23) begin
                        hour <= 5'd0;
                    end else begin
                        hour <= hour + 1;
                    end
                end else begin
                    min <= min + 1;
                end
            end else begin
                sec <= sec + 1;
            end
        end
    end
endmodule
```

在这里能注意到有一个目前的两份代码里还没有定义的信号 `time_count_en` (计时使能)，会在 `clock_controller.v` 里定义，由状态机决定它的值，这么设计是为了在进入校准调时模式或闹钟设置模式的时候暂停时钟。

---

### *3.Decoding & Display*

有了时间数据，我们还需要把它变成人能看懂的样子。
功能：将二进制时间数据转换为七段数码管信号。

- **逻辑和原理**:
  这个过程分两步：
  1. **数据翻译 (Decoding)**: 一个数码管只能显示0-9的数字。我们需要一个“翻译官”模块，它接收一个4位的二进制数（例如 4'b0111 代表7），然后输出7个控制信号，告诉数码管的 a,b,c,d,e,f,g 七个段哪些该亮，哪些该灭，从而拼凑出数字“7”的形状。这是一个纯粹的**组合逻辑**，输入变，输出立刻变。
  2. **动态扫描 (Scanning)**: 我们有6个数码管，但为了节省FPGA的引脚，我们不希望同时驱动它们。取而代之的是，我们以极快的速度（人眼无法察觉的频率）轮流点亮每一个数码管。例如，在第一个瞬间点亮秒的个位，第二个瞬间点亮秒的十位，...，第六

个瞬间点亮时的十位，然后又回到秒的个位。由于视觉暂留效应，人眼看到的就是6个数字稳定地亮在那里。

- **实现模块**: `display_decoder.v` (译码器), `display_scanner.v` (扫描控制器)

```
// display_decoder.v
// 功能：将一个4位的BCD码数字（0-9）翻译成七段数码管的段选信号
// 假设开发板上的数码管是“共阴极”类型，即高电平（1）点亮LED段

module display_decoder (
    input   wire [3:0]  num_in,     // 输入的数字 (0-9)
    output  reg  [6:0]  seg_out     // 输出的七段码 (对应 g,f,e,d,c,b,a)
);

    // 这是一个纯组合逻辑电路，所以使用 always @(*)
    // 意味着只要输入 num_in 发生任何变化，就立刻重新计算输出
    always @(*) begin
        case(num_in)
            4'd0: seg_out = 7'b0111111; // 显示 "0"
            4'd1: seg_out = 7'b0000110; // 显示 "1"
            4'd2: seg_out = 7'b1011011; // 显示 "2"
            4'd3: seg_out = 7'b1001111; // 显示 "3"
            4'd4: seg_out = 7'b1100110; // 显示 "4"
            4'd5: seg_out = 7'b1101101; // 显示 "5"
            4'd6: seg_out = 7'b1111101; // 显示 "6"
            4'd7: seg_out = 7'b0000111; // 显示 "7"
            4'd8: seg_out = 7'b1111111; // 显示 "8"
            4'd9: seg_out = 7'b1101111; // 显示 "9"
            default: seg_out = 7'b0000000; // 如果输入的不是0-9，则全灭
        endcase
    end

endmodule
```

***再次注意：位宽=索引的最高位减最低位的绝对值+1***，所以是用[3:0]来表示一个四位的二进制数，用来给输入的十个数字0到9编码

Q：为什么输入数字用wire，输出的七段码用reg呢？

A：简单来讲是因为num_in (输入) 用 wire，是因为它在 display_decoder 模块中从未被赋值，它只是一个被动接收数据的“电线”。seg_out (输出) 用 reg，是因为它在一个 always 块内部被赋值了。在Verilog的语法规定中，凡是在 always 块（或 initial 块）内部被赋值的变量，必须被声明为 reg 类型。

---

> ***1. wire (电线/网络)***

**特性**: wire 类型的变量不能存储值。它就像一根真实的电线，它的值完全由驱动它的东西来决定。如果没有东西驱动它，它的值就是高阻态 (z)。

**赋值方式**: 只能通过连续赋值 (Continuous Assignment)来驱动，也就是使用 assign 语句。assign 语句描述的是一种**持续的、并行的**连接关系，而不是一个事件。

**用途**:
模块的输入端口 (input 和输入输出端口 (inout) 默认就是 wire 类型。
用来连接模块实例之间的端口。
作为 assign 语句的输出目标。

***2. reg (寄存器/变量)***

**特性**: reg 类型的变量**可以存储值**。它就像一个变量，在两次赋值之间，它会**保持**上一次被赋予的值。
**赋值方式**: 只能在**过程块 (Procedural Blocks)** 内部被赋值，也就是在 always 块或 initial 块里。
**注意：**声明为 reg 并不意味着它一定会被综合成一个物理的寄存器（触发器）这取决于 always 块的写法。

---

```
// src/display_scanner.v

module display_scanner (
    input   wire        clk,
    input   wire        rst,
    input   wire [4:0]  hour,
    input   wire [5:0]  min,
    input   wire [5:0]  sec,
    input   wire [2:0]  display_mode, // Receives mode from controller
    output  reg  [3:0]  num_to_decode,
    output  reg  [5:0]  digit_sel
);
    parameter SIMULATION = 0; // 仿真设为1, 硬件设为0

    // State definitions (must match the controller)
    parameter S_ADJ_H     = 3'd1;
    parameter S_ADJ_M     = 3'd2;
    parameter S_ALARM_H   = 3'd3;
    parameter S_ALARM_M   = 3'd4;

    // Scan enable signal generation
    localparam SCAN_CNT_MAX = (SIMULATION == 1) ? 4 : 50_000; // ~100ns
for sim, ~1ms for hardware
    reg [$clog2(SCAN_CNT_MAX)-1:0] scan_counter;
    wire scan_en = (scan_counter == SCAN_CNT_MAX - 1);
```

