浙江大学

本科实验报告

课程名称：	数字系统设计实验
姓　　名：	黄嘉欣
学　　院：	信息与电子工程学院
系：	信息与电子工程学系
专　　业：	信息工程
学　　号：	3190102060
指导教师：	屈民军、唐奕

2021 年 6 月 8 日

专业：信息工程
姓名：黄嘉欣
学号：3190102060
日期：2021 年 6 月 8 日
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：音乐播放实验　实验类型：设计性实验

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

1、设计并仿真一个 DDS 正弦信号发生器，要求：

① 采样频率 $f_c = 48\text{kHz}$；

② 正弦信号频率范围为 20Hz～20kHz；

③ 正弦信号序列宽度 16 位，包括一位符号；

2、设计一个音乐播放器，要求：

① 可以播放四首乐曲，设置 play/pause_button、next_button、reset 三个按键。按 play/pause_button 键，音乐在播放和暂停之间切换；按 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 指示当前乐曲序号。

三、实验原理

1、DDS 的基本原理：

为了在数字域产生正弦信号，可以用一个存储器（ROM/RAM）存储一张正弦表；然后将存于表中的正弦样品取出，经数模转换器 D/A，形成模拟量波形。若要实时改变输出信号的频率，可以通过改变查表寻址的频率或寻址的步长来实现。DDS 技术即采用了后面这种方法，步长为对数字波形查表的相位增量，输出正弦频率与相位增量成线

性关系，其基本原理框图如图 3.1.1 所示，由相位累加器、正弦查询表（Sine ROM）、D/A 转换器和低通滤波器组成：

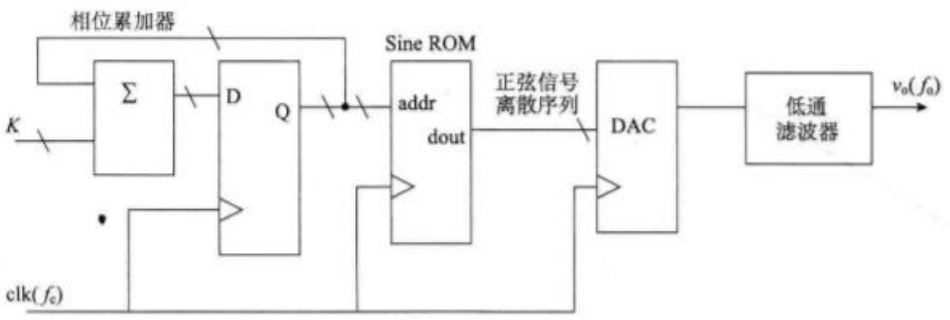

图 3.1.1 DDS 原理框图

正弦查询表（Sine ROM）中存放有一个完整的正弦信号样品，其由如下的映射关系构成：

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

式中，m 为 Sine ROM 地址线位数，n 为 ROM 数据线宽度，$S(i)$的数据形式为补码。

f_c 为取样时钟 clk 的频率，K 为相位增量（也称频率控制字），输出正弦信号的频率 f_o 由 f_c 和 K 共同决定，即：

$$f_o=\frac{K\times f_c}{2^m}$$

可以看出，正弦信号的频率 f_o与相位增量 K 呈正比关系。相位累加器的位数由 m 位整数和 p 位小数组成，其高 m 位整数部分作为 Sine ROM 的地址。

在实际应用当中，DDS 的最高输出频率由允许输出的杂散电平决定，一般取值为 $f_o \le 40\%\times f_c$，因此 K 的最大值一般为$40\%\times 2^{m-1}$。除此之外，DDS 可以很容易实现正弦信号与余弦信号正交两路输出，只需用相位累加器的输出同时驱动固化有正弦信号波形的 Sine ROM 和余弦信号波形的 Cos ROM，并各自经数模转换器和低通滤波器输出即可。另外，DDS 也很容易实现调幅和调频，其原理框图分别如图 3.1.2 和图 3.1.3 所示：

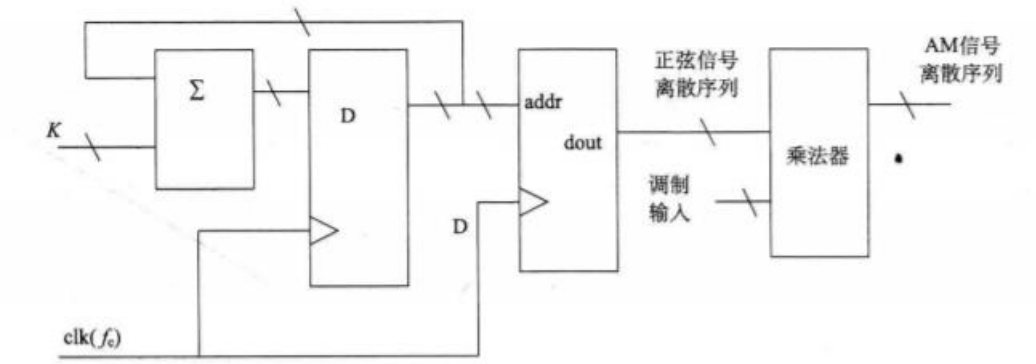

图 3.1.2 DDS 实现调幅原理框图

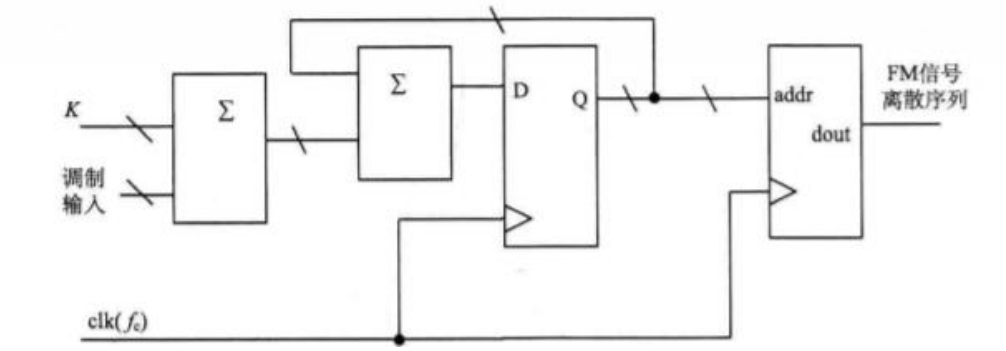

图 3.1.3 DDS 实现调频原理框图

2、DDS 实验设计原理：

根据 DDS 输出信号的最低频率要求，有：

$$f_o(\min)=\frac{f_c}{2^m}\leqslant 20\text{Hz}$$

代入 $f_c=48\text{KHz}$，可得 $m=12$，但为了得到更准确的正弦信号频率，相位累加器位数会增加 10 位小数。所以，相位累加器为 22 位累加器，其高 12 位为 Sine ROM 的地址。

因为 $m=12$，所以存储一个完整周期的正弦信号样品就需要 $2^{12}\times 16$ bits 的 ROM。但由于正弦波形的对称性，如图 3.2.1 所示，可以将正弦波形分为四个区域，如此只需要在 Sine ROM 中存储四分之一的正弦信号样品（0 区）即可。此时 Sine ROM 的容量可减少为$2^{10}\times 16$ bits，即 10 位地址，存储四分之一的正弦信号样品（共 1024 个）。需要注意的是，四分之一周期的正弦信号样品未给出 90°的样品值，因此在 ROM 地址为 1024（即 90°）时可取地址为 1023 的值（实际上地址为 1023 时，正弦信号样品已达最大值）。

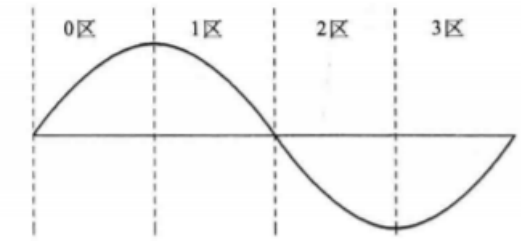

图 3.2.1 正弦信号波形

经过上述优化后，DDS 的结构也必须进行必要处理，其优化后的结构如图所示：

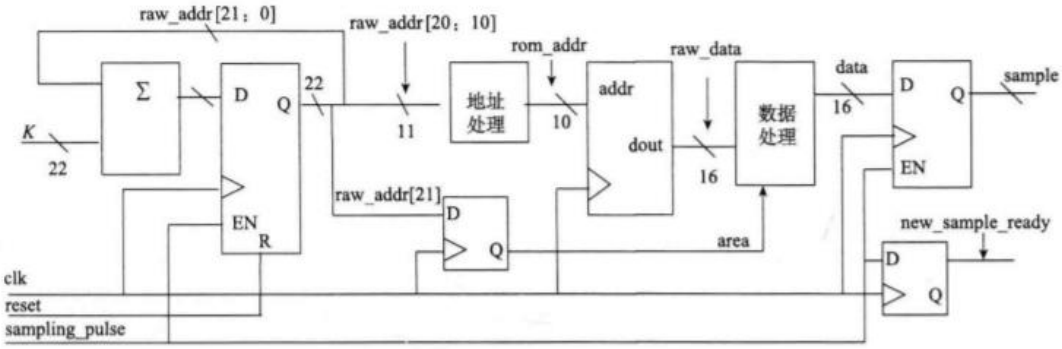

图 3.2.2 优化后的 DDS 结构

图中，sampling_pulse 为采样脉冲，一般情况下，其频率应低于时钟 clk 频率，而宽度必须为一个时钟 clk 周期。若采样脉冲与时钟信号为同一个信号，可将 sampling_pulse 接高电平即可。相位累加得到的 22 位原始地址 raw_addr[21:0]，整数部分 raw_addr[21:10]即为完整周期正弦信号样品的地址，其中高两位地址 raw_addr[21:20]可区分正弦的四个区域。由于 sine_rom 只保存了四分之一周期的 1024 个样品，所以 raw_addr[21:10]不能直接作为 sine_rom 地址，必须进行必要处理，其处理方法如表 3.2 所示：

