

第六届

全国大学生集成电路创新创业大赛

报告类型\*： Upsampling IP设计说明

参赛杯赛\*： 景嘉微杯

作品名称\*： 一种应用于图形显示的 Upsampling IP

队伍编号\*： CICC1837

团队名称\*： 啊对对对对对对

# 目录

[一、算法设计说明 1](#_Toc111820224)

[1.1软件算法设计 1](#_Toc111820225)

[1.1.1最近邻插值与新型双线性插值 1](#_Toc111820226)

[1.1.2均值调整 2](#_Toc111820227)

[1.1.3边缘插值与高斯平滑 3](#_Toc111820228)

[1.1.4各种组合算法及其性能 5](#_Toc111820229)

[1.2硬件设计框架 9](#_Toc111820230)

[1.2.1PL端硬件架构 9](#_Toc111820231)

[1.2.2单颜色分量插值模块 12](#_Toc111820232)

[1.2.3边缘插值+均值调整模块和高斯平滑+均值调整模块 15](#_Toc111820233)

[1.2.4PS端设计 15](#_Toc111820234)

[二、实现函数说明 17](#_Toc111820235)

[2.1头文件 17](#_Toc111820236)

[2.1.1Upsampling.h 17](#_Toc111820237)

[2.1.2interpolation.h 17](#_Toc111820238)

[2.2源文件 18](#_Toc111820239)

[2.2.1interpolating.c 18](#_Toc111820240)

[2.2.2upsampling.c 19](#_Toc111820241)

[2.2.3test.c 20](#_Toc111820242)

[三、寄存器说明 21](#_Toc111820243)

[3.1存储1k像素5行3列寄存器 21](#_Toc111820244)

[3.2边缘插值+均值调整所需的9点寄存器 21](#_Toc111820245)

[3.3边缘插值+均值调整模块内17点寄存器 22](#_Toc111820246)

[3.4边缘插值+均值调整后的4k图片6行9列寄存器 22](#_Toc111820247)

[3.5高斯平滑+均值调整所需的36点寄存器 23](#_Toc111820248)

[3.6高斯平滑+均值调整模块内17点寄存器 23](#_Toc111820249)

[四、RTL模块设计说明 25](#_Toc111820250)

[4.1top 25](#_Toc111820251)

[4.1.1模块框图 25](#_Toc111820252)

[4.1.2连线端口 25](#_Toc111820253)

[4.1.3模块功能 26](#_Toc111820254)

[4.2single\_color\_top 26](#_Toc111820255)

[4.2.1模块框图 26](#_Toc111820256)

[4.2.2连线端口 26](#_Toc111820257)

[4.2.3模块功能 29](#_Toc111820258)

[4.3interpolate\_16point 29](#_Toc111820259)

[4.3.1模块框图 29](#_Toc111820260)

[4.3.2连线端口 29](#_Toc111820261)

[4.3.3模块功能 32](#_Toc111820262)

[4.4gauss\_filter\_dot\_36\_4k 32](#_Toc111820263)

[4.4.1模块框图 32](#_Toc111820264)

[4.4.2连线端口 32](#_Toc111820265)

[4.4.3模块功能 35](#_Toc111820266)

[4.5edge\_16\_generate 35](#_Toc111820267)

[4.5.1模块框图 36](#_Toc111820268)

[4.5.2连线端口 36](#_Toc111820269)

[4.5.3模块功能 37](#_Toc111820270)

[4.6average\_adjust\_16point 37](#_Toc111820271)

[4.6.1模块框图 37](#_Toc111820272)

[4.6.2连线端口 37](#_Toc111820273)

[4.6.3模块功能 39](#_Toc111820274)

[五、仿真验证环境及说明 40](#_Toc111820275)

[5.1仿真流程说明 40](#_Toc111820276)

[5.2仿真代码与波形图说明 41](#_Toc111820277)

[六、性能评估说明 44](#_Toc111820278)

[6.1算法性能 44](#_Toc111820279)

[6.2硬件资源 47](#_Toc111820280)

[七、FPGA验证报告 49](#_Toc111820281)

[7.1FPGA验证 49](#_Toc111820282)

[7.2设计优缺点 50](#_Toc111820283)

# 一、算法设计说明

## 1.1软件算法设计

### 1.1.1最近邻插值与新型双线性插值

（1）最近邻插值

最近邻插值是最简单的插值算法，其原理是令变换后像素的像素值等于距它最近的输入像素的像素值。以放大2倍为例，其原理可用下图1.1表示，其中颜色相同的像素点表示其像素值也相同。



图 1.1 最近邻插值原理图

（2）新型双线性插值

双线性插值即同时在x，y两个方向上进行线性插值，在某一方向上，插值的像素点与其两边的像素点在同一直线上。因此，如图1.2所示，为了求得某一点P的像素值，可以先进行水平方向的线性插值，求出点R1和R2的像素值，再进行竖直方向的线性插值，求出P的像素。最终，为求得点P处的像素值，其计算公式如下：

 (1-1)



图 1.2 双线性插值原理图

对于传统的双线性插值放大而言，一方面，目标图像会利用原有图像的像素点，而在本设计中，由于原有图像是经过均值下采样获得的，其本身像素点的值并不准确；另一方面，传统的双线性插值算法会导致整体图像的偏移，进而影响性能。因此，出于这两方面的考虑，本文提出了一种新型的双线性插值。该方法采用9个1k图片的像素点来求得中心1k像素点对应的4k图片的16个像素点，原1k图片的像素点未被直接赋值给4k图片，下图1.3为新型双线性插值的原理图，其中红色圆点为已知的1k图片的像素点，蓝色虚线框包围的16个小圆点为所求的对应的4k图片的16个像素点。

现求解图中的蓝色小圆点P以便演示新型双线性插值算法，从图中可以看出P点相对于的偏移为，带入传统的双线性插值公式，可得P点的像素为：。可以看出，新型双线性插值算法生成的4k像素并未被直接赋值为原1k像素值，仅利用了周围9点的相对和绝对大小，另一方面，此方法生成的4k图片与原1k图片是中心对齐的，避免了因放大造成的图像偏移。



图 1.3 新型双线性插值原理图

### 1.1.2均值调整

在本设计中，1k图像是经过均值下采样获得的，其有一特殊的性质，即原4k图像的16点的平均值等于相应位置1k图片的像素值，在还原的4k图像中，我们也希望还原的4k图像仍然保持这一性质，因此提出了均值调整这一操作。

不妨设所求的1k像素为average，利用最近邻、新型双线性或下文中边缘插值求出的1k像素代表的4k像素的16个点为dot\_16\_4k，经过均值调整过后的新的16个点为adjust\_dot\_16\_4k，则有如下等式成立：

 (1-2)

其中sum为求和函数，双重sum表示对16点的求和，fix为向0取整函数，在此情况下其等价于向下取整。对于新的16点而言，若其值小于0，则赋值为0，若其值大于255，则赋值为255。这样，经过均值调整这一步骤后，新的16点就近似满足生成的4k图像的16点的平均值等于相应位置1k图像的像素值这一性质。

### 1.1.3边缘插值与高斯平滑

（1）边缘插值

均值下采样与双线性插值放大过程中均会导致高频分量的丢失，进而影响整体的性能，为了弥补这一缺失，本文提出了一种边缘插值的算法，其利用1k图片的9个像素点产生中心点对应的4k图片的16个像素点。由于本边缘插值算法通过水平和竖直方向分别产生4k图片的16个点，然后将水平方向产生的16点和竖直方向产生的16点进行均值求和作为最终结果的16点，水平方向产生16点与竖直方向产生16点原理相同，因此，以下仅介绍水平方向产生4k图片16点像素的流程。

对于水平方向边缘插值的第一行而言，先用线性插值求出左右两边的像素值，即图1.4中的蓝色圆点pixel\_left和pixel\_right，可得：

 (1-3)

然后求出第一行左右两点相对于中间点的梯度d\_left和d\_right，即：

 (1-4)

其中

 (1-5)



图 1.4 左右像素和梯度求解示意图

为了增强图像的高频部分，本文采用下图1.5所示方法求解第一行的4个像素点：



图 1.5 求解4点像素示意图

即采用阶跃函数的形式，将像素分为左右两个部分，而我们所求的4个点的位置位于，当所求像素的横坐标小于跳变点x1，其像素值等于pixel\_left，当所求像素的横坐标大于x1时，其像素值等于pixel\_right，而x1的计算公式如下：

 (1-6)

根据以上公式可知（为避免分母为0，实际软硬件实现时已将分母消去），当1k图片的某一点处于边界处时，利用上述方法可以达到增强边界的效果，即实现下图所示的功能：



图 1.6 边界增强示意图

同理，我们也可以求得其余三行的像素点，只不过3、4行的像素点运用的是9点中的下面两行，而1、2行运用的是9点中的上面两行，至此我们求出了水平方向边缘插值的16个像素点。根据水平方向边缘插值的原理，我们同样可以求得竖直方向边缘插值的16个点，然后我们将水平与竖直方向求得的点取平均，即最终边缘插值求得的16个点。对于1k图像的边界点，我们采用两次边界扩展，然后再进行边缘插值。

（2）高斯平滑

从上文中的边缘插值可以看出，该插值方法过分的拉大了边缘的梯度，中间不存在过渡区，为了缓解这一问题，我们在生成4k图片的基础上对4k图片进行的高斯平滑，其模板如下：



