基于高层次综合的
轻量级嵌入式车牌识别系统

摘要

车牌识别技术对于实现交通自动化管理，建设智能交通城市有着现实意义。传统的车牌识别系统所使用的算法，大多基于 CPU 或 GPU 处理器框架，FPGA 由于其相对较高的开发难度而应用面稍窄。近年，Xilinx 公司推出了 FPGA+ARM 架构的 Zynq 芯片，兼具 FPGA 的高实时性与 CPU 的灵活性，结合同样新推出的 Vitis 统一软件平台，使得在嵌入式设备上实现高复杂度的算法成为可能。在这样的客观条件下，本小组研究实现了一款基于高层次综合的嵌入式轻量级车牌识别系统。该系统以功耗较低的轻量级开发板 Zedboard 为硬件平台， 完成了从视频采集、缓存到图像处理再到字符识别的全过程。其中，字符识别功能是由我们自主部署于 Zedboard 上的 LeNet 神经网络完成的，并能达到 20 毫秒/字的识别速率和 9% 以上的准确率，具有一定实时性和准确性，总体性能优良。若将本系统部署于性能更优秀、资源更充足的硬件平台，识别速率和准确度将获得更大的提升。

关键词：车牌识别，FPGA，Zynq，软硬件协同，高层次综合

目录

1 项目背景 3
1.1 车牌识别技术的应用场景 3
1.2 开发环境 3
2 系统框架与方案论证 4
2.1 系统框架 4
2.2 方案论证 4
2.2.1 摄像头视频采集方案 4
2.2.2 二值化处理方案 5
2.2.3 车牌区域定位方案选择 5
2.2.4 车牌字符切割方案选择 6
2.2.5 车牌字符识别方案选择 6
3 系统模块设计 6
3.1 基于 AXI4-Stream 的视频采集、缓存、处理、显示 6
3.1.1 摄像头视频采集 6
3.1.2 视频流的格式转换 7
3.1.3 视频流的缓存机制 8
3.1.4 多路视频流的叠层显示 9
3.1.5 HDMI 输出驱动 9
3.2 基于 FPGA+ARM 的车牌图像处理 10
3.2.1 FPGA 完成车牌图像的二值化处理 11
3.2.2 ARM 实现车牌定位与字符分割 13
3.3 基于高层次综合的 LENET 神经网络推理实现 16
3.3.1 使用 Tensoflow 训练 LENET 模型 16
3.3.2 基于权重量化的神经网络压缩 17
3.3.3 基于合理组织循环顺序的神经网络加速 18
4 验证与测试 24
4.1 硬件连线 24
4.2 视频采集-缓存-显示验证测试 25
4.3 视频采集-缓存-处理-显示测试 26
4.4 车牌区域检测效果测试 27
4.5 车牌字符切割效果测试 28
4.6 车牌字符识别正确率测试 29
4.7 整体测试 30
5 项目创新点 36
5.1 使用了 FPGA 搭建车牌识别系统 36
5.2 采用轻量级图像处理算法完成车牌定位与字符分割 36
5.3 采用了 FPGA+ARM 的方式完成图像处理算法 36
5.4 采用高层次综合的方法实现 LeNet 神经网络的推理 36
6 总结与期望 37

1 项目背景

1.1 车牌识别技术的应用场景

车牌识别是图像识别技术在车辆牌照识别领域中的具体应用，具有极高的应用价值。例如：高速公路的 ETC 收费系统可利用车牌识别实现自动扣费，无需车辆停车取卡，提高了收费站的工作效率；高速公路的测速监测点可自动抓拍超速车辆并识别其车牌，并将违章车牌号码发送至执法人员处，节省警力，快捷高效……可见，车牌识别技术具有极其广泛的应用场景。