```
    always @(posedge clk or posedge rst) begin
        if (rst)
            scan_counter <= 0;
        else if (scan_en)
            scan_counter <= 0;
        else
            scan_counter <= scan_counter + 1;
    end

    // BCD conversion 通过除法和取余运算将两位数的时间拆分为十位和个位
    wire [3:0] hour1 = hour / 10; wire [3:0] hour0 = hour % 10;
    wire [3:0] min1  = min / 10;  wire [3:0] min0  = min % 10;
    wire [3:0] sec1  = sec / 10;  wire [3:0] sec0  = sec % 10;

    // Scan position counter
    reg [2:0] scan_pos;
    always @(posedge clk or posedge rst) begin
        if (rst)                scan_pos <= 3'd0;
        else if (scan_en)       scan_pos <= (scan_pos == 3'd5) ? 3'd0 :
scan_pos + 1;
    end

    // Blinking logic
    reg [23:0] blink_counter;
    always @(posedge clk or posedge rst) begin
        if (rst)    blink_counter <= 0;
        else        blink_counter <= blink_counter + 1;
    end
    wire blink_off = blink_counter[23]; // ~2Hz blink rate

    // Core display logic
    always @(*) begin
        // Default assignment
        num_to_decode = 4'dx;
        digit_sel = 6'b111111; // Default off

        case(scan_pos) //实现一个6选1的多路选择器
            3'd0: begin num_to_decode = sec0;  digit_sel = 6'b111110; end
            3'd1: begin num_to_decode = sec1;  digit_sel = 6'b111101; end
            3'd2: begin num_to_decode = min0;  digit_sel = 6'b111011; end
            3'd3: begin num_to_decode = min1;  digit_sel = 6'b110111; end
            3'd4: begin num_to_decode = hour0; digit_sel = 6'b101111; end
            3'd5: begin num_to_decode = hour1; digit_sel = 6'b011111; end
        endcase

        // Blinking override 在基础显示逻辑之上，增加一个更高优先级的 if 判断
        if (blink_off) begin
            if ((display_mode == S_ADJ_H || display_mode == S_ALARM_H) &&
(scan_pos == 3'd4 || scan_pos == 3'd5))
                digit_sel = 6'b111111; // Turn off hour digits
```

```
            if ((display_mode == S_ADJ_M || display_mode == S_ALARM_M) &&
(scan_pos == 3'd2 || scan_pos == 3'd3))
                digit_sel = 6'b111111; // Turn off minute digits
        end
    end
endmodule
```

目前的基础功能只要加上`clock_controller.v`就已经全部实现了（并且留了一些进阶功能和扩展功能的接口，可以看到目前的代码里有一些信号是还没有被定义和驱动的，还不完整，因为和高级功能发生了交互作用，那些接口都是在后来增加了进阶功能以后才写上去的）其实如果我把这个文档写得好一点，可以遵循真实的设计逻辑，不断往基础代码上逐渐增加功能和修改并展示过程，那样会更符合学习和设计思考的逻辑，而不是像现在一样拿着结果反推。

我们看一下`clock_controller.v`

```
// src/clock_controller.v
module clock_controller (
    input   wire        clk,
    input   wire        rst,
    input   wire        key_mode_pulse,
    input   wire        key_inc_pulse,        //校准时间相关逻辑
    input   wire        key_alarm_off_pulse,  //闹钟逻辑相关
    input   wire [4:0]  hour_in,
    input   wire [5:0]  min_in,
    input   wire [5:0]  sec_in,
    output  reg         time_count_en,
    output  reg         load_en,
    output  reg [4:0]   hour_out,
    output  reg [5:0]   min_out,
    output  wire        alarm_on_flag,   //闹钟逻辑相关
    output  reg [2:0]   display_mode
);

    // 状态定义
    parameter S_NORMAL    = 3'd0;
    parameter S_ADJ_H     = 3'd1;
    parameter S_ADJ_M     = 3'd2;
    parameter S_ALARM_H   = 3'd3;
    parameter S_ALARM_M   = 3'd4;

    reg [2:0] current_state, next_state;

    // FSM Segment 1: 状态寄存器 (时序逻辑)
    always @(posedge clk or posedge rst) begin
        if (rst)    current_state <= S_NORMAL;
        else        current_state <= next_state;
    end
```

```
    // FSM Segment 2: 次态逻辑 (组合逻辑)
    always @(*) begin
        case (current_state) //用三元操作符写出了组合逻辑的顺序
            S_NORMAL:  next_state = key_mode_pulse ? S_ADJ_H     :
S_NORMAL;
            S_ADJ_H:   next_state = key_mode_pulse ? S_ADJ_M     :
S_ADJ_H;
            S_ADJ_M:   next_state = key_mode_pulse ? S_ALARM_H   :
S_ADJ_M;
            S_ALARM_H: next_state = key_mode_pulse ? S_ALARM_M   :
S_ALARM_H;
            S_ALARM_M: next_state = key_mode_pulse ? S_NORMAL    :
S_ALARM_M;
            default:   next_state = S_NORMAL;
        endcase
    end

    // FSM Segment 3: 输出逻辑与内部状态更新 (时序逻辑)
    reg [4:0] alarm_hour_reg;
    reg [5:0] alarm_min_reg;
    reg       is_alarming;
    reg       sec_is_59_reg; //  【优化】将此 reg 移入主 always 块

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            time_count_en  <= 1'b1;
            load_en        <= 1'b0;
            hour_out       <= 5'd0;
            min_out        <= 6'd0;
            display_mode   <= S_NORMAL;
            alarm_hour_reg <= 5'd6;
            alarm_min_reg  <= 6'd0;
            is_alarming    <= 1'b0;
            sec_is_59_reg  <= 1'b0;
        end else begin
            // 默认行为
            load_en <= 1'b0;
            hour_out <= hour_in;
            min_out <= min_in;
            display_mode <= current_state;

            //  【优化】sec_is_59_reg 的更新逻辑合并于此
            sec_is_59_reg <= (sec_in == 6'd59);

