中 国 矿 业 大 学

China University of Mining and Technology

2024-2025（2）《FPGA 数字系统设计》
音频采集处理与播放
课程分析设计报告

时　　间：2025 年 6 月 9 日

目录

1 设计背景

随着多媒体与智能语音系统的发展，音频处理在嵌入式系统中占据越来越重要的地位。传统的音频采集多依赖于专用音频处理芯片，但随着 FPGA 性能提升，其高度并行和可编程的特性使其成为音频处理的理想平台。

本设计基于 Spartan-6 FPGA 与 WM8731 音频编解码器，完成音频信号的采集模块设计，为后续音频处理（如滤波、播放、语音识别）提供基础。

2 系统组成

2.1 硬件平台

1. FPGA 芯片：Spartan-6 AX309（黑金开发板）
2. 音频模块：AN831（集成 WM8731 芯片）
3. FPGA 时钟晶振：50MHz
4. 控制接口：I2C（用于配置 WM8731 寄存器）
5. 音频接口：LINE IN / MIC IN / HEADPHONE OUT
6. 显示与调试：按键开关

2.2 软件工具

1. ISE 14.7（Verilog 代码开发与综合）
2. Modelsim/Chipscope（波形分析）

REGISTER ADDRESS	BIT	LABEL	DEFAULT	DESCRIPTION
0000111 Digital Audio Interface Format	1:0	FORMAT[1:0]	10	Audio Data Format Select 11 = DSP Mode, frame sync + 2 data packed words 10 = I^2S Format, MSB-First left-1 justified 01 = MSB-First, left justified 00 = MSB-First, right justified
	3:2	IWL[1:0]	10	Input Audio Data Bit Length Select 11 = 32 bits 10 = 24 bits 01 = 20 bits 00 = 16 bits
	4	LRP	0	DACLRC phase control (in left, right or I^2S modes) 1 = Right Channel DAC data when DACLRC high 0 = Right Channel DAC data when DACLRC low (opposite phasing in I^2S mode) or DSP mode A/B select (in DSP mode only) 1 = MSB is available on 2nd BCLK rising edge after DACLRC rising edge 0 = MSB is available on 1st BCLK rising edge after DACLRC rising edge
	5	LRSWAP	0	DAC Left Right Clock Swap 1 = Right Channel DAC Data Left 0 = Right Channel DAC Data Right
	6	MS	0	Master Slave Mode Control 1 = Enable Master Mode 0 = Enable Slave Mode
	7	BCLKINV	0	Bit Clock Invert 1 = Invert BCLK 0 = Don't invert BCLK

图 1 wm8731 7 号寄存器

3 总体功能设计

官方提供的案例中，wm8731 配置为主模式，即时钟由 wm8731 提供，寄存器 7 采用 8'h40，默认 16bit 有效字长，右对齐模式，采样频率为 48kHz。

因为本次设计中需要采用滤波器，所以需要使用 DSP 模式，经过考虑选用 DSPmodeB 为宜。根据芯片手册配置寄存器 7，配置为 8'h0f,即选择 DSPmodeB，字长根据官方代码选择 32bit 改动较少，所以这里有效字长选择 32bit，选择从模式，由 FPGA 来控制时钟信号便于语音信号的处理。

