1 千兆以太网传图接收 TFT 显示系统设计

工程源码	|-ACZ702 开发板 |-- acz702_eth_udp_rgmii_ddr3_tft
相关视频课程	无
说明	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本节实验通过以太网传图工具将图像数据传输到 ACZ702 开发板上 PS 端的 DDR3 进行存储，最终图像数据由 TFT 控制器主动从 DDR3 中读取出来，送往 TFT 显示屏进行显示。

1.1 系统整体设计

系统的整体设计如下所示：

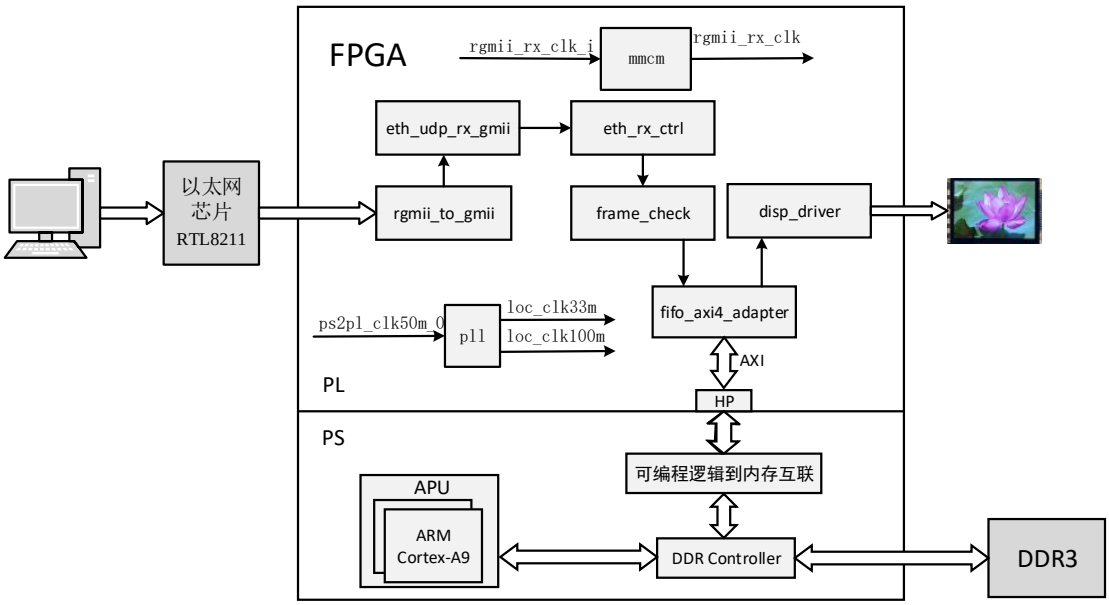

图 1-1 千兆以太网传图 TFT 显示系统整体设计框图

本章实验仅针对于上述图中的 PL 部分的代码设计进行讲解，PS 部分的内容请参看：

【ACZ702】ZYNQ PL 读写 PS DDR3 双端口读写控制器设计

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29312

PL 部分的模块说明如下：

1. pll 模块：锁相环模块，生成本次实验每个模块所需要的工作时钟，输

入时钟 50M，由 PS 输出给 PL；输出 100M 的时钟给到 DDR3 控制器使用；输出 33M 的时钟给到 TFT 控制器（disp_driver）使用。

2. mmcm 模块：锁相环模块，将 rgmii 接口时钟信号 rgmii_rx_clk_i 偏移 90°得到 rgmii_rx_clk 时钟信号，这样做是为了在时钟信号的上升沿/下降沿取数据时，取得数据刚好是数据信号 rgmii_rxd 的正中间，使得采样的数据处于最稳定的状态，如下图 1-2 所示。

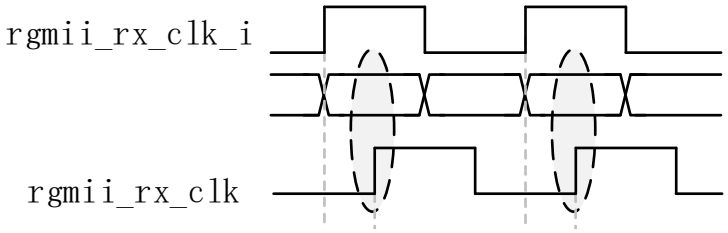

图 1-2 采集数据波形图

3. rgmii_to_gmii 模块：以太网接口转换模块，将 rgmii 接口转换成 gmii 接口。
4. eth_udp_rx_gmii 模块：以太网接收模块，接收上位机传输过来的图像数据，详细设计请参看 702 逻辑教程以太网相关章节的内容。
5. eth_rx_ctrl 模块：以太网接收控制模块，将以太网接收的报文数据存入 FIFO 中。
6. frame_check 模块：判断每个包数据的序号是否连续，如果不连续将整帧图像数据丢弃。
7. disp_driver 模块：产生符合 TFT 显示屏的驱动时序，并从 DDR3 控制器中取出图像数据送到 TFT 显示屏，详细设计请 702 逻辑教程参看“TFT 显示屏驱动设计与验证”一节的内容
8. fifo_axi4_adapter 模块：fifo 接口到 AXI4 接口的转换模块(含 2 个 FIFO)，详细设计请参看前面 DDR 相关章节的内容。

本章需要设计的模块包括eth_rx_ctrl 模块和 frame_check 模块。

1.2 模块设计

1.2.1 以太网接收控制模块

本节将介绍以太网控制模块的代码设计及其仿真分析。

1.2.1.1 太网控制模块设计

以太网接收控制模块，主要是将以太网接收的报文数据存入 FIFO 中，其模块的结构框图如下图 1-3 所示。从图中可以看出输入数据为 8 位，输出数据为 16 位，这样做的目的是因为以太网接收模块输出数据是 8 位的，而传输过来的图像数据是 16 位的，所以这里需要将两个连续的 8 位数据还原成 16 位的图像数据。

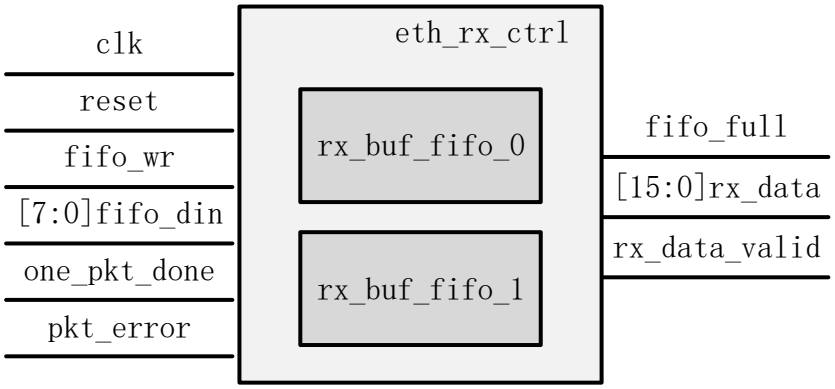

图 1-3 以太网接收控制模块

对于每个信号说明如下表 1-1 所示。

表 1-1 以太网接收控制模块信号说明表

接口名称	I/O	信号意义
clk	I	模块工作时钟信号
reset	I	模块复位信号，高有效
fifo_wr	I	FIFO 的写使能信号
fifo_din[7:0]	I	需要向 FIFO 中写入的 8 位数据
one_pkt_done	I	以太网单包数据接收完成标志信号
pkt_error	I	以太网包接收错误标志信号
fifo_full	I	FIFO 写满标志信号
rx_data[15:0]	O	从 FIFO 读出的 16 位的数据信号
rx_data_valid	O	FIFO 读出数据有效信号

