同济大学计算机系

数字逻辑课程综合实验报告

学　　号　2250748

姓　　名　王渝鹞

专　　业　信息安全

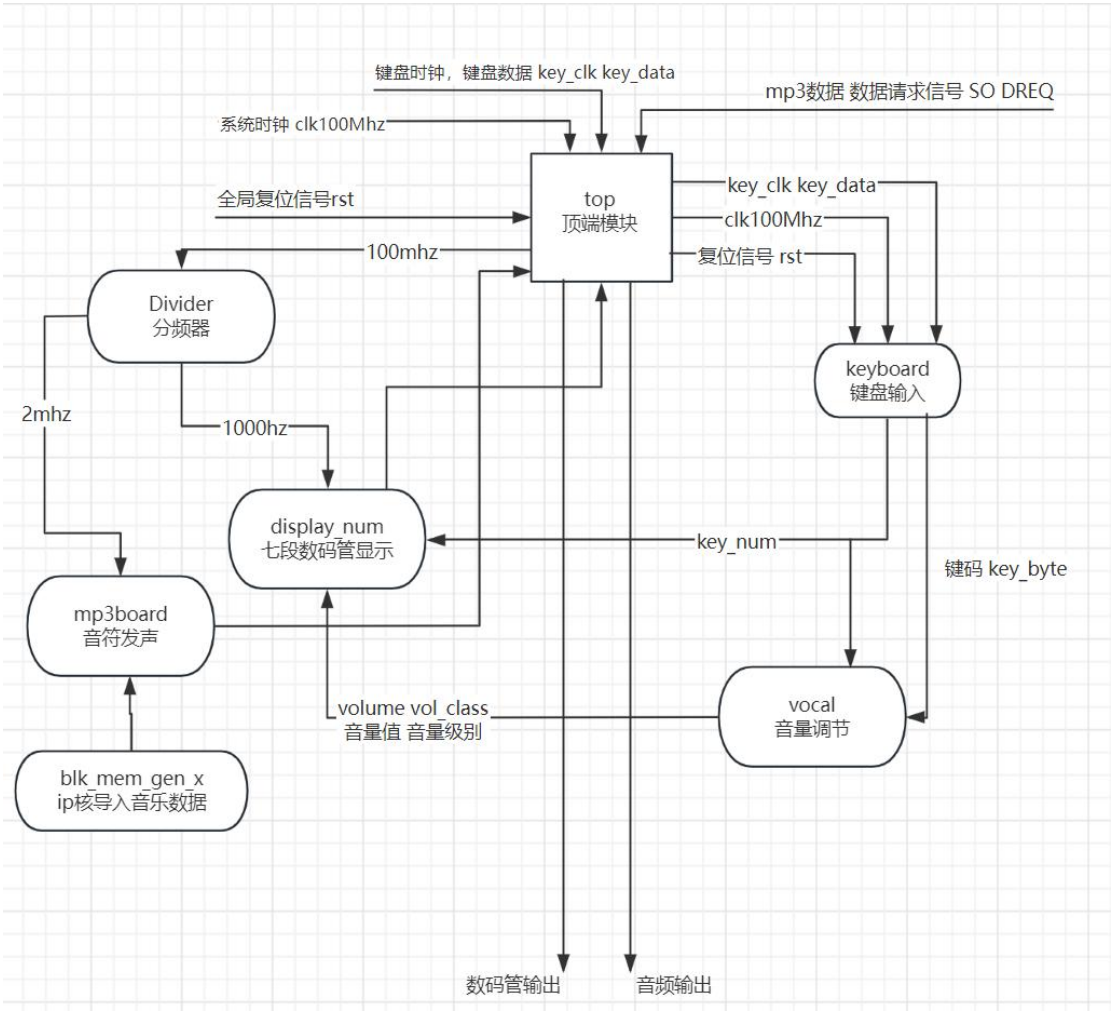
授课老师　张冬冬

一、实验内容

本项目为基于 MP3（VS1003B）、PS2 键盘、NEXYS-4 的电子琴数字系统。根据键盘按下不同的按键，由 MP3 播放事先存储在 IP 核中的不同音（允许调节音量），板上七段数码管显示当前按下音，模拟实现简易的三个八度电子琴。

生成 bit 流下板后，连接键盘、MP3（接耳机或音响），将 J15 置 1，便可以自行演奏。键盘的 1-7、Q-U、A-J 分别对应低、中、高三个八度的 do、re、mi、fa、so、la、xi。数码管的 1、2 两位显示当前按下的按键音（抬起或按下无效键则显示 00），数码管的第 8 位显示当前音量（总共有 0-9）10 个等级，音量可通过键盘的↑↓来调节。

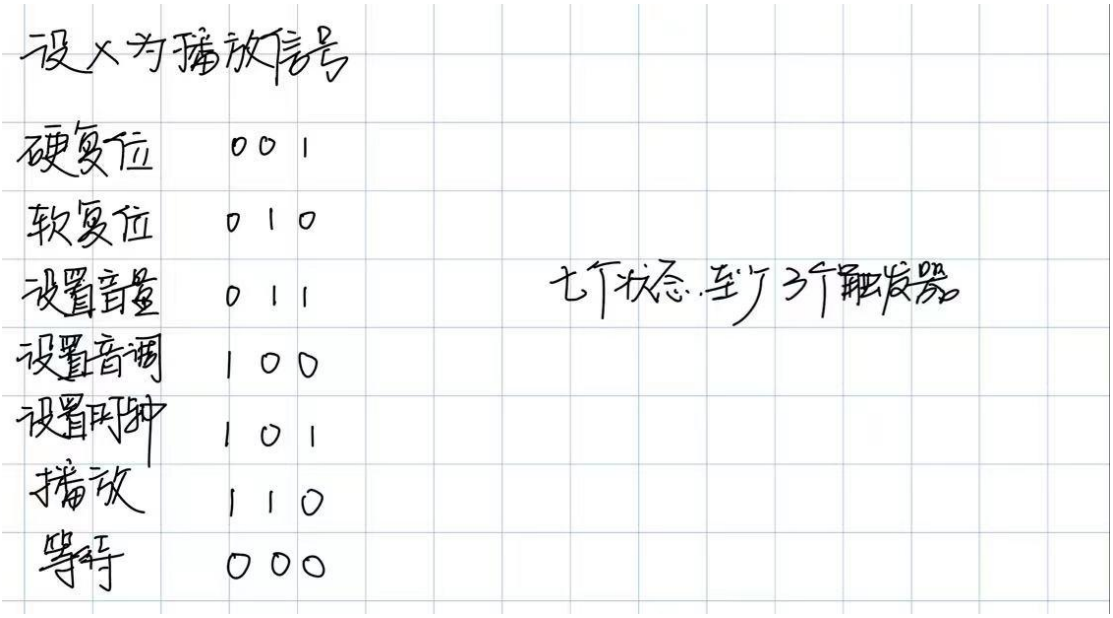
二、电子琴数字系统总框图



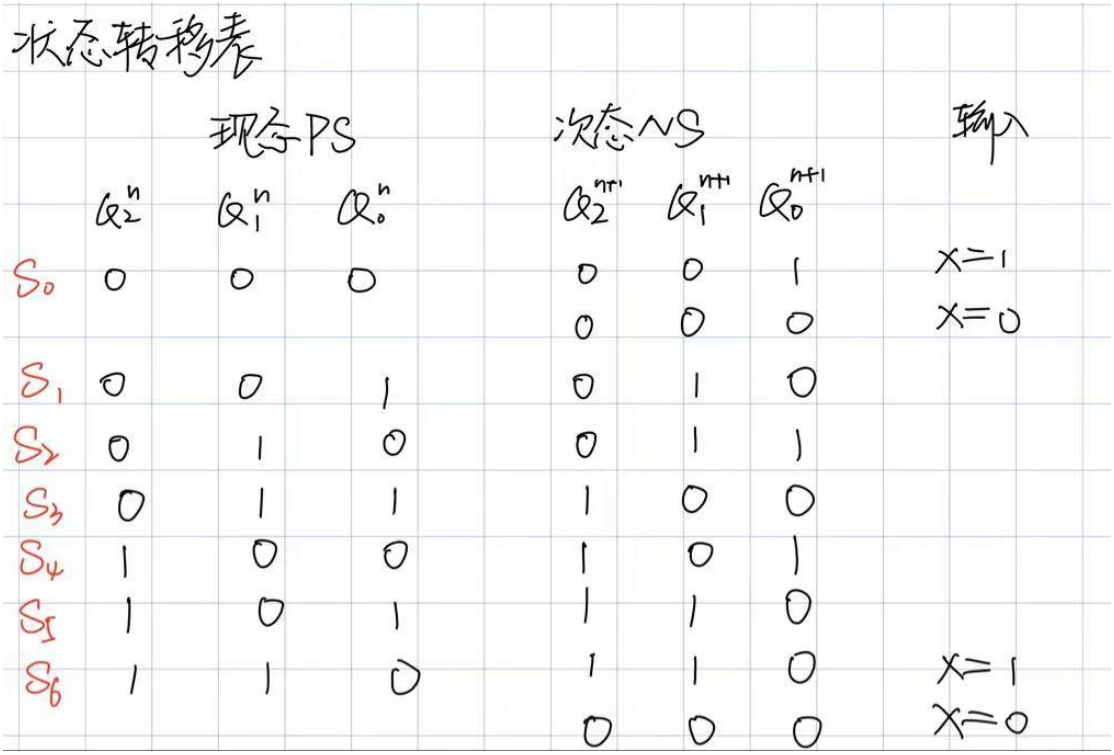
三、系统控制器设计

本系统的输入较多，连接两个外设，但总体流程可简化为收到播放信号，才开始执行系统的主要功能，核心状态图在 MP3 中。“播放信号”有两个——全局复位信号 Y 和播放信号 X。当键盘按下指定键时 X 才会激活。

状态分析如下图所示



可写状态转移表



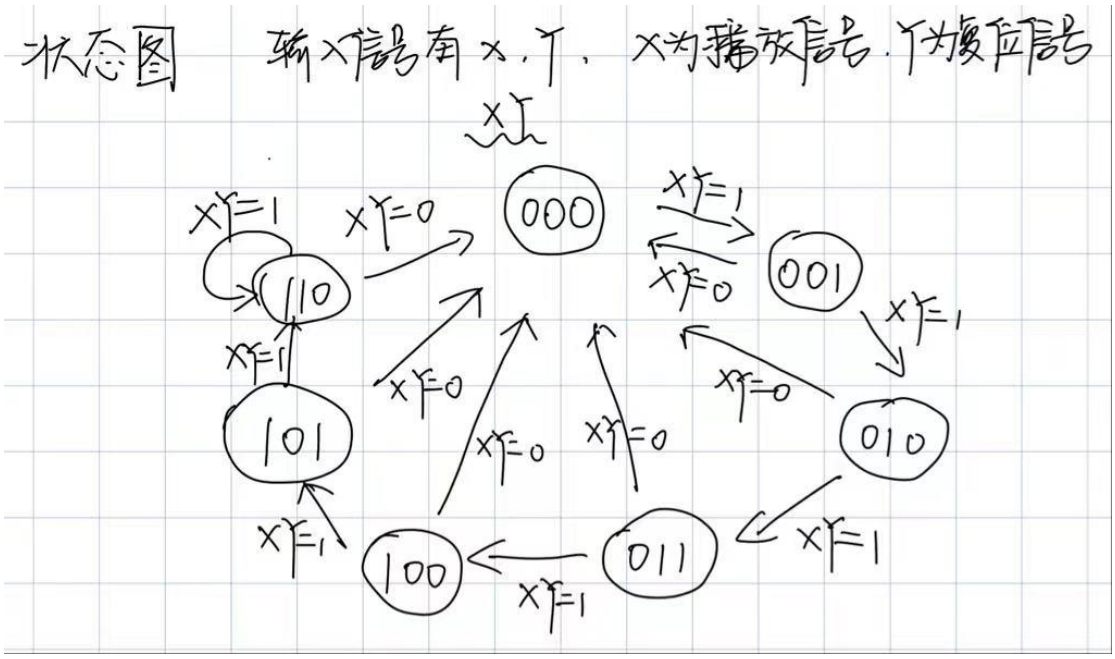
控制命令逻辑表达式

$$Q_2^{n+1} = Q_1 Q_0 \overline{Q_2}^n + (\overline{Q_1}^n + Q_1^n \overline{Q_0}^n X) Q_2^n$$

