基于FPGA实现ICMP协议（包含源工程文件）

基于FPGA的以太网相关文章导航，点击查看。

前文对IP协议和ICMP协议格式做了讲解，本文通过 FPGA 实现ICMP协议，PC端向开发板产生回显请求，FPGA接收到回显请求时，向PC端发出回显应答。为了不去手动绑定开发板的MAC地址和IP地址，还是需要ARP模块。

1、顶层设计

顶层模块直接使用vivado工程截图，如下图所示，顶层包括6个模块，按键消抖模块key、ARP收发模块、RGMII与GMII转换模块rgmii_to_gmii、锁相环模块在前文ARP协议实现时均已详细讲解，故本文不再赘述。

由于arp和icmp的发送模块需要使用同一组数据线，所以需要一个arp_icmp_ctrl模块去确定输出哪一路信号。

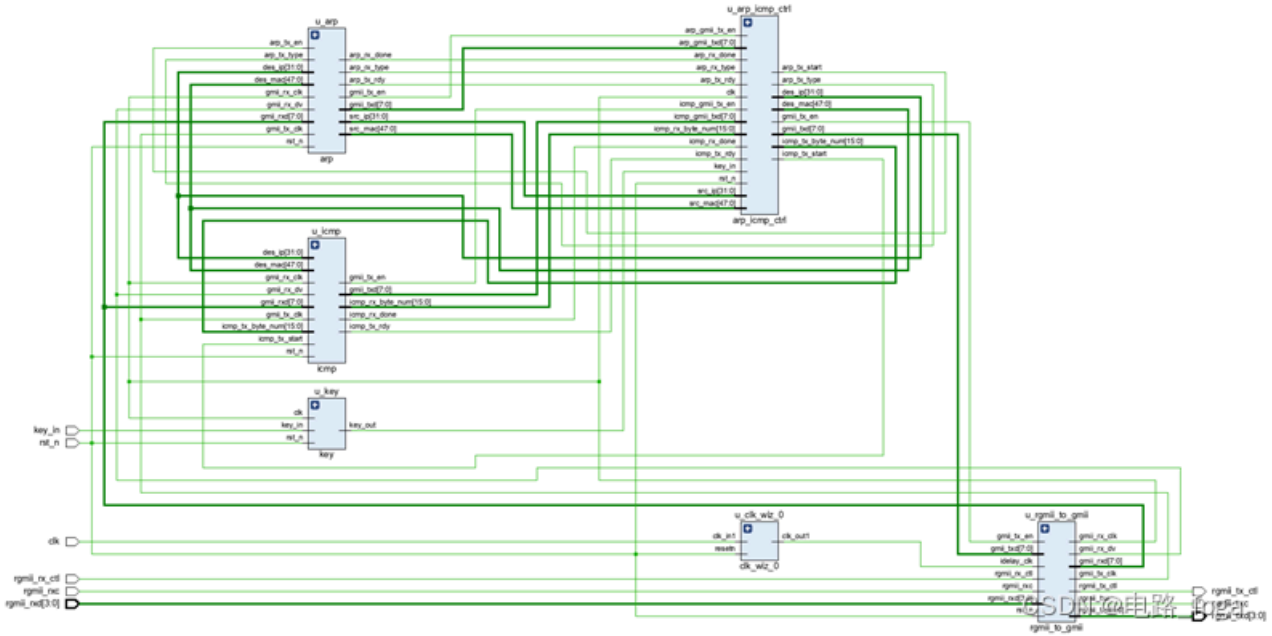

图1 顶层信号流向

顶层参考代码如下所示：

```
    //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_ctrl  u_arp_icmp_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；
        .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_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     ),
        .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_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。
    .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;
)
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；
    );

    /*ila_0 u_ila_0 (
        .clk        ( gmii_rx_clk                  ),//input wire clk
        .probe0     ( gmii_rx_dv                   ),//input wire [0:0]  probe0
        .probe1     ( gmii_rxd                     ),//input wire [7:0]  probe1
```

```
        .probe2     ( gmii_tx_en                    ),//input wire [0:0]  probe2
        .probe3     ( gmii_txd                      ),//input wire [7:0]  probe3
        .probe4     ( u_icmp.u_icmp_rx.state_n      ),//input wire [6:0]  probe4
        .probe5     ( u_icmp.u_icmp_rx.state_c      ),//input wire [6:0]  probe5
        .probe6     ( icmp_gmii_tx_en               ),//input wire [0:0]  probe6
        .probe7     ( icmp_gmii_txd                 ),//input wire [7:0]  probe7
        .probe8     ( icmp_tx_rdy                   ),//input wire [0:0]  probe8
        .probe9     ( icmp_rx_done                  ),//input wire [0:0]  probe9
        .probe10    ( u_icmp.u_icmp_rx.error_flag   ),//input wire [0:0]  probe10
        .probe11    ( u_icmp.u_icmp_rx.fifo_wr      ),//input wire [0:0]  probe11
        .probe12    ( u_icmp.u_icmp_rx.cnt          ),//input wire [7:0]  probe12
        .probe13    ( u_icmp.u_icmp_rx.cnt_num      ),//input wire [7:0]  probe13
        .probe14    ( u_icmp.u_icmp_rx.gmii_rxd_r[0]),//input wire [7:0]  probe14
        .probe15    ( u_icmp.u_icmp_rx.fifo_wdata   ) //input wire [7:0]  probe15
    );*/
```

icmp模块实现对ICMP协议的接收和发送，下图是该模块的内部模块分布图，包括ICMP接收模块icmp_rx、ICMP发送模块icmp_tx，两个CRC校验模块，分别对接收和发送的数据进行CRC校验。因为回显应答必须把回显请求的数据原封不动的发送出去，因此使用一个FIFO对回显请求的数据进行暂存。

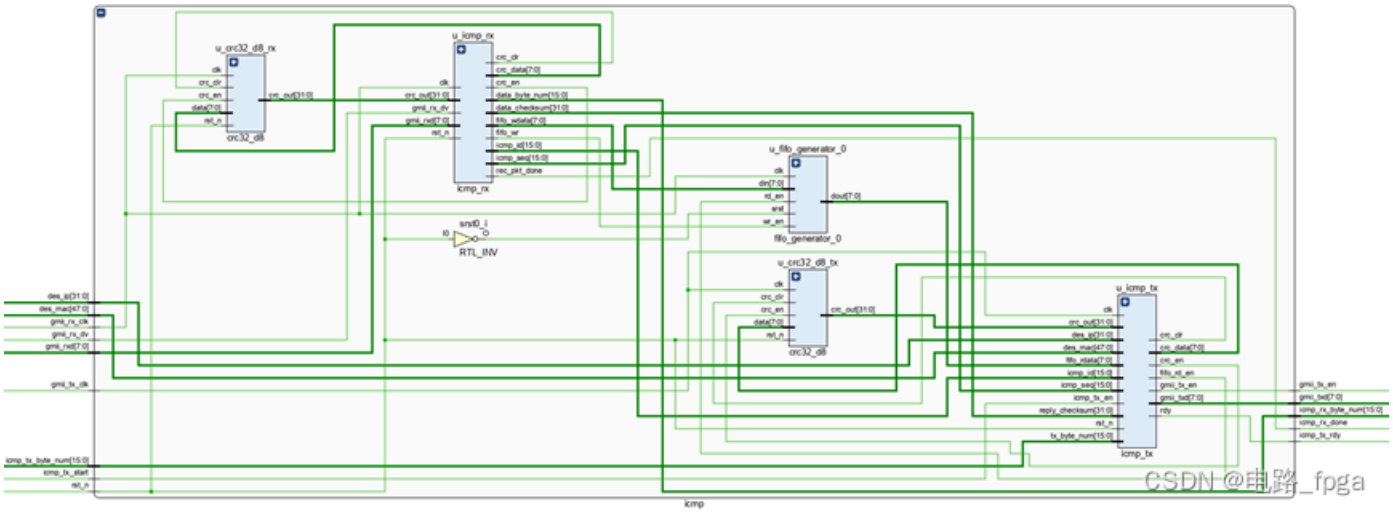

图2 ICMP模块信号流向

ICMP顶层参考代码如下所示：

```
    //例化ICMP接收模块；
    icmp_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_icmp_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；
        .rec_pkt_done   ( icmp_rx_done      ),//ICMP接收完成信号，高电平有效；
        .fifo_wr        ( fifo_wr_en        ),//fifo写使能。
        .fifo_wdata     ( fifo_wdata        ),//fifo写数据，将接收到的ICMP数据写入FIFO中。
        .data_byte_num  ( icmp_rx_byte_num  ),//以太网接收的有效数据字节数 单位:byte
        .icmp_id        ( icmp_id           ),//ICMP标识符;
        .icmp_seq       ( icmp_seq          ),//ICMP序列号;
        .data_checksum  ( data_checksum     ),//ICMP数据段的校验和;
        .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ICMP发送模块；
    icmp_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;
        .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_icmp_tx (
        .clk            ( gmii_tx_clk       ),//时钟信号;
        .rst_n          ( rst_n             ),//复位信号，低电平有效;
        .reply_checksum ( data_checksum     ),//ICMP数据段的校验和;
        .icmp_id        ( icmp_id           ),//ICMP标识符;
        .icmp_seq       ( icmp_seq          ),//ICMP序列号;
        .icmp_tx_en     ( icmp_tx_start     ),//ICMP发送使能信号;
        .tx_byte_num    ( icmp_tx_byte_num  ),//ICMP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;
        .fifo_rd_en     ( fifo_rd_en        ),//FIFO读使能信号。
        .fifo_rdata     ( fifo_rdata        ),//从FIFO读出，以太网需要发送的数据。
        .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            ( icmp_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;
    );

    //例化FIFO;
    fifo_generator_0 u_fifo_generator_0 (
        .clk    ( gmii_rx_clk   ),//input wire clk
        .srst   ( ~rst_n        ),//input wire srst
        .din    ( fifo_wdata    ),//input wire [7 : 0] din
        .wr_en  ( fifo_wr_en    ),//input wire wr_en
```

```
        .rd_en  ( fifo_rd_en    ),//input wire rd_en
        .dout   ( fifo_rdata    ),//output wire [7 : 0] dout
        .full   (               ),//output wire full
        .empty  (               ) //output wire empty
    );
```

