
小梅哥以太网计和开发教程

本文档所有内容均由小梅哥本人编写，未经作者许可，严禁网络传播

前言

以太网（Ethernet）是一种计算机局域网技术。IEEE 组织的 IEEE 802.3 标准制定了以太网的技术标准，它规定了包括物理层的连线、电子信号和介质访问层协议的内容。

以太网是在 20 世纪 70 年代研制开发的一种基带局域网技术，使用同轴电缆作为网络媒体，采用载波多路访问和冲突检测(CSMA/CD)机制，数据传输速率达到 10MBPS。随着技术的发展，当前以太网已经能够做到 100Gbps 的超高传输速率了。

自打问世以来，以太网就以其灵活的互联特性和强大的数据传输能力，成为了几乎所有电脑设备的标配。通过一根网线和一个网络访问账号，用户就可以在支持以太网的设备上访问世界各地服务器上的数据。从最开始的电话线传输，发展到 10M、100M、1Gbps 以太网，以及现在的 10G、40G、100G 以太网。从最早期只能传输简单文本信息，到现在可以实时传输超高清视频数据流，以太网一直是我们通过 Internet 进行工作、学习、生活、娱乐的强力支撑。

虽然目前以太网的传输速率已经可以达到 10Gbps、40Gbps、100Gbps 的速率，但是它们主要用于服务器内部进行高速数据交换，或者电信骨干网络。对于一般的消费级和工业应用，大部分都还是以百兆和千兆以太网为主。

上面提到，以太网主要规定的是物理层和介质访问层协议的内容，没有对更上层的协议进行定义。而我们常用的以太网通信应用，例如网络聊天，视频聊天，远程文件下载等，都是基于以太网物理层，通过定义上层的具体数据传输协议实现的。例如 QQ 聊天消息收发是基于 UDP 协议加腾讯自定义上层协议，实时视频数据的传输也多采用 UDP 协议，而文件传输则使用 FTP 传输协议。除开这些协议，还有应用非常广泛的网页传输协议 HTTP 和可靠数据传输使用的 TCP 协议等。

如今，工业以太网非常火热，使用较多的有 5 大主流协议，包括 Ethernet/IP、PROFINET、POWERLINK、EtherCAT、SERCOSIII。笔者了解较多的是 EtherCAT 协议，该协议主要用作于运动控制系统，很多的自动化应用系统都选择采用 EtherCAT 协议进行通信，完成多轴机械臂的协同控制。而使用 FPGA 实现 EtherCAT 从站，使用 SoC FPGA 实现 EtherCAT 主站，能够带来更高的实时效率。

除了工业控制，以太网在视频监控领域也使用较多，比较典型的一类就是工业相机。采用以太网接口的工业相机，能够使用网线连接到接收端，相较于使用 USB 传输方案的工业相机，采用以太网传输能够显著增加数据传输距离，一般的千兆工业相机在无中继情况下传输距离能够达到 100 米，而 USB 的可靠传输距离大概在 5M 左右。

对于 FPGA 来说，网络通信是其最经典的应用之一。无论是移动通信基站，还是大型数据交换中心，都大量的使用了 FPGA 作为通信设备的核心器件。掌握以太网通信基本原理和使用 FPGA 实现网络通信的方法，不仅具有极强的学习价值，还拥有非常明确的实用、商用价值。

FPGA 的以太网应用

在 FPGA 中，以太网通信也有着非常广泛的应用案例。除了上述提到的大型数据交换中心，另一个非常基础的应用就是大型 LED 显示屏。例如各种广场上的 LED 广告屏，大厦外墙的 LED 广告屏，这些显示屏一般安置在户外，其显示的内容则由放置在室内的计算机主机传输，计算机主机和 LED 显示屏之间，由于需要实时传输大量的图像数据，一般都使用以太网作为传输方式，在 LED 显示屏中使用 FPGA 实现以太网，完成实时图像数据的接收并驱动 LED 点阵显示。

由于 FPGA 实现以太网具有高效，低延时的特点，目前也有众多的安全厂商开发基于 FPGA 的硬件网络防火墙，来防御对服务器的恶意网络攻击。

对于很多专用定制型场合，为了进行高速数据采集和传输，也有众多使用 FPGA 实现的以太网来进行实时高效数据传输的应用，例如笔者就曾参与设计过一个基于 FPGA 和以太网接口的高速数据采集传输系统，用来解决数据的采集和实时无丢失传输问题。

开发板硬件概述

为了让大家能够学习使用 **FPGA** 进行以太网应用开发的思路和方法，我们在芯路恒出品的众多 **FPGA** 开发板上，都设提供了以太网通信接口，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。



FPGA 实现以太网通信协议

接触过以太网协议的用户，应该最常听说的就是 **TCP/IP** 协议。确实，在计算机和嵌入式系统中，**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 **TCP/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，而且功能和易用性都无法得到保证。

那么怎样在 **FPGA** 上实现 **TCP/IP** 协议呢？答案就是 **SOPC** 技术。即使用可编程片上系统技术，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，因此很多用户都难以上手，而且实现的以太网从性能上来说和

其他通用 CPU 或处理器相比，也没有任何的优势，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。这种传输方式从理论上讲是无法保证发送方发送的每一个数据包接收方都能正确接收的，但是在很多场合，又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 大屏显示系统中非常广泛，所以本节内容所讨论的基于 FPGA 的以太网通信就是围绕着实现高效的 UDP 数据传输这种目的展开的，至于基于软核 CPU 实现 TCP/IP 传输的方式，不作为本节内容讨论的重点。

基于 FPGA 的以太网通信实验实操

以太网通信涉及到的内容和知识点非常的庞杂，学习需要花费一定的时间和精力。为了让大家快速上手 FPGA 的应用，本节将先介绍我们提供的 3 个基于 FPGA 的应用实验的操作方法。让大家先从最直观的角度体验 FPGA 实现以太网通信的魅力。

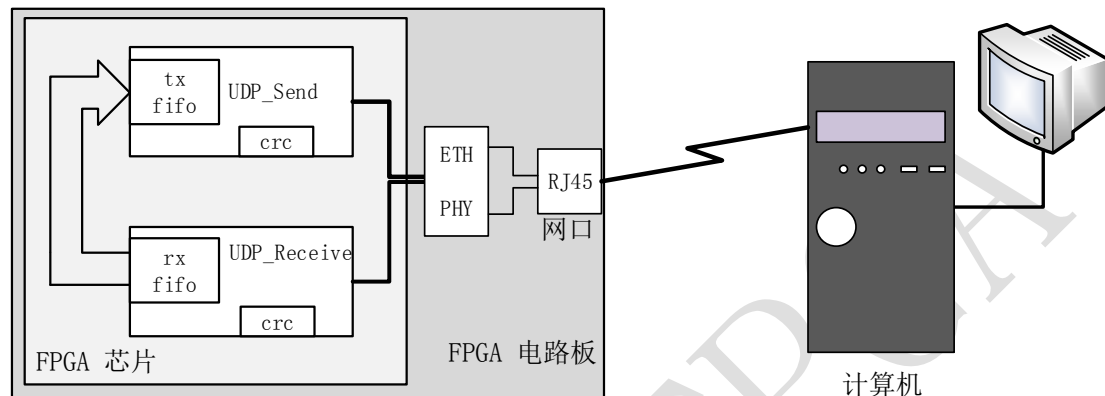
说明：实验中所有用到的软件都可在开发板资料包的“05_常用软件”目录下找到。

以太网回环收发测试

在测试以太网通信速率和通信质量的应用中，最常用的测试方式就是进行回环测试。所谓回环测试，就是由网络连接中的一个节点作为主机，运行测试程序，该测试程序会向被测试的设备发送以太网数据包，被测试设备收到数据后，原封不动的将接收到的数据内容通过以太网发送给主机，主机收到被测设备发回的数据后，与自己发送的数据进行比对，或者以文本的形式显示，供测试者人工对比分析。

UDP MII 回环测试工程介绍

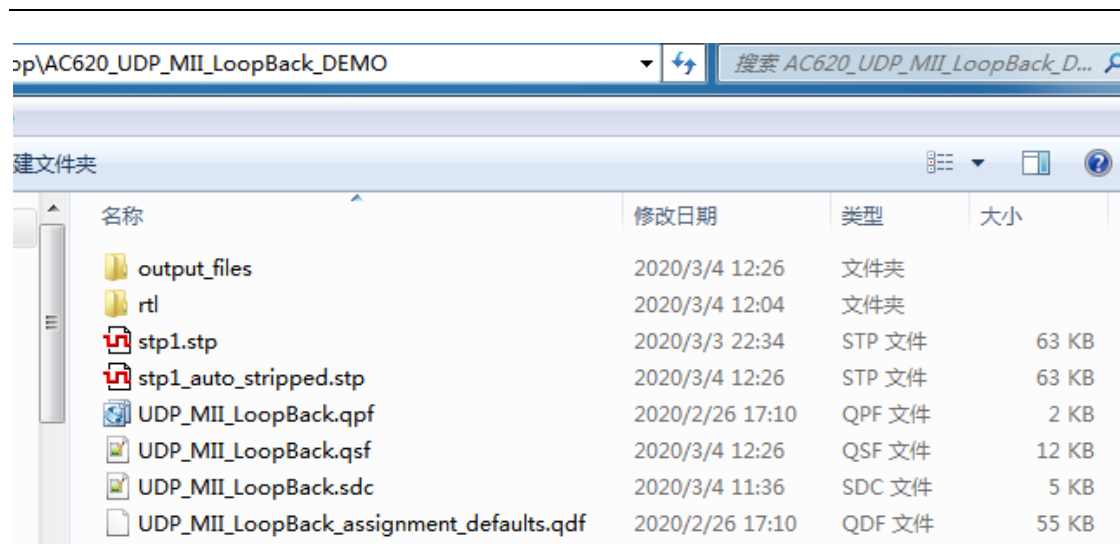
为了完成基本的以太网回环测试，我们提供了一个基于 FPGA 的以太网回环测试 DEMO。该 DEMO 使用 UDP 协议，接收 PC 发送的 UDP 数据包，提取出其中的数据部分并使用 UDP 协议发回给 PC。整个系统框图如下图所示：



本例程在小梅哥团队出品的 AC620 开发板上使用 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。

本例对应的例程工程压缩包名为 AC620_UDP_MII_LoopBack_DEMO.rar，该文件可在我们提供的配套资料中找到。在我们的官网 www.corecourse.cn 搜索 MII_LoopBack 关键词也可找到。

解压 AC620_UDP_MII_LoopBack_DEMO.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：



图

其中 rtl 文件夹下存放的为程序源码，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
UDP_MII_LoopBack.qpf	工程文件，使用 Quartus II 13.0 打开。
UDP_MII_LoopBack.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。
UDP_MII_LoopBack.sdc	工程的时序约束文件，约束了时钟和以太网的 IO 时序。
stp1.stp	SignalTap II 工具的设置文件，抓取 MII 接口的发送和接收接口的时序信息
stp1_auto_stripped.stp	stp1.stp 运行后抓取的上一次数据的记录
-rtl/eth_rx ---UDP_Receive.v	UDP 协议接收模块，该模块能够完整的解析从以太网 PHY 的 MII 接口收到的数据并解析得到 UDP 报文的数据部分。事实上，该模块是实现了 MAC 层、IP 层、UDP 协议层的多层解析之后，最终直接得到传输的用户数据部分。
-rtl/eth_rx ---eth_rx_fifo.v	UDP 接收数据缓存 FIFO，从 UDP 协议中接收并解析到的正确的数据部分会存储在该 FIFO 中。
-rtl/eth_rx crc32_bitN.v	CRC32 校验逻辑，实现对 MAC 层数据的 CRC32 校验功能
-rtl/eth_tx UDP_Send.v	UDP 协议发送模块，该模块将用户输入的需要发送的数据内容经过 UDP、IP、MAC 层协议层层打包后，通过 MII 接口输出给以太网 PHY 芯片，以完成数据的发送。
-rtl/eth_tx eth_tx_fifo.v	UDP 发送数据缓存 FIFO，用户将需要发送的数据写入该 FIFO，然后由 UDP_Send 中的发送逻辑将数据读出并最终通过 UDP 协议发送。
-rtl/eth_tx crc32_d4.v	CRC32 校验逻辑，计算经由 MAC 层发送的数据的 CRC 值并附加在 MAC 结尾发送。

-rtl ---UDP_MII_LoopBack.v	整个工程的设计顶层文件，例化了 UDP 接收和 UDP 发送以及回环数据搬运模块，实现完整的回环测试功能。
-rtl ---UDP_MII_LoopBack_tb.v	工程的仿真测试激励文件，该文件对 UDP_MII_LoopBack.v 进行测试，为了产生测试的以太网数据包，使用了已经经过单独验证无误的 UDP 发送模块（UDP_Send）作为以太网数据包的数据产生激励。
-rtl ---LoopBack_CRTL.v	回环数据搬运模块，将 UDP 接收模块（UDP_Receive）中接收并存储在 FIFO 中的数据读出，写入到 UDP 发送模块（UDP_Send）的发送 FIFO 中。
-rtl/mdio ---mdio_bit_shift.v	MDIO 底层传输协议实现，完成基于 MDIO 协议对 PHY 寄存器的一次读或者写操作
-rtl/mdio --- phy_reg_config.v	以太网 PHY 芯片的寄存器配置模块，控制 mdio_bit_shift 模块对以太网 PHY 的指定寄存器进行读写操作以完成以太网 PHY 的工作模式设置。
-output_files ---UDP_MII_LoopBack.sof	预先编译好的 FPGA 配置文件，将该文件下载到开发板中即可进行测试。

UDP MII 回环测试工程实操

- 1、 使用网线将开发板上的以太网接口和您当前调试测试工程用的 PC 机的网口连接起来。连接好下载线缆并给开发板上电。
- 2、 确认开发板上的以太网接口模式设置开关拨在 GMII 侧。
- 3、 双击 UDP_MII_LoopBack.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 打开，使用其他版本打开或编译遇到问题，请自行解决）。
- 4、 下载 UDP_MII_LoopBack.sof 文件到开发板中，注意这一步一定要先于下面的步骤执行，否则以下操作无法正常进行。
- 5、 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本地连接存在，才表明开发板和电脑的网络才已经连通。此时如果重新下载 sof 文件到开发板中，会发现此本地连接会先消失，然后再重新出现。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。



- 6、 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百兆速率（100.0 Mbps）



- 7、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。如下图所示。

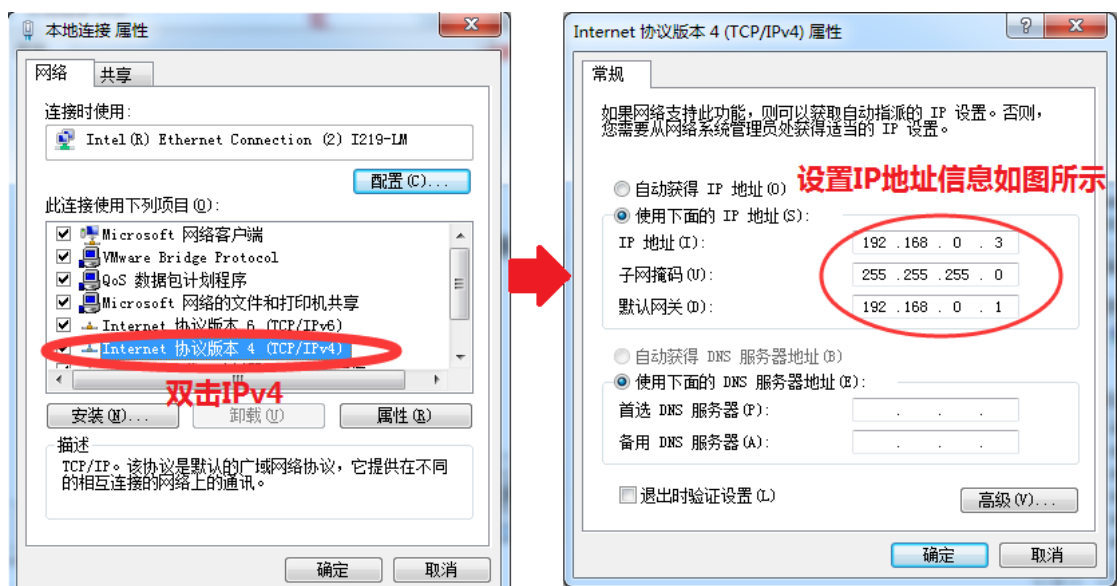


图 0-1 修改 PC 的 IP 地址

- 7、 由于本测试工程不支持 ARP 协议，因此只能通过静态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。

操作时先以管理员身份运行 cmd.exe 程序(该文件在 C:\Windows\System32 路径下)，也就是我们常说的命令行窗口。由于有用户反应在使用时无法成功绑定 arp，经过分析就是操作权限不够，所以这里强调要以管理员身份运行 cmd.exe。然后在窗口中输入下述命令：

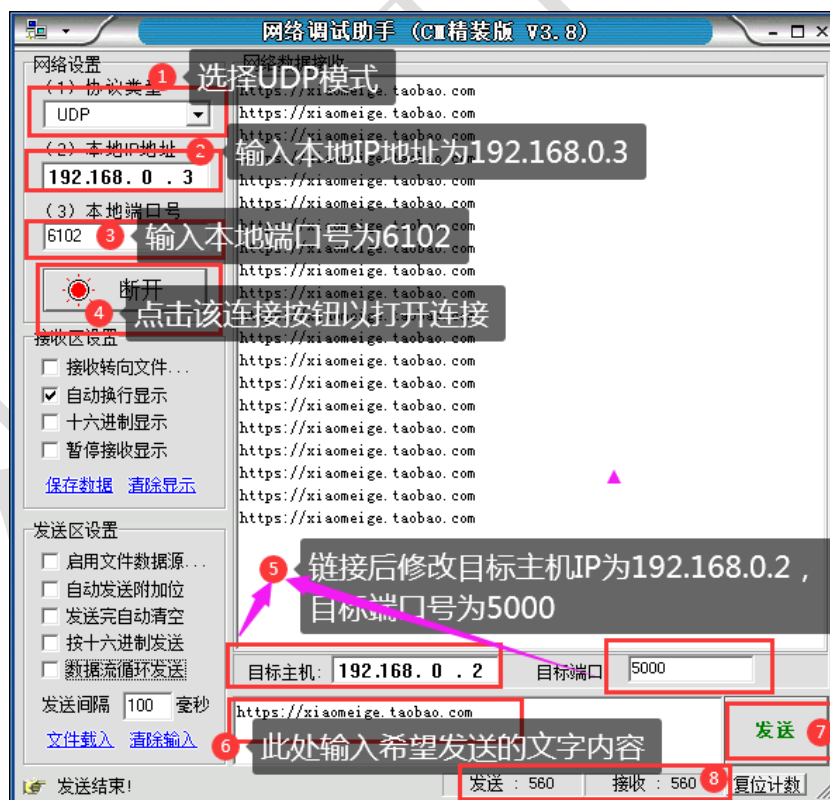
```
arp -s 192.168.0.2 00-07-ed-ac-62-00
```

绑定后我们可以用 arp -a 命令来查看 PC 上绑定的结果，如下图所示：



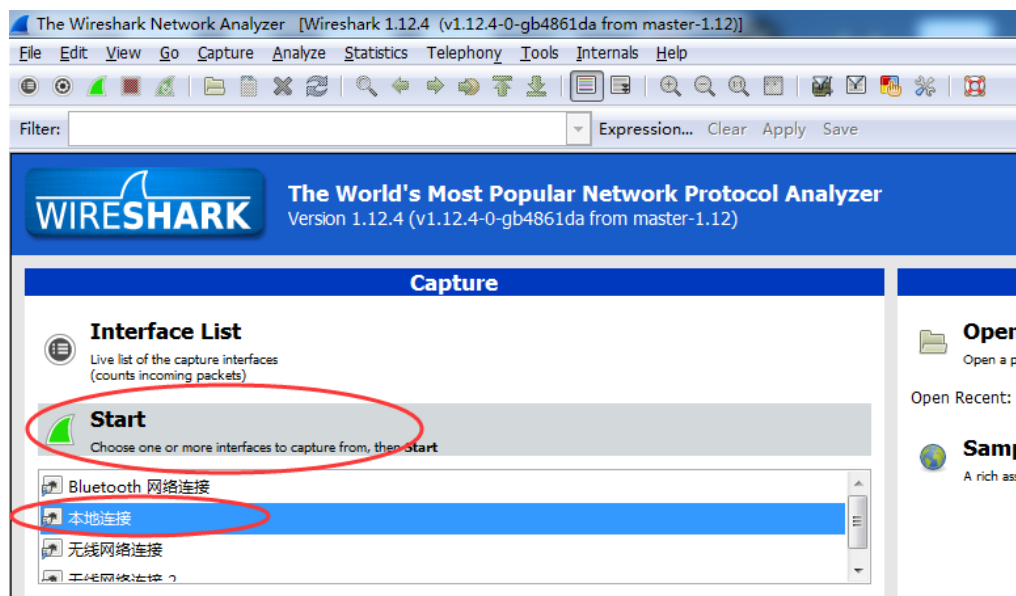
8、 打开网络调试助手（NetAssist.exe）并按照如下所述设置各项参数。

- ①选择协议类型为 UDP。
- ②设置本地 IP 地址为 192.168.0.3。
- ③设置本地端口号为 6102。
- ④点击【连接】按钮以创建连接，连接上后该按钮为红色“断开”字样。
- ⑤连接上后，设置目标主机为 192.168.0.2，目标端口为 5000。
- ⑥在文本框中输入希望发送的文本内容。
- ⑦点击发送按钮，即可在上方的接收窗口中显示 FPGA 发回的数据，与发送的内容一致。
- ⑧在下方的计数器中，检查发送的数据个数与接受的数据个数是否一致。

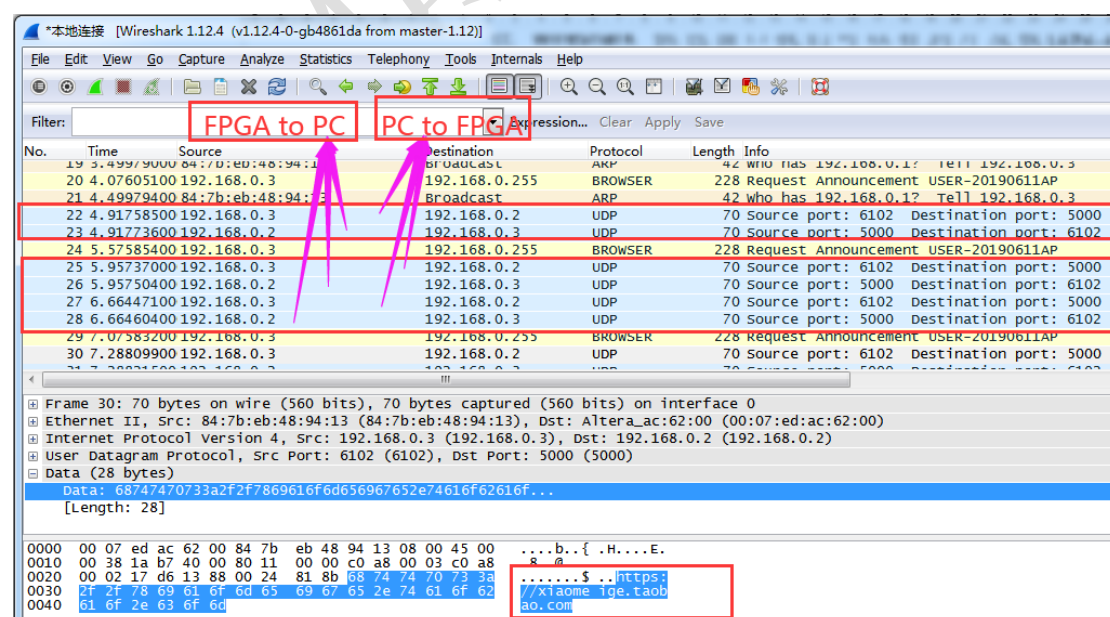


9、 在发送窗口中输入你希望发送的数据，然后点击发送，则上方的接收窗口会接收到与发送框完全一样的内容，每点击一次发送，就会收到一次数据。

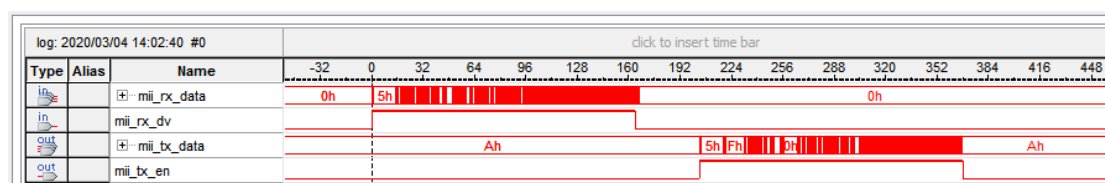
- 10、 安装网络抓包工具 Wireshark, 我们在实验的时候可以用这工具来查看 PC 网口发送的数据和接收到的数据。
- 11、 打开安装好的 wireshark 抓包工具。在软件界面选择您 PC 的有线网卡, 按开始按钮开始抓包。(不同的版本界面有所差异, 我们提供的是 Wireshark_win64_V1.12.4_setup.1427187922.exe)。



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。



12、 双击 `stp1.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：



至此，则完成了基于 `FPGA` 的百兆以太网 `MII` 接口的 `UDP` 协议数据收发回环测试。在本例中介绍的设置 `IP` 地址、绑定 `ARP` 地址、`wireshark` 抓包、以太网调试软件的使用等内容，在后续的实验中都有可能需要，请大家留意。

实验常见问题和解答：

Arp 绑定不成功：

- 1、 先确保你的电脑已经正确的设置了有线网卡的 `IP` 地址
- 2、 确保你是以管理员身份运行的 `cmd.exe`

接收不到数据

- 1、 接收不到数据的情况下，使用 `wireshark` 抓包看能否抓到包，如果能抓到包，则可能是电脑防火墙的问题，请暂时先关闭所有防火墙。
- 2、 如果 `wireshark` 也抓不到包，则检查本地连接是否有正确的连接上，检查链接速度是否为 `100.0Mbps`，`FPGA` 中是否已经下载好了程序。

Wireshark 软件操作差异

部分网友下载的是更新版本的 `wireshark` 工具，新版本的 `wireshark` 软件在选择网卡界面和笔者所使用的版本略有不同，下图为新版 `wireshark` 软件中选择网卡并抓包的界面示意图。



更多问题可上 www.corecourse.cn 搜索“以太网实验常见问题”关键词搜索相关帖子。

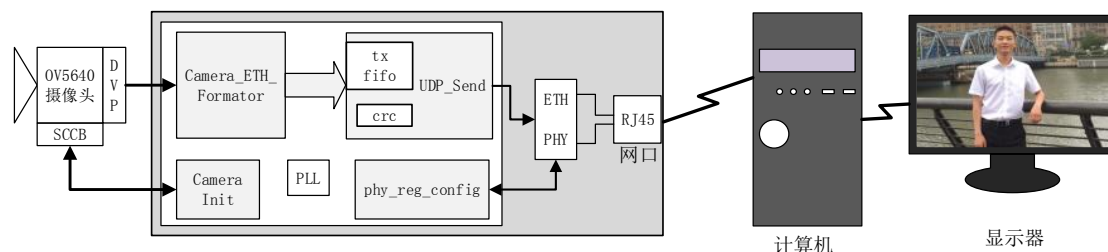
以太网图像发送 PC 接收显示实验测试

通过 FPGA 实现以太网的一个主要目的就是使用以太网将 FPGA 采集到的各种数据发送到 PC，如高速 ADC 实时采集数据，或者图像传感器采集的图像数据。此类应用传输过程中数据量大，实时性要求较高。使用基于 CPU 架构实现的软件以太网协议栈，受限于 CPU 计算性能和数据组包的规律，一般效率都无法做到很高。而使用 FPGA 实现以太网传输，则可以提供稳定且高效的传输能力。

无论是 ADC 实时采集的模拟数据，还是图像传感器或其他传感器采集的数据，其数据特点都是高速且高实时性。因此本节将以一个以太网图像传输实例为例，介绍使用以太网传输图像到 PC 并显示的方式。

以太网图像传输工程介绍

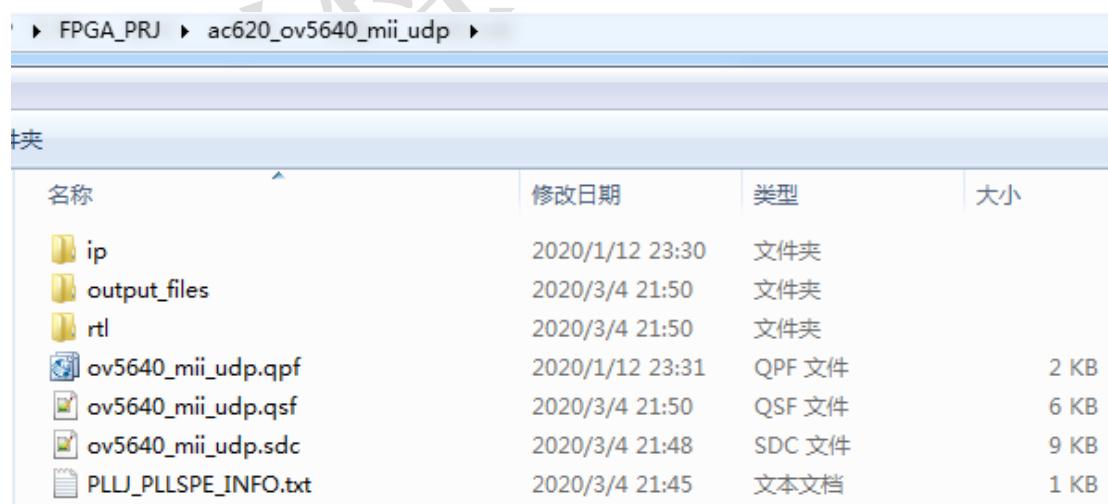
基于以太网的图像传输工程，使用 UDP 协议，将 OV5640 摄像头采集到的图像数据经由以太网传输到 PC 端，再由 PC 端的图像显示软件接收图像数据并绘制在屏幕上，以还原图像内容。整个系统框图如下图所示：



本例程在小梅哥团队出品的 AC620 开发板上使用 verilog 实现以太网 UDP 协议通信。FPGA 采集 OV5640 输出的 RGB565 格式的图像数据，并将采集到的图像数据使用一定的编号方式编号后以行为单位，使用 UDP 协议经由以太网发送到 PC 机。PC 机上使用我们专门开发的显示软件（小梅哥 UDP 摄像头 V3.exe）接收 FPGA 发送的编号后的图像数据，解析后还原为图像内容绘制在 PC 机显示屏上。

本例对应的例程工程压缩包名为 ac620_ov5640_mii_udp.rar，该文件可在我们提供的配套资料中找到。在我们的官网 www.corecourse.cn 搜索 ov5640_mii 关键词也可找到。

解压 ac620_ov5640_mii_udp.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，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
---------	------------

ov5640_mii_udp.qpf	工程文件，使用 Quartus II 13.0 打开。
ov5640_mii_udp.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。
ov5640_mii_udp.sdc	工程的时序约束文件，约束了时钟和以太网的 IO 时序。
-rtl/eth UDP_Send.v	UDP 协议发送模块，该模块将用户输入的需要发送的数据内容经过 UDP、IP、MAC 层协议层层打包后，通过 MII 接口输出给以太网 PHY 芯片，以完成数据的发送。
-rtl/eth eth_tx_fifo.v	UDP 发送数据缓存 FIFO，用户将需要发送的数据写入该 FIFO，然后由 UDP_Send 中的发送逻辑将数据读出并最终通过 UDP 协议发送。
-rtl/eth crc32_d4.v	CRC32 校验逻辑，计算经由 MAC 层发送的数据的 CRC 值并附加在 MAC 结尾发送。
-rtl --- Camera_ETH_Formator.v	整个工程的设计顶层文件，例化了 UDP 接收和 UDP 发送以及回环数据搬运模块，实现完整的回环测试功能。
-rtl --- ov5640_mii_udp.v	整个工程的设计顶层文件，例化了 UDP 发送和摄像头 DVP 接口图像数据捕获编码模块，实现完整的图像采集传输功能。
-rtl/mdio ---mdio_bit_shift.v	MDIO 底层传输协议实现，完成基于 MDIO 协议对 PHY 寄存器的一次读或者写操作
-rtl/mdio --- phy_reg_config.v	以太网 PHY 芯片的寄存器配置模块，控制 mdio_bit_shift 模块对以太网 PHY 的指定寄存器进行读写操作以完成以太网 PHY 的工作模式设置。
-output_files ---UDP_MII_LoopBack.sof	预先编译好的 FPGA 配置文件，将该文件下载到开发板中即可进行测试。

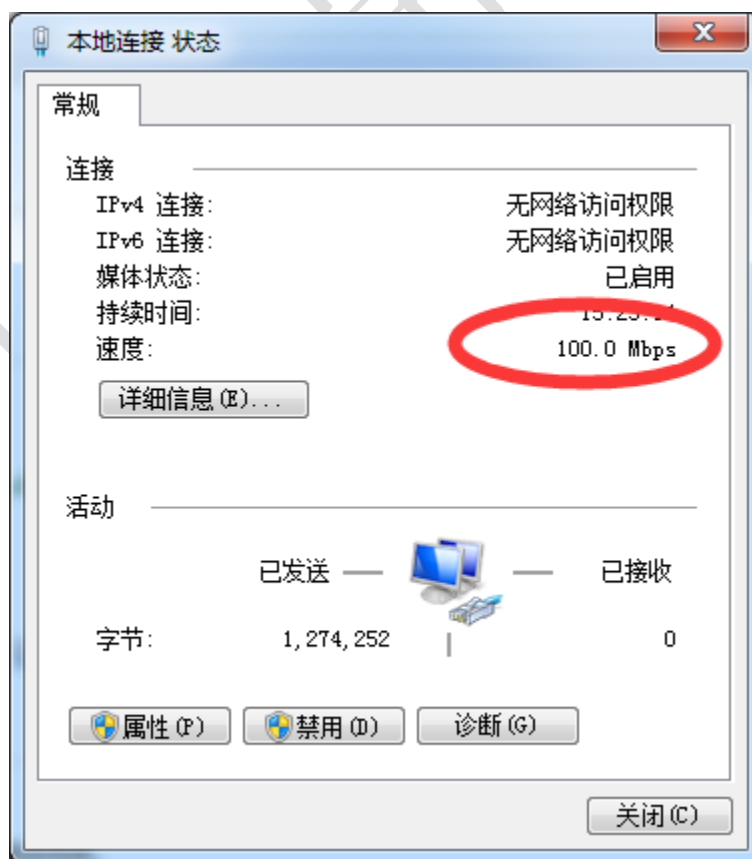
以太网图像传输工程实操

- 1、 使用网线将开发板上的以太网接口和您当前调试测试工程用的 PC 机的网口连接起来。连接好下载线缆并给开发板上电。
- 2、 将 OV5640 摄像头插入开发板的摄像头接口中。
- 3、 确认开发板上的以太网接口模式设置开关拨在 GMII 侧。
- 4、 双击 ov5640_mii_udp.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 打开，使用其他版本打开或编译遇到问题，请自行解决）。
- 5、 下载 ov5640_mii_udp.sof 文件到开发板中，注意这一步一定要先于下面的步骤执行，否则以下操作无法正常进行。
- 6、 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本地连接存在，才表明开发板和电脑的网络才已经连通。此时如果重新下载 sof 文件到开发板中，会发现此本地连接会先消失，然后再重新出现。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。



- 7、 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百兆速率（100.0 Mbps）



7、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，如在前一个实验已经进行了设置，则此处可不用再重复设置。如下图所示。

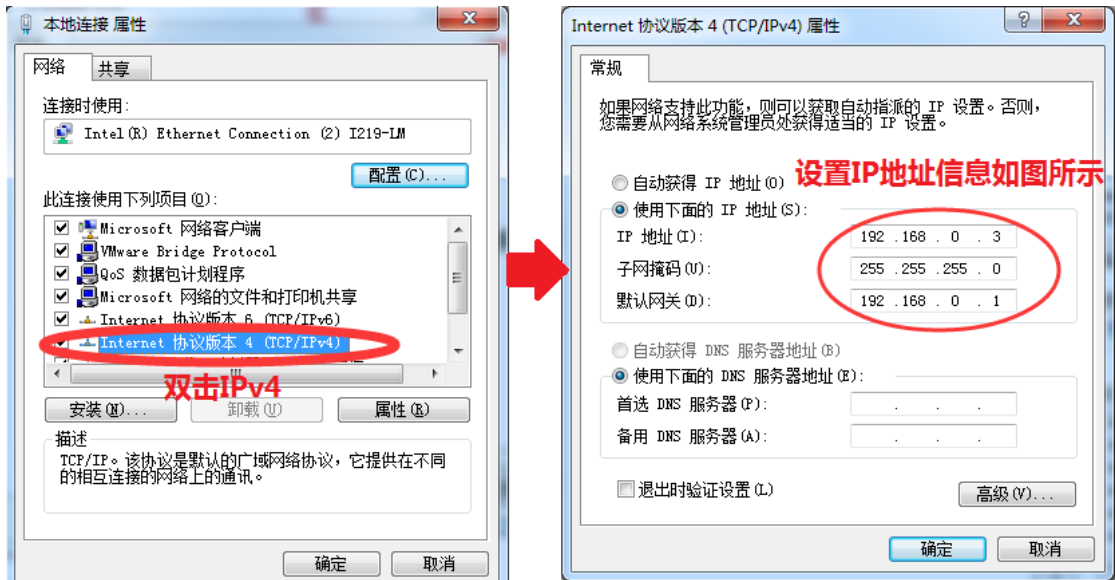


图 0-2 修改 PC 的 IP 地址

8、由于本测试工程不支持 ARP 协议，因此只能通过静态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。