表 3.2 sine_rom 的地址和数据处理方法

正弦区域	Sine ROM 地址	正弦样品 data	备注
0	raw_addr[19:10]	raw_data[15:0]	
1	当 raw_addr[20:10]=1024 时，rom_addr 取 1023，其他情况取~raw_addr[19:10]+1	raw_data[15:0]	地址镜像翻转
2	raw_addr[19:10]	~raw_data[15:0]+1	数据取反
3	当 raw_addr[20:10]=1024 时，rom_addr 取 1023，其他情况取~raw_addr[19:10]+1	~raw_data[15:0]+1	地址镜像翻转 数据取反

3、音乐播放器的顶层设计：

根据实验任务可将系统划分为时钟管理模块（DCM）、按键处理、主控制器、乐曲读取、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 3.3.1 所示。

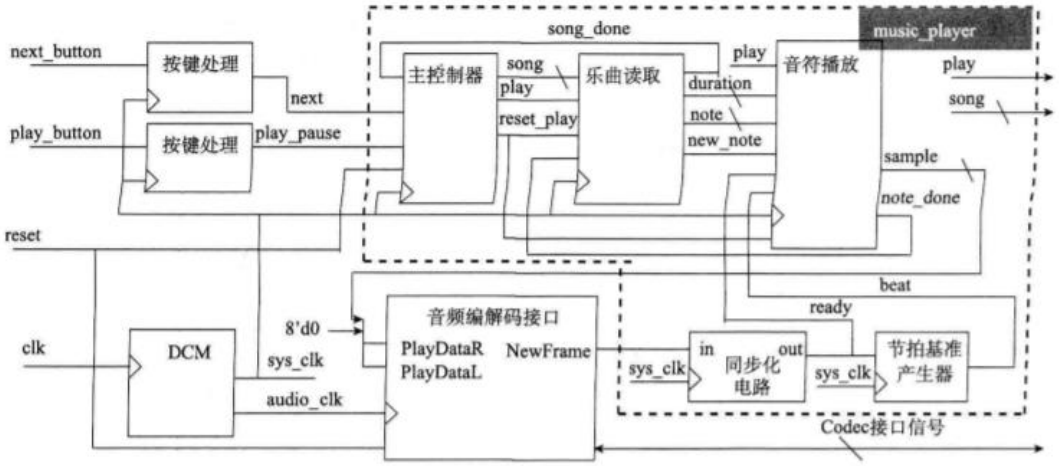

图 3.3.1 音乐播放器的顶层设计

各主要子模块作用如下：

① 时钟管理模块（DCM）产生 100MHz 的系统时钟 sys_clk 和 12.5MHz 的音频时钟 audio_clk；

② 主控制器（mcu）模块接收按键信息，通知 song_reader 模块是否要播放（play）及播放哪首乐曲（song）；

③ 乐曲读取（song_reader）模块根据 mcu 模块的要求，逐个取出音符信息{note, duration}送给 note_player 模块播放，当一首乐曲播放完毕，回复 mcu 模块乐曲播放结束信号（song_done）；

④ 音符播放接收到需播放的音符，在音符的持续时间内，以 48kHz 速率送出该音符的正弦波样品给音频编解码接口模块。当一个音符播放结束，向 song_reader 模块发送一个 note_done 脉冲索取新的音符；

⑤ 音频编解码接口模块负责将音符的正弦波样品转换为串行输出并发送给音频编解码芯片 ADAU1761。音频编解码芯片 ADAU1761 接收正弦波样品，再进行 AD 转换并放大，最后送至扬声器播放。注意，note_player 模块产生的正弦波样品为 16 位二进制，需在低位加 8 个 0 后送入音频编解码接口模块；

⑥ 由于音频编解码模块与系统使用不同时钟，因此需要同步化电路协调两部分电路；

⑦ 节拍基准产生器产生 48Hz 的节拍定时基准脉冲信号（beat），而 ready 信号频率为 48kHz，因此，节拍基准产生器即为分频比为 1000 的分频器；

⑧ 按键处理模块完成输入同步化、防颤动和脉宽变换等功能。

4、音乐播放器设计原理：

① 主控制模块 mcu 的设计

主控制模块 mcu 有响应按键信息、控制系统播放两大任务，下表为其端口含义：

表 3.4.1 主控制模块 mcu 的端口含义

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

根据设计要求，模块 mcu 的原理框图如图 3.4.1.1 所示。图中的 2 位二进制计数器用来计算乐曲序号（song）：

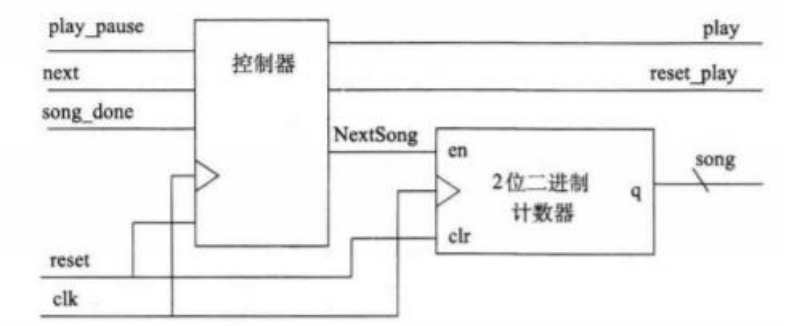

图 3.4.1.1 mcu 的结构框图

其控制器的工作流程图如图 3.4.1.2 所示：

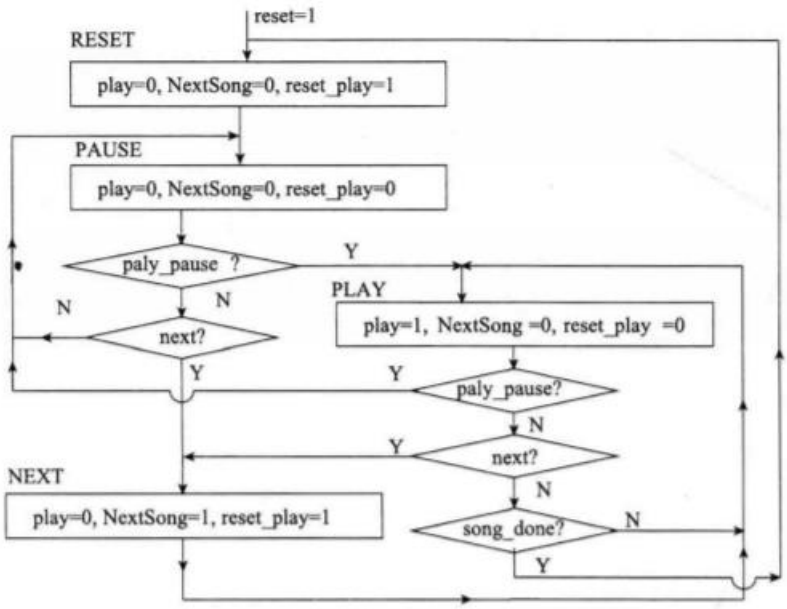

图 3.4.1.2 mcu 控制器的算法流程图

② 乐曲读取模块 song_reader 的设计

乐曲读取模块 song_reader 的任务有：

（1）根据 mcu 模块的要求，选择播放乐曲；

（2）响应 note_player 模块请求，从 song_rom 中逐个取出音符{note, duration}送给 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。

根据 song_reader 模块的任务要求，其需包含表 3.4.2 所示的输入、输出端口：

表 3.4.2 乐曲读取模块 song_reader 的端口含义

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

song_rom 是一个只读存储器，用来存放乐曲，容量为 $2^7 \times 12$ bits。其中共存放有四首乐曲，每首乐曲占用 $2^5 \times 12$ bits 空间，即每首乐曲最长由 32 个音符组成。因此，song_rom 高2位地址决定哪首乐曲，而低5位地址决定这首乐曲的哪个音符。song_rom

每个地址存放一个音符信息，音符信息由 12 位二进制组成，高 6 位表示音符标记 note，低 6 位表示音长 duration。

根据 song_reader 模块的功能及 song_rom 结构，可画出图 3.4.2.1 所示的结构框图，控制器主要负责接收 mcu 模块与 note_player 模块的控制信号，并做出响应。