ICMP顶层TestBench参考代码如下所示：

```
`timescale 1 ns/1 ns
module test();
    parameter   CYCLE       =   10                          ;//系统时钟周期，单位ns，默认10ns；
    parameter   RST_TIME    =   10                          ;//系统复位持续时间，默认10个系统时钟周期；
    parameter   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；
    parameter   BOARD_MAC   =   48'h00_11_22_33_44_55       ;
    parameter   BOARD_IP    =   {8'd192,8'd168,8'd1,8'd10}  ;
    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

    reg                         clk                         ;//系统时钟，默认100MHz；
    reg                         rst_n                       ;//系统复位，默认低电平有效；
    reg         [7 : 0]         gmii_rxd                    ;
    reg                         gmii_rx_dv                  ;
    wire                        icmp_tx_start               ;
    wire        [15 : 0]        icmp_tx_byte_num            ;
    wire        [47 : 0]        des_mac                     ;
    wire        [31 : 0]        des_ip                      ;
    wire                        gmii_tx_en                  ;
    wire        [7 : 0]         gmii_txd                    ;
    wire                        icmp_rx_done                ;
    wire        [15 : 0]        icmp_rx_byte_num            ;
    wire                        icmp_tx_rdy                 ;

    reg         [7 : 0]         rx_data [255 : 0]           ;//申请256个数据的存储器

    assign icmp_tx_start = icmp_rx_done;
```

```
    assign icmp_tx_byte_num = icmp_rx_byte_num;
    assign des_mac = 0;
    assign des_ip = 0;

    icmp #(
        .BOARD_MAC  ( BOARD_MAC ),
        .BOARD_IP   ( BOARD_IP  ),
        .DES_MAC    ( DES_MAC   ),
        .DES_IP     ( DES_IP    ),
        .ETH_TYPE   ( ETH_TYPE  )
    )
    u_icmp (
        .rst_n              ( rst_n             ),
        .gmii_rx_clk        ( clk               ),
        .gmii_rx_dv         ( gmii_rx_dv        ),
        .gmii_rxd           ( gmii_rxd          ),
        .gmii_tx_clk        ( clk               ),
        .icmp_tx_start      ( icmp_tx_start     ),
        .icmp_tx_byte_num   ( icmp_tx_byte_num  ),
        .des_mac            ( des_mac           ),
        .des_ip             ( des_ip            ),
        .gmii_tx_en         ( gmii_tx_en        ),
        .gmii_txd           ( gmii_txd          ),
        .icmp_rx_done       ( icmp_rx_done      ),
        .icmp_rx_byte_num   ( icmp_rx_byte_num  ),
        .icmp_tx_rdy        ( icmp_tx_rdy       )
    );

    reg                 crc_clr         ;
    reg                 gmii_crc_vld    ;
    reg     [7 : 0]     gmii_rxd_r      ;
    reg                 gmii_rx_dv_r    ;
    reg                 crc_data_vld    ;
    reg     [9 : 0]     i               ;
    reg     [15 : 0]    num             ;
    wire    [31 : 0]    crc_out         ;

    //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;gmii_rxd = 0; gmii_rx_dv = 0;gmii_crc_vld = 1'b0;num=0;
        gmii_rxd_r=0;gmii_rx_dv_r=0;crc_clr=0;
        for(i = 0 ; i < 256 ; i = i + 1)begin
            #1;
            rx_data[i] = {$random} % 256;//初始化存储体;
        end
        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(4)begin//发送2帧数据;
            gmii_tx_test(18);
            gmii_crc_vld = 1'b1;
            gmii_rxd_r = crc_out[7 : 0];
            #(CYCLE);
            gmii_rxd_r = crc_out[15 : 8];
            #(CYCLE);
            gmii_rxd_r = crc_out[23 : 16];
            #(CYCLE);
            gmii_rxd_r = crc_out[31 : 24];
            #(CYCLE);
            gmii_crc_vld = 1'b0;
            crc_clr = 1'b1;
            #(CYCLE);
            crc_clr = 1'b0;
            @(posedge icmp_rx_done);
            #(50*CYCLE);
        end
        #(20*CYCLE);
        $stop;//停止仿真;
    end

    task gmii_tx_test(

```

```
        input [15 : 0]  data_num   //需要把多少个存储体中的数据进行发送，取值范围[18,255]；
    );
        reg [31 : 0] ip_check;
        reg [15 : 0] total_num;
        reg [31 : 0] icmp_check;
        begin
            total_num = data_num + 28;
            icmp_check =  16'h1 + 16'h8;//ICMP首部相加；
            ip_check = DES_IP[15:0] + BOARD_IP[15:0] + DES_IP[31:16] + BOARD_IP[31:16] + 16'h4500 + total_num + 16'h4000 + num + 16'h8001;

            if(~data_num[0])begin//ICMP数据段个数为偶数；
                for(i=0 ; 2*i < data_num ; i= i+1)begin
                    #1;//计算ICMP数据段的校验和。
                    icmp_check = icmp_check + {rx_data[i][7:0],rx_data[i+1][7:0]};
                end
            end
            else begin//ICMP数据段个数为奇数；
                for(i=0 ; 2*i < data_num+1 ; i= i+1)begin
                    #1;//计算ICMP数据段的校验和。
                    if(2*i + 1 == data_num)
                        icmp_check = icmp_check + {rx_data[i][7:0]};
                    else
                        icmp_check = icmp_check + {rx_data[i][7:0],rx_data[i+1][7:0]};
                end
            end
            crc_data_vld = 1'b0;
            #(CYCLE);
            repeat(7)begin//发送前导码7个8'H55；
                gmii_rxd_r = 8'h55;
                gmii_rx_dv_r = 1'b1;
                #(CYCLE);
            end
            gmii_rxd_r = 8'hd5;//发送SFD，一个字节的8'hd5；
            #(CYCLE);
            crc_data_vld = 1'b1;
            //发送以太网帧头数据；
            for(i=0 ; i<6 ; i=i+1)begin//发送6个字节的目的MAC地址；
                gmii_rxd_r = BOARD_MAC[47-8*i -: 8];
                #(CYCLE);
            end

```

```
            for(i=0 ; i<6 ; i=i+1)begin//发送6个字节的源MAC地址；
                gmii_rxd_r = DES_MAC[47-8*i -: 8];
                #(CYCLE);
            end
            for(i=0 ; i<2 ; i=i+1)begin//发送2个字节的以太网类型；
                gmii_rxd_r = ETH_TYPE[15-8*i -: 8];
                #(CYCLE);
            end
            //发送IP帧头数据；
            gmii_rxd_r = 8'H45;
            #(CYCLE);
            gmii_rxd_r = 8'd00;
            ip_check = ip_check[15 : 0] + ip_check[31:16];
            icmp_check = icmp_check[15 : 0] + icmp_check[31:16];
            #(CYCLE);
            gmii_rxd_r = total_num[15:8];
            ip_check = ip_check[15 : 0] + ip_check[31:16];
            icmp_check = icmp_check[15 : 0] + icmp_check[31:16];
            #(CYCLE);
            gmii_rxd_r = total_num[7:0];
            ip_check = ~ip_check[15 : 0];
            icmp_check = ~icmp_check[15 : 0];
            #(CYCLE);
            gmii_rxd_r = num[15:8];
            #(CYCLE);
            gmii_rxd_r = num[7:0];
            #(CYCLE);
            gmii_rxd_r = 8'h40;
            #(CYCLE);
            gmii_rxd_r = 8'h00;
            #(CYCLE);
            gmii_rxd_r = 8'h80;
            #(CYCLE);
            gmii_rxd_r = 8'h01;
            #(CYCLE);
            gmii_rxd_r = ip_check[15:8];
            #(CYCLE);
            gmii_rxd_r = ip_check[7:0];
            #(CYCLE);
            for(i=0 ; i<4 ; i=i+1)begin//发送6个字节的源IP地址；

```

```
                gmii_rxd_r = DES_IP[31-8*i -: 8];
                #(CYCLE);
            end
            for(i=0 ; i<4 ; i=i+1)begin//发送4个字节的目的IP地址；
                gmii_rxd_r = BOARD_IP[31-8*i -: 8];
                #(CYCLE);
            end
            //发送ICMP帧头及数据包；
            gmii_rxd_r = 8'h08;//发送回显请求。
            #(CYCLE);
            gmii_rxd_r = 8'h00;
            #(CYCLE);
            gmii_rxd_r = icmp_check[31:16];
            #(CYCLE);
            gmii_rxd_r = icmp_check[15:0];
            #(CYCLE);
            gmii_rxd_r = 8'h00;
            #(CYCLE);
            gmii_rxd_r = 8'h01;
            #(CYCLE);
            gmii_rxd_r = 8'h00;
            #(CYCLE);
            gmii_rxd_r = 8'h08;
            #(CYCLE);
            for(i=0 ; i<data_num ; i=i+1)begin
                gmii_rxd_r = rx_data[i];
                #(CYCLE);
            end
            crc_data_vld = 1'b0;
            gmii_rx_dv_r = 1'b0;
            num = num + 1;
        end
    endtask

    crc32_d8  u_crc32_d8_1 (
        .clk        ( clk           ),
        .rst_n      ( rst_n         ),
        .data       ( gmii_rxd_r    ),
        .crc_en     ( crc_data_vld  ),
        .crc_clr    ( crc_clr       ),
```

```
        .crc_out    ( crc_out       )
    );

    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            gmii_rxd <= 8'd0;
            gmii_rx_dv <= 1'b0;
        end
        else if(gmii_rx_dv_r || gmii_crc_vld)begin
            gmii_rxd <= gmii_rxd_r;
            gmii_rx_dv <= 1'b1;
        end
        else begin
            gmii_rx_dv <= 1'b0;
        end
    end

endmodule
```

2、ICMP接收模块

前文对ICMP的数据报做了详细讲解，ICMP数据报文的构成如下所示，包括前导码和帧起始符、以太网帧头、IP首部、ICMP首部、ICMP数据、CRC校验等几个模块。

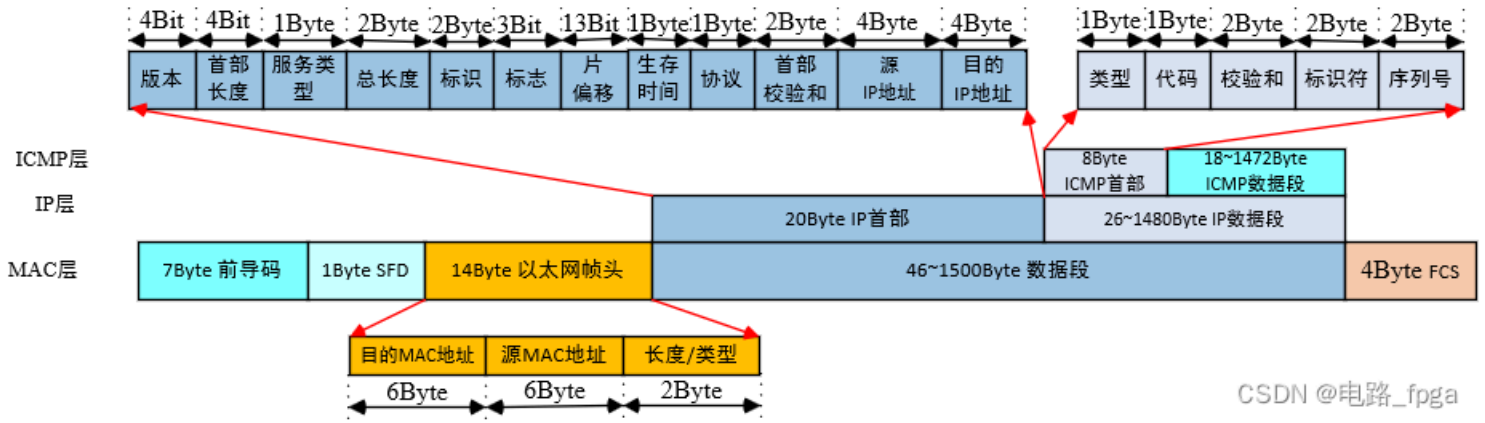

图3 以太网的ICMP数据报格式

本文检测接收的数据是不是ICMP回显请求，需要将回显请求的标识符、序列号、ICMP数据段保存下来，便于回显应答时使用。同时在接收ICMP数据时，应该把数据端的校验和计算出来，回显应答时就不再需要时间去计算数据段的校验和了。

