

## 实验十 音频输出实验

2020 年秋季学期

然而在整整六次这样的变调之后，原来的 *C* 小调又魔术般地恢复了！所有的声部都恰好比原来高八度。

– 《哥德尔、艾舍尔、巴赫 – 集异壁之大成》，Douglas R. Hofstadter

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。

### 10.1 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。

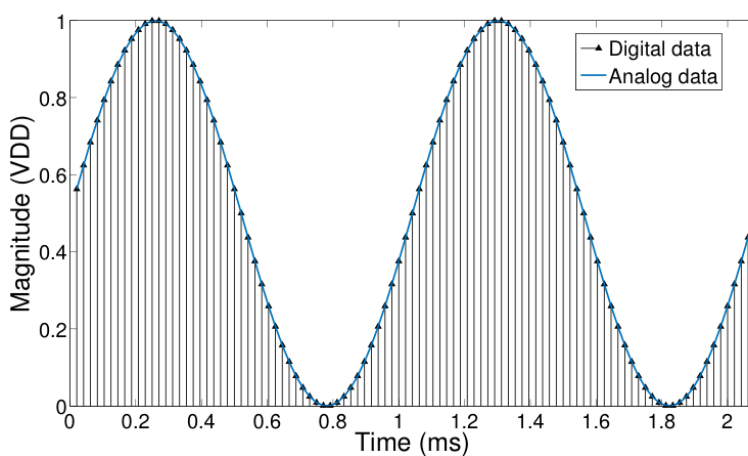


图 10-1: 数字信号到模拟信号的转换

图 10-1 显示了数字信号到模拟信号转换的基本原理。在 48kHz 的采样率下，我们每秒输出 48000 次，即每间隔 1/48000 秒（1/48 毫秒）的时间产生一个数字输出样本点，如图中黑色小三角所示。该输出经过平滑后，会产生一个对应的模拟信号。

假设我们需要产生一个人耳能听到的单频率  $f = 960\text{Hz}$  的正弦波信号，我们需要在合适的时间点上设置（或输出）合适的数字值来形成正弦波形。对于一个正弦波信号  $s(t)$ ，其数学表达式是：

$$s(t) = \sin(2\pi ft) \quad (10-1)$$

其中  $f$  为频率， $t$  为时间。在数字信号中，我们用整数  $n$  来标记各个数字样本，样本编号顺序依次为  $n = 0, 1, 2, 3, 4, \dots$ 。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 (10-1) 就变成：

$$s(n) = \sin(2\pi fn/48000) \quad (10-2)$$

代入  $f = 960\text{Hz}$ ，我们得到  $s(n) = \sin\left(\frac{2\pi \times 960n}{48000}\right) = \sin\left(\frac{2\pi n}{50}\right)$ 。所以，对于整数  $n$  来说，每 50 个点对应正弦波的一个周期。仔细观察图 10-1 中的 960Hz 正弦波可知，其周期为 1/960 秒（1.042 毫秒），对应 50 个样本点。

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）<sup>①</sup>。这时，我们就可以通过循环输出 50 个点的整数值  $\bar{s}(n) = \text{round}(s(n) \times 32767)$  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

这时，如果我们用一个计数器不停从 0 至 49 计数作为  $n$ ，用查找表输出  $n$

<sup>①</sup>图 10-1 中的电压范围是 0~1V，直流（均值）为 0.5V。在本实验中建议使用直流（均值）为 0V 的正弦波。

对应的  $\delta(n)$  整数值就可以不断产生类似图 10-1 的连续 960Hz 正弦波。量化过程中采用的整数位数越多，信号的精度也就更高。

上例中的正弦波是固定频率为 960Hz 的。实际应用中，我们如果要产生不同频率的正弦波，就不能采用简单计数直接查表的方式，而需要先按频率计算出样本点对应的相位，然后查三角函数表获取对应幅度值的方式。

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  $k=0\dots1023$  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

$$\text{round}\left(\sin\left(\frac{2\pi k}{1024}\right) \times 32767\right) \quad (10-3)$$

感兴趣的同学可以用高级语言生成该函数表，我们也提供了一张 1024 点  $\sin$  函数表的 mif 文件。

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

$$\text{round}\left(\sin\left(\frac{2\pi n f}{48000}\right) \times 32767\right) \quad (10-4)$$

