智能停车场车牌识别与管理系统

田修诚　张海航　陈一鹏

摘要

随着科技的迅猛发展，尤其是智能交通系统和物联网（IoT）的普及，车牌识别技术和相关的车载设备变得越来越智能化。FPGA 作为一种高度灵活且性能强大的硬件技术，能够在高并发和低延迟的条件下进行图像处理和数据处理。它在实时性、处理能力、灵活性和可拓展性方面的优势是其他器件无可比拟的。

本队伍的作品是基于**图像处理、卷积神经网络**等技术开发出的一套基于车牌识别的智能停车场管理系统。本系统采用 ov5640 作为图像采集模块。通过将 ov5640 的视频流信号进行转 YCbCr、二值化、腐蚀运算、Soble 边缘检测等一系列图像处理，得到关于车牌边框的定位以及字符的分割。之后利用**卷积同或模板匹配**进行字符的判断，首先通过卷积核以及池化核对识别的字符像素提取特征矩阵并进行稀疏化处理，在保证数据特征的情况下降低匹配难度。同时本系统的中间过程图像在卷积加速后进行**分行 Block RAM 存储**，减少传统的 DDR3 存储的时间消耗, 使系统处理速度提升了 **264.40%**。之后对稀疏化的特征矩阵与标准特征矩阵进行同或模板匹配得到识别的字符，最终的车牌定位和字符分割和识别结果通过 **LCD 显示屏**进行输出显示。

本系统有如下三点创新:(1) 针对识别结果出现相近字符抖动的情况, 我们提出了 **2-Z 模糊字符智能判定算法**。通过检测字符水平投影的差异, 对 2-Z 字符进行二次判定，有效地减少了相近字符的识别抖动，对 2 和 Z 的识别准确率提高了 **275.03%**。(2) 本系统的另一大特点是加入**距离省电模块**，将超声波测距传感器与 FPGA 开发板综合。通过对车辆的测距，在车辆驶入测量范围内时启动工作模式，进行车牌的识别；待机时只有超声波传感器工作。通过两种系统状态控制整个 FPGA 开发板工作，减小 FPGA 芯片资源的无效占用，从而降低整个系统硬件的维护成本。(3) 本系统创造性的使用智能交互模块将车牌识别的结果与提前录入的车牌库进行比对，控制舵机的转动，实现智能控制的功能。该模块的可拓展性强，未来我们会针对使用场景进行蓝牙、物联网等多功能的拓展，使其能够更好地应用在智能交通领域。

经过多个性能参数测试，本文提出的智能停车场管理系统在识别准确率、算法加速、省电出发精度等方面表现良好，稳定可靠，对未来的进一步拓展设计具有较大的意义。

关键字：　距离省电控制　卷积存储加速　卷积同或模板匹配　2-Z 模糊字符二次判定

目录

一、 FPGA 设计概述 …… 3

1.1 设计目的 …… 3

1.2 应用领域 …… 3

1.3 主要技术特点 …… 4

1.4 关键性能指标 …… 4

1.5 主要创新点 …… 4

二、 系统组成及其功能说明 …… 5

2.1 整体介绍 …… 5

2.2 图像采集与 LCD 输出模块 …… 6

2.3 图像处理模块 …… 7

2.3.1 RGB565 转 YCbCr444 …… 7

2.3.2 腐蚀运算算法: …… 9

2.3.3 Sobel 边缘检测算法: …… 11

2.3.4 膨胀算法: …… 12

2.3.5 卷积运算识别模块 …… 13

2.4 Z 与 2 模糊字符智能判定 …… 15

2.5 距离省电控制模块 …… 17

2.6 库识别与智能交互模块 …… 18

2.7 算法加速模块 …… 19

三、 完成情况及性能输出参数 …… 21

3.1 车牌基本情况介绍 …… 21

3.2 初步测试 …… 21

3.3 性能参数 …… 22

3.3.1 测试 1：车牌识别系统稳定性 …… 23

3.3.2 测试 2: 含有 2 和 Z 的车牌识别: …… 24

3.3.3 测试 3: 省电模式触发精度: …… 24

3.3.4 测试 4:FPGA 硬件加速效果: …… 25

3.3.5 测试 5: 智能车牌库读写精度: …… 25

3.3.6 测试 6: 低速运动车辆识别: …… 26

3.3.7 测试 7: 车牌的三维旋转倾斜识别: ····· 27
3.3.8 测试 8: 低环境亮度识别: ····· 28
四、 总结 ····· 28
4.1 可拓展之处 ····· 28
4.2 心得体会 ····· 29
参考文献 ····· 29
A 附录 代码 ····· 30

一、 FPGA 设计概述

1.1 设计目的

本系统的设计是为了能够自动化地从图像或视频流中识别和提取车牌信息从而进行智能化控制管理。该系统通过车辆图像的采集、车牌位置的确定以及字符区域的分割，从中划分出车牌字符读取车牌号码，从而实现对车辆的快速识别和管理。它利用计算机视觉和模式识别技术，通过预设的特征值与输入的车牌候选区域字符进行卷积同或模板匹配, 准确识别车牌上的字符。同时基于 FPGA 设计的车牌识别系统巧妙利用了 FPGA 的可编程性、可移植性以及高速计算性, 使其识别的速度显著快于传统情况的车牌识别, 并具有较强的拓展性。同时 FPGA 车牌识别模块的设计考虑了实际应用中的多种情况，如不同角度拍摄、光暗条件限制等，以确保在各种环境下都能稳定工作。

1.2 应用领域

i 智能停车场管理:
现有的停车场中, 车辆在进出时往往还需要停下等待系统响应和车牌识别。在智能停车场中,FPGA 快速车牌识别技术可实现车辆免停快速出入, 并进行车位引导、自动计费等功能的拓展, 大大提高了停车场的停车效率, 减少了不必要的等候。

ii 电子监控:
车牌识别技术可用于公安和交警系统的智能电子监控识别领域，通过布设在关键路段的摄像头和 FPGA 快速车牌识别系统, 实现对过往车辆的自动抓拍和记录，将车辆违法行为进行记录, 并将所识别的车牌进行网络联网，方便公安部门进行嫌疑车辆的自动追踪, 为公安和交警工作开展提供有力支撑。

iii 高速公路收费管理:
在高速公路车辆管理中, 车牌识别技术得到广泛应用, 尤其是在电子收费 (ETC) 系统中, 可以应用 FPGA 快速车牌识别系统作为识别车辆身份的主要手段, 保障车辆在 ETC 收费口的快速驶过, 减小高峰期收费站的压力。

FPGA 的可编程能力会允许我们未来针对此系统进行进一步改进, 将车牌同电子标签结合, 真正实现针对车牌、汽车的智能控制和联网管理系统。

1.3 主要技术特点

图像处理和模式识别技术：车牌识别技术基于计算机视觉与模式识别技术, 通过图像处理算法对车辆图像进行分析, 提取车牌号码及颜色信息。

自动化和高效率：该技术能够自动化地从车辆图像中提取车牌信息, 大大提高了交通管理和监控的效率。

多步骤识别流程：包括图像采集、预处理、车牌定位、字符分割和字符识别等关键步骤, 确保了识别过程的准确性和可靠性。

深度学习算法的应用：深度学习技术, 尤其是卷积神经网络 (CNN), 成功实现标准特征值与图像中字符的特征值进行相似度匹配, 在字符识别中发挥着重要作用, 提高了识别准确率。

1.4 关键性能指标

车牌识别系统的关键性能指标主要包括以下几个方面:

识别准确率: 这是衡量车牌识别系统性能的重要指标之一，指的是特征信息识别正确的图片数与总有效图片数的百分比。它反映了系统在识别车牌号码时的准确性，包括日间和夜间的识别准确率。

识别速度: 识别速度是指自动识别软件读取机动车图像到给出识别结果所需的时间。快速的识别速度对于需要实时处理的应用场景尤为重要, 如停车场出入口管理等。

鲁棒性: 鲁棒性是指系统在各种环境和条件下保持性能稳定的能力。评价鲁棒性的指标包括抗噪性、抗模糊性、抗遮挡性和抗光照变化性。

实时性: 实时性衡量了系统处理车牌图像并识别出车牌号所需的时间。这对于需要快速响应的应用场景非常重要, 如交通监控和停车场管理。

1.5 主要创新点

我们的创意在于利用现场可编程门阵列的并行处理优势，实现对车牌图像的快速、高效识别。该系统通过高清摄像头捕捉车辆图像，并运用 FPGA 的高速并行计算能力，对图像进行实时处理和分析。创意的核心在于将图像预处理、车牌定位、字符分割和识别等复杂计算任务分配给 FPGA 的不同逻辑模块，同时执行，从而大幅缩短识别时间。

(1). 2 与 Z 模糊综合判定算法

在车牌识别的过程中，相似字符的识别往往是一大难题，在车牌标准的制定中已经排除了一部分相似字符，如”O” 和”0”、”1” 和”I”。但是”2” 与”Z” 的识别仍是一大难点，

对此，基于卷积同或模板匹配的基础上，我们引入了 2 与 Z 模糊综合判定算法。在识别 2 与 Z 的过程中，单一的卷积同或特征模板匹配会引起字符识别在 2 与 Z 之间不断地抖动，影响最终的识别效果。而 2 与 Z 模糊综合判定算法是基于 2 与 Z 的竖直投影的峰值差异，进一步对其进行判断。并考虑到图像采集的角度对识别相似字符的影响，利用三维旋转进行不同角度的模拟，使算法的识别更加精准。

(2). 近距离启动与省电设计

本系统创造性添加了超声波模块，将整个车牌识别系统划分为待机状态和工作状态，通过超声波测距来实现判断是否有车辆经过，在没有车辆经过时系统处于待机状态，此时只有超声波模块在工作，并不进行图像的采集和处理。当超声波检测到车辆距离系统较近时，会自动传入一个启动信号，系统接收到信号后由待机状态转为工作状态，迅速启动图像的处理。这样保证了系统在没有车辆经过时仅工作超声波这一模块，有效降低整个系统的能耗。同时防止 FPGA 在长时间地进行图像处理中产生不必要的算力资源浪费，也解决了 FPGA 长时间处理的过热问题。

(3). 智能车牌库识别

考虑到本系统在停车场等应用场景的实用性，我们在系统中添加了智能车牌库识别模块，即将部分车牌存入车牌库中，作为该应用场景中的识别部分，当车辆进入激光测距模块的范围中时，车牌库识别模块会启动，将车牌识别后的结果与车牌库中的部分进行比对，如果符合车牌库中的部分就会输出“通过”信号，反之则输出“不通过”信号，实现了 FPGA 与现实生活的交互，同时，也可以对于库中的车牌进行添加与删除，增加了本系统在日常生活中的实用性。我们在检验中通过对于车牌库中部分车牌的添加与删除，进而观察舵机的状态，验证了本智能车牌库识别模块在 FPGA 中的可实践性。

二、 系统组成及其功能说明

2.1 整体介绍

本系统主要包括图像采集模块、图像处理模块、智能省电模块以及人机交互模块，系统模块的整体框架如图所示：

本系统使用的使 AMD 厂商制作的 xc7a100tfgg484 芯片，基于此芯片通过 VIVADO 软件进行 FPGA 的开发与设计，实现设想功能。我们使用的开发板是搭载该芯片，由正点原子设计制造的“达芬奇 Pro 开发板”，开发板如下图所示：

本智能车牌识别系统系统所用到的硬件有：达芬奇 Pro 开发板、ov5640 摄像头、HC-SR04 超声波传感器、SG90 9G 舵机、800*480:RGB LCD 模块以及各类接口线构成。具体如图所示：

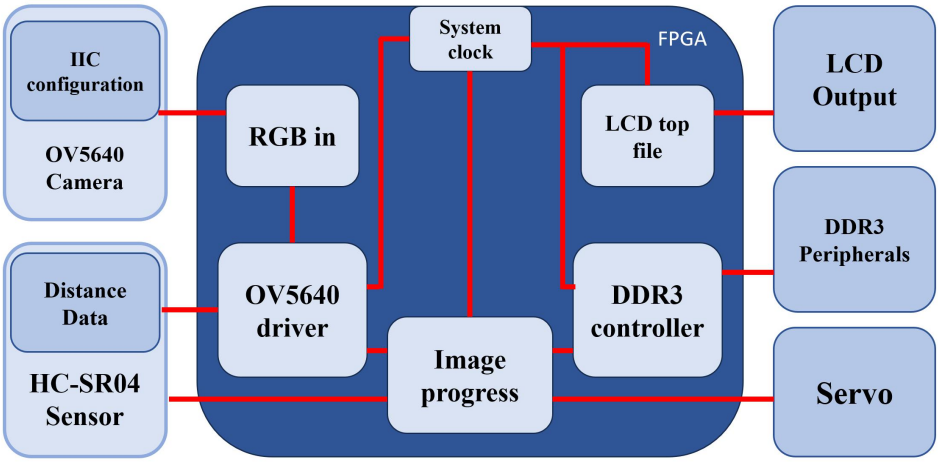
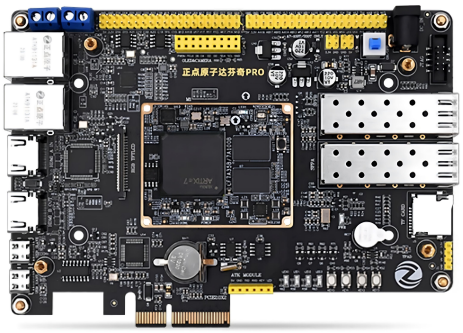
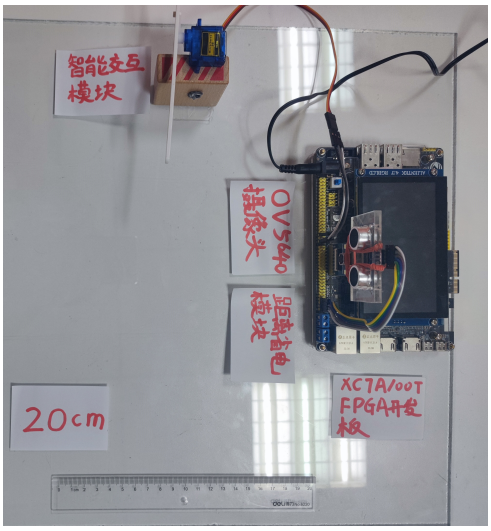


图 **1**　系统框架图



(a) 开发板展示



(b) 硬件全貌

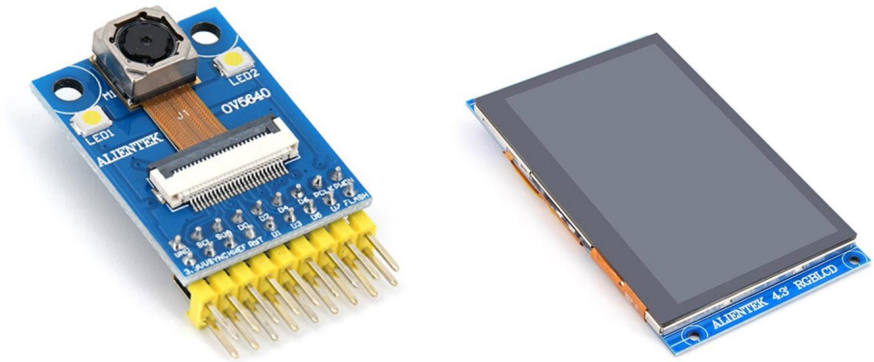
图 **2**　系统硬件

2.2 图像采集与 LCD 输出模块

我们采用 ov5640 摄像头作为图像采集模块使用 DVP 接口与 FPGA 连接实现图像的传输，并通过 I2C 总线进行配置, 将 ov5640 的图像数据直接传输给 FPGA 芯片，FPGA 将接收到的图像数据存储到 DDR3 中，通过直接读取 DDR3 中的数据进行 LCD 显示。

该系统使用的使 4.8 寸 LCD 显示屏作为输出显示模块, 我们将 LCD 模块通过并行接口与 FPGA 连接, 并通过 FPGA 芯片生成控制信号与 LCD 模块通信, 利用 FPGA 发送 ov5640 采集的视频流数据以及车牌识别结果到 LCD 模块, 以达到效果展示的功能。

下图是摄像头和 LCD 屏幕展示：



(a) 摄像头　　**(b) LCD**

图 **3**　图像采集与 **LCD** 输出模块

2.3 图像处理模块

图像处理模块为系统的重点模块, 在作用上分为车牌定位、字符分割以及字符特征提取与识别三部分。车牌定位与字符分割主要通过 RGB56 5 转 YCbCr444、腐蚀运算、Sobel 边缘检测等算法来实现。字符特征提取与识别主要通过卷积同或模板匹配来实现。下图是通过 MATLAB 仿真演示的车牌定位与字符分割的图像处理流程图。

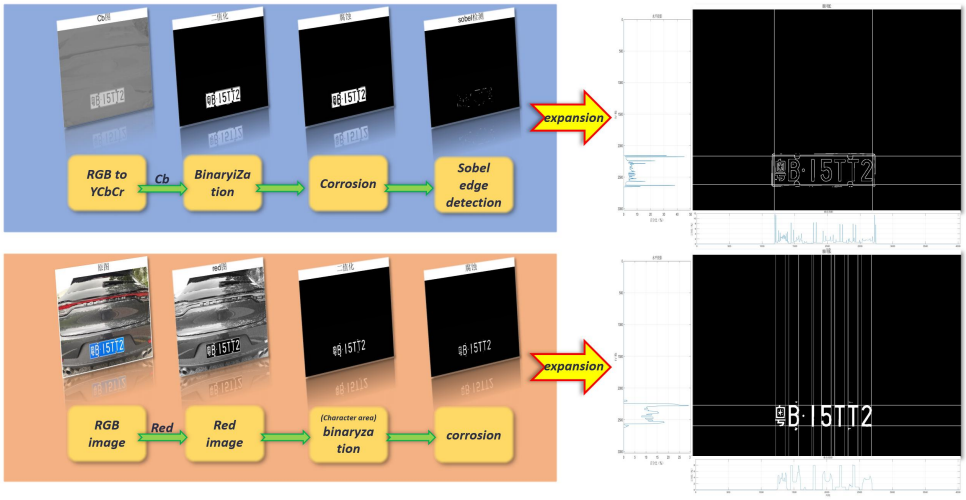


图 **4**　图像处理流程

2.3.1 RGB565 转 YCbCr444

在上面的摄像头图像采集模块我们生成了关于图像的 RGB565 视频流, 想要进一步转化为 YCbCr444。考虑到 YCbCr444 中的 Y、Cb、Cr 采样率格式一样, 要将 RGB565 进一步转化, 首先需要将 RGB565 中的红色值左移三位得到八位的红色分量, 同理，控制绿色值和蓝色值进行移位生成八位的分量，使其红 (R) 绿 (G) 蓝 (B) 三色的分量值都为

8Bit, 成功转化为 RG B888 的视频流。

之后我们需要将得到的 RGB888 格式为 YCbCr444 格式的视频流, 其中 YCbCr444 是亮度 Y、蓝色色度 Cb 与红色色度 Cr 的颜色空间, 转换的实质是色度空间的转换, 通过查询 ov5640 的技术手册我们可以得知 RGB888 转 YCbCr 的公式如下所示:

$$Y = 0.299R + 0.587G + 0.114B$$

$$Cb = 0.568(B - Y) + 128 = -0.172R - 0.339G + 0.511B + 128$$

$$Cr = 0.713(R - Y) + 128 = 0.511R - 0.428G - 0.083B + 128$$

但是由于 Verilog 无法进行数据的浮点运算, 我们将数据扩大 256 倍, 再采用右移 8Bit 的方式进行公式的转换, 得到如下公式:

$$Y = ((77R + 150G + 29B) >> 8)$$

$$Cb = ((-43R - 85G + 128B) >> 8) + 128$$

$$Cr = ((128R - 107G - 21B) >> 8) + 128$$

在算法移植前, 为了防止在 Verilog 程序运行中出现负数, 我们先将 128 提取到括号内, 所得的公式如下所示：

$$Y = (77R + 150G + 29B) >> 8$$

$$Cb = (-43R - 85G + 128B + 32768) >> 8$$

$$Cr = (128R - 107G - 21B + 32768) >> 8$$

此时 Y 必然为正值, 同时 Cb 括号内最小值为:256*1。因此 Cb 括号内必然大于 0, 同样可以推断出 Cr 必然大于 0。经过上述变换后, 可以进行算法移植, 而不用考虑运算结果溢出以及结果不符合程序运算规则的问题。

