小梅哥 AC620 FPGA 开发板进阶设计教程

全功能高性价比 FPGA 开发板 AC620 火热销售中，配套 800 页原创电子书。

购买链接：

https://item.taobao.com/item.htm?spm=a1z10.1-c-s.w4004-13687301132.6.8uPVJ6&id=544830995588

目录

AC620 FPGA 以太网电路介绍 4
第 1 章 基于 RTL8201 的以太网 UDP 通信测试 5
第 2 章 以太网 MAC 层基本原理 13
MII 接口介绍： 13
以太网（MAC）帧介绍 14
第 3 章 ARP 协议的原理与 FPGA 实现 17
使用以太网发包工具组 ARP 包 19
使用以太网发包工具发包 20
使用 wireshark 软件抓包 21
使用 CRC 计算软件计算 CRC 校验值 21
使用 FPGA signaltap 抓取 MII 接口接收信号 22
CRC 计算软件结果与以太网帧 CRC 值关系 22
第 4 章 以太网 ARP 帧发包实例（手动 CRC） 23
第 5 章 以太网 ARP 帧发包实例（自动 CRC） 30
第 6 章 UDP 协议原理与 FPGA 实现 39
UDP 协议介绍 39
UDP 数据报格式 40
IP 协议介绍 42
第 7 章 以太网 UDP 帧发包实例（手动 IPchecksum） 50
第 8 章 以太网 UDP 帧发包实例（自动 IPchecksum） 63
第 9 章 基于 FIFO 接口的 UDP 数据包发送 76
第 10 章 以太网图像传输实验 76

声　　明

本文如有更新或修改，将不另行通知，如有疑问，可以邮件咨询小梅哥，或前往芯航线 FPGA 技术支持群(615381411)寻找最新版本。本文在发烧友论坛地址：

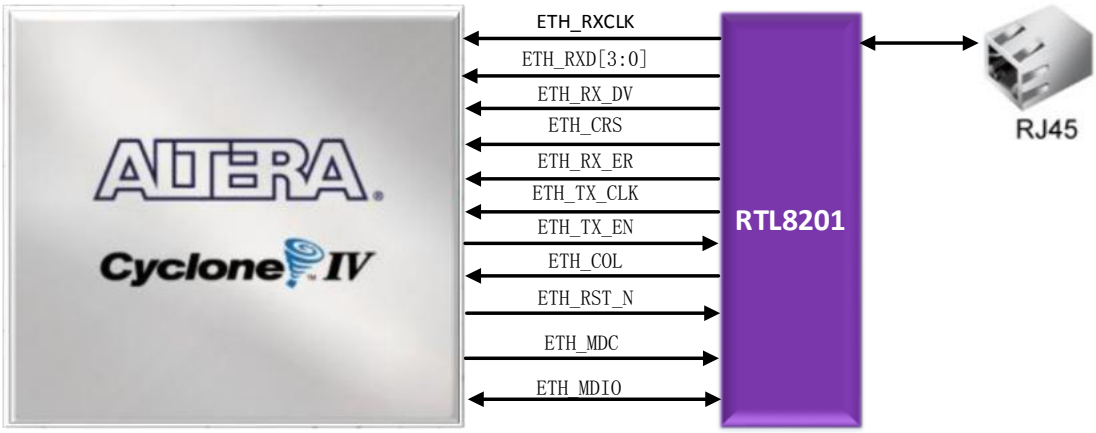
http://bbs.elecfans.com/jishu_1152310_1_1.html

如果用户发现本文有任何错误或者难以理解的地方，欢迎发送邮件给小梅哥沟通（xiaomeige_fpga@foxmail.com）

严禁一切未经作者授权的转载或商业使用，如用于培训班、开发板配套资料等。

AC620 FPGA 以太网电路介绍

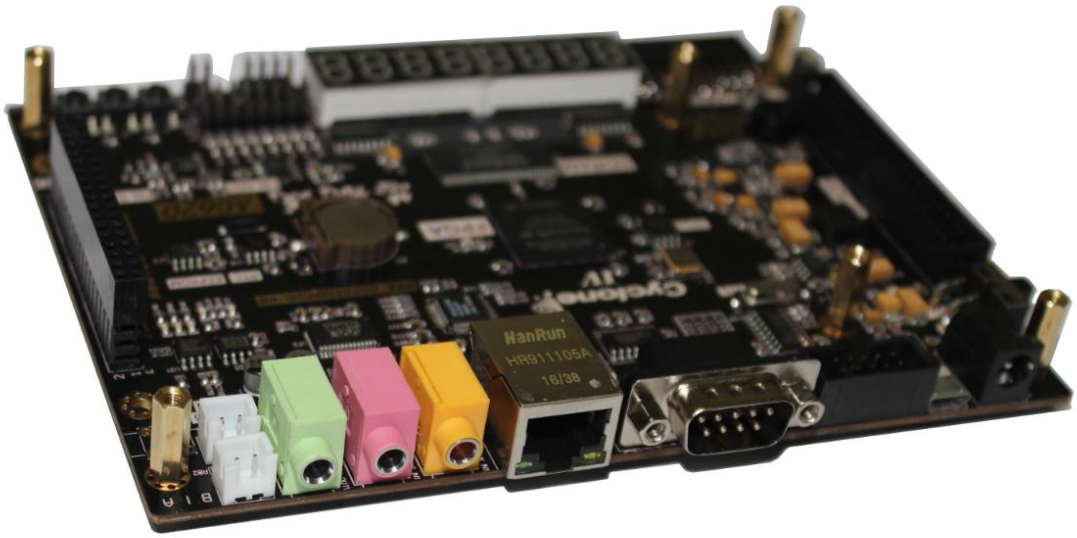
AC620 开发板通过一片 Realtek 的 RTL8201 以太网 PHY 提供对以太网连接的支持，RTL8201 是一片 10M/100M 自适应以太网收发器，提供 MII/SNI 接口的 MAC 连接。在 Cyclone IV E 器件中，调用三速以太网 IP 核（MAC），实现完整的以太网连接。或者用户使用 Verilog 编写的自定义用户逻辑来实现以太网连接。下图为 RTL8201 与 Cyclone IV E 的连接关系。



RTL8201 与 Cyclone IV E 的连接关系

以太网收发器管脚分配表

Signial Name	FPGA pin No.	Signial Name	FPGA pin No.
ETH0_TX_EN	PIN_C14	ETH0_RX_DV	PIN_A15
ETH0_TXCLK	PIN_D11	ETH0_RX_ER	PIN_F11
ETH0_TXD0	PIN_C11	ETH0_RXCLK	PIN_A13
ETH0_TXD1	PIN_B12	ETH0_RXD0	PIN_E10
ETH0_TXD2	PIN_A12	ETH0_RXD1	PIN_B14
ETH0_TXD3	PIN_B11	ETH0_RXD2	PIN_A14
ETH0_MDC	PIN_E11	ETH0_RXD3	PIN_B13
ETH0_MDIO	PIN_D12	ETH0_COL	PIN_A11
ETH0_RST_N	PIN_D14	ETH0_CRS	PIN_F10



第 1 章 基于 RTL8201 的以太网 UDP 通信测试

小梅哥编写，未经许可，文章内容不得用于其他商业销售的板卡

在芯航线 AC620 开发板上，设计了一路 MII 接口的百兆以太网电路，通过该以太网电路，用户可以将 FPGA 采集或运算得到的数据传递给其他设备如 PC 或服务器，或者接收其他设备传输过来的数据并进行处理。

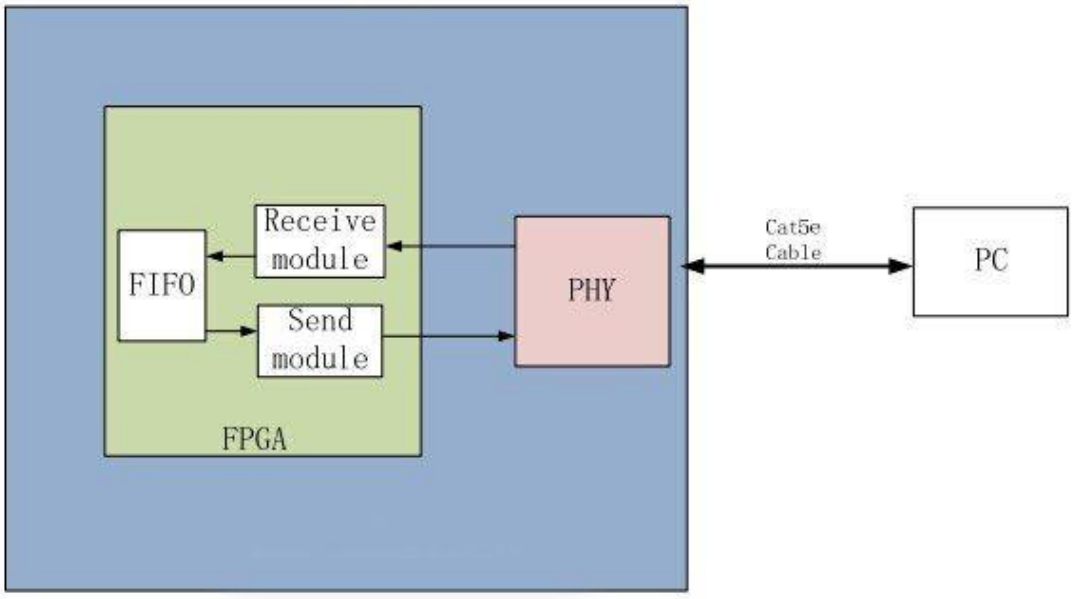
接触过以太网的用户，应该最常听说的是 TCP/IP 协议，确实，在 PC 端或者嵌入式系统中，TCP/IP 协议应用非常广泛，因此，当大家看到 FPGA 上带有以太网接口时，可能第一个想到的也是实现 TC/IP 协议。这里，首先可以很肯定的告诉大家，使用 FPGA 实现 TCP/IP 协议是完全没有问题的，但是，实现的方式却不是大家最期望的直接使用 Verilog 编写协议层代码来实现。FPGA 发展到现在，三十多年了，却鲜见有成功商用的 RTL 级的 TCP/IP 的设计，而大部分使用 Verilog 或者 VHDL 实现的以太网传输，都是基于非常简单的 UDP 协议的。当然，探索或者实现其中部分功能的人还是有的，只是，很难做到像 PC 那样灵活应用。

个人理解，TCP/IP 协议设计之初就是根据软件灵活性设计的，因此在很多设计考虑上，并不适合使用硬线逻辑实现。TCP/IP 协议非常的复杂，如果使用硬件逻辑实现，工程量必然十分浩大，而且功能和性能都无法得到保证。

那么怎样在 FPGA 上实现 TCP/IP 协议呢？答案就是 SOPC 技术。即使用嵌入式软核技术，在 FPGA 上搭建软核 CPU 系统，再通过 CPU 来运行软件 TCP/IP 协议，从而实现相应功能。但是这种实现方式对于很多用户来说，前期系统创建的过程比较繁琐，因此很多朋友都难以上手，因此这种方式使用的也并不是很广泛。

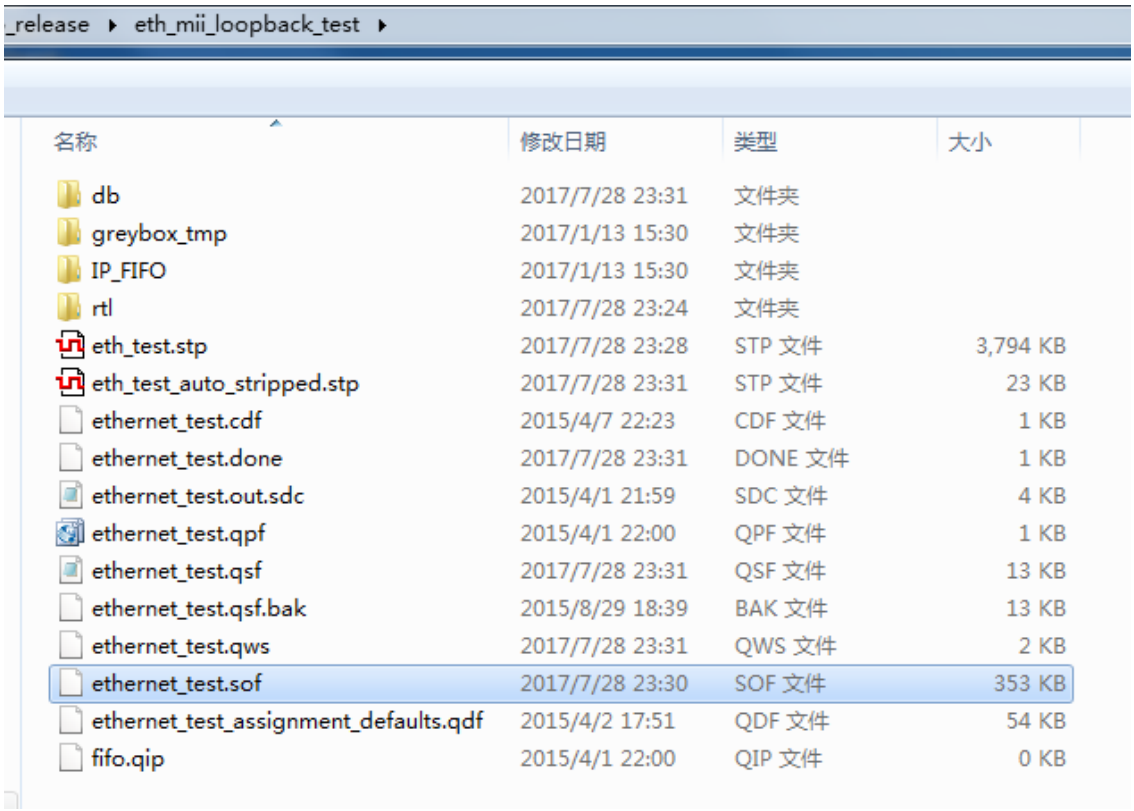
上面说过，在 FPGA 上，可以使用 Verilog 实现 UDP 协议来进行数据的传输。UDP 协议是一种不可靠传输，发送方只负责将数据发送出去，而不管接收方是否正确的接收。非常类似于 UART 串口传输。但是，在很多场合，是可以接受这种潜在的不可靠性的，例如视频实时传输显示。在这类系统中，由于数据并不需要进行运算并得到非常精确的结果用于其他功能，而仅仅是显示在屏幕上，因此可以接受一定程度的丢包或者误码。此类应用在 LED 大屏显示系统中应用非常广泛。本节就介绍提供给大家的一个基于 UDP 传输例程的使用方法。

本例程在小梅哥团队出品的 AC620 开发板或 Starter 开发板+多功能通信扩展板上用 verilog 实现以太网 UDP 协议通信。FPGA 程序接收到上位机发来的 UDP 数据包，通过解析目标 MAC address 来确定是否是发给 FPGA 的数据包。如果是的话，把数据包中的数据部分保存到 fifo 中。然后 FPGA 的发送程序再把 fifo 的数据包发送回上位机。

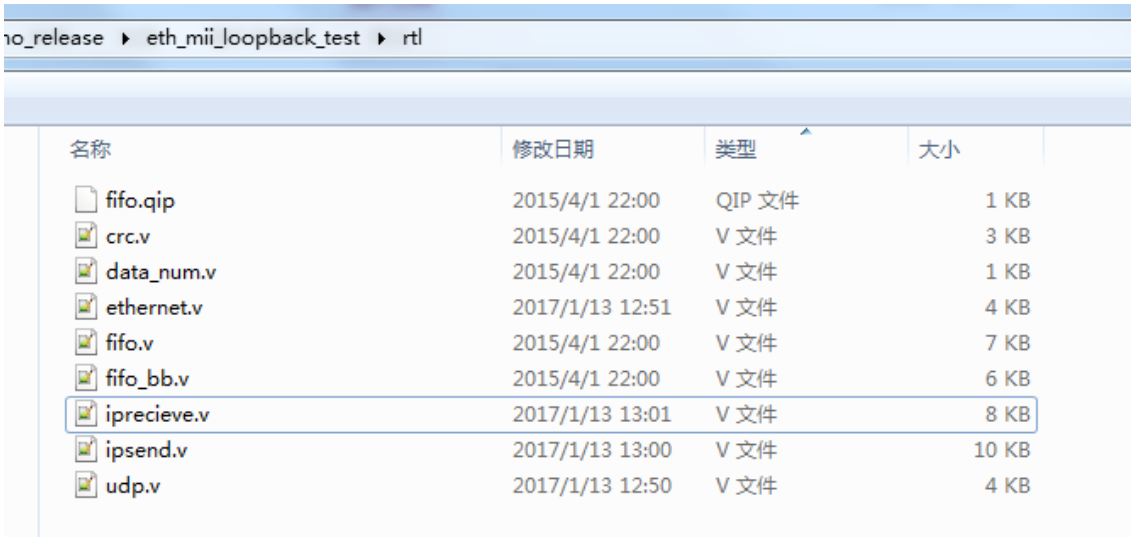


1、本例提供的 UDP 传输例程工程压缩包名为 eth_mii_loopback_test.rar，该文件可在我们提供的配套资料中找到。也可前往 www.corecourse.cn 网站下载

2、解压 eth_mii_loopback_test.rar 到不含中文或者空格的目录中，如 D:\fpga。解压后工程目录下内容如下所示：



其中 rtl 文件夹下存放的为程序源码，如下图所示：



这里对其中几个重要文件（夹）简单说明下功能：

文件或文件夹名	文件或文件夹功能描述
ethernet_test.qpf	工程文件，使用 Quartus II 13.0 打开。
ethernet_test.qsf	工程配置文件，记录了工程相关的各个设置，包括引脚分配。
ethernet_test.sof	工程编译后得到的配置文件，用以下载到 FPGA 中运行
eth_test.stp	signaltap ii 设置文件，测试时双击运行此文件以查看 MII 接口上发送和接受的数据内容，通过抓取的波形内容分析收发是否正确。

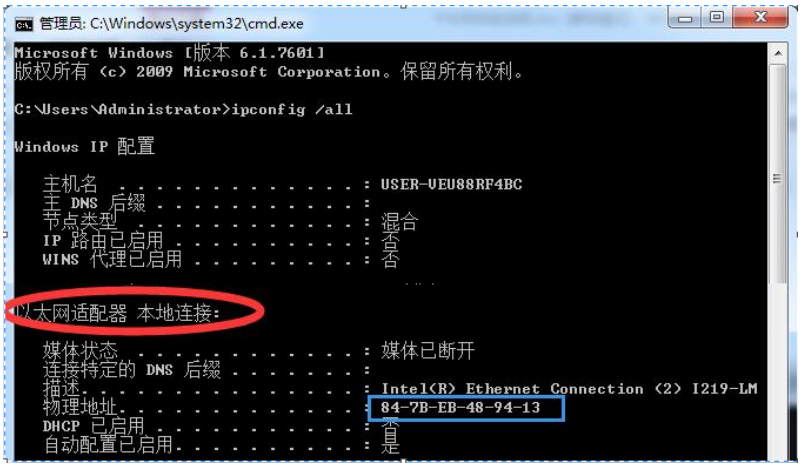
ethernet.v	测试工程顶层，例化了以太网 UDP 协议收发代码和测试代码，组成完整的测试工程。
udp.v	UDP 协议顶层，例化了 UDP 协议接受和发送代码，以及 CRC32 校验电路，以实现对发送的数据追加 32 位校验码的功能。
ipsend.v	UDP 协议发送模块，实现将发送缓存（RAM）中的数据组建成 UDP 协议包并发送出去的功能
iprecieve.v	UDP 协议接收模块，实现将以太网口接收到的数据解包、得到最后的用户数据部分并写入到接收缓存（RAM）中。
crc.v	CRC32 校验逻辑，实现对数据的 CRC32 校验功能
fifo.qip	数据包缓存，本例中，网口接受到的数据会被直接写入到该 fifo 中，所以，每当网口接受到数据后，就会更新发送缓存中的内容，然后 FPGA 的以太网发送逻辑从 fifo 中读取数据并发送回电脑
data_num.v	以太网包数据个数统计模块，统计每个以太网包的，实时以太网发送模块当前包已经发送的数据个数

3、双击 ethernet.qpf 以打开工程（强烈建议使用工程创建时候对应的版本即 Quartus II 13.0，使用其他版本打开或编译遇到问题，请邮件告知我们，以获得解决方案邮箱：xiaomeige_fpga@foxmail.com）

