58 实用性 UDP 接收逻辑设计与实现

工程源码	|----02_设计实例 |--------ch58_acz7015_eth_udp_rx_rgmii
相关视频课程	
说明	无

章节导读

上一章中，我们完成了高性能 UDP 发送模块的设计。而完整的 UDP 应用，除了 UDP 发送外，必然还包含 UDP 接收。故此，本章将带大家实现高性能 UDP 接收模块的设计。

58.1UDP 接收模块代码设计

以太网 udp 接收是以太网 udp 发送的逆过程，因此这里不再对 udp 发送原理讲解，用户可以参考前述章节。参考上一章节，我们可以设计出如图 58-1 所示接收模块：

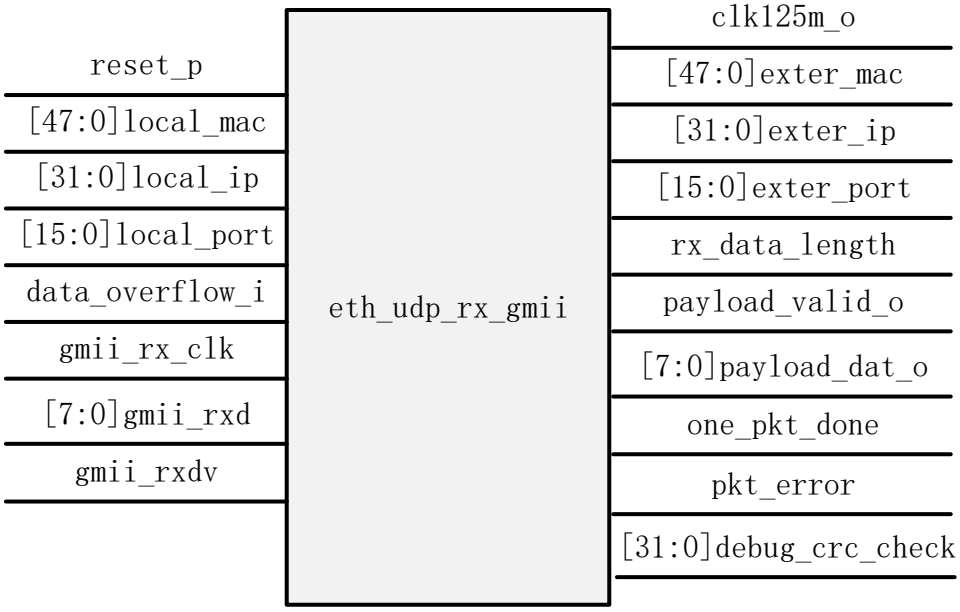

图 58-1 以太网接收模块的基本结构图

该模块的信号说明如下表 58-1 所示。

表 58-1 以太网接收模块信号说明表

接口名称	I/O	信号意义
reset_p	I	模块复位信号，高有效
local_mac[47:0]	I	本地 mac 地址

local_ip [31:0]	I	本地 ip 地址
local_port[15:0]	I	本地端口号
data_overflow_i	I	输入数据溢出标志信号
gmii_rx_clk	I	接收数据参考时钟，时钟频率为 125MHz。RX_CLK 是由 PHY 侧提供的
gmii_rxdv	I	即 Reveive Data Valid，接收数据有效信号，作用类似于发送通道的 TX_EN
gmii_rxd[7:0]	I	即 ReceiveData，数据接收信号，共 8 根信号线
exter_mac[47:0]	O	目的 mac 地址
exter_ip[31:0]	O	目的 IP 地址
exter_port[15:0]	O	目的端口号
rx_data_length[15:0]	O	接收数据长度信号
payload_valid_o	O	输出数据有效信号
payload_dat_o[7:0]	O	8 位数据输出信号
one_pkt_done	O	以太网包传输完成信号
pkt_error	O	接收数据错误标志信号
debug_crc_check	O	调试 CRC 检验信号

在前面我们对以太网 UDP 帧格式做了讲解，UDP 帧格式包括前导码+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，以太网接收模块同样是按照该格式接收数据。这里，我们主要通过状态机的方式实现以太网接收模块的功能，该模块的状态转移图如下图 58-2 所示。

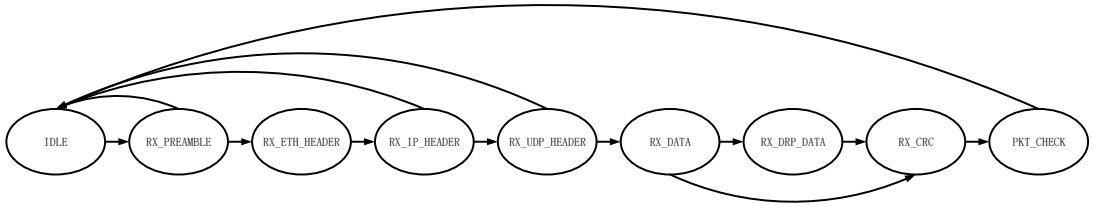

图 58-2 以太网接收模块状态转移图

下面将对各个状态的实现及功能进行简要介绍。

1. IDLE

空闲状态，当产生接收数据有效信号时，进入 RX_PREAMBLE 状态，否则处于 IDLE 状态，代码如下所示：

```
IDLE:
if(!rx_datav_dly2 && rx_datav_dly1)
    next_state = RX_PREAMBLE;
else
    next_state = IDLE;
```