### 1.1.4各种组合算法及其性能

对于以下不同组合算法而言，其性能指标PSNR、SSIM以及LPIPS均采用官方提供的Measure.py进行测量获得。

组合1：新型双线性插值+均值调整

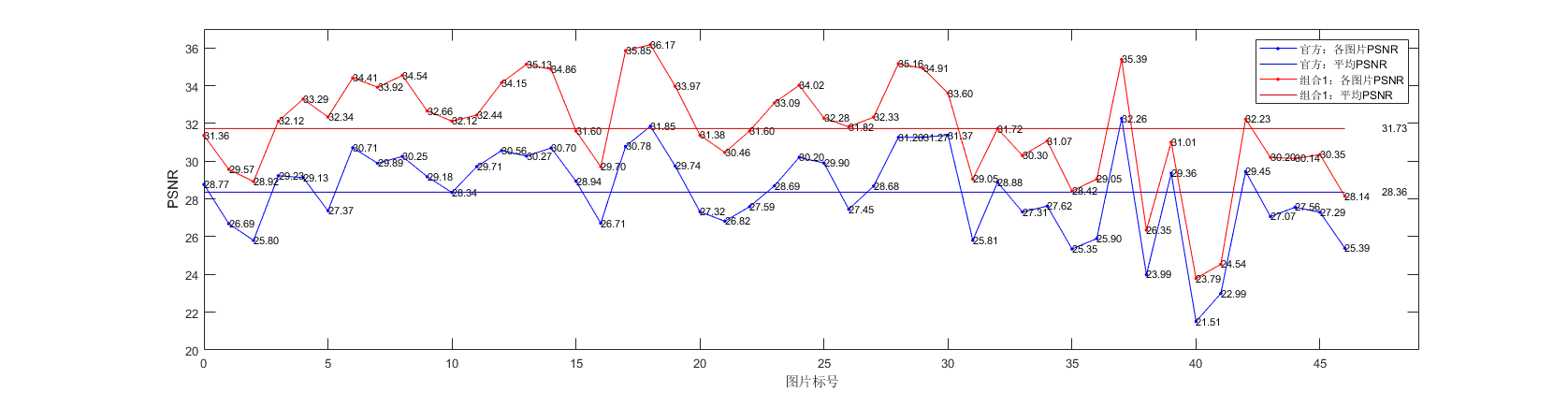


图 1.7 组合1 PSNR性能

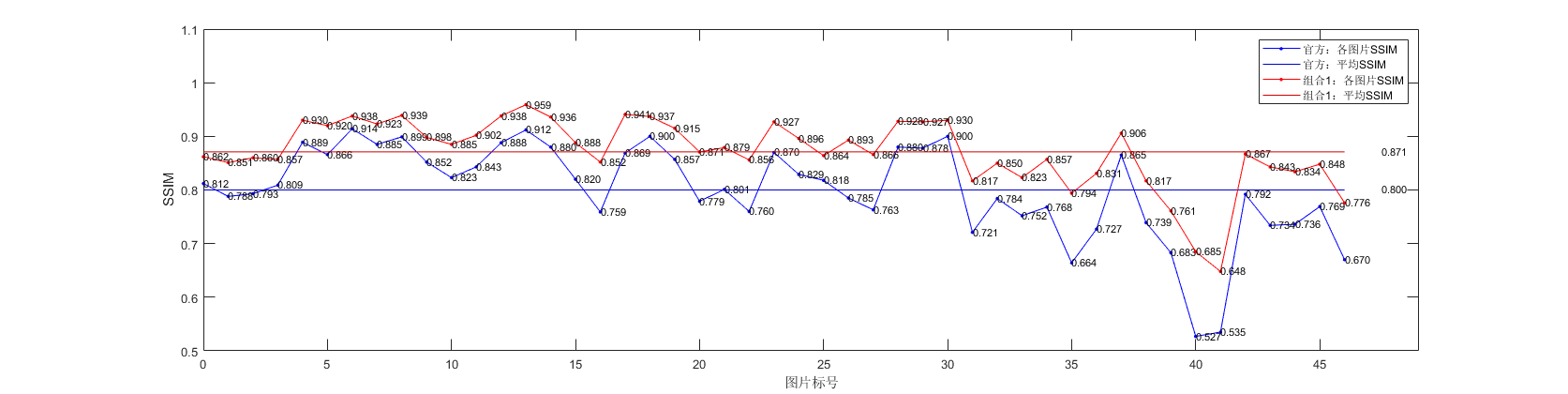


图 1.8 组合1 SSIM性能

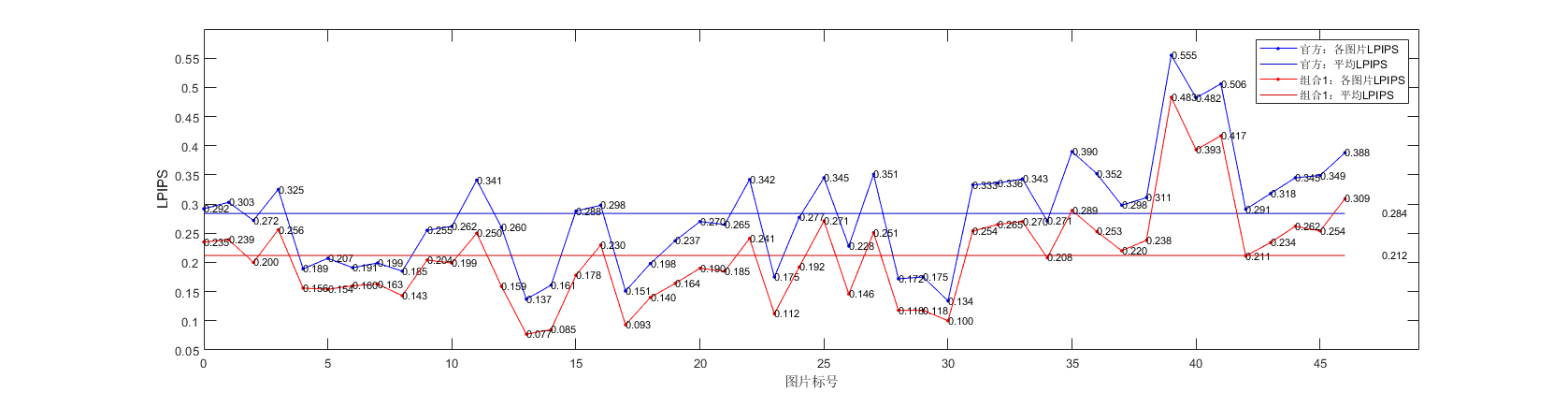


图 1.9 组合1 LPIPS性能

组合2：新型双线性插值+均值调整+高斯平滑+均值调整

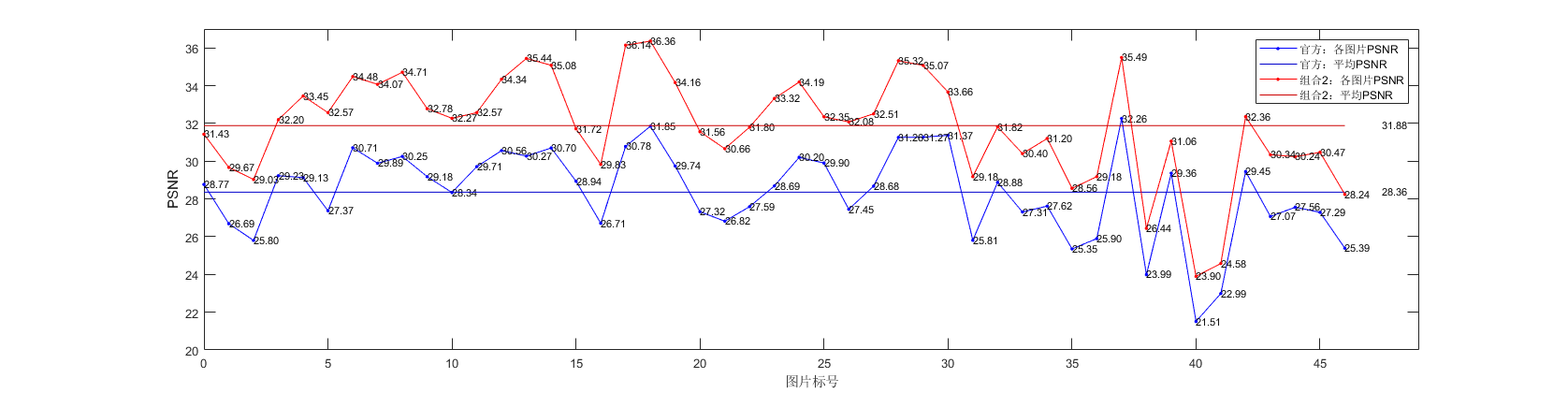


图 1.10 组合2 PSNR性能

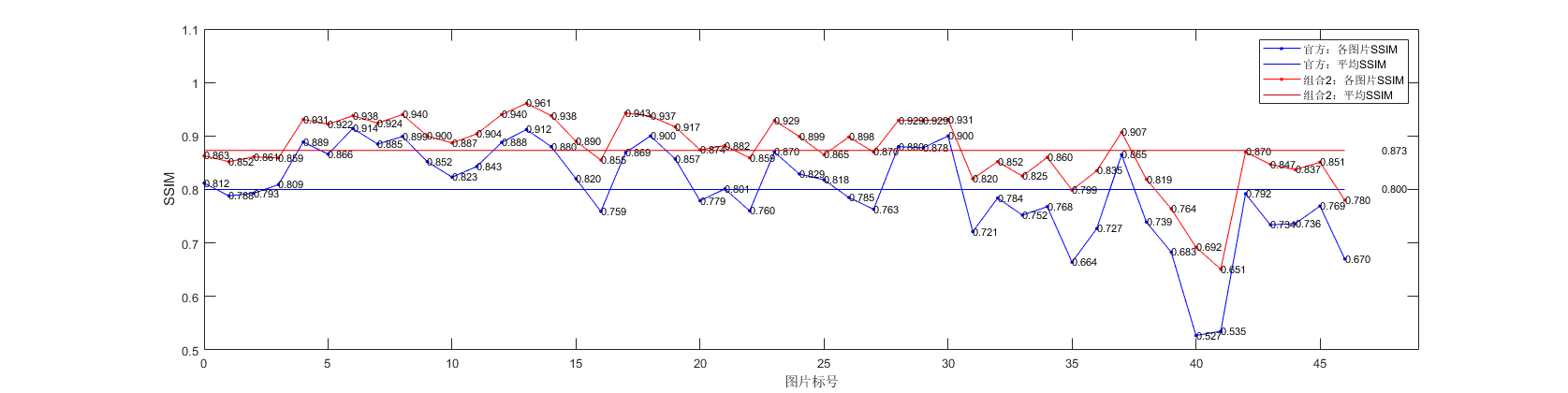


图 1.11 组合2 SSIM性能

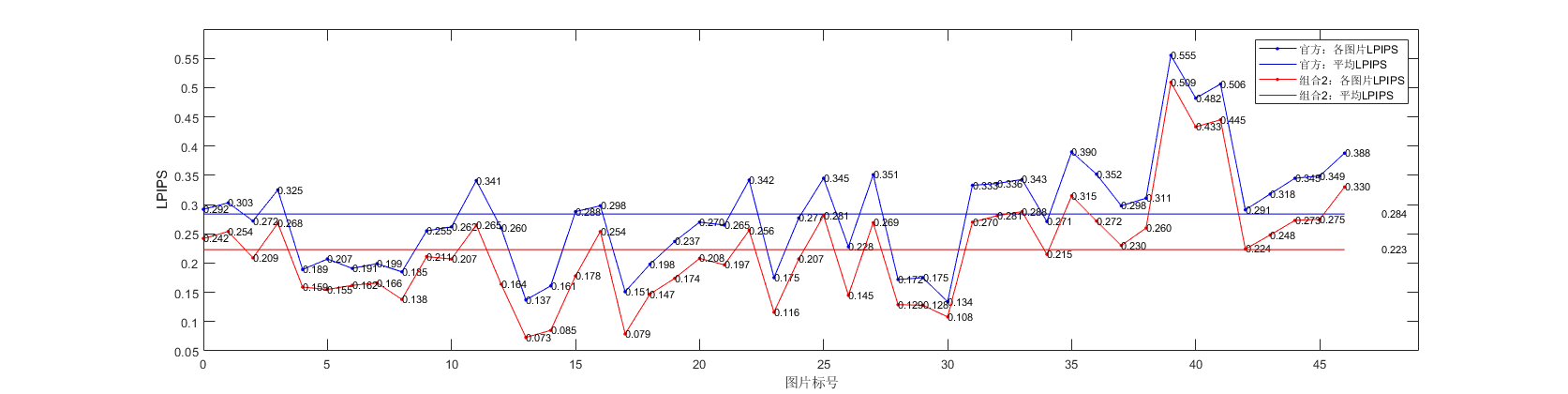


图 1.12 组合2 LPIPS性能

组合3：分区域插值+均值调整+高斯平滑+均值调整

此处的分区域插值指的是利用33的水平和竖直sobel算子与插值的9个像素点进行卷积，求出水平梯度d\_r和竖直梯度d\_c，若且，其中，则该9点采用最近邻插值，若或，其中，则该9点采用边缘插值，其余区域采用新型双线性插值。