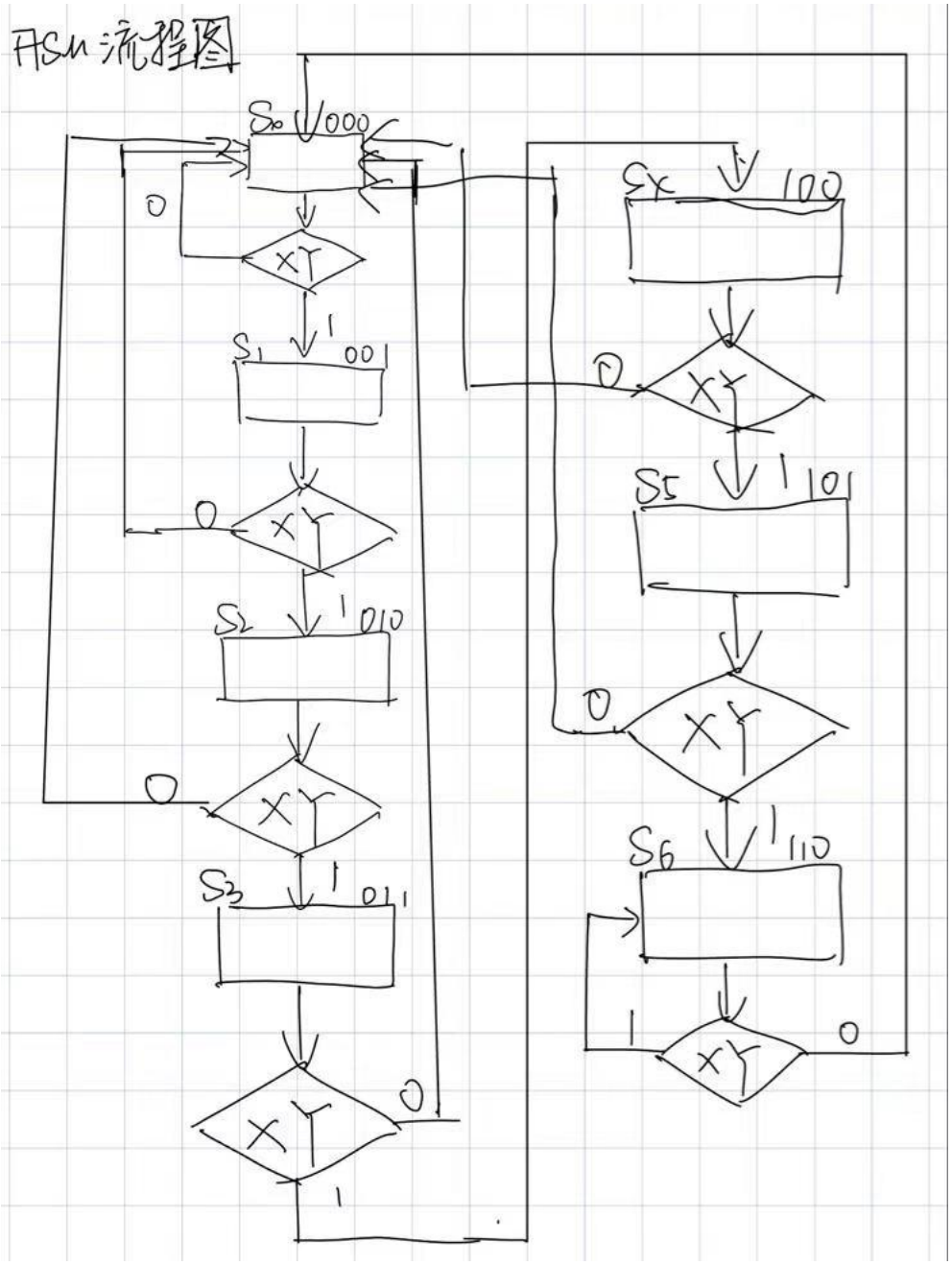
$$Q_1^{n+1} = (\overline{Q_2}^n \overline{Q_0}^n + \overline{Q_0}^n X) Q_1^n$$

$$Q_0^{n+1} = (\overline{Q_2}^n + X + \overline{Q_2}^n Q_1^n + Q_2^n \overline{Q_1}^n) \overline{Q_0}^n$$

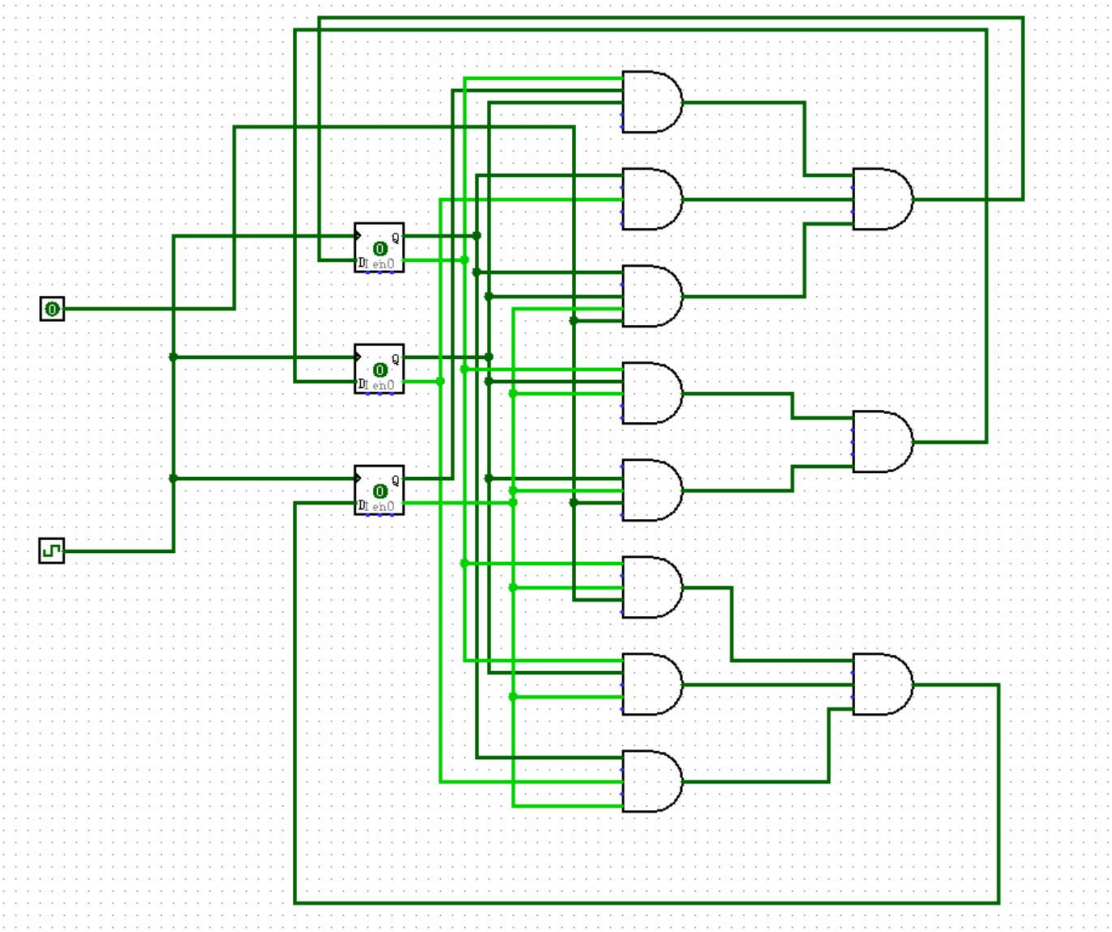
状态图



ASM 流程图

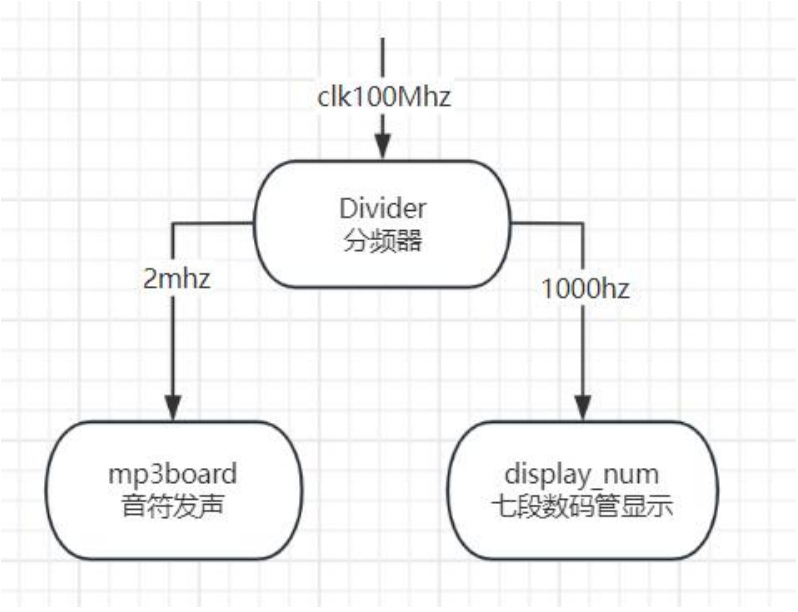

ASM流程图
S0 000
0
XY
1
S1 001
0
XY
S2 1 010
0
XY
S3 1 011
0
XY
1
Sx 100
0
XY
S5 1 101
0
XY
S6 1 110
1
XY
0

Logisim



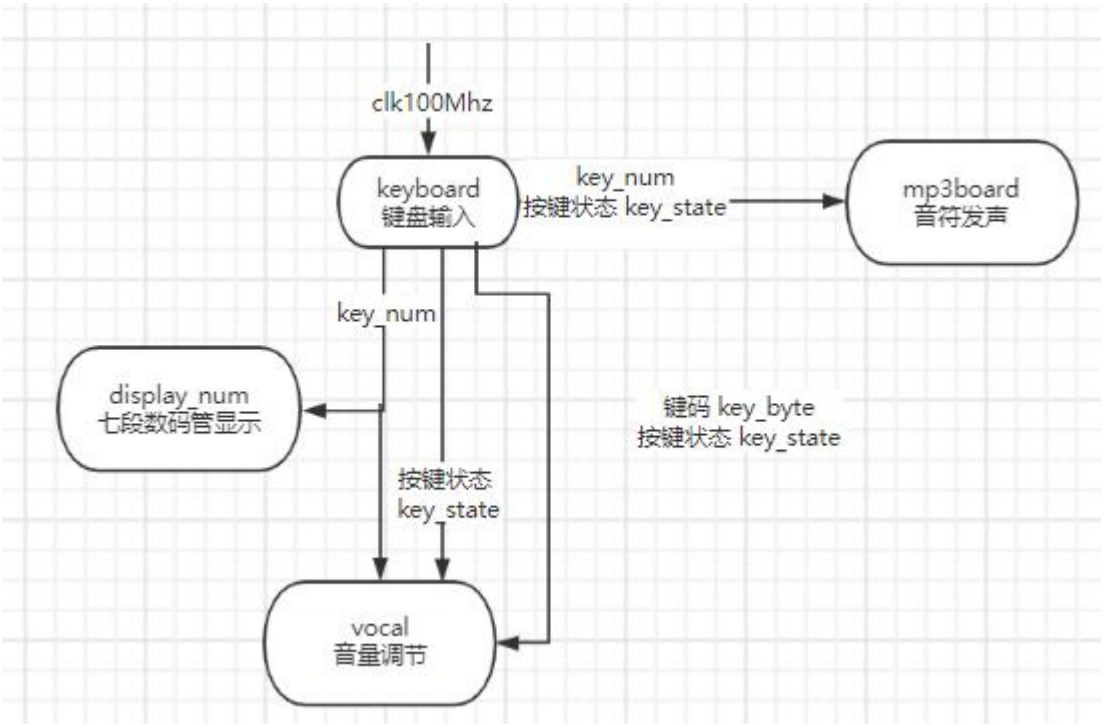
四、子系统模块建模

1、module divider



input clk100Mhz
output clk1000hz
output clk2Mhz
分频器，可以将输入的 100MHz 的时钟信号分频为 1000hz 和 2MHz，用于其他模块的时钟输入。
output clk1000hz：内部实现了一个计数器 cnt，它在每个 100MHz 时钟的上升沿增加。当计数器达到 50000 时（因为 100MHz 时钟每 10ns 跳动一次，所以 50000 次就是 50000 * 10ns = 500us，对应 1000Hz 的周期的一半），计数器重置为 0，并且 clk1000hz 信号翻转其状态（从 0 变为 1 或从 1 变为 0）。这样，clk1000hz 在每 500us 翻转一次，形成一个 1000Hz 的方波信号。
output clk2Mhz：使用一个单独的计数器 cnt2MHz，在每个 100MHz 时钟的上升沿增加。当计数器达到 25 时（100MHz 时钟每 10ns 跳动一次，所以 25 次就是 250ns，对应 2MHz 的周期的一半），计数器重置为 0，并且 clk2Mhz 信号翻转其状态。因此，clk2Mhz 在每 250ns 翻转一次，生成一个 2MHz 的方波信号。