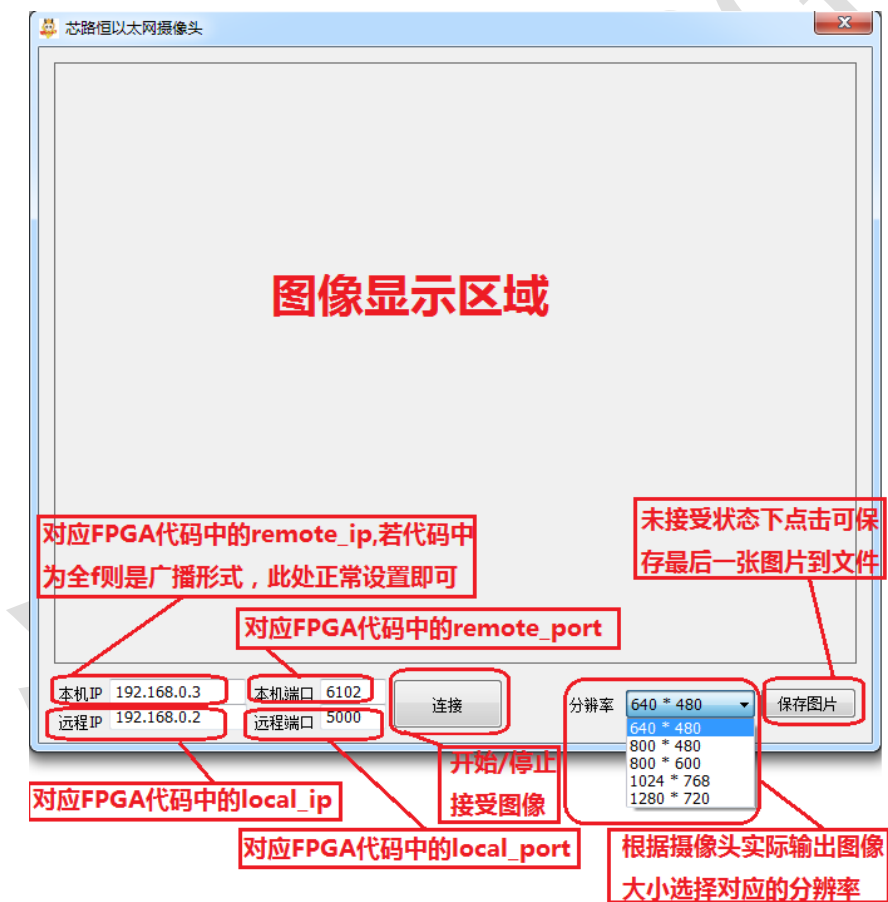
操作时先以管理员身份运行 cmd.exe 程序(该文件在 C:\Windows\System32 路径下)，也就是我们常说的命令行窗口。由于有用户反应在使用时无法成功绑定 arp，经过分析就是操作权限不够，所以这里强调要以管理员身份运行 cmd.exe。然后在窗口中输入下述命令：

```
arp -s 192.168.0.2 00-07-ed-ac-62-00
```

绑定后我们可以用 arp -a 命令来查看 PC 上绑定的结果，如下图所示：



9、 打开网络调试助手(小梅哥 UDP 摄像头.exe)并按照如下所述设置各项参数。

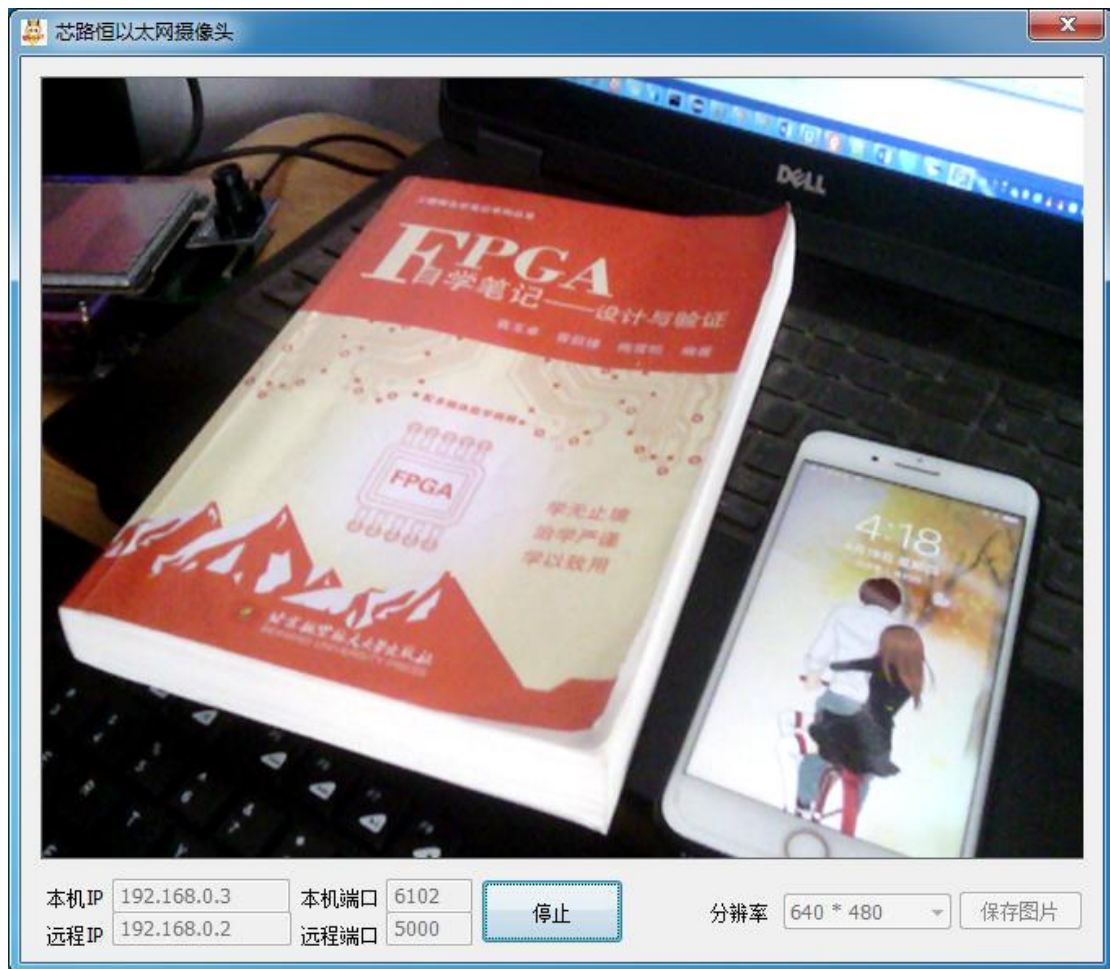


- 本机 IP，对应当前实验所用电脑的 IP 地址，前面已经按照要求设置为了 192.168.0.3，所以这里直接填写该地址。
- 本机端口，对应当前实验所用电脑接收图像时使用的网络端口，该端口在 FPGA 程序中定义为了 6102，所以这里也要设置为 6102 才能正确显示图像。

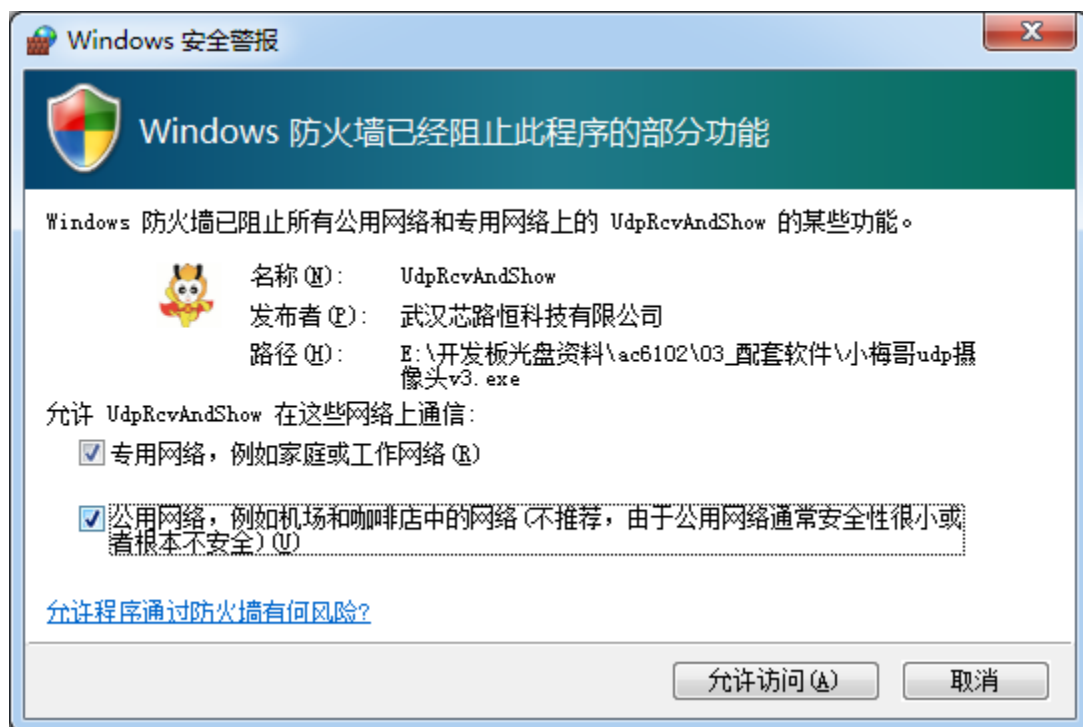
需要解释的是，早期我们默认使用的是 6000 这个端口号，但是发现该端口号与 National Instruments 的软件有冲突，装有 NI 软件的客户无法连接上，所以特修改为 6102 端口号。

- 远程 IP，对应 FPGA 开发板使用的 IP 地址，也就是代码中的 local_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 local_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，百兆 MII 接口使用的分辨率为 640*480。
- 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。

10、 设置好参数后，点击连接按钮即可开始接收并显示图像内容。



- 11、 该软件运行时需要关闭防火墙，软件自带关闭防火墙功能，点击连接时会提示是否关闭防火墙，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。



至此,则完成了基于 FPGA 的百兆以太网 MII 接口的 UDP 协议图像发送到 PC 显示功能实验。本实验操作简单,趣味性和实用性都较强,适合大家用来学习练习。

实验常见问题和解答:

无法连接:

- 1、有可能是该 6102 端口号在你电脑上恰好被其他软件占用,可以更换任意一个端口号,再检查是否能连接上(不在乎能否出图,能连接上就行)来确定该问题,如果换了端口号能正常连接,则是端口号被占用,可在 FPGA 代码中修改 `remote_port` 的值为能连接的端口号后重新编译再测试。
- 2、检查你的电脑有线网络本地连接是否连接上了。
- 3、检查你的电脑 IP 地址是否已经设置为了 192.168.0.3.

接收不到图像

- 1、接收不到图像的情况下,手动确认防火墙是否已经被关闭,软件自带的临时关闭防火墙功能需要软件是以管理员身份运行的,可重新使用管理员身份运

行软件，如果还是无法关闭，则可在电脑设置中人工关闭防火墙。

2、 确保 OV5640 已经正确的接入了开发板。

3、 确保开发板上的 PHY 模式选择开关（右上角）拨在 GMII 方向。

更多问题可上 www.corecourse.cn 搜索“以太网实验常见问题”关键词搜索相关帖子。

以太网相关通信接口详解

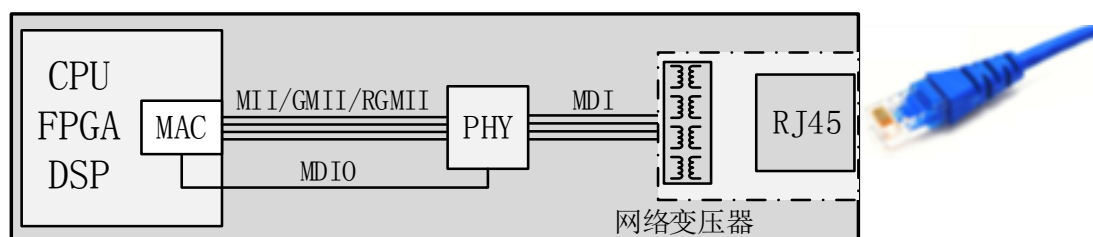
对于以太网来说，其存在的目的是实现不同设备间的联网通信，而实现以太网通信，对于大多数人来说，所熟知的就是一个插接网线的接口，那么这个插接网线的接口背后，又包含着怎样的电路结构呢？一般的数字芯片，只能输出 TTL 电平的信号，那么他们又是怎么最终实现了能够插接网线的以太网接口了，中间用到了怎样的转换电路，这些转换电路又是使用的怎样的数字接口进行数据交互的呢，本节内容将通过详细的图文说明，介绍这些相关的知识点。

以太网 MAC 层接口介绍

对于网口大家相信都不陌生，无论是笔记本、台式机，还是交换机、路由器，上面都有网口，事实上，从专业名词的角度来说，这些网口都应称之为 RJ45 接口。



但从功能来说，这些网口只是起到一个信号连接的作用，本身无主动通信的能力，一个典型的网络通信电路如下图所示。



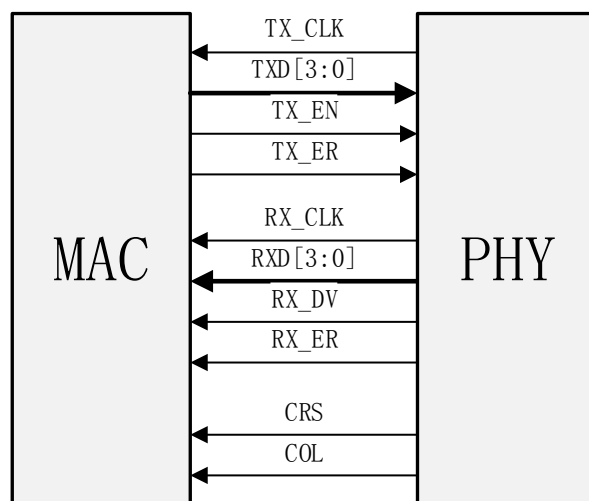
最终送到 RJ45 接口上的电信号要求为高速串行信号，数据是通过串行方式发送的，千兆以太网的位速率为 1.25Gbps、百兆以太网的位速率为 125Mbps。而典型的处理器如 CPU、FPGA、DSP 一般都不具备直接产生符合以太网物理链路层传输的电信号的能力（高端器件往往集成了高速串行收发器，可以支持，这里不做为讨论），只能产生 TTL 电平标准的并行数据，所以在 RJ45 接口和处理器之间，需要使用一个能够实现 TTL 电平和高速串行数据收发模式之间的转换器，这个转换器就是我们常说的 PHY 芯片。

PHY 芯片的最基本功能就是实现并行以太网数据到符合以太网物理层链路数据传输格式的电平信号之间的转换。在 RJ45 接口连接的一侧，传输的是经过了 8b/10b(千兆)或 4b/5b（百兆）编码后，由高速串行发送器驱动的高速串行数据信号，而在连接处理器 MAC 的一侧则根据具体的速率和接口模式，提供 GMII、RGMII、MII、RMII 等多种并行接口。

无论是哪种接口，其传输的信号意义都是一样的，都至少包括接收数据信号（RXD[n:0]）、接收数据有效信号（RX_DV）、接收数据时钟信号（RX_CLK）、发送数据信号（TXD[n:0]）、发送数据使能信号（TX_EN）、发送数据时钟信号（TX_CLK）。区别在于接收数据信号和发送数据信号的位宽。以下将分别介绍 MII、GMII、RGMII 这 3 种接口。

MII 接口

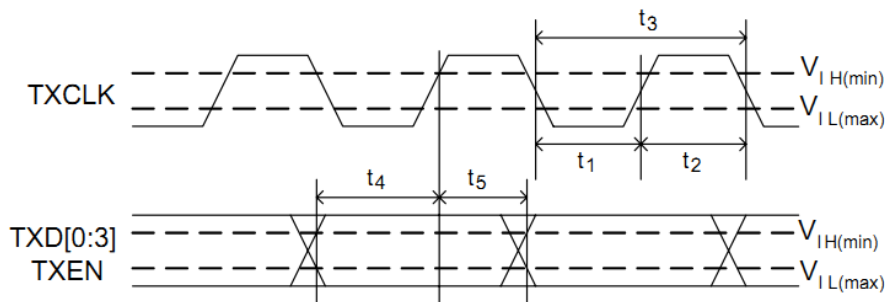
MII 接口应用于 100Mbps 和 10Mbps 以太网模式下，其接口信号连接关系及各信号的介绍如下。



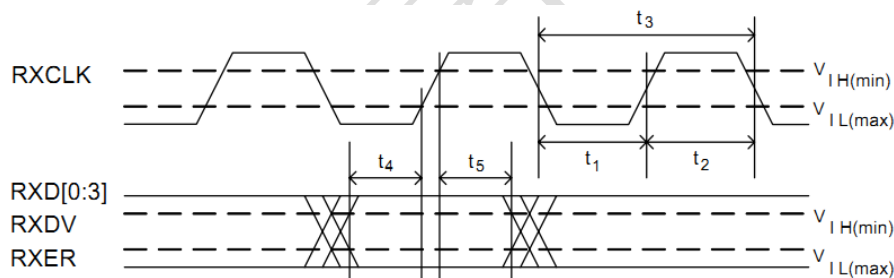
信号	方向	位宽	说明
TX_ER	O	1	即 Transmit Error，发送数据错误提示信号，同步于 TX_CLK，高电平有效，表示 TX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，TX_ER 不起作用。
TX_EN	O	1	即 Transmit Enable，发送使能信号，只有在 TX_EN 有效期内 TXD 上的数据才会传入 PHY 芯片并进行编码传输。
TX_CLK	I	1	发送参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz。需要注意的是，TX_CLK 时钟的方向是从 PHY 指向 MAC 的，此时钟是由 PHY 提供的。
TXD	O	4	即 Transmit Data，数据发送信号，共 4 根信号线，TX_EN 为高电平的情况下，每个时钟周期传输 4 位数据。
RX_ER	I	1	即 Receive Error，接收数据错误提示信号，同步于 RX_CLK，高电平有效，表示 RX_ER 有效期内传输的数据无效。对于 10Mbps 速率下，RX_ER 不起作用。
RX_DV	I	1	即 Receive Data Valid，接收数据有效信号，作用类似于发送通道的 TX_EN。
RXD	I	4	即 ReceiveData，数据接收信号，共 4 根信号线。
RX_CLK	I	1	接收数据参考时钟，100Mbps 速率下，时钟频率为 25MHz，10Mbps 速率下，时钟频率为 2.5MHz。RX_CLK 也是由 PHY 侧提供的。
CRS	I	1	即 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。需要注意的是 CRS 只有 PHY 在半双工模式下有效。
COL	I	1	即 Collision Detectcd，冲突检测信号，不需要同步于参考时钟。需要注意的是 CRS 只有 PHY 在半双工模式下有效。

其中，MAC 侧向 PHY 侧传输数据的时序图如下，参数 t1~t5 是 PHY 芯片能正常接收到数据需要满足的时序参数，VIL 和 VIH 是高低电平的电压标准，电压低于 VIL 表示低电平，电压高于 VIH 表示高电平，详细参数参见具体的 PHY 芯片

手册。从波形图可以看出，发送数据信号 TXD 和发送使能信号需要在时钟上升沿保持稳定。

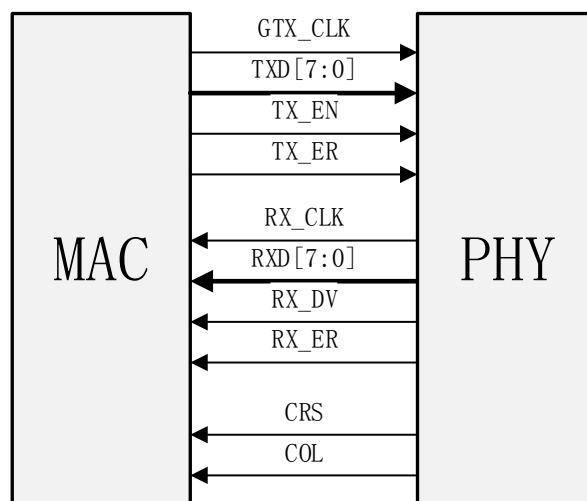


PHY 侧向 MAC 侧传输数据（也就是 MAC 侧接收 PHY 侧传过来的数据）的时序图如下，参数 $t_1 \sim t_5$ 是 PHY 芯片输出数据和与时钟之间的时序参数， V_{IL} 和 V_{IH} 是高低电平的电压标准，电压低于 V_{IL} 表示低电平，电压高于 V_{IH} 表示高电平，详细参数参见具体的 PHY 芯片手册。从波形图可以看出，PHY 芯片传出（也就是 MAC 接收）数据信号 RXD 和数据有效信号在时钟上升沿保持稳定。在 MAC 接收数据时，需要根据传过来数据信号的时序特点进行正确的接收。



GMII 接口

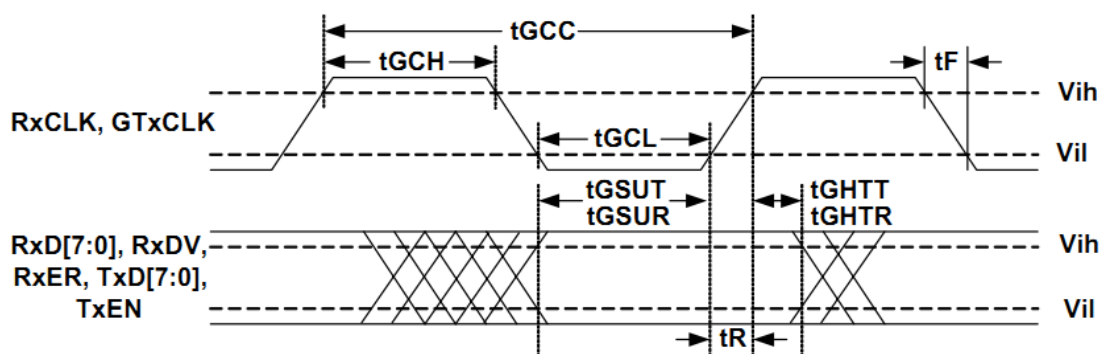
GMII 接口应用于 1000Mbps 以太网模式下，其接口信号连接关系及各信号的介绍如下。



信号	方向	位宽	说明
TX_ER	O	1	即 Transmit Error, 发送数据错误提示信号, 同步于 GTX_CLK, 高电平有效, 表示 TX_ER 有效期内传输的数据无效。
TX_EN	O	1	即 Transmit Enable, 发送使能信号, 只有在 TX_EN 有效期内传的数据才有效。
GTX_CLK	O	1	发送参考时钟, 时钟频率为 125MHz。需要注意的是, GTX_CLK 时钟的方向是 从 MAC 侧指向 PHY 侧 的, 此时钟是由 MAC 提供的, 这里与 MII 接口有所差别的地方。
TXD	O	8	即 Transmit Data, 数据发送信号, 共 8 根信号线
RX_ER	I	1	即 Receive Error, 接收数据错误提示信号, 同步于 RX_CLK, 高电平有效, 表示 RX_ER 有效期内传输的数据无效。
RX_DV	I	1	即 Receive Data Valid, 接收数据有效信号, 作用类似于发送通道的 TX_EN。
RXD	I	8	即 ReceiveData, 数据接收信号, 共 8 根信号线。
RX_CLK	I	1	接收数据参考时钟, 时钟频率为 125MHz。RX_CLK 是由 PHY 侧提供的。
CRS	I	1	即 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。需要注意的是 CRS 只有 PHY 在半双工模式下有效。
COL	I	1	即 Collision Detectd, 冲突检测信号, 不需要同步于参考时钟。需要注意的是 CRS 只有 PHY 在半双工模式下有效。

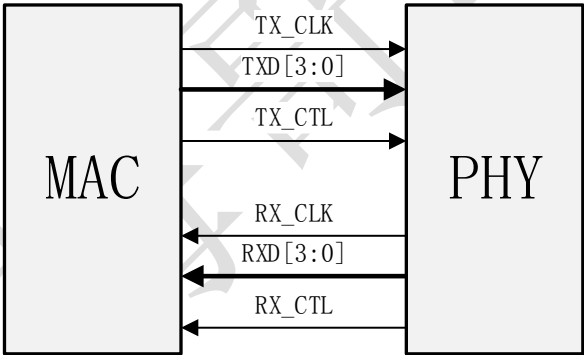
与 MII 接口类似, MAC 侧与 PHY 侧之间传输数据的时序图如下, 图中的参数是 PHY 芯片能正常接收到数据和 PHY 发出数据需要满足的时序参数, V_{IL} 和 V_{IH} 是高低电平的电压标准, 电压低于 V_{IL} 表示低电平, 电压高于 V_{IH} 表示高电平, 详细参数参见具体的 PHY 芯片手册。从波形图可以看出, 发送数据信号 TXD 和发送使能信号需要在时钟 GTX_CLK 的上升沿保持稳定; 同样的 PHY 芯片传出 (也就是 MAC 接收) 数据信号 RXD 和数据有效信号在时钟 RX_CLK 上升沿保持稳定。

在 MAC 发送/接收数据时，需要满足这些时序要求才能让 PHY 正确收到数据和正确接收到 PHY 传过来数据。



RGMII 接口

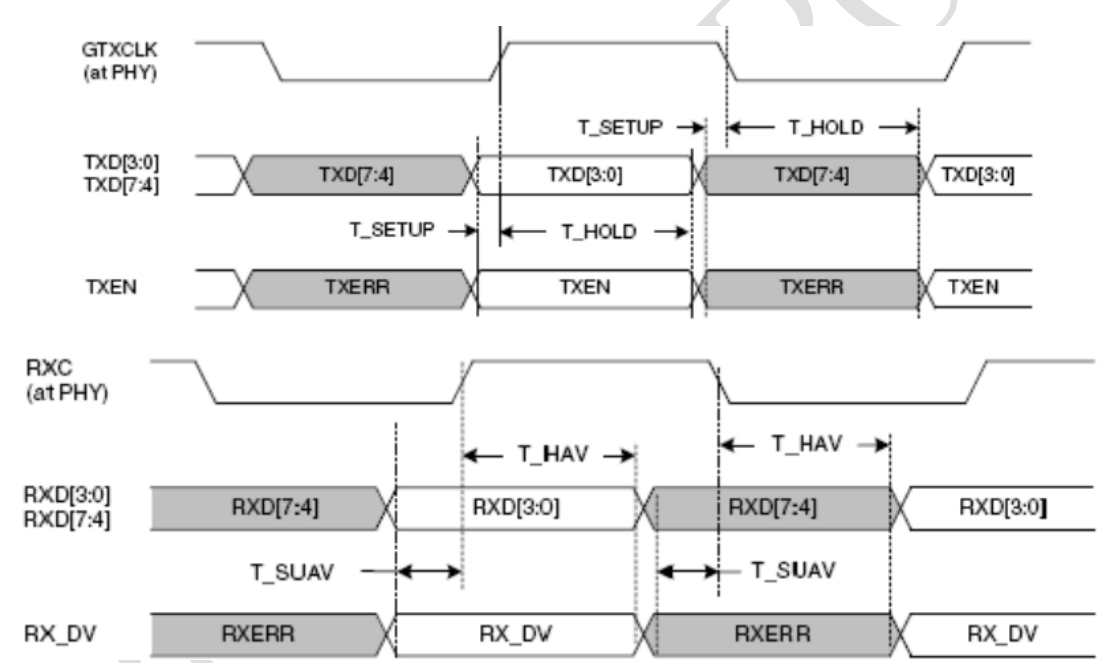
RGMII 即 ReducedGMII，是 GMII 的简化版本，将接口信号线数量从 24 根减少到 14 根，时钟频率仍旧为 125MHz，TX/RX 数据宽度从 8 变为 4 位。RGMII 接口信号连接关系及各信号的介绍如下。



信号	方向	位宽	说明
TX_CTL	O	1	该信号线上传送 GMII 接口中的 TX_EN 和 TX_ER 两种信息，在 TX_CLK 的上升沿发送 TX_EN，下降沿发送 TX_ER。
TX_CLK	O	1	发送参考时钟，1000Mbps 速率下，时钟频率为 125MHz，TX_CLK 时钟是由 MAC 提供的，与 GMII 接口中方向一致。
TXD	O	4	在时钟 TX_CLK 的上升沿发送 GMII 接口中的 TXD[3:0]，在时钟 TX_CLK 的下降沿发送 GMII 接口中的 TXD[7:4]。
RX_CTL	I	1	该信号线上传送 GMII 接口中的 RX_DV 和 RX_ER 两种信息，在 RX_CLK 的上升沿传输 RX_DV，下降沿传输 RX_ER。
RXD	I	4	时钟 RX_CLK 的上升沿传输 GMII 接口中的 RXD[3:0]，在时钟 RX_CLK 的下降沿发送 GMII 接口中的 RXD[7:4]。
RX_CLK	I	1	接收数据参考时钟，1000Mbps 速率下，时钟频率为

		125MHz。RX_CLK 由 PHY 侧提供的。
--	--	---------------------------

RGMII 接口为了保持 1000Mbps 的传输速率不变，RGMII 接口在时钟的上升沿和下降沿都采样数据。在参考时钟的上升沿发送 GMII 接口中的 TXD[3:0]/RXD[3:0]，在参考时钟的下降沿发送 GMII 接口中的 TXD[7:4]/RXD[7:4]。RGMII 同时也兼容 100Mbps 和 10Mbps 两种速率，此时参考时钟速率分别为 25MHz 和 2.5MHz。TX_EN 信号线上传送 TX_EN 和 TX_ER 两种信息，在 TX_CLK 的上升沿，下降沿发送 TX_ER；同样的，RX_DV 信号线上也传送 RX_DV 和 RX_ER 两种信息，在 RX_CLK 的上升沿传输 RX_DV，下降沿传输 RX_ER。具体时序如下。关于 RTL8211 PHY 芯片具体时序参数与配置的 TXDLY 和 RXDLY 管脚的电平有关，详细可参见 RTL8211 芯片手册。



MII GMII RGMII 接口对比

上述分别介绍了 MII、GMII、RGMII 接口，接下来对比分析这三种接口的特点。

	MII	GMII	RGMII
以太网速率	100Mbps/10Mbps	1000Mbps	1000M/bps
数据线	收发各 4 位	收发各 8 位	收发各 4 位
信号是否时钟双沿传输	单沿传输（SDR）	单沿传输（SDR）	双沿传输（DDR）
TX_CLK 方向	PHY -> MAC	MAC -> PHY	MAC -> PHY
TX_ER	有	有	编码进了 TX_EN

RX_ER	有	有	编码进了 RX_EN
COL、CRS 信号	有	有	无

几个要点说明：

1. MII 接口时，TX_CLK 由 PHY 芯片提供给 MAC，其他接口模式下均由 MAC 提供给 PHY 芯片。
2. 除 GMII 接口外，其他接口均使用 4bit 数据线用来收发数据。
3. RGMII 接口的信号传输采用 DDR 接口，也就是在 CLK 的上升沿和下降沿各传输一次数据。
4. RGMII 接口将 TX_ER 和 RX_ER 信号编码进了 TX_CTL 和 RX_CTL 信号中，不再使用独立的信号线。

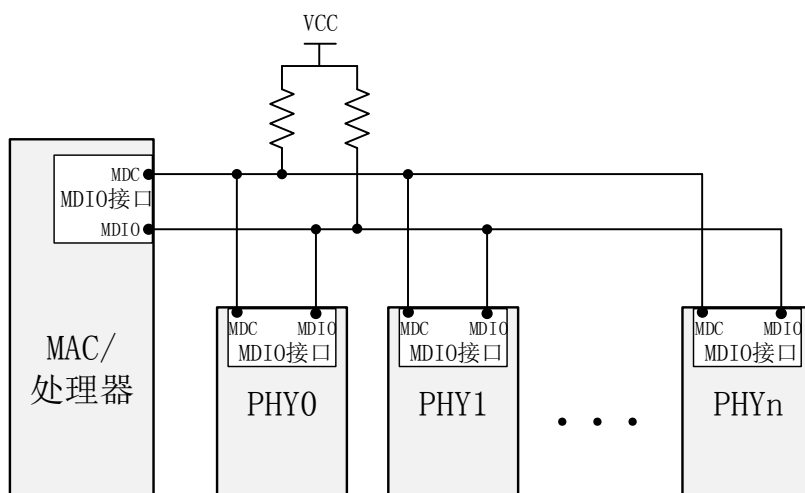
PHY 管理接口 MDIO 介绍

MDIO 接口介绍

PHY 芯片根据不同的需求，会有不同的工作模式，例如需要设置其以太网通信速率为千兆、百兆、十兆或自动协商，而处理器也需要知道其连接状态，例如需要知道网络是否已经连接通，以及当前的链接速率是多少。为了实现这些功能，IEEE 在以太网标准 IEEE 802.3 中使用若干条款定义了用处理器用来管理 PHY 芯片的标准控制接口——MDIO。

MDIO 是一种简单的双线串行接口，将管理器件(如 MAC 控制器、微处理器)与具备管理功能的 PHY 芯片相连接，从而控制收发器并从收发器收集状态信息。可收集的信息包括链接状态、传输速度与选择、断电、低功率休眠状态、TX/RX 模式选择、自动协商控制、环回模式控制等。除了拥有 IEEE 要求的功能之外，收发器厂商还可添加更多的信息收集功能。

和大家熟悉的 I2C 接口相似，MDIO 总线也支持多设备模式，即一个 MDIO 总线上最多可连接 32 个 PHY 芯片的 MDIO 接口，通过在传输过程中的 PHY 器件地址字段传输不同的器件地址值来指定具体操作哪个 PHY 芯片。



每个 PHY 芯片都有一个硬件地址，该地址为 5 位，由固定部分和可变部分组成，可变部分可以在芯片应用电路中通过对地址设置脚上下拉到电源或 GND 设置。若多个 PHY 芯片同时连接在同一个 MDIO 总线上，需要设置不同的 PHY 器件地址以示区分。

每个 PHY 芯片中都有最多 32 个寄存器可供读写（32 个寄存器，需要 5 位寄存器地址），包括由 IEEE 802.3 规范定义的前 16 个标准功能寄存器和后续最多 16 个由 PHY 芯片厂家自定义的功能寄存器。所有 PHY 芯片的前 16 个寄存器功能定义都必须相同，后续 0~16 个寄存器的功能由 PHY 芯片厂家根据实际需求自定义。

MDIO 总线包括两个信号：MDC 和 MDIO。MDC 是管理数据的时钟信号，由管理器（如处理器或 MAC）输出给 PHY 芯片，最高速率可超过 5MHz。MDIO 是管理数据的输入输出双向接口，数据是与 MDC 时钟同步的。

MDIO 时序介绍

介绍了 MDIO 接口之后，要想正确的使用 MDIO 接口来管理 PHY 芯片，还需要了解 MDIO 接口的时序流程。MDIO 的工作流程为：

- 1、空闲：MDIO 接口在没有传输数据的空闲状态（IDLE）数据线 MDIO 处于高阻态。
- 2、开始：MAC 或处理器驱动 MDIO 产生一个 2bit 的开始标识码(01)，标志着一次读、写传输的开始。

- 3、读写标记：MAC 或处理器驱动 MDIO 产生一个 2bit 数据来标识本次操作是读操作(10)还是写操作(01)。
- 4、PHY 器件地址：MAC 或处理器驱动 MDIO 产生 5 位的 PHY 器件地址数据，用来指明此次操作是针对的具体那一个 PHY 芯片。
- 5、PHY 寄存器地址：MAC 或处理器驱动 MDIO 产生 5 位的 PHY 寄存器地址数据，用来指明此次操作是针对的 PHY 中具体哪一个寄存器。
- 6、MDIO 控制权切换：在读操作中，由于在 PHY 移出寄存器中的数据时 MDIO 信号将由 PHY 芯片接管驱动，既 MDIO 信号存在一个驱动切换的情形，为了避免切换过程中 MAC 或处理器驱动和 PHY 芯片同时驱动 MDIO 信号以出现冲突干扰，在传输完 PHY 寄存器地址后，需要传输 2bit 的 TA（Turnaround）信号，传输该信号时，MAC 或处理器 MDIO 信号为高阻状态，即不驱动 MDIO 信号至低电平，对于读操作，PHY 芯片将在这 2bit 信号的第 2 个 bit 时将 MDIO 信号拉低以标记读出数据传输传输。同时，这第 2 个 bit 还将作为类似于 I2C 总线中的应答位，如果在读操作中，该 bit 没有被拉低，则表明读 PHY 芯片操作失败。对于写操作，第一个 bit 时 MAC 或处理器和 PHY 芯片都保持为高阻状态，MDIO 被上拉电阻拉为高电平，在第 2 个 bit 时由 MAC 或处理器将该位拉低。
- 7、读写数据：MDIO 串行读出/写入 16bit 的寄存器数据。
- 8、IDLE 状态：MDIO 再次进入高阻状态，对于部分 PHY 芯片，MDC 将继续产生至少 7 个时钟周期之后方可停止。

注意，在空闲和开始状态之间，MDIO 需要保持高阻态至少 32 个 MDC 时钟周期，该段称为前导信号（preamble）。

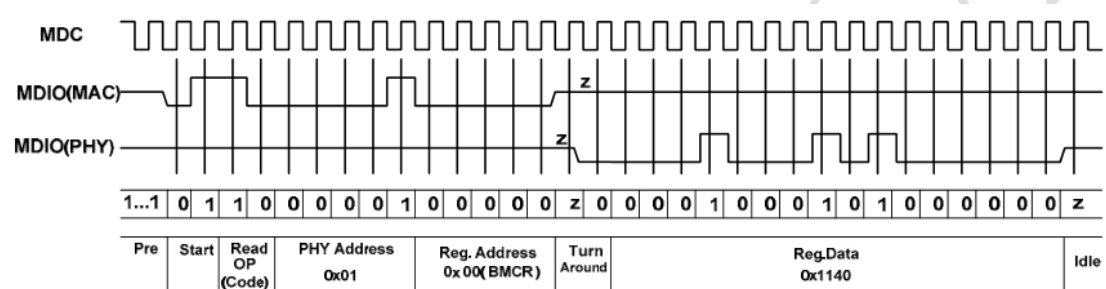
下表为一次读写操作的数据帧格式：

	Preamble (32bit)	ST (2bit)	OP (2bit)	PHYAD (5bit)	REGAD (5bit)	TA (2bit)	DATA (16bit)	IDLE
Read	1...1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD	Z
Write	1...1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD	Z