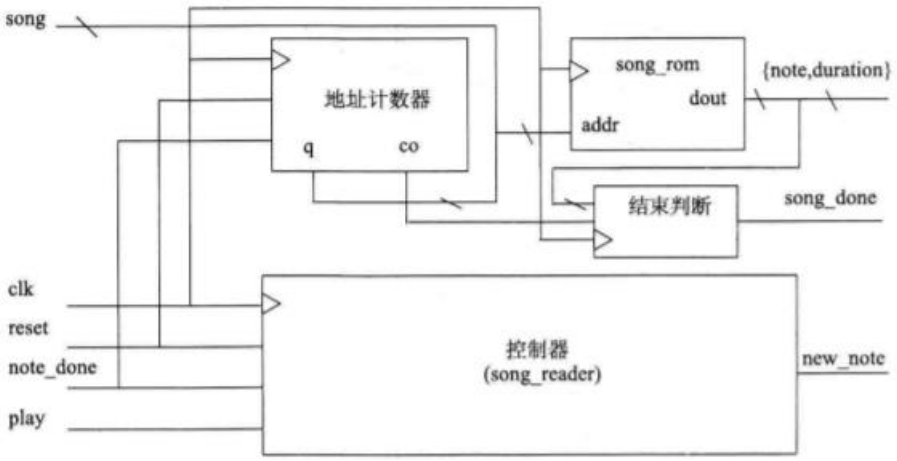

图 3.4.2.1 song_reader 的结构框图

控制器算法流程图如图 3.4.2.2 所示：

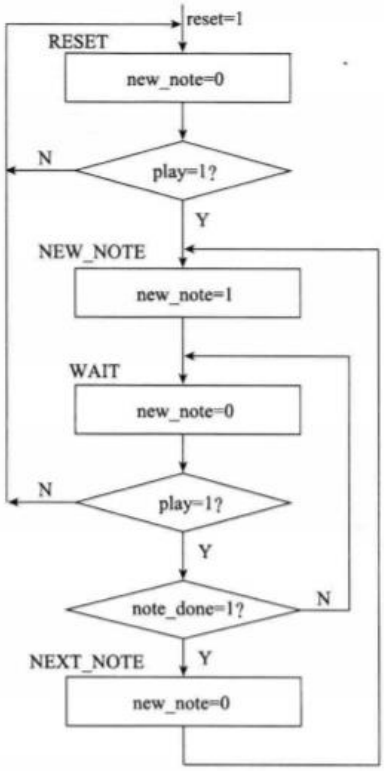

图 3.4.2.2 song_reader 控制器的算法流程图

③ 音乐播放模块 note_reader 设计

音符播放模块 note_player 是本实验的核心模块，其主要任务包括：

（1）从 song_reader 模块接收需播放的音符{note, duration}；

（2）根据 note 值找出 DDS 的相位增量 k；

（3）以 48kHz 速率从 Sine ROM 取出正弦样品送给音频编解码器接口模块；

（4）当一个音符播放完毕，向 song_reader 模块索取新的音符。

根据 note_player 模块的任务，可进一步划分功能单元，如图 3.4.3.1 所示，图中 FreqROM 为只读存储器，完成音符标记 note 与 DDS 模块的相位增量 k 查找表关系。

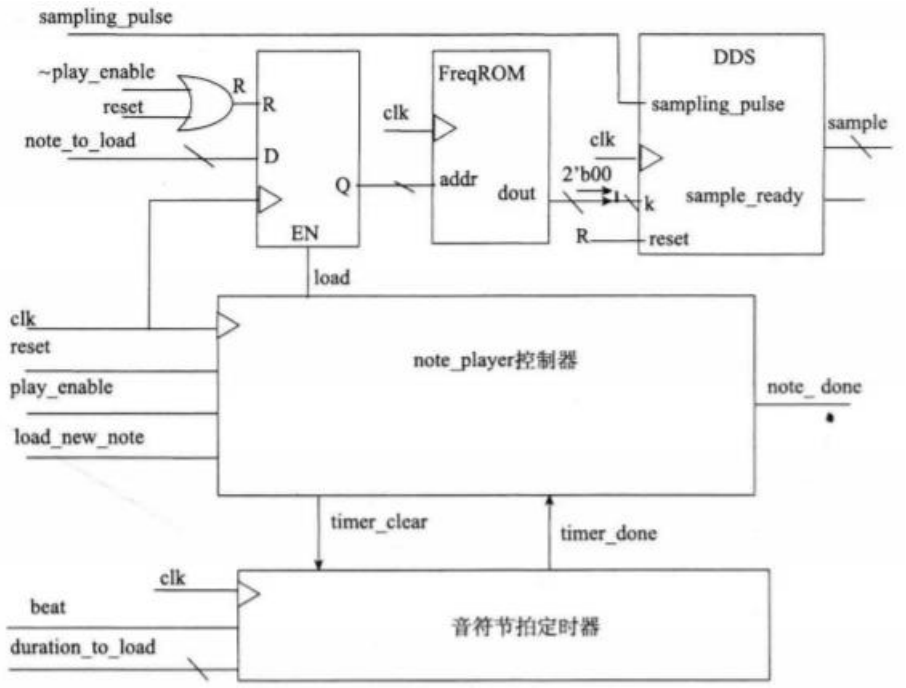

图 3.4.3.1 note_player 的结构框图

表 3.4.3 为 note_player 模块的端口含义：

表 3.4.3 note_player 模块的端口含义

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

note_player 控制器负责与 song_reader 模块接口，读取音符信息，并根据音符信息从 Frequency ROM 中读取相应相位增量 k 送给 DDS 子模块。另外，note_player 控制器还需要控制音符播放时间，其算法流程如图 3.4.3.2 所示：

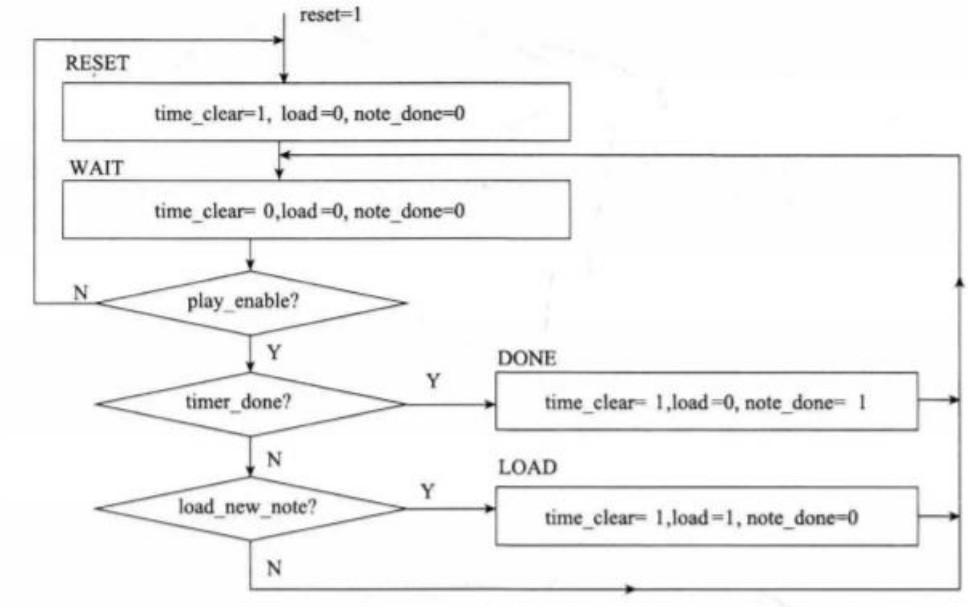

图 3.4.3.2 note_player 控制器的算法流程图

音符定时器为 6 位二进制计数器，beat、timer_clear 分别为使能、清 0 信号，均为高电平有效。定时时间由音长信号 duration_to_load 决定，即 duration_to_load-1 个 beat 周期，timer_done 为定时结束标志。

子模块 DDS 的功能就是利用 DDS 技术产生正弦样品。需要注意的是，DDS 模块的输入 k 为 22 位二进制，因此，Frequency ROM 输出的 20 位相位增量需要高位加两

个 0 后再接入 DDS。

④ 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，需通过同步化处理，产生与 sys_clk 同步且脉冲宽度为一个 sys_clk 时钟周期的信号 ready。其由同步器和脉冲宽度变换电路组成，电路结果如图所示：

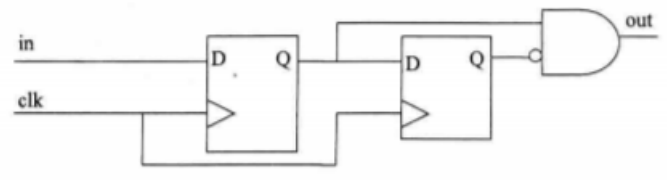

图 3.4.4.1 同步化电路的结构框图

⑤ 时钟管理模块（DCM）

IP 内核时钟管理模块的输入时钟 clk 频率为 100MHz，产生 100MHz 的系统时钟和 12.50MHz 的音频时钟。

四、主要仪器设备

① 装有 Vivado 和 ModelSim SE 软件的计算机；

② Nexys Video Artix-7 FPGA 多媒体音视频只能互联开发系统；

③ 有源音箱或耳机。

五、实验内容

① 从网站下载提供的文件；

② 编写 DDS 模块的 Verilog HDL 代码，并用 ModelSim 仿真验证；

③ 编写 mcu 模块的 Verilog HDL 代码，并用 ModelSim 仿真验证；

④ 编写 song_reader 模块的 Verilog HDL 代码，并用 ModelSim 仿真验证；

⑤ 编写 note_player 模块的 Verilog HDL 代码，并用 ModelSim 仿真验证；

⑥ 编写 1000 分频器模块的 Verilog HDL 代码及其测试代码，并用 ModelSim 仿真验证；

⑦ 编写同步化电路的 Verilog HDL 代码及其测试代码，并用 ModelSim 仿真验证；

⑧ 编写次顶层 music_player 模块的 Verilog HDL 代码及其测试代码，并用 ModelSim

仿真验证；

⑨ 新建 music_player 的 Vivado 工程，编写顶层 music_player_top 模块的 Verilog HDL 代码，生成符合要求的 DCM 内核，添加需要的文件，对工程进行综合、约束、实现，并下载工程文件到 Nexys Video 开发板中；

