北京邮电大学课程设计报告

<table>
<tr><td>课程设计名称</td><td colspan="2">数字逻辑与数字系统课程设计</td><td>学 院</td><td>计算机学院</td><td>指导教师</td><td>靳秀国</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2023211311</td><td>11</td><td colspan="2">2023211203</td><td>丁乐航</td><td colspan="2"></td></tr>
<tr><td>2023211311</td><td>4</td><td colspan="2">2023211196</td><td>马一民</td><td colspan="2"></td></tr>
<tr><td>2023211311</td><td>17</td><td colspan="2">2023211209</td><td>肖璨</td><td colspan="2"></td></tr>
<tr><td>2023211311</td><td>10</td><td colspan="2">2023211202</td><td>张鑫溢</td><td colspan="2"></td></tr>
<tr><td>课程设计内容</td><td colspan="6">主要内容：基于 Altera EPM7128 平台，使用 verilog 语言设计电子钟和药片装瓶系统。
教学目的：掌握数字设计的基本原理和技术，提高学生的创造性和实践能力，培养学生的综合素质，包括设计思维、创新能力、团队合作能力，帮助学生了解数字设计的应用场景，为后续的课程及未来职业打下基础。
实验方法：课下设计、编译、仿真，课上进行上机下载、调试。
团队分工：见附录 3</td></tr>
<tr><td>学生课程设计报告
(附页)</td><td colspan="6"></td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">遵照实践教学大纲并根据以下四方面综合评定成绩：
1、课程设计目的任务明确，选题符合教学要求，份量及难易程度
2、团队分工是否恰当与合理
3、综合运用所学知识，提高分析问题、解决问题及实践动手能力的效果
4、是否认真、独立完成属于自己的课程设计内容，课程设计报告是否思路清晰、文字通顺、书写规范
评语：

成绩：

指导教师签名：

2025 年 6 月 10 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

Table of Contents

一、课题硬件环境及总体描述

一、课题硬件环境

课程设计基于 TEC-8（PLUS）实验平台，核心为 Altera EPM7128 芯片，支持 Verilog/VHDL 与原理图混编。平台配备数据开关、主时钟（多档位频率）等输入设备，数码管、扬声器等输出设备，以及电源、逻辑笔接口等资源，可实现状态显示与调试监测。
硬件层面需要充分利用实验平台资源，例如电子钟中，EPM7128 芯片作为主控制器，处理计时逻辑；数据开关用于时间参数设置；数码管显示模块展示实时时间；扬声器实现报时功能。软件设计采用层次化开发方法，底层通过 Verilog 实现时钟分频、数码管驱动等基础模块，上层构建状态机控制系统流程。而在药片装瓶系统中，数据开关设置装瓶参数；数码管显示统计数据；平台七段显示数码管的显示模拟装瓶状态，同样采用层次化开发方法。

二、总体描述

本实验需要进行两个课题的研究，第一个是电子钟的设计，需要实现电子钟的基本功能，如 24 小时制时钟、整点报时、时间设置及模式切换功能。同时第二个则是药片装瓶系统设计，需要在实验平台中实现类似药片生产工厂的设计，如药片装瓶计数、每瓶药片数及总数量限定等功能。除了基本功能外，上述课题均可实现附加功能，以便适配多变的环境需求。
本次课程设计分两个阶段完成：期中之前主要进行电子钟系统和药片系统的基础设计与实现，包括功能分析、硬件搭建、程序编写和系统调试等工作；期末阶段重点完成电子钟系统和药片装瓶系统的优化，涵盖增加新功能，资源优化等环节。两个项目共用实验平台资源，并预留了平台熟悉和总结汇报的时间，在课程周期内顺利完成全部设计任务。

二、课题一：电子钟系统设计

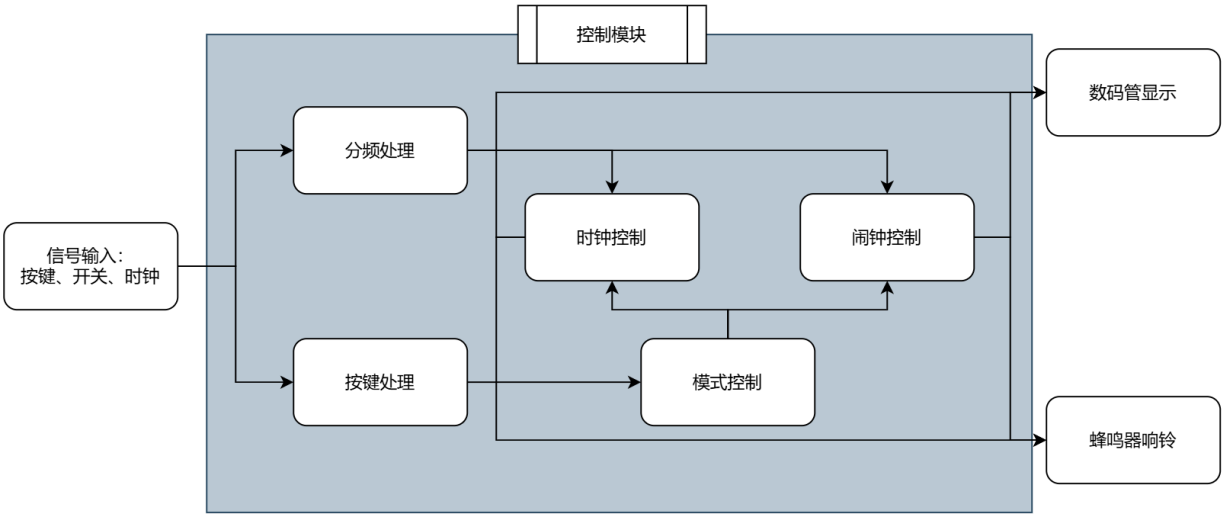
一、 需求分析

为了实现在 TEC-8 上实现电子钟，要求设计时钟控制系统，该系统应实现功能如下：
1. 实现 24 小时制时钟功能
2. 实现时间设置功能
3. 实现闹钟设置功能
4. 实现整点报时、闹钟闹铃功能
5. 实现不同状态切换、设置状态时闪烁显示

二、 概要设计

1. 模块架构与输入输出

电子钟要求实现时间计数、时间与闹钟设置、闹钟响铃、整点报时等功能，将其抽象化为模块，整体架构如下



输入设置：

当复位信号 rst_n 被置 1 时，时钟开始运行。通过开关置入 set_clock 或 set_alarm 信号，可以分别进入时钟设置模式和闹钟设置模式。在两个设置模式中，通过按钮输入 set_time 和 set_shift 脉冲，可以分别改变时钟对应位数字和时钟位。

控制系统：

将输入按键进行一定处理转换，使其能够设置时钟每一位数字大小，能够设置精度最高到分钟的闹钟，并且能够控制模块切换；同时时钟控制和闹钟控制功能也会根据输入时钟 clk 和输入控制

实现不同的运行方案

输出设置：

通过时钟控制和闹钟控制对显示数字和蜂鸣器响铃进行调整，并在模块中进一步处理闪烁控制和音调控制

综上所述，电子钟系统的输入具体如下：

信号名	说明
clk	主系统 1kHz 时钟，提供全局驱动
rst_n	系统复位，低电平有效
set_clock	用户进入时钟设置模式的控制信号
set_alarm	用户进入闹钟设置模式的控制信号
set_shift_pre	PULSE 按键脉冲输入，用于移动设置位置
set_time_pre	QD 按键脉冲输入，用于对选中位加 1

电子钟系统的输出如下：

信号名	说明
seven_led	七段数码管的段选编码，用于显示秒钟位数字
display_sec_tens	秒钟十位
display_min_ones	分钟个位
display_min_tens	分钟十位
display_hour_ones	小时个位
display_hour_tens	小时十位
speaker	蜂鸣器控制输出，用于发声

2. 状态机设计

系统运行过程中有两个状态机：模式状态机和位切换状态机，控制放在 mode_control 模块中

模式状态机接受模式控制信号 set_clock 和 set_alarm，对模式 mode 进行控制；位切换状态机接受 set_shift_pre 按键处理后的信号 set_shift，对当前模式进行转换

1. 模式状态机

状态定义：主状态机定义了三个状态，通过 2 位寄存器 mode 表示：

NORMAL_MODE = 2'b00; 正常运行状态，时钟正常计数

CLOCK_SET_MODE = 2'b01; 时钟控制状态，时钟停止，可以设置当前时间

ALARM_SET_MODE = 2'b10; 闹钟控制状态，时钟仍在运行，但是此时可以设置一个闹钟时间

触发信号：

控制信号：set_clock、set_alarm，用于状态转换。

复位信号：rst_n（低电平有效），将状态机复位到 NORMAL_MODE。

2.位切换状态机

状态定义：位切换状态机定义了六个状态，分别表示时钟的六位，通过 3 位寄存器 pos 表示；根据时钟控制模块和闹钟控制模块自动机的转换略有不同

触发信号：

控制信号：PLUSE（正边沿触发）。

复位信号：rst_n（低电平有效），将状态机复位到 3'0 。

三、 设计详解

我们的电子钟项目架构是一个具有闹钟功能的数字时钟系统，采用模块化设计，包含时钟计时、闹钟设置、模式控制、显示控制和音响提示等功能。整个程序由一个顶层文件 clock_top.v 作为入口，负责输入输出所有参数，并连接所有子模块，以下是各个模块功能的详细介绍：

1. 顶层模块(clock_top)

模块	功能实现
Hz_12	时钟分频模块，将系统时钟分频为 1Hz 和 2Hz 信号。
key_handle	按键处理模块，对按键信号进行处理
mode_control	模式控制模块，根据按键切换 “正常模式”“设置时钟”“设置闹钟” 模式，并输出当前调整位置 pos 用以指示当前模式
clock_control	时钟控制模块，实现时钟逻辑，在正常模式下按 1Hz 递增时分秒，在设置模式下根据按键调整对应数位。
alarm_control	闹钟控制模块，实现闹钟设置；对比当前时间与闹钟时间，触发闹钟响铃 alarm_ring 和整点报时 time_ring。
display_control	显示控制模块，根据模式选择显示时间模式；输出对应 bcd 码或者七段 led 码显示；实现闪烁。

clock_top 作为电子钟系统的顶层模块，采用模块化设计理念整合 6 个子模块实现完整功能。

clock_top 的输入信号包含系统时钟 clk 和初始化寄存器与计数器的低电平有效异步复位信号 rst_n；用户交互信号包括进入时钟设置模式的 set_clock、进入闹钟设置模式的 set_alarm、用于循环切换调整位置的原始位置调整键 set_shift_pre 和用于递增当前选中位置数值的时间调整键 set_time_pre；同时还有 1Hz 和 2Hz 的标准时钟信号。

输出信号主要信号有指示信号和输出信号，指示信号包括 alarm_ring 在当前时间与闹钟时间匹配时触发闹钟响铃，time_ring 在每小时整点触发报时；输出信号中 speaker 为结合 tone_divide 生成不同音调的蜂鸣器驱动信号，tone_divide 是调节蜂鸣器频率的 2 位音调分频控制信号，实现声音模块的输出；显示输出信号里，seven_led[6:0]为控制数码管显示数字 0-9 的七段数码管信号，6 组 display_*信号分别控制秒、分、时的十位和个位显示。另外 sec_ten_zero/min_ten_zero/hour_ten_zero* 为对应数位为 0 且无需显示时置高的辅助。

2. 模式控制模块(mode_control)

mode_control 模块是数字时钟的模式与位置控制模块，主要功能包括切换系统工作模式（正常显示、时钟设置、闹钟设置）和设置模式下的时间位数位置调整。

具体实现上，通过 clk 和 rst_n 实现时序同步与复位初始化，复位时默认进入正常模式（mode=00）且位置指针 pos=0；当 set_clock 或 set_alarm 为高电平时，分别进入时钟设置（mode=01）或闹钟设置模式（mode=10），若两者均释放且当前为设置模式，则自动返回正常模式。位置调整逻辑中，仅在设置模式下响应 set_shift 键：时钟设置模式下，pos 从 5（秒个位）递减至 0（时十位）循环切换（如 5→4→3→2→1→0→5）；闹钟设置模式下，pos 仅在 0-3（时十位至分个位）间递减循环（如 3→2→1→0→3），实现位切换逻辑。

3. 按键处理模块 (key_handle)

该模块主要将用户交互的按键产生的信号进行处理，key_handle 模块通过同步寄存器缓存输入信号、组合逻辑检测上升沿、最后时序逻辑生成单周期脉冲，把上升沿脉冲信号转换为一个周期的高电平信号，确保在后续代码实现中能可靠的检测到对应按键动作以实现相关功能。

4. 时钟分频模块 (Hz_12)

时钟分频模块的功能是将输入的 1kHz 时钟信号进行分频，分别输出 1Hz 的脉冲信号和 2Hz 的方波信号。通过一个共享的 10 位二进制计数器，精确控制两个时钟信号的输出频率，具备逻辑结构简洁、资源占用少的特点。

接口说明：

信号名称	方向	位宽	说明
clk	输入	1	系统输入时钟信号，频率为 1kHz
clk_1hz	输出	1	输出的 1Hz 脉冲信号
clk_2hz	输出	1	输出的 2Hz 方波信号

该模块通过一个 10 位宽的二进制计数器 counter，对 1kHz 的输入时钟进行分频。其中计数器范围为 0~999，共 1000 个时钟周期，正好对应 1 秒的时间。我们根据计数器当前值，在计数为 999 时对 clk_1hz 信号进行拉高，使之成为 1hz 脉冲信号。而分别在计数为 499 和 999 时对 clk_2hz 信号进行翻转，从而实现 2hz 的方波信号。

5. 时钟控制模块 (clock_control)

本模块为数字时钟的核心计数控制模块，负责实现 24 小时制的时钟计时功能及手动调时功能。

它根据输入的秒脉冲信号 clk_1hz 实现正常时间累加计数，同时支持通过按键进行当前选中位置的时间调整，具有时分秒的完整显示输出。

接口说明：

输入信号：

信号名称	位宽	说明
clk	1	主时钟信号（用于组合逻辑的时序控制）
clk_1hz	1	1Hz 脉冲信号，用于每秒更新时间
set_time	1	调整时间按键（按下时对选中位加 1）
rst_n	1	复位信号，低电平有效，复位后时间归零
mode	2	模式选择信号：00=正常计时，01=调时模式
pos	3	时间调整位置选择信号，用于指定要调整的时间位

