“FPGA+MDIO总线+UART串口=高效读写PHY芯片寄存器！ “（含源代码）

基于FPGA的以太网相关文章导航，点击查看。

1、概述

前文对88E1518芯片的端口芯片及原理图进行了讲解，对MDIO的时序也做了简单的讲解。本文通过 Verilog HDL去实现MDIO，但是88E1518芯片对不同页的寄存器读写需要切换页，无法直接访问寄存器，如果通过代码读写某些固定寄存器的话会比较麻烦。

为了简化调试，所以采用UART串口来控制MDIO的读写，PC端通过UART向 FPGA 发送读写PHY芯片寄存器的指令，FPGA通过MDIO总线从PHY芯片读取指定寄存器地址的数据后，通过UART将读取的数据发送到PC端的串口助手进行显示。

使用这种方式，以后就可以通过串口读写各种MDIO接口的寄存器了，而不再只是对88E1518单个芯片的调试有效了。

顶层模块的框图如图1所示：

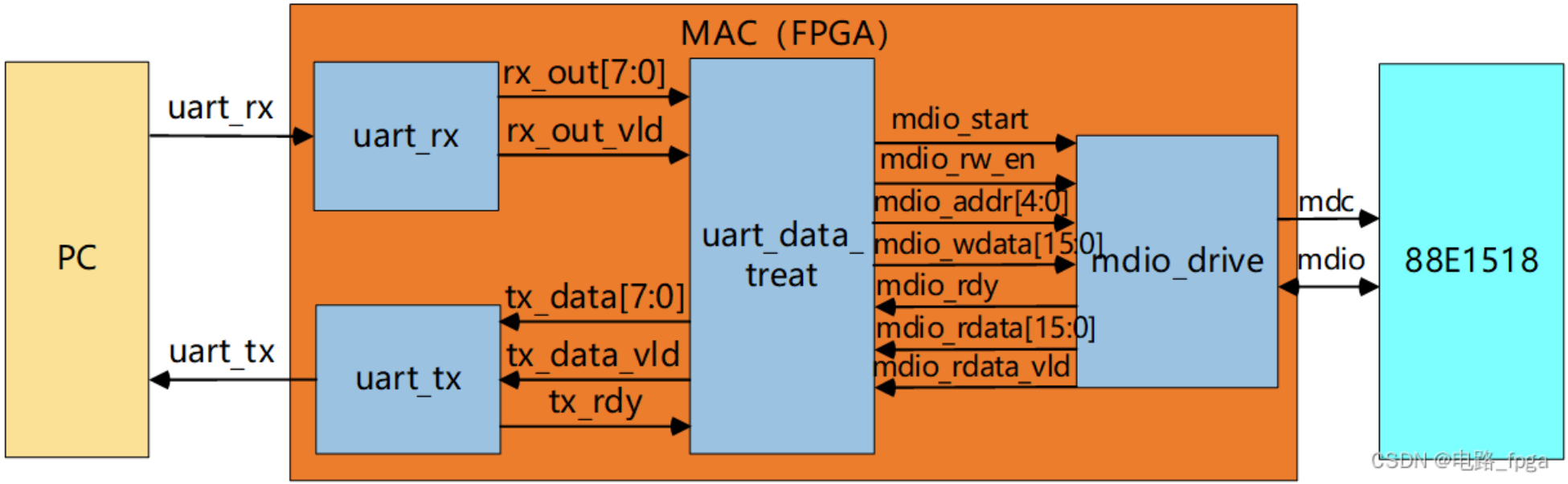

图1 顶层模块框图

对应的端口信号如表1所示（位宽均为1位）：

表1 顶层信号

信号名	I/O	含义
clk	I	系统时钟，100MHz
rst_n	I	系统复位，低电平有效
uart_rx	I	串口接收信号
uart_tx	O	串口发送引脚
mdc	O	MDIO的时钟信号，最大不能超过12MHz。
mdio	IO	MDIO接口双向数据线

参考代码如下所示：

```
module top #(
    parameter           MDIO_DATA_W     =       16              ,//MDIO数据位宽；
    parameter           FCLK            =       100_000_000     ,//系统时钟频率，默认100MHz；
    parameter           FCLKMDC         =       10_000_000      ,//mdc时钟频率，88e1518最大不能超过12MHz；
    parameter           PHY_ADDR        =       5'b0_0000       ,//PHY芯片的地址，88E1518芯片高四位默认为0，最低位由config引脚状态决定。
    parameter           BPS             =       115200          ,//串口波特率；
    parameter           UART_DATA_W     =       8               ,//串口数据位宽;
    parameter           CHECK_W         =       2'b00           ,//校验位，2'b00代表无校验位，2'b01表示奇校验，2'b10表示偶校验，2'b11按无校验处理。
    parameter           STOP_W          =       2'b01            //停止位，2'b01表示1位停止位，2'b10表示2位停止位，2'b11表示1.5位停止位；
)(
    input                                       clk             ,//系统时钟信号；
    input                                       rst_n           ,//系统复位信号，高电平有效；

    output                                      mdc             ,//mdio的时钟信号；
    inout                                       mdio            ,//mdio双向数据信号；
    output                                      mdio_out_en     ,

    input                                       uart_rx         ,//串口输入信号；
    output                                      uart_tx          //串口输出信号；
);
    wire            [UART_DATA_W - 1 : 0]       rx_out          ;
    wire                                        rx_out_vld      ;
    wire            [UART_DATA_W - 1 : 0]       tx_data         ;

```

```
    wire                                    tx_data_vld     ;

    wire                                    tx_rdy          ;
    wire                                    mdio_rdy        ;

    wire                                    mdio_start      ;
    wire                                    mdio_rw_en      ;
    wire                [4 : 0]             mdio_addr       ;
    wire                [MDIO_DATA_W - 1 : 0]   mdio_wdata      ;
    wire                [MDIO_DATA_W - 1 : 0]   mdio_rdata      ;
    wire                                    mdio_rdata_vld  ;

    //例化串口接收模块;
    uart_rx #(
        .FCLK       ( FCLK          ),//系统时钟频率，默认100MHZ;
        .BPS        ( BPS           ),//串口波特率;
        .DATA_W     ( UART_DATA_W   ),//接收数据位数以及输出数据位宽;
        .CHECK_W    ( CHECK_W       ),//校验位，0代表无校验位;
        .STOP_W     ( STOP_W        ) //1位停止位;
    )
    u_uart_rx (
        .clk            ( clk           ),//系统工作时钟100MHZ;
        .rst_n          ( rst_n         ),//系统复位信号，低电平有效;
        .uart_rx        ( uart_rx       ),//UART接口输入信号;
        .rx_out         ( rx_out        ),//数据输出信号;
        .rx_out_vld     ( rx_out_vld    ) //数据有效指示信号;
    );

    //例化串口发送模块;
    uart_tx #(
        .FCLK       ( FCLK          ),//系统时钟频率，默认100MHZ;
        .BPS        ( BPS           ),//串口波特率;
        .DATA_W     ( UART_DATA_W   ),//接收数据位数以及输出数据位宽;
        .CHECK_W    ( CHECK_W       ),//校验位，0代表无校验位;
        .STOP_W     ( STOP_W        ) //1位停止位;
    )
    u_uart_tx (
        .clk            ( clk           ),//系统工作时钟100MHZ;
        .rst_n          ( rst_n         ),//系统复位信号，低电平有效;
        .tx_data        ( tx_data       ),//数据输入信号。

```

```
    .tx_data_vld    ( tx_data_vld   ),//数据有效指示信号，高电平有效。
    .uart_tx        ( uart_tx       ),//uart接口数据输出信号。
    .tx_rdy         ( tx_rdy        ) //模块忙闲指示信号；
);

//例化串口数据处理模块；
uart_data_treat #(
    .UART_DATA_W    ( UART_DATA_W   ),//串口数据位宽;
    .MDIO_DATA_W    ( MDIO_DATA_W   ) //MDIO数据位宽；
)
u_uart_data_treat (
    .clk            ( clk               ),//系统工作时钟100MHZ；
    .rst_n          ( rst_n             ),//系统复位信号，低电平有效；
    .rx_data        ( rx_out            ),
    .rx_data_vld    ( rx_out_vld        ),
    .tx_rdy         ( tx_rdy            ),
    .mdio_rdata     ( mdio_rdata        ),
    .mdio_rdata_vld ( mdio_rdata_vld    ),
    .mdio_rdy       ( mdio_rdy          ),
    .tx_data        ( tx_data           ),
    .tx_data_vld    ( tx_data_vld       ),
    .mdio_start     ( mdio_start        ),
    .mdio_rw_en     ( mdio_rw_en        ),
    .mdio_wdata     ( mdio_wdata        ),
    .mdio_addr      ( mdio_addr         )
);

//例化mdio接口模块；
mdio_drive #(
    .DATA_W     ( MDIO_DATA_W   ),//数据位宽；
    .FCLK       ( FCLK          ),//系统时钟频率，默认100MHz；
    .FCLKMDC    ( FCLKMDC       ),//mdc时钟频率，88e1518最大不能超过12MHz；
    .PHY_ADDR   ( PHY_ADDR      ) //PHY芯片的地址，88E1518芯片高四位默认为0，最低位由config引脚状态决定。
)
u_mdio_drive (
    .clk            ( clk           ),//系统时钟信号；
    .rst_n          ( rst_n         ),//系统复位信号，高电平有效；
    .start          ( mdio_start    ),//开始写入或者读取信号；
    .rw_en          ( mdio_rw_en    ),//读写使能，高电平表示读数据，低电平表示写数据；
    .addr_reg       ( mdio_addr     ),//读写寄存器地址；
```

```
        .wr_data        ( mdio_wdata    ),//需要写入的数据；
        .mdc            ( mdc           ),//mdio的时钟信号；
        .mdio_out_en    ( mdio_out_en   ),//mdio三态门使能信号，高电平有效；用于仿真；
        .rd_data        ( mdio_rdata    ),//读出数据；
        .rd_data_vld    ( mdio_rdata_vld),//读出数据有效指示信号，高电平有效；
        .rdy            ( mdio_rdy      ),//模块忙闲指示信号，高电平表示模块空闲，可以接收上游数据；
        .mdio           ( mdio          ),//mdio双向数据信号；
        .rd_ack         (               )
    );

    ila_0 u_ila_0 (
        .clk        ( clk                   ),//input wire clk
        .probe0     ( mdc                   ),//input wire [0:0]  probe0
        .probe1     ( u_mdio_drive.mdio_in  ),//input wire [0:0]  probe1
        .probe2     ( mdio_out_en           ),//input wire [0:0]  probe2
        .probe3     ( u_mdio_drive.state_c  ),//input wire [4:0]  probe3
        .probe4     ( u_mdio_drive.cnt_data ),//input wire [5:0]  probe4
        .probe5     ( mdio_rdata            ),//input wire [15:0]  probe5
        .probe6     ( mdio_rdata_vld        ),//input wire [0:0]  probe6
        .probe7     ( u_mdio_drive.start    ),//input wire [0:0]  probe7
        .probe8     ( u_mdio_drive.rdy      ),//input wire [0:0]  probe8
        .probe9     ( mdio_wdata            ),//input wire [15:0]  probe9
        .probe10    ( mdio_addr             ),//input wire [4:0]  probe10
        .probe11    ( mdio_rw_en            ),//input wire [0:0]  probe11
        .probe12    ( uart_rx               ),//input wire [0:0]  probe12
        .probe13    ( rx_out                ),//input wire [7:0]  probe13
        .probe14    ( rx_out_vld            ),//input wire [0:0]  probe14
        .probe15    ( tx_data               ),//input wire [7:0]  probe15
        .probe16    ( tx_data_vld           ) //input wire [0:0]  probe16
    );

    endmodule
```

