同济大学计算机系

数字逻辑课程综合实验报告

学　　号 2452487

姓　　名 胡中芃

专　　业 信息安全

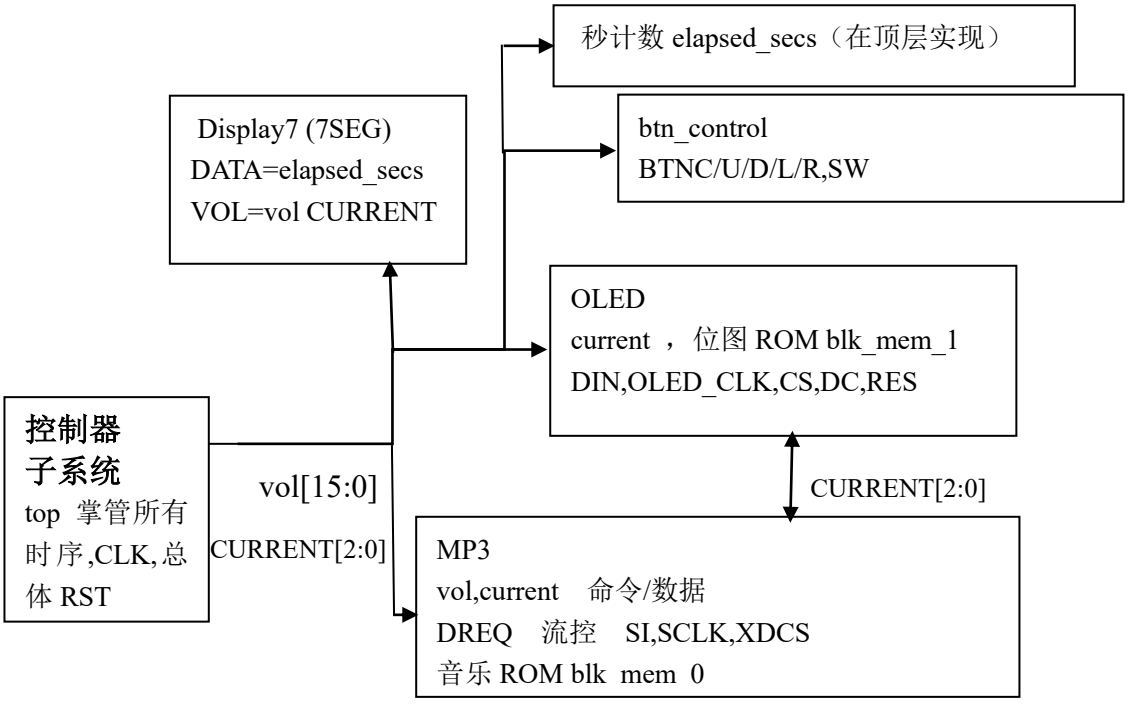
授课老师 张冬冬

一、实验内容

本实验利用 Nexys4 DDR FPGA 开发板、MP3 和 OLED 外设实现了一个集显示与音频播放于一体的数字系统，运用 Verliog 语言、IP 核实现，总体采用了自顶向下的设计方法，通过 top.v 模块调用各个功能，通过 OLED 屏幕显示预置图片（此处可为音乐的专辑封面）；通过 VS1003 音频解码器播放存储在片内 ROM 的多首歌曲；7 段数码管显示播放时间（分秒）、音量级别和当前歌曲编号；利用板载五个按键和 16 位滑动开关完成音量调节/曲目切换/默认复位等操作。

系统的主要思想是以输入子系统采集人机交互信号，由控制器子系统根据输入状态协调显示子系统（七段/OLED）与 VS1003 音频芯片子系统，并通过时钟与计时子系统提供稳定的时序与时间基准。各子系统通过顶层模块互联，形成结构清晰、职责分明的完整数字系统。

二、音乐播放器数字系统总框图



上述数字系统总框图中为了让 OLED 对应能展示当前 MP3 放的歌曲的图片需要二者同步，采用**双向箭头**连接方式，用 CURRENT[2:0]记录当前歌曲下标。

2.1 输入子系统

主要包括板载五个按键 BTNC/BTNU/BTND/BTNL/BTNR 和 16 位开关 SW，接口为顶层采样 CLK 与低电平有效 RST；去抖同步后输出音量编码 vol[15:0]与当前歌曲索引 CURRENT[2:0]（支持放最多 8 首歌）。内部采用两级触发器同步，边沿检测产生按键事件；对 SW 采用电平同步。具体而言，BTNU/BTND 控制音量加/减，每次步进 0x1010（左右声道同时变化），上限 0xF0F0，下限 0x0000；BTNL/BTNR：上一曲/下一曲，循环选择 0-7；BTNC：恢复默认（复位到音量 0x4040，曲目 0）；SW[7:0]为直接选择曲目，优先级高于按键循环。使用完全组合-时序混合逻辑，不依赖外部存储，是系统的输入与命令源。

2.2 输出子系统 1（7 段数码管显示）

由八位七段显示与一个小数点组成，接口包括输入 CLK，接收时间 DATA[15:0]（单位秒）、音量 VOL[15:0]、曲目编号 CURRENT[2:0]；输出段码 SEG[6:0]、位选 SHIFT[7:0]、小数点 DOT。内部含 Divider 分频产生扫描时钟；以动态扫描的方式逐位点亮，DOT 在中间两位形成冒号效果。

具体显示格式为左两位显示曲目编号（十位/个位），中两位显示音量级别（十位/个位），右四位显示时间 `mm:ss`。时间由顶层模块将秒数拆分为分/秒的两个 BCD 字。特性为纯显示，不产生控制信号；段码译码使用 0-9 的逻辑。

2.3 输出子系统 2（OLED 显示）

组成包括 OLED 屏和控制信号 DIN/OLED_CLK/CS/DC/RES。接口为输入 CLK/RST/current；输出上述五个控制信号。这个子系统内置初始化命令序列表，并使用 IP 核 blk_mem_gen_1 提供按 current 选择的位图数据。

使用状态机实现，分为命令阶段和数据阶段。命令阶段逐块发送初始化指令；数据阶段逐行（64 行）发送 192 字节像素数据；在 shift_byte_out 态由 2MHz 时钟对 DIN 移位输出，CS/DC 按态置位。

2.4 音频播放 VS1003 子系统（MP3）

由 VS1003 芯片接口 DREQ/XDCS/XCS/RSET/SI/SCLK 与片上音乐 ROM blk_mem_gen_0 组成。接口为输入 CLK/RST/vol/current/DREQ；输出数据与命令片选、时钟、复位及歌曲 LED。数据通路实现方法为从 blk_mem_gen_0 以地址{current, song_word_addr}读出 16bit 数据字，送入 VS1003 SDI；当 DREQ=1 时

允许继续送数据或命令。上电后发送两帧启动命令；音量改变时构造 32bit SCI 命令并在 DREQ=1 时写入；移位由 1MHz 分频驱动。

2.5 时钟与计时子系统

由分频器与顶层计时组成，严格来说也可以算是两个子系统，但都是其他模块的基础，尤其是起到了**时序控制**的作用。

2.6 控制器子系统

顶层 top 将输入与显示/子系统进行互联，具体而言，btn_control 输出的 vol 与 CURRENT 同时驱动 MP3 与 OLED；七段显示读取顶层秒计数；RST 为全局低电平有效。顶层本身完成模块实例化，不具备复杂控制逻辑。

三、系统控制器设计

3.1 状态和条件

系统主要的控制器由 MP3.v 实现，下面详细分析该控制器，首先先给出状态和条件，这些是后续分析的必要前提：

状态：设 S0: boot_delay 上电/曲目切换延时，保持 MP3_RST=0,RSET=0；

S1: boot_wait_cmd_ready— 启动命令发送阶段的等待态，测 DREQ 与计数；

S2: shift_cmd_bits 发送 32bit 启动命令帧（SCI），XCS=0，SCLK 翻转移位；

S3: stream_data 正常播放态，若音量变化则转音量命令通道，否则在 DREQ=1 时发数据；

S4: shift_data_bits 发送 16bit 音频数据字（SDI），XDCS=0；

S5: wait_vol_cmd_ready 等待 DREQ，准备发送 32bit 音量命令；

S6: shift_vol_bits 发送音量命令（SCI），完成后回到 S3。

条件：p = (~RST) | song_change（复位或曲目改变）；

d = DREQ；

delay_done：延时计数到 DELAY_TIME；

c_done：启动命令帧已全部发送；

bit_cmd_done/bit_data_done/bit_vol_done：移位计数到 32/16/32；

vol_changed：当前 vol 与保存的音量字段不一致。

3.2 ASM 流程图

判断逻辑如下：

S0：若 p 或未到 delay_done，留在 S0；若 delay_done，转 S1；

S1：若 c_done，转 S3；否则若 d=1，转 S2 发送下一帧启动命令；

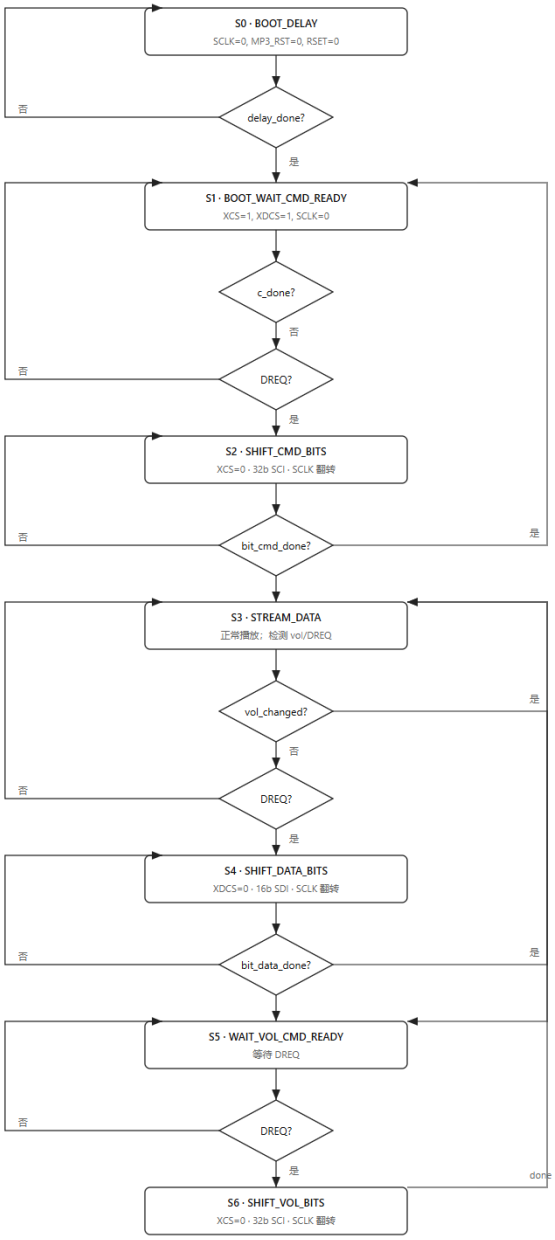
S2：移位 32bit，完成后回 S1；

S3：若 vol_changed 转 S5；否则若 d=1 转 S4；

S4：移位 16bit，完成后回 S3；

S5：等待 d=1 转 S6；

S6：移位 32bit，完成后回 S3。



3.3 状态转移真值表

现态 PS	次态 NS	转换条件
S0	S0	~delay_done+p
S0	S1	delay_done ×~p
S1	S3	c_done
S1	S2	~c_done×d
S2	S1	bit_cmd_done
S3	S5	vol_changed
S3	S4	~vol_changed ×d
S4	S3	bit_data_done
S5	S6	d
S6	S3	bit_vol_done

3.4 系统控制器的次态激励函数表达式和控制命令逻辑表达式

次态激励函数表达式：

$$q_0^{n+1} = q_0 \times (\overline{\text{delay\_done}} + p)$$

$$q_1^{n+1} = q_0 \times \text{delay\_done} \times \overline{p} + q_2 \times \text{bit\_cmd\_done}$$

$$q_2^{n+1} = q_1 \times \overline{\text{c\_done}} \times d$$

$$q_3^{n+1} = q_1 \times \text{c\_done} + q_4 \times \text{bit\_data\_done} + q_6 \times \text{bit\_vol\_done}$$

$$q_4^{n+1} = q_3 \times \overline{\text{vol\_changed}} \times d$$

$$q_5^{n+1} = q_3 \times \text{vol\_changed}$$

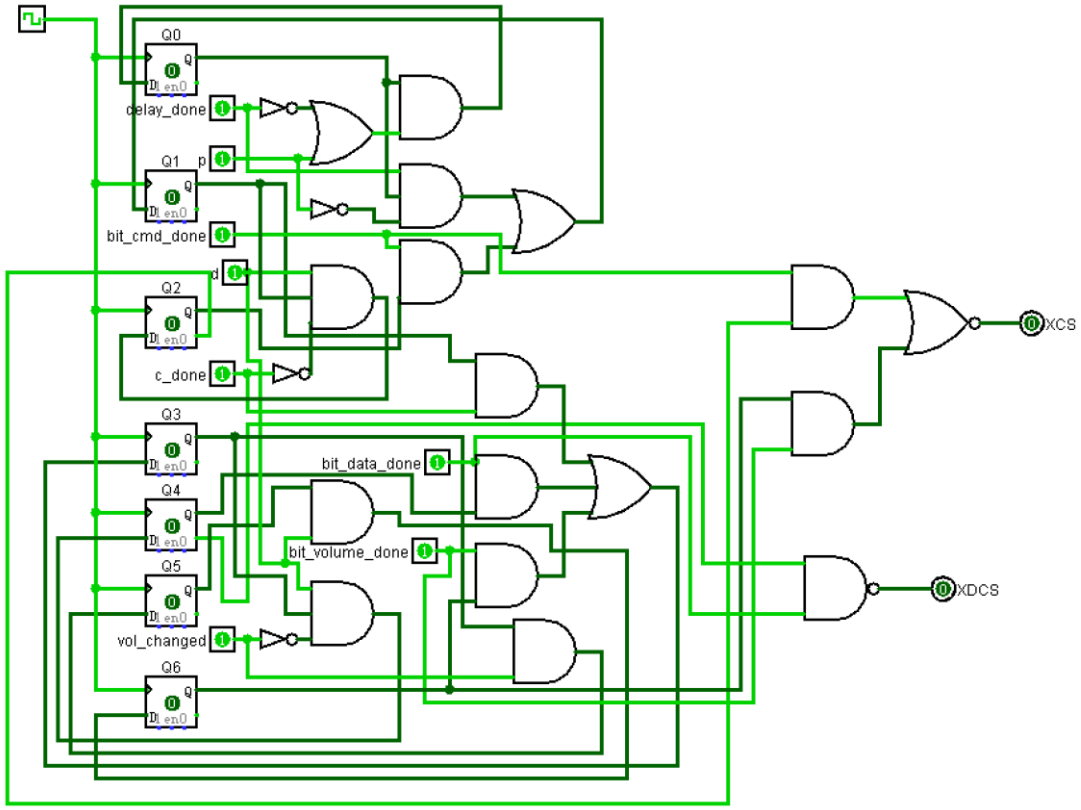
$$q_6^{n+1} = q_5 \times d$$

控制命令逻辑表达式：

$$XCS = \overline{(\overline{q_2} \times \overline{\text{bit\_cmd\_done}})} + \overline{(\overline{q_6} \times \overline{\text{bit\_vol\_done}})}$$

$$XDCS = \overline{\overline{q_4} \times \overline{\text{bit\_data\_done}}}$$

3.5 Logicism 原理图



四、子系统模块建模

4.1 输入与按键处理子系统（btn_control）

```
module btn_control(
    input CLK,
    input RST,              // 低电平有效复位
    input BTNC,
    input BTNU,
    input BTND,
    input BTNL,
    input BTNR,
    input [15:0] SW,        // 滑动开关用于直接选择歌曲
    output reg [15:0] vol,
    output reg [2:0]  CURRENT
);
```