2、module keyboard

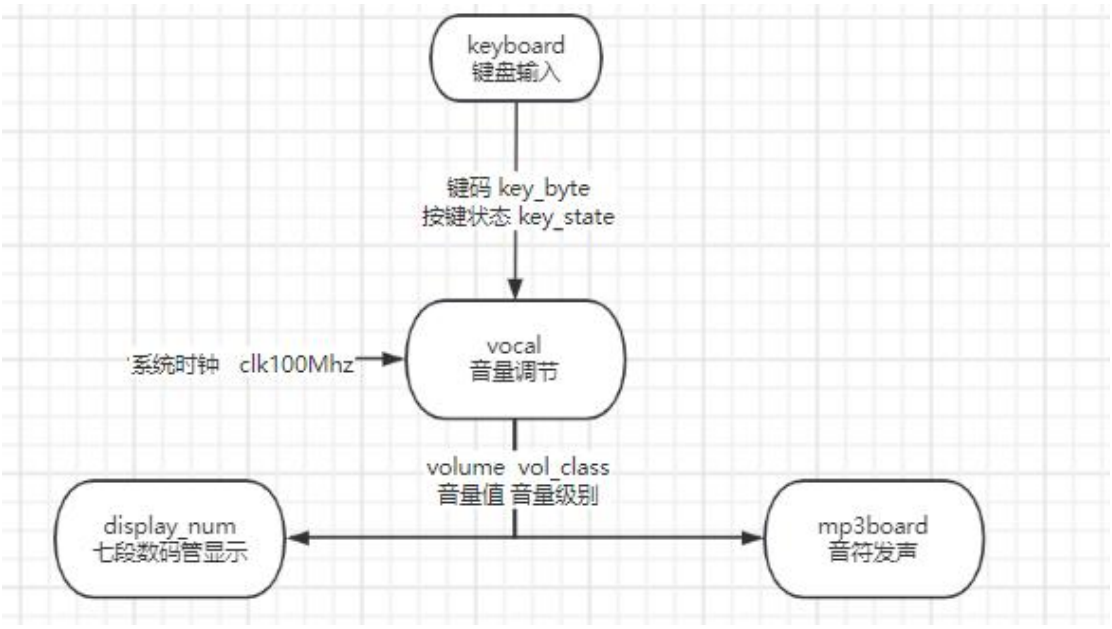


input	clk_in,	//系统时钟
input	rst,	//系统复位，低有效
input	key_clk,	//PS2 键盘时钟输入 f4
input	key_data,	//PS2 键盘数据输入 b2
output	key_state,	//键盘的按下状态，按下置 1，松开置 0
output	[7:0]key_num,	//按键键值对应的数字
output	[7:0] key_byte	// 原始键盘扫描码

接受 PS2 键盘输入的数据，将键盘的扫描码转换为自定义的数字键值，输出自定义的对应数字以及键盘的原始扫描码。

首先，通过一系列触发器锁存键盘时钟和数据信号，以检测键盘时钟信号的下降沿。当检测到下降沿时，模块会读取 8 位数据，这些数据是通过一个计数器在每个下降沿处序列地从键盘数据线捕获的。读取的数据首先作为原始扫描码输出（key_byte）。
此外，模块还能够识别键盘的断码（通常是 0xF0），这用于确定按键是被按下还是被释放。如果收到断码，模块会设置一个标志，并在下一个数据周期输出按键释放的状态；如果没有收到断码，则表示按键被按下，模块将输出相应的按键状态。
最后，模块包含一个解码逻辑，它将原始的扫描码转换成对应的数字键值，这个键值可以用于驱动其他模块，显示输出。

3、module vocal

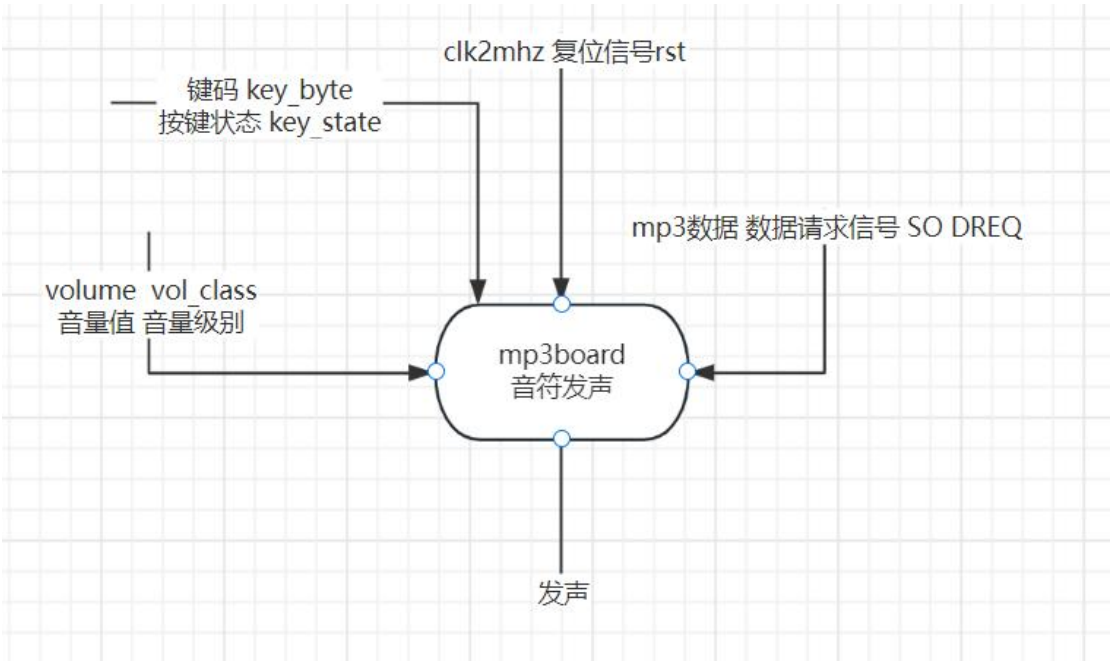


input clk,
input [7:0] key_num,　　// 键码
input key_state,　　// 键盘的按下状态，按下置 1，松开置 0
output [3:0] vol_class=9,//音量等级，最低为 0-最高为 9
output [15:0] vol=16'h0000//音量数值，用于后续 MP3 模块的具体赋值

控制电子琴音量。处理由键盘输入的键码，调节音量大小，同时输出音量等级和大小供后续模块的调整和显示输出。

主要实现音量调节和音量等级映射两个功能。
音量调节：模块利用输入的 key_num（键码）和 key_state（键盘状态）来判断是否需要调整音量。当用户按下与音量调节相关的键时（例如，键码 8'b01110101 用于音量增加，键码 8'b01110010 用于音量减少），模块内的 vol（音量控制值）将按照预定的步长 16'h197f 增加或减少。这一过程是通过一个内部计数器 clk_cnt 同步于 clk（时钟信号）控制的，确保音量调整在指定的时钟周期后发生。
音量级别映射：模块还将内部的音量数值 vol 映射到 vol_class（音量级别）输出，该输出是一个从 0 到 9 的数字，其中 0 代表最大音量，而 9 代表静音。这种映射允许模块输出一个简化的音量级别指标，可以用于直观显示当前的音量大小。

4、module mp3board

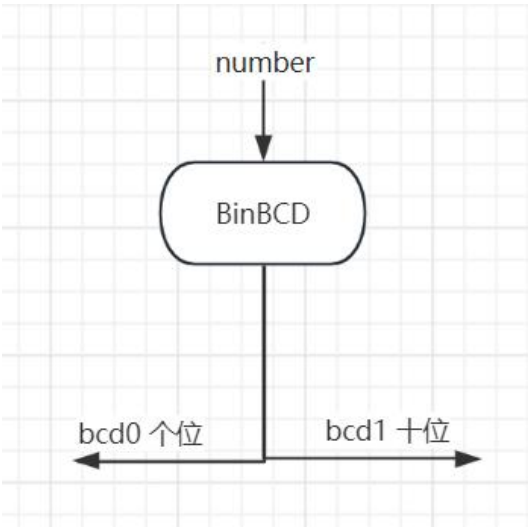



```
Input          clk,                  //2mhz
input          rst,                  //系统复位，低有效
input          play,                 //开始播放始播放请求
input          SO,                   //传出
input          DREQ,                 //数据请求，高电平时可传输数据
input          [7:0]music_id,        //第几首歌
input [15:0] volume,                 // 音量控制
output reg     XCS,                  //SCI 传输读写指令
output reg     XDCS,                 //SDI 传输数据
output         SCK,                  //时钟
output reg     SI,                   //传入 mp3
output reg     XRESET,               //硬件复位，低电平有效
```

模块通过内部状态机管理播放器的状态，包括复位、设置音量、播放音乐。
具体流程及各个功能如下。
（1）初始化(negedge clk 下的 if 分支)
当 rst 为低电平或者音乐 ID 变化，或者音量改变时，状态机复位到初始状态，准备新的播放流程。
（2）硬复位 (H_RESET 状态)
置 XRESET 为低电平来对 MP3 解码器进行硬件复位。完成后，切换到软复位状态。
（3）软复位 (S_RESET 状态)
发送软复位指令 cmd 到 MP3 解码器。通过循环发送 32 位软复位指令，每个时钟周期发送一位，直到所有位都发送完毕。
（4）设置音量 (SET_VOL 状态)
模块发送音量设置指令，类似于软复位，它通过 SCI 接口发送音量设置命令到 MP3 解码器。
（5）播放(PLAY 状态)

在播放状态，检查 play 信号，如果为低（即停止播放请求），则停止播放并复位到等待状态。如果 play 信号为高，则继续播放音乐。
将从块内存生成器读取的音乐数据（data0 到 data7）发送到 MP3 解码器。拉低 XDCS 信号，并在 DREQ 信号为高时将数据位 music_data[cntSended - 1]发送给解码器。
（6）音量变化 (posedge clk 下的 always 块)
每次时钟上升沿，检查音量 volume 是否与上次存储的 last_volume 不同。如果不同，设置 volume_changed 标志，并更新 last_volume。
（7）IP 核的块内存生成器（blk_mem_gen_x）
将事先存在 ip 核中的音乐文件根据 music_id 从相应的生成器中读取数据，写入 datax。

