浙江大学

数字系统设计实验报告

课程名称：	数字系统设计实验
姓　　名：	王若鹏
学　　院：	信息与电子工程学院
专　　业：	电子科学与技术
学　　号：	3170105582
指导教师：	屈民军 唐奕
上课时间：	周四 9,10 节

2018 年 6 月 4 日

专业：电子科学与技术
姓名：王若鹏
学号：3170105582
日期：2018.6.4
地点：东四 223

浙江大学实验报告

课程名称：数字系统设计实验　　指导老师：屈民军 唐奕　　成绩：
实验名称：音乐播放器实验　　实验类型：设计型

一、实验目的

1）掌握音符产生的方法，了解 DDS 技术的应用。
2）了解音频编解码的应用。
3）掌握系统“自顶而下”的数字系统设计方法。

二、实验任务与要求

2.1 实验任务

设计一个音乐播放器：
（1）可以播放四首乐曲，设置 play/pause_button、next_button、reset 三个按键。按下 play/pause_button，音乐在播放和暂停之间切换；按下 next_button 播放下一首乐曲。
（2）LED0 指示播放情况（播放时点亮）、LED2 和 LED3 指示当前乐曲信号。

2.2 实验要求

（1)完成 DDS、mcu、song_reader、note_reader、1000 分频器、同步化电路、次顶层 music_player 等模块的编写。
（2）分别在 ModelSim 中建立工程文件，对各模块进行仿真，并检验波形是否符合要求。
（3）建立 Vivado 工程，生成符合要求的 DCM 内核，添加相关文件，并对工程进行综合、约束、实现等，再下载至开发板中，验证设计结果是否正确。

三、实验主要仪器与设备

装有 Vivado 和 ModelSim 软件的计算机、Nexys Video Artix-7 FPGA 多媒体音视频智能互联开发系统、耳机。

四、实验原理与方案设计

4.1 实验原理

4.1.1 音符产生原理

在简谱中，记录音的高低和长短的符号叫做音符。音符最主要的要素是“音的高低”和“音的长短”。“音的高低”指每个音符的频率不同，“音的长短”用节拍表示，这里的一拍表示一个想对时间度量单位。一拍的长短没有限制，一般来说，一拍的长度可以定义为 1s。

在本实验中，用数组{note，duration}表示音符，note、duration 均为 6 位二进制数。note 为音符记号，作为对应存储器的地址，存储器中存有该音符的相关信息，以查找表的方式实现音符信息的存储与搜索。Duration 则表示该音符的长短。相关音符的对照频率可以参考教材，这里不一一给出。

在本实验中，采取直接数字频率合成（Direct Digital Synthesis）技术生成不同频率的正弦信号，并传送给音频编解码系统，有音频编解码系统将样品转化为电压，驱动扬声器发声。DDS 采用改变寻址的步长来改变输出信号的频率，步长即为对数字波形的相位增量，且正弦频率与相位增量呈线性关系。DDS 的基本原理框图如图 1 所示。

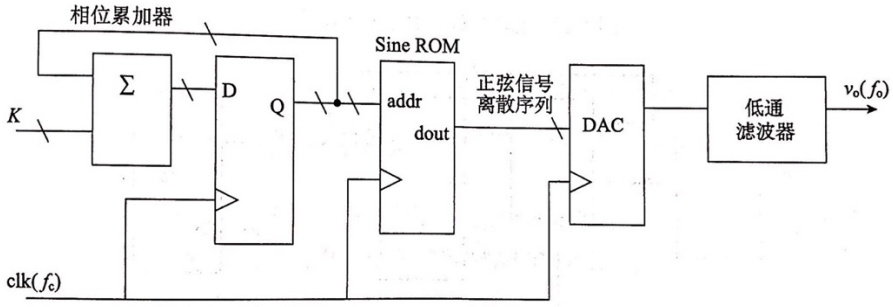

图 1 DDS 原理框图

框图中 Sine ROM 中存放一个完整的正弦信号样品，正弦信号样品根据以下式子的映射关系构成：

$$\mathrm{S(i)} = (2^{n-1} - 1) \times \sin\left(\frac{2\pi i}{2^m}\right) \quad (i = 0, 1, 2 \ldots 2^{m-1})$$

式中，m 为存储器地址线位数，n 为存储器的数据线宽度，S(i)的数据形式为补码。

当 f_c 为取样时钟 clk 的频率，k 为相位增量时，输出正弦信号的频率 f_0 为：

$$f_0 = \frac{k \times f_c}{2^m}$$

从式中可以看出，正弦信号的频率与相位增量成正比关系。在本实验中，由于 m=12，f_c 为 48kHz，f_0 即为对应音符频率，可以得出相位增量 k 与音符频率之间的关系：

$$\mathrm{k} = \frac{f(Hz)}{11.7}$$

计算出各个音符频率后，将 k 的对应数据存储上上面所提到的存储器中。根据 note 查找该音符的相位增量，经过特定的电路实现音符的产生。

在实验 15，详细介绍了 DDS 的原理，经过优化后的原理框图如图 2 所示。

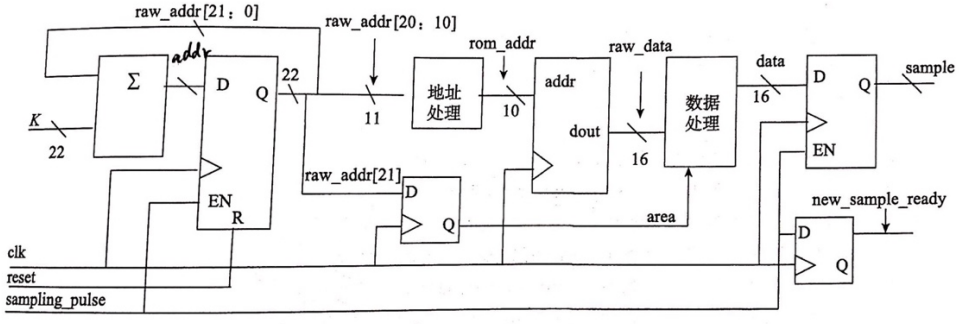

图 2 优化后的 DDS 原理框图

4.1.2 系统顶层

根据实验任务可以将系统划分为 DCM、按键处理、主控制器（mcu）、乐曲读取（song_reader）、音符播放（note_player）、同步化电路、节拍基准产生器和音频编解码接口电路等子模块，如图 3 所示。

