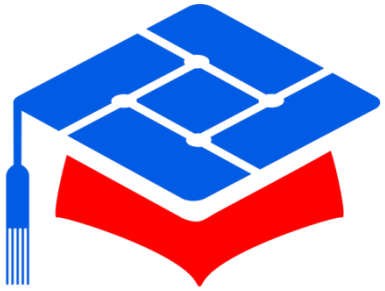

第六届

全国大学生集成电路创新创业大赛

报告类型*：设计方案

参赛杯赛*：紫光同创杯

作品名称*：基于 PGL22G 的 MobileNet-YOLO 水果识别系统

队伍编号*：CICC1084

团队名称*：点灯稳过

目录

1　项目背景

随着我国经济水平与消费水平的增长，人民对于美好生活的需要也日益增长，这其中就包括了对于各种水果的需求，因此也大大增加了物流分装过程中的水果筛选需求。但我国目前的水果筛选与分装仍停留在人工和半机械阶段，且易受到环境、视力、疲劳的影响，效率较低。因此，一个高识别率与高效率的水果识别系统是十分必要的。

而当今市场中的水果识别系统，大多是基于传统的 CV、ML 算法，通过识别色块、获取轮廓、分割前后景并对图片进行适当调整，从而提取图片特征，匹配与之最接近模型，识别目标水果。这种方法易受背景、水果图片特征混淆干扰，准确度低，不适用于大量水果的识别与商业化的推广。

基于这些问题，为了提高识别率、加快识别速率，本项目针对算法和 FPGA 进行创新，基于紫光同创 PGL22G 平台进行 CNN 水果识别算法的 FPGA 部署，并将其应用于适应复杂光线变化的物流分装中的水果筛选。

2　系统功能介绍

本水果识别系统实现了 1024*768p 识别视频输入和串口数据输出的全流程，包括水果数量识别、水果颜色识别、水果类别识别三个功能：

在水果数量识别方面，系统能够识别 10 个以下水果的数量，并通过串口输出识别结果数字。

在水果颜色识别方面，系统能够识别水果的颜色，并根据结果指定串口输出 RED、ORANGE、YELLOW、GREEN、PURPLE、WHITE、BROWN、MAROON 九种颜色单词。

在水果类别识别方面，系统能够识别苹果、香蕉、葡萄、火龙果、梨子、芒果、猕猴桃、橙子、山竹、无花果十种水果，并通过串口输出识别结果 APPLE、BANANA、GRAPE、PITAYA、PEAR、MANGO、KIWI、ORANGE、STEEN、FIG 十种水果类别单词。

接下来举例系统对于葡萄的识别过程：

1. 将葡萄放置在开发板摄像头前；

2. 打开设备电源，连接串口，上位机将收到空数据；

3. 首先按下板载的 KEY4 按钮，对应 LED4 亮起，权重逐次从 SD 卡加载到设备，识别开始；
4. 按下板载的 KEY2 或 KEY3 按钮可切换识别模式，分别有颜色识别模式、水果种类识别模式和水果数量识别模式；
5. 模型训练时将一串葡萄的水果数量设置为 1，因此得到结果：颜色为紫色（purple）、种类为葡萄（grape）、数量为 1；

具体操作可参考附件中的演示视频

3 系统架构

本项目基于紫光同创 PGL22G 开发板设计，全部使用开发板板载资源实现。图像数据通过 OV5640 摄像头输入，第一帧图片优先缓存到片上资源实现的 BRAM 中，超过缓存容量的图片数据会经过 AXI-MIG 转移到片外 DDR3 等待读取。图像数据同时输入到 CNN 加速电路和 KNN 实时颜色识别电路，CNN 加速电路采用宽位并行输入以适配后续的流水线;KNN 加速电路以 ISP 形式实现,基于摄像头的 PCLK 时序实时处理数据。一套按钮状态机处理板上控制按钮按下情况并对数据流进行分配，输出模式也由该状态机指定。KNN 电路输出颜色识别码到串口模块，串口模块会对颜色识别码进行译码处理并输出具体的颜色；CNN 电路输出种类识别码到串口模块，同时输出识别框坐标序列到计数器模块，串口模块直接译码出种类发送到上位机，计数器模块记录同一批次输入图像的识别状态，并对识别框序列进行计数，从而得到每帧图像上的水果数量。系统还会通过一个数据选择器输出摄像头的实时图像或经过 CNN 电路检测目标后的图像数据，由一个 HDMI 控制器进行输出。系统架构框图如图 3.1 所示。

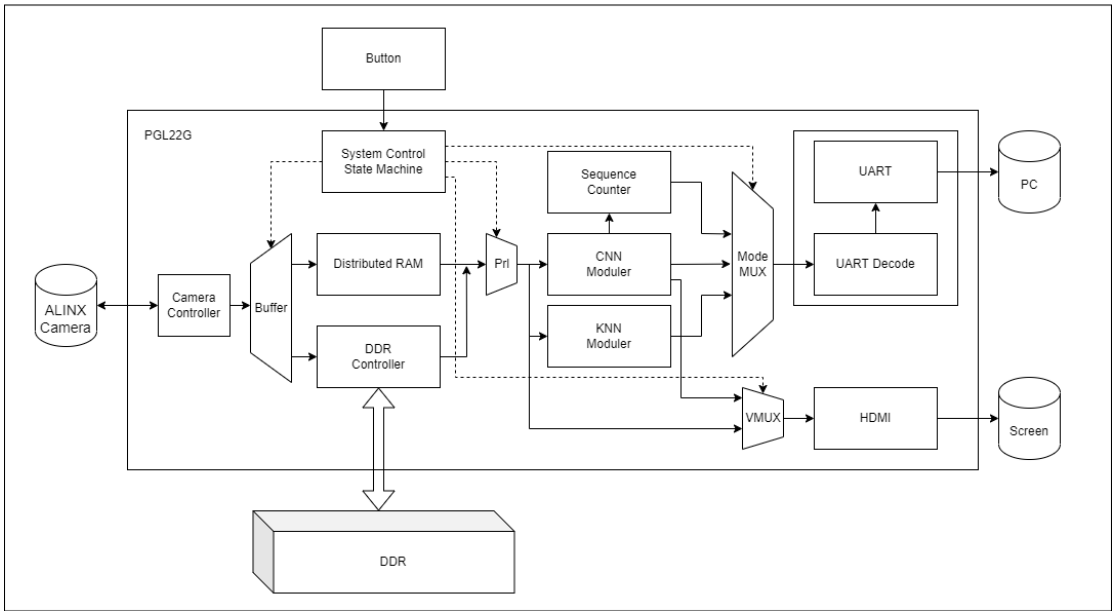

图 3.1

3.1 IO 设备设计

设备搭载了摄像头模块、MIG-AXI 转换模块、SD 卡控制模块、按键处理模块、HDMI 输出模块、串口输出模块用于数据的输入输出。

摄像头模块采用典型的 DVP 总线驱动电路和 SCCB 总线驱动电路，摄像头数据输入后会通过一个摄像头控制器异步 FIFO（DCMI_FIFO）与主体程序耦合，同时避免了跨时钟域数据传输问题。

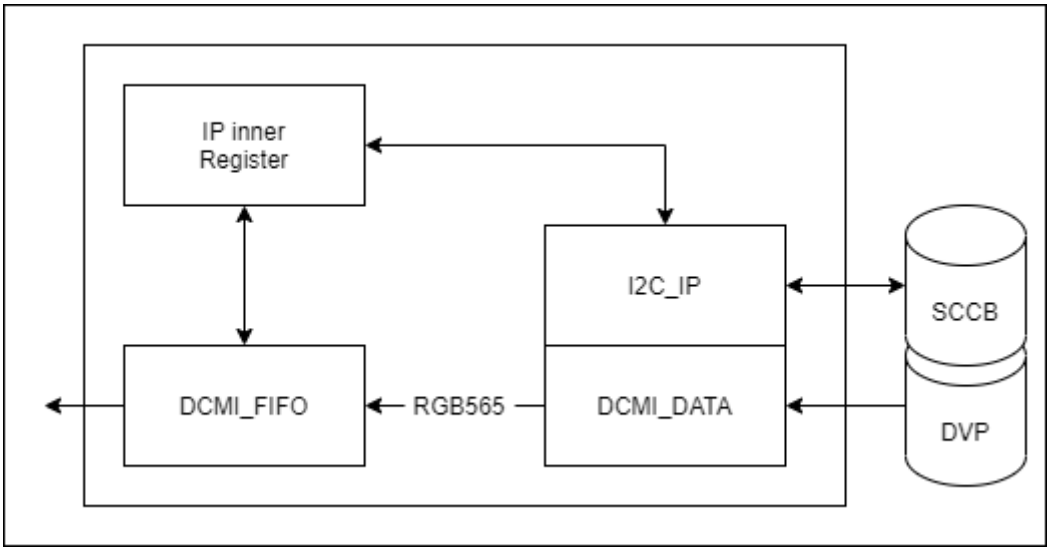

图 3.1.1

MIG-AXI 转换模块使用紫光同创提供的 MIG IP 实现，接口时序与官方数据手册一致。AXI 模块实现了标准 AXI 主设备的输入输出逻辑，包括普通 IO 和突发 IO 功能，并配有简化的的 AXI 从设备-简单握手协议总线桥。MIG 连接到 AXI 主设备上，AXI 主设备和总线桥直接连接，总线桥直接连接到片上缓存和摄像头控制器 IP。

按键处理模块主要由按键消抖子模块和按键状态机子模块组成，按下开发板上按钮后，按键消抖子模块滤除按键抖动，输出一个高电平脉冲，按键状态机采集脉冲并调整当前系统运行模式。设备内部所有数据流多路选择器完全由这个状态机控制。除了既定的三种识别模式外，还设置了低功耗运行的无输出模式，该模式下 CNN、KNN 模块都不会启动，系统只会采集摄像头数据并通过 HDMI 接口输出。

HDMI 输出模块负责将原始视频数据或来自 CNN 模块的处理后数据转换成 HDMI 时序，采用来自黑金 ALINX 的 HDMI IP 实现。

SD 卡控制模块用于读取保存在 SD 卡上的模型权重信息。进入识别模式后，CNN 模块中每一层算子电路会在需要工作时读取 SD 上的权重信息，根据对应权重进行运算。SD 卡控制模块包含 SDIO 子模块、SD 卡读写子模块、权重识别子模块、识别码解析子模块。CNN 模块中每一层都被分配了一个 8 位的识别码，识别码解析子模块会对发出申请的算子识别码译码，权重识别子模块根据译码结果找到对应权重文件，读写子模块读取相应文件，加载到算子后再向算子发送允许信号，算子即可进行计算。网络权重文件从 ONNX 文件转换成二进制格式，规定文件头为算子识别码-文件长度-奇偶校验码-结束位，权重识别子模块会扫描 SD 卡数据，如果在文件头的位置检测到对应的识别码，会先检测文件长度和结束位是否对应，再检查奇偶校验码，通过后即将对应权重数据加载到网络。由于网络结构中的 BN 层已经被合并到卷积层，所以网络权重相对规整，SD 卡读写造成的性能损失较小。

