浙江大学

本科实验报告

实验设计报告

课程名称：	数字系统设计实验
姓　　名：	箫宇
学　　院：	信息与电子工程学院
系：	
专　　业：	信息工程
学　　号：	
指导老师：	屈民军、唐奕

2023 年 7 月 2 日

浙江大学实验报告

专业：信息工程
姓名：箫宇
学号：
日期：2023 年 7 月 2 日
地点：东 4-223

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：实验设计报告　实验类型：设计实验　同组学生姓名：

一　实验目的和要求

1.　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”自顶向下”的数字系统设计方法

2.　实验要求

设计出一个能够播放四首乐曲的音乐播放器，满足如下两个要求：

(1) 设置 play/pause_button、next_button、reset 三个按键.play/pause_button 按键实现乐曲在暂停与播放之间切换，按下 next_button 可以播放下一首乐曲；

(2) 设置 3 个 LED 灯，LED0 显示目前的播放状态 (亮为播放，灭为暂停)，LED1 和 LED2 显示目前乐曲号

二　实验内容和原理

1.　主控制器模块

1.1　设计说明

主控制器模块主要作用为响应用户按键信息、控制系统播放两大任务，算法流程图和书上类似，此处不表。

1.2　主控制器代码

Listing 1: 主控制器模块代码

```
1 //主控制器
module mcu (
3     input clk , reset , play_pause , next , song_done,
    output reg play,reset_play,
```

```
    output [1:0] song
);
    reg NextSong;

    //状态编码
    parameter RESET = 0 , PAUSE = 1 , PLAY = 2 , NEXT = 3;
    reg [1:0] state , nextstate;

    //时序逻辑，控制状态切换
    always @(posedge clk) begin
        if(reset) state = RESET;
        else state = nextstate;
    end


    //组合部分：控制次态以及控制器部分输出
    always @(*) begin
        //初始状态
        play = 0 ; NextSong = 0 ; reset_play = 1;
        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)begin
                    nextstate = PLAY;
                end
                else if(next)begin
                    nextstate = NEXT;
                end
                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;
49          end
        endcase
51  end

53  //实例化两位二进制计数器得到song
    counter_n #(.n(4) , .counter_bits(2)) counter(.en(NextSong) , .r(reset
        ) , .clk(clk) , .q(song) , .co());
55
endmodule
```

2. song_reader 模块

2.1 设计说明

song_reader 模块任务如下：

(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。

我们采用了书上给出的电路结构实现了上述功能，其中 song_rom 模块是一个只读存储器，用于存放乐曲；地址计数器计算播放的音符数，它的进位输出作为；结束判断模块利用地址计数器进位输出和从 song_rom 中取出的 duration 判断是否输出 song_done 信号。

2.2 结束判断模块设计

结束判断模块采用了老师建议的状态机方法实现：
一、地址计数器进位输出 co 作为 reset 信号，duration 单独输入，共四个状态实现，代码如下：

Listing 2: 判断结束模块一

```
//判断乐曲是否播放结束模块
2 module is_over (
      input [5:0] duration ,
4     input reset , clk ,
      output reg done
6 );
      parameter RESET = 0 , PAUSE = 1 , PLAY = 2 , OUT = 3;
8     reg [1:0] state=0 , nextstate;

10    always @(posedge clk) begin
          if(reset) state = RESET;
12        else state = nextstate;
```

```
    end

    always @(*) begin
        done = 0;
        case (state)
            RESET:begin
                done = 1;
                nextstate = PAUSE;
            end
            PAUSE:begin
                done = 0;
                if(duration)begin
                    nextstate = PLAY;
                end
                else nextstate = PAUSE;
            end
            PLAY:begin
                done = 0;
                if(duration) nextstate = PLAY;
                else nextstate = OUT;
            end
            OUT:begin
                done = 1;
                nextstate = PAUSE;
            end
        endcase
    end
endmodule
```