在该模块中，使用了两个 FIFO IP 模块存储以太网接收过来的数据。这里使用两个 FIFO 实现双缓存的目的是在 CRC 校验出现问题时把这包数据丢掉。以太网在传输数据的过程中只有一包的数据接收完成之后才能判断数据是否出现错误，出现问题之后再清空 FIFO 要保证不能影响下一包数据的接收，当只有一个 FIFO 进行缓存时，有可能存在当前一包数据接收完正在判断是否丢弃过程中，下一包数据正在往 FIFO 里写，如果判断当前包有问题清空过程中会把下一包部分数据清掉，这样下一包数据就不完整了，而双缓存可以避免这个问题。下面

将介绍该模块中部分代码的实现。

FIFO 的写使能信号的产生如下所示：

```
assign fifo0_wr_en = fifo_wr && (wrfifo_sel == 1'b0);
assign fifo1_wr_en = fifo_wr && (wrfifo_sel == 1'b1);

always@(posedge clk or posedge reset)
begin
  if(reset)
    wrfifo_sel <= 1'b0;
  else if(one_pkt_done)
    wrfifo_sel <= ~wrfifo_sel;
  else
    wrfifo_sel <= wrfifo_sel;
end
```

上述代码中，可以看出当 wrfifo_sel 为 0 并且 fifo_wr 有效时，向 FIFO0 中写入数据，当 wrfifo_sel 为 1 并且 fifo_wr 有效时，向 FIFO1 中写入数据。当产生以太网包传输完成标志信号 one_pkt_done 之后，对 wrfifo_sel 信号进行取反，也就是一个包传输完成之后，就切换一个 FIFO 进行数据的存储。本次实验中的 fifo_wr 信号连接的是以太网接收模块的 payload_valid_o 信号，也就是当以太网开始接收数据之后，fifo_wr 就有效。

FIFO 的清除代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo0_srst <= 1'b1;
  else if(one_pkt_done && pkt_error && (wrfifo_sel == 1'b0))
    fifo0_srst <= 1'b1;
  else
    fifo0_srst <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo1_srst <= 1'b1;
  else if(one_pkt_done && pkt_error && (wrfifo_sel == 1'b1))
    fifo1_srst <= 1'b1;
  else
    fifo1_srst <= 1'b0;
end
```

从上述代码中可以看出，当产生一个包接收完成标志信号 one_pkt_done、接收数据出现错误标志信号 pkt_error，wrfifo_sel 信号为 0 的时候，fifo0_srst 为

1，清除 FIFO0； wrfifo_sel 信号为 1 的时候，fifo1_srst 为 1，清除 FIFO1。只有数据没有出现错误，对应的 FIFO 复位信号 fifo0_srst 和 fifo1_srst 为 0，保留 FIFO 中的数据。

FIFO 的读使能信号如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo0_rd_en <= 1'b0;
  else if((fifo0_rd_data_cnt > 1'b1) && (rdfifo_sel == 1'b0))
    fifo0_rd_en <= 1'b1;
  else
    fifo0_rd_en <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    fifo1_rd_en <= 1'b0;
  else if((fifo1_rd_data_cnt > 1'b1) && (rdfifo_sel == 1'b1))
    fifo1_rd_en <= 1'b1;
  else
    fifo1_rd_en <= 1'b0;
end
```

当对应的 FIFO 中有数据时，也就是 fifo0_rd_data_cnt > 1'b1 或者 fifo1_rd_data_cnt > 1'b1，代表 FIFO 中的数据是有效的，此时若 rdfifo_sel 为 0 则读取 FIFO0 中的数据，若 rdfifo_sel 为 1 则读取 FIFO1 中的数据。

产生对应的 FIFO 的读使能之后，则将对应的 FIFO 中的数据读取出来，并产生数据有效信号 rx_data_valid，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    rx_data <= 'd0;
  else if(fifo0_rd_en_dly1)
    rx_data <= fifo0_dout;
  else if(fifo1_rd_en_dly1)
    rx_data <= fifo1_dout;
  else
    rx_data <= 'd0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
```

```
    rx_data_valid <= 1'b0;
  else if(fifo0_rd_en_dly1)
    rx_data_valid <= 1'b1;
  else if(fifo1_rd_en_dly1)
    rx_data_valid <= 1'b1;
  else
    rx_data_valid <= 1'b0;
end
```

1.2.1.2 以太网控制模块仿真

以太网控制模块设计完成之后，需要通过仿真对其进行测试，通过仿真波形，查看是否实现功能：每个包传输过来的时候，会交替写入到两个 FIFO 中，当数据是正确的时，将数据输出，数据出现错误时，将数据清除不输出。如果满足该功能，则认为仿真正确。

首先在 tb 文件中编写两个 task 事件，分别代表写入的是正确的数据和错误的数据，写入错误数据时设置 pkt_error 为高电平，最终通过仿真查看最终输出的数据是否为正确数据，两个 task 事件代码如下所示：

```
task write_data;
    input [7:0]data_begin;
    input [10:0]data_cnt;
    begin
    fifo_wr = 0;
    fifo_din = data_begin;
    one_pkt_done = 0;
    pkt_error = 0;

    @(posedge clk);
    #1 fifo_wr = 1;
    repeat(data_cnt) begin
        @(posedge clk);
        #1 fifo_din = fifo_din + 1;
    end
    fifo_wr = 0;
    @(posedge clk);
    #1 one_pkt_done = 1;
    @(posedge clk);
    #1 one_pkt_done = 0;
    end
endtask

task write_data_error;
    input [7:0]data_begin;
    input [10:0]data_cnt;
```

```
    begin
    fifo_wr = 0;
    fifo_din = data_begin;
    one_pkt_done = 0;
    pkt_error = 0;

    @(posedge clk);
    #1 fifo_wr = 1;
    repeat(data_cnt) begin
        @(posedge clk);
        #1 fifo_din = fifo_din + 1;
    end
    fifo_wr = 0;
    @(posedge clk);#1;
    one_pkt_done = 1;
    pkt_error = 1;
    @(posedge clk);#1;
    one_pkt_done = 0;
    pkt_error = 0;
    end
endtask
```

然后在 tb 文件中例化 eth_rx_ctrl 模块，并且写入正确或错误的数据，代码如下所示：

```
eth_rx_ctrl
#(
.O_DATA_WIDTH (16  )
)eth_rx_ctrl
(
.clk          (clk          ),
.reset        (~reset_n     ),

.fifo_full    (fifo_full    ),
.fifo_wr      (fifo_wr      ),
.fifo_din     (fifo_din     ),