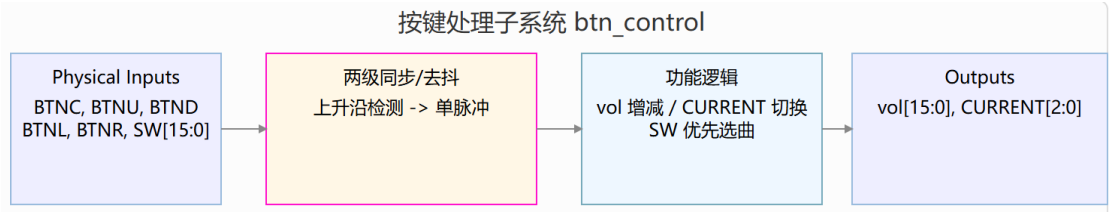
输入：CLK（主时钟），RST（低电平有效复位），BTNC、BTNU、BTND、BTNL、BTNR（五个按键），SW[15:0]（滑动开关）

输出：vol[15:0]（音量编码），CURRENT[2:0]（当前歌曲索引）

按键处理模块负责把板上物理按键和滑动开关上的电平变化转换为系统可用的控制信号，它既要做消抖、同步，又要实现按键功能的语义（音量加减、上

一曲/下一曲、中键复位）与滑动开关。设计中采用两级同步寄存器对每个按键信号进行寄存，从而在主时钟域内产生稳定的上升沿检测脉冲。**中键被约定为恢复默认设置**（vol=16'h4040，CURRENT=0），上下键按固定步长调整音量并在边界处截断，而**左右键实现循环式的曲目切换**（0 的上一曲为 7，7 的下一曲回到 0），这些操作都在时钟上做寄存以保证与系统其他模块同步。

对于直接选曲，模块优先考虑滑动开关的状态：**当低位若干开关被置位时，立即以开关对应的编号覆盖按键引起的曲目选择。**音量采用 16 位对称编码表示（例如 0x4040 为中等音量，步长为 0x1010），这样在显示与命令构造时能直接切分高位以获得十进制等级。总体实现思路是把输入接口处理为时间稳定、事件化的控制信号并把系统行为限定为在主时钟边沿发生更新——这样任何对 vol 与 CURRENT 的采样都只需在下游模块的时序约束内读寄存器即可。



4.2 数码管显示子系统（Display7）

```
module Display7(
    input CLK,
    input [15: 0] DATA,    // 时间（秒） mm:ss
    input [15: 0] VOL,     // 音量编码，例如 0x0000, 0x1010..0xF0F0
    input [2:0]  CURRENT, // 当前歌曲索引 0..7
    output reg [6: 0] SEG,
    output reg [7: 0] SHIFT,
    output reg DOT
);
```

输入：CLK，DATA[15:0]（时间 秒），VOL[15:0]（音量编码），CURRENT[2:0]（歌曲索引）

输出：SEG[6:0]（段码），SHIFT[7:0]（位选流水），DOT（小数点/冒号控制）

数码管显示模块要以八段流水位选方式同时显示当前歌曲编号、音量等级以及经过时间（mm:ss），并用中间两位的小数点形成视觉上的冒号（七段数码管上面的冒号只是装饰无对应管脚无法实际点亮）。模块内部将所有要显示的信息

组织成一个 32 位的字，左两位承载歌曲编号与音量两位，右四位承载时间的四个十进制位；扫描时钟由参数化的分频器生成，在该较慢时钟的上升沿更新位选与段码，从而把多位显示硬件以时分复用方式驱动起来。为保证数字和符号的一致性，时间字段采用秒为单位的整数，通过经典的整数除与模运算拆分成分与秒再做 BCD 式显示；音量从 vol 的高位字段转换为 1..16 的十进制等级后拆为两位显示。

在实现上特别注意扫描频率与段码更新的配合，频率过低会产生闪烁，过高则可能造成外设无法响应，因此分频参数需结合目标板的主时钟与驱动能力选取。对 0~9 给出标准七段编码，其它值缺省为熄灭。小数点的控制与位选同步处理：在扫描到显示冒号位置的位选时拉低 DOT 以显示冒号，从而通过位选时序和点位信号共同塑造最终的视觉效果。该子系统的设计目标是保证与来自按键模块和时间计数模块的数据交互在单一时钟域内完成。

4.3 OLED 图像显示子系统（oled）

```
module oled(
    input CLK,
    input RST,
    input [2: 0] current,
    output reg DIN, // 串行数据输入
    output reg OLED_CLK,
    output reg CS, // 片选
    output reg DC, // 数据/命令选择
    output reg RES
);
```

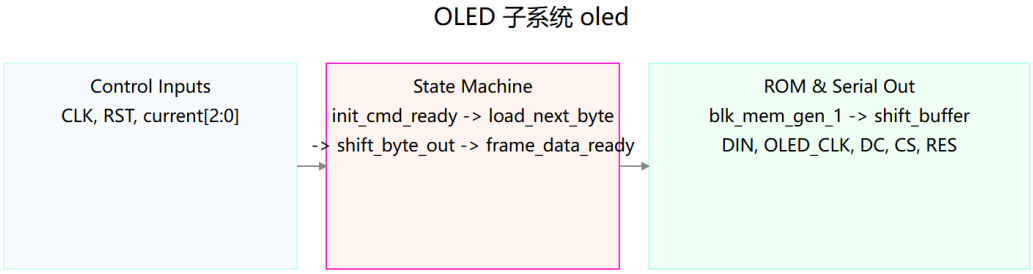
输入：CLK，RST，current[2:0]（图片索引）

输出：DIN（串行数据），OLED_CLK（串口时钟），CS（片选），DC（数据/命令选择），RES（复位）

OLED 子系统负责初始化显示控制器并周期性地从内部块 ROM 读出位图数

据逐行发送到显示器，模块既实现初始化命令表的串行下发，也实现帧数据的按行移出。设计将初始化命令与显示数据分为两个工作阶段：上电或复位后先执行初始化命令序列（每条命令按字节发送，发送期间将 DC 置为命令态），初始化完成后切换为帧数据阶段，按行从 blk_mem_gen_1 读取 1536 位的行位图并以字节为单位拆分通过 DIN/OLED_CLK 串行移出，发送期间将 DC 置为数据态并按 OLED 手册惯例控制 CS 与时序。为了稳定可靠地向外设移位，模块采用局部分频生成合适的串行时钟（例如 2MHz）并在该时钟的边沿上完成位输出与时钟翻转，从而保证移位与时钟沿的确定性。

实现要点包括：用状态机管理初始化与数据两类写入，将每条待写的数据装入移位缓冲区并以 out_byte_reg 逐位输出；通过 bytes_left 计数器与 row_idx、row_addr 管理行数据发送与 ROM 读地址的推进；对 RES 线进行正确的复位管理，在复位释放后才开始发送初始化命令序列。ROM 地址以{current,row_addr}拼接以支持按索引选择不同图片，便于在歌曲切换时展示相应封面或界面。这里最关键的点在于要充分理解 **SPI 协议**，且需要逐比特**读取 BMP 图像信息**从而在 OLED 上还原出正确的图像。



4.4 MP3 音频子系统及 VS1003 播放状态机

```
module MP3(
    input CLK, // 主时钟输入
    input DREQ,// VS1003 数据请求输入
    input RST, // 低电平有效复位
    input [15:0] vol,// 音量值（来自按键模
    input [2: 0] current, // 歌曲选择
    output reg XDCS, // 数据片选
    output reg XCS,  // 命令片选
    output reg RSET,
    output reg SI,  // 数据输入
    output reg SCLK, // VS1003 时钟
    output reg MP3_RST,
    output reg [15:0] led // 音量显示
);
```

输入：CLK，RST，DREQ（外设数据请求），

vol[15:0]（音量编码），current[2:0]（歌曲索引）

输出：XDCS（数据片选），XCS（命令片选），RSET，SI（串行数据输入），SCLK（串行时钟），MP3_RST，led[15:0]

MP3 子系统与外部 VS1003 类解码器之间的交互采用**状态机**来管理启动、寄存器设置、音量调整与数据流传输，总体上播放流程可划分为若干状态：WAITING（等待）、H_RESET（硬件复位）、S_RESET（软件复位/命令发送）、SET_VOL（设置音量）、LOAD_DATA（准备数据）以及 PLAY（数据传输/播放）。系统在 WAITING 状态会先做必要的延时以确保不会错过播放信号——这是因为若在设备尚未稳定时就立即发送 SCI 指令，可能导致控制总线错误或解码器进入不可预测状态；因此延时是保证后续命令可靠传输的防护措施。延时结束进入 H_RESET 以后，通过控制外设的复位引脚完成硬件级别的初始化，随后进入 S_RESET 阶段逐位通过 SCI 向芯片写入初始化命令，写入过程须在 DREQ 为高时进行并在每次命令传输期间将 XCS 拉低直至 32 位完成，然后再把 XCS 拉高。

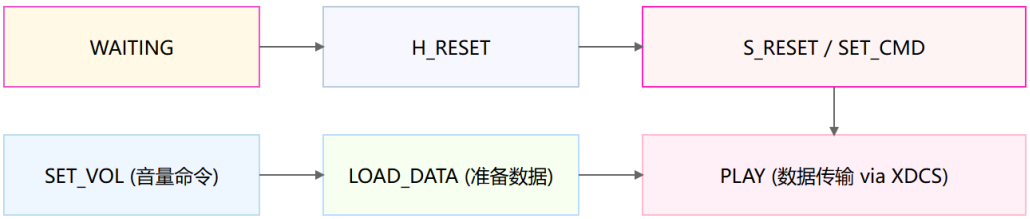
音量设置（SET_VOL）与软件复位阶段在实现细节上类似：构造好要写入的 sci_cmd 并在 DREQ 为高且 SCLK 按约定状态下逐位传输，XCS 在整个 32 位传输期间保持低电平。**为了支持播放过程中的动态音量调整，设计中引入了 LOAD_DATA 状态作为中间态**：当正在播放且检测到 vol 发生变化时，状态机会把当前传输停在安全点，将 XCS 拉高以终止命令通道，再跳转到 SET_VOL 以发送新的音量命令；命令发送完成后返回 LOAD_DATA 或直接回到 PLAY 以继续数据流。LOAD_DATA 的存在使得音量变更能够被及时响应且不会破坏流数据的完整性。

在 PLAY 状态下，音频数据以字为单位（例如 16 位）从内部 ROM 读取到移位寄存器并在 DREQ 为高的情况下逐位移出到 SI，同时将 XDCS 拉低以表示数据片选，传输完成后再拉高 XDCS 并递增 ROM 地址以准备下一字。一个关键的实现细节是对 DREQ 的检测频度：建议在每个位或每字传输时都检查 DREQ，而不是仅在一整块数据传输完成后再检测，这样一旦 DREQ 失效可以立即中断当前字的传输并释放片选，从而避免因持续传输而造成的播放停顿或外设缓存溢出。状态机在位传输层面还需严格管理 SCLK 的翻转与 SI 的稳定性——这里采用**手动翻转 SCLK** 的方法：在每一位的时序操作后显式对 SCLK 作取反以产生

可控时钟沿，保证移位寄存器在确定的时钟沿采样或输出。

为保证片选与时序的正确性，命令传输与数据传输使用不同的片选线（XCS 用于 SCI，XDCS 用于 SDI 数据），且必须保证在任何一次传输过程中两类片选不会冲突；在进入发送命令或发送数据前务必对片选进行适当的置位/复位并在传输完成后立即恢复默认电平。最后，ROM 的寻址采用{current,song_word_addr}的拼接方式，以实现多首歌曲线性存储和索引；状态机对 song_word_addr 的推进负责连续播放的连续性，而在曲目切换或复位事件发生时须把该地址复位以避免数据错位。

MP3 子系统与播放状态机



4.5 时钟分频与计时子系统（Divider 与 time_counter）

```
module Divider #(parameter Time=20)
(
    input I_CLK,
    output reg O_CLK
);
```

