实验十

音频输出实验

实验报告　　日期：2020 年 11 月 20 日

姓名： 朱嘉琦

学号： 191220185

班级： 数电实验一班

邮箱： 1477194584@qq.com

实验十报告—— 音频输出实验

191220185 朱嘉琦

一、实验目的

声音是一种重要的人机交互手段。音频信号可以通过扬声器或者耳机输 出，由麦克风输入计算机。在输入和输出过程中一般需要对信号进行数字/模拟或模拟/数字转换。

本实验的主要目的是学习音频信号的输出方式以及如何将数字信号转换为模拟信号的基本原理。

二、实验原理

音频输出原理

人耳可以听到的声音的频率范围是 20-20kHz。音频设备如扬声器或耳机等所接收的音频信号一般是模拟信号，即时间上连续的信号。但是，由于数字器件只能以固定的时间间隔产生数字输出，我们需要通过数字/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（每秒钟产生的 数字样本数量）应不低于信号频率的两倍。所以，数字音频一般采用 44.1kHz (CD 音频) 或 48kHz 的采样率，以保证 20kHz 的信号不会失真。

对于 一个正弦波信号 $s(t)$，其数学表达式是：

$$s(t) = sin(2\pi ft)$$

其中 f 为频率，t 为时间。当采样率是 48kHz 时，每两个点之间的间隔是 1/48 毫秒。此时，我们可以将 t 改写成 $t = n/48000$ 秒。这样式子变为

$$s(n) = sin(2\pi fn/48000)$$

代入 f = 960Hz, 我们得到

$$s(n) = sin(\frac{2\pi \times 960n}{48000} = sin(\frac{2\pi n}{50}))$$

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。这个过程称为量化 (Quantization)。假设我们用带符号的 16bit 整数 （补码）来表示单个样本点，此时 32767 即对应输出的最大值（例如 +1V 电 压），-32768 即对应输出的最小值（例如-1V 电压）。这时，我们就可以通过循环输出 50 个点的整数值 s'(n) = round (s(n) × 32767) 来产生一个 sin 波形，如 表 10-1所示：

表 10–1: 960Hz 数字信号示例

n	0	1	2	3	. . .	48	49	50	51
$s(n)$	0	0.125	0.249	0.368	. . .	-0.249	-0.125	0	0.125
$\bar{s}(n)$	0	4107	8149	12062	. . .	-8149	-4107	0	4107

首先我们需要存储器中存储一张 1024 点的 sin 函数表。即存储器中以地址 k =0. . .1023 存储了 1024 个三角函数值（以 16bit 补码整数表示），地址为 k 的数值设置为

$$round(sin(\frac{2\pi k}{1024}) \times 32767)$$

对于任意频率为 f 的正弦波，我们在第 n 个样本点需要输出的值为

$$round(sin(\frac{2\pi n f}{48000}) \times 32767)$$

但是，在 FPGA 中要计算乘除法及取整操作耗费资源较多，我们实际应用 中采取累加的方法。我们观察到 n 每增加 1，对应的 k 会增加$\frac{f\times1024}{48000}$。

因此，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 $d = \frac{f\times65536}{48000}$ 。
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 16 位无符号整数计数器的高 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。

音频接口

在生成完每个时间点上的音频波形后，我们需要将音频信号通过音频接口送给耳机或者扬声器。

DE10-Standard 开发板上的音频接口

DE10-Standard 开发板上集成了一块 WM8731 音频编解码芯片，其参考手册可以在课程网站上找到。该音频编解码芯片提供 24bit 的音频接口，支持 8kHz 到 96kHz 的采样率。在我们的实验中仅考虑 48kHz 采样率，每个样本点 16 比特的情况。FPGA 和音频编码器的接口如图 10-2所示。