对应的TestBench代码如下：

```
`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   MDIO_DATA_W     =   16          ;
    parameter   FCLK            =   100_000_000 ;
    parameter   FCLKMDC         =   10_000_000  ;
    parameter   PHY_ADDR        =   5'b0_0000   ;
    parameter   BPS             =   115200      ;
    parameter   UART_DATA_W     =   8           ;
    parameter   CHECK_W         =   2'b00       ;
    parameter   STOP_W          =   2'b01       ;
    localparam  BPS_CNT         =   FCLK/BPS    ;//波特率对应时钟数，不用手动修改该参数；

    reg                         clk             ;//系统时钟，默认100MHz；
    reg                         rst_n           ;//系统复位，默认高电平有效；
    reg                         uart_rx         ;

    wire                        uart_tx         ;
    wire                        mdc             ;
    wire                        mdio            ;
    wire                        mdio_out_en     ;

    reg  mdio_out;
    assign mdio = (~mdio_out_en) ? mdio_out : 1'bz;

    top #(
        .MDIO_DATA_W    ( MDIO_DATA_W   ),
        .FCLK           ( FCLK          ),
        .FCLKMDC        ( FCLKMDC       ),
        .PHY_ADDR       ( PHY_ADDR      ),
        .BPS            ( BPS           ),
        .UART_DATA_W    ( UART_DATA_W   ),
        .CHECK_W        ( CHECK_W       ),
        .STOP_W         ( STOP_W        )
    )
    u_top (
        .clk            ( clk           ),
        .rst_n          ( rst_n         ),
        .uart_rx        ( uart_rx       ),

```

```
    .mdc            ( mdc           ),
    .mdio_out_en    ( mdio_out_en   ),
    .uart_tx        ( uart_tx       ),
    .mdio           ( mdio          )
);

//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
    rst_n = 1;uart_rx = 0;mdio_out =0;
    #1;rst_n = 0;//开始时复位10个时钟;
    #(RST_TIME*CYCLE);
    rst_n = 1;
    mdio_rw_task(1'b1,5'd17,0);//读17号寄存器数据;
    mdio_rw_task(1'b0,5'd22,8'd1);//向22号寄存器写入1
    repeat(20)begin
        mdio_rw_task(1'b1,({$random} % 32),0);//读寄存器数据;
        mdio_rw_task(1'b0,({$random} % 32),({$random} % 256));//写寄存器;
    end
    $stop;//停止仿真;
end

task mdio_rw_task(
    input                           rw_flag ,//mdio读写标志，高电平表示读操作，低电平表示写操作;
    input   [4 : 0]                 addr    ,//mdio读写寄存器地址信号;
    input   [MDIO_DATA_W - 1 : 0]   wdata    //mdio进行写操作时，需要写入的地址;
);
    reg                 rw_flag_r;
    begin
        rw_flag_r = rw_flag;
        uart_rx_task(8'h5a);//首先发送帧头，0x5a
        uart_rx_task({7'd0,rw_flag_r});//发送读写操作;
        uart_rx_task({3'd0,addr});//发送读写寄存器地址;
        if(~rw_flag_r)begin//mdio进行写操作时，需要写入的地址;
            uart_rx_task(wdata[15:8]);
```

```
                uart_rx_task(wdata[7:0]);
            end
        end
    endtask

    //模拟串口发送函数，1位起始位，1位停止位，无校验位，8位数据，先发低位；
    task uart_rx_task(
        input   [UART_DATA_W-1:0]   data //串口待发送数据；
    );
        integer i;//用于控制循环次数；
        begin
            @(posedge clk);//延迟一个时钟后发送起始位；
            uart_rx = 1'b0;
            repeat(BPS_CNT) @(posedge clk);//延迟BPS_CNT个时钟；
            for(i=0 ; i<8 ; i=i+1)begin
                uart_rx = data[i];
                repeat(BPS_CNT) @(posedge clk);//延迟BPS_CNT个时钟；
            end
            if(CHECK_W == 2'b01)begin
                uart_rx = ~(^data);//奇校验时，发送数据；
                repeat(BPS_CNT) @(posedge clk);//延迟BPS_CNT个时钟；
            end
            else if(CHECK_W == 2'b10)begin
                uart_rx = (^data);//偶校验时，发送数据；
                repeat(BPS_CNT) @(posedge clk);//延迟BPS_CNT个时钟；
            end
            @(posedge clk);//延迟一个时钟后发送停止位；
            uart_rx = 1'b1;
            if(STOP_W == 2'b01)//1位停止位；
                repeat(BPS_CNT) @(posedge clk);//延迟BPS_CNT个时钟；
            else if(STOP_W == 2'b10)//2位停止位；
                repeat(2*BPS_CNT) @(posedge clk);//延迟2*BPS_CNT个时钟；
            else if(STOP_W == 2'b11)//1.5位停止位；
                repeat(BPS_CNT*3/2) @(posedge clk);//延迟1.5*BPS_CNT个时钟；
        end
    endtask

endmodule
```

2、串口数据处理模块

至于uart串口接收模块和uart串口发送模块，前文已经对UART全模式接收和发送的代码设计进行过详细讲解，不在赘述。

uart_rx模块接收数据后，需要对数据进行判断，确定对应的数据是对寄存器进行读操作还是写操作，以及是接收的数据是地址还是寄存器的数据？读操作只需要发送读指示信号和寄存器地址即可，而写还需要发送写入HPY芯片内部寄存器的16位数据，所以读写不同，串口发送的数据长度用过也不同。

因此此处规定一下PC端UART发送数据的格式，没发送一次完整的读写操作码及数据称为一帧数据。帧的起始位为8’h5A，FPGA检测到PC端发了8’h5A，表示后面的数据就是对PHY芯片进行读写的操作码和数据。

帧起始后面跟1字节的读写指示信号，第0位为高电平表示进行读操作，低电平表示对PHY内部寄存器进行写操作，其余7位数据无效。

之后PC端发送需要读写PHY芯片的寄存器地址，由于PHY芯片的寄存器地址为5位，所以发送的地址数据只有低5位有效，高3位无效。

如果是读取PHY芯片内部寄存器数据，那么到此结束，等待数据返回即可。如果是要写入数据到PHY芯片内部寄存器，那么还需要发送两字节的写入数据，先发送需要写入数据的高8位，后发送低8位数据。

上述就是我们自己为了方便设置的通信格式，实际上可以简化，可以把读写方式跟寄存器地址合并为1字节数。帧起始码也可以去掉，根据第一字节某些无效位进行判断，但是为了更加直观的查看数据，就不进行简化了。读写PHY芯片寄存器的串口助手指令格式如下所示：

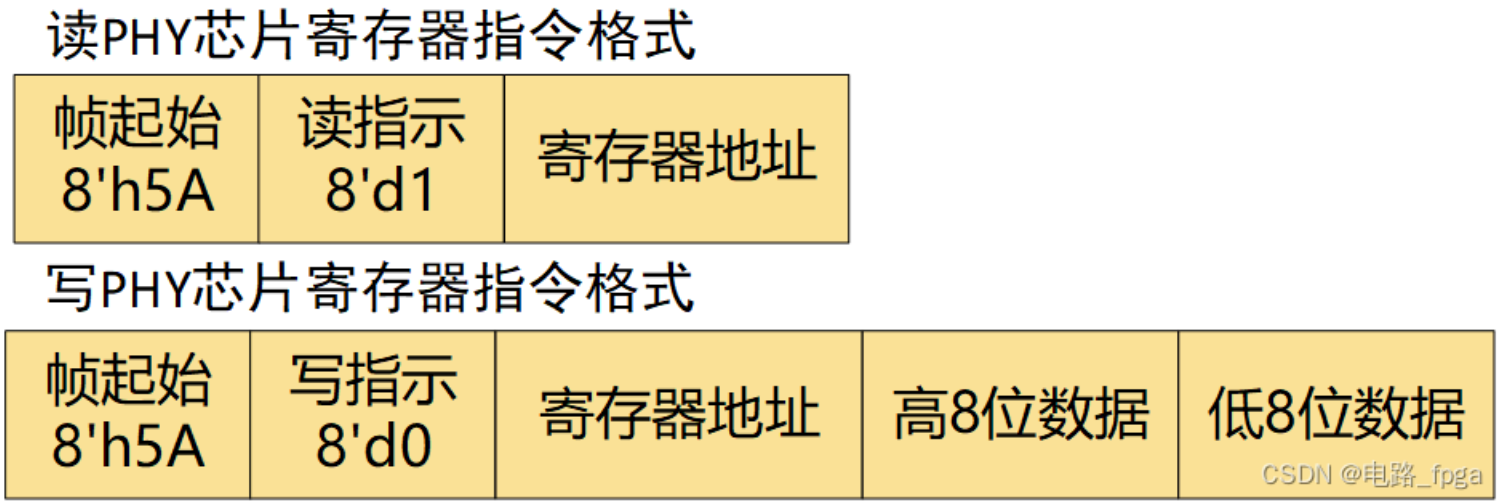

图2 串口数据读写PHY寄存器的指令格式

上面都是对该协议的讲述，本模块就是来对该协议进行解析，并且把MDIO驱动模块从PHY芯片指定寄存器地址读取的数据输出到串口发送模块，将读取的数据最终发送到电脑的串口调试助手上。

该模块的端口信号如表2所示：

表2 数据解析模块端口信号

信号名	位宽	IO	含义
clk	1	I	系统时钟，100MHz。
rst_n	1	I	系统复位，低电平有效。
Rx_out	8	I	串口接收数据。
Rx_out_vld	1	I	串口接收数据有效指示信号。
Tx_data	8	O	串口需要发送的数据
Tx_data_vld	1	O	串口发送数据有效指示信号。
Tx_rdy	1	I	串口发送模块空闲指示信号。
Start	O	1	开始读写PHY寄存器，高电平有效。
rw_en	O	1	高电平表示读取PHY内部寄存器数据，低电平表示往PHY内部寄存器写入数据。
Addr	O	5	读写寄存器地址。
Wdata	O	16	需要写入PHY寄存器的数据。
Rdata	I	16	从PHY内部寄存器读出的数据。
Rdata_vld	I	1	读出的数据有效指示信号。
Mdio_rdy	I	1	MDIO驱动模块空闲指示信号，高电平有效。

该模块的代码比较简单，此处做简单介绍，模块内部代码分为两部分，一部分对接收到PC端发送的UART数据进行检测，检测到帧头8'h5A后，就启用一个计数器对后面接收的串口数据进行计数，第2字节的最低位能够表示此次进行读操作还是写操作，将最低位作为rw_en信号输出，计数器的长度也与该位数据取值有关，高电平表示读操作，那这一帧数据除去帧头就2字节，此时计数器最大值应该为2-1，低电平表示写操作，除去帧头应该有4字节数据，那么计数器的最大值应该是4-1。

然后就是接收第3字节数据的低5位数据作为PHY芯片寄存器地址，如果是读操作，此时就应该拉高start。如果是写操作，还需要接收2字节数据作为wdata输出，才能拉高start信号。但实际上start信号还与MDIO驱动模块是否空闲有关，如果此时MDIO处于工作状态，则等MDIO驱动模块空闲后在拉高start信号。

本模块的另一部分功能就是将MDIO驱动模块从PHY内部寄存器读出的16位数据发送转换成串口发送模块的8位数据，然后传输给电脑。这部分比较简单，因为串口模块需要发送指令，驱动模块才会进行读操作，所以MDIO驱动模块读出数据是有限的，而且PC发送指令到MDIO驱动模块读出数据所需要的时间大于把MDIO读出数据通过串口发送到PC的时间，所以就不会存在数据丢失，不需要使用FIFO、RAM等存储结构暂存读出的数据。

当接收到MDIO读取的数据，并且串口发送模块空闲时，先把读取数据的高8位发送，发送完成后在发送低8位数据。这里会用到一个计数器来记录本次发送的是高位数据还是低位数据。