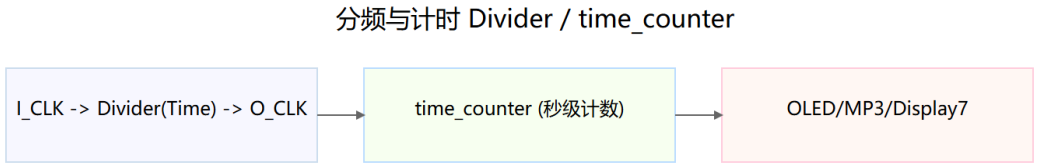
Divider: I_CLK（输入主时钟），O_CLK（输出分频时钟），参数 Time（分频周期参数）

time_counter: CLK（主时钟），RST，Time[15:0]（输出秒计数）

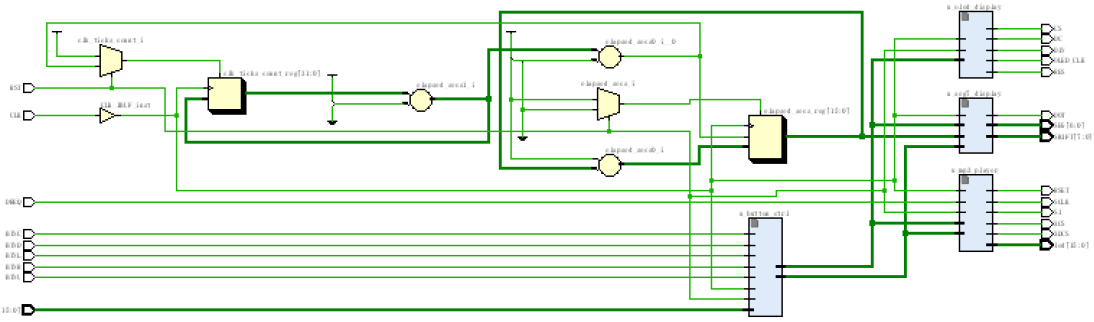
系统中多个子模块需要不同频率的时钟：OLED 的串行时钟、MP3 的 1MHz 发送时钟以及数码管的扫描时钟都来自于主时钟的分频器，因此采用参数化的 Divider 模块以在硬件层面生成稳定的低速时钟信号。Divider 通过计数到预设阈值后翻转输出，从而近似产生 50%占空比的方波，分频参数可按目标外设要求设置；对时钟分频的实现要点是保证输出时钟在设计的时钟域内与移位逻辑同步，

同时注意分频参数的奇偶会对占空比产生影响。另一方面，系统的秒级计时由 time_counter 实现，主时钟周期计数到与主频相符的阈值（例如 100000000 以匹配 100MHz 主频）后递增秒计数，该时间值被 Display7 直接采样用于显示 mm:ss。

在工程实践中要谨慎处理由分频器生成的多个时钟域之间的信号交互。虽然在局部状态机中直接使用分频时钟可以简化时序，但跨域信号如果不做同步会引入风险。为此，设计应尽量把控制信号的存取限定在单个时钟域内，或者在跨域边界处使用两级同步寄存器机制稳健地传递数据。总之，此子系统的核心任务是为视频/音频/显示子模块提供节拍与时间基准，同时保证各自时序约束下的数据一致性与可靠性。



4.6 RTL 图分析



RTL 图的最左侧是系统的时钟与复位入口，并行出现了若干进行条件选择和寄存的基础单元。主时钟 CLK 与低电平有效复位 RST 首先进入若干组合/同步单元：有一个三态的多输入选择器和一个或多个 RTL_REG_SYNC 寄存块，这些单元负责把原始外部输入（按键、滑动开关等）采样并同步到主时钟域。紧邻这些同步单元的是顶层的秒计时器逻辑：从图中可见有一个加法器和一个比较器/等于判断器，等于判断器在检测到计时计数达到阈值（在源码中为 100000000）时触发把计数器复位并对 elapsed_secs 做 +1 操作；这部分在 Verilog 中对应顶层 always 块中对 clk_ticks_count 的累加与 elapsed_secs 的递增。也可以在

RTL 图中看到多个 RTL_REG_SYNC（寄存器组）用于在时序路径中打断组合链，从而把“计数产生->比较->条件选择->写回寄存器”变为时钟门控下的同步数据流。

在计时器附近有一组从外部引入的 DREQ、五个按键 BTNC/BTNU/BTND/BTNL/BTNR 与 SW[15:0] 总线，它们首先进入 btn_control 的实例（图中 u_button_ctrl ），说明按键与滑动开关的同步与去抖逻辑已被模块化封装。btn_control 的输出包括 vol[15:0] 与 CURRENT[2:0]，它们以总线形式分发到右侧的多个消费者（Display7、MP3、OLED）。从 RTL 图结构可见，btn_control 输出被寄存并驱动多路用途。

沿着图从左向右，会看到 elapsed_secs 的寄存块（RTL_REG_SYNC）把计时值稳定后分出多路，这些输出同时驱动 Display7（数码管）以实现 mm:ss 显示。中间区域可辨的电路包括数个加法/比较/复位选择器，它们实现顶层对 elapsed_secs 的比较、加法与条件更新（如等于比较用于判断某个阈值触发）。这些算术/比较器与寄存电路均在主时钟域中并行存在——换言之，设计把“计数”和“显示采样”放在同一时钟域，通过寄存器共享计时结果，避免跨域问题。从结构上看，中部还包含多个走线将 CURRENT 与 vol 广播到右侧模块，以及一条粗线连接 elapsed_secs 的高位到一个等于比较器，这正对应 Verilog 中把时间计数转成 mm:ss 的分支逻辑（显示层面的拆分）。图中每个信号穿过的 RTL_REG_SYNC 表示在信号跨越多个区域时做了寄存以改善时序。

最右侧是若干功能模块的实例化块：有 u_oled_display、u_seg7_display（Display7）和 u_mp3_player（MP3）。这些模块以矩形块显示，块的左侧接入控制/数据输入（CLK、RST、current、vol、DATA/elapsed_secs 等），右侧是对外的物理引脚（例如 OLED 的 DIN/OLED_CLK/CS/DC/RES、MP3 的 RSET/SCLK/SI/XCS/XDCS 以及 led[15:0]），以及内部连接到 ROM 的地址/数据总线。

OLED 块的地址端与一个名为 blk_mem_gen_1 的 ROM 相连，ROM 的地址由 {current, row_addr} 组合而成（图中表现为多线合并后作为 ROM addra），ROM 的输出（大位宽）被送入 OLED 模块内部的 shift_buffer/移位器，然后通过 DIN 和 OLED_CLK 以串行方式输出到外设。RTL 图把 init_cmds（初始化

命令序列）和帧数据路径都在 oled 的内部状态机中实现，图上也表现出 DC（命令/数据选择）与 CS 的控制线路，说明控制信号与数据信号在模块内部被按状态机驱动。

Display7（数码管）模块在 RTL 图上接收 DATA（elapsed_secs）、VOL 与 CURRENT，其输出为 SEG[6:0] 与 SHIFT[7:0]、DOT。Display7 在图上被描绘为一个较小的逻辑块，输入被寄存并在 Divider 生成的扫描时钟下驱动段码与位选；RTL 层面上这些是查表译码与位移寄存的组合，图中并不显示内部详图但能看到寄存/输出链路。

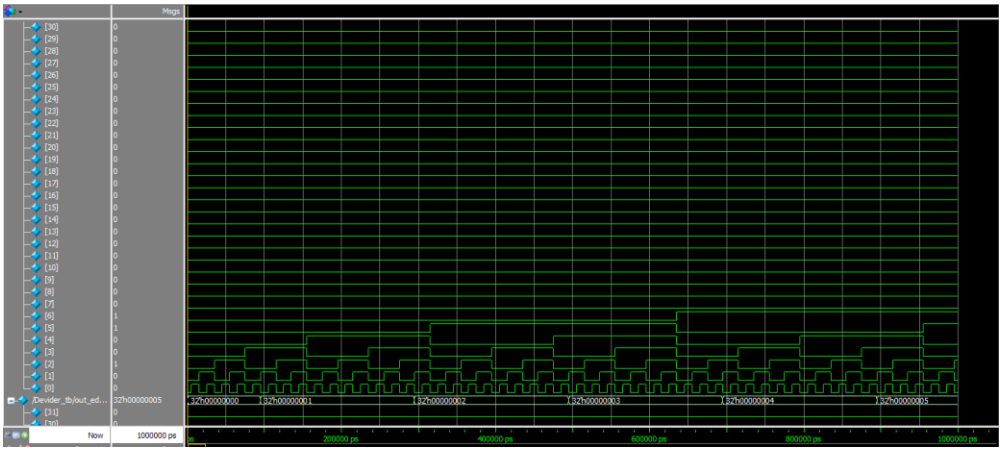
MP3 模块一目了然连接到 blk_mem_gen_0（音乐数据 ROM），ROM 的地址线同样由 {current, song_word_addr} 组成，输出的 16 位数据进入 MP3 的 payload_shift_reg 并在 SCLK 时钟控制下逐位移出到 SI。RTL 图显示 MP3 对 DREQ 的输入直接接到状态机逻辑，且 MP3 对 XCS 与 XDCS 的控制在模块内部通过状态机生成；LED 总线从 MP3 模块输出到顶层并驱动板上 LED（一热显示当前歌曲）。图中还可以看到 MP3 模块右侧对 SCLK 的手动翻转样式（在综合网表中表现为对 SCLK 寄存与翻转逻辑），这符合源码中在分频时钟上显式翻转 SCLK 的做法。

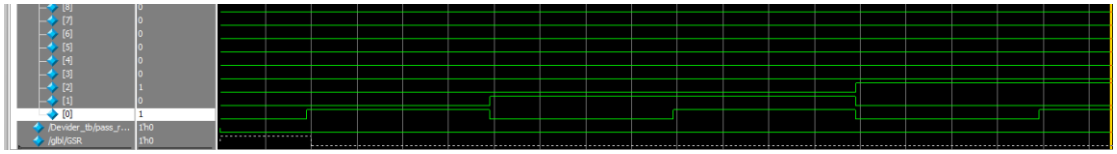
总的来说，外部按键和开关经过 u_button_ctrl 同步去抖生成 vol 与 CURRENT，这两路控制总线分别被广播到 u_mp3_player（用于选择当前歌曲和音量命令构造）、u_oled_display（用于选择封面）与 u_seg7_display（用于数码管显示）。计时器 elapsed_secs 在顶层计数并写回寄存，其寄存输出同时被送到数码管模块以显示 mm:ss。两个 ROM（blk_mem_gen_0 用于音乐数据，blk_mem_gen_1 用于位图）通过拼接地址 {current, address} 支持多首歌曲和多幅图片的索引机制，使得切歌时只需改变 current 即可从 ROM 读取新数据。

控制信号方面，MP3 与 OLED 的外设交互都通过模块内部状态机来管理时序：MP3 在收到外设 DREQ 时才进行数据位传输（图里 DREQ 线直接连到 MP3），并在传输时控制 XDCS/XCS 与 SCLK，而 OLED 在初始化和帧写入期间控制 DC/CS/DIN/OLED_CLK。

五、测试模块建模

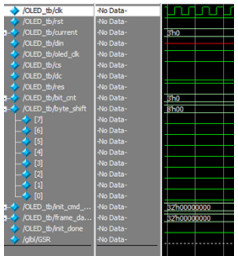
5.1 Devider_tb 测试模块





上图展示了测试的 in 波和分频后的 out 波。1μs 窗口内，输入时钟为 100MHz（周期 10ns）且保持稳定方波，每个上升沿驱动输入沿计数器累计，至第 100 次上升沿时触发一次判据检查；被测分频器 Divider(Time=20) 将输入等比分为 5MHz（周期 200ns）输出，因此在同一窗口内输出时钟呈现占空比近 50% 的规律方波，其上升沿大约每 200ns 发生一次，输出沿计数在 1μs 内应增长 5 次；实际波形中 in_edges==100 且 out_edges==5 与理论一致，判据信号 pass_ratio 被置位为 1，定量（沿数比值）与定性（周期测量）两方面同时验证了 f_out = f_in / Time 的分频关系与输出时序稳定性，该模块在 6.7 小作业里也已经测试过，可以正常完成分频，为各个模块提供最底层的时序基础。

5.2 OLED_tb 测试模块

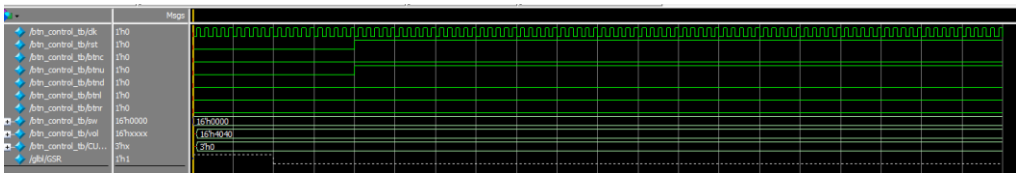


OLED_tb 模块的复位信号 rst 在初始阶段拉高，随后释放。被测模块 oled 的 SPI 接口信号（din、oled_clk、cs、dc、res）与内部测试计数（bit_cnt、byte_shift、init_cmd_bytes、frame_data_bytes、init_done）均被添加至波形窗口，便于分阶段观测。仿真初期，res 由低变高，模块进入初始化命令写入阶段。此时 cs 拉低、dc 保持为 0，oled_clk 持续翻转，SPI 串行数据 din 按字节移入。每累计 8 位（bit_cnt 归零）即完成一个字节写入，init_cmd_bytes 计数随之递增。波形中可见 init_cmd_bytes 从 0 逐步增加，直至达到 48，init_done 信号随之拉高，说明初始化正常。

进入数据写入阶段后，dc 由 0 跳变为 1，cs 依然周期性拉低，oled_clk 继续翻转。此时每 8 位 SPI 数据写入时，frame_data_bytes 计数持续递增，表明帧数据流已开始并稳定输出。若在仿真窗口内切换 current，可进一步验证页地址推进与数据流不中断。