⑩ 将耳机接入实验开发板音频输出插座，操作 reset（中间按钮）、play/pause（右边按钮）、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变化情况，验证设计结果是否正确。

六、实验结果与仿真分析（此部分代码语句内换行皆是由于页面限制，原文件可查阅 Solutions）

① DDS 模块

（1）Verilog HDL 代码设计

DDS 模块由顶层模块 dds.v 和子模块加法器 full_adder.v、D 触发器 dffre.v 组成，其顶层代码如下：

```
module dds (clk, reset, sampling_pulse, k, sample, new_sample_ready);
    input clk, reset, sampling_pulse;
    input[21:0] k;  // 相位增量
    output[15:0] sample;  // 正弦信号
    output new_sample_ready;

    // 中间变量
    wire[21:0] raw_addr;  // 地址处理输入
    wire[9:0] rom_addr;   // 地址处理输出
    wire[15:0] raw_data, data;  // 数据处理
    wire[21:0] sum;  // 加法器结果
    wire area;   // 区域

    // 加法器实例
    full_adder adder0(.a(k), .b(raw_addr), .s(sum), .co()); // 进位输出空脚

    // D 触发器实例
    dffre #(.n(22)) dff0(.d(sum), .en(sampling_pulse), .r(reset),
                         .clk(clk), .q(raw_addr)); // 产生 raw_addr
    dffre #(.n(1)) dff1(.d(raw_addr[21]), .en(1), .r(0), .clk(clk),
                        .q(area));     // 产生 area
    dffre #(.n(16)) dff2(.d(data), .en(sampling_pulse), .r(0), .clk(clk),
                         .q(sample));  // 产生正弦信号
    dffre #(.n(1)) dff3(.d(sampling_pulse), .en(1), .r(0), .clk(clk),
                        .q(new_sample_ready));
```

```
    // 地址处理
    assign rom_addr[9:0] = raw_addr[20] ? ((raw_addr[20:10] == 1024) ?
                          1023 : (~raw_addr[19:10]+1)) : raw_addr[19:10];

    // 正弦查找表
    sine_rom rom0(.clk(clk), .addr(rom_addr), .dout(raw_data));

    // 数据处理
    assign data[15:0] = area ? (~raw_data[15:0]+1) : raw_data[15:0];
endmodule
```

（2）仿真结果

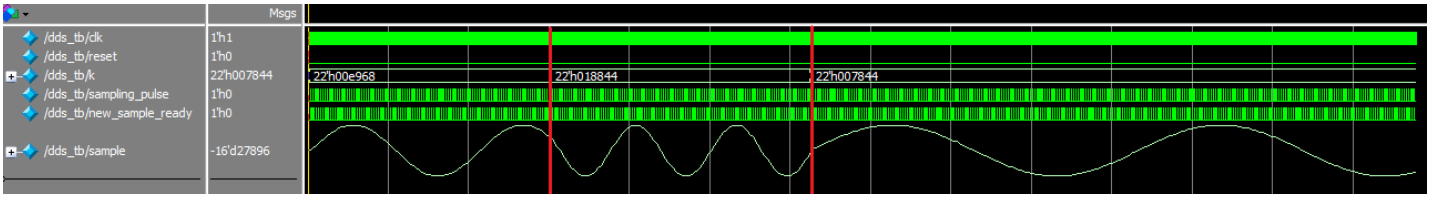

图 6.1 DDS 仿真结果

分析：如图，当 k 增大时，正弦输出信号的 sample 的频率变大；当 k 减小时，正弦输出信号的 sample 的频率变小。k 值变化的点即为频率转折点（图中体现为波形发生变化），满足频率关系。故设计符合要求。

② mcu 模块

（1）Verilog HDL 代码设计

mcu 模块由顶层模块 mcu.v 和子模块控制器 mcu_ctrl.v、D 触发器 dffre.v 组成。

i）顶层代码：

```
module mcu (clk, reset, play_pause, next,
            song_done, play, reset_play, song);
    input clk, reset;
    input play_pause; // play_pause 按钮
    input next; // next 按钮
    input song_done; // 乐曲播放结束
    output play;
    output reset_play; // 脉冲复位
    output[1:0] song; // 乐曲序号
    wire NextSong; // 控制器输出

    // 控制器实例
```

```
    mcu_ctrl ctrl0(.clk(clk), .reset(reset), .play_pause(play_pause),
                    .next(next), .song_done(song_done), .play(play),
                   .reset_play(reset_play), .NextSong(NextSong));

    // 2 位二进制计数器实例
    counter_n #(.n(4), .counter_bits(2)) song_cnt(.clk(clk),
                                  .en(NextSong), .r(0), .q(song), .co());
endmodule
```

ii）控制器代码：

```
module mcu_ctrl(clk, reset, play_pause, next, song_done,
                play, reset_play, NextSong);
    input clk, reset;
    input play_pause; // play_pause 按钮
    input next; // next 按钮
    input song_done; // 乐曲播放结束
    output reg play;
    output reg reset_play; // 脉冲复位
    output reg NextSong; // 控制器输出

    parameter RESET = 0, PAUSE = 1, PLAY = 2, NEXT = 3; // 状态编码
    reg[1:0] state, nextstate; // 当前状态与下一个状态

    // 状态寄存器
    always @ (posedge clk)
    begin
        if (reset)
            state = RESET; // 只要按下 RESET 按钮即复位
        else
            state = nextstate; // 否则继续运行
    end

    // 下一个状态和输出
    always @ (*)
    begin
        // 初始
        play = 0; NextSong = 0; reset_play = 0;
        case (state)
            // RESET 状态
            RESET: begin nextstate = PAUSE; reset_play = 1; end
            PAUSE: begin
                    if (play_pause) nextstate = PLAY;
                    else
                    begin
```

```
                    if (next) nextstate = NEXT;
                    else nextstate = PAUSE;
                end
            end // PAUSE 状态
        PLAY: begin
                play = 1;
                if (play_pause) nextstate = PAUSE;
                else
                begin
                    if (next) nextstate = NEXT;
                    else
                    begin
                        if (song_done) nextstate = RESET;
                        else nextstate = PLAY;
                    end
                end
            end // PLAY 状态
        NEXT: begin nextstate = PLAY; NextSong = 1; reset_play = 1; end
        // NEXT 状态
        default: nextstate = RESET;
    endcase
  end
endmodule
```

（2）仿真结果

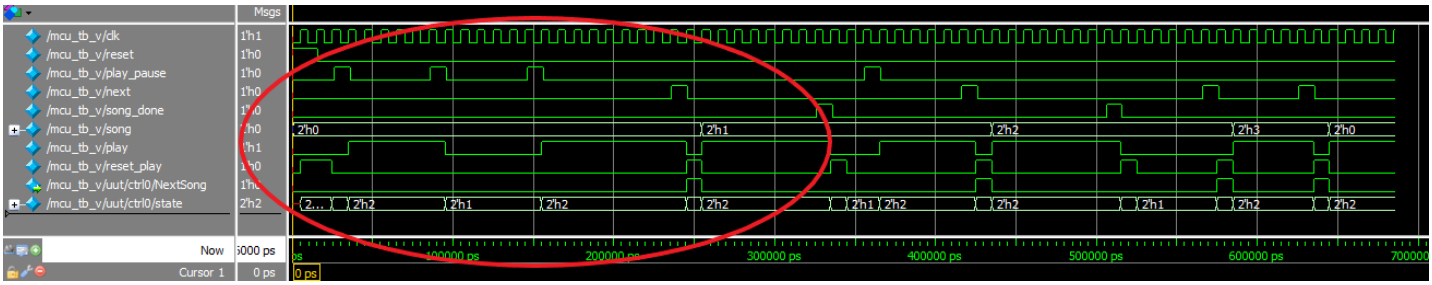

图 6.2.1 mcu 仿真结果（全图）

如图，将图中红线圈出部分放大，有：

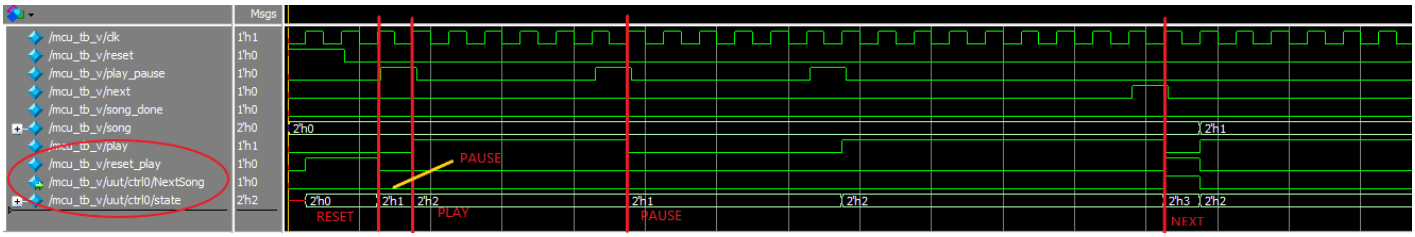

图 6.2.2 mcu 仿真结果（局部放大图）

分析：当第一个时钟上升沿到来时，reset = 1，于是控制器状态跳为 RESET（状态编码 0），此时输出 play = 0，NextSong = 0，reset_play = 1；