上述代码中的 rx_datav_dly1 信号是将接收数据有效信号 gmii_rxdv 寄存之后打一拍得到的，rx_datav_dly2 信号是将 rx_datav_dly1 信号打一拍得到的，将 rx_datav_dly2 信号取反与 rx_datav_dly1 相与得到接收数据有效脉冲，得到该信号之后，进入到 RX_PREAMBLE 状态，rx_datav_dly1 信号和 rx_datav_dly2 信号

的实现代码如下所示，代码中对 gmii_rxd 信号也进行了寄存和打拍操作，该信号的使用将会在后面进行说明。

```
//将以太网输入的接收信号寄存
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
    reg_gmii_rxd  <= 8'h00;
    reg_gmii_rxdv <= 1'b0;
end
else
begin
    reg_gmii_rxd  <= gmii_rxd;
    reg_gmii_rxdv <= gmii_rxdv;
end

//将以太网输入的接收信号寄存后打拍
always@(posedge clk125m)
begin
    rx_data_dly1  <= reg_gmii_rxd;
    rx_data_dly2  <= rx_data_dly1;
    rx_datav_dly1 <= reg_gmii_rxdv;
    rx_datav_dly2 <= rx_datav_dly1;
end
```

2. RX_PREAMBLE 状态

处于 RX_PREAMBLE 状态的时候，当以太网接收到帧界定符（D5）和 5 个前导码（55）时，进入到 RX_ETH_HEADER 状态，如果接收超过 7 个前导码，则表明此时数据接收错误，进入 IDLE 状态，代码如下所示：

```
RX_PREAMBLE:
if(rx_data_dly2 == 8'hd5 && cnt_preamble > 4'd5)
    next_state = RX_ETH_HEADER;
else if(cnt_preamble > 4'd7)
    next_state = IDLE;
else
    next_state = RX_PREAMBLE;
```

上述代码中的 rx_data_dly2 信号就是将 gmii_rxd 信号寄存之后打两拍得到的，cnt_preamble 信号是用来计数的，也就是处于 RX_PREAMBLE 状态时，得到的前导码的数据个数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_preamble <= 4'd0;
else if(curr_state == RX_PREAMBLE && rx_data_dly2 == 8'h55)
    cnt_preamble <= cnt_preamble + 1'b1;
```

```
else
    cnt_preamble <= 4'd0;
```

3. RX_ETH_HEADER

处于 RX_ETH_HEADER 状态时，接收以太网头部数据，当接收完 14 个以太网头部数据之后，进入到 RX_IP_HEADER 状态，代码如下所示：

```
RX_ETH_HEADER:
if(cnt_eth_header == 4'd13)
    next_state = RX_IP_HEADER;
else
    next_state = RX_ETH_HEADER;
```

cnt_eth_header 信号在状态处于 RX_ETH_HEADER 时，进入计数，否则清零，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_eth_header <= 4'd0;
else if(curr_state == RX_ETH_HEADER)
    cnt_eth_header <= cnt_eth_header + 1'b1;
else
    cnt_eth_header <= 4'd0;
```

然后当处于该状态的时候，根据 cnt_eth_header 的值，依次得到 14 个字节的以太网头部数据，分别是 MAC 目的地址（6 个字节）、MAC 源地址（6 个字节）和以太网类型（2 个字节），代码如下所示：

```
//eth_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  rx_dst_mac  <= 48'h00_00_00_00_00_00;
  rx_src_mac  <= 48'h00_00_00_00_00_00;
  rx_eth_type <= 16'h0000;
end
else if(curr_state == RX_ETH_HEADER)
begin
  case(cnt_eth_header)
    4'd0 :rx_dst_mac[47:40] <= rx_data_dly2;
    4'd1 :rx_dst_mac[39:32] <= rx_data_dly2;
    4'd2 :rx_dst_mac[31:24] <= rx_data_dly2;
    4'd3 :rx_dst_mac[23:16] <= rx_data_dly2;
    4'd4 :rx_dst_mac[15:8]  <= rx_data_dly2;
    4'd5 :rx_dst_mac[7:0]   <= rx_data_dly2;

    4'd6 :rx_src_mac[47:40] <= rx_data_dly2;
    4'd7 :rx_src_mac[39:32] <= rx_data_dly2;
    4'd8 :rx_src_mac[31:24] <= rx_data_dly2;
```

```
    4'd9 :rx_src_mac[23:16] <= rx_data_dly2;
    4'd10:rx_src_mac[15:8]  <= rx_data_dly2;
    4'd11:rx_src_mac[7:0]   <= rx_data_dly2;

    4'd12:rx_eth_type[15:8] <= rx_data_dly2;
    4'd13:rx_eth_type[7:0]  <= rx_data_dly2;
    default: ;
  endcase
end
else
begin
  rx_dst_mac  <= rx_dst_mac;
  rx_src_mac  <= rx_src_mac;
  rx_eth_type <= rx_eth_type;
end
```