.one_pkt_done (one_pkt_done ),
.pkt_error    (pkt_error    ),

.rx_data      (rx_data      ),
.rx_data_valid(rx_data_valid)
);

initial begin
reset_n = 1'b0;
fifo_wr = 0;
fifo_din = 0;
```

```
one_pkt_done = 0;
pkt_error = 0;
#201;
reset_n = 1'b1;
#200;

write_data(0,50);
#2000;
write_data(100,50);
#2000;
write_data_error(200,50);
#2000;
write_data(300,50);
#2000;
write_data_error(400,50);
#2000;
write_data(500,50);
#2000;
$stop;
end
```

完整的 tb 文件请自行查看例程（eth_udp_rgmii_ddr3_tft\eth_udp_rgmii_ddr3_tft.srcs\sources_1\new\tb），下面对仿真波形进行分析。

首先发送两个包的数据，每包数据为 50 个（并不是实际以太网包数据，这里是为了方便仿真），可以看到产生对应的写使能信号分别写入 FIFO0 和 FIFO1 中，波形图如下图 1-4 所示：

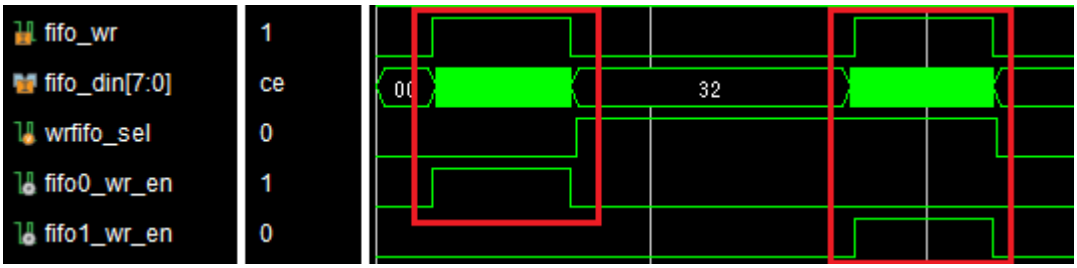

图 1-4 交互写入两个 FIFO 中

前两个包的数据是正确的，所以会产生对应的读使能信号，将得到的数据进行输出，并产生数据有效信号，波形图如下图 1-5 所示：

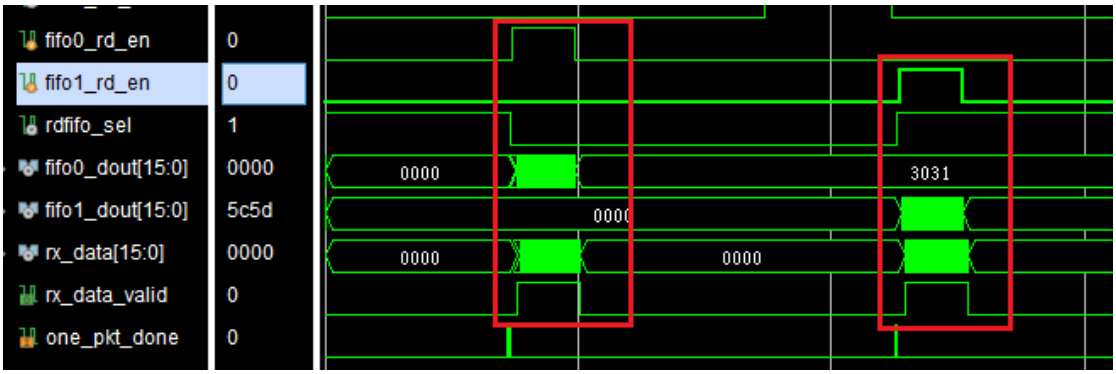

图 1-5 正确数据输出波形图

以第一包为例，写入的 0~49 一共 50 个数据，最终输出的波形图如下图 1-6 所示，可以看出写入和读出的数据一致。

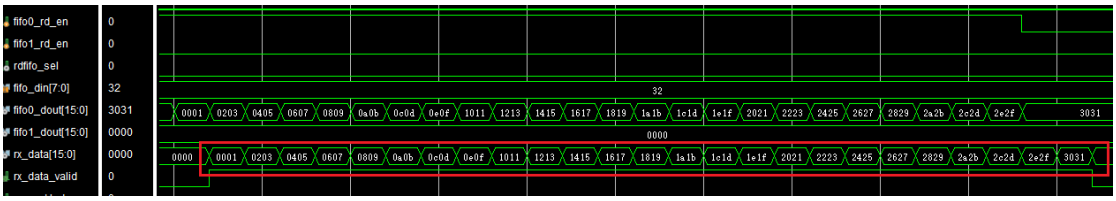

图 1-6 输出数据波形图

传输的第三个包为错误的包，此时应该向 FIFO0 写入，但是由于是错误的，写入之后，通过判断后会清除 FIFO0 中的数据（产生 fifo0_srst 信号），最终也就不会输出，波形图如下图 1-7 所示：

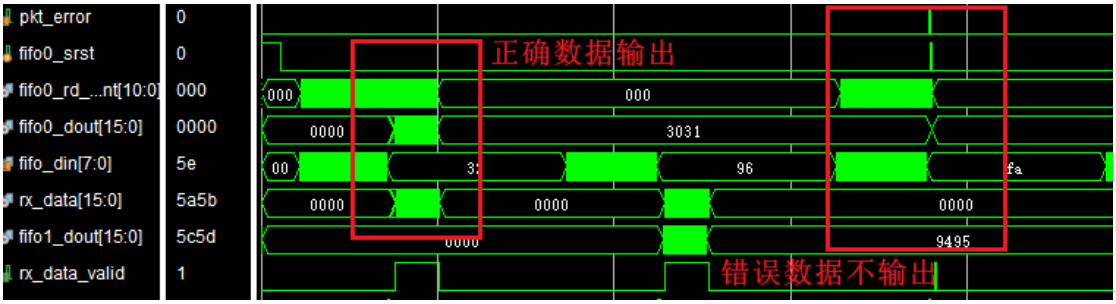

图 1-7 错误数据清除波形图

通过上述分析波形图可以看出，每个包传输过来的时候，会交替写入到两个 FIFO 中，当数据是正确的时，将数据输出，数据出现错误时，将数据清除不输出，符合设计预期的功能。

1.2.2 数据包检测模块

本节将介绍数据包检测模块的代码设计及其仿真分析。

1.2.2.1 数据包检测模块设计

数据包检测模块（frame_check 模块）判断每包数据的序号是否连续，如果不连续将整帧图像数据丢弃。其模块的结构框图如下图 1-8 所示：

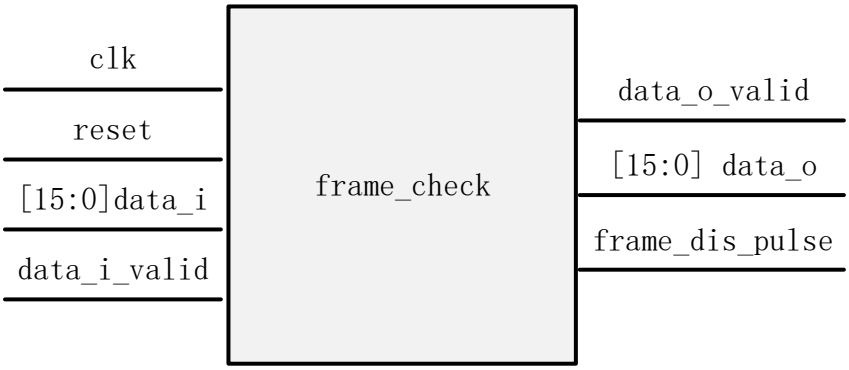

图 1-8 数据包检测模块结构框图

对每个信号的说明如下表 1-2 所示：

表 1-2 数据包检测模块信号说明表

信号名称	I/O	信号意义
clk	I	模块工作时钟信号
reset	I	模块复位信号，高有效
data_i[15:0]	I	从 FIFO 中读取出来的数据
data_i_valid	I	读取数据有效信号
data_o_valid	O	输出数据有效信号
data_o[15:0]	O	模块最终数据的数据
frame_dis_pulse	O	接收数据包错误标志脉冲信号

在我们所设计的以太网上位机软件中，每一个包传输的前两个字节代表的是需要传输图像的序号。通过检测每个包的数据的前两个字节的序号是否连续，来判断是否需要将数据丢弃。下面将介绍一下该模块部分代码实现。

首先，需要将传输图像数据的序号提取出来，通过检测到数据有效信号的上升沿来得到。将数据有效信号 data_i_valid “打两拍”得到 data_i_valid_dly1 信号和 data_i_valid_dly2 信号，当 data_i_valid_dly1 为高电平，data_i_valid_dly2 为低电平时，得到数据有效 data_i_valid_rising 脉冲信号，代码如下所示：