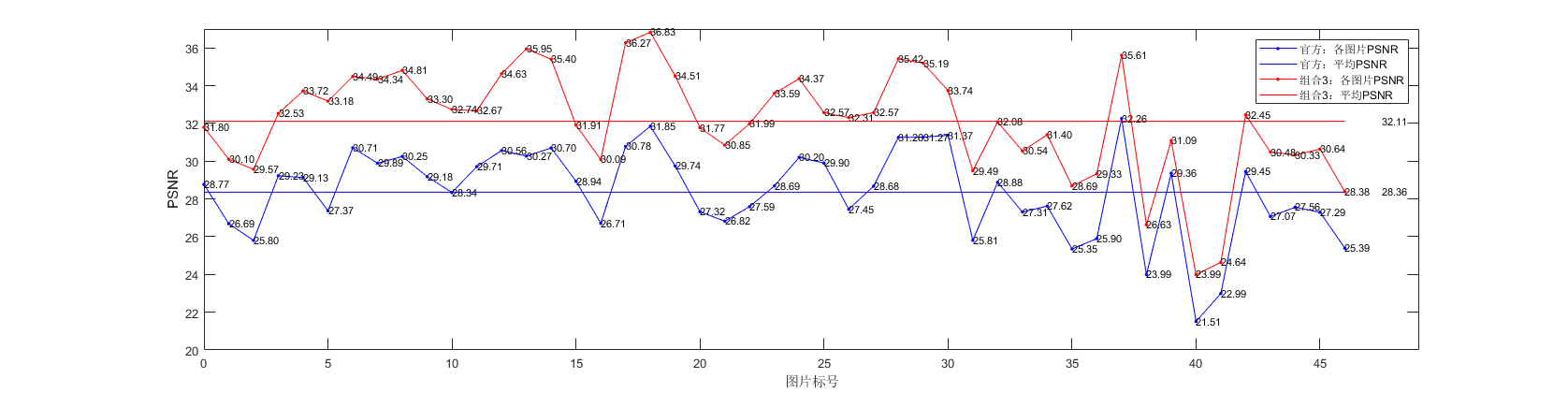


图 1.13 组合3 PSNR性能

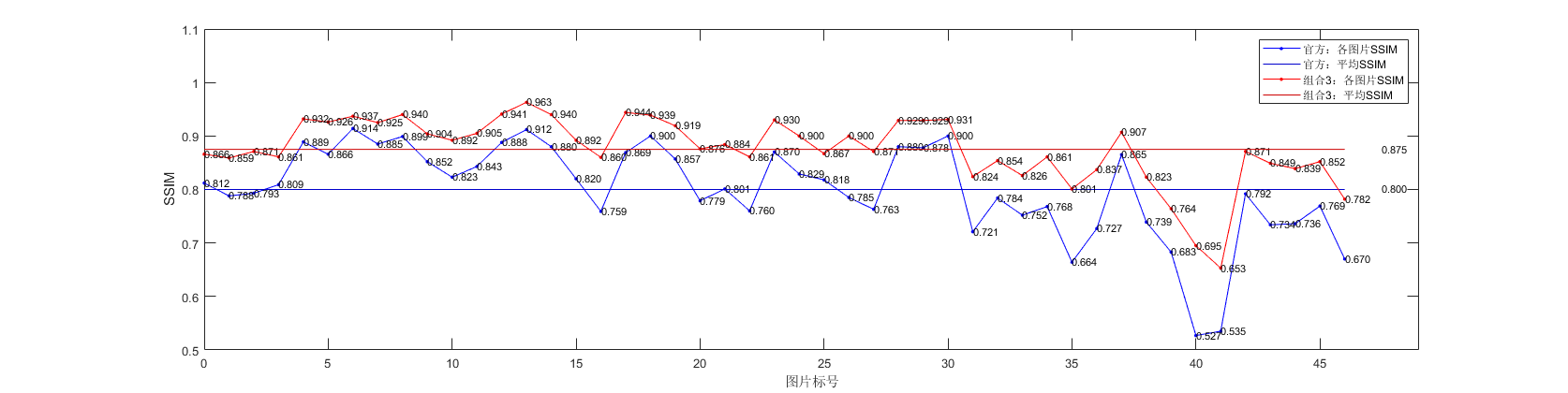


图 1.14 组合3 SSIM性能

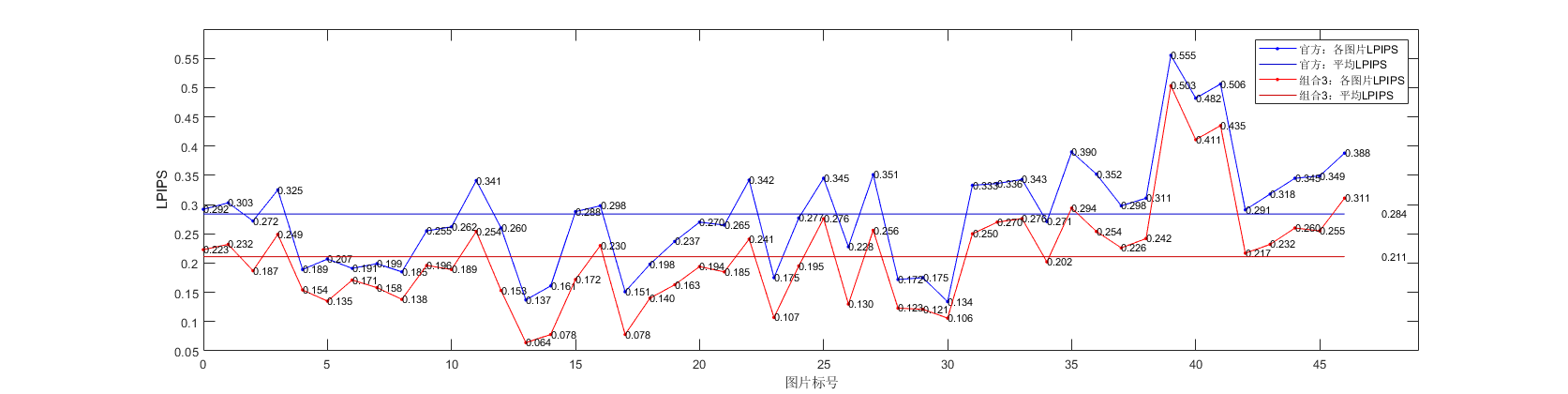


图 1.15 组合3 LPIPS性能

组合4：3区域均采用边缘插值+均值调整+高斯平滑+均值调整

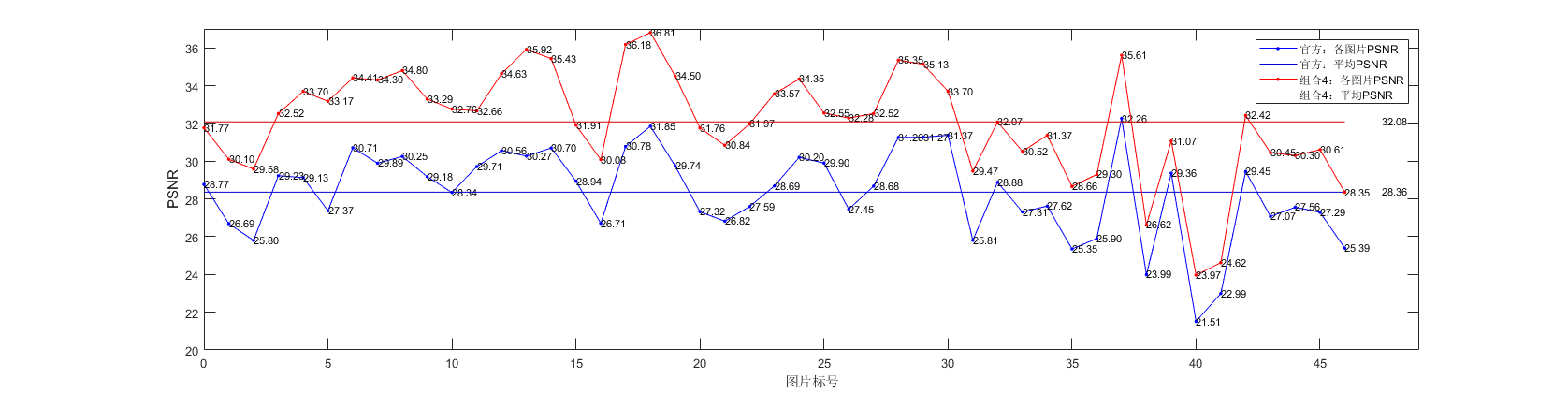


图 1.16 组合4 PSNR性能

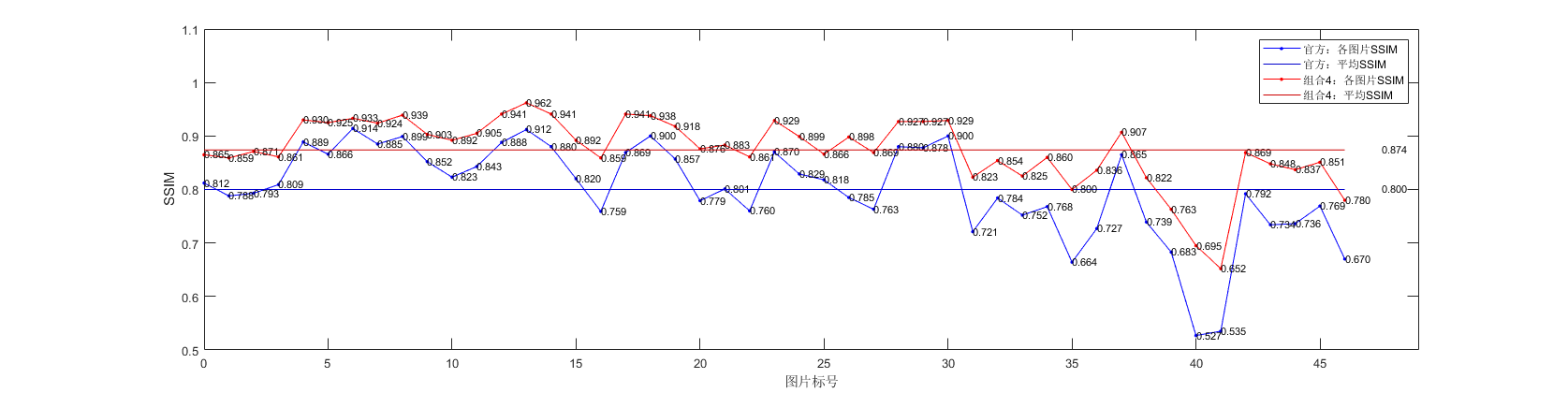


图 1.17 组合4 SSIM性能

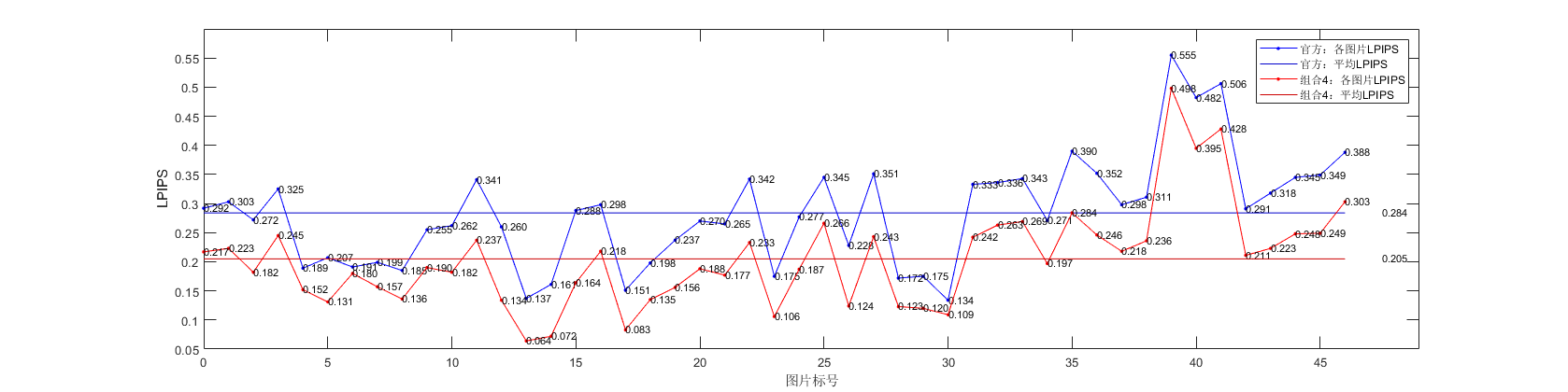


图 1.18 组合4 LPIPS性能

组合5：3区域均采用边缘插值+均值调整+高斯平滑

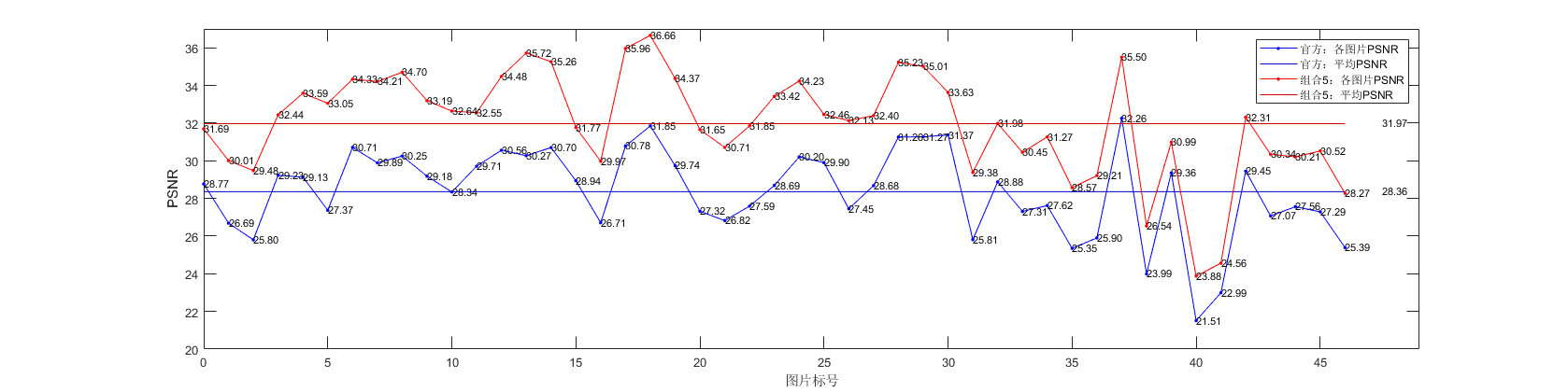


图 1.19 组合5 PSNR性能

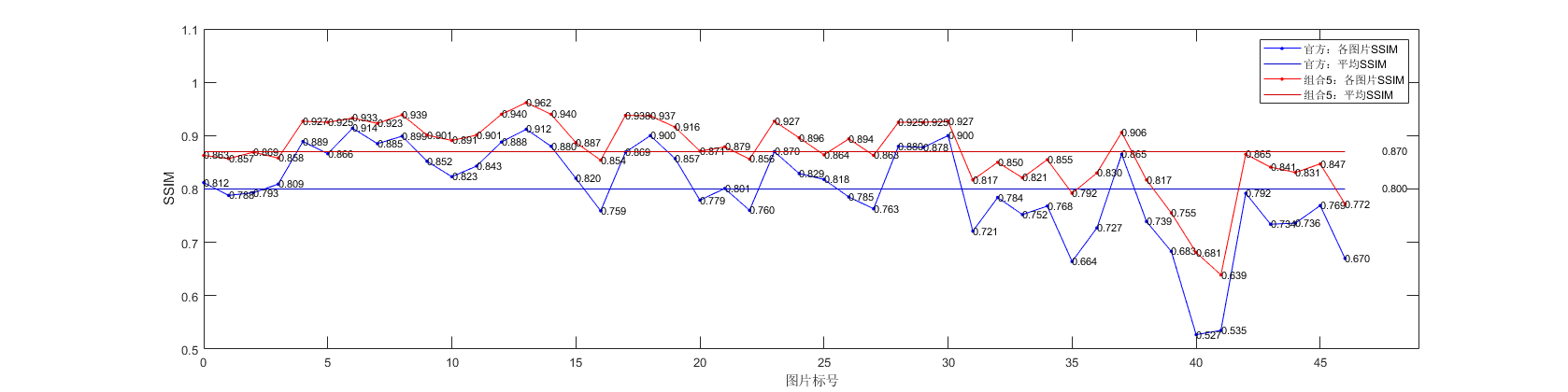


图 1.20 组合5 SSIM性能

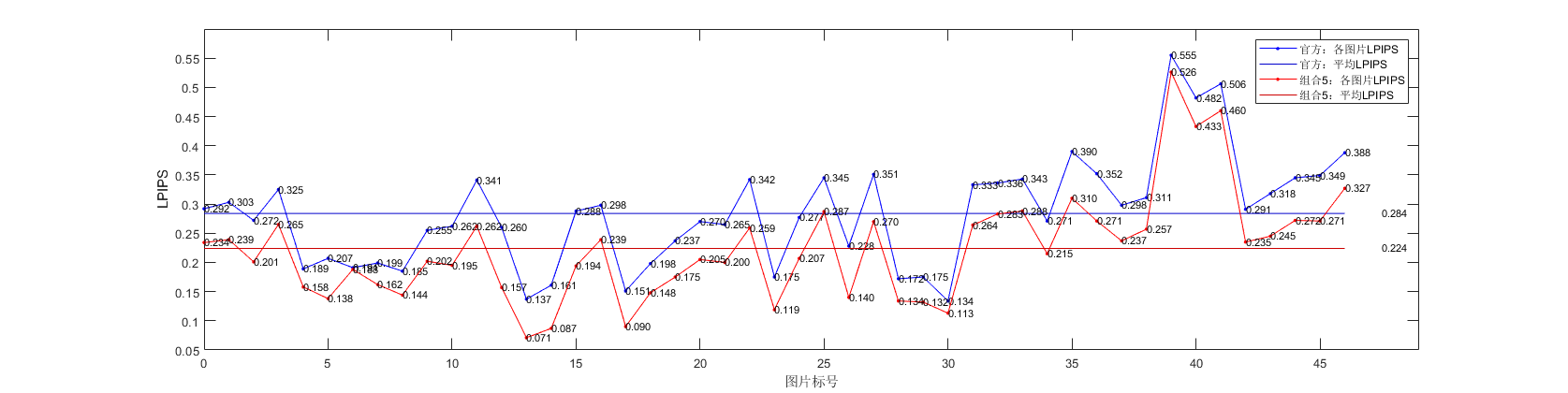


图 1.21 组合5 LPIPS性能

我们分别采用单线程以及RGB独立3线程进行插值，生成4k图片，结果表明两者生成的数据一致，且3线程的总运行时间明显少于单线程，以下是处理23张1k图片单线程与3线程所需要的时间。

|  |  |
| --- | --- |
| 23张1k图片单线程运行总时间 | 23张1k图片三线程运行总时间 |
| 2m53.696s | 1m6.773s |

对于以上各种组合算法，可以看出，不同组合方式具有不同的性能和算法复杂度。其中，组合1复杂度最低且具有相对高的性能，组合3复杂度最高，PSNR与SSIM性能也最优，而组合4在3个区域内均采用边缘插值，相比于组合3复杂度有所下降，而性能基本不变，因此，我们选取组合4作为最终硬件实现的算法。在经过四舍五入后，组合4的平均PSNR=32.08，SSIM=0.874，LPIPS=0.205，相比于官方给的PSNR=28.36，SSIM=0.800，LPIPS=0.284均有着较大幅度的提升，且算法复杂度不高，具有广阔的应用前景。

## 1.2硬件设计框架

### 1.2.1PL端硬件架构

如上文所述，我们采用组合4作为最终硬件实现的算法，即边缘插值+均值调整+高斯平滑+均值调整，从高层次的角度，其可以划分为边缘插值+均值调整与高斯平滑+均值调整两个步骤。

下图1.22为整体的系统架构。其总体流程为首先PS端将SD卡内的1k图片写入到DDR3，当存储完毕后写入1k存储完成标志位，而PL端通过AXI\_HP0接口一直访问1k图片存储完成标志位，若存储完成，PL端开始从DDR3读取1k像素数据，并将经RGB三颜色插值模块生成的4k像素数据写回DDR3。当4k图片数据存储完毕时，PL端会向DDR3中写入4k图片存储完成标志位，而PS端会一直读取4k存储完成标志位的值，若满足要求，则PS端开始从SD卡中读取下一张1k图片，循环往复，直至完成所有图片的插值。当所有图片插值完成后，PS端会通过AXI\_GP0接口实现对VDMA的配置与初始化，并将DDR3中的4k数据通过AXI\_HP1接口输入到VDMA，进而将像素数据输出到外接显示屏上。当我们在PS端对VDMA进行重新配置时，即可改变VDMA的显示内容，进而实现分时显示4k图片的功能。



图 1.22 系统硬件框架图

其中axi\_ddr\_rw用来控制对DDR3的存与取，也即上文说的从DDR3中读取1k图片存储完成标志位，然后开始突发读取1k图片数据并将串行数据转换为并行数据，经插值模块生成4k图片数据后，将生成的4k并行数据转换为串行数据并突发存入DDR3中，接着循环往复读取1k数据存储4k数据，直至整个4k图片数据完成，最后向DDR3写入4k存储完成标志，此时又跳转到读取1k图片存储完成标志位这一状态。而PS端会一直读取4k存储完成标志位，当成功读到时又会将下一张1k图片的数据存入DDR3，开始下一张1k图片的插值。其主要完成图1.23所示状态机的功能：



图 1.23 axi\_ddr\_rw实现功能状态机

为了使得读写操作简便，我们在PS端对1k图片数据进行2次扩展，然后按如下流程将1k数据写入DDR3：

（1）将扩展后1k图片的第1列1至5行写入DDR3中，然后连续三次写入0，以满足能够利用AXI4-full使得突发读取长度为8。

（2）重复（1）的操作将所有列的1至5行写入DDR3中，此时就能求出1k图片的第一行对应的4k图片数据。

（3）将2至6行的所有列按上述规则写数DDR3中，然后依此类推完成整个1k图片数据的写入。

从以上描述中可以看出，该做法会浪费存储资源，但这方式一方面可以利用AXI4-full总线以便加快PS端与PL端的通信，另一方面读取地址也无需进行跳变。

从0计数，当列数大于等于4时，RGB三颜色插值模块开始输出16个有效的4k像素数据，我们按下图所示顺序将生成的4k并行数据转换为串行数据。



图 1.24 生成4k数据传输顺序

从以上描述可知，存入到DDR3的数据并非是最终的4k图片数据，而是乱序的数据，因此需要PS端将数据从DDR3存入SD卡时进行顺序变换。这种传输方式的优点是操作简单，无需进行存储地址的跳变，且能直接采用AXI4-full突发长度16进行数据存储，加快了PS与PL端的通信速率，缺点是需要PS端进行顺序变换。

我们利用vivado2018.3进行整个系统的实现，当按键key按下时，开始从DDR3中读取数据，即所有图片插值过程的开始，当输出done信号拉高时，PL端的LED0变亮，提示所有图片插值过程完成，此时，所有的1k图片均插值为4k图片并保存在SD卡中。以下是系统bd框图：

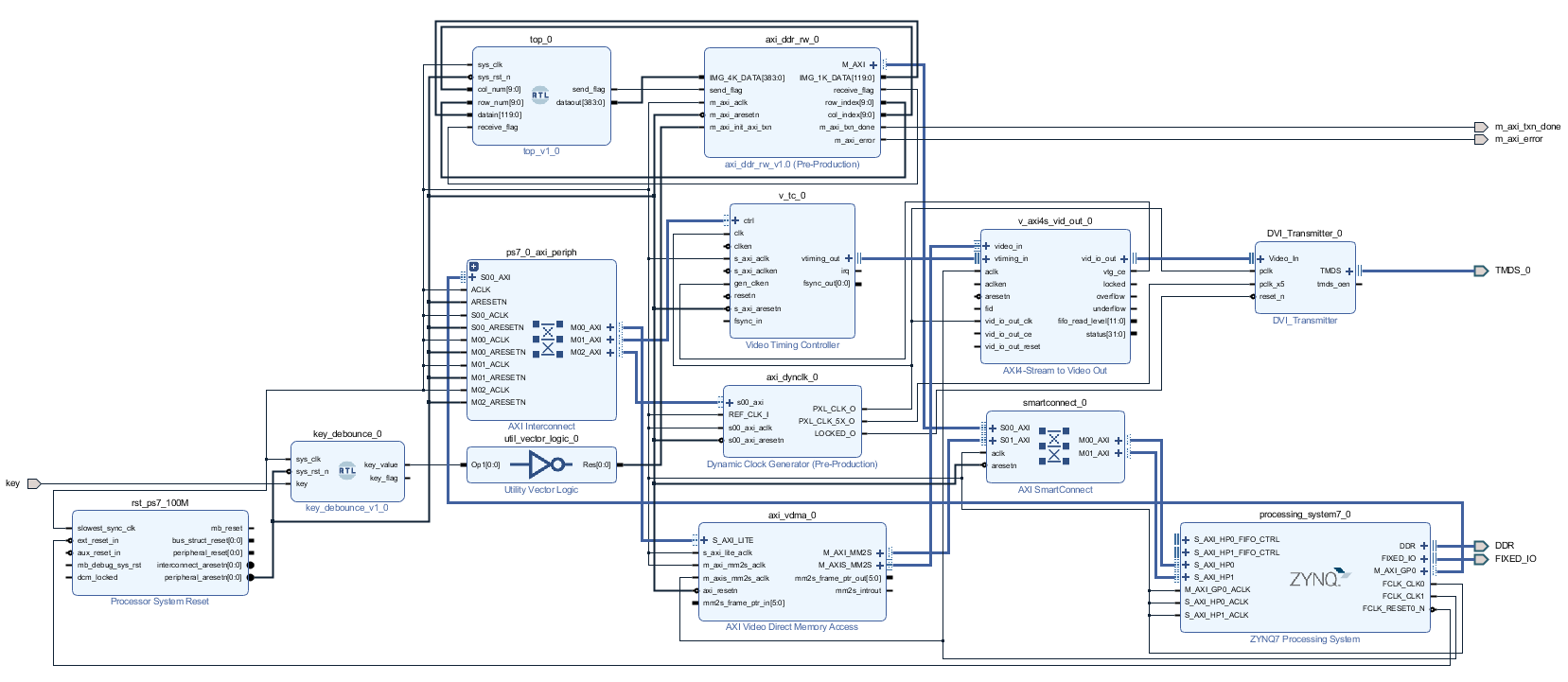


图 1.25 系统bd框图

### 1.2.2单颜色分量插值模块

我们的插值IP采用流的数据输入形式，即模块每次接收到1k图片一列的5个像素点，然后利用当前像素点对前一阶段存储的1k像素点进行边缘插值+均值调整，在此前提下，又可对前前阶段经边缘插值+均值调整得到的4k像素点进行高斯平滑+均值调整，以下介绍插值流程：



图 1.26 插值流程示意图1

上图1.26中，大圆点表示1k图片像素点，小圆点表示4k图片像素点，空心圆点则代表经两次边缘扩展生成的点，其值与最近的1k图片像素点一致。在当前阶段，当接收到1k图片一列的5个像素点时，即图中的红色大圆点以及扩展的大圆点，利用原先存储的1k像素点，我们可对先前一列的中间3个1k像素点进行边缘插值+均值调整，得到图中的红色小圆点。



图 1.27 插值流程示意图2

对于最中间的1k像素点，即图1.27中的蓝色大圆点，我们已经求得其周边的共36个4k像素点，即图中的蓝色小圆点，此时可以对这部分像素点进行高斯平滑+均值调整，得到该1k像素点最终对应的4k图片的16个像素点。由于对1k图片的边界进行了两次扩展，因此只有当第5列数据输入模块时，才能实际输出所需要的结果，当接收到第964列时才实际完成1k图片一行像素的转换，即完成一行的插值需要964个时钟。同理，我们可以求得1k图片其余各行对应的4k图片的像素点，进而求得整个4k图片。

在进行边缘插值+均值调整的过程中，由于我们要对3个1k像素点进行边缘插值+均值调整，每个点需要花费1个时钟，即整个插值模块需要花费3个时钟才能求出一个1k像素点对应的16个4k像素点，在设定100M的时钟下，若仅考虑理想状态，每张图片需要的时钟数为：



因此，1s钟能够处理的图片数为帧。然而，由于我们需要从DDR3中取出1k数据以及存储4k数据，而我们采用的DDR3数据位宽为32bit，每个时钟仅能存取一个像素点，因此，实际处理速率会受到很大的影响，其可以通过异步fifo来提高对DDR3的存取时钟来缓解，本设计中不再讨论。对于最高工作频率而言，由于我们的插值模块并未包含反馈环路，因此其不存在迭代边界，即可通过流水线技术，理论上时钟周期可以任意小，另一方面，在确定电路结构后，工作频率也与其工艺直接相关，不同工艺下也会导致不同的最高工作频率，由于测试时我们的插值模块能满足100M时钟的要求，因此不再讨论。

对于单颜色分量插值模块(single\_color\_top)而言，其功能为接收1k图片一列的5个单颜色分量数据，并利用先前存储的1k像素，实现边缘插值+均值调整+高斯平滑+均值调整的功能，进而求得最终4k图片的单个颜色分量的像素值。根据上文介绍的工作流程，我们的单颜色分量插值模块设计如图1.28所示：



图 1.28 单颜色分量插值模块框架

最终用vivado综合出来的框图为：

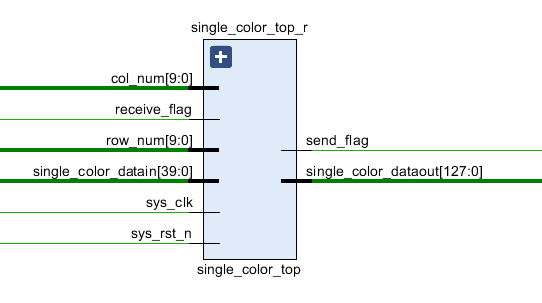


图 1.29 vivado综合出的单颜色分量插值模块框图

其中col\_num表示扩展的1k图片当前输入一列像素的列数，row\_num表示当前处理的1k像素点的行数（此模块中实际没有用到），single\_color\_datain和receive\_flag同步且为单个时钟周期的有效数据，single\_color\_dataout和send\_flag同步且为单个时钟周期的有效数据。

以下是是单颜色分量插值模块内部电路的结构图

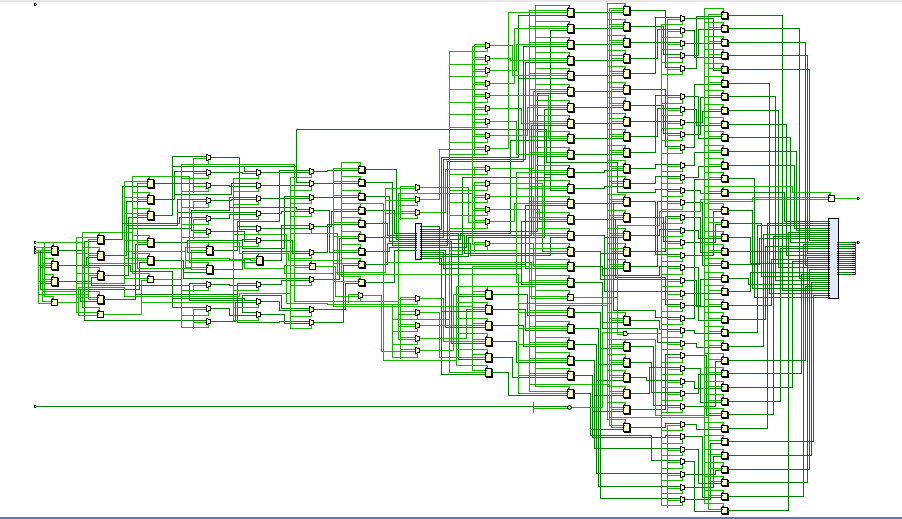


图 1.30 单颜色分量插值模块内部电路图

### 1.2.3边缘插值+均值调整模块和高斯平滑+均值调整模块

由于单颜色分量插值模块中的边缘插值+均值调整和高斯平滑+均值调整均为组合逻辑加上缩短逻辑延时的寄存器，其功能在软件算法介绍部分已有详细的说明，此处仅给出vivado综合出的框图。

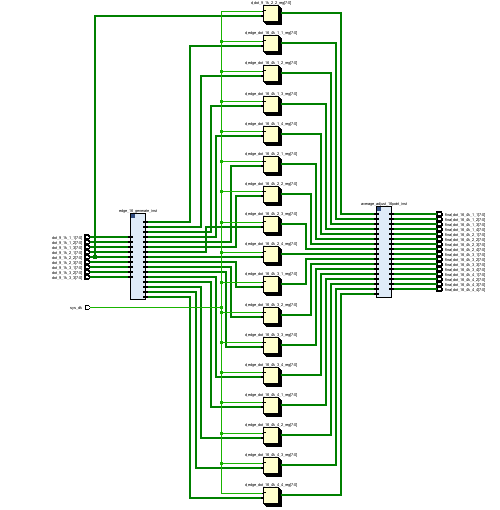


图 1.31 vivado综合出的边缘插值+均值调整模块框图

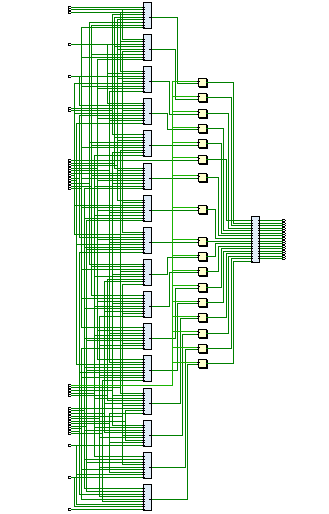


图 1.32 vivado综合出的高斯平滑+均值调整模块框图

### 1.2.4PS端设计

下图1.33展示了PS端将1k数据从SD卡写入DDR3并将生成的4k数据存入SD的关键代码，其流程为将1k像素数据写入DDR3，然后向1k存储完成标志位写入标志，接着一直读取4k图片存储完成标志位，当成功读取到4k存储完成标志位时，将图片从DDR3中读出，并按指定路径存入SD卡，此时判断是否完成了所有图片，若完成了则退出循环，否则开始存储下一张1k图片，其流程与上文所述一致。