            // FSM 输出逻辑
            case (current_state)
                S_NORMAL:  time_count_en <= 1'b1;
                S_ADJ_H:   begin
                    time_count_en <= 1'b0;
                    if (key_inc_pulse) begin load_en <= 1'b1; hour_out <=
```

```
(hour_in == 5'd23) ? 5'd0 : hour_in + 1; end
                end
                S_ADJ_M:   begin
                    time_count_en <= 1'b0;
                    if (key_inc_pulse) begin load_en <= 1'b1; min_out  <=
(min_in == 6'd59) ? 6'd0 : min_in + 1; end
                end
                S_ALARM_H: begin
                    time_count_en <= 1'b1;
                    if (key_inc_pulse) alarm_hour_reg <= (alarm_hour_reg
== 5'd23) ? 5'd0 : alarm_hour_reg + 1;
                end
                S_ALARM_M: begin
                    time_count_en <= 1'b1;
                    if (key_inc_pulse) alarm_min_reg  <= (alarm_min_reg ==
6'd59) ? 6'd0 : alarm_min_reg + 1;
                end
                default:   time_count_en <= 1'b1;
            endcase

            // 闹钟触发/关闭逻辑
            if (key_alarm_off_pulse) begin
                is_alarming <= 1'b0;
            end else if (key_mode_pulse) begin
                is_alarming <= 1'b0;
            end else if (is_alarming == 1'b0 &&
                         hour_in == alarm_hour_reg &&
                         min_in == alarm_min_reg &&
                         sec_in == 6'd0) begin
                is_alarming <= 1'b1;
            end
        end
    end

    // 组合逻辑输出，代码清晰，没有歧义。
    wire hourly_chime = time_count_en && (min_in == 6'd0) && (sec_in ==
6'd0) && sec_is_59_reg;
    assign alarm_on_flag = is_alarming || hourly_chime;

endmodule
```

1. **状态机核心 (FSM Core)**:
   - **两段式实现**: current_state 的时序更新和 next_state 的组合逻辑计算被分开。
   - **逻辑**: next_state 的计算是在任何状态下，如果没有 key_mode_pulse，状态就保持不变 (next_state = ... : current_state)；如果有 key_mode_pulse，就跳转到下一个预设的状态。这是一个简单的**摩尔型 (Moore-type) 状态机**，因为它的次态只依赖于当前状态和输入。

2. **主时序逻辑块 (always @(posedge clk or posedge rst))**: 这是模块的“执行中心”。
   - **复位逻辑**: if (rst) 确保了系统上电后所有内部状态（闹钟时间、响铃标志等）和输出控制信号都处于一个已知的、安全的初始状态。
   - **默认行为**: 在 else 块的开头，代码首先定义了“在没有特殊事件发生时，下一拍应该是什么状态”。
     - load_en <= 1'b0;: 脉冲信号默认拉低。
     - hour_out <= hour_in;: 默认情况下，传递给 time_counter 的加载值就是当前时间，这样即使 load_en 意外为高，时间也不会跳变。这是一个很好的防御性设计。
     - display_mode <= current_state;: 显示模式总是反映状态机的当前状态。
   - **FSM 输出逻辑 (case 语句)**: 根据 current_state 的值，**覆盖**上面的默认行为。例如，在 S_ADJ_H 状态下，它会覆盖 time_count_en 为 0。如果此时 key_inc_pulse 有效，它还会进一步覆盖 load_en 为 1 和 hour_out 为 hour_in + 1。这种“默认->覆盖”的写法让逻辑非常清晰。
   - **闹钟逻辑**: 这是一个独立的 if/else if 链，专门处理 is_alarming 标志位的变化。它的逻辑优先级被明确地定义（手动关闭 > 模式切换关闭 > 时间到达触发）。
3. **组合逻辑输出 (Combinational Outputs)**:
   - hourly_chime 和 alarm_on_flag 是用 assign 和 wire 定义的纯组合逻辑。它们根据当前的时序状态（如 is_alarming, sec_in 等）立即计算出结果。sec_is_59_reg 的使用将一个时序状态引入到组合逻辑判断中，从而实现了边沿检测。

---

现在来看闹钟功能和按键功能的关键组成部分——发声单元和按键消抖逻辑，加上这两个组件之后，整个电子钟的基础功能+进阶功能就已经实现了

`key_debounce.v`：
该模块解决物理按键的“机械抖动”问题。它接收一个不稳定的、可能带有毛刺的输入信号 key_in，并从中提取出一个干净的、只有一个时钟周期宽度的有效按键脉冲 key_pulse

```
// key_debounce.v

module key_debounce (
    input wire clk,
    input wire rst,
    input wire key_in,
    output reg key_pulse
);
    parameter SIMULATION = 0;

    localparam DEBOUNCE_CYCLES = (SIMULATION == 0   ) ? 1000 : 1000000;
    localparam CNT_WIDTH = $clog2(DEBOUNCE_CYCLES);

    reg [1:0] key_state_sync;
    reg [CNT_WIDTH-1:0] counter;
```

```
    reg key_state_q;

    // Step 1: Synchronize input to clock domain (2-stage synchronizer)
    //外部的按键信号 key_in 是一个异步信号，它可能在时钟的任何时刻发生变化。
    //为了避免亚稳态，必须先将其同步到 clk 时钟域。
    always @(posedge clk or posedge rst) begin
        if (rst)
            key_state_sync <= 2'b11;
        else
            key_state_sync <= {key_state_sync[0], key_in};
    end

    // Step 2: Debounce Counter Logic (Corrected)
    // This logic ensures the counter runs ONLY when the input is stable.
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 0;
            key_state_q <= 1'b1;
        end else begin
            if (key_state_sync[1] == key_state_q) begin
                // If input is stable, reset the counter
                counter <= 0;
            end else begin
                // If input has changed, start counting
                if (counter < DEBOUNCE_CYCLES - 1) begin
                    counter <= counter + 1;
                end else begin
                    // If counter is full, the new state is stable
                    key_state_q <= key_state_sync[1];
                end
            end
        end
    end