二、听了老师的讲解之后，意识到将两个信号按”duration == 0 || co” 进行输入状态机会更加简单，如是将代码简化至如下：

Listing 3: 判断结束模块二

```
//判断乐曲是否播放结束模块
module is_over (
    input [5:0] duration ,
    input reset , clk ,
    output reg done
);
    parameter RESET = 0 , PAUSE = 1 , PLAY = 2 , OUT = 3;
    reg [1:0] state=0 , nextstate;

    always @(posedge clk) begin
```

```
        if(reset) state = RESET;
        else state = nextstate;
    end

    always @(*) begin
        done = 0;
        case (state)
            RESET:begin
                done = 1;
                nextstate = PAUSE;
            end
            PAUSE:begin
                done = 0;
                if(duration)begin
                    nextstate = PLAY;
                end
                else nextstate = PAUSE;
            end
            PLAY:begin
                done = 0;
                if(duration) nextstate = PLAY;
                else nextstate = OUT;
            end
            OUT:begin
                done = 1;
                nextstate = PAUSE;
            end
        endcase
    end
endmodule
```

2.3 song_reader 模块代码

Listing 4: song_reader 模块

```
    module song_reader (
        input clk,reset,play,note_done,
        input [1:0] song,
        output reg new_note,
        output song_done,
        output [5:0] note , duration
    );
        //状态编码
```

```
    parameter RESET = 0 , NEW_NOTE = 1 , WAIT = 2 , NEXT_NOTE = 3;
    reg [1:0] state , nextstate;

    wire [4:0] lowaddr;//song_rom的低五位地址
    wire judge;//歌曲结束标志一

    //控制器时序部分
    always @(posedge clk) begin
        if(reset) state = RESET;
        else state = nextstate;
    end

    //控制器组合部分
    always @(*) begin
        //默认输出
        new_note = 0;
        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 == 0) nextstate = RESET;
                else if(note_done) nextstate = NEXT_NOTE;
                else nextstate = WAIT;
            end
            NEXT_NOTE:begin
                new_note = 0;
                nextstate = NEW_NOTE;
            end
        endcase
    end

    //实例化地址计数器
    counter_n #(.n(32) , .counter_bits(5)) addrCounter(.clk(clk) , .r(
       reset) , .en(note_done) , .q(lowaddr) , .co(judge));

```

```
        //实例化song_rom, 取出音符
52      song_rom song_rom(.clk(clk) , .dout({note,duration}) , .addr({song
            ,lowaddr}));

54      //实例化判断模块
        over is_over(.signal((duration==0)||co) , .reset(reset) , .clk(clk
            ) , .done(song_done));
56
    endmodule
```

3. note_player 模块

3.1 设计说明

音符播放模块 note_ player 是本实验的核心模块, 它主要任务包括以下几方面。

(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) 当一个音符播放完成，向 song_reader 模块索取新的音符

进一步划分模块可将 note_player 划分为一下各个模块：作为控制单元的控制器、记录音符标记 note 和 DD 模块相位增量 k 查找表关系的 FreqROM、DDS 模块、音符节拍计时器。

下面给出 DDS 模块、节拍计时器、note_player 代码

3.2 代码

Listing 5: DDS 模块

```
1   module dds (
        K , clk , reset ,sampling_pulse , sample , new_sample_ready
3   );
        input [21:0] K;
5       input clk , reset , sampling_pulse;
        output [15:0] sample;
7       output new_sample_ready;

9       wire [21:0] temp ;//作为加法器输出
        wire [21:0] raw_addr; //待处理地址
11      wire [9:0] rom_addr; //ROM地址
        wire area;//区域
13      wire [15:0] raw_data;//原始数据
        wire [15:0] data;//处理后的数据
```

```
15
    //实例化加法器
17  adder_n #(.n(22)) adder(.adder1(K) , .adder2(raw_addr) , .result(temp)
        , .co());

19  //实例化D型寄存器
    dffre #(.n(22)) D1(.d(temp) , .en(sampling_pulse) , .r(reset) , .clk(
        clk) , .q(raw_addr));
21
    //处理地址
23  //rom_addr应该可以不指定位数
    assign rom_addr = raw_addr[20]?((raw_addr[20:10]==1024)?1023:(~
        raw_addr[19:10]+1)):raw_addr[19:10];
25
    //实例化D触发器得到area
27  dffre #(.n(1)) D2(.d(raw_addr[21]) , .en(1) , .r(0) , .clk(clk) , .q(
        area));

29  //取出原始数据
    sine_rom rom(.clk(clk) , .addr(rom_addr) , .dout(raw_data));
31
    //处理原始数据
33  assign data = area?(~raw_data+1):raw_data;

35  //实例化D寄存器得到取样值sample
    dffre #(.n(16)) D3(.d(data) , .en(sampling_pulse) , .r(0) , .clk(clk)
        , .q(sample));
37
    //实例化D触发器得到new_sample_ready信号
39  dffre #(.n(1)) D4(.d(sampling_pulse) , .en(1) , .r(0) , .clk(clk) , .q
        (new_sample_ready));

41  endmodule
```