3.1 系统框架图

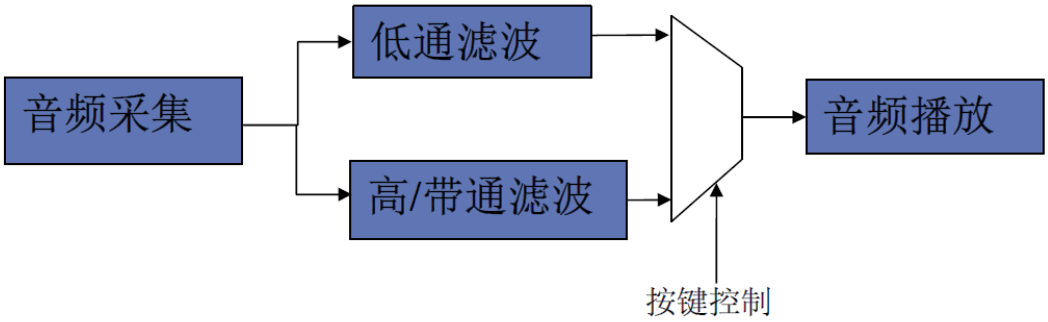


图 2 系统框架图

3.2 主要文件结构

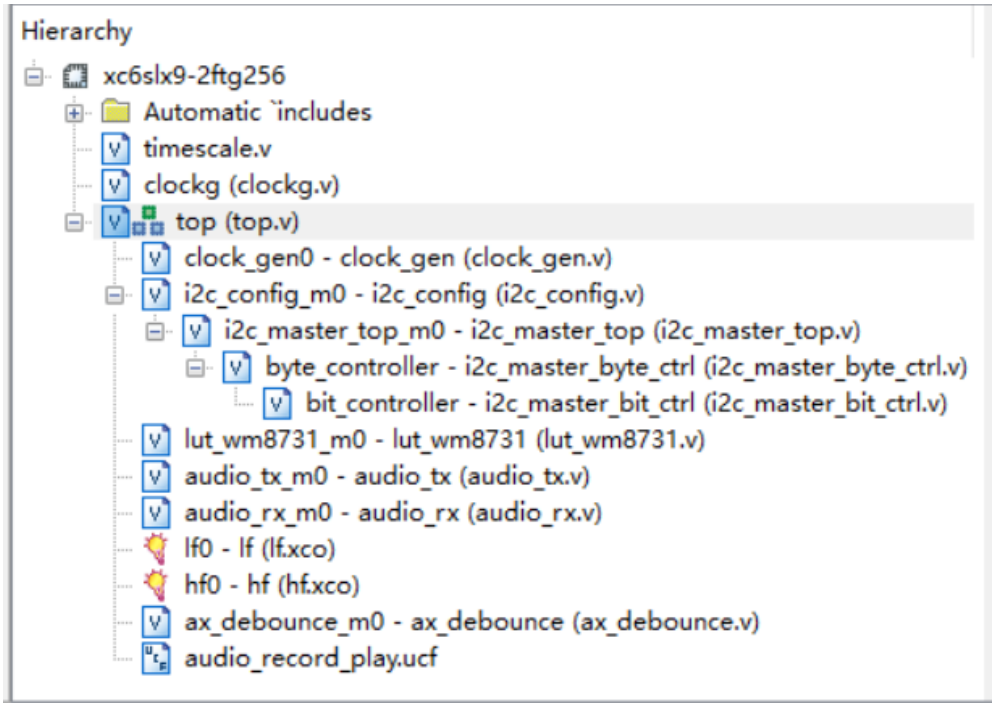


图 3 主要工程文件

首先根据设计需求，将整个系统划分为以下功能模块：

- **音频控制模块（I²C）**：配置 WM8731 芯片工作模式；
- **音频采集模块（audio_rx）**：接收 WM8731 输出的 ADC 数据；
- **音频发送模块（audio_tx）**：将处理后的数据通过 DAC 发送至音频输出；
- **滤波模块（lf、hf）**：对音频数据进行低通或高通处理；
- **时钟模块（clock_gen）**：生成满足 WM8731 的 BCLK 与 LRCK；
- **按键消抖模块（ax_debounce）**：处理按键输入，控制滤波模式切换；
- **顶层模块（top）**：实现各子模块的互联与整体控制逻辑。

4 模块功能实现

4.1 时序产生

通过下图右对齐和 DSPmodeB 的时序图分析可知，Fs=48kHz，所以 bclk=64*0.048=3.072MHz，LRC=48kHz，且需要注意 bclk 下降沿对应 LRC 上升沿。

本地时钟为 50MHz，所以需要通过分频将 50MHz 的时钟分频成 3.072MHz 的 bclk 信号，再通过 bclk 信号分频生成 LRC 信号，满足时序要求。

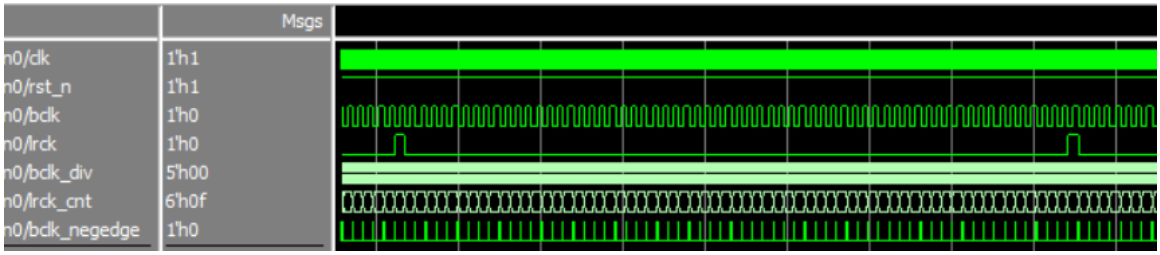


图 4 modelsim 时序信号仿真

4.2 通信接口设计

直接采用黑金官方提供的案例中的 i2c 模块直接使用即可。但是对于 wm8731 寄存器配置寄存器 7 需要配置成 8'h0f。

Top 顶层调用：//I2C master controller

```
i2c_config i2c_config_m0(
    .rst                    (~rst_n              ),
    .clk                    (clk_bufg            ),
    .clk_div_cnt             (16'd500             ),
    .i2c_addr_2byte          (1'b0                ),
    .lut_index              (lut_index           ),
    .lut_dev_addr            (lut_data[31:24]       ),
    .lut_reg_addr            (lut_data[23:8]        ),
    .lut_reg_data            (lut_data[7:0]         ),
    .error                  (                    ),
    .done                   (                    ),
    .i2c_scl                 (wm8731_scl            ),
    .i2c_sda                 (wm8731_sda            )
);
```

