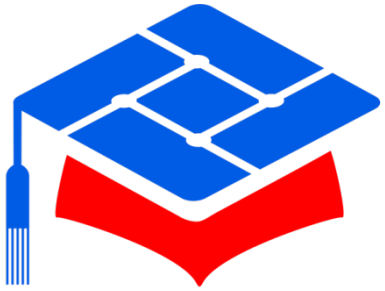

第六届

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

1.1 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。

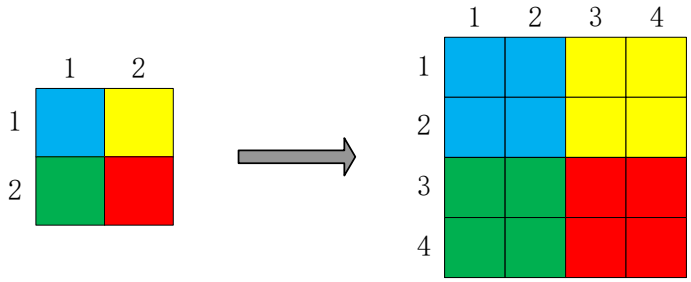

图 1.1 最近邻插值原理图

（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：

$$P=(1-x)(1-y)Q_{11}+x(1-y)Q_{12}+(1-x)yQ_{21}+xyQ_{22} \tag{1-1}$$

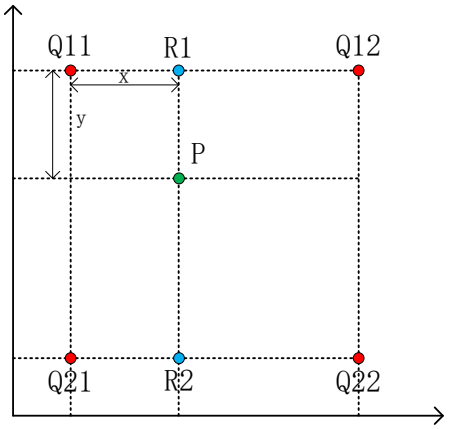

图 1.2 双线性插值原理图

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 4×4 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 Q_{21} 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 Q_{11} 的偏移为 $x=\frac{1}{8}, y=\frac{5}{8}$，带入传统的双线性插值公式，可得 P 点的像素为：$P=\frac{21}{64}Q_{11}+\frac{3}{64}Q_{12}+\frac{35}{64}Q_{21}+\frac{5}{64}Q_{22}$。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。

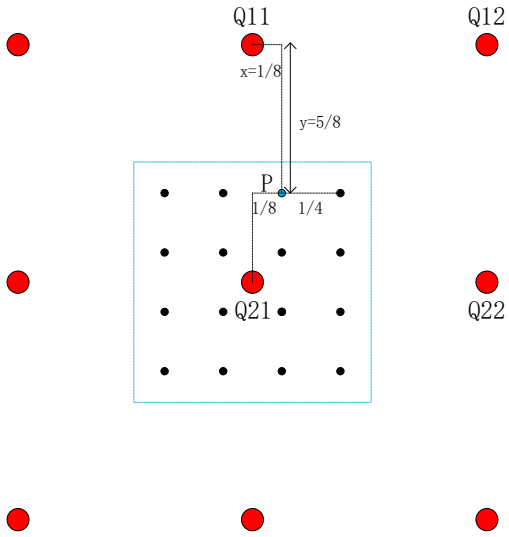

图 1.3 新型双线性插值原理图

1.1.2 均值调整

在本设计中，1k 图像是经过 4×4 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：

$$\text{adjust\_dot\_16\_4k}=\text{dot\_16\_4k}-(\text{fix}(\frac{\text{sum}(\text{sum}(\text{dot\_16\_4k}))}{16})-\text{average}) \tag{1-2}$$

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：

$$\begin{aligned}\text{pixel\_left}&=\text{fix}(\frac{5}{8}\times\text{dot\_9\_2\_1}+\frac{3}{8}\times\text{dot\_9\_1\_1})\\ \text{pixel\_right}&=\text{fix}(\frac{5}{8}\times\text{dot\_9\_2\_3}+\frac{3}{8}\times\text{dot\_9\_1\_3})\end{aligned} \tag{1-3}$$

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：

$$\begin{aligned} d\_left&=fix(\frac{5}{8}\times d2\_left+\frac{3}{8}\times d1\_left)\\ d\_right&=fix(\frac{5}{8}\times d2\_right+\frac{3}{8}\times d1\_right) \end{aligned} \tag{1-4}$$

其中

$$\begin{aligned} d1\_left&=dot\_9\_1\_2-dot\_9\_1\_1\\ d1\_right&=dot\_9\_1\_3-dot\_9\_1\_2\\ d2\_left&=dot\_9\_2\_2-dot\_9\_2\_1\\ d2\_right&=dot\_9\_2\_3-dot\_9\_2\_2 \end{aligned} \tag{1-5}$$

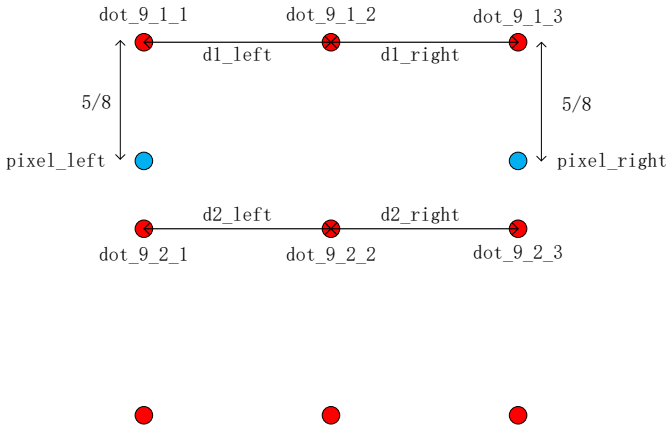

图 1.4 左右像素和梯度求解示意图

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：

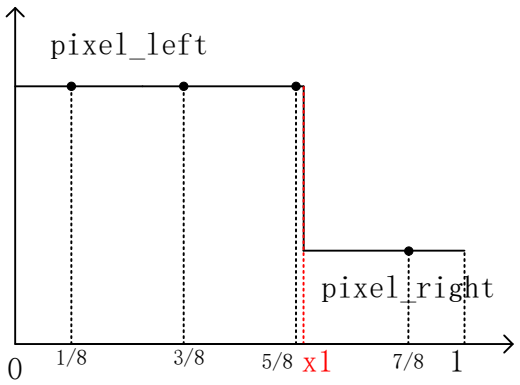

图 1.5 求解 4 点像素示意图

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$\frac{1}{8},\frac{3}{8},\frac{5}{8},\frac{7}{8}$，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：

$$x1=\frac{abs(d\_right)}{abs(d\_right)+abs(d\_left)} \tag{1-6}$$

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：

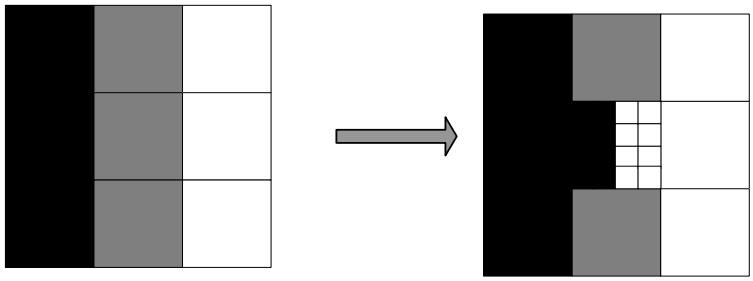

图 1.6 边界增强示意图

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 3×3 的高斯平滑，其模板如下：

$$\frac{1}{16}\times\begin{pmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{pmatrix}$$