Listing 6: 节拍计时器

```
1   module timer (
        r , en , clk ,  done , n
3   );
    parameter counter_bits = 6;
5   input [counter_bits-1:0] n;
    input r , en , clk;
7   output  done;
```

```
reg [counter_bits:1]q;
assign done = en && (q==n-1);
always @(posedge clk) begin
    if(r) q = 0;
    else begin
        if(en)
            begin
                if(q==n-1) q=0;
                else q = 1+q;
            end
        else begin
            q=q;
        end
    end
end
endmodule
```

Listing 7: note_player 模块

```
module note_player (
    input clk , reset , play_enable ,
    input [5:0] note_to_load,duration_to_load,
    input load_new_note,sampling_pulse,beat,
    output reg note_done,
    output [15:0] sample,
    output sample_ready);
    //状态编码
    parameter RESET = 0 , WAIT = 1 , DONE = 2 , LOAD = 3;
    reg [1:0] state , nextstate;

    reg timer_clear , load;
    wire timer_done;
    wire [5:0] addr;
    wire [19:0] klow;

    always @(posedge clk) begin
        if(reset) state = RESET;
        else state = nextstate;
    end

    always @(*) begin
        //默认状态
        timer_clear = 1 ; load = 0 ; note_done = 0;
```

```
25
        case (state)
27          RESET:begin
                timer_clear = 1 ; load = 0 ; note_done = 0;
29              nextstate = WAIT;
            end
31          WAIT:begin
                timer_clear = 0 ; load = 0 ; note_done = 0;
33              if(play_enable == 0) nextstate = RESET;
                else if(timer_done) nextstate = DONE;
35              else if(load_new_note) nextstate = LOAD;
                else nextstate = WAIT;
37          end
            DONE:begin
39              timer_clear = 1 ; load = 0 ; note_done = 1;
                nextstate = WAIT;
41          end
            LOAD:begin
43              timer_clear = 1 ; load = 1 ; note_done = 0;
                nextstate = WAIT;
45          end
        endcase
47  end

49  //实例化音符节拍定时器
    timer #(.counter_bits(6)) timer1(.clk(clk) , .en(beat) , .r(
       timer_clear) , .done(timer_done) , .n(duration_to_load));
51
    //实例一个D寄存器
53  dffre #(.n(6)) D(.d(note_to_load) , .en(load) , .r(~play_enable||reset
       ) , .clk(clk) , .q(addr));

55  //实例化FreqROM
    frequency_rom FreqROM(.clk(clk) , .dout(klow) , .addr(addr));
57
    //实例化DDS
59  dds DDS(.K({2'b00,klow}) , .clk(clk) , .reset(~play_enable||reset) , .
       sampling_pulse(sampling_pulse) , .sample(sample) , .
       new_sample_ready(sample_ready));

61 endmodule
```

4. 同步化电路

4.1 设计说明

因为音频解码接口模块和其他模块采用了不同的时钟，我们需要将二者进行同步化

4.2 代码

Listing 8: 同步化电路

```
//利用两个D触发器作为同步器使用
module synch (
    asynch_in , clk , asynch_out
);
    input asynch_in , clk;
    output asynch_out;
    wire Q1,Q2;
    dffre #(.n(1)) d1(.en(1) , .r(0) , .clk(clk) , .d(asynch_in) , .q(Q1))
       ;
    dffre #(.n(1)) d2(.en(1) , .r(0) , .clk(clk) , .d(Q1) , .q(Q2));
    assign asynch_out = Q1 & (~Q2);
endmodule
```

三 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 多媒体音视频智能互联开发系统。

(3) 有源音箱或耳机。

四 操作方法和实验步骤

(1) 按照书中提供的框图，将音乐播放器次顶层划分为一下几个模块：主控制器、乐曲读取、音符播放、同步化电路以及节拍基准产生器

(2) 参考实验 15 的资料，学习 DDS 技术相关知识，编写 DDS 模块并进行仿真

(3) 根据书上的指导，依次编写剩下模块并进行仿真验证，测试其是否符合要求

(4) 编写次顶层模块，并在其中设置参数 sim 方便仿真

(5) 新建 Vivado 工程，生成符合要求的 DCM 模块，将自己编写的模块以及老师提供的网表文件及接口文件加入工程。对工程进行综合、约束、实现，并下载到开发板中进行验证

