# 1 实用性 UDP 接收逻辑设计与实现

| | |
|---|---|
| 工程源码 | \|-ACZ702 开发板<br>\|-- acz702_eth_udp_rx_rgmii |
| 相关视频课程 | 无 |
| 说明 | 如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。 |

章节导读

本章实验将结合前面章节中介绍过的以太网各层级数据字段的内容以及串口接收模块的设计，实现网口接收数据，串口发送数据的功能，从而验证以太网接收模块是否工作正常。

## 1.1 系统整体设计

本小节的内容，适配于工程 acz702_eth_udp_rx_rgmii，该工程实现功能：通过网口接收 PC 端发送过来的数据，然后通过串口将接收到的数据发送出去，最后判断数据是否一致，以此验证以太网接收模块是否工作正常。系统的整体设计如下图 1-1 所示：

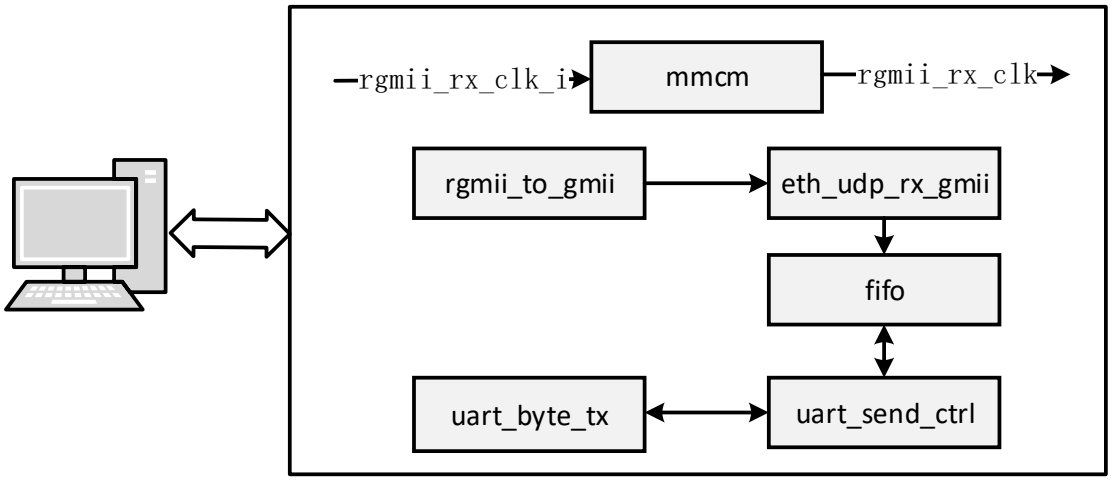


图 1-1 以太网接收串口发送实验整体设计框图

下面对上述图中的模块进行简单的说明：

1. mmcm 模块：锁相环模块，将 rgmii 接口时钟信号 rgmii_rx_clk_i 偏移 90°得到 rgmii_rx_clk 时钟信号，这样做是为了在时钟信号的上升沿/下降沿取数据时，取得数据刚好是数据信号 rgmii_rxd 的正中间，使得采样的数据处于最稳定的状态，如下图 1-2 所示。

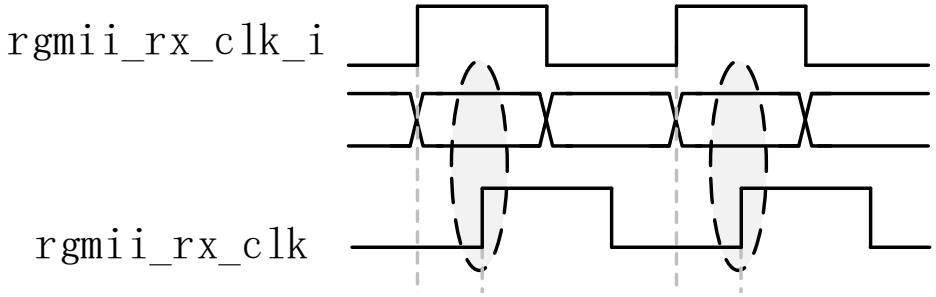


图 1-2 采集数据波形图

2. rgmii_to_gmii 模块：以太网接收 rgmii 转 gmii 模块，RX 的数据位宽从 8 位变为 4 位。
3. eth_udp_rx_gmii 模块：以太网接收模块，接收 PC 端传输过来的数据，最终将接收到的数据进行输出。
4. fifo 模块：FIFO IP，以太网接收模块的工作时钟是 125M，串口发送模块的工作时钟是 50M，为了解决跨时钟域传输的问题，需要一个双时钟 FIFO 用于缓存数据。
5. uart_send_ctrl 模块：串口发送控制模块，控制从 FIFO 中读取数据，并控制串口发送模块发送数据。
6. uart_byte_tx 模块：串口发送模块，发送从网口接收到的数据。