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

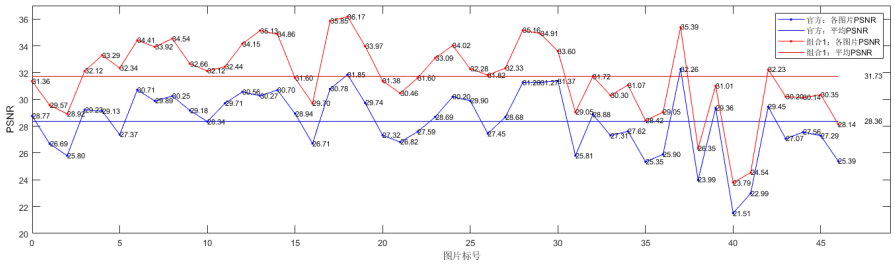

图 1.7 组合 1 PSNR 性能

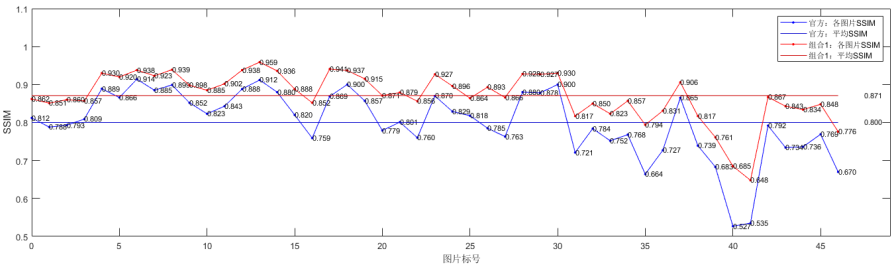

图 1.8 组合 1 SSIM 性能

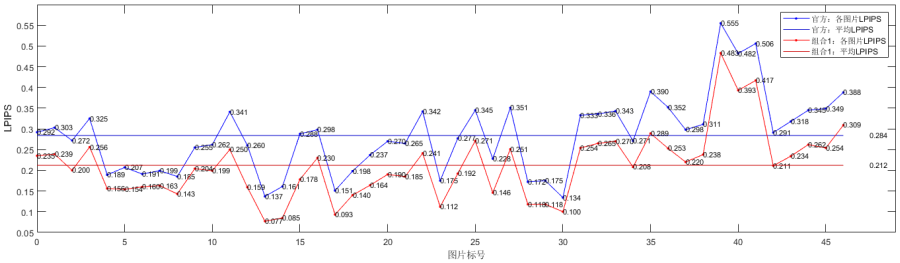

图 1.9 组合 1 LPIPS 性能

组合 2：新型双线性插值+均值调整+高斯平滑+均值调整

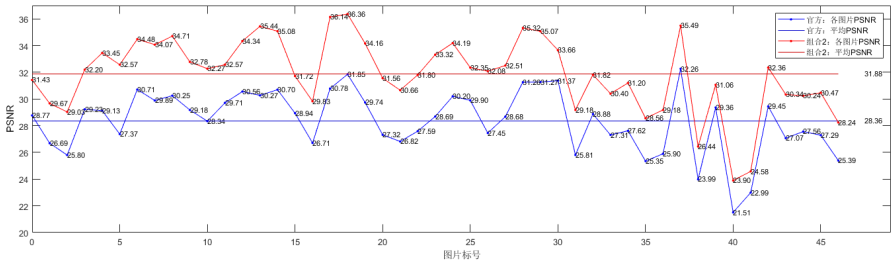

图 1.10 组合 2 PSNR 性能

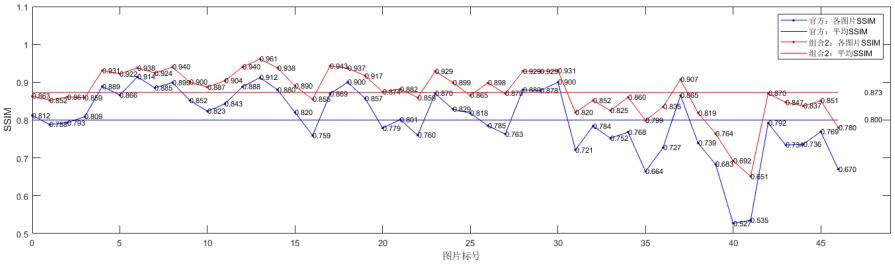

图 1.11 组合 2 SSIM 性能

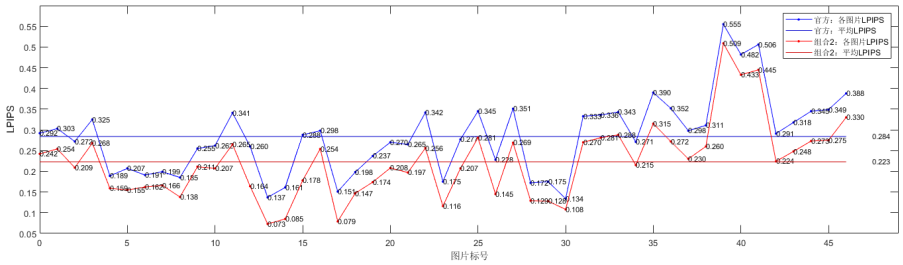

图 1.12 组合 2 LPIPS 性能

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 3×3 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 $d\_r \le T\_\min$ 且 $d\_c \le T\_\min$，其中 $T\_\min = 10$，则该 9 点采用最近邻插值，若 $d\_r \ge T\_\max$ 或 $d\_c \ge T\_\max$，其中 $T\_\max = 120$，则该 9 点采用边缘插值，其余区域采用新型双线性插值。