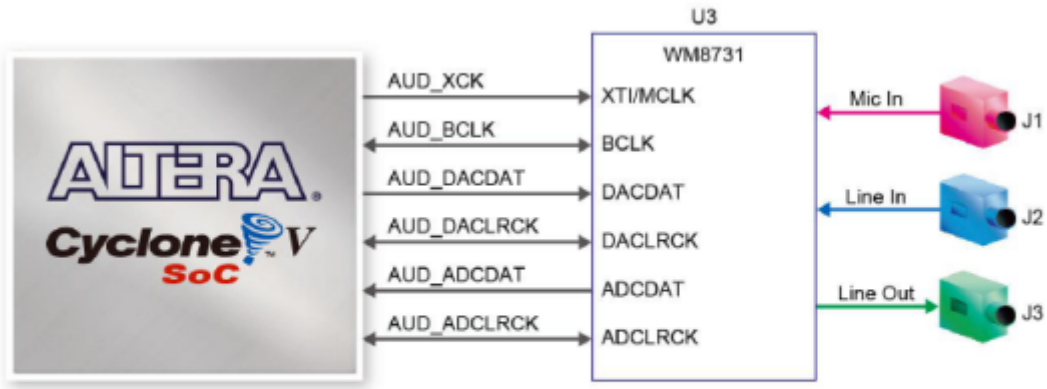


图 10-2: FPGA 和编解码芯片的接口

FPGA 与音频编解码芯片的接口包括两大部分。一部分是控制接口，该接口是利用通用 I^2C 总线实现的。该接口的功能主要是在音频编解码芯片的控制 寄存器内写入配置信息，控制音频编解码芯片的工作方式。另一部分是音频信号接口，主要是通过 I^2S 音频协议实现的，包括 DAC 和 ADC 两个方向。在本实验中只使用 DAC 输出音频信号的方向。在实验中用到的音频接口引脚如图 10-3 所示。

Signal Name	FPGA Pin No.	Description	I/O Standard
AUD_ADCLRCK	PIN_AH29	Audio CODEC ADC LR Clock	3.3V
AUD_ADCDAT	PIN_AJ29	Audio CODEC ADC Data	3.3V
AUD_DACLRCK	PIN_AG30	Audio CODEC DAC LR Clock	3.3V
AUD_DACDAT	PIN_AF29	Audio CODEC DAC Data	3.3V
AUD_XCK	PIN_AH30	Audio CODEC Chip Clock	3.3V
AUD_BCLK	PIN_AF30	Audio CODEC Bit-stream Clock	3.3V
I2C_SCLK	PIN_Y24 or PIN_E23	I2C Clock	3.3V
I2C_SDAT	PIN_Y23 or PIN_C24	I2C Data	3.3V

图 10-3: FPGA 和编解码芯片的接口引脚配置

I^2C接口

I^2C是一种常用的集成电路总线。在本实验中，我们利用I^2C来设置音频芯片。I^2C采用两根双向的漏极开路线来实现多个设备之间的近距离通信。所谓漏极开路是指输出端上通过一个上拉电阻与 VCC 相连，这时如果输出为低态时，信号就会被导通至低电平。而当输出端是断开时，信号被上拉电阻保持在高电平。这时，可以将多个设备连在同一根线上，只要有一个设备为低态，该总线即为低态，实现“线与” 的功能。

I^2C接口的两根线分别是数据线 SDIN（或 SDAT）和时钟线 SCLK。通信 中一般分为主节点和从节点。主节点产生时钟信号并发起对从节点的通信，在我们的实验中 FPGA 为主节点，WM8731 音频编解码芯片为从节点。时钟速率缺省为 100kbps(bit/s)。但在我们的实验中采用 10kbps 的低速模式即可。

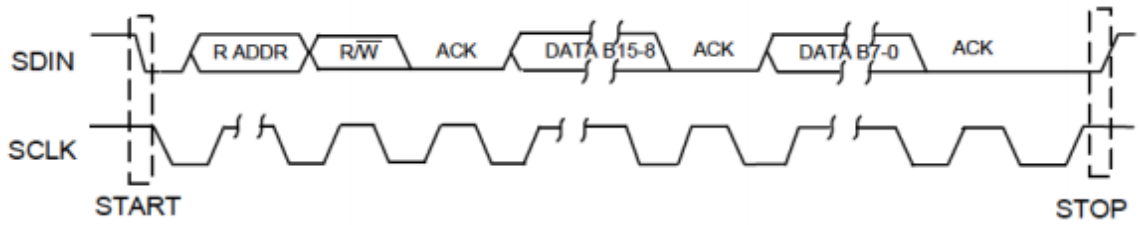


图 10-4: I²C 接口基本时序

图 10-4显示了I^2C接口的基本通信方式。主节点首先拉低 SDIN 数据线， 并保持 SCLK 为高电平，发出起始信号。随后，主节点拉低 SCLK，并将第一 个数据位放在 SDIN 上，从节点在 SCLK 的上升沿接收第一个 bit 数据。主节点首先会发送 7 个 bit 的地址来寻找从节点（高位 MSB 先发送）。在我们的板子上音频芯片的地址是 7’b0011010。地址之后，主节点会发送一个 bit 的 R/W 信号，低电平表示写入寄存器，高电平表示读取。在本实验中，音频芯片只接受写入命令。因此，此位总是置低。随后，主节点会将 SDIN 置为高阻一个周期，在这个周期里，被选中的从节点可以发送 ACK 信号。ACK 信号总是低电平。随后，主节点发送 2 次 8bit 的数据，每次数据发送完后主节点都会高阻一个周期，由从节点发送 1bit 的 ACK 信号。最终，主节点拉低 SDIN 一个周期后将 SDIN 和 SCLK 置为高阻，完成停止位的发送。

