5.9. Bayer 阵列 RGB888 恢复算法的 HDL-VIP 实现

详见 Crazy_FPGA_Examples 下“18_CMOS_OV7725_RAW8_RGB888”工程。

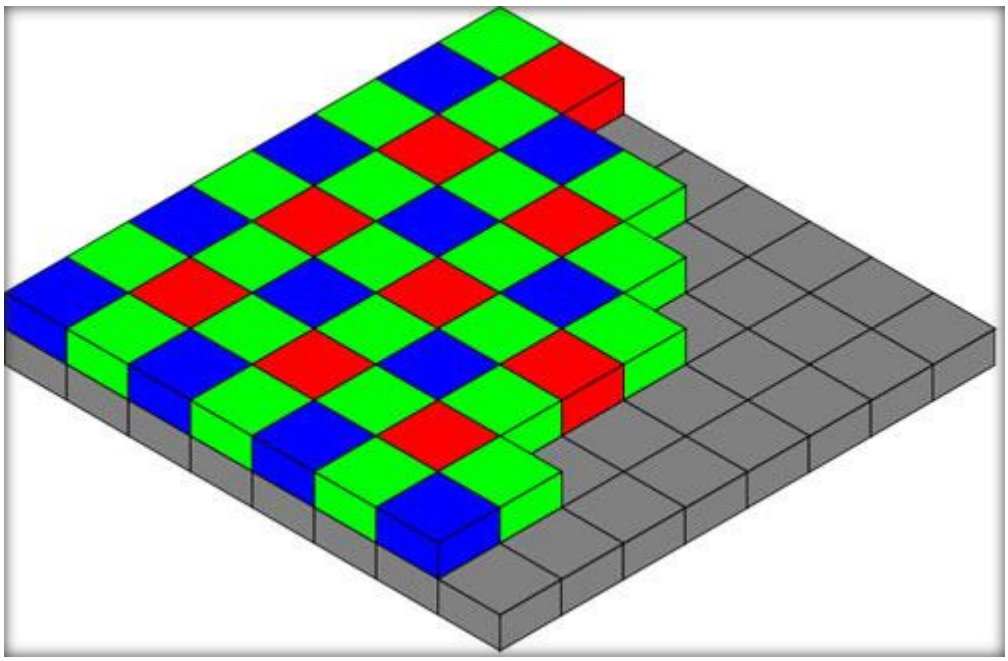
5.9.1. 写在前面的话

经过了前面几节的内容，我们已经完成了 RGB565 格式视频的采集，YCbCr422→YCbCr444→RGB888 视频格式的转换，RGB565→RGB888→YCbCr444 视频格式的转换。同时，我们已经完成了多种滤波算法、Soble 边缘检测算法、腐蚀、膨胀运算等算法。对于视频图像算法的处理而言，如果读者已经读到此处，并且完全掌握了以上内容，那恭喜你，你已经进入了视频图像算法处里的 HDL-VIP 实现之门，后续的路，希望你能自己走。

不过我们依然存在着一个空缺，前面我们处理的都是 OV7725 DSP 后的视频流数据，包括 RGB565、YCbCr422，同时我们的算法也都是基于灰度实现的。倘若我们需要进行彩色图像的算法实现，即便前面的 RGB565→RGB888，或者 YCbCr422 尖头 RGB888 已经能够得到 24Bit 的彩色图像，但由于 CMOS Sensor 原始为 Bayer 阵列，不可避免地在内部 DSP 处理中，插值算法或者其他处理，降低了图像的质量。

因此，本节中，我们将直接捕获 CMOS 输出的 8Bit Bayer 阵列的 RAW 数据，通过 RAW 转 RGB888 算法，从硬件中实现功能，以便于后续的彩色图像处理的硬件加速。

5.9.2. Bayer 阵列的介绍



Bryce Bayer 所发明的拜耳阵列被广泛运用于现代数码相机、摄像机和手机摄像头中。该阵列于1976年注册专利，是实现 CCD 或 CMOS 传感器拍摄彩色图像的主要技术之一。

据国外媒体报道，伊士曼 柯达公司科学家，现代彩色数码摄影拜耳阵列发明人 Bryce Bayer，于2012-11月13日在美国缅因州 Bath 逝世，享年83岁。

拜耳阵列模拟人眼对色彩的敏感程度，采用1红2绿1蓝的排列方式将灰度信息转换成彩色信息。采用这种技术的传感器实际每个像素仅有一种颜色信息，需要利用反马赛克算法进

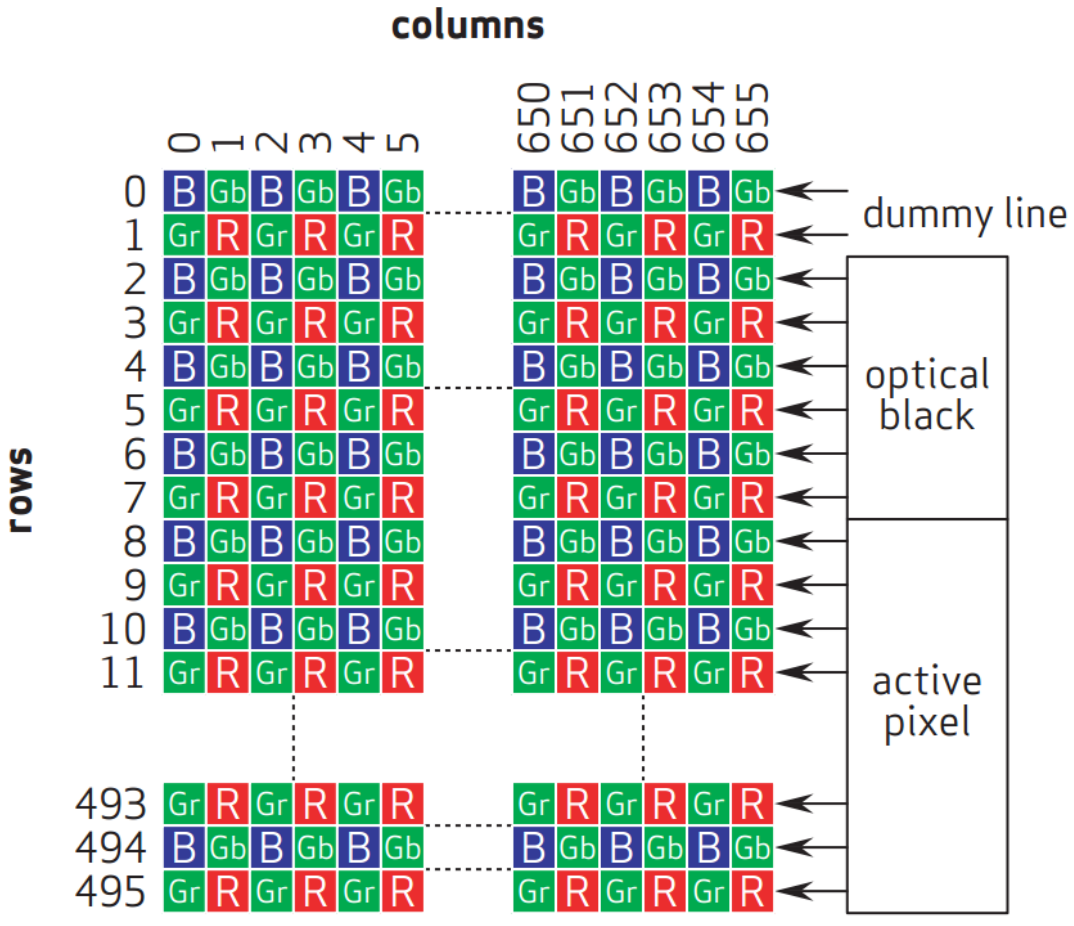
行插值计算，最终获得一张图像。近年已有一些公司提出了新的传感器排列方式，如适马公司的 Foveon X3或富士公司的 X-Trans 传感器，但在目前市场上拜耳阵列传感器仍然占有绝对的统治地位。