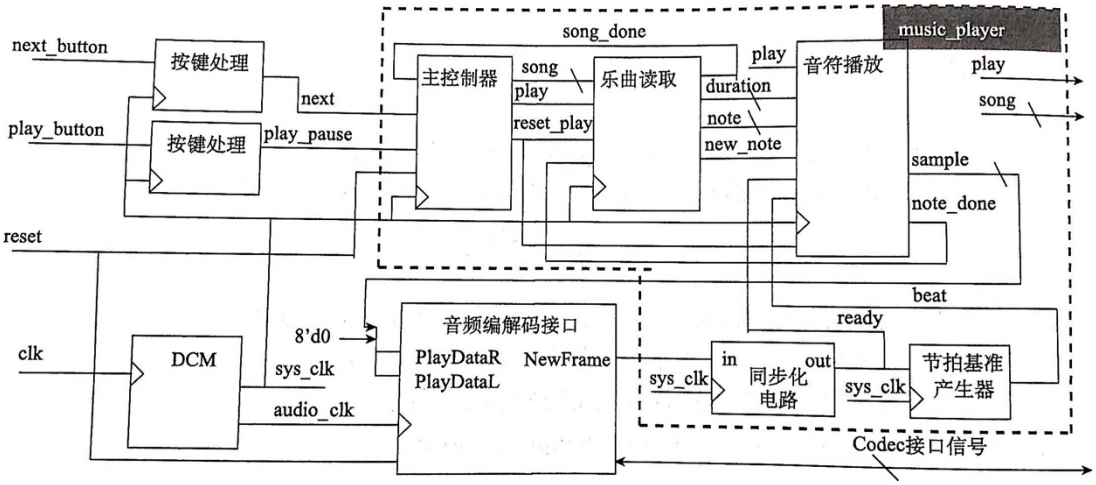

图 3 系统顶层框图

4.1.3 各模块功能

1）时钟管理模块（DCM）：产生 100MHz 的系统时钟和 12.5MHz 的音频时钟。

2）按键处理：由于 next、play 信号为异步输入信号，故需考虑其对系统的影响。于是先由按键处理模块先对异步输入信号进行处理。完成输入同步化、防颤动和脉宽变化等功能。

3）主控制器（mcu）：该模块接收按键信息，通知 song_reader 模块是否要播放音乐以及播放哪首音乐。

4）乐曲读取（song_reader）：该模块根据 mcu 模块的要求，逐个取出音符信息送给音符播放模块播放，当一首乐曲播放完时，向 mcu 模块乐曲发出结束信号。

5）音符播放（note_player）：当该模块接收到需要播放的音符时，在音符持续时间内，以 48kHz 速率送出该音符的正弦波样品给音频解码接口模块。当一个音符播放结束，向乐曲读取模块发送一个 note_done 脉冲索取新的音符。

6）音频解码接口：该模块将正弦波样品转化为串行输出并发送给音频解码芯片。

7）节拍基准发生器（beat_base）：该模块用于产生 48Hz 的节拍定时基准脉冲信号，而 ready 信号的频率为 48kHz，所以节拍基准发生器即为分频比 1000 的分频器。

8）同步化电路（syn_circuit）：由于音频编解码模块与系统使用不同的时钟，因此需要同步化电路协调两部分电路。

9）虚框内即为 music_player 模块，包含以上 3、4、5、7、8 共五个子模块。

4.2 方案设计

4.2.1 DCM 时钟管理模块

本实验输入的时钟频率为 100MHz，要求 DCM 模块产生两种时钟：100MHz 的系统时钟和 12.50MHz 的音频时钟。因此，需要 DCM 锁相频率合成技术实现。可在 Vivado 工程内通过 IP Catalog 命令生成 DCM 内核。

另外，音频编解码接口模块与按键处理模块的代码由指导老师以 BlackBox 方式提供。

4.2.2 按键处理模块

针对异步输入的亚稳态、不确定宽度、开关颤动等问题，设计按键处理模块，包括同步器、脉冲宽度变换电路、开关防颤动电路。按键处理模块原理框图如图 4 所示。原理框图后的附加测试电路则用来测试按键处理模块的功能。各端口说明如表 1 所示。

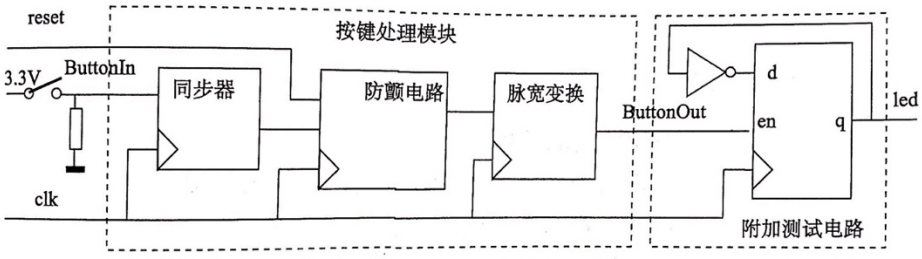

图 4 按键处理原理框图

引脚名称	I/O	引脚说明
clk	Input	100MHz 的系统时钟
ButtonIn	Input	异步信号输入端口
reset	Input	按键处理模块重置信号
ButtonOut	Output	信号输出端口

表 1 按键处理模块端口说明

4.2.3 主控制模块（mcu）

主控制模块有响应按键信息、控制系统播放两大任务。因此其原理框图如图 5 所示。各端口说明如表 2 所示。控制器的算法流程图如图 6 所示。

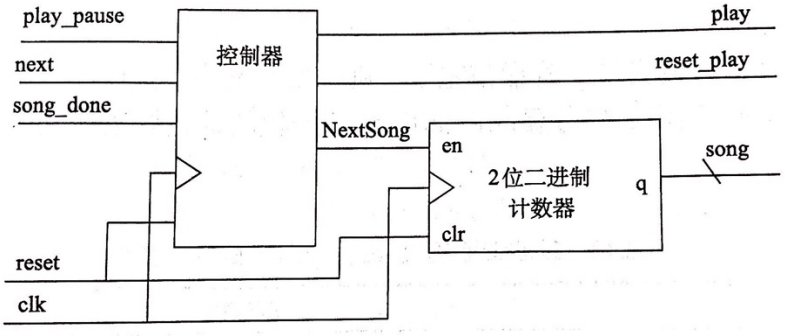

图 5 主控制器的原理框图

引脚名称	I/O	引脚说明
clk	Input	100MHz 时钟信号
reset	Input	复位信号，高电平有效
play_pause	Input	来自按键处理模块的“播放/暂停”控制信号，一个时钟周期宽度的脉冲
next	Input	来自按键处理模块的“下一曲” 控制信号，一个时钟周期宽度的脉冲
play	Output	输出控制信号，高电平表示播放，控制 song_reader 模块是否要播放
reset_play	Output	时钟周期宽度的高电平复位脉冲 reset_play，用于同时复位模块 song_reader 和 note_player
song_done	Input	song_reader 模块的应答信号，一个时钟周期宽度的高电平脉冲，表示一曲播放结束
song[1:0]	Output	当前播放乐曲的序号