总体思路就是这样，参考代码如下所示：

```
module uart_data_treat #(
    parameter           UART_DATA_W             =   8               ,//uart传输数据位宽；
    parameter           MDIO_DATA_W             =   16               //mdio读写数据位宽；
)(
    input                                           clk             ,//系统时钟信号；
    input                                           rst_n           ,//系统复位信号，低电平有效；

    input               [UART_DATA_W - 1 : 0]       rx_data         ,//uart接收到的数据；
    input                                           rx_data_vld     ,//uart接收到的数据有效指示信号，高电平有效；
    input                                           tx_rdy          ,//uart发送模块空闲指示信号，高电平有效；
    output  reg         [UART_DATA_W - 1 : 0]       tx_data         ,//uart需要发送的数据；
    output  reg                                     tx_data_vld     ,//uart需要发送数据有效指示信号，高电平有效；

    input               [MDIO_DATA_W - 1 : 0]       mdio_rdata      ,//mdio读取的数据；
    input                                           mdio_rdata_vld  ,//mdio读取的数据有效指示信号，高电平有效；
    input                                           mdio_rdy        ,//mdio接口模块空闲指示信号，高电平有效；
    output  reg                                     mdio_start      ,//mdio开始进行读写操作信号；
    output  reg                                     mdio_rw_en      ,//mdio接口模块执行读写操作，高电平表示读操作，低电平表示写操作；
    output  reg         [MDIO_DATA_W - 1 : 0]       mdio_wdata      ,//mdio在写操作时写入寄存器的数据；
    output  reg         [4 : 0]                     mdio_addr        //mdio读写寄存器的地址；
);
    reg                                             rx_done         ;//
    reg                                             uart_rx_flag    ;//
    reg                 [1 : 0]                     cnt_rx          ;//
    reg                 [2 : 0]                     cnt_rx_num      ;
    reg                 [MDIO_DATA_W - 1 : 0]       mdio_rdata_r    ;//
    reg                                             uart_tx_flag    ;//
    reg                                             cnt_tx          ;//

    wire                                            add_cnt_tx      ;
    wire                                            end_cnt_tx      ;
```

```
wire                                        add_cnt_rx      ;
wire                                        end_cnt_rx      ;

/************** 处理FPGA通过接收到PC端产生的数据开始 ****************/
//规定一下串口数据的格式，首先得有个帧头8‘h5a，然后需要发送读写寄存器的地址和读写操作，
//第二字节的最低位表示读写操作，高电平表示mdio读操作，低电平表示mdio进行写操作。
//同时可以根据该位判断该帧数据长度，如果是读操作，则后面只有1字节的寄存器地址，如果是写操作，则地址后面应该还有2字节写数据；
//第三字节的低5位表示读写寄存器的地址，高三位无效，可随意设置；
//如果是读操作，则没有其余操作了，等待后文模块将读出数据通过串口发送到PC即可。
//如果是写操作，还需要接收2字节的写数据，之后才能产生start信号。

//标志信号，初始值为0，当检测到帧头8'h5a时拉高，当接收完一帧数据时拉低；
//该操作表示，如果PC串口发送数据过快，则只接收最开时接收到的一帧数据，发送完成后在接收其余数据。
always@(posedge clk)begin
    if(rst_n==1'b0)begin//初始值为0;
        uart_rx_flag <= 1'b0;
    end
    else if(end_cnt_rx)begin//接收完一帧数据；
        uart_rx_flag <= 1'b0;
    end//当接收到数据帧头且没有已经接收但没有发出的数据时拉高；
    else if(rx_data_vld && (rx_data == 8'h5a) && (~rx_done))begin
        uart_rx_flag <= 1'b1;
    end
end

//接收数据寄存器，当uart_rx_flag信号拉高后接收到有效数据时加一。
//注意帧头并不会被计数器计数，所以在计算计数器接收数据个数时不包括帧头。
//当把读写操作、读写地址、写数据接收完成时清零。
always@(posedge clk)begin
    if(rst_n==1'b0)begin//
        cnt_rx <= 0;
    end
    else if(add_cnt_rx)begin
        if(end_cnt_rx)
            cnt_rx <= 0;
        else
            cnt_rx <= cnt_rx + 1;
    end
end


```

```
    assign add_cnt_rx = uart_rx_flag && rx_data_vld;//当uart_rx_flag信号有效且接收数据有效时拉高；
    assign end_cnt_rx = add_cnt_rx && cnt_rx == cnt_rx_num - 1;//当接收到指定个有效数据时拉高，表示接收数据完成。

    //根据读写操作判断计数器的长度，从而实现接收数据的长度变化。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为4;
            cnt_rx_num <= 3'd4;
        end
        else if(cnt_rx==0 && add_cnt_rx)begin
            if(~rx_data[0])//当接收到的表示进行写操作时，表示总共需要接收4字节数据。
                cnt_rx_num <= 3'd4;
            else//否则表示接收的是读操作的数据，则只需要接收2字节数据；
                cnt_rx_num <= 3'd2;
        end
    end
    //mdio进行读写操作的指示信号，高电平表示读。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdio_rw_en <= 1'b0;
        end//接收到的第一字节最低位数据。
        else if(cnt_rx==0 && add_cnt_rx)begin
            mdio_rw_en <= rx_data[0];
        end
    end
    //mdio读写操作的寄存器地址。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdio_addr <= 5'd0;
        end//接收到的第二字节数据低5位是读写寄存器地址。
        else if(cnt_rx==1 && add_cnt_rx)begin
            mdio_addr <= rx_data[4:0];
        end
    end

    //将串口发送的2字节数据进行拼接，作为mdio的写数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdio_wdata <= 16'd0;
        end
        else if(add_cnt_rx)begin

```

```
            if(cnt_rx==2)//串口先发高八位数据；
                mdio_wdata <= {rx_data , mdio_wdata[7:0]};
            else if(cnt_rx==3)//再发低八位数据；
                mdio_wdata <= {mdio_wdata[15:8] , rx_data};
        end
    end

    //接收串口一帧数据的指示信号。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            rx_done <= 1'b0;
        end
        else if(mdio_start)begin//当mdio模块将接收到的数据发送，所以拉低；
            rx_done <= 1'b0;
        end
        else if(end_cnt_rx)begin//当
            rx_done <= 1'b1;
        end
    end

    //mdio读写寄存器的开始信号，当mdio发送数据模块空闲且接收到串口发送的完整数据时拉高，其余时间拉低。
    always@(posedge clk)begin
        if(rx_done && mdio_rdy)begin
            mdio_start <= 1'b1;
        end
        else begin
            mdio_start <= 1'b0;
        end
    end
    /************* 处理FPGA通过接收到PC端产生的数据结束 ****************/

    /************** 把mdio读取数据通过串口发送给PC开始 *****************/
    //由于mdio每次最多读取2字节数据，且每次读取数据都需要PC端先通过串口设置读写指令及读写寄存器地址，还有帧头数据。
    //所以串口发送和接收波特率相同的情况下，FPGA给PC端发送数据时比较空闲的，数据比较少，不需要用FIFO之类的做缓冲。
    //直接使用寄存器暂存即可，不会丢失mdio读出的数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdio_rdata_r <= {{MDIO_DATA_W}{1'b0}};
        end//当mdio读出数据有效且没有未发送数据时将数据暂存；
        else if(mdio_rdata_vld && ~uart_tx_flag)begin

```

```
            mdio_rdata_r <= mdio_rdata;
        end
    end

    //有未发送数据指示信号，初始值为0，当mdio读取有效数据时拉高。
    //当接收的数据发送完成时拉低。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            uart_tx_flag <= 1'b0;
        end
        else if(end_cnt_tx)begin
            uart_tx_flag <= 1'b0;
        end
        else if(mdio_rdata_vld)begin
            uart_tx_flag <= 1'b1;
        end
    end

    //发送数据字节数计数器，初始值为0，当有未发送数据且下游串口发送模块空闲时加一。
    //当发送完mdio读取的2字节数据时清零。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//
            cnt_tx <= 0;
        end
        else if(add_cnt_tx)begin
            if(end_cnt_tx)
                cnt_tx <= 0;
            else
                cnt_tx <= cnt_tx + 1;
        end
    end

    assign add_cnt_tx = uart_tx_flag && tx_rdy;//当有未发送数据且串口发送模块空闲时拉高。
    assign end_cnt_tx = add_cnt_tx && cnt_tx == 2 - 1;//当发送完2字节数据时拉高；

    //串口发送数据，先发送高八位数据，后发送低8位数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            tx_data <= {{UART_DATA_W}{1'b0}};
        end

```

```
        else if(add_cnt_tx)begin
            if(cnt_tx==0)//先发送高八位数据；
                tx_data <= mdio_rdata_r[15:8];
            else//后发送低8位数据；
                tx_data <= mdio_rdata_r[7:0];
        end
    end

    //生成串口发送数据有效指示信号，当发送数据时拉高，其余时间拉低。
    always@(posedge clk)begin
        if(add_cnt_tx)begin//初始值为0;
            tx_data_vld <= 1'b1;
        end
        else begin
            tx_data_vld <= 1'b0;
        end
    end
    /************** 把mdio读取数据通过串口发送给PC结束 *****************/

endmodule
```

该模块的仿真结果如下所示，当接收到PC端发送的帧头8'h5A后，计数器开始工作，对后续数据进行解析。

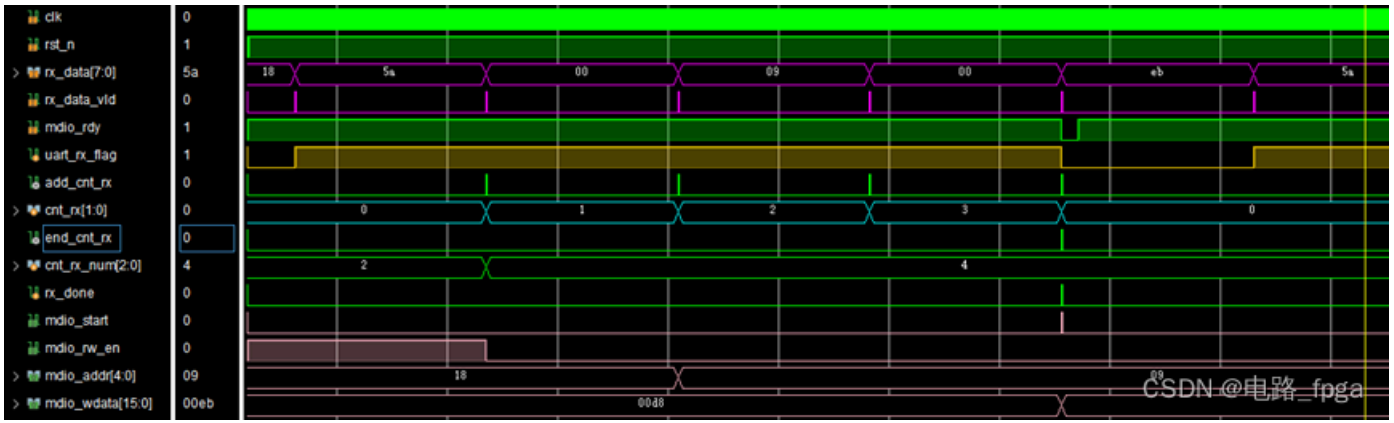

图3 发送写指令的整体时序

上图检测到帧头8’h5A，然后下一字节数据最低位为0表示写指令，此时需要接收4字节数据，cnt_rx_num则赋值为4，然后依次接收寄存器地址和数据，最后将接收数据结尾的仿真截图放大，得到图4。

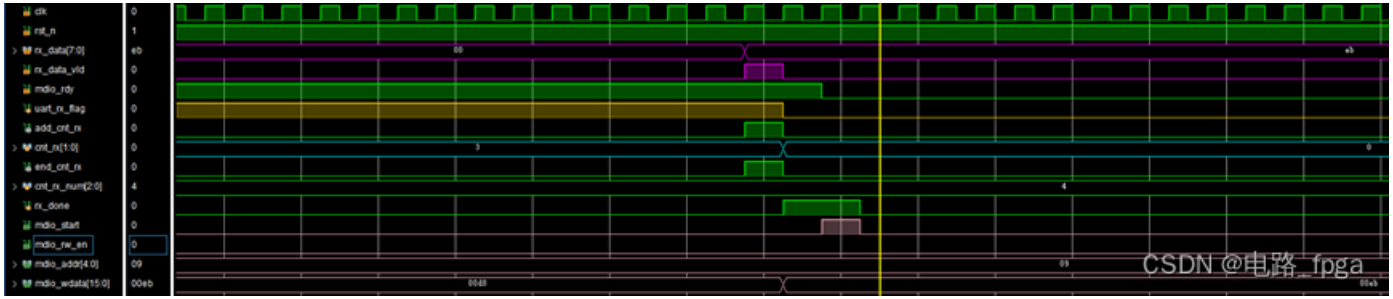

图4 发送写指令的结束部分

下图是PC端发送读寄存器指令的时序，首先检测到帧头8’h5A，下一字节的最低位为1，表示进行读操作，除去帧头只有2字节数据，所以计数器的最大值cnt_rx_num为2。