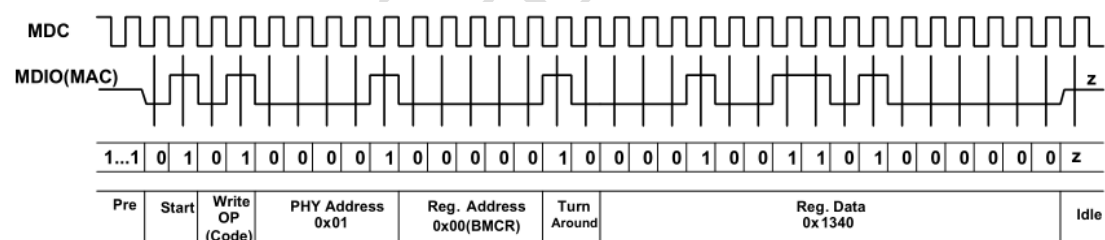
其中各段的意义和功能如下表所示：

名称	功能和意义描述
Preamble	mac 端在 mdc 时钟下，通过 mdio，发送 32 个连续的逻辑 1
ST	帧起始，用 01 来表示
OP	操作符，用 01 来表示写操作，用 10 来表示读操作
PHYAD	物理地址，最多可以将 8 个物理连接到一个 MAC，高 2bit 为 0
REGAD	寄存器地址，5 位宽度，用于设置当前操作 PHY 的 32 个寄存器中的哪一个
TA	这两位是用来指示方向的，Z 是高阻态
DATA	总线上传输的 16 位数据
IDLE	无传输时的状态，该状态时候 PHY 的 MDIO 设置为高阻态，其电平将由 MDIO 引脚上的上拉电阻上拉到高电平

下图为 MDIO 接口读和写操作的具体时序图：



MDC/MDIO 读时序



MDC/MDIO 写时序

了解了 MDIO 接口的基本时序，就可以对各种 PHY 芯片进行读写操作了。具体在什么模式下需要对什么寄存器进行操作，需要根据实际情况查询该 PHY 芯片的寄存器功能描述获知。网络上有一篇名为“以太网 PHY 寄存器分析”的文章，较为详细的介绍了 PHY 中各个寄存器的功能，可以作为参考。

MDIO 接口注意事项

1. 部分 PHY 芯片（如 RTL8211）要求在一次读写操作完成，转为 IDLE 状态

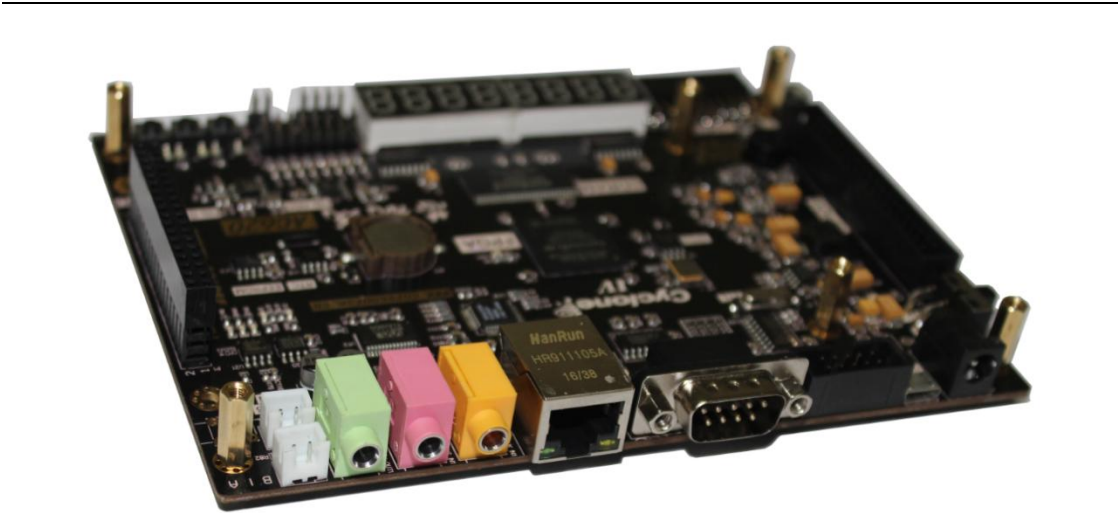
之后，还需要 MDC 信号持续至少 7 个时钟周期以确保 PHY 芯片正常完成该操作。

2. PHY 芯片的 MDIO 引脚需要连接至少 1.5K Ohm 的上拉电阻。
3. 在 PHY 芯片成功 Link 之前，不得向 PHY 芯片提供发送使能信号，即 TX_EN 信号在 PHY 芯片没有成功 Link 之前不得出现高电平，否则可能会影响 PHY 芯片的自动协商和工作模式设置。
4. PHY 芯片一般都有一个硬件复位引脚 reset，一般 PHY 芯片都会做出说明，在该复位信号释放后需要等待至少多久的时间（例如，对于 RTL8211，这个值为 30ms）才能开始使用 MDIO 接口对 PHY 芯片进行读写操作。

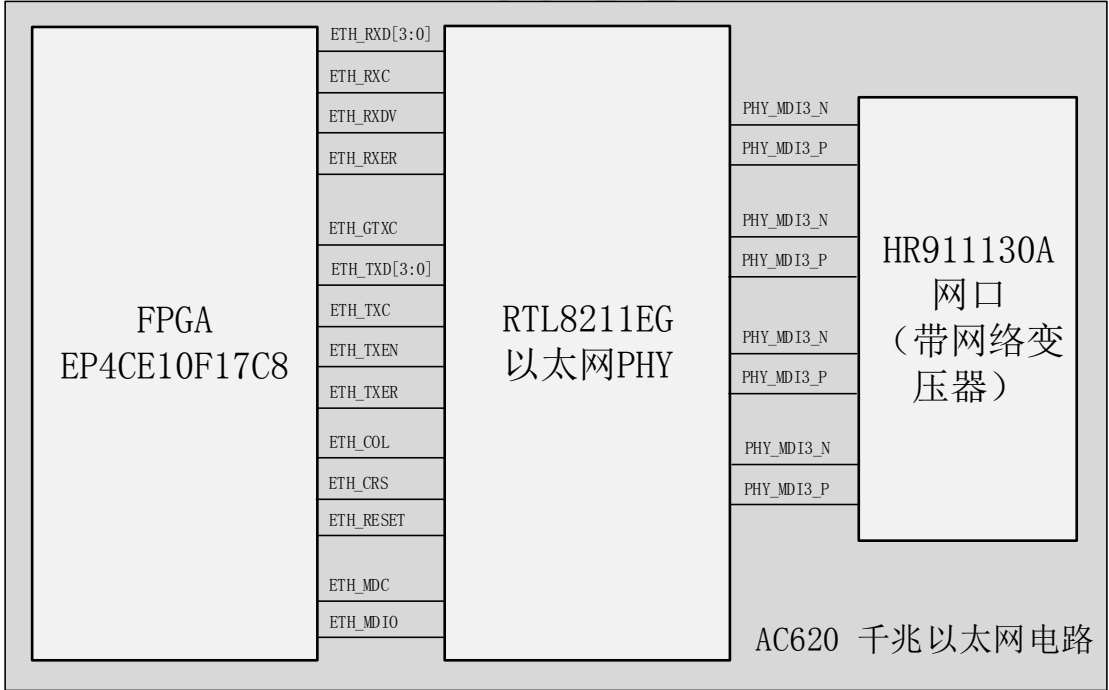
AC620 FPGA 以太网电路介绍

AC620 开发板通过一片 Realtek 的 RTL8211 以太网 PHY 提供对以太网连接的支持，RTL8211 是一片 10M/100M/1000Mbps 自适应以太网收发器，提供 MII/GMII/RGMII 接口的 MAC 连接。在 Cyclone IV E 器件中，可以调用三速以太网 IP 核（MAC），配合 NIOS II 处理器实现完整的以太网连接和以太网传输协议。或者用户使用 Verilog 编写的自定义用户逻辑来实现简单高效的点对点以太网连接与数据传输。在芯路恒科技出品的众多 FPGA 板卡中，都用到了该芯片作为以太网物理层。这里，先介绍完整的 RTL8211 应用模式的电路设计，再根据具体的板卡应用情况分别介绍。

下图为 AC620 开发板上的网口示意图



AC620 开发板上，提供了一个千兆以太网接口，该接口由千兆以太网 PHY 和网络变压器接口组成。当需要发送以太网数据时，FPGA 把数据发送给 PHY 芯片，PHY 芯片将数据编码后，通过网络变压器将数据加载到网线上。数据经由网络传递到接收方。而远端发送过来的数据，经由网线传递给网络变压器，网络变压器的输出连接到 PHY 芯片上，PHY 芯片对信号进行解码后，得到实际的数据，然后将数据传递给 FPGA 芯片。FPGA 实现千兆以太网数据传输的功能框图如下所示：



PHY 芯片 RTL8211EG

AC620 开发板上使用的是一款功能齐全，性能优异，性价比高的

10/100/1000Mbps 以太网 PHY 芯片——RTL8211EG。RTL8211EG 是一款支持 GMII、RGMII、MII 接口的以太网物理层收发器，能够工作在 10M、100M 或 1000M Base 模式，对 MAC 层可提供 GMII、RGMII、MII 三种接口模式，并提供了标准的 MDIO 管理接口与处理器相连。

注意，大家自己在设计基于这个芯片的电路时需要留意。RTL8211EG 芯片的 EG 后缀比较重要，该芯片有很多型号，根据尾缀的不同其封装和接口功能也有差异，例如 RTL8211EG 是 64Pin 封装，支持 GMII 接口，而 RTL8211F 则是 40pin 引脚，仅支持 RGMII，不支持 GMII 接口，例如在我们的 SOC FPGA 开发板上使用的就是 RTL8211F 系列的 PHY 芯片，使用 RGMII 接口。

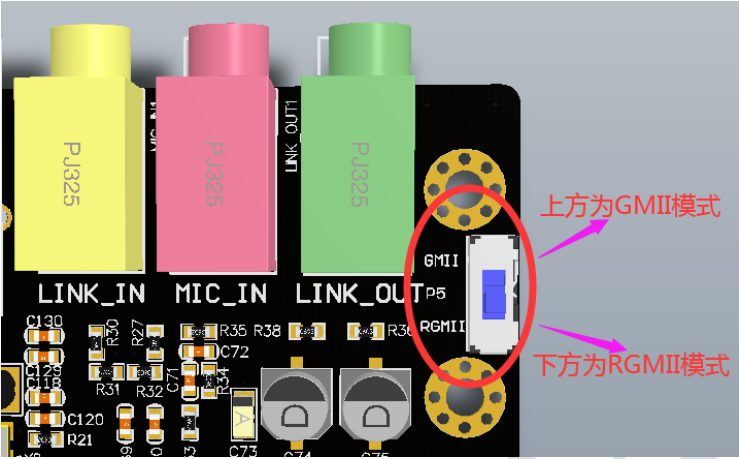
AC620 上 PHY 的接口模式

在 AC620 开发板上，由于开发板本身集成功能较多，FPGA 的 I/O 资源紧张，而 GMII 接口应用时，接收和发送都需要占用 8 根信号线，会占用较多的 I/O 资源，因此在设计电路时接收和发送各使用了 4 根信号线。所以，在 AC620 开发板上以太网电路不对 GMII 接口提供支持，仅支持 RGMII 接口和 MII 接口。事实上，这种情况虽看似有所舍，但也是一个趋势。在现在的电子系统设计中，基本都已经使用 RGMII 接口来作为 PHY 与 MAC 连接的方式了，很少还有使用 GMII 接口来作为通信接口的。在 FPGA 中，只要设计出了 RGMII 接口的逻辑，那么 GMII 接口也就是已经先行存在的了，GMII 接口和 RGMII 接口之间，只差了一个双数据速率（DDR）接口转换模块，GMII 接口的实现比 RGMII 接口少一个步骤，更加的简单。所以从学习角度来说，也没有任何的损失。

RTL8211EG 芯片具体工作在什么接口，是由其硬件管脚 COL 在上芯片上电时刻的高低电平状态决定的。芯片在上电时会检测该引脚的状态，并根据该引脚的状态确定工作模式，若上电时该引脚为高电平，则 RTL8211 将工作在 RGMII 模式，若上电时该引脚为低电平，则 RTL8211 将工作在 GMII 模式，

需要说明的是，如果需要 RTL8211EG 芯片工作在常规的 MII 接口，则需要先通过拉低 COL 引脚设置 RTL8211EG 的接口为 GMII 模式，然后再通过 MDIO 接口设置 RTL8211EG 芯片的模式寄存器来强制其工作在 100M Base 模式。

为了方便用户快速切换 RGMII 模式和 GMII 模式，AC620 开发板上设计了一个拨码开关来实现该功能。该开关位于开发板右上角，拨到上方则选择 GMII 模式，拨到下方则选择 RGMII 模式，如下图所示：



再次说明一下，通过拨码开关选择 GMII 模式的目的是不是要真的使用 GMII 接口，而是为了兼容标准的 MII 百兆接口，因为 RTL8211EG 只有在 GMII 模式下才能通过配置寄存器运行在标准的 MII 接口模式下。

RTL8211 芯片器件地址

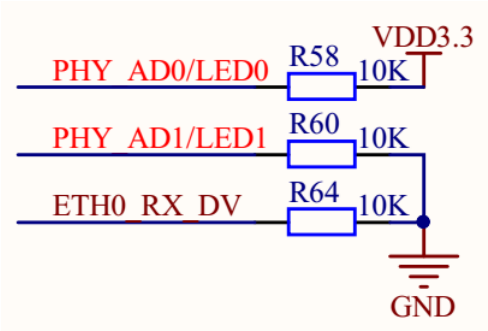
以太网物理层芯片都有一个器件地址，该器件地址就是在介绍 PHY 管理接口 MDIO 时所说的 PHY 器件地址，该器件地址用来由 MDIO 主机（如 MAC 或处理器）寻址 MDIO 总线上连接的指定的 PHY 芯片。

对于 RTL8211EG 芯片，该地址分为固定部分和硬件可设定两部分。器件地址共有 5 位，其中高 2 位为固定的 00，低三位通过三个地址设置脚在芯片上电或复位时候设置， 这三位分别对应 RTL8211EG 芯片的 50、51、18 脚。需要说明的是，这三个脚并不是单纯的器件地址设置脚，而是复用功能的。对于器件地址的设置仅在 RTL8211EG 芯片刚从复位状态恢复的那一小段时间起作用，而后就又恢复为正常的功能模式。下表为这三个脚功能说明。

功能 1（上电时功能）		正常工作时功能
Pin No.50	PHY_ADDR0	LED0，指示收发状态，在收发过程中闪烁
Pin No.51	PHY_ADDR1	LED1，指示连接状态，点亮时表明网络已经连通
Pin No.18	PHY_ADDR2	RX_DV，接收有效标志信号。

在 AC620 开发板上，这三个脚通过三个电阻分别上/下拉到了指定电平，具

体来说，PHY_ADDR1 和 PHY_ADDR2 通过电阻下拉到了低电平，PHY_ADDR0 通过电阻上拉到了高电平，所以 AC620 开发板上的 PHY 器件地址为 00001。



PHY 与 FPGA 连接管脚说明

AC620 开发板上与 RTL8211EG 芯片的引脚连接如下表所示：

功能标号	FPGA 管脚	功能标号	FPGA 管脚
ETH_TXEN	PIN_C14	ETH_RXDV	PIN_A15
ETH_TXD3	PIN_B11	ETH_RXD3	PIN_B13
ETH_TXD2	PIN_A12	ETH_RXD2	PIN_A14
ETH_TXD1	PIN_B12	ETH_RXD1	PIN_A13
ETH_TXD0	PIN_C11	ETH_RXD0	PIN_E10
ETH_TXC	PIN_D11	ETH_RXC	PIN_M15
ETH_GTXC	PIN_B14	ETH_MDIO	PIN_D12
ETH_RESET	PIN_D14	ETH_MDC	PIN_E11

其他未列出管脚均为无必要与 FPGA 连接的信号。

以太网通信协议解析与实现

以太网（MAC）帧协议介绍

以太网帧概述

以太网是目前最流行的一种局域网组网技术（其他常见局域网组网技术还有令牌环局域网、无线局域网、ATM 局域网），以太网技术的正式标准是 IEEE 802.3 标准，它规定了在以太网中传输的数据帧结构，如下图所示。

前同步码	分隔符	目的地址	源地址	长度/类型	数据和填充	校验字段
------	-----	------	-----	-------	-------	------

Preamble	SFD	DES_MAC	SRC_MAC	length/type	Data and Pad	FCS
7 字节	1 字节	6 字节	6 字节	2 字节	46~1500 字节	4 字节

每个字段的功能和说明如下表所示：

字段	字段长度 (字节)	说明
前导码 (Preamble)	7	7 个字节 的 0x55，用于帧的同步
帧开始符 (SFD)	1	值为 0xd5，表明下一个字节为目的 MAC 字段
目的 MAC 地址	6	指明该以太网帧的接收方网卡地址
源 MAC 地址	6	标记该以太网帧的发送方网卡的地址
长度/类型 (Length/ Type)	2	当这两个字节的值小于 1518 时，那么它就代表其后数据字段的长度；如果这两个字节的值大于 1518，则表示该以太网帧中的数据属于哪个上层协议（例如 0x800，代表 IP 数据包；0x806，代表 ARP 数据包等。
数据和填充 (Data and Pad)	46~1500	高层的数据，通常为 3 层协议数据单元。对于 UDP、TCP/IP 协议，该部分为 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，使用 CRC32 计算方法。

前导码和分隔符

在物理层上看，一个完整的以太网帧有 7 个字段。事实上，前两个字段前导码和分隔符并不能算是真正意义上的以太网数据帧，这 8 个字节的值在任何以太网帧中都是一样的，7 字节的前导码值为 0x55，1 字节的分隔符值为 0xD5。前导码存在的意义是为了标记一帧以太网帧即将开始传输，分隔符存在的意义是指示前导码传输完毕，接下来将要传输的就是正式的以太网帧的有用部分。

前导码的值为 7 字节的 0x55，0x55 用二进制表示就是 01010101b，也就是说前导码的意义就是驱动网络链路电平开始交替翻转，以标记帧的开始。而分隔符的值为 0xD5，0xD5 用二进制表示就是 11010101b，由于以太网 PHY 在将数据编码为以太网物理链路层的位传输信号时是从低位开始编码的，所以从信号链路的电平来看，就是先连续的产生 62 位 0 和 1 交替变化的电平，最后再以 2 位连续的高电平来标记以太网数据帧部分的到来。

MAC 地址

按照规定，每个连接到互联网的网卡都必须要有个唯一的物理地址，该地址称为该网卡的 **MAC 地址**，这是一个 **48 位**的数据，如同人的身份证一样唯一。数据在网卡之间传递时，就是依靠这个 **MAC 值**来确定网络上的数据是否是传给该网卡的，如果数据中的 **MAC 地址值**与该网卡的 **MAC 地址值**相同，则该网卡会接收网络上的数据内容，否则就会忽略。

事实上，**MAC 地址**的对应并不是唯一的，在网络传输中，总共有三种对应方式，分别为单播地址、多播地址和广播地址。单播地址通常与一个具体网卡的 **MAC 地址**相对应，也就是一对一，唯一指定一个数据帧的接收方，它要求第一个字节的 **bit0**（即最先发出去的位）必须是 **0**；多播地址要求以太网帧中目的 **MAC 地址**第一个字节的 **bit0** 强制设置为 **1**，这样，在网络中多播地址不会与任何网卡的 **MAC** 相同，此时网卡依旧会接收该数据帧，然后交由上层 **IP 协议**来解多播地址，也就是说多播数据可以被很多个网卡同时接收；而广播地址的所有 **48 位**全为 **1**（即 **FF-FF-FF-FF-FF-FF**），同一局域网中的所有网卡可以接收广播数据包。

每个电脑的 **MAC 地址**可以通过在 **cmd** 窗口中使用 **ipconfig -all** 命令查看，其中物理地址值就是该网卡的 **MAC 地址**，如下图所示：

```
管理员: C:\Windows\system32\cmd.exe

C:\Users\Administrator>ipconfig -all

Windows IP 配置

主机名 . . . . . : USER-20190611AP
主 DNS 后缀 . . . . . :
节点类型 . . . . . : 混合
IP 路由已启用 . . . . . : 否
WINS 代理已启用 . . . . . : 否

无线局域网适配器 无线网络连接:

连接特定的 DNS 后缀 . . . . . :
描述 . . . . . : Intel(R) Dual Band Wireless-AC 8260
物理地址. . . . . : E4-73
DHCP 已启用 . . . . . : 是
自动配置已启用 . . . . . : 是
本地连接 IPv6 地址 . . . . . : fe80::692b:e95d:58b4:a6b6%19<首选>
IPv4 地址 . . . . . : 192.168.43.193<首选>
子网掩码 . . . . . : 255.255.255.0
获得租约的时间 . . . . . : 2020年3月6日 18:59:04
租约过期的时间 . . . . . : 2020年3月6日 20:29:04
```

每个联网设备的 MAC 地址必须是全世界唯一的，为了实现这种唯一性，所有的设备厂商要想让自家产的设备作为标准设备并能够联网，都需要向美国的 IEEE 申请，设备厂商申请的不是一个个具体的 MAC 地址值，而是一个 MAC 地址段。48 位的 MAC 地址由两部分组成，其中前 24 位由 IEEE 批复提供，后 24 位则由设备厂商自行定义。

在我们基于 FPGA 的以太网实验中，关于开发板的 MAC 地址，经常有读者会问这个地址是怎么确定的，这里简单做个解释：

1. 对于需要公开销售连接到互联网上的设备，例如计算机，路由器，必须有全球唯一的 MAC 地址，这个地址需要设备制造商向 IEEE 组织申请并购买。
2. Altera、Xilinx 等制造官方 FPGA 板卡时，由于板载网口，也都会向 IEEE 组织购买 MAC 地址端。
3. MAC 地址主要面向直接连接互联网的设备，要求网络上所有设备的 MAC 地址唯一，目的是为了防止冲突。如果设备不接入互联网，只接入局域网，则只要局域网内所有的设备 MAC 地址不一样即可避免冲突。
4. 我们在设计基于 FPGA 的简易系统时，不需要设备接入公共互联网，一

般只直接对电脑传输数据，所以从防止冲突的角度考虑，只要不与接入的电脑的 MAC 地址冲突即可，所以我们可以任意找一个 MAC 地址使用。

5. 如果用户使用 FPGA 设计了能够接入公共互联网的设备，则需要自行购买 MAC 地址段并设置。
6. 所以我们实验中借用了 Altera 官方开发板的一个 MAC 地址进行实验。

数据和填充字段

本小节介绍数据和填充字段的内容和其长度限制的由来。

数据和填充字段内容

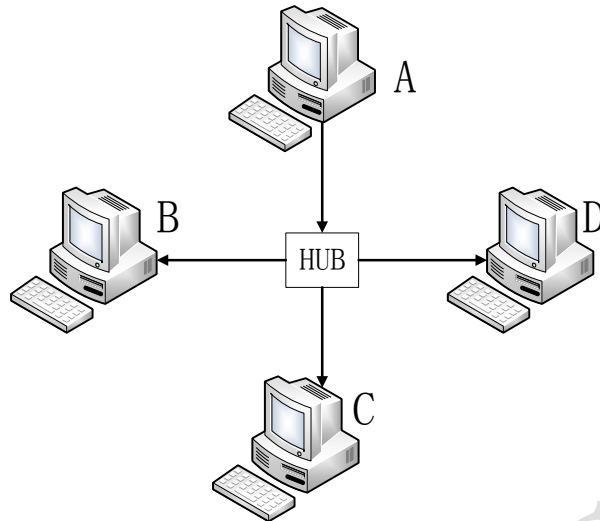
以太网帧的数据和填充字段内容就是希望通过以太网帧传输的“数据”部分，当然，这个数据并不仅仅是用户数据，所谓用户数据，就是我们希望发送的数据内容，如聊天时发送的“你好，很高兴认识你”这句话就是用户数据。以太网帧中的数据部分一般都是另一个上层协议，如 TCP/IP 协议。用户数据都是包含在该上层协议中的。

以太网帧的数据和填充字段要求该段数据长度至少为 46 个字节，而最多不超过 1500 个字节。也就是整个以太网帧的数据长度最小为 64 字节。那么该长度需求的原因是什么呢？为什么至少要 46 个字节，最多不超过 1500 个字节呢？要理解这个要求的由来，还要从以太网数据帧在网络上传输的原理说起。

46 字节最少数据长度限制

首先来讲，以太网传输是不可靠的，这意味着发送方并不知道接收方有没有收到自己发出的数据包，如果他发出的数据包发生错误，就需要重传。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但需要合理应对的。

假设一个局域网内有 A，B，C，D 四台网络设备，示意图如下。



其中，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。

以上图中设备 A 向设备 D 发送数据为例说明以太网中网卡之间的数据传输过程。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 主机在发送网络封包前，需要先对网络媒体进行监听，确认没有其他设备在使用后，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多份，然后传送给所有连接到此集线器的其他网络设备。也就是说，A 主机送出的数据，B、C、D 三个计算机都能够接收的到。但由于目标是 D 设备，因此 B 与 C 会将此讯框数据丢弃，而 D 设备则会抓下来处理；
3. 碰撞侦测 (Collision Detection): 该讯框数据附有检测能力，在 A 主机发送给数据过程中，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），此时这些讯框就会损毁，那么 A 与 B 就会各自随机等待一个时间，然后重新通过第一步再传送一次该讯框数据。

某一时刻，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。

根据 IEEE 定义的这个标准，最小长度是对最初的 10Mb/s 以太网设计的标准，该标准是将一个以太网的长度限制在 2500m 情况下定义的（通过 4 个中继器连接的 5 个 500m 的电缆段）。

根据电子在铜缆中传播速度约为 $0.77c$ ($231M\text{ m/s}$)，可得到 64 字节采用 10Mb/s 时的传输时间为 $64 \times 8 / 10,000,000 = 51.2\mu\text{s}$ ，那么最小尺寸的帧（64 字节）能在电缆中传输约为 $231 \times 51.2 \approx 12000\text{m}$ 。如果采用一条最长为 2500m 的电缆，从一个站到另一个站之间的最大往返距离为 5000m。以太网设计者确定最小帧长度考虑到安全因素（大于 2 倍的 2500m 长度限制），在完全兼容（和很多不兼容）的情况下，一个输出帧的最后位在所需时间后仍处于传输过程中，这个时间是信号到达位于最大距离的接收器并返回的时间。如果这时检测到一个冲突，传输中的站能知道哪个帧发生冲突，即当前正在传输中的那个帧。在这种情况下，该站发送一个干扰信号（高电压）提醒其他站，然后启动一个随机的二进制指数退避过程。

总结：最小数据帧的设计原因和以太网电缆长度（以太网的长度限制为 2500m）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，以太网设计者规定的争用期是 $51.2\mu\text{s}$ （10M 以太网传输 64byte 数据的时间，该时间是有考虑到充足的裕量）。

1500 字节最大数据帧长度限制

要理解以太网帧的数据段长度要求在 1500 字节以内，需要先了解 IP 数据包

的传输机制。

以太网 MAC 帧中传输的数据段中，出现频次最高的就是 IP 协议。IP 协议定义一帧 IP 数据帧最大可以为 65536 字节，加上 Ethernet Frame 头和尾，整个以太网帧的长度可以有 $65535+14+4=65553$ byte。如果在 10Mbps 以太网（数据位速率为 100ns）上，一次性传输 65553 字节（ $65553*8=524424\text{bit}$ ）的数据，将会占用共享链路超过 50ms（ $65553*8*100\text{ns}=52.4\text{ms}$ ），这将严重影响其它主机的通信，特别是对延迟敏感的应用是无法接受的。由于线路质量差而引起的丢包，发生在大包的概率也比小包概率大得多。

传输大包在丢包率较高的线路上是不明智的。但是如果选择一个比较小的长度，传输效率又不高，拿 TCP 应用来说，如果选择以太网长度为 218byte， $\text{TCP payload} = 218 - \text{Ethernet Header} - \text{IP Header} - \text{TCP Header} = 218 - 18 - 20 - 20 = 160$ byte 那有效传输效率= $160/218=73\%$ 而如果以太网长度为 1518，那有效传输效率= $1460/1518=96\%$ 通过比较，选择较大的帧长度，有效传输效率更高。而更大的帧长度同时也会造成上述的问题，可能是基于这些考虑，选择了一个折中的 1518 字节。对应的 IP packet 就是 1500 字节。

上述分析是基于最早的以太网是通过 Hub 或集线器来工作的事实，集线器在任意时刻只能有一台主机发送，这种共享方式发送效率很低。现代高速交换机则让每个连接交换机的主机工作在独占模式，带宽独享，可以同时收发。而且现在早已不是早期的 10Mbps 的带宽，而是 1000M、10000M，所以即使发送大包也不会影响别的主机，影响的只是交换机的接收和发送队列，既然发送大包效率要比小包效率高，而且特定的应用也有发大包的需求，比如 NFS 文件系统，既然物理限定已经一定程度上减弱或消失了，所以增大一次传输的数据帧长度也是合理的了。这是一个好主意，所以现代的网卡、交换机、路由器网络接口等都可以实现更大的 MTU，可以达到 >9000 字节的大小，这种远大于标准以太帧尺寸的帧就是我们常说的巨型帧（Jumbo Frame）。

校验序列

上面已经从各种物理角度考虑了以太网帧的各个字段的物理意义，就剩下最

后的校验序列了。校验序列又是个什么数据，为什么有这样的一个字段存在呢？

首先来说，以太网通信的时候数据最终都是通过网线从发送方传输到接收方，在这个传输路径中，网线会经过各种情况，遭受各种可能的电磁干扰，万一数据在网线上传输的过程中受到干扰，数据中的某一位或多个数据位因为干扰其电平状态发生了变化，例如 0 变 1、1 变 0，那么接收方在接收到该数据包时，收到的内容和发送方发送的内容就不再一致了，但是接收方本身是无法知道这些数据在传输过程中到底有没有受到干扰，有没有发生变化。这就存在接收数据错误的风险。

为了解决这个问题，以太网 MAC 帧发送方会该帧发送的所有内容，包括 MAC 帧头（包括目的 MAC 地址、源地址、长度/类型，不含前导码和分隔符）、数据和填充字段采用一种特殊的算法（CRC32）进行计算，得到一个 32 位的数，附加在数据和填充字段的结尾发送。而接收方接收到一帧 MAC 帧时，也会对该帧的 MAC 帧头和数据填充字段使用完全相同的算法进行计算，得到一个 32 位的数，并将这计算得到的 32 位的数据与接收到的数据帧的最后 32 位数据进行比较，若相同则表明接收和发送的数据内容完全一致，若不同则意味着该帧数据中肯定有至少 1 位数据的值在传输过程中发生了变化，整个这一帧内容都会被直接舍弃。

总结

至此，对于以太网 MAC 帧的介绍就结束了。可以看到，以太网 MAC 帧的结构还是非常简单的，重点是理解整个以太网 MAC 帧中各个字段的含义。接下来，就可以编写相应的网络实现协议逻辑了。

在实现网络协议时，最重要的一点就是要了解和掌握使用的接口方式。因为不管编写的协议逻辑如何，最终都是要通过对应的接口输出或者输入。目前，千兆以太网接口大多采用 RGMII 接口实现。下节内容，我们将先介绍 RGMII 接口的实现，为后续实现以太网数据的接收和发送打下基础。

RGMII 与 GMII 转换电路设计

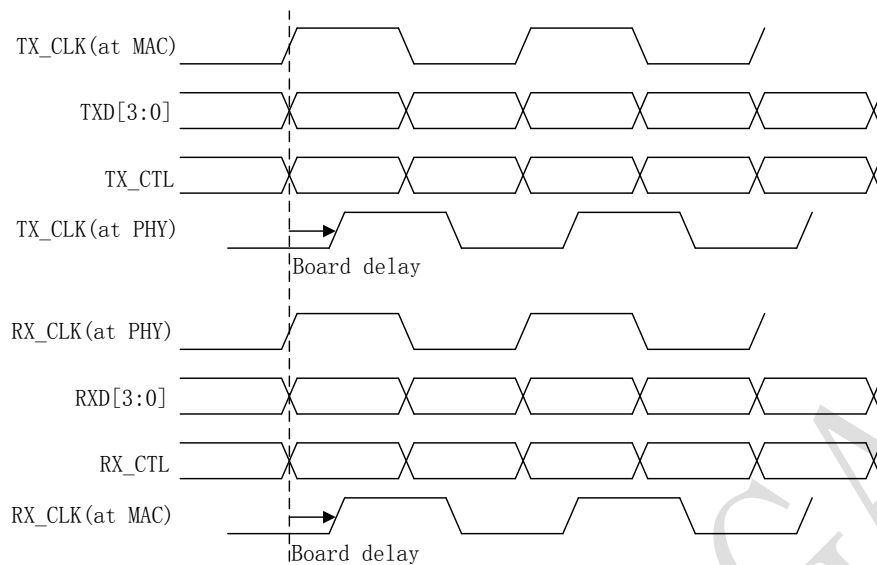
在上一节的内容中，我们了解了 MAC 帧的基本内容。在了解了以太网 MAC 帧的结构之后，相信读者都有一定的兴趣和自信，能够编写出对应的以太网帧发送和接收逻辑，但是在这之前，还需要先介绍下以太网 PHY 与 MAC（PHY）的连接实现方法，解决了接口问题，才能编写对应的网络协议实现逻辑。

RGMII 是 IEEE 802.3z 标准中定义的千兆媒体独立接口（Gigabit Medium Independent Interface）GMII 的一个替代品。相较于 GMII 接口，RGMII 接口可以在保证传输速率不变的情况下减少管脚数。RGMII 接口通过过在时钟的上升和下降沿分别传输 1 次数据，并对控制信号采用多路复用的方式来降低管脚数量的。

RGMII 数据在时钟的上升沿和下降沿均进行采样。通常，来自 RGMII PHY 的时钟和数据同时生成，即边缘对齐，因此必须在 PCB 上对时钟信号加入布线延迟来使得数据在时钟的上升沿时处于稳定状态。下表列出了 RGMII 接口的信号说明（以下信号方向皆是从以太网 MAC 层角度来看的）。

信号	I/O 方向	信号描述
TX_CLK	output	由 MAC 控制器输出的发送时钟
TXD[3:0]	output	TX_CLK 的上升沿传输 8 位数据中的低 4 位（bit 3:0） TX_CLK 的下降沿传输 8 位数据中的高 4 位（bit 7:4）
TX_CTL	output	TX_CLK 的上升沿传输控制信号中的 TX_EN 信号 TX_CLK 的下降沿沿传输控制信号中的 TX_EN ^ TX_ERR
RX_CLK	input	接收数据的参考时钟，由 PHY 输出
RXD[3:0]	input	RX_CLK 的上升沿传输 8 位数据中的低 4 位（bit 3:0） RX_CLK 的下降沿传输 8 位数据中的高 4 位（bit 7:4）
RX_CTL	input	RX_CLK 的上升沿传输控制信号中的 RX_DV 信号 RX_CLK 的下降沿沿传输控制信号中的 RX_DV ^ RX_ERR

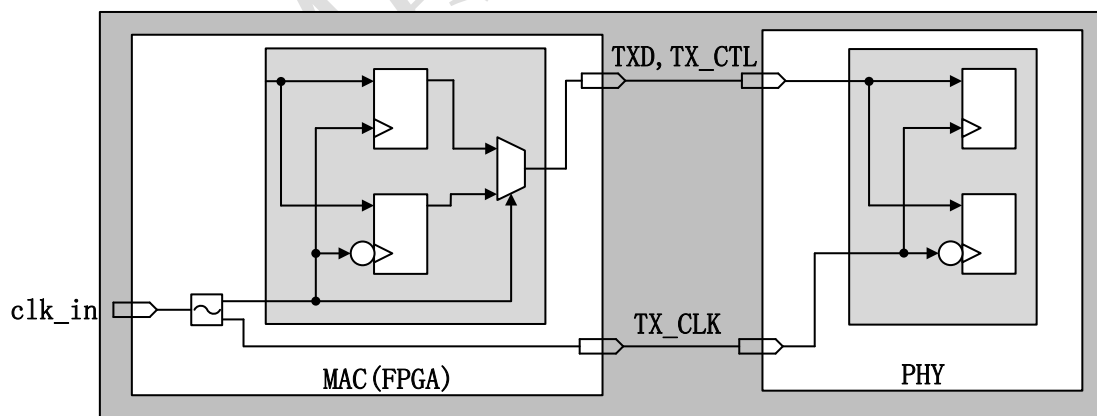
下图展示了时钟和数据边沿对齐情况下通过板级 PCB 对时钟加入延迟的方式实现目的端时钟和数据中心对齐的形式。



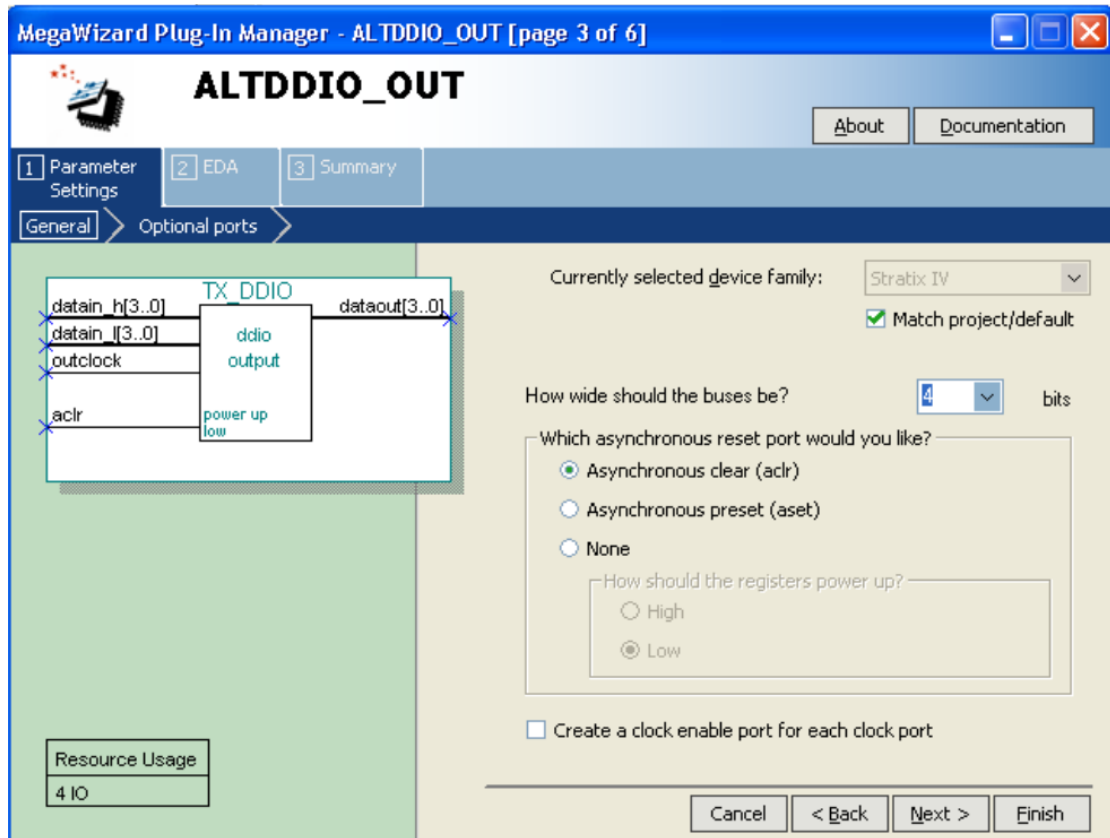
数据边沿对齐的形式使得 PCB 设计变得更加的复杂，所以，后来的 PHY 芯片都针对 RGMII 接口提供了可选的内部延迟电路，这些 PHY 芯片支持通过引脚管脚（例如 RTL8211）或配置寄存器（例如 KSZ9031、88E1512）的方式对时钟加入延迟，这样就可以降低 PCB 的布线要求。