表 2 mcu 主控制器端口说明

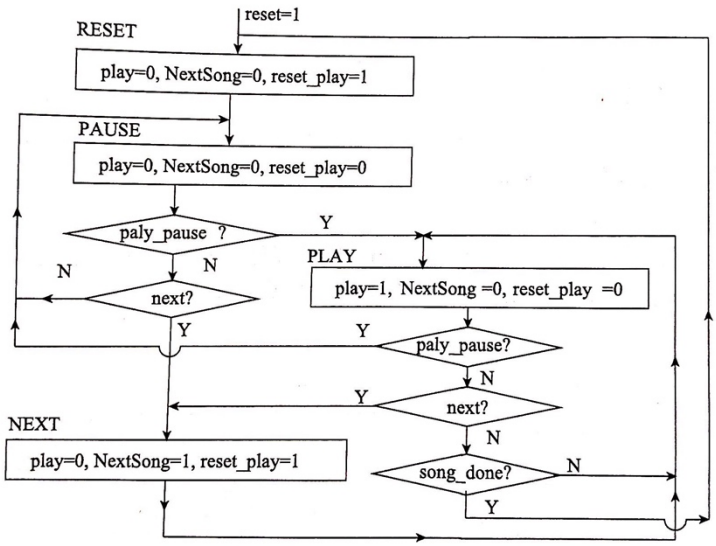

图 6 mcu 算法流程图

mcu 原理图中的 2 位二进制计数器则采用 n=2，counter_bits=2 的计数器，clk 接入系统时钟，en 端接入 NextSong 信号，q 作为输出端口。

4.2.4 乐曲读取模块（song_reader）

乐曲读取模块的任务有：（1）根据 mcu 模块的信号，播放乐曲；（2）响应 note_player 模块的请求，从 song_rom 里逐个取出音符{note，duration}送至 note_player 模块播放；（3）判断乐曲是否播放完毕，若播放完毕，则回复 mcu 模块应答信号。其原理图如图 7 所示，各端口说明如表 3 所示，控制器算法流程图如图 8 所示。

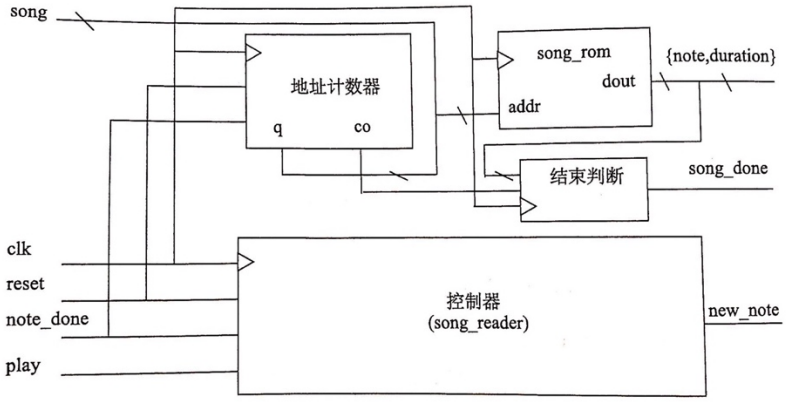

图 7 song_reader 原理框图

引脚名称	I/O	引脚说明
clk	Input	100MHz 时钟信号
reset	Input	复位信号，高电平有效
play	Input	来自 mcu 的控制信号，高电平要求播放
song[1:0]	Input	来自 mcu 的控制信号，当前播放乐曲的序号
note_done	Input	即模块 note_player 的应答信号，一个时钟周期宽度的脉冲，表示一个音符播放结束并索取新音符
song_done	Output	给 mcu 的应答信号，当乐曲播放结束，输出一个时钟周期宽度的脉冲，表示乐曲播放结束
note[5:0]	Output	音符标记
duration[5:0]	Output	音符的持续时间
new_note	Output	给模块 note_playe 的控制信号，一个时钟周期宽度的高电平脉冲，表示新的音符需播放

表 3 song_reader 端口说明

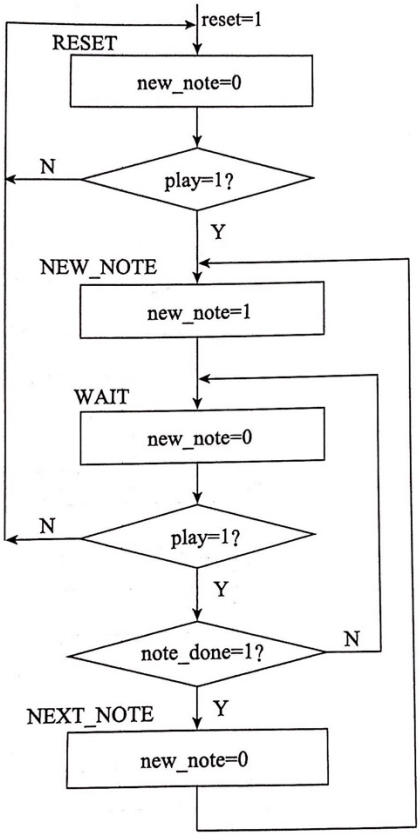

图 8 song_reader 控制器算法流程图

地址计数器为 5 位二进制计数器，其中使能端接入 note_done 信号。计数器的状态 q 为 song_rom 的低 5 位地址，song[1:0]为 song_rom 的高两位地址。

结束判断采用状态机实现。当 duration 为 0 或者地址计数器出现进位时，表示乐曲结束。其算法流程图如图 9 所示。

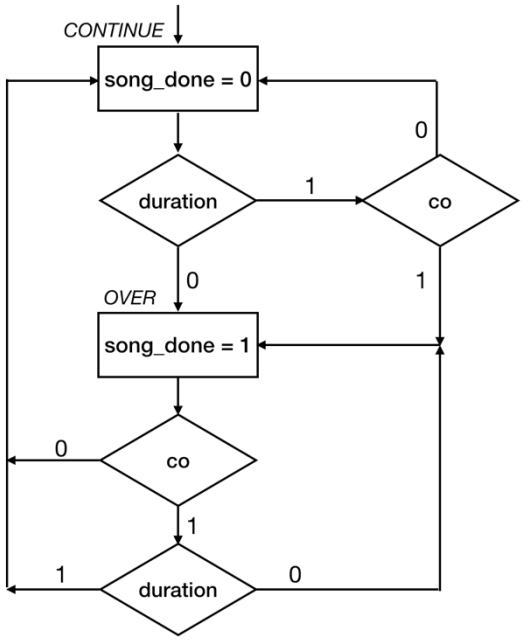

图 9 结束判断控制器算法流程图

4.2.5 音符播放（note_player）模块

音符播放模块的主要任务有：（1）从 song_reader 模块接收所需播放的音符{note, duration}；（2）根据 note 值找出 DDS 的相位增量 k；（3）以 48kHz 的速率从 Sine ROM 中取出正弦样品送给音频解码器接口模块；（4）当一个音符播放完毕，向 song_reader 模块索取新的音符。其原理框图如图 10 所示，端口说明如表 4 所示，控制器算法流程图如图 11 所示。