4、修改 UDP 发送模块(ipsend.v)中的目标 mac 地址(mema[27:16])为你使用的网卡的 mac address,修改后重新编译一边。

```
//此段为调试主机（例如用户的PC）的MAC地址，在调试时候，需要根据自己的电脑MAC地址修改本段内容
mema2[16]<=4'h4;     //目的MAC地址 84-7b-eb-48-94-13
mema2[17]<=4'h8;
mema2[18]<=4'hb;
mema2[19]<=4'h7;
mema2[20]<=4'hb;
mema2[21]<=4'he;
mema2[22]<=4'h8;
mema2[23]<=4'h4;
mema2[24]<=4'h4;
mema2[25]<=4'h9;
mema2[26]<=4'h3;
mema2[27]<=4'h1;
```

如果不知道自己 PC 网卡的 mac address，就在 DOS 命令窗口，用 ipconfig –all 命令看一下。



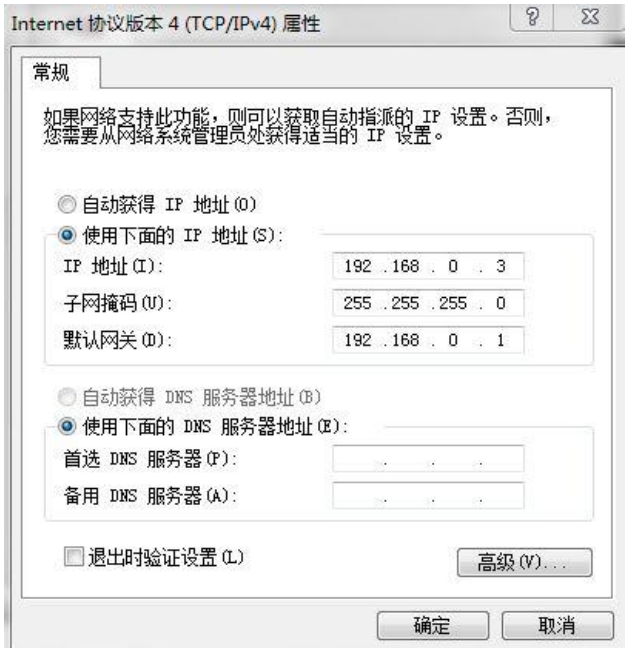
5、修改 iprecieve.v 中 185 行 mymac[39:0]==40'h847beb4894，后面的数字修改为您的 PC MAC 地址的前五个字节，如果您的 MAC 显示为 847beb489413（16 进制），那么此处应该输入 40'h847beb4894。

```
182                 state_counter<=5'd0;
183                 //if((mymac[87:72]==16'h000a)&&(mymac[71:
184                 //判断目标MAC Address是否为本FPGA
185                 if(mymac[39:0]==40'h847beb4894)
186                     state<=rx_IP_Protocol;
187                 else
188                     state<=rx_wait;
```

6、修改 PC 的 IP 地址为 192.168.0.3。PC 的 IP Address 需要和发送模块(ipsend.v)中设置一致，不然 PC 端会接收不到开发板发送的 UDP 数据包。当然，用户也可以修改代码中的目标 IP 地址，如下所示：

```
152         start:begin
153                 mema[0]<={16'h4500,total_len};          //total length is 44 bytes
154                 mema[1][31:16]<=mema[1][31:16]+1;
155                 mema[1][15:0]<=16'h4000;
156                 mema[2]<=32'h80110000;                  //mema[2][15:0] ip  jiaoyanhe
157                 mema[3]<=32'hc0a80002;                  //192.168.0.2源地址
158                 mema[4]<=32'hc0a80003;                  //192.168.0.3目的地址广播地址
159                 mema[5]<=32'h80008000;                  //2个字节的源端口号和2个字节的目的端口号
160                 mema[6]<={mydata_num,16'h0000};             //2个字节的数据长度(24byte数据)和2个字节的校验和（无）
161                 i<=0;
162                 stat<=make;
```

代码中本机（FPGA）和目标机（PC）IP 地址

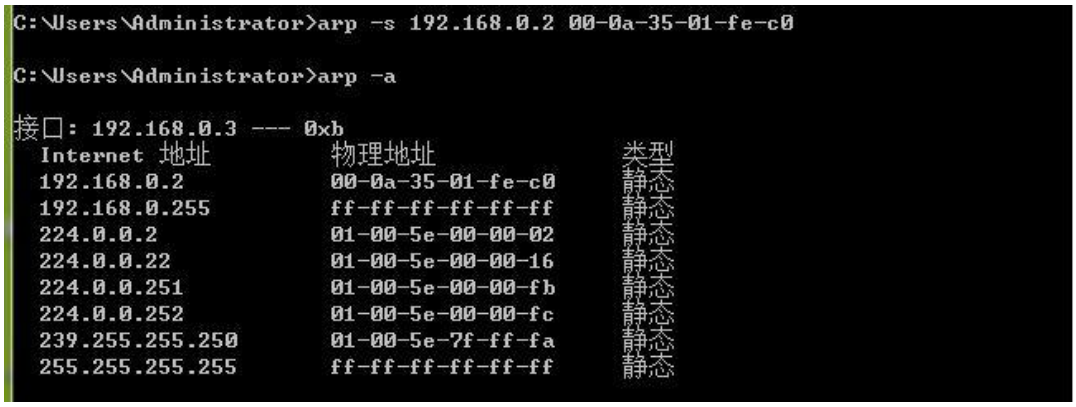


修改 PC 的 IP 地址

7、在 DOS 命令窗口绑定开发板的 IP 地址和 MAC 地址，（由于本测试工程不支持 ARP 协议，因此只能通过这种 IP 和 MAC 绑定的方式来强制将开发板的 IP 地址和 MAC 地址关联在一起，这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。）

运行命令：ARP -s 192.168.0.2 00-0a-35-01-fe-c0

绑定后我们可以用 arp -a 命令来查看 PC 上绑定的结果。



如果运行 ARP 出现加载失败，换另一种方法绑定

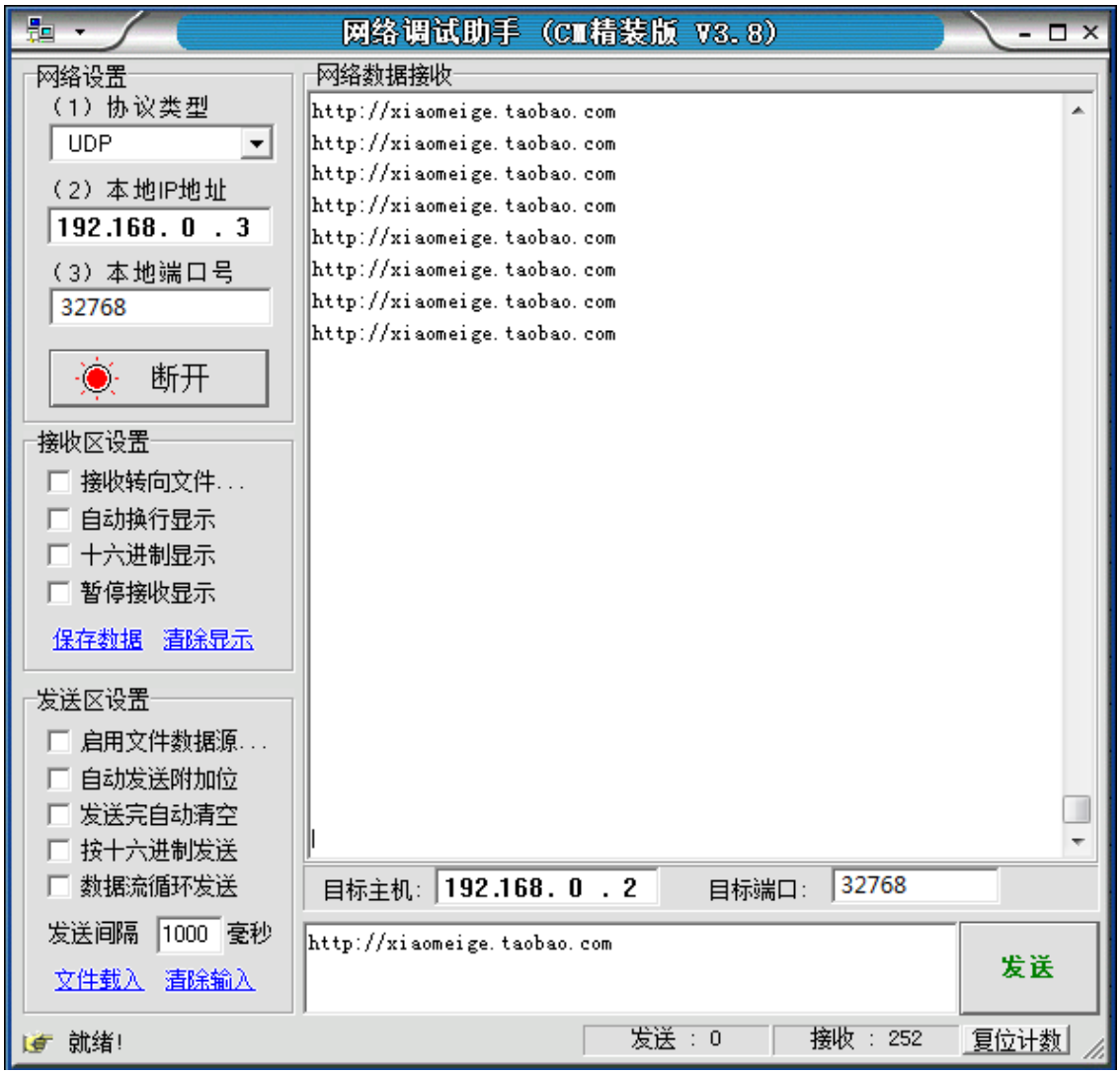
使用 netsh i i show in 命令查看本地连接的 idx 编号，如“11”

使用 netsh -c “i i” add neighbors 11(idx 编号) “192.168.0.2” “00-0a-35-01-fe-c0”

使用 arp -a 命令来查看 PC 上帮定的结果

8、打开 Quartus II 的 Programer，选择下载器和需要下载的文件，然后点击下载以开始下载 ethernet.sof 文件到开发板中。

9、打开网络调试助手并按照如图所示设置各项参数，再按连接按钮(这里的本地的 IP 地址为 PC 的 IP Address，本地端口需要跟 FPGA 程序中的一致，为 32768，注意代码中为 16 进制的数 8000，因此对应的 10 进制数为 32768)。

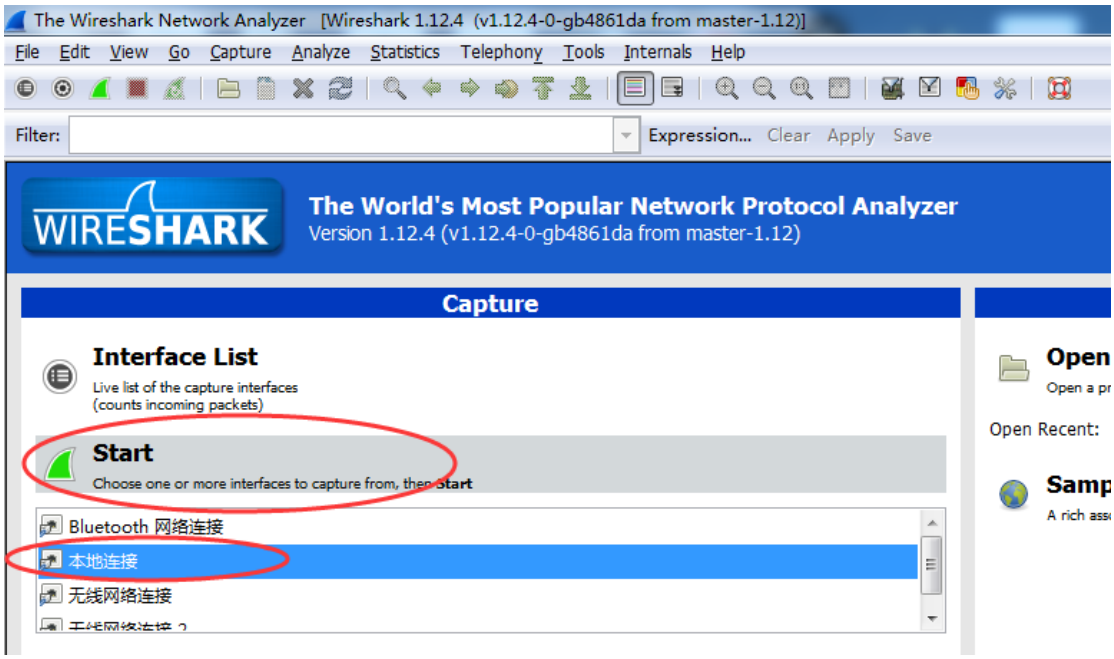


点就连接后，目标主机(192.168.0.2)和目标端口(32768)都是默认值。

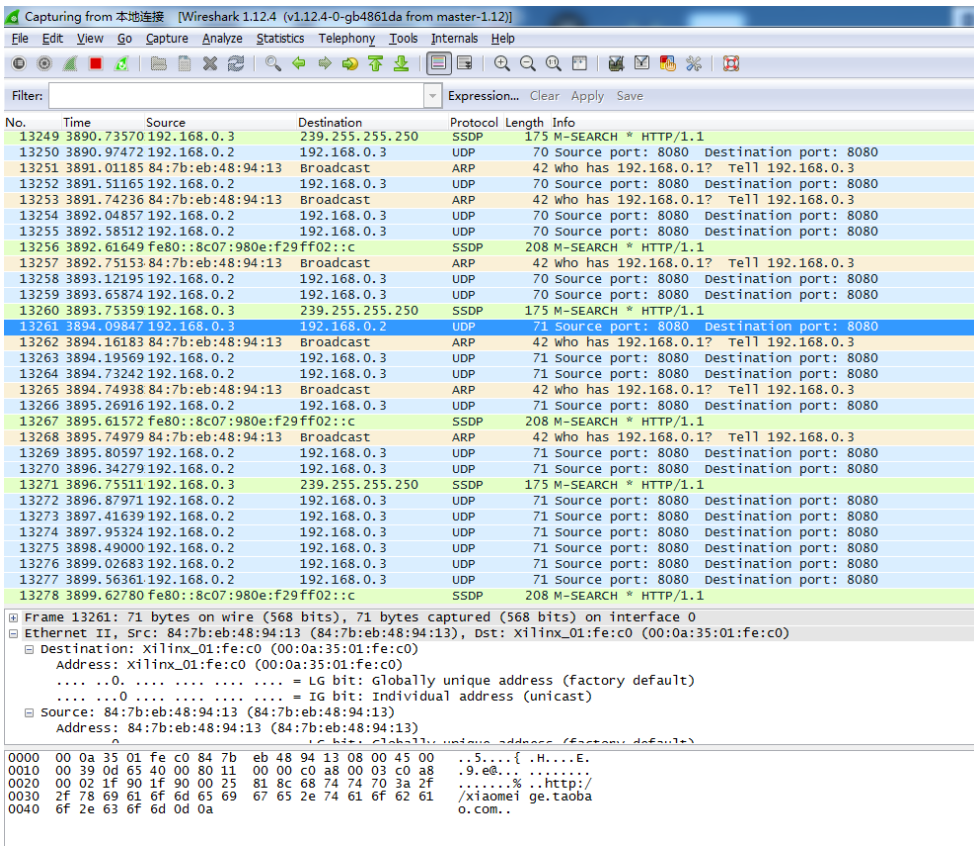
10、在发送窗口中输入你希望发送的数据，然后点击发送，则上方的接收窗口会接收到与发送框完全一样的内容，每点击一次发送，就会收到一次数据。

11、安装网络抓包工具 Wireshark，我们在实验的时候可以用这工具来查看 PC 网口发送的数据和接收到的数据。

12、打开安装好的 wireshark 抓包工具。在软件界面选择您 PC 的有线网卡，按开始按钮开始抓包。

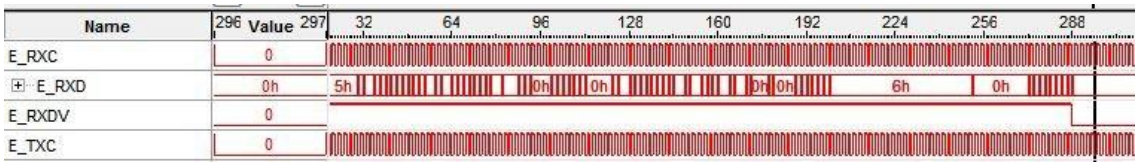


在 wireshark 抓包窗口我们可以看到开发板(192.168.0.2)向 PC 网口(192.168.0.3)发来的数据包。



13、注意：以太网的数据帧的传输有包长的要求，一般在 46～1500 字节。所以在发送以太网数据包的时候,数据帧的长度不能太短，不然会导致 PC 数据包发送而 FPGA 收不到数据包的情况。

双击 eth_test.stp 文件以打开嵌入式逻辑分析仪工具，然后 run，PC 端发送数据，则可以在 Signaltap II 上抓取到数据波形，如下图所示：

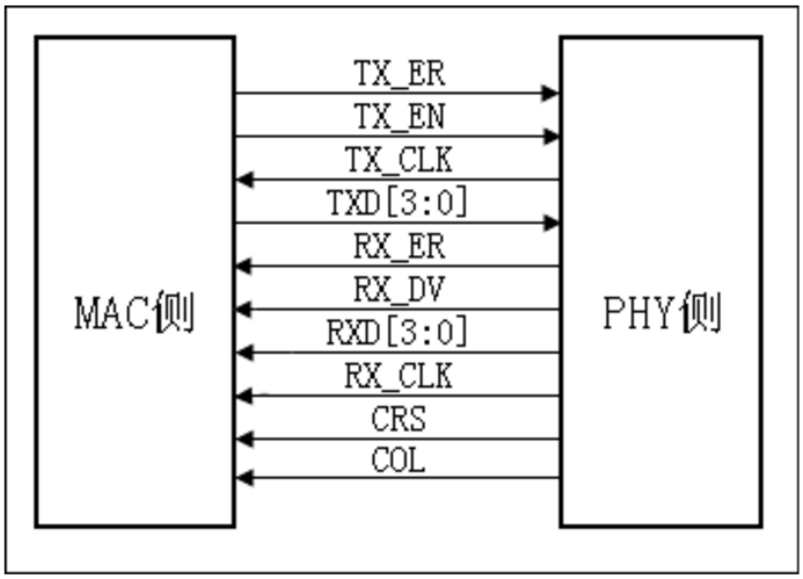


以太网 MII 接口接收数据波形

以太网 MII 接口发送数据波形

第 2 章 以太网 MAC 层基本原理

MII 接口介绍：



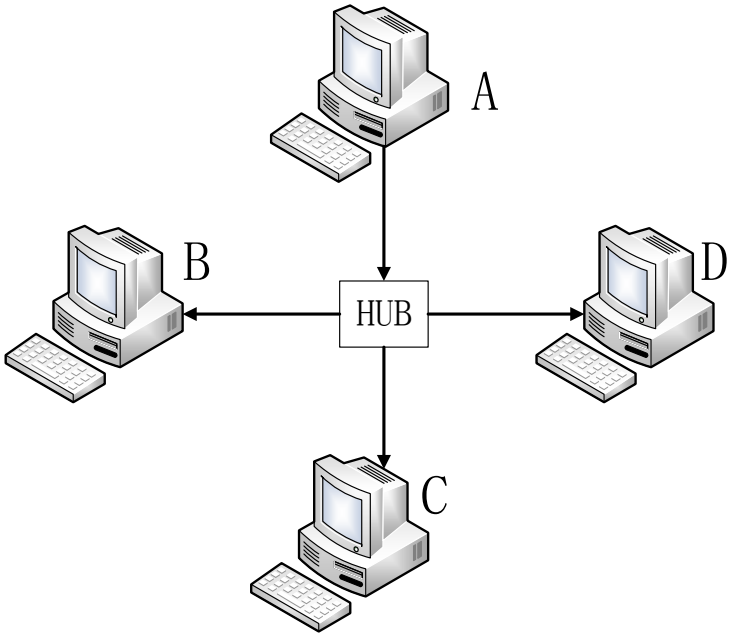
	信号	方向	宽度	功能
发送	mii_tx_clk	in	1	MII 接口发送时钟，由 PHY 芯片产生，100Mbps 时为 25MHz，10Mbps 时为 2.5MHz
	mii_tx_en	out	1	MII 接口发送数据使能信号，高电平有效
	mii_tx_er	out	1	发送错误，用以破坏数据包发送
	mii_tx_data	out	4	MII 接口发送数据线，FPGA 通过该数据线将需要发送的数据依次送给 PHY 芯片
接收	mii_rx_clk	in	1	MII 接口接收时钟，由 PHY 芯片产生，100Mbps 时为 25MHz，10Mbps 时为 2.5MHz
	mii_rx_dv	in	1	MII 接口接收数据有效信号，高电平有效
	mii_rx_er	in	1	接收错误，本实例中暂时忽略该信号
	mii_rx_data	in	4	MII 接口数据总线，FPGA 通过该数据线读取 PHY 芯片接收到的以太网数据
	CRS			Carrier Sense，载波侦测信号，不需要同步于参考时钟，只要有数据传输，CRS 就有效，另外，CRS 只有 PHY 在半双工模式下有效
	COL			Collision Detectd，冲突检测信号，不需要同步于参考时钟，只有 PHY 在半双工模式下有效。