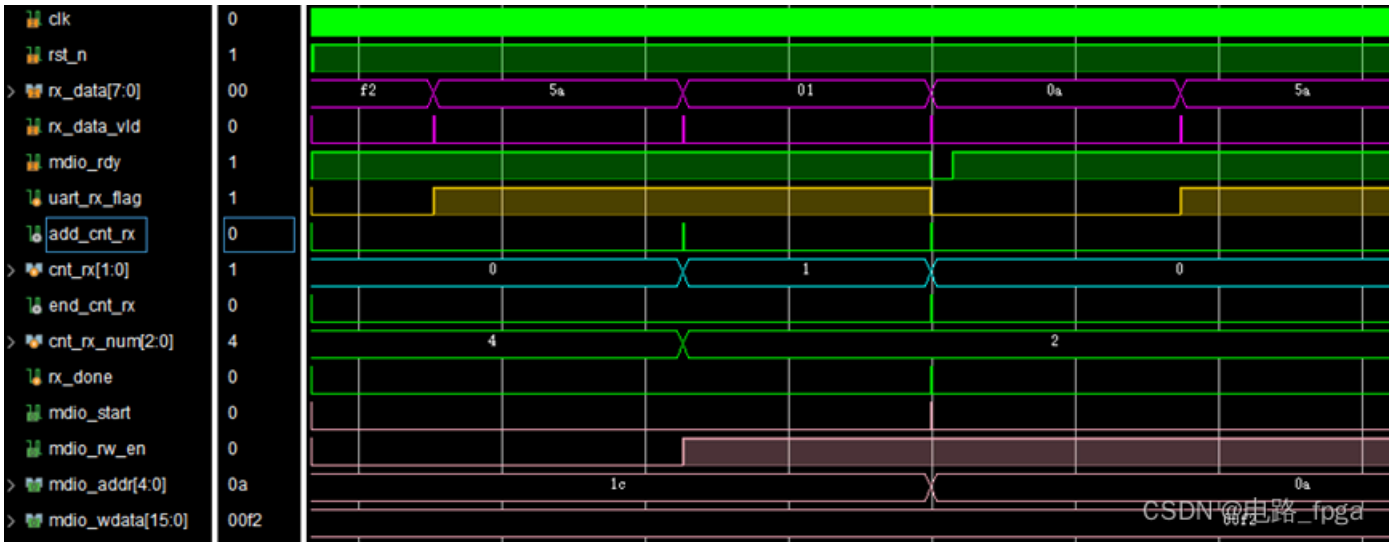

图5 发送读指令的整体时序

将开始部分放大后如下图所示，

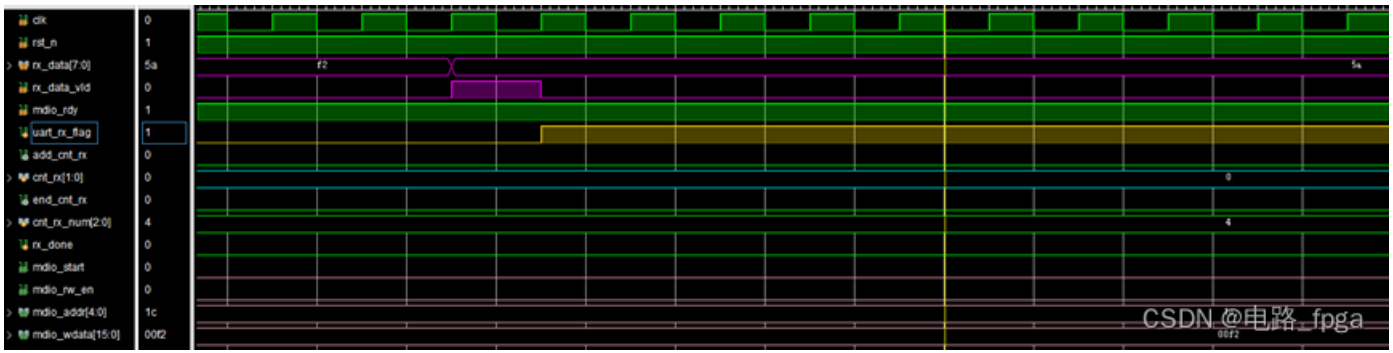

图6 检测到帧头8’h5A

结束部分的时序如下图所示，计数器和标志信号这些都要清零处理，将start信号拉高一个时钟周期。

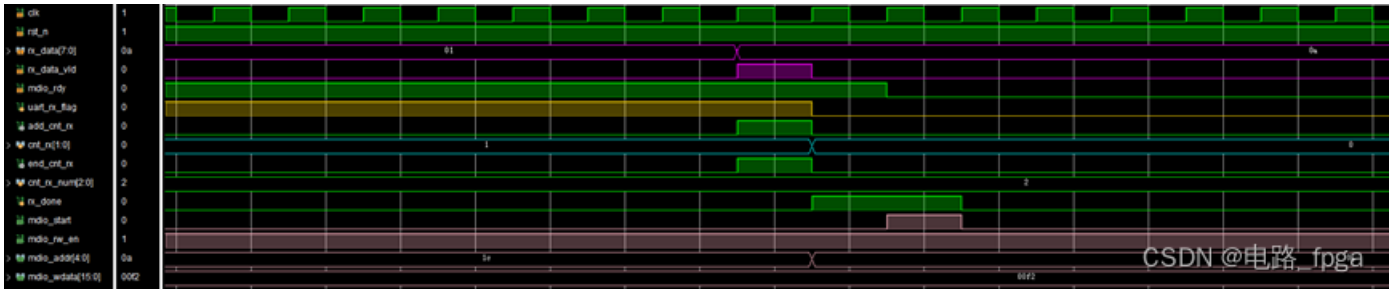

图7 一帧数据结束解析部分

另一部分的仿真功能此处没有做过多处理，因为MDIO从机的程序并没有进行编写，所以仿真时返回的数据始终为0，所以这部分仿真看起来比较简单，如下图所示：

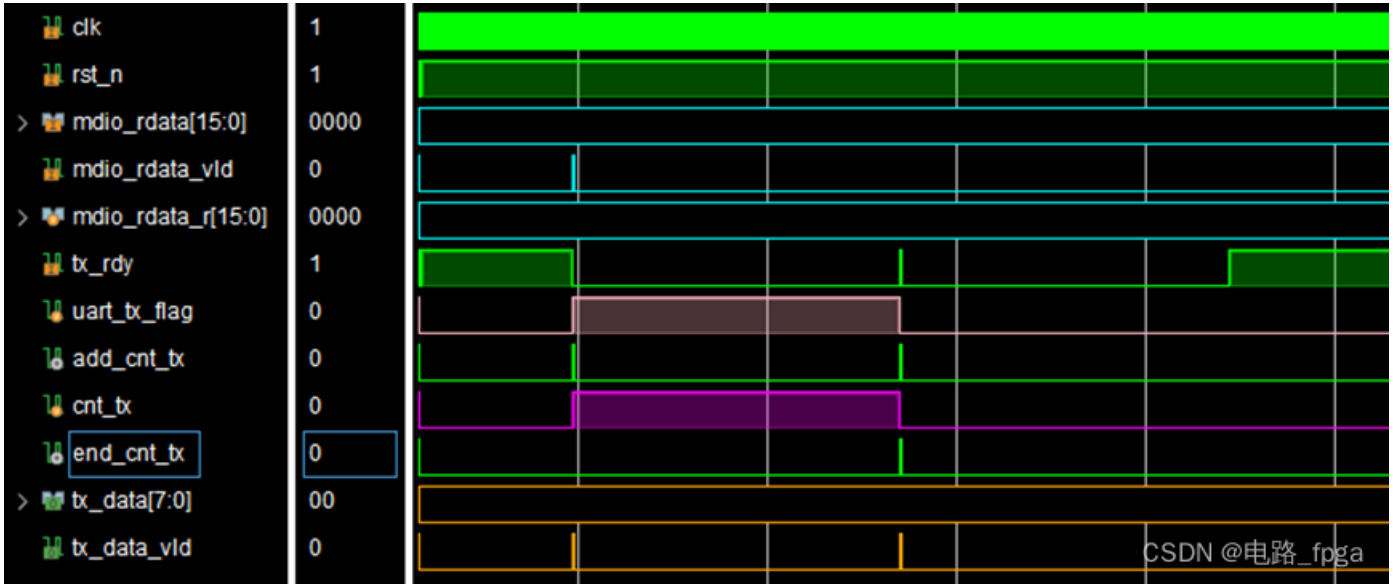

图8 将读取数据发送

当从PHY芯片的寄存器读取到数据mdio_rdata_vld拉高，并且串口发送模块空闲（tx_rdy为高电平）时，将flag信号拉高，并且把mdio_rdata的高8位数据输出给串口发送模块进行发送，将tx_data_vld拉高一个时钟周期。开始传输数据的细节如下图所示：

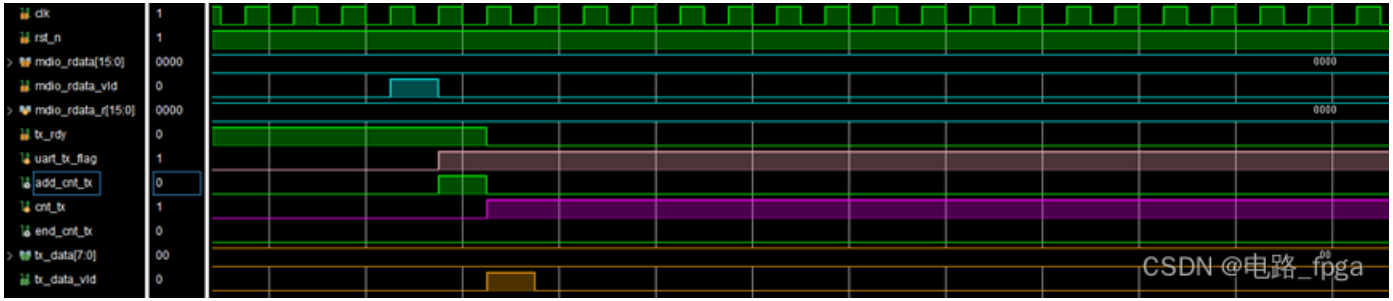

图9 开始传输高8位数据

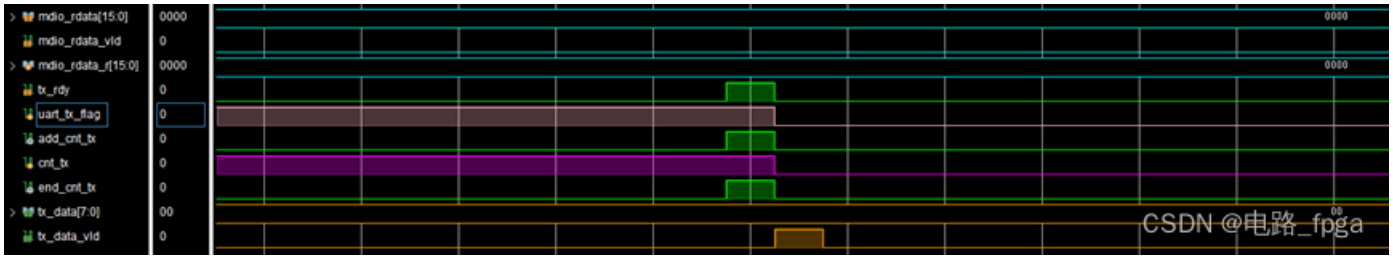

图10 发送低8位数据

由于发送的数据始终为0，所以上述仿真可能不是很直观，有兴趣的后续可以通过ILA直接抓取该模块的信号，那样更加简单，不必写MDIO的从机，或者可以对此模块单独仿真。

