

40 彩色图像灰度化的设计实现(HDMI 和 TFT 显示)

工程源码	---02_设计实例 ---ch40_acz7015_uart_ddr3_tft_hdmi_rgb2gray
相关视频课程	
	如果您手头的硬件不支持本实验，您可以学习本实验的理论内容，也可以跳过本节内容，继续后续内容的学习。

章节导读

本章节主要是在前述串口传图章节的基础上，加入图像处理模块，实现在 PC 端通过上位机下发尺寸为 400*480 大小的彩色图像数据到 FPGA 的串口，FPGA 通过串口接收的彩色图像数据并进行实时彩色图像灰度化处理，然后将原始彩色图像和处理后的图像拼接在一起并缓存在 DDR3 中，最终在 TFT 屏上同时显示处理前的彩色图像和处理后的灰度图像。

40.1 图像处理基础知识

在上一节，我们讲解了基于 DDR3 的串口传图帧缓存系统设计实现方法，而从本章节开始，我们将延续上一章节内容，正式开启图像处理这一部分的内容学习。

40.1.1 数字图像处理技术的诞生背景

数字图像处理是指将图像信号转换成数字信号并利用计算机对其进行处理的过程。图像处理技术最早出现于 20 世纪 50 年代，当时的电子计算机技术已经发展到一定水平，人们已经开始利用计算机来处理图形和图像信息。数字图像处理作为一门学科大约形成于 20 世纪 60 年代初期。早期的图像处理的目的是改善图像的质量，它以人为对象，以改善人的视觉效果为目的。图像处理中，输入的是质量低的图像，输出的是改善质量后的图像，常用的图像处理方法有图像增强、复原、编码、压缩等。

数字图像处理常用方法有以下几种：图像变换、图像编码压缩、图像增强和复原、图像分割、图像描述、图像分类（识别）。接下来，我们对这几种常用方法的应用场合进行逐一介绍。

40.1.2 图像变换

由于图像分辨率越来越高，导致图像阵列占用的存储空间越来越大。如果直接在空间域中对图像阵列进行处理，涉及计算量很大。因此，往往采用各种图像变换的方法，如傅立叶变换、沃尔什变换、离散余弦变换等间接处理技术，将空间域的处理转换为变换域处理。这样做的好处是不仅可减少计算量，而且可获得更有效的处理效果（如傅立叶变换可在频域中进行数字滤波处理）。目前新兴研究的小波变换在时域和频域中都具有良好的局部优化特性，它在图像处理中也有着广泛而有效的应用。

40.1.3 图像编码压缩

图像编码压缩技术可减少描述图像的数据量（即比特数），以便节省图像传输、处理时间和减少所占用的存储器容量。压缩可以在不失真的前提下获得，也可以在允许的失真条件下进行。编码是压缩技术中最重要的方法，它在图像处理技术中是发展最早且比较成熟的技术。

40.1.4 图像增强和复原

图像增强和复原的目的是提高图像的质量，如去除噪声，提高图像的清晰度等。图像增强不考虑图像降质的原因，只突出图像中所感兴趣的部分。如强化图像高频分量，可使图像中物体轮廓清晰，细节明显；如强化低频分量可减少图像中噪声影响。图像复原要求对图像降质的原因有一定的了解，一般讲应根据降质过程建立“降质模型”，再采用某种滤波方法，恢复或重建原来的图像。

40.1.5 图像分割

图像分割是数字图像处理中的关键技术之一。图像分割的目标，是将图像中有意义的特征部分提取出来。这些有意义的特征包括图像中的边缘、区域等。利用这些特征，可以进一步进行图像识别、分析和理解。虽然目前已研究出不少边缘提取、区域分割的方法，但目前还没有一种普遍适用于分割提取各种图像的有效方法。因此，学界对图像分割的研究还在不断深入之中。同时，图像分割也是目前图像处理中研究的热点之一。

40.1.6 图像描述

图像描述是图像识别和理解的必要前提。作为最简单的二值图像可采用其

几何特性描述物体的特性，一般图像的描述方法采用二维形状描述，它有边界描述和区域描述两类方法。对于特殊的纹理图像可采用二维纹理特征描述。随着图像处理研究的深入发展，学界已经开始进行三维物体描述的研究，提出了体积描述、表面描述、广义圆柱体描述等方法。

40.1.7 图像分类（识别）

图像分类与识别属于模式识别的范畴，其主要内容是图像经过某些预处理（增强、复原、压缩）后，进行图像分割和特征提取，从而进行判决分类。图像分类常采用经典的模式识别方法，有统计模式分类和句法（结构）模式分类，近年来新发展起来的模糊模式识别和人工神经网络模式分类在图像识别中也越来越受到重视。

随着计算机技术的发展，图像处理技术已经深入到我们生活中的方方面面，其中，在娱乐休闲上的应用已经深入人心。图像处理技术在娱乐中的应用主要包括：电影特效制作、电脑电子游戏、数码相机、视频播放、数字电视等。

电影特效制作：自从 20 世纪 60 年代以来，随着电影中逐渐运用了计算机技术，一个全新的电影世界展现在人们面前，这也是一次电影的革命。越来越多的计算机制作的图像被运用到了电影作品的制作中。其视觉效果的魅力有时已经大大超过了电影故事的本身。如今，我们已经很难发现一部没有利用任何计算机数码元素的新电影。

电脑电子游戏：电脑电子游戏的画面，是近年来电子游戏发展最快的部分之一。从 1996 年到现在，游戏画面的进步简直可以用突飞猛进来形容，随着图像处理技术的发展，众多在几年前无法想像的画面在今天已经成为了平平常常的东西。

数码相机：所谓数码相机，是一种能够进行拍摄，并通过内部处理把拍摄到的景物转换成以数字格式存放图像的特殊照相机。与普通相机不同，数码相机并不使用胶片，而是使用固定的或者是可拆卸的半导体存储器来保存获取的图像。数码相机可以直接连接到计算机、电视机或者打印机上。在一定条件下，数码相机还可以直接连接到移动式电话机或者手持 PC 机上。由于图像是内部处理的，所以使用者可以马上检查图像是否正确，而且可以立刻打印出来或是通过电子邮件传送出去。

视频播放与数字电视：家庭影院中的 VCD，DVD 播放器和数字电视中，大量使用了视频编码解码等图像处理技术，而视频编码解码等图像处理技术的

发展，也推动了视频播放与数字电视向高清晰，高画质方向发展。

后续章节，我们将依次介绍基于 FPGA 实现数字图像处理的相关内容。在硬件上，后续图像处理章节将基于前述串口传图到 DDR3 然后显示在 TFT/HDMI 显示器实验的程序框架，对需要送往显示设备显示的图像数据进行常见数字图像处理的相关运算。这些图像处理算法包括：RGB 彩色图像转灰度图像、图像中值滤波算法、图像均值滤波算法、图像高斯滤波算法以及基于 Sobel 算子的边缘检测算法等。

学习数字图像处理相关章节的内容，核心在于对目标任务的实现算法的理解，在对算法的理解基础上，利用 Verilog HDL 语言实现这些算法。

数字图像处理算法理论的建立和推导是一个严谨且科学的数学过程，当前广泛采用的数字图像处理算法，其理论基础已经由图像处理相关领域的专家进行了深度的分析和论证。本章内容，重点在于介绍各种数字图像处理算法在 FPGA 平台上的具体实现方法，而对于各种数字图像处理算法的理论推导和验证，本章内容不会过多涉及。各位读者有兴趣的，可自行根据专业的数字图像处理教材进行学习和验证。

那么接下来，我们就正式开启图像处理章节相关课程。首先，我们先来分析彩色图像灰度化的设计实现。

40.2 灰度图像的相关概念

在正式入题之前先给大家讲解一下灰度（gray）图像，YUV 图像以及 Ycbcr 图像。

Gray 图像：灰度（gray）图像就是我们常说的黑白图像，由黑到白为灰阶，其值域为 0-255(8bit)。

YUV 图像：YUV 是被欧洲电视系统采用的一种颜色编码方法（属于 PAL），是 PAL 和 SECAM 模拟彩色电视制式采用的颜色空间。在现代彩色电视系统中，通常采用三管彩色摄影机或彩色 CCD 摄影机进行取像，然后把取得的彩色图像信号经分色、分别放大校正后得到 RGB 图像，RGB 图像经过矩阵变换电路得到亮度信号 Y 和两个色差信号 $B-Y$ （即 U）、 $R-Y$ （即 V），最后发送端将亮度和色差三个信号分别进行编码，用同一信道发送出去。这种色彩的表示方法就是所谓的 YUV 色彩空间表示。YUV 色彩空间的特点是它的亮度信号 Y 和色度信号 U、V 是分离的。YUV 主要用于优化彩色视频信号的传输，使其向后兼容老式黑白电视。与 RGB 视频信号传输相比，它最大的优点在于只需占用极少的

频宽（RGB 要求三个独立的视频信号同时传输）就能完成图像传输。

对于 YUV 三种图像传输信号分量，每一种分量其实也都是有其物理意义的。其中“Y”表示明亮度（Luminance 或 Luma），也就是灰阶值；而“U”和“V”表示的则是色度（Chrominance 或 Chroma），作用是描述影像色彩及饱和度。如果 U 和 V 明确，那么像素的颜色就是可以指定和还原的。

由于计算机的普及，一种适合于在计算机显示系统中得到应用的 YUV 制式，即 YCbCr 得到更加广泛的运用。

如果将 RGB 信号的特定部分叠加到一起，则可以明确“亮度”值。“色度”U 和 V 则定义了颜色的两个方面——色调与饱和度，分别用 Cr 和 Cb 来表示。其中，Cr 反映了 RGB 输入信号红色部分与 RGB 信号亮度值之间的差异。而 Cb 反映的是 RGB 输入信号蓝色部分与 RGB 信号亮度值之间的差异。