五 实验数据记录和处理

1. DDS 模块

1.1 仿真图

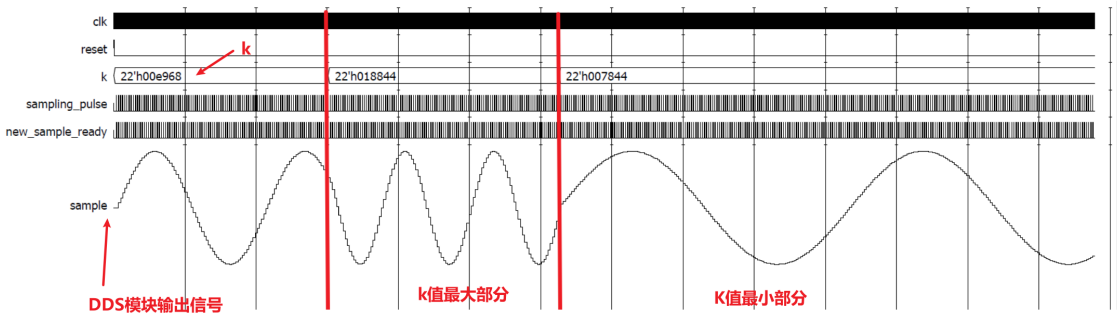

图 1: DDS 模块仿真图

1.2 结果分析

因为 $f_o = \frac{K \times f_c}{2^m}$，所以当 K 值增大时，DDS 模块输出的 f 值应该增加。
在图中可以看出，k 值越大，sample 信号峰值之间的距离越小，频率越大，符合 DDS 模块要求。

2. mcu 模块

2.1 仿真图

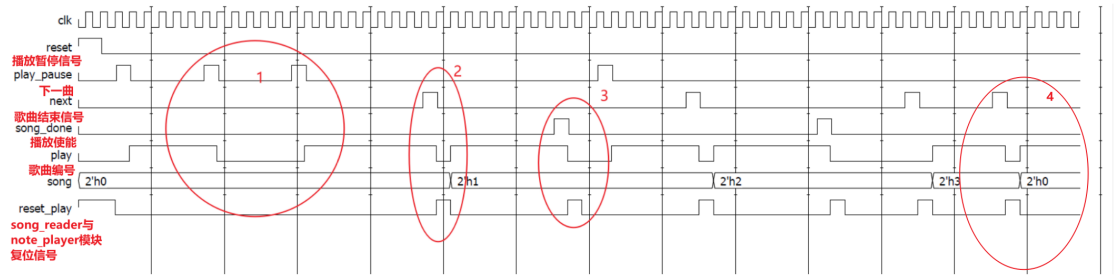

图 2: MCU 模块仿真图

2.2 结果分析

由圆圈 1 中波形可知，当输入一个 play_pause 信号后，mcu 的播放信号进行了反转，说明 play_-puase 信号能够正确控制乐曲播放的暂停与否；

由源泉 2 中波形可知，当按下了 next 按键输入了 next 信号后，首先 play 信号在下一时钟周期切换为 0，同时输出一个长度为一时钟周期的 reset_player 信号将 song_reader 与 note_player 模块复

位，在下一时钟周期，歌曲编号 song 增加 1，使 song_reader 模块开始读取下一首歌的音符。由此可见，mcu 控制切换下一首歌的功能正常。

由圆圈 3 中波形可知，当 mcu 得到来自 song_reader 的 song_done 信号后，能够将 play 信号切换到 0，并且输出一个 reset_play 信号将 song_reader 模块复位至乐曲的开始部分，由此，mcu 控制歌曲结束功能正常。