1.2 开发环境

FPGA 开发板：Xilinx Zynq-7000 系列开发板 Zedboard

开发软件：Vivado、Vivado HLS、Vitis 统一软件平台

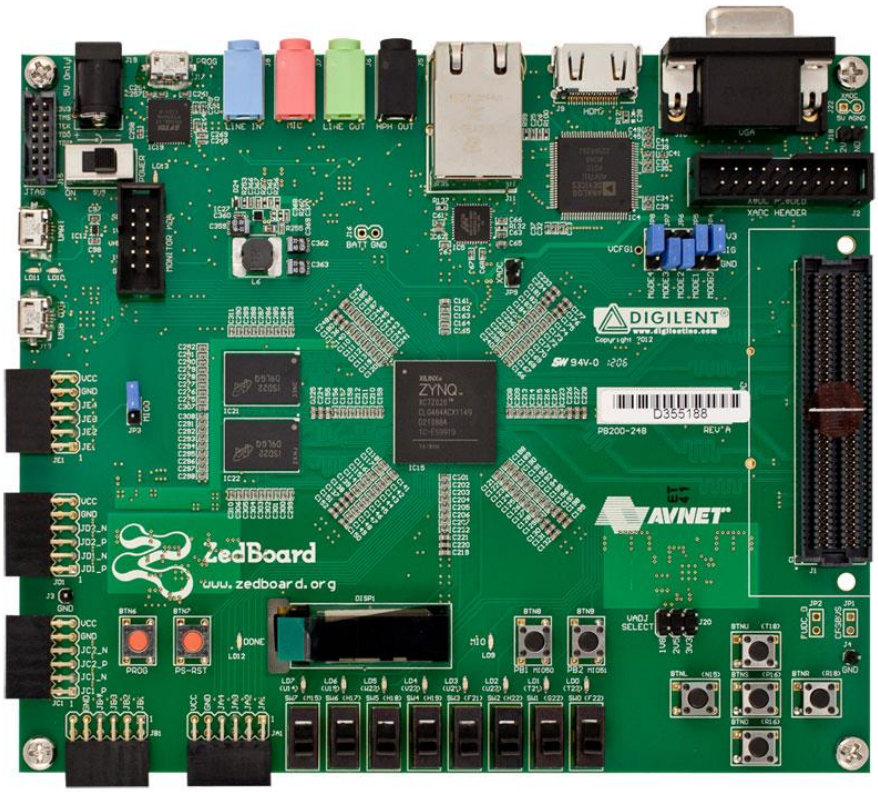

图 2-1 Zedboard

2 系统框架与方案论证

2.1 系统框架

我们的车牌识别系统主要由三部分组成：首先，摄像头采集视频并进行缓存；随后对采集的视频进行处理，包括二值化去噪、车牌定位和字符分割；最后逐个识别单个车牌字符。系统框图如下：

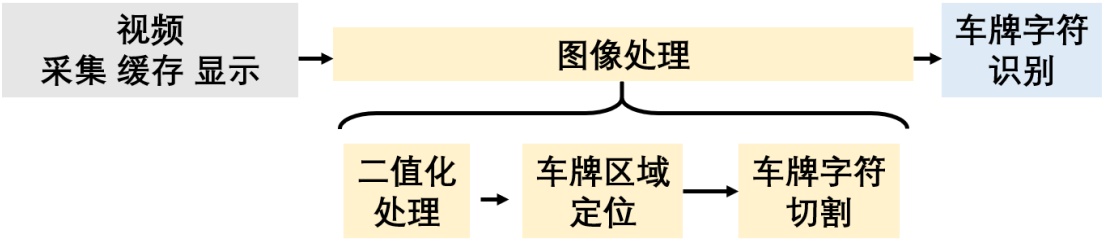

图 3-1 系统框图

2.2 方案论证

对于上述的每一个子模块，都有许多方案可供选择。具体讨论如下：

2.2.1 摄像头视频采集方案

我们采用 OV5640 摄像头。该摄像头采用 DVP 接口，可以通过转接板与 Zedboard 上的 PMOD 接口相连，适用于资源有限的 FPGA 开发板，且价格便宜，是 FPGA 上进行视频采集的最佳传感器。

视频缓存方案

方案一：将摄像头采集到的视频通过单缓存法缓存至 Block RAM。该方法优点是实现简单，无需了解如何使用 Vivado 的 IP 核。该方法缺点是占用了太多的 Block RAM 的空间。

方案二：将摄像头采集到的视频转换为 AXI4-Stream 的视频流形式，通过 VDMA 缓存至 DDR。该方法不占用 Block RAM 空间，且可以采用多缓存机制。该方法缺点是需要学习如何正确使用 VDMA IP 核。

