基于 FPGA 的彩色图像灰度化的实现

一、图像灰度化处理的基本原理

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。

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。

$$\text{gray}_1(i,j) = R(i,j)$$
$$\text{gray}_2(i,j) = G(i,j)$$
$$\text{gray}_3(i,j) = B(i,j)$$

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。

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。如下：

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

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，这里将分母 3 改为 256，这里公式变为如下：

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

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：

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

方法 4：加权平均法

根据重要性及其它指标，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+0.144=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，为了能避免低速的浮点运算以及除法运算，可以将式子缩放 1024 倍来实现运算算法，如下：

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

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：

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

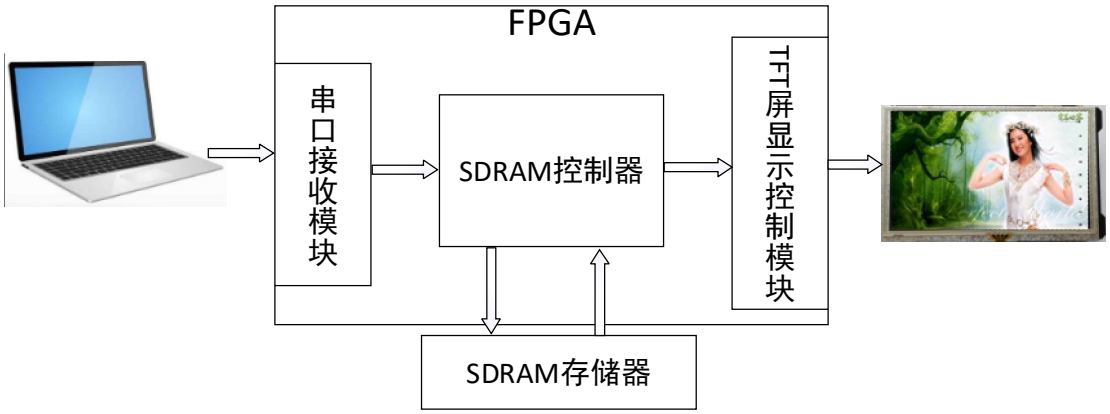
也可以压缩到 8 位以内，式子变为如下：

$$\text{gray(i,j)} = [75 * R(i,j) + 147 * G(i,j) + 36 * B(i,j)] \gg 8$$

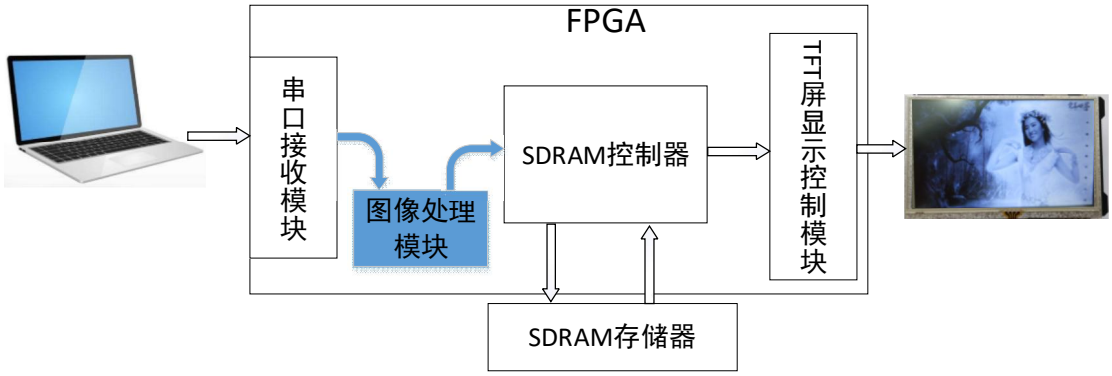
二、图像的灰度化处理方法实现

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，并结合小梅哥团队出品《自学笔记——设计与验证》一书中串口传图工程对三种不同实现方法进行板级的验证。

首先从整体结构框架来看如何在已有的串口传图工程中加入图像处理算法模块。如下图所示为串口传图工程的整体框架。



在已有工程框架的串口模块之后，数据缓存到 SDRAM 之前加入图像处理模块（这里指 RGB 转 Gray 算法），其他模块之间保持不变，可以充分利用已有的工程。



接下来主要对 RGB 转 Gray 算法 FPGA 实现做相关的描述。考虑到图片最后显示在 5 寸（800*480）TFT 屏，找来一张像素为 800*480 的的彩色图片如下：