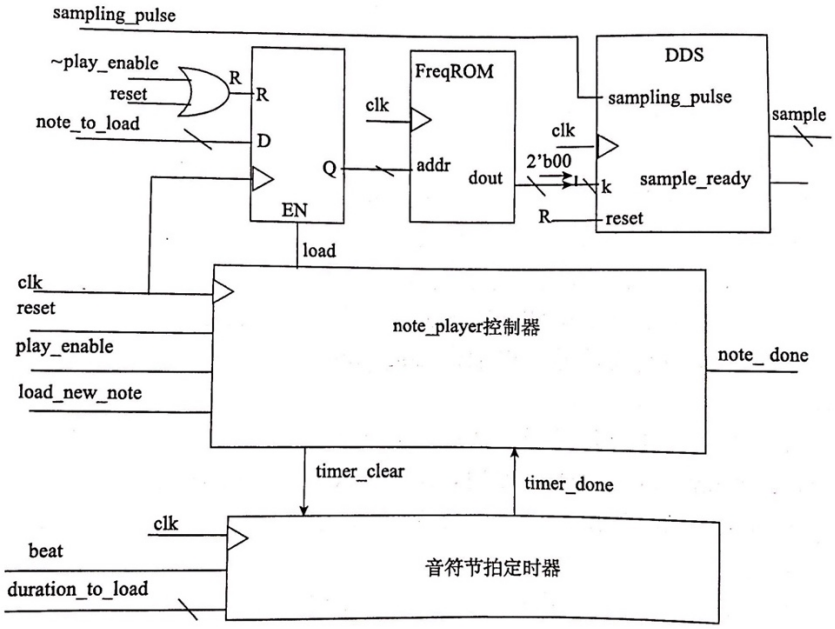

图 10 note_player 原理框图

引脚名称	I/O	引脚说明
clk	Input	系统时钟信号，外接 sys_clk
reset	Input	复位信号，高电平有效，外接 mcu 模块的 reset_play
play_enable	Input	来自 mcu 模块的 play 信号，高电平表示播放
note_to_load[5:0]	Input	来自 song_reader 模块的音符标记 note，表示需播放的音符
duration_to_load[5:0]	Input	来自 song_reader 模块的音符持续时间 duration，表示需播放音符的音长
load_new_note	Input	来自 song_reader 模块的 new_note 信号，一个时钟周期宽度的高电平脉冲，表示新的音符需播放
note _ done	Output	给 song_reader 模块的应答信号，一个时钟周期宽度的高电平脉冲，表示音符播放完毕
sampling_pulse	Input	来自同步化电路模块的 ready 信号，频率 48kHz，一个时钟周期宽度的高电平脉冲，表示索取新的正弦样品
beat	Input	定时基准信号，频率为 48Hz 脉冲，一个时钟周期宽度的高电平脉冲
sample [15:0]	Output	正弦样品输出

表 4 note_player 端口说明

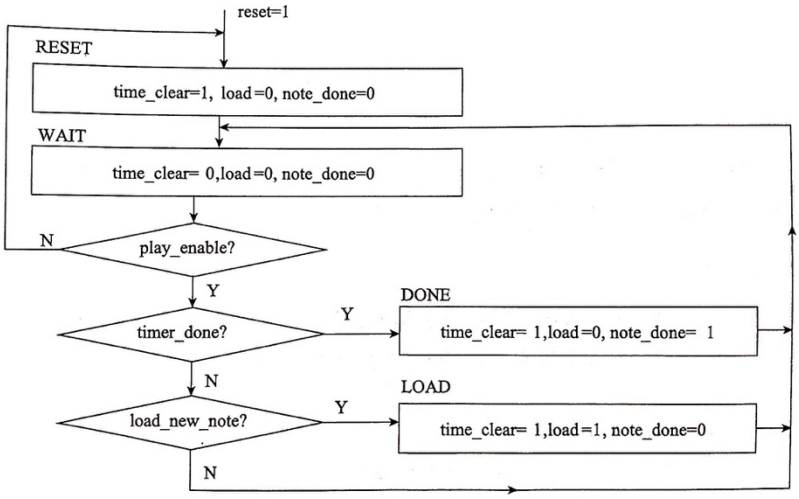

图 11 note_player 控制器算法流程图

节拍定时器为 6 位二进制计数器，beat、timer_clear 分别为使能、清零信号，均为高电平有效。定时时间由吟唱信号 duration_to_load 决定，即 duration_to_load-1 个 beat 周期，timer_done 为定时结束标志。counter_n 为计数器，comp 为比较器，aeb 作为输出端口，当 q 的值与（duration_to_load-1）的值相同时，为高电平。节拍定时器的原理框图如图 12 所示。

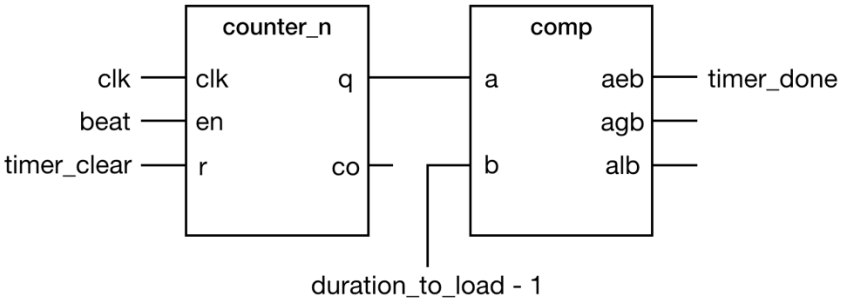

图 12 节拍定时器原理图

4.2.6 同步化电路

由于音频编解码接口模块和其他模块采用不同的时钟，因此两者之间的控制及应答信号须进行同步化处理。本例中音频编解码接口模块的输出信号 NewFrame 的脉冲宽度为一个 audio_clk 时钟周期，需经过同步化处理，产生与 sys_clk 同步且宽度为一个 sys_clk 时钟周期的信号 ready。其电路原理图如图 13 所示，由同步器和脉冲宽度变换电路组成。

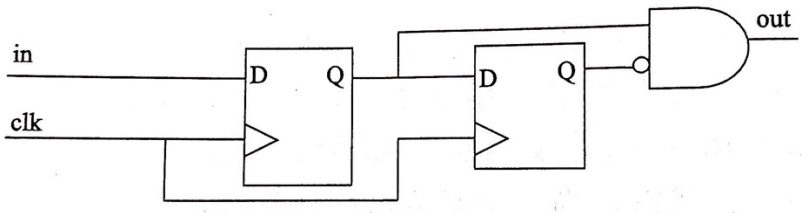

图 13 同步化电路原理图

五、实验步骤

1） 根据实验 15 的原理，编写 DDS 模块的 Verilog HDL 代码，并用 Modelsim 仿真验证。
2） 分别编写 mcu、song_reader、note_player 模块、1000 分频器模块、同步化电路、次顶层 music_player 模块的 Verilog HDL 代码，并用 Modelsim 仿真验证。
3） 新建 vivado 工程文件，生成符合要求的 DCM 内核，添加各个模块到工程中，并对工程进行综合、约束、实现，并下载工程文件到 Nexys Video 开发实验板中。
4） 将耳机接入实验开发板音频输出插座，操作 play/pause(右边按钮)、reset(中间按钮)、next(下面按钮)三个按键，试听耳机中的乐曲并观察实验板上指示灯的变化情况，验证设计结果是否正确。

六、实验记录与结果分析

6.1 代码设计

鉴于整个工程代码过长，且整体设计思路皆已经在第四部分实验原理与方案设计中一一说明，代码所需的工作大部分是将电路原理图以结构型描述风格的代码进行呈现，这部分不再进行详细说明。如有需求，请老师查看源代码文件。