在计算机数字信号的绝对统治力下，YCbCr 也逐渐成为了 YUV 制式的典型代表。但是各位读者需明白，YCbCr 和 YUV 是两个不同的概念，YCbCr 制式只是 YUV 的其中一种最适宜用于计算机显示的数据信号格式。

使用灰度图像可以实现：在保留图像显示的内容信息的同时，弱化图像的颜色信息。虽然人眼观察经过灰度化处理后的图像信息，体验感没有彩色图像逼真，但相较于彩色图像而言，灰白图像的后续分析会大大减轻硬件负担。这样，不太关心图像体验感的用户可以更高效地利用存储资源和总线资源，尽快获得图像中期望的内容。

用一句俗话来说：颜色信息，是非常“吃硬件”的，于是有人就想，能不能剥离颜色信息，而进行图像数据的关键信息判断呢？打个不恰当的比方如下：如果你是汽车驾驶员，对于驾驶汽车这个任务，只关心前方是否有人，有坑，有障碍物，前车距离是多远，而并不关心道路是什么颜色，道路两旁的房屋漂不漂亮。驾驶员的大脑，并不需要对过多无关的颜色信息进行分析（当然，红绿灯还是要看一看的），而只需要简单地判断前方是否安全即可。

40.3 系统整体设计

介绍了灰度图像的实现意义以后，我们来看 RGB 彩色图像灰度化实现的整体设计方案。

最终 TFT 显示的要求如下。

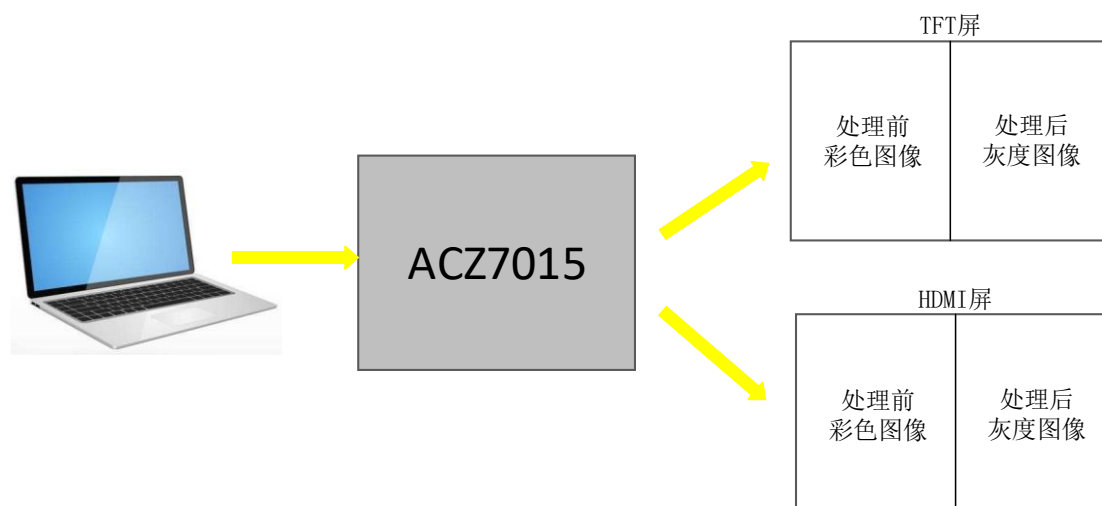


图 40-1 彩色图像灰度化的实验目标

系统整体设计框图如下。从 FPGA 设计架构来说，彩色图像灰度化与“基于 DDR3 的串口传图帧缓存系统设计实现”架构基本一致，增加了彩色图像灰度化处理模块 `rgb2gray` 模块和图像合并模块 `image_stitch_x` 模块。

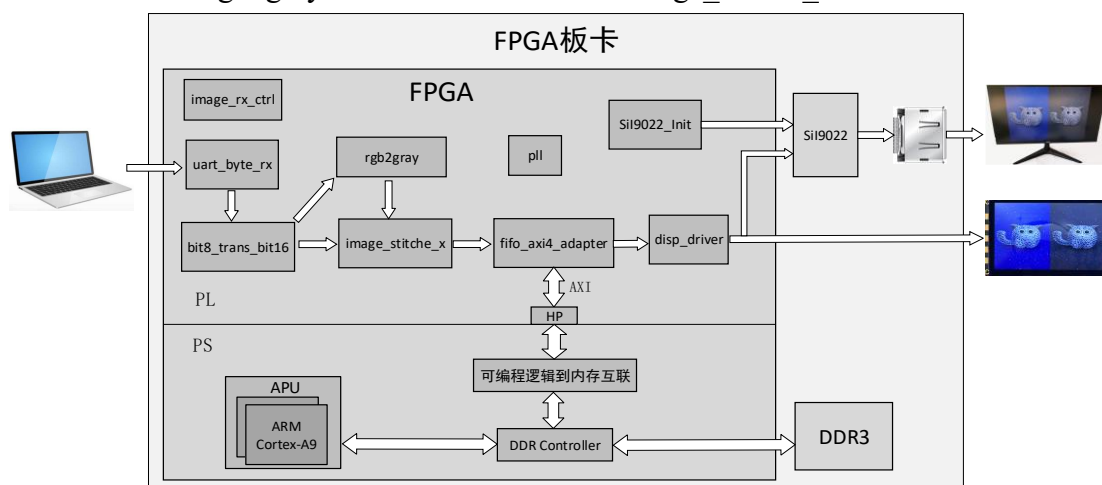


图 40-2 系统设计框图

其中，

- (1) `uart_byte_rx` 模块：负责串口图像数据的接收，该模块的设计前面章节已经有讲。
- (2) `bit8_trans_bit16` 模块：将串口接收的每两个 8bit 数据转换成一个 16bit 数据（图像数据是 16bit 的 RGB565 的数据，电脑是通过串口将一个像素点数据分两次发送到 FPGA，FPGA 需将串口接收数据重组为 16bit 的图像数据），实现过程相对比较简单。
- (3) `rgb2gray` 模块：彩色图像灰度化处理，对串口接收的彩色图像数据实

时进行灰度化处理。

- (4) `image_stitch_x` 模块：将串口接收的尺寸为 400*480 大小的彩色图像与灰度化处理后的 400*480 大小的图像数据以左右形式合并成一张 800*480 的图像。
- (5) `disp_driver` 模块：tft 屏显示驱动控制，对缓存在 DDR3 中的图像数据进行显示。
- (6) `fifo_axi4_adapter` 模块组：包含 `wr_ddr3_fifo`、`rd_ddr3_fifo`、`fifo2axi4` 以模块，完成采集的图像数据缓存。
- (7) `pll` 模块：生成上述各个模块所需时钟。
- (8) `image_rx_ctrl` 模块。用于显示接收进度，给出接收完成标志。
- (9) `SiI9022_Init` 模块：SiI9022 芯片初始化模块。
- (10) `disp_driver` 模块：显示驱动模块，用于产生显示所需的时序信号。

本系统就是在上一系统基础上添加图像处理模块搭建系统。除去使用 IP 和前面章节讲过的模块外，还需要设计的模块包括 `rgb2gray` 模块和 `image_stitch_x` 模块。

40.4 彩色图像灰度化处理模块设计

40.4.1 基本原理

将彩色图像转化为灰度图像的过程称为图像灰度化处理。常见的 24 位深度彩色图像 RGB888 中的每个像素的颜色由 R、G、B 三个分量决定，并且三个分量各占 1 个字节，每个分量可以取值 0~255，这样一个像素点可以有 1600 多万（ $255*255*255$ ）的颜色的变化范围。而灰度图像是 R、G、B 三个分量相同的一种特殊的彩色图像，其一个像素点的变化范围为 0~255。对于一幅彩色图来说，其对应的灰度图则是只有 8 位的图像深度，这也说明了用灰度图做图像处理所需的计算量确实要少。不过需要注意的是，虽然丢失了一些颜色等级，但是从整幅图像的整体和局部的色彩以及亮度等级分布特征来看，灰度图描述与彩色图的描述是一致的。一般有分量法、最大值法、平均值法、加权平均法四种方法对彩色图像进行灰度化。

40.4.2 彩色图像灰度化处理方法介绍

40.4.2.1 方法 1：分量法

将彩色图像中的三分量的亮度作为三个灰度图像的灰度值，可根据应用需要选取一种灰度图像。具体表达式如下。

$$\text{gray}_1(i, j) = R(i, j)$$

$$\text{gray}_2(i, j) = G(i, j)$$

$$\text{gray}_3(i, j) = B(i, j)$$

其中， $\text{gray}_1(i, j), \text{gray}_2(i, j), \text{gray}_3(i, j)$ 为转换后的灰度图像在 (i, j) 处的灰度值， $R(i, j), G(i, j), B(i, j)$ 分别为转换前的彩色图像在 (i, j) 处 R、G、B 三个分量的值。

40.4.2.2 方法 2：最大值法

将彩色图像中的三分量亮度 R，G，B 的最大值作为灰度图的灰度值。具体表达式如下。

$$\text{gray}(i, j) = \max[R(i, j), G(i, j), B(i, j)]$$

40.4.2.3 方法 3：平均值法

将彩色图像中的三分量亮度求平均得到一个灰度值。如下：

$$\text{gray}(i, j) = \frac{R(i, j) + G(i, j) + B(i, j)}{3}$$

上式中有除法，考虑到在 FPGA 中实现除法比较的消耗资源，这里在实现前可以先做如下的近似处理。可以将上面公式乘以 $3/256$ ，这样就需要同时乘以 $256/3$ 保证公式的正确性。公式处理过程如下：

$$\begin{aligned}\text{gray}(i, j) &= \frac{[R(i, j) + G(i, j) + B(i, j)]}{3} * \frac{3}{256} * \frac{256}{3} \\ &= \frac{[R(i, j) + G(i, j) + B(i, j)]}{256} * \frac{256}{3}\end{aligned}$$