5、module Bin2BCD



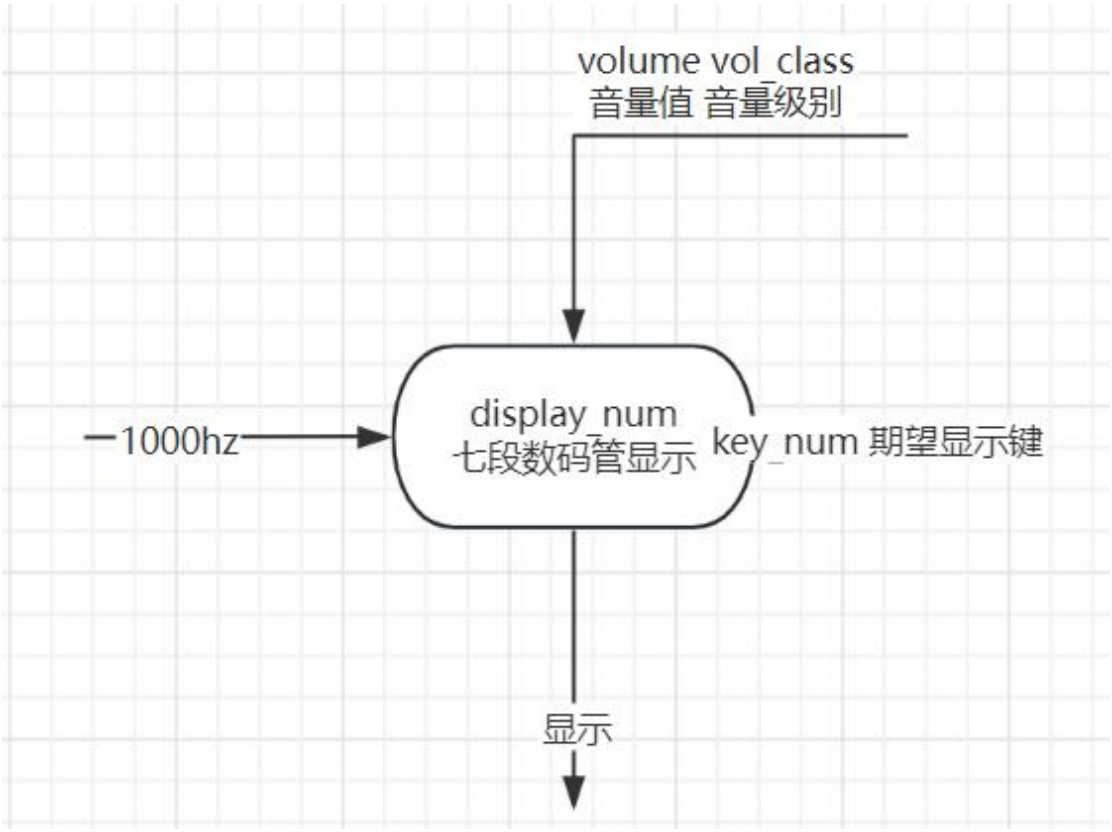
input [7:0] number, //处理 8 位数字
output reg [3:0] bcd0,//个位
output reg [3:0] bcd1,//十位

将一个 8 位的二进制数转换为二进制编码的十进制（BCD）。使得十进制数的显示和处理更为直观。

转换过程使用了双倍加 3 算法在 always @(number)块中，每次 number 发生变化时，以下步骤会被执行：

（1）初始化临时寄存器 temp_bcd0 和 temp_bcd1 为 0，这两个临时寄存器用来存储中间的 BCD 值。
（2）对于 number 的每一位（从最高位到最低位），执行以下步骤：
检查 temp_bcd0 和 temp_bcd1 是否大于 4。如果是，就在相应的寄存器上加 3。这是因为 BCD 的每一个数字不能超过 9，在二进制到 BCD 的转换过程中，任何大于 4 的 4 位组在左移之前必须加 3 来保持十进制的有效性。
将 temp_bcd0 和 temp_bcd1 向左移动一位（相当于乘以 2），将 number 的当前位添加到 temp_bcd0 的最低位。
（3）循环完成后，将临时寄存器的值赋给输出寄存器 bcd0 和 bcd1。

6、module display_num




```
input clk_1000hz,
input [7:0] score,          // 输入的由键码转换的对应希望显示的数
input [3:0] vol_class,   // 新增音量级别输入
output reg [7:0] shift,   // 控制 8 个数码管
output reg [6:0] oData, // 输出的数字
```

数码管显示模块，用于显示音量级别和音符。利用人眼的视觉暂留性实现单个数码管的对应控制显示。

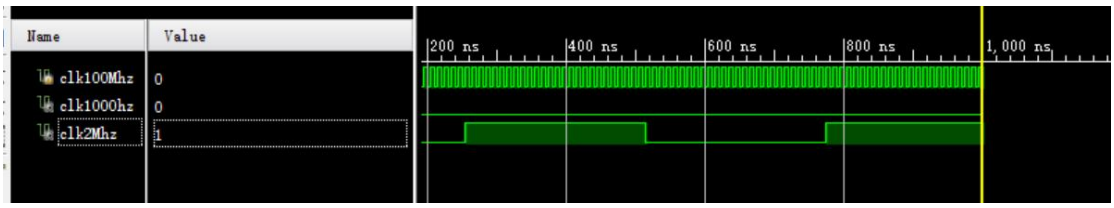
首先使用一个 Bin2BCD 子模块实例，它将 score 转换为两个 BCD 数字 Data[0]和 Data[1]。这样可以将分数值在数码管上以十进制形式显示。
在 always@(posedge clk_1000hz)块中，使用一个计数器 cnt 来遍历所有的数码管，根据 cnt 的值，使用 shift 信号来激活特定的数码管。cnt 的值从 0 增加到 8，对应于 8 个数码管。通过将 shift 置为全 1，然后将特定位设为 0 来激活特定的数码管。
接下来，根据 cnt 的值，使用 oData 信号来确定显示的内容：
当 cnt 大于 1 且小于 6 时，即 1-6 号数码管不显示任何内容（关闭）。
当 cnt 为 7 时，即 7 号数码管，显示 vol_class 的值，音量级别。使用 case 语句来为 oData 选择正确的 7 段信号组合，以显示对应的数字。
当 cnt 为 0 或 1 时，0-1 号数码管，显示分数的 BCD 表示，同样使用 case 语句为 oData 选择正确的信号组合。

五、测试模块建模

1、module divider

生成一个 100MHz 的时钟信号。这是通过一个 initial 块来实现的，用一个 forever 循环来不断翻转 clk100Mhz 的值。100MHz 时钟周期为 10ns，所以每 5ns 翻转一次。

通过观察波形图可以看到 clk2Mhz 是符合分频预期的，因为 1000hz 分频太小所以无法在波形图观察，但两种分频的思路完全相同，由 2MHz 正确可以保证分频达到预期效果。



2、module Bin2BCD

因为 Bin2BCD 模块是一个二进制到二进制编码的十进制（BCD）转换器，它能处理一个 8 位的二进制数，等价的十进制个位和十位分开输出给 bcd0 和 bcd1.

所以 testbench 的测试逻辑就是通过给输入的 number 赋不同的值，观察波形对应输出的 bcd0 和 bcd1，如果输出如期望值，则测试通过。

首先，将输入 number 初始化为 0，并等待 100 纳秒以确保所有的信号稳定下来。

然后，逐步给 number 赋予不同的值。

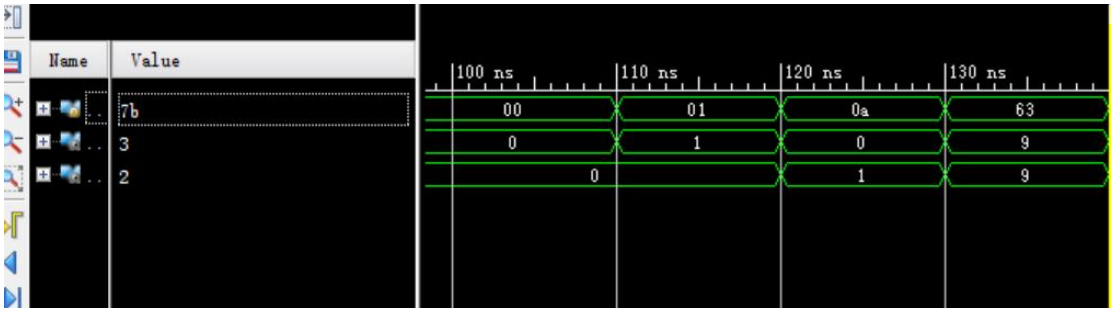
当 number 为 0 时，预期的 BCD 输出应为 00。

当 number 为 1 时，预期的 BCD 输出应为 01。

当 number 为 10 时，预期的 BCD 输出应为 10（注意，这是十进制数 10 的 BCD 编码，表示为 0001 0000）。

当 number 为 99 时，预期的 BCD 输出应为 99。

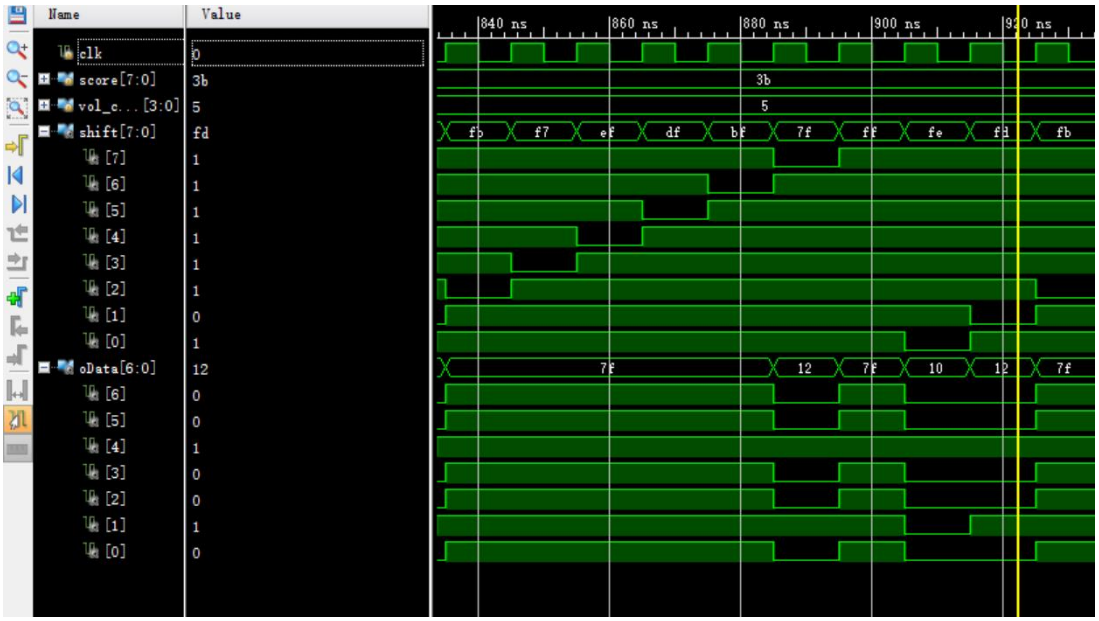
波形图如期望所示，测试通过。



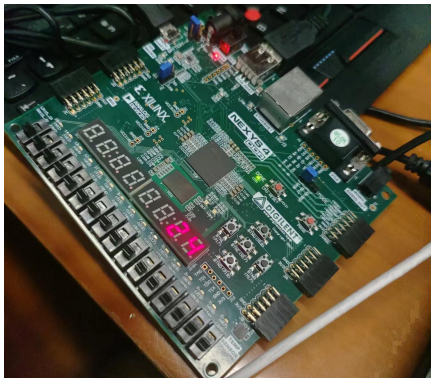
3、module display_num

七段数码管显示模块我选择先仿真测试，再下板测试。

Testbench 的测试逻辑就是输入一个符合要求的 score（以 59）为例，按照上面刚分析过的模块的原理，根据当前激活的数码管（由 shift 确定），oData 将设置相应的段以显示数字或字母。期望在切换到 shift[0]和 shift[1]时 oData[0]和 oData[1]的值能符合预期。如波形图所示，刚好对应的是 9 和 5 的显示预期（不是看数值，而是七段数码管的显示规律），并在其他状态都显示为 0，符合设定。



下板测试实际上就是测试时钟频率显示能否如预期般能够让人眼观察到。下板测试，此时键盘模块尚未完成，所以直接规定输入的 score 值，而不接入键盘，以 24 为例，能观察到此时显示如预期所示，如图。



4、module keyboard

由于此时数码管显示模块和分频器模块已经成功测试完毕，对键盘模块的测试使用数码管实时显示效率更高，这也是数码管显示模块显示保留 8-F 的原因，实际音色显示不会用到这些显示，但在键盘模块测试时，按下键盘会在数码管实时显示接收到的键码，同时将 key_state 连接到板上的 led 灯来判断按下松开处理是否正确。所以键盘模块选择下板测试。

