将Xilinx DDR3 MIG IP核的APP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。

1、概括

前文完成了xilinx DDR3 MIG IP的仿真和上板测试，对 MIG IP的读、写需要去通过使能信号和应答信号进行握手。这对于图像处理、AD采集等大量数据的存储不太方便，常见的使用方式是把MIG IP的用户接口封装成FIFO的接口。

如下图所示，如果要存储大量数据，只需要在开始时指定存储数据的起始地址和终止地址，并且确定每次写入数据的个数，即可向wr_fifo中写入数据。当wr_fifo中的数据个数大于一次突发写入长度时，ddr3_rw模块读取wr_fifo中的数据，以MIG用户接口写入数据的格式，将数据写到DDR3 中。

对于读数据也是同样的道理，需要确定起始读数据的起止地址和每次突发读数据的长度。当ddr3_rw检测到rd_fifo中的数据少于一次突发传输的数据时，通过MIG从DDR3中读出一次突发长度的数据到rd_fifo中。用户只需要通过拉高rd_fifo的读使能信号，即可获取数据。

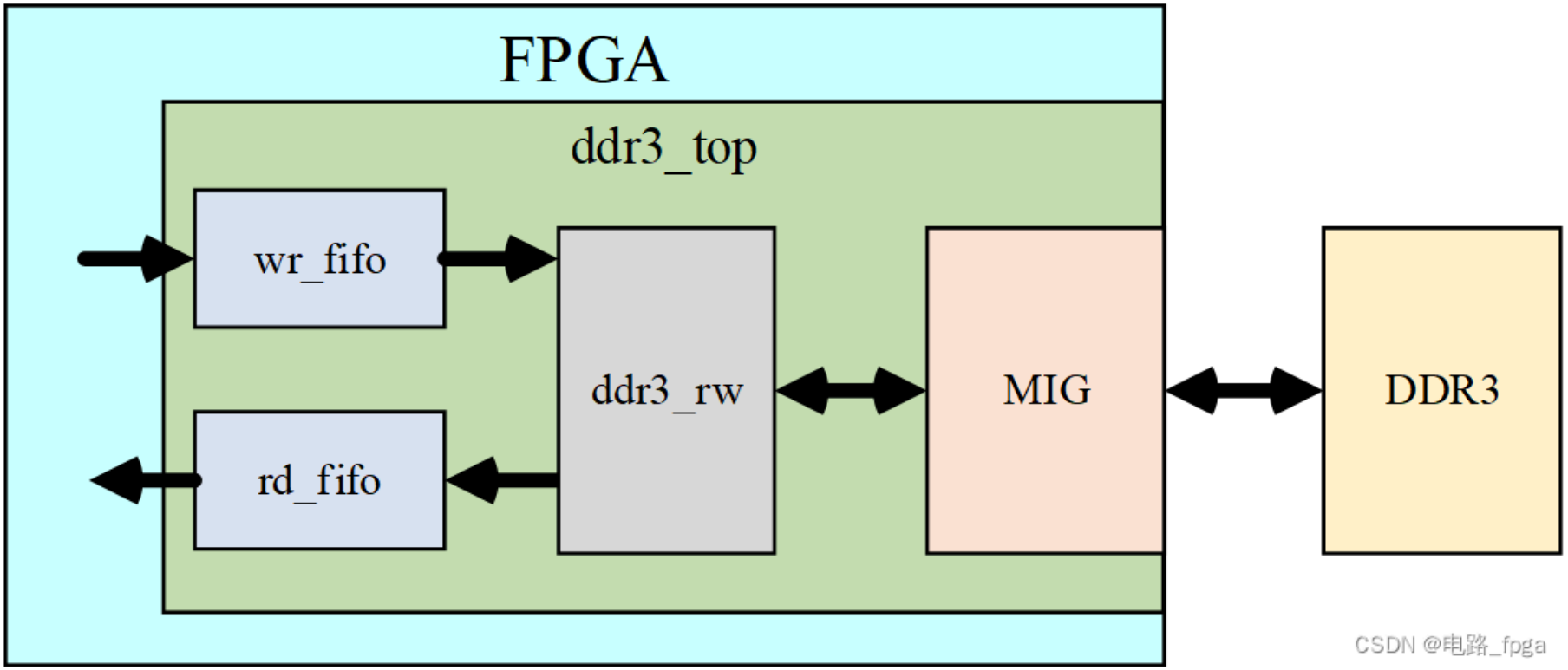

图1 ddr3封装示意图

上述两个fifo主要是存储读写的数据，同时处理数据跨时钟域的问题。读写端口均支持复位功能，当复位有效时，会将各自fifo复位，并且把ddr3_rw中相应的地址复位。

2、ddr3_rw设计

该模块主要负责将写FIFO中的数据写入DDR3中，从DDR3中指定地址读出数据存储到读FIFO中，处理复位相关问题。

在实际使用的过程中，比如图像传输，可能导致读写的地址范围出现在DDR3的同一段地址区域，整张画面造成上半部分界面已经刷新，下半部分还没刷新的尴尬局面，这种现象在后文验证模块的时候会有体现。所以本模块增加一个乒乓模式，该模式将DDR3的读、写放在两段不同的地址，读取的数据始终是更新完成的数据，就不会出现上述现象。

本模块以状态机作为主体，下图为状态转换图（由于文字较多，所以看起来不咋清晰）。

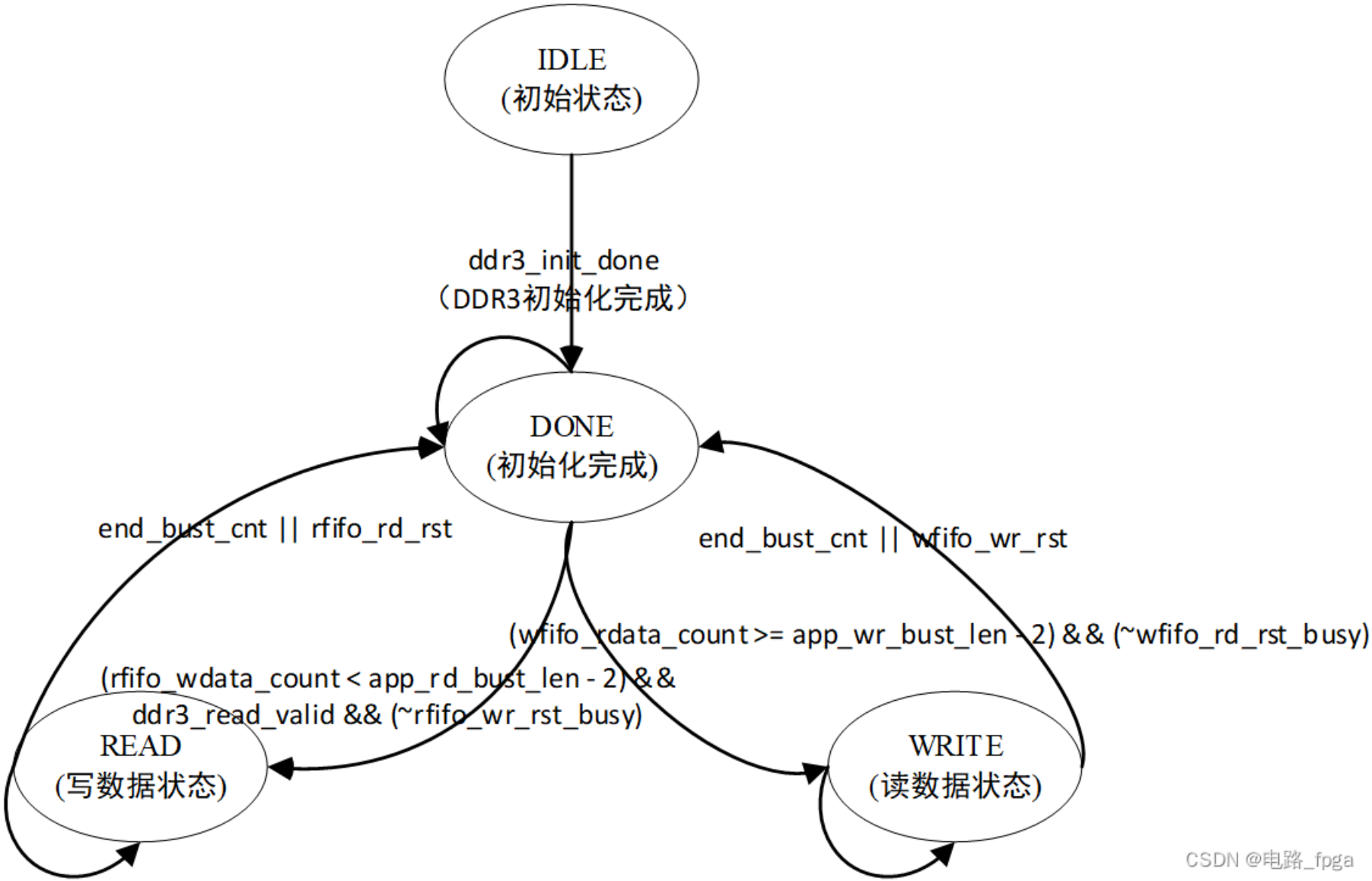

图2 状态转换图

end_bust_cnt表示一次突发读或写完成，rfifo_rd_rst表示读FIFO复位，wfifo_wr_rst表示写FIFO复位，rfifo_wdata_count表示读FIFO以写侧数据位宽和时钟查看的数据个数，app_rd_bust_len表示一次读突发传输的数据个数，rfifo_wr_rst_busy高电平表示读FIFO处于复位状态，写FIFO对应的信号含义一致。