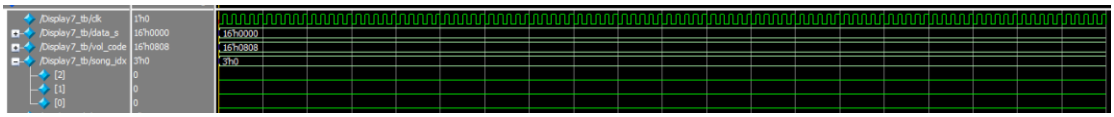
整个仿真过程中，oled_clk、cs、dc、din 等信号时序与 SPI 协议严格对应，init_cmd_bytes、frame_data_bytes 的计数增长与阶段切换均与设计预期一致，证明 OLED 控制器的初始化与数据写入流程正确无误。

5.3 btn_control_tb 测试模块



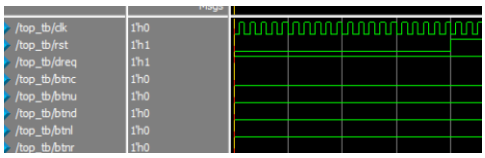
btn_control_tb 模块采用 100MHz 主时钟驱动，复位信号 rst 在初始阶段拉高，随后释放。testbench 依次对 btnu、btnd、btnl、btnr 等输入端口施加脉冲激励，并多次切换 sw 信号，模拟音量调节、歌曲切换和滑块强制选曲等操作。仿真波形中，btnu、btnd、btnl、btnr 等信号在指定时刻产生高电平脉冲，sw 信号在仿真过程中多次变化。btn_control 对这些激励做出响应，输出的 vol 和 CURRENT 信号随按键和 sw 的变化而发生跳变。具体表现为，按下 btnu 时 vol 递增，按下 btnd 时 vol 递减，按下 btnl 和 btnr 时 CURRENT 在 0~7 之间循环切换，sw 信号有效时 CURRENT 直接跳转到对应的歌曲编号。整个仿真过程可以验证 btn_control 模块的音量调节、歌曲切换和滑块优先级等功能逻辑是正确的。

5.4 Display7_tb 测试模块



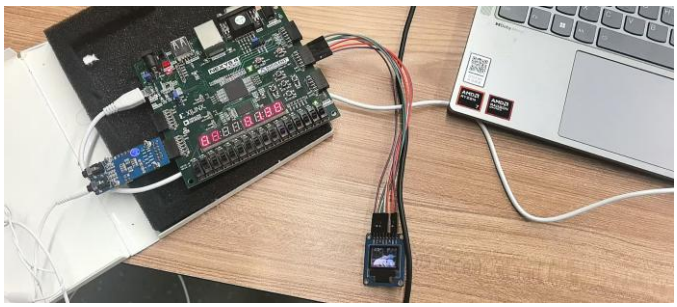
data_s 按周期递增，vol_code 由 0808 变为 1010，song_idx 切至 5；显示扫描稳定运行，shift 循环移位选位，dot 在中间两位点亮形成冒号效果；seg 按当前选位的 BCD 编码输出(0→1000000,1→1111001,2→0100100...9→0010000)，扫描与编码映射正常。

5.5 top_tb 模块

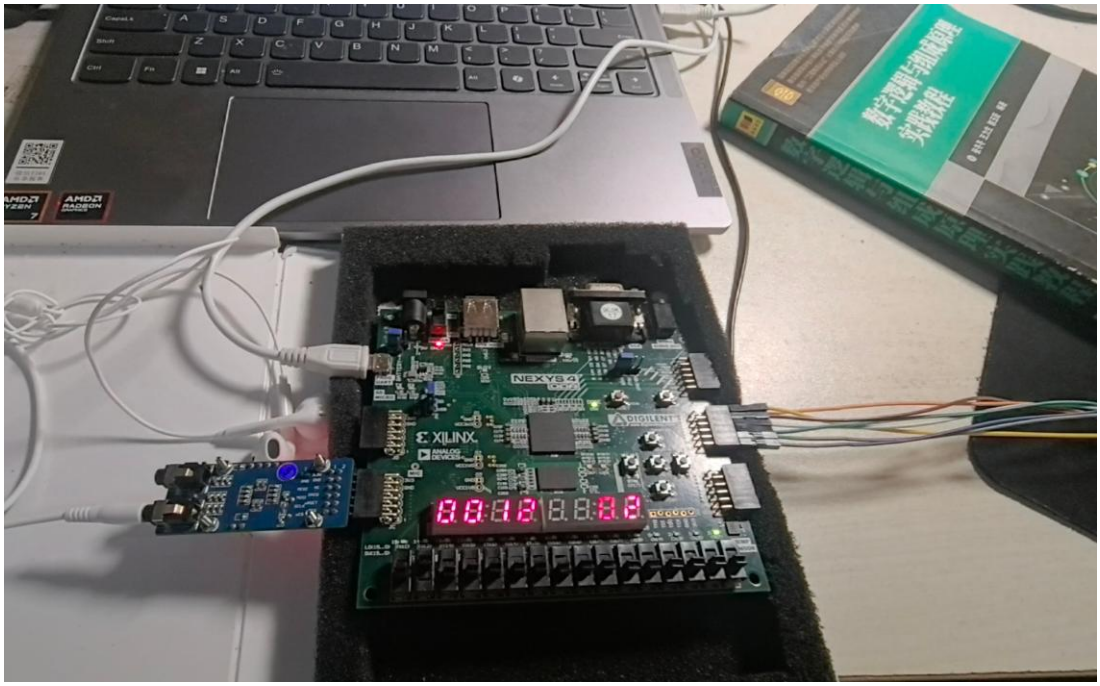


顶层输入按键 btnl/btnr 在指定时刻产生高电平脉冲，sw 由 0000 变为 0001 触发一次测试；OLED 侧 cs 在窗口中部两次拉低并保持一段时间，期间 dc 为 1 说明处于数据写入阶段，紧随其后 cs 回到高电平，表明本次数据片选周期结束；MP3 侧 xdcs/xcs 在窗口内均保持高电平未活动，si/sclk 稳定为高电平，说明本窗口内无音频命令/数据传输(与 dreq 设定相关)；Display7 侧 shift 为 8'h7f，当前选位为最低位，seg 输出为 7'hxx，dot 为 x(初期未赋值)，在更长的时间窗内 shift 循环移位、seg 按选位 BCD 编码变化；顶层 led 为 16'h0001，呈一热显示当前歌曲索引 0，符合顶层设计的歌曲指示方法。

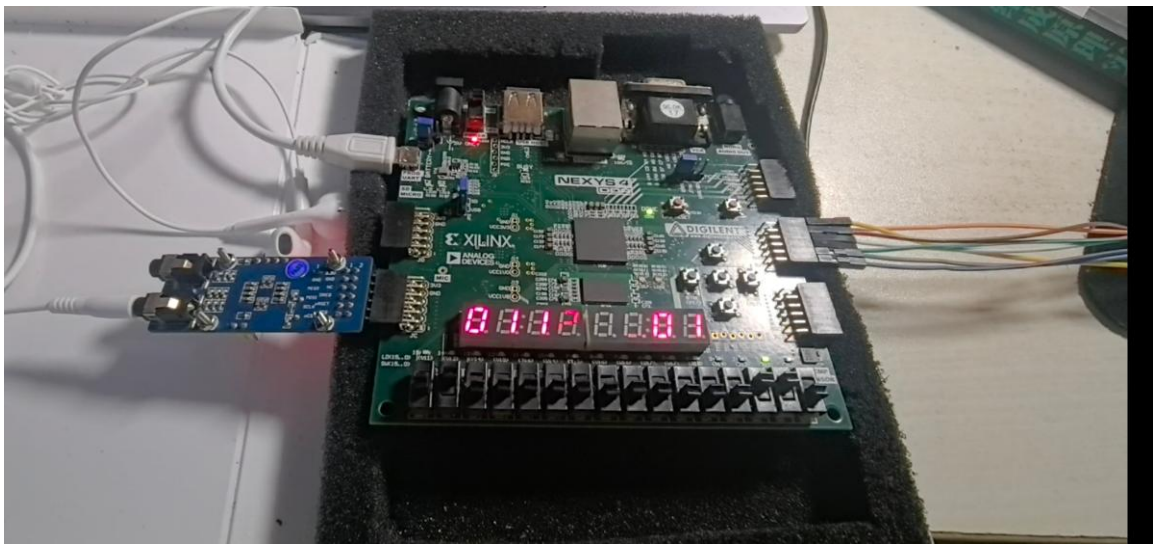
六、实验结果



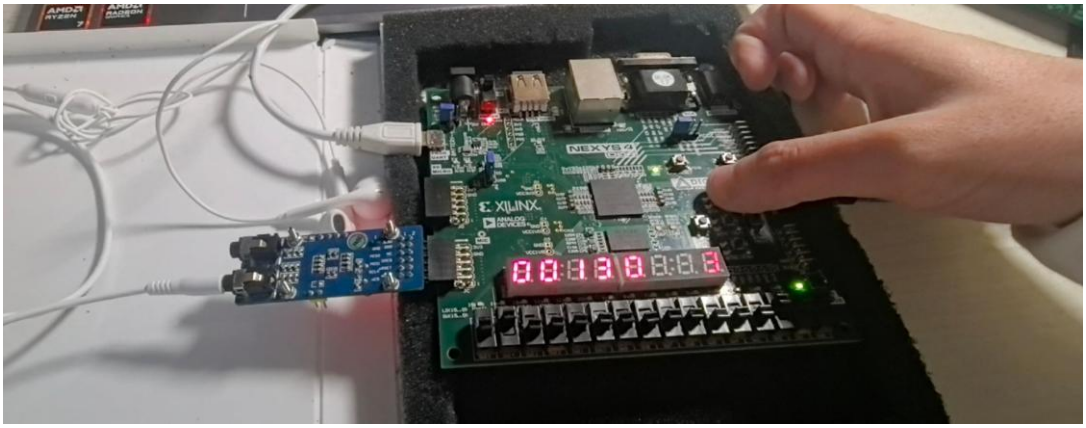
上图为开发板、MP3 模块、OLED 模块初次完美运行时所拍摄。



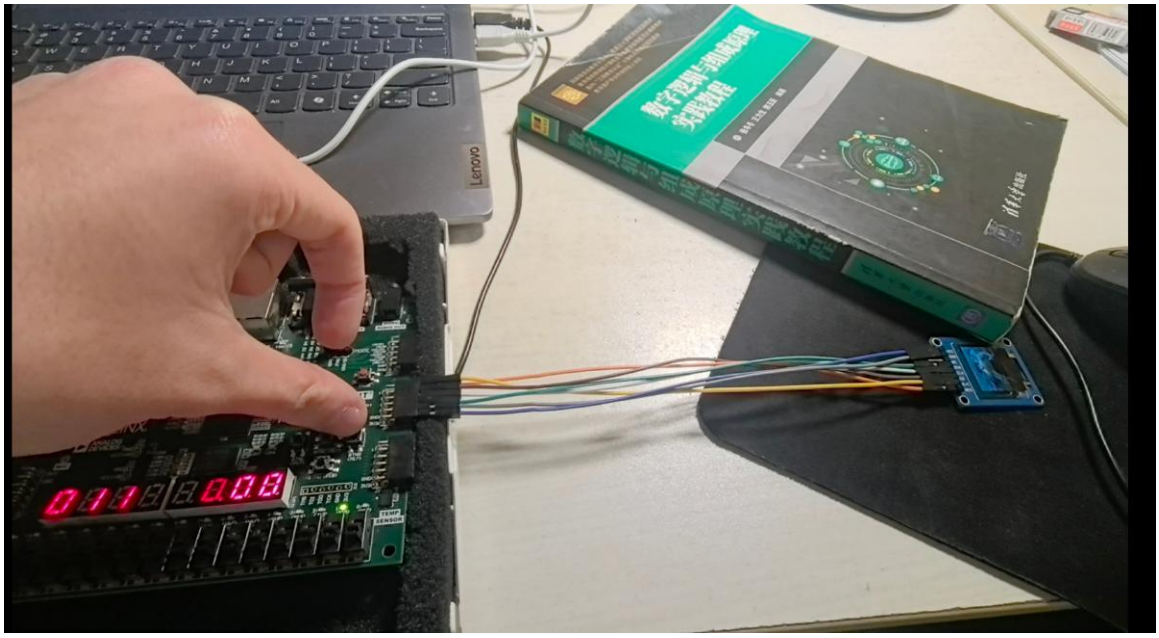
初始状态，**SW[14]为高电平，数字系统开始运行**，七段数码管 00 表示当前播放歌曲为 0，LED 灯的 LD[0]亮起起到相同的效果，提示现在是第 0 首歌曲；12 为音量（共 16 级，1-16 依次音量增大），初始化为 12，12 接近耳机可被外部听到的最低音量，右侧 00.02 表示当前歌曲进度条为 00 分 02 秒。



SW[0]-[7]开关为高电平时可直接跳转到相应序号的歌曲，上图 SW[2]和 SW[1]都为高电平，有多个高电平时为防止播放混乱采用了类似**优先 38 译码器**的设计思路，实际播放最低数码对应歌曲，此例播放 01 号歌曲，LED 灯和数码管前 2 位都可以证明这一点。



上图为通过按上下按钮调节音量的示意图，可以看到音量变为了 13 级。



上图按左右按钮可以调节播放第几首歌曲，同时右侧的 OLED 模块展示了当前歌曲对应的图片（可人为设定为和歌手/歌曲相关的图片），MP3 和 OLED 会通过 CURRENT[2:0]同步变化。

七、结论

本次综合实验主要完成了 MP3 播放、OLED 显示与七段数码管显示的协同。系统在 100MHz 主时钟下运行，顶层以同步复位为主线，将 btn_control、MP3、oled、Display7 与 Divider 等子模块进行集成，外部通过 VS1003 完成音频解码，通过 blk_mem_gen_0 与 blk_mem_gen_1 分别提供歌曲与位图数据。最终系统达成预期：