图 1.33 PS端main.c关键代码

# 二、实现函数说明

## 2.1头文件

### 2.1.1Upsampling.h

该头文件主要包含结构体，引用的头文件，定义的数据类型以及宏定义。

（1）结构体

BMP文件头(BITMAPFILEHEADER)：含有BMP文件的类型，文件大小和位图起始位置等信息。

BMP信息头(BITMAPINFOHEADER)：用于说明位图的尺寸（宽度，高度，带下，分辨率），像素的位数等信息。

颜色表(RGBQUAD)：用于说明位图中的颜色，有若干个表项，每个表项是一个RGBQUAD类型的结构，定义一种颜色。

位图信息(BITMAOINFO)：位图信息头和颜色表组成位图信息。

（2）引用的库函数

Stdio.h，stdlib.h，malloc.h，string.h，math.h，pthread.h(实现多线程函数)

（3）数据类型

typedef unsigned char BYTE;

typedef unsigned short WORD;

typedef unsigned int DWORD;

（4）宏定义

#define MAXSIZE 4000；

#define REAL\_WIDTH bitInfo->bmiHeader->biWidth；

#define REAL\_HEIGHT bitInfo->bmiHeader->biHeight；

### 2.1.2interpolation.h

该头文件定义了插值函数，分别是

void near\_16\_generate //最近邻插值

void binear\_16\_generate //改进的双线性插值

void edge\_16\_generate //边缘插值

void average\_adjust\_16point //16点的均值调整

void interpolate\_16point //根据sobel算子选择边缘插值，双线性插值，

//最近邻插值

void interpolation //根据输入的25点1K图片像素，得到最终的

//4K图片的16个像素

void final\_interpolation\_x4 //求出4K图像的r,g,b中的一个

## 2.2源文件

### 2.2.1interpolating.c

该源文件包含了实现将1k图片放大到4k图片的插值函数。

（1）near\_16\_generate

void near\_16\_generate(BYTE(\*near\_dot\_16\_4k)[4],BYTE(\*dot\_9\_1k)[3])

函数用途：实现最近邻插值算法。

函数参数：BYTE(\*near\_dot\_16\_4k)[4]：4kBMP图片的16个像素点，BYTE(\*dot\_9\_1k)[3]：1kBMP图片的9个像素点，由1kBMP数据中的9个点生成4kBMP数据中的16个点。

（2）binear\_16\_generate

void binear\_16\_generate(BYTE(\*binear\_dot\_16\_4k)[4], BYTE(\*dot\_9\_1k)[3])

函数用途：实现新型双线性插值算法。

函数参数：BYTE(\*near\_dot\_16\_4k)[4]：4kBMP图片的16个像素点，BYTE(\*dot\_9\_1k)[3]：1kBMP图片的9个像素点，由1kBMP数据中的9个点生成4kBMP数据中的16个点。

（3）edge\_16\_generate

void edge\_16\_generate(BYTE(\*edge\_dot\_16\_4k)[4], BYTE(\*dot\_9\_1k)[3])

函数用途：实现边缘插值算法。

函数参数：BYTE(\*near\_dot\_16\_4k)[4]：4kBMP图片的16个像素点，BYTE(\*dot\_9\_1k)[3]：1kBMP图片的9个像素点，由1kBMP数据中的9个点生成4kBMP数据中的16个点。

（4）average\_adjust\_16point

Void average\_adjust\_16point(BYTE(\*adjust\_dot\_16\_4k)[4],BYTE(\*

dot\_16\_4k)[4],BYTE average)

函数用途：实现均值调整函数。

函数参数：BYTE(\*adjust\_dot\_16\_4k)[4]：由均值调整函数生成的4k图像中的16个像素点, BYTE(\*dot\_16\_4k)[4]：由插值算法生成的4K图像中的16个像素点, BYTE average:九个1k像素点中的中间那个点的像素数据。

（5）interpolate\_16point

void interpolate\_16point(BYTE(\*final\_dot\_16\_4k)[4], BYTE(\*dot\_9\_1k)[3])

函数用途：根据Sobel算子选择边缘插值，双线性插值和最近邻插值。

函数参数：BYTE(\*final\_dot\_16\_4k)[4]：经过该插值函数得到的4k数据的16个像素点, BYTE(\*dot\_9\_1k)[3]：1kBMP图片数据的9个像素点数据。

（6）interpolation

void interpolation(BYTE (\*final\_dot\_16\_4k)[4], BYTE (\*dot\_25\_1k)[5]);

函数用途：根据输入的25点1k图片像素，经16点插值+均值调整+高斯平滑+均值调整得到最终的4k图片的16个像素。

函数参数：BYTE (\*final\_dot\_16\_4k)[4]：最终得到的4k图片中的16个像素点， BYTE (\*dot\_25\_1k)[5]：输入的1k图片的25个像素点。

（7）final\_interpolation\_x4

void final\_interpolation\_x4(BYTE(\*image\_4k\_data)[3840],BYTE

(\*image\_1k\_data)[960])

函数用途：实现最终的改进的插值算法。

函数参数：BYTE(\*image\_4k\_data)[3840]：由最终改进的插值算法得到的4kBMP图像的数据, BYTE(\*image\_1k\_data)[960]：1kBMP图像的数据。

### 2.2.2upsampling.c

该文件含三个BMP图像的处理函数，包括读取，读RGB和保存。

（1）readBMP函数

void readBMP(BITMAPINFO\* bitInfo, char path[])

函数用途：读取BMP图像。

函数参数：需要读取的图像路径，位图信息结构体。

（2）readRGBby24 函数

void readRGBby24(BITMAPINFO\* bitInfo, FILE\* pfile)

函数用途：读真彩BMP图像，将readBMP函数读取的pfile文件中的数据存入位图信息结构体。

函数参数：FILE\* pfile ：readBMP函数读取的文件（1KBMP图像数据）pfile，BITMAPINFO\* bitInfo：位图信息结构体。

（3）savePicture 函数

void savePicture(BITMAPINFO\* bitInfo, char path[])

函数用途：保存BMP图像。

函数参数：char path[]：test.c文件中输入的保存图像时的路径，BITMAPINFO\* bitInfo：位图信息结构体。 函数实现：将BMP图像中的信息写入位图信息结构体中。

### 2.2.3test.c

此源文件含有main函数，采用多线程（R,G,B三个线程）分别处理，最后保存图片到指定路径。

（1）func\_b

void \* func\_b (BITMAPINFO\* bitInfo)

函数用途：对颜色b的数据进行处理。

函数参数：BITMAPINFO\* bitInfo：位图信息结构体。

（2）func\_g

void \* func\_g (BITMAPINFO\* bitInfo)

同func\_b, 此函数对颜色g的数据进行处理。

（3）func\_r

void \* func\_r(BITMAPINFO\* bitInfo)

同func\_b, 此函数对颜色r的数据进行处理。

（4）main函数

函数用途：保存图片

函数参数：位图信息结构体。

函数实现：R,G,B三个线程在main函数中统一保存，最终得到所要的4k图片。

# 三、寄存器说明

由于在我们的设计中仅单颜色分量插值模块内有寄存器，顶层模块中只是例化了该模块三次，因此，以下仅介绍单颜色分量插值模块内的寄存器，即上文中单颜色分量插值模块框图中标注的寄存器。

## 3.1存储1k像素5行3列寄存器

当单颜色分量插值模块接收到并行的5个1k像素数据，该寄存器会向左移动，并在最右端存储接收到的5个1k像素数据，此寄存器的大小为bit。在verilog代码中其定义如下：

reg [7:0] column1\_1k[4:0] ;

reg [7:0] column2\_1k[4:0] ;

reg [7:0] column3\_1k[4:0] ;

## 3.2边缘插值+均值调整所需的9点寄存器

在本设计中，进行边缘插值+均值调整需要1k图片的9个点，存储时共需bit。利用5行3列1k像素寄存器，可以求解出中间列的中间3个点经边缘插值+均值调整后的4k像素点，为了节省计算资源，我们每个时钟从5行3列1k像素寄存器中取出9个点，经3个时钟即可完成了所需要计算的3个1k像素点，从而达到分时计算的目的。由于边缘插值+均值调整模块所需要的计算资源远大于存储9个点需要的存储资源，且分时计算理论性能仍能满足每秒60帧的帧率，因此，这样做是十分有利的。在verilog中其定义如下：

reg [7:0] dot\_9\_1k\_1\_1 ;

reg [7:0] dot\_9\_1k\_1\_2 ;

reg [7:0] dot\_9\_1k\_1\_3 ;

reg [7:0] dot\_9\_1k\_2\_1 ;

reg [7:0] dot\_9\_1k\_2\_2 ;

reg [7:0] dot\_9\_1k\_2\_3 ;

reg [7:0] dot\_9\_1k\_3\_1 ;

reg [7:0] dot\_9\_1k\_3\_2 ;

reg [7:0] dot\_9\_1k\_3\_3 ;

## 3.3边缘插值+均值调整模块内17点寄存器

为了缩短逻辑延时，我们在边缘插值模块以及均值调整模块之间插入了无复位的17点寄存器，存储边缘插值生成的16个4k像素点以及一个1k像素点，总共需要存储的比特数为bit。在verilog中其定义如下：

reg [7:0] d\_edge\_dot\_16\_4k\_1\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_4 ;

reg [7:0] d\_dot\_9\_1k\_2\_2 ;

## 3.4边缘插值+均值调整后的4k图片6行9列寄存器

为了进行高斯平滑+均值调整，我们需要得到1k像素点对应的经边缘插值+均值调整后的16个4k像素点以及其周边的一圈，共需个点。在当前阶段能产生6行4列经边缘插值+均值调整后的4k像素点，此时该寄存器向左移动4列，并存储当前列生成的6行4列数据，这时已经拥有了前一阶段进行高斯平滑+均值调整所需的36像素点，总共需要存储的比特数为bit。在verilog中其定义如下：

reg [7:0] column1\_4k[5:0] ;

reg [7:0] column2\_4k[5:0] ;

reg [7:0] column3\_4k[5:0] ;

reg [7:0] column4\_4k[5:0] ;

reg [7:0] column5\_4k[5:0] ;

reg [7:0] column6\_4k[5:0] ;

reg [7:0] column7\_4k[5:0] ;

reg [7:0] column8\_4k[5:0] ;

reg [7:0] column9\_4k[5:0] ;

## 3.5高斯平滑+均值调整所需的36点寄存器

对1k图片中一点经边缘插值+均值调整得到的16点，若对其进行高斯平滑+均值调整，则需要一共包含36像素点，此处的寄存器就是为了从上文中6行9列寄存器中取出36点，即前6列数据，一共需要存储bit。在verilog中其定义如下：

reg [7:0] column1\_gauss\_4k[5:0] ;

reg [7:0] column2\_gauss\_4k[5:0] ;

reg [7:0] column3\_gauss\_4k[5:0] ;

reg [7:0] column4\_gauss\_4k[5:0] ;

reg [7:0] column5\_gauss\_4k[5:0] ;

reg [7:0] column6\_gauss\_4k[5:0] ;

## 3.6高斯平滑+均值调整模块内17点寄存器

为了缩短逻辑延时，我们在高斯平滑模块以及均值调整模块之间插入了无复位的17点寄存器，存储高斯平滑生成的16个4k像素点以及一个1k像素点，总共需要存储的比特数为bit。在verilog中其定义如下：

reg [7:0] d\_4k\_pixel\_1\_1 ;

reg [7:0] d\_4k\_pixel\_1\_2 ;

reg [7:0] d\_4k\_pixel\_1\_3 ;

reg [7:0] d\_4k\_pixel\_1\_4 ;

reg [7:0] d\_4k\_pixel\_2\_1 ;

reg [7:0] d\_4k\_pixel\_2\_2 ;

reg [7:0] d\_4k\_pixel\_2\_3 ;

reg [7:0] d\_4k\_pixel\_2\_4 ;

reg [7:0] d\_4k\_pixel\_3\_1 ;

reg [7:0] d\_4k\_pixel\_3\_2 ;

reg [7:0] d\_4k\_pixel\_3\_3 ;

reg [7:0] d\_4k\_pixel\_3\_4 ;

reg [7:0] d\_4k\_pixel\_4\_1 ;

reg [7:0] d\_4k\_pixel\_4\_2 ;

reg [7:0] d\_4k\_pixel\_4\_3 ;

reg [7:0] d\_4k\_pixel\_4\_4 ;

reg [7:0] d\_average ;

# 四、RTL模块设计说明

## 4.1top

### 4.1.1模块框图

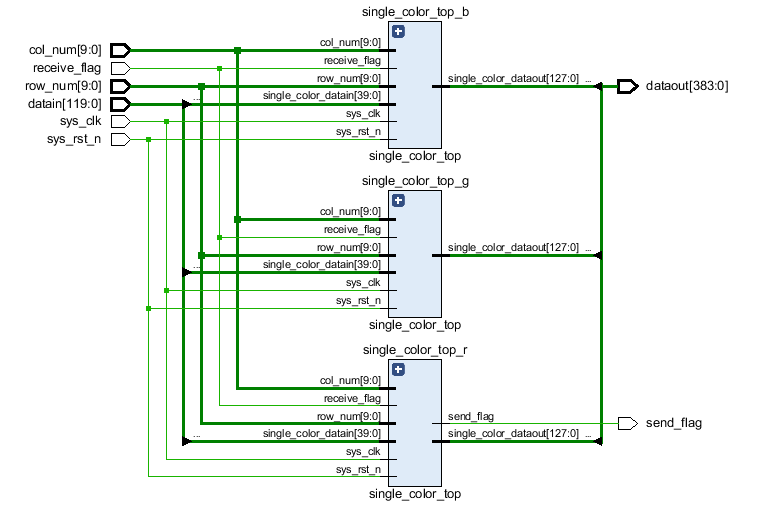


图 4.1 top模块框图

### 4.1.2连线端口

(1)I/O端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| sys\_clk | I | 1 | 系统时钟 |
| sys\_rst\_n | I | 1 | 系统异步复位 |
| col\_num | I | 10 | 扩展1k图片列数 |
| row\_num | I | 10 | 扩展1k图片行数 |
| datain | I | 120 | 输入1k像素数据 |
| receive\_flag | I | 1 | 1k像素数据接收标志位 |
| dataout | O | 384 | 输出4k像素数据 |
| send\_flag | O | 1 | 4k像素数据输出标志位 |

(2)中间变量

//取出输入各颜色分量传入单颜色插值模块

wire [5\*8-1:0] datain\_b;

wire [5\*8-1:0] datain\_g;

wire [5\*8-1:0] datain\_r;

//单颜色插值模块输出

wire [16\*8-1:0] dataout\_b;

wire [16\*8-1:0] dataout\_g;

wire [16\*8-1:0] dataout\_r;

### 4.1.3模块功能

将输入1k像素数据按颜色分量拆分，然后送至单颜色分量插值模块进行插值，得到最终的4k像素数据，然后又将4k像素数据拼接，传送至dataout端口。

## 4.2single\_color\_top

### 4.2.1模块框图

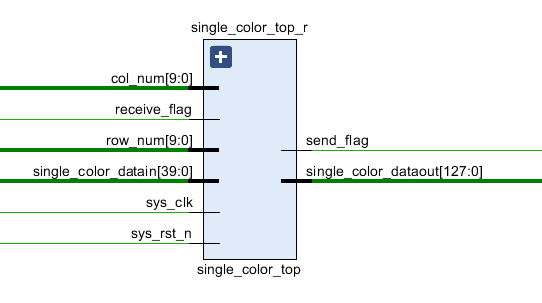


图 4.2 single\_color\_top模块框图

### 4.2.2连线端口

(1)I/O端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| sys\_clk | I | 1 | 系统时钟 |
| sys\_rst\_n | I | 1 | 系统异步复位 |
| col\_num | I | 10 | 扩展1k图片列数 |
| row\_num | I | 10 | 扩展1k图片行数 |
| datain | I | 40 | 输入1k像素单颜色数据 |
| receive\_flag | I | 1 | 1k像素数据接收标志位 |
| dataout | O | 128 | 输出4k像素单颜色数据 |
| send\_flag | O | 1 | 4k像素数据输出标志位 |

(2)中间变量

//将输入的5个像素数据改为数组形式

wire [7:0] pixel[4:0];

//存储所需要的5行3列1k像素数据

reg [7:0] column1\_1k[4:0];

reg [7:0] column2\_1k[4:0];

reg [7:0] column3\_1k[4:0];

//5行3列存储完成，可以开始边缘插值标志

reg interpolate\_start\_flag1 ;

reg interpolate\_start\_flag2 ;

reg interpolate\_start\_flag3 ;

//取出需要进行边缘插值的1k图片的9个点

reg [7:0] dot\_9\_1k\_1\_1 ;

reg [7:0] dot\_9\_1k\_1\_2 ;

reg [7:0] dot\_9\_1k\_1\_3 ;

reg [7:0] dot\_9\_1k\_2\_1 ;

reg [7:0] dot\_9\_1k\_2\_2 ;

reg [7:0] dot\_9\_1k\_2\_3 ;

reg [7:0] dot\_9\_1k\_3\_1 ;

reg [7:0] dot\_9\_1k\_3\_2 ;

reg [7:0] dot\_9\_1k\_3\_3 ;

//边缘插值+均值调整后的4k图片的像素值

wire [7:0] final\_dot\_16\_4k\_1\_1 ;

wire [7:0] final\_dot\_16\_4k\_1\_2 ;

wire [7:0] final\_dot\_16\_4k\_1\_3 ;

wire [7:0] final\_dot\_16\_4k\_1\_4 ;

wire [7:0] final\_dot\_16\_4k\_2\_1 ;

wire [7:0] final\_dot\_16\_4k\_2\_2 ;

wire [7:0] final\_dot\_16\_4k\_2\_3 ;

wire [7:0] final\_dot\_16\_4k\_2\_4 ;

wire [7:0] final\_dot\_16\_4k\_3\_1 ;

wire [7:0] final\_dot\_16\_4k\_3\_2 ;

wire [7:0] final\_dot\_16\_4k\_3\_3 ;

wire [7:0] final\_dot\_16\_4k\_3\_4 ;

wire [7:0] final\_dot\_16\_4k\_4\_1 ;

wire [7:0] final\_dot\_16\_4k\_4\_2 ;

wire [7:0] final\_dot\_16\_4k\_4\_3 ;

wire [7:0] final\_dot\_16\_4k\_4\_4 ;

