

1 基于 OV5640 的以太网 RGMII 图像传输系统设计

工程源码	-ACZ702 开发板 -- ov5640_udp_rgmi
相关视频课程	

如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

本节导读

FPGA 实现以太网传输的一个主要应用就是使用以太网将 FPGA 采集到的各种数据发送到 PC，继而实现如高速 ADC 实时采集数据，或者图像传感器采集图像数据等功能。这些功能在传输过程中具有数据量大，实时性要求较高的特点。如果使用基于 CPU 架构实现软件以太网协议栈，受限于 CPU 计算性能和数据组包的规律，无法达到很高的效率。而使用 FPGA 实现以太网传输，则该方案拥有稳定且高效的传输能力。

本节主要讲述了一种对数据以行为单位的编码方法。该方法采用摄像头采集数据后经由 FPGA 的 RGMII 接口通过 UDP 协议实现以太网图像传输。以该方法进行 UDP 协议下的以太网图像传输，可以有效缓解 UDP 协议的差错控制缺陷带来的传输质量不佳问题。

1.1 UDP 协议的特点

在前面的学习内容中，我们对 UDP 协议的特点和适用场景进行过分析。

我们通过以太网进行数据传输时，通常对于一些对数据正确性要求较低（允许少量丢失）但是不能有较大时延的场景，都会使用 UDP 协议。UDP 协议是传输层的一种协议，该协议具有以下特点：

- 1、不可靠传输。在进行数据传输时 UDP 提供尽最大努力的交付，但不保证可靠交付。
- 2、高效而无连接性。UDP 协议在传输数据前不建立连接，不对数据报进行检查与修改，无须等待对方的应答，所以会出现分组丢失、重复、乱序等问题，如果因为网络原因没有传送到对端，UDP 也不会给应用层返回错误信息。但正因为 UDP 在进行发送时不需要建立连接，所以其在网络开销较小

的同时工作时效也相当高。

- 3、**UDP 没有拥塞控制**。应用层能够更好的控制要发送的数据和发送时间，网络中的拥塞控制也不会影响主机的发送速率，因此 UDP 协议常常被用于某些对数据发送速率有一定要求，能容忍一些数据的丢失，但是不能允许有较大的时延的场景，如直播，视频等。

UDP 在现场测控领域，往往面向的是分布化的控制器、监测器等应用。现场测控领域的电磁环境往往较为复杂，基于此，现场通信中，若某一应用要将一组数据传送给网络中的另一个节点，可由 UDP 进程将数据加上报头后传送给 IP 进程。由于 UDP 协议省去了建立连接和拆除连接的过程，取消了重发检验机制，所以能够达到较高的通信速率。

正因为 UDP 协议有如上种种优点，所以可以对 UDP 协议图像传输的策略进行改进，而降低干扰的影响后，UDP 协议仍然可以作为图像传输的一种优质方案。

1.2 图像数据编码原理

我们前面说过：以太网图像数据的传输通常采用 UDP 协议，图像数据在传输时一旦受到外界干扰发生数据丢失，而数据的丢失会导致图像发生断层，割裂，压缩等现象。

正因为如此，我们可以进行如下有针对性的改进后，仍保留使用 UDP 协议：如果我们对图像数据进行处理，将摄像头采集到的图像数据按照一定的编号方式例如以行为单位编号后，再通过 UDP 协议经由以太网传输到电脑，电脑接收 FPGA 发送的图像数据解析后按照编号将相应的图像内容绘制到电脑显示屏的对应位置上，就能有效解决该问题。

经过该种编码方法改进，当数据在通过 UDP 协议传输的时候，理论上即使发生了少量数据丢失也不会出现断层割裂等效果。而对于图像传输显示这类场景，对于图像数据的准确性要求较低，即使一幅图像中有一两行数据的丢失，只要不是每幅图像都有该丢失情况，在每秒几十上百帧的刷新率下，其影响也极小。

为了让上述理论能够更加通俗易懂，我们结合图形，来对该理论进行讲解。

本次设计原理如图 1-1 所示：

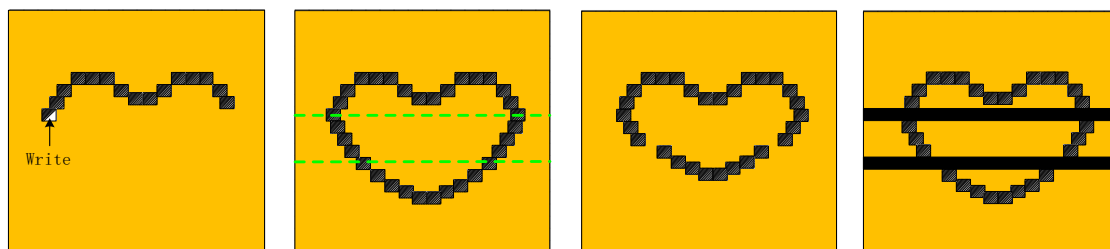


图 1-1 数据丢失后的不同结果

左边第一张为图像的写入过程，图像数据从左至右被依次写入。

第二张图为正常写入后所应展现出的画面，假设被标绿线的行在传输过程中丢失了，在未对行号进行编码的情况下，将会得到第三张图中被压缩的结果，而后续如果仍有图像传输过来，甚至可能出现两张图像重叠的情况，会影响后续图像数据的显示效果。

如果我们对每行图像数据进行编号，即使数据在传输过程中有部分行的数据丢失了，但因为序号的存在，它们仍会排列在自己所应处的位置。

举例来说：上图 1-1 中，假设绿色行发生丢失，数据在显示屏上的显示画面如第四张图，被丢失的行由于没有数据，会显示黑色，即使数据丢失，图像也不会发生压缩等现象，只会在当前帧有影响，不会影响后续帧的图像。在每秒几十帧的刷新率下，一两行数据的丢失对人眼而言甚至无法识别到，或者只是看到黑线一闪而过。

在理解了本次设计最终目的的实现原理后，接下来可以开始进行工程的设计。

1.3 系统总体设计

我们可以结合前面开发的内容，作如下设计。本次板级验证我们将在 ACZ702 开发板上结合前面章节学习过的以太网传图工程对设计模块进行验证。

本次板级验证的工程名为 `ov5640_rgmii_udp`，整个工程的系统框图如图 1-2:

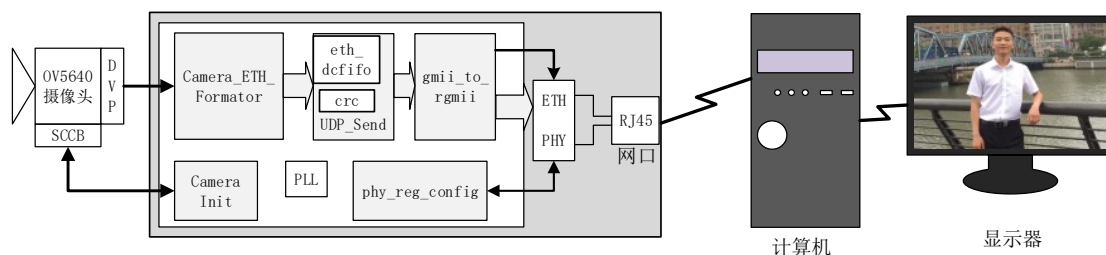


图 1-2 系统框图

结合板级验证的硬件搭建环境，本章节的原理可以以 ACZ702 硬件环境为背景作如下对应与深化讲解：

- 1、整个系统在开始工作前，首先按照初始化 table 对摄像进行初始化。
- 2、随后摄像头将采集到的数据经由 DVP 接口输送给 Camera_ETH_Formator 模块，也就是本次设计的模块，该模块会根据输入数据行场同步信号的高低电平，以行为单位，对其进行编码。
- 3、随后数据进入一个双口 FIFO 中，当 FIFO 中的值足够进行一次以太网数据传输时，数据就会被发送模块读出。
- 4、使用 UDP 协议将读出的数据经过 crc 校验后，经由以太网发送到 PC 机。
- 5、PC 端使用我们专门开发的显示软件（小梅哥 UDP 摄像头 V3.2.exe）接收 FPGA 发送的包含行编号的图像数据，解析后还原为图像内容绘制在 PC 机显示屏上。

根据前面学习内容的基础，在本章，我们着重讲解图像编码模块的作用，并讲解如何为前面讲解的实用型 UDP 收发器添加 FIFO 缓存。

1.4 图像编码模块介绍

1.4.1 图像编码模块作用

根据前面介绍的设计思想，本次模块设计的目的是实现对图像数据以行为单位进行编号。如果实现数据以行为单位编号，就需要知道当前输入的数据位于每帧图像数据的哪一行，这一需求是根据定义行同步信号和场同步信号并对其进行计数而获得满足的。

模块 Camera_ETH_Formator 在整个系统内，可以起到防止在图像显示的过程中由于某一行数据的丢失，导致的图像压缩断层现象发生的作用。在加入了该模块后，数据会按照其排列序号一行行进行排列，即使有一行或多行数据丢失，其余行数据也会排列在自己所应处的位置。

综合以上信息，设计模块 Camera_ETH_Formator 的结构如图 1-3：

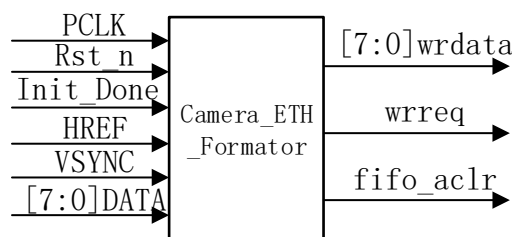


图 1-3 图像编码模块

其中各个信号的含义如表 1-1 所示：

表 1-1 Camera_ETH_Formator 模块端口列表

信号名称	I/O	位宽	信号功能
PCLK	I	1	像素时钟
Rst_n	I	1	模块复位
Init_Done	I	1	摄像头初始化完成信号
HREF	I	1	行同步信号
VSYNC	I	1	场同步信号
DATA	I	8	写入数据
wrdata	O	8	读出数据
wrreq	I	1	数据输出使能

当 HREF 由低电平转为高电平时，DATA 开始输入到图像数据编码模块，当 HREF 从高电平转为低电平时代表一行的数据输出完成，当模块中的数据处理完成后便会产生 wrreq 信号，并输出 8 位的 wrdata 数据。

