将Xilinx DDR3 MIG IP核的AXI_FULL接口封装成FIFO接口（含源码）

基于FPGA的DDR相关知识导航界面，点击查看。

基于FPGA的以太网相关文章导航，点击查看。

MIG IP除了支持前文讲解的APP接口，还支持axi_full接口，因此本文使用MIG IP的axi_full接口封装为FIFO接口，取代 以太网 传输图片工程中的DDR3读写控制模块。

回顾前文APP接口的 DDR3 读写控制模块，框图如下所示，本文需要将MIG IP的APP接口更换为axi_full接口时序，从而ddr3_rw模块需要重新设计。

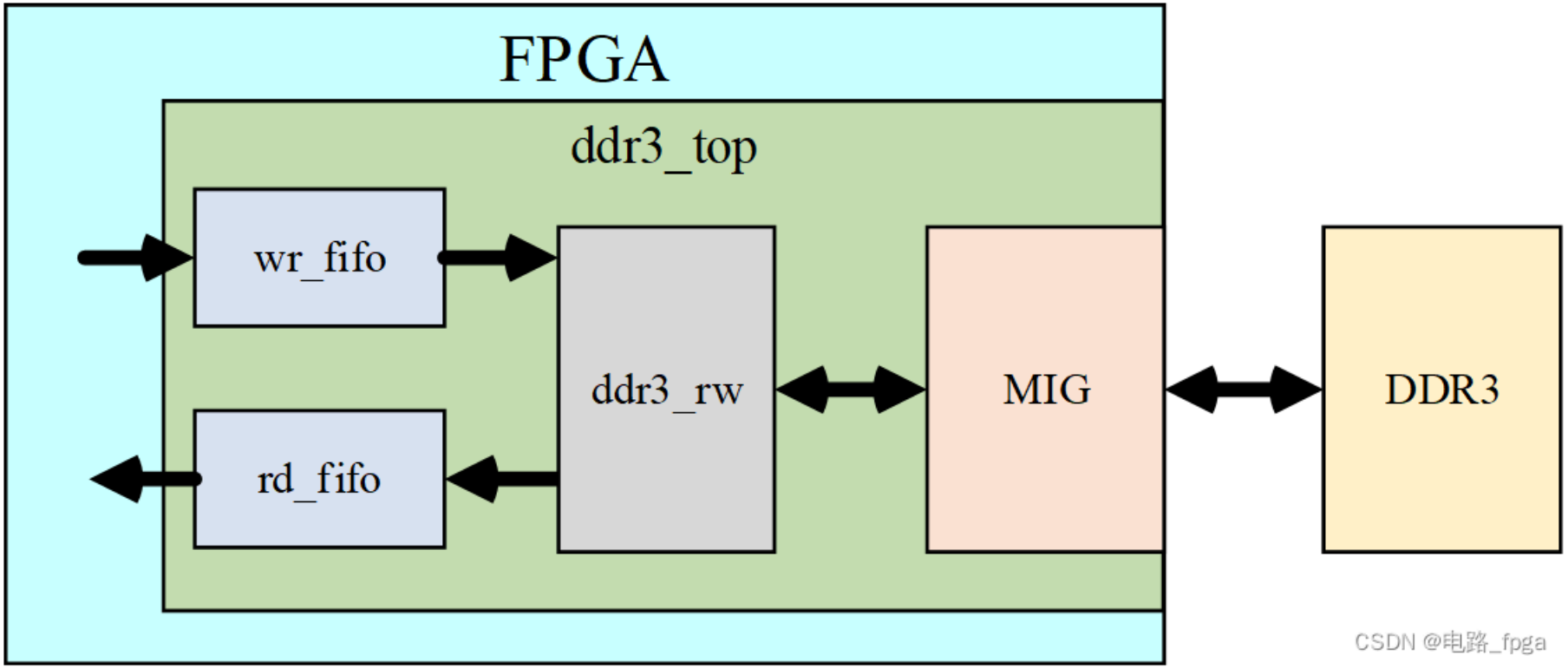

图1 DDR3读写控制模块

1、生成axi_full接口的MIG IP

关于MIG IP的详细设置可以参考前文，本文只对AXI接口的生成以及参数设置进行讲解，其余内容不再赘述。

首先在生成MIG IP的下面界面中勾选使用AXI4接口，然后点击下一步，对AXI接口的一些参数进行设置。

图2 使用AXI4接口

启用AXI4接口之后，在对DDR3时钟、芯片型号、数据位宽配置完成之后，需要对AXI4的一些接口参数进行配置，如下图所示。如果不使用AXI4接口，将没有这个页面的参数配置。

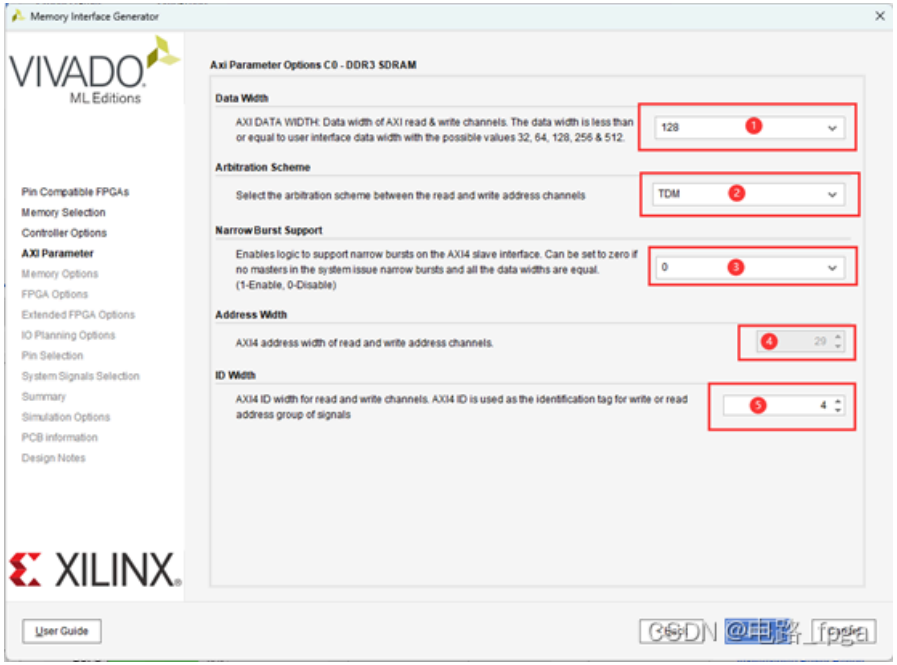

图3 AXI4参数设置

1、Data Width：表示axi_full接口读写数据的位宽，如果前面将DDR3芯片时钟设置为用户时钟频率的4倍，那么此处将读写数据位宽设置为DDR3数据位宽的8倍时，读写速率达到最大。