4. RX_IP_HEADER

接收以太网 IP 头部数据状态 RX_IP_HEADER，首先得对接收的以太网 IP 头部数据进行计数，定义一个计数器 cnt_ip_header，当处于该状态的时候进行计数，否则清零，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
    cnt_ip_header <= 5'd0;
else if(curr_state == RX_IP_HEADER)
    cnt_ip_header <= cnt_ip_header + 1'b1;
else
    cnt_ip_header <= 5'd0;
```

然后当处于 RX_IP_HEADER 状态时，获取以太网 IP 头部数据，根据 cnt_ip_header 的值，一共需要获取 20 个字节的数据，分别为 IP 版本（rx_ip_ver）、首部长度（rx_ip_hdr_len）、服务类型（rx_ip_tos）、数据报总长度（rx_total_len）、标识主机发送的每一份数据报（rx_ip_id）、标志位（rx_ip_rsv、rx_ip_df、rx_ip_mf）、段偏移量（rx_ip_frag_offset）、生存期（rx_ip_ttl）、IP 的协议封装类型（rx_ip_protocol）、头部校验和（rx_ip_check_sum）、源 IP 地址（rx_src_ip）和目的 IP 地址（rx_dst_ip），代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  {rx_ip_ver,rx_ip_hdr_len}     <= 8'h0;
  rx_ip_tos                     <= 8'h0;
  rx_total_len                  <= 16'h0;
  rx_ip_id                      <= 16'h0;
  {rx_ip_rsv,rx_ip_df,rx_ip_mf} <= 3'h0;
```

```
  rx_ip_frag_offset             <= 13'h0;
  rx_ip_ttl                     <= 8'h0;
  rx_ip_protocol                <= 8'h0;
  rx_ip_check_sum               <= 16'h0;
  rx_src_ip                     <= 32'h0;
  rx_dst_ip                     <= 32'h0;
end
else if(curr_state == RX_IP_HEADER)
begin
  case(cnt_ip_header)
    5'd0:{rx_ip_ver,rx_ip_hdr_len}          <= rx_data_dly2;
    5'd1:rx_ip_tos                          <= rx_data_dly2;
    5'd2:rx_total_len[15:8]                 <= rx_data_dly2;
    5'd3:rx_total_len[7:0]                  <= rx_data_dly2;
    5'd4:rx_ip_id[15:8]                     <= rx_data_dly2;
    5'd5:rx_ip_id[7:0]                      <= rx_data_dly2;
    5'd6:{rx_ip_rsv,rx_ip_df,rx_ip_mf,rx_ip_frag_offset[12:8]}<=rx_dat
a_dly2;
    5'd7:rx_ip_frag_offset[7:0]             <= rx_data_dly2;
    5'd8:rx_ip_ttl                          <= rx_data_dly2;
    5'd9:rx_ip_protocol                     <= rx_data_dly2;
    5'd10:rx_ip_check_sum[15:8]             <= rx_data_dly2;
    5'd11:rx_ip_check_sum[7:0]              <= rx_data_dly2;
    5'd12:rx_src_ip[31:24]                  <= rx_data_dly2;
    5'd13:rx_src_ip[23:16]                  <= rx_data_dly2;
    5'd14:rx_src_ip[15:8]                   <= rx_data_dly2;
    5'd15:rx_src_ip[7:0]                    <= rx_data_dly2;
    5'd16:rx_dst_ip[31:24]                  <= rx_data_dly2;
    5'd17:rx_dst_ip[23:16]                  <= rx_data_dly2;
    5'd18:rx_dst_ip[15:8]                   <= rx_data_dly2;
    5'd19:rx_dst_ip[7:0]                    <= rx_data_dly2;
    default: ;
  endcase
end
```

在 RX_ETH_HEADER 状态时，我们获取了以太网头部数据，进入本状态 RX_IP_HEADER 之后，需要判断之前获取的以太网头部数据是否正确，当获取的数据类型等于 ETH_type（0x0800），得到的 MAC 地址等于代码中设定的值 local_mac_reg 或者等于广播地址 FF_FF_FF_FF_FF_FF 时，则代表以太网头部数据接收成功将 eth_header_check_ok 信号拉高，否则为低，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  eth_header_check_ok <= 1'b0;
else if(rx_eth_type == ETH_type && (rx_dst_mac == local_mac_reg ||
rx_dst_mac == 48'hFF_FF_FF_FF_FF_FF))
  eth_header_check_ok <= 1'b1;
```

```
else
  eth_header_check_ok <= 1'b0;
```

在 RX_IP_HEADER 状态时，当 cnt_ip_header 计数等于 2 并且 eth_header_check_ok 为低时，则代表数据接收错误，进入 IDLE 状态，当 cnt_ip_header 计数到 19，也就是接收完 20 个 IP 头部书数据之后，进入 RX_UDP_HEADER 状态，代码如下所示：

```
RX_IP_HEADER:
  if(cnt_ip_header == 5'd2 && eth_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_ip_header == 5'd19)
    next_state = RX_UDP_HEADER;
  else
    next_state = RX_IP_HEADER;
```

5. RX_UDP_HEADER

接收 UDP 头部数据状态 RX_UDP_HEADER，进入该状态之后，首先设置一个计数信号 cnt_udp_header 用来计数得到的 UDP 头部数据的个数，代码如下所示：

