小眼睛科技

提供一站式 FPGA&嵌入式解决方案

盘古 100K MINI 开发板
图像实验指导手册

小眼睛科技 盘古 676 系列

盘古 100K MINI（MES2L676-100HP-MINI-MINI）开发板实验教程

紫光同创 Logos2 系列 PG2L100H 开发平台

文档版本修订记录

版本号	发布日期	修订记录
V1.0	2025/10/20	初始版本

公司名称: 深圳市小眼睛科技有限公司

地址: 深圳市宝安区西乡街道 F518 时尚创意园

官方网址: www.meyesemi.com

官方淘宝店铺: 小眼睛半导体

B 站: 小眼睛半导体 (视频教程免费学)

* 加入 FPGA 开发者技术交流与 5000+FPGA 开发者实时沟通

QQ2 群: 442106123QQ3 群: 882634519)

*配套资料下载、技术答疑请登录逻辑矩阵技术论坛

LOGIC MATRIX 逻辑矩阵

逻辑矩阵技术论坛欢迎各位发烧友加入

让我们共建开源生态, 持续赋能行业发展

https://www.szlogicmatrix.com/

*扫码注册开源技术论坛

*扫一扫关注官微

* 官方旗舰店

目录

1. 图像仿真平台搭建

1.1. 实验简介

实验目的:

搭建一个可以快速验证图像算法的仿真平台。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b(可以用别的版本)

硬件环境:

MES2L676-100HP-MINI

1.2. 实验原理

首先需要了解 VESA 视频时序, 了解视频的标准时序是怎样的, 然后通过仿真模拟视频流。最后在 Matlab 读取 Modelsim 仿真出来的 txt 文件, 将 txt 文件转为 png 图片, 即可观察到仿真后的图像效果。至于 Matlab 语言,使用过的话,大家其实可以通过 AI 来生成一个脚本来处理 txt 文本。不熟悉的同学需要去学习一下。

1.2.1.VESA 时序介绍

屏幕的显示原理, 大家可以百度去了解一下, 这里笔者就不做说明, 只介绍简单的显示原理。目前, 我们常用的屏幕的分辨率基本都是 1080P@60HZ,也有 75HZ, 120HZ, 170HZ 等。刷新率越高, 意味着屏幕每秒刷新更多次, 画面显示更流畅, 尤其是视频画面高速运动时, 帧率高低的差别就更能体现。一般来说, 我们的显示是从屏幕的左上方开始, 从上到下, 从左到

右,一行一行的显示,直到显示完一帧。如图 1.2-1 所示,如果是 1080P 那么一行就会有 1920 个像素点, 总共有 1080 行, 即 1920*1080。

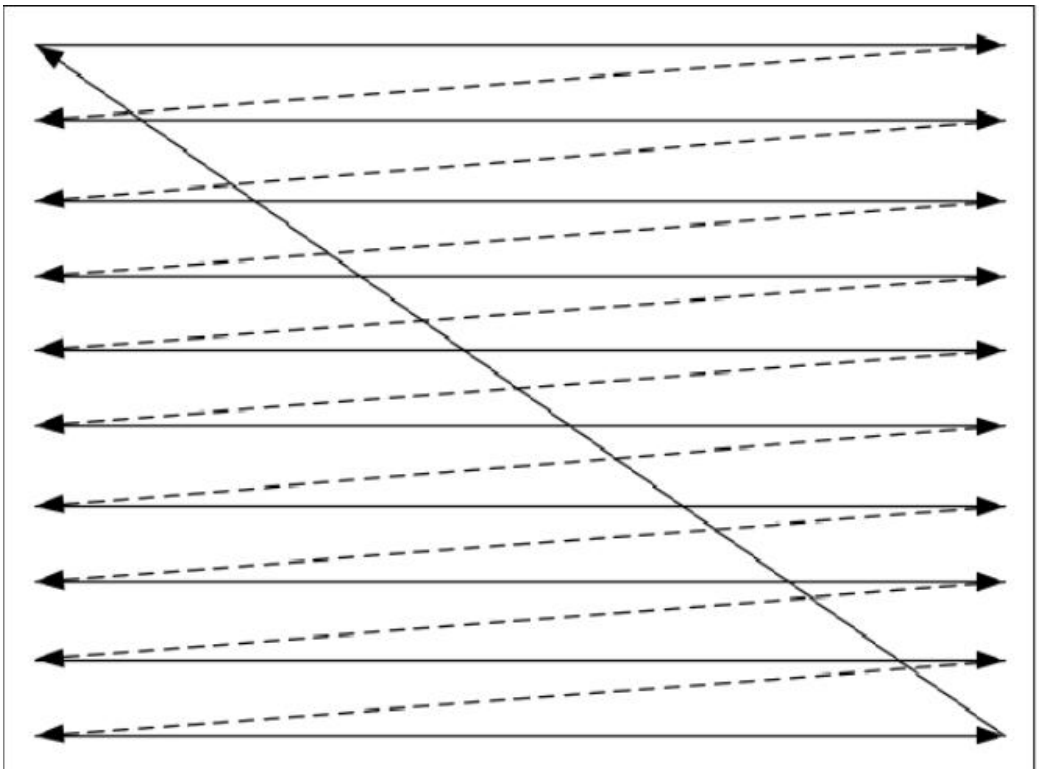


图 1.2-1

为了适应匹配不同厂家的显示器，其视频传输接口都有一套标准的时序接口, 只有遵循该时序标准, 才能正确的进行图像信息的显示。而一般我们都以 VESA VGA 的标准时序作为参考, 以 VESA 标准时序去驱动不同厂家的显示器。

接下来我们直接介绍 VESA 1080P 的显示时序。通过其时序来了解怎么去生成测试图像。包括后续的 HDMI 显示也是如此, 只要生成了其对应的显示时序, 就可以点亮屏幕。而我们的开发板上有一个视频编码芯片, 会把该时序编码为 TMDS 信号输出, 因此我们不需要去了解 TMDS 编码, 给出正确的时序即可。

1.2.2.显示时序

在介绍具体分辨率的显示时序时, 我们要先了解一个概念, 就是消隐区间, 它又由显示前

沿、显示后沿和同步信号构成。

首先介绍同步信号，这里我们引出两个信号叫行同步信号和场同步信号，行同步信号一般用来指示一行的开始或结束。场同步信号用来指示一帧的开始或结束。这两个信号其实是有极性的，具体看下面四张图：

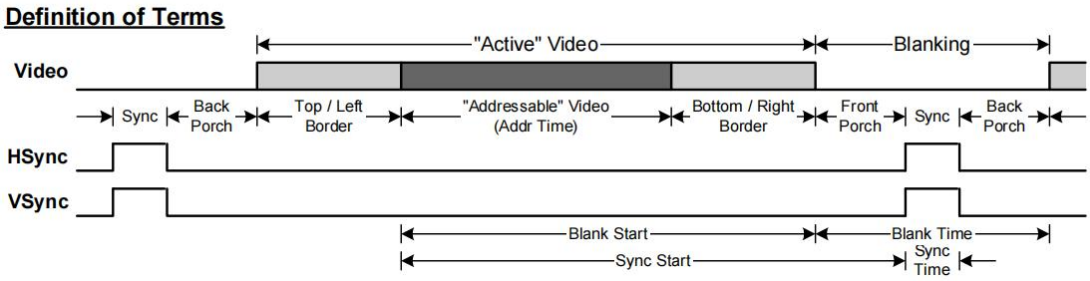


图 1.2-2

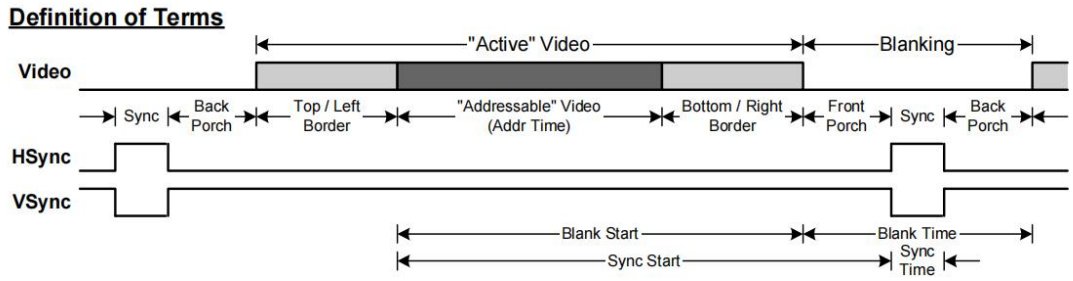


图 1.2-3

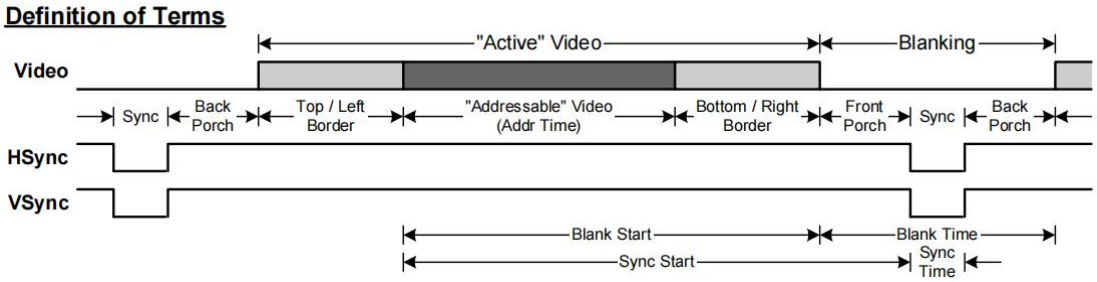


图 1.2-4

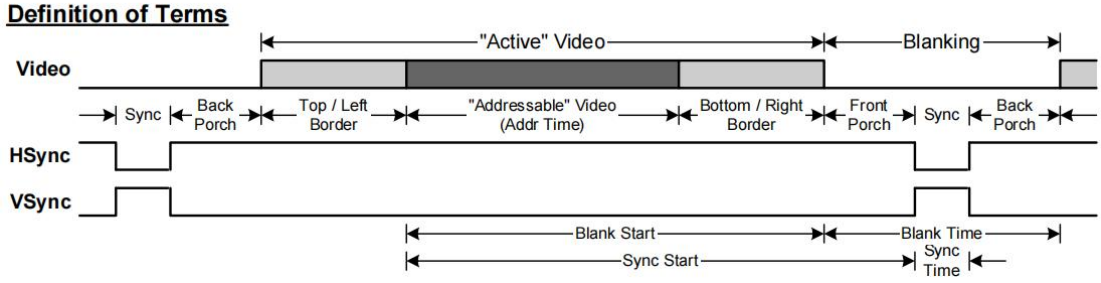


图 1.2-5

我们看上面四个图, 细心的话可以发现, HSync(行同步信号)和 VSync(场同步信号)会有四种不同的组合方式, 所以大家要确定好当图像数据有效, 也就是 Active Video 时, 我们的 HSync 和 VSync 分别是高电平还是低电平。例如图 1.2-2,我们可以看到在 SYNC(同步)阶段时,HSync 和 VSync 都是高电平,此时数据无效,处于同步阶段,我们称此时的 HSync 和 VSync 是**高电平有效**, 注意, 这里高电平有效不是指数据有效, 是指同步有效, 此时是没有数据的, 注意区分好这个高电平有效。而图 1.2-4 对应的就是低电平有效。图 1.2-3 就是 HSync 高电平有效、VSync 低电平有效。

接下来我们关注一下 Video 信号, 也就是我们的像素数据, 以图 1.2-2 为例子, 我们看到其由 SYNC(同步)、Back Porch(显示后沿)、Top/Left Border(上/左显示边框)、ADDRes sable Video(有效显示区间)、Botton/Right Border(下/右显示边框)、Front Porch(显示前沿)构成, 然后我们看到 Blanking, 即消隐区间, 可以看到由 Front Porch、SYNC、Back Porch 构成。有些读者可能会疑惑, 在最开始的时候, 为什么同步后, 就是显示后沿呢。

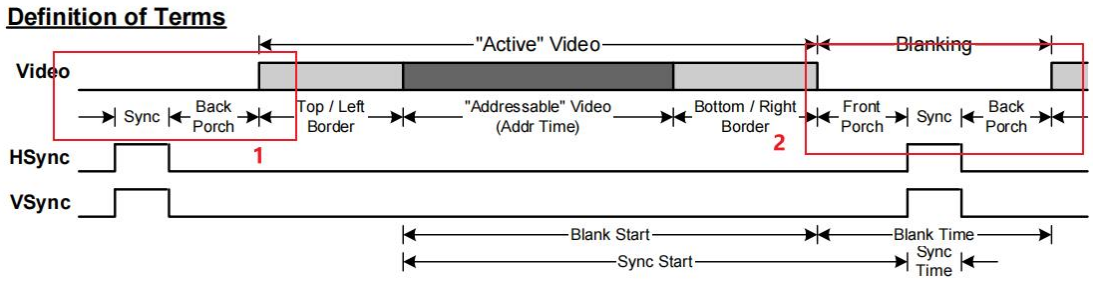


图 1.2-6

我们看图 1.2-6 的红框 2 处, 可以看到在那一时刻, 其实已经是一帧的结尾, 当下一个 VSYNC 到达时, 其实就是下一帧的开始, 故在这个 SYNC 前, 是叫显示前沿, 同步结束后, 就是我们的显示后沿, 所以看红框 2 处, 应该很好理解的, SYNC 阶段是可以用来判断一帧是否开始, 所以我们显示下一帧的前沿就在 SYNC 之前, 经过 SYNC 后, 新的一帧开始, 所以经历

这个显示后沿。这样的话就比较好理解了。

还有要注意的是, 在实际之中, 我们的 Boder, 不管是 Top 还是 Left 还是 Bottom 还是 Right, 一般情况下, 均为 0。所以, 大部分情况下是不需要管 Boder 的。

所以大家可以看到, 我们的有效区域其实就是中间 Addressable Video 那一段区间, 其余都是无效数据, 所以我们把总的时间称为行扫描周期和场扫描周期。以 1080P 为例子, 其总的行扫描周期为 2200, 场扫描周期为 1125, 有效的宽度是 1920, 有效的高度是 1080。这里的有效宽度和有效高度指的就是 Addressable Video 那一段区间,从当前帧的同步开始到下一帧的同步开始的时间就是行扫描周期*场扫描周期。

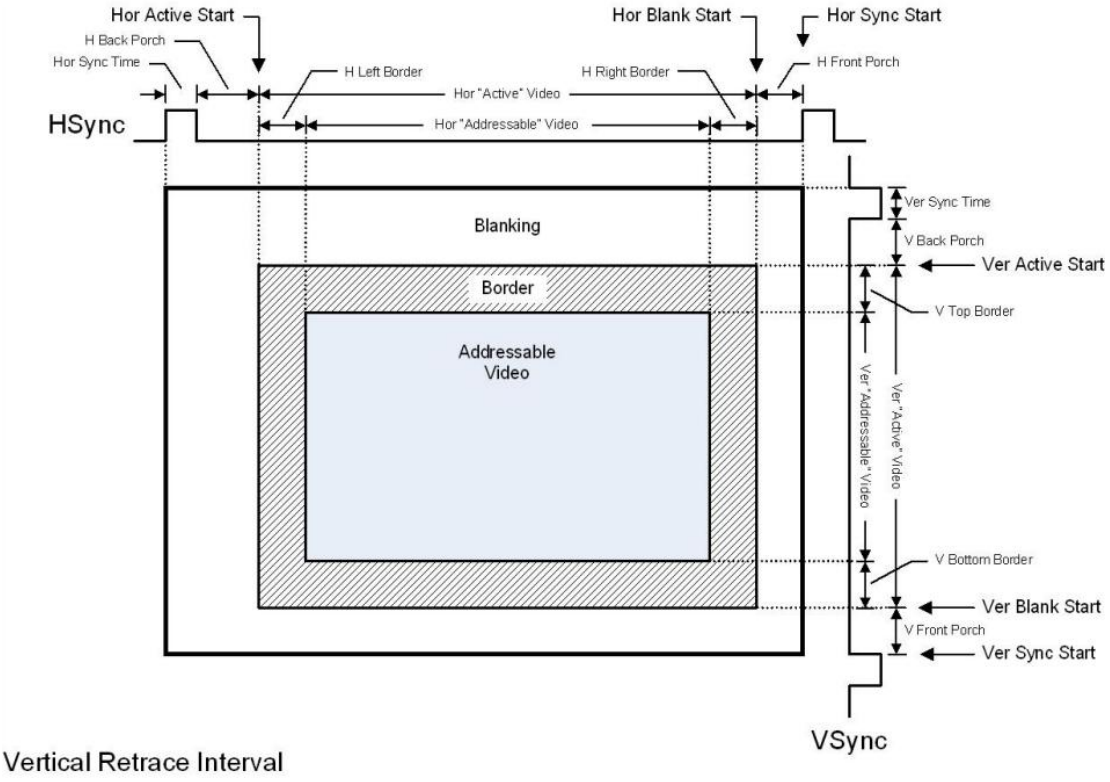


图 1.2-7

我们再看图 1.2-7, 再来深刻理解一下显示的时序, 可以很快看出, 该 HSync 和 VSync 均为高电平有效, 可以看到有三个矩形, 黑实线就是总的开窗的大小, 虚线部分就是 Border,

一般为 0, 最后的淡蓝色部分就是我们的有效区间, 在实际之中, 一帧的开始, VSync 会先拉高, 之后 HSync 再拉高, 表示一行开始, 假设我们的 VSync 为 5 的话, 那要经过 5 个 HSync, 也就是扫描 5 行后,VSync 才会拉低,当 HSync 和 VSync 都拉低后,还要经过 Back Porch(显示及后沿)后我们的视频数据才有效。

注意, 默认 Border 为 0, 所以后文将不再提起这个信号, 我们看 Blanking 部分, 就是白色部分, 可以看到白色区域所对的信号是 Front Porch、SYNC、Back Porch 构成的。然后中间的 Addressable Video 区域则是由 HSync==0 && VSync==0 得到的一个矩形区间。

综上, 这就是我们的标准显示时序。大家在实际之中要自己生成显示时序的话就需要直到 Front Porch、SYNC、Back Porch 和有效显示宽度/高度, 就可以自己生成显示时序了。可以定义一个行扫描计数器和场扫描计数器, 按照上述的显示时序去计数, 先经过 SYNC, 再经过 Back Porch, 然后就是有效区域, 然后再经过 Front Porch。然后再重新经过 SYNC, 反复循环即可生成对应的显示时序, 具体可以看后续的代码讲解。同样的, 我们还需要获得对应的像素时钟, 这个需要根据要显示的帧率来得到我们的像素时钟。接下来, 我们来介绍该时钟如何计算。

1.2.3.像素时钟频率计算

一般来说, 根据 vesa 时序标准, 驱动显示的时钟频率的计算公式如下: (以 1080P 为例)

$$2200 * 1125 * 60 = 148500000\text{HZ} = 148.5\text{MHZ}$$

其实就是行扫描周期*场扫描周期*帧率, 所以如果要 120HZ, 那就是 297MHZ 的时钟。

1.3. 接口列表

video_data_gen.v 模块, 用来模拟生成视频流数据。

端口	I/O	位宽	描述

TOTAL_WIDTH	/	12	总宽度
IMG_WIDTH	/	12	图像有效宽度
H_SYNC	/	12	行同步
H_BP	/	12	行后沿
H_FP	/	12	行前沿
TOTAL_HEIGHT	/	12	总高度
IMG_HEIGHT	/	12	图像有效高度
V_SYNC	/	12	场同步
V_BP	/	12	场后沿
V_FP	/	12	场前沿
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
video_vs	output	1	场信号, 低电平有效
video_de	output	1	图像数据有效信号
video_data	output	24	图像数据

要修改输出的模拟视频流的分辨率大小, 只需要根据对应分辨率的时序修改对应的行场分辨率参数即可。默认是 1280*720@60HZ 的参数, 视频流的具体帧率取决于 video_clk 的大小, 可以根据像素时钟的计算公式去计算。

1.4. 工程说明

1.4.1.代码模块说明

主要介绍 Matlab 代码和 verilog 代码。介绍如何生成测试数据,并在 Modelsim 仿真中保存出 txt 文件, 并用 Matlab 读取后转成 png 图片显示。

1.4.1.1. Matlab 代码说明

我们可以通过 Matlab 将我们的图片转成 txt 文件以便 Modelsim 进行读取, 同样的, Modelsim 对 txt 的数据仿真处理后的 txt, 也可以通过 Matlab 将其恢复为图片格式然后显示,以便观察仿真结果。当然,也可以直接在 Modelsim 中保存出 BMP 图片,但需要按照 BMP 格式去写入才能完成。也可以用 Python 搭建。具体如何编写, 其实用 GPT 等 AI 工具, 然后再人工改改, 挺快的。本次介绍以 Matlab 为例子。

首先可以准备一张分辨率为 1280*720 的图片,后续的算法仿真都用这个分辨率为例子。

图 1.4-1

以下是 Matlab 代码

img2txt.m

```
%% 将图片转为 txt 以供 modelsim 读取
clc;
clear all;

%% 读取图像文件 换成自己图片的路径 bmp/png/jpg 均可
image_rgb = imread('testst.png');

%% 显示图像
imshow(image_rgb);

%% 获取图像的尺寸信息
[image_height, image_width, num_channels] = size(image_rgb);

%% 打开文件以写入
file_id = fopen('img.txt', 'w+');



%% 遍历每一个像素并写入到文件中
for row_index = 1:image_height
   for col_index = 1:image_width
      for channel_index = 1:num_channels
```

```
        % 将每个像素值写入文件, 以十六进制格式表示
        fprintf(file_id, '%02x', image_rgb(row_index, col_index, channel_index));

    end
    % 每行像素结束后换行
    fprintf(file_id, '\n');
  end
end

%% 关闭文件
fclose(file_id);
```

图像可以是 RGB888 格式或者灰度格式, 代码的第 6 行是读取图片, 大家要换成自己图片的路径。代码的第 12 行会自动获取图像的分辨率以及通道数。代码的 18-27 行就是遍历图像的每一个数据并将其转成 16 进制, 然后写入一个文件里, 每写完一行像素就在 txt 里换行, 直到整张图片写入完成, 关闭文件。第 15 行, 就是创建一个 txt 文件。

接下来是将 txt 数据恢复成图片。

fpga_hex2img.m

```
%% 处理 RGB 三通道
clear all;
clc;
close;
%% 定义图像分辨率
```

```
image_width  = 1280;
image_height = 720;

%% 读取文本数据 16 进制 三通道
[hex1, hex2, hex3] = textread('img_process.txt', '%s %s %s');

%% 16 进制转 10 进制
image_data_dec = hex2dec([hex1, hex2, hex3]);  % 16 进制转 10 进制

%% 根据分辨率构建图像矩阵
image_matrix = reshape(image_data_dec, image_width, image_height, 3);

%% 转为无符号 8bit
image_uint8 = uint8(image_matrix);

%% 旋转 使图像正常显示
rotated_image = imrotate(image_uint8, -90);

%% 将图片进行水平翻转
flipped_image = flip(rotated_image, 2);

%% 显示部分
```

```
subplot(121)
%% 读取未处理的原图像
img = imread("night1.bmp");
imshow(img);
title('原图像')
subplot(122);
%% 保存图片
imwrite(flipped_image, 'modelsim_process.bmp');
%% 显示 Modelsim 处理后的图片
imshow('modelsim_process.bmp');
title('modelsim 仿真图像')
```

代码的第 6-7 行, 所定义的分辨率一定要和代码里面定义的一致。第 10 行是读取 txt 里面的数据, 按照 RGB888 的格式去读取, 因为我们的仿真代码里也是按照 RGB888 去保存的。由于我们的图像是无符号的 8 位格式, 所以在代码的第 19 行需要强制转为 uint8 格式。这时候其实可以恢复出图像了, 但是此时恢复出来的图像其方向不太一样。是竖直方向。如下图所示:

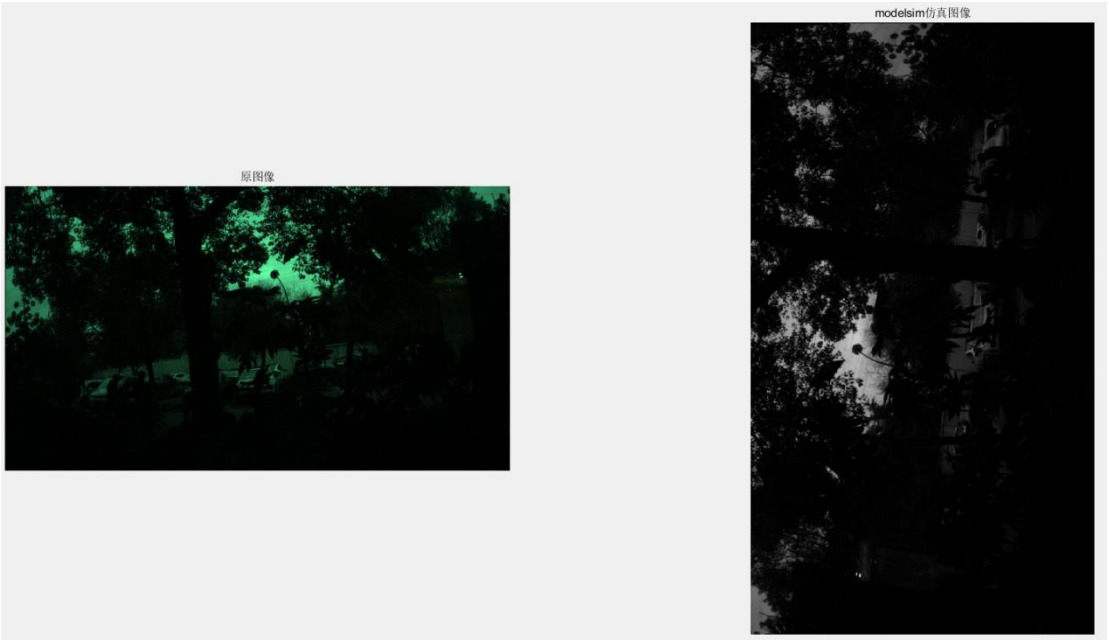


图 1.4-2

经过测试, 需要向右转 90 度再进行水平翻转。即可恢复成正常显示的图像。

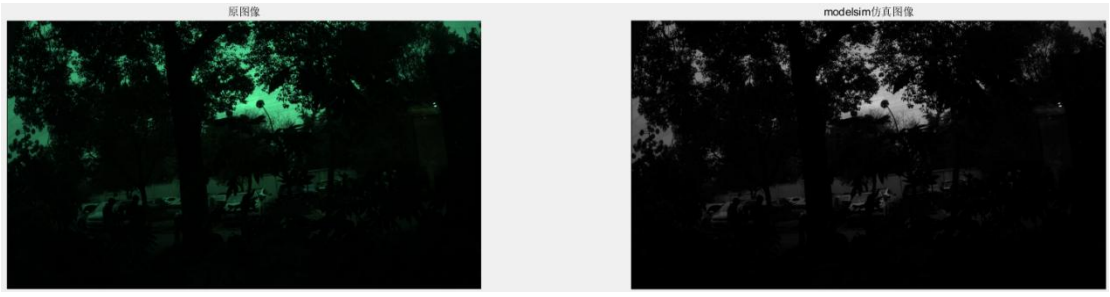


图 1.4-3

代码的 27-38 行完成图像的显示, 左边显示的是原图像, 右边显示的是 Modelsim 处理后的图像。

1.4.1.2. Modelsim 代码说明

对于 Modelsim 的仿真, 接下来就是要读取 Matlab 生成的 txt 文件来生成图像数据。

搭

代码如下(使用的 systerm verilog, 语法都是一样的, 只是仿真比较方便):

img_data_read.sv

```
//仿真使用, 用来读取图片数据
//读取彩色图像
//默认 1280*720 @60hz
module  video_data_gen
#(
  //同步+后沿+有效+前沿 = 总
  parameter  TOTAL_WIDTH   =  12'd1650   ,  //总宽度
  parameter  IMG_WIDTH     =  12'd1280   ,  //有效宽度
  parameter  H_SYNC        =  12'd40     ,  //同步
  parameter  H_BP          =  12'd220    ,  //后沿
  parameter  H_FP          =  12'd110    ,  //前沿

  parameter  TOTAL_HEIGHT  =  12'd750    ,  //总高度
  parameter  IMG_HEIGHT    =  12'd720    ,  //有效高度
  parameter  V_SYNC        =  12'd5      ,  //同步
  parameter  V_BP          =  12'd20     ,  //后沿
  parameter  V_FP          =  12'd5         //前沿
)
(
```

```
    input   wire        video_clk   ,   //50MHZ 100MHZ
    input   wire        rst_n       ,

    //输出
    output  wire        video_vs    ,   //场同步信号 低电平有效
    output  wire        video_de    ,

    output  wire  [23:0]  video_data
);
//存放图像数据的数组  默认 1280*720 避免仿真过长其实可以改小
reg [23:0]  img_data_reg   [IMG_WIDTH*IMG_HEIGHT];

integer i;
//图像数组初始化为 0
initial
begin
    for (i=0; i<IMG_WIDTH*IMG_HEIGHT; i=i+1)
        img_data_reg[i] = 0;
end

//读取图像数据
```

```
initial
begin
   $readmemh("D:/pango_isp/img_test_pg/img_test_pg/01_led_test/sim/img.
txt",img_data_reg);
end



//统计一帧图片 包括行场同步信号这些  只要 不超过总的行场就行
reg [11:0]  x_cnt;
reg [11:0]  y_cnt;

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
   begin
      x_cnt   <=   12'd0;
      y_cnt   <=   12'd0;
   end
   else   if(x_cnt == TOTAL_WIDTH-1 && y_cnt == TOTAL_HEIGHT-1)   //一帧
   begin
      x_cnt   <=   12'd0;
```

```
        y_cnt  <=  12'd0;
    end
    else  if(x_cnt == TOTAL_WIDTH-1)
    begin
        x_cnt  <=  12'd0;
        y_cnt  <=  y_cnt + 1'b1;
    end
    else
        x_cnt  <=  x_cnt + 1'b1;
end

reg video_vs_d;
reg video_hs_d;
//video_vs_d 的上升沿就是一帧的开始  下降沿就是一帧结束
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d  <=  1'd0;
    else  if(y_cnt > (V_SYNC - 1) && y_cnt <= (V_SYNC+V_BP+IMG_HEIGHT+V_FP- 1)
)
        video_vs_d  <=  1'd1;
    else
        video_vs_d  <=  1'd0;
```

```
end



always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    video_hs_d  <=  1'd0;
  else  if(x_cnt > (H_SYNC - 1) && x_cnt <=  (H_SYNC+H_BP+IMG_WIDTH+H_FP -
1))
    video_hs_d  <=  1'd1;
  else
    video_hs_d  <=  1'd0;
end



wire   video_de_d;//有效信号
assign  video_de_d = (x_cnt > (H_SYNC+H_BP - 1) && x_cnt <=  (H_SYNC+H_BP+I
MG_WIDTH - 1) && y_cnt > (V_SYNC+V_BP-1) && y_cnt <= (V_SYNC+V_BP+IMG_HEI
GHT-1))?1'b1:1'b0;

//数组坐标计数
reg [31:0]  img_index;   ///数组的索引

```

```
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    img_index  <=  32'd0;
  else  if(img_index == IMG_WIDTH*IMG_HEIGHT - 1)  //读完一帧
    img_index  <=  32'd0;
  else  if(video_de_d)
    img_index  <=  img_index + 1'b1;
  else
    img_index  <=  img_index;
end


//图像数据读出
reg [23:0]  video_data_reg;
always@(posedge video_clk or negedge rst_n) begin
  if(video_de_d)  //当读数据有效
    video_data_reg  <=  img_data_reg[img_index];
  else
    video_data_reg  <=  24'd0;
end
//数据延迟 video_de_d 一个 clk
reg video_de_delay;
```

```
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    video_de_delay  <=  1'd0;
  else
    video_de_delay  <=  video_de_d;
end

assign video_vs  = video_vs_d     ;
assign video_de  = video_de_delay  ;
assign video_data= video_data_reg  ;


endmodule
```

该部分模块建议是看视频来理解, 这里文字只会对重点部分进行介绍, 有些地方也可以看代码注释。

看代码的 7-17 行定义了视频时序参数, 默认是 1280*720@60HZ 的参数, 该值的确定, 大家可以去看 VESA 手册, 如图 1.4-4 所示

```
Timing Name          = 1280 x 720 @ 60Hz;
Hor Pixels           = 1280;          // Pixels
Ver Pixels           = 720;           // Lines
Hor Frequency        = 45.000;        // KHz    =   22.2 usec   /  line
Ver Frequency        = 60.000;        // Hz     =   16.7 msec   /  frame
Pixel Clock          = 74.250;        // MHz    =   13.5 nsec      ± 0.5%
Character Width      = 1;             // Pixels =   13.5 nsec
Scan Type            = NONINTERLACED;                 // H Phase =    3.3 %
Hor Sync Polarity    = POSITIVE;      // HBlank =   22.4% of HTotal
Ver Sync Polarity    = POSITIVE;      // VBlank =    4.0% of VTotal
Hor Total Time       = 22.222;        // (usec) =   1650 chars  =  1650 Pixels
Hor Addr Time        = 17.239;        // (usec) =   1280 chars  =  1280 Pixels
Hor Blank Start      = 17.239;        // (usec) =   1280 chars  =  1280 Pixels
Hor Blank Time       = 4.983;         // (usec) =    370 chars  =   370 Pixels
Hor Sync Start       = 18.721;        // (usec) =   1390 chars  =  1390 Pixels
// H Right Border    = 0.000;         // (usec) =      0 chars  =     0 Pixels
// H Front Porch     = 1.481;         // (usec) =    110 chars  =   110 Pixels
Hor Sync Time        = 0.539;         // (usec) =     40 chars  =    40 Pixels
// H Back Porch      = 2.963;         // (usec) =    220 chars  =   220 Pixels
// H Left Border     = 0.000;         // (usec) =      0 chars  =     0 Pixels
Ver Total Time       = 16.667;        // (msec) =    750 lines     HT – (1.06xHA)
Ver Addr Time        = 16.000;        // (msec) =    720 lines       = 3.95
Ver Blank Start      = 16.000;        // (msec) =    720 lines
Ver Blank Time       = 0.667;         // (msec) =     30 lines
Ver Sync Start       = 16.111;        // (msec) =    725 lines
// V Bottom Border   = 0.000;         // (msec) =      0 lines
// V Front Porch     = 0.111;         // (msec) =      5 lines
Ver Sync Time        = 0.111;         // (msec) =      5 lines
// V Back Porch      = 0.444;         // (msec) =     20 lines
// V Top Border      = 0.000;         // (msec) =      0 lines
```