观察圆圈 4 中的 song 信号，我们可以发现在歌曲编号到 3 之后再次输入 next 信号会回到编号为 0 的歌曲，这保证了音乐播放器能够重复播放。

3. song_reader 模块

3.1 仿真图

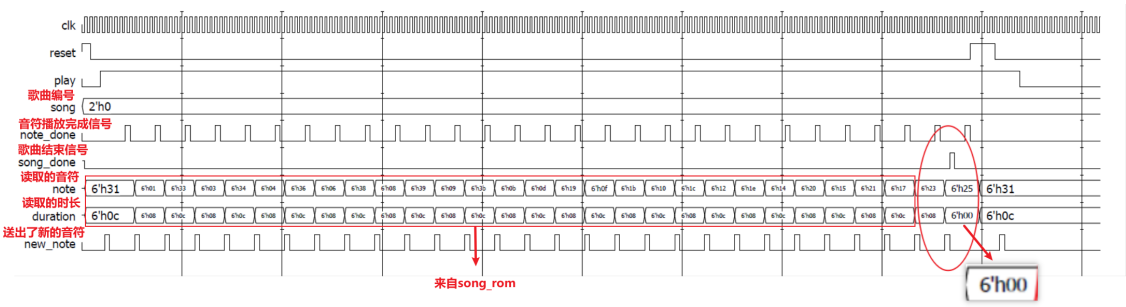

图 3: song_reader 模块仿真图

3.2 结果分析

观察波形图方框中的数值，将其与 song_rom 中 00-31 位置的值进行对比，可以发现值完全相等，这说明了 song_reader 能够正确地从 song_rom 中读取出音符，将其送入 note_player 模块；

观察最后的方框，可以发现，当 duration 为 0 时，song_reader 向 muc 模块发出来一个 song_done 信号。

上诉说明 song_reader 模块工作正常。

4. note_player 模块

4.1 仿真图

见图 4

4.2 结果分析

由上图可以发现，sample 信号的频率随着 note_to_load 信号的增大而逐渐增大，同时观察信号改变时经历的 beat 信号数目，可以发现是和 duration_to_load 相等的，同时在一个音符结束之后也会输出一个 note_done，说明 note_player 模块正常。