输出信号：

信号名称	位宽	说明
sec_ones	4	秒的个位（0~9）
sec_tens	3	秒的十位（0~5）
min_ones	4	分的个位（0~9）
min_tens	3	分的十位（0~5）
hour_ones	4	时的个位（0~9）
hour_tens	2	时的十位（0~2）

本模块主要完成两种功能：一是基于 1Hz 时钟信号的正常计时功能，二是通过按键实现的人工调时功能。其行为由外部输入信号 mode 决定，其中 mode=2'b00 表示正常模式，mode=2'b01 表示时间设置模式。

在正常模式下，模块会在每一个 clk_1hz 信号的上升沿触发一次时间递增操作。该过程遵循标准的 24 小时制计时逻辑，从秒开始逐层向上进位。当秒个位 sec_ones 达到 9 时清零，并使秒十位 sec_tens 加 1；若 sec_tens 达到 5 则清零，进而带动分钟个位 min_ones 加 1。此进位链依次延续至分钟十位 min_tens 和小时两位。当小时计数达到 23（即 hour_tens=2 且 hour_ones=3）时，下一秒将整体归零为 00:00:00，从而构成一个完整的时钟循环。

在时间设置模式下，模块允许用户通过按键对当前选中的时间位进行手动加 1 操作。具体控制由 pos 信号决定，表示当前选中的时间位（例如 pos=6 表示秒个位）。在检测到 set_time 有效时，根据所选位置判断当前位的最大值并执行加 1 或归零操作。为了保证时间的有效性，模块在设置小时时会进行约束检查。例如当小时十位被设置为 2 时，小时个位不能超过 3，若超出则自动调整为 3 或归零，确保设置后的时间始终在合法范围内。

6. 闹钟设置模块 (alarm_control)

alarm_control 模块主要用于实现电子时钟系统中的闹钟功能，包括闹钟时间的设置、闹钟触发响铃、以及整点报时功能。该模块能够接收当前时间的时分秒数据，在指定的时刻产生响铃信号（alarm_ring），并在每小时整点发出整点报时信号（time_ring）。此外，该模块支持用户通过按

键选择并逐位设置闹钟时间，配合模式控制信号 mode 实现不同的功能切换。

模块支持两种模式：

1. **正常模式（NORMAL_MODE）**：用于进行当前时间与闹钟时间的匹配判断，并控制响铃和整点报时逻辑。
2. **闹钟设置模式（ALARM_SET_MODE）**：用于设置闹钟时间。用户可逐位调整小时和分钟的各位数，或直接将当前时间同步为闹钟时间。

接口说明：

信号名	方向	位宽	说明
clk	输入	1	系统时钟信号
rst_n	输入	1	异步复位信号，低有效
clk_1hz	输入	1	1Hz 节拍信号
set_time	输入	1	设置按键，按下时调节所选位
mode	输入	2	模式选择信号
pos	输入	3	当前设置位置（用于设置哪一位）
hour_tens~second_ones	输入	多位	当前系统时钟值各位
alarm_hour_tens~alarm_minute_ones	输出	多位	当前设定的闹钟时间
alarm_ring	输出	1	闹钟响铃输出信号（高电平有效）
time_ring	输出	1	整点报时信号（高电平有效）

本模块内部由两个 always 块构成，分别用于闹钟时间的设置与匹配判断、闹钟与报时信号的输出控制。

在设置闹钟时间时，系统首先判断 mode 是否为闹钟设置模式（ALARM_SET_MODE）。如果 pos 值为 3'b000，表示需要将当前时间直接赋值为闹钟时间。此操作方便用户快速同步设置，避免手动逐位设置的繁琐。而若 set_time 按键按下，模块则根据 pos 所指示的位置（小时十位、小时个位、分钟十位、分钟个位）分别进行逐位加 1 操作，并在达到各自上限时回绕至 0。例如小时十位最大为 2，分钟十位最大为 5，个位为 9。设置过程中 alarm_set 标志位被置为 1，表示当前已成功设定闹钟时间。

在正常模式下，模块会持续对当前时钟值和设置的闹钟时间进行比较。一旦时、分均相等，且秒为 00:00，表示闹钟时间到达，此时模块将清除 alarm_set 标志，同时在状态寄存器 alarm_ring_active 中置位，从而通过 alarm_ring 产生一次 1 秒钟高电平的响铃信号。该状态寄存器在下一个 clk_1hz 到来时自动清除，以确保输出脉冲宽度固定。

除了闹钟响铃外，模块还内置了整点报时功能。当系统当前时间满足“XX:00:00”条件（即分钟和秒都为 0），即判定为整点时刻。模块此时将 time_ring_active 状态寄存器置位，从而通过 time_ring 输出报时信号，同样在下一秒自动清除。

为防止在设置模式下出现误触发，模块在非正常模式下会主动将两个状态寄存器清零，避免由于时间巧合或状态滞留导致闹钟/报时错误触发。

7. 显示控制模块 (display_control)

本模块为数字时钟的显示控制核心模块，负责根据工作模式（正常显示、时钟设置、闹钟设置）选择显示的对应时间，主要以七段数码管实现最后一位以及其他位的 bcd 码输出，同时支持选中位置闪烁。

接口说明：

输入信号：

信号名称	说明
clk	主时钟信号，用于组合逻辑的时序控制
clk_1hz	1Hz 脉冲信号，用于每秒更新时间
rst_n	复位信号，低电平有效，复位后时间归零
mode	模式选择信号：00 正常显示，01 时钟设置，10 闹钟设置
sec_ones/sec_tens	时钟秒数据（个位 / 十位）
min_ones/min_tens	时钟分钟数据（个位 / 十位）
hour_ones/hour_tens	时钟小时数据（个位 / 十位）
pos	当前调整位置（0-6，对应时十位至秒个位）

输出信号：

信号名称	说明
seven_led	七段数码管显示信号
display_	对应各数的 bcd 码输出

display_control 模块是时钟的显示模块，基于 mode 信号实现多模式显示：正常模式显示时钟数据，根据实验台不同位数以不同逻辑输出显示信号，设置模式下选中位置（由 pos 指定）配合 clk_2hz 闪烁（输出全灭码或空白码），闹钟模式显示闹钟时分（秒固定为 0）。

8. 声音控制模块（ring）

ring 模块为声音控制核心模块，主要负责处理闹钟触发信号 ring1 和整点报时信号 ring2，根据不同触发信号生成升调、降调的铃声，并通过扬声器输出。。

接口说明

输入信号

信号名称	说明

clk	主时钟信号，用于时序逻辑控制
clk_1hz_posedge	1Hz 脉冲信号，用于每秒更新闹钟状态和音调变化
ring1	闹钟触发信号，高电平有效，优先级高于 ring2
ring2	整点报时触发信号，高电平有效

输出信号

信号名称	说明
tone_divide [1:0]	音调分频控制信号，通过不同取值（00-11）控制扬声器输出音调的频率
speaker	扬声器控制信号，输出对应方波

模块实现了声音的核心逻辑，采用优先级机制处理两种闹钟信号。当检测到 ring1 信号且其未处于活动状态时，立即激活声音信号，禁止其他声音响应，标记 ring1 为活动状态并将音调初始化。每秒通过 clk_1hz_posedge 信号更新音调：ring1 模式下音调从高到低逐秒递减，当降至最低音调（tone_divide=2'b00）时自动停止闹钟；ring2 模式下音调从低到高逐秒递增，升至最高音调（tone_divide=2'b11）时结束，通过状态标记和时序控制确保两种声音信号有序响应、避免冲突并且输出闹钟与整点报时的两种不同音调。

9. 总体运行流程

当置位状态 rst_n 被设置成高电平时，整个系统开始工作：

- **默认状态（正常模式 mode=00）**

系统首先进入默认状态，其功能如下：

  - 显示当前时间，时钟每秒递增
  - 整点自动触发报时（升调蜂鸣），与闹钟时间匹配则触发闹钟（降调蜂鸣）

- **设置时钟模式（mode=01）**

  - 按下 set_clock 进入，set_shift 切换设置位（时/分/秒）
  - set_time 对当前位加 1，当前位闪烁显示，实时更新当前时间

- **设置闹钟模式（mode=10）**

  - 按下 set_alarm 进入，set_shift 切换设置位（时/分）
  - set_time 对当前位加 1，设置位闪烁；在小时十位时按下可将当前时间快速设为闹钟时间

- **系统响应**

- 显示模块根据模式显示当前时间或闹钟时间，当进入设置时钟数据时，数码管进行闪烁提示
- 声音模块根据 time_ring 或 alarm_ring 控制输出升/降调蜂鸣音，并优先进行闹钟响铃

四、 详细代码

1. clock_top.v

```
module clock_top (
    input wire clk,           // 系统时钟
    input wire rst_n,         // 复位信号

    input wire set_clock,     // 设置时钟控制
    input wire set_alarm,     // 设置闹钟控制
    input wire set_shift_pre,     // 位置调整键
    input wire set_time_pre,      // 时间调整键

    output wire alarm_ring,          // 闹钟响铃
    output wire time_ring,           // 整点报时

    // 显示输出
    output wire sec_ten_zero,
    output wire min_ten_zero,
    output wire hour_ten_zero1,
    output wire hour_ten_zero2,

    output wire [6:0] seven_led,  // 秒个位七段 LED 输出
    output wire [2:0] display_sec_tens,
    output wire [3:0] display_min_ones,
    output wire [2:0] display_min_tens,
    output wire [3:0] display_hour_ones,
    output wire [1:0] display_hour_tens,
    output wire speaker
);
    wire clk_1hz;           // 1Hz 时钟信号
    wire clk_2hz;           // 2Hz 时钟信号
    wire set_shift;        // 位置调整信号
    wire set_time;         // 时间调整信号
```

```
wire [1:0] mode;        // 模式控制
wire [2:0] pos;         // 位置控制

// 时钟数据
wire [3:0] sec_ones;
wire [3:0] min_ones;
wire [3:0] hour_ones;
wire [2:0] sec_tens;
wire [2:0] min_tens;
wire [1:0] hour_tens;

// 闹钟数据
wire [1:0] alarm_hour_tens;
wire [3:0] alarm_hour_ones;
wire [2:0] alarm_minute_tens;
wire [3:0] alarm_minute_ones;

// 实例化 1Hz 时钟分频器 (修复语法错误)
Hz_12 u_Hz(
    .clk(clk),
    .clk_1hz(clk_1hz),
    .clk_2hz(clk_2hz),
);


key_handle u_key_handle(
    .clk(clk),
    .set_shift_pre(set_shift_pre),
    .set_time_pre(set_time_pre),
    .set_shift(set_shift),
    .set_time(set_time),
);
// 模式控制模块，分为设置时钟、设置闹钟和正常模式
mode_control u_mode_controller(
    .clk(clk),
    .rst_n(rst_n),
    .set_clock(set_clock),
    .set_alarm(set_alarm),
    .set_shift(set_shift),
    .mode(mode),
    .pos(pos)
);
```

```
    // 时钟控制模块
    clock_control u_clock_controller(
        .clk(clk),
        .clk_1hz(clk_1hz),
        .set_time(set_time),
        .rst_n(rst_n),
        .mode(mode),
        .pos(pos),
        .sec_ones(sec_ones),
        .sec_tens(sec_tens),
        .min_ones(min_ones),
        .min_tens(min_tens),
        .hour_ones(hour_ones),
        .hour_tens(hour_tens)
    );

    // 闹钟控制模块
    alarm_control u_alarm_controller(
        .clk(clk),
        .rst_n(rst_n),
        .clk_1hz(clk_1hz),          // 注意这里与注释中不同
        .set_time(set_time),        // 添加了时间调整信号
        .mode(mode),
        .pos(pos),
        .hour_tens(hour_tens),
        .hour_ones(hour_ones),
        .minute_tens(min_tens),
        .minute_ones(min_ones),
        .second_tens(sec_tens),
        .second_ones(sec_ones),
        .alarm_hour_tens(alarm_hour_tens),
        .alarm_hour_ones(alarm_hour_ones),
        .alarm_minute_tens(alarm_minute_tens),
        .alarm_minute_ones(alarm_minute_ones),
        .alarm_ring(alarm_ring),
        .time_ring(time_ring)
    );


    ring u_ring(
        .clk(clk),
        .clk_1hz_posedge(clk_1hz),
        .ring1(alarm_ring),
        .ring2(time_ring),
```

```
        .tone_divide(tone_divide),
        .speaker(speaker)
    );

    // 显示控制模块
display_control u_display_controller(
    .clk(clk),
    .rst_n(rst_n),
    .mode(mode),
    .sec_ones(sec_ones),
    .sec_tens(sec_tens),
    .min_ones(min_ones),
    .min_tens(min_tens),
    .hour_ones(hour_ones),
    .hour_tens(hour_tens),
    .alarm_hour_tens(alarm_hour_tens),
    .alarm_hour_ones(alarm_hour_ones),
    .alarm_minute_tens(alarm_minute_tens),
    .alarm_minute_ones(alarm_minute_ones),
    .pos(pos),
    .sec_ten_zero(sec_ten_zero),
    .min_ten_zero(min_ten_zero),
    .hour_ten_zero1(hour_ten_zero1),
    .hour_ten_zero2(hour_ten_zero2),
    .clk_1hz(clk_1hz),
    .clk_2hz(clk_2hz),
    .seven_led(seven_led),
    .display_sec_tens(display_sec_tens),
    .display_min_ones(display_min_ones),
    .display_min_tens(display_min_tens),
    .display_hour_ones(display_hour_ones),
    .display_hour_tens(display_hour_tens)
);
endmodule
```