图 1.4-4

至于其余的分辨率, 比如 1024*768@60HZ 等, 也可以从里面找到。

代码的第 30 行定义了一个数组, 大小 24bit, 深度是图像的分辨率大小, 在仿真中可以这么去操作, 实际上班综合不建议这样写, 会消耗大量 LUT 资源, 还是用 DRM 资源去生成比较好。

代码的第 34-38 行是通过 for 循环语句将数组初始化为 0。代码的 42-45 行是通过 $readmemh 函数去读取 txt 文件的数据,该语句会把 img.txt 里的数据放到 img_data_reg 里面。读者使用时需要注意文件路径。

代码的 54-72 行完成行场扫描周期的计数, x_cnt 是行周期计数, y_cnt 是场周期计数。

首先看代码第 60 行的判断条件, 当 x_cnt 和 y_cnt 分别到达行场扫描周期的最大值, 本次使用 1280*720 参数, 至于减 1, 是因为我们从 0 开始计数, 所以就是 0-1649 和 0-749,

所以, 就是计数达到行场扫描周期最大值时, 就把 x_cnt 和 y_cnt 清空。第 65 行的条件就是每完成一次行扫描周期就把 x_cnt 清 0, 并让 y_cnt 加 1。其余情况下就让 x_cnt 不断累加, y_cnt 保持不变即可。这两个计数器还是比较简单的。

代码的 77-84 行完成场信号的生成, 根据我们的 VESA 时序标准, 首先要先经过场同步时间, 而代码中是低电平有效, 同步期间 VS 信号保持低电平, 同步完成后 VS 拉高, 直到一帧结束, 根据前文的显示时序可以看到, VS 拉高将在同步之后直到 V_FP 结束后才拉低, 所以条件为(y_cnt>(V_SYNC-1)&& y_cnt<=(V_SYNC+V_BP+IMG_HEIGHT+V_FP-1))。如图 1.4-5 所示, 默认 Border 均为 0, 如果大家还不太理解的话, 请认真看图 1.4-5 的时序。看 VS 信号在哪个区间拉高。

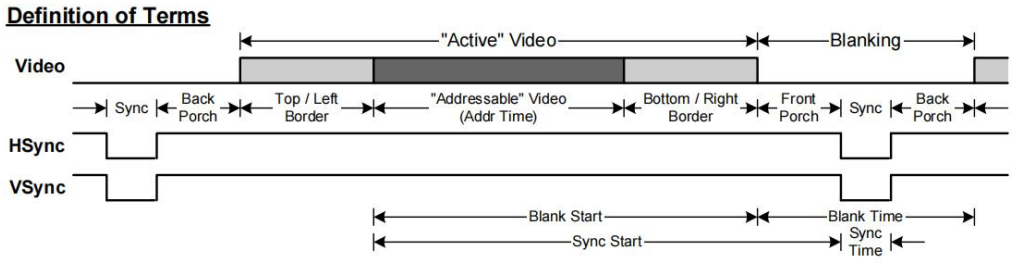


图 1.4-5

代码的 87-94 行完成行信号的生成, 同样的时序如同图 1.4-5 所示, 一样是低电平有效, 所以 HS 在同步期间是低电平, 同步结束后拉高, 直到 H_FP 结束后 HS 信号才拉低。所以条件为(x_cnt>(H_SYNC-1)&& x_cnt<=(H_SYNC+H_BP+IMG_WIDTH+H_FP-1))。

代码的 98 行是图像有效信号的生成即 video_de, 其有效条件其实就在显示后沿结束和显示前沿之前这段区间, 如果大家忘记了, 可以看图 224 中间的淡蓝色方框, 可以明确看到其有效条件。因此只要判断行常计数器均处在显示后沿和显示前沿之间时就是图像有效显示区域。

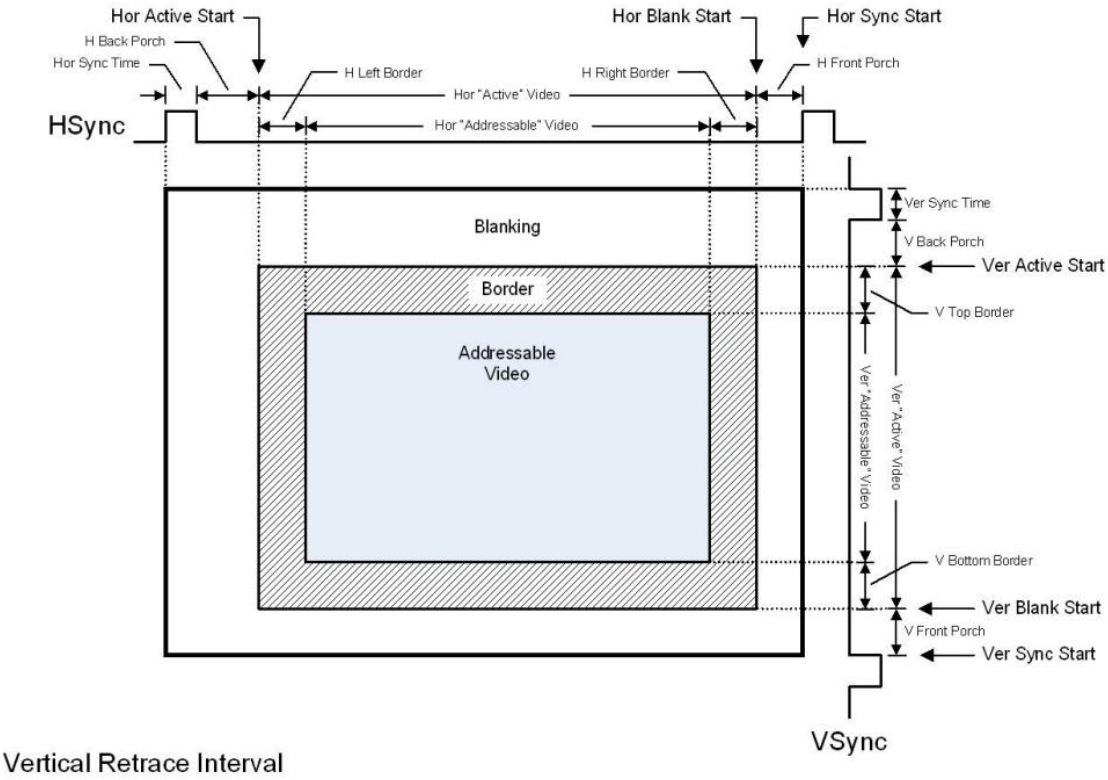


图 1.4-6

代码的 103-112 行完成数组索引的计数,图像数据我们均存在 img_data_reg 这个数组里面, 因此需要一个计数器来充当数组的地址, 然后将里面的每个图像数据读出。当 video_de_d 有效时表示数据有效, 此时让 img_index+1 即可, 计数到一帧即 IMG_WIDTH*IMG_HEIGHT-1 的时候, 就清空,

表示已经读出一帧。其余情况让 img_index 保持不变, 千万不能清空, 因为 video_de_d 并不是一直有效的,到了消隐区间,video_de_d 就会拉低,所以其余情况下不能让 img_index 清 0, 否则会导致计数不正常。

代码的 117-122 行,将 img_data_reg 数组的数据读出存放到 video_data_reg 寄存器里面。

代码的 126-131 行主要是将 video_de_d 延迟一个时钟周期, 因为我们的 video_data_reg 是在 video_de_d 有效时, 通过时序逻辑输出。所以会滞后 video_de_d 一

个时钟周期, 故打一拍让数据有效信号和数据同步。

代码的 133-135 行, 通过组合逻辑把数据有效信号和视频场信号和图像数据输出。

1.4.2.仿真测试

接下来看 Testbench 是如何编写的, 具体代码如下所示:

```
//图像处理仿真模块
`timescale 1ns/1ns
module  img_process_tb();

//图片高度宽度
//仿真需要改小视频大小 避免太大 默认 1280*720

//时序参考模板  仿真中不需要严格按照 vesa 时序标准
parameter IMG_WIDTH = 16'd1280;//有效区域
parameter H_FP = 16'd110;   //前沿
parameter H_SYNC = 16'd40;  //同步
parameter H_BP = 16'd220 ;  //后沿
parameter TOTAL_WIDTH = IMG_WIDTH  + H_FP  +H_SYNC + H_BP;

parameter IMG_HEIGHT = 16'd720; //有效区域
parameter V_FP  = 16'd5;    //前沿
parameter V_SYNC  = 16'd5;  //同步
parameter V_BP  = 16'd20;   //后沿
```

```
parameter TOTAL_HEIGHT = IMG_HEIGHT + V_FP + + V_SYNC + V_BP;



reg video_clk   ;
reg rst_n       ;

wire          video_vs;
wire          video_de;
wire    [23:0]  video_data;

wire          y_vs;
wire          y_hs;
wire          y_de;
wire    [7:0]         y_data;

//定义处理后的文件
integer output_file;
initial
begin
   video_clk   =   1'd0;
   rst_n       =   1'd0;
   #20
```

```
    rst_n      =  1'd1;
    output_file =  $fopen("D:/pango_isp/img_test_pg/img_test_pg/01_led_test/sim/img_process.txt","w");
end
//生成 100MHZ
always#5 video_clk = ~video_clk;



//读取图片数据

video_data_gen#(
    .TOTAL_WIDTH  ( 12'd1650 ),
    .IMG_WIDTH    ( 12'd1280 ),
    .H_SYNC       ( 12'd40 ),
    .H_BP         ( 12'd220 ),
    .H_FP         ( 12'd110 ),
    .TOTAL_HEIGHT ( 12'd750 ),
    .IMG_HEIGHT   ( 12'd720 ),
    .V_SYNC       ( 12'd5 ),
    .V_BP         ( 12'd20 ),
    .V_FP         ( 12'd5 )
)u_video_data_gen(
```

```
    .video_clk   ( video_clk   ),
    .rst_n       ( rst_n       ),
    .video_vs    ( video_vs    ),
    .video_de    ( video_de    ),
    .video_data  ( video_data  )
);



//灰度化

RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n     ),
    .vsync_in  ( video_vs  ),
    .hsync_in  ( video_de  ),
    .de_in    ( video_de    ),
    .red      ( video_data[23:19]   ),
    .green    ( video_data[15:10]   ),
    .blue     ( video_data[7:3]     ),
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out   ( y_de  ),
```

```
    .y      ( y_data ),
    .cb     (      ),
    .cr     (      )
);

GTP_GRS GRS_INST(
    .GRS_N(1'b1)
    ) ;

//写数据
reg   video_vs_d   ; //打拍寄存
reg   img_done     ;
wire  frame_flag   ;


always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d <=  1'd0;
    else
        video_vs_d <=  y_vs;
end
```

```
assign frame_flag = ~y_vs & video_vs_d;    //下降沿


always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end



always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;    //停止仿真
    end
    else if(y_de)    //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",y_data,y_data,y_data);    //16 进制写
入
```

```
    end
end

endmodule
```

代码的 9-19 行定义了视频显示的相关参数, 默认是 1280*720, 可以通过修改这部分去修改模拟生成的视频流的分辨率,需要注意的是使用的图片,以及 Matlab 里的图片分辨率的参数均要保持一致。

代码的 35-43 行主要完成部分信号的初始化以及创建一个 txt 文件, 可以通过 integer 声明一个变量 output_file。使用$fopen 函数可以访问 txt 文件, 如果没有对应的 txt 文件, 会自动创建一个 txt 文件, 最后的参数给" w" 表示写入。rst_n 一开始给 0, 延时 20ns 后将其拉高, 表示复位结束。

代码的 45 行生成了像素时钟,每隔 5ns 翻转一次,因此是 100MHZ 的时钟。因此我们产生的视频流帧率其实是 100MHZ/1650/750≈80.8HZ。如果生成 74.25MHZ 的时钟, 那就是 60HZ。

代码的 50-67 行例化了 video_data_gen 模块。用来提供图像数据。

代码的 72-87 行例化了灰度化的模块, 完成对图像的灰度化处理, 这里暂时不讲其具体内容, 其代码讲解在第十一章会提到, 这里先不仔细讲解。

代码的 89-91 行使用了 GTP_GRS 的原语, 当用到 FIFO IP 和 RAM/ROM IP 时, 需要添加该语句。

代码的 99-104 行对处理后的场信号 VS 打一拍寄存, 用来获取下降沿。

代码的 106 行通过 video_vs_d&~y_vs 来获取场信号的下降沿,因为我们的代码里时 vs 高电平时图像数据有效, 所以一帧的结束是在 vs 拉低的时候, 也就是下降沿。所以这里的下

降沿表示一帧完成。即 frame_flag 表示一帧完成, 该信号持续一个时钟周期。

代码的 108-115 行用 img_done 寄存一帧完成信号, 当 frame_flag 拉高, 则将 img_done 拉高并保持不变, 指示已经完成了一帧图像的处理。

代码的 118-125 行当 img_done 拉高, 表示一帧已经完成, 此时停止仿真, 故该仿真实际只完成了一帧的处理,如果读者想持续处理,只需要去除这里的$stop 即可。如果 img_done 未拉高, 则表示一帧还未结束, 此时根据 y_de(灰度化后的图像有效信号), 如果数据有效就通过$fdisplay 函数将图像数据 y_data 写入到 txt 文件里面, 写入的格式为 16 进制格式, RGB888 格式, 即 24bit, 由于灰度数据只有 8bit, 所以复制 3 份即可。/t 表示四个空格。即在 txt 中, 两个数据之间间隔四个空格。

至此, 图像的仿真平台搭建就此结束。接下来看实际的仿真效果。

1.4.2.1. 仿真结果

打开 PDS 工程, 执行联合仿真, run100ms, 当仿真结束时, 会弹出以下提示。

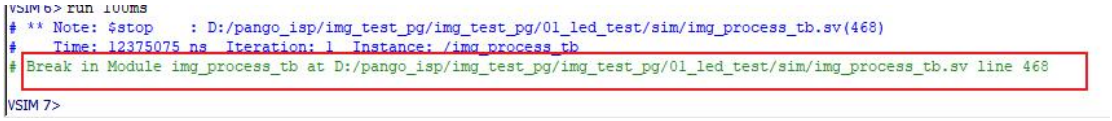


图 1.4-7

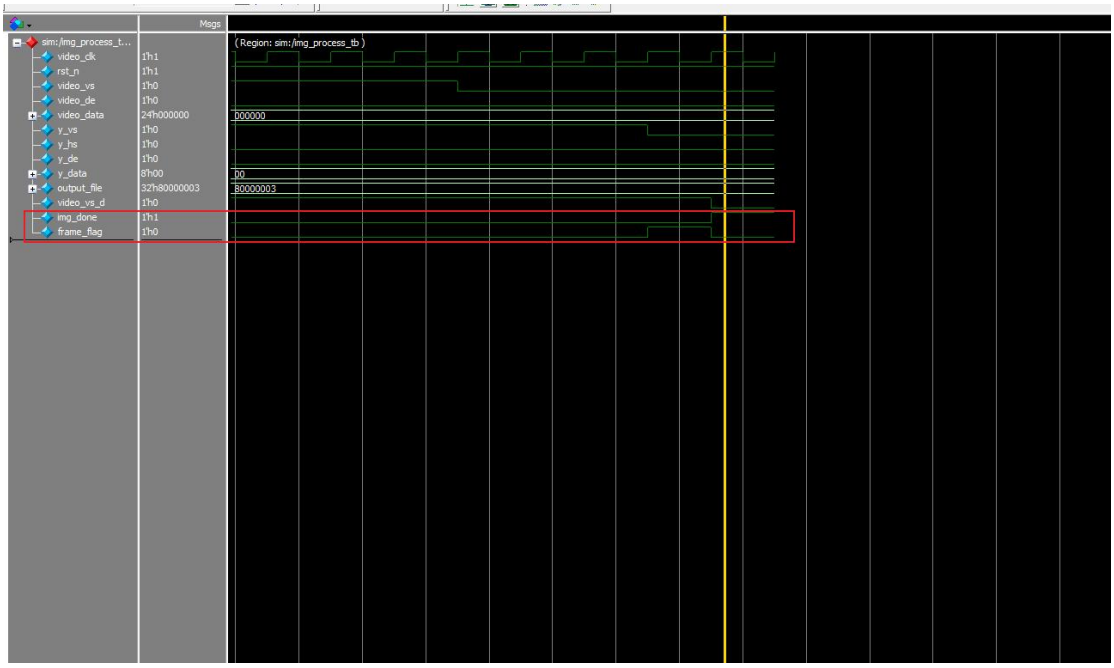


图 1.4-8

打开波形界面观察各个信号的波形, 具体可以看视频讲解, 这里主要看 img_done 信号。可以看到, 当 frame_flag 拉高时, 正好是 y_vs 信号从高变低的时候, 即获取了下降沿, 下个时钟周期, img_done 拉高, 一帧完成, 仿真结束。

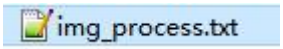


图 1.4-9

之后打开文件夹可以看到有个 img_process.txt 文件生成, 读者需要注意一下文件的生成路径。

之后打开 Matlab 工程, 运行脚本。将 txt 读出, 转为图片形式, 运行 fpga_hex2img.m 文件。

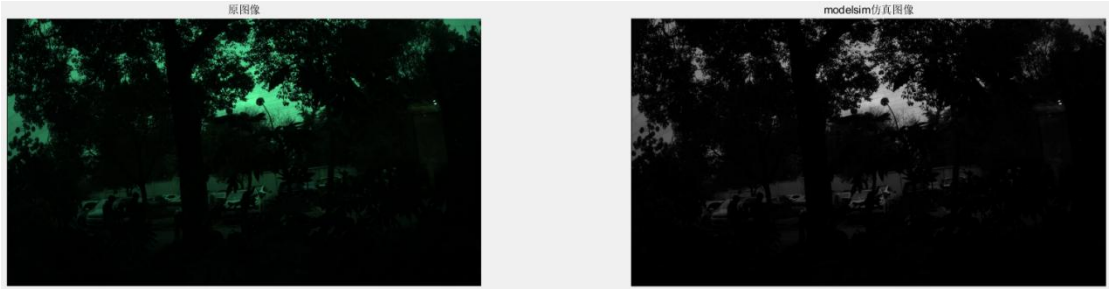


图 1.4-10

左边为原图像, 右边为 Modelsim 仿真处理后输出的图像。

2. 灰度化

2.1. 实验简介

实验目的:

完成图像 RGB 到 YUV 的转换,输出灰度图像.

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

2.2. 实验原理

2.2.1.图像格式介绍

1、RGB 格式:RGB 色彩就是常说的光学三原色, R 代表 Red (红色), G 代表 Green (绿色), B 代表 Blue (蓝色),自然界中肉眼所能看到的任何色彩都可以由这三种色彩混合叠加而成, 因此也称为加色模式。我们常用的 RGB 格式有 RGB565, RGB888。其中 RGB565:每个像素用 16bit 来表示, 占 2 个字节, 第一字节的前 5 位是 R, 后三位+第二字节前三位是 G, 第二字节后 5 位是 B。RGB888:每个像素用 24bit 来表示, 占 3 个字节, R、G、B、各占 8bit。还有

一种是 RGB32 格式, 在 RGB888 的基础上增加了透明度 Alpha, 用 A 表示。

2、YUV(YCBCR)格式: 常用于描述影像的饱和度和色调, 也可以简化图像信息、去除冗余信息、提升计算效率。RGB 与 YUV 的转换实际上是色彩空间的转换, 即将 RGB 的三原色色彩空间转换为 YUV 所表示的亮度与色度的色彩空间模型。Y 表示亮度、U 和 V 表示色度。而 YCbCr 是通过 YUV 信号的发展而来的, 一般来说。这两种信号是用在不同的场景, 但目前在一般情况下, 两者是等价的。Y 还是表示亮度、Cb 表示蓝色分量、Cr 表示红色分量。一般有 YUV4:4:4,YUV4:2:2。其表示的含义就是各个分量采样率的不同。比如 YUV4:2:2, Y 的采样率是 U 和 V 的两倍。通过显示 Y 分量即可得到灰度图。

总结: RGB 着重于人眼对色彩的感应, YUV 则着重于视觉对于亮度的敏感程度。同时灰度图像只包含了亮度信息其大小在 0-255 之间, 在某些情况下, 可以减少计算所需的数据量, 提高计算效率。除此之外还有很多别的格式, 比如还有 RAW 格式显示的图像。

2.2.2.计算公式介绍

常用的 RGB 转 YUV 的公式如下所示:

$$\mathrm{Y} = 0.299\mathrm{R} + 0.587\mathrm{G} + 0.114\mathrm{B}$$
$$\mathrm{Cb} = -0.172\mathrm{R} - 0.339\mathrm{G} + 0.511\mathrm{B} + 128$$
$$\mathrm{Cr} = 0.511\mathrm{R} - 0.428\mathrm{G} - 0.083\mathrm{B} + 128$$

最后我们只需要显示 Y 分量, 即可得到灰度图, 但是我们可以发现, 该公式涉及到大量的小数运算。在 Verilog 中, 我们是不可能直接定义一个值为 0.299 的常量的。因此我们需要将将小数转为化整数。可以通过将公式整体扩大 256 倍, 例如 0.299*256≈77。这样做会损失部分精度, 要求高精度的话可以扩大更大的倍数, 倍数越大, 结果越准确。所以 0.299R 就转为 77R, 最后再把计算结果>>8 位即可, 相当于除以 256, 我们通过移位操作可以大大降低运算难度。

具体公式如下所示:

$$Y = (77R + 150G + 29B) >> 8$$
$$Cb = (-43R - 85G + 128B + 32768) >> 8$$
$$Cr = (128R - 107G - 21B + 32768) >> 8$$

该公式避免了小数也避免了除法, 大大降低了运算的难度。

后续笔者会先通过 Matlab 完成图像的灰度化, 先通过 Matlab 自带的灰度化函数来完成图像灰度化, 再利用上述公式完成图像灰度化, 然后再通过 Moldelsim 仿真得到灰度化后的 txt 数据, Matlab 再将其恢复为图片显示, 最后对比灰度化的结果是否准确。

2.3. 接口列表

工程顶层模块, 后面的工程均是这个模块为顶层, 故后面的章节不再列出。

以下为 RGB2YCbCr.v 模块的接口列表。

端口	I/O	位宽	描述
clk	input	1	像素时钟
rst_n	input	1	系统复位
vsync_in	input	1	处理前的视频场信号
hsync_in	input	1	处理前的视频行信号
de_in	input	1	数据有效信号
red	input	5	红色分量
greed	input	6	绿色分量
blue	input	5	蓝色分量
vsync_out	output	1	处理后的视频场信号

hsync_out	output	1	处理后的视频场信号
de_out	output	1	处理后的数据有效信号
y	output	8	亮度数据(灰度数据)
cb	output	8	蓝色色度数据(预留)
cr	output	8	红色色度数据(预留)

2.4. 工程说明

图 2.4-1

该图为本次工程框架, 通过 HDMI 输入, 经过 MS7000 芯片解码出 RGB 数据, 然后通过灰度化模块输出 Y 分量, 然后再经过 DDR3 缓存, 最后读出到屏幕上显示。显示的分辨率为 1080P@60HZ。

2.4.1.代码模块说明

主要介绍灰度化模块, 具体如下所示:

```
//延迟 3 个 clk
module RGB2YCbCr
```

```
(
  //module clock
  input          clk          ,
  input          rst_n        ,


  input          vsync_in ,  // vsync 信号
  input          hsync_in ,  // hsync 信号
  input          de_in    ,  //
  input    [4:0] red      ,
  input    [5:0] green    ,
  input    [4:0] blue     ,


  output         vsync_out,  // vsync 信号
  output         hsync_out,  // hsync 信号
  output         de_out   ,  // data enable 信号
  output   [7:0] y        ,
  output   [7:0] cb       ,
  output   [7:0] cr
);
//reg define
```

```
reg  [15:0]   rgb_r_m0, rgb_r_m1, rgb_r_m2;
reg  [15:0]   rgb_g_m0, rgb_g_m1, rgb_g_m2;
reg  [15:0]   rgb_b_m0, rgb_b_m1, rgb_b_m2;
reg  [15:0]   y0 ;
reg  [15:0]   cb0;
reg  [15:0]   cr0;
reg  [ 7:0]   y1 ;
reg  [ 7:0]   cb1;
reg  [ 7:0]   cr1;
reg  [ 2:0]   vsync_in_d;
reg  [ 2:0]   hsync_in_d;
reg  [ 2:0]   de_in_d   ;

//wire define
wire [ 7:0]   rgb888_r;
wire [ 7:0]   rgb888_g;
wire [ 7:0]   rgb888_b;


//RGB565 to RGB 888
assign rgb888_r       = {red  , red[4:2]  };    // 3'd0  red[4:2] 提高图像的亮度
assign rgb888_g       = {green, green[5:4]};
```

```
assign rgb888_b        = {blue , blue[4:2] };



//输出
assign vsync_out = vsync_in_d[2]     ;
assign hsync_out = hsync_in_d[2]     ;
assign de_out    = de_in_d[2]        ;
assign y          = hsync_out ? y1 : 8'd0;
assign cb          = hsync_out ? cb1: 8'd0;
assign cr          = hsync_out ? cr1: 8'd0;


//------------------------------------------
//RGB 888 to YCbCr
/*******************************************************
        RGB888 to YCbCr
 Y  = 0.299R +0.587G + 0.114B
 Cb = 0.568(B-Y) + 128 = -0.172R-0.339G + 0.511B + 128
 CR = 0.713(R-Y) + 128 = 0.511R-0.428G -0.083B + 128

 Y  = (77 *R    +    150*G    +    29 *B)>>8
 Cb = (-43*R    -    85 *G    +    128*B)>>8 + 128
 Cr = (128*R    -    107*G    -    21 *B)>>8 + 128
```

```
Y  = (77 *R   +   150*G   +   29 *B      )>>8
Cb = (-43*R   -   85 *G   +   128*B + 32768)>>8
Cr = (128*R   -   107*G   -   21 *B + 32768)>>8
*******************************************************/

//step1 pipeline mult
//delay :1 clk
always @(posedge clk or negedge rst_n) begin
  if(!rst_n) begin
     rgb_r_m0 <= 16'd0;
     rgb_r_m1 <= 16'd0;
     rgb_r_m2 <= 16'd0;
     rgb_g_m0 <= 16'd0;
     rgb_g_m1 <= 16'd0;
     rgb_g_m2 <= 16'd0;
     rgb_b_m0 <= 16'd0;
     rgb_b_m1 <= 16'd0;
     rgb_b_m2 <= 16'd0;
  end
  else begin
     rgb_r_m0 <= rgb888_r * 8'd77 ;
```

```
        rgb_r_m1 <= rgb888_r * 8'd43 ;
        rgb_r_m2 <= rgb888_r << 3'd7 ;
        rgb_g_m0 <= rgb888_g * 8'd150;
        rgb_g_m1 <= rgb888_g * 8'd85 ;
        rgb_g_m2 <= rgb888_g * 8'd107;
        rgb_b_m0 <= rgb888_b * 8'd29 ;
        rgb_b_m1 <= rgb888_b << 3'd7 ;
        rgb_b_m2 <= rgb888_b * 8'd21 ;
    end
end

//step2 pipeline add
//delay 1 clk
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        y0  <= 16'd0;
        cb0 <= 16'd0;
        cr0 <= 16'd0;
    end
    else begin
        y0  <= rgb_r_m0 + rgb_g_m0 + rgb_b_m0;
        cb0 <= rgb_b_m1 - rgb_r_m1 - rgb_g_m1 + 16'd32768;
```

```
        cr0 <= rgb_r_m2 - rgb_g_m2 - rgb_b_m2 + 16'd32768;
    end

end

//step3 pipeline div
//delay  1 clk
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        y1  <= 8'd0;
        cb1 <= 8'd0;
        cr1 <= 8'd0;
    end
    else begin
        y1  <= y0 [15:8];//y0>>8
        cb1 <= cb0[15:8];
        cr1 <= cr0[15:8];
    end
end

//总共延迟 3 个时钟周期
//n 个时钟周期
```

```
// reg    [n-1:0]    data_sasadasda;
// reg    data_sasadasda <= {data_sasadasda[n-2:0], dsadada};
// assign sa = data_sasadasda[n-1];  data_sasadasda[1]   data_sasadasda[2]
always@(posedge clk or negedge rst_n) begin
   if(!rst_n) begin
      vsync_in_d <= 3'd0;
      hsync_in_d <= 3'd0;
      de_in_d    <= 3'd0;
   end
   else begin
      vsync_in_d <= {vsync_in_d[1:0], vsync_in};
      hsync_in_d <= {hsync_in_d[1:0], hsync_in};
      de_in_d    <= {de_in_d[1:0]   , de_in   };
   end
end


endmodule
```

公式其实是非常简单的,部分人可能会直接 assign y = (77*R+150*G+29*B)>>8 来实现,这种写法乍一看好像没问题, 实际上会造成非常大的延迟, 因为这个组合逻辑需要计算三个乘法最后相加再移位, 延迟是非常大的, 非常容易造成时序违例, 导致工作时钟频率降低。所以我们通过流水线的处理方式来改善该路径的延时, 在中间穿插了三级流水线, 分成三个 clk 来计算得出灰度化结果, 可以有效避免时序违例。接下来开始分析代码, 看看是如何完成的,

后续的算法都将是流水线思想来完成。

代码的 45-47 行将输入的 RGB565 转为 RGB888 格式。因为, 公式是基于 RGB888 的。

代码的 77-100 行是第一级流水线, 可以看到该 always 块的主要内容是完成乘法的计算, 把公式里的所有乘法用一个时钟周期来计算得到, 因此在数据有效的第一个时钟周期里, 会先得到乘法的结果。如果是乘 128, 可以用<<7 来代替, 其余正常的用*来实现, 也可以用乘法器来实现。还有要注意的是, 因为整体都扩大了 256 倍, 所以乘法得到的结果会扩大 256 倍, 也就是<<8, 位宽会扩大 8 为, 所以定义寄存器的时候应该定义为[15:0], 避免溢出。

代码的 104-116 行是第二级流水线, 该 always 块主要是完成加减法的内容, 将每个乘法的结果全部相加, 同样的, 为了避免溢出, 寄存器的位宽也定义为 16bit 即可。实际上为了避免减法得到负数, cb0 和 cr0 的计算应该判断系数之间的大小再进行减法, 避免得到负数造成结果不正常, 读者需要注意涉及到减法时, 需要避免负数, 因为我们的计算通常都是无符号的, 出现负数会导致结果异常。

代码的 120-131 行是第三级流水线, 也就是最后一级流水线, 完成除以 256 的操作, 通过>>8 来代替, [15:8]和>>8 是等价的。

2.1.3.2 代码仿真

2.4.1.1. Matlab 仿真介绍

```
clear all; clc;
% 读取图片
image = imread('testst.png');

%% 图像输入提取三通道 转 double 扩大位宽
R = double(image(:,:,1));
```

```
G = double(image(:,:,2));
B = double(image(:,:,3));
%% 将图片转换为灰度图像
grayImage = rgb2gray(image);
%% 根据公式转灰度 >>8 等价 /256
 %Y  = (77 *R    +    150*G    +    29 *B       )>>8
 %Cb = (-43*R    -    85 *G    +    128*B + 32768)>>8
 %Cr = (128*R    -    107*G    -    21 *B + 32768)>>8
Y = (77 *R+150*G+29 *B)/256.0;
Cb = (-43*R-85 *G+128*B+32768)/256.0;
Cr = (128*R-107*G-21 *B+32768)/256.0;
%% 显示部分
% 显示原始图片
subplot(1, 3, 1);
imshow(image);
title('原始图片');

% 显示灰度图像
subplot(1, 3, 2);
imshow(grayImage);
title('matlab 自带函数灰度图片');
imwrite(grayImage,'gray.png');
```

```
% 显示灰度图像
subplot(1, 3, 3);
imshow(uint8(Y));
title('公式灰度化图片');
imwrite(uint8(Y),'ngray.png');
%比较差异
diff = uint8(Y) - uint8(grayImage);
```

代码的第 3 行使用 imread()函数读取了一张图片。

代码的 6-8 行单独提取了该图片的 R、G、B 三通道，并转为 double 类型，扩大位宽。

代码的第 10 行是使用 Matlab 自带的函数 rgb2gray()完成图像的灰度化，与后面我们使用公式转化来进行对比。

代码的 15-17 行通过公式来完成图像的灰度化。

代码的 20-35 行完成图像的显示，利用 subplot()开多个窗口，subplot(1,3,1)表示一行三列，然后选择第一个窗口，imshow()是显示图片。最后将原图和 matlab 自带的灰度图以及利用公式计算得出的灰度图一起显示出来，方便对比。