4.3 音频接收发送模块（audio_rx.v 和 audio_tx.v）

DSP Mode B 是一种帧同步方式的串行音频格式，使用 FSYNC（帧同步信号）作为起始标志，左声道紧接其后，右声道紧接左声道，不使用 LRC（左右声道标志位），数据是右对齐的。

图 5 DSPmodeB 时序图

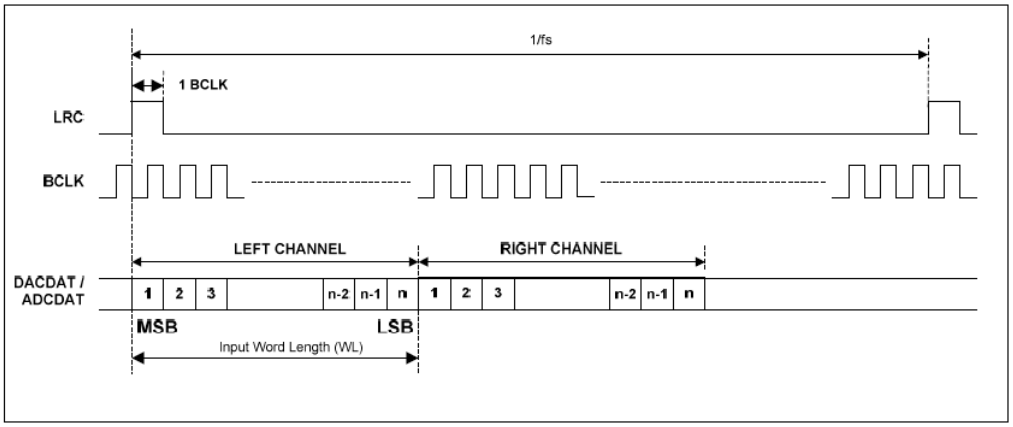


Figure 30 DSP/PCM Mode Audio Interface (mode B, LRP=0)

图 6 DSPmodeB 时序

因此，可以利用右对齐的代码稍加修改。只需要一个 64 位的左移寄存器，因为在 DSPmodeB 中，左右声道数据是紧密相连的。

注意：这里直接采用官方的右对齐时序的采集发送代码，会导致只有右声道数据被采集，下载到 FPGA 上会产生许多的杂音。（通过 chipscope 观察到）

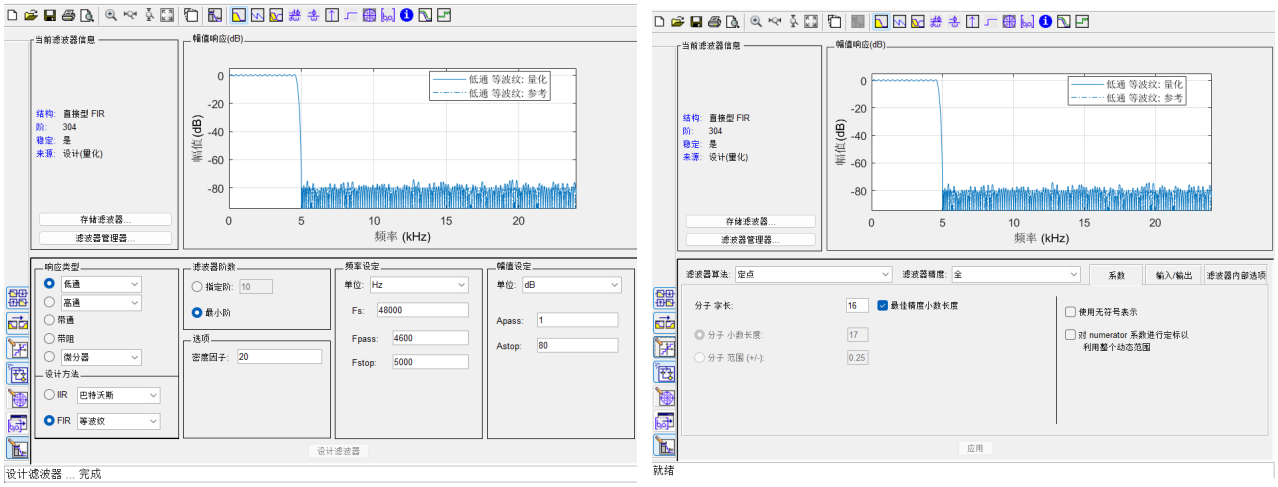
4.4 滤波器设计

4.4.1 coe 文件的生成

采用 matlab 的 filterDesigner 设计低通滤波器和高通滤波器的 coe 文件。

均选择分子字长 16 位。

生成之后，点击目标生成 xlinx 系数。



(a)低通设计（高通类似）　　(b)选择定点

图 7 滤波器参数选择

4.4.2 生成 IP 核

在项目中新建 IP 文件，在 DSP 中选择 FIRCompiler，进入 IP 核配置页面，Coefficients File 导入系数文件，即可进行配置，设置采样频率，时钟，两通道，输入数据 32 位等。

