基于FPGA的UDP实现（包含源工程文件）

基于FPGA的以太网相关文章导航，点击查看。

1、概括

前文通过 FPGA 实现了ARP和ICMP协议，ARP协议一般用来获取目的IP地址主机的MAC地址，ICMP通过回显请求和回显应答来判断 以太网 链路是否通畅，这两个协议都不是用来传输用户数据的。如果用户需要向PC端传输大量数据，那么就必须使用TCP或者UDP协议了。

网上关于UDP和TCP的优缺点对比其实很多，可以自行搜索，本文简要概括一下优缺点。

TCP优点是稳定，接收端接收到TCP数据报文后会回复发送端，如果接收的报文有误，发送端会把错误的报文重新发送一遍。而且TCP本来就有握手机制，所以数据的传输会更可靠。正是由于握手机制，导致实现的TCP协议的逻辑比较复杂，传输速度也不会很高，还需要更多存储资源取存储已经发送的数据，直到收到该数据传输无误后才能丢弃。因此FPGA一般不会采用该协议进行大量数据的传输（当然如果通过Verilog HDL实现可靠的TCP协议，那还是很有用的，毕竟这块的代码很贵）。

UDP优点是协议简单，没有握手机制，传输数据的速度就很快，这对于FPGA传输图像数据之类的设计比较实用。由于UDP没有握手机制，可靠性相比TCP就会低很多，有得必有失嘛。

因此，FPGA一般通过UDP协议向PC端发送大量数据，所以本文通过FPGA实现UDP协议。

2、UDP协议讲解

UDP协议的框图如下所示，与前文的 ICMP协议 构成类似，UDP协议数据报文位于IP的数据段，IP首部只有协议类型与ICMP协议类型参数不一致，ICMP的IP协议类型编号为1，UDP的IP协议类型编号为17。

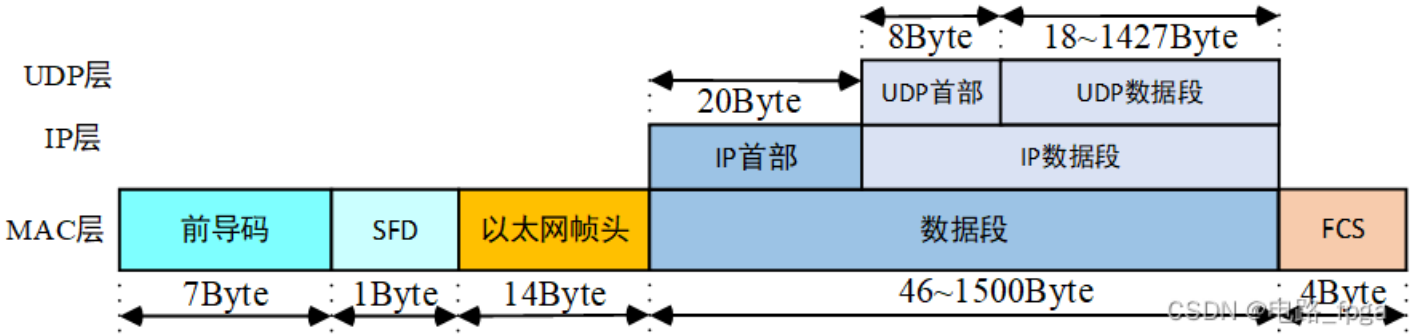

图1 UDP协议框图

前导码、帧起始符、以太网帧头、IP首部、FCS校验在前文讲解ARP协议和ICMP协议的时候都详细讲解过，所以本文就不再赘述了。

UDP的首部组成如下所示，包括源UDP源端口地址、UDP目的端口地址、UDP长度、UDP校验码。

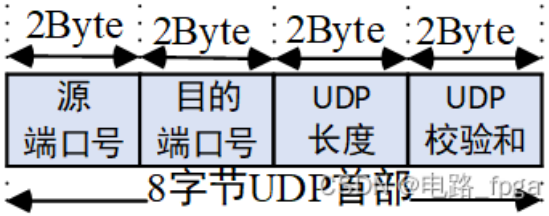

图2 UDP首部组成

源端口号：2个字节的发送端端口号，用于区分不同的发送端口。

目的端口号：2个字节的接收端端口号。

UDP长度：UDP首部和数据段的长度，单位字节，对于接收方来说该长度其实作用不大，因为UDP数据段的长度可以通过IP首部的总长度和IP首部长度计算出来。

UDP校验和：计算方式与IP首部校验和一致，需要对UDP伪首部、UDP首部、UDP数据进行校验。伪首部包括源IP地址、目的IP地址、协议类型、UDP长度。

这种校验方式其实对于FPGA来说很麻烦，因为校验码需要在数据之前发送，而计算校验码有需要得到数据，就意味着如果想要计算校验码，就必须使用存储资源把待发送的数据存起来，计算出校验码以后，才开始传输数据。比较友好的是该校验码可以直接置零处理（如果不做校验，该值必须为0，否则校验失败的数据报文会被直接丢弃）。不校验数据的UDP协议变得特别简单。

UDP协议的数据组成如下所示：

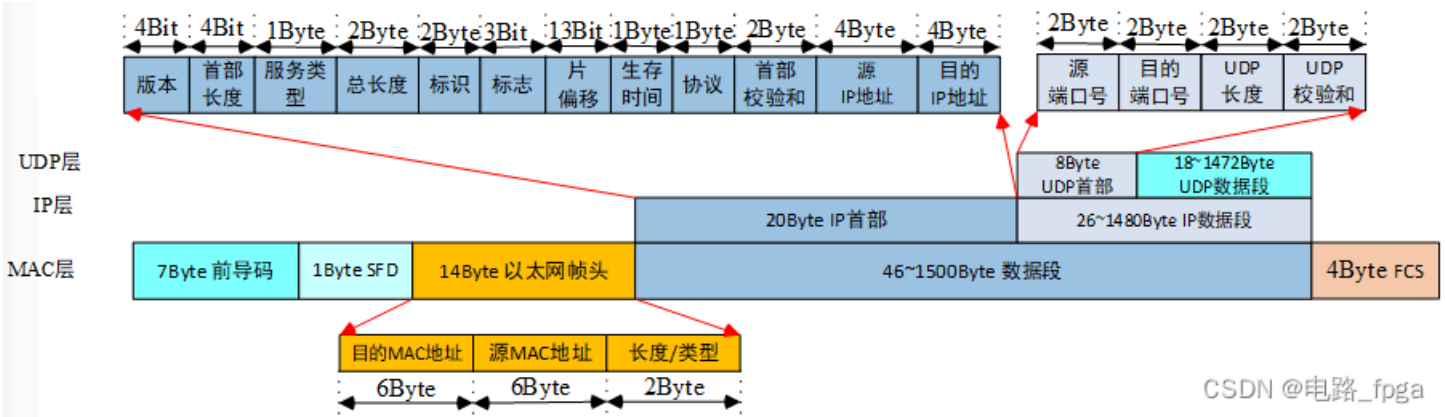

图3 以太网UDP协议帧组成

以太网的协议组成就介绍这么多了，最后注意在发送数据时，数据段必须大于等于18字节，少于18字节数据时，应补零凑齐18字节数据发送。

3、UDP顶层模块

UDP的设计与前文的ARP、ICMP模块设计差不多，UDP顶层模块如下图所示，包括UDP接收模块udp_rx、UDP接收的CRC校验模块、UDP的发送模块udp_tx、UDP发送的CRC校验模块。

UDP接收模块内部没有做IP首部校验，只做了CRC校验模块，在加上FPGA逻辑判断，基本上都能判断对错了，最后把接收的数据和数据个数输出。发送模块检测到开始发送信号后，开始发送信号，当发送到数据段之后，把数据请求信号拉高，从外部输入需要发送的数据流。

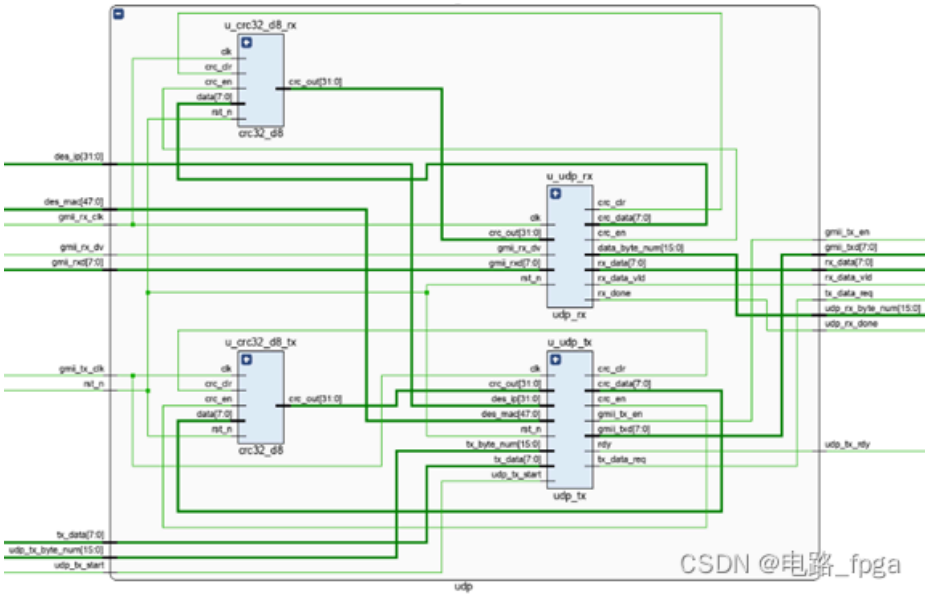

图4 UDP顶层模块

UDP顶层模块的参考代码如下所示：

```
    //例化udP接收模块；
    udp_rx #(
        .BOARD_MAC      ( BOARD_MAC     ),//开发板MAC地址 00-11-22-33-44-55；
        .BOARD_IP       ( BOARD_IP      ) //开发板IP地址 192.168.1.10；
    )
    u_udp_rx (
        .clk            ( gmii_rx_clk       ),//时钟信号；
        .rst_n          ( rst_n             ),//复位信号，低电平有效；
        .gmii_rx_dv     ( gmii_rx_dv        ),//GMII输入数据有效信号；
```

```
        .gmii_rxd       ( gmii_rxd          ),//GMII输入数据;
        .crc_out        ( rx_crc_out        ),//CRC校验模块输出的数据;
        .rx_done        ( udp_rx_done       ),//UDP接收完成信号，高电平有效;
        .rx_data_vld    ( rx_data_vld       ),//以太网接收到有效数据指示信号;
        .rx_data        ( rx_data           ),//以太网接收数据。
        .data_byte_num  ( udp_rx_byte_num   ),//以太网接收的有效数据字节数 单位:byte
        .des_port       (                   ),//UDP接收的目的端口号;
        .source_port    (                   ),//UDP接收到的源端口号;
        .crc_data       ( rx_crc_data       ),//需要CRC模块校验的数据;
        .crc_en         ( rx_crc_en         ),//CRC开始校验使能;
        .crc_clr        ( rx_crc_clr        ) //CRC数据复位信号;
    );

    //例化接收数据时需要的CRC校验模块;
    crc32_d8  u_crc32_d8_rx (
        .clk        ( gmii_rx_clk   ),//时钟信号;
        .rst_n      ( rst_n         ),//复位信号，低电平有效;
        .data       ( rx_crc_data   ),//需要CRC模块校验的数据;
        .crc_en     ( rx_crc_en     ),//CRC开始校验使能;
        .crc_clr    ( rx_crc_clr    ),//CRC数据复位信号;
        .crc_out    ( rx_crc_out    ) //CRC校验模块输出的数据;
    );

    //例化UDP发送模块;
    udp_tx #(
        .BOARD_MAC      ( BOARD_MAC     ),//开发板MAC地址 00-11-22-33-44-55;
        .BOARD_IP       ( BOARD_IP      ),//开发板IP地址 192.168.1.10;
        .DES_MAC        ( DES_MAC       ),//目的MAC地址 ff_ff_ff_ff_ff_ff;
        .DES_IP         ( DES_IP        ),//目的IP地址 192.168.1.102;
        .BOARD_PORT     ( BOARD_PORT    ),//板子的UDP端口号;
        .DES_PORT       ( DES_PORT      ),//源端口号;
        .ETH_TYPE       ( ETH_TYPE      ) //以太网帧类型，16'h0806表示ARP协议，16'h0800表示IP协议;
    )
    u_udp_tx (
        .clk            ( gmii_tx_clk       ),//时钟信号;
        .rst_n          ( rst_n             ),//复位信号，低电平有效;
        .udp_tx_start   ( udp_tx_start      ),//UDP发送使能信号;
        .tx_byte_num    ( udp_tx_byte_num   ),//UDP数据段需要发送的数据。
        .des_mac        ( des_mac           ),//发送的目标MAC地址;
        .des_ip         ( des_ip            ),//发送的目标IP地址;