以太网（MAC）帧介绍

字段	字段长度（字节）	目的
前导码（Preamble）	7	同步
帧开始符（SFD）	1	标明下一个字节为目的 MAC 字段
目的 MAC 地址	6	指明帧的接受者
源 MAC 地址	6	指明帧的发送者
长度/类型（Length/ Type）	2	帧的数据字段的长度（长度或类型）
数据和填充（Data and Pad）注	46~1500	高层的数据，通常为 3 层协议数据单元。对于 TCP/IP 是 IP 数据包
帧校验序列（FCS）	4	对接收网卡提供判断是否传输错误的一种方法，如果发现错误，丢弃此帧

以太网帧 64 字节（数据段 46 字节）最小长度原因

以太网是不可靠的，这意味着它并不知道对方有没有收到自己发出的数据包，但如果他发出的数据包发生错误，需要进行重传。以太网的错误主要是发生碰撞，碰撞是指两台机器同时监听到网络是空闲的，同时发送数据，就会发生碰撞，碰撞对于以太网来说是正常的。



集线器（HUB）是一种网络共享媒体，所谓网络共享媒体，即在单一时间点内， 仅能被一部主机所使用。

理解了共享媒体的意义后，再来，我们就得要讨论，那么以太网络的网卡之间是如何传输的呢？我们以上图中的 A 要发给 D 网卡为例好了，简单的说， CSMA/CD 搭配上述的环境，它的传输情况需要有以下的流程：

1. 监听媒体使用情况 (Carrier Sense)：A 主机要发送网络封包前，需要先对网络媒体进行监听，确认没有人在使用后， 才能够发送出讯框；

2. 多点传输 (Multiple Access)：A 主机所送出的数据会被集线器复制一份，然后传送给所有连接到此集线器的主机！ 也就是说， A 所送出的数据， B, C, D 三部计算机都能够接收的到！但由于目标是 D 主机，因此 B 与 C 会将此讯框数据丢弃，而 D 则会抓下来处理；

3. 碰撞侦测 (Collision Detection)：该讯框数据附有检测能力，若其他主机例如 B 计算机也刚好在同时间发送讯框数据时， 那么 A 与 B 送出的数据碰撞在一块 (出车祸) ，此时这些讯框就是损毁，那么 A 与 B 就会各自随机等待一个时间， 然后重新透过第一步再传送一次该讯框数据。

我们来看一下，假设 A 检测到网络是空闲的，开始发数据包，尽力传输，当数据包还没有到达 B 时，B 也监测到网络是空闲的，开始发数据包，这时就会发生碰撞，B 首先发现发生碰撞，开始发送碰撞信号，所谓碰撞信号，就是连续的 01010101 或者 10101010，十六进制就是 55 或 AA。这个碰撞信号会返回到 A，如果碰撞信号到达 A 时，A 还没有发完这个数据包，A 就知道这个数据包发生了错误，就会重传这个数据包。但如果碰撞信号会返回到 A 时，数据包已经发完，则 A 不会重传这个数据包。

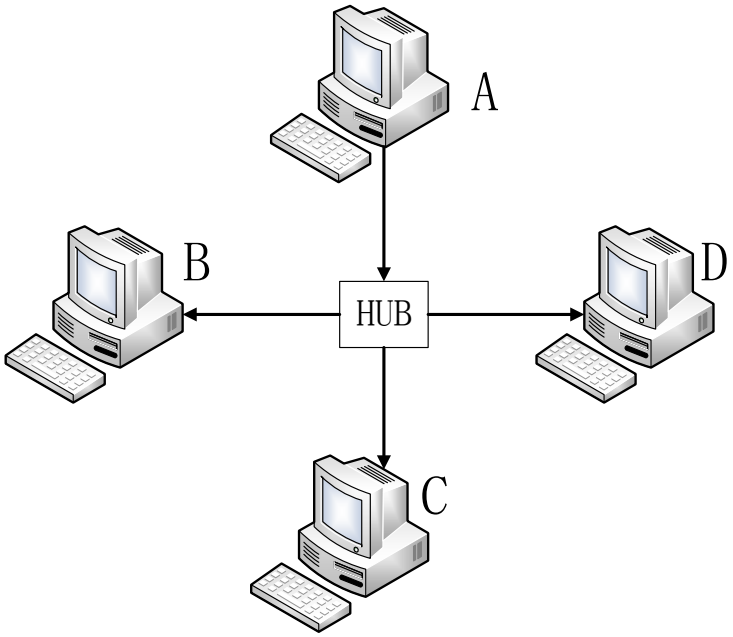
要保证以太网的重传，必须保证 A 收到碰撞信号的时候，数据包没有传完，要实现这一要求，A 和 B 之间的距离很关键，也就是说信号在 A 和 B 之间传输的来回时间必须控制在一定范围之内。IEEE 定义了这个标准，一个碰撞域内，最远的两台机器之间的 round-trip time 要小于 512bit time.(来回时间小于 512 位时，所谓位时就是传输一个比特需要的时间)。这也是我们常说的一个碰撞域的直径。

512 个位时，也就是 64 字节的传输时间，如果以太网数据包大于或等于 64 个字节，就能保证碰撞信号到达 A 的时候，数据包还没有传完。

总结：最小数据帧的设计原因和以太网电缆长度有关，为的是让两个相距最远的站点能够感知到双方的数据发生了碰撞;最远两端数据的往返时间就是争用期，以太网的争用期是 51.2 微妙，正好发送 64byte 数据。

注：为什么以太网的争用期是 51.2us ?

在极限条件下，一个局域网中两个收发器间（允许接 4 个中继器）的最大距离为 2500m，往返 5000m，同轴电缆的时延特性为 5us/km，即如遇冲突，端到端往返时延为 25us。然而这是理想的时延，考虑到中继器的额外时延，最坏情况下取估计时延为 45us，再加上强化冲突需发送 48bit，接受方要接受到 48bit 后才确认冲突，即在增加 4.8us，共 49.8us，所以通常以太网取 51.2us 为争用期的时间长度(传输 512bit，即 64 字节时间)，即帧的长度至少为 64 字节。



第 3 章 ARP 协议的原理与 FPGA 实现

ARP 帧介绍：

ARP 帧存在的目的：

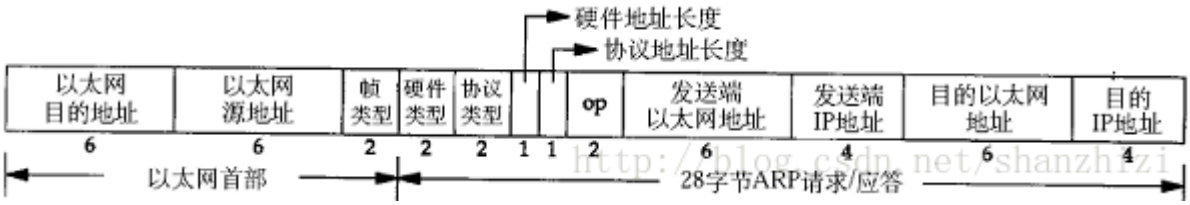
在网络通讯时，源主机的应用程序知道目的主机的 IP 地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP 协议就起到这个作用。

ARP 帧工作原理：

源主机发出 ARP 请求，询问“IP 地址是 192.168.0.1 的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填 FF:FF:FF:FF:FF:FF 表示广播），目的主机接收到广播的 ARP 请求，发现其中的 IP 地址与本机相符，则发送一个 ARP 应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个 ARP 缓存表，可以用 arp -a 命令查看。缓存表中的表项有过期时间（一般为 20 分钟），如果 20 分钟内没有再次使用某个表项，则该表项失效，下次还要发 ARP 请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？ARP 数据报的格式如下图所示（该图出自[TCPIP]）：

ARP 数据报格式

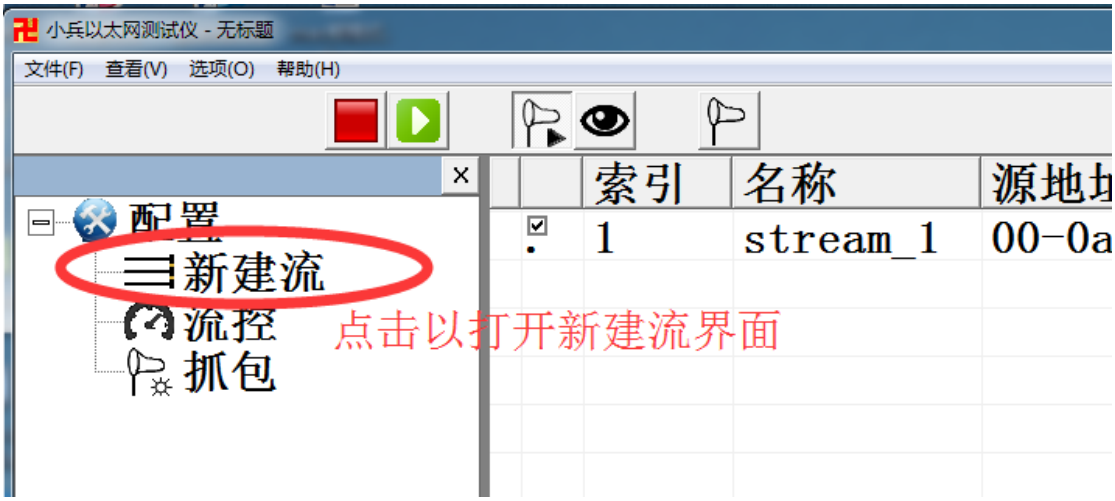


注意到源 MAC 地址、目的 MAC 地址在以太网首部和 ARP 请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1 为以太网，协议类型指要转换的地址类型，0x0800 为 IP 地址，后面两个地址长度对于以太网地址和 IP 地址分别为 6 和 4（字节），op 字段为 1 表示 ARP 请求，op 字段为 2 表示 ARP 应答。

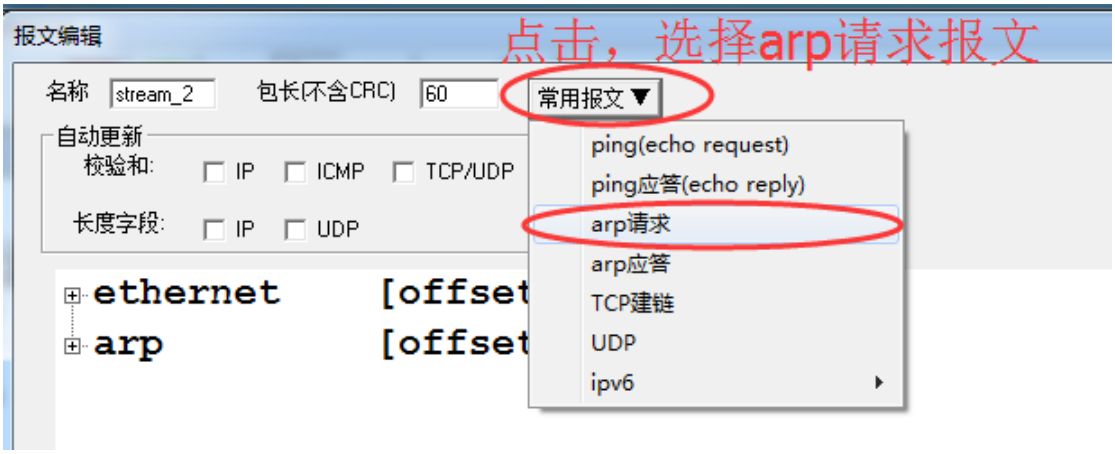
以太网首部	字段	长度	默认值	备注
	接收方 MAC	6		目的 MAC 地址
	发送方 MAC	6		源 MAC 地址
	Ethertype	2	0x0806	0x0806 是 ARP 帧的类型值
ARP 字段	硬件类型	2	0x0001	以太网类型值
	上层协议类型	2	0x0800	上层协议为 IP 协议
	MAC 地址长度	1	0x6	以太网 MAC 地址长度为 6
	IP 地址长度	1	0x4	IP 地址长度为 4
	操作码	2	0x1/0x2	0x1 表示 ARP 请求包,0x2 表示应答包
	发送方 MAC	6		源 MAC 地址
	发送方 IP	4		源 IP 地址
	接收方 MAC	6		目的 MAC 地址
	接收方 IP	4		目的 IP 地址
	填充数据	18		因为物理帧最小长度为 64 字节,前面的 42 字节再加上 4 个 CRC 校验字节,还差 18 个字节

以太网校验	校验字	4		CRC32 校验值，以上所有数据参与校验的结果

使用以太网发包工具组 ARP 包

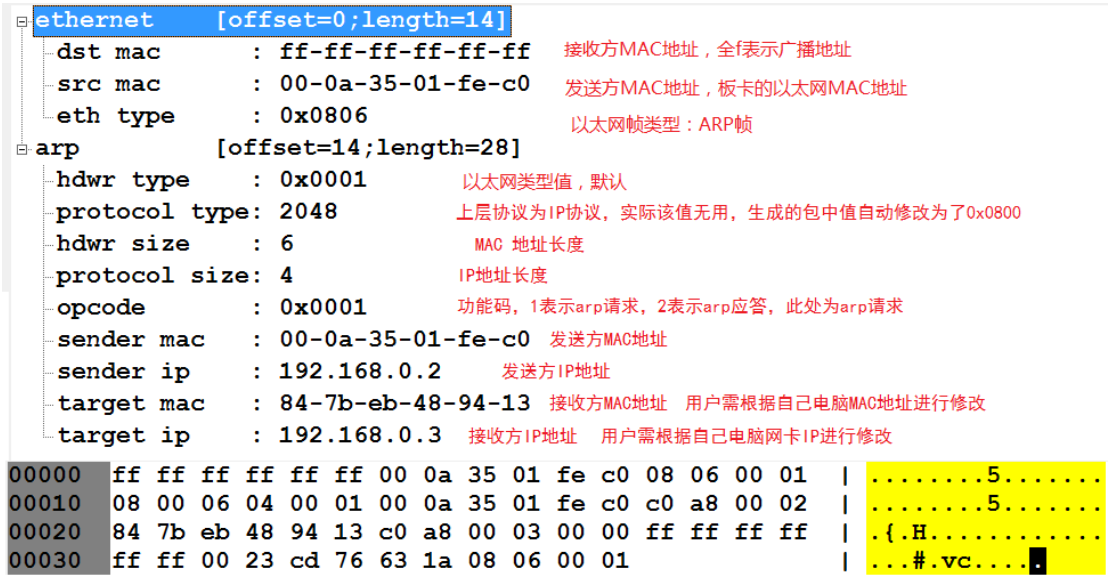


在常用报文中选择 arp 请求格式报文



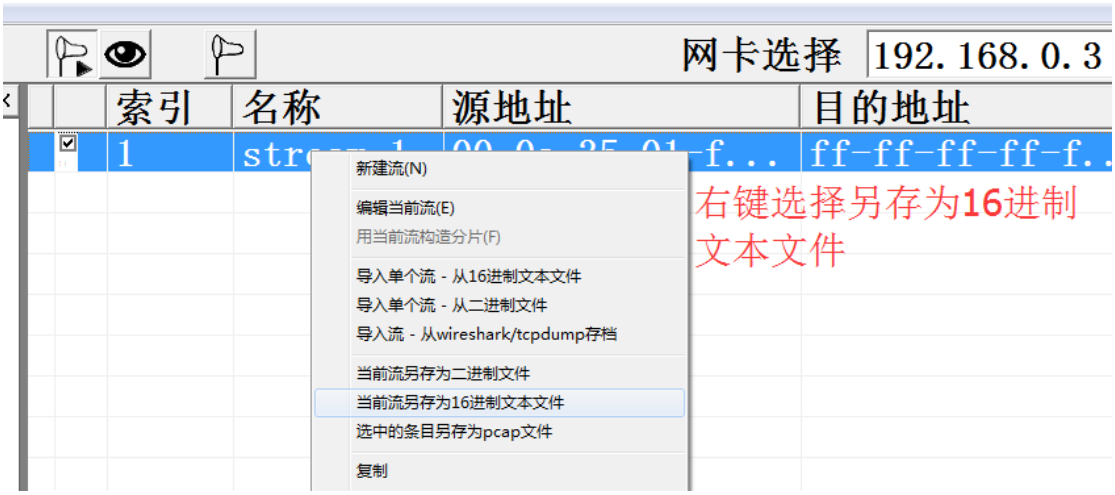
修改以太网包和 arp 包中各字段内容如下所示

数据帧内容：ARP



配置好之后点击确定。

将设置好的数据包导出为 16 进制格式文本文件

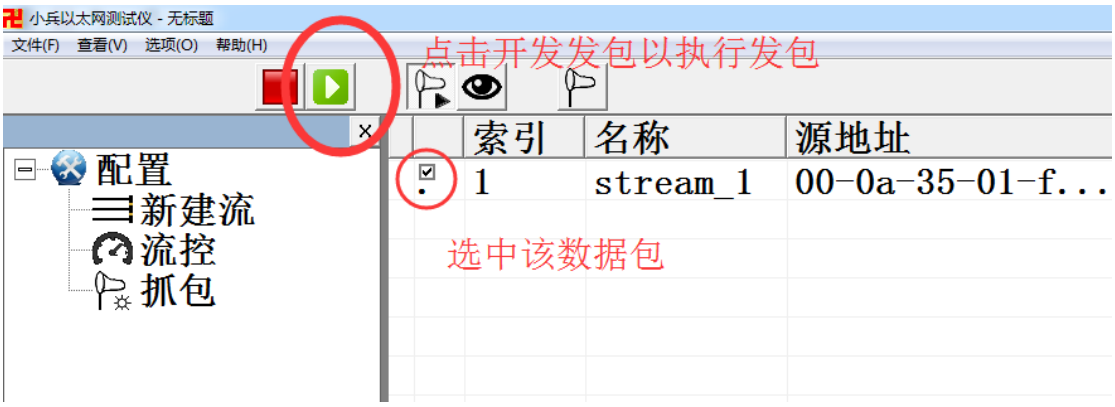


得到的文本中数据内容如下所示：

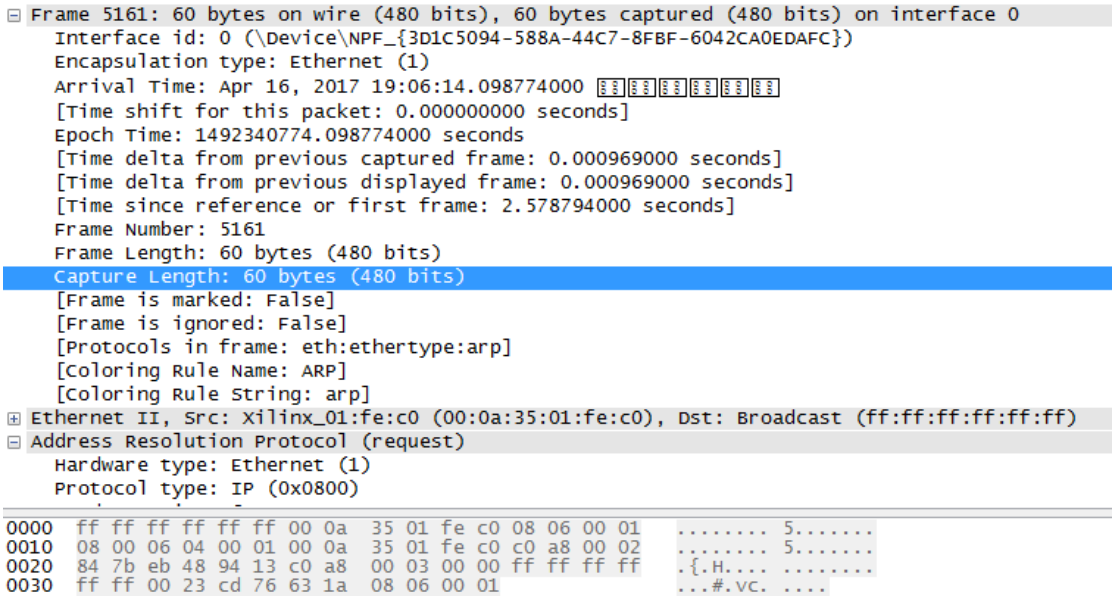
0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x0a 0x35 0x01 0xfe 0xc0 0x08 0x06 0x00 0x01 0x08 0x00 0x06 0x04 0x00 0x01 0x00 0x0a 0x35 0x01 0xfe 0xc0 0xc0 0xa8 0x00 0x02 0x84 0x7b 0xeb 0x48 0x94 0x13 0xc0 0xa8 0x00 0x03 0x00 0x00 0xff 0xff 0xff 0xff 0xff 0xff 0x00 0x23 0xcd 0x76 0x63 0x1a 0x08 0x06 0x00 0x01

