

1 OV5640 实时 RAW2RGB 采集显示系统

工程源码	-02_设计实例 -- acz702_ov5640_ddr3_raw2rgb.zip -- acz702_ov5640_ddr3_raw2rgb_tft_hdmi.zip -- RAW2RGB.zip
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

RAW 格式数据作为数字图像传感器输出的原始数据，由于没有经过各种转换处理，图像数据失真较小，十分适合进行图像处理，在许多场景下都有广泛的用途。本节以 RAW 格式数据为实验对象，以 RAW2RGB 模块为核心，实现了 RAW 格式数据到 RGB888 格式数据的转换。

本节首先对 RAW 数据及其数据格式进行讲解，随后讲述了 RAW2RGB 模块的工作原理，包括如何在节省硬件资源的情况下获得相邻的四个 RAW 数据像素，如何确定四个像素点的排列顺序，以及如何根据排列顺序计算出相应的 RGB 数据值，并在图像采集显示系统中应用此模块，完成摄像头 RAW 数据的采集转换和显示。

1.1 原始数据 RAW

在基于 FPGA 的图像采集显示系统中，我们逃不开的一点就是对图像数据的采集与处理。图像数据在被摄像头的感光单元捕捉后会形成 Bayer 格式的原始图像数据，也就是图像的电子底片。而我们平常所接触到的数据格式，如 RGB565、RGB888、YUV422 等，都是在原始 RAW 数据基础上经过摄像头内置的 ISP 等模块加工处理后得到的。由于数据经过了加工处理，往往会有部分数据会被舍弃，因而很难获得最佳画质。

作为镜头捕捉到的最原始的数据，RAW 格式数据与其说是图像文件，不如说是一个数据包。这个数据包由于未经过相机内的影像生成器转换，所以除了曝光量是提前设置好的外，其余的如对比度，色温，饱和度等都可以按自己需求在后期调整。因为是直接对原数据进行处理，所以对图像的损失会非常小。

同时，RAW 格式数据对黑白之间影调的范围也非常的大。JPEG 格式的文件都是 8 位的数据文件，只包含 256 级影调变化，OV5640 摄像头支持 8 位或 10

位的 RAW 格式数据输出，也就代表着其最多有 1024 级影调变化。通常 250 级左右的影调便能表达出柔和、自然的画面，所以一般 250 级左右的影调图像文件便能满足大部分的需求。考虑到摄像头输出接口位宽为 8，本次实验采用的是 8 位位宽的 RAW 格式数据。

RAW 格式本身的优点非常多，而其适合图像处理的这一点也非常符合本次实验的需求，通过将 RAW 数据与其相邻的三个数据进行插值计算，我们可以转换出该点对应的 RGB888 像素值。每个点都会与其相邻的像素点进行四次运算，最终每个像素点都能实现从 RAW 到 RGB888 的转变，同时经由相应的模块输送到显示设备上显示。

1.2 Bayer 色彩矩阵空间

1.2.1 色彩感光原理

要想实现对 RAW 格式数据到 RGB888 格式数据的转换，我们首先要了解 RAW 格式数据的结构。这里以 OV5640 这款 500W 像素的 CMOS 摄像头为例，介绍其成像原理。

OV5640 是一个典型的 CMOS 图像传感器，作为一个图像传感器，其主要作用就是将现实中的各种光线转换为数字系统能够识别的数字信号。光线中三元色各个颜色的强度本身是模拟信号，所以图像传感器的最基本的原理就是进行模数转换，将光线这个模拟量转换为数字信号。

仅仅有模数转换功能还不够，我们还得先搞清楚光线是一种怎样的模拟量。在初中物理中已经有过详细的介绍：自然界中的光，实际上是三种基本单色光的组合，这三种基本单色光为红（RED）、绿（GREEN）、蓝（BLUE），我们称之为三原色。通过将这三种基本颜色按照不同的比例混合，就可以得到其他的任意颜色。例如纯黄色是由红色和绿色按照一比一的比例混合得到的，蓝色量为 0。下图图 1-1 为三种颜色混合得到几种常见颜色的示意图。

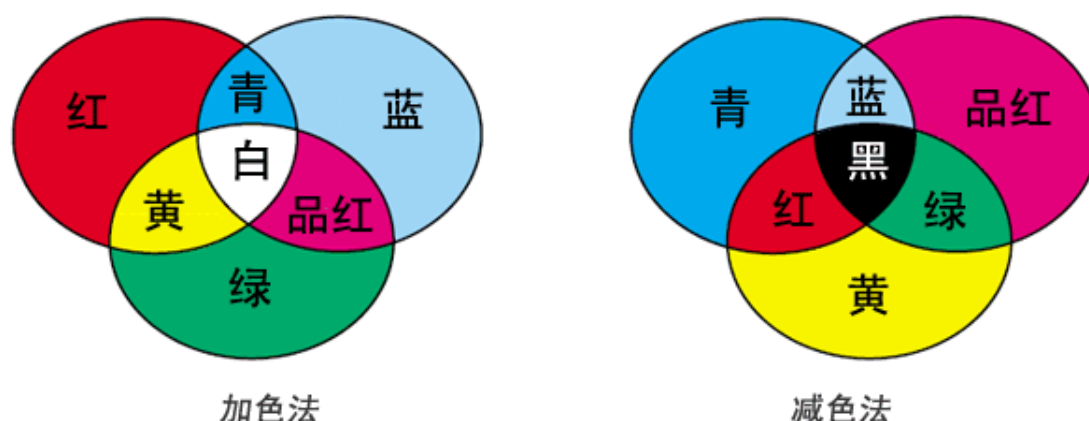


图 1-1 几种常见颜色示意图

既然已经知道，每一束光线都可以理解为三原色按照不同比例混合得到的效果，那么我们只要想办法知道该束光线的三原色的比例，然后用颜色加比例的表示方法，就能唯一确定这束光线的最终颜色了。所以图像传感器里面所谓的模数转换，实质就是对一束光线的三原色的强度进行转换，将三原色中每一种颜色的强度转化为数字信号。再用颜色加数字的方式来表示该束光线的真实颜色。

这里，假如我们对三原色中的每一种基本颜色的强度都分为 256 级，那么每一种基本颜色的强度就可以用一个 8 位的数字来表示，0 表示该颜色强度最弱，或者说无该颜色分量，255 表示该颜色强度最强。这样一来，就可以用一个 24 位二进制的数字来唯一表示该光线的颜色了。上图中几种颜色的对应三原色的数值如表 1-1 所示：

表 1-1 几种常见颜色的三原色表示

	红	绿	蓝	黄	品红	青	白	黑
红	255	0	0	255	0	255	255	0
绿	0	255	0	255	255	0	255	0
蓝	0	0	255	0	255	255	255	0

这种表示颜色的方法就是最常见的 RGB888 格式。所谓 RGB888 就是使用 3 个 8 位的数据表示一种颜色，其中高八位表示红色分量，中八位表示绿色分量，低八位表示蓝色分量，上述 8 种颜色用 RGB888 格式表示就如下表表 1-2 所示：

表 1-2 几种常见颜色的 RGB888 表示

	红	绿	蓝	黄	品红	青	白	黑
RGB888	0xff0000	0x00ff00	0x0000ff	0xffff00	0x00ffff	0xff00ff	0xffffffff	0x000000

通过上面的分析我们就可以知道，只要模数转换器能够对每一束光的三原

色的强度进行转换得到数字信号，就能唯一表示该颜色了。但是接下来，另一个问题又出现了。如何才能对一束自然光中的三原色的强度分别进行模数转换呢？这就涉及到对一束光的三原色分离。

所谓对光的三原色分离就是通过某种手段，将该束光线中的三种颜色分别独立提取出来，当三种颜色都独立的提取到之后，就能使用模数转换器对该颜色的强度进行转换了。三原色分离的原理其实非常简单，就是使用单色滤光片，如图 1-2：

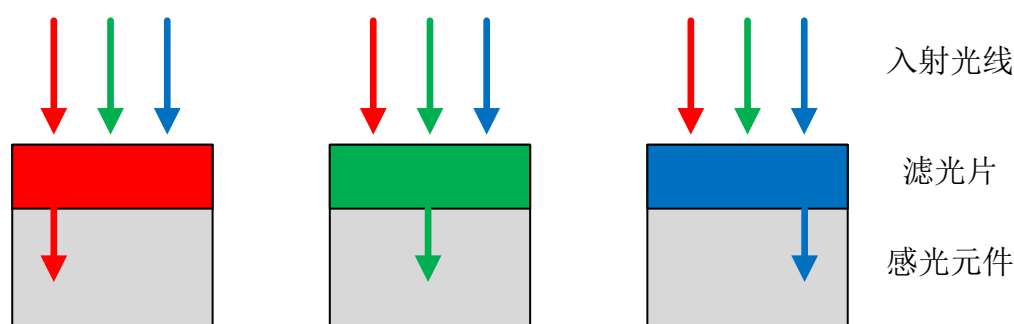


图 1-2 单色滤光片

通过加入滤光片，就能让对应颜色的光线通过滤光片到达感光元件，通过这种方式，只需要三个不同颜色的滤光片，就能对一束入射的复合光线进行分离，得到三原色，然后使用模数转换器对感光元件感应到的单色光的光照强度进行转换，就能得到该单色光的强度数字值了。

通过上述分析我们也发现，一个感光元件只能对一种颜色的光进行感应，如果要对一束光线中的三种颜色分别感应，就需要三个感光元件。如果对应到 CMOS 图像传感器里面的概念来说，这每一个滤光片加感光元件都是一个像素。注意这个概念，每一个滤光片加感光元件都是一个像素，那么从反面来说就是，每个像素都只能感应一种颜色的光线。

既然每个像素只能感应一种颜色，那么新一个问题又来了——如何才能得到某束光线的三种颜色分量的值呢？答案就是使用至少 3 个像素，每个像素感应该束光的一种颜色，然后三个像素就能把该束光的三种颜色分量全提取出来了。