    // Step 3: Edge detection to generate the pulse
    reg key_state_q_prev;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            key_state_q_prev <= 1'b1;
            key_pulse <= 1'b0;
        end else begin
            key_state_q_prev <= key_state_q;

            // Generate a pulse on the falling edge (1 -> 0) of the FINAL
debounced signal
            if (key_state_q_prev == 1'b1 && key_state_q == 1'b0)
                key_pulse <= 1'b1;
            else
                key_pulse <= 1'b0;
        end
```

```
    end

endmodule
```

1. **第一步：同步化 (Synchronization)**
   - **原理**: 外部的按键信号 key_in 是一个异步信号，它可能在时钟的任何时刻发生变化。为了避免亚稳态，必须先将其同步到我们的 clk 时钟域。
   - **实现**: 采用一个标准的两级同步器（也叫两级触发器）。key_in 信号首先被一个触发器采样，其输出再被第二个触发器采样。reg [1:0] key_state_sync; 和 key_state_sync <= {key_state_sync[0], key_in}; 就是这个过程的简洁实现。经过两级同步后，key_state_sync[1] 就是一个与 clk 同步的、相对稳定的按键状态信号。
2. **第二步：消抖确认 (Debounce Confirmation)**
   - **原理**: 为了确认一次电平变化（例如从1到0）是用户真实的按压动作，而不是一次微小的抖动毛刺，我们采用“延时确认”的策略。当检测到电平变化后，我们启动一个计时器，如果在计时期间（例如20ms）电平能**一直保持**新的状态，我们就认为这次变化是有效的。
   - **实现**:
     - reg key_state_q;: 用于存储**上一个被确认的稳定状态**。
     - if (key_state_sync[1] == key_state_q): 这行代码是消抖的核心。它判断“当前时刻的输入”和“上一个稳定状态”是否相同。
       - 如果相同，说明输入没有变化，一切稳定，counter 保持清零。
       - 如果不同，说明检测到了电平变化，此时启动 counter 开始计数。
     - 在计数期间，如果输入抖动了（key_state_sync[1] 变回了 key_state_q 的值），计数器会立刻被清零，本次消抖失败。
     - 只有当 counter 成功计满 DEBOUNCE_CYCLES（意味着输入在这段时间内一直保持新状态），key_state_q 才会被更新为这个新的稳定状态。
3. **第三步：边沿检测生成脉冲 (Edge Detection & Pulse Generation)**
   - **原理**: 控制器（如 clock_controller）需要的是一个“事件”信号，而不是一个“状态”信号。也就是说，它需要知道“按键**被按下**的那一瞬间”，而不是“按键**正被按着**”。因此，我们需要从 key_state_q 这个稳定的状态信号中，检测出它发生**下降沿**（从1变为0）的那个瞬间。
   - **实现**:
     - reg key_state_q_prev;: 用于存储**上一拍**的 key_state_q 的值。
     - if (key_state_q_prev == 1'b1 && key_state_q == 1'b0): 这行代码描述了下降沿的特征——“上一拍是一，这一拍是零”。
     - 只有在这个条件满足的**一个时钟周期**内，key_pulse 才会被置为高电平，其他所有时间都为低电平，从而生成了一个单周期脉冲。

当物理按键被按下时，key_in 信号首先经过同步器进入FPGA时钟域。然后，消抖逻辑通过一个延时计数器来确认这次电平变化是稳定有效的，并更新内部的稳定状态 key_state_q。最

后，边沿检测逻辑捕捉到 key_state_q 从1到0跳变的那个瞬间，并生成一个干净的单周期脉冲 key_pulse 作为最终输出。

---

`buzzer_controller.v`：
该模块作为一个音频方波发生器，在接收到外部 alarm_on 触发信号时，生成一个特定频率（约1kHz）的方波信号，用于驱动物理蜂鸣器发出声音。

```
// src/buzzer_controller.v
// 功能：在接收到报警触发信号时，产生驱动蜂鸣器的音频方波

module buzzer_controller(
    input wire clk,       // 50MHz 主时钟
    input wire rst,
    input wire alarm_on, // 报警触发信号 (来自 clock_controller)
    output reg beep     // 连接到物理蜂鸣器的输出引脚
);
    // 为了产生约 1kHz 的音频，我们需要一个周期为 50,000 个时钟周期的计数器
    // 50MHz / 50,000 = 1kHz
    reg [15:0] counter;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 16'd0;
            beep <= 1'b0; // 复位时，不响
        end else if (alarm_on) begin //  【核心】只有在报警触发信号为高电平时才工
作
            if (counter == 16'd49999) begin
                counter <= 16'd0;
            end else begin
                counter <= counter + 1;
            end

            // 当计数器在前一半周期时，输出高电平；后一半周期时，输出低电平
            // 这就产生了一个占空比为 50% 的方波
            if (counter < 16'd25000) begin
                beep <= 1'b1;
            end else begin
                beep <= 1'b0;
            end
        end else begin
            // 如果没有报警，计数器和蜂鸣器都保持静默
            counter <= 16'd0;
            beep <= 1'b0;
        end
    end
endmodule
```

1. **控制层 (Control Logic)**:
   - **原理**: 模块的行为完全由 alarm_on 这个输入信号控制。它像一个电源开关，决定了整个模块是处于“工作状态”还是“静默状态”。
   - **实现**: 在主 always 块中，使用了一个 if (rst) ... else if (alarm_on) ... else ... 结构。
     - if (rst): 复位时，强制所有内部状态（计数器）和输出（beep）回到静默的初始状态。
     - else if (alarm_on): **只有当 alarm_on 为高电平时**，才进入核心的方波生成逻辑。
     - else: 当 alarm_on 为低电平时，将计数器和 beep 输出清零，确保蜂鸣器静音。
2. **频率生成层 (Frequency Generation)**:
   - **原理**: 要产生一个1kHz的音频，需要一个周期为1ms的信号。在50MHz的主时钟下，这意味着需要 50,000,000 Hz / 1,000 Hz = 50,000 个时钟周期。因此，模块使用一个16位的计数器 counter，让它从0数到49,999，然后归零，以此来精确地“度量”出50,000个时钟周期的时长。
   - **实现**: 在 if (alarm_on) 分支内，通过 if (counter == 16'd49999) ... else ... 逻辑，实现了一个模50,000的循环计数器。
3. **波形生成层 (Waveform Generation)**:
   - **原理**: 为了产生一个占空比为50%的方波，模块将计数周期一分为二。
   - **实现**:
     - if (counter < 16'd25000): 当计数器处于周期的**前半段**（0 到 24,999）时，将 beep 输出置为高电平。
     - else: 当计数器处于周期的**后半段**（25,000 到 49,999）时，将 beep 输出置为低电平。
   - **效果**: 随着 counter 的循环计数，beep 信号就会稳定地输出高、低、高、低...的方波，从而驱动蜂鸣器发声。

**逻辑流程总结**:
模块平时处于静默状态。一旦 alarm_on 信号变为高电平，内部的计数器开始以50MHz的频率计数，同时根据计数器的值是在前半周期还是后半周期，来决定 beep 输出是高还是低，从而产生1kHz的音频信号。当 alarm_on 信号消失后，模块立即恢复静默。

---

我们通过`DigitalClock.v`将所有模块组装起来