串口输出模块由串口时序发生模块、输出数据选择器、串口译码模块组成。串口译码模块根据当前系统工作模式，接收来自序列计数器、CNN 模块、KNN 模块的水果种类、颜色、数量识别码，译码成可读的 ASCII 码后通过串口时序发送到上位机。输出数据选择器根据按钮状态机指示的输出状态选择输出水果种类、

颜色及数量。

3.2 CNN 加速电路接口时序

CNN 加速电路模块和外围模块连接中采用简单快速的 valid-ready 握手协议，内部采用易于实现的高电平触发信号协议，可以高速实现对 INT8 整型的卷积和比较运算。

CNN 模块内部，每层算子完成运算后都会发送一个时钟周期的高电平到下一层算子，下一层算子在接收到高电平后会从上一层算子中接收输出的数据，等待来自 SD 卡的权重输入，直到权重输入完成后再运算，上一层的输出数据在此期间将保持直到下一层的忙信号拉高。大部分算子都属于 CBL 算子，包含实现卷积运算的时序电路和实现 ReLU 运算的组合逻辑电路（BN 层已经在训练后与卷积层合并，无需在推理时运算），因此延迟最低可以达到一个机器周期（即一层卷积电路对一副图片进行运算的时间）。在时序通路中稍加优化即可达到较好的时序效果，同时能够实现流水线操作。

卷积层需要从片外 SD 卡读取权重。当收到上一层电路发送的高电平触发时，首先发送请求信号到 SD 卡模块，等待权重加载。SD 卡模块会成批加载权重到 CNN 电路以提高速率。权重加载完毕后，该层向上一层发送忙信号。

CNN 加速电路内部接口中使用到的端口如表 3.2.1 所示

表 3.2.1 CNN 模块内部接口时序表

端口名	功能
work_en 计算使能	接收上一层的计算完成触发信号
work_fin 计算完成	发送到下一层的计算完成触发信号
busy 忙信号	标志本层正在进行运算的触发信号

CNN 模块与外围模块使用宽位并行接口连接，使用简单的 valid-ready 握手协议进行通信。从并行数据转换器输出的图像信号，首先转换为 RGB565 格式，按行同步输入 CNN 模块的前处理子模块，完成图像缩放，变换成 YOLOv4 能够处

理的 416*416 像素 RGB888 数据。完成转换后，前处理子模块向 CNN 控制器子模块发送 ready 信号，CNN 控制器子模块根据前一帧图像的运算情况选择抛弃下一帧图像或将下一帧图像输入 CNN 加速器，若决定输入图像则会输出 valid 信号，并完成数据读取。

外围模块之间的数据通路也采用 valid-ready 握手协议，当模块空闲时，会向要读取数据的模块发送 ready 信号，若此时数据发送模块 valid 信号拉高，表示数据有效，后一模块将数据读取到内部进行处理。

表 3.2.2　CNN 模块外部接口时序表

端口名	功能
valid 允许输出信号	拉高代表数据可信，能够读取
ready 准备完成信号	拉高代表可以输入数据

3.3　KNN 实时颜色识别电路

KNN 电路内部中使用了 valid-ready 时序，外部使用 VGA 时序。在 KNN 模块信号输入端，前处理模块会通过 VGA 时序读取来自并行数据转换器的图像信号，完成 RGB-HSV 转换和直方图统计，输出数据通过上述 valid-ready 协议传输到 KNN 电路。

详细内容将在 KNN 子模块中介绍。

3.4　图像缓存

摄像头数据输入后会根据当前运行模式和数据处理情况被缓存到片上 DRAM 或片外DDR,该过程由按键状态机输出的模式控制码和FIFO状态控制,如表3.4.1所示。

表 3.4.1 图像缓存选择

系统工作模式	FIFO 状态	缓存目标
任意	满	DDR

水果颜色识别	空	DRAM
水果种类识别、数量识别	空	DDR

4 算法设计

团队对现有的三种主流目标识别算法进行分析，发现传统上基于图像的机器视觉算法存在准确度低、难以适应复杂光照环境的缺点；基于最新研究的脉冲神经网络（SNN）或视觉 Transformer（ViT）的神经网络算法存在实现难度过高、片上资源消耗巨大的缺点；而近些年来占据视觉神经网络结构主流的卷积神经网络很好地平衡了二者。三者的特点列举如下：

算法	主要优点	主要缺点
传统 CV 算法	实现简单，易于部署	抗干扰能力差
卷积神经网络	泛化能力强，可适应更多环境	片上资源消耗大，实现难度大，需要额外采集数据集
ViT、SNN 等	准确度高、前景广阔	片上资源消耗大，实现难度极高，需要额外采集数据集

综上，项目选择在 FPGA 上移植适合的卷积神经网络模型作为主体算法；但考虑到神经网络算法的训练过程中需要使用大量人工标注的数据集，如果将颜色识别功能一并作为神经网络的分类输出，无疑会导致数据集需求量倍增，因此团队将颜色识别功能独立出来，在主神经网络电路旁挂一个 K 最近邻网络（KNN）电路，基于直方图统计和 KNN 算法进行颜色识别，使用传统的机器学习算法来减小工作量，节省片上资源。

4.1 改进的 YOLO-MobileNet 算法

项目设计过程中，团队首先基于 PyTorch，使用 Python 高级语言在已有实现上针对 FPGA 环境和项目应用改进了 YOLOv3 算法，主要改进点包括：

1. 将 Backbone 部分从 ResNet-53 替换为 MobileNetv2，大幅减少资源消耗；
2. 将 YOLOv4 的 YoloBody 部分移植到 YOLOv3，并将主体中所有使用了传统 3x3 卷积核的 CBL 块改为使用 1x1 卷积核的 inverse CBL 块；

3. 将每个特征图输出的五层卷积都改为四层，减少一定资源消耗；
4. 将上采样阶段多余卷积层删除，只留下基本的上采样层；
5. 借鉴 YOLOv4 的 masaic 数据增强、KNN 预规划选择框等技巧，并使用基于低阶矩的自适应估计的 Adam 算法进行训练。

经过上述改进，本项目中的算法（称为 Yolo-MobileNet）最终在 PC 测试平台（Nvidia RTX 2060）上达到约 27 帧的识别速率；官方的 YOLOv3-tiny 实现能够达到相同速率，但准确度（AP）大不如本模型；YOLOv4-tiny 实现的准确度与本模型相差约 10%；YOLOv4-n 实现则只能达到约 20 帧的速率。在初始 float32 权重下，模型所需的网络总大小减少到 7710.43MB，相比原版 YOLOv4-tiny 减少了 2000MB 左右。网络权重大小增加了约 20MB，但所需的网络层数大大减少，有利于 FPGA 端部署。

```
          Conv2d-315          [-1, 255, 1
================================================
Total params: 10,801,149
Trainable params: 10,801,149
Non-trainable params: 0
------------------------------------------------
Input size (MB): 1.98
Forward/backward pass size (MB): 7667.25
Params size (MB): 41.20
Estimated Total Size (MB): 7710.43
------------------------------------------------
```

图 4.1.1　改进版 YOLOv3

```
          Conv2d-87           [-1, 255, 2
================================================
Total params: 6,056,606
Trainable params: 6,056,606
Non-trainable params: 0
------------------------------------------------
Input size (MB): 1.98
Forward/backward pass size (MB): 9962.41
Params size (MB): 23.10
Estimated Total Size (MB): 9987.49
------------------------------------------------
```

图 4.1.2　原版 YOLOv4-tiny

```
        Conv2d-433          [-1, 255, 13, 13]
================================================
Total params: 64,363,101
Trainable params: 64,363,101
Non-trainable params: 0
------------------------------------------------
Input size (MB): 1.98
Forward/backward pass size (MB): 1670.17
Params size (MB): 245.53
Estimated Total Size (MB): 1917.67
------------------------------------------------
```

图 4.1.3　原版 YOLOv4-n

项目算法的整体优化思路即舍弃部分实时性和参数大小，换取更高的算法效率，从而让 FPGA 上的电路实现更节省片上资源，达到相对更高的硬件效率；同时尽量避免使用不同大小的卷积层，从而让硬件电路可以在相对简单的时序条件下形成全流水，加快图像处理速度。