2. mode_control.v

```
module mode_control (
    input wire clk,
    input wire rst_n,
    input wire set_clock,     // 设置时钟控制
```

```
    input wire set_alarm,     // 设置闹钟控制
    input wire set_shift,     // 位置调整键
    output reg [1:0] mode,    // 模式：00-正常，01-设置时钟，10-设置闹钟
    output reg [2:0] pos      // 位置：0-时十位，1-时个位，2-分十位，3-分个位，
4-秒十位，5-秒个位
);
    // 模式定义
    localparam NORMAL_MODE = 2'b00;
    localparam CLOCK_SET_MODE = 2'b01;
    localparam ALARM_SET_MODE = 2'b10;

    // 模式切换逻辑
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            mode <= NORMAL_MODE;
        end else begin
            if (set_clock) begin
                mode <= CLOCK_SET_MODE;
            end else if (set_alarm) begin
                mode <= ALARM_SET_MODE;
            end else if (!set_clock && !set_alarm && (mode !=
NORMAL_MODE)) begin
                mode <= NORMAL_MODE;
            end
            else
            begin
                // 保持当前模式
                mode <= mode;

            end
        end
    end

    // 位置控制逻辑
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            pos <= 3'd0;
        end else if (mode == NORMAL_MODE) begin
            pos <= 3'd0;
        end else if (set_shift) begin
            // 在设置模式下，按下位置调整键循环切换位置
            if(mode == CLOCK_SET_MODE)
            begin
            if (pos <= 3'd0)
```

```
                pos <= 3'd6;
            else
                pos <= pos - 1'b1;
            end
            else
            begin
            if (pos <= 3'd0)
                pos <= 3'd4;
            else
                pos <= pos - 1'b1;
            end
        end
    end
endmodule
```

3. key_handle.v

```
module key_handle(
    input wire clk,                 // 系统时钟
    input wire set_shift_pre,       // 位置调整键脉冲信号
    input wire set_time_pre,        // 时间调整键脉冲信号
    output reg set_shift,           // 位置调整键电平信号
    output reg set_time             // 时间调整键电平信号
);
    //将上升沿转换成一次一个时钟周期的电平信号
    // 信号延时一周期
    reg set_shift_pre_r;
    always @(posedge clk) begin
        set_shift_pre_r <= set_shift_pre;
    end
    //// 检测上升沿：当前为高电平且前一个时钟周期为低电平
    wire set_shift_pos = set_shift_pre & (~set_shift_pre_r);

    reg set_time_pre_r;
    always @(posedge clk) begin
        set_time_pre_r <= set_time_pre;
    end
    wire set_time_pos = set_time_pre & (~set_time_pre_r); // 上升沿检测

    // 将上升沿转换为电平信号
    always @(posedge clk) begin
        if(set_shift_pos)
```

```
            set_shift <= 1'b1;
        else
            set_shift <= 1'b0;
    end

    always @(posedge clk) begin
        if(set_time_pos)
            set_time <= 1'b1;
        else
            set_time <= 1'b0;
    end

endmodule
```

4. Hz_12.v

```
module Hz_12 (
    input  wire clk,        // 输入时钟 (1kHz)
    output reg clk_1hz,     // 1Hz 输出
    output reg clk_2hz      // 2Hz 输出
);
    reg [9:0] counter;

    // 使用单个计数器为两个时钟分频
    always @(posedge clk) begin
        clk_1hz <= 1'b0;
        if (counter >= 10'd999) begin
            counter <= 10'd0;
            // 1Hz 时钟脉冲赋值
            clk_1hz <= 1'b1;
            // 2Hz 时钟电平翻转
            clk_2hz <= ~clk_2hz;
        end else if (counter == 10'd499) begin
            clk_2hz <= ~clk_2hz;
            counter <= counter + 1'b1;
        end else begin
            counter <= counter + 1'b1;
        end
    end

    // 初始化
```

```
    initial begin
        counter = 10'd0;
        clk_1hz = 1'b0;
        clk_2hz = 1'b0;
    end

endmodule
```

5. clock_control.v

```
module clock_control (
    input wire clk,
    input wire clk_1hz,
    input wire set_time,     // 时间调整键（按下后当前选中位+1）
    input wire rst_n,
    input wire [1:0] mode,   // 模式控制
    input wire [2:0] pos,    // 位置控制

    output reg [3:0] sec_ones,  // 秒个位
    output reg [2:0] sec_tens,  // 秒十位
    output reg [3:0] min_ones,  // 分个位
    output reg [2:0] min_tens,  // 分十位
    output reg [3:0] hour_ones, // 时个位
    output reg [1:0] hour_tens  // 时十位
);
    // 模式定义
    localparam NORMAL_MODE = 2'b00;
    localparam CLOCK_SET_MODE = 2'b01;

    // 时钟更新逻辑
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // 复位时时间为 00:00:00
            sec_ones <= 4'd0;
            sec_tens <= 3'd0;
            min_ones <= 4'd0;
            min_tens <= 3'd0;
            hour_ones <= 4'd0;
            hour_tens <= 2'd0;
        end else begin
            // 按键时钟设置，每一位都是一个计数器
```

```
            if (mode == CLOCK_SET_MODE && set_time) begin
                case (pos)
                    3'd6: begin // 秒个位
                        if (sec_ones == 4'd9)
                            sec_ones <= 4'd0;
                        else
                            sec_ones <= sec_ones + 1'b1;
                    end
                    3'd5: begin // 秒十位
                        if (sec_tens == 3'd5)
                            sec_tens <= 3'd0;
                        else
                            sec_tens <= sec_tens + 1'b1;
                    end
                    3'd4: begin // 分个位
                        if (min_ones == 4'd9)
                            min_ones <= 4'd0;
                        else
                            min_ones <= min_ones + 1'b1;
                    end
                    3'd3: begin // 分十位
                        if (min_tens == 3'd5)
                            min_tens <= 3'd0;
                        else
                            min_tens <= min_tens + 1'b1;
                    end
                    3'd2: begin // 时个位
                        if (hour_tens == 2'd2 && hour_ones == 4'd3)
                            hour_ones <= 4'd0;
                        else if (hour_ones == 4'd9)
                            hour_ones <= 4'd0;
                        else
                            hour_ones <= hour_ones + 1'b1;
                    end
                    3'd1: begin // 时十位
                        if (hour_tens == 2'd2)
                            hour_tens <= 2'd0;
                        else
                            hour_tens <= hour_tens + 1'b1;
                        // 如果从 1 变为 2，且小时个位>3，需要修正
                        if (hour_tens == 2'd1 && hour_ones > 4'd3)
                            hour_ones <= 4'd3;
                    end
                endcase
```

```
            end
            // 正常模式下时钟运行，只在 clk_1hz 上升沿时计数
            else if (mode!= CLOCK_SET_MODE&&clk_1hz) begin//就减去一个
mode 判断条件，多了 9 个资源
                // 秒个位进位
                if (sec_ones == 4'd9) begin
                    sec_ones <= 4'd0;

                    // 秒十位进位
                    if (sec_tens == 3'd5) begin
                        sec_tens <= 3'd0;

                        // 分个位进位
                        if (min_ones == 4'd9) begin
                            min_ones <= 4'd0;

                            // 分十位进位
                            if (min_tens == 3'd5) begin
                                min_tens <= 3'd0;

                                // 时位进位
                               if (hour_tens == 2'd2 && hour_ones == 4'd3)
begin
                                    hour_tens <= 2'd0;
                                    hour_ones <= 4'd0;
                                end else if (hour_ones == 4'd9) begin
                                    hour_ones <= 4'd0;
                                    hour_tens <= hour_tens + 1'b1;
                                end else begin
                                    hour_ones <= hour_ones + 1'b1;
                                end
                            end else begin
                                min_tens <= min_tens + 1'b1;
                            end
                        end else begin
                            min_ones <= min_ones + 1'b1;
                        end
                    end else begin
                        sec_tens <= sec_tens + 1'b1;
                    end
                end else begin
                    sec_ones <= sec_ones + 1'b1;
                end
            end
```

```
        end
    end
endmodule
```

6. alarm_control.v

```
module alarm_control (
    input wire clk,
    input wire rst_n,
    input wire clk_1hz,           // 1Hz 时刻脉冲信号
    input wire set_time,          // 时间调整键
    input wire [1:0] mode,        // 模式控制
    input wire [2:0] pos,         // 位置控制

    // 当前时钟值
    input wire [1:0] hour_tens,
    input wire [3:0] hour_ones,
    input wire [2:0] minute_tens,
    input wire [3:0] minute_ones,
    input wire [2:0] second_tens,
    input wire [3:0] second_ones,

    // 闹钟设置值输出
    output reg [1:0] alarm_hour_tens,
    output reg [3:0] alarm_hour_ones,
    output reg [2:0] alarm_minute_tens,
    output reg [3:0] alarm_minute_ones,
    // output reg [2:0] alarm_second_tens,
    // output reg [3:0] alarm_second_ones,
    // output reg alarm_set, // 闹钟设置标志

    // 闹钟响铃和整点报时输出
    output reg alarm_ring,
    output reg time_ring
);
    localparam NORMAL_MODE = 2'b00; // 假设正常模式是 00
    localparam ALARM_SET_MODE = 2'b10;

    // 移除计数器 reg [3:0] alarm_counter;
    // 移除计数器 reg [3:0] time_counter;
```

```
    // 状态寄存器，用于保持响铃/报时状态一秒
    reg alarm_set = 1'b0; // 闹钟设置标志
    reg alarm_ring_active = 1'b0;
    reg time_ring_active = 1'b0;

    // // 闹钟有效标志 (保持不变)
    // wire alarm_valid = ((alarm_hour_tens < 2'd2) ||
    //                     (alarm_hour_tens == 2'd2 && alarm_hour_ones <=
4'd3)) &&
    //                     (alarm_minute_tens <= 3'd5) &&
    //                     (alarm_minute_ones <= 4'd9) &&
    //                     (alarm_second_tens <= 3'd5) &&
    //                     (alarm_second_ones <= 4'd9);

    // 闹钟设置逻辑 (保持不变)
    always @(posedge clk or negedge rst_n) begin
        // ... (内容不变) ...
        if (!rst_n) begin
            // 复位闹钟设置为非法值(25:00:00)，确保初始未设置时不会响
            alarm_hour_tens <= 2'd0;
            alarm_hour_ones <= 4'd0;
            alarm_minute_tens <= 3'd0;
            alarm_minute_ones <= 4'd0;
            // alarm_second_tens <= 3'd0;
            // alarm_second_ones <= 4'd0;
            alarm_set <= 1'b0; // 复位闹钟设置标志
        // ...
        end
        // else if (mode == NORMAL_MODE && alarm_set == 1'b0)//这里能占3
个
        // begin
        //     alarm_hour_ones <= hour_ones; // 直接使用当前时钟值
        //     alarm_hour_tens <= hour_tens; // 直接使用当前时钟值
        //     alarm_minute_ones <= minute_ones; // 直接使用当前时钟值
        //     alarm_minute_tens <= minute_tens; // 直接使用当前时钟值
        // end
        else if(mode == ALARM_SET_MODE && pos == 3'd0)begin
            alarm_hour_ones <= hour_ones; // 直接使用当前时钟值
            alarm_hour_tens <= hour_tens; // 直接使用当前时钟值
            alarm_minute_ones <= minute_ones; // 直接使用当前时钟值
            alarm_minute_tens <= minute_tens; // 直接使用当前时钟值
        end
        else if (mode == ALARM_SET_MODE && set_time) begin
            // 在闹钟设置模式下，根据位置调整闹钟
```

```
            alarm_set <= 1'b1; // 设置闹钟标志
            case (pos) // pos 的值从 0 递增
            3'd1: begin // pos=0 对应 时十位
                if (alarm_hour_tens == 2'd2)
                    alarm_hour_tens <= 2'd0;
                else
                    alarm_hour_tens <= alarm_hour_tens + 1'd1;
                // 修正超出范围的小时值 (应该在设置时个位时处理更佳)
                // if (alarm_hour_tens == 2'd2 && alarm_hour_ones >
4'd3)
                //     alarm_hour_ones <= 4'd3;
            end
            3'd2: begin // pos=1 对应 时个位
                if (alarm_hour_tens == 2'd2 && alarm_hour_ones == 4'd3)
                    alarm_hour_ones <= 4'd0;
                else if (alarm_hour_ones == 4'd9)
                    alarm_hour_ones <= 4'd0;
                else
                    alarm_hour_ones <= alarm_hour_ones + 1'd1;
            end
            3'd3: begin // pos=2 对应 分十位
                if (alarm_minute_tens == 3'd5)
                    alarm_minute_tens <= 3'd0;
                else
                    alarm_minute_tens <= alarm_minute_tens + 1'd1;
            end
            3'd4: begin // pos=3 对应 分个位
                if (alarm_minute_ones == 4'd9)
                    alarm_minute_ones <= 4'd0;
                else
                    alarm_minute_ones <= alarm_minute_ones + 1'd1;
            end
            // 注意:如果已经移除了闹钟秒设置,pos 在 ALARM_SET_MODE 下不会等于
4 或 5
            // 如果仍然保留秒设置，则需要添加:
            /*
            3'd4: begin // pos=4 对应 秒十位
                if (alarm_second_tens == 3'd5)
                    alarm_second_tens <= 3'd0;
                else
                    alarm_second_tens <= alarm_second_tens + 1'd1;
            end
            3'd5: begin // pos=5 对应 秒个位
                if (alarm_second_ones == 4'd9)
```

```
                    alarm_second_ones <= 4'd0;
                else
                    alarm_second_ones <= alarm_second_ones + 1'd1;
            end
            */
            default: ; // 处理未预期的 pos 值
            endcase
        end
        else if(mode == NORMAL_MODE && alarm_set)
        begin
            if (hour_tens == alarm_hour_tens &&
                    hour_ones == alarm_hour_ones &&
                    minute_tens == alarm_minute_tens &&
                    minute_ones == alarm_minute_ones &&
                    // 根据是否设置闹钟秒来决定是否比较秒
                    second_tens == 3'b0 && // 比较秒十位
                    second_ones == 4'b0) begin // 比较秒个位
                    // 闹钟时间到达，清除设置标志 (响铃由第二个 always 块处理)
                    alarm_set <= 1'b0;
                end
        end
// ...
    end

    // 闹钟响铃和整点报时控制 - 使用状态寄存器和 clk_1hz 脉冲
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            alarm_ring <= 1'b0;
            time_ring <= 1'b0;
            alarm_ring_active <= 1'b0; // 复位状态
            time_ring_active <= 1'b0; // 复位状态
        end else begin
            // 默认输出等于状态寄存器的值
            alarm_ring <= alarm_ring_active;
            time_ring <= time_ring_active;

            // 当 1Hz 脉冲到来时，清除激活状态
            if (clk_1hz) begin
                alarm_ring_active <= 1'b0;
                time_ring_active <= 1'b0;
            end

            // 检查触发条件 (这会覆盖同一时钟周期的 clk_1hz 清除操作,如果条件满足
的话)
```

```
            if (mode == NORMAL_MODE) begin
                // 处理闹钟响铃触发
                if (
                    hour_tens == alarm_hour_tens &&
                    hour_ones == alarm_hour_ones &&
                    minute_tens == alarm_minute_tens &&
                    minute_ones == alarm_minute_ones &&
                    second_tens == 3'b0 &&
                    second_ones == 4'b0 &&
                    alarm_set) begin
                    alarm_ring_active <= 1'b1; // 激活闹钟状态
                end

                // 处理整点报时触发
                if (minute_tens == 3'd0 && minute_ones == 4'd0 &&
                    second_tens == 3'd0 && second_ones == 4'd0) begin
                    time_ring_active <= 1'b1; // 激活报时状态
                end
            end else begin
                // 在设置模式下，确保状态不被激活 (即使时间恰好匹配)
                // 并且如果因为模式切换导致 clk_1hz 没有清除状态,这里也强制清除
                 alarm_ring_active <= 1'b0;
                 time_ring_active <= 1'b0;
            end
        end
    end
endmodule
```