I^2S接口

在完成对 WM8731 的配置后，我们就可以通过I^2S接口来发送和接收数字音频信号了。I^2S接口包括 AUD_XCK, AUD_BCLK, AUD_DACDAT, AUD_DACLRCK, AUD_ADCDAT, AUD_ADCLRCK 等多条信号线。其中AUD_XCK 为音频信号的基准时钟，AUD_BCLK 为音频数据每个比特同步时钟，AUD_DACDAT 为输出数字信号数据，AUD_DACLRCK 用于输出的左右声道同步。

音频信号的基准时钟 AUD_XCK 一般设置为采样频率的 256 倍或者 384 倍。在我们的实验中，我们将采样频率设置为 48kHz，AUD_XCK 设为其 384 倍。因此，AUD_XCK 为 48000 × 384 = 18.432MHz。

这里我们需要调用 Quartus 提供的标准 IP 库来产生这类特殊的时钟。在 IP 库中选择 `Library→Basic Functions→Clocks;PLLs and Resets→PLL→Altera PLL` 该 IP 核使用锁相环产生时钟，可以生成特殊频率。

三、实验环境/器材

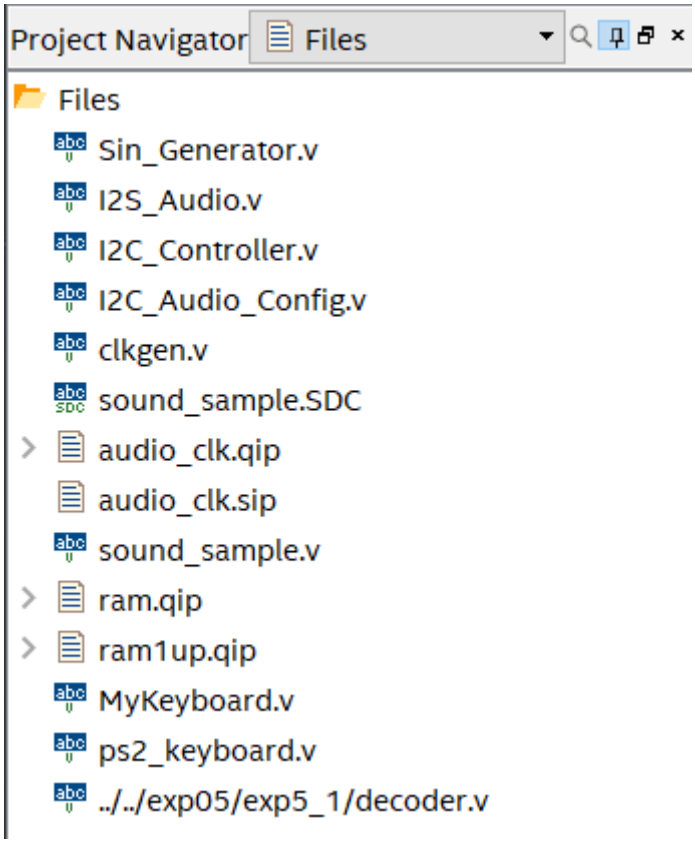
实验环境是Quarters 17.1 Lite，实验器材是DE10 开发板

四、程序代码或流程图

1. 工程文件

本次实验涉及到多个模块

SOUND_SAMPLE



2. 音频控制接口模块

1）I2S_Audio.v

```
module I2S_Audio(AUD_XCK,
                 reset_n,
                 AUD_BCK,
                      AUD_DATA,
                      AUD_LRCK,
```

```
                        audiodata);
input AUD_XCK;
input reset_n;
output reg AUD_BCK;
output AUD_DATA;
output reg AUD_LRCK;
input [15:0] audiodata;


reg [7:0] bck_counter;
reg [7:0] lr_counter;
wire [7:0] bitaddr;

//generate BCK 1.536MHz
always @ (posedge AUD_XCK or negedge reset_n)
begin
       if(!reset_n)
         begin
            bck_counter <= 8'd0;
             AUD_BCK     <= 1'b0;
         end
         else
         begin

             if(bck_counter >= 8'd5) //div XCK by 12
             begin
                 bck_counter <= 8'd0;
                  AUD_BCK <= ~AUD_BCK;
             end
             else
               bck_counter <= bck_counter + 8'd1;
         end

end

//generate LRCK, 48kHz, at negedge of BCK

always @ (negedge AUD_BCK or negedge reset_n)
begin
       if(!reset_n)
         begin
            lr_counter <= 8'd0;
             AUD_LRCK     <= 1'b0;
         end
         else
         begin

             if(lr_counter >= 8'd15) //div BCK by 32
             begin
                 lr_counter <= 8'd0;
                  AUD_LRCK <= ~AUD_LRCK;
             end
             else
               lr_counter <= lr_counter + 8'd1;
        end

end
```

```
//send data, input data avaible at posedge of lrclk, prepared at the posedge of
bck

assign  bitaddr  = 8'd15- lr_counter;
assign  AUD_DATA = audiodata[bitaddr[3:0]];



endmodule
```

