实验名称：实验十 音频输出实验

姓名：张涵之

学号：191220154

班级：周一 5-6

邮箱：191220154@smail.nju.edu.cn

实验时间：2020/11/22

8.4 实验内容

请将之前实验实现的键盘与本实验的音频输出结合，实现一个简单的键盘电子琴功能。可以根据按下的键的键值，决定播放的正弦的频率，从而实现电子琴的功能。

基本要求：实现至少 8 个音符。支持每次按下单个按键。按键按下后开始发音，按键期间持续发音，松开后停止发音。按无关键不发音。无杂音和爆破音等干扰。

扩展要求：可调节音量。

实验目的：将键盘与音频输出结合，实现一个简单的键盘电子琴。

实验原理：

1）音频输出原理：音频设备如扬声器或耳机等所接收的音频信号是模拟信号，即时间上连续的信号。由于数字器件只能以固定的时间间隔产生输出，需要通过数字/模拟转换将数字信号转换成模拟信号输出。根据采样定律，数字信号的采样率应不低于信号频率的两倍。在实际信号输出时，我们一般不采用浮点数而选用整数值来表示每个样本点的大小。

生成频率为 f 的正弦波的过程如下：

1. 根据频率 f 计算递增值 d = f×65536/48000；
2. 在系统中维持一个 16bit 无符号整数计数器，每个样本点递增 d；
3. 根据计数器的高 10 位来获取查表地址 k，并查找 1024 点的正弦函数表；
4. 使用查表结果作为当前的数字输出。

实验提供了钢琴音高频率表如下：

表 10-4: 钢琴音高频率表

音名/八度	3	4	5	6
C	130.81Hz	261.63 Hz	523.25 Hz	1046.5 Hz
$C^{\#}$	138.59 Hz	277.18 Hz	554.37 Hz	1108.7 Hz
D	146.83 Hz	293.66 Hz	587.33 Hz	1174.7 Hz
$D^{\#}$	155.56 Hz	311.13 Hz	622.25 Hz	1244.5 Hz
E	164.81 Hz	329.63 Hz	659.26 Hz	1318.5 Hz
F	174.61 Hz	349.23 Hz	698.46 Hz	1396.9 Hz
$F^{\#}$	185.00 Hz	369.99 Hz	739.99 Hz	1480.0 Hz
G	196.00 Hz	392.00 Hz	783.99 Hz	1568.0 Hz
$G^{\#}$	207.65 Hz	415.30 Hz	830.61 Hz	1661.2 Hz
A	220.00 Hz	440.00 Hz	880.00 Hz	1760.0 Hz
$A^{\#}$	233.08 Hz	466.16 Hz	932.33 Hz	1864.7 Hz
B	246.94 Hz	493.88 Hz	987.77 Hz	1975.5 Hz

则可列表出 C5，D5，… ，B5，C6 等音符对应的递增值、字母和键码：

字符	Q	W	E	R	T	Y	U	I
键码	15	1d	24	2d	2c	35	3c	43
音名	C5	D5	E5	F5	G5	A5	B5	C6
频率	523.25	587.33	659.26	698.46	783.99	880.00	987.77	1046.5
递增	714	802	900	954	1070	1201	1349	1429

字符	A	S	D	F	G	H	J	K
键码	1c	1b	23	2b	34	33	3b	42
音名	C#5	D#5		F#5	G#5	A#5		C#6
频率	554.37	622.25		739.99	830.61	932.33		1108.7
递增	757	850		1010	1134	1273		1514

2）音频接口：生成每个时间点上的音频波形后，信号通过音频接口送给耳机或者扬声器。

3）I2S 接口：通过 I2S 接口来发送和接收数字音频信号。音频信号的基准时钟 AUD_XCK 设为采样频率的 256 倍或 384 倍。本实验中采样频率为 48kHz，AUD_XCK 设为其 384 倍，即 48000 × 384=18.432MHz。调用 Quartus 提供的标准 IP 库来产生这类特殊的时钟。
在生成完时钟信号后，只需要按要求将每个样本点生成的 16bit 有符号整数数据按高位前发送即可。可以将左右声道设置为一样的数据，以实现单声道播放。

4）音量调整：查表可得寄存器低七位为音量，调节范围为 0110000~1111111。

0000010 Left Headphone Out	6:0	LHPVOL [6:0]	1111001 (0dB)	Left Channel Headphone Output Volume Control 1111111 = +6dB . . 1dB steps down to 0110000 = -73dB 0000000 to 0101111 = MUTE
	7	LZCEN	0	Left Channel Zero Cross detect Enable 1 = Enable 0 = Disable
	8	LRHPBOTH	0	Left to Right Channel Headphone Volume, Mute and Zero Cross Data Load Control 1 = Enable Simultaneous Load of LHPVOL[6:0] and LZCEN to RHPVOL[6:0] and RZCEN 0 = Disable Simultaneous Load