使用以太网发包工具发包

选中该数据包，执行开始发布



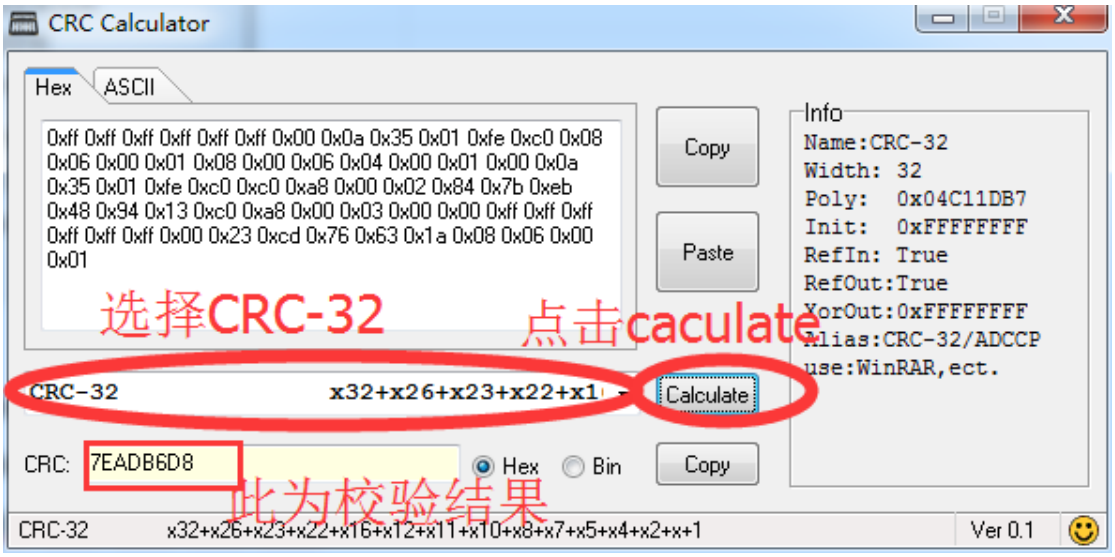
使用 **wireshark** 软件抓包



使用 **CRC** 计算软件计算 **CRC** 校验值

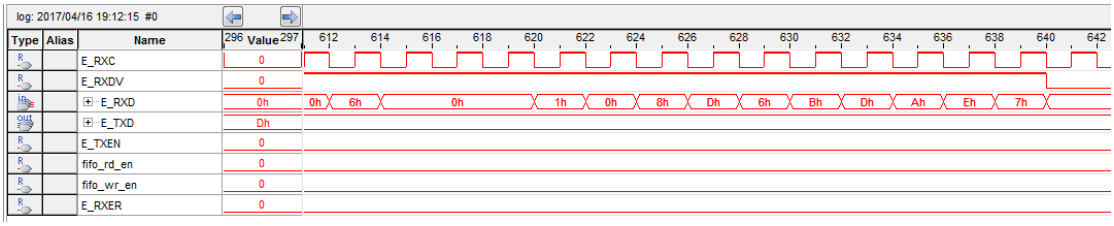
完整的以太网数据包结尾是帧校验（FCS），即 CRC32 值，由于整个过程中都没有出现 CRC 校验字段的内容，而我们使用 FPGA 发包需要计算 CRC 校验字段，CRC 校验字段的产生可以使用软件或硬件的形式产生，本节实验，为了降低难度，使用 CRC 计算软件手动计算出数据报的 CRC 值，因此使用专用 CRC 校验工具对数据内容进行计算以得到校验字：

CRC 校验软件计算校验内容



使用 FPGA signaltap 抓取 MII 接口接收信号

使用 FPGA 抓取 MII 接口的接收部分信号，抓到的网卡发送数据如下所示：



CRC 计算软件结果与以太网帧 CRC 值关系

CRC 软件计算结果：低字节在前，高字节在后。使用 CRC 校验结果时，应以字节位为单位调换高低顺序，如 7EADB6D8 应该调换为 D8B6AD7E。

```
//整个数据包 CRC 校验值，本例中使用 CRC 计算软件计算得出。

parameter CRC = 32'h7eadb6d8;



wire [31:0]CRC_result;

//调整 CRC 校验字字节顺序以符合以太网数据包校验格式

assign CRC_result = {CRC[7:0],CRC[15:8],CRC[23:16],CRC[31:24]};
```

第 4 章 以太网 ARP 帧发包实例（手动 CRC）

基于 AC620 开发板上的以太网接口，设计一个能够发送 ARP 帧的 FPGA 系统。其中以太网包和 ARP 包采用分层组包的形式。即底层为以太网帧（MAC 帧）生成层。上层为 ARP 包生成层。本设计实例提供的工程名称为 *arp_send.qar*

eth_send.v 为以太网 MAC 帧组包逻辑，完成以太网 MAC 帧的建立和发送。

Eth_send_test.v 为 ARP 包组包工具，并调用 eth_send 完成完整的 ARP 报文的发送。

例化以太网帧发送模块，并将目的地址设置为广播地址 ffffffff、源地址为板卡地址。这里选择某 Xilinx 官板的 mac 地址，以避免与其他实际地址冲突。协议类型为 ARP(0806)。数据长度是整个发送的 arp 协议报文的长度字节数的两倍（MII 接口将一个字节拆成 2 个 4 位数分两次发送，因此数据长度为实际发送的 arp 报文长度的 2 倍）。

```
eth_send eth_send(

        .rst_n(rst_n),

        .tx_go(tx_go),

        .data_length(12'd92),

        .des_mac(48'hff_ff_ff_ff_ff_ff),

        .src_mac(48'h00_0a_35_01_fe_c0),

        .type_length(16'h08_06),

        .CRC_Result(CRC_result),

        .fifo_rdreq(fifo_rdreq),

        .fifo_rddata(fifo_rddata),

        .fifo_rdclk(fifo_rdclk),

        .mii_tx_clk(mii_tx_clk),

        .mii_tx_en(mii_tx_en),

        .mii_tx_er(mii_tx_er),
```

```
        .mii_tx_data(mii_tx_data)

    );
```

Tx_go 为单次发送使能信号，每个 tx_go 信号的高脉冲启动一次发送。这里使用一个 24 位计数器进行计数，每计数一轮，tx_go 有效一次，即启动一次发送。整个计数周期为 671ms 左右（40ns *2^24）。

```
    //发送间隔计数器
    reg [23:0]cnt;


    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
        cnt <= #1 0;
        else //计数器自增，不考虑溢出，接受溢出自动清零
        cnt <= #1 cnt + 1'b1;


    //每 671ms 启动一次发送数据
    assign tx_go = (cnt == 24'd1);
```

以太网帧发送模块发送的数据段部分采用 fifo 接口，本例中仅为测试作用，因此使用查找表加自动地址累加的方式代替 fifo 输出 arp 数据包给以太网帧发送模块。在实际使用 fifo 时，需要将 fifo 设置为 showahead 模式。

```
//计数发送数据个数，用于产生对应的数据
    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
        data_cnt <= #1 12'd0;
```

```
    else if(fifo_rdreq)//fifo 读请求有效时，数据计数器累加。
        data_cnt <= #1 data_cnt + 1'b1;
    else
        data_cnt <= #1 12'd0;
```

以下使用查找表形式依次输出 arp 协议的报文内容。

```
//ARP 包数据段
    always@(*)
    begin
        case(data_cnt)
            //hdwr type
            00: fifo_rddata =   4'h0;
            01: fifo_rddata =   4'h0;
            02: fifo_rddata =   4'h1;
            03: fifo_rddata =   4'h0;

            //protocol type
            04: fifo_rddata =   4'h8;
            05: fifo_rddata =   4'h0;
            06: fifo_rddata =   4'h0;
            07: fifo_rddata =   4'h0;

            //hdwr size
            08: fifo_rddata =   4'h6;
            09: fifo_rddata =   4'h0;

            //protocol size
            10: fifo_rddata =   4'h4;
```

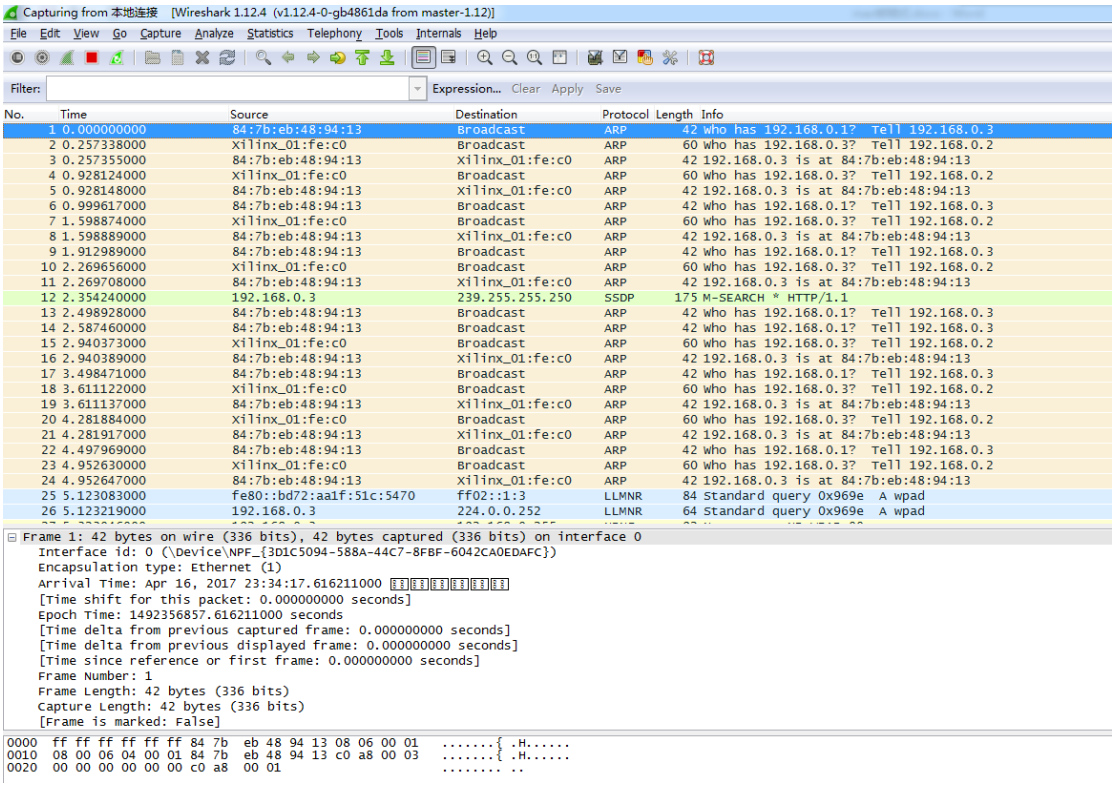
```
            11: fifo_rddata =   4'h0;


            //opcode
            12: fifo_rddata =   4'h0;
            13: fifo_rddata =   4'h0;
            14: fifo_rddata =   4'h1;
            15: fifo_rddata =   4'h0;


            //sender mac
            16: fifo_rddata =   4'h0;
            17: fifo_rddata =   4'h0;
            18: fifo_rddata =   4'ha;
            19: fifo_rddata =   4'h0;
            20: fifo_rddata =   4'h5;
            21: fifo_rddata =   4'h3;
            22: fifo_rddata =   4'h1;
            23: fifo_rddata =   4'h0;
            24: fifo_rddata =   4'he;
            25: fifo_rddata =   4'hf;
            26: fifo_rddata =   4'h0;
            27: fifo_rddata =   4'hc;


            //sender ip : 192.168.0.2
            28: fifo_rddata =   4'h0;//192
            29: fifo_rddata =   4'hc;


            30: fifo_rddata =   4'h8;//168
            31: fifo_rddata =   4'ha;


            32: fifo_rddata =   4'h0;//0
```

```
            33: fifo_rddata =   4'h0;


            34: fifo_rddata =   4'h2;
            35: fifo_rddata =   4'h0;//2


            //target mac
            36: fifo_rddata =   4'h4;
            37: fifo_rddata =   4'h8;
            38: fifo_rddata =   4'hb;
            39: fifo_rddata =   4'h7;
            40: fifo_rddata =   4'hb;
            41: fifo_rddata =   4'he;
            42: fifo_rddata =   4'h8;
            43: fifo_rddata =   4'h4;
            44: fifo_rddata =   4'h4;
            45: fifo_rddata =   4'h9;
            46: fifo_rddata =   4'h3;
            47: fifo_rddata =   4'h1;


            //target ip : 192.168.0.3
            48: fifo_rddata =   4'h0;//192
            49: fifo_rddata =   4'hc;


            50: fifo_rddata =   4'h8;//168
            51: fifo_rddata =   4'ha;


            52: fifo_rddata =   4'h0;//0
            53: fifo_rddata =   4'h0;


            54: fifo_rddata =   4'h3;//3
```

```
55: fifo_rddata =   4'h0;

//填充字段，以使整个数据帧长度达到 64 字节
56: fifo_rddata =   4'h0;
57: fifo_rddata =   4'h0;
58: fifo_rddata =   4'h0;
59: fifo_rddata =   4'h0;
60: fifo_rddata =   4'hf;
61: fifo_rddata =   4'hf;
62: fifo_rddata =   4'hf;
63: fifo_rddata =   4'hf;
64: fifo_rddata =   4'hf;
65: fifo_rddata =   4'hf;
66: fifo_rddata =   4'hf;
67: fifo_rddata =   4'hf;
68: fifo_rddata =   4'hf;
69: fifo_rddata =   4'hf;
70: fifo_rddata =   4'hf;
71: fifo_rddata =   4'hf;
72: fifo_rddata =   4'h0;
73: fifo_rddata =   4'h0;
74: fifo_rddata =   4'h3;
75: fifo_rddata =   4'h2;
76: fifo_rddata =   4'hd;
77: fifo_rddata =   4'hc;
78: fifo_rddata =   4'h6;
79: fifo_rddata =   4'h7;
80: fifo_rddata =   4'h3;
81: fifo_rddata =   4'h6;
82: fifo_rddata =   4'ha;
```

```
            83: fifo_rddata =   4'h1;
            84: fifo_rddata =   4'h8;
            85: fifo_rddata =   4'h0;
            86: fifo_rddata =   4'h6;
            87: fifo_rddata =   4'h0;
            88: fifo_rddata =   4'h0;
            89: fifo_rddata =   4'h0;
            90: fifo_rddata =   4'h1;
            91: fifo_rddata =   4'h0;
            default:fifo_rddata = 4'h0;
        endcase
    end
```

用户根据自己的电脑 MAC 地址，使用以太网发包工具组件新的包，并使用 CRC 计算软件计算得到新的 CRC 校验值，将新的数据导入到工程中，编译工程得到 sof 文件，烧写到 AC620 开发板中，在 wireshark 工具中就能连续不断的收到由开发板发出的 arp 请求了。如下图所示。（注意，由于板卡使用的 MAC 地址参考了某 Xilinx 官方板卡的 MAC 地址，因此 wireshark 软件自带识别出了硬件名称并将 source 名称显示为 Xilinx_01 ）



第 5 章 以太网 ARP 帧发包实例（自动 CRC）

上节例子中，以太网帧的 FCS 字段使用 PC 端 CRC 校验工具生成，该种方式仅限于以太网帧发送模块设计初期检查协议的数据流控制，当数据报中任何一个字节的内容更改之后，CRC 的值都需要重新计算，这在实际应用中是不现实的。因此本节将引入自动 CRC 校验值的生成。本设计实例提供的工程名称为 *arp_send_with_crc.qar*

关于 CRC 校验能够降低数据链路通信出错概率的理论推导，这里不过多介绍，请有兴趣的自行通过网络学习。本节仅使用一个公版的 CRC32 校验程序，作为以太网帧校验字段的运算单元。

```
`timescale 1ns/1ns

module crc32_d4 (

    Clk,

    Rst_n,

    Data,
```

```
    Enable,
    Initialize,
    Crc,
    CrcError,
    Crc_eth
);



parameter Tp = 1;


input Clk;  //运算时钟，与数据时钟同步
input Rst_n;    //复位
input [0:3] Data;   //待运算数据
input Enable;   //使能计算
input Initialize;   //初始化


output [31:0] Crc;  //实时 CRC 结果
output [31:0] Crc_eth;  //以太网帧适用的 CRC 运算结果


output CrcError;


reg  [31:0] Crc;


wire [31:0] CrcNext;


assign CrcNext[0] = Enable & (Data[0] ^ Crc[28]);
assign CrcNext[1] = Enable & (Data[1] ^ Data[0] ^ Crc[28] ^
Crc[29]);
```

```
assign CrcNext[2] = Enable & (Data[2] ^ Data[1] ^ Data[0] ^ Crc[28]
^ Crc[29] ^ Crc[30]);

assign CrcNext[3] = Enable & (Data[3] ^ Data[2] ^ Data[1] ^ Crc[29]
^ Crc[30] ^ Crc[31]);

assign CrcNext[4] = (Enable & (Data[3] ^ Data[2] ^ Data[0] ^
Crc[28] ^ Crc[30] ^ Crc[31])) ^ Crc[0];

assign CrcNext[5] = (Enable & (Data[3] ^ Data[1] ^ Data[0] ^
Crc[28] ^ Crc[29] ^ Crc[31])) ^ Crc[1];

assign CrcNext[6] = (Enable & (Data[2] ^ Data[1] ^ Crc[29] ^
Crc[30])) ^ Crc[ 2];

assign CrcNext[7] = (Enable & (Data[3] ^ Data[2] ^ Data[0] ^
Crc[28] ^ Crc[30] ^ Crc[31])) ^ Crc[3];

assign CrcNext[8] = (Enable & (Data[3] ^ Data[1] ^ Data[0] ^
Crc[28] ^ Crc[29] ^ Crc[31])) ^ Crc[4];

assign CrcNext[9] = (Enable & (Data[2] ^ Data[1] ^ Crc[29] ^
Crc[30])) ^ Crc[5];

assign CrcNext[10] = (Enable & (Data[3] ^ Data[2] ^ Data[0] ^
Crc[28] ^ Crc[30] ^ Crc[31])) ^ Crc[6];

assign CrcNext[11] = (Enable & (Data[3] ^ Data[1] ^ Data[0] ^
Crc[28] ^ Crc[29] ^ Crc[31])) ^ Crc[7];

assign CrcNext[12] = (Enable & (Data[2] ^ Data[1] ^ Data[0] ^
Crc[28] ^ Crc[29] ^ Crc[30])) ^ Crc[8];

assign CrcNext[13] = (Enable & (Data[3] ^ Data[2] ^ Data[1] ^
Crc[29] ^ Crc[30] ^ Crc[31])) ^ Crc[9];

assign CrcNext[14] = (Enable & (Data[3] ^ Data[2] ^ Crc[30] ^
Crc[31])) ^ Crc[10];

assign CrcNext[15] = (Enable & (Data[3] ^ Crc[31])) ^ Crc[11];

assign CrcNext[16] = (Enable & (Data[0] ^ Crc[28])) ^ Crc[12];

assign CrcNext[17] = (Enable & (Data[1] ^ Crc[29])) ^ Crc[13];

assign CrcNext[18] = (Enable & (Data[2] ^ Crc[30])) ^ Crc[14];

assign CrcNext[19] = (Enable & (Data[3] ^ Crc[31])) ^ Crc[15];

assign CrcNext[20] = Crc[16];

assign CrcNext[21] = Crc[17];

assign CrcNext[22] = (Enable & (Data[0] ^ Crc[28])) ^ Crc[18];

assign CrcNext[23] = (Enable & (Data[1] ^ Data[0] ^ Crc[29] ^
Crc[28])) ^ Crc[19];
```

```
assign CrcNext[24] = (Enable & (Data[2] ^ Data[1] ^ Crc[30] ^
Crc[29])) ^ Crc[20];

assign CrcNext[25] = (Enable & (Data[3] ^ Data[2] ^ Crc[31] ^
Crc[30])) ^ Crc[21];

assign CrcNext[26] = (Enable & (Data[3] ^ Data[0] ^ Crc[31] ^
Crc[28])) ^ Crc[22];

assign CrcNext[27] = (Enable & (Data[1] ^ Crc[29])) ^ Crc[23];

assign CrcNext[28] = (Enable & (Data[2] ^ Crc[30])) ^ Crc[24];

assign CrcNext[29] = (Enable & (Data[3] ^ Crc[31])) ^ Crc[25];

assign CrcNext[30] = Crc[26];

assign CrcNext[31] = Crc[27];



//CRC 值产生

always @ (posedge Clk or negedge Rst_n)

begin

  if (!Rst_n)

    Crc <= #1 32'hffffffff;

  else

  if(Initialize)

    Crc <= #Tp 32'hffffffff;

  else if(Enable)

    Crc <= #Tp CrcNext;

end



//将 CRC 计算结果转换成以太网协议适用的格式（每 4 位高低位对调并取反）

assign Crc_eth = ~{

               CrcNext[28], CrcNext[29], CrcNext[30], CrcNext[31],

               Crc[24], Crc[25], Crc[26], Crc[27],

               Crc[20], Crc[21], Crc[22], Crc[23],
```

```
                Crc[16], Crc[17], Crc[18], Crc[19],

                Crc[12], Crc[13], Crc[14], Crc[15],

                Crc[ 8], Crc[ 9], Crc[10], Crc[11],

                Crc[ 4], Crc[ 5], Crc[ 6], Crc[ 7],

                Crc[ 0], Crc[ 1], Crc[ 2], Crc[ 3]};



assign CrcError = Crc[31:0] != 32'hc704dd7b;  // CRC not equal to
magic number


endmodule
```