除了在彩色影像方面的贡献外，Bayer 还开发很多在数码影像的存储、增强和打印方面的关键算法。Bryce Bayer 长期供职于柯达公司，并于90年代从柯达退休。他的工作和发明为人类带来的巨大的利益。我们向他在数码影像方面所做出的杰出贡献深表敬意，向他的离去表达诚挚哀悼。

5.9.3. OV7725 Bayer 阵列介绍

输出原始的 Bayer 阵列的图像数据，即为 RAW 数据。原始的 Bayer 阵列，具有 10Bit 的深度，但一般我们只用高 8 位，即 RAW 8Bit。关于 OV7725 RAW 格式输出的配置，Bingo 在 6.6 节中已经详细的介绍过，在此，只就事论事，OV7725 的 Bayer 阵列，及 Bayer 阵列的恢复方法。

OV7725 的确能输出 RAW 原始数据，但其手册蛋疼之处，在于竟然没有给出 OV7725 Sensor Bayer 阵列的排布方式，Bingo 在 OV7740 中找到 Bayer 阵列的排列方式，如下所示：



从图中可知，默认输出的 RAW 格式的视频流，奇数行与偶数行的像元即便不一样，但均有一定的规律，如下：

（1） 奇数行：line_cnt == 0、2、4、6、…478

a) 奇数点：pixel_cnt == 0、2、4、6、…638

除了第一个和最后一个像素需要镜像 3X3，其他像素均如此，B 在中心，R 与 G 分布在四个角落。

b) 偶数点：pixel_cnt == 1、3、5、7、…639

除了第一个和最后一个像素需要镜像实现 3X3，其他像素均如此，G 在中心，上下 R，左右为 B。

（2） 偶数行：line_cnt == 1、3、5、7、…499

a) 奇数点：pixel_cnt == 0、2、4、6、…638

除了第一个和最后一个像素需要镜像 3X3，其他像素均如此，G 在中心，上下为 B，左右为 R。

b) 偶数点：pixel_cnt == 1、3、5、7、…639

除了第一个和最后一个像素需要镜像 3X3，其他像素均如此，R 在中心，B 与 G 分布在四个角落。

因此，通过分析，如果我们能够得到完整的 3X3 的像素阵列，那实现 3*3 的 Bayer 阵列的恢复，显而易见的好简单有木有？

5.9.4. 简易 3*3 像素阵列的 HDL 实现

Bingo 当年作者方便设计的时候，查询过很多资料，3*3 阵列的获取，大概有以下三种方式：

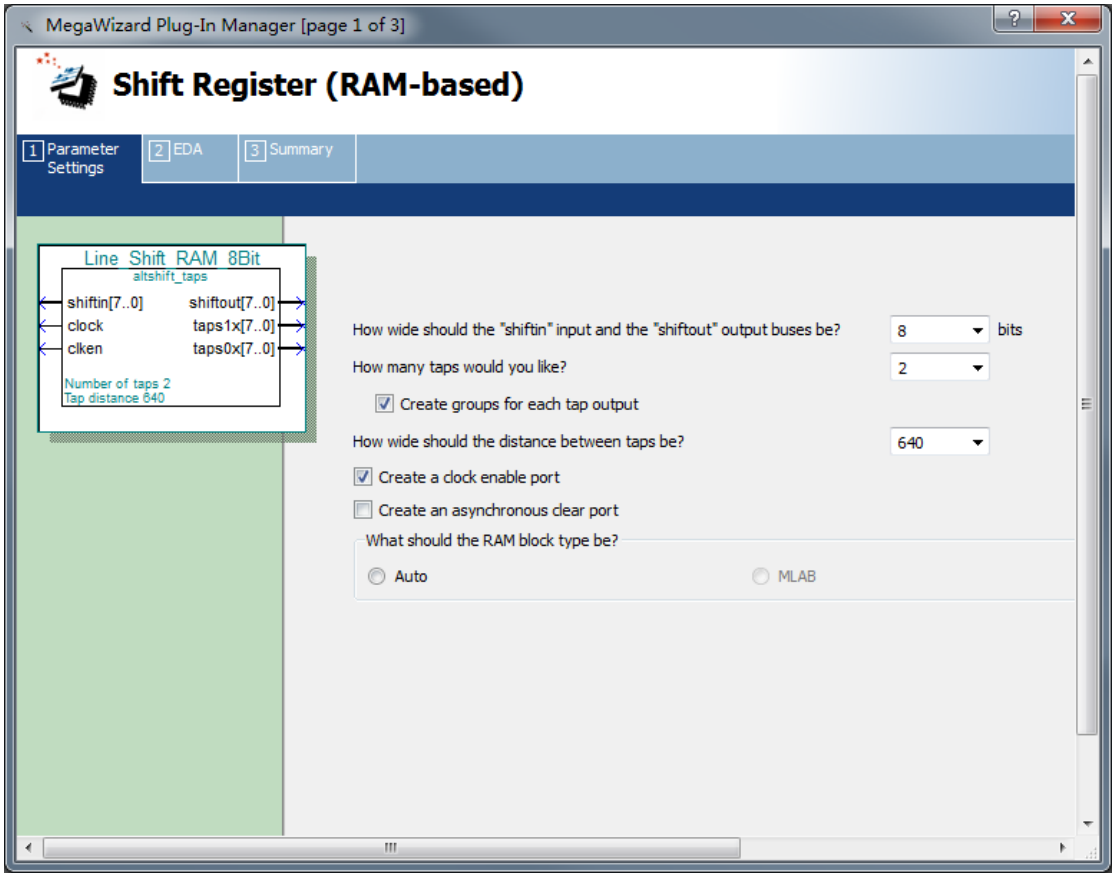
（4） 通过 2 个或 3 个 RAM 的存储，来实现 3*3 像素阵列

（5） 通过 2 个或 3 个 FIFO 的存储，来实现 3*3 像素阵列

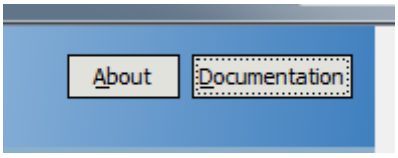
（6） 通过 2 行或 3 行 Shift_RAM 的移位存储，来实现 3*3 像素阵列

不过 Bingo 的经验告诉大家，最方便的实现方式，非 Shift_RAM 莫属了，都感觉 Shift_RAM 甚至是为实现 3*3 阵列而生的！

首先介绍一下 Shift_RAM，宏定义模块如下图所示：

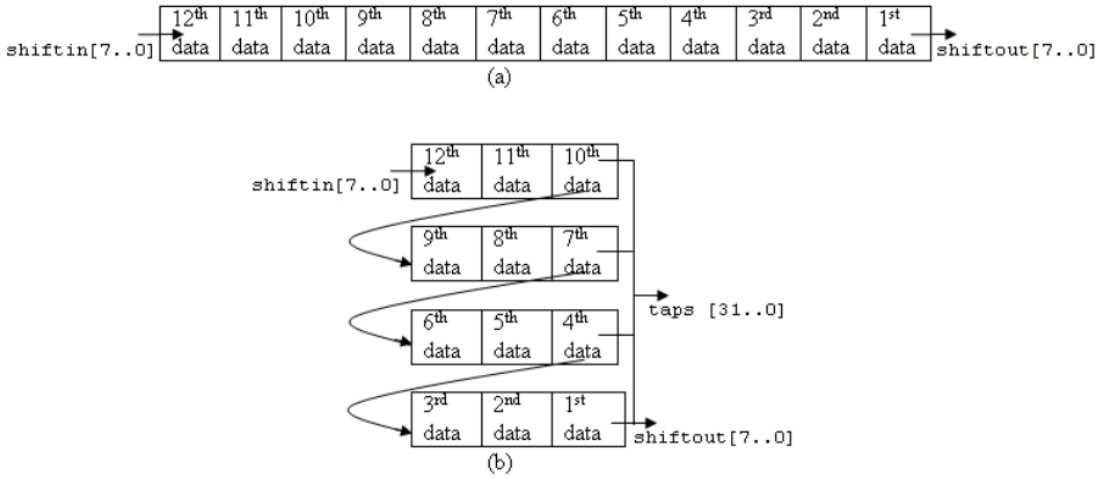


Shift_RAM 可定义数据宽度、移位的行数、每行的深度。这里我们固然需要 8Bit，640 个数据每行，同时移位寄存 2 行即可（原因看后边）。同时选择时钟使能端口 clken。详细的关于 Shift_RAM 的手册参数，可在宏定义窗口右上角 Document 中查看，如下：