2) I2C_Controller.v

```
module I2C_Controller (
    CLOCK,
    I2C_SCLK,//I2C CLOCK
    I2C_SDAT,//I2C DATA
    I2C_DATA,//DATA:[SLAVE_ADDR,SUB_ADDR,DATA]
    GO,      //GO transfor
    END,     //END transfor
    W_R,     //W_R
    ACK,      //ACK
    RESET_N,
    //TEST
    //SD_COUNTER,
    //SDO
);
    input  CLOCK;
    input  [23:0]I2C_DATA;
    input  GO;
    input  RESET_N;
    input  W_R;
    inout  I2C_SDAT;
    output I2C_SCLK;
    output END;
    output reg [2:0] ACK;

//TEST
//  output [5:0] SD_COUNTER;
//  output SDO;


reg SDO;
reg SCLK;
reg END;
reg [23:0]SD;
reg [5:0]SD_COUNTER;

wire I2C_SCLK=SCLK | ( ((SD_COUNTER >= 4) & (SD_COUNTER <=30))? ~CLOCK :0 );
wire I2C_SDAT=SDO?1'bz:0 ;

reg ACK1,ACK2,ACK3;

//--I2C COUNTER
always @(negedge RESET_N or posedge CLOCK ) begin
if (!RESET_N) SD_COUNTER=6'b111111;
else begin
if (GO==0)
```

```
    SD_COUNTER=0;
    else
    if (SD_COUNTER < 6'b111111) SD_COUNTER=SD_COUNTER+1;
end
end
//----

always @(negedge RESET_N or  posedge CLOCK ) begin
if (!RESET_N) begin SCLK=1;SDO=1; ACK1=0;ACK2=0;ACK3=0; END=1; end
else
case (SD_COUNTER)
    6'd0  : begin ACK1=0 ;ACK2=0 ;ACK3=0 ; END=0; SDO=1; SCLK=1; ACK=3'b0; end
    //start
    6'd1  : begin SD=I2C_DATA;SDO=0;end
    6'd2  : SCLK=0;
    //SLAVE ADDR
    6'd3  : SDO=SD[23];
    6'd4  : SDO=SD[22];
    6'd5  : SDO=SD[21];
    6'd6  : SDO=SD[20];
    6'd7  : SDO=SD[19];
    6'd8  : SDO=SD[18];
    6'd9  : SDO=SD[17];
    6'd10 : SDO=SD[16];
    6'd11 : SDO=1'b1;//ACK

    //SUB ADDR
    6'd12  : begin SDO=SD[15]; ACK1=I2C_SDAT; end
    6'd13  : SDO=SD[14];
    6'd14  : SDO=SD[13];
    6'd15  : SDO=SD[12];
    6'd16  : SDO=SD[11];
    6'd17  : SDO=SD[10];
    6'd18  : SDO=SD[9];
    6'd19  : SDO=SD[8];
    6'd20  : SDO=1'b1;//ACK

    //DATA
    6'd21  : begin SDO=SD[7]; ACK2=I2C_SDAT; end
    6'd22  : SDO=SD[6];
    6'd23  : SDO=SD[5];
    6'd24  : SDO=SD[4];
    6'd25  : SDO=SD[3];
    6'd26  : SDO=SD[2];
    6'd27  : SDO=SD[1];
    6'd28  : SDO=SD[0];
    6'd29  : SDO=1'b1;//ACK


    //stop
    6'd30 : begin SDO=1'b0; SCLK=1'b0; ACK3=I2C_SDAT; end
    6'd31 : begin SCLK=1'b1; ACK={ACK1,ACK2,ACK3}; end
    6'd32 : begin SDO=1'b1; END=1; end

endcase
end
```

```
endmodule
```

3) I2C_Audio_Config.v

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

    input clk_i2c;  //10k I2C clock
    input reset_n;
     input [1:0]Volume;

    output I2C_SCLK;
    output [2:0] testbit;
    inout  I2C_SDAT;

    reg [23:0] mi2c_data;
    reg  mi2c_go;
    wire mi2c_end;
    reg  [1:0] mi2c_state; //state 0: stop, state 1: sendnext;
                           //state 2: wait for finish, state 3:move index
    wire [2:0] mi2c_ack;
    wire [7:0] audio_addr;

    reg [3:0] cmd_count;
    reg [6:0] audio_reg [15:0]; //register to write
    reg [8:0] audio_cmd [15:0]; //register content


initial
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]=9'h79; //Left Volume used: 79
    audio_reg[4]= 7'h03; audio_cmd[4]=9'h79; //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


assign audio_addr={7'b0011010,1'b0}; //WM8731 addr, always write
assign testbit = cmd_count[2:0];