在第一条直线处，reset = 0，时钟上升沿到来后，控制器状态自然跳到 PAUSE（状态编码 1），此时输出为 play = 0，NextSong = 0，reset_play = 0；

在第二条直线处，reset = 0，play_pause = 1，时钟上升沿到来后，控制器状态跳为 PLAY（状态编码 2），此时输出 play = 1，NextSong = 0，reset_play = 0；

在第三条直线处，reset = 0，play_pause = 1，时钟上升沿到来后，控制器状态跳为 PAUSE，此时输出 play = 0，NextSong = 0，reset_play = 0；

在第四条直线处，前一刻状态为 PLAY，时钟上升沿到来时，reset = 0，play_pause = 0，next = 1，控制器状态跳为 NEXT，此时输出 play = 0，NextSong = 1，reset_play = 1；

综上可知，状态转换与输出均满足 ASM 图，模块设计正确。

③ song_reader 模块

（1）Verilog HDL 代码设计

该模块由顶层模块 song_reader.v 和子模块控制器 song_reader_ctrl.v、地址计数器模块 counter_n.v、歌曲存储模块 song_rom.v 和结束判断模块 song_over.v 组成，结束判断模块采用状态机方法实现。

i）顶层代码：

```
module song_reader(clk, reset, play, song, note_done,
                   song_done, note, duration, new_note);
    input clk, reset;
    input play; // 来自 mcu 的控制信号，高电平要求播放
    input note_done; // 模块 note_player 的应答信号，表示音符播放结束
    input[1:0] song; // 当前播放乐曲的序号
    output song_done; // 乐曲播放结束
    output new_note; // 表示新的音符需播放
    output[5:0] note; // 音符标记
    output[5:0] duration; // 音符持续时间

    wire[4:0] q; // 乐曲音符地址
    wire co; // 地址计数器进位

    // 地址计数器(5bits)
    counter_n #(.n(32), .counter_bits(5)) addr_cnt(.clk(clk),
                        .en(note_done), .r(reset), .q(q), .co(co));
```

```
    // 只读存储器
    song_rom rom0(.clk(clk), .dout({note, duration}), .addr({song, q}));

    // 结束判断
    song_over over0(.clk(clk), .ci(co), .din(duration),
                    .dout(song_done));

    // 控制器
    song_reader_ctrl ctrl0(.clk(clk), .reset(reset),
                           .note_done(note_done), .play(play),
                           .new_note(new_note));
endmodule
```

ii）控制器代码：

```
module song_reader_ctrl(clk, reset, note_done, play, new_note);
    input clk, reset;
    input note_done; // 音符播放结束
    input play; // 高电平要求播放
    output  reg new_note; // 新的音符需要播放

    parameter RESET = 0, NEW_NOTE = 1, WAIT = 2, NEXT_NOTE = 3; // 状态编码
    reg[1:0] state, nextstate;

    always @ (posedge clk)
    begin
        if (reset) state = RESET; // 若复位，则系统处于 RESET 状态
        else state = nextstate;
    end

    always @ (*)
    begin
        new_note = 0;
        case (state)
            RESET: begin // 处于 RESET 状态
                    if(play) nextstate = NEW_NOTE;
                    else nextstate = RESET;
                  end
            NEW_NOTE: begin // 处于 NEW_NOTE 状态
                        new_note = 1;
                        nextstate = WAIT;
                      end
            WAIT: begin // 若继续播放且乐符已播完，则 NEXT_NOTE
                    if (play)
                    begin
```

```
                        if (note_done) nextstate = NEXT_NOTE;
                        else nextstate = WAIT;
                    end
                    else nextstate = RESET;
                end
            NEXT_NOTE: nextstate = NEW_NOTE; // 准备播放下一个音符
            default: nextstate = RESET;
        endcase
    end
endmodule
```

iii）结束判断模块代码：

```
module song_over(clk, ci, din, dout);
    input clk, ci; // ci 为地址计数器进位，高电平表示乐曲结束
    input[5:0] din; // song_rom 输出 duration
    output reg dout; // 即 song_done，乐曲播放结束

    parameter PLAY = 0, OVER = 1; // 状态编码
    reg state, nextstate;

    // 状态寄存器
    always @ (posedge clk)
    begin
        if (ci)
        begin
            state = OVER; // 若计数器进位，则播放结束
        end
        else state = nextstate;
    end

    // 下一个状态和输出
    always @ (*)
    begin
        // 初始
        dout = 0;
        case (state)
            PLAY: begin
                    if (!din) // 若 duration 为 0，则播放结束
                    begin
                        nextstate = OVER;
                        dout = 1;
                    end
                    else nextstate = PLAY;  // 否则继续播放
                end
```

```
            OVER: begin
                    if (!din) nextstate = OVER;
                    // 若 duration 为 0，则继续为 OVER 状态
                    else nextstate = PLAY; // 否则播放
                  end
            default: begin nextstate = OVER; dout = 1; end
        endcase
    end
endmodule
```

iv）歌曲存储模块

新增有一首歌曲《送别》，新增代码为（详情参见 song_rom.v）：

```
// 《送别》
assign memory[  96  ] = {6'd35, 6'd24}  ;   // Note:    3G
assign memory[  97  ] = {6'd32, 6'd12}  ;   // Note:    3E
assign memory[  98  ] = {6'd35, 6'd12}  ;   // Note:    3G
assign memory[  99  ] = {6'd40, 6'd48}  ;   // Note:    4C
assign memory[  100 ] = {6'd37, 6'd24}  ;   // Note:    4A
assign memory[  101 ] = {6'd40, 6'd24}  ;   // Note:    4C
assign memory[  102 ] = {6'd35, 6'd48}  ;   // Note:    3G
assign memory[  103 ] = {6'd35, 6'd24}  ;   // Note:    3G
assign memory[  104 ] = {6'd28, 6'd12}  ;   // Note:    3C
assign memory[  105 ] = {6'd30, 6'd12}  ;   // Note:    3D
assign memory[  106 ] = {6'd32, 6'd24}  ;   // Note:    3E
assign memory[  107 ] = {6'd30, 6'd12}  ;   // Note:    3D
assign memory[  108 ] = {6'd28, 6'd12}  ;   // Note:    3C
assign memory[  109 ] = {6'd30, 6'd48}  ;   // Note:    3D
assign memory[  110 ] = {6'd0, 6'd24}   ;   // Note:    rest
assign memory[  111 ] = {6'd0, 6'd24}   ;   // Note:    rest
assign memory[  112 ] = {6'd35, 6'd24}  ;   // Note:    3G
assign memory[  113 ] = {6'd32, 6'd12}  ;   // Note:    3E
assign memory[  114 ] = {6'd35, 6'd12}  ;   // Note:    3G
assign memory[  115 ] = {6'd40, 6'd36}  ;   // Note:    4C
assign memory[  116 ] = {6'd39, 6'd24}  ;   // Note:    4B
assign memory[  117 ] = {6'd37, 6'd24}  ;   // Note:    4A
assign memory[  118 ] = {6'd40, 6'd24}  ;   // Note:    4C
assign memory[  119 ] = {6'd35, 6'd48}  ;   // Note:    3G
assign memory[  120 ] = {6'd35, 6'd24}  ;   // Note:    3G
assign memory[  121 ] = {6'd30, 6'd12}  ;   // Note:    3D
assign memory[  122 ] = {6'd32, 6'd12}  ;   // Note:    3E
assign memory[  123 ] = {6'd33, 6'd36}  ;   // Note:    3F
assign memory[  124 ] = {6'd27, 6'd12}  ;   // Note:    2B
assign memory[  125 ] = {6'd28, 6'd48}  ;   // Note:    3C
```

```
assign memory[  126 ] = {6'd0, 6'd24}   ;   // Note:     rest
assign memory[  127 ] = {6'd0, 6'd24}   ;   // Note:     rest
```

（2）仿真结果

i）顶层及控制器

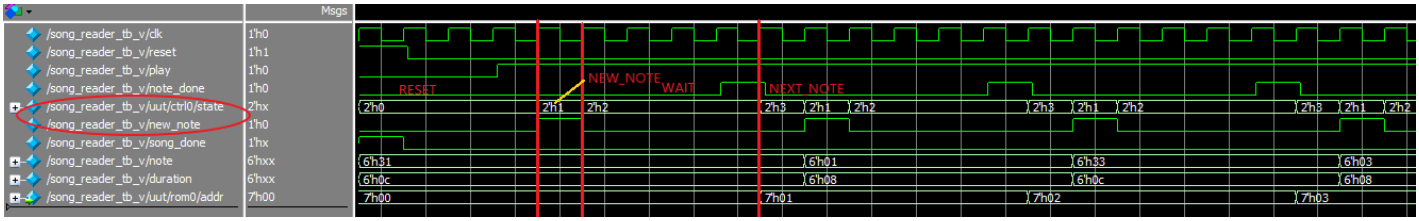

图 6.3.1 song_reader 控制器仿真结果

分析：当第一个时钟上升沿到来时，reset = 1，于是控制器状态 state = RESET，输出 new_note = 0。在后续时钟周期中，由于 play = 0，故控制器状态始终保持为 RESET；

