AC920 开发板

基于 TCP 传输系统的 ACFL3432 数据采集与显示说明

武汉芯路恒科技有限公司

教学产品项目组 编写

修订说明：

V1.0.0	2025-11-17	第一版本发行

版权所有 © 2025 武汉芯路恒科技有限公司

免责声明

本文档并未授予任何知识产权的许可，并未以明示或暗示，或以禁止发言或其它方式授予任何知识产权许可。除芯路恒在其产品的销售条款和条件中声明的责任之外，武汉芯路恒概不承担任何法律或非法律责任。武汉芯路恒对芯路恒产品的销售和／或使用不作任何明示或暗示的担保，包括对产品的特定用途适用性、适销性或对任何专利权、版权或其它知识产权的侵权责任等，均不作担保。芯路恒对文档中包含的文字、图片及其它内容的准确性和完整性不承担任何法律或非法律责任，芯路恒保留修改文档中任何内容的权利，恕不另行通知。芯路恒不承诺对这些文档进行适时的更新。

联系我们

厂家名称：武汉芯路恒科技有限公司

联系地址：武汉东湖新技术开发区大学园路长城园路 8 号光谷精工科技园 A 栋 301-5 室

联系电话： 027-59265620

技术网站：fpga.cn

微信公众号（提供在线客服）：小梅哥电子

在线商城（淘宝）：https://xiaomeige.taobao.com

在线商城（天猫）：https://xiaomeigesm.tmall.com/

目 录

第1章 基于 TCP 传输系统的 ACFL3432 数据采集与显示说明

工程源码	|-- AC920 开发板 |-- AC920_CM3432_DualChannel_TCP
相关视频课程	
参考官方文档	

章节导读

使用 AC920 搭载 ACFL3432 模块实现 250Msps 高速数据采集。将 ACFL3432 模块采集到的数据通过 DMA 搬运到 DDR4 中存储，再经由 PS 端千兆以太网使用 TCP 协议将数据发送到上位机中进行波形绘制。

1.1 ACFL3432 模块简介

CM3432 芯片是北京士模微电子有限责任公司研发的 14 位双通道 250M 采样速率高速 ADC 芯片，ACFL3432 模块基于 CM3432 芯片实现高速采集。配合前端模拟信号调理电路，实现±5V 电压范围内信号的高速采样。该模块一共使用 2 路完全相同的信号调理电路，构成了双通道高速 AD 采样电路。两通道 AD 电路完全独立，结构和元器件参数相同，确保了两个通道有较高的一致性，ACFL3432 模块图如下图所示。

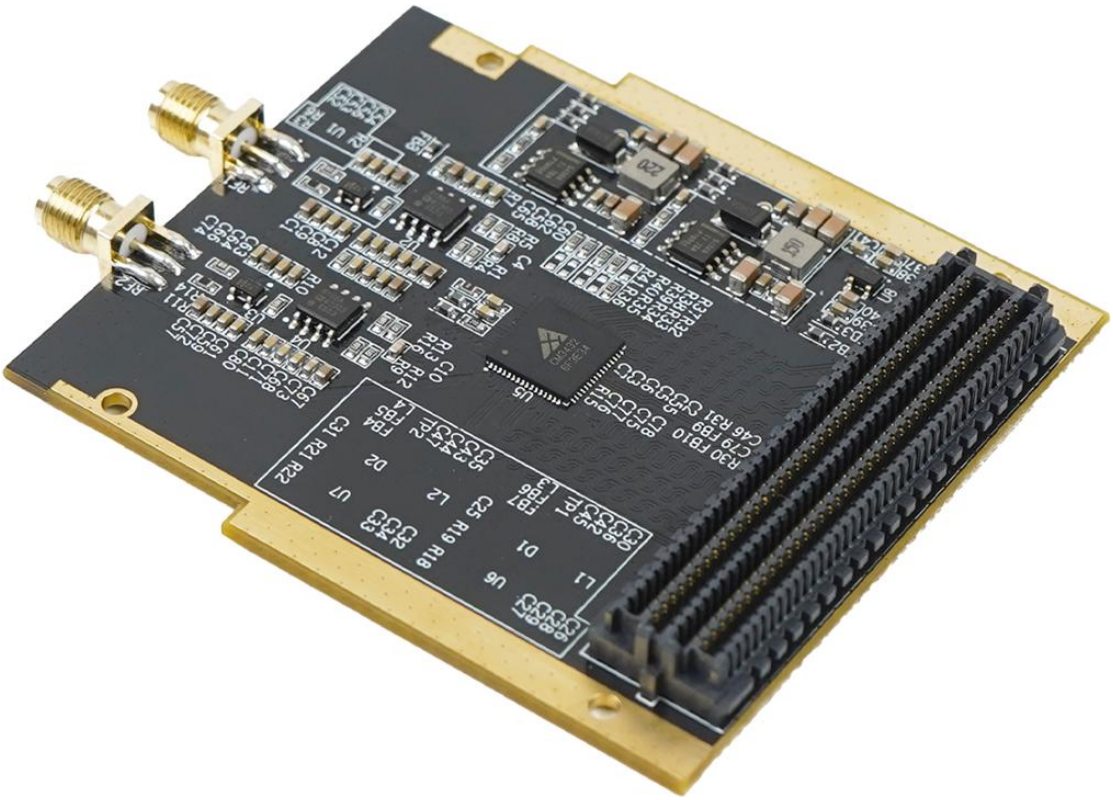

图 1-1 ACFL3432 模块实物图

ACFL3432 模块与 FPGA 的连接采用 FMC_LPC 接口，该模块提供灵活的接口配置：支持 1.8 V CMOS 或 LVDS 并行输出，该模块接口图如下图所示。

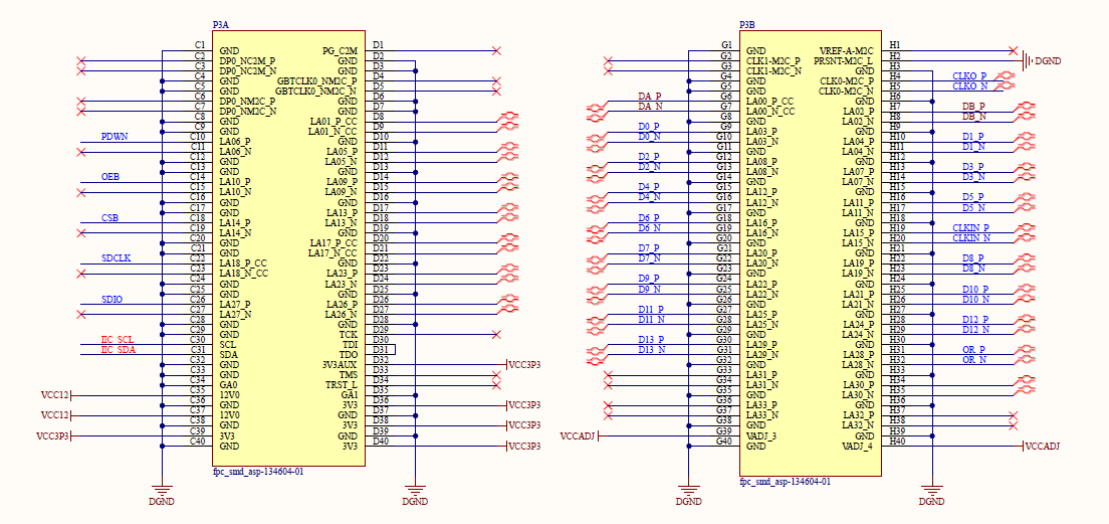

图 1-2 ACFL3432 模块 FMC_LPC 接口原理图

使用该模块时，我们首先需要通过 SPI 协议来配置该模块的寄存器，比如配置模拟信号电压，时钟分频倍频等，并且还需要提供一对差分时钟信号，根据芯片型号，我们这里提供 250Mhz 的差分时钟信号。

寄存器配置完成后，ADC 会根据我们提供的时钟以及时钟分频倍频寄存器参数输出一对时钟差分信号， 并在每个时钟周期输出一个 14 位的采样结果。

当 CM3432 芯片模拟输入端接-5V 至+5V 之间变化的正弦波电压信号时，其转换后的数据也是成正弦波波形变化，转换波形如下图所示，从图中可以看出 CM3432 芯片采集到的数据是无符号数据。

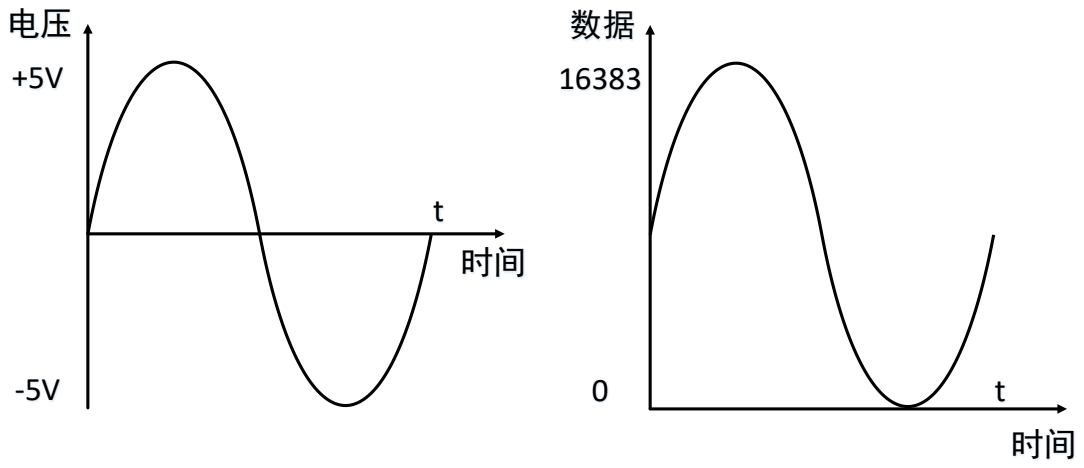

图 1-3 CM3432 正弦波模拟电压值（左）、数据（右）

1.2 整体系统设计

本章实验，通过小梅哥团队提供的 QT 上位机下发采集相关指令，比如采集通道、采集数量、采样频率以及配置 ACFL3432 模块寄存器指令；

将 ACFL3432 模块采集到的数据通过 DMA 搬运到 DDR4 中存储，再经由 PS 端千兆以太网使用 TCP 协议将数据发送到上位机中进行波形绘制。

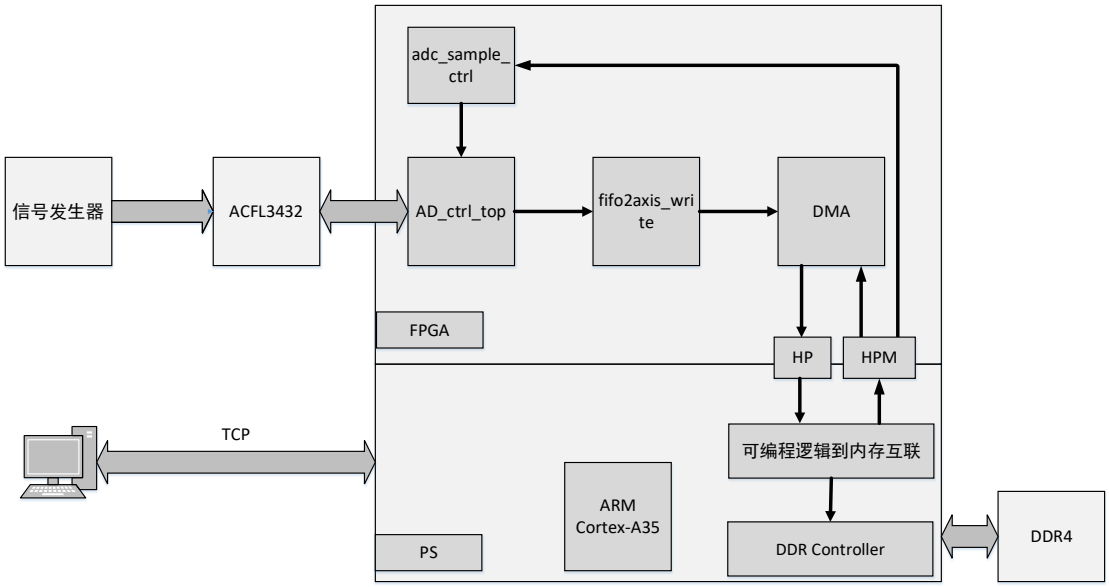

图 1-4　ACFL3432 采集、QT 上位机显示示意图

1.3　代码设计

设计整体会由 PL 与 PS 两部分构成：

1. PL 端负责搭建硬件逻辑系统，实现对 ACFL3432 模块的寄存器配置、采样参数设置以及搭建 DMA 通路。
2. PS 端负责通过 C 代码，解包用户下发的 TCP 指令交由 PL 端去驱动 ACFL3432 模块；在 AXI DMA 中断触发时，驱动 AXI DMA 完成数据搬运；采样结束后，从 DDR 中读出数据，通过 TCP 协议传输给 PC 端。

1.3.1 PL 端逻辑系统设计

PL 端硬件逻辑系统通过 BD 部分和逻辑部分组合搭建，完整结构如下图，图片如果不清晰，用户可直接参看例程。

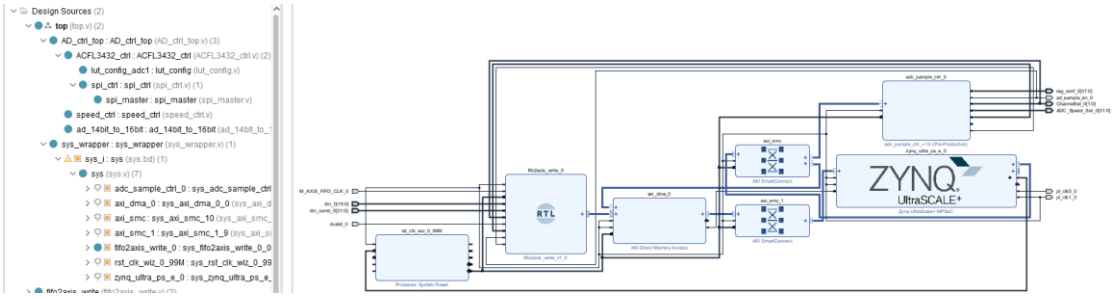

图 1-5　硬件逻辑系统

1.3.1.1AD 控制顶层模块

该模块负责对 ACFL3432 的寄存器进行配置；同时将 ACFL3432 模块采集输入的差分信号拆分，还原为 14 位原始数据；并对其原始数据进行位宽扩展为 16 位数据输出；并根据上位机传输来的频率控制指令进行采集速度控制。

1. ACFL3432_ctrl 模块负责通过 spi 接口对 CM3432 的寄存器进行配置；同时将 ACFL3432 模块采集输入的数据差分信号和时钟差分信号进行还原处理，得到 14 位原始数据和时钟；

2. speed_ctrl 模块负责根据上位机下发的指令，控制采样速率。
3. ad_14bit_to_16bit 模块负责将 ADC 采集数据进行有符号数位宽扩展。