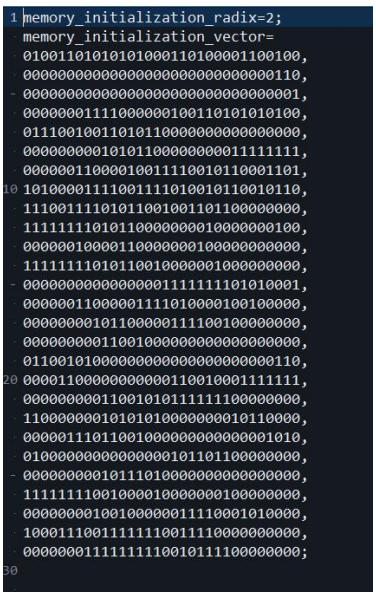
5、module vocal

音量模块部分的逻辑比较简单，接收键码，检查键码是否是上下箭头，更改音量大小和级别，输出。音量级别在最后的设计中本身就要在数码管显示，所以本模块也选择下板测试。按下上下箭头键，数码管显示如期望增大减小即测试完成。

6、module mp3board

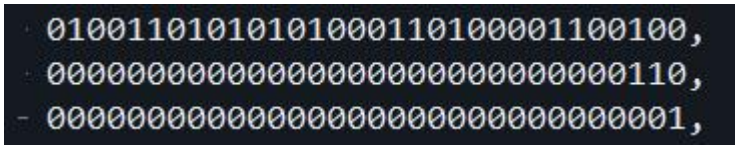
（1）对 MP3 模块的测试，首先要实现在 IP 核中音乐文件的存储，这需要有期望播放音乐的单轨道 midi 文件转换成的 coe 文件。对于每个音符的单轨道 midi 文件制作，我使用 Musescore 软件制作，并使用 Pic2Mif 软件将 midi 文件转换为 mif 文件（与 coe 文件格式相

近便于处理）。Mif 文件格式如左图，将其内容改为右图并将后缀调整为.coe 即可得到单音符的 coe 文件。



得到处理好的 coe 文件后将文件导入 ip 核，并在 MP3 模块实例化即可。

（2）由于是测试的最后一个模块，一开始选择直接运行 top 模块下板测试，但部分音不发声，不确定是 MP3 模块问题还是 coe 文件不正确，所以最终选择写 testbench 观察波形图测试文件读入是否正确。多次测试发现，只有 coe 文件的第三行是 00000001 的文件可以成功发声，00010001 的则不发声，推测是代表单双轨道的文件头。



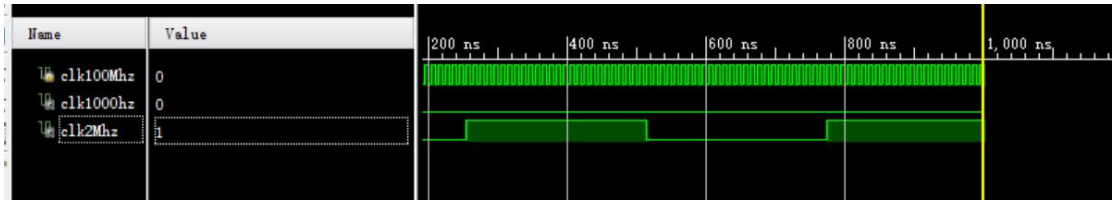
另外，对于 midi 文件的轨道数的确定，实际上我还使用了 c++代码辅助判断，通过读取文件头来判断轨道数，可以用于自己制作的 midi 文件，但不完全准确，因为本质上文件头可以修改，所以要保证测试的文件没有经过人为修改。当然我可以保证。

六、实验结果

1、子模块的仿真测试波形图以及下板图（含测试历程）

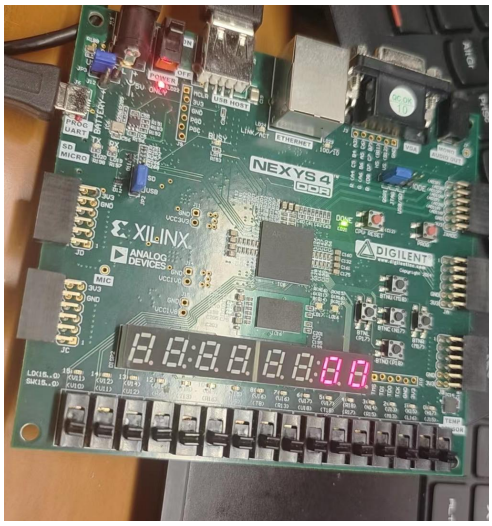
（1）module divider

分频器的验证比较顺利，一遍过，贴图如下。

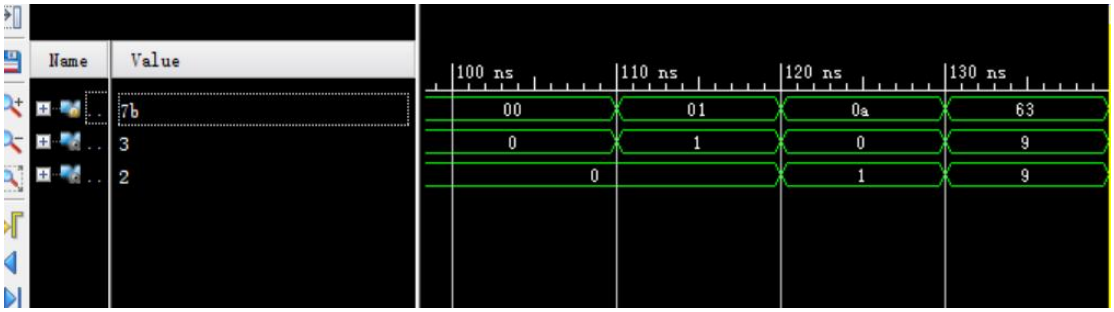


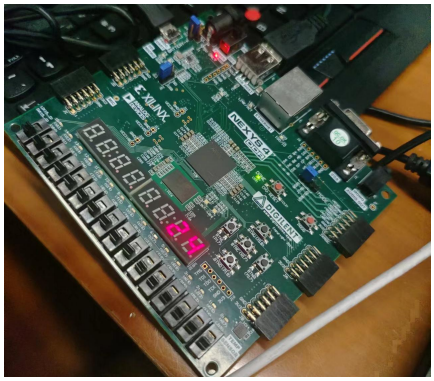
（2）module display_num（module Bin2BCD）

七段数码管的显示选择下板测试。此时下板测试的实例化模块仅有分频器和数码管显示模块。下板后可以成功在对应的数码管显示默认值 00（如图）。



由于此时没有完成键盘等模块，所以人为设置 display_num 的实例化模块的 score 数值，发现无论设置什么数值，显示始终为 00。这里总结出确实应该自顶向上测试，应当选择先测试 display_num 模块中的 Bin2BCD 模块，书写 Bin2BCD 的 testbench，观察波形图发现是 Bin2BCD 模块的处理逻辑有问题，修改后测试通过。下图一为 Bin2BCD 模块测试的波形图，下图二为修改过后可以正确显示的数码管。





至此 Bin2BCD 模块和 display_num 模块都已经调试完毕确认无误。接下来 top 模块同时调用 keyboard 模块。但键盘按下任何按键均无反应。经询问了解是学校的 thinkpad 键盘的问题，更换键盘后成功。

（3）module keyboard

对键盘模块的测试分为按键抬起落下部分、读取键码部分。

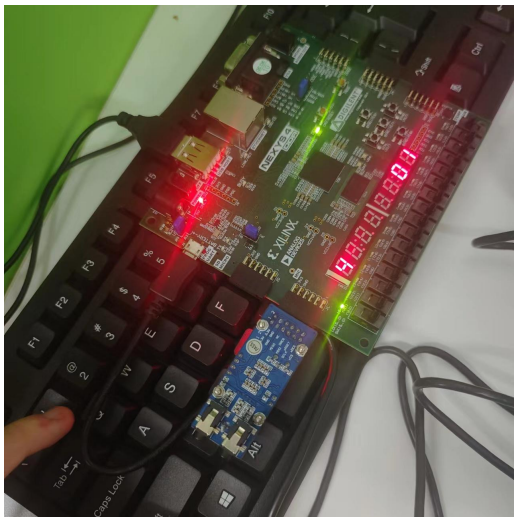
抬起落下部分主要是观察 key_state 的值，由于换成波形图测试反而增加工作量，所以直接将 key_state 作为 top 模块最终输出的量之一，对应板上的 V11led 灯，按下时亮，抬起灭。

读取键码部分，之前完成的七段数码管部分特意保留了对 8-F 的显示（尽管最终显示用不到），但对于读入键码的直接显示起到了很大帮助，主板的使用手册上告知的键码有部分错误，通过这种方法可以直观的得到按键的键码，便于在读入部分设置转换的 ascii 值。

至此键盘可以成功的抬起落下、读入正确的键码，可以成功处理并输出期望显示的数给 display_num 模块。

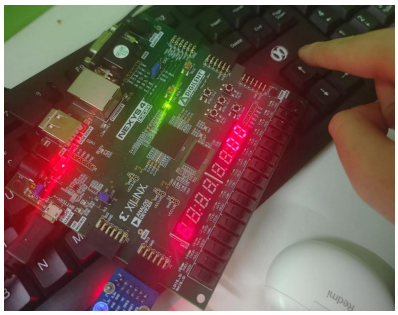
但仍存在一个遗留问题。

当键盘按下后需要等待大概 2 秒钟，数码管显示才会改变。接下来如果连续按，则可以无时差显示。另外，键盘支持热插拔，在热插拔后有了意外发现，重新插上后无需按下任何键，键盘都会输入 70 键码，key_state 亮起，推测是键盘设置的一些休眠问题。解决方法是后接入主板，则可以无延时完成键码的接收传递处理显示。如图是最终调整后按下按键，显示对应的数值，key_state 指示灯亮起，符合预期。



（4）module vocal

vocal 部分的测试逻辑同键盘部分相似，同样是按下键盘的某个键，对应显示某个值，如图是按下↑↓后数码管首位显示。

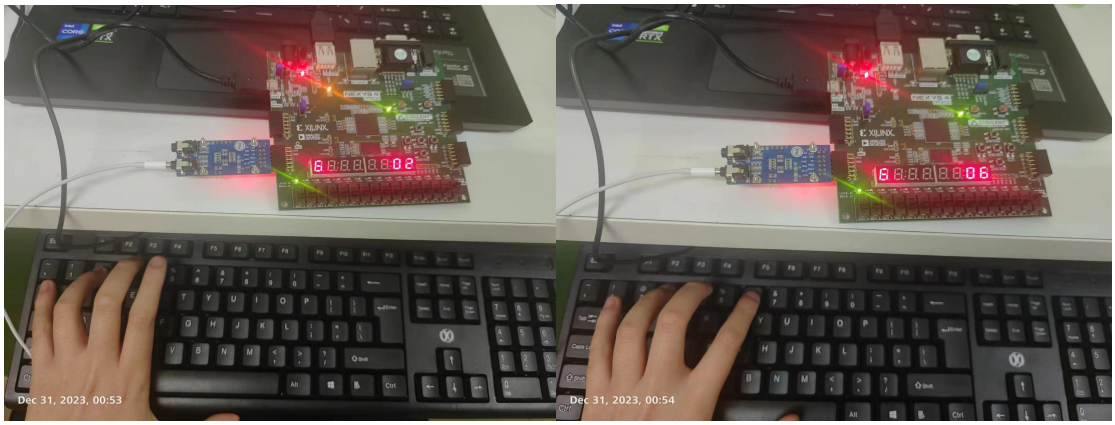


（5）module mp3board

MP3 部分的测试遇到的问题是，按下按键后期望在同时响起对应的音，这还与 MP3 播放器不同，如果是播放完整的音乐，按下后延时几秒开始播放也无伤大雅，可是电子琴就完全不能延时播放。由于 mp3 是最后一个测试的模块，所以推测问题就在本模块而非其他模块造成的延时。经过反复检查，MP3 的 PLAY 部分会有默认的延时值设置，只要将数值改到无限小即消除延时。