//将由边缘插值+均值调整的4k像素寄存标志位

wire save\_flag1 ;

wire save\_flag2 ;

reg save\_flag3 ;

//存储需要进行高斯平滑的6行9列数据

reg [7:0] column1\_4k[5:0] ;

reg [7:0] column2\_4k[5:0] ;

reg [7:0] column3\_4k[5:0] ;

reg [7:0] column4\_4k[5:0] ;

reg [7:0] column5\_4k[5:0] ;

reg [7:0] column6\_4k[5:0] ;

reg [7:0] column7\_4k[5:0] ;

reg [7:0] column8\_4k[5:0] ;

reg [7:0] column9\_4k[5:0] ;

//取出需要进行高斯平滑的36点标志位

reg gauss\_flag ;

//取出进行高斯平滑的36个数据

reg [7:0] column1\_gauss\_4k[5:0] ;

reg [7:0] column2\_gauss\_4k[5:0] ;

reg [7:0] column3\_gauss\_4k[5:0] ;

reg [7:0] column4\_gauss\_4k[5:0] ;

reg [7:0] column5\_gauss\_4k[5:0] ;

reg [7:0] column6\_gauss\_4k[5:0] ;

//平均值

reg [7:0] average ;

### 4.2.3模块功能

按流的形式对1k单颜色分量数据进行边缘插值+均值调整+高斯平滑+均值调整，得到最终的4k像素数据。

## 4.3interpolate\_16point

### 4.3.1模块框图

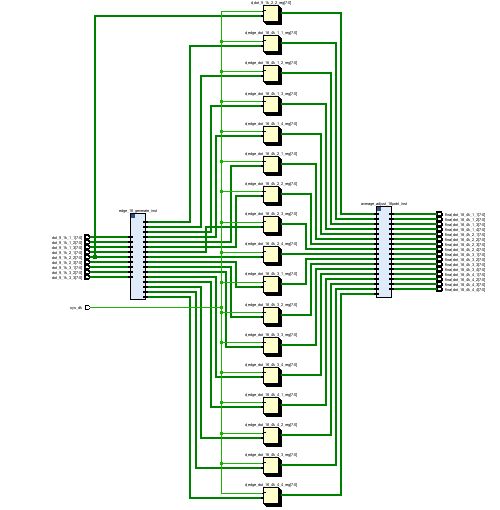


图 4.3 interpolate\_16point模块框图

### 4.3.2连线端口

(1)I/O端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| dot\_9\_1k\_1\_1 | I | 8 | 输入1kbmp像素点（1,1） |
| dot\_9\_1k\_1\_2 | I | 8 | 输入1kbmp像素点（1,2） |
| dot\_9\_1k\_1\_3 | I | 8 | 输入1kbmp像素点（1,3） |
| dot\_9\_1k\_2\_1 | I | 8 | 输入1kbmp像素点（2,1） |
| dot\_9\_1k\_2\_2 | I | 8 | 输入1kbmp像素点（2,2） |
| dot\_9\_1k\_2\_3 | I | 8 | 输入1kbmp像素点（2,3） |
| dot\_9\_1k\_3\_1 | I | 8 | 输入1kbmp像素点（3,1） |
| dot\_9\_1k\_3\_2 | I | 8 | 输入1kbmp像素点（3,2） |
| dot\_9\_1k\_3\_3 | I | 8 | 输入1kbmp像素点（3,3） |
| final\_dot\_16\_4k\_1\_1 | O | 8 | 输出4kbmp像素值（1,1） |
| final\_dot\_16\_4k\_1\_2 | O | 8 | 输出4kbmp像素值（1,2） |
| final\_dot\_16\_4k\_1\_3 | O | 8 | 输出4kbmp像素值（1,3） |
| final\_dot\_16\_4k\_1\_4 | O | 8 | 输出4kbmp像素值（1,4） |
| final\_dot\_16\_4k\_2\_1 | O | 8 | 输出4kbmp像素值（2,1） |
| final\_dot\_16\_4k\_2\_2 | O | 8 | 输出4kbmp像素值（2,2） |
| final\_dot\_16\_4k\_2\_3 | O | 8 | 输出4kbmp像素值（2,3） |
| final\_dot\_16\_4k\_2\_4 | O | 8 | 输出4kbmp像素值（2,4） |
| final\_dot\_16\_4k\_3\_1 | O | 8 | 输出4kbmp像素值（3,1） |
| final\_dot\_16\_4k\_3\_2 | O | 8 | 输出4kbmp像素值（3,2） |
| final\_dot\_16\_4k\_3\_3 | O | 8 | 输出4kbmp像素值（3,3） |
| final\_dot\_16\_4k\_3\_4 | O | 8 | 输出4kbmp像素值（3,4） |
| final\_dot\_16\_4k\_4\_1 | O | 8 | 输出4kbmp像素值（4,1） |
| final\_dot\_16\_4k\_4\_2 | O | 8 | 输出4kbmp像素值（4,2） |
| final\_dot\_16\_4k\_4\_3 | O | 8 | 输出4kbmp像素值（4,3） |
| final\_dot\_16\_4k\_4\_4 | O | 8 | 输出4kbmp像素值（4,4） |

(2)中间变量

//边缘插值输出

wire [7:0] edge\_dot\_16\_4k\_1\_1;

wire [7:0] edge\_dot\_16\_4k\_1\_2;

wire [7:0] edge\_dot\_16\_4k\_1\_3;

wire [7:0] edge\_dot\_16\_4k\_1\_4;

wire [7:0] edge\_dot\_16\_4k\_2\_1;

wire [7:0] edge\_dot\_16\_4k\_2\_2；

wire [7:0] edge\_dot\_16\_4k\_2\_3 ;

wire [7:0] edge\_dot\_16\_4k\_2\_4 ;

wire [7:0] edge\_dot\_16\_4k\_3\_1;

wire [7:0] edge\_dot\_16\_4k\_3\_2;

wire [7:0] edge\_dot\_16\_4k\_3\_3;

wire [7:0] edge\_dot\_16\_4k\_3\_4;

wire [7:0] edge\_dot\_16\_4k\_4\_1;

wire [7:0] edge\_dot\_16\_4k\_4\_2;

wire [7:0] edge\_dot\_16\_4k\_4\_3;

wire [7:0] edge\_dot\_16\_4k\_4\_4;

//寄存器缩短关键路径

reg [7:0] d\_edge\_dot\_16\_4k\_1\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_1\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_2\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_3\_4 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_1 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_2 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_3 ;

reg [7:0] d\_edge\_dot\_16\_4k\_4\_4 ;

reg [7:0] d\_dot\_9\_1k\_2\_2 ;

### 4.3.3模块功能

利用1k图片的9个像素点进行边缘插值+均值调整得到4k图片16个像素点。

## 4.4gauss\_filter\_dot\_36\_4k

### 4.4.1模块框图

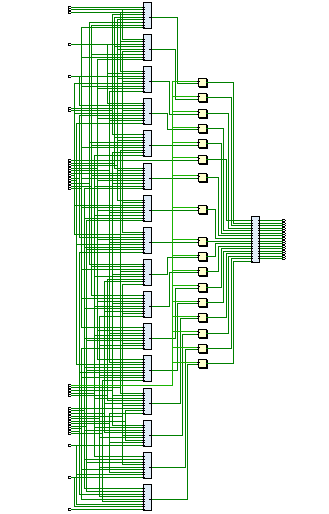


图 4.4 gauss\_filter\_dot\_36\_4k模块框图

### 4.4.2连线端口

(1)I/O端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| dot\_36\_4k\_1\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（1,1） |
| dot\_36\_4k\_1\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（1,2） |
| dot\_36\_4k\_1\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（1,3） |
| dot\_36\_4k\_1\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（1,4） |
| dot\_36\_4k\_1\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（1,5） |
| dot\_36\_4k\_1\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（1,6） |
| dot\_36\_4k\_2\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（2,1） |
| dot\_36\_4k\_2\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（2,2） |
| dot\_36\_4k\_2\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（2,3） |
| dot\_36\_4k\_2\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（2,4） |
| dot\_36\_4k\_2\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（2,5） |
| dot\_36\_4k\_2\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（2,6） |
| dot\_36\_4k\_3\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（3,1） |
| dot\_36\_4k\_3\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（3,2） |
| dot\_36\_4k\_3\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（3,3） |
| dot\_36\_4k\_3\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（3,4） |
| dot\_36\_4k\_3\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（3,5） |
| dot\_36\_4k\_3\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（3,6） |
| dot\_36\_4k\_4\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（4,1） |
| dot\_36\_4k\_4\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（4,2） |
| dot\_36\_4k\_4\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（4,3） |
| dot\_36\_4k\_4\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（4,4） |
| dot\_36\_4k\_4\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（4,5） |
| dot\_36\_4k\_4\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（4,6） |
| dot\_36\_4k\_5\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（5,1） |
| dot\_36\_4k\_5\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（5,2） |
| dot\_36\_4k\_5\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（5,3） |
| dot\_36\_4k\_5\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（5,4） |
| dot\_36\_4k\_5\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（5,5） |
| dot\_36\_4k\_5\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（5,6） |
| dot\_36\_4k\_6\_1 | I | 8 | 边缘插值+均值调整后的4k像素点（6,1） |
| dot\_36\_4k\_6\_2 | I | 8 | 边缘插值+均值调整后的4k像素点（6,2） |
| dot\_36\_4k\_6\_3 | I | 8 | 边缘插值+均值调整后的4k像素点（6,3） |
| dot\_36\_4k\_6\_4 | I | 8 | 边缘插值+均值调整后的4k像素点（6,4） |
| dot\_36\_4k\_6\_5 | I | 8 | 边缘插值+均值调整后的4k像素点（6,5） |
| dot\_36\_4k\_6\_6 | I | 8 | 边缘插值+均值调整后的4k像素点（6,6） |
| average | I | 8 | 1k像素点平均值 |
| adjust\_dot\_16\_4k\_1\_1 | O | 8 | 高斯平滑+均值调整后的4k像素点（1,1） |
| adjust\_dot\_16\_4k\_1\_2 | O | 8 | 高斯平滑+均值调整后的4k像素点（1,2） |
| adjust\_dot\_16\_4k\_1\_3 | O | 8 | 高斯平滑+均值调整后的4k像素点（1,3） |
| adjust\_dot\_16\_4k\_1\_4 | O | 8 | 高斯平滑+均值调整后的4k像素点（1,4） |
| adjust\_dot\_16\_4k\_2\_1 | O | 8 | 高斯平滑+均值调整后的4k像素点（2,1） |
| adjust\_dot\_16\_4k\_2\_2 | O | 8 | 高斯平滑+均值调整后的4k像素点（2,2） |
| adjust\_dot\_16\_4k\_2\_3 | O | 8 | 高斯平滑+均值调整后的4k像素点（2,3） |
| adjust\_dot\_16\_4k\_2\_4 | O | 8 | 高斯平滑+均值调整后的4k像素点（2,4） |
| adjust\_dot\_16\_4k\_3\_1 | O | 8 | 高斯平滑+均值调整后的4k像素点（3,1） |
| adjust\_dot\_16\_4k\_3\_2 | O | 8 | 高斯平滑+均值调整后的4k像素点（3,2） |
| adjust\_dot\_16\_4k\_3\_3 | O | 8 | 高斯平滑+均值调整后的4k像素点（3,3） |
| adjust\_dot\_16\_4k\_3\_4 | O | 8 | 高斯平滑+均值调整后的4k像素点（3,4） |
| adjust\_dot\_16\_4k\_4\_1 | O | 8 | 高斯平滑+均值调整后的4k像素点（4,1） |
| adjust\_dot\_16\_4k\_4\_2 | O | 8 | 高斯平滑+均值调整后的4k像素点（4,2） |
| adjust\_dot\_16\_4k\_4\_3 | O | 8 | 高斯平滑+均值调整后的4k像素点（4,3） |
| adjust\_dot\_16\_4k\_4\_4 | O | 8 | 高斯平滑+均值调整后的4k像素点（4,4） |

(2)中间变量

//高斯平滑输出

wire [7:0] \_4k\_pixel\_1\_1 ;

wire [7:0] \_4k\_pixel\_1\_2 ;

wire [7:0] \_4k\_pixel\_1\_3 ;

wire [7:0] \_4k\_pixel\_1\_4 ;

wire [7:0] \_4k\_pixel\_2\_1 ;

wire [7:0] \_4k\_pixel\_2\_2 ;

wire [7:0] \_4k\_pixel\_2\_3 ;

wire [7:0] \_4k\_pixel\_2\_4 ;

wire [7:0] \_4k\_pixel\_3\_1 ;

wire [7:0] \_4k\_pixel\_3\_2 ;

wire [7:0] \_4k\_pixel\_3\_3 ;

wire [7:0] \_4k\_pixel\_3\_4 ;

wire [7:0] \_4k\_pixel\_4\_1 ;

wire [7:0] \_4k\_pixel\_4\_2 ;

wire [7:0] \_4k\_pixel\_4\_3 ;

wire [7:0] \_4k\_pixel\_4\_4 ;

//寄存器缩短关键路径

reg [7:0] d\_4k\_pixel\_1\_1 ;

reg [7:0] d\_4k\_pixel\_1\_2 ;

reg [7:0] d\_4k\_pixel\_1\_3 ;

reg [7:0] d\_4k\_pixel\_1\_4 ;

reg [7:0] d\_4k\_pixel\_2\_1 ;

reg [7:0] d\_4k\_pixel\_2\_2 ;

reg [7:0] d\_4k\_pixel\_2\_3 ;

reg [7:0] d\_4k\_pixel\_2\_4 ;

reg [7:0] d\_4k\_pixel\_3\_1 ;

reg [7:0] d\_4k\_pixel\_3\_2 ;

reg [7:0] d\_4k\_pixel\_3\_3 ;

reg [7:0] d\_4k\_pixel\_3\_4 ;

reg [7:0] d\_4k\_pixel\_4\_1 ;

reg [7:0] d\_4k\_pixel\_4\_2 ;

reg [7:0] d\_4k\_pixel\_4\_3 ;

reg [7:0] d\_4k\_pixel\_4\_4 ;

reg [7:0] d\_average ;

### 4.4.3模块功能

对边缘插值+均值调整后得到的4k像素数据进行高斯平滑+均值调整，得到最终的4k像素数据。

## 4.5edge\_16\_generate

### 4.5.1模块框图

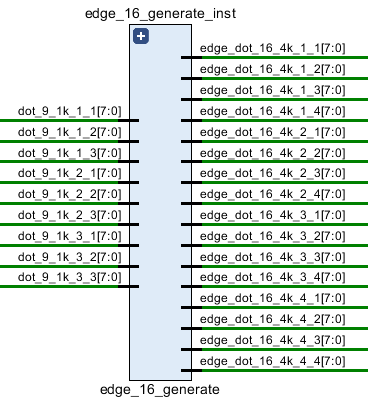


图 4.5 edge\_16\_generate k模块框图

### 4.5.2连线端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| dot\_9\_1k\_1\_1 | I | 8 | 输入1kbmp像素点（1,1） |
| dot\_9\_1k\_1\_2 | I | 8 | 输入1kbmp像素点（1,2） |
| dot\_9\_1k\_1\_3 | I | 8 | 输入1kbmp像素点（1,3） |
| dot\_9\_1k\_2\_1 | I | 8 | 输入1kbmp像素点（2,1） |
| dot\_9\_1k\_2\_2 | I | 8 | 输入1kbmp像素点（2,2） |
| dot\_9\_1k\_2\_3 | I | 8 | 输入1kbmp像素点（2,3） |
| dot\_9\_1k\_3\_1 | I | 8 | 输入1kbmp像素点（3,1） |
| dot\_9\_1k\_3\_2 | I | 8 | 输入1kbmp像素点（3,2） |
| dot\_9\_1k\_3\_3 | I | 8 | 输入1kbmp像素点（3,3） |
| edge\_dot\_16\_4k\_1\_1 | O | 8 | 输出4kbmp像素值（1,1） |
| edge \_dot\_16\_4k\_1\_2 | O | 8 | 输出4kbmp像素值（1,2） |
| edge \_dot\_16\_4k\_1\_3 | O | 8 | 输出4kbmp像素值（1,3） |
| edge \_dot\_16\_4k\_1\_4 | O | 8 | 输出4kbmp像素值（1,4） |
| edge \_dot\_16\_4k\_2\_1 | O | 8 | 输出4kbmp像素值（2,1） |
| edge \_dot\_16\_4k\_2\_2 | O | 8 | 输出4kbmp像素值（2,2） |
| edge \_dot\_16\_4k\_2\_3 | O | 8 | 输出4kbmp像素值（2,2） |
| edge \_dot\_16\_4k\_2\_4 | O | 8 | 输出4kbmp像素值（2,4） |
| edge \_dot\_16\_4k\_3\_1 | O | 8 | 输出4kbmp像素值（3,1） |
| edge \_dot\_16\_4k\_3\_2 | O | 8 | 输出4kbmp像素值（3,2） |
| edge \_dot\_16\_4k\_3\_3 | O | 8 | 输出4kbmp像素值（3,3） |
| edge \_dot\_16\_4k\_3\_4 | O | 8 | 输出4kbmp像素值（3,4） |
| edge \_dot\_16\_4k\_4\_1 | O | 8 | 输出4kbmp像素值（4,1） |
| edge \_dot\_16\_4k\_4\_2 | O | 8 | 输出4kbmp像素值（4,2） |
| edge \_dot\_16\_4k\_4\_3 | O | 8 | 输出4kbmp像素值（4,3） |
| edge \_dot\_16\_4k\_4\_4 | O | 8 | 输出4kbmp像素值（4,4） |