I2C_Controller  u0  (   .CLOCK(clk_i2c),        //  Controller Work Clock
                        .I2C_SCLK(I2C_SCLK),        //  I2C CLOCK
```

```
                        .I2C_SDAT(I2C_SDAT),        //  I2C DATA
                        .I2C_DATA(mi2c_data),       //  DATA:
[SLAVE_ADDR,SUB_ADDR,DATA]
                        .GO(mi2c_go),               //  GO transfor
                        .END(mi2c_end),             //  END transfor
                        .ACK(mi2c_ack),             //  ACK
                        .RESET_N(reset_n)   );

always @ (posedge clk_i2c or negedge reset_n)
begin
     if(!reset_n)
      begin
         cmd_count  <= 4'b0;
         mi2c_state <= 2'b0;
         mi2c_go    <= 1'b0;
          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;  //Left Volume
            audio_reg[4]= 7'h03;  //Right Volume
            case (Volume)
                2'b00: begin audio_cmd[3]=9'h49;audio_cmd[4]=9'h49; end
                2'b01: begin audio_cmd[3]=9'h59;audio_cmd[4]=9'h59; end
                2'b10: begin audio_cmd[3]=9'h69;audio_cmd[4]=9'h69; end
                2'b11: begin audio_cmd[3]=9'h79;audio_cmd[4]=9'h79; end
                default: begin audio_cmd[3]=9'h79;audio_cmd[4]=9'h79; end
            endcase
            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
      else
      begin
         case(mi2c_state)
          2'd0: begin  //stop
                    if(cmd_count ==4'b0)
                           mi2c_state <= 2'd1;
                end
          2'd1: begin
                    mi2c_data <= {audio_addr, audio_reg[cmd_count],
audio_cmd[cmd_count]};
                        mi2c_go    <= 1'b1;
                        mi2c_state<= 2'd2;
                end
          2'd2: begin
                     if(mi2c_end)
                         begin
                                mi2c_state <= 2'd3;
                                  mi2c_go    <= 1'b0;
                         end
                end
          2'd3: begin
                    cmd_count <= cmd_count + 4'd1;
                        if(cmd_count + 4'd1 < total_cmd)
                           mi2c_state <= 2'd1;  //start next
                       else
                           mi2c_state <= 2'd0;  //last cmd
```

```
                end
            endcase
        end

end


endmodule
```

4）分频器clkgen.v

```
module clkgen(
    input clkin,
    input rst,
    input clken,
    output reg clkout
    );
    parameter clk_freq=1000;
    parameter countlimit=50000000/2/clk_freq; //
   //integer countlimit=8388;
   reg[31:0] clkcount;
   always @ (posedge clkin)
    if(rst)
     begin
        clkcount=0;
        clkout=1'b0;
     end
    else
    begin
     if(clken)
        begin
            clkcount=clkcount+1;
            if(clkcount>=countlimit)
             begin
                clkcount=32'd0;
                clkout=~clkout;
             end
            else
                clkout=clkout;
         end
      else
        begin
             clkcount=clkcount;
             clkout=clkout;
        end
     end
endmodule
```

3. 键盘模块

1）ps2_keyboard.v

```
module ps2_keyboard(clk,clrn,ps2_clk,ps2_data,data,ready,nextdata_n,overflow);
    input clk,clrn,ps2_clk,ps2_data;
    input nextdata_n;
    output [7:0] data;
```

```
    output reg ready;
    output reg overflow; // fifo overflow
    // internal signal, for test
    reg [9:0] buffer; // ps2_data bits
    reg [7:0] fifo[7:0]; // data fifo
    reg [2:0] w_ptr,r_ptr; // fifo write and read pointers
    reg [3:0] count; // count ps2_data bits
    // detect falling edge of ps2_clk
    reg [2:0] ps2_clk_sync;

    always @(posedge clk) begin
        ps2_clk_sync <= {ps2_clk_sync[1:0],ps2_clk};
    end

    wire sampling = ps2_clk_sync[2] & ~ps2_clk_sync[1];