7. display_control.v

```
module display_control(
    input wire clk,
    input wire rst_n,
    input wire [1:0] mode,          // 模式: 00-正常, 01-设置时钟, 10-设置闹钟

    // 时钟数据
    input wire [3:0] sec_ones,
    input wire [2:0] sec_tens,
    input wire [3:0] min_ones,
    input wire [2:0] min_tens,
    input wire [3:0] hour_ones,
```

```
    input wire [1:0] hour_tens,

    // 闹钟数据
    input wire [1:0] alarm_hour_tens,
    input wire [3:0] alarm_hour_ones,
    input wire [2:0] alarm_minute_tens,
    input wire [3:0] alarm_minute_ones,
    // input wire [2:0] alarm_second_tens,
    // input wire [3:0] alarm_second_ones,

    input wire [2:0] pos,           // 当前位置
    input wire clk_1hz,             // 1Hz 时钟
    input wire clk_2hz,             // 2Hz 时钟

    // 零位抑制控制
    output wire sec_ten_zero,       // 秒十位为 0 时置 1
    output wire min_ten_zero,       // 分十位为 0 时置 1
    output wire hour_ten_zero1,     // 时十位为 0 时置 1(显示时)
    output wire hour_ten_zero2,     // 时十位为 0 时置 1(设置时)

    // 显示输出
    output reg [6:0] seven_led,     // 秒个位七段 LED
    output reg [2:0] display_sec_tens,
    output reg [3:0] display_min_ones,
    output reg [2:0] display_min_tens,
    output reg [3:0] display_hour_ones,
    output reg [1:0] display_hour_tens
);
    // 模式定义
    localparam NORMAL_MODE = 2'b00;
    localparam CLOCK_SET_MODE = 2'b01;
    localparam ALARM_SET_MODE = 2'b10;

    // 闪烁常量定义
    localparam BCD_BLANK = 4'd15;  // BCD 码闪烁时的显示值



    // 闪烁使能信号 - 当处于设置模式且为选中位时有效
    wire blink_enable = (mode != NORMAL_MODE) && clk_2hz;

    // 位置对应的闪烁控制 (1 为闪烁)
    wire hour_tens_blink = blink_enable && (pos == 3'd1);
    wire hour_ones_blink = blink_enable && (pos == 3'd2);
```

```
    wire min_tens_blink = blink_enable && (pos == 3'd3);
    wire min_ones_blink = blink_enable && (pos == 3'd4);
    wire sec_tens_blink = blink_enable && (pos == 3'd5);
    wire sec_ones_blink = blink_enable && (pos == 3'd6);

    // 零位抑制 - 特定位为 0 时输出 1
    assign sec_ten_zero = ((mode == NORMAL_MODE) ? 3'd0 : ((pos ==
3'd5)&&sec_tens_blink));
    assign min_ten_zero = ((mode == NORMAL_MODE) ? 3'd0 : ((pos ==
3'd3)&&min_tens_blink));
    assign hour_ten_zero1 = ((mode == NORMAL_MODE) ? 2'd0 : ((pos ==
3'd1)&&hour_tens_blink));
    assign hour_ten_zero2 = ((mode == NORMAL_MODE) ? 2'd0 : ((pos ==
3'd1)&&hour_tens_blink));

    // 7 段译码器 - 将 BCD 码转为 7 段显示码（低电平有效）
    function [6:0] bcd_to_seg;
        input [3:0] bcd;
        begin
            case (bcd)
                4'd0: bcd_to_seg = 7'b0111111; // 0
                4'd1: bcd_to_seg = 7'b0000110; // 1
                4'd2: bcd_to_seg = 7'b1011011; // 2
                4'd3: bcd_to_seg = 7'b1001111; // 3
                4'd4: bcd_to_seg = 7'b1100110; // 4
                4'd5: bcd_to_seg = 7'b1101101; // 5
                4'd6: bcd_to_seg = 7'b1111100; // 6
                4'd7: bcd_to_seg = 7'b0000111; // 7
                4'd8: bcd_to_seg = 7'b1111111; // 8
                4'd9: bcd_to_seg = 7'b1100111; // 9
                default: bcd_to_seg = 7'b0000000; // 全灭
            endcase
        end
    endfunction

    // 秒个位显示 - 七段译码
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            seven_led <= 7'b1000000; // 显示 0
        end else begin
            // 秒个位显示 - 根据模式选择数据源
            if (mode == ALARM_SET_MODE)
                // 闹钟模式 - 闪烁时熄灭，否则显示闹钟值
```

```
                seven_led <= sec_ones_blink ? 7'b0000000 :
bcd_to_seg(4'b0); // 闹钟秒个位为 0
            else
                // 普通或设置时钟模式 - 闪烁时熄灭，否则显示时钟值
                seven_led <= sec_ones_blink ? 7'b0000000 :
bcd_to_seg(sec_ones);
        end
    end

    // 其他位显示 - 直接输出 BCD
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            display_sec_tens <= 3'd0;
            display_min_ones <= 4'd0;
            display_min_tens <= 3'd0;
            display_hour_ones <= 4'd0;
            display_hour_tens <= 2'd0;
        end else begin
            // 秒十位 (3 位转为 4 位 BCD 码输出，闪烁时为全 1)
            if (mode == ALARM_SET_MODE)
                display_sec_tens <= 4'd0; // 闹钟模式下不显示秒十位
            else
                display_sec_tens <= sec_tens_blink ? BCD_BLANK[2:0] :
sec_tens;

            // 分个位 (本身就是 4 位 BCD 码)
            if (mode == ALARM_SET_MODE)
                display_min_ones <= min_ones_blink ? BCD_BLANK :
alarm_minute_ones;
            else
                display_min_ones <= min_ones_blink ? BCD_BLANK :
min_ones;

            // 分十位 (3 位转为 4 位 BCD 码输出，闪烁时为全 1)
            if (mode == ALARM_SET_MODE)
                display_min_tens <= min_tens_blink ? BCD_BLANK[2:0] :
alarm_minute_tens;
            else
                display_min_tens <= min_tens_blink ? BCD_BLANK[2:0] :
min_tens;

            // 时个位 (本身就是 4 位 BCD 码)
            if (mode == ALARM_SET_MODE)
```

```
                display_hour_ones <= hour_ones_blink ? BCD_BLANK :
alarm_hour_ones;
            else
                display_hour_ones <= hour_ones_blink ? BCD_BLANK :
hour_ones;

            // 时十位 (2 位转为 4 位 BCD 码输出，闪烁时为全 1)
            if (mode == ALARM_SET_MODE)
                display_hour_tens <= hour_tens_blink ? BCD_BLANK[1:0] :
alarm_hour_tens;
            else
                display_hour_tens <= hour_tens_blink ? BCD_BLANK[1:0] :
hour_tens;
        end
    end

endmodule
```

8. ring.v

```
module ring(
    input wire clk,       // 时钟信号
    input wire clk_1hz_posedge,
    input wire ring1,
    input wire ring2,      // ring 信号
    // output reg [1:0] tone_divide, // 音调分频器
    output reg speaker    // 扬声器控制信号?

);

    // 闹钟 1k，最小频率?260 模最大约 20
    reg [3:0] counter;

    // 音调   时钟假设 10kHz，此时对应 330
    reg [1:0] tone_divide;

    reg ban;
    reg ring;
    // reg [1:0] tone_timer; // 计时

    reg ring1_active; // 跟踪 ring1 是否处于活动状态
```

```
    reg ring2_active; // 跟踪 ring2 是否处于活动状态

    initial begin
        ring1_active = 1'b0;
        ring2_active = 1'b0;
        ban = 1'b0;
        ring = 1'b0;
        tone_divide = 2'b00;
        counter = 4'b0000;
        speaker = 1'b0;
    end

    // 闹钟控制逻辑 - 优先处理 ring1
    always @(posedge clk) begin
        if (ring1 && !ring1_active) begin
            // ring1 触发，开始闹铃，优先级高
            ring <= 1'b1;
            ban <= 1'b1;
            ring1_active <= 1'b1;
            ring2_active <= 1'b0; // 确保 ring2 不活动
            tone_divide <= 2'b11;  // 开始使用第一个频率(330Hz)
        // end else if (!ring1) begin
        //     ring1_active <= 1'b0; // 重置 ring1 活动状态
        end else if (ring2 && !ban && !ring2_active && !ring1_active)
begin
            // 只有当 ban 为 0 且 ring1 不活动时，ring2 才能激活
            ring <= 1'b1;
            ban <= 1'b1;
            ring2_active <= 1'b1;
            tone_divide <= 2'b00;  // ring2 使用不同的起始频率
        // end else if (!ring2) begin
        //     ring2_active <= 1'b0; // 重置 ring2 活动状态
        end else if (clk_1hz_posedge) begin
            if (ring1_active) begin
                // ring1 模式的频率变化
                if (tone_divide == 2'b00) begin
                    ring <= 1'b0;
                    ban <= 1'b0;
                    ring1_active <= 1'b0;
                end else begin
                    tone_divide <= tone_divide - 1'b1;  // 逐渐降低频率
                end
            end else if (ring2_active) begin
                // ring2 模式的频率变化
```

```
                if (tone_divide == 2'b11) begin
                    ring <= 1'b0;
                    ban <= 1'b0;
                    ring2_active <= 1'b0;
                end else begin
                    tone_divide <= tone_divide + 1'b1;  // 逐渐增高频率
                end
            end
        end
    end
    // 计数器逻辑
    always @(posedge clk) begin
        // if (!ring1 && !ring2) begin
        if(!ring) begin
            counter <= 0;
        end else begin
            if (counter == tone_divide+1'b1) begin
                counter <= 0;
            end else begin
                counter <= counter + 1;
            end
        end
    end

    // 扬声器逻辑
    always @(posedge clk) begin
        if (!ring) begin
            speaker <= 1'b0;
        end else begin
            speaker <= (counter <= (tone_divide+1'b1)/2)? 1'b1 : 1'b0;
        end
    end

endmodule
```

五、 调试过程中问题及讨论

1. 为什么仿真和上机调试的显示不同？

在实验过程中，我们发现仿真结果与上机调试显示不一致，输出信号异常。起初怀疑是代码逻

辑或引脚配置错误，因此逐步检查了时钟输入、复位信号、引脚约束和仿真激励波形，均未发现明显问题。随后尝试多次重新编译和下载，观察问题依然存在。经过反复对比调试环境，最终确认是由于使用 Programmer 工具下载代码时，未选择最新生成的 pof 文件，导致开发板运行的是旧版本逻辑。更新正确的 pof 文件后，实验现象与仿真结果一致，问题解决。

2. 为什么在自己机器上可以编译的程序在实验室电脑上不能正常编译，反而资源超限？

在实验的最后阶段，我们还遇到在个人电脑上可以正常编译的工程，在实验室电脑上却出现资源超限报错的问题。为查明原因，我们对比了两台设备的 FPGA 软件版本、目标芯片设置、优化选项和工程配置文件。逐步排查后发现，实验室电脑上缺失部分工程配置文件，导致默认设置导致资源使用增加。由于编译环境的细微差异可能影响综合结果，最终我们采用直接将已编译好的 pof 文件传至实验室设备的方式成功完成下载和测试。

六、 设计调试小结

本电子钟系统基于 Verilog 硬件描述语言，运用自顶向下的模块化设计方法，整合时钟分频、时间控制、按键处理、模式控制、显示控制、声音控制等 6 大核心模块，成功实现 24 小时制计时、双模式时间与闹钟设置、整点报时及闹钟响铃等功能，同时通过数码管闪烁提示和不同铃声音乐进行交互优化。在完整的设计流程中，从需求分析明确功能目标，到模块设计细化各部分逻辑，再经仿真调试验证模块功能与协同性，最后进行硬件验证实现系统落地，这一过程极大深化了对数字系统设计的理解。

实验过程中，团队成员不断优化逻辑、优化模块设计、优化资源占用，以达到工程实验的实际效果落地，克服了环境差异导致的编译问题、逻辑错误引发的功能异常、文件上传的问题、资源超限等，通过协作与技术排查逐一解决，不仅积累了宝贵的工程经验，也完成了对 Verilog 语言编程和 FPGA 开发技能的实践检验，实现了从理论知识到实际应用的系统性训练，为后续复杂数字系统的设计与开发奠定了坚实基础。

三、课题二：药片装瓶系统设计

一、需求分析

药片装瓶控制系统应能够实现的功能以及约束条件如下：

基本功能需求：
1. 有清零状态、设置状态和工作状态，能够实现状态指示和状态间正确切换设置。
2. 工作状态能够同时显示药瓶以及药片数量。
3. 有告警指示，当用户做出错误操作时能够告警。