对 $256/3$ 做近似取整处理，将 $256/3$ 替换成 85，则公式变为如下。

$$\text{gray}(i, j) \approx \frac{[R(i, j) + G(i, j) + B(i, j)]}{256} * 85$$

这样式子中除以 256 就可以采用移位方法来处理，式子变为如下：

$$\text{gray}(i, j) = \{[R(i, j) + G(i, j) + B(i, j)] * 85\} \gg 8$$

上面处理过程中使用是对 256/3 的近似处理，当然这里可以采用其他数据，比如 512/3、1024/3、2048/3 等等，基本的原则是将平均公式法中分母的 3 替换成 2 的幂次的数，这样除法就可以使用移位的方式实现，减小 FPGA 中由于存在除法带来的资源消耗。

40.4.2.4 方法 4：加权平均法

根据重要性及其他指标，将三个分量以不同的权值进行加权平均。有一个很著名的心理学公式：

$$\text{gray}(i,j) = 0.299 * R(i,j) + 0.587 * G(i,j) + 0.114 * B(i,j)$$

这里 $0.299+0.587+0.114=1$ ，刚好是满偏，这是通过不同的敏感度以及经验总结出来的公式，一般可以直接用这个。在实际应用时，为了避免低速的浮点运算以及除法运算，可以先将式子缩放 1024 倍来实现运算算法，如下：

$$\text{gray}(i,j) = [0.299 * R(i,j) + 0.587 * G(i,j) + 0.114 * B(i,j)] * 1024 / 1024$$

通过近似取整处理后得到近似公式如下。

$$\text{gray}(i,j) \approx [306 * R(i,j) + 601 * G(i,j) + 117 * B(i,j)] / 1024$$

式子中除以 1024（这里是 2 的 n 次方就可以，n 不同，结果会略微有差别）可以采用移位方法来处理，式子变为如下：

$$\text{gray}(i,j) \approx [306 * R(i,j) + 601 * G(i,j) + 117 * B(i,j)] \gg 10$$

也可以压缩到 8 位以内，式子变为如下。具体压缩到多少位可以根据实际需求。

$$\text{gray}(i,j) = [77 * R(i,j) + 150 * G(i,j) + 29 * B(i,j)] \gg 8$$

40.4.3 彩色图像灰度化处理模块设计验证

上述中方法 1、2 实现起来相对容易，这里就不多说。接下来主要是对方法 3、4 在 FPGA 上的实现做讲解。

40.4.3.1 方法 3 平均值法实现

该方法实现起来并不复杂，通过上面的计算公式可以知道，计算公式里只有加法、乘法和移位计算，这里的乘法通过移位相加的方式进行计算，计算具体实现见下面代码。

```
//求平均法 GRAY = (R+B+G)/3= ((R+B+G)*85) >>8
wire [9:0]sum;
reg [15:0]gray_r;
```

```
assign sum = red_8b_i + green_8b_i + blue_8b_i;

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        gray_r <= 16'd0;
    else if(rgb_valid)
        gray_r <= (sum << 6)+(sum << 4)+(sum << 2)+ sum;
    else
        gray_r <= 16'd0;
end

assign gray_8b_o = gray_r[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid;
    gray_hs    <= rgb_hs;
    gray_vs    <= rgb_vs;
end
```

对该模块的仿真也相对比较简单，只需要在 testbench 中给时钟复位激励以及 RGB 三通道的图像数据即可，具体代码如下，仿真中分别给 R、G、B 通道不同的是起始数据值，然后通过递增加 1 的形式改变 R、G、B 通道数据，验证设计的正确性。

```
initial begin
    reset_p    = 1;
    rgb_valid  = 0;
    red_8b_i   = 0;
    green_8b_i = 0;
    blue_8b_i  = 0;
    #(`CLK_PERIOD*200+1);
    reset_p    = 0;
    red_8b_i   = 56;
    green_8b_i = 124;
    blue_8b_i  = 203;
    #2000;

    rgb_valid = 1;
    repeat(256)begin
        #(`CLK_PERIOD)
        red_8b_i   = red_8b_i   + 1;
        green_8b_i = green_8b_i + 1;
        blue_8b_i  = blue_8b_i  + 1;
    end
end
```

```
rgb_valid = 0;  
  
#2000;  
$stop;  
end
```

为了验证采用平均值法实现彩色图像灰度化计算的正确性，在仿真代码中加入了如下代码。直接通过平均法的原始公式产生一组对比数据。

```
always@(posedge clk)  
begin  
    if(rgb_valid == 1'b1)  
        comp1_gray <= (red_8b_i + green_8b_i + blue_8b_i)/3;  
    else  
        comp1_gray <= 0;  
end
```

这样可以比较容易地通过对比 comp1_gray 和 gray_8b_o 的值来验证设计模块的正确性。其仿真波形图如下：

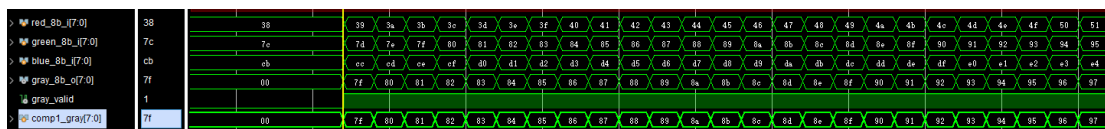


图 40-3 平均值法仿真数据对比



图 40-4 平均值法仿真数据对比（续）

从图 40-4 可以看到，仿真波形中，从 aa 开始 comp1_gray 和 gray_8b_o 的值相差 1。这里我们可以分析下出现这个问题的原因，其实 gray_8b_o 和 comp1_gray 的计算公式并非完全一样，gray_8b_o 的计算公式是为了避免除法做了一定的近似，而在仿真文件中 comp1_gray 是直接求的平均。可以通过 red、green、blue 这 3 个数据分别对 gray_8b_o 和 comp1_gray 进行计算，计算结果与仿真波形结果是一致的。这也说明了，避免除法做近似处理计算的 gray 和直接通过除法求的平均值 comp1_gray 是稍存在偏差的。这个是可以接受的误差范围。

40.4.3.2 方法 4 加权平均法实现

对于加权平均法可通过两种方式实现，公式直接计算法和查找表法。

40.4.3.2.1 公式直接计算法

公式直接计算法与方法 3 实现类似，通过转换公式直接进行计算，只是具

体计算数值发生了变化，同样乘法采用移位相加的方式实现。具体代码如下：

```
//典型灰度转换公式 Gray = R*0.299+G*0.587+B*0.114=(R*77 + G*150 + B*29) >>8
wire [15:0]red_x77;
wire [15:0]green_x150;
wire [15:0]blue_x29;
reg [15:0]sum;

//乘法转换成移位相加方式
assign red_x77 = (red_8b_i << 6) + (red_8b_i << 3) + (red_8b_i << 2) + red_8b_i;
assign green_x150 = (green_8b_i<< 7) + (green_8b_i<< 4) + (green_8b_i<< 2) + (green_8b_i<<1);
assign blue_x29 = (blue_8b_i << 4) + (blue_8b_i << 3) + (blue_8b_i << 2) + blue_8b_i;

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        sum <= 16'd0;
    else if(rgb_valid)
        sum <= red_x77 + green_x150 + blue_x29;
    else
        sum <= 16'd0;
end

assign gray_8b_o = sum[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid;
    gray_hs <= rgb_hs;
    gray_vs <= rgb_vs;
end
```

40.4.3.2.2 公查找表法

通过观察计算公式发现，R、G、B 数据值均乘以一个定值，然后对乘法之后的结果相加，最后右移 8 位，上面采用直接算法实现是对常数乘法采用的移位相加方法计算，对于这类固定范围内的数值，同时可取数据不多情况下（这里 R、G、B 数值范围在 0~255，可取的数据有限）乘以一个常数，可以采用查找表方法实现，该方法主要的优势是直接通过访问 ROM 内的数据，相对使用移位相加实现乘法使用的 LUT 资源会少点，但占用的存储器会更多。因为需要将 R、G、B 乘以系数之后的数值存储在 ROM 中，然后通过读取 ROM 方式

来得到计算之后的数值。这里使用 Vivado 添加 3 个 ROM IP 核，分别通过 R*75、G*147、B*36 ($0 \leq R \leq 255$, $0 \leq G \leq 255$, $0 \leq B \leq 255$) 的计算值建立 3 个初始化 coe 文件，然后在 ROM IP 核中分别添加 coe 文件进行初始化。具体代码如下：代码中 rom_red_x77、rom_green_x150、rom_blue_x29 分别存储着 R*75、G*147、B*36 ($0 \leq R \leq 255$, $0 \leq G \leq 255$, $0 \leq B \leq 255$) 256 个数值。在提供的工程目录下的\uart_ddr3_tft_rgb2gray.srcs\sources_1\ip\coe 文件夹中有提供这 3 个 coe 文件。使用查找表法实现彩色图像灰度化的代码如下。

```
//查找表方式，可以省去公式法中乘法运算 Gray =(R*77 + G*150 + B*29) >>8，将
3 个分量乘以系数后的数值存储在 ROM 中
wire [14:0]red_x77;
wire [15:0]green_x150;
wire [13:0]blue_x29;
reg [15:0]sum;
reg      rgb_valid_dly1;
reg      rgb_hs_dly1;
reg      rgb_vs_dly1;

rom_red_x77 rom_red_x77(
    .clka (clk      ), // input wire clka
    .ena  (rgb_valid ), // input wire ena
    .addra (red_8b_i ), // input wire [7 : 0] addra
    .douta (red_x77   ) // output wire [14 : 0] douta
);

rom_green_x150 rom_green_x150(
    .clka (clk      ), // input wire clka
    .ena  (rgb_valid ), // input wire ena
    .addra (green_8b_i ), // input wire [7 : 0] addra
    .douta (green_x150 ) // output wire [15 : 0] douta
);

rom_blue_x29 rom_blue_x29(
    .clka (clk      ), // input wire clka
    .ena  (rgb_valid ), // input wire ena
    .addra (blue_8b_i ), // input wire [7 : 0] addra
    .douta (blue_x29   ) // output wire [13 : 0] douta
);

always@(posedge clk)
begin
    rgb_valid_dly1 <= rgb_valid;
    rgb_hs_dly1    <= rgb_hs;
    rgb_vs_dly1    <= rgb_vs;
end
```



```

always@(posedge clk or posedge reset_p)
begin
    if(reset_p)
        sum <= 16'd0;
    else if(rgb_valid_dly1)
        sum <= red_x77 + green_x150 + blue_x29;
    else
        sum <= 16'd0;
end

assign gray_8b_o = sum[15:8];

always@(posedge clk)
begin
    gray_valid <= rgb_valid_dly1;
    gray_hs    <= rgb_hs_dly1;
    gray_vs    <= rgb_vs_dly1;
end