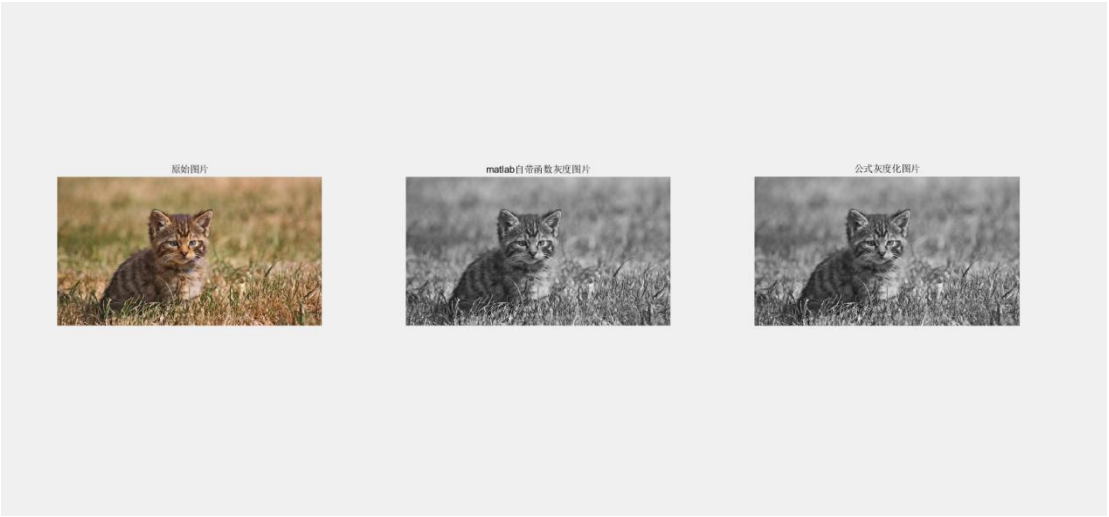
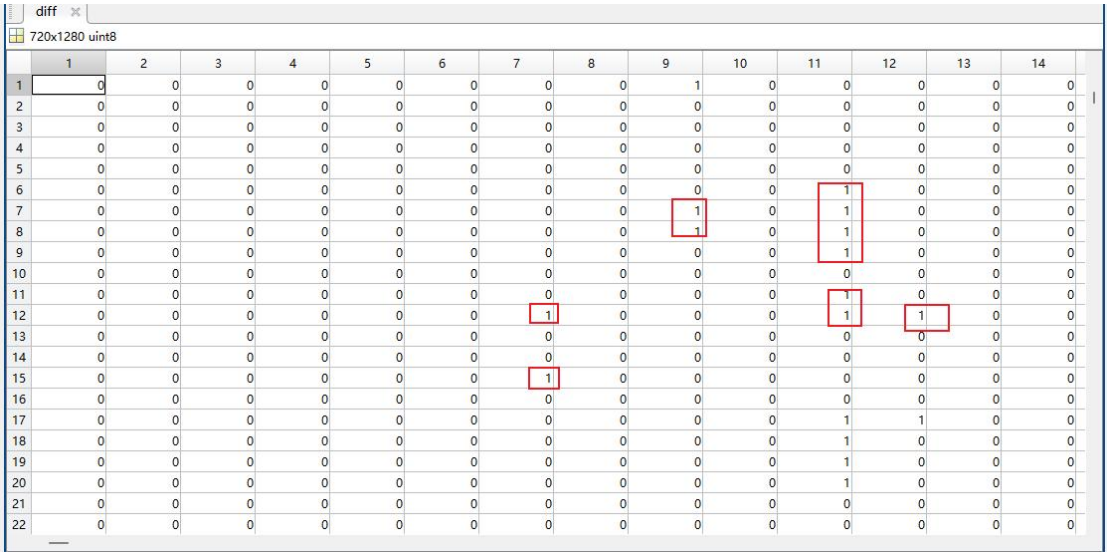


图 2.4-2

最后的 diff 是两张图片相减, 会将对应的每个像素点做差, 用来看两者之间的差异。



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	1	0	1	0	0	0
8	0	0	0	0	0	0	0	0	1	0	1	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	1	0	0	0
12	0	0	0	0	0	0	1	0	0	0	1	1	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	1	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	1	1	0	0
18	0	0	0	0	0	0	0	0	0	0	1	0	0	0
19	0	0	0	0	0	0	0	0	0	0	1	0	0	0
20	0	0	0	0	0	0	0	0	0	0	1	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 2.4-3

查看该变量, 可以看到基本是 0, 偶尔会有 1 出现, 可以认为没有误差。

2.4.1.2. Modelsim 仿真介绍

Moldeism 的仿真框架和第 10 节的平台一模一样, 因此这里不再做介绍, 灰度化模块也在 2.3.3.1 中解释, 剩下的步骤与第十节一致。

2.4.2.实验现象

连接好下载器, 电源、HDMI_IN 口、HDMD_OUT 口, 然后下载程序。

图 2.4-5

上图为测试原图像。

图 2.4-6

上图为灰度化后的图像。

1.1.3X3 图像矩阵生成

2.5. 实验简介

实验目的:

实现 3X3 图像矩阵对应 9 个像素点图像数据的读取

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

2.6. 实验原理

在数字图像处理中, 部分算法需根据每个像素点及其周围一定范围内的像素点数据进行计算, 图像矩阵的生成至关重要。本系列图像处理实验主要采用 3X3 图像矩阵实现对应图像处理算法。

例如一幅完整的 5X5 图像如下图所示, 每个方格表示一个像素点, 方格内的数字表示对应像素点的像素数据。

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

图 2.3-1

3X3 图像矩阵的行对应图像一行中三个连续的像素点, 3X3 图像矩阵的列对应图像一列中三个连续的像素点,矩阵的元素对应图像的像素数据。在 5X5 的图像中,按图像的扫描方式从左到右, 从上到下地获取对应 3X3 图像矩阵的数据

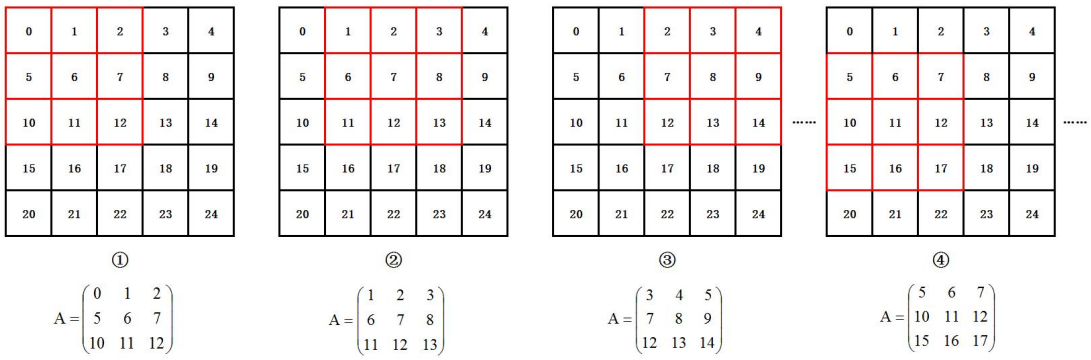


图 2.3-2

在图像处理算法中, 若通过每个 3X3 图像矩阵计算中心像素点的像素数据, 为保证图像分辨率不变, 在图像左侧两列和上方两行的位置对像素数据补 “0” 操作, 如下图所示:

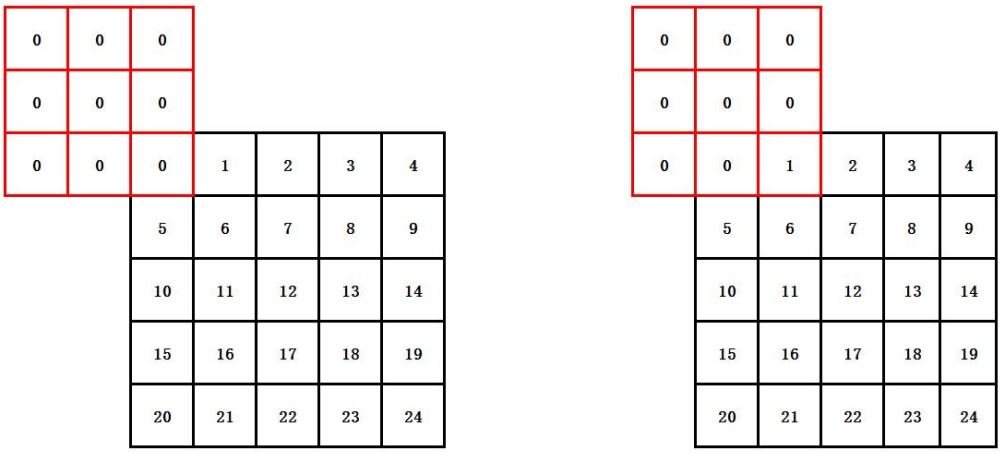


图 2.3-3

基于以上分析, 在图像数据按行输入的情况下, 为获取 3X3 图像矩阵需同时输出三行图像中的数据, 因此采用 FPGA 的缓存资源 (fifo/ram)将接收的每一行数据进行缓存, 并且在

输出 3X3 图像矩阵数据时输出当前第 n 行对应位置的数据、第 n-1 行对应位置的数据以及第 n-2 行对应位置的数据。另外, 为缓存第 n-1 行图像数据和第 n-2 行图像数据, 需调用 2 个 fifo 缓存模块, 并且实现当前数据缓存到第 n-1 行图像数据缓存 fifo, 从第 n-1 行图像数据缓存 fifo 读出后再缓存到第 n-2 行图像数据缓存 fifo, 确保 3X3 矩阵输出时读取两个 fifo 可同时获得第 n-1 行和第 n-2 行的数据。FPGA 实现 3X3 图像矩阵具体方案框图如下:

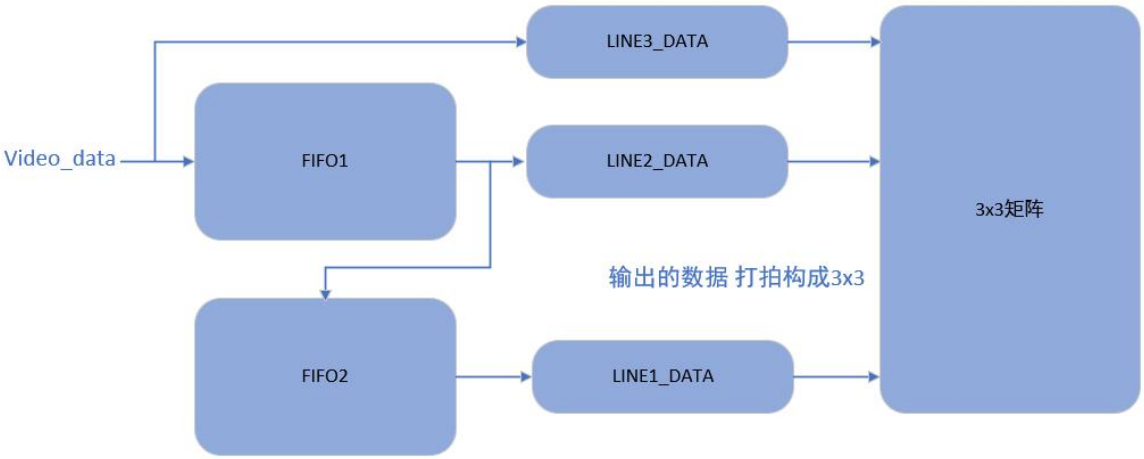


图 2.3-4

按行输入的图像数据 video_data 在数据有效期间缓存到 fifo1, 同时 fifo1 读出一行数据进入 fifo2 缓存, 且 fifo2 同时读出一行数据, 若 fifo1 或 fifo2 为空则输出 0。数据输出过程如下图所示:

video_data输入	line1	line2	line3	line4	……	line(n)
fifo1输出	0	line1	line2	line3	……	line(n-1)
fifo2输出	0	0	line1	line2	……	line(n-2)

图 2.3-5

此外, 实现同时输出连续三行图像数据后, 需注意所输出的三个数据对应在一行中的位置保持一致。

2.7. 接口列表

以下为 matrix_3x3.v 模块的接口列表。

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
video_vs	input	1	视频场同步信号
video_de	input	1	视频行有效信号
video_data	input	8	视频图像灰度数据
matrix_de	output	1	矩阵输出有效信号
matrix11	output	8	3X3 图像矩阵 a11 元素数据
matrix12	output	8	3X3 图像矩阵 a12 元素数据
matrix13	output	8	3X3 图像矩阵 a13 元素数据
matrix21	output	8	3X3 图像矩阵 a21 元素数据
matrix22	output	8	3X3 图像矩阵 a22 元素数据
matrix23	output	8	3X3 图像矩阵 a23 元素数据
matrix31	output	8	3X3 图像矩阵 a31 元素数据
matrix32	output	8	3X3 图像矩阵 a32 元素数据

matrix33	output	8	3X3 图像矩阵 a33 元素数据

2.8. 工程说明

2.8.1.代码模块说明

3X3 图像矩阵生成模块代码如下所示:

```
//3x3 矩阵生成
module  matrix_3x3
#(
  parameter  IMG_WIDTH  =  11'd1920   ,
  parameter  IMG_HEIGHT =  11'd1080
)
(
  input  wire        video_clk    ,
  input  wire        rst_n        ,

  //
  input  wire        video_vs     ,
  input  wire        video_de     ,
  input  wire  [7:0] video_data   ,

  //3x3 矩阵输出
```

```
    output  wire        matrix_de    ,
    output  reg   [7:0] matrix11     ,
    output  reg   [7:0] matrix12     ,
    output  reg   [7:0] matrix13     ,

    output  reg   [7:0] matrix21     ,
    output  reg   [7:0] matrix22     ,
    output  reg   [7:0] matrix23     ,

    output  reg   [7:0] matrix31     ,
    output  reg   [7:0] matrix32     ,
    output  reg   [7:0] matrix33
);

//wire define
wire   [7:0]  line3_data; //第三行数据
wire   [7:0]  line2_data; //第二行数据
wire   [7:0]  line1_data; //第一行数据

wire          wr_fifo_en; //写 FIFO 使能
wire          rd_fifo_en; //读 FIFO 使能
//reg define
```

```
reg [10:0]  x_cnt;
reg [10:0]  y_cnt;

//列计数
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    x_cnt  <=  11'd0;
  else  if(x_cnt == IMG_WIDTH - 1)//计数一行
    x_cnt  <=  11'd0;
  else  if(video_de)  //数据有效
    x_cnt  <=  x_cnt + 1'b1;
  else
    x_cnt  <=  x_cnt;
end

//行计数
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    y_cnt  <=  11'd0;
  else  if(y_cnt == IMG_HEIGHT - 1 && x_cnt == IMG_WIDTH - 1)//计数一帧
    y_cnt  <=  11'd0;
  else  if(x_cnt == IMG_WIDTH - 1)  //数据有效
```

```
    y_cnt  <=  y_cnt + 1'b1;
  else
    y_cnt  <=  y_cnt;
end
```

以上代码根据模块接口列表, 定义模块端口及对输入数据进行行/列计数

```
    //3x3 矩阵  第一行和最后一行无法构成
    assign wr_fifo_en = video_de && (y_cnt < IMG_HEIGHT-1);
    assign rd_fifo_en = video_de && (y_cnt > 0);

    reg wr_fifo_en_1d;
    reg [7:0]video_data_1d;

    always@(posedge video_clk or negedge rst_n) begin
        if(!rst_n)
        begin
            wr_fifo_en_1d    <=    1'd0;
            video_data_1d    <=    8'd0;
        end
        else
        begin
            wr_fifo_en_1d <= wr_fifo_en;
            video_data_1d <= video_data;
        end
    end

    //通过两个 FIFO 与当前的输入一起构成 3x3 矩阵
    assign line3_data = video_data_1d;

    //fifo1
    fifo_line_buffer u1_fifo_line_buffer (
      .wr_clk(video_clk),                // input
      .wr_rst(~rst_n),                // input
      .wr_en(wr_fifo_en),                  // input
      .wr_data(video_data),              // input [7:0]
```

```
    .wr_full(),              // output
     .almost_full(),      // output

    .rd_clk(video_clk),                // input
    .rd_rst(~rst_n),                // input
     .rd_en(rd_fifo_en),                   // input
    .rd_data(line2_data),              // output [7:0]
    .rd_empty(u1_empty),            // output
    .almost_empty()     // output
  );
  //fifo2
  fifo_line_buffer u2_fifo_line_buffer (
    .wr_clk(video_clk),                // input
    .wr_rst(~rst_n),                // input
    .wr_en(wr_fifo_en_1d),                   // input
    .wr_data(line2_data),              // input [7:0]
    .wr_full(),              // output
    .almost_full(),      // output

    .rd_clk(video_clk),                // input
    .rd_rst(~rst_n),                // input
    .rd_en(rd_fifo_en),                   // input
    .rd_data(line1_data),              // output [7:0]
    .rd_empty(),            // output
    .almost_empty()     // output
  );
```

以上代码实现两级 fifo 对图像数据按行缓存。为确保三行数据输出保持同步,需注意 fifo 读出数据相较于 fifo 读使能信号延时 1 个时钟周期, 因此 line3_data 数据需将当前输入数据延迟 1 个时钟周期,与 fifo 输出的 line2_data 和 line1_data 保持同步,并且 fifo1 输出数据写入 fifo2 时也需要将 fifo1 的读使能信号延时 1 个时钟周期再作为 fifo2 的写使能信号。

```
//数据延迟 de 延迟

reg video_de_d0;

reg video_de_d1;

always@(posedge video_clk or negedge rst_n) begin

  if(!rst_n)
```

```
    begin
        video_de_d0 <=  1'd0;
        video_de_d1 <=  1'd0;
    end
    else
    begin
        video_de_d0 <=  video_de;
        video_de_d1 <=  video_de_d0;
    end
end
//矩阵数据生成
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        {matrix11, matrix12, matrix13} <= 24'd0;
        {matrix21, matrix22, matrix23} <= 24'd0;
        {matrix31, matrix32, matrix33} <= 24'd0;
    end
    else  if(video_de_d0)
    begin
        {matrix11, matrix12, matrix13} <= {matrix12, matrix13, line1_data};
        {matrix21, matrix22, matrix23} <= {matrix22, matrix23, line2_data};
```

```
        {matrix31, matrix32, matrix33} <= {matrix32, matrix33, line3_data};
    end
    else
    begin
        {matrix11, matrix12, matrix13} <= 24'd0;
        {matrix21, matrix22, matrix23} <= 24'd0;
        {matrix31, matrix32, matrix33} <= 24'd0;
    end
end


    //矩阵数据有效输出
    assign matrix_de = video_de_d1;
    assign matrix_vs = video_vs;
    endmodule
```

以上代码实现 3X3 图像矩阵的生成。三行数据同时输出，每一行的数据按“打拍”方式移位实现矩阵的位置的移动，需注意输出数据在生成矩阵时产生 1 个时钟周期的延时，因此最终数据的矩阵有效信号 matrix_de 也需延时 1 个时钟周期。

2.8.2.代码仿真

在 modelsim 平台，matrix_3x3 模块的仿真激励模块 matrix_tb.v 生成 5x5 的图像数据源，仿真分析结果如下：

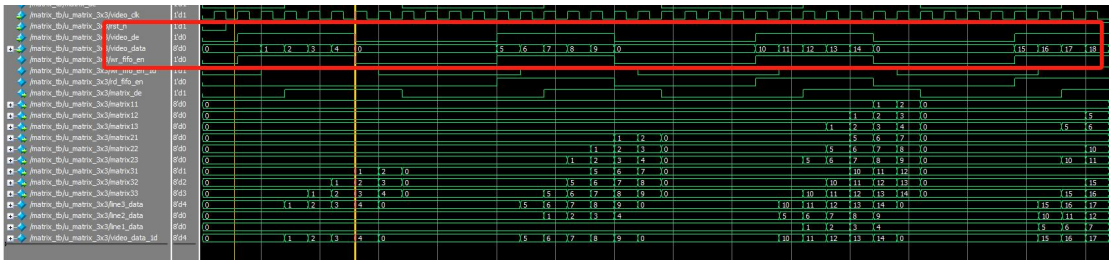


图 2.5-1

5x5 的图像数据源符合以下图像示意图:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

图 2.5-2

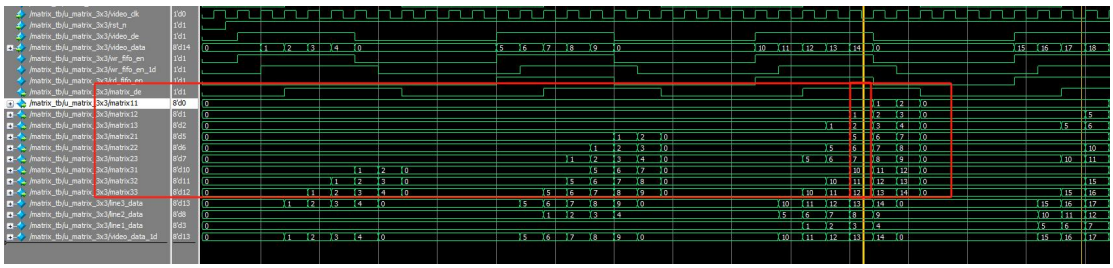


图 2.5-3

3X3 图像矩阵的生成符合预期。

3. 局部二值化/全局二值化

3.1. 实验简介

实验目的:

实现图像的全局二值化处理和局部二值化处理

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

3.2. 实验原理

图像二值化, 将图像的每个像素点的灰度值 (8bit) 设置为 0 或 255, 使图像呈现出明显的只有黑和白的视觉效果。图像二值化将数据简化, 许多视觉算法都依赖二值图像。对于一张图像, 如果不关注色彩以及图像的细节内容, 而是关注其轮廓、边缘, 或者识别某一特征, 比如在一颗蓝色小球和一颗红色小球识别红色小球, 可以通过二值化处理进行区分。本实验中图像二值化采用两种方法: 全局二值化和局部二值化。

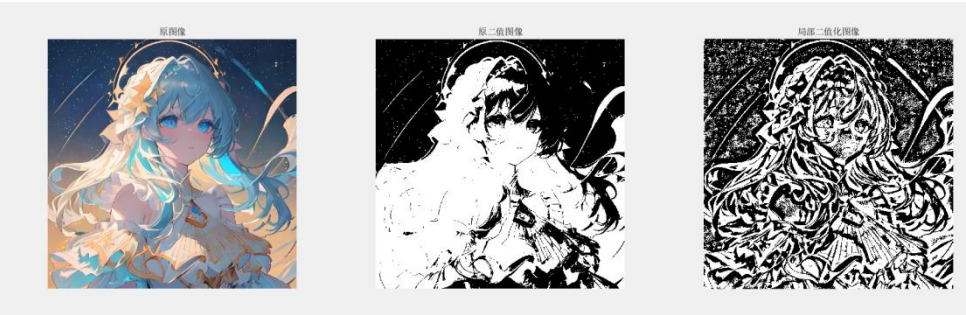


图 3.2-1

全局二值化采用一个固定阈值, 若对应像素点的灰度值(8bit)大于此阈值则设置为 255, 若小于此阈值则设置为 0, 是最简单的一个二值化方法, 但实现效果较差。

局部二值化通过生成 3X3 图像矩阵, 对每个 3x3 窗口里的灰度值数据求均值, 将均值作为阈值, 若 3X3 图像矩阵中心像素点的灰度值大于该阈值则将灰度值设置为 255, 否则设置为 0, 局部二值化能更好的提取边缘信息。

2.1.2 接口列表

以下为全局二值化处理 binarization.v 模块的接口列表:

端口	I/O	位宽	描述

clk	input	1	像素时钟
rst_n	input	1	系统复位
vsync_in	input	1	输入场同步信号
hsync_in	input	1	输入行同步信号
de_in	input	1	输入行有效信号
y_in	input	8	视频图像灰度数据
vsync_out	output	8	输出场同步信号
hsync_out	output	8	输出行同步信号
de_out	output	8	输出行有效信号
pix	output	8	二值化处理结果

以下为全局二值化处理 area_bin.v 模块的接口列表:

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
matrix_de	input	8	矩阵输入有效信号
matrix_vs	input	1	矩阵输入场同步信号
matrix11	input	8	3X3 图像矩阵 a11 元素数据

matrix12	input	8	3X3 图像矩阵 a12 元素数据
matrix13	input	8	3X3 图像矩阵 a13 元素数据
matrix21	input	8	3X3 图像矩阵 a21 元素数据
matrix22	input	8	3X3 图像矩阵 a22 元素数据
matrix23	input	8	3X3 图像矩阵 a23 元素数据
matrix31	input	8	3X3 图像矩阵 a31 元素数据
matrix32	input	8	3X3 图像矩阵 a32 元素数据
matrix33	input	8	3X3 图像矩阵 a33 元素数据
area_bin_vs	output	1	局部二值化输出场同步信号
area_bin_de	output	1	局部二值化输出行有效信号
area_bin_data	output	1	局部二值化输出结果

2.1.3 工程说明

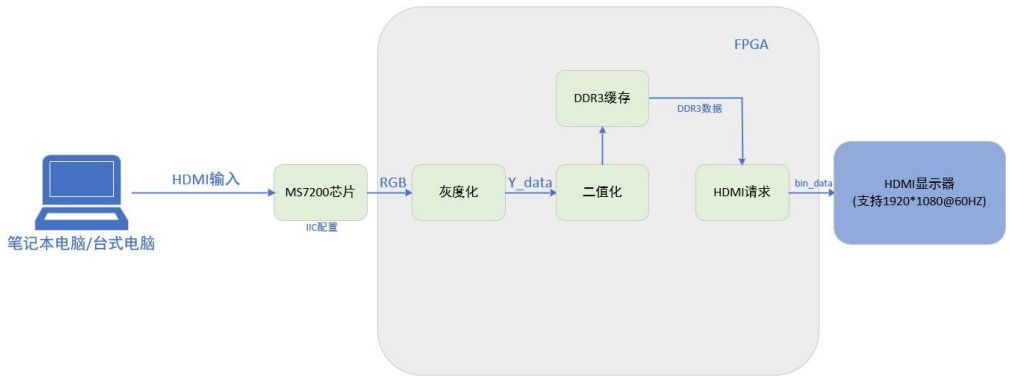


图 3.2-2

使用盘古 100K-676 开发板实现图像二值化处理方案框图如下所示, FPGA 接收 HDMI 接口视频源信号, 提取图像灰度信息 (图像灰度化处理请参考《第十一章 灰度化》内容), 采用固定阈值对图像进行全局二值化处理或通过生成 3X3 图像矩阵 (图像矩阵生成请参考《第十二章 3X3 图像矩阵生成》内容) 对图像进行局部二值化处理, 处理后的灰度图像经 DDR3 缓存再由 HDMI 接口输出显示。

3.2.1.代码模块说明

以下是全局二值化处理模块代码:

```
module binarization(

    input           clk         ,
    input           rst_n       ,
    input           vsync_in    , // vs in
    input           hsync_in    , // hs in
    input           de_in       , // de i
    input   [7:0]     y_in      ,

    output          vsync_out   , // vs o
    output          hsync_out   , // hs o
    output          de_out      , // de o
    output  reg       pix
```

```
);
//reg define
reg   vsync_in_d;
reg   hsync_in_d;
reg   de_in_d  ;
parameter Binar_THRESHOLD = 128;

assign  vsync_out = vsync_in_d  ;
assign  hsync_out = hsync_in_d  ;
assign  de_out    = de_in_d     ;

//二值化
always @(posedge clk or negedge rst_n) begin
   if(!rst_n)
      pix <= 1'b0;
   else if(y_in >= Binar_THRESHOLD)  //阈值
      pix <= 1'b1;
   else
      pix <= 1'b0;
end

//延时 1 拍以同步时钟信号
```

```
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        vsync_in_d <= 1'd0;
        hsync_in_d <= 1'd0;
        de_in_d    <= 1'd0;
    end
    else begin
        vsync_in_d <= vsync_in;
        hsync_in_d <= hsync_in;
        de_in_d    <= de_in   ;
    end
end

endmodule
```

代码中参数 Binar_THRESHOLD 设定阈值为 128, 若像素点灰度值大于该参数则二值化输出结果为 1, 反之为 0。在灰度值的判断中, 时序电路会产生 1 个 clk 延时, 因此为保持输出数据与行/场同步等信号的关系, 需将 vsync_in, hsync_in, de_in 也 “打一拍” 延迟 1 个 clk 再输出。需要注意二值化输出结果为 1 表示灰度值设置为 8’b1111_1111,二值化输出结果为 1 表示灰度值设置为 8’b0000_0000。

以下是局部二值化处理模块代码:

```
//局部二值化
module  area_bin
```

```
(
    input   wire        video_clk     ,
    input   wire        rst_n         ,
    //矩阵数据输入
    input   wire        matrix_de     ,
    input   wire        matrix_vs     ,
    input   wire  [7:0] matrix11      ,
    input   wire  [7:0] matrix12      ,
    input   wire  [7:0] matrix13      ,

    input   wire  [7:0] matrix21      ,
    input   wire  [7:0] matrix22      ,
    input   wire  [7:0] matrix23      ,

    input   wire  [7:0] matrix31      ,
    input   wire  [7:0] matrix32      ,
    input   wire  [7:0] matrix33      ,

    output  wire        area_bin_vs    ,
    output  wire        area_bin_de    ,
    output  wire        area_bin_data
);
```

```
/************************************************************
step1 每行相加  delay:1clk
************************************************************/
reg [9:0]   line1_sum;
reg [9:0]   line2_sum;
reg [9:0]   line3_sum;
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    line1_sum   <=   10'd0;
    line2_sum   <=   10'd0;
    line3_sum   <=   10'd0;
  end
  else   if(matrix_de)
  begin
    line1_sum   <=   matrix11 + matrix12 + matrix13  ;
    line2_sum   <=   matrix21 + matrix22 + matrix23  ;
    line3_sum   <=   matrix31 + matrix32 + matrix33  ;
  end
  else
  begin
    line1_sum   <=   10'd0;
```

```
        line2_sum   <=   10'd0;
        line3_sum   <=   10'd0;
    end
end
/************************************************************
step2 矩阵总和 delay:1clk
************************************************************/
reg [11:0]  data_sum;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        data_sum    <=   12'd0;
    else
        data_sum    <=   line1_sum + line2_sum + line3_sum;
end
/************************************************************
step3 求均值 /9 delay:1clk
************************************************************/
//除法转乘法   *113 再>>10
reg [17:0]  thre_data   ;   //均值当作阈值

always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
```

```
    thre_data  <=  18'd0;
  else
    thre_data  <=  data_sum * 113; //后续要>>10
end
```

通过图像灰度数据矩阵求和再求平均值作为二值化判断的阈值,除以 9 通过乘以 113 再右移 10bit 实现。

```
//中心像素点灰度值 delay:3clk, 与 thre_data 保持相对关系
reg [7:0]matrix22_0d;
reg [7:0]matrix22_1d;
reg [7:0]matrix22_2d;
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    matrix22_0d <=  8'd0;
    matrix22_1d <=  8'd0;
    matrix22_2d <=  8'd0;
  end
  else
  begin
    matrix22_0d <=  matrix22;
    matrix22_1d <=  matrix22_0d;
    matrix22_2d <=  matrix22_1d;
```

```
    end
end
```

从矩阵输入到计算出对应均值结果延迟 3 个 clk, 再利用均值作为阈值与中心像素点灰度值作对比时,应将中心像素点灰度值也延迟 3 个 clk,保持两者对应关系。若不延迟 matrix22,则造成前 3 个矩阵的均值作为当前矩阵中心像素点二值化判断的阈值。

```
/*************************************************************
step4 中心像素点与阈值进行比较   delay:1clk
*************************************************************/
reg bin_data;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        bin_data    <=  1'd0;
    else    if(matrix22_2d >= thre_data[17:10])
        bin_data    <=  1'd1;
    else
        bin_data    <=  1'd0;
end
/*************************************************************
时钟延迟 一共延迟 4clk
*************************************************************/
reg [3:0]   video_de_reg;
reg [3:0]   video_vs_reg;
```

```
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    video_de_reg   <=  4'd0;
    video_vs_reg   <=  4'd0;
  end
  else
  begin
    video_de_reg   <=  {video_de_reg[2:0],matrix_de};
    video_vs_reg   <=  {video_vs_reg[2:0],matrix_vs};
  end
end

assign  area_bin_de    =  video_de_reg[3] ;
assign  area_bin_vs    =  video_vs_reg[3] ;
assign  area_bin_data  =  bin_data      ;

endmodule
```

图像灰度值进行局部二值化处理后总共延时 4 个 clk, 为保持二值化结果与矩阵场同步信号及行有效信号的相对关系, 需将 matrix_de 信号和 matrix_vs 信号延迟 4 个 clk。

2.1.3.2 代码仿真

3.2.1.1. Matlab 仿真介绍

```
%% 读取图像
originalImage = imread('tttt.jpeg');
figure;
%% 显示原图
subplot(2, 3, 1);
imshow(originalImage);
title('原图像');
%% 灰度化
grayImage = rgb2gray(originalImage);
%% 图像尺寸
[img_height, img_width] = size(grayImage);
binaryImage = zeros(img_height, img_width);
%% 定义参数
windowSize = 27;  % 窗口大小, 奇数
%% 遍历像素 计算窗口均值
halfWindowSize = floor(windowSize / 2);
for i = 1:img_height
   for j = 1:img_width
      % 窗口边界
      r1 = max(i - halfWindowSize, 1);    %窗口上边界
      r2 = min(i + halfWindowSize, img_height);%窗口下边界
```

```
        c1 = max(j - halfWindowSize, 1);%窗口左边界
        c2 = min(j + halfWindowSize, img_width);%窗口右边界

        % 提取窗口
        window = grayImage(r1:r2, c1:c2);

        % 计算窗口内的均值
        localMean = mean(window(:));

        % 计算阈值
        threshold = localMean*0.9;

        % 局部二值化处理
        if grayImage(i, j) < threshold
            binaryImage(i, j) = 0;
        else
            binaryImage(i, j) = 255;
        end
    end
end
```

主要看代码的 9-41 行, size()函数可以获得图像的尺寸, 通过 zero()函数创建一个和原

图像一样大小的图像矩阵。19-23 行是计算窗口的上下边界, 来获得矩阵, 矩阵大小由 windowSize 来决定。mean()函数计算窗口均值, 本次阈值将均值结果*0.9 来得到, 之后让每个像素点与阈值比较, 比阈值小即视为黑色, 比阈值大则为白色。效果如下:

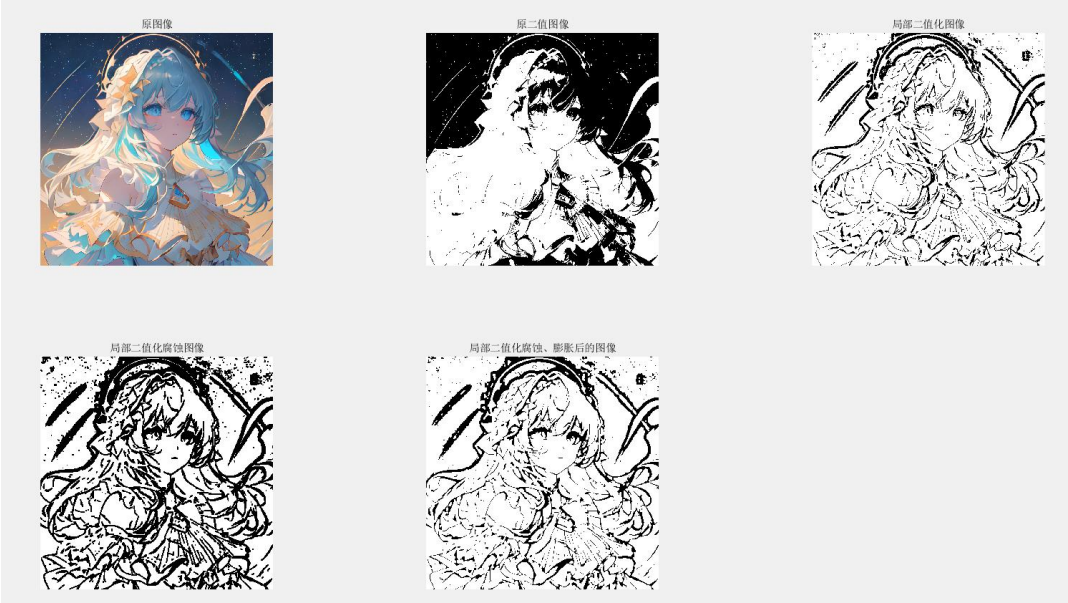


图 3.2-3

3.2.1.2. Modelsim 仿真介绍

在 modelsim 平台使用 img_process_tb 仿真激励模块, 将视频源经灰度化处理后接到全局二值化处理模块, 或将灰度化处理接的数据接到 matrix_3x3 模块生成 3X3 图像矩阵后经局部二值化模块处理, 以下为全局二值化和局部二值化仿真结果分析:

全局二值化处理仿真结果:

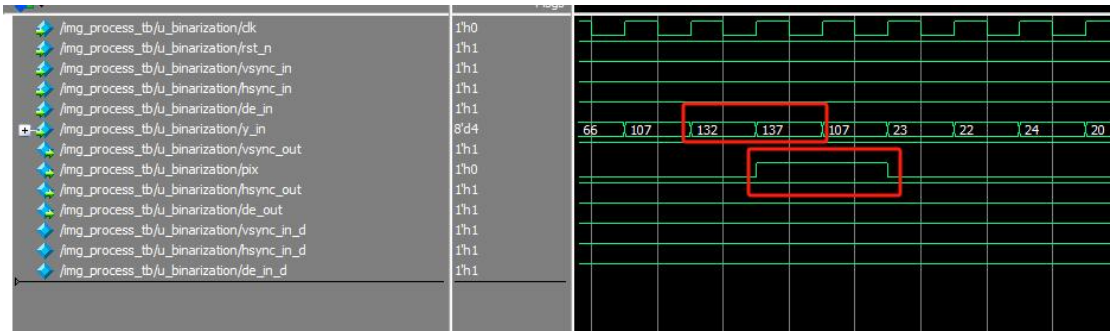


图 3.2-4

在对应灰度值大于等于 128 时, 二值化结果 pix 信号为高电平, 延时 1clk。

图 3.2-5

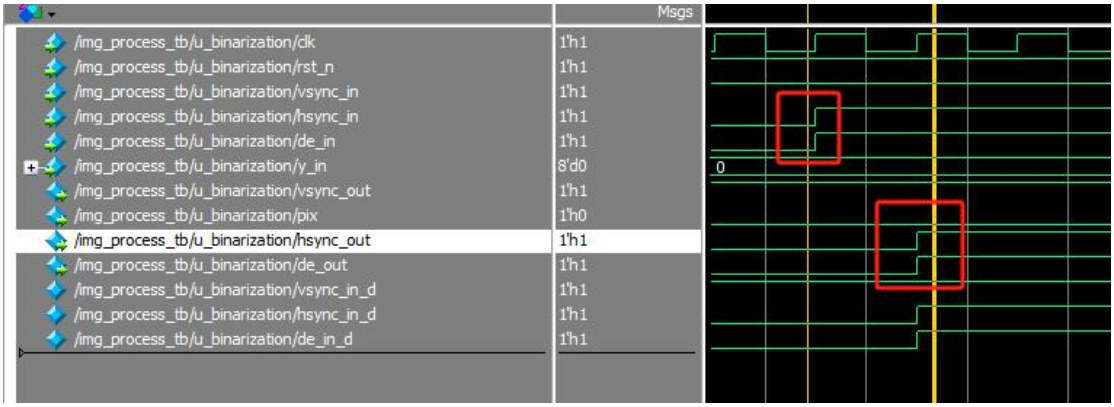


图 3.2-6

同时, vsync_in, hsync_in, de_in 也延时 1clk 再输出, 保持与 pix 对应关系。

局部二值化处理仿真结果:

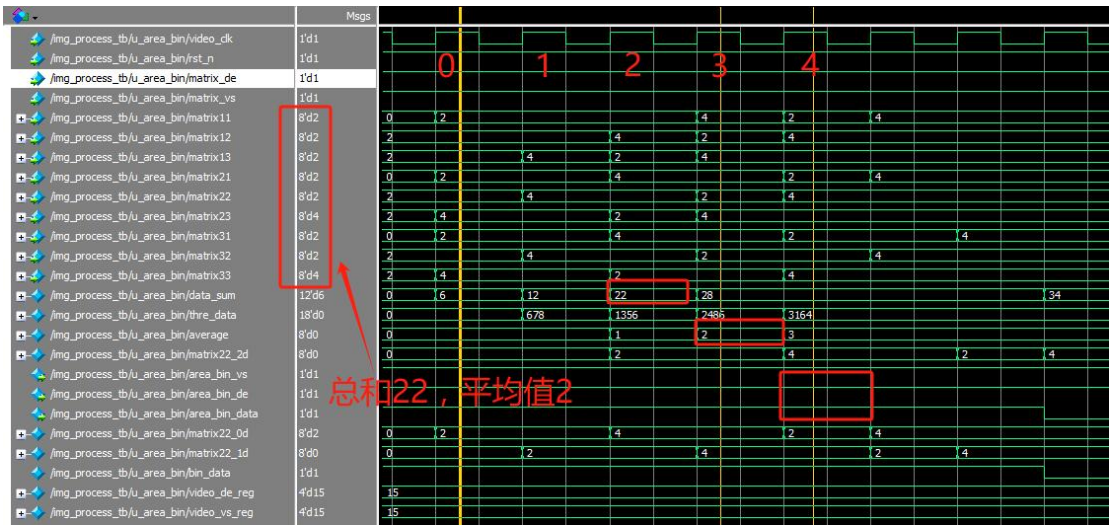


图 3.2-7

在标“1”处时钟上升沿所采的矩阵数据先计算行总和在标“2”处时钟上升沿计算出三行总和为 22, 在下一个时钟上升沿“3”计算出 thre_data 为 2486 (即 2486 除以 2^10 再取整, 平均值为 2), 在下一个时钟上升沿“4” matrix22_2d 为 2 等于平均值 2 得出局部二值化结果为 1

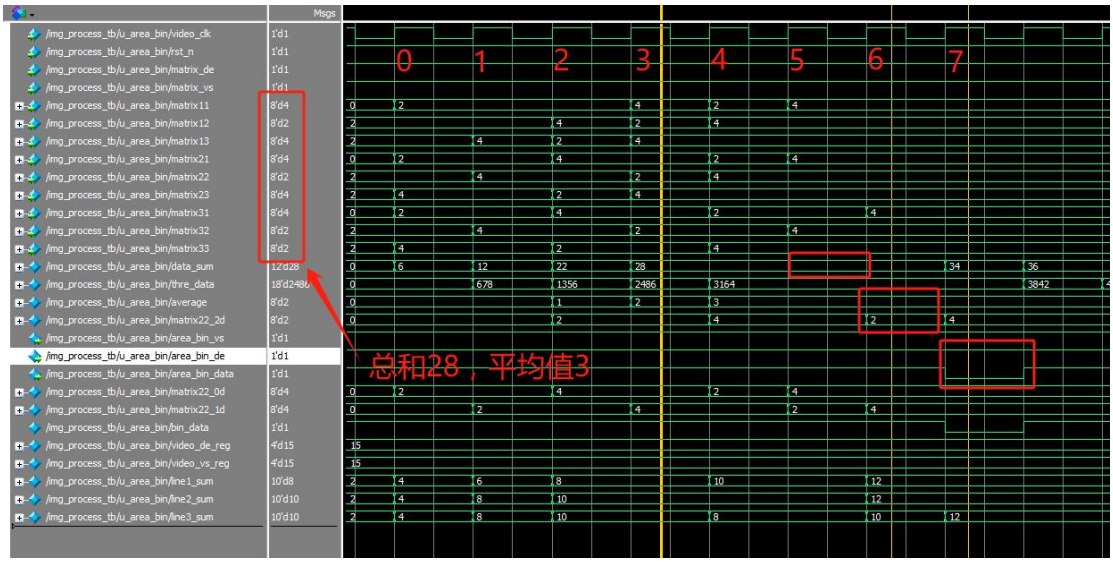


图 3.2-8

同样的, 在标“4”处时钟上升沿所采的矩阵数据先计算行总和在标“5”处时钟上升沿计算出三行总和为 28,在下一个时钟上升沿“6”计算出 thre_data 为 3164(即 3164 除以 2^10 再取整, 平均值为 3), 在下一个时钟上升沿“7” matrix22_2d 为 2 小于平均值 3 得出局部二值化结果为 0。

以下是局部二值化仿真后的 txt 文件到 matlab 恢复出的图像:

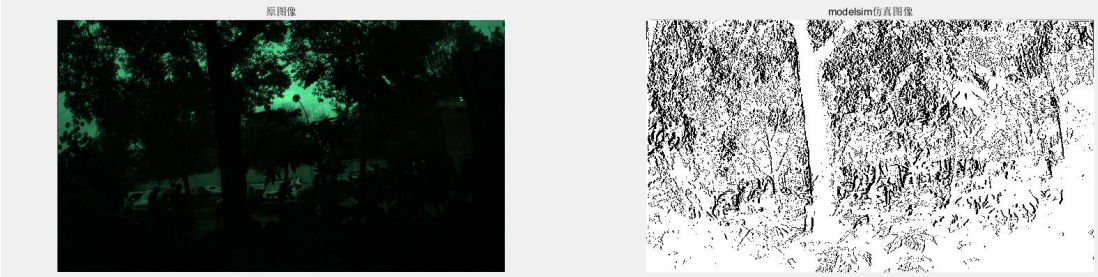


图 3.2-9

可以通过修改矩阵大小和阈值来改善质量。

3.2.2.实验现象

连接好下载器，电源、HDMI_IN 口连接 PC、HDMD_OUT 口连接显示器，然后下载程序。

图 3.2-10

图 3.2-11

上图为原图像。

图 3.2-12

上图为二值化后的图像。

4. 腐蚀膨胀

4.1. 实验简介

实验目的:

了解如何将二值化图像进行腐蚀和膨胀。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

4.2. 实验原理

腐蚀膨胀是针对二值化图像来处理的。腐蚀的作用是可以消除一些边界点, 消除一些细小物体, 让图像在某些边界可以平滑。膨胀的作用是填充某些空洞, 是某些部分向外扩张以达到连接边界等作用。

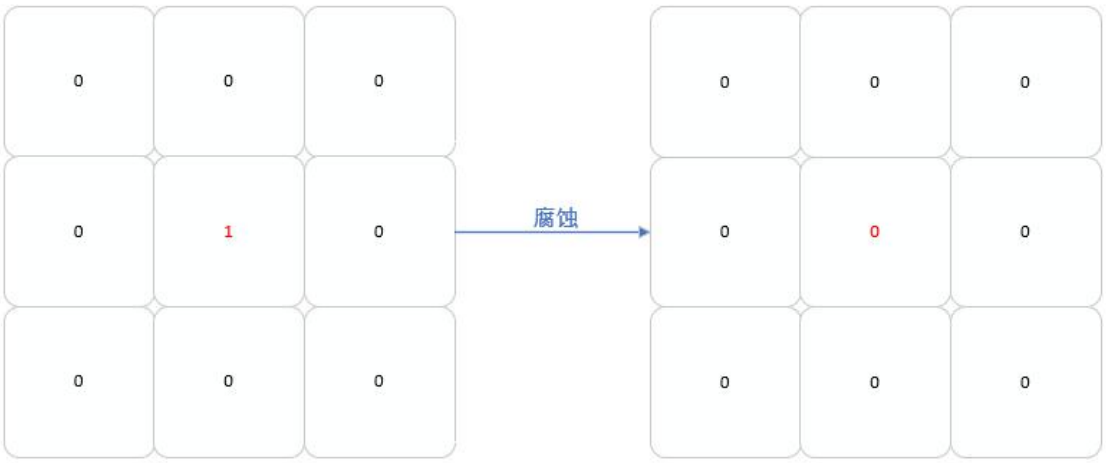


图 4.2-1

当 3x3 窗口内有一个像素为 0 那就认为当前像素的输出为 0。本质就是输出等于 9 个数据相与。

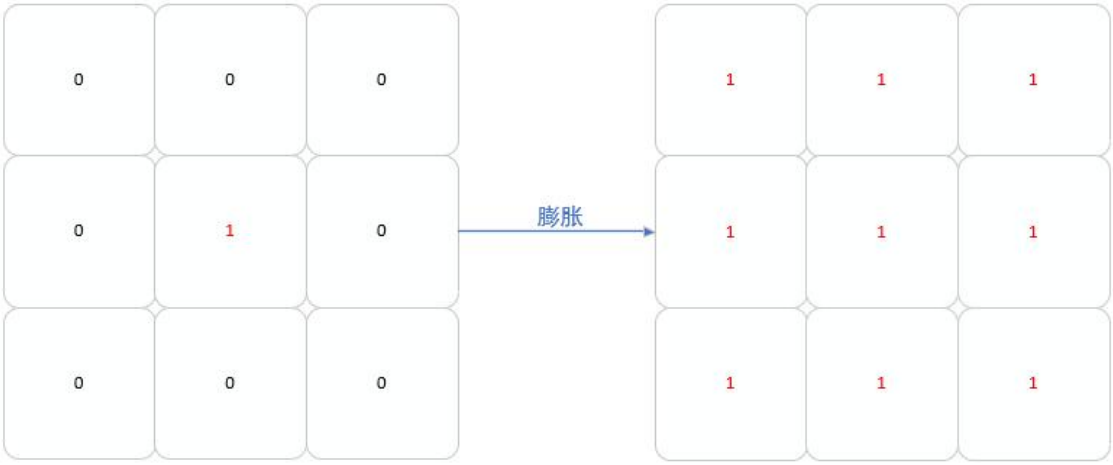


图 4.2-2

当 3x3 窗口内有一个像素为 1 那就认为当前像素的输出为 1。本质就是输出等于 9 个数据相或。

了解其原理后，可以开始编写代码。只要进行相与或者相或即可。

还有个要注意的点就是之前的 3x3 矩阵生成的是 8bit,然而二值化数据只有 1bit,因此需要把 3x3 矩阵改成 1bit 的.

只需要把端口,以及变量改为 1bit,FIFO IP 的数据位宽也改为 1bit 即可.如下所示:

```
module  matrix_3x3_1bit
#(
  parameter  IMG_WIDTH  =  11'd1920   ,
  parameter  IMG_HEIGHT =  11'd1080
)
(
```

```
    input   wire        video_clk     ,
    input   wire        rst_n         ,

    //
    input   wire        video_vs      ,
    input   wire        video_de      ,
    input   wire        video_data    ,

    //3x3 矩阵输出
    output  wire        matrix_de     ,
    output  reg         matrix11      ,
    output  reg         matrix12      ,
    output  reg         matrix13      ,

    output  reg         matrix21      ,
    output  reg         matrix22      ,
    output  reg         matrix23      ,

    output  reg         matrix31      ,
    output  reg         matrix32      ,
    output  reg         matrix33

```

```
);
//矩阵数据生成
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    {matrix11, matrix12, matrix13} <= 3'd0;
    {matrix21, matrix22, matrix23} <= 3'd0;
    {matrix31, matrix32, matrix33} <= 3'd0;
  end
  else  if(video_de)
  begin
    {matrix11, matrix12, matrix13} <= {matrix12, matrix13, line1_data};
    {matrix21, matrix22, matrix23} <= {matrix22, matrix23, line2_data_d0};
    {matrix31, matrix32, matrix33} <= {matrix32, matrix33, line3_data_d1};
  end
  else
  begin
    {matrix11, matrix12, matrix13} <= 3'd0;
    {matrix21, matrix22, matrix23} <= 3'd0;
    {matrix31, matrix32, matrix33} <= 3'd0;
  end
```

end

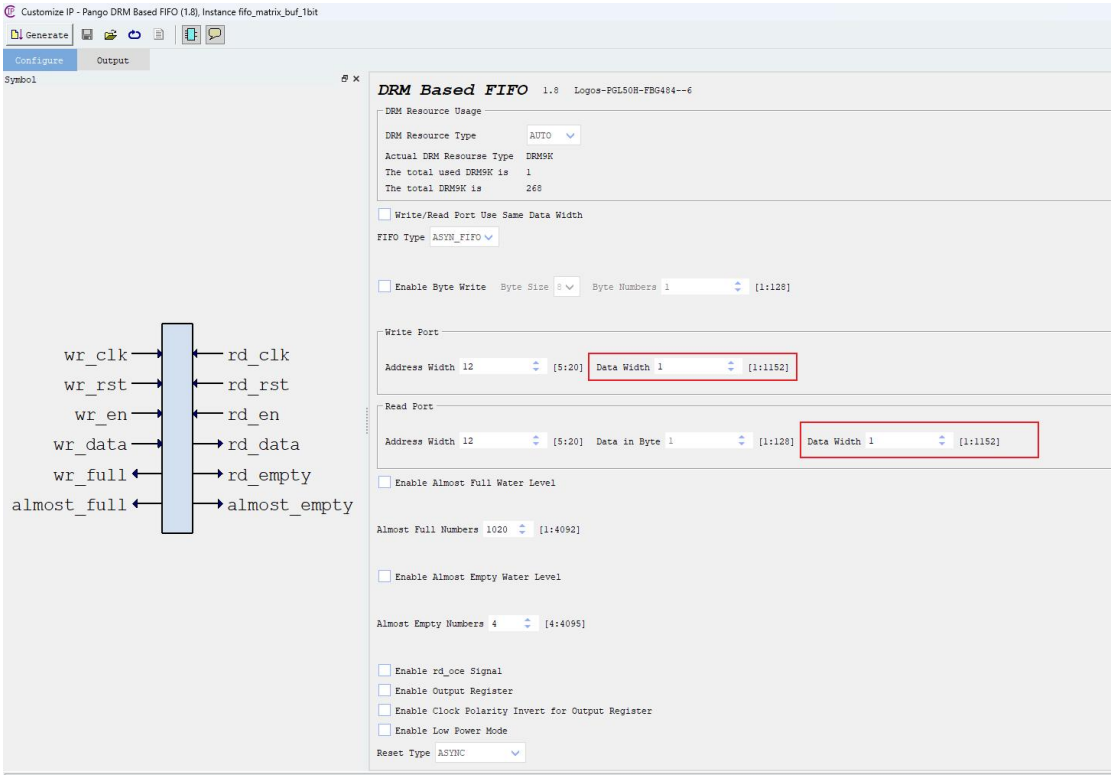


图 4.2-3

FIFO IP 的设置修改如上.

2.1.2 接口列表

erosion.v 腐蚀模块接口,延迟 2clk.

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
bin_vs	input	1	处理前的视频场信号
bin_de	input	1	数据有效信号

bin_data11	input	1	3X3 矩阵第 1 个二值像素
bin_data12	input	1	3X3 矩阵第 2 个二值像素
bin_data13	input	1	3X3 矩阵第 3 个二值像素
bin_data21	input	1	3X3 矩阵第 4 个二值像素
bin_data22	input	1	3X3 矩阵第 5 个二值像素
bin_data23	input	1	3X3 矩阵第 6 个二值像素
bin_data31	input	1	3X3 矩阵第 7 个二值像素
bin_data32	input	1	3X3 矩阵第 8 个二值像素
bin_data33	input	1	3X3 矩阵第 9 个二值像素
erosion_vs	output	1	输出腐蚀处理后的场信号
erosion_de	output	1	输出腐蚀处理后的数据有效信号
erosion_data	output	1	输出腐蚀处理后的数据

dilatw.v 膨胀模块接口,延迟 2clk.

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
bin_vs	input	1	处理前的视频场信号
bin_de	input	1	数据有效信号

bin_data11	input	1	3X3 矩阵第 1 个二值像素
bin_data12	input	1	3X3 矩阵第 2 个二值像素
bin_data13	input	1	3X3 矩阵第 3 个二值像素
bin_data21	input	1	3X3 矩阵第 4 个二值像素
bin_data22	input	1	3X3 矩阵第 5 个二值像素
bin_data23	input	1	3X3 矩阵第 6 个二值像素
bin_data31	input	1	3X3 矩阵第 7 个二值像素
bin_data32	input	1	3X3 矩阵第 8 个二值像素
bin_data33	input	1	3X3 矩阵第 9 个二值像素
dilate_vs	output	1	输出膨胀处理后的场信号
dilate_de	output	1	输出膨胀处理后的数据有效信号
dilate_data	output	1	输出膨胀处理后的数据

2.1.3 工程说明

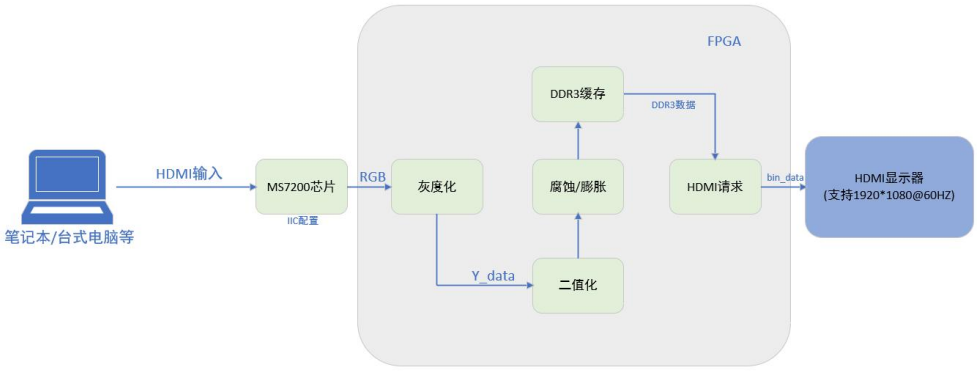


图 4.2-4

上图为本次实验的工程框架, 在二值化的基础上添加腐蚀膨胀后再缓存到 DDR 中, 再将数据读出到 HDMI 显示器上进行显示.

4.2.1.代码模块说明

```
//erosion 腐蚀
module  erosion
(
   input   wire        video_clk   ,  //像素时钟
   input   wire        rst_n       ,

   //输入二值化数据
   input   wire        bin_vs      ,
   input   wire        bin_de      ,
   input   wire        bin_data_11 ,
   input   wire        bin_data_12 ,
```

```
    input   wire        bin_data_13 ,
    input   wire        bin_data_21 ,
    input   wire        bin_data_22 ,
    input   wire        bin_data_23 ,
    input   wire        bin_data_31 ,
    input   wire        bin_data_32 ,
    input   wire        bin_data_33 ,

    output  wire        erosion_vs  ,
    output  wire        erosion_de  ,
    output  wire        erosion_data

);

/*****************************************************
wire define
*****************************************************/


/*****************************************************
reg define
*****************************************************/
```

```
reg erosion_vs_d   ;
reg erosion_vs_d1  ;
reg erosion_de_d   ;
reg erosion_de_d1  ;
reg erosion_data_d ;
reg erosion_line0  ;
reg erosion_line1  ;
reg erosion_line2  ;
// 1clk 行腐蚀 相与
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    erosion_line0  <=  1'd0;
    erosion_line1  <=  1'd0;
    erosion_line2  <=  1'd0;
  end
  else  if(bin_de)
  begin
    erosion_line0  <=  bin_data_11 && bin_data_12 && bin_data_13;
    erosion_line1  <=  bin_data_21 && bin_data_22 && bin_data_23;
    erosion_line2  <=  bin_data_31 && bin_data_32 && bin_data_33;
  end
```

```
end



// 1clk  腐蚀 相与
always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      erosion_data_d  <=  1'd0;
   else
      erosion_data_d  <=  erosion_line0 && erosion_line1 && erosion_line2;
end



// 延迟 2clk
always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
   begin
      erosion_vs_d   <=  1'd0;
      erosion_vs_d1  <=  1'd0;
```

```
        erosion_de_d   <=  1'd0;
        erosion_de_d1  <=  1'd0;

    end
    else
    begin
        erosion_vs_d   <=  bin_vs;
        erosion_vs_d1  <=  erosion_vs_d;
        erosion_de_d   <=  bin_de;
        erosion_de_d1  <=  erosion_de_d;
    end
end
assign  erosion_data   = erosion_data_d;

assign  erosion_vs     = erosion_vs_d1 ;
assign  erosion_de     = erosion_de_d1 ;

endmodule
```

同样的,笔者没有通过组合逻辑把 9 个数据直接相与,而是通过二级流水线来完成腐蚀操作.

代码的 43-56 行完成每行数据相与,当 bin_de 有效时将三行数据分别相与.

代码的 63-68 行将每行相与的结果再次相与,至此完成了 9 个数据相与.

因为是两级流水线,所以延迟了 2 个 clk.故在代码的 73-89 行对有效信号和场信号打两拍,延迟两个时钟周期,让数据保持同步.

最后通过组合逻辑将数据和有效信号以及场信号输出.

dilate 模块介绍.

```
//dilate 膨胀
module  dilate
(
  input   wire        video_clk   ,  //像素时钟
  input   wire        rst_n       ,

  //输入二值化数据
  input   wire        bin_vs      ,
  input   wire        bin_de      ,
  input   wire        bin_data_11 ,
  input   wire        bin_data_12 ,
  input   wire        bin_data_13 ,
  input   wire        bin_data_21 ,
  input   wire        bin_data_22 ,
  input   wire        bin_data_23 ,
  input   wire        bin_data_31 ,
  input   wire        bin_data_32 ,
  input   wire        bin_data_33 ,
```

```
    output  wire        dilate_vs   ,
    output  wire        dilate_de   ,
    output  wire        dilate_data
    
);

/************************************************************
wire define
************************************************************/
/************************************************************
reg define
************************************************************/
reg dilate_vs_d ;
reg dilate_vs_d1    ;
reg dilate_de_d ;
reg dilate_de_d1    ;
reg dilate_data_d   ;
reg dilate_line0    ;
reg dilate_line1    ;
reg dilate_line2    ;
// 1clk  行膨胀 相或
```

```
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
     dilate_line0   <=   1'd0;
     dilate_line1   <=   1'd0;
     dilate_line2   <=   1'd0;
  end
  else   if(bin_de)
  begin
     dilate_line0   <=   bin_data_11 || bin_data_12 || bin_data_13;
     dilate_line1   <=   bin_data_21 || bin_data_22 || bin_data_23;
     dilate_line2   <=   bin_data_31 || bin_data_32 || bin_data_33;
  end
end


// 1clk  膨胀 相或
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
     dilate_data_d  <=  1'd0;
  else
     dilate_data_d  <=  dilate_line0 || dilate_line1 || dilate_line2;
end
```

```
// 延迟 2clk
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    dilate_vs_d <=  1'd0;
    dilate_vs_d1  <= 1'd0;
    dilate_de_d <=  1'd0;
    dilate_de_d1  <= 1'd0;

  end
  else
  begin
    dilate_vs_d   <=  bin_vs;
    dilate_vs_d1   <= dilate_vs_d;
    dilate_de_d   <=  bin_de;
    dilate_de_d1   <= dilate_de_d;
  end
end
assign  dilate_data    = dilate_data_d;
```

```
assign  dilate_vs      = dilate_vs_d1  ;
assign  dilate_de      = dilate_de_d1  ;

endmodule
```

细心的读者其实可以发现 erosion 模块和 dilate 模块的代码基本是一致的,只是把相与操作换成相或操作.整个代码架构是基本不变的,还是使用二级流水线.

代码的 41-62 行,把 erosion 的相与操作改成了相或操作,两个 clk 后得到 9 个数据相或的结果.

因为是两级流水线,所以延迟了 2 个 clk.故在代码的 67-83 行对有效信号和场信号打两拍,延迟两个时钟周期,让数据保持同步.

最后通过组合逻辑将数据和有效信号以及场信号输出.

4.2.2.代码仿真

4.2.2.1. Matlab 仿真介绍

基本的开窗操作,以及读取图像在前文以及介绍过,因此,后文涉及到相关操作将不在详细介绍,只介绍关键算法部分.本节介绍其中的腐蚀膨胀部分.

```
for i = 1:img_height
  for j = 1:img_width
    % 窗口边界
    r1 = max(i - halfWindowSize, 1);   %窗口上边界
    r2 = min(i + halfWindowSize, img_height); %窗口下边界
    c1 = max(j - halfWindowSize, 1);   %窗口左边界
```

```
        c2 = min(j + halfWindowSize, img_width); %窗口右边界

        % 提取窗口
        window = binaryImage(r1:r2, c1:c2);

        % 计算窗口内的最小值
        localMin = min(window(:));

        % 进行腐蚀操作
        erodedImage(i, j) = localMin;
    end
end
```

腐蚀其实就是 3x3 窗口内有一个是 0,那结果就为 0,而二值化的数要不就是 0 要不就是 1,所以其实 0 就是窗口内的最小值,所以代码的第 13 行通过 min()函数,直接找到该窗口里的最小值,如果 9 个数据里有一个是 0,那 16 行的结果肯定就是 0,如果 9 个数据全为 1 的话,那最小值就是 1,那 16 行的结果就是 1.