比较式 (10-3) 和 (10-4)，我们得到  $\frac{k}{1024} = \frac{n f}{48000}$ 。因此，在我们的函数表中最接近这个值的表项应该是  $k = \text{round}\left(\frac{n f \times 1024}{48000}\right) \bmod 1024$ 。函数表中的项目越多，查找到的函数值越精确，但一般情况下人的耳朵往往可以容忍 1%-5% 的误差，所以我们使用 1024 点的函数表已经基本够用。

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  $\frac{f \times 1024}{48000}$ 。这样，我们可以通过每个样本点将  $k$  递增  $\frac{f \times 1024}{48000}$  来避免乘除法。例如， $f = 960\text{Hz}$  时，我们可以每个样本点对  $k$  递增  $\frac{960 \times 1024}{48000} = 20.48$ 。这样，50 个点后正好  $k$  增加了 1024，完成一个周期。但是，我们这里的  $k$  是整数，如何能够每次递增一个小数值呢？这里，我们可以采用定点小数的方式，即用 16bit 来表示  $k$ 。其中前 10bit 是整数部分，用来查三角函数表，后面 6bit 是小数部分，用来提高精度。这时，如果将此 16bit 数看成是一个无符号整数的话，每个周期是 65536，而对应前 10 比特循环的周期是 1024 点。因此， $n$  每增加 1，我们需要  $k$  递增  $\frac{960 \times 65536}{48000} = 1311$ ，对应的小数值是  $1311/64 = 20.4843$ 。从整数的角度来看， $n$  变化 50 个样本点时  $k$  递增了  $1311 \times 50 = 65550$ ，这个值略大于 65536 一些，会对周期带来一些小的误差，但是这样的误差对于人耳来说是可以容忍的。

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  $d = \frac{f \times 65536}{48000}$ 。
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  $d^{\textcircled{1}}$ 。
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。

## 10.2 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。

### 10.2.1 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。

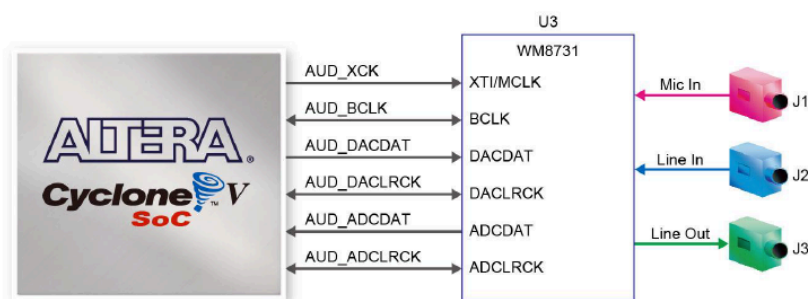


图 10-2: FPGA 和编解码芯片的接口

FPGA 通过音频数字模拟转换 (DA) 和模拟数字转换 (AD) 接口来与音频编解码芯片通信。FPGA 输出的数字样本信号被转换成模拟信号 (DA 转换)，通过绿色的 Line Out 接口输出。在实验中使用耳机接入 Line Out 接口。同时，系统还支持通过 Mic In 或 Line In 接口接入麦克风或模拟信号，将其转换为数字信号 (AD 转换) 然后发送给 FPGA。在本实验中我们仅考虑输出音频的情况。对录音感兴趣的同学可以自行研究。

<sup>①</sup>FPGA 中的无符号数递增溢出时直接丢掉溢出位，所以等价于取模，可以直接用于循环生成三角函数表地址。

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<sup>2</sup>C 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<sup>2</sup>S 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_DACLCK	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

在音频发送的实现上同样也分为两个步骤。第一步，是**配置音频编解码芯片**。这步有点像我们打开空调时，首先要用遥控器调整温度，风量，冷热模式这些参数。在数字系统中，大量的设置是通过读写设备上或芯片上的寄存器来实现的。每个寄存器会有一个独立的地址，并会有自己特定的功能。CPU 或主控单元通过在寄存器上写入特定的值来将设备设置到特定状态，或者让设备执行特定的动作。例如，**要对音频芯片进行 reset，只需要在地址 0F 的寄存器中写入 00 即可**。在操作系统中的设备驱动程序中，也通常使用写入寄存器的方式来对外设进行控制。通过对寄存器的读取可以了解到设备当前的状态，也是设备向主控发送信息的一种方式。在本实验中我们通过 I<sup>2</sup>C 接口对音频编解码芯片进行设置，只有在设置完全正确时，音频流才能够正常输出到耳机中。不幸的是，板载的音频芯片不支持读取操作，我们只能对芯片进行设置而无法获取芯片信息。调试时如果出现问题，需要尝试一系列不同的设置。