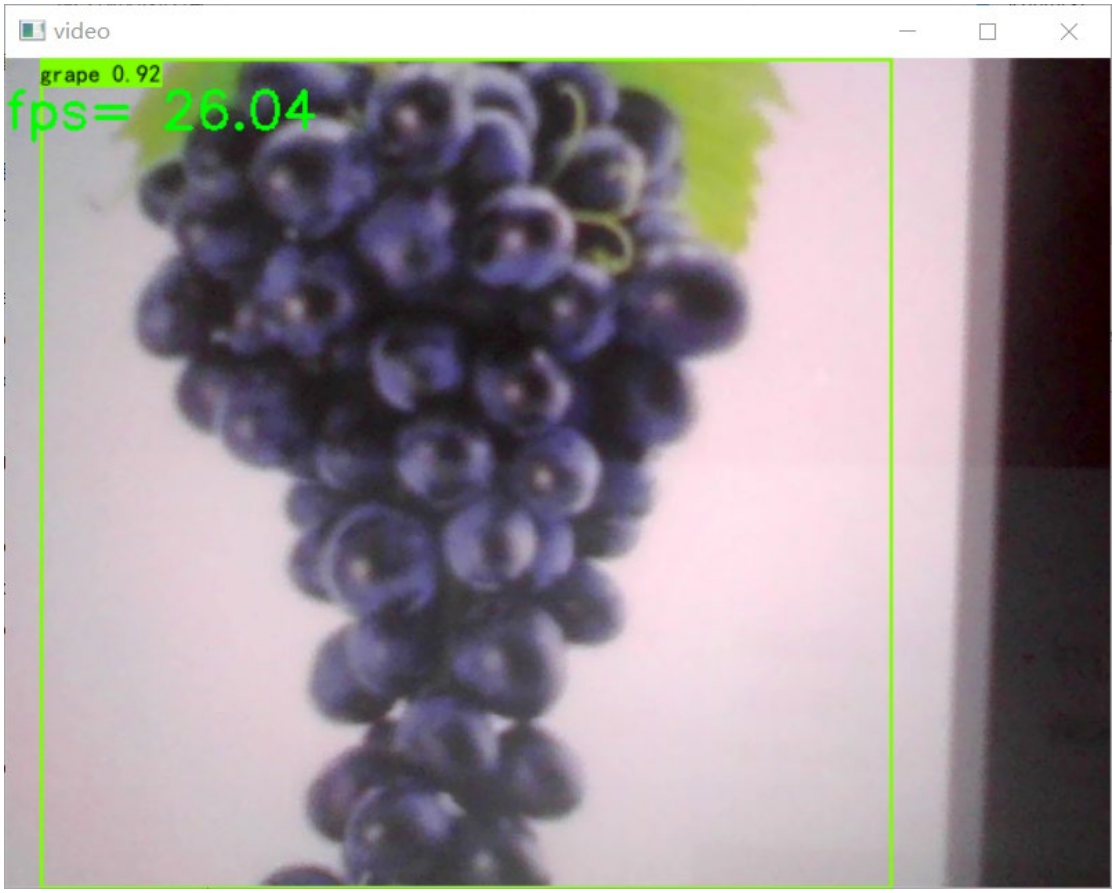

图 4.1.4　算法实测效果（摄像头前放置水果图片进行测试）

模型训练后得到的参数经过 ONNX INT8 量化，转换为便于 FPGA 运算的 8 位

整型，权重通过二进制文件的形式保存到 SD 卡上，FPGA 实时读取加载 SD 卡上的权重，根据运算流程，分配到各层算法电路。图片在处理过程中先被分散为数帧，多个通道的卷积核会对其并行处理。量化过程中也实现了卷积层和 BN 层合并，进一步简化了网络实现。

4.2 基于 KNN 的水果颜色识别算法

项目实现了一套 KNN 水果颜色识别算法并部署在 FPGA 端。算法参数由上位机预训练提供，由于权重较小，直接将其转换成 coe 格式存储在 FPGA 的片上 ROM 资源，系统启动后自动加载。

整体算法数据流如图 4.2.1 所示，摄像头数据会被优先存入 DRM，如果存放不及时，数据会被缓存到外部 DDR 中。片上存储的数据会优先转换成并行模式分发给预处理模块和 KNN 模块，由一个按钮控制状态机负责系统识别状态切换。

进入颜色识别模式下，KNN 模块的参数被加载，首先进行 RGB 到 HSV 颜色空间的转换，并对图片像素信息逐行进行直方图统计，直方图统计结果经 KNN 电路运算后可以输出最接近的颜色。

在水果种类识别和数量识别模式下，主要任务由 YOLO 神经网络模块完成，图像帧会受输出情况控制，从而实现算法流水处理，YOLO 神经网络输出的数据分为水果种类信息和水果数量信息，种类信息会被直接输出到串口和 HDMI 接口，数量信息由 YOLO 输出预测框的序列组成，会被送入一个计数模块，对 YOLO 输出预测框的数量计数后得到当前帧图片内的水果数量。

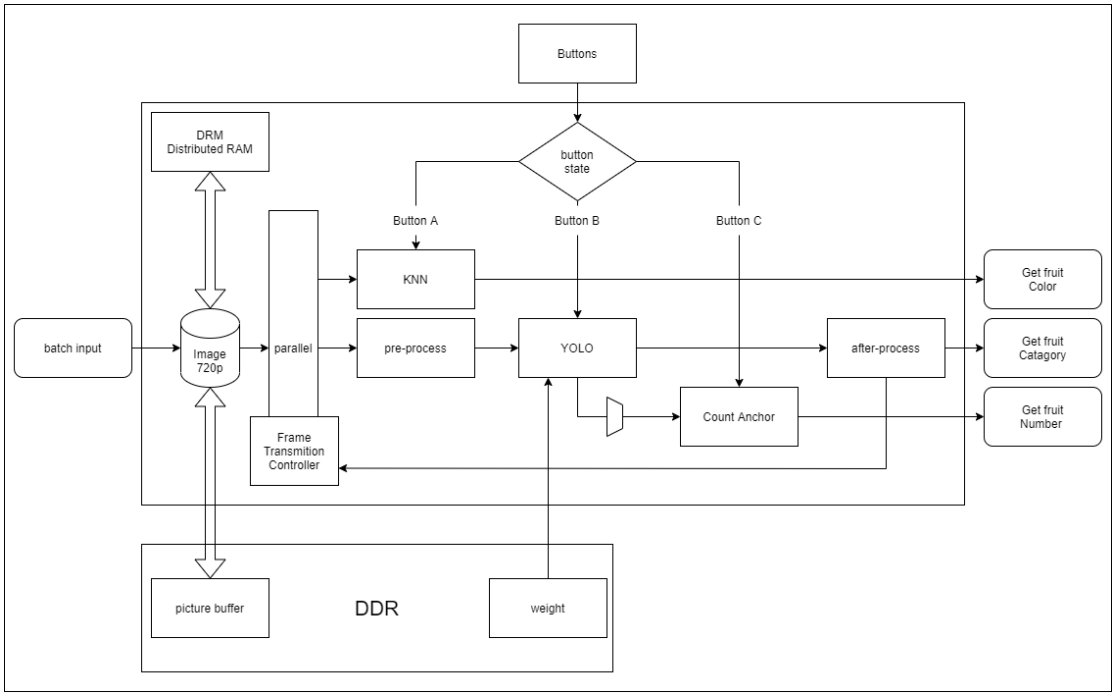

图 4.2.1　算法数据流图

```python
def responseOfNeighbors(neighbors):
    """最近邻回归"""
    all_possible_neighbors = {}

    for x in range(len(neighbors)):
        response = neighbors[x][-1]
        if response in all_possible_neighbors:
            all_possible_neighbors[response] += 1
        else:
            all_possible_neighbors[response] = 1

    sortedVotes = sorted(all_possible_neighbors.items(),
                         key=operator.itemgetter(1), reverse=True)

    return sortedVotes[0][0]

```

图 4.2.2　关键代码

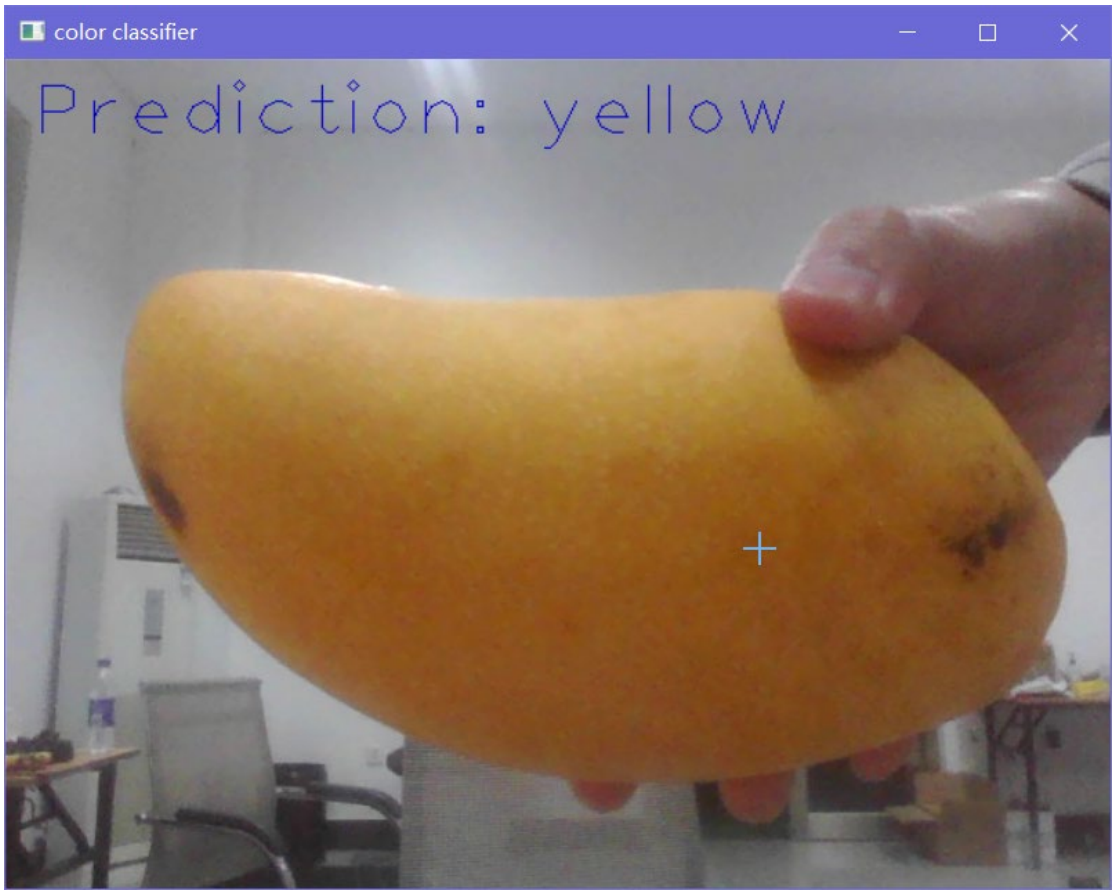

图 4.2.3　算法测试效果

5　子模块设计

5.1　硬件加速模块

5.1.1　CBL 结构

CBL 结构即卷积运算与 ReLU 非线性运算的结合，在完成卷积运算后对所得数据进行 ReLU 运算，即实现分段函数：$y=\begin{cases}x, x>0\\0, x\le 0\end{cases}$ 最后输出结果。

此结构由四个模块组成，其分别是:conv_top、conv_buffer、conv_search、conv_cal 组成，同时还例化了 pds 提供的 IP 核 Multiply-Accumulator 作为卷积计算核心乘加器。conv_top 作为 CBL 结构的顶层模块，例化 conv_buffer、conv_search、conv_cal 三个子模块，并完成三个模块间的信号调度。

conv_buffer 模块是卷积层的储存缓冲区。conv_search 模块是 conv_buffer 储

存缓冲区数据的检索模块。conv_cal 模块是卷积运算与 Relu 运算的计算模块。接下来将对此四个模块具体功能进行描述。

conv_top 模块主要完成三个子模块的互联并进行信号调度，其输入输出端口如图 5.1.1.1 所示，img 为待处理数据，weight 则是权重数据，bias 是偏置数据，conv_en 是卷积层接收上一级工作完成后发来的工作使能信号，conv_fin 是卷积层在结束工作时给输出给下一级的工作使能信号。

```
module conv_top #(
    parameter weight_width = 2,
    parameter weight_height = 2,

    parameter img_width = 4,
    parameter img_height = 4,

    parameter padding_enable = 0,
    parameter padding = 0,

    parameter stride = 2,
    parameter bitwidth = 3,
    parameter result_width = (img_width-weight_width+2*padding)/stride+1,
    parameter result_height = (img_height-weight_height+2*padding)/stride+1,
    parameter expand = 1,        //expand the bitwidth of result
    parameter threshold = 0

)(
    input clk_en,
    input rst_n,
    input conv_en,              //if 1,the conv is on

    input [img_width*img_height*bitwidth-1:0]  img,
    input [weight_width*weight_height*bitwidth-1:0] weight,
    input signed [bitwidth-1:0] bias,

    output [expand*2*result_width*result_height*bitwidth-1:0]  result,
    output conv_fin //if 1, the result of the conv is correct
);
```