读侧多一个信号ddr3_read_valid，思考一个问题，当读FIFO中数据不足一次突发传输时，就会从DDR3中读取数据，最开始读FIFO中没有数据，那么会立即从DDR3中读取数据，但是刚上电时DDR3中也没有有效的数据，所以此时不能从DDR3中读取数据。

本设计增加了一个ddr3_read_valid信号，当对DDR3起、止地址之间的空间全部写入数据后，将ddr3_read_valid拉高，此时才能从DDR3中读取数据填充到读FIFO中，以保证数据的正确性。

app_rd_bust_len减2是因为FIFO使用的超前模式，FIFO会在输出总线上输出一个数据，当读使能有效时，对应的数据就是有效的，读数据与读使能对齐，由于MIG IP的写需要应答，所以FIFO使用此模式会方便很多。

设计思路比较简单，下面依次解析下代码，如下所示，MIG IP输出的复位信号是高电平有效的，将其取反作为复位。MIG IP的命令使能和数据使能信号都需要应答，所以将应答信号均为高电平时才拉高对应的使能信号，确保命令和数据个数保持一致。

```
    assign rst_n = ~ui_clk_sync_rst;//将MIG IP输出的复位信号取反作为复位信号；

    //状态机在写状态MIG空闲且写有效，或者状态机在读状态MIG空闲时加1，其余时间为低电平；
    assign app_en = ((state_c == WRITE && app_rdy && app_wdf_rdy) || (state_c == READ && app_rdy));
    assign app_wdf_wren = (state_c == WRITE && app_rdy && app_wdf_rdy);//状态机在写状态且写入数据有效时拉高；
    assign app_wdf_end = app_wdf_wren;//由于DDR3芯片时钟和用户时钟的频率4:1，突发长度为8，故两个信号相同；
    assign app_cmd = (state_c == READ) ? 3'd1 :3'd0;//处于读的时候命令值为1，其他时候命令值为0;
    assign wfifo_rd_en = app_wdf_wren;//写FIFO读使能信号，读出数据与读使能对齐。
    assign app_wdf_data = wfifo_rdata;//将写FIFO读出的数据传输给MIG IP的写数据；
    assign rfifo_wr_en = app_rd_data_valid;//将MIG IP输出数据有效指示信号作为读FIFO的写使能信号；
    assign rfifo_wdata = app_rd_data;//将从MIG IP读出的数据作为读FIFO的写数据；
```

之所以使用组合逻辑，是因为app_en和app_wdf_en等信号均需要应答，如果使用时序逻辑，数据会延迟一个时钟周期，且整体逻辑（包含计数器）会麻烦很多。

下面是状态机的跳转，与状态转换图对应。

```
    //状态机次态到现态的跳转；
    always@(posedge ui_clk or negedge rst_n)begin
        if(!rst_n)begin//初始为空闲状态；
            state_c <= IDLE;
        end
        else begin
            state_c <= state_n;
        end
    end

    //状态机次态的跳转；
```

```
    always@(*)begin
        case(state_c)
            IDLE : begin
                if(init_calib_complete)begin//如果DDR3初始化完成，跳转到DDR3初始化完成状态；
                    state_n = DONE;
                end
                else begin
                    state_n = state_c;
                end
            end
            DONE : begin//如果写FIFO中数据多于一次写突发的长度且写FIFO不处于复位状态时，跳转到写状态；
                if((wfifo_rdata_count >= app_wr_bust_len - 2) && (~wfifo_rd_rst_busy))begin
                    state_n = WRITE;
                end//如果读FIFO中的数据少于一次读突发的长度且读FIFO不处于复位状态时，开始读出数据；
                else if((rfifo_wdata_count <= app_rd_bust_len - 2) && ddr3_read_valid && (~rfifo_wr_rst_busy))begin
                    state_n = READ;
                end
                else begin
                    state_n = state_c;
                end
            end
            WRITE : begin
                if(end_bust_cnt || wfifo_wr_rst)begin//写入指定个数的数据回到完成状态或者写复位信号有效；
                    state_n = DONE;
                end
                else begin
                    state_n = state_c;
                end
            end
            READ : begin
                if(end_bust_cnt || rfifo_rd_rst)begin//读出指定个数的数据回到完成状态或者读复位信号有效；
                    state_n = DONE;
                end
                else begin
                    state_n = state_c;
                end
            end
            default:begin
                state_n = IDLE;
            end
```

```
        endcase
    end
```

下面是一个突发计数器，当状态机在读状态或者写状态，发出对应的指令信号时加1，当计数到指定突发长度时清零。由于读、写突发的长度可能不同，所以需要一个信号来记录本次计数器的最大值。

```
    //突发读写个数计数器bust_cnt，用于记录突发读写的数据个数；
    always@(posedge ui_clk)begin
        if(rst_n==1'b0)begin//初始值为0；
            bust_cnt <= 0;
        end
        else if(state_c == DONE)begin//状态机位于初始化完成状态时清零；
            bust_cnt <= 0;
        end
        else if(add_bust_cnt)begin
            if(end_bust_cnt)
                bust_cnt <= 0;
            else
                bust_cnt <= bust_cnt + 1;
        end
    end

    //状态机在写状态MIG空闲且写有效且写FIFO中有数据，或者状态机在读状态MIG空闲且读FIFO未满时加1，其余时间为低电平；
    assign add_bust_cnt = ((state_c == WRITE && app_rdy && app_wdf_rdy) || (state_c == READ && app_rdy));
    assign end_bust_cnt = add_bust_cnt && bust_cnt == bust_cnt_num;//读写的突发长度可能不同，所以需要根据状态机的状态判断读写状态最大值；

    //用于存储突发的最大长度；
    always@(posedge ui_clk)begin
        if(state_c == READ)begin//如果状态机位于读状态，则计数器的最大值对应读突发的长度；
            bust_cnt_num <= app_rd_bust_len - 1;
        end
        else begin//否则为写突发的长度；
            bust_cnt_num <= app_wr_bust_len - 1;
        end
    end
```

然后就是DDR3读、写地址的控制了，由于读写操作可能穿插进行，比如一次读突发完成，然后进行写突发，所以需要两个计数器来对地址进行计数。

开始时为初始地址，当复位信号有效时回到初始地址，状态机在写状态，写到设置的最大地址时回到初始地址，否则每次写入数据后地址加8，MIG IP一次会向DDR3中8个地址写入数据（DDR3的8倍预取）。

```
    //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(rst_n==1'b0)begin//初始值为0;
            app_addr_wr <= app_addr_wr_min;
        end
        else if(wfifo_wr_rst)begin//复位时地址回到最小值；
            app_addr_wr <= app_addr_wr_min;
        end
        //当计数器加以条件有效且状态机处于写状态时，如果写入地址达到最大，则进行复位操作，否则加8；
        else if(add_bust_cnt && (state_c == WRITE))begin
            if(app_addr_wr >= app_addr_wr_max - 8)
                app_addr_wr <= app_addr_wr_min;
            else//否则，每次地址加8，因为DDR3每次突发会写入8次数据；
                app_addr_wr <= app_addr_wr + 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(rst_n==1'b0)begin//初始值为0;
            app_addr_rd <= app_addr_rd_min;
        end
        else if(rfifo_rd_rst)begin//复位时地址回到最小值；
            app_addr_rd <= app_addr_rd_min;
        end
        else if(add_bust_cnt && (state_c == READ))begin
            if(app_addr_rd >= app_addr_rd_max - 8)begin
                app_addr_rd <= app_addr_rd_min;
            end
            else
```

```
                app_addr_rd <= app_addr_rd + 8;
        end
    end
```

下面这段代码根据是否启用乒乓操作，综合出不同的电路，如果使用乒乓操作，那么需要使用一个读写页的控制信号，让读、写操作在不同的地址进行，始终读取完整数据的地址段。

```
    //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(rst_n==1'b0)begin
                    waddr_page <= 1'b1;
                    raddr_page <= 1'b0;
                end
                else if(add_bust_cnt)begin
                    if((state_c == WRITE) && (app_addr_wr >= app_addr_wr_max - 8))
                        waddr_page <= ~waddr_page;
                    else if((state_c == READ) && (app_addr_rd >= app_addr_rd_max - 8))
                        raddr_page <= ~waddr_page;
                end
            end
            //将数据读写地址赋给ddr地址
            always @(*) begin
                if(state_c == READ )
                    app_addr <= {2'b0,raddr_page,app_addr_rd[25:0]};
                else
                    app_addr <= {2'b0,waddr_page,app_addr_wr[25:0]};
            end
        end
        else begin//如果没有使能乒乓操作，则综合以下代码；
            //将数据读写地址赋给ddr地址
            always @(*) begin
```

```
                if(state_c == READ )
                    app_addr <= {3'b0,app_addr_rd[25:0]};
                else
                    app_addr <= {3'b0,app_addr_wr[25:0]};
            end
        end
    endgenerate
```

当一段地址的数据被写入完后，将地址的某一位翻转，之后写入的数据就在另一段地址中，读数据始终从写入完整数据的地址段读取数据，保证输出数据流的完整，不会出现正在更新的数据。