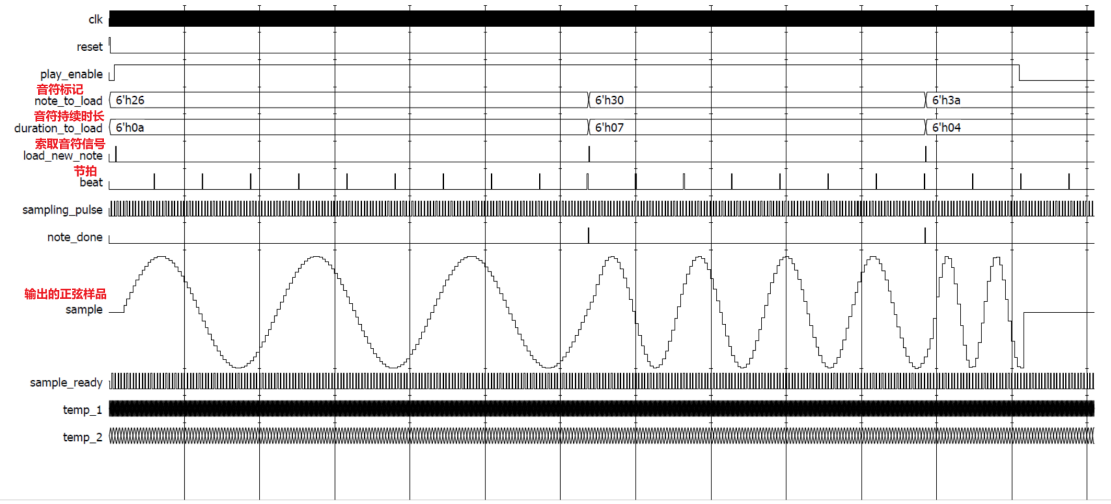

图 4: note_player 模块仿真图

5. 次顶层

5.1 仿真图

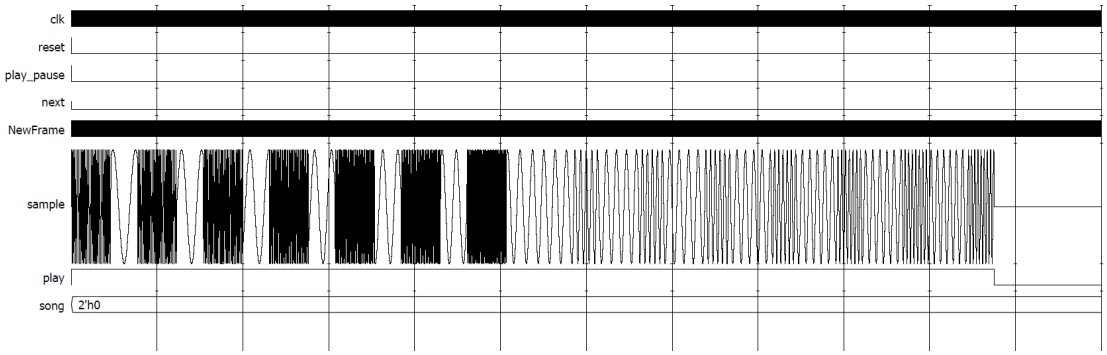

图 5: 次顶层模块仿真图

5.2 结果分析

和书上仿真结果对比，可以看出符和要求，次顶层工作正常

6. Vivado 上板测试

通过仿真图可以看出，各个模块工作均正常，达到了预期的工作效果，组合而成的次顶层模块仿真结果也与预期相同。

在 vivado 中生成二进制流上板测试后，经过调试 song_reader 模块中的结束判断模块，最终实现了所有的预期功能。

六 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？

reset 信号为复位信号，当 reset=1 后，系统内部已经完成复位。后续 reset 信号无论为 0 或者 1，对系统而言并没有影响，所以可以不对 reset 信号进行消颤动及同步化处理。

但是 next_button 、play_pause_button 信号的值不同会影响系统状态的转换。如果 next_button 不进行消颤动及同步化处理，系统可能会接收到很多个 next_button 信号，这就会导致跳过了很多首歌，无法达到切换下一首的目标；同理，如果 play_pause_button 不进行相应处理，会导致歌曲断断续续，不够连贯。不仅达不到预期效果，同时更会加剧系统的损耗

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？

存在。在 RESET->PAUSE 以及 NEXT->PLAY 状态切换的过程中，mcu 无法接收到按键输入。但是因为这两个状态都只有一个始终节拍，所以概率很小，没有必要修改设计。

七 心得体会

本次实验带给我的第一个感受就是完事开头难。相比于前面做的实验，这个实验无论是代码量还是难度都是远超其余实验的，而且数电实验书也不像前面几个实验的部分一样保姆级教程，使得我最开始拿到题目的时候不知道从何处开始。最终，花费了一个下午加一个晚上搞懂了 DDS 模块的原理以及完成了响应代码的编写，在 DDS 模块过了测试之后，顿时就对这个实验有了更多的信心。于是，在接下来的一天里，通过学习教材并不断试错，最终用半天时间完成了第一版代码的编写。

在编写各个模块的过程中，逐步体会到了什么叫做“自顶向下”的系统设计方法。采用这种方法设计系统时，能够很明白地理清各个模块之间的关系，而且由于各个模块都是相对独立的，在调试的时候也会更加简单，可以通过分析实验中反映出的各种信息初步得知是哪一个模块出了问题，改 bug 的时候也更加简单。同时，由于模块相对独立，也更加不容易出现牵一发而动全身的连锁反应。

当然，在本次实验中也遇到了以下两个主要问题：

(1) 春学期做实验的时候过于急功近利，导致没有打好基础。在拿到音乐播放器项目时一时陷入了茫然，不知从何处下手，最终通过从头学习前几次实验的代码，一步步地理解各个操作的含义，解决了这一问题。这一问题也让我意识到在学习一个新的知识时，不管它看上去有多简单，也不能够急功近利地不去理解，基础没打好后续一定会让你付出更加惨重的代价。

(2) song_done 模块由于结束判断没有做好始终不对。最开始我并没有认真阅读老师的课件，凭借着自己的“勇气”一头扎进了代码编写之中，忽视了结束判断的重要性，在上板调试的时候浪费了接近一个上午解决这一问题，最终还是屈老师指出了我的这一问题。在此感谢屈老师耐心解答我这一问题，后续认真学习了这个实验才知道这个问题是有多不应该。在懂了结束判断模块的编写方法后，我采用了将地址计数器和 duration 信号分别输入同一状态机进行处理的方案，最终虽然能够正常运行，但是状态机也显得过于复杂，不那么容易让人理解，后面旁听屈老师给别的同学讲解这一模块的编写时的思路让我明白了如何设计更加简洁的状态机，这也是为什么在我的 song_reader 模块里面会有 over 与 is_over 两个模块的原因。

除了以上两个主要问题，当然也有变量名写错、数据传输错误之类的小错误，在这一类错误里，最值得人注意的就是变量名写错，因为 modelsim 仿真时这类错误不会报错，所以在实例化各个模块时一定要仔细。

至于在这次实验中的收获，我觉得除了对 Verilog 语言的熟悉，更重要的是这次试验让我学会了踏踏实实地做事，不要妄想一步登天，还有就是学会了如何管理自己的文件。