http://www.altera.com/literature/ug/ug_shift_register_ram_based.pdf

手册给出了一个非常形象的一位寄存示意图，如下所示：



看懂这个示意图，没有任何继续往下看手册的意义!!! 直接上战场!!! Bingo 的思路是这样子的，Shift_RAM 中存 2 行数据，同时与当前输入行的数据，组成 3 行的阵列。这里新建 VIP_Matrix_Generate_3X3_8Bit 文件，在此实现 8Bit 宽度的 3*3 像素阵列功能。具体的实现步骤如下：

（6） 首先，将输入的信号用像素使能时钟同步一拍，以保证数据与 Shift_RAM 输出的数据保持同步，如下：

```
//sync row3_data with per_frame_clken & row1_data & raw2_data
wire  [7:0]   row1_data;  //frame data of the 1th row
wire  [7:0]   row2_data;  //frame data of the 2th row
reg       [7:0]   row3_data;  //frame data of the 3th row
always@(posedge clk or negedge rst_n)
begin
  if(!rst_n)
      row3_data <= 0;
  else
      begin
      if(per_frame_clken)
          row3_data <= per_img_Y;
      else
          row3_data <= row3_data;
      end
end
```

（7） 接着，例化并输入 row3_data，此时可从 Modelsim 中观察到 3 行数据同时存在了，HDL 如下：

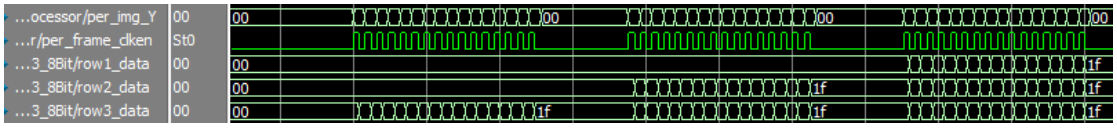
```
//--------------------------------------
//module of shift ram for raw data
wire  shift_clk_en = per_frame_clken;
Line_Shift_RAM_8Bit
#(
  .RAM_Length (IMG_HDISP)
```

```
)
u_Line_Shift_RAM_8Bit
(
  .clock      (clk),
  .clken      (shift_clk_en), //pixel enable clock
//    .aclr       (1'b0),

  .shiftin    (row3_data),    //Current data input
  .taps0x     (row2_data),    //Last row data
  .taps1x     (row1_data),    //Up a row data
  .shiftout   ()
);
```

这里需要注意的是，本模块使用的 Shift_RAM 不仅仅通过 LPM 得到，Bingo 对此作了手脚，即每行的深度通过宏定义输入，这样方便了仿真，以及未来不同分辨率之间图像的移植。打开 Shift_RAM 文件可见 Bingo 做的手脚，所以建议大家不要乱动！

```
module Line_Shift_RAM_8Bit
#(
    parameter    [9:0]   RAM_Length = 10'd640     //640*480
)
......
        ALTSHIFT_TAPS_component.tap_distance = RAM_Length,
        ALTSHIFT_TAPS_component.width = 8;
```

在经过 Shift_RAMd 移位存储后，我们得到的 row0_data，row1_data，row2_data 的仿真示意图如下所示：



数据从 row3_data 输入，满 3 行后刚好唯一 3 行阵列的第一。从图像第三行输入开始，到图像的最后一行，我们均可从 row_data 得到完整的 3 行数据，基为实现 3*3 阵列奠定了基础。不过这样做有 2 个不足之处，即第一行与第二行不能得到完整的 3*3 阵列。但从主到次，且不管算法的完美型，我们先验证 3X3 模板实现的正确性。因此直接在行有效期间读取 3*3 阵列，机器方便快捷的实现了我们的目的。

（8） Row_data 读取信号的分析及生成

这里涉及到了一个问题，数据从 Shift_RAM 存储耗费了一个时钟，因此 3*3 阵列的读取使能与时钟，需要进行一个 clock 的偏移，如下所示：

```
//----------------------------------------
```

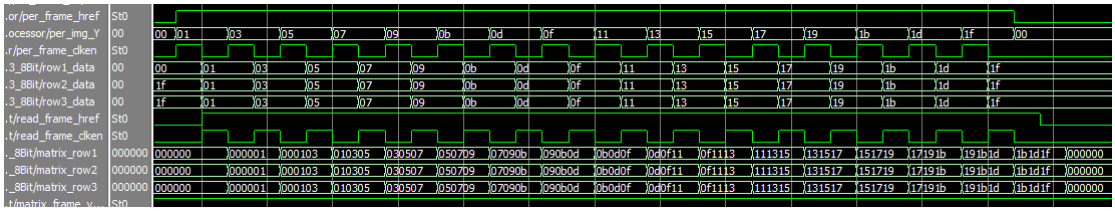
```
//lag 2 clocks signal sync
reg   [1:0]   per_frame_vsync_r;
reg   [1:0]   per_frame_href_r;
reg   [1:0]   per_frame_clken_r;
always@(posedge clk or negedge rst_n)
begin
  if(!rst_n)
      begin
      per_frame_vsync_r <= 0;
      per_frame_href_r <= 0;
      per_frame_clken_r <= 0;
      end
  else
      begin
      per_frame_vsync_r   <=  {per_frame_vsync_r[0], per_frame_vsync};
      per_frame_href_r    <=  {per_frame_href_r[0],  per_frame_href};
      per_frame_clken_r   <=  {per_frame_clken_r[0], per_frame_clken};
      end
end
//Give up the 1th and 2th row edge data caculate for simple process
//Give up the 1th and 2th point of 1 line for simple process
wire  read_frame_href     =   per_frame_href_r[0];    //RAM  read  href  sync
signal
wire  read_frame_clken    =   per_frame_clken_r[0];   //RAM read enable
```

（9） Okay，此时根据 read_frame_href 与 read_frame_clken 信号，直接读取 3*3 像素阵列。读取的 HDL 实现如下（**请读者详细思考这一段代码**）：

```
//------------------------------------------------
/*********************************************
  (1) Read data from Shift_RAM
  (2) Caculate the Sobel
  (3) Steady data after Sobel generate
*********************************************/
//wire    [23:0]  matrix_row1 = {matrix_p11, matrix_p12, matrix_p13};//Just
for test
//wire    [23:0]  matrix_row2 = {matrix_p21, matrix_p22, matrix_p23};
//wire    [23:0]  matrix_row3 = {matrix_p31, matrix_p32, matrix_p33};
always@(posedge clk or negedge rst_n)
begin
  if(!rst_n)
      begin
      {matrix_p11, matrix_p12, matrix_p13} <= 24'h0;
      {matrix_p21, matrix_p22, matrix_p23} <= 24'h0;
      {matrix_p31, matrix_p32, matrix_p33} <= 24'h0;
      end
  else if(read_frame_href)
      begin
      if(read_frame_clken)    //Shift_RAM data read clock enable
          begin
          {matrix_p11, matrix_p12, matrix_p13} <= {matrix_p12, matrix_p13,
row1_data};  //1th shift input
```

```
          {matrix_p21, matrix_p22, matrix_p23} <= {matrix_p22, matrix_p23,
row2_data};  //2th shift input
          {matrix_p31, matrix_p32, matrix_p33} <= {matrix_p32, matrix_p33,
row3_data};  //3th shift input
          end
      else
          begin
          {matrix_p11, matrix_p12, matrix_p13} <= {matrix_p11, matrix_p12,
matrix_p13};
          {matrix_p21, matrix_p22, matrix_p23} <= {matrix_p21, matrix_p22,
matrix_p23};
          {matrix_p31, matrix_p32, matrix_p33} <= {matrix_p31, matrix_p32,
matrix_p33};
          end
      end
  else
      begin
      {matrix_p11, matrix_p12, matrix_p13} <= 24'h0;
      {matrix_p21, matrix_p22, matrix_p23} <= 24'h0;
      {matrix_p31, matrix_p32, matrix_p33} <= 24'h0;
      end
end
```