上电初始化之后，当规定的起止地址之间全部写入一次数据后，将该信号拉高，之后从该段地址中读取数据进行输出。

```
    //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(rst_n==1'b0)begin//初始值为0;
            ddr3_read_valid <= 1'b0;
        end//当状态机位于写状态写入一帧数据之后拉高，之后保持高电平不变。
        else if(app_addr_wr >= app_addr_wr_max - 8)begin
            ddr3_read_valid <= 1'b1;
        end
    end
```

下面这段代码就是为复位服务的，复位分为写侧和读侧的复位，复位来自读写时钟域，与MIG IP的ui_clk为异步信号，首先需要通过两个触发器同步。

对于写需要在写入数据前对写FIFO进行复位，所以检测其上升沿，然后将复位信号持续多个时钟周期（xilinx的FIFO复位需要多个时钟周期），对写FIFO复位。

```
    //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

```

```
//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(rst_n==1'b0)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(rst_n==1'b0)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
```

而对于读复位，一般是在读取一帧或者读完起止地址之间的数据后，对读FIFO进行复位，清除FIFO中的残留数据，从DDR3起始地址读取数据进行数据，防止上一帧数据的错误影响下一帧数据，与写复位原理一致。

```
//写复位信号，初始值为0，当读FIFO读复位上升沿到达时有效，当计数器计数结束时清零；
always@(posedge ui_clk)begin
    if(rst_n==1'b0)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(rst_n==1'b0)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
```

通过上述代码可知，读写DDR3数据的起、止地址，突发长度与ui_clk也是异步信号，但是并没有做异步处理。这是因为这些信号在使用时，可能很长时间（至少一帧数据的传输时间不变）不变，甚至可能是常数，所以没必要做异步处理。

整个模块的代码设计就完成了，比较简单，后面进行仿真和上板测试。

3、读、写FIFO

需要配置读、写FIFO IP，两个FIFO的读写数据信号位宽会有点区别，所以需要使用两个IP。当然可以使用一个IP，然后通过代码实现位宽的转变，这样代码会多一点，本文不采用。

对于写FIFO，一般输入数据的位宽为16位，输出数据作为MIG IP写数据，位宽为128位，输出采用超前模式，并且需要输出FIFO中的数据个数以及复位状态指示信号，相关配置如下图所示。

选择异步FIFO，如下图所示。

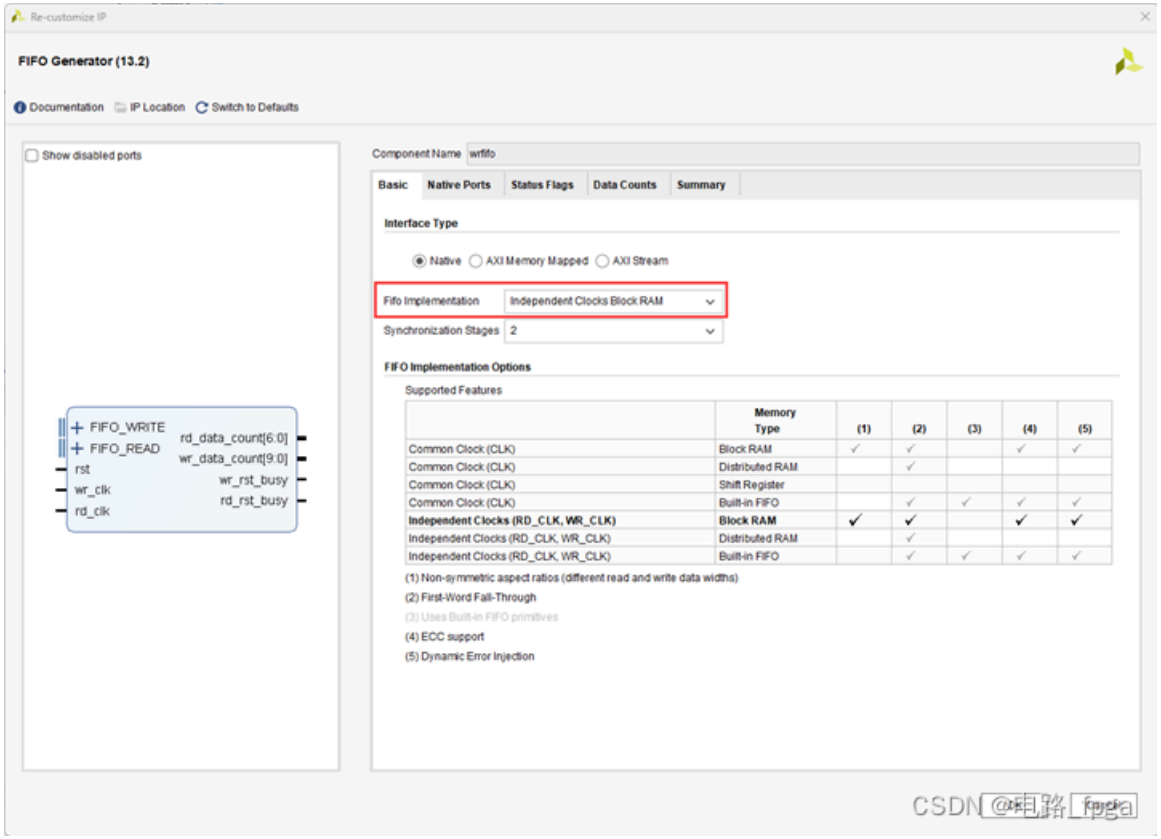

图3 异步FIFO配置

如下图所示，选择First Word Fall Through模式，在该模式下FIFO的读数据与读使能信号对齐。当读写控制模块需要数据时，就能立即得到数据，数据不会被延迟。

该FIFO输入数据位宽设置为16位，因为很多图像数据或者AD数据均为16位，当然后面也可以根据实际情况修改，不影响。

深度这里测试使用的1024，之后设置输出数据的位宽，因为MIG写数据的位宽为128位，因此设置为128，之后就会得到输出数据的深度。

因为读写数据的位宽不同，所以这个FIFO能够存储的数据个数对于读写数据来说也是不一样的。但是要保证输入数据位宽输入深度等于输出数据位宽输出深度。

另外注意FIFO工作在这种模式下其实就是输出数据线上提前输出了一个数据，当读使能有效后，下个时钟输出下一个数据。所以输出深度为128的FIFO实际能够存储129个数据。