我们以 1280*720 的显示屏为例，在摄像头将采集到的 RGB565 格式数据通过以太网发送给 PC 的显示屏显示的过程中，每一行有两倍 1280 个字节即 2560 字节有效数据。依照前文我们介绍的设计思想：为了让部分数据即使在丢失的情况下其余行数据也能显示在自己应处的位置，我们可以在每一行的数据前加上 2 个字节的行号。也就是说，以太网每一帧，需要发送一行的数据外加两个字节的行号。

1.4.2 图像编码模块功能实现

为了方便理解 and 设计，我们绘制了本次模块设计各个信号之间的时序图，如图 1-4:

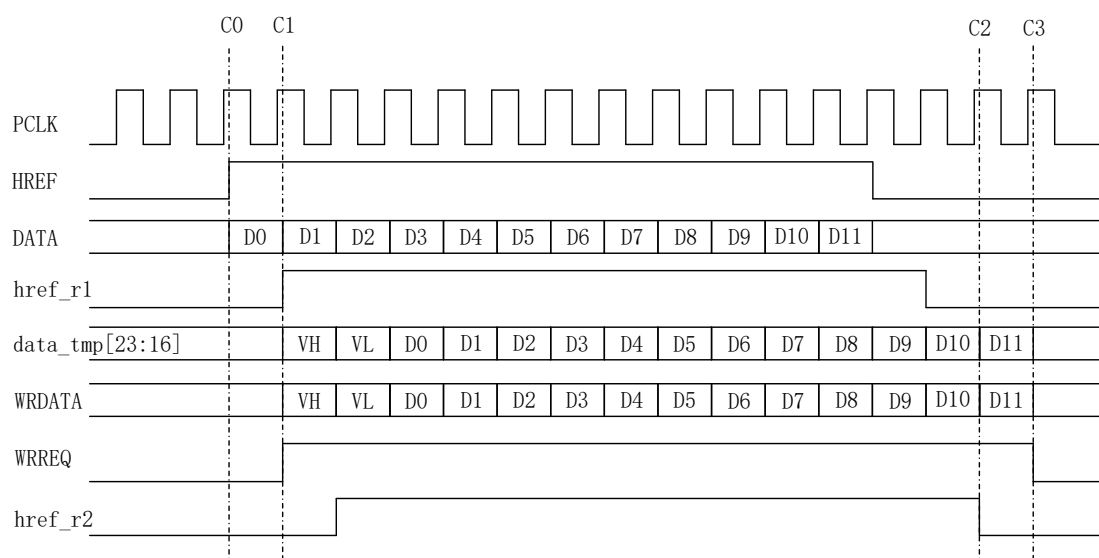


图 1-4 图像编码模块时序图

上图中：href_r1 和 href_r2 为对摄像头行同步 HREF 信号寄存后得到的信号，data_tmp 为 24 位的寄存器，用来完成对数据的移位转换。

如：时序图中 C0 时刻，当 HREF 出现上升沿时，代表数据开始输入。此时我们可以用 HREF 与 href_r1 位拼接后的值表示，当 { href_r1, HREF } 的值为 2'b01 时，HREF 的电平由低到高，数据开始输入；当 { href_r1, HREF } 的值为 2'b10 时，HREF 的电平由高到低，一行数据输入完毕，数据停止输入。

本次模块设计的目的是以行为单位，对数据进行编号，根据 HREF 的电平变化，我们可以利用计数器对行信号进行下降沿次数的累加生成行号，该部分代码实现如下：

```
reg [15:0]Vcnt;

always@(posedge PCLK or negedge Rst_n)
if(!Rst_n)
    Vcnt <= 0;
else if(VSYNC)
    Vcnt <= 0;
else if({href_r1,HREF} == 2'b10)
    Vcnt <= Vcnt + 1'd1;
```

代码中：Vcnt 为行号计数器，每当 HREF 电平状态由高变低时，代表一行数据输出完毕，此时我们只需使计数器自加一即可，当 VSYNC 信号为高电平，即代表一幅图像数据输入完成，此时将计数器清零。

完成了行号的产生，接下来就是将行号写入数据中。如图 1-4 中所示，C0 处检测到 HREF 信号的上升沿，由于该上升沿在时钟上升沿之后才到来，所以

实质上该信号是在下一个时钟上升沿到来时才被采集到。该信号被采集后，随即将行号写入 data_tmp 寄存器中。当 2 字节的行号输出完成后，再开始输出图像数据，相应的代码如下：

```
assign wrdata = data_tmp[23:16];

always@(posedge PCLK)begin
    href_r1 <= HREF;
    href_r2 <= href_r1;
end

always@(posedge PCLK or negedge Rst_n)
if(!Rst_n)
    data_tmp <= 0;
else if({href_r1,HREF} == 2'b01)
    data_tmp <= {Vcnt[7:0],Vcnt[15:8],DATA};
else
    data_tmp <= {data_tmp[15:0],DATA};
```

这里需要注意的一点是，网络在传输数据时，使用的是大端字节序，即先存放高位再存放低位，而计算机在存储数据时使用的是小端字节序，即先存放数据的低位再存放数据的高位，所以在输出行号时，为了防止数据错误，需要将其高八位与低八位的位置互换，这也就是为什么代码中 Vcnt[7:0]先输出而 Vcnt[15:8]后输出。

确定了数据的输出顺序后，接下来便是确定数据的输出使能信号。

观察时序图，每当 data_tmp 中被写入数据时，即 C1 时刻，输出使能信号 wrreq 转为高电平。此时 wrreq 信号有效，wrdata 开始输出数据，其相应的表达式为{href_r1,HREF}=2'b01；当 data_tmp 中数据输出完毕时，即 C3 时刻，wrreq 转为低电平。此时 wrreq 信号无效，wrdata 停止输出数据，相应的表达式为{ href_r2 | href_r1} = 1。

该部分代码实现如下：

```
always@(posedge PCLK)
if({href_r1,HREF} == 2'b01)
    wrreq <= 1;
else if(href_r2 | href_r1)
    wrreq <= 1;
else
    wrreq <= 0;
```

另外，我们在设计之中需要给出缓存 fifo 的清零时机。当摄像头初始化完成，且场同步信号拉高时，给出清零条件。具体实现代码如下：

```
always @ (posedge PCLK or negedge Rst_n)
```

```
if (!Rst_n)
    fifo_aclr <= #1 1'b1;
else if (Init_Done && VSYNC ) //等到初始化摄像完成且头场同步信号出现，释放清零信号，开始写入数据
    fifo_aclr <= #1 1'b0;
else
    fifo_aclr <= #1 fifo_aclr;
```

至此该模块的设计便完成了，接下来就是对本次模块设计的仿真验证。

1.4.3 图像编码模块仿真验证

1.4.3.1 图像编码模块仿真文件设计

在进行仿真验证时，仿真程序验证的激励部分可以作如下设计：

- 1、令 DATA 在 HREF 信号为高电平时从零开始自加。
- 2、令场同步信号 VSYNC 为低电平，将行同步信号 HREF 设为高电平。
- 3、延时一段时间后将行同步信号 HREF 置为低电平。
- 4、重复数次，观察各数据关系是否正确。
- 5、令场同步信号变为高电平，延时一段时间后拉低，再将 HREF 信号设为高电平，延时一段时间后变为低电平，观察行号是否重新进行计数。

该部分代码设计如下：

```
`timescale 1ns/1ns

module Camera_ETH_Formator_tb;

    reg Rst_n;
    reg PCLK;
    reg HREF;
    reg VSYNC;
    reg [7:0]DATA;

    wire [7:0]wrdata;
    wire wrreq;

    initial PCLK = 1;
    always#10 PCLK = ~PCLK;

    Camera_ETH_Formator Camera_ETH_Formator(
        .Rst_n(Rst_n),
```



```
.PCLK(PCLK),
.HREF(HREF),
.VSYNC(VSYNC),
.DATA(DATA),

.wrddata(wrddata),
.wrreq(wrreq)
);

initial begin
    Rst_n = 0;
    HREF = 0;
    VSYNC = 0;
    #201;
    Rst_n = 1;
    repeat (10)begin
        HREF = 1;
        #201;
        HREF = 0;
        #201;
    end
    #201;
    VSYNC = 1;
    #201;
    VSYNC = 0;
    repeat (10)begin
        HREF = 1;
        #201;
        HREF = 0;
        #201;
    end
    $stop;
end

always@(posedge PCLK)
if(!Rst_n)
    DATA <= #1 0;
else if(HREF)
    DATA <= #1 DATA + 1'b1;
else
    DATA <= #1 DATA;

endmodule
```

设计完激励测试文件后，接下来进行仿真，进而得到仿真输出结果。

1.4.3.2 仿真结果分析

本次仿真文件为 Camera_ETH_Formator_tb，仿真后得到的波形如图 1-5 所示：

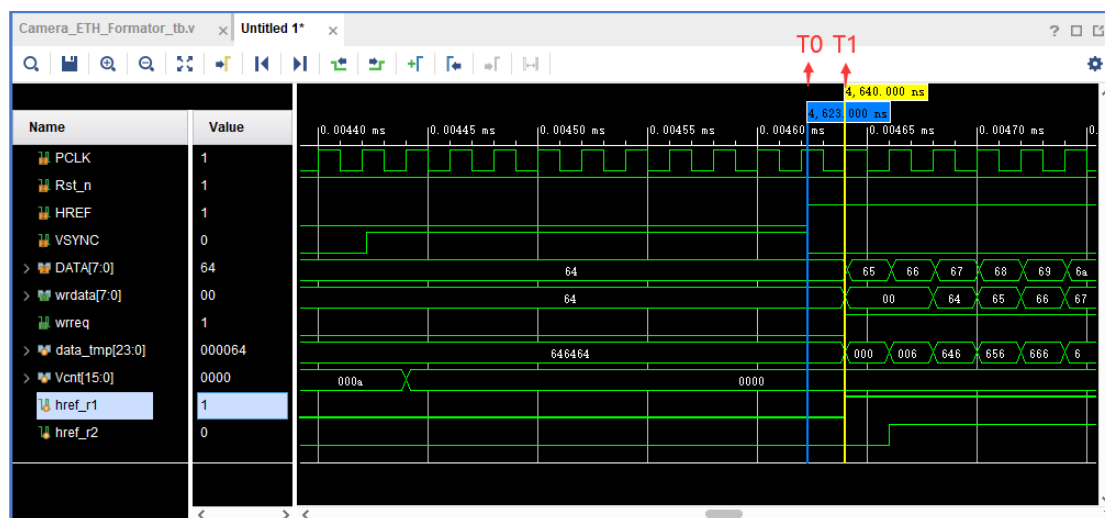


图 1-5 仿真波形图

为了方便描述不同时刻的波形，我们将图 1-5 中左右两个时刻分别命名为 T0 和 T1。可以看到 T0 时刻场同步信号 VSYNC 由高电平转为低电平，行同步信号 HREF 由低电平转为高电平，模拟了一幅图像的第一行数据开始传输。接下来对信号波形进行分析：