我们对方案一与方案二都进行了尝试。方案一占用过多的 BRAM，导致后续的图像处理和神经网络可用的资源不足。因此采取方案二。

2.2.2 二值化处理方案

我们采用的是基于 HSV 色彩空间的方法进行图像的二值化处理。由于常见的车牌大多为蓝色，我们设置阈值将蓝色区域保留，将非蓝色区域去除，以此来实现图像的二值化。二值化的具体实现方案有如下选择：

方案一：采用 Vivado HLS 的 hls_video.h 视频处理库完成图像的二值化处理。该方法优点是开发简单，且图像处理速度极快，可达到超高的实时性。该方法缺点是 hls_video.h 库所提供的函数甚少，难以完成较为复杂的图像处理。

方案二：采用 Vitis Vision 库完成图像处理。该视觉库功能强大，提供了较多的 API，能实现复杂的图像处理。但是该库推出时间较短，参考资料较少。

由于时间有限，我们采取了较为简单的方案一作为图像二值化处理的方案。

2.2.3 车牌区域定位方案选择

方案一：采用 C++的 OpenCV 中的 Findcontour()函数，将二值图中的轮廓全部找出来，并以特定的层次结构将轮廓坐标存至 Vector 中。该方法的优点是车牌区域定位效果极佳，缺点是 Vivado HLS 中没有该函数，手动实现非常困难。

方案二：采用广度搜索算法遍历二值图，搜索出所有连通域的上下左右的边界坐标。该优点是易于实现，缺点是效果较差，耗时较久。

由于 Findcontour 函数的手动实现过于困难，我们采用传统的广度搜索算法来实现车牌区域的定位。

2.2.4 车牌字符切割方案选择

方案一：根据车牌字符的等间距排列特点，直接将车牌区域七等份完成切割。该方法实现简单，速度快，但效果不佳，准确度低

方案二：使用垂直投影的方法进行车牌字符切割，设置投影阈值，从而划分车牌字符的边界。该方法实现较复杂，但效果较好，准确度高。

由于切割准确度直接影响识别准确度，所以我们采用方案二实现车牌切割。

2.2.5 车牌字符识别方案选择

方案一：采用传统的机器学习方法，如 KNN，回归，支持向量机等方法进行实现。该方案计算量小，识别速度快，但识别准确率较低。

方案二：采用 LeNet 卷积神经网络进行实现。该方案计算量较大，但识别准确率高。

为了尽可能地提高我们的车牌识别系统的性能，我们决定采用卷积神经网络来实现车牌字符的识别。

3 系统模块设计

3.1 基于 AXI4-Stream 的视频采集、缓存、处理、显示

3.1.1 摄像头视频采集

我们采取 CMOS 类型数字图像传感器 OV5640 作为视频采集设备。该传感器支持输出最大为 500 万像素的图像，支持使用 VGA 时序输出图像数据，输出图像的数据格式支持 YUV(422/420)，YCbCr422，RGB565 以及 JPEG 格式。它还可以对采集得的图像进行补偿，支持伽玛曲线、白平衡、饱和度、色度等基础处理。根据不同的分辨率配置，传感器输出图像数据的帧率从 15-60 帧可调。

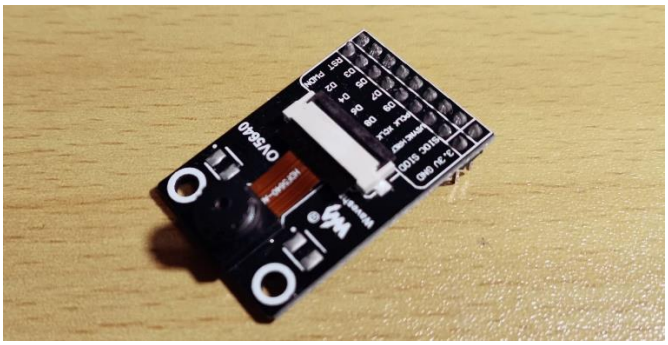

图 4-1 OV5640 摄像头

根据 OV5640 数据手册的寄存器配置方式，我们在 PS 端调用 I2C 接口对 OV5640 的相关寄存器进行配置。根据 OV5640 数据手册上的视频时序图，我们在 PL 端使用 Verilog 编写 OV5640 摄像头的驱动程序。我们以摄像头的输出时钟 PCLK 为“节拍”设计 always 块，捕获 VSYNC 和 HREF 信号，接收 DATA[7:0] 数据，把 OV5640 输出的视频时序信号转换成符合 Video In to AXI4-Stream 输入接口的时序信号（即下图的橙色高亮引脚）。