图 5　**RGB565 转 YCbCr888 的 MATLAB 仿真**

RGB565 转 YCbCr444 算法的 HDL 实现代码见附录所示。

灰度图像二值化算法

我们在获得 YCbCr444 格式的视频流后, 与要进一步将其处理为黑白格式的视频流, 这里我们使用局部阈值法对获取的信号亮度 Y 进行二值化处理。

局部阈值法是图像二值化的一种高级的方法，它克服了全局阈值法对光照不均匀图像处理的不足。与全局阈值法使用单一阈值不同，局部阈值法会根据图像每个像素的邻域信息动态计算阈值。

局部阈值法将图像分成若干个小区域（窗口），并为每个区域单独计算阈值。这样，即使图像的光照不均匀，每个区域也能根据自身的亮度分布选择合适的阈值，从而获得更准确的二值化结果。设为原始灰度图像，为二值化后的图像，为像素处的局部阈值，则二值化过程可以表示为：

$$g(x,y)=\begin{cases}1 & f(x,y)>T(x,y)\\ 0 & f(x,y)\le T(x,y)\end{cases}$$

在这里，我们使用高斯加权均值法计算局部阈值的大小，如下所示：

$$T(x,y)=\sum G(i,j)*f(i,j)/\sum G(i,j)$$

$$G(i,j)\text{为高斯函数在像素}(x,y)\text{处的值。}$$

灰度图像二值化算法的具体 Verilog HDL 实现代码见附录所示。

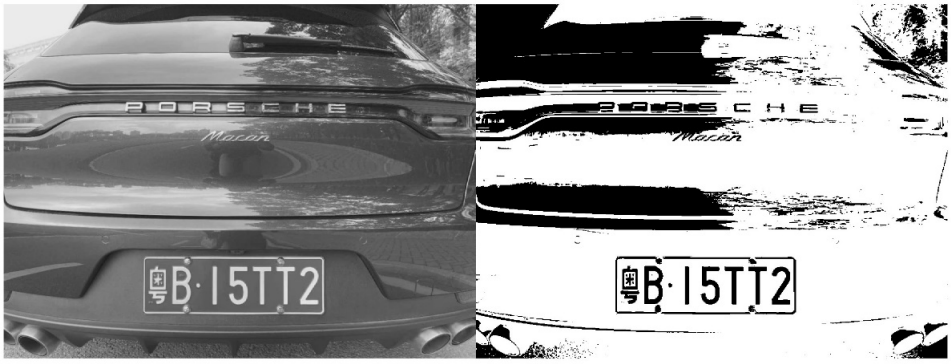


图 **6**　局部阈值法对灰度图像二值化算法的 **MATLAB** 仿真

2.3.2 腐蚀运算算法:

腐蚀运算是利用结构元素的预定义形状来探测图像。结构元素通常是一个小的二值图像，其中像素值为 1 的像素表示结构元素的“前景”，像素值为 0 的像素表示结构元素的“背景”。这里我们用 3*3 像素阵列来通俗的解释:

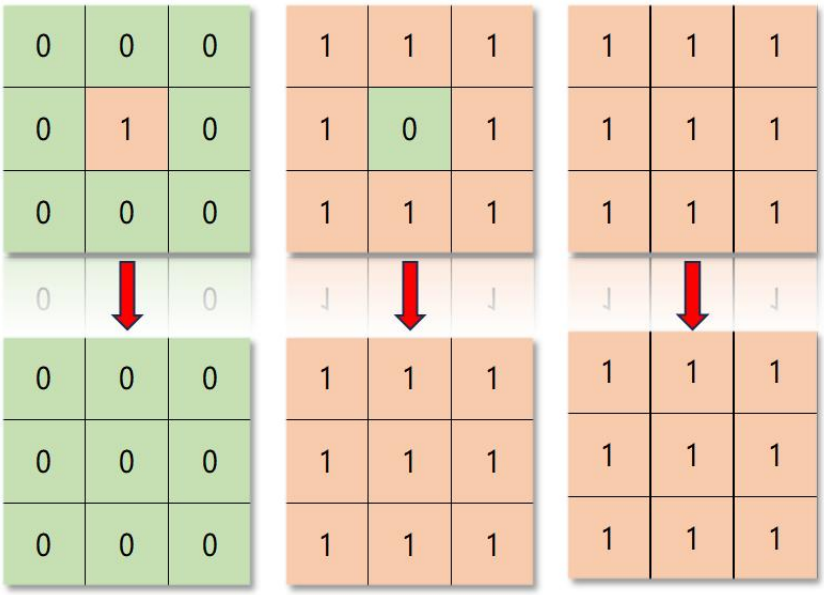


图 **7**　腐蚀运算示意图

关于算法的实现，可以用下式子来表示，即 3*3 像素的与运算：

$$P = P_{11}\&P_{12}\&P_{13}\&P_{21}\&P_{22}\&P_{23}\&P_{31}\&P_{32}\&P_{33}$$

在 Verilog HDL 中，为了通过面积去换取速度，我们将上式改变如下：

$$\begin{cases} P1 = P_{11}\&P_{12}\&P_{13} \\ P2 = P_{21}\&P_{22}\&P_{23} \\ P3 = P_{31}\&P_{32}\&P_{33} \end{cases}$$

通过 2 个时钟的算法, 实现腐蚀运算的结果:
腐蚀运算算法的 Verilog HDL 实现代码见附录所示。

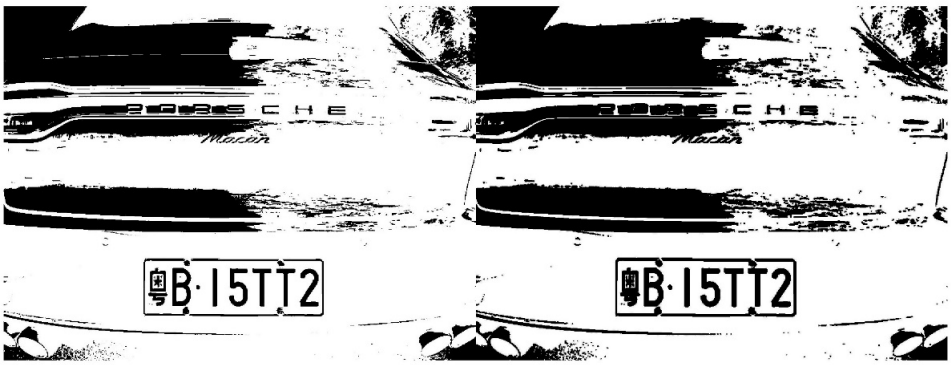


图 **8**　腐蚀运算 **MATLAB** 仿真

2.3.3 Sobel 边缘检测算法:

我们使用索贝尔算子 (Sobel operator) 作边缘检测，在技术上，它是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。在图像的任何一点使用此算子，将会产生对应的灰度矢量或是其法矢量。

Sobel 卷积因子为:

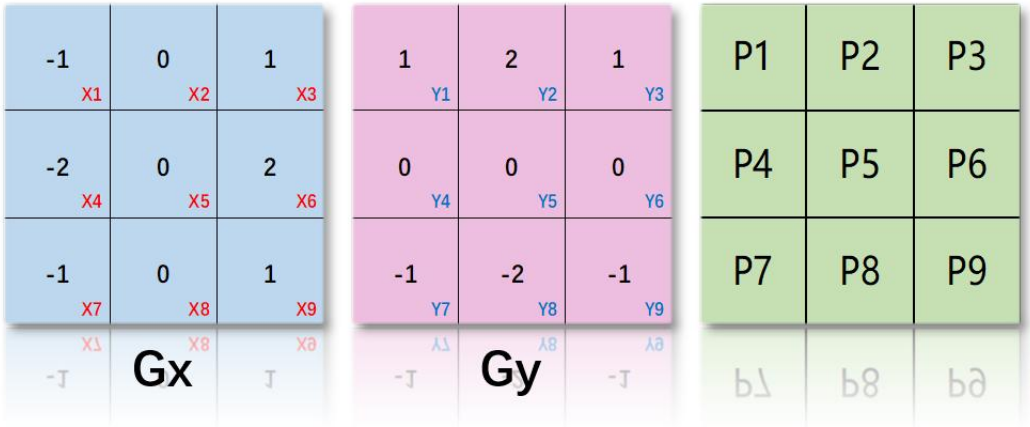


图 9　**Sobel** 卷积因子

该算子包含两组 3x3 的矩阵, 分别为横向及纵向, 将之与图像作平面卷积, 即可分别得出横向及纵向的亮度差分近似值。如果以 A 代表原始图像,Gx 及 Gy 分别代表经横向及纵向边缘检测的图像灰度值, 其公式如下：

$$G_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * A \quad and \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

图像的每一个像素的横向及纵向灰度值通过以下公式结合，来计算该点灰度的大小：

$$G = \sqrt{G_x^2 + G_y^2}$$

如果梯度大于某一阀值, 则认为该点 (x,y) 为边缘点。

以下是 Verilog HDL 代码示例:

```
if(temp3 > THRESHOLD)
IMG_Sobel(i,j) = 0; //Black
else
IMG_Sobel(i,j) = 255; //White
end
```

然后利用以下公式计算梯度方向 (只要检测边缘，则不用计算方向):

$$\theta = \tan^{-1}\left(\frac{G_X}{G_y}\right)$$

Sobel 边缘检测算法的 HDL 实现代码见附录所示。

图 **10**　**Sobel** 边缘检测算法 **MATLAB** 仿真

2.3.4 膨胀算法:

膨胀算法是一种图像处理的基本操作，主要用于扩大图像中的目标区域。它可以用来填补目标区域中的孔洞、连接相邻的目标区域，以及增强目标的特征。

膨胀运算与腐蚀运算相对，其工作原理是将结构元素在图像上滑动，并将结构元素的原点与图像上的每个像素对齐。对于每个像素位置，如果结构元素覆盖的区域中至少有一个像素值为“1”(或“白色”)，则将该像素设置为“1”。

这里我们仍旧使用 3*3 像素阵列来通俗的解释：

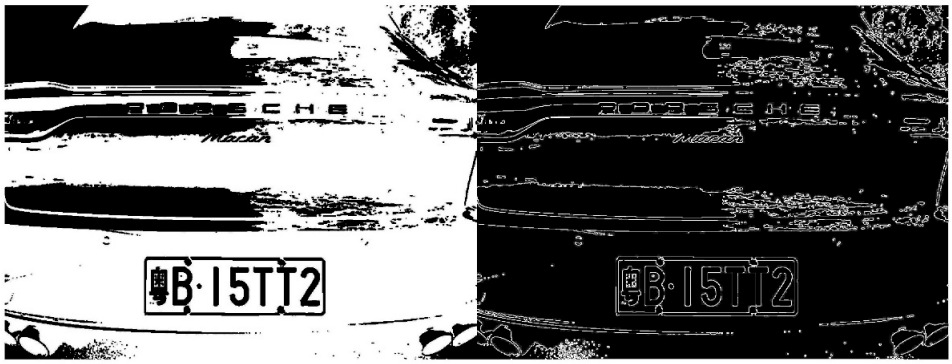


图 **11**　**Sobel** 边缘检测算法 **MATLAB** 仿真

关于算法的实现，可以用下式子来表示，即 3*3 像素的与运算：

$$P = P_{11} \mid P_{12} \mid P_{13} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{31} \mid P_{32} \mid P_{33}$$

跟腐蚀算法类似，在 Verilog HDL 中，为了通过面积去换取速度，我们将上式改变如下：

$$\begin{cases} P_1 = P_{11} \mid P_{12} \mid P_{13} \\ P_2 = P_{21} \mid P_{22} \mid P_{23} \\ P_3 = P_{31} \mid P_{32} \mid P_{33} \end{cases}$$

$$P = P_1 \mid P_2 \mid P_3$$

即我们通过 2 个时钟的算法，实现膨胀运算的结果。

膨胀算法的 Verilog HDL 实现代码见附录所示。

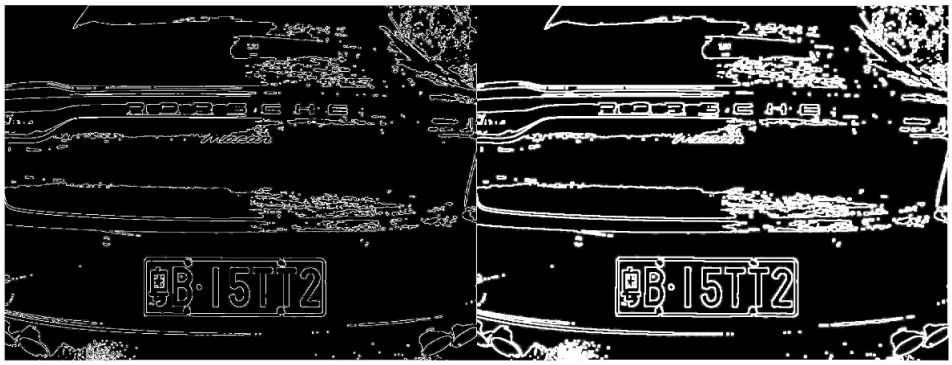


图 **12**　膨胀算法 **MATLAB** 仿真

2.3.5 卷积运算识别模块

特征值提取

在之前的车牌定位与字符分割模块中我们得到了各个字符的位置并将其划分为单个像素点集合，接下来我们会对单个像素点集合进行“行/场”方向计数并计算其区域内的像素总数。得到像素点总数后，接下来我们对单个像素点的黑白呈现形式进行判断，将白色区域占比超过 50% 的像素点判定为 1，其余像素点判定为 0，从而将像素点的抽象信号转换为零一信号输出，得到单个字符的特征值组并将其以二维数组的形式存储。

在得到单个字符的特征值组之后，想要进行对应的字符匹配，还需要得到标准的字符特征值组。为得到标准特征值组，我们采用大模型训练，使用 modelsim 对多个字符进行多次仿真，提取出其平均特征值组作为标准特征值组，并存储为二维数组格式。

卷积匹配

在之前的模块中，我们成功得到了各个字符的标准特征值组和实时特征值组，将其作为该系统的数据输入。

卷积层：采用 3*3 卷积核在特征值组上进行卷积匹配，卷积核在原有的特征值组上进行滑动计算。卷积核上的数字与特征图上对应位置的数字进行相乘并相加得到新的特征值 (卷积结果)。在进行卷积的过程中卷积核的通道 (深度) 与输入特征层的通道 (深度) 一致的，输出的特征矩阵通道 (深度) 与卷积核的个数是相同的，保证了进行卷积之后的图像特征的完整。

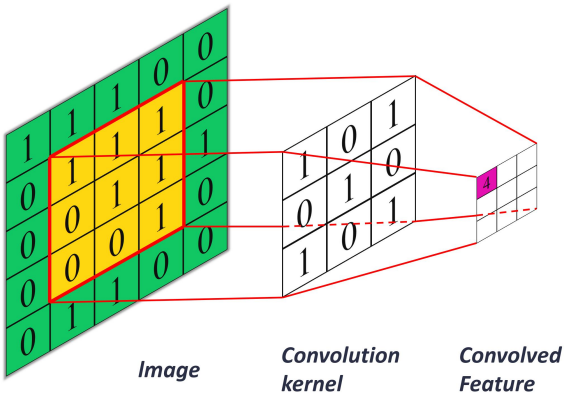


图 **13**　卷积核的滑动计算

池化层：目的是对特征值进行进一步的稀疏处理，减小图像数据的运算量。我们采用的是 2*2 池化核进行平均池化，对应位置的特征值进行平均化处理，得到更加精简的特征矩阵。

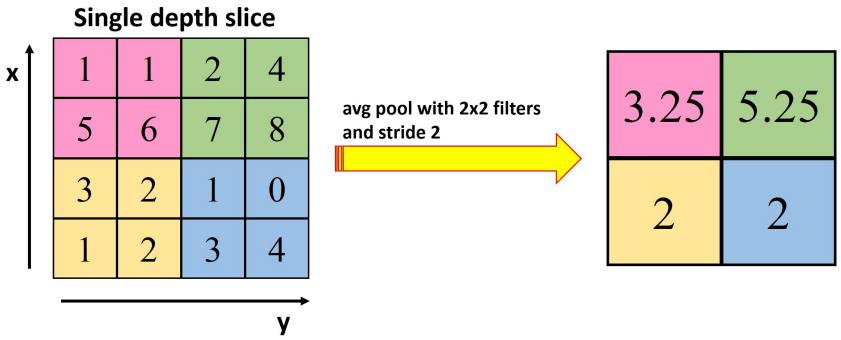


图 **14**　池化核的平均计算

通过应用以上两种算法，本系统成功将字符的特征值进行二次提取与稀疏，较大程度降低了特征值的复杂度，将得到的实时字符特征值组与标准特征值组生成精简特征矩阵。

将得到的两个精简特征矩阵进行同或模板匹配，相同位置的特征值进行比较，将偏差率在 5% 以内的定义为特征值相匹配，并统计这两个特征矩阵的总体匹配率。之后比较当前字符的特征值组与其它标准字符特征值组的总体匹配率，选取对应总体匹配率最高的标准字符为所识别的字符。

2.4 Z 与 2 模糊字符智能判定

通过查阅《中华人民共和国机动车号牌》(GA36-2018)5.9.1 序号编码规则，我们发现序号编码规则共有三种，分别是：

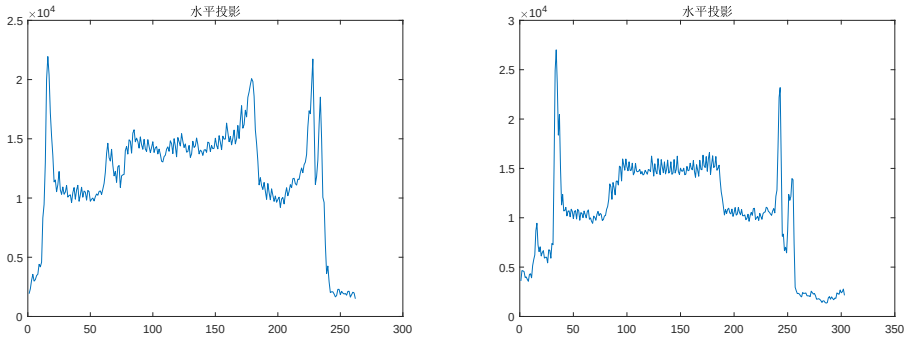
i 序号的每一位都使用阿拉伯数字;

ii 序号的每一位可单独使用英文字母，26 个英文字母中 O 和 I 不能使用;

iii 序号中允许出现 2 位英文字母，26 个英文字母中 O 和 I 不能使用。

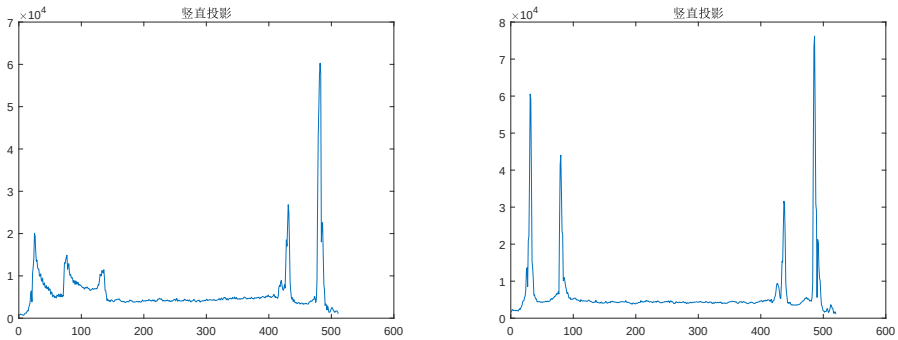
这项条例的目的是为了避免在车牌识别的过程中出现 0 与 O，1 与 I 的判定不准确这一问题。但在实际操作中我们发现，由于 2 与 Z 形状的相似，两者的判定仍旧较为模糊，导致在车牌识别的过程中会出现 2-Z 来回抖动的问题，严重影响了识别效果。针对此问题，我们提出了 2-Z 模糊字符的二次界定方法。

我们首先用 MATLAB 软件进行对字符 2 与字符 Z 的 Sobel 边缘检测仿真，这里，我们使用从多个角度拍摄得到的单个字符 2 与 Z 作为输入，Sobel 边缘检测后得到的像素梯度幅值总和作为输出。这样我们得到了 2 与 Z 的水平投影与竖直投影，如下图所示:



(a) 2 水平投影　　**(b) Z** 水平投影

图 **15**　**2** 与 **Z** 水平投影

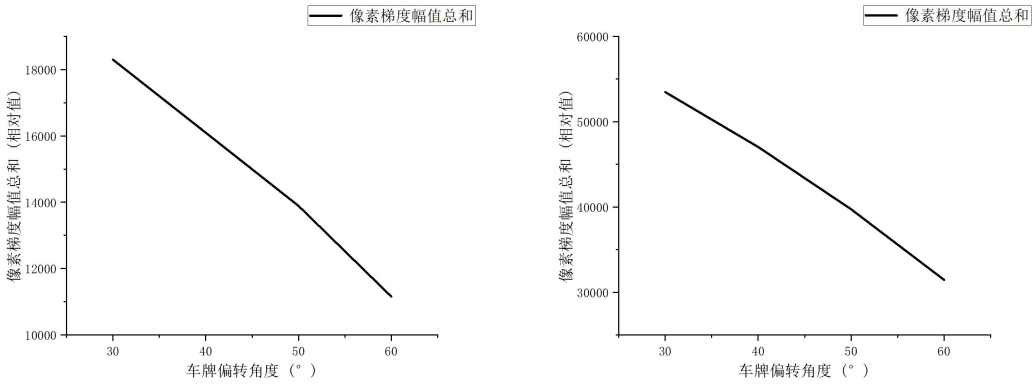


(a) 2 竖直投影　　**(b) Z** 竖直投影

图 **16**　**2** 与 **Z** 竖直投影

通过观察 MATLAB 仿真的数据结果来看，2 与 Z 的水平投影分布较为相似，很难从水平投影来进行 2 与 Z 的二次判定。

然而我们通过观察竖直投影的结果可以得到，2 与 Z 的第一个峰值有较明显差异，因此我们利用二者的这一特性来对他们进行基于卷积特征匹配的二次判定。同时为了减小图像拍摄角度对字符识别的影响，我们对字符 2 与 Z 进行 X 的三维旋转来模拟出车辆在不同角度的识别情况，得到了车牌偏转角度与像素梯度幅值总和 (第一个峰值) 的变化关系：



(a) 字符 **2** **(b)** 字符 **Z**

图 **17** 字符 **2** 与 **Z** 车牌偏转角度与像素梯度幅值总和关系

观察上面两幅图我们发现，车牌偏转角度基本与像素梯度幅值 (第一个峰值) 呈反比例关系，且字符 Z 的最小像素梯度幅值总和 (第一个峰值) 为 31462.2，字符 2 的最大像素梯度幅值总和 (第一个峰值) 为 20092.7。因此，我们使用二者均值作为二次判定的阈值：

$$threshold_value = 25777.45$$

下面是 2-Z 模糊字符判定算法的 verilog 伪代码:

```
if (match_template==2 or Z) begin
// Sobel边沿检测 获取竖直投影第一个峰值
peak_value = Sobel_and_get_peak(image);
if (peak_value > 25777.45) begin
    result = "Z";
end else begin
    result = "2";
end
end else begin
```

```
result = match_template; // 保留原始字符
end
```

以下是完整的 2-Z 模糊字符判定算法流程:

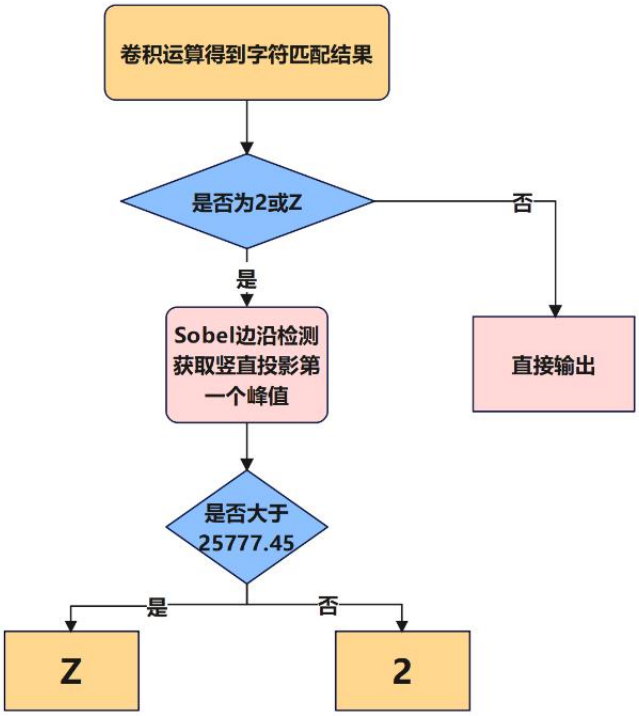


图 **18**　模糊字符判定算法流程图

将该算法部署到 FPGA 上后, 我们对含有 2 或 Z 的车牌通过 modelsim 仿真进行鲁棒性检验，发现改进识别方法之后对识别准确度有较大提升，证明该算法克服了单一的卷积特征匹配对 2 与 Z 的判断模糊这一问题。

2.5 距离省电控制模块

当车辆驶入系统监测范围，距离省电模块会打开芯片供电通道，启动 FPGA 芯片对图像进行处理和计算。在该模块中我们使用 HC-SR04 超声波距离测量模块作为测距传感器。

在该模块中我们首先通过分频使能时钟模块将 200Mhz 系统时钟分频为周期 1us 的使能时钟，使能时钟为超声波触发模块 (vlg_trig) 和回响信号模块 (vlg_echo) 提供 1us 的计数单位；超声波触发模块负责产生触发脉冲驱动硬件传感器工作，回响接收模块处理硬件传感器传回的回响信号，对回响脉冲高电平时间计数；然后输出给数学运算模块 (vlg_cal)，将时间转换成距离，将该距离与预先定义的识别距离相比较，满足识别距离之时，模块会输出一个高电平信号用于图像识别系统的启动。

距离省电模块布置在 FPGA 上的控制流程图如下所示：

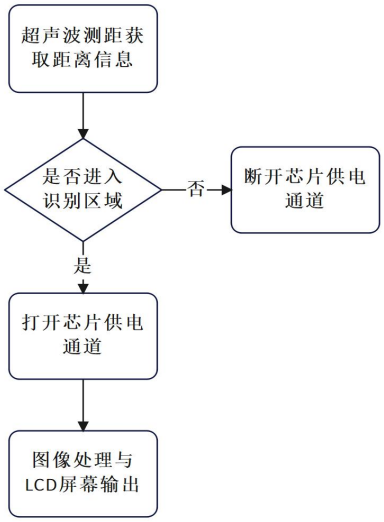


图 **19**　距离省电控制流程图

下图为该模块的原理图文件，其中输出信号 key 控制着芯片的供电通道。

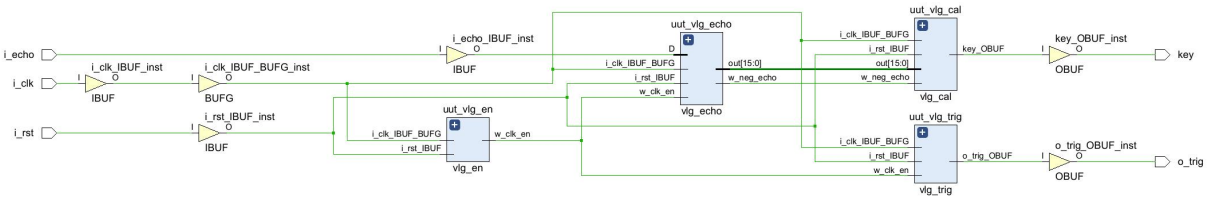


图 **20**　距离省电控制模块原理图

由于本产品为微缩展示模型，故将识别区域设定为距离摄像头 20cm 内的范围，当超声波雷达测量得出摄像头 20cm 范围内有物体靠近，便会将 key 信号置为 1，进而打开芯片的供电通道，系统转化为工作状态。当摄像头 20cm 范围内没有物体时，系统为待机状态，此时仅有超声波雷达模块进行工作，这样便良好的实现了省电模式的功能。

2.6 库识别与智能交互模块

库识别与交互控制是利用以上车牌所识别出的结果，与建立的车牌库中的内容进行比对，进而输出一个是否通过信号，并根据输出的信号进行智能交互的控制，从而达到整个系统智能化的目的。

对于车牌库的建立，我们采用车牌识别结果的编号内容，将其作为车牌库中所识别的内容，同时也可以通过这种方式进行对于车牌库中元素的添加与删除，而对于车牌库的识别，我们将车牌识别的结果作为需要比对的内容，将其与车牌库中的内容进行相似性的检验，若其与车牌库中的一个元素完全相同，则输出一个结果为“1”，若其与车牌库中所有的元素均不相符，则输出结果为“0”。

对于智能交互系统，我们将舵机模块作为本系统的智能交互控制，利用了库识别的输出结果，对舵机进行控制，从而可以实现落杆与抬杆的操作，实现了本系统在停车场这一应用场景中的实现，是实现系统实用性的重要部分。

当舵机模块收到来自库识别模块的结果时，会根据其所输入的结果进行是否抬杆的判断，当收到的结果是不通过时，舵机模块并不会抬杆，反之，舵机模块就会生成一个脉冲宽度调制 (PWM) 信号，实现对于舵机的控制，进而实现所需的抬杆操作，通过这一模块，我们实现了本系统在智能交互输出中的应用。

以下是舵机模块综合后生成的原理图文件，其中输出 pwm 是系统控制舵机的 pwm 信号:

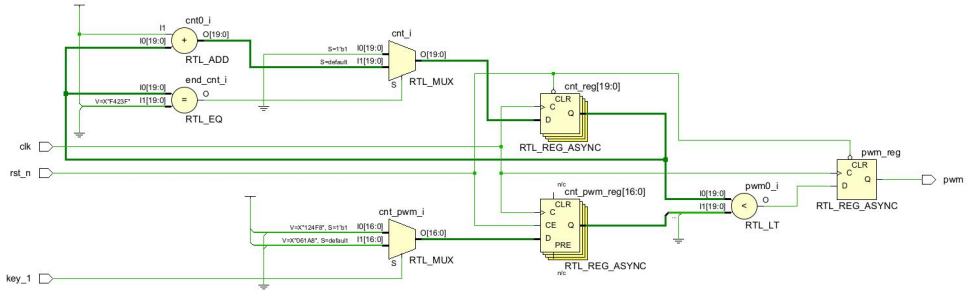


图 **21**　舵机原理图

库识别与交互控制模块的 Verilog HDL 模块见附录所示。

2.7 算法加速模块

在传统的车牌识别系统中，车牌图像处理的速度问题一直是一大缺陷。系统运行较慢的主要因素是处理后图像的存储以及对实时视频流字符的匹配。单独的同或模板匹配需要根据像素点将复杂的图像转化为特征矩阵，匹配时处理的特征值较多，对 FPGA 的资源占用以及芯片算力有较大的要求。同时，实时视频流图像的存储与调用问题也是影响速度的关键因素，如果将处理完成的图像全部存入 DDR3 之中，存储的过程以及之后进行模板匹配的调用过程都会耽误一定的时间。在基于 FPGA 的车牌识别系统中我们利用了 FPGA 的可加速与并行计算特性提升计算速度

针对图像的匹配复杂问题，我们采用卷积的方式，首先在卷积层通过卷积核对复杂图像的特征值进行提取，在保证复杂图像特征值完整的基础上进行特征值的压缩，之后在池化层运用池化核对卷积得到的特征值进行进一步的稀疏处理，通过卷积层和池化层两部分的计算，将图像的特征值进行了提取与转化，成功将特征值进行化简压缩。通过卷积减小了之后同或模板匹配过程中关于系统 DSP 资源占用过多的问题，加快了速度。

针对卷积前后实时视频流的大数据图像存储问题，我们放弃在此处使用 DDR3 存

储，采用 Ram 型资源进行缓存。由于数据量较大，单一的 Ram 无法完成存储工作，采用专门的 Block Ram 硬件资源可以很好的解决问题。在存储形式上，我们将卷积后的数据通过 IP 核进行图像数据的每周期移位寄存求和，再将卷积结果移入寄存器 Block Ram 阵列中进行下一步的准备。这样降低数据的缓存行数，缩短了系统在进行卷积同或模板匹配所需的时间。

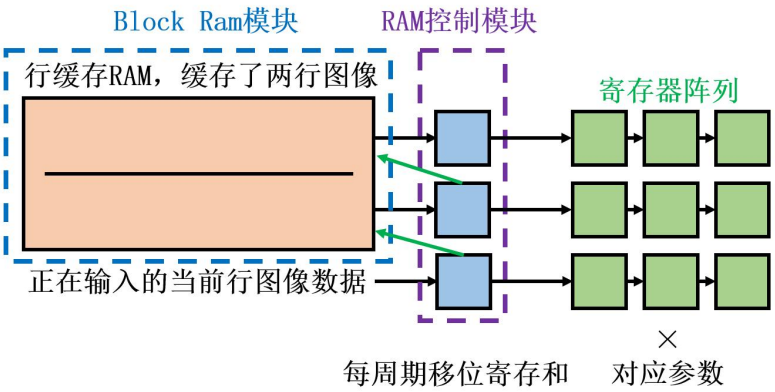


图 **22**　卷积加速原理

以下是关于使用 IP 核移位以及数据存储到 Block Ram 的 Verilog HDL 伪代码：

```
// 处理数据流的逻辑
always @(posedge clk or posedge rst) begin
    if (rst) begin
        // 重置状态
        conv_accum <= 0;
        addr <= 0;
         we <= 0;
    end else if (pixel_valid) begin
        // 滚动窗口更新，移位寄存器填充数据
        // 更新窗口内的数据
        shift_reg[0][0] <= shift_reg[0][1];
        shift_reg[0][1] <= shift_reg[0][2];
        shift_reg[1][0] <= shift_reg[1][1];
        shift_reg[1][1] <= shift_reg[1][2];
        shift_reg[2][0] <= pixel_data;  // 新输入的数据填充到窗口的最后一行
        // 卷积操作：计算卷积结果
        conv_accum <= 0;  // 每次新数据进入时，重置累加器
        for (int i = 0; i < KERNEL_SIZE; i = i + 1) begin
            for (int j = 0; j < KERNEL_SIZE; j = j + 1) begin
                conv_accum <= conv_accum + (shift_reg[i][j] * kernel[i][j]);
            end
        end
        // 设置 Block RAM 的地址和数据
        bram_addr <= addr;
        bram_data <= conv_accum;
        we <= 1;  // 使能信号开启
        // 更新地址以便写入 Block RAM
        addr <= addr + 1;
    end else begin
```

```
            we <= 0;  // 数据无效时禁用写使能
        end
    end
    assign bram_we = we;
    assign conv_result = conv_accum;
```

三、 完成情况及性能输出参数

3.1 车牌基本情况介绍

依据《中华人民共和国机动车号牌》，蓝色车牌用于中型以下载客、载货汽车和专项作业车，以及轻便摩托车，黄色车牌用于中型（含）以上载客、载货汽车和专项作业车，半挂牵引车，电车，挂车，教练车，普通摩托车和低速车，白色车牌用于警车，消防救援车辆，警用摩托车和消防救援摩托车，绿色车牌用于新能源汽车，包括小型新能源汽车和大型新能源汽车（黄绿双拼色）。

在本车牌识别系统中，我们针对的是蓝色车牌，针对其蓝色，我们采用了灰度图像二值化算法，对车牌进行二值化，将其处理为黑白格式的视频流。根据车牌大小 440*140mm，我们在车牌的定位模块中加入对尺寸的限制，在定位的过程中控制车牌的长宽比为 22：7，其中设置识别车牌尺寸的偏差不超过 7%, 以便于对于车牌的位置进行精准定位。

在车牌字符的编码规则上，车牌第一位是汉字，代表省级行政区简称，车牌第二位是英文字母，代表地级行政区代码，车牌第三位至第七位是序号，可以是数字、字母或数字与字母的组合。根据车牌不同位置字符类型的不同，为了减少匹配量，第一个字符只匹配汉字，第二个字符只匹配字母，剩余五个字符是在字母和数字之中进行匹配，将分割之后的字符进行识别。

3.2 初步测试

将上述各模块综合后，考虑到作品的展示问题，我们将超声波测距模块和摄像头模块固定在一起，并采用车牌微缩模型对实验效果进行初步测试。

在这里，我们将“粤 B15TT2”与“沪 CZV299”作为目标车牌对该系统的呈现效果进行模拟测试。在测试前，我们提前将车牌号“粤 B15TT2”录入到车牌库当中, 并分别将两个车牌先后置于距离超声波测距模块 30cm 和 10cm 处进行测试。

下图为 LCD 输出模块和智能交互模块的效果：

如上图所示，由于我们设定的待机状态识别范围为超声波测距模块 20cm 内的区域。当车辆距离超声波测距模块 30cm 处时，LCD 屏幕熄灭，系统工作指示灯熄灭，舵机不发生偏转，系统不进行图像处理，处于省电待机状态。

当车辆位于距离超声波测距模块 10cm 处时，LCD 屏幕唤醒，系统工作指示灯点亮，系统处于工作状态，开始进行图像的处理，并在 LCD 屏幕上实时显示 OV5640 摄像头采集到的画面与车牌区域划分和字符识别结果。

当车牌号为“粤 B15TT2”的车辆进入检测范围内，该系统精准划分车牌对应位置，并输出“粤 B15TT2”作为识别结果。由于该车牌已存储在车牌库当中，系统自动控制智能交互模块输出 PWM 信号，驱动舵机抬升 90°，实现自动抬杆效果。

当车牌号为“沪 CZV299”的车辆进入检测范围内，由于该车牌并未存储在车牌库当中，系统识别后检测不与库中车牌匹配，不驱动舵机抬升，舵机处于复位状态，阻止了陌生车辆进入。

3.3 性能参数

在对系统功能进行初步测试后，我们提出了一套综合测试方案对该系统各项性能参数进行全面测试，下面是我们的测试方案：

序号	测试具体内容	测试方法
1	车牌识别系统稳定性	模拟不同车牌的情况，测量系统识别精度
2	含有 2 和 Z 的车牌识别	2 和 Z 的字符模拟，测量识别精度与提升效果
3	省电模式的触发精度	不同距离模拟驶入，测量触发精度
4	FPGA 硬件加速效果	与市面现有方案比较图像处理速度
5	智能车牌库读写精度	模拟不同车牌的情况，判断舵机信号接收
6	低速运动车辆识别	模拟不同低速接近，测量识别精度与速度
7	车牌三维旋转倾斜识别	模拟不同角度识别，测量识别精度
8	低环境亮度识别	模拟低亮度环境，测量识别精度

测试集的构建:

卷积运算消耗大量数字信号处理模块 (DSP) 的资源，由于芯片资源限制，且为了保证识别准确度与处理速度，该系统单次只能识别汉字、数字、英文字母共计 13 种字符。在综合各字符在家用车牌中的使用频率以及车牌样本采样的难易程度后，我们最终选取汉字“粤”、“沪”，数字“1”、“2”、“5”、“9”，字母“B”、“C”、“J”、“T”、“U”、“V”、“Z”作为最终示例字符。同时，我们也使用 modelsim 仿真得到了所有数字和英文字母以及绝大多数汉字的特征值并注释在了代码中，方便后续芯片资源扩充和程序移植时使用。

针对上述情况，由于芯片资源的限制，导致我们无法一次性满足所有车牌的测试。因此，我们构建了两组局部测试集，每一组测试集使用不同的 13 种字符进行测试。这样，该测试便能将市面上常用的字符全部测试。

该局部测试集包含三类图像：独立车牌图像，车辆前后实拍图像，AI 模拟图像。独

立车牌图像与车辆前后实拍图像均为我们团队在校园中和网络上搜集到的高清真实相片。同时，为了扩充测试集的样本总数，我们使用 AI 模拟得到了一些符合要求的车牌图像。此时，综合两组局部测试集，我们便构建起了一套完整的测试集，该测试集大约包含 200 张车牌照片，下面是测试集的局部示意图：

图 **23**　测试集的局部示意图

3.3.1 测试 1：车牌识别系统稳定性

我们选取测试集中的部分车牌，将车牌图片传输到平板上，利用平板模拟车辆驶入的图片信号，记录识别的准确率，下面是测试的部分结果图以及准确率计算：

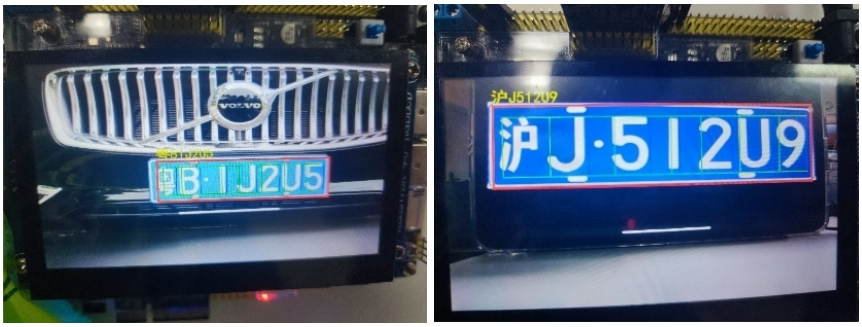


图 **24**　测试 **1** 结果图

由此，我们得到两个局部测试集的综合准确率为 91.74%，高于 90%，证明该系统识别准确率较高，能够从复杂环境中精准定位车牌的位置并准确识别。

测试集	测量总数	识别准确数	识别准确率	综合准确率
局部测试集 1	43	40	93.02%	91.47%
局部测试集 2	21	19	90.47%	

表 **1**　识别准确率计算

3.3.2 测试 2: 含有 2 和 Z 的车牌识别:

在市面上现有的车牌识别方案中，由于 2 和 Z 在形状上的相似，单一的基于特征值的卷积模板匹配并不能对 2 和 Z 进行精准有效的区分，进而导致识别准确率下降和识别车牌状态抖动的问题，严重影响了识别效果。

针对这一问题，我们在上文中提出了 2 与 Z 模糊字符智能判定算法，来提升识别准确度。这里，我们从测试集中选取含有 2 和 Z 比例较高的车牌进行测试，并使用 AI 生成的两种极端情况的车牌 (粤 Z22222 、粤 ZZZZZZ) 进行极端测试，得到了以下测试结果与提升效果：

图 **25**　极端情况识别结果图

算法	测量总数	识别准确数	识别准确率	准确提升率
原算法	12	4	33.33%	275.03%
改进算法	12	11	91.67%	

表 **2**　含 **2-Z** 车牌识别准确率计算

3.3.3 测试 3: 省电模式触发精度:

由于本系统为微缩演示模型，故将识别区域设定为距离摄像头 20cm 内的范围。我们针对这项指标进行进一步测试。

我们分别将车辆置于距离摄像头 22cm、21.5cm、21cm、20.5cm、20cm 、19.5cm、19cm 处，观察系统的唤醒情况，得到了下面的测试结果：

与摄像头距离 (cm)	系统状态
22	待机
21.5	待机
21	待机
20.5	待机与工作状态频繁切换
20	待机与工作状态频繁切换
19.5	待机与工作状态频繁切换
29	工作状态

表 **3**　省电模式触发精度测算

根据测试结果，我们发现，当车辆距离摄像头 21 19cm 之间时，由于使能信号的不稳定，会导致系统在待机状态与启动状态见频繁切换，因此，该系统的触发精度是。由于在日常应用中，车辆与摄像头之间的距离远超 1c m，因此该误差可忽略不计，并不会影响系统的触发判定。

3.3.4 测试 4:FPGA 硬件加速效果:

我们在 Github 网站上寻找其他车牌识别方案，记录其识别输出帧率。同时，我们在系统内部添加了额外计数器对输出的有效视频帧进行计数，并在每一秒读取计数器的值，通过该测算方法得到了 FPGA 硬件加速后系统输出帧率，如下表所示：

方案	LCD 屏幕输出帧率	提升百分比
Arm(无加速)	<3fps	264.40%
Arm+FPGA 加速	7.932fps	

表 **4**　加速效果参数表

上述表格证明，使用 FPGA 硬件加速方案后，实时车牌识别帧率得到了显著提升。

3.3.5 测试 5: 智能车牌库读写精度:

我们在局部测试集 1 当中选取 40 各不同的车牌号，作为该项指标的测试集。其中，我们提前在车牌库中录入了 20 个车牌号，剩下 20 个车牌号作为陌生车牌，分别将其置

于摄像头前 15cm 处，观察人机交互模块的抬杆情况，计算智能车牌库读写精度，如下表所示：

	抬杆个数	未抬杆个数	准确率
已录入车牌号	20	0	100%
陌生车牌	1	19	95%

表 **5**　智能车牌库读写精度

根据测试结果，所有录入的车牌号均实现了抬杆，证明智能车牌库能够调用识别结果并与库内车牌进行匹配分析，最终控制人机交互模块实现抬杆运动。

而陌生车牌中有一个车牌也实现了抬杆，经过多次观察与测量我们发现，车辆在驶入识别区域的过程中，会出现一段短暂的识别结果跳动的情况，进而导致误匹配的情况发生。我们后续会通过调用稳定信号作为识别结果，来减少误抬杆的概率。

3.3.6 测试 6: 低速运动车辆识别:

由于该系统的应用场景为智能停车场，车辆在识别的过程中均为静态，故上面所有的测试都是静态车辆车牌的识别。

然而，为了节约车辆在停车场交通栏杆前的等待时间，提高停车场的运营效率，我们在这里对低速运动的车辆进行识别，期望实现在车辆还没停稳的情况下实现抬杆的同时还保证识别的准确率，下面是车辆以不同速度驶入识别范围的准确率变化图：

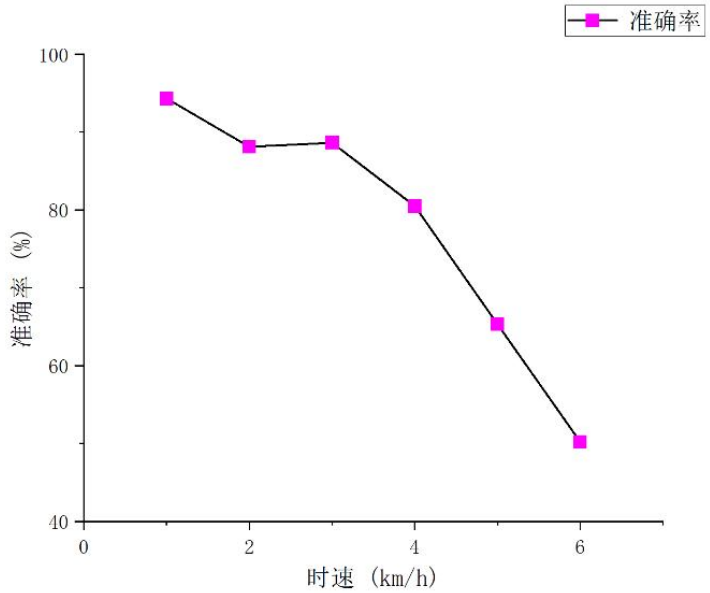
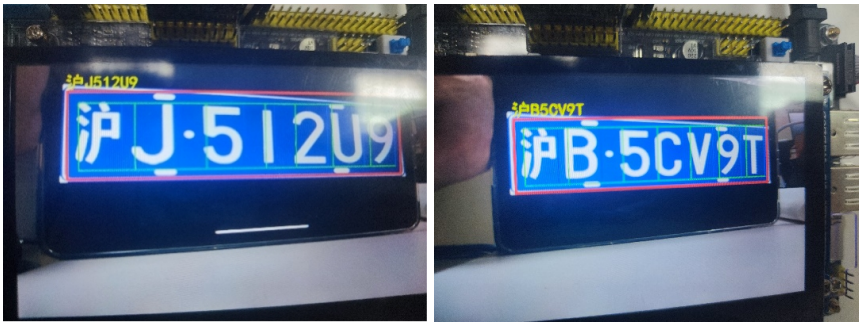


图 **26**　速度与准确率变化图

由上图可知，在时速 3km/h 以内的车辆，识别准确率仍旧能保持在 85% 以上这样一个良好的级别，当车速大于 4km/h 后，车牌识别准确率大幅度下降，无法实现正常识别功能。因此需要控制车辆速度在 3km/h 以下才能维持系统正常运行。由于系统的应用场景为停车场车牌识别，这样的要求也是容易实现的。

3.3.7 测试 7: 车牌的三维旋转倾斜识别:

在实际情况中，无法保证每次识别时摄像头都能与车牌正对应，因此，我们加入了车牌三维旋转倾斜识别，来测试车辆和摄像头倾斜时的识别效果与准确率，下面是识别效果图和偏转角度与准确率的变化图：



(a) 偏转 **15°** **(b)** 偏转 **30°**

图 **27** 识别效果图

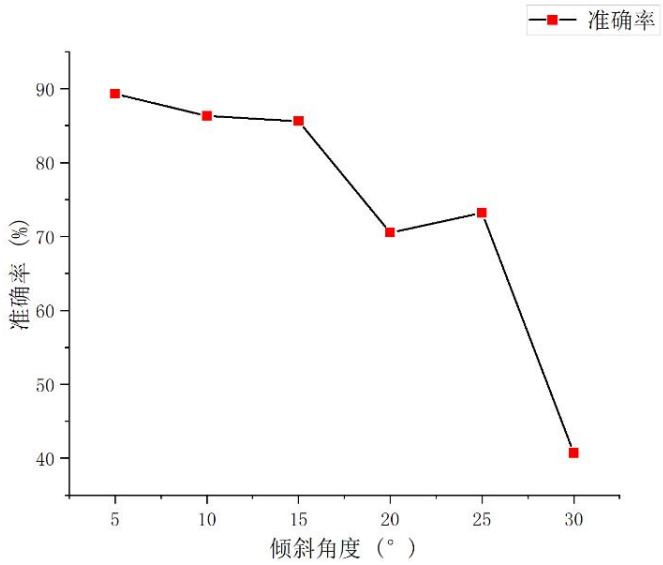


图 **28** 偏转角度与准确率变化图

由上图可知，倾斜角度与准确率呈现负相关的关系，且倾斜角度大于 15 度时，准确率有一个明显的下降；大于 30 度时，准确率已经降到 40%，此时系统已经无法正常

工作。因此，我们需要限制车辆驶入的角度，来保持车辆与摄像头的倾斜角度小于 15 度，这样才能保证系统正常运行。

3.3.8 测试 8: 低环境亮度识别:

根据调查显示，夜间是停车场出入的一个高峰，意味着系统要适应在低环境亮度下的车牌识别。我们针对这一情况已提出了一个解决方案：当车辆驶入识别范围内时，系统被唤醒，同时 OV5640 摄像头模块上的两个 LED 灯会点亮，为待识别车辆提供短暂照明。这里，我们将对这一解决方案的效果进行进一步测量，测试结果如下表所示：

图 **29** **OV5640** 上提供照明的 **LED**

测量总数	识别准确数	识别准确率
17	13	76.47%

表 **6** 低环境亮度 **(30lx)** 下识别准确率

该测试得到在 30lx 的环境下识别准确率大概为 76.47%，证明低亮度会对识别准确率造成一定的影响，但这种影响是可接受的，这反映出该系统在低环境亮度下表现良好，后续可以通过添加外加光源来减少这种影响。

四、 总结

4.1 可拓展之处

数据处理与云服务：该车牌识别系统可以与大数据和云计算技术结合，实现数据的云端化处理和存储。这样不仅可以提高数据处理能力，还可以实现数据的远程访问和分析。

结合物联网：该车牌识别系统可以与物联网进行结合，在智能停车场中，车牌识别技术可以实现车辆的快速进出管理、车位引导、自动计费等功能，大大提高了停车效率，可以与其他智能系统如智能路灯、环境监测系统等集成，实现资源的高效管理和利用。

安全性与隐私保护：在该车牌识别系统中可以加入更多的安全措施，如数据加密、用户身份验证等，以保护用户数据的安全和隐私

4.2 心得体会

参加 FPGA 竞赛是一次极具挑战性的经历，它不仅考验了我们对 FPGA 芯片的使用、对硬件描述语言 Verilog 的掌握，还锻炼了我们的时间管理能力和团队协作精神。

在竞赛中，我们深刻体会到了理论与实践相结合的重要性。从最初的设计概念到程序的基本框架再到最终的硬件实现，我们学会了如何在紧张的时间压力下，快速地进行新知识的学习并从理论转向实践，将抽象的想法转化为具体的设计方案。同时我们意识到，每个成员都有自己的专长和视角，通过有效的沟通和协作，我们可以互补不足，共同克服难题。

在面对整个系统的设计挑战时，我们学会了如何分配任务，确保每个成员都能在其最擅长的领域发挥最大的作用。并且，我们学会了如何制定详细的计划，合理分配时间，以确保项目能够按时完成。这次竞赛经历不仅提升了我们的技术能力，也增强了我们解决问题的能力和团队合作精神，为我们未来的职业道路打下了坚实的基础。