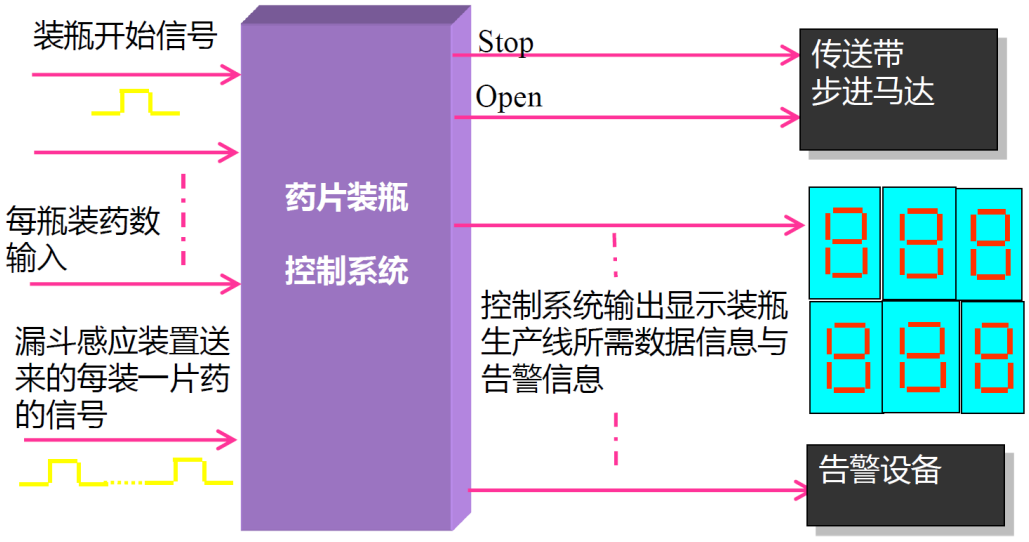
额外功能需求：
1. 实现设置状态时闪烁显示
2. 工作状态时可以切换显示初始设置
3. 可以设定每瓶药片数以及药片数量限定

约束条件：
1. 需要能够与外接设备的信号传递兼容
2. 需要令用户的操作尽可能简便
3. 实验使用的 CPLD 芯片存在一定的资源限制，需要查资料精简设计和优化硬件语言的表达，在实现基本功能需求的基础上最大限度地实现额外功能。

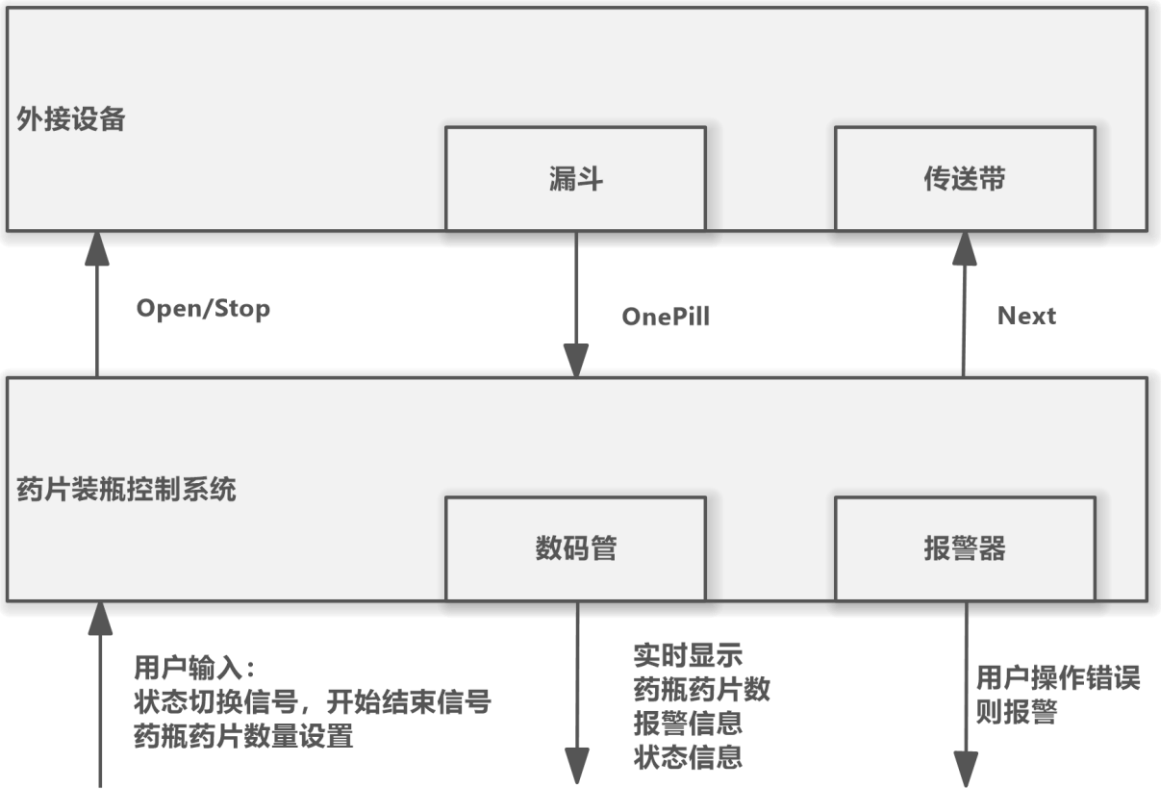
二、概要设计

1. 整体架构与输入输出

为了明确输入输出，首先必须说明药片装瓶控制系统的外部状况。课件给出的环境如下：



我们的系统在课件的基础上进行考虑，设计出整个药片装瓶系统的架构如下，由外接设备和控制系统来完成。



外接设备：

由于没有实际设备，不做考虑，仅作如下的假设：从控制系统接收 Open、Stop 信号控制开始和结束，接收到 Next 信号时传送带移动将下一个药瓶对准漏斗，漏斗含有光传感器，每检测到一个药片就发出一个 OnePill 信号。

控制系统：

能够调控要装的总瓶数，每瓶所装的药片数，启动、暂停或关闭时能够发出对应 Open、Stop 信号，药瓶装满时能够发出 Next 信号，接收到 OnePill 信号时药片计数加一，按照设置的数值控制外接设备工作，完成装瓶。

结合上述设计，得到药片装瓶控制系统的输入如下：

信号名称	说明
clk	系统时钟 1hz
warning_inclk	警告时钟输入 1000hz
QD	状态切换输入
reset_state	状态重置信号（K15）
display_setting	显示设置模式切换（K0）
suspend	暂停操作信号（K1）
PULSE	脉冲信号，用于设置值增加
CLR	清除信号，用于设置值清零
inc_five	增加 5 个单位的信号（K2）

药片装瓶控制系统的输出如下：

信号名称	说明
state_LED[6:0]	状态 LED 显示
bottle_LED_1[3:0]、bottle_LED_0[3:0]	药瓶数量的十位和个位显示
pill_LED_1[3:0]、pill_LED_0[3:0]	药片数量的十位和个位显示
error_display[3:0]	错误显示
warning_outclk	警告输出，蜂鸣器响

整体工作流程：

系统先通过用户设置目标药瓶数和每瓶药片数，启动后自动计数灌装的药片，当一个药瓶装满后提示更换，全部完成后进入报告状态，报告已装的药瓶数药片数。系统具备暂停/继续功能，可随时中断或恢复操作，并内置异常检测机制，能够识别无效设置并发出警告。

2. 模块划分

药片装瓶系统主要用于精确控制和监测药片灌装过程。系统由主控模块、计数器模块、显示模块、设置修改模块、状态显示模块等功能单元组成，采用模块化设计思想，各模块功能明确，接口简洁。通过不同的 LED 显示方式，系统能直观地向用户传达当前工作状态和灌装进度。

本组将易于区分的功能列出如下：

计数功能：准确记录并显示药片数量和药瓶数量，支持设定目标值。

显示功能：通过 LED 数码管实时显示当前灌装状态、设置参数和错误信息。

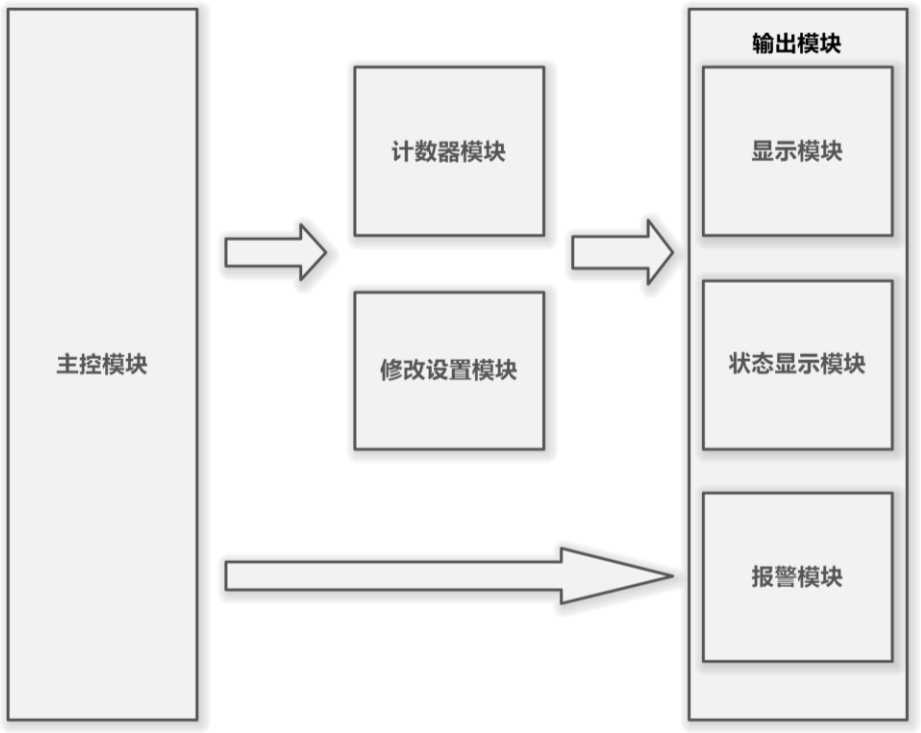
控制功能：支持启动、暂停、继续和复位操作，实现灌装过程的灵活控制。

设置功能：允许用户设置目标药瓶数量和每瓶药片数量，支持单步和快速增加两种调整方式。

状态指示：通过不同图形/字符直观显示系统工作状态(运行、暂停、换瓶、完成等)。

异常处理：检测无效设置和不当操作，发出警告信号并阻止错误继续。

按照以上功能进行模块划分，得到如下的结构：



主控模块(Pill_Bottling_System)：系统的顶层模块，负责协调各个子模块的工作
计数器模块(Counter_Module)：负责药片和药瓶的计数功能
显示模块(Display_Module)：处理系统状态、药片数量和药瓶数量的显示
设置修改模块(Modify_Setting_Module)：用于修改目标药片数和药瓶数的设置
状态显示模块(State_Display_Module)：显示系统当前工作状态
报警模块(Alarm_Module)：用于系统异常状态的报警提示

3. 状态机设计

系统运行过程中有两个状态机：主状态机和设置状态机。
主状态机包含三种主要状态：初始状态(无操作)、运行状态(执行灌装)和报告状态(完成罐装)，用于控制装瓶的开始和结束；设置状态机也包含三种主要状态：无设置状态，瓶数设置状态，片数设置状态。
两种状态机均接受 QD 正边沿进行转换，设置信号 display_setting 用于切换两种状态机，高电平时只能操作设置状态机，低电平时只能操作主状态机。

1. 主状态机

状态定义：主状态机定义了三个状态，通过 2 位寄存器 current_state 和 next_state 表示：

s_zero（2'b00）：初始/空闲状态，表示系统未运行或复位状态。
s_operation（2'b01）：操作状态，表示系统正在进行药片计数和装瓶。
s_report（2'b11）：报告状态，用于完成操作后报告当前装瓶数。

触发信号：

时钟信号：QD（正边沿触发），用于状态转换。
复位信号：reset_state（低电平有效），将状态机复位到 s_zero。

2.设置状态机

状态定义：设置状态机同样定义了三个状态，通过 2 位寄存器 current_display_state 和 next_display_state 表示：

d_standard（2'b00）：无设置模式，显示当前计数信息。
d_setting_bottle（2'b01）：瓶数设置模式，用于设置目标药瓶数量。
d_setting_pill（2'b11）：片数设置模式，用于设置目标药片数量。

触发信号：

时钟信号：QD（正边沿触发）。
显示设置信号：display_setting（低电平触发），用于复位到 d_standard。

三、设计详解

逐个对模块进行设计的详细介绍：

1. 主模块（Pill_Bottling_System）

1.1 功能介绍

主模块实现了两个自动机的处理逻辑，并实现了所有模块的集成，通过内部信号连接它们，系统状态信号从主模块传递到计数器模块和状态显示模块，计数器模块的输出（药瓶数、药片数）传递到显示模块，设置修改模块的输出（目标药瓶数、目标药片数）传递到计数器模块和显示模块。自动机实现为摩尔型状态机，采用两段式的写法，将状态转移单独写一个模块，状态的操作和判断写到另一个模块中，即两个 always 块。因此，两个状态机共计四个 always 块。

主模块的状态判断中包括报警信息的判断，该模块定义了 warning_flag 两位报警标志位，用于指示系统运行中的特定异常或不当操作。这些标志位通过组合逻辑设置，并在 Warning_Module 和 Display_Module 中被使用，用于驱动报警指示灯或在显示屏上显示错误信息。

具体来说，warning_flag[0]在主状态机处于 s_zero（空闲/初始化）状态且系统处于操作模式时被置位。如果此时目标瓶数或目标药片数中的任何一个为零，则 warning_flag[0]置为 1，表示操作前未设置目标数量的警告，阻止系统进入操作状态。而 warning_flag[1]则在主状态机处于 s_operation（操作中）状态时被置位，这意味着当系统正在执行装瓶任务时，warning_flag[1]置为 1，发出操作进行中，禁止修改设置的警告，防止用户在关键操作期间进行参数调整。