但是，使用 3 个像素来提取一束光的三种颜色分量，虽然理论上可行，实际上却存在 3 个像素间摆放位置的物理限制。首先是如何让该束光线能够均匀的照射到感应三种颜色分量光线的像素上的问题，其次还要知道，一个图像传感器，并不是只感应一束光，是要感应成千上万束细小的光束，每一束光线都

需要有对应的像素组来提取其三种颜色分量。因此，像素和像素之间的物理摆放问题也是需要认真考虑的。同时，还得兼顾考虑这种摆放模式下图像传感器的可生产性问题。

正是为了解决这些问题，诞生了著名的拜尔（以下简称 Bayer）矩阵。

1.2.1.1 Bayer 矩阵定义

感光像素矩阵中，每个奇数行间隔放置绿色和红色感应像素，每个偶数行间隔放置绿色和蓝色感应像素，每个奇数列间隔放置绿色和蓝色感应像素，每个偶数列间隔放置红色和绿色感应像素，其输出数据格式如图 1-3 所示：

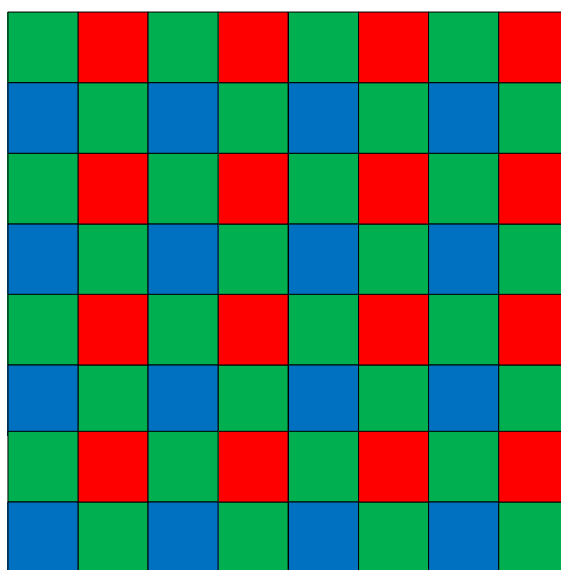


图 1-3 Bayer 矩阵

根据这种分布规律，在 Bayer 矩阵中，以相邻的四个像素作为一组，在该组中，有两个感应绿色分量的像素呈对角分布，另外两个像素则分别对应感应红色分量和感应蓝色分量。任取一个像素，其与相邻的 3 个像素组成的矩阵中，总符合该规律。不同的只是三种颜色的像素所在的位置的差异。图 1-4 为图像传感器手册中给出的像素物理分布图。

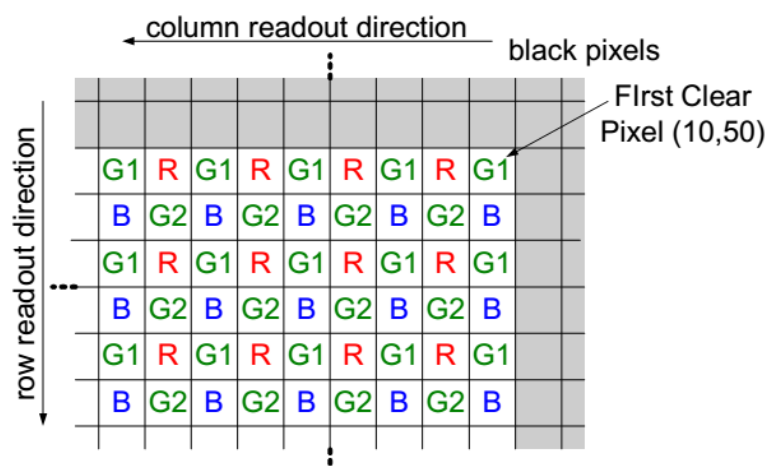


图 1-4 像素物理分布图

通过对上述 8*8 的矩阵进行分析发现，在整个矩阵中，像素的位置关系有且只有图 1-5 所示四种情况：

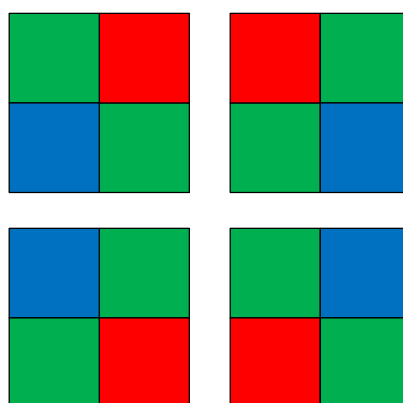


图 1-5 像素的四种位置关系

所以，在实际颜色提取时，需要通过数据转换，将相邻四个像素的数据通过插值算法合并为一个 RGB 像素颜色，此种转换算法名为 RAW2RGB，这里取左上角四个像素点的数据为例，具体颜色转换算法如图 1-6 所示：



图 1-6 相邻四种颜色分量所代表含义

保留相邻四个像素中的红色和蓝色分量，而对两个绿色分量求平均，得到新的绿色分量，此三种颜色分量组成一个新的 RGB 格式的像素，新的像素色彩组成如图 1-7 所示：



图 1-7 像素色彩计算

按照这样的思路，整个图像传感器中的每一个像素都可以以不同的角色参与 4 次运算，并最终得到 4 个 RGB 颜色值，所以理论来说，还是可以认为一个图像传感器有多少个物理像素，就能得到多少个 RGB 格式的像素值，虽然每个物理像素都只能感应一种颜色。但是经过插值运算后，其能输出 RGB 的数据格式的像素数量还是等于其物理像素个数的。

有了 Bayer 像素矩阵后，只需要使用一个模数转换器，依次对每一个像素的感光元件感应到的模拟量进行转换并输出，就能得到一幅完整图像的原始像素信息了。

1.3 RAW 转 RGB 算法的 FPGA 实现

在有了一幅完整图像的 RAW 数据后我们便能开始着手对其进行插值运算了，但是直接将整幅图像存储下来再进行运算将会消耗大量的存储资源。观察 Bayer 矩阵我们可以发现对任意一个像素点进行差值运算时，只要知道其右边点的数据及下一行与这两个点相邻的数据，也就是只要知道下一个数据以及下一行该点正下方和右下方的数据即可。至于如何获取这些数据，就涉及到我们接下来要讲的 2x2 插值矩阵了。

1.3.1 2x2 插值矩阵的获取

在基于 FPGA 的图像处理中，我们一般通过使用 RAM-based Shift Register IP 核的方式来实现插值矩阵的获取。该 IP 核为移位寄存器，是一种存储器模型，用户可以设置移位寄存器的位宽和深度来控制对数据的延迟。

如果让移位寄存器的位宽与 RAW 格式数据一致，深度等于一行图像的像素个数，那么当一行图像的像素点依次移入到移位寄存器中后，接下来再输入进来的两个数据必然为下一行图像的前两位数据。取移位寄存器输出端（最先移入移位寄存器的数据会最先出现在输出端）的连续两个数据，与最新从图像传感器输出的连续的两个像素的数据，就能刚好组合成一个 2x2 插值运算矩阵，

以图 1-8 例：

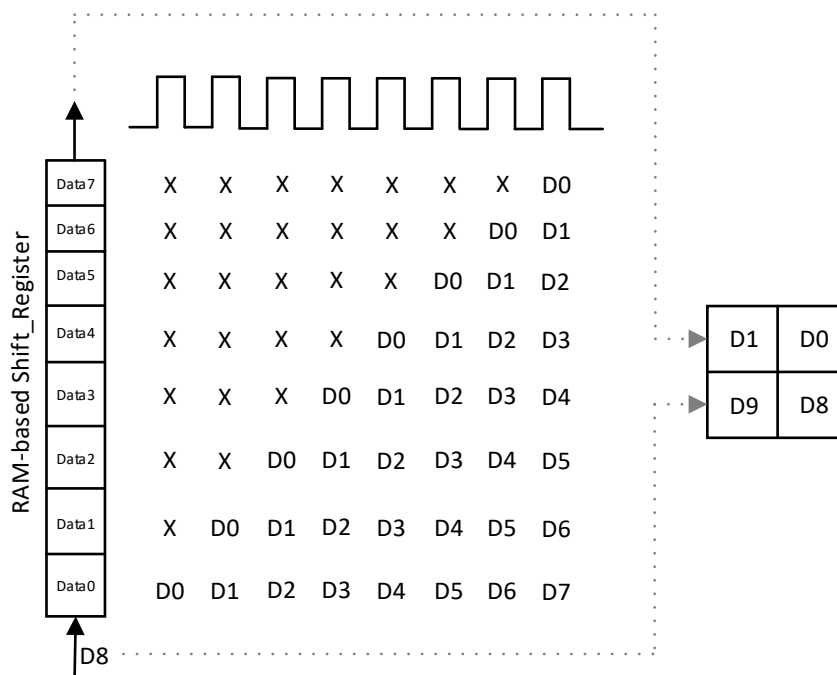


图 1-8 RAM-based shift 寄存器与两级缓存

假设我们一行只有 8 个数据，第一行第一个数据为 D0，第二个数据为 D1，依次累加，那么第二行第一个数据，第二个数据分别为 D8、D9。D0、D1 为第一行输入的前两个数，D8、D9 刚好为第二行输入的前两个数，知道了这四个值，我们就能按照其排列顺序计算出 D8 的 RGB 像素值。注意图 8 中的 2X2 矩阵是寄存器中的排列关系，转换到 2X2 插值运算矩阵中 D0 与 D1，D8 与 D9 的顺序是相反的。

1.3.1.1 RAM-based Shift Register

上文提到的 RAM-based Shift Register 为移位寄存器，是一种存储器模型，Vivado 中提供了实现该存储器的 IP 核，在 IP Catalog 中搜索其名字便能找到该 IP。