一是功能完整。**按键实现音量增减、上一曲/下一曲与默认复位，SW[7:0]支持直接选曲；OLED 稳定显示与歌曲索引关联的位图；七段数码管扫描显示mm:ss、音量等级与歌曲编号；MP3 模块**在 DREQ 下实现命令与数据分路，能在播放过程中即时插入音量命令并继续数据流。

二是时序与接口关系清晰。系统把输入先同步去抖，再在主时钟域内寄存后扩展到各模块；外设侧采用分频得到局部串行时钟（如 MP3 的 1MHz、OLED 的 2MHz），移位在局部时钟沿上进行。通过**寄存与有限状态机约束**，XCS/XDCS、DC/CS、SCLK/OLED_CLK 等引脚与数据路径保持合理正确的因果关系。

三是资源与可扩展性兼顾。ROM 寻址采用{current,addr}拼接，既简洁又易扩展；**只需扩展 current 位宽、更新 COE 文件与地址数量，即可增加歌曲/图片数量**。

四是验证路径明确。**工程包含所有 module 对应的仿真 testbench**，可供随时调试。对 VS1003 的 DREQ 时序、命令/数据片选分离以及 SCLK 翻转策略，均有可观测的信号。

综上，系统在功能、时序与可维护性方面达到了设计目标。

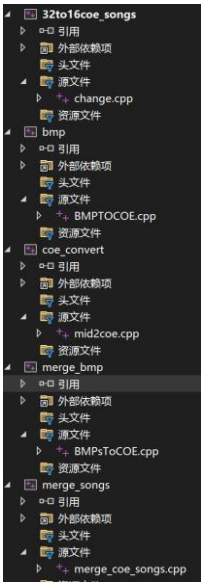
八、心得体会及建议

本次实验对我而言最直观的体会是数字系统设计的复杂性远超理论学习。之前小作业各模块功能明确、独立，但在实际项目中，其间错综复杂的互联的时序匹配以及严密的数据流控制等问题构成的系统非常复杂。将抽象功能需求转化为具体的硬件描述语言 Verilog 代码，并最终使其在 FPGA 芯片上稳定运行，需要严谨的逻辑思维、扎实的基础知识及解决实际问题的耐心。和软件的调试比起来硬件的调试往往花的时间很长且很难定位问题在哪里，这个时候就需要认真对照参考文档和实际代码，尝试找到差异，或者和同学交流共性问题出在哪里。

项目初期，**模块分解和系统架构设计的重要性尤为突出**。MP3 音乐播放器系统最主要需要学习的内容是怎么处理时序，以及怎么设计 IP 核从而正确地把音频文件解码成对应的 COE 格式。在代码编写之前，应该投入充足时间进行详细设计规划，绘制系统框图、时序图及状态机流程图，能显著减少后续开发和调试阶段的返工。以下是代码编写的时候总结出的一些重点：

1. 本次作业最大的挑战在于要学习 IP 核使用和 SCI 协议，也是之前小作业里从未涉及过的内容。网上、参考文档里都有很详尽的资料，但依然有很多易错点，比如 blk_mem_gen 的位宽/深度、初始化文件路径、端序排列。位宽要与外设帧宽一致，否则移位边界会错；深度要与资源长度配平，否则会出现读穿或听到杂音/看到花屏。

2. 无论是 VS1003、OLED 控制器，还是 Vivado 的 IP 核向导，最省时间的做法就是对照手册逐项核对。比如 VS1003 的 SCI/SDI 时序、命令帧格式、DREQ 语义，OLED 的初始化命令表、DC 与 CS 的时序窗口、字节流顺序，这些如果不先过一遍手册，很容易在板上反复试错。



3. 有经验的开发会大大提升效率。本学期面向对象程序设计课程里涉及到过对 BMP 不同位色存储格式的详细解析，在处理 OLED 图像从 BMP 往 COE 转化的时候起到了很大作用，研究 OLED 的显示所花的时间很短。对于一些诸如 COE 文件转化的内容，可以考虑编写脚本处理。右侧是使用 C++方式编写的单个/多个 BMP/MID 向 COE 转化的程序，这样如果需要改数字系统的音频/照片就只需要有原始的 BMP/MID 格式，运行脚本后再重新配置 IP 核生成比特流即可，可以提高数字系统的扩展性。

关于建议方面，由于硬件上手难度大且调试时间长，可以考虑在前期小作业就加入一些对 IP 核或者 SCI 协议的讲解（也可放在理论课上），这样会降低大作业的上手难度。

九、附录

top.v:

```
module top(
    input CLK, //  输入主时钟，系统时钟
    input RST, //  低电平有效复位
    // MP3，接 JC
    input DREQ,          // VS1003  数据请求信号
    output XDCS, //  数据片选
    output XCS,   //  命令片选
    output RSET,
    output SI,     //  数据输入
    output SCLK,   // VS1003  时钟
    //  按键
    input BTNC,
    input BTNU,
    input BTND,
    input BTNL,
    input BTNR,
    //  开关用于直接选择歌曲
    input [15:0] SW,
    // 7  段数码管
    output [6: 0] SEG,
    output [7: 0] SHIFT,
    output DOT,
    // OLED  接口，接 JB
    output DIN,
    output OLED_CLK,
    output CS,
    output DC,
    output RES,
    //  音量显示  LED
    output [15: 0] led
);
    wire [15:0] vol_code_bus;   //  音量（由按键模块驱动）
    wire [2:0] song_idx_bus;//  当前歌曲索引，目前最多 8 首
    reg [15:0] elapsed_secs;//  时间计数  (秒)
    integer clk_ticks_count; //  主时钟计数，用于秒级累加
    wire mp3_rst_sync; //  用于  MP3  的复位/显示复位信号
     // 7  段数码管显示
        Display7 u_seg7_display(
            .CLK(CLK),
            .DATA(elapsed_secs),
```

```
            .VOL(vol_code_bus),
            .CURRENT(song_idx_bus),
            .SEG(SEG),
            .SHIFT(SHIFT),
            .DOT(DOT)
        );
    //  按键控制
    btn_control u_button_ctrl(
            .CLK(CLK),
            .RST(RST),
            .BTNC(BTNC),
            .BTNU(BTNU),
            .BTND(BTND),
            .BTNL(BTNL),
            .BTNR(BTNR),
            .SW(SW),
            .vol(vol_code_bus),
            .CURRENT(song_idx_bus)
    );
    // OLED  模块
    oled u_oled_display(
            .CLK(CLK),
            .RST(RST),
            .current(song_idx_bus),
            .DIN(DIN),
            .OLED_CLK(OLED_CLK),
            .CS(CS),
            .DC(DC),
            .RES(RES)
    );
    // MP3  模块
    MP3 u_mp3_player(
            .CLK(CLK),
            .RST(RST),
            .DREQ(DREQ),
            .vol(vol_code_bus),
            .current(song_idx_bus),
            .XDCS(XDCS),
            .XCS(XCS),
            .RSET(RSET),
            .SI(SI),
            .SCLK(SCLK),
            .MP3_RST(mp3_rst_sync),
            .led(led)
```

```
    );
    //计数，顶层模块
    always @ (posedge CLK) begin
        if(!RST) begin
            elapsed_secs <= 16'd0;
            clk_ticks_count <= 0;
        end else if((clk_ticks_count+1)==100000000) begin
            clk_ticks_count <= 0;
            elapsed_secs <= elapsed_secs + 1;
        end else begin
            clk_ticks_count <= clk_ticks_count + 1;
        end
    end
endmodule
```

MP3.v:

```
module MP3(
    input CLK, //  主时钟输入
    input DREQ,// VS1003  数据请求输入
    input RST, //  低电平有效复位
    input [15:0] vol,//  音量值（来自按键模块）
    input [2: 0] current, //  歌曲选择
    output reg XDCS, //  数据片选
    output reg XCS,    //  命令片选
    output reg RSET,
    output reg SI,    //  数据输入
    output reg SCLK, // VS1003  时钟
    output reg MP3_RST,
    output reg [15:0] led //  音量显示
);
    //  状态定义
    parameter boot_wait_cmd_ready = 0;
    parameter shift_cmd_bits = 1;
    parameter stream_data = 2;
    parameter shift_data_bits = 3;
    parameter boot_delay = 4;
    parameter wait_vol_cmd_ready = 5;
    parameter shift_vol_bits = 6;
    parameter pause_state = 7;
    reg [2: 0] mp3_state;

    //  基本参数
    parameter DELAY_TIME = 500000;
```

```
    parameter CMD_NUM = 2;

    // 1MHz 分频时钟（使用 Divider）
    wire clock_divider_1m;
    Divider #(.Time(100)) u_clock_divider_1m(CLK, clock_divider_1m);

    // 歌曲地址选择
    reg[2: 0] prev_song;
    reg[11:0] song_word_addr;

    // IP 核 ROM
    wire [15: 0] song_word_in;
    reg [15: 0] payload_shift_reg;
    blk_mem_gen_0                    music_rom(.clka(CLK),.ena(1),.addra({current,
song_word_addr}),.douta(song_word_in));

    // 命令寄存器
    reg pause_flag;
    reg [63: 0] pause_cmd_frame = {32'h02000808, 32'h02000800};
    reg [63: 0] cmd_frame = {32'h02000804, 32'h020B0000};
    reg [2: 0] frame_idx = 0;

    // 变量
    integer boot_tick_count = 0;
    integer bit_index = 0;
    reg [31: 0] vol_cmd_frame;

    // 合并 vol_control：一热 LED 显示当前歌曲（在主时钟上采样）
    always @ (posedge CLK) begin
        led <= (16'b1 << current);
    end

    always @ (posedge clock_divider_1m) begin
        prev_song <= current;
        if(!RST || prev_song!=current) begin
            MP3_RST <= 0;
            RSET <= 0;
            SCLK <= 0;
            XCS <= 1;
            XDCS <= 1;
            boot_tick_count <= 0;
            mp3_state <= boot_delay;
            frame_idx <= 0;
            song_word_addr <= 0;
```

```
end
else begin
    case (mp3_state)
        boot_wait_cmd_ready: begin
                SCLK <= 0;
                if(frame_idx == CMD_NUM) begin
                    mp3_state <= stream_data;
                end
                else if(DREQ) begin
                    mp3_state <= shift_cmd_bits;
                    bit_index <= 0;
                end
            end

        shift_cmd_bits: begin
                if(DREQ) begin
                    if(CLK) begin
                        if(bit_index==32) begin
                            frame_idx <= frame_idx+1;
                            XCS <= 1;
                            mp3_state <= boot_wait_cmd_ready;
                            bit_index <= 0;
                        end
                        else begin
                            XCS <= 0;
                            SI <= cmd_frame[63];
                            cmd_frame <= {cmd_frame[62: 0], cmd_frame[63]};
                            bit_index <= bit_index+1;
                        end
                    end
                    SCLK <= ~SCLK;
                end
            end

        stream_data: begin
                // 检测音量变化
                if(vol[15:0] != cmd_frame[15: 0]) begin
                    mp3_state <= wait_vol_cmd_ready;
                    vol_cmd_frame <= {16'h020B, vol};
                    cmd_frame[15: 0] <= vol[15: 0];
                end
                else if(DREQ) begin
                    SCLK <= 0;
                    mp3_state <= shift_data_bits;
```

```
                    payload_shift_reg <= song_word_in;
                    bit_index <= 0;
                end
            end

            shift_data_bits: begin
                if(SCLK) begin
                    if(bit_index == 16) begin
                        XDCS <= 1;
                        song_word_addr <= song_word_addr+1;
                        mp3_state <= stream_data;
                    end
                    else begin
                        XDCS <= 0;
                        SI <= payload_shift_reg[15];
                        payload_shift_reg <= {payload_shift_reg[14:0],
payload_shift_reg[15]};
                        bit_index <= bit_index+1;
                    end
                end
                SCLK = ~SCLK;
            end

            wait_vol_cmd_ready: begin
                if(DREQ) begin
                    mp3_state <= shift_vol_bits;
                    bit_index <= 0;
                end
            end

            shift_vol_bits: begin
                if(DREQ) begin
                    if(SCLK) begin
                        if(bit_index==32) begin
                            XCS <= 1;
                            mp3_state <= stream_data;
                            bit_index <= 0;
                        end
                        else begin
                            XCS <= 0;
                            SI <= vol_cmd_frame[31];
                            vol_cmd_frame <= {vol_cmd_frame[30: 0],
vol_cmd_frame[31]};
                            bit_index <= bit_index+1;
```

```
                    end
                end
                SCLK <= ~SCLK;
            end
        end

        boot_delay: begin
            if(boot_tick_count == DELAY_TIME) begin
                boot_tick_count <= 0;
                MP3_RST <= 1;
                mp3_state <= boot_wait_cmd_ready;
                RSET <= 1;
            end
            else boot_tick_count <= boot_tick_count+1;
        end
        default: ;

    endcase
  end
end
endmodule
```