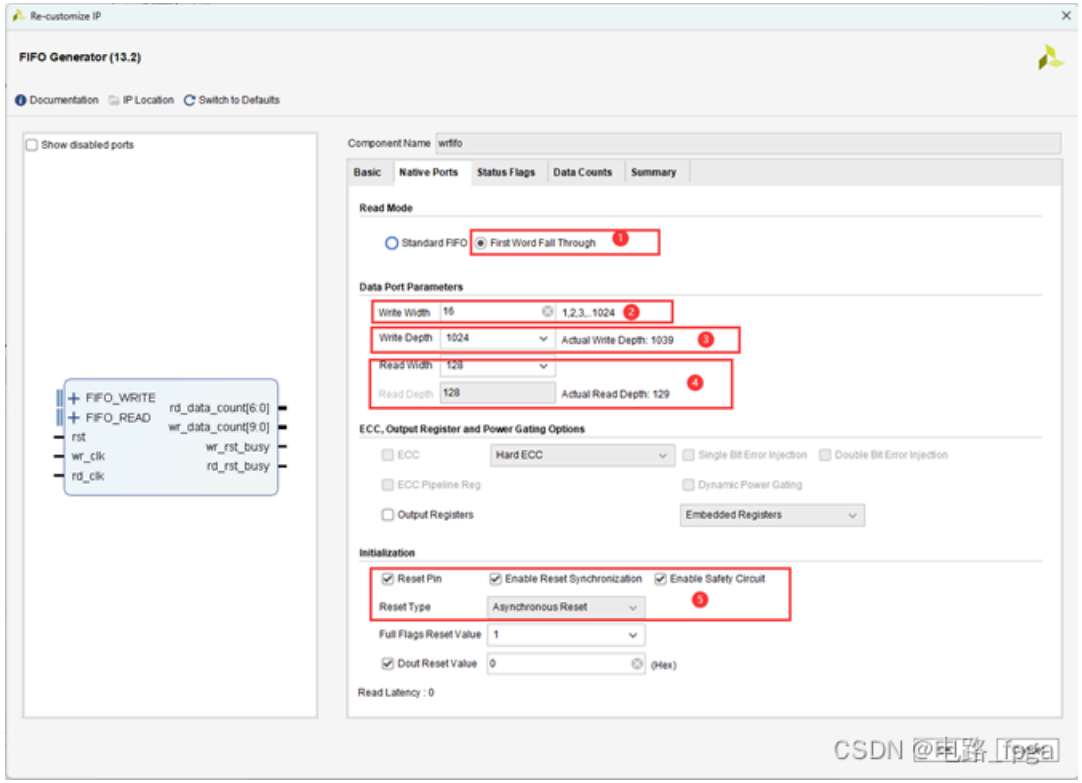

图4 配置输入输出数据位宽

注意复位信号的设置，采用高电平复位，并且将复位状态的指示信号输出，当FIFO不处于复位状态才对FIFO进行读、写操作。

通过下图设置，将指示FIFO中有多少个数据的信号输出，由于采用异步时钟信号，所以读、写时钟域都要输出一个信号。

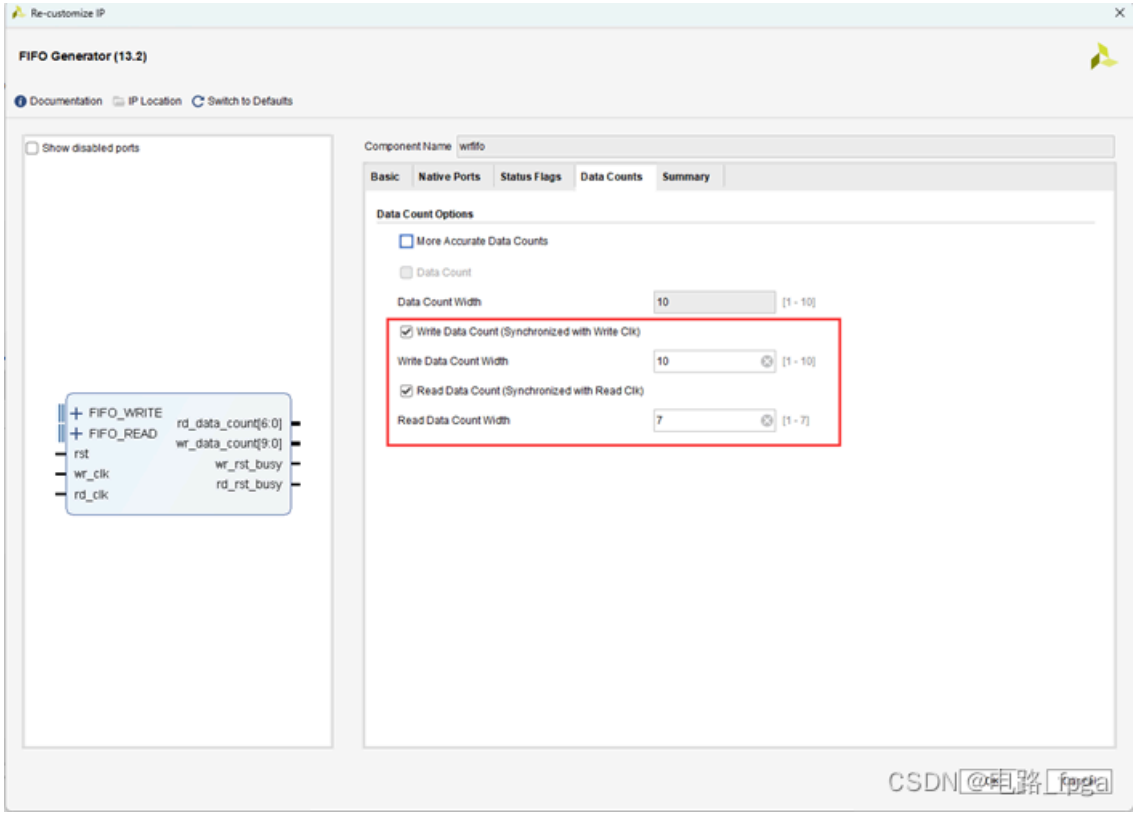

图5 输出FIFO中数据个数

写FIFO的相关设置就完成了，之后生成读FIFO，读FIFO与写FIFO只有读写数据位宽的设置不同，其余均一致。

由于MIG需要把从DDR3中读出的数据写入到读FIFO中，所以读FIFO写侧的数据位宽为128位。用户通过拉高读使能信号从读FIFO中获取数据，所以读FIFO读侧的数据位宽为16位。