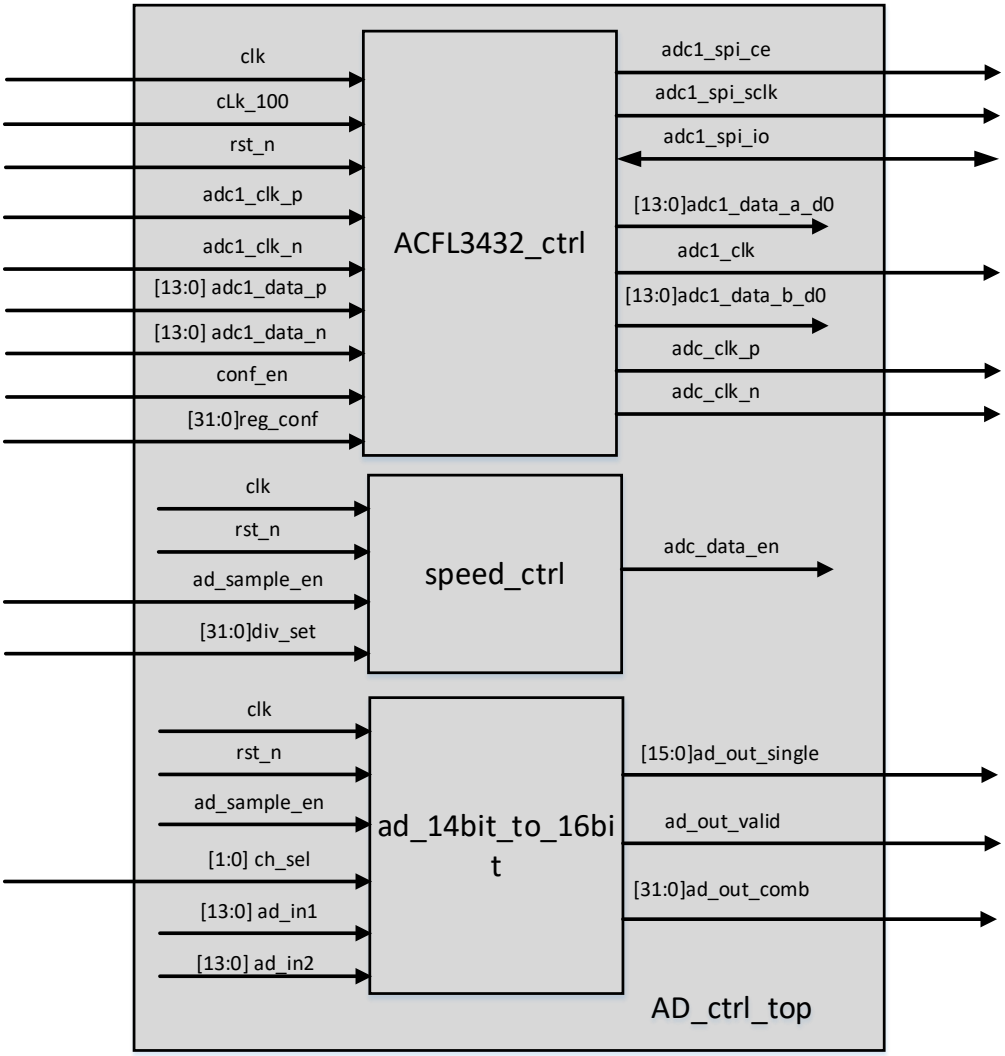

图 1-6 AD_ctrl_top 模块示意图

1.3.1.1.1 配置 CM3432 寄存器顶层

该模块包含 lut_config 模块和 spi_ctrl 模块；

1. 其中 lut_config 模块负责对 CM3432 的寄存器进行配置，将一些重要的寄存器地址和参数进行提前参数配置，程序开始运行就会读取该模块中的寄存器参数，去配置 CM3432 芯片。
2. spi_ctrl 模块负责通过 spi 协议，将 lut_config 模块中的寄存器地址和对应的寄存器参数写入到 CM3432 芯片的对应寄存器中，完成配置 CM3432 芯片。

程序中使用 OBUFDS 原语完成单端时钟信号的差分信号转换；IBUFDS 原语完成输入的差分时钟信号转单端信号；

```
OBUFDS  OBUFDS_inst (
  .O(adc_clk_p),     // Diff_p output (connect directly to top-level port)
  .OB(adc_clk_n),   // Diff_n output (connect directly to top-level port)
  .I(clk_100)      // Buffer input
);
```

```
IBUFDS IBUFDS_adc1_clk (
    .O(adc1_clk),  // Buffer output
    .I(adc1_clk_p),  // Diff_p buffer input (connect directly to top-level port)
    .IB(adc1_clk_n) // Diff_n buffer input (connect directly to top-level port)
);
```

使用 IBUFDS 原语配合 for 循环，从输入差分数据信号中还原出 14 位原始数据。

```
genvar i;
generate
    for (i = 0; i < 14; i = i + 1) begin:IBUFDS_DATAS
        IBUFDS  IBUFDS_adc1_data (
            .O(adc1_data[i]),  // Buffer output
            .I(adc1_data_p[i]),  // Diff_p buffer input
            .IB(adc1_data_n[i]) // Diff_n buffer input
        );

        IDDRE1 #(
            .DDR_CLK_EDGE("OPPOSITE_EDGE"), // IDDRE1 mode
            .IS_CB_INVERTED(1'b1),          // 启用 CB 端口内部反相
            .IS_C_INVERTED(1'b0)            // C 端口不反相
        )
        IDDR_adc1_data (
            .Q1(adc1_data_a[i]), // 1-bit output: Registered parallel output 1
            .Q2(adc1_data_b[i]), // 1-bit output: Registered parallel output 2
            .C(adc1_clk),        // 1-bit input: High-speed clock
            .CB(adc1_clk),       // 1-bit input: 连接到与 C 相同的时钟信号
            .D(adc1_data[i]),    // 1-bit input: Serial Data Input
            .R(1'b0)             // 1-bit input: Active-High Async Reset
        );
    end
endgenerate
```

ACFL3432_ctrl 模块整体示意图如下：

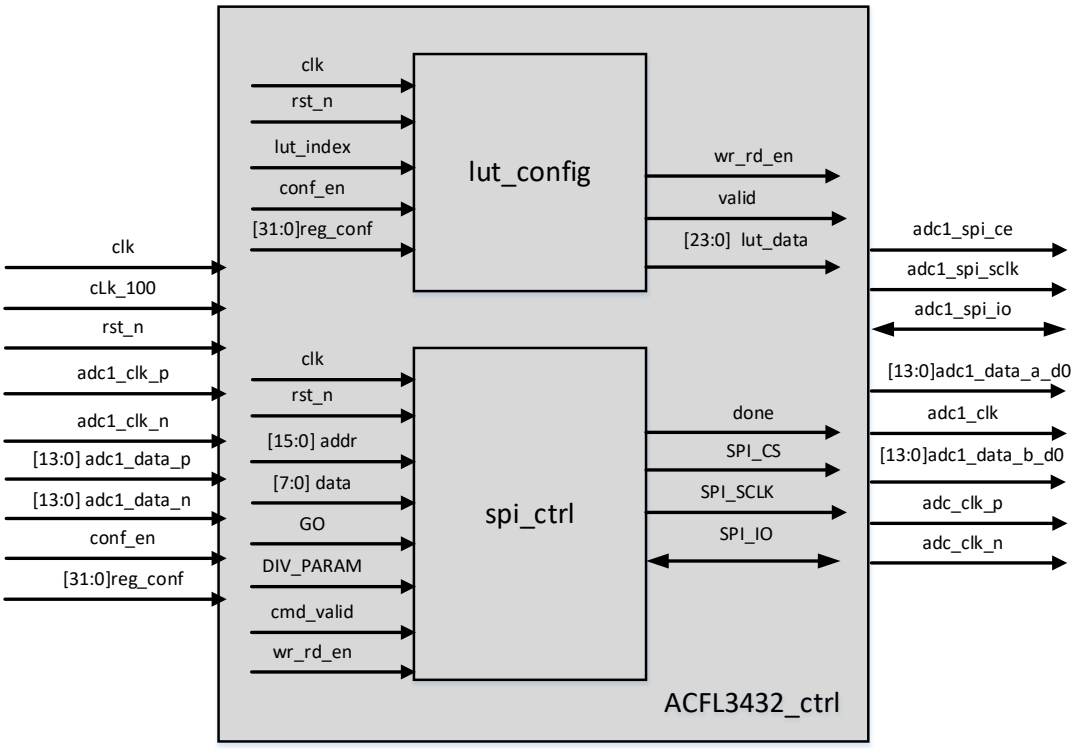

图 1-7 ACFL3432_ctrl 模块示意图

1.3.1.1.1.1配置 CM3432 寄存器操作

lut_config 模块负责对 CM3432 的寄存器进行配置，通过输入的 lut_index 信号启动状态机跳转，lut_config 模块输出对应的 25 位配置数据 lut_data（高 16 位为寄存器地址，低 8 位为写入值），该寄存器部分配置代码如下所示：


```
reg [9:0] state;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        state <= 0;
    else if (lut_index)
        state <= state + 1;
    else
        state <= state;
end

reg [3:0] conf_state;
always @(posedge clk or negedge rst_n) begin
if (!rst_n) begin
    wr_rd_en <= 1;      // 默认写
    valid    <= 0;
    conf_state <= 0;
    lut_data <= {16'hffff, 8'hff};
end else begin
    case(conf_state)
        0   :   begin conf_state <= conf_state + 1;end
```

```
        1   :   begin
                case (state)
                10'd0: begin lut_data <= {16'h0014 , 8'h05};valid <= 1; end
                10'd1: begin lut_data <= {16'h0017 , 8'h85};valid <= 1; end
                10'd2: begin lut_data <= {16'h000b , 8'h00};valid <= 1; end
                10'd3: begin lut_data <= {16'h0018 , 8'h0F};valid <= 1; end
                10'd4: begin lut_data <= {16'h00ff , 8'h01};valid <= 1; end
                10'd5: begin conf_state <= conf_state + 1; valid <= 0; end
                default: lut_data <= {16'hffff, 8'hff};
                endcase
            end

        2   :   if(conf_en_reg)begin
                    lut_data <= reg_conf[23:0];
                    valid <= 1;
                    conf_state <= conf_state + 1;
                 end else begin
                    valid <= 0;
                    conf_state <= conf_state;
                 end

        3   :    if(lut_index)begin
                    lut_data <= {16'h00ff , 8'h01};
                    valid <= 1;
                    conf_state <= conf_state + 1;
                end else
                    conf_state <= conf_state;

        4   :   begin
                valid <= 0;
                if(lut_index)begin

                    conf_state <= 2;
                  end else
                    conf_state <= conf_state;
                end
        default :   conf_state <= 0;
    endcase
  end
end
```

初始化 0 状态，向地址 0x14（输出模式寄存器）写入 0x05，也就是 CM3434 芯片输出数据格式为二进制补码，如下图所示：

OUTPUT MODE (LOC, 0x14)

位	名称	访问类型	复位	描述
7:5	RSV	RW	000	保留位。
4	OUTPUT ENABLE	RW	0	通道 ADC 输出使能。若只关闭一个通道，则关闭通道的输出会重复另一通道。
3	RSV	RW	0	保留位。
2	OUTPUT INVERT	RW	1	输出反向。 0：反向输出 1：正常输出
1:0	OUTPUT FORMAT	RW	01	输出格式。 00: 二进制移码 01: 二进制补码 10: 格雷码 11: 保留

图 1-8 输出数据格式

最后的地址 0xFF（同步更新配置寄存器）写入 0x01，使能影子寄存器，最低位使能后会将主寄存器的值更新至从寄存器，从而控制功能生效。更新完成后此位将自动清零。如下图所示：

TRANSFER (GLO，0xFF)

位	名称	访问类型	复位	描述
7:1	RSV	RW	0x00	保留位。
0	TRANSFER	RW	0	影子寄存器同步使能。该位使能后会将主寄存器的值更新至从寄存器，从而控制功能生效。更新完成后此位将自动清零。

图 1-9 影子寄存器同步使能

1.3.1.1.2 有符号数位宽扩展

ad_14bit_to_16bit 模块负责将 ADC 采集数据进行有符号数位宽扩展，并且还加入了测试数据选项，最后根据通道选择指令，将测试数据、通道 1 数据、通道 2 数据以及双通道数据进行有符号数位宽扩展处理后输出（因为当前寄存器设置之后，芯片输出的数据已经是有符号数，所以这里不需要进行有符号处理），这里将双通道数据同时拼接为 32 位数据输出使用。

```
always @(posedge clk or negedge rst_n)
if(~rst_n) begin
    ad_out <= 16'd0;
    ad_out_comb <= 32'd0;
end else if (ad_sample_en) begin
    case(ch_sel)
        2'b00: ad_out <= {2'd0,adc_test_data};  /// 测试数据
        2'b01: ad_out <= {2'd0,s_ad_in1};       /// 通道一
        2'b10: ad_out <= {2'd0,s_ad_in2};       /// 通道二
        2'b11: ad_out_comb <={2'd0,s_ad_in1,2'd0,s_ad_in2}; /// 双通道组合
    endcase
end else begin
```

```
    ad_out <= 16'd0;
    ad_out_comb <= 32'd0;
end
```

1.3.1.1.3 采样速率控制

speed_ctrl 模块用来控制 ACFL3432 模块的采样速率。

ACFL3432 模块支持最高时钟频率为 250Mhz，同时模块的最大采样速率为 250Msps，如需使用低于时钟频率的采样率，可以依旧给 ADC 提供 250MHz 的时钟信号，但在 FPGA 内部，对 250Msps 的采样结果数据进行抽取重采样的方法实现。

比如期望以 25Msps 的采样速率采样，则只需要每间隔 10 个采样数据取一个结果存储或使用，其他 9 个数据直接舍弃，这样就能实现 25Msps 的采样率了。该模块可以通过计数器计数实现所需功能，代码设计如下所示：

```
reg [31:0]div_cnt;

always@(posedge clk or negedge reset_n)
if(!reset_n)
    div_cnt <= 0;
else if(ad_sample_en)begin
    if(div_cnt >= div_set)
        div_cnt <= 0;
    else
        div_cnt <= div_cnt + 1'd1;
end
else
    div_cnt <= 0;

always@(posedge clk or negedge reset_n)
if(!reset_n)
    adc_data_en <= 0;
else if(div_cnt == div_set)
    adc_data_en <= 1;
else
    adc_data_en <= 0;
```

1.3.1.2数据流协议格式转换

fifo2axis_write 模块主要作用是将 16 位或者 32 位数据转换为符合数据流协议的数据格式。为了对接 16 位或者 32 位数据位宽数据，这里准备了两个 FIFO 来完成位宽转换。