行计数器 Vcnt 变化值分析

T0 时刻行场同步信号发生变化，由于该变化发生在时钟上升沿之后，所以数据在下一个时钟沿到来才会变化。根据 D 触发器的特性，数据 DATA 会经过一小段的延迟即 T1 时刻后再发生改变。T1 时刻，按设计，行计数器 Vcnt 值应该为 0，仿真中行计数器 Vcnt 的值确实为 0，证明该部分逻辑设计正确。

数据移位寄存器 data_tmp 变化值分析

T1 时刻，Vcnt 的值被写入 data_tmp[23:8]中，输入数据 DATA 被写入 data_tmp[7:0]中，由于此时 Vcnt 为 0，DATA 为 16'h64，所以 T1 时刻 data_tmp 的值为 24'h000064。

在下一个时钟上升沿 data_tmp[23:16]将会被输出，data_tmp 中的数据左移 8 位，新输入的 DATA 的值被写入到 data_tmp[7:0]中。可以看到仿真中得到的结

果与理论上应该得到的一致，因此这部分仿真逻辑正确。

输出数据 wrdata 变化值分析

wrdata 在每个时钟，上升沿都会输出 data_tmp 中的高 8 位。由于 T1 时刻 data_tmp 的值为 24'h000064，所以从 T1 时刻开始的三个时钟周期，wrdata 输出的值依次应该为 16'h0， 16'h0， 16'h64。仿真波形中的结果与该数据一致，因此该部分逻辑设计正确。

需要注意的一点是，wrdata 在输出 data 数据时是按顺序输出的。而在输出行号数据时，为了匹配大端字节序与小端字节序的顺序，我们将行号的高 8 位与低 8 位数据进行了互换。所以 wrdata 输出的 16'h0， 16'h0， 16'h64 分别对应 T1 时刻的 Vcnt[7:0]， Vcnt [15:8]， DATA。

输出使能信号 wrreq 变化值分析

wrreq 为 wrdata 的使能信号，每当检测到有数据输入时有效，变为高电平；当检测到没有数据输入后该信号将变为低电平。仿真结果正确，因而该部分逻辑代码设计正确。

1.5 phy_reg_config 控制器模块例化

我们在前面 phy_reg_config 控制器设计的章节，也专门着重讲解了 phy_reg_config 控制器的用途和实现方法。它的任务主要是向网卡控制芯片写入配置寄存器初值。在这里，我们直接将该模块在顶层进行例化即可。

```
wire phy_init;
phy_reg_config phy_reg_config_inst(
    .Clk(clk_50m),
    .Rst_n(global_rst_n),
    .Phy_rst_n(eth_rst_n),
    .mdc(eth_mdc),
    .mdio(eth_mdio),
    .Phy_init_done(phy_init)
);
```

1.6 实用型 UDP 收发器详细设计

数据在实现了编号后即可进行输出，所以我们需要一个输出信号，由于 UDP 协议需要大数据量的传输，在信号输出之前我们可以使用一个 FIFO 将编序好的行输出数据进行缓存。数据的存储速率和读出速率不同，所以我们这里需

要使用异步时钟 dcfifo。设计了 FIFO 以后，还可以通过设计一个写请求信号接口实现数据能够有序而受控写入。

1.6.1 实用型 UDP 收发器 FIFO 缓存配置

由于图像编码模块一次只能输出一个字节的的数据，因此数据在通过以太网发送之前，可以利用一个 FIFO 进行缓存，考虑到以太网工作的时钟为 125MHz，图像编码模块的工作时钟为 50MHz，这里我们使用了一个异步 dcfifo。每当 wrreq 为高电平时图像编码模块将数据输出并存储到 FIFO 中，当 FIFO 中的值足够以太网进行一次发送时，便给出读请求信号，以太网从中读取出前面介绍过的带行号的 2562 字节数据，并经过 UDP 编码后发送到 PC 上。

本次实验中由于输入输出数据位宽为 8 位，所以 FIFO 的输入输出位宽设为 8。因为以太网一次发送需要从 FIFO 中读取最少 2562 字节的数据，所以这里我们设置一个大于该值的存储深度 4096 即可。

为了方便实验，勾选输入输出端口的 rd_data_count 和 wr_data_count 信号，其余配置保持默认，这样可以方便知道当前 FIFO 中数据写入情况。

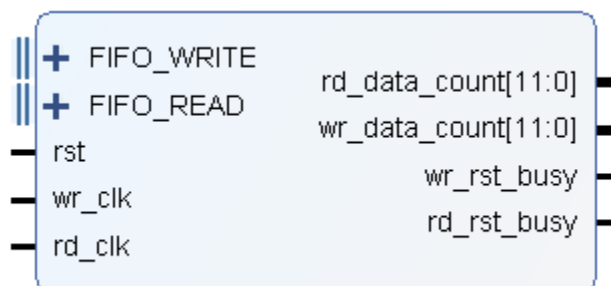


图 1-6 UDP 收发器缓存端口

本次设计 dcfifo 的配置如图 1-7、图 1-8、图 1-9、图 1-10 和图 1-11 所示：

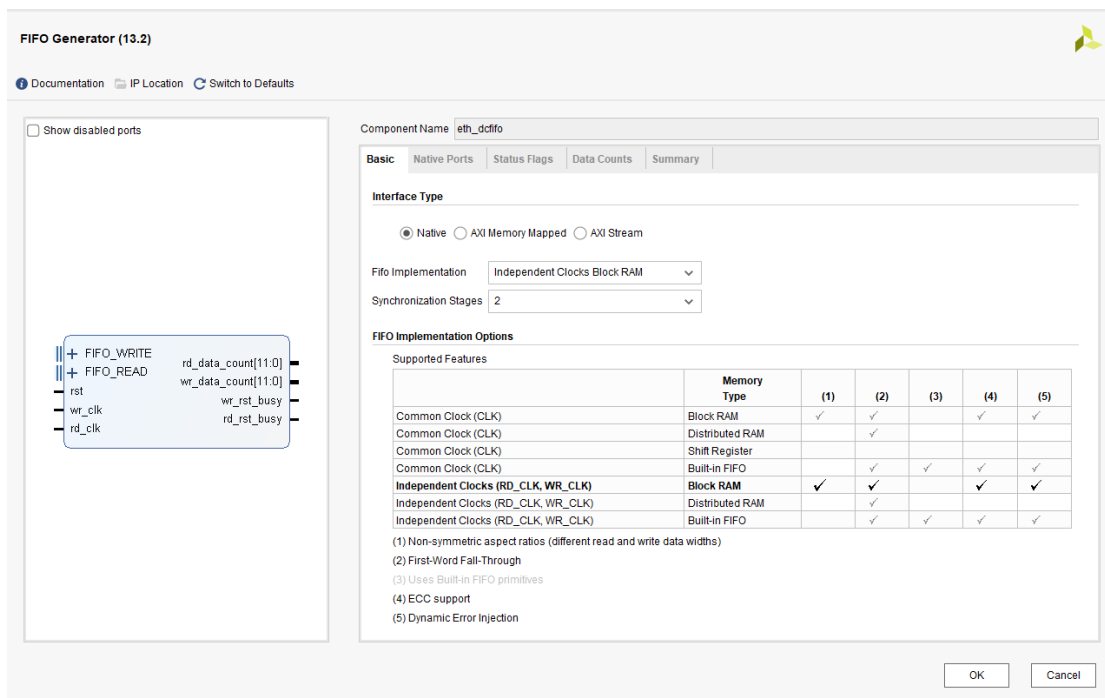


图 1-7 dcfifo 配置 1

fifo 模块配置存储深度大于 2562，这里配置为 4096。同时，特别注意 Native Ports 选择 First World Fall Through 模式。

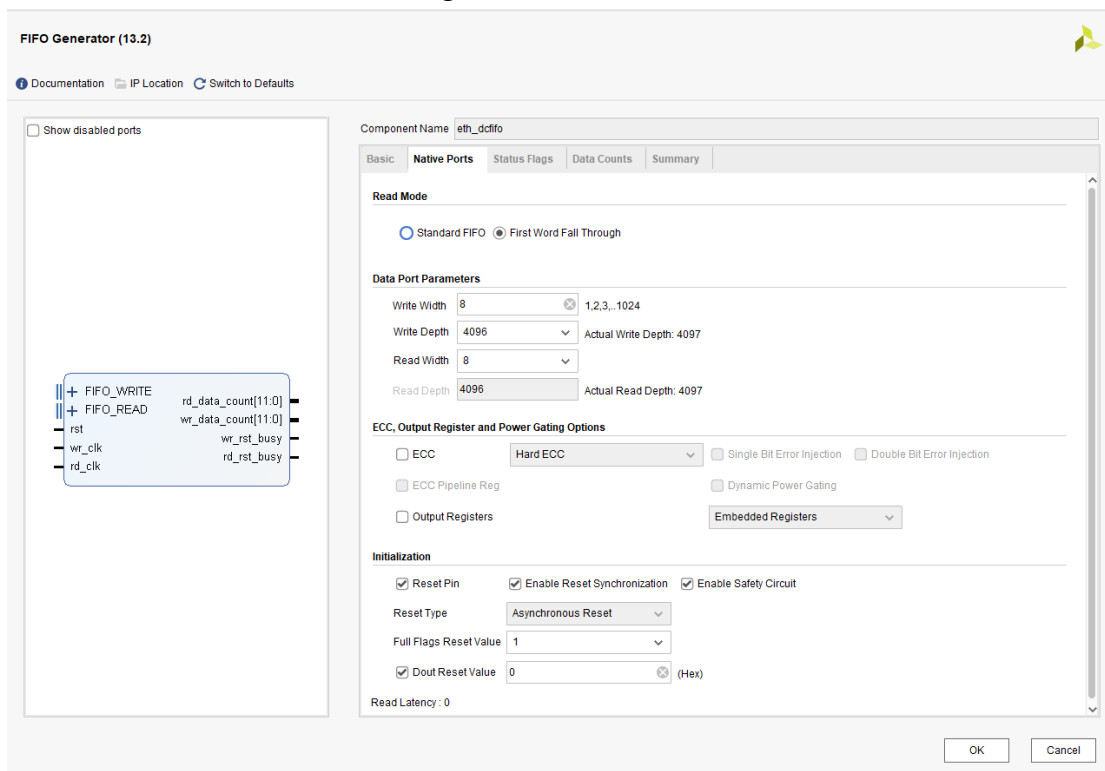


图 1-8 dcfifo 配置 2

Sataus Flags 配置保持默认。

店铺: <https://xiaomeige.taobao.com>

技术博客: <http://www.cnblogs.com/xiaomeige/>

官方网站: www.corecourse.cn

技术群组:

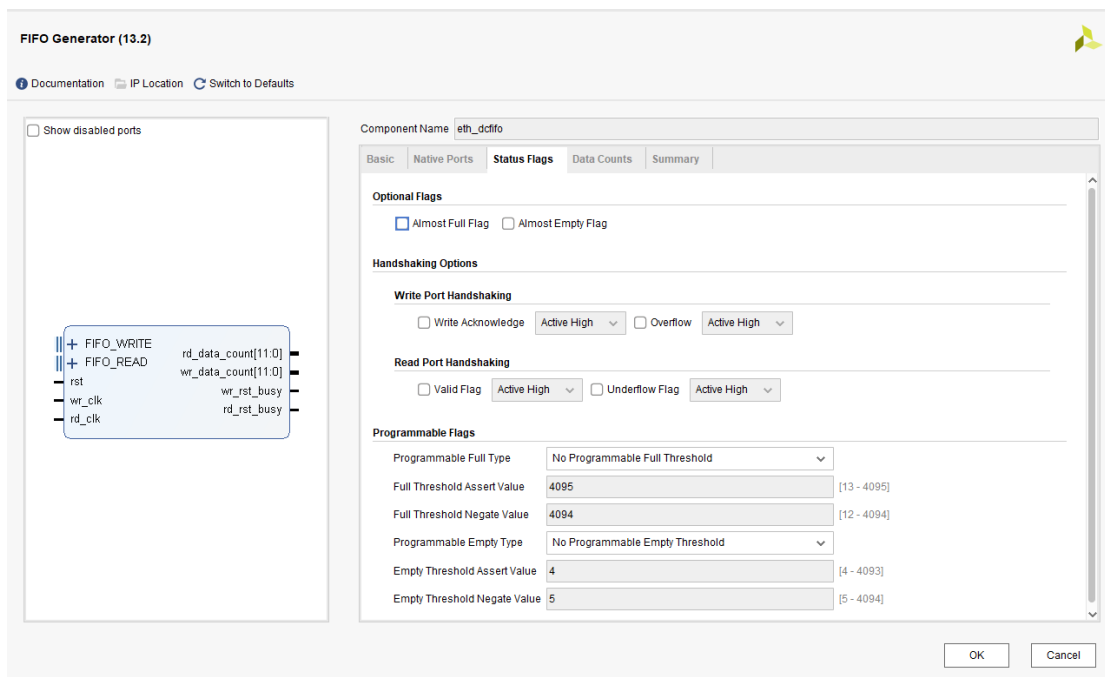


图 1-9 dcfifo 配置 3

读写数据统计。

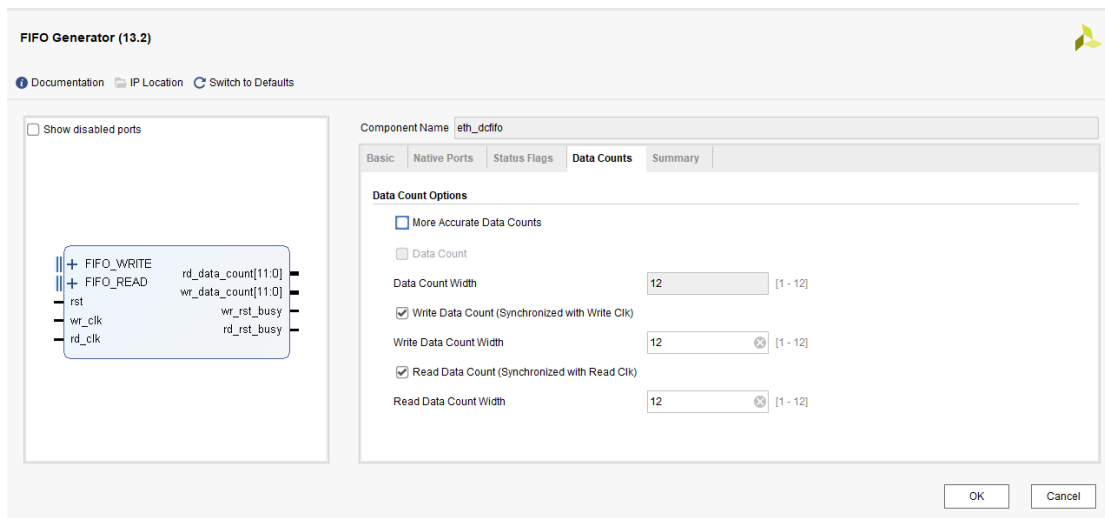


图 1-10 dcfifo 配置 4

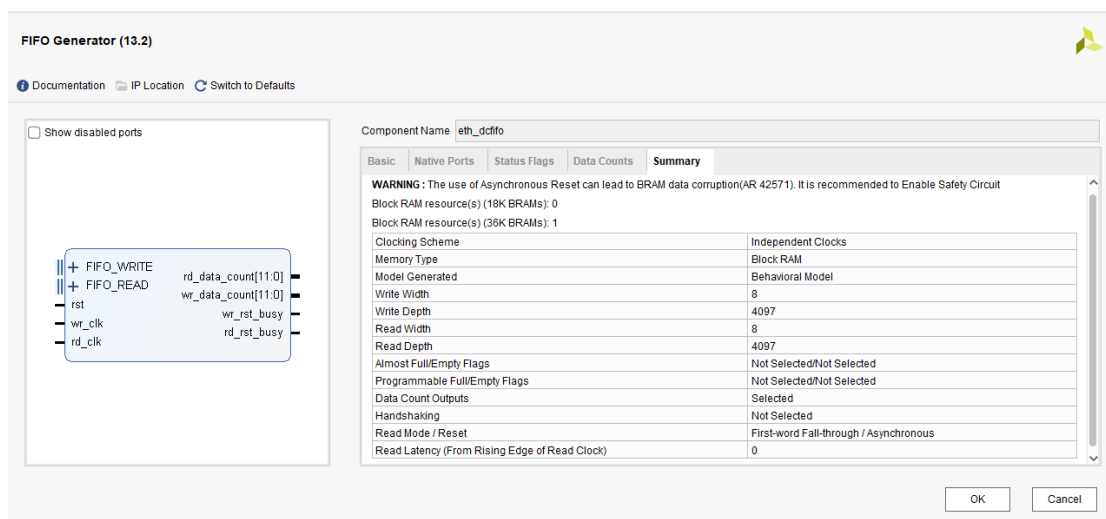


图 1-11 dcfifo 配置信息汇总页

前文已经提到过：FIFO 的配置除了位宽与深度外，大部分保持默认值。

1.6.2 UDP 发送模块设计（核心内容）

本次设计的 UDP 发送模块，由前面的章节实用型 UDP 收发逻辑设计与实现的理论框架设计而来。该模块将 eth_dcfifo 缓存，以及 CRC32_D8 校验模块例化其中，并添加设计一些发送控制信号。

我们在前面讲解 UDP 协议原理与 FPGA 实现以及实用型 UDP 收发逻辑的理论知识的时候，也专门讲解了校验文件的两种设计方法，我们也就不再进一步对校验文件进行分析了，而是直接将其例化引用。

这里，我们给设计文件添加一个顶层 UDP_Send 模块，在顶层中同时例化刚刚讲解的 fifo 以及 CRC32_D8 两个模块。

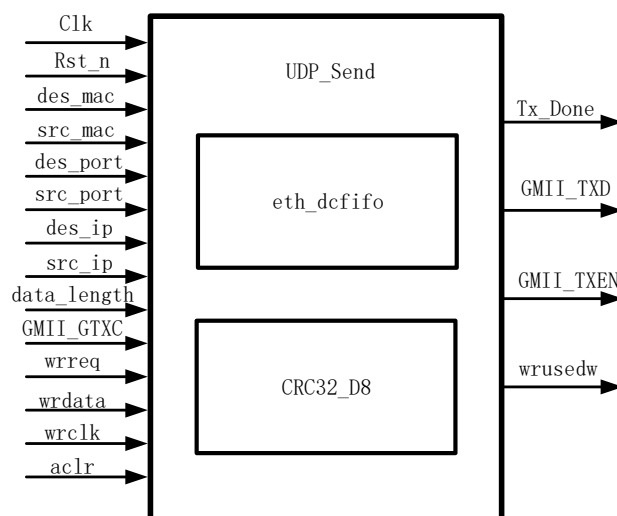


图 1-12 UDP_Send 模块框图

表 1-2 UDP_Send 模块接口功能描述

接口名称	I/O	位宽	功能描述
Clk	I	1	模块工作时钟
Rst_n	I	1	模块复位
des_mac	I	48	目标 mac 地址，相对于本工程 FPGA 而言，是 PC 机地址
src_mac	I	48	源 mac 地址，相对于本工程而言，是 FPGA 地址
des_port	I	16	目标端口号
src_port	I	16	源端口号
des_ip	I	32	目标 IP 地址
src_ip	I	32	源 IP 地址
data_length	I	16	数据长度
GMII_GTXC	I	1	GMII 发送时钟
wrreq	I	1	写请求信号
wrdata	I	8	写数据
wrclk	I	1	fifo 写时钟
aclr	I	1	fifo 复位信号
Tx_Done	O	1	发送完成信号
GMII_TXD	O	8	GMII 端口形式的发送数据
GMII_TXEN	O	1	GMII 发送使能
wrusedw	O	12	写数据计数器