```
//cnt_udp_header
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_udp_header <= 4'd0;
else if(curr_state == RX_UDP_HEADER)
  cnt_udp_header <= cnt_udp_header + 1'b1;
else
  cnt_udp_header <= 4'd0;
```

在 RX_UDP_HEADER 状态时，需要判断前一个状态获取的以太网 IP 头部数据是否正确，当获取的数据正确时，将 ip_header_check_ok 信号拉高，否则为低，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  ip_header_check_ok <= 1'b0;
else if({IP_ver,IP_hdr_len,IP_protocol,cal_check_sum,local_ip_reg} ==
        {rx_ip_ver,rx_ip_hdr_len,rx_ip_protocol,rx_ip_check_sum,rx_dst
_ip})
  ip_header_check_ok <= 1'b1;
else
  ip_header_check_ok <= 1'b0;
```

当 cnt_udp_header 计数到 2 并且 ip_header_check_ok 信号为 0 时，则代表接收数据出错，返回 IDLE 状态；当 cnt_udp_header 计数到 7 并且 udp_header_check_ok 信号为 0 时，也代表接收的数据是错误的，返回 IDLE 状

态；当 cnt_udp_header 计数到 7，则代表 UDP 头部数据接收完成，进入 RX_DATA 状态，一共接收了 8 个字节数据分别是源端口号（rx_src_port）、目的端口号（rx_dst_port）、接收的 16 位包括首部在内的 UDP 报文段长度（rx_udp_length）、16 位的 UDP 报头校验和。综上所述，得到 RX_UDP_HEADER 状态机的代码，如下所示：

```
RX_UDP_HEADER:
  if(cnt_udp_header == 4'd2 && ip_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_udp_header == 4'd7 && udp_header_check_ok == 1'b0)
    next_state = IDLE;
  else if(cnt_udp_header == 4'd7)
    next_state = RX_DATA;
  else
    next_state = RX_UDP_HEADER;
```

6. RX_DATA

接收数据状态 RX_DATA，进入该状态之后，对接收的数据个数进行计数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_data <= 16'd0;
else if(curr_state == RX_DATA)
  cnt_data <= cnt_data + 1'b1;
else
  cnt_data <= 16'd0;
```

以太网 UDP 的帧数据段个数=以太网接收的帧长度（rx_total_len）-20 个 IP 头部数据-8 个 UDP 头部数据，当处于 RX_IP_HEADER 状态时并且接收完 20 个以太网头部 IP 数据时，计算出以太网 UDP 的数据段个数 rx_data_length，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  rx_data_length <= 16'd0;
else if(curr_state == RX_IP_HEADER && cnt_ip_header == 5'd19)
  rx_data_length <= rx_total_len - 8'd20 - 8'd8;
else
  rx_data_length <= rx_data_length;
```

处于 RX_DATA 状态时，当接收的帧长度 rx_total_len 小于 46，也就是接收的数据段个数小于 18 时，说明此时接收的个数较少，进入到 RX_DRP_DATA 状态，当接收的数据个数等于以太网 UDP 的帧数据段个数时，进入到 RX_CRC 状态，否则处于 RX_DATA 状态，代码如下所示：

```
RX_DATA:
  if((rx_data_length < 5'd18) && (cnt_data == rx_data_length - 1'b1))
    next_state = RX_DRP_DATA;
  else if(cnt_data == rx_data_length - 1'b1)
    next_state = RX_CRC;
  else
    next_state = RX_DATA;
```

7. RX_DRP_DATA

接收填充数据状态 RX_DRP_DATA，首先对接收到的数据个数进行计数，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  cnt_drp_data <= 5'd0;
else if(curr_state == RX_DRP_DATA)
  cnt_drp_data <= cnt_drp_data + 1'b1;
else
  cnt_drp_data <= 5'd0;
```

当接收到的数据长度 rx_data_length 小于 18 的时候，此时表明我们主机发送的数据长度小于 18，此时为了满足以太网的最小帧长度要求，会在需要发送的数据后面补 0，此时补 0 的数据个数就应该等于 18 减去接收到的数据长度，当 cnt_drp_data 等于补 0 的数据个数时（18 - rx_data_length -1=17- rx_data_length），进入到 RX_CRC 状态，代码如下所示：

```
RX_DRP_DATA:
  if(cnt_drp_data == 5'd17 - rx_data_length)
    next_state = RX_CRC;
  else
    next_state = RX_DRP_DATA;
```

8. RX_CRC

接收FCS校验数据状态，如果接收的数据为0，则进入PKT_CHECK状态，否则一直处于 RX_CRC 状态，代码如下所示：

```
RX_CRC:
  if(rx_datav_dly2 == 1'b0)
    next_state = PKT_CHECK;
  else
    next_state = RX_CRC;
```

9. PKT_CHECK

进入 PKT_CHECK 状态之后，直接返回 IDLE 状态，重新进入一轮数据的接收，代码如下所示：