一个 FIFO 用来缓存通道一或者通道二或者测试数据这种单通道 16 位数据，另外一个 FIFO 用来缓存两个通道同时采集的 32 位数据。两个 FIFO 配置如下，一个 FIFO 输入 16 位，输出 32 位，另外一个 FIFO 输入 32 位，输出也是 32 位。

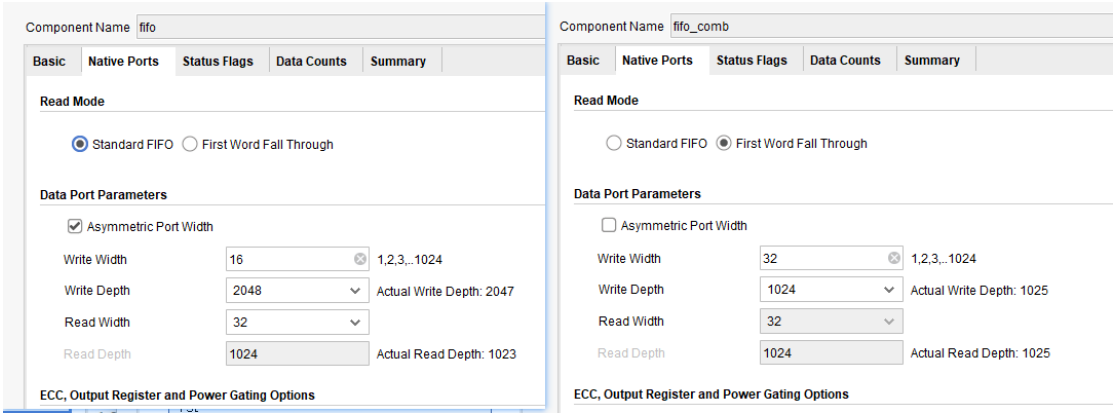

图 1-10　两个 FIFO 配置

为了控制两个 FIFO 的写入和读取相关信号，这里使用“ch_sel==2'b11”进行选择控制。

```
wire wr_fifo;
assign wr_fifo = dvalid && (state >= 2) && (ch_sel!=2'b11);

wire wr_fifo_comb;
assign wr_fifo_comb = dvalid && (state >= 2) && (ch_sel==2'b11);

wire [31:0] dout_fifo;
wire [31:0] dout_fifo_comb;
assign dout = (ch_sel==2'b11)? dout_fifo_comb : dout_fifo;

wire [9:0]rd_data_count_fifo;
wire [9:0]rd_data_count_comb;
assign rd_data_count = (ch_sel==2'b11)? rd_data_count_comb : rd_data_count_fifo;
```


fifo2axis_write 模块示意图如下所示。

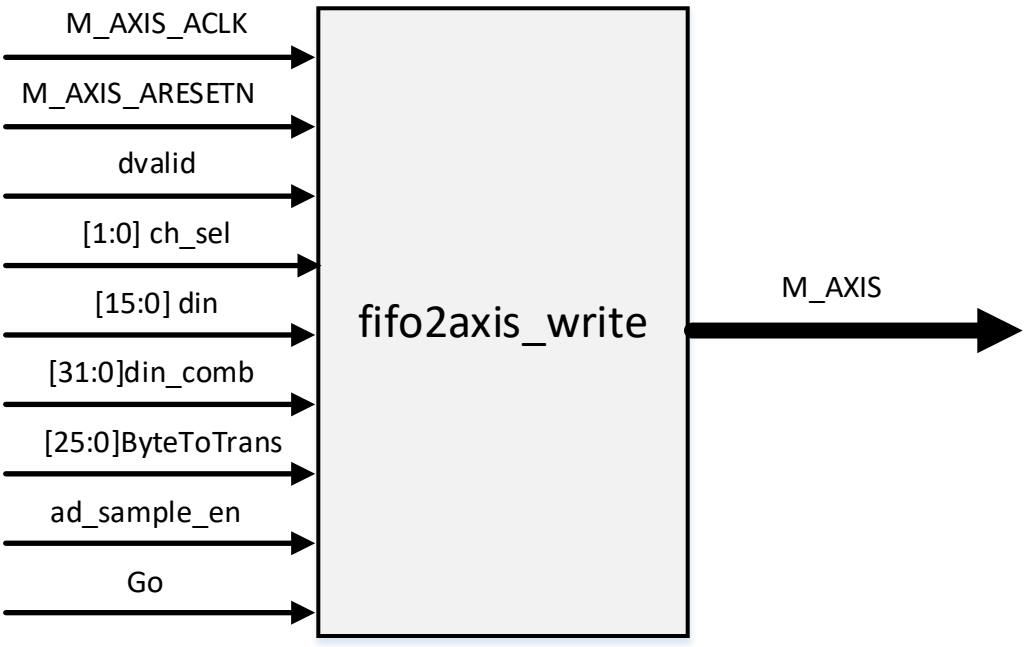

图 1-11 fifo2axis_write 模块示意图

1.3.1.3数据搬运模块

DMA IP 对接 fifo2axis_write 模块输出的数据流，将数据写入到 DDR4 中，不过本次实验中，只需要将数据写入 DDR4，并不需要 DMA IP 进行读取操作，所以 DMA IP 配置中，取消了读取通道使能选项，勾选了写入使能选项。

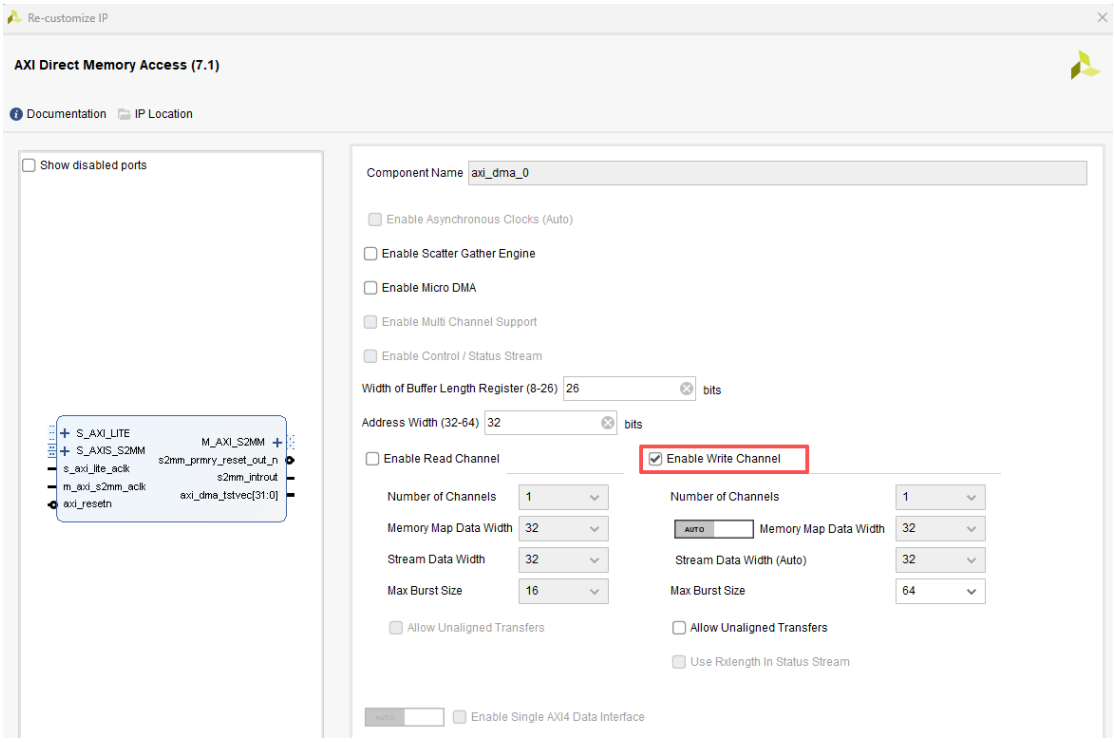

图 1-12　DMA IP 配置

1.3.1.4指令接收转控制模块

上位机下发的指令需要进行解析，这里使用自定义 IP adc_sample_ctrl 模块来解析指令，解析完成的指令给到其他模块使用。

1. 将 slv_reg0 的 bit0 位给到 ad_sample_en 信号，作为 ADC 采样使能信号；
2. 将 slv_reg0 的 bit8 位给到 Go 信号，作为 DMA 传输使能信号；
3. 将 slv_reg1 的 bit1 和 bit0 两位给到 ChannelSel 信号，作为通道选择信号；
4. 将 slv_reg2 给到 ADC_Speed_Set 信号，作为速度设置；
5. 将 slv_reg3 的低 26 位给到 ByteToTrans 信号，作为一轮 DMA 传输字节数量；
6. 将 slv_reg4 给到 Data_Sum 信号，作为采样总字节数；
7. 将 slv_reg5 给到 reg_conf 信号，作为需要修改的具体寄存器和具体值；

```
assign ad_sample_en     = slv_reg0[0];
assign Go               = slv_reg0[8];
assign ChannelSel       = slv_reg1[1:0];
assign ADC_Speed_Set    = slv_reg2;
assign ByteToTrans      = slv_reg3[25:0];
assign Data_Sum         = slv_reg4;
assign reg_conf         = slv_reg5;
```

1.3.1.5Zyng UltraScale+ MPSoC IP 配置

根据传输需求，Zyng UltraScale+ MPSoC IP 配置如下：

1.3.1.5.1 配置 Bank 电压

由硬件决定 bank 电压，这里将 Bank0 和 Bank2 电压设置为 LVCMOS18，Bank1 和 Bank3 电压保持 LVCMOS33 不变。

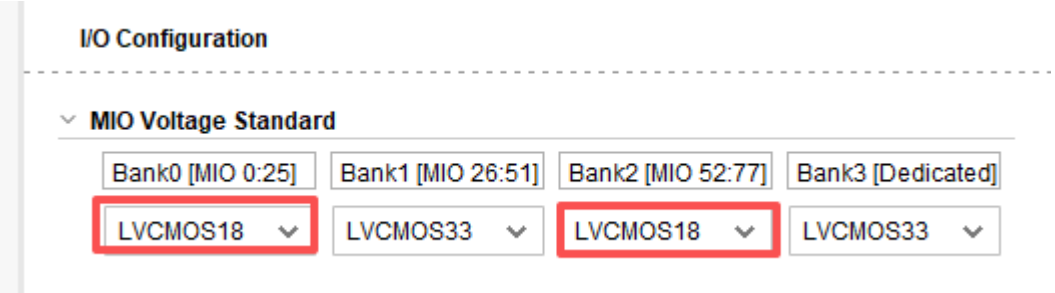

图 1-13 bank 电压设置

1.3.1.5.2 使能 PS 端千兆以太网

本章实验通过 PS 端千兆以太网使用 TCP 协议接收上位机下发的控制指令和将缓存的数据发送到上位机中进行波形绘制。所以在 Zyng UltraScale+ MPSoC IP 中勾选 GEM3，并根据原理图使用 MIO64..75 引脚作为 PS 端千兆以太网引脚，同时还需要勾选 MDIO3 信号。