可以在 keyboard 上层模块 key2note 中通过键码的识别控制和修改 volume，将 volume 作为参数传入 I2C_Audio_Config 模块，在 always 语句中修改 audio_cmd 的值。

程序代码或流程图：
ps2_keyboard 获取键码 —> \
key2note 根据键码/音名/频率关系得出递增值 —> \
Sin_Generator 根据递增值得到 audiodata —> \
I2C_Audio_Config 与 I2S_Audio 控制音频元件 —> \
audio_clk 和 clkgen 生成时钟 —> 顶层模块综合

audio_clk //调用 Quartus 标准 IP 库产生的 18.432MHz 的时钟
clk_gen //用开发板上的 CLOCK_50 生成 10k 的 I2C 时钟
ps2_keyboard //键盘控制器模块，用于从键盘获取输入（键码）

key2note　　//键盘上层控制模块，通过键码判断音调和音量

```
module key2note(
    input clk,
    input clrn,
    output reg en,
    output reg [15:0] freq,
    inout ps2_clk,
    inout ps2_dat,
    output reg [8:0] volumn
);

    initial begin
        volumn = 9'b000111000;
    end

    wire [7:0] code;
    wire ready,overflow;
    reg nextdata = 0;
    reg flag = 0;
    reg [7:0] scancode;

    ps2_keyboard board(clk,clrn,ps2_clk,ps2_dat,code,ready,nextdata,overflow);

    always @ (posedge clk) begin
        if (ready && nextdata == 1'b1) begin
            nextdata <= 0;
            scancode <= code;
            if (code == 8'hf0) flag <= 1;
            else begin
                if (flag) begin
                    en <= 0;
                    flag <= 0;
                    if (code == 8'h41) volumn[6:0] = volumn[6:0] - 9'd1;
                    if (code == 8'h49) volumn[6:0] = volumn[6:0] + 9'd1;
                end
                else en <= 1;
            end
        end
        else nextdata <= 1;
    end

    always @(scancode) begin
        case (scancode)
            8'h15  : freq = 16'd714;
            8'h1d  : freq = 16'd802;
            8'h24  : freq = 16'd900;
            8'h2d  : freq = 16'd954;
            8'h2c  : freq = 16'd1070;
            8'h35  : freq = 16'd1201;
            8'h3c  : freq = 16'd1349;
            8'h43  : freq = 16'd1429;

            8'h1c  : freq = 16'd757;
            8'h1b  : freq = 16'd850;
            8'h2b  : freq = 16'd1010;
            8'h34  : freq = 16'd1134;
            8'h33  : freq = 16'd1273;
            8'h42  : freq = 16'd1514;

            default: freq = 16'd0;
        endcase
    end

endmodule
```

//clk 为时钟信号，clrn 为清零信号
//nextdata，ready 和 overflow 即键盘控制器中的 nextdata_n，ready 和 overflow
//scancode 用于暂存从键盘获得的键码，volumn 用于音量控制
//en 为是否发出声音的使能控制端，在顶层模块中被调用
//flag 为调节 en 的控制信号，在按键松开（接受 0xf0 后一个键值）时 en 置零
其中 QWERTYUI 为 C5，D5，…，C6，下面为对应的升调（如果有的话）
音量控制为按下松开按键，<(,) 表示音量减小，>(.) 表示音量增大。

Sin_Generator　　//根据递增值生成相应的正弦波信号
I2C_Controller　　//I2C 接口单个命令发送

I2C_Audio_Config　　//音频芯片配置示例

```
module I2C_Audio_Config ( clk_i2c,
                          reset_n,
                          I2C_SCLK,
                          I2C_SDAT,
                          volumn);
    parameter  total_cmd = 9;

    input clk_i2c;  //10k I2C clock
    input reset_n;
    input [8:0] volumn;
```

//加入了输入 volumn，用于音量大小的控制
//接下来试了很多种方法，都没有成功

I2S_Audio　　//与开发板上的音频模块交互
exp10　　//顶层综合模块，读取按键，选择音调并调节音量播放

```
//=======================================================
//  REG/WIRE declarations
//=======================================================

wire clk_i2c;
wire reset,en;
wire [15:0] audiodata;
wire [15:0] freq_out;
wire [15:0] freq;
wire [8:0] volumn;

//=======================================================
//  Structural coding
//=======================================================

assign reset = ~KEY[0];
assign LEDR[8:0] = volumn[8:0];
assign freq = freq_out & {16{en}};

audio_clk u1(CLOCK_50,reset,AUD_XCK,LEDR[9]);
key2note key(CLOCK_50,SW[0],en,freq_out,PS2_CLK,PS2_DAT,volumn);

//I2C part
clkgen #(10000) my_i2c_clk(CLOCK_50,reset,1'b1,clk_i2c);  //10k I2C clock
I2C_Audio_Config myconfig(clk_i2c,KEY[0],FPGA_I2C_SCLK,FPGA_I2C_SDAT,volumn);
I2S_Audio myaudio(AUD_XCK,KEY[0],AUD_BCLK,AUD_DACDAT,AUD_DACLRCK,audiodata);
Sin_Generator sin_wave(AUD_DACLRCK,KEY[0],freq,audiodata);//

endmodule
```