    always @(posedge clk) begin
        if (clrn == 0) begin // reset
            count <= 0; w_ptr <= 0; r_ptr <= 0; overflow <= 0; ready<= 0;
        end
        else begin
            if (ready) begin // read to output next data
                if(nextdata_n == 1'b0) //read next data
                begin
                    r_ptr <= r_ptr + 3'b1;
                    if(w_ptr==(r_ptr+1'b1)) //empty
                        ready <= 1'b0;
                end
            end
            if (sampling) begin
                if (count == 4'd10) begin
                    if ((buffer[0] == 0) &&     // start bit
                        (ps2_data)          &&  // stop bit
                        (^buffer[9:1])) begin   // odd parity
                        fifo[w_ptr] <= buffer[8:1]; // kbd scan code
                        w_ptr <= w_ptr+3'b1;
                        ready <= 1'b1;
                        overflow <= overflow | (r_ptr == (w_ptr + 3'b1));
                    end
                    count <= 0; // for next
                end else begin
                    buffer[count] <= ps2_data; // store ps2_data
                    count <= count + 3'b1;
                end
            end
        end
    end
    assign data = fifo[r_ptr]; //always set output data

endmodule
```

2) MyKeyboard.v

```
module MyKeyboard(input clk,
                  input clrn,
                  input ps2_clk,
```

```
                    input ps2_data,
                    //output reg [7:0] data,
                    output [7:0] ascii,
                    output [6:0] d_digit_low,
                    output [6:0] d_digit_high,
                    output reg [6:0] a_digit_low,
                    output reg [6:0] a_digit_high,
                    output [6:0] c_digit_low,
                    output [6:0] c_digit_high,
                    //output reg[15:0] count,
                    output reg ctrl,
                    output reg shift,
                    output reg up,
                    output reg caps
                    );

reg [7:0] data;
reg[15:0] count;
reg clk_t;
reg ctrl_f,shift_f,caps_f;
integer count_clk;
reg nextdata_n;
wire ready;
wire overflow;
wire [7:0] temp_data;
//wire [7:0] ascii;
wire [7:0] ascii_u;
wire [6:0] tmp_digit_low1,tmp_digit_high1,tmp_digit_low2,tmp_digit_high2;
reg pre;

initial
begin
    clk_t=0;
    count_clk=0;
    nextdata_n=0;
    count=8'b00000000;
    data=8'b00000000;
    pre=1;
    ctrl=0;
    ctrl_f=0;
    up=0;
    shift=0;
    shift_f=0;
    caps=0;
    caps_f=0;
end

//divider
always @ (posedge clk) begin
    if(count_clk>=100) begin
        count_clk<=0;
        clk_t=~clk_t;
    end
    else
        count_clk=count_clk+1;
end

//get ps2 info
```

```
    ps2_keyboard ps2(
                        .clk(clk),
                        .clrn(clrn),
                        .ps2_clk(ps2_clk),
                        .ps2_data(ps2_data),
                        .data(temp_data),
                        .ready(ready),
                        .nextdata_n(nextdata_n),
                        .overflow(overflow)
                    );

    //trans key code to ascii
    ram r1  (
                        .address(data),
                        .clock(clk),
                        .data(8'b00000000),
                        .rden(1'b1),
                        .wren(1'b0),
                        .q(ascii)
                );
    ram1up r2(
                        .address(data),
                        .clock(clk),
                        .data(8'b00000000),
                        .rden(1'b1),
                        .wren(1'b0),
                        .q(ascii_u)
                );
    //trans data to HEX
    /*
    decoder d1(data[3:0],d_digit_low);
    decoder d2(data[7:4],d_digit_high);

    decoder d3(ascii[3:0],tmp_digit_low1);
    decoder d4(ascii[7:4],tmp_digit_high1);
    decoder d3u(ascii_u[3:0],tmp_digit_low2);
    decoder d4u(ascii_u[7:4],tmp_digit_high2);

    decoder d5(count[3:0],c_digit_low);
    decoder d6(count[7:4],c_digit_high);
    */

    decoder d1({~pre,data[3:0]},d_digit_low);
    decoder d2({~pre,data[7:4]},d_digit_high);

    decoder d3({~pre,ascii[3:0]},tmp_digit_low1);
    decoder d4({~pre,ascii[7:4]},tmp_digit_high1);
    decoder d3u({~pre,ascii_u[3:0]},tmp_digit_low2);
    decoder d4u({~pre,ascii_u[7:4]},tmp_digit_high2);

    decoder d5({1'b0,count[3:0]},c_digit_low);
    decoder d6({1'b0,count[7:4]},c_digit_high);
    always @(posedge clk) begin

        if(up==0) begin
            a_digit_low<=tmp_digit_low1;
            a_digit_high<=tmp_digit_high1;
        end
```