```
// src/DigitalClock.v
// 功能：顶层模块，修正了子模块的连线以适配新的设计

module DigitalClock (
    input   wire        clk,
    input   wire        rst,
    input   wire        key_mode,
    input   wire        key_inc,
    input   wire        key_alarm_off,
    output  wire        beep,
```

```
    output  wire [6:0]  seg_out,
    output  wire [5:0]  digit_sel
);

    // 内部信号线
    wire clk_1hz_en_wire; //  【改动】信号重命名，表示它是一个使能信号
    wire [4:0] hour_from_counter;
    wire [5:0] min_from_counter;
    wire [5:0] sec_from_counter;
    wire [3:0] num_to_decode_wire;
    wire key_mode_pulse, key_inc_pulse, key_alarm_off_pulse;
    wire time_count_en_wire, load_en_wire;
    wire [4:0] hour_to_counter;
    wire [5:0] min_to_counter;
    wire alarm_on_flag_wire;
    wire [2:0] display_mode_wire;

    // 实例化按键消抖模块
    key_debounce debounce_mode (.clk(clk), .rst(rst), .key_in(key_mode),
.key_pulse(key_mode_pulse));
    key_debounce debounce_inc (.clk(clk), .rst(rst), .key_in(key_inc),
.key_pulse(key_inc_pulse));
    key_debounce debounce_alarm_off (.clk(clk), .rst(rst),
.key_in(key_alarm_off), .key_pulse(key_alarm_off_pulse));

    // 实例化主控制器
    clock_controller u_controller (
        .clk(clk), .rst(rst),
        .key_mode_pulse(key_mode_pulse), .key_inc_pulse(key_inc_pulse),
.key_alarm_off_pulse(key_alarm_off_pulse),
        .hour_in(hour_from_counter), .min_in(min_from_counter),
.sec_in(sec_from_counter),
        .time_count_en(time_count_en_wire), .load_en(load_en_wire),
        .hour_out(hour_to_counter), .min_out(min_to_counter),
        .alarm_on_flag(alarm_on_flag_wire),
        .display_mode(display_mode_wire)
    );

    // 实例化蜂鸣器控制器 (假设存在)
    buzzer_controller u_buzzer (.clk(clk), .rst(rst),
.alarm_on(alarm_on_flag_wire), .beep(beep));

    // 实例化时钟分频器 (现在是使能信号生成器)
    clk_divider u_clk_divider (
        .clk_in(clk),
        .rst(rst),
        .clk_1hz_en(clk_1hz_en_wire) //  【改动】连接到新的输出端口
    );

    // 实例化时间计数器
```

```
    time_counter u_time_counter (
        .clk(clk),                    //  【改动】连接主时钟
        .clk_1hz_en(clk_1hz_en_wire),//  【改动】连接1Hz使能信号
        .rst(rst),
        .time_count_en(time_count_en_wire), .load_en(load_en_wire),
        .hour_in(hour_to_counter), .min_in(min_to_counter),
        .hour(hour_from_counter), .min(min_from_counter),
.sec(sec_from_counter)
    );

    // 实例化显示扫描模块
    display_scanner u_display_scanner (
        .clk(clk), .rst(rst),
        .hour(hour_from_counter), .min(min_from_counter),
.sec(sec_from_counter),
        .display_mode(display_mode_wire),
        .num_to_decode(num_to_decode_wire),
        .digit_sel(digit_sel)
    );

    // 实例化显示译码模块
    display_decoder u_display_decoder (.num_in(num_to_decode_wire),
.seg_out(seg_out));

endmodule
```

---

现在已经实现了目标功能，编译通过后可以写一个testbench来模拟各种输入行为例如按按键、设置闹钟等，再写一个run-simulation.do脚本和wave_format.do来自动在modelsim里编译和生成波形图并将信号分组显示

## 2.仿真脚本

```
// testbench/full_functionality_tb.v
// --- A Comprehensive Testbench to Demonstrate ALL Clock Features ---

`timescale 1ns / 1ps

module full_functionality_tb;

    // -- Testbench Internal Signals --
    reg clk;
    reg rst;
    reg key_mode;
    reg key_inc;
    reg key_alarm_off;
```

```
    // -- Instantiate the Unit Under Test (UUT) --
    DigitalClock uut (
        .clk(clk),
        .rst(rst),
        .key_mode(key_mode),
        .key_inc(key_inc),
        .key_alarm_off(key_alarm_off),
        .beep(),
        .seg_out(),
        .digit_sel()
    );

    // -- Set Simulation Parameters for Child Modules --
    defparam uut.u_clk_divider.SIMULATION = 1;
    defparam uut.debounce_mode.SIMULATION = 1;
    defparam uut.debounce_inc.SIMULATION = 1;
    defparam uut.debounce_alarm_off.SIMULATION = 1;
    defparam uut.u_display_scanner.SIMULATION = 1;