这里，由于 Crc_eth 默认是以 4 位形式组织的，在以太网帧发送单元里，为了匹配，修改之前的 CRC 字段发送时每个字节高低四位的先后顺序。下表左侧为上节实验，使用手动方式计算 CRC 结果时，CRC 以字节为单位，需要调换每个字节中高低四位顺序。下表右侧为本节实验中，使用自动 CRC 校验产生单元后，CRC 结果以 4 位为单位，因此无需再调换每个字节的高低四位顺序。

手动 CRC 值计算时 CRC 结果发送顺序	自动 CRC 值计算时 CRC 结果发送顺序
//发送 CRC 校验结果	//发送 CRC 校验结果
46: mii_tx_data <= #1 CRC_Result[27:24];	46: mii_tx_data <= #1 CRC_Result[31:28];
47: mii_tx_data <= #1 CRC_Result[31:28];	47: mii_tx_data <= #1 CRC_Result[27:24];
48: mii_tx_data <= #1 CRC_Result[19:16];	48: mii_tx_data <= #1 CRC_Result[23:20];
49: mii_tx_data <= #1 CRC_Result[23:20];	49: mii_tx_data <= #1 CRC_Result[19:16];
50: mii_tx_data <= #1 CRC_Result[11:8];	50: mii_tx_data <= #1 CRC_Result[15:12];
51: mii_tx_data <= #1 CRC_Result[15:12];	51: mii_tx_data <= #1 CRC_Result[11:8];
52: mii_tx_data <= #1 CRC_Result[3:0];	52: mii_tx_data <= #1 CRC_Result[7:4];
53: mii_tx_data <= #1 CRC_Result[7:4];	53: mii_tx_data <= #1 CRC_Result[3:0];

crc32_d4.v 为 4 位数据位宽的并行 32 位 CRC 计算单元

eth_send.v 为以太网 MAC 帧组包逻辑，完成以太网 MAC 帧的建立和发送。

Eth_send_test.v 为 ARP 包组包工具，并调用 eth_send 和 crc32_d4.v 完成完整的 ARP 报文的发送。

例化以太网帧发送模块，并将目的地址设置为广播地址 ffffffff、源地址为板卡地址。这里选择某 Xilinx 官板的 mac 地址，以避免与其他实际地址冲突。协议类型为 ARP（0806）。数据长度是整个发送的 arp 协议报文的长度字节数的两倍（MII 接口将一个字节拆成 2 个 4 位数分两次发送，因此数据长度为实际发送的 arp 报文长度的 2 倍）。

由于使用了自动 CRC 校验，因此用户可以手动修改源 MAC 地址，并使用以太网发包工具重新组建对应的 ARP 包，替换查找表中的值，然后发送数据。组包完成后无需再用 PC 端 CRC 计算工具计算 CRC 值。以下为 Eth_send_test.v 中部分代码内容：

```
    wire tx_go;


    wire fifo_rdreq;    //读发送缓存 fifo 请求
    reg [3:0] fifo_rddata; //发送缓存 fifo 数据输出
    wire fifo_rdclk;    //读发送缓存 fifo 时钟
    reg [11:0]data_cnt;


    wire [31:0]Crc_eth;
    wire CRC_EN;//CRC 校验单元校验使能信号


    eth_send eth_send(
        .rst_n(rst_n),
        .tx_go(tx_go),
        .data_length(12'd92),
        .des_mac(48'hff_ff_ff_ff_ff_ff),//目的网卡号
        .src_mac(48'h00_0a_35_01_fe_c0),//源（FPGA 板卡）网卡号
        .type_length(16'h08_06),    //协议类型 ARP 请求
```

```
        .CRC_Result(Crc_eth),       //发送数据的 CRC 实时校验值
        .CRC_EN(CRC_EN),              //CRC 校验单元校验使能信号
        .fifo_rdreq(fifo_rdreq),    //读发送缓存 fifo 请求
        .fifo_rddata(fifo_rddata),  //发送缓存 fifo 数据输出
        .fifo_rdclk(fifo_rdclk),    //读发送缓存 fifo 时钟
        .mii_tx_clk(mii_tx_clk),
        .mii_tx_en(mii_tx_en),
        .mii_tx_er(mii_tx_er),
        .mii_tx_data(mii_tx_data)
    );

    crc32_d4 crc32_d4(
        .Clk(mii_tx_clk),
        .Rst_n(rst_n),
        .Data(mii_tx_data),
        .Enable(CRC_EN),
        .Initialize(~mii_tx_en),
        .Crc(),
        .CrcError(),
        .Crc_eth(Crc_eth)
    );

    //计数发送数据个数，用于产生对应的数据
    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
```

```
        data_cnt <= #1 12'd0;
    else if(fifo_rdreq)
        data_cnt <= #1 data_cnt + 1'b1;
    else
        data_cnt <= #1 12'd0;

    //UDP 包
    always@(*)
    begin
        case(data_cnt)
            //hdwr type
            00: fifo_rddata =   4'h0;
            ……
            91: fifo_rddata =   4'h0;
            default:fifo_rddata = 4'h0;
        endcase
    end

    //发送间隔计数器
    reg [23:0]cnt;

    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
        cnt <= #1 0;
        else //计数器自增，不考虑溢出，接受溢出自动清零
```

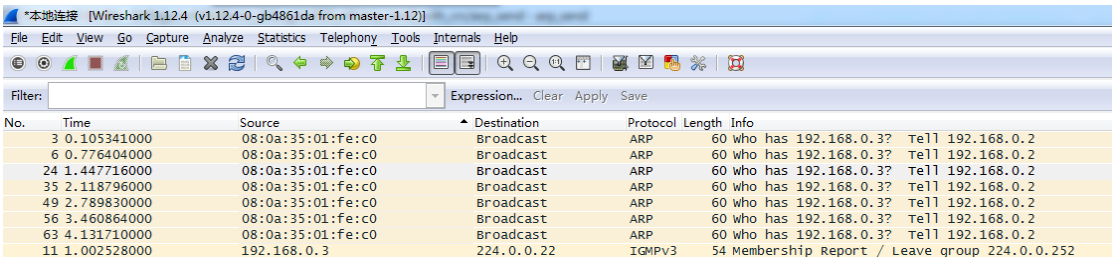
```
        cnt <= #1 cnt + 1'b1;



    //每 671ms（该值无特殊要求，这里取计数器自溢出，接近 0.5ms 发送一次数据的目的）启动一次发送数据

    assign tx_go = (cnt == 24'd1);
```

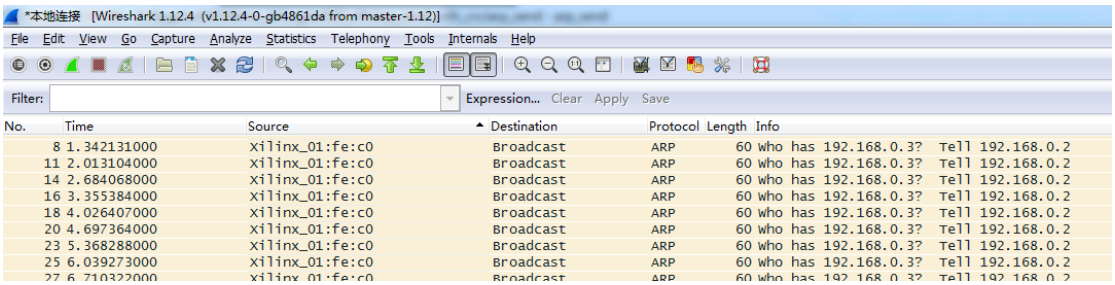
以下简单做个测试，分别使用 00-0a-35-01-fe-c0 和 08-0a-35-01-fe-c0 作为源 MAC 地址，发送 ARP 包，由于整个以太网帧中已经有数据内容发生了变化，因此 CRC 校验值不会相同，以此来检查是否不同的数据包 CRC 计算都能通过。修改时仅修改了以下内容：

```
47    eth_send eth_send(
48        .rst_n(rst_n),
49        .tx_go(tx_go),
50        .data_length(12'd92),
51        .des_mac(48'hff_ff_ff_ff_ff_ff),//目的网卡号
52    //    .src_mac(48'h00_0a_35_01_fe_c0),//源（FPGA板卡）网卡号
53        .src_mac(48'h08_0a_35_01_fe_c0),//源（FPGA板卡）网卡号
54        .type_length(16'h08_06),    //协议类型 ARP请求
55        .CRC_Result(Crc_eth)        //发送数据的CRC实时校验值
```

```
113                15: fifo_rddata = 4'h0;
114
115                //sender mac
116    //          16: fifo_rddata = 4'h0;
117                16: fifo_rddata = 4'h8;
118                17: fifo_rddata = 4'h0;
```

下图为分别设置源 MAC 为 00-0a-35-01-fe-c0 和 08-0a-35-01-fe-c0 时，PC 端以太网抓包工具抓到的数据。





可以看到，修改了数据报内容后整个发送模块也能自动得到对应的正确的 CRC 校验字并发送到 PC 端。

第 6 章 UDP 协议原理与 FPGA 实现

UDP 协议介绍

UDP 是 User Datagram Protocol 的简称， 中文名是用户数据报协议，是 OSI（Open System Interconnection，开放式系统互联） 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，IETF RFC 768 是 UDP 的正式规范。UDP 在 IP 报文的协议号是 17(即 0x11)。

UDP 协议全称是用户数据报协议[1] ，在网络中它与 TCP 协议一样用于处理数据包，是一种无连接的协议。在 OSI 模型中，在第四层——传输层，处于 IP 协议的上一层。UDP 有不提供数据包分组、组装和不能对数据包进行排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP 用来支持那些需要在计算机之间传输数据的网络应用。包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都需要使用 UDP 协议。UDP 协议从问世至今已经被使用了很多年，虽然其最初的光彩已经被一些类似协议所掩盖，但是即使是在今天 UDP 仍然不失为一项非常实用和可行的网络传输层协议。

与所熟知的 TCP（传输控制协议）协议一样，UDP 协议直接位于 IP（网际协议）协议的顶层。根据 OSI（开放系统互连）参考模型，UDP 和 TCP 都属于传输层协议。UDP 协议的主要作用是将网络数据流量压缩成数据包的形式。一个典型的数据包就是一个二进制数据的传输单位。每一个数据包的前 8 个字节用来包含报头信息，剩余字节则用来包含具体的传输数据。

UDP 数据报格式

UDP 协议使用端口号为不同的应用保留其各自的数据传输通道。UDP 和 TCP 协议正是采用这一机制实现对同一时刻内多项应用同时发送和接收数据的支持。数据发送一方（可以是客户端或服务器端）将 UDP 数据包通过源端口发送出去，而数据接收一方则通过目标端口接收数据。有的网络应用只能使用预先为其预留或注册的静态端口；而另外一些网络应用则可以使用未被注册的动态端口。因为 UDP 报头使用两个字节存放端口号，所以端口号的有效范围是从 0 到 65535。一般来说，大于 49151 的端口号都代表动态端口。

数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到 8192 字节。

UDP 协议使用报头中的校验值来保证数据的安全。校验值首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算值将不会相符，由此 UDP 协议可以检测是否出错。这与 TCP 协议是不同的，后者要求必须具有校验值。

许多链路层协议都提供错误检查，包括流行的以太网协议，也许你想知道为什么 UDP 也要提供检查和校验。其原因是链路层以下的协议在源端和终端之间的某些通道可能不提供错误检测。虽然 UDP 提供有错误检测，但检测到错误时，UDP 不做错误校正，只是简单地把损坏的消息段扔掉，或者给应用程序提供警告信息。

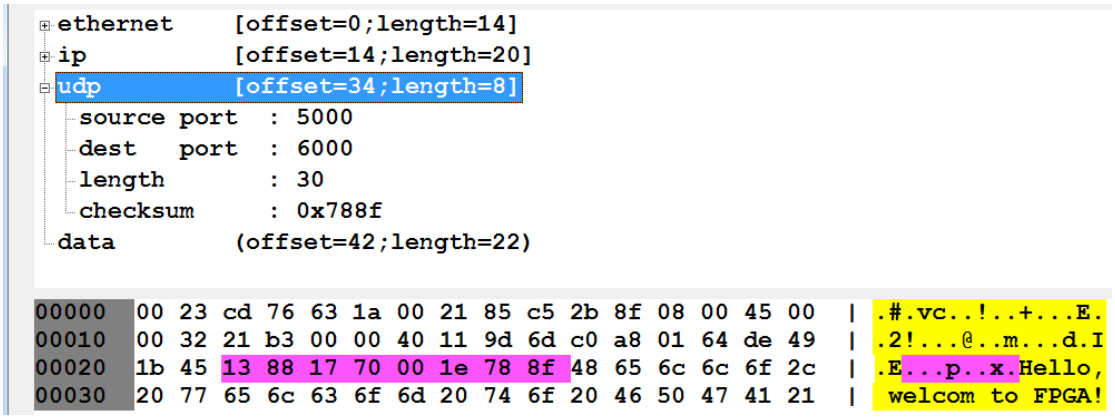
UDP 字段	源端口号	2	例如 5000	发送方的网络端口地址
	目的端口号	2	例如 6000	接收方的网络端口地址
	UDP 长度	2	UDP 报文长度	UDP 报文（含数据）长度

	UDP 校验和	2	UDP 报头校验和	UDP 协议的报头和数据的和校验
	UDB 数据	n		

由于每次需要发送的数据都不相同，而且校验和内容在发送数据段之前就需要计算出来，不像 MAC 层是在所有数据都发送完成之后才发送 CRC 校验值，因此在 UDP 组包时，校验和值的计算是一个不太好处理的地方，即使通过专用的计算单元计算出校验和，在待发送数据输入完成到数据开始通过 UDP 包发送之间，也有一个计算校验和的时间段。所幸，UDP 校验和在一些要求不太严格的地方，是可以忽略的，因此，在使用 Verilog 实现时，为了提升效率并节约 FPGA 资源，将校验和字段忽略。这样一来，UDP 数据包的组包就变得非常简单了。例如本机端口号为 5000(0x1388)，要给端口号为 6000(0x1770)的目标主机发送“Hello, welcom to FPGA!”，数据为 22 个字节，UDP 首部 8 个字节，合计 30 个字节，则 UDP 数据段可以组包为：

0x13	0x88	0x17	0x70
0x00	0x1e	0x00	0x00
‘H’(0x48)	‘e’(0x65)	‘l’(0x6c)	‘l’(0x6c)
‘o’(0x6f)	‘,’(0x2c)	‘ ’(0x20)	‘w’(0x77)
‘e’(0x65)	‘l’(0x6c)	‘c’(0x63)	‘o’(0x6f)
‘m’(0x6d)	‘ ’(0x20)	‘t’(0x74)	‘o’(0x6f)
‘ ’(0x20)	‘F’(0x46)	‘P’(0x50)	‘G’(0x47)
‘A’(0x41)	‘!’(0x21)		

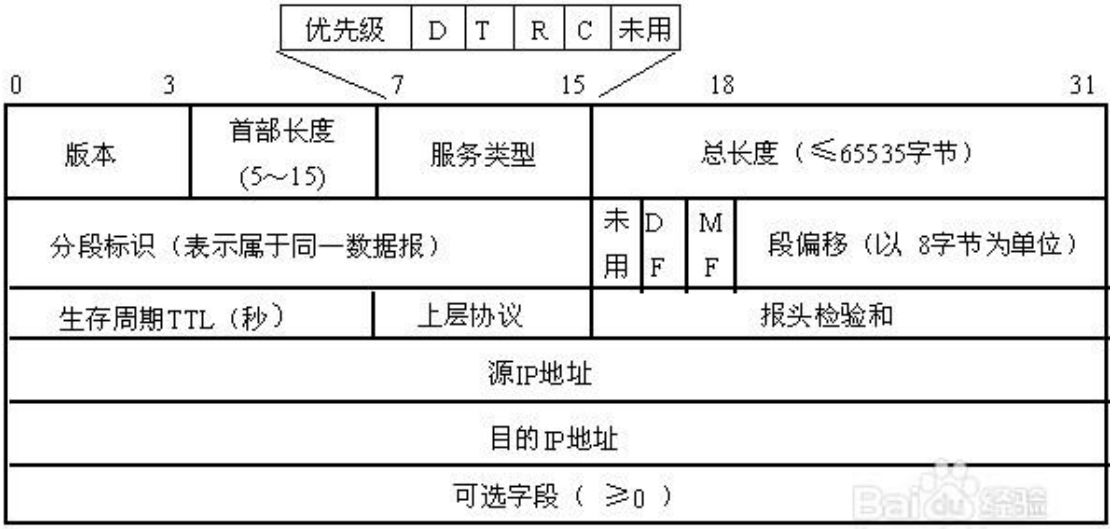
以下为通过以太网组包工具组包出来的该报文 UDP 部分内容：



IP 协议介绍

IP 是 TCP/IP 协议族中最核心的协议，所有的 TCP、UDP、ICMP、IGMP 数据都以 IP 数据报的格式传输。IP 仅提供尽力而为的传输服务，如果发生某种错误，IP 会丢失该数据，然后发送 ICMP 消息给信源端。另外，IP 数据报可以不按发送顺序接受

IP 数据报的格式如下：



DF：是否分段； MF：是否有后续分段

前 20 字节和紧接其后的选项部分是 IP 数据报的首部，前 20 个字节是固定的，选项可有可无。首部的每一行是一个 32 位字的单位，最高位在左边，为 0bit，最低位在右边，为 31bit。4 字节的 32bit 值按照以下次序传输：首先 0-7bit，其次 8-15 比特，然后 16-23bit，最后是 24-31bit，这种传输次序称为 big endian 字节序(我们在 C 语言写位操作的算法时常用到该词)。TCP/IP 首部中的所有二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序，其他形式存储的二进制数据，如 little endian 格式，则必须在传输数据之前把首部转化成网络字节序。

首部长度是指首部占 32bit 字的数目，因为 4 位的最大值为 15，因此首部最长为 60 字节，也即是说选项部分的最大值为 40 字节，不够 4 的倍数，要用 0 填充，使数据部分的起始地址为 4 的倍数。

总长度指整个 IP 数据报的长度，包括首部和数据部分，16bit，最长可达 65535 字节。尽管理论上可以传送一个长达 65535 的 IP 数据报，但实际上还要考虑网络的最大承载能力等因素。

3 个标志位主要用来标识分片的 IP 数据报，片位移为分片的数据报的首个字节偏离整个原始数据报的位置。

IP 字段	版本+首部长度	1	IP 协议版本 IP 首部长度	版本就是 IPV4 或者 IPV6，一般选择 IPV4，即版本值为 4。 IP 首部长度即有多少个 32 位数，当 IP 首部长度为 20 时（即无可选字段），该值为 5.（5*4 =20）
	服务类型	1	指示了报文的优先权等	默认全部置 0 即可
	总长度	2	IP 报文长度	IP 报文（报头+数据）长度
	分段标识	2	是否属于同一数据段	IP 报文的分段 ID
	段标识和段偏移	2	可设为 0	
	生存周期（TTL）	1	可以经过的最大路由数	生存时间字段设置了数据报可以经过的最多路由器数,表示数据包在网络上生存多久。TTL 的初始值由源主机设置(通常为 32 或 64),一旦经过一个处理它的路由器,它的值就减去 1。当该字段的值为 0 时,数据报就被丢弃,并发送 ICMP 消息通知源主机。这样当封包在传递过程中由於某些原因而未能抵达目的地的时候就可以避免其一直充斥在网路上面。占 8 位。
	上层协议类型	1	指该封包所使用的网络协议类型,如 ICMP、DNS 等。占 8 位。	常用协议号： 00：IP 01：ICMP 06：TCP 17：UDP
	报头校验和	2	IP 报头的校验和	
	源 IP 地址	4	发送端的 IP 地址	如 192.168.0.2
	目的 IP 地址	4	接收端的 IP 地址	如 192.168.0.3
	可选字段		可选	没有的时候长度可以为 0