```
        else begin
            a_digit_low<=tmp_digit_low2;
            a_digit_high<=tmp_digit_high2;
        end

    end

    always @ (posedge clk_t) begin
        if(ready==1)
        begin
            if(temp_data[7:0]==8'h58) begin //caps
                if((pre==1)&&caps==0) begin
                    up=~up;
                    caps=1;
                end
                else if(pre==0) begin
                    caps=0;
                end
            end

            if(temp_data[7:0]==8'h12||temp_data[7:0]==8'h59) begin  //shift
                if(pre==1&&shift==0) begin
                    up=~up;
                    shift=1;
                end
                else if(pre==0) begin
                    up=~up;
                    shift=0;
                end
            end

            if(temp_data[7:0]==8'h14) begin     //ctrl
                if(pre==1&&ctrl==0) begin
                    ctrl=1;
                end
                else if(pre==0) begin
                    ctrl=0;
                end
            end


            if((temp_data[7:0]!=8'hf0)&&(pre==1)) begin //keep the data
                pre=1;
                data=temp_data;
            end
            else if(temp_data[7:0]==8'hf0) begin
                pre=0;
                count=count+1;
                data=temp_data;
            end
            else if(pre==0)
                pre=1;

            nextdata_n=0;
        end
        else
            nextdata_n=1;
    end
```

```
endmodule
```

4. 顶层模块

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module sound_sample(

    ////////////// CLOCK //////////
    input                       CLOCK2_50,
    input                       CLOCK3_50,
    input                       CLOCK4_50,
    input                       CLOCK_50,

    ////////////// KEY //////////
    input            [3:0]      KEY,

    ////////////// SW //////////
    input            [9:0]      SW,

    ////////////// LED //////////
    output           [9:0]      LEDR,

    ////////////// Seg7 //////////
    output           [6:0]      HEX0,
    output           [6:0]      HEX1,
    output           [6:0]      HEX2,
    output           [6:0]      HEX3,
    output           [6:0]      HEX4,
    output           [6:0]      HEX5,

    ////////////// Audio //////////
    input                       AUD_ADCDAT,
    inout                       AUD_ADCLRCK,
    inout                       AUD_BCLK,
    output                      AUD_DACDAT,
    inout                       AUD_DACLRCK,
    output                      AUD_XCK,

    ////////////// PS2 //////////
    inout                       PS2_CLK,
    inout                       PS2_CLK2,
    inout                       PS2_DAT,
    inout                       PS2_DAT2,

    ////////////// I2C for Audio and Video-In //////////
    output                      FPGA_I2C_SCLK,
    inout                       FPGA_I2C_SDAT
);
```

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

reg [15:0] d_new;
reg [15:0] d_old;
reg [15:0] freq;
reg [15:0] freq_x;

wire ctrl;
wire shift;
wire up;
wire caps;

wire clk_i2c;
wire reset;
wire [15:0] audiodata;

wire [7:0] ascii;


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

assign reset = ~KEY[0];

audio_clk u1(CLOCK_50, reset,AUD_XCK, LEDR[9]);

MyKeyboard(         .clk(CLOCK2_50),
                        .clrn(1'b1),
                        .ps2_clk(PS2_CLK),
                        .ps2_data(PS2_DAT),
                        //output reg [7:0] data,
                        .ascii(ascii),
                        .d_digit_low(HEX0),
                        .d_digit_high(HEX1),
                        .a_digit_low(HEX2),
                        .a_digit_high(HEX3),
                        .c_digit_low(HEX4),
                        .c_digit_high(HEX5),
                        //output reg[15:0] count,
                        .ctrl(ctrl),
                        .shift(shift),
                        .up(up),
                        .caps(caps)
                        );

//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],SW[1:0],FPGA_I2C_SCLK,FPGA_I2C_SDAT,LEDR[2:0]);
```

```
I2S_Audio myaudio(AUD_XCK, KEY[0], AUD_BCLK, AUD_DACDAT, AUD_DACLRCK,
audiodata);

Sin_Generator sin_wave(AUD_DACLRCK, KEY[0], freq,freq_x,caps, audiodata);//

always @ (posedge CLOCK_50) begin
    case(ascii[7:0])
    8'h31:begin d_new <= 523.25; if(!ctrl) d_old <= 523.25; end
    8'h32:begin d_new <= 587.23; if(!ctrl) d_old <= 587.23; end
    8'h33:begin d_new <= 659.26; if(!ctrl) d_old <= 659.26; end
    8'h34:begin d_new <= 698.46; if(!ctrl) d_old <= 698.46; end
    8'h35:begin d_new <= 783.99; if(!ctrl) d_old <= 783.99; end
    8'h36:begin d_new <= 880;     if(!ctrl) d_old <= 880;    end
    8'h37:begin d_new <= 987.77; if(!ctrl) d_old <= 987.77; end
    8'h38:begin d_new <= 1046.5; if(!ctrl) d_old <= 1046.5; end
    default:begin d_new <=0; end
    endcase
end