原因在于DDR3芯片的突发一般是8，如果DDR3一次传输16位，8次传输128位数据，且DDR3的时钟是用户时钟的4倍，双沿传输数据。那么DDR3芯片突发一次，用户也只能传输一次数据，当用户一次传输数据位宽等于DDR3一次突发传输数据时，效率达到最大。

2、仲裁机制，由于DDR3芯片只有一组数据，而axi_full的读、写接口支持同时传输数据。当选用“TDM”时，读、写操作同时到达时，将交替进行。由于MIG IP自带仲裁机制，用户侧就不需要在考虑读写仲裁问题了。

3、Narrow Burst设置位0即可。

4、地址位宽会根据DDR3芯片型号自动得到，不能设置。

5、ID位宽，axi_full协议每个通道都有ID信号，这里需要设置该信号位宽，默认即可。

关于AXI接口的设置就完成了，其余设置于前文的MIG IP设置保持一致即可，之后生成IP，得到其对应的信号列表如下所示。

这些信号要么在前面讲解MIG IP仿真时讲过，要么就是axi_full的信号，因此本文不再赘述其含义。

2、设计MIG读写控制模块

2.1、设计思路

本文的主要目的在于掌握axi_full接口的设计，axi_full的读、写操作可以同时进行，再加上MIG IP能够自己仲裁，用户就不用去考虑地址冲突了。

因此读写操作就可以不使用状态机，直接使用一个标志信号进行控制即可。

写操作设计思路：当写FIFO中的数据大于一次突发写传输的数据，且写FIFO不处于复位状态，且不处于写状态时，将写标志信号拉高。然后开始生成写地址通道信号，从机应答写地址通道后，开始写数据，对写入数据个数进行计数，直到写入突发长度数据时结束，之后写应答通道握手成功后，写标志信号拉低，结束一次写操作。

在此过程中，写地址会递增，作为下次写突发的首地址。

读操作思路：当读FIFO中的数据小于一次突发传输的长度时，且读FIFO不处于复位状态，写完一次所有地址，且不处于读状态时，将读标志信号拉高。读地址通道信号握手成功后，开始接收从机的数据写入到读FIFO中，接收完数据后读标志信号拉低。

此模块还包括生成对读、写FIFO的复位信号，xilinx需要把FIFO复位信号拉高多个时钟才能有效复位，因此会使用计数器对FIFO进行复位，复位FIFO的同时会将axi_full接口的读、写地址清零，从头开始存取数据，保证数据正确性。

下文结合代码分析实际的设计细节。

2.2、axi_full写操作

首先是该模块的接口信号，由于外部需要连接两个FIFO和axi_full接口的MIG IP，因此这些信号分为这三种写FIFO的读端口信号、读FIFO的写端口信号、axi_full相关接口。

```
module ddr3_rw #(
    parameter   PINGPANG_EN         =       1'b0                ,//乒乓操作是否使能;
    parameter   USE_ADDR_W          =       29                  ,//用户需要写入地址的位宽;
    parameter   USE_BUST_LEN_W      =       8                   ,//用户侧读写数据突发长度的位宽;
    parameter   USE_DATA_W          =       16                  ,//用户侧读写数据的位宽;
    parameter   DDR_ADDR_W          =       29                  ,//MIG IP读写数据地址位宽;
    parameter   DDR_DATA_W          =       128                  //MIG IP读写数据的位宽;
)(
    //MIG IP用户侧接口信号
    input                                   ui_clk              ,//mig ip 用户时钟;
    input                                   ui_clk_sync_rst     ,//复位,高有效;
    input                                   ddr3_init_done      ,//DDR3初始化完成;
    //AXI写地址通道信号
    output      [3 : 0]                     s_axi_awid          ,//写地址ID;
    output  reg [DDR_ADDR_W - 1 : 0]        s_axi_awaddr        ,//写突发传输的首地址;
```

```
    output      [7 : 0]                     s_axi_awlen         ,//突发写的长度-1；
    output      [2 : 0]                     s_axi_awsize        ,//突发写的大小，每次写数据的字节数；
    output      [1 : 0]                     s_axi_awburst       ,//突发写的类型；
    output      [0 : 0]                     s_axi_awlock        ,//锁类型；
    output      [3 : 0]                     s_axi_awcache       ,//存储类型；
    output      [2 : 0]                     s_axi_awprot        ,//保护类型，表示传输的特权级及安全等级；
    output      [3 : 0]                     s_axi_awqos         ,//质量服务；
    output  reg                             s_axi_awvalid       ,//写地址有效指示信号，高电平有效；
    input                                   s_axi_awready       ,//写地址应答应答信号，高电平有效；
    //AXI写数据通道信号
    output      [DDR_DATA_W - 1 : 0]        s_axi_wdata         ,//写数据。
    output      [DDR_DATA_W / 8 - 1 : 0]    s_axi_wstrb         ,//写数据掩码信号，低电平有效；
    output  reg                             s_axi_wlast         ,//突发写的最后一个数据，高电平有效；
    output  reg                             s_axi_wvalid        ,//写数据有效指示信号，高电平有效；
    input                                   s_axi_wready        ,//写数据应答信号，高电平有效；
    //AXI写应答通道信号
    input        [3 : 0]                    s_axi_bid           ,//写响应ID，必须与AWID的数值匹配；
    input        [1 : 0]                    s_axi_bresp         ,//表明写事务的状态，为0时表示写入正确；
    input                                   s_axi_bvalid        ,//写响应有效指示信号，高电平有效；
    output  reg                             s_axi_bready        ,//接收到从机的响应信号，高电平有效；
    //AXI读地址通道信号
    output      [3 : 0]                     s_axi_arid          ,//读地址ID；
    output  reg [DDR_ADDR_W - 1 :0]         s_axi_araddr        ,//读突发传输的首地址；
    output      [7 : 0]                     s_axi_arlen         ,//突发读的长度-1；
    output      [2 : 0]                     s_axi_arsize        ,//突发读的大小，每次读数据的字节数；
    output      [1 : 0]                     s_axi_arburst       ,//突发读的类型；
    output      [0 : 0]                     s_axi_arlock        ,//锁类型；
    output      [3 : 0]                     s_axi_arcache       ,//存储类型；
    output      [2 : 0]                     s_axi_arprot        ,//保护类型，表示传输的特权级及安全等级；
    output      [3 : 0]                     s_axi_arqos         ,//质量服务；
    output  reg                             s_axi_arvalid       ,//读地址有效指示信号，高电平有效；
    input                                   s_axi_arready       ,//读地址应答应答信号，高电平有效；
    //AXI读数据通道信号
    input     [3 : 0]                       s_axi_rid           ,//读ID，必须与ARID的数值匹配；
    input     [DDR_DATA_W - 1 : 0]          s_axi_rdata         ,//读数据；
    input     [1 : 0]                       s_axi_rresp         ,//表明读事务的状态，为0时表示写入正确；
    input                                   s_axi_rlast         ,//突发读的最后一个数据，高电平有效；
    input                                   s_axi_rvalid        ,//读数据有效指示信号，高电平有效；
    output reg                              s_axi_rready        ,//读数据应答信号，高电平有效；
    //写FIFO相关信号
```

```
    input                                   wfifo_empty         ,//写FIFO空指示信号；
    input                                   wfifo_rd_rst_busy   ,//写FIFO的复位忙指示信号，高电平表示处于复位状态；
    input       [DDR_DATA_W - 1 : 0]        wfifo_rdata         ,//写FIFO读数据；
    input       [USE_BUST_LEN_W - 1 : 0]    wfifo_rdata_count   ,//写FIFO读侧的数据个数；
    output                                  wfifo_rd_en         ,//写FIFO读使能；
    output reg                              wfifo_wr_rst        ,//写FIFO复位信号；
    //读FIFO相关信号
    input                                   rfifo_full          ,//读FIFO满指示信号；
    input                                   rfifo_wr_rst_busy   ,//读FIFO的复位忙指示信号，高电平表示处于复位状态；
    input       [USE_BUST_LEN_W - 1 : 0]    rfifo_wdata_count   ,//读FIFO写侧的数据个数；
    output reg                              rfifo_wr_en         ,//读FIFO写使能；
    output reg                              rfifo_rd_rst        ,//读FIFO复位信号；
    output reg  [DDR_DATA_W - 1 : 0]        rfifo_wdata         ,//读FIFO写数据；
    //MIG IP读写数据突发地址限制
    input       [DDR_ADDR_W - 1 : 0]        app_addr_wr_min     ,//写DDR3的起始地址；
    input       [DDR_ADDR_W - 1 : 0]        app_addr_wr_max     ,//写DDR3的结束地址；
    input       [USE_BUST_LEN_W - 1 : 0]    app_wr_bust_len     ,//向DDR3中写数据时的突发长度减1.
    input       [DDR_ADDR_W - 1 : 0]        app_addr_rd_min     ,//读DDR3的起始地址；
    input       [DDR_ADDR_W - 1 : 0]        app_addr_rd_max     ,//读DDR3的结束地址；
    input       [USE_BUST_LEN_W - 1 : 0]    app_rd_bust_len     ,//从DDR3中读数据时的突发长度；
    input                                   wr_rst              ,//写复位信号，上升沿有效，持续时间必须大于ui_clk的周期；
    input                                   rd_rst               //读复位信号，下降沿沿有效，持续时间必须大于ui_clk周期；
);
```