RGMII 发送的 FPGA 实现方案

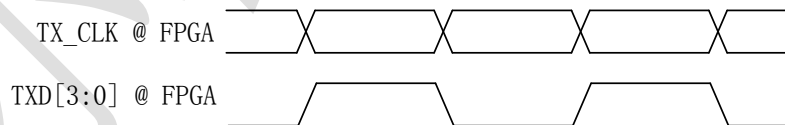
对于 FPGA 来说，实现 RGMII 接口的发送是一个非常直接的过程，整个发送逻辑框图如下图所示。



设计实现时，可以通过调用 Altera® ALTDDIO_OUT IP 核将该接口使用任何支持双数据速率的 I/O（DDIO）寄存器实现，如下图所示。

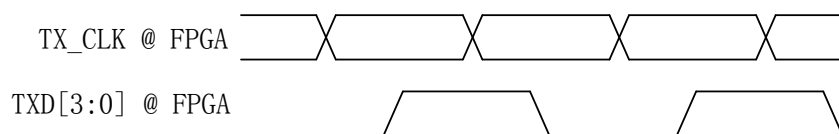


对于数据和时钟传输，大多数支持 RGMII 的 PHY 芯片都提供了一个对发送时钟 TX_CLK 添加延迟的选项。可以根据设计要求启用或禁用此选项。当启用延迟 PHY 设备内 TX_CLK 的选项时，FPGA 必须生成与数据和波形边缘对齐的时钟，如下图所示。

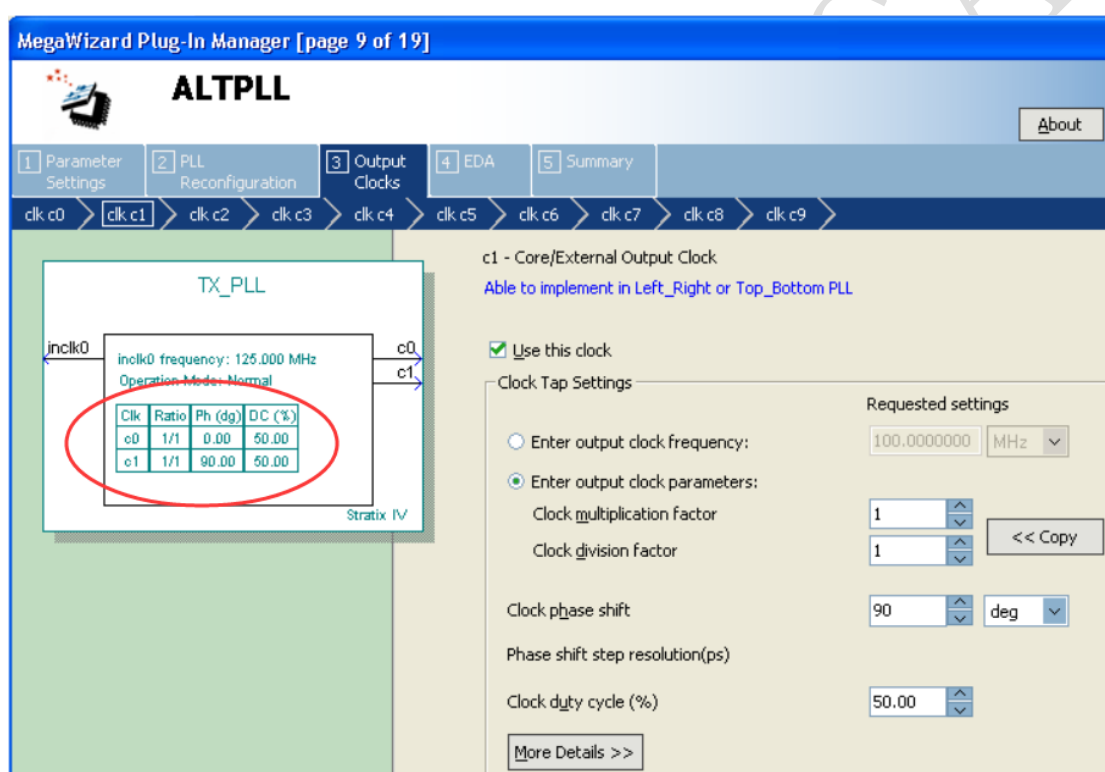


这种情况下，PHY 芯片会移动时钟的相位来捕获数据。

如果 PHY 的该功能被关闭，则 FPGA 必须对时钟信号进行一定的相位调整后作为 TX_CLK 输出，以使 TXD 和 TX_CLK 成中心对齐关系。当然，如果 PCB 上的 TX_CLK 信号线路本身就引入了一定的延迟，则需要将这一部分延迟考虑在内，再计算 FPGA 需要对 TX_CLK 调整的相位值，其传输波形如下图所示。



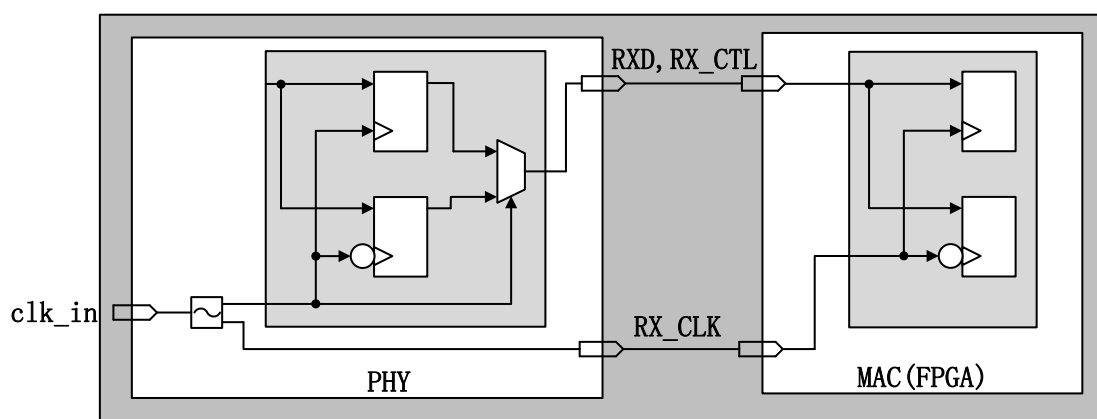
将源时钟与数据对齐的方法有多种。例如可以使用 PLL 产生 2 路相位相差 90 度的时钟信号，一路作为 DDIO_OUT 寄存器的工作时钟，一路作为 TX_CLK 输出。当然，如果考虑到 PCB 板上的 TX_CLK 相对于 TXD 的延迟值，该相位可以进一步调整以确保 TX_CLK 到达 PHY 时与 TXD 呈中心对齐关系。如下图所示：



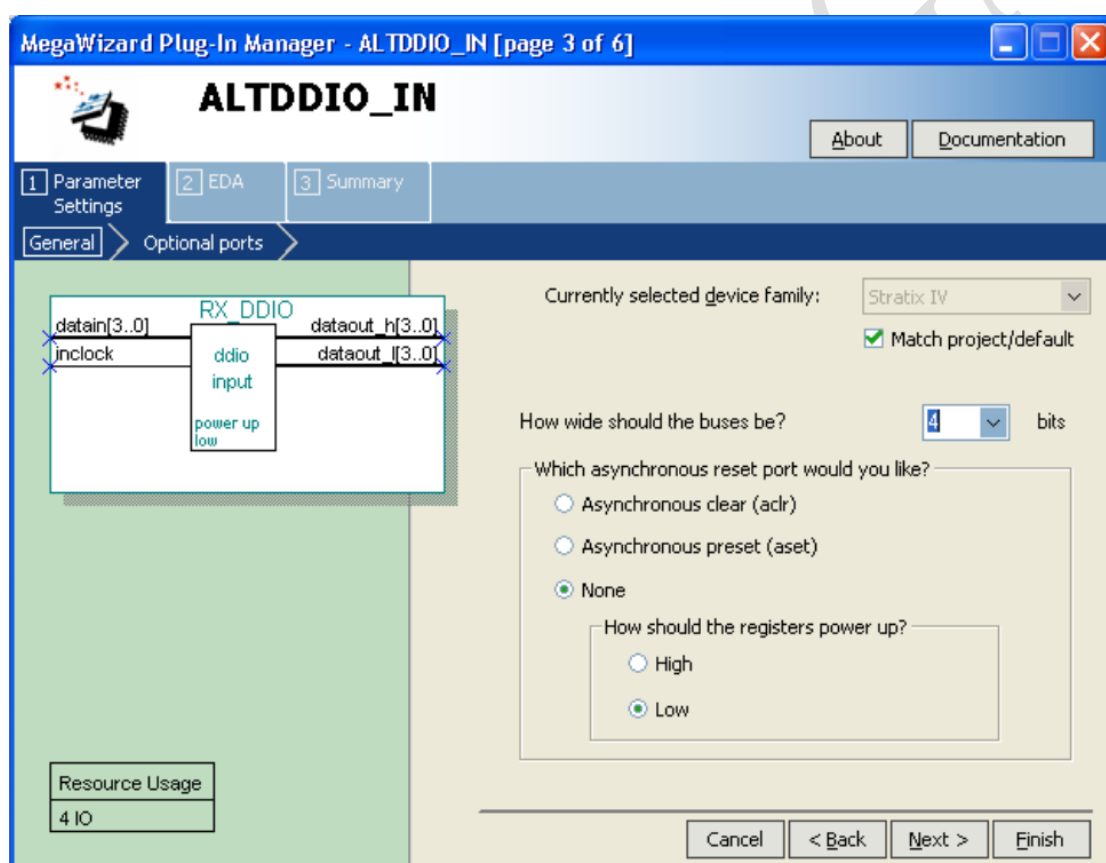
可以看到，通过 FPGA 侧实现 TX_CLK 和 TXD 的中心对齐需要使用到 PLL 资源，因此，使用 PHY 内部的 TX_CLK 延迟功能将降低对 FPGA 的资源占用。

RGMII 接收的 FPGA 实现方案

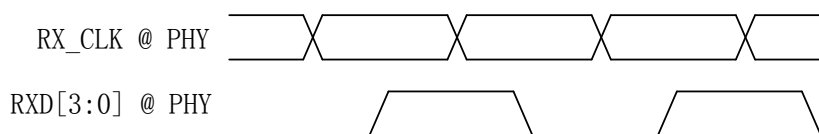
对于 FPGA 来说，实现 RGMII 接口的接收同样是一个非常直接的过程，整个接收逻辑框图如下图所示。



设计实现时，可以通过调用 Altera® ALTDDIO_IN IP 核将该接口使用任何支持双数据速率的 I/O（DDIO）寄存器实现，如下图所示。



为了让接收的数据能够更加容易的被 MAC 捕获，大多数支持 RGMII 的 PHY 芯片都提供了一个对接收时钟 RX_CLK 添加延迟的选项。可以根据设计要求启用或禁用此选项。当启用延迟 PHY 设备内 RX_CLK 的选项时，PHY 输出的 RX_CLK 与 RXD 会呈中心对齐的关系，如下图所示。在这种情况下，FPGA 就可以直接使用 RX_CLK 来捕获输入的数据，而不需要再对 RX_CLK 添加 PCB 板级延迟，或者是在 FPGA 内部对 RX_CLK 添加延迟。



如果 PHY 对 RX_CLK 添加延迟的功能被关闭（波形图如下图所示），则 FPGA 必须对 RX_CLK 时钟信号进行一定的相位调整后再用来捕获数据。这种情况，可以使用 FPGA 内的 PLL 将该信号进行一定的相位移动后再用来捕获数据。当然，让 RX_CLK 能够进入 PLL，必须要求该信号是从 FPGA 器件的专用时钟输入管脚或 DQS 脚输入。



可以看到，通过 FPGA 侧实现 RX_CLK 和 RXD 的中心对齐需要使用到 PLL 资源，因此，使用 PHY 内部的 RX_CLK 延迟功能将降低对 FPGA 的资源占用。推荐有条件的情况下尽量选择 PHY 内部增加延迟的方案。

使用 FPGA 实现 RGMII 接口

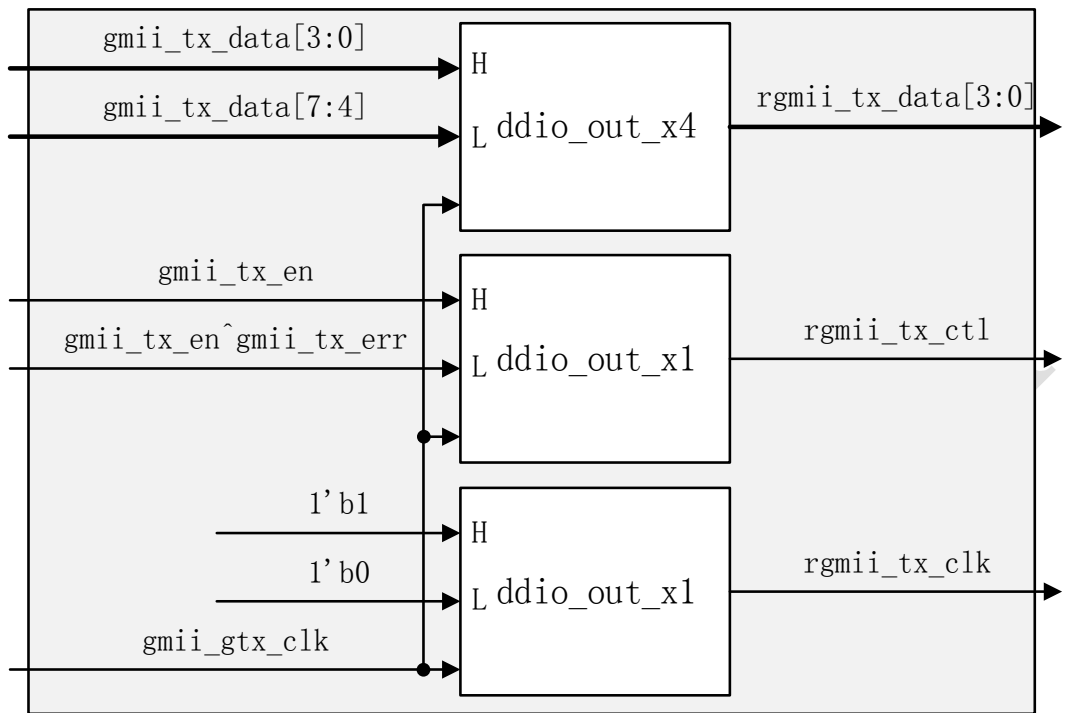
了解了 RGMII 在 FPGA 内部的实现原理和方法后，实现就非常的方便了。

RGMII 接口实现分为接收和发送两部分，而接收和发送中，又分为时钟、控制信号与数据两部分。在前面的小节，本文已经介绍了实现发送和接收以及数据与时钟对齐的方法，这里就以图文的形式展示一个具体的实现过程。

RGMII 发送接口实现

对于 RGMII 发送，主要调用两个 ALTDDIO_OUT 的 IP 核就可以实现，其中一个 ALTDDIO_OUT 核的端口位宽为 4 位，用来发送 4 位的 TXD 信号；另一个 ALTDDIO_OUT 核的端口位宽为 1 位，用来发送 TX_EN 和 TX_ER 信号。同时，也有方案使用 ALTDDIO_OUT 核来输出 TX_CLK 信号的，当使用 ALTDDIO_OUT 核输

出 TX_CLK 时，也只需要使用 1 位即可。下图为使用 ALTDDIO_OUT 核实现 RGMII 发送的系统框图。



据图可知，实现 GMII 转 RGMII 接口，甚至不需要我们自己编写任何逻辑代码，只需要分别调用一个 4 位和一个 1 位的 DDIO_OUT 核，然后将信号连接到对应的端口即可。关于 DDIO_OUT 核的调用，已经在介绍“RGMII 发送的 FPGA 实现方案”时讲过，只需要按图分别生成一个 4 位和 1 个 1 位的 DDIO_OUT 核即可。

除了通过 MegaWizard Plug-In Manager（Quartus 13.0 以后版本叫 IP Catalog）调用 DDIO_OUT 核，也可以直接使用原语的方式调用，下述代码为使用原语方式得到两个 DDIO_OUT 核的模板。事实上，这些所谓的模板就是先用 MegaWizard Plug-In Manager 生成该 IP 核，然后从该 IP 核的 V 文件直接提取出来的。我们直接使用该 V 文件来调用，而不使用 MegaWizard Plug-In Manager 生成，是为了方便程序的快速移植。

4 位 DDIO_OUT 核调用模板

```
module ddio_out_x4 (  
    datain_h,
```

```

datain_l,
outclock,
dataout);

input  [3:0] datain_h;
input  [3:0] datain_l;
input   outclock;
output [3:0] dataout;

wire [3:0] sub_wire0;
wire [3:0] dataout = sub_wire0[3:0];

altd dio_out ALTDDIO_OUT_component (
    .datain_h (datain_h),
    .datain_l (datain_l),
    .outclock (outclock),
    .dataout (sub_wire0),
    .aclr (1'b0),
    .aset (1'b0),
    .oe (1'b1),
    .oe_out (),
    .outclocken (1'b1),
    .sclr (1'b0),
    .sset (1'b0));

defparam
    ALTDDIO_OUT_component.extend_oe_disable = "OFF",
    ALTDDIO_OUT_component.invert_output = "OFF",
    ALTDDIO_OUT_component.lpm_hint = "UNUSED",
    ALTDDIO_OUT_component.lpm_type = "altd dio_out",
    ALTDDIO_OUT_component.oe_reg = "UNREGISTERED",
    ALTDDIO_OUT_component.power_up_high = "OFF",
    ALTDDIO_OUT_component.width = 4;

endmodule

```

1 位 DDIO_OUT 核调用模板

```

module ddio_out_x1 (
    datain_h,
    datain_l,
    outclock,
    dataout);

```

```

input  [0:0] datain_h;
input  [0:0] datain_l;
input   outclock;
output [0:0] dataout;

wire [0:0] sub_wire0;
wire [0:0] dataout = sub_wire0[0:0];

altddio_out ALTDDIO_OUT_component (
    .datain_h (datain_h),
    .datain_l (datain_l),
    .outclock (outclock),
    .dataout (sub_wire0),
    .aclr (1'b0),
    .aset (1'b0),
    .oe (1'b1),
    .oe_out (),
    .outclocken (1'b1),
    .sclr (1'b0),
    .sset (1'b0));

defparam
    ALTDDIO_OUT_component.extend_oe_disable = "OFF",
    ALTDDIO_OUT_component.invert_output = "OFF",
    ALTDDIO_OUT_component.lpm_hint = "UNUSED",
    ALTDDIO_OUT_component.lpm_type = "altddio_out",
    ALTDDIO_OUT_component.oe_reg = "UNREGISTERED",
    ALTDDIO_OUT_component.power_up_high = "OFF",
    ALTDDIO_OUT_component.width = 1;

endmodule

```

通过这种方法，当需要将该模块应用到新的工程中时，直接将 V 文件拷贝过去加入工程并例化即可，无需再重新通过 MegaWizard Plug-In Manager 创建，对于 PLL、FIFO、RAM 等 IP 核都可以通过此种思路实现。

rgmii_tx_clk 的输出

在上面的的系统框图中，相信已经有读者发现了一个差异，即对于 rgmii_tx_clk 的输出，图中也是使用的一个 ddio_out 核输出的，而非使用前面介

绍“RGMII 发送的 FPGA 实现方案”时说的，使用 PLL 产生并输出。

使用 DDIO_OUT 输出 rgmii_tx_clk 信号的最大优势在于，可以最大程度上保证输出的 rgmii_tx_clk 与 rgmii_tx_data 保持边沿对齐关系。因为从实现原理上来讲，rgmii_tx_clk 和 rgmii_tx_data 都是 DDIO_OUT 核在 gmii_gtx_clk 的驱动下输出其高位和低位的数据，这样能够保证两者的传输路径的时序模型完全一致，从而使得 rgmii_tx_clk 和 rgmii_tx_data 都能在同一时刻发生变化，实现更加可靠的时钟和数据边沿对齐特性。只是这种情况下，在 PHY 侧，要么通过设置，在 PHY 内部对 rgmii_tx_clk 加入时间延迟，要么通过控制 PCB 走线来为 rgmii_tx_clk 增加一定时间的延迟。如果对于某些设计时时序没有严格控制好的平台板卡，在 FPGA 侧使用边沿对齐和中心都无法保证数据和时钟到达 PHY 时恰好呈中心对齐或边沿对齐模式，则需要使用 PLL 来直接输出 rgmii_tx_clk，并通过实验调整其与 FPGA 内部使用的 gmii_gtx_clk 的相位来最终达到稳定输出数据的目的。在我们设计的开发板上，一般都是通过设置 PHY 内部对 rgmii_tx_clk 加入时间延迟来实现的。无需使用 PLL 来调整发送时钟 rgmii_tx_clk 的相位。

GMII 转 RGMII 顶层代码

在上面，我们已经完整的介绍了实现 GMII 转 RGMII 的方案和各个模块的细节，这里，只需要通过一个顶层模块将上述内容例化到一起，就能最终实现前面框图所介绍的系统了。完整的 GMII 转 RGMII 代码如下所示。

```
module gmii_to_rgmii(  
    gmii_gtx_clk,  
    gmii_tx_data,  
    gmii_tx_en,  
    rgmii_gtx_clk,  
    rgmii_tx_data,  
    rgmii_tx_en  
);  
    input gmii_gtx_clk;  
    input [7:0] gmii_tx_data;  
    input gmii_tx_en;  
    output rgmii_gtx_clk;  
    output [3:0] rgmii_tx_data;  
    output rgmii_tx_en;
```

```

ddio_out_x4 ddio_out_x4(
    .datain_h(gmii_tx_data[3:0]),
    .datain_l(gmii_tx_data[7:4]),
    .outclock(gmii_gtx_clk),
    .dataout(rgmii_tx_data)
);

wire gmii_tx_err;
assign gmii_tx_err = 1'b0; //不产生发送错误信号
ddio_out_x1 ddio_out_ctl(
    .datain_h(gmii_tx_en),
    .datain_l(gmii_tx_en ^ gmii_tx_err),
    .outclock(gmii_gtx_clk),
    .dataout(rgmii_tx_en)
);

ddio_out_x1 ddio_out_clk(
    .datain_h(1'b1),
    .datain_l(1'b0),
    .outclock(gmii_gtx_clk),
    .dataout(rgmii_gtx_clk)
);

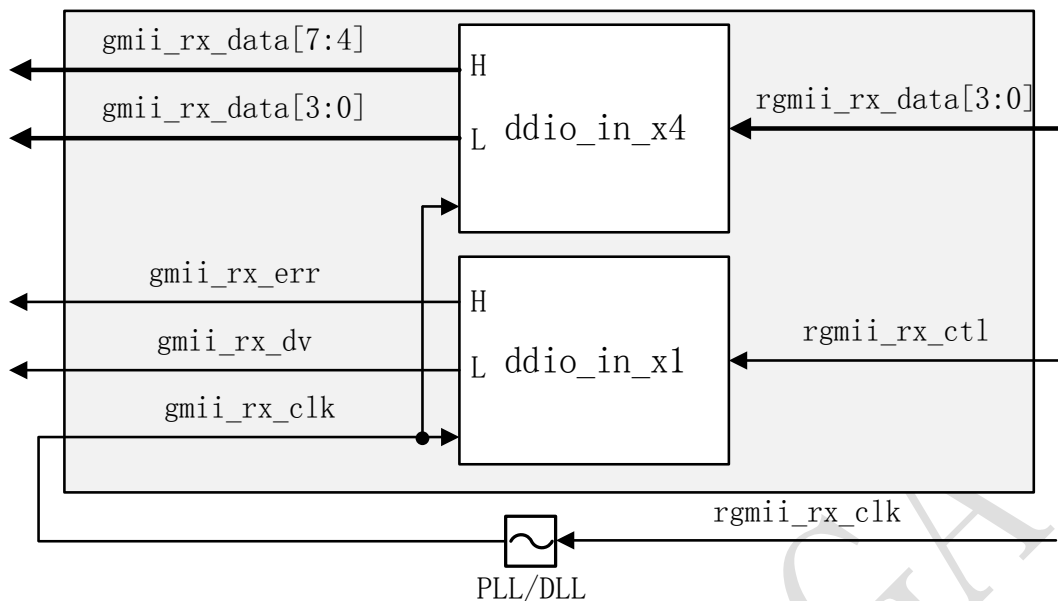
endmodule

```

至此，GMII 转 RGMII 的逻辑功能就设计完成了，在后续设计千兆发送逻辑时，只需要先完全按照 GMII 接口设计逻辑代码，然后在输出到 FPGA 管脚时，加入该转换逻辑，即可快速实现 RGMII 接口的千兆以太网发送逻辑。

RGMII 接收接口实现

对于 RGMII 接收，同样是调用两个 ALTDDIO_IN 的 IP 核就可以实现，其中一个 ALTDDIO_IN 核的端口位宽为 4 位，用来接收 4 位的 RXD 信号；另一个 ALTDDIO_IN 核的端口位宽为 1 位，用来接收 RX_EN 和 RX_ER 信号。下图为使用 ALTDDIO_IN 核实现 RGMII 接收的系统框图。



据图可知，实现 RGMII 转 GMII 接口，也不需要我们自己编写任何逻辑代码，只需要分别调用一个 4 位和一个 1 位的 DDIO_IN 核，然后将信号连接到对应的端口即可。关于 DDIO_IN 核的调用，已经在介绍“RGMII 接收的 FPGA 实现方案”时讲过，只需要按图分别生成一个 4 位和 1 个 1 位的 DDIO_IN 核即可。

和 DDIO_OUT 核一样，DDIO_IN 核也可以使用 MegaWizard Plug-In Manager（Quartus 13.0 以后版本叫 IP Catlog）或原语的方式调用。下述代码为使用原语方式得到两个 DDIO_IN 核的模板。

4 位 DDIO_IN 核调用模板

```
module ddio_in_x4 (
    datain,
    inclock,
    dataout_h,
    dataout_l);

    input  [3:0] datain;
    input    inclock;
    output [3:0] dataout_h;
    output [3:0] dataout_l;

    wire [3:0] sub_wire0;
    wire [3:0] sub_wire1;
    wire [3:0] dataout_h = sub_wire0[3:0];
```

```

wire [3:0] dataout_l = sub_wire1[3:0];

altddio_in ALTDDIO_IN_component (
    .datain (datain),
    .inclock (inclock),
    .dataout_h (sub_wire0),
    .dataout_l (sub_wire1),
    .aclr (1'b0),
    .aset (1'b0),
    .inclocken (1'b1),
    .sclr (1'b0),
    .sset (1'b0));

defparam
    ALTDDIO_IN_component.invert_input_clocks = "OFF",
    ALTDDIO_IN_component.lpm_hint = "UNUSED",
    ALTDDIO_IN_component.lpm_type = "altddio_in",
    ALTDDIO_IN_component.power_up_high = "OFF",
    ALTDDIO_IN_component.width = 4;

endmodule

```

1 位 DDIO_IN 核调用模板

```

module ddio_in_x1 (
    datain,
    inclock,
    dataout_h,
    dataout_l);

input  [0:0] datain;
input    inclock;
output [0:0] dataout_h;
output [0:0] dataout_l;

wire [0:0] sub_wire0;
wire [0:0] sub_wire1;
wire [0:0] dataout_h = sub_wire0[0:0];
wire [0:0] dataout_l = sub_wire1[0:0];

altddio_in ALTDDIO_IN_component (
    .datain (datain),
    .inclock (inclock),

```

```

        .dataout_h (sub_wire0),
        .dataout_l (sub_wire1),
        .aclr (1'b0),
        .aset (1'b0),
        .inclocken (1'b1),
        .sclr (1'b0),
        .sset (1'b0));

defparam
    ALTDDIO_IN_component.invert_input_clocks = "OFF",
    ALTDDIO_IN_component.lpm_hint = "UNUSED",
    ALTDDIO_IN_component.lpm_type = "altddio_in",
    ALTDDIO_IN_component.power_up_high = "OFF",
    ALTDDIO_IN_component.width = 1;

endmodule

```

通过这种方法，当需要将该模块应用到新的工程中时，直接将 V 文件拷贝过去加入工程并例化即可，无需再重新通过 MegaWizard Plug-In Manager 创建。

rgmii_rx_clk 的接收

对于 RGMII 接口来说，rgmii_rx_clk 时钟是由 PHY 产生，提供给 FPGA 的。而该时钟在到达 FPGA 内部接收 rgmii_rx_data 和 rgmii_rx_ctl 的 ddio_in 寄存器时需要与 rgmii_rx_data 和 rgmii_rx_ctl 信号呈中心对齐的关系。而 rgmii_rx_ctl 时钟不仅需要作为 ddio_in 的接口时钟，在 FPGA 内部还会作为以太网数据的解包逻辑时钟，是一个典型的时钟信号。所以如果直接使用普通的 FPGA IO 接入该时钟，势必会导致时钟质量下降，从而引起逻辑运行的不稳定。而且，一旦该时钟信号和 rgmii_rx_data 不满足中心对齐的关系，要调整会非常的麻烦。所以综合考虑，将该时钟信号接入了 FPGA 的专用时钟管脚，使其能够直接进入 FPGA 内部的 PLL，也能够直接上 FPGA 的全局时钟资源。在保证时钟质量的同时，还能确保万一该时钟相位无法满足 rgmii_rx_data 的捕获要求，还能通过 FPGA 内部的 PLL 进行相位调制，以确保最终 rgmii_rx_data 能够被正确接收。所以在上述框图中，rgmii_rx_clk 接入 FPGA 后并没有直接接到 ddio_in 的时钟脚，而是先进入了 PLL，由 PLL 调整到合适的相位之后，再用来捕获 rgmii_rx_data。

RGMII 转 GMII 顶层代码

在上面，我们已经完整的介绍了实现 RGMII 转 GMII 的方案和各个模块的细

节，这里，只需要通过一个顶层模块将上述内容例化到一起，就能最终实现前面框图所介绍的系统了。完整的 RGMII 转 GMII 代码如下所示。

```
module rgmii_to_gmii(  
    input rgmii_rx_clk,  
    input  [3:0] rgmii_rx_data,  
    input rgmii_rx_ctl,  
    output gmii_rx_clk,  
    output [7:0] gmii_rx_data,  
    output gmii_rx_ctl  
);  
  
    ddio_in_x4 ddio_in_x4(  
        .datain(rgmii_rx_data),  
        .inclock(rgmii_rx_clk),  
        .dataout_h(gmii_rx_data[7:4]),  
        .dataout_l(gmii_rx_data[3:0])  
    );  
  
    ddio_in_x1 ddio_in_x1(  
        .datain(rgmii_rx_ctl),  
        .inclock(rgmii_rx_clk),  
        .dataout_h(),  
        .dataout_l(gmii_rx_ctl)  
    );  
  
    assign gmii_rx_clk = rgmii_rx_clk;  
  
endmodule
```

至此，RGMII 转 GMII 的逻辑功能就设计完成了，在后续设计千兆接收逻辑时，只需要先完全按照 GMII 接口设计逻辑代码，然后在连接到 FPGA 管脚时，加入该转换逻辑，即可快速实现 RGMII 接口的千兆以太网接收逻辑。

总结

至此，关于 RGMII 接口与 GMII 接口的互转逻辑设计就已经介绍完成了，有了这个互转逻辑，在 FPGA 中设计以太网的接收和发送逻辑时，只需要按照 GMII

接口的形式，先设计出对应的发送和接收逻辑，再将对应的端口连接到 RGMII 与 GMII 接口转换逻辑上，就能够完成基于 RGMII 接口的以太网接收和发送了。

ARP 协议的原理与 FPGA 实现

上一节，我们介绍了整个以太网传输中最底层的 MAC 帧格式，也提到，MAC 帧并不是直接对用户的数据帧，一段用户信息要想通过以太网传输还需要经过各种传输层协议的层层打包才能最终送入 MAC 帧的数据字段进行传输。无论是传输用户数据还是一些网络通信辅助相关的信息，都需要将数据编码为指定协议后再打包进 MAC 层传输。所以本节将选择一个最简单的协议 ARP 协议，来介绍以太网 MAC 帧的组包方式，并使用 FPGA 编写 MAC 帧发送器，来发送预先设定好的一段 ARP 协议数据内容，并通过以太网抓包查看数据内容的方式来直观验证 MAC 帧发送器的设计成果。

之所以选择 ARP 协议来辅助进行以太网 MAC 帧发送器的设计，是因为 ARP 协议是所有 MAC 帧上层协议中最简单的协议，使用该协议，我们在生成以太网帧的数据和填充字段的内容时会非常的简单方便，而 ARP 协议又能够在 PC 的以太网抓包工具中正确抓取并显示，非常适合我们用来进行最直观的连通性测试。

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(MAC)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。然后源主机收到该应答报文，就能从报文中解析得知目的主机的硬件地址是多少，然

后就能正确的发送数据了。

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。由于网络上的 IP 地址本身是动态分配的（Ipv4 资源严重不足，所以采用动态分配机制），上网时每个设备的 IP 地址在不同的时间点其地址都有可能不同，所以某个网络设备在不同的时间，其 IP 地址可能不一样，所以需要用这种缓存表项定期失效的机制来避免 MAC 地址和不同时刻 IP 地址绑定的冲突。下表为完整的 ARP 帧每个字段的数据意义（包含了以太网帧头部字段）。

14 字节以太网帧首部			28 字节 ARP 请求/应答帧内容								
以太网 目的地址	以太网 源地址	帧 类 型	硬件 类型	协议 类型	硬件 地址 长度	协议 地址 长度	操 作 码	发送端 以太网 地址	发送端 IP 地址	目的 以太网 地址	目的 IP 地址
6	6	2	2	2	1	1	2	6	4	6	4

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，这两个字段的内容值是不一样的。在发送以太网帧时，以太网首部中的目的 MAC 地址为广播地址，也就是 6 字节的 0xFF。而 ARP 数据报中当发送请求报文时由于目的 MAC 地址由于暂时无法知晓，所以将其值填充为 0。ARP 返回报文中的值由于已经知晓，所以以太网首部和 ARP 数据报中的值用实际值填充。

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。每个字段具体的功能描述如下表所示。

	字段	度	默认值	备注
以太网首部	接收方 MAC	6		- 发送 ARP 请求报文时，该值为广播地址（FF-FF-FF-FF-FF-FF） - 发送 ARP 应答报文时，该值为对应的发送请求报文设备的实际 MAC 地址。
	发送方 MAC	6		数据帧发送方 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 请求报文 ● 0x2 表示 ARP 应答报文
	发送方 MAC	6		源 MAC 地址
	发送方 IP	4		源 IP 地址
	接收方 MAC	6		● 发送 ARP 请求报文时,该值由于当前未知,用全 0 填充。 ● 发送 ARP 应答报文时,该值为对应的发送请求报文设备的实际 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

手工组建 ARP 数据帧

在了解了以上以太网帧和 ARP 帧的具体格式之后,只要我们确定一个发送方和一个接收方,就能手工组装一个数据帧了。以下以一个具体的场景为例来进行介绍。

假设现在有两台设备,分别命名为 A 和 B。

A 设备 MAC 地址为 00-07-ed-ac-62-00, IP 地址为 192.168.0.2 (C0-A8-00-02)

B 设备 MAC 地址为 84-7b-eb-48-94-13, IP 地址为 192.168.0.3 (C0-A8-00-03)

注意, B 设备实际就是笔者实验用的电脑,其 MAC 地址就是笔者实验电脑的实际 MAC 地址, IP 地址则是我们以太网实验中统一使用的 PC 的 IP 地址,需要提前设置好,方能在后续进行实验时不遇到问题。

A 设备发起 ARP 请求包,请求 IP 地址为 192.168.0.3 的设备的 MAC 地址, B 设备收到该 ARP 请求包,且经过匹配,发现请求的 IP 地址与自己一致,然后发出 ARP 应答包。



MAC:00-07-ed-ac-62-00

IP: 192.168.0.2

设备A

MAC: 84-7b-eb-48-94-13

IP: 192.168.0.3

设备B

据此，除 CRC32 字段需要根据数据内容计算得出以外，ARP 请求报文和 ARP 接收报文的每个字段的准确值都可以手动组建出。而一旦这些值确定了，对应的 CRC32 校验值也就可以通过计算得到唯一值。下表为根据上述参数手动组建出的 ARP 请求包和 ARP 应答包的所有字段的值。

数据帧字段	ARP 请求帧	ARP 应答帧
接收方 MAC	FF-FF-FF-FF-FF-FF	00-07-ed-ac-62-00
发送方 MAC	00-07-ed-ac-62-00	84-7b-eb-48-94-13
类型	08-06	08-06
硬件类型	00-01	00-01
协议类型	08-00	08-00
MAC 地址长度	06	06
IP 地址长度	04	04
操作代码	01	02
发送方 MAC	00-07-ed-ac-62-00	84-7b-eb-48-94-13
发送方 IP	C0-A8-00-02	C0-A8-00-03
接收方 MAC	00-00-00-00-00-00	00-07-ed-ac-62-00
接收方 IP	C0-A8-00-03	C0-A8-00-02
填充字段	18 个 00	18 个 00
校验字段	由实际帧内容计算	由实际帧内容计算