此模块没有做IP首部校验和以及ICMP校验和，只做了CRC校验和，因为FPGA根据接收到的数据特征，其实能够大致判断是否正确，加上CRC校验无误就基本上不会出现错误了。

本模块以状态机为主体，嵌套一个计数器cnt实现，下图是状态机的状态转换图。

本模块会使用移位寄存器把输入数据gmii_rxd暂存7个时钟周期，便于前导码和帧起始符的检测，该移位寄存器的数据也可以在后续中使用，所以使用移位寄存器会比较方便。

空闲状态(IDLE)：这个状态就一直检测前导码和帧起始符，检测到后把start信号拉高，表示开始接收数据，状态机跳转到接收以太网帧头的状态。状态机的现态与移位寄存器gmii_rxd[0]的数据对齐，后续数据大多都来自该移位寄存器最低位置存储的数据。

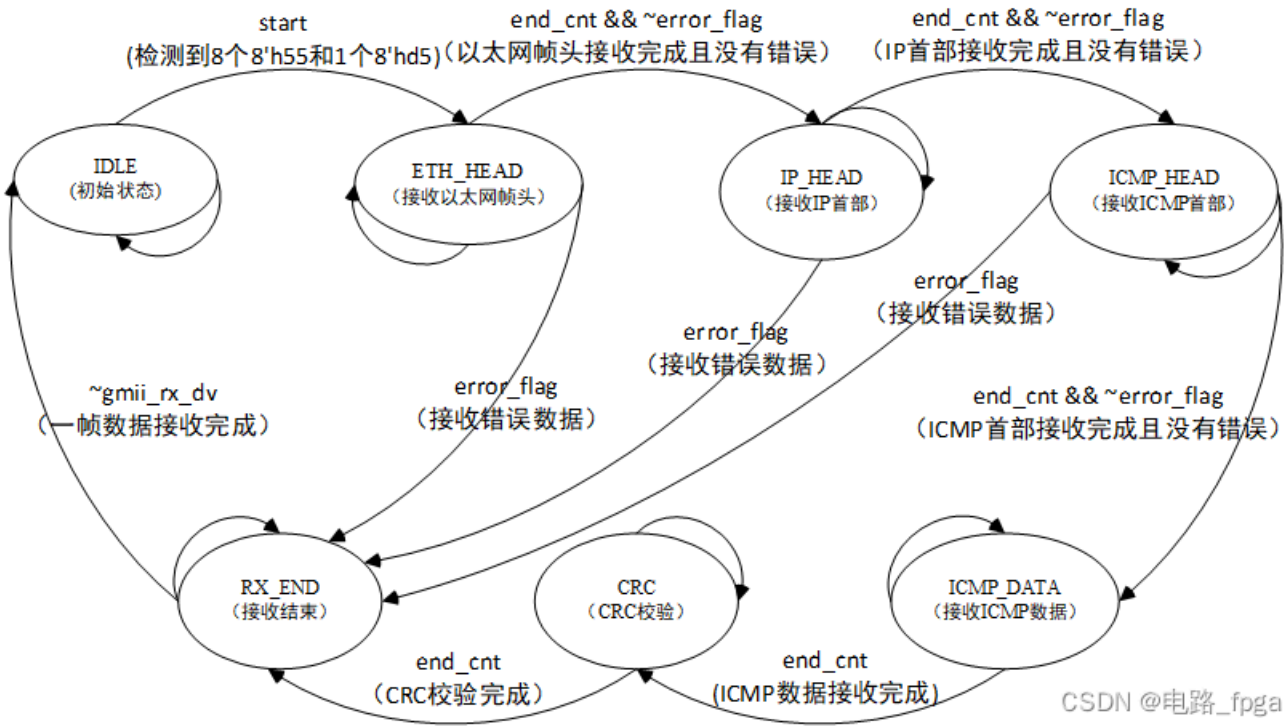

图4 ICMP接收模块状态转换图

后面各个状态分别接收对应数据，计数器cnt用来计数每个状态应该接收的数据个数。其中ICMP数据段的数据个数通过IP首部的总长度减去IP首部长度，在减去ICMP首部长度得到，ICMP数据段的长度还要输出，在后续进行回显应答时，从FIFO中读取对应个数的数据输出。

然后就是错误标志信号error_flag，就是接收的数据不是ICMP或者不是发送给开发板的数据时，就会拉高，此时就会把接收的数据报文丢弃。比如在以太网帧头部分检测到接收的目的MAC地址不是开发板MAC地址或广播地址，此时error_flag拉高，表示该数据报不是发送给开发板的，直接丢弃，不在继续接收。又比如在接收ICMP首部时，检测到该数据报文不是回显请求，则error_flag拉高，直接丢弃该报文，后续的数据不需要存入FIFO中。

注意在接收ICMP数据时，需要将接收的两字节数据拼接后相加，得到校验和（这是因为回显应答时需要先发送ICMP校验和，后发送ICMP数据，且需要发送的ICMP的数据存在FIFO中，提前取出不方便，所以在接收的时候就把数据段相加，得到数据段的累加和，后续在回显应答时直接使用即可）。也就是前文介绍的IP首部校验和计算方式，但是此处只把接收到的两字节数据相加，因为ICMP的校验和还包括ICMP首部，其余运算在ICMP发送时才能继续。

其余部分都比较简单，可以自行查看工程对应文件，参考代码如下：

```
    //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ICMP协议数据。
                state_n = ICMP_HEAD;
            end
            else begin
                state_n = state_c;
            end
        end
        ICMP_HEAD:begin
            if(error_flag)begin//在接收ICMP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ICMP帧头数据，且没有出现错误，则继续接收ICMP数据。
                state_n = ICMP_DATA;
            end
            else begin
                state_n = state_c;
            end
        end
        ICMP_DATA:begin
            if(error_flag)begin//在接收ICMP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ICMP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。
                ICMP_HEAD: cnt_num <= 16'd8;//ICMP帧头为8字节数据。
                ICMP_DATA: cnt_num <= icmp_data_length;//ICMP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ICMP协议，停止解析数据。
                            error_flag <= (gmii_rxd_r[0] != ICMP_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
                ICMP_HEAD : begin
                    if(add_cnt && cnt == 1)begin//ICMP报文类型不是回显请求。
                        error_flag <= (icmp_type != ECHO_REQUEST);
                    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;
        icmp_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 : icmp_data_length <= ip_total_length - ip_head_byte_num - 8;//计算ICMP报文数据段的长度，ICMP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ICMP首部相关数据;
always@(posedge clk)begin
    if(rst_n==1'b0)begin//初始值为0;
        icmp_type <= 8'd0;
        icmp_code <= 8'd0;
        icmp_checksum <= 16'd0;
        icmp_id <= 16'd0;
        icmp_seq <= 16'd0;
    end
    else if(state_c == ICMP_HEAD && add_cnt)begin
        case(cnt)
            16'd0 : icmp_type <= gmii_rxd_r[0];//接收ICMP报文类型。
            16'd1 : icmp_code <= gmii_rxd_r[0];//接收ICMP报文代码。
            16'd2,16'd3 : icmp_checksum <= {icmp_checksum[7:0],gmii_rxd_r[0]};//接收ICMP报文帧头和数据的校验和。
            16'd4,16'd5 : icmp_id <= {icmp_id[7:0],gmii_rxd_r[0]};//接收ICMP的ID。
            16'd6,16'd7 : icmp_seq <= {icmp_seq[7:0],gmii_rxd_r[0]};//接收ICMP报文的序列号。
            default: ;
        endcase
    end
end

//计算接收到的数据的校验和。
```

```
235
236     always@(posedge clk)begin
237         if(rst_n==1'b0)begin//初始值为0;
238             reply_checksum_add <= 16'd0;
239         end
240         else if(state_c == RX_END)begin//累加器清零。
241             reply_checksum_add <= 16'd0;
242         end
243         else if(state_c == ICMP_DATA && add_cnt)begin
244             if(end_cnt && icmp_data_length[0])begin//如果计数器计数结束且数据个数为奇数个，那么直接将当前数据与累加器相加。
245                 reply_checksum_add <= reply_checksum_add + {8'd0,gmii_rxd_r[0]};
246             end
247             else if(cnt[0])//计数器计数到奇数时，将前后两字节数据拼接相加。
248                 reply_checksum_add <= reply_checksum_add + {gmii_rxd_r[1],gmii_rxd_r[0]};
249         end
250     end
251
252     //控制FIFO使能信号，以及数据信号。
253     always@(posedge clk)begin
254         fifo_wdata <= (state_c == ICMP_DATA) ? gmii_rxd_r[0] : fifo_wdata;//在接收ICMP数据阶段时，接收数据。
255         fifo_wr <= (state_c == ICMP_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。
256     end
257
258     //生产CRC校验相关的数据和控制信号。
259     always@(posedge clk)begin
260         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。
261         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。
262         crc_en <= (state_c != IDLE) && (state_c != RX_END) && (state_c != CRC);//CRC校验使能信号。
263     end
264
265     //接收PC端发送来的CRC数据。
266     always@(posedge clk)begin
267         if(rst_n==1'b0)begin//初始值为0;
268             des_crc <= 24'hff_ff_ff;
269         end
270         else if(add_cnt && state_c == CRC)begin//先接收的是低位数据；
271             des_crc <= {gmii_rxd_r[0],des_crc[23:8]};
272         end
273     end
274
275     //生成相应的输出数据。
```

```
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
        rec_pkt_done <= 1'b0;
        data_byte_num <= 16'd0;
        data_checksum <= 16'd0;
        end//如果CRC校验成功，把ICMP协议接收完成信号拉高，把接收到ICMP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
            rec_pkt_done <= 1'b1;
            data_byte_num <= icmp_data_length;
            data_checksum <= reply_checksum_add;
        end
        else begin
            rec_pkt_done <= 1'b0;
        end
    end
```

仿真结果如下图所示，TestBench与ICMP发送模块共用，在下文ICMP顶层模块处提供。

如下图所示，当移位寄存器和输入数据gmii_rdv检测到前导码和帧起始符后，start信号拉高(天蓝色信号)，然后状态机（紫红色信号分别是状态机的次态跟现态）跳转到接收以太网帧头的状态，并且可以看到移位寄存器的最低数据gmii_rxd_r[0]数据与状态机的现态state_c是对齐的。