协议号

协议号	协议	协议号	协议
00	IP	22	XNS-IDP
01	ICMP	27	RDP
02	IGMP	29	ISO-TP4
03	GGP	36	XTP
04	IP-ENCAP	37	DDP

05	ST	39	IDPR-CMTP
06	TCP	73	RSPF
08	EGP	81	VMTP
12	PUP	89	OSPFIGP
17	UDP	94	IPIP
20	HMP	98	ENCAP

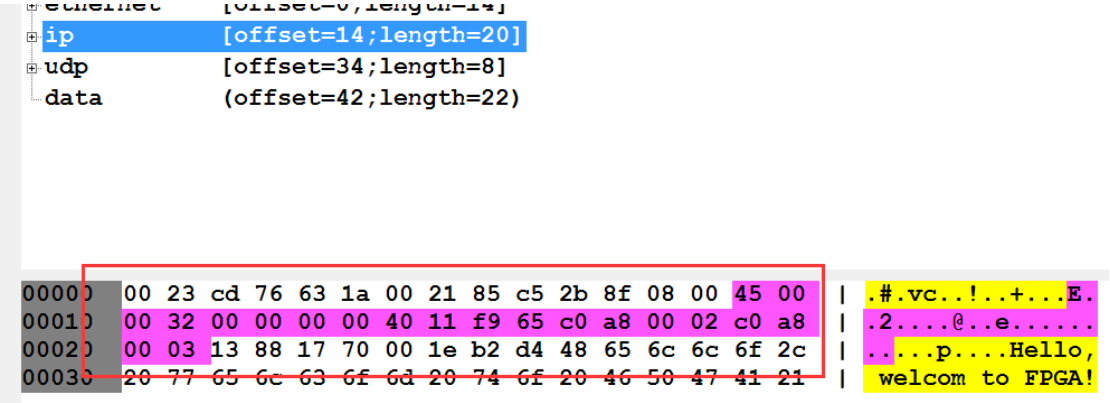
指 IPv4 数据报包头的校验和。这个数值用来检错用的,用以确保封包被正确无误的接收到。当封包开始进行传送后,接收端主机会利用这个检验值会来检验余下的封包,如果一切无误就会发出确认信息表示接收正常。与 UDP 和 TCP 协议包头中的校验和作用是一样的。占 16 位。

首部检验和字段是根据 IP 首部计算的检验和码,不对首部后面的数据进行计算。其计算方法为：

在发送数据时，为了计算数 IP 据报的校验和。应该按如下步骤：

- 将校验和字段置为 0,然后将 IP 包头按 16 比特分成多个单元，如包头长度不是 16 比特的倍数，则用 0 比特填充到 16 比特的倍数；
- 对各个单元采用反码加法运算(即高位溢出位会加到低位,通常的补码运算是直接丢掉溢出的高位),将得到的和的反码填入校验和字段；

例如，我们使用 IP 协议发送一个数据长度为 30 个字节的数据包，发送端 IP 为 192.168.0.2，接收端 IP 为 192.168.0.3。则 IP 报头可如下所示：



协议类型 报头长度	服务类型 tos	IP 报文长度 total_len		分段标识 id		分段和偏移 offset	
0x45	0x00	0x00	0x32	0x00	0x00	0x00	0x00
v4 20B	普通	50 字节					
生存周期 ttl	上层协议 protocol	报头校验和 checksum		源地址 src_ip			
0x40	0x11	0xf9	0x65	0xc0	0xa8	0x00	0x02
64 级	17 UDP			192	168	0	2
目标地址 dst_ip							
0xc0	0xa8	0x00	0x03				
192	168	0	3				

按照上述提到的 IP 首部校验和的方法计算 IP 首部校验和，即：

1. 0x4500 + 0x0032 + 0x0000 + 0x0000 + 0x4011 + 0x0000(计算时强制置 0) + 0xc0a8 + 0x0002 + 0xc0a8 + 0x0003 = 0x20698。

2. 0x0002 + 0x0698 = 0x069a

3. checksum = ~0x069a = 0xf965

计算结果与实际发包软件计算的一致。

该部分使用 Verilog 编写计算单元非常的方便，可以用纯组合逻辑实现，也可以用时序逻辑实现。

```
module checksum
```

```
(
    input [3:0]ver, //版本
    input [3:0]hdr_len,//首部长度
    input [7:0]tos, //服务类型
    input [15:0]total_len,  //IP 报文总长
    input [15:0]id, //分段标识
    input [15:0]offset, //偏移
    input [7:0]ttl, //生存周期
    input [7:0]protocol, //上层协议类型
    input [31:0]src_ip, //源 IP 地址
    input [31:0]dst_ip, //目的 IP 地址

    output [15:0]checksum_result //校验和
);

    wire [31:0]sum;

    assign sum = {ver,hdr_len,tos} + total_len + id
                    + offset + {ttl,protocol} + src_ip[31:16]
                    + src_ip[15:0] + dst_ip[31:16] + dst_ip[15:0];

    assign checksum_result = ~(sum[31:16] + sum[15:0]);

endmodule
```

测试脚本如下所示：

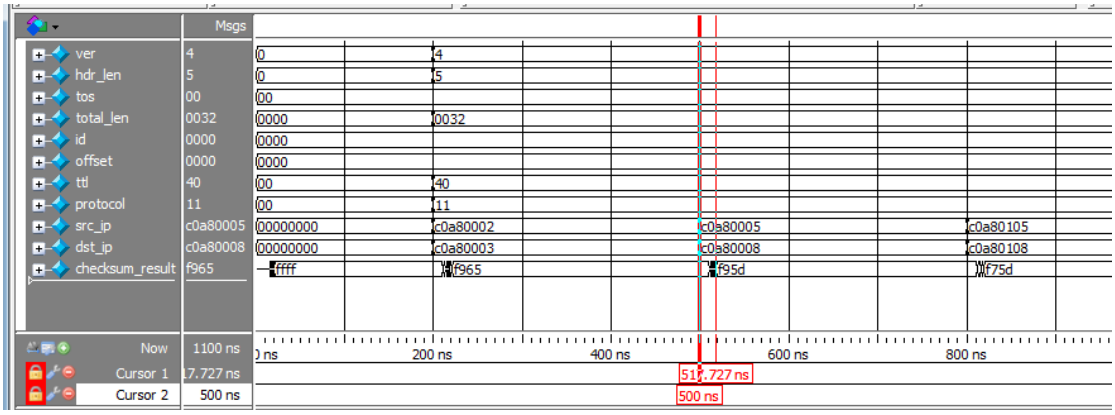
```
`timescale 1ns/1ns
```

```
module checksum_tb;


    reg [3:0]ver;
    reg [3:0]hdr_len;
    reg [7:0]tos;
    reg [15:0]total_len;
    reg [15:0]id;
    reg [15:0]offset;
    reg [7:0]ttl;
    reg [7:0]protocol;
    reg [31:0]src_ip;
    reg [31:0]dst_ip;


    wire [15:0]checksum_result;


    checksum checksum
    (
        .ver(ver),
        .hdr_len(hdr_len),
        .tos(tos),
        .total_len(total_len),
        .id(id),
        .offset(offset),
        .ttl(ttl),
        .protocol(protocol),
        .src_ip(src_ip),
```

```
        .dst_ip(dst_ip),
        .checksum_result(checksum_result)
    );


    initial begin
        ver = 0;
        hdr_len = 0;
        tos = 0;
        total_len = 0;
        id = 0;
        offset = 0;
        ttl = 0;
        protocol = 0;
        src_ip = 0;
        dst_ip = 0;
        #200;


        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80002;
        dst_ip = 32'hc0a80003;
```

```
        #300;


        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80005;
        dst_ip = 32'hc0a80008;
        #300;


        ver = 4'h4;
        hdr_len = 4'h5;
        tos = 8'h0;
        total_len = 16'h0032;
        id = 16'h0000;
        offset = 16'h0000;
        ttl = 8'h40;
        protocol = 8'h11;
        src_ip = 32'hc0a80105;
        dst_ip = 32'hc0a80108;
        #300;
        $stop;
    end
```

```
endmodule
```



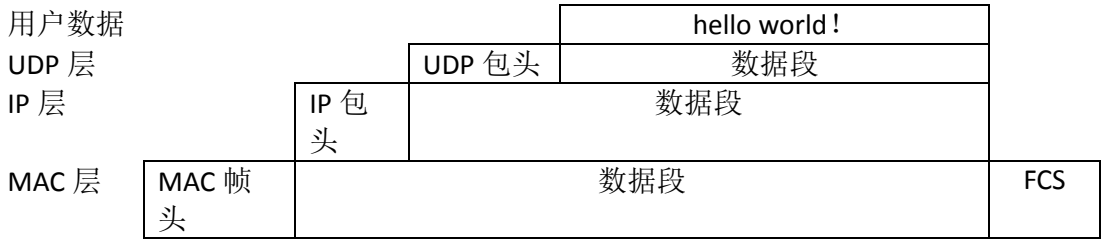
使用 Modelsim 进行时序仿真，可知从输入数据变化到输出结果稳定，至少需要 17.7ns。从 IP 首部所有数据确定，到开始发送校验和字段，百兆以太网 MII 接口是中间最起码有 20 个时钟周期（20*20ns = 400ns），千兆以太网 GMII 接口是 10 个时钟周期 (10*8ns = 80ns)，即完全满足时序要求。

第 7 章 以太网 UDP 帧发包实例（手动 IPchecksum）

设计实例： UDP 数据报发送实验

关于用户数据、UDP、IP、MAC 四个报文的关系：

用户数据是打包在 UDP 协议中、UDP 协议又是基于 IP 协议之上的，IP 协议又是走 MAC 层发送的，即从包含关系来说：MAC 帧中的数据段为 IP 数据报，IP 报文中的数据段为 UDP 报文，UDP 报文中的数据段为用户希望传输的数据内容，如“hello world！”。下图为使用 UDP 协议发送“hello world！”的数据层层打包示意图：

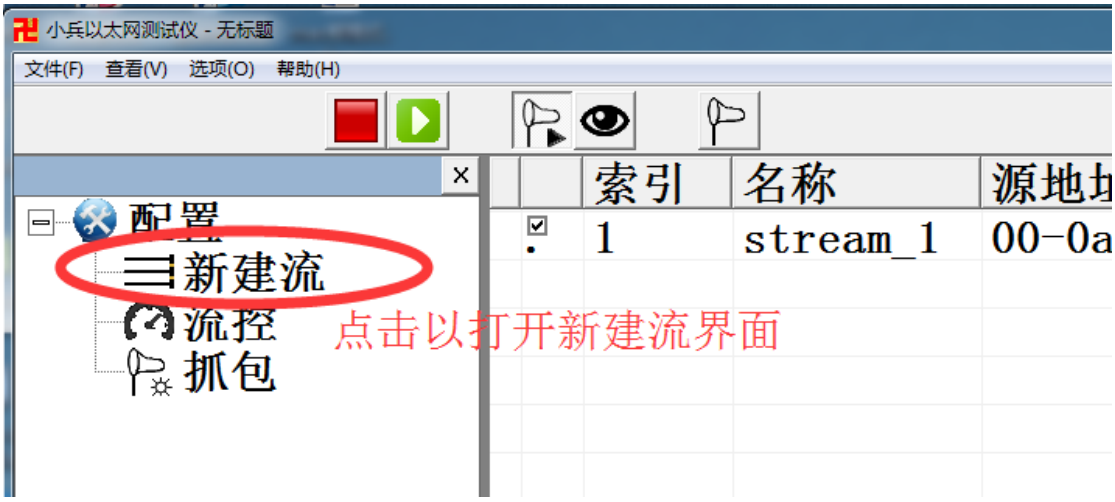


即使用 UDP 协议发送“hello world！”，按照以下顺序进行：

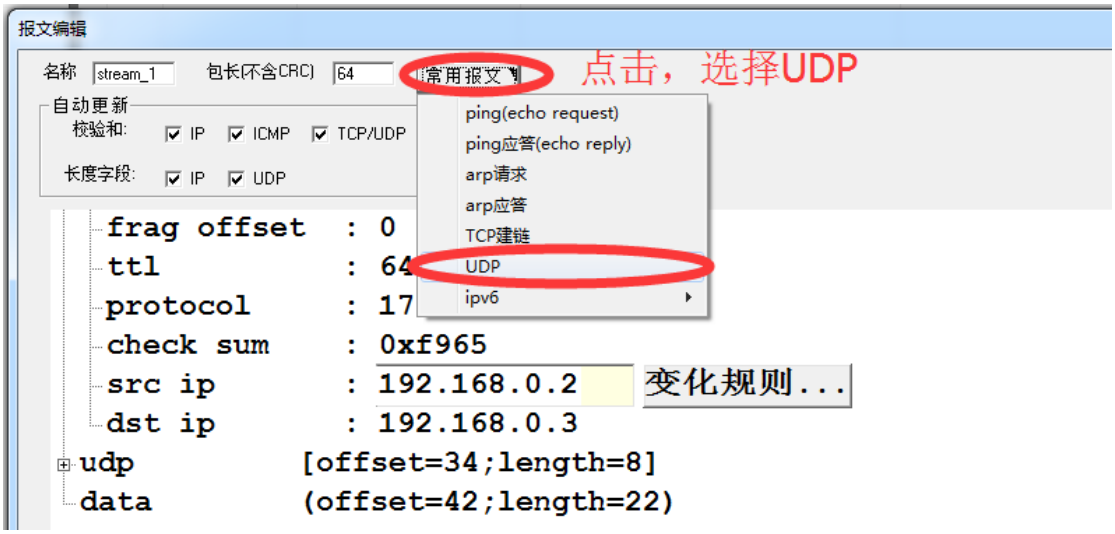
1. 将“hello world！”内容打包进 UDP 数据报
2. 将 UDP 协议打包成 IP 数据报
3. 将 IP 数据报送入以太网帧发送单元（MAC 层），组织成标准 MAC 帧后发送。

这里，我们将要发送的数据定义为“Hello, welcom to FPGA!”

本实验我们使用以太网发包工具组件一个 UDP 包，然后将该包使用我们上一节设计的带 CRC 校验的 ARP 发送工程来发送出去。设计时，只需替换查找表中的内容即可。本设计实例提供的工程名称为 *udp_send.qar*

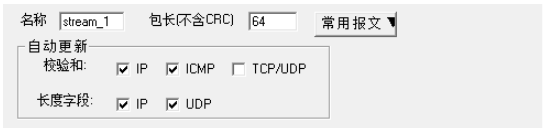


在常用报文中选择 UDP 格式报文



修改以太网包和 arp 包中各字段内容如下所示

数据帧内容：UDP




```
ethernet     [offset=0;length=14]
  dst mac       : 00-23-cd-76-63-1a
  src mac       : 00-21-85-c5-2b-8f
  eth type      : 0x0800
ip             [offset=14;length=20]
  ver           : 4
  hdr_len       : 5
  tos           : 0
  total len     : 50
  id            : 0
  rsv           : 0
  df            : 0
  mf            : 0
  frag offset   : 0
  ttl           : 64
  protocol      : 17
  check sum     : 0xf965
  src ip        : 192.168.0.2
  dst ip        : 192.168.0.3
udp            [offset=34;length=8]
  source port   : 5000
  dest   port   : 6000
  length        : 30
  checksum      : 0xb2d4
  data          (offset=42;length=22)
```

MAC目的地址任意，该部分在以太网帧发送代码中确定
MAC源地址任意，该部分在以太网帧发送代码中确定
以太网类型为IP协议

IPv4
IP首部长度为5个32位数据，即20字节
普通
IP报总长度为20+30合计50个字节
分段标识，填0即可

包生存时间：64级路由
上层协议类型：UDP
校验和，由工具自动计算得出
源IP地址
目的IP地址

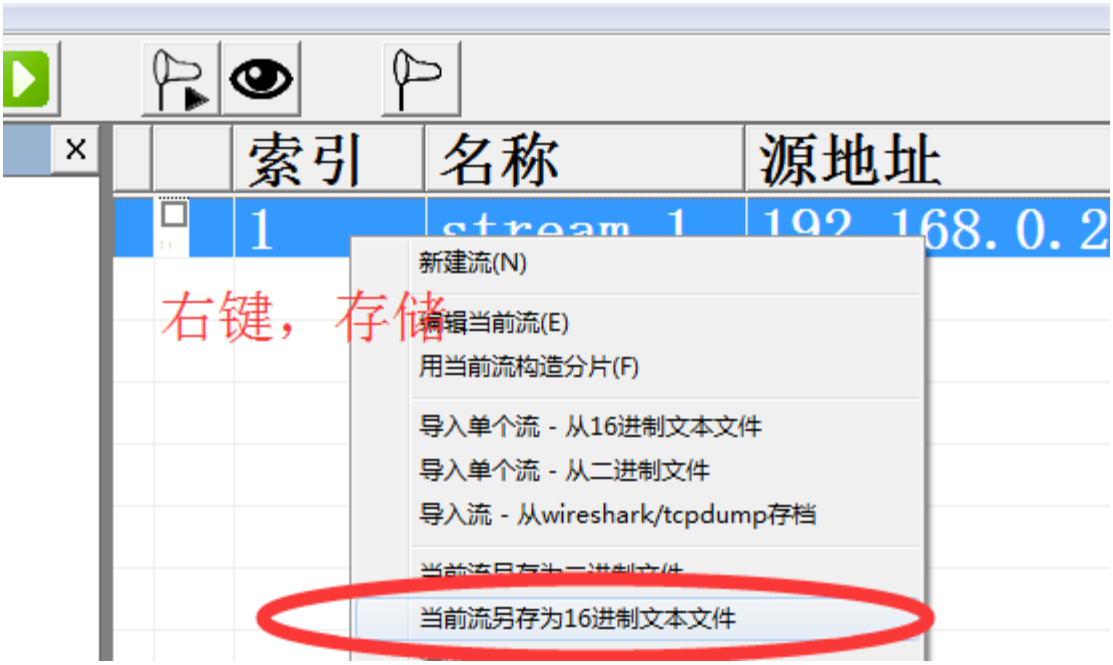
UDP源端口号：5000　，用户可自行修改
UDP目的端口号：6000，用户可自行修改
UDP数据包长度8+22 = 30字节

数据段：Hello, welcom to FPGA!

```
00000  00 23 cd 76 63 1a 00 21 85 c5 2b 8f 08 00 45 00  | .#.vc..!..+...E.
00010  00 32 00 00 00 00 40 11 f9 65 c0 a8 00 02 c0 a8  | .2....@..e......
00020  00 03 13 88 17 70 00 1e b2 d4 48 65 6c 6c 6f 2c  | .....p....Hello,
00030  20 77 65 6c 63 6f 6d 20 74 6f 20 46 50 47 41 21  |  welcom to FPGA!
```

配置好之后点击确定。

将设置好的数据包导出为 16 进制格式文本文件



得到的文本中数据内容如下所示：

0x00, 0x23, 0xcd, 0x76, 0x63, 0x1a, 0x00, 0x21, 0x85, 0xc5, 0x2b, 0x8f, 0x08, 0x00, 0x45, 0x00,

0x00, 0x32, 0x00, 0x00, 0x00, 0x00, 0x40, 0x11, 0xf9, 0x65, 0xc0, 0xa8, 0x00, 0x02, 0xc0, 0xa8,

0x00, 0x03, 0x13, 0x88, 0x17, 0x70, 0x00, 0x1e, 0xb2, 0xd4, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c,

0x20, 0x77, 0x65, 0x6c, 0x63, 0x6f, 0x6d, 0x20, 0x74, 0x6f, 0x20, 0x46, 0x50, 0x47, 0x41, 0x21。

其中我们只需要 IP 层以及 UDP 层，MAC 层的数据直接丢弃，因为我们的以太网帧发送逻辑会自动组建 MAC 数据。所以我们只需要保留深色背景的内容即可。

例化以太网帧发送模块，并将目的地址设置为用户的电脑 MAC 地址，如实验运行的 PC 机网卡 MAC 地址为：`48'h84_7b_eb_48_94_13`、源地址为板卡地址。这里选择某 Xilinx 官板的 mac 地址，以避免与其他实际地址冲突，地址为 `48'h00_0a_35_01_fe_c0`。协议类型为 IP（0800）。数据长度是整个发送的 IP 协议报文的长度字节数的两倍（MII 接口将一个字节拆成 2 个 4 位数分两次发送，因此数据长度为实际发送的 IP 报文长度的 2 倍），这里为 100。