使用以太网测试工具生成数据帧

上述组包方式是在我们充分掌握了以太网 MAC 帧和 ARP 帧的每一个字段的物理意义后手动组建出来的，若对于这两个帧结构掌握不是十分牢固，则有可能在组建过程中出错，所以这里再给大家介绍一种使用工具组包的方法。该工具名为“小兵以太网测试仪”，软件安装包名为“小兵以太网测试仪.rar”或者“以太网帧生成器.rar”或者“xb_ether_tester_v3.2.8.rar”。

该软件是基于 wincap 包开发的以太网数据帧发送和接收工具。安装并打开该软件，在界面中点击新建流按钮以打开新报文编辑界面，如下图所示。

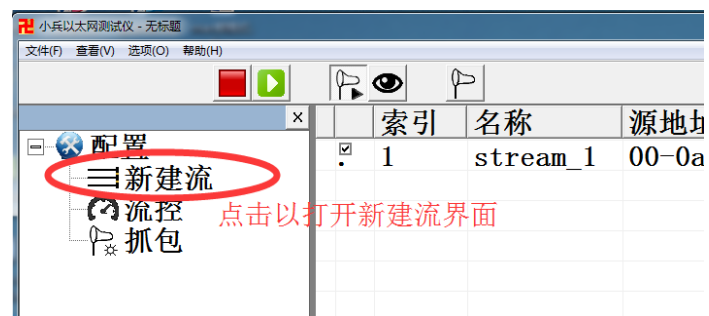


图 0-3

在常用报文中选择 arp 请求格式报文，自动更新栏中的所有选项在本报文生成中开启和关闭都可以，因为不涉及到一些动态数据的生成。

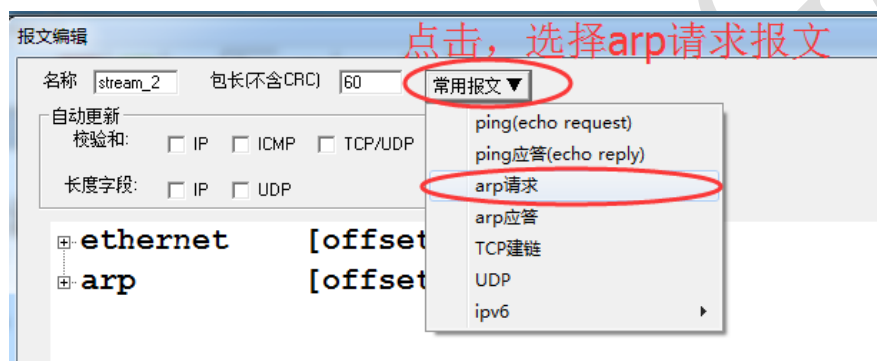


图 0-4

修改以太网包和 arp 包中各字段内容如下所示：

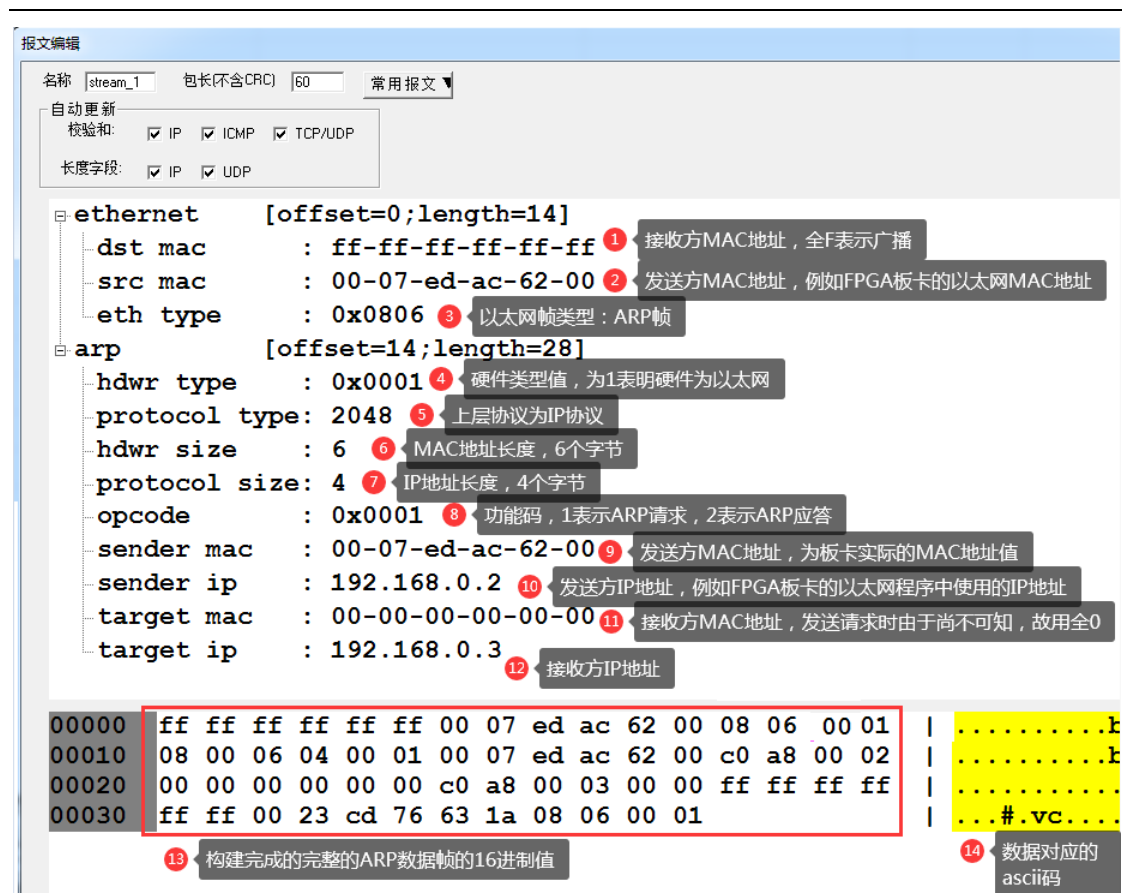


图 0-5

配置好之后点击确定，然后将构建好的数据包导出为 16 进制格式文本文件，如下图所示。

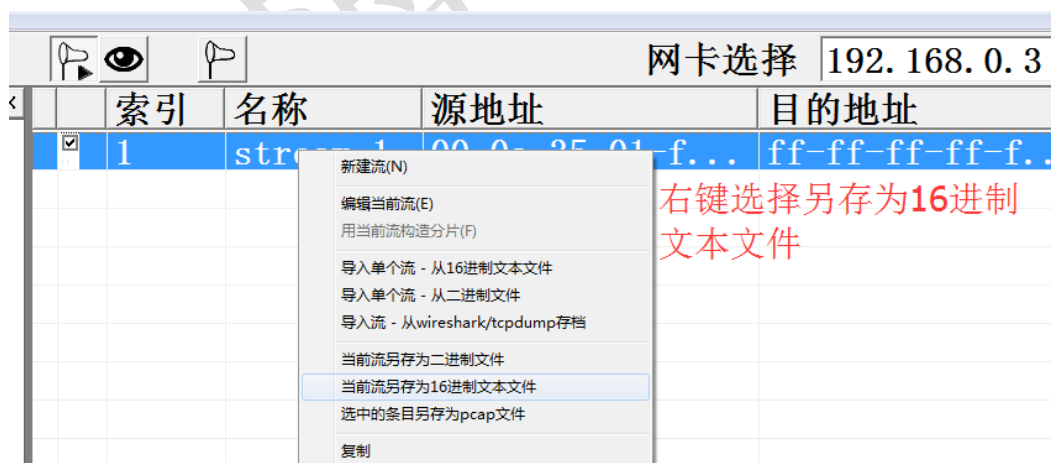


图 0-6

得到的文本中数据内容如下所示：

0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x07, 0xed, 0xac, 0x62, 0x00, 0x08, 0x06, 0x00,

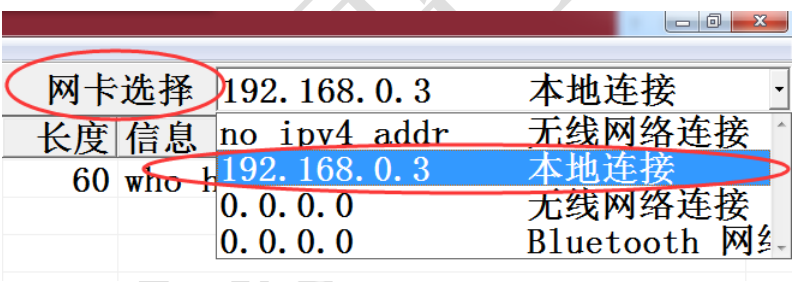
0x01, 0x08, 0x00, 0x06, 0x04, 0x00, 0x01, 0x00, 0x07, 0xed, 0xac, 0x62, 0x00, 0xc0, 0xa8, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0xa8, 0x00, 0x03, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00, 0x23, 0xcd, 0x76, 0x63, 0x1a, 0x08, 0x06, 0x00, 0x01,

使用 Wireshark 抓包验证

上述组建好的报文，可以直接使用小兵以太网测试仪通过网卡发包，然后我们可以使用 wireshark 抓取网卡上的该报文。

要想 wireshark 能够正常抓有限网卡上的数据包，首先需要让你的电脑的有线网卡处于连接的活动状态，在做 FPGA 实验时，例如下载我们提供的以太网回环测试例程，就能让电脑的网卡连接上，处于活动状态。也可以将电脑网卡连接到交换机，路由器等能够让网卡检测到连接的设备上。

其次，在小兵以太网测试仪的右上角切换网卡为你的有线网卡，如下图所示：



选中刚刚组建好的 ARP 数据包，然后点击开始发包按钮执行发包。

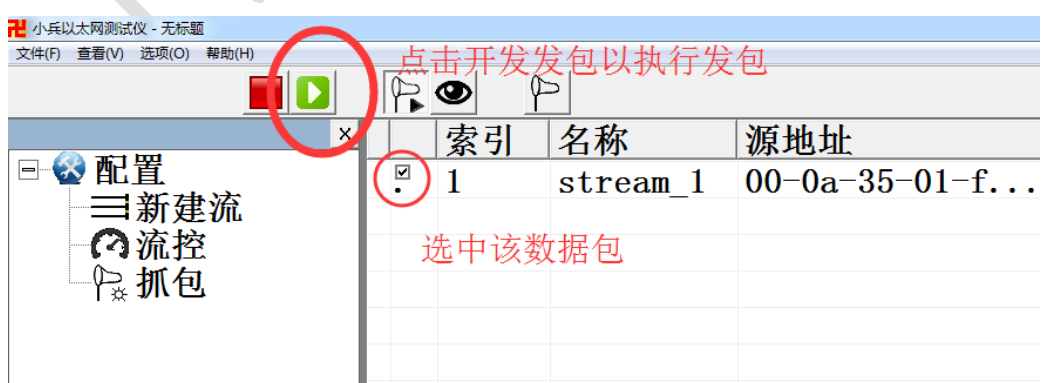
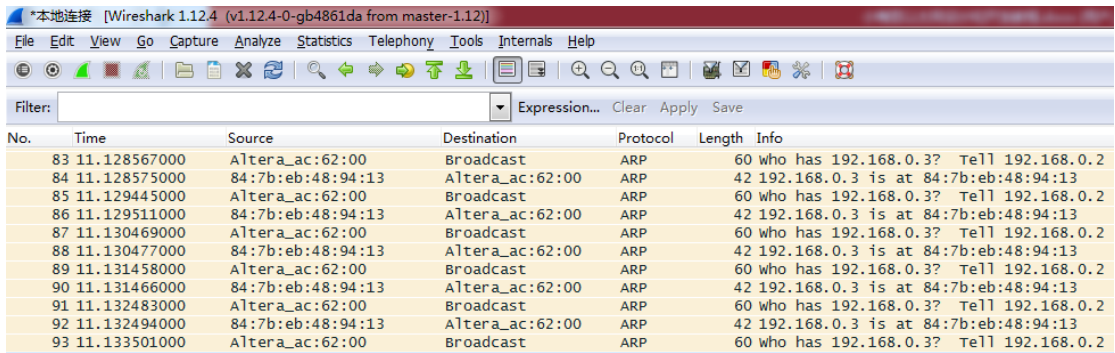


图 0-7

打开 Wireshark 软件并抓取有线网卡的连接，则可以抓取到大量的 ARP 请求

报文和应答报文，如下图所示。



The screenshot shows the Wireshark interface with a packet list table. The table has columns: No., Time, Source, Destination, Protocol, Length, and Info. The packets are ARP requests and replies between 'Altera_ac:62:00' and '84:7b:eb:48:94:13'.

No.	Time	Source	Destination	Protocol	Length	Info
83	11.128567000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
84	11.128575000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
85	11.129445000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
86	11.129511000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
87	11.130469000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
88	11.130477000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
89	11.131458000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
90	11.131466000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
91	11.132483000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
92	11.132494000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
93	11.133501000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2

其中，Destination 项为 Broadcast 的包就是 ARP 请求包，紧随其后的一个包就是该 ARP 请求包的应答包。

注意，在整个实验中，B 设备实际就是笔者实验用的电脑，其 MAC 地址就是笔者实验电脑的实际 MAC 地址，IP 地址则是我们以太网实验中统一使用的 PC 的 IP 地址，所以在上述抓包过程中，会有 ARP 应答包存在。读者自己在实验时，也需要先将自己的实验电脑有线网卡 IP 地址设置为 192.168.0.3。

选择一个 ARP 请求包，查看其详细的帧内容，如下图所示：

107 11.140460000 Altera_ac:62:00 Broadcast ARP 60 who has 19

108 11 140466000 84:7h:eh:48:04:13 Altera_ac:62:00 ARP 42 192 168 0

Frame 107: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
 Interface id: 0 (\Device\NPF_{6A7D2546-2437-47DF-AA53-F4C8CB15DB85})
 Encapsulation type: Ethernet (1)
 Arrival Time: Mar 7, 2020 18:31:07.335889000 [Time shift for this packet: 0.000000000 seconds]
 Epoch Time: 1583577067.335889000 seconds
 [Time delta from previous captured frame: 0.000993000 seconds]
 [Time delta from previous displayed frame: 0.000993000 seconds]
 [Time since reference or first frame: 11.140460000 seconds]
 Frame Number: 107
 Frame Length: 60 bytes (480 bits)
 Capture Length: 60 bytes (480 bits)
 [Frame is marked: False]
 [Frame is ignored: False]
 [Protocols in frame: eth:ethertype:arp]
 [Coloring Rule Name: ARP]
 [Coloring Rule String: arp]

Ethernet II, Src: Altera_ac:62:00 (00:07:ed:ac:62:00), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
 Destination: Broadcast (ff:ff:ff:ff:ff:ff)
 Address: Broadcast (ff:ff:ff:ff:ff:ff)
 1. = LG bit: Locally administered address (this is NOT the
 1. = IG bit: Group address (multicast/broadcast)
 Source: Altera_ac:62:00 (00:07:ed:ac:62:00)
 Address: Altera_ac:62:00 (00:07:ed:ac:62:00)
 0. = LG bit: Globally unique address (factory default)
 0. = IG bit: Individual address (unicast)
 Type: ARP (0x0806)
 Padding: 0000ffffffffffff0023cd76631a08060001 1 无意义填充内容

Address Resolution Protocol (request)
 Hardware type: Ethernet (1)
 Protocol type: IP (0x0800)
 Hardware size: 6
 Protocol size: 4
 Opcode: request (1)
 Sender MAC address: Altera_ac:62:00 (00:07:ed:ac:62:00)
 Sender IP address: 192.168.0.2 (192.168.0.2)
 Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
 Target IP address: 192.168.0.3 (192.168.0.3)

0000 ff ff ff ff ff ff 00 07 ed ac 62 00 08 06 00 01b.....
 0010 08 00 06 04 00 01 00 07 ed ac 62 00 c0 a8 00 02b.....
 0020 00 00 00 00 00 00 c0 a8 00 03 00 00 ff ff ff ff
 0030 ff ff 00 23 cd 76 63 1a 08 06 00 01 2 蓝色为无意义的填充内容，确保帧长的

图 0-8

图中最后的蓝色字样部分为软件为保证整个数据帧长度（不含 FCS 字段）不小于 64 个字节而填充的无意义内容。

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 4 个字节的帧校验（FCS）字段，即 CRC32 值。而在上述的整个组包和抓包过程中，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，由于 CRC32 的计算本身又涉及到了较多的理论知识，实现也有一定的难度。本节实验为了降低难度，先暂时回避该算法，使用 CRC 计算软件手动计算出数据报的 CRC32 值，然后使用该计算得到的 CRC32 值来让 FPGA 的网卡先体

验一波发包，先看到直观的现象。相信这也是大家学习的最爱。

计算以太网数据的 CRC 字段，可以使用专用 CRC 校验工具实现，该工具名为“CRC_Calc+v0.1.exe”。

打开 CRC_Calc+v0.1.exe”校验软件，在计算多项式中选择倒数第二个多项式 CRC-32 作为进行 CRC32 计算的多项式，该多项式一般用于 WinRAR 压缩软件和以太网传输中。

当选定一个计算多项式后，右侧会展示出该计算方法对应的一些参数的具体值，对于这些参数的意义介绍如下：

- Name: 参数模型名称。
- Width: 宽度，即 CRC 比特数。
- Poly: 生成项的简写，以 16 进制表示。例如：CRC-32 即是 0x04C11DB7，忽略了最高位的“1”，即完整的生成项是 0x104C11DB7。
- Init: 这是算法开始时 CRC 的初始化预置值，十六进制表示。
- Refin: 待测数据的每个字节是否按位反转，True 或 False。
- Refout: 在计算后之后，异或输出之前，整个数据是否按位反转，True 或 False。
- Xorout: 计算结果与此参数异或后得到最终的 CRC 值。

将之前保存的记事本中的数据内容复制粘贴到计算软件的文本框中，然后点击 Calculate 即可计算出这串数据的 CRC32 值，如下图所示。

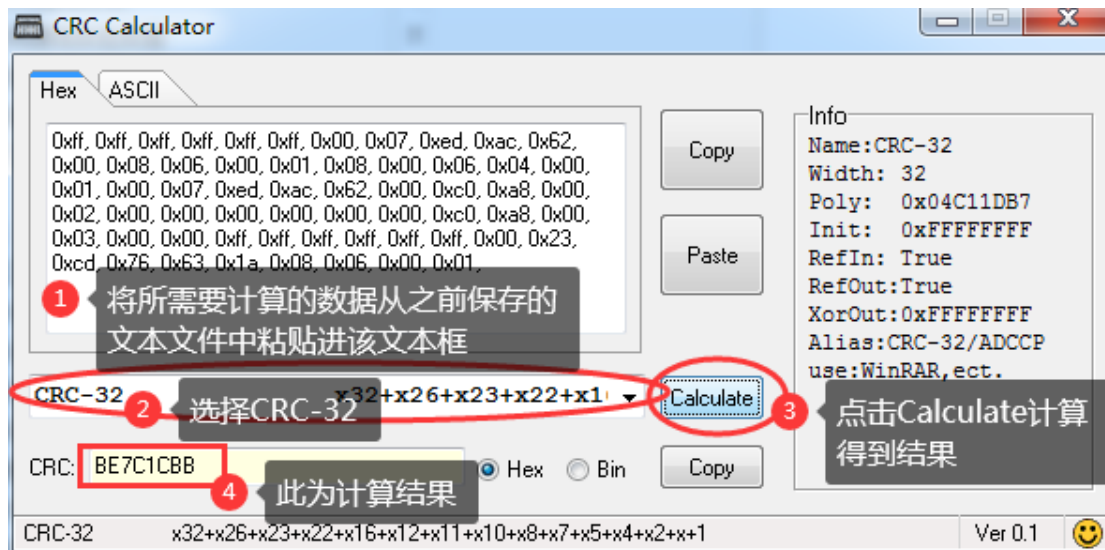


图 0-9

注意，小兵以太网测试仪在保存得到文本文件时，是以 16 个数据为一行，然后换行继续输出的，所以直接从该文本中赋值内容会因为换行符号的存在而导致本计算软件计算结果异常，结果全为 0。所以如果你计算出来的结果为 0，可以采取下列方式解决：

- 安装有 notepad++ 软件的，使用 notepad++ 软件打开该文件，能自动显示换行的位置，将文件中所有换行符删除后再复制到 CRC 计算软件中。
- 如果没有安装 notepad++ 软件，可以在 quartus 中新建一个 verilog 文件，然后将该文件中的内容复制到该 verilog 文件中，也会自动显示换行，将文件中所有换行符删除后再复制到 CRC 计算软件中。

需要注意的是，CRC 软件计算结果与以太网使用时的关系为

CRC 计算软件结果：低字节在前，高字节在后。

以太网使用时：高字节在前，低字节在后，即应以字节位为单位调换 4 个字节的高低顺序，如 BE7C1CBB 应该调换为 BB1C7CBE。

```
//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'hBE7C1CBB;

wire [31:0]CRC_result;
```

```
//调整 CRC 校验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]};
```

至此，ARP 以太网数据报的所有字段的内容就都得到了。接下来，就可以使用该数据报内容来进行 FPGA 设计以太网的初探了。

以太网 ARP 帧发包实例（手动 CRC）

工程源码	--盘 A_AC620 开发板标准配套资料 ----02_设计实例 -----AC620_Verilog_Example -----以太网 -----arp_send_rgmii.rar（千兆 RGMII 接口） -----arp_send_mii.rar（百兆 MII 接口）
相关视频课程	--小梅哥 FPGA 入门到实战特训班课程 ----0808_02 以太网基本概念与 MAC 层协议介绍.mp4 ----0808_03 MAC 层数据传输接口 MII_GMII_RGMII.mp4 ----0809_01 以太网 MAC 层板级调试方法.mp4 ----0809_02 以太网 IP 层协议详解.mp4 ----0810_02 UDP 协议详解.mp4 ----0812_01 以太网传输项目总结.mp4

本节实验将基于 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。本设计实例针对 AC620 开发板上的网口工作模式，提供两个工程。

- arp_send_rgmii.rar 对应开发板上网口工作在千兆 RGMII 接口模式；
- arp_send_mii.rar 对应开发板上网口工作在百兆 MII 接口模式；

工程中包含了两个逻辑设计文件，其中：

- 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。

MAC 帧生成和发送

`eth_send` 模块实现的是 MAC 帧发送的核心内容。其实该内容也非常简单，就是一个序列机，按照顺序将 MAC 帧中的前导码、分隔符、目的 MAC 地址、源 MAC 地址、长度/类型发出，然后开始根据上层传入的数据长度（`data_length`）值，从上层 `fifo` 中读取对应个字节的数据发出，最后再发出 4 字节的 CRC 字段即可。

对于千兆 GMII/RGMII 接口和百兆 MII 接口，两者实现数据发送时有较大的差别，千兆 GMII/RGMII 接口每个时钟周期发送 8 位数据，而 MII 接口每次发送 4 位数据，所以序列机的实现上来说，MII 的发送需要比 GMII/RGMII 序列长 1 倍。下面为采用序列机方式实现 MII 接口以太网数据发送的代码内容。可以看到，每个计数值输出当前阶段需要输出数据的其中 4 位数据，前导码占用 15 个计数值，分隔符占用两个计数值。对于一个字节的数据，MIII 在发送时，先发送低 4 位，再发送高四位。

```
/*
    序列机部分，根据不同的时刻，切换 MII 接口数据线上的内容，
    包含前导码、分隔符、目的地址、源地址、以太网帧类型/长度、
    数据段数据、结尾 CRC 校验值
*/

always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    mii_tx_data <= #1 4'd0;
else begin
    case(cnt)
        1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15:
            mii_tx_data <= #1 4'h5;    //前导码

        16: mii_tx_data <= #1 4'hd;    //分隔符

        //目的 MAC 地址
        17: mii_tx_data <= #1 r_des_mac[43:40];
        18: mii_tx_data <= #1 r_des_mac[47:44];
        19: mii_tx_data <= #1 r_des_mac[35:32];
        20: mii_tx_data <= #1 r_des_mac[39:36];
        21: mii_tx_data <= #1 r_des_mac[27:24];
```

```

22: mii_tx_data <= #1 r_des_mac[31:28];
23: mii_tx_data <= #1 r_des_mac[19:16];
24: mii_tx_data <= #1 r_des_mac[23:20];
25: mii_tx_data <= #1 r_des_mac[11:8];
26: mii_tx_data <= #1 r_des_mac[15:12];
27: mii_tx_data <= #1 r_des_mac[3:0];
28: mii_tx_data <= #1 r_des_mac[7:4];

//源 MAC 地址
29: mii_tx_data <= #1 src_mac[43:40];
30: mii_tx_data <= #1 src_mac[47:44];
31: mii_tx_data <= #1 src_mac[35:32];
32: mii_tx_data <= #1 src_mac[39:36];
33: mii_tx_data <= #1 src_mac[27:24];
34: mii_tx_data <= #1 src_mac[31:28];
35: mii_tx_data <= #1 src_mac[19:16];
36: mii_tx_data <= #1 src_mac[23:20];
37: mii_tx_data <= #1 src_mac[11:8];
38: mii_tx_data <= #1 src_mac[15:12];
39: mii_tx_data <= #1 src_mac[3:0];
40: mii_tx_data <= #1 src_mac[7:4];

//以太网帧类型/长度
41: mii_tx_data <= #1 r_type_length[11:8];
42: mii_tx_data <= #1 r_type_length[15:12];
43: mii_tx_data <= #1 r_type_length[3:0];
44: mii_tx_data <= #1 r_type_length[7:4];

//发送数据
45: mii_tx_data <= #1 fifo_rddata;

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

54: mii_tx_data <= #1 4'd0;
default: mii_tx_data <= #1 4'd0;
endcase

```

```
end
```

下述代码则为采用序列机方式实现 GMII/RGMII 接口以太网数据发送的代码内容。可以看到，每个计数值输出当前阶段需要输出数据的其中 8 位数据，前导码占用 7 个计数值，分隔符占用 1 个计数值。

```
/*
    序列机部分，根据不同的时刻，切换 MII 接口数据线上的内容，
    包含前导码、分隔符、目的地址、源地址、以太网帧类型/长度、
    数据段数据、结尾 CRC 校验值
*/

always@(posedge gmii_tx_clk or negedge rst_n)
if(!rst_n)
    gmii_tx_data <= #1 8'd0;
else begin
    case(cnt)
        1, 2, 3, 4, 5, 6, 7:
            gmii_tx_data <= #1 8'h55; //前导码

        8: gmii_tx_data <= #1 8'hd5; //分隔符

        //目的 MAC 地址
        9: gmii_tx_data <= #1 r_des_mac[47:40];
        10: gmii_tx_data <= #1 r_des_mac[39:32];
        11: gmii_tx_data <= #1 r_des_mac[31:24];
        12: gmii_tx_data <= #1 r_des_mac[23:16];
        13: gmii_tx_data <= #1 r_des_mac[15:8];
        14: gmii_tx_data <= #1 r_des_mac[7:0];

        //源 MAC 地址
        15: gmii_tx_data <= #1 src_mac[47:40];
        16: gmii_tx_data <= #1 src_mac[39:32];
        17: gmii_tx_data <= #1 src_mac[31:24];
        18: gmii_tx_data <= #1 src_mac[23:16];
        19: gmii_tx_data <= #1 src_mac[15:8];
        20: gmii_tx_data <= #1 src_mac[7:0];

        //以太网帧类型/长度
        21: gmii_tx_data <= #1 r_type_length[15:8];
        22: gmii_tx_data <= #1 r_type_length[7:0];
```

```

//发送数据
23: gmii_tx_data <= #1 fifo_rddata;

//发送 CRC 校验结果
24: gmii_tx_data <= #1 CRC_Result[31:24];
25: gmii_tx_data <= #1 CRC_Result[23:16];
26: gmii_tx_data <= #1 CRC_Result[15:8];
27: gmii_tx_data <= #1 CRC_Result[7:0];

28: gmii_tx_data <= #1 8'd0;
default: gmii_tx_data <= #1 8'd0;
endcase
end

```

通过对比可以发现，GMII/RGMII 接口实现更加简单。这也是我们在已经有更加高速高效的千兆以太网传输能力的情况下，依旧要费力讲解 MII 接口的原因，因为 MII 接口实现比 GMII/RGMII 更加的复杂，两者对比实现，能够让大家对这两种接口都更加扎实的掌握。

在代码中可以看到，对于 MII 接口，在序列计数器计数值为 45，对于 GMII 接口，在序列计数器计数值为 23 的情况下，既开始发送 MAC 层的数据和填充字段了，所以根据此计数值来产生对上层 FIFO 的读请求信号。如下所示（以 MII 接口为例）：

```

//帧中数据发送使能信号
assign en_tx_data = (cnt == 45) && (data_num > 1);

//待发送数据计数器，每发送一个数据段中的数据，本计数器减 1.
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    data_num <= #1 0;
else if(tx_go)
//将长度数据乘以 2，就是需要发送的 4 位数的个数
    data_num <= #1 data_length << 1;
else if(en_tx_data)
    data_num <= #1 data_num - 1'b1;
else
    data_num <= #1 data_num;

```

由于在发送整个数据和填充字段的过程中序列计数器需要保持在 45（MII 接口）或 23（GMII 接口）不变，所以使用上述产生的 en_tx_data 信号来控制计数器，让其在发送数据字段的时候停止变化，该部分内容如下所示。

```
//根据发送启动信号产生内部发送使能信号
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    en_tx <= #1 1'd0;
else if(tx_go)
    en_tx <= #1 1'd1;
else if(cnt >= 53) //一帧数据发送完成，清零发送使能信号
    en_tx <= #1 1'd0;

//主序列机计数器
always@(posedge mii_tx_clk or negedge rst_n)
if(!rst_n)
    cnt <= #1 6'd0;
else if(en_tx)begin
    if(!en_tx_data)
        cnt <= #1 cnt + 6'd1;
    else //在发送整个帧中的数据部分时，计数器暂停
        cnt <= #1 cnt;
end
else
    cnt <= #1 6'd0;
```

ARP 数据报生成和发送

在 eth_send_test 模块中例化调用以太网帧发送模块 eth_send，并将目的地址设置为广播地址 ffffffff、源地址为板卡地址。这里选择的源地址 00_07_ed_ac_62_00 实际是由 Altera 官方拥有的地址段和我们的开发板型号组合而成的，该地址如果在 wireshark 中抓包则可以看到 Altera-ac-62-00 的字样，大家可以去前面 wireshark 抓包验证环节查看。

数据长度为 ARP 数据包的长度，这里为 46 字节。

协议类型为 ARP(0806)。。

```

eth_send eth_send(

    .rst_n(rst_n),
    .tx_go(tx_go),
    .data_length(12'd46),
    .des_mac(48'hff_ff_ff_ff_ff_ff),
    .src_mac(48'h00_07_ed_ac_62_00),
    .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。整个计数周期在 MII 模式下，由于时钟频率是 25M，所以发送周期为 671ms 左右 ($40\text{ns} * 2^{24}$)，在 GMII/RGMII 模式下，由于时钟频率为 125M，所以发送周期为 134ms 左右 ($8\text{ns} * 2^{24}$)。以下以千兆速率下 GMII/RGMII 接口模式为例。

```

//发送间隔计数器
reg [23:0]cnt;

always@(posedge gmii_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;

//24 位 cnt 计满一次启动一次发送，该时间大约为 134ms
assign tx_go = (cnt == 24'd1);

```

设计这个发送速率控制逻辑的目的在于降低 FPGA 每秒发包的次数，避免发包太快给 PC 网卡带来负担。发包太多太快没有任何意义。

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。以下以千兆速率下 GMII/RGMII 接口模式为例。

```
//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;
```

以下使用查找表形式依次输出 arp 协议的报文内容。

MII 模式下 ARP 数据包查找表	RGMII/GMII 模式下 ARP 数据包查找表
<pre>//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;</pre>	<pre>//ARP 包数据段 always@(*) begin case(data_cnt) //hdwr type 00: fifo_rddata = 8'h00; 01: fifo_rddata = 8'h01; //protocol type 02: fifo_rddata = 8'h08; 03: fifo_rddata = 8'h00; //hdwr size 04: fifo_rddata = 8'h06; //protocol size 05: fifo_rddata = 8'h04; //opcode 06: fifo_rddata = 8'h00; 07: fifo_rddata = 8'h01; //sender mac: 00-07-ed-ac-62-00 08: fifo_rddata = 8'h00;</pre>

<pre> //opcode 12: fifo_rddata = 4'h0; 13: fifo_rddata = 4'h0; 14: fifo_rddata = 4'h1; 15: fifo_rddata = 4'h0; //sender mac: 00-07-ed-ac-62-00 16: fifo_rddata = 4'h0; 17: fifo_rddata = 4'h0; 18: fifo_rddata = 4'h7; 19: fifo_rddata = 4'h0; 20: fifo_rddata = 4'hd; 21: fifo_rddata = 4'he; 22: fifo_rddata = 4'hc; 23: fifo_rddata = 4'ha; 24: fifo_rddata = 4'h2; 25: fifo_rddata = 4'h6; 26: fifo_rddata = 4'h0; 27: fifo_rddata = 4'h0; //sender ip : 192.168.0.2 28: fifo_rddata = 4'h0; 29: fifo_rddata = 4'hc; 30: fifo_rddata = 4'h8; 31: fifo_rddata = 4'ha; 32: fifo_rddata = 4'h0; 33: fifo_rddata = 4'h0; 34: fifo_rddata = 4'h2; 35: fifo_rddata = 4'h0; //target mac 36: fifo_rddata = 4'h0; 37: fifo_rddata = 4'h0; 38: fifo_rddata = 4'h0; 39: fifo_rddata = 4'h0; 40: fifo_rddata = 4'h0; 41: fifo_rddata = 4'h0; 42: fifo_rddata = 4'h0; 43: fifo_rddata = 4'h0; 44: fifo_rddata = 4'h0; 45: fifo_rddata = 4'h0; </pre>	<pre> 09: fifo_rddata = 8'h07; 10: fifo_rddata = 8'hed; 11: fifo_rddata = 8'hac; 12: fifo_rddata = 8'h62; 13: fifo_rddata = 8'h00; //sender ip : 192.168.0.2 14: fifo_rddata = 8'hc0; 15: fifo_rddata = 8'ha8; 16: fifo_rddata = 8'h00; 17: fifo_rddata = 8'h02; //target mac 18: fifo_rddata = 8'h00; 19: fifo_rddata = 8'h00; 20: fifo_rddata = 8'h00; 21: fifo_rddata = 8'h00; 22: fifo_rddata = 8'h00; 23: fifo_rddata = 8'h00; //target ip : 192.168.0.3 24: fifo_rddata = 8'hc0; 25: fifo_rddata = 8'ha8; 26: fifo_rddata = 8'h00; 27: fifo_rddata = 8'h03; //填充字段, //以使整个数据帧长度达到 64 字节 28: fifo_rddata = 8'h00; 29: fifo_rddata = 8'h00; 30: fifo_rddata = 8'hff; 31: fifo_rddata = 8'hff; 32: fifo_rddata = 8'hff; 33: fifo_rddata = 8'hff; 34: fifo_rddata = 8'hff; 35: fifo_rddata = 8'hff; 36: fifo_rddata = 8'h00; 37: fifo_rddata = 8'h23; 38: fifo_rddata = 8'hcd; 39: fifo_rddata = 8'h76; 40: fifo_rddata = 8'h63; 41: fifo_rddata = 8'h1a; 42: fifo_rddata = 8'h08; 43: fifo_rddata = 8'h06; </pre>
--	--

```
46: fifo_rddata = 4'h0;
47: fifo_rddata = 4'h0;

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

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

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

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

```
44: fifo_rddata = 8'h00;
45: fifo_rddata = 8'h01;
default: fifo_rddata = 8'd0;

endcase
end
```

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	

用户实验时，只需要先将自己的电脑 IP 地址设置为程序中询问的 192.168.0.3，然后将工程编译结果 sof 文件下载到 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 地址为使用 Altera 拥有的 MAC 地址段加上我们实验用的开发板型号组成，因此 wireshark 软件自带识别出了硬件名称并将 source 名称显示为 Altera）

The screenshot shows the Wireshark interface with a packet list table containing 10 rows of ARP traffic. The 'Filter' bar is empty, and the 'Expression...' button is visible. The table columns are No., Time, Source, Destination, Protocol, Length, and Info.

No.	Time	Source	Destination	Protocol	Length	Info
83	11.128567000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
84	11.128575000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
85	11.129445000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
86	11.129511000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
87	11.130469000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
88	11.130477000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
89	11.131458000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
90	11.131466000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
91	11.132483000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
92	11.132494000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
93	11.133501000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2

图 0-10

本节实验从应用上来讲，并没有任何实际意义，因为 CRC 校验值都是用软件生成的，数据内容也都是确定的，但是从学习角度，仍旧具有非常重要的价值。

本节实验介绍了 ARP 数据报的格式，并向着使用 FPGA 成功发送数据报的目的出发，过程中介绍了小兵以太网测试仪软件的使用、CRC 计算软件的使用，分别提供了基于 MII 百兆接口和 GMII/RGMII 千兆接口的 ARP 发包实例，然后介绍使用 Wireshark 软件抓取以太网包和分析的基本流程。为后续继续学习打下了坚实的基础。

以太网 ARP 帧发包实例（自动 CRC）

工程源码	--盘 A_AC620 开发板标准配套资料 ----02_设计实例 -----AC620_Verilog_Example -----以太网 -----arp_send_mii_with_crc.rar(百兆 MII 接口) -----arp_send_rgmii_with_crc.rar(千兆 RGMII 接口)
相关视频课程	--小梅哥 FPGA 入门到实战特训班课程 ----0808_02 以太网基本概念与 MAC 层协议介绍.mp4 ----0808_03 MAC 层数据传输接口 MII_GMII_RGMII.mp4 ----0809_01 以太网 MAC 层板级调试方法.mp4 ----0809_02 以太网 IP 层协议详解.mp4 ----0810_02 UDP 协议详解.mp4 ----0812_01 以太网传输项目总结.mp4

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。

以太网报文的校验字段 FCS 的计算

以太网报文校验字段 FCS 采用的是 CRC32 计算，关于 CRC 计算的 Verilog 代码实现已经做的很成熟，网上也有可直接生成 CRC 计算 Verilog 代码的网站。关于 CRC 计算原理感兴趣的可自行网上查找相关资料。这里主要是通过一个网站在线生成 CRC32 的 Verilog 代码，然后通过仿真对代码进行仿真验证。本例以千兆以太网的 GMII/RGMII 接口的 8 位数据模式为例进行介绍。

在线生成 CRC32 的 Verilog 代码的网站如下，点击进入网站是如下界面。

<http://www.easics.com/webtools/crctool>,

Polynomial

Specify the generator polynomial by clicking on the polynomial terms, or select a predefined value from the list below.