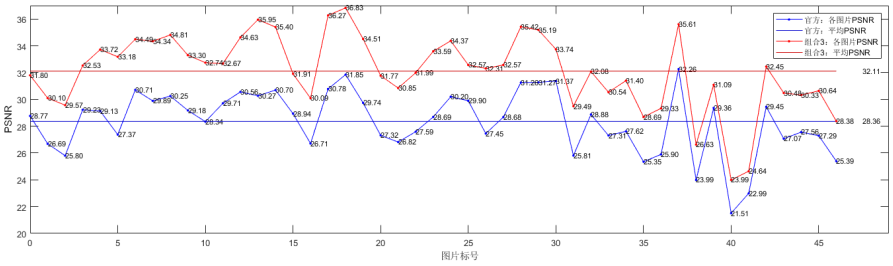

图 1.13 组合 3 PSNR 性能

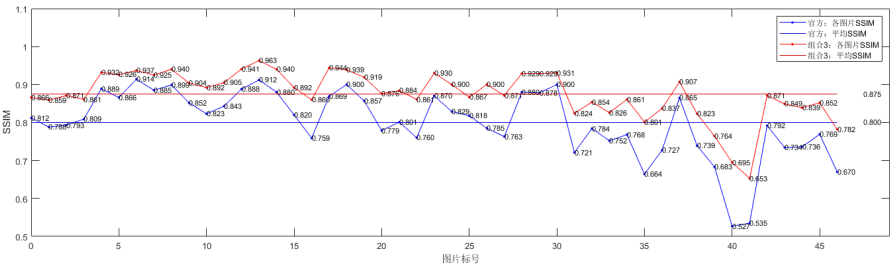

图 1.14 组合 3 SSIM 性能

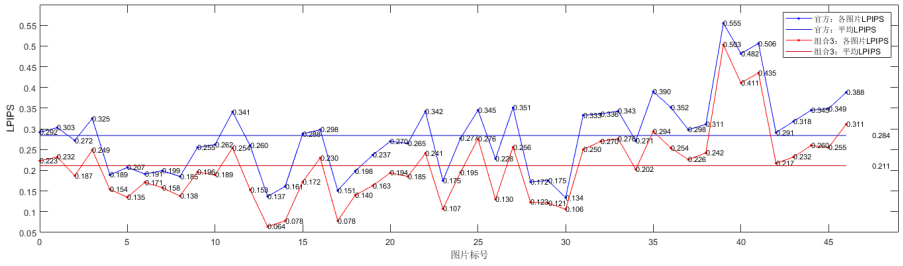

图 1.15 组合 3 LPIPS 性能

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

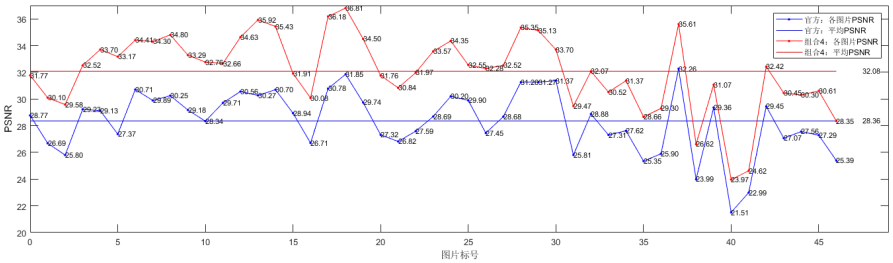

图 1.16 组合 4 PSNR 性能

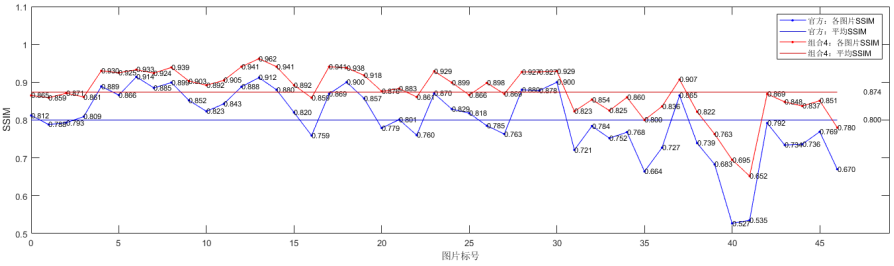

图 1.17 组合 4 SSIM 性能

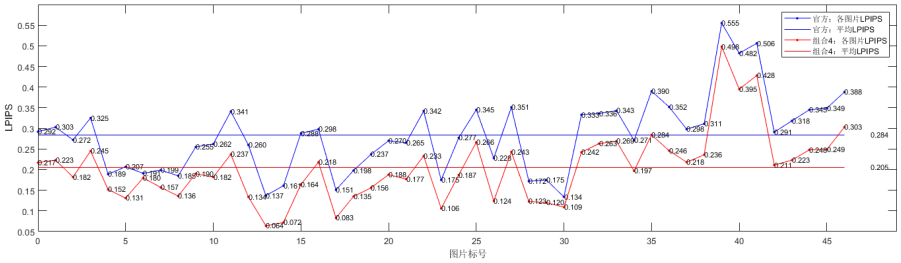

图 1.18 组合 4 LPIPS 性能

组合 **5**：3 区域均采用边缘插值+均值调整+高斯平滑

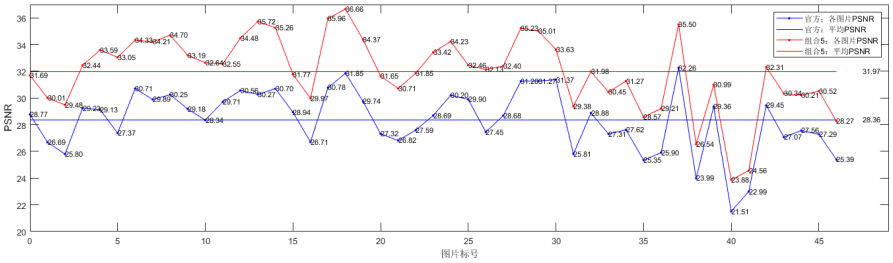

图 1.19 组合 5 PSNR 性能

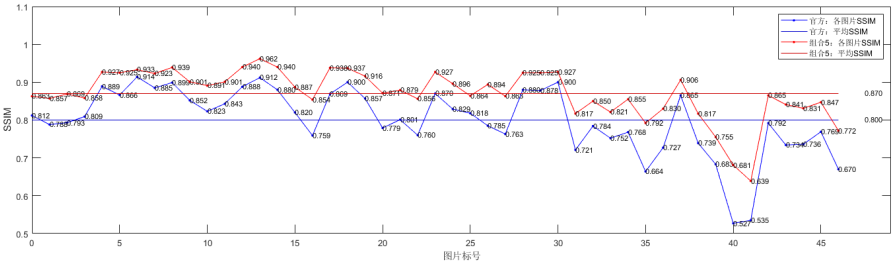

图 1.20 组合 5 SSIM 性能

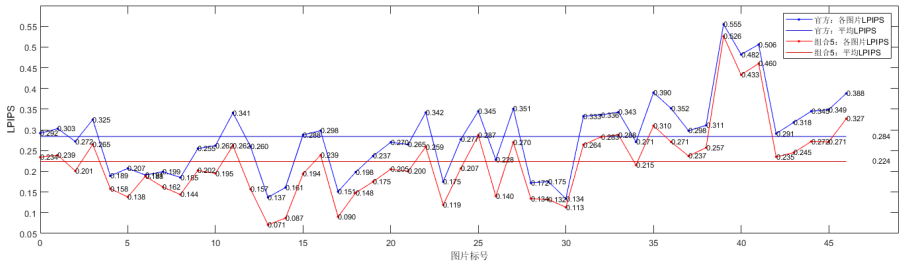

图 1.21 组合 5 LPIPS 性能

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。

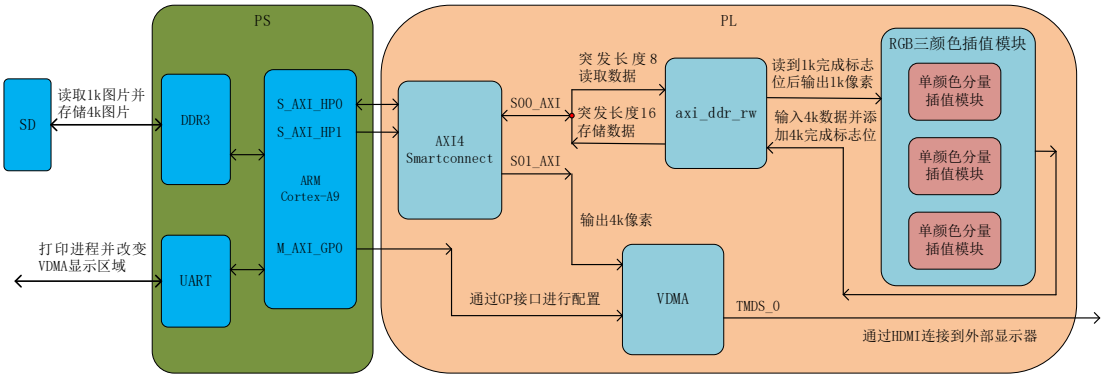

图 1.22 系统硬件框架图

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：

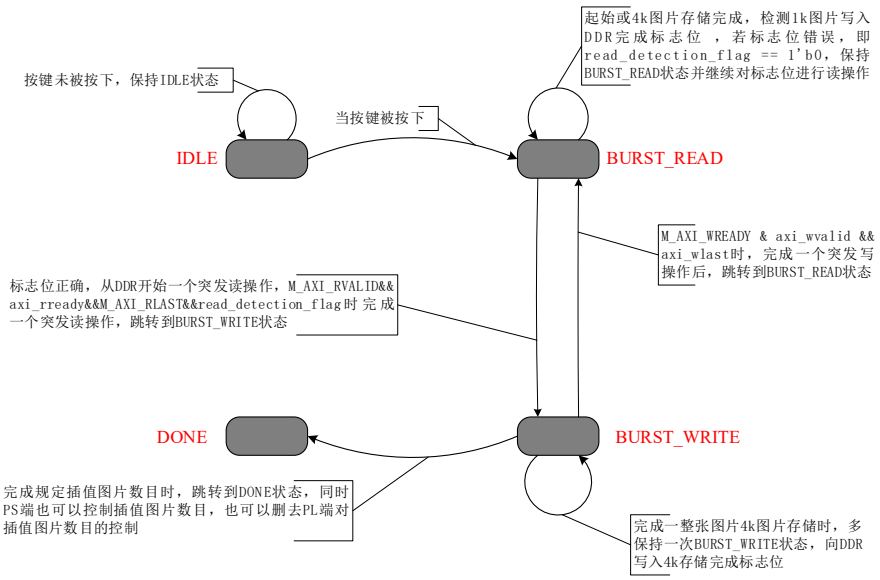

图 1.23 axi_ddr_rw 实现功能状态机

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。

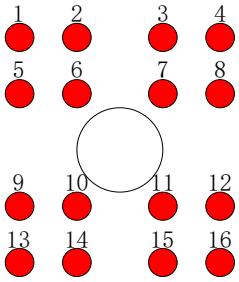

图 1.24 生成 4k 数据传输顺序

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：

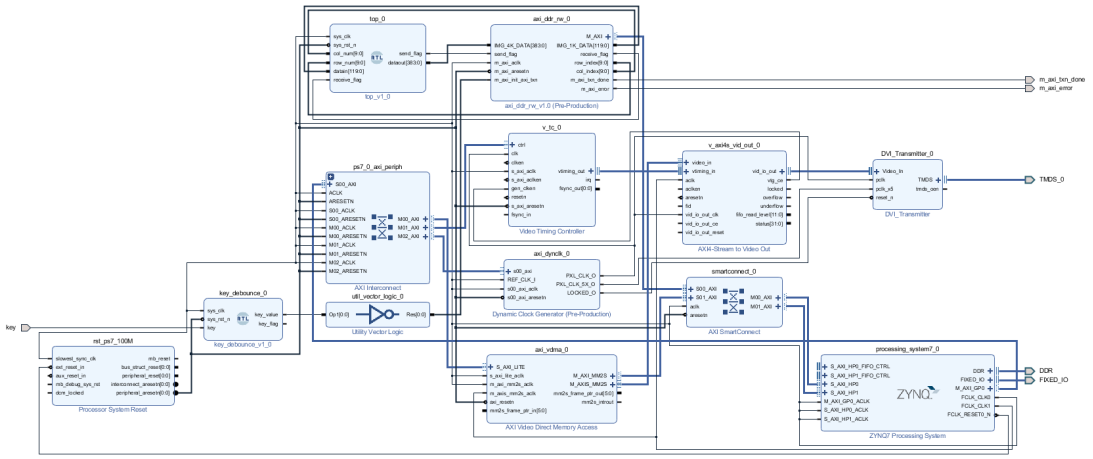

图 1.25 系统 bd 框图

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：

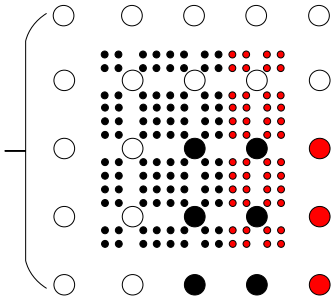

图 1.26 插值流程示意图 1

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。

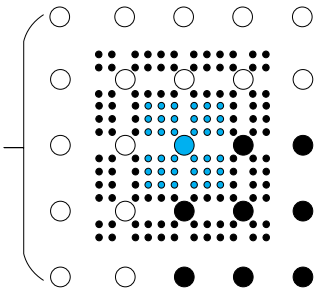

图 1.27 插值流程示意图 2

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6×6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：

$$3\times960\times540=1555200$$

因此，1s 钟能够处理的图片数为$\frac{100\times10^6}{1555200}=64.3$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：

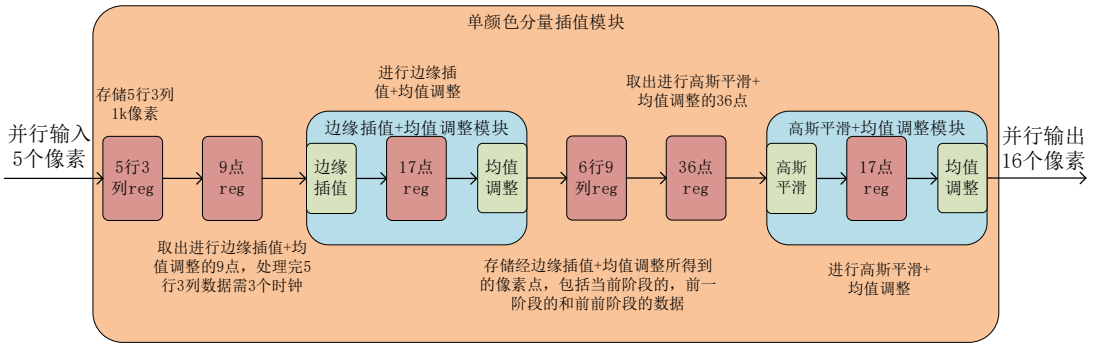

图 1.28 单颜色分量插值模块框架

最终用 vivado 综合出来的框图为：

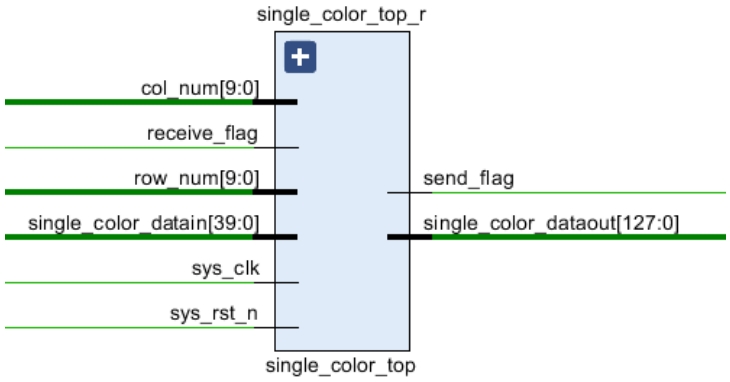

图 1.29 vivado 综合出的单颜色分量插值模块框图

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

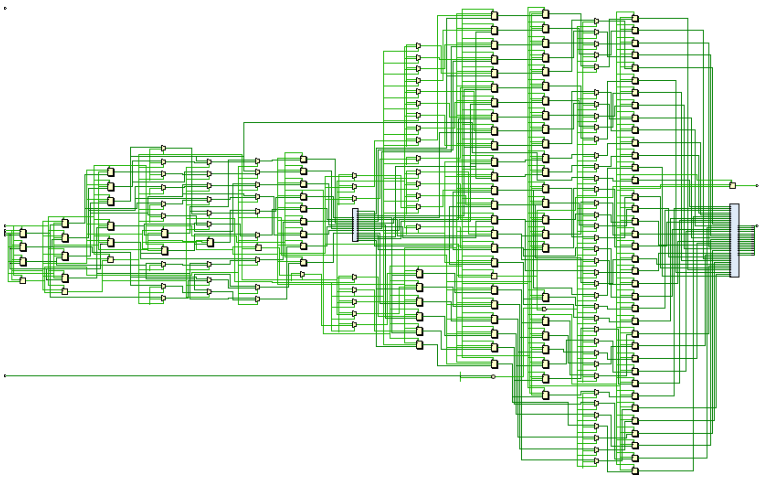

图 1.30 单颜色分量插值模块内部电路图

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。

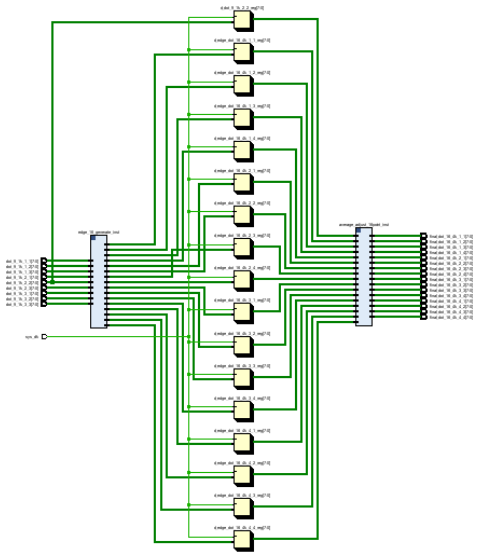