之后是写开始信号和写标志信号，拉高条件一致，由于FIFO采用的超前模式，所以FIFO中实际数据个数等于wfifo_data_count+2。当写应答通道握手时把写标志信号拉低，其余时间保持不变。

```
    /*****************生成突发写的开始信号和标志信号*****************/
    //写开始信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            wr_start <= 1'b0;
        end
        else begin//当DDR3初始化完成 且 写FIFO不处于复位状态 且 写FIFO中的数据大于等于一次写突发传输的数据 且 不处于写数据状态时拉高。
            wr_start <= ddr3_init_done && (~wfifo_rd_rst_busy) && (wfifo_rdata_count >= app_wr_bust_len - 2) && (~wr_flag);
        end
```

```
    end

    //写数据标志信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            wr_flag <= 1'b0;
        end
        else if(s_axi_bvalid && s_axi_bready)begin//当写应答通道应答时拉低。
            wr_flag <= 1'b0;
        end//当开始写入数据时拉高；
        else if(ddr3_init_done && (~wfifo_rd_rst_busy) && (wfifo_rdata_count >= app_wr_bust_len - 2) && (~wr_flag))begin
            wr_flag <= 1'b1;
        end
    end
```

然后写地址相关信号如下所示，需要注意突发长度的数值等于用户设置数值减1，因为从0开始计数的。突发类型设置为递增类型，突发写的大小是使用函数计算的常数。

```
    /*****************生成写地址通道的信号*****************/
    assign s_axi_awid    = 4'd0;//写地址ID；
    assign s_axi_awlen   = app_wr_bust_len - 1;//突发写的长度-1；
    assign s_axi_awsize  = DDR_DATA_BYTE_W;//突发写的大小，每次写数据的字节数；
    assign s_axi_awburst = 2'b01;//突发写的类型；
    assign s_axi_awlock  = 1'b0;//锁类型；
    assign s_axi_awcache = 4'b0000;//存储类型；
    assign s_axi_awprot  = 3'b000;//保护类型，表示传输的特权级及安全等级；
    assign s_axi_awqos   = 4'b0000;//质量服务；
```

写地址计数器用于记录写突发的首地址，初始值为用户设置的起始地址，由于axi_full的地址是以字节为单位，因此每次地址握手后，需要增加突发长度乘以写数据位宽除8，作为下次写突发传输的首地址。如果达到用户设置最大地址减去一次突发长度对应的地址，则回到首地址，写FIFO复位时回到基地址。

```
    //计算写突发传输最大地址，等于用户设置最大地址减去一次写突发传输的地址。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            wr_burst_max_addr <= app_addr_wr_max;
        end
```

```
        else begin
            wr_burst_max_addr <= app_addr_wr_max - app_wr_bust_len * (DDR_DATA_W/8);
        end
    end

    //生成MIG IP的写地址，初始值为写入数据的最小地址。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            wr_addr_cnt <= app_addr_wr_min;
        end
        else if(wfifo_wr_rst)begin//复位时地址回到最小值;
            wr_addr_cnt <= app_addr_wr_min;
        end
        //当完成一次写突发地址传输完成之后。
        else if(s_axi_awvalid && s_axi_awready)begin
            if(wr_addr_cnt >= wr_burst_max_addr)//如果达到最大地址，则清零。
                wr_addr_cnt <= app_addr_wr_min;
            else//否则，增加地址数据作为下次突发写的首地址，每次增加app_wr_bust_len * (DDR_DATA_W/8)
                wr_addr_cnt <= wr_addr_cnt + app_wr_bust_len * (DDR_DATA_W/8);
        end
    end
```

此处只是写地址的计数器，并不是写地址信号，写地址信号在下文生成。

然后写地址有效指示信号与从机的应答信号握手，如下所示。

```
    //生成写地址有效指示信号，初始值为0。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            s_axi_awvalid <= 1'b0;
        end
        else if(wr_start)begin//当开始写信号有效时拉高。
            s_axi_awvalid <= 1'b1;
        end
        else if(s_axi_awready)begin//当从机应答后拉低。
            s_axi_awvalid <= 1'b0;
```

```
        end
    end
```