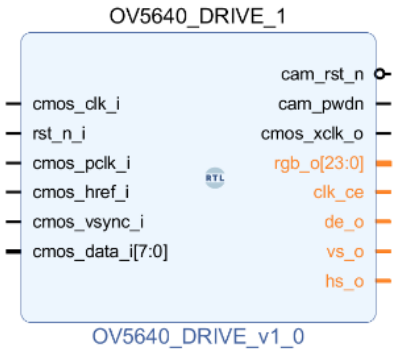

图 4-2 OV5640 RTL Module

3.1.2 视频流的格式转换

我们不能直接处理 OV5640 输出的视频时序信号。Xilinx 所有与视频有关的 IP 均遵循一套 AXI4-Stream 视频流协议，Vivado 中的视频信号都以 AXI4-Stream 协议进行高性能高速传输，所以我们需要通过 Video In to AXI4-Stream 将 OV5640 输出的视频时序信号转换成 AXI4-Stream，以便于后续与 VDMA 和 HLS IP 核进行连接。OV5640 RTL Module 与 Video In to AXI4-Stream 的具体连接如下图所示。

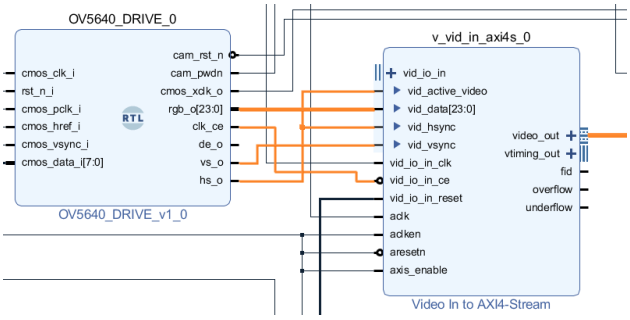

图 4-3 OV5640 RTL Module 与 Video In to AXI4-Stream 的连接

3.1.3 视频流的缓存机制

AXI VDMA（Video Direct Memory Access）是 Xilinx 提供的软核 IP，用于将 AXI Stream 格式的数据流转换为 Memory Map 格式或将 Memory Map 格式的数据转换为 AXI Stream 数据流，从而实现与 DDR 进行通信。

许多视频类应用都需要帧缓存来处理帧率变化或者进行图像的缩放、裁剪等尺寸变换操作。AXI VDMA 设计的初衷就是用于高效地实现 AXI4-Stream 视频流接口和 AXI4 接口之间的数据传输。我们的车牌识别项目的流程为“采集-缓存-处理-显示”，其中“缓存”就需要用到 VDMA。下图为我们系统 VDMA IP 的连接图。

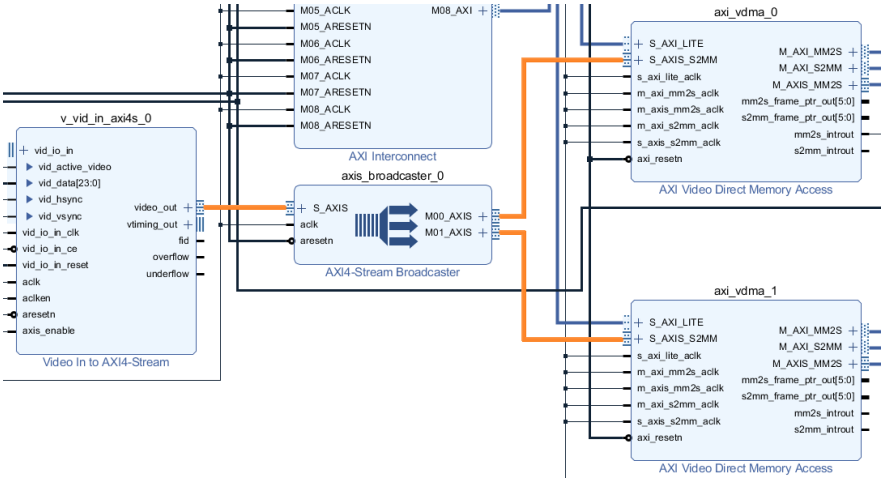

图 4-4 Block Design 中的 VDMA

如图所示，Video In to AXI4-Stream 的视频流输出通过 AXI4-Stream Broadcaster“兵分两路”，分别进入两个 VDMA 进行缓存，这样可以方便我们后续将对这两路视频流做不同的处理。