至此每个子模块的测试均已完成，由于是前面模块共同参与后面模块的测试，所以 MP3 模块测试完成后即整个系统通过。

2、完成后的总体下板图



七、结论

（1）已完成内容

成功处理 PS2 键盘的输入并将其转化为期望数字显示在数码管上，成功处理 MP3 各种播放设置并实现 ip 核数据文件的载入，将键盘、MP3、Nexys-4 板连接后各部件可以协同工作，基本实现预期设计的电子琴功能。

（2）可改进部分

本项目的电子琴实现是靠 MP3 播放单个音符的音乐，实际上用 MP3 播放一整首音乐的逻辑与之完全相同，可以添加一个功能，就是播放完整曲目（例曲），其实我也在项目文件中放了一首完整歌曲的 coe 文件可供 ip 核导入 MP3 播放，只是没有为之配套对应的数码管显示，所以并未就此功能继续延伸。

八、心得体会及建议

（1）期望与实践

经历本次项目完成后较为直观感受到了理论代码与实际测试的差距，接通硬件后有很多时候结果不同预期，其中原因有对硬件协议时序了解的不足、代码设计的不完善等原因，更多强调实践，要从多个方面去思考原因。选择的测试方法也决定了某时的测试效率，不用一味的 testbench，有时通过直观显示来得到结果会更快更直观一些。

（2）额外知识

对 MP3 部件有了更深刻的了解，使用前时序的设计，状态机的测试等等。本次在 midi 文件

部分的耗时较长，尽管使用手册中已经告知可处理的是单音轨的 midi 文件，但对 midi 文件的制作以及轨道数的判断都不是很了解。网络上并没有期望的单音轨单音符谜底文件，所以在此过程中先是学习了一个制作音乐的 Musescore 软件，自己制作完全部的 midi 文件。下一步是 coe 文件的获取，几经波折才了解到 midi 转 mif，最后得到 coe 文件。一整个流程下来，对 midi 文件格式、coe 文件格式都有了更深刻的了解，也算是额外收获。

（3）不必要的耗时

在整个项目的完成过程中同样有一些不必要的耗时。一是学校下发的 thinkpad 键盘，起初完成时始终无反应，卡在键盘部分好久，主要是当时是不知道是键盘原因还是代码原因，多次修改无果后更换键盘才知道 thinkpad 的问题，希望学校可以更换键盘的品牌，我测试了联想、罗技、戴尔的 PS2 键盘都可以跑通，不知是我代码不够完善还是 thinkpad 或许并不遵循 PS2 协议？

（4）协议

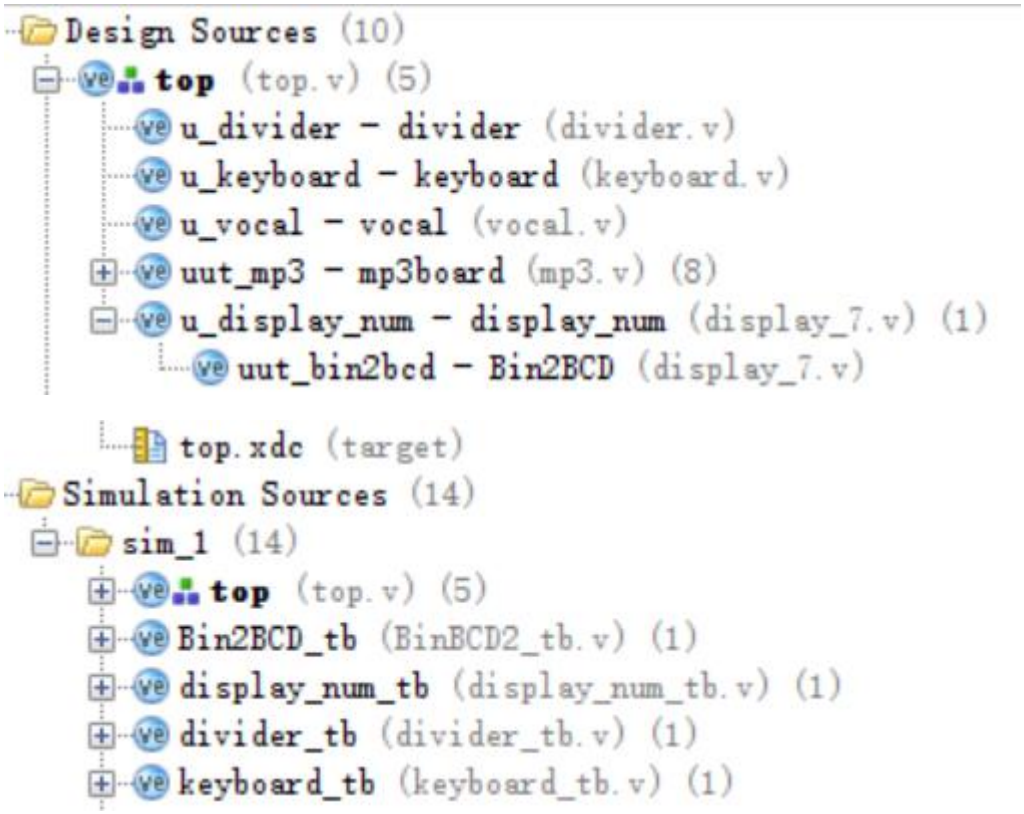
本项目是我第一次接触到协议相关，对硬件的理解更深了一些，不可避免要自行寻找配置 xdc 文件，这是之前没有过的，所以对接口的作用、意义有了更深刻的了解。

（5）建议

希望在与课程知识关系不大的部分有直接提供的资料，例如 midi 转 coe 方法、coe 文件格式。另外希望学校在外设部分能有更详细的中文资料下发。

九、附录

文件总览



1、设计文件

（1）top.v（顶层模块）

```
module top(
    input clk100Mhz,          // 主时钟信号, 用于系统同步
    input rst,                // 全局复位信号, 用于初始化系统
    input key_clk,            // 键盘时钟信号, 用于键盘数据的同步
    input key_data,           // 键盘数据信号, 用于接收键盘的数据

    output [7:0] shift,       // 数码管片选信号, 用于控制哪个数码管显示
    output [6:0] oData,       // 数码管显示信号, 用于控制数码管显示的内容

    // MP3 模块的接口
    input          SO,        // MP3 模块的串行输出信号
    input          DREQ,      // 数据请求信号, 高电平表示可以传输数据
    output         XCS,       // MP3 模块控制信号, 用于控制 SCI 读写操作
    output         XDCS,      // MP3 模块数据传输控制信号, 用于 SDI 数据传输
    output         SCK,       // 时钟信号, 用于 MP3 模块的数据同步
    output         SI,        // MP3 模块的串行输入信号
    output         XRESET,    // 硬件复位信号, 用于重置 MP3 模块

    output wire key_state   // 按键状态指示, 按下时为高电平
);

    // 内部信号定义
    wire [7:0] key_num;       // 从键盘模块得到的按键对应的数字
    wire [7:0] key_byte;      // 从键盘模块得到的按键扫描码
    wire clk1000hz;           // 从分频器模块得到的 1000Hz 时钟信号
    wire clk2Mhz;             // 从分频器模块得到的 2MHz 时钟信号
    wire [15:0] volume;       // 从音量控制模块得到的音量值
    wire [3:0] vol_class;     // 从音量控制模块得到的音量级别

    // 分频器模块实例化
    divider u_divider(
        .clk100Mhz(clk100Mhz),
        .clk1000hz(clk1000hz),
        .clk2Mhz(clk2Mhz)
    );

    // 键盘控制模块实例化
    keyboard u_keyboard(
        .clk_in(clk100Mhz),
        .rst(rst),
        .key_clk(key_clk),
        .key_data(key_data),
        .key_state(key_state),
        .key_num(key_num),
        .key_byte(key_byte)
    );

    // 音量控制模块实例化
    vocal u_vocal(
        .clk(clk2Mhz),
        .key_num(key_byte),
        .key_state(key_state),
        .vol(volume),
        .vol_class(vol_class)
    );

    // MP3 播放控制模块实例化
    mp3board uut_mp3(
        .clk(clk2Mhz),
        .rst(rst),
        .play(key_state),
        .SO(SO),
        .DREQ(DREQ),
        .XCS(XCS),
```

```
        .XDCS(XDCS),
        .SCK(SCK),
        .SI(SI),
        .XRESET(XRESET),
        .music_id(key_num),
        .volume(volume)
    );
```

```
    // 数码管显示控制模块实例化
    display_num u_display_num(
        .clk_1000hz(clk1000hz),
        .score(key_num),
        .shift(shift),
        .oData(oData),
        .vol_class(vol_class)
    );
endmodule
```

（2）divider.v

```
module divider(
    input clk100Mhz,
    output reg clk1000hz = 0,
    output reg clk2Mhz = 0 // 新增的 2MHz 时钟输出
);
    integer cnt = 0;
    integer cnt2MHz = 0; // 用于 2MHz 时钟的计数器

    always @(posedge clk100Mhz) begin
        // 1000Hz 时钟分频器
        if (cnt < 50000) begin
            cnt <= cnt + 1;
        end else begin
            cnt <= 0;
            clk1000hz <= ~clk1000hz;
        end

        // 2MHz 时钟分频器
        if (cnt2MHz < 25) begin
            cnt2MHz <= cnt2MHz + 1;
        end else begin
            cnt2MHz <= 0;
            clk2Mhz <= ~clk2Mhz;
        end
    end
Endmodule
```

（3）keyboard.v

```
module keyboard//将键盘键码转化为 ASCII 码
(
input                   clk_in,
        //系统时钟
input                   rst,        //系统复位，低有效
input                   key_clk,
    //PS2 键盘时钟输入 f4
input                   key_data,
    //PS2 键盘数据输入 b2
output  reg             key_state,
        //键盘的按下状态，按下置 1，松开置 0
output  reg     [7:0]key_num,       //按键键值对应的数字
output reg [7:0] key_byte   // 原始键盘扫描码
);

reg     key_clk_r0 = 1'b1,key_clk_r1 = 1'b1;
reg     key_data_r0 = 1'b1,key_data_r1 = 1'b1;
//对键盘时钟数据信号进行延时锁存
always @ (posedge clk_in or negedge rst) begin
    if(!rst) begin
        key_clk_r0 <= 1'b1;
        key_clk_r1 <= 1'b1;
        key_data_r0 <= 1'b1;
        key_data_r1 <= 1'b1;
    end else begin
        key_clk_r0 <= key_clk;
```

```
            key_clk_r1 <= key_clk_r0;
            key_data_r0 <= key_data;
            key_data_r1 <= key_data_r0;
        end
end

//键盘时钟信号下降沿检测
wire key_clk_neg = key_clk_r1 & (~key_clk_r0);

reg [3:0]cnt;
reg [7:0]temp_data;
//根据键盘的时钟信号的下降沿读取数据
always @ (posedge clk_in or negedge rst) begin
    if(!rst) begin
        cnt <= 4'd0;
        temp_data <= 8'd0;
    end else if(key_clk_neg) begin
        if(cnt >= 4'd10) cnt <= 4'd0;
        else cnt <= cnt + 1'b1;
        case (cnt)
            4'd0: ;     //起始位
            4'd1:       temp_data[0]    <= key_data_r1;   //数据位 bit0
            4'd2:       temp_data[1]    <= key_data_r1;   //数据位 bit1
            4'd3:       temp_data[2]    <= key_data_r1;   //数据位 bit2
            4'd4:       temp_data[3]    <= key_data_r1;   //数据位 bit3
            4'd5:       temp_data[4]    <= key_data_r1;   //数据位 bit4
            4'd6:       temp_data[5]    <= key_data_r1;   //数据位 bit5
            4'd7:       temp_data[6]    <= key_data_r1;   //数据位 bit6
            4'd8:       temp_data[7]    <= key_data_r1;   //数据位 bit7
            4'd9: ;     //校验位
            4'd10:;     //结束位
            default: ;
        endcase
    end
end

reg key_break = 1'b0;

//根据通码和断码判定按键的当前是按下还是松开
always @ (posedge clk_in or negedge rst) begin
    if(!rst) begin
        key_break <= 1'b0;
        key_state <= 1'b0;
        key_byte <= 1'b0;
    end else if(cnt==4'd10 && key_clk_neg) begin
        if(temp_data == 8'hf0) key_break <= 1'b1;  //收到断码（8'hf0）表示按键松开，下一个数据为断码，设置断码标示为 1
        else if(!key_break) begin       //当断码标示为 0 时，表示当前数据为按下数据，输出键值并设置按下标示为 1
            key_state <= 1'b1;
            key_byte <= temp_data;
        end else begin //当断码标示为 1 时，标示当前数据为松开数据，断码标示和按下标示都清 0
            key_state <= 1'b0;
            key_break <= 1'b0;
            key_byte<=0;
        end
    end
end

//将键盘返回的有效键值转换为按键字母对应的 ASCII 码
always @ (key_byte) begin
    case (key_byte)         //translate key_byte to key_ascii
        8'h16: key_num <= 8'd1;//01
        8'h1e: key_num <= 8'd2;//02
        8'h26: key_num <= 8'd3;//03
        8'h25: key_num <= 8'd4;//04
        8'h2e: key_num <= 8'd5;//05
        8'h36: key_num <= 8'd6;//06
```

```
            8'h3d: key_num <= 8'd7;//07
            8'h15: key_num <= 8'd11;//11 Q
            8'h1d: key_num <= 8'd12;//12 W
            8'h24: key_num <= 8'd13;//13 E
            8'h2d: key_num <= 8'd14;//14 R
            8'h2c: key_num <= 8'd15;//15 T
            8'h35: key_num <= 8'd16;//16 Y
            8'h3c: key_num <= 8'd17;//17 U
            8'h1c: key_num <= 8'd21;//21 A
            8'h1b: key_num <= 8'd22;//22 S
            8'h23: key_num <= 8'd23;//23 D
            8'h2b: key_num <= 8'd24;//24 F
            8'h34: key_num <= 8'd25;//25 G
            8'h33: key_num <= 8'd26;//26 H
            8'h3b: key_num <= 8'd27;//27 J
            default: key_num=8'd0;        //改一
下看看
        endcase
    end
    endmodule
```