参考文献

[1] GA 36-2018. 中华人民共和国机动车号牌 [S].null,2018-04-20

[2] 张家瑞. 面向智能书法评价系统的图像分割及二值化处理 [D].null,2024

[3] 徐冰冰，岑科廷，黄俊杰，沈华伟，程学旗. 图卷积神经网络综述 [J]. 计算机学报,2020,43: 755-780

[4] 徐旭，陈曦. 基于 FPGA 的 sobel 边缘检测算法的实现 [J]. 电子测试,2022,36(23): 63-65

[5] 王国军. 基于 HLS 的 FPGA 硬件加速电路设计与实现 [D].null,2023

附录 A　代码

图像处理模块

```
module rgb2ycbcr
(
    //module clock
    input               clk             ,
// 模块驱动时钟
    input               rst_n           ,
// 复位信号

    //图像处理前的数据接口
    input               pre_frame_vsync ,
// vsync信号
    input               pre_frame_hsync ,
// hsync信号
    input               pre_frame_de    ,
// data enable信号
    input       [4:0]   img_red         ,
// 输入图像数据R
    input       [5:0]   img_green       ,
// 输入图像数据G
    input       [4:0]   img_blue        ,
// 输入图像数据B

    //图像处理后的数据接口
    output              post_frame_vsync,
// vsync信号
    output              post_frame_hsync,
// hsync信号
    output              post_frame_de   ,
// data enable信号
    output      [7:0]   img_y           ,
// 输出图像Y数据
    output      [7:0]   img_cb          ,
// 输出图像Cb数据
    output      [7:0]   img_cr
// 输出图像Cr数据
);

//reg define
reg  [15:0]   rgb_r_m0, rgb_r_m1, rgb_r_m2;
reg  [15:0]   rgb_g_m0, rgb_g_m1, rgb_g_m2;
reg  [15:0]   rgb_b_m0, rgb_b_m1, rgb_b_m2;
reg  [15:0]   img_y0 ;
reg  [15:0]   img_cb0;
reg  [15:0]   img_cr0;
reg  [ 7:0]   img_y1 ;
reg  [ 7:0]   img_cb1;
reg  [ 7:0]   img_cr1;
reg  [ 2:0]   pre_frame_vsync_d;
reg  [ 2:0]   pre_frame_hsync_d;
reg  [ 2:0]   pre_frame_de_d    ;

//wire define
wire [ 7:0]   rgb888_r;
wire [ 7:0]   rgb888_g;
wire [ 7:0]   rgb888_b;



//RGB565 to RGB 888
assign rgb888_r         = {img_red  , img_red
 [4:2]  };
assign rgb888_g         = {img_green,
 img_green[5:4]};
assign rgb888_b         = {img_blue ,
 img_blue[4:2] };
//同步输出数据接口信号
assign post_frame_vsync = pre_frame_vsync_d
 [2]      ;
assign post_frame_hsync = pre_frame_hsync_d
 [2]      ;
assign post_frame_de    = pre_frame_de_d[2]
        ;
assign img_y            = post_frame_hsync ?
 img_y1 : 8'd0;
assign img_cb           = post_frame_hsync ?
 img_cb1: 8'd0;
assign img_cr           = post_frame_hsync ?
 img_cr1: 8'd0;

//
 ----------------------------------------
//RGB 888 to YCbCr
//step1 pipeline mult
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        rgb_r_m0 <= 16'd0;
        rgb_r_m1 <= 16'd0;
        rgb_r_m2 <= 16'd0;
        rgb_g_m0 <= 16'd0;
        rgb_g_m1 <= 16'd0;
        rgb_g_m2 <= 16'd0;
        rgb_b_m0 <= 16'd0;
        rgb_b_m1 <= 16'd0;
        rgb_b_m2 <= 16'd0;
    end
    else begin
        rgb_r_m0 <= rgb888_r * 8'd77 ;
        rgb_r_m1 <= rgb888_r * 8'd43 ;
        rgb_r_m2 <= rgb888_r << 3'd7 ;
        rgb_g_m0 <= rgb888_g * 8'd150;
        rgb_g_m1 <= rgb888_g * 8'd85 ;
        rgb_g_m2 <= rgb888_g * 8'd107;
```

```
        rgb_b_m0 <= rgb888_b * 8'd29 ;
        rgb_b_m1 <= rgb888_b << 3'd7 ;
        rgb_b_m2 <= rgb888_b * 8'd21 ;
    end
end

//step2 pipeline add
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        img_y0  <= 16'd0;
        img_cb0 <= 16'd0;
        img_cr0 <= 16'd0;
    end
    else begin
        img_y0  <= rgb_r_m0 + rgb_g_m0 +
rgb_b_m0;
        img_cb0 <= rgb_b_m1 - rgb_r_m1 -
rgb_g_m1 + 16'd32768;
        img_cr0 <= rgb_r_m2 - rgb_g_m2 -
rgb_b_m2 + 16'd32768;
    end

end

//step3 pipeline div
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        img_y1  <= 8'd0;
        img_cb1 <= 8'd0;
        img_cr1 <= 8'd0;
    end
    else begin
        img_y1  <= img_y0 [15:8];
        img_cb1 <= img_cb0[15:8];
        img_cr1 <= img_cr0[15:8];
    end
end

//延时3拍以同步数据信号
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        pre_frame_vsync_d <= 3'd0;
        pre_frame_hsync_d <= 3'd0;
        pre_frame_de_d    <= 3'd0;
    end
    else begin
        pre_frame_vsync_d <= {
pre_frame_vsync_d[1:0], pre_frame_vsync};
        pre_frame_hsync_d <= {
pre_frame_hsync_d[1:0], pre_frame_hsync};
        pre_frame_de_d    <= {pre_frame_de_d
[1:0]   , pre_frame_de   };
    end
end

endmodule
```

```
对输入的图像数据进行二值化处理
module binarization(
    input               clk            ,   //
 时钟信号
    input               rst_n          ,   //
 复位信号（低有效）

    input               per_frame_vsync,
    input               per_frame_href ,
    input               per_frame_clken,
    input       [7:0]   per_img_Y  ,

    output  reg         post_frame_vsync,
    output  reg         post_frame_href ,
    output  reg         post_frame_clken,
    output  reg         post_img_Bit  ,

    input       [7:0]   Binary_Threshold
);

//二值化
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
        post_img_Bit <= 1'b0;
    else begin
        if(per_img_Y > Binary_Threshold)  //
 阈值
            post_img_Bit <= 1'b1;
        else
            post_img_Bit <= 1'b0;
    end
end


always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        post_frame_vsync <= 1'd0;
        post_frame_href  <= 1'd0;
        post_frame_clken <= 1'd0;
    end
    else begin
        post_frame_vsync <= per_frame_vsync;
        post_frame_href  <= per_frame_href ;
        post_frame_clken <= per_frame_clken;
    end
end

endmodule
腐蚀模块
`timescale 1ns/1ns
module VIP_Bit_Erosion_Detector
#(
    parameter   [9:0]   IMG_HDISP = 10'd640,
     //640*480
    parameter   [9:0]   IMG_VDISP = 10'd480
)
```

```
(
    //global clock
    input                clk,
//cmos video pixel clock
    input                rst_n,
//global reset

    //Image data prepred to be processd
    input                per_frame_vsync,
//Prepared Image data vsync valid signal
    input                per_frame_href,
//Prepared Image data href vaild  signal
    input                per_frame_clken,
//Prepared Image data output/capture enable
clock
    input                per_img_Bit,
//Prepared Image Bit flag outout(1: Value,
0:inValid)

    //Image data has been processd
    output               post_frame_vsync,
//Processed Image data vsync valid signal
    output               post_frame_href,
//Processed Image data href vaild  signal
    output               post_frame_clken,
//Processed Image data output/capture enable
 clock
    output               post_img_Bit
//Processed Image Bit flag outout(1: Value,
0:inVaild)
);

//
-----------------------------------------------

//Generate 1Bit 3X3 Matrix for Video Image
Processor.
//Image data has been processd
wire            matrix_frame_vsync; //
Prepared Image data vsync valid signal
wire            matrix_frame_href;  //
Prepared Image data href vaild  signal
wire            matrix_frame_clken; //
Prepared Image data output/capture enable
clock
wire            matrix_p11, matrix_p12,
matrix_p13; //3X3 Matrix output
wire            matrix_p21, matrix_p22,
matrix_p23;
wire            matrix_p31, matrix_p32,
matrix_p33;
matrix_generate_3x3_1bit
u_matrix_generate_3x3_1bit(
    //global clock
    .clk                    (clk),
        //cmos video pixel clock
    .rst_n                  (rst_n),
        //global reset

    //Image data prepred to be processd
    .per_frame_vsync        (per_frame_vsync)
,       //Prepared Image data vsync valid
signal
    .per_frame_href         (per_frame_href),
        //Prepared Image data href vaild
signal
    .per_frame_clken        (per_frame_clken)
,       //Prepared Image data output/capture
enable clock
    .per_img_y              (per_img_Bit),
        //Prepared Image brightness input

    //Image data has been processd
    .matrix_frame_vsync     (
matrix_frame_vsync),   //Processed Image
data vsync valid signal
    .matrix_frame_href      (
matrix_frame_href),    //Processed Image
data href vaild  signal
    .matrix_frame_clken     (
matrix_frame_clken),   //Processed Image
data output/capture enable clock
    .matrix_p11(matrix_p11),    .matrix_p12(
matrix_p12),    .matrix_p13(matrix_p13),
//3X3 Matrix output
    .matrix_p21(matrix_p21),    .matrix_p22(
matrix_p22),    .matrix_p23(matrix_p23),
    .matrix_p31(matrix_p31),    .matrix_p32(
matrix_p32),    .matrix_p33(matrix_p33)
);

//Step1
reg post_img_Bit1,  post_img_Bit2,
post_img_Bit3;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        post_img_Bit1 <= 1'b0;
        post_img_Bit2 <= 1'b0;
        post_img_Bit3 <= 1'b0;
        end
    else
        begin
        post_img_Bit1 <= matrix_p11 &
matrix_p12 & matrix_p13;
        post_img_Bit2 <= matrix_p21 &
matrix_p22 & matrix_p23;
        post_img_Bit3 <= matrix_p21 &
matrix_p32 & matrix_p33;
        end
end
```

```
//Step 2
reg post_img_Bit4;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        post_img_Bit4 <= 1'b0;
    else
        post_img_Bit4 <= post_img_Bit1 &
 post_img_Bit2 & post_img_Bit3;
end

//----------------------------------------
//lag 2 clocks signal sync
reg [1:0]   per_frame_vsync_r;
reg [1:0]   per_frame_href_r;
reg [1:0]   per_frame_clken_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        per_frame_vsync_r <= 0;
        per_frame_href_r <= 0;
        per_frame_clken_r <= 0;
        end
    else
        begin
        per_frame_vsync_r   <=  {
 per_frame_vsync_r[0],  matrix_frame_vsync};
        per_frame_href_r    <=  {
 per_frame_href_r[0],   matrix_frame_href};
        per_frame_clken_r   <=  {
 per_frame_clken_r[0],  matrix_frame_clken};
        end
end
assign  post_frame_vsync    =
 per_frame_vsync_r[1];
assign  post_frame_href     =
 per_frame_href_r[1];
assign  post_frame_clken    =
 per_frame_clken_r[1];
assign  post_img_Bit        =
 post_frame_href ? post_img_Bit4 : 1'b0;

endmodule

module  matrix_generate_3x3_1bit(
    input           clk,
    input           rst_n,

    input           per_frame_vsync,
    input           per_frame_href,
    input           per_frame_clken,
    input           per_img_y,

    output          matrix_frame_vsync,
    output          matrix_frame_href,
    output          matrix_frame_clken,
    output  reg     matrix_p11, matrix_p12,
 matrix_p13,   //3X3 Matrix output
    output  reg     matrix_p21, matrix_p22,
 matrix_p23,
    output  reg     matrix_p31, matrix_p32,
 matrix_p33
);

//wire define
wire        row1_data;
wire        row2_data;
wire        read_frame_href;
wire        read_frame_clken;

//reg define
reg         row3_data;
reg  [1:0] per_frame_vsync_r;
reg  [1:0] per_frame_href_r;
reg  [1:0] per_frame_clken_r;


assign read_frame_href    = per_frame_href_r
 [0] ;
assign read_frame_clken   = per_frame_clken_r
 [0];
assign matrix_frame_vsync = per_frame_vsync_r
 [1];
assign matrix_frame_href  = per_frame_href_r
 [1] ;
assign matrix_frame_clken = per_frame_clken_r
 [1];

//当前数据放在第3行
always@(posedge clk or negedge rst_n) begin
    if(!rst_n)
        row3_data <= 0;
    else begin
        if(per_frame_clken)
            row3_data <= per_img_y ;
        else
            row3_data <= row3_data ;
    end
end

//用于存储列数据的RAM
line_shift_ram_8bit  u_line_shift_ram_8bit
(
    .clock          (clk),
    .clken          (per_frame_clken),
    .per_frame_href (per_frame_href),

    .shiftin        ({8{per_img_y}}),
    .taps0x         (row2_data),
    .taps1x         (row1_data)
```

```
);

//将同步信号延迟两拍，用于同步化处理
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        per_frame_vsync_r <= 0;
        per_frame_href_r  <= 0;
        per_frame_clken_r <= 0;
    end
    else begin
        per_frame_vsync_r <= {
per_frame_vsync_r[0], per_frame_vsync };
        per_frame_href_r  <= {
per_frame_href_r[0],  per_frame_href  };
        per_frame_clken_r <= {
per_frame_clken_r[0], per_frame_clken };
    end
end

//在同步处理后的控制信号下，输出图像矩阵
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 3'd0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 3'd0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 3'd0;
    end
    else if(read_frame_href) begin
        if(read_frame_clken) begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p12, matrix_p13,
row1_data};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p22, matrix_p23,
row2_data};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p32, matrix_p33,
row3_data};
        end
        else begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p11, matrix_p12,
matrix_p13};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p21, matrix_p22,
matrix_p23};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p31, matrix_p32,
matrix_p33};
        end
    end
    else begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 3'd0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 3'd0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 3'd0;
    end
end

endmodule

module line_shift_ram_8bit(
    input          clock,
    input          clken,
    input          per_frame_href,

    input   [7:0]  shiftin,
    output  [7:0]  taps0x,
    output  [7:0]  taps1x
);

//reg define
reg  [2:0]  clken_dly;
reg  [9:0]  ram_rd_addr;
reg  [9:0]  ram_rd_addr_d0;
reg  [9:0]  ram_rd_addr_d1;
reg  [7:0]  shiftin_d0;
reg  [7:0]  shiftin_d1;
reg  [7:0]  shiftin_d2;
reg  [7:0]  taps0x_d0;


//在数据来到时，ram地址累加
always@(posedge clock)begin
    if(per_frame_href)
        if(clken)
            ram_rd_addr <= ram_rd_addr + 1 ;
        else
            ram_rd_addr <= ram_rd_addr ;
    else
        ram_rd_addr <= 0 ;
end

//时钟使能信号延迟三拍
always@(posedge clock) begin
    clken_dly <= { clken_dly[1:0] , clken };
end


//将ram地址延迟二拍
always@(posedge clock ) begin
    ram_rd_addr_d0 <= ram_rd_addr;
    ram_rd_addr_d1 <= ram_rd_addr_d0;
end

//输入数据延迟三拍
always@(posedge clock)begin
    shiftin_d0 <= shiftin;
```

```
    shiftin_d1 <= shiftin_d0;
    shiftin_d2 <= shiftin_d1;
end

//用于存储前一行图像的RAM
blk_mem_gen_0  u_ram_1024x8_0(
    .clka   (clock),
    .wea    (clken_dly[2]),     //在延迟的第三
个时钟周期，当前行的数据写入RAM0
    .addra (ram_rd_addr_d1),
    .dina  (shiftin_d2),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps0x)             //延迟一个时钟
周期，输出RAM0中前一行图像的数据
);

//寄存一次前一行图像的数据
always@(posedge clock ) begin
    taps0x_d0 <= taps0x;
end

blk_mem_gen_0  u_ram_1024x8_1(
    .clka   (clock),
    .wea    (clken_dly[1]),     //在延迟的第二
个时钟周期，将前一行图像的数据写入RAM1
    .addra (ram_rd_addr_d0),
    .dina  (taps0x_d0),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps1x)             //延迟一个时钟
周期，输出RAM1中前前一行图像的数据
);

endmodule
利用Sobel算子检测图像中的边缘
module Sobel_Edge_Detector #(
    parameter  SOBEL_THRESHOLD = 250 //Sobel
阈值
)(
    input       clk,            //cmos 像素
时钟
    input       rst_n,
    //处理前数据
    input       per_frame_vsync,
    input       per_frame_href,
    input       per_frame_clken,
    input [7:0] per_img_y,
    //处理后的数据
    output      post_frame_vsync,
    output      post_frame_href,
    output      post_frame_clken,
    output      post_img_bit
);
//reg define
reg [9:0]  gx_temp2; //第三列值
reg [9:0]  gx_temp1; //第一列值
reg [9:0]  gx_data;  //x方向的偏导数
reg [9:0]  gy_temp1; //第一行值
reg [9:0]  gy_temp2; //第三行值
reg [9:0]  gy_data;  //y方向的偏导数
reg [20:0] gxy_square;
reg [15:0] per_frame_vsync_r;
reg [15:0] per_frame_href_r;
reg [15:0] per_frame_clken_r;

//wire define
wire        matrix_frame_vsync;
wire        matrix_frame_href;
wire        matrix_frame_clken;
wire [10:0] dim;
//输出3X3 矩阵
wire [7:0]  matrix_p11;
wire [7:0]  matrix_p12;
wire [7:0]  matrix_p13;
wire [7:0]  matrix_p21;
wire [7:0]  matrix_p22;
wire [7:0]  matrix_p23;
wire [7:0]  matrix_p31;
wire [7:0]  matrix_p32;
wire [7:0]  matrix_p33;



assign post_frame_vsync = per_frame_vsync_r
[10];
assign post_frame_href  = per_frame_href_r
[10] ;
assign post_frame_clken = per_frame_clken_r
[10];
assign post_img_bit     = post_frame_href ?
post_img_bit_r : 1'b0;

//3x3矩阵
matrix_generate_3x3_8bit
u_matrix_generate_3x3_8bit(
    .clk                (clk),
    .rst_n              (rst_n),
    //预处理数据
    .per_frame_vsync    (per_frame_vsync),
    .per_frame_href     (per_frame_href),
    .per_frame_clken    (per_frame_clken),
    .per_img_y          (per_img_y),

    //处理后的数据
    .matrix_frame_vsync  (matrix_frame_vsync)
,
    .matrix_frame_href   (matrix_frame_href),
    .matrix_frame_clken  (matrix_frame_clken)
,
    .matrix_p11          (matrix_p11),
    .matrix_p12          (matrix_p12),
```

```
    .matrix_p13          (matrix_p13), //输出
 3X3 矩阵
    .matrix_p21          (matrix_p21),
    .matrix_p22          (matrix_p22),
    .matrix_p23          (matrix_p23),
    .matrix_p31          (matrix_p31),
    .matrix_p32          (matrix_p32),
    .matrix_p33          (matrix_p33)
);

//Step 1 计算x方向的偏导数
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        gy_temp1 <= 10'd0;
        gy_temp2 <= 10'd0;
        gy_data <=  10'd0;
    end
    else begin
        gy_temp1 <= matrix_p13 + (matrix_p23
<< 1) + matrix_p33;
        gy_temp2 <= matrix_p11 + (matrix_p21
<< 1) + matrix_p31;
        gy_data <= (gy_temp1 >= gy_temp2) ?
gy_temp1 - gy_temp2 :
                  (gy_temp2 - gy_temp1);
    end
end

//Step 2 计算y方向的偏导数
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        gx_temp1 <= 10'd0;
        gx_temp2 <= 10'd0;
        gx_data <=  10'd0;
    end
    else begin
        gx_temp1 <= matrix_p11 + (matrix_p12
<< 1) + matrix_p13;
        gx_temp2 <= matrix_p31 + (matrix_p32
<< 1) + matrix_p33;
        gx_data <= (gx_temp1 >= gx_temp2) ?
gx_temp1 - gx_temp2 :
                  (gx_temp2 - gx_temp1);
    end
end

//Step 3 计算平方和
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
        gxy_square <= 21'd0;
    else
        gxy_square <= gx_data * gx_data +
gy_data * gy_data;
end

//Step 4 开平方（梯度向量的大小）
cordic u_cordic(
    .aclk                    (clk),
    .s_axis_cartesian_tvalid (1'b1),
    .s_axis_cartesian_tdata  (gxy_square),
    .m_axis_dout_tvalid      (),
   . m_axis_dout_tdata       (dim)
  );

//Step 5 将开平方后的数据与预设阈值比较
reg post_img_bit_r;
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)
        post_img_bit_r <= 1'b0; //初始值
    else if(dim >= SOBEL_THRESHOLD)
        post_img_bit_r <= 1'b1; //检测到边缘1
    else
    post_img_bit_r <= 1'b0; //不是边缘 0
end

//延迟5个周期同步
always@(posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        per_frame_vsync_r <= 0;
        per_frame_href_r  <= 0;
        per_frame_clken_r <= 0;
    end
    else begin
        per_frame_vsync_r  <=  {
per_frame_vsync_r[14:0],matrix_frame_vsync};
        per_frame_href_r   <=  {
per_frame_href_r[14:0],matrix_frame_href};
        per_frame_clken_r  <=  {
per_frame_clken_r[14:0],matrix_frame_clken};
    end
end

endmodule

module  matrix_generate_3x3_8bit
(
    input              clk,
    input              rst_n,

    input              per_frame_vsync,
    input              per_frame_href,
    input              per_frame_clken,
    input       [7:0]  per_img_y,

    output             matrix_frame_vsync,
    output             matrix_frame_href,
    output             matrix_frame_clken,
    output reg [7:0]   matrix_p11,
    output reg [7:0]   matrix_p12,
    output reg [7:0]   matrix_p13,
    output reg [7:0]   matrix_p21,
    output reg [7:0]   matrix_p22,
```

```
    output reg [7:0]  matrix_p23,
    output reg [7:0]  matrix_p31,
    output reg [7:0]  matrix_p32,
    output reg [7:0]  matrix_p33
);

//wire define
wire [7:0] row1_data;
wire [7:0] row2_data;
wire       read_frame_href;
wire       read_frame_clken;

//reg define
reg  [7:0] row3_data;
reg  [1:0] per_frame_vsync_r;
reg  [1:0] per_frame_href_r;
reg  [1:0] per_frame_clken_r;


assign read_frame_href    = per_frame_href_r
[0] ;
assign read_frame_clken   = per_frame_clken_r
[0];
assign matrix_frame_vsync = per_frame_vsync_r
[1];
assign matrix_frame_href  = per_frame_href_r
[1] ;
assign matrix_frame_clken = per_frame_clken_r
[1];

//当前数据放在第3行
always@(posedge clk or negedge rst_n) begin
    if(!rst_n)
        row3_data <= 0;
    else begin
        if(per_frame_clken)
            row3_data <= per_img_y ;
        else
            row3_data <= row3_data ;
    end
end

//用于存储列数据的RAM
line_shift_ram_8bit  u_line_shift_ram_8bit
(
    .clock          (clk),
    .clken          (per_frame_clken),
    .per_frame_href (per_frame_href),

    .shiftin        (per_img_y),
    .taps0x         (row2_data),
    .taps1x         (row1_data)
);

//将同步信号延迟两拍，用于同步化处理
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        per_frame_vsync_r <= 0;
        per_frame_href_r  <= 0;
        per_frame_clken_r <= 0;
    end
    else begin
        per_frame_vsync_r <= {
per_frame_vsync_r[0], per_frame_vsync };
        per_frame_href_r  <= {
per_frame_href_r[0],  per_frame_href  };
        per_frame_clken_r <= {
per_frame_clken_r[0], per_frame_clken };
    end
end

//在同步处理后的控制信号下，输出图像矩阵
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 24'h0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 24'h0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 24'h0;
    end
    else if(read_frame_href) begin
        if(read_frame_clken) begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p12, matrix_p13,
row1_data};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p22, matrix_p23,
row2_data};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p32, matrix_p33,
row3_data};
        end
        else begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p11, matrix_p12,
matrix_p13};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p21, matrix_p22,
matrix_p23};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p31, matrix_p32,
matrix_p33};
        end
    end
    else begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 24'h0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 24'h0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 24'h0;
```

```
    end
end

endmodule

module line_shift_ram_8bit(
    input          clock,
    input          clken,
    input          per_frame_href,

    input   [7:0]  shiftin,
    output  [7:0]  taps0x,
    output  [7:0]  taps1x
);

//reg define
reg  [2:0]  clken_dly;
reg  [9:0]  ram_rd_addr;
reg  [9:0]  ram_rd_addr_d0;
reg  [9:0]  ram_rd_addr_d1;
reg  [7:0]  shiftin_d0;
reg  [7:0]  shiftin_d1;
reg  [7:0]  shiftin_d2;
reg  [7:0]  taps0x_d0;


//在数据来到时，ram地址累加
always@(posedge clock)begin
    if(per_frame_href)
        if(clken)
            ram_rd_addr <= ram_rd_addr + 1 ;
        else
            ram_rd_addr <= ram_rd_addr ;
    else
        ram_rd_addr <= 0 ;
end

//时钟使能信号延迟三拍
always@(posedge clock) begin
    clken_dly <= { clken_dly[1:0] , clken };
end


//将ram地址延迟二拍
always@(posedge clock ) begin
    ram_rd_addr_d0 <= ram_rd_addr;
    ram_rd_addr_d1 <= ram_rd_addr_d0;
end

//输入数据延迟三拍
always@(posedge clock)begin
    shiftin_d0 <= shiftin;
    shiftin_d1 <= shiftin_d0;
    shiftin_d2 <= shiftin_d1;
end

//用于存储前一行图像的RAM
blk_mem_gen_0  u_ram_1024x8_0(
    .clka   (clock),
    .wea    (clken_dly[2]),     //在延迟的第三
个时钟周期，当前行的数据写入RAM0
    .addra (ram_rd_addr_d1),
    .dina  (shiftin_d2),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps0x)            //延迟一个时钟
周期，输出RAM0中前一行图像的数据
);

//寄存一次前一行图像的数据
always@(posedge clock ) begin
    taps0x_d0 <= taps0x;
end

blk_mem_gen_0  u_ram_1024x8_1(
    .clka   (clock),
    .wea    (clken_dly[1]),     //在延迟的第二
个时钟周期，将前一行图像的数据写入RAM1
    .addra (ram_rd_addr_d0),
    .dina  (taps0x_d0),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps1x)            //延迟一个时钟
周期，输出RAM1中前前一行图像的数据
);

endmodule
膨胀模块
`timescale 1ns/1ns
module VIP_Bit_Dilation_Detector
#(
    parameter   [9:0]   IMG_HDISP = 10'd640,
    //640*480
    parameter   [9:0]   IMG_VDISP = 10'd480
)
(
    //global clock
    input               clk,
//cmos video pixel clock
    input               rst_n,
//global reset

    //Image data prepred to be processd
    input               per_frame_vsync,
//Prepared Image data vsync valid signal
    input               per_frame_href,
//Prepared Image data href vaild  signal
    input               per_frame_clken,
//Prepared Image data output/capture enable
clock
    input               per_img_Bit,
//Prepared Image Bit flag outout(1: Value,
```

```
0:inVaild)

    //Image data has been processd
    output              post_frame_vsync,
//Processed Image data vsync valid signal
    output              post_frame_href,
//Processed Image data href vaild  signal
    output              post_frame_clken,
//Processed Image data output/capture enable
 clock
    output              post_img_Bit
//Processed Image Bit flag outout(1: Value,
0:inVaild)
);

//
-----------------------------------------------------

//Generate 1Bit 3X3 Matrix for Video Image
Processor.
//Image data has been processd
wire            matrix_frame_vsync; //
Prepared Image data vsync valid signal
wire            matrix_frame_href;  //
Prepared Image data href vaild  signal
wire            matrix_frame_clken; //
Prepared Image data output/capture enable
clock
wire            matrix_p11, matrix_p12,
matrix_p13; //3X3 Matrix output
wire            matrix_p21, matrix_p22,
matrix_p23;
wire            matrix_p31, matrix_p32,
matrix_p33;

matrix_generate_3x3_1bit
u_matrix_generate_3x3_1bit(
    //global clock
    .clk                    (clk),
        //cmos video pixel clock
    .rst_n                  (rst_n),
        //global reset

    //Image data prepred to be processd
    .per_frame_vsync        (per_frame_vsync)
,       //Prepared Image data vsync valid
signal
    .per_frame_href         (per_frame_href),
       //Prepared Image data href vaild
signal
    .per_frame_clken        (per_frame_clken)
,       //Prepared Image data output/capture
enable clock
    .per_img_y              (per_img_Bit),
        //Prepared Image brightness input

    //Image data has been processd
    .matrix_frame_vsync     (
matrix_frame_vsync),   //Processed Image
data vsync valid signal
    .matrix_frame_href      (
matrix_frame_href),    //Processed Image
data href vaild  signal
    .matrix_frame_clken     (
matrix_frame_clken),   //Processed Image
data output/capture enable clock
    .matrix_p11(matrix_p11),    .matrix_p12(
matrix_p12),    .matrix_p13(matrix_p13),
//3X3 Matrix output
    .matrix_p21(matrix_p21),    .matrix_p22(
matrix_p22),    .matrix_p23(matrix_p23),
    .matrix_p31(matrix_p31),    .matrix_p32(
matrix_p32),    .matrix_p33(matrix_p33)
);

//Step1
reg post_img_Bit1,  post_img_Bit2,
post_img_Bit3;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        post_img_Bit1 <= 1'b0;
        post_img_Bit2 <= 1'b0;
        post_img_Bit3 <= 1'b0;
        end
    else
        begin
        post_img_Bit1 <= matrix_p11 |
matrix_p12 | matrix_p13;
        post_img_Bit2 <= matrix_p21 |
matrix_p22 | matrix_p23;