在第一条直线处，reset = 0，play = 1，当时钟上升沿到来时，控制器状态跳为 NEW_NOTE，输出 new_note = 1；

在第二条直线处，控制器自然跳到 WAIT 状态，输出 new_note = 0。此时由于 play 始终为 1，note_done 始终为 0，故控制器保持为 WAIT 状态；

在第三条直线处，由于 play = 1，note_done = 1，当时钟上升沿到来时，控制器跳转为 NEXT_NOTE 状态，输出 new_note = 0；

综上可知，控制器状态转换与输出均满足 ASM 图，设计符合要求。

ii）结束判断

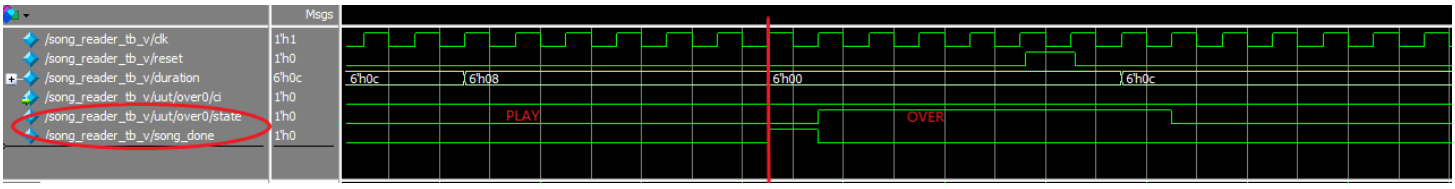

图 6.3.2 song_over 仿真结果

分析：如图，地址计数器进位输出始终为 0，当 duration = 0 时（图中红线处），结束判断模块输出 song_done = 1，其状态在下一个时钟信号到来时由 PLAY（状态编码为 0）跳转为 OVER（状态编码为 1）。当 duration 不再等于 0 时，其状态又将回到 PLAY 状态，满足设计要求。

iii）控制器、地址计数器与 ROM 的时序关系

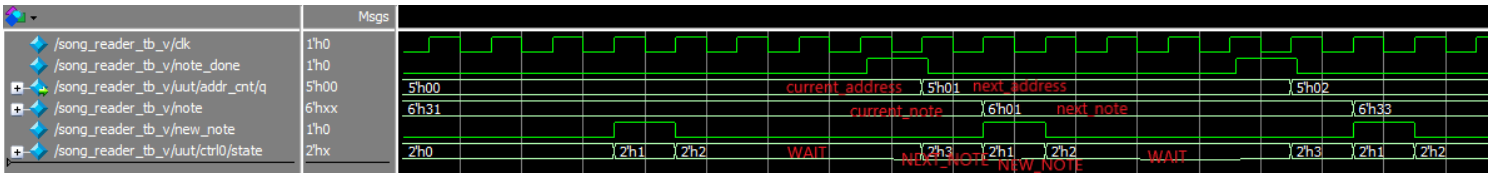

图 6.3.3 控制器、地址计数器与 ROM 的时序关系图

分析：如图，song_reader 控制器、地址计数器与 ROM 的时序关系与书中要求一致，设计正确。

④ note_player 模块

（1）Verilog HDL 代码设计

该模块由顶层模块 note_player.v 和子模块控制器 note_player_ctrl.v、音符节拍定时器 note_timer.v、D 触发器 dffre.v、只读存储器 FreqROM.v、DDS 模块 dds.v 组成。

i）顶层代码：

```
module note_player(clk, reset, play_enable, note_to_load,
                   duration_to_load, load_new_note, note_done,
                   sampling_pulse, beat, sample, sample_ready);

    input clk, reset;
    input play_enable; // 高电平表示播放
    input[5:0] note_to_load; // 需播放的音符
    input[5:0] duration_to_load; // 需播放音符的音长
    input load_new_note; // 新的音符需播放
    input sampling_pulse; // 索取新的正弦样品
    input beat; // 定时基准信号
    output note_done; // 音符播放完毕
    output[15:0] sample; // 正弦样品输出
    output sample_ready; // 下一个正弦信号

    wire[5:0] q; // FreqROM 地址输入
    wire[19:0] dout; // FreqROM 输出
    wire timer_clear, timer_done; //音符定时器输入输出
    wire load; // D 触发器使能输入

    // D 触发器实例
    dffre #(.n(6)) d0(.d(note_to_load), .en(load),
                      .r(reset || ~play_enable),.clk(clk), .q(q));

    // Frequency ROM，dout 为 DDS 模块 k 的后 20 位
```

```
    frequency_rom rom0(.clk(clk), .dout(dout), .addr(q));

    // DDS 实例
    dds dds0(.clk(clk), .reset(reset || ~play_enable),
             .sampling_pulse(sampling_pulse), .k({2'b00, dout}),
             .sample(sample), .new_sample_ready(sample_ready));

    // note_player 控制器
    note_player_ctrl ctrl0(.clk(clk), .reset(reset),
                           .play_enable(play_enable),
                           .load_new_note(load_new_note), .load(load),
                           .timer_clear(timer_clear),
                           .timer_done(timer_done),
                           .note_done(note_done));

    // 音符节拍定时器
    note_timer timer0(.clk(clk), .beat(beat),
                      .duration_to_load(duration_to_load),
                      .timer_clear(timer_clear),
                      .timer_done(timer_done));
endmodule
```

ii）控制器代码：

```
module note_player_ctrl(clk, reset, play_enable, load_new_note,
                        load, timer_clear, timer_done, note_done);
    input clk, reset;
    input play_enable; // 来自 mcu 的 play 信号，高电平表示播放
    input load_new_note; // 来自 song_reader 的 new_note 信号，表示新音符需要播放
    input timer_done; // 定时结束标志
    output reg load; // D 触发器的使能输入
    output reg timer_clear; // 清 0 信号
    output reg note_done; // 给 song_reader 的应答信号，表示音符播放完毕

    parameter RESET = 0, WAIT = 1, DONE = 2, LOAD = 3; // 状态编码
    reg[1:0] state, nextstate;

    always @ (posedge clk)
    begin
        if (reset) state = RESET; // 若复位，则处于 RESET 状态
        else state = nextstate;
    end

    always @ (*)
    begin
```

```
        timer_clear = 0; load = 0; note_done = 0; // 初始化
        case (state)
            RESET: begin timer_clear = 1; nextstate = WAIT; end //RESET 状态
            WAIT: begin
                    if (play_enable)
                    begin
                        if (timer_done) nextstate = DONE; // 定时结束
                        else
                        begin
                            if (load_new_note) nextstate = LOAD;//读取新音符
                            else nextstate = WAIT;
                        end
                    end
                    else nextstate = RESET;
                  end
            DONE: begin // 音符播放完毕
                    timer_clear = 1; note_done = 1;
                    nextstate = WAIT;
                  end
            LOAD: begin // 读取新的音符
                    timer_clear = 1; load = 1;
                    nextstate = WAIT;
                  end
            default: nextstate = RESET; // 否则处于 RESET 状态
        endcase
    end
endmodule
```

iii）音符节拍定时器代码：

```
module note_timer(clk, beat, duration_to_load, timer_clear, timer_done);
    input clk;
    input beat; // 定时基准信号，频率为 48Hz 的脉冲(使能输入)
    input[5:0] duration_to_load; // 播放音符的音长
    input timer_clear; // 清 0 信号
    output timer_done; // 定时结束
    reg[5:0] cnt = 0; // 计数，因为音长为 5bits(32)，故 cnt 长为 5 位

    assign timer_done = (cnt == (duration_to_load - 1)); // 定时结束

    always @ (posedge clk)
    begin
        if (timer_clear) cnt = 0; // 清 0
        else
        begin
```

```
            if (beat) cnt = cnt + 1;
            else cnt = cnt; // 若 beat 高电平则计数，否则保持
        end
    end
endmodule
```

（2）仿真结果

i）顶层设计

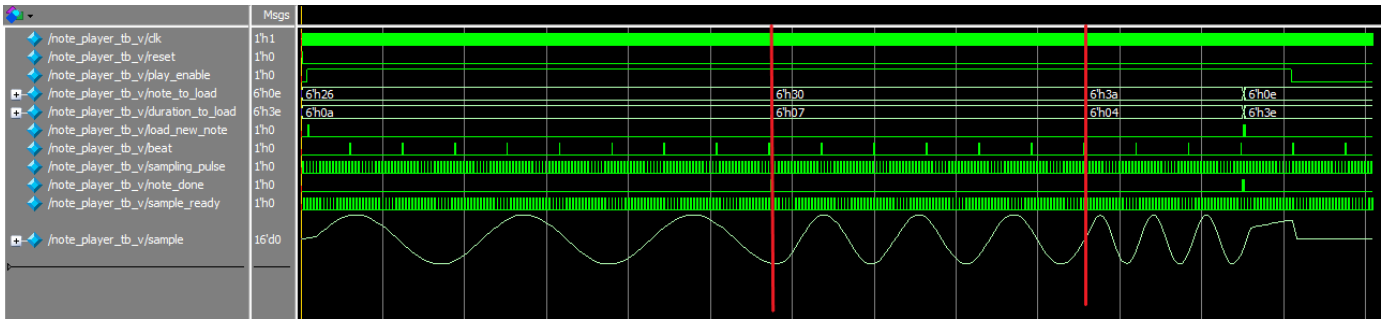

图 6.4.1 note_player 仿真结果

分析：如图，当 note_to_load 增大时，输出 sample 正弦信号频率增大，两者呈正比关系；note_to_load 变化的点即为频率转折点。当 play_enable = 0 时，sample = 0，符合设计要求。