```
always@(posedge clk)
begin
  data_i_valid_dly1 <= data_i_valid;
  data_i_valid_dly2 <= data_i_valid_dly1;
  data_i_dly1 <= data_i;
end

assign data_i_valid_rising=data_i_valid_dly1 && (~data_i_valid_dly2);
```

当检测到 data_i_valid_rising 信号之后，将此时从 FIFO 中读取到的 data_i_dly1 的值提取出来，得到的数据就是图像数据的序号，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    data_row_num <= 'd0;
  else if(data_i_valid_rising)
    data_row_num <= data_i_dly1;
  else
    data_row_num <= 'd0;
end
```

其次，将每个包的序号提取出来之后，还得判断每个包序号是否连续。每次检测到 data_i_valid_rising 信号时，将 check_row_num 值加 1，然后判断 check_row_num 的值和 data_row_num 的值来判断序号是否一致。其中每个包的序号代表的就是所需要传输图像的行像素值，也就是 frame_check 模块的 IMAGE_HEIGHT 的值，所以 check_row_num 最大的值不超过 IMAGE_HEIGHT-1'b1。check_row_num 的变化代码如下：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    check_flag <= 1'b0;
  else if(data_i_valid_rising)
    check_flag <= 1'b1;
  else
    check_flag <= 1'b0;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    check_row_num <= 'd0;
  else if(check_flag == 1'b1)
  begin
    if(check_row_num == IMAGE_HEIGHT - 1'b1)
      check_row_num <= 'd0;
    else
      check_row_num <= check_row_num + 1'd1;
  end
  else if(data_i_valid_rising && (data_i_dly1 == 'd0))
    check_row_num <= 'd0;
  else
    check_row_num <= check_row_num;
end
```

当 check_row_num 的值和 data_row_num 的值不一致时，也就是说以太网采集得到的图像数据序号不一致，此时产生数据错误标志信号 frame_dis_flag，当重新开始一副图像开始传输时将 frame_dis_flag 信号拉低，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
    frame_dis_flag <= 1'b0;
  else if(data_i_valid_rising && (data_i_dly1 == 'd0))
    frame_dis_flag <= 1'b0;
  else if(check_flag && (check_row_num != data_row_num))
    frame_dis_flag <= 1'b1;
  else
    frame_dis_flag <= frame_dis_flag;
end
```

通过将 frame_dis_flag 信号“打两拍”得到 frame_dis_flag_dly1 和 frame_dis_flag_dly2 信号，当 frame_dis_flag 信号为高电平，frame_dis_flag_dly2 为低电平时，得到检测数据错误脉冲信号 frame_dis_pulse，最终将该信号从该模块中进行输出提供给其他模块使用，代码如下所示：

```
always@(posedge clk)
begin
  frame_dis_flag_dly1 <= frame_dis_flag;
  frame_dis_flag_dly2 <= frame_dis_flag_dly1;
end

always@(posedge clk or posedge reset)
begin
  if(reset)
    frame_dis_pulse <= 1'b0;
  else if(frame_dis_flag && (~frame_dis_flag_dly2))
    frame_dis_pulse <= 1'b1;
  else
    frame_dis_pulse <= 1'b0;
end
```

最后，当数据有效并且得到的图像的数据信号没有出现错误的情况下，将数据进行输出，并且产生输出数据有效信号 data_o_valid，代码如下所示：

```
always@(posedge clk or posedge reset)
begin
  if(reset)
  begin
    data_o       <= 'd0;
    data_o_valid <= 'd0;
  end
  else if(data_i_valid_dly1 && data_i_valid_dly2 && (~frame_dis_flag))
```

```
  begin
    data_o       <= data_i_dly1;
    data_o_valid <= 1'd1;
  end
  else
  begin
    data_o       <= 'd0;
    data_o_valid <= 'd0;
  end
end
```

1.2.2.2 数据包检测模块仿真

数据包检测模块设计完成之后，需要通过仿真对其进行测试，通过仿真波形，查看是否实现功能：每包数据的序号不连续将整帧图像数据丢弃，序号连续则保留输出。如果满足该功能，则认为仿真正确。

在“frame_check_tb”文件中例化 frame_check 模块，然后设置分别写入 50 个 16'h0505、16'hffff 和 16'h0a0a，在 tb 文件中设置 IMAGE_HEIGHT 为 50，则每次写 50 个数据之后，行像素序号加 1，代码如下所示：

```
`timescale 1ns/1ns
`define PERIOD_CLK 8

module frame_check_tb();
reg       clk;
reg       reset_n;
reg [15:0]data_i;
reg       data_i_valid;
wire      data_o_valid;
wire[15:0]data_o;
wire      frame_dis_pulse;
integer i;
initial clk = 1'b1;
always #(`PERIOD_CLK/2)clk = ~clk;
initial begin
  reset_n = 1'b0;
  data_i = 16'd0;
  data_i_valid = 1'b0;
  #(`PERIOD_CLK*200+1)
  reset_n = 1'b1;
  #200;

  for(i=0;i<50;i=i+1) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
```

```
    data_i = i;
    repeat(50) begin
    @(posedge clk)
    #1 data_i = 16'h0505;
    end
    data_i_valid = 1'b0;
    #200;
  end

  for(i=0;i<100;i=i+2) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
    data_i = i;
    repeat(50) begin
    @(posedge clk)
    #1 data_i = 16'hffff;
    end
    data_i_valid = 1'b0;
    #200;
  end

  for(i=0;i<50;i=i+1) begin
    @(posedge clk)
    #1 data_i_valid = 1'b1;
    data_i = i;
    repeat(50) begin
    @(posedge clk)
    #1 data_i = 16'h0a0a;
    end
    data_i_valid = 1'b0;
    #200;
  end
  #200;
  $stop;
end
frame_check
#(
  .IMAGE_WIDTH  (  50 ),
  .IMAGE_HEIGHT (  50 ),
  .DATA_WIDTH   (  16 )
)frame_check
(
  .clk            (clk            ),
  .reset          (~reset_n        ),
  .data_i         (data_i         ),
  .data_i_valid   (data_i_valid   ),
  .data_o_valid   (data_o_valid   ),
```

```
  .data_o         (data_o          ),
  .frame_dis_pulse(frame_dis_pulse )
);
endmodule
```

下面将对仿真波形图进行分析。

首先是获取数据序号，波形图如下图 1-9 所示。从图中可以看出，当检测到数据有效信号之后，会产生 data_i_vaild_rising 信号，此时将 data_i 中的数据提取出来，得到了数据序号并赋值给了 data_row_num。