最后得到的 matrix_p11、p12、p13、p21、p22、p23、p31、p32、p33 即为得到的 3*3 像素阵列，仿真时序图如下所示：

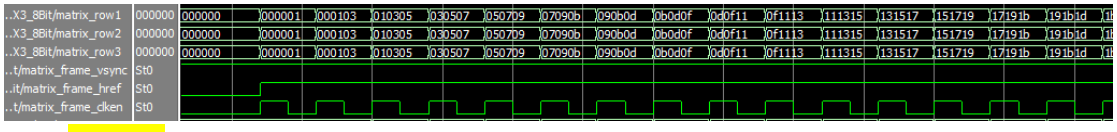


（10） 完整的 Bingo 版 VIP 时序

前面 Bingo 说过，所有 Video_Image_Processor 包含的图像处理架构，均用统一的 Bingo 版时序，因此这里 3X3 的像素阵列生成也不例外。前面 Shift_RAM 存储耗费了一个时钟，同时 3*3 阵列的生成耗费了一个时钟，因此我们需要人为的将行场信号、像素使能读取信号移动 2 个时钟，如下所示：

```
assign    matrix_frame_vsync  =   per_frame_vsync_r[1];
assign    matrix_frame_href   =   per_frame_href_r[1];
assign    matrix_frame_clken  =   per_frame_clken_r[1];
```

至此我们得到了完整的 3*3 像素阵列的模块，同时行场、使能时钟信号与 VIP 时序保持一致，Modelsim 仿真图如下所示：

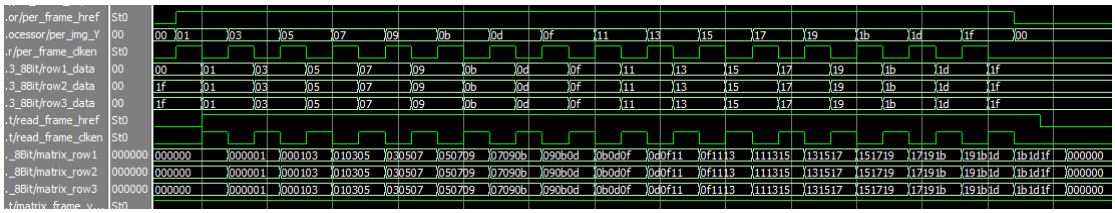


注意：这样做我们快速方便地得到了 3*3 像素阵列，不过计算第一次与第

二次得到的 3*3 行像素不完整，同时在计算中，最后一行的像素没有参与。同时每行的第一、第二、最后个像素也是如此。Bingo 这里给出的只是一个 VIP 算法 3X3 像素阵列生成的模板，更完美的方式可以通过镜像方法实现，希望读者自己加倍努力吧！！！

5.9.5. 行阈割、水平像素镜像的优化版 3X3 实现

在前面，我们无论是进行中值滤波，还是 Sobel 边缘检测算法算法中，我们都是用的到了 3X3 的矩阵，我们已经通过 VIP_Matrix_Generate_3X3_8Bit，简单的得到了没有经过边缘镜像后的阵列。最后得到的 matrix_p11、p12、p13、p21、p22、p23、p31、p32、p33 即为得到的 3*3 像素阵列，仿真时序图如下所示：



从图中第一个像素输出为 24’h000001，实际上应该为 030103，同时最后一个为中心的矩阵没有输出。由于前面的算法，均只需要考虑 3X3 阵列，而不需要考虑像素的奇偶分布坐标，因此认为的忽略边缘，即便最终处理出现了边缘的 Bug，我们都将以实现 VIP 算法为首要目的，而忽略这些细节。但是 Bayer 阵列需要考虑奇偶像素/行的分布，因此，此处考虑优化的可能性，将行、列边缘像素通过镜像实现，最后输出完整的 3X3 的阵列。

此处 Bingo 将介绍通过像素缓存机制实现的行边缘镜像。与前面 YCbCr422 转 YCbCr444 类似，我们首先经过像素的缓存，延迟到第二个像素时，通过与前面像素的组合，来实现镜像。同时，由于像素的迟滞，我们需要人为的将 per_cmos_href 扩展一个像素周期，来实现完整的序列操作。

首先，给出像素的行有效周期使能扩展，read_frame_href 的生成如下所示：

```
//-----------------------------------------
//lag 2 clocks signal sync
reg [2:0]   per_frame_vsync_r;
reg [2:0]   per_frame_href_r;
reg [2:0]   per_frame_clken_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        per_frame_vsync_r <= 0;
        per_frame_href_r <= 0;
        end
    else
        begin
        per_frame_vsync_r   <=  {per_frame_vsync_r[1:0],    per_frame_vsync};
        per_frame_href_r    <=  {per_frame_href_r[1:0],     per_frame_href};
        end
end
//Give up the 1th and 2th row edge data caculate for simple process
//Give up the 1th and 2th point of 1 line for simple process
wire    read_frame_href     =   per_frame_href_r[0]|per_frame_href_r[1];    /
```

其次，通过此信号，以及像素计数电路，生成数据的寄存、迟滞，已实现镜像，关键部分代码如下所示：

```
else if(read_frame_href)
    begin
    pixel_cnt <=  (pixel_cnt < IMG_HDISP) ? pixel_cnt + 1'b1 : 10'd0;   //Point Counter
    {row1_data1, row1_data0} <= {row1_data0, row1_data};
    {row2_data1, row2_data0} <= {row2_data0, row2_data};
    {row3_data1, row3_data0} <= {row3_data0, row3_data};
    if(pixel_cnt == 0)
        begin
        {matrix_p11, matrix_p12, matrix_p13} <= 0;
        {matrix_p21, matrix_p22, matrix_p23} <= 0;
        {matrix_p31, matrix_p32, matrix_p33} <= 0;
        end
    else if(pixel_cnt == 1)         //First point
        begin
        {matrix_p11, matrix_p12, matrix_p13} <= {row1_data, row1_data0, row1_data};
        {matrix_p21, matrix_p22, matrix_p23} <= {row2_data, row2_data0, row2_data};
        {matrix_p31, matrix_p32, matrix_p33} <= {row3_data, row3_data0, row3_data};
        end
    else if(pixel_cnt == IMG_HDISP) //Last Point
        begin
        {matrix_p11, matrix_p12, matrix_p13} <= {row1_data1, row1_data, row1_data1};
        {matrix_p21, matrix_p22, matrix_p23} <= {row2_data1, row2_data, row2_data1};
        {matrix_p31, matrix_p32, matrix_p33} <= {row3_data1, row3_data, row3_data1};
        end
    else                            //2 ~ IMG_HDISP-1 Point
        begin
        {matrix_p11, matrix_p12, matrix_p13} <= {row1_data1, row1_data0, row1_data};
        {matrix_p21, matrix_p22, matrix_p23} <= {row2_data1, row2_data0, row2_data};
        {matrix_p31, matrix_p32, matrix_p33} <= {row3_data1, row3_data0, row3_data};
        end
    end
```