//用按钮 KEY[0]控制重置，SW[0]控制键盘使能，LEDR[8:0]显示音量
//当 en 为 1 时说明按键按下了，根据声音频率得到相应递增量，否则置零

实验环境/器材：实验箱一个，笔记本电脑一台，键盘一个。

实验步骤/过程：
设计键盘按键和音名的对应关系，查表得到频率，计算得到对应的递增量；
根据实验八的键盘模块进行适当修改，实现键码到递增量的转换和音量选择；
阅读和理解示例工程中的代码，将其添加到自己的工程中；
对示例代码进行适当修改，增加参数和调整语句以实现音量控制；
在顶层模块中调用上述几个模块，实现键盘输入、获取音调、调节音量等功能。

测试方法：按下键盘，从耳机中听到声音。
实验结果：耳机中音调的输出符合预期，音量始终无法更改。

实验中遇到的问题及解决办法：感到提供的音频实现代码十分难懂。
　　解决办法：多读几遍，思考每个变量的含义和作用，在顶层模块中该如何调取。
换了好几种方法都没能实现音量控制。

解决方法：首先将 initial 模块改成 always，试图在其中直接赋值：

```
always
begin
    audio_reg[0]= 7'h0f; audio_cmd[0]=9'h0;  //reset
    audio_reg[1]= 7'h06; audio_cmd[1]=9'h0;  //Disable Power Down
    audio_reg[2]= 7'h08; audio_cmd[2]=9'h2;  //Sampling Control
    audio_reg[3]= 7'h02; audio_cmd[3]=volumn; //Left Volume
    audio_reg[4]= 7'h03; audio_cmd[4]=volumn; //Right Volume
    audio_reg[5]= 7'h07; audio_cmd[5]=9'h1;  //I2S format
    audio_reg[6]= 7'h09; audio_cmd[6]=9'h1;  //Active
    audio_reg[7]= 7'h04; audio_cmd[7]=9'h16; //Analog path
    audio_reg[8]= 7'h05; audio_cmd[8]=9'h06;  //Digital path

end
```

效果：似乎仍然只有初始化的时候赋了一次值，后续没有任何更改；

增加了一个 always 模块，当 volumn 发生变化时进行赋值

```
always @ (volumn) begin
    audio_cmd[3]=volumn; //Left Volume
    audio_cmd[4]=volumn; //Right Volume
end
```

效果：根本就没有声音了！这就是课件里说的“如果试图在同一周期内读取或写入这个 RAM 中的两个地址的数据，系统将无法综合这块 RAM 的硬件，并且不会报错，直接结果就是编译通过但没有声音”吗！改回去了。

其次怀疑生成的 volumn 参数有问题，将其接到二极管上显示：

```
assign reset = ~KEY[0];
assign LEDR[8:0] = volumn[8:0];
assign freq = freq_out & {16{en}};
```

效果：按下音量键时二极管显示发生了变化，看来 volumn 参数有修改；

接下来怀疑对参数的理解有问题，查手册看寄存器含义：

效果：只有后七位是表示音量的，前面两位也不知道在干啥。

总之音量的调节范围是 0110000~1111111，看起来没什么问题；

怀疑是每次增量过小听不出来，在 initial 模块分别赋差别很大的值，听声音效果：

效果：至少赋值 0110000 和 1111111 区别很大，一个听不见，一个快聋了

二极管跳来跳去声音不变，说明还是我的代码有问题

在 I2C_Audio_Config 下面的 case 语句里面给 mi2c_data 直接赋值，如：

```
2'd1: begin
          if(cmd_count == 4'd3)
             mi2c_data <= {audio_addr, audio_reg[cmd_count], 9'b000110000};
          else if(cmd_count == 4'd4)
             mi2c_data <= {audio_addr, audio_reg[cmd_count], 9'b001111111};
          else begin
             mi2c_data <= {audio_addr, audio_reg[cmd_count], audio_cmd[cmd_count]};
             mi2c_go   <= 1'b1;
             mi2c_state<= 2'd2;
          end
    end
```

效果：仍然没有任何变化！左右声道都还是 initial 里面的初始值！

它到底有没有运行到这里过！还是我对 mi2c_data 的写入有什么误解！

综上所述，因为我的朋友全都没做出来，也没人可以问，所以我也不做了。

实验得到的启示：学会放弃。

意见和建议：尽管手册里有“请在设计时注意读取和写入需要按 RAM 的操作规范进行”，遗憾的是，我至今仍然不能十分理解 RAM 的操作规范，也不知道问题出在哪里。

参考代码已经实现了大部分的功能，但对于学生来说阅读和理解这些代码都在干啥是很费劲的，修改和扩展这些代码的时候往往也很忐忑，明明需要写的代码量很少，却让人感觉很累而且学不到东西。也许可以出一些简单的、参考代码较少、学生自主性更强的实验？