OLED.v:

```
module oled(
    input CLK,
    input RST,
    input [2: 0] current,
    output reg DIN, //  串行数据输入
    output reg OLED_CLK,
    output reg CS, //  片选
    output reg DC, //  数据/命令选择
    output reg RES
);
    parameter DELAY_TIME = 25000;
    // DC  标志位
    parameter CMD = 1'b0;
    parameter DATA = 1'b1;
    //  初始化命令表
    reg [47:0] init_cmds [9:0];
    initial
        begin
            init_cmds[0]= {8'hAE, 8'hA0, 8'h76, 8'hA1, 8'h00, 8'hA2};
```

```
            init_cmds[1]= {8'h00, 8'hA4, 8'hA8, 8'h3F, 8'hAD, 8'h8E};
            init_cmds[2]= {8'hB0, 8'h0B, 8'hB1, 8'h31, 8'hB3, 8'hF0};
            init_cmds[3]= {8'h8A, 8'h64, 8'h8B, 8'h78, 8'h8C, 8'h64};
            init_cmds[4]= {8'hBB, 8'h3A, 8'hBE, 8'h3E, 8'h87, 8'h06};
            init_cmds[5]= {8'h81, 8'h91, 8'h82, 8'h50, 8'h83, 8'h7D};
            init_cmds[6]= {8'h15, 8'h00, 8'h5F, 8'h75, 8'h00, 8'h3F};
            init_cmds[7]= {8'hAF, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00};
        end

    // 像素，
    wire [1535:0] row_bitmap_word;
    reg [5: 0] row_addr;
    blk_mem_gen_1                                   bmp_rom(.clka(CLK),.ena(1),.addra({current,
row_addr}),.douta(row_bitmap_word));

    // 状态定义
    parameter shift_byte_out = 0;
    parameter load_next_byte = 1;
    parameter DELY = 3;
    parameter init_cmd_ready = 4;
    parameter frame_data_ready = 5;

    // 2MHz 时钟分频
    wire clock_divider_2m;
    Divider #(.Time(20)) u_clock_divider_2m(CLK, clock_divider_2m);

    // 变量
    reg [1535:0] shift_buffer_1536;
    reg [15: 0] row_idx;
    reg [7: 0] out_byte_reg;
    reg [3: 0] oled_state;
    reg [3: 0] next_state_after_write;
    integer bit_index = 0;
    integer bytes_left = 0;

    // 状态机：初始化命令写入或像素数据写入
    always @ (posedge clock_divider_2m) begin
        if(!RST) begin
            oled_state <= init_cmd_ready;
            row_idx <= 0;
            CS <= 1'b1;
            RES <= 0;
        end
        else begin
```

```
                RES <= 1;
                case(oled_state)
                    // 准备写命令，将 cmds 行装入 temp
                    init_cmd_ready: begin
                            if(row_idx == 8) begin
                                row_idx <= 0;
                                row_addr <= 0;
                                oled_state <= frame_data_ready;
                            end
                            else begin
                                shift_buffer_1536 <= init_cmds[row_idx];
                                oled_state <= load_next_byte;
                                next_state_after_write <= init_cmd_ready;
                                bytes_left <= 6;
                                DC <= CMD;
                            end
                        end
                    // 准备写像素数据
                    frame_data_ready: begin
                            if(row_idx == 64) begin
                                row_idx <= 0;
                                oled_state <= frame_data_ready;
                            end
                            else begin
                                shift_buffer_1536 <= row_bitmap_word;
                                oled_state <= load_next_byte;
                                next_state_after_write <= frame_data_ready;
                                bytes_left <= 192;
                                DC <= DATA;
                            end
                        end
                    // 将 temp 拆成若干 8-bit 寄存器
                    load_next_byte: begin
                            if(bytes_left == 0) begin
                                row_idx <= row_idx+1;
                                row_addr <= row_addr+1;
                                oled_state <= next_state_after_write;
                            end
                            else begin
                                out_byte_reg[7:                    0]                    <=
(next_state_after_write==init_cmd_ready)? shift_buffer_1536[47: 40]: shift_buffer_1536[1535:
1528];
                                shift_buffer_1536                                        <=
(next_state_after_write==init_cmd_ready)? {shift_buffer_1536[39: 0], shift_buffer_1536[47: 40]}:
```

```
{shift_buffer_1536[1527: 0], shift_buffer_1536[1535: 1528]};
                                    oled_state <= shift_byte_out;
                                    OLED_CLK <= 0;
                                    bit_index <= 0;
                                end
                            end
                        // 将 8 位移入 DIN
                        shift_byte_out: begin
                                if(OLED_CLK) begin
                                    if(bit_index == 8) begin
                                        CS <= 1;
                                        bytes_left <= bytes_left-1;
                                        oled_state <= load_next_byte;
                                    end
                                    else begin
                                        CS <= 0;
                                        DIN <= out_byte_reg[7];
                                        bit_index <= bit_index+1;
                                        out_byte_reg<={out_byte_reg[6:0], out_byte_reg[7]};
                                    end
                                end
                                OLED_CLK <= ~OLED_CLK;
                            end
                        default:;
                    endcase
                end
            end
endmodule
```

btn_control.v:

```
module btn_control(
    input CLK,
    input RST,                  // 低电平有效复位
    input BTNC,
    input BTNU,
    input BTND,
    input BTNL,
    input BTNR,
    input [15:0] SW,          // 滑动开关用于直接选择歌曲
    output reg [15:0] vol,
    output reg [2:0]    CURRENT
);
    // 两级同步寄存器用于消抖
```

```
    reg sync_c_0, sync_c_1, sync_u_0, sync_u_1, sync_d_0, sync_d_1, sync_l_0, sync_l_1,
sync_r_0, sync_r_1;
    reg [15:0] sw_sync_0, sw_sync_1;
    wire rise_c =   sync_c_1 & ~sync_c_0;
    wire rise_u =   sync_u_1 & ~sync_u_0;
    wire rise_d =   sync_d_1 & ~sync_d_0;
    wire rise_l =   sync_l_1 & ~sync_l_0;
    wire rise_r =   sync_r_1 & ~sync_r_0;
    // 滑动开关电平同步
    wire [15:0] sw_level_sync = sw_sync_1;

    // 两级同步消抖
    always @(posedge CLK) begin
        if(!RST) begin
            sync_c_0 <= 1'b0; sync_c_1 <= 1'b0;
            sync_u_0 <= 1'b0; sync_u_1 <= 1'b0;
            sync_d_0 <= 1'b0; sync_d_1 <= 1'b0;
            sync_l_0 <= 1'b0; sync_l_1 <= 1'b0;
            sync_r_0 <= 1'b0; sync_r_1 <= 1'b0;
            sw_sync_0 <= 16'h0000; sw_sync_1 <= 16'h0000;
        end else begin
            sync_c_0 <= BTNC; sync_c_1 <= sync_c_0;
            sync_u_0 <= BTNU; sync_u_1 <= sync_u_0;
            sync_d_0 <= BTND; sync_d_1 <= sync_d_0;
            sync_l_0 <= BTNL; sync_l_1 <= sync_l_0;
            sync_r_0 <= BTNR; sync_r_1 <= sync_r_0;
            sw_sync_0 <= SW;    sw_sync_1 <= sw_sync_0;
        end
    end

    // 功能逻辑：音量增减、上一曲下一曲、直接选择
    always @(posedge CLK) begin
        if(!RST) begin
            vol <= 16'h4040;          // 默认音量，显示为 12
            CURRENT <= 3'd0;          // 默认歌曲 0
        end else begin
            // 中键复位到默认
            if(rise_c) begin
                vol <= 16'h4040;
                CURRENT <= 3'd0;
            end
            // 音量减
            if(rise_d) begin
                if(vol <= 16'h0000) vol <= 16'h0000;
```

```
                else vol <= vol - 16'h1010;
            end
            // 音量加
            if(rise_u) begin
                if(vol >= 16'hF0F0) vol <= 16'hF0F0;
                else vol <= vol + 16'h1010;
            end
            // 上一曲（循环）
            if(rise_l) begin
                CURRENT <= (CURRENT==0) ? 3'd7 : (CURRENT - 1);
            end
            // 下一曲（循环）
            if(rise_r) begin
                CURRENT <= (CURRENT==3'd7) ? 3'd0 : (CURRENT + 1);
            end
            // 滑动开关直接选择（优先）
            if(|sw_level_sync[7:0]) begin
                if(sw_level_sync[0]) CURRENT <= 3'd0;
                else if(sw_level_sync[1]) CURRENT <= 3'd1;
                else if(sw_level_sync[2]) CURRENT <= 3'd2;
                else if(sw_level_sync[3]) CURRENT <= 3'd3;
                else if(sw_level_sync[4]) CURRENT <= 3'd4;
                else if(sw_level_sync[5]) CURRENT <= 3'd5;
                else if(sw_level_sync[6]) CURRENT <= 3'd6;
                else if(sw_level_sync[7]) CURRENT <= 3'd7;
            end
        end
    end
endmodule
```

Display7.v：

```
module Display7(
    input CLK,
    input [15: 0] DATA,     // 时间（秒） mm:ss
    input [15: 0] VOL,      // 音量编码，例如 0x0000,0x1010..0xF0F0
    input [2:0]   CURRENT, // 当前歌曲索引 0..7
    output reg [6: 0] SEG,
    output reg [7: 0] SHIFT,
    output reg DOT
);
    wire scan_clock_divider;
    Divider #(.Time(200000)) u_scan_clock_div(CLK, scan_clock_divider);//分频
    initial SHIFT = 8'b01111111;
```

```
reg [31: 0] scan_digits; // [31:16] 左侧两位（歌曲/音量），[15:0] 右侧四位（时间）
integer volume_level;

reg [4: 0] digit_select_index;
always @ (posedge scan_clock_divider) begin
    SHIFT <= {SHIFT[6:0], SHIFT[7]};
    digit_select_index <= digit_select_index+4;// 在扫描中间两位时点亮小数点以显示:
    if(SHIFT[1]==0)
        DOT <= 0;
    else
        DOT <= 1;
    scan_digits[3: 0]   <= DATA % 10;                    // 秒 1
    scan_digits[7: 4]   <= (DATA / 10) % 6;             // 秒 10
    scan_digits[11: 8] <= (DATA / 60) % 10;            // 分 1
    scan_digits[15:12] <= (DATA / 600);                 // 分 10
    // 3-4 位 VOL 级别 1..16
    volume_level = 16-(VOL[15:12]);                      // 0x0..0xF   1..16
    scan_digits[23:20] <= (volume_level / 10);              // 音量十位
    scan_digits[19:16] <= (volume_level % 10);               // 音量个位
    // 左两位显示歌曲编号
    scan_digits[31:28] <= (CURRENT / 10);             // 歌曲十位
    scan_digits[27:24] <= (CURRENT % 10);              // 歌曲个位
    //下面直接用之前小作业的七段数码管模块
    case        ({scan_digits[digit_select_index+3],        scan_digits[digit_select_index+2],
scan_digits[digit_select_index+1], scan_digits[digit_select_index]})
        4'b0000: begin
            SEG<=7'b1000000;
        end
        4'b0001: begin
            SEG<=7'b1111001;
        end
        4'b0010: begin
            SEG<=7'b0100100;
        end
        4'b0011: begin
            SEG<=7'b0110000;
        end
        4'b0100: begin
            SEG<=7'b0011001;
        end
        4'b0101: begin
            SEG<=7'b0010010;
        end
        4'b0110: begin
```

```
                SEG<=7'b0000010;
            end
            4'b0111: begin
                SEG<=7'b1111000;
            end
            4'b1000: begin
                SEG<=7'b0000000;
            end
            4'b1001: begin
                SEG<=7'b0010000;
            end
            default: begin
                SEG<=7'b1111111;
            end
        endcase
    end
endmodule
```

Devider.v：

```
module Divider #(parameter Time=20)
(
    input I_CLK,
    output reg O_CLK
);
    integer div_count=0;
    initial O_CLK = 0;
    always @ (posedge I_CLK)
    begin
        if((div_count+1)==Time/2)
        begin
            div_count <= 0;
            O_CLK <= ~O_CLK;
        end
        else
            div_count <= div_count+1;
    end
endmodule
```

top_tb.v:

```
module top_tb;
    reg clk = 0; always #5 clk = ~clk; // 100MHz
    reg rst = 0;
```

```
    reg dreq = 1;

    reg btnc=0, btnu=0, btnd=0, btnl=0, btnr=0;
    reg [15:0] sw = 16'h0000;

    wire xdcs, xcs, rset, si, sclk;
    wire din, oled_clk, cs, dc, res;
    wire [6:0] seg; wire [7:0] shift; wire dot;
    wire [15:0] led;

    top dut(
        .CLK(clk), .RST(rst),
        .DREQ(dreq), .XDCS(xdcs), .XCS(xcs), .RSET(rset), .SI(si), .SCLK(sclk),
        .BTNC(btnc), .BTNU(btnu), .BTND(btnd), .BTNL(btnl), .BTNR(btnr),
        .SW(sw), .SEG(seg), .SHIFT(shift), .DOT(dot),
        .DIN(din), .OLED_CLK(oled_clk), .CS(cs), .DC(dc), .RES(res),
        .led(led)
    );

    initial begin
        rst = 0; #200; rst = 1;
        //
        #200000; btnu = 1; #100000; btnu = 0;
        #200000; btnr = 1; #100000; btnr = 0;
        //      SW w
        #300000; sw = 16'h0004;
        //          ~3ms
        #2300000; $finish;
    end
endmodule
```

MP3_tb.v：

```
module MP3_tb;
    // 100MHz  主时钟
    reg clk = 0;
    always #5 clk = ~clk;

    reg rst = 0;
    reg dreq = 1;
    reg [15:0] vol = 16'h0808; //  中等音量
    reg [2:0]    current = 3'd0;

    wire xdcs, xcs, rset, si, sclk, mp3_rst;
```

```
    wire [15:0] led;

    MP3 dut(
        .CLK(clk), .DREQ(dreq), .RST(rst), .vol(vol), .current(current),
        .XDCS(xdcs), .XCS(xcs), .RSET(rset), .SI(si), .SCLK(sclk), .MP3_RST(mp3_rst), .led(l
ed)
    );

    // 加快仿真：缩短上电延时
    defparam dut.DELAY_TIME = 1000; // 默认 500000 -> 1000

    // 统计命令/数据传输帧数（波形观测）
    integer cmd_bits = 0;
    integer cmd_words32 = 0;
    integer data_bits = 0;
    integer data_words16 = 0;

    always @(posedge sclk) begin
        if (xcs==0) begin
            cmd_bits <= cmd_bits + 1;
            if (cmd_bits==31) begin
                cmd_bits <= 0;
                cmd_words32 <= cmd_words32 + 1;
            end
        end
        if (xdcs==0) begin
            data_bits <= data_bits + 1;
            if (data_bits==15) begin
                data_bits <= 0;
                data_words16 <= data_words16 + 1;
            end
        end
    end
    initial begin
        rst = 0; #200; rst = 1;
        // 音量变化以触发音量写命令
        #200000; vol = 16'h1010;
        #200000; vol = 16'h2020;
        // 切歌
        #200000; current = 3'd1;
        // 总时长 ~2ms
        #1400000; $finish;
    end
endmodule
```