1

x

x²

x³

x⁴

x⁵

x⁶

x⁷

x⁸

x⁹

x¹⁰

x¹¹

x¹²

x¹³

x¹⁴

x¹⁵

x¹⁶

x¹⁷

x¹⁸

x¹⁹

x²⁰

x²¹

x²²

x²³

x²⁴

x²⁵

x²⁶

x²⁷

x²⁸

x²⁹

x³⁰

x³¹

x³²

x³³

x³⁴

x³⁵

x³⁶

x³⁷

x³⁸

x³⁹

x⁴⁰

x⁴¹

x⁴²

x⁴³

x⁴⁴

x⁴⁵

x⁴⁶

x⁴⁷

x⁴⁸

x⁴⁹

x⁵⁰

x⁵¹

x⁵²

x⁵³

x⁵⁴

x⁵⁵

x⁵⁶

x⁵⁷

x⁵⁸

x⁵⁹

x⁶⁰

x⁶¹

x⁶²

x⁶³

x⁶⁴

x⁶⁵

x⁶⁶

x⁶⁷

1

Select a predefined value:

2

Data Width

Select the number of data bits to be processed in one step:

3

1

Output Language

VHDL

Verilog

4

VHDL Options

Select the bit-vector type to be used:

5

std_logic_vector

Download

6

(1) CRC 生成多项式的设置，可根据多项式在窗口选择点击多项式中系数为 1 的 x 的幂次方。比如这里的以太网的 CRC32 的多项式为 $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ ，则只需将 x^{32} 、 x^{26} 、 x^{23} 、 x^{22} 、 x^{16} 、 x^{12} 、 x^{11} 、 x^{10} 、 x^8 、 x^7 、 x^5 、 x^4 、 x^2 、 x 、 1 点击选择上，在窗口下方会根据你点击选择的 x 的幂次方会将 CRC 生成公式呈现出来，可与以太网的 CRC 生成多项式进行比较核对，确保多项式的设置没有问题。

Polynomial

Specify the generator polynomial by clicking on the polynomial terms, or select a predefined value from the list below.

1

x

x²

x³

x⁴

x⁵

x⁶

x⁷

x⁸

x⁹

x¹⁰

x¹¹

x¹²

x¹³

x¹⁴

x¹⁵

x¹⁶

x¹⁷

x¹⁸

x¹⁹

x²⁰

x²¹

x²²

x²³

x²⁴

x²⁵

x²⁶

x²⁷

x²⁸

x²⁹

x³⁰

x³¹

x³²

x³³

x³⁴

x³⁵

x³⁶

x³⁷

x³⁸

x³⁹

x⁴⁰

x⁴¹

x⁴²

x⁴³

x⁴⁴

x⁴⁵

x⁴⁶

x⁴⁷

x⁴⁸

x⁴⁹

x⁵⁰

x⁵¹

x⁵²

x⁵³

x⁵⁴

x⁵⁵

x⁵⁶

x⁵⁷

x⁵⁸

x⁵⁹

x⁶⁰

x⁶¹

x⁶²

x⁶³

x⁶⁴

x⁶⁵

x⁶⁶

x⁶⁷

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

(2) 这里同样是 CRC 生成多项式的设置，与步骤①不同的地方是这里提供了几种常用的 CRC 生成多项式供选择，对于以太网的 CRC 就直接在下拉框中选择 CRC32 Ethernet/AAL5 即可，如下图所示。这里的不同的选择，同样会在步骤 1 中的窗口中同步更新显示，步骤 2 与步骤 1 都是 CRC 生成多项式的设置，步骤 1 更加灵活，可设置任意的 CRC 生成多项式，而步骤 2 就只能选择几种常见多项式中的一种。两个地方只要选择一个地方进行设置即可。

Select a predefined value:

CRC32 Ethernet/AAL5

CRC32 Ethernet/AAL5

CRC16 USB Data

CRC16 X25

CRC10

CRC8 ATM HEC

CRC5 USB Token

Data Width

Select the number of data bits to be processed in one step:

(3) 设置输入数据流的位宽，如果是 MII 接口，则设置为 4，如果是用于 GMII/RGMII 接口则设置为 “8”。

(4) 选择生成 HDL 代码的语言，选择 “Verilog”。

(5) 设置 VHDL 语言下要使用的位矢量类型，步骤 4 选择的是 Verilog 语言，就没有需要设置这个的地方。

(6) 下载生成 Verilog 代码的入口，点击可进行下载。

配置完后的界面如下，这个网站设置后有时候更新很慢，需要等待一会时间。

Polynomial

Specify the generator polynomial by clicking on the polynomial terms, or select a predefined value from the list below.

1	x	x ²	x ³	x ⁴	x ⁵	x ⁶	x ⁷	x ⁸	x ⁹	x ¹⁰	x ¹¹	x ¹²	x ¹³	x ¹⁴	x ¹⁵	x ¹⁶
x ¹⁷	x ¹⁸	x ¹⁹	x ²⁰	x ²¹	x ²²	x ²³	x ²⁴	x ²⁵	x ²⁶	x ²⁷	x ²⁸	x ²⁹	x ³⁰	x ³¹	x ³²	x ³³
x ³⁴	x ³⁵	x ³⁶	x ³⁷	x ³⁸	x ³⁹	x ⁴⁰	x ⁴¹	x ⁴²	x ⁴³	x ⁴⁴	x ⁴⁵	x ⁴⁶	x ⁴⁷	x ⁴⁸	x ⁴⁹	x ⁵⁰
x ⁵¹	x ⁵²	x ⁵³	x ⁵⁴	x ⁵⁵	x ⁵⁶	x ⁵⁷	x ⁵⁸	x ⁵⁹	x ⁶⁰	x ⁶¹	x ⁶²	x ⁶³	x ⁶⁴	x ⁶⁵	x ⁶⁶	x ⁶⁷

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

Select a predefined value:

CRC32 Ethernet/AAL5

Data Width

Select the number of data bits to be processed in one step:

8

Output Language

☐ VHDL ☒ Verilog

[Download](#)

点击 **Download**，保存生成一个命名为 CRC32_D8 的代码文件，生成的具体代码如下。

```
module CRC32_D8;

    // polynomial: x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 +
    // x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1
    // data width: 8
    // convention: the first serial bit is D[7]
    function [31:0] nextCRC32_D8;

        input [7:0] Data;
        input [31:0] crc;
        reg [7:0] d;
        reg [31:0] c;
        reg [31:0] newcrc;
    begin
        d = Data;
        c = crc;

        newcrc[0] = d[6] ^ d[0] ^ c[24] ^ c[30];
        newcrc[1] = d[7] ^ d[6] ^ d[1] ^ d[0] ^ c[24] ^ c[25] ^ c[30] ^ c[31];
```

```

newcrc[2] = d[7] ^ d[6] ^ d[2] ^ d[1] ^ d[0] ^ c[24] ^ c[25] ^ c[26] ^
c[30] ^ c[31];
newcrc[3] = d[7] ^ d[3] ^ d[2] ^ d[1] ^ c[25] ^ c[26] ^ c[27] ^ c[31];
newcrc[4] = d[6] ^ d[4] ^ d[3] ^ d[2] ^ d[0] ^ c[24] ^ c[26] ^ c[27] ^
c[28] ^ c[30];
newcrc[5] = d[7] ^ d[6] ^ d[5] ^ d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[24] ^
c[25] ^ c[27] ^ c[28] ^ c[29] ^ c[30] ^ c[31];
newcrc[6] = d[7] ^ d[6] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ c[25] ^ c[26] ^
c[28] ^ c[29] ^ c[30] ^ c[31];
newcrc[7] = d[7] ^ d[5] ^ d[3] ^ d[2] ^ d[0] ^ c[24] ^ c[26] ^ c[27] ^
c[29] ^ c[31];
newcrc[8] = d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[0] ^ c[24] ^ c[25] ^ c[27] ^
c[28];
newcrc[9] = d[5] ^ d[4] ^ d[2] ^ d[1] ^ c[1] ^ c[25] ^ c[26] ^ c[28] ^
c[29];
newcrc[10] = d[5] ^ d[3] ^ d[2] ^ d[0] ^ c[2] ^ c[24] ^ c[26] ^ c[27] ^
c[29];
newcrc[11] = d[4] ^ d[3] ^ d[1] ^ d[0] ^ c[3] ^ c[24] ^ c[25] ^ c[27] ^
c[28];
newcrc[12] = d[6] ^ d[5] ^ d[4] ^ d[2] ^ d[1] ^ d[0] ^ c[4] ^ c[24] ^
c[25] ^ c[26] ^ c[28] ^ c[29] ^ c[30];
newcrc[13] = d[7] ^ d[6] ^ d[5] ^ d[3] ^ d[2] ^ d[1] ^ c[5] ^ c[25] ^
c[26] ^ c[27] ^ c[29] ^ c[30] ^ c[31];
newcrc[14] = d[7] ^ d[6] ^ d[4] ^ d[3] ^ d[2] ^ c[6] ^ c[26] ^ c[27] ^
c[28] ^ c[30] ^ c[31];
newcrc[15] = d[7] ^ d[5] ^ d[4] ^ d[3] ^ c[7] ^ c[27] ^ c[28] ^ c[29] ^
c[31];
newcrc[16] = d[5] ^ d[4] ^ d[0] ^ c[8] ^ c[24] ^ c[28] ^ c[29];
newcrc[17] = d[6] ^ d[5] ^ d[1] ^ c[9] ^ c[25] ^ c[29] ^ c[30];
newcrc[18] = d[7] ^ d[6] ^ d[2] ^ c[10] ^ c[26] ^ c[30] ^ c[31];
newcrc[19] = d[7] ^ d[3] ^ c[11] ^ c[27] ^ c[31];
newcrc[20] = d[4] ^ c[12] ^ c[28];
newcrc[21] = d[5] ^ c[13] ^ c[29];
newcrc[22] = d[0] ^ c[14] ^ c[24];
newcrc[23] = d[6] ^ d[1] ^ d[0] ^ c[15] ^ c[24] ^ c[25] ^ c[30];
newcrc[24] = d[7] ^ d[2] ^ d[1] ^ c[16] ^ c[25] ^ c[26] ^ c[31];
newcrc[25] = d[3] ^ d[2] ^ c[17] ^ c[26] ^ c[27];
newcrc[26] = d[6] ^ d[4] ^ d[3] ^ d[0] ^ c[18] ^ c[24] ^ c[27] ^ c[28]
^ c[30];
newcrc[27] = d[7] ^ d[5] ^ d[4] ^ d[1] ^ c[19] ^ c[25] ^ c[28] ^ c[29]
^ c[31];
newcrc[28] = d[6] ^ d[5] ^ d[2] ^ c[20] ^ c[26] ^ c[29] ^ c[30];
newcrc[29] = d[7] ^ d[6] ^ d[3] ^ c[21] ^ c[27] ^ c[30] ^ c[31];
newcrc[30] = d[7] ^ d[4] ^ c[22] ^ c[28] ^ c[31];

```

```

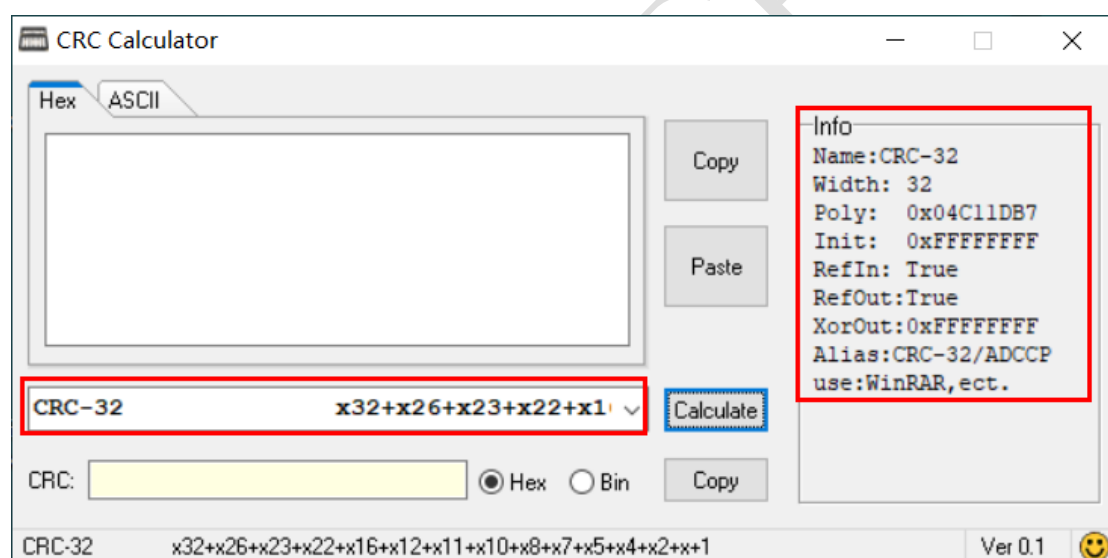
newcrc[31] = d[5] ^ c[23] ^ c[29];
nextCRC32_D8 = newcrc;

end
endfunction
endmodule

```

生成的代码里面仅为一个计算 CRC32 的一个函数 function。CRC 的计算与除了与生成多项式相关外，还与 CRC 的初始值，输入输出数据是否位反向以及计算结果是否取反等操作相关。上面生成的 CRC 计算的 Verilog 代码仅仅是与多项式有关的计算过程，未涉及到初始值等因素，所以要实现符合以太网要求的 CRC32 的完整计算需要添加一些计算操作。

再通过 CRC 计算工具回顾下以太网 CRC32 的计算的各种条件，打开 CRC 计算工具 CRC_Calculator，软件的界面如下，多项式选择 CRC-32。



从软件右边的 Info 可以看到以太网 CRC32 计算的相关信息，具体含义如下。

- Name: 参数模型名称。
- Width: 宽度，即 CRC 比特数。
- Poly: 生成项的简写，以 16 进制表示。例如：CRC-32 即是 0x04C11DB7，忽略了最高位的"1"，即完整的生成项是 0x104C11DB7。
- Init: 这是算法开始时 CRC 的初始化预置值，十六进制表示。

- Refin: 待测数据的每个字节是否按位反转, True 或 False。
- Refout: 在计算后之后, 异或输出之前, 整个数据是否按位反转, True 或 False。
- Xorout: 计算结果与此参数异或后得到最终的 CRC 值。

根据上面的信息, 可以了解到计算 CRC32 与初始值, 输入输出数据的操作有关, 所以要将生成的 CRC 计算的 verilog 代码用在以太网上还需要做一定的添加修改, 首先需要对该模块添加输入输出端口。代码如下。

```
module crc32_d8
(
    input      clk      ,
    input      reset_n  ,

    input  [7:0] data    ,
    input      crc_init  ,
    input      crc_en    ,
    output [31:0] crc_result
);
```

端口信号含义如下

端口名	方向	描述
clk	input	模块工作时钟
reset_n	input	模块复位, 低电平有效
data[7:0]	input	需要计算 CRC 校验值的输入数据, 位宽为 8bit
crc_init	input	设置 CRC 计算的的初始值的使能信号, 为高时, 将 crc_result 为初始值 32 'hfffffff
crc_en	input	计算 CRC 的使能信号, 当在该信号为 1 时, 计算输入数据的 CRC 值
crc_result[31:0]	output	计算的 CRC 结果

CRC32 的 CRC 的初始值为 0xFFFFFFFF, 在每次计算 CRC 前需将 CRC 的值回归到初始值, 然后在需要的时候开始计算 CRC, 初始值 crc_init 为高电平的时候回到初始值, 具体实现代码如下。

```
always @(posedge clk or negedge reset_n)
begin
```

```

if(!reset_n)
    crc_result_o <= 32'hffff_ffff;
else if(crc_init)
    crc_result_o <= 32'hffff_ffff;
else if(crc_en)
    crc_result_o <= nextCRC32_D8( data_i, crc_result_o);
else
    crc_result_o <= crc_result_o;
end

```

CRC 计算前对输入的数据有按位反转的要求，也就是需要高低 bit 调换顺序。具体代码如下。

```

assign data_i = {data[ 0],data[ 1],data[ 2],data[ 3],
                 data[ 4],data[ 5],data[ 6],data[ 7]};

```

CRC 计算后的结果也需要做一定的处理，根据要求需要进行先进行位反转，然后与数据 0xFFFFFFFF 进行位异或操作（也就是取反，0 与 1 异或为 1，1 与 1 异或为 0）。这两个操作可同时进行，具体代码如下。

```

assign crc_result =
~{crc_result_o[00],crc_result_o[01],crc_result_o[02],crc_result_o[03],
  crc_result_o[04],crc_result_o[05],crc_result_o[06],crc_result_o[07],
  crc_result_o[08],crc_result_o[09],crc_result_o[10],crc_result_o[11],
  crc_result_o[12],crc_result_o[13],crc_result_o[14],crc_result_o[15],
  crc_result_o[16],crc_result_o[17],crc_result_o[18],crc_result_o[19],
  crc_result_o[20],crc_result_o[21],crc_result_o[22],crc_result_o[23],
  crc_result_o[24],crc_result_o[25],crc_result_o[26],crc_result_o[27],
  crc_result_o[28],crc_result_o[29],crc_result_o[30],crc_result_o[31]};

```

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。

自动 CRC 校验实验测试

crc32_d4.v 和 crc32_d8 分别为百兆 MII 接口和千兆 GMII/RGMII 接口的 4 位 /8 位数据位宽的并行 32 位 CRC 计算单元。

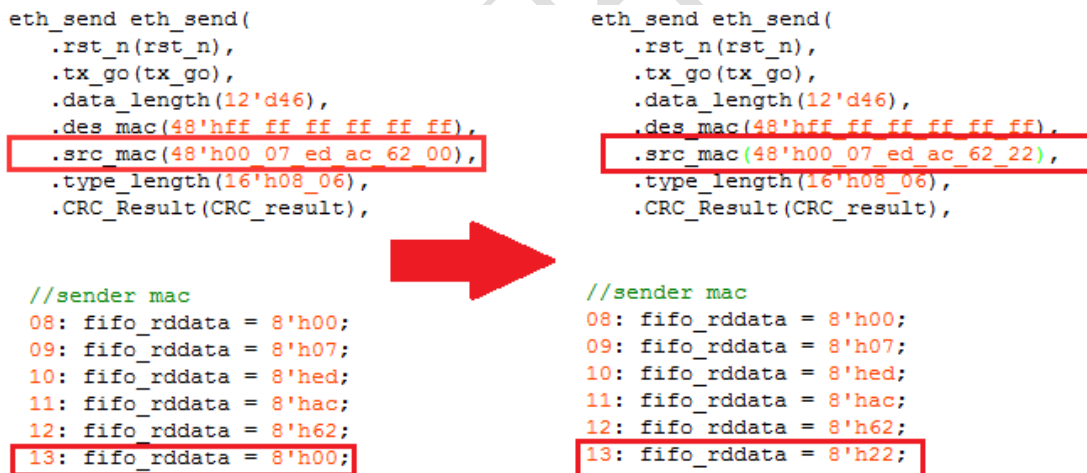
eth_send.v 为以太网 MAC 帧组包逻辑，完成以太网 MAC 帧的建立和发送。

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。这里继续使用 00_07_ed_ac_62_00 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 46。

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。

以下简单做个测试，分别使用 00_07_ed_ac_62_00 和 00_07_ed_ac_62_22 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 eth_send_test 模块中的以下内容：



```
eth_send eth_send(  
    .rst_n(rst_n),  
    .tx_go(tx_go),  
    .data_length(12'd46),  
    .des_mac(48'hff ff ff ff ff ff),  
    .src_mac(48'h00_07_ed_ac_62_00),  
    .type_length(16'h08_06),  
    .CRC_Result(CRC_result),  
  
    //sender mac  
    08: fifo_rddata = 8'h00;  
    09: fifo_rddata = 8'h07;  
    10: fifo_rddata = 8'hed;  
    11: fifo_rddata = 8'hac;  
    12: fifo_rddata = 8'h62;  
    13: fifo_rddata = 8'h00;  
  
eth_send eth_send(  
    .rst_n(rst_n),  
    .tx_go(tx_go),  
    .data_length(12'd46),  
    .des_mac(48'hff ff ff ff ff ff),  
    .src_mac(48'h00_07_ed_ac_62_22),  
    .type_length(16'h08_06),  
    .CRC_Result(CRC_result),  
  
    //sender mac  
    08: fifo_rddata = 8'h00;  
    09: fifo_rddata = 8'h07;  
    10: fifo_rddata = 8'hed;  
    11: fifo_rddata = 8'hac;  
    12: fifo_rddata = 8'h62;  
    13: fifo_rddata = 8'h22;
```

图 0-11

下图为分别设置源 MAC 为 00_07_ed_ac_62_00 和 00_07_ed_ac_62_22 时，PC 端以太网抓包工具抓到的数据。

Top Screenshot Data:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
2	0.000042000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
3	0.134055000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
4	0.134092000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
5	0.268214000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
6	0.268261000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
7	0.402474000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
8	0.402511000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
9	0.536703000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
10	0.536740000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
11	0.670885000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
12	0.670926000	84:7b:eb:48:94:13	Altera_ac:62:00	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
13	0.805033000	Altera_ac:62:00	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2

Bottom Screenshot Data:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
2	0.000015000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
3	0.134262000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
4	0.134302000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
5	0.268505000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
6	0.268542000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
7	0.402643000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
8	0.402681000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
9	0.536922000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
10	0.536960000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
11	0.671061000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2
12	0.671099000	84:7b:eb:48:94:13	Altera_ac:62:22	ARP	42	192.168.0.3 is at 84:7b:eb:48:94:13
13	0.805271000	Altera_ac:62:22	Broadcast	ARP	60	who has 192.168.0.3? Tell 192.168.0.2

图 0-12

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。

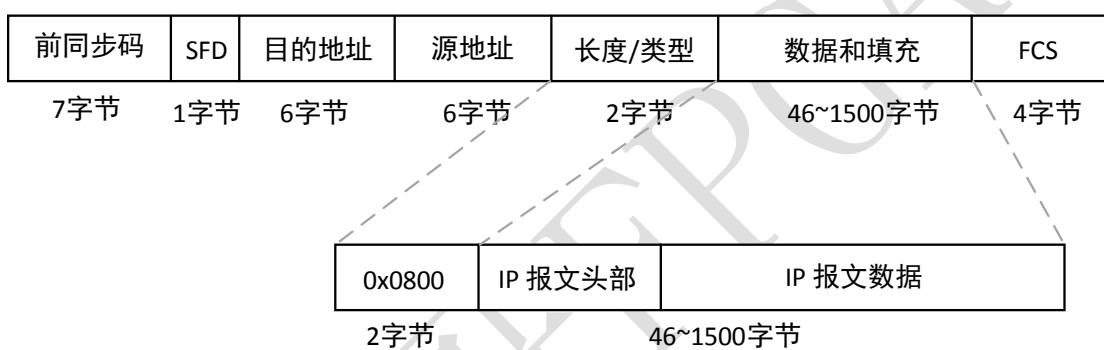
总结

至此，我们完成了以太网首发过程中最底层，也是最重要的部分——MAC 帧的发送工作，并实现了 GMII/RGMII 和 MII 两种接口的 ARP 协议报文的发送实验。本实验的最终目的不在于完成 ARP 协议的发送。事实上，对于 ARP 协议的内容，到此为止，后续也不会再应用，本节内容通过最简单的应用协议，让大家掌握了以太网开发调试过程中各种软件的操作方法，并解决了以太网帧中的难点——CRC 校验算法。在后续的内容中，我们还会多次用到本实验中各种软件的使用和开发思路。

IP 协议介绍与 IP 校验和算法实现

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收。

IP 数据报的格式如下，IP 数据报的长度 / 类型段的数值为 0x0800，数据和填充段包括 IP 头部数据和 IP 数据两个部分。



其中，和以太网帧具有帧头一样，IP 数据报也包含了一个 IP 报头部分，与 IP 协议相关的一些信息如 IP 地址，数据包长度等会被打包进 IP 报头中，然后再与需要传输的 IP 报文数据一起，作为 MAC 帧的数据和填充字段送往 MAC 层发送，接下来，将着重介绍 IP 协议首部的详细信息。

IP 协议首部详解

IP 数据报的格式如下：

0~3	4~7	7~15	16~18			19~31		
版本	首部长度	服务类型	总长度					
分段标识，标识属于同一数据报			NC	DF	MF	段偏移		
生存周期 TTL(秒)		上层协议	报头校验和					
源 IP 地址								
目的 IP 地址								
可选字段（可无）								
IP 数据报数据部分，N 个字节								

其中每个字段的功能简介如下表所示

IP	版本号	4 位	IP 协议版本	版本就是 IPV4 或者 IPV6，一般选择
----	-----	-----	---------	------------------------

字 段				IPV4, 即版本值为 4。
	首部长	4 位	IP 首部长	IP 首部长即有多少个 32 位数, 当 IP 首部长为 20 时(即无可选字段), 该值为 5。(5*4=20)
	服务类型	8 位	指示了报文的优先权等	简单的发送可以全部置 0 即可
	总长度	16 位	IP 报文长度	IP 报文(报头+数据)长度
	分段标识	16 位	是否属于同一数据段	IP 报文的分段 ID
	段标识和段偏移	16 位	可设为 0	简单的发送可以全部置 0 即可
	生存周期 (TTL)	8 位	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、64 或 128), 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于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。
	上层协议类型	8 位	指该 IP 包中数据段内容所使用的网络协议类型, 如 ICMP、DNS 等。	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	对 IP 报头计算得出
	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

下表为 IP 数据报中上层协议字段常见协议与其对应的协议号。

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	OSPF
17	UDP	94	IPIP
20	HMP	98	ENCAP

在 IP 报头中，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，可选字段则可有可无。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。以下对 IP 报头中各个字段的内容详细介绍。

- 版本为当前使用的 IP 协议版本号，大家最熟悉，用的最多的是 IPv4 版本，当然现在 IPv6 版本用的也越来越多了。本实验使用的是 IPv4 版本，所以在版本字段为 4 就表示版本号为 IPv4。
- 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。
- 服务类型：略。
- 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。
- 报头校验和指 IPv4 数据报报头的校验和。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。
- 源 IP 地址和目的 IP 地址则是该 IP 报文的发送方和接收方的网络地址，

例如在我们前面的实验中，说 FPGA 的 IP 地址为 192.168.0.2，电脑的 IP 地址为 192.168.0.3，那么 FPGA 在向电脑发送数据包时，源 IP 地址就是 192.168.0.2，目的 IP 地址为 192.168.0.3。

IP 首部校验和算法介绍

首部校验和字段是根据 IP 首部计算的校验和码，不对首部后面的数据进行计算。具体的计算和实现方法将在后文详细介绍。其计算方法为：

1. 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；
2. 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 报头可如下图中阴影部分标记内容所示：

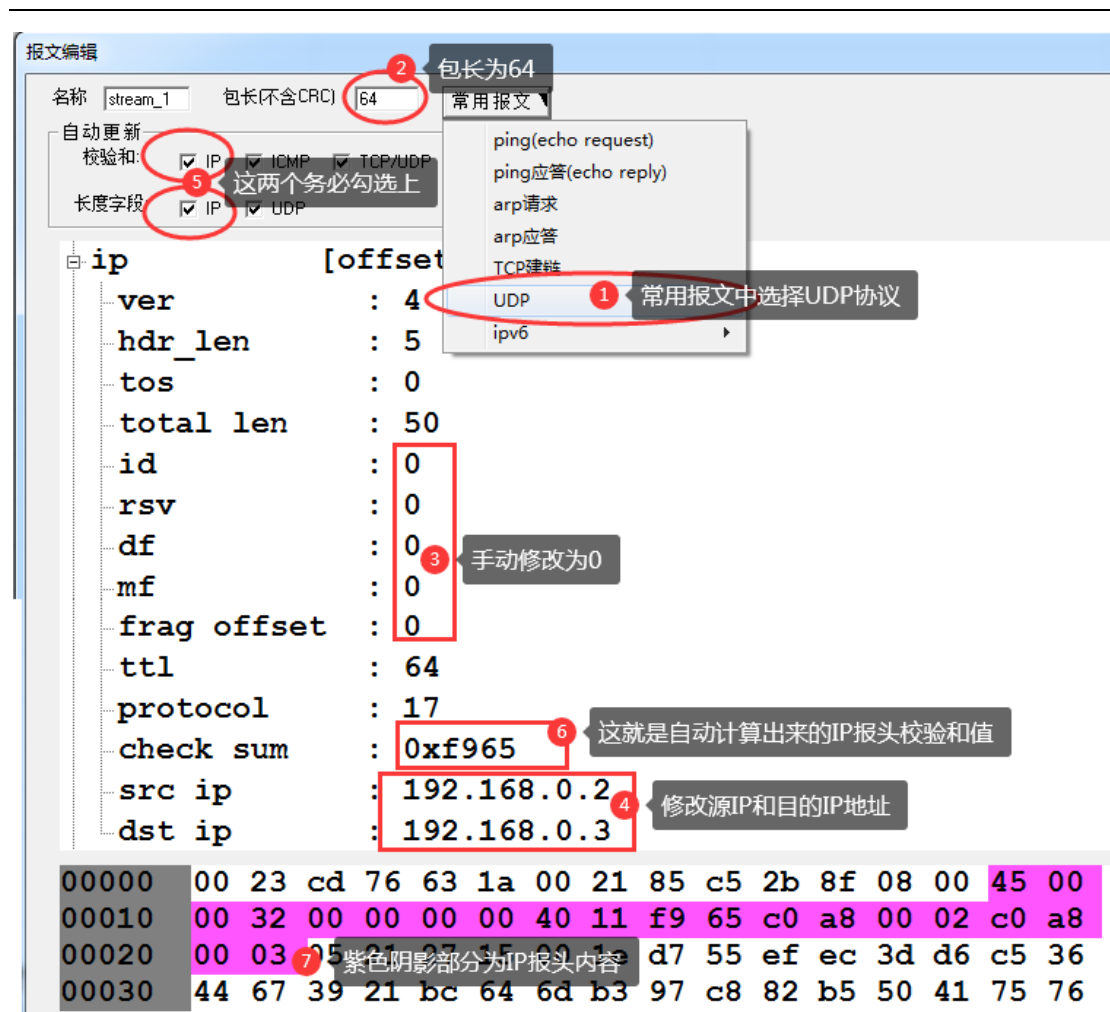


图 0-13

IP 报头中每个字段的内容详细如下表所示:

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:

$0x4500 + 0x0032 + 0x0000 + 0x0000 + 0x4011 + 0x0000$ (计算时强制置 0) +

0xc0a8 + 0x0002 + 0xc0a8 + 0x0003 = 0x20698。

0x0002 + 0x0698 = 0x069a

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。以下为使用 Verilog 编写的 checksum 的逻辑代码。

```
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
```

该实现方法看似简单且合理，但是在实际使用过程中，有读者反映在某些情况下会发生错误，该读者举了一个例子，假设某帧数据的 IP 报头如下所示：

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	0x05	0xd6	0x70	0xf6	0x40	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			
0x80	0x11			0xc0	0xa8	0x0	0x6e
64 级	17 UDP			192	168	0	2
目标地址 dst_ip							
0xc0	0xa8	0x01	0x64				
192	168	0	3				

该数据帧中，如果按照上述方法计算，

第一步：4500+05D6+70F6+4000+8011+C0A8+016E+C0A8+0164=2FFFF

第二步：ffff + 2 = 1

第三步：~1 = fffe

既此算法得到的结果为 0xffffe，而使用以太网发送时（或者使用小兵以太网测试仪），实际的值为 0xfffd。

出错的原因在于 ffff + 2 会发生溢出，而上述设计没有考虑到该情况。解决方案很简单，对第二步的计算结果再根据高 16 位的值判断是否需要再执行一次加法操作。

因此可将设计改进如下：

```

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]suma,sumb;

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

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

endmodule

```

根据 IP 头部校验计算步骤，代码中 `suma` 就是经过步骤 1 计算的结果，`sumb` 是步骤 2 计算的和。`checksum_result` 为根据步骤 3 需要根据高 16bit 是否为零来判断是否需要继续相加求和并取反输出的值。

部分网友在看到这里后产生了疑问，表示既然在第二步的时候出现了相加溢出的情况，那么现在加了一步，在步骤三中再根据步骤 2 结果的高 16 位是否不为 0 执行了再一次的相加操作，那么这一次的相加操作有没有可能继续溢出呢，是否还需要再循环加下去，直到高 16 位不为 0 了再输出呢，事实上这种担心是多余的，该结果完全可以使用理论计算的方式验证。

对于 `sumb = suma[31:16]+suma[15:0]` 来说，该公式包含了高 16bit 不为 0 继续相加求和，高 16bit 为 0 情况下，低 16bit 数据加 0。不管哪种情况 2 个 16bit 数据相加的结果取值范围为 17'h00001~17'h1fffe（当 16'hffff+16'hffff 时，求和值最大），也就是说在第三步再次进行求和时，出现的结果最大时候的情况就是 ffffe+1，其值为 ffff，高 16 位不可能再大于 0，所以无需再继续求和。