3.1.4 多路视频流的叠层显示

由 AXI4-Stream Broadcaster 产生的两路视频流经过不同的处理后，我们希望能对这两路视频流进行切换显示。Video Mixer（视频混屏器）可以实现这个功能。

Video Mixer 允许我们通过在 PS 端对相应的 Layer 进行 Enable 或 Disable，即可对该层进行 On/Off 操作。除此之外，Video Mixer 还有其他更强大的功能：支持多个层色彩透明；各层既可以是内存映射的 AXI4 接口，也可以是 AXI4-Stream；提供可编程背景色彩；提供可编程层位置及尺寸；可按 1 倍、2 倍或 4 倍提供层缩放；可选内建色彩空间转换等。下面是 Video Mixer 在 Block Design 的连线。

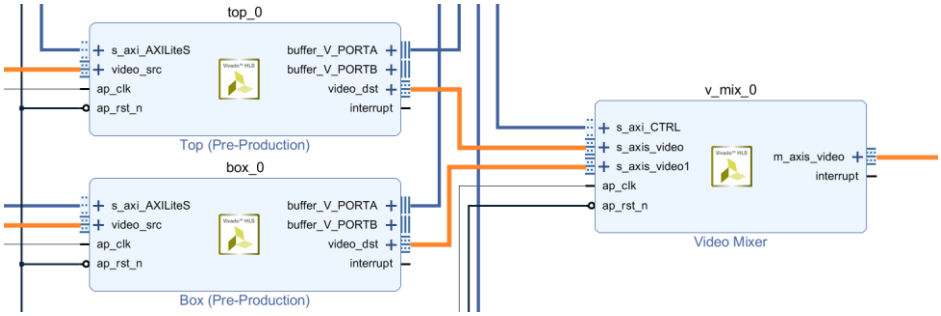

图 4-5 Block Design 中的 Video Mixer

3.1.5 HDMI 输出驱动

将 Video Mixer 输出的视频流和 Video Timing Controller 输出的时序连接到 AXI4-Stream to Video Out IP，即可将视频流转换成包含有帧同步（vid_hsync），行同步（vid_vsync）的视频时序信号。HDMI_Display IP 对这两个信号进行捕捉，接收 RGB 数据，并转换为 HDMI 显示器所需的视频时序信号，从而可在 HDMI 显示器上显示 Video Mixer 输出的视频流。下面是 AXI4-Stream to Video Out 在 Block Design 的连线：

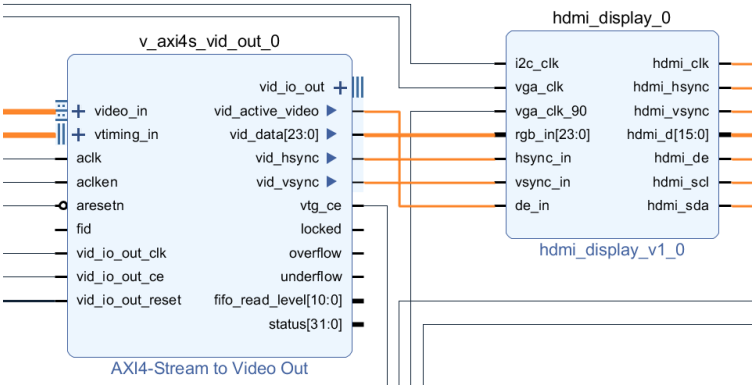

图 4-6 AXI4-Stream to Video Out 在 Block Design 的连线

以上就是视频的“采集-缓存-处理-显示”整个过程所需要用到的所有 IP 核。由于视频数据以 AXI4-Stream 形式进行传输，所以在传输过程中，若把视频流看作水流，其流向如下图所示。