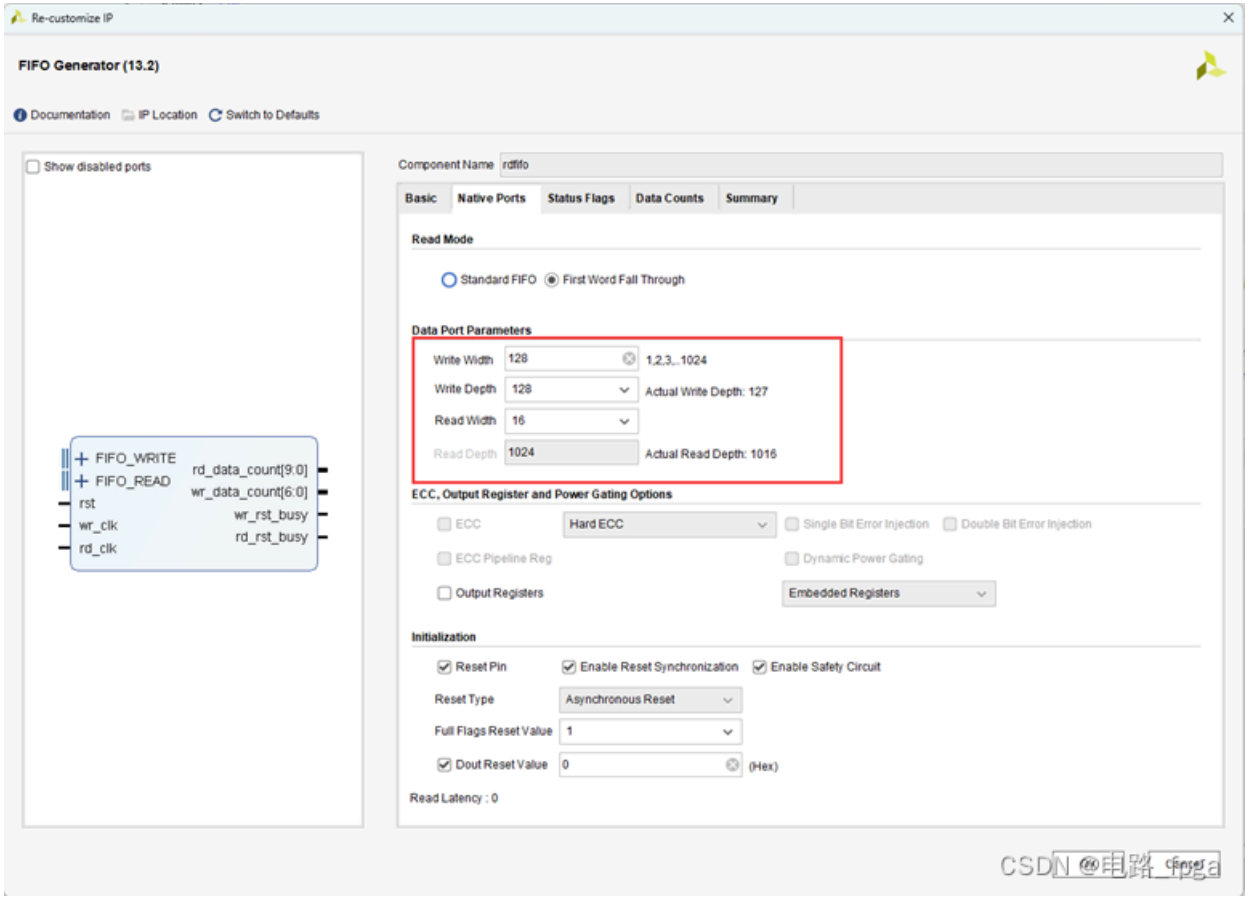

图6 读FIFO数据位宽设置

要特别注意上图中读FIFO的实际深度为1016，后面要考，在这里栽了一次。。。

关于读、写FIFO IP的相关配置就完成了，主要关注数据的流向，就能清除的理解各个细节了。

4、顶层模块

顶层模块需要把MIG IP与ddr3读写控制模块和读、写FIFO相应接口连起来，并且把用户接口和DDR3芯片接口引出，对应的顶层模块RTL视图如下所示。

注意读、写FIFO的复位信号宽度必须大于一个ui_clk的宽度，且FIFO不处于复位状态才能写入或者读出数据。

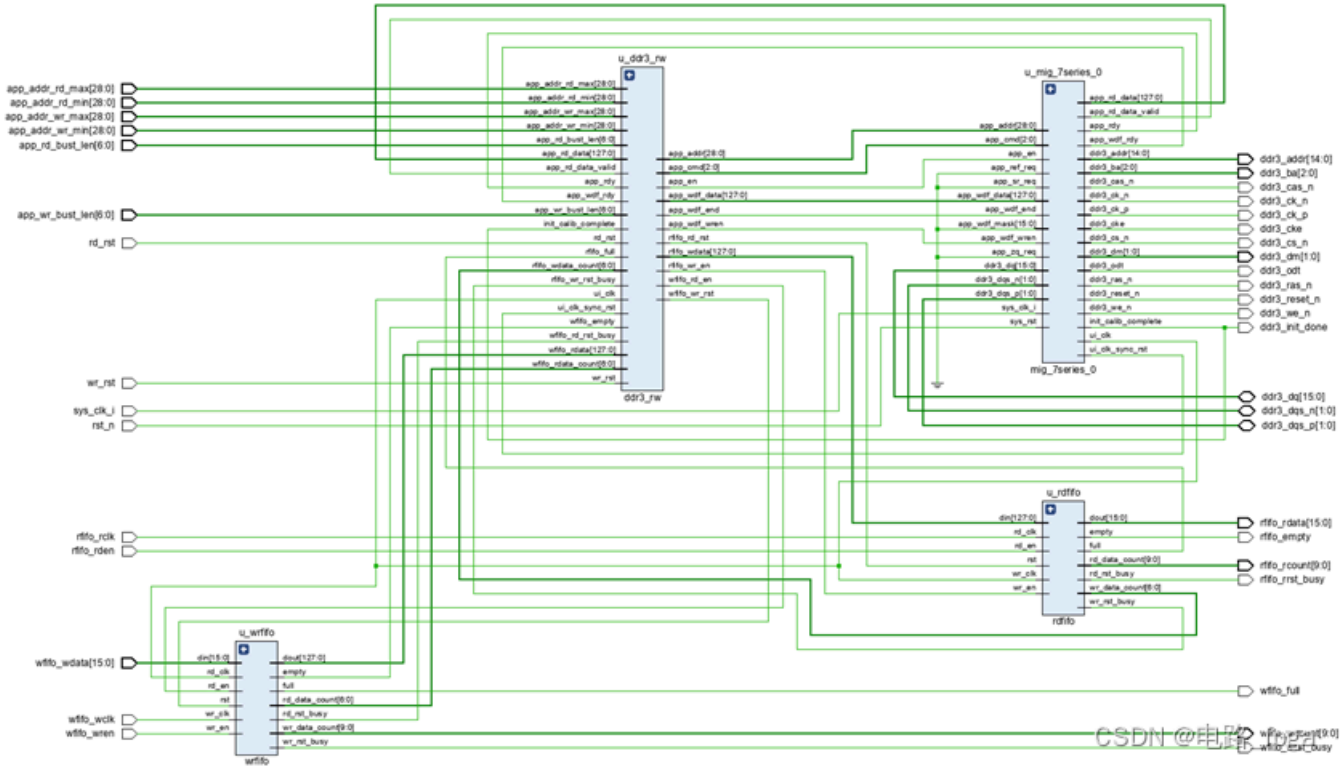

图7 顶层模块的RTL视图

顶层模块参考代码如下所示：

```
    //例化写FIFO IP；
    wrfifo u_wrfifo (
        .rst            ( wfifo_wr_rst      ),//input wire rst;
        .wr_clk         ( wfifo_wclk        ),//input wire wr_clk;
        .rd_clk         ( ui_clk            ),//input wire rd_clk;
        .din            ( wfifo_wdata       ),//input wire [15 : 0] din;
        .wr_en          ( wfifo_wren        ),//input wire wr_en;
        .rd_en          ( wfifo_rd_en       ),//input wire rd_en;
        .dout           ( wfifo_rdata       ),//output wire [127 : 0] dout;
        .full           ( wfifo_full        ),//output wire full;
        .empty          ( wfifo_empty       ),//output wire empty;
        .rd_data_count  ( wfifo_rcount      ),//output wire [6 : 0] rd_data_count;
        .wr_data_count  ( wfifo_wcount      ),//output wire [9 : 0] wr_data_count;
        .wr_rst_busy    ( wfifo_wrst_busy   ),//output wire wr_rst_busy;
```

```
        .rd_rst_busy    ( wfifo_rd_rst_busy ) //output wire rd_rst_busy;
    );

    //
    rdfifo u_rdfifo (
        .rst            ( rfifo_rd_rst      ),//input wire rst;
        .wr_clk         ( ui_clk            ),//input wire wr_clk;
        .rd_clk         ( rfifo_rclk        ),//input wire rd_clk;
        .din            ( rfifo_wdata       ),//input wire [127 : 0] din;
        .wr_en          ( rfifo_wr_en       ),//input wire wr_en;
        .rd_en          ( rfifo_rden        ),//input wire rd_en;
        .dout           ( rfifo_rdata       ),//output wire [15 : 0] dout;
        .full           ( rfifo_full        ),//output wire full;
        .empty          ( rfifo_empty       ),//output wire empty;
        .rd_data_count  ( rfifo_rcount      ),//output wire [9 : 0] rd_data_count;
        .wr_data_count  ( rfifo_wcount      ),//output wire [6 : 0] wr_data_count;
        .wr_rst_busy    ( rfifo_wrst_busy   ),//output wire wr_rst_busy;
        .rd_rst_busy    ( rfifo_rrst_busy   ) //output wire rd_rst_busy;
    );

    //例化DDR3读写控制模块;
    ddr3_rw #(
        .PINGPANG_EN            ( PINGPANG_EN       ),//乒乓操作是否使能;
        .USE_ADDR_W             ( USE_ADDR_W        ),//用户需要写入数据的位宽;
        .USE_BUST_LEN_W         ( USE_BUST_LEN_W    ),//用户侧读写数据突发长度的位宽;
        .USE_DATA_W             ( USE_DATA_W        ),//用户侧读写数据的位宽;
        .DDR_ADDR_W             ( DDR_ADDR_W        ),//MIG IP读写数据地址位宽;
        .DDR_DATA_W             ( DDR_DATA_W        ) //MIG IP读写数据的位宽;
    )
    u_ddr3_rw (
        //MIG IP用户侧相关信号;
        .ui_clk                 ( ui_clk            ),//;
        .ui_clk_sync_rst        ( ui_clk_sync_rst   ),//;
        .init_calib_complete    ( ddr3_init_done    ),//;
        .app_rdy                ( app_rdy           ),
        .app_wdf_rdy            ( app_wdf_rdy       ),
        .app_rd_data            ( app_rd_data       ),
        .app_rd_data_valid      ( app_rd_data_valid ),
        .app_en                 ( app_en            ),
        .app_cmd                ( app_cmd           ),

```

```
        .app_addr               ( app_addr          ),
        .app_wdf_wren           ( app_wdf_wren      ),
        .app_wdf_end            ( app_wdf_end       ),
        .app_wdf_data           ( app_wdf_data      ),
        //用户设置接口
        .app_addr_wr_min        ( app_addr_wr_min   ),
        .app_addr_wr_max        ( app_addr_wr_max   ),
        .app_wr_bust_len        ( app_wr_bust_len   ),
        .app_addr_rd_min        ( app_addr_rd_min   ),
        .app_addr_rd_max        ( app_addr_rd_max   ),
        .app_rd_bust_len        ( app_rd_bust_len   ),
        .wr_rst                 ( wr_rst            ),
        .rd_rst                 ( rd_rst            ),
        //写FIFO读侧信号
        .wfifo_wr_rst           ( wfifo_wr_rst      ),
        .wfifo_empty            ( wfifo_empty       ),
        .wfifo_rd_rst_busy      ( wfifo_rd_rst_busy ),
        .wfifo_rd_en            ( wfifo_rd_en       ),
        .wfifo_rdata            ( wfifo_rdata       ),
        .wfifo_rdata_count      ( wfifo_rcount      ),
        //读FIFO写侧信号
        .rfifo_rd_rst           ( rfifo_rd_rst      ),
        .rfifo_full             ( rfifo_full        ),
        .rfifo_wr_rst_busy      ( rfifo_wrst_busy   ),
        .rfifo_wdata_count      ( rfifo_wcount      ),
        .rfifo_wr_en            ( rfifo_wr_en       ),
        .rfifo_wdata            ( rfifo_wdata       )
    );

    //例化MIG IP
    mig_7series_0 u_mig_7series_0 (
        // Memory interface ports
        .ddr3_addr              ( ddr3_addr         ),//output [14:0]  ddr3_addr
        .ddr3_ba                ( ddr3_ba           ),//output [2:0]   ddr3_ba
        .ddr3_cas_n             ( ddr3_cas_n        ),//output         ddr3_cas_n
        .ddr3_ck_n              ( ddr3_ck_n         ),//output [0:0]   ddr3_ck_n
        .ddr3_ck_p              ( ddr3_ck_p         ),//output [0:0]   ddr3_ck_p
        .ddr3_cke               ( ddr3_cke          ),//output [0:0]   ddr3_cke
        .ddr3_ras_n             ( ddr3_ras_n        ),//output         ddr3_ras_n
        .ddr3_reset_n           ( ddr3_reset_n      ),//output         ddr3_reset_n

```

```
        .ddr3_we_n              ( ddr3_we_n         ),//output         ddr3_we_n
        .ddr3_dq                ( ddr3_dq           ),//inout [15:0]   ddr3_dq
        .ddr3_dqs_n             ( ddr3_dqs_n        ),//inout [1:0]    ddr3_dqs_n
        .ddr3_dqs_p             ( ddr3_dqs_p        ),//inout [1:0]    ddr3_dqs_p
        .init_calib_complete    ( ddr3_init_done    ),//output         init_calib_complete
        .ddr3_cs_n              ( ddr3_cs_n         ),//output [0:0]   ddr3_cs_n
        .ddr3_dm                ( ddr3_dm           ),//output [1:0]   ddr3_dm
        .ddr3_odt               ( ddr3_odt          ),//output [0:0]   ddr3_odt
        // Application interface ports
        .app_addr               ( app_addr          ),//input [28:0]   app_addr
        .app_cmd                ( app_cmd           ),//input [2:0]    app_cmd
        .app_en                 ( app_en            ),//input          app_en
        .app_wdf_data           ( app_wdf_data      ),//input [127:0]  app_wdf_data
        .app_wdf_end            ( app_wdf_end       ),//input          app_wdf_end
        .app_wdf_wren           ( app_wdf_wren      ),//input          app_wdf_wren
        .app_rd_data            ( app_rd_data       ),//output [127:0] app_rd_data
        .app_rd_data_end        (                   ),//output         app_rd_data_end
        .app_rd_data_valid      ( app_rd_data_valid ),//output         app_rd_data_valid
        .app_rdy                ( app_rdy           ),//output         app_rdy
        .app_wdf_rdy            ( app_wdf_rdy       ),//output         app_wdf_rdy
        .app_sr_req             ( 0                 ),//input          app_sr_req
        .app_ref_req            ( 0                 ),//input          app_ref_req
        .app_zq_req             ( 0                 ),//input          app_zq_req
        .app_sr_active          (                   ),//output         app_sr_active
        .app_ref_ack            (                   ),//output         app_ref_ack
        .app_zq_ack             (                   ),//output         app_zq_ack
        .ui_clk                 ( ui_clk            ),//output         ui_clk
        .ui_clk_sync_rst        ( ui_clk_sync_rst   ),//output         ui_clk_sync_rst
        .app_wdf_mask           ( 16'd0             ),//input [15:0]   app_wdf_mask
        // System Clock Ports
        .sys_clk_i              ( sys_clk_i         ),//系统输入200MHz时钟作为MIG参考和工作时钟;
        .sys_rst                ( rst_n             ) //input sys_rst
    );
```