同样的, 如果是膨胀操作的话, 就把 min(window(:))改为 max(window(:))即可。

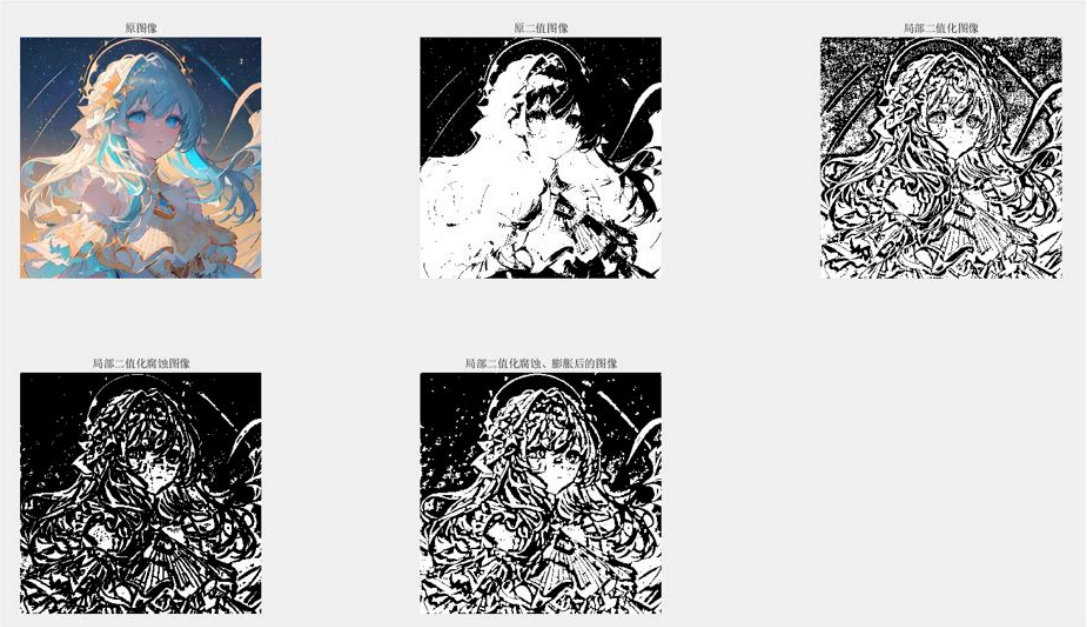


图 4.2-5

上图为 Matlab 仿真后的图。

4.2.2.2. Modelsim 仿真介绍

具体工程架构还是和之前一致,在灰度化,二值化后的基础上添加了腐蚀膨胀模块.

```
//图像处理仿真模块
`timescale 1ns/1ns
module  img_process_tb();

//图片高度宽度
//仿真需要改小视频大小 避免太大

//时序参考模板  仿真中不需要严格按照 vesa 时序标准
parameter IMG_WIDTH = 16'd1280;//有效区域
```

```
parameter H_FP = 16'd110;   //前沿
parameter H_SYNC = 16'd40;  //同步
parameter H_BP = 16'd220 ;  //后沿
parameter TOTAL_WIDTH = IMG_WIDTH  + H_FP  +H_SYNC + H_BP;

parameter IMG_HEIGHT = 16'd720; //有效区域
parameter V_FP  = 16'd5;    //前沿
parameter V_SYNC  = 16'd5;  //同步
parameter V_BP  = 16'd20;   //后沿
parameter TOTAL_HEIGHT = IMG_HEIGHT + V_FP + + V_SYNC + V_BP;

localparam  HREF_DELAY  =  5;
localparam  VSYNC_DELAY =  5;

reg video_clk  ;
reg rst_n     ;



wire        video_vs;
wire        video_de;
```

```
wire   [23:0]  video_data;


wire         y_vs;
wire         y_hs;
wire         y_de;
wire   [7:0]   y_data;

//全局二值化数据
wire    bin_vs  ;
wire    bin_hs  ;
wire    bin_de  ;
wire    bin_data;


wire          matrix_de;
wire   [7:0]    matrix11 ;
wire   [7:0]    matrix12 ;
wire   [7:0]    matrix13 ;
wire   [7:0]    matrix21 ;
wire   [7:0]    matrix22 ;
wire   [7:0]    matrix23 ;
wire   [7:0]    matrix31 ;
```

```
wire   [7:0]   matrix32 ;
wire   [7:0]   matrix33 ;



//局部二值化数据
wire   area_bin_vs  ;
wire   area_bin_hs  ;
wire   area_bin_de  ;
wire   area_bin_data;

//腐蚀数据
wire   erosion_vs;
wire   erosion_de;
wire   erosion_data;
//膨胀数据
wire   dilate_vs;
wire   dilate_de;
wire   dilate_data;

//定义处理后的文件
integer output_file;
initial
```

```
begin
    video_clk   =   1'd0;
    rst_n       =   1'd0;
    #20
    rst_n       =   1'd1;
    output_file =   $fopen("D:/pango_isp/img_test_pg/img_test_pg/01_led_test/
sim/img_process.txt","w");
end
//生成 100MHZ
always#5 video_clk = ~video_clk;



//读取图片数据

video_data_gen#(
    .TOTAL_WIDTH  ( 12'd1650 ),
    .IMG_WIDTH    ( 12'd1280 ),
    .H_SYNC       ( 12'd40 ),
    .H_BP         ( 12'd220 ),
    .H_FP         ( 12'd110 ),
    .TOTAL_HEIGHT ( 12'd750 ),
    .IMG_HEIGHT   ( 12'd720 ),
```

```
    .V_SYNC     ( 12'd5 ),
    .V_BP       ( 12'd20 ),
    .V_FP       ( 12'd5 )
)u_video_data_gen(
    .video_clk  ( video_clk  ),
    .rst_n      ( rst_n      ),
    .video_vs   ( video_vs   ),
    .video_de   ( video_de   ),
    .video_data ( video_data )
);


//灰度化

RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n    ),
    .vsync_in ( video_vs ),
    .hsync_in ( video_de ),
    .de_in    ( video_de    ),
    .red      ( video_data[23:19]  ),
    .green    ( video_data[15:10]  ),
```

```
    .blue     ( video_data[7:3]     ),
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out    ( y_de  ),
    .y        ( y_data ),
    .cb       (       ),
    .cr       (       )
);



//全局二值化
binarization u_binarization(
    .clk       ( video_clk       ),
    .rst_n     ( rst_n     ),
    .vsync_in   ( y_vs   ),
    .hsync_in   ( y_hs   ),
    .de_in     ( y_de     ),
    .y_in      ( y_data      ),
    .vsync_out  ( bin_vs  ),
    .hsync_out  ( bin_hs  ),
    .de_out    ( bin_de    ),
```

```
    .pix      ( bin_data      )
);

//二值化矩阵
matrix_3x3_1bit#(
    .IMG_WIDTH   ( IMG_WIDTH ),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3_1bit(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n   ),
    .video_vs    ( bin_vs       ),
    .video_de    ( bin_de       ),
    .video_data  ( bin_data     ),
    .matrix_de   ( matrix_de_1bit   ),
    .matrix11    ( matrix11_1bit    ),
    .matrix12    ( matrix12_1bit    ),
    .matrix13    ( matrix13_1bit    ),
    .matrix21    ( matrix21_1bit    ),
    .matrix22    ( matrix22_1bit    ),
    .matrix23    ( matrix23_1bit    ),
    .matrix31    ( matrix31_1bit    ),
    .matrix32    ( matrix32_1bit    ),
```

```
    .matrix33   ( matrix33_1bit   )
);
//腐蚀模块
erosion u_erosion(
    .video_clk   ( video_clk   ),
    .rst_n       ( rst_n       ),
    .bin_vs      ( area_bin_vs      ),
    .bin_de      ( matrix_de_1bit      ),
    .bin_data_11  ( matrix11_1bit  ),
    .bin_data_12  ( matrix12_1bit  ),
    .bin_data_13  ( matrix13_1bit  ),
    .bin_data_21  ( matrix21_1bit  ),
    .bin_data_22  ( matrix22_1bit  ),
    .bin_data_23  ( matrix23_1bit  ),
    .bin_data_31  ( matrix31_1bit  ),
    .bin_data_32  ( matrix32_1bit  ),
    .bin_data_33  ( matrix33_1bit  ),
    .erosion_vs   ( erosion_vs   ),
    .erosion_de   ( erosion_de   ),
    .erosion_data  ( erosion_data  )
);
//腐蚀后的 3x3 矩阵
```

```
matrix_3x3_1bit#(
    .IMG_WIDTH   ( IMG_WIDTH),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3_erosion(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n   ),
    .video_vs    ( erosion_vs       ),
    .video_de    ( erosion_de       ),
    .video_data  ( erosion_data     ),
    .matrix_de   ( matrix_de_erosion   ),
    .matrix11    ( matrix11_erosion    ),
    .matrix12    ( matrix12_erosion    ),
    .matrix13    ( matrix13_erosion    ),
    .matrix21    ( matrix21_erosion    ),
    .matrix22    ( matrix22_erosion    ),
    .matrix23    ( matrix23_erosion    ),
    .matrix31    ( matrix31_erosion    ),
    .matrix32    ( matrix32_erosion    ),
    .matrix33    ( matrix33_erosion    )
);
 //膨胀模块
dilate u_dilate(
```

```
    .video_clk   ( video_clk   ),
    .rst_n       ( rst_n       ),
    .bin_vs      ( erosion_vs   ),
    .bin_de      ( matrix_de_erosion   ),
    .bin_data_11  ( matrix11_erosion  ),
    .bin_data_12  ( matrix12_erosion  ),
    .bin_data_13  ( matrix13_erosion  ),
    .bin_data_21  ( matrix21_erosion  ),
    .bin_data_22  ( matrix22_erosion  ),
    .bin_data_23  ( matrix23_erosion  ),
    .bin_data_31  ( matrix31_erosion  ),
    .bin_data_32  ( matrix32_erosion  ),
    .bin_data_33  ( matrix33_erosion  ),
    .dilate_vs   ( dilate_vs   ),
    .dilate_de   ( dilate_de   ),
    .dilate_data  ( dilate_data  )
);



GTP_GRS GRS_INST(
    .GRS_N(1'b1)
    ) ;
```

```

//写数据
reg video_vs_d ;  //打拍寄存
reg img_done   ;
wire   frame_flag;


always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
     video_vs_d  <=  1'd0;
  else
     video_vs_d  <=  dilate_vs ;
end

assign frame_flag = ~dilate_vs& video_vs_d;   //下降沿

always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
     img_done <= 1'b0;
  else if(frame_flag)   //下降沿 判断一帧结束
     img_done <= 1'b1;
  else
```

```
        img_done <= img_done;
end


always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;    //停止仿真
    end
    else if(dilate_de)    //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",8{dilate_data  },8{dilate_data  },8{dilate
_data  });    //16 进制写入
    end
end


endmodule
```

代码的基本框架没有改变,在代码的 145-223 行可以看到在二值化的基础上添加了腐蚀膨胀以及 3x3_1bit 矩阵生成的模块.

注意把 236-264 行的代码里的数据有效信号和长信号均替换为膨胀处理后的数据有效信号和场信号。

最后代码的 262 行由于是 8bit,16 进制写入到 txt,而二值化数据只有 1bit,因此使用

8{dilate_data}将其 1bit 复制 8 份扩充为 8bit,然后写入到 txt。

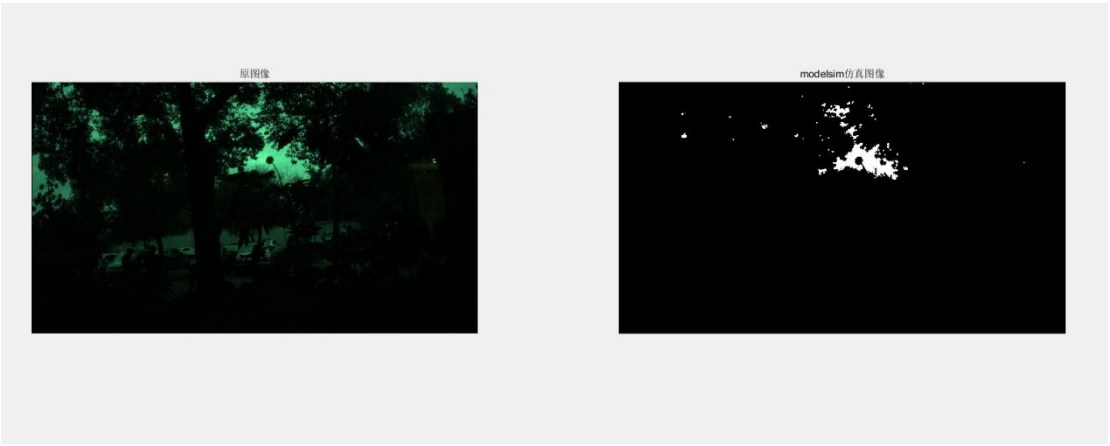


图 4.2-6

4.2.3.实验现象

连接好下载器, 电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器, 然后下载程序。

图 4.2-7

图 4.2-8

上图为实验原图像。

图 4.2-9

上图为二值化腐蚀膨胀后的图像。

5. 中值滤波

5.1. 实验简介

实验目的:

生成 3x3 矩阵, 并完成中值滤波。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

5.2. 实验原理

在图像处理的入门算法中, 中值滤波算法是一种经典的噪声平滑算法, 它可以帮助我们过滤图片上的椒盐噪声, 非常适合孤立的噪声点处理。

以目前主流的 rgb888 图像显示编码方案来说, 图像的颜色深浅, 是由 RGB 三色搭配后呈 24 位线性变化显示颜色的差异的, 如果一张图片中某一点的值异于周边的值太多, 则我们判定其为噪声。下面我们用通俗举例子的方法, 一边讲解中值滤波的 FPGA 排序算法, 一边分析滤波后算法的结果。

举例来说, 有 3x3 的图像数据矩阵如下:

2	4	80
3	5	4
6	3	2

图 5.2-1

从上面 3x3 的数据矩阵中,不难发现,有数据 80 和周边的点有明显的颜色显示数值差异。反映到图像中, 应该是一个比周边值更亮的噪声孤点。

如果采用中值滤波算法, 对以上原始数据矩阵进行处理:

第一步: 首先将 9 个图像数据按行分成 3 组, 分别对三组数据进行排序。

(1)对 L11, L12, L13 进行排序得到:

L1max (即 80), L1mid (即 4), L1min (即 2);

(2)对 L21, L22, L23 进行排序得到 :

L2max (即 5), L2mid (即 4), L2min (即 3);

(3)对 L31, L32, L33 进行排序得到:

L3max (即 6), L3mid (即 3), L3min (即 2)。

处理后, 得到一个临时过渡的矩阵, 内容为:

80	4	2
5	4	3
6	3	2

图 5.2-2

第二步: 分别对三组像素数据中的 3 个最大, 3 个中间和 3 个最小分别进行排序。

(1)对 L1max, L2max, L3max 进行排序得到 :

Lmaxmax (即 80), Lmaxmid (即 6), Lmaxmin (即 5);

(2)对 L1mid, L2mid, L3mid 进行排序得到:

Lmidmax (即 4), Lmidmid (即 4), Lmidmin (即 3);

(3)对 L1min, L2min, L3min 进行排序得到:

Lminmax (即 3), Lminmid (即 2), Lminmin (即 2);

处理后, 得到这样第二个临时的过渡矩阵, 内容为:

80	6	5
4	4	3
3	2	2

图 5.2-3

第三步: 对最大的最小(Lmaxmin), 中间的中间(Lmidmid)以及最小的最大(Lminmax)进行排序, 排序得到的中间数据为最终的中值。

5（第一行末，最大的最小）	4（第二行中，中间的中间）	3（第三行首，最小的最大）

最终，得到该图像矩阵中值为 4。

这个中值，最终作为我们数据 9 宫格的第二行第二列的值。

由于 FPGA 的图像显示数据采用的是逐个像素点的行场扫描输出方案，因此，当扫描到噪声信号点（如图中数据 80）后，经过中值运算点阵的处理，该点是不会从矩阵输出的，即实现了图片噪声信号滤波。

总结下，中值滤波方法是，对待处理的当前像素，选择一个模板，该模板为其邻近的若干个像素组成，对模板的像素由小到大进行排序，再用模板的中值来替代原像素的值的方法。

当我们使用 3x3 窗口后获取邻域中的 9 个像素，就需要对 9 个像素值进行排序，为了提高排序效率，排序算法思想如下图所示：

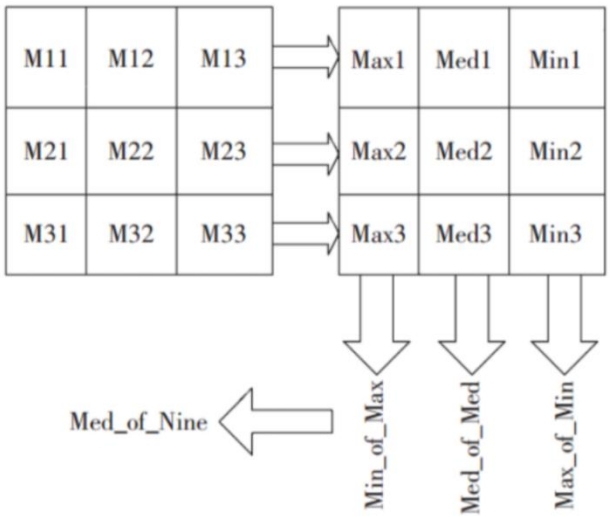


图 5.2-4

（1）对窗内的每行像素按降序排序，得到最大值、中间值和最小值；

（2）把三行的最小值相比较, 取其中的最大值;

（3）把三行的最大值相比较, 取其中的最小值;

（4）把三行的中间值相比较, 再取一次中间值;

（5）把前面得到的三个值再做一次排序, 获得的中值即该窗口的中值。

关于如何形成 3*3 的像素矩阵, 请参考第十二章的内容, 本章节重点讲解中值滤波的算法实现。

5.3. 接口列表

以下为 median_filter_3x3.v 模块的接口列表

端口	I/O	位宽	描述
clk	input	1	像素时钟
rst_n	input	1	复位信号
vsync_in	input	1	处理前的视频场信号
hsync_in	input	1	处理前的视频行信号
de_in	input	1	数据有效信号
data11	input	8	第 1 行第 1 列的像素数据。
data12	input	8	第 1 行第 2 列的像素数据。
data13	input	8	第 1 行第 3 列的像素数据。
data21	input	8	第 2 行第 1 列的像素数据。
data22	input	8	第 2 行第 2 列的像素数据。

data23	input	8	第 2 行第 3 列的像素数据。
data31	input	8	第 3 行第 1 列的像素数据。
data32	input	8	第 3 行第 2 列的像素数据。
data33	input	8	第 3 行第 3 列的像素数据。
target_data	output	8	3x3 像素窗口的中间值（中值）
vsync_out	output	1	处理后的视频场信号
hsync_out	output	1	处理后的视频行信号
de_out	output	1	处理后的数据有效信号

5.4. 工程说明

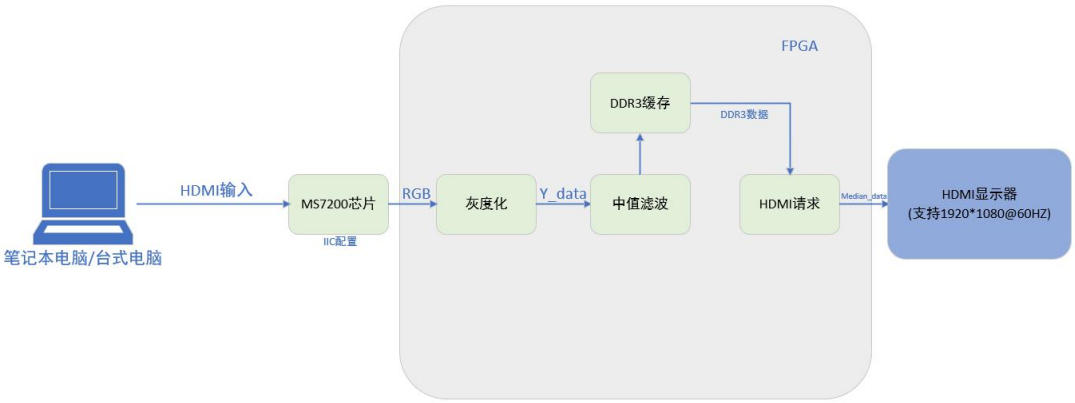


图 5.4-1

上图为本次工程架构，HDMI 输入后经过 MS7200 芯片解码出 RGB 数据，然后经过灰度化后进行中值滤波，然后将数据存到 DDR3 之中，然后再从 DDR3 中读出到 HDMI 上显示。

5.4.1.代码模块说明

本章节主要对中值滤波模块进行说明, 以下是中值滤波 verilog 代码:

```
//中值滤波    delay 3clk

module median_filter_3x3(

    input wire              clk         ,

    input wire              rst_n       ,

    input wire              vsync_in    ,

    input wire              hsync_in    ,

    input wire              de_in       ,



    input wire    [7:0]   data11     ,

    input wire    [7:0]   data12     ,

    input wire    [7:0]   data13     ,

    input wire    [7:0]   data21     ,

    input wire    [7:0]   data22     ,
```

```
    input wire     [7:0]   data23     ,

    input wire     [7:0]   data31     ,

    input wire     [7:0]   data32     ,

    input wire     [7:0]   data33     ,



    output wire    [7:0]   target_data,

    output wire            vsync_out   ,

    output wire            hsync_out   ,

    output wire            de_out

);



//------------------------------------------------------------------------------
---

//FPGA Median Filter Sort order

//         Pixel -- Sort1 -- Sort2 -- Sort3
```

```
// [ P1   P2   P3 ]    [    Max1   Mid1    Min1 ]

// [ P4   P5   P6 ]    [    Max2   Mid2    Min2 ] [Max_min, Mid_mid, Min_max]

mid_valid

// [ P7   P8   P9 ]    [    Max3   Mid3    Min3 ]




//reg define

reg [2:0]    vsync_in_r;

reg [2:0]    hsync_in_r;

reg [2:0]    de_in_r;

//wire define

wire [7:0] max_data1;

wire [7:0] mid_data1;

wire [7:0] min_data1;

wire [7:0] max_data2;

wire [7:0] mid_data2;
```

```
wire [7:0] min_data2;

wire [7:0] max_data3;

wire [7:0] mid_data3;

wire [7:0] min_data3;

wire [7:0] max_min_data;

wire [7:0] mid_mid_data;

wire [7:0] min_max_data;



//*****************************************************

//**                          main code

//*****************************************************



assign vsync_out = vsync_in_r[2];

assign hsync_out   = hsync_in_r[2];

assign de_out = de_in_r[2];
```

```
//Step1  对 stor3 进行三次例化操作
sort3   u_sort3_1(       //第一行数据排序
    .clk        (clk),
    .rst_n      (rst_n),

    .data1      (data11),
    .data2      (data12),
    .data3      (data13),

    .max_data (max_data1),
    .mid_data (mid_data1),
    .min_data (min_data1)
);
```

```
sort3   u_sort3_2(       //第二行数据排序

    .clk        (clk),

    .rst_n      (rst_n),



    .data1      (data21),

    .data2      (data22),

    .data3      (data23),



    .max_data (max_data2),

    .mid_data (mid_data2),

    .min_data (min_data2)

);



sort3   u_sort3_3(       //第三行数据排序

    .clk        (clk),
```

```
    .rst_n     (rst_n),



    .data1     (data31),

    .data2     (data32),

    .data3     (data33),



    .max_data (max_data3),

    .mid_data (mid_data3),

    .min_data (min_data3)

);



//Step2  对三行像素取得的排序进行处理

sort3 u_sort3_4(          //取三行最大值的最小值

    .clk       (clk),

    .rst_n     (rst_n),
```

```
    .data1     (max_data1),
    .data2     (max_data2),
    .data3     (max_data3),

    .max_data (),
    .mid_data (),
    .min_data (max_min_data)
);

sort3 u_sort3_5(          //取三行中值的最小值
    .clk       (clk),
    .rst_n     (rst_n),

    .data1     (mid_data1),
```

```
    .data2     (mid_data2),
    .data3     (mid_data3),



    .max_data (),
    .mid_data (mid_mid_data),
    .min_data ()
);



sort3 u_sort3_6(          //取三行最小值的最大值
    .clk       (clk),
    .rst_n     (rst_n),



    .data1     (min_data1),
    .data2     (min_data2),
    .data3     (min_data3),
```

```
    .max_data (min_max_data),
    .mid_data (),
    .min_data ()
);

//step3  将 step2  中得到的三个值, 再次取中值
sort3 u_sort3_7(
    .clk        (clk),
    .rst_n      (rst_n),

    .data1      (max_min_data),
    .data2      (mid_mid_data),
    .data3      (min_max_data),
```

```
    .max_data (),

    .mid_data (target_data),

    .min_data ()

);



//延迟三个周期进行同步

always@(posedge clk or negedge rst_n)begin

    if(!rst_n)begin

        vsync_in_r <= 0;

        hsync_in_r   <= 0;

        de_in_r <= 0;

    end

    else begin

        vsync_in_r <= {vsync_in_r[1:0],vsync_in};

        hsync_in_r   <= {hsync_in_r [1:0], hsync_in};
```

```
        de_in_r <= {de_in_r[1:0],de_in};

    end

end



endmodule
```

这个模块实现了一个 3x3 的中值滤波器, 用于图像处理。其主要功能是对输入的 3x3 图像矩阵进行中值计算, 并输出处理后的中心像素值。中值滤波是一种常用的去噪技术, 它通过选择局部区域内的中间值来平滑图像, 同时保留图像的边缘细节。

中值滤波器的核心在于排序 3x3 邻域的像素值, 并选择其中的中间值作为输出。中值滤波可以有效去除椒盐噪声, 并且在保持边缘的情况下平滑图像。

代码的第 54-56 行, 延迟了输入同步信号三个时钟周期, 使得这些信号与输出的滤波结果 target_data 对齐。这个延迟通过移位寄存器(vsync_in_r, hsync_in_r, de_in_r)实现。

代码的第 58-96 行,通过三个 sort3 模块,对 3x3 矩阵的每一行进行排序。每个 sort3 模块接收三个像素值, 输出它们的最大值、中间值和最小值。这样可以分别得到每行的最大、中间和最小值（max_data1, mid_data1, min_data1 等）。

代码的第 98-136 行, 使用额外的三个 sort3 模块来处理每一列的最大值、中值和最小值。

第一个 sort3 模块（u_sort3_4）对三行的最大值（max_data1, max_data2, max_data3）进行排序, 提取出这三个最大值中的最小值（max_min_data）。

第二个 sort3 模块 (u_sort3_5) 对三行的中值(mid_data1, mid_data2, mid_data3)进行排序, 提取出这三个中值中的最小值 (mid_mid_data)。

第三个 sort3 模块 (u_sort3_6) 对三行的最小值 (min_data1, min_data2, min_data3)进行排序, 提取出这三个最小值中的最大值 (min_max_data)。

代码的第 138-150 行, 进行最终的中值提取。在前两步中, 我们得到的 max_min_data、mid_mid_data 和 min_max_data 分别是 3x3 矩阵中最大、最小和中间值的集合中的值。最后一步再使用一个 sort3 模块 (u_sort3_7) 来对这三个值进行排序, 并提取中间值(mid_data)作为滤波后的输出 target_data。

代码的第 152-164 行, 通过一个 always 块, vsync_in、hsync_in 和 de_in 信号被延迟了三个时钟周期, 这样确保这些信号与处理后的数据 target_data 同步输出, 避免了数据对齐问题。

5.4.2.代码仿真

5.4.2.1. Matlab 仿真介绍

```
%% 读取图像

originalImage = imread('noise.png');

figure;

%% 显示原图

subplot(1, 3, 1);

imshow(originalImage);

title('原图像');

%% 灰度化

grayImage = rgb2gray(originalImage);

%% 图像尺寸

[img_height, img_width] = size(grayImage);

medianImage = zeros(img_height, img_width);

%% 定义参数
```

```
windowSize = 3; %  窗口大小, 奇数

%% 遍历像素  计算窗口均值

halfWindowSize = floor(windowSize / 2);

for i = 1:img_height

for j = 1:img_width

%  窗口边界

r1 = max(i - halfWindowSize, 1); %窗口上边界

r2 = min(i + halfWindowSize, img_height);%窗口下边界

c1 = max(j - halfWindowSize, 1);%窗口左边界

c2 = min(j + halfWindowSize, img_width);%窗口右边界

%  提取窗口

window = grayImage(r1:r2, c1:c2);

%  计算窗口内的中值

localmedian = median(window(:));

medianImage(i,j) = localmedian;
```

```
end

end

%% 无符号 8bit

medianImage = uint8(medianImage);

%% 将原图像转换为二值图像

binarizedImage = imbinarize(grayImage);



% 显示原图和处理后的图像

subplot(1, 3, 2);

imshow(grayImage);

title('灰度化图像');



subplot(1, 3, 3);

imshow(medianImage);

imwrite(medianImage,'matlab_median.png')
```

```
title('中值滤波图像');
```

代码首先读取要处理的图像并将其转换为灰度图像。

然后初始化一个空的矩阵 filtered_img 来存储滤波后的图像。获取图像的尺寸, 用于设置循环的边界。

代码的 15-30 行是核心部分, 使用双重循环遍历每个像素并应用中值滤波。

双重循环 (for i = 2:rows-1, for j = 2:cols-1): 循环从 2 到 rows-1, 从 2 到 cols-1, 以确保在提取 3x3 邻域时不会超出图像边界。使用这种方式遍历图像的每一个像素。

·提取 3x3 邻域 (window = gray_img(i-1:i+1, j-1:j+1)): window 变量获取以当前像素为中心的 3x3 邻域。MATLAB 中的矩阵切片 (gray_img(i-1:i+1, j-1:j+1)) 用于提取图像矩阵中的子矩阵。

·计算中值 (filtered_img(i, j) = median(window(:))): 将 window 转换为一个列向量 (window(:))。使用 median 函数计算列向量的中值。将中值赋给 filtered_img 的相应位置, 实现了对图像的中值滤波。

最后显示滤波前后的图像, 并将结果保存到文件中。

5.4.2.2. Modelsim 仿真介绍

Tb 文件其实整体框架基本一致, 只需要修改中间的例化模块和最后的有效信号和场信号以及数据即可, 仅给出修改后的部分, 具体波形可以看视频或者读者自行仿真, 只放置最后的 matlab 复原结果。

```
//读取图片数据
video_data_gen#(
    .TOTAL_WIDTH  ( 12'd1650 ),
```

```
    .IMG_WIDTH    ( 12'd1280 ),
    .H_SYNC       ( 12'd40 ),
    .H_BP         ( 12'd220 ),
    .H_FP         ( 12'd110 ),
    .TOTAL_HEIGHT ( 12'd750 ),
    .IMG_HEIGHT   ( 12'd720 ),
    .V_SYNC       ( 12'd5 ),
    .V_BP         ( 12'd20 ),
    .V_FP         ( 12'd5 )
)u_video_data_gen(
    .video_clk    ( video_clk    ),
    .rst_n        ( rst_n        ),
    .video_vs     ( video_vs     ),
    .video_de     ( video_de     ),
    .video_data   ( video_data   )
);

//灰度化
RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n     ),
    .vsync_in ( video_vs  ),
```

```
    .hsync_in  ( video_de  ),
    .de_in     ( video_de     ),
    .red       ( video_data[23:19]   ),
    .green     ( video_data[15:10]   ),
    .blue      ( video_data[7:3]     ),
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out    ( y_de  ),
    .y         ( y_data ),
    .cb        (       ),
    .cr        (       )
);

//矩阵生成
matrix_3x3#(
    .IMG_WIDTH   ( IMG_WIDTH ),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n          ),
    .video_vs    ( y_vs           ),
    .video_de    ( y_de           ),
```

```
    .video_data  ( y_data     ),
    .matrix_de   ( matrix_de   ),
    .matrix11    ( matrix11    ),
    .matrix12    ( matrix12    ),
    .matrix13    ( matrix13    ),
    .matrix21    ( matrix21    ),
    .matrix22    ( matrix22    ),
    .matrix23    ( matrix23    ),
    .matrix31    ( matrix31    ),
    .matrix32    ( matrix32    ),
    .matrix33    ( matrix33    )
);

//中值滤波
median_filter_3x3 u_median_filter_3x3
(
    .clk        ( video_clk       ),
    .rst_n      ( rst_n      ),
    .vsync_in    ( y_vs    ),
    .hsync_in    ( matrix_de    ),
    .de_in      ( matrix_de      ),
    .data11     ( matrix11     ),
```

```
    .data12     ( matrix12     ),
    .data13     ( matrix13     ),
    .data21     ( matrix21     ),
    .data22     ( matrix22     ),
    .data23     ( matrix23     ),
    .data31     ( matrix31     ),
    .data32     ( matrix32     ),
    .data33     ( matrix33     ),
    .target_data ( median_data ),
    .vsync_out   ( median_vs   ),
    .hsync_out   ( median_hs   ),
    .de_out     ( median_de     )
);
```

读取图片数据后经过灰度化模块, 然后通过 3x3 矩阵后再通过中值滤波模块。

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d  <=  1'd0;
    else
        video_vs_d  <=  median_vs;
end


assign frame_flag = ~median_vs & video_vs_d;    //下降沿
```

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end


always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;    //停止仿真
    end
    else if(median_de)    //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",median_data,median_data,median_d
ata);    //16 进制写入
    end
end
```

最后的信号改成中值滤波后的信号即可, 然后开启联合仿真。

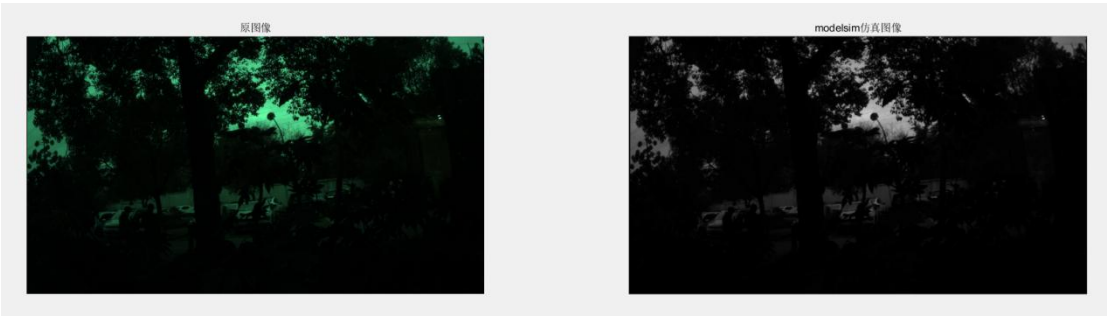


图 5.4-2

5.4.3.实验现象

连接好下载器, 电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器, 然后下载程序。

图 5.4-3

图 5.4-4

上图为测试原图像, 可以看到添加了很多椒盐噪声。

图 5.4-5

上图为中值滤波的处理结构, 对比原图像, 椒盐噪声基本滤除。

6. 均值滤波

6.1. 实验简介

实验目的:

完成图像的均值滤波

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

6.2. 实验原理

6.2.1.均值滤波概念介绍

均值滤波（Mean Filtering）是一种简单的图像处理技术，用于平滑图像和处理噪声。其基本思想是用像素点周围的邻域像素的平均值来代替该像素的值，其原理是使用一个固定大小的窗口（如 3x3 或 5x5）覆盖目标像素及其邻域，计算该区域内所有像素值的均值，并用这个均值替代窗口中心像素的值，再将窗口滑动到图像的每个像素位置重复该操作。

均值滤波简单易实现，计算量小，能够有效平滑图像和减少噪声，但同时也会导致图像细节和边缘模糊，可能会损失某些特征，因此对于需要保留边缘细节的应用，通常会选择更高级的滤波方法。

6.2.2.均值滤波原理介绍

我们以 3x3 大小的区域为例子, 它其实是将这个范围内的 9 个值进行求和的平均值, 以这个新值来代替这个区域的中心值。

我们配合下图进行理解:

图 6.2-1

可以看到 3x3 区域中中间像素值由 40 被替换为 107, 其计算过程为:

$$(197+25+106+107+40+107+163+149+71)/9=107$$

接着移动 3x3 窗口对图片进行逐行逐列进行均值运算, 即可完成图片的均值滤波操作

具体操作过程如下:

假设有一张 8 位灰度图 (其像素值区间在 0~255)进行第一次运算后值 145 替换为 127

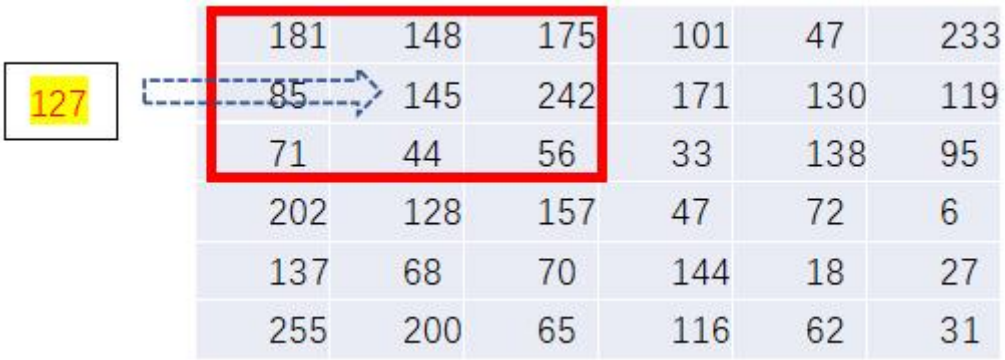


图 6.2-2

第二次运算后 242 被替换为 123:

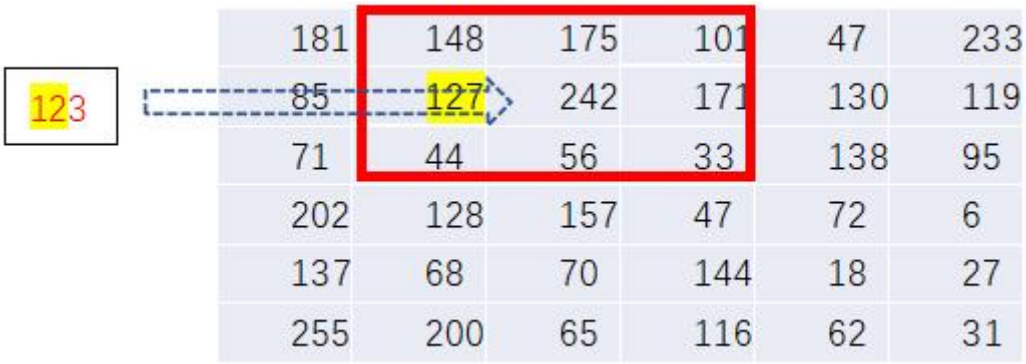


图 6.2-3

以此类推逐行逐列进行均值运算(其实就是使用一个 3x3 卷积核对整张图片进行卷积操作再除以 9, 只不过卷积核为全 1), 最终实现整张图片的滤波操作。

6.3. 接口列表

以下为 aver_filter.v 模块的接口列表

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	复位信号
matrix_de	input	1	矩阵数据输入行信号
matrix_vs	input	1	矩阵数据输入场信号
matrix11	input	8	第一行第一列矩阵数据
matrix12	input	8	第一行第二列矩阵数据
matrix13	input	8	第一行第三列矩阵数据
matrix21	input	8	第二行第一列矩阵数据

matrix22	input	8	第二行第二列矩阵数据
matrix23	input	8	第二行第三列矩阵数据
matrix31	input	8	第三行第一列矩阵数据
matrix32	input	8	第三行第二列矩阵数据
matrix33	input	8	第三行第三列矩阵数据
aver_filter_vs	output	1	均值滤波后数据场信号
aver _filter_de	output	1	均值滤波后数据行信号
aver _filter_data	output	8	均值滤波后数据

6.4. 工程说明

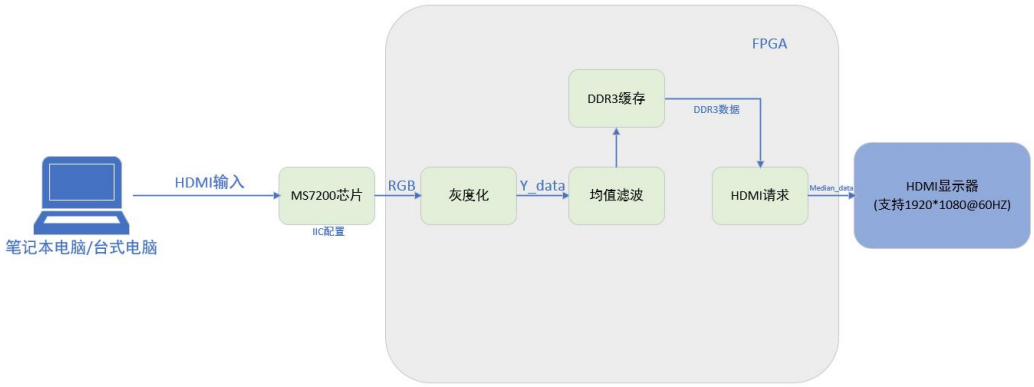


图 6.4-1

上图为本次工程架构, HDMI 输入后经过 MS7200 芯片解码出 RGB 数据, 然后经过灰度化后进行均值滤波, 然后将数据存到 DDR3 之中, 然后再从 DDR3 中读出到 HDMI 上显示。

6.4.1.代码模块说明

本章节主要对均值滤波模块进行说明, 以下是均值滤波 verilog 代码:

```
//均值滤波
module  aver_filter
(
  input   wire        video_clk     /*synthesis PAP_MARK_DEBUG="1"*/,
  input   wire        rst_n         ,

  //矩阵数据输入
  input   wire        matrix_de     /*synthesis PAP_MARK_DEBUG="1"*/,
  input   wire        matrix_vs     /*synthesis PAP_MARK_DEBUG="1"*/,
  input   wire  [7:0] matrix11      ,
  input   wire  [7:0] matrix12      ,
  input   wire  [7:0] matrix13      ,

  input   wire  [7:0] matrix21      ,
  input   wire  [7:0] matrix22      ,
  input   wire  [7:0] matrix23      ,

  input   wire  [7:0] matrix31      ,
  input   wire  [7:0] matrix32      ,
  input   wire  [7:0] matrix33      ,
```

```
    output  wire          aver_filter_vs  ,
    output  wire          aver_filter_de  ,
    output  wire   [7:0]  aver_filter_data

);

/************************************************************
step1 每行相加  delay:1clk
************************************************************/
reg [9:0]   line1_sum;
reg [9:0]   line2_sum;
reg [9:0]   line3_sum;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        line1_sum   <=  10'd0;
        line2_sum   <=  10'd0;
        line3_sum   <=  10'd0;
    end
    else    if(matrix_de)
    begin
```

```
    line1_sum  <=  matrix11 + matrix12 + matrix13  ;
    line2_sum  <=  matrix21 + matrix22 + matrix23  ;
    line3_sum  <=  matrix31 + matrix32 + matrix33  ;
  end
  else
  begin
    line1_sum  <=  10'd0;
    line2_sum  <=  10'd0;
    line3_sum  <=  10'd0;
  end
end

/************************************************************
step2 矩阵总和 delay:1clk
************************************************************/
reg [11:0]  data_sum;
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
    data_sum   <=  12'd0;
  else
    data_sum   <=  line1_sum + line2_sum + line3_sum;
end
```

```
/************************************************************
step3 求均值 /9 delay:1clk
************************************************************/
//除法转乘法   *228>>11
reg [18:0]  aver_filter_mux ;   //均值

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      aver_filter_mux <=   19'd0;
   else
      aver_filter_mux <=   data_sum * 228; //后续要>>11
end



/************************************************************
step4 中心像素点与阈值进行比较   delay:1clk
************************************************************/
reg [7:0]   aver_filter_reg;
always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
```

```
    aver_filter_reg <=  1'd0;
  else
    aver_filter_reg <=  aver_filter_mux[18:11];
end



/************************************************************
时钟延迟 一共延迟 4clk
************************************************************/
reg [3:0]  video_de_reg;
reg [3:0]  video_vs_reg;

always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    video_de_reg   <=  4'd0;
    video_vs_reg   <=  4'd0;
  end
  else
  begin
```

```
    video_de_reg   <=  {video_de_reg[2:0],matrix_de};
    video_vs_reg   <=  {video_vs_reg[2:0],matrix_vs};
  end
end

assign  aver_filter_vs    =  video_vs_reg[3] ;
assign  aver_filter_de    =  video_de_reg[3] ;
assign  aver_filter_data  =  aver_filter_reg ;



endmodule
```

这个模块实现了一个简单的均值滤波器, 用于处理图像矩阵数据。其主要功能是对输入的 3x3 图像矩阵进行均值计算, 并输出处理后的中心像素值。

代码在第 34 - 53 行, 在 matrix_de 为高时 (输入矩阵数据有效时)对每行的 3 个像素值进行相加, 分别存储到 `ine1_sum、`ine2_sum 和 line3_sum 中。

代码在第 58 - 64 行, 将 3 行的和相加, 得到 3x3 矩阵的总和, 并存储在 data_sum 中。注意 data_sum 需要考虑位宽, 例如如果 9 个像素的值都是 255, 那么 data_sum 在计算过程中的最大值为 2295, 转换为二进制为 12 位宽, 为了防止数据溢出`data_sum`位宽声明需要大于 12 位。

为了提高硬件电路的效率, 通常避免直接使用除法操作, 因为除法在硬件中实现相对复

杂且占用资源较多。为了简化运算, 常常将除法转化为乘法和移位的形式。代码在第 70 - 90 行,利用乘法代替除法的方式(*228>>11)来计算均值,将总和乘以 228,然后右移 11 位(即除以 9), 得到的结果存储在 aver_filter_mux 中。

这里使用了除法转乘法的思想, 具体解释如下:

我们知道右移操作在硬件上是快速高效的,对数据右移 1 位相当于将数据除以 2,右移 n 位相当于将数据除以 2^n。要实现 data_sum / 9, 可以找到一个常数 C, 使得 data_sum * C >> n, 接近于 data_sum / 9。这里选择 C = 228, n=11, 这是因为 228/2^11 =0.1113

而 `1/9 ≈ 0.1111`, 这个常数 `228` 近似达到了 `1/9` 的效果。乘以 228 后, 右移 11 位 (>> 11) 来实现除法的效果。移位操作相当于除以 2^{11},这与之前的乘法结合起来实现接近于 除以 9 的运算。

通过乘以 228 然后右移 11 位 (代码中以截取的方式实现, 这两种方式都是可行的), 成功地将原本复杂的除以 9 操作转化为一个较简单的乘法和移位操作。这种方法在硬件设计中非常常见, 可以有效减少运算延迟和硬件资源的使用。

最后代码 98-116 行将 matrix_de 和 matrix_vs 信号延后了一个时钟, 使其与输出数据对齐。video_de_reg 和 video_vs_reg 用于存储这些延迟后的信号。将延迟后的信号和数据输出到 aver_filter_vs、aver_filter_de 和 aver_filter_data。

通过这些步骤, 整个模块实现了对输入的 3x3 图像矩阵进行均值滤波的功能。

6.4.2.代码仿真

6.4.2.1. Matlab 仿真介绍

```
%% 读取图像
originalImage = imread('noise.png');
figure;
```

```
%% 显示原图
subplot(1, 4, 1);
imshow(originalImage);
title('原图像');
%% 灰度化
grayImage = rgb2gray(originalImage);
%% 图像尺寸
[img_height, img_width] = size(grayImage);
medianImage = zeros(img_height, img_width);
%% 定义参数
windowSize = 3;  % 窗口大小, 奇数
%% 遍历像素 计算窗口均值
halfWindowSize = floor(windowSize / 2);
for i = 1:img_height
  for j = 1:img_width
      % 窗口边界
      r1 = max(i - halfWindowSize, 1);    %窗口上边界
      r2 = min(i + halfWindowSize, img_height);%窗口下边界
      c1 = max(j - halfWindowSize, 1);%窗口左边界
      c2 = min(j + halfWindowSize, img_width);%窗口右边界
```

```
        % 提取窗口
        window = grayImage(r1:r2, c1:c2);

        % 计算窗口内的均值  /9 =  (*228) >> 11
        localmedian = sum(window(:))*228.0;

        medianImage(i,j) = localmedian/2048.0;
    end
end
%% 无符号 8bit
medianImage = uint8(medianImage);
%% 将原图像转换为二值图像
binarizedImage = imbinarize(grayImage);

% 显示原图和处理后的图像
subplot(1, 4, 2);
imshow(grayImage);
imwrite(grayImage,'gray.png')
title('灰度化图像');

subplot(1, 4, 3);
imshow(medianImage);
```

```
imwrite(medianImage,'matlab_aver.png')
title('均值滤波图像');

subplot(1, 4, 4);
medianImage = imread('matlab_median.png');
imshow(medianImage);
title('中值滤波图像');
```

代码第 1 行 imread('noise.png');读取名为 noise.png 的图像文件, 并将其存储在变量 originalImage 中。

代码第 8 行 grayImage = rgb2gray(originalImage); 将原始图像转换为灰度图像, 并存储在 grayImage 中。灰度化的目的是减少颜色信息, 仅保留亮度信息以简化后续处理。

代码第 9 行 [img_height, img_width] = size(grayImage); 获取灰度图像的高度和宽度。

代码第 10 行 medianImage = zeros(img_height, img_width); 初始化一个与灰度图像相同大小的矩阵 medianImage, 用于存储均值滤波后的图像。

代码第 12 行 windowSize = 3;定义窗口大小为 3x3, 这是进行均值滤波时的窗口。

代码第 14-23 行 halfWindowSize = floor(windowSize / 2);计算窗口的一半大小, 用于确定窗口边界。使用双重循环遍历每个像素, 计算每个像素的均值滤波值。r1 和 r2 分别确定窗口的上边界和下边界。c1 和 c2 分别确定窗口的左边界和右边界。window = grayImage(r1:r2, c1:c2);提取当前像素点周围的窗口区域。

代码第 26 行 localmedian = sum(window(:))*228.0; 计算窗口内像素值的总和并乘以 228, 这是用于验证 verilog 中近似计算均值的方法。

代码第 27 行 medianImage(i,j) = localmedian/2048.0; 将计算结果除以 2048(模拟右移 11 位), 得到最终的均值滤波后的像素值, 并存储在 medianImage 中。medianImage = uint8(medianImage);`将滤波后的图像矩阵转换为 8 位无符号整数格式, 用于图像显示。

通过这些步骤, 实现了对图像进行灰度化、均值滤波处理, 并展示原图像、灰度图像、均值滤波图像以及中值滤波图像。

6.4.2.2. Modelsim 仿真介绍

Tb 文件其实整体框架基本一致, 只需要修改中间的例化模块和最后的有效信号和场信号以及数据即可, 仅给出修改后的部分, 具体波形可以看视频或者读者自行仿真, 只放置最后的 matlab 复原结果。

```
//读取图片数据
video_data_gen#(
    .TOTAL_WIDTH  ( 12'd1650 ),
    .IMG_WIDTH    ( 12'd1280 ),
    .H_SYNC       ( 12'd40 ),
    .H_BP         ( 12'd220 ),
    .H_FP         ( 12'd110 ),
    .TOTAL_HEIGHT ( 12'd750 ),
    .IMG_HEIGHT   ( 12'd720 ),
    .V_SYNC       ( 12'd5 ),
    .V_BP         ( 12'd20 ),
    .V_FP         ( 12'd5 )
)u_video_data_gen(
```

```
    .video_clk   ( video_clk   ),
    .rst_n       ( rst_n       ),
    .video_vs    ( video_vs    ),
    .video_de    ( video_de    ),
    .video_data  ( video_data  )
);

//灰度化
RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n     ),
    .vsync_in ( video_vs  ),
    .hsync_in ( video_de  ),
    .de_in    ( video_de  ),
    .red      ( video_data[23:19]  ),
    .green    ( video_data[15:10]  ),
    .blue     ( video_data[7:3]    ),
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out   ( y_de  ),
    .y        ( y_data ),
    .cb       (      ),
```

```
    .cr      (      )
);

//矩阵生成
matrix_3x3#(
    .IMG_WIDTH   ( IMG_WIDTH ),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n          ),
    .video_vs    ( y_vs           ),
    .video_de    ( y_de           ),
    .video_data  ( y_data         ),
    .matrix_de   ( matrix_de      ),
    .matrix11    ( matrix11       ),
    .matrix12    ( matrix12       ),
    .matrix13    ( matrix13       ),
    .matrix21    ( matrix21       ),
    .matrix22    ( matrix22       ),
    .matrix23    ( matrix23       ),
    .matrix31    ( matrix31       ),
    .matrix32    ( matrix32       ),
```

```
    .matrix33   ( matrix33   )
);
//均值滤波
aver_filter u_aver_filter(
    .video_clk      ( video_clk      ),
    .rst_n          ( rst_n          ),
    .matrix_de      ( matrix_de      ),
    .matrix_vs      ( y_vs      ),
    .matrix11       ( matrix11       ),
    .matrix12       ( matrix12       ),
    .matrix13       ( matrix13       ),
    .matrix21       ( matrix21       ),
    .matrix22       ( matrix22       ),
    .matrix23       ( matrix23       ),
    .matrix31       ( matrix31       ),
    .matrix32       ( matrix32       ),
    .matrix33       ( matrix33       ),
    .aver_filter_vs  ( aver_filter_vs  ),
    .aver_filter_de  ( aver_filter_de  ),
    .aver_filter_data  ( aver_filter_data  )
);
```

读者可以看到 tb 结构一九和中值滤波一致, 读取图片数据, 灰度化, 生成矩阵是一致的,

只不过将中值滤波模块换成了均值滤波模块, 并把后续的信号也换成均值滤波后的信号即可,

如下所示:

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d  <=   1'd0;
    else
        video_vs_d  <=   aver_filter_vs;
end

assign frame_flag = ~aver_filter_vs & video_vs_d;    //下降沿

always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end


always@(posedge video_clk or negedge rst_n) begin
```

```
    if(img_done)
    begin
        $stop;   //停止仿真
    end
    else if(aver_filter_de)   //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",aver_filter_data,aver_filter_data,aver_
filter_data);   //16 进制写入
    end
end
```

最后放到 matlab 中复原, 如下所示:

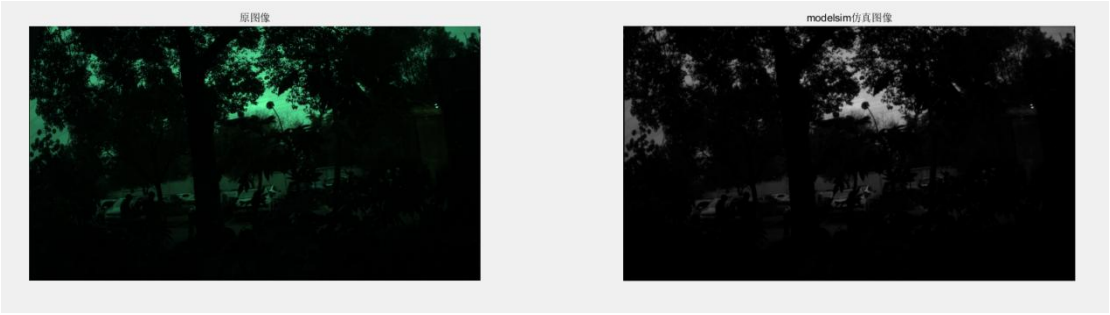


图 6.4-2

6.4.3.实验现象

连接好下载器, 电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器, 然后下载程序。

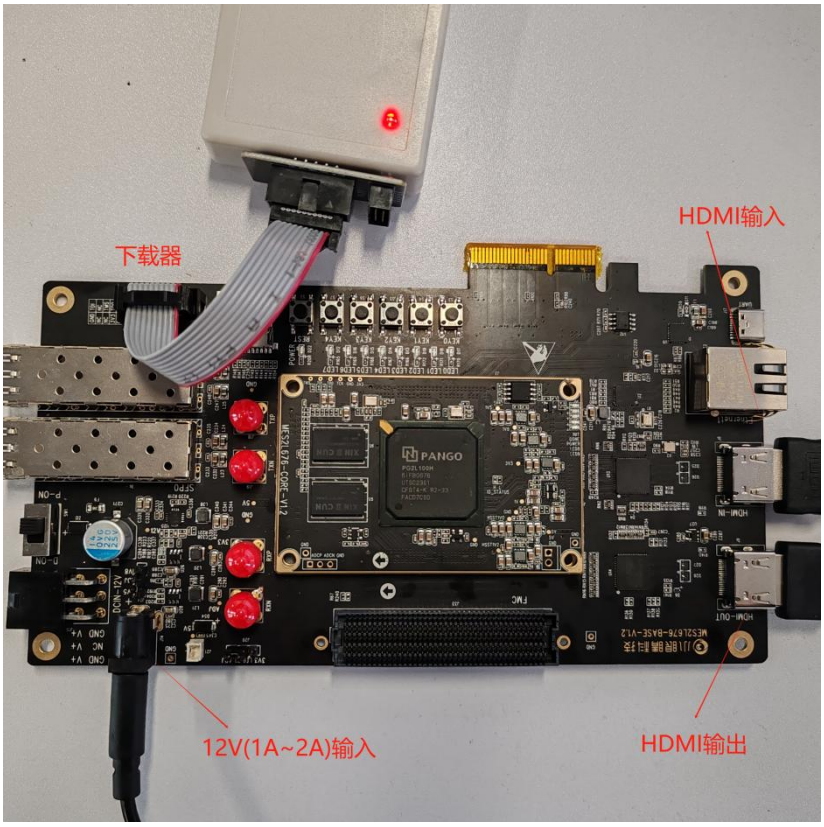


图 6.4-3

图 6.4-4

上图为添加了椒盐噪声的图像。

图 6.4-5

可以看到椒盐噪声变的平滑, 比原图像有所改善, 但效果比中值滤波差, 均值滤波主要让图像平滑, 无法过滤掉椒盐噪声。

7. 高斯滤波

7.1. 实验简介

实验目的:

生成 3x3 矩阵, 完成高斯滤波。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

7.2. 实验原理

7.2.1.高斯滤波概念介绍

高斯噪声指的是概率密度函数服从高斯分布（即正态分布）的一类噪声。如果一个噪声，它的幅度分布服从高斯分布，而它的功率谱密度又是均匀分布的，则称它为高斯白噪声。在图像处理领域中，由于环境温度，不良光照等等原因可能使得相机 cmos 采集到的信号引入高斯噪声。

图 7.2-1

左图为正常灰度图, 右图为添加了高斯噪声的灰度图

高斯滤波是一种常用的线性平滑滤波, 适用于消除高斯噪声。 通俗的讲, 高斯滤波就是通过设计卷积核, 通过卷积运算对整幅图像进行加权平均的过程, 每一个像素点的值, 都由其本身和邻域内的其他像素值经过加权平均后得到。

对于均值滤波和中值滤波来说, 其进行矩阵划分的每个像素的权重是相等的。而在高斯滤波中, 会将中心点的权重值加大, 远离中心点的权重值减小, 在此基础上计算邻域内各个像素值不同权重的和。

7.2.2.高斯滤波卷积核设计

首先介绍二维高斯函数:

$$G(x,y)=\frac{1}{2\pi\sigma_x\sigma_y}\exp(-(\frac{(x-u_x)^2}{2\sigma_x^2}+\frac{(y-u_y)^2}{2\sigma_y^2}))$$

这是二维高斯概率密度分布函数, 其表示随机变量在二维平面服从高斯分布, $G(x,y)$ 代表二维平面上相应坐标点的概率密度,x 表示二维平面中的 x 轴方向,其中 y 表示二维平面中的 y 轴方向, u_x , u_y 分别为 x 方向, y 方向随机变量的数学期望, σ_x , σ_y 分别为 x 方向, y 方向随机变量的方差。

假定卷积核中心点坐标为 (0, 0), 那么它相邻的八个点坐标如下:

(-1, 1)	(0, 1)	(1, 1)
(-1, 0)	(0, 0)	(1, 0)
(-1, -1)	(0, -1)	(1, -1)

选取随机变量数学期望 u =0, 方差 σ =0.8 将坐标带入二维高斯概率密度分布函数得:

0.05212607	0.11385381	0.05212607

0.11385381	0.24867959	0.11385381
0.05212607	0.11385381	0.05212607

可以观察出中心概率密度最大,且概率密度沿四周逐渐减小,接下来对其进行归一化(卷积核的所有权重值都需要缩放到相同的范围内,并且总和为 1,以保证卷积操作对图像的影响是平衡的,避免卷积过后改变原来图像的亮度,使之偏亮或者偏暗):

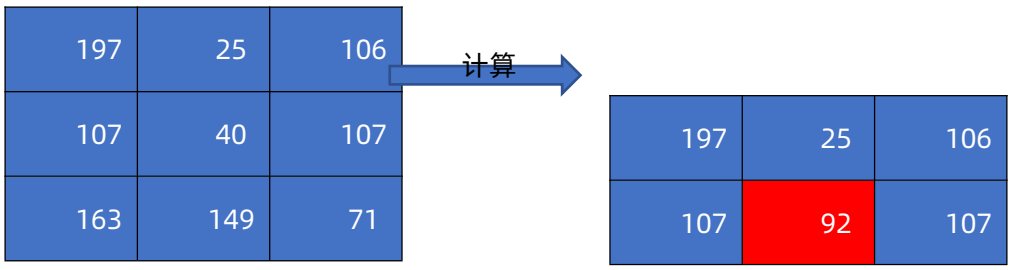
0.05711825	0.12475774	0.05711825
0.12475774	0.27249597	0.12475774
0.05711825	0.12475774	0.05711825

最后由于在 FPGA 中对小数的运算较为复杂,这里将卷积核扩大 16 倍取近似的整数,FPGA 卷积完成后再将数据缩小 16 倍,以达到避开浮点数运算的目的,近似后的卷积核为:

1	2	1
2	4	2
1	2	1

7.2.3.高斯滤波 FPGA 实现

与均值滤波相同只是使用了不同的 3x3 卷积核对图片逐行逐列进行卷积操作再除以 16。我们配合下图来理解,假设如下左图是一个 3x3 的像素点区域,使用我们设计的卷积核进行高斯滤波计算后,中间的值被替换为 92:



具体计算过程为:

163	149	71

$$\left(\left(197+106+163+71\right)+\left(25+107+107+149\right)*2+40*4\right)\Big/16=92$$

然后使用这个方式对整张图片进行逐行逐列的操作, 即可完成对图片的高斯滤波操作

7.3. 接口列表

以下为 gauss_filter.v 模块的接口列表。

端口	I/O	位宽	描述
video_clk	input	1	像素时钟
rst_n	input	1	复位信号
matrix_de	input	1	矩阵数据输入行信号
matrix_vs	input	1	矩阵数据输入场信号
matrix11	input	8	第一行第一列矩阵数据
matrix12	input	8	第一行第二列矩阵数据
matrix13	input	8	第一行第三列矩阵数据
matrix21	input	8	第二行第一列矩阵数据
matrix22	input	8	第二行第二列矩阵数据
matrix23	input	8	第二行第三列矩阵数据
matrix31	input	8	第三行第一列矩阵数据
matrix32	input	8	第三行第二列矩阵数据

matrix33	input	8	第三行第三列矩阵数据
gauss_filter_vs	output	1	高斯滤波后数据场信号
gauss_filter_de	output	1	高斯滤波后数据行信号
gauss_filter_data	output	8	高斯滤波后数据

7.4. 工程说明

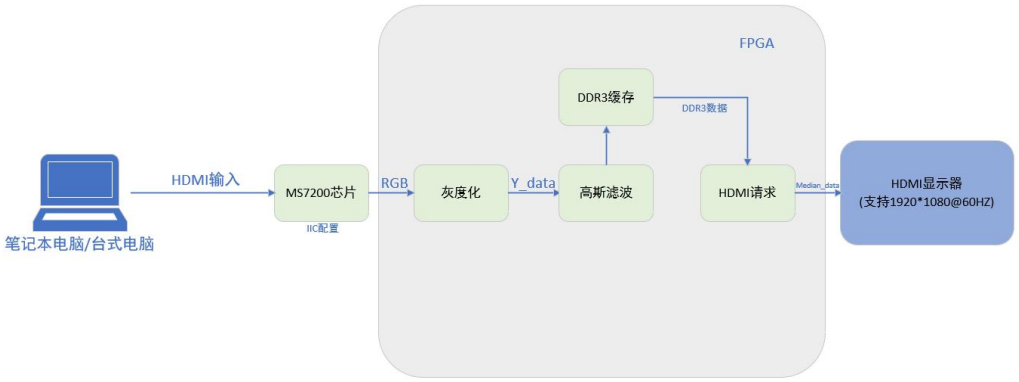


图 7.4-1

7.4.1.代码模块说明

本工程与前置章节工程构造相同, 唯一区别是更换了卷积核, 详细说明请参考前置章节

```
//高斯滤波
module  gauss_filter
(
  input   wire        video_clk     ,
  input   wire        rst_n         ,

  //矩阵数据输入
```

```
    input   wire          matrix_de     ,
    input   wire          matrix_vs     ,
    input   wire   [7:0]  matrix11      ,
    input   wire   [7:0]  matrix12      ,
    input   wire   [7:0]  matrix13      ,

    input   wire   [7:0]  matrix21      ,
    input   wire   [7:0]  matrix22      ,
    input   wire   [7:0]  matrix23      ,

    input   wire   [7:0]  matrix31      ,
    input   wire   [7:0]  matrix32      ,
    input   wire   [7:0]  matrix33      ,

    output  wire          gauss_filter_vs ,
    output  wire          gauss_filter_de ,
    output  wire   [7:0]  gauss_filter_data

);
/*
高斯核
| 1 2 1 |
```

```
| 2 4 2 |      sum 高斯核 = 16  ->  sum/16=sum>>4
| 1 2 1 |
*/

/************************************************************
step1 每行相加  delay:1clk
************************************************************/
reg [9:0]   line1_sum;
reg [9:0]   line2_sum;
reg [9:0]   line3_sum;
always@(posedge video_clk or negedge rst_n) begin
  if(!rst_n)
  begin
    line1_sum   <=  10'd0;
    line2_sum   <=  10'd0;
    line3_sum   <=  10'd0;
  end
  else  if(matrix_de)
  begin
    line1_sum   <=  matrix11 + matrix12<<1 + matrix13 ;
    line2_sum   <=  matrix21<<1 + matrix22<<2 + matrix23<<1  ;
    line3_sum   <=  matrix31 + matrix32<<1 + matrix33 ;
```

```
    end
    else
    begin
        line1_sum  <=  10'd0;
        line2_sum  <=  10'd0;
        line3_sum  <=  10'd0;
    end
end

/************************************************************
step2 矩阵总和 delay:1clk
************************************************************/
reg [11:0]  data_sum;
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        data_sum   <=  12'd0;
    else
        data_sum   <=  line1_sum + line2_sum + line3_sum;
end

/************************************************************
step3 右移 4 位 delay:1clk
```

```
************************************************************/
//移位实现/16
reg [7:0]   gauss_filter_reg    ;   //均值

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      gauss_filter_reg    <=   8'd0;
   else
      gauss_filter_reg    <=   data_sum[11:4]; //
end


/************************************************************
时钟延迟 一共延迟 3clk
************************************************************/
reg [2:0]   video_de_reg;
reg [2:0]   video_vs_reg;

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
   begin
      video_de_reg    <=   3'd0;
```

```
    video_vs_reg   <=  3'd0;
  end
  else
  begin
    video_de_reg   <=  {video_de_reg[1:0],matrix_de};
    video_vs_reg   <=  {video_vs_reg[1:0],matrix_vs};
  end
end

assign  gauss_filter_vs    =  video_vs_reg[2] ;
assign  gauss_filter_de    =  video_de_reg[2] ;
assign  gauss_filter_data  =  gauss_filter_reg   ;


endmodule
```

该模块接受了 3x3 图像矩阵数据并使用三级流水线思想对矩阵数据进行了卷积操作。这样的流水线思想有助于改善时序, 提高时钟频率。

代码第 37 行到第 60 行进行了第一级流水线操作, 消耗了一个时钟。声明 line1_sum, line2_sum, line3_sum 在 matrix_de 信号有效时对矩阵数据按行与卷积核做移位操作再相加。这里的乘法使用移位的方式实现, 有助于减少资源消耗, 改善时序状况。在进行小规模乘法除法操作时, 使用移位的方式是常用的方法。

代码 61 行到 70 行进行了第二级流水线操作。消耗一个时钟周期将 line1_sum, line2_sum, line3_sum 的值相加后赋值给 data_sum。

代码 78 行到 83 行进行了第三级流水线操作,消耗一个时钟周期将 data_sum 用截取高位的方式将 data_sum 赋值给 gauss_filter_reg。

最后声明了中间寄存器使用移位拼接的方式对矩阵数据的行场同步进行延迟操作, 一共延迟了 3 个时钟。使得输出的高斯滤波数据与延迟后的行场同步信号相匹配。整个模块代码简洁高效。

7.4.2.代码仿真

7.4.2.1. Matlab 仿真介绍

```
clc;clear all;

%% 读取图像

originalImage = imread('noise.png');

figure;

%% 定义高斯核

gaussianKernel = [1, 2, 1; 2, 4, 2; 1, 2, 1];%和为 16

%% 显示原图

subplot(1, 3, 1);

imshow(originalImage);
```

```
title('原图像');

%% 灰度化

grayImage = rgb2gray(originalImage);

%% 图像尺寸

[img_height, img_width] = size(grayImage);

medianImage = zeros(img_height, img_width);

%% 定义参数

windowSize = 4;  % 窗口大小 实际大小

%% 遍历像素 计算窗口均值

halfWindowSize = floor(windowSize / 2);

for i = 1:img_height

  for j = 1:img_width

    % 窗口边界

    r1 = max(i - halfWindowSize, 1);    %窗口上边界

    r2 = min(i + halfWindowSize, img_height);%窗口下边界
```

```
        c1 = max(j - halfWindowSize, 1);%窗口左边界

        c2 = min(j + halfWindowSize, img_width);%窗口右边界



        % 提取窗口

        window = double(grayImage(r1:r2, c1:c2));



        % 计算窗口内的卷积

        sum_gauss = window(1,1)*gaussianKernel(1,1)+window(1,2)*gaussianKer
nel(1,2)+window(1,3)*gaussianKernel(1,3)+...

                window(2,1)*gaussianKernel(2,1)+window(2,2)*gaussianKernel(2,2)
+window(2,3)*gaussianKernel(2,3)+...

                window(3,1)*gaussianKernel(3,1)+window(3,2)*gaussianKernel(3,2)
+window(3,3)*gaussianKernel(3,3);

        medianImage(i,j) = sum_gauss/16;

    end

end
```

```
%% 无符号 8bit

medianImage = uint8(medianImage);

%% 将原图像转换为二值图像

binarizedImage = imbinarize(grayImage);



% 显示原图和处理后的图像

subplot(1, 3, 2);

imshow(grayImage);

imwrite(grayImage,'gray.png')

title('灰度化图像');



subplot(1, 3, 3);

imshow(medianImage);

imwrite(medianImage,'matlab_gauss.png')

title('高斯滤波图像');
```

代码代码的第 3 行使用 imread()函数读取了一张图片并使用 rgb2gray()番薯将其转换为灰度图像。

代码第 6 行定义了一个 3x3 的高斯核, 用于卷积操作。

代码第 20 行使用 for 循环遍历灰度图像的每个像素, 通过一个定义好的窗口, 对该窗口内的像素应用高斯核进行加权求和, 得到滤波后的像素值。

滤波后的像素值存储在 medianImage 矩阵中, 并通过将结果除以 16(因为高斯核的和为 16)来归一化处理。最后, 该处理后的图像转换为 8 位无符号整数类型, 并与原始的灰度图像和二值化后的图像一同显示。

整个流程包括: 读取图像、灰度化、定义卷积窗口、计算卷积、归一化处理、图像类型转换、图像显示以及保存处理后的图像。

7.4.2.2. Modelsim 仿真介绍

Tb 文件其实整体框架基本一致, 只需要修改中间的例化模块和最后的有效信号和场信号以及数据即可, 仅给出修改后的部分, 具体波形可以看视频或者读者自行仿真, 只放置最后的 matlab 复原结果。

```
//读取图片数据
video_data_gen#(
    .TOTAL_WIDTH  ( 12'd1650 ),
    .IMG_WIDTH    ( 12'd1280 ),
    .H_SYNC       ( 12'd40 ),
    .H_BP         ( 12'd220 ),
    .H_FP         ( 12'd110 ),
    .TOTAL_HEIGHT ( 12'd750 ),
```

```
    .IMG_HEIGHT   ( 12'd720 ),
    .V_SYNC       ( 12'd5 ),
    .V_BP         ( 12'd20 ),
    .V_FP         ( 12'd5 )
)u_video_data_gen(
    .video_clk    ( video_clk    ),
    .rst_n        ( rst_n        ),
    .video_vs     ( video_vs     ),
    .video_de     ( video_de     ),
    .video_data   ( video_data   )
);

//灰度化
RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n     ),
    .vsync_in  ( video_vs  ),
    .hsync_in  ( video_de  ),
    .de_in    ( video_de     ),
    .red      ( video_data[23:19]   ),
    .green    ( video_data[15:10]   ),
    .blue     ( video_data[7:3]     ),
```

```
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out    ( y_de  ),
    .y         ( y_data ),
    .cb        (       ),
    .cr        (       )
);

//矩阵生成
matrix_3x3#(
    .IMG_WIDTH   ( IMG_WIDTH ),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n          ),
    .video_vs    ( y_vs           ),
    .video_de    ( y_de           ),
    .video_data  ( y_data         ),
    .matrix_de   ( matrix_de   ),
    .matrix11    ( matrix11    ),
    .matrix12    ( matrix12    ),
    .matrix13    ( matrix13    ),
```

```
    .matrix21    ( matrix21    ),
    .matrix22    ( matrix22    ),
    .matrix23    ( matrix23    ),
    .matrix31    ( matrix31    ),
    .matrix32    ( matrix32    ),
    .matrix33    ( matrix33    )
);
//高斯滤波
gauss_filter u_gauss_filter(
    .video_clk       ( video_clk       ),
    .rst_n           ( rst_n           ),
    .matrix_de       ( matrix_de       ),
    .matrix_vs       ( y_vs       ),
    .matrix11        ( matrix11        ),
    .matrix12        ( matrix12        ),
    .matrix13        ( matrix13        ),
    .matrix21        ( matrix21        ),
    .matrix22        ( matrix22        ),
    .matrix23        ( matrix23        ),
    .matrix31        ( matrix31        ),
    .matrix32        ( matrix32        ),
    .matrix33        ( matrix33        ),
```

```
    .gauss_filter_vs  ( gauss_filter_vs  ),
    .gauss_filter_de  ( gauss_filter_de  ),
    .gauss_filter_data  ( gauss_filter_data  )
);
```

Tb 的结构依旧和之前保持一致, 将对应的滤波模块换成高斯滤波即可。

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d  <=  1'd0;
    else
        video_vs_d  <=  gauss_filter_vs;
end

assign frame_flag = ~gauss_filter_vs & video_vs_d;    //下降沿

always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end
```

```
always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;   //停止仿真
    end
    else if(gauss_filter_de)   //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",gauss_filter_data,gauss_filter_data,ga
uss_filter_data);   //16 进制写入
    end
end
```

最后处理后的 txt 文件在 matlab 中复原, 如下所示:

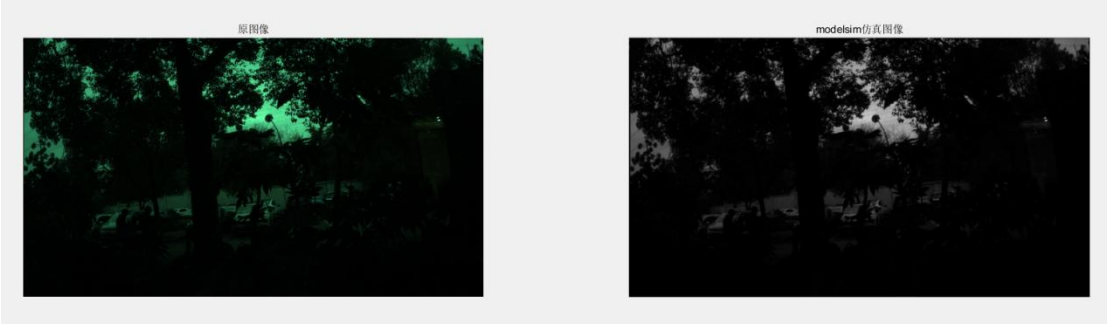


图 7.4-2

7.4.3.实验现象

连接好下载器, 电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器.