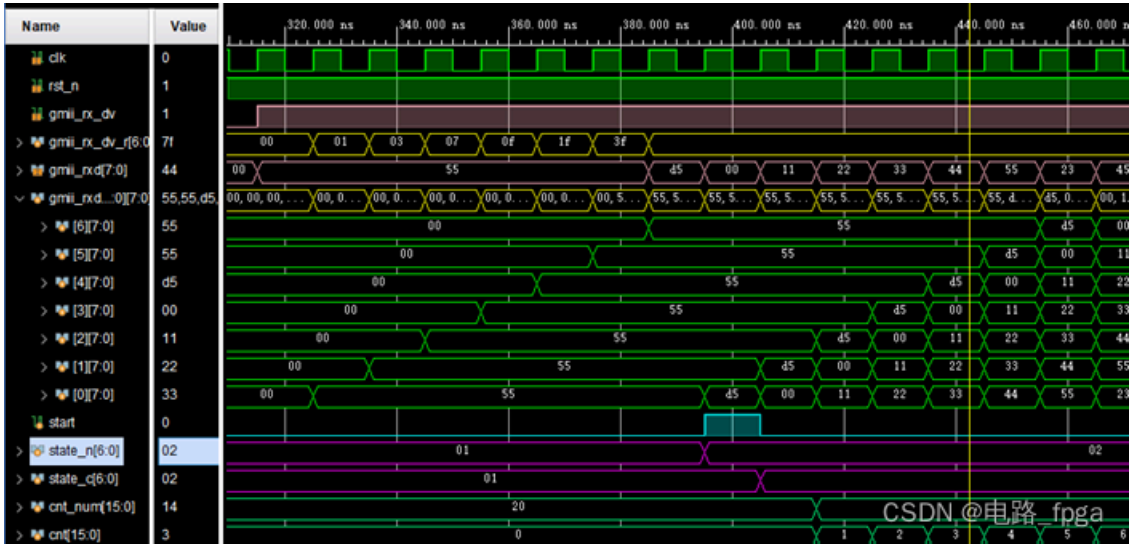

图5 ICMP接收模块仿真

然后接收以太网帧头，接收到目的MAC地址为48'h001122334455，与开发板的目的MAC一致，则继续接收数据。并且后续协议类型为16'h0800，是IP协议 ，则状态跳转到接收IP头部数据。

把crc校验模块的使能信号拉高，且把接收到的数据gmii_rxd_r[0]的数据输出给crc校验模块进行计算，如下图四个紫红色与crc有关的信号就是crc校验模块相关的信号。

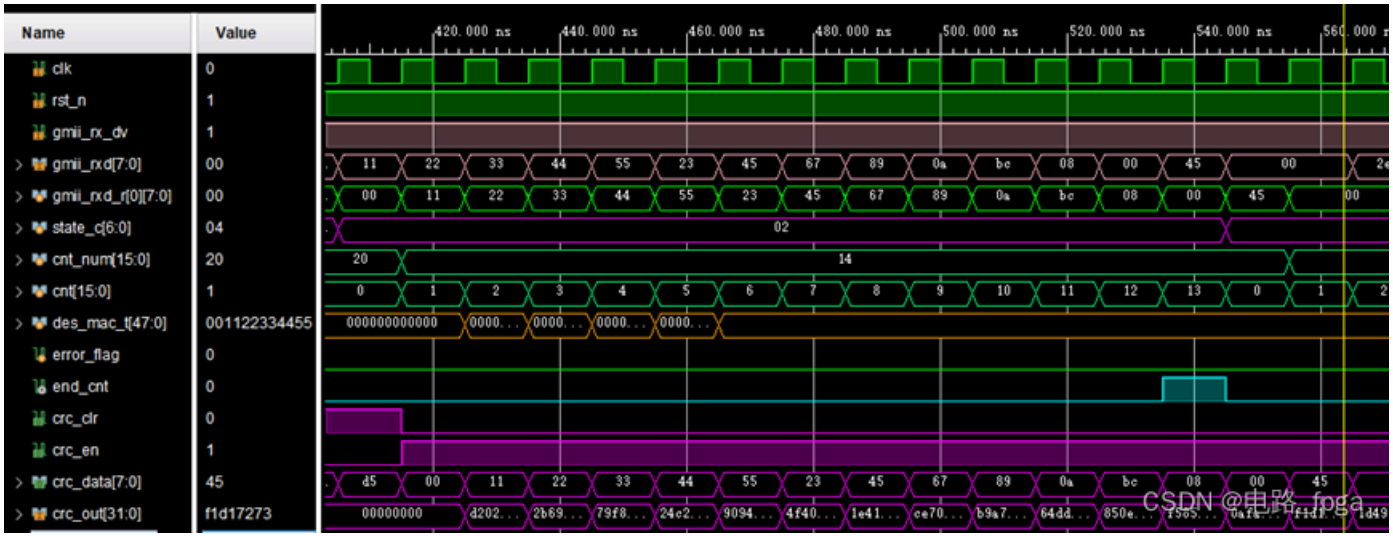

图6 接收以太网帧头数据

下图是接收IP头部数据，首先接收IP头部长度为20字节，然后接收IP报文总长度为46字节，计算出ICMP数据段为18字节的长度。在计数器cnt为9时，gmii_rxd_r[0]为1，表示后面是ICMP协议，计数器cnt等于18时，{des_ip,gmii_rxd_r[0],gmii_rxd}=32'hc0a8010a，与开发板的目的IP地址一致，则接收的数据报是发送给开发板的ICMP数据报。

计数器计数到最大值后，状态机跳转到接收ICMP首部数据状态。整个过程中CRC校验模块一直在对接收的数据进行校验计算。

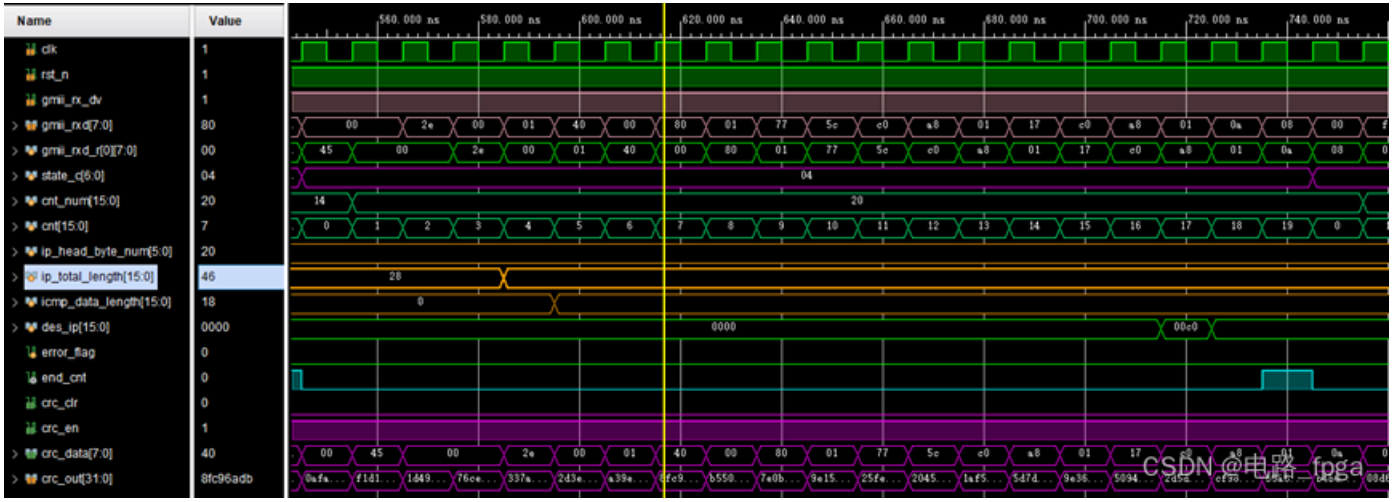

图7 接收IP头部数据

状态机在接收ICMP首部数据的仿真如下图所示，接收到的类型为8，代码为0，则表示该ICMP数据报是ICMP回显请求。继续接收ICMP的标识符为1，序列号为8，将序列号和标识符输出。注意crc校验模块依旧在对接收的数据进行计算。ICMP首部接收完成后状态机跳转到接收ICMP数据状态。

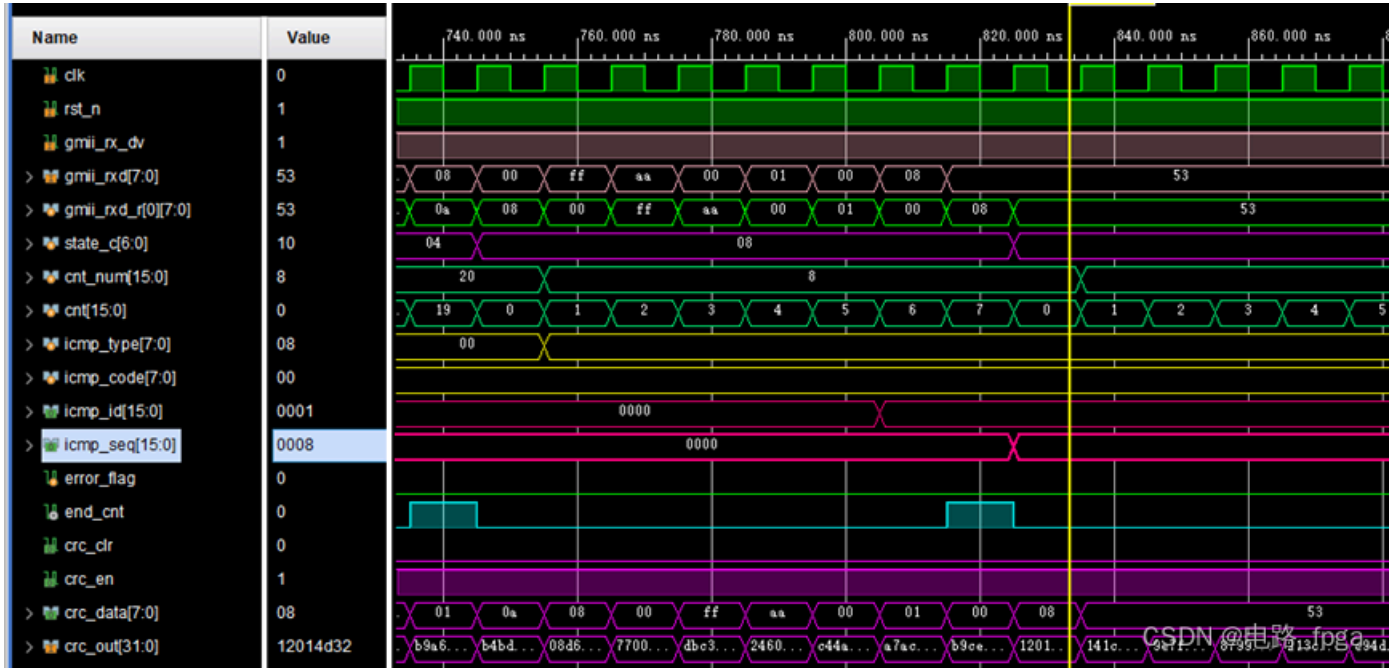

图8 接收ICMP头部数据

下图是接收ICMP数据的时序仿真，前文计算出需要接收18字节的ICMP数据，所以计数器cnt最大值为17。计数器为奇数时，将接收的前两字节数据拼接并与校验和数据相加，得到最后的校验和数据。注意如果接收数据个数是单数，则计数器结束时，把接收的数据直接与校验和相加。得到ICMP数据段校验和数据为32'h0003C4C9，输出给ICMP发送模块使用。

还需要把该数据段输出给外部FIFO进行暂存，将FIFO的写使能拉高，gmii_rxd_r[0]赋值给FIFO写数据。该状态结束时，crc校验模块也对接收的这帧数据校验完成，由图可知校验结果为32'h8c2aff78。