5、仿真

由于该模块的仿真需要DDR3芯片的仿真模型，想要模拟DDR3芯片工作模式对于我们来说还是十分困难的。但是前文对MIG IP仿真时，官方提供了DDR3的仿真模型，可以从该工程中获取仿真模型，添加到本工程，然后仿真。

打开前文生成的官方测试例程的imports文件夹，如下图所示，将ddr3_model.sv和ddr3_model_parameters.vh文件复制，然后加入自己工程中。

ddr3_model.sv就是官方使用System Verilog编写的DDR3仿真模型，而ddr3_model_parameters.vh存放的是ddr3_model.sv需要的一些parameter常量。

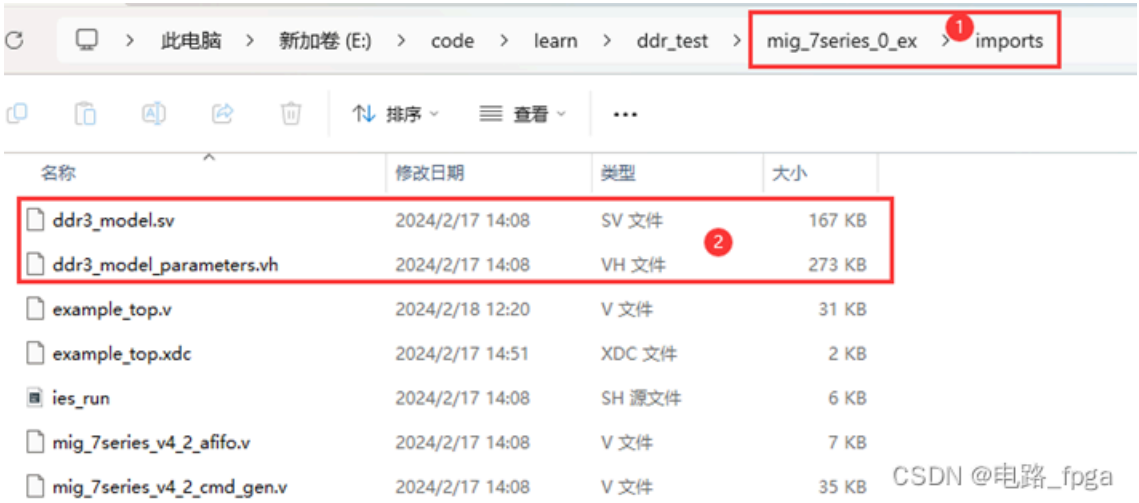

图8 从官方工程提取仿真模型

加入工程后，如下图所示。

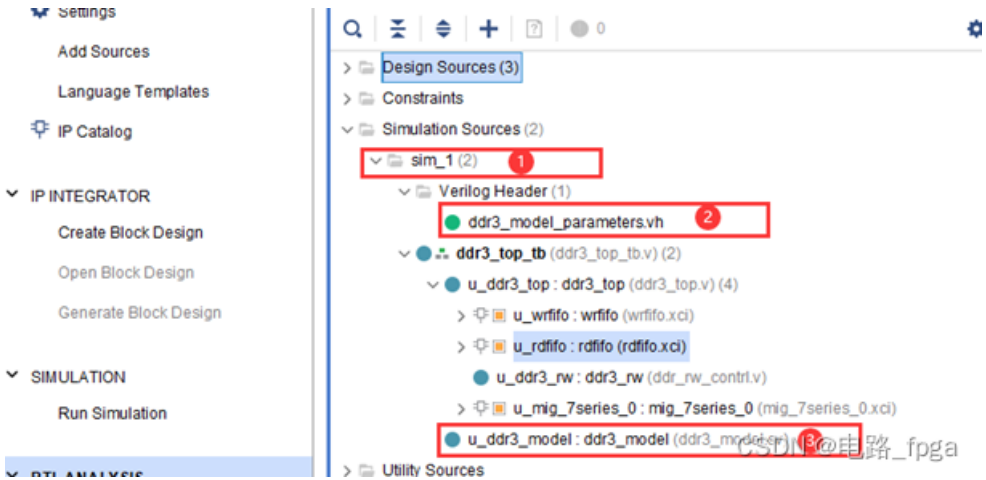

图9 仿真文件

仿真的测试文件很简单，先向DDR3中0~4095地址空间写入数据，每次突发写入512个数据。之后在把写入的数据依次读出，对应的TestBench代码如下所示：

```
`timescale 1ns/1ns
module ddr3_top_tb;
    localparam      DDR3_CYCLE      =   5           ;//DDR3时钟周期；
    localparam      WFIFO_CYCLE     =   8           ;//写FIFO的写时钟周期；
    localparam      RFIFO_CYCLE     =   20          ;//读FIFO读时钟周期；
    localparam      RST_TIME        =   10          ;//系统复位持续时间，默认10个系统时钟周期；
    localparam      STOP_TIME       =   1000        ;//仿真运行时间，复位完成后运行1000个系统时钟后停止；

    reg                                 sys_clk_i   ;
    reg                                 rst_n       ;
    reg                                 wr_rst      ;
    reg                                 rd_rst      ;
    reg                                 wrfifo_clk  ;
    reg                                 wfifo_wren  ;
    reg     [15 : 0]                    wfifo_wdata ;
    reg                                 rdfifo_clk  ;
    reg                                 rfifo_rden  ;

    wire  [14 : 0]                      ddr3_addr   ;
    wire  [2 : 0]                       ddr3_ba     ;
    wire                                ddr3_ras_n  ;
    wire                                ddr3_cas_n  ;
    wire                                ddr3_we_n   ;
    wire                                ddr3_reset_n;
    wire                                ddr3_ck_p   ;
    wire                                ddr3_ck_n   ;
    wire                                ddr3_cke    ;
    wire                                ddr3_cs_n   ;
    wire  [1 : 0]                       ddr3_dm     ;
    wire                                ddr3_odt    ;
    wire  [9 : 0]                       wfifo_wcount;
    wire                                wfifo_full  ;
    wire                                wfifo_wrst_busy ;
    wire  [15 : 0]                      rfifo_rdata ;
    wire  [9 : 0]                       rfifo_rcount;
    wire                                rfifo_empty ;

```

```
wire                                rfifo_rrst_busy ;
wire                                ddr3_init_done  ;
wire  [15 : 0]                      ddr3_dq     ;
wire  [1 : 0]                       ddr3_dqs_n  ;
wire  [1 : 0]                       ddr3_dqs_p  ;

//生成DDR3参考时钟信号200MHz。
initial begin
    sys_clk_i = 0;
    forever #(DDR3_CYCLE/2) sys_clk_i = ~sys_clk_i;
end

//生成写FIFO写时钟信号;
initial begin
    wrfifo_clk = 0;
    forever #(WFIFO_CYCLE/2) wrfifo_clk = ~wrfifo_clk;
end

//生成读FIFO读时钟信号;
initial begin
    rdfifo_clk = 0;
    forever #(RFIFO_CYCLE/2) rdfifo_clk = ~rdfifo_clk;
end

initial begin
    rst_n = 1'b0;
    wfifo_wren = 1'b0;
    wfifo_wdata  = 16'd0;
    rfifo_rden = 1'b0;
    wr_rst  = 1'b0;
    rd_rst  = 1'b0;
    #201;
    rst_n = 1'b1;
    @(posedge ddr3_init_done);
    wr_rst  = 1'b1;
    repeat(4)@(posedge wrfifo_clk);
    wr_rst  = 1'b0;
    repeat(50)@(posedge wrfifo_clk);
    repeat(4)begin
        wr_data(16'd100,16'd1024);