第二步是**音频流发送**。如 10.1 所述，我们的**音频流是一串音频的电压数字**。我们需要通过 I<sup>2</sup>S 接口将这串数字传输给音频芯片。传输的具体格式是在第一步中通过寄存器设置的。一般音频是包括左右两个声道的，我们**需要以 48kHz 的速率发送两个声道的数据给音频芯片**。在设置正确时，音频芯片就会将对应的数字值转换为耳机的输出，放出声音来。

## 10.2.2 I<sup>2</sup>C 接口

I<sup>2</sup>C 是一种常用的集成电路总线<sup>①</sup>。在本实验中，我们利用 I<sup>2</sup>C 来设置音频芯片。I<sup>2</sup>C 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。具体可以参考数字设计教科书 3.7.4 节。

I<sup>2</sup>C 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。

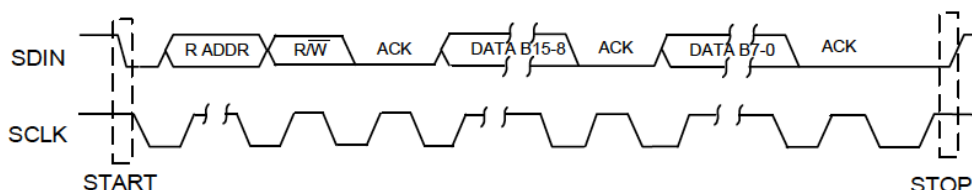


图 10-4: I<sup>2</sup>C 接口基本时序

图 10-4 显示了 I<sup>2</sup>C 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/ $\overline{W}$  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。

对于 WM8731 芯片，我们通过 I<sup>2</sup>C 接口发送的 16bit 数据（分为两个 8bit）主要是用来设置芯片的配置寄存器。其中前 7 个比特为寄存器地址（高位 MSB

<sup>①</sup>另一种常见的类似接口是 SPI 接口。



先发送），后 9 个 bit 为对寄存器设置的值。通过设置这些寄存器，我们可以调节设备的音频通道、改变音量、调整采样率和音频数据格式等等。具体的寄存器说明请参考我们提供的 WM8731 芯片手册。

REGISTER	BIT[8]	BIT[7]	BIT[6]	BIT[5]	BIT[4]	BIT[3]	BIT[2]	BIT[1]	BIT[0]	DEFAULT
<b>R0 (00h)</b> Left Line In	LRINBOTH	LINMUTE	0	0	LINVOL[4:0]					0_1001_0111
<b>R1 (01h)</b> Right Line In	RLINBOTH	RINMUTE	0	0	RINVOL[4:0]					0_1001_0111
<b>R2 (02h)</b> Left Headphone Out	LRHPBOTH	LZCEN	LHPVOL[6:0]							0_0111_1001
<b>R1 (01h)</b> Right Headphone Out	RLHPBOTH	RZCEN	RHPVOL[6:0]							0_0111_1001
<b>R4 (04h)</b> Analogue Audio Path Control	0	SIDEATT[1:0]		SIDETONE	DACSEL	BYPASS	INSEL	MUTEMIC	MICBOOST	0_0000_1010
<b>R5 (05h)</b> Digital Audio Path Control	0	0	0	0	HPOR	DACMU	DEEMPH[1:0]		ADCHPD	0_0000_1000
<b>R6 (06h)</b> Power Down Control	0	POWEROFF	CLKOUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD	0_1001_1111
<b>R7 (07h)</b> Digital Audio Interface Format	0	BCLKINV	MS	LRSWAP	LRP	IWL[1:0]		FORMAT[1:0]		0_1001_1111
<b>R8 (08h)</b> Sampling Control	0	CLKODIV2	CLKIDIV2	SR[3:0]				BOSR	USB/ NORMAL	0_0000_0000
<b>R9 (09h)</b> Active Control	0	0	0	0	0	0	0	0	Active	0_0000_0000
<b>R15 (0Fh)</b> Reset	RESET[8:0]									not reset