图 5.1.1.1

conv_buffer 模块的主要功能完成对权重数据的静态缓存以及对待卷积数据的动态缓存，其输入输出端口如图 5.1.1.2 所示，anchor_l、anchor_c 是定位坐标，用于定位待卷积数据的输入起始位置;buf_l、buf_c 是缓冲区数据检索地址;img_cal、wei_cal 则是输出卷积计算所需的数据。

待卷积数据的动态缓冲区的构建如图 5.1.1.3 所示，权重数据的静态缓冲区的构建如图 5.1.1.4 所示。设计中搭建了数据选择器电路来完成数据的更新与定向提取，动态缓冲区的设计则可以有效的减少片内占用资源，该缓冲区数据的定

位则依赖于 anchor_l、anchor_c 两组输入信号。

```
module conv_buffer #(
    parameter weight_width = 2,
    parameter weight_height = 2,

    parameter img_width = 4,
    parameter img_height = 4,

    parameter padding_enable = 0,
    parameter padding = 0,

    parameter stride = 1,
    parameter bitwidth = 3,
    parameter result_width = (img_width-weight_width+2*padding)/stride+1,
    parameter result_height = (img_height-weight_height+2*padding)/stride+1,
    parameter expand = 1
) (
    input clk_en,
    input rst_n,
    input conv_on,

    input [31:0] anchor_l,
    input [31:0] anchor_c,

    input [3:0] buf_l,
    input [3:0] buf_c,

    input [img_width*img_height*bitwidth-1:0]  img,
    input [weight_width*weight_height*bitwidth-1:0] weight,

    output signed [bitwidth-1:0] img_cal,
    output signed [bitwidth-1:0] wei_cal
);
```

图 5.1.1.2

```
reg [bitwidth-1:0] img_buffer [0:weight_width-1][0:weight_height];
reg [9:0] i,j;
always@(posedge clk_en)begin
    if(!rst_n)begin
        for(i=0;i<weight_height;i=i+1)begin
            for(j=0;j<weight_width;j=j+1)begin
                img_buffer[i][j]=0;
            end
        end
    end
    else begin
        if(conv_on)begin
            for(i=0;i<weight_height;i=i+1)begin
                for(j=0;j<weight_width;j=j+1)begin
                    if (padding_enable) begin
                        if((anchor_l+i)<padding|(anchor_l+i)>img_height|(anchor_c+j)<padding|(anchor_c+j)>img_width)begin
                            img_buffer[i][j]=0;
                        end
                        else begin
                        img_buffer[i][j]=img[((anchor_l+i-padding)*img_width+(anchor_c-padding+j))*bitwidth +:bitwidth];
                        end
                    end
                    else begin
                        img_buffer[i][j]=img[((anchor_l+i)*img_width+(anchor_c+j))*bitwidth +:bitwidth];
                    end
                end
            end
        end
        else begin
            for(i=0;i<weight_height;i=i+1)begin
                for(j=0;j<weight_width;j=j+1)begin
                    img_buffer[i][j]=0;
                end
            end
        end
    end
end
```

图 5.1.1.3

```
reg [bitwidth-1:0] weight_buffer [0:weight_width-1][0:weight_height];
reg [9:0] m,n;
always@(*)begin
    if(!rst_n)begin
        for(m=0;m<weight_height;m=m+1)begin
            for(n=0;n<weight_width;n=n+1)begin
                weight_buffer[m][n]=0;
            end
        end
    end
    else begin
        if(conv_on)begin
            for(m=0;m<weight_height;m=m+1)begin
                for(n=0;n<weight_width;n=n+1)begin
                    weight_buffer[m][n]=weight[(m*weight_width+n)*bitwidth+:bitwidth];
                end
            end
        end
        else begin
            for(m=0;m<weight_height;m=m+1)begin
                for(n=0;n<weight_width;n=n+1)begin
                    weight_buffer[m][n]=0;
                end
            end
        end
    end
end
```

图 5.1.1.4

conv_search 模块是完成对所需数据的检索，并输出相应的检索坐标，其输入输出端口如图 5.1.1.5 所示，此模块的输出信号基本都是 conv_buffer 模块的输入信号，用于完成所需数据的定位、检索与提取。

本模块主要搭建了一系列的加法时序电路来更新动态缓冲区输入数据的起始坐标更新，完成对缓冲区中存有的待卷积数据以及权重数据的遍历，更新结果缓冲区的储存坐标。为了让各数据的更新有序进行并且提高运行效率，在此模块中设计了三级流水时序，此流水时序将在一个缓冲区遍历完成前三个时钟周期触发，流水顺序分别是定位坐标的更改-缓冲区的更新以及遍历坐标的更新-结果缓冲区储存坐标的更新。如图 5.1.1.6 展示缓冲区数据的遍历代码，其它坐标的更新大同小异。

```
module conv_search #(
    parameter weight_width = 2,
    parameter weight_height = 2,

    parameter img_width = 4,
    parameter img_height = 4,

    parameter padding_enable = 0,
    parameter padding = 0,

    parameter stride = 1,
    parameter bitwidth = 3,
    parameter result_width = (img_width-weight_width+2*padding)/stride+1,
    parameter result_height = (img_height-weight_height+2*padding)/stride+1,
    parameter expand = 1
) (
    input clk_en,
    input rst_n,
    input conv_on,

    output [9:0] anchor_l,
    output [9:0] anchor_c,

    output [9:0]  buf_l,
    output [9:0]  buf_c,

    output [9:0] rlt_l,
    output [9:0] rlt_c,

    output chge_rlt_o,
    output chge_rlt_q_o,
    output srh_fin
);
```

图 5.1.1.5

```
//search the addr of the buff
reg [9:0] rbuf_l;
reg [9:0] rbuf_c;
wire chge_buf_l;
assign buf_l=rbuf_l;
assign buf_c=rbuf_c;
assign chge_buf_l=(rbuf_c==weight_width-1)?1:0;
assign chge_buf=(rbuf_l==weight_height-1)&(rbuf_c==weight_width-2)?1:0;
always @(posedge clk_en) begin
    if(!rst_n)begin
        rbuf_l<=0;
        rbuf_c<=0;
    end
    else begin
        if(conv_on_q)begin
            if(keep_srh)begin
                rbuf_l<=rbuf_l;
                rbuf_c<=rbuf_c;
            end
            else begin
                if(~chge_srh&chge_srh_q)begin
                    rbuf_l<=0;
                    rbuf_c<=0;
                end
                else if(chge_srh&~chge_srh_q)begin
                    rbuf_l<=rbuf_l;
                    rbuf_c<=rbuf_c;
                end
                else begin
                    if(chge_buf_l)begin
                        rbuf_l<=rbuf_l+1;
                        rbuf_c<=0;
                    end
                    else begin
                        rbuf_l<=rbuf_l;
                        rbuf_c<=rbuf_c+1;
                    end
                end
            end
        end
        else begin
            rbuf_l<=0;
            rbuf_c<=0;
        end
    end
end
```

图 5.1.1.6

conv_cal 模块的功能是对从缓冲区提取出的待卷积数据以及权重数据进行卷积运算，并且加上偏置后运算结果再进行 relu 运算，将最终结果输入到结果缓冲区中，完成结果的串转并行输出。其输入与输出端口如图 5.1.1.7 所示，result 即是最终结果的输出端口。

卷积的计算使用了 IP 核 Multiply-Accumulator，提高了卷积运算在 PGL22G 芯片上的适应性。当一组卷积完成之后，Multiply-Accumulator 中累加的值将会输出到结果缓冲区中。结果缓冲区数据会在卷积结束之后将数据转为并行数据，在添加偏置后通过 result 输出口输出。IP 核的例化如图 5.1.1.8，偏置的添加、relu 运算与输出如图 5.1.1.9，其中 rdata 即是结果缓冲区。

```
module conv_cal #(
    parameter weight_width = 2,
    parameter weight_height = 2,

    parameter img_width = 4,
    parameter img_height = 4,

    parameter padding_enable = 0,
    parameter padding = 0,

    parameter stride = 1,
    parameter bitwidth = 3,
    parameter result_width = (img_width-weight_width+2*padding)/stride+1,
    parameter result_height = (img_height-weight_height+2*padding)/stride+1,
    parameter expand = 1,     //expand the bitwidth of result
    parameter threshold = 0
) (
    input clk_en,
    input rst_n,
    input conv_on,
    input srh_fin,

    input chge_rlt,
    input chge_rlt_q,

    input [3:0] rlt_l,
    input [3:0] rlt_c,

    input signed [bitwidth-1:0] bias,
    input signed [bitwidth-1:0] img_cal,
    input signed [bitwidth-1:0] wei_cal,

    output [expand*2*result_width*result_height*bitwidth-1:0] result
);
```

图 5.1.1.7

```
//the signal to reload the data_o to zero
wire reload;
assign reload = chge_rlt|srh_fin;
mult_acc mult_acc_inst (
  .a        (img_cal),                    // input [2:0]
  .b        (wei_cal),                    // input [2:0]
  .clk      (clk_en),                  // input
  .rst      (~rst_n),                  // input
  .ce       (conv_on),                    // input
  .acc_addsub   (0),    // input
  .reload   (reload),           // input
  .p        (data_link)                       // output [23:0]
);
```

图 5.1.1.8

```
//switch parallel to serial
generate
    genvar j;
    for(j=0;j<result_width*result_height;j=j+1)begin
        assign result[j*(expand*2*bitwidth)+:(expand*2*bitwidth)]=(rdata[j]+bias)>threshold?(rdata[j]+bias):threshold;
    end
endgenerate
```