下面给出 UDP 发送层的模块设计。受篇幅所限，我们将代码中描述本模块输入输出接口的代码篇幅省略。特定需要参考的同学，既可以拿本章节配套源码进行学习，又可以根据刚刚给出的端口列表进行复现。

在模块开始，给出本次发送的“长度/类型”字段。

第 1 部分：模块头部

```
`timescale 1 ns/ 1 ns
module UDP_Send(
.....
);
.....
localparam FRAME_TYPE = 16'h0800;
```

对于本帧发送的一些关键信息，在本帧数据发送前作出缓存，便于后续操作。

第 2 部分：将所有输入信号储存进入寄存器

```
always@(posedge GMII_GTXC)
if(Go)begin
    des_mac_reg <= #1 des_mac;
    src_mac_reg <= #1 src_mac;
    des_port_reg <= #1 des_port;
    src_port_reg <= #1 src_port;
    des_ip_reg <= #1 des_ip;
    src_ip_reg <= #1 src_ip;
```

```
data_length_reg <= #1 data_length;
end
```

输出端口添加一级寄存器，作为输出缓存。

第 2 部分：对输出端口加一级寄存器，方便 IO 输出寄存器

```
reg [7:0]GMII_TXD_reg;
reg GMII_TXEN_reg;

always@(posedge GMII_GTXC)begin
    GMII_TXD <= #1 GMII_TXD_reg;
    GMII_TXEN <= #1 GMII_TXEN_reg;
end
```

将上一小节讲解的输出缓存 fifo，例化到本小节内容中。

第 3 部分：例化以太网输出缓存 fifo

```
wire [7:0]fifo_rddata;
reg fifo_rdreq;

wire [11:0]rdusedw;
eth_dcfifo eth_dcfifo (
    .rst(ac1r),                // input wire rst
    .wr_clk(wrclk),           // input wire wr_clk
    .rd_clk(GMII_GTXC),       // input wire rd_clk
    .din(wrdata),             // input wire [7 : 0] din
    .wr_en(wrreq),            // input wire wr_en
    .rd_en(fifo_rdreq),       // input wire rd_en
    .dout(fifo_rddata),       // output wire [7 : 0] dout
    .full( ),                 // output wire full
    .empty( ),                // output wire empty
    .rd_data_count(rdusedw),  // output wire [11 : 0] rd_data_count
    .wr_data_count(wrusedw),  // output wire [11 : 0] wr_data_count
    .wr_rst_busy( ),          // output wire wr_rst_busy
    .rd_rst_busy( )           // output wire rd_rst_busy
);
```

接下来，描述出一个专门针对 GO 信号的辅助状态机。由于 GO 信号既有可能在本工程中通过数据和状态的变化而获得拉高条件，又有可能在别的工程中，和别的模块进行数据交互而获得拉高条件，这样，我们通过单独设计 GO 信号的状态机变化，将其从主状态机中剥离，保持主状态机设计的独立性和完整性，避免 GO 信号在切换来源时，修改状态机主体。

在状态 0，如果从 fifo 中读出的数据大于等于数据长度（本例中对应图像传输的行像素数），则进入状态 1 开始发送。状态机通过判断 Tx_done 是否拉高，判断发送是否完成，如果发送完成，则进入等待延时状态 2，完成等待延时状态 2 后，进入过渡状态 3，随后跳转回归到状态 0。

第 4 部分：UDP 发送控制状态机以及延时

```
reg Go;
reg [1:0]ctrl_state;
reg [7:0]delay_cnt;

always@(posedge GMII_GTXC or negedge Rst_n)
if(!Rst_n)begin
    ctrl_state <= #1 0;
    Go <= #1 1'b0;
    delay_cnt <= #1 0;
end
else begin
    case(ctrl_state)
        0:
            if(rdusedw >= data_length)begin
                ctrl_state <= #1 2'd1;
                Go <= #1 1'b1;
            end
            else begin
                ctrl_state <= 0;
                Go <= #1 1'b0;
            end

        1:
            begin
                Go <= #1 1'b0;
                if(Tx_Done)
                    ctrl_state <= #1 2'd2;
                else
                    ctrl_state <= #1 2'd1;
            end

        2:
            if(delay_cnt == 8'd255)begin
                ctrl_state <= #1 2'd3;
                delay_cnt <= #1 0;
            end
            else begin
                ctrl_state <= #1 2'd2;
                delay_cnt <= #1 delay_cnt + 1'b1;
            end

        3: ctrl_state <= #1 2'd0;
    endcase
end
```

接下来，可以进行 IP 层的数据报报头排列。

第 5 部分：IP 数据报发送设计

```
reg [31:0]ip_header[4:0];
reg [31:0]des_ip_reg;
reg [31:0]src_ip_reg;
reg [15:0]data_length_reg;
wire [15:0]ip_checksum;

always@(posedge GMII_GTXC)
begin
    ip_header[0][31:24] <= #1 8'h45;    //协议版本+首部长度
    ip_header[0][23:16] <= #1 8'h00;    //服务类型
    //IP 数据报总长度 (IP 报头+数据)
    ip_header[0][15:0] <= #1 data_length_reg + 8'd28;
    ip_header[1][31:0] <= #1 32'd0; //数据包标识 + 标识+分段偏移
    ip_header[2][31:24] <= #1 8'd64;    //生存时间 64
    ip_header[2][23:16] <= #1 8'd17;    //UDP 协议
    ip_header[2][15:0] <= #1 ip_checksum; //IP 校验和
    ip_header[3][31:0] <= #1 src_ip_reg; //源 IP 地址
    ip_header[4][31:0] <= #1 des_ip_reg; //目的 IP 地址
end
```

完成 IP 层的数据报报头设计后，接下来完成 UDP 层数据报的报头设计。

第 6 部分：UDP 报头设计

```
reg [31:0]udp_header[1:0];
reg [15:0]des_port_reg;
reg [15:0]src_port_reg;

always@(posedge GMII_GTXC)
begin
    udp_header[0][31:16] <= #1 src_port_reg;    //源端口号
    udp_header[0][15:0] <= #1 des_port_reg;    //目的端口号
    //UDP 数据报总长度 (UDP 报头+数据)
    udp_header[1][31:16] <= #1 data_length_reg + 8'd8;
    udp_header[1][15:0] <= #1 16'h00;    //UDP 报头校验和 忽略
end
```

接下来就是 IP 数据报的各字段求和，例化校验模块。

第 7 部分：IP 数据报求和及校验模块例化

```
wire [31:0]suma,sumb;
assign suma = ip_header[0][31:16] + ip_header[0][15:0]
              + ip_header[1][31:16] + ip_header[1][15:0]
              + ip_header[2][31:16] + ip_header[3][31:16]
              + ip_header[3][15:0] + ip_header[4][31:16]
              + ip_header[4][15:0];

assign sumb = (suma[31:16] + suma[15:0]);
```

```

assign ip_checksum = sumb[31:16]? ~(sumb[31:16] +
sumb[15:0]):~sumb;

wire CRC_Reset;
reg CRC_EN;

assign CRC_Reset = (state == IDLE);

CRC32_D8 CRC32_D8(
    .Clk(GMII_GTXC),
    .Reset(CRC_Reset),
    .Data_in(GMII_TXD_reg),
    .Enable(CRC_EN),
    .Crc(),
    .CrcNext(),
    .Crc_eth(crc_result)
);

```

在下一步，就是描述 MAC、IP、UDP 层层打包的主状态机。状态机的每一个状态，依次就是一个数据帧发送的每一个数据字段。为了讲解清晰，各位读者可以把状态机和 MAC 层数据组包图进行对比分析。

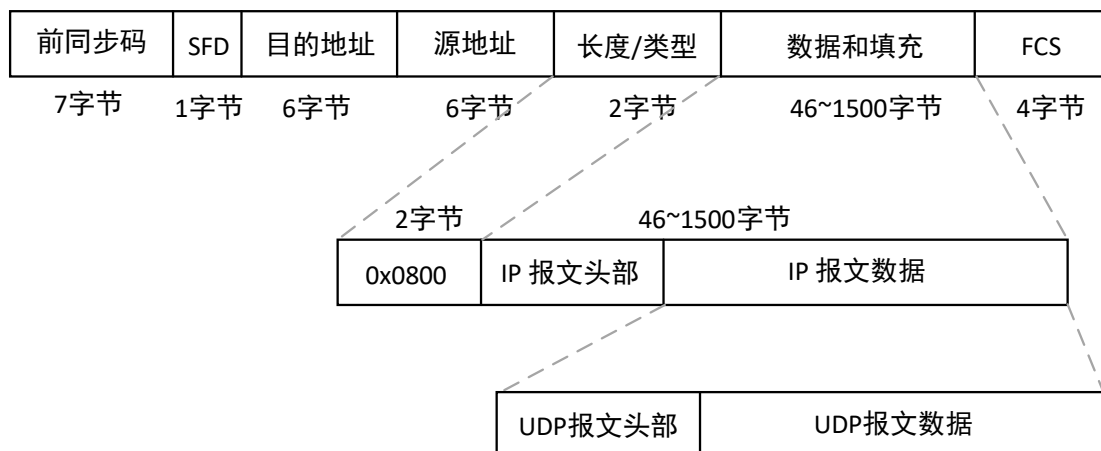


图 1-13 MAC 层打包示意图回顾

第 8 部分：MAC 层发送状态机（关于层层打包的状态机描述）

```

reg [47:0]des_mac_reg;
reg [47:0]src_mac_reg;

reg [4:0]cnt_header;    //前导码计数器
reg [4:0]cnt_des_mac;   //目标 MAC 地址计数器
reg [4:0]cnt_src_mac;   //源 MAC 地址计数器
reg [1:0]cnt_frame_type; //帧类型计数器

reg [4:0]cnt_ip_header; //IP 报头计数器