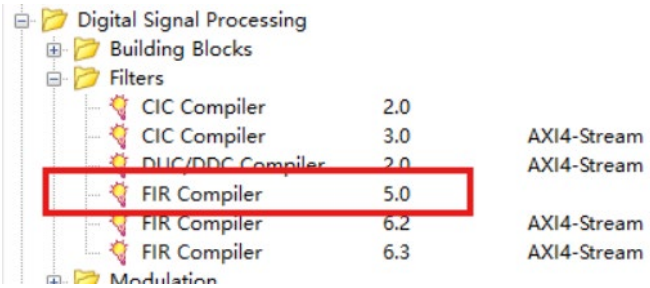


图 8 选择 FIR Compiler

Summary

Component Name :	lf
Filter Type :	Single Rate
Number of Channels :	1
Clock Frequency :	50
Input Sampling Frequency :	0.048
Sample Period :	N/A
Input Data Width :	32
Input Data Fractional Bits :	0
Number of Coefficients :	305
Calculated Coefficients :	305
Number of Coefficient Sets :	1
Reloadable Coefficients :	No
Coefficient Structure :	Symmetric
Coefficient Width :	16
Coefficient Fractional Bits :	0
Quantization Mode :	Integer_Coefficients
Gain due to Maximizing	
Dynamic Range of Coefficients :	N/A
Rounding Mode :	Full Precision
Output Width :	51 (full precision = 51 bits)
Output Fractional Bits :	0
Cycle Latency :	164
Filter Architecture :	Systolic Multiply Accumulate
Control Options :	None

(a)低通参数

Summary

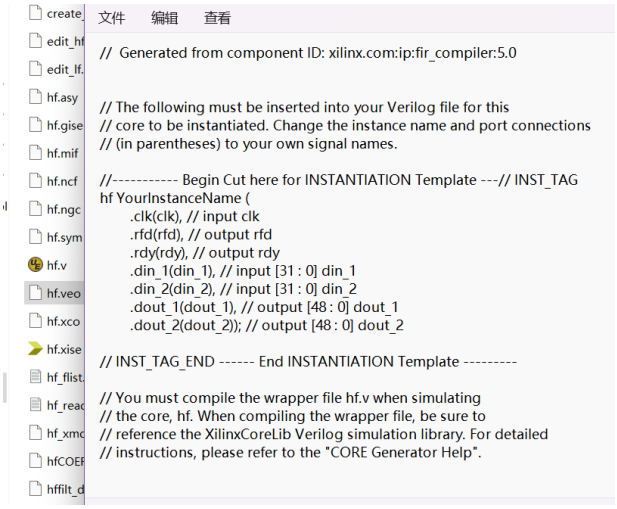
Component Name :	hf
Filter Type :	Single Rate
Number of Channels :	1
Clock Frequency :	50
Input Sampling Frequency :	0.048
Sample Period :	N/A
Input Data Width :	32
Input Data Fractional Bits :	0
Number of Coefficients :	331
Calculated Coefficients :	331
Number of Coefficient Sets :	1
Reloadable Coefficients :	No
Coefficient Structure :	Symmetric
Coefficient Width :	16
Coefficient Fractional Bits :	0
Quantization Mode :	Integer_Coefficients
Gain due to Maximizing	
Dynamic Range of Coefficients :	N/A
Rounding Mode :	Full Precision
Output Width :	49 (full precision = 49 bits)
Output Fractional Bits :	0
Cycle Latency :	177
Filter Architecture :	Systolic Multiply Accumulate
Control Options :	None

(b)高通参数

图 9 IP 核全体参数

4.4.3 IP 核的调用

找到工程文件下 IP 核生成的位置，选择相应 IP 核的 veo 文件，查看可以看到相应的调用格式。



(a)veo 文件内容

```
27  );
28
29  //---------- Begin Cut here for INSTANTIATION Templat
30  lf lf0 (
31      .clk(clk_bufg), // input clk
32      .rfd(), // output rfd
33      .rdy(), // output rdy
34      .din_1(rx_left_data), // input [31 : 0] din_1
35      .din_2(rx_right_data), // input [31 : 0] din_2
36      .dout_1(l_left_data), // output [50 : 0] dout_1
37      .dout_2(l_right_data)); // output [50 : 0] dout_2
38
39  hf hf0 (
40      .clk(clk_bufg), // input clk
41      .rfd(), // output rfd
42      .rdy(), // output rdy
43      .din_1(rx_left_data), // input [31 : 0] din_1
44      .din_2(rx_right_data), // input [31 : 0] din_2
45      .dout_1(h_left_data), // output [50 : 0] dout_1
46      .dout_2(h_right_data)); // output [50 : 0] dout_2
47
48
49
50
```