本章实验将讲解 eth_udp_rx_gmii 模块和 uart_send_ctrl 模块的设计，其余模块的设计请参看前面章节的实验内容。

## 1.2 以太网接收模块（eth_udp_rx_gmii）

以太网接收是以太网发送的逆过程，其模块的基本框图如下图 1-3 所示：

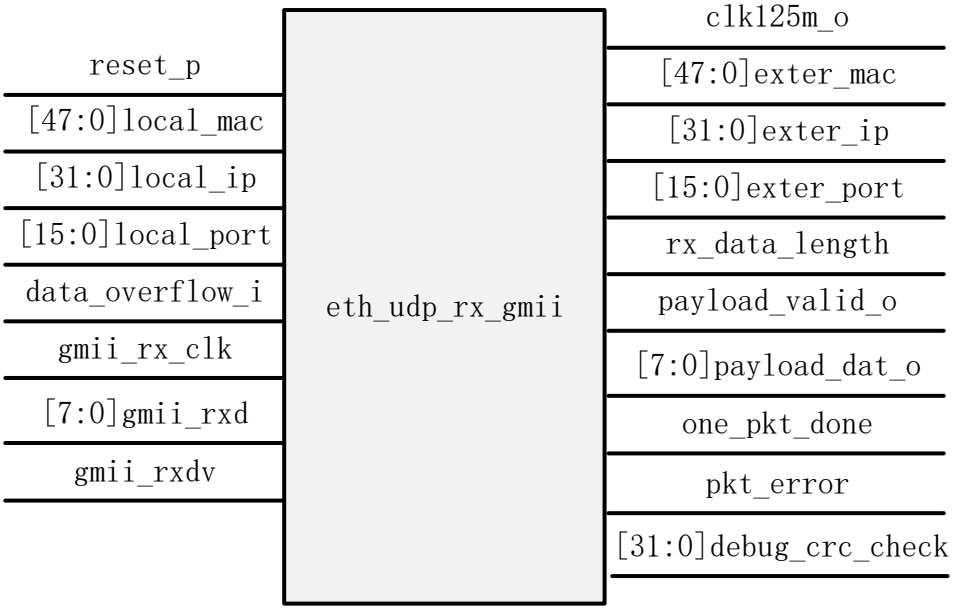


图 1-3 以太网接收模块的基本结构图

该模块的信号说明如下表 1-1 所示。

表 1-1 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 1-4 所示。

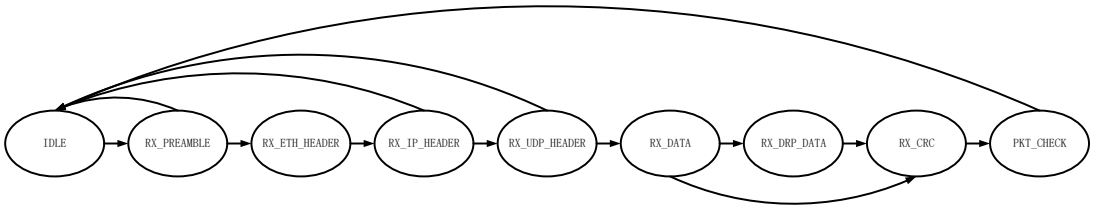


图 1-4 以太网接收模块状态转移图

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，否者处于 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 个的前导码（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，否者清零，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、x_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_data_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，否者为低，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，否者处于 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，关于该模块的完整代码请自行查看工程中的源代码文件。

## 1.3 串口发送控制模块

在验证以太网发送模块时，我们是通过以太网接收，串口发送来实现的，但是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，为了解决这一问题，首先通过一个双时钟 FIFO 缓存以太网接收过来的数据，然后通过串口发送控制模块控制 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 1-5 所示。