6.1.1 DDS 模块代码设计

该模块由五个文件构成：phase_adder.v、dffre.v、addr_deal、data_deal、sine_rom.v。

phase_adder.v 文件为相位累加器，addr_deal 为地址处理模块、data_deal 为数据处理模块、sine_rom.v 为用 C 程序生成的正弦查找表。设计原理参考图 2，具体代码见附录。

6.1.2 mcu 代码设计

该模块的代码由三个文件构成：mcu.v、control_mcu.v、counter_n.v。

mcu.v 文件调用了 control_mcu.v 与 counter_n.v 两个模块，control_mcu.v 为 mcu 的控制器模块，算法流程图在前面已经详细阐述，在硬件描述语言中，采用二段式进行描述。具体代码见附录。

6.1.3 song_reader 代码设计

该模块由五个文件构成：song_reader.v、reader_control.v、judge_control.v、counter_n.v、song_rom.v。

song_reader.v 调用了其余四个模块，reader_control.v 为该模块的控制器，算法流程图在前面已经详细阐述，在硬件描述语言中，采用二段式进行描述。judge_control.v 则采用状态机的方法判断结束判断。song_rom.v 为寄存器的代码，用来存储 4 首乐曲。counter_n.v 则是地址计数器，用来地址进行计数。具体代码见附录。

6.1.4 note_player 代码设计

该模块由八个文件构成：note_player.v、note_player_control.v、frequency_rom.v、dffre.v、counter_n.v、comp.v、beat_timer.v、dds.v。

Note_player.v 调用了 note_player_control.v、frequency_rom.v、dffre.v、beat_timer.v、dds.v 共五个模块。note_player_control.v 为该模块的控制器，算法流程图在前面已经详细阐述，在硬件描述语言中，采用二段式进行描述。frequency_rom.v 用来存储各音符对应的相位增量 k，以查找表的方式实现。将 k 的值穿给 DDS，以产生对应波形。beat_timer.v 调用了 comp.v 和

counter_n.v 两个模块，用于产生定时结束标志。具体代码见附录。

6.1.5 1000 分频器代码设计

1000 分频器的代码文件为：beat_base.v，由计数器 counter_n.v 的代码进行改进，用作节拍基准发生器。具体代码见附录。

6.1.6 同步化电路代码设计

同步化电路代码文件为：syn_circuit.v，根据电路原理图以结构型 HDL 描述该模块。具体代码见附录。

6.1.7 次顶层 music_player 代码设计

根据系统的顶层框图设计，该次顶层调用了 mcu.v、song_reader.v、note_player.v、syn_circuit.v、beat_base.v 五个模块及它们的相应子模块。具体代码见附录。

6.2 仿真波形分析

6.2.1 DDS 模块仿真

该模块的仿真波形如图 14 所示。经过分析，正弦输出信号 sample 的频率与相位增量 k、取样脉冲 sampling_pulse 的频率符合设计要求。

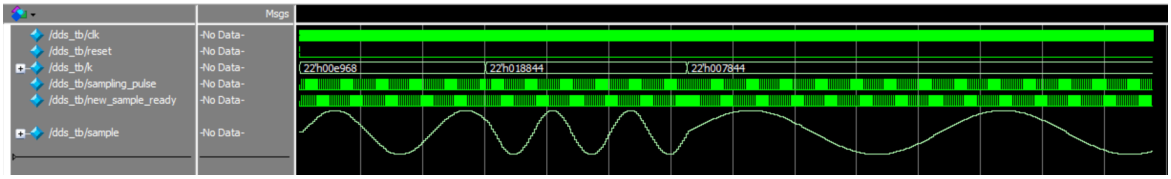

图 14 DDS 模块总体波形

6.2.2 mcu 模块仿真波形分析

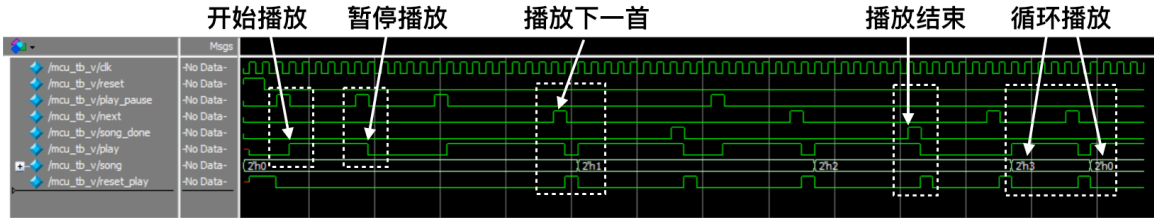

图 15 mcu 模块总体波形

上面五行为 mcu 的输入端口信号，下三行为输出端口信号。各端口信号含义在上文表 4 中已予以说明。

next 为高电平，时钟上升沿时 play 信号变为低电平（表示停止播放）；使 song 信号的变为 1，表示播放第二首乐曲（song 信号由 0 开始计数）；将 reset_play 信号置为高电平，复位其他模块。当 song 信号为 3 时，next 为高电平后，song 信号变为 0。表示当歌曲为第四首时，按下 next 按键，乐曲重新回到第一首进行播放。经检验，mcu 主控制模块的 next 功能正常。

play_pause 为高电平，在时钟的上升沿，使 play 信号翻转，即使乐曲从播放状态变为暂停状态，或使乐曲从暂停状态变为播放状态。经检验，mcu 主控制模块的 play_pause 功能正常。