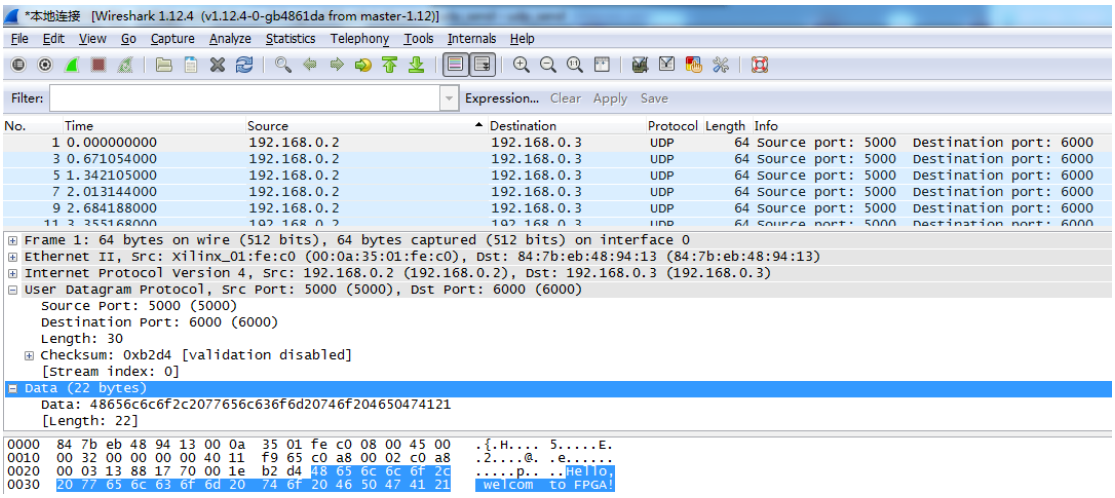
由于使用了自动 CRC 校验，因此用户可以手动修改源 MAC 地址，并使用以太网发包工具重新组建对应的 ARP 包，替换查找表中的值，然后发送数据。组包完成后无需再用 PC 端 CRC 计算工具计算 CRC 值。以下为 Eth_send_test.v 中部分代码内容：

```
    wire tx_go;


    wire fifo_rdreq;    //读发送缓存 fifo 请求
    reg [3:0] fifo_rddata; //发送缓存 fifo 数据输出
    wire fifo_rdclk;    //读发送缓存 fifo 时钟
    reg [11:0]data_cnt;


    wire [31:0]Crc_eth;
    wire CRC_EN;//CRC 校验单元校验使能信号


    eth_send eth_send(
        .rst_n(rst_n),
        .tx_go(tx_go),
        .data_length(12'd100),
        .des_mac(48'h84_7b_eb_48_94_13),//目的网卡号
        .src_mac(48'h00_0a_35_01_fe_c0),//源（FPGA 板卡）网卡号
        .type_length(16'h08_00),    //协议类型 IP
        .CRC_Result(Crc_eth),       //发送数据的 CRC 实时校验值
        .CRC_EN(CRC_EN),              //CRC 校验单元校验使能信号
        .fifo_rdreq(fifo_rdreq),    //读发送缓存 fifo 请求
        .fifo_rddata(fifo_rddata),  //发送缓存 fifo 数据输出
        .fifo_rdclk(fifo_rdclk),    //读发送缓存 fifo 时钟
        .mii_tx_clk(mii_tx_clk),
        .mii_tx_en(mii_tx_en),
        .mii_tx_er(mii_tx_er),
```

```
        .mii_tx_data(mii_tx_data)
    );


    crc32_d4 crc32_d4(
        .Clk(mii_tx_clk),
        .Rst_n(rst_n),
        .Data(mii_tx_data),
        .Enable(CRC_EN),
        .Initialize(~mii_tx_en),
        .Crc(),
        .CrcError(),
        .Crc_eth(Crc_eth)
    );


    //计数发送数据个数，用于产生对应的数据
    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
        data_cnt <= #1 12'd0;
    else if(fifo_rdreq)
        data_cnt <= #1 data_cnt + 1'b1;
    else
        data_cnt <= #1 12'd0;


    //UDP 包
    always@(*)
    begin
        case(data_cnt)
```

```
            0   : fifo_rddata = 4'h5;//首部长度
            1   : fifo_rddata = 4'h4;//协议版本
            ……
            99  : fifo_rddata = 4'h2;
            default:fifo_rddata = 4'h0;
        endcase
    end

    //发送间隔计数器
    reg [23:0]cnt;

    always@(posedge mii_tx_clk or negedge rst_n)
    if(!rst_n)
        cnt <= #1 0;
        else //计数器自增，不考虑溢出，接受溢出自动清零
        cnt <= #1 cnt + 1'b1;

    //每 671ms（该值无特殊要求，这里取计数器自溢出，接近 0.5ms 发送一次数据的目的）启动一次发送数据
    assign tx_go = (cnt == 24'd1);

endmodule
```

修改完成后全编译工程，烧写到 AC620FPGA 开发板上，打开 wireshark 工具，对本地连接进行抓包，抓取到的数据如下所示：



可以看到，IP 源地址为 192.168.0.2、目标地址为 192.168.0.3、协议类型为 UDP、源端口号为 5000、目的端口号为 6000。数据内容为：”Hello, welcome to FPGA!”。

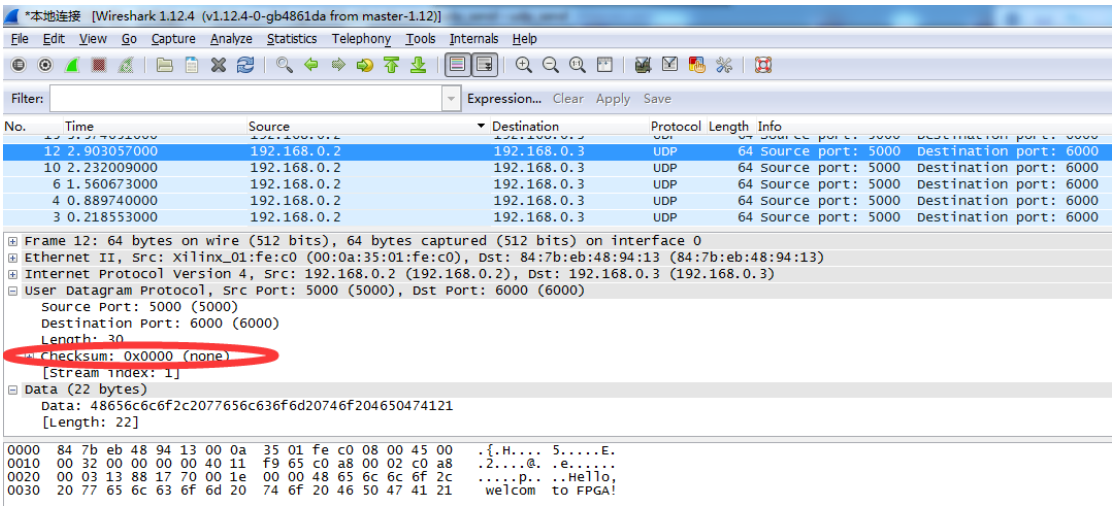
打开网络调试工具（可能需要暂时先关闭其他网卡，方便该工具自动选择有线网卡为默认网卡），设置协议类型为 UDP，本地 IP 地址为 192.168.0.3，本地端口号为 6000，勾选自动换行显示，然后点击链接。可以看到，网络数据接收窗口大概每 700ms 接收到一次数据，数据内容为”Hello, welcome to FPGA!”。

接下来我们测试 UDP 协议中忽略校验和

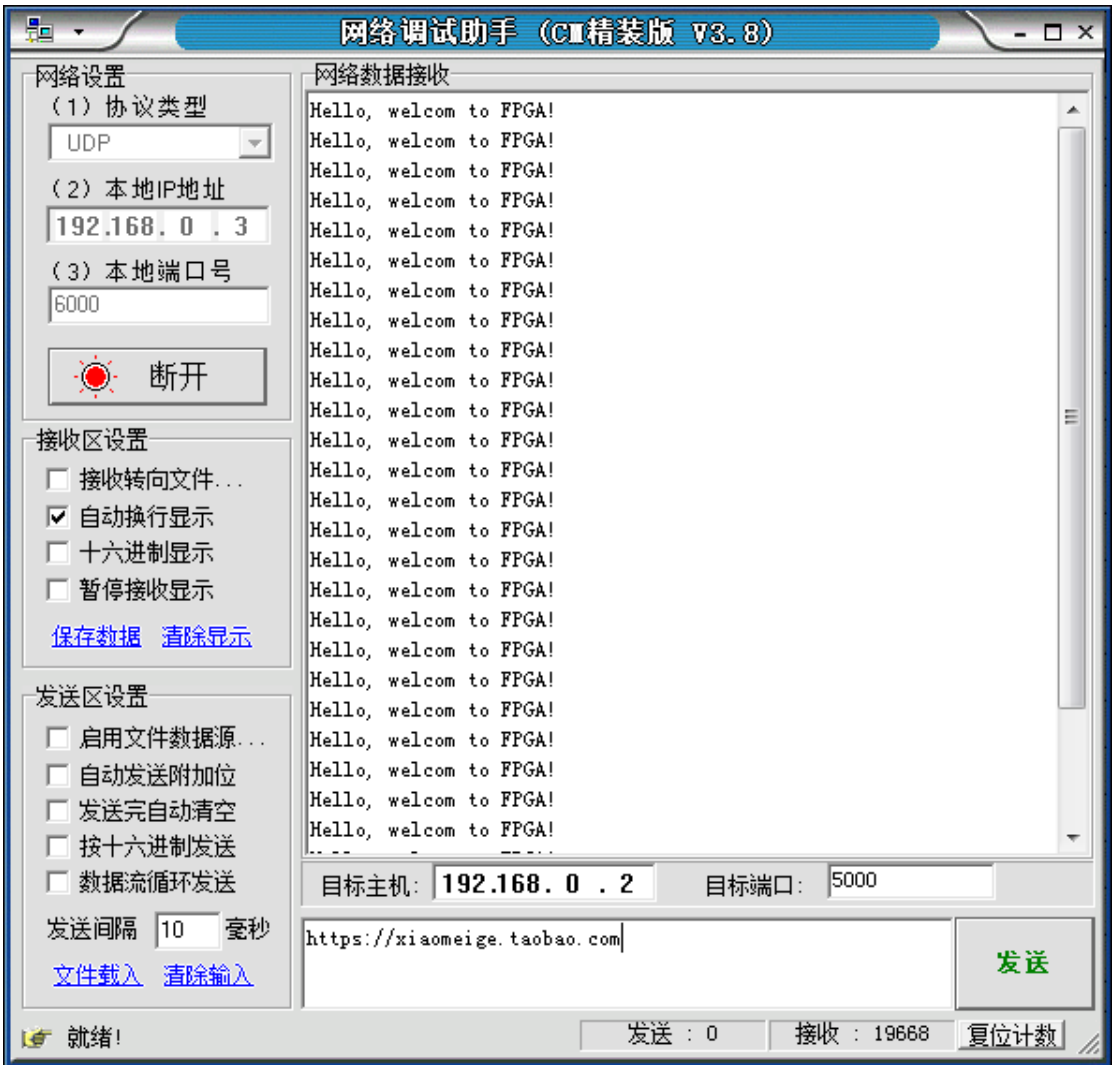
我们在代码中将 UDP 校验和部分数据修改为 0x0000，然后重新编译并烧写 AC620 开发板。

```
163             50  : fifo_rddata = 4'he;
164             51  : fifo_rddata = 4'h1;
165
166     //      //UDP报头校验和
167     //      52  : fifo_rddata = 4'h2;
168     //      53  : fifo_rddata = 4'hb;
169     //      54  : fifo_rddata = 4'h4;
170     //      55  : fifo_rddata = 4'hd;
171
172             //UDP报头校验和  忽略
173             52  : fifo_rddata = 4'h0;
174             53  : fifo_rddata = 4'h0;
175             54  : fifo_rddata = 4'h0;
176             55  : fifo_rddata = 4'h0;
```

在 wireshark 中抓取数据包，可以看到，依旧能够正常抓取到数据，在 UDP 的 checksum 一栏，显示的状态为 none，即无 UDP 校验和。



在网络调试工具中，依然可以正确的接收到数据。

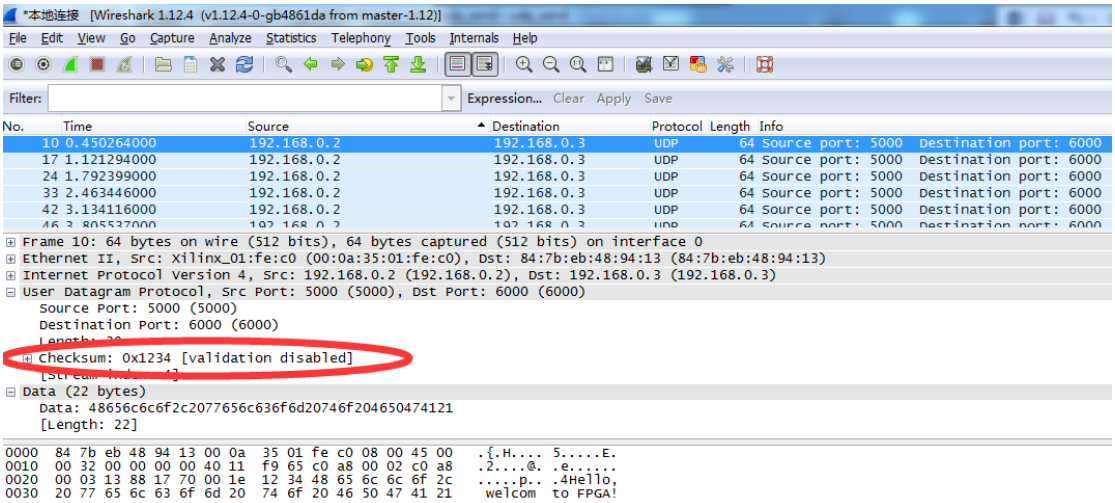


接下来测试 UDP 校验和错误的情况。

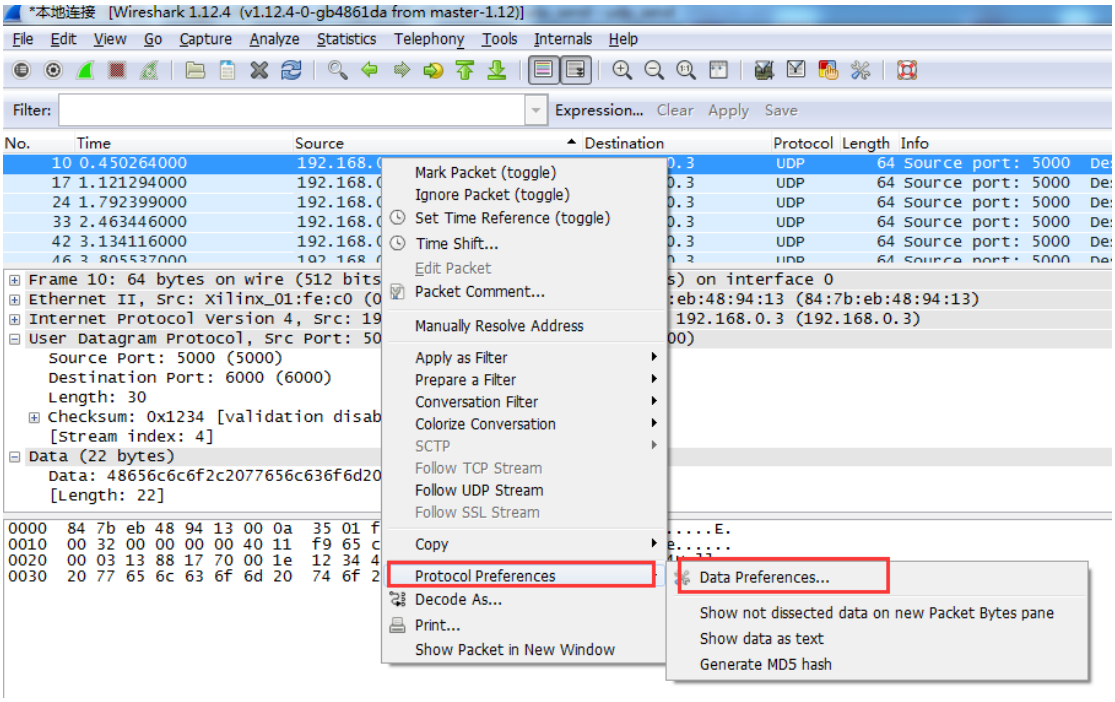
我们在代码中将 UDP 校验和部分数据修改为非 0 但又非争取的值，如 0x1234，然后重新编译并烧写 AC620 开发板。

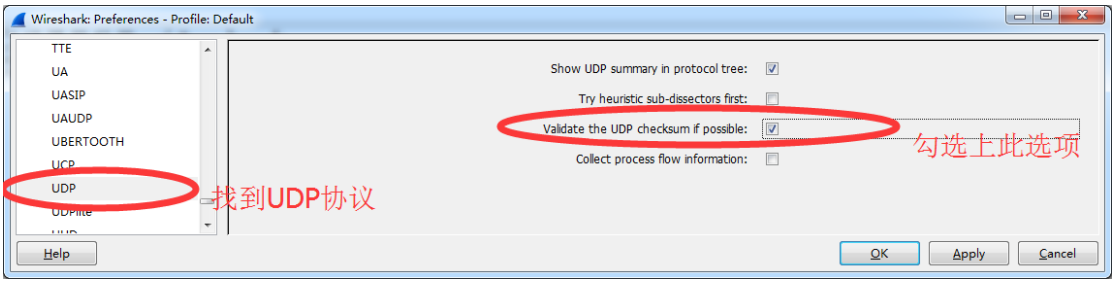
```
165
166     //        //UDP报头校验和
167     //        52  : fifo_rddata = 4'h2;
168     //        53  : fifo_rddata = 4'hb;
169     //        54  : fifo_rddata = 4'h4;
170     //        55  : fifo_rddata = 4'hd;
171
172     //        //UDP报头校验和  忽略
173     //        52  : fifo_rddata = 4'h0;
174     //        53  : fifo_rddata = 4'h0;
175     //        54  : fifo_rddata = 4'h0;
176     //        55  : fifo_rddata = 4'h0;
177
178             //UDP报头校验和  错误校验和
179             52  : fifo_rddata = 4'h2;
180             53  : fifo_rddata = 4'h1;
181             54  : fifo_rddata = 4'h4;
182             55  : fifo_rddata = 4'h3;
183
```

在 wireshark 中抓取数据包，可以看到，依旧能够正常抓取到数据，在 UDP 的 checksum 一栏，显示的状态为 validation disabled，即软件关闭了 UDP 校验和的验证。

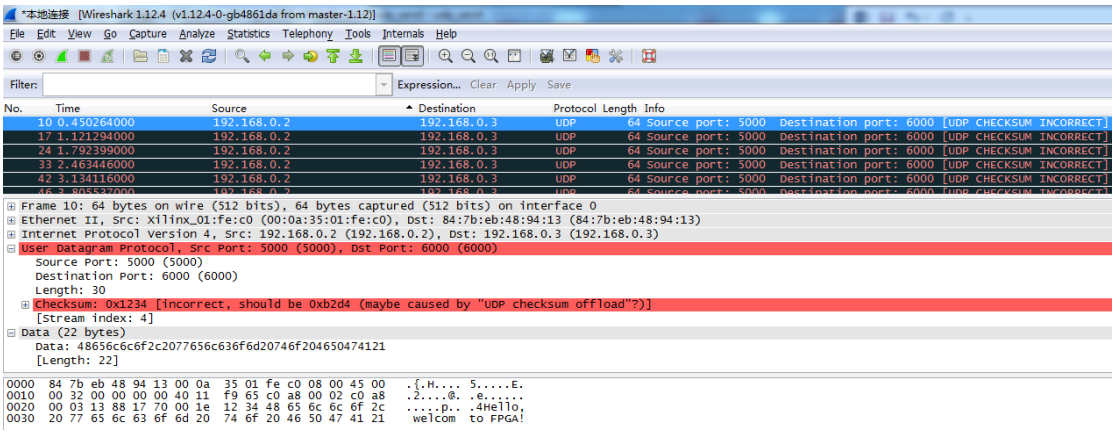


我们可以手动打开 wireshark 的 UDP 校验和验证，方法为选中任意一个抓取到的 UDP 数据包，右键，选择【Protocol Preferences】-> 【Data Preferences】，左侧找到 UDP 协议，勾选上 Validate the UDP checksum if possible。然后点击 apply，再 OK。