    // -- Clock Generation --
    initial begin
        clk = 1'b0;
        forever #10 clk = ~clk; // 50MHz clock
    end

    // ========================================================================
    //         Simple, Robust, Parameter-less Tasks
    // ========================================================================
    task press_key_mode;
    begin
        @(posedge clk); key_mode = 1'b0; #21000; @(posedge clk); key_mode
= 1'b1; @(posedge clk);
    end
    endtask

    task press_key_inc;
    begin
        @(posedge clk); key_inc = 1'b0; #21000; @(posedge clk); key_inc =
1'b1; @(posedge clk);
    end
    endtask

    task press_key_alarm_off;
    begin
        @(posedge clk); key_alarm_off = 1'b0; #21000; @(posedge clk);
key_alarm_off = 1'b1; @(posedge clk);
    end
    endtask
    // ========================================================================
```

```
    // --- Main Test Sequence ---
    initial begin
        $display("--- FULL FUNCTIONALITY TESTBENCH START ---");

        // --- STEP 1: SYSTEM RESET ---
        $display("\n[1. RESET] Initializing and resetting the clock...");
        key_mode = 1'b1; key_inc = 1'b1; key_alarm_off = 1'b1;
        rst = 1'b1; #200; rst = 1'b0;
        #50000; // Wait for system to stabilize
        $display("   >> Reset complete. Time should be 00:00:00. State
should be S_NORMAL.");


        // --- STEP 2: ADJUST REAL TIME ---
        $display("\n[2. ADJUST TIME] Setting time to 23:58:00...");

        // Enter ADJ_H mode
        press_key_mode; #50000;
        $display("   >> Entered S_ADJ_H mode. Adjusting Hour to 23...");

        // Set hour to 23
        repeat (23) begin press_key_inc; #50000; end

        // Enter ADJ_M mode
        press_key_mode; #50000;
        $display("   >> Entered S_ADJ_M mode. Adjusting Minute to 58...");

        // Set minute to 58
        repeat (58) begin press_key_inc; #50000; end

        // Cycle through alarm modes back to NORMAL
        press_key_mode; #50000; // -> S_ALARM_H
        press_key_mode; #50000; // -> S_ALARM_M
        press_key_mode; #50000; // -> S_NORMAL
        $display("   >> Returned to S_NORMAL. Time is now set to
23:58:00.");


        // --- STEP 3: VERIFY TIME COUNT & CARRY-OVER ---
        $display("\n[3. VERIFY COUNTING] Letting time run for 125 seconds
to observe carry-over...");
        // Wait for 125 simulated seconds. Each second is 1us in sim time
(50 cycles * 20ns).
        // A large delay ensures we see the clock tick over from 23:59:59
to 00:00:00.
        #130_000_000; // Wait for 130 simulated seconds (130us)
        $display("   >> Time has advanced. Should be past midnight. Check
waveform for 23:59:59 -> 00:00:00 transition.");
```

```
        // --- STEP 4: SET ALARM ---
        $display("\n[4. SET ALARM] Setting alarm time to 00:02:00...");

        // Enter ALARM_H mode
        press_key_mode; #50000; // -> S_ADJ_H
        press_key_mode; #50000; // -> S_ADJ_M
        press_key_mode; #50000; // -> S_ALARM_H
        $display("   >> Entered S_ALARM_H. Default alarm hour is 6.
Setting to 0...");

        // Default alarm hour is 6. To get to 0, we need 24-6=18 presses.
        repeat(18) begin press_key_inc; #50000; end

        // Enter ALARM_M mode
        press_key_mode; #50000; // -> S_ALARM_M
        $display("   >> Entered S_ALARM_M. Setting alarm minute to 2...");

        // Set alarm minute to 2
        repeat(2) begin press_key_inc; #50000; end

        // Return to NORMAL mode
        press_key_mode; #50000;
        $display("   >> Returned to S_NORMAL. Alarm is now armed for
00:02:00.");


        // --- STEP 5: TRIGGER AND CANCEL ALARM ---
        $display("\n[5. TRIGGER ALARM] Waiting for real time to reach
00:02:00...");

        // We know the time is currently around 00:00:05. We need to wait
about 115 more seconds.
        #120_000_000; // Wait for another 120 simulated seconds
        $display("   >> Real time is now 00:02:xx. The alarm should be
beeping. Check waveform for 'beep' signal.");

        // Let the beep run for a bit
        #100_000; // 100us

        $display("   >> Now, pressing ALARM_OFF key to cancel the
beep...");
        press_key_alarm_off; #50000;
        $display("   >> Alarm should now be silent. Check 'beep' signal is
low.");


        // --- STEP 6: TEST HOURLY CHIME (整点报时) ---
        $display("\n[6. HOURLY CHIME] Setting time near the next hour
(00:59:55) to test chime...");
```

```
        // Go to ADJ_M and set minute to 59
        press_key_mode; #50000; // S_ADJ_H
        press_key_mode; #50000; // S_ADJ_M
        // Current minute is ~2. 59-2 = 57 presses.
        repeat(57) begin press_key_inc; #50000; end

        // Return to normal
        press_key_mode; #50000; // S_ALARM_H
        press_key_mode; #50000; // S_ALARM_M
        press_key_mode; #50000; // S_NORMAL
        $display("   >> Time set to 00:59:xx. Waiting for the hour to
change...");

        #10_000_000; // Wait 10 seconds
        $display("   >> Clock should have passed 01:00:00. A short beep
for the hourly chime should have occurred.");