(b)工程上的调用

图 10 IP 核的调用

4.4.4 滤波结果有效位截取

低通输出 51 位，高通输出 49 位，均选择最后 32 位，有正确有效输出。

5 验证

5.1 仿真测试验证

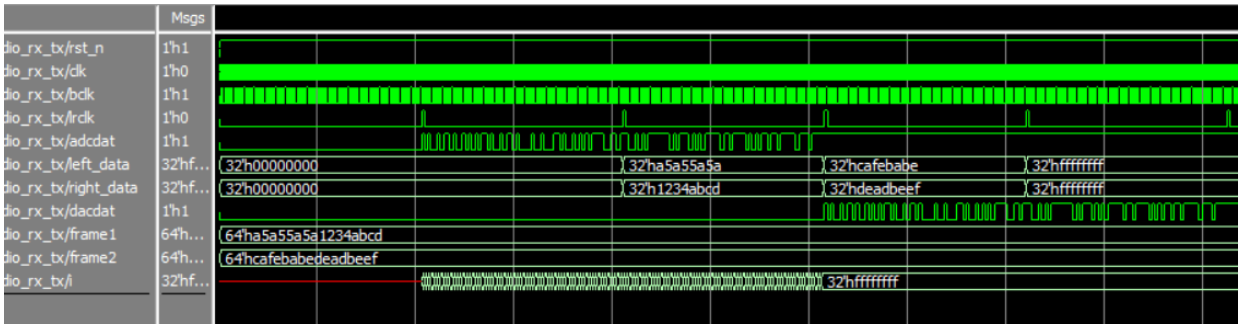


图 11 modelsim 仿真验证收发模块

由上图看出收发模块时序对齐满足 DSPmodeB 的时序要求。

5.2 Chipscope 数据抓取

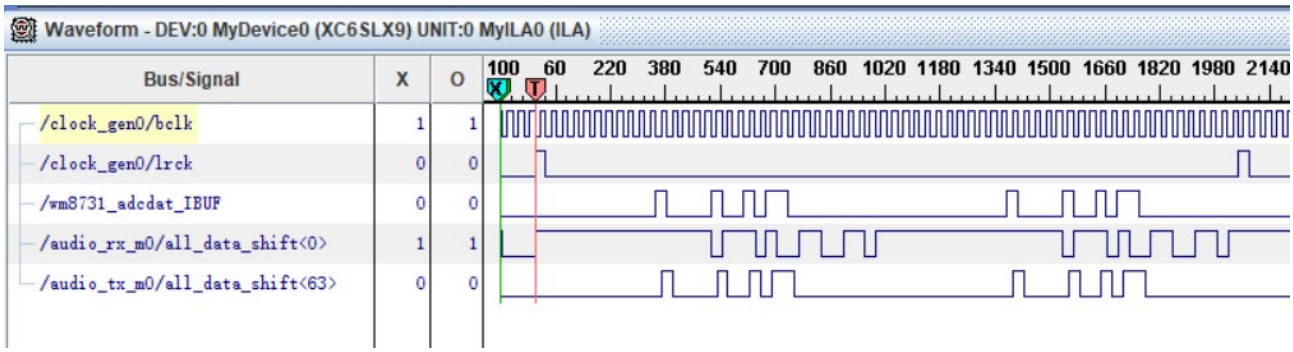


图 12 chipscope 数据抓取

5.3 板上测试

将程序下载到 FPGA 板子上，并插入两个耳机，一个用来录音，一个用来监听，开始进行测试。

刚打开时，没有声音，但能听到通道开启，此时默认输出低通滤波数据，播放音乐，与之前听到的差别不大；按下按键，切换为输出高通数据，能明显听到的音乐得到削弱。

6 设计资源分析

6.1 总体资源使用情况

Slice 寄存器使用：1,199 / 11,440，约占 10%。其中大部分为 D 触发器（1,198 个），仅有 1 个锁存器，说明设计主要以同步逻辑为主，结构清晰。Slice LUT 使用：880 / 5,720，约占 15%。其中用于组合逻辑的为 646 个，用作 Shift Register 的为 132 个，未使用片上 RAM。Shift

Register 使用较多，全部以 LUT 实现而非 BRAM，仍有较大资源冗余空间。Route-thru 使用：102 个，主要服务于局部寄存器和进位链连接，表明逻辑布局紧凑但未过度拥挤。