```

仿真过程与方法 3 类似，这里就不重复描述。对于方法 4 的两种不同实现方式，可以对比下综合后消耗的资源情况。图 40-5 和图 40-6 分别为使用公式直接计算法和查找表法实现使用资源情况。

Name	Slice LUTs (20800)	Bonded IOB (250)	BUFGCTRL (32)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)
rgb2gray	83	36	1	9	28	83

图 40-5 公式直接计算法使用资源情况

Name	Slice LUTs (20800)	Block RAM Tile (50)	Bonded IOB (250)	BUFGCTRL (32)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)
rgb2gray	21	1.5	36	1	10	7	21
> PROC_LUT_rom_blue_x29 (rom_blue_x29)	0	0.5	0	0		0	0
> PROC_LUT_rom_green_x150 (rom_green_x150)	0	0.5	0	0		0	0
> PROC_LUT_rom_red_x77 (rom_red_x77)	0	0.5	0	0		0	0

图 40-6 查找表法使用资源情况

比较两种方法使用资源情况，可以很明显地看出查找表方式消耗的 LUT 相对较少，但消耗了 1.5 个 Block RAM 资源。

对于同一功能多种不同实现方法的模块代码如何整合到一起呢？当然每种方法作为一个单独的模块使用一个.v 文件保存肯定是没有问题的，这个就不太便于后期的维护和使用。如果能将多种实现方法整合到一个模块保存在一个.v 文件，使用起来就更加的方便。方法肯定是有的，而且还不止一种。下面提供两种方式，宏定义法和使用 generate -if 方法。宏定义法相对比较好理解，通过

不同的宏定义条件编译方式进行选择某种实现方式，实现的部分代码如下。该方法可通过修改模块代码的宏定义选择不同的方法，还算是比较方便的。

```
//`define AVERAGE    //求平均法
//`define FORMULA     //直接公式法
`define LUT           //查找表法

module rgb2gray (
    clk,
    rst_n,
    rgb_valid,
    red_8b_i,
    green_8b_i,
    blue_8b_i,
    gray_8b_o,
    gray_valid
);
`ifndef AVERAGE //求平均法 GRAY=(R+B+G)/3= ((R+B+G)*85) >>8
...//方法 1
`endif

`ifndef FORMULA //灰度转换公式 Gray = R*0.299+G*0.587+B*0.114
...//方法 2
`endif

`ifndef LUT//查找表方式
...//方法 3
`endif

endmodule
```

使用 generate-if 方法也是比较好的办法。先看该种方法的代码如下。

```
module rgb2gray
#(
    parameter PROC_METHOD = "AVERAGE" // "AVERAGE" :求平均法
                                         //or "FORMULA" :直接公式法
                                         //or "LUT" :查找表法
)
(
    clk,
    rst_n,
    rgb_valid,
    red_8b_i,
    green_8b_i,
    blue_8b_i,
    gray_8b_o,
    gray_valid
);
```

```
generate
  if (PROC_METHOD == "AVERAGE") begin: PROC_AVERAGE
    //-----
    //求平均法 GRAY = (R+B+G)/3= ((R+B+G)*85) >>8
    //方法 1
    //-----
  end
  else if (PROC_METHOD == "FORMULA") begin: PROC_FORMULA
    //-----
    //典型灰度转换公式 Gray = R*0.299+G*0.587+B*0.114=(R*77 + G*150 +
    B*29) >>8
    //方法 2
    //-----
  end
  else if (PROC_METHOD == "LUT") begin: PROC_LUT
    //-----
    //查找表方式，可以省去公式法中乘法运算 Gray =(R*77 + G*150 + B*29) >>8，将
    3 个分量乘以系数后的数值存储在 ROM 中
    //方法 3
    //-----
  end
endgenerate

endmodule
```

该种方式相对于宏定义条件编译来说更加方便，就是在模块被例化调用时，不管需要使用哪种方式都不需要去通过修改模块源代码方式去改变具体实现方法，直接在例化模块时就可通过重定义参数 `PROC_METHOD` 实现不同方法的设置。具体代码如下。

```
rgb2gray
#(
  .PROC_METHOD("FORMULA") // "AVERAGE" : 求平均法
                           // or "FORMULA" : 直接公式法
                           // or "LUT"    : 查找表法
)rgb2gray_average
(
  .clk      (clk      ),
  .reset_p  (reset_p  ),
  .rgb_valid(rgb_valid),
  .red_8b_i (red_8b_i ),
  .green_8b_i(green_8b_i),
  .blue_8b_i (blue_8b_i ),
  .gray_8b_o (gray_8b_o ),
  .gray_valid(gray_valid)
```

```
);
```

而且这种方式可以实现多次例化时还可以让每个被例化的模块采用指定的实现方法，具体可参看提供例程的工程目录下\uart_ddr3_tft_hdmi_rgb2gray.srcs\sim_1\new中的仿真文件rgb2gray_tb中的应用。

40.5 图像合并模块设计

要实现图像以左右形式合并，首先要清楚“基于 DDR3 的串口传图帧缓存系统”是如何实现在显示屏上显示一张图片的。串口接收到的数据会经过数据拼接后写入 DDR3 的指定地址区域，在需要的时候，再从这个指定地址区域中依次读出数据，用于显示屏的显示。

存储数据的内存范围由待显示的图像数据决定，例如 5 寸屏的分辨率为 800*480，显示一帧图像需要 384000 个 16 位的图像数据。这些数据在 DDR3 中需要占用 $384000 * 16 / 8 = 768000$ 的地址，如果图像数据写入的起始地址为 0，那么图像数据写入的结束地址便是 $0 + 768000 - 1 = 767999$ 。我们只需要按照顺序从地址 0 开始读取这些数据一直到地址 767999，就能获得一副完整的图像，实现一帧图像的显示。要想图像持续显示，就需要不断地读取地址 0~767999 的数据。

而如果我们向 DDR3 中写入的是两张 400*480 的图像数据，那么显示的就是一张混合后拼接成的 800*480 的图像。但是如何写入才能达到左右拼接的效果呢？先写入第一张 400*480 的图像数据，然后再写入第二张 400*480 的图像数据？并不是。要知道如何去写入这两张图片数据，需要知道 TFT 显示图片的扫描方式，这个在前面 TFT 显示驱动章节已经讲过，扫描方式是一行一行的，也就是这个缓存图片数据的“RAM”存储图片数据的内容结构如下图所示，地址 0~1599 存储的是第 1 行图像数据，地址 1600~3199 存储的是第 2 行图像数据……，地址 766400~767999 存储的是第 480 行图像数据。

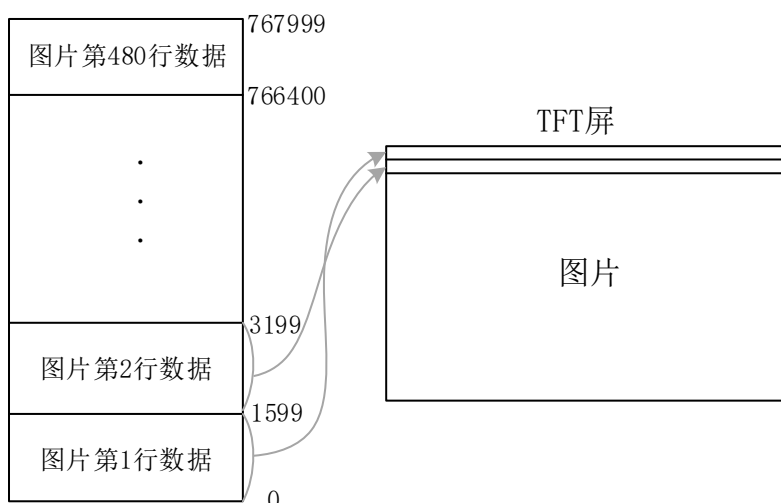


图 40-7 图片数据存储和显示对应关系 1

如果要想实现左右形式的图片拼接显示，就需要往 DDR3 中交错写的写入存储两张 400*480 的图片数据，左右形式拼接的 DDR3 中存储的数据与 TFT 屏显示的对对应关系如下图所示。从图中可以看出，存储两张 400*480 图片的方式是，先存储左边图像（图片 1）的一行数据，然后再存储右边图像（图片 2）的一行数据，直到两张图像的 480 行数据均存储完成。