3、MDIO驱动模块

MDIO接口的时序在前文讲解88E1518芯片时已经讲解过了，读写时序如下图所示，所以本文就不再赘述。

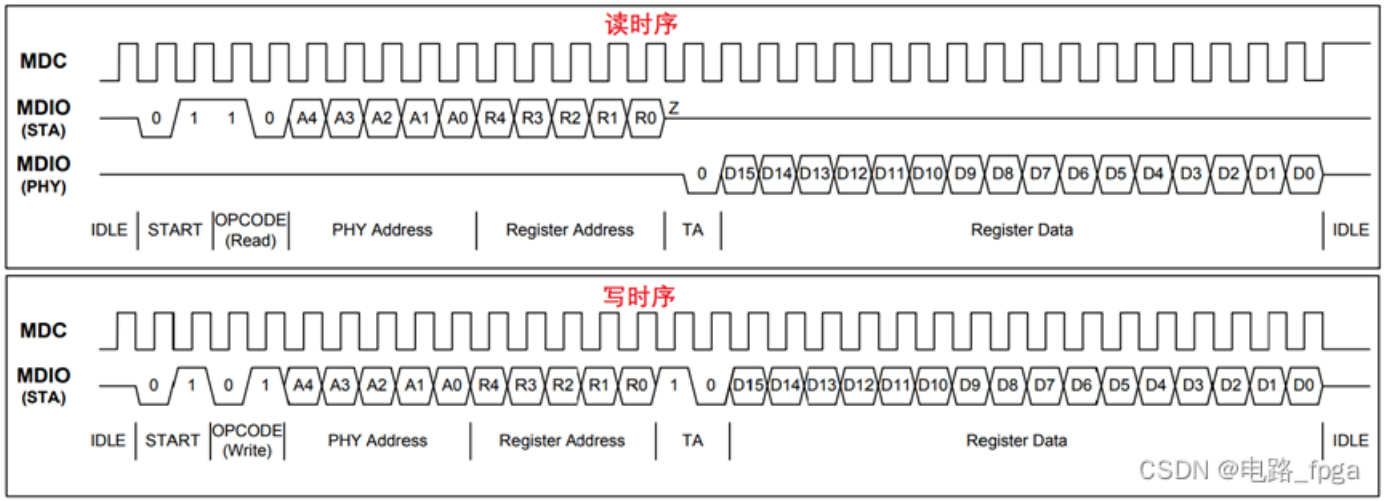

图11 MDIO读写时序

本模块的端口信号如下表3所示：

表3 MDIO驱动模块端口信号

信号名	位宽	IO	含义
clk	1	I	系统时钟，100MHz。
rst_n	1	I	系统复位，低电平有效。
start	I	1	开始读写PHY寄存器，高电平有效。
rw_en	I	1	高电平表示读取PHY内部寄存器数据，低电平表示往PHY内部寄存器写入数据。
addr	I	5	读写寄存器地址。
w data	I	16	需要写入PHY寄存器的数据。

信号名	位宽	IO	含义
rdata	O	16	从PHY内部寄存器读出的数据。
rdata_vld	O	1	读出的数据有效指示信号。
mdio_rdy	O	1	MDIO驱动模块空闲指示信号，高电平有效。
mdc	O	1	MDIO接口的时钟信号，最大不超过12MHz。
mdio	IO	1	MDIO双向数据线。

通过图5的读写时序图，使用状态机会比较简单，其实使用计数器位主架构会更加简单。本文使用状态机，总共划分为5个状态，如图12所示，空闲状态时没有读写操作，此时rdy信号为高电平。本模块需要生成MDC时钟信号，系统时钟100MHz，MDC采用10MHz，便于分频实现。

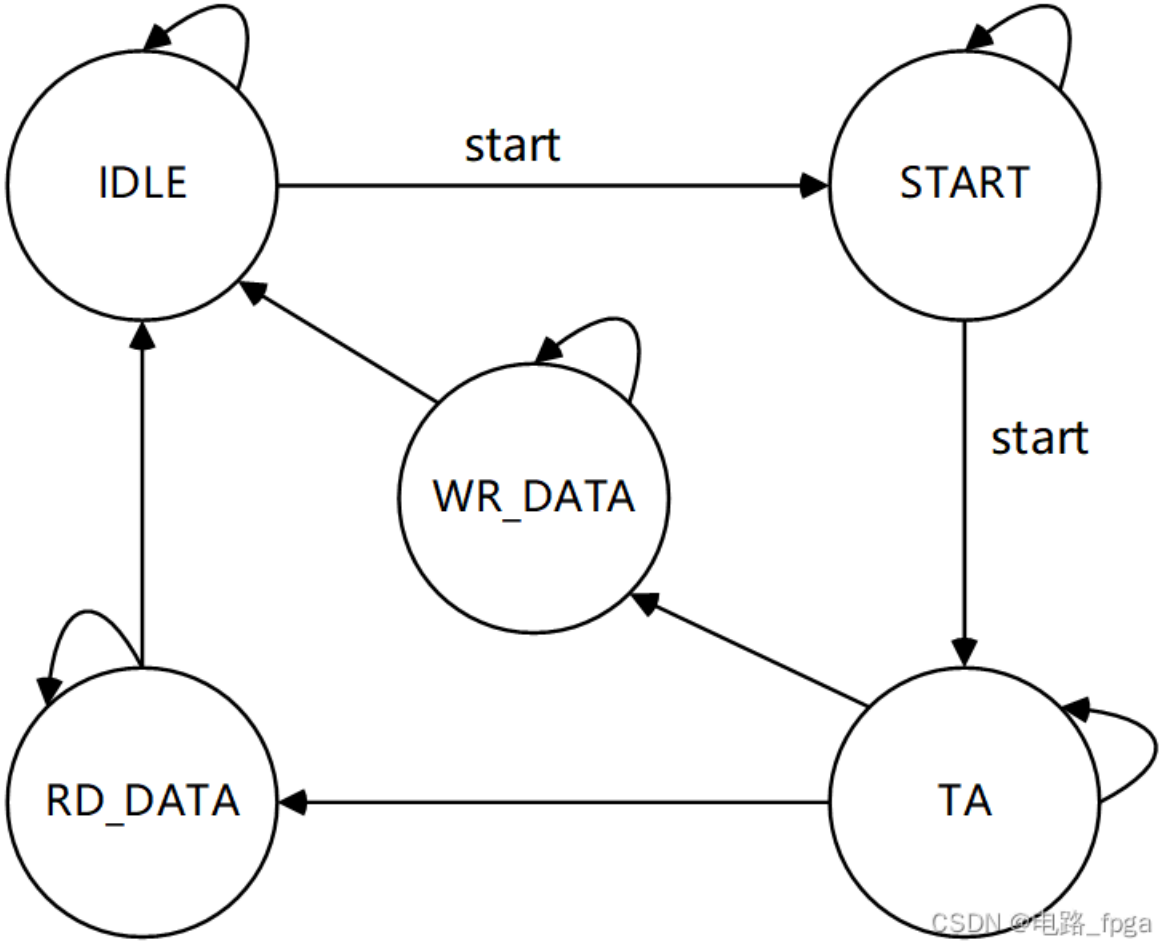

图12 状态转换图

在检测到上游模块的开始读写寄存器信号start为高电平，并且MDC下降沿到来时，状态机由空闲状态跳转到START状态，该状态会发送32位前导码，2位起始位，2位读写指示信号，5位PHY地址，5位寄存器地址，不管进行读操作还是写操作，都需要进行发送这些数据，所以将这些数据全部归为START状态，简化状态机。

发送完寄存器地址后，跳转到TA状态，如果是写操作，发送2位数据10，如果是读操作，则释放总线。之后根据读写操作，分别跳转到读数据和写数据状态。数据在MDC下降沿进行输出或读取，此处通过分频计数器的值来判断MDC的下降沿和上升沿，最好不要把MDC作为触发器的时钟信号，FPGA尽量全部使用同步时钟信号。

在读数据或者写数据状态时，代码中并不只是停留了16个时钟，而是24个时钟，原因在于图5每次进行读写数据后，都会回到空闲状态，间隔了8个时钟周期。为了保险就将这段时间合并在读写寄存器数据的状态了，只不过后面8个时钟周期直接释放总线，达到相同效果。

上述就是状态机的跳转，当然还需要一个计数器用来计数MDC的时钟个数，用作状态机跳转的判断依据，以及记录发送的数据个数，当状态机不在空闲状态且分频计数器计数结束时，该计数器就加1，状态机处于不同状态，这个计数器的最大值不一样。所以需要另一个信号来记录计数器的最大值。

注意写数据时先写高位，读数据时也是先读取高位数据，代码的总体思路就是这样了，当然还有些暂存寄存器地址，读写指示信号等，这些可以自行研究。对应的参考代码如下所示：

```
module mdio_drive #(
    parameter           DATA_W      =       16              ,//数据位宽；
    parameter           FCLK        =       100_000_000     ,//系统时钟频率，默认100MHz；
    parameter           FCLKMDC     =       10_000_000      ,//mdc时钟频率，88e1518最大不能超过12MHz；
    parameter           PHY_ADDR    =       5'b0_0000        //PHY芯片的地址，88E1518芯片高四位默认为0，最低位由config引脚状态决定。
)(
    input                                   clk             ,//系统时钟信号；
    input                                   rst_n           ,//系统复位信号，高电平有效；

    output reg                              mdc             ,//mdio的时钟信号；
    inout                                   mdio            ,//mdio双向数据信号；
    output reg                              mdio_out_en     ,//mdio三态门使能信号，高电平有效；用于仿真；

    input                                   start           ,//开始写入或者读取信号；
    input                                   rw_en           ,//读写使能，高电平表示读数据，低电平表示写数据；
    input               [4 : 0]             addr_reg        ,//读写寄存器地址；
    input               [DATA_W - 1 : 0]    wr_data         ,//需要写入的数据；
    output reg          [DATA_W - 1 : 0]    rd_data         ,//读出数据；
    output reg                              rd_data_vld     ,//读出数据有效指示信号，高电平有效；
    output reg                              rdy             ,//模块忙闲指示信号，高电平表示模块空闲，可以接收上游数据；
    output reg                              rd_ack           //读应答，高电平表示PHY芯片应答了读操作，本设计并未使用。
    );
    //处理计数器的参数；
```

```
    localparam          DIV_NUM     =       FCLK / FCLKMDC  ;//分频系数；
    localparam          DIV_NUM_W   =       clogb2(DIV_NUM-1)   ;//分频计数器位宽计算，该计数器只会计数到最大值减一；

    //状态机的状态定义；
    localparam          IDLE        =       5'b00001        ;//空闲状态；
    localparam          ADDR        =       5'b00010        ;//发送前导码，地址，读写方式的状态；
    localparam          TA          =       5'b00100        ;//根据读写转换总线状态；
    localparam          WR_DATA     =       5'b01000        ;//写数据状态；
    localparam          RD_DATA     =       5'b10000        ;//读数据状态；

    reg                                     mdio_out        ;//mdio输出数据；
    //reg                                     mdio_out_en     ;//mdio三态门使能信号，高电平有效；
    reg                                     start_r         ;//将开始信号暂时保存，与mdc信号对齐；
    reg                                     rw_en_r         ;//将读写指示信号暂存；
    reg                 [15 : 0]            wr_data_r       ;//将写数据暂存；
    reg                 [4 : 0]             state_n         ;//状态机的次态；
    reg                 [4 : 0]             state_c         ;//状态机的现态；
    reg                 [45 : 0]            start_addr      ;//将32位前导码，2位起始位，2位读写指示位，5位PHY地址，5位寄存器地址拼接；
    reg                 [5 : 0]             cnt_data_num    ;//计数器cnt_data在不同状态下需要发送或读取数据的个数；
    reg                 [DIV_NUM_W - 1 : 0] cnt_div         ;//分频计数器，用于生成mdc时钟；
    reg                 [5 : 0]             cnt_data        ;//用于计数状态机不再空闲状态下发送的数据位数；

    wire                                    add_cnt_data    ;
    wire                                    end_cnt_data    ;
    wire                                    end_cnt_div     ;
    wire                                    mdio_in         ;
    wire                                    idl2addr_start  ;
    wire                                    addr2ta_start   ;
    wire                                    ta2wr_start     ;
    wire                                    ta2rd_start     ;
    wire                                    wr2idl_start    ;
    wire                                    rd2idl_start    ;

    //mdio的三态接口；
    assign mdio = mdio_out_en ? mdio_out : 1'bz;
    assign mdio_in = mdio;

    //自动计算位宽函数；
    function integer clogb2(input integer depth);begin
        if(depth == 0)

```

```
            clogb2 = 1;
        else if(depth != 0)
            for(clogb2=0 ; depth>0 ; clogb2=clogb2+1)
                depth=depth >> 1;
        end
    endfunction

    //分频计数器，计数到分频系数减一清零；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//
            cnt_div <= 0;
        end
        else if(end_cnt_div)
            cnt_div <= 0;
        else
            cnt_div <= cnt_div + 1;
    end
    //分频计数器结束条件，计数到分频系数减一时清零；
    assign end_cnt_div = cnt_div == DIV_NUM - 1;

    //MDIO的时钟信号，当分频计数器计数结束时拉低，计数到一半时拉高；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdc <= 1'b0;
        end
        else if(cnt_div == DIV_NUM - 1)begin
            mdc <= 1'b0;
        end
        else if(cnt_div == DIV_NUM/2 - 1)begin
            mdc <= 1'b1;
        end
    end

    //把开始读写信号暂存，为了mdio的数据与mdc时钟对齐；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            start_r <= 1'b0;
        end
        else if(start)begin
            start_r <= 1'b1;