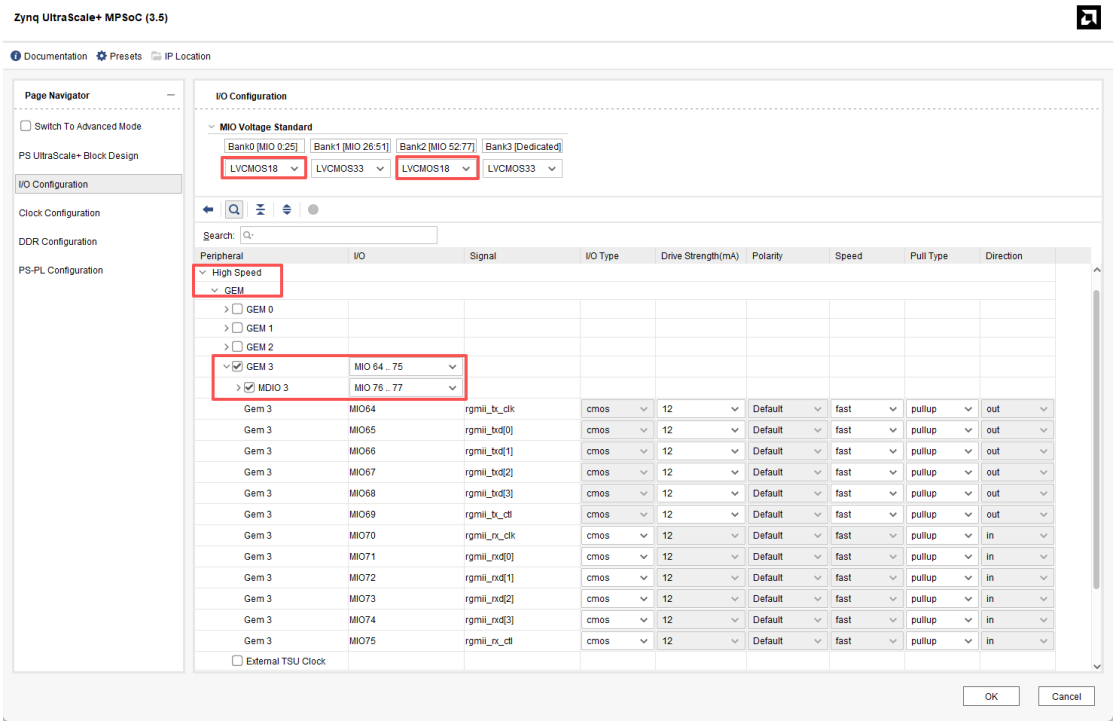

图 1-14 使能 PS 端千兆以太网

1.3.1.5.3 使能 PS 串口

使能 UART 1，引脚分配 MIO32..33，用于方便代码调试和打印相关信息。

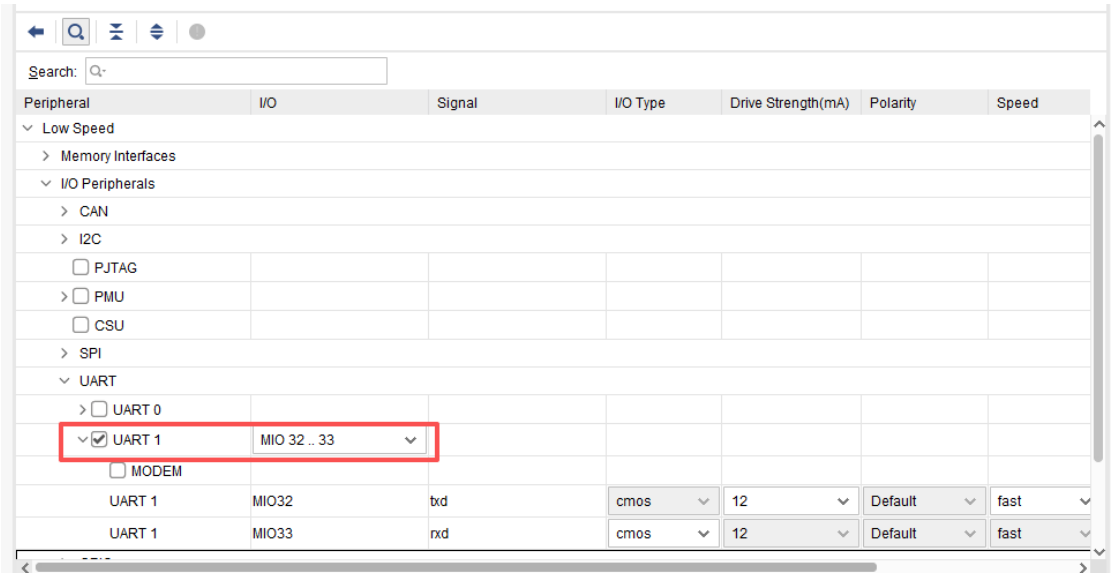

图 1-15 使能 PS 串口

1.3.1.5.4 使能固化

使能 SD 与 QSPI（方便在需要时，进行程序固化），其中 QSPI 设置如下：

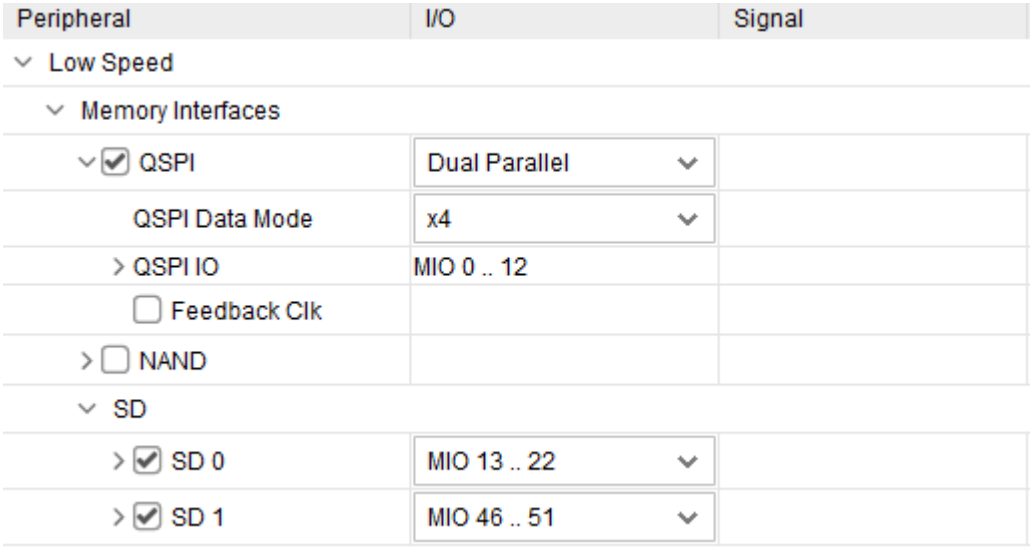

图 1-16 使能固化

其中 SD 0 设置如下：

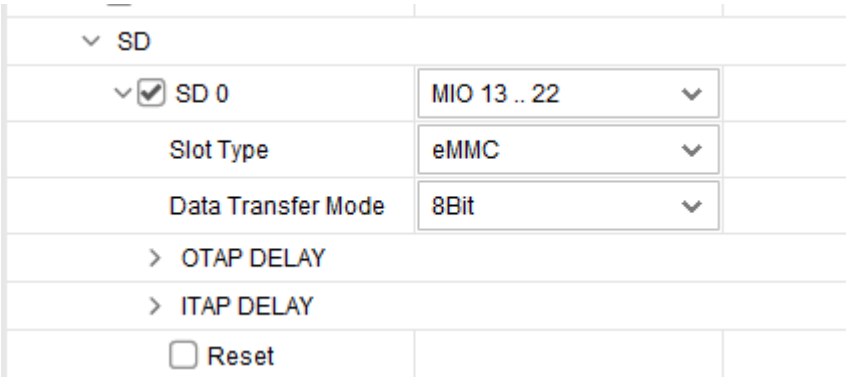

图 1-17 SD 0 设置

其中 SD 1 设置如下：

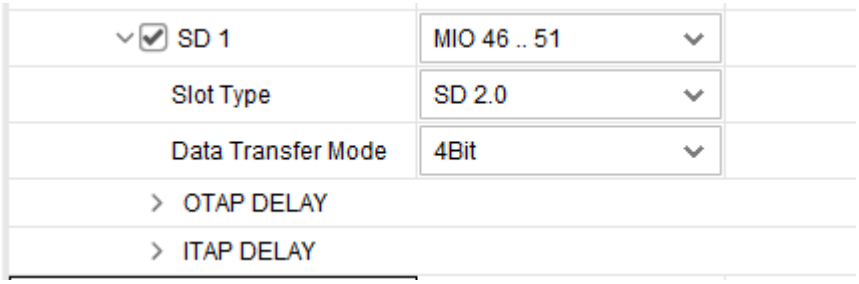

图 1-18 SD 1 设置

1.3.1.5.5 使能 AXI HP0 FPD 接口

使能 AXI HP0 FPD 接口，用于 DMA 数据的高速搬运接口。

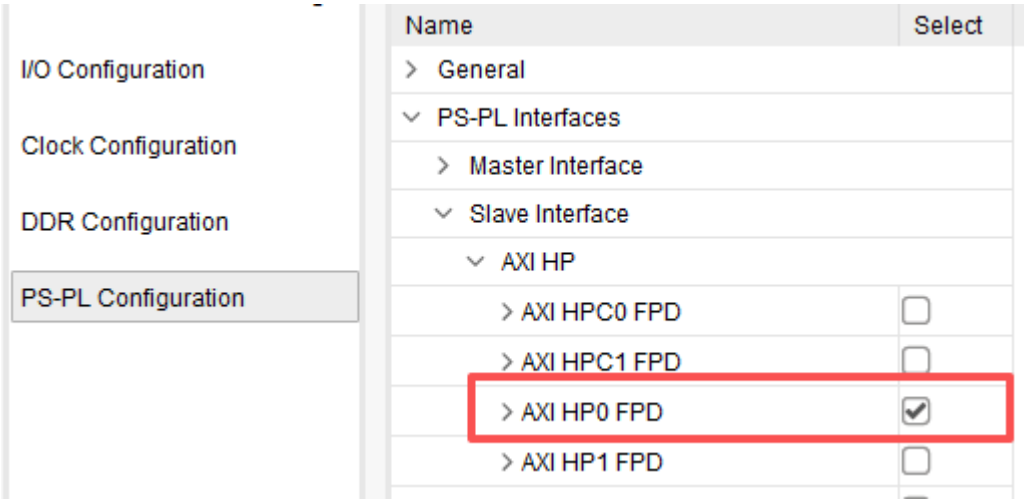

图 1-19 使能 AXI HP0 FPD 接口

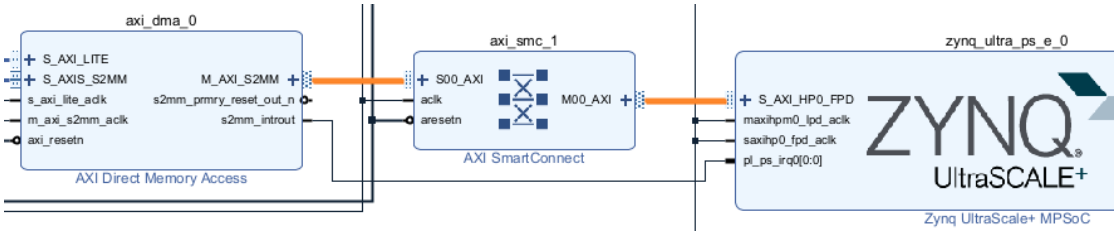

图 1-20 DMA 模块和 AXI HP0 FPD 接口连接

1.3.1.5.6 使能 AXI HPM0 LPD 接口

使能 AXI HPM0 LPD 接口，用于配置 DMA 模块和将上位机下发的指令给到自定义 IP adc_sample_ctrl 模块来解析指令。

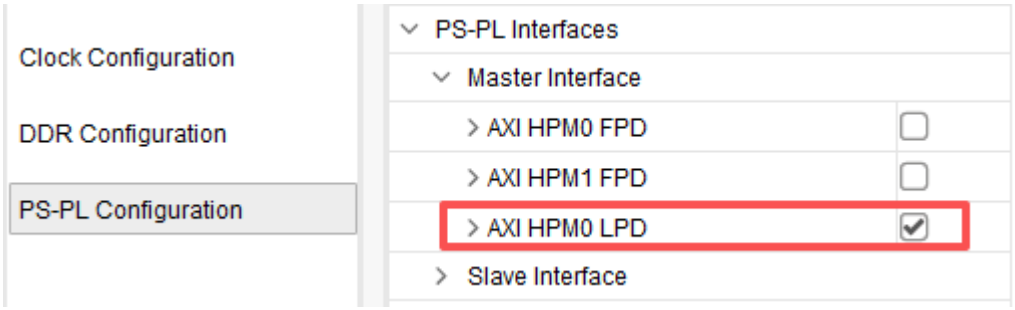

图 1-21 使能 AXI HPM0 LPD 接口

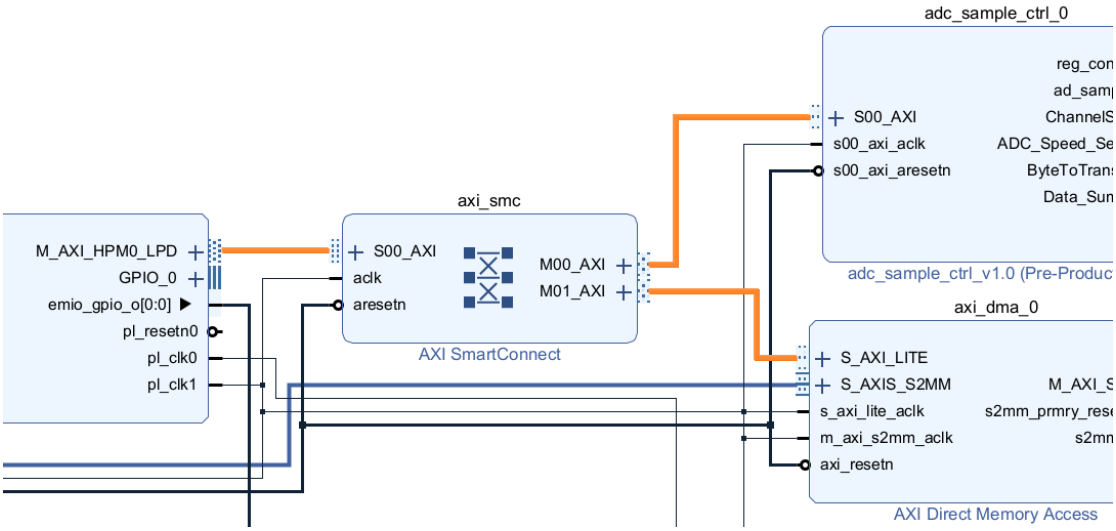

图 1-22 AXI HPM0 LPD 接口连接 DMA 模块和 adc_sample_ctrl 模块

1.3.1.5.7 使能中断

使能 PL-PS 中断（IRQ_0[0-7]）

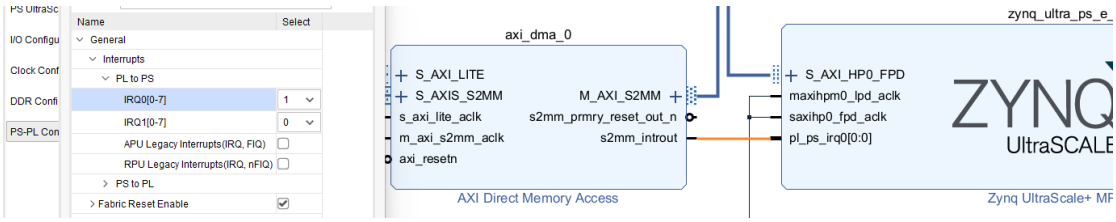

图 1-23 使能 PL-PS 中断

1.3.1.5.8 使能时钟

设置 50Mhz 和 250Mhz 时钟输出，其中 50Mhz 作为配置 CM3432 芯片的 SPI 逻辑相关工作时钟；另外的 250Mhz 提供给 ACFL3432 模块。