```

```
reg [4:0]cnt_udp_header;    //UDP 报头计数器
reg [11:0]cnt_data; //数据计数器
reg [4:0]cnt_crc;    //crc 计数器

wire [31:0]crc_result;
reg [9:0]state;

localparam
    IDLE =          10'b0000000001,
    SEND_HEADER =    10'b0000000010,
    SEND_DES_MAC =   10'b0000000100,
    SEND_SRC_MAC =   10'b0000001000,
    SEND_FRAME_TYPE = 10'b0000010000,
    SEND_IP_HEADER =  10'b0000100000,
    SEND_UDP_HEADER = 10'b0001000000,
    SEND_DATA =       10'b0010000000,
    SEND_CRC =        10'b0100000000;

always@(posedge GMII_GTXC or negedge Rst_n)
if(!Rst_n)begin
    state <= #1 IDLE;
    cnt_header <= #1 0; //前导码计数器
    cnt_des_mac <= #1 0; //目标 MAC 地址计数器
    cnt_src_mac <= #1 0; //源 MAC 地址计数器
    cnt_frame_type <= #1 0; //帧类型计数器
    cnt_ip_header <= #1 0; //IP 报头计数器
    cnt_udp_header <= #1 0; //UDP 报头计数器
    cnt_data <= #1 0; //数据计数器
    cnt_crc <= #1 0; //crc 计数器
    GMII_TXD_reg <= #1 0;
    GMII_TXEN_reg <= #1 0;
    fifo_rdreq <= #1 1'b0;
    Tx_Done <= #1 1'b0;
    CRC_EN <= #1 1'b0;
end
else begin
    case(state)
        IDLE:
            begin
                GMII_TXEN_reg <= #1 0;
                Tx_Done <= #1 1'b0;
                if(Go)
                    state <= #1 SEND_HEADER; // 发送前导码
                else
                    state <= #1 IDLE;
            end
    endcase
end
```



```
SEND_HEADER:
begin
    GMII_TXEN_reg <= #1 1;
    if(cnt_header >= 7)begin
        cnt_header <= #1 0;
        state <= #1 SEND_DES_MAC;
    end
    else begin
        cnt_header <= #1 cnt_header + 1'b1;
        state <= #1 SEND_HEADER;
    end
    case(cnt_header)
        0,1,2,3,4,5,6:GMII_TXD_reg <= #1 8'h55;
        7:GMII_TXD_reg <= #1 8'hd5;
        default:GMII_TXD_reg <= #1 8'h55;
    endcase
end

SEND_DES_MAC:
begin
    CRC_EN <= #1 1'b1;
    if(cnt_des_mac >= 5)begin
        cnt_des_mac <= #1 0;
        state <= #1 SEND_SRC_MAC;
    end
    else begin
        cnt_des_mac <= #1 cnt_des_mac + 1'b1;
        state <= #1 SEND_DES_MAC;
    end
    case(cnt_des_mac)
        0:GMII_TXD_reg <= #1 des_mac_reg[47:40];
        1:GMII_TXD_reg <= #1 des_mac_reg[39:32];
        2:GMII_TXD_reg <= #1 des_mac_reg[31:24];
        3:GMII_TXD_reg <= #1 des_mac_reg[23:16];
        4:GMII_TXD_reg <= #1 des_mac_reg[15:8];
        5:GMII_TXD_reg <= #1 des_mac_reg[7:0];
        default:GMII_TXD_reg <= #1 8'hff;
    endcase
end

SEND_SRC_MAC:
begin
    if(cnt_src_mac >= 5)begin
        cnt_src_mac <= #1 0;
        state <= #1 SEND_FRAME_TYPE;
    end
    else begin
```

```
        cnt_src_mac <= #1 cnt_src_mac + 1'b1;
        state <= #1 SEND_SRC_MAC;
    end
    case(cnt_src_mac)
        0:GMII_TXD_reg <= #1 src_mac_reg[47:40];
        1:GMII_TXD_reg <= #1 src_mac_reg[39:32];
        2:GMII_TXD_reg <= #1 src_mac_reg[31:24];
        3:GMII_TXD_reg <= #1 src_mac_reg[23:16];
        4:GMII_TXD_reg <= #1 src_mac_reg[15:8];
        5:GMII_TXD_reg <= #1 src_mac_reg[7:0];
        default:GMII_TXD_reg <= #1 8'hff;
    endcase
end

SEND_FRAME_TYPE:
begin
    if(cnt_frame_type >= 1)begin
        cnt_frame_type <= #1 0;
        state <= #1 SEND_IP_HEADER;
    end
    else begin
        cnt_frame_type <= #1 cnt_frame_type + 1'b1;
        state <= #1 SEND_FRAME_TYPE;
    end
    case(cnt_frame_type)
        0:GMII_TXD_reg <= #1 FRAME_TYPE[15:8];
        1:GMII_TXD_reg <= #1 FRAME_TYPE[7:0];
        default:GMII_TXD_reg <= #1 8'hff;
    endcase
end

SEND_IP_HEADER:
begin
    if(cnt_ip_header >= 19)begin
        cnt_ip_header <= #1 0;
        state <= #1 SEND_UDP_HEADER;
    end
    else begin
        cnt_ip_header <= #1 cnt_ip_header + 1'b1;
        state <= #1 SEND_IP_HEADER;
    end
    case(cnt_ip_header)
        0 :GMII_TXD_reg <= #1 ip_header[0][31:24];
        1 :GMII_TXD_reg <= #1 ip_header[0][23:16];
        2 :GMII_TXD_reg <= #1 ip_header[0][15:8];
        3 :GMII_TXD_reg <= #1 ip_header[0][7:0];
        4 :GMII_TXD_reg <= #1 ip_header[1][31:24];
```

```
5 :GMII_TXD_reg <= #1 ip_header[1][23:16];
6 :GMII_TXD_reg <= #1 ip_header[1][15:8];
7 :GMII_TXD_reg <= #1 ip_header[1][7:0];
8 :GMII_TXD_reg <= #1 ip_header[2][31:24];
9 :GMII_TXD_reg <= #1 ip_header[2][23:16];
10:GMII_TXD_reg <= #1 ip_header[2][15:8];
11:GMII_TXD_reg <= #1 ip_header[2][7:0];
12:GMII_TXD_reg <= #1 ip_header[3][31:24];
13:GMII_TXD_reg <= #1 ip_header[3][23:16];
14:GMII_TXD_reg <= #1 ip_header[3][15:8];
15:GMII_TXD_reg <= #1 ip_header[3][7:0];
16:GMII_TXD_reg <= #1 ip_header[4][31:24];
17:GMII_TXD_reg <= #1 ip_header[4][23:16];
18:GMII_TXD_reg <= #1 ip_header[4][15:8];
19:GMII_TXD_reg <= #1 ip_header[4][7:0];
default:GMII_TXD_reg <= #1 8'hff;
endcase
end

SEND_UDP_HEADER:
begin
if(cnt_udp_header >= 7)begin
cnt_udp_header <= #1 0;
fifo_rdreq <= #1 1'b1;
state <= #1 SEND_DATA;
end
else begin
cnt_udp_header <= #1 cnt_udp_header + 1'b1;
state <= #1 SEND_UDP_HEADER;
end
case(cnt_udp_header)
0 :GMII_TXD_reg <= #1 udp_header[0][31:24];
1 :GMII_TXD_reg <= #1 udp_header[0][23:16];
2 :GMII_TXD_reg <= #1 udp_header[0][15:8];
3 :GMII_TXD_reg <= #1 udp_header[0][7:0];
4 :GMII_TXD_reg <= #1 udp_header[1][31:24];
5 :GMII_TXD_reg <= #1 udp_header[1][23:16];
6 :GMII_TXD_reg <= #1 udp_header[1][15:8];
7 :GMII_TXD_reg <= #1
udp_header[1][7:0];
default:GMII_TXD_reg <= #1 8'hff;
endcase
end

SEND_DATA:
begin
if(cnt_data >= data_length_reg - 1)begin
```

```
        cnt_data <= #1 0;
        state <= #1 SEND_CRC;
        fifo_rdreq <= #1 1'b0;
        GMII_TXD_reg <= #1 fifo_rddata;
    end
    else begin
        cnt_data <= #1 cnt_data + 1'b1;
        state <= #1 SEND_DATA;
        GMII_TXD_reg <= #1 fifo_rddata;
    end
end

SEND_CRC:
begin
    CRC_EN <= #1 1'b0;
    if(cnt_crc >= 3)begin
        cnt_crc <= #1 0;
        state <= #1 IDLE;
        Tx_Done <= #1 1'b1;
    end
    else begin
        cnt_crc <= #1 cnt_crc + 1'b1;
        state <= #1 SEND_CRC;
    end
    case(cnt_crc)
        0 :GMII_TXD_reg <= #1 crc_result[31:24];
        1 :GMII_TXD_reg <= #1 crc_result[23:16];
        2 :GMII_TXD_reg <= #1 crc_result[15:8];
        3 :GMII_TXD_reg <= #1
crc_result[7:0];
        default:GMII_TXD_reg <= #1 8'hff;
    endcase
end
default:begin state <= #1 IDLE;end
endcase
end
```

到这里，担任 UDP 发送任务的核心模块，就分析完毕了。

1.7 GMII 转 RGMII 模块调用例化

我们在前面的章节中，已经讲解了 GMII 转 RGMII 和 RGMII 转 GMII 的方法。在本节中，由于我们需要将 UDP_Send 模块生成的 GMII 发送信号转换为本工程制定的 RGMII 发送接口信号。所以，我们在工程顶层，直接例化 GMII 转 RGMII 模块即可。

```
gmii_to_rgmii gmii_to_rgmii(  
    .reset_n(rst_n),  
    .gmii_tx_clk(GMII_GTXC),  
    .gmii_txd(GMII_TXD),  
    .gmii_txen(GMII_TXEN),  
    .gmii_txer(0),  
    .rgmii_tx_clk(eth_gtxc),  
    .rgmii_txd(eth_txd),  
    .rgmii_txen(eth_txen)  
);
```

这样，整个工程的数据发送链路，就设计完成了。

1.8 其他涉及模块说明

由于本章节的讲授重点放在以太网发送部分的设计之上，所以本章中涉及到的锁相环和摄像头初始化部分的内容，就不作过多分析讲解。实际工程应用时，只需将其按工作进行配置，并在顶层例化使用即可。