```

```
            #2000;
        end
        rd_rst  = 1'b1;
        repeat(4)@(posedge rdfifo_clk);
        rd_rst  = 1'b0;
        repeat(50)@(posedge rdfifo_clk);
        repeat(4)begin
            rd_data(16'd1024);
            #2000;
        end
        #1000;
        $stop;
    end

    task wr_data;
        input [15:0] data_begin;
        input [15:0] wr_data_cnt;
        begin
            wfifo_wren <= 1'b0;
            wfifo_wdata  <= data_begin;
            @(posedge wrfifo_clk);
            wfifo_wren <= 1'b1;
            repeat(wr_data_cnt)begin
                @(posedge wrfifo_clk);
                wfifo_wdata <= wfifo_wdata + 1'b1;
            end
            wfifo_wren <= 1'b0;
        end
    endtask

    task rd_data;
        input [15:0] rd_data_cnt;
        begin
            rfifo_rden <= 1'b0;
            @(posedge rdfifo_clk);
            rfifo_rden <= 1'b1;
            repeat(rd_data_cnt)begin
                @(posedge rdfifo_clk);
            end
            rfifo_rden <= 1'b0;
```

```
    end
endtask

//例化DDR3顶层模块;
ddr3_top #(
    .PINGPANG_EN    ( 1'b0 ),
    .USE_ADDR_W     ( 29   ),
    .USE_BUST_LEN_W ( 7    ),
    .USE_DATA_W     ( 16   ),
    .DDR_ADDR_W     ( 29   ),
    .DDR_DATA_W     ( 128  )
)
u_ddr3_top (
    .sys_clk_i          ( sys_clk_i         ),
    .rst_n              ( rst_n             ),
    .app_addr_wr_min    ( 0                 ),
    .app_addr_wr_max    ( 4096              ),
    .app_wr_bust_len    ( 64                ),
    .app_addr_rd_min    ( 0                 ),
    .app_addr_rd_max    ( 4096              ),
    .app_rd_bust_len    ( 64                ),
    .wr_rst             ( wr_rst            ),
    .rd_rst             ( rd_rst            ),
    //写FIFO相关信号
    .wfifo_wclk         ( wrfifo_clk        ),
    .wfifo_wren         ( wfifo_wren        ),
    .wfifo_wdata        ( wfifo_wdata       ),
    .wfifo_wcount       ( wfifo_wcount      ),
    .wfifo_full         ( wfifo_full        ),
    .wfifo_wrst_busy    ( wfifo_wrst_busy   ),
    //读FIFO相关信号
    .rfifo_rclk         ( rdfifo_clk        ),
    .rfifo_rden         ( rfifo_rden        ),
    .rfifo_rdata        ( rfifo_rdata       ),
    .rfifo_rcount       ( rfifo_rcount      ),
    .rfifo_empty        ( rfifo_empty       ),
    .rfifo_rrst_busy    ( rfifo_rrst_busy   ),
    //DDR3端口
    .ddr3_addr          ( ddr3_addr         ),
    .ddr3_ba            ( ddr3_ba           ),
```

```
        .ddr3_ras_n         ( ddr3_ras_n        ),
        .ddr3_cas_n         ( ddr3_cas_n        ),
        .ddr3_we_n          ( ddr3_we_n         ),
        .ddr3_reset_n       ( ddr3_reset_n      ),
        .ddr3_ck_p          ( ddr3_ck_p         ),
        .ddr3_ck_n          ( ddr3_ck_n         ),
        .ddr3_cke           ( ddr3_cke          ),
        .ddr3_cs_n          ( ddr3_cs_n         ),
        .ddr3_dm            ( ddr3_dm           ),
        .ddr3_odt           ( ddr3_odt          ),
        .ddr3_init_done     ( ddr3_init_done    ),
        .ddr3_dq            ( ddr3_dq           ),
        .ddr3_dqs_n         ( ddr3_dqs_n        ),
        .ddr3_dqs_p         ( ddr3_dqs_p        )
    );

    //例化DDR3仿真模型;
    ddr3_model u_ddr3_model (
        .rst_n   ( ddr3_reset_n ),
        .ck      ( ddr3_ck_p    ),
        .ck_n    ( ddr3_ck_n    ),
        .cke     ( ddr3_cke     ),
        .cs_n    ( ddr3_cs_n    ),
        .ras_n   ( ddr3_ras_n   ),
        .cas_n   ( ddr3_cas_n   ),
        .we_n    ( ddr3_we_n    ),
        .dm_tdqs ( ddr3_dm      ),
        .ba      ( ddr3_ba      ),
        .addr    ( ddr3_addr    ),
        .dq      ( ddr3_dq      ),
        .dqs     ( ddr3_dqs_p   ),
        .dqs_n   ( ddr3_dqs_n   ),
        .tdqs_n  (              ),
        .odt     ( ddr3_odt     )
    );

endmodule
```

然后直接仿真即可，只不过DDR3因为需要初始化，且初始化流程比较复杂，所以需要等待较长时间。仿真结果如下所示。

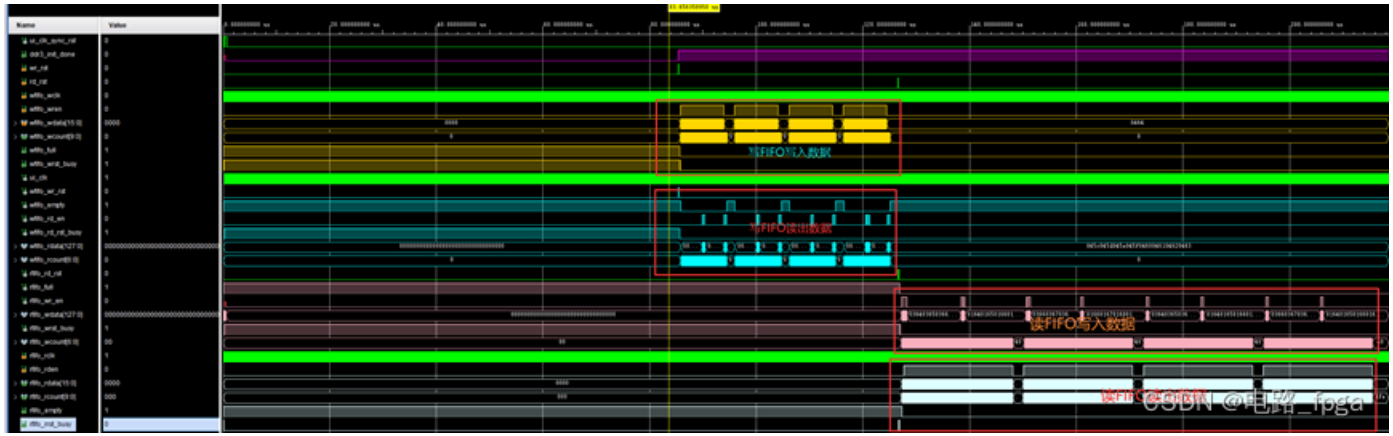

图10 仿真结果

如下图所示，TestBench设置一次读、写突发的长度对于MIG IP的128位数据来说是64，当写FIFO中的数据大于等于64-2时，从写FIFO中读出数据存入DDR3中，如下图所示。

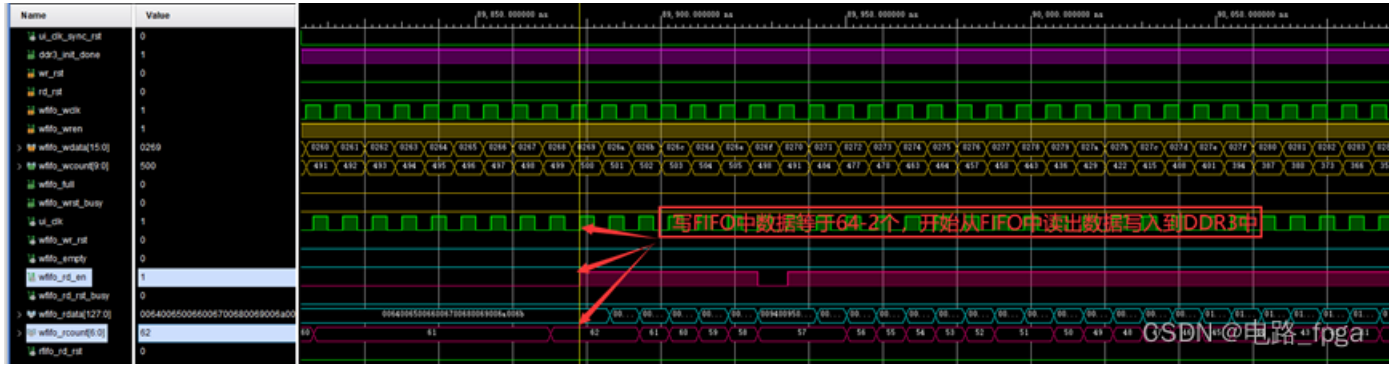

图11 读出写FIFO中的数据

由于内部设置，只有当DDR3的起止地址之间全部被写入一次数据之后，如果读FIFO中的数据少于一次突发读传输的数据（此处为64个128位数据），且读FIFO不处于复位状态，则从DDR3中读取数据存入读FIFO中，如下图所示。