```
PKT_CHECK:
```

```
  next_state = IDLE;
```

通过以上对状态机的描述，以太网接收模块的基本功能就已经完成了，接下来对该模块比较常用的信号进行说明。

首先是 payload_valid_o 信号和 payload_dat_o 信号，当状态处于接收数据状态 RX_DATA 时，将输出数据有效信号 payload_valid_o 拉高并且将此时接收到的数据输出给 payload_dat_o 信号，代码如下所示：

```
//payload output
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  payload_valid_o <= 1'b0;
  payload_dat_o   <= 8'h0;
end
else if(curr_state == RX_DATA)
begin
  payload_valid_o <= 1'b1;
  payload_dat_o   <= rx_data_dly2;
end
else
begin
  payload_valid_o <= 1'b0;
  payload_dat_o   <= 8'h0;
end
```

然后就是 one_pkt_done 信号和 pkt_error 信号，当状态处于 PKT_CHECK 时，将传输完成标志信号 one_pkt_done 拉高，当 CRC 校验的结果 crc_check 等于 32'h2144DF1C 并且 reg_data_overflow 信号为低电平的时候，说明接收数据没错，让 pkt_error 信号为低电平，否则接收数据出错，将 pkt_error 信号拉高，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
begin
  one_pkt_done <= 1'b0;
  pkt_error    <= 1'b0;
end
else if(curr_state == PKT_CHECK)
begin
  one_pkt_done <= 1'b1;
  if(crc_check == 32'h2144DF1C && reg_data_overflow == 1'b0)
    pkt_error  <= 1'b0;
  else
    pkt_error  <= 1'b1;
end
```

```
else
begin
  one_pkt_done <= 1'b0;
  pkt_error    <= 1'b0;
end
```

上述代码中的 reg_data_overflow 信号，代表输入数据溢出，也就是当状态处于接收数据状态时并且输入数据溢出信号 data_overflow_i 为高电平时，将 reg_data_overflow 信号拉高，代表数据溢出，否则为低电平，代码如下所示：

```
always@(posedge clk125m or posedge reset_p)
if(reset_p)
  reg_data_overflow <= 1'b0;
else if(curr_state == RX_DATA && data_overflow_i == 1'b1)
  reg_data_overflow <= 1'b1;
else
  reg_data_overflow <= reg_data_overflow;
```

综上所述，以太网接收模块的代码就基本设计完成。再结合前面章节中的 RGMII 与 GMII 转换电路设计，我们便能实现 RGMII 接口的千兆以太网接收。关于该模块的完整代码请自行查看工程中的源代码文件。

58.2串口发送控制模块

为了验证以太网接收模块功能是否正常，我们计划使用串口打印出以太网接收到的数据，通过对比打印出的数据判断模块能否正常工作。

考虑到以太网发送模块的工作时钟为 125M，串口的工作时钟为 50M，两者时钟速率不匹配，设计首先需要添加一个 FIFO 来解决跨时钟域的问题。随后还需要一个串口发送控制模块，用于控制 FIFO 的读出以及串口发送模块的工作。

首先，添加一个双时钟 FIFO IP，设置输入输出位宽为 8 位，深度为 1024，配置界面如下图 58-3 所示。

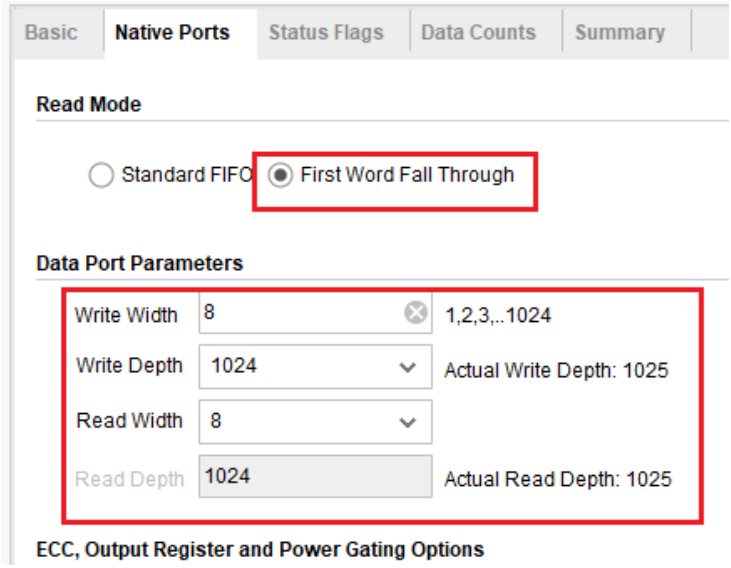

图 58-3 FIFO 配置界面

FIFO IP 的输入时钟为 125M，输入的数据由以太网接收模块输出的数据信号 rx_payload_dat，写使能为以太网接收的输出数据有效信号 rx_payload_valid，输出时钟为串口发送模块的 50M 时钟，读使能由串口发送控制模块产生，FIFO IP 的例化代码如下所示：

```
fifo_generator_0 fifo (
    .rst(reset_p),                  // input wire rst
    .wr_clk(clk125m),             // input wire wr_clk
    .rd_clk(Clk),             // input wire rd_clk
    .din(rx_payload_dat),                  // input wire [7 : 0] din
    .wr_en(rx_payload_valid),              // input wire wr_en
    .rd_en(fifo_rd_req),              // input wire rd_en
    .dout(dout),                // output wire [7 : 0] dout
    .full(),                // output wire full
    .empty(rx_empty),              // output wire empty
    .wr_rst_busy(),  // output wire wr_rst_busy
    .rd_rst_busy()  // output wire rd_rst_busy
);
```