图 10-5: WM8731 的寄存器含义

表 10-2 为 DE10-Standard 开发版提供的 I<sup>2</sup>C 接口代码。调用一次该代码可以发送单个命令（将特定数据写入一个寄存器）。该接口通过 GO 信号指示进行发送（GO 需要在发送期间保持高电平），END 信号拉高说明发送结束。CLOCK 可以使用 10kHz 的时钟，I2C\_DATA 是上层模块提供的待发送 24bit 数据，包含芯片地址、读写位、寄存器地址、寄存器值等。接口中可以添加信号 SD\_COUNTER 和 SDO 用于测试模块内部的状态。ACK 信号返回发送后从节点的 3 个 ACK 比特。

表 10-2: I<sup>2</sup>C 接口单个命令发送代码

```
1 module I2C_Controller (  
2     CLOCK,  
3     I2C_SCLK, //I2C CLOCK  
4     I2C_SDAT, //I2C DATA  
5     I2C_DATA, //DATA:[SLAVE_ADDR,SUB_ADDR,DATA]  
6     GO,       //GO transfor  
7     END,      //END transfor
```

```
8         ACK,          //ACK
9         RESET_N,
10        //TEST
11        //SD_COUNTER,
12        //SDO
13    );
14    input  CLOCK;
15    input  [23:0] I2C_DATA;
16    input  GO;
17    input  RESET_N;
18    inout  I2C_SDAT;
19    output I2C_SCLK;
20    output END;
21    output reg [2:0] ACK;
22    //TEST
23    //output [5:0] SD_COUNTER;
24    //output SDO;
25
26    reg SDO;
27    reg SCLK;
28    reg END;
29    reg [23:0] SD;
30    reg [5:0] SD_COUNTER;
31
32    //SET CLK and SDIN
33    wire I2C_SCLK=SCLK | ( ((SD_COUNTER>=4)&(SD_COUNTER<=30)) ? ~CLOCK:0 );
34    wire I2C_SDAT=SDO?1'bz:0 ;
35
36    reg ACK1,ACK2,ACK3;
37
38    //--I2C COUNTER
39    always @(negedge RESET_N or posedge CLOCK ) begin
40        if (!RESET_N) SD_COUNTER=6'b111111;
41        else begin
42            if (GO==0)
43                SD_COUNTER=0;
44            else
```



```

45         if (SD_COUNTER < 6'b111111) SD_COUNTER=SD_COUNTER+1;
46     end
47 end
48 //-----
49
50 always @(negedge RESET_N or posedge CLOCK ) begin
51     if (!RESET_N) begin SCLK=1;SDO=1; ACK1=0;ACK2=0;ACK3=0; END=1; end
52     else
53     case (SD_COUNTER)
54         6'd0 : begin
55             ACK1=0 ;ACK2=0 ;ACK3=0 ; END=0;
56             SDO=1; SCLK=1; ACK=3'b0;
57             end
58         //start
59         6'd1 : begin SD=I2C_DATA;SDO=0; end
60         6'd2 : SCLK=0;
61         //SLAVE ADDR
62         6'd3 : SDO=SD[23];
63         6'd4 : SDO=SD[22];
64         6'd5 : SDO=SD[21];
65         6'd6 : SDO=SD[20];
66         6'd7 : SDO=SD[19];
67         6'd8 : SDO=SD[18];
68         6'd9 : SDO=SD[17];
69         6'd10 : SDO=SD[16]; //WR
70         6'd11 : SDO=1'b1; //wait for ACK
71
72         //SUB ADDR
73         6'd12 : begin SDO=SD[15]; ACK1=I2C_SDAT; end
74         6'd13 : SDO=SD[14];
75         6'd14 : SDO=SD[13];
76         6'd15 : SDO=SD[12];
77         6'd16 : SDO=SD[11];
78         6'd17 : SDO=SD[10];
79         6'd18 : SDO=SD[9];
80         6'd19 : SDO=SD[8];
81         6'd20 : SDO=1'b1; // wait for ACK