从上述实现流程分析，在数据读取中，持续的寄存上一列的 3 个像素，此外根据像素点来分配输出，如下：

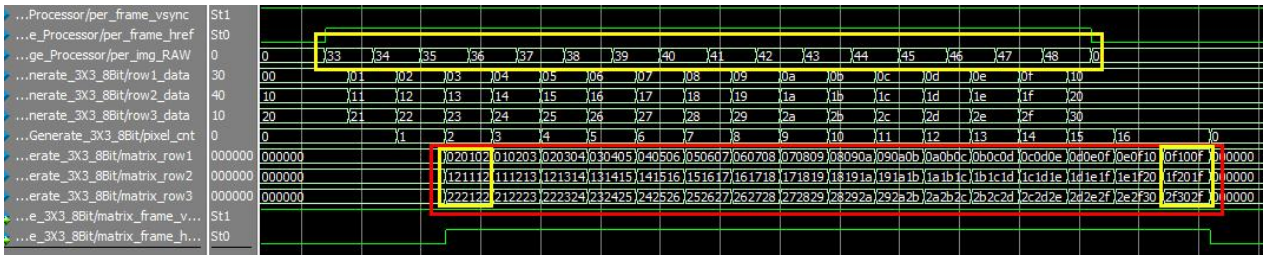
（1） 当 pixel_cnt == 0 时，不输出 3X3 阵列。

（2） 当 pixel_cnt == 1 时，输出镜像后的边缘（第一个）像素 3X3 阵列

（3） 当 pixel_cnt == IMG_HDISP，输出镜像后的边缘（最后一个）像素 3X3 阵列

（4） 在 1~IMG_HDISP 之间，输出当前像素为中心的 3X3 阵列

这部分行边缘镜像的 3X3 的实现，波形如下所示：



从图中可见，第一种 3X3 方案中，第一个 3X3 阵列为 000001，而此时我们已经通过迟滞+镜像法则，第一个像素输出标准的 3X3 阵列；同时最后一个阵列也是如此。这样，我们便实现了水平方向上的像素镜像。

关于竖直方向上的像素镜像，同样需要一行的迟滞，以实现行的镜像。但由于我们只有 480 个行信号，因此再通过 Shift_RAM 之后，还需要认为的生成一个行信号，来补偿镜像输出后的行信号损耗。这一部分的实现，希望读者自己去实现，Bingo 点到为止。

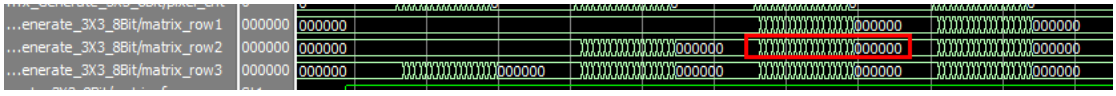
5.9.6. 3X3 Bayer 阵列恢复的 HDL-VIP 实现

由于 Bayer 阵列的恢复，密切关系到奇偶行、即奇偶像素的坐标，因此如果没有处理好这方面的参数，势必会导致 Bayer 阵列恢复的失败。这里通过奇偶行、像素的地址，来实现 Bayer 阵列的恢复算法，如下所示：

```
//---------------------------------------
//Convet RAW 2 RGB888 Format
localparam  OddLINE_OddPOINT    =   2'b00;  //odd lines + odd point
localparam  OddLINE_Even_POINT  =   2'b01;  //odd lines + even point
localparam  EvenLINE_OddPOINT   =   2'b10;  //even lines + odd point
localparam  EvenLINE_EvenPOINT  =   2'b11;  //even lines + even point
reg [9:0]   post_img_red_r;
reg [9:0]   post_img_green_r;
reg [9:0]   post_img_blue_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        post_img_red_r  <=  0;
        post_img_green_r<=  0;
        post_img_blue_r <=  0;
        end
    else
        begin
        case({line_cnt[0], pixel_cnt[0]})
        //-------------------------BGBG...BGBG--------------------//
        OddLINE_OddPOINT:   //odd lines + odd point
            begin   //Center Blue
            post_img_red_r  <=  (matrix_p11 + matrix_p13 + matrix_p31 + matrix_p33)>>2;
            post_img_green_r<=  (matrix_p12 + matrix_p21 + matrix_p23 + matrix_p32)>>2;
            post_img_blue_r <=  matrix_p22;
            end
        OddLINE_Even_POINT: //odd lines + even point
            begin   //Center Green
            post_img_red_r  <=  (matrix_p12 + matrix_p32)>>1;
            post_img_green_r<=  matrix_p22;
            post_img_blue_r <=  (matrix_p21 + matrix_p23)>>1;
            end
        //-------------------------GRGR...GRGR--------------------//
        EvenLINE_OddPOINT:  //even lines + odd point
            begin   //Center Green
            post_img_red_r  <=  (matrix_p21 + matrix_p23)>>1;
            post_img_green_r<=  matrix_p22;
            post_img_blue_r <=  (matrix_p12 + matrix_p32)>>1;
            end
        EvenLINE_EvenPOINT: //even lines + even point
            begin   //Center Red
            post_img_red_r  <=  matrix_p22;
            post_img_green_r<=  (matrix_p12 + matrix_p21 + matrix_p23 + matrix_p32)>>2;
            post_img_blue_r <=  (matrix_p11 + matrix_p13 + matrix_p31 + matrix_p33)>>2;
            end
        endcase
        end
end
```

其中 OddLINE_OddPOINT 为奇数行奇数像素，OddLINE_Even_POINT 为奇数行偶数像素，EvenLINE_OddPOINT 为偶数行，奇数像素，EvenLINE_EvenPOINT 为偶数行，偶数像素。通过 CASE 语句，描述了最简单的 3X3 的 Bayer 阵列恢复。

不过分析前文中，我们优化过的 3X3 阵列生成方案，得到的波形如下所示：



由于我们只进行了行边缘像素的镜像操作，列边缘像素的镜像操作略复杂，这里没有做处理，从图中可知，书粗话的 3X3 阵列，奇数行的中心实际上为原图像的偶数行。

因此我们认为的修改奇、偶行的参数，如下所示：

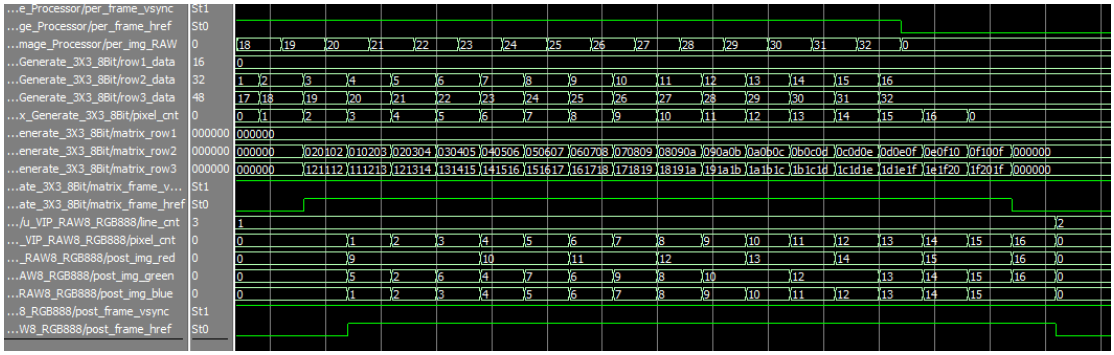
```
//---------------------------------------
//Convet RAW 2 RGB888 Format
localparam  OddLINE_OddPOINT    =   2'b10;//00; //odd lines + odd point
localparam  OddLINE_Even_POINT  =   2'b11;//01; //odd lines + even point
localparam  EvenLINE_OddPOINT   =   2'b00;//10; //even lines + odd point
localparam  EvenLINE_EvenPOINT  =   2'b01;//11; //even lines + even point
```

此时刚好将列对换了，但由于我们的行为完整的镜像操作，因此保持可以直接转换过去。再次我们只需修改 localparam 参数即可。。。这一步非常的重要!!