总体利用率低。

6.2 时序路径分析

项目	内容
路径类型	Setup Path（建立时间路径）
起点元件	U_DIRTY_LDC（锁存器）
终点元件	U_TDO（D 触发器）
时钟控制信号	从 icon_control0<13> 的**下降沿** 到 icon_control0<0> 的**上升沿**
数据路径延迟	4.117 ns（共 **4 层**组合逻辑）
总路径延迟	4.152 ns（包含时钟不确定性）
时钟偏移/不确定性	0 ns/0.035 ns（由抖动和相位误差引入）
系统抖动（TSJ）	0.070 ns
输入抖动（TIJ）	0.000 ns
离散抖动（DJ）	0.000 ns
相位误差（PE）	0.000 ns

表格 1 时序分析报告（Timing Report）关键路径分析

该路径为本设计的关键路径，数据路径延迟为 **4.117ns**，总延迟为 **4.152ns**，尚处于合理范围内。由于起点为锁存器，终点为 D 触发器，设计注意锁存器可能引入的不确定性，在后续设计中尽量统一时钟边沿控制，减少 latch 使用。

此外，由于该路径为跨时钟域传输（下降沿到上升沿），应进一步检查是否已进行时钟域同步，防止出现亚稳态问题。

Timing Summary：

Timing errors: 0 Score: 0 (Setup/Max: 0, Hold: 0)

Constraints cover 11754 paths, 0 nets, and 5853 connections ，由此可知该设计共计覆盖 **11754 条路径，5853 个连接。**

6.3 时钟资源使用分析

Clock Net	Routed	Resource	Locked	Fanout	Net Skew(ns)	Max Delay(ns)
clk_bufg_BUFG	ROUTED	BUFGMUX_X3Y13	No	322	0.117000	1.509000
icon_control0<0>	ROUTED	BUFGMUX_X2Y3	No	81	0.118000	1.509000
U_icon_pro/U0/iUPDATE...	ROUTED	Local		1	0.000000	0.978000
icon_control0<13>	ROUTED	Local		4	0.000000	1.245000

图 13 Clock report

分析：

1．全体时钟均已成功路由

所有列出的时钟网（包括主时钟、调试相关控制时钟）均已完成布线（Routed），说明没有时钟丢失或布线失败的问题，设计在全局时钟配置上是完整的。

2．主时钟资源使用情况良好

- clk_bufg_BUFG 使用的是 BUFGMUX 全局缓冲器，扇出高达 **322**，为全系统主时钟；

- 该主时钟的 **Skew（偏斜）为 0.117ns**，**Max Delay 为 1.509ns**，属于可接受范围内，说明布线和分布均衡；
- **icon_control0<0>** 也是通过全局 **BUFGMUX**，说明调试模块也采用了全局时钟网络，扇出为 **81**，同样保持较低的偏斜。

3．局部时钟

- **U_icon_pro/U0/iUPDATE** 和 **icon_control0<13>** 这两个时钟未使用全局缓冲器，而是采用本地布线（**Local**），说明这部分逻辑主要为调试或控制路径，布线资源优化；
- 偏斜为 **0ns**，延迟分别为 **0.978ns** 和 **1.245ns**，均无异常。

7 总结与体会

7.1 设计过程总结

在功能设计过程中，特别重视以下几点：

1. **音频采集与发送模块**：严格按照 **WM8731** 数据手册中 **DSP Mode B** 格式设计 **BCLK** 和 **LRCK** 的时序，同时保证 **audio_tx** 与 **audio_rx** 模块对数据进行正确的时钟对齐。
2. **滤波模块**：分别设计了低通（**lf**）和高通（**hf**）两个模块，对 **WM8731** 的 **ADC** 左右声道输出进行并行处理。采用了高位宽输出（**51bit** 和 **49bit**）确保滤波精度，在顶层模块中适当截位恢复到 **32bit**。
3. **模式切换逻辑**：使用按键高电平控制低通滤波输出，低电平切换到高通滤波输出，确保用户交互逻辑简单直观。

7.2 收获与反思

本次基于 **FPGA** 的音频滤波系统设计，让我对数字信号处理、硬件接口控制以及模块化系统设计有了更加系统和深入的理解。在实际开发与调试过程中，我也积累了许多宝贵的经验，尤其对“查阅芯片手册”和“工具调试能力”的重要性有了深刻体会。查阅芯片手册保证了设计的正确性。在模块设计完成后，我充分使用了 **ModelSim** 进行仿真，验证了音频数据采集发送等模块的功能。通过波形观察，我能精准定位时序上的细节错误，比如 **audio_tx** 模块数据提前或延后一个 **BCLK** 等。

这些问题若不通过仿真，仅靠板上调试几乎无法准确定位。**ModelSim** 能让我们在代码级别就找到逻辑漏洞，极大地节省了开发调试时间。

在将系统部署到 **FPGA** 实际平台后，我使用 **ChipScope** 监控关键信号的时序和变化，发现 **audio_rx** 输出数据延迟、滤波输出未刷新等问题。通过在 **ChipScope** 中插入 **BCLK**、**LRCK** 滤波输出数据等信号，我获得了“可视化”的调试手段。