```

```
82
83     //DATA
84     6'd21 : begin SDO=SD[7]; ACK2=I2C_SDAT; end
85     6'd22 : SDO=SD[6];
86     6'd23 : SDO=SD[5];
87     6'd24 : SDO=SD[4];
88     6'd25 : SDO=SD[3];
89     6'd26 : SDO=SD[2];
90     6'd27 : SDO=SD[1];
91     6'd28 : SDO=SD[0];
92     6'd29 : SDO=1'b1; //ACK
93
94
95     //stop
96     6'd30 : begin SDO=1'b0;      SCLK=1'b0; ACK3=I2C_SDAT; end
97     6'd31 : begin SCLK=1'b1; ACK={ACK1,ACK2,ACK3}; end
98     6'd32 : begin SDO=1'b1; END=1; end
99
100 endcase
101 end
102
103 endmodule
```

表 10-3 是一个简单的音频芯片配置示例。其在每次 reset 后根据预置的命令顺序配置音频芯片。该配置设置了正常音量 (0dB)，使用 48kHz 双声道每个声道 16bit 的 I<sup>2</sup>S 通信设置，并且将 I<sup>2</sup>S 的 XCLK 设置为 18.432MHz (48kHz 的 384 倍)。

表 10-3: 音频芯片配置示例

```

1 module I2C_Audio_Config ( clk_i2c,
2                           reset_n,
3                           I2C_SCLK,
4                           I2C_SDAT);
5     parameter total_cmd = 9; //sending 9 commands
6
7     input clk_i2c;  //10k I2C clock
8     input reset_n;
9     output I2C_SCLK;
10    inout I2C_SDAT;
11
12    reg [23:0] mi2c_data;
13    reg mi2c_go;
14    wire mi2c_end;
15    reg [1:0] mi2c_state; //state 0: stop, state 1: send next;
16                        //state 2: wait for finish, state 3: move index
17    wire [2:0] mi2c_ack;
18    wire [7:0] audio_addr;
19    reg [3:0] cmd_count;
20    reg [6:0] audio_reg [15:0]; //register to write
21    reg [8:0] audio_cmd [15:0]; //register content
22
23    initial //predefine all the commands
24    begin
25        audio_reg[0] = 7'h0f; audio_cmd[0] = 9'h0; //reset
26        audio_reg[1] = 7'h06; audio_cmd[1] = 9'h0; //Disable Power Down
27        audio_reg[2] = 7'h08; audio_cmd[2] = 9'h2; //Sampling Control
28        audio_reg[3] = 7'h02; audio_cmd[3] = 9'h79; //Left Volume
29        audio_reg[4] = 7'h03; audio_cmd[4] = 9'h79; //Right Volume
30        audio_reg[5] = 7'h07; audio_cmd[5] = 9'h1; //I2S format
31        audio_reg[6] = 7'h09; audio_cmd[6] = 9'h1; //Active

```

---

```
32     audio_reg[7]= 7'h04; audio_cmd[7]=9'h16; //Analog path
33     audio_reg[8]= 7'h05; audio_cmd[8]=9'h06; //Digital path
34 end
35
36
37 assign audio_addr={7'b0011010,1'b0}; //WM8731 addr, always write
38
39 I2C_Controller u0(.CLOCK(clk_i2c),      //Controller Work Clock
40                  .I2C_SCLK(I2C_SCLK), //I2C CLOCK
41                  .I2C_SDAT(I2C_SDAT), //I2C DATA
42                  .I2C_DATA(mi2c_data), //DATA:[SLAVE_ADDR,SUB_ADDR,DATA]
43                  .GO(mi2c_go),          //GO transfer
44                  .END(mi2c_end),        //END transfer
45                  .ACK(mi2c_ack),        //ACK
46                  .RESET_N(reset_n)      );
47
48 always @ (posedge clk_i2c or negedge reset_n)
49 begin
50     if(!reset_n)
51     begin
52         cmd_count  <= 4'b0;
53         mi2c_state <= 4'b0;
54         mi2c_go    <= 1'b0;
55     end
56     else
57     begin
58         case(mi2c_state)
59             2'd0: begin //stop
60                 if(cmd_count ==4'b0)
61                     mi2c_state <= 2'd1;
62             end
63             2'd1: begin
64                 mi2c_data <= {audio_addr, audio_reg[cmd_count],
65                             audio_cmd[cmd_count]};
66                 mi2c_go    <= 1'b1;
67                 mi2c_state<= 2'd2;
68             end
69         end
```

```

69     2'd2: begin
70         if(mi2c_end)
71             begin
72                 mi2c_state <= 2'd3;
73                 mi2c_go    <= 1'b0;
74             end
75         end
76     2'd3: begin
77         cmd_count <= cmd_count + 4'd1;
78         if(cmd_count + 4'd1 < total_cmd)
79             mi2c_state <= 2'd1; //start next
80         else
81             mi2c_state <= 2'd0; //last cmd
82         end
83     endcase
84 end
85 end
86
87 endmodule