OLED_tb.v：

```
module OLED_tb;
    // 100MHz 主时钟
    reg clk = 0;
    always #5 clk = ~clk;
    reg rst = 0;
    reg [2:0] current = 3'd0;

    wire din, oled_clk, cs, dc, res;

    oled dut(
        .CLK(clk),
        .RST(rst),
        .current(current),
        .DIN(din),
        .OLED_CLK(oled_clk),
        .CS(cs),
        .DC(dc),
        .RES(res)
    );

    // SPI 统计初始化命令字节数与帧数据字节数
    reg [2:0] bit_cnt = 0;
    reg [7:0] byte_shift = 8'h00;
    integer init_cmd_bytes = 0;     // 期望达到 48
    integer frame_data_bytes = 0; // 期望快速增长
    reg init_done = 0;
    // 在像素时钟上采样，CS 为低时移入 DIN
    always @(posedge oled_clk) begin
        if (!cs) begin
            byte_shift <= {byte_shift[6:0], din};
            bit_cnt <= bit_cnt + 1;
            if (bit_cnt==3'd7) begin
                if (dc==1'b0) init_cmd_bytes <= init_cmd_bytes + 1;
                else          frame_data_bytes <= frame_data_bytes + 1;
            end
        end else begin
            bit_cnt <= 0; // 片选释放后重新计数
        end
    end
    // 初始化阶段完成标志
    always @(*) begin
        init_done = (init_cmd_bytes >= 48);
```

```
    end

    initial begin
        rst = 0; //  释放复位
        #200; // 200ns  保持复位
        rst = 1;
        //  中途切换一次图片
        #500000; // 0.5ms
        current = 3'd1;
        //  总仿真时间  ~2ms，足以完成初始化并进入数据阶段
        #1500000;
        $finish;
    end
endmodule
```

btn_control_tb.v：

```
module btn_control_tb;
    reg clk = 0; always #5 clk = ~clk; // 100MHz
    reg rst = 0;
    reg btnc = 0, btnu = 0, btnd = 0, btnl = 0, btnr = 0;
    reg [15:0] sw = 16'h0000;
    wire [15:0] vol;
    wire [2:0] CURRENT;
    btn_control dut(
        .CLK(clk), .RST(rst), .BTNC(btnc), .BTNU(btnu), .BTND(btnd), .BTNL(btnl), .BTNR(
btnr), .SW(sw),
        .vol(vol), .CURRENT(CURRENT)
    );

    initial begin
        rst = 0; #200; rst = 1;
        //  音量上/下
        btnu = 1; #100000; btnu = 0;
        #200000; btnd = 1; #100000; btnd = 0;
        //  上一曲/下一曲
        #200000; btnl = 1; #100000; btnl = 0;
        #200000; btnr = 1; #100000; btnr = 0;
        //  滑块强制选曲
        #200000; sw = 16'h0002;
        #1000000; $finish;
    end
endmodule
```

Devider_tb.v：

```
module Devider_tb;
    // 100MHz 时钟
    reg clk = 0;
    always #5 clk = ~clk;

    wire oclk;
    // 待测分频器，参数 Time=20 -> 2.5MHz 输出
    Divider #(.Time(20)) dut (
        .I_CLK(clk),
        .O_CLK(oclk)
    );

    // 计数：输入/输出沿计数，无任何打印
    integer in_edges = 0;
    integer out_edges = 0;
    reg pass_ratio = 0;

    always @(posedge clk) begin
        in_edges <= in_edges + 1;
        // 每经过一段观察窗口，检查比值是否接近 Time/2
        if (in_edges==2000) begin
            // 期望 out_edges ≈ in_edges/(Time/2) = 2000/10 = 200
            if (out_edges==200)
                pass_ratio <= 1; // 在波形中观察该标志即可
        end
    end

    always @(posedge oclk) begin
        out_edges <= out_edges + 1;
    end

    initial begin
        // 2us 仿真窗口，足以完成一次统计
        #2000;
        $finish;
    end
endmodule
```

Display7_tb.v：

```
module Display7_tb;
```

```
    reg clk = 0;
    always #5 clk = ~clk; // 100MHz

    reg [15:0] data_s = 16'd0;
    reg [15:0] vol_code = 16'h0808;
    reg [2:0]    song_idx = 3'd0;

    wire [6:0] seg;
    wire [7:0] shift;
    wire dot;

    Display7 dut(
        .CLK(clk), .DATA(data_s), .VOL(vol_code), .CURRENT(song_idx),
        .SEG(seg), .SHIFT(shift), .DOT(dot)
    );

    // 简单扫变化，观察波形
    initial begin
        repeat (1200) begin
            #83333; // ~12kHz 刷新窗口下的一段时间
            data_s <= data_s + 1;
        end
        vol_code <= 16'h1010;
        #1000000; song_idx <= 3'd5;
        #1000000; $finish;
    end
endmodule
```

合成歌曲 COE 的 C++程序：

```
#include <algorithm>
#include <cctype>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <regex>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

// 将字符串转为大写（小写字母转为大写）
```

```
static string to_upper(const string &s){
    string out = s;
    for(size_t i=0;i<out.size();++i){
        char c = out[i];
        if(c >= 'a' && c <= 'z') out[i] = char(c - 'a' + 'A');
    }
    return out;
}

// 判断是否为 COE 文件（根据关键字）
static bool is_coe_file(const string &text_lower){
    return text_lower.find("memory_initialization_vector") != string::npos;
}

// 从输入流解析 COE 数据区，返回按 16-bit（4 hex）表示的字符串向量（HI then LO）
static vector<string> parse_coe_16words(istream &in){
    // 读取全部内容
    ostringstream oss; oss << in.rdbuf();
    string content = oss.str();

    // 小写副本用于查找关键字
    string low = content;
    for(size_t i=0;i<low.size();++i) low[i] = (char)tolower((unsigned char)low[i]);
    size_t pos = low.find("memory_initialization_vector");
    if(pos == string::npos) throw runtime_error("Not a COE file (missing memory_initialization_vector)");

    // 找到 '=' 后到第一个 ';' 为数据区域
    size_t eq = low.find('=', pos);
    size_t semi = low.find(';', pos);
    size_t start = (eq != string::npos) ? (eq + 1) : pos;
    size_t end = (semi != string::npos) ? semi : content.size();
    string region = content.substr(start, end - start);

    // 手动提取十六进制 token（1..8 hex）
    vector<string> tokens;
    size_t i = 0;
    while(i < region.size()){
        // 跳过非 hex 字符
        while(i < region.size() && !isxdigit((unsigned char)region[i])) ++i;
        if(i >= region.size()) break;
        size_t j = i;
        while(j < region.size() && isxdigit((unsigned char)region[j]) && (j - i) < 16) ++j; // 限制长度
```

```
            string tok = region.substr(i, j - i);
            if(!tok.empty()) tokens.push_back(tok);
            i = j;
        }
        if(tokens.empty()) throw runtime_error("No hex tokens in COE data region");

        // 转换为 16-bit 字符串（HI then LO）
        vector<string> out16; out16.reserve(tokens.size()*2);
        for(string t : tokens){
            t = to_upper(t);
            if(t.size() > 8) t = t.substr(t.size()-8);
            // 将 hex 转为数值（使用 strtoul）
            unsigned long v = strtoul(t.c_str(), nullptr, 16);
            if(t.size() <= 4){
                char buf[5]; snprintf(buf, sizeof(buf), "%04lX", v & 0xFFFFul);
                out16.emplace_back(buf);
            } else {
                unsigned long hi = (v >> 16) & 0xFFFFul;
                unsigned long lo = v & 0xFFFFul;
                char bh[5], bl[5];
                snprintf(bh, sizeof(bh), "%04lX", hi);
                snprintf(bl, sizeof(bl), "%04lX", lo);
                out16.emplace_back(bh);
                out16.emplace_back(bl);
            }
        }
        return out16;
}

// 将 16-bit 字符串 vector 写入 COE（每行 items_per_line）
static void write_coe16(const vector<string> &words, const string &out_path, int items_per_line){
        ofstream fout(out_path);
        if(!fout) throw runtime_error("无法打开输出文件: " + out_path);
        fout << "memory_initialization_radix=16;\n";
        fout << "memory_initialization_vector=\n";
        for(size_t i=0;i<words.size();++i){
            bool last = (i+1==words.size());
            fout << words[i] << (last?';':',');
            if(items_per_line>0 && ((i+1)%items_per_line)==0) fout << '\n';
        }
        fout << '\n';
}
```

```
int main(int argc, char** argv){
    if(argc < 4){
        cerr << "Usage: merge_coe_songs <out.coe> <per_song_words> <song0.coe> [song1.coe ...] [--items-per-line N]\n";
        cerr << "Notes: Inputs can be 16-bit COE or 32-bit COE; output is 16-bit COE.\n";
        return 1;
    }
    string out_path = argv[1];
    int per_song = atoi(argv[2]);
    if(per_song <= 0) { cerr << "Invalid per_song_words\n"; return 2; }
    int items_per_line = 16;

    vector<string> inputs;
    for(int i=3;i<argc;++i){
        string a = argv[i];
        if(a == string("--items-per-line")){
            if(i+1<argc){ items_per_line = max(1, atoi(argv[++i])); }
        } else {
            inputs.push_back(a);
        }
    }
    if(inputs.empty()){ cerr << "No input songs provided\n"; return 3; }

    vector<string> merged; merged.reserve((size_t)per_song * inputs.size());

    for(const auto &in_path : inputs){
        ifstream fin(in_path, ios::binary);
        if(!fin){ cerr << "无法打开输入文件: " << in_path << "\n"; return 4; }
        try {
            vector<string> words16 = parse_coe_16words(fin);
            if((int)words16.size() > per_song){
                words16.resize(per_song);
            } else if((int)words16.size() < per_song){
                words16.insert(words16.end(), per_song - (int)words16.size(), string("0000"));
            }
            merged.insert(merged.end(), words16.begin(), words16.end());
            cout << "添加: " << in_path << ": " << per_song << " words" << endl;
        } catch(const exception &e){
            cerr << "处理失败: " << in_path << " -> " << e.what() << endl;
            return 5;
        }
    }
```

```
    try{ write_coe16(merged, out_path, items_per_line); }
    catch(const exception &e){ cerr << e.what() << endl; return 6; }

    cout << "写入总计 " << merged.size() << " words 到 " << out_path << endl;
    return 0;
}
```

合并 BMP 为 COE 的 C++程序：

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
using namespace std;

#pragma pack(push,1)
struct BmpFileHeader {
    uint16_t bfType;          // 'BM' = 0x4D42
    uint32_t bfSize;
    uint16_t bfReserved1;
    uint16_t bfReserved2;
    uint32_t bfOffBits;     // offset to pixel array
};
struct BmpInfoHeader {       // BITMAPINFOHEADER (size >= 40)
    uint32_t biSize;          // header size
    int32_t   biWidth;
    int32_t   biHeight;       // positive: bottom-up; negative: top-down
    uint16_t biPlanes;
    uint16_t biBitCount;    // 1/4/8/24/32
    uint32_t biCompression;// 0 = BI_RGB (no compression)
    uint32_t biSizeImage;    // may be 0 for BI_RGB
    int32_t   biXPelsPerMeter;
    int32_t   biYPelsPerMeter;
    uint32_t biClrUsed;     // palette entries used
    uint32_t biClrImportant;
};
#pragma pack(pop)

// RGB24 -> RGB565 转换
static inline uint16_t rgb24_to_rgb565(uint8_t r, uint8_t g, uint8_t b) {
    uint16_t R = (uint16_t)(r >> 3);
```

```
    uint16_t G = (uint16_t)(g >> 2);
    uint16_t B = (uint16_t)(b >> 3);
    return (uint16_t)((R << 11) | (G << 5) | B);
}

// 简单的图像容器（RGBA）
struct ImageRGB {
    int w = 0, h = 0;
    vector<uint8_t> rgba; // 存放解码后的像素，按 RGBA
};

// 从 BMP 文件读取并解码为 RGBA（支持常见位深）
static ImageRGB load_bmp_any(const string& path) {
    ifstream fin(path, ios::binary);
    if (!fin) throw runtime_error("无法打开 BMP: " + path);
    BmpFileHeader fh{}; BmpInfoHeader ih{};
    fin.read(reinterpret_cast<char*>(&fh), sizeof(fh));
    fin.read(reinterpret_cast<char*>(&ih), sizeof(ih));
    if (!fin) throw runtime_error("读取 BMP 头失败: " + path);
    if (fh.bfType != 0x4D42) throw runtime_error("不是 BMP 文件: " + path);
    if (ih.biPlanes != 1) throw runtime_error("不支持 biPlanes != 1");
    if (!(ih.biBitCount == 1 || ih.biBitCount == 4 || ih.biBitCount == 8 || ih.biBitCount == 24 ||
ih.biBitCount == 32))
        throw runtime_error("不支持的位深: " + to_string(ih.biBitCount));
    if (!(ih.biCompression == 0 || (ih.biCompression == 3 && ih.biBitCount == 32)))
        throw runtime_error("不支持的压缩方式: " + to_string(ih.biCompression));
    const int srcW = ih.biWidth;
    const int srcH = abs(ih.biHeight);
    const bool bottomUp = (ih.biHeight > 0);

    // 调色板（indexed）
    uint32_t paletteEntries = 0;
    if (ih.biBitCount <= 8) {
        paletteEntries = ih.biClrUsed ? ih.biClrUsed : (1u << ih.biBitCount);
    }
    vector<uint8_t> palette;
    if (paletteEntries) {
        palette.resize((size_t)paletteEntries * 4u);
        fin.read(reinterpret_cast<char*>(palette.data()), (streamsize)palette.size());
        if (!fin) throw runtime_error("读取调色板失败: " + path);
    }