最后，由于 Bayer 阵列的恢复消耗了 2 个时钟，因为认为的将行场信号偏移 2 个时钟，如下所示：

```
//-----------------------------------------
//lag n clocks signal sync
reg [1:0]   post_frame_vsync_r;
reg [1:0]   post_frame_href_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        post_frame_vsync_r  <=  0;
        post_frame_href_r   <=  0;
        end
    else
        begin
        post_frame_vsync_r  <=  {post_frame_vsync_r[0], matrix_frame_vsync};
        post_frame_href_r   <=  {post_frame_href_r[0],  matrix_frame_href};
        end
end
assign  post_frame_vsync    =   post_frame_vsync_r[0];
assign  post_frame_href     =   post_frame_href_r[0];
```

这样，我们便得到了完整的，Bayer 阵列恢复后的，RGB888 的视频流数据。这部分的 Modelsim 仿真波形，如下所示：



VIP_RAW8_RGB888 的例化如下所示：

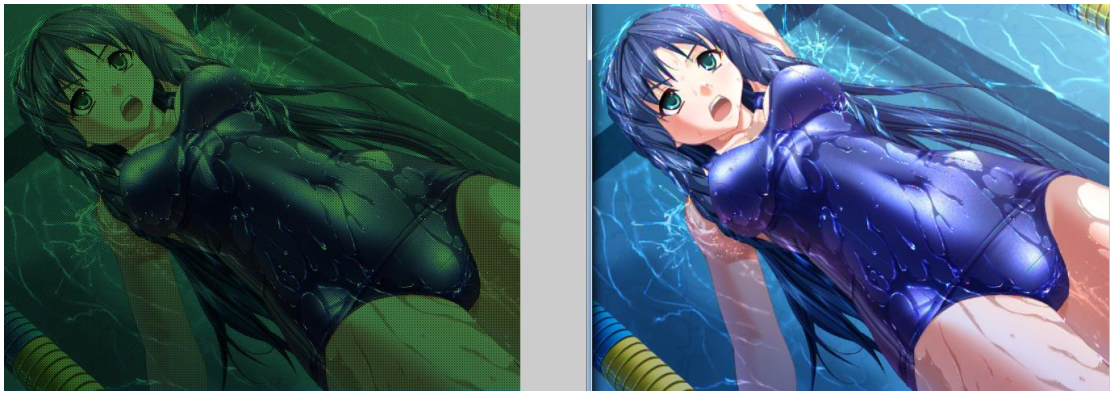
```
//---------------------------------------
//Convert the RAW format to RGB888 format.
VIP_RAW8_RGB888
#(
    .IMG_HDISP   (IMG_HDISP),    //640*480
    .IMG_VDISP   (IMG_VDISP)
)
u_VIP_RAW8_RGB888
(
    //global clock
    .clk                    (clk),                  //cmos video pixel clock
    .rst_n                  (rst_n),                //system reset

    //Image data prepred to be processd
    .per_frame_vsync        (per_frame_vsync),      //Prepared Image data vsync valid signal
    .per_frame_href         (per_frame_href),       //Prepared Image data href vaild  signal
    .per_img_RAW            (per_img_RAW),          //Prepared Image data 8 Bit RAW Data

    //Image data has been processd
    .post_frame_vsync       (post_frame_vsync),     //Processed Image data vsync valid signal
    .post_frame_href        (post_frame_href),      //Processed Image data href vaild  signal
    .post_img_red           (post_img_red),         //Prepared Image green data to be processed
    .post_img_green         (post_img_green),       //Prepared Image green data to be processed
    .post_img_blue          (post_img_blue)         //Prepared Image blue data to be processed
);
```

最后，将 Video_Image_Processor 模块输出的 RGB888，输入至 24Bit 的 SDRAM 控制器。全编译并下载测试，得到恢复后的 RGB 图像。这里 Bingo 通过 Bayer 阵列图像的显示、恢复后的彩色图像显示进行对比分析，如下所示：

（1） Bayer 阵列图像通过灰度显示：

从图中可见，直接将 RAW，通过 R = G = B = RAW 也能显示，近似于灰度的图像，但由于相邻像素的彩色深度不同，会有网格的现象。这种方式用来验证 CMOS Sensor 输出的 Bayer 阵列图像是否正确，是一种有效的方法。

如果想更透彻的研究 Bayer，可以借助 Matlab 来做实验，如下是 Bingo 将通过 RGB 图像转换你的 Bayer 阵列图像，以及原图（仅做参考）：



（2） Bayer 阵列恢复后的 RGB888 图像

通过 3X3 阵列恢复，将得到的 Bayer 阵列恢复成了彩色图像。

（3） 直接输出的 RGB565 图像

上图为 CMOS 直接输出的 RGB565 图像，与（2）中由 Bayer 阵列恢复得到的彩色图像对比，可见恢复后的图像色彩更鲜艳，而 CMOS 直接输出的 RGB565 图像，细节略显模糊。。。这主要是由于 CMOS 内部 Bayer 阵列恢复的局限性。当然强大的 FPGA，是没法比的了。。

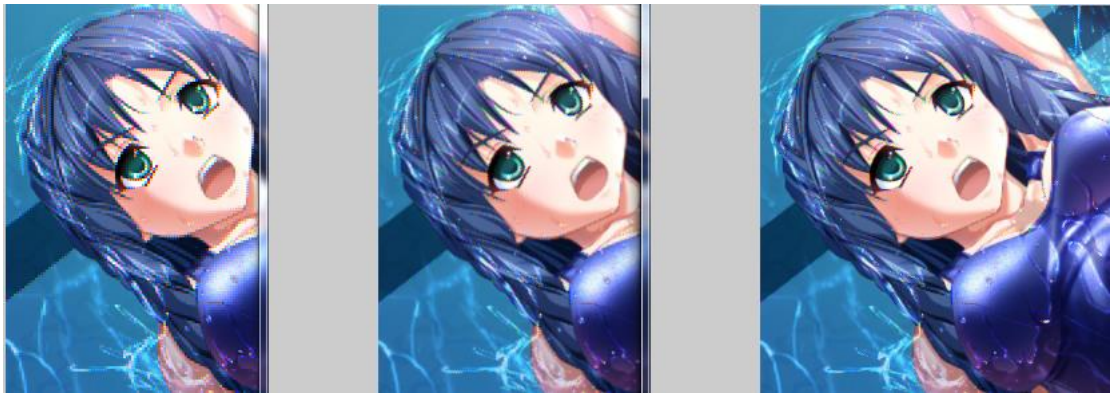
但从算法层面考虑，我们已经实现了基本的 3*3 Bayer 阵列的恢复算法。至于这种算法的优越性，是否有提升的空间，那是自然的。

5.9.7. 3X3 Bayer 阵列恢复算法优化方案

Bayer 阵列的恢复算法，有很多种。最简单的可以通过 2*2，即 2 行通过插值来实现 RGB 阵列的恢复。据说 OV7725 内部就是通过 2 行插值实现的 Bayer 阵列恢复，因此输出的彩色图像，在质量上都欠佳。

Bingo 在前面一节中设计的是 3X3 的 Bayer 阵列恢复，即通过 3 行的值来实现 RGB 阵列的恢复。3X3 阵列恢复的算法，明显比 2X2 恢复后的图像要鲜艳，同时更清晰的细节。但是仔细的观察上述工程中恢复得到的图像，如下所示：