然后就是设计一个串口发送控制模块，该模块我们通过一个状态机实现，当处于状态 0 时，当 fifo_empty 为低电平，也就是 FIFO 中被写入数据之后，产生 FIFO 的读请求信号 fifo_rd_req，然后进入到状态 1；当处于状态 1 时，将 fifo_rd_req 信号拉低，并将串口发送使能信号 uart_send_en 拉高，同时把 FIFO 读出来的数据 fifo_rd_data 交由串口发送，然后进入状态 2；处于状态 2 时，将 uart_send_en 拉低，当得到串口发送完成信号 uart_tx_done 时，进入状态 0 重新开始读取数据然后发送，串口发送控制模块中状态机代码如下所示：

```
always@(posedge clk or posedge reset_p)
```

```
if(reset_p) begin
    state <= 1'd0;
    uart_tx_data <= 8'd0;
    fifo_rd_req <= 1'd0;
    uart_send_en <= 1'd0;
end
else begin
    case(state)
        0:
        begin
            if(!fifo_empty) begin       //如果 FIFO 不为空，可以开始发送
                fifo_rd_req <= 1'd1;
                state <= 2'd1;
            end
            else begin
                fifo_rd_req <= 1'd0;
                state <= 2'd0;
            end
        end

        1:
        begin
            fifo_rd_req <= 1'd0;
            uart_send_en <= 1'd1;
            uart_tx_data <= fifo_rd_data;
            state <= 2'd2;
        end

        2:
        begin
            uart_send_en <= 1'd0;
            if(uart_tx_done)
                state <= 2'd0;
            else
                state <= 2'd2;
        end

        default:;
    endcase
end
```

至此，串口发送控制模块就设计完成了，在本次实验中，还需要添加一个串口发送模块 uart_byte_tx，这个模块在前面的实验中已经介绍过了，本次实验就不再做讲解，只需要将该模块的源文件添加至本工程中，然后在顶层模块中完成例化即可，在工程中我们还可以添加 ILA 信号，抓取以太网发送模块的信号，进一步验证数据是否无误。

58.3系统板级测试

通过上述步骤，代码部分的设计就完成了，随后分配管脚并约束电平。本次设计的引脚分配表如下：

Pin Name	Signial Name	Pin NO.		Pin Name	Signial Name	Pin NO.
PL_ENET0_RX_DATA3	rgmii_rxd[3]	Y12		PL_ENET0_RX_DV	rgmii_rxdv	Y15
PL_ENET0_RX_DATA2	rgmii_rxd[2]	Y13		PL_ENET0_RESET	eth_reset_n	U1
PL_ENET0_RX_DATA1	rgmii_rxd[1]	W12		FPGA_UART_TX	uart_tx	V18
PL_ENET0_RX_DATA0	rgmii_rxd[0]	W13		FPGA_KEY0	reset_n	R4
PL_ENET0_RX_CLK	rgmii_rx_clk	Y14		FPGA_GCLK1	Clk	L5