        post_img_Bit3 <= matrix_p21 |
matrix_p32 | matrix_p33;
        end
end

//Step 2
reg post_img_Bit4;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        post_img_Bit4 <= 1'b0;
    else
        post_img_Bit4 <= post_img_Bit1 |
post_img_Bit2 | post_img_Bit3;
end

//-----------------------------------------
//lag 2 clocks signal sync
reg [1:0]   per_frame_vsync_r;
reg [1:0]   per_frame_href_r;
```

```
reg [1:0]   per_frame_clken_r;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        per_frame_vsync_r <= 0;
        per_frame_href_r <= 0;
        per_frame_clken_r <= 0;
        end
    else
        begin
        per_frame_vsync_r   <=  {
per_frame_vsync_r[0],  matrix_frame_vsync};
        per_frame_href_r    <=  {
per_frame_href_r[0],   matrix_frame_href};
        per_frame_clken_r   <=  {
per_frame_clken_r[0],  matrix_frame_clken};
        end
end
assign  post_frame_vsync    =
per_frame_vsync_r[1];
assign  post_frame_href     =
per_frame_href_r[1];
assign  post_frame_clken    =
per_frame_clken_r[1];
assign  post_img_Bit        =
post_frame_href ? post_img_Bit4 : 1'b0;

endmodule

module  matrix_generate_3x3_1bit(
    input               clk,
    input               rst_n,

    input               per_frame_vsync,
    input               per_frame_href,
    input               per_frame_clken,
    input               per_img_y,

    output              matrix_frame_vsync,
    output              matrix_frame_href,
    output              matrix_frame_clken,
    output  reg         matrix_p11, matrix_p12,
 matrix_p13,   //3X3 Matrix output
    output  reg         matrix_p21, matrix_p22,
 matrix_p23,
    output  reg         matrix_p31, matrix_p32,
 matrix_p33
);

//wire define
wire        row1_data;
wire        row2_data;
wire        read_frame_href;
wire        read_frame_clken;
```

```
//reg define
reg         row3_data;
reg  [1:0] per_frame_vsync_r;
reg  [1:0] per_frame_href_r;
reg  [1:0] per_frame_clken_r;


assign read_frame_href    = per_frame_href_r
[0] ;
assign read_frame_clken   = per_frame_clken_r
[0];
assign matrix_frame_vsync = per_frame_vsync_r
[1];
assign matrix_frame_href  = per_frame_href_r
[1] ;
assign matrix_frame_clken = per_frame_clken_r
[1];

//当前数据放在第3行
always@(posedge clk or negedge rst_n) begin
    if(!rst_n)
        row3_data <= 0;
    else begin
        if(per_frame_clken)
            row3_data <= per_img_y ;
        else
            row3_data <= row3_data ;
    end
end

//用于存储列数据的RAM
line_shift_ram_8bit  u_line_shift_ram_8bit
(
    .clock          (clk),
    .clken          (per_frame_clken),
    .per_frame_href (per_frame_href),

    .shiftin        ({8{per_img_y}}),
    .taps0x         (row2_data),
    .taps1x         (row1_data)
);

//将同步信号延迟两拍，用于同步化处理
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        per_frame_vsync_r <= 0;
        per_frame_href_r  <= 0;
        per_frame_clken_r <= 0;
    end
    else begin
        per_frame_vsync_r <= {
per_frame_vsync_r[0], per_frame_vsync };
        per_frame_href_r  <= {
per_frame_href_r[0],  per_frame_href  };
        per_frame_clken_r <= {
per_frame_clken_r[0], per_frame_clken };
```

```
    end
end

//在同步处理后的控制信号下，输出图像矩阵
always@(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 3'd0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 3'd0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 3'd0;
    end
    else if(read_frame_href) begin
        if(read_frame_clken) begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p12, matrix_p13,
row1_data};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p22, matrix_p23,
row2_data};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p32, matrix_p33,
row3_data};
        end
        else begin
            {matrix_p11, matrix_p12,
matrix_p13} <= {matrix_p11, matrix_p12,
matrix_p13};
            {matrix_p21, matrix_p22,
matrix_p23} <= {matrix_p21, matrix_p22,
matrix_p23};
            {matrix_p31, matrix_p32,
matrix_p33} <= {matrix_p31, matrix_p32,
matrix_p33};
        end
    end
    else begin
        {matrix_p11, matrix_p12, matrix_p13}
<= 3'd0;
        {matrix_p21, matrix_p22, matrix_p23}
<= 3'd0;
        {matrix_p31, matrix_p32, matrix_p33}
<= 3'd0;
    end
end

endmodule

module line_shift_ram_8bit(
    input          clock,
    input          clken,
    input          per_frame_href,

    input   [7:0]  shiftin,
    output  [7:0]  taps0x,
    output  [7:0]  taps1x
);

//reg define
reg  [2:0]  clken_dly;
reg  [9:0]  ram_rd_addr;
reg  [9:0]  ram_rd_addr_d0;
reg  [9:0]  ram_rd_addr_d1;
reg  [7:0]  shiftin_d0;
reg  [7:0]  shiftin_d1;
reg  [7:0]  shiftin_d2;
reg  [7:0]  taps0x_d0;



//在数据来到时，ram地址累加
always@(posedge clock)begin
    if(per_frame_href)
        if(clken)
            ram_rd_addr <= ram_rd_addr + 1 ;
        else
            ram_rd_addr <= ram_rd_addr ;
    else
        ram_rd_addr <= 0 ;
end

//时钟使能信号延迟三拍
always@(posedge clock) begin
    clken_dly <= { clken_dly[1:0] , clken };
end


//将ram地址延迟二拍
always@(posedge clock ) begin
    ram_rd_addr_d0 <= ram_rd_addr;
    ram_rd_addr_d1 <= ram_rd_addr_d0;
end

//输入数据延迟三拍
always@(posedge clock)begin
    shiftin_d0 <= shiftin;
    shiftin_d1 <= shiftin_d0;
    shiftin_d2 <= shiftin_d1;
end

//用于存储前一行图像的RAM
blk_mem_gen_0  u_ram_1024x8_0(
    .clka   (clock),
    .wea    (clken_dly[2]),     //在延迟的第三
个时钟周期，当前行的数据写入RAM0
    .addra (ram_rd_addr_d1),
    .dina  (shiftin_d2),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps0x)            //延迟一个时钟
周期，输出RAM0中前一行图像的数据
```

```
);

//寄存一次前一行图像的数据
always@(posedge clock ) begin
    taps0x_d0 <= taps0x;
end

blk_mem_gen_0  u_ram_1024x8_1(
    .clka   (clock),
    .wea   (clken_dly[1]),    //在延迟的第二
个时钟周期，将前一行图像的数据写入RAM1
    .addra (ram_rd_addr_d0),
    .dina  (taps0x_d0),
    .clkb  (clock),
    .addrb (ram_rd_addr),
    .doutb (taps1x)           //延迟一个时钟
周期，输出RAM1中前前一行图像的数据
);

endmodule
图像投影处理模块
水平投影
`timescale 1ns/1ns
module VIP_horizon_projection # (
    parameter   [9:0]   IMG_HDISP = 10'd640,
     //640*480
    parameter   [9:0]   IMG_VDISP = 10'd480
)(
    //global clock
    input               clk,
//cmos video pixel clock
    input               rst_n,
//global reset

    //Image data prepred to be processd
    input               per_frame_vsync,
//Prepared Image data vsync valid signal
    input               per_frame_href,
//Prepared Image data href vaild  signal
    input               per_frame_clken,
//Prepared Image data output/capture enable
clock
    input               per_img_Bit,
//Prepared Image Bit flag outout(1: Value,
0:inValid)

    //Image data has been processd
    output              post_frame_vsync,
//Processed Image data vsync valid signal
    output              post_frame_href,
//Processed Image data href vaild  signal
    output              post_frame_clken,
//Processed Image data output/capture enable
 clock
    output              post_img_Bit,
//Processed Image Bit flag outout(1: Value,
0:inValid)

    output reg [9:0]    max_line_up ,
//边沿坐标
    output reg [9:0]    max_line_down,

    input      [9:0]    horizon_start,
//投影起始列
    input      [9:0]    horizon_end
//投影结束列
);

reg [9:0]   max_pixel_up  ;
reg [9:0]   max_pixel_down;

reg         per_frame_vsync_r;
reg         per_frame_href_r;
reg         per_frame_clken_r;
reg         per_img_Bit_r;

reg         per_frame_vsync_r2;
reg         per_frame_href_r2;
reg         per_frame_clken_r2;
reg         per_img_Bit_r2;

assign  post_frame_vsync    =
per_frame_vsync_r2;
assign  post_frame_href     =
per_frame_href_r2;
assign  post_frame_clken    =
per_frame_clken_r2;
assign  post_img_Bit        =
per_img_Bit_r2;

//-----------------------------------------
//lag 1 将输入信号进行两级延迟，便于流水线处
理后同步
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        per_frame_vsync_r2  <= 0;
        per_frame_href_r2   <= 0;
        per_frame_clken_r2  <= 0;
        per_img_Bit_r2      <= 0;
        end
    else
        begin
        per_frame_vsync_r2  <=
per_frame_vsync_r   ;
        per_frame_href_r2   <=
per_frame_href_r    ;
        per_frame_clken_r2  <=
per_frame_clken_r   ;
        per_img_Bit_r2      <=  per_img_Bit_r
       ;
```

```
        end
end

always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
        per_frame_vsync_r    <= 0;
        per_frame_href_r     <= 0;
        per_frame_clken_r    <= 0;
        per_img_Bit_r        <= 0;
        end
    else
        begin
        per_frame_vsync_r    <=
per_frame_vsync ;
        per_frame_href_r     <=
per_frame_href  ;
        per_frame_clken_r    <=
per_frame_clken ;
        per_img_Bit_r        <=  per_img_Bit
    ;
        end
end

//得到场同步信号的边沿
wire vsync_pos_flag;
wire vsync_neg_flag;
wire hrefr_pos_flag;
wire hrefr_neg_flag;
assign vsync_pos_flag =   per_frame_vsync_r
& (~per_frame_vsync_r2);
assign vsync_neg_flag = (~per_frame_vsync_r)
&   per_frame_vsync_r2;
assign hrefr_pos_flag =   per_frame_href_r  &
 (~per_frame_href_r2);
assign hrefr_neg_flag = (~per_frame_href_r) &
 per_frame_href_r2;

//对输入的像素进行"行/场"方向计数，得到其纵横
 坐标
reg [9:0]   x_cnt;
reg [9:0]   y_cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(vsync_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(hrefr_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= y_cnt + 1'b1;
    end
    else if(per_frame_clken_r) begin
        x_cnt <= x_cnt + 1'b1;
    end
end

//寄存"行/场"方向计数——延迟一个周期
reg [9:0]   x_cnt_r;
reg [9:0]   y_cnt_r;

always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
            x_cnt_r <= 10'd0;
            y_cnt_r <= 10'd0;
        end
    else begin
            x_cnt_r <= x_cnt;
            y_cnt_r <= y_cnt;
        end
end


//水平方向投影
reg         ram_wr;     //ram写使能
reg  [9:0]  ram_wr_data;//ram写数据
wire [9:0]  ram_rd_data;//ram读数据
wire [9:0] ram_wr_addr;//ram写地址
wire [9:0] ram_rd_addr;//ram读地址
//reg  [9:0] cur_h_data; //当前行的数据

//ram写使能——除了当前帧的最后一行，都写使能
always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        ram_wr <= 1'b0;
    end
    else if(per_frame_clken && (y_cnt !=
 IMG_VDISP - 1'b1))
        ram_wr <= 1'b1;
    else
        ram_wr <= 1'b0;
end

//对所有列进行水平投影
always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        ram_wr_data <= 10'd0;
    end
    else if(y_cnt == 10'd0)
        ram_wr_data <= 10'd0;
    else if(hrefr_pos_flag)
        ram_wr_data <= 10'd0;
    else if((x_cnt > horizon_start) && (x_cnt
  < horizon_end))
        ram_wr_data <= ram_wr_data +
```

```
per_img_Bit_r;
end

//在当前帧的第一行，对RAM所有地址都写入0
assign ram_wr_addr = (y_cnt == 10'd0)  ?
x_cnt : y_cnt_r;
//在当前帧的第一行和最后一行，需要遍历RAM中的
数据
assign ram_rd_addr = ((y_cnt == 10'd0) || (
y_cnt == IMG_VDISP - 1'b1))  ?  x_cnt :
y_cnt;


projection_ram u_projection_ram (
    .clka  (clk            ),// input wire
clka
    .wea   (ram_wr         ),// input wire [0
: 0] wea
    .addra (ram_wr_addr    ),// input wire [9
: 0] addra
    .dina  (ram_wr_data    ),// input wire [9
: 0] dina

    .clkb  (clk            ),// input wire
clkb
    .addrb (ram_rd_addr    ),// input wire [9
: 0] addrb
    .doutb (ram_rd_data    ) // output wire [9
 : 0] doutb
);

reg [9:0] rd_data_d1;
reg [9:0] rd_data_d2;
reg [9:0] rd_data_d3;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        rd_data_d1 <= 10'd0;
        rd_data_d2 <= 10'd0;
    end
    else if(per_frame_clken) begin
        rd_data_d1 <= ram_rd_data;
        rd_data_d2 <= rd_data_d1;
        rd_data_d3 <= rd_data_d2;
    end
end

reg [9:0] max_num1  ;
reg [9:0] max_y1    ;
reg [9:0] max_num2  ;
reg [9:0] max_y2    ;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        max_y1   <= 10'd0;
        max_num1 <= 10'd0;
        max_y2   <= 10'd0;
        max_num2 <= 10'd0;
    end
    else if(per_frame_clken) begin

        if(y_cnt == IMG_VDISP - 1'b1) begin
  //图像的最后一行，遍历RAM中的数据，求极值
            if( (rd_data_d2 > rd_data_d1) &&
(rd_data_d2>10'd30) && (max_y1==10'd0))begin
//第一个极大值大于30
                max_y1      <= x_cnt_r-3;
                max_num1    <= rd_data_d1;
            end
            //下降沿判决
            if( (rd_data_d2 < rd_data_d1) &&
(rd_data_d2>10'd30)) begin//最后一个极大值大
于30
                max_y2      <= x_cnt_r-3;
                max_num2    <= rd_data_d1;
            end
        end

    end
    else if(vsync_neg_flag)begin
        max_y1   <= 10'd0;
        max_num1 <= 10'd0;
        max_y2   <= 10'd0;
        max_num2 <= 10'd0;
    end
end

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        max_line_up  <= 10'd0;
        max_line_down <= 10'd0;
        max_pixel_up  <= 10'd0;
        max_pixel_down <= 10'd0;
    end
    else if(vsync_pos_flag) begin
        max_line_up   <= max_y1;
        max_pixel_up  <= max_num1;

        max_line_down  <= max_y2;
        max_pixel_down <= max_num2;
    end
end

endmodule
竖直投影
`timescale 1ns/1ns
module VIP_vertical_projection #(
    parameter   [9:0]   IMG_HDISP = 10'd640,
     //640*480
    parameter   [9:0]   IMG_VDISP = 10'd480
)(
    //global clock
```

```
    input                clk,
//cmos video pixel clock
    input                rst_n,
//global reset

    //Image data prepred to be processd
    input                per_frame_vsync,//
Prepared Image data vsync valid signal
    input                per_frame_href ,//
Prepared Image data href vaild  signal
    input                per_frame_clken,//
Prepared Image data output/capture enable
clock
    input                per_img_Bit    ,//
Prepared Image Bit flag outout(1: Value, 0:
inValid)

    //Image data has been processd
    output               post_frame_vsync,
//Processed Image data vsync valid signal
    output               post_frame_href,
//Processed Image data href vaild  signal
    output               post_frame_clken,
//Processed Image data output/capture enable
 clock
    output               post_img_Bit,
//Processed Image Bit flag outout(1: Value,
0:inValid)

    output reg [9:0]     max_line_left ,
//边沿坐标
    output reg [9:0]     max_line_right,

    input      [9:0]     vertical_start,
//投影起始行
    input      [9:0]     vertical_end
//投影结束行
);

reg [9:0]   max_pixel_left ;
reg [9:0]   max_pixel_right;

reg         per_frame_vsync_r;
reg         per_frame_href_r;
reg         per_frame_clken_r;
reg         per_img_Bit_r;

reg         per_frame_vsync_r2;
reg         per_frame_href_r2;
reg         per_frame_clken_r2;
reg         per_img_Bit_r2;

assign  post_frame_vsync    =
per_frame_vsync_r2;
assign  post_frame_href     =
per_frame_href_r2;
assign  post_frame_clken    =
per_frame_clken_r2;
assign  post_img_Bit        =
per_img_Bit_r2;

//-----------------------------------------
//lag 1 clocks signal sync
always@(posedge clk or negedge rst_n)begin
if(!rst_n)begin
    per_frame_vsync_r2  <= 0;
    per_frame_href_r2   <= 0;
    per_frame_clken_r2  <= 0;
    per_img_Bit_r2      <= 0;
end
else begin
    per_frame_vsync_r2  <=  per_frame_vsync_r
    ;
    per_frame_href_r2   <=  per_frame_href_r
    ;
    per_frame_clken_r2  <=  per_frame_clken_r
    ;
    per_img_Bit_r2      <=  per_img_Bit_r
     ;
end
end

//-----------------------------------------
//lag 1 clocks signal sync
always@(posedge clk or negedge rst_n)begin
if(!rst_n) begin
    per_frame_vsync_r   <= 0;
    per_frame_href_r    <= 0;
    per_frame_clken_r   <= 0;
    per_img_Bit_r       <= 0;
end
else begin
    per_frame_vsync_r   <=  per_frame_vsync ;
    per_frame_href_r    <=  per_frame_href  ;
    per_frame_clken_r   <=  per_frame_clken ;
    per_img_Bit_r       <=  per_img_Bit     ;
end
end

wire vsync_pos_flag;
wire vsync_neg_flag;
wire hrefr_neg_flag;
assign vsync_pos_flag =   per_frame_vsync_r
& (~per_frame_vsync_r2);
assign vsync_neg_flag = (~per_frame_vsync_r)
&   per_frame_vsync_r2;
assign hrefr_neg_flag = (~per_frame_href_r) &
 per_frame_href_r2;


//对输入的像素进行"行/场"方向计数，得到其纵横
 坐标
```

```
reg [9:0]   x_cnt;
reg [9:0]   y_cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(vsync_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(hrefr_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= y_cnt + 1'b1;
    end
    else if(per_frame_clken_r) begin
        x_cnt <= x_cnt + 1'b1;
    end
end

//-----------------------------------------
//寄存"行/场"方向计数
reg [9:0]   x_cnt_d1;
reg [9:0]   y_cnt_d1;
reg [9:0]   x_cnt_r;
reg [9:0]   y_cnt_r;

always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
            x_cnt_r <= 10'd0;
            y_cnt_r <= 10'd0;
        end
    else begin
            x_cnt_d1 <= x_cnt;
            x_cnt_r  <= x_cnt_d1;
            y_cnt_d1 <= y_cnt;
            y_cnt_r  <= y_cnt_d1;
        end
end

//-----------------------------------------
//竖直方向投影
reg         clken_d1;
reg         ram_wr;
wire  [9:0] ram_wr_data;
wire  [9:0] ram_rd_data;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        clken_d1  <= 1'b0;
        ram_wr    <= 1'b0;
    end
    else if(per_frame_clken)begin
        clken_d1  <= 1'b1;
        ram_wr <= clken_d1;
    end
    else begin
        clken_d1  <= 1'b0;
        ram_wr <= clken_d1;
    end
end

//在指定的行数之间进行投影

assign ram_wr_data = (y_cnt == 10'd0) ? 10'd0
  :                       //第一行，初始化RAM为
0
                          ((y_cnt >
vertical_start) && (y_cnt < vertical_end)) ?
 (ram_rd_data + per_img_Bit_r) :
                              ram_rd_data;
projection_ram u_projection_ram (
  .clka  (clk           ),// input wire clka
  .wea   (ram_wr        ),// input wire [0 :
0] wea
  .addra (x_cnt_d1       ),// input wire [9 :
 0] addra
  .dina  (ram_wr_data   ),// input wire [9 :
0] dina

  .clkb  (clk           ),// input wire clkb
  .addrb (x_cnt         ),// input wire [9 :
0] addrb
  .doutb (ram_rd_data   ) // output wire [9 :
 0] doutb
);

reg [9:0] rd_data_d1;
reg [9:0] rd_data_d2;
reg [9:0] rd_data_d3;
reg [9:0] rd_data_d4;
reg [9:0] rd_data_d5;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        rd_data_d1 <= 10'd0;
        rd_data_d2 <= 10'd0;
        rd_data_d3 <= 10'd0;
        rd_data_d4 <= 10'd0;
        rd_data_d5 <= 10'd0;
    end
    else if(per_frame_clken) begin
        rd_data_d1 <= ram_rd_data;
        rd_data_d2 <= rd_data_d1;
        rd_data_d3 <= rd_data_d2;
        rd_data_d4 <= rd_data_d3;
        rd_data_d5 <= rd_data_d4;
    end
end
```

```
reg [9:0] max_num1  ;
reg [9:0] max_x1    ;
reg [9:0] max_num2  ;
reg [9:0] max_x2    ;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        max_num1 <= 10'd0;
        max_x1   <= 10'd0;
        max_num2 <= 10'd0;
        max_x2   <= 10'd0;
    end
    else if(per_frame_clken) begin

        if(y_cnt == IMG_VDISP - 1'b1) begin
                    //上升沿判决
            if( (rd_data_d2 > rd_data_d1) &&
(rd_data_d2>10'd30) && (max_x1==10'd0))begin
//第一个极大值大于30
                max_x1      <= x_cnt_r-2;
                max_num1    <= rd_data_d1;
            end
            //下降沿判决
            if( (rd_data_d2 < rd_data_d1) &&
(rd_data_d2>10'd30)) begin//最后一个极大值大
于30
                max_x2      <= x_cnt_r-2;
                max_num2    <= rd_data_d1;
            end
        end
    end
    else if(vsync_neg_flag) begin
        max_num1 <= 10'd0;
        max_x1   <= 10'd0;
        max_num2 <= 10'd0;
        max_x2   <= 10'd0;
    end
end

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        max_line_left  <= 10'd0;
        max_line_right <= 10'd0;
        max_pixel_left  <= 10'd0;
        max_pixel_right <= 10'd0;
    end
    else if(vsync_pos_flag) begin
        max_line_left   <= max_x1;
        max_pixel_left  <= max_num1;

        max_line_right  <= max_x2;
        max_pixel_right <= max_num2;
    end
end
```

```
endmodule
字符二值化
module char_binarization # (
    parameter BIN_THRESHOLD = 128 //二值化阈
 值
)(
    input               clk             ,
// 时钟信号
    input               rst_n           ,
// 复位信号（低有效）
    //输入视频流
    input               per_frame_vsync,
    input               per_frame_href ,
    input               per_frame_clken,
    input       [7:0]   per_frame_Red   ,

    //车牌边界
    input    [9:0]      plate_boarder_up    ,
//输入的车牌候选区域
    input    [9:0]      plate_boarder_down  ,
    input    [9:0]      plate_boarder_left  ,
    input    [9:0]      plate_boarder_right ,
    input               plate_exist_flag    ,
    //输出视频流
    output  wire        post_frame_vsync,
    output  wire        post_frame_href ,
    output  wire        post_frame_clken,
    output  reg         post_frame_Bit
);

//-----------------------------------------
//对输入视频流进行延迟，获取其边沿

reg         per_frame_vsync_r;
reg         per_frame_href_r ;
reg         per_frame_clken_r;
reg [7:0]   per_frame_Red_r  ;

reg         per_frame_vsync_r2;
reg         per_frame_href_r2 ;
reg         per_frame_clken_r2;
reg [7:0]   per_frame_Red_r2  ;

wire vsync_pos_flag;
wire vsync_neg_flag;
wire hrefr_neg_flag;

assign  post_frame_vsync =
 per_frame_vsync_r2;
assign  post_frame_href  =  per_frame_href_r2
  ;
assign  post_frame_clken =
 per_frame_clken_r2;
//assign  post_frame_Bit   =
 per_frame_Red_r2  ;
```

```
always@(posedge clk or negedge rst_n)
if(!rst_n)begin
    per_frame_vsync_r    <= 0;
    per_frame_href_r     <= 0;
    per_frame_clken_r    <= 0;
    per_frame_Red_r      <= 0;

    per_frame_vsync_r2   <= 0;
    per_frame_href_r2    <= 0;
    per_frame_clken_r2   <= 0;
    per_frame_Red_r2     <= 0;
end
else begin
    per_frame_vsync_r    <=  per_frame_vsync ;
    per_frame_href_r     <=  per_frame_href  ;
    per_frame_clken_r    <=  per_frame_clken ;
    per_frame_Red_r      <=  per_frame_Red   ;

    per_frame_vsync_r2   <=  per_frame_vsync_r
;
    per_frame_href_r2    <=  per_frame_href_r
;
    per_frame_clken_r2   <=  per_frame_clken_r
;
    per_frame_Red_r2     <=  per_frame_Red_r
;
end



assign vsync_pos_flag =   per_frame_vsync_r
& (~per_frame_vsync_r2);
assign vsync_neg_flag = (~per_frame_vsync_r)
&   per_frame_vsync_r2;
assign hrefr_neg_flag = (~per_frame_href_r)
&   per_frame_href_r2;

//-----------------------------------------
//对输入的像素进行"行/场"方向计数，得到其纵横
 坐标
reg [9:0]   x_cnt;
reg [9:0]   y_cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(vsync_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(hrefr_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= y_cnt + 1'b1;
    end
    else if(per_frame_clken_r) begin
        x_cnt <= x_cnt + 1'b1;
    end
end
//-----------------------------------------
//二值化
always @(posedge clk or negedge rst_n)
if(!rst_n)
    post_frame_Bit <= 1'b0;
else if(plate_exist_flag)
    if((y_cnt > plate_boarder_up  ) && (y_cnt
  < plate_boarder_down ) &&
       (x_cnt > plate_boarder_left) && (x_cnt
  < plate_boarder_right)  ) begin
        if(per_frame_Red_r > BIN_THRESHOLD)
 //阈值
            post_frame_Bit <= 1'b1;
        else
            post_frame_Bit <= 1'b0;
    end
    else
        post_frame_Bit <= 1'b0;
else begin
    if(per_frame_Red_r > BIN_THRESHOLD)  //阈
 值
        post_frame_Bit <= 1'b1;
    else
        post_frame_Bit <= 1'b0;
end
endmodule
字符投影处理模块
水平投影
`timescale 1ns/1ns
module char_horizon_projection # (
    parameter    [9:0]    IMG_HDISP = 10'd640
 ,    //640*480
    parameter    [9:0]    IMG_VDISP = 10'd480
)(
    //global clock
    input                clk,
   //cmos video pixel clock
    input                rst_n,
   //global reset

    //Image data prepred to be processd
    input                per_frame_vsync,
 //Prepared Image data vsync valid signal
    input                per_frame_href ,
 //Prepared Image data href vaild  signal
    input                per_frame_clken,
 //Prepared Image data output/capture enable
 clock
    input                per_img_Bit    ,
 //Prepared Image Bit flag outout(1: Value,
 0:inValid)
```

```
    //Image data has been processd
    output wire          post_frame_vsync,
//Processed Image data vsync valid signal
    output wire          post_frame_href,
//Processed Image data href vaild  signal
    output wire          post_frame_clken,
//Processed Image data output/capture enable
 clock
    output wire          post_img_Bit,
//Processed Image Bit flag outout(1: Value,
0:inValid)

    output reg [9:0]     max_line_up ,
 //边沿坐标
    output reg [9:0]     max_line_down,

    input      [9:0]     horizon_start,
  //投影起始列
    input      [9:0]     horizon_end
   //投影结束列
);

reg [9:0]     max_pixel_up  ;
reg [9:0]     max_pixel_down;

reg           per_frame_vsync_r;
reg           per_frame_href_r ;
reg           per_frame_clken_r;
reg           per_img_Bit_r    ;

reg           per_frame_vsync_r2;
reg           per_frame_href_r2 ;
reg           per_frame_clken_r2;
reg           per_img_Bit_r2    ;

assign  post_frame_vsync  =
 per_frame_vsync_r2;
assign  post_frame_href   =
 per_frame_href_r2 ;
assign  post_frame_clken  =
 per_frame_clken_r2;
assign  post_img_Bit      =  per_img_Bit_r2
   ;

//-----------------------------------------
//lag 1 将输入信号进行两级延迟，便于流水线处
 理后同步
always@(posedge clk or negedge rst_n)
if(!rst_n) begin
    per_frame_vsync_r2 <= 1'b0;
    per_frame_href_r2  <= 1'b0;
    per_frame_clken_r2 <= 1'b0;
    per_img_Bit_r2     <= 1'b0;
end
else begin
    per_frame_vsync_r2 <= per_frame_vsync_r;
    per_frame_href_r2  <= per_frame_href_r ;
    per_frame_clken_r2 <= per_frame_clken_r;
    per_img_Bit_r2     <= per_img_Bit_r    ;
end

always@(posedge clk or negedge rst_n)
if(!rst_n) begin
    per_frame_vsync_r  <= 1'b0;
    per_frame_href_r   <= 1'b0;
    per_frame_clken_r  <= 1'b0;
    per_img_Bit_r      <= 1'b0;
end
else begin
    per_frame_vsync_r  <= per_frame_vsync  ;
    per_frame_href_r   <= per_frame_href   ;
    per_frame_clken_r  <= per_frame_clken  ;
    per_img_Bit_r      <= per_img_Bit      ;
end

//得到场同步信号的边沿
wire vsync_pos_flag;
wire vsync_neg_flag;
wire hrefr_pos_flag;
wire hrefr_neg_flag;
assign vsync_pos_flag =   per_frame_vsync_r
 & (~per_frame_vsync_r2);
assign vsync_neg_flag = (~per_frame_vsync_r)
 &   per_frame_vsync_r2;
assign hrefr_pos_flag =   per_frame_href_r  &
  (~per_frame_href_r2);
assign hrefr_neg_flag = (~per_frame_href_r) &
    per_frame_href_r2;

//-----------------------------------------
//对输入的像素进行"行/场"方向计数，得到其纵横
 坐标
reg [9:0]   x_cnt;
reg [9:0]   y_cnt;
always@(posedge clk or negedge rst_n)
if(!rst_n)begin
    x_cnt <= 10'd0;
    y_cnt <= 10'd0;
end
else if(vsync_pos_flag)begin
    x_cnt <= 10'd0;
    y_cnt <= 10'd0;
end
else if(hrefr_neg_flag)begin
    x_cnt <= 10'd0;
    y_cnt <= y_cnt + 1'b1;
end
else if(per_frame_clken_r) begin
    x_cnt <= x_cnt + 1'b1;
end

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