ii）控制器

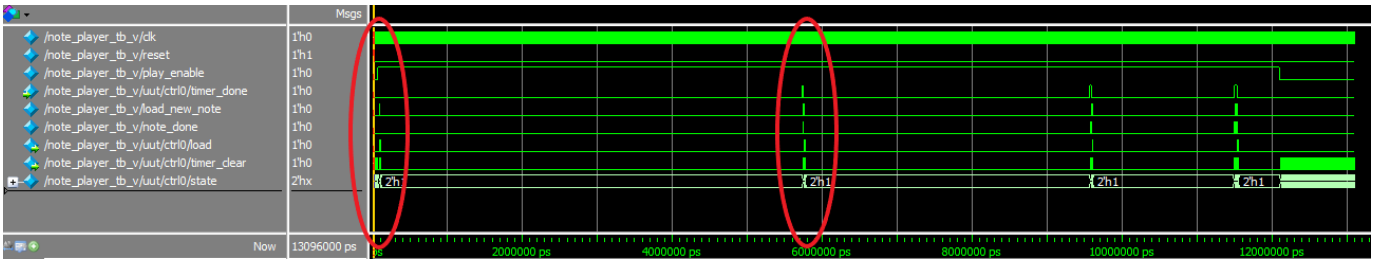

图 6.4.2.1 控制器仿真结果（全图）

前、后两处放大图分别为：

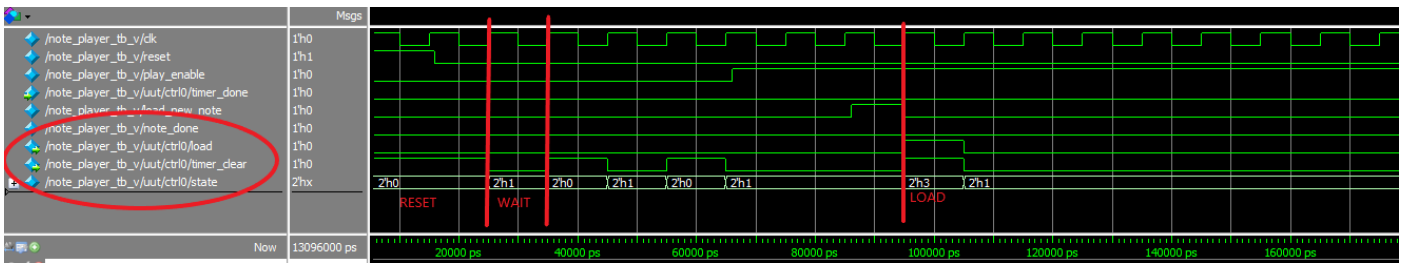

图 6.4.2.2 控制器仿真局部放大图（前）

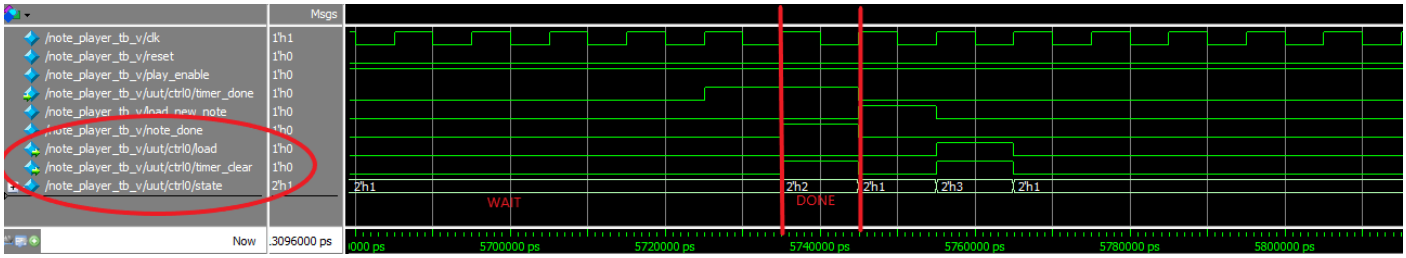

图 6.4.2.3 控制器仿真局部放大图（后）

分析：如图 6.4.2.2 所示，当 reset = 1 时，控制器 state = 0，即为 RESET 状态，此时输出 time_clear = 1，load = 0，note_done = 0；

在第一条直线处，时钟上升沿到来时，控制器自然进入 WAIT 状态（状态编码为 1），此时 time_clear = 0，load = 0，note_done = 0；

在第二条直线处，由于 play_enable = 0，控制器回到 RESET 状态，与 ASM 图要求一致；

在第三条直线处，前一刻控制器状态为 WAIT，由于时钟上升沿到来时，play_enable = 1，timer_done = 0，load_new_note = 1，控制器状态跳转为 LOAD，输出 time_clear = 1，load = 1，note_done = 0；

如图 6.4.2.3，在第一条直线处，前一刻控制器状态为 WAIT，当时钟上升沿到来时，play_enable =1，timer_done = 1，使得控制器状态跳为 DONE，输出 time_clear = 1，load = 0，note_done = 1；

在第二条直线处，控制器自然跳转到 WAIT 状态，符合 ASM 图要求。

综上可知，控制器设计正确。

iii）音符节拍定时器

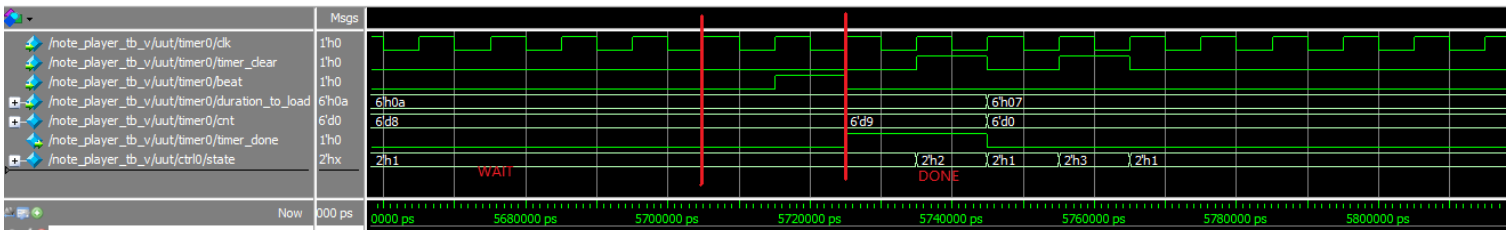

图 6.4.3 音符节拍定时器仿真局部放大图

分析：如图，在第一条直线处，beat = 0，即使能信号为低电平，定时器不计数，cnt 保持为 8；在第二条直线处，beat = 1，定时器计数，由于此时 duration = 10，而 cnt = 9 = 10-1，定时结束，timer_done = 1，符合要求。可以看见，此时控制器的状态由 WAIT 转为 DONE，

与 ASM 图一致。综上，定时器设计正确。

⑤ 1000 分频器模块

（1）Verilog HDL 代码设计

1000 分频器即为模 1000 的计数器，其状态位数应设为 10 位，因此，只需将 counter_n.v 中的两个参数 n、counter_bits 设为对应值即可，对应测试代码可查看 counter_n_tb.v 文件。

（2）仿真结果

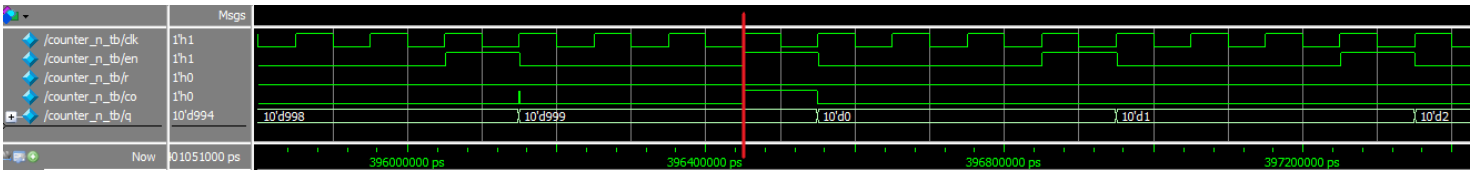

图 6.5 1000 分频器仿真结果

分析：如图，当使能信号 en = 1 时，分频器正常计数；当分频器计数到 999，en = 1，且时钟上升沿到来时，分频器进位输出 co = 1，随后计数值清 0，满足设计要求。

⑥ 同步化电路

（1）Verilog HDL 代码设计

由原理图可知，同步化电路由两个 D 触发器构成，其设计代码如下：

```
module syn_circ(in, clk, out);
    input in, clk; // 输入
    output out;
    reg q1, q2; // 两个触发器的输出

    always @ (posedge clk)
    begin
        q1 <= in; // 非阻塞赋值
        q2 <= q1;
    end
    assign out = q1 && (~q2);
endmodule
```