图 1.31 vivado 综合出的边缘插值+均值调整模块框图

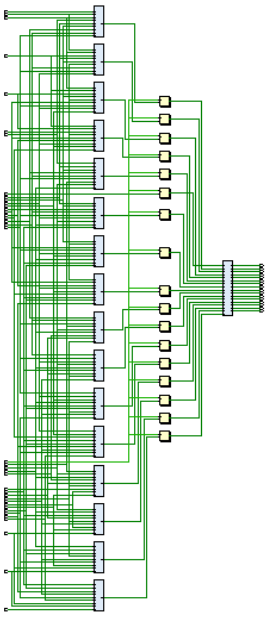

图 1.32 vivado 综合出的高斯平滑+均值调整模块框图

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。

```
    //插值部分
    int i=0;
    while(1)
    {
        status=readBMP(bitInfo, path_in[i]);//根据输入路径读取图片
        while(status!=FR_OK)
        {
            printf("read picture again\n");
            status=readBMP(bitInfo, path_in[i]);//根据输入路径读取图片
            sleep(1);
        }
        expand_1k_image(bitInfo,image_1k_data);//对1k图片进行2重扩展
        store_1k_to_ddr(image_1k_data);//将扩展后的1k按要求存入ddr
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000), (dWORD)i);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+4), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+8), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+12), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+16), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+20), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+24), (dWORD)0xffffffff);
        Xil_Out32((UINTPTR)(base_addr_1k+0x04000000+28), (dWORD)0xffffffff);

        while(1)//等，直到读到4k插值完成标志位
        {
            finish_4k_flag=Xil_In32((UINTPTR)(base_addr_4k+0x04000000));
            if(finish_4k_flag==i)
            {
                finish_4k_flag=Xil_In32((UINTPTR)(base_addr_4k+0x04000000+4));
                if(finish_4k_flag==0x00ffffff)
                {
                    printf("read 4k finish flag\n");
                    break;
                }
            }
            usleep(10);
        }

        read_4k_from_ddr(image_4k_data);//从ddr中按要求读出4k图片
        store_4k_for_vdma((dWORD*)image_4k_data);//将4k图片数据分4个区域存入ddr，以便vdma显示
        savePicture(bitInfo,image_4k_data,path_out[i]);//根据输出路径path将4k图片存入sd卡
        i=i+1;
        printf("N.%d image interpolation done\n",i);
        if(i == fileCnt||i == 5)//完成了要求的图片数
        {
            printf("all required images interpolation done\n");
            break;
        }
    }
```