该 IP 核共有三个配置页面，第一个配置页面如图 1-9 所示：

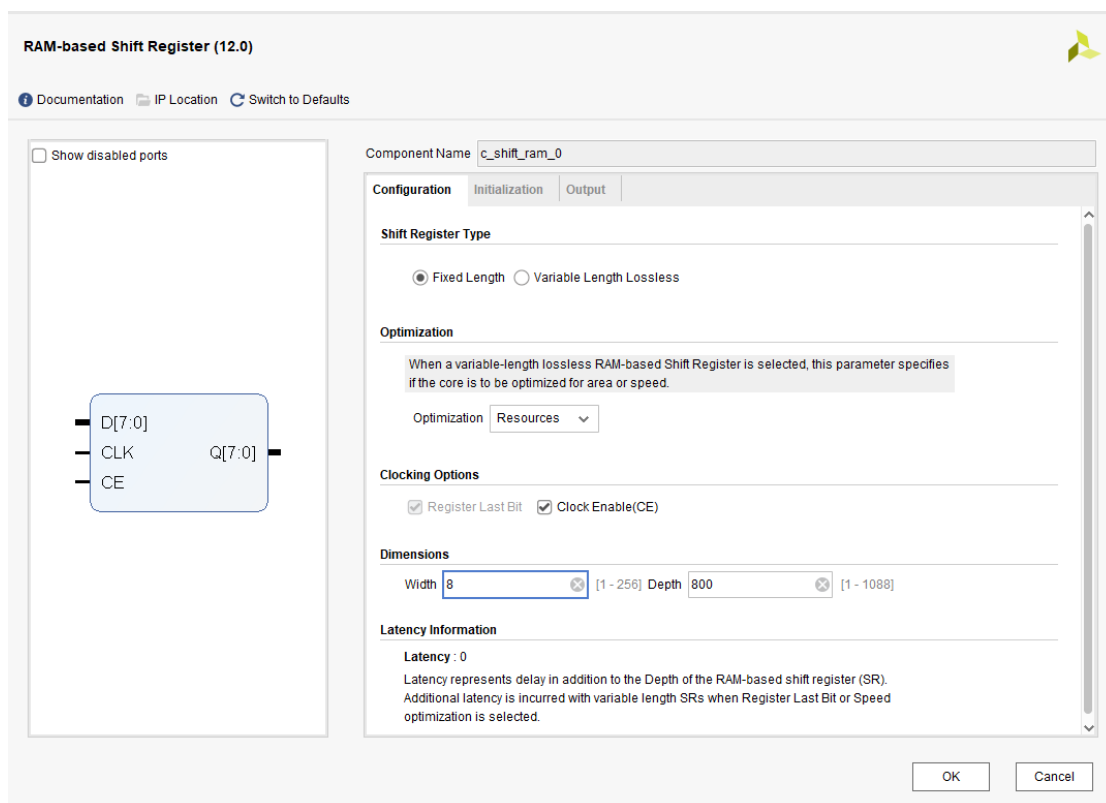


图 1-9 RAM-based Shift Register 配置页面

这里简单为大家介绍下其中较常用到的几个配置项：

- **Fixed Length:** 固定长度模式，该模式下，数据从进入寄存器到从寄存器中输出所经历的延迟（时钟数）为设定的固定值。
- **Variable Length Lossless:** 无损可变长度模式，该模式下，数据从进入寄存器到从寄存器中输出所经历的延迟是可变的，由该模式下新增的接口 A[M:0]的输入值决定。
- **Clock Enable:** 时钟使能控制，只有当时钟使能时，移位寄存器才会工作
- **Width:** 移位寄存器输入数据的位宽
- **Depth:** 移位寄存器的深度，固定长度模式下决定着数据从进入寄存器到从寄存器中输出所经历的延迟。

第二个配置页面如图 1-10 所示：

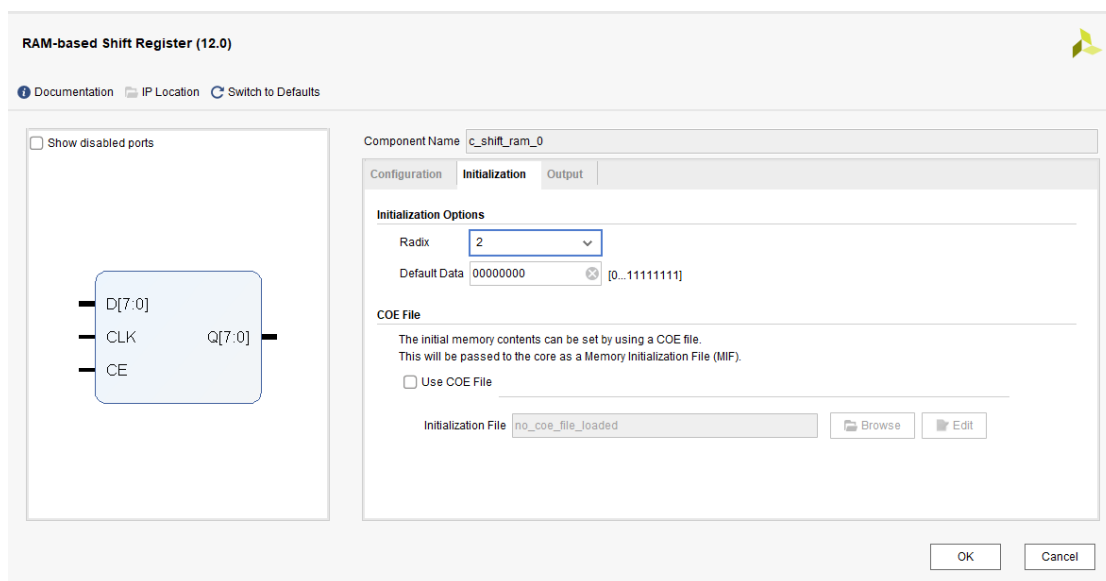


图 1-10 初始化配置页面

可以看到该页面用来配置移位寄存器的初始化，页面中的几个配置项功能如下：

- **Radix:** 设置初始化时默认值的进制
- **Default DATA:** 设置初始化时，移位寄存器内部数据的默认值
- **Use COE File:** 勾选该项后，IP 将会支持使用 COE 文件来设置 IP 核的初始值，用户指定有效的 COE 文件路径后，寄存器在初始化时会按照 COE 文件中的内容设置移位寄存器内部的数据。

第三个配置页面如图 1-11 所示：

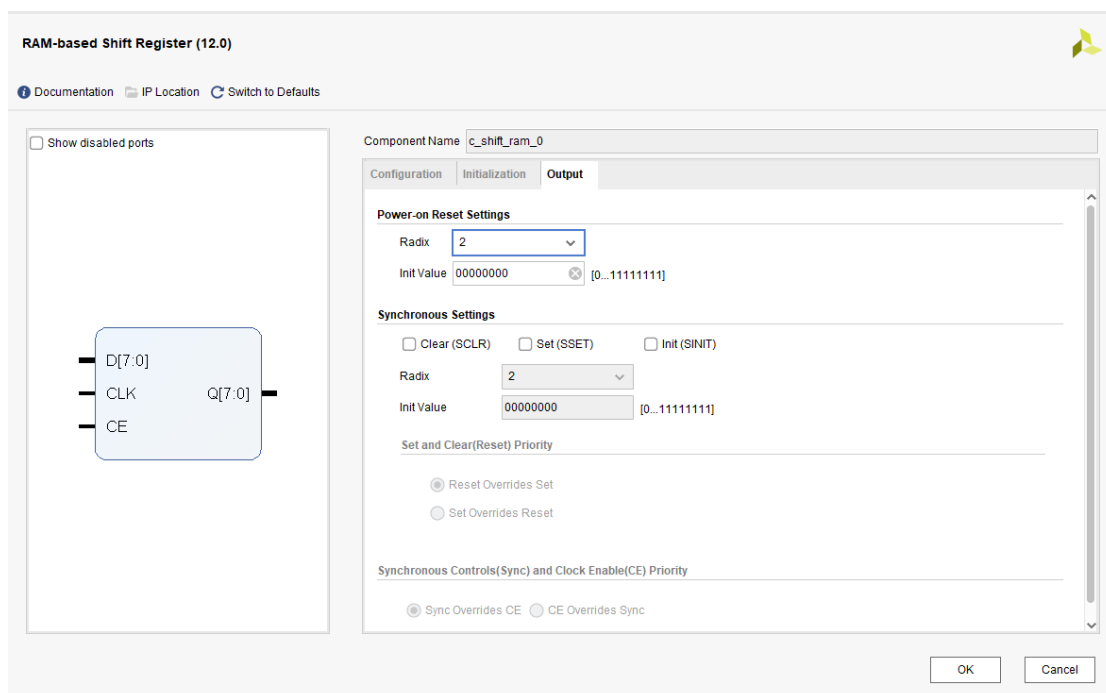


图 1-11 输出配置页面

该页面用来配置输出项，这里简单介绍下其中几个配置项：

- Clear(SCLR): 可选同步清除。当该引脚为高电平时，强制输出为 0
- Set(SSET): 可选同步设置，当该引脚为高电平时，强制每位输出为 1
- Init(SINIT): 同步初始化，当该引脚为高电平时，强制输出为默认值
- Set and Clear(Reset)Priority: 该项控制 SCLR 和 SSET 的优先级。
- Synchronous Controls (Sync) and Clock Enable (CE) Priority: 该项用来设置 Sync (SCLR、SSET、SINIT) 与 CE 的优先级。

以本次设计为例，摄像头输出的是 8 位的 RAW 格式数据，因此，移位寄存器的 Width 就需要设置为 8。假设我们传输的图像数据尺寸为 800*480，为了获得 2x2 插值矩阵，我们就需要让图像数据延迟一个固定的时钟周期，也就是一行图像的数据量，即 800。因此，我们就可以设置移位寄存器为固定长度模式，将 Depth 设置为 800。而为了能够随时控制移位寄存器的工作状态，我们还需要勾选 Clock Enable。至于其余项，我们保持默认值即可。