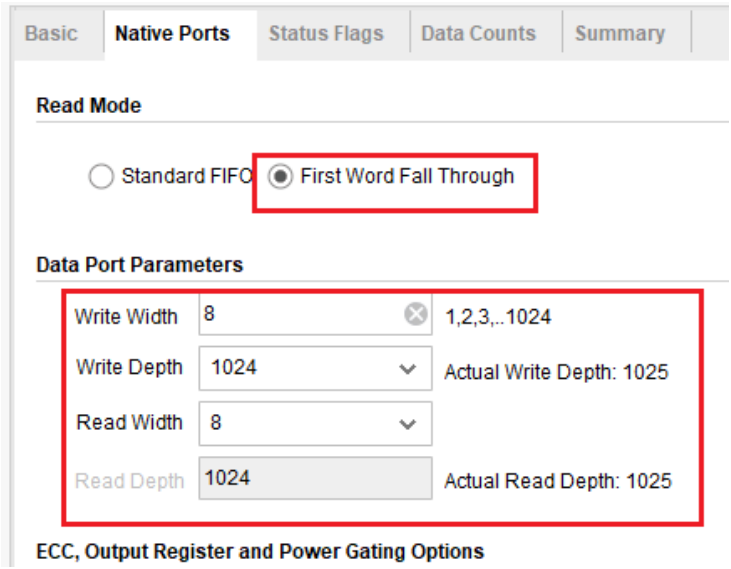


图 1-5 FIFO 配置界面

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。

# 1.4 管脚绑定

通过上述描述，代码部分的设计就完成了，然后进行管脚约束，管脚约束如下，最后生成 bit 文件，进行板级验证。

表 1-2 管脚约束表

| Pin Name | Pin NO |
|---|---|
| Clk | U18 |
| reset_n | F20 |
| rgmii_rxd[3] | P15 |
| rgmii_rxd[2] | Y16 |
| rgmii_rxd[1] | V15 |
| rgmii_rxd[0] | P14 |
| rgmii_rxdv | Y14 |
| rgmii_rx_clk_i | N18 |
| uart_tx | J16 |

# 1.5 系统板级测试

## 1.5.1 系统所需硬件

1. ACZ702 开发板一个
2. 电源线一根（可选）
3. Type-C 下载线两根，一根用于供电和下载，一根用于接拓展板上的串口
4. EDA 扩展板一块
5. 网线一根

## 1.5.2 硬件连接

本次系统设计硬件方便连接如图 1-6 所示：

1. 将 EDA 扩展版连接在 ACZ702 开发板的扩展接口上，开发板上的扩展接口引脚数与 EDA 扩展板的引脚数都为 40，因此正确连接情况下是不会多出引脚的。
2. 连接好下载线，注意是靠近电源接口的 Type-C 口，开发板上标有 JTAG。
3. 连接好扩展板上的串口，另一端连接至电脑的 USB 口上。
4. 用网线连接开发板上 PL 侧的网口和电脑的网口