```

```
        .crc_out        ( tx_crc_out        ),//CRC校验数据；
        .crc_en         ( tx_crc_en         ),//CRC开始校验使能；
        .crc_clr        ( tx_crc_clr        ),//CRC数据复位信号；
        .crc_data       ( tx_crc_data       ),//输出给CRC校验模块进行计算的数据；
        .tx_data_req    ( tx_data_req       ),//需要发送数据请求信号；
        .tx_data        ( tx_data           ),//需要发送的数据；
        .gmii_tx_en     ( gmii_tx_en        ),//GMII输出数据有效信号；
        .gmii_txd       ( gmii_txd          ),//GMII输出数据；
        .rdy            ( udp_tx_rdy        ) //模块忙闲指示信号，高电平表示该模块处于空闲状态；
    );

    //例化发送数据时需要的CRC校验模块；
    crc32_d8  u_crc32_d8_tx (
        .clk        ( gmii_tx_clk   ),//时钟信号；
        .rst_n      ( rst_n         ),//复位信号，低电平有效；
        .data       ( tx_crc_data   ),//需要CRC模块校验的数据；
        .crc_en     ( tx_crc_en     ),//CRC开始校验使能；
        .crc_clr    ( tx_crc_clr    ),//CRC数据复位信号；
        .crc_out    ( tx_crc_out    ) //CRC校验模块输出的数据；
    );
```

对应的TestBench文件如下所示：

```
`timescale 1 ns/1 ns
module test();
    localparam  CYCLE       =   8                           ;//系统时钟周期，单位ns，默认8ns；
    localparam  RST_TIME    =   10                          ;//系统复位持续时间，默认10个系统时钟周期；
    localparam  STOP_TIME   =   1000                        ;//仿真运行时间，复位完成后运行1000个系统时钟后停止；
    localparam  BOARD_MAC   =   48'h00_11_22_33_44_55       ;
    localparam  BOARD_IP    =   {8'd192,8'd168,8'd1,8'd10}  ;
    localparam  BOARD_PORT  =   16'd1234                    ;//开发板的UDP端口号；
    localparam  DES_PORT    =   16'd5678                    ;//UDP目的端口号；
    localparam  DES_MAC     =   48'h23_45_67_89_0a_bc       ;
    localparam  DES_IP      =   {8'd192,8'd168,8'd1,8'd23}  ;
    localparam  ETH_TYPE    =   16'h0800                    ;//以太网帧类型 IP
```

```
13
14  reg                         clk                         ;//系统时钟，默认100MHz；
15  reg                         rst_n                       ;//系统复位，默认低电平有效；
16  reg         [7 : 0]         tx_data                     ;
17  reg                         udp_tx_start                ;
18  reg        [15 : 0]         udp_tx_byte_num             ;
19
20  wire         [7 : 0]        gmii_rxd                    ;
21  wire                        gmii_rx_dv                  ;
22  wire                        gmii_tx_en                  ;
23  wire        [7 : 0]         gmii_txd                    ;
24  wire                        udp_rx_done                 ;
25  wire        [15 : 0]        udp_rx_byte_num             ;
26  wire                        udp_tx_rdy                  ;
27  wire                        tx_data_req                 ;
28  wire                        rx_data_vld                 ;
29  wire        [7 : 0]         rx_data                     ;
30
31  assign gmii_rx_dv = gmii_tx_en;
32  assign gmii_rxd = gmii_txd;
33
34  udp #(
35      .BOARD_MAC  ( BOARD_MAC ),
36      .BOARD_IP   ( BOARD_IP  ),
37      .DES_MAC    ( DES_MAC   ),
38      .DES_IP     ( DES_IP    ),
39      .BOARD_PORT ( BOARD_PORT),//板子的UDP端口号；
40      .DES_PORT   ( DES_PORT  ),//源端口号；
41      .ETH_TYPE   ( ETH_TYPE  )
42  )
43  u_udp (
44      .rst_n              ( rst_n             ),
45      .gmii_rx_clk        ( clk               ),
46      .gmii_rx_dv         ( gmii_rx_dv        ),
47      .gmii_rxd           ( gmii_rxd          ),
48      .gmii_tx_clk        ( clk               ),
49      .udp_tx_start       ( udp_tx_start      ),
50      .udp_tx_byte_num    ( udp_tx_byte_num   ),
51      .des_mac            ( BOARD_MAC         ),
52      .des_ip             ( BOARD_IP          ),
53
```

```
        .gmii_tx_en         ( gmii_tx_en        ),
        .gmii_txd           ( gmii_txd          ),
        .udp_rx_done        ( udp_rx_done       ),
        .udp_rx_byte_num    ( udp_rx_byte_num   ),
        .udp_tx_rdy         ( udp_tx_rdy        ),
        .rx_data            ( rx_data           ),
        .rx_data_vld        ( rx_data_vld       ),
        .tx_data_req        ( tx_data_req       ),
        .tx_data            ( tx_data           )
    );

    //生成周期为CYCLE数值的系统时钟;
    initial begin
        clk = 0;
        forever #(CYCLE/2) clk = ~clk;
    end

    //生成复位信号;
    initial begin
        #1;udp_tx_start = 0; udp_tx_byte_num = 19;tx_data = 0;
        rst_n = 1;
        #2;
        rst_n = 0;//开始时复位10个时钟;
        #(RST_TIME*CYCLE);
        rst_n = 1;
        #(20*CYCLE);
        repeat(3)begin
            udp_tx_start = 1'b1;
            udp_tx_byte_num = {$random} % 64;//只产生64以内随机数，便于测试，不把数据报发的太长了;
            #(CYCLE);
            udp_tx_start = 1'b0;
            #(CYCLE);
            @(posedge udp_tx_rdy);
            #(100*CYCLE);
        end
        #(20*CYCLE);
        $stop;//停止仿真;
    end

    always@(posedge clk)begin

```

```
95          if(tx_data_req)begin//产生0~255随机数作为测试；
96              tx_data <= {$random} % 256;
97          end
98      end

endmodule
```

4、UDP接收模块

UDP接收模块与前文的ICMP接收模块的设计类似，都可以采用状态机嵌套一个计数器进行实现，状态机对应的状态转换图如下所示。

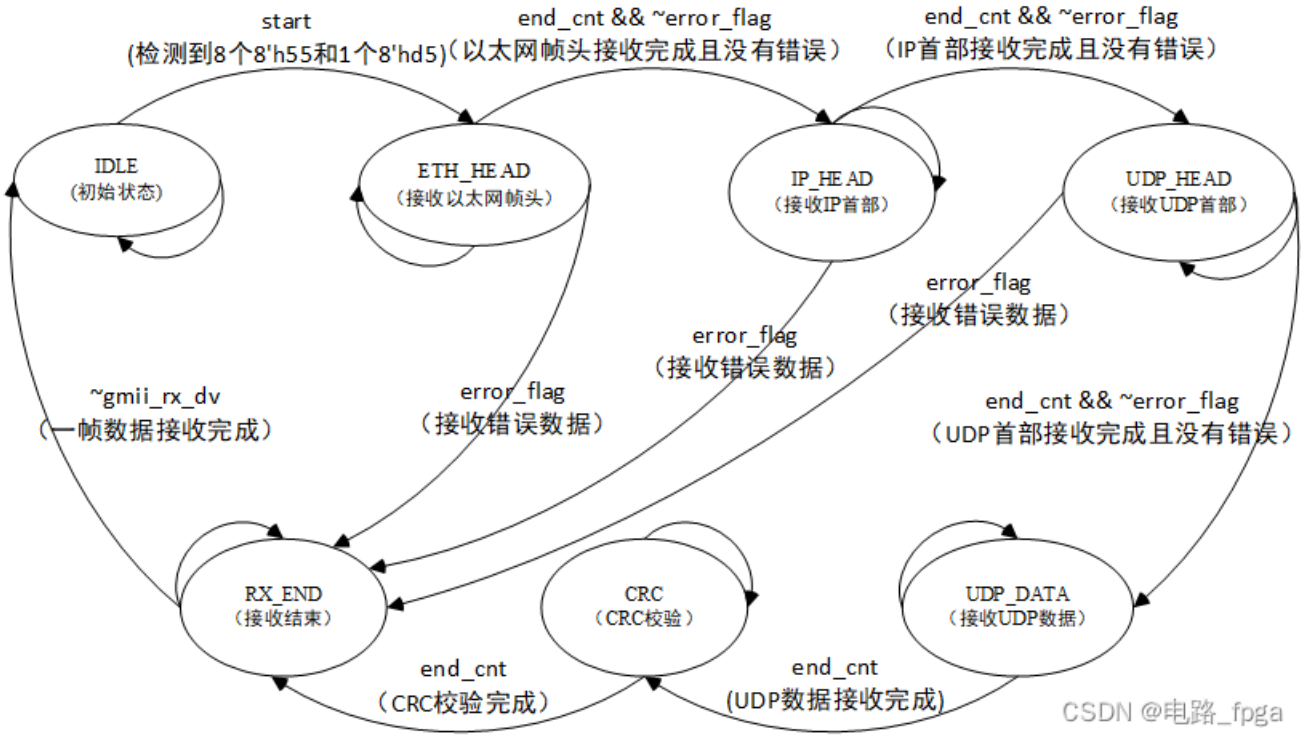

图5 状态转换图

需要注意判断UDP首部的目的端口地址是不是开发板的端口地址，其余部分与ICMP的接收模块差不多，不在赘述了。

该模块对应的代码如下所示：

```
    //The first section: synchronous timing always module, formatted to describe the transfer of the secondary register to the live register 
    always@(posedge clk)begin
        if(!rst_n)begin
            state_c <= IDLE;
        end
        else begin
            state_c <= state_n;
        end
    end

    //The second paragraph: The combinational logic always module describes the state transition condition judgment.
    always@(*)begin
        case(state_c)
            IDLE:begin
                if(start)begin//检测到前导码和SFD后跳转到接收以太网帧头数据的状态。
                    state_n = ETH_HEAD;
                end
                else begin
                    state_n = state_c;
                end
            end
            ETH_HEAD:begin
                if(error_flag)begin//在接收以太网帧头过程中检测到错误。
                    state_n = RX_END;
                end
                else if(end_cnt)begin//接收完以太网帧头数据，且没有出现错误，则继续接收IP协议数据。
                    state_n = IP_HEAD;
                end
                else begin
                    state_n = state_c;
                end
            end
            IP_HEAD:begin
                if(error_flag)begin//在接收IP帧头过程中检测到错误。
                    state_n = RX_END;
                end
                else if(end_cnt)begin//接收完以IP帧头数据，且没有出现错误，则继续接收UDP协议数据。
                    state_n = UDP_HEAD;

```

```
            end
            else begin
                state_n = state_c;
            end
        end
        UDP_HEAD:begin
            if(error_flag)begin//在接收UDP协议帧头过程中检测到错误。
                state_n = RX_END;
            end
            else if(end_cnt)begin//接收完以UDP帧头数据，且没有出现错误，则继续接收UDP数据。
                state_n = UDP_DATA;
            end
            else begin
                state_n = state_c;
            end
        end
        UDP_DATA:begin
            if(error_flag)begin//在接收UDP协议数据过程中检测到错误。
                state_n = RX_END;
            end
            else if(end_cnt)begin//接收完UDP协议数据且未检测到数据错误。
                state_n = CRC;
            end
            else begin
                state_n = state_c;
            end
        end
        CRC:begin
            if(end_cnt)begin//接收完CRC校验数据。
                state_n = RX_END;
            end
            else begin
                state_n = state_c;
            end
        end
        RX_END:begin
            if(~gmii_rx_dv)begin//检测到数据线上数据无效。
                state_n = IDLE;
            end
            else begin
```

```
                    state_n = state_c;
                end
            end
            default:begin
                state_n = IDLE;
            end
        endcase
    end

    //将输入数据保存6个时钟周期，用于检测前导码和SFD。
    //注意后文的state_c与gmii_rxd_r[0]对齐。
    always@(posedge clk)begin
        gmii_rxd_r[6] <= gmii_rxd_r[5];
        gmii_rxd_r[5] <= gmii_rxd_r[4];
        gmii_rxd_r[4] <= gmii_rxd_r[3];
        gmii_rxd_r[3] <= gmii_rxd_r[2];
        gmii_rxd_r[2] <= gmii_rxd_r[1];
        gmii_rxd_r[1] <= gmii_rxd_r[0];
        gmii_rxd_r[0] <= gmii_rxd;
        gmii_rx_dv_r <= {gmii_rx_dv_r[5 : 0],gmii_rx_dv};
    end

    //在状态机处于空闲状态下，检测到连续7个8'h55后又检测到一个8'hd5后表示检测到帧头，此时将介绍数据的开始信号拉高，其余时间保持为低电平。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            start <= 1'b0;
        end
        else if(state_c == IDLE)begin
            start <= ({gmii_rx_dv_r,gmii_rx_dv} == 8'hFF) && ({gmii_rxd,gmii_rxd_r[0],gmii_rxd_r[1],gmii_rxd_r[2],gmii_rxd_r[3],gmii_rxd_r[4],
        end
    end

    //计数器，状态机在不同状态需要接收的数据个数不一样，使用一个可变进制的计数器。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//
            cnt <= 0;
        end
        else if(add_cnt)begin
            if(end_cnt)
                cnt <= 0;
```

```
            else
                cnt <= cnt + 1;
        end
        else begin
            cnt <= 0;
        end
    end
    //当状态机不在空闲状态或接收数据结束阶段时计数，计数到该状态需要接收数据个数时清零。
    assign add_cnt = (state_c != IDLE) && (state_c != RX_END) && gmii_rx_dv_r[0];
    assign end_cnt = add_cnt && cnt == cnt_num - 1;

    //状态机在不同状态，需要接收不同的数据个数，在接收以太网帧头时，需要接收14byte数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为20;
            cnt_num <= 16'd20;
        end
        else begin
            case(state_c)
                ETH_HEAD : cnt_num <= 16'd14;//以太网帧头长度位14字节。
                IP_HEAD  : cnt_num <= ip_head_byte_num;//IP帧头为20字节数据。
                UDP_HEAD : cnt_num <= 16'd8;//UDP帧头为8字节数据。
                UDP_DATA : cnt_num <= udp_data_length;//UDP数据段需要根据数据长度进行变化。
                CRC      : cnt_num <= 16'd4;//CRC校验为4字节数据。
                default: cnt_num <= 16'd20;
            endcase
        end
    end

    //接收目的MAC地址，需要判断这个包是不是发给开发板的，目的MAC地址是不是开发板的MAC地址或广播地址。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            des_mac_t <= 48'd0;
        end
        else if((state_c == ETH_HEAD) && add_cnt && cnt < 5'd6)begin
            des_mac_t <= {des_mac_t[39:0],gmii_rxd_r[0]};
        end
    end

    //判断接收的数据是否正确，以此来生成错误指示信号，判断状态机跳转。
    always@(posedge clk)begin