```

### 10.2.3 I<sup>2</sup>S 接口

在完成对 WM8731 的配置后，我们就可以通过 I<sup>2</sup>S 接口来发送和接收数字音频信号了。I<sup>2</sup>S 接口包括 AUD\_XCK, AUD\_BCLK, AUD\_DACDAT, AUD\_DACLCK, AUD\_ADCDAT, AUD\_ADCLK 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\_DACLCK 用于输出的左右声道同步。AUD\_ADCDAT, AUD\_ADCLK 是用于输入音频信号的，只有在录音时需要，我们这里不用。

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  $48000 \times 384 = 18.432\text{MHz}$ 。我们板上的标准时钟是 50MHz，如何能够产生 18.432MHz 的时钟呢？

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。同其他 IP 核一样，设置 Verilog 文件名。请等待一些时间，直至 IP 核设置弹出，如图 10-6 所示。我们选择 **Fractional-N PLL**，并使用 50MHz 时钟为输入，channel spacing 设为 1kHz，输出频率设置为 18.432M。这样就可以直接生成基准时钟 AUD\_XCK 了。锁相环锁定信号 locked 可以不用处理。

**注意：**如果 Windows 系统用户名是中文，有可能在生成 IP 核的时候出现找不到 temp 目录的问题。请将 Window 系统当前用户的 TEMP 或 TMP 目录设置改成不含中文的目录。参考修改方式：在非系统盘如 D 盘下新建文件夹 Temp，然后右击 **我的电脑**，选择 **属性 → 高级 → 环境变量**，在弹出的窗口里点击 **用户变量** 下的 TEMP 和 TMP 变量，将两个都改成 D:\Temp。如果不行，需要到注册表编辑器中修改 **%USERPROFILE%\Local Settings\Temp** 下对应选项内容。不同版本的 Windows 可能方式不同。