图 7.4-3

图 7.4-4

上图为添加了椒盐噪声的图像。

图 7.4-5

可以看到高斯滤波也让图像平滑、降噪，主要用在边缘检测、模糊处理时的预处理。其也是一种低通滤波，可以除去高频噪声。

8. 边沿检测

8.1. 实验简介

实验目的:

在图像灰度化的基础上完成基于 sobel 算子的边沿检测。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

8.2. 实验原理

sobel 算子是广泛应用的微分算子之一,可以计算图像处理中的边缘检测,计算图像的灰度地图。在技术上, 它是一个离散的一阶差分算子, 用来计算图像亮度函数的一阶梯度之近似值。在图像的任何一点使用此算子, 将会产生该点对应的梯度矢量或是其法矢量原理就是基于图像的卷积来实现在水平方向与垂直方向检测对于方向上的边缘。

水平方向的梯度算子的计算公式为:

$$G_x = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix} * A$$

垂直方向的梯度算子的计算公式为:

$$G_y = \begin{vmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{vmatrix} * A$$

最后再把水平方向的算子和垂直方向的算子平方相加再开根号即可得到结果。如下所示:

$$G = \sqrt{G_x^2 + G_y^2}$$

平方的操作在 fpga 中还是比较容易实现的,开根号的话一般使用 cordic IP 来完成 sqrt 操作, 但紫光的软件中没有提供这种 IP 核, 因此需要手写一个 sqrt 的模块。但是我们将该公式变换一下, 根据不等式(有条件限制, 可以做近似计算)可以换成以下公式:

$$G = |G_X + G_Y|$$

可以大大降低所消耗的资源, 当然最后的误差可能会比较大, 可以通过调节阈值来弥补, 追求完美的效果的话, 还是需要使用 cordic 来完成, 或者开源的 sart 代码来完成。

最后计算得到的结果 G 与阈值比较,大于阈值视为边沿,可以将其像素值置为 0 或者 255, 取决于想把边沿定位什么颜色, 以 8bit 灰度数据为例子, 255 就是白色, 0 就是黑色。

8.3. 接口列表

sobel.v 模块接口列表

端口	I/O	位宽	描述
SOBEL_THRESHOLD	/	8	sobel 阈值
video_clk	input	1	像素时钟
rst_n	input	1	系统复位
matrix_de	input	1	矩阵数据有效信号
matrix_vs	input	1	视频场信号
matrix11	input	8	3X3 矩阵第 1 个 8bit 灰度像素
matrix12	input	8	3X3 矩阵第 2 个 8bit 灰度像素

matrix13	input	8	3X3 矩阵第 3 个 8bit 灰度像素
matrix21	input	8	3X3 矩阵第 4 个 8bit 灰度像素
matrix22	input	8	3X3 矩阵第 5 个 8bit 灰度像素
matrix23	input	8	3X3 矩阵第 6 个 8bit 灰度像素
matrix31	input	8	3X3 矩阵第 7 个 8bit 灰度像素
matrix32	input	8	3X3 矩阵第 8 个 8bit 灰度像素
matrix33	input	8	3X3 矩阵第 9 个 8bit 灰度像素
sobel_vs	output	1	输出 sobel 处理后的场信号
sobel_de	output	1	输出 sobel 处理后的数据有效信号
sobel_data	output	8	输出 sobel 处理后的数据

8.4. 工程说明

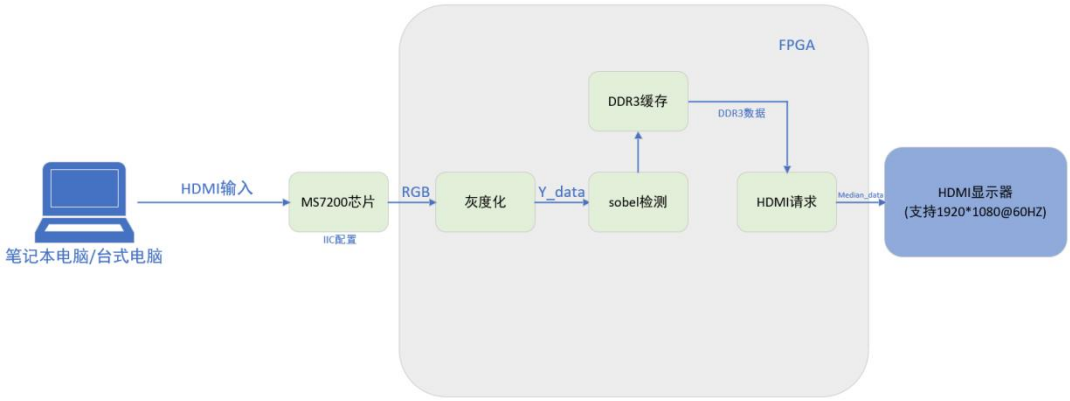


图 8.4-1

该工程架构与之前的工程基本一致, 只是在灰度化后, 新增了 sobel 处理模块, 然后再将

数据缓存到 DDR3 之中, 然后再把数据从 DDR3 中读出到 HDMI 显示器上显示.

8.4.1.代码模块说明

主要介绍 sobel 算子模块,如下所示:

```
module  sobel
#(
   parameter   SOBEL_THRESHOLD = 64
)
(
   input     wire          video_clk     ,
   input     wire          rst_n         ,

   //矩阵数据输入
   input     wire          matrix_de     ,
   input     wire          matrix_vs     ,
   input     wire  [7:0]     matrix11      ,
   input     wire  [7:0]     matrix12      ,
   input     wire  [7:0]     matrix13      ,

   input     wire  [7:0]     matrix21      ,
   input     wire  [7:0]     matrix22      ,
   input     wire  [7:0]     matrix23      ,

```

```
    input     wire   [7:0]     matrix31      ,
    input     wire   [7:0]     matrix32      ,
    input     wire   [7:0]     matrix33      ,
    //sobel 数据输出
    output    wire             sobel_vs      ,
    output    wire             sobel_de      ,
    output    wire   [7:0]     sobel_data
);

/*****************************************************************
%    -1  0 +1   %    +1  2 +1
% gx = -2  0 +2   % gy =  0  0 0
%    -1  0 +1   %    -1 -2 -1
*****************************************************************/

/*****************************************************************
wire define
*****************************************************************/

/*****************************************************************
reg define
*****************************************************************/

reg [9:0]   gx_temp1;
reg [9:0]   gx_temp2;
```

```
reg [9:0]   gy_temp1;
reg [9:0]   gy_temp2;

/*************************************************************
step1 计算卷积
*************************************************************/
//水平方向
always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
   begin
      gx_temp1   <=  9'd0;
      gx_temp2   <=  9'd0;
   end
   else   if(matrix_de)
   begin
      gx_temp1   <=  matrix13 + 2*matrix23 + matrix33;
      gx_temp2   <=  matrix11 + 2*matrix21 + matrix31;
   end
   else
   begin
      gx_temp1   <=  9'd0;
      gx_temp2   <=  9'd0;
```

```
    end
end

//竖直方向
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
    begin
        gy_temp1   <=  9'd0;
        gy_temp2   <=  9'd0;
    end
    else   if(matrix_de)
    begin
        gy_temp1   <=  matrix31 + 2*matrix32 + matrix33;
        gy_temp2   <=  matrix11 + 2*matrix12 + matrix13;
    end
    else
    begin
        gy_temp1   <=  9'd0;
        gy_temp2   <=  9'd0;
    end
end
```

```
/*************************************************************
step2 求卷积和
*************************************************************/
reg [9:0]   gx_data;
reg [9:0]   gy_data;


always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      gx_data <=   10'd0;
   else    if(gx_temp1 >= gx_temp2)
      gx_data <=   gx_temp1 - gx_temp2;
   else
      gx_data <=   gx_temp2 - gx_temp1;
end


always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      gy_data <=   10'd0;
   else    if(gy_temp1 >= gy_temp2)
      gy_data <=   gy_temp1 - gy_temp2;
   else
      gy_data <=   gy_temp2 - gy_temp1;
```

```
end

/*************************************************************
step3 绝对值相加
*************************************************************/
reg [11:0]  sobel_data_reg;

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
      sobel_data_reg  <=  12'd0;
   else
      sobel_data_reg  <=  gx_data + gy_data;
end

/*************************************************************
时钟延迟 一共延迟 3clk
*************************************************************/
reg [2:0]   video_de_reg;
reg [2:0]   video_vs_reg;

always@(posedge video_clk or negedge rst_n) begin
   if(!rst_n)
```

```
    begin
        video_de_reg    <=  3'd0;
        video_vs_reg    <=  3'd0;
    end
    else
    begin
        video_de_reg    <=  {video_de_reg[1:0],matrix_de};
        video_vs_reg    <=  {video_vs_reg[1:0],matrix_vs};
    end
end

assign  sobel_vs      =  video_vs_reg[2]       ;
assign  sobel_de      =  video_de_reg[2]       ;
assign  sobel_data    =  (sobel_data_reg>SOBEL_THRESHOLD)?8'd0:8'd255 ;

endmodule
```

同样的,依旧是使用三级流水线来完成本次算法。

第一级流水线,代码的 50-66 行和 69-85 行按照卷积的定义,将这 9 个数据与对应的系数相乘。

第二级流水线,代码的 93-100 行和 102-109 行将相乘后的结果进行相加减,此时需要注意一个点,涉及到减法时,要避免负数的出现,一旦出现小减大会导致出现负数,而定义的都是无符号数,会导致结果出现很大的误差,所以我们先比较大小,避免出现负数,并且,这样处理的

同时也完成了取绝对值的操作。

第三级流水线,代码的 116-121 行,将水平方向的算子和垂直方向的算子相加, 此时相当于绝对值相加了, 因为第二级已经做了绝对值处理。

本次算法从数据输入到结果输出需要 3 个 clk, 故在代码的 129-140 行对数据有效信号和场信号打 3 拍, 延迟 3 个 clk, 让其与数据保持同步。这里打拍通过移位寄存器进行打拍。

2.1.3.2 代码仿真

2.1.3.2.1Matlab 仿真介绍

```
for i = 1:img_height
  for j = 1:img_width
    % 窗口边界
    r1 = max(i - halfWindowSize, 1);   %窗口上边界
    r2 = min(i + halfWindowSize, img_height);%窗口下边界
    c1 = max(j - halfWindowSize, 1);%窗口左边界
    c2 = min(j + halfWindowSize, img_width);%窗口右边界

    % 提取窗口
    window = double(grayImage(r1:r2, c1:c2));

    % 计算窗口 GX GY
    GX = window(1,3)+2*window(2,3)+window(3,3)-window(1,1)-2*window(2,
1)-window(3,1);
```

```
        GY = window(1,1)+2*window(1,2)+window(1,3)-window(3,1)-2*window(3,
2)-window(3,3);
        %计算 G
        G= sqrt(GX^2+GY^2);
        %G=abs(GX)+abs(GY);
        if G > thre %比阈值大 视为边沿
            sobelImage(i,j) = 0;
        else
            sobelImage(i,j) = 255;
        end
    end
end
```

Matlab 生成矩阵的方法这里不再做详细介绍, 主要看代码的 13-14 行, 计算 GX 和 GY, 可以看到也是按照卷积的运算法则, 将对应数据乘上对应系数然后相加。然后看 16-17 行, 这里提供了两种方法,16 行就是正常的计算,平方相加然后开根号。17 行就和我们在 verilog 中计算的方法一致。最后与阈值比较, 比阈值大, 视为边沿, 视为黑色。

2.1.3.2.2Modelsim 仿真介绍

依旧是直接的 tb 文件, 然后添加了 sobel 处理模块, 如下所示:

```
//灰度化
RGB2YCbCr u_RGB2YCbCr(
    .clk      ( video_clk ),
    .rst_n    ( rst_n     ),
```

```
    .vsync_in  ( video_vs  ),
    .hsync_in  ( video_de  ),
    .de_in     ( video_de     ),
    .red       ( video_data[23:19]   ),
    .green     ( video_data[15:10]   ),
    .blue      ( video_data[7:3]     ),
    .vsync_out ( y_vs ),
    .hsync_out ( y_hs ),
    .de_out    ( y_de  ),
    .y         ( y_data ),
    .cb        (       ),
    .cr        (       )
);

//矩阵生成
matrix_3x3#(
    .IMG_WIDTH   ( IMG_WIDTH ),
    .IMG_HEIGHT  ( IMG_HEIGHT )
)u_matrix_3x3(
    .video_clk   ( video_clk      ),
    .rst_n       ( rst_n          ),
    .video_vs    ( y_vs           ),
```

```
    .video_de   ( y_de      ),
    .video_data ( y_data    ),
    .matrix_de  ( matrix_de ),
    .matrix11   ( matrix11  ),
    .matrix12   ( matrix12  ),
    .matrix13   ( matrix13  ),
    .matrix21   ( matrix21  ),
    .matrix22   ( matrix22  ),
    .matrix23   ( matrix23  ),
    .matrix31   ( matrix31  ),
    .matrix32   ( matrix32  ),
    .matrix33   ( matrix33  )
);

//sobel 算子
sobel#(
    .SOBEL_THRESHOLD ( 55 )
)u_sobel(
    .video_clk  ( video_clk ),
    .rst_n      ( rst_n     ),
    .matrix_de  ( matrix_de ),
    .matrix_vs  ( y_vs ),
```

```
    .matrix11   ( matrix11   ),
    .matrix12   ( matrix12   ),
    .matrix13   ( matrix13   ),
    .matrix21   ( matrix21   ),
    .matrix22   ( matrix22   ),
    .matrix23   ( matrix23   ),
    .matrix31   ( matrix31   ),
    .matrix32   ( matrix32   ),
    .matrix33   ( matrix33   ),
    .sobel_vs   ( sobel_vs   ),
    .sobel_de   ( sobel_de   ),
    .sobel_data ( sobel_data )
);
```

该部分是例化了灰度化模块、3x3 矩阵生成模块、sobel 处理模块。

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        video_vs_d  <=  1'd0;
    else
        video_vs_d  <=  sobel_vs;
end


assign frame_flag = ~sobel_vs & video_vs_d;    //下降沿
```

```
always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end


always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;    //停止仿真
    end
    else if(sobel_de)    //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",sobel_data,sobel_data,sobel_data);
//16 进制写入
    end
end
```

将对应的部分换成 sobel 处理后的场信号和数据有效信号即可。接下来进行联合仿真后，等待仿真结束，然后将仿真后的 txt 文本数据给到 matlab 去还原成图像。

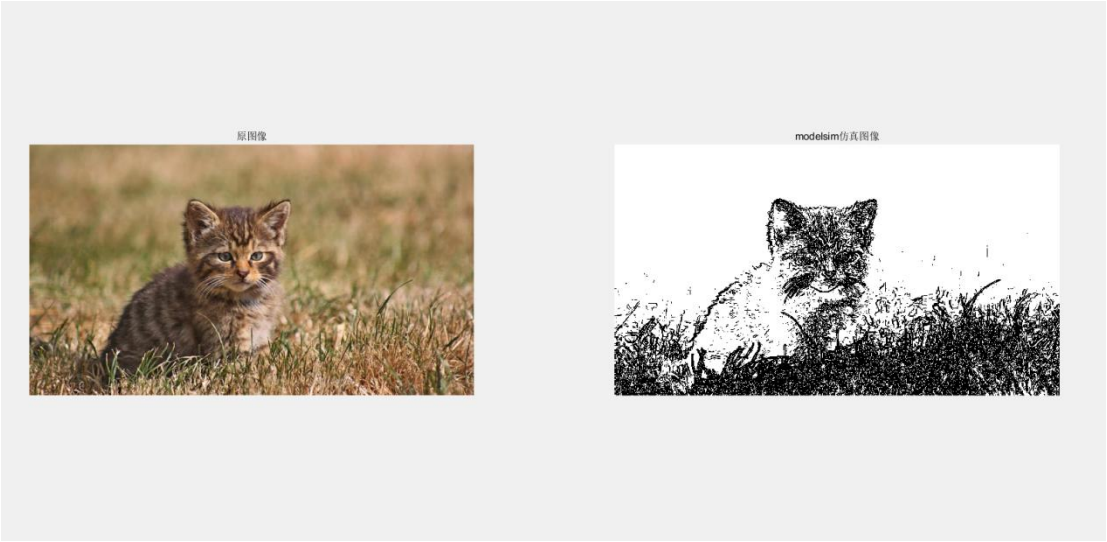


图 8.4-2

8.4.2.实验现象

连接好下载器，电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器，然后下载程序。

图 8.4-3

图 8.4-4

上图为测试原图像。

图 8.4-5

上图为 sobel 算子的边沿检测效果, 读者可以调节阈值来改善效果。

9. gamma 变化

9.1. 实验简介

实验目的:

完成对图像的亮度调节。

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

硬件环境:

MES2L676-100HP-MINI

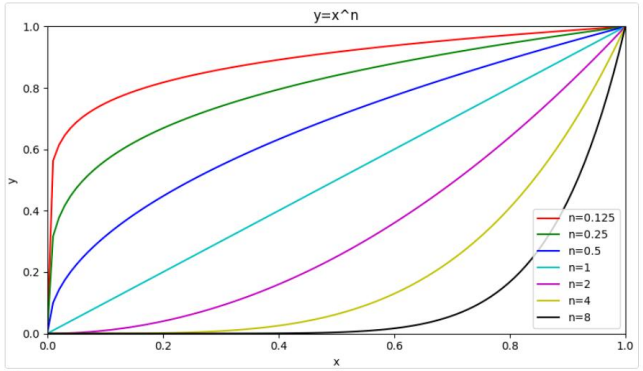
9.2. 实验原理

Gamma 校正是基于人眼对亮度感知非线性的特性, 人眼对亮度的敏感度随着亮度的增加而减少, 也就是说, 当图像的亮度较低时, 人眼对亮度的变化更为敏感。

O = C * I^γ

其中C和γ都是常数，C为亮度和对比度缩放系数，通常取值为1，γ为gamma值，I为归一化的值，范围为【0~1】，O为输出值。

不同γ的曲线如下图:



从图中可以看出:

（1）当γ小于1时，会拉伸图像中灰度级较低的区域，同时会压缩灰度级较高的部分，使图像的暗部更亮，亮部更暗。

（2）当γ大于1时，会拉伸图像中灰度级较高的区域，同时会压缩灰度级较低的部分，使图像的暗部更暗，亮部更亮。

图 9.2-1

具体的公式为:

$$V_{out} = \frac{255}{255^x} * V_{IN}^x$$

x 大于 1 时, 图像变暗。

x 小于 1 时, 图像变亮。

x 等于 1 时, 图像不变。

同样的, 我们会发现一个问题, 该公式涉及到指数和除法, 用 verilog 来实现的难度是比较大的, 我们可以换个角度思考, 本次的亮度调整的像素是 8bit, 因此是 0-255, 我们可以通过查找表的方法来完成本次算法, x 是变量, 是用来调节亮度的参数, Vin 的输入范围是 0-255, 假设, x 为 2.2, 我们可以通过 Matlab 用 for 循环遍历 0-255, 然后计算出当 x=2.2 的时候, 0-255 中的每个数的输出值是多少, 提前计算出来, 做成查找表, 在 verilog 中, 我们就可以通过 case 语句来完成本次算法, 大大降低了运算的复杂度。具体实现过程可以看后续的 Matlab 仿真部分。

9.3. 接口列表

gamma_lookuip.v 模块接口

端口	I/O	位宽	描述
video_data	input	8	处理前的像素数据
gamma_data	output	8	gamma 变化后的数据

因为做成了查找表, 所以接口非常简单, 仅有输入像素数据和输出像素数据。

9.4. 工程说明

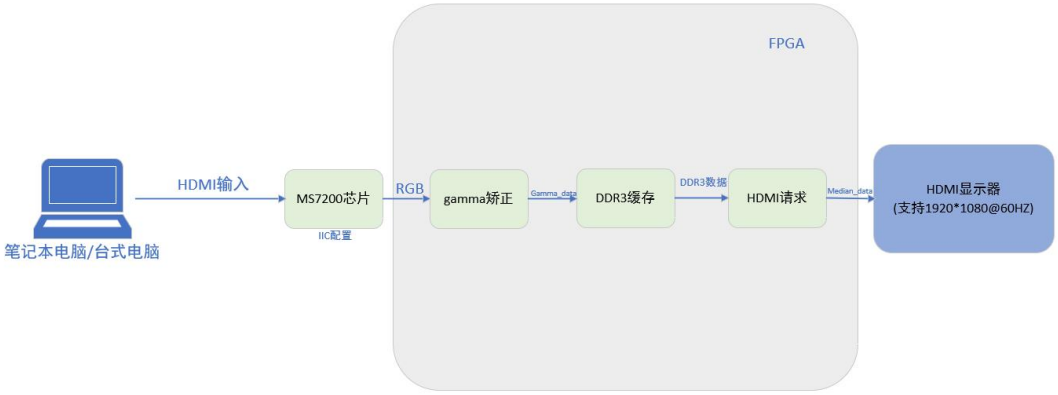


图 9.4-1

本次工程可以直接对输入进来的 RGB888 数据进行 gamma 矫正提高亮度,或者先进行灰度化后, 再进行 gamma 矫正都可以。对于 RGB 的图像, 需要先分出 R、G、B 三个通道, 每个通道均为 8bit, 然后例化三个 gamma 矫正模块即可。

9.4.1.代码模块说明

主要介绍 gamma_lookuptable 模块。

```
//gamma_table   1/2.2
module gamma_lookuptable
(
  input      [7:0]  video_data,
  output  reg [7:0]  gamma_data
);

always@(*)
```

```
begin
  case(video_data)
  8'd0 : gamma_data = 8'd0;
  8'd1 : gamma_data = 8'd21;
  8'd2 : gamma_data = 8'd28;
  8'd3 : gamma_data = 8'd34;
  8'd4 : gamma_data = 8'd39;
  8'd5 : gamma_data = 8'd43;
  8'd6 : gamma_data = 8'd46;
  8'd7 : gamma_data = 8'd50;
  8'd8 : gamma_data = 8'd53;
  8'd9 : gamma_data = 8'd56;
  8'd10 : gamma_data = 8'd59;
  8'd11 : gamma_data = 8'd61;
  8'd12 : gamma_data = 8'd64;
  8'd13 : gamma_data = 8'd66;
  8'd14 : gamma_data = 8'd68;
  8'd15 : gamma_data = 8'd70;
  8'd16 : gamma_data = 8'd72;
  8'd17 : gamma_data = 8'd74;
  8'd18 : gamma_data = 8'd76;
  8'd19 : gamma_data = 8'd78;
```

```
8'd20 : gamma_data = 8'd80;
8'd21 : gamma_data = 8'd82;
8'd22 : gamma_data = 8'd84;
8'd23 : gamma_data = 8'd85;
8'd24 : gamma_data = 8'd87;
8'd25 : gamma_data = 8'd89;
8'd26 : gamma_data = 8'd90;
8'd27 : gamma_data = 8'd92;
8'd28 : gamma_data = 8'd93;
8'd29 : gamma_data = 8'd95;
8'd30 : gamma_data = 8'd96;
8'd31 : gamma_data = 8'd98;
8'd32 : gamma_data = 8'd99;
8'd33 : gamma_data = 8'd101;
8'd34 : gamma_data = 8'd102;
8'd35 : gamma_data = 8'd103;
8'd36 : gamma_data = 8'd105;
8'd37 : gamma_data = 8'd106;
8'd38 : gamma_data = 8'd107;
8'd39 : gamma_data = 8'd109;
8'd40 : gamma_data = 8'd110;
8'd41 : gamma_data = 8'd111;
```

```
8'd42 : gamma_data = 8'd112;
8'd43 : gamma_data = 8'd114;
8'd44 : gamma_data = 8'd115;
8'd45 : gamma_data = 8'd116;
8'd46 : gamma_data = 8'd117;
8'd47 : gamma_data = 8'd118;
8'd48 : gamma_data = 8'd119;
8'd49 : gamma_data = 8'd120;
8'd50 : gamma_data = 8'd122;
8'd51 : gamma_data = 8'd123;
8'd52 : gamma_data = 8'd124;
8'd53 : gamma_data = 8'd125;
8'd54 : gamma_data = 8'd126;
8'd55 : gamma_data = 8'd127;
8'd56 : gamma_data = 8'd128;
8'd57 : gamma_data = 8'd129;
8'd58 : gamma_data = 8'd130;
8'd59 : gamma_data = 8'd131;
8'd60 : gamma_data = 8'd132;
8'd61 : gamma_data = 8'd133;
8'd62 : gamma_data = 8'd134;
8'd63 : gamma_data = 8'd135;
```

```
8'd64 : gamma_data = 8'd136;
8'd65 : gamma_data = 8'd137;
8'd66 : gamma_data = 8'd138;
8'd67 : gamma_data = 8'd139;
8'd68 : gamma_data = 8'd140;
8'd69 : gamma_data = 8'd141;
8'd70 : gamma_data = 8'd142;
8'd71 : gamma_data = 8'd143;
8'd72 : gamma_data = 8'd144;
8'd73 : gamma_data = 8'd144;
8'd74 : gamma_data = 8'd145;
8'd75 : gamma_data = 8'd146;
8'd76 : gamma_data = 8'd147;
8'd77 : gamma_data = 8'd148;
8'd78 : gamma_data = 8'd149;
8'd79 : gamma_data = 8'd150;
8'd80 : gamma_data = 8'd151;
8'd81 : gamma_data = 8'd151;
8'd82 : gamma_data = 8'd152;
8'd83 : gamma_data = 8'd153;
8'd84 : gamma_data = 8'd154;
8'd85 : gamma_data = 8'd155;
```

```
8'd86 : gamma_data = 8'd156;
8'd87 : gamma_data = 8'd156;
8'd88 : gamma_data = 8'd157;
8'd89 : gamma_data = 8'd158;
8'd90 : gamma_data = 8'd159;
8'd91 : gamma_data = 8'd160;
8'd92 : gamma_data = 8'd160;
8'd93 : gamma_data = 8'd161;
8'd94 : gamma_data = 8'd162;
8'd95 : gamma_data = 8'd163;
8'd96 : gamma_data = 8'd164;
8'd97 : gamma_data = 8'd164;
8'd98 : gamma_data = 8'd165;
8'd99 : gamma_data = 8'd166;
8'd100 : gamma_data = 8'd167;
8'd101 : gamma_data = 8'd167;
8'd102 : gamma_data = 8'd168;
8'd103 : gamma_data = 8'd169;
8'd104 : gamma_data = 8'd170;
8'd105 : gamma_data = 8'd170;
8'd106 : gamma_data = 8'd171;
8'd107 : gamma_data = 8'd172;
```

```
8'd108 : gamma_data = 8'd173;
8'd109 : gamma_data = 8'd173;
8'd110 : gamma_data = 8'd174;
8'd111 : gamma_data = 8'd175;
8'd112 : gamma_data = 8'd175;
8'd113 : gamma_data = 8'd176;
8'd114 : gamma_data = 8'd177;
8'd115 : gamma_data = 8'd178;
8'd116 : gamma_data = 8'd178;
8'd117 : gamma_data = 8'd179;
8'd118 : gamma_data = 8'd180;
8'd119 : gamma_data = 8'd180;
8'd120 : gamma_data = 8'd181;
8'd121 : gamma_data = 8'd182;
8'd122 : gamma_data = 8'd182;
8'd123 : gamma_data = 8'd183;
8'd124 : gamma_data = 8'd184;
8'd125 : gamma_data = 8'd184;
8'd126 : gamma_data = 8'd185;
8'd127 : gamma_data = 8'd186;
8'd128 : gamma_data = 8'd186;
8'd129 : gamma_data = 8'd187;
```

```
8'd130 : gamma_data = 8'd188;
8'd131 : gamma_data = 8'd188;
8'd132 : gamma_data = 8'd189;
8'd133 : gamma_data = 8'd190;
8'd134 : gamma_data = 8'd190;
8'd135 : gamma_data = 8'd191;
8'd136 : gamma_data = 8'd192;
8'd137 : gamma_data = 8'd192;
8'd138 : gamma_data = 8'd193;
8'd139 : gamma_data = 8'd194;
8'd140 : gamma_data = 8'd194;
8'd141 : gamma_data = 8'd195;
8'd142 : gamma_data = 8'd195;
8'd143 : gamma_data = 8'd196;
8'd144 : gamma_data = 8'd197;
8'd145 : gamma_data = 8'd197;
8'd146 : gamma_data = 8'd198;
8'd147 : gamma_data = 8'd199;
8'd148 : gamma_data = 8'd199;
8'd149 : gamma_data = 8'd200;
8'd150 : gamma_data = 8'd200;
8'd151 : gamma_data = 8'd201;
```

```
8'd152 : gamma_data = 8'd202;
8'd153 : gamma_data = 8'd202;
8'd154 : gamma_data = 8'd203;
8'd155 : gamma_data = 8'd203;
8'd156 : gamma_data = 8'd204;
8'd157 : gamma_data = 8'd205;
8'd158 : gamma_data = 8'd205;
8'd159 : gamma_data = 8'd206;
8'd160 : gamma_data = 8'd206;
8'd161 : gamma_data = 8'd207;
8'd162 : gamma_data = 8'd207;
8'd163 : gamma_data = 8'd208;
8'd164 : gamma_data = 8'd209;
8'd165 : gamma_data = 8'd209;
8'd166 : gamma_data = 8'd210;
8'd167 : gamma_data = 8'd210;
8'd168 : gamma_data = 8'd211;
8'd169 : gamma_data = 8'd212;
8'd170 : gamma_data = 8'd212;
8'd171 : gamma_data = 8'd213;
8'd172 : gamma_data = 8'd213;
8'd173 : gamma_data = 8'd214;
```