方法 3 平均值法的实现

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：

```
1   module RGB2Gray(
2       clk,
3       rst_n,
4       rgb_inen,
5       red,
6       green,
7       blue,
8       gray,
9       gray_outen
10  );
11      input clk;          //时钟
12      input rst_n;        //异步复位
13      input rgb_inen;     //rgb 输入有效标识
14      input [7:0]red;     //R 输入
15      input [7:0]green;   //G 输入
16      input [7:0]blue;    //B 输入
17      output [7:0]gray;   //GRAY 输出
18      output reg gray_outen; //gray 输出有效标识
19
```

```
20  //求平均法 GRAY = (R+B+G)/3=（(R+B+G)*85）>>8
21      wire [9:0]sum;
22      reg [15:0]gray_r;
23
24      assign sum = red + green + blue;
25
26      always@(posedge clk or negedge rst_n)
27      begin
28          if(!rst_n)
29              gray_r <= 16'd0;
30          else if(rgb_inen)
31              gray_r <= {sum,6'b000000}+{sum,4'b0000}+ {sum,2'b00} + sum;
32          else
33              gray_r <= 16'd0;
34      end
35
36      assign gray = gray_r[15:8];
37
38      always@(posedge clk or negedge rst_n)
39      begin
40          if(!rst_n)
41              gray_outen <= 1'b0;
42          else if(rgb_inen)
43              gray_outen <= 1'b1;
44          else
45              gray_outen <= 1'b0;
46      end
47
48  endmodule
```

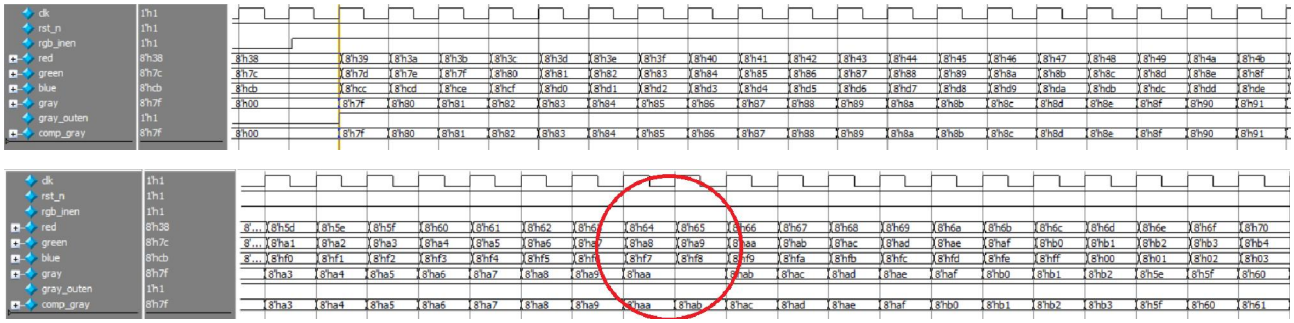
接下来进行仿真，仿真文件代码如下：

```
1   `timescale 1ns/1ns
2   `define CLK_PERIOD 20
3
4   module RGB2Gray_tb;
5       reg clk;
6       reg rst_n;
7       reg rgb_inen;
8       reg [7:0]red;
9       reg [7:0]green;
10      reg [7:0]blue;
11      wire [7:0]gray;
12      wire gray_outen;
13
14      reg [7:0]comp_gray;
15
```

```
16      RGB2Gray RGB2Gray(
17          .clk(clk),
18          .rst_n(rst_n),
19          .rgb_inen(rgb_inen),
20          .red(red),
21          .green(green),
22          .blue(blue),
23          .gray(gray),
24          .gray_outen(gray_outen)
25      );
26
27      initial clk = 1'b1;
28      always #(`CLK_PERIOD/2)clk = ~clk;
29
30      initial begin
31          rst_n = 0;
32          rgb_inen = 0;
33          red = 0;
34          green = 0;
35          blue = 0;
36
37          #(`CLK_PERIOD*200+1);
38          rst_n = 1;
39          red = 56;
40          green = 124;
41          blue = 203;
42          #2000;
43
44          rgb_inen = 1;
45          repeat(256)begin
46              #(`CLK_PERIOD)
47              red = red + 1;
48              green = green + 1;
49              blue = blue + 1;
50          end
51          rgb_inen = 0;
52
53          #2000;
54          $stop;
55      end
56
57      always@(posedge clk)
58      if(rgb_inen == 1'b1)
59          comp_gray <= ({2'b00,red}+{2'b00,green}+{2'b00,blue})/3;
60      else
```

```
61          comp_gray <= 0;
62
63  endmodule
```