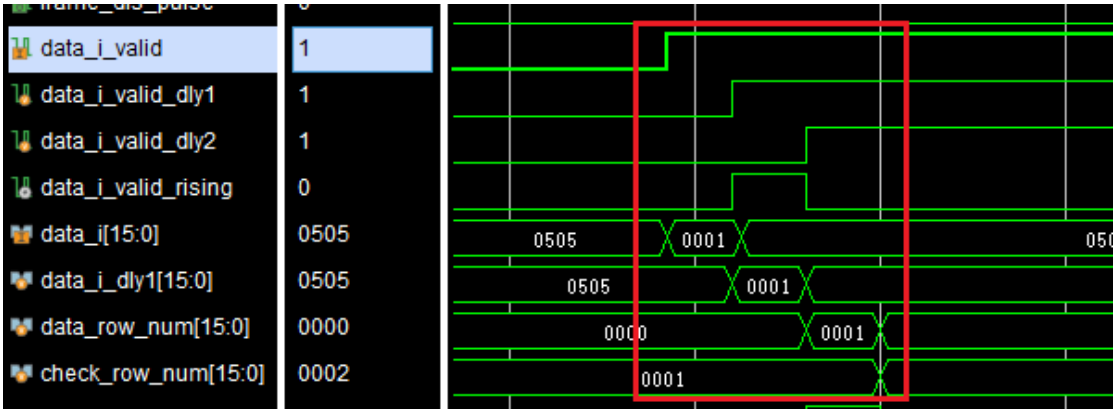

图 1-9 获取数据序号波形图

随后与 check_row_num 的值进行对比，判断序号是否一致，波形图如下图 1-10 所示。从波形图中可以看出，此时序号一致，代表数据正确，最终将数据进行输出并产生数据有效信号 data_o_valid。

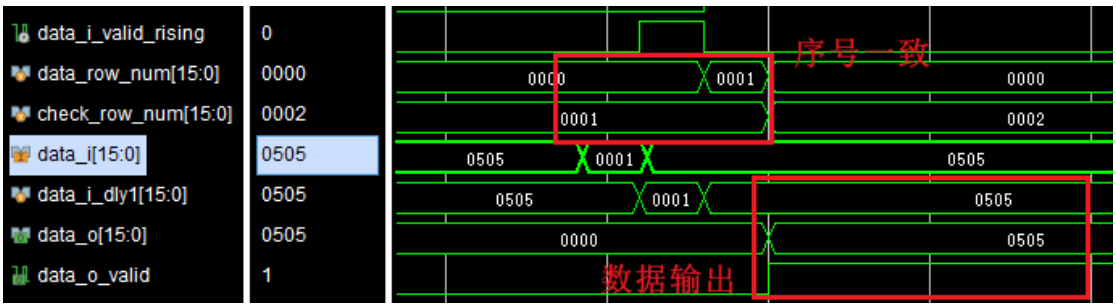

图 1-10 正确数据输出波形图、

当传输 16'hffff 的时候，波形图如下图 1-11 所示。从图中可以看出，当出现序号不一致的情况，此时说明这帧数据出现错误，会出现 frame_dis_pulse 信号，数据不会输出。

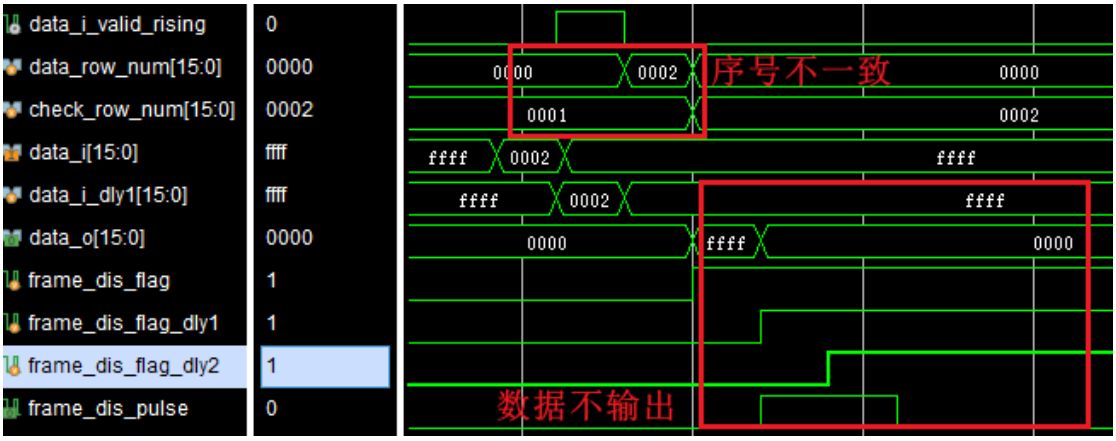

图 1-11 序号不一致数据不输出波形图

通过上述对仿真波形的分析看出，只有每包数据序号连续的情况下，数据才会输出，否则数据不输出，符合设计预期的功能。

1.2.3 fifo_axi4_adapter 模块

fifo_axi4_adapter 模块的详细设计请参看：

【ACZ702】ZYNQ PL 读写 PS DDR3 双端口读写控制器设计

http://www.corecourse.cn/forum.php?mod=viewthread&tid=29312

本章只需要例化使用即可，本次实验将使用该模块，将以太网采集得到的数据存入 PS 端的 DDR 中。

首先是 DDR 控制器的写入控制，写入的数据是从以太网接收而来，所以其写入 FIFO 时钟为 125M，当数据出现错误或者产生复位信号时，清除 FIFO。frame_check 模块最终输出的信号 image_data 为待写入的数据。FIFO 的写使能是 frame_check 模块产生的输出数据有效信号 image_data_valid，代码如下所示：

```
assign wrfifo_clr  = reset | frame_dis_pulse;
assign wrfifo_clk  = eth_clk125m;
assign wrfifo_din  = image_data;
assign wrfifo_wren = image_data_valid;
```

其次是 DDR 控制器的读出控制，最终读出的数据是提供给 TFT 屏进行显示的，所以读出时钟为 TFT 屏的工作时钟 33M，FIFO 的读使能信号为 disp_driver 模块的屏幕可见显示区标识信号 DataReq，FIFO 的读清除信号为 disp_driver 模块的一帧图像起始标识信号 Disp_Sof，最终读出的数据提供给 disp_driver 模块的输入数据信号 disp_data，代码如下所示：

```
assign rdfifo_clr  = disp_sof;
assign rdfifo_clk  = loc_clk33m;
assign rdfifo_rden = disp_data_req;
assign disp_data   = rdfifo_dout;
```