```
8'd174 : gamma_data = 8'd214;
8'd175 : gamma_data = 8'd215;
8'd176 : gamma_data = 8'd215;
8'd177 : gamma_data = 8'd216;
8'd178 : gamma_data = 8'd217;
8'd179 : gamma_data = 8'd217;
8'd180 : gamma_data = 8'd218;
8'd181 : gamma_data = 8'd218;
8'd182 : gamma_data = 8'd219;
8'd183 : gamma_data = 8'd219;
8'd184 : gamma_data = 8'd220;
8'd185 : gamma_data = 8'd220;
8'd186 : gamma_data = 8'd221;
8'd187 : gamma_data = 8'd221;
8'd188 : gamma_data = 8'd222;
8'd189 : gamma_data = 8'd223;
8'd190 : gamma_data = 8'd223;
8'd191 : gamma_data = 8'd224;
8'd192 : gamma_data = 8'd224;
8'd193 : gamma_data = 8'd225;
8'd194 : gamma_data = 8'd225;
8'd195 : gamma_data = 8'd226;
```

```
8'd196 : gamma_data = 8'd226;
8'd197 : gamma_data = 8'd227;
8'd198 : gamma_data = 8'd227;
8'd199 : gamma_data = 8'd228;
8'd200 : gamma_data = 8'd228;
8'd201 : gamma_data = 8'd229;
8'd202 : gamma_data = 8'd229;
8'd203 : gamma_data = 8'd230;
8'd204 : gamma_data = 8'd230;
8'd205 : gamma_data = 8'd231;
8'd206 : gamma_data = 8'd231;
8'd207 : gamma_data = 8'd232;
8'd208 : gamma_data = 8'd232;
8'd209 : gamma_data = 8'd233;
8'd210 : gamma_data = 8'd233;
8'd211 : gamma_data = 8'd234;
8'd212 : gamma_data = 8'd234;
8'd213 : gamma_data = 8'd235;
8'd214 : gamma_data = 8'd235;
8'd215 : gamma_data = 8'd236;
8'd216 : gamma_data = 8'd236;
8'd217 : gamma_data = 8'd237;
```

```
8'd218 : gamma_data = 8'd237;
8'd219 : gamma_data = 8'd238;
8'd220 : gamma_data = 8'd238;
8'd221 : gamma_data = 8'd239;
8'd222 : gamma_data = 8'd239;
8'd223 : gamma_data = 8'd240;
8'd224 : gamma_data = 8'd240;
8'd225 : gamma_data = 8'd241;
8'd226 : gamma_data = 8'd241;
8'd227 : gamma_data = 8'd242;
8'd228 : gamma_data = 8'd242;
8'd229 : gamma_data = 8'd243;
8'd230 : gamma_data = 8'd243;
8'd231 : gamma_data = 8'd244;
8'd232 : gamma_data = 8'd244;
8'd233 : gamma_data = 8'd245;
8'd234 : gamma_data = 8'd245;
8'd235 : gamma_data = 8'd246;
8'd236 : gamma_data = 8'd246;
8'd237 : gamma_data = 8'd247;
8'd238 : gamma_data = 8'd247;
8'd239 : gamma_data = 8'd248;
```

```
    8'd240 : gamma_data = 8'd248;
    8'd241 : gamma_data = 8'd249;
    8'd242 : gamma_data = 8'd249;
    8'd243 : gamma_data = 8'd249;
    8'd244 : gamma_data = 8'd250;
    8'd245 : gamma_data = 8'd250;
    8'd246 : gamma_data = 8'd251;
    8'd247 : gamma_data = 8'd251;
    8'd248 : gamma_data = 8'd252;
    8'd249 : gamma_data = 8'd252;
    8'd250 : gamma_data = 8'd253;
    8'd251 : gamma_data = 8'd253;
    8'd252 : gamma_data = 8'd254;
    8'd253 : gamma_data = 8'd254;
    8'd254 : gamma_data = 8'd255;
    8'd255 : gamma_data = 8'd255;
    endcase
end


endmodule
```

该代码其实均有 matlab 脚本自动生成。代码整体不难，通过查找表的方法找到输入的像素数据对应 gamma 处理后的值，然后输出即可。需要注意的是 always@(*)表示的是组合

逻辑, 所以并不会延迟一个时钟周期, 故不需要对数据有效信号打拍。

9.4.2.代码仿真

9.4.2.1. Matlab 仿真介绍

```
%% 清空
clc;clear all;
%% 读取图像
originalImage = imread('night1.bmp');
%% 读取 RGB 三通道
R = originalImage(:,:,1);
G = originalImage(:,:,2);
B = originalImage(:,:,3);
%% 设定 gamma 值
gamma_value = 1/2.2;
figure;
%% 显示原图
subplot(1, 2, 1);
imshow(originalImage);
title('原图像');
%% 图像尺寸
[img_height, img_width,channel] = size(originalImage);
medianImage = zeros(img_height, img_width);
```

```
gamma_r = zeros(img_height,img_width);
gamma_g = zeros(img_height,img_width);
gamma_b = zeros(img_height,img_width);

for i = 1:img_height
    for j = 1:img_width
        gamma_r(i,j) = (255/255.^(gamma_value))*double(R(i,j)).^(gamma_value);
    end

end

for i = 1:img_height
    for j = 1:img_width
        gamma_g(i,j) = (255/255.^(gamma_value))*double(G(i,j)).^(gamma_value);
    end

end

for i = 1:img_height
    for j = 1:img_width
        gamma_b(i,j) = (255/255.^(gamma_value))*double(B(i,j)).^(gamma_value);
    end
```

```
end

gamma_img = cat(3,uint8(gamma_r),uint8(gamma_g),uint8(gamma_b));

subplot(1, 2, 2);
imshow(gamma_img);
imwrite(gamma_img,'D:\pango_isp\img_test_pg\img_test_pg\01_led_test\sim\matlab_gamma.png')
title('gamma 矫正后的图像图像');
```

重复的操作将不在介绍, 只介绍关键部分。

在代码的 6-8 行, 分离图片的 RGB 三通道。(:,:,1)表示取出红色分量。

代码的 20-22 行定义 gamma 变化后的三个通道的二维矩阵,让其大小和我们读取的图像的大小一致。

代码 24-43 行通过 for 循环分别对 RGB 三通道里的每个像素进行 gamma 变化, 按照公式进行计算。

代码的 45 行将 gamma 变化后的结果强制转为 uint8 格式, 然后利用 cat 函数, 将三通道重新拼成一个三维图像矩阵。

最后通过 imshow(gamma_img)将 gamma 变化后的结果显示出来, imwrite 用来将图片保存到本地。

运行结果如下所示:

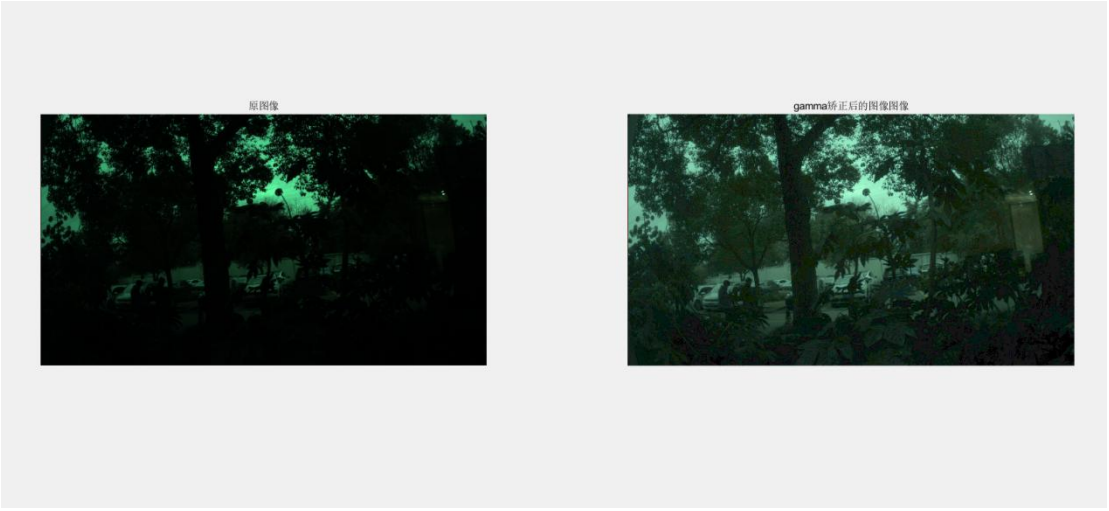


图 9.4-2

9.4.2.2. Matlab 生成查找表

```
%% 清空
clear all; close all; clc;
gamma_value = 1/2.2;%
%% 打开文件
% Open file to write gamma lookup table
fpga_gamma = fopen('gamma_table.v', 'w');
fprintf(fpga_gamma, '//gamma_table\n');
fprintf(fpga_gamma, 'module gamma_lookuptable\n');
fprintf(fpga_gamma, '(\n');
fprintf(fpga_gamma, '   input\t\t[7:0]\tvideo_data,\n');
fprintf(fpga_gamma, '   output\treg\t[7:0]\tgamma_data\n');
fprintf(fpga_gamma, ');\n\n');
```

```
fprintf(fpga_gamma, 'always@(*)\n');
fprintf(fpga_gamma, 'begin\n');
fprintf(fpga_gamma, '\tcase(video_data)\n');

% Initialize gamma array
gamma_array = zeros(1, 256);
for i = 1:256
    gamma_array(1, i) = (255/255.^gamma_value) * (i-1).^gamma_value;
    gamma_array(1, i) = uint8(gamma_array(1, i));
    fprintf(fpga_gamma, '\t8''d%d : gamma_data = 8''d%d; \n',i-1,gamma_array
(1, i));
end

fprintf(fpga_gamma, '\tendcase\n');
fprintf(fpga_gamma, 'end\n');
fprintf(fpga_gamma, '\nendmodule\n');
fclose(fpga_gamma);
```

其实这样的脚本我们可以通过 AI 工具来快速生成, 提高开发效率, 例如使用 GPT、文心一言等, 都可以快速生成类似脚本。

fprintf()函数会在打开的文本里写入数据, '\n'表示换行, 所以按照我们的 verilog 的语法来一步一步将数据写入即可。

代码的 6-15 行就是定义了模块的基本框架比如端口等。

代码的 18-23 行, 通过 for 循环遍历 0-255 共 256 个像素, 把每个像素经过 gamma 变化后的结果写入到文本之中。按照十进制写入, '\t'表示空四个格。

最后在末尾写入 endcase、end 以及 endmodule 即可。

下面便是通过 Matlab 脚本生成的查找表。

```
//gamma_table
module gamma_lookuptable
(
  input      [7:0]   video_data,
  output   reg [7:0]   gamma_data
);

always@(*)
begin
   case(video_data)
   8'd0 : gamma_data = 8'd0;
   8'd1 : gamma_data = 8'd21;
   8'd2 : gamma_data = 8'd28;
   8'd3 : gamma_data = 8'd34;
   8'd4 : gamma_data = 8'd39;
   8'd5 : gamma_data = 8'd43;
   8'd6 : gamma_data = 8'd46;
   8'd7 : gamma_data = 8'd50;
```

```
    8'd8 : gamma_data = 8'd53;
    8'd9 : gamma_data = 8'd56;
    8'd10 : gamma_data = 8'd59;
    8'd11 : gamma_data = 8'd61;
    8'd12 : gamma_data = 8'd64;
    8'd13 : gamma_data = 8'd66;
    8'd14 : gamma_data = 8'd68;
    8'd15 : gamma_data = 8'd70;
    8'd16 : gamma_data = 8'd72;
    8'd17 : gamma_data = 8'd74;
    8'd18 : gamma_data = 8'd76;
    8'd19 : gamma_data = 8'd78;
    ..........................................
    省略中间部分
    ..........................................
    8'd255:gamma_data=8'd255
    endcase
end
endmodule
```

9.4.2.3. Modelsim 仿真介绍

仿真的 TB 只需要例化三个 gamma 模块, 将 RGB 分出三通道, 每个通道都是 8bit。具体如下所示:

```
gamma_lookuptable u_gamma_lookuptable_r(
   .video_data ( video_data[23:16] ),

   .gamma_data  ( gamma_data_r  )
);

gamma_lookuptable u_gamma_lookuptable_g(
   .video_data ( video_data[15:8] ),

   .gamma_data  ( gamma_data_g  )
);

gamma_lookuptable u_gamma_lookuptable_b(
   .video_data ( video_data[7:0] ),

   .gamma_data  ( gamma_data_b  )
);
```

将 RGB888 分别给到 gamma 模块进行处理即可。

```
always@(posedge video_clk or negedge rst_n) begin
```

```
    if(!rst_n)
        video_vs_d  <=   1'd0;
    else
        video_vs_d  <=   video_vs;
end

assign frame_flag = ~video_vs & video_vs_d;    //下降沿

always@(posedge video_clk or negedge rst_n) begin
    if(!rst_n)
        img_done <= 1'b0;
    else if(frame_flag)    //下降沿 判断一帧结束
        img_done <= 1'b1;
    else
        img_done <= img_done;
end


always@(posedge video_clk or negedge rst_n) begin
    if(img_done)
    begin
        $stop;    //停止仿真
```

```
    end
    else if(video_de)   //写入数据
    begin
        $fdisplay(output_file,"%h\t%h\t%h",gamma_data_r,gamma_data_g,gam
ma_data_b);   //16 进制写入
    end
end
```

最后的有效信号、场信号和数据按上述代码修改即可。最后运行联合仿真，将生成的 txt 在 matlab 中恢复为图片显示。

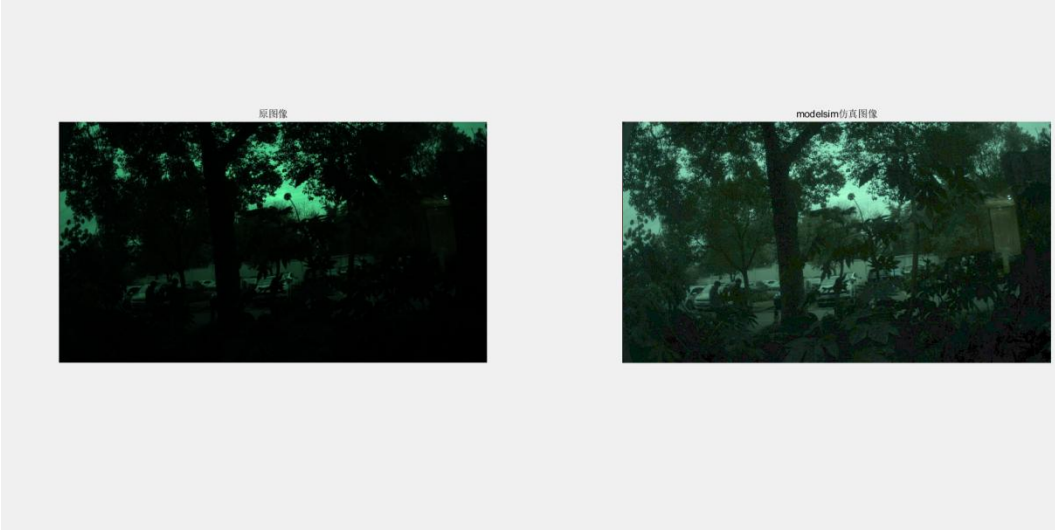


图 9.4-3

9.4.3.实验现象

连接好下载器，电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器，然后下载程序。

图 9.4-4

图 9.4-5

上图为测试原图像。

图 9.4-6

上图为经过 gamma 矫正后的图像, 相对原图, 经过 gamma 矫正后的图像细节更加清晰, 读者可以通过修改 gamma 的值去改善图像增强的效果。

10. 字符叠加

10.1. 实验简介

实验目的:

了解如何取模并完成字符叠加显示

实验环境:

Window11

PDS2022.SP6.4

Modelsim10.6c

MatlabR2023b

PCtoLCD2002

硬件环境:

MES2L676-100HP-MINI

10.2. 实验原理

单个字符的显示可以用 PCtolLCD 软件来提取字模然后显示在屏幕上。字符（包括汉字、字母和符号等）的本质都是点阵字符的大小决定了字符显示区域内像素点的数目，而字符的样式（字体、颜色等）则决定了各像素点的颜色值。因此，在显示字符之前，我们需要先指定字符的大小、样式，然后获取该字符的点阵，这个过程我们称之为“提取字模”，或简称“取模”。我们一般使用 0 和 1 的组合来描述字符的点阵排列：点阵中每个像素点用一位（1 bit）数据来表示，其中用于表征字符的像素点用数字 1 来表示，其他的像素点作为背景用数字 0 来表示。在 FPGA 中，我们只要统计显示的坐标，然后判断字模是不是为 1，是的话就改变颜色，比如背景是白色，那我们可以将该点改成紫色，最后就可以显示出一个紫色的字符。

多个字符显示我们可以先在字符模式下输入 n 个字符,然后保存成 bmp 图片,然后在图形模式下提取字模, 其实就是相当于把多个字符看作一个整体, 当成一个字符来显示。

10.2.1. 字模提取

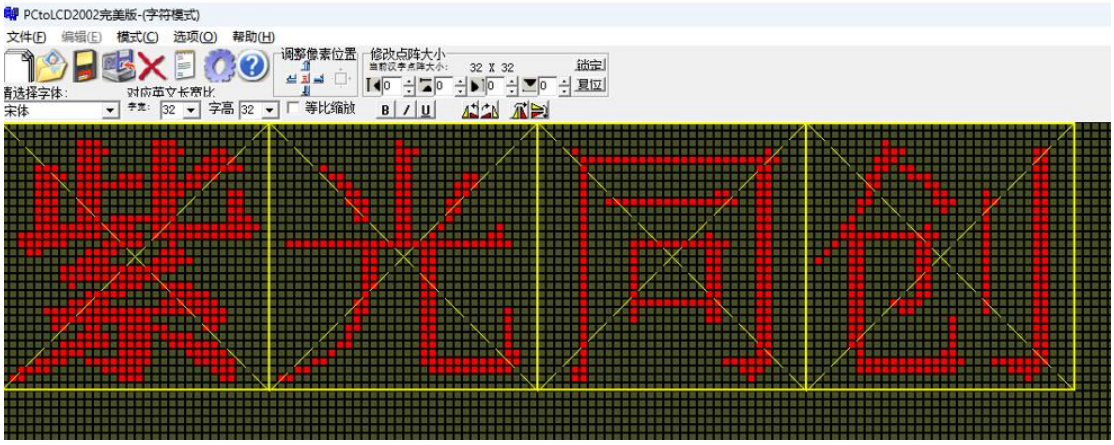


图 10.2-1

字宽和字高均选择 32, 然后输入紫光同创。因为本次要显示多字符, 所以输入完成后, 点击右上角文件, 将其另存为 bmp 图片。

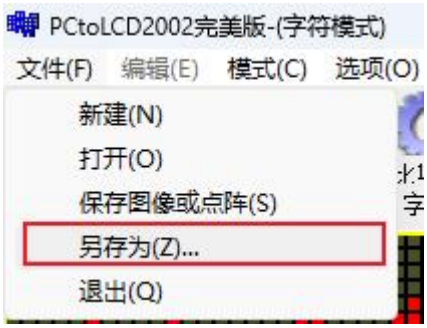


图 10.2-2

之后选取电脑的一个路径, 保存该 bmp 图片即可。然后点击模式, 切换为图形模式, 如下所示:

图 10.2-3

然后点击文件->打开, 选择刚才保存的 BMP 图片。

图 10.2-4

打开后如下所示:

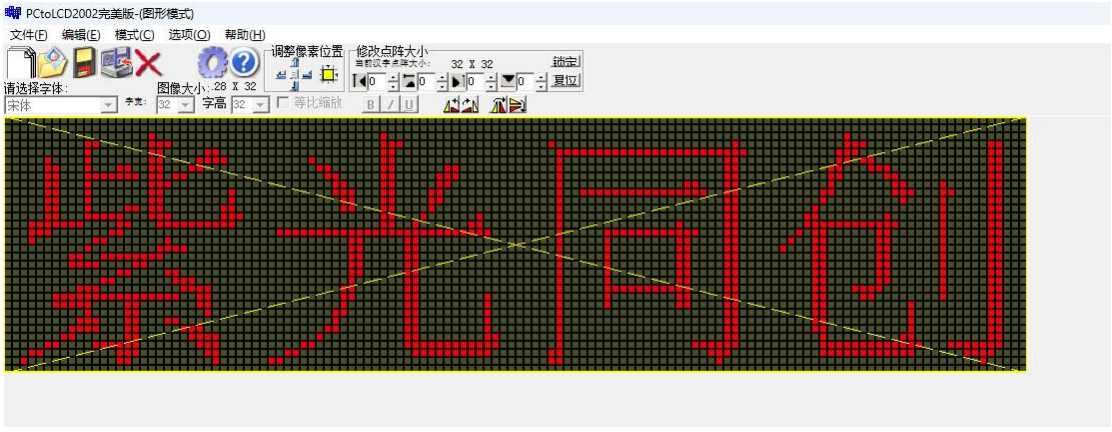


图 10.2-5

之后点击上方的选项, 如下所示:

图 10.2-6

具体设置如下所示：

图 10.2-7

然后点击生成字模，在软件下方会生成对应的字模，将其复制出来。

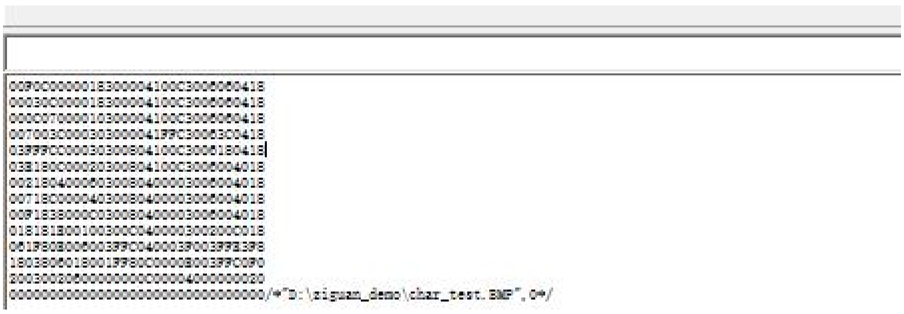


图 10.2-8

-

```
128'h00000000000000000000000000000000
128'h00000000000000000000000000000000
128'h00002000000100000000000000400000
128'h0030380000001C0000800001000700018
128'h00203060000180000 7FFFFF800E00018
128'h042030F00201800004000030000D80018
128'h06233180018181800400003001 8E0018
128'h063FB60000C1838004000030018300018
128'h06203800006183000400063001018418
128'h0620300800718600040FFFF300301C618
128'h0620300800318600040000300200 8418
128'h0621B018003184000400003004000418
128'h063E381C000188100400003000C020418
128'h1FC31FF8000190300400083000FFF0418
128'h1C0600003FFFFF8041FFE3016060418
128'h1018080000183000041000C3026060418
128'h00201E0000183000041000C3046060418
128'h00CFF0000018300004100C3006060418
128'h00F0C0000018300004100C3006060418
128'h00030C0000183000041000C3006060418
128'h000C0700001030000410 0C3006060418
128'h007003C000303000041FFC30063C0418
128'h03FFFCC00030300804100C3006180418
128'h03E180C00020300804100C3006004018
128'h0021804000603008040000300600 4018
128'h00718C0000403008040000300600 4018
128'h00F1838000C0300804000030060 04018
128'h018181E00100300C040000300200C018
128'h061F80E006003FFC040003F003FFE3F8
128'h18038060180001FF80C0000E003FFC0F0
128'h2003002060000000C00004000000020
128'h00000000000000000000000000000000/*"D:\ziguan_demo\char_test.BMP",0*/
```

图 10.2-9

在字模前手动添加 128’h, 即可拿去工程中使用。

10.3. 接口列表

char_proc.v 接口列表

端口	I/O	位宽	描述
pix_clk	input	1	像素时钟
rst_n	input	1	系统复位
i_de	input	1	数据有效信号
x_act	input	12	图像显示的横坐标
y_act	input	12	图像显示的纵坐标

o_de	output	1	叠加字符后的有效信号
o_rgb	output	1	叠加字符 b 标志

10.4. 工程说明

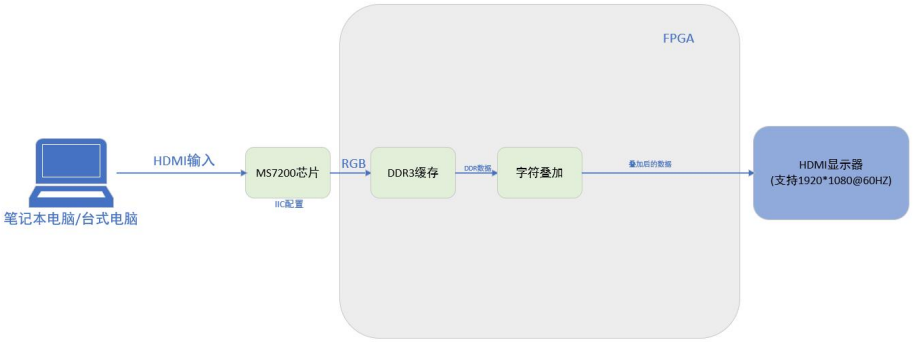


图 10.4-1

该工程结构为 HDMI 接口接收数据，然后传输至 DDR3 进行缓存。随后，系统从 DDR3 中读取这些数据，并在这些数据上叠加必要的字符信息（如文字、符号等）。完成字符叠加处理后，这些数据被送往显示屏幕，最终实现在屏幕上展示包含字符叠加信息的视频内容。

10.4.1. 代码模块说明

```
//字符叠加
module char_proc(
    input           pix_clk,    //像素时钟
    input           rst_n,
    input           i_de ,
    input [11:0]      x_act,    //横
    input [11:0]      y_act,    //纵
```

```
    output reg      o_de   ,
    output reg      o_rgb

);

localparam CHAR_X_START = 12'd1500;//起始的横坐标
localparam CHAR_Y_START = 12'd750;//起始的纵坐标
localparam CHAR_WIDTH = 12'd128;//字符宽度
localparam CHAR_HEIGHT = 12'd32;//字符高度

//字模
wire [127:0] char_1 = 128'h00000000000000000000000000000000 ;
wire [127:0] char_2 = 128'h00000000000000000000000000000000 ;
wire [127:0] char_3 = 128'h00002000000100000000000000400000 ;
wire [127:0] char_4 = 128'h003038000001C0000800001000700018 ;
wire [127:0] char_5 = 128'h00203060000180000 7FFFFF800E00018 ;
wire [127:0] char_6 = 128'h042030F00201800004000030 00D80018 ;
wire [127:0] char_7 = 128'h06233180018181800400003 0018E0018 ;
wire [127:0] char_8 = 128'h063FB60000C183800400003001830018 ;
wire [127:0] char_9 = 128'h06203800006183000400063001018418 ;
```

```
wire [127:0] char_10= 128'h062030080071860004FFFF300301C618 ;
wire [127:0] char_11= 128'h0620300800318600040000300200 8418 ;
wire [127:0] char_12= 128'h0621B018003184000400003004000418 ;
wire [127:0] char_13= 128'h063E381C00018810040000300C020418 ;
wire [127:0] char_14= 128'h1FC31FF800019030040008300FFF0418 ;
wire [127:0] char_15= 128'h1C0600003FFFFF8041FFE3016060418 ;
wire [127:0] char_16= 128'h1018080000183000041 00C3026060418 ;
wire [127:0] char_17= 128'h00201E000018300004100C3046060418 ;
wire [127:0] char_18= 128'h00CFF0000018300004100C3006060418 ;
wire [127:0] char_19= 128'h00F0C0000018300004100C3006060418 ;
wire [127:0] char_20= 128'h00030C0000183000041 00C3006060418 ;
wire [127:0] char_21= 128'h000C07000010300004100C3006060418 ;
wire [127:0] char_22= 128'h007003C000303000041FFC30063C0418 ;
wire [127:0] char_23= 128'h03FFFCC00030300804100C3006180418 ;
wire [127:0] char_24= 128'h03E180C00020300804100C3006004018 ;
wire [127:0] char_25= 128'h0021804000603008040000 3006004018 ;
wire [127:0] char_26= 128'h00718C0000403008040000 3006004018 ;
wire [127:0] char_27= 128'h00F1838000C03008040000 3006004018 ;
wire [127:0] char_28= 128'h018181E00100300C040000300200C018 ;
wire [127:0] char_29= 128'h061F80E006003FFC040003F003FFE3F8 ;
wire [127:0] char_30= 128'h1803806018001FF80C0000E003FFC0F0 ;
wire [127:0] char_31= 128'h20030020600000000C00004000000020 ;
```

```
wire [127:0] char_32= 128'h00000000000000000000000000000000  ;


reg [11:0] x_cnt;
reg char_de;
always@(posedge pix_clk or negedge rst_n)begin
   if(!rst_n) begin
      x_cnt <= 12'd0;
      char_de <= 1'b0;
   end
   else begin
      x_cnt <= x_act-CHAR_X_START;
      char_de <= (x_act >= CHAR_X_START) && (x_act < CHAR_X_START+CHAR_WIDTH)
&&(y_act >= CHAR_Y_START) && (y_act < CHAR_Y_START+CHAR_HEIGHT);
   end
end


always@(posedge pix_clk)
begin
   if(char_de) begin
      case(y_act-CHAR_Y_START)
         15'd0 : begin if(char_1[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
 ;end
```

```
        15'd1 : begin if(char_2[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd2 : begin if(char_3[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd3 : begin if(char_4[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd4 : begin if(char_5[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd5 : begin if(char_6[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd6 : begin if(char_7[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd7 : begin if(char_8[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd8 : begin if(char_9[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<= 0
;end
        15'd9 : begin if(char_10[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0 ;end
        15'd10: begin if(char_11[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd11: begin if(char_12[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
```

```
        15'd12: begin if(char_13[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd13: begin if(char_14[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd14: begin if(char_15[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd15: begin if(char_16[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd16: begin if(char_17[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd17: begin if(char_18[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd18: begin if(char_19[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd19: begin if(char_20[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd20: begin if(char_21[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd21: begin if(char_22[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd22: begin if(char_23[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
```

```
        15'd23: begin if(char_24[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd24: begin if(char_25[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd25: begin if(char_26[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd26: begin if(char_27[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd27: begin if(char_28[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd28: begin if(char_29[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd29: begin if(char_30[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd30: begin if(char_31[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end
        15'd31: begin if(char_32[CHAR_WIDTH-1-x_cnt])  o_rgb<= 1'b1; else o_rgb<=
0; end

      default: o_rgb<= 0;
      endcase
    end
```

```
    else
        o_rgb<= 0;
end

reg    o_de_d;
always@(posedge pix_clk or negedge rst_n)    begin
    if(!rst_n)
    begin
        o_de_d  <=    1'd0;
        o_de    <=    1'd0;
    end
    else
    begin
        o_de_d  <=    i_de;
        o_de    <=    o_de_d;
    end
end
endmodule
```

代码的 15-16 行定义了字符显示的起始位置。

代码的 17-18 行定义了字符的宽度 CHAR_WIDTH, 字符的高度 CHAR_HEIGHT, 需要修改字符大小的话, 除了修改这两个参数外, 取模软件的点阵大小也要进行设置。

代码的 21-52 行就是通过取模软件生成的字模。128*32 的大小。本次显示四个字符” 紫

光同创”, 因此一个字符的宽度是 32, 4 个就是 4*32=128, 高度均为 32.

代码的 56-65 行, x_cnt 是用来计算当前显示的坐标与要显示字符的起始坐标之间的水平距离。char_de 是用来指示字符显示区域 128*32 的区域,当当前的横纵坐标来到要显示的字符区间时, char_de 拉高。

代码的 67-109 行当 char_de 有效时, 判断当前的纵坐标与要显示的字符的起始坐标的

垂直距离, 因为字符的高度是 32, 所以会有 32 种结果, 通过 case 语句判断, 然后要注意的一个点, 其显示的条件是 char_0[CHAR_WIDTH-1-x_cnt], 可以看到, 当显示字符时, 是从 char_0 的高位往低位一个一个取出来, 因为我们的字模”紫光同创”第一个字的字模是存在高位的, 也就是高 32bit 是存放”紫”, 然后存放”光”, 以此类推, 最后的低 32bit 存放”创”。所以要正确显示出”紫光同创”, 应该先判断高位。然后当判断到对应的 bit 为 1, 那就意味着要显示出字符, 所以将 o_rgb 置 1, 当作标志位。其余情况置 0.

代码的 112-116 行, 将数据有效信号打两拍输出, 与 o_rgb 保持同步。

```
parameter  color_d = 24'h9B30FF;//9B30FF   紫色
wire   [7:0]   rgb_data_r;
wire   [7:0]   rgb_data_g;
wire   [7:0]   rgb_data_b;


assign   rgb_data_r = o_rgb?color_d[23:16]:{o_rgb565[15:11],3'd0} ;
assign   rgb_data_g = o_rgb?color_d[15:8]:{o_rgb565[10:5],2'd0} ;
assign   rgb_data_b = o_rgb?color_d[7:0]:{o_rgb565[4:0],3'd0} ;


always  @(posedge pix_clk)begin
```

```
    vs_out     <= vs_reg      ;
    hs_out     <= hs_reg      ;
    de_out     <= o_de        ;
    r_out      <= rgb_data_r   ;
    g_out      <= rgb_data_g   ;
    b_out      <= rgb_data_b   ;

end
```

最后输出图像的时候, 只需要通过组合逻辑判断 o_rgb 的值是否为 1 即可, 如果是 1 就赋值 coloer_d, 即输出紫色, 如果 o_rgb 为 0 则输出原图像信息, 即可完成叠加。

10.4.2. 实验现象

连接好下载器, 电源、HDMI_IN 口连接电脑、HDMD_OUT 口连接显示器, 然后下载程序。

图 10.4-2

图 10.4-3

上图为测试的原图像。

图 10.4-4

可以看到, 图像右下方显示出” 紫光同创” 四个字符。