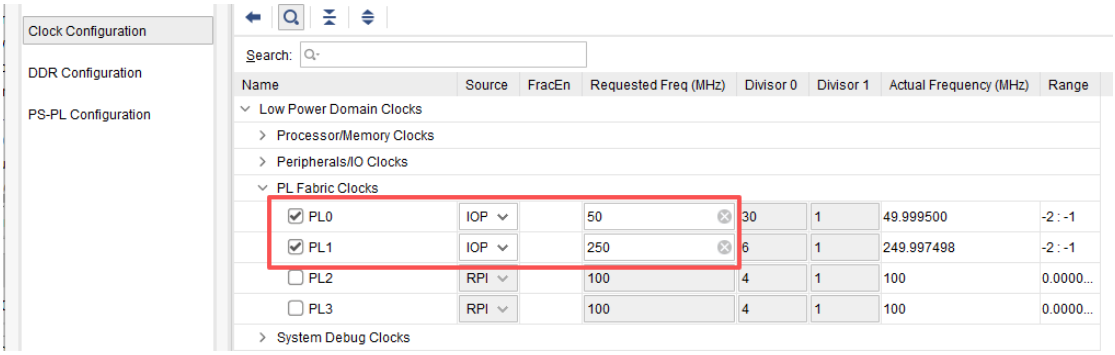

图 1-24 使能时钟

1.3.1.5.9 使能 DDR4

DDR4 设置如下：

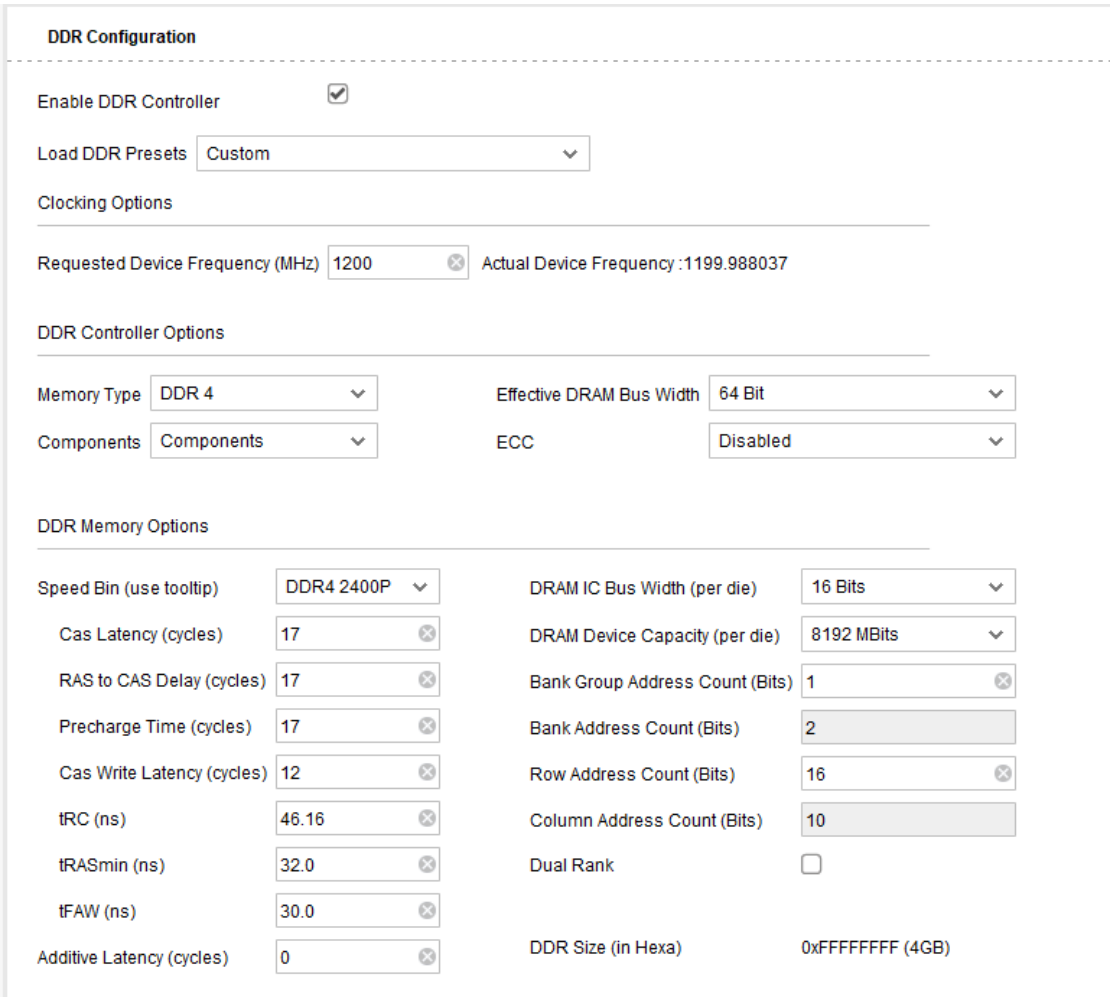

图 1-25 DDR4 相关配置

以上，便是整个 PL 端的硬件逻辑系统搭建和配置。随后是管脚约束，使用 AC920 开发板上的 FMC_LPC 接口，详细的管脚约束在例程的 xdc 文件中提供。

```
set_property PACKAGE_PIN AH14 [get_ports reset_n_0]
set_property IOSTANDARD LVCMOS33 [get_ports reset_n_0]
```

```
set_property PACKAGE_PIN D7 [get_ports adc1_clk_p_0]
set_property PACKAGE_PIN F2 [get_ports adc_clk_p_0]
set_property PACKAGE_PIN C9 [get_ports adc1_spi_ce_0]
set_property PACKAGE_PIN B3 [get_ports adc1_spi_io_0]
set_property PACKAGE_PIN B5 [get_ports adc1_spi_sclk_0]

set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports adc1_clk_p_0]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports adc_clk_p_0]
set_property IOSTANDARD LVCMOS18 [get_ports adc1_spi_ce_0]
set_property IOSTANDARD LVCMOS18 [get_ports adc1_spi_io_0]
set_property IOSTANDARD LVCMOS18 [get_ports adc1_spi_sclk_0]

##################
set_property PACKAGE_PIN G8 [get_ports {adc1_data_p_0[0]}]
set_property PACKAGE_PIN G5 [get_ports {adc1_data_p_0[1]}]
set_property PACKAGE_PIN G6 [get_ports {adc1_data_p_0[2]}]
set_property PACKAGE_PIN B4 [get_ports {adc1_data_p_0[3]}]
set_property PACKAGE_PIN D4 [get_ports {adc1_data_p_0[4]}]
set_property PACKAGE_PIN C3 [get_ports {adc1_data_p_0[5]}]
set_property PACKAGE_PIN E5 [get_ports {adc1_data_p_0[6]}]
set_property PACKAGE_PIN G1 [get_ports {adc1_data_p_0[7]}]
set_property PACKAGE_PIN G3 [get_ports {adc1_data_p_0[8]}]
set_property PACKAGE_PIN J5 [get_ports {adc1_data_p_0[9]}]
set_property PACKAGE_PIN H9 [get_ports {adc1_data_p_0[10]}]
set_property PACKAGE_PIN J6 [get_ports {adc1_data_p_0[11]}]
set_property PACKAGE_PIN L7 [get_ports {adc1_data_p_0[12]}]
set_property PACKAGE_PIN K8 [get_ports {adc1_data_p_0[13]}]

set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[13]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[12]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[11]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[10]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[9]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[8]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[7]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[6]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[5]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[4]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[3]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[2]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[1]}]
set_property IOSTANDARD DIFF_HSTL_I_18 [get_ports {adc1_data_p_0[0]}]
################
```

1.3.2 PS 端应用程序设计

PS 端的应用程序需要基于 xilinx 提供的 LWIP Echo Server 模板进行搭建，也就是在创建 platform project 时，在 standalone on psu cortexa53 0 下的 Board Support Package，需要勾选 lwip220 选项，如下图所示：

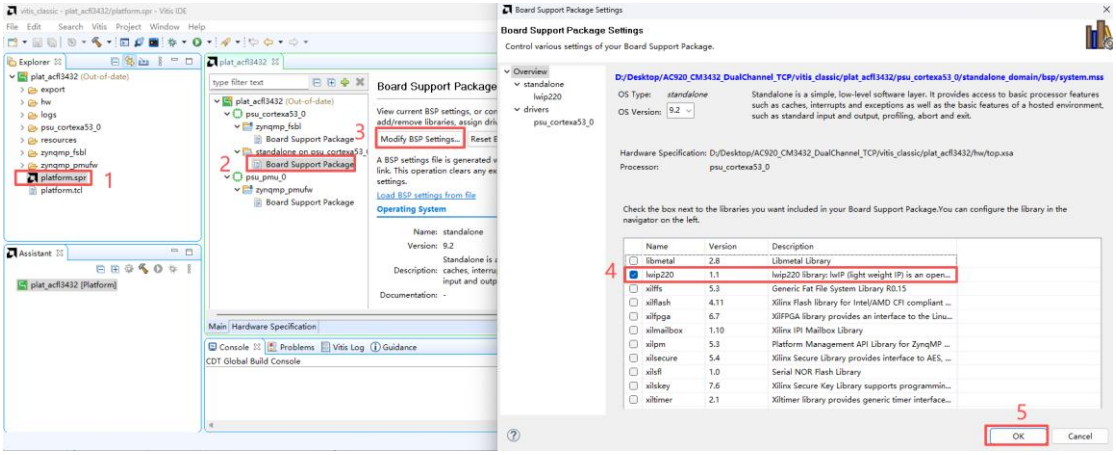

图 1-26 platform 勾选 lwip220 选项

勾选之后需要将创建的 platform 进行构建，然后在去创建 application project 时，选择 LWIP Echo Server 模板，如图所示：

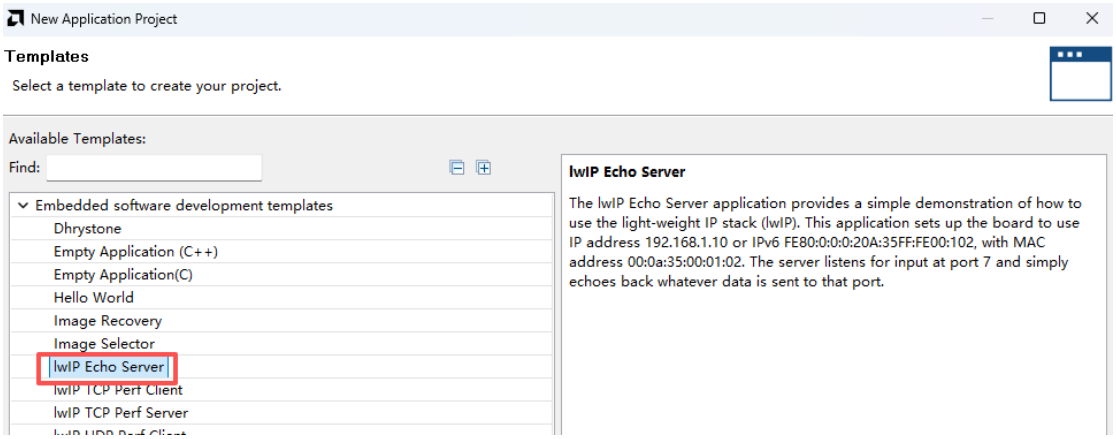

图 1-27 选择 LWIP Echo Server 模板

该模板工程会将开发板设置为服务端，并设置开发板的 IP 地址和 MAC 地址，侦听指定端口上的 TCP 数据包，回环发送。

为了实现设计目标，需要对其中的 echo.c、echo.h、main.c 等多个文件内容进行修改，由于涉及到的内容较多，用户可以直接将例程中这 16 个文件拷贝替换当前新建的工程文件。

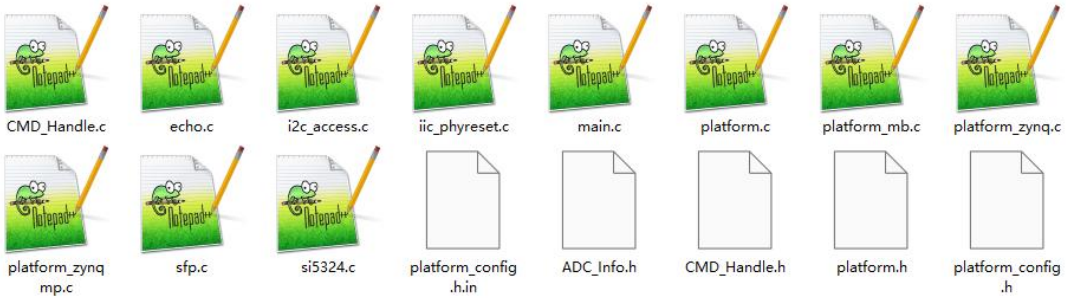

图 1-28　16 个拷贝替换文件

1.3.2.1 LWIP 功能初始化

LWIP 功能的初始化，该部分内容涉及较多，首先是 plantform 的初始化，通过 init_platform()函数完成。该函数位于 plantform_zynqmp.c 中，函数内容如下：

```
void init_platform()
{
    platform_setup_timer();
    platform_setup_interrupts();
    platform_setup_axidma();

    return;
 }
```

该函数中调用了以下三个函数：

表 1-1 硬件初始化相关函数

函数	功能
platform_setup_interrupts()	初始化 GIC，注册异常中断、私有定时器中断和 AXI DMA 接收中断的处理函数，开启 GIC 中对于私有定时器和 AXI DMA 接收中断的使能
platform_setup_timer()	初始化私有定时器，设置装载值
platform_setup_axidma()	初始化 AXI DMA，设定中断触发极性和优先级

这里的私有定时器，用于定时处理 TCP 超时事件，因此设计会用到私有定时器中断。而 AXI DMA 的数据搬运也会用到中断，为了避免出现多中断冲突的问题，设计中将 AXI DMA 中断的初始化也放在了 platform_setup_interrupts 函数中。这里 AXI DMA 的接收中断处理函数被链接为 RxIntrHandler，函数内容会在采样数据发送小节讲解。

硬件初始化完成后，代码中通过 lwip_init()函数初始化 LWIP，然后使能硬件中断，初始化网络接口。该过程中会修改开发板的 IP 地址和网关地址，为了方便实验，这里将默认的 IP 地址修改为 192.168.0.2，网关地址修改为 192.168.0.1。这部分位于 main.c 中。

```
xil_printf("Configuring default IP of 192.168.0.2\r\n");
IP4_ADDR(&(echo_netif->ip_addr),  192, 168,   0, 2);
IP4_ADDR(&(echo_netif->netmask), 255, 255, 255,  0);
IP4_ADDR(&(echo_netif->gw),      192, 168,   0,  1);
```

除了 IP 地址之外，代码中也将 MAC 地址修改为了 00_0a_35_01_fe_c0。

```
/* the mac address of the board. this should be unique per board */
```

```
unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x01, 0xfe, 0xc0 };
```

1.3.2.2建立 TCP 通信

TCP 通信的建立由 start_application()函数完成，函数位于 echo.c 中，具体代码如下：

```
int start_application()
{
    struct tcp_pcb *pcb;
    err_t err;
    unsigned port = 5000;

    /* create new TCP PCB structure */
    pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!pcb) {
        xil_printf("Error creating PCB. Out of Memory\n\r");
        return -1;
    }

    /* bind to specified @port */
    err = tcp_bind(pcb, IP_ANY_TYPE, port);
    if (err != ERR_OK) {
        xil_printf("Unable to bind to port %d: err = %d\n\r", port, err);
        return -2;
    }

    /* we do not need any arguments to callback functions */
    tcp_arg(pcb, NULL);

    /* listen for connections */
    pcb = tcp_listen(pcb);
    if (!pcb) {
        xil_printf("Out of memory while tcp_listen\n\r");
        return -3;
    }

    /* specify callback to use for incoming connections */
    tcp_accept(pcb, accept_callback);

    xil_printf("TCP echo server started @ port %d\n\r", port);

    return 0;
}
```

函数主要是实现 TCP 端口的绑定以及将服务端设置为监听模式， 代码中涉及到的主要函数如下表：

表 1-2 建立 TCP 通信相关函数

函数	功能

tcp_new_ip_type()	创建新的 TCB_PCB
tcp_bind()	将本地 IP 地址、端口号与 PCB 绑定，将绑定完毕的控制块插入 pcb 链表中
tcp_arg()	设置 TCP 连接的回调函数参数
tcp_listen()	将服务端设置为监听模式，等待 TCP 客户端的连接。当有 TCP 客户端连接时，将会调用 tcp_accept()函数进行处理
tcp_accept()	监听模式下，TCP 客户端连接时的回调函数

结合表格可知，start_application()函数依次进行了以下四个操作:

1. 通过 tcp_new_ip_type ()函数创建新 PCB；
2. 使用 tcp_bind ()绑定 IP 地址和端口号；
3. 使用 tcp_listen()对连接进行监听；
4. 使用 tcp_accept ()设置连接回调函数；

这也是 TCP 服务端与客户端建立连接的一般步骤。值得注意的是，这里的 TCP_PCB 又被称之为 TCP 控制块，是一个十分重要的结构体。其内部定义了大量的成员变量，基本定义了整个 TCP 协议运作过程的所有需要的东西，如发送窗口、接收窗口、数据缓冲区、超时处理、拥塞控制、滑动窗口等等。