为了能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57~61 行的代码，这样可以通过对比 comp_gray 和 gray 的值就可以很好的验证设计模块的正确性。其仿真波形图如下：



在后面有个地方可以看到从一个地方开始 comp_gray 和 gray 的值相差 1，可以分析下出现这个问题的原因，其实 gray 和 comp_gray 的计算公式并非完全一样，gray 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 comp_gray 是直接求的平均。可以通过 red、green、blue 这 3 个数据分别对 gray 和 comp_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 comp_gray 是稍存在偏差的。

接下来就是将该模块加入到顶层文件进行板级验证，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，这里就涉及到 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

24ibt 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 RGB656 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 RGB656 R4 R3 R2 R1 R0 G5 G4 G3 G2 G1 G0 B4 B3 B2 B1 B0

24ibt 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。

以下代码是例化彩色图像灰度化模块的代码，这里直接将转换关系写入了进去。

```
    //彩色图像灰度化
    RGB2Gray RGB2Gray(
        .clk(Clk50M),
        .rst_n(Rst_n),
        .rgb_inen(rgb565_en),
        .red({rgb565[15:11],rgb565[13:11]}),
        .green({rgb565[10:5],rgb565[6:5]}),
        .blue({rgb565[4:0],rgb565[2:0]}),
        .gray(gray),
        .gray_outen(gray_outen)
    );
```

整个工程全编译下载到 AC620，通过自己编写的简易串口发送图片数据上位机将彩色图片数据发送给 AC620，板子上 5 寸 TFT 屏显示结果如下：

从实际效果图可以看出，将彩色图片转换成了黑白图片，达到了预期的效果。

方法 4 加权平均法的实现

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。

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：

```
1   module RGB2Gray(
2       clk,
3       rst_n,
4       rgb_inen,
5       red,
6       green,
7       blue,
8       gray,
9       gray_outen
10  );
11      input clk;          //时钟
```

```
12      input rst_n;        //异步复位
13      input rgb_inen;     //rgb 输入有效标识
14      input [7:0]red;     //R 输入
15      input [7:0]green;   //G 输入
16      input [7:0]blue;    //B 输入
17      output [7:0]gray;   //GRAY 输出
18      output reg gray_outen; //gray 输出有效标识
19
20  //典型灰度转换公式 Gray = R*0.299+G*0.587+B*0.114=(R*75 + G*147 + B*36) >>8
21      wire [15:0]w_R;
22      wire [15:0]w_G;
23      wire [15:0]w_B;
24      reg [16:0]sum;
25
26      assign w_R = {red,6'b000000} + {red,3'b000} + {red,1'b0} + red;
27      assign w_G = {green,7'b0000000} + {green,4'b0000} + {green,1'b0} +
green;
28      assign w_B = {blue,5'b00000} + {blue,2'b00};
29
30      always@(posedge clk or negedge rst_n)
31      begin
32          if(!rst_n)
33              sum <= 17'd0;
34          else if(rgb_inen)
35              sum <= w_R + w_G + w_B;
36          else
37              sum <= 17'd0;
38      end
39
40      assign gray = sum[15:8];
41
42      always@(posedge clk or negedge rst_n)
43      begin
44          if(!rst_n)
45              gray_outen <= 1'b0;
46          else if(rgb_inen)
47              gray_outen <= 1'b1;
48          else
49              gray_outen <= 1'b0;
50      end
51
52  endmodule
```