之后开始生成写数据通道的相关信号，首先需要一个计数器wr_bust_cnt对写入数据个数计数，当写数据通道握手时加1，当写入用户指定数据时清零。

写FIFO的读使能直接使用该计数器的加一条件，读出的数据直接作为axi_full接口的写数据。特别需要注意，将写数据掩码信号所有位拉高。

```
    /*****************生成写数据通道的信号*****************/
    //生成写FIFO的读使能信号，初始值为0.
    assign wfifo_rd_en = add_wr_bust_cnt;//当写入一次数据完成时拉高，其余时间均为低电平。
    assign s_axi_wdata = wfifo_rdata;//将FIFO输出数据直接赋值给AXI写数据接口。
    assign s_axi_wstrb = {{DDR_DATA_W/8}{1'b1}};//默认从FIFO中读出的128位数据都是有效数据，均需要写入DDR3中，掩码信号无效。

    //写数据突发计数器，初始值为0。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0.
            wr_bust_cnt <= 0;
        end
        else if(add_wr_bust_cnt)begin
            if(end_wr_bust_cnt)
                wr_bust_cnt <= 0;
            else
                wr_bust_cnt <= wr_bust_cnt + 1;
        end
    end

    assign add_wr_bust_cnt = (s_axi_wvalid && s_axi_wready);//当写入数据时加1。
    assign end_wr_bust_cnt = add_wr_bust_cnt && (wr_bust_cnt == app_wr_bust_len - 1);//当写入一次突发的数据时清零。

```

然后生成写数据通道的握手信号和写入最后一个数据的标志信号，如下所示。

```
    //生成写突发数据有效指示信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；

```

```
            s_axi_wvalid <= 1'b0;
        end
        else if(end_wr_bust_cnt)begin//当写突发结束时拉低；
            s_axi_wvalid <= 1'b0;
        end
        else if(s_axi_awvalid && s_axi_awready)begin//当开始写地址信号写入有效时拉高。
            s_axi_wvalid <= 1'b1;
        end
    end
```

之后写应答通道握手，表示一次写突发传输结束。

```
    //生成写突发结束信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            s_axi_wlast <= 1'b0;
        end
        else if(end_wr_bust_cnt)begin//当写突发结束时拉低；
            s_axi_wlast <= 1'b0;
        end
        else if(add_wr_bust_cnt && (wr_bust_cnt == app_wr_bust_len - 2))begin//当写入最后一次数据时拉高；
            s_axi_wlast <= 1'b1;
        end
    end

    /*****************生成写应答通道的信号*****************/
    //生成应答通道的主机应答信号。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            s_axi_bready <= 1'b0;
        end
        else if(s_axi_bvalid)begin//当从机的应答信号有效时拉低。
            s_axi_bready <= 1'b0;
        end
        else if(s_axi_wlast && add_wr_bust_cnt)begin//当一次写突发操作完成时拉高。
            s_axi_bready <= 1'b1;
        end
    end
```

2.3、axi_full读操作

读地址通道的多数信号与写地址通道一致，就不再详细讲解，如下所示：

```
    /*****************生成写应答通道的信号*****************/
    //生成应答通道的主机应答信号。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            s_axi_bready <= 1'b0;
        end
        else if(s_axi_bvalid)begin//当从机的应答信号有效时拉低。
            s_axi_bready <= 1'b0;
        end
        else if(s_axi_wlast && add_wr_bust_cnt)begin//当一次写突发操作完成时拉高。
            s_axi_bready <= 1'b1;
        end
    end

    /*****************生成突发读的开始信号和标志信号*****************/
    //写开始信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rd_start <= 1'b0;
        end
        else begin//当DDR3初始化完成 且 写FIFO不处于复位状态 且 读FIFO中的数据小于一次写突发传输的数据 且 不处于读数据状态时拉高。
            rd_start <= ddr3_init_done && (~rfifo_wr_rst_busy) && (rfifo_wdata_count < app_rd_bust_len - 2) && (~rd_flag) && ddr3_read_valid;
        end
    end

    //写数据标志信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rd_flag <= 1'b0;
        end
        else if(s_axi_rlast & s_axi_rvalid)begin//当读出最后一个数据时拉低。

```

```
            rd_flag <= 1'b0;
        end//当开始写入数据时拉高；
        else if(ddr3_init_done && (~rfifo_wr_rst_busy) && (rfifo_wdata_count < app_rd_bust_len - 2) && (~rd_flag) && ddr3_read_valid)begin
            rd_flag <= 1'b1;
        end
    end

    /*****************生成读地址通道的信号*****************/
    assign s_axi_arid    = 4'd0;//读地址ID；
    assign s_axi_arlen   = app_rd_bust_len - 1;//突发读的长度-1；
    assign s_axi_arsize  = DDR_DATA_BYTE_W;//突发读的大小，每次读数据的字节数；
    assign s_axi_arburst = 2'b01;//突发读的类型；
    assign s_axi_arlock  = 1'b0;//锁类型；
    assign s_axi_arcache = 4'b0000;//存储类型；
    assign s_axi_arprot  = 3'b000;//保护类型，表示传输的特权级及安全等级；
    assign s_axi_arqos   = 4'b0000;//质量服务；

    //计算读突发传输最大地址，等于用户设置最大地址减去一次读突发传输的地址。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rd_burst_max_addr <= 0;
        end
        else begin
            rd_burst_max_addr <= app_addr_rd_max - app_rd_bust_len * (DDR_DATA_W/8);
        end
    end

    //生成MIG IP的读地址，初始值为读出数据的最小地址。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rd_addr_cnt <= app_addr_rd_min;
        end
        else if(rfifo_rd_rst)begin//复位时地址回到最小值；
            rd_addr_cnt <= app_addr_rd_min;
        end
        //当完成一次地址写入后进行判断。
        else if(s_axi_arvalid && s_axi_arready)begin
            if(rd_addr_cnt >= rd_burst_max_addr)//如果达到最大地址，则清零。
                rd_addr_cnt <= app_addr_rd_min;
            else//否则，增加地址数据作为下次突发写的首地址，每次增加app_rd_bust_len * (DDR_DATA_W/8)
```

```
                rd_addr_cnt <= rd_addr_cnt + app_rd_bust_len * (DDR_DATA_W/8);
        end
    end

     //生成读地址有效指示信号，初始值为0。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            s_axi_arvalid <= 1'b0;
        end
        else if(rd_start)begin//当开始读信号有效时拉高。
            s_axi_arvalid <= 1'b1;
        end
        else if(s_axi_arready)begin//当从机应答后拉低。
            s_axi_arvalid <= 1'b0;
        end
    end
```

下面是axi_full读、写地址信号，在综合时通过设置是否支持乒乓操作，从而综合处不同功能的电路，与前面DDR3读写模块的设计基本一致，区别在于本模块的读写操作可能同时进行，因此需要单独生成读、写地址。