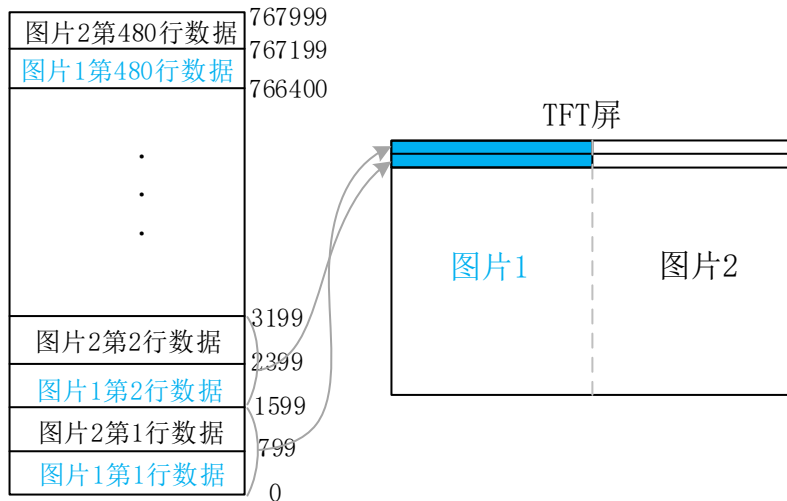


图 40-8 图片数据存储和显示对应关系 2

想要实现上面存储方式，考虑到两张图片可能是同时从其他模块进入，但由于图片 1 和图片 2 的存储到 DDR3 有先后顺序，避免在存储其中一张图片时，另一张图片数据的丢失，需要在存储到 DDR3 前为图片 1 和图片 2 的数据各加一个 fifo，进行少量数据的缓存。图像合并模块的整体设计框架如下图所示。图像合并模块的两个图片输入端口，分别作为图片 1 和图片 2 数据流的输入。通

过逻辑控制交替输出两张图片的行数据给到下一级存储模块，到达实现图片合并的显示效果。

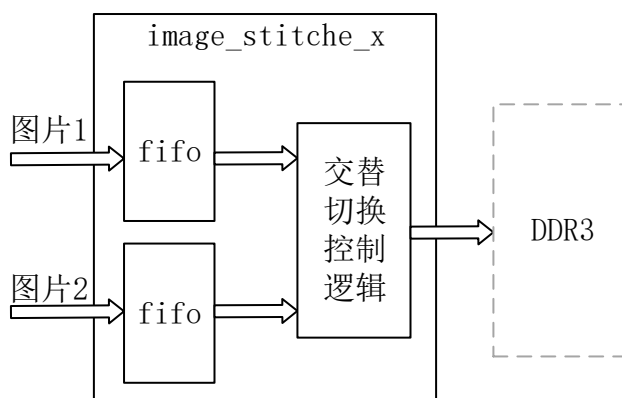


图 40-9 图片合并模块的用途示意

所谓的交替，就是先读取图片 1 的 FIFO 的 400 个像素数据进行输出，读取完之后，切换到读取图片 2 的 FIFO 的 400 个像素数据进行输出，同样读取完之后，切换到读取图片 1 的 FIFO 的 400 个像素数据进行输出，如此反复进行这样的切换操作。（这里需要注意的是每次读取的数据个数是图片 1 或图片 2 的一行数据的像素点个数，这里图片 1/图片 2 均为 400*480 大小图片，所以每次读取 400 个像素数据）

根据上面的设计思路，采用状态机进行设计，画出状态转移图如下图所示。

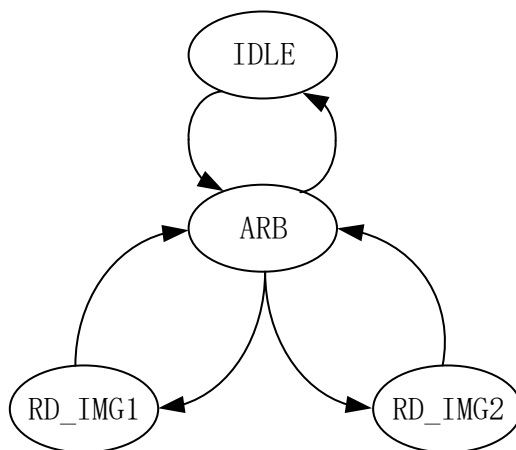


图 40-10 图像合并模块的状态转移图

- IDLE：初始状态，上电或复位后处于该状态；
- ARB：仲裁状态，根据当前的情况决定下一状态跳转到哪个状态。如果跳转到 ARB 的上一状态是读取图片 1 的 FIFO 数据状态 RD_IMG1，则下一状态跳转到读取图片 2 的 FIFO 数据状态 RD_IMG2；否则如果跳转到 ARB 的上一状态是读取图片 2 的 FIFO 数据状态 RD_IMG2，则下

一状态跳转到读取图片 1 的 FIFO 数据状态 RD_IMG1，读取 FIFO 的交替切换就是在本状态实现。

- RD_IMG1：读取图片 1 的 FIFO 数据状态，在一行图像数据（400 个数）读取完成后，状态跳转到 ARB 状态。
- RD_IMG2：读取图片 2 的 FIFO 数据状态，在一行图像数据（400 个数）读取完成后，状态跳转到 ARB 状态。

状态机具体实现代码如下。

```
//*****  
//State Machine  
//*****  
always@(posedge clk_image_out or posedge reset_p)  
begin  
    if(reset_p)  
        curr_state <= S_IDLE;  
    else  
        curr_state <= next_state;  
end  
  
always@(*)  
begin  
    case(curr_state)  
        S_IDLE:  
            begin  
                if(rst_busy_o == 1'b0)  
                    next_state = S_ARB;  
                else  
                    next_state = S_IDLE;  
            end  
  
        S_ARB:  
            begin  
                if((rd_image_sel == 1'b0) && (image1_buf_empty == 1'b0))  
                    next_state = S_RD_IMG1;  
                else if((rd_image_sel == 1'b1) && (image2_buf_empty == 1'b0))  
                    next_state = S_RD_IMG2;  
                else  
                    next_state = S_ARB;  
            end  
  
        S_RD_IMG1:  
            begin  
                if(image1_buf_rden && (rd_data_cnt == IMAGE1_WIDTH_IN - 1'b1))  
                    next_state = S_ARB;
```

```
        else
            next_state = S_RD_IMG1;
        end

S_RD_IMG2:
begin
    if(image2_buf_rden && (rd_data_cnt == IMAGE2_WIDTH_IN - 1'b1))
        next_state = S_ARB;
    else
        next_state = S_RD_IMG2;
    end

default: next_state = S_IDLE;
endcase
end
```

IDLE 状态中判断条件信号 `rst_busy_o` 为复位情况下, FIFO 的复位忙信号, FIFO 的复位需要一段时间, 需要在 FIFO 复位完全结束, 也就是 FIFO 的复位忙信号均变为 0 后才跳转到 ARB 状态。

RD_IMG1 和 RD_IMG2 状态中的 `rd_image_sel` 为读 FIFO 的切换选择标识信号, 每读完图片 1 (或图片 2) FIFO 的一行图片数据 (400 个), 信号 `rd_image_sel` 电平就变化一次。具体代码如下。

```
always@(posedge clk_image_out or posedge reset_p)
begin
    if(reset_p)
        rd_image_sel <= 1'b0;
    else if((curr_state == S_RD_IMG1) && (rd_data_cnt ==
IMAGE1_WIDTH_IN - 1'b1))
        rd_image_sel <= 1'b1;
    else if((curr_state == S_RD_IMG2) && (rd_data_cnt ==
IMAGE2_WIDTH_IN - 1'b1))
        rd_image_sel <= 1'b0;
    else
        rd_image_sel <= rd_image_sel;
end
```

不同的 `rd_image_sel` 信号电平表示的是需要读取哪个 FIFO 的图片数据, `rd_image_sel` 为 0 表示需要读取图片 1 的 FIFO 的数据, `rd_image_sel` 为 1 表示需要读取图片 2 的 FIFO 的数据。

代码中 `rd_data_cnt` 表示读取 FIFO 的数据个数计数, 在读取 FIFO 状态 (RD_IMG1 或 RD_IMG2 状态), 每读取一个 FIFO 数据, 计数器加 1, 具体代码如下。

```
always@(posedge clk_image_out or posedge reset_p)
```

```
begin
    if(reset_p)
        rd_data_cnt <= 'd0;
    else if(curr_state == S_RD_IMG1)begin
        if(image1_buf_rden == 1'b1)
            rd_data_cnt <= rd_data_cnt + 1'b1;
        else
            rd_data_cnt <= rd_data_cnt;
        end
    else if(curr_state == S_RD_IMG2)begin
        if(image2_buf_rden == 1'b1)
            rd_data_cnt <= rd_data_cnt + 1'b1;
        else
            rd_data_cnt <= rd_data_cnt;
        end
    else
        rd_data_cnt <= 'd0;
    end
end
```

读 FIFO 信号 image1_buf_rden 和 image2_buf_rden 分别根据当前所处状态和 FIFO 的空满标识控制产生，直接使用组合逻辑产生。具体代码如下。

```
assign image1_buf_rden = (curr_state == S_RD_IMG1) &
(image1_buf_empty == 1'b0) & (data_out_ready_i == 1'b0);
assign image2_buf_rden = (curr_state == S_RD_IMG2) &
(image2_buf_empty == 1'b0) & (data_out_ready_i == 1'b0);
```

data_out_ready_i 信号表示的是下游其他模块的入口 FIFO 的将满信号，主要是考虑到下游模块的入口 FIFO 如果要满的情况下，是不去读本模块的 FIFO 数据的，因为读取的本模块的 FIFO 数据是要输出给下游的，如果下游 FIFO 满了，还往下游传数据会导致数据的丢失的。