设计完成后，可以编写 `testbench` 对该算法进行计算，`testbench` 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

```

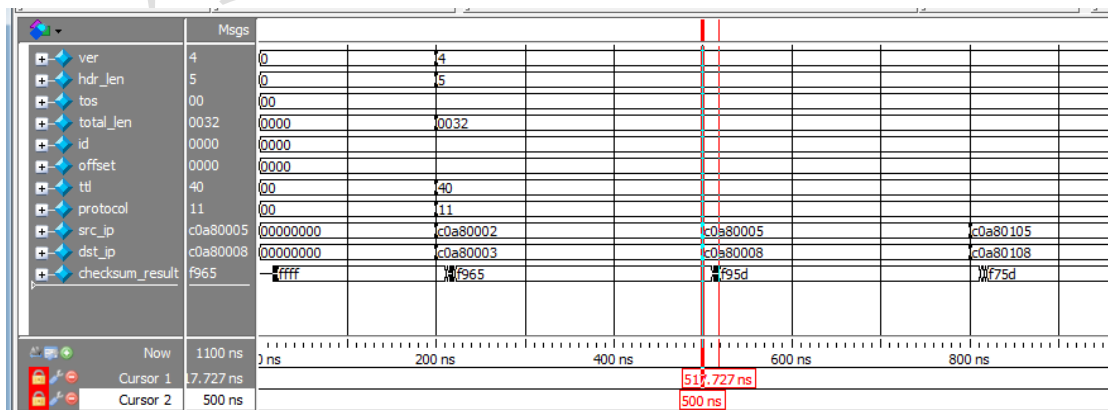


图 0-14

使用 Modelsim 进行时序仿真，可以验证该逻辑计算的结果与使用小兵以太网测试仪计算出的 IP 首部校验和一致，所以证明该算法实现是正确的。

另外，有读者曾经反应，整个算法实现全部使用的是组合逻辑，这样的情况会不会影响整个设计的时序性能。能否影响设计的时序性能，主要看该计算产生到该结果被使用时的时间差。通过 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，千兆以太网 MII 接口是中间最起码有 20 个时钟周期($20 \times 20\text{ns} = 400\text{ns}$)，千兆以太网 GMII 接口是 10 个时钟周期($10 \times 8\text{ns} = 80\text{ns}$)，即完全满足时序要求。

总结

至此，我们就完成了 IP 数据报中最重要的部分，IP 报头校验和的计算逻辑设计，同时，相信大家也已经了解了 IP 报头的具体内容，IP 数据包就是紧跟在 IP 报头后面的。然后，IP 的数据部分也还并不是单纯的用户数据，我们在网络应用时，还需要将我们的用户数据进一步打包到比 IP 协议更上一层的协议中，再通过 IP 协议发送，该内容我们将在下一节内容讲到。

UDP 协议原理与 FPGA 实现

UDP 协议介绍

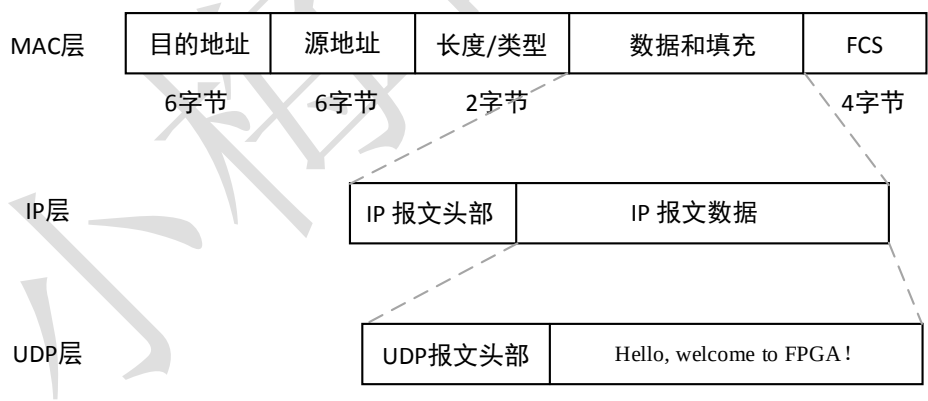
UDP 是 User Datagram Protocol 的简称，中文名是用户数据报协议，是 OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，IETF RFC 768 是 UDP 的正式规范。UDP 在 IP 报文的协议号是 17(即 0x11)。

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。

以太网 UDP 帧的用户数据是打包在 UDP 协议中，而 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 数据报文，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, welcome to FPGA！”。下图为使用 UDP 协议发送“Hello, welcome to FPGA！”数据的层层打包示意图。



其中，和以太网帧、IP 报文具有帧头一样，UDP 数据报也包含了一个 UDP 报头部分，与 UDP 协议相关的一些信息如端口号，数据包长度等会被打包进 UDP 报头中，然后再与需要传输的 UDP 报文数据一起，作为 IP 报文的数据段送往 IP 层发送。

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。下包为 UDP 报文的格式。

U D P 字 段	字段名称	长度	功能说明	解释
	源端口号	2 字节	例如 5000	发送方的网络端口地址
	目的端口号	2 字节	例如 6000	接收方的网络端口地址
	UDP 长度	2 字节	UDP 报文长度	UDP 报文（含数据）长度
	UDP 校验和	2 字节	UDP 报头校验和	UDP 协议的报头和数据之和校验
	UDP 数据	n 字节		

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 6102(0x17D6)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 数据段可以组包如下表所示：

0x13	0x88	0x17	0xD6
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)	'l'(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：

报文编辑

名称 stream_1 包长(不含CRC) 64 常用报文

自动更新

校验和: ☒ IP ☒ ICMP ☒ TCP/UDP

长度字段: ☒ IP ☒ UDP

ethernet [offset=0;length=14]

ip [offset=14;length=20]

udp [offset=34;length=8]

source port : 5000

dest port : 6102

length : 30

checksum : 0xb26e

data (offset=42;length=22)

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 d6 00 1e b2 6e 48 65 6c 6c 6f 2c |nHello,

00030 20 77 65 6c 63 6f 6d 20 74 6f 20 46 50 47 41 21 | welcom to FPGA!

图 0-15

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, welcom to FPGA!”。下图为使用 UDP 协议发送“Hello, welcom to FPGA!”的数据层层打包示意图：

用户数据				Hello, welcom to FPGA!	
UDP 层			UDP 报头	数据段	
IP 层		IP 报头	数据段		
MAC 层	MAC 帧头	数据段			FCS

即使用 UDP 协议发送“hello world！”，按照以下顺序进行：

1. 将“Hello, welcom to FPGA!”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。

以太网 UDP 帧发包实例（手动 IPchecksum）

工程源码	--盘 A_AC620 开发板标准配套资料 ----02_设计实例 -----AC620_Verilog_Example -----以太网 -----udp_send.qar 或 rar
相关视频课程	--小梅哥 FPGA 入门到实战特训班课程 ----0808_02 以太网基本概念与 MAC 层协议介绍.mp4 ----0808_03 MAC 层数据传输接口 MII_GMII_RGMII.mp4 ----0809_01 以太网 MAC 层板级调试方法.mp4 ----0809_02 以太网 IP 层协议详解.mp4 ----0810_02 UDP 协议详解.mp4 ----0812_01 以太网传输项目总结.mp4

本实验我们使用以太网发包工具构建一个 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_mii.rar（百兆 MII 接口）

和 udp_send_rgmii.rar（千兆 RGMII 接口）。

使用以太网测试仪构建 UDP 数据报

打开小兵以太网测试仪，新建一个数据流，如下图所示。

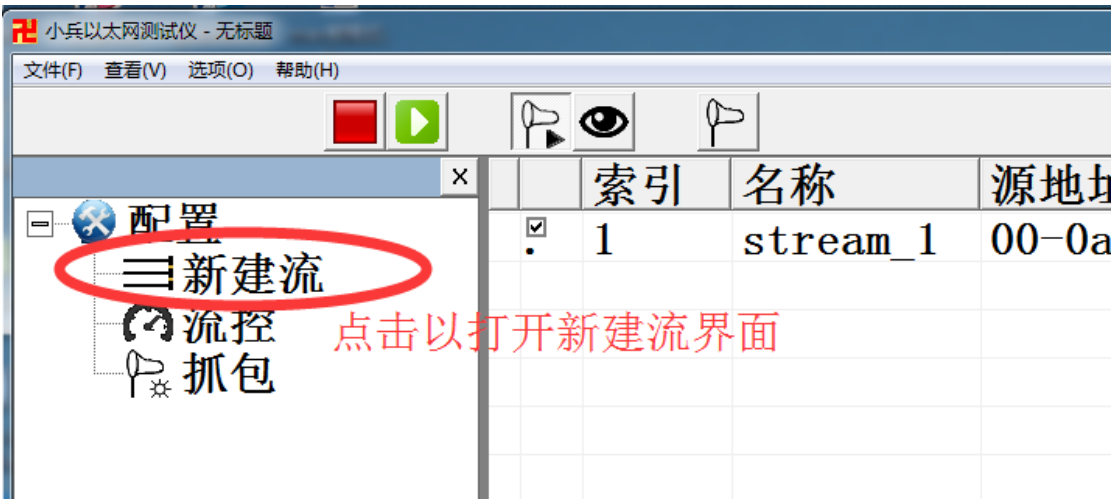


图 0-16

在常用报文中选择 UDP 格式报文，设置包长为 64，自动更新的所有项全部勾选上。如下图所示。

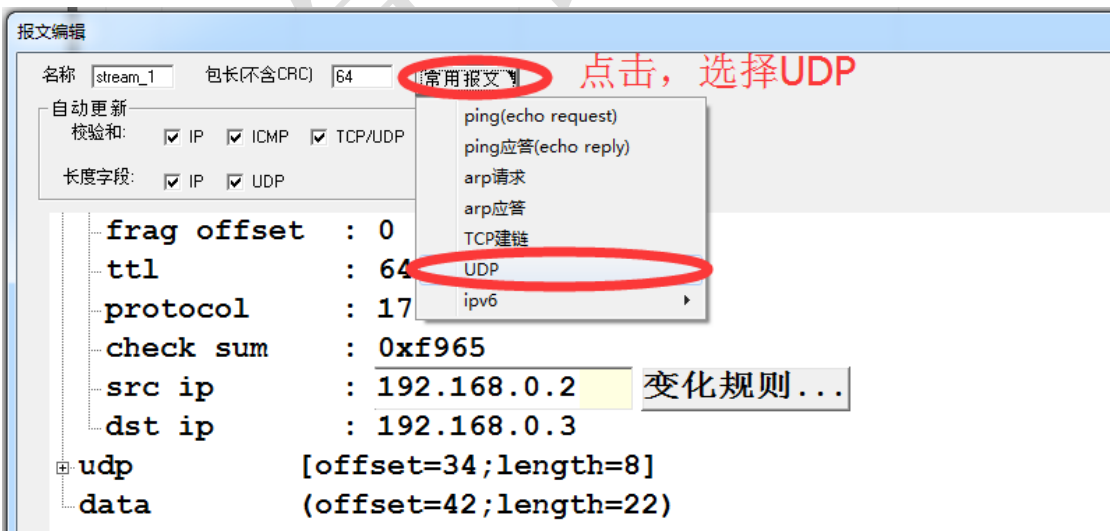


图 0-17

修改以太网包和 IP、UDP 包中各字段内容如下所示。

名称: stream_1 包长(不含CRC): 64 常用报文 ▼

自动更新

校验和: ☒ IP ☒ ICMP ☐ TCP/UDP

长度字段: ☒ IP ☒ 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 : 6102

length : 30

checksum : 0xb26e

data (offset=42;length=22)

数据段: Hello, welcom to FPGA!

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: 6102, 用户可自行修改

UDP数据包长度8+22 = 30字节

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 d6 00 1e b2 6e 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!

图 0-18

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

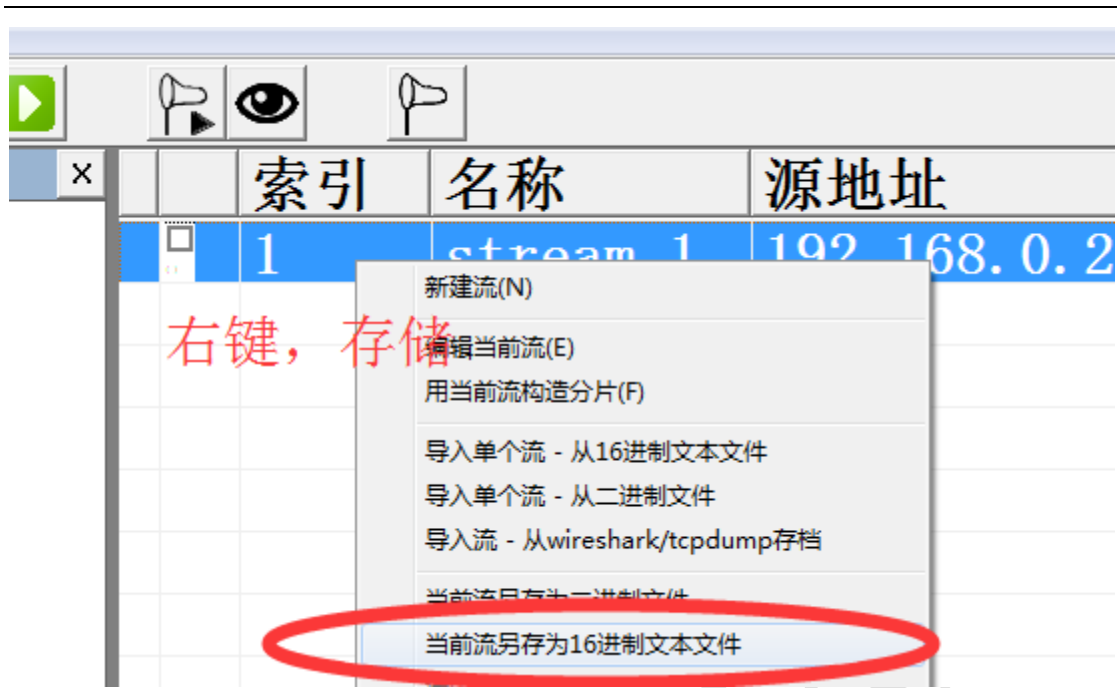


图 0-19

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, 0xd6, 0x00, 0x1e, 0xb2, 0x6e,
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。

使用 FPGA 发送 UDP 数据

修改 eth_send 的例化内容，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。这里继续使用前面生成的 Altera 地址段的 mac 地址，以避免与其他实际地址冲突，地址为 48'h00_07_ed_ac_62_00。协议类型为 IP (0800)。数据长度为 50。

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 UDP 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'd50),
    .des_mac(48'h84_7b_eb_48_94_13), //目的网卡号
    .src_mac(48'h00_07_ed_ac_62_00), //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：

*本地连接 [Wireshark 1.12.4 (v1.12.4-0-gb4861da from master-1.12)]										
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help										
Filter: Expression... Clear Apply Save										
No.	Time	Source	Destination	Protocol	Length	Info				
1	0.000000000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
2	0.670950000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
3	1.341877000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
4	2.012952000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
5	2.683825000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
6	3.354823000	192.168.0.2	192.168.0.3	UDP	64	Source port: 5000 Destination port: 6102				
Frame 9: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface 0										
Ethernet II, Src: Altera_ac:62:00 (00:07:ed:ac:62:00), Dst: 84:7b:eb:48:94:13 (84:7b:eb:48:94:13)										
Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 192.168.0.3 (192.168.0.3)										
Version: 4										
Header Length: 20 bytes										
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))										
Total Length: 50										
Identification: 0x0000 (0)										
Flags: 0x00										
Fragment offset: 0										
Time to live: 64										
Protocol: UDP (17)										
Header checksum: 0xf965 [validation disabled]										
Source: 192.168.0.2 (192.168.0.2)										
Destination: 192.168.0.3 (192.168.0.3)										
[Source GeoIP: Unknown]										
[Destination GeoIP: Unknown]										
User Datagram Protocol, Src Port: 5000 (5000), Dst Port: 6102 (6102)										
Source Port: 5000 (5000)										
Destination Port: 6102 (6102)										
Length: 30										
Checksum: 0xb26e [validation disabled]										
[Stream index: 0]										
Data (22 bytes)										
Data: 48656c6c662c2077656c63666d20746f204650474121										
[Length: 22]										
0000	84	7b	eb	48	94	13	00	07	ed ac 62 00 08 00 45 00	.{.H.... ..b...E.
0010	00	32	00	00	00	00	40	11	f9 65 c0 a8 00 02 c0 a8	.2....@. .e.....
0020	00	03	13	88	17	d6	00	1e	b2 6e 48 65 6c 6c 6f 2c	nHello,
0030	20	77	65	6c	63	6f	6d	20	74 6f 20 46 50 47 41 21	welcom to FPGA!

图 0-20

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 6102。数据内容为：“Hello, welcome to FPGA!”。

使用以太网助手接收数据

打开网络调试工具，设置协议类型为 UDP，本地 IP 地址为 192.168.0.3，本地端口号为 6102，然后点击连接，接着在目的 IP 地址和目的端口号处分别填入 192.168.0.2 和 5000，然后点击一次发送按钮（此处点击发送按钮的目的是为了通过这种方式激活刚刚设置的目的 IP 和端口号，并不是真的为了发数据），则软件上就会源源不断的开始收到 FPGA 板卡发出的数据内容了，如下图所示。



图 0-21

忽略 UDP 校验和测试

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。

```

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'he;
170 // 55 : fifo_rddata = 4'h6;
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;

```

图 0-22

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。

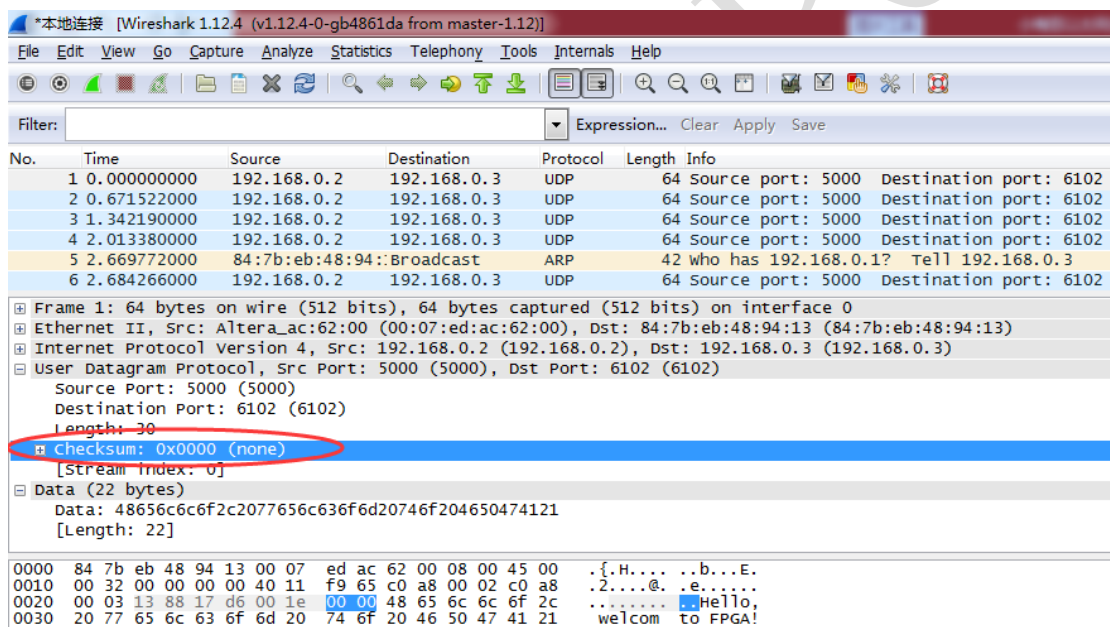


图 0-23

在网络调试工具中，依然可以正确的接收到数据。



图 0-24

错误 UDP 校验和测试

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。

```

166 // //UDP报头校验和
167 // 52 : fifo_rddata = 4'h2;
168 // 53 : fifo_rddata = 4'hb;
169 // 54 : fifo_rddata = 4'he;
170 // 55 : fifo_rddata = 4'h6;
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;

```

图 0-25

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。

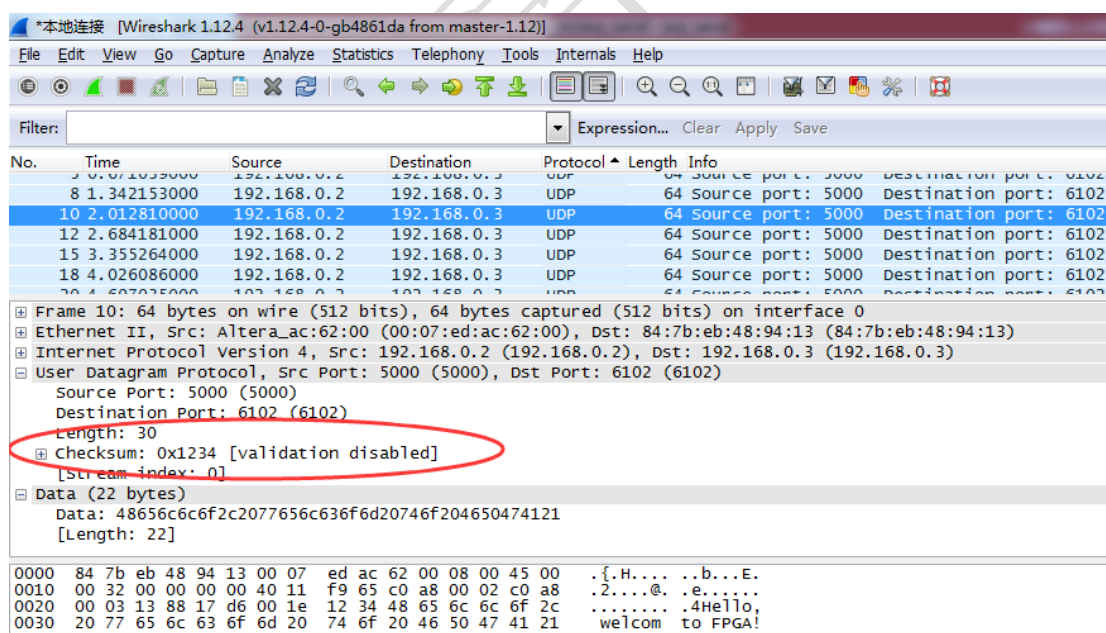


图 0-26

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。

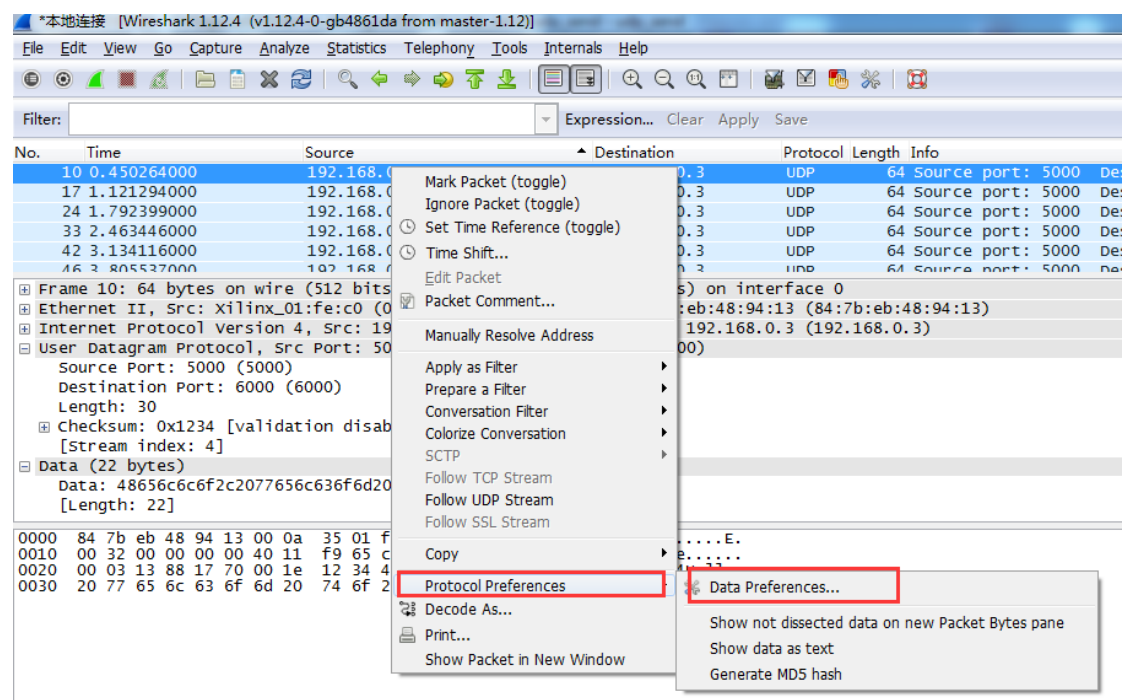


图 0-27

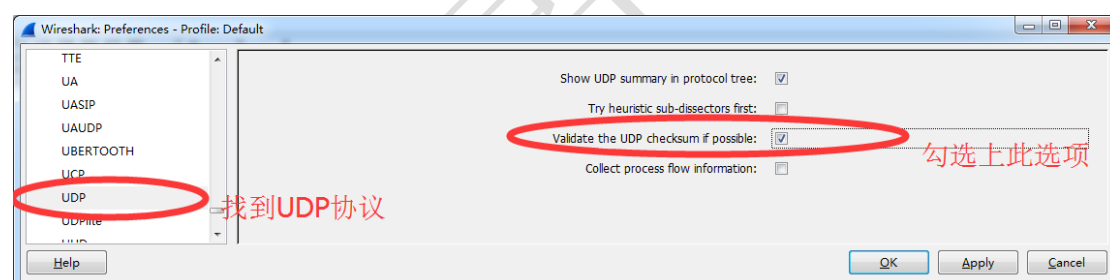


图 0-28

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，如下图所示。

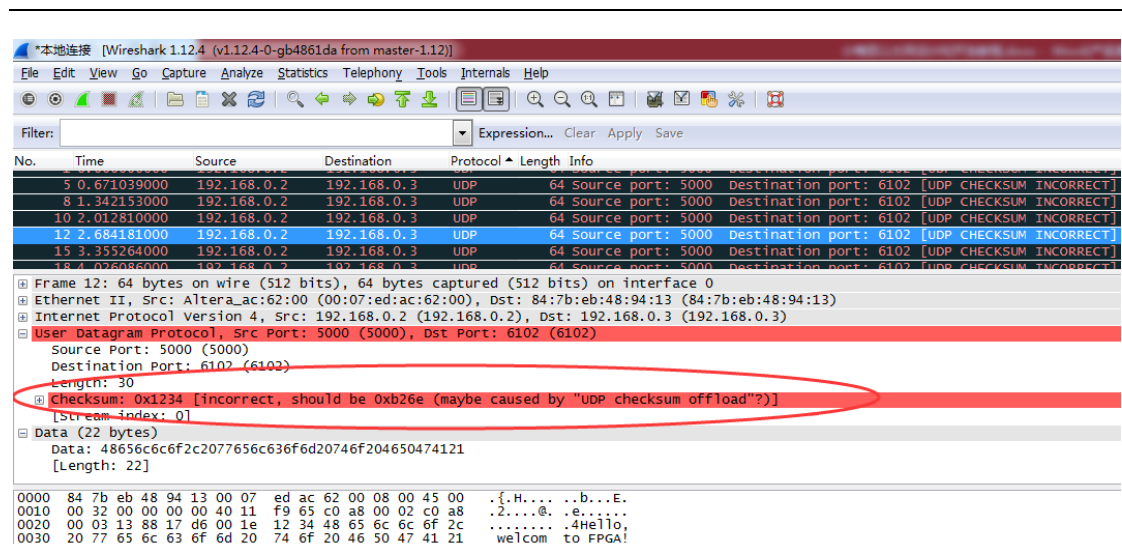


图 0-29

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。

总结

本节实验继续使用小兵以太网测试仪生成现成的 UDP 完整数据报，并采用查找表的方式发送。同时还验证了 UDP 报头中 UDP 校验和字段为正确、忽略、错误三种情况下的传输情况。可知 UDP 报头中的校验和字段即使忽略（设为全 0），也不会影响 UDP 数据报的传输。

由于 IP 校验和字段目前还是使用以太网测试工具生成的，并非自动生成，因此一旦修改 IP 报头中的任何一个数据，例如 IP 数据报长度，IP 校验和字段就无法校验通过，在以太网助手就无法正常接收到数据。所以本节实验结果依旧不具备实用意义。要具备实用意义，需要把 IP 报头校验和字段也像 CRC 字段一样，由程序实时自动生成。在 IP 协议原理与 IP 校验和算法小节中，实际我们已经实现了 IP 报头校验和的计算逻辑，后续我们只需要将其加进本节实验内容中，即可实现自动计算。

以太网 UDP 帧发包实例（自动 IPchecksum）

工程源码	--盘 A_AC620 开发板标准配套资料 ----02_设计实例 -----AC620_Verilog_Example -----以太网 -----udp_send_mii_with_checksum.rar
相关视频课程	--小梅哥 FPGA 入门到实战特训班课程 ----0808_02 以太网基本概念与 MAC 层协议介绍.mp4 ----0808_03 MAC 层数据传输接口 MII_GMII_RGMII.mp4 ----0809_01 以太网 MAC 层板级调试方法.mp4 ----0809_02 以太网 IP 层协议详解.mp4 ----0810_02 UDP 协议详解.mp4 ----0812_01 以太网传输项目总结.mp4

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。

加入自动 IP 报头校验逻辑

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。

实现该功能非常简单，只需要将 checksum 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。

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;
```

发送可变内容长度的数据

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的数据发送序列。即使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
```

图 0-30

```
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;
```

图 0-31

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;//1
61 : fifo_rddata = 4'h6;

62 : fifo_rddata = 4'hc;//1
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;//1
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。

功能验证

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:

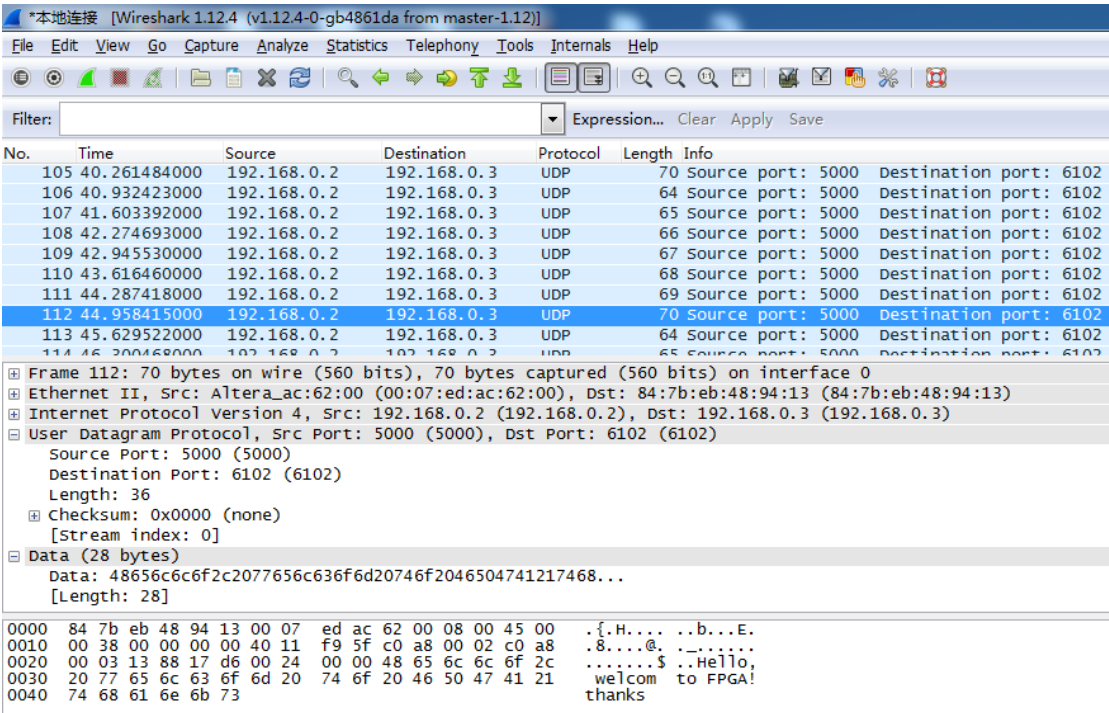


图 0-32

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, 按图设置好各项参数, 然后点击一次发送按钮, 即可接收到数据, 接收到的数据内容如下所示:

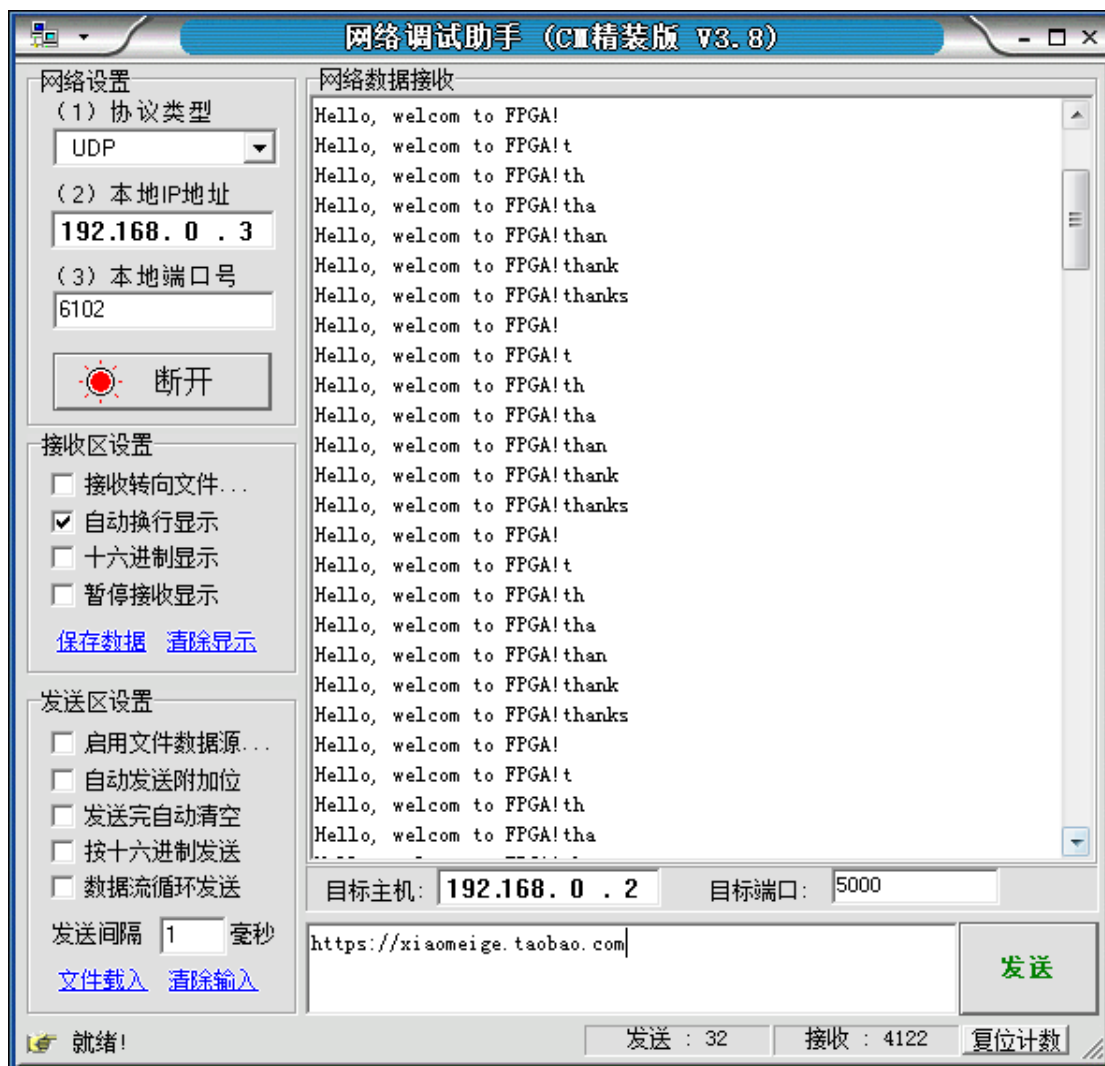


图 0-33

至此，基于 FPGA 的以太网 UDP 数据发送协议就设计完成了。

总结

虽然本设计完成了以太网的 UDP 发送逻辑，但是该设计实际还是没有实际意义，因为发送的数据内容都是预先写好的，是确定值。而我们实际在应用以太网发送数据时，不可能发这些固定的数据，更多的发的是实时在变，且无法预先预知值的内容，例如摄像头采集的图像数据，ADC 采样的数据。这种情况下，需要发送的应该是从某个数据源或者存储器直接取到的数据。所以要想本发送逻辑拥有实用价值，我们应该为其设计一个 FIFO 接口，让数据源产生的数据直接写入该 FIFO，而本发送逻辑则将数据部分修改为从 FIFO 中取数，这样就具备了相

当的实用价值。下一节，我们将实现该实用性的 UDP 发送逻辑。

实用性 UDP 收发逻辑设计与实现

通用 FIFO 接口 UDP 发送逻辑

高性能 UDP 发送逻辑设计

在上一节的内容中，我们从以太网 MAC 帧结构开始，图文并茂的介绍了以太网 MAC 帧、IP 协议、UDP 协议的详细结构，并针对每个协议都提供了对应的实验内容，带领大家一步一步的掌握了各个协议中的每个细节。相信进过前一节内容的认真学习，读者对于每个协议的每个字段的内容都有了非常清晰的认识。随着实验的进行，我们编写了 MAC 层、IP 层和 UDP 层的逻辑并最终实现了带 FIFO 的通用 UDP 发送接口。只是，相信大家也都发现了，由于我们采用的是分步分层设计的方法，尤其是在 MAC 与 UDP 层的交互数据时，大量使用了组合逻辑的多路器的方式来在不同的时刻输出不同的数值，这样的写法虽然实现功能没问题，但是从时序性能的角度分析，必然不是一个优秀的设计。在 FPGA 中布线资源充足的情况下，也许能够满足时序要求，但是一旦系统中还存在其他逻辑，导致留给以太网的部分布局布线资源紧张的情况下，就有可能出现时序不满足的情况，所以设计一个性能优异的 UDP 发送逻辑是很有必要的。

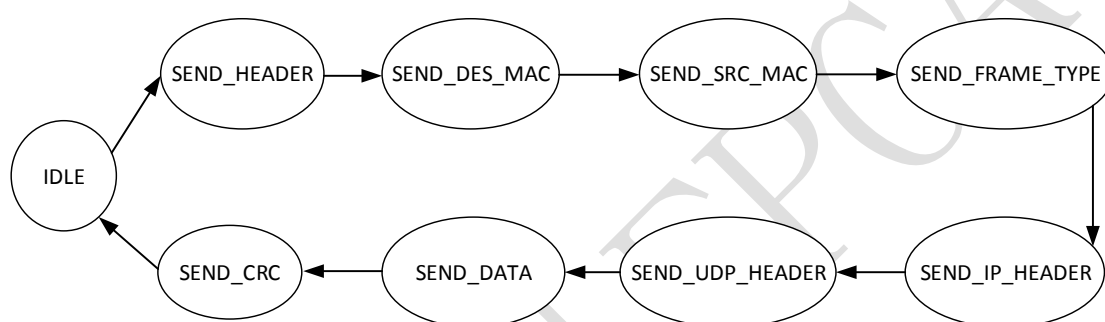
提升该设计的性能有两种途径，一是将其中的组合逻辑多路器部分改为时序逻辑。这种方案理论来说可行，但是在实际调试时，各层之间的数据交互要协调好有一定的困难，往往最终要通过凑时序的方式才能实现。而另一种方案则是将三层协议当做一个整体来对待，依次编码，这样就不存在多层之间的数据交互问题。

本节，将不再延续“以太网通信协议解析与实现”章节的分层介绍，分层实现的思路，而是直接将 MAC 层、IP 层、UDP 层看做一个整体，当做同一层来实现，通俗一点讲，就是将 MAC 帧头、IP 报头、UDP 报头等看成一个整体，按照

顺序依次发送这些报头后，再发送用户数据，这样就能够非常直观的构建以太网帧了。

发送状态机

以太网 UDP 帧的格式在前面已经做了讲解，并且帧内各种校验字段的产生也做了实现，接下来是对整个以太网 UDP 帧的发送进行实现。UDP 帧根据格式包括前导码+帧界定符、以太网头部数据、IP 头部数据、UDP 头部数据、UDP 数据、FCS 数据。根据格式对 UDP 帧发送的设计的状态机转移图如下。



通过状态图可以看到，整个的发送过程就是一个顺序执行的过程，不存在状态的分支，以下为每个状态的功能说明。

- IDLE：空闲状态
- SEND_HEADER：发送以太网的前导码和 1 字节的帧首定界符状态
- SEND_DES_MAC：发送 MAC 帧的目的网卡地址字段
- SEND_SRC_MAC：发送 MAC 帧的本机网卡地址字段
- SEND_FRAME_TYPE：发送 MAC 帧的长度类型字段
- SEND_IP_HEADER：发送 IP 报文报头部数据字段
- SEND_UDP_HEADER：发送 UDP 报文报头部数据字段
- SEND_DATA：发送 UDP 数据字段
- SEND_CRC：发送 FCS 校验数据字段

每一个状态中，实际又分成了很多的小的状态，以发送 MAC 帧的目的网卡地址字段（SEND_DES_MAC）为例，该状态中需要发送 48 位，也就是 6 个字节的数据，这对于 RGMII/GMII 接口来说，需要在该状态执行 6 个时钟周期，每个周期发送目的 MAC 值中的 8 位数据；而对于 MII 接口来说，由于每个时钟周期只能发送 4 位数据，所以需要在该状态中执行 12 个时钟周期。对于其他状态也是一样，该状态需要发送 N 个字节的数据，就需要在该状态执行 N（RGMII/GMII）或 2N（MII）个时钟周期。由于在一个确定状态中的不同时刻，需要发送的数据内容也是不一样的，所以每个状态中需要使用序列机的思想来决定在不同的时刻发送不同是数据。

协议与序列编码

根据前面内容学习的关于 MAC 层、IP 层、UDP 层的帧结构，我们甚至可以用相同的方法，通过一张表的形式来确定每个状态的每个时刻应该发送哪个数值。下表为笔者整理 RGMII/GMII 发送接口实现时每个状态的每个时刻应该发送的数据值。

状态	状态中时刻	发送值
SEND_HEADER	0~6	0x55
	7	0xd5
SEND_DES_MAC	0	des_mac[47:40]
	1	des_mac[39:32]
	2	des_mac[31:24]
	3	des_mac[23:16]
	4	des_mac[15:8]
	5	des_mac[7:0]
SEND_SRC_MAC	0	src_mac[47:40]
	1	src_mac[39:32]
	2	src_mac[31:24]
	3	src_mac[23:16]
	4	src_mac[15:8]
	5	src_mac[7:0]
SEND_FRAME_TYPE	0	frame_type[15:8]
	1	frame_type[7:0]
SEND_IP_HEADER	0	ip_header[0][31:24]
	1	ip_header[0][23:16]
	2	ip_header[0][15:8]

	3	ip_header[0][7:0]
	4	ip_header[1][31:24]
	5	ip_header[1][23:16]
	6	ip_header[1][15:8]
	7	ip_header[1][7:0]
	8	ip_header[2][31:24]
	9	ip_header[2][23:16]
	10	ip_header[2][15:8]
	11	ip_header[2][7:0]
	12	ip_header[3][31:24]
	13	ip_header[3][23:16]
	14	ip_header[3][15:8]
	15	ip_header[3][7:0]
	16	ip_header[4][31:24]
	17	ip_header[4][23:16]
	18	ip_header[4][15:8]
	19	ip_header[4][7:0]
SEND_UDP_HEADER	0	udp_header[0][31:24]
	1	udp_header[0][23:16]
	2	udp_header[0][15:8]
	3	udp_header[0][7:0]
	4	udp_header[1][31:24]
	5	udp_header[1][23:16]
	6	udp_header[1][15:8]
	7	udp_header[1][7:0]
SEND_DATA	-	fifo_rddata[7:0]
SEND_CRC	0	crc_result[31:24]
	1	crc_result[23:16]
	2	crc_result[15:8]
	3	crc_result[7:0]

发送逻辑端口设计

有了上述列表，实现时只需要先给定各个状态中需要发送的值，然后启动发送，整个状态机就会按照顺序依次发送每一个字段的内容。各个状态需要发送的值，例如 MAC 地址，IP 报头、UDP 报头的值，既可以根据实际的特定应用场合设置为一个固定值，也可以通过端口的方式，由其他模块提供。事实上，由端口的形式输入更加灵活，当需要使用固定值时，在例化时直接填入固定值即可，当需要由其他模块传递时，则连接到其他模块的对应信号即可。对于这参数，也不是每一个值都需要通过端口输入，只需输入真正需要的部分，例如对于 IP 报头

中的两个 IP 地址和长度是需要的，而其他各个字段则本身是固定值或者可以设置为固定值的，就不需要再通过端口输入。以下为笔者设计的高性能 UDP 发送模块的端口描述。

端口名和属性	端口功能介绍
input Rst_n	复位信号
input Go	启动发送信号，当该信号高脉冲出现时，启动一次完整的数据帧发送
output reg Tx_Done	发送完成标志信号，当该信号出现一个时钟周期的高脉冲时标志着一次数据帧发送完成
input [47:0]des_mac	数据帧接收方的网卡地址
input [47:0]src_mac	数据帧发送方，也就是本机（这里指 FPGA 板卡）的网卡地址
input [15:0]des_port	UDP 协议中接收方接收数据使用的端口号
input [15:0]src_port	UDP 协议中发送方（本机，FPGA 板卡）发送数据使用的端口号
input [31:0]des_ip	IP 协议中接收方的 IP 地址
input [31:0]src_ip	IP 协议中发送方（本机，FPGA 板卡）的 IP 地址
input [15:0]data_length	发送的数据长度，IP 报头和 UDP 报头中的数据长度值都是由该值计算得到
input GMII_GTXC	GMII 接口的发送时钟，也是整个发送逻辑的工作时钟
output reg [7:0]GMII_TXD	GMII 接口的发送数据
output reg GMII_TXEN	GMII 接口的数据发送使能信号
input wrreq	以太网数据发送缓冲 FIFO 写请求信号
input [7:0]wrdata	以太网数据发送缓冲 FIFO 写数据内容
input wrclk	以太网数据发送缓冲 FIFO 写入侧时钟信号
input aclr	以太网数据发送缓冲 FIFO 清零信号。
output [12:0]wrusedw	以太网数据发送缓冲 FIFO 写空间已使用值

用户数据输入

为了让该发送逻辑有更加实际的应用价值，设计时在该逻辑中加入了一个发送 FIFO，发送 FIFO 中存储需要发送的数据内容，在以太网帧发送过程中由发送逻辑在对应的时刻依次读取指定长度的值并发送。所以，使用该发送逻辑发送用户数据时，只需要将需要发送的数据写入 FIFO，给出需要发送的数据的长度值（data_length），然后再产生一个发送启动信号（Go）即可。

性能优化

设计该逻辑的一个最大原则就是让整个逻辑的时序性能尽可能的优异，真正

的 FPGA 设计师都明白优异的时序是设计出来的，而不是靠约束出来的。所以要想得到优异的时序性能，必须在设计时就考虑好很多细节，并按照最优的设计方式进行。本节将重点介绍笔者在设计高性能的 UDP 发送逻辑时采用了哪些能够确实提升设计的逻辑的时序性能的方法。

序列计数器优化

设计时，对于整个发送逻辑中的每个状态所需要的计数，可以使用同一个计数器，在每次进入该状态之前清零，然后计数到最大值后让状态跳转到写一个状态，但是这种写法，由于每个状态都有对其执行技术，限制计数最大值，以及清零计数值的操作，会导致该计数器的计数和清零逻辑变的异常复杂，降低整个设计的时序性能。

另外也有读者表示可以使用一个计数器从头计到尾的方式，既中间不加清零逻辑，和之前分层设计以太网协议逻辑时候的方法一样。这样的方式，确实可以降低计数器计数本身的控制逻辑，但是这样会导致整计数器的计数值比较多，需要判断的值也比较多，反而又会影响整个系统的运行频率。

所以，为了尽可能的提升系统的时序性能，设计中可以对每个状态使用一个独立的计数器进行计数，这样每个状态对应的计数器的计数和清零条件就变得固定且简单了，能够有效提升设计的时序性能。

设计中，对于每个状态，都提供了一个对应的计数器来控制该状态中每个时刻应该发送的数据内容，下表为每个状态和其对应的计数器的名称以及计数最大值关系。

状态	序列计数器名称	功能描述	计数最大值
SEND_HEADER	cnt_header	前导码计数器	7
SEND_DES_MAC	cnt_des_mac	目标 MAC 地址计数器	5
SEND_SRC_MAC	cnt_src_mac	源 MAC 地址计数器	5
SEND_FRAME_TYPE	cnt_frame_type	帧类型计数器	1
SEND_IP_HEADER	cnt_ip_header	IP 报头计数器	19
SEND_UDP_HEADER	cnt_udp_header	UDP 报头计数器	7
SEND_DATA	cnt_data	用户数据计数器	data_length_reg - 1
SEND_CRC	cnt_crc	crc 计数器	3

当然了，上述分法，其实也存在继续优化的情况，比如均匀的控制每个计数器的技术最大值，保证每个计数器的计数最大值不超过 7，即使得计数器的位宽不超过 3 位，加上其他可能存在的控制逻辑，能够保证该计数器的清零判断逻辑尽可能的在一个 LUT 中完成，从而进一步降低组合逻辑链路延迟。当然了，这些建议只是方向，具体的实现情况还需要根据 EDA 软件最后的布局布线情况来确定。

输入输出端口优化

由于整个设计存在多个输入和输出端口，这些端口的值随时都有可能发生变化，为了提升逻辑的时序性能，对于输入和输出端口，均采用先寄存再使用的设计方法，能够有效缩短输入端口到逻辑电路和逻辑电路到输出端口的数据链路长度，对于 EDA 软件对设计进行布局布线是非常有利的。所以可以对输入端口的数据，输入后先使用寄存器寄存一次之后，内部的发送逻辑使用时仅使用寄存之后的数据。对于输出端口，先将需要输出的信号通过寄存器寄存一拍之后再输出。下图 1 为不添加输入输出寄存器时的可能的逻辑电路布局情况，下图 2 为添加了输入输出寄存器后可能的逻辑电路布局情况。

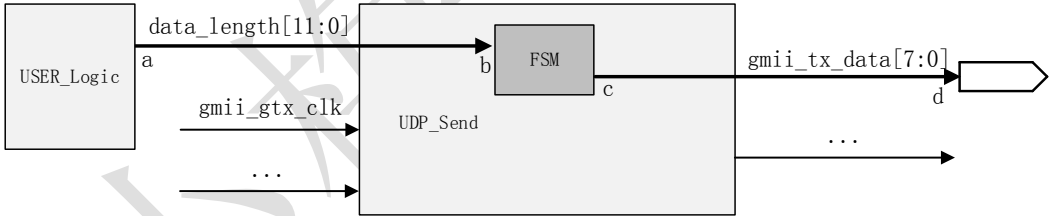


图 1

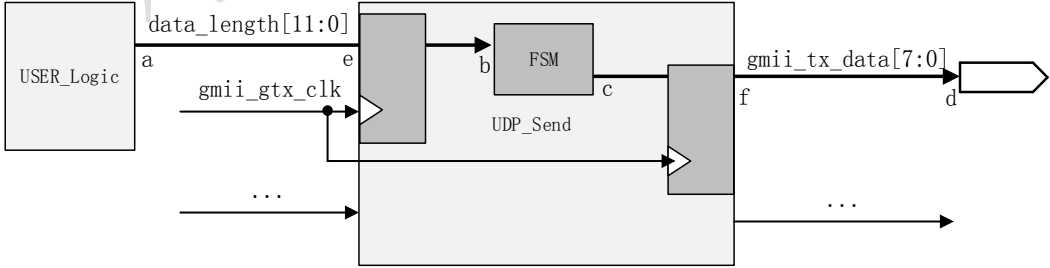


图 2

在图中，a 点到 b 点为发送逻辑的输入端口，以 data_length 这个端口为例；

c 点到 d 点为发送逻辑的输出端口，以 gmii_tx_data 端口为例。图 1 中，用户逻辑产生的 data_length 信号直接输入到发送逻辑内部的状态机中作为状态机的跳转条件。而状态机输出的 gmii_tx_data 信号也直接输出到 FPGA 的物理管脚。这样的实现方式存在两个问题：

- 1、对于输入端口，由于输入端口的数据是由其他逻辑产生，而其他逻辑本身有多复杂，在布局布线资源已经十分紧缺的 FPGA 系统中，这个其他逻辑到 UDP 发送逻辑的布线距离有多远？如果太远，则有可能会引入相当大的数据传输延迟。在 EDA 软件开了优化的情况下，为了降低这个延迟，EDA 软件可能会将其他逻辑与 UDP 发送逻辑拆散打乱后混合布局布线，从而对整体系统时序性能带来不可预料的影响。
- 2、对于输出端口，如果由状态机产生的信号直接输出到 FPGA 管脚，这之间的信号传输路径就比较长了，而且受 EDA 软件布局布线的影响，这个信号即使是由寄存器输出的，也有可能无法将该寄存器安置到 FPGA 管脚的快速输出寄存器上，从而降低系统时序性能。

而在输入和输出端口上插入寄存器先寄存一拍，则可以较好的解决这些问题。对于输入端口，该寄存器相当于在两个独立的逻辑块之间架起了一座桥梁，两个模块在布局布线时以该寄存器作为分界，能够有效缩短传输路径，也能够避免为了优化该路径传输延迟而对两个逻辑块混合布局布线的影响。对于输出端口，加入该寄存器后，该寄存器首先能够将发送逻辑和 FPGA 管脚的这条路径切分为 2 条较短路径，降低传输延迟，而且由于该寄存器没有其他复杂的功能，可以很容易的安置到 FPGA 管脚的快速输出寄存器，从而进一步提升系统时序性能。

以下为对输入端口进行寄存的逻辑代码，需要注意的是，这些端口上的数据在一次以太网帧发送的过程中必须保持稳定，避免在发送过程中这些参数发生变化导致发送失败。所以这些端口仅在发送使能信号 (Go) 出现的时候更新一次，其他时刻全部保持自身值不变。