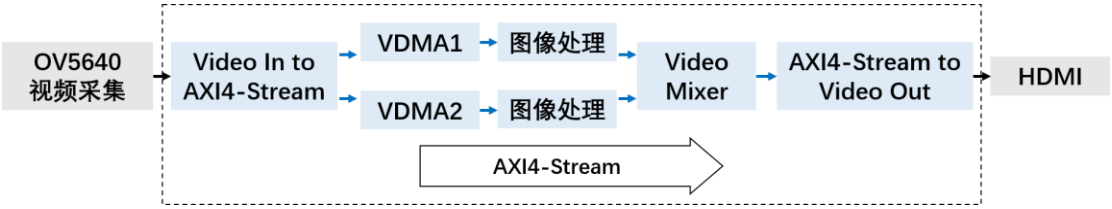

图 4-7 视频数据总流向

3.2 基于 FPGA+ARM 的车牌图像处理

我们的车牌图像处理算法充分利用了 Zynq 系列芯片的 FPGA+ARM 的架构，合理地将图像处理任务分配至 PL 端（FPGA 端）与 PS 端（ARM 端），令 FPGA 和 ARM 各自发挥优势，协同作用，实现高性能的图像处理功能。

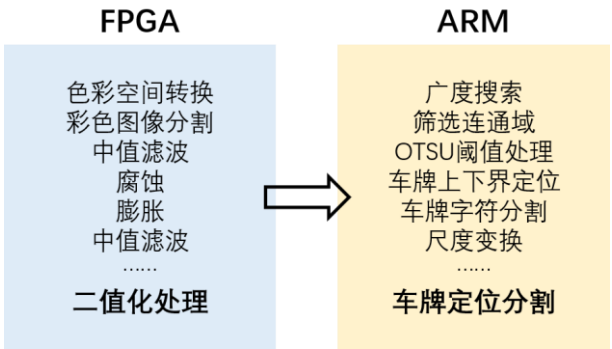

图 4-8 FPGA 与 ARM 协同作用

3.2.1 FPGA 完成车牌图像的二值化处理

图像的二值化处理主要的过程为：从摄像头采集的图像中定位车牌位置。首先将输入图像转换到 HSV 颜色空间，设置蓝色车牌底色阈值范围，进行颜色过滤，得到保留蓝色区域的二值化图像。颜色过滤后，采用中值滤波去除一些较小的孤立白点。为了使用广度搜索筛选车牌位置，二值化图像中的车牌区域必须尽量连通，因此我们采取一系列的腐蚀膨胀操作，确保二值化图像中的白色相邻区域连通，便于车牌区域的提取。上述一系列操作的流程图如下所示。