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3×5×8＝120 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 9×8＝72 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$17\times8=136$ 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$6\times6=36$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$6\times9\times8=432$ 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$36\times8=288$ 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$17\times8=136$ 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

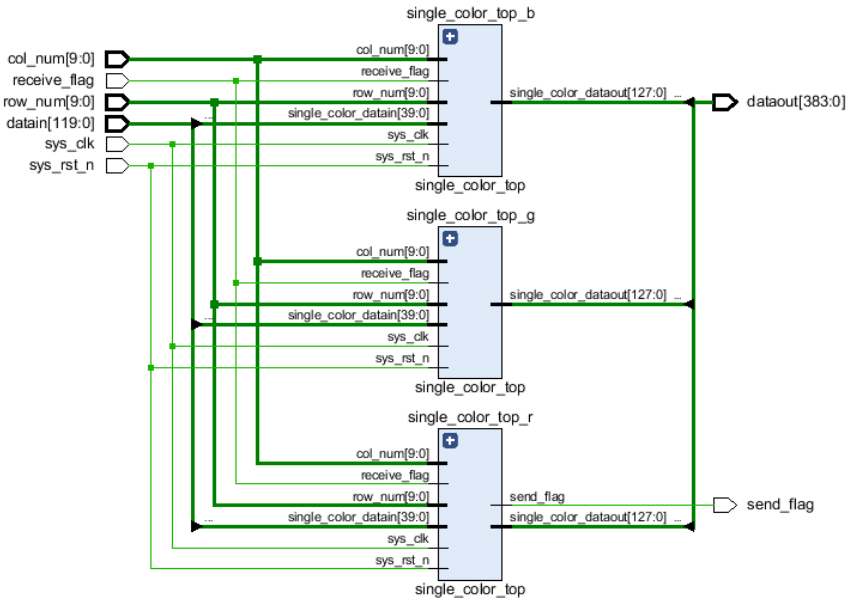

图 4.1 top 模块框图

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

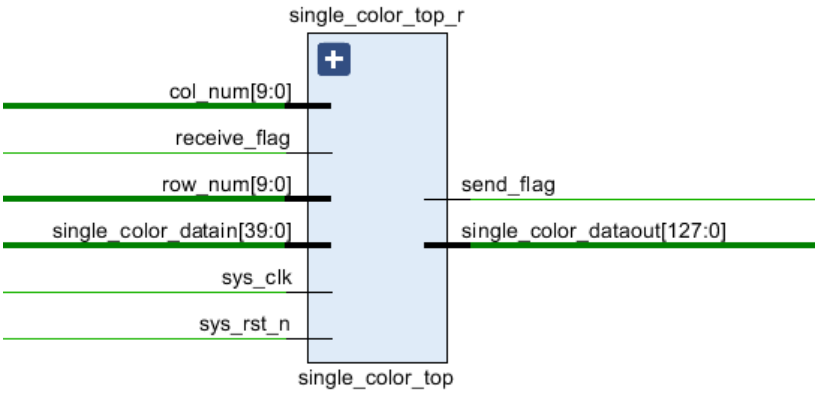

图 4.2 single_color_top 模块框图

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

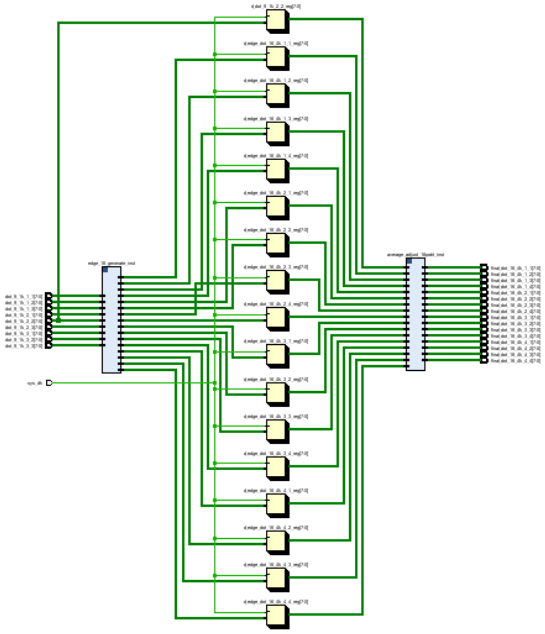

图 4.3 interpolate_16point 模块框图

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

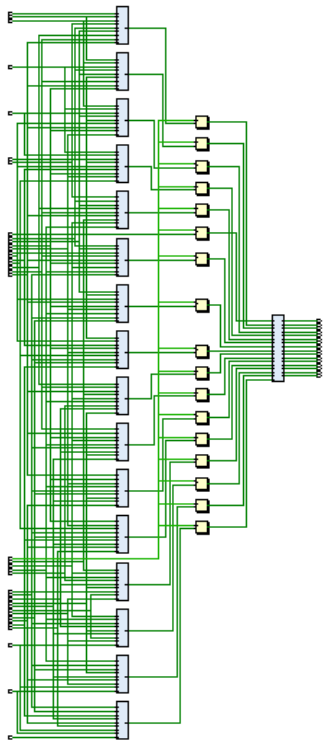

图 4.4 gauss_filter_dot_36_4k 模块框图

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

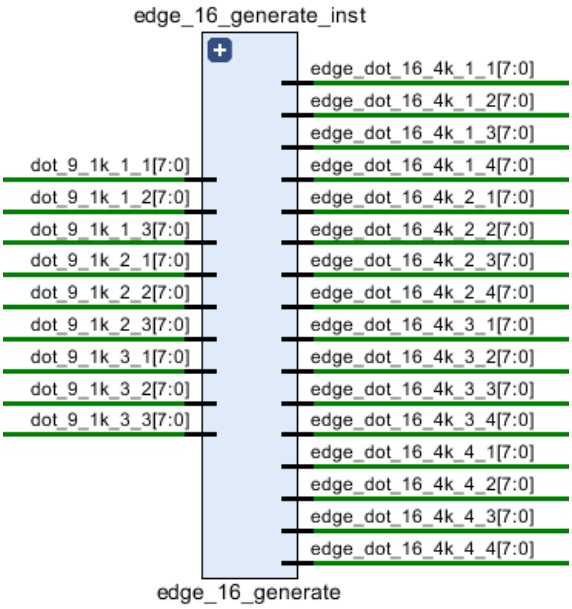

图 4.5 edge_16_generate k 模块框图

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

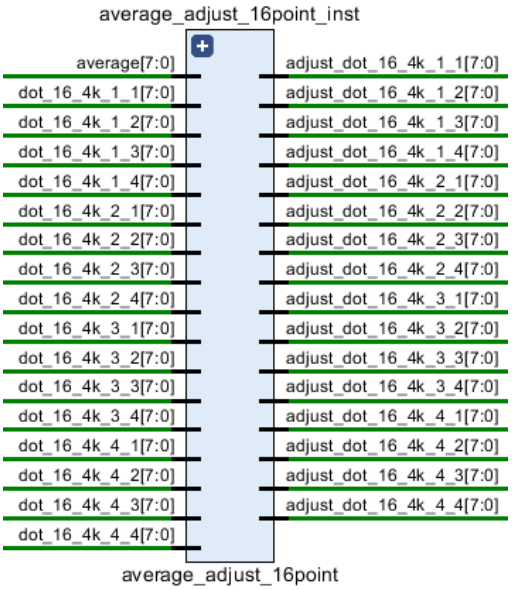

图 4.6 average_adjust_16point 模块框图

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。

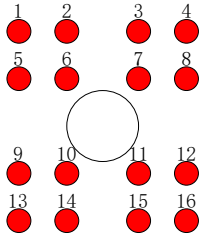

图 5.1 生成 4k 数据传输顺序

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：

```
integer fid;
initial begin
    $readmemb("C:/Users/master/Desktop/image_process/vivado/ram_1k_data.v",ram_1k_data);
    fid=$fopen("C:/Users/master/Desktop/image_process/vivado/ram_4k_data.v");
    sys_clk=1'b0;
    sys_rst_n<=1'b0;
    #20
    sys_rst_n<=1'b1;
end
```