可以看到，所有 UDP 数据包的颜色变成了深黑色，并在下方提示 checksum 错误，并给出了正确的值应该为 0xb2d4。



回到以太网调试工具中，此时，以太网工具也无法收到数据了。

总结，UDP 报文的 checksum 字段可以为正确，也可以忽略（0x0000），但是不能为错，为错误一些标准的以太网工具将会将这样的数据包直接丢弃。

第 8 章 以太网 UDP 帧发包实例（自动 IPchecksum）

在上一个实验中，我们使用了以太网组包工具组建了 UDP 数据包并发送到 PC 端。并测试了 UDP 校验和正确、忽略和错误时网络抓包工具 wireshark 的处理以及常用的网络调试工具对待这三种情况的处理，可以知道，UDP 校验和正确或者忽略（0x0000），网络调试工具都能够收到数据。而 UDP 校验和错误时，则无法收到。在这个实例中，我们没有关心 IP 首部校验和的正确与否，默认使用发包工具计算提供的值，该值是正确的。但是这种方式只适用于 IP 首部中的所有值都是确定的情况下，如确定的目标 IP 和源 IP、确定的数据长度。那么当我们需要每次发送的数据长度都可能不同，或者 IP 地址不同时，就需要 FPGA 来自动生成 IP 校验和了。关于 IP 首部校验和的计算方法和计算 Verilog 代码，我们在介绍 IP 协议内容时就已经给出，本实验将使用这个代码，对 IP 首部进行实时自动校验。本设计实例提供的工程名称为 *udp_send_with_checksum.qar*。

使用时非常简单，将该模块代码例化到 eth_send_test 模块中，然后连接对应端口即可，如下所示：

```
    checksum checksum(

        .ver(4'h4),

        .hdr_len(4'h5),

        .tos(8'h0),

        .total_len(ip_total_len),

        .id(16'h0),

        .offset(16'h0),

        .ttl(8'h40),

        .protocol(8'h11),

        .src_ip(src_ip),

        .dst_ip(dst_ip),
```

```
        .checksum_result(ip_checksum)

    );
```

由于 IP 报文中，报头很多参数在实际使用时都是固定的，因此在模块例化时，就直接将这些参数固定，例如 IP 版本（ver），首部长度（hdr_len）、tos、id、offset、ttl、上层协议类型（protocol）。最终可能每次发送数据时需要变化的，也就是数据长度（ip_total_len）、源 IP 地址（src_ip）、目的 IP 地址（dst_ip）。Ip_checksum 为实时校验和的值。

由于每次发送时，以太网帧（即 MAC 层）所需传输的数据个数是 2 倍的 IP 总报文长度，所以直接将 ip_total_len 的值左移 1 位（乘以 2），来作为 MAC 层数据长度的个数。在 eth_send 模块例化时，直接将 ip_total_len 的值左移 1 位赋给 data_length。

```
    eth_send eth_send(

        .rst_n(rst_n),

        .tx_go(tx_go),

        .data_length(ip_total_len<<1),

        .des_mac(48'h84_7b_eb_48_94_13),//目的网卡号
```

目的 IP 地址和源 IP 地址一般在某个具体应用中是固定的，因此，直接使用一个定值，当然，也可以通过寄存器的方式，使用其他控制逻辑来修改。在具体应用时，可以通过修改这个值来确定系统的 IP 地址。

```
    wire [31:0]src_ip;

    wire [31:0]dst_ip;



    assign src_ip = 32'hc0_a8_00_02;

    assign dst_ip = 32'hc0_a8_00_03;
```

每次发送数据时，用户数据长度定义为 data_total_len，该值在每次发送数据前确定。

每个 UDP 报文的长度为用户数据长度（data_total_len）+UDP 报头长度（8）代码如下所示：

```
    wire [15:0]udp_total_len;

    assign udp_total_len = data_total_len + 8'd8;
```

每个 IP 报文的长度为用户数据长度（data_total_len）+ UDP 首部长度（8）+ IP 首部长度。一般取 IP 首部长度为 20，则 IP 数据包长度

```
wire [15:0]ip_total_len;

assign ip_total_len = data_total_len + 8'd28;
```

为了测试 IP 首部校验和代码是否确实能够完成 IP 首部的校验和计算，在这里设计一个可变长度的数据发送序列。即每次发送数据的长度都不一样，用户数据长度从 22 到 28 个字节循环变化，发送的内容为：

```
Hello, welcom to FPGA!

Hello, welcom to FPGA!t

Hello, welcom to FPGA!th

Hello, welcom to FPGA!tha

Hello, welcom to FPGA!than

Hello, welcom to FPGA!thank

Hello, welcom to FPGA!thanks
```

这样，就能模拟出不同数据包长度时 IP 首部校验和是否正确。控制每次发送数据包长度不一样的方法很简单，即每发完一次数据，就将 data_total_len 的值加 1，这样在 22 到 28 个字节长度之间循环变化。

为了确定上一帧数据发送完成的时间，对以太网帧发送模块 eth_send 模块代码进行一定改写，改写的内容包括

1、添加一个发送完成标志信号输出端口 send_done

2、每当一帧数据发送完成，send_done 信号产生一个高脉冲

```
31      input [47:0]src_mac; //本机/源MAC地址
32      input [15:0]type_length;    //数据帧类型
33      input [31:0]CRC_Result; //CRC校验结果
34      output CRC_EN;
35      output reg send_done; //发送完成信号
36      input [11:0]data_length;    //数据长度（因为MII接口
37
```

```
184     //每次发送完成，产生发送完成标志信号
185     always@(posedge mii_tx_clk or negedge rst_n)
186     if(!rst_n)
187        send_done <= #1  1'b0;
188     else if(cnt == 53)
189        send_done <= #1  1'b1;
190     else
191        send_done <= #1  1'b0;
```

然后，每次 send_done 信号产生高电平，则 data_total_len 自加 1，该部分代码在 eth_send_test 中如下所示：

```
//每次发送完成，将发送的数据长度累加 1
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    data_total_len <= 22;
else if(send_done)begin
    if(data_total_len >= 28)
        data_total_len <= 22;
    else
        data_total_len <= data_total_len + 1'b1;
end
else
    data_total_len <= data_total_len;
```

最后，在数据输出查找表中，将 UDP 和 IP 报头中可变部分替换为代码中定义的寄存器或者线网。（下述代码中深色背景部分即为替换后的可变部分）

```
//UDP 包
always@(*)
begin
    case(data_cnt)
        0   : fifo_rddata = 4'h5;//首部长度
```

```
            1   : fifo_rddata = 4'h4;//协议版本


            //服务类型
            2   : fifo_rddata = 4'h0;
            3   : fifo_rddata = 4'h0;


            //IP 数据报总长度（IP 报头+数据）
            4   : fifo_rddata = ip_total_len[11:8];
            5   : fifo_rddata = ip_total_len[15:12];
            6   : fifo_rddata = ip_total_len[3:0];
            7   : fifo_rddata = ip_total_len[7:4];


            //数据包标识
            8   : fifo_rddata = 4'h0;
            9   : fifo_rddata = 4'h0;
            10  : fifo_rddata = 4'h0;
            11  : fifo_rddata = 4'h0;


            //标识+分段偏移
            12  : fifo_rddata = 4'h0;
            13  : fifo_rddata = 4'h0;
            14  : fifo_rddata = 4'h0;
            15  : fifo_rddata = 4'h0;


            //生存时间
```

```
            16  : fifo_rddata = 4'h0;
            17  : fifo_rddata = 4'h4;

            //数据报类型 17： UDP
            18  : fifo_rddata = 4'h1;
            19  : fifo_rddata = 4'h1;

            //IP 报头校验和 使用自动 IP 和校验逻辑生成的校验和值
            20  : fifo_rddata = ip_checksum[11:8];
            21  : fifo_rddata = ip_checksum[15:12];
            22  : fifo_rddata = ip_checksum[3:0];
            23  : fifo_rddata = ip_checksum[7:4];

            //源地址 192.168.0.2
            24  : fifo_rddata = src_ip[27:24];
            25  : fifo_rddata = src_ip[31:28];
            26  : fifo_rddata = src_ip[19:16];
            27  : fifo_rddata = src_ip[23:20];
            28  : fifo_rddata = src_ip[11:8];
            29  : fifo_rddata = src_ip[15:12];
            30  : fifo_rddata = src_ip[3:0];
            31  : fifo_rddata = src_ip[7:4];

            //目的地址 192.168.0.3
            32  : fifo_rddata = dst_ip[27:24];
```

```
            33  : fifo_rddata = dst_ip[31:28];
            34  : fifo_rddata = dst_ip[19:16];
            35  : fifo_rddata = dst_ip[23:20];
            36  : fifo_rddata = dst_ip[11:8];
            37  : fifo_rddata = dst_ip[15:12];
            38  : fifo_rddata = dst_ip[3:0];
            39  : fifo_rddata = dst_ip[7:4];

            //源端口号 5000(0x1388)
            40  : fifo_rddata = src_port[11:8];
            41  : fifo_rddata = src_port[15:12];
            42  : fifo_rddata = src_port[3:0];
            43  : fifo_rddata = src_port[7:4];

            //目的端口号
            44  : fifo_rddata = dst_port[11:8];
            45  : fifo_rddata = dst_port[15:12];
            46  : fifo_rddata = dst_port[3:0];
            47  : fifo_rddata = dst_port[7:4];

            //UDP 数据报总长度（UDP 报头+数据）
            48  : fifo_rddata = udp_total_len[11:8];
            49  : fifo_rddata = udp_total_len[15:12];
            50  : fifo_rddata = udp_total_len[3:0];
            51  : fifo_rddata = udp_total_len[7:4];
```

```
            //UDP 报头校验和  忽略
            52  : fifo_rddata = 4'h0;
            53  : fifo_rddata = 4'h0;
            54  : fifo_rddata = 4'h0;
            55  : fifo_rddata = 4'h0;

//          //UDP 报头校验和  错误校验和
//          52  : fifo_rddata = 4'h2;
//          53  : fifo_rddata = 4'h1;
//          54  : fifo_rddata = 4'h4;
//          55  : fifo_rddata = 4'h3;

            //用户数据：Hello, welcom to FPGA!
            56  : fifo_rddata = 4'h8;//H
            57  : fifo_rddata = 4'h4;

            58  : fifo_rddata = 4'h5;//e
            59  : fifo_rddata = 4'h6;

            60  : fifo_rddata = 4'hc;//l
            61  : fifo_rddata = 4'h6;

            62  : fifo_rddata = 4'hc;//l
            63  : fifo_rddata = 4'h6;
```

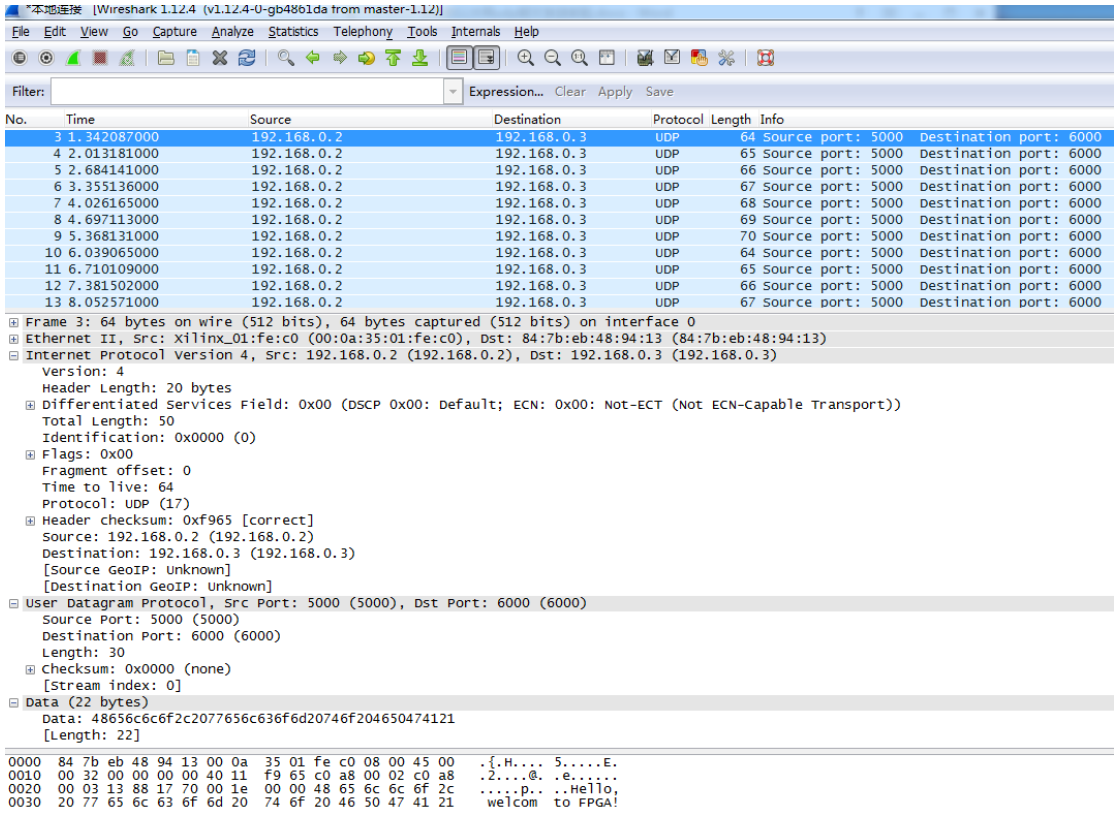
```
64  : fifo_rddata = 4'hf;//0
65  : fifo_rddata = 4'h6;

66  : fifo_rddata = 4'hc;//,
67  : fifo_rddata = 4'h2;

68  : fifo_rddata = 4'h0;//
69  : fifo_rddata = 4'h2;

70  : fifo_rddata = 4'h7;//w
71  : fifo_rddata = 4'h7;

72  : fifo_rddata = 4'h5;//e
73  : fifo_rddata = 4'h6;

74  : fifo_rddata = 4'hc;//l
75  : fifo_rddata = 4'h6;

76  : fifo_rddata = 4'h3;//c
77  : fifo_rddata = 4'h6;

78  : fifo_rddata = 4'hf;//o
79  : fifo_rddata = 4'h6;

80  : fifo_rddata = 4'hd;//m
```

```
            81  : fifo_rddata = 4'h6;


            82  : fifo_rddata = 4'h0;//
            83  : fifo_rddata = 4'h2;


            84  : fifo_rddata = 4'h4;//t
            85  : fifo_rddata = 4'h7;


            86  : fifo_rddata = 4'hf;//o
            87  : fifo_rddata = 4'h6;


            88  : fifo_rddata = 4'h0;//
            89  : fifo_rddata = 4'h2;


            90  : fifo_rddata = 4'h6;//F
            91  : fifo_rddata = 4'h4;


            92  : fifo_rddata = 4'h0;//P
            93  : fifo_rddata = 4'h5;


            94  : fifo_rddata = 4'h7;//G
            95  : fifo_rddata = 4'h4;


            96  : fifo_rddata = 4'h1;//A
            97  : fifo_rddata = 4'h4;
```

```
            98  : fifo_rddata = 4'h1;//!
            99  : fifo_rddata = 4'h2;

            100 : fifo_rddata = 4'h4;//t
            101 : fifo_rddata = 4'h7;

            102 : fifo_rddata = 4'h8;//h
            103 : fifo_rddata = 4'h6;

            104 : fifo_rddata = 4'h1;//a
            105 : fifo_rddata = 4'h6;

            106 : fifo_rddata = 4'he;//n
            107 : fifo_rddata = 4'h6;

            108 : fifo_rddata = 4'hb;//k
            109 : fifo_rddata = 4'h6;

            110 : fifo_rddata = 4'h3;//s
            111 : fifo_rddata = 4'h7;
            default:fifo_rddata = 4'h0;
        endcase
    end
```

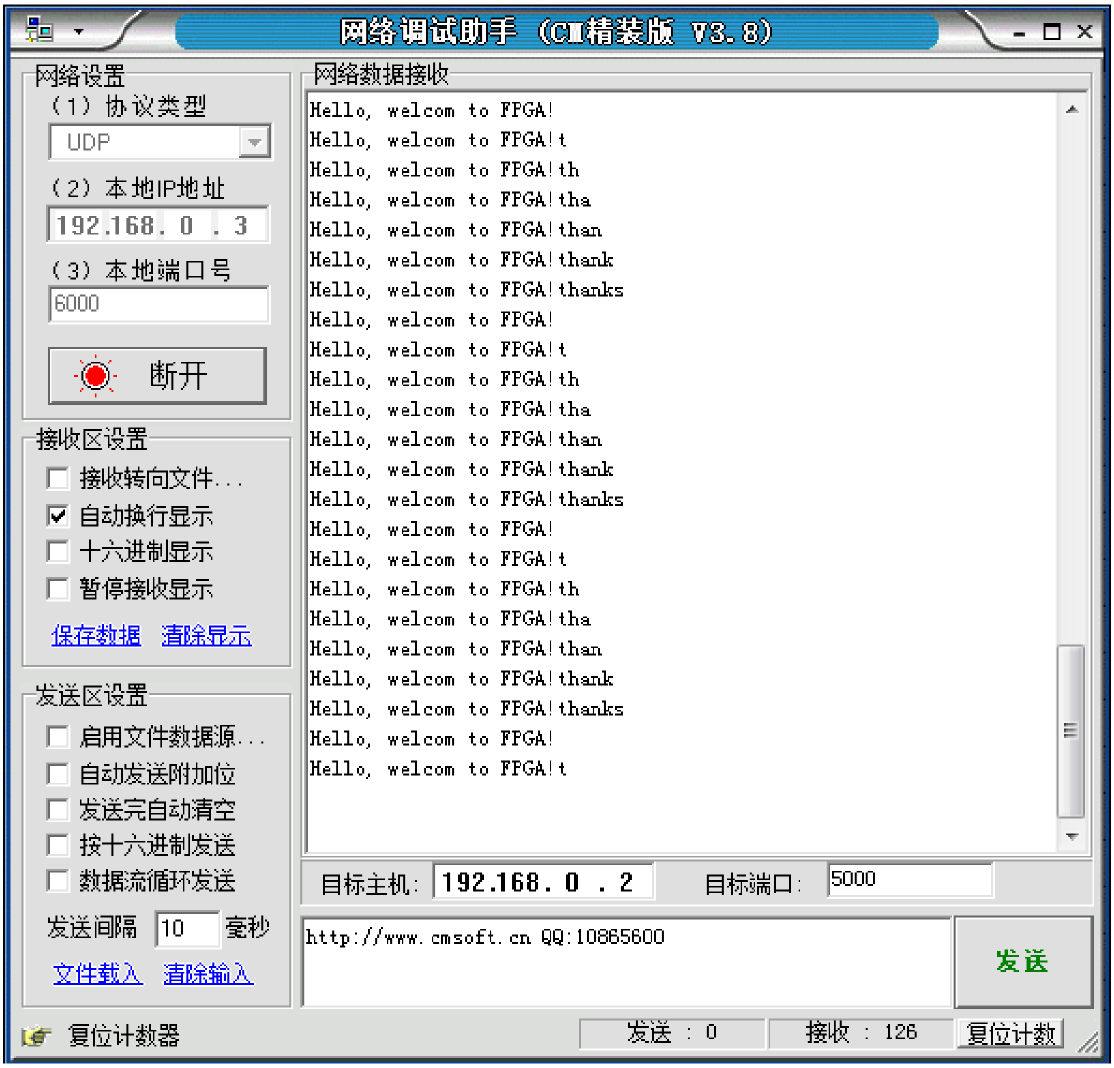
Data_cnt 从 100 到 111 是在上一个实验的内容上增加的 6 个字节，其内容为“thanks”，每次发送时，根据设定的 data_total_len 的值，在“Hello, welcome to FPGA!”的末尾依次多发送 thanks 的一个到 6 个字符。

按照上述说明修改好后，全编译工程并下载 sof 文件到 AC620 FPGA 开发板，使用 wireshark 工具抓取 PC 网卡，按照之前打开 UDP 校验和检查的方式，打开 wireshark 中对 ip 校验和的验证。对抓取的报文分析如下所示：



可以看到，AC620 开发板循环发出 64 到 70 个字节长度的数据包，且 IP 校验和都是正确的。表明本设计能够应对各种数据长度的 UDP 报文发送。

打开网络调试工具，接收到的数据内容如下所示：



至此，基于 FPGA 的以太网 UDP 数据发送协议就设计完成了

第 9 章 基于 FIFO 接口的 UDP 数据包发送

本设计实例提供的工程名称为 *udp_send_fifo_test.qar*。

本设计相关教程暂不提供文档。

第 10 章 以太网图像传输实验

该实验源码仅对 AC620 板卡客户开放，实验效果图如下所示

基于 OV5640 的以太网图像传输效果（图片由网友拍摄提供）

基于 OV7670 的以太网图像传输效果（图片由网友拍摄提供）