这次设计让我学会了使用调试工具，查看芯片手册，如何正确调用 **IP** 核，收获良多。

代码

```
`timescale 1ps/1ps
module top
(
    input   clk, //clock input
    input  rst_n,   //reset input
    input  key,  //record play button
    output  wm8731_bclk, //audio bit clock
    output wm8731_daclrc,
    output wm8731_dacdat,
    output wm8731_adclrc,
    input  wm8731_adcdat,
    inout  wm8731_scl,  //I2C clock
```

```
    inout   wm8731_sda  //I2C data
);
wire[9:0]                   lut_index;
wire[31:0]                  lut_data;
wire                       clk_bufg;
wire lrck;
//generate SDRAM controller clock
IBUFG IBUFGP_INST
(
  .O(clk_bufg),
  .I(clk)
);
assign wm8731_daclrc=lrck;
assign wm8731_adclrc=lrck;
wire[31:0] rx_left_data;
wire[31:0] rx_right_data;

reg[31:0]  tx_left_data;
reg[31:0]  tx_right_data;

wire[50:0] l_left_data;
wire[50:0] l_right_data;

wire[48:0] h_left_data;
wire[48:0] h_right_data;

clock_gen clock_gen0 (
    .clk(clk_bufg), // 主时钟，比如 50MHz
    .rst_n(rst_n),  // 异步复位
    .bclk(wm8731_bclk), // 3.072 MHz BCLK
    .lrck(lrck)    // 48 kHz LRCK 脉冲
);
//I2C master controller
i2c_config i2c_config_m0(
    .rst  (~rst_n),
    .clk  (clk_bufg  ),
    .clk_div_cnt  (16'd500  ),
    .i2c_addr_2byte (1'b0  ),
    .lut_index  (lut_index),
    .lut_dev_addr  (lut_data[31:24] ),
    .lut_reg_addr   (lut_data[23:8] ),
    .lut_reg_data (lut_data[7:0]),
    .error    (   ),
    .done   (  ),
    .i2c_scl  (wm8731_scl  ),
    .i2c_sda (wm8731_sda )
);
//configure look-up table
lut_wm8731 lut_wm8731_m0(
    .lut_index   (lut_index   ),
    .lut_data (lut_data   )
);

wire button_out;
audio_tx audio_tx_m0
(
    .rst        (~rst_n   ),
    .clk        (clk_bufg),
    .sck_bclk    (wm8731_bclk),
    .ws_lrc      (wm8731_adclrc ),
    .sdata       (wm8731_dacdat ),
    .left_data    (tx_left_data ),
   .right_data   (tx_right_data),
    .read_data_en ()
);
//audio receiver
audio_rx audio_rx_m0
(
    .rst        (~rst_n),
    .clk        (clk_bufg  ),
    .sck_bclk    (wm8731_bclk ),
    .ws_lrc      (wm8731_adclrc ),
    .sdata       (wm8731_adcdat ),
    .left_data    (rx_left_data  ),
    .right_data   (rx_right_data ),
    .data_valid   (   )
);

lf lf0 (
    .clk(clk_bufg), // input clk
    .rfd(), // output rfd
    .rdy(), // output rdy
    .din_1(rx_left_data),
    .din_2(rx_right_data),
    .dout_1(l_left_data),
    .dout_2(l_right_data));
hf hf0 (
    .clk(clk_bufg), // input clk
```

```
    .rfd(), // output rfd
    .rdy(), // output rdy
    .din_1(rx_left_data),
    .din_2(rx_right_data),
    .dout_1(h_left_data),
    .dout_2(h_right_data));
always@(posedge clk_bufg or negedge rst_n)
begin

if(~rst_n)
begin
tx_left_data<=32'd0;
tx_right_data<=32'd0;
end
else if(button_out)
begin
tx_left_data<=l_left_data[50:19];
tx_right_data<=l_right_data[50:19];
end
else if(~button_out)
begin
tx_left_data<=h_left_data[48:17];
tx_right_data<=h_right_data[48:17];
end
end
ax_debounce ax_debounce_m0
(
    .clk           (clk_bufg         ),
    .rst           (~rst_n       ),
    .button_in      (key         ),
    .button_posedge  (),
    .button_negedge  (),
    .button_out     (button_out  )
);


Endmodule

module clock_gen (
    input wire clk,
    input wire rst_n,
    output reg bclk,
    output reg lrck
);

    reg [4:0] bclk_div;
    parameter BCLK_DIV_MAX = 16 - 1;

    // 6-bit LRCK 计数器，每 64 个 BCLK 产生一个
脉冲
    reg [5:0] lrck_cnt;

    // ====== BCLK 生成 ================
    always @(posedge clk or negedge rst_n)
begin
        if (!rst_n) begin
            bclk_div <= 0;
            bclk <= 0;
        end else begin
            if  (bclk_div  ==  BCLK_DIV_MAX)
begin
                bclk_div <= 0;
                bclk <= ~bclk;
            end else begin
                bclk_div <= bclk_div + 1;
            end
        end
    end

    // ========= LRCK 生成 ================
    wire  bclk_negedge  =  (bclk_div  ==