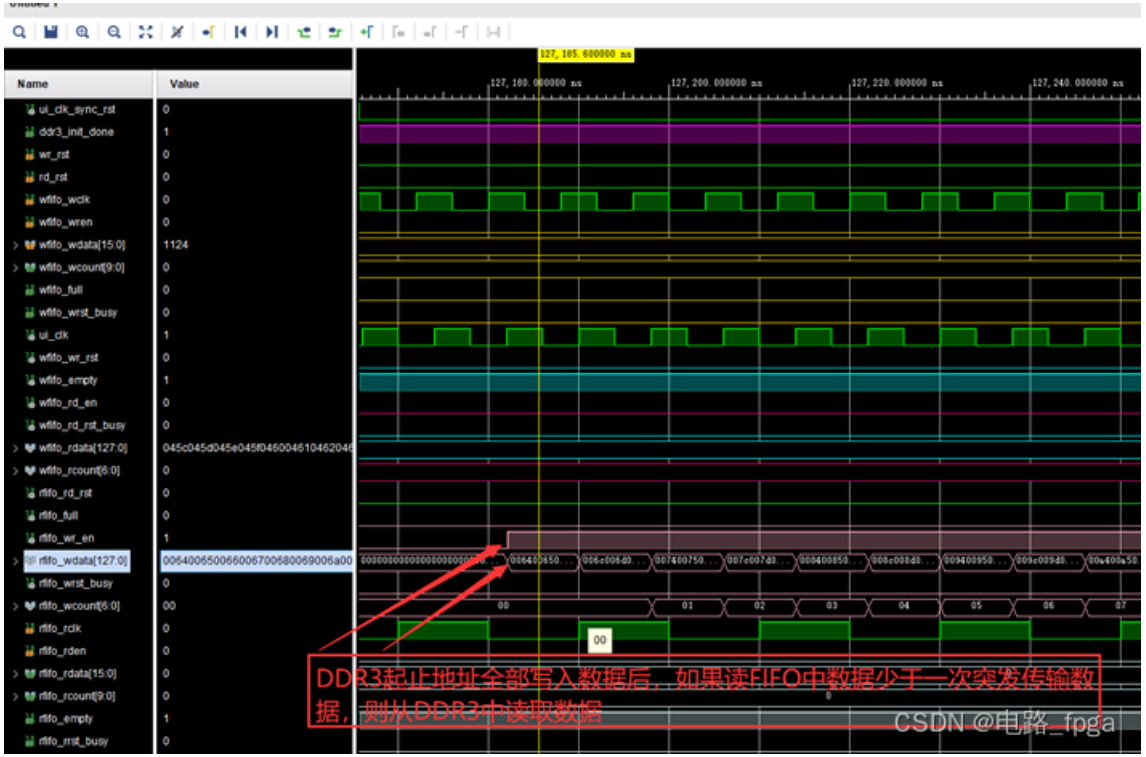

图12 从DDR3读取数据存入FIFO

之后用户就可以通过拉高读FIFO的读使能信号，从读FIFO中读出数据了，如下图所示。

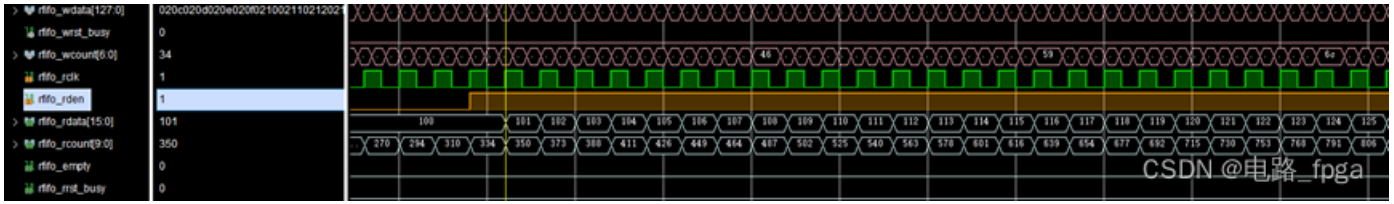

图13 用户读取数据

上图表示读出的数据是从100开始累加的，下图是用户写入DDR3中的数据，也是从100开始累加的，所以写入和读出的开始数据正确。

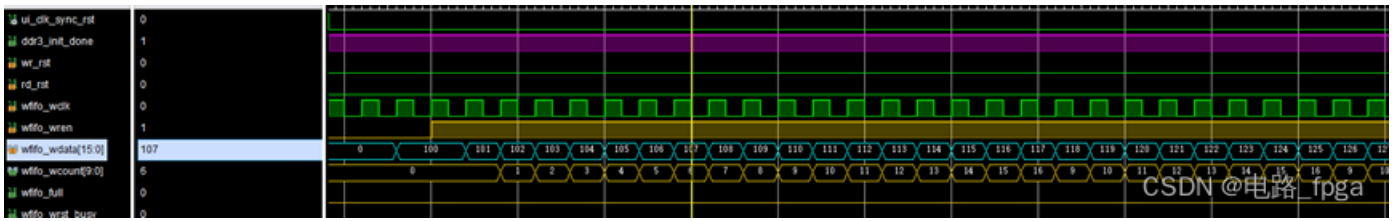

图14 用户写入的数据

第一帧写入的结束数据位1123，如下图所示。

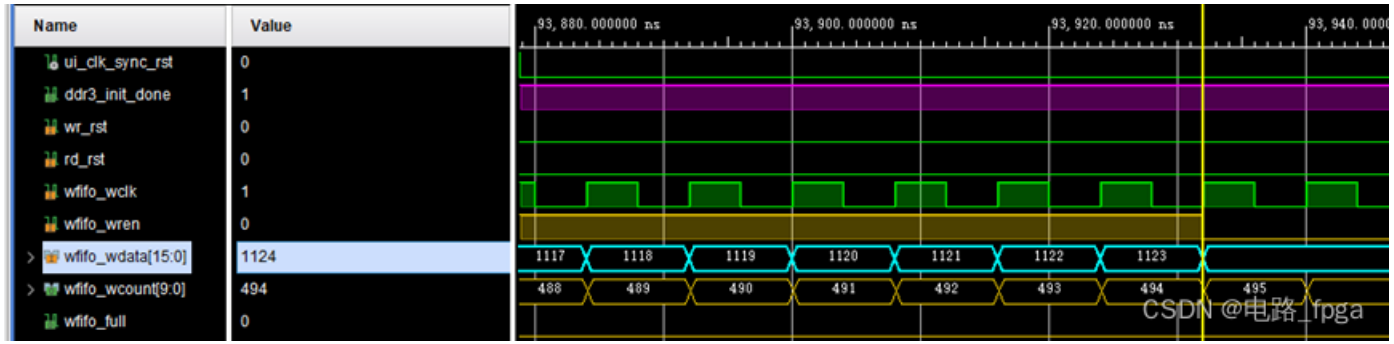

图15 写入数据

第一帧读出的数据如下图所示，也是1123，读写数据一致，证明整个写入和读出的控制逻辑没有问题。

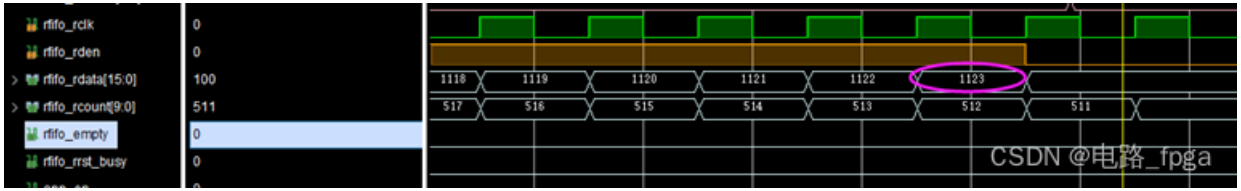

图16 读出数据

最后要注意一下xilinx的FIFO仿真，发现这个FIFO必须在复位之后才能进行读、写，不然无法写入数据。

以写FIFO举例，如下图所示，仿真开始后，写侧和读侧的复位指示信号一直位高电平，表示FIFO处于复位状态。此时将FIFO的复位信号拉高一段时间，之后将复位信号拉低，过一段时间后，写侧和读侧的复位指示信号才会拉低，此时才能向FIFO中写入数据，否则无法写入数据。

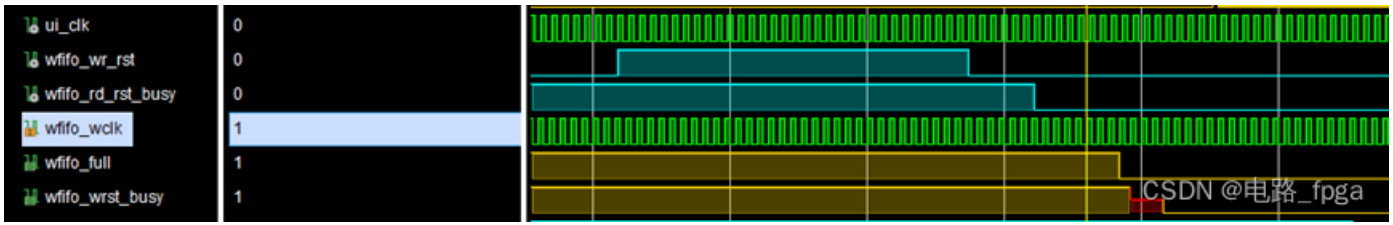

图17 FIFO复位信号

注意上图中复位信号拉低之后，满指示信号也会持续一段时间才会无效，但是持续的时间并没有复位指示信号长，所以依旧可以根据复位指示信号来判断复位是否结束。

这种现象只存在于仿真，实际上板时经过实测，即使不复位FIFO，也能写入数据，并且读出正确数据，可能这是Xilinx为了让我们养成复位的习惯在仿真时添加的限制吧，哈哈。

对于该模块的仿真就结束了，不在对状态机的跳转进行分析，因为逻辑比较简单。另外乒乓操作也没仿真，如果觉得有必要可以自己写TestBench进行仿真，乒乓操作就是地址的切换，其余逻辑均一致，所以不会出现问题，后续直接使用即可。

本工程不用于上板，只是进行仿真，后续使用该模块存储数据时，在上板观察使用结果。

如果需要本次工程，在公众号后台回复“**MIG IP封装成FIFO**”（不包括引号）即可。

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！