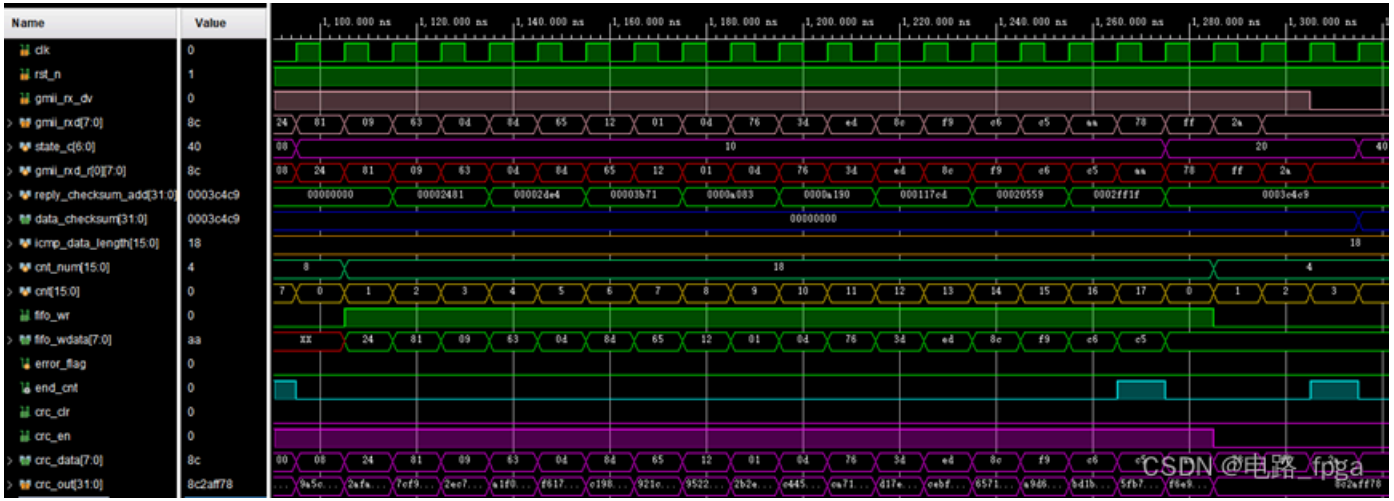

图9 接收ICMP数据

下图是接收CRC校验阶段，如图所示，计数器为3时，{ gmii_rxd_r[0],des_crc} = 32'h8c2aff78，与crc校验模块计算的结果一致，则表示接收的数据正确，把rec_pkt_done信号拉高一个时钟周期，表示接收数据完成，把ICMP数据段的长度和数据校验和输出，便于后面ICMP回显应答使用。

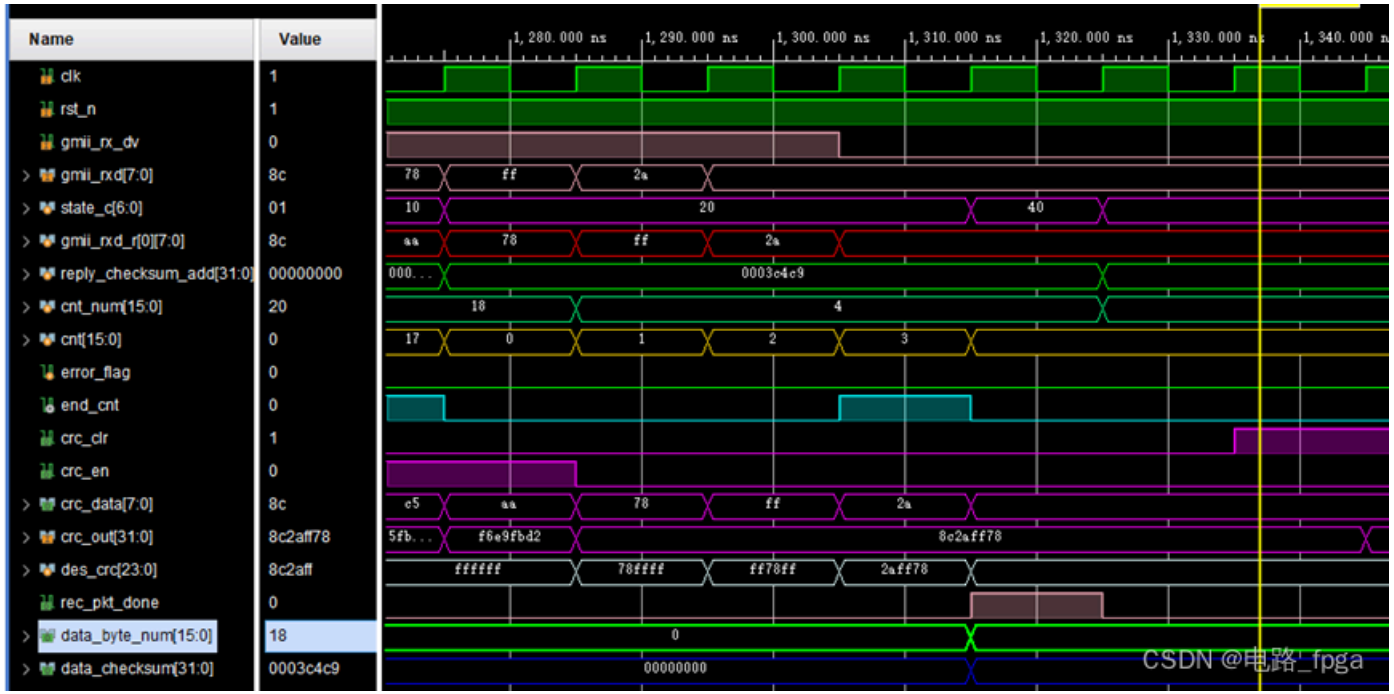

图10 接收CRC校验数据

3、ICMP发送模块

该模块设计比较简单，通过一个状态机，嵌套计数器就可以完成。状态转换图如下所示。

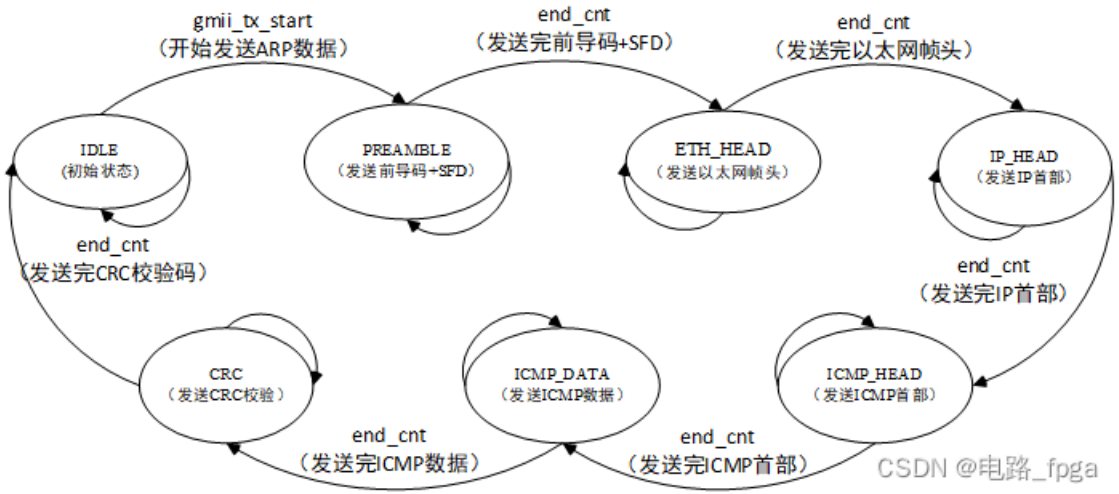

图11 ICMP发送模块状态转换图

设计思路与ARP发送模块没有太大区别，相比ARP发送模块，会稍微复杂一点，需要注意两点：

1. ICMP发送模块需要在发送IP首部和ICMP首部之前计算校验码，本设计是在发送以太网帧头的时候，同步计算出IP首部校验和、ICMP校验和，然后发送IP首部和ICMP首部时直接使用即可，也不会占用额外的时钟周期。

2. ICMP的数据段需要从外部FIFO（FIFO的配置在后文出现）中读取数据，本文使用的FIFO工作在超前模式，也就是读使能有效的时候，读数据就是有效的，不需要提前产生读使能。特别注意FIFO输出数据与数据流的对接问题。

计数器cnt的位宽扩展到16位，因为ICMP数据段可能会很长，所有计数器就与IP首部的总长度位宽保持一致。

其余设计与ARP发送模块基本一致，本文不再赘述，参考代码如下：

```
    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;
            icmp_head[0] <= 32'd0;
            icmp_head[1] <= 32'd0;
            ip_head_check <= 32'd0;
            icmp_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ICMP需要发送的数据个数进行暂存。
        else if(state_c == IDLE && icmp_tx_en)begin
            icmp_head[1] <= {icmp_id,icmp_seq};//16位ICMP标识符和16位序列号。
            icmp_check <= reply_checksum;//将数据段的校验和暂存。
            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ICMP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ICMP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'd01,16'd0};//分别表示生存时间，协议类型，1表示ICMP，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。
                    icmp_head[0] <= {ECHO_REPLY,24'd0};//8位类型与8位代码，16位的校验码。
                end
                16'd1 : begin//开始计算IP头部和ICMP的校验和数据，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];
                    icmp_check <= icmp_check + icmp_head[0][31 : 16];
                end
                16'd2 : begin
                    ip_head_check <= ip_head_check + ip_head[1][31 : 16];
                    icmp_check <= icmp_check + icmp_head[1][31 : 16];
                end

```

```
                16'd3 : begin
                    ip_head_check <= ip_head_check + ip_head[1][15 : 0];
                    icmp_check <= icmp_check + icmp_head[1][15 : 0];
                end
                16'd4 : begin
                    ip_head_check <= ip_head_check + ip_head[2][31 : 16];
                    icmp_check <= icmp_check[31 : 16] + icmp_check[15 : 0];//可能出现进位,累加一次。
                end
                16'd5 : begin
                    ip_head_check <= ip_head_check + ip_head[3][31 : 16];
                    icmp_check <= icmp_check[31 : 16] + icmp_check[15 : 0];//可能出现进位,再累加一次。
                end
                16'd6 : begin
                    ip_head_check <= ip_head_check + ip_head[3][15 : 0];
                    icmp_head[0][15:0] <= ~icmp_check[15 : 0];//按位取反得到校验和。
                    icmp_check <= 32'd0;//将校验和清零，便于下次使用。
                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
                    icmp_check <= 32'd0;//将校验和清零，便于下次使用。
                    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(icmp_tx_en)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 = ICMP_HEAD;
```

```
                end
                else begin
                    state_n = state_c;
                end
            end
            ICMP_HEAD:begin
                if(end_cnt)begin//发送完ICMP帧头数据；
                    state_n = ICMP_DATA;
                end
                else begin
                    state_n = state_c;
                end
            end
            ICMP_DATA:begin
                if(end_cnt)begin//发送完icmp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。
            ICMP_HEAD : cnt_num <= 16'd8;//发送8字节的ICMP帧头数据。
            ICMP_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];
            ICMP_HEAD : if(cnt < 4)//发送ICMP帧头数据。
                            crc_data <= icmp_head[0][31 - 8*cnt -: 8];
                        else
                            crc_data <= icmp_head[1][31 - 8*(cnt-4) -: 8];
            ICMP_DATA : if(cnt_num >= 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 <= fifo_rdata;//如果满足最小要求，将从FIFO读出的数据输出即可。
                        else if(cnt < cnt_num)//不满足最小要求时，先将需要发送的数据发送完。
                            crc_data <= fifo_rdata;//将从FIFO读出的数据输出即可。
                        else//剩余数据补充0.
                            crc_data <= 8'd0;
            default : ;
        endcase
    end
end

//fifo读使能信号，初始值为0，当发送完ICMP帧头时拉高，当发送完ICMP数据时拉低。
always@(posedge clk)begin
    if(rst_n==1'b0)begin//初始值为0;
        fifo_rd_en <= 1'b0;
    end
    else if(state_c == ICMP_HEAD && end_cnt)begin
```

```
            fifo_rd_en <= 1'b1;
        end
        else if(state_c == ICMP_DATA && end_cnt)begin
            fifo_rd_en <= 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(icmp_tx_en || state_c != IDLE)
            rdy = 1'b0;
        else
            rdy = 1'b1;
    end
```