（2）仿真结果

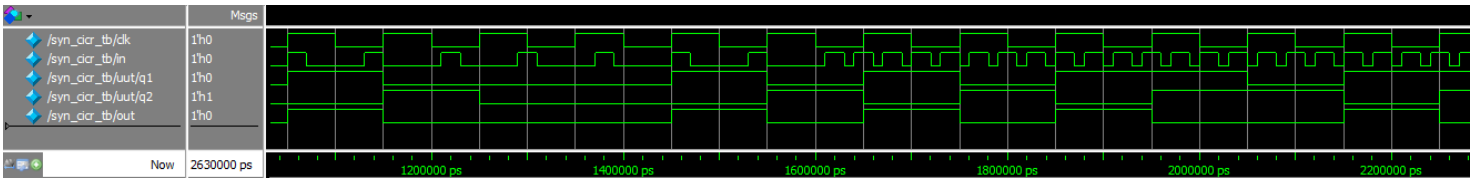

图 6.6 同步化电路仿真结果

分析：如图，经过一段时间后，输出信号 out 与输入时钟 in 同步，且其脉冲宽度为一个 in 的时钟周期，设计符合要求。

⑦ 次顶层模块

（1）Verilog HDL 代码设计

music_player 次顶层模块由子模块主控制器 mcu.v、乐曲读取 song_reader.v、音符播放 note_player.v、同步化电路 synch.v 和节拍基准产生器 counter_n.v 组成，其代码如下：

```
module music_player(clk, reset, play_pause, next,
                    NewFrame, sample, play, song);
    input clk, reset; // 高电平有效
    input play_pause; // 播放/暂停
    input next; // 下一首输入
    input NewFrame; // 高电平非同步脉冲，索取新的样品
    output[15:0] sample; // 正弦样品输出
    output play; // 播放状态指示
    output[1:0] song; // 曲号指示

    wire reset_play; // mcu 复位模块输出
    wire song_done, new_note; // song_reader 输出信号
    wire[5:0] note, duration; // 音符标记及持续时间
    wire note_done, sample_ready; // note_player 输出信号
    wire ready; // 同步化电路输出
    wire beat; // 节拍基准产生器输出
    parameter sim = 0;

    // 主控制器 mcu 实例
    mcu m0(.clk(clk), .reset(reset), .play_pause(play_pause), .next(next),
          .song_done(song_done), .play(play), .reset_play(reset_play),
          .song(song));

    // 乐曲读取 song_reader 实例
    song_reader s0(.clk(clk), .reset(reset_play), .play(play),
                   .song(song), .note_done(note_done),
```

```
                .song_done(song_done), .note(note),
                .duration(duration), .new_note(new_note));

    // 音符播放 note_player 实例
    note_player n0(.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),
                .sample_ready(sample_ready));

    // 同步化电路 syn_circ 实例
    syn_circ syn0(.in(NewFrame), .clk(clk), .out(ready));

    // 节拍基准产生器实例，根据 sim 的值选择对应模式
    counter_n #(.n(sim ? 64 : 1000), .counter_bits(sim ? 6 : 10))
              c0(.clk(clk), .en(ready), .r(0), .q(), .co(beat));
endmodule
```

（2）仿真结果

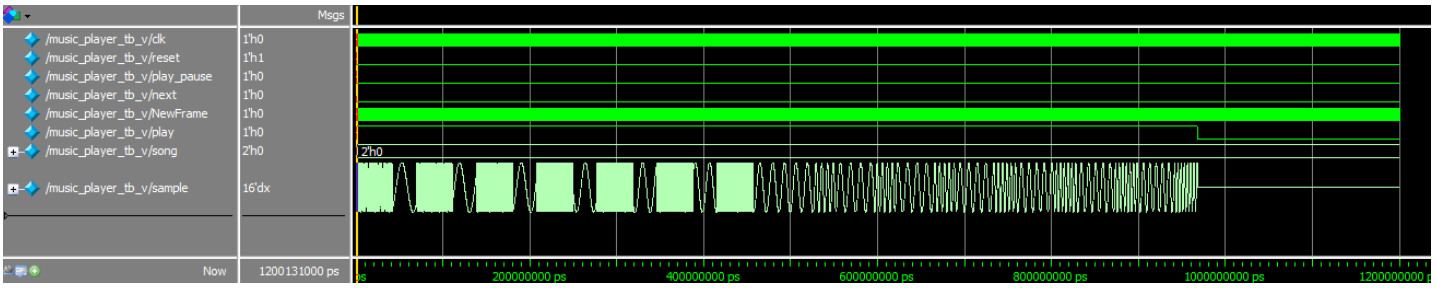

图 6.7 次顶层模块仿真结果

分析：如图，仿真结果与书中要求基本一致。

⑧ Vivado 工程建立

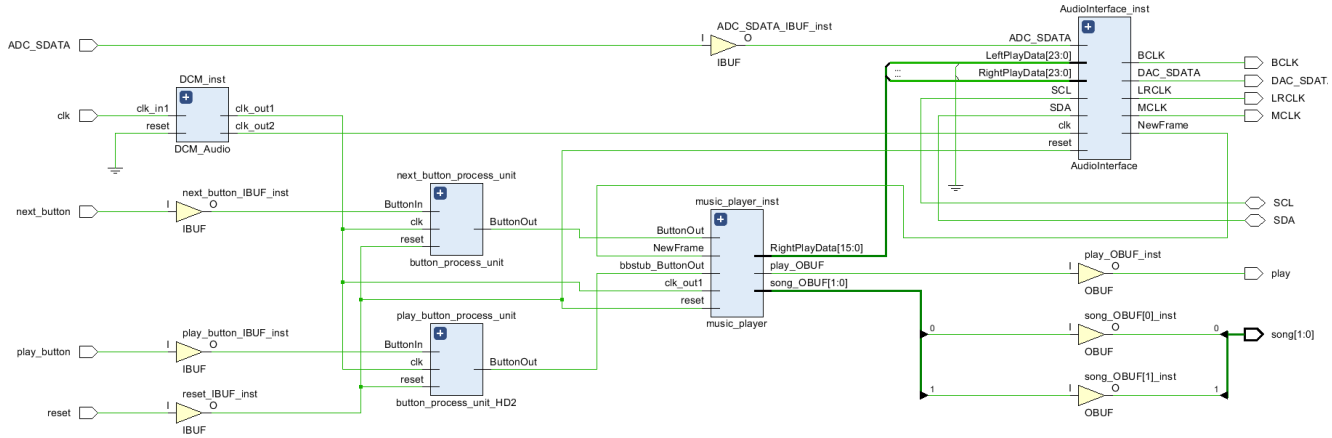

图 6.8 Vivado 工程建立

七、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 reset 为置零信号，当 reset = 1 时，系统置零，之后不论 reset 为 0 或 1，系统内部应置零的信号都为 0，其颤动对系统无影响。而 next_button 和 play_pause_button 都会影响系统内部状态的转换，若 next 输入不稳定，可能会使播放的下一首歌连续跳转，引起不确定；若 play_pause 输入不稳定，则系统会在播放和暂停之间不断切换，导致输出断断续续，并增加系统的损耗，影响系统的使用效果。

2、在主控制器（mcu）设计中，是否存在接收不到按键信息？若存在，概率多大？有没必要修改设计？

答：存在。由控制器 ASM 图可知，从 RESET 到 PAUSE 转换，以及从 NEXT 到 PLAY 转换时，并没有判断 play_pause 和 next，这期间按下按钮将接收不到按键信息。由于状态转换的间隔只有一个时钟周期，接收不到按键信息的概率很小，因此没有必要修改设计。

八、心得与体会

此次实验，我们在完成前期组合逻辑电路和时序逻辑电路的基础上，设计完成了一个音乐播放器，不仅学会了状态机的描述，也能够自主编写测试代码，对模块进行测试、纠错，更重要的是对“自顶而下”的数字系统设计方法有了更加深刻的认识，受益颇丰。

总的来说，这次实验是比较困难的。由于最初看不懂算法流程图，我花了较长的时间进行摸索，对自己设计出来的控制器也将信将疑。直到学习了学在浙大上的视频，我对状态机描述的方法有了一个大体的了解，结合书中的 ASM 图，方才掌握了一些门道。根据模块化的设计方法，借助书中的结构框图和端口含义，我在逐一完成各个功能模块的同时，也对数字系统设计的流程和方法有了更新的认识。子功能模块的编写、顶层设计中的调用，将功能和结构两者有机地结合起来，不仅使代码结构清晰明了，也有利于我们的编写和调试。当然，在实验过程当中，我也遇到了其他一些问题，主要有以下两个：

1、端口、变量的名称写错。在对音符播放部分进行仿真时，我发现 timer_done 信号始终为 0，导致控制器出错。通过对音符定时器模块进行调试，对整个部分的代码进行检查，我发现原来是因为自己将顶层设计中的单词 load 错写成了 laod，导致变量名

出错。修正之后进行仿真，所得结果正确；

2、变量长度忘了设置。同样是在音符播放部分，在调试过程中，我发现 DDS 模块的输入 *k* 始终为高阻态，使得 sample 无法输出正弦信号。通过检查相关模块的波形，我发现 dout 存在错误。查看代码，原来是其长度忘了设置，导致传递给 DDS 的变量出错。修正之后进行仿真测试，所得结果正确。

从上述两个问题当中，我已经深刻认识到了细致的重要性。相较而言，任何一点小小的错误，在工程当中都可能会产生巨大的影响。不论是在课堂中的实验，还是我们以后的工作生涯，都要时刻保持细心谨慎、掌握原理和方法，才能把事情做得更好。