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。

```
//将生成的数据写入ram_4k_data
integer j;
always@(posedge sys_clk) begin
    if(save_picture_finish_flag==1'b1) begin
        $fclose(fid);
        $finish;
    end
    else if(col_num>=3'd4&&d_send_lasting_flag==1'b1) begin
        for(j=0;j<8;j=j+1) begin
            $fwrite(fid,"%b",dataout[7-j]);
        end
        $fwrite(fid,"\n");
    end

end
```

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。

```
single_color_top  single_color_top_inst
(
    .sys_clk            (sys_clk            )    ,
    .sys_rst_n          (sys_rst_n          )    ,
    .col_num            (col_num            )    ,
    .row_num            (row_num            )    ,
    .single_color_datain (single_color_datain  )    ,//二维图像一列的元素最上边像素位于该变量的高地址
    .receive_flag       (receive_flag       )    ,

    .send_flag          (send_flag          )    ,
    .single_color_dataout(single_color_dataout )
);
```

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。

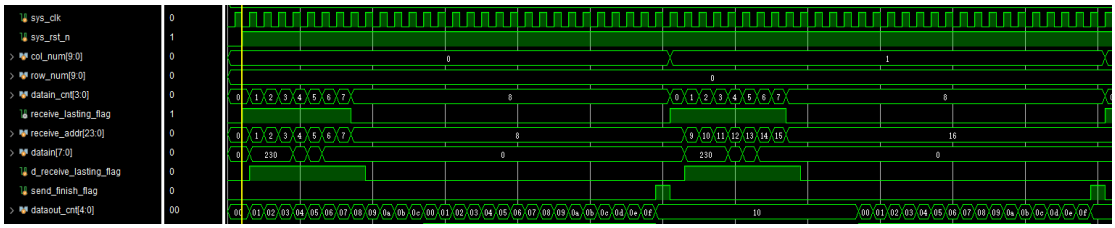

图 5.5 开始读取 1k 数据波形图

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。

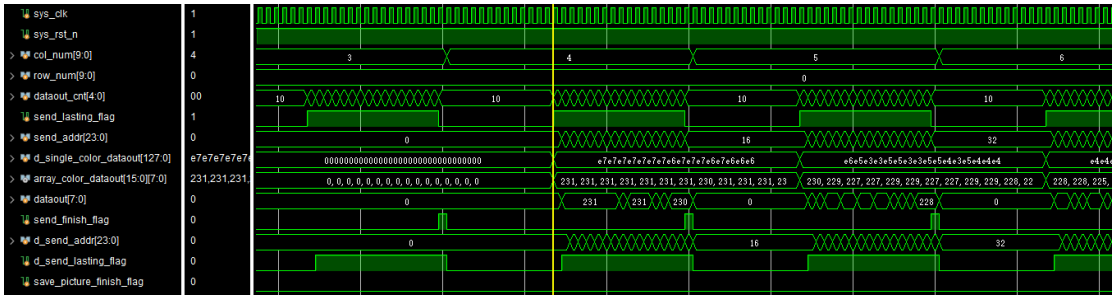

图 5.6 开始存储 4k 数据波形图

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。

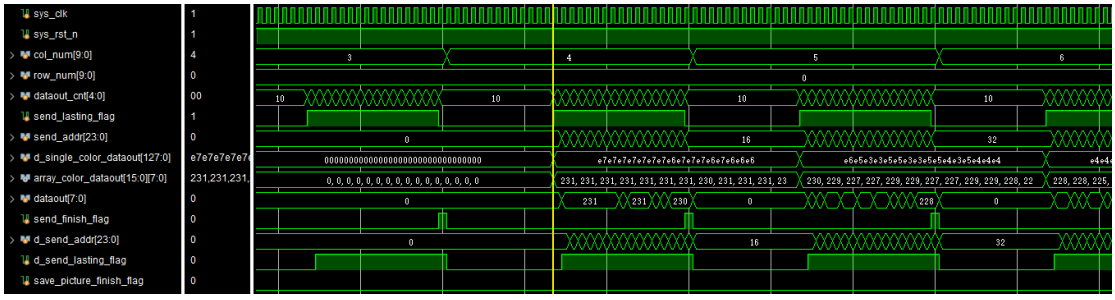

图 5.7 1k 图片插值完成波形图

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 $3840\times2160=8294400$ 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。