当我们设置好移位寄存器对一行图像数据缓存后，接下来我们就需要想办法得到相邻的四个数据。

1.3.1.2 得到 2*2 矩阵

在上面 RAW2RGB 原理中有说到，当移位寄存器中一次存储了一行数据的时候，只需要对其输入端与输出端的前两位进行差值计算就能得出输出端第一位数据的 RGB 像素值。所以我们只需在数据输入和输出 RAM-based Shift Register 时对其进行两级缓存即可，该部分代码如下：

```
wire [7:0]RAW_Data;
reg [7:0]r0_RAW_Data;
reg [7:0]r1_RAW_Data;
wire [7:0]RAW_Data_s;
reg [7:0]r0_RAW_Data_s;
reg [7:0]r1_RAW_Data_s;

always@(posedge Clk)begin
    r0_RAW_Data <= #1 r1_RAW_Data;
    r1_RAW_Data <= #1 RAW_Data;

    r0_RAW_Data_s <= #1 r1_RAW_Data_s;
    r1_RAW_Data_s <= #1 RAW_Data_s;
end
```

这段代码中，RAW_Data 为移位寄存器的输入数据，RAW_Data_s 为移位寄存器的输出数据，这两个数据经过两级寄存后便能得到四个数据，构成一个 2X2 插值矩阵，如图 1-12 所示：

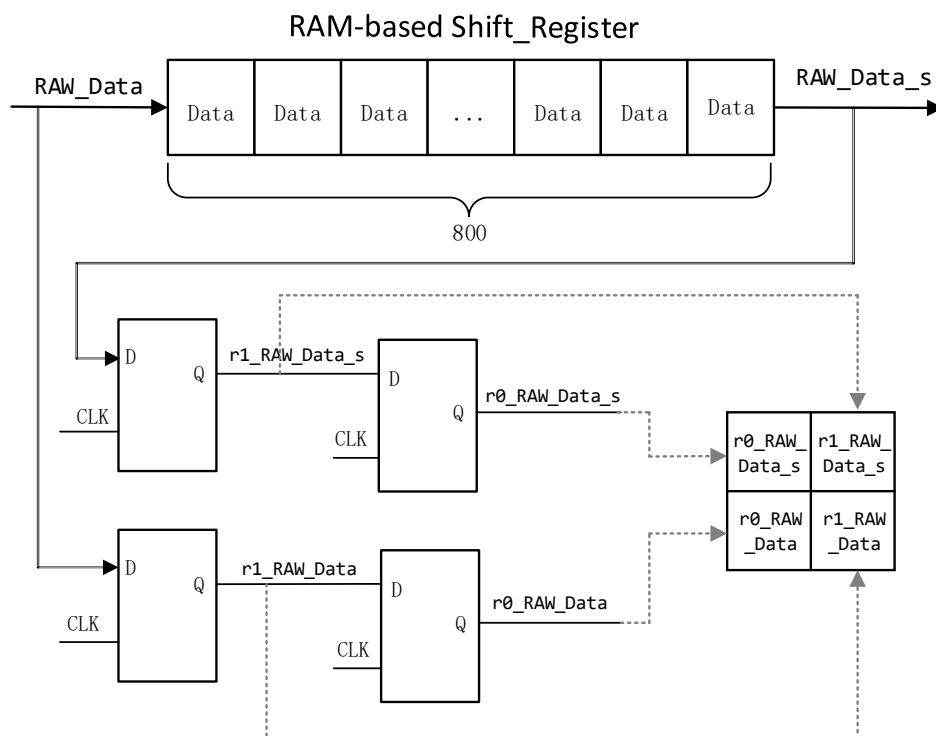


图 1-12 插值矩阵获取

这里需要注意的一点是，数据在移位寄存器中的排列顺序与在像素矩阵（或 2x2 矩阵）中的排列顺序是不一样的。例如数据在写入移位寄存器中时，新写入的数据总在左侧，即 r1_RAW_Data 应该在 r0_RAW_Data 的左侧。而这两个数据在经过两级寄存后，先被寄存的 r0_RAW_Data 会先输出到 2x2 矩阵中，并存放于相对左侧的位置，后寄存的 r1_RAW_Data 则位于右侧。数据在从移位寄存器中输出，经过两级寄存器后在 2x2 矩阵中的位置关系也同理。

至此，我们便获得了所需的 2x2 矩阵，接下来只需对该 2X2 矩阵进行插值运算就可以得到相应点的 RGB 像素值。但这也产生了一个新的问题，那就是我们该如何确定 2x2 矩阵中颜色分量的排布关系。

1.3.2 像素位置的确定和颜色转换

在 OV5640 的官方数据手册中给出了如图 1-13 像素物理分布图：

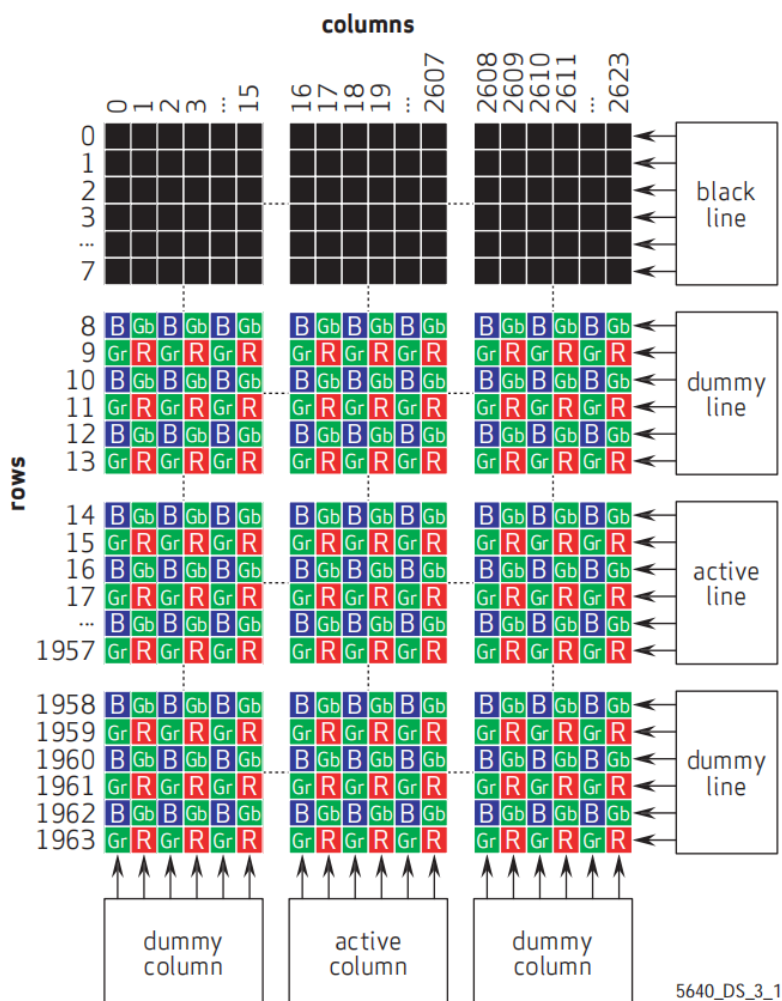


图 1-13 像素物理分布

由该分布图可看到，OV5640 内部有一个 2624 X 1964 的像素矩阵，但是实际能够输出的部分仅仅为 active line 与 active column 交汇的像素点，也就是中间的像素矩阵，其余部分用于校准和差值。由于感光像素矩阵中，每个奇数行间隔放置绿色(Gr)和红色(R)感应像素，每个偶数行间隔放置绿色(Gb)和蓝色(B)感应像素，每个奇数列间隔放置绿色和蓝色感应像素，每个偶数列间隔放置红色和绿色感应像素观察像素的行和列。我们很容易就可以发现只要知道了像素点坐标的奇偶值就能判断出它对应的颜色分量，又由于在 2X2 矩阵中红色与蓝色为对角关系，两个绿色为对角关系，我们用 0 和 1 代表像素点行与列的奇偶值，0 代表偶数，1 代表奇数，按图 3 所得到的规律，从图 1-13 像素物理分布截取了四个像素矩阵得到图 1-14 像素位置关系各像素的位置与像素排列的关系：

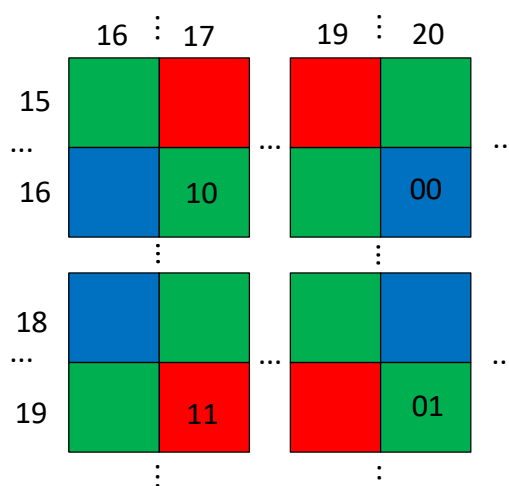


图 1-14 像素位置关系

可以看到，当坐标像素点为第 16 行第 17 列时，其相应的坐标则为 (17,16)，取坐标的奇偶值便能得出该点为 10，此时红色分量与蓝色分量分别位于 2X2 像素矩阵的右上角和左下角。同理，也可推出其余三种情况。

在知道了像素的排列位置后，我们便能按照相应的插值计算求出相应点的 RGB 像素值。至于像素点的位置坐标，我们只需通过取 disp_driver 模块中场同步扫描计数器与列同步扫描计数器的最低位即可得到。为了方便设计，我们取 2X2 矩阵中最新输入的数据，即右下角的像素位置作为定位，也就是两级缓存的四个数据中的 r1_RAW_Data。

为了方便理解，我们对 2X2 矩阵中四个位置进行命名，左上角为 DLU，D 为 data，L 为 left，U 为 up；右上角为 DRU，R 为 right；左下角为 DLD，第二

个 D 为 down; 右下角为 DRD, 可得出图 1-15 的关系:

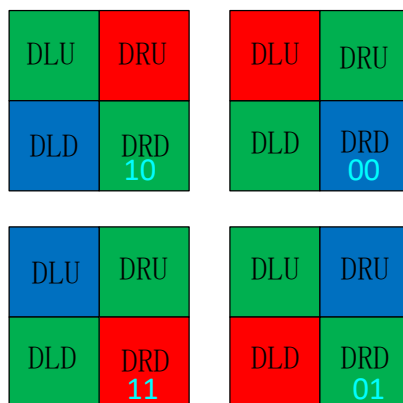


图 1-15 2X2 插值矩阵中像素四种对应关系

当右下角行与列奇偶值为 10 时, 绿色像素分量值为 $(DLU+DRD)/2$, 红色像素分量值为 DRU, 蓝色像素分量值为 DLD。同理其他情况下三种颜色分量的值也能一一求出。该部分代码设计如下:

```

wire [7:0]DLU = r0_RAW_Data_s;
wire [7:0]DRU = r1_RAW_Data_s;
wire [7:0]DLD = r0_RAW_Data;
wire [7:0]DRD = r1_RAW_Data;
always@(posedge Clk)begin
r_Xaddr <= #1 Xaddr;
r_Yaddr <= #1 Yaddr;
end

always@(posedge Clk)
case({r_Xaddr,r_Yaddr})
// R Gr
// Gb B
2'b00:
begin
RED <= #1 DLU;
GREEN_r <= #1 DLD + DRU;
BLUE <= #1 DRD;
end

// Gb B
// R Gr
2'b01:
begin
RED <= #1 DLD;
GREEN_r <= #1 DLU + DRD;
BLUE <= #1 DRU;
end
end
    
```

```
// Gb R
// B Gr
2'b10:
begin
    RED <= #1 DRU;
    GREEN_r <= #1 DLU + DRD;
    BLUE <= #1 DLD;
end

// B Gb
// Gr R
2'b11:
begin
    RED <= #1 DRD;
    GREEN_r <= #1 DLD + DRU;
    BLUE <= #1 DLU;
end
endcase
```

通过对 Xaddr, Yaddr 像素在矩阵中行列坐标的最后一位, 进行一拍寄存, 我们可以得到 2x2 矩阵右下角, 也就是 r1_RAW_Data 信号的行场奇偶坐标。利用该坐标, 我们就能够判定出 2X2 矩阵中各像素分量的位置排列进而计算该点颜色分量。

在完成了像素位置定位后, 该模块的设计也已经基本完成, 接下来即可对模块进行仿真验证了。

1.4 RAW 转 RGB 模块仿真验证

为了模拟 RAW2RGB 模块的应用场景, 我们单独设计了一个名为 RAW2RGB 的工程, 工程中添加了 RAW2RGB 模块和 disp_driver 模块, 其中 RAW2RGB 模块包含了上文我们提到的 RAM-based Shift Register 核, 用于获取相邻的四个数据; RAW2RGB 模块用于数据从 RAW 格式到 RGB888 格式转换; disp_driver 模块用于产生对图像数据的行场同步计数, 进而对像素定位。

1.4.1 仿真测试激励

为了方便验证, 我们将输入 RAW2RGB 模块的数据设定为固定值, 当场同步信号为奇数值, 输入数据为 00f7; 当场同步信号为偶数值, 输入数据为 f700, 即最终得到的值为 R=F7,G=00,B=7F, 观察各个信号的数据变化。本例提供的工程的仿真激励代码如下:

```
`timescale 1ns/1ns

module RAW2RGB_tb;

    reg Clk;
    reg Rst_n;
    wire DinReq;
    wire [7:0]RAW_Data;

    wire [11:0]H_Addr;
    wire [11:0]V_Addr;

    reg DoutReq;
    wire [7:0]RED;
    wire [7:0]GREEN;
    wire [7:0]BLUE;
    wire [7:0]Disp_Red;
    wire [7:0]Disp_Green;
    wire [7:0]Disp_Blue;
    wire      DataReq;
    wire      Disp_HS;
    wire      Disp_VS;
    wire      Disp_DE;
    wire      Disp_PCLK;

    wire [15:0]data;

    RAW2RGB RAW2RGB(
        .Clk(Clk),
        .Rst_n(Rst_n),
        .DinReq(DinReq),
        .RAW_Data(RAW_Data),
        .Xaddr(H_Addr[0]),
        .Yaddr(V_Addr[0]),
        .RED(RED),
        .GREEN(GREEN),
        .BLUE(BLUE),
        .DoutReq(DataReq)
    );

    disp_driver disp_driver(
        .ClkDisp(Clk),
        .Rst_n(Rst_n),
        .Data({RED, GREEN, BLUE}),
        .DataReq(DataReq),
        .H_Addr(H_Addr),
        .V_Addr(V_Addr),
```

```
.Disp_HS(Disp_HS),
.Disp_VS(Disp_VS),
.Disp_Red(Disp_Red),
.Disp_Green(Disp_Green),
.Disp_Blue(Disp_Blue),
.Disp_DE(Disp_DE),
.Disp_PCLK(Disp_PCLK)
);

initial Clk = 1;
always#10 Clk = ~Clk;

initial begin
    Rst_n = 0;
    DoutReq = 0;
    #201;
    Rst_n = 1;
    #20000000;
    $stop;
end

reg flag;
always@(posedge Clk)
if(DinReq)
    flag <= #1 ~flag;
else
    flag <= 0;

assign data = V_Addr[0]?16'h00f7:16'h7f00;

assign RAW_Data = DinReq?flag?data[7:0]:data[15:8]:0;

endmodule
```

由于数据在输入时有 16 位，而 RAW2RGB 模块一次只能处理 8 位数据，因此增加了一个 flag 信号，让其在数据进来时每时钟翻转一次，从而依次读取输入数据的高 8 位，低 8 位。

1.4.2 仿真结果分析

本次设计仿真工程位于 RAW2RGB 文件夹下，双击运行即可仿真波形如下：

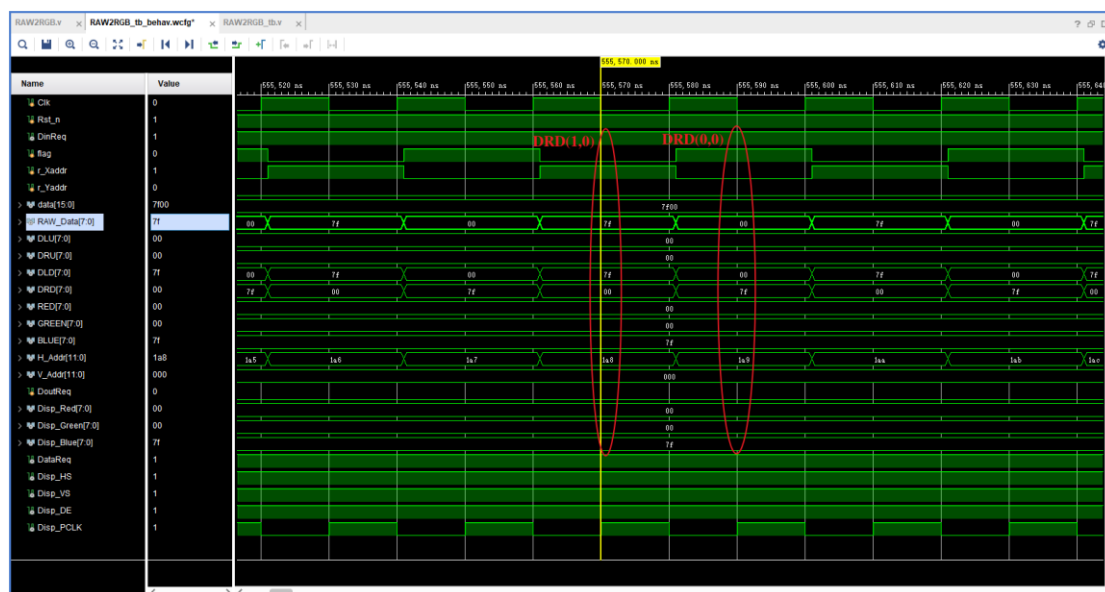


图 1-16 DRD（1,0）与 DRD（0,0）仿真波形

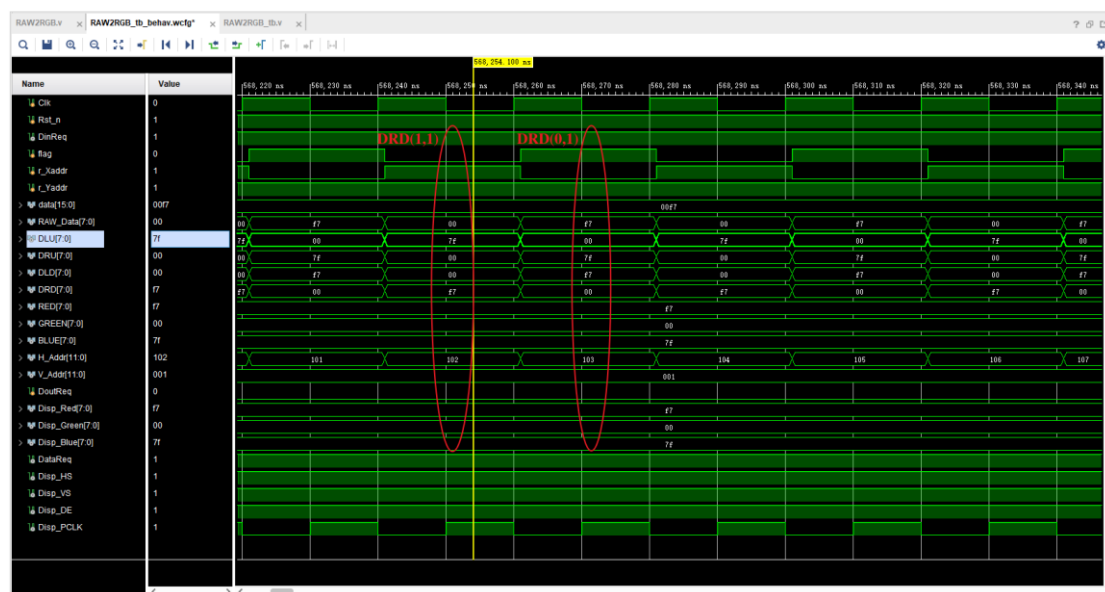


图 1-17 DRD（1,1）与 DRD（0,1）仿真波形

这里的 `r_Xaddr` 信号和 `r_Yaddr` 信号分别为行、列有效数据扫描计数器最低位的打拍信号（打拍是为了与数据同步），用于计算 RGB888 格式数据。