```
//寄存"行/场"方向计数--延迟一个周期
reg [9:0]   x_cnt_r;
reg [9:0]   y_cnt_r;

always@(posedge clk or negedge rst_n)
if(!rst_n) begin
    x_cnt_r <= 10'd0;
    y_cnt_r <= 10'd0;
end
else begin
    x_cnt_r <= x_cnt;
    y_cnt_r <= y_cnt;
end

//-----------------------------------------
//水平方向投影
reg        ram_wr;     //ram写使能
reg  [9:0] ram_wr_data;//ram写数据
wire [9:0] ram_rd_data;//ram读数据
wire [9:0] ram_wr_addr;//ram写地址
wire [9:0] ram_rd_addr;//ram读地址

//ram写使能--除了当前帧的最后一行，都写使能
always @ (posedge clk or negedge rst_n)
if(!rst_n)
    ram_wr <= 1'b0;
else if(per_frame_clken && (y_cnt !=
 IMG_VDISP - 1'b1))
    ram_wr <= 1'b1;
else
    ram_wr <= 1'b0;

//对所有列进行水平投影
always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        ram_wr_data <= 10'd0;
    end
    else if(y_cnt == 10'd0)//图像的第一行，
 RAM清零
        ram_wr_data <= 10'd0;
    else if(hrefr_pos_flag)
        ram_wr_data <= 10'd0;
    else if((x_cnt > horizon_start) && (x_cnt
  < horizon_end))
        ram_wr_data <= ram_wr_data +
 per_img_Bit_r;
end

//在当前帧的第一行，对RAM所有地址都写入0
assign ram_wr_addr = (y_cnt == 10'd0)  ?
 x_cnt : y_cnt_r;
//在当前帧的第一行和最后一行，需要遍历RAM中的
 数据
assign ram_rd_addr = ((y_cnt == 10'd0) || (
 y_cnt == IMG_VDISP - 1'b1))  ?  x_cnt :
 y_cnt;

projection_ram u_projection_ram (
    .clka  (clk           ),// input wire
 clka
    .wea   (ram_wr        ),// input wire [0
 : 0] wea
    .addra (ram_wr_addr   ),// input wire [9
 : 0] addra
    .dina  (ram_wr_data   ),// input wire [9
 : 0] dina

    .clkb  (clk           ),// input wire
 clkb
    .addrb (ram_rd_addr   ),// input wire [9
 : 0] addrb
    .doutb (ram_rd_data   ) // output wire [9
  : 0] doutb
);

reg [9:0] rd_data_d1;
reg [9:0] rd_data_d2;
reg [9:0] rd_data_d3;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        rd_data_d1 <= 10'd0;
        rd_data_d2 <= 10'd0;
    end
    else if(per_frame_clken) begin
        rd_data_d1 <= ram_rd_data;
        rd_data_d2 <= rd_data_d1;
        rd_data_d3 <= rd_data_d2;
    end
end

reg [9:0] max_num1  ;
reg [9:0] max_y1    ;
reg [9:0] max_num2  ;
reg [9:0] max_y2    ;

reg [9:0] max_y_first  ;
reg [9:0] min_y_first  ;
reg [9:0] min_y_last   ;
reg [9:0] max_y_last   ;
reg [3:0] edge_cnt     ;


always @ (posedge clk or negedge rst_n)
if(!rst_n) begin
    max_y_first  <= 10'd0;
    min_y_first  <= 10'd0;
    min_y_last   <= 10'd0;
    max_y_last   <= 10'd0;
    edge_cnt     <= 4'd0 ;
end
else if(per_frame_clken) begin
```

```
    if(y_cnt == IMG_VDISP - 1'b1) begin     //图像的最后一行，遍历RAM中的数据，求极值
        case(edge_cnt)
            //第一个大于30的极大值或陡升的上升沿
            4'd0:begin
                if(rd_data_d3==10'd0 && rd_data_d1>10'd150)begin//该条件用于检测陡升的上升沿
                    min_y_first <= x_cnt_r - 3;
                    edge_cnt    <= edge_cnt + 1'b1;
                end
                else if((rd_data_d2 > rd_data_d1) && (rd_data_d2>10'd30))begin//第一个大于30的极大值
                    max_y_first <= x_cnt_r - 3;
                    edge_cnt    <= edge_cnt + 1'b1;
                end
            end
            //遇到的第一个极小值（上升趋势大于8像素）
            4'd1:begin
                if(rd_data_d2 + 10'd8 < rd_data_d1)begin
                    min_y_first <= x_cnt_r - 3;
                    edge_cnt    <= edge_cnt + 1'b1;
                end
            end
            //不断更新遇到的极小值（上升趋势大于8像素）
            4'd2:begin
                if(rd_data_d2 + 10'd8 < rd_data_d1)begin
                    min_y_last <= x_cnt_r - 3;
                    edge_cnt    <= edge_cnt + 1'b1;
                end
            end
            //遇到的最近一个大于30的极大值
            4'd3:begin
                if( (rd_data_d2 > rd_data_d1) && (rd_data_d2>10'd30))begin
                    max_y_last <= x_cnt_r - 3;
                    edge_cnt   <= edge_cnt + 1'b1;
                end
            end
            //捕捉降低到0的下降沿
            4'd4:begin
                if(rd_data_d1==10'd0 && rd_data_d3>=10'd150)
                    min_y_last <= x_cnt_r - 3;
                //观测是否还有新的上升趋势（上升趋势大于8像素）
                else if((rd_data_d2+10'd8 < rd_data_d1) && (rd_data_d2!=10'd0))begin
                    min_y_last <= x_cnt_r - 3;
                    edge_cnt   <= 4'd3;
                end
            end
            default: edge_cnt <= 4'd0;
        endcase
    end
    else begin
        edge_cnt     <= 4'd0 ;
    end
end
else if(vsync_neg_flag)begin
    max_y_first <= 10'd0;
    min_y_first <= 10'd0;
    min_y_last  <= 10'd0;
    max_y_last  <= 10'd0;
end

always @ (posedge clk or negedge rst_n)
if(!rst_n) begin
    max_line_up  <= 10'd0;
    max_line_down <= 10'd0;
end
else if(vsync_pos_flag) begin
    max_line_up   <= min_y_first;
    max_line_down <= min_y_last;
end

endmodule
竖直投影
`timescale 1ns/1ns
module char_vertical_projection #(
    parameter    [9:0]    IMG_HDISP = 10'd640,    //640*480
    parameter    [9:0]    IMG_VDISP = 10'd480)(
    //global clock
    input                clk,
  //cmos video pixel clock
    input                rst_n,
  //global reset

    //Image data prepred to be processd
    input                per_frame_vsync,//Prepared Image data vsync valid signal
    input                per_frame_href ,//
```

```
Prepared Image data href vaild  signal
    input                per_frame_clken,//
Prepared Image data output/capture enable
clock
    input                per_img_Bit     ,//
Prepared Image Bit flag outout(1: Value, 0:
inValid)

    input      [9:0]     vertical_start,  //
投影起始行
    input      [9:0]     vertical_end,    //
投影结束行
    //输出边沿坐标
    output reg [9:0]    char1_line_left ,//垂
直投影结果
    output reg [9:0]    char1_line_right,
    output reg [9:0]    char2_line_left ,
    output reg [9:0]    char2_line_right,
    output reg [9:0]    char3_line_left ,
    output reg [9:0]    char3_line_right,
    output reg [9:0]    char4_line_left ,
    output reg [9:0]    char4_line_right,
    output reg [9:0]    char5_line_left ,
    output reg [9:0]    char5_line_right,
    output reg [9:0]    char6_line_left ,
    output reg [9:0]    char6_line_right,
    output reg [9:0]    char7_line_left ,
    output reg [9:0]    char7_line_right,

    //Image data has been processd
    output                post_frame_vsync,//
Processed Image data vsync valid signal
    output                post_frame_href ,//
Processed Image data href vaild  signal
    output                post_frame_clken,//
Processed Image data output/capture enable
clock
    output                post_img_Bit     //
Processed Image Bit flag outout(1: Value, 0:
inValid)
);

reg [9:0]     max_pixel_left ;
reg [9:0]     max_pixel_right;

reg            per_frame_vsync_r;
reg            per_frame_href_r;
reg            per_frame_clken_r;
reg          per_img_Bit_r;

reg            per_frame_vsync_r2;
reg            per_frame_href_r2;
reg            per_frame_clken_r2;
reg         per_img_Bit_r2;

assign    post_frame_vsync     =
per_frame_vsync_r2;
assign    post_frame_href      =
per_frame_href_r2;
assign    post_frame_clken     =
per_frame_clken_r2;
assign    post_img_Bit         =
per_img_Bit_r2;

//-----------------------------------------
//lag 1 clocks signal sync
always@(posedge clk or negedge rst_n)begin
if(!rst_n)begin
    per_frame_vsync_r2     <= 0;
    per_frame_href_r2     <= 0;
    per_frame_clken_r2     <= 0;
    per_img_Bit_r2        <= 0;
end
else begin
    per_frame_vsync_r2     <=
per_frame_vsync_r     ;
    per_frame_href_r2    <=
per_frame_href_r     ;
    per_frame_clken_r2     <=
per_frame_clken_r     ;
    per_img_Bit_r2        <=
per_img_Bit_r        ;
end
end

//-----------------------------------------
//lag 1 clocks signal sync
always@(posedge clk or negedge rst_n)begin
if(!rst_n) begin
    per_frame_vsync_r     <= 0;
    per_frame_href_r     <= 0;
    per_frame_clken_r     <= 0;
    per_img_Bit_r        <= 0;
end
else begin
    per_frame_vsync_r     <=
per_frame_vsync    ;
    per_frame_href_r    <=     per_frame_href
    ;
    per_frame_clken_r     <=
per_frame_clken    ;
    per_img_Bit_r        <=     per_img_Bit
      ;
end
end

wire vsync_pos_flag;
wire vsync_neg_flag;
wire hrefr_neg_flag;
assign vsync_pos_flag =   per_frame_vsync_r
& (~per_frame_vsync_r2);
assign vsync_neg_flag = (~per_frame_vsync_r)
```