图 5.1.1.9

卷积层仿真结果如图 5.1.1.10 所示，设定步长为 1，无 padding,偏置 bias 为-5，其中:img 待卷积数据矩阵为:

$$\begin{bmatrix} 3 & -2 & 4 & 1 \\ 2 & 0 & 6 & 2 \\ 6 & 7 & 1 & 2 \\ 5 & 6 & 4 & 2 \end{bmatrix}$$

weight 权重数据矩阵为:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

预计结果矩阵为：

$$\begin{bmatrix} 0 & 0 & 1 \\ 4 & 0 & 3 \\ 7 & 6 & 0 \end{bmatrix}$$

经对比仿真数据无误：

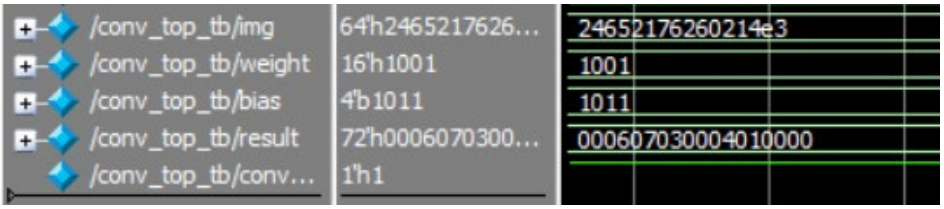

图 5.1.1.10

5.1.2 线性层

本层的作用是完成神经网络中的线性运算。

线性层只由一个 linear 模块构成,输入输出端口如图 5.1.2.1,in_features 是输入数据,bias 为偏置数据,outfeatures 为输出数据，flag 是线性层接收上一级工作完成后发来的工作使能信号，en_out 是线性层在结束工作时给输出给下一级的工作使能信号。其中例化了一个 IP 核 Simple Multiplier 作为乘法计算核心。为了提高运行效率，本模块中设计了两层流水电路，其流水顺序为输入数据和权重相乘-中间结果累加得到最终结果。如图 5.1.2.2 展示了累加电路设计。

```
module linear#(
    parameter DATABITWIDTH = 8,
    parameter DATALENGTH = 28 * 28,
    parameter OUTFEATURES = 20,
    parameter BIASBITWIDTH = 8
)(
    input clk,
    input rstn,
    input flag, //串口中断标志位
    input signed [DATABITWIDTH - 1 : 0] in_features,
    input signed [BIASBITWIDTH * OUTFEATURES -1 : 0] bias,
    output reg signed [OUTFEATURES * DATABITWIDTH * 2 - 1 : 0] outfeatures,
    output reg en_out
);
```

图 5.1.2.1

```
//每个时钟上升沿加一组数据
integer n;
always @(posedge clk) begin
    for(n = 0; n < OUTFEATURES; n = n + 1) begin
        if(n == (OUTFEATURES - 1))
            resultArray[n] <= (resultArray[n] + intermediateArray[n] + biasArray[n]);
        else
            resultArray[n] <= (resultArray[n] + intermediateArray[n]);
    end
end
```

图 5.1.2.2

5.1.3 上采样层

本层的作用主要是完成数据的上采样中的最近邻功能完成对数据的处理。

上采样层由 u_sampling_top 与 u_sampling 两个模块组成，u_sampling_top 作为 u_sampling 的顶层模块完成各种信号的连接，u_sampling 则进行上采样处理。

u_sampling_top 模块的代码如图 5.1.3.1 所示，主要功能是例化 u_sampling 模块与完成信号的连接。

```
module u_sampling_top #(
    parameter data_i_width = 2,                 //输入数据矩阵长
    parameter data_i_height = 2,                //输入数据矩阵宽

    parameter scale_factor = 2,                 //最近邻采样大小倍数

    parameter data_o_width = data_i_width*scale_factor,     //输出数据矩阵长
    parameter data_o_height = data_i_height*scale_factor,   //输出数据矩阵宽
    parameter bitwidth = 3              //位宽
) (
    input [data_i_width*data_i_height*bitwidth-1:0] data_i,
    output [data_o_width*data_o_height*bitwidth-1:0] data_o
);

u_sampling #(
    data_i_width,
    data_i_height,
    scale_factor,
    data_o_width,
    data_o_height,
    bitwidth
)
u_sampling_inst(
    .data_i         (data_i),
    .data_o         (data_o)
);

endmodule
```

图 5.1.3.1

u_sampling 模块功能是完成上采样的最近邻处理，代码如图 5.1.3.2，实现方法是搭建逻辑接口连线电路，把相对应的输入输出口连接即可。

```
module u_sampling #(
    parameter data_i_width = 2,                 //输入数据矩阵长
    parameter data_i_height = 2,                //输入数据矩阵宽

    parameter scale_factor = 2,                 //最近邻采样大小倍数

    parameter data_o_width = data_i_width*scale_factor,     //输出数据矩阵长
    parameter data_o_height = data_i_height*scale_factor,   //输出数据矩阵宽
    parameter bitwidth = 3
) (
    input [data_i_width*data_i_height*bitwidth-1:0] data_i,
    output [data_o_width*data_o_height*bitwidth-1:0] data_o
);
//输入和对应的输出连起来
generate
genvar i,j,m,n;
for(i=0;i<data_i_height;i=i+1)begin
    for(j=0;j<data_i_width;j=j+1)begin
        for(m=i*scale_factor;m<i*scale_factor+scale_factor;m=m+1)begin
            for(n=j*scale_factor;n<j*scale_factor+scale_factor;n=n+1)begin
                assign data_o [(m*data_o_width+n)*bitwidth +:bitwidth] = data_i [(i*data_i_width+j)*bitwidth +:bitwidth];
            end
        end
    end
end
endgenerate


endmodule
```

图 5.1.3.2

仿真结果如图 5.1.3.3，经观察可发现已成功完成上采样操作。

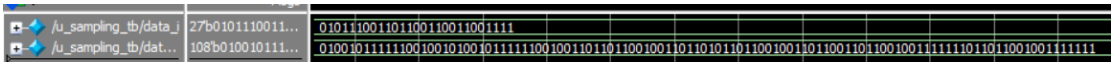

图 5.1.3.3

5.1.4 最大池化层

本层的作用是对数据进行最大池化处理。

最大池化层由 maxpool_top、maxpool_pick、maxpool_compare、maxpool_poolturn 四个模块组成。maxpool_top 的功能是进行各子模块之间信号的调度。maxpool_pick 的功能是开辟缓冲区暂时储存待处理信号，并且根据遍历坐标输出对应的数据。maxpool_compare 的功能是将从缓冲区提取出来的信号和临时值进行比较，确定一组数据中的最大值。

maxpool_top 的功能是完成各子模块信号之间的调度，输入输出端口如图 5.1.4.1，data_i 是处理数据输入,data_o 是处理完成的数据输出，work_en 是最大池化层接收上一级工作完成后发来的工作使能信号，work_fin 是最大池化层在结束工作时给输出给下一级的工作使能信号。

```
module maxpool_top #(
    parameter datai_width = 4,                  //输入数据矩阵长
    parameter datai_height = 4,                 //输入数据矩阵宽

    parameter kernel_width = 2,                 //最大池范围矩阵长
    parameter kernel_height = 2,                //最大池范围矩阵宽
    parameter stride = 2,                       //步长

    parameter padding_en =0,                    //padding开关
    parameter padding = 0,                      //padding行列数

    parameter datao_width = ((datai_width-kernel_width+2*padding)/stride)+1,        //输出矩阵的长
    parameter datao_height = ((datai_height-kernel_height+2*padding)/stride)+1,     //输出矩阵的宽

    parameter bitwidth = 3                      //位宽
) (
    input clk_en,                               //时钟
    input reset_n,                              //复位

    input work_en,                              //工作使能
    input [datai_width*datai_height*bitwidth-1:0] data_i,       //数据输入

    output [datao_width*datao_height*bitwidth-1:0] data_o,      //数据输出
    output work_fin                                             //工作结束
);
```

图 5.1.4.1

maxpool_poolturn 模块的功能是对 maxpool_pick 中的数据缓冲区进行遍历，输入输出端口如图 5.1.4.2，data_l 和 data_c 是遍历坐标输入端口。

在本模块中如图 5.1.4.3 开辟了一块逻辑电路用于存放输入数据，另外设计了如图 5.1.4.4 的数据选择器用于挑选数据输出。

```
module maxpool_pick #(
    parameter datai_width = 4,
    parameter datai_height = 4,

    parameter kernel_width = 2,
    parameter kernel_height = 2,
    parameter stride = 2,

    parameter padding_en =0,
    parameter padding = 0,

    parameter datao_width = ((datai_width-kernel_width+2*padding)/stride)+1,
    parameter datao_height = ((datai_height-kernel_height+2*padding)/stride)+1,

    parameter bitwidth = 3
) (
    input clk_en,
    input reset_n,
    input pool_on,

    input [datai_width*datai_height*bitwidth-1:0] data_i,          //输入数据

    input [3:0] data_l,                    //输入数据行
    input [3:0] data_c,                    //输入数据列

    output [bitwidth-1:0] data_o           //提取出的输入矩阵数据
);
```

图 5.1.4.2

```
//矩阵展开
wire [bitwidth-1:0] data [0:datai_height+2*padding-1][0:datai_width+2*padding-1];       //数据数据矩阵
generate
    genvar i,j;
    for(i=0;i<datai_height+2*padding;i=i+1)begin
        for (j=0;j<datai_width+2*padding;j=j+1)begin
            if (padding_en) begin       //padding处理块
                if (i<padding|i>datai_height|j<padding|j>datai_width) begin
                    assign data[i][j] = 0;
                end
                else begin
                    assign data[i][j] = data_i[((i-padding)*datai_width+j-padding)*bitwidth +:bitwidth];
                end
            end
            else begin
                assign data[i][j] = data_i[(i*datai_width+j)*bitwidth +:bitwidth];
            end
        end
    end
endgenerate
```

图 5.1.4.3

```
//将相应的输入数据提出
reg [3:0] rdata_l;
reg [3:0] rdata_c;
assign data_o = data[rdata_l][rdata_c];
always @(*)begin
    if(pool_on)begin
        rdata_l=data_l;
        rdata_c=data_c;
    end
    else begin
        rdata_l=0;
        rdata_c=0;
    end
end
```

图 5.1.4.4

maxpool_poolturn 模块的功能是对输入数据缓冲区与输出数据缓冲区两个缓冲区的存储位置进行遍历，输入与输出端口如图 5.1.4.5 所示，其中 resu_l 和 resu_c 是输出数据缓冲区遍历的一组坐标。