除了该控制块外，为了减少资源占用，设计中还有一个专用于监听状态的控制块 tcp_pcb_listen()，这种控制块只包含 TCP_PCB 的部分字段信息。上述的 tcp_listen()在执行时，便会新建一个 tcp_pcb_listen()，从原来的 TCP_PCB 中拷贝需要用于监听的字段后，便会删除并释放原来的 TCP_PCB。当有客户端进行连接时，再将 tcp_pcb_listen()中的内容拷贝到新建的 TCP_PCB 中，并补充空缺的字段。

当有客户端进行连接时，accept_callback()便会调用 recv _callback()函数，该函数的代码如下：

```
err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err)
{
    static int connection = 1;

    /* set the receive callback for this connection */
    tcp_recv(newpcb, recv_callback);

    /* just use an integer number indicating the connection id as the
       callback argument */
    tcp_arg(newpcb, (void*)(UINTPTR)connection);

    /* increment for subsequent accepted connections */
    connection++;
    serpcb=newpcb; // 将建立的 pcb 赋予全局变量 serpcb;
    return ERR_OK;
}
```

这段代码主要实现以下 2 个功能：

1. 设置了接收回调函数 recv_callback（），并将连接 ID 作为回调参数；
2. 拷贝当前 TCP 控制块，用于主函数中服务端将 ADC 采样的数据发送到客户端；

1.3.2.3接收数据处理

当接收到客户端的数据包后，回调函数 recv_callback()会被调用。此时，我们需要它根据接收到的数据还原出指令。该函数的代码内容如下：

```
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                               struct pbuf *p, err_t err)
{
    int i;
    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);

    //对接收到的数据进行分析，还原出指令内容
    unsigned char cmd_num = (p->len)/8;//计算指令数

    if (tcp_sndbuf(tpcb) > p->len) {
        for(i=0;i<cmd_num;i++)
        {
            cmd_analysis(p->payload + 8*i);
        }

//  /* echo back the payload */
//  /* in this case, we assume that the payload is < TCP_SND_BUF */
//  if (tcp_sndbuf(tpcb) > p->len) {
//      err = tcp_write(tpcb, p->payload, p->len, 1);
    } else
        xil_printf("no space in tcp_sndbuf\n\r");

    /* free the received pbuf */
    pbuf_free(p);

    return ERR_OK;
}
```

其中涉及到的函数，功能如下：

表 1-3 函数功能表

函数	功能
tcp_close()	关闭 PCB 连接
tcp_recv()	更新接收窗口，并返回已处理的数据
tcp_sndbuf()	获取发送 buffer 的大小

pbuf_free()	释放数据包 pbuf 资源
cmd_analysis()	指令分析，将接收到的数据按照帧格式转换为对应的指令

回调函数会确认接收数据的长度，并根据长度计算出包含多少个完整的指令，只要数据不超过接收缓冲区，就会通过 cmd_analysis()函数对数据进行解析。解析完成后，通过 pbuf_free()释放掉 pbuf 所占用的内存，以免造成内存资源浪费。

这里 cmd_analysis()函数代码位于 CMD_Handle.c，具体内容如下：

```
void cmd_analysis(uint8_t *cmd_buffer)
{
    unsigned char cmd[8] = {0};
    for(int i=0;i<8;i++){
        cmd[i] = *(cmd_buffer+i);
    }

    if((cmd[0] == 0x55) && (cmd[1] == 0xA5) && (cmd[7] == 0xF0))
    {
        switch(cmd[2])
        {
            case 0: restart = 1;break;
            case 1: ChannelSel = cmd[6];break;
            case 2: DataNum = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 3: Speed_Set = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 4: reg_conf = (cmd[3] << 24) | (cmd[4] << 16) | (cmd[5] << 8) |
cmd[6];break;
            case 255: devinfo = 1; break;
            default:break;
        }
    }
}
```

函数会以每 8 个字节为一组，还原出指令内容和对应的指令数据，并给出对应的标志信号。这里支持的指令共有 5 个，指令 0~3 为一组，用来控制采样使能、采样通道、采样个数、采样速率，当采样使能为 1 时指令才会被执行；指令 255 为一组，用于上位机查询当前 ADC 的相关参数，从而方便上位机进行参数设定。

最终，这些指令和指令数据会在主函数的 while 循环中进行处理。

1.3.2.4指令处理

首先是指令 255 的处理，当上位机与开发板上的通讯端口进行连接时，会下发指令查询 ADC 的参数。对应到本次设计，所用的通讯端口便是网口 tcp 协议。设计中使用结构体的形式对 ADC 参数进行存储，该部分内容位于 ADC_Info.h 中，对应代码如下：

```
typedef struct{
    unsigned int ID;//ID 序列号
    char Label[16];//产品型号
```

```
    unsigned int Max_Sample;//当前最大采样率
    unsigned char Channel_Num;//最大通道数
    unsigned char Data_Type;//数据类型,1 为补码，0 为无符号
    unsigned char Resolution;//分辨率
    unsigned char mode;//模式
    unsigned int reserved;//预留

    float Max_Volta;//最大输入电压
    float Min_Volta;//最小输入电压
    unsigned int Clk_Freq;//工作时钟 Hz
    float Cycle;//周期 ns
}__attribute__((__packed__))ADC_info;


static const ADC_info ACM9238_info = {
    .ID = 0x00000001,
    .Label = "ACM9238",
    .Max_Sample = 50000000,
    .Channel_Num= 2,
    .Data_Type   = 1,
    .Resolution   = 12,
    .mode = 1,
    .reserved = 0,
    .Max_Volta  = 5.0,
    .Min_Volta  = -5.0,
    .Clk_Freq   = 50000000,
    .Cycle   = 20,
};
```

在主函数的 while 循环中，当检测到上位机的查询指令时，便会将该结构体中的内容通过 tcp 协议发送给上位机，代码位于 CMD_Handle.c，具体内容如下：

```
void Send_ADC_Info()
{
    if(devinfo == 1)
    {
        memcpy(buffer, &ACM9238_info, sizeof(ADC_info));
        tcp_write(serpcb,&ACM9238_info,sizeof(ADC_info),1);
        tcp_output(serpcb);
        pbuf_free(serpcb->refused_data);
        devinfo = 0;
    }
}
```

需要注意的是，这里的 tcp_write()函数只是用于组建 TCP 报文，实际上数据的发送是通过 tcp_output()函数实现。发送完成后，通过 pbuf_free()函数释放掉内存，然后将 devinfo 标志信号置 0。

由于每个 ADC 数据占两字节（14 位按两字节处理），所以这里的采样个数 DataNum 需要乘 2 得到对应的字节数。然后为了避免 FIFO 中有残余的数据对传输造成影响，这里会先通过 Xil_Out32()复位 PS 提供给 PL 的 FCLK_CLK0 时钟，从而复位整个 PL 系统，以清除 FIFO 内部。

再然后，通过 start_trans_data()函数使能 AXI DMA 接收中断，设定要接收的字节数为 num。代码位于 CMD_Handle.c，具体内容如下：

```
/*****************************************************************************
**  @brief  开始 DMA 传输
**  @param  len  待搬运字节数
**  @param  RxBufferPtr  存储数据的起始地址
**  Sample: start_trans_data(1024，0x1800000);//将 1024 字节数据搬运到以 0x1800000 为起始地址的内存空间中
*****************************************************************************/
int start_trans_data(uint32_t len,UINTPTR RxBufferPtr)
{
    int Status;

    Xil_DCacheDisable();//禁用 dcache，确保数据一致性
    /* Disable all interrupts before setup */

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    XAxiDma_IntrEnable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);

    /* Initialize flags before start transfer test  */
    RxDone = 0;
    Error = 0;

    Status = XAxiDma_SimpleTransfer(&AxiDmaInstance,RxBufferPtr,
            len, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}
```

其中的 XAxiDma_SimpleTransfer()函数便是用于向系统提交一次 AXI DMA 的传输申请，函数中，接收数据的存储起始地址被设定为 0x1800000。

设定完 AXI DMA 中断后，通过 XGpio_WriteReg()函数，使能 sample ctrl 和 speed_ctrl 两个 AXI GPIO 的通道输出。然后根据指令，设定通道输出对应的值从而配置 ADC 的重采样。

1.3.2.5 DMA 中断处理与数据发送

ADC 重采样开始后，DMA 会将重采样的数据搬运到 DDR 的指定地址中，当搬运的数据量达到预设值时，便会触发接收完成中断。此时，我们需要将结束 DMA 传输使能信号，对应的 DMA 中断处理位于 plantform_zynqmp.c，具体内容如下，如下：

```
static void RxIntrHandler(void *Callback)
{
    uint32_t IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /*
     * If no interrupt is asserted, we do not do anything
     */
    if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK)) {
        return;
    }

    /*
     * If error interrupt is asserted, raise error flag, reset the
     * hardware to recover from the error, and return with no further
     * processing.
     */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {

        Error = 1;

        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);

        TimeOut = RESET_TIMEOUT_COUNTER;

        while (TimeOut) {
```

```
            if(XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }

            TimeOut -= 1;
        }

        return;
    }

    /*
     * If completion interrupt is asserted, then set RxDone flag
     */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x100);//禁用
sample_en 和 Go; bit[8]和 bit[0]
        RxDone = 1;
    }
}
```

当中断状态为 XAXIDMA_IRQ_IOC_MASK，也就是中断完成时，通过 ADC_SAMPLE_CTRL_mWriteReg()函数将 Go 信号置 0，关闭 DMA 传输使能信号。同时，拉高接收完成标志信号 RxDone，用于下一步处理。

考虑到当需要重采样的数据量过大时，数据的搬运可能需要通过多次 DMA 中断完成。因此，当 RxDone 被拉高后，除了禁用 DMA 中断外，我们还需要判断数据是否搬运完成，从而决定是否开启下一次 DMA 中断。而当所有数据搬运完毕后，我们便可以通过 TCP 将数据传输给上位机进行波形绘制。

设计中，这部分内容通过 Data_Recive_Trans()函数实现，代码位于 CMD_Handle.c，具体内容如下：

```
/*****************************************************************************
*************************
**  @brief  数据搬运及发送
**  @param  无
**  Sample: Data_Recive_Trans();//开始 DMA 搬运，并在数据全部搬运完成后，进行 TCP 发送
*****************************************************************************
*************************/
void Data_Recive_Trans()
{
    uint32_t buff;

    if(RxDone) {
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);
        XAxiDma_IntrDisable(&AxiDmaInstance, XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
```

```
        Xil_DCacheEnable();//打开 Dcache，确保效率

        if(data_rest >0){

            RxDone = 0;
            RX_BUFFER_BASE = RX_BUFFER_BASE + 0x3ff0000;

            if(data_rest > 0x3ff0000){
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,BYTES_Trans,0x3ff0000);//
//通道选择
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x101);//使
能 sample_en 和 Go; bit[8]和 bit[0]
                start_trans_data(0x3ff0000, RX_BUFFER_BASE);
                data_rest = data_rest - 0x3ff0000;
            }
            else {
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,BYTES_Trans,data_rest);//
//通道选择
                ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0x101);//使
能 sample_en 和 Go; bit[8]和 bit[0]
                start_trans_data(data_rest, RX_BUFFER_BASE);
                data_rest = 0;
                RX_BUFFER_BASE = ADC_Data_Addr;
            }
        }
        else{
            ADC_SAMPLE_CTRL_mWriteReg(SAMPLE_CTRL_ADDR,SAMPLE_EN_GO,0);//置低
sample_en 和 Go; bit[8]和 bit[0]
            buff = tcp_sndbuf(serpcb);
            if(buff >= send_data){
                tcp_write(serpcb,(void*)(0x1800000 + uiIdx),send_data,1);
                tcp_output(serpcb);
                RxDone = 0;
                send_data = 0;
                sample_send_flag = 0;
                uiIdx = 0;
                pbuf_free(serpcb->refused_data);
            }
            else{
                tcp_write(serpcb,(void*)(0x1800000 + uiIdx),buff,1);
                tcp_output(serpcb);
                send_data = send_data - buff;
                uiIdx += buff;
                pbuf_free(serpcb->refused_data);
            }
        }
```

```
    }
}
```

为了确保 CPU 的工作效率，在 DMA 搬运完数据后，重新使能 dcache。由于用户指令给出的采样字节数可能会大于一次 DMA 中断所能传输的字节数，所以每当 DMA 接收完成中断触发后，函数都会对剩余待接收的字节数（data_rest）进行判断，根据 data_rest 的值判断是否需要使能下一次 DMA 接收中断，以及下一次 DMA 接收中断所需接收的字节数。

而为了确保接收数据的连续，代码会将 DMA 搬运的数据存储到以 ADC_Data_Addr（0x1800000）为起始地址的空间中，等到采样的 ADC 数据满足用户指令给出的字节数时，才会通过 TCP 将数据发送给上位机。

发送前需要先判断待发送数据与 tcp 发送 buffer 容量的关系，如果数据大于发送 buffer 容量，需要分多次读取发送。这里使用 uiIdex 来记录当前发送了多少字节数据，因此待发送数据的地址可以用数据起始地址 0x1800000+uiIdx 组成。

当数据发送完成后，将接收完成标志位 RxDone 拉低，将 uiIdex 清零。同时，为了避免内存的占用，还需要通过 pbuf_free()函数释放掉 tcp 发送 buffer 所占的内存。