当乐曲播放完毕时，从 song_reader 模块返回一个高电平 song_done 信号，mcu 产生高

电平 reset_play 信号，并使 play 信号置为低电平。停止播放音乐，并复位其他模块。经检验，mcu 主控制器的播放结束这一功能正常。

经以上所有检验，可知 mcu 主控制模块所有功能正常，设计成功。

6.2.3 song_reader 模块仿真波形分析

song_reader 模块仿真的总体波形如图 16 所示，局部波形如 17 所示。

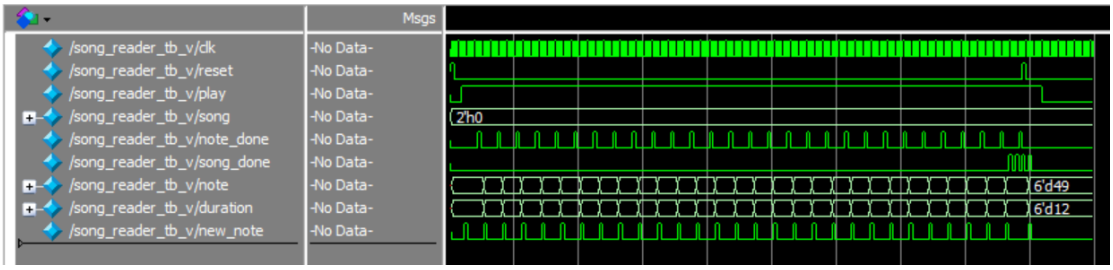

图 16 song_reader 总体波形

上面五行为 song_reader 的输入端口信号，下四行为 song_reader 的输出端口信号。各端口的含义在上文表 5 中已予以说明。

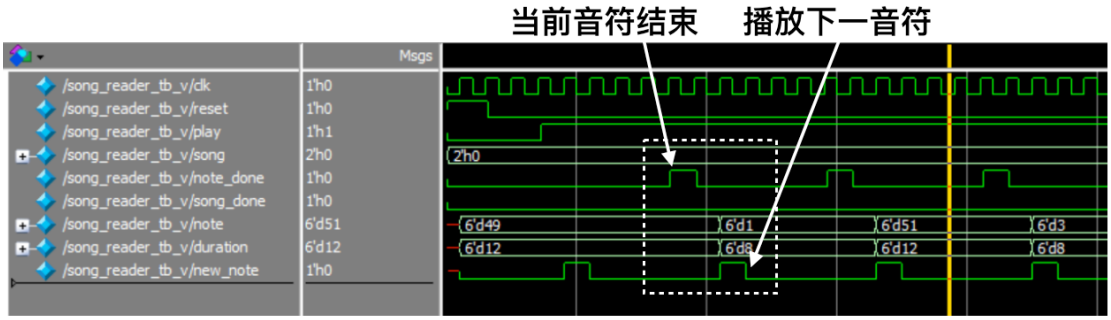

图 17 音符控制波形

当 note_player 模块向 song_reader 模块索取新的音符（即 note_done 为高电平）时，song_reader 的 note 与 duration 输出端口给出下一音符的信息，并发送 new_note 信号（即 new_note 为高电平），表示有新的音符需要播放。经检验，note_player 模块音符控制功能正常。

当曲子结束时，song_reader 模块输出乐曲播放结束信号（即 note_done 信号为高电平），且该信号的脉冲宽度为 1 个时钟周期。经检验，该功能符合设计要求，song_reader 模块指示乐曲播放结束功能正常。

经以上检验，song_reader 模块所有功能正常，设计成功。

6.2.4 note_reader 模块仿真波形分析

note_player 模块仿真的总体波形如图 18 所示。当 play_enable 为低电平时，note_reader 模块的输出信号 sample 为 0。即 note_reader 模块的停止播放功能正常。

如图 18 所示，当前 duration_to_load 信号为 7，收到 7 个 beat 信号后，note_player 模块输出一个 note_done 高电平信号。表示当前音符播放完毕。向 song_reader 模块索取下一音符。同时，从 sample 信号中可以看出，正弦波的频率也发生了变化。

经以上检验，note_player 模块功能正常。

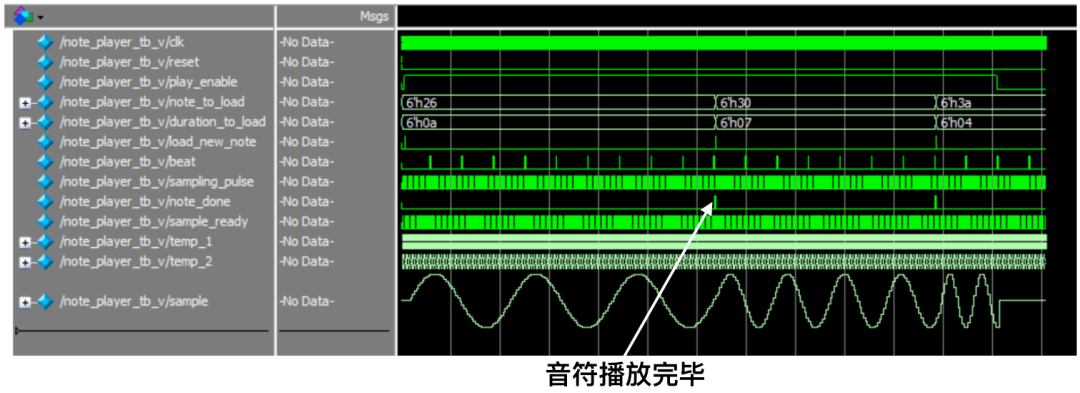

图 18 note_reader 总体波形

6.2.5 1000 分频器模块仿真分析

节拍基准发生器即 1000 分频器模块的仿真如图 19 所示，计数到 1000 时，co 便会输出一个进位，同时计数器归零，符合设计要求。

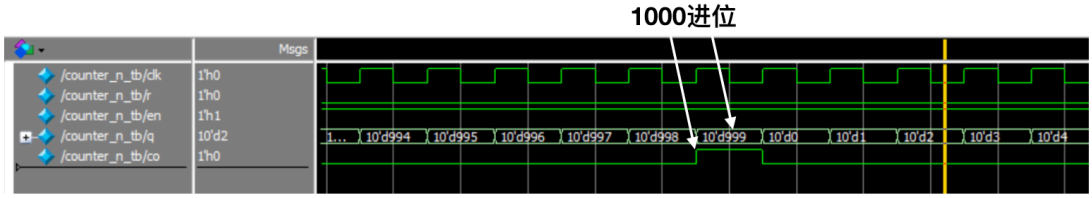

图 19 counter_1000 总体波形

6.2.6 同步化电路仿真分析

同步化电路模块 syn_circuit 仿真波形如图 20 所示。可见，输入的被同步信号 in 经过同步化处理，输出信号 out 的宽度为 clk 的一个时钟周期，符合设计要求。

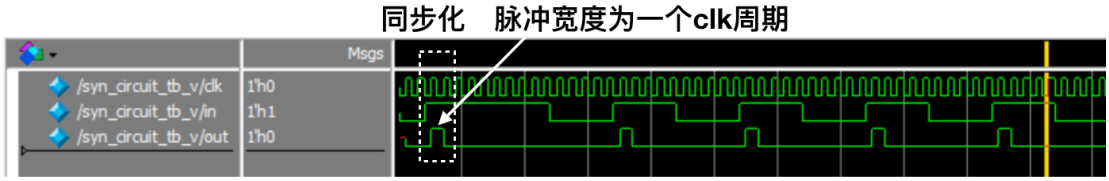

图 20 同步化电路总体波形

6.2.7 次顶层 music_player 模块波形分析

music_player 模块仿真的波形如图 21 所示。

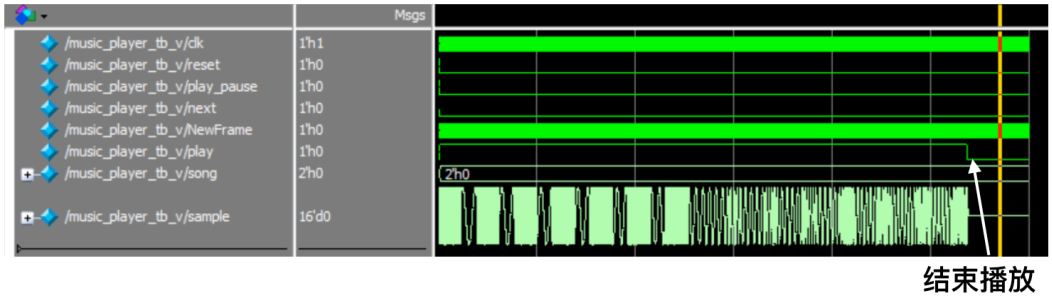

图 21 music_player 模块波形

可以看出，该模块的 sample 信号可以产生不同频率的正弦波，并在 play 为低电平后，停止产生正弦波。模块总体功能正常。

6.3 开发板验证

新建 Vivado 工程，按照教材上的要求，对工程进行综合、约束、实现，并将其下载到 Nexys Video 开发板上后，验证设计结果。一切功能正常，已由指导老师验收。