### 4.5.3模块功能

利用1k图片的3行3列9个点实现边缘插值，生成16个点。

## 4.6average\_adjust\_16point

### 4.6.1模块框图

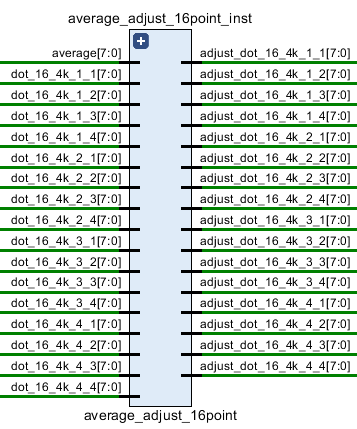


图 4.6 average\_adjust\_16point模块框图

### 4.6.2连线端口

|  |  |  |  |
| --- | --- | --- | --- |
| Port name | I/O | Width/bit | Function |
| dot\_16\_4k\_1\_1 | I | 8 | 输入4kbmp像素值（1,1） |
| dot\_16\_4k\_1\_2 | I | 8 | 输入4kbmp像素值（1,2） |
| dot\_16\_4k\_1\_3 | I | 8 | 输入4kbmp像素值（1,3） |
| dot\_16\_4k\_1\_4 | I | 8 | 输入4kbmp像素值（1,4） |
| dot\_16\_4k\_2\_1 | I | 8 | 输入4kbmp像素值（2,1） |
| dot\_16\_4k\_2\_2 | I | 8 | 输入4kbmp像素值（2,2） |
| dot\_16\_4k\_2\_3 | I | 8 | 输入4kbmp像素值（2,3） |
| dot\_16\_4k\_2\_4 | I | 8 | 输入4kbmp像素值（2,4） |
| dot\_16\_4k\_3\_1 | I | 8 | 输入4kbmp像素值（3,1） |
| dot\_16\_4k\_3\_2 | I | 8 | 输入4kbmp像素值（3,2） |
| dot\_16\_4k\_3\_3 | I | 8 | 输入4kbmp像素值（3,3） |
| dot\_16\_4k\_3\_4 | I | 8 | 输入4kbmp像素值（3,4） |
| dot\_16\_4k\_4\_1 | I | 8 | 输入4kbmp像素值（4,1） |
| dot\_16\_4k\_4\_2 | I | 8 | 输入4kbmp像素值（4,2） |
| dot\_16\_4k\_4\_3 | I | 8 | 输入4kbmp像素值（4,3） |
| dot\_16\_4k\_4\_4 | I | 8 | 输入4kbmp像素值（4,4） |
| average | I | 8 | 平均值 |
| adjust\_dot\_16\_4k\_1\_1 | O | 8 | 输出均值调整后的4kbmp像素值（1,1） |
| adjust\_dot\_16\_4k\_1\_2 | O | 8 | 输出均值调整后的4kbmp像素值（1,2） |
| adjust\_dot\_16\_4k\_1\_3 | O | 8 | 输出均值调整后的4kbmp像素值（1,3） |
| adjust\_dot\_16\_4k\_1\_4 | O | 8 | 输出均值调整后的4kbmp像素值（1,4） |
| adjust\_dot\_16\_4k\_2\_1 | O | 8 | 输出均值调整后的4kbmp像素值（2,1） |
| adjust\_dot\_16\_4k\_2\_2 | O | 8 | 输出均值调整后的4kbmp像素值（2,2） |
| adjust\_dot\_16\_4k\_2\_3 | O | 8 | 输出均值调整后的4kbmp像素值（2,3） |
| adjust\_dot\_16\_4k\_2\_4 | O | 8 | 输出均值调整后的4kbmp像素值（2,4） |
| adjust\_dot\_16\_4k\_3\_1 | O | 8 | 输出均值调整后的4kbmp像素值（3,1） |
| adjust\_dot\_16\_4k\_3\_2 | O | 8 | 输出均值调整后的4kbmp像素值（3,2） |
| adjust\_dot\_16\_4k\_3\_3 | O | 8 | 输出均值调整后的4kbmp像素值（3,3） |
| adjust\_dot\_16\_4k\_3\_4 | O | 8 | 输出均值调整后的4kbmp像素值（3,4） |
| adjust\_dot\_16\_4k\_4\_1 | O | 8 | 输出均值调整后的4kbmp像素值（4,1） |
| adjust\_dot\_16\_4k\_4\_2 | O | 8 | 输出均值调整后的4kbmp像素值（4,2） |
| adjust\_dot\_16\_4k\_4\_3 | O | 8 | 输出均值调整后的4kbmp像素值（4,3） |
| adjust\_dot\_16\_4k\_4\_4 | O | 8 | 输出均值调整后的4kbmp像素值（4,4） |

### 4.6.3模块功能

对4k像素点进行均值调整操作。

# 五、仿真验证环境及说明

## 5.1仿真流程说明

为了验证图像插值IP的正确性与否，我们在Vivado2018.3上编写仿真文件tb\_single\_color\_top，并利用vivado自带的仿真器进行仿真。为了效仿实际整个系统，我们首先在MATLAB上对0.bmp中的红色分量进行2重扩展，然后将其写入文本ram\_1k\_data.v中，其写入顺序如下：

（1）将扩展后1k图片的第1列1至5行写入ram\_1k\_data.v中，然后连续三次写入0，以满足能够利用AXI4-full使得突发读取长度为8。

（2）重复（1）的操作将所有列的1至5行写入ram\_1k\_data.v中，此时就能求出1k图片的第一行对应的4k图片数据。

（3）将2至6行的所有列按上述规则写数ram\_1k\_data.v中，然后依此类推完成整个ram\_1k\_data.v文本的写入。

从以上描述中可以看出，该做法会浪费存储资源，但这方式一方面可以利用AXI4-full总线以便加快PS端与PL端的通信，另一方面读取地址也无需进行跳变。出于简便的目的，我们在PS端将1k像素数据传入DDR3时也采用了该做法。

从0计数，当列数大于等于4时，每从ram\_1k\_data.v突发读取一次，即可得到4k图片的16个点，我们按下图所示顺序将生成的4k并行数据转换为串行数据，然后依次写入ram\_4k\_data.v。



图 5.1 生成4k数据传输顺序

从以上描述可知，存入到ram\_4k\_data.v的数据并非是最终的4k图片数据，而是乱序的数据，因此在仿真完成后我们在MATLAB中将其转换为正常顺序的数据，然后与软件产生的数据进行比对。这种传输方式的优点是操作简单，无需进行存储地址的跳变，且能直接采用AXI4-full突发长度16进行数据存储，加快了PS与PL端的通信速率，但缺点是PS端将数据从DDR3存入SD卡时需要进行顺序变换。出于简便的目的，我们在PL端将数据存入DDR3时也采用了该做法。

## 5.2仿真代码与波形图说明

以下介绍关键代码以及仿真波形：

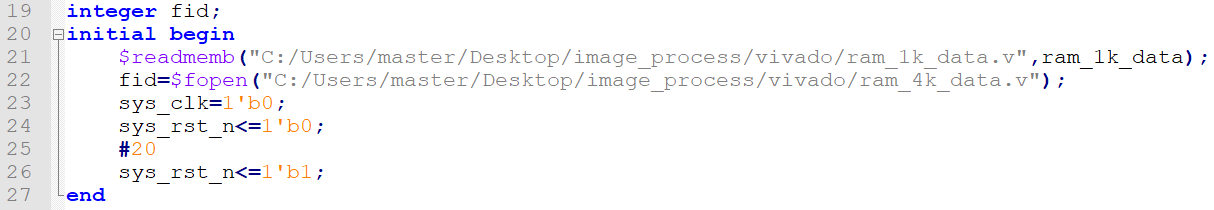


图 5.2 读取ram\_1k\_data.v的数据并生成ram\_4k\_data.v文件

以上代码的功能是将文件ram\_1k\_data.v中的数据读入到寄存器ram\_1k\_data中，并生成文件ram\_4k\_data.v。

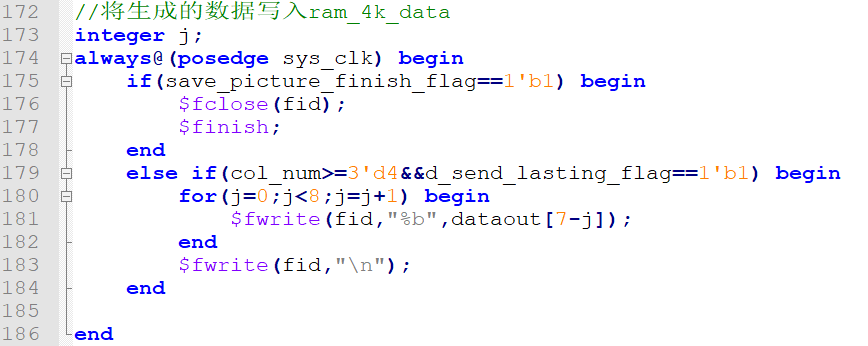


图 5.3 关闭ram\_4k\_data.v退出仿真程序或向ram\_4k\_data.v写入数据

以上代码的功能是判断整张图片是否存储完成，即save\_picture\_finish\_flag拉高时，关闭文件ram\_4k\_data.v然后中止仿真程序，否则的话在相应时间段将数据存入ram\_4k\_data.v。

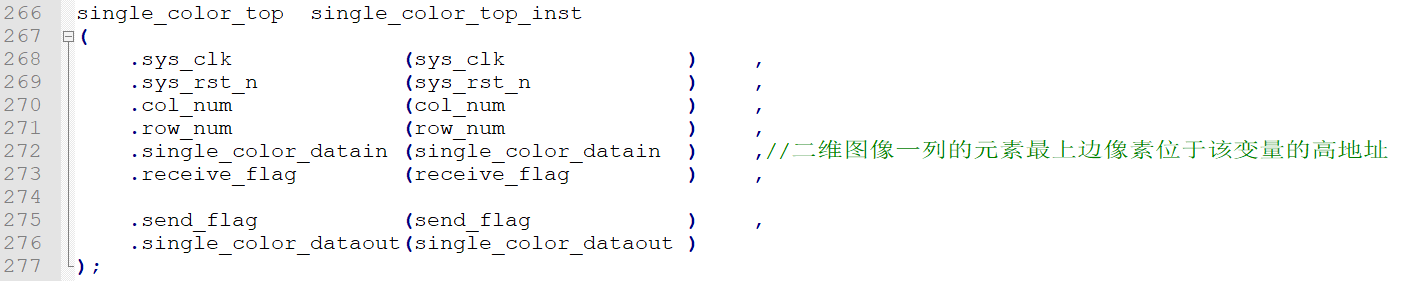


图 5.4 例化单颜色分量插值模块

以上代码的功能是在仿真文件中例化单颜色分量插值模块，以便验证该模块功能是否正确。

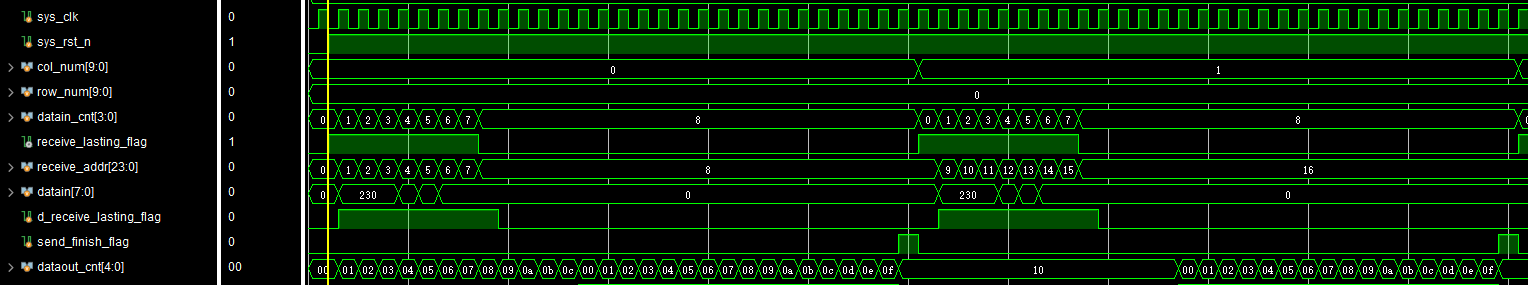


图 5.5 开始读取1k数据波形图

图5.5表示在复位过后开始从ram\_1k\_data中读取1k像素数据，可以看到，此处也是突发读取8个像素数据，但仅有前5个像素是有效的，这与通过AXI4\_full突发读取8个数据是相对应的。

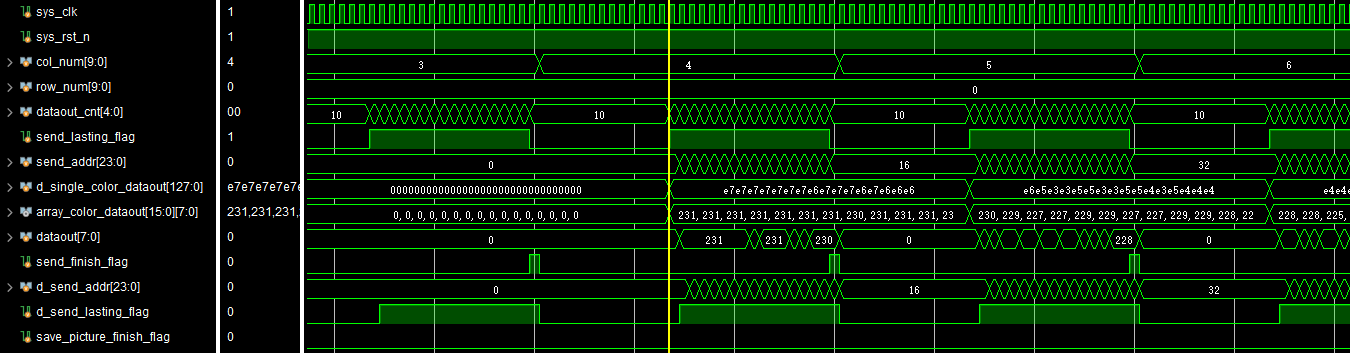


图 5.6 开始存储4k数据波形图

图5.6为开始存储4k数据的波形图，可以看到此时列数col\_num为4，即此时共接收到5列数据才有最终的4k像素数据输出。

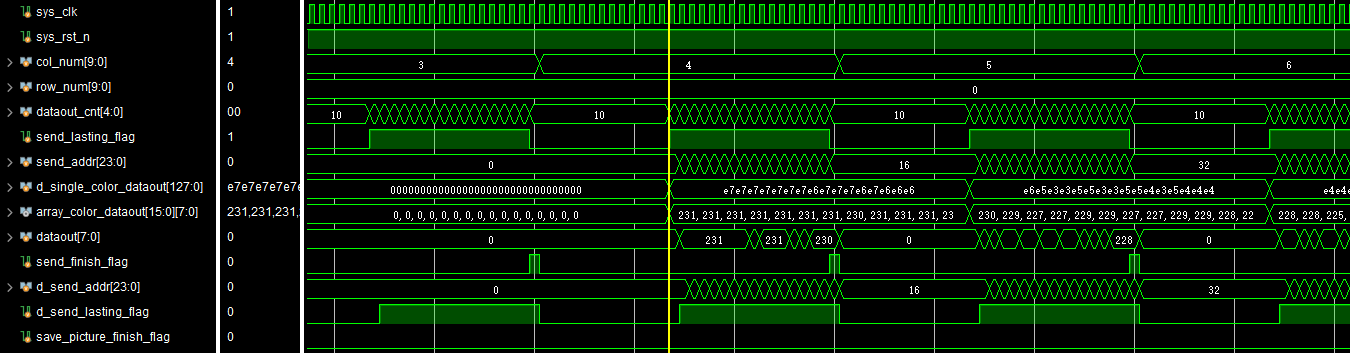


图 5.7 1k图片插值完成波形图

从图5.7可以看到，当行数row\_num和列数col\_num均到达最大值，且send\_finish\_flag拉高时，表示一张4k图片已经插值并存储完成，此时，save\_picture\_flag拉高一个时钟周期，而行列均归0，dataout输出正确的4k像素数据，可以看到，发送地址d\_send\_addr最终的值为8294399，即共存储了8294400个像素数据，而4k图片行数与列数之积与之相匹配，证明我们的仿真代码以及单颜色插值模块是正确的。

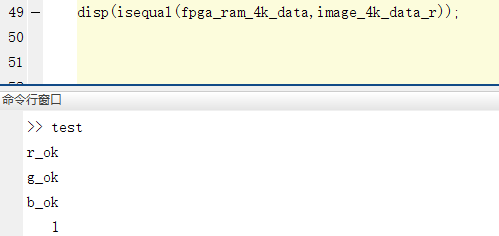


图 5.8 仿真产生图片与软件产生图片一致截图

我们将生成的ram\_4k\_data.v导入MATLAB并进行顺序转换，得到正序的4k像素数据，然后调用MATLAB自带的比较函数isequal来判断软件生成的图片与仿真生成的图片是否一致，以上可以看出，最终输出的结果为1，表明两者一致。

由于我们对生成一整张4k图片进行了仿真，而且仿真结果与软件一致，这表明我们的整个插值模块是正确的，且仿真的代码覆盖率为100%，因此不再进行单独的代码覆盖率测试。