图 1-16 与图 1-17 中圈出了 RGB 颜色分量的四种排列情况下的仿真波形，以图中 DRD（1,1）时刻为例，可以看到，此时 `V_Addr[0]` 为 1，所以 `data` 为 00f7，由于 `flag` 值为 0，所以 `RAW_Data` 值为高 8 位，即 00，证明数据的输入无误。

由于 `r_Xaddr` 与 `r_Yaddr` 值都为 1，所以该点的行场奇偶坐标为（1,1），因此该点的颜色计算方式如图 1-18 中左下角的情况：

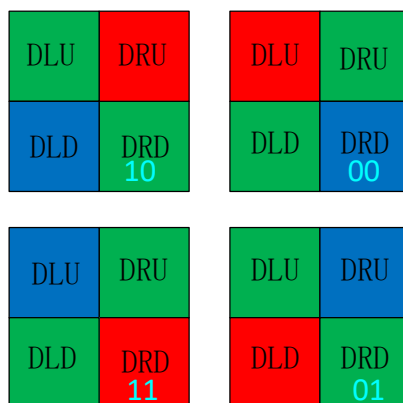


图 1-18 2X2 插值矩阵中像素四种对应关系

DLU, DRU, DLD, DRD 的含义在上文代码中已经讲过, 所以 RED 的值为 DRD 的值, 即 f7; GREEN 的值为 $(DLD+DRUD)/2$, 即 00; BLUE 的值为 DLU, 即 7f, 证明数据的输出无误。

同样的, 其他三种情况下的各颜色分量计算同理。

通过对各信号数据的对比计算, 验证了我们设计的 RAW2RGB 模块在仿真方面是没有问题的。设计 RAW 转 RGB 算法模块的最终目的, 是希望将其应用到实际的案例中, 将真实的图像数据转换为 RGB 图像, 以进行进一步的显示或处理。因此我们可以将其应用到具体的工程实例中, 对其进行板级调试和验证。

1.5 基于 OV5640 的实时 RAW2RGB 采集显示系统

要想直观地看到该模块是否能正常工作, 我们需要一个采集显示系统。因而我们将 RAW2RGB 模块与前面章节设计完成的 ov5640_ddr3_vga_io_hdmi 采集显示系统相结合, 设计得到一个能够将摄像头采集到的 RAW 格式图像数据, 在 FPGA 内部完成 RAW 转 RGB 转换后, 再将转换得到的 RGB 图像输出到显示设备上的工程。考虑到显示效果展示, 我们将工程中的 VGA 显示修改为 5 寸 TFT 显示, 用于表明设计能够实现开发板配套 TFT 屏的显示。TFT 屏本身只能显示 RGB565 格式数据, 原工程中的 HDMI 显示系统则正好用来验证设计的转换结果。

1.5.1 图像采集转换显示系统介绍

对于设计而言, 我们可以在 RAW 格式数据被存入 DDR3 前使用 RAW2RGB 模块进行 RGB888 的转换, 也可以在 RAW 格式数据被从 DDR3 中读出后进行 RGB888 的转换。考虑到前一种方式中, RAW 在转换为 RGB888 后, 会由原来

的 8 位数据变为 24 位数据，因此存储 RGB888 格式数据到 DDR 时将需要 3 倍于存储 RAW 格式数据时所需的带宽。出于节约性能的考虑，设计选择了后者，所以整个系统结构如图 1-19 所示：

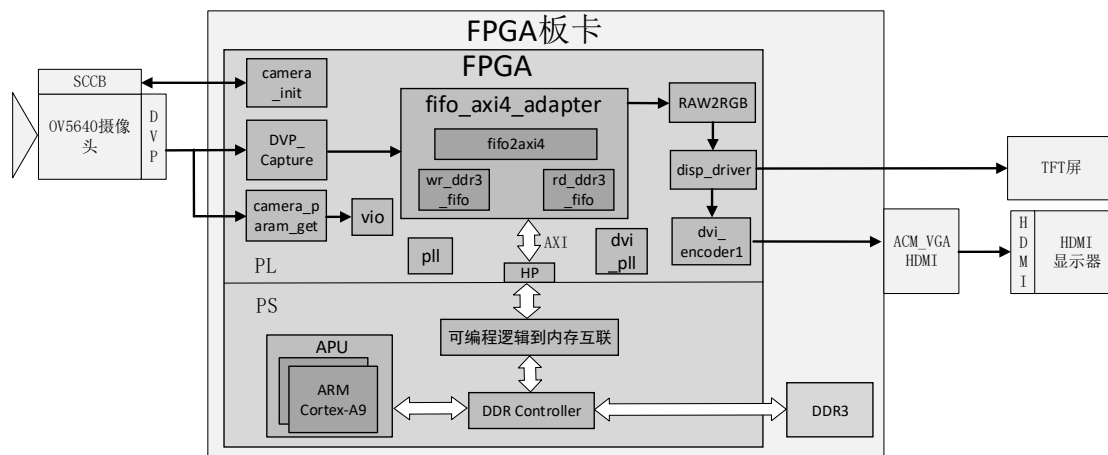


图 1-19 系统结构

考虑到 5 寸 TFT 屏的所支持的最大分辨率为 800*480，设计中摄像头输出的图像分辨率不能大于 800*480。整个系统在工作时，FPGA 首先会通过 SCCB 接口对 OV5640 配置，使其将采集到的图像数据以 RAW 格式输出 800*480 分辨率的原始图像数据。随后，摄像头通过 DVP 接口将 8 位的 RAW 格式数据输出给 FPGA。FPGA 内部再通过 DVP_Capture 模块将连续的两个 8 位数据拼接为 16 位写入到片外存储器 DDR3 中。数据在需要时被读出，送入到 RAW2RGB 模块中，并被转换为 RGB888 格式数据后输出给 disp_driver 模块。disp_driver 模块会根据设定的时序参数结合数据产生相应的时序信号，与数据一起输出。此时的 RGB888 图像数据和时序信号会被送给 TFT 屏和 dvi_encoder 模块。TFT 屏只能显示 RGB565 格式数据，因此需要对数据的各个颜色分量低位进行舍弃；RGB888 格式图像数据和时序在被送到 dvi_encoder 模块后会转换成符合 HDMI 接口的格式输出，驱动 HDMI 显示器显示。

1.5.2 配置 OV5640 输出 RAW 格式数据

设计使用原工程中的 OV5640 初始化寄存器表配置 OV5640 摄像头，该配置表中将 OV5640 摄像头相关寄存器地址以及地址对应的配置值存于二维数组 ROM 中。ROM 的位宽为 24 位，其中高 16 位用于存放寄存器地址，低 8 位用于存放对应的配置值。程序在运行时，摄像头初始化模块会读取该配置表，根据其中的地址以及配置值使用 IIC 初始化 OV5640 摄像头。

该配置表默认配置 OV5640 输出 RGB565 格式数据，设计需要的是 RAW 格式数据。这里我们需要修改 OV5640 中 0x4300 寄存器的值为 00，0x501f 寄存器的值为 03。前者用于控制输出数据的格式，后者用于控制摄像头 mux 的控制格式。这两个寄存器各个位的具体功能较为复杂，这里暂不详细介绍，用户可以参考 OV5640 手册“OV5640_CSP3_DS_2.01_Ruisipusheng”。

这两个寄存器存的地址以及配置值存放于配置表的 ROM[56]和 ROM[57]数组中，因此我们需要将 rom[56]与 rom[57]中的值分别修改为 24'h4300_00 和 24'h501f_03，如下：

```
rom[56] = 24'h4300_00; // RAW
rom[57] = 24'h501f_03; // RAW
```

除此之外，还需要设置摄像头的输出图像分辨率为 800*480，OV5640 摄像头的输出分辨率由 0x3808~380b 寄存器控制。配置表中使用了 parameter 对相关参数进行了全局修饰，因此，只需在顶层中，将 IMAGE_WIDTH 与 IMAGE_HEIGHT 信号的值分别改为 800 和 480 即可，代码如下：

```
parameter IMAGE_WIDTH = 800;
parameter IMAGE_HEIGHT = 480;
```

配置修改完成后，OV5640 摄像头便能按照需求，产生我们所需要的 800*480 分辨率的 RAW 格式数据。这些数据最终被存储进 DDR 中，在需要时，由 RAW2RGB 模块读出。

1.5.3 添加 RAW2RGB 模块

RAW2RGB 模块包含一个 RAM-based Shift Register IP 核，在添加 RAW2RGB 模块时，需要检查设计中是否包含该 IP。本次设计中该 IP 核的配置如图 1-20 和图 1-21：