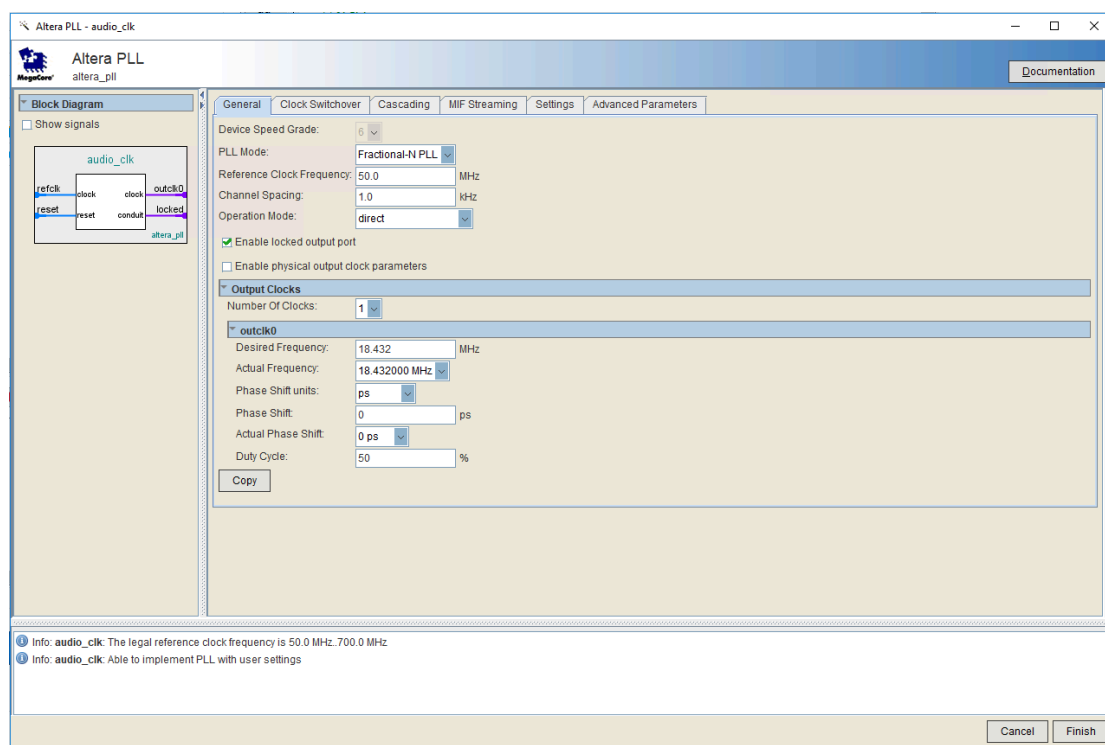


图 10-6: 音频时钟 PLL IP 核配置

在本实验中我们采用 I<sup>2</sup>S 接口的 **Left Justified**（左对齐）的音频信号的传输模式，如图 10-7 所示。在每一个样本点时间内，即 1/48 毫秒的时间内，我们需要传输 16bit 左声道数据和 16bit 右声道数据。因此，我们的 AUD\_BCLK



需要的频率为  $48000 \times 32 = 1.536\text{MHz}$ ，是基准频率  $18.432\text{MHz}$  的 12 分之一。所以 `AUD_BCLK` 可以用 `AUD_XCK` 计数分频获取。`AUD_DACLRC` 确定当前传输的是左声道还是右声道，左声道为高电平，右声道为低电平。其频率为  $48\text{kHz}$ ，正好为 `AUD_BCLK` 的 32 分之一。此处需要注意的是，`AUD_DACLRC` 和数据线 `AUD_DACDAT` 上的信号都是和 `AUD_BCLK` 的下降沿对齐的。

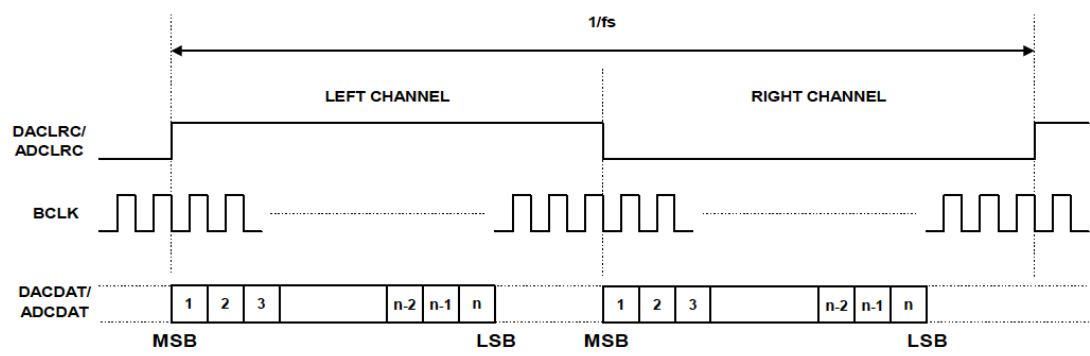


图 10-7: 左对齐的音频信号传输模式

在生成完时钟信号后，我们只需要按要求将每个样本点的 16bit 有符号整数数据按高位在前发送即可。对于我们的实验，可以将左右声道设置为一样的数据，实现单声道播放。

### 10.3 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。钢琴上的不同音高对应着不同的频率，如表 10-4 所示。我们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，建议可以实现 C5, D5, ..., B5, C6 这些音符。
  - 只需要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。
- ✎ 可选扩展要求
- 可调节音量。

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 <sup>#</sup>	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 <sup>#</sup>	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 <sup>#</sup>	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 <sup>#</sup>	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 <sup>#</sup>	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

- 支持多个键同时按下的和声。例如，同时按下 C5，E5，G5 时发出大三和弦，即对应三个音相加的结果，可以只支持同时发两个音。




本实验请自备耳机，在实验中请勿直接佩戴耳机调试，注意音量调节，避免伤害耳朵。



在试图通过修改 `audio_cmd` 中的内容来调节音量时请注意，`audio_cmd` 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。



在进行和声数值相加时请注意查表数据为带符号整数，直接相加会溢出，按比例缩小数值后再相加。

 **最低要求：** 本实验有一定难度，最低完成要求是能够放出不同音调的声音（可用拨动开关控制音高），仅满足最低要求无法获得全部分数。