从图中可见，图像的边缘有锯齿，细节不够清晰。那么，使用 3X3 的阵列恢复，是否有还有优化的余地呢？？？答案是必然的。这里 Bingo 给出通过 Matlab 仿真测试的 2X2 恢复、3X3 恢复、3X3 梯度恢复后的三种图像，对比如下所示：



可见从做到右，边缘细节明显增强，画质也更加优越。。3X3 梯度恢复方案，Bingo 也不是凭空想象出来的。当年 Bingo 通过查阅 n 多论文，最后得到了针对 3X3 Bayer 阵列恢复的优化方案。这里 Bingo 给出方案，但希望读者自己去实现，自己动手，丰衣足食。

3X3 梯度 Bayer 阵列优化方案：

原有的 3x3 模板不变，无需使用不同像素的通道信息，考虑 **G 分量的水平**、竖直**梯度**，以及 R/B 的对角线梯度（主要是 G 在四周的情况，即中间为 R 或者 B），进一步保留图像的边缘信息，改善奇数行以及偶数行 3*3 矩阵处理的方案！公式及算法如下所示：

1) 奇数行

a) **奇数列**，B 在中心

$$H_g = \left|\text{Bayer}(i, j-1) - Bayer(i, j+1)\right|, V_g = \left|Bayer(i-1, j) - Bayer(i+1, j)\right|$$

$$D_{r1} = \left|Bayer(i-1, j-1) - Bayer(i+1, j+1)\right|, D_{r2} = \left|Bayer(i-1, j+1) - Bayer(i+1, j-1)\right|$$

$$\begin{cases} R = \begin{cases} (Bayer(i-1, j-1) + Bayer(i+1, j+1))/2, & D_{r1} < D_{r2} \\ (Bayer(i-1, j+1) - Bayer(i+1, j-1))/2, & D_{r1} > D_{r2} \\ (Bayer(i-1, j-1) + Bayer(i+1, j+1) + Bayer(i-1, j+1) - Bayer(i+1, j-1))/4, & D_{r1} = D_{r2} \end{cases} \\ G_{in} = \begin{cases} (\text{Bayer}(i, j-1) + Bayer(i, j+1))/2, & H_g < V_g \\ (Bayer(i-1, j) + Bayer(i+1, j))/2, & H_g > V_g \\ (\text{Bayer}(i, j-1) + Bayer(i, j+1) + Bayer(i-1, j) + Bayer(i+1, j))/4, & H_g = V \end{cases} \\ B = Bayer(i, j) \end{cases}$$

2) 偶数行

a) **偶数列**，R 在中心

$$H_g = \left|\text{Bayer}(i, j-1) - Bayer(i, j+1)\right|, V_g = \left|Bayer(i-1, j) - Bayer(i+1, j)\right|$$

$$D_{b1} = \left|Bayer(i-1, j-1) - Bayer(i+1, j+1)\right|, D_{b2} = \left|Bayer(i-1, j+1) - Bayer(i+1, j-1)\right|$$

$$\begin{cases} R = Bayer(i, j) \\ G = \begin{cases} (\text{Bayer}(i, j-1) + Bayer(i, j+1))/2, & H_g < V_g \\ (Bayer(i-1, j) + Bayer(i+1, j))/2, & H_g > V_g \\ (\text{Bayer}(i, j-1) + Bayer(i, j+1) + Bayer(i-1, j) + Bayer(i+1, j))/4, & H_g = V \end{cases} \\ B = \begin{cases} (Bayer(i-1, j-1) + Bayer(i+1, j+1))/2, & D_{b1} < D_{b2} \\ (Bayer(i-1, j+1) - Bayer(i+1, j-1))/2, & D_{b1} > D_{b2} \\ (Bayer(i-1, j-1) + Bayer(i+1, j+1) + Bayer(i-1, j+1) - Bayer(i+1, j-1))/4, & D_{b1} = D_{b2} \end{cases} \end{cases}$$

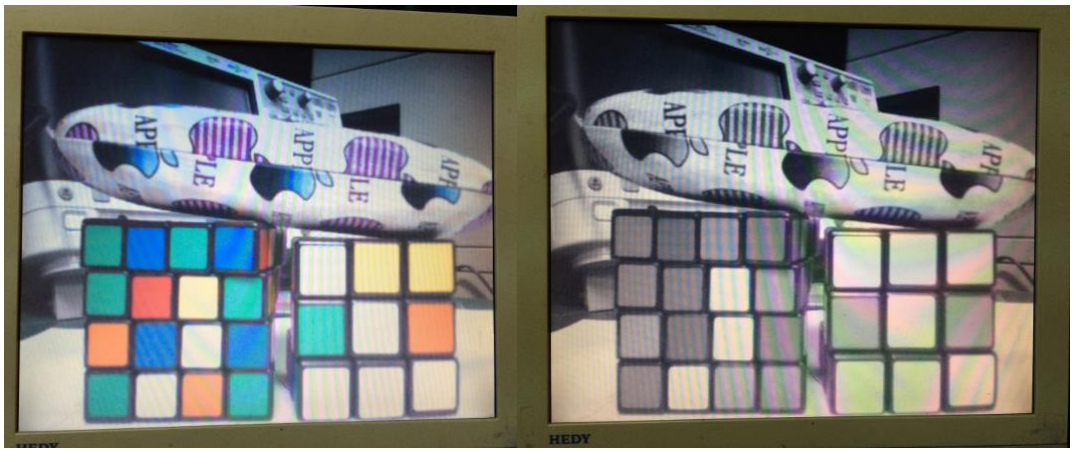
由于在处理过程中边界行列的舍取，相对于 5X5 而言，3x3 更多的减少了边界效应。3X3 如下图所示，采用一般的 3X3 领域插值算法，与采用边缘导向算法，处理后的得到图像，进行边沿提取进行对比，可见边缘导向算法后边缘和细节更加完整，同时，由于在改进型插值方案中，采取了对角线梯度法，红色和蓝色分量也更加接近真实。

这部分为发挥部分，虽然我只是“轻描淡写”的过去了，但是实际上在安防监控、机器视觉中，RAW 转 RGB 极其的重要。这部分在 ISP 中的学明为 Demosaic，好的 Demosaic 要求 1080P 图像分辨力（不同于分辨率）达到 1000 线以上，同时消除摩尔纹、辅助调整色彩，极限恢复原始图像。好的 Demosaic 决定了 ISP 的初衷。

5.10. 本章总结

Okay，本章内容到此结束，但其实我们才真正开始 VIP 的旅程。YCbCr422 转 RGB888、Bayer 阵列的恢复，均值滤波、中值滤波、Sobel 边缘检测、腐蚀运算、膨胀运算，这些只是图像处理中最基本的一些算法，只不过我们将算法移植到 FPGA 中后，实现了实时视频图像算法的硬件加速功能。