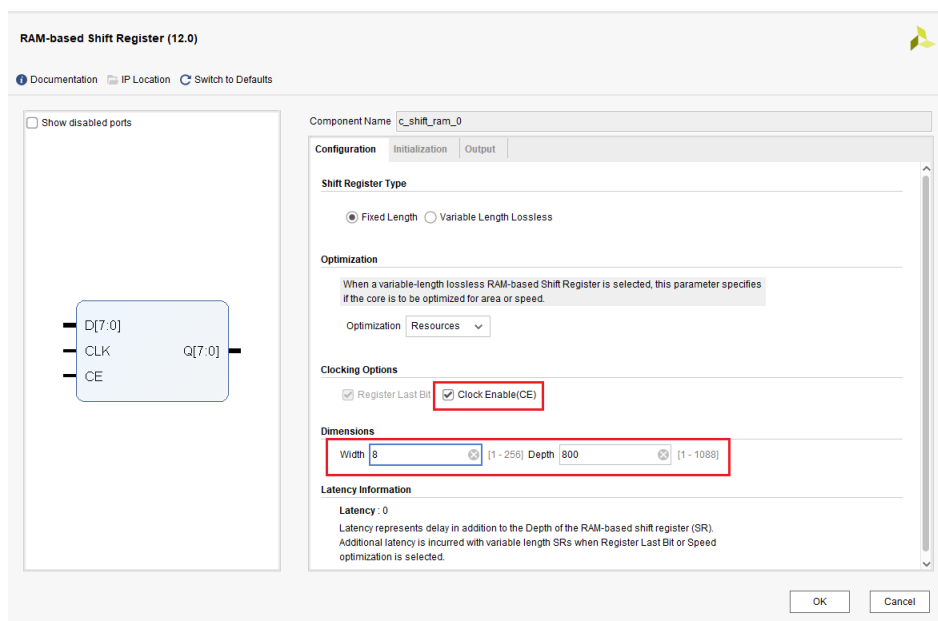


图 1-20 配置 RAM-based Shift Register

这里使能 CE，设置输入输出位宽为 RAW 数据的位宽，即 8；设置深度为一行图像的像素量，即 800。

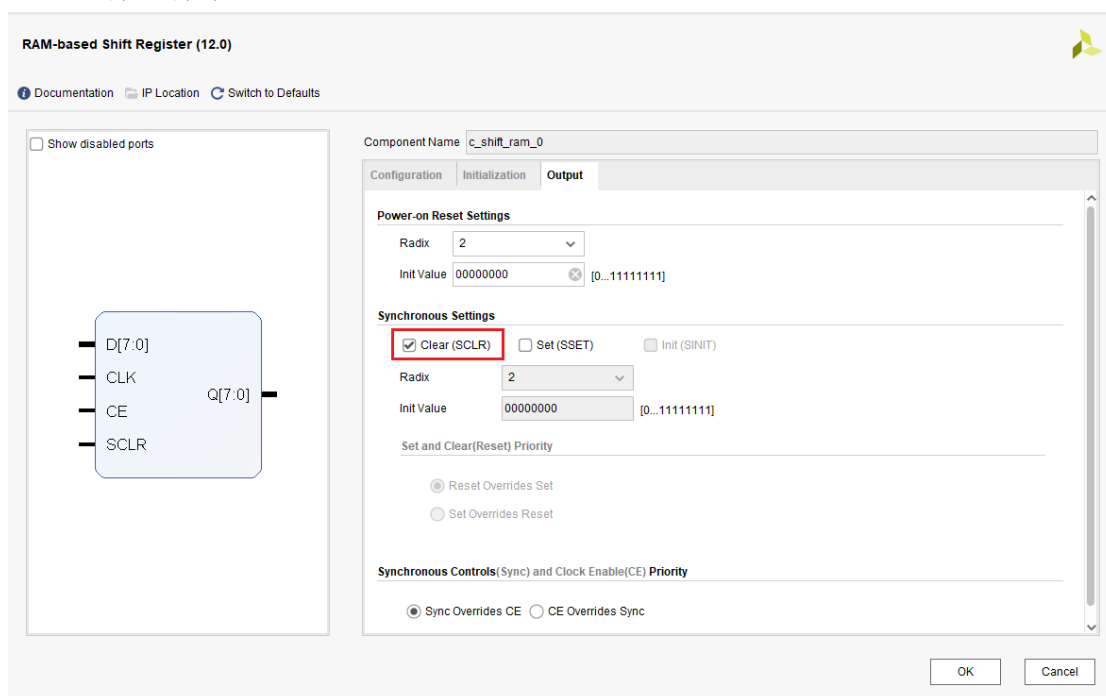


图 1-21 配置 RAM-based Shift Register

在输出项中勾选 clear，用于 IP 核的复位，该信号为高电平有效。配置并添加 RAM-based Shift Register IP 核后，将 RAW2RGB 模块添加进工程中，并在顶层例化。由于 RAW2RGB 模块一次只能接收 8 位数据，而 rd_ddr3_fifo 模块每次

输出 16 位数据，因此我们需要在添加一个标志信号，用来控制 RAW2RGB 模块依次接收这 16 位数据的高 8 位与低 8 位，相关代码以及 RAW2RGB 模块的例化如下：

```
//RAW2RGB
wire      DataReq;
wire [7:0] RAW_Data;
reg       flag;
wire      DinReq;
wire [7:0] RED;
wire [7:0] GREEN;
wire [7:0] BLUE;

always@(posedge disp_clk)
if(DinReq)
    flag <= ~flag;
else
    flag <= 0;

assign RAW_Data = (~flag)?rdfifo_dout[15:8]:rdfifo_dout[7:0];

RAW2RGB RAW2RGB(
    .Clk(disp_clk),
    .Rst_n(!reset),
    .DinReq(DinReq),
    .RAW_Data(RAW_Data),
    .Xaddr(H_Addr[0]),
    .Yaddr(V_Addr[0]),
    .RED(RED),
    .GREEN(GREEN),
    .BLUE(BLUE),
    .DoutReq(DataReq)
);
```

其中像素数据地址的奇偶坐标由显示驱动模块（disp_driver）产生。

1.5.4 产生读请求信号

完成了 RAW2RGB 模块的添加和例化后，我们还需要产生一个读请求信号，用于通过 rd_ddr3_fifo 从 ddr3 中读取存储的 RAW 图像数据。RAW2RGB 模块每次会读取 8 位数据，rd_ddr3_fifo 每次传输 16 位数据，因此可以将 flag 信号与 DinReq 信号相与，产生对 rd_ddr3_fifo 的请求信号如下：

```
assign rdfifo_rden = flag & DinReq;
```

1.5.5 修改 disp_driver 配置参数

在 RAW2RGB 成功读取数据并转换为 RGB888 后，数据会送到 disp_driver 模块，disp_driver 模块会根据设定好的时序参数产生相关时序信号。为了方便使用，我们将常用的参数以条件编译的方式预先存放在 disp_parameter_cfg 中。而设计中此时使用的是 1280*720 分辨率下 RGB565 的时序参数，我们需要将其修改为我们所需要的 800*480 分辨率下 RGB888 的时序参数。这里我们将其中 49~83 行附近代码修改为如下：

```
//使用 4.3 寸 480*272 分辨率显示屏
//`define HW_TFT480x272

//使用 5 寸 800*480 分辨率显示屏
//`define HW_TFT800x480

//使用 VGA 显示器，默认为 640*480 分辨率，24 位模式，其他分辨率或需 16 位模式
//可在代码 63 行至 75 行进行重配置
`define HW_VGA

//=====
//以下宏定义选择用于根据显示设备进行位模式和分辨率 2 个参数的设置
//=====
`ifndef HW_TFT480x272 //使用分辨率 480*272 显示屏
    `define MODE_RGB565
    `define Resolution_480x272 1 //时钟为 9MHz

`elsif HW_TFT800x480 //使用分辨率 800*480 显示屏
    `define MODE_RGB565
    `define Resolution_800x480 1 //时钟为 33MHz

`elsif HW_VGA //使用 VGA 显示器，默认为 640*480 分辨率，24 位模式
//=====
//可选择其他分辨率和 16 位模式，需用户根据实际需求设置
//代码 70~71 行设置位模式
//代码 73~78 行设置分辨率
//=====
    `ifndef MODE_RGB565
        `define MODE_RGB888
    // `define Resolution_640x480 1 //时钟为 25.175MHz
    `define Resolution_800x480 1 //时钟为 33MHz
    //`define Resolution_800x600 1 //时钟为 40MHz
    //`define Resolution_1024x600 1 //时钟为 51MHz
    //`define Resolution_1024x768 1 //时钟为 65MHz
    // `define Resolution_1280x720 1 //时钟为 74.25MHz
    //`define Resolution_1920x1080 1 //时钟为 148.5MHz
```

```
`endif
```

这里将输出图像格式设置为了 RGB888，将分辨率设置为了 800*480。而根据条件编译的结果，在 131~144 行附近，可以看到此时输出图像的时序参数如下：

```
`elsif Resolution_800x480
`define H_Total_Time      12'd1056
`define H_Right_Border    12'd0
`define H_Front_Porch     12'd40
`define H_Sync_Time       12'd128
`define H_Back_Porch      12'd88
`define H_Left_Border     12'd0

`define V_Total_Time      12'd525
`define V_Bottom_Border  12'd8
`define V_Front_Porch     12'd2
`define V_Sync_Time       12'd2
`define V_Back_Porch      12'd25
`define V_Top_Border      12'd8
```

1.5.6 修改显示驱动时钟

本次设计使用的 800*480 分辨率，显示屏所需的驱动时钟为 33MHz。而 io_hdmi 想要正常工作还需要一个五倍的时钟，因此修改 dvi_pll 的输出时钟如下：

Clocking Wizard (6.0)

IP Symbol

Resource

Show disabled ports

resetn

clk_in1

clk_out1

clk_out2

locked

Component Name

dvi_pll

Clocking Options

Output Clocks

Port Renaming

MMCM Settings

Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)	Requested	Actual	Phase (degrees)	Requested	Actual	Duty Cycle (%)	Requested	Actual	Drives
☒ clk_out1	clk_out1	33	33.000	33.000	0.000	0.000	50.000	50.0	50.000	50.0	BUFG
☒ clk_out2	clk_out2	165	165.000	165.000	0.000	0.000	50.000	50.0	50.000	50.0	BUFG
☐ clk_out3	clk_out3	100.000	N/A	0.000	0.000	N/A	50.000	N/A	50.000	N/A	BUFG
☐ clk_out4	clk_out4	100.000	N/A	0.000	0.000	N/A	50.000	N/A	50.000	N/A	BUFG
☐ clk_out5	clk_out5	100.000	N/A	0.000	0.000	N/A	50.000	N/A	50.000	N/A	BUFG
☐ clk_out6	clk_out6	100.000	N/A	0.000	0.000	N/A	50.000	N/A	50.000	N/A	BUFG
☐ clk_out7	clk_out7	100.000	N/A	0.000	0.000	N/A	50.000	N/A	50.000	N/A	BUFG