本模块设计了时序加法电路，为顺利完成遍历，代码中设计了两层流水，其在一组遍历完成前两拍触发，流水顺序为最大池处理区域更新-最大池数据遍历坐标回归左上角与输出数据缓冲区存储坐标更新。如图 5.1.4.6 展示输入数据缓冲区坐标遍历部分代码，其它遍历代码大同小异。

```
module maxpool_poolturn #(
    parameter datai_width = 4,
    parameter datai_height = 4,

    parameter kernel_width = 2,
    parameter kernel_height = 2,
    parameter stride = 1,

    parameter padding_en =0,
    parameter padding = 0,

    parameter datao_width = ((datai_width-kernel_width+2*padding)/stride)+1,
    parameter datao_height = ((datai_height-kernel_height+2*padding)/stride)+1,

    parameter bitwidth = 3
) (
    input clk_en,
    input reset_n,
    input pool_on,

    output [3:0] data_l,        //数据行
    output [3:0] data_c,        //数据列

    output [3:0] resu_l,        //结果矩阵行
    output [3:0] resu_c,        //结果矩阵列

    output part_fin,            //一个范围完成
    output turn_fin             //全部遍历完成
);
```

图 5.1.4.5

```
reg [3:0] rdata_l;              //输入数据行
reg [3:0] rdata_c;              //输入数据列
assign data_l = rdata_l;
assign data_c = rdata_c;
always @(posedge clk_en)begin
    if(!reset_n)begin
        rdata_l<=0;
        rdata_c<=0;
    end
    else if (pool_on) begin
        if(turn_fin)begin
            rdata_l<=rdata_l;
            rdata_c<=rdata_c;
        end
        else begin
            if(part_fin&datac_fin)begin
                rdata_l<=anchor_l;
                rdata_c<=anchor_c;
            end
            else if(~part_fin&datac_fin) begin
                rdata_l<=rdata_l+1;
                rdata_c<=anchor_c;
            end
            else begin
                rdata_l<=rdata_l;
                rdata_c<=rdata_c+1;
            end
        end
    end
```

图 5.1.4.6

maxpool_compare 的功能是完成数据的比较，并找出最大池区域内的最大值存到输出数据缓冲区内，最后将缓冲区内的数据转为并行数据进行输出。其输入输出端口如图 5.1.4.7 所示。

此模块搭建了比较器电路进行数据的比较，采用的方法为使用前一个比较出来的最大值和接下来的值作比较，若前值大，则保留；若后值大，则替换前值。在设计中，比较出来的值间接值将直接存入输出数据缓冲区，替换也直接和输出数据缓冲区进行替换，以提高运行效率。如图 5.1.4.8 展示了比较电路设计的部分代码。

```
module maxpool_compare #(
    parameter datai_width = 4,
    parameter datai_height = 4,

    parameter kernel_width = 2,
    parameter kernel_height = 2,
    parameter stride = 2,

    parameter padding_en =0,
    parameter padding = 0,

    parameter datao_width = ((datai_width-kernel_width+2*padding)/stride)+1,
    parameter datao_height = ((datai_height-kernel_height+2*padding)/stride)+1,

    parameter bitwidth = 3
) (
    input clk_en,
    input reset_n,
    input pool_on,
    input part_fin,
    input turn_fin,

    input [bitwidth-1:0] data,
    input [3:0] resu_l,
    input [3:0] resu_c,

    output [datao_width*datao_height*bitwidth-1:0] data_o,
    output all_fin
);
```

图 5.1.4.7

```
    ----
    else if (pool_on) begin
        if (turn_fin) begin
            for(i=0;i<datao_height;i=i+1)begin
                for(j=0;j<datao_width;j=j+1)begin
                    result_array[i][j]<=result_array[i][j];
                end
            end
        end
        else begin
            if(part_fin_q)begin
                result_array[resu_l][resu_c]<=data;
            end
            else begin
                if(result_array[resu_l][resu_c]<data)begin
                    result_array[resu_l][resu_c]<=data;
                end
                else begin
                    result_array[resu_l][resu_c]<=result_array[resu_l][resu_c];
                end
            end
        end
    end
    else begin
        for(i=0;i<datao_height;i=i+1)begin
            for(j=0;j<datao_width;j=j+1)begin
                result_array[i][j]<=0;
            end
        end
```

图 5.1.4.8

仿真结果如图 5.1.4.9 所示，设定步长为 2，padding 为 1，其中：

输入数据矩阵为：

$$\begin{bmatrix} 2 & 1 & 3 & 4 \\ 6 & 5 & 2 & 3 \\ 0 & 1 & 4 & 5 \\ 6 & 3 & 2 & 6 \end{bmatrix}$$

输出数据矩阵应为：

$$\begin{bmatrix} 2 & 3 & 4 \\ 6 & 5 & 5 \\ 6 & 3 & 6 \end{bmatrix}$$

经比较仿真结果无误。

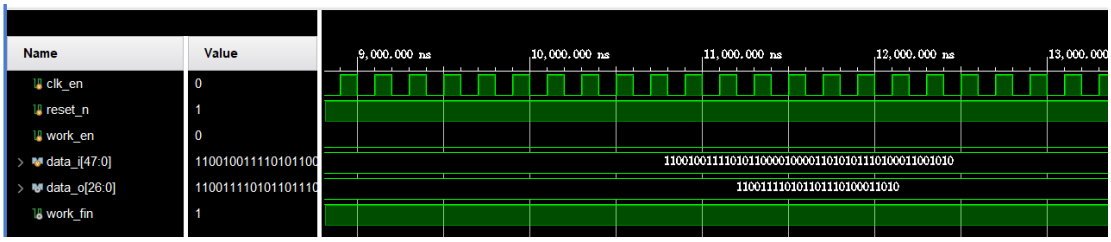

图 5.1.4.9

5.2 颜色识别模块

5.2.1 KNN 结构

KNN 结构由四个模块组成，其分别是 knn_top、knn_buf、knn_dis、knn_compare。knn_top 模块作为顶层模块，负责调度各子模块的信号。knn_buf 模块负责存储训练集。knn_dis 模块负责计算测试数据与各训练集数据之间的欧氏距离。knn_compare 模块负责对距离计算结果进行排序并且根据 k 值的大小统计输出识别结果。

knn_top 作为颜色识别模块的顶层模块，其中例化了 knn_buf、knn_dis、knn_compare，其输入输出端口如图 5.2.1.1 所示。其中除了 clk_en、rst_n 分别为时钟信号与复位信号输入端口，其余端口 knn_en 为 Knn 模块的使能信号，test_data 为测试数据输入端口，result 为识别结果输出端口，knn_fin 为识别完成信号输出接口。

```
module knn_top #(
    parameter train_amount=362,        //the number of the train data
    parameter RGB_bitwidth=24,         //the format of RGB
    parameter color_bitwidth=3,        //the bitwidth of color, five colors need 3 bitwdith
    parameter color_amount=5,          //the number of color
    parameter buf_bitwidth=9,          //the bitwidth of buf, 362 train data need 9 bitwidth
    parameter dis_bitwidth=25,         //the bitwidth of distance
    parameter k=3,
    parameter judge_bitwidth=dis_bitwidth+color_bitwidth,
    parameter train_bitwidth=RGB_bitwidth+color_bitwidth   //low 3 bits is color,0:red,1:orange,2:yellow,3:green,4:purple
) (
    input clk_en,                //clock
    input rst_n,                 //reset
    input knn_en,

    input [RGB_bitwidth-1:0]    test_data,

    output [color_bitwidth-1:0] result,
    output knn_fin
);
```

图 5.2.1.1

在 knn_top 模块中，设计了一组状态机来完成各子模块的调度与控制，同时状态机的设计也便于各子模块之间的调试。此状态机参照三段式状态机设计，如图 5.2.1.2、图 5.2.1.3、图 5.2.1.4 所示。以下对各个状态进行说明：IDLE_F 为 FPGA 刚开机时 Knn 模块进入的首次待机状态，此时的特点为 knn_fin 为低电

平；IDLE_C 为非首次待机状态，此状态在 Knn 模块完成一次工作之后就会进入，特点为 knn_fin 会输出高电平代表 Knn 模块已经完成了识别工作;CALCULATE 为欧式距离计算状态，此状态的特点为使能 knn_dis 模块进行工作;COMPARE 为结果处理状态,此状态的特点为使能 knn_compare 模块进行工作,同时停止 knn_dis 模块的工作。

```
),
//FSM
localparam  IDLE_F=0;
localparam  IDLE_C=1;
localparam  CALCULATE=2;
localparam  COMPARE=3;

reg [1:0] cur_state;
reg [1:0] nex_state;
always @(posedge clk_en or negedge rst_n) begin
    if (!rst_n) begin
        cur_state<=IDLE_F;
    end
    else begin
        cur_state<=nex_state;
    end
end
```

图 5.2.1.2

```
reg calculate_en;
wire calculate_fin;
reg compare_en;
wire compare_fin;
always @(*) begin
    case (cur_state)
        IDLE_F:begin
            if(knn_en)begin
                nex_state=CALCULATE;
            end
            else begin
                nex_state=IDLE_F;
            end
        end
        IDLE_C:begin
            if(knn_en)begin
                nex_state=CALCULATE;
            end
            else begin
                nex_state=IDLE_C;
            end
        end
        CALCULATE:begin
            if(calculate_fin)begin
                nex_state=COMPARE;
            end
            else begin
                nex_state=CALCULATE;
            end
        end
        COMPARE:begin
            if(compare_fin)begin
                nex_state=IDLE_C;
            end
            else begin
                nex_state=COMPARE;
            end
        end
        default:nex_state=IDLE_F;
    endcase
end
```

图 5.2.1.3

```
reg rknn_fin;
assign knn_fin=rknn_fin;
always @(posedge clk_en or negedge rst_n) begin
    if(!rst_n)begin
        calculate_en<=0;
        compare_en<=0;
        rknn_fin<=0;
    end
    else begin
        case(cur_state)
            IDLE_F:begin
                rknn_fin<=0;
            end
            IDLE_C:begin
                compare_en<=0;
                rknn_fin<=1;
            end
            CALCULATE:begin
                rknn_fin<=0;
                calculate_en<=1;
            end
            COMPARE:begin
                calculate_en<=0;
                compare_en<=1;
            end
            default:begin
               rknn_fin<=0;
               compare_en<=0;
               calculate_en<=0;
            end
        endcase
    end
end
```