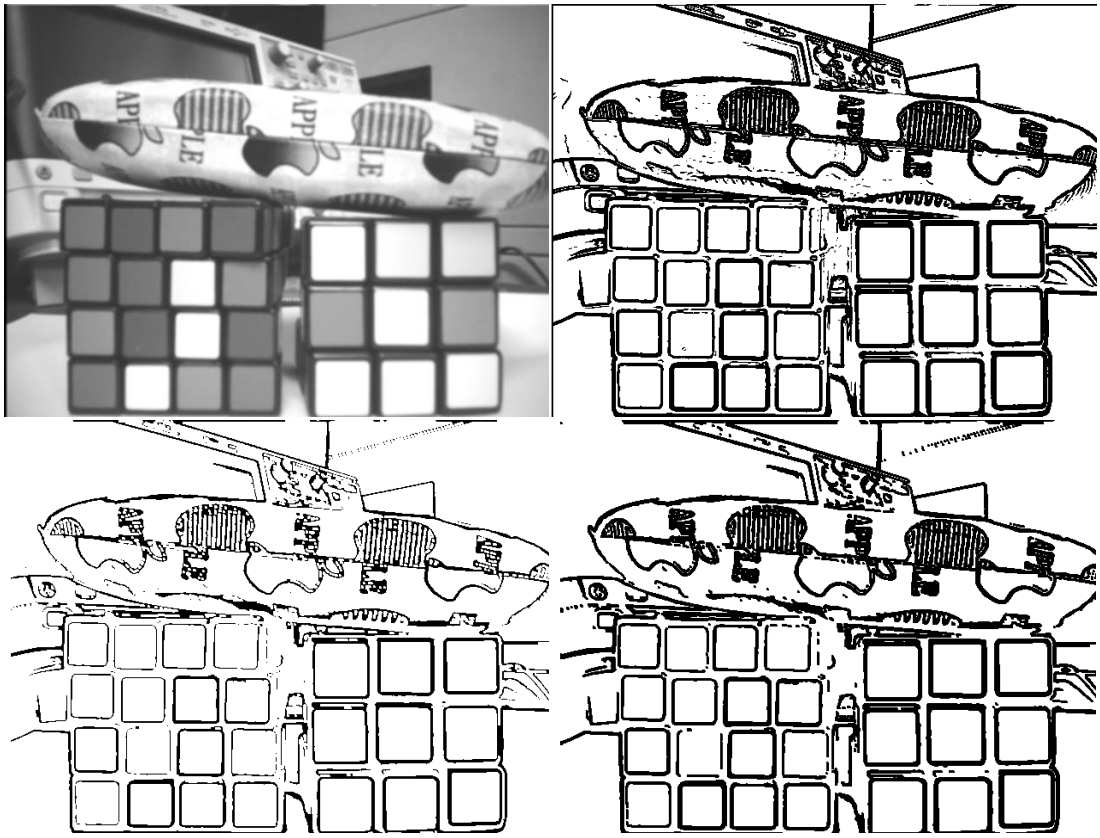
最初我们得到了 RGB565 的图像，如下图左。接着我们通过 RGB888 转 YCbCr 算法，得到了灰度的图像，如下图右所示：



但若当 RGB 转 YCbCr 实现了，YcbCr 转 RGB888 同样得实现，我们通过 YCbCr422→YcbCr444→RGB888，恢复了 OV7725 输出的 YcbCr 视频，得到了 RGB888 的视频图像。

此外，我们开始进行基本的视频图像算法处理，发挥 FPGA 硬件加速的优势。我们又进行了灰度图像的均值滤波、中值滤波、Sobel 边缘检测、腐蚀运

算、膨胀运算等算法。得到的图像分别如下所示（为了更清晰的看清楚细节，Bingo 是采用 VIP_Board Mini 板卡的 USB 功能采集的图像，但算法保持一致）：



最后，我们再次优化图像元，通过恢复 OV7725 的 Bayer 阵列，得到 RGB888 的 24Bit 图像，为未来彩色图像算法处理打下基础，一切未雨绸缪……

至此，我们完成了基本的 HDL-VIP 算法的范例，我们设计实现了 10 余种图像处理算法。当然这只是我们的一个开端，路漫漫其修远兮……FPGA 作为视频图像处理的硬件平台，以其并行高速的特性，以流水线的处理功能，完败了处理器的性能，这便是 HDL-VIP 的灵魂所在。

其实我们能做的还是很多，包括 3A 技术：即自动对焦(AF)、自动曝光(AE)和自动白平衡(AWB)，非局部均匀，直方图均衡，对比度增强、低照度算法，2D/3D 降噪等等等等等等等等…

大家如果有问题，欢迎 Email：thereturnofbingo@gmail.com，我愿意回答一切我愿意回答的问题!!! 同时，独家论坛 http://group.chinaaet.com/116，承担“视频图像显示开发”的问答汇总，有问题的尽管问，Bingo 将会在 NO.1 时间给大家答复。

基于 VIP_Board Big 在 VIP_Board Big 的基础上，发挥了更强大的硬件优势。FPGA 硬件加速实现逻辑功能的基础上，发挥强大的并行处理能力。有了算法，是放你的创造力，小小的平台，你能利用它实现图像算法的 VIP 验证、处理，你便赋予了它灵魂，掌握了 HDL-VIP 的精髓。

最后，请勿使用本教程作为其他商业性质的推广，或者未经允许抠取教程内容作为书记出版，请尊重版权。

“既然选择了 HDL-VIP，便不顾风雨兼程，一路走下去……”，期待读者能够释放你的创造力，在小平台上实现大智慧，Come on Together，准备爆发，我们的世界，一定会很精彩!!!!

6. HDL-VIP CMOS 视频图像算法处理 2

从无到有，是创造；从有到优，是进化。

前面五部分写与 2013-4 年，4 个青春已经度过，这一部分与 17 年新增。主要是提一些算法相关的思路，以及看心情更新一些 VIP Board Big V3.x 板卡上算法实现的结果。

先感慨一下：慢慢的，年纪大了（虽然快三十而立，但我觉得我心理年龄大哈），人也变得沉稳了。曾经付出了很多多，被黑金、睿智、艾曼抄了个遍；曾经也愤世嫉俗过，慢慢的发现淘宝到处都是我的影子反而觉得有点骄傲。那么多年的蹉跎，反而现在更愿意奉献 FPGA 届了。近期不断地在整理以前的成果，希望帮助像我一样曾经在学习道路上坎坷的朋友；既然所有资料都慢慢的公开，也希望那些淘宝的无良知的朋友，尽量多吸收一点我的精髓，少赚学生一点 RMB。在这个世界上，唯一可以永存的，那就是人格魅力！

下一步（2017 年上半年）我将会把我的精力投入到完善夏宇闻爷爷的《Verilog 数字系统教材》一书中区，完善下 SystemVerilog 相关内容，同事为这个世界更多的做一些贡献。

言归正传，如下：

6.1. RAW→RGB→Gray→中值滤波

详见 Crazy_FPGA_Examples 下“21_VIP_RAW2RGB2Gray_Medium”工程。

大部分机器视觉，都是在灰度域处理的，因此我们需要纯净的 Gray 图像。

对于机器视觉或者工业领域等对图像要求高的，还需要做一定的预处理，此时肯定不会用 Sensor 本身输出的 RGB565 图像，因此本工程作为所有后续算法升级演变的模型，首选将 Bayer 转 RGB，继而再通过 YUV 转换提取出灰度图像，最后做一步简单的中值滤波了事。

备注：可优化部分，希望有能力的人能在此基础上做的更好：

（1）优化 RAW2RGB 位 5*5 的模式，且考虑方向梯度

（2）优化滤波，修改为高斯滤波，甚至非局部平均滤波、BM3D 等 2D 滤波算法

6.2. RAW→RGB→Gray→Median→Sobel→中值滤波→腐蚀→膨胀

详见 Crazy_FPGA_Examples 下

“22_VIP_RAW2RGB2Gray_Medium_Sobel_Erosion_Dilation”工程。

本章作为上面 6.1 部分的衍生，在得到的中值滤波灰度图像后，进行了第五章 Sobel 边缘检测、腐蚀运算、膨胀运算的移植。虽然结果看起来有点花哨，但估计小朋友们还不知道有啥用处：

（1）用于轮廓、二维码，姿态等识别

（2）作为机器视觉的硬件加速模块

6.3. 未完待续，欢迎更多朋友与我探讨共进！

后续打算继续奉献，期待更多朋友与我探讨共进。

/**/

官方论坛：	http://www.crazyfpga.com/	欢迎您的加入
官方淘宝：	http://crazyfpga.taobao.com/	期待您的拥有
官方公众号：	crazyfpga	期待您的陪伴
官方 QQ 群：	248619895	期待您的加入
CrazyBingo 邮箱：	crazyfpga@vip.qq.com	期待您的斧正

/**/