# 六、性能评估说明

## 6.1算法性能

组合1：新型双线性插值+均值调整

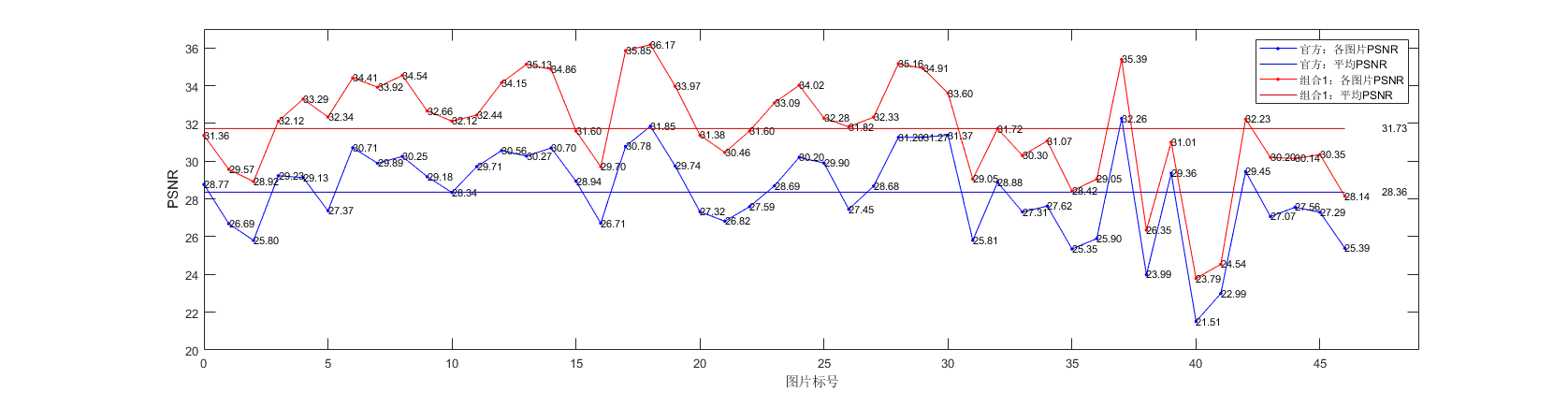


图 6.1 组合1 PSNR性能

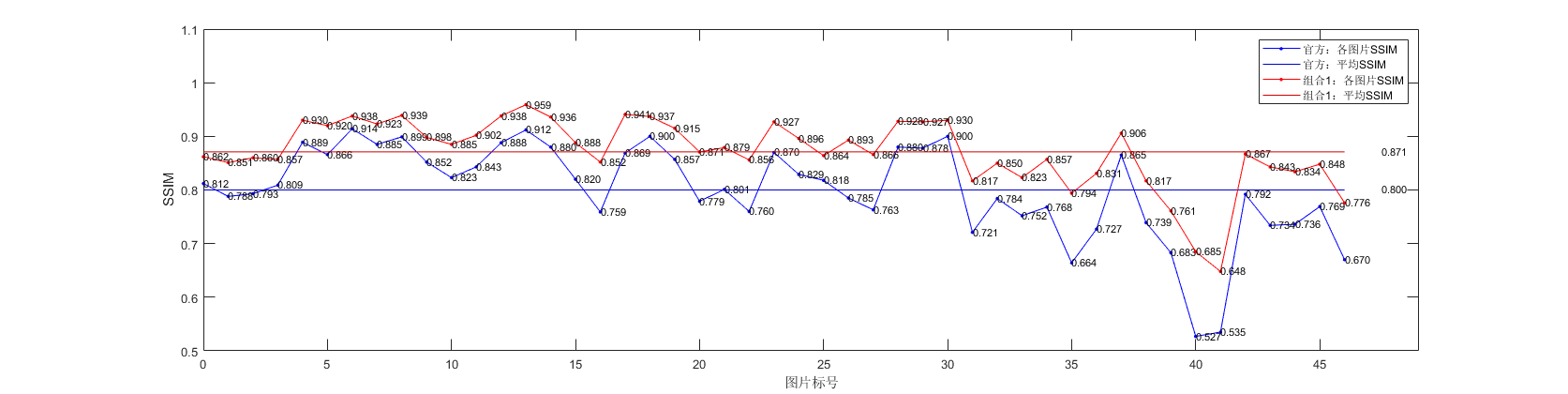


图 6.2 组合1 SSIM性能

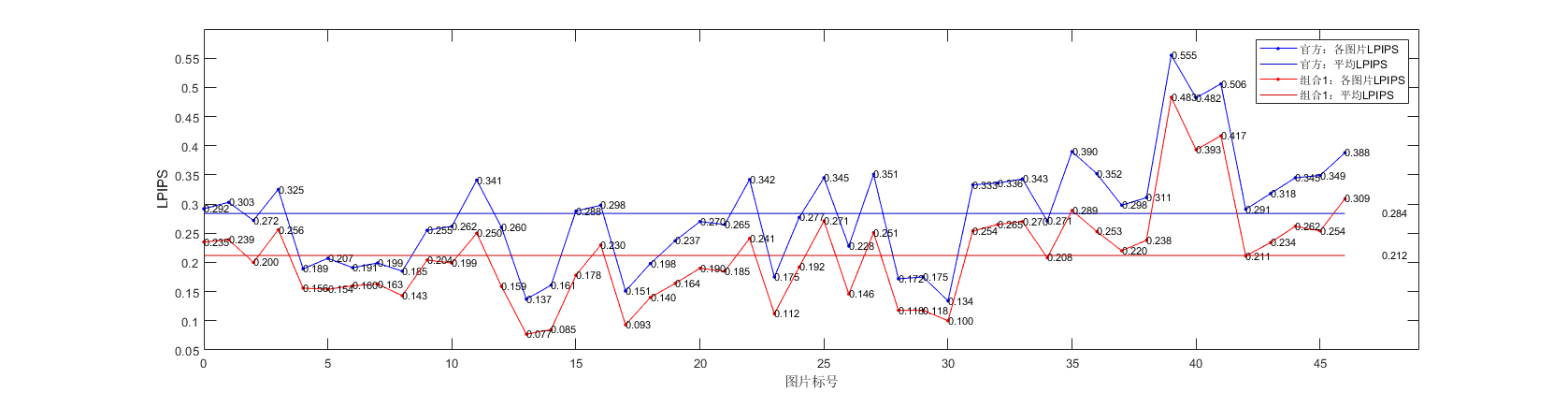


图 6.3 组合1 LPIPS性能

组合2：新型双线性插值+均值调整+高斯平滑+均值调整

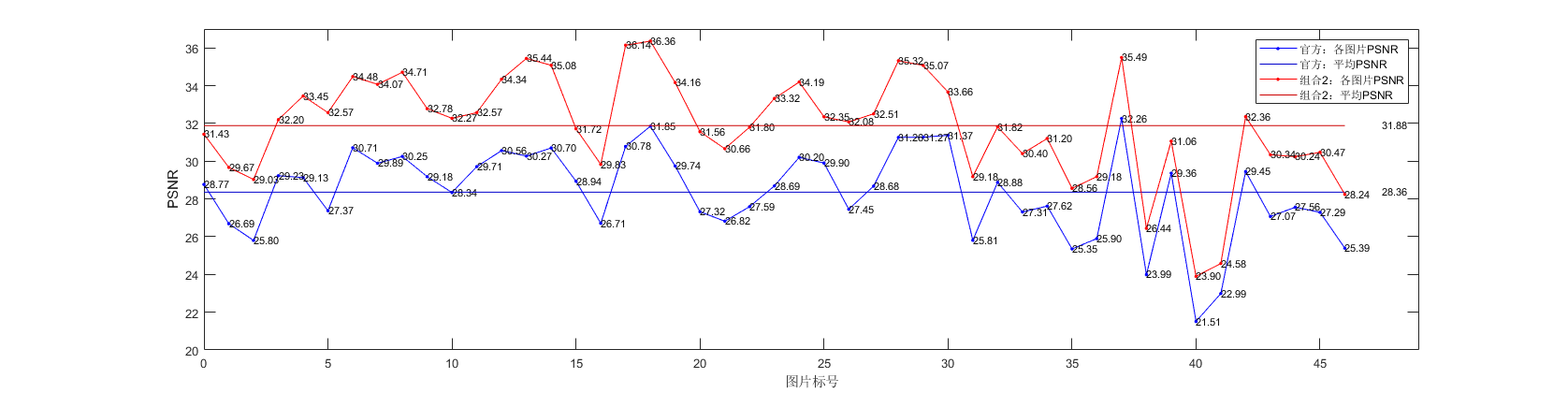


图 6.4 组合2 PSNR性能

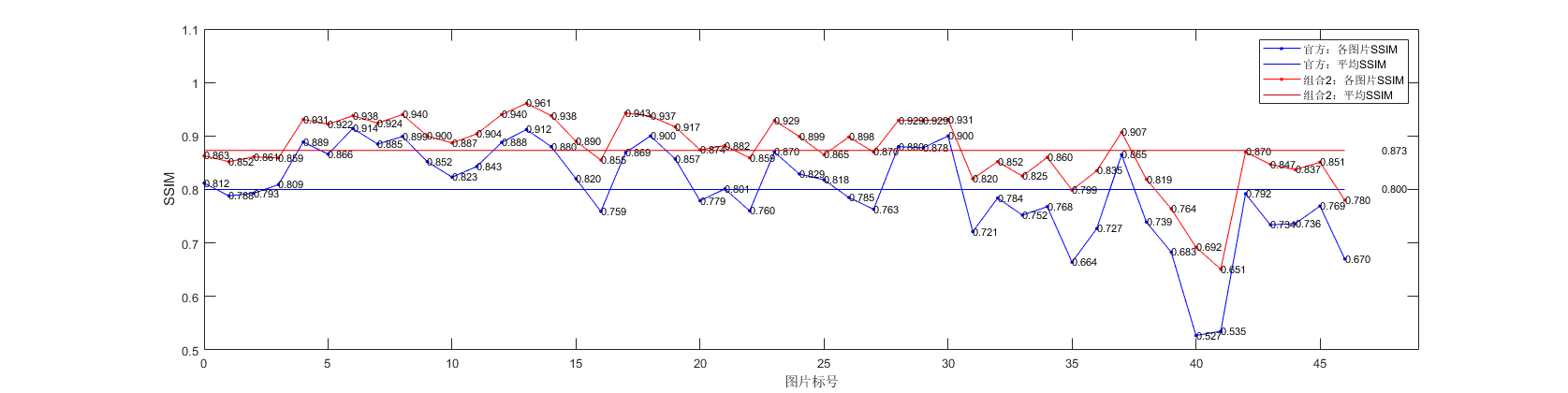


图 6.5 组合2 SSIM性能

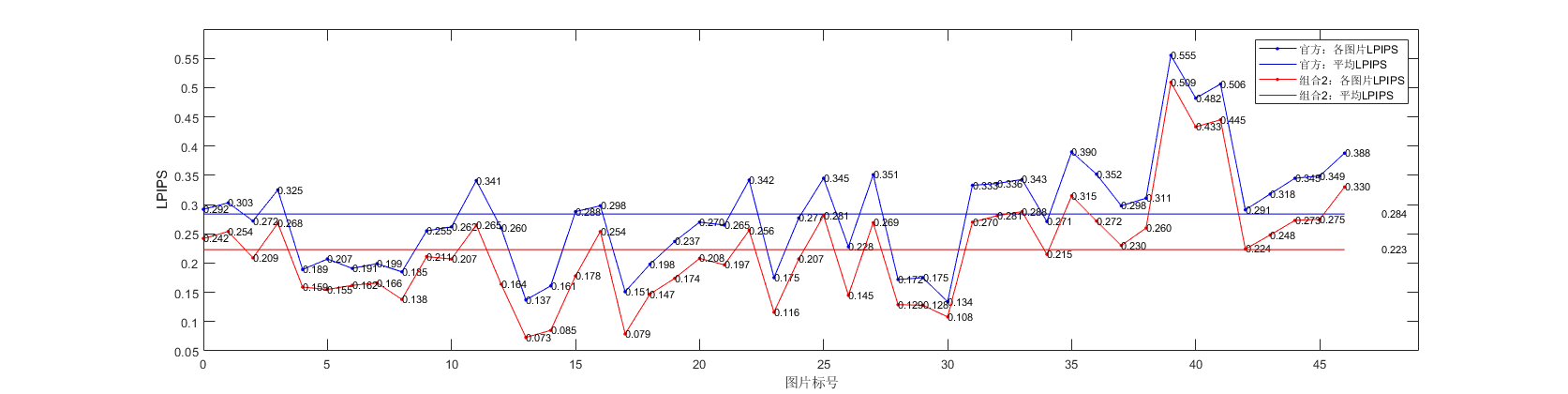


图 6.6 组合2 LPIPS性能

组合3：分区域插值+均值调整+高斯平滑+均值调整

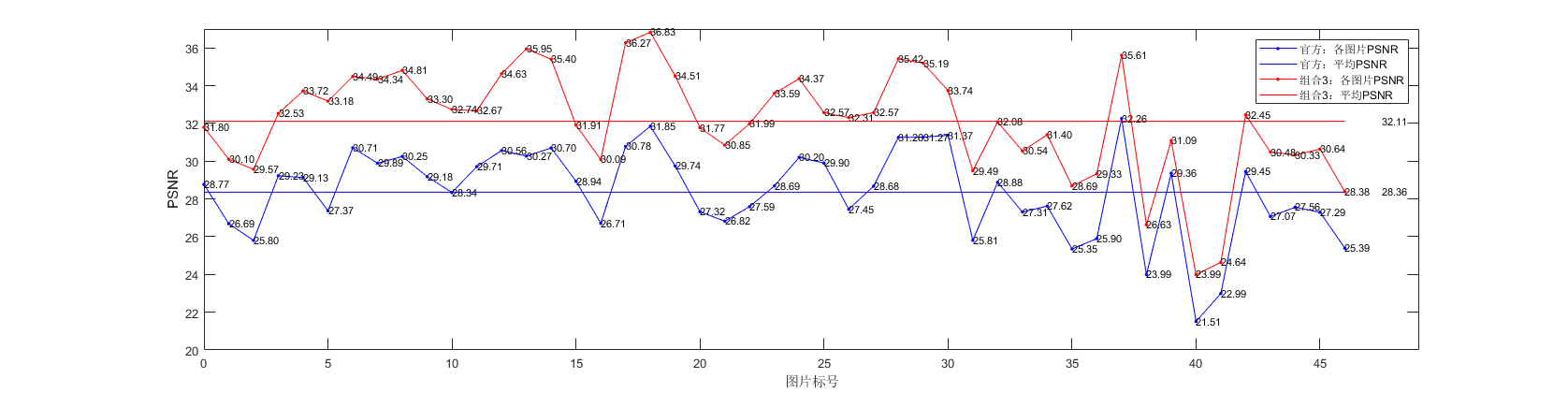


图 6.7 组合3 PSNR性能

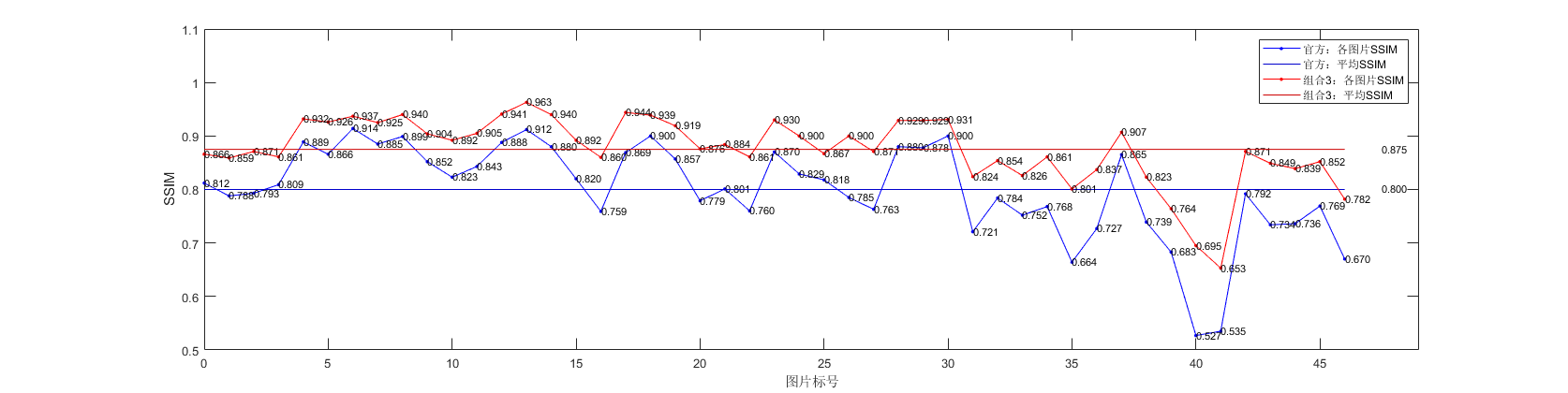


图 6.8 组合3 SSIM性能

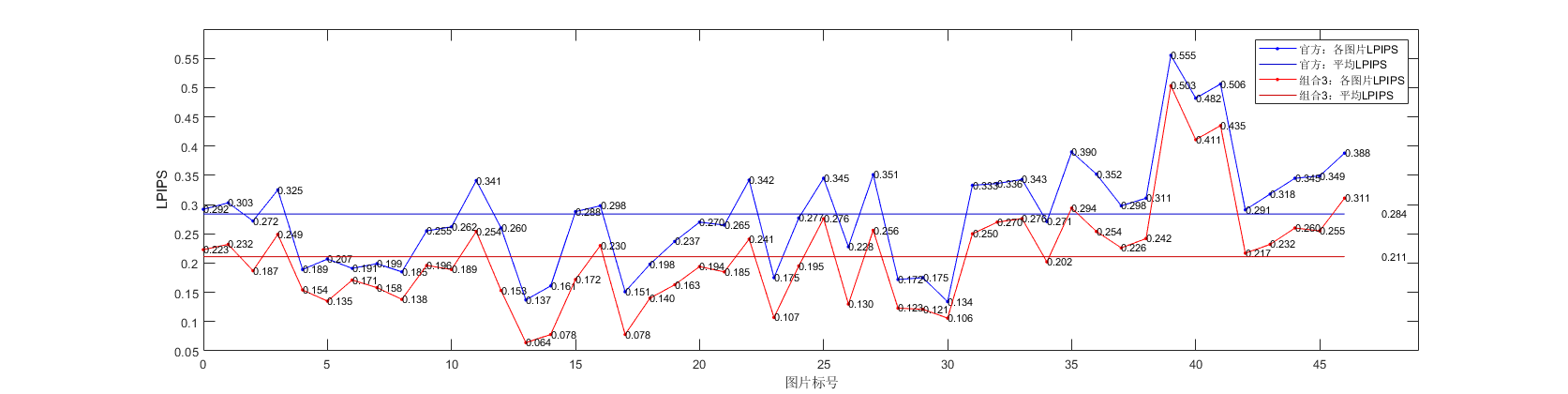


图 6.9 组合3 LPIPS性能

组合4：3区域均采用边缘插值+均值调整+高斯平滑+均值调整

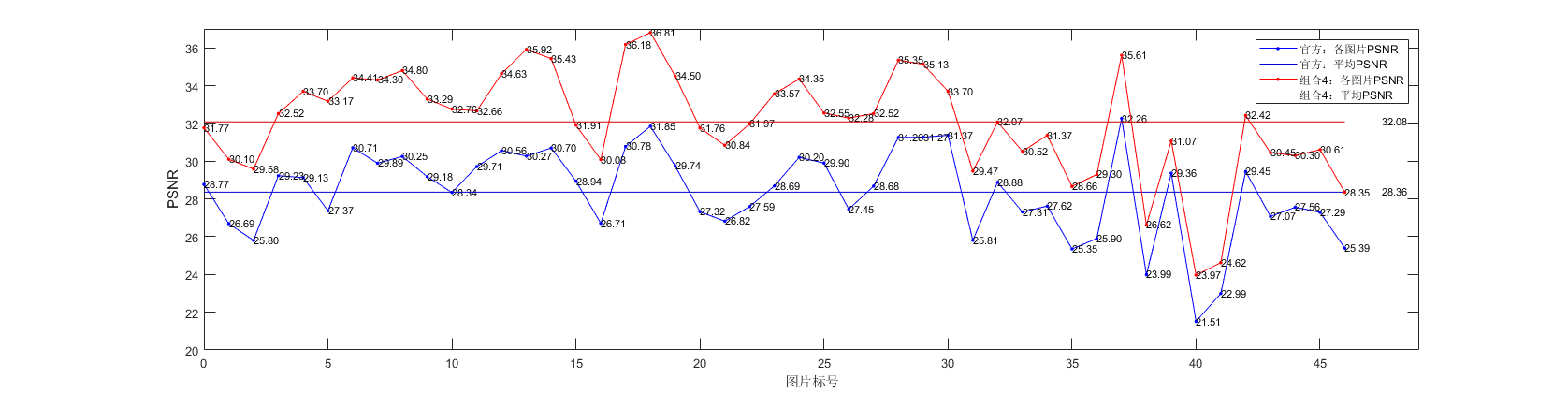


图 6.10 组合4 PSNR性能

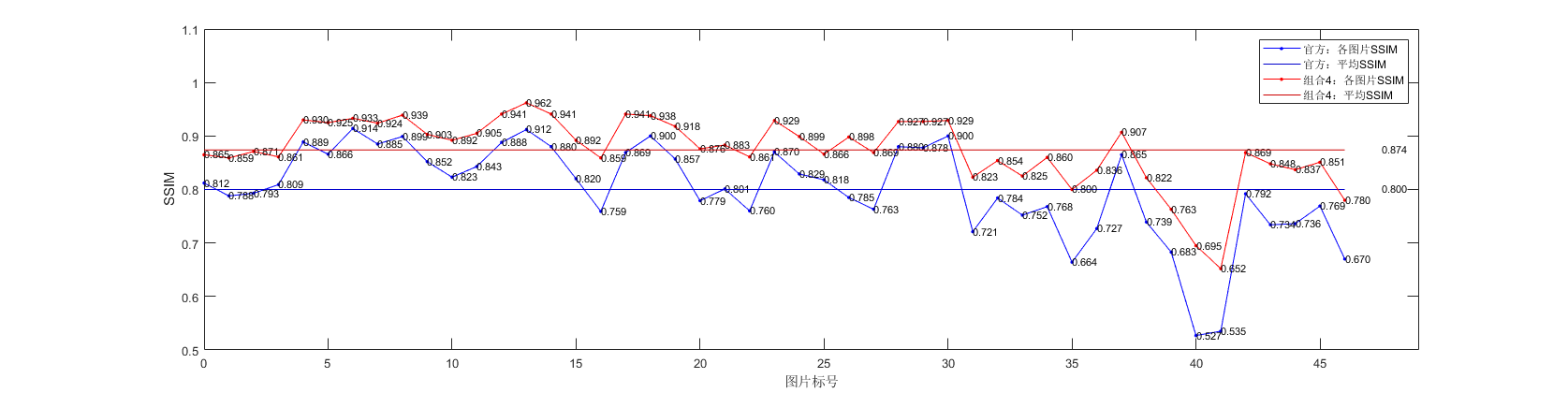


图 6.11 组合4 SSIM性能

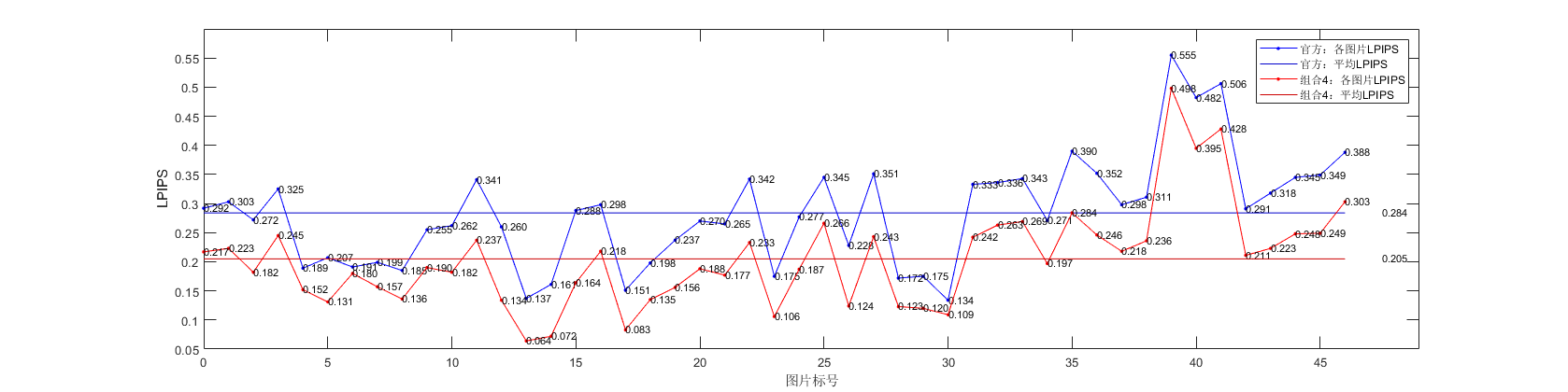


图 6.12 组合4 LPIPS性能

组合5：3区域均采用边缘插值+均值调整+高斯平滑

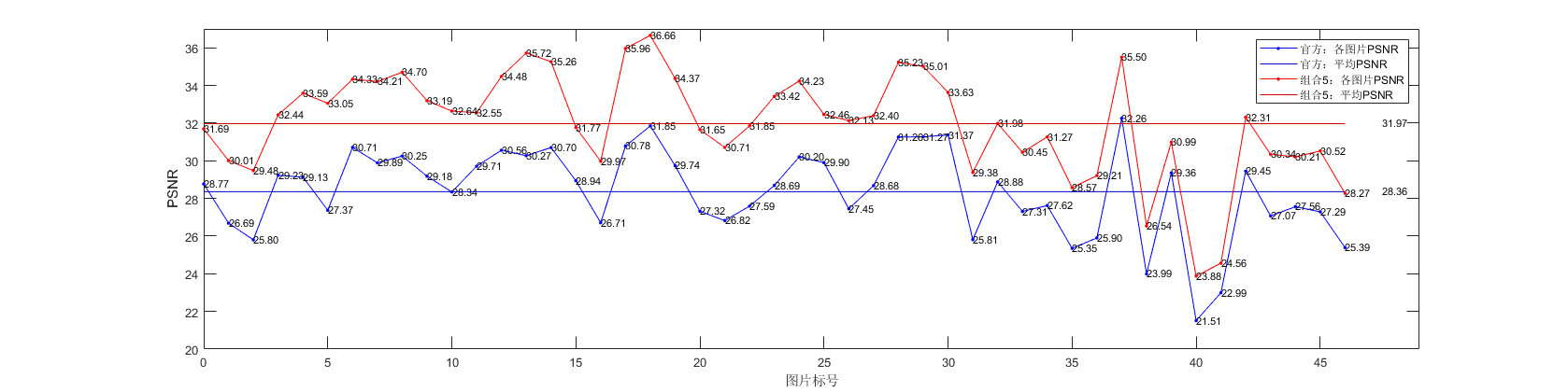


图 6.13 组合5 PSNR性能

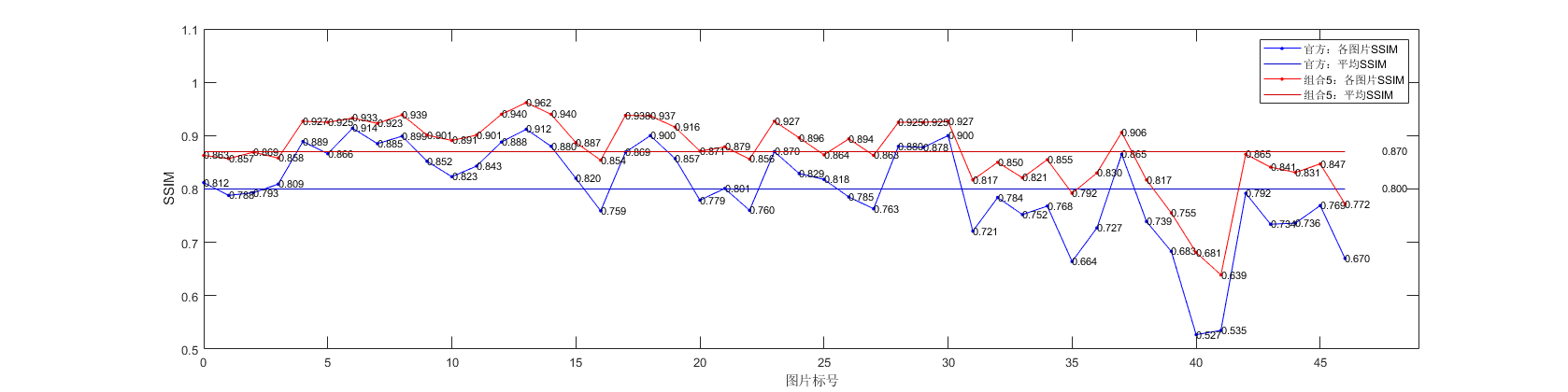


图 6.14 组合5 SSIM性能

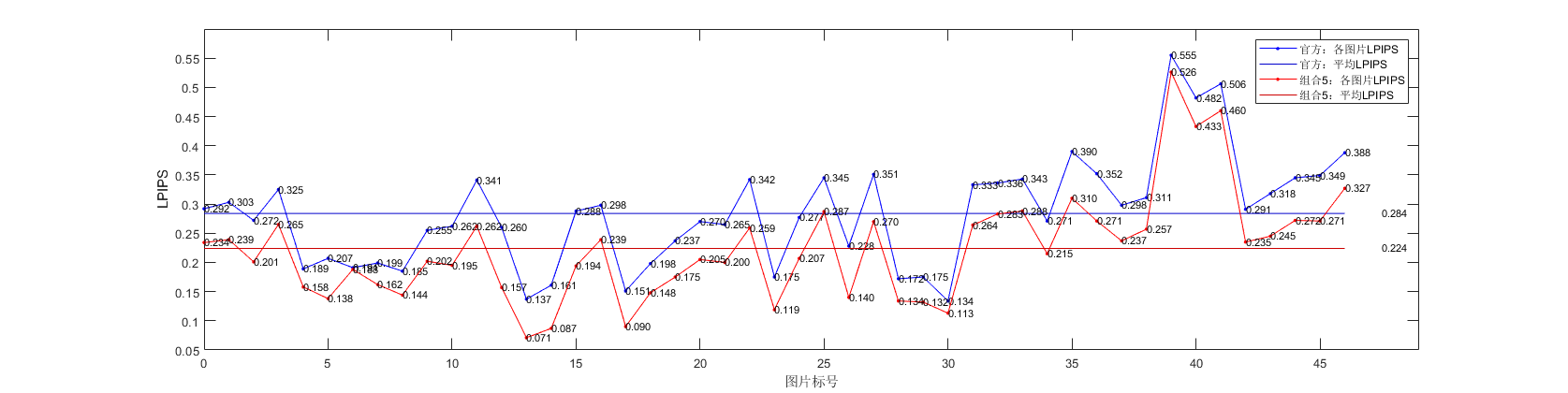


图 6.15 组合5 LPIPS性能

从以上性能仿真图中可以看出，不同组合方式具有不同的性能和算法复杂度，其中组合1复杂度最低且具有相对高的性能，组合3复杂度最高，PSNR与SSIM性能也最优，而组合4在3个区域内均采用边缘插值，相比于组合3复杂度有所下降，而性能基本不变，因此我们选取组合4作为最终硬件实现的算法。在经过四舍五入后，组合4的平均PSNR=32.08，SSIM=0.874，LPIPS=0.205，相比于官方给的PSNR=28.36，SSIM=0.800，LPIPS=0.284均有着较大幅度的提升，且算法复杂度不高，具有广阔的应用前景。

## 6.2硬件资源

下图为整个顶层模块所耗资源示意图，其使用LUT数为11220，使用FF数为3586，可以看出，整体算法复杂度相对较低，所用资源也较少。

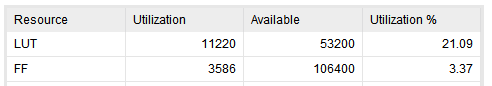


图 6.16 顶层资源消耗

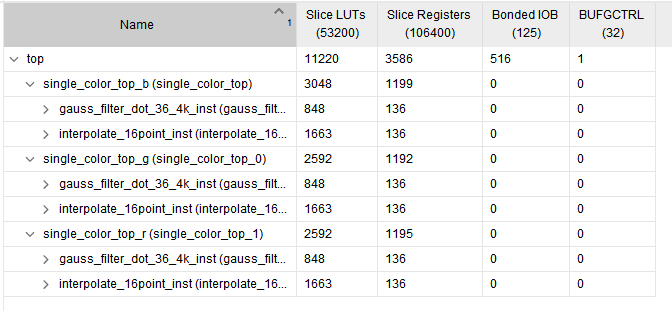


图 6.17 顶层模块资源消耗分布