☐ USE CLOCK SEQUENCING

Output Clock

Sequence Number

clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Clocking Feedback

Source

Signaling

☒ Automatic Control On-Chip

☐ Automatic Control Off-Chip

☐ User-Controlled On-Chip

☐ User-Controlled Off-Chip

☒ Single-ended

☐ Differential

Enable Optional Inputs / Outputs for MMCM/PLL

Reset Type

OK

Cancel

图 1-22 修改 dvi_pll 输出时钟

修改完 IP 核后，对应的例化如下：

```
p1l p1l
(
    // Clock out ports
    .clk_out1 (loc_clk200m      ), // output clk_out1
    .clk_out2 (loc_clk100m      ), // output clk_out2
    .clk_out3 (loc_clk24m       ), // output clk_out3
    // Status and control signals
    .resetn   (pl_reset_n      ), // input reset
    .locked    (pll_locked      ), // output locked
    // Clock in ports
    .clk_in1   (ps2pl_clk50m_0 ) // input clk_in1
);

dvi_pll dvi_pll
(
    // Clock out ports
    .clk_out1(disp_clk),      // output clk_out1
    .clk_out2(disp_clk5x),    // output clk_out2
    // Status and control signals
    .resetn(!reset), // input resetn
    .locked(),        // output locked
    // Clock in ports
    .clk_in1(loc_clk100m)
);
```

1.5.7 修改顶层模块连接

完成以上操作之后，我们还需要修改 disp_driver 模块与 TFT 屏接口以及 dvi_encoder 模块的连接。由于设计结构的变化，disp_driver 模块不再直接从 rd_ddr3_fifo 读取数据，而是通过 DataReq 信号来获取 RAW2RGB 模块转化好的 RGB888 格式数据。并结合设定好的时序参数，产生时序信号。数据按照各个颜色分量从 disp_driver 模块输出，同时交由 TFT 屏和 dvi_encoder 模块。考虑到 TFT 屏使用的 RGB565 格式数据，因此需要舍弃对应颜色分量的低位，以满足信号格式需求。同时，为了方便区分信号，我们需要将 VGA 相关信号修该成 TFT 相关信号。因此，disp_driver 模块和 dvi_encoder 模块修改后的顶层例化如下：

```
//tft
wire      frame_begin;
wire      disp_data_req;
wire [15:0] disp_data;
```

```
wire [11:0] H_Addr;
wire [11:0] V_Addr;
wire [7:0] Disp_Red;
wire [7:0] Disp_Green;
wire [7:0] Disp_Blue;

disp_driver disp_driver
(
    .ClkDisp      (disp_clk),
    .Rst_p        (g_rst_p      ),

    .Data         ({RED, GREEN, BLUE} ),
    .DataReq      (DataReq      ),

    .H_Addr       (H_Addr       ),
    .V_Addr       (V_Addr       ),

    .Disp_HS      (TFT_hs       ),
    .Disp_VS      (TFT_vs       ),
    .Disp_Red     (Disp_Red ),
    .Disp_Green   (Disp_Green ),
    .Disp_Blue    (Disp_Blue ),
    .Frame_Begin  (frame_begin ),
    .Disp_DE      (TFT_de       ),
    .Disp_PCLK    (dvi_clk)
);

assign TFT_clk = disp_clk;
assign TFT_rgb = {Disp_Red[7:3], Disp_Green[7:2], Disp_Blue[7:3]};
assign TFT_pwm = 1'b1;

//HDMI
dvi_encoder dvi_encoder1(
    .pixelclk      (disp_clk ),
    .pixelclk5x    (disp_clk5x ),
    .rst_p         (reset      ),
    .blue_din      (Disp_Blue  ),
    .green_din     (Disp_Green ),
    .red_din       (Disp_Red   ),
    .hsync         (TFT_hs     ),
    .vsync         (TFT_vs     ),
    .de            (TFT_de     ),
    .tmbs_clk_p    (hdmi_clk_p ),
    .tmbs_clk_n    (hdmi_clk_n ),
    .tmbs_data_p   (hdmi_dat_p ),
    .tmbs_data_n   (hdmi_dat_n )
);
```

最后，在顶层添加一个输出信号，用于驱动 TFT 的背光显示，该信号高电平有效，代码如下：

```
//TFT Interface
output          TFT_pwm          , //TFT 背光控制

assign TFT_pwm = 1'b1;
```

至此便完成了整个设计代码层面的修改，对设计进行引脚约束后便可以开始生成 bit 流了。

1.5.8 引脚约束

本次设计所涉及的主要是 TFT 屏相关引脚，虽然 disp_driver 模块能够驱动 VGA 接口和 TFT 屏，但是由于二者在开发板上所使用引脚信号不同，我们需要去除掉所有 VGA 的引脚分配并为 TFT 屏分配引脚，本次设计 TFT 屏的引脚分配表如表 1-3 所示：

表 1-3 TFT 屏引脚约束表

Pin Name	Signal Name	Pin NO.		Pin Name	Signal Name	Pin NO.
LCD_HSYNC	TFT_hs	U14		LCD_G6	TFT_rgb[9]	U17
LCD_VSYNC	TFT_vs	W14		LCD_G5	TFT_rgb[8]	V18
LCD_CLK	TFT_clk	U15		LCD_G4	TFT_rgb[7]	T16
LCD_DE	TFT_de	W15		LCD_G3	TFT_rgb[6]	R16
LCD_BL	TFT_pwm	R17		LCD_G2	TFT_rgb[5]	U19
LCD_R7	TFT_rgb[15]	W20		LCD_B7	TFT_rgb[4]	Y19
LCD_R6	TFT_rgb[14]	W19		LCD_B6	TFT_rgb[3]	W18
LCD_R5	TFT_rgb[13]	V17		LCD_B5	TFT_rgb[2]	Y18
LCD_R4	TFT_rgb[12]	V16		LCD_B4	TFT_rgb[1]	W16
LCD_R3	TFT_rgb[11]	T15		LCD_B3	TFT_rgb[0]	Y17
LCD_G7	TFT_rgb[10]	V20				

分配完成后，便可以开始生成 bit 流。等待 bit 生成完成，将包含 bit 的硬件平台导出到 SDK 中，接下来便可以连接硬件，准备开始烧录验证了。

1.6 板级验证

本节将进行 ACZ702 开发板上使用 OV5640 摄像头和 5 寸 TFT 屏以及 HDMI 显示器进行“基于 OV5640 的实时 RAW2RGB 采集显示系统”的验证。

1.6.1 系统所需硬件

1. ACZ702 开发板 x1

2. OV5640 摄像头 x1
3. ACM_VGAHDMI 模块 x1
4. type-c 线缆 x1
5. 5 寸 TFT 屏 x1
6. 软排线 x1
7. HDMI 显示器 x1
8. dc 电源线 x1（可选）
9. HDMI 线缆 x1

1.6.2 硬件连接

将 OV5640、5 寸 TFT 屏、typ-c 线、ACM_VGAHDMI 模块依次连接开发板，将 HDMI 缆线一端连接到 ACM_VGAHDMI 模块的 HDMI 接口上，另一端连接 HDMI 显示器，如图 1-23 所示：



图 1-23 硬件连接图 1

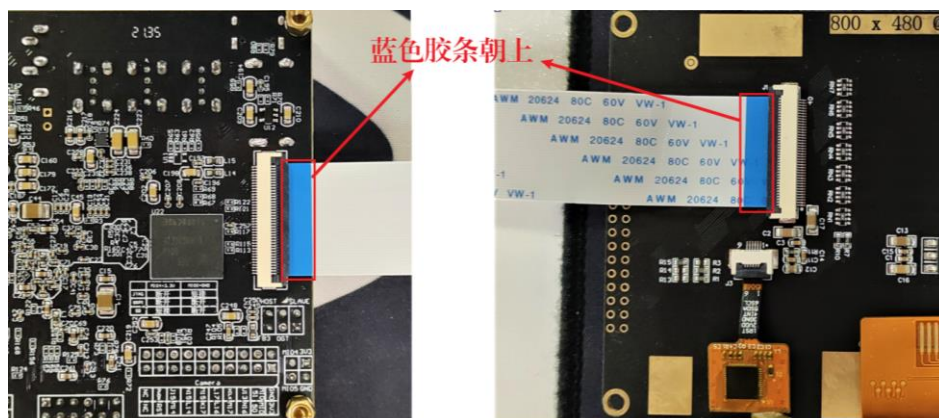


图 1-24 硬件连接图 2

在连接 HDMI 接口时，需要注意：**HDMI 线缆必须一端连接 ACM_VGAHDMI 的 HDMI 接口，另一端连接显示器或电视机的 HDMI 接口。**

店铺：<https://xiaomeige.taobao.com>

技术博客：<http://www.cnblogs.com/xiaomeige/>

官方网站：www.corecourse.cn

技术群组：

连接完毕后拨动开关为开发板上电，通过 SDK 将设计烧录到开发板中。系统开始工作，实物场景、显示效果以及整体场景分别如图 1-25、图 1-26 和图 1-27 所示：



图 1-25 实物场景



图 1-26 HDMI 显示效果



图 1-27 整体场景

可以看到图像显示正常，没有出现任何撕裂重叠现象，颜色层次清晰，说明设计成功。

1.7 常见问题说明

对于本节工程需要注意的有以下几点：

1. 数据是移位写入寄存器的，而在像素矩阵中是从左到右依次写入的，因而其左右顺序与寄存器中是相反的。
2. 摄像头默认输出为 RGB565 数据格式的数据，所以读者在进行板级验证前需要找到摄像头的 `init_table` 表，将 `rom[56]` 与 `rom[57]` 寄存器的值分别改为 `24'h4300_00` 和 `24'h501f_03` 即可，这两个寄存器控制摄像头的输出格式，详细可以参考 OV5640 的数据手册。
3. 实验时，HDMI 线的另一端一定是接独立的 HDMI 接口的显示器，不是接用户自己笔记本电脑或台式电脑主机的 HDMI 接口如果用户手边没

有独立的 HDMI 接口的显示器，则可以在开发板配套的 TFT 显示屏上观察相应的图像内容。

1.8 总结与思考

本节设计实现了从 RAW 数据到 RGB 格式数据的转换，通过调用一个移位寄存器，实现了对相邻行图像数据的提取，本次设计的重点是对数据的定位，以及相应排列顺序时不同的计算方式。建议读者能够跟随本实验内容，完整的进行整个实验。