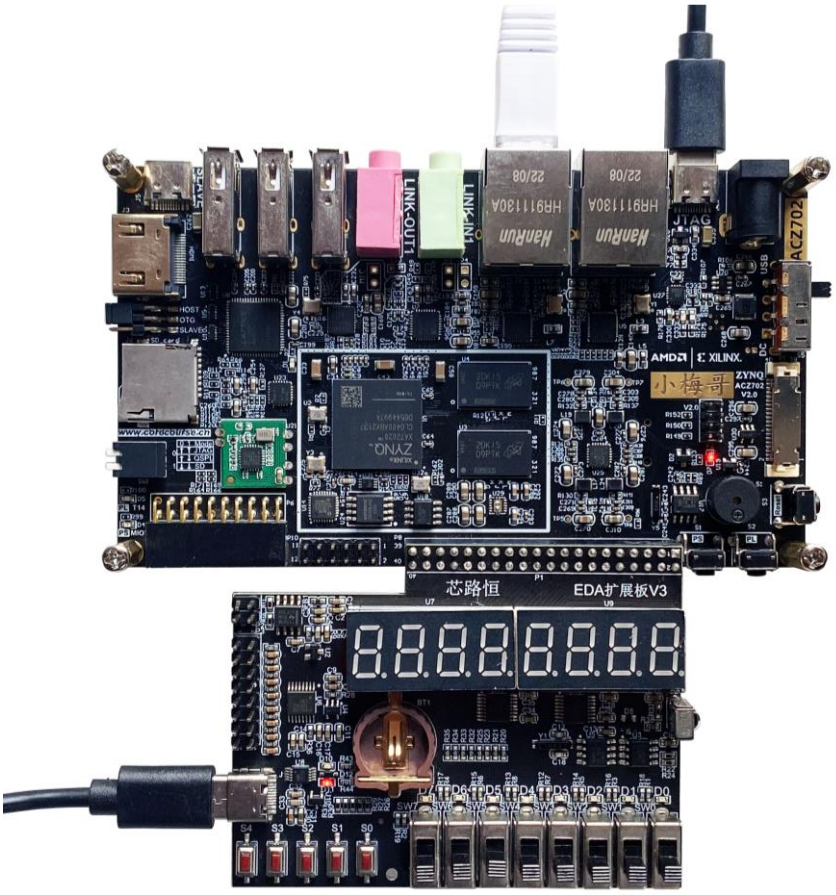


图 1-6 硬件连接图

## 1.5.3 下载验证

首先下载本次实验生成的 bit 文件，如下图 1-7 所示：

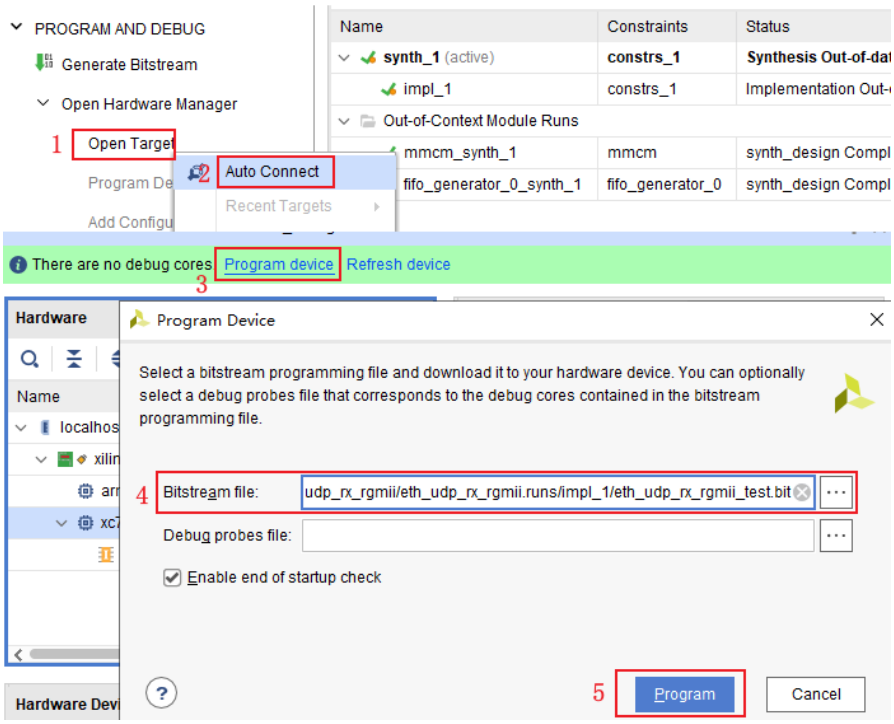


图 1-7 下载 bit 文件

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 1-8 所示和图 1-9 所示。


图 1-8 串口助手配置界面


图 1-9 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 1-10、图 1-11 所示。

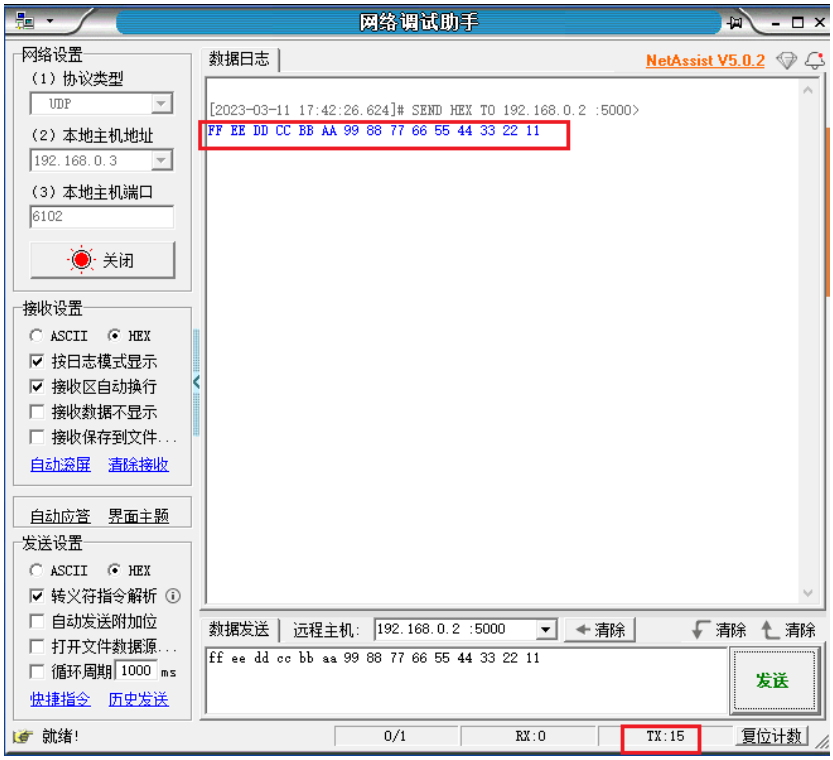