        $display("\n--- FULL FUNCTIONALITY TEST COMPLETE ---");
        $stop;
    end
endmodule
```

这份 Testbench 模拟了以下六个用户场景：

**1. 场景一：开机与复位 (STEP 1: SYSTEM RESET)**

- **模拟行为**: 模拟了电子钟**上电或按下硬件复位按钮**的瞬间。rst 信号被拉高一小段时间后拉低。
- **测试目的**: 验证系统的**清零功能**是否正常。

**2. 场景二：首次校准时间 (STEP 2: ADJUST REAL TIME)**

- **模拟行为**: 模拟用户拿到新手表后，第一次设置当前时间。脚本将时间从 00:00:00 调整到接近午夜的 23:58:00。
  - 按下 key_mode 进入“调时”模式。
  - 通过 repeat 循环，**连续、快速地**按下23次 key_inc 来设置小时。
  - 再按 key_mode 进入“调分”模式。
  - 再连续按58次 key_inc 设置分钟。
  - 最后，连续按 key_mode 跳过“闹钟设置”，返回正常模式。
- **测试目的**: 验证**状态机切换**、**时间加载**、**按键递增**以及**暂停计时**等核心交互功能。

**3. 场景三：跨天进位观察 (STEP 3: VERIFY TIME COUNT & CARRY-OVER)**

- **模拟行为**: 模拟用户将手表设置好后，**让其自然走时**，并特别关注一天中最重要的时刻——从 23:59:59 到 00:00:00 的跨天转换。脚本通过一个巨大的延时 `#130_000_000` 来

快进时间。
- **测试目的**: 验证**长时间运行的稳定性**和最复杂的**进位逻辑**（秒->分，分->时，时->天）是否正确。

**4. 场景四：设置闹钟 (STEP 4: SET ALARM)**

- **模拟行为**: 模拟用户在某个时间点（此时已是第二天凌晨）决定设置一个早晨的闹钟。脚本将闹钟时间设置为 00:02:00。
  - 通过 key_mode 进入“闹钟小时设置”模式。
  - 通过 key_inc 将默认的闹钟小时6点调整为0点。
  - 再按 key_mode 进入“闹钟分钟设置”模式。
  - 通过 key_inc 将闹钟分钟设置为2分。
  - 最后返回正常模式，闹钟设置完毕。
- **测试目的**: 验证**闹钟时间的写入逻辑**，并确认在设置闹钟期间**正常计时没有被中断**。

**5. 场景五：闹钟触发与手动关闭 (STEP 5: TRIGGER AND CANCEL ALARM)**

- **模拟行为**: 模拟用户等待闹钟响起，并在响铃后将其关闭的完整过程。
  - 脚本再次使用长延时来快进时间，直到接近 00:02:00。
  - 等待闹钟触发。
  - 在闹钟响了一小段时间后，按下 key_alarm_off。
- **测试目的**: 验证**闹钟时间比较逻辑**、**触发信号 (is_alarming)** 以及**手动关闭功能**是否全部正常。

**6. 场景六：整点报时测试 (STEP 6: TEST HOURLY CHIME)**

- **模拟行为**: 这是一个专门为“整点报时”功能设计的极限测试。脚本将时间手动设置为接近下一个小时的时刻（xx:59:xx），然后等待时间自然跨过整点。
- **测试目的**: 验证**整点报时 (hourly_chime)** 的逻辑是否能在正确的时间点（xx:00:00）触发一个短暂的蜂鸣。

---

## 如何检查波形图

1. **第一步：找到时间基准 (STEP 1)**
   - 将波形图缩放到最左侧，找到 rst 信号从 1 变为 0 的地方。
   - **检查**: 在 rst 变低后，hour/min/sec_from_counter 是否都变成了 0？current_state 是否为 S_NORMAL (3'b000)？
2. **第二步：验证时间设置 (STEP 2)**
   - 找到 Transcript 中打印 [2. ADJUST TIME] 的时间点，在波形图上定位到该处。
   - **检查 key_mode**: 你会看到一个 key_mode 的低电平脉冲。
   - **检查 current_state**: 紧接着 key_mode 脉冲后，current_state 应该从 000 变为 001 (S_ADJ_H)。

- **检查 key_inc 和 hour**: 观察 key_inc 信号，你会看到23个密集的脉冲。同时，hour_from_counter 的值应该随着每个脉冲从 0 一步步增加到 23。
- **重复检查**: 按照同样的方法，验证分钟从 0 被设置为 58 的过程。
- **最终确认**: 在该步骤的末尾，确认 hour_from_counter 是 23，min_from_counter 是 58，并且 current_state 最终回到了 000 (S_NORMAL)。

3. **第三步：寻找跨天瞬间 (STEP 3)**
   - 这是一个长延时，你需要拖动时间轴。一个快捷方法是**搜索信号值的变化**。在 hour_from_counter 信号上右键，选择 Find -> Transition，寻找从 23 变为 0 的那个点。
   - **检查**: 在那个精确的时刻，min_from_counter 是否也从 59 变为 0？sec_from_counter 是否也从 59 变为 0？
4. **第四步：核对闹钟设置 (STEP 4)**
   - 定位到 [4. SET ALARM] 的时间点。
   - **检查 current_state**: 确认 current_state 依次切换到了 S_ALARM_H (011) 和 S_ALARM_M (100)。
   - **检查 alarm_hour/min_reg**: 这两个是内部信号，需要提前加到波形图中。确认在 S_ALARM_H 状态下，alarm_hour_reg 被设置为 0；在 S_ALARM_M 状态下，alarm_min_reg 被设置为 2。
5. **第五步：见证闹钟触发 (STEP 5)**
   - 再次拖动或搜索时间轴，找到 hour=0, min=2, sec=0 的瞬间。
   - **核心检查**:
     - 在这个瞬间，is_alarming 信号是否从 0 变为 1？
     - 紧接着，alarm_on_flag 是否也变为 1？
     - beep 信号是否开始输出1kHz的方波？
   - **检查关闭功能**: 找到 key_alarm_off 的脉冲。紧接着这个脉冲，is_alarming, alarm_on_flag, 和 beep 是否都立即恢复为 0？
6. **第六步：捕捉整点报时 (STEP 6)**
   - 定位到 [6. HOURLY CHIME] 的时间点。
   - 搜索 hour_from_counter 从 0 变为 1 的瞬间（因为之前是0点多）。
   - **检查**: 在 hour=1, min=0, sec=0 的那个精确时刻，beep 信号是否出现了一个**短暂的**（持续1秒）的方波信号？

---

波形设定：