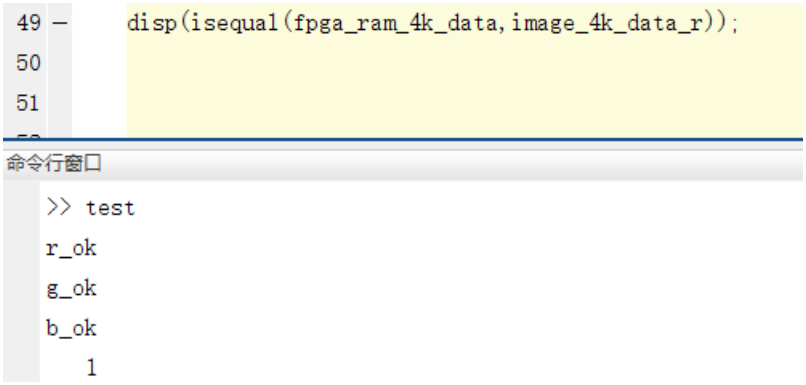

图 5.8 仿真产生图片与软件产生图片一致截图

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

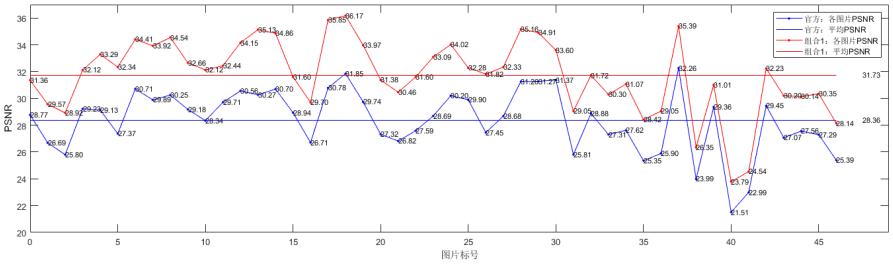

图 6.1 组合 1 PSNR 性能

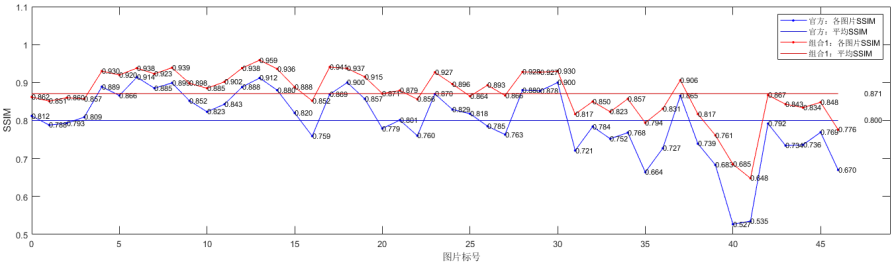

图 6.2 组合 1 SSIM 性能

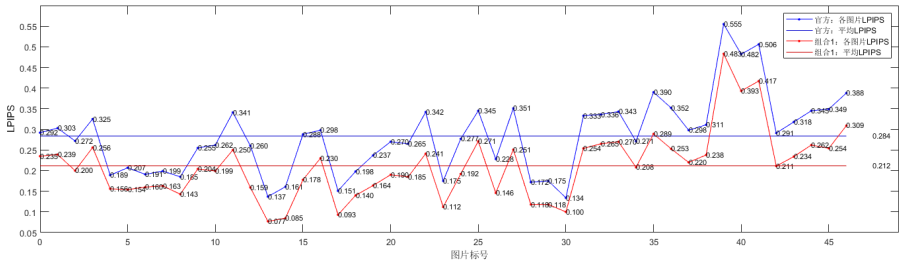

图 6.3 组合 1 LPIPS 性能

组合 2：新型双线性插值+均值调整+高斯平滑+均值调整

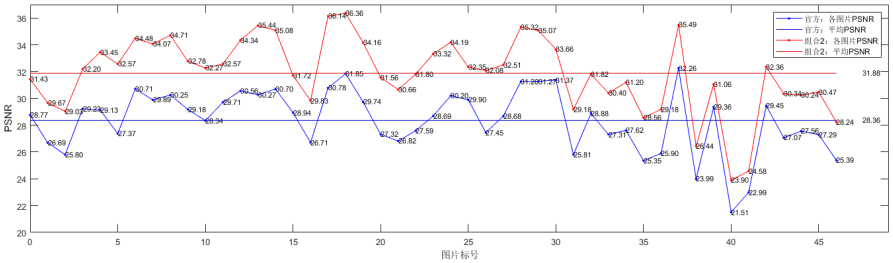

图 6.4 组合 2 PSNR 性能

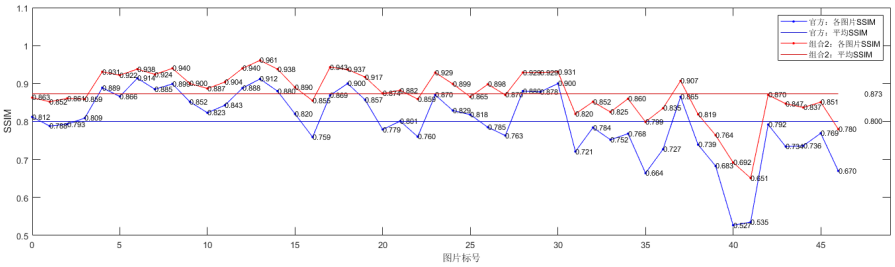

图 6.5 组合 2 SSIM 性能

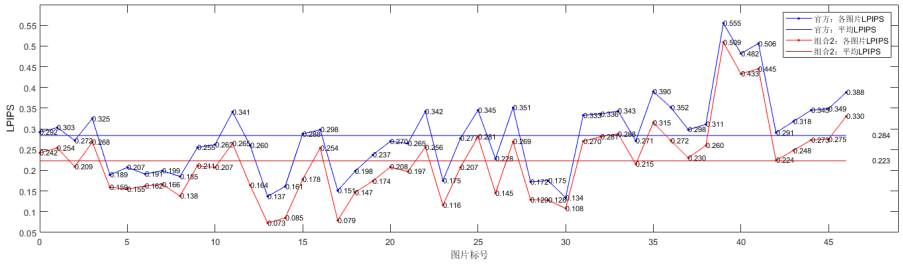

图 6.6 组合 2 LPIPS 性能

组合 3：分区域插值+均值调整+高斯平滑+均值调整

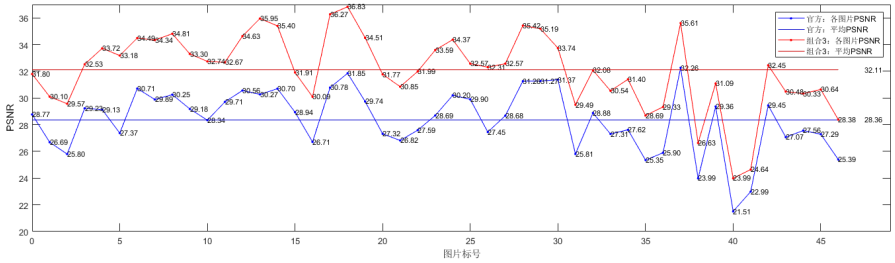

图 6.7 组合 3 PSNR 性能

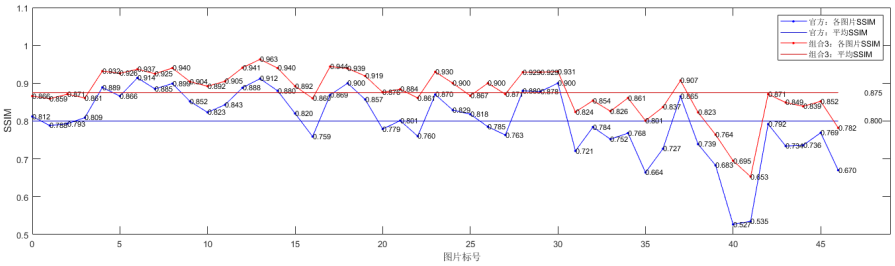

图 6.8 组合 3 SSIM 性能

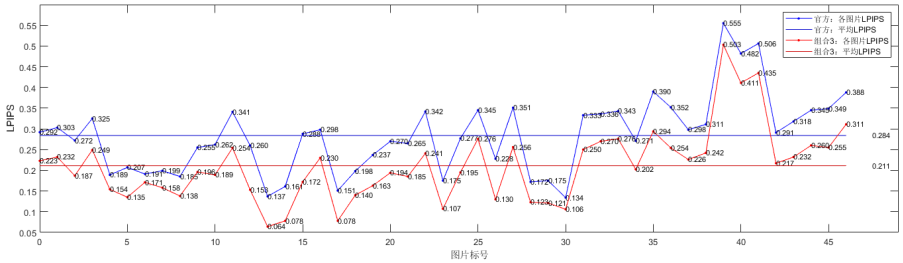

图 6.9 组合 3 LPIPS 性能

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

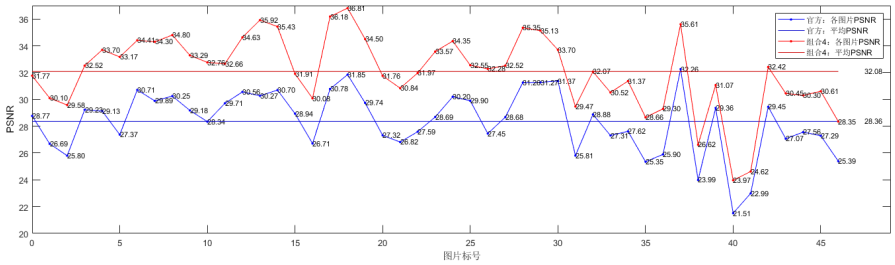

图 6.10 组合 4 PSNR 性能

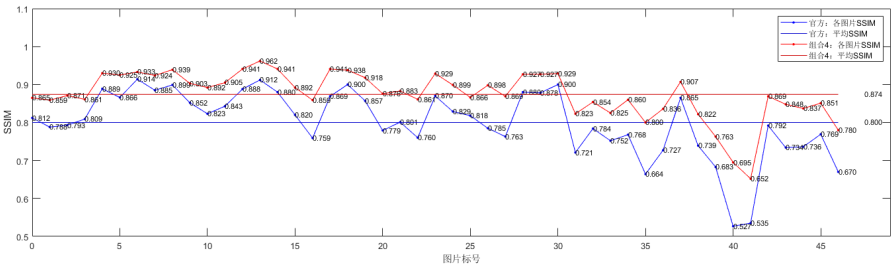

图 6.11 组合 4 SSIM 性能

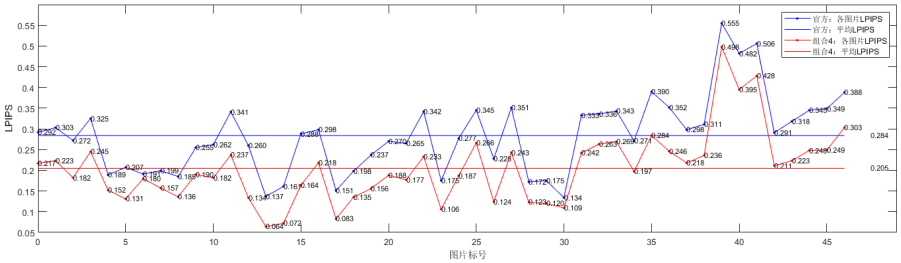

图 6.12 组合 4 LPIPS 性能

组合 5：3 区域均采用边缘插值+均值调整+高斯平滑

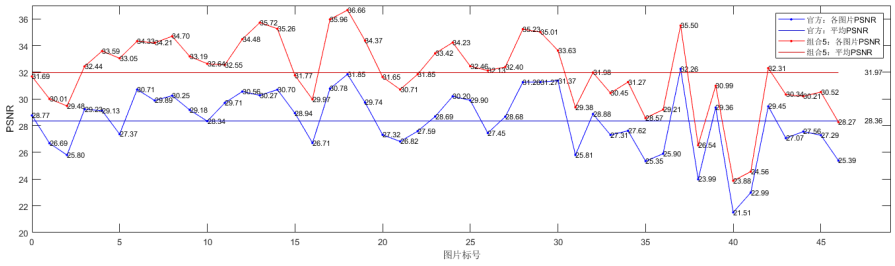

图 6.13 组合 5 PSNR 性能

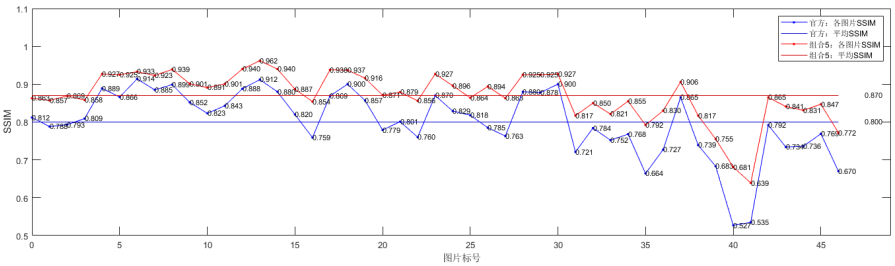

图 6.14 组合 5 SSIM 性能

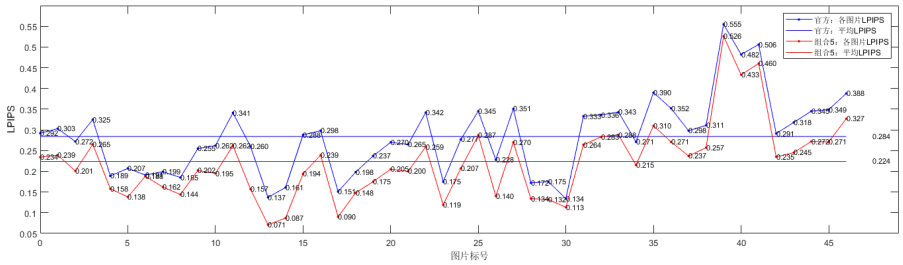

图 6.15 组合 5 LPIPS 性能

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。

Resource	Utilization	Available	Utilization %
LUT	11220	53200	21.09