1.2 输入输出接口

主模块的输入、输出与概要设计中的输入输出相同，不再赘述。

1.3 局部变量介绍

用于存储状态机状态的变量：

```
reg   [1:0]current_state,next_state;
parameter s_zero=2'b00,s_operation=2'b01,s_report=2'b11;
reg         [1:0]current_display_state,next_display_state;
parameter d_standard=2'b00,d_setting_bottle=2'b01,d_setting_pill=2'b11;
wire [1:0]flash;
```

这里存储的是当前的药瓶数药片数，以及目标药瓶数药片数。

```
wire        [5:0]bottle_num;
wire        [5:0]pill_num;
wire        [5:0]target_bottle_num;
wire        [5:0]target_pill_num;
wire next_bottle;
wire finish;
```

下面的 2 位寄存器用于警报标识

```
reg   [1:0]warning_flag;
```

2. 计数器模块(Counter_Module)

2.1 功能介绍

计数器模块用于控制药瓶和药片的计数过程。它通过输入时钟信号 in_clk、状态信号 in_state、暂停信号 in_suspend 以及目标瓶数 in_target_bottle_num 和目标药片数 in_target_pill_num，管理输出寄存器 out_bottle_num（当前瓶数）、out_pill_num（当前药片数）、out_finish（完成标志）和 out_next_bottle（下一瓶标志）。

模块根据 in_state 的值在三种状态下操作：s_zero 将所有输出清零；s_operation 在非暂停时递增药片数，并在药片数达到目标时递增瓶数并重置药片数，同时置位 out_next_bottle；当瓶数达到目标时，置位 out_finish 表示完成。其余情况（default）则将 out_next_bottle 和 out_finish 清零，确保模块在无效状态下保持稳定输出。

支持暂停(suspend)功能，暂停时计数不增加。

计数达到要求状态后自动停止增加。

2.2 输入输出接口

输入接口：

信号名称	说明
in_clk	时钟输入
in_state[1:0]	系统状态输入
in_suspend	暂停信号（1=暂停，0=继续）
in_target_bottle_num[5:0]	目标药瓶数量
in_target_pill_num[5:0]	目标药片数量

输出接口：

信号名称	说明
out_bottle_num[5:0]	当前已装瓶数量
out_pill_num[5:0]	当前瓶中已装药片数量
out_finish	完成信号，当达到目标瓶数时置 1
out_next_bottle	下一个药瓶信号，当一个药瓶装满时置 1

3. 显示模块(Display_Module)

3.1 功能介绍

显示模块用于根据输入信号控制瓶数和药片数的 LED 显示输出。它接收显示设置 in_display_setting、闪烁控制 in_flash、时钟信号 in_clk，以及当前和目标的瓶数 in_bottle_num、in_target_bottle_num 和药片数 in_pill_num、in_target_pill_num，还有警告标志 in_warning_flag 和使能信号 in_warning_enable，输出四组 4 位 LED 信号（out_bottle_LED_1、out_bottle_LED_0、out_pill_LED_1、out_pill_LED_0）和错误显示 out_error_display。

模块通过组合逻辑实现灵活的显示控制：

默认情况下，out_bottle_LED_1 和 out_bottle_LED_0 显示当前瓶数 in_bottle_num 的十位和个位，out_pill_LED_1 和 out_pill_LED_0 显示当前药片数 in_pill_num 的十位和个位。
设置状态下，即当 in_display_setting 为 1 时，根据 in_flash 的值选择显示目标瓶数或药片数（in_target_bottle_num 或 in_target_pill_num 的十位和个位），并通过 in_clk 控制对应 LED 的闪烁效果（高电平时显示数值，低电平时全 1 表示熄灭）。

出现错误时，错误显示 out_error_display 在 in_warning_enable 为 1 时输出 in_warning_flag，否则置为全 1，即熄灭。

模块通过模运算和除法将输入数值转换为十位和个位，便于 LED 显示，同时确保警告信息在使能时正确传递。

3.2 输入输出接口

输入接口：

信号名称	说明
in_display_setting	显示设置模式标志
in_flash[1:0]	闪烁控制信号
in_clk	时钟输入
in_bottle_num[5:0]、in_pill_num[5:0]	当前药瓶和药片数量
in_target_bottle_num[5:0]、 in_target_pill_num[5:0]	目标药瓶和药片数量
in_warning_flag[1:0]	警告标志
in_warning_enable	警告使能信号

输出接口：

信号名称	说明
out_bottle_LED_1[3:0]、out_bottle_LED_0[3:0]	药瓶数量显示（十位和个位）
out_pill_LED_1[3:0]、out_pill_LED_0[3:0]	药片数量显示（十位和个位）
out_error_display[3:0]	错误显示输出

3.3 局部变量

下面的局部变量是根据除法器和取余器计算出来的每一位的值，用于直接输出给 LED。

```
wire [3:0] bottle_1, bottle_0;
wire [3:0] pill_1, pill_0;
wire [3:0] target_bottle_1, target_bottle_0;
wire [3:0] target_pill_1, target_pill_0;
```

4. 设置修改模块(Modify_Setting_Module)

4.1 功能介绍

设置修改模块用于设置目标药瓶数量和每瓶药片数量，支持单步增加(+1)，快速增加(+5)和清零三种调整方式，向显示模块传递闪烁标志位，配合显示模块一起在相应的设置模式下激活闪烁提示。它接收输入信号包括显示状态 in_display_state、设置使能 in_display_setting、清零信号 in_CLR、脉

冲信号 in_PULSE 和增量选择 in_inc_five，输出闪烁控制 out_flash、目标瓶数 out_target_bottle_num 和目标药片数 out_target_pill_num。

模块包含两个主要逻辑块：第一个 always 块是组合逻辑，根据 in_display_state 设置 out_flash，在 d_standard 状态下关闭闪烁，在 d_setting_bottle 或 d_setting_pill 状态下分别置位 out_flash 为 2'b10 或 2'b01，以控制对应显示的闪烁效果；默认情况下关闭闪烁。第二个 always 块在 in_PULSE 上升沿或 in_CLR 下降沿触发，若 in_CLR 为低，则根据 in_display_setting 和 in_display_state 清零对应的 out_target_bottle_num 或 out_target_pill_num；否则，在设置模式下根据 in_inc_five 选择将目标值递增 1 或 5，非设置模式下保持不变。

4.2 输入输出接口

输入接口：

信号名称	说明
in_display_state[1:0]	显示状态
in_display_setting	显示设置标志
in_CLR	清除信号
in_PULSE	脉冲信号，用于增加设置值
in_inc_five	增加 5 个单位的信号

输出接口：

信号名称	说明
out_flash[1:0]	闪烁控制信号
out_target_bottle_num[5:0]	目标药瓶数量设置
out_target_pill_num[5:0]	目标药片数量设置

5. 状态显示模块(State_Display_Module)

5.1 功能介绍

通过 7 段 LED 显示系统当前状态
显示不同的字符或图形来表示不同状态：
零状态：显示"0"
运行状态：显示上横线
暂停状态：显示"P"
换瓶状态：显示三横线
报告状态：显示下横线
设置模式：显示"S"
错误状态：显示"E"

根据系统状态和控制信号驱动 7 段 LED 显示器，展示不同的状态或提示信息。它接收输入信号包括 2 位状态 in_state、暂停信号 in_suspend、完成信号 in_finish、下一瓶信号 in_next_bottle、设置模式信号 in_setting 和警告使能信号 in_warning_enable，输出 7 位寄存器信号 out_7_LED 用于控制 7 段显示器。

模块通过组合逻辑实现显示控制，优先级依次为：当 in_warning_enable 为 1 时，显示字符 “E”

（错误）；当 in_setting 为 1 时，显示字符 “S”（设置中）。否则，根据 in_state 的值选择显示内容：在 s_zero 状态显示字符 “0”；在 s_operation 状态下，根据 in_finish 和 in_next_bottle 的组合显示不同图案（完成时显示下划线，下一瓶时显示三横线或暂停时的 “P”，正常运行时显示上划线）；默认状态显示下划线。

5.2 输入输出接口

输入接口：

信号名称	说明
in_state[1:0]	系统状态
in_suspend	暂停信号
in_finish	完成信号
in_next_bottle	下一个药瓶信号
in_setting	设置模式标志
in_warning_enable	警告使能信号

输出接口：

out_7_LED[6:0]	7 段 LED 显示输出

6. 报警模块(Warning_Module)

6.1 功能介绍

系统通过双位警报标志(warning_flag)实现异常监测与处理机制。该机制在两种关键场景下发挥作用：首先，当系统处于初始状态(s_zero)准备开始工作时，会检查关键参数是否合法。若发现目标瓶数或药片数设置为零，系统会激活 warning_flag[0]，拒绝转入工作状态，强制用户先完成必要参数设置。其次，在系统运行过程中(s_operation 状态)，若检测到设置修改尝试，会立即触发 warning_flag[1]，以防止操作进行时的参数变更可能导致的系统不稳定或结果错误。这些警报标志通过警报控制模块(Warning_Module)转换为可视/可听信号。具体实现上，当任一警报条件满足且倒计时有效时，系统将输入时钟信号(warning_inclk)转化为警报输出(warning_outclk)，同时在显示模块上呈现对应错误代码，提醒操作者采取相应措施。

6.2 输入输出接口

由于该模块位于顶层模块设计中，没有实例化，故不再说明。

7. 总体运行流程

7.1 系统初始化

设置阶段：

1. 按下 display_setting 键进入设置模式：首先进入药瓶数设置，相应数码管闪烁
2. 通过 PULSE 按键增加药瓶数(+1)，或切换按键快速增加(+5)

3. 再次按下设置键切换到药片数设置，同样通过 PULSE 和切换按键设置每瓶药片数量
4. 第三次按下显示设置键完成设置，返回标准模式，若设置的药瓶数或药片数为零，系统会显示错误代码并激活警告

7.2 启动运行阶段

启动检查：

按下 QD 键启动系统，系统检查设置值是否有效，若无效则维持在错误状态并发出警告，若设置有效，则转入运行状态开始运行

药片灌装过程：

系统进入运行状态，7 段 LED 显示为上横线(工作中)，随着时钟(clk)每个上升沿，药片计数值增加 1

当前灌装状态通过数码管实时显示(当前药瓶数和当前药片数)

暂停控制：

运行过程中可通过暂停按键暂停操作，暂停时，7 段 LED 显示字符"P"，计数器停止增加，系统保持当前状态，再次按下暂停键恢复运行

7.3 药瓶切换

单瓶灌装完成：

当药片达到目标值时，触发装瓶信号，7 段 LED 显示切换为三横线图形，提示需要更换药瓶

药瓶计数加 1，药片计数重置为 0

药瓶更换操作：

操作者更换药瓶后，系统自动继续灌装新的药瓶

7.4 结束阶段

全部灌装完成时：

当药瓶等于设定目标瓶数时，系统停止运行，7 段 LED 显示为下横线，表示完成运行

结果显示：

数码管显示最终灌装的药瓶数和药片数，系统等待用户确认或自动转入最初状态

7.5 异常处理流程

无效设置处理：

若药瓶或者药片数量为零，系统发出警告，信号警告灯闪烁，提示用户调整设置

运行中设置尝试：

若在正常运行状态尝试进入设置模式，系统发出警告信号，拒绝进入设置模式，维持当前运行状态

警告倒计时：

警告触发后启动倒计时，警告持续一段时间后自动关闭，错误显示区域显示对应的错误代码

7.6 复位功能

随时可通过复位信号将系统复位到初始状态，复位后所有计数器清零，系统等待新的设置和启动。

四、详细代码

1. 顶层模块 Pill_Bottling_System.v

```
module Pill_Bottling_System(
input   clk,
input   warning_inclk,
//main state switch input
input   QD,reset_state,         //K15
input   display_setting,        //K0
//s_opration
input   suspend,               //K1
//setting
input   PULSE,
input   CLR,
input   inc_five,              //K2
//display output
output  [6:0]state_LED,
output  [3:0]bottle_LED_1,
output  [3:0]bottle_LED_0,
output  [3:0]pill_LED_1,
output  [3:0]pill_LED_0,
output  [3:0]error_display,
//warning about
output  warning_outclk
);

//state
reg     [1:0]current_state,next_state;
parameter s_zero=2'b00,s_operation=2'b01,s_report=2'b11;

//display_mode
reg     [1:0]current_display_state,next_display_state;
parameter
d_standard=2'b00,d_setting_bottle=2'b01,d_setting_pill=2'b11;
wire    [1:0]flash;

//counter_use
wire    [5:0]bottle_num;
wire    [5:0]pill_num;
wire    [5:0]target_bottle_num;
wire    [5:0]target_pill_num;
```

```
wire    next_bottle;
wire    finish;

//alarm_use
reg     [1:0]warning_flag;
reg     [1:0]warning_countdown;
wire    warning_enable;

//=============================================
//State_Change
//=============================================
always@(posedge QD,negedge reset_state)
begin
    if(!reset_state)
        current_state<=s_zero;
    else if(!display_setting)
        current_state<=next_state;
    else
        current_state<=current_state;
end

always@(*)
begin
    case(current_state)
        s_zero:begin
            if((display_setting==0)&&(target_bottle_num==0||target_pill_
num==0))begin
                next_state=s_zero;
                warning_flag[0]=1'b1;end
            else begin
                warning_flag[0]=1'b0;
                next_state=s_operation;end
        end
        s_operation:begin
            if(finish)
                next_state=s_zero;
            else
                next_state=s_report; end
        s_report:begin
            next_state=s_zero; end
        default:begin
            next_state=s_zero; end
    endcase
end
```

```
always@(posedge QD,negedge display_setting)
begin
    if(!display_setting)
        current_display_state<=d_standard;
    else
        current_display_state<=next_display_state;
end

always@(*)
begin
    case(current_display_state)
        d_standard:begin
            if(current_state==s_operation)begin
                warning_flag[1]=1'b1;
                next_display_state=d_standard;end
            else begin
                next_display_state=d_setting_bottle;
                warning_flag[1]=1'b0;end
        end
        d_setting_bottle:begin
            next_display_state=d_setting_pill;
        end
        d_setting_pill:begin
            next_display_state=d_standard;
        end
        default:begin
            next_display_state=d_standard;
        end
    endcase
end

//=============================================
//Warning_Module
//=============================================
assign warning_outclk=warning_enable&&warning_inclk;
assign
warning_enable=(warning_flag!=2'b0)&&(warning_countdown!=2'b00);
always@(posedge clk,posedge QD)
begin
    if(QD)
        if((!(display_setting)&&(target_bottle_num==0||target_pill_num=
=0))||(display_setting))
            warning_countdown<=2'b10;
```

```
        else
            warning_countdown<=2'b00;
    else begin
        if(warning_countdown!=2'b00)
            warning_countdown<=warning_countdown-2'b1;
        else
            warning_countdown<=2'b0;
    end
end

//=============================================
//Counter_Module
//=============================================
Counter_Module counter_module(
.in_clk(clk),
.in_state(current_state),
.in_suspend(suspend),
.in_target_bottle_num(target_bottle_num),
.in_target_pill_num(target_pill_num),
.out_bottle_num(bottle_num),
.out_pill_num(pill_num),
.out_next_bottle(next_bottle),
.out_finish(finish)
);

//=============================================
//State_Display_Module
//=============================================
State_Display_Module state_display_module(
.in_state(current_state),
.in_suspend(suspend),
.in_finish(finish),
.in_next_bottle(next_bottle),
.in_setting(display_setting),
.in_warning_enable(warning_enable),
.out_7_LED(state_LED)
);

//=============================================
//Display_Module
//=============================================
Display_Module display_module(
.in_display_setting(display_setting),
.in_flash(flash),
```

```
.in_clk(clk),
.in_bottle_num(bottle_num),
.in_pill_num(pill_num),
.in_target_bottle_num(target_bottle_num),
.in_target_pill_num(target_pill_num),
.in_warning_flag(warning_flag),
.in_warning_enable(warning_enable),
.out_bottle_LED_1(bottle_LED_1),
.out_bottle_LED_0(bottle_LED_0),
.out_pill_LED_1(pill_LED_1),
.out_pill_LED_0(pill_LED_0),
.out_error_display(error_display)
);

//=============================================
//Modify_Setting_Module
//=============================================
Modify_Setting_Module modify_setting_module(
.in_display_state(current_display_state),
.in_display_setting(display_setting),
.in_CLR(CLR),
.in_PULSE(PULSE),
.in_inc_five(inc_five),
.out_flash(flash),
.out_target_bottle_num(target_bottle_num),
.out_target_pill_num(target_pill_num)
);

endmodule
```