七、思考题与心得

7.1 思考题

（1）在实验中，为什么 next_button、play_pause_button 两个按键需要消颤动及同步化处理，而 reset 按键不需要消颤动及同步化处理？

答：因为 next_button 不经过处理，则有可能按一次按键，却产生多次高电平信号，无法使歌曲准确调至下一首进行播放。play_pause_button 也是同理。而 reset 若产生多次高电平信号，最终的结果也同样是对模块进行复位，没有差别。为了简化整体模块，故可以不采用消颤动及同步化处理。

（2）在主控制器（mcu）设计中，是否存在接受不到按键信息？若存在，概率多大？有没有必要修改设计？

答：当状态机从 NEXT 状态转到 PLAY 状态或者是从 RESET 状态转到 PAUSE 状态时，这一段时间内接受不到按键信息。但由于系统时钟周期极短，这一段时间也极短。概率非常小，几乎为 0。所以没有必要修改设计。

7.2 心得

时间如白驹过隙，夏学期不知不觉就要过去了。经过短短七周时间，我已经从一个只写过组合逻辑电路的小白，成长为一个能写音乐播放器这类工程的人了。这一个个.v 文件，一幅幅波形图，见证着我的蜕变……

数字系统设计实验这门课与其他课有所不同，需要课下自己学习很多东西，不能依赖于老师，这对我们学习能力的锻炼是十分有益的，也让我对 HDL 有了基本的认识并掌握了初级的编程方式。尤其是在调试阶段，Verilog 程序的调试与此前习惯的 C/C++程序调试方式不同，需要观察编译记录、结合波形输出对代码和模块进行联调，非常考验耐心和细心的程度，同时也让我初步掌握了这类工程调试的方法。每每遇到问题，很幸运能得到老师的讲解与同学的帮助，让我有了不小的进步，收获颇丰。

经过本次的工程训练，给予我启发最大的是：对于一项工程，最重要的就是要把它分成若干模块，每个模块可以互相调用，也可以调用更小的子模块。这样子整个项目的结构就会非常清晰，debug 时一个个模块一个个信号分析下来省时省力，调试起来会方便许多。

印象最深的，便是最后的 music_player 次顶层模块仿真时，得到的结果总是曲子播放提前结束。我花了很长时间，仔细检查了各个模块的控制器和算法，可是并没有发现任何问题。最后在老师的协助下，发现是 1000 分频器的编写出现了逻辑错误，修改过后，仿真一切正常。

虽然经历了很多的坎坷，但是当最后在开发板上进行验证的时候，音乐响起的那一刻，我的心里还是十分激动的，不论过程怎样艰辛，自己历时两个月的努力终于有了回报，一切努力都是值得的。现在，我觉得自己的工程能力有了一些提高，非常感谢两位指导老师，在我遇到困难时热心为我答疑解惑！谢谢你们的耐心与鼓励！

附录：部分关键模块代码

dds.v（DDS 模块）

```
module dds(clk,reset,k,sampling_pulse,new_sample_ready,sample);
    input clk;
    input reset;
    input sampling_pulse;
    input [21:0] k;
    wire [21:0] addr;
    wire [21:0] raw_addr;
    wire [9:0] rom_addr;
    wire [15:0] raw_data;
    wire [15:0] data;
    wire area;
    output [15:0] sample;
    output new_sample_ready;
    phase_adder #(22) sum(
        .a(raw_addr[21:0]),
        .b(k[21:0]),
        .s(addr),
        .ci(0),
        .co());
    dffre #(.n(22)) dffre_addre(
        .d(addr),
        .en(sampling_pulse),
        .r(reset),
        .clk(clk),
        .q(raw_addr));
    dffre #(.n(1)) area_add(
        .d(raw_addr[21]),
        .en(1),
        .r(0),
        .clk(clk),
        .q(area));
    addr_deal addr_deal(
        .raw_addr(raw_addr[20:10]),
        .rom_addr(rom_addr));
    sine_rom sine_rom(
        .clk(clk),
        .addr(rom_addr),
        .dout(raw_data));
    data_deal #(.N(16)) data_deal(
        .area(area),
        .raw_data(raw_data),
```

```
        .data(data));
    dffre #(.n(16)) data_adde(
        .d(data),
        .en(sampling_pulse),
        .r(0),
        .clk(clk),
        .q(sample));
    dffre #(.n(1)) ready_add(
        .d(sampling_pulse),
        .en(1),
        .r(0),
        .clk(clk),
        .q(new_sample_ready));
endmodule
```

mcu.v（主控制模块）

```
module mcu(clk,reset,play_pause,next,play,song,reset_play,song_done);
    input play_pause,clk,reset,next,song_done;
    output play,reset_play;
    output [1:0] song;
    wire nextsong;
    control_mcu control(
        .reset(reset),
        .clk(clk),
        .play_pause(play_pause),
        .next(next),
        .song_done(song_done),
        .nextsong(nextsong),
        .reset_play(reset_play),
        .play(play)
    );
    counter_n #(.n(4), .counter_bits(2)) counter_n(
        .clk(clk),
        .en(nextsong),
        .r(reset),
        .q(song),
        .co()
    );
endmodule
```

control_mcu.v（主控制模块控制器）

```
module
control_mcu(reset,clk,play_pause,next,song_done,nextsong,reset_play,play);
```

```
input reset,clk,song_done,next,play_pause;
output reg nextsong,reset_play,play;
parameter RESET=0,PAUSE=1,PLAY=2,NEXT=3;
reg[1:0] state,nextstate;
always@(posedge clk)
    if(reset) state=RESET; else state=nextstate;
always@(*)
    case(state)
        RESET:
        begin
        play=0;
        nextsong=0;
        reset_play=1;
        nextstate=PAUSE;
        end

        PAUSE:
        begin
        play=0;
        nextsong=0;
        reset_play=0;
        if(play_pause) nextstate=PLAY;
        else if(next) nextstate=NEXT;
        else nextstate=PAUSE;
        end

        PLAY:
        begin
        play=1;
        nextsong=0;
        reset_play=0;
        if(play_pause) nextstate=PAUSE;
        else if(next) nextstate=NEXT;
        else if(song_done) nextstate=RESET;
        else nextstate=PLAY;
        end

        NEXT:
        begin
        play=0;
        nextsong=1;
        reset_play=1;
        nextstate=PLAY;
        end
```

```
        endcase
endmodule
```

song_reader.v（乐曲读取模块）

```
module
song_reader(clk,reset,play,song,song_done,note,duration,new_note,note_done);
    input clk,reset,play,note_done;
    input [1:0] song;
    output song_done,new_note;
    output[5:0] note, duration;
    wire[4:0] low_rom;
    wire co;
    counter_n #(.n(32),.counter_bits(5)) addr_counter(
        .clk(clk),
        .en(note_done),
        .r(reset),
        .q(low_rom),
        .co(co)
    );
    song_rom song_rom(
        .clk(clk),
        .dout({note,duration}),
        .addr({song,low_rom})
    );
    reader_control song_reader_control(
        .clk(clk),
        .reset(reset),
        .note_done(note_done),
        .play(play),
        .new_note(new_note)
    );
    judge_control judge(
        .clk(clk),
        .in_duration(duration),
        .in_co(co),
        .song_done(song_done)
    );
endmodule
```