TestBench与ICMP接收模块共用，在后文出现，仿真如下所示，当检测到开始发送数据信号有效时，将ICMP数据长度、数据段的校验和reply_checksum、目的MAC地址、目的IP地址保存，计算出IP的报文总长度。

状态机跳转到发送前导码和帧起始符状态，crc_data这个数据延时一拍就会作为输出数据gmii_txd。

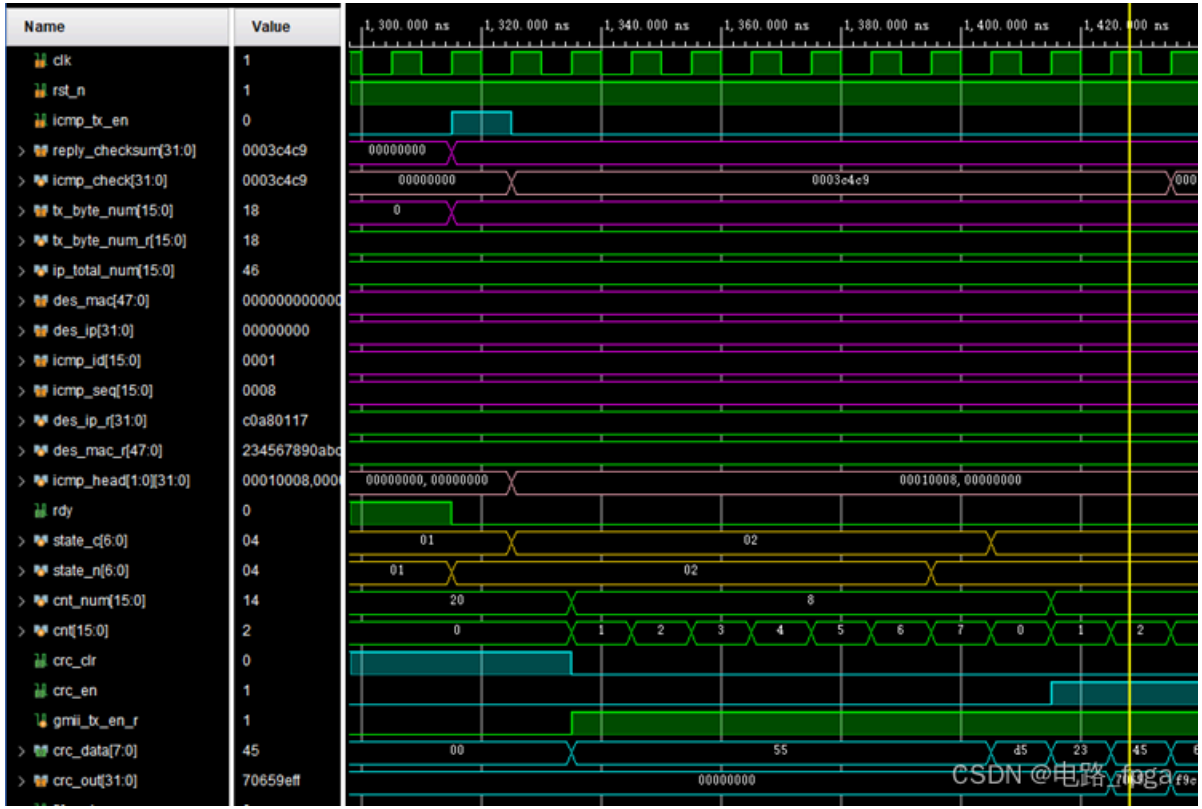

图12 开始发送数据

状态机处于发送以太网帧头状态时，还在计算IP首部和ICMP的校验和，并且将计算结果存储到IP首部和ICMP首部存储体的对应位置，仿真如下图所示。

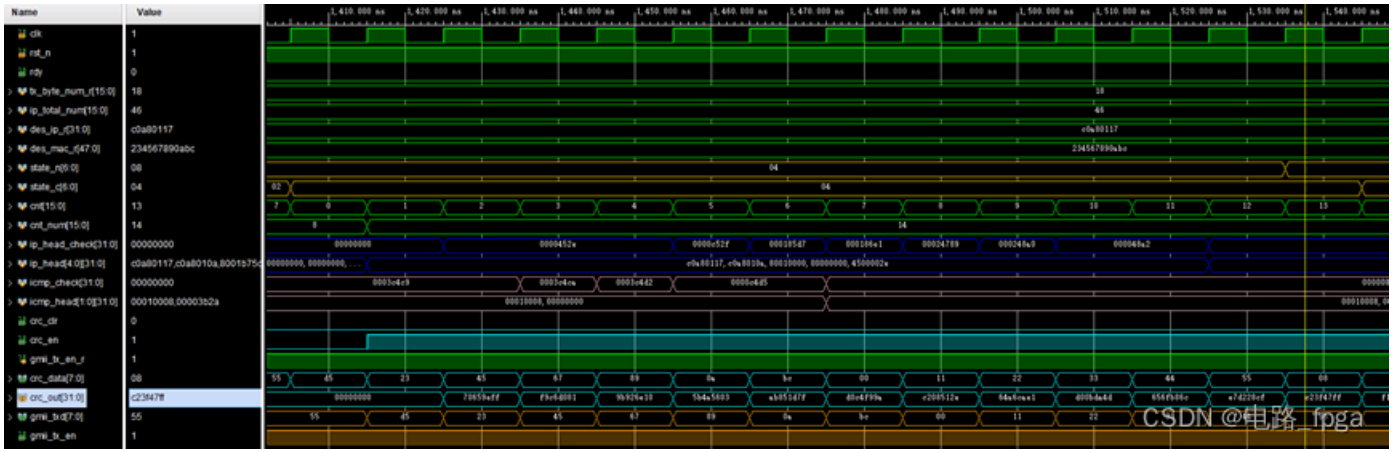

图13 产生以太网帧头且计算校验和

下图时状态机处于发送IP首部状态，将IP首部存储体中的数据依次输出，蓝色信号为IP首部存储体数据，crc_data是输出给crc校验模块计算的数据，该信号延迟一个时钟周期后得到gmii_txd输出信号。