乒乓操作时，当写完一页之后，向另一段地址中写入数据。始终读与写地址相反的一段，从而避免图像刷新的残影问题。不启用乒乓操作时，读写操作将在同一段地址上进行。

```
    //根据是否使用乒乓功能，综合成不同的电路；
    generate
        if(PINGPANG_EN)begin//如果使能乒乓操作，地址信号将执行下列信号；
            reg  waddr_page ;
            reg  raddr_page ;
            //相当于把bank地址进行调整，使得读写的地址空间不再同一个范围；
            always@(posedge ui_clk)begin
                if(ui_clk_sync_rst)begin//最开始向第1页中写入数据；
                    waddr_page <= 1'b1;
                end//当写完1页数据后，接下来写下一页数据；
                else if(s_axi_bvalid && s_axi_bready && (wr_addr_cnt >= wr_burst_max_addr))begin
                    waddr_page <= ~waddr_page;
                end
            end
```

```
15
16              always@(posedge ui_clk)begin
17                  if(ui_clk_sync_rst)begin//最开始从第0页中读出数据；
18                      raddr_page <= 1'b0;
19                  end//当读完0页数据后，之后读取与写相反页的数据；
20                  else if(s_axi_rlast && (rd_addr_cnt >= rd_burst_max_addr))begin
21                      raddr_page <= ~waddr_page;
22                  end
23              end
24
25              //生成写地址信号；
26              always@(posedge ui_clk)begin
27                  if(ui_clk_sync_rst)begin//初始值为0；
28                      s_axi_awaddr <= 0;
29                  end//开始写入数据时，输出写地址信号；
30                  else if(wr_start)begin
31                      s_axi_awaddr <= {2'b0,waddr_page,wr_addr_cnt[25:0]};
32                  end
33              end
34
35              //生成读地址信号；
36              always@(posedge ui_clk)begin
37                  if(ui_clk_sync_rst)begin//初始值为0；
38                      s_axi_araddr <= 0;
39                  end//开始读出数据时，输出读地址信号；
40                  else if(rd_start)begin
41                      s_axi_araddr <= {2'b0,raddr_page,rd_addr_cnt[25:0]};
42                  end
43              end
44          end
45          else begin//不需要乒乓操作；
46              //生成写地址信号；
47              always@(posedge ui_clk)begin
48                  if(ui_clk_sync_rst)begin//初始值为0；
49                      s_axi_awaddr <= 0;
50                  end//开始写入数据时，输出写地址信号；
51                  else if(wr_start)begin
52                      s_axi_awaddr <= {3'b0,wr_addr_cnt[25:0]};
53                  end
54              end
55
```

```
            //生成读地址信号；
            always@(posedge ui_clk)begin
                if(ui_clk_sync_rst)begin//初始值为0；
                    s_axi_araddr <= 0;
                end//开始读出数据时，输出读地址信号；
                else if(rd_start)begin
                    s_axi_araddr <= {3'b0,rd_addr_cnt[25:0]};
                end
            end
        end
    endgenerate
```

之后将读出的数据存入读FIFO中，由于不需要统计个数，因此可以不设计计数器。

```
    /*****************生成读数据通道的信号*****************/
    //生成读FIFO的写使能信号和写数据。
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            rfifo_wr_en <= 1'b0;
            rfifo_wdata <= 0;
        end
        else begin//将MIG读出数据存入读FIFO中。
            rfifo_wr_en <= s_axi_rvalid;
            rfifo_wdata <= s_axi_rvalid ? s_axi_rdata : rfifo_wdata;
        end
    end

    //生成读数据通道的应答信号；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0；
            s_axi_rready <= 1'b0;
        end
        else if(s_axi_rlast & s_axi_rvalid)begin//当读出所有数据时拉低。
            s_axi_rready <= 1'b0;
        end
```

```
        else if(s_axi_arvalid && s_axi_arready)begin//当读地址发送完成后拉高；
            s_axi_rready <= 1'b1;
        end
    end
```

同样会生成一个读使能信号，初始值为0，当所有地址写入一遍数据后拉高，之后才能进行读操作，保证每次都能读出有效数据。

```
    //生成读使能信号，最开始的时候DDR3中并没有数据，必须向DDR3中写入数据后才能从DDR3中读取数据；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            ddr3_read_valid <= 1'b0;
        end//当写入一帧数据之后拉高，之后保持高电平不变。
        else if(wr_addr_cnt >= wr_burst_max_addr)begin
            ddr3_read_valid <= 1'b1;
        end
    end
```

2.4、FIFO复位

首先给用户提供两个复位引脚，用来控制两个FIFO的复位，上升沿有效。需要把用户输入的复位信号使用两级D触发器进行同步，然后检测器上升沿。

```
    //后面考虑复位信号的处理，复位的时候应该对FIFO和写地址一起复位，复位FIFO需要复位信号持续多个时钟周期；
    //因此需要计数器，由于读写的复位是独立的，可能同时到达，因此计数器不能共用。
    //写复位到达时，如果状态机位于写数据状态，应该回到初始状态，等待清零完成后再进行跳转。
    //同步两个FIFO复位信号，并且检测上升沿，用于清零读写DDR的地址，由于状态机跳转会检测FIFO是否位于复位状态。
    always@(posedge ui_clk)begin
        wr_rst_r <= {wr_rst_r[0],wr_rst};//同步复位脉冲信号；
        rd_rst_r <= {rd_rst_r[0],rd_rst};//同步复位脉冲信号；
    end
```

检测到用户输入写复位上升沿之后，将写FIFO的复位信号拉高，使用一个计数器对时钟计数，当复位计数器溢出时，FIFO复位信号拉低，从而完成对FIFO复位。可以通过修改复位计数器位宽，来调节复位电平位宽。

```
    //生成写复位信号，由于需要对写FIFO进行复位，所以复位信号必须持续多个时钟周期；
    always@(posedge ui_clk)begin
```

```
        if(ui_clk_sync_rst)begin//初始值为0;
            wfifo_wr_rst <= 1'b0;
        end
        else if(wr_rst_r[0] && (~wr_rst_r[1]))begin//检测wfifo_wr_rst上升沿拉高复位信号;
            wfifo_wr_rst <= 1'b1;
        end//当写复位计数器全为高电平时拉低，目前是持续32个时钟周期，如果不够，修改wrst_cnt位宽即可。
        else if(&wr_rst_cnt)begin
            wfifo_wr_rst <= 1'b0;
        end
    end

    //写复位计数器，初始值为0，之后一直对写复位信号持续的时钟个数进行计数;
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            wr_rst_cnt <= 0;
        end
        else if(wfifo_wr_rst)begin
            wr_rst_cnt <= wr_rst_cnt + 1;
        end
    end
```