```

```
        if(rst_n==1'b0)begin//初始值为0;
            error_flag <= 1'b0;
        end
        else begin
            case(state_c)
                ETH_HEAD : begin
                    if(add_cnt)
                        if(cnt == 6)//判断接收的数据是不是发送给开发板或者广播数据。
                            error_flag <= ((des_mac_t != BOARD_MAC) && (des_mac_t != 48'HFF_FF_FF_FF_FF_FF));
                        else if(cnt ==12)//判断接收的数据是不是IP协议。
                            error_flag <= ({gmii_rxd_r[0],gmii_rxd} != ETH_TPYE);
                end
                IP_HEAD : begin
                    if(add_cnt)begin
                        if(cnt == 9)//如果当前接收的数据不是UDP协议，停止解析数据。
                            error_flag <= (gmii_rxd_r[0] != UDP_TYPE);
                        else if(cnt == 16'd18)//判断目的IP地址是否为开发板的IP地址。
                            error_flag <= ({des_ip,gmii_rxd_r[0],gmii_rxd} != BOARD_IP);
                    end
                end
                default: error_flag <= 1'b0;
            endcase
        end
    end

    //接收IP首部相关数据;
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            ip_head_byte_num <= 6'd20;
            ip_total_length <= 16'd28;
            des_ip <= 16'd0;
            udp_data_length <= 16'd0;
        end
        else if(state_c == IP_HEAD && add_cnt)begin
            case(cnt)
                16'd0 : ip_head_byte_num <= {gmii_rxd_r[0][3:0],2'd0};//接收IP首部的字节个数。
                16'd2 : ip_total_length[15:8] <= gmii_rxd_r[0];//接收IP报文总长度的高八位数据。
                16'd3 : ip_total_length[7:0] <= gmii_rxd_r[0];//接收IP报文总长度的低八位数据。
                16'd4 : udp_data_length <= ip_total_length - ip_head_byte_num - 8;//计算UDP报文数据段的长度，UDP帧头为8字节数据。
                16'd16,16'd17: des_ip <= {des_ip[7:0],gmii_rxd_r[0]};//接收目的IP地址。
```

```
                default: ;
            endcase
        end
    end

    //接收UDP首部相关数据;
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            des_port <= 16'd0;//目的端口号;
            source_port <= 16'd0;//源端口号;
        end
        else if(state_c == UDP_HEAD && add_cnt)begin
            case(cnt)
                16'd0,16'd1 : source_port <= {source_port[7:0],gmii_rxd_r[0]};//接收源端口号。
                16'd2,16'd3 : des_port <= {des_port[7:0],gmii_rxd_r[0]};//接收目的端口号。
                default: ;
            endcase
        end
    end

    //接收UDP的数据段，并输出使能信号。
    always@(posedge clk)begin
        rx_data <= (state_c == UDP_DATA) ? gmii_rxd_r[0] : rx_data;//在接收UDP数据阶段时，接收数据。
        rx_data_vld <= (state_c == UDP_DATA);//在接收数据阶段时，将FIFO写使能信号拉高，其余时间均拉低。
    end

    //生产CRC校验相关的数据和控制信号。
    always@(posedge clk)begin
        crc_data <= gmii_rxd_r[0];//将移位寄存器最低位存储的数据作为CRC输入模块的数据。
        crc_clr <= (state_c == IDLE);//当状态机处于空闲状态时，清除CRC校验模块计算。
        crc_en <= (state_c != IDLE) && (state_c != RX_END) && (state_c != CRC);//CRC校验使能信号。
    end

    //接收PC端发送来的CRC数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            des_crc <= 24'hff_ff_ff;
        end
        else if(add_cnt && state_c == CRC)begin//先接收的是低位数据;
            des_crc <= {gmii_rxd_r[0],des_crc[23:8]};

```

```
        end
    end

    //生成相应的输出数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
        rx_done <= 1'b0;
        data_byte_num <= 16'd0;
        end//如果CRC校验成功，把UDP协议接收完成信号拉高，把接收到UDP数据个数和数据段的校验和输出。
        else if(state_c == CRC && end_cnt && ({gmii_rxd_r[0],des_crc[23:0]} == crc_out))begin
            rx_done <= 1'b1;
            data_byte_num <= udp_data_length;
        end
        else begin
            rx_done <= 1'b0;
        end
    end
```

该模块的仿真结果如下图所示，仿真表示该模块接收到三帧UDP数据，UDP数据段长度分别为36字节、19字节、54字节。橙色信号是gmii_rxd_r[0]，紫红色信号是状态机现态，粉色信号是计数器和计数器的最大值，黄色信号是CRC校验模块的清零、使能、输入信号、计算结果。天蓝色信号是接收到的UDP数据段信号。

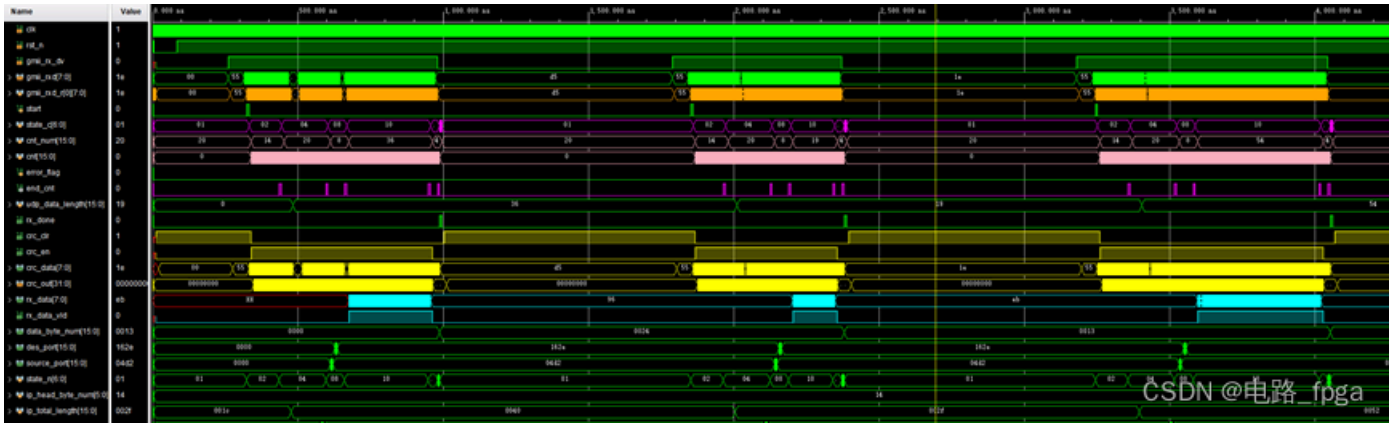

图6 UDP接收模块仿真

将UDP接收模块接收的第二帧数据放大，如下图所示，天蓝色信号将UDP数据段内容稳定输出。当CRC校验无误后，将rx_done信号拉高，表示接收完一帧UDP数据报文。

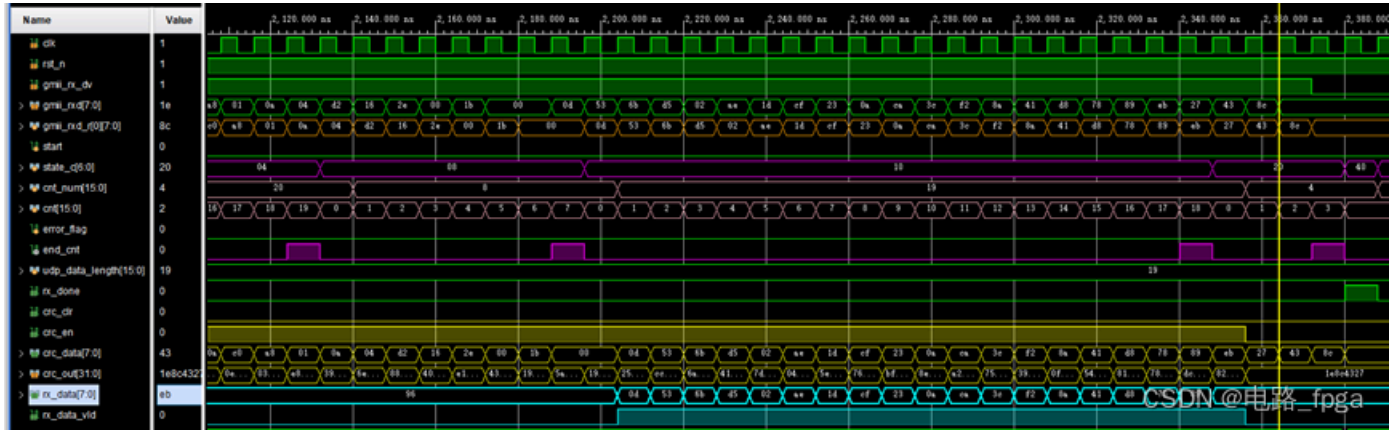

图7 UDP第二帧数据段放大

UDP接收模块的仿真就这么多了，需要详细了解的可以打开工程进行查看，工程中有对应的TestBench文件。

5、UDP发送模块

UDP发送模块同样可以采用状态机和计数器作为主体架构实现，状态机对应的状态转换图如下所示。该模块的实现相对于ICMP发送模块会简单一点，不需要计算UDP校验码，只需要计算IP首部校验码即可。最后需要注意如果UDP数据段不足18个字节数据，需要补零填充到18字节数据。

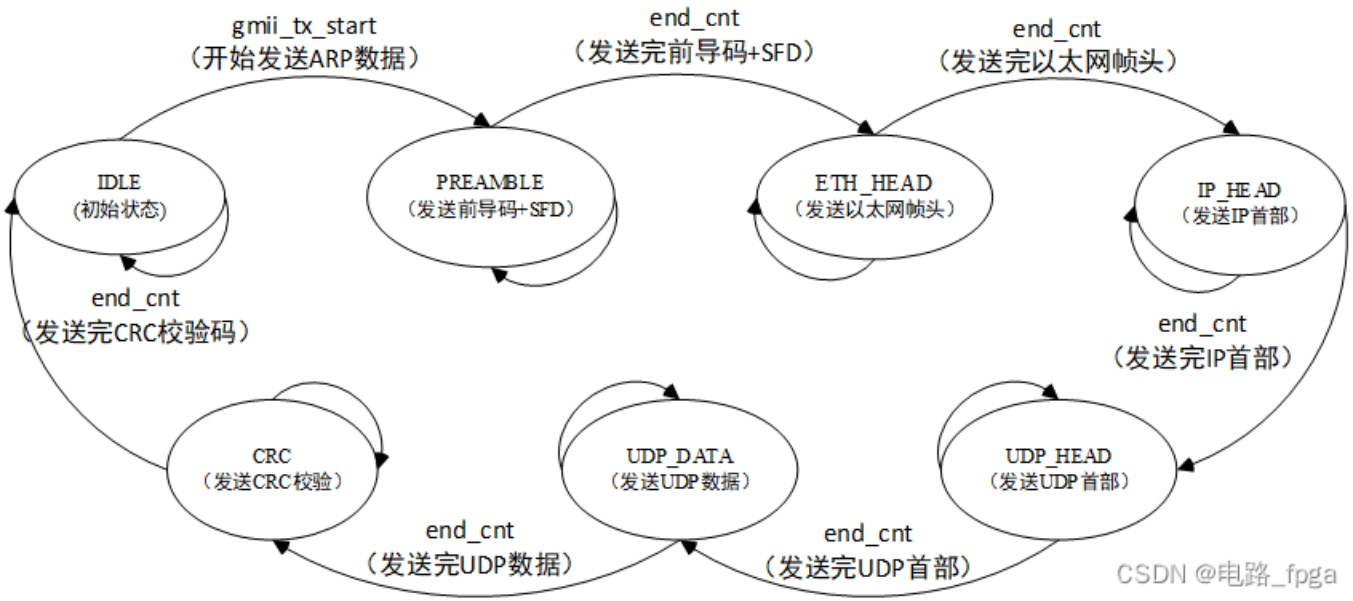

图8 状态转换图

该模块的核心代码如下所示，完整代码在工程中查看。

```
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            ip_head[0] <= 32'd0;
            ip_head[1] <= 32'd0;
            ip_head[2] <= 32'd0;
            ip_head[3] <= 32'd0;
            ip_head[4] <= 32'd0;
            udp_head[0] <= {BOARD_PORT,DES_PORT};
            udp_head[1] <= 32'd0;
            ip_head_check <= 32'd0;
            des_ip_r <= DES_IP;
            des_mac_r <= DES_MAC;
            tx_byte_num_r <= MIN_DATA_NUM;
            ip_total_num <= MIN_DATA_NUM + 28;
        end
        //在状态机空闲状态下，上游发送使能信号时，将目的MAC地址和目的IP以及UDP需要发送的数据个数进行暂存。
        else if(state_c == IDLE && udp_tx_start)begin
            udp_head[0] <= {BOARD_PORT,DES_PORT};//16位源端口和目的端口地址。