reader_control.v（乐曲读取控制器）

```
module reader_control(clk,reset,note_done,play,new_note);
    input clk,reset,note_done,play;
    output reg new_note;
    reg[1:0]state,nextstate;
```

```
    parameter RESET=0,NEW_NOTE=1,WAIT=2,NEXT_NOTE=3;
    always@(posedge clk)
        if(reset) state=RESET; else state=nextstate;
    always@(*)
        case(state)
            RESET:
            begin
            new_note=0;
            if(play) nextstate=NEW_NOTE;
            else nextstate=RESET;
            end

            NEW_NOTE:
            begin
            new_note=1;
            nextstate=WAIT;
            end

            WAIT:
            begin
            new_note=0;
            if(!play) nextstate=RESET;
            else if(note_done) nextstate=NEXT_NOTE;
            else nextstate=WAIT;
            end

            NEXT_NOTE:
            begin
            new_note=0;
            nextstate=NEW_NOTE;
            end
        endcase
endmodule
```

judge_control.v（结束判断）

```
module judge_control(clk,in_duration,in_co,song_done);
    input clk,in_co;
    input [5:0] in_duration;
    output reg song_done;
    reg state,nextstate;
    parameter CONTINUE=0,OVER=1;
    initial state=CONTINUE;
    always @(posedge clk)
        state=nextstate;
```

```
    always @(*)
        case(state)
            CONTINUE:
            begin
            song_done=0;
            if(in_duration==6'b000000) nextstate=OVER;
            else if(in_co) nextstate=OVER;
            else nextstate=CONTINUE;
            end

            OVER:
            begin
            song_done=1;
            if(~in_co) nextstate=CONTINUE;
            else if(in_duration==6'b000000) nextstate=OVER;
            else nextstate=CONTINUE;
            end
        endcase
endmodule
```

note_player.v（音符播放模块）

```
module
note_player(clk,reset,play_enable,note_to_load,duration_to_load,load_new_not
e,note_done,sampling_pulse,beat,sample);
    input clk,reset,play_enable,load_new_note,sampling_pulse,beat;
    input [5:0] note_to_load,duration_to_load;
    output note_done;
    output[15:0]sample;
    wire timer_clear_wire,timer_done_wire,load_wire;
    note_player_control player_control(
        .clk(clk),
        .reset(reset),
        .play_enable(play_enable),
        .load_new_note(load_new_note),
        .timer_clear(timer_clear_wire),
        .timer_done(timer_done_wire),
        .note_done(note_done),
        .load(load_wire)
    );
    beat_timer beat_timer(
        .clk(clk),
        .beat(beat),
        .duration_to_load(duration_to_load),
        .timer_clear(timer_clear_wire),
```

```
        .timer_done(timer_done_wire)
    );
    wire rin;
    wire [5:0] qout;
    assign rin=reset||(~play_enable);
    dffre #(.n(6)) dffre_con(
        .r(rin),
        .d(note_to_load),
        .en(load_wire),
        .clk(clk),
        .q(qout)
    );
    wire[19:0] dout_wire;
    frequency_rom FreqROM(
        .clk(clk),
        .dout(dout_wire),
        .addr(qout)
    );
    dds DDS(
        .clk(clk),
        .reset(rin),
        .k({2'b00,dout_wire}),
        .sampling_pulse(sampling_pulse),
        .sample(sample),
        .new_sample_ready()
    );
endmodule
```

note_player_control.v（音符播放控制器）

```
module
   note_player_control(clk,reset,play_enable,load_new_note,timer_clear,timer
   _done,note_done,load);
    input clk,reset,play_enable,load_new_note,timer_done;
    output reg note_done,timer_clear,load;
    reg[1:0]state,nextstate;
    parameter RESET=0,PLAY=1,DONE=2,LOAD=3;
    always @(posedge clk)
        if(reset) state=RESET; else state=nextstate;
    always @(*)
        case(state)
            RESET:
            begin
            timer_clear=1;
            load=0;
```

```
            note_done=0;
            nextstate=PLAY;
            end

            PLAY:
            begin
            timer_clear=0;
            load=0;
            note_done=0;
            if(~play_enable) nextstate=RESET;
            else if(timer_done) nextstate=DONE;
            else if(load_new_note) nextstate=LOAD;
            else nextstate=PLAY;
            end

            DONE:
            begin
            timer_clear=1;
            load=0;
            note_done=1;
            nextstate=PLAY;
            end

            LOAD:
            begin
            timer_clear=1;
            load=1;
            note_done=0;
            nextstate=PLAY;
            end
        endcase
endmodule
```

syn_circuit.v（同步化电路）

```
module syn_circuit(sys_clk,in,out);
    input sys_clk, in;
    output out;
    wire q1,q2;
    dffre #(.n(1)) dff1(
    .d(in),.en(1),.r(0),.clk(sys_clk),.q(q1));
    dffre #(.n(1)) dff2(
    .d(q1),.en(1),.r(0),.clk(sys_clk),.q(q2));
    assign
        out=q1&&(~q2);
```

```
endmodule
```

beat_base.v（节拍基准产生器）

```
module beat_base(ready,sys_clk,beat);
    parameter n=1000;
    parameter counter_bits=10;
    input sys_clk;
    input ready;
    output  beat;
    reg [counter_bits-1:0] q=0;
    assign beat=ready&&(q==(n-1));
    always@(posedge sys_clk)
    begin
        if(ready)
            begin
                if(q==(n-1)) q=0;
                else q=q+1;
            end
    end
endmodule
```

music_player.v（次顶层模块）

```
module music_player #(parameter sim=0)(
    input clk,
    input reset,
    input play_pause,
    input next,
    input NewFrame,
    output [15:0] sample,
    output play,
    output [1:0] song);
    wire reset_play;
    wire song_done;
    mcu MCU(
        .clk(clk),
        .reset(reset),
        .play_pause(play_pause),
        .next(next),
        .play(play),
        .song(song),
        .reset_play(reset_play),
        .song_done(song_done));
    wire ready;
    syn_circuit SYN(
```

```
        .sys_clk(clk),
        .in(NewFrame),
        .out(ready));
    wire beat;
    beat_base BB(
        .ready(ready),
        .sys_clk(clk),
        .beat(beat));
    wire [5:0] note;
    wire [5:0] duration;
    wire new_note;
    wire note_done;
    song_reader SR(
        .clk(clk),
        .reset(reset_play),
        .play(play),
        .song(song),
        .song_done(song_done),
        .note(note),
        .duration(duration),
        .new_note(new_note),
        .note_done(note_done));
    note_player NP(
        .clk(clk),
        .reset(reset_play),
        .play_enable(play),
        .note_to_load(note),
        .duration_to_load(duration),
        .load_new_note(new_note),
        .note_done(note_done),
        .sampling_pulse(ready),
        .beat(beat),
        .sample(sample));
endmodule
```