读FIFO的复位信号生成如下所示，与写FIFO复位信号生成一致。

```
    //写复位信号，初始值为0，当读FIFO读复位下降沿到达时有效，当计数器计数结束时清零;
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rfifo_rd_rst <= 1'b0;
        end
        else if(rd_rst_r[0] && (~rd_rst_r[1]))begin
            rfifo_rd_rst <= 1'b1;
        end
        else if(&rd_rst_cnt)begin
            rfifo_rd_rst <= 1'b0;
        end
    end

```

```
    //读复位计数器，初始值为0，当读复位有效时进行计数；
    always@(posedge ui_clk)begin
        if(ui_clk_sync_rst)begin//初始值为0;
            rd_rst_cnt <= 0;
        end
        else if(rfifo_rd_rst)begin
            rd_rst_cnt <= rd_rst_cnt + 1;
        end
    end
```

3、仿真

该模块的仿真流程与前文的DDR3读写控制模块一致，可以使用同一套TestBench，仿真结果如下，将突发长度设置为128，最大地址设置为8192，即进行四次写操作之后就能进行读操作，减小仿真时间。

在DDR3初始化完成之后，首先对写FIFO复位，清除FIFO中的数据（注意xilinx的FIFO在仿真时必须复位之后才能写入数据）。

下图黄色信号是写FIFO复位信号，高电平有效，粉色信号是复位状态指示信号，高电平表示FIFO未初始化或者处于复位状态。

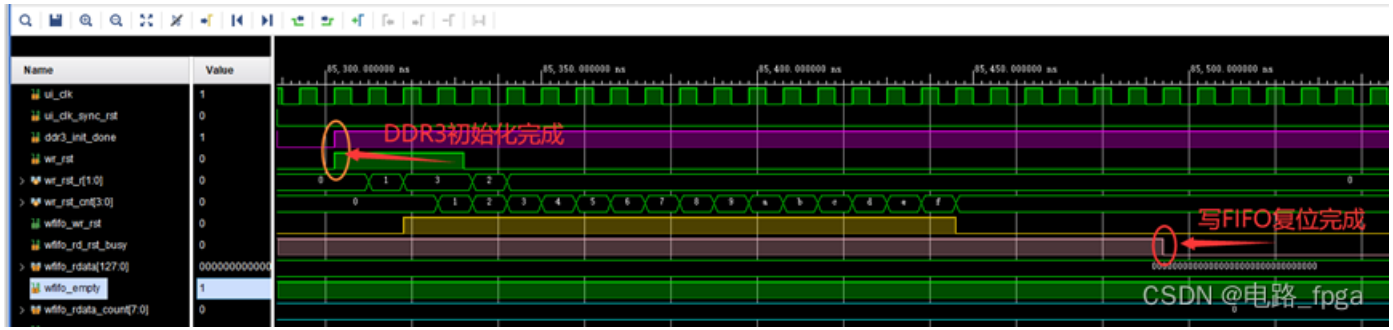

图4 写FIFO复位

当写FIFO中的数据大于等于一次写突发长度时，将写标志和开始信号拉高，写地址通道传输数据，完成握手后，从FIFO中读取数据写入MIG IP中，使用计数器对写入数据个数计数，写数据完成后。在写应答通道进行握手，最后写标志信号拉低，完成一次突发传输，对应的仿真如下图所示。