1.9 板级验证

由于 ACZ702 开发板本身自带有摄像头接口，千兆网口接口，满足本次系统设计的需求。因此，这里可以用 ACZ702 开发板来进行本项目的板级验证。

1.9.1 系统所需硬件：

- 1、ACZ702 开发板
- 2、电源线一根（可选）
- 3、OV5640 摄像头一个
- 4、type-c 线一根
- 5、千兆网线一根

1.9.2 硬件连接

本次系统设计硬件方面连接如图 1-14 所示：

- 1、将摄像头连接到摄像头接口上
- 2、用网线将开发板网口与电脑网口连接

- 3、将 type-c 线一端连接开发板 Jtag 口，一端连接电脑 USB 接口
- 4、连接好电源，电源线连接电源适配器为开发板供电（可选）

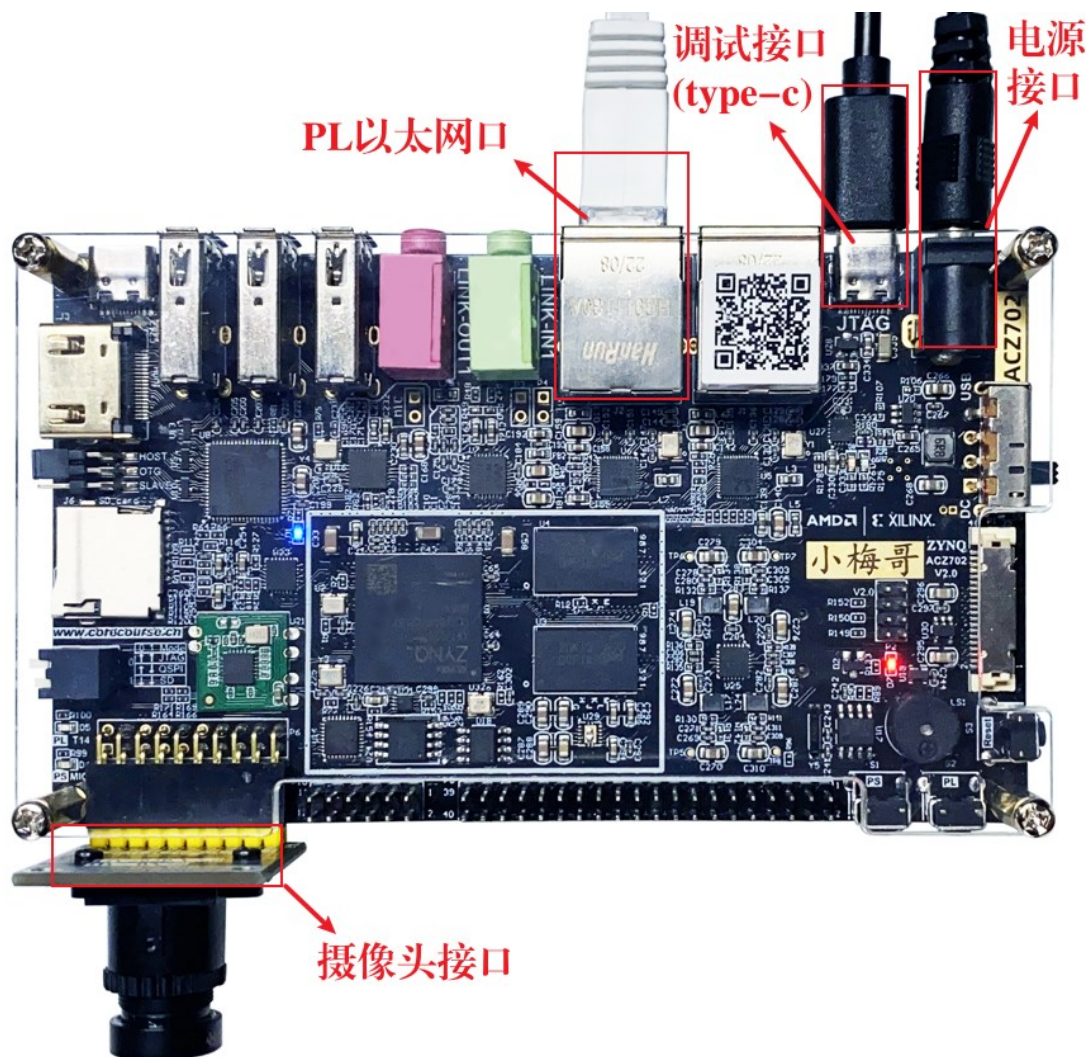


图 1-14 硬件连接图

在连接网线时，一端应当连接 FPGA 的网口，另一端则与电脑的网口相连，如果 FPGA 端网口的指示灯一边长亮一边闪烁，则代表连接成功。

连接完毕后接下来即可进行管脚绑定了。

1.9.3 管脚绑定

对工程进行分析和综合，确认设计无误后，为设计行管脚分配并约束电平。本次设计引脚分配表如表 1-3 所示：

表 1-3 引脚分配图

Pin Name	Signal Name	Pin NO.	Pin Name	Signal Name	Pin NO.
----------	-------------	---------	----------	-------------	---------

店铺：<https://xiaomeige.taobao.com>

技术博客：<http://www.cnblogs.com/xiaomeige/>

官方网站：www.corecourse.cn

技术群组：

CMOS_D7	camera_data[7]	L16		CMOS_XCLK	camera_xclk	A20
CMOS_D6	camera_data[6]	L17		PL_ENET0_TX_DATA3	eth_txd[3]	U20
CMOS_D5	camera_data[5]	K17		PL_ENET0_TX_DATA2	eth_txd[2]	T20
CMOS_D4	camera_data[4]	M15		PL_ENET0_TX_DATA1	eth_txd[1]	T19
CMOS_D3	camera_data[3]	L14		PL_ENET0_TX_DATA0	eth_txd[0]	R18
CMOS_D2	camera_data[2]	N15		PL_ENET0_GTX_CLK	eth_gtxc	P16
CMOS_D1	camera_data[1]	M14		PL_ENET0_MDC	eth_mdc	C20
CMOS_D0	camera_data[0]	J15		PL_ENET0_MDIO	eth_mdio	B20
CMOS_PCLK	camera_pclk	M17		PL_ENET0_RESET	eth_rst_n	B19
CMOS_HREF	camera_href	U13		PL_ENET0_TX_EN	eth_txen	T17
CMOS_SCLK	camera_sclk	N17		FPGA_GCLK1	clk	U18
CMOS_SDAT	camera_sdat	P18		FPGA_KEY0	rst_n	F20
CMOS_VS	camera_vsync	N16				

分配完管脚后还需为这些管脚设置电平标准为 LVCMOS33*, 完成后如图 1-15 所示:

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	IO Std	Vccio	Vref	Drive Strength	Slew Type	Pull Type	Off-Chip Termination	IN_TERM
write_clk_642 (1)	IN			✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_pclk	IN		M17	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[7]	IN		L16	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[6]	IN		L17	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[5]	IN		K17	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[4]	IN		M15	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[3]	IN		L14	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[2]	IN		N15	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[1]	IN		M14	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_data[0]	IN		J15	✓	35	LVCMOS33*	3.300				NONE	NONE	
eth_txd[3]	OUT		U20	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_txd[2]	OUT		T20	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_txd[1]	OUT		T19	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_txd[0]	OUT		R18	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
camera_href	IN		U13	✓	34	LVCMOS33*	3.300				NONE	NONE	
camera_sclk	OUT		N17	✓	34	LVCMOS33*	3.300	12	12	SLOW	PULLUP	FP_VTT_50	
camera_sdat	INOUT		P18	✓	34	LVCMOS33*	3.300	12	12	SLOW	PULLUP	FP_VTT_50	
camera_vsync	IN		N16	✓	35	LVCMOS33*	3.300				NONE	NONE	
camera_xclk	OUT		A20	✓	35	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
clk	IN		U18	✓	34	LVCMOS33*	3.300				NONE	NONE	
eth_gtxc	OUT		P16	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_mdc	OUT		C20	✓	35	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_mdio	INOUT		B20	✓	35	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_rst_n	OUT		B19	✓	35	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
eth_txen	OUT		T17	✓	34	LVCMOS33*	3.300	12	12	SLOW	NONE	FP_VTT_50	
rst_n	IN		F20	✓	35	LVCMOS33*	3.300				NONE	NONE	

图 1-15 管脚分配与电平约束

完成以上操作后保存设计, 接下来我们还需要为设计添加约束语句。对于设计中, 像 camera_pclk 这类时钟属性的引脚, 如果直接使用普通 IO 管脚, 将会影响该时钟的整体质量。因此, 我们可以通过约束语句, 将其连接到全局时钟树上, 具体的约束语句如下:

```
create_clock -period 13.888 -name cam_pclk [get_ports camera_pclk]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets camera_pclk_IBUF]
```

添加完语句后便完成了对设计的约束, 接下来就可以生成 bit, 准备进行板级验证环节了。

1.9.4 下载与验证

本次板级验证的工程名为 ov5640_rgmii_udp，连接好硬件后对应的下载与验证操作如下：

- 1、将开发板左侧的电源拨码开关拨到 ON 侧，将开发板右侧的跳线帽插接拨到 RGMII 档。
- 2、将工程生成的 bit 文件下载至开发板中，注意这一步一定要先于下面的步骤执行。
- 3、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，如图 1-16 网络连接状态。需要看到在活动网络中有本地连接存在，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。



图 1-16 网络连接状态

- 4、点击“本地连接”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 1-17 以太网速率



图 1-17 以太网速率

- 5、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，然后再弹出的属性对话框中设置静态 IP 地址。如图 1-18 所示。