图 5.2.1.4

knn_buf 模块为训练集数据存储模块，其输入输出端口如图 5.2.1.5。此模块例化了 IP 核 Logos DRM Based ROM 用于存储训练集数据，配置界面以及例化接口如图 5.2.1.6 和图 5.2.1.7。训练集数据在 IP 核初始化时即会存入 rom 中，rom 会根据 buf_addr 接口输入的地址数据将对应的数据通过 train_data 接口输出此模块。训练集数据为如图 5.2.1.8 中的存储于后缀为.dat 文件中的一系列数据，数据位宽为 27 位，按照最右边为最低位的定位标准，前 24 位为 RGB888 数据，后 3 位为颜色对应的序号，其中定义 000 代表红色,001 代表橙色,010 代表黄色,011 代表绿色,100 代表紫色。

```
module knn_buf #(
    parameter train_amount=362,        //the number of the train data
    parameter RGB_bitwidth=24,         //the format of RGB
    parameter color_bitwidth=3,        //the bitwidth of color, five colors need 3 bitwdith
    parameter color_amount=5,          //the number of color
    parameter buf_bitwidth=9,          //the bitwidth of buf, 362 train data need 9 bitwidth
    parameter dis_bitwidth=25,         //the bitwidth of distance
    parameter k=3,
    parameter judge_bitwidth=dis_bitwidth+color_bitwidth,
    parameter train_bitwidth=RGB_bitwidth+color_bitwidth   //low 3 bits is color,0:red,1:orange,2:yellow,3:green,4:purple
) (
    input clk_en,
    input rst_n,
    input calculate_en,

    input [buf_bitwidth-1:0] buf_addr,

    output  [train_bitwidth-1:0]    train_data
);
```

图 5.2.1.5

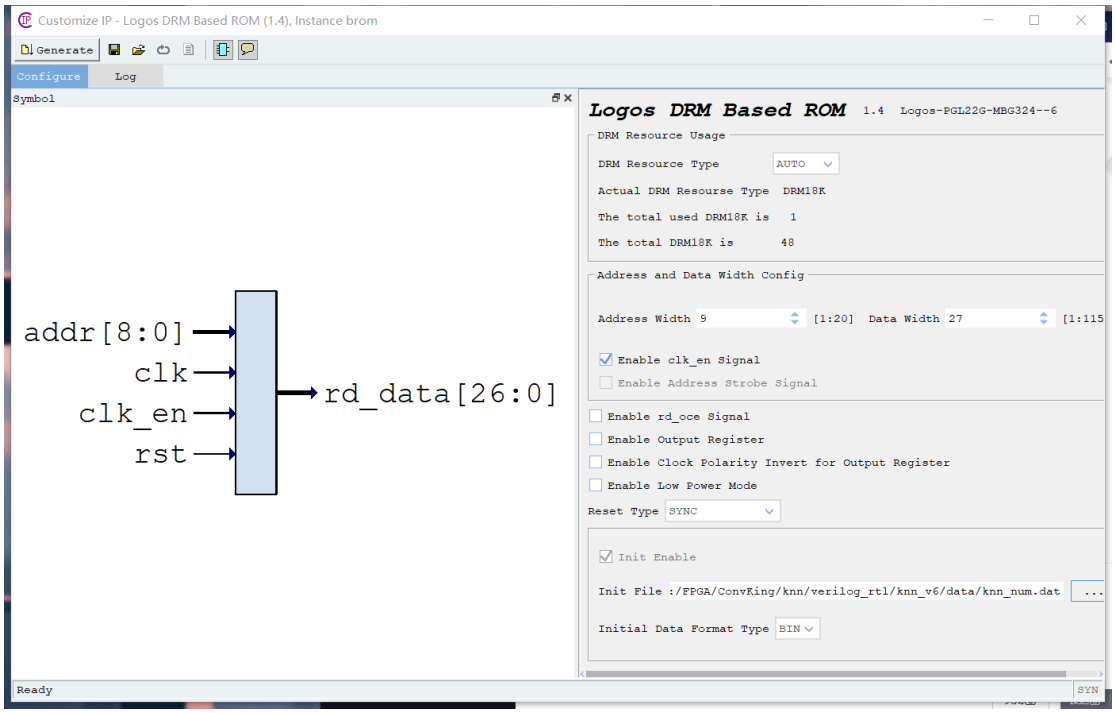

图 5.2.1.6

```
,,
//read the data from the brom and store in the buffer
wire [train_bitwidth-1:0]    rd_data;
assign train_data=rd_data;

//inst brom ip
brom brom_inst(
    .clk        (clk_en),
    .rst        (!rst_n),
    .addr       (buf_addr),
    .clk_en     (calculate_en),
    .rd_data    (rd_data)
);
```

图 5.2.1.7

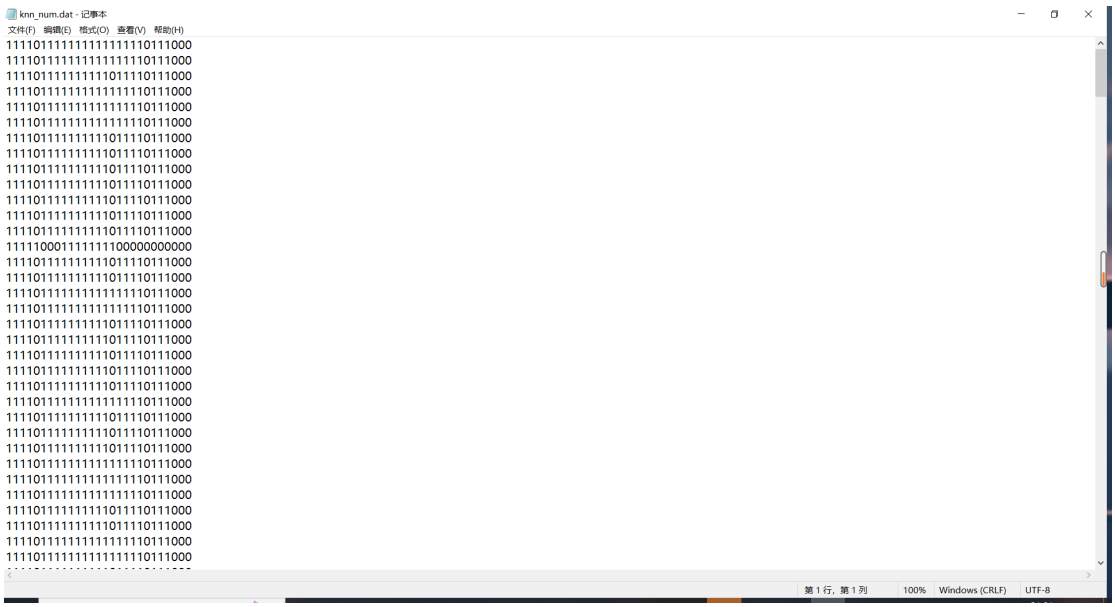

图 5.2.1.8

knn_dis 模块的功能是计算测试集数据与训练集数据之间的欧式距离，其输入输出端口如图 5.2.1.9 所示。模块中通过设计加法计数器并输出 buf_addr 的值对 knn_buf 模块中的训练集数据进行遍历。模块将 knn_buf 中传回的训练集数据即 train_data 与测试数据 test_data 进行欧式距离计算，将结果再组合后通过 judge_data 端口输出。通过组合，judge_data 低 3 位依旧代表了此结果对应的训练集颜色，而剩下的位数则代表了欧式距离的大小。模块主要代码如图 5.2.1.10 所示。

```
module knn_dis #(
    parameter train_amount=15,
    parameter RGB_bitwidth=24,
    parameter color_bitwidth=3,
    parameter buf_bitwidth=4,
    parameter dis_bitwidth=32,
    parameter judge_bitwidth=dis_bitwidth+color_bitwidth,
    parameter train_bitwidth=RGB_bitwidth+color_bitwidth    //low 3 bits is color,0:red,1:yellow,2:green,3:orange,4:purple
) (
    input clk_en,
    input rst_n,
    input calculate_en,

    input [train_bitwidth-1:0]  train_data,
    input [RGB_bitwidth-1:0]    test_data,

    output [buf_bitwidth-1:0] buf_addr,
    output [judge_bitwidth-1:0] judge_data,
    output calculate_fin
);
```

图 5.2.1.9

```
// calculate Euclidean Distance
assign judge_data[color_bitwidth-1:0]=train_data[color_bitwidth-1:0];
assign judge_data[judge_bitwidth-1:color_bitwidth]=(train_data[color_bitwidth+7:color_bitwidth]-test_data[7:0])*(train_data[color_bitwidth+7:color_bitwidth]-test_data[7

//the addr of the buffer
reg [buf_bitwidth-1:0]  rbuf_addr;
reg rcalculate_fin;
assign calculate_fin=rcalculate_fin;
assign buf_addr=rbuf_addr;
always @(posedge clk_en or negedge rst_n) begin
    if(!rst_n)begin
        rbuf_addr<=0;
        rcalculate_fin<=0;
    end
    else if(calculate_en) begin
        if(rbuf_addr==train_amount)begin
            rcalculate_fin<=1;
            rbuf_addr<=rbuf_addr;
        end
        else begin
            rbuf_addr<=rbuf_addr+1;
        end
    end
    else begin
        rcalculate_fin<=0;
        rbuf_addr<=0;
    end
end
```

图 5.2.1.10

knn_compare 模块的功能是根据定义的 k 值的大小，选出欧式距离最小的前 k 个计算结果，并且将这 k 个中间结果进行投票，找出 k 个中间结果中出现最多的颜色，最后将此颜色作为识别结果进行输出，输入输出端口如图 5.2.1.11。本模块例化了 Logos DRM Based Single Port RAM 来储存由 knn_dis 模块输出的欧式距离计算结果，配置界面以及例化接口如图 5.2.1.12 和图 5.2.1.13。为了节省计算资源，本模块中在选出前 k 个欧式距离最小值的时候并没有采用传统的排序挑选算法。在本模块的设计中，使用加法计数器对欧式距离结果进行遍历，而 k 值的大小决定了对所有欧式距离结果的遍历次数。在每一次遍历中都会找出

一个最小值，同时在一次遍历结束后又会将 ram 中此地址的值中除后三位之外的所有位数置 1，以此来避免这个地址的值下一次遍历时还会被选为最小值。由此在 k 此循环中即可挑选出前 k 个最小值，部分关键代码如图 5.2.1.14 与图 5.2.1.15 所示。在模块设计了一个寄存器来对这 k 个结果作直方图统计，并找出出现最多的颜色，关键代码如图 5.2.1.16 所示。最终识别结果会从 result 端口输出。

```
module knn_compare #(
    parameter train_amount=15,
    parameter RGB_bitwidth=24,
    parameter color_bitwidth=3,
    parameter color_amount=5,
    parameter buf_bitwidth=4,
    parameter dis_bitwidth=25,
    parameter k=3,
    parameter judge_bitwidth=dis_bitwidth+color_bitwidth,
    parameter train_bitwidth=RGB_bitwidth+color_bitwidth    //low 3 bits is color,0:red,1:yellow,2:green,3:orange,4:purple
) (
    input clk_en,
    input rst_n,
    input compare_en,
    input calculate_en,

    input [judge_bitwidth-1:0]  judge_data,
    input [buf_bitwidth-1:0]    buf_addr,

    output [color_bitwidth-1:0] result,
    output compare_fin
);
```