完成以上步骤后，生成 bit 并连接硬件准备板级验证。本次设计硬件连接如下：

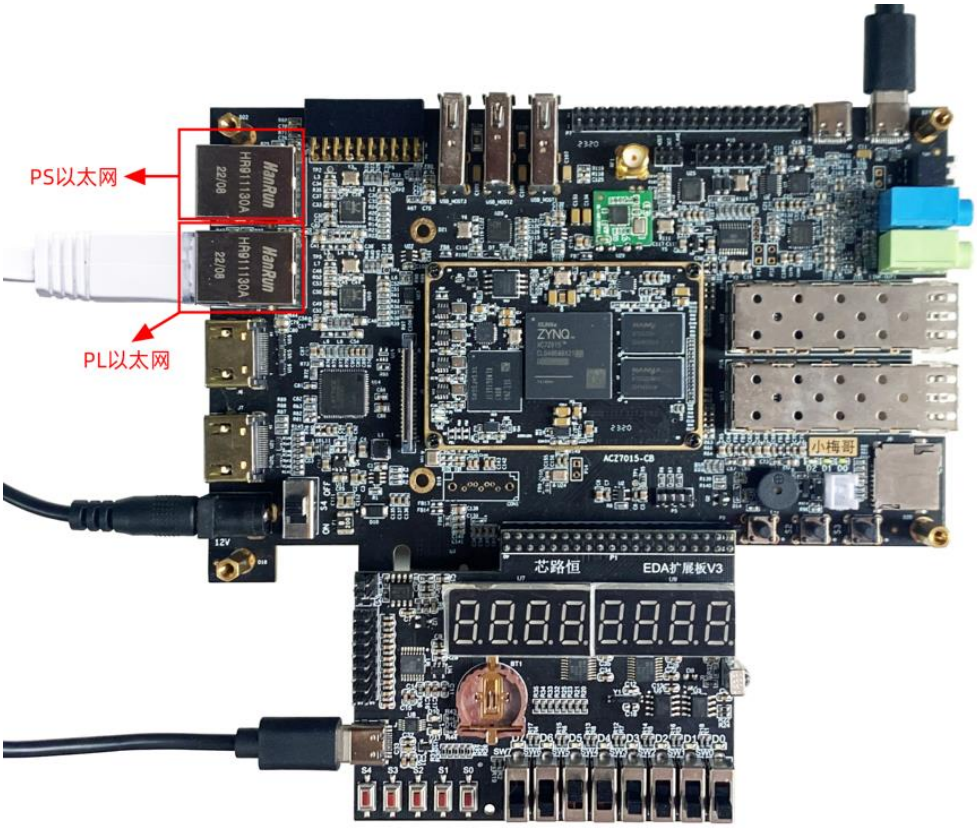

图 58-4 硬件连接图

网线一端插入 ZYNQ 开发板的 PL 端网口，另一端插入到电脑网口。由于需要使用到 PL 端串口，所以本次需要使用到 EDA 拓展板。串口线的一端接在 EDA 串口上，另一端插入电脑的 USB 口。连接完成之后，给开发板上电，并下载 bit 文件。