```

```
            udp_head[1][31:16] <= (((tx_byte_num >= MIN_DATA_NUM) ? tx_byte_num : MIN_DATA_NUM) + 8);//计算UDP需要发送报文的长度。
            tx_byte_num_r <= tx_byte_num;
            //如果需要发送的数据多余最小长度要求，则发送的总数居等于需要发送的数据加上UDP和IP帧头数据。
            ip_total_num <= (((tx_byte_num >= MIN_DATA_NUM) ? tx_byte_num : MIN_DATA_NUM) + 28);
            if((des_mac != 48'd0) && (des_ip != 48'd0))begin//当接收到目的MAC地址和目的IP地址时更新。
                des_ip_r <= des_ip;
                des_mac_r <= des_mac;
            end
        end
        //在发送以太网帧头时，就开始计算IP帧头和UDP的校验码，并将计算结果存储，便于后续直接发送。
        else if(state_c == ETH_HEAD && add_cnt)begin
            case (cnt)
                16'd0 : begin//初始化需要发送的IP头部数据。
                    ip_head[0] <= {IP_VERSION,IP_HEAD_LEN,8'h00,ip_total_num[15:0]};//依次表示IP版本号，IP头部长度，IP服务类型，IP包的总长度。
                    ip_head[2] <= {8'h80,8'd17,16'd0};//分别表示生存时间，协议类型，1表示UDP，2表示IGMP，6表示TCP，17表示UDP协议，低16位校验和先默认为0；
                    ip_head[3] <= BOARD_IP;//源IP地址。
                    ip_head[4] <= des_ip_r;//目的IP地址。
                end
                16'd1 : begin//开始计算IP头部校验和数据，并且将计算结果存储到对应位置。
                    ip_head_check <= ip_head[0][31 : 16] + ip_head[0][15 : 0];
                end
                16'd2 : begin
                    ip_head_check <= ip_head_check + ip_head[1][31 : 16];
                end
                16'd3 : begin
                    ip_head_check <= ip_head_check + ip_head[1][15 : 0];
                end
                16'd4 : begin
                    ip_head_check <= ip_head_check + ip_head[2][31 : 16];
                end
                16'd5 : begin
                    ip_head_check <= ip_head_check + ip_head[3][31 : 16];
                end
                16'd6 : begin
                    ip_head_check <= ip_head_check + ip_head[3][15 : 0];
                end
                16'd7 : begin
                    ip_head_check <= ip_head_check + ip_head[4][31 : 16];
                end
                16'd8 : begin
```

```
                ip_head_check <= ip_head_check + ip_head[4][15 : 0];
            end
            16'd9,16'd10 : begin
                ip_head_check <= ip_head_check[31 : 16] + ip_head_check[15 : 0];
            end
            16'd11 : begin
                ip_head[2][15:0] <= ~ip_head_check[15 : 0];
                ip_head_check <= 32'd0;//校验和清零，用于下次计算。
            end
            default: begin
                ip_head_check <= 32'd0;//校验和清零，用于下次计算。
            end
        endcase
    end
    else if(state_c == IP_HEAD && end_cnt)
        ip_head[1] <= {ip_head[1][31:16]+1,16'h4000};//高16位表示标识，每次发送数据后会加1，低16位表示不分片。
end

//The first section: synchronous timing always module, formatted to describe the transfer of the secondary register to the live register
always@(posedge clk)begin
    if(!rst_n)begin
        state_c <= IDLE;
    end
    else begin
        state_c <= state_n;
    end
end

//The second paragraph: The combinational logic always module describes the state transition condition judgment.
always@(*)begin
    case(state_c)
        IDLE:begin
            if(udp_tx_start)begin//在空闲状态接收到上游发出的使能信号；
                state_n = PREAMBLE;
            end
            else begin
                state_n = state_c;
            end
        end
        PREAMBLE:begin
```

```
            if(end_cnt)begin//发送完前导码和SFD；
                state_n = ETH_HEAD;
            end
            else begin
                state_n = state_c;
            end
        end
        ETH_HEAD:begin
            if(end_cnt)begin//发送完以太网帧头数据；
                state_n = IP_HEAD;
            end
            else begin
                state_n = state_c;
            end
        end
        IP_HEAD:begin
            if(end_cnt)begin//发送完IP帧头数据；
                state_n = UDP_HEAD;
            end
            else begin
                state_n = state_c;
            end
        end
        UDP_HEAD:begin
            if(end_cnt)begin//发送完UDP帧头数据；
                state_n = UDP_DATA;
            end
            else begin
                state_n = state_c;
            end
        end
        UDP_DATA:begin
            if(end_cnt)begin//发送完udp协议数据；
                state_n = CRC;
            end
            else begin
                state_n = state_c;
            end
        end
        CRC:begin
```

```
                if(end_cnt)begin//发送完CRC校验码；
                    state_n = IDLE;
                end
                else begin
                    state_n = state_c;
                end
            end
            default:begin
                state_n = IDLE;
            end
        endcase
    end

    //计数器，用于记录每个状态机每个状态需要发送的数据个数，每个时钟周期发送1byte数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//
            cnt <= 0;
        end
        else if(add_cnt)begin
            if(end_cnt)
                cnt <= 0;
            else
                cnt <= cnt + 1;
        end
    end

    assign add_cnt = (state_c != IDLE);//状态机不在空闲状态时计数。
    assign end_cnt = add_cnt && cnt == cnt_num - 1;//状态机对应状态发送完对应个数的数据。

    //状态机在每个状态需要发送的数据个数。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为20;
            cnt_num <= 16'd20;
        end
        else begin
            case (state_c)
                PREAMBLE : cnt_num <= 16'd8;//发送7个前导码和1个8'hd5。
                ETH_HEAD : cnt_num <= 16'd14;//发送14字节的以太网帧头数据。
                IP_HEAD : cnt_num <= 16'd20;//发送20个字节是IP帧头数据。
                UDP_HEAD : cnt_num <= 16'd8;//发送8字节的UDP帧头数据。

```

```
            UDP_DATA : if(tx_byte_num_r >= MIN_DATA_NUM)//如果需要发送的数据多余以太网最短数据要求，则发送指定个数数据。
                            cnt_num <= tx_byte_num_r;
                        else//否则需要将指定个数数据发送完成，不足长度补零，达到最短的以太网帧要求。
                            cnt_num <= MIN_DATA_NUM;
            CRC : cnt_num <= 6'd5;//CRC在时钟1时才开始发送数据，这是因为CRC计算模块输出的数据会延后一个时钟周期。
            default: cnt_num <= 6'd20;
        endcase
    end
end

//根据状态机和计数器的值产生输出数据，只不过这不是真正的输出，还需要延迟一个时钟周期。
always@(posedge clk)begin
    if(rst_n==1'b0)begin//初始值为0;
        crc_data <= 8'd0;
    end
    else if(add_cnt)begin
        case (state_c)
            PREAMBLE : if(end_cnt)
                            crc_data <= 8'hd5;//发送1字节SFD编码；
                        else
                            crc_data <= 8'h55;//发送7字节前导码；
            ETH_HEAD : if(cnt < 6)
                            crc_data <= des_mac_r[47 - 8*cnt -: 8];//发送目的MAC地址，先发高字节；
                        else if(cnt < 12)
                            crc_data <= BOARD_MAC[47 - 8*(cnt-6) -: 8];//发送源MAC地址，先发高字节；
                        else
                            crc_data <= ETH_TYPE[15 - 8*(cnt-12) -: 8];//发送源以太网协议类型，先发高字节；
            IP_HEAD : if(cnt < 4)//发送IP帧头。
                            crc_data <= ip_head[0][31 - 8*cnt -: 8];
                        else if(cnt < 8)
                            crc_data <= ip_head[1][31 - 8*(cnt-4) -: 8];
                        else if(cnt < 12)
                            crc_data <= ip_head[2][31 - 8*(cnt-8) -: 8];
                        else if(cnt < 16)
                            crc_data <= ip_head[3][31 - 8*(cnt-12) -: 8];
                        else
                            crc_data <= ip_head[4][31 - 8*(cnt-16) -: 8];
            UDP_HEAD : if(cnt < 4)//发送UDP帧头数据。
                            crc_data <= udp_head[0][31 - 8*cnt -: 8];
                        else

```

```
                                crc_data <= udp_head[1][31 - 8*(cnt-4) -: 8];
                UDP_DATA : if(tx_byte_num_r >= MIN_DATA_NUM)//需要判断发送的数据是否满足以太网最小数据要求。
                                crc_data <= tx_data;//如果满足最小要求，将需要配发送的数据输出。
                            else if(cnt < tx_byte_num_r)//不满足最小要求时，先将需要发送的数据发送完。
                                crc_data <= tx_data;//将需要发送的数据输出即可。
                            else//剩余数据补充0.
                                crc_data <= 8'd0;
                default : ;
            endcase
        end
    end

    //生成数据请求输入信号，外部输入数据延后该信号一个时钟周期，所以需要提前产生一个时钟周期产生请求信号；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            tx_data_req <= 1'b0;
        end
        //在数据段的前三个时钟周期拉高；
        else if(state_c == UDP_HEAD && add_cnt && (cnt == cnt_num - 2))begin
            tx_data_req <= 1'b1;
        end//在ICMP或者UDP数据段时，当发送完数据的前三个时钟拉低；
        else if(state_c == UDP_DATA && add_cnt && (cnt == cnt_num - 2))begin
                tx_data_req <= 1'b0;
         end
    end

    //生成一个crc_data指示信号，用于生成gmii_txd信号。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            gmii_tx_en_r <= 1'b0;
        end
        else if(state_c == CRC)begin
            gmii_tx_en_r <= 1'b0;
        end
        else if(state_c == PREAMBLE)begin
            gmii_tx_en_r <= 1'b1;
        end
    end

    //生产CRC校验模块使能信号，初始值为0，当开始输出以太网帧头时拉高，当ARP和以太网帧头数据全部输出后拉低。

```

```
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            crc_en <= 1'b0;
        end
        else if(state_c == CRC)begin//当ARP和以太网帧头数据全部输出后拉低.
            crc_en <= 1'b0;
        end//当开始输出以太网帧头时拉高。
        else if(state_c == ETH_HEAD && add_cnt)begin
            crc_en <= 1'b1;
        end
    end

    //生产CRC校验模块清零信号，状态机处于空闲时清零。
    always@(posedge clk)begin
        crc_clr <= (state_c == IDLE);
    end

    //生成gmii_txd信号，默认输出0。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            gmii_txd <= 8'd0;
        end//在输出CRC状态时，输出CRC校验码，先发送低位数据。
        else if(state_c == CRC && add_cnt && cnt>0)begin
            gmii_txd <= crc_out[8*cnt-1 -: 8];
        end//其余时间如果crc_data有效，则输出对应数据。
        else if(gmii_tx_en_r)begin
            gmii_txd <= crc_data;
        end
    end

    //生成gmii_txd有效指示信号。
    always@(posedge clk)begin
        gmii_tx_en <= gmii_tx_en_r || (state_c == CRC);
    end

    //模块忙闲指示信号，当接收到上游模块的使能信号或者状态机不处于空闲状态时拉低，其余时间拉高。
    //该信号必须使用组合逻辑产生，上游模块必须使用时序逻辑检测该信号。
    always@(*)begin
        if(udp_tx_start || state_c != IDLE)
            rdy = 1'b0;
```

```
        else
            rdy = 1'b1;
    end
```