```

```
        end
        else if(end_cnt_div)begin
            start_r <= 1'b0;
        end
    end

    //将读写指示信号、写数据暂存，状态机不在空闲时，外部输入数据无效；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            rw_en_r <= 1'b0;
            wr_data_r <= 16'd0;
        end
        else if(start && state_c == IDLE)begin
            rw_en_r <= rw_en;
            wr_data_r <= wr_data;
        end
    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==1'b0)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(idl2addr_start)begin
                    state_n = ADDR;
                end
                else begin
                    state_n = state_c;
                end
            end
            ADDR:begin

```

```
147                 if(addr2ta_start)begin
148                     state_n = TA;
149                 end
150                 else begin
151                     state_n = state_c;
152                 end
153             end
154             TA:begin
155                 if(ta2wr_start)begin
156                     state_n = WR_DATA;
157                 end
158                 else if(ta2rd_start)begin
159                     state_n = RD_DATA;
160                 end
161                 else begin
162                     state_n = state_c;
163                 end
164             end
165             WR_DATA:begin
166                 if(wr2idl_start)begin
167                     state_n = IDLE;
168                 end
169                 else begin
170                     state_n = state_c;
171                 end
172             end
173             RD_DATA:begin
174                 if(rd2idl_start)begin
175                     state_n = IDLE;
176                 end
177                 else begin
178                     state_n = state_c;
179                 end
180             end
181             default:begin
182                 state_n = IDLE;
183             end
184         endcase
185     end
186
187
```

```
    // Third paragraph: Design transfer conditions;
    assign idl2addr_start = state_c==IDLE && start_r && end_cnt_div;//状态机处于空闲状态，接收到上游模块发送的开始信号且分频计数器计数结束；
    assign addr2ta_start = state_c==ADDR && end_cnt_data;//处于发送地址，前导码状态，且数据发送完成（计数器cnt_data计数结束）；
    assign ta2wr_start = state_c==TA && end_cnt_data && (~rw_en_r);//处于TA状态，且经过固定时钟周期后，如果进行写操作，则跳转到写数据阶段；
    assign ta2rd_start = state_c==TA && end_cnt_data && rw_en_r;//处于TA状态，且经过固定时钟周期后，如果进行读操作，则跳转到读数据阶段；
    assign wr2idl_start = state_c==WR_DATA && end_cnt_data;//处于写数据状态，且写完所有数据；
    assign rd2idl_start = state_c==RD_DATA && end_cnt_data;//处于读数据状态，且读完所有数据；

    //cnt_data计数器，用来计数状态机不处于空闲状态时，在各个状态下发送的数据个数；
    //初始值为0，当状态机不处于空闲状态且分频计数器计数结束时加1。
    //当计数器计数到cnt_data_num-1时表示该状态的数据已经写入或读取完成，此时计数器清零；
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//
            cnt_data <= 0;
        end
        else if(add_cnt_data)begin
            if(end_cnt_data)
                cnt_data <= 0;
            else
                cnt_data <= cnt_data + 1;
        end
    end

    assign add_cnt_data = ((state_c != IDLE) && end_cnt_div);
    assign end_cnt_data = add_cnt_data && cnt_data == cnt_data_num - 1;

    //状态机各个状态需要发送数据或者读取数据的个数；
    always@(posedge clk)begin
        if(state_c == TA)begin//此处最多需要发送2位数据；
            cnt_data_num <= 6'd2;
        end
        else if(state_c == WR_DATA || state_c == RD_DATA)begin//读取或者写入的数据都是16位,但是手册里让每次数据发送完成和接收完成后，需要释放总线一段时间z
            cnt_data_num <= 6'd25;
        end
        else begin//在ADDR状态下，需要发送32位前导码，2位起始位，2位读写指示位，5位PHY地址，5位寄存器地址；
            cnt_data_num <= 32+2+2+5+5;
        end
    end

    //当状态机处于空闲状态且开始信号有效时，将状态机需要在ADDR状态发送的数据拼接；
```

```
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值全为高电平；
            start_addr <= 46'h3fff_ffff_ffff;
        end
        else if(start && state_c == IDLE)begin//状态机在空闲状态下，检测到开始发送信号时，将前导码，起始位，读写状态，PHY地址，寄存器地址拼接。
            start_addr <= {32'hffff_ffff,2'b01,{rw_en,~rw_en},PHY_ADDR,addr_reg};
        end
    end

    //输出mdio的数据。
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            mdio_out <= 1'b0;
        end
        else if(state_c == ADDR)begin//输出前导码，地址等数据；
            mdio_out <= start_addr[45 - cnt_data];
        end
        else if(state_c == TA)begin//此过程输出2'b10，读的时候将三态使能信号拉低即可释放总线，与数据线状态无关，所以不会影响。
            if(cnt_data == 0)
                mdio_out <= 1'b1;
            else
                mdio_out <= 1'b0;
        end
        else if(state_c == WR_DATA && (cnt_data < 16))begin//写状态时，将16位数据输出，先输出高位数据；
            mdio_out <= wr_data_r[15 - cnt_data];
        end
    end

    //三态门使能信号；
    always@(posedge clk)begin//当状态机处于ADDR 或者 写数据 或者 TA状态且写有效时将三态门使能；
        if(state_c == ADDR || (state_c == WR_DATA && (cnt_data < 16)) || (state_c == TA && ~rw_en_r))begin
            mdio_out_en <= 1'b1;
        end
        else begin//其余时间三态门使能关闭，释放总线；
            mdio_out_en <= 1'b0;
        end
    end

    //读应答，只有在TA阶段，MDC下降沿当数据线被PHY芯片拉低时，表示PHY芯片应答了，此时将应答标志拉高，其余时间均为低电平；
    always@(posedge clk)begin
```

```
        if(state_c == TA && rw_en_r && (cnt_div == DIV_NUM - 1))begin
            rd_ack <= ~mdio_in;
        end
        else begin
            rd_ack <= 1'b0;
        end
    end

    //读取采集到的数据;
    always@(posedge clk)begin
        if(rst_n==1'b0)begin//初始值为0;
            rd_data <= 16'd0;
        end//状态机处于读取状态时，在分频时钟上升沿沿读取数据，先读取高位数据;
        else if(state_n == RD_DATA && (cnt_div == DIV_NUM/2 - 1) && (cnt_data < 16))begin
            rd_data[15 - cnt_data] <= mdio_in;
        end
    end

    //生成读取数据有效指示信号;
    always@(posedge clk)begin//当读取完所有数据时，输出有效指示信号拉高，其余时间拉低;
        rd_data_vld <= ((state_c == RD_DATA) && (cnt_data == 15) && (cnt_div == DIV_NUM/2 - 1));
    end

    //忙闲指示信号,
    always@(*)begin
        if(start || start_r || (state_c != IDLE))
            rdy = 1'd0;
        else
            rdy = 1'b1;
    end

endmodule
```

该模块的仿真读寄存器的整体时序如下图所示：

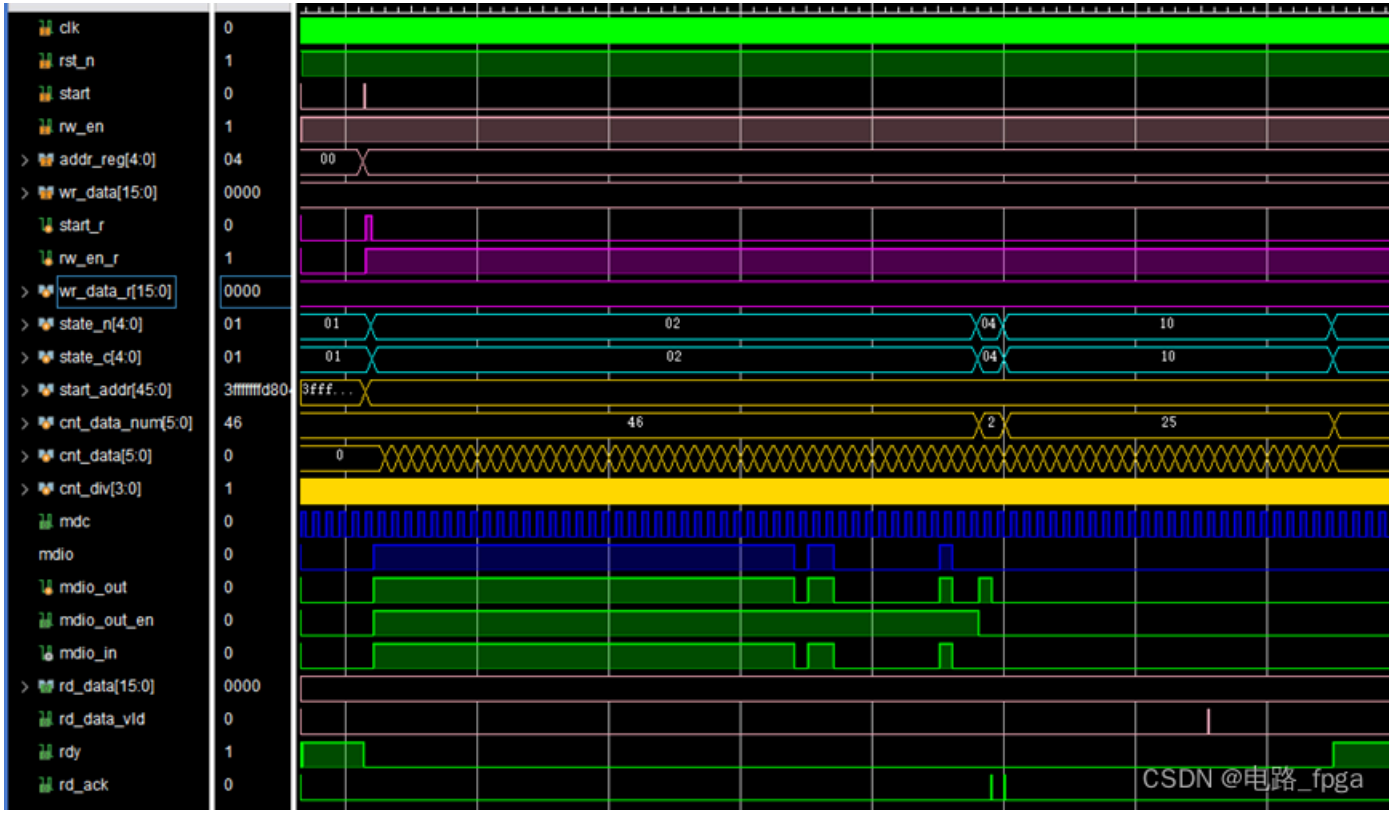

图13 读寄存器的整体时序

放大后如图14所示，首先前导码输出32个高电平。

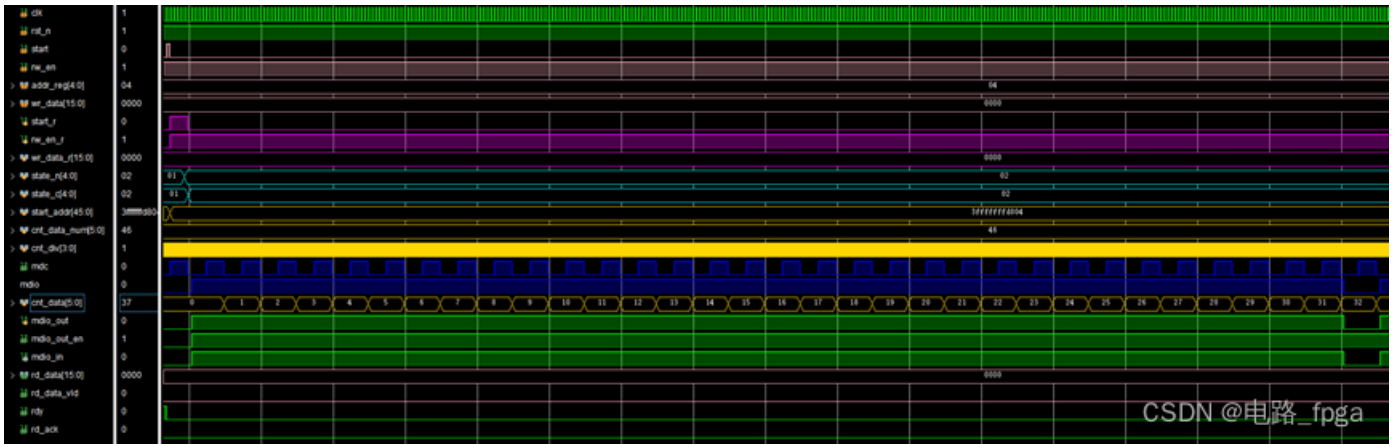

图14 发送前导码

然后依次发送2为起始位，2为读指示信号，5位PHY地址（本次使用88E1518的PHY地址设置为5’d0和5’d1，由硬件电路决定），5位寄存器地址，然后释放总线，mdio_out_en是三态门的使能信号，低电平表示释放总线。之后经过在16个MDC时钟的下降沿读取mdio数据线上的信号，读取完毕后将rdata_vld拉高，表示读取数据有效。