2. 计数模块 Counter_Module.v

```
module Counter_Module(
input in_clk,
input [1:0]in_state,
input in_suspend,
input [5:0] in_target_bottle_num,
input [5:0] in_target_pill_num,
output reg [5:0] out_bottle_num,
output reg [5:0] out_pill_num,
output reg out_finish,
output reg out_next_bottle
```

```
);

parameter s_zero=2'b00,s_operation=2'b01,s_report=2'b11;

always@(posedge in_clk)
begin
    case(in_state)
        s_zero: begin
            out_bottle_num <= 6'b0;
            out_pill_num <= 6'b0;
            out_next_bottle<=1'b0;
            out_finish<=1'b0; end
        s_operation: begin
            if(out_bottle_num==in_target_bottle_num)begin
                out_finish<=1'b1;
                out_next_bottle<=1'b0;end
            else begin
                if (out_pill_num == in_target_pill_num-1'b1) begin
                    out_bottle_num <=(in_suspend)?out_bottle_num:
out_bottle_num + 1;
                    out_pill_num <=(in_suspend)?out_pill_num: 0;
                    out_next_bottle <= (in_suspend)?0:1;
                    out_finish<=1'b0; end
                else begin
                    out_pill_num<=(in_suspend)?out_pill_num:out_pill_num+
1'b1;
                    out_next_bottle <= 0;
                    out_finish<=1'b0; end
            end
        end
        default:begin
            out_next_bottle<=0;
            out_finish<=1'b0;end
    endcase
end
endmodule
```

3. 显示模块 Display_Module.v

```
module Display_Module(
input in_display_setting,
input [1:0]in_flash,
```

```
input in_clk,

input [5:0]in_bottle_num,
input [5:0]in_pill_num,
input [5:0]in_target_bottle_num,
input [5:0]in_target_pill_num,

input [1:0]in_warning_flag,
input in_warning_enable,

output reg [3:0]out_bottle_LED_1,
output reg [3:0]out_bottle_LED_0,
output reg [3:0]out_pill_LED_1,
output reg [3:0]out_pill_LED_0,
output [3:0]out_error_display
);

wire [3:0] bottle_1, bottle_0;
wire [3:0] pill_1, pill_0;
wire [3:0] target_bottle_1, target_bottle_0;
wire [3:0] target_pill_1, target_pill_0;

always@(*)
begin
    case({in_display_setting,in_flash})
        3'b100:begin
            out_bottle_LED_1=target_bottle_1;
            out_bottle_LED_0=target_bottle_0;
            out_pill_LED_1=target_pill_1;
            out_pill_LED_0=target_pill_0;end
        3'b110:begin
            out_bottle_LED_1=(in_clk)?target_bottle_1:4'b1111;
            out_bottle_LED_0=(in_clk)?target_bottle_0:4'b1111;
            out_pill_LED_1=target_pill_1;
            out_pill_LED_0=target_pill_0;end
        3'b101:begin
            out_bottle_LED_1=target_bottle_1;
            out_bottle_LED_0=target_bottle_0;
            out_pill_LED_1=(in_clk)?target_pill_1:4'b1111;
            out_pill_LED_0=(in_clk)?target_pill_0:4'b1111;end
        default begin
            out_bottle_LED_1=bottle_1;
            out_bottle_LED_0=bottle_0;
            out_pill_LED_1=pill_1;
```

```
            out_pill_LED_0=pill_0;end
    endcase
end

assign bottle_1 = in_bottle_num/10;
assign bottle_0 = in_bottle_num-bottle_1%10;
assign pill_1 = in_pill_num / 10;
assign pill_0 = in_pill_num %10;

assign target_bottle_1 = in_target_bottle_num / 10;
assign target_bottle_0 = in_target_bottle_num % 10;
assign target_pill_1 = in_target_pill_num / 10;
assign target_pill_0 = in_target_pill_num % 10;

assign
out_error_display=(in_warning_enable)?{2'b00,in_warning_flag}:4'b1111;

endmodule
```

4. 设置模块 Modify_Setting_Module.v

```
module Modify_Setting_Module(
input [1:0]in_display_state,
input in_display_setting,
input in_CLR,
input in_PULSE,
input in_inc_five,
output reg [1:0]out_flash,
output reg [5:0]out_target_bottle_num,
output reg [5:0]out_target_pill_num
);

parameter
d_standard=2'b00,d_setting_bottle=2'b01,d_setting_pill=2'b11;
parameter s_zero=2'b00,s_operation=2'b01,s_report=2'b11;

always@(*)
begin
    case(in_display_state)
        d_standard:begin
            out_flash=2'b00;end
        d_setting_bottle:begin
```

```
            out_flash=2'b10;end
        d_setting_pill:begin
            out_flash=2'b01;end
        default:begin
            out_flash=2'b00;end
    endcase
end

always @(posedge in_PULSE,negedge in_CLR)
begin
    if (!in_CLR) begin
        out_target_bottle_num <= (in_display_setting &&
(in_display_state == d_setting_bottle) ) ? 0 : out_target_bottle_num;
        out_target_pill_num   <= (in_display_setting &&
(in_display_state == d_setting_pill) ) ? 0 : out_target_bottle_num;
    end else begin
        out_target_bottle_num <=
            (in_display_setting && (in_display_state ==
d_setting_bottle)) ?
((in_inc_five)?out_target_bottle_num+5:out_target_bottle_num + 1):
out_target_bottle_num;
        out_target_pill_num   <=
            (in_display_setting && (in_display_state ==
d_setting_pill))   ? ((in_inc_five)?out_target_pill_num +
5:out_target_pill_num+1)   : out_target_pill_num;
    end
end

endmodule
```

5. 状态显示模块 State_Display_Module.v

```
module State_Display_Module(
input [1:0]in_state,
input in_suspend,
input in_finish,
input in_next_bottle,
input in_setting,
input in_warning_enable,

output reg [6:0]out_7_LED
);
```

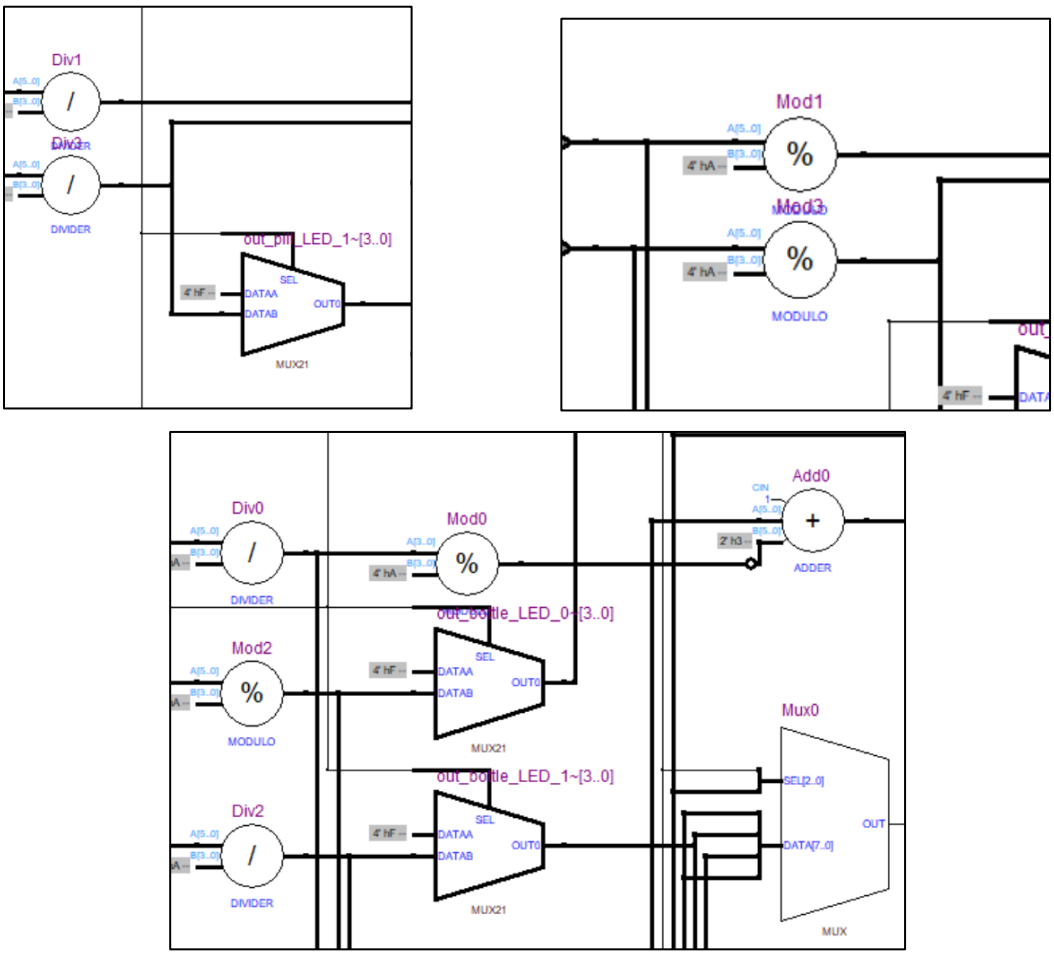
```
parameter s_zero=2'b00,s_operation=2'b01,s_report=2'b10;
parameter
d_standard=2'b00,d_setting_bottle=2'b01,d_setting_pill=2'b11;

parameter c_E=7'b1001111;//79 character_Error
parameter c_S=7'b1011011;//91 character_Setting
parameter c_P=7'b1100111;//103 character_Pause
parameter c_0=7'b1111110;//126 character_Zero
parameter g_u=7'b1000000;//64 graph_up_line        for working
parameter g_b_report=7'b0001000;//8 graph_under_line    for report
parameter g_t=7'b1001001;//73 graph_three_line         for next bottle

always@(*)
begin
    if(in_warning_enable)
        out_7_LED=c_E;
    else if(in_setting)
        out_7_LED=c_S;
    else begin
        case(in_state)
            s_zero:
                out_7_LED=c_0;
            s_operation:begin
                case({in_finish,in_next_bottle})
                    2'b10:
                        out_7_LED=g_b_report;
                    2'b01:
                        out_7_LED=(in_suspend)?c_P:g_t;
                    default:
                        out_7_LED=(in_suspend)?c_P:g_u;
                endcase end
            default:begin
                out_7_LED=g_b_report;end
        endcase
    end
end

endmodule
```

五、调试过程问题及讨论

在这次地调试过程中，主要的代码上的问题集中在后期，前期的问题与讨论大多与代码无关，而是集中于一些设备问题，例如不知道调到硬连线模式，例如线接反了等等。这里主要总结一下后期遇到的问题和讨论，后期主要的问题其实只有一个，就是如何节约资源。

首先我们的药片系统在代码上最大的问题是资源占用太多，导致无法加入新功能，有想法却不能实现。代码中占用最多资源最多的是显示模块，因为我们使用寄存器两位两位地存储数据，但是显示到 LED 上必须转换为一位一位的 8421 码。这就引入了四个除法器和四个取余器，占用资源量就会很大。