合并后的输出信号就相对简单，将读出的 FIFO 的数据直接输出即可，考虑到读 FIFO 的读使能与读出的数据有效之间有一个周期的延时，所以读出的数据需要在读使能延迟一拍后获取数据，具体代码如下：

```
always@(posedge clk_image_out or posedge reset_p)
begin
    if(reset_p)begin
        image1_buf_rden_dly1 <= 1'b0;
        image2_buf_rden_dly1 <= 1'b0;
    end
    else begin
        image1_buf_rden_dly1 <= image1_buf_rden;
        image2_buf_rden_dly1 <= image2_buf_rden;
    end
end
end
```

```
always@(posedge clk_image_out or posedge reset_p)
begin
    if(reset_p)
        data_valid_o <= 1'b0;
    else if(image1_buf_rden_dly1 | image2_buf_rden_dly1)
        data_valid_o <= 1'b1;
    else
        data_valid_o <= 1'b0;
end

always@(posedge clk_image_out or posedge reset_p)
begin
    if(reset_p)
        data_pixel_o <= 'd0;
    else if(image1_buf_rden_dly1)
        data_pixel_o <= image1_buf_dout;
    else if(image2_buf_rden_dly1)
        data_pixel_o <= image2_buf_dout;
    else
        data_pixel_o <= 'd0;
end
```

考虑到模块的可移植性和可扩展性将图片的尺寸或数据位宽使用参数化表示，输入的两张图片的大小和合并输出的图片的大小尺寸均用参数化表示，具体代码如下。

```
module image_stitch_x
#(
    parameter DATA_WIDTH      = 16,    //16 or 24
    //image1_in: 400*480
    parameter IMAGE1_WIDTH_IN  = 400,
    parameter IMAGE1_HEIGHT_IN = 480,
    //image2_in: 400*480
    parameter IMAGE2_WIDTH_IN  = 400,
    parameter IMAGE2_HEIGHT_IN = 480,
    //image_out: 800*480
    parameter IMAGE_WIDTH_OUT   = 800,
    parameter IMAGE_HEIGHT_OUT  = 480
)
(
    input          clk_image1_in      ,
    input          clk_image2_in      ,
    input          clk_image_out      ,
    input          reset_p             ,

    output         rst_busy_o          ,
    output         image_in_ready_o    ,

```

```
input      [DATA_WIDTH-1:0] image1_data_pixel_i,
input      image1_data_valid_i,

input      [DATA_WIDTH-1:0] image2_data_pixel_i,
input      image2_data_valid_i,

input      data_out_ready_i    ,
output reg [DATA_WIDTH-1:0] data_pixel_o    ,
output reg data_valid_o
);
```

考虑到用户在例化使用模块时对参数进行重定义时可能会出现由于笔误或其他不小心的错误使得图片的尺寸设置出现，输入的两张图片大小的尺寸参数的宽度之和不等于输出图片大小尺寸，或者输入图片的高度与输出图片的高度不一致问题，导致最终无法达到模块的功能而不便于定位错误。本模块与“彩色图像灰度化处理模块”设计采用类似的方式，使用 `generate-if` 的形式对错误参数的设置输出固定的数据，或者采取某种固定的输出，这样便于根据实际仿真和上板的现象快速定位是否为参数配置错误导致。具体代码如下。

```
generate
  if ((IMAGE1_WIDTH_IN + IMAGE2_WIDTH_IN != IMAGE_WIDTH_OUT) ||
      (IMAGE1_HEIGHT_IN != IMAGE_HEIGHT_OUT) || (IMAGE2_HEIGHT_IN !=
      IMAGE_HEIGHT_OUT))
    begin: error_set_pro    //错误设置输入/输出图片参数情况的处理
    //-----
    always@(posedge clk_image_out or posedge reset_p)
    begin
      if(reset_p)
        data_pixel_o <= 'd0;
      else
        data_pixel_o <= {DATA_WIDTH{1'b1}};
    end

    always@(posedge clk_image_out or posedge reset_p)
    begin
      if(reset_p)
        data_valid_o <= 'd0;
      else
        data_valid_o <= 1'b1;
    end
    //-----
  end
else
  begin: correct_set_pro    //正确设置输入/输出图片参数情况的处理
  //-----
  .....//正确参数配置情况下的代码，也就文档前面分析设计的代码
```

```
//-----  
end  
endgenerate
```

至此，图像左右合并模块的设计基本完成，完整代码和仿真的 tb 文件参见提供的工程文件。仿真波形如下，图片 1 通道写入了两行（每行 50 个）数据，数据分别为 0~49 和 100~149。



图 40-11 通道 1 写入第一行数据



图 40-12 通道 1 写入第二行数据



图 40-13 通道 1 第二行数据写完

之后，图片 2 通道写入了两行（每行 50 个）数据，数据分别为 50~99 和 150~199；

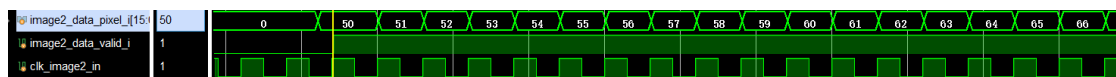


图 40-14 通道 2 写入第一行数据

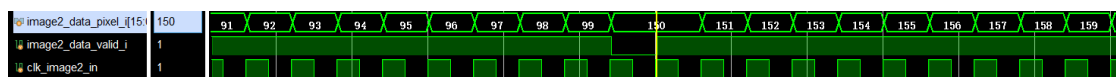


图 40-15 通道 2 写入第二行数据

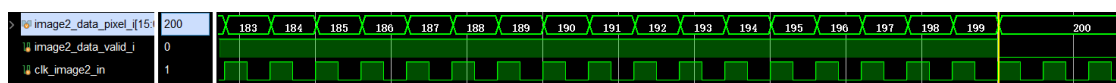


图 40-16 通道 2 第二行数据写完

理论上，合并之后的输出数据应该为 0~199 的 200 个数据，可以观察仿真波形，仿真结果与期望一致。



图 40-17 输出数据 1



图 40-18 输出数据 2

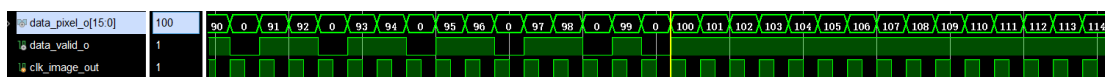


图 40-19 输出数据 3



图 40-20 输出数据 4



图 40-21 输出数据 5

40.6 顶层模块设计

各个子模块设计完成后，顶层的设计就相对容易些，根据整体设计框图对子模块端口信号进行连接即可。由于串口发送的图片数据以及在 TFT 屏上显示的图片数据是 16bit 的 RGB565 数据，而彩色图像灰度化处理是对 24bit 的 RGB888 数据进行转换，这里就涉及 RGB565→RGB888 的转换和 RGB888→RGB565 的转换。以下转换关系是参考于网上。

24bit RGB888 -> 16bit RGB565 的转换
24bit RGB888 R7 R6 R5 R4 R3 R2 R1 R0 G7 G6 G5 G4 G3 G2 G1 G0 B7 B6 B5 B4 B3 B2 B1 B0
16bit RGB565 R7 R6 R5 R4 R3 G7 G6 G5 G4 G3 G2 B7 B6 B5 B4 B3
从 8bit 到 5bit 或 6bit，取原 8bit 的高位，数据位上做了压缩，却损失了精度。
16bit RGB565 -> 24bit RGB888 的转换
16bit RGB565 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B4 B3 B2 B1 B0
24bit RGB888 R4 R3 R2 R1 R0 R2 R1 R0 G5 G4 G3 G2 G1 G0 G1 G0 B4 B3 B2 B1 B0 B2 B1 B0
从 5bit 或 6bit 到 8bit，取原 5bit 或 6bit 的低 3 位或低 2 位做补全成 8 位。

两种转换关系在顶层中均有用到，具体代码参见提供的工程源码文件。

顶层的仿真与“基于 DDR3 的串口传图帧缓存系统”类似，这里就不做详细讲解，读者自己建立仿真文件完成顶层的仿真。为了方便用户测试，在例程的顶层中我们还通过条件编译为大家设置了测试数据。用户可以通过是否定义 USE_TEST_DATA 信号来选择是否使用测试数据。通过设置 DISP_MODE 信号的值，用户可以选择不同的测试数据：

- 模式 0：由蓝到黑的纵向条纹渐变
- 模式 1：由蓝到黑的横向条纹渐变
- 模式 2：彩条显示

● 模式 3：以彩条显示为背景的小梅哥 logo 动态显示

测试数据具体实现原理这里就不为大家讲解了，大家可以打开例程参考。

在顶层设计分析综合没有错误并且顶层仿真确认设计功能没有问题后，为设计分配管脚。本次设计的管脚约束表如表 40-1：

表 40-1 引脚分配表

Pin Name	Signal Name	Pin NO.	Pin Name	Signal Name	Pin NO.
FPGA_UART_RX	uart_rx	W18	TFT_rgb[12]	Display_R1	K3
FPGA_KEY0	reset_n	R4	TFT_rgb[11]	Display_R0	K4
AUD_I2C_SCL	SiI9022_sclk	J3	TFT_rgb[10]	Display_G5	H8
AUD_I2C_SDA	SiI9022_sdat	K2	TFT_rgb[9]	Display_G4	L4
FPGA_LED0	led	P7	TFT_rgb[8]	Display_G3	R8
TFT_clk	Display_PCLK	M1	TFT_rgb[7]	Display_G2	J6
TFT_de	Display_DE	P6	TFT_rgb[6]	Display_G1	J7
TFT_pwm	Display_BL	P5	TFT_rgb[5]	Display_G0	M3
TFT_hs	Display_HSYNC	N1	TFT_rgb[4]	Display_B4	M2
TFT_vs	Display_VSYNC	P1	TFT_rgb[3]	Display_B3	L1
TFT_rgb[15]	Display_R4	M4	TFT_rgb[2]	Display_B2	L2
TFT_rgb[14]	Display_R3	K5	TFT_rgb[1]	Display_B1	J1
TFT_rgb[13]	Display_R2	J5	TFT_rgb[0]	Display_B0	J2