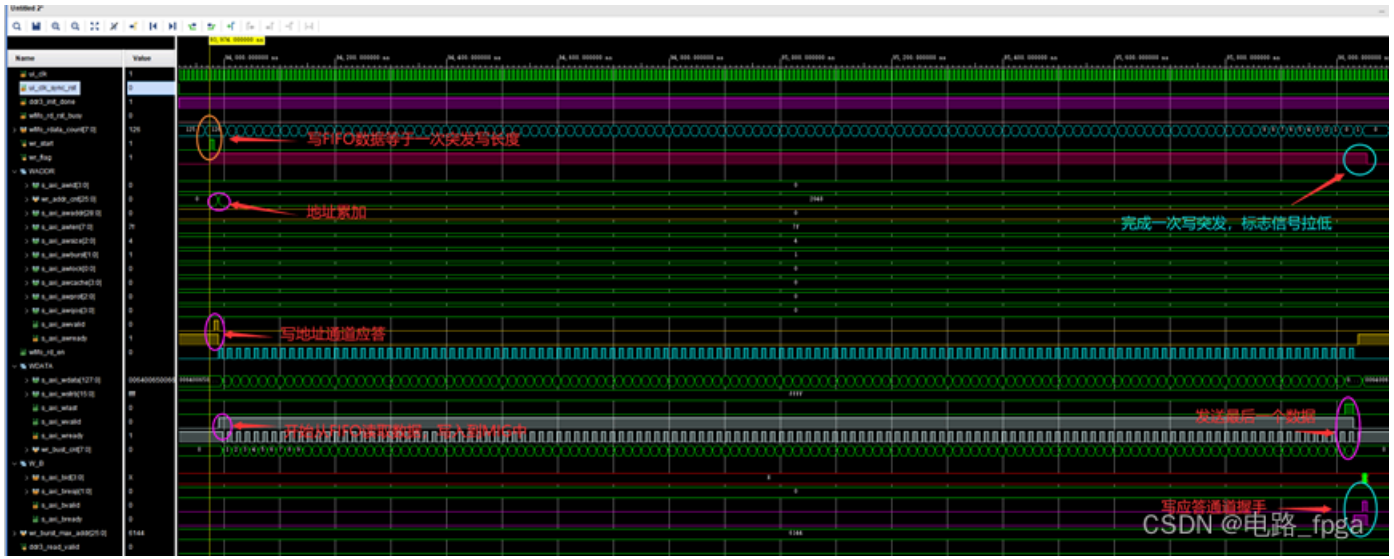

图5 写突发全局仿真

将上图的开始和结尾放大，分别如下图所示，记录数据，方便与后文读出数据进行对比。

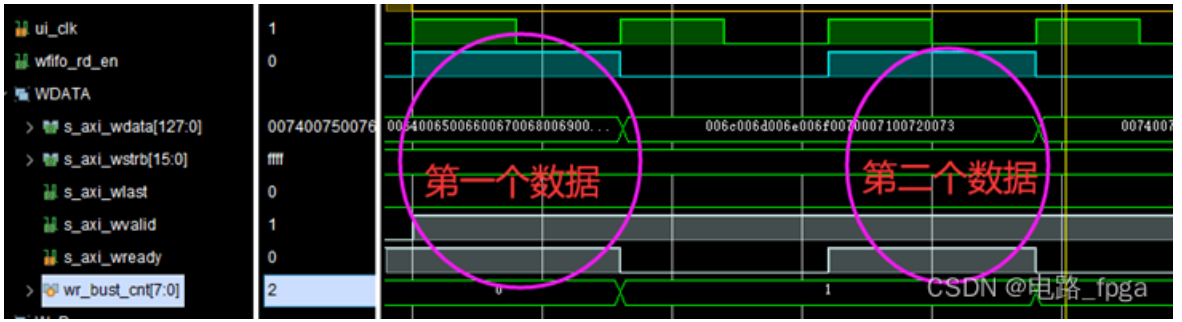

图6 开始写入数据

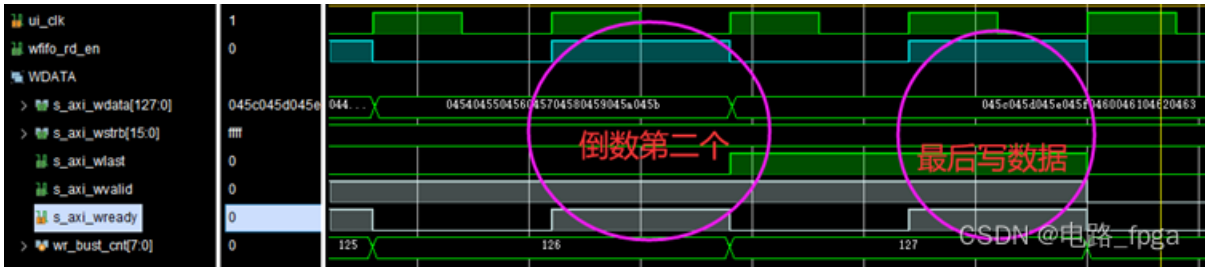

图7 写入结束数据

当所有地址都写入数据后，读FIFO中的数据少于一次突发传输数据，将读开始信号和读标志信号拉高。读地址通道先传输数据进行握手，之后主机开始接收从机数据输出，将接收的数据写入读FIFO中。当读出最后一个数据后，将读标志信号拉低，表示一次读突发传输完成，对应的时序图如下所示。

图8 读突发传输时序

将上图的开始和结尾放大，与前文写入数据进行对比，进而判断读、写操作是否有误。

由于读操作从机输出数据不连续，会有多张图，如下是读出第一个数据，与图6中写入的第一个数据一致。

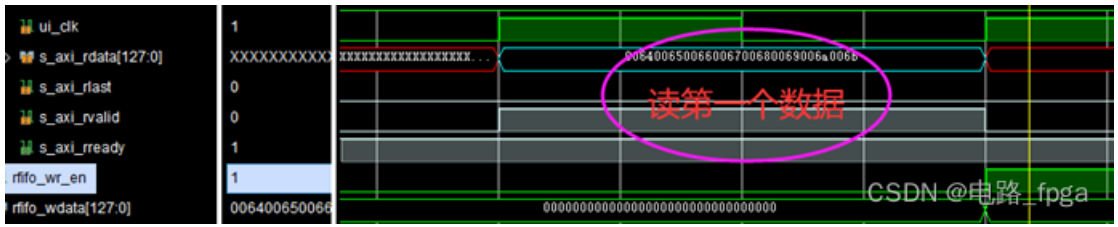

图9 读第一个数据

下图是读出的第二个数据，与图6中写入的第二个数据一致。

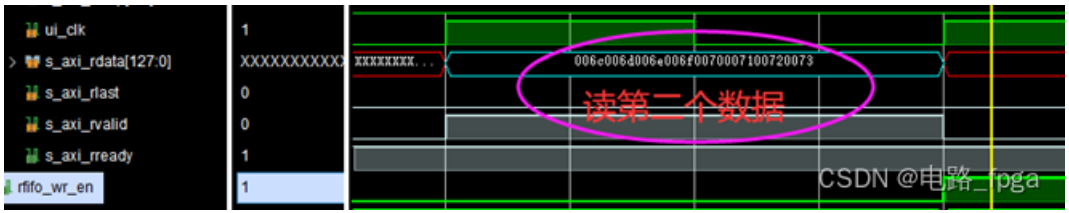

图10 读第二个数据

下图是读出一次突发最后的数据，与图7中最后写入两个数据一致，由此证明在读写过程中没有数据丢失。

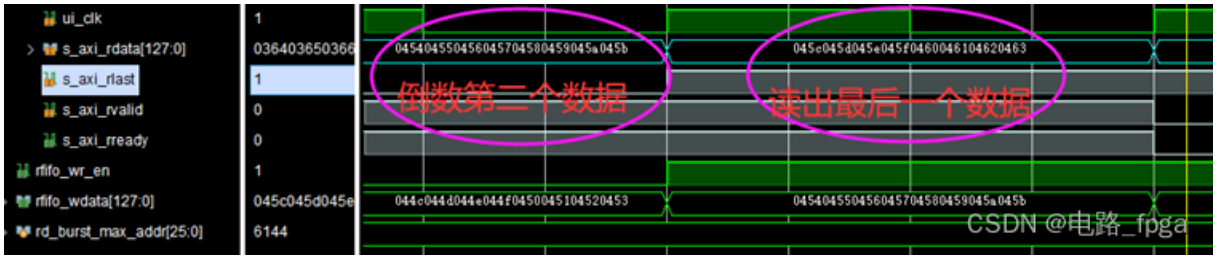

图11 读最后两个数据

关于仿真就到此结束吧，如果需要详细了解，自己可以在工程中进行查看。

4、上板实测

本文直接将此模块添加到以太网传输图片的工程中，替换之前的MIG IP和DDR3读写控制模块以及DDR3顶层模块，其余模块均保持不动，由于信号过多，代码太长，文中不贴模块代码，需要的可以获取工程自行查看，之后进行上板测试。