    // 读取像素数据
    fin.seekg(fh.bfOffBits, ios::beg);
```

```
    size_t rowStride = 0;
    if (ih.biBitCount == 32) rowStride = (size_t)srcW * 4u;
    else if (ih.biBitCount == 24) rowStride = (((size_t)srcW * 3u) + 3u) & ~3u;
    else if (ih.biBitCount == 8)    rowStride = (((size_t)srcW * 1u) + 3u) & ~3u;
    else if (ih.biBitCount == 4)    rowStride = ((((size_t)srcW + 1u) / 2u) + 3u) & ~3u;
    else if (ih.biBitCount == 1)    rowStride = ((((size_t)srcW + 7u) / 8u) + 3u) & ~3u;
    vector<uint8_t> pixels; pixels.resize((size_t)srcH * rowStride);
    fin.read(reinterpret_cast<char*>(pixels.data()), (streamsize)pixels.size());
    if (!fin) throw runtime_error("读取像素数据失败: " + path);

    // 解码为 RGBA
    ImageRGB img; img.w = srcW; img.h = srcH; img.rgba.resize((size_t)srcW * (size_t)srcH *
4u);
    for (int y = 0; y < srcH; ++y) {
        int fileRow = bottomUp ? (srcH - 1 - y) : y;
        const uint8_t* rowPtr = pixels.data() + (size_t)fileRow * rowStride;
        for (int x = 0; x < srcW; ++x) {
            uint8_t r = 0, g = 0, b = 0;
            if (ih.biBitCount == 32) {
                const uint8_t* px = rowPtr + (size_t)x * 4u; // BGRA
                b = px[0]; g = px[1]; r = px[2];
            }
            else if (ih.biBitCount == 24) {
                const uint8_t* px = rowPtr + (size_t)x * 3u; // BGR
                b = px[0]; g = px[1]; r = px[2];
            }
            else if (ih.biBitCount == 8) {
                uint8_t idx = rowPtr[x];
                if (palette.empty()) { b = g = r = idx; }
                else {
                    const uint8_t* pe = &palette[(size_t)idx * 4u];
                    b = pe[0]; g = pe[1]; r = pe[2];
                }
            }
            else if (ih.biBitCount == 4) {
                uint8_t byte = rowPtr[x / 2];
                uint8_t idx = (x % 2 == 0) ? (byte >> 4) : (byte & 0x0F);
                if (palette.empty()) { b = g = r = idx * 17; }
                else {
                    const uint8_t* pe = &palette[(size_t)idx * 4u];
                    b = pe[0]; g = pe[1]; r = pe[2];
                }
            }
            else if (ih.biBitCount == 1) {
```

```
                    uint8_t byte = rowPtr[x / 8];
                    uint8_t bit = 7 - (x % 8);
                    uint8_t idx = (byte >> bit) & 0x01u;
                    if (palette.empty()) {
                        b = g = r = idx ? 255 : 0;
                    }
                    else {
                        const uint8_t* pe = &palette[(size_t)idx * 4u];
                        b = pe[0]; g = pe[1]; r = pe[2];
                    }
                }
                size_t outOff = ((size_t)y * (size_t)srcW + (size_t)x) * 4u;
                img.rgba[outOff + 0] = r;
                img.rgba[outOff + 1] = g;
                img.rgba[outOff + 2] = b;
                img.rgba[outOff + 3] = 255;
            }
        }
        return img;
}

// 将一行像素按输出分辨率缩放（最近邻）并返回每像素 RGB565 的十六进制字节串
static string make_row_hex_rgb565(const ImageRGB& img, int outW, int outH, int rowY) {
        string hexRow; hexRow.reserve((size_t)outW * 4);
        int srcH = img.h, srcW = img.w;
        int srcY_nn = (int)((int64_t)rowY * srcH / outH);
        if (srcY_nn < 0) srcY_nn = 0; if (srcY_nn >= srcH) srcY_nn = srcH - 1;
        for (int x = 0; x < outW; ++x) {
            int srcX_nn = (int)((int64_t)x * srcW / outW);
            if (srcX_nn < 0) srcX_nn = 0; if (srcX_nn >= srcW) srcX_nn = srcW - 1;
            size_t off = ((size_t)srcY_nn * (size_t)srcW + (size_t)srcX_nn) * 4u;
            uint8_t r = img.rgba[off + 0];
            uint8_t g = img.rgba[off + 1];
            uint8_t b = img.rgba[off + 2];
            uint16_t c = rgb24_to_rgb565(r, g, b);
            uint8_t hi = (uint8_t)((c >> 8) & 0xFFu);
            uint8_t lo = (uint8_t)(c & 0xFFu);
            char buf[3];
            snprintf(buf, sizeof(buf), "%02X", hi);
            hexRow.append(buf);
            snprintf(buf, sizeof(buf), "%02X", lo);
            hexRow.append(buf);
        }
        return hexRow;
```

```
}

// 将所有行写入 COE 文件
static void write_coe_rows(const vector<string>& rowHexTokens, const string& out_path) {
    ofstream fout(out_path);
    if (!fout) throw runtime_error("无法打开输出文件: " + out_path);
    fout << "memory_initialization_radix=16;\n";
    fout << "memory_initialization_vector=\n";
    for (size_t i = 0; i < rowHexTokens.size(); ++i) {
        bool last = (i + 1 == rowHexTokens.size());
        fout << rowHexTokens[i] << (last ? ';' : ',') << '\n';
    }
}

int main(int argc, char** argv) {
    if (argc < 3) {
        cerr << "Usage: BMPsToCOE <output.coe> <img1.bmp> [img2.bmp ...] [--width-px W=96] [--height H=64]" << endl;
        cerr << "Notes:\n  - Supports 1/4/8/24/32bpp uncompressed BMP (BI_RGB).\n  - Generates COE with one 1536-bit item per row (96px * 2 bytes RGB565).\n  - Multiple BMPs are concatenated as pages: page0 rows 0..63, then page1 rows 0..63, etc." << endl;
        return 1;
    }
    string out_path = argv[1];
    int outW = 96, outH = 64;
    vector<string> inputs;
    for (int i = 2; i < argc; ++i) {
        string a = argv[i];
        if (a == string("--width-px") && i + 1 < argc) { outW = atoi(argv[++i]); }
        else if (a == string("--height") && i + 1 < argc) { outH = atoi(argv[++i]); }
        else { inputs.push_back(a); }
    }
    if (inputs.empty()) { cerr << "未提供任何 BMP 输入\n"; return 2; }
    if (outW <= 0 || outH <= 0) { cerr << "无效的输出尺寸\n"; return 3; }

    vector<string> allRows; allRows.reserve((size_t)inputs.size() * (size_t)outH);
    size_t pagesAdded = 0;
    for (size_t idx = 0; idx < inputs.size(); ++idx) {
        const string& path = inputs[idx];
        try {
            ImageRGB img = load_bmp_any(path);
            for (int y = 0; y < outH; ++y) {
                string row = make_row_hex_rgb565(img, outW, outH, y);
                allRows.emplace_back(move(row));
```

```
                }
                ++pagesAdded;
                cout << "添加页面: " << path << " (" << img.w << "x" << img.h << ") -> " <<
outW << "x" << outH << endl;
            }
            catch (const exception& e) {
                cerr << "跳过 '" << path << "': " << e.what() << endl;
                continue;
            }
        }

        try { write_coe_rows(allRows, out_path); }
        catch (const exception& e) { cerr << e.what() << endl; return 4; }

        cout << "写入 COE 行数: " << allRows.size() << " (pages=" << pagesAdded << ", height="
<< outH << ")\n 输出: " << out_path << endl;
        return 0;
}
```

top.xdc:

```
# 第 1 部分---LD 系列，灯
set_property IOSTANDARD LVCMOS33 [get_ports {led[15]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[14]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {led[0]}]
set_property PACKAGE_PIN V11 [get_ports {led[15]}]
set_property PACKAGE_PIN V12 [get_ports {led[14]}]
set_property PACKAGE_PIN V14 [get_ports {led[13]}]
set_property PACKAGE_PIN V15 [get_ports {led[12]}]
set_property PACKAGE_PIN T16 [get_ports {led[11]}]
set_property PACKAGE_PIN U14 [get_ports {led[10]}]
```

```
set_property PACKAGE_PIN T15 [get_ports {led[9]}]
set_property PACKAGE_PIN V16 [get_ports {led[8]}]
set_property PACKAGE_PIN U16 [get_ports {led[7]}]
set_property PACKAGE_PIN U17 [get_ports {led[6]}]
set_property PACKAGE_PIN V17 [get_ports {led[5]}]
set_property PACKAGE_PIN R18 [get_ports {led[4]}]
set_property PACKAGE_PIN N14 [get_ports {led[3]}]
set_property PACKAGE_PIN J13 [get_ports {led[2]}]
set_property PACKAGE_PIN K15 [get_ports {led[1]}]
set_property PACKAGE_PIN H17 [get_ports {led[0]}]
# 第 2 部分---SW 系列，开关；SW[14]留作复位灯
set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[8]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[9]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[10]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[11]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[12]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[13]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[15]}]
set_property PACKAGE_PIN J15 [get_ports {SW[0]}]
set_property PACKAGE_PIN L16 [get_ports {SW[1]}]
set_property PACKAGE_PIN M13 [get_ports {SW[2]}]
set_property PACKAGE_PIN R15 [get_ports {SW[3]}]
set_property PACKAGE_PIN R17 [get_ports {SW[4]}]
set_property PACKAGE_PIN T18 [get_ports {SW[5]}]
set_property PACKAGE_PIN U18 [get_ports {SW[6]}]
set_property PACKAGE_PIN R13 [get_ports {SW[7]}]
set_property PACKAGE_PIN T8   [get_ports {SW[8]}]
set_property PACKAGE_PIN U8   [get_ports {SW[9]}]
set_property PACKAGE_PIN R16 [get_ports {SW[10]}]
set_property PACKAGE_PIN T13 [get_ports {SW[11]}]
set_property PACKAGE_PIN H6   [get_ports {SW[12]}]
set_property PACKAGE_PIN U12 [get_ports {SW[13]}]
set_property PACKAGE_PIN V10 [get_ports {SW[15]}]
# 第 3 部分---按钮系列，五个 BT 按键
set_property IOSTANDARD LVCMOS33 [get_ports BTNC]
set_property IOSTANDARD LVCMOS33 [get_ports BTNU]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports BTND]
set_property IOSTANDARD LVCMOS33 [get_ports BTNL]
set_property IOSTANDARD LVCMOS33 [get_ports BTNR]
set_property PACKAGE_PIN N17 [get_ports BTNC]
set_property PACKAGE_PIN M18 [get_ports BTNU]
set_property PACKAGE_PIN P18 [get_ports BTND]
set_property PACKAGE_PIN P17 [get_ports BTNL]
set_property PACKAGE_PIN M17 [get_ports BTNR]
# 第 4 部分---七段数码管，包括分和秒之间的小数点
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SHIFT[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEG[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports DOT]
set_property PACKAGE_PIN U13 [get_ports {SHIFT[7]}]
set_property PACKAGE_PIN K2  [get_ports {SHIFT[6]}]
set_property PACKAGE_PIN T14 [get_ports {SHIFT[5]}]
set_property PACKAGE_PIN P14 [get_ports {SHIFT[4]}]
set_property PACKAGE_PIN J14 [get_ports {SHIFT[3]}]
set_property PACKAGE_PIN T9  [get_ports {SHIFT[2]}]
set_property PACKAGE_PIN J18 [get_ports {SHIFT[1]}]
set_property PACKAGE_PIN J17 [get_ports {SHIFT[0]}]
set_property PACKAGE_PIN L18 [get_ports {SEG[6]}]
set_property PACKAGE_PIN T11 [get_ports {SEG[5]}]
set_property PACKAGE_PIN P15 [get_ports {SEG[4]}]
set_property PACKAGE_PIN K13 [get_ports {SEG[3]}]
set_property PACKAGE_PIN K16 [get_ports {SEG[2]}]
set_property PACKAGE_PIN R10 [get_ports {SEG[1]}]
set_property PACKAGE_PIN T10 [get_ports {SEG[0]}]
set_property PACKAGE_PIN H15 [get_ports DOT]

# 下面是 2 个外设
# 第 5 部分---MP3 系列，其他的用 JC PMOD；RST 顶层引脚绑定到 SW14(U11)
```

```
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property PACKAGE_PIN E3 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports DREQ]
set_property IOSTANDARD LVCMOS33 [get_ports RSET]
set_property IOSTANDARD LVCMOS33 [get_ports RST]
set_property IOSTANDARD LVCMOS33 [get_ports SCLK]
set_property IOSTANDARD LVCMOS33 [get_ports SI]
set_property IOSTANDARD LVCMOS33 [get_ports XCS]
set_property IOSTANDARD LVCMOS33 [get_ports XDCS]
set_property PACKAGE_PIN U11 [get_ports RST]
set_property PACKAGE_PIN F6 [get_ports RSET]
set_property PACKAGE_PIN J3 [get_ports SCLK]
set_property PACKAGE_PIN J4 [get_ports SI]
set_property PACKAGE_PIN E7 [get_ports XCS]
set_property PACKAGE_PIN J2 [get_ports DREQ]
set_property PACKAGE_PIN K1 [get_ports XDCS]
# 第 6 部分---OLED 系列，用 JA PMOD
set_property IOSTANDARD LVCMOS33 [get_ports CS]
set_property IOSTANDARD LVCMOS33 [get_ports DC]
set_property IOSTANDARD LVCMOS33 [get_ports DIN]
set_property IOSTANDARD LVCMOS33 [get_ports OLED_CLK]
set_property IOSTANDARD LVCMOS33 [get_ports RES]
set_property PACKAGE_PIN D18 [get_ports CS]
set_property PACKAGE_PIN C17 [get_ports DC]
set_property PACKAGE_PIN G17 [get_ports DIN]
set_property PACKAGE_PIN E18 [get_ports OLED_CLK]
set_property PACKAGE_PIN D17 [get_ports RES]
```