分配完管脚后为引脚设置电平约束，随后生成 Bit 文件，将硬件文件导出到 SDK，连接硬件准备板级验证。

40.7 板级验证

本节将进行基于 ACZ7015 开发板的彩色图像灰度化设计的板级验证，设计需要使用到 PL 端的串口，因此需要通过 40pin 拓展接口外接 EDA 拓展板。

40.7.1 系统所需硬件

本次设计所需硬件如下，相关模块资料可以点击超链接查看：

1. [ACZ7015 开发板](#) x1
2. [EDA 扩展板](#) x1
3. dc 电源线 x1
4. Type-c 线 x2
5. HDMI 线缆 x1
6. 支持 HDMI 接口的液晶显示器 x1

40.7.2板级验证需求

彩色图像灰度化的板级验证，主要验证以下三个方面：

1. 能够正确地全屏点亮 TFT 屏和 HDMI 显示器，显示稳定。
2. 能否正确地显示两个画面，即左侧为正常彩色画面，右侧为灰度图像。
3. 能否正确地定位坐标，即实现在指定的位置显示对应的数据。

40.7.3硬件连接

本次设计硬件连接如图 40-22 和图 40-23 所示：

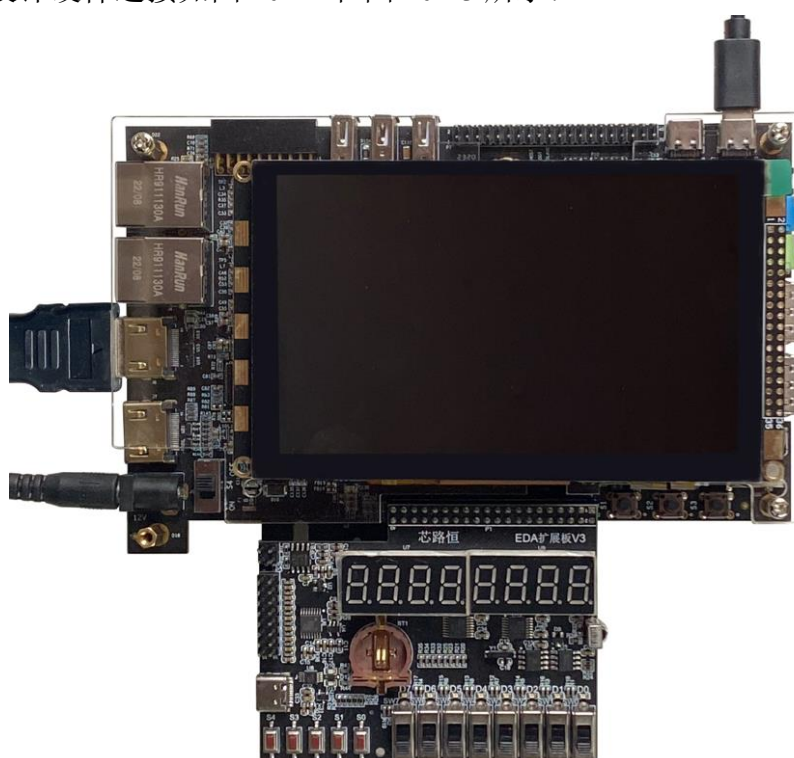


图 40-22 硬件连接

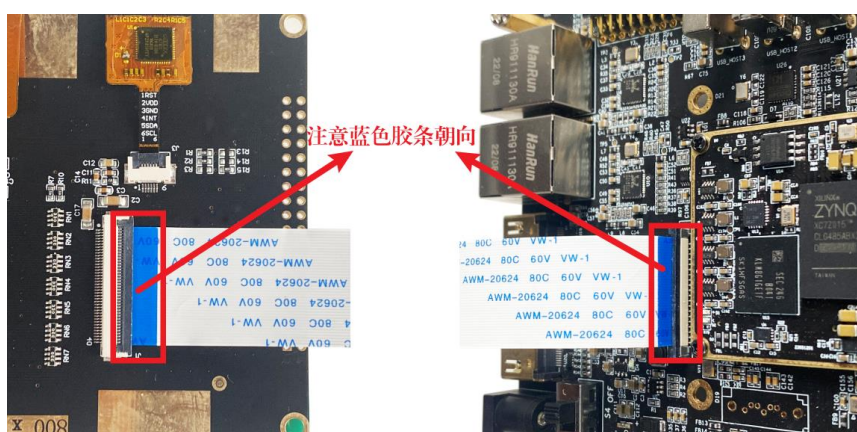


图 40-23 硬件连接

使用 HDMI 线缆连接开发板与 HDMI 显示器，注意线缆一端连接开发板 HDMI_1 接口，一端连接显示器 HDMI 接口；使用软排线连接开发板与 TFT 屏，连接时注意软排线朝向；将 EDA 拓展板插接在开发板 40pin 接口上，正确连接时，接口应该一一接合；两个 type-c 线一根连接开发板 PS 侧调试接口和主机，一根连接 EDA 拓展板上的串口。

在连接完硬件后将电源拨码开关拨动到对应启动 ON 侧，接下来便可以准备烧录了。

40.7.4显示效果

这里我们首先测试串口传图效果，在 SDK 中创建配置任务，将设计烧录到开发板中。随后，在设备管理器中查询串口端口号，打开小梅哥串口传图工具，设置图像分辨率为 400*480，波特率为 1562500，连接串口。如图 40-24 所示：

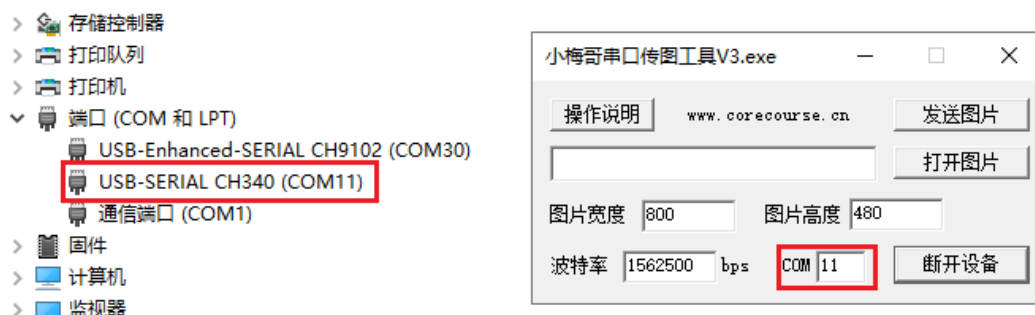


图 40-24 连接小梅哥串口传图工具

这里我们选用的是图 40-25 所示图片，该图片为我们提前为用户准备好的素材。



图 40-25 准备下载的图片

关于图片的信息可通过右键图片查看其属性，在属性的详细信息窗口可以看到图片大小，位深度等信息。**注：**这里提供的上位机要求图片为位深度为 24 的 bmp 格式图片，图片的宽度和高度需要为 400*480（插 5 寸屏情况下）。



图 40-26 准备下载的图片像素信息

在将图片传输到开发板后，TFT 屏以及 HDMI 显示器上的显示效果如图 40-27 所示：

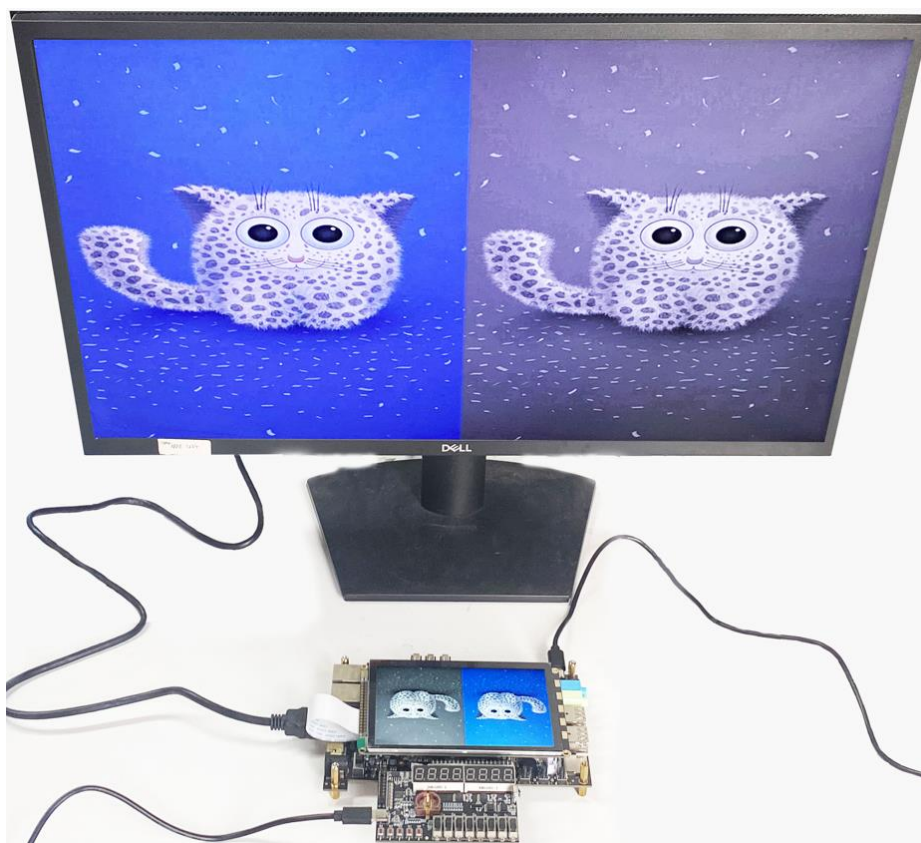


图 40-27 串口传图灰度转换显示效果

通过对比可以看出，图像左右两边分别显示的是原始的彩色图像和灰度化之后的灰度图像。显示的图像完整且正确，说明设计成功。

接下来，再来演示测试数据下的显示效果。在设计顶层中通过条件编译，使用测试数据，将 `DISP_MODE` 的值设置为 `2'd0`，使用测试模式 0 的数据。重新编译设计并导出硬件描述文件后，再次烧录，TFT 屏和 HDMI 显示器上的显示效果如图 40-28 所示：