（4）vocal.v

```
module vocal(
        input clk,
        input [7:0] key_num,      // 键码
        input  key_state,                 //  更 名 为
key_state
        output reg [3:0] vol_class=9,
        output reg [15:0] vol=16'h0000
);
        wire [15:0] adjusted_vol;
        assign adjusted_vol = vol;
        integer clk_cnt = 0;

        always @(posedge clk) begin
            if (clk_cnt == 200000) begin
                clk_cnt <= 0;
                //  当 key_state 为+电平，即键
被按下时
                if (key_state) begin
                    case (key_num) //
                        8'b01110101:  begin//
上 11
                                vol  <=  (vol  ==
16'h0000) ? 16'h0000 : (vol - 16'h197f);
                        end
                        8'b01110010:  begin//
下 0E
                                vol  <=  (vol  ==
16'hfef6) ? 16'hfefe : (vol + 16'h197f);
                        end
                    endcase
                end
            end else begin
                clk_cnt <= clk_cnt + 1;
            end
        end

        always @(posedge clk) begin
            case (vol)
                16'he577: vol_class <= 0;
                16'hcbf8: vol_class <= 1;
                16'hb279: vol_class <= 2;
                16'h98fa: vol_class <= 3;
                16'h7f7b: vol_class <= 4;
                16'h65fc: vol_class <= 5;
                16'h4c7d: vol_class <= 6;
                16'h32fe: vol_class <= 7;
                16'h197f: vol_class <= 8;
                16'h0000: vol_class <= 9;
            endcase
        end
Endmodule
```

（5）mp3board.v

```
module mp3board(
    input clk,                          //2mhz

    input       rst,
    input       play,                   //开始播放始播放请求

    input       SO,                     //传出
    input       DREQ,                   //数据请求，高电平时可传输数据
    input       [7:0]music_id,          //第几首歌
    input [15:0] volume,                // 音量控制
    output reg  XCS,                    //SCI传输读写指令
    output reg  XDCS,                   //SDI传输数据
    output      SCK,                    //时钟
    output reg  SI,                     //传入mp3
    output reg  XRESET                  //硬件复位，低电平有效
    );
    parameter   H_RESET     = 4'd0,     //硬复位
                S_RESET     = 4'd1,     //软复位
                SET_CLOCKF  = 4'd2,     //设置时钟寄存器
                SET_BASS    = 4'd3,     //设置音调寄存器
                SET_VOL     = 4'd4,     //设置音量
                WAITE       = 4'd5,     //等待
                PLAY        = 4'd6,     //播放
                END         = 6'd7;     //结束
    reg [3:0]       state       = WAITE; //状态
    reg [31:0]      cntdown     = 32'd0; //延时
    reg [31:0]      cmd         = 32'd0; //指令与地
    reg [7:0]       cntData     = 8'd32; //SCI 指令地址位数计数

    reg [31:0]      music_data  = 32'd0;    //音乐数据
    reg [31:0]      cntSended   = 32'd32;   //SDI 当前 4 字节已传送 BIT

    reg [16:0]      addra       = 16'd0; //ROM 中的地址
     // 定义音乐数据线
        wire [31:0] data0, data1, data2, data3, data4, data5, data6, data7, data8,
                    data9, data10, data11, data12, data13, data14, data15, data16,
                    data17, data18, data19, data20;
    reg             ena         = 0;
    reg             [7:0]pre_id = 0;
        reg [31:0] volcmd = 0;

    assign SCK = (clk & ena);
    //速度控制
    reg [31:0] mp3Speed=5;//延迟
    always @(music_id) begin
        case (music_id)
            8'd0 : begin
                mp3Speed <= 5;
            end
            8'd1 : begin
                mp3Speed <= 5;
            end
            8'd2 : begin
                mp3Speed <= 5;
            end
```

```
                8'd3 : begin
                    mp3Speed <= 5;
                end
                8'd4 : begin
                    mp3Speed <= 5;
                end
            default: begin
                    mp3Speed <= 5;
            end
            endcase
        end

        always @(negedge clk) begin
            if(!rst || pre_id!=music_id|| volume_changed) begin
                pre_id <= music_id;
                XDCS <= 1'b1;
                ena <= 0;
                SI <= 1'b0;
                XCS <= 1'b1;
                state <= WAITE;
                XRESET <= 1'b1;
                addra <= 17'd0;
                cntSended <= 32'd32;
                music_data <= 32'd0;

            end
            else begin
                case (state)
                    /*---------------- 等 待 ----------------*/
                    WAITE:begin
                    if(cntdown > 0)
                        cntdown <= cntdown - 1'b1;
                    //转到硬复位
                    else begin
                        cntdown <= 32'd1000;
                        state <= H_RESET;
                    end
                    end
                    /*---------------- 硬 复 ----------------*/
                    H_RESET:begin
                    if(cntdown > 0)
                        cntdown <= cntdown - 1'b1;
                    else begin
                        XCS <= 1'b1;
                        XRESET <= 1'b0;
                        cntdown <= 32'd16700;      //复位后延时一段时

                        state <= S_RESET;
                    //转移到软复位
                        cmd <= 32'h02_00_08_04;      //软复位指
                        cntData <= 8'd32;
                    //指令、地、数据长度
                    end
                    end
                    /*---------------- 软 复 ----------------*/
                    S_RESET:begin
                    if(cntdown > 0) begin
                        XRESET <= (cntdown < 32'd16650);
                        cntdown <= cntdown - 1'b1;
                    end
                    else if(cntData == 0) begin     //软复位结
                        cntdown <= 32'd16600;

                        state <= SET_VOL;
                    //转移到设置 VOL
                        cmd <= 32'h02_0b_00_00;
                        cntData <= 8'd32;

                        XCS <= 1'b1;
                    //拉高 XCS
                        ena <= 1'b0;
                    //关闭输入时钟
                        SI <= 1'b0;
```

```
                end                                 1'b0;
                else if(DREQ) begin                                 SI <= 1'b0;
//当 DREQ 有效时开始软复位                                          cntSended
                    XCS <= 1'b0;                    <= 32'd32;
                    ena <= 1'b1;                                    case
                    SI <= cmd[cntData -             (music_id)
1];                                                                     8'b1:
                    cntData <= cntData -            music_data <= data0;
1'b1;                                                                   8'd2:
                end                                 music_data <= data1;
                else begin                                              8'd3:
                    XCS <= 1'b1;                    music_data <= data2;
//DREQ 无效时继续等                                                     8'd4:
                    ena <= 1'b0;                    music_data <= data3;
                    SI <= 1'b0;                                         8'd5:
                end                                 music_data <= data4;
                end                                                     8'd6:
                                                    music_data <= data5;
                /*---------- 播 放 音 乐                                8'd7:
----------*/                                        music_data <= data6;
/*----------播放音乐----------*/                                       8'd11:
                PLAY: begin                         music_data <= data7;
                    if(cntdown > 0) begin                               8'd12:
                        cntdown <=                  music_data <= data8;
cntdown - 1'b1;                                                         8'd13:
                    end                             music_data <= data9;
                    else if(!play) begin                                8'd14:
// 检查是否收到停止播放的请求                       music_data <= data10;
                        // 重置相关信                                   8'd15:
号以停止播放                                        music_data <= data11;
                        XDCS <= 1'b1;                                   8'd16:
                        ena <= 0;                   music_data <= data12;
                        SI <= 1'b0;                                     8'd17:
                        state <= WAITE;             music_data <= data13;
// 返回等待状态                                                         8'd21:
                    end                             music_data <= data14;
                    else begin                                          8'd22:
                        XDCS <= 1'b0;               music_data <= data15;
                        ena <= 1'b1;                                    8'd23:
                        if(cntSended ==             music_data <= data16;
0) begin                 //传输 4 字节                                  8'd24:
                            XDCS <=                 music_data <= data17;
1'b1;                      //拉高 XDCS                                  8'd25:
                            ena <=                  music_data <= data18;
```

```
                                        8'd26:
music_data <= data19;
                                        8'd27:
music_data <= data20;

default: ;
                                    endcase
                                    addra    <=
addra + 1'b1;
                                end
                                else begin
                                    //当 DREQ
有效 或当前字节尚未发送完 则继续传
                                    if(DREQ ||
(cntSended != 32 && cntSended != 24 &&
cntSended != 16 && cntSended != 8)) begin
                                        SI   <=
music_data[cntSended - 1];

cntSended <= cntSended - 1'b1;
                                        ena
<= 1;
                                        XDCS
<= 1'b0;
                                    end
                                    else   begin
//DREQ 拉低，停止传
                                        ena
<= 1'b0;
                                        XDCS
<= 1'b1;
                                        SI   <=
1'b0;
                                    end
                                end
                            end
                        end
                                        // 新
增加的设置音量状态
                                    SET_VOL:
begin

if (cntData == 0) begin

state <= PLAY;

XCS <= 1;

ena <= 0;

SI <= 0;

end else if (DREQ) begin

XCS <= 0;

ena <= 1;

SI <= volcmd[31];

volcmd <= {volcmd[30:0], volcmd[31]};

cntData <= cntData - 1;

end else begin

XCS <= 1;

ena <= 0;

SI <= 0;

end

end
                        /*---------------------- 寄 存
器配------------------*/
                        default:
                        if(cntdown > 0)
                            cntdown <= cntdown
- 1'b1;
                        else   if(cntData   ==   0)
begin                   //结束次 SCI 写入
                            if(state              ==
SET_CLOCKF) begin
                                cntdown       <=
mp3Speed;//32'd1700000;
                                state <= PLAY;
```

```
                    end
                    else    if(state    ==
SET_BASS) begin
                        cntdown     <=
32'd2100;
                        cmd         <=
32'h02_03_70_00;
                        state       <=
SET_CLOCKF;
                    end
                    else            begin
//SET_VAL
                        cntdown     <=
32'd2100;
                        cmd         <=
32'h02_02_00_00;
                        state       <=
SET_BASS;
                    end
                    cntData <= 8'd32;
                    XCS <= 1'b1;
                    ena <= 1'b0;
                    SI <= 1'b0;
                end
                else    if(DREQ)    begin
//写入 SCI 指令、地、数
                    XCS <= 1'b0;
                    ena <= 1'b1;
                    SI  <= cmd[cntData -
1];
                    cntData <= cntData -
1'b1;
                end
                else                begin
//DREQ 拉低，等
                    XCS <= 1'b1;
                    ena <= 1'b0;
                    SI <= 1'b0;
                end
            endcase
        end
    end

// 检测音量是否改变
reg [15:0] last_volume = 0;   // 用于存储上
次的音量值
reg volume_changed = 0;         // 音量改变
标志
always @(posedge clk) begin
    if (last_volume != volume) begin
        last_volume <= volume;
        volume_changed <= 1;   // 设置音
量改变标志
    end
    else begin
            // 如果音量没有变化，清除
volume_changed 标志
            volume_changed <= 0;
        end
end

// data0 对应的块内存生成器
blk_mem_gen_0 d1 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data0)               // 数据输出
);

// data1 对应的块内存生成器
blk_mem_gen_1 d2 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data1)               // 数据输出
);

// data2 对应的块内存生成器
blk_mem_gen_2 d3 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data2)               // 数据输出
);

// data3 对应的块内存生成器
blk_mem_gen_3 d4 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data3)               // 数据输出
);
```

```
// data4 对应的块内存生成器
blk_mem_gen_4 d5 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data4)               // 数据输出
);

// data5 对应的块内存生成器
blk_mem_gen_5 d6 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data5)               // 数据输出
);

// data6 对应的块内存生成器
blk_mem_gen_6 d7 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data6)               // 数据输出
);

// data7 对应的块内存生成器
blk_mem_gen_7 m1 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data7)               // 数据输出
);

// data8 对应的块内存生成器
blk_mem_gen_8 m2 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data8)               // 数据输出
);
// data9 对应的块内存生成器
blk_mem_gen_9 m3 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data9)               // 数据输出
);

// data10 对应的块内存生成器
blk_mem_gen_10 m4 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data10)              // 数据输出
);

// data11 对应的块内存生成器
blk_mem_gen_11 m5 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data11)              // 数据输出
);

// data12 对应的块内存生成器
blk_mem_gen_12 m6 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data12)              // 数据输出
);

// data13 对应的块内存生成器
blk_mem_gen_13 m7 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data13)              // 数据输出
);
// data14 对应的块内存生成器
blk_mem_gen_14 h1 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data14)              // 数据输出
);

// data15 对应的块内存生成器
blk_mem_gen_15 h2 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data15)              // 数据输出
);

// data16 对应的块内存生成器
blk_mem_gen_16 h3(
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data16)              // 数据输出
```

```
);

// data17 对应的块内存生成器
blk_mem_gen_17 h4 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data17)              // 数据输出
);

// data18 对应的块内存生成器
blk_mem_gen_18 h5 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data18)              // 数据输出
);
```