BCLK_DIV_MAX) && (bclk == 1);

    always @(posedge clk or negedge rst_n)
begin
        if (!rst_n) begin
            lrck_cnt <= 0;
            lrck <= 0;
        end else begin
            if (bclk_negedge) begin
                if (lrck_cnt == 63) begin
                    lrck_cnt <= 0;
                    lrck <= 1;
                end else begin
                    lrck_cnt <= lrck_cnt + 1;
                    lrck <= 0;
                end
            end
```

```
       end
    end

endmodule

`timescale 1ns/1ps
module audio_tx
(
    input     rst,
    input     clk,
    input    sck_bclk,
    input    ws_lrc,
    output reg   sdata,
    input[31:0]   left_data,
    input[31:0]  right_data,
    output reg   read_data_en
);
reg            sck_bclk_d0;
reg            sck_bclk_d1;
reg          ws_lrc_d0;
reg          ws_lrc_d1;

reg[63:0]       all_data_shift;

always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
    begin
        sck_bclk_d0 <= 1'b0;
        sck_bclk_d1 <= 1'b0;
        ws_lrc_d0 <= 1'b0;
        ws_lrc_d1 <= 1'b0;
    end
    else
    begin
        //delay
        sck_bclk_d0 <= sck_bclk;
        sck_bclk_d1 <= sck_bclk_d0;
        ws_lrc_d0 <= ws_lrc;
        ws_lrc_d1 <= ws_lrc_d0;
    end
end
always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        all_data_shift <= 64'd0;
    else if(ws_lrc_d1 == 1'b0 && ws_lrc_d0
== 1'b1)
        all_data_shift<=
{left_data,right_data};
    else   if(sck_bclk_d1   ==   1'b1   &&
sck_bclk_d0 == 1'b0)//ws_lrc = 0 ,sck_bclk
negedge
        all_data_shift<=
{all_data_shift[62:0],1'b0};
end

always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        sdata <= 1'd0;
    else
        sdata <= all_data_shift[63];
end
always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        read_data_en <= 1'b0;
    else if(ws_lrc_d1 == 1'b0 && ws_lrc_d0
== 1'b1)//ws_lrc posedge  read the next audio
data
        read_data_en <= 1'b1;
    else
        read_data_en <= 1'b0;
end
endmodule

`timescale 1ns/1ps
module audio_rx
(
    input                       rst,
    input                       clk,
    input                       sck_bclk,
    input                       ws_lrc,
    input                       sdata,
   output reg[31:0]             left_data,
   output reg[31:0]            right_data,
    output reg                  data_valid
```

```
);
reg             sck_bclk_d0;
reg               sck_bclk_d1;
reg              ws_lrc_d0;
reg             ws_lrc_d1;

reg[63:0]
all_data_shift;//right  channel  audio  data
shift register

always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
    begin
        sck_bclk_d0 <= 1'b0;
        sck_bclk_d1 <= 1'b0;
        ws_lrc_d0 <= 1'b0;
        ws_lrc_d1 <= 1'b0;
    end
    else
    begin
        sck_bclk_d0 <= sck_bclk;
        sck_bclk_d1 <= sck_bclk_d0;
        ws_lrc_d0 <= ws_lrc;
        ws_lrc_d1 <= ws_lrc_d0;
    end
end


always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        all_data_shift <= 32'd0;
    else if(ws_lrc_d1 == 1'b0 && ws_lrc_d0
== 1'b1)//ws_lrc posedge
        all_data_shift <= 32'd0;
    else   if(sck_bclk_d1   ==   1'b0   &&
sck_bclk_d0 == 1'b1)//ws_lrc = 0 ,sck_bclk
posedge
        all_data_shift                    <=
{all_data_shift[62:0],sdata};
end

always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
    begin
        left_data <= 32'd0;
        right_data <= 32'd0;
    end
    else if(ws_lrc_d1 == 1'b0 && ws_lrc_d0
== 1'b1)//ws_lrc posedge
    begin
        left_data <= all_data_shift[63:32];
        right_data <= all_data_shift[31:0];
    end
end

always@(posedge clk or posedge rst)
begin
    if(rst == 1'b1)
        data_valid <= 1'b0;
    else if(ws_lrc_d1 == 1'b0 && ws_lrc_d0
== 1'b1)//ws_lrc posedge
        data_valid <= 1'b1;
    else
        data_valid <= 1'b0;
end

endmodule
```