以太网发送数据模块仿真结果如下所示，发送了三帧数据。

图9 发送数据模块仿真

该模块需要注意什么？其实需要考虑的是CRC校验模块输出的数据会滞后输入一个时钟周期，为了实现数据对齐，需要把crc_data延迟一个时钟周期得到gmii_txd，仿真结果如下所示。

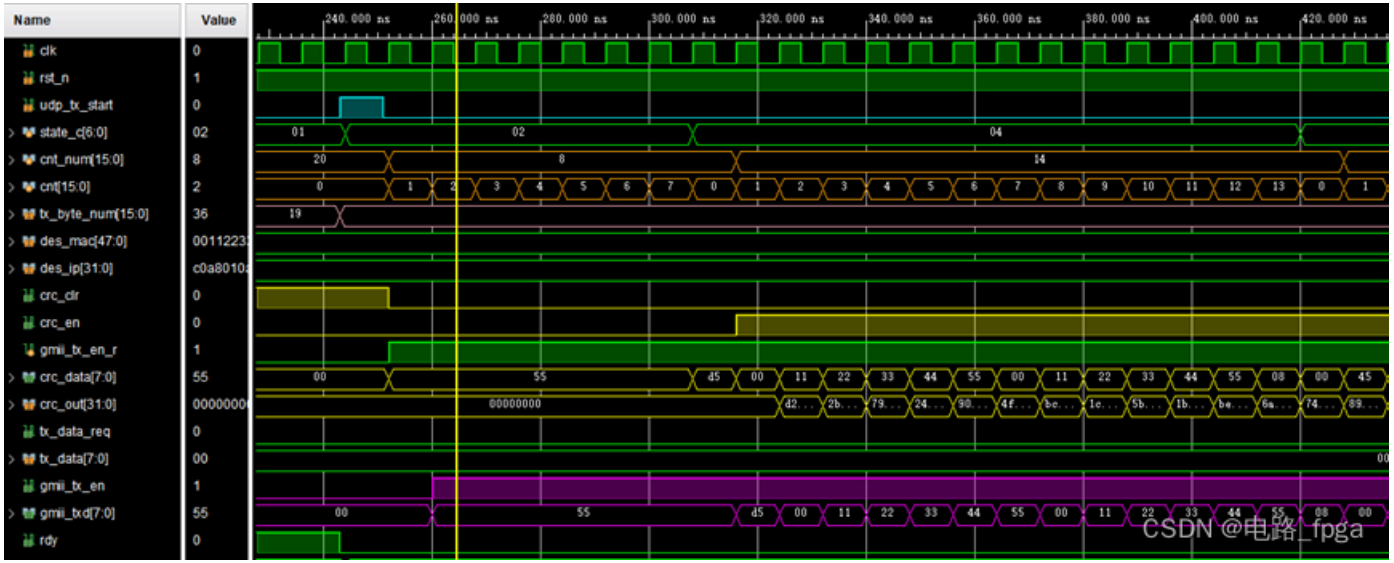

图10 开始发送数据

前文在实现ICMP发送模块的时候，FIFO使用了超前模式，读使能与读数据对齐，但是UDP发送的数据未必来自FIFO，更多情况可能是数据会滞后请求信号一个时钟。所以本文把FIFO换成常规模式，输出的数据会滞后读使能一个时钟周期。

那么就需要提前一个时钟周期产生请求信号，对应的仿真结果如下所示，在状态机发送UDP首部最后一个字节数据时，将请求信号req拉高。

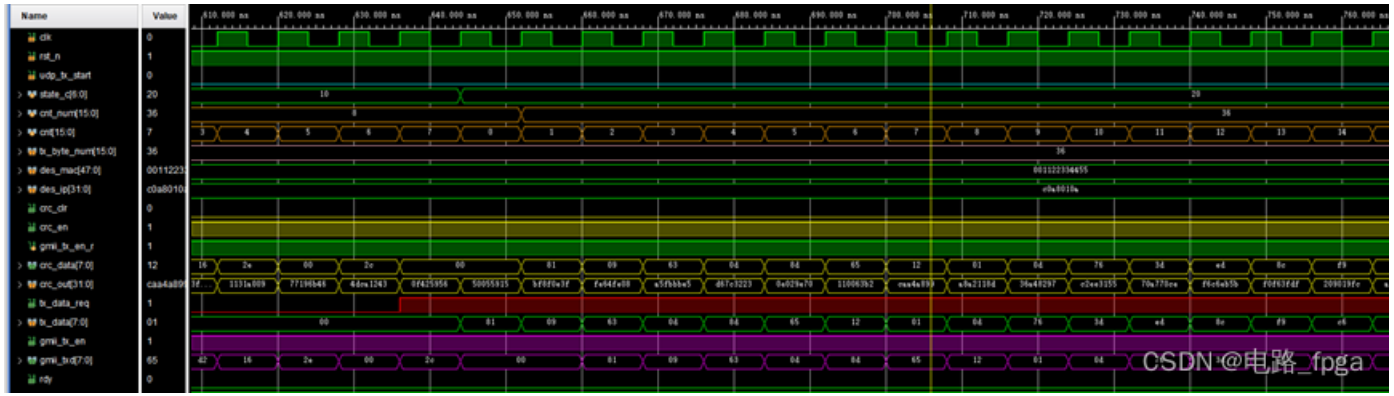

图11 数据请求仿真

该模块的仿真到此结束，具体的CRC仿真还有IP首部、UDP首部这些细节就不再赘述了，与前文的ARP和ICMP道里差不多，需要详细了解的可以在公众号获取工程文件自行查看。

6、ARP、ICMP、UDP控制模块

本文实现UDP的回环，为了不去手动绑定开发板的IP地址和MAC地址，所以需要ARP模块，还要能够判断以太网链路是否畅通，就需要使用ICMP协议。开发板只使用一个网口，但是ARP、ICMP、UDP均会输出gmii_txd信号，所以就需要一个控制模块对三个模块的输出进行仲裁。

该模块接收到ARP请求时，就会使能ARP发送模块，向PC端发出ARP应答指令。当开发板上某个按键被按下后，也会向PC端发出ARP请求指令。当接收到PC端发出的回显请求指令时，该模块使能ICMP发送模块向PC端发送回显应答指令。最后当接收到UDP数据报文后，将接收的数据通过UDP发送模块传送给PC，实现数据回环。

该模块的核心代码如下所示，由于篇幅原因，需要代码可以从公众号的工程获取。

```
    //ARP发送数据报的类型。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            arp_tx_type <= 1'b0;
        end
        else if(arp_rx_done && ~arp_rx_type)begin//接收到PC的ARP请求时，应该回发应答信号。
            arp_tx_type <= 1'b1;
        end
        else if(key_in || (arp_rx_done && arp_rx_type))begin//其余时间发送请求指令。
            arp_tx_type <= 1'b0;
        end
    end

    //接收到ARP请求数据报文时，将接收到的目的MAC和IP地址输出。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            arp_tx_start <= 1'b0;
            des_mac <= 48'd0;
            des_ip <= 32'd0;
        end
        else if(arp_rx_done && ~arp_rx_type)begin
            arp_tx_start <= 1'b1;
            des_mac <= src_mac;
            des_ip <= src_ip;
        end
        else if(key_in)begin
            arp_tx_start <= 1'b1;
        end
        else begin
            arp_tx_start <= 1'b0;
        end
    end
```

```
    //接收到ICMP请求数据报文时，发送应答数据报。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            icmp_tx_start <= 1'b0;
            icmp_tx_byte_num <= 16'd0;
        end
        else if(icmp_rx_done)begin
            icmp_tx_start <= 1'b1;
            icmp_tx_byte_num <= icmp_rx_byte_num;
        end
        else begin
            icmp_tx_start <= 1'b0;
        end
    end

    //接收到UDP数据报文后，将数据发送回源端。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            udp_tx_start <= 1'b0;
            udp_tx_byte_num <= 16'd0;
        end
        else if(udp_rx_done)begin
            udp_tx_start <= 1'b1;
            udp_tx_byte_num <= udp_rx_byte_num;
        end
        else begin
            udp_tx_start <= 1'b0;
        end
    end

    //对三个模块需要发送的数据进行整合。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            gmii_tx_en <= 1'b0;
            gmii_txd <= 8'd0;
        end//如果ARP发送模块输出有效数据，且ICMP发送模块和UDP发送模块都处于空闲状态，则将ARP相关数据输出。
        else if(arp_gmii_tx_en && icmp_tx_rdy && udp_tx_rdy)begin
            gmii_tx_en <= arp_gmii_tx_en;
            gmii_txd <= arp_gmii_txd;
```

```
74          end//如果ICMP发送模块输出有效数据且ARP发送模块和UDP发送模块均处于空闲，则将ICMP相关数据输出。
75          else if(icmp_gmii_tx_en && arp_tx_rdy && udp_tx_rdy)begin
76              gmii_tx_en <= icmp_gmii_tx_en;
77              gmii_txd <= icmp_gmii_txd;
78          end//如果udP发送模块输出有效数据且ARP发送模块和idmp发送模块均处于空闲，则将ICMP相关数据输出。
79          else if(udp_gmii_tx_en && arp_tx_rdy && icmp_tx_rdy)begin
80              gmii_tx_en <= udp_gmii_tx_en;
81              gmii_txd <= udp_gmii_txd;
82          end
83          else begin
84              gmii_tx_en <= 1'b0;
85          end
    end
```