（6）display_num.v

```
module Bin2BCD(
    input [7:0] number,   // 仅处理 8 位数字
    output reg [3:0] bcd0,
    output reg [3:0] bcd1
);
    reg [3:0] temp_bcd0, temp_bcd1;
    integer i;

    always @(number) begin
        temp_bcd0 = 0;
        temp_bcd1 = 0;

        for (i = 7; i >= 0; i = i - 1) begin
            // Check if adding 3 to BCD digits is necessary
            if (temp_bcd0 > 4)
                temp_bcd0 = temp_bcd0 + 3;
            if (temp_bcd1 > 4)
                temp_bcd1 = temp_bcd1 + 3;

            // Shift left by one (multiply by 2)
            temp_bcd1 = temp_bcd1 << 1;
            temp_bcd1[0] = temp_bcd0[3];
            temp_bcd0 = temp_bcd0 << 1;
            temp_bcd0[0] = number[i];
        end

        bcd0 = temp_bcd0;
        bcd1 = temp_bcd1;
    end
endmodule
```

```
// data19 对应的块内存生成器
blk_mem_gen_19 h6 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data19)              // 数据输出
);

// data20 对应的块内存生成器
blk_mem_gen_20 h7 (
    .clka(clk),                 // 时钟
    .addra(addra),              // 地址
    .douta(data20)              // 数据输出
);
```

```
module display_num(
    input clk_1000hz,
    input [7:0] score,
    input [3:0] vol_class,   // 新增音量级别输入
    output reg [7:0] shift,   // 控制 8 个数码管
    output reg [6:0] oData
);
    wire [3:0] Data[1:0];   // 只需前两个 BCD 数据
    reg [3:0] cnt = 0;        // 4 位计数器

    // 转换为 BCD
    Bin2BCD uut_bin2bcd(
        .number(score),
        .bcd0(Data[0]),
        .bcd1(Data[1])
    );
```

```
    // 片选输出
    always@(posedge clk_1000hz) begin
        if(cnt == 4'd8)
            cnt <= 0;
        else
            cnt <= cnt + 1;
        shift <= 8'b1111_1111;
        shift[cnt] <= 0;   // 选择一个数码管进行输出

       // 当 cnt 大于 1 且小于 6 时，关闭数码管
                if(cnt > 1 && cnt < 7) begin
                    oData <= 7'b1111111; // 关闭数码管
                end
                // 显示音量级别在数码管 7 和 8
                    else if (cnt == 7) begin
                        case (vol_class)

4'b0000: oData <= 7'b1000000;//0

4'b0001: oData <= 7'b1111001;//1

4'b0010: oData <= 7'b0100100;//2

4'b0011: oData <= 7'b0110000;//3

4'b0100: oData <= 7'b0011001;//4

4'b0101: oData <= 7'b0010010;//5

4'b0110: oData <= 7'b0000010;//6

4'b0111: oData <= 7'b1111000;//7

4'b1000: oData <= 7'b0000000; // 8

4'b1001: oData <= 7'b0010000; // 9

default: oData <= 7'b1111111;
                        endcase
                    end
        else begin
            case (Data[cnt])
                4'b0000:    oData    <= 7'b1000000;//0
                4'b0001:    oData    <= 7'b1111001;//1
                4'b0010:    oData    <= 7'b0100100;//2
                4'b0011:    oData    <= 7'b0110000;//3
                4'b0100:    oData    <= 7'b0011001;//4
                4'b0101:    oData    <= 7'b0010010;//5
                4'b0110:    oData    <= 7'b0000010;//6
                4'b0111:    oData    <= 7'b1111000;//7
                4'b1000:    oData    <= 7'b0000000; // 8
                    4'b1001:  oData  <= 7'b0010000; // 9
                    4'b1010:  oData  <= 7'b0001000; // A
                    4'b1011:  oData  <= 7'b0000011; // b
                    4'b1100:  oData  <= 7'b1000110; // C
                    4'b1101:  oData  <= 7'b0100001; // d
                    4'b1110:  oData  <= 7'b0000110; // E
                    4'b1111:  oData  <= 7'b0001110; // F
                default:    oData    <= 7'b1111111;
            endcase
        end
    end
endmodule
```

2、xdc 文件

Top.xdc

```
set_property PACKAGE_PIN E3 [get_ports clk100Mhz]
set_property IOSTANDARD LVCMOS33 [get_ports clk100Mhz]

set_property PACKAGE_PIN J15 [get_ports rst]
set_property IOSTANDARD LVCMOS33 [get_ports rst]

set_property PACKAGE_PIN F4 [get_ports key_clk]
set_property PACKAGE_PIN B2 [get_ports key_data]
set_property IOSTANDARD LVCMOS33 [get_ports key_clk]
set_property IOSTANDARD LVCMOS33 [get_ports key_data]

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 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 IOSTANDARD LVCMOS33 [get_ports {oData[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {oData[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {oData[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {oData[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {oData[2]}]
set_property IOSTANDARD LVCMOS33
```

```
[get_ports {oData[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {oData[0]}]
set_property PACKAGE_PIN L18 [get_ports {oData[6]}]
set_property PACKAGE_PIN T11 [get_ports {oData[5]}]
set_property PACKAGE_PIN P15 [get_ports {oData[4]}]
set_property PACKAGE_PIN K13 [get_ports {oData[3]}]
set_property PACKAGE_PIN K16 [get_ports {oData[2]}]
set_property PACKAGE_PIN R10 [get_ports {oData[1]}]
set_property PACKAGE_PIN T10 [get_ports {oData[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports XCS]
set_property IOSTANDARD LVCMOS33 [get_ports XDCS]
set_property IOSTANDARD LVCMOS33 [get_ports DREQ]
set_property IOSTANDARD LVCMOS33 [get_ports SO]
set_property IOSTANDARD LVCMOS33 [get_ports SI]
set_property IOSTANDARD LVCMOS33 [get_ports XRESET]
set_property IOSTANDARD LVCMOS33 [get_ports SCK]
set_property PACKAGE_PIN K1 [get_ports XDCS]
set_property PACKAGE_PIN F6 [get_ports XRESET]
set_property PACKAGE_PIN J2 [get_ports DREQ]
set_property PACKAGE_PIN E7 [get_ports XCS]
set_property PACKAGE_PIN J3 [get_ports SCK]
set_property PACKAGE_PIN J4 [get_ports SI]
set_property PACKAGE_PIN E6 [get_ports SO]

set_property IOSTANDARD LVCMOS33 [get_ports key_state]
set_property PACKAGE_PIN V11 [get_ports key_state]
```

3、测试文件

（1）Bin2BCD_tb.v

```
module Bin2BCD_tb;

    reg [7:0] number;

    wire [3:0] bcd0;
    wire [3:0] bcd1;

    Bin2BCD uut (
        .number(number),
        .bcd0(bcd0),
        .bcd1(bcd1)
    );

    initial begin
        number = 0;

        #100;

        // Add stimulus here
        number = 8'd0;      //
        #10;
        number = 8'd1;      //
        #10;
        number = 8'd10;     //
        #10;
```

```
        number = 8'd99;    //
        #10;
        number = 8'd123; //
        #10;
        number = 8'd255; //
```

（2）display_num_tb.v

```
`timescale 1ns / 1ps

module display_num_tb;

    // 输入信号
    reg clk;   // 系统时钟
    reg [7:0] score;   // 分数输入
    reg [3:0] vol_class;   // 音量级别输入

    // 输出信号
    wire [7:0] shift;   // 数码管片选信号
    wire [6:0] oData;   // 数码管显示数据

    // 实例化待测试模块
    display_num uut (
        .clk_1000hz(clk),
        .score(score),
        .vol_class(vol_class),
        .shift(shift),
        .oData(oData)
    );

    // 生成更快的时钟信号
    initial begin
        clk = 0;
        // 假设要生成 4kHz 的时钟，周期为 250us
        forever #5 clk = ~clk; // 每 250 微秒翻转一次，对应 4kHz 的频率
    end

    // 测试用例
    initial begin
        // 初始化输入
        score = 0;
        vol_class = 0;

        // 等待时钟稳定
        #5; // 等待 1ms

        // 提供测试刺激
        score = 8'd59; // 示例分数
        vol_class = 4'd5; // 示例音量级别
        #1000; // 等待几个时钟周期

        score = 8'd123; // 另一个示例分数（超出 BCD 范围，检查显示行为）
        vol_class = 4'd9; // 另一个示例音量级别
        #1000; // 再等待几个时钟周期

    end

Endmodule
```

（3）divider_tb.v

```
`timescale 1ns / 1ps

module divider_tb;

    // Inputs
    reg clk100Mhz;

    // Outputs
    wire clk1000hz;
    wire clk2Mhz;

    // 实例化 divider 模块
    divider uut (
        .clk100Mhz(clk100Mhz),
        .clk1000hz(clk1000hz),
        .clk2Mhz(clk2Mhz)
    );
```

```
    // 初始化并生成 100MHz 的时钟
    initial begin
        clk100Mhz = 0;
        forever #5 clk100Mhz = ~clk100Mhz; // 100MHz 时钟周期为 10ns，所以每 5ns 翻转一次
    end

    // 测试持续时间
    initial begin
        // 测试运行 1ms，足够观察时钟分频行为
        #1000000000000000000;
    end

Endmodule
```

4、音乐数据文件

无代码形式，放在压缩包中。