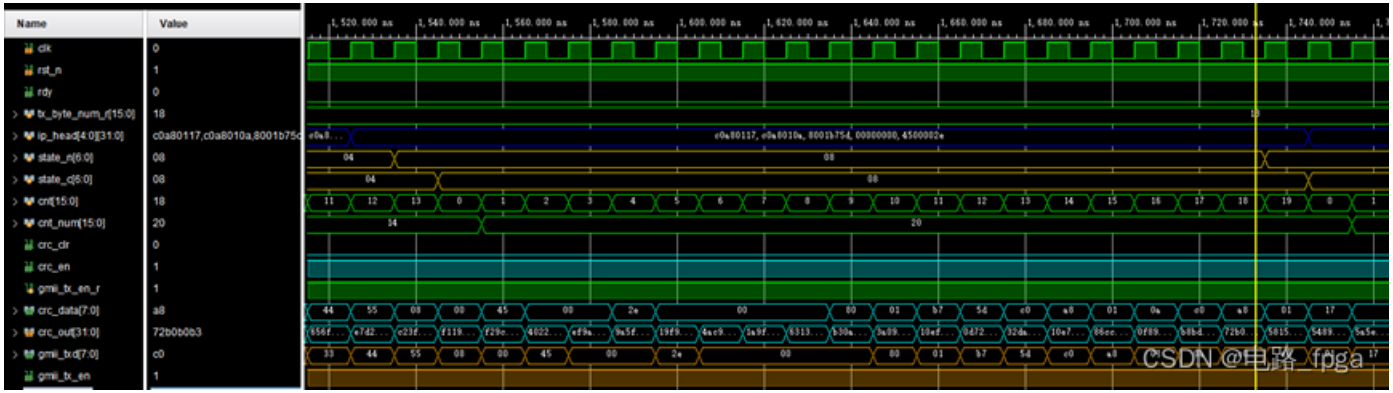

图14 发送IP首部数据

下图是发送ICMP首部存储体中的数据，与上图类似。

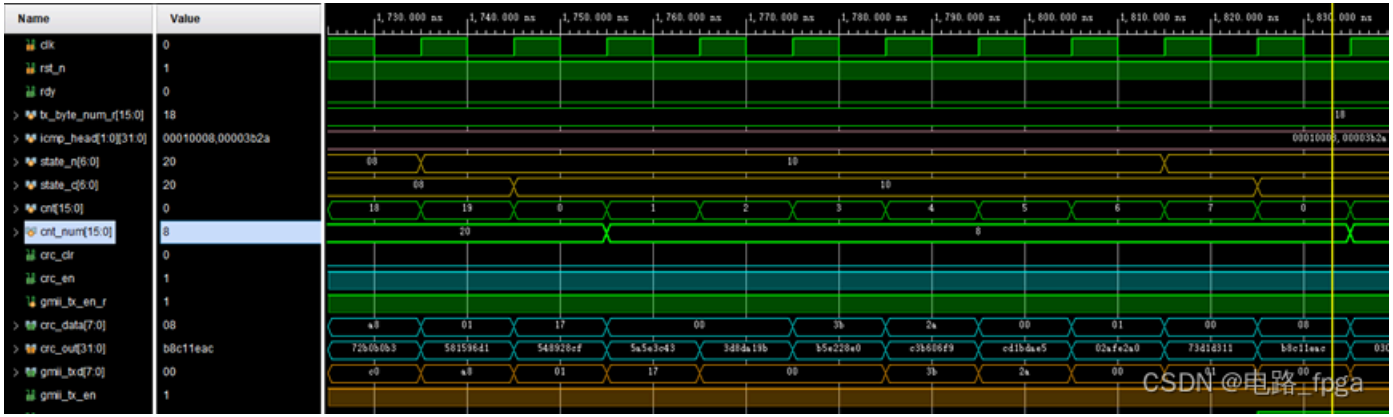

图15 发送ICMP首部数据

发送完ICMP首部数据后，从fifo中读取tx_byte_num_r个数据输出，如下图所示。FIFO读使能与读数据对齐，所以直接使用即可。

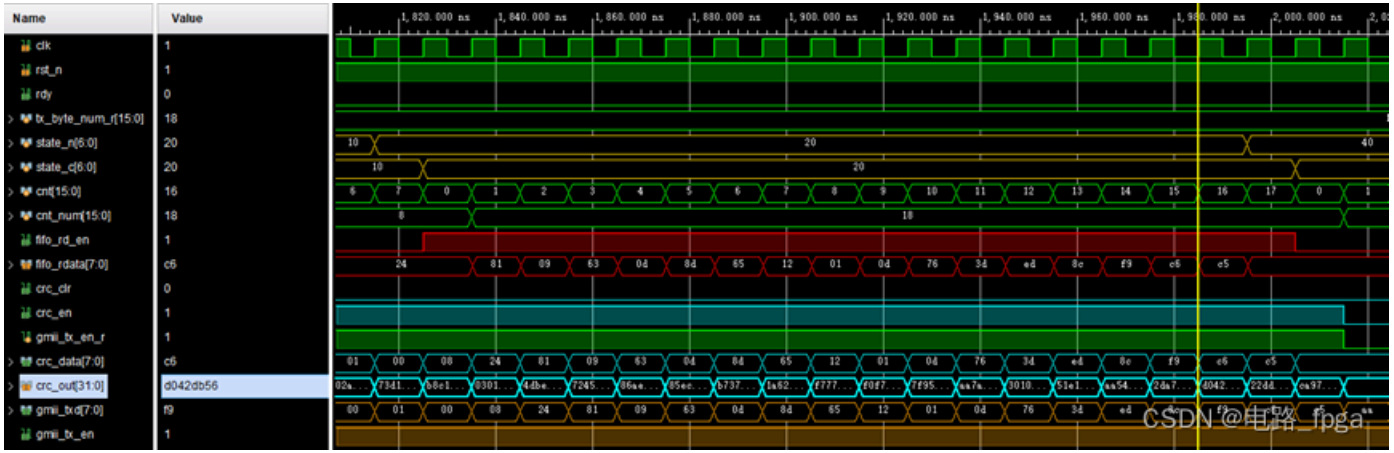

图16 发送ICMP数据

最后就是CRC校验，由于CRC校验模块输出数据会滞后输入数据一个时钟周期，导致需要把crc_data延迟一个时钟周期后在接上CRC校验模块输出的数据，才算正确，这也是为什么需要把crc_data延时一个时钟得到gmii_txd的原因。

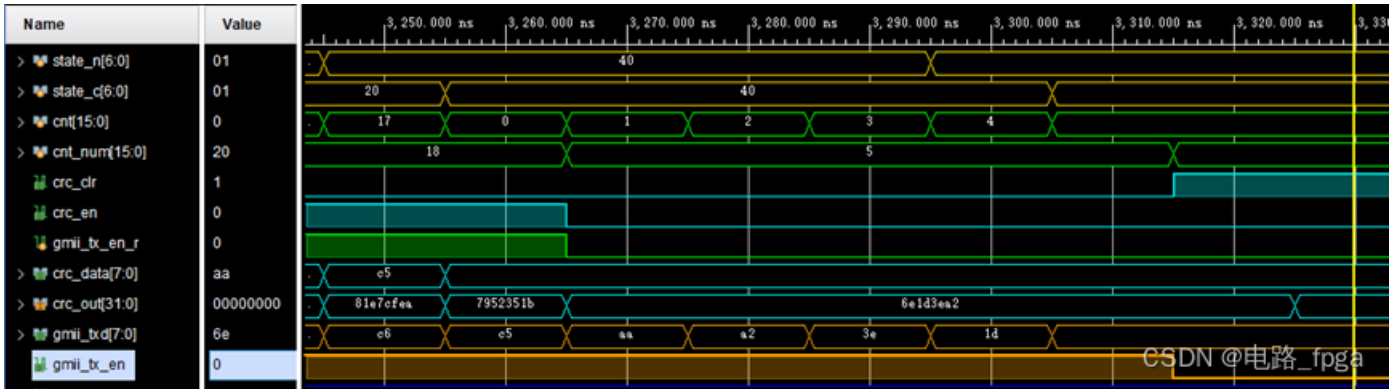

图17 发送CRC数据

ICMP发送模块的设计和仿真到此结束了。

4、FIFO IP设置

FIFO IP设置为超前模式，这样读数据时，读使能和读数据就能直接对齐了，读数据不会滞后读使能，这样用起来更方便。

位宽设置为8位，数据深度设置为1024字节，设置为2048更好。

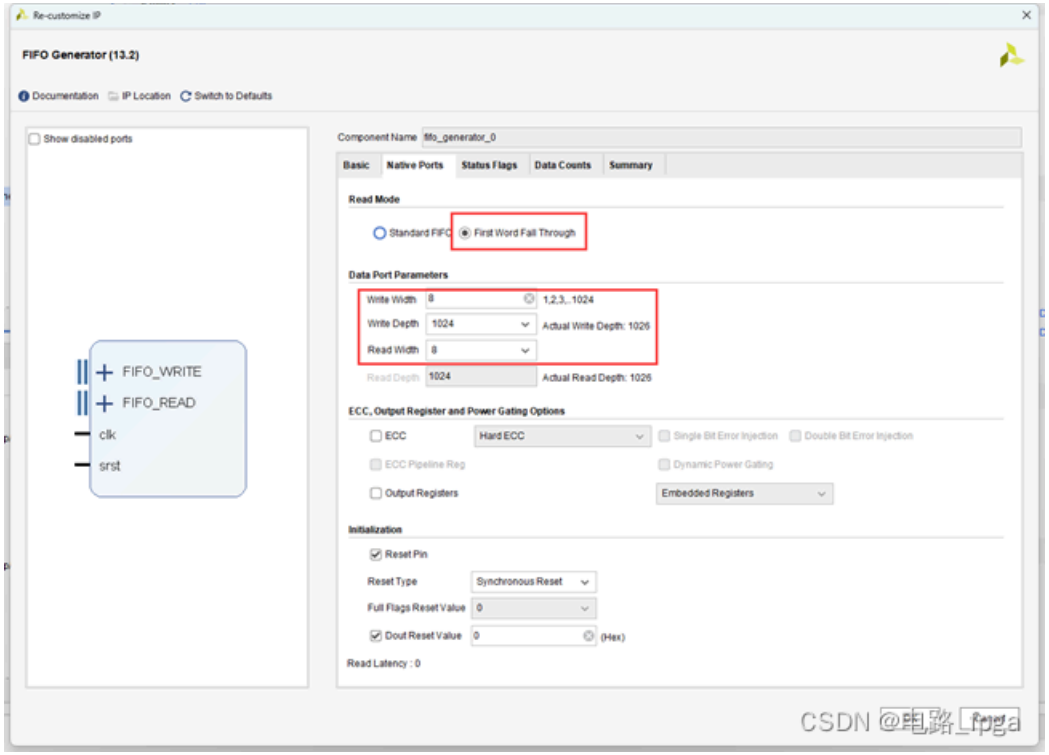

图18 FIFO IP设置

其余设置默认即可，复位采用低电平有效。

5、ARP和ICMP控制模块

arp和icmp的控制模块如下所示，当arp发送模块输出数据且icmp发送模块空闲时，将arp发送模块的输出作为gmii_txd的数据。如果icmp发送模块输出有效数据且arp发送模块空闲时，将icmp发送模块的输出作为gmii_txd的数据。

当arp接收模块接收到数据后，将arp发送模块使能信号拉高，当icmp接收模块到回显请求时，把icmp发送模块的使能信号拉高，实现回显应答。

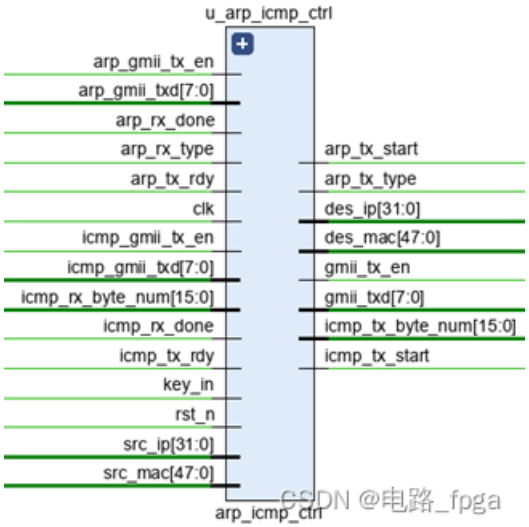

图19 控制模块

该模块的参考代码如下所示：

```
    //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

    //对两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发送模块处于空闲状态，则将ARP相关数据输出。
        else if(arp_gmii_tx_en && icmp_tx_rdy)begin
            gmii_tx_en <= arp_gmii_tx_en;
            gmii_txd <= arp_gmii_txd;
        end//如果ICMP发送模块输出有效数据且ARP发送模块处于空闲，则将ICMP相关数据输出。
        else if(icmp_gmii_tx_en && arp_tx_rdy)begin
            gmii_tx_en <= icmp_gmii_tx_en;
```

```
            gmii_txd <= icmp_gmii_txd;
        end
        else begin
            gmii_tx_en <= 1'b0;
        end
    end
```

由于模块比较简单，所以不再单独仿真，后文直接上板测试即可。

6、上板测试

在工程中加入ILA，综合工程，然后下载到开发板，最后打开wireshark软件，该软件在ARP实现文中已经使用过，不再赘述。将电脑的IP设置为顶层文件的目的IP地址，设置方式在ARP文中也做了详细介绍，不知道怎么设置的可以去看看。

图20 设置电脑的IP

然后以管理员身份选打开命令提示符，然后运行wireshark，把gmii_rx_dv上升沿作为ILA的触发条件，连续抓取32帧数据。在命令提示符中发送ping 192.168.1.10，如下图所示。

图图21 ping指令

Ping指令运行结果如下所示，一般PC会发送4次回显请求，如果四次回显请求都被应答，则认为ping通了，丢失为0。

图22 ping结果

Wireshark抓取的数据报如下所示，粉色信号就是ICMP的回显请求和回显应答数据报。比如第9和10数据报，分别是PC发给FPGA的回显请求和FPGA发送给PC端的回显应答。注意两个报文的标识符和序列号是一致的，这也是分别应答和请求数据报的对应关系。