fifo_axi4_adapter 模块的完整的例化代码如下所示：

```
fifo_axi4_adapter #(
    .FIFO_DW                (16                     ),
    .WR_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR+1'b1 ),
    .WR_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR+DISP_WIDTH*DISP_HEIGHT*2),
    .RD_AXI_BYTE_ADDR_BEGIN (DDR_BASE_ADDR+1'b1 ),
    .RD_AXI_BYTE_ADDR_END   (DDR_BASE_ADDR+DISP_WIDTH*DISP_HEIGHT*2),

    .AXI_DATA_WIDTH         (64                     ),
    .AXI_ADDR_WIDTH         (32                     ),
    .AXI_ID                 (4'b0000                ),
    .AXI_BURST_LEN          (8'd15                  )
  (
    //clock reset
    .clk                (loc_clk100m         ),
    .reset              (~ps2pl_resetn_0     ),
    //wr_fifo Interface
    .wrfifo_clr         (wrfifo_clr          ),
    .wrfifo_clk         (wrfifo_clk         ),
    .wrfifo_wren        (wrfifo_wren         ),
    .wrfifo_din         (wrfifo_din          ),
    .wrfifo_full        (                    ),
    .wrfifo_wr_cnt      (                    ),
    //rd_fifo Interface
    .rdfifo_clr         (rdfifo_clr          ),
    .rdfifo_clk         (rdfifo_clk          ),
    .rdfifo_rden        (rdfifo_rden         ),
    .rdfifo_dout        (rdfifo_dout         ),
    .rdfifo_empty       (                    ),
    .rdfifo_rd_cnt      (                    ),
    // Master Interface Write Address Ports
    .m_axi_awid         (s_axi_awid          ),
    .m_axi_awaddr       (s_axi_awaddr        ),
    .m_axi_awlen        (s_axi_awlen         ),
    .m_axi_awsize       (s_axi_awsize        ),
    .m_axi_awburst      (s_axi_awburst       ),
    .m_axi_awlock       (s_axi_awlock        ),
    .m_axi_awcache      (s_axi_awcache       ),
    .m_axi_awprot       (s_axi_awprot        ),
    .m_axi_awqos        (s_axi_awqos         ),
    .m_axi_awregion     (s_axi_awregion      ),
    .m_axi_awvalid      (s_axi_awvalid       ),
    .m_axi_awready      (s_axi_awready       ),
    // Master Interface Write Data Ports
    .m_axi_wdata        (s_axi_wdata         ),
    .m_axi_wstrb        (s_axi_wstrb         ),
    .m_axi_wlast        (s_axi_wlast         ),
```

```
    .m_axi_wvalid        (s_axi_wvalid        ),
    .m_axi_wready        (s_axi_wready        ),
    // Master Interface Write Response Ports
    .m_axi_bid           (4'b0000             ),
    .m_axi_bresp         (s_axi_bresp         ),
    .m_axi_bvalid        (s_axi_bvalid        ),
    .m_axi_bready        (s_axi_bready        ),
    // Master Interface Read Address Ports
    .m_axi_arid          (s_axi_arid          ),
    .m_axi_araddr        (s_axi_araddr        ),
    .m_axi_arlen         (s_axi_arlen         ),
    .m_axi_arsize        (s_axi_arsize        ),
    .m_axi_arburst       (s_axi_arburst       ),
    .m_axi_arlock        (s_axi_arlock        ),
    .m_axi_arcache       (s_axi_arcache       ),
    .m_axi_arprot        (s_axi_arprot        ),
    .m_axi_arqos         (s_axi_arqos         ),
    .m_axi_arregion      (s_axi_arregion      ),
    .m_axi_arvalid       (s_axi_arvalid       ),
    .m_axi_arready       (s_axi_arready       ),
    // Master Interface Read Data Ports
    .m_axi_rid           (4'b0000             ),
    .m_axi_rdata         (s_axi_rdata         ),
    .m_axi_rresp         (s_axi_rresp         ),
    .m_axi_rlast         (s_axi_rlast         ),
    .m_axi_rvalid        (s_axi_rvalid        ),
    .m_axi_rready        (s_axi_rready        )
);
```

模块设计完成之后，只需要在顶层文件中对各个模块之间的接口信号进行连接，完整的顶层文件代码请自行查看例程文件。

1.3 建立 SDK 工程

经过以上工作，代码设计部分的任务已经全部完成，但是用到了 PS 部分的内容，所以我们需要新建一个空的 SDK 工程用于下载程序，步骤如下。

1. 编译工程，然后依次点击 File->Export->Export Hardware，导出 bit 文件，如下图 1-12 所示。

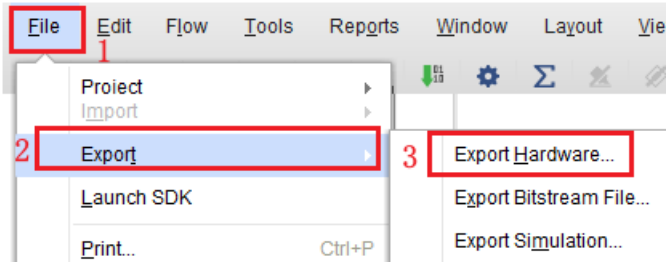

图 1-12 导出 bit 文件

2. 进入 SDK 软件，点击 File->Launch SDK，如下图 1-13 所示。

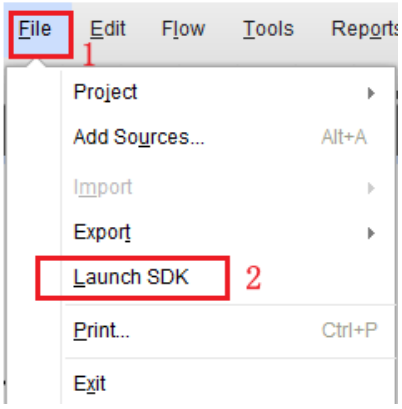

图 1-13 进入 SDK 软件

3. 进入 SDK 软件之后，依次点击 File->New->Application Project，建立一个 SDK 工程，如下图 1-14 所示，然后给工程命名，比如命名为 eth_udp_rgmii_ddr3_tft，然后点击 Next，选择 Empty Application，就完成了工程的创建。

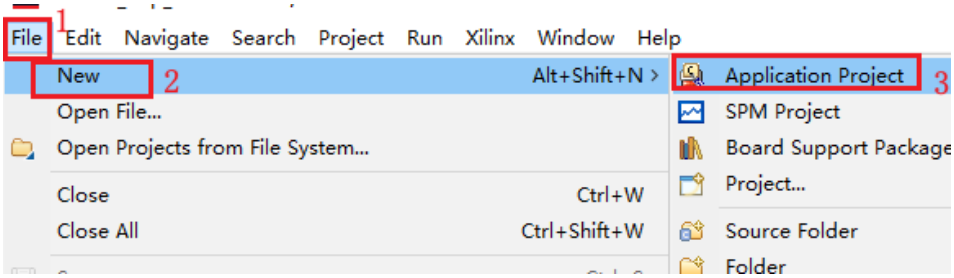

图 1-14 新建 SDK 工程

4. src 文件夹下，新建一个源文件，如下图 1-15 所示，然后在弹出的界面中给文件命名为 main.c。

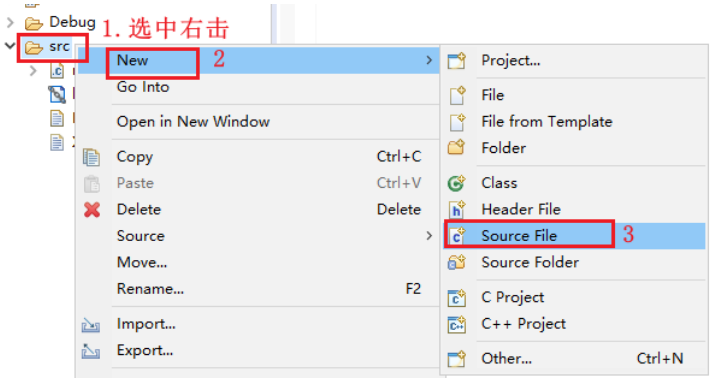

图 1-15 新建源文件

5. 源文件中添加代码，如果不添加会报错，这里添加没有意义的一句代码，如下所示：

```
int main(void)
{
    return 0;
}
```

最后 crtl+b 编译工程，没有报错，这样本次实验的设计就完成了，接下来便可以进行板级验证了，需要注意的是，如果你修改了 vivado 中的代码，你需要进行上面的步骤 1，导出 bit 文件，然后在 SDK 中重新编译工程，这样才算修改成功了。