以上，便是 PL 端逻辑系统设计和 PS 端应用程序设计的内容，接下来，我们通过实际的硬件来对设计进行验证。

1.4　板级验证

1.4.1 实验所需硬件

1. AC920 开发板 x1
2. ACFL3432 模块 x1
3. 信号发生器 x1
4. 千兆网线 x1
5. DC 电源线 x1
6. Type-c 下载线 x1

1.4.2 硬件连接

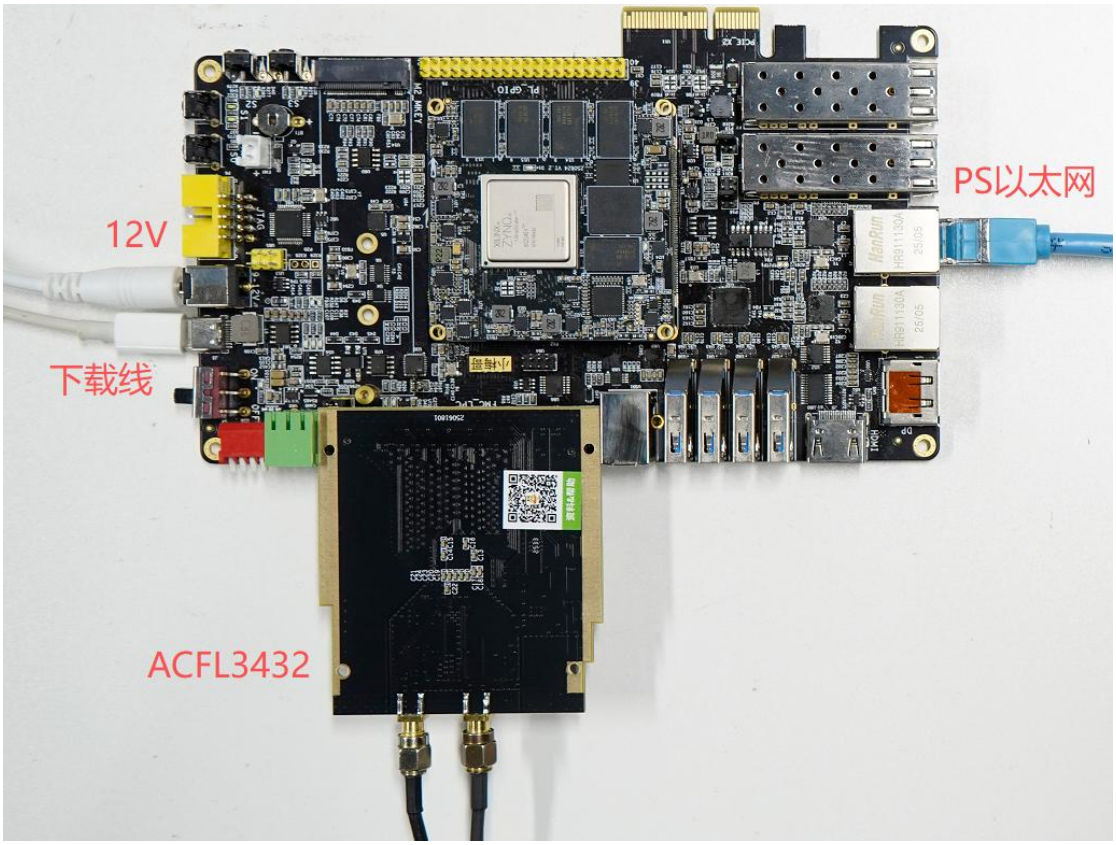

图 1-29 硬件连接图

设计使用到的是 PS 侧网口，因此使用千兆网线时，注意网线一端连接开发板 PS 网口（开发板背面丝印 PS_ETH），一端连接电脑网口，使用 1.8v 跳线帽，ACFL3432 模块与开发板上的 FMC_LPC 进行插接。连接好硬件后，为开发板上电。

1.4.3 修改电脑 IP 地址

在 LWIP 功能初始化小节中，我们通过代码将开发板 IP 地址设置为了 192.168.0.2，因此，我们需要将电脑 IP 地址设为同一网段。

在电脑上进入【控制面板】->【网络和 Internet】->【查看网络状态和任务】，查看网络连接状态。需要看到在活动网络中有以太网连接存在，才表明开发板和电脑的网络才已经连通。至于显示的无法连接到网络选项，意思是指无法连接到互联网获取网络上的数据，这是正常的，无需在意，如图：

图 1-30 查看网络连接状态

点击“以太网”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如图：

图 1-31 查看网络速度

在上述本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址（默认网关可以不设置）。如图：

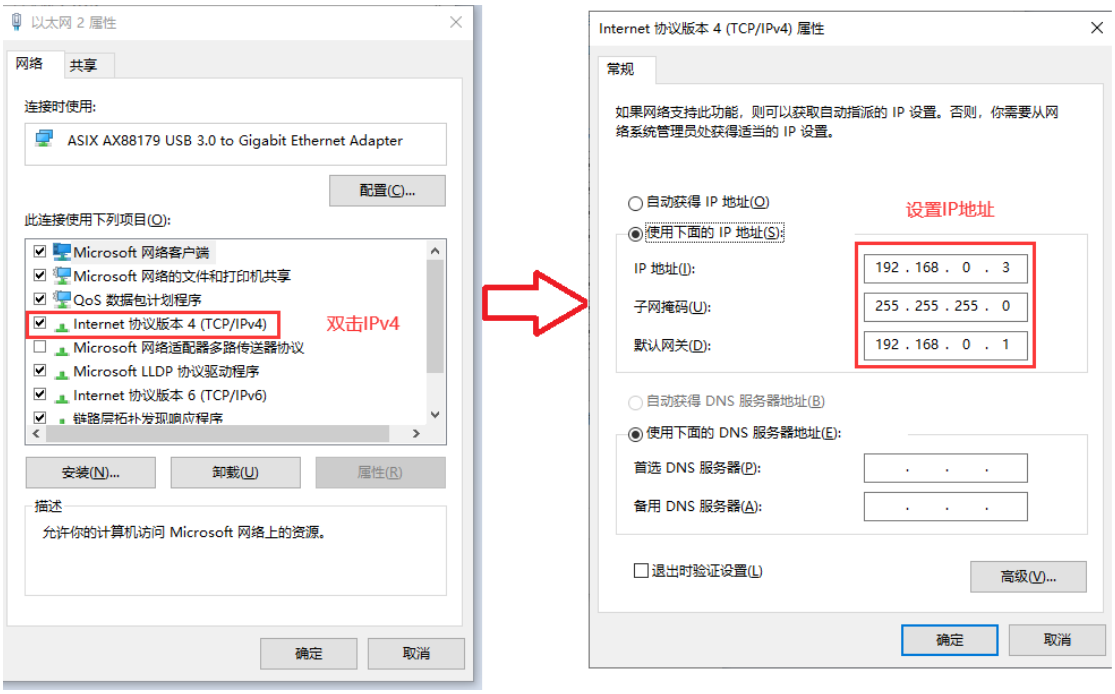

图 1-32 修改 PC 的 IP 地址

如果用户电脑中相关驱动缺失文件，会出现弹窗报错“出现了一个意外的情况，不能完成所有你在设置中所要求的更改。”导致修改失败，对于该情况用户可以参考下帖：

[设置静态 IP 时提示“出现了一个意外的情况，不能完成所有你在设置中所要求的更改”]

更改完 IP 地址后，还需要检查是否有 arp 绑定，以管理员权限打开 CMD，输入以下命令查询 arp：

```
arp -a
```

查询界面如图：

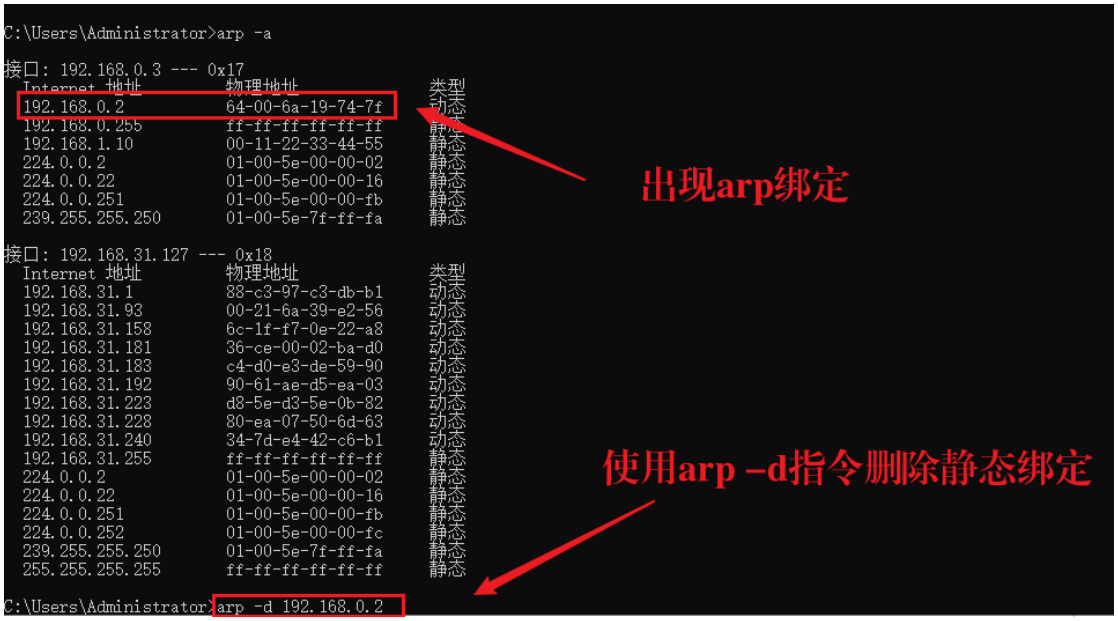

图 1-33 arp 查询与删除

如果出现了图中的静态绑定，则需要使用以下指令进行删除:

```
arp -d 192.168.0.2
```

1.4.4 代码烧录

由于 LWIP 的初始化需要一段时间，因此在烧录代码之前，我们需要在 SDK 终端中连接串口，通过串口打印的信息观察。串口的端口号需要通过设备管理查询，在端口栏中，名字为 CH9102 的便是 PS 端串口。譬如，这里笔者电脑上分配给开发板 PS 端串口的端口号为 COM20。

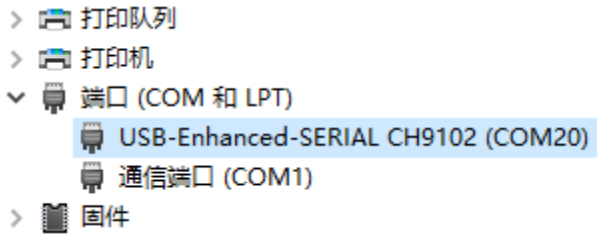

图 1-34 查询端口号

读者连接串口时，请以自己电脑上分配的端口号为准。连接完串口后，便可以烧录程序，LWIP 的初始化需要等待一段时间，当串口打印出如下图所示信息时，便代表初始化完成，此时便可以连接以太网上位机进行功能验证。

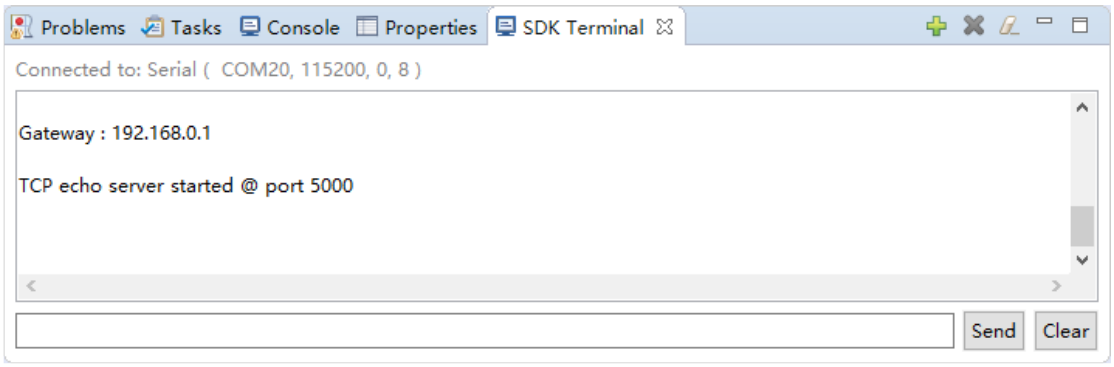

图 1-35 LWIP 初始化完成

1.4.5 功能验证

这里我们需要通过信号发生器为 ACFM3432 的 2 通道输入一路 200KHz，Vpp5V 的正弦波，1 通道输入一路 100KHz，Vpp 5V 的锯齿波。接着，通过我们提供的上位机软件“ 小梅哥数据采集仪”采集数据。该软件的最新下载链接如下所示：

小梅哥数据采集系统上位机软件使用说明【基于 QT 开发】

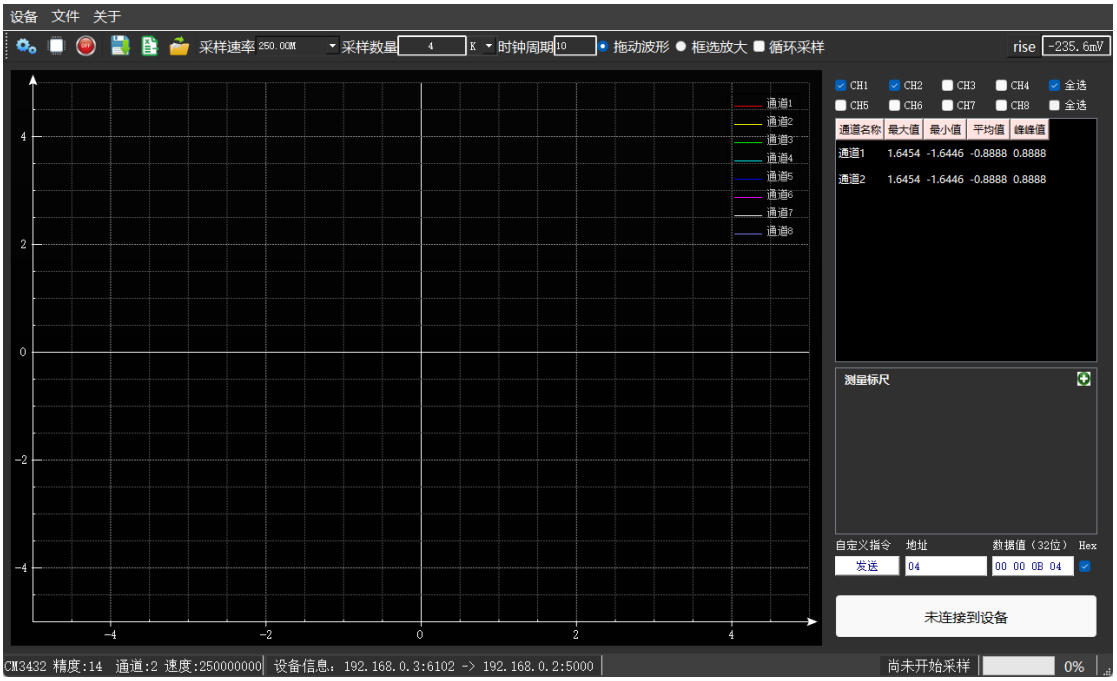

图 1-36 上位机软件初始界面显示

注：根据版本不同，软件界面可能存在一定的差异。

这里我们需要点击齿轮图标，选择通信方式为以太网 TCP 并连接，如图:

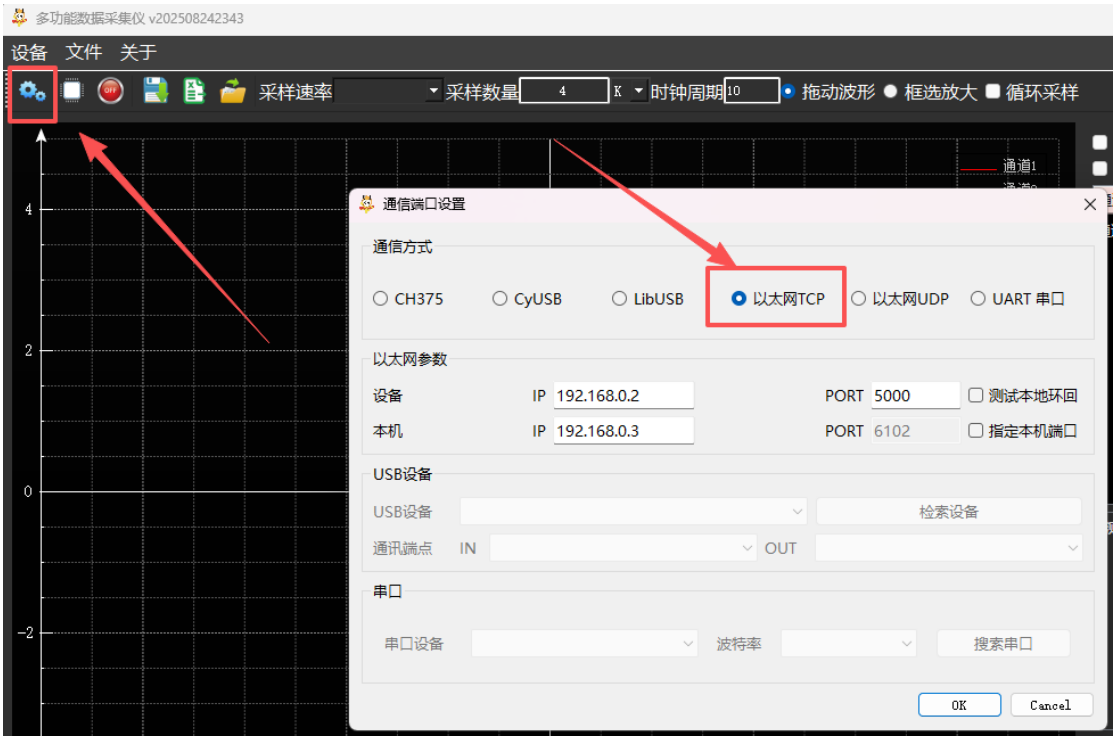

图 1-37 连接通信端口

随后，点击设备，指定 ADC 型号，这里选择 CM3432，如图

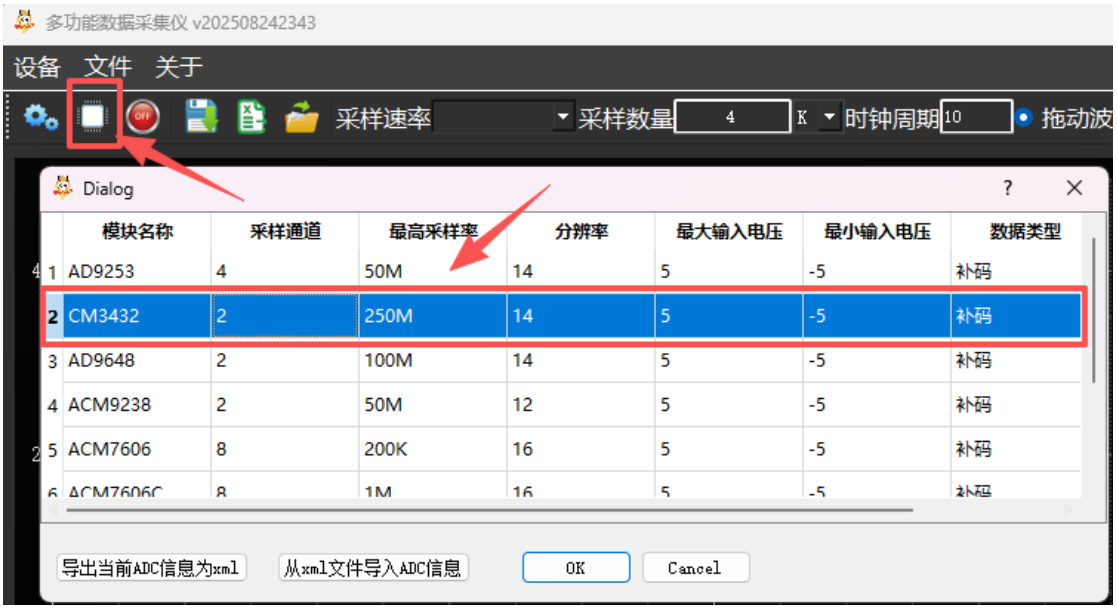

图 1-38 选择 ADC 型号

点击连接，连接成功后，软件会出现点击开始采样按钮，如图:

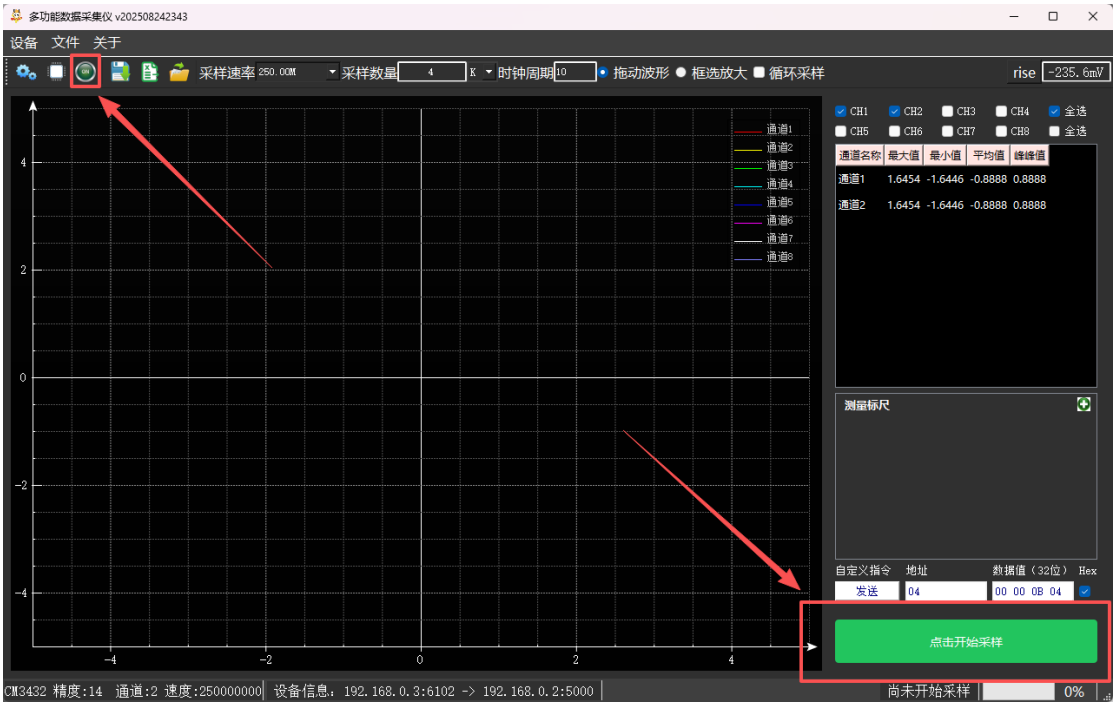

图 1-39 上位机连接成功

这里采样速率默认为 250M，采样通道默认为双通道，采样数量默认为 4K，用户也可以手动进行更改。

接下来，点击开始采样，采集到的波形如图：

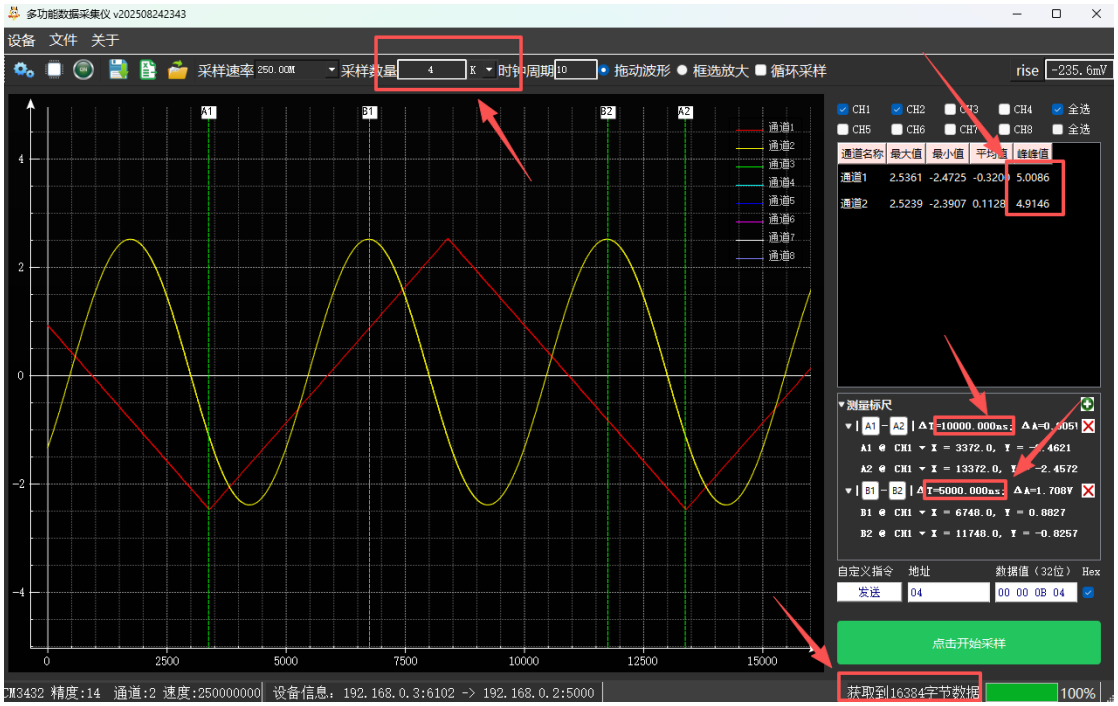

图 1-40 第一次采集波形

可以看到，通道 2 采集到的是正弦波，通道 1 采集到的是三角波，对应的峰峰值分别为 5.0086V 和 4.9146V，属于正常范围内。同时，默认设置的采样数量为 4K，每个数据大小为 2 字节，因此双通道情况下一共需要采集 4*1024*2*2=16384 字节，实际获取到的字节数与该数量一致。

经过测量，在右边 A1 和 A2 之间间隔 10000ns，也就是 100Khz；B1 和 B2 之间间隔 5000ns，也就是 200Khz。

综上所述，采集绘制的通道 1 为 100Khz 的三角波，Vpp 为 5.0086V；采集绘制的通道 2 为 200Khz 的三角波，Vpp 为 4.9146V；并且波形完整，并未出现毛刺现象。

最后，再来测试一下大数据量时的表现（数据字节数大于一次 DMA 中断的最大值，即大于 64MB），这里设置采样数量为 40M，双通道的情况下对应需要采集 160MB 字节数据。并且修改信号源的通道 1 输出的 Vpp 为 3V，通道 2 输出为 500Khz，其余的参数不变，测试结果如图：

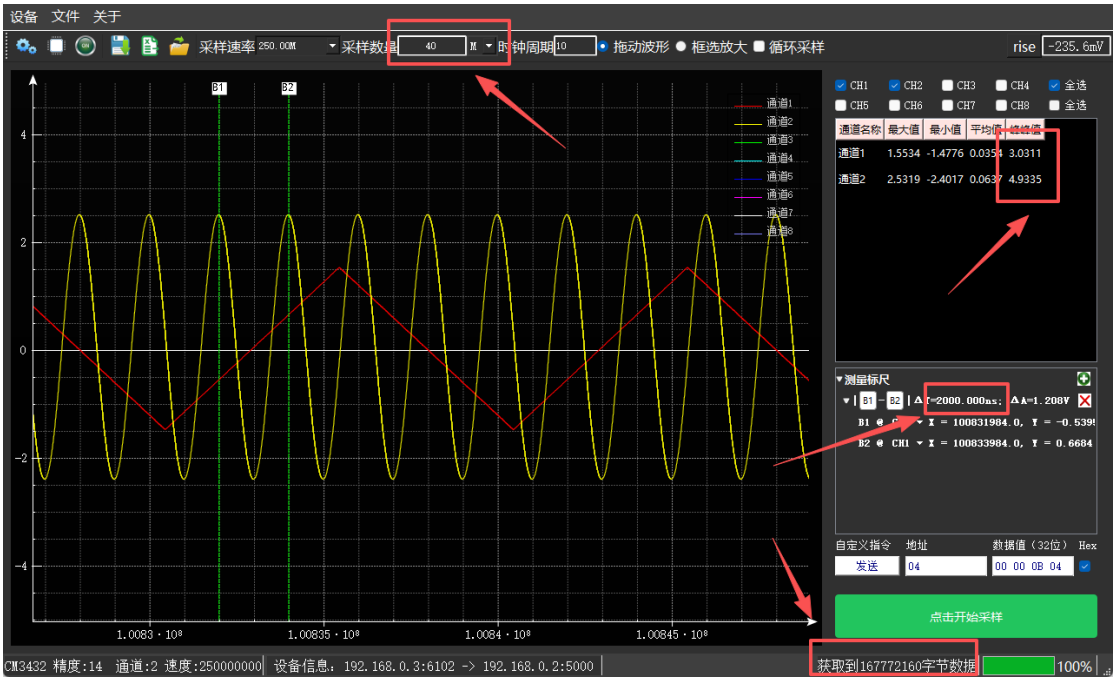

图 1-41 第二次采集（局部放大后）

可以看到通道 1 的 Vpp 为 3.0311；通道 2 的频率为 500Khz（1s/2000ns）；总共获取了 167772160 字节数据，转换后正是预期的 160MB，且采集到的波形流畅无毛刺，与预期中一致。

综上，也就说明设计无误，代码能够按照预期运行。

1.5 思考与总结

本次实验介绍了基于 ACFL3432 的双通道采集以太网 TCP 收发设计，通过 AXI DMA+DDR+以太网 TCP+上位机的方式，实现了对 ADC 的采样控制、数据存储、波形显示。通过这样一套方案，用户可以灵活简便地控制 ADC 以预期的模式运行，并且直观地观察到采样结果。

设计中只是 ACFL3432 以及以太网 TCP 为例进行讲解，实际使用时，用户也可以基于该架构，根据自身需求，修改 ADC 驱动以及通讯端口，从而应用于更多的场景。

最后，这里再为大家补充两个可能遇到的问题：

1. 无法修改 IP 为静态 IP 地址

出现这种情况，通常都是相关驱动缺少部分文件造成的，此时我们可以使用管理员权限打开 cmd，在其中输入以下代码来修改静态 IP：

```
netsh interface ip set address "以太网名" static 静态 IP 255.255.255.0
```

其中，红色部分为需要用户自己输入的部分，以太网名通常由“以太网”+“空格”+“数字”组成，用户输入时，注意不要忘记空格。

如果想将 IP 恢复到动态 IP 地址，只需要使用以下语句：

```
netsh interface ip set address name="以太网名" source=dhcp
```

2. 修改LwIP源工程模板

前面我们对模板例程的修改，只是针对本次创建的模板例程而言的。一旦我们重新生成 bsp，或者新建一个同样模板工程时，都需要再次修改。

对此，我们可以直接修改工程模板的源文件（修改前请一定要进行备份，方便恢复），具体步骤如下：

a) 找到 Vivado 安装目录下的 Xilinx\SDK\2018.3\data\embeddedsw\ThirdParty\sw_services\lwip202_v1_2\src\contrib\ports\xilinx\netif 路径下的 xemacpsif_physpeed.c 文件

b) 按照前文中的修改方法来修改它，然后保存。修改完成后，新创建的 LwIP 模板例程，都不需要再进行修改。