```
# simulation/modelsim/wave_format.do
# --- FINAL & CORRECTED FOR DIRECT SIGNAL ACCESS ---

onerror {resume}
quietly WaveActivateNextPane {} 0
```

```
# --- Group 1: Core Inputs & Clock Enables ---
# Testbench-level signals
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/clk
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/rst
# 【修正】直接引用UUT内部的信号
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/uut/clk_1hz_en_wire
add wave -noupdate -divider {User Keys}
# Testbench-level signals
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/key_mode
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/key_inc
add wave -noupdate -expand -group {Inputs & Clocks} -radix binary
/full_functionality_tb/key_alarm_off

# --- Group 2: Key Debouncing & Pulse Generation ---
# 【修正】直接引用UUT内部的信号
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/debounce_mode/key_state_sync
add wave -noupdate -expand -group {Key Processing} -radix unsigned
/full_functionality_tb/uut/debounce_mode/counter
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/debounce_mode/key_state_q
add wave -noupdate -divider {Generated Pulses}
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/key_mode_pulse
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/key_inc_pulse
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/key_alarm_off_pulse
add wave -noupdate -expand -group {Key Processing} -radix binary
/full_functionality_tb/uut/debounce_mode/key_in
add wave -noupdate -divider {Debounce Internals}

# --- Group 3: Controller FSM & Control Signals ---
# 【修正】直接引用UUT内部的信号
add wave -noupdate -expand -group {Controller} -radix symbolic
/full_functionality_tb/uut/u_controller/current_state
add wave -noupdate -expand -group {Controller} -radix binary
/full_functionality_tb/uut/time_count_en_wire
add wave -noupdate -expand -group {Controller} -radix binary
/full_functionality_tb/uut/load_en_wire
add wave -noupdate -divider {Controller Outputs to Counter}
add wave -noupdate -expand -group {Controller} -radix unsigned
/full_functionality_tb/uut/hour_to_counter
add wave -noupdate -expand -group {Controller} -radix unsigned
/full_functionality_tb/uut/min_to_counter
```

```
# --- Group 4: Time Counter Data ---
# 【修正】直接引用UUT内部的信号
add wave -noupdate -expand -group {Time Data} -radix unsigned
/full_functionality_tb/uut/hour_from_counter
add wave -noupdate -expand -group {Time Data} -radix unsigned
/full_functionality_tb/uut/min_from_counter
add wave -noupdate -expand -group {Time Data} -radix unsigned
/full_functionality_tb/uut/sec_from_counter

# --- Group 5: Alarm Logic ---
# 【修正】直接引用UUT内部的信号
add wave -noupdate -expand -group {Alarm} -radix unsigned
/full_functionality_tb/uut/u_controller/alarm_hour_reg
add wave -noupdate -expand -group {Alarm} -radix unsigned
/full_functionality_tb/uut/u_controller/alarm_min_reg
add wave -noupdate -divider {Alarm Flags}
add wave -noupdate -expand -group {Alarm} -radix binary
/full_functionality_tb/uut/alarm_on_flag_wire
add wave -noupdate -expand -group {Alarm} -radix binary
/full_functionality_tb/uut/beep

# --- Waveform Window Configuration ---
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ps} 0}
quietly wave cursor active 1
configure wave -namecolwidth 350
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {0 ps} {400000 ps}
```

---

modelsim里运行的脚本：

```
# ================================================================
# ModelSim Simulation Script: run_manual_debug.do
# ================================================================
```

```
# This script is for step-by-step manual debugging.
#
# USAGE: In ModelSim Transcript, type: do
F:/FPGA/quartus/bin64/simulation/modelsim/run_simulation.do
# ==========================================================================

echo "--- Starting Automated Script for MANUAL DEBUGGING ---"

# --- Step 1: Clean and Prepare Environment ---
cd F:/FPGA/quartus/bin64/simulation/modelsim
if {[file exists work]} {
    vdel -lib work -all
    echo "--- Old 'work' library deleted. ---"
}
vlib work
vmap work work
echo "--- New 'work' library created and mapped. ---"

# --- Step 2: Compile All 8 Design Source Files ---
echo "--- Compiling 8 design files... ---"
vlog -work work F:/FPGA/quartus/bin64/src/clk_divider.v
vlog -work work F:/FPGA/quartus/bin64/src/display_decoder.v
vlog -work work F:/FPGA/quartus/bin64/src/key_debounce.v
vlog -work work F:/FPGA/quartus/bin64/src/time_counter.v
vlog -work work F:/FPGA/quartus/bin64/src/display_scanner.v
vlog -work work F:/FPGA/quartus/bin64/src/buzzer_controller.v
vlog -work work F:/FPGA/quartus/bin64/src/clock_controller.v
vlog -work work F:/FPGA/quartus/bin64/src/DigitalClock.v

# --- Step 3: Compile the MANUAL DEBUG Testbench ---
# 【核心改动】: 编译新的、简单的Testbench
echo "--- Compiling manual debug testbench file... ---"
vlog -work work F:/FPGA/quartus/bin64/testbench/full_functionality_tb.v

# --- Step 4: Launch the Simulator ---
# 【核心改动】: 启动新的Testbench
echo "--- Launching simulator with full_functionality_tb... ---"
vsim -voptargs="+acc" work.full_functionality_tb

# --- Step 5: Configure Waveform and Run Simulation ---
echo "--- Loading custom wave format and running... ---"
do F:/FPGA/quartus/bin64/simulation/modelsim/wave_format.do
run -all

# --- Script Finished ---
echo "--- Simulation paused. Analysis can begin. ---"
```

在自己电脑上运行时需要把路径修改一下。可以用Ctrl+F在Vscode里全局查找替换。

2025.9.20 cx