1.4 板级验证

经过以上工作，代码设计部分的任务已经全部完成，接下来将进行板级验证。本次实验的板级验证环节，主要验证：通过以太网传图工具将图像数据传输到 ACZ702 开发板上的 DDR3 进行存储，然后图像数据由 TFT 控制器主动从 DDR3 中读取出来，送往 TFT 显示屏进行显示，最终查看 TFT 屏上的图像数据是否显示正常。

1.4.1 系统所需硬件

1. ACZ702 开发板一块
2. 网线一根
3. 5 寸 TFT 屏一个
4. Type-C 下载线一根
5. 电源线一根

1.4.2 硬件连接

本次设计系统硬件连接如下图 1-16 所示：

1. 使用 Type-C 线连接开发板调试接口（靠近电源接口）和电脑 USB 口
2. 将开发板电源拨码开关拨到 DC 供电。
3. 使用 12V 的电源给开发板供电，
4. 将网线连接至 PL 侧的网口上
5. 使用软排线连接开发板和 TFT 屏，软排线的蓝色部分要在外边。



图 1-16 硬件连接图

1.4.3 烧录程序

在 SDK 软件中，依次点击 Run->Run Configurations，如下图 1-17 所示。

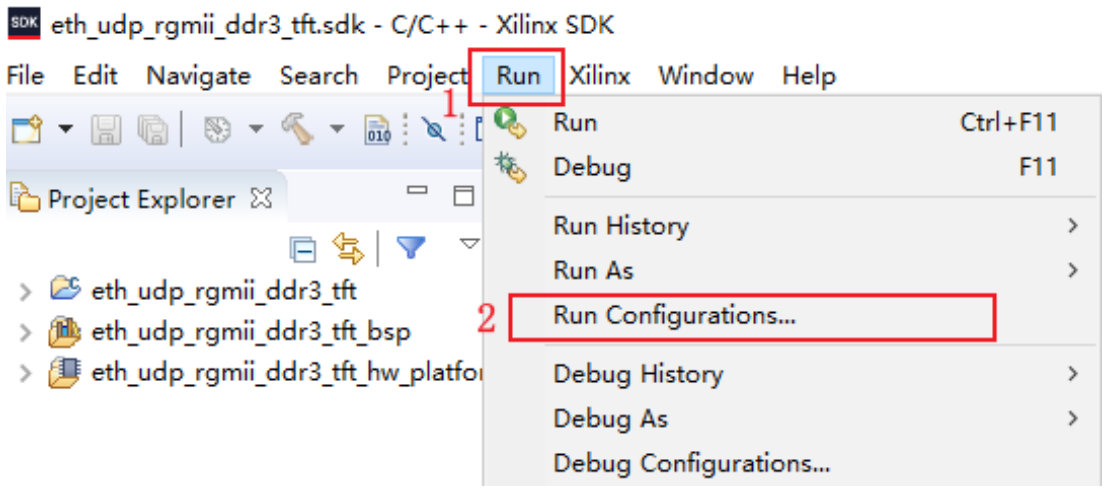

图 1-17 进入下载界面示意图

进入下载界面之后，下载 bit 文件，如下图 1-18 所示。

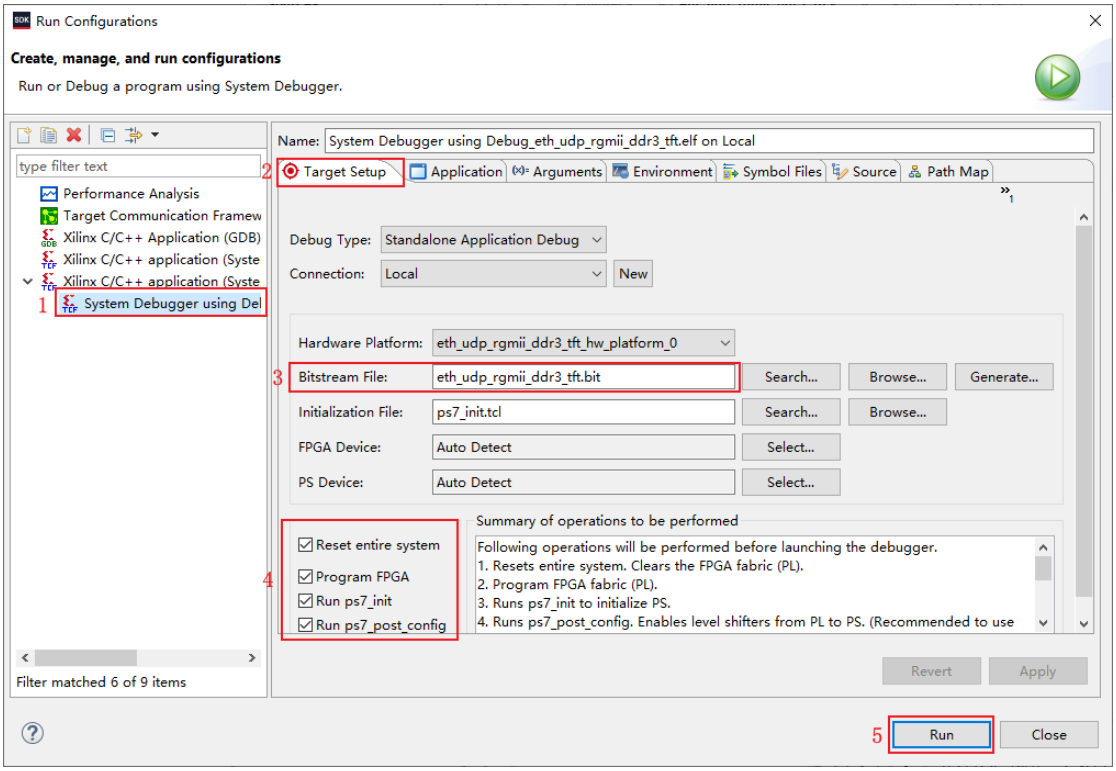

图 1-18 下载 bit 文件

下载成功之后，TFT 显示屏上会显示杂乱的图像，如下图 1-19 所示，这是因为当前 DDR3 中还没有有意义的数据的原因，当前存储的都是随机数。



图 1-19 TFT 显示花屏示意图

1.4.4 修改电脑 IP 地址

本次实验设定了目标 IP 地址（PC 端）为 192.168.0.3，用户需要将自己电脑上的以太网 IP 地址修改为该地址，在本地连接状态中，点击属性，并在弹出的属性对话框中双击【Internet 协议版本 4 （TCP/Ipv4）】选项，然后在弹出的属性对话框中设置静态 IP 地址。如下图 1-20 所示。