always @ (CLOCK_50)
begin
    freq = d_new*65536/48000;
    if (d_new==0)
        freq_x = 0;
    else
        freq_x = d_old*65536/48000;
end


endmodule
```

五、实验步骤

- 根据实验讲义学习音频的输出原理，理解如何根据声音的频率计算给音频芯片的值。
- 阅读讲义中I^2C和I^2S接口的相关代码，理解各个输入和输出之间的关系。

- 首先直接编译包含分频器、I^2C和I^2S接口的相关代码的工程文件，检查是否有语法错误，最终编译通过。连接到FGPA实验板上，发现可以发出恒定的声音，证明这些模块可以正常工作。
- 在代码中修改Sin_Generator中freq的值以及I2C_Audio_Config中audio_cmd中的值，尝试修改声音频率和分贝，利用实验板再次验证。
- 在顶层模块中设计一个小的状态机，根据ascii码的值分配状态，定义了两个值d_new和d_old，用来记录前后两个声音的频率，再根据d的值，利用课件中的公式计算出freq的值，将其作为参数传入sin_generator模块中，或者声音数据。
- 检查是否有语法错误，最终编译通过。连接到FGPA实验板上，可以根据键盘的输入输出对应的声音频率，已经能够实现基础功能。
- 接下来开始实现高级功能，现在看来声音响度的控制稍微简单一点，先做控制音量：音量的值在I2C_Audio_Config模块中，首先为其设计一个Volume输入，其次要修改audio_cmd的值，但不能简单得给其赋值，因为audio_reg 是一块 RAM，系统只能综合在给定时钟沿和给定地址条件下读取或写入这个 RAM。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。请在设计时注意对 audio_cmd 的读取和写入需要按 RAM 的操作规范进行。所以在模块中利用一个always语句，在时钟周期内为写入整块 RAM 的值。

```
if(!reset_n)
      begin
         cmd_count  <= 4'b0;
         mi2c_state <= 2'b0;
         mi2c_go    <= 1'b0;
            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;  //Left Volume
            audio_reg[4]= 7'h03;  //Right Volume
            case (Volume)
                2'b00: begin audio_cmd[3]=9'h49;audio_cmd[4]=9'h49; end
                2'b01: begin audio_cmd[3]=9'h59;audio_cmd[4]=9'h59; end
                2'b10: begin audio_cmd[3]=9'h69;audio_cmd[4]=9'h69; end
                2'b11: begin audio_cmd[3]=9'h79;audio_cmd[4]=9'h79; end
                default: begin audio_cmd[3]=9'h79;audio_cmd[4]=9'h79; end
            endcase
            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
```

- 检查是否有语法错误，最终编译通过。连接到FGPA实验板上，可以实现根据两位SW输入实现四种大小的音量输出。
- 其次要实现和声，由于之前已经准备了d_new和d_old，我利用ctrl作为控制键，当ctrl按下的时候才处理和声，利用d_new和d_old这两个值计算为freq后传入sin_gen，在sin_gen中根据两个频率计算出其freq_counter（为带符号整数直接相加会溢出， 要按比例缩小数值后再相加。）并在mif文件中取其dataout并输出。
- 检查是否有语法错误，最终编译通过。连接到FGPA实验板上，自此可以实现音量控制和和声，所有高级功能全部实现。

六、测试方法

1. 将耳机连接到FPGA的音频输出端，编译后将sof文件导入开发板中实际测试电路首先测试没有键盘模块的音频。
2. 将键盘模块加入，编译后将sof文件导入开发板中实际测试利用键盘输入控制音频输出。
3. 最后在代码中添加功能，实现控制音量和和声的功能。

七、实验结果——简单电子琴

可用键盘输入数字1~8控制音频输出，共有八个音高，用SW[1:0]控制声音大小，同时按住CTRL和1~8其中的两个可以实现输出和声。（完成所有高级功能并且已经给助教验收）

八、思考

我一开始直接就从键盘模块开始做，完全根据自己阅读课件的理解写，发现连接后完全没有反应，而且debug也无从下手，在看了debug建议后，我从头开始，先去除键盘模块，从单一的声音输出开始实验，逐步添加功能，就顺利的将简单电子琴做了出来，所以在以后做更庞大的项目时一定要从最基础的功能开始，将模块精简化，自顶向下逐步设计，才能顺利debug。

九、实验中遇到的问题及解决方法

键盘输入模块我设计的比较顺利，但在修改声音大小的时候遇到了比较大的障碍，虽然课件中提醒了audio_cmd的修改一定要注意，但我也仅是在时钟周期内为audio_cmd[3]和[4]赋值，发现没用声音，后来我在always语句中为整个RAM赋值后解决了这个问题。

十、 意见与建议

无