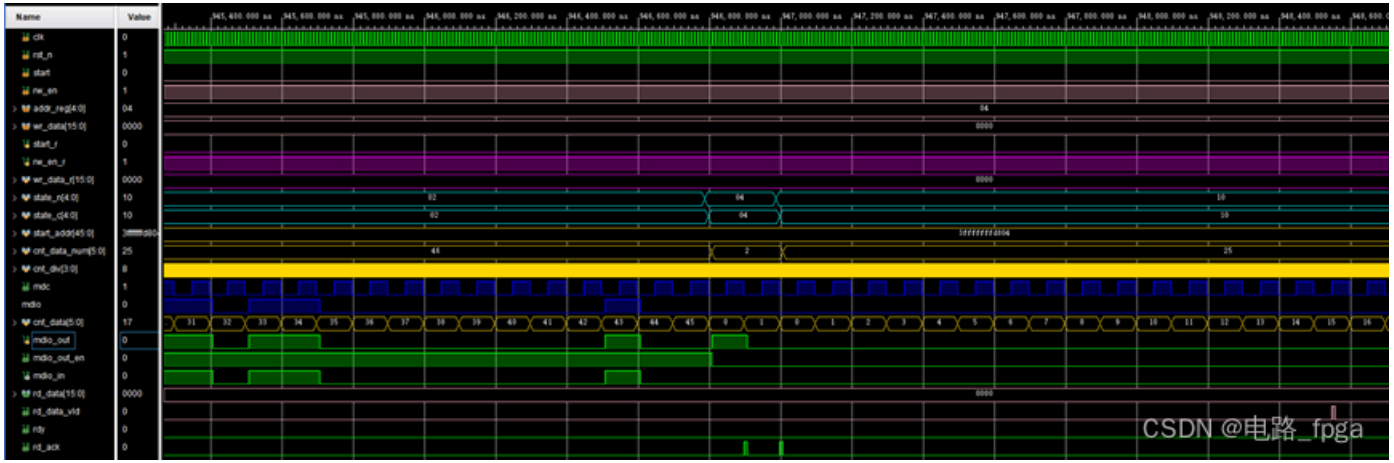

图15 其余位的时序

写寄存器的总体仿真时序如下图所示：

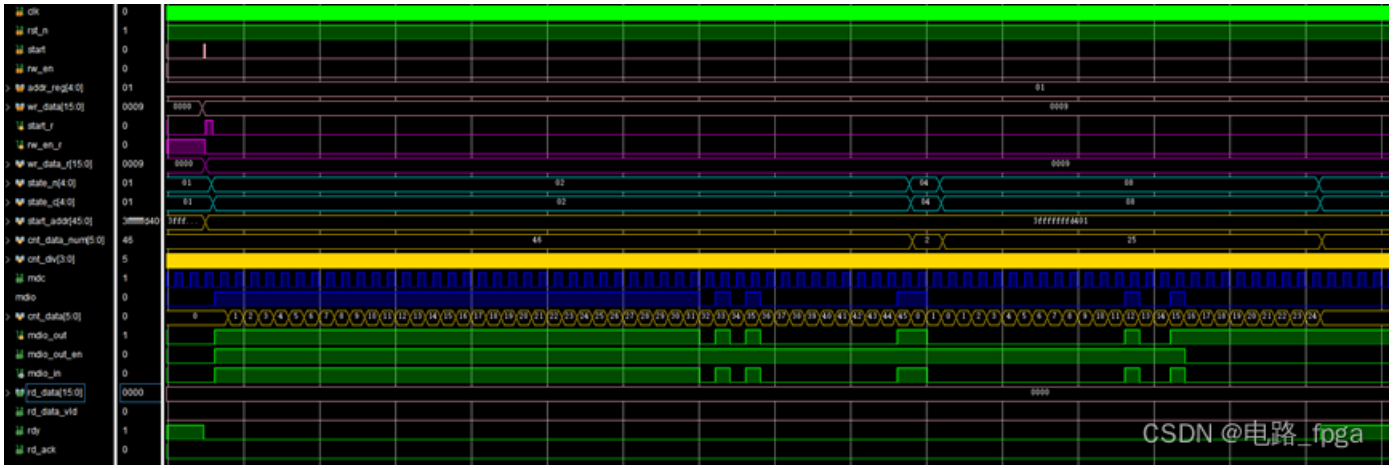

图16 写寄存器整体仿真时序

起始时序如下图所示：

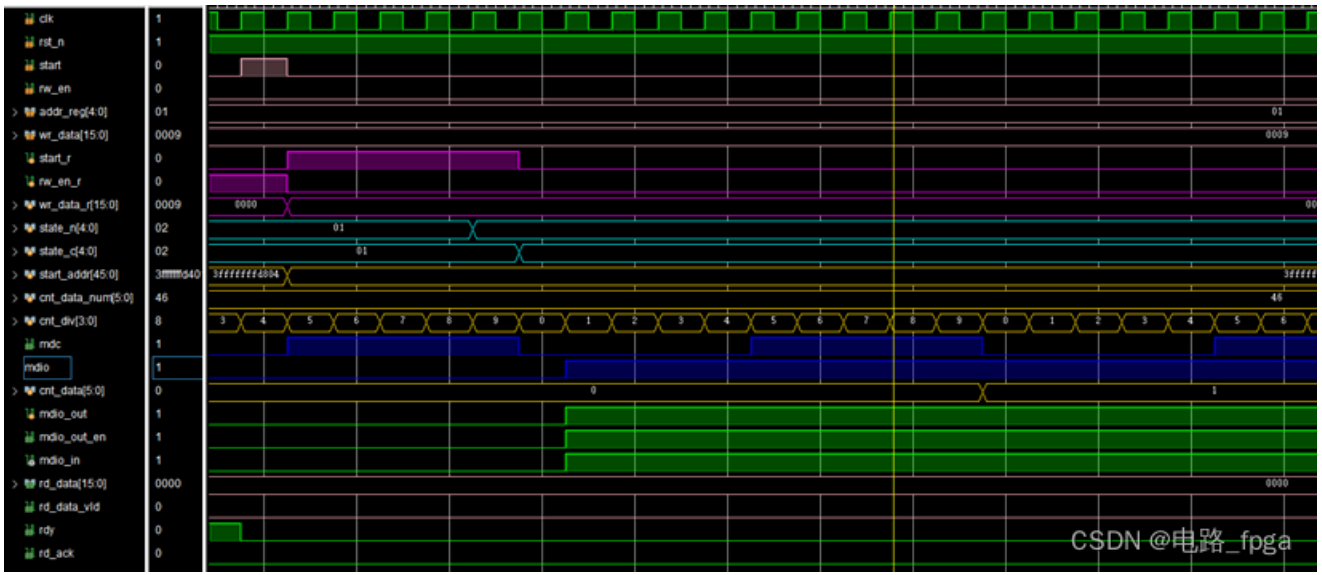

图17 写寄存器起始时序

前导码发送之后的时序如下图所示，当数据发送完毕后把总线释放。

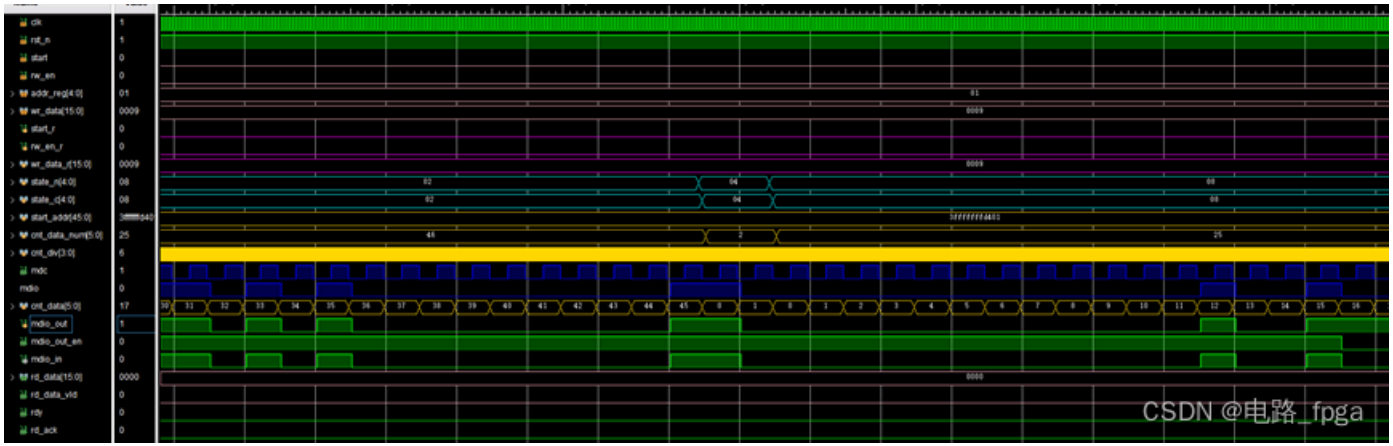

图18 前导码之后的时序

4、上板测试

为了查看MDIO接口时序，所以在顶层文件中加入了一个ILA模块，用来抓取需要查看的一些信号，便于调试。然后开发板插上千兆网线，串口数据线，下载器，最后打开电源下载程序，如下所示。

串口助手读写PHY寄存器

图19 板卡接线图

程序下载完成后，查看电脑上此时网口的传输速率，在搜索栏中搜索“查看网络连接”，如下图所示。

图20 查看网络连接

如下图所示，如果将电脑通过网线与开发板的网口连接，则会出现 以太网　字样，选中后鼠标右键，点击状态。

图21 查看以太网通信速率

如果电脑网卡速率大于等于1Gbps，那么此时会如下图显示一致，使用1Gbps进行传输，因为此时PC和FPGA的PHY芯片的通信速率是通过自动协商完成的，所以会选择都支持的最高通信速率。

图22 以太网通信速率

PC端的最大通信速率可以手动修改，如图所示，鼠标右击以太网，选中属性。

图23 修改以太网速率（1）

然后点击配置，如下图所示：

图24 修改以太网速率（2）

如下图所示，选择高级，然后下拉点击连接速度和双工模式，之后可以发现默认是自动侦测，此时我们手动修改位100Mbps 全双工。

图25 修改以太网速率（3）

之后在查看以太网的通信速率，结果如图所示，此时就是100Mbps 全双工通信模式了。

图26 修改以太网速率（4）

这是PC端进行修改，本文需要实现FPGA通过修改内部寄存器实现以太通信速率的修改，所以先将PC端修改回自动侦测，然后我们通过配置PHY芯片寄存器达到修改通信速率的效果。

首先打开串口调试助手，将波特率设置为115200（代码中默认设置的115200，使用其他波特率需要修改顶层文件的波特率数值），将数据位设置为8位，起始位1位，无校验位，1位停止位，接收和发送的数据均以16进制数据进行显示，设置如下图所示：

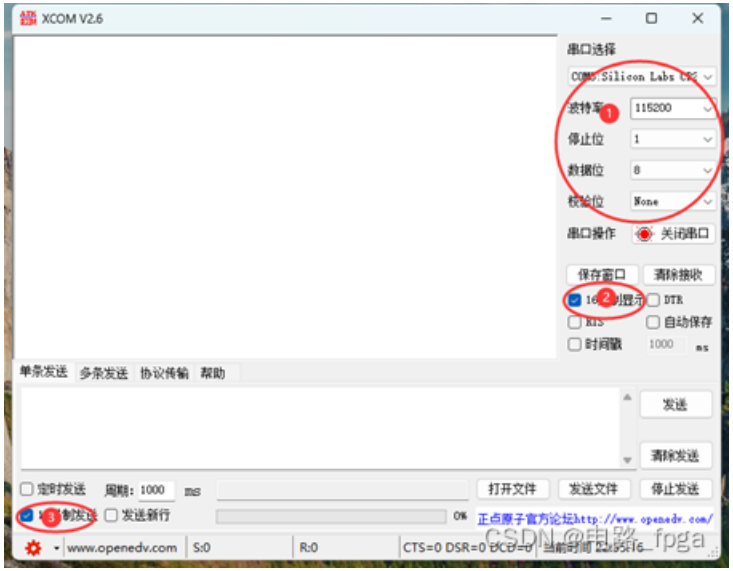

图27 串口调试助手设置

首先将ILA的start位高电平作为触发条件，然后通过串口调试助手发送读取指令，首先读取17_0号寄存器的数据，通过bit15和bit14判断当前PHY工作速率，如下图所示，发送帧头5A，然后读指示数据01，然后跟读取寄存器地址，17的16进制为11。发送指令后，上面会返回读取到该寄存器的数据为16'hAC48，bit15:14为2'b10，则表示此时的通信速率为1000Mbps，bit13为高电平表示全双工模式。