该模块的思路比较简单，就不再仿真，后续直接上板即可。

7、顶层模块

顶层模块主要将ARP、ICMP、UDP、RGMII与GMII转换模块、按键消抖模块、暂存UDP数据的FIFO模块、锁相环模块的输入输出端口进行连线。

顶层对应的框图如下所示，由于比较复杂，直接采用vivado的RTL视图。

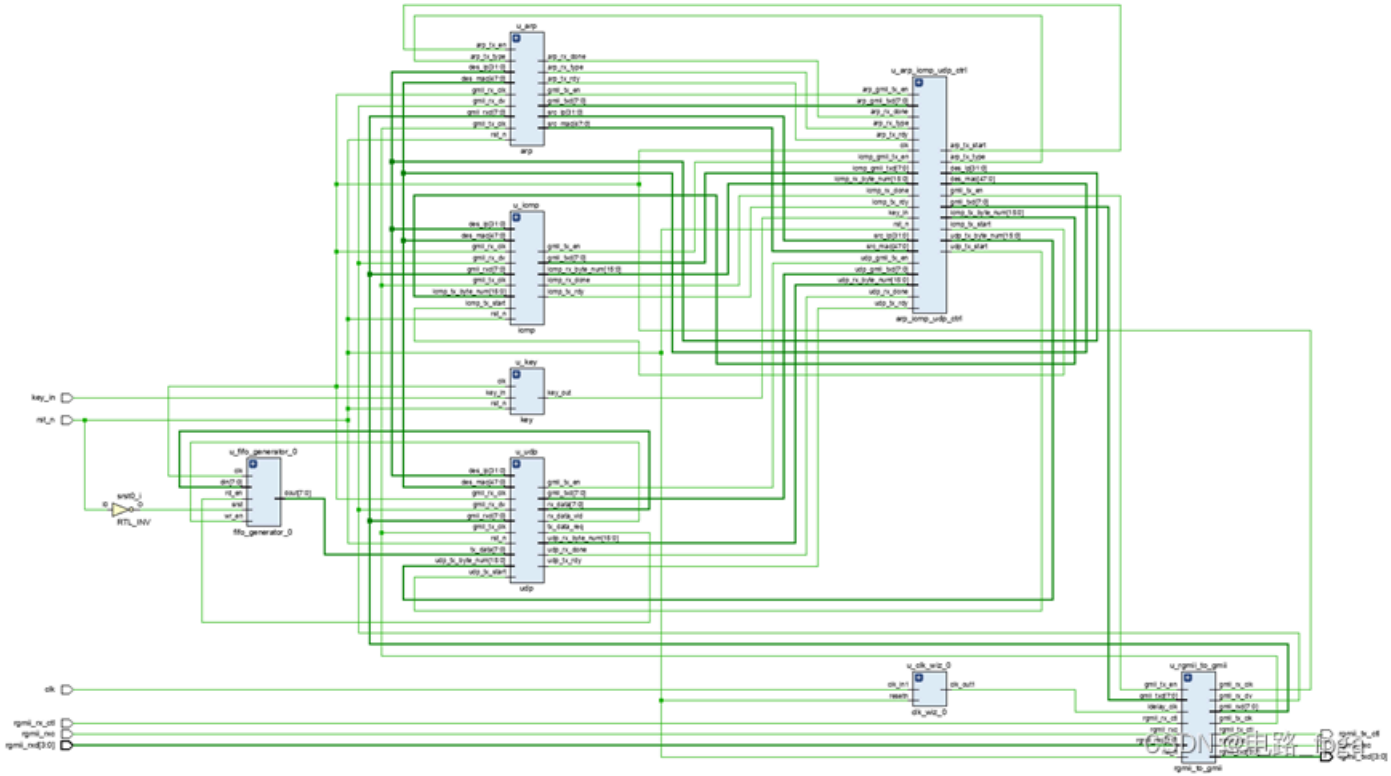

图12 顶层模块框图

该模块暂存UDP数据和ICMP数据的FIFO设置如下所示，使用一般模式即可，对应的需要提前产生数据请求输入信号。

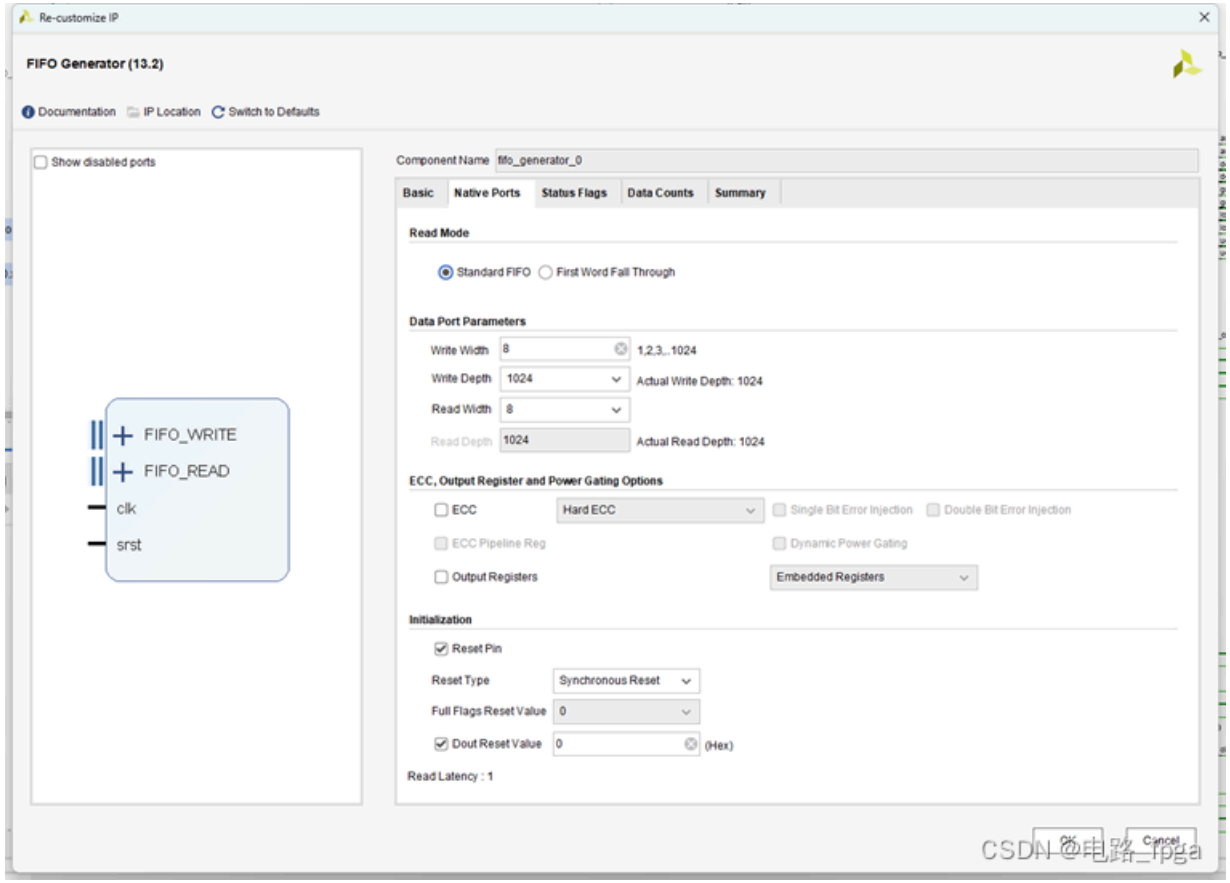

图13 FIFO配置

该模块对应核心代码如下所示：

```
    //例化锁相环，输出200MHZ时钟，作为IDELAYECTRL的参考时钟。
    clk_wiz_0 u_clk_wiz_0 (
        .clk_out1   ( idelay_clk),//output clk_out1;
        .resetn     ( rst_n     ),//input resetn;
        .clk_in1    ( clk       ) //input clk_in1;
    );

    //例化按键消抖模块。
    key #(
        .TIME_20MS  ( TIME_20MS ),//按键抖动持续的最长时间，默认最长持续时间为20ms。
        .TIME_CLK   ( TIME_CLK  ) //系统时钟周期，默认8ns。
```

```
    )
    u_key (
        .clk        ( gmii_rx_clk   ),//系统时钟，125MHz。
        .rst_n      ( rst_n         ),//系统复位，低电平有效。
        .key_in     ( key_in        ),//待输入的按键输入信号,默认低电平有效；
        .key_out    ( key_out       ) //按键消抖后输出信号，当按键按下一次时，输出一个时钟宽度的高电平；
    );

    //例化ARP和ICMP的控制模块
    arp_icmp_udp_ctrl  u_arp_icmp_udp_ctrl (
        .clk                ( gmii_rx_clk       ),//输入时钟；
        .rst_n              ( rst_n             ),//复位信号，低电平有效；
        .key_in             ( key_out           ),//按键按下，高电平有效；
        .des_mac            ( des_mac           ),//发送的目标MAC地址。
        .des_ip             ( des_ip            ),//发送的目标IP地址。
        //ARP
        .arp_rx_done        ( arp_rx_done       ),//ARP接收完成信号；
        .arp_rx_type        ( arp_rx_type       ),//ARP接收类型 0:请求  1:应答；
        .src_mac            ( src_mac           ),//ARP接收到目的MAC地址。
        .src_ip             ( src_ip            ),//ARP接收到目的IP地址。
        .arp_tx_rdy         ( arp_tx_rdy        ),//ARP发送模块忙闲指示信号。
        .arp_tx_start       ( arp_tx_start      ),//ARP发送使能信号；
        .arp_tx_type        ( arp_tx_type       ),//ARP发送类型 0:请求  1:应答；
        .arp_gmii_tx_en     ( arp_gmii_tx_en    ),
        .arp_gmii_txd       ( arp_gmii_txd      ),
        //ICMP
        .icmp_rx_done       ( icmp_rx_done      ),//ICMP接收完成信号；
        .icmp_rx_byte_num   ( icmp_rx_byte_num  ),//以太网接收的有效字节数 单位:byte。
        .icmp_tx_rdy        ( icmp_tx_rdy       ),//ICMP发送模块忙闲指示信号。
        .icmp_gmii_tx_en    ( icmp_gmii_tx_en   ),
        .icmp_gmii_txd      ( icmp_gmii_txd     ),
        .icmp_tx_start      ( icmp_tx_start     ),//ICMP发送使能信号；
        .icmp_tx_byte_num   ( icmp_tx_byte_num  ),//以太网发送的有效字节数 单位:byte。
        //udp
        .udp_rx_done        ( udp_rx_done       ),//UDP接收完成信号；
        .udp_rx_byte_num    ( udp_rx_byte_num   ),//以太网接收的有效字节数 单位:byte。
        .udp_tx_rdy         ( udp_tx_rdy        ),//UDP发送模块忙闲指示信号。
        .udp_gmii_tx_en     ( udp_gmii_tx_en    ),
        .udp_gmii_txd       ( udp_gmii_txd      ),
        .udp_tx_start       ( udp_tx_start      ),//UDP发送使能信号；

```

```
        .udp_tx_byte_num    ( udp_tx_byte_num   ),//以太网发送的有效字节数 单位:byte。

        .gmii_tx_en         ( gmii_tx_en        ),
        .gmii_txd           ( gmii_txd          )
    );

    //例化ARP模块;
    arp #(
        .BOARD_MAC      ( BOARD_MAC     ),//开发板MAC地址 00-11-22-33-44-55;
        .BOARD_IP       ( BOARD_IP      ),//开发板IP地址 192.168.1.10;
        .DES_MAC        ( DES_MAC       ),//目的MAC地址 ff_ff_ff_ff_ff_ff;
        .DES_IP         ( DES_IP        ),//目的IP地址 192.168.1.102;
        .ETH_TYPE       ( 16'h0806      ) //以太网帧类型，16'h0806表示ARP协议，16'h0800表示IP协议;
    )
    u_arp (
        .rst_n          ( rst_n             ),//复位信号，低电平有效。
        .gmii_rx_clk    ( gmii_rx_clk       ),//GMII接收数据时钟。
        .gmii_rx_dv     ( gmii_rx_dv        ),//GMII输入数据有效信号。
        .gmii_rxd       ( gmii_rxd          ),//GMII输入数据。
        .gmii_tx_clk    ( gmii_tx_clk       ),//GMII发送数据时钟。
        .arp_tx_en      ( arp_tx_start      ),//ARP发送使能信号。
        .arp_tx_type    ( arp_tx_type       ),//ARP发送类型 0:请求  1:应答。
        .des_mac        ( des_mac           ),//发送的目标MAC地址。
        .des_ip         ( des_ip            ),//发送的目标IP地址。
        .gmii_tx_en     ( arp_gmii_tx_en    ),//GMII输出数据有效信号。
        .gmii_txd       ( arp_gmii_txd      ),//GMII输出数据。
        .arp_rx_done    ( arp_rx_done       ),//ARP接收完成信号。
        .arp_rx_type    ( arp_rx_type       ),//ARP接收类型 0:请求  1:应答。
        .src_mac        ( src_mac           ),//接收到目的MAC地址。
        .src_ip         ( src_ip            ),//接收到目的IP地址。
        .arp_tx_rdy     ( arp_tx_rdy        ) //ARP发送模块忙闲指示指示信号，高电平表示该模块空闲。
    );

    //例化ICMP模块。
    icmp #(
        .BOARD_MAC  ( BOARD_MAC ),//开发板MAC地址 00-11-22-33-44-55;
        .BOARD_IP   ( BOARD_IP  ),//开发板IP地址 192.168.1.10;
        .DES_MAC    ( DES_MAC   ),//目的MAC地址 ff_ff_ff_ff_ff_ff;
        .DES_IP     ( DES_IP    ),//目的IP地址 192.168.1.102;
        .ETH_TYPE   ( 16'h0800  ) //以太网帧类型，16'h0806表示ARP协议，16'h0800表示IP协议;

```

```
    )
    u_icmp (
        .rst_n              ( rst_n             ),//复位信号，低电平有效。
        .gmii_rx_clk        ( gmii_rx_clk       ),//GMII接收数据时钟。
        .gmii_rx_dv         ( gmii_rx_dv        ),//GMII输入数据有效信号。
        .gmii_rxd           ( gmii_rxd          ),//GMII输入数据。
        .gmii_tx_clk        ( gmii_tx_clk       ),//GMII发送数据时钟。
        .gmii_tx_en         ( icmp_gmii_tx_en   ),//GMII输出数据有效信号。
        .gmii_txd           ( icmp_gmii_txd     ),//GMII输出数据。
        .icmp_tx_start      ( icmp_tx_start     ),//以太网开始发送信号.
        .icmp_tx_byte_num   ( icmp_tx_byte_num  ),//以太网发送的有效字节数 单位:byte。
        .des_mac            ( des_mac           ),//发送的目标MAC地址。
        .des_ip             ( des_ip            ),//发送的目标IP地址。
        .icmp_rx_done       ( icmp_rx_done      ),//ICMP接收完成信号。
        .icmp_rx_byte_num   ( icmp_rx_byte_num  ),//以太网接收的有效字节数 单位:byte。
        .icmp_tx_rdy        ( icmp_tx_rdy       ) //ICMP发送模块忙闲指示指示信号，高电平表示该模块空闲。
    );

    //例化UDP模块。
    udp #(
        .BOARD_MAC  ( BOARD_MAC ),//开发板MAC地址 00-11-22-33-44-55;
        .BOARD_IP   ( BOARD_IP  ),//开发板IP地址 192.168.1.10;
        .DES_MAC    ( DES_MAC   ),//目的MAC地址 ff_ff_ff_ff_ff_ff;
        .DES_IP     ( DES_IP    ),//目的IP地址 192.168.1.102;
        .BOARD_PORT ( BOARD_PORT),//板子的UDP端口号;
        .DES_PORT   ( DES_PORT  ),//源端口号;
        .ETH_TYPE   ( 16'h0800  ) //以太网帧类型，16'h0806表示ARP协议，16'h0800表示IP协议;
    )
    u_udp (
        .rst_n              ( rst_n             ),//复位信号，低电平有效。
        .gmii_rx_clk        ( gmii_rx_clk       ),//GMII接收数据时钟。
        .gmii_rx_dv         ( gmii_rx_dv        ),//GMII输入数据有效信号。
        .gmii_rxd           ( gmii_rxd          ),//GMII输入数据。
        .gmii_tx_clk        ( gmii_tx_clk       ),//GMII发送数据时钟。
        .gmii_tx_en         ( udp_gmii_tx_en    ),//GMII输出数据有效信号。
        .gmii_txd           ( udp_gmii_txd      ),//GMII输出数据。

        .udp_tx_start       ( udp_tx_start      ),//以太网开始发送信号.
        .udp_tx_byte_num    ( udp_tx_byte_num   ),//以太网发送的有效字节数 单位:byte。
        .des_mac            ( des_mac           ),//发送的目标MAC地址。