图 1-10 网络调试助手界面

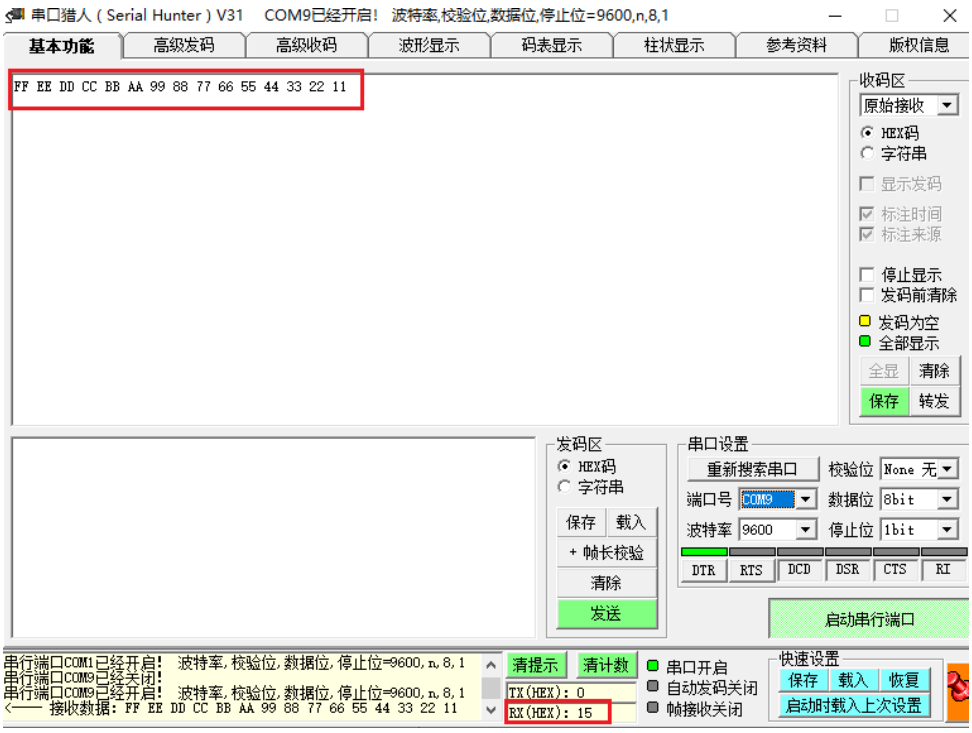


图 1-11 串口调试助手界面

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。

## 1.6 章节总结

本章实验通过对比 PC 端通过网络调试助手发送的数据和串口助手接收到的数据可以看出，接收和发送的数据一致，这也就说明以太网发送模块工作正常，在后续需要通过以太网去接收数据时，都可以调用该模块去接收。

需要注意的是，本次实验在使用串口助手去打开端口的时候，会存在两个端口，这时就需要我们打开设备管理器查看，如下图 1-12 所示才是串口助手需要打开的端口号。


图 1-12 设备管理显示端口号