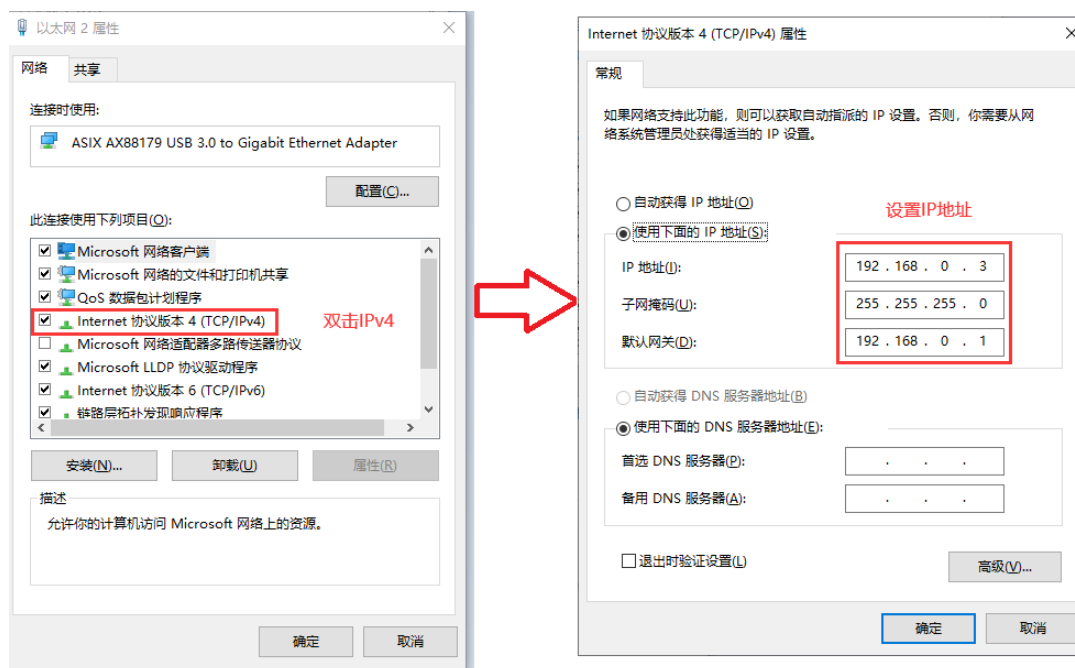


图 1-18 设置静态 IP 地址

6、开启电脑巨型帧。在“设备管理器\网络适配器\Realtek PCIe GbE Family Controller 属性\高级\巨型帧”，如图 1-19 所示

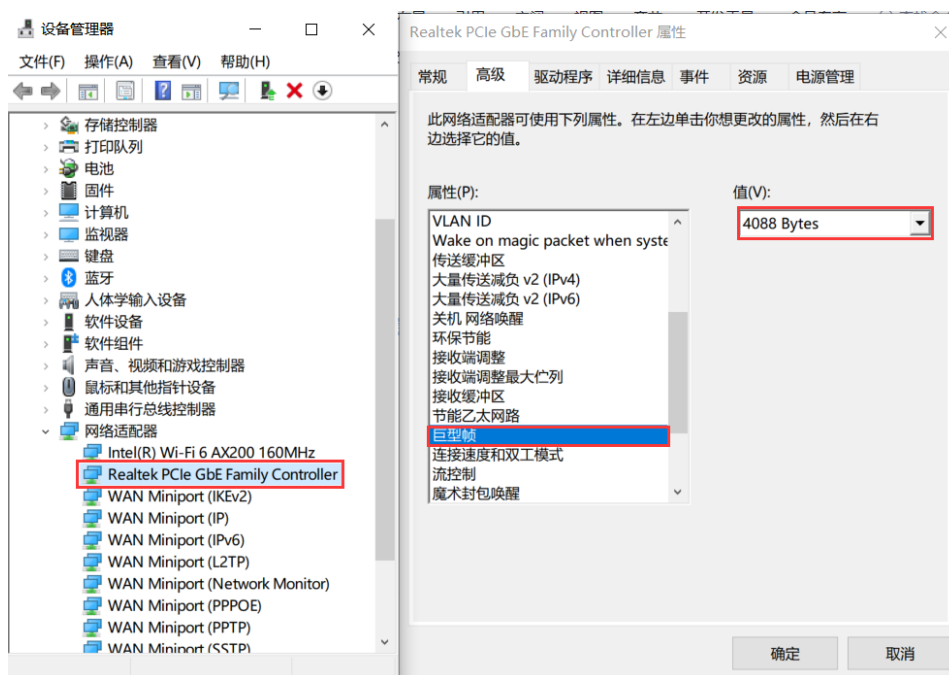


图 1-19 开启巨型帧

7、打开小梅哥 UDP 摄像头 V3.2 并按照如下所述设置各项参数。



图 1-20 软件设置

■ 本机 IP，对应当前实验所用电脑的 IP 地址，前面已经按照要求设置为了 192.168.0.3，所以这里直接填写该地址。

- 本机端口，对应当前实验所用电脑接收图像时使用的网络端口，该端口在 FPGA 程序中定义为了 6102，所以这里也要设置为 6102 才能正确显示图像。
 - 远程 IP，对应 FPGA 开发板使用的 IP 地址，也就是代码中的 `src_ip` 值。例程使用的为 192.168.0.2。
 - 远程端口，对应 FPGA 开发板使用的端口号，也就是代码中的 `src_port` 值，例程使用的为 5000。
 - 分辨率，对应了摄像头输出的图像分辨率大小，本实验中，千兆 GMII 接口使用的分辨率为 1280*720。
 - 保存图片，该按钮在软件停止接受图像后，点击可以保存最后界面上显示的图像内容到文件。
 - 连接/停止，该按钮可以开始和停止接受并显示图像，当所有参数设置好之后，点击该按钮即可开始接收并显示图像。
- 8、设置好参数后，点击链接按钮即可开始接收并显示图像内容
- 9、该软件运行时需要关闭防火墙，软件自带关闭防火墙功能，点击连接时防火墙会提示，勾选专用网络和公共网络两个选项，然后点击【允许访问按钮】即可。如果还是无法关闭，考虑软件权限不够，以管理员身份运行本软件即可，如图 1-21 所示。

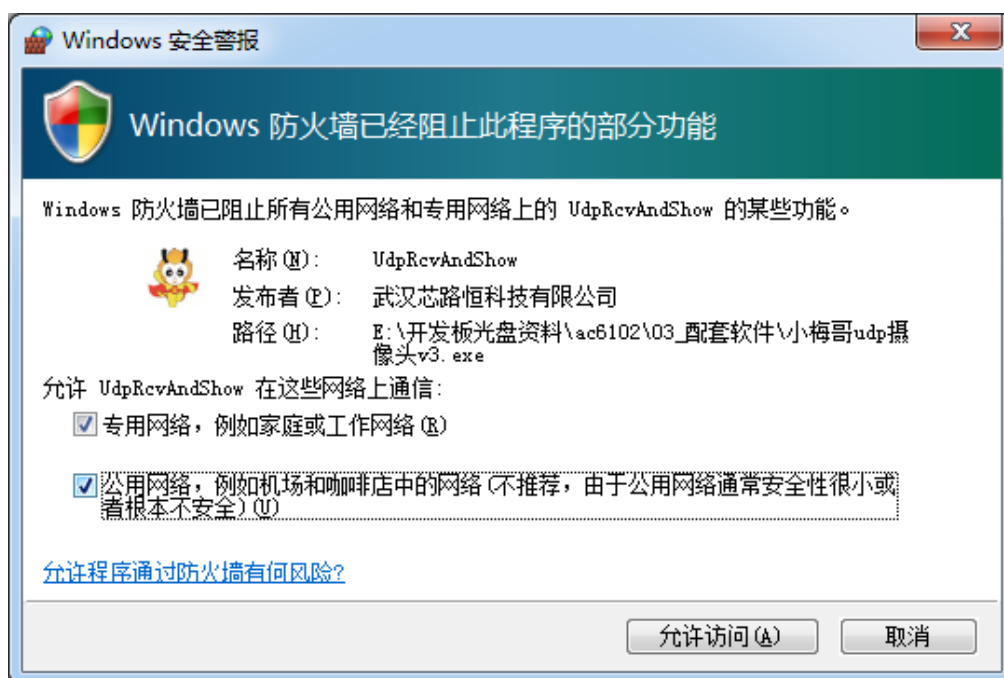


图 1-21

连接成功后，可以看到软件显示图像如图 1-22 所示

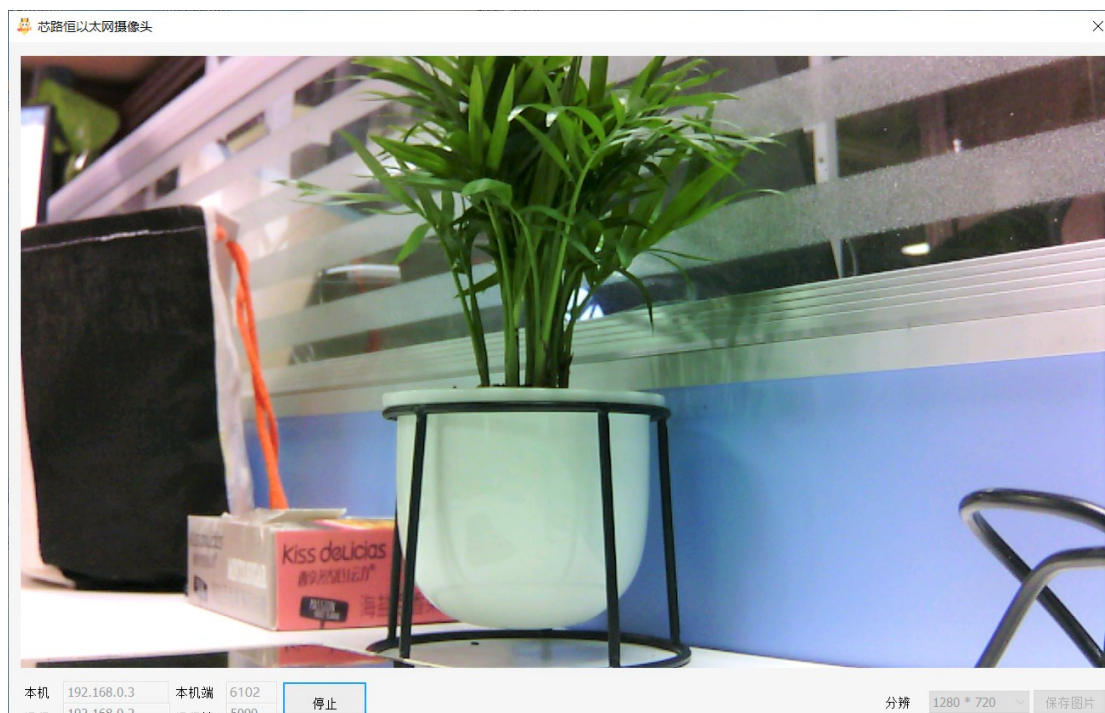


图 1-22 采集显示图像

可以看到显示出来的画面色彩鲜艳亮丽，图像整体饱和度高，图像比例正常，没有任何压缩错位现象，本次以太网图像编码模块的设计成功。

1.10总结与思考

本节设计实现了对摄像头采集的数据以行为单位进行编码排序，以防止数据在通过以太网进行传输显示时由于数据丢失而导致的图像压缩和错位现象，学习本节时读者可以参考相应的视频教程相互理解印证，建议读者能够跟随本实验内容，完整的进行整个实验。