```

```
    .des_ip             ( des_ip            ),//发送的目标IP地址。
    .udp_rx_done        ( udp_rx_done       ),//UDP接收完成信号。
    .udp_rx_byte_num    ( udp_rx_byte_num   ),//以太网接收的有效字节数 单位:byte。
    .udp_tx_rdy         ( udp_tx_rdy        ),//UDP发送模块忙闲指示指示信号，高电平表示该模块空闲。
    .rx_data            ( udp_rx_data       ),
    .rx_data_vld        ( udp_rx_data_vld   ),
    .tx_data            ( udp_tx_data       ),
    .tx_data_req        ( udp_tx_data_req   )
);

//例化FIFO;
fifo_generator_0 u_fifo_generator_0 (
    .clk    ( gmii_rx_clk       ),//input wire clk
    .srst   ( ~rst_n            ),//input wire srst
    .din    ( udp_rx_data       ),//input wire [7 : 0] din
    .wr_en  ( udp_rx_data_vld   ),//input wire wr_en
    .rd_en  ( udp_tx_data_req   ),//input wire rd_en
    .dout   ( udp_tx_data       ),//output wire [7 : 0] dout
    .full   (                   ),//output wire full
    .empty  (                   ) //output wire empty
);

//例化gmii转RGMII模块。
rgmii_to_gmii u_rgmii_to_gmii (
    .idelay_clk             ( idelay_clk    ),//IDELAY时钟;
    .rst_n                  ( rst_n         ),
    .gmii_tx_en             ( gmii_tx_en    ),//GMII发送数据使能信号;
    .gmii_txd               ( gmii_txd      ),//GMII发送数据;
    .gmii_rx_clk            ( gmii_rx_clk   ),//GMII接收时钟;
    .gmii_rx_dv             ( gmii_rx_dv    ),//GMII接收数据有效信号;
    .gmii_rxd               ( gmii_rxd      ),//GMII接收数据;
    .gmii_tx_clk            ( gmii_tx_clk   ),//GMII发送时钟;

    .rgmii_rxc              ( rgmii_rxc     ),//RGMII接收时钟;
    .rgmii_rx_ctl           ( rgmii_rx_ctl  ),//RGMII接收数据控制信号;
    .rgmii_rxd              ( rgmii_rxd     ),//RGMII接收数据;
    .rgmii_txc              ( rgmii_txc     ),//RGMII发送时钟;
    .rgmii_tx_ctl           ( rgmii_tx_ctl  ),//RGMII发送数据控制信号;
    .rgmii_txd              ( rgmii_txd     ) //RGMII发送数据;
);
```

8、上板测试

将顶层模块中的ILA注释取消，然后将程序综合、实现，最后下载到开发板中进行测试。打开电脑的控制面板->网络和Internet->网络连接，鼠标右击以太网，双击Internet协议版本4，进行如下设置，与代码顶层模块设置的目的IP一致，具体步骤可以查看前文。

图14 电脑IP设置

然后把wirrshark和网络调试助手打开，如下所示：

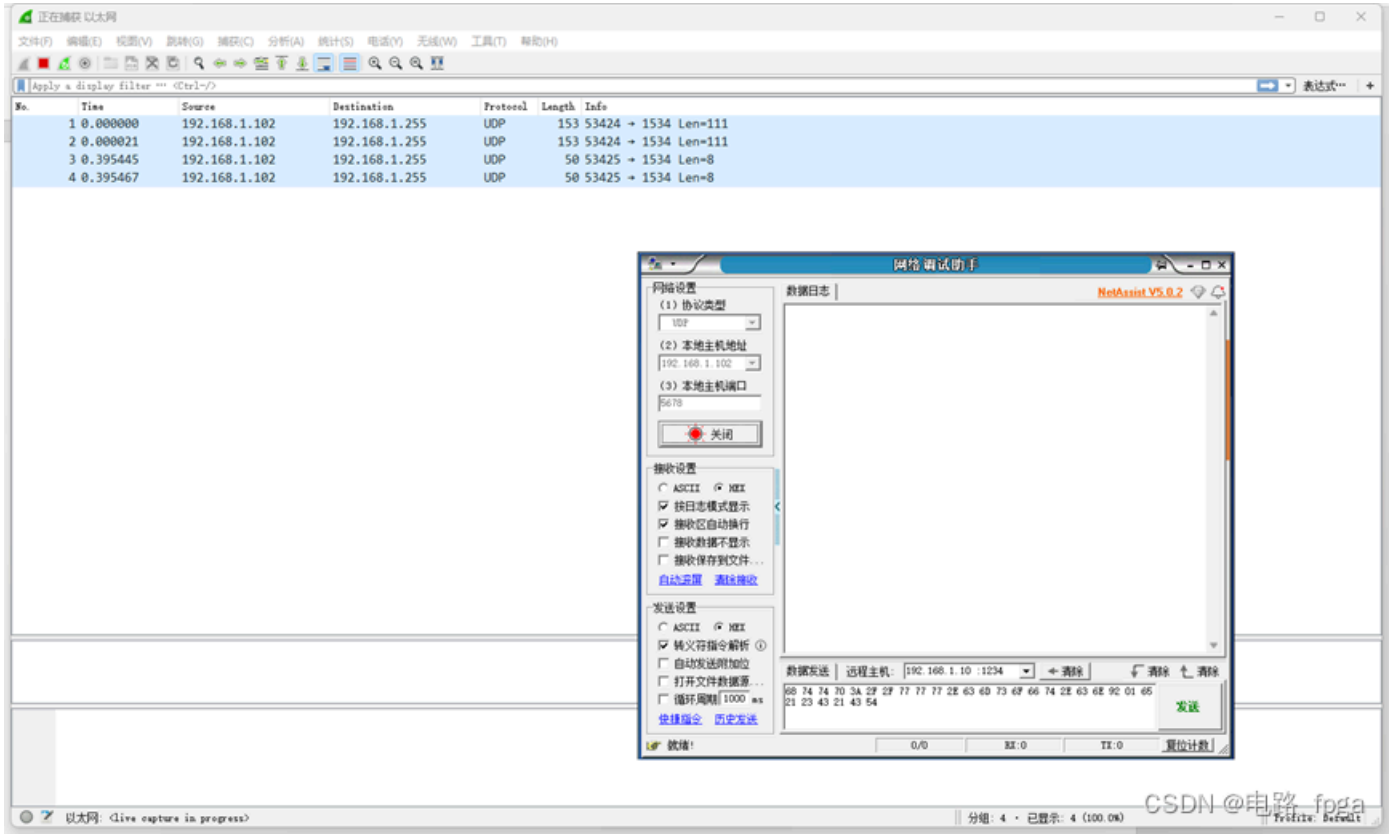

图15 wireshark与网络调试助手

网络调试助手需要设置协议类型为UDP，PC端的IP地址和UDP地址，需要与顶层文件的数值保持一致。然后打开连接，就会显示出FPGA的IP地址和UDP端口地址，如果该地址与开发板的地址不一样，可以手动进行修改。

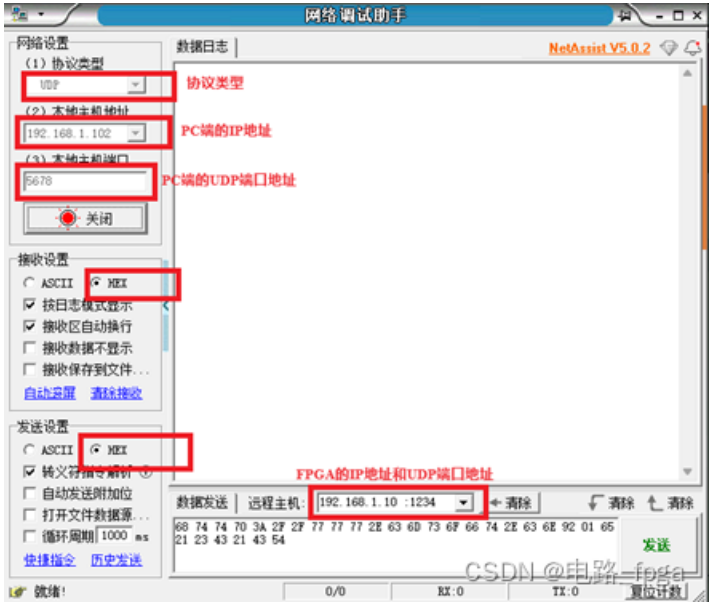

图16 网络调试助手的设置

之后将ILA设置为gmii_rx_dv的上升沿触发，连续抓取32个数据报文，然后wireshark也运行，最后点击网络调试助手的发送指令，即可抓取相关数据。网络调试助手发送三帧数据，如下图所示，FPGA向PC端返回接收到的三帧数据（蓝色数据是PC端通过UDP向FPGA发送的数据，绿色数据是FPGA通过UDP向PC端发送的数据）。

图17 网络调试助手收发数据

对比网络调试助手收发数据一致，由此证明FPGA接收和发送数据无误。

然后查看wireshark在这段时间抓取的数据报文，如下图所示。

PC端在通过UDP向FPGA发送数据报文之前，先通过广播的形式发送了一个ARP请求指令，去获取开发板的MAC地址，FPGA接收到ARP请求后，也是向PC端返回了ARP应答数据报文。