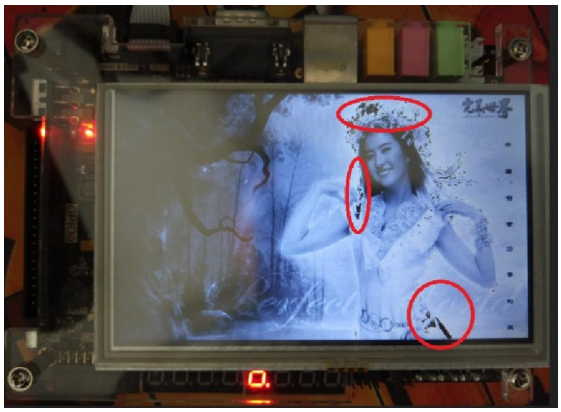
该方法在实现过程中遇到一个错误，通过仿真方式很容易的发现并进行了改正，发生错误的地方是在计算 w_R、w_G、w_B 的地方，错误代码如下：

```
assign w_R = red<<6 + red<<3 + red<<1 + red;
assign w_G = green<<7 + green<<4 + green<<1 + green;
assign w_B = blue<<5 + blue<<2;
```

在等式左边计算过程中由于 red、green、blue 是位宽 8bit 的数，在移位后数据高位部分就丢失掉了，比如，green<<7 得到的是{green[0],7'b0000000},并非我期望的 green 乘以 2 的 7 次方。后来改为如下正确的代码后就可以得到正确的结果了。像这样的地方不止一处，类似这样的移位都可能会出现这样的问题，应该需要我们注意。

```
assign w_R = {red,6'b000000} + {red,3'b000} + {red,1'b0} + red;
assign w_G = {green,7'b0000000} + {green,4'b0000} + {green,1'b0} + green;
assign w_B = {blue,5'b00000} + {blue,2'b00};
```

仿真过程与方法 3 类似，这里就不重复描述，该方法实现效果图如下：



通过观察显示的图片发现有几个偏白色的地方出现的是黑色，感觉不太正常。通过分析发现这些地方在处理前是纯白色（R:255,G:255,B:255），通过公式 Gray = (R*75 + G*147 + B*36) >>8 进行计算时，前面的乘法相加后的结果为 17'h100FE，数值超过了 16bit，这样通过截取的 8~15 位获得的 Gray 为 8'h00，导致最终显示为黑色。为了对显示的结果做一定的优化，对计算结果超过 16 位（也就是 sum[16]为 1 时）让 Gray 值为最大值 255，将上面代码中第 40 行代码做如下调整，代码调整如下：

```
assign gray = sum[16]?8'hFF:sum[15:8];
```

做了调整后的实现效果如下：

显示效果就比之前要好了。

2、查找表法

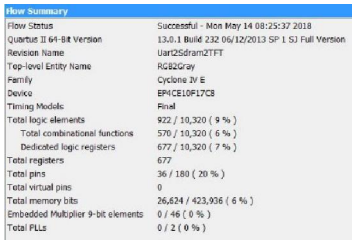
通过观察计算公式发现，R、G、B 前均乘以了一个定值，然后对乘法之后的结果相加，最后右移 8 位，上面采用直接计算法实现是对乘法采用的移位相加方法计算，为了对这步计算时间的进行减少，采用查找表方法实现，该方法主要的优势是速度快（结论主要来源于网上，本次实验未找到比较好的证明方法），但占用的存储器会更多，因为需要将 R、G、B 乘以系数之后的数值存储在 ROM 中，然后通过读取 ROM 方式来得到计算之后的数值。这里使用 quartusII 软件添加 3 个 ROM IP 核，分别对 R*75、G*147、B*36（0≤R≤255，0≤G≤255，0≤B≤255）建立 3 个 mif 文件，然后在 ROM IP 核中分别添加 mif 文件进行初始化。具体代码如下：代码中 ROM_R、ROM_G、ROM_B 分别存储着 R*75、G*147、B*36（0≤R≤255，0≤G≤255，0≤B≤255）256 个数值。

```
1  module RGB2Gray(
2      clk,
3      rst_n,
4      rgb_inen,
5      red,
6      green,
7      blue,
8      gray,
9      gray_outen
10 );
11     input clk;          //时钟
12     input rst_n;        //异步复位
13     input rgb_inen;     //rgb 输入有效标识
14     input [7:0]red;     //R 输入
15     input [7:0]green;   //G 输入
16     input [7:0]blue;    //B 输入
17     output [7:0]gray;   //GRAY 输出
18     output reg gray_outen; //gray 输出有效标识
19
20 //查找表方式，可以省去乘法运算 Gray =(R*75 + G*147 + B*36) >>8，将 3 个分量乘以系数后的数值存储在 ROM 中
21 wire [14:0]w_R;
22 wire [15:0]w_G;
23 wire [13:0]w_B;
24
25 reg [16:0]sum;
26 reg [1:0]r_gray_outen;
27
28 ROM_R ROM_R(
29     .address(red),
```