```
 &   per_frame_vsync_r2;
assign hrefr_neg_flag = (~per_frame_href_r) &
  per_frame_href_r2;

//-----------------------------------------
//对输入的像素进行"行/场"方向计数，得到其纵横
 坐标
reg [9:0]      x_cnt;
reg [9:0]   y_cnt;
always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(vsync_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= 10'd0;
    end
    else if(hrefr_neg_flag)begin
        x_cnt <= 10'd0;
        y_cnt <= y_cnt + 1'b1;
    end
    else if(per_frame_clken_r) begin
        x_cnt <= x_cnt + 1'b1;
    end
end

//-----------------------------------------
//寄存"行/场"方向计数
reg [9:0]      x_cnt_d1;
reg [9:0]   y_cnt_d1;
reg [9:0]      x_cnt_r;
reg [9:0]   y_cnt_r;

always@(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        begin
            x_cnt_r <= 10'd0;
            y_cnt_r <= 10'd0;
        end
    else begin
            x_cnt_d1 <= x_cnt;
            x_cnt_r  <= x_cnt_d1;
            y_cnt_d1 <= y_cnt;
            y_cnt_r  <= y_cnt_d1;
        end
end

//-----------------------------------------
//竖直方向投影
reg          clken_d1;
reg          ram_wr;
wire  [9:0]    ram_wr_data;
wire  [9:0]    ram_rd_data;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        clken_d1  <= 1'b0;
        ram_wr    <= 1'b0;
    end
    else if(per_frame_clken)begin
        clken_d1  <= 1'b1;
        ram_wr <= clken_d1;
    end
    else begin
        clken_d1  <= 1'b0;
        ram_wr <= clken_d1;
    end
end

assign ram_wr_data = (y_cnt == 10'd0) ? 10'd0
  :                      //第一行，初始化RAM为
 0
                         ((y_cnt >
 vertical_start) && (y_cnt < vertical_end)) ?
  (ram_rd_data + per_img_Bit_r) :
                              ram_rd_data;

projection_ram u_projection_ram (
  .clka  (clk           ),// input wire clka
  .wea   (ram_wr        ),// input wire [0 :
 0] wea
  .addra (x_cnt_d1       ),// input wire [9 :
  0] addra
  .dina  (ram_wr_data   ),// input wire [9 :
 0] dina

  .clkb  (clk           ),// input wire clkb
  .addrb (x_cnt         ),// input wire [9 :
 0] addrb
  .doutb (ram_rd_data   ) // output wire [9 :
  0] doutb
);

reg [9:0] rd_data_d1;
reg [9:0] rd_data_d2;
reg [9:0] rd_data_d3;
reg [9:0] rd_data_d4;
reg [9:0] rd_data_d5;

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        rd_data_d1 <= 10'd0;
        rd_data_d2 <= 10'd0;
        rd_data_d3 <= 10'd0;
        rd_data_d4 <= 10'd0;
        rd_data_d5 <= 10'd0;
    end
    else if(per_frame_clken) begin
        rd_data_d1 <= ram_rd_data;
```

```
        rd_data_d2 <= rd_data_d1;
        rd_data_d3 <= rd_data_d2;
        rd_data_d4 <= rd_data_d3;
        rd_data_d5 <= rd_data_d4;
    end
end

//-----------------------------------------
//进行最后的判决
reg [3:0] char_left_cnt;//对字符的左边沿进行计数
reg [3:0] char_right_cnt;//对字符的右边沿进行计数
reg [5:0] char_edge_cnt;//对字符的边沿进行计数
reg [9:0] max1_line_left ;
reg [9:0] max1_line_right;
reg [9:0] max2_line_left ;
reg [9:0] max2_line_right;
reg [9:0] max3_line_left ;
reg [9:0] max3_line_right;
reg [9:0] max4_line_left ;
reg [9:0] max4_line_right;
reg [9:0] max5_line_left ;
reg [9:0] max5_line_right;
reg [9:0] max6_line_left ;
reg [9:0] max6_line_right;
reg [9:0] max7_line_left ;
reg [9:0] max7_line_right;

wire char_posedge;
wire char_negedge;
assign char_posedge = (rd_data_d3==10'd0 && rd_data_d1>=10'd5);
assign char_negedge = (rd_data_d3>=10'd5 && rd_data_d1==10'd0);

always @ (posedge clk or negedge rst_n)
if(!rst_n) begin
//    char_left_cnt    <= 4'd0;
//    char_right_cnt   <= 4'd0;
    char_edge_cnt   <= 6'd0 ;
    max1_line_left  <= 10'd0;
    max1_line_right <= 10'd0;
    max2_line_left  <= 10'd0;
    max2_line_right <= 10'd0;
    max3_line_left  <= 10'd0;
    max3_line_right <= 10'd0;
    max4_line_left  <= 10'd0;
    max4_line_right <= 10'd0;
    max5_line_left  <= 10'd0;
    max5_line_right <= 10'd0;
    max6_line_left  <= 10'd0;
    max6_line_right <= 10'd0;
    max7_line_left  <= 10'd0;
    max7_line_right <= 10'd0;
end
else if(per_frame_clken) begin
    if(y_cnt == IMG_VDISP - 1'b1) begin
        //逐个进行边沿的判断
        case(char_edge_cnt)
            //汉字边界
            6'd0:begin
                max1_line_left  <= char_posedge ? (x_cnt_r-2) : max1_line_left;
                if(char_negedge)begin
                    char_edge_cnt   <= char_edge_cnt+1'b1;
                    max1_line_right <= x_cnt_r-2;
                end
            end
            6'd1:begin
                max1_line_right <= char_negedge ? (x_cnt_r-2) : max1_line_right;
                if(char_posedge)begin
                    char_edge_cnt   <= char_edge_cnt+1'b1;
                    max2_line_left  <= x_cnt_r-2;
                end
            end
            //第一个字母边界
            6'd2:begin
                max2_line_left  <= char_posedge ? (x_cnt_r-2) : max2_line_left;
                if(char_negedge)begin
                    char_edge_cnt   <= char_edge_cnt+1'b1;
                    max2_line_right <= x_cnt_r-2;
                end
            end
            6'd3:begin
                max2_line_right <= char_negedge ? (x_cnt_r-2) : max2_line_right;
                if(char_posedge)begin
                    char_edge_cnt   <= char_edge_cnt+1'b1;
                end
            end
            //点
            6'd4:begin
                if(char_negedge)begin
                    char_edge_cnt   <= char_edge_cnt+1'b1;
                end
            end
            6'd5:begin
                if(char_posedge)begin
```

```
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max3_line_left   <=
x_cnt_r-2;
                end
            end
            //数字1
            6'd6:begin
                max3_line_left   <=
char_posedge ? (x_cnt_r-2) : max3_line_left;
                if(char_negedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max3_line_right <=
x_cnt_r-2;
                end
            end
            6'd7:begin
                max3_line_right <=
char_negedge ? (x_cnt_r-2) : max3_line_right
;
                if(char_posedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max4_line_left   <=
x_cnt_r-2;
                end
            end
            //数字2
            6'd8:begin
                max4_line_left   <=
char_posedge ? (x_cnt_r-2) : max4_line_left;
                if(char_negedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max4_line_right <=
x_cnt_r-2;
                end
            end
            6'd9:begin
                max4_line_right <=
char_negedge ? (x_cnt_r-2) : max4_line_right
;
                if(char_posedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max5_line_left   <=
x_cnt_r-2;
                end
            end
            //数字3
            6'd10:begin
                max5_line_left   <=
char_posedge ? (x_cnt_r-2) : max5_line_left;
                if(char_negedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max5_line_right <=
x_cnt_r-2;
                end
            end
            6'd11:begin
                max5_line_right <=
char_negedge ? (x_cnt_r-2) : max5_line_right
;
                if(char_posedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max6_line_left   <=
x_cnt_r-2;
                end
            end
            //数字4
            6'd12:begin
                max6_line_left   <=
char_posedge ? (x_cnt_r-2) : max6_line_left;
                if(char_negedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max6_line_right <=
x_cnt_r-2;
                end
            end
            6'd13:begin
                max6_line_right <=
char_negedge ? (x_cnt_r-2) : max6_line_right
;
                if(char_posedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max7_line_left   <=
x_cnt_r-2;
                end
            end
            //数字5
            6'd14:begin
                max7_line_left   <=
char_posedge ? (x_cnt_r-2) : max7_line_left;
                if(char_negedge)begin
                    char_edge_cnt    <=
char_edge_cnt+1'b1;
                    max7_line_right <=
x_cnt_r-2;
                end
            end
            6'd15:begin
                max7_line_right <=
char_negedge ? (x_cnt_r-2) : max7_line_right
;
            end
            default: char_edge_cnt    <= 6'd0
;
```

```
            endcase
        end
        else begin
//          char_left_cnt     <= 4'd0;
//          char_right_cnt    <= 4'd0;
        char_edge_cnt    <= 6'd0 ;
        end
    end

    //场同步上升沿更新数据
    always @ (posedge clk or negedge rst_n)
    if(!rst_n) begin
        char1_line_left  <= 10'd0;
        char1_line_right <= 10'd0;
        char2_line_left  <= 10'd0;
        char2_line_right <= 10'd0;
        char3_line_left  <= 10'd0;
        char3_line_right <= 10'd0;
        char4_line_left  <= 10'd0;
        char4_line_right <= 10'd0;
        char5_line_left  <= 10'd0;
        char5_line_right <= 10'd0;
        char6_line_left  <= 10'd0;
        char6_line_right <= 10'd0;
        char7_line_left  <= 10'd0;
        char7_line_right <= 10'd0;
    end
    else if(vsync_pos_flag) begin
        char1_line_left  <= max1_line_left ;
        char1_line_right <= max1_line_right;
        char2_line_left  <= max2_line_left ;
        char2_line_right <= max2_line_right;
        char3_line_left  <= max3_line_left ;
        char3_line_right <= max3_line_right;
        char4_line_left  <= max4_line_left ;
        char4_line_right <= max4_line_right;
        char5_line_left  <= max5_line_left ;
        char5_line_right <= max5_line_right;
        char6_line_left  <= max6_line_left ;
        char6_line_right <= max6_line_right;
        char7_line_left  <= max7_line_left ;
        char7_line_right <= max7_line_right;
    end

    endmodule
```

超声波模块

```
    `timescale 1ns/1ps

    module vlg_ultrawave(
        input i_clk,
        input i_rst,
        input i_echo,
        output o_trig,
        output key,
        output [15:0] o_s_mm
        );
    localparam SYS_CLK_STEP = 20;
    wire w_clk_en;
    wire [15:0] w_t_us;
    wire [15:0] w_t_filt_us;
    wire w_echo_en;
    /使能时钟产生模块
    vlg_en #(
        .SYS_CLK_STEP(SYS_CLK_STEP)

        )
    uut_vlg_en(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .o_clk_en(w_clk_en)
        );


    /产生触发信号o_trig驱动超声波模块工作
    vlg_trig    uut_vlg_trig(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_clk_en(w_clk_en),
        .o_trig(o_trig)
        );


    /超声波模块回响信号i_echo引脚高电平时间测量
    vlg_echo    uut_vlg_echo(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_echo(i_echo),
        .i_clk_en(w_clk_en),
        .o_t_us(w_t_us)
        //.o_echo_en(w_echo_en)
        );


    /将回响信号高电平时间(us)通过公式转换成距离(
     mm)
    vlg_cal     uut_vlg_cal(
        .i_clk(i_clk),
        .i_rst(i_rst),
        .i_t_us(w_t_us),
        .o_s_mm(o_s_mm),
        .key(key)
        );
    endmodule
    /*
        模块功能：产生周期1us的使能脉冲
    */
    `timescale 1ns/1ps

    module vlg_en #(
```

```
    parameter SYS_CLK_STEP = 20  //系统时钟一
 周期20ns
    )
(
    input i_clk,
    input i_rst,
    output reg o_clk_en
    );

/1us计数器
localparam DIV_CNT_MAX = 1000/SYS_CLK_STEP;
reg [7:0] r_divcnt;

always @(posedge i_clk)
    if(i_rst)   r_divcnt <= 'b0;
    else
     begin
        if(r_divcnt < (DIV_CNT_MAX-1))
 r_divcnt <= r_divcnt + 1'b1;
        else    r_divcnt <= 'b0;
     end
/产生使能脉冲
always @(posedge i_clk)
    if(i_rst)   o_clk_en <= 1'b0;
    else if(r_divcnt == (DIV_CNT_MAX-1))
 o_clk_en <= 1'b1;
    else    o_clk_en <= 1'b0;
endmodule
/*
    模块功能：产生100ms为周期、高电平持续10us
 的脉冲
*/

'timescale 1ns/1ps

module vlg_trig (
    input i_clk,
    input i_rst,
    input i_clk_en,
    output reg o_trig
    );


localparam TRIG_CNT_MAX = 100_000;
localparam TRIG_CNT_10US = 10;
reg [31:0] r_trig_cnt;

/100ms计数
always @(posedge i_clk)
    if(i_rst)   r_trig_cnt <= 'b0;
    else if(i_clk_en)
     begin
        if(r_trig_cnt < (TRIG_CNT_MAX-1))
 r_trig_cnt <= r_trig_cnt + 1'b1;
        else    r_trig_cnt <= 'b0;
     end
    else ;
/产生10us的高脉冲
always @(posedge i_clk)
    if(i_rst)   o_trig <= 1'b0;
    else if((r_trig_cnt > 0)&&(r_trig_cnt <=
 TRIG_CNT_10US))    o_trig <= 1'b1;
    else    o_trig <= 1'b0;


endmodule
/*模块功能：测量echo引脚返回的高脉冲时间 */
'timescale 1ns/1ps

module vlg_echo (
    input i_clk,
    input i_rst,
    input i_echo,
    input i_clk_en,
    output reg [15:0] o_t_us,
    output reg o_echo_en
    );

reg [1:0] r_echo;
wire w_pos_echo,w_neg_echo;
reg r_cnt_en;
reg [15:0] r_cnt_echo;

/获取边沿检测信号
always @(posedge i_clk)
    if(i_rst)   r_echo <= 'b0;
    else    r_echo <= {r_echo[0],i_echo};

assign w_pos_echo = r_echo[0] & ~r_echo[1];
assign w_neg_echo = ~r_echo[0] & r_echo[1];

/产生计数使能信号
always @(posedge i_clk)
    if(i_rst)   r_cnt_en <= 1'b0;
    else if(w_pos_echo) r_cnt_en <= 1'b1;
    else if(w_neg_echo) r_cnt_en <= 1'b0;
    else;

/对使能信号高脉冲进行计数
always @(posedge i_clk)
    if(i_rst)   r_cnt_echo <= 'b0;
    else if(r_cnt_en)
     begin
        if(i_clk_en)    r_cnt_echo <=
 r_cnt_echo + 1'b1;
        else;
     end
    else    r_cnt_echo <= 'b0;


/对计数值锁存输出
always @(posedge i_clk)
```

```
    if(i_rst)   o_t_us <= 'b0;
    else if(w_neg_echo) o_t_us <= r_cnt_echo;


/使能信号有效输出
always @(posedge i_clk)
    if(i_rst)   o_echo_en <= 1'b0;
    else    o_echo_en <= w_neg_echo;
endmodule

/模块功能：乘法器/

`timescale 1ns/1ps

module vlg_cal (
    input i_clk,
    input i_rst,
    input  [15:0] i_t_us,
    output [15:0] o_s_mm,
    output reg key
    );

/s=0.173*t变成整数乘法的过程/
//  s*4096 = 0.173*4096*t
//  =>  s*4096 = 709*t
//  =>  s = 709*t / 4096
//  =>  s = 709*t >> 12

localparam  MUL_N = 10'd709;
reg [25:0] w_mult_result;

always @(posedge i_clk)
    if(i_rst)   w_mult_result <= 'b0;
    else    w_mult_result <= MUL_N * i_t_us;

/乘法器IP例化，实现709*t的功能
mult_gen_0 your_instance_name (
  .CLK(i_clk),      // input wire CLK
  .A(MUL_N),        // input wire [9 : 0] A
  .B(i_t_us),       // input wire [15 : 0] B
  .P(w_mult_result) // output wire [25 : 0] P
);

assign o_s_mm = {2'b00,w_mult_result[25:12]};

always @(*) begin
    if (o_s_mm>= 8'o310) begin
      key = 1'b0;
    end else begin
      key = 1'b1;
    end
  end

endmodule
```

舵机模块

```
`timescale 1ns / 1ps
module fpga_sevo
    (
        input               clk,
        input               rst_n,
        input               key_1,
        output reg          pwm
    );
    //参数
    parameter       TIME_20MS   = 100_0000;
    parameter       pwm_0d5ms   = 2_5000;
    parameter       pwm_1ms     = 5_0000;
    parameter       pwm_1d5ms   = 7_5000;
    parameter       pwm_2ms     = 10_0000;
    parameter       pwm_2d5ms   = 12_5000;

    //内部信号
    reg [16:0]          cnt_pwm;
    reg [19:0]          cnt;
    wire                add_cnt;
    wire                end_cnt;

    //功能

    //20ms计数
    always @(posedge clk or negedge rst_n)
begin
        if(!rst_n)begin
            cnt <= 0;
        end
        else if(add_cnt)begin
            if(end_cnt)
                cnt <= 0;
            else
                cnt <= cnt + 1;
        end
    end

    assign add_cnt = 1 ;
    assign end_cnt = add_cnt && cnt==
TIME_20MS - 1  ;

    //按键情况列出
    always  @(posedge clk or negedge rst_n)
begin
        if(rst_n==1'b0)begin

        end
        else if(key_1 == 1)
            cnt_pwm <= pwm_1d5ms;
        else
            cnt_pwm <= pwm_0d5ms;
    end
```

```
        //输出pwm
        always  @(posedge clk or negedge rst_n)
    begin
            if(rst_n==1'b0)begin
                pwm <= 0;
            end
            else if(cnt < cnt_pwm) begin
                pwm <= 1;
            end
            else
                pwm <= 0;
        end

        endmodule
输出图像尺寸和帧率配置模块
module picture_size (
    input               rst_n         ,
    input               clk           ,
    input       [15:0] ID_lcd         ,

    output      [12:0] cmos_h_pixel   ,
    output      [12:0] cmos_v_pixel   ,
    output      [12:0] total_h_pixel ,
    output      [12:0] total_v_pixel ,
    output      [23:0] sdram_max_addr
);

reg [12:0] cmos_h_pixel;
reg [12:0] cmos_v_pixel;
reg [12:0] total_h_pixel;
reg [12:0] total_v_pixel;
reg [23:0] sdram_max_addr;

//parameter define
parameter  ID_4342 =   16'h4342;
parameter  ID_7084 =   16'h7084;
parameter  ID_7016 =   16'h7016;
parameter  ID_1018 =   16'h1018;


//配置摄像头输出尺寸的大小
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        cmos_h_pixel <= 13'b0;
        cmos_v_pixel <= 13'd0;
        sdram_max_addr <= 23'd0;
    end
    else begin
        case(ID_lcd )
            16'h4342 : begin
                cmos_h_pixel   = 13'd480;
                cmos_v_pixel   = 13'd272;
                sdram_max_addr = 23'd130560;
            end
            16'h7084 : begin
                cmos_h_pixel   = 13'd800;
                cmos_v_pixel   = 13'd480;
                sdram_max_addr = 23'd384000;
            end
            16'h7016 : begin
                cmos_h_pixel   = 13'd1024;
                cmos_v_pixel   = 13'd600;
                sdram_max_addr = 23'd614400;
            end
            16'h1018 : begin
                cmos_h_pixel   = 13'd1280;
                cmos_v_pixel   = 13'd800;
                sdram_max_addr = 23'd1024000;
            end
        default : begin
                cmos_h_pixel   = 13'd800;
                cmos_v_pixel   = 13'd480;
                sdram_max_addr = 23'd384000;
        end
        endcase
    end
end

//对HTS及VTS的配置会影响摄像头输出图像的帧率
always @(*) begin
    case(ID_lcd)
        ID_4342 : begin
            total_h_pixel = 13'd1800;
            total_v_pixel = 13'd1000;
        end
        ID_7084 : begin
            total_h_pixel = 13'd1800;
            total_v_pixel = 13'd1000;
        end
        ID_7016 : begin
            total_h_pixel = 13'd2200;
            total_v_pixel = 13'd1000;
        end
        ID_1018 : begin
            total_h_pixel = 13'd2570;
            total_v_pixel = 13'd980;
        end
    default : begin
            total_h_pixel = 13'd1800;
            total_v_pixel = 13'd1000;
    end
    endcase
end

endmodule
```