然后PC端通过UDP向FPGA发送三个数据报文，如下图所示，FPGA也对该报文进行了应答。

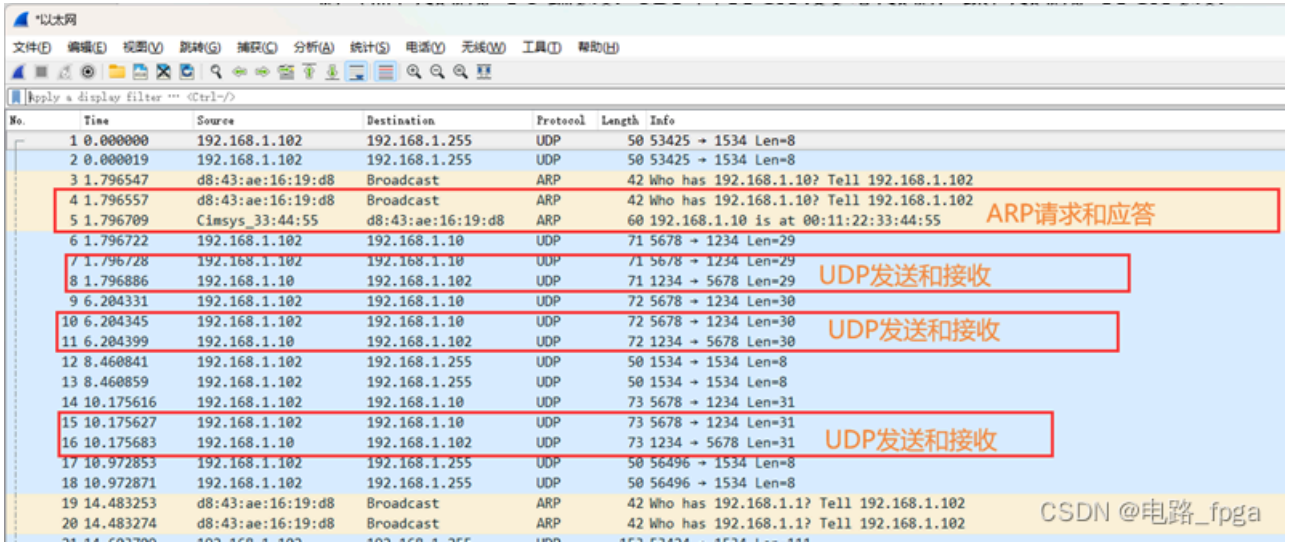

图18 wireshark抓取数据报文

前文对ARP的报文已经做了详细讲解，所以此处不对其报文进行分析了，我们双击UDP报文，查看其发送的数据段，如下图蓝色背景文字部分，与图17中第一帧数据保持一致。

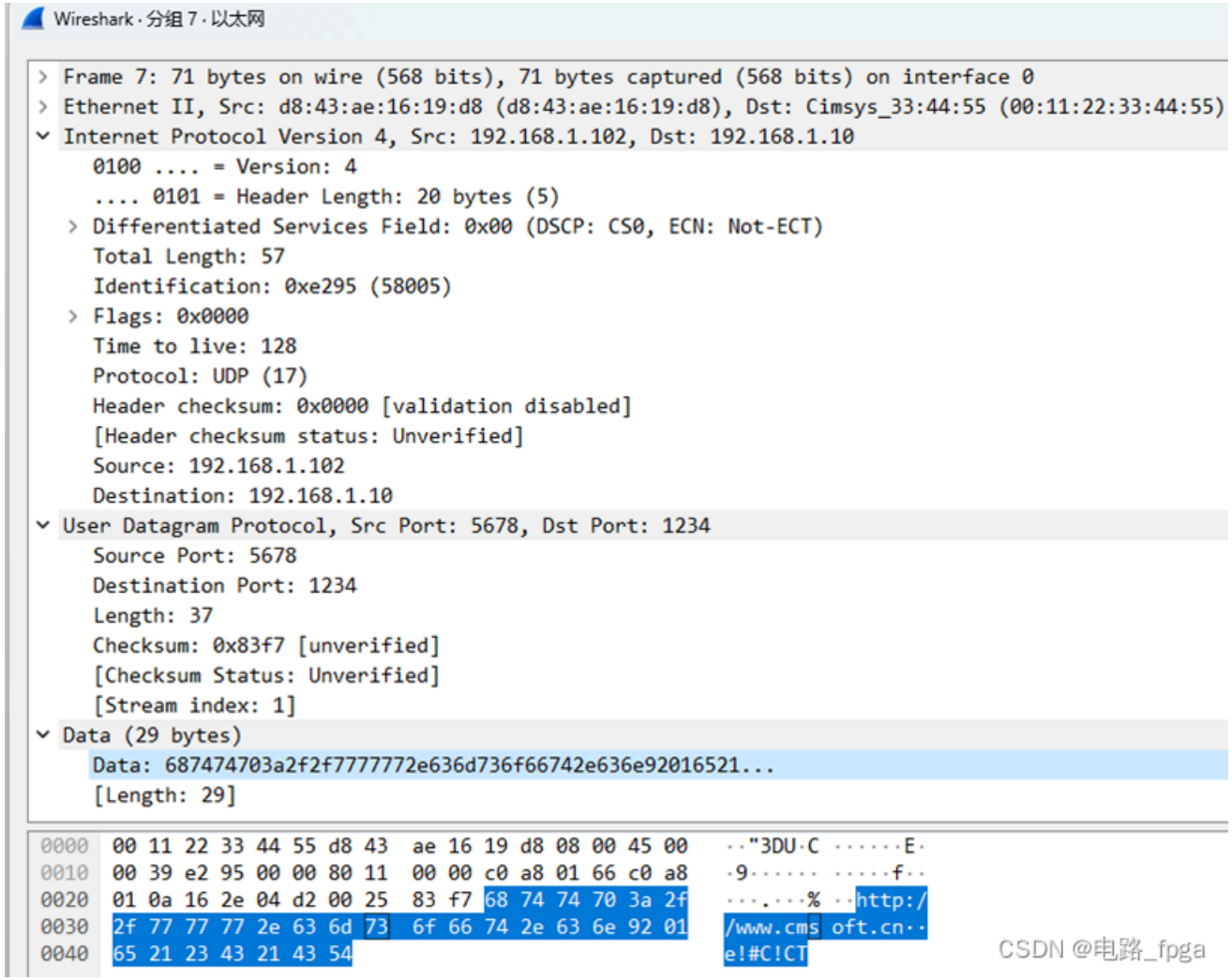

图19 wireshark抓取发送的第一帧数据报文

如下图是wirshark抓取的FPGA通过UDP给PC端发送的第一帧报文，可以从红框处得知源MAC和源IP地址为开发板，目的MAC和目的IP都是PC端的地址。接收的UDP数据就是蓝色文字，与图19PC端发送的数据保持一致。

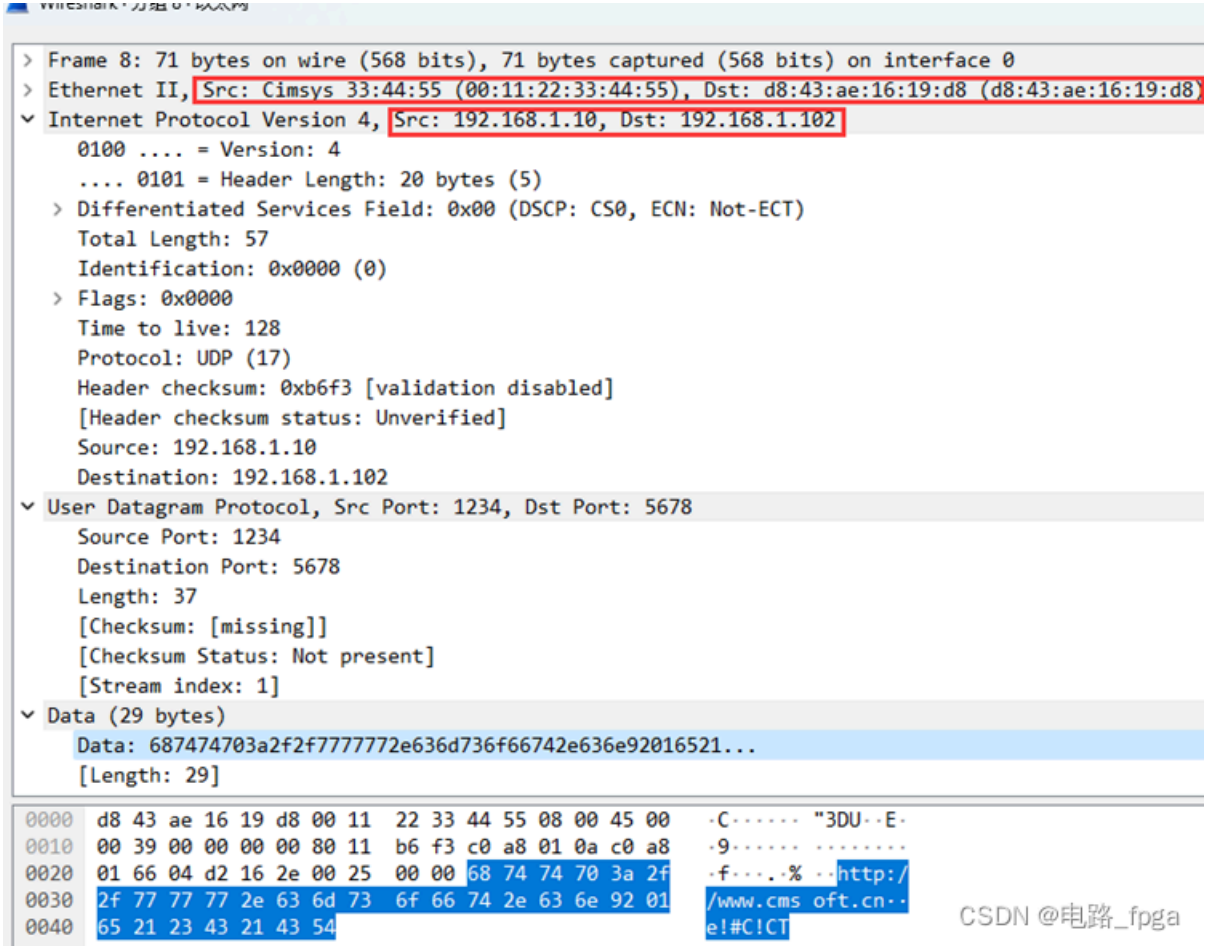

图20 wireshark抓取接收的第一帧数据报文

上述的数据报文通过ILA抓取如下所示，紫红色信号就是接收的报文数据信号。

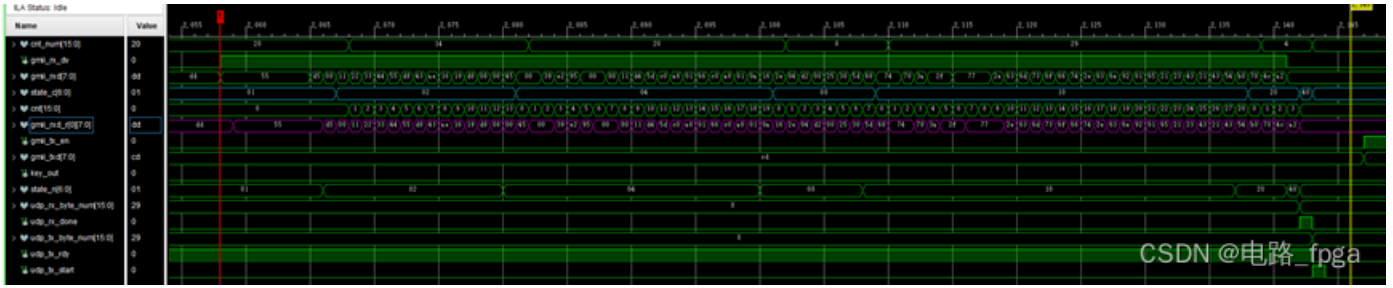

图21 ILA抓取接收的第一帧数据报文

将接收的UDP数据段放大后如下图所示，与图19和图17PC端发送的第一帧数据保持一致，因此FPGA这边接收数据没有问题。

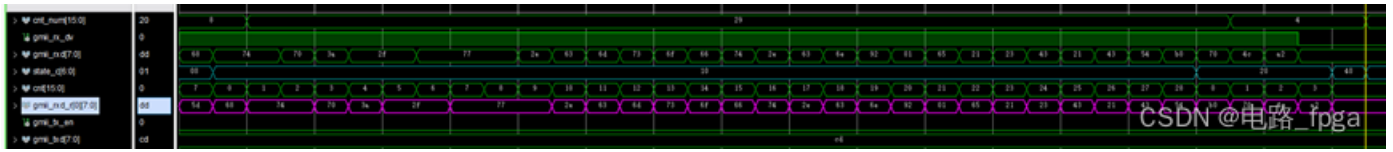

图22 UDP数据段放大

当FPGA接收到UDP数据包后，立马回复一帧UDP数据，如下所示，紫红色信号是接收的数据报文，橙色信号是发送的数据报文。

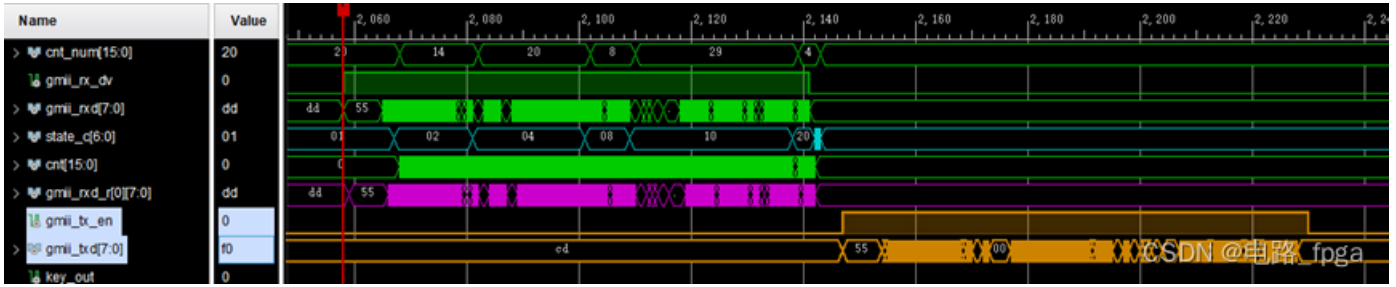

图23 UDP接收和发送报文

将发送报文的数据段放大，结果如下所示，与图17和图20wireshark抓取的数据一致，由此证明该设计接收和发送数据均没有问题。

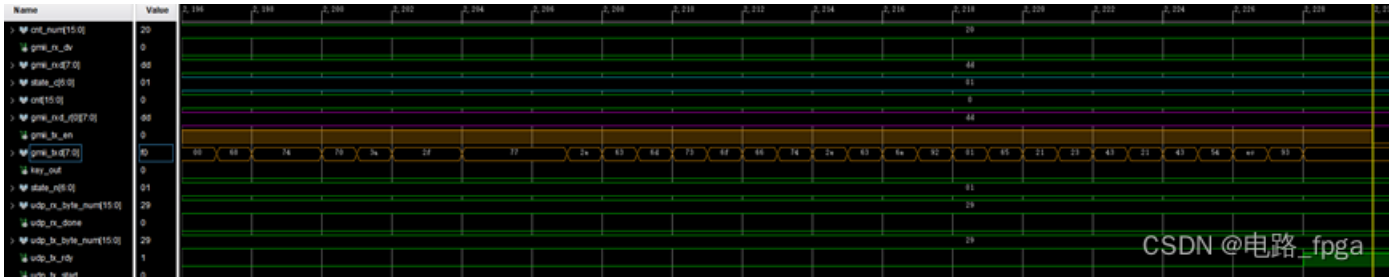

图24 发送报文数据段

最后就是验证ICMP的问题了，直接打开命令提示符，然后输入ping 192.168.1.10指令，运行结果如下所示：

图25 ping指令验证

上图表示FPGA接收到PC端的回显请求时，能够向PC端发送回显应答数据报文，以此验证以太网链路是否通畅。

关于UDP的发送和接收本文就做这么多讲解，当然这并不是我们最终想要使用的模块，因为ARP、UDP、ICMP这三个模块其实很多地方都是类似的，使用三个独立的模块完全没有必要，会额外消耗很多资源。

后文会把这三个模块进行整合设计，将模块合成一个eth模块，该模块可以实现对ARP、ICMP、UDP报文的接收，并根据需要发送相应报文。

获取本文工程的方式是在公众号后台回复“**基于FPGA的UDP回环设计**”（不包括引号）。

您的支持是我更新的最大动力！将持续更新工程，如果本文对您有帮助，还请多多点赞👍、评论💬和收藏⭐！