图 5.2.1.11

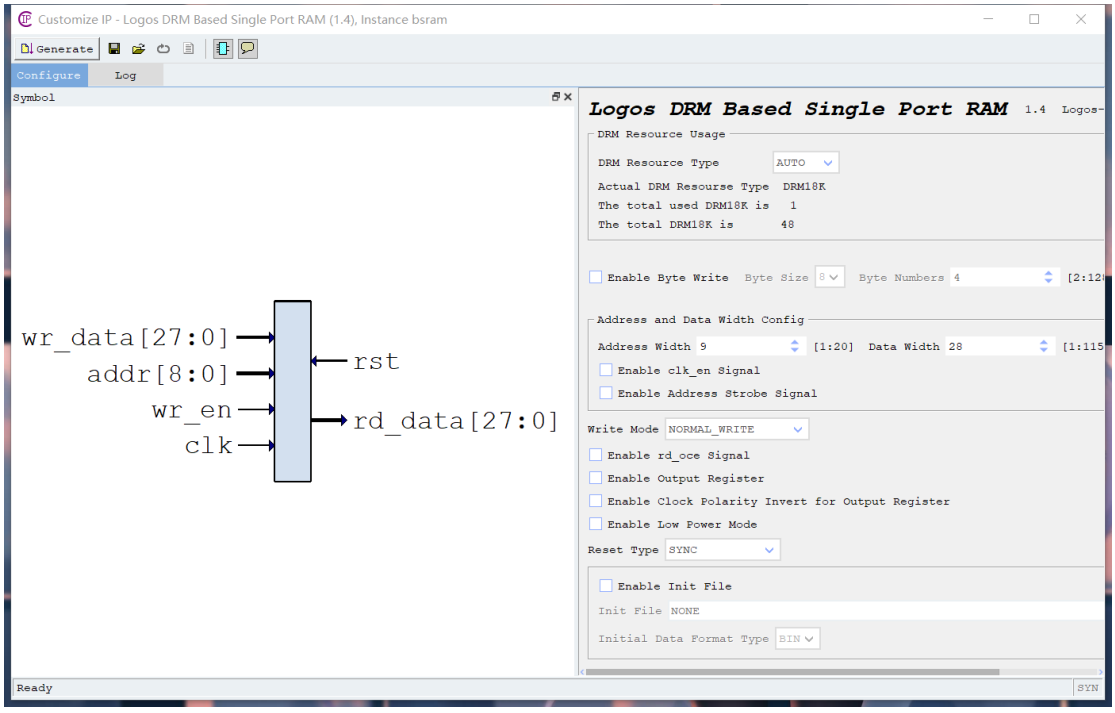

图 5.2.1.12

```
//the ram to store the data after calculating and to take the data for compare
wire [judge_bitwidth-1:0] judge_rd;
reg  [judge_bitwidth-1:0] judge_wr;
wire [judge_bitwidth-1:0] ram_wr;
wire [buf_bitwidth-1:0] ram_addr;
assign ram_addr=calculate_en?buf_addr_q:(cnt_inside_fin?judge_mark:cnt_inside);
assign ram_wr=calculate_en?judge_data:judge_wr;
bsram   bsram_inst(
    .clk        (clk_en),
    .rst        (!rst_n),
    .wr_en      (calculate_en|cnt_inside_fin),
    .addr       (ram_addr),

    .wr_data    (ram_wr),
    .rd_data    (judge_rd)
);
```

图 5.2.1.13

```
//compare
reg [judge_bitwidth-1:0]  judge_temp;
always @(*) begin
    if (!rst_n) begin
        judge_temp=0;
        judge_mark=0;
    end
    else if(compare_en) begin
        if(cnt_inside_q==0)begin
            judge_temp[judge_bitwidth-1:0]=judge_rd[judge_bitwidth-1:0];
            judge_mark=cnt_inside;
        end
        else begin
            if(judge_temp[judge_bitwidth-1:color_bitwidth]>judge_rd[judge_bitwidth-1:color_bitwidth])begin
                judge_temp[judge_bitwidth-1:0]=judge_rd[judge_bitwidth-1:0];
                judge_mark=cnt_inside_q;
            end
            else begin
                judge_temp=judge_temp;
                judge_mark=judge_mark;
            end
        end
    end
    else begin
        judge_temp=0;
        judge_mark=0;
    end
end
```

图 5.2.1.14

```
end
//the fowllowing codes is about voting
//vote
reg [color_bitwidth-1:0]  vote_temp;
always @(*) begin
    if(!rst_n)begin
        vote_temp=0;
        judge_wr=0;
    end
    else if(compare_en) begin
        if(cnt_inside_fin)begin
            judge_wr[judge_bitwidth-1:0]={{25{1'b1}},judge_temp[color_bitwidth-1:0]};//change to the max num
            vote_temp=judge_temp[color_bitwidth-1:0];
        end
        else begin
            vote_temp=vote_temp;
        end
    end
    else begin
        judge_wr=judge_data;
    end
end
```

图 5.2.1.15

```
//find the max of the vote
reg [3:0] cnt_outside_q0;
always @(posedge clk_en or negedge rst_n) begin
    if(!rst_n)begin
        cnt_outside_q0<=0;
    end
    else begin
        cnt_outside_q0<=cnt_outside;
    end
end

reg [color_bitwidth-1:0] max_vote;
always @(posedge clk_en or negedge rst_n) begin
    if (!rst_n) begin
        max_vote<=0;
    end
    else if (compare_en) begin
        if(cnt_inside_fin)begin
            if(cnt_outside_q0==0)begin
                max_vote<=vote_temp;
            end
            else begin
                if(vote[max_vote]<(vote[vote_temp]+1))begin
                    max_vote<=vote_temp;
                end
                else begin
                    max_vote<=max_vote;
                end
            end
        end
        else begin
            max_vote<=max_vote;
        end
    end
    else begin
        max_vote<=0;
    end
end
```

图 5.2.1.16

仿真结果如图 5.2.1.17，其中：

测试数据为 24'b0010_1010_1000_1100_1000_1100，预期结果为绿色，即输出为 011。

训练集数据在图 5.2.1.7 中已做展示。

观察仿真结果， result 输出为 010，即绿色代表的序号，颜色识别正确。

此外在 test_bench 中还准备了其它的测试数据，如图 5.2.1.18。经过测试，均能输出正确结果。

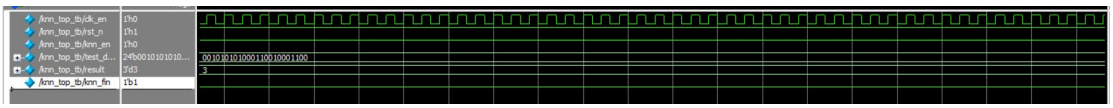

图 5.2.1.17

```
initial begin
    test_data=24'b0010_1010_1000_1100_1000_1100;//green,011
    //test_data=24'b1111_1010_1111_1010_1111_1010;//yellow,010
    //test_data=24'b1100_0100_1000_0111_0000_0000;//orange,001
    //test_data=24'b0010_0011_0010_1000_0101_1100;//yellow,010
    //test_data=24'b0100_0000_0100_1101_1111_1111;//green,011

end
```

图 5.2.1.18

5.3 前处理与后处理模块

CNN 和 KNN 加速电路与前处理模块独立，前处理模块用于进行图片缩放、色彩空间变换等工作。后处理模块主要用于水果种类、数量、颜色识别码的生成和 VGA 时序匹配。

CNN 电路的前处理模块使用均匀下采样算法实现将 1024*768 像素的 RGB565 图像转换为 416*416 的 RGB888 图像，并行输出三个通道数据给后续的 CNN 电路处理。KNN 电路的前处理模块将输入图像从 RGB 颜色空间转换到 HSV 颜色空间，进行直方图统计后输出每个通道对应的像素数；KNN 模块接收对应像素值，进行运算后输出可能的颜色值。

CNN 电路的后处理模块根据 CNN 的输出坐标对预测框进行标记，对应像素点被填充为红色，输出识别后的图像；模块实现了 SoftMax 算法，根据置信度从高到低排序后根据最高置信度对水果种类识别码编码。

5.3.1 RGB2HSV 模块

rgb2hsv 模块用于将输入图像从 RGB 颜色空间转换到 HSV 颜色空间。模块采用 ISP 的设计思路，使用 VGA 时序的 HSYNC 时钟、VSYNC 时钟、PCLK 时钟驱动，按像素处理输入的图像数据，由于数据计算中只需要等待两个时钟周期，因此可以实现实时的图像格式转换。

5.3.2 直方图统计模块

直方图统计模块根据 OpenCV 开源计算机视觉库的直方图统计算法实现。基本思路即遍历输入图像，统计其中每个通道内具有不同通道值的所有像素数目，并将数值最大的通道值输出。

硬件实现分成两个主要模块，分别执行遍历输入图像任务和获取最大值任务。以 RAM 的不同地址作为不同通道值，每个像素输入时根据其每个通道值对 RAM

寻址，并将该地址中数据自增 1，也就是说对应地址内存储的数据即单帧中该值对应的像素数目；在每个时钟周期内都可以遍历到一个像素，因此能够直接使用像素时钟 PCLK 作为模块时钟；当遍历完一帧数据后直接将每个地址对应的值输出到最大值比较模块，由于帧时序由帧同步时钟控制，因此可以直接将帧同步时钟作为输出控制。双口 RAM 能够同时完成计数和读写任务，利用双口 RAM 同时读写的特性将复杂的图像遍历转换成实时计数算法，从而大大加快算法执行速率。最大值比较模块基于比较移位寄存器（FIFO 比较器），RAM 的地址值会按时钟移位到 FIFO，如果下一个 RAM 地址中的值比当前 FIFO 头数据对应地址中的值大，则将下一个地址存到当前 FIFO 头，在帧同步时钟到达即一帧图像遍历完成后，仍然留在 FIFO 头的地址值即直方图统计的输出值。

6 设计参数

使用到的板上资源：

按钮 x4、LEDx4、HDMI 接口、UART 接口、DVP 摄像头接口、SPI FLASH

片上资源空载使用量：

LUTs：15.63%

Registers：5.62%

DRM18K：8.33%

片上资源带权重使用量：

LUTs：84.11

Registers：75.38%

DRM18K：30.12%

功耗：约 5W

权重大小：41.3MB

空载速率：约 30 帧

识别速率：约 10 帧