DDR3模块顶层RTL视图如下所示，端口信号稍微有点多。

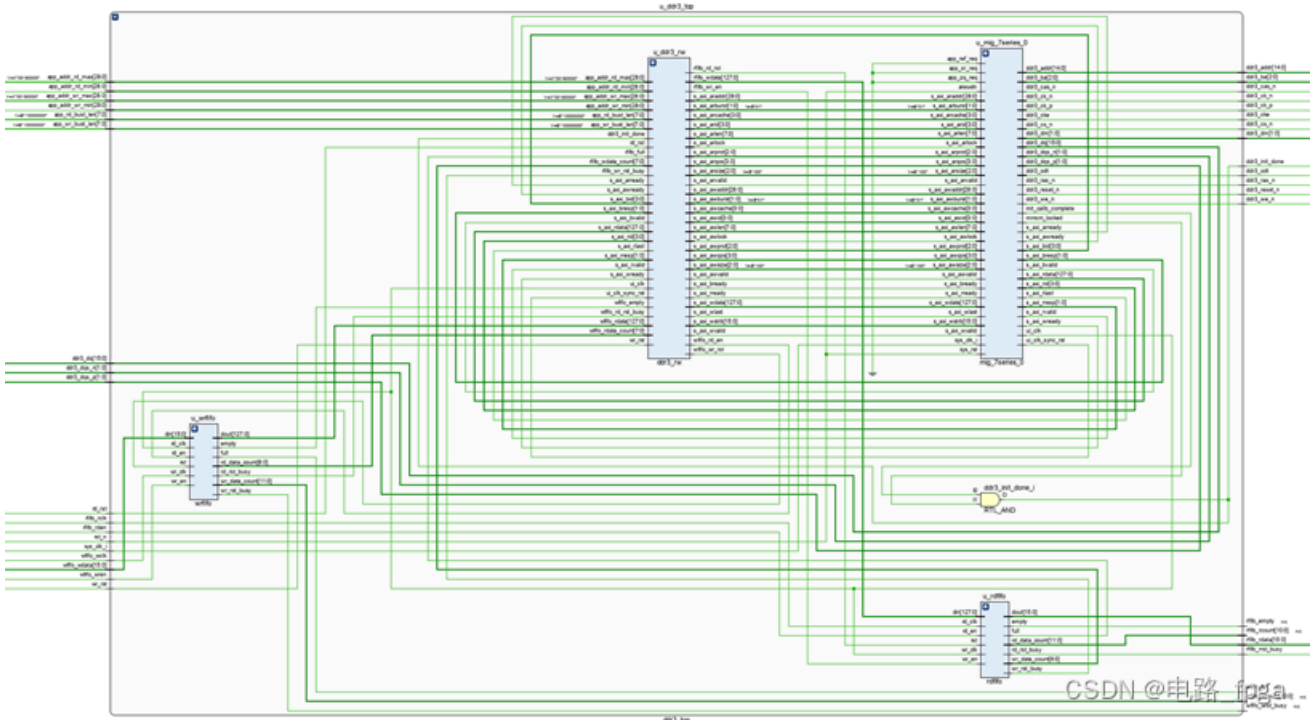

图12 DDR3顶层模块

之后综合工程，对乒乓操作和一般模式都进行验证，从而确认该模块的设计是否存在问题。

在bit下载到开发板之后，开发板连接以太网HDMI显示器，然后打开命令提示符，使用arp -a和ping指令查看以太网链路是否畅通，这些步骤在前文均有讲解，不再赘述。

当ping指令应答后，如下图所示，使用arp -a查看开发板的MAC地址和IP地址是否绑定。

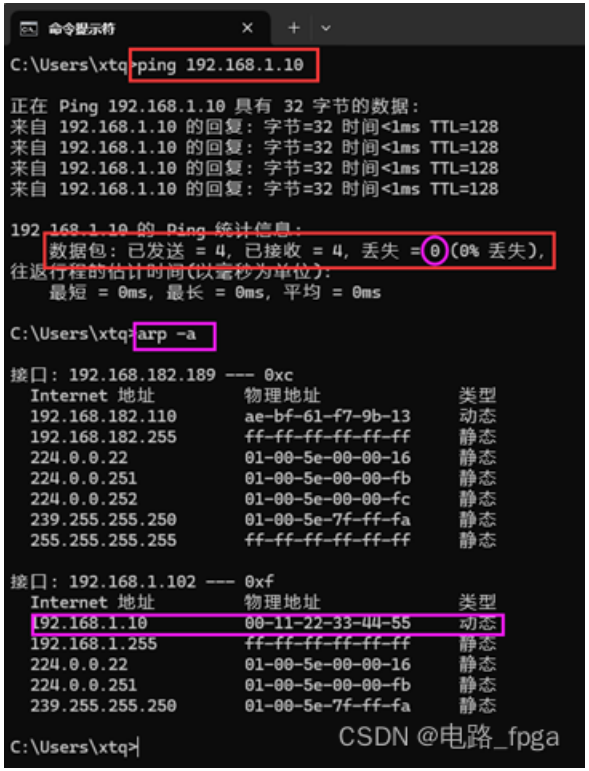

图13 MAC与IP绑定

之后打开网络调试助手，向开发板传输图片，最终显示结果如下所示。

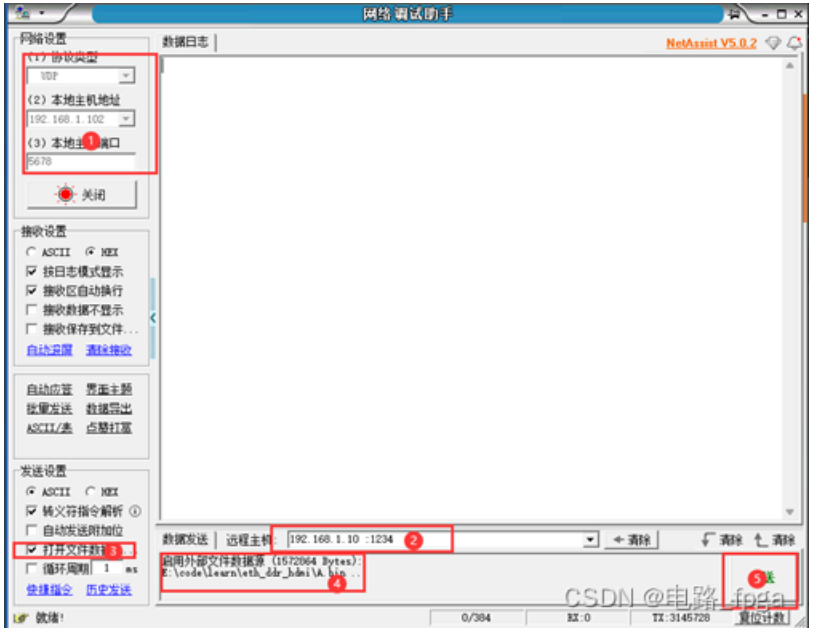

图14 网络调试助手发送图片

传输的图片在HDMI显示器上显示结果如下：

图15 显示器显示图片

之后打开网络调试助手传输第二张图片，结果如下视频所示，不使用乒乓操作将出现这种刷新画面。

图16 图片刷新

然后将DDR3顶层模块的乒乓使能参数设置为1，启用乒乓操作，重新综合工程，然后下载到开发板进行验证，发送第二张图片的过程如下，不会出现上面视频的现象，这就是乒乓操作的作用。

图17图片刷新

由此正常该模块的设计没有问题，其实axi_full接口还是比较简单，而且MIG IP自己就解决了仲裁问题，可以省去一些麻烦。

通过手动实现axi_full接口协议之后，应该不会害怕信号多的接口了吧，在设计时，每次只考虑相关的信号，对于无关信号可以暂时不考虑，这样就可以简化设计思路。

比如在设计写数据通道信号时，读数据相关的信号便可以不考虑，反之如此。

如果需要此次工程，依然在后台回复“**mig的axi_full接口应用**”（不包括引号）。

如果对文章内容理解有疑惑或者对代码不理解，可以在评论区或者后台留言，看到后均会回复！

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