```
//将所有输入信号存储进入寄存器
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

```

以下为对输出端口进行寄存的逻辑代码，对于输出端口，加上寄存器的目的是为了使其能够更加方便的放置到 FPGA 管脚上的快速输出寄存器，以使数据的输出延迟最小，也能保证多位宽数据的各个信号输出到 FPGA 物理 I/O 管脚上能够尽量的对齐。

内部信号寄存器

对于 IP、UDP 报头数据，在组织时，也尽量采用寄存器的方式存储数据。对于 IP 报头数据，设计时直接建立了一个深度为 5，位宽为 32 的二维寄存器，将对应的字段直接存入该寄存器中对应的位即可。对于 UDP 报头，也是按照相同的思路，建立了深度为 2，位宽为 32 的二维寄存器来存储报头中的各个字段，如下所示。

```

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

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
```

UDP 发送逻辑调试验证要点

至此，关于高性能 UDP 发送逻辑的设计要点就介绍完毕了。在经过了前面对以太网 MAC 帧、IP 报文、UDP 报文的详细解析和实现实验后，对于本系统的设计，就重点在于通过仿真的方式查看仿真结果，以验证该设计是否能够正常的发送希望发送的数据内容了。

验证时，通过仿真和 wireshark 软件抓包的方式配合调试，并最终要网络调试住手能够完整的收到需要发送的数据才能算作设计成功。

如果 wireshark 无法接收到 FPGA 发出的任何数据报，先检查设计中的目的 MAC 地址是否与实验用的主机的有线网卡的 MAC 地址一致，如果不确定，可以更换目的 MAC 地址为广播地址（FF-FF-FF-FF-FF-FF）进行测试。

同时，检查本地连接的速率确定为 1Gbps，100Mbps 的速率是无法正确的传输 RGMII/GMII 接口上提供的数据内容的。

如果上述情况都正确，证明以太网帧的 MAC 字段有问题，最大的可能就是 CRC 字段错误，此时可以使用 CRC 计算小软件计算所有数据的 CRC 校验值，然后与设计中计算出的 CRC 值进行对比，以确定 CRC 计算部分计算结果是否正确。CRC 计算结果不正确的可能主要有两个，一是计算逻辑本身设计有问题，二是参与 CRC 计算的数据内容不对，例如以太网帧的前导码和分隔符不参与计算，在发送 CRC 字段的时候，CRC 计算的过程应该停止，不再继续计算。

如果 wireshark 能够接收到以太网帧，但是不能正确的识别 IP 报文，检查以太网帧中的“长度/类型”字段值是否是 IP 协议对应的 0x0800。IP 报头校验和的错误不会影响 wireshark 识别 IP 报文，只会用深色背景标记该帧，并提示该帧中的 IP 校验和字段值错误。

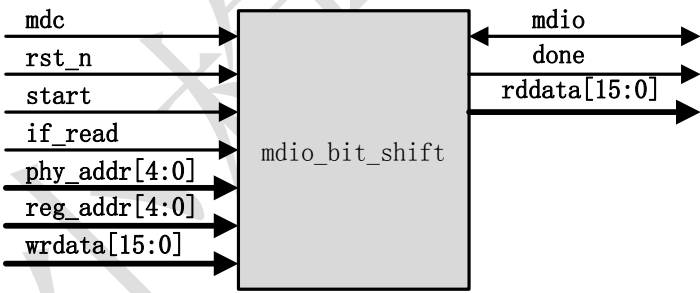
如果 wireshark 能够收到 UDP 帧，但是网络调试软件无法收到数据内容，检查 PC 的有线网卡 IP 地址是否设置了与 FPGA 设计中的目的 IP 地址一致。同时检查网络调试软件的端口号与设计中的端口号是否一致，还要确保该端口号没有被其他应用占用，可以修改端口号来测试。

MDIO 控制器设计

1.1 管理接口数据帧格式

1.4 mdio_bit_shift 模块设计

根据协议帧的格式和上面的读写时序图，我们发现这个协议帧里面的变化内容有 OP、PYHAD、REGAD、以及传输的数据（DATA）等字段，在这里我们可以把这些作为端口，同时加入传输开始 start 和传输完成 done 等握手信号来指示当前的模块工作状态，因此设计出如下模块单元：



9、 mdio_bit_shift 模块接口功能描述

接口名称	I/O	功能描述
mdc	I	时钟接口
rst_n	I	模块复位，低电平有效
start	I	开始传输标志
if_read	I	读写方向控制 1:读，0:写
phy_addr	I	5 位的 phy 地址输入信号，最高 2 位为 0
reg_addr	I	5 位的 reg 地址输入信号
wrddata	I	要写入 phy 寄存器的 16 位数据
mdio	I/O	数据接口

done	0	操作完成标志
rddata	0	从 phy 寄存器读出的 16 位数据

整个流程用状态机，按照读写时序图来描述，对于上面的每个字段都单独定义了一个状态，这里采用独热码来定义状态，定义如下：

```
localparam IDLE = 8'h01,
           PRE  = 8'h02,
           ST   = 8'h04,
           OP   = 8'h08,
           PHYAD = 8'h10,
           REGAD = 8'h20,
           TA   = 8'h40,
           DATA = 8'h80;
```

在对应的状态里面，按照时序图对差异部分进行对应的描述，从而实现对 phy 的寄存器进行配置和读取，状态机部分代码如下：

```
case (state)
  IDLE:
    begin
      mdio_o <= 1'b1;
      mdio_oe <= 1'b0;
      done <= 1'b0;
      rddata <= 'd0;
      if (start_pos)
        begin
          cnt <= 'd0;
          mdio_oe <= 1'b1;
          state <= PRE;
        end
      end
    PRE://PRE 32'hffff_ffff 32bit
    begin
      mdio_o <= 1'b1;
      cnt <= cnt + 1'b1;
      if (cnt > 'd30)
        begin
          cnt <= 'd0;
```

```

        state <= ST;
    end
end
ST: //ST 01 2bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd1)
    begin
        cnt <= 'd0;
        state <= OP;
    end
    case (cnt)
        0 : mdio_o <= 1'b0;
        1 : mdio_o <= 1'b1;
    endcase
end
OP: //OP 01:write,10:read 2bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd1)
    begin
        cnt <= 'd0;
        state <= PHYAD;
    end
    case (cnt)
        0 : mdio_o <= if_read;
        1 : mdio_o <= !if_read;
    endcase
end
PHYAD: //PHYAD 5bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd4)
    begin
        cnt <= 'd0;
        state <= REGAD;
    end
    mdio_o <= phy_addr[4 - cnt[2:0]];
end
REGAD: //REGAD 5bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd4)
    begin

```

```

        cnt <= 'd0;
        state <= TA;
    end
    mdio_o <= reg_addr[4 - cnt[2:0]];
end
TA:    //TA    2bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd1)
    begin
        cnt <= 'd0;
        state <= DATA;
    end
    case (cnt)
        0 : begin
            mdio_o <= !if_read;
            mdio_oe <= !if_read;
        end
        1 : mdio_o <= 1'b0;
    endcase
end
DATA:  //DATA    16bit
begin
    cnt <= cnt + 1'b1;
    if (cnt >= 'd15)
    begin
        cnt <= 'd0;
        state <= IDLE;
        mdio_o <= 1'b1;
        mdio_oe <= 1'b0;
        done <= 1'b1;
    end
    if (if_read)
        rddata <= {rddata[14:0], mdio};
    else
        mdio_o <= wrdata[15 - cnt[3:0]];
    end
endcase

```

当然按照芯片手册的描述，这里的 mdio 接口是采用外部上拉电阻进行上拉处理，所以我们代码中对 mdio 接口操作就不能直接的赋值操作，要通过三态门的特性，高电平让外部上拉电阻进行上拉来达到效果，代码部分如下：

```
reg mdio_o;  
  
reg mdio_oe;  
assign mdio = mdio_oe ? mdio_o : 1'bz;
```

1.5 mdio_bit_shift 模块仿真验证

上面通过对管理数据接口以及状态描述了这么多，那么我们设计的这个模块是否达到我们要的效果呢，接下来我们来仿真一下：

```
`timescale 1ns/1ps  
  
module mdio_bit_shift_tb;  
  
    reg          mdc;          //时钟接口  
    wire         mdio;         //数据接口  
    reg          rst_n;        //模块复位，低电平有效  
    reg          if_read;      //读写方向控制 1:读，0:写  
    reg [4:0]    phy_addr;     //phy 地址  
    reg [4:0]    reg_addr;     //reg 地址  
    reg [15:0]   wrdata;       //写入 phy 寄存器的数据  
    reg          start;        //开始传输标志  
    wire         done;         //操作完成标志  
    wire [15:0]  rddata;       //phy 寄存器读出的数据  
  
    pullup PUP(mdio); //模拟外部上拉电阻  
    mdio_bit_shift DUT(  
        .mdc      (mdc      ),  
        .mdio     (mdio     ),  
        .rst_n    (rst_n    ),  
        .if_read  (if_read  ),  
        .phy_addr (phy_addr ),  
        .reg_addr (reg_addr ),  
        .wrdata   (wrdata   ),  
        .start    (start    ),  
        .done     (done     ),  
        .rddata   (rddata   )  
    );  
    always #10 mdc = ~mdc;  
    initial begin  
        mdc = 1;  
        rst_n = 0;  
        if_read = 0;
```

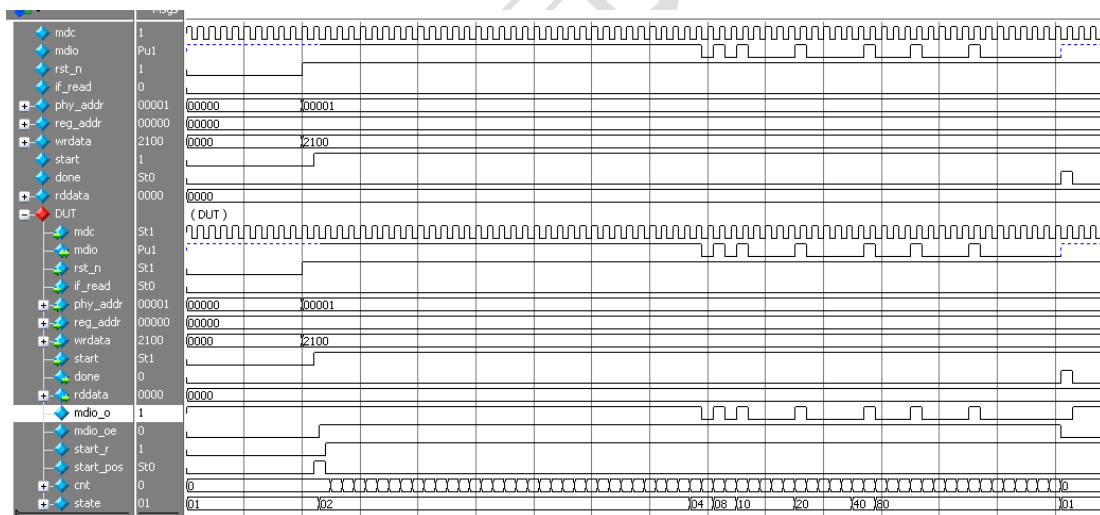
```

    phy_addr = 5'b00000;
    reg_addr = 5'b00000;
    wrdata = 16'd0;
    start = 0;
    #201;

    rst_n = 1;
    if_read = 0;
    phy_addr = 5'b00001;
    reg_addr = 5'b00000;
    wrdata = 16'h2100;
    @(posedge mdc);
    start = 1;
    @(posedge done);
    #200;
    $stop;
end
endmodule

```

在上面代码中，这里我们是对物理地址为 5'b00001 的寄存器 0（BMCR）进行一次写操作，写入的 16 位数据位 16'h2100,仿真波形如下：

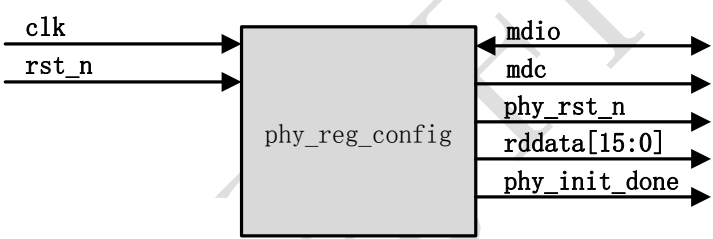


在上述波形中当 state 为 02（也就是 PRE 状态），mdio 发送了连续的 32 个逻辑 1，当 state 为 04（也就是 ST 状态），mdio 先后产生了 01，当 state 为 08（也就是 OP 状态），mdio 依次产生了 01 说明是写操作，当 state 为 10（也就是 PHYAD 状态），mdio 依次产生了 00001 表示操作的 phy 地址，当 state 为 20（也就是 REGAD 状态），mdio 依次产生了 00000 表示操作的 phy 的寄存器 0(BMCR)，

当 state 为 40（也就是 TA 状态），mdio 依次产生了 10，当 state 为 80（也就是 DATA 状态），mdio 依次产生了 0010_0001_0000_0000(16'h2100)表示写入寄存器 0 的值为 16'h2100，符合我们预期的结果。

1.6 phy_reg_config 模块设计

我们发现上面的 mdio_bit_shift 模块虽然能实现基本的 phy 寄存器的读写操作，但是是一次性了，不能连续的配置，那么此时我们就需要一个上层的模块来控制这个 mdio_bit_shift 模块，这样对于用户来说也方便一些。根据功能我们加入 phy 芯片寄存器配置完成标志信号 phy_init_done，从 phy 寄存器读出的数据 rddata，以及 phy 芯片的复位信号 phy_rst_n，因此设计出如下模块单元：



10、 phy_reg_config 模块接口功能描述

接口名称	I/O	功能描述
clk	I	模块时钟 50MHz
rst_n	I	模块复位，低电平有效
mdio	I/O	管理接口数据总线
mdc	O	管理接口时钟总线
phy_rst_n	O	phy 芯片复位，低电平有效
rddata	O	从 phy 寄存器读出的 16 位数据
phy_init_done	O	phy 寄存器初始化操作完成标志

还是那实际的例子来说明，开发板默认的是 GMII 模式，是千兆模式，那么我们现在要让它工作在百兆模式可以么，答案是肯定的，通过查询收据手册我们，寄存器 0（BMCR 寄存器）里面有对应的速率控制，前提是要将自动协商给关闭了，当然为了能正确的配置，我们首先要让芯片先重新上电。

芯片手册寄存器 0 表如下图，更多的寄存器表请查看芯片手册：

Table 24. BMCR (Basic Mode Control Register, Address 0x00)

Bit	Name	Type	Default	Description
0.15	Reset	RW, SC	0	Reset. 1: PHY reset 0: Normal operation Register 0 (BMCR) and register 1 (BMSR) will return to default values after a software reset (set Bit15 to 1). This action may change the internal PHY state and the state of the physical link associated with the PHY.
0.14	Loopback	RW	0	Loopback Mode. 1: Enable PCS loopback mode 0: Disable PCS loopback mode

Bit	Name	Type	Default	Description															
0.13	Speed[0]	RW	0	Speed Select Bit 0. In forced mode, i.e., when Auto-Negotiation is disabled, bits 6 and 13 determine device speed selection. <table><tr><th>Speed[1]</th><th>Speed[0]</th><th>Speed Enabled</th></tr><tr><td>1</td><td>1</td><td>Reserved</td></tr><tr><td>1</td><td>0</td><td>Reserved</td></tr><tr><td>0</td><td>1</td><td>100Mbps</td></tr><tr><td>0</td><td>0</td><td>10Mbps</td></tr></table>	Speed[1]	Speed[0]	Speed Enabled	1	1	Reserved	1	0	Reserved	0	1	100Mbps	0	0	10Mbps
Speed[1]	Speed[0]	Speed Enabled																	
1	1	Reserved																	
1	0	Reserved																	
0	1	100Mbps																	
0	0	10Mbps																	
0.12	ANE	RW	1	Auto-Negotiation Enable. 1: Enable Auto-Negotiation 0: Disable Auto-Negotiation															
0.11	PWD	RW	0	Power Down. 1: Power down (only Management Interface and logic are active; link is down) 0: Normal operation															
0.10	Isolate	RW	0	Isolate. 1: RGMII/GMII interface is isolated; the serial management interface (MDC, MDIO) is still active. When this bit is asserted, the RTL8211E-VB(VL)/RTL8211EG-VB ignores TXD[7:0], and TXCTL inputs, and presents a high impedance on TXC, RXC, RXCTL, RXD[7:0]. 0: Normal operation															
0.9	Restart_AN	RW, SC	0	Restart Auto-Negotiation. 1: Restart Auto-Negotiation 0: Normal operation															
0.8	Duplex	RW	1	Duplex Mode. 1: Full Duplex operation 0: Half Duplex operation This bit is valid only in force mode, i.e., NWay is disabled.															
0.7	Collision Test	RW	0	Collision Test. 1: Collision test enabled 0: Normal operation															
0.6	Speed[1]	RW	1	Speed Select Bit 1. Refer to bit 0.13.															
0.5:0	RSVD	RO	000000	Reserved.															

第一步，通过寄存器表我们可知道要让 phy 芯片先掉电，可以通过配置寄存器 0 的 bit11 位来实现，同时这个时候我们也顺便可以配置让芯片工作在双工模式下，通过配置寄存器 0 的 bit8 位来实现，通过上表我们可以得出此时这个寄存

器应该配置成 0000_1001_0000_0000 (16'h0900)。

第二步,通过寄存器表我们可知道要让 phy 芯片工作在百兆模式,那么可以通过配置寄存器 0 的 bit13 和 bit6 来实现,此时 bit13 里面的注释也说了,要先把自动协商禁用了,我们知道寄存器 0 的 bit12 就可以控制自动协商,此时我们把寄存器 0 的 bit12 置为 0 就可以了,自动协商禁用了,我们就可以配置速度了,要工作在百兆模式,那么 bit13 置为 1, bit6 置为 0,通过上表我们可以得出此时这个寄存器应该配置成 0010_0001_0000_0000 (16'h2100)。

知道了要配置的寄存器值,那我们就可以写个序列机下来一次初始化,代码如下:

```
//配置 phy 寄存器
reg [7:0] reg_cnt;
always @ (reg_cnt)
begin
    case (reg_cnt)
        0:reg_data <= {5'd0, 16'h0900}; //掉电

        //禁止网卡自动协商,让网卡工作在百兆模式
        1:reg_data <= {5'd0, 16'h2100};
        default:reg_data <= {5'd0, 16'h2100};
    endcase
end
```

当然上面配置的是两个寄存器,如果需要配置其他的寄存器,可以自行在序列机里面加上, reg_data 的高 5bit 是 phy 的寄存器地址,低 16 位是配置进去的值。

有了待配置的寄存器值,这个时候就需要一个模块来控制每一次的寄存器操作,状态控制部分代码如下:

```
//配置 PHY 寄存器状态控制
reg state;
parameter MAX_CNT = 2; //要配置的寄存器个数
always @ (posedge mdc)
begin
```

```

if (!Go) begin
    state <= 1'b0;
    start <= 1'b0;
    reg_cnt <= 'd0;
end
else if (reg_cnt < MAX_CNT) begin
    case (state)
        0:begin //开始写寄存器
            start <= 1'b1;
            state <= 1'b1;
            mdio_data <= reg_data;
        end
        1:begin //写寄存器完成
            if (done) begin
                start <= 1'b0;
                state <= 1'b0;
                reg_cnt <= reg_cnt + 1'b1;
            end
        end
    endcase
end
end
end

```

开始的时候将 start 拉高，开始操作 phy 寄存器，操作完成后 done 信号就会被拉高，此时就把 start 信号拉低跳转到写寄存器状态，这样一次寄存器操作就完成了，reg_cnt 在这里是控制上面序列机里面要操作的对应 phy 寄存器。

整个过程的工作都是依赖于 mdc 时钟的，这里的 mdc 时钟用的是 50KHz，产生 mdc 时钟部分代码如下：

```

//系统时钟采用 50MHz
parameter SYS_CLOCK = 50_000_000;
//MDC 总线时钟采用 50kHz
parameter MDC_CLOCK = 50_000;
//产生时钟 SCL 计数器最大值
localparam MDC_CNT_M = SYS_CLOCK/MDC_CLOCK/2 - 1;

reg [19:0] div_cnt;
always@(posedge clk or negedge rst_n)
if (!rst_n)
    div_cnt <= 'd0;

```

```

else if(div_cnt < MDC_CNT_M)
    div_cnt <= div_cnt + 1'b1;
else
    div_cnt <= 'd0;

wire mdc_pulse = div_cnt == MDC_CNT_M;

reg mdc_clk;
always@(posedge clk or negedge rst_n)
if (!rst_n)
    mdc_clk <= 1'b0;
else if(mdc_pulse)
    mdc_clk <= ~mdc_clk;

```

1.7 phy_reg_config 模块仿真验证

介绍完 phy_reg_config 模块后我们来仿真这个模块，看看能否按照我们的期望，把这两个寄存器的值都配置进去了，仿真代码如下：

```

`timescale 1ns/1ns

module mdio_tb;
    reg        clk;
    reg        rst_n;
    wire       eth_rst_n;
    wire       eth_mdc;
    wire       eth_mdio;
    wire [15:0] rddata;
    wire       init_done;

    pullup PUP(eth_mdio); //模拟外部上拉电阻

    phy_reg_config u_phy_config(
        .clk        (clk        ),
        .rst_n      (rst_n      ),
        .phy_rst_n  (eth_rst_n ),
        .mdc        (eth_mdc   ),
        .mdio       (eth_mdio  ),
        .rddata     (rddata    ),
        .phy_init_done(init_done )
    );

    always #10 clk = ~clk;
    initial begin

```

```

        clk = 1;
        rst_n = 0;
        #201;
        rst_n = 1;

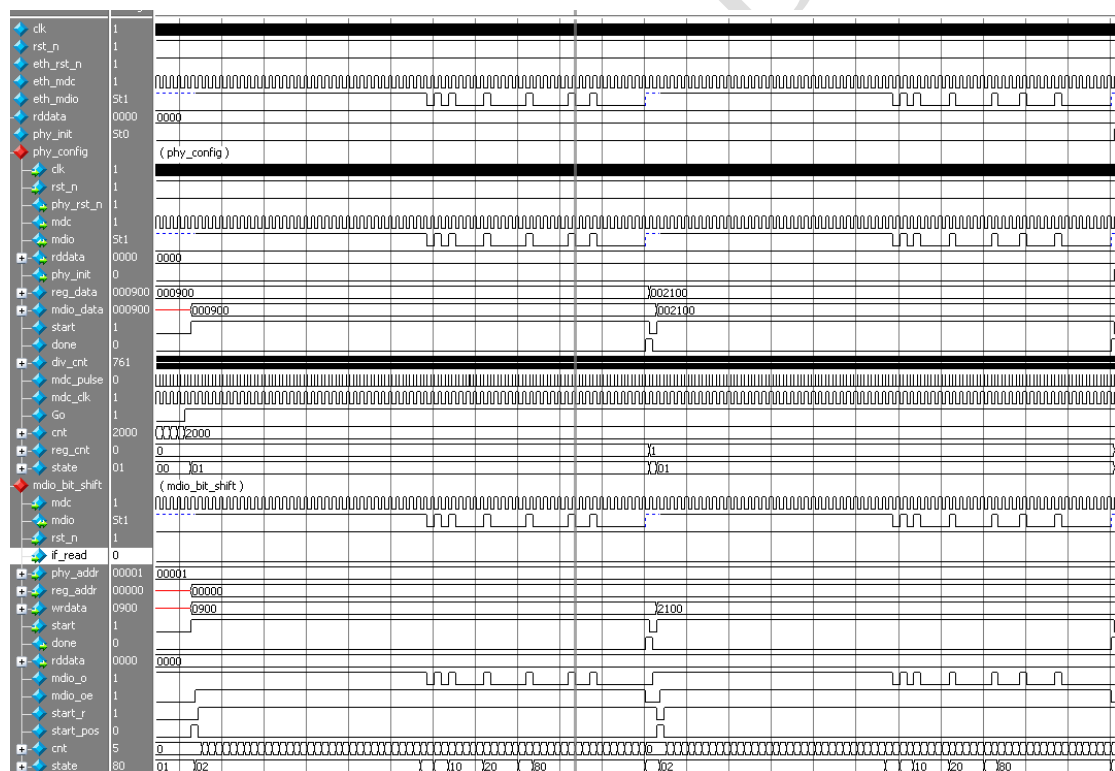
        @(posedge init_done);
        #2000;
        $stop;

    end

endmodule

```

在上面代码中，这里我们是对物理地址为 5'b00001 的寄存器 0（BMCR）进行两次写操作，首先写入的 16 位数据位 16'h0900，之后写入的 16 位数据位 16'h2100，仿真波形如下：



从上面的波形可以看得出来，start_pos 将操作分成了两次，第一次将 16'h0900 写入了 phy 的寄存器 0，第二次将 16'h2100 写入了 phy 的寄存器 0，每次完成后 done 信号都会被拉高，用来跟上层的 phy_reg_config 模块握手，两个寄存器都初始化完成后，phy_init 信号就被拉高指示寄存器初始化完成。

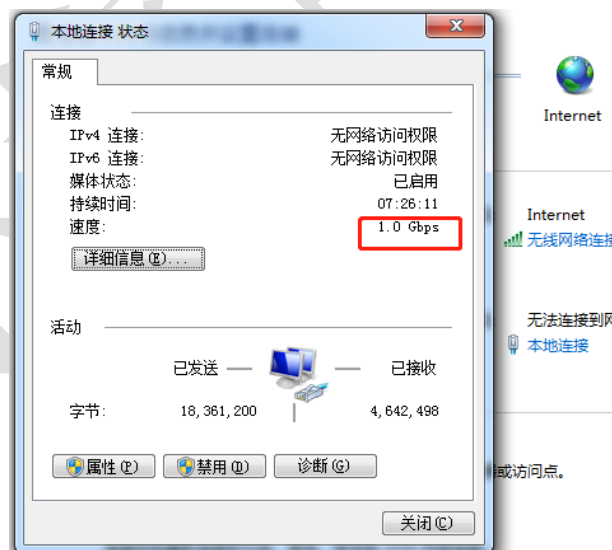
1.8 板级验证

上面的仿真没有问题，那么我现在就把这个绑定管脚全编译后生成配置文件，下载到 FPGA 里面看看是否真的有效。

没有将配置文件下载到 FPGA 之前我们来看一下网卡是工作在什么模式，打开网络共享中心，点击图中的红色框框里面的本地连接



点开后，如图所示显示的是 1.0Gbps，也就是千兆模式。



接下来我们将配置文件下载到 FPGA，之后再来看，如图所示显示的是 100Mbps，此时已经被成功配置成百兆模式了。



Internet

Internet
无线网络连接

无法连接到网
本地连接

或访问点。