图6.18为各个模块LUT的资源消耗量，新型双线性插值模块其使用LUT数为837，由算法设计说明中可知，其复杂度与传统双线性插值一致；边缘插值模块所用LUT数为1815，近似为新型双线性插值模块的2.2倍，由于水平方向插值与竖直方向插值原理一致，因此可通过多消耗一个时钟进行硬件复用，从而使该插值模块复杂度降低一倍，使得其近似等于新型双线性插值，但在本设计中未采用；16点均值调整模块使用LUT数为605；单个点进行高斯平滑所用LUT数为40个，若对16点进行高斯平滑，其使用总LUT数为。

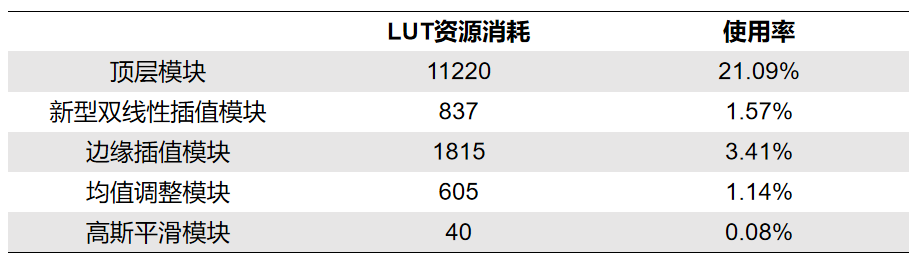


图 6.18 各模块LUT资源消耗

# 七、FPGA验证报告

## 7.1FPGA验证

根据赛题要求，我们进行了多种图像插值与处理算法的设计与仿真，并对其性能与硬件实现进行评估，最终采用组合4：边缘插值+均值调整+高斯平滑+均值调整的方案，具体见算法设计说明部分。基于此，我们在Xilinx Zynq-7020 FPGA开发板上搭建整个系统。

针对1k图像集的图片，我们首先通过软件仿真得到相应的上采样4k图片，再将1k图像集存入SD卡中，通过FPGA对其进行上采样，同样得到相应的上采样4k图片，最后将软件与硬件得到的4k图片导入MATLAB进行数据对比，结果表明软硬件对比一致，FPGA验证通过。

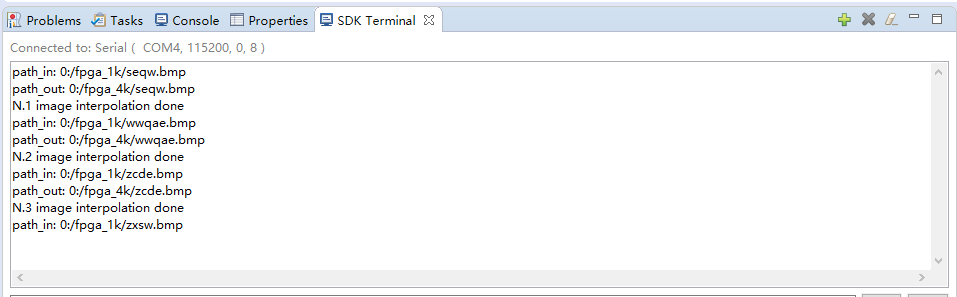


图 7.1 插值处理过程

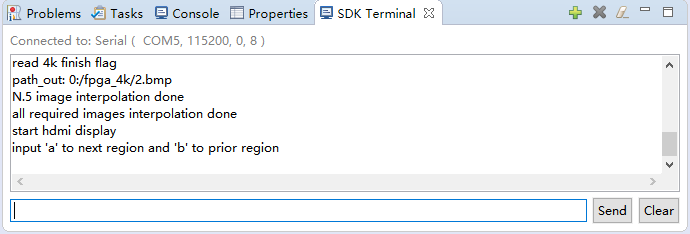


图 7.2 插值完成开始hdmi显示

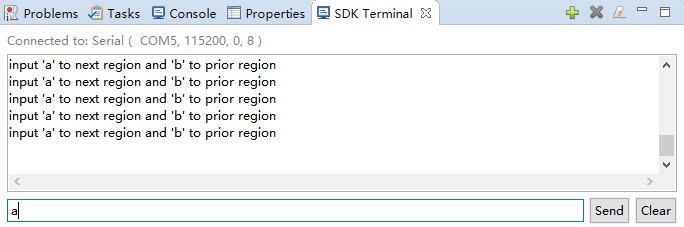


图 7.3 输入字符控制显示区域

图7.1打印的是各个图片插值处理的过程。图7.2打印的是所有图片插值完成，配置VDMA并开始HDMI显示，此时我们可以通过输入字符a以及字符b来改变显示区域，其中a表示下一区域，b表示上一区域。图7.3打印了改变显示区域的过程。

|  |  |
| --- | --- |
| C:\Users\master\Desktop\final\比赛素材\图3.jpg | C:\Users\master\Desktop\final\比赛素材\图4.jpg |
| C:\Users\master\Desktop\final\比赛素材\图1.jpg | C:\Users\master\Desktop\final\比赛素材\图2.jpg |

图 7.4 改变显示区域效果图

可以看到，图7.4展示了在PS端改变显示区域时，外接显示屏的显示内容也跟着变化，从而实现了对一张4k图片分时显示的功能。

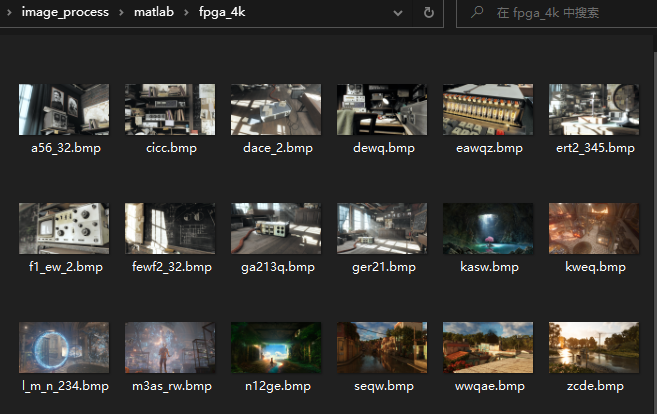


图 7.5 FPGA生成4k图片截图

## 7.2设计优缺点

本设计的优点一方面体现在算法的设计上，我们在此设计中提出了新型双线

性插值以及边缘插值，其算法复杂度低，逻辑简单且性能较为优异，而且可以通过不同的组合方式来得到不同性能与资源消耗的算法，应用范围广。对于边缘插值而言，其采用阶跃函数来拟合图像的梯度变化，不像常规的多项式拟合在无穷处会发散，也有别于变换域超分辨率算法或深度学习算法，提供了一种值得深入研究的新的方向。

本设计的缺点为采用了全局时钟，利用32位的DDR3存取1k图片以及生成的4k图片，每次仅能读取或存储一个像素点，处理速度会受到严重的限制，可以通过异步时钟来提高DDR3存取时钟而插值模块时钟保持不变来解决。其次，在对1k图片进行边缘插值+均值调整时，我们花费了3个时钟来复用边缘插值+均值调整模块，但对于第一次和第三次边缘插值+均值调整所得的8行数据，我们只取了其中2行数据，造成了资源的浪费。另外，对于边缘插值模块，由于水平与竖直插值原理一致，因此可通过多花一个时钟将硬件复用，进而节省该模块近一半的资源，从而使得顶层模块LUT消耗减少约2700个。