FF	3586	106400	3.37

图 6.16 顶层资源消耗

Name	Slice LUTs (53200)	Slice Registers (106400)	Bonded IOB (125)	BUFGCTRL (32)
top	11220	3586	516	1
single_color_top_b (single_color_top)	3048	1199	0	0
gauss_filter_dot_36_4k_inst (gauss_filt...	848	136	0	0
interpolate_16point_inst (interpolate_16...	1663	136	0	0
single_color_top_g (single_color_top_0)	2592	1192	0	0
gauss_filter_dot_36_4k_inst (gauss_filt...	848	136	0	0
interpolate_16point_inst (interpolate_16...	1663	136	0	0
single_color_top_r (single_color_top_1)	2592	1195	0	0
gauss_filter_dot_36_4k_inst (gauss_filt...	848	136	0	0
interpolate_16point_inst (interpolate_16...	1663	136	0	0

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 3×3 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 $40\times16=640$ 。

	LUT资源消耗	使用率
顶层模块	11220	21.09%
新型双线性插值模块	837	1.57%
边缘插值模块	1815	3.41%
均值调整模块	605	1.14%
高斯平滑模块	40	0.08%

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。

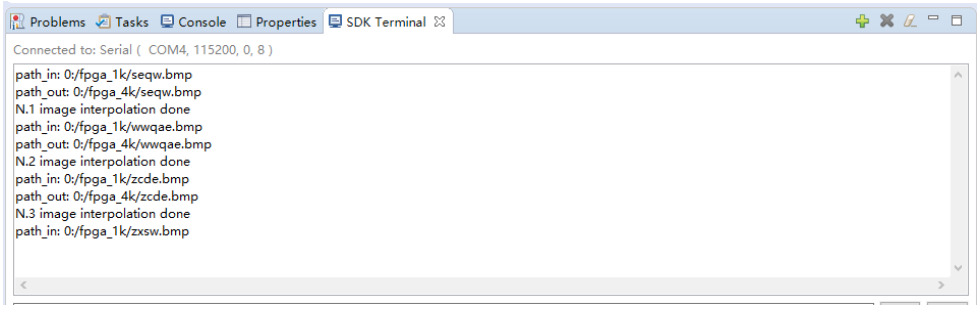

图 7.1 插值处理过程

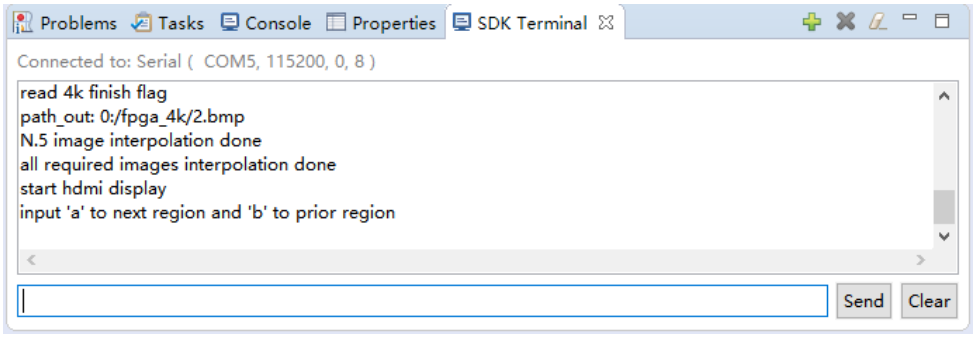

图 7.2 插值完成开始 hdmi 显示

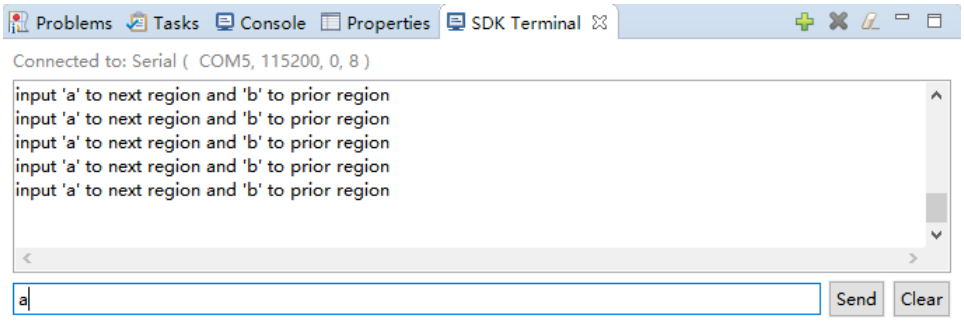

图 7.3 输入字符控制显示区域

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。

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。

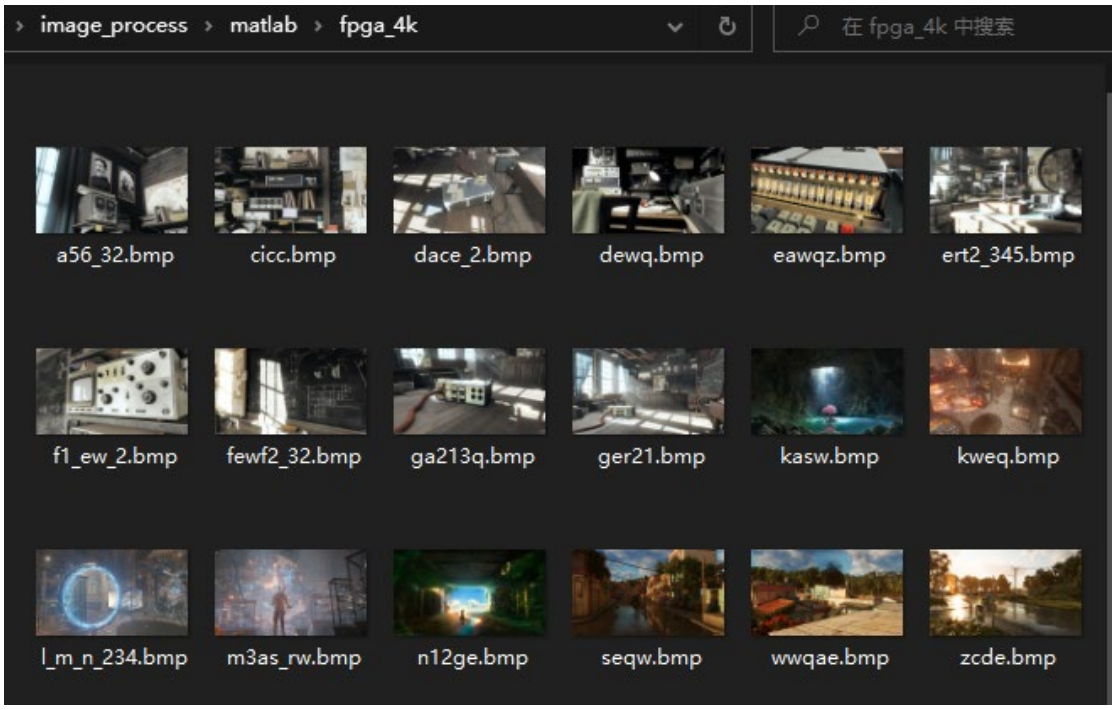

图 7.5 FPGA 生成 4k 图片截图

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。