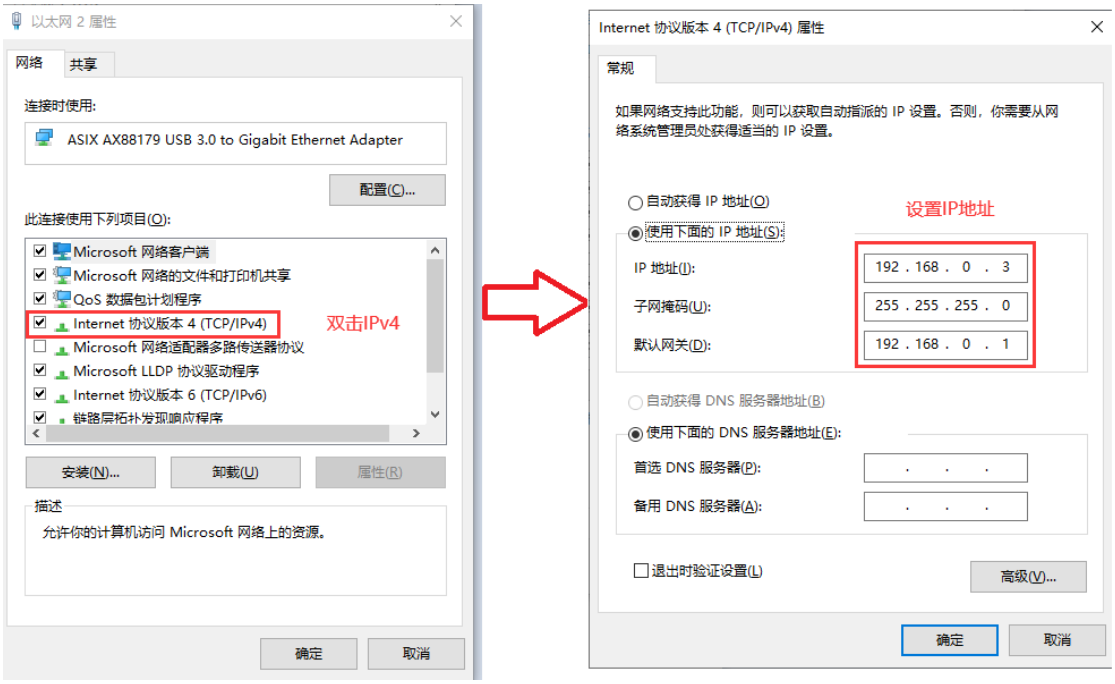

图 1-20 修改电脑 IP 地址

1.4.5 绑定 ARP

本工程不支持 ARP 协议，只能通过静态绑定的方式来强制将开发板的 IP 地址和 MAC 地址关联在一起。这样，当 PC 发送给 192.168.0.2 的数据包的时候，目标 MAC 地址自动为开发板的 MAC 地址。

关于 ARP 的绑定请查看以下帖子内容：

以太网通信静态 ARP 绑定方法与常见问题解决方案

http://www.corecourse.cn/forum.php?mod=viewthread&tid=28645

1.4.6 Picture2Hex 生成图片数据文件

1. 读者在找到我们提供的 Picture2Hex.exe 软件，软件打开之后，配置界面如下图 1-21 所示，将图像数据的 width 和 high 设置为 800*480。

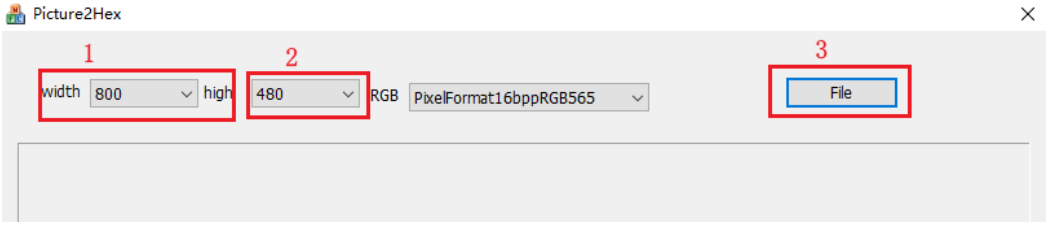

图 1-21 Picture2Hex 软件配置界面

2. 点击 File 之后，找到待显示的图片，图片支持 bmp，png，jpg 格式，选择完图片后，在 work 目录下会生成一个 logo.c 文件，这个文件就是图片转换的数据文件，如下图 1-22 所示。

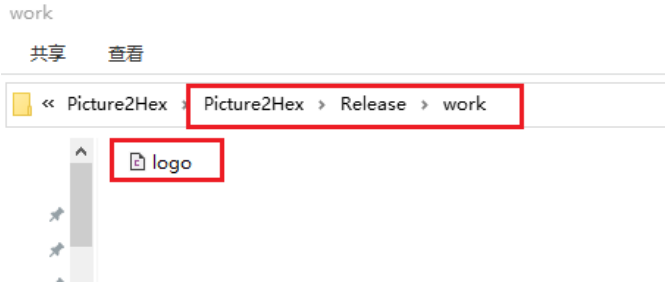

图 1-22 生成的图像.c 文件

3. 将 logo.c 改名为 test800x480.c。
4. 将 test800x480.c 拷贝到以太网传图上位机的 img_data 目录下，如下图 1-23 所示。



图 1-23 拷贝文件只以太网上位机文件夹下

1.4.7 运行结果显示

运行以太网传图上位机“以太网传图.exe”，这样可以看到开发板 TFT 屏上显示 PC 下传的图片。显示效果如下图 1-24 所示。



图 1-24 最终效果显示图

1.5 常见问题说明

当实验现象无法出现时，可以考虑从如下方面查看问题。

1. 检查你的电脑有线网络本地连接是否连接上了。检查你的电脑 IP 地址是否已经设置为了 192.168.0.3.
2. 接收不到图像的情况下，手动确认防火墙是否已经被关闭。
3. 确保开启巨型帧。开启巨型帧的方式如下图 1-25 所示。

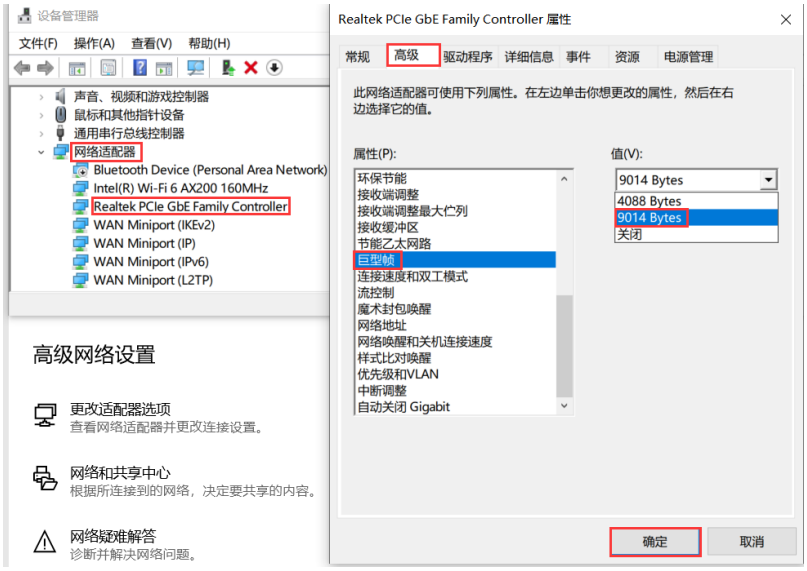

图 1-25 电脑开启巨型帧方式

4. 点击“本地连接”文字，以查看该网络状态，确认当前连接速度为千兆速率（1000.0 Mbps），如下图 1-26 所示。



图 1-26 确保连接速度为千兆速率

1.6 思考与总结

本章实验通过以太网上传图工具，将需要传输的图像的数据通过以太网传

输至 DDR3 中进行存储，最终由 TFT 控制器主动读取出来，送给 TFT 屏进行显示。

本章实验主要讲解了以太网控制模块和数据包检测模块的设计与仿真，这两个模块主要都是用来检测传输的数据是否正确，在介绍以太网控制模块时，提到了使用两个 FIFO 进行双缓存，这样做是为了在 CRC 校验出现问题时把这包数据丢掉，并且确保不影响下一包数据的传输。使用双 FIFO 缓存就是一个“乒乓”操作，是一个用于数据流控制的处理技巧，也是本次实验一个比较核心的内容，希望读者可以好好理解，方便在之后自己的工程中可以学以致用。