如下图网表文件所示的一共四个除法器，取余器：



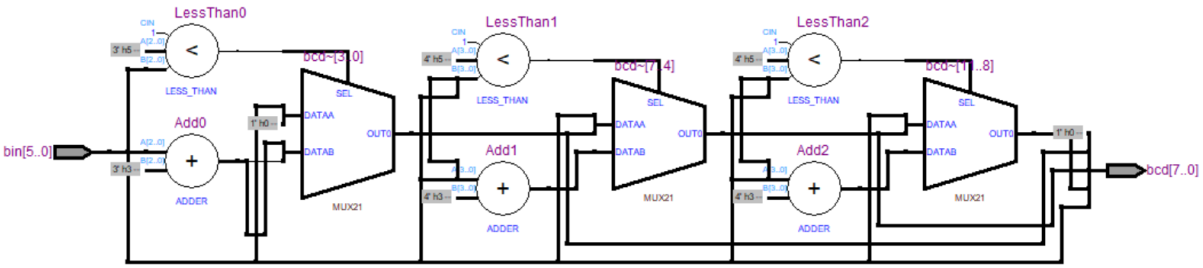
1. 按位存储的尝试

首先经过讨论，我们发现其他小组包括本小组时钟都是使用按位操作，猜测也许使用按位存储药瓶数药片数要好得多，这样就避免了除法器和取余器的使用。然而时钟都是按位设置的，我们的药片也要按位设置吗？经过讨论，我们的看法是既然我们已经实现了相当多的功能。如果为了引入更多的功能而改为按位操作（按位操作是不符合常识的，一般的产品不会是按位操作的），损失了用户体验，不如不这样做。所以我们准备自己制作按位的进位模块，最终成功在中期检查后又一次完成了一个版本，可是这样制作出来的版本比使用除法器占用资源差不多，甚至更多。于是只好放弃。

2. 使用二进制转 BCD 代替除法器功能。

第二种方法是自己设计除法器，我们查阅资料，根据加三移位法，设计出了二进制转 BCD 模块，

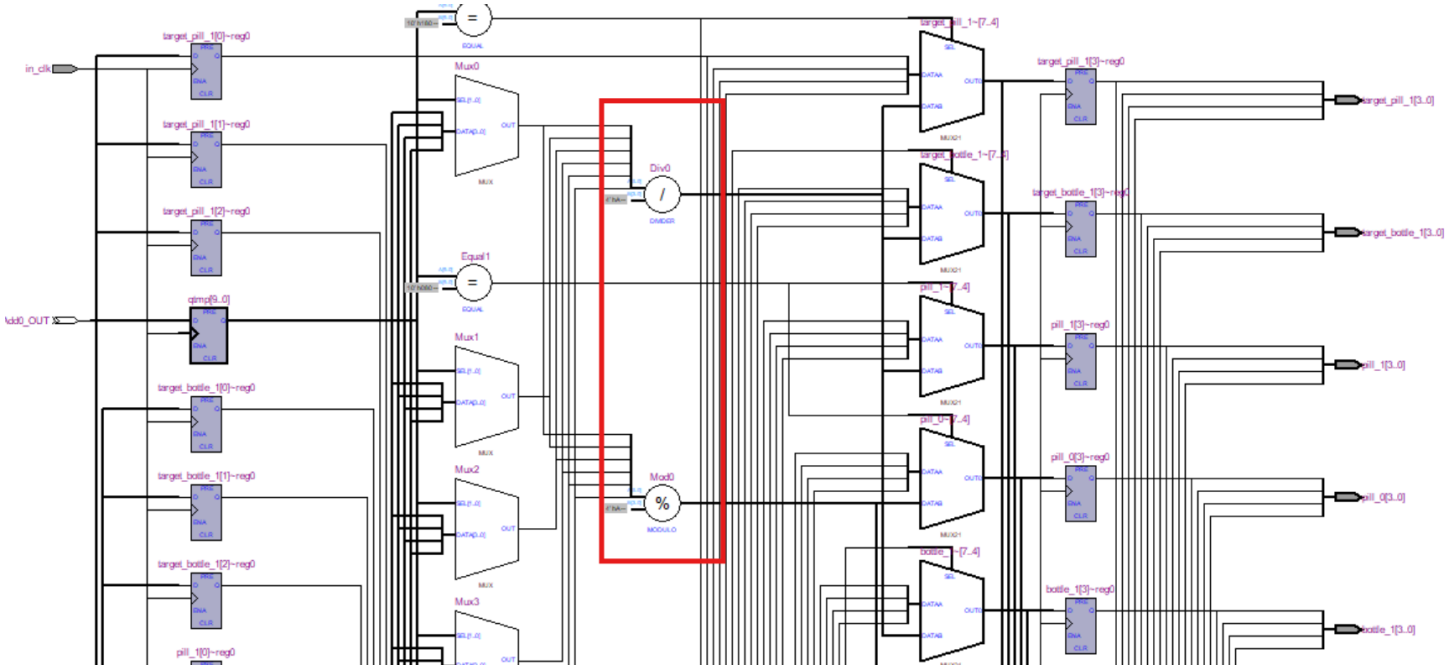
使用四个模块代替了原来四个除法器和取余器。下面是二进制转 BCD 模块的网表文件。



然而这样设计占用资源仍旧是和除法器差不多。这种方法不能采用。

3. 时分复用的尝试

既然几乎不能放弃除法器和取余器，能不能用时分复用的方式，令四个除法器和取余器时分复用，只用一个除法器和取余器。我们引入 1000hz 的时钟（之所以不适用 100hz 是因为 100hz 和 1hz 占用同一个接口，而 1hz 正在使用），令分别在计数为 0，250，500，750 的时候就进行一次运算，共四次运算，但是只使用了一个除法器和取余器。我们的成果如下：



但是这种方法引入了多余的寄存器，导致资源占用比之前两个还要多一点。于是不能采用。

使用三个方法后，我们还想尝试新的方法，奈何时间已经临近期末，没有时间和精力再做出新的改动了。于是最终未能实现药片系统的资源化简。

六、设计调试小结

本药片装瓶系统基于 Verilog 硬件描述语言，运用自顶向下的模块化设计方法，整合主控、计数器、显示、设置修改、状态显示、报警等 6 大核心模块，成功实现清零、设置、工作状态间的正确切换，药瓶与药片数量实时显示，目标值设定与调整（支持单步与快速增加），暂停/继续功能，以及无效操作告警等功能，同时通过数码管闪烁进行优化。在完整的设计流程中，从需求分析明确功能目标，到模块设计细化各部分逻辑，再经仿真调试验证模块功能与协同性，最后进行硬件验证

实现系统落地，这一过程极大深化了我们对数字系统设计的理解。

实验过程中，团队成员不断优化逻辑、优化模块设计、优化资源占用，特别针对 CPLD 芯片的资源限制进行了精简设计和硬件语言优化，以达到工程实验的实际效果落地，克服了环境差异导致的编译问题、逻辑错误引发的功能异常、文件上传的问题、资源超限等，通过协作与技术排查逐一解决，不仅积累了宝贵的工程经验，也完成了对 Verilog 语言编程和 FPGA 开发技能的实践检验，实现了从理论知识到实际应用的系统性训练，为后续复杂数字系统的设计与开发奠定了坚实基础。

附录一、各成员心得总结

丁乐航：在电子钟和药片装瓶系统的开发过程中，从最初的方案设计到最终的调试实现，每一个环节都让我对数字系统的设计流程有了更直观的认识。特别是在使用 EPM7128 芯片进行开发时，我不仅巩固了 Verilog 编程技能，更学会了如何根据实际硬件资源进行设计优化。实验过程中遇到的时序问题和信号干扰等挑战，促使我不断思考解决方案，大大提升了我的调试能力和工程思维。同时，团队协作完成项目的经历也让我认识到沟通与分工的重要性。这次实践不仅加深了我对数字逻辑设计的理解，更为今后的工程实践积累了宝贵经验。

马一民：通过本次实验，加强了我用 verilog 设计数字系统的能力。在编写程序的过程中，我对于 verilog 的理解和掌握得到了提升。在实验过程中需要不断地进行调试和测试，加深了我对于硬件电路的理解。为了使得占用资源最小，我们不得不想出新的解决办法，在这一过程中，我们分析网表文件的能力加强，更进一步地理解了硬件和软件的区别。

肖璨：本次数字逻辑课程设计通过电子钟和药片装瓶两个系统的实现，加深了我对状态机设计、模块化开发及硬件调试的理解。项目过程中，团队合作与合理分工使设计效率大大提升，大家各司其职，同时保证了进度和质量。面对多信号同步和资源限制等挑战时，我不仅学会了优化设计，还增长了对调试技巧的认识。这些经历极大地锻炼了我的硬件设计与调试能力，让我对硬件设计的实际应用有了更深认识。

张鑫溢：在数字逻辑课程设计中，通过电子钟与药片装瓶两大系统开发，运用 Verilog 语言结合实际实验平台，完成模块化设计编写与硬件功能实现；围绕工程实验实际需求，持续开展逻辑优化、模块设计迭代与资源占用精细化管理。在这一过程中，解决了编译异常、逻辑错误、文件上传障碍以及资源超限等问题。在此期间，我们实践与理论相结合，将理论运用于实践，为后续学习筑牢了技术根基。

附录二：工作日志

第 2 周　3 月 5 日

听课

第 3 周　3 月 12 日

听课，初步分工

第 4 周　3 月 19 日

四人讨论：讨论器材选择。设立初步了解的任务。
这次讨论中，我们商讨了课程设计中将要使用的设备，最终决定使用 TEC-8 作为实验系统，使用 quartus 9.1　作为开发平台。同时，在这一次讨论中，我们还对实验任务进行了初步分化，决定先搭建每一个任务的基本框架，再依次共同完成每一个实验。

第 5 周　3 月 26 日

时钟：
初步实现计数功能，并在尝试加入时钟控制功能时出现逻辑错误。其后尝试使用例化方式对每一个模块进行分化，并修改时、分、秒计数使用 60 进制计数器和 24 进制计数器，且尝试实现显示模块

药片：
完成基本框架设计。进行测试，发现显示为乱码。尝试修改数码管的引脚后解决乱码的问题，但是仍旧无法正常运行。尝试修改状态机设置，仍旧无法运行。

第 6 周　4 月 2 日

时钟：
尝试在显示模块中加入闪烁功能，尝试使用时钟分频得到 1hz 时钟，初步尝试加入闹钟模块

药片：
继续上周进行测试，发现逻辑混乱，能够显示，但是按下任何按钮都无响应，猜测是状态机逻辑有误，错误地认为状态机不能使用 QD 作为上升沿信号，于是重写状态机，仍旧无响应，长时间调试无果。快要下课时请教老师了解到需要把状态切换为“硬连线“，否则会出现一些奇怪的错误。

第 7 周　4 月 9 日

时钟：
加入按键处理模块和模式控制模块，初步实现自动机式的模式切换以及上升沿转换电平信号的功能

药片：
继续上周进行调试，切换为硬连线后确实排除了错误，但是发现只有在有的机器上才能运行，而且各种机器报错都不同。其中一台机器甚至出现了 1hz 和 1000hz 引脚相反的情况。后来才发现原来是因为每次接花线时都不关注正反，有的机器接对了，有的机器接错了，所以引脚相反甚至完全无响应。正确接线后一切正常。截至至此，初步实现了设置药瓶药片数，正常运行和显示功能。

第 8 周　4 月 16 日

时钟：
资源超限，修改每一个时钟位为单独的计数器，并优化调整各按键

药片：
这周增加了许多实现细节，如可以支持暂停，可以使用快速设置，即按一下 QD 增加 5，增加 CLR 可以清零设置的功能。

第 9 周　4 月 23 日

时钟：
修改闪烁逻辑，尝试使其 1 秒闪烁一次，调整整体显示逻辑

药片：
增加告警功能，分别为由清零状态进入计数状态时，如果药瓶数或者药片数为 0，则告警，如果在计数过程中，尝试修改设置也会告警。

第 10 周　4 月 30 日

时钟：
多次修改调整，中期验收
重写时钟控制逻辑，重写闹钟控制逻辑，新增 2hz 闪烁逻辑，重写显示逻辑，初步实现除响铃外所有功能

药片：
中期验收，目前完成的功能有：基本状态切换，计数，设置，告警，工作状态可以切换初始设置，闪烁。

第 11 周　5 月 7 日

时钟：
调试重新上传，实现响铃声音的初步实现

药片：
尝试更换底层的药瓶药片寄存器为按位的寄存器，来节省资源。但是我们不希望设置药瓶药片数时按位操作，而是仍旧能够一次操作两位，所以需要额外的逻辑判断进位。最后发现并不能节约资源，故放弃该方案。

第 12 周　5 月 14 日

时钟：
修改响铃铃声，实现初步音调

药片：
尝试使用了二进制转 BCD 方法代替除法器，尝试使用时分复用减少除法器的数目，均未能达到理想效果。后续课业压力增大，故在此结束了药片装瓶系统的设计。

第 13 周　5 月 21 日

第 14 周　5 月 28 日

第 15 周　6 月 4 日

第 16 周　6 月 11 日

附录三，贡献度表

2025 年课程设计 已实现的功能（在实现的功能前打勾） 组号：B9 验收时间 6 月 6 日				
电子钟系统设计	**基本功能**	√ 24 小时制时钟功能 √ 整点报时功能 √ 时间设置功能 √ 切换工作模式	**附加功能**	√ 音乐整点报时功能 √ 闹钟功能 √ 设置状态闪烁显示 ◊ 其他：_____
药片装瓶系统设计	**基本功能**	√ 药片装瓶功能 √ 显示药瓶及药片数量 √ 工作状态及告警指示 √ 切换工作模式	**附加功能**	√ 设置状态时闪烁显示 √ 工作状态时显示初始设置 √ 设定每瓶药片数和总药片数 ◊ 其他：_____

学 号	姓名	承担的工作	贡献度（总数 100%）
2023211209	肖璨	①负责时钟部分的顶层模块设计规划，同时负责时钟分频、按键处理，并完成时钟计数模块设计 ②完成药片部分的计数模块设计。③负责电子钟的顶层设计；④负责电子钟的调试优化。	25
2023211	张鑫溢	①负责电子钟的顶层设计；②负责电子钟的调试优化；③负责显示模块的实现，包含电子时钟的显示以及药片对应的显示实现；④负责电子钟的闹钟、整点报时音乐以及药片的对应声音功能。	25
2023211196	马一民	①负责药片的整体架构，完成了药片中状态转换模块（顶层模块）和报警模块。②完成了时钟的状态转换模块。③负责药片的顶层设计；④负责药片的调试优化。	25
2023211203	丁乐航	①负责药片药瓶修改的设计，完成了设置修改模块。②完成时钟的闹钟控制模块。③负责药片的顶层设计；④负责药片的调试优化。	25