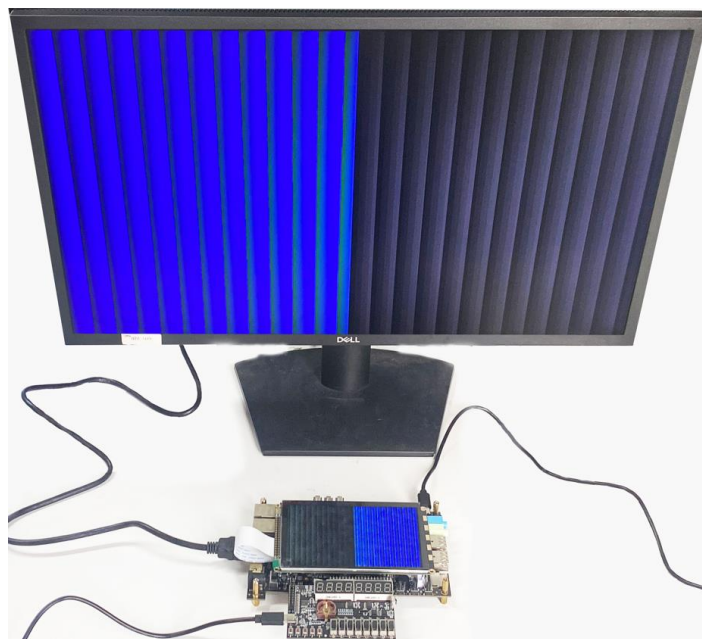


图 40-28 使用测试数据，模式 0

接下来分别测试模式 1、模式 2、模式 3，对应的显示效果分别如下：

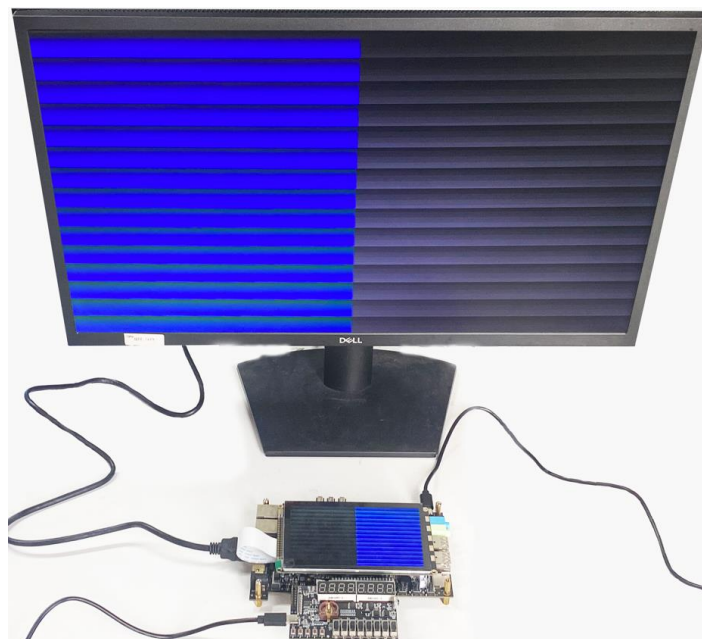


图 40-29 使用测试数据，模式 1

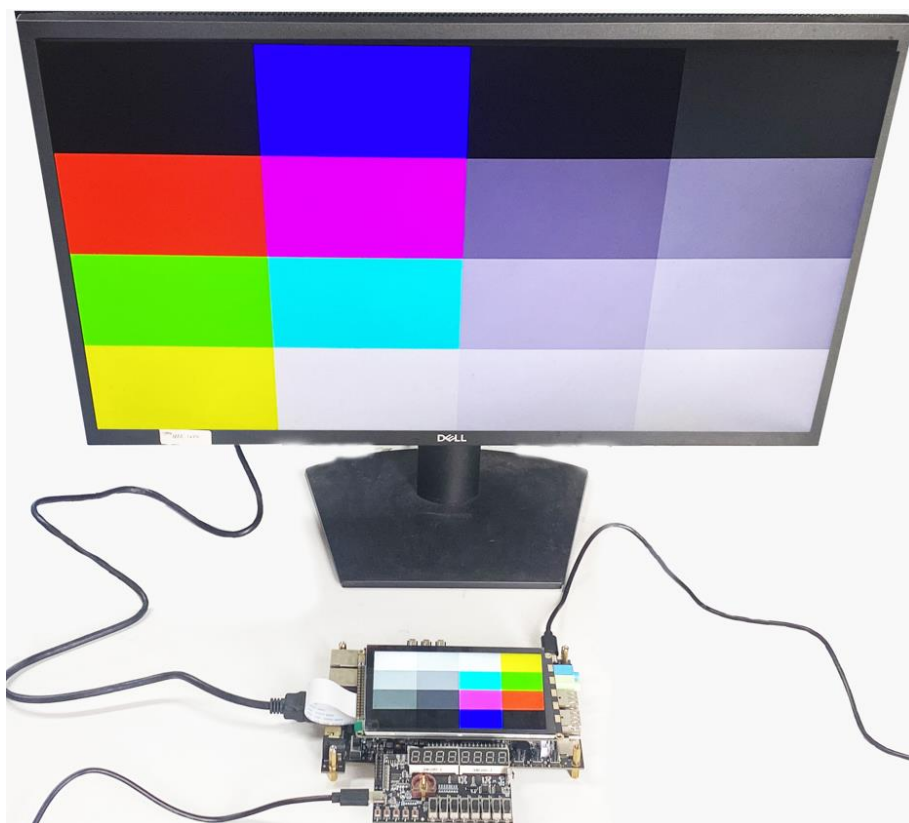


图 40-30 使用测试数据，模式 2



图 40-31 使用测试数据，模式 3

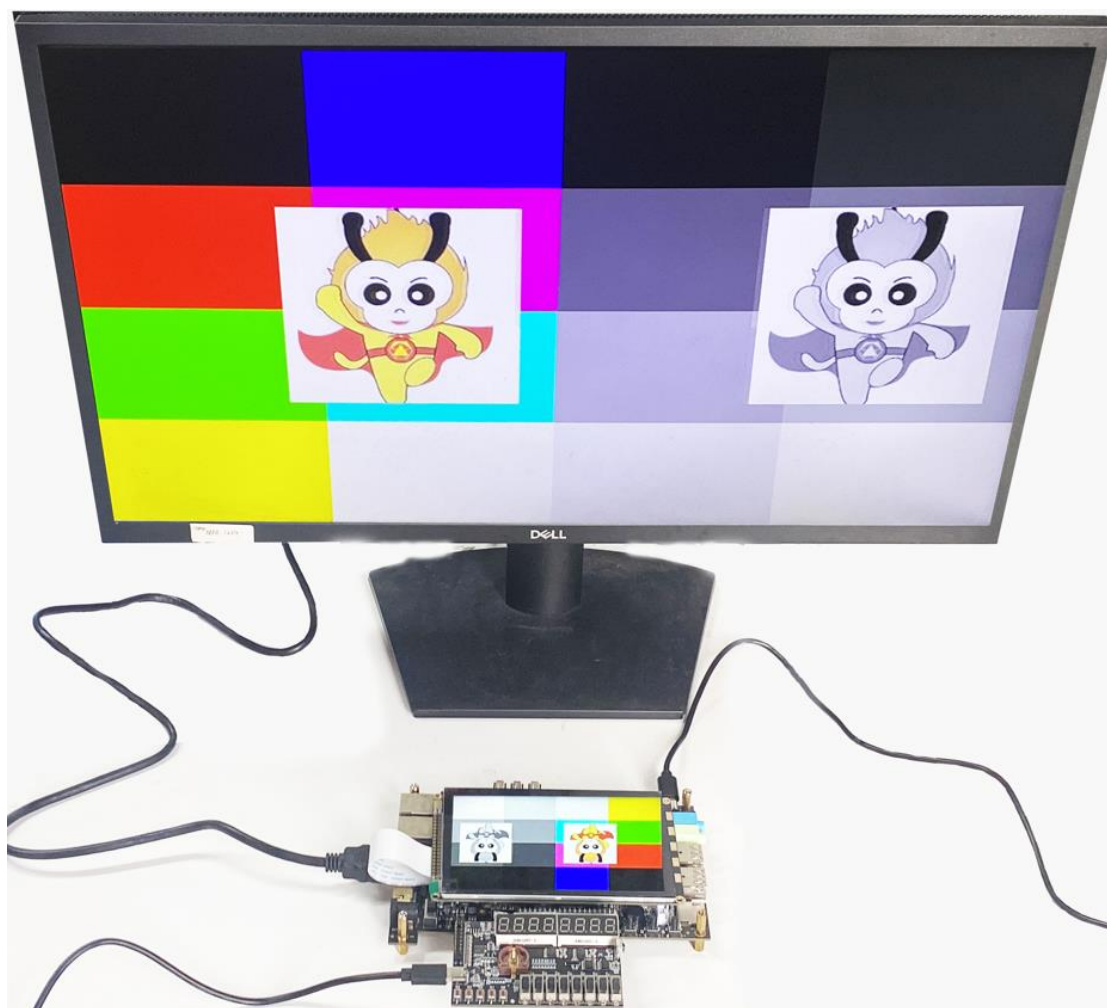


图 40-32 使用测试数据，模式 3（续）

其中模式 3 中的小梅哥 log 会从图像右侧逐渐向左侧移动，在达到对应图像左侧 1/8 处时，log 会向右移动，如此循环往复。

可以看到，无论是测试数据还是串口传图，数据都能够被有效且正确的灰度化并显示到正确位置，本次设计成功。

40.8 总结

实现彩色图像灰度化，是实现后续图像滤波的基础。经过灰度化处理后的图像，能够有效减少图像上的非关注信息存储容量。这对于后期图像的处理和识别，是有很大帮助的。