bit 文件下载成功之后，给 LIA 添加测试信号，并设置输出数据有效信号的上升沿作为触发条件，如下图 58-5 所示。

图 58-5 添加 ILA 测试信号

然后同时打开串口调试助手和网络调试助手，分别配置如下图 58-6 所示和图 58-7 所示。

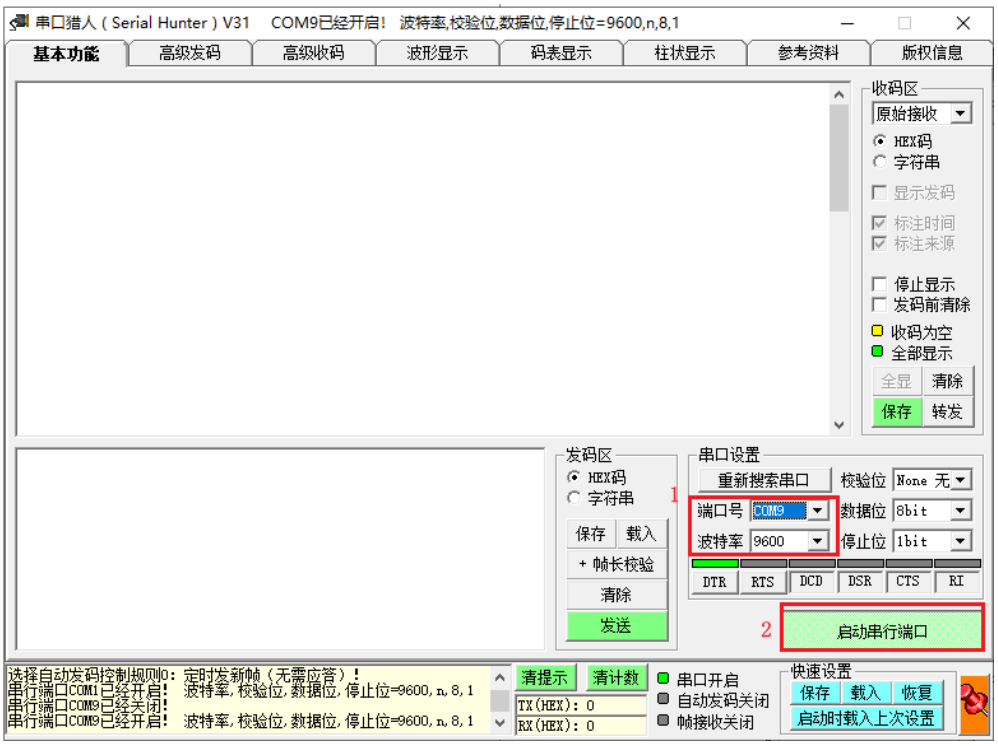

图 58-6 串口助手配置界面

图 58-7 网口配置界面

点击网络调试助手中的发送之后，可以看到网络调试助手界面和串口调试助手界面如下图 58-8、图 58-9 所示。

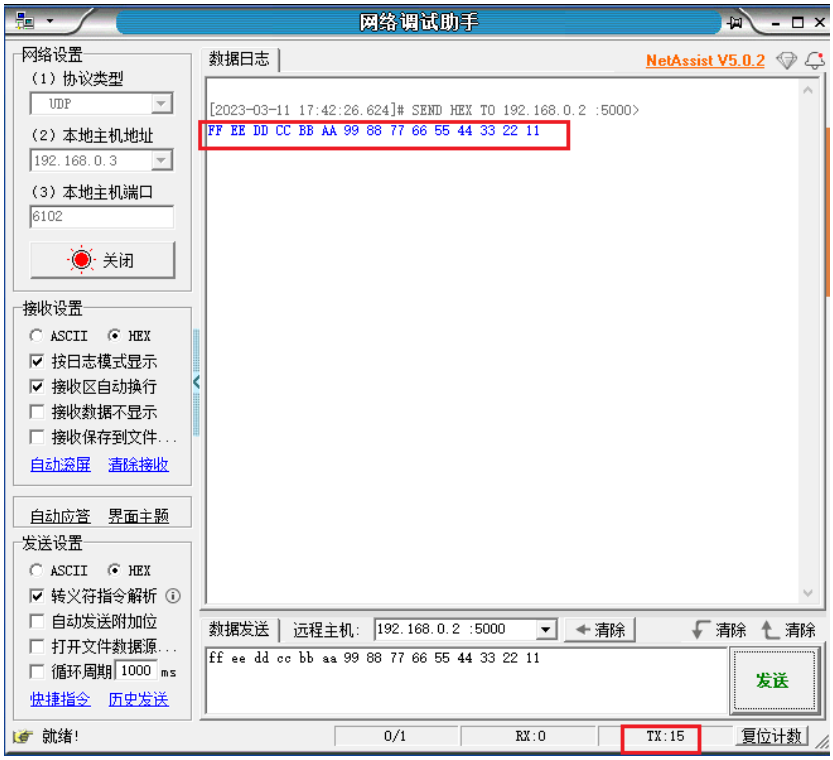

图 58-8 网络调试助手界面

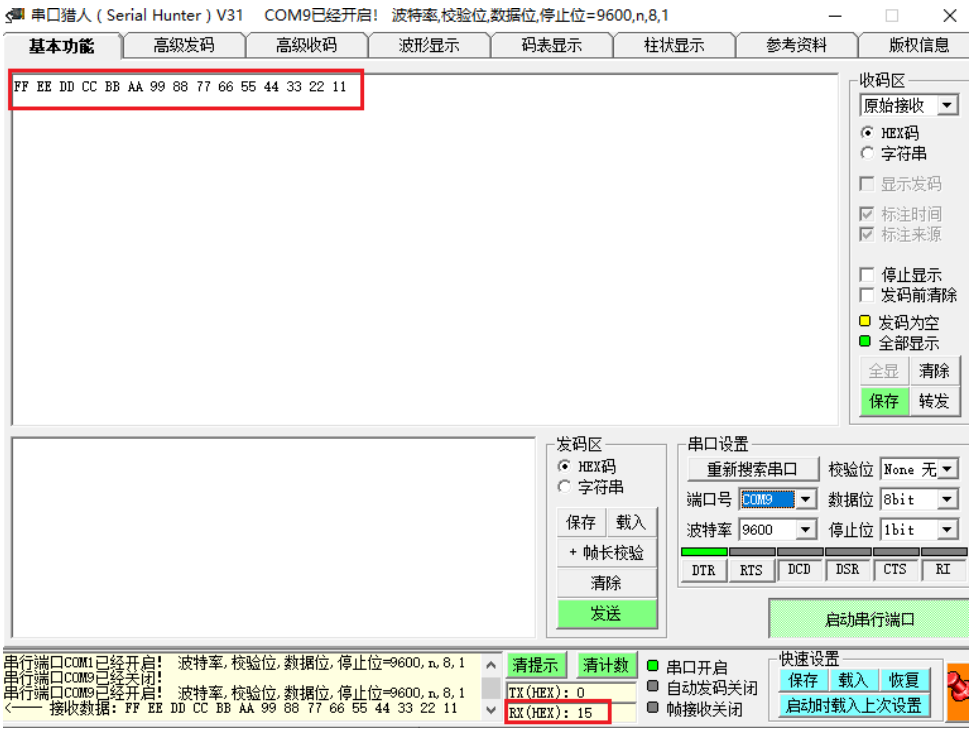

图 58-9 串口调试助手界面

从上述图中可以看出串口接收的数据和网口发送的数据一致，接收的数据个数也一致，这说明以太网接收模块的功能正常，这时，我们再查看 ILA 抓取的波形如图 58-10 所示。

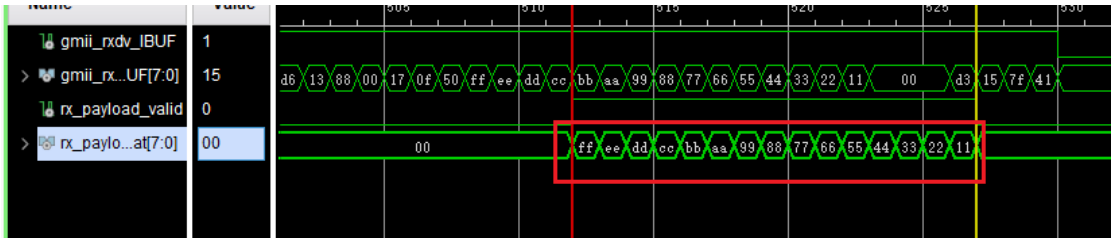

图 58-10 ILA 抓取波形数据图

从上述图中可以看出，LIA 抓取的以太网接收模块接收到的数据和网络调试助手发送的一致，进一步验证了以太网发送模块的设计没有问题，符合设计预期功能。