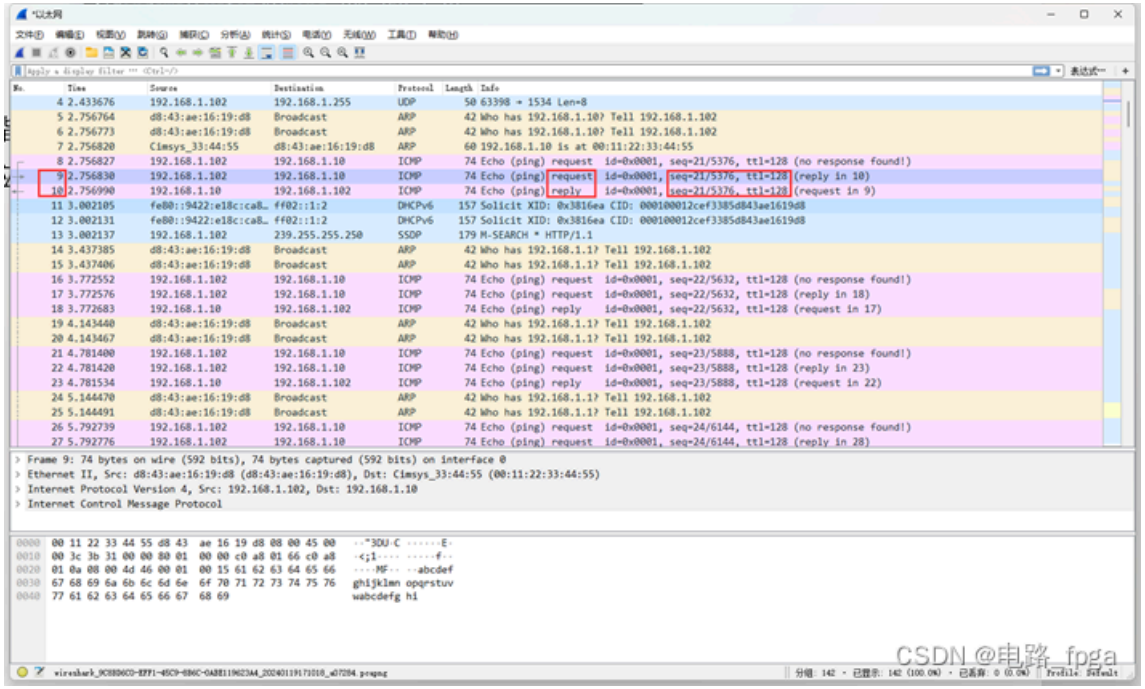

图23 wireshark抓取的回显请求数据报

ILA抓取的PC端发送的回显请求数据报如下所示。

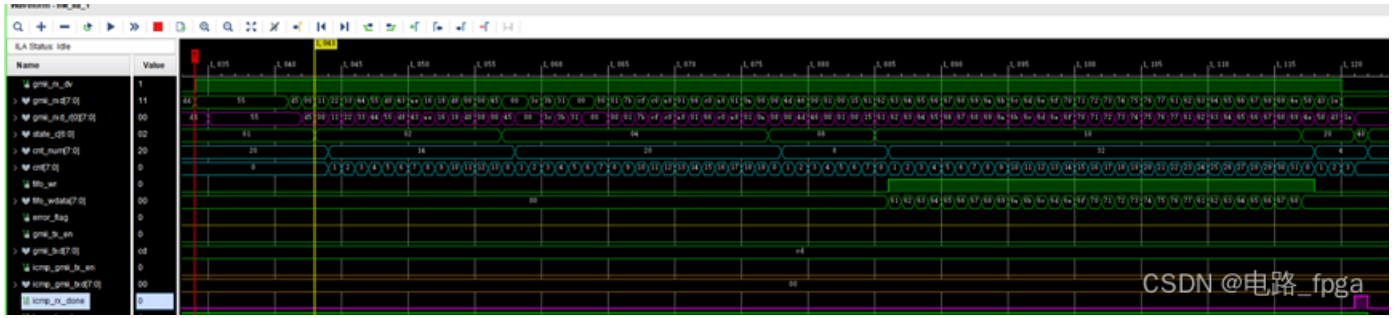

图24 ILA抓取的回显请求数据报

将回显请求数据报的数据段放大，如下图所示，并且与Wireshark的9号数据报的数据段进行对比，可知FPGA接收的数据正确。

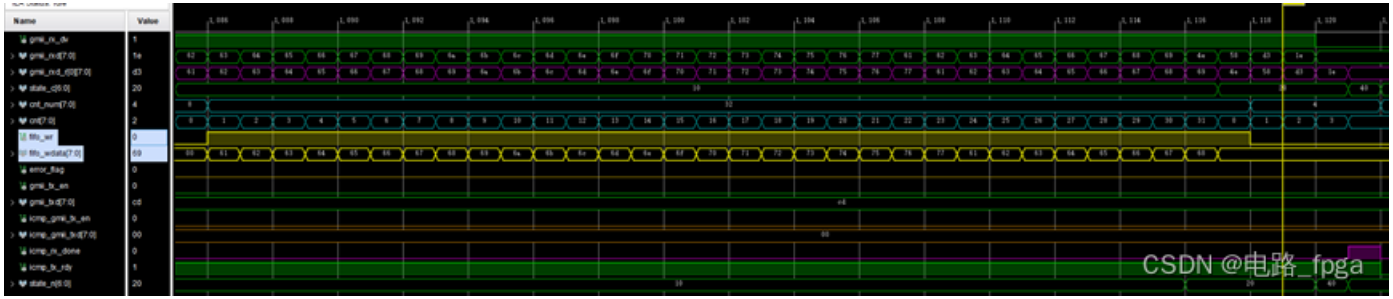

图25 回显请求数据段

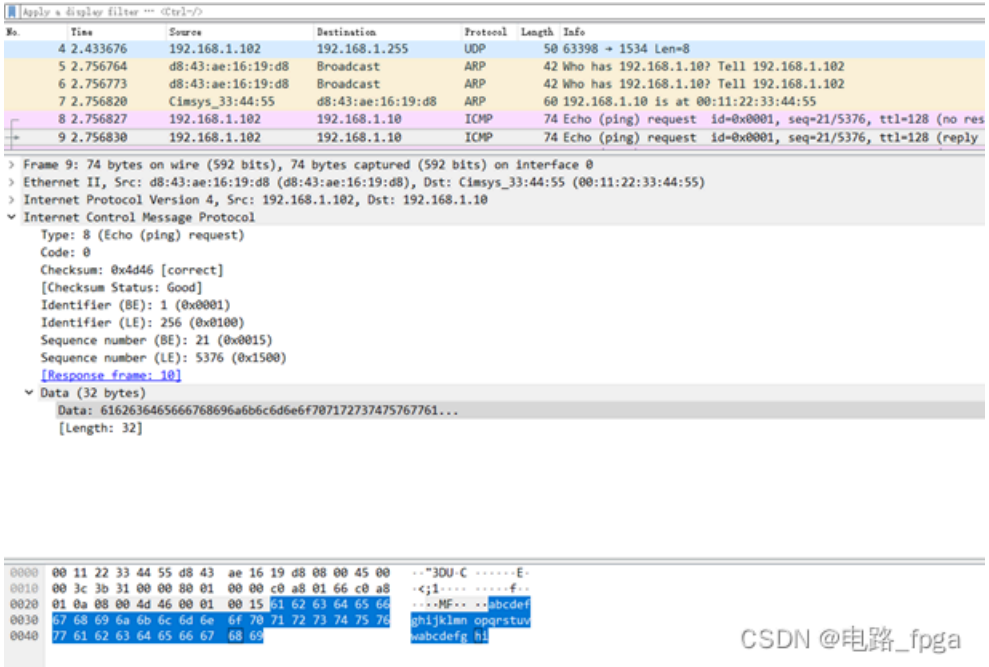

图26 Wireshark回显请求数据报

FPGA在接收到回显请求时，给PC端发出回显应答数据报，ILA抓取该数据报如下图所示。

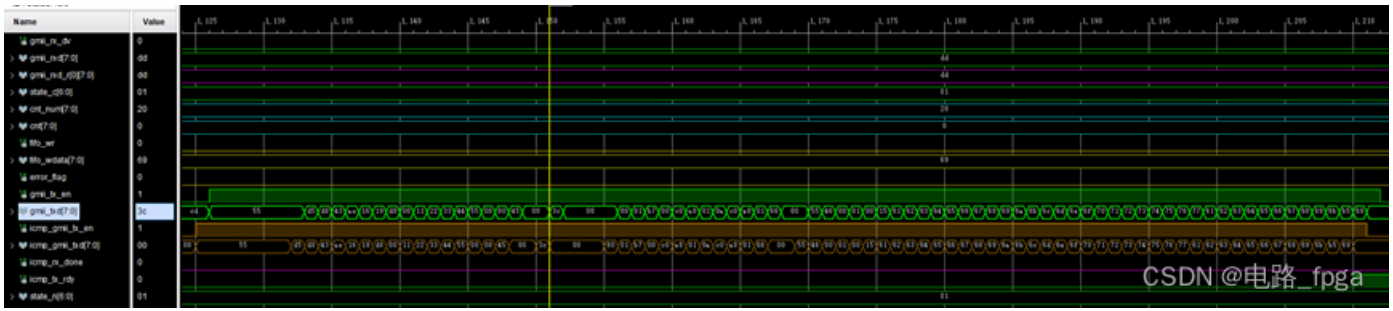

图27 ILA抓取的回显应答数据报

Wireshark抓取的回显应答数据报如下所示，感兴趣的可以使用工程查看。

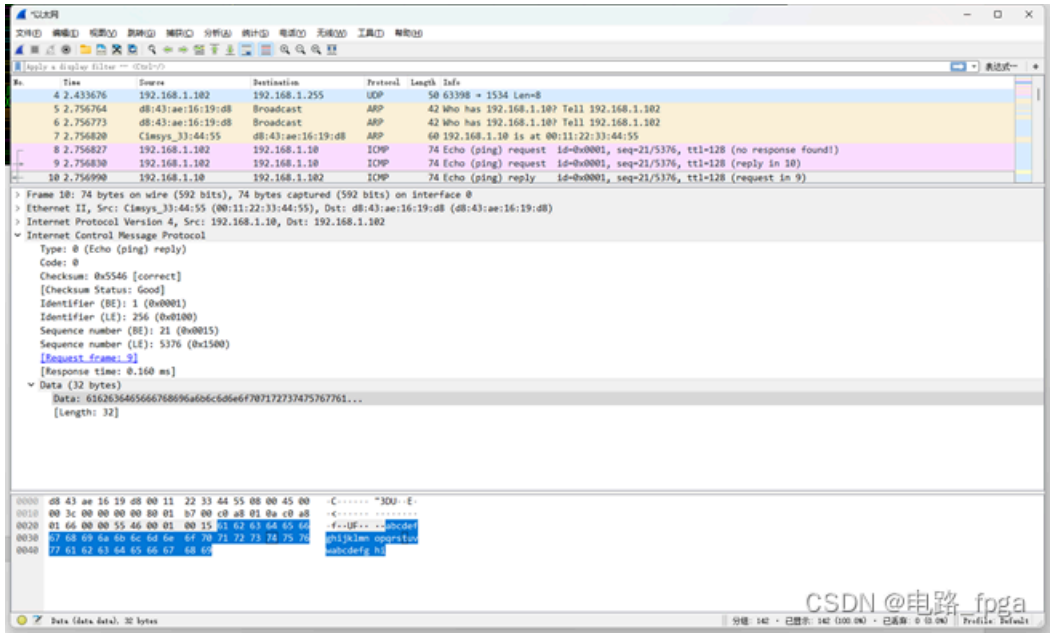

图28 Wireshark回显应答数据报

至于CRC校验这些，与ARP实现的文中是一致的，本文不再赘述，需要了解的可以查看前文。

本工程可以在公众号后台回复“**基于FPGA的ICMP实现**”（不包含引号）获取。

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⭐！