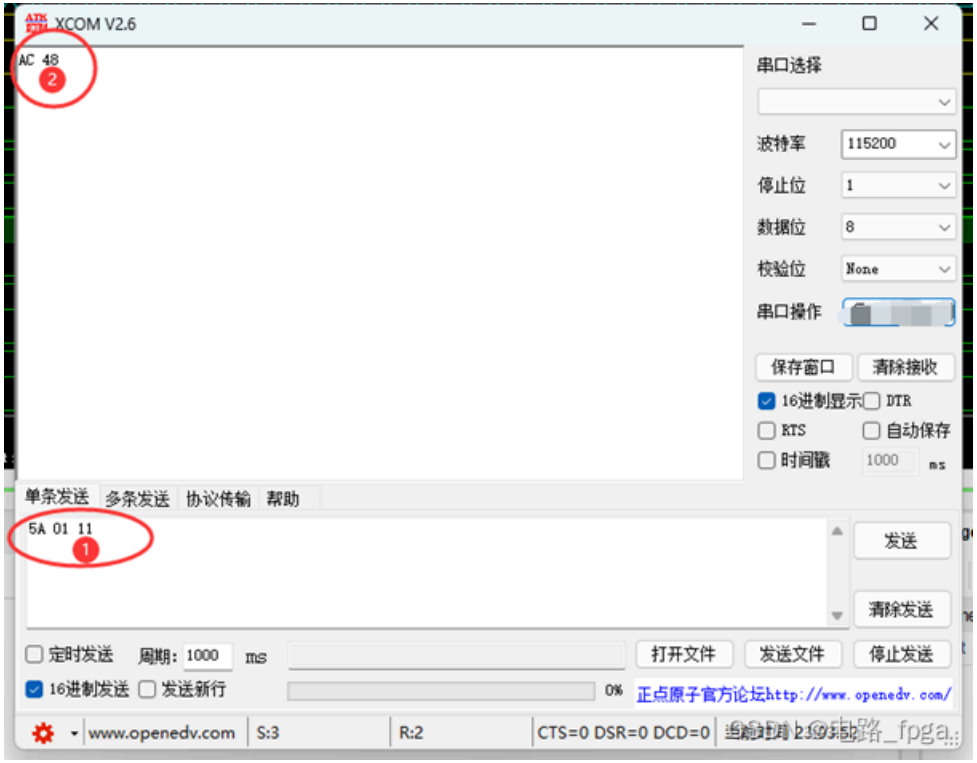

图28 串口助手发送读寄存器指令

此时可以通过PC端查看，也为1Gbps传输速率，然后查看ILA触发的波形时序，如下图所示，FPGA释放MDIO总线后，MDIO总线会被上拉电阻拉高，然后下个时钟会被PHY芯片拉低，然后就在MDC时钟上升沿输出对应寄存器的数据，FPGA接收到的数据也是16'hAC48，与串口助手返回的数据一致。

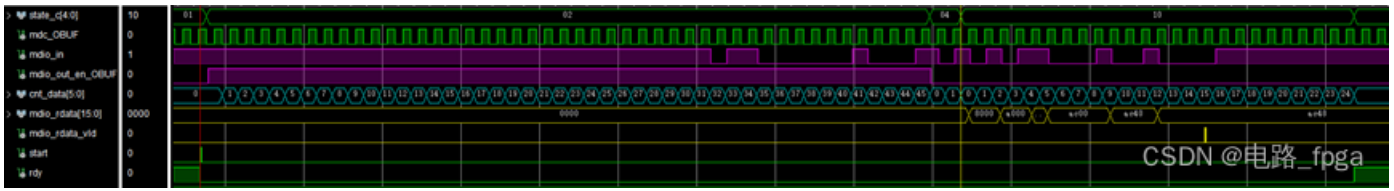

图29 ILA抓取MDIO读时序

所以PHY芯片默认会使用1000Mbps全双工模式进行通信，然后我们要修改PHY芯片通信模式，则需要将PHY芯片的自动协商关闭(0_0.12写入0)，此时将通信速率更改为10Mbps，则0_0.13和0_0.6均写入0。并且使用半双工模式，那么0_0.8写入0，想要0_0.13和0_0.6，0_0.8写入生效，就必须在0_0.15或者0_0.9写入1，此处在0_0.9写入1重启自动协商，所以需要写入的数据是16'h0200，串口助手发送写指令的格式如下所示。

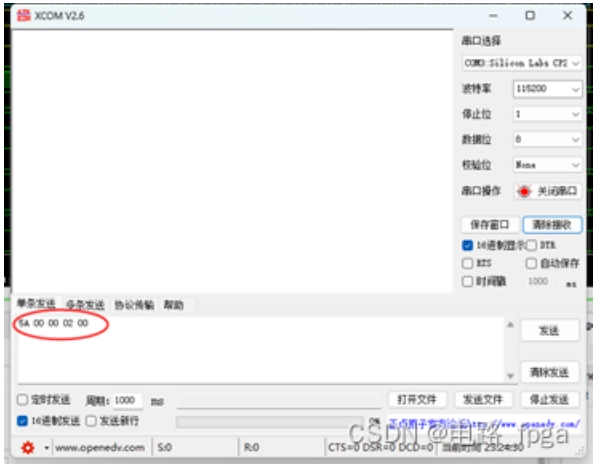

图30 串口助手发送写寄存器指令

向0号寄存器吸入16'h0200，ILA抓取的时序如下所示：

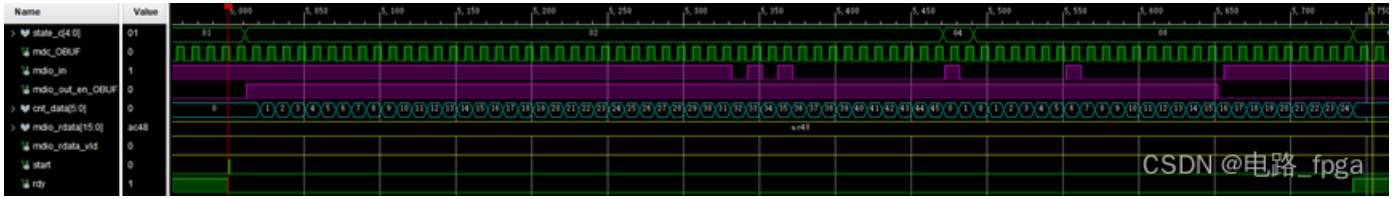

图31 ILA抓取的写寄存器时序

上述时序图可能太小，将图放大，如下面两图所示：

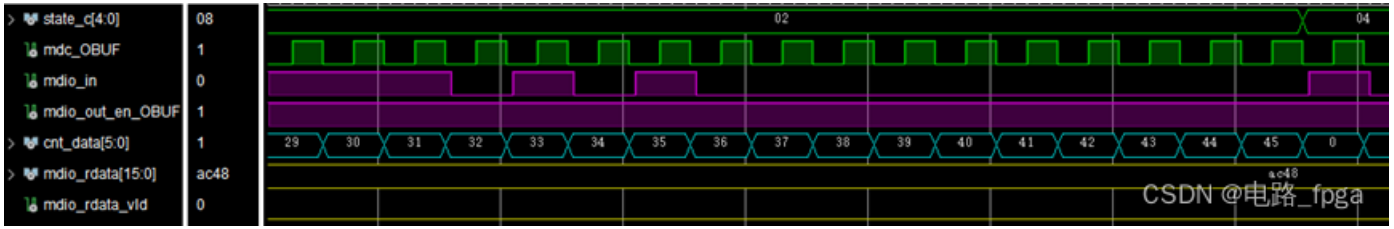

图32 ILA抓取的写寄存器时序（放大部分1）

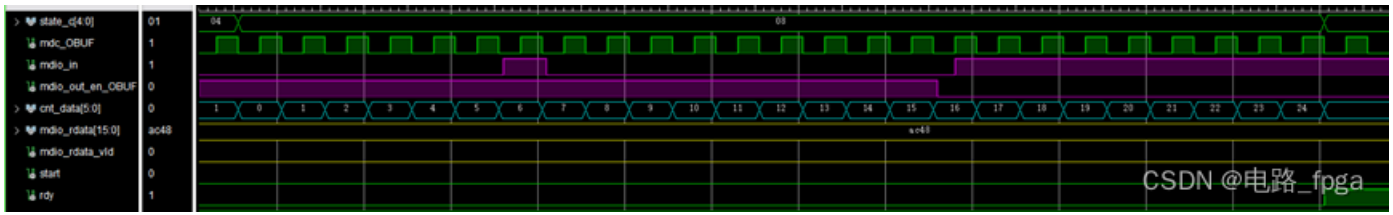

图33 ILA抓取的写寄存器时序（放大部分2）

然后继续读取17_0.15:13寄存器，查看PHY芯片此时的通信速率，以及工作模式，读取的数据如下图所示，返回数据为16'h0C08，表明此时PHY芯片以10Mbps半双工模式进行通信，表示写入0_0寄存器的数据有效。

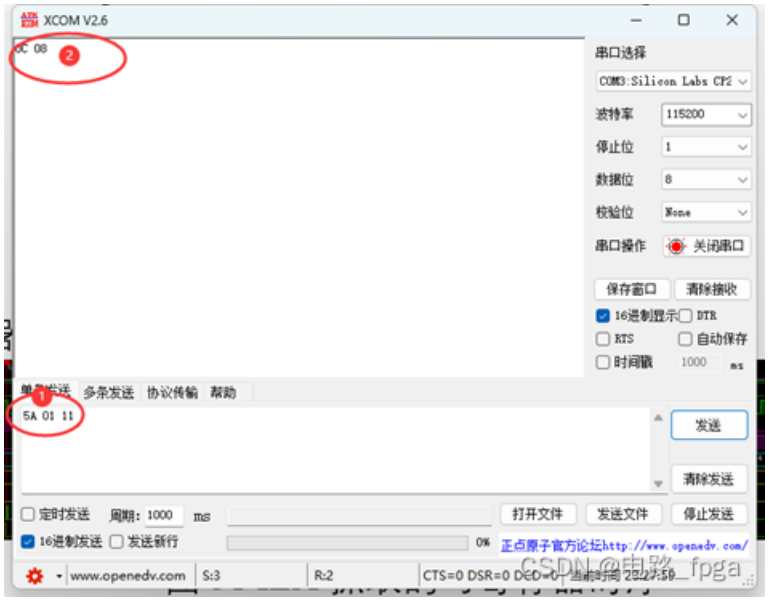

图34 读17_0寄存器

此时查看PC端以太网的通信速率如下图所示，也为10Mbps的通信速率，因此也可以判断数据的正确性。

图35 PC端此时通信速率

注意在设置0_0寄存器时，需要重点关注自动协商是否被关闭，如果没有关闭自动协商，对PHY通信速率设置是不会起作用的。

综上所述，本文通过串口调试助手间接读写PHY芯片内部的寄存器，这种方式对于开始了解一个接口时非常有效，可以对你的猜想快速进行验证，可以对内部任何寄存器进行读写操作，只需要通过串口助手修改发送的数据和地址即可，不需要对代码做出任何调整。

本文也是对MDIO的时序以及实现做了充分验证，由兴趣的可以读取其他页寄存器数据验证换页的操作，只需要提前向22号寄存器写入对应页即可跳转到指定页。工程文件可以在公众号后台回复“**FPGA实现MDIO控制器**”（不含引号）即可。

对了，放一个我刚开始读写这些寄存器的视频，由于开始没注意自动协商，怎么改寄存器都达不到效果，浪费了一些时间。

串口助手调试PHY芯片

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⭐！