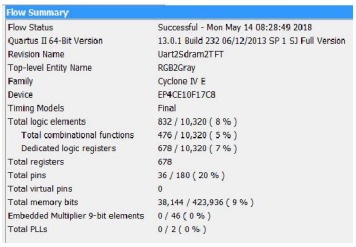
```
30      .clock(clk),
31      .rden(rgb_inen),
32      .q(w_R)
33  );
34
35  ROM_G ROM_G(
36      .address(green),
37      .clock(clk),
38      .rden(rgb_inen),
39      .q(w_G)
40  );
41
42  ROM_B ROM_B(
43      .address(blue),
44      .clock(clk),
45      .rden(rgb_inen),
46      .q(w_B)
47  );
48
49  always@(posedge clk)
50      {gray_outen,r_gray_outen} <= {r_gray_outen,rgb_inen};
51
52  always@(posedge clk or negedge rst_n)
53  begin
54      if(!rst_n)
55          sum <= 17'd0;
56      else if(r_gray_outen[1])
57          sum <= w_R + w_G + w_B;
58      else
59          sum <= 17'd0;
60  end
61
62  assign gray = sum[16]?8'hFF:sum[15:8];
63
64  endmodule
```

该方法实现的仿真与板级验证效果图与采用直接计算方法的结果是一样的。

对直接采用公式计算法和采用查找表法两种不同实现方式全编译后所占资源情况，左边是直接采用公式计算法实现的，所占 memory 要比右边图采用查找表法的要少。关于速度的快慢，还没想到怎么去验证。



Flow Summary	
Flow Status	Successful - Mon May 14 08:25:37 2018
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	Uart2SdramZTFT
Top-level Entity Name	RGB2Gray
Family	Cyclone IV E
Device	EP4CE10F17C8
Timing Models	Final
Total logic elements	922 / 10,320 (9 %)
Total combinational functions	570 / 10,320 (6 %)
Dedicated logic registers	677 / 10,320 (7 %)
Total registers	677
Total pins	36 / 180 (20 %)
Total virtual pins	0
Total memory bits	26,624 / 423,936 (6 %)
Embedded Multiplier 9-bit elements	0 / 46 (0 %)
Total PLLs	0 / 2 (0 %)



Flow Summary	
Flow Status	Successful - Mon May 14 08:28:49 2018
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	Uart2SdramZTFT
Top-level Entity Name	RGB2Gray
Family	Cyclone IV E
Device	EP4CE10F17C8
Timing Models	Final
Total logic elements	832 / 10,320 (8 %)
Total combinational functions	476 / 10,320 (5 %)
Dedicated logic registers	678 / 10,320 (7 %)
Total registers	678
Total pins	36 / 180 (20 %)
Total virtual pins	0
Total memory bits	38,144 / 423,936 (9 %)
Embedded Multiplier 9-bit elements	0 / 46 (0 %)
Total PLLs	0 / 2 (0 %)

方法 3 实现的效果与方法 4 有略微的差别，拍的照片没法明显看出来，但在实现过程中，从一种方法 3 切换到方法 4，是能看到图像有明显变亮。

对于上述中的几种实现方法可以一起放在一个文件中，通过条件编译的方式进行选择某种实现方式，实现的部分代码如下：

```
//`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_inen,
    red,
    green,
    blue,
    gray,
    gray_outen
);

    input clk;          //时钟
    input rst_n;        //异步复位
    input rgb_inen;     //rgb 输入有效标识
    input [7:0]red;     //R 输入
    input [7:0]green;   //G 输入
    input [7:0]blue;    //B 输入
    output [7:0]gray;   //GRAY 输出
    output reg gray_outen; //gray 输出有效标识

`ifdef AVERAGE  //求平均法 GRAY=(R+B+G)/3=（(R+B+G)*85）>>8
...
`endif

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

```
`ifdef LUT//查找表方式
...
`endif

endmodule
```