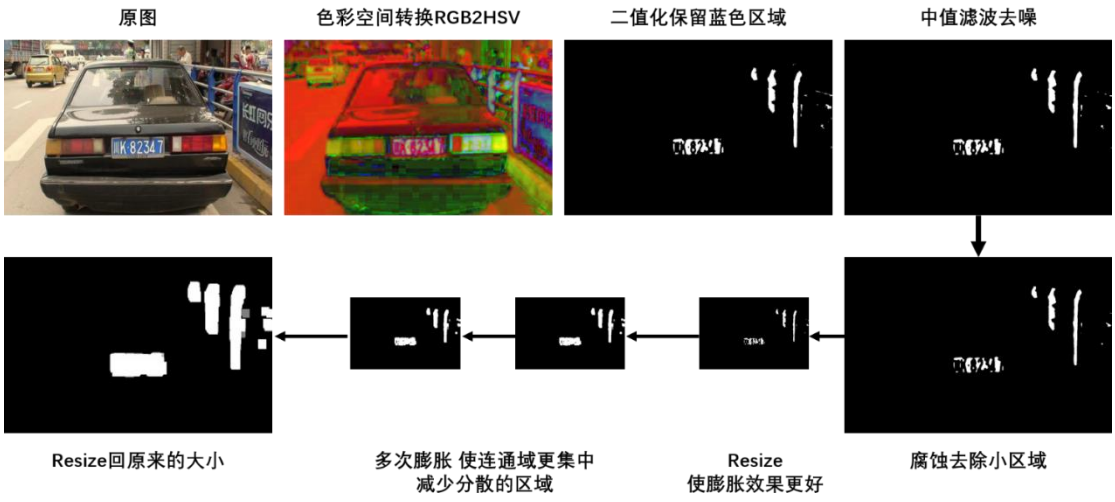

图 4-9 PL 端处理流程图

我们使用高层次综合（Vivado HLS）在 FPGA 端完成整个二值化处理过程。不同于 CPU 和 GPU 以帧为单位进行图像处理，FPGA 进行图像处理不需要等待整一帧缓存完毕，而是以“行”为单位进行流水线运算。若在 FPGA 上对一个输入图像执行一个 3x3 的高斯滤波操作，FPGA 只需耗费缓存两行的延时时间，便可以开始计算第一个输出像素点。而 CPU 或 GPU 则需等待整一帧缓存完毕后，才能开始第一个输出像素点的计算。FPGA 的这一特性，使它成为许多低延迟任务的不二选择。

传统的使用基于 RTL 硬件语言的 FPGA 开发模式进行图像处理难度大且耗时久，若使用 Vivado HLS 工具则方便不少。Vivado HLS 中带有 hls_video 库，可对视频流进行色彩空间转换（hls::Cvtcolor），高斯滤波（hls::GaussianBlur），腐蚀（hls::Erode），膨胀（hls::Dilate）等操作，可以非常方便地进行图像处理。但是对于 hls_video 库中未包含的图像处理函

数，我们就只能自己手动实现了。例如：在 hls_video 库中仅有高斯滤波函数。我们便自主实现了滤波效果更好的中值滤波函数。

不同于普通的面向过程或者面向对象，使用 Vivado HLS 编写程序时需要有硬件思维。使用 Vivado HLS 实现图像处理函数时，最重要的是启动间距（Initiation Interval）需要达到 1，即实现每一个 Clock 输出一个像素。否则视频流就会在该函数处阻塞，无法正常运行。经过合理地编程，添加合适的优化条件后，我们自己手动实现的 5x5 中值滤波函数的启动间距（Initiation Interval）达到了要求。

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- L1_L2	921610	921610	12	1	1	921600	yes

图 4-10 启动间距

```
void MedianBlur_5(IMAGE_C1 &src,IMAGE_C1 &dst){
    short int r, c;
    PIXEL_C1 pixelt1,pixelt2;
    pix_t pix, med, window[KMED_5x5], pixel[KMED_5];
    static pix_t line_buffer[KMED_5][MAX_WIDTH];
    #pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1
    L1:for(r = 0; r < MAX_HEIGHT; r++) {
        L2:for(c = 0; c < MAX_WIDTH; c++){
            #pragma HLS PIPELINE II=1
            for(int i = 0; i < KMED_5-1; i++){
                line_buffer[i][c] = line_buffer[i+1][c];
                pixel[i] = line_buffer[i][c];
            }
            src>>pixelt1;
            pixel[KMED_5-1]=line_buffer[KMED_5-1][c]=pixelt1.val[0];
            for(int i = 0; i < KMED_5; i++)
                for(int j = 0; j < KMED_5-1; j++)
                        window[i*KMED_5+j] = window[i*KMED_5+j+1];
            for(int i = 0; i < KMED_5; i++)
                window[i*KMED_5+KMED_5-1] = pixel[i];
            med = median_5(window);
            if ((r>=KMED_5-1)&&(r<MAX_HEIGHT)&&(c>=KMED_5-1)&&(c<MAX_WIDTH)) pixelt2.val[0] = med;
            Else pixelt2.val[0] = 0;
            dst<<pixelt2;
        }
    }
    return;
}
```

图 4-11 自主实现的 5x5 中值滤波函数

最终，我们在 Vivado HLS 所实现的二值化处理程序仿真综合后的资源占用率报告如下。受限于 Zedboard 自身的资源，二值化处理占用了较大比例的资源空间。但依旧有充足的资源完成后续的算法。

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1850301	2139214	20.538 ms	23.745 ms	929938	2139187	dataflow

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	36	-
FIFO	0	-	288	1260	-
Instance	26	34	26987	32958	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	-	-	6	-	-
Total	26	34	27281	34290	0
Available	280	220	106400	53200	0
Utilization (%)	9	15	25	64	0

图 4-12 二值化资源占用报告

3.2.2 ARM 实现车牌定位与字符分割

使用 FPGA 进行图像处理最大的优势在于其高速性与低延时性，而这一优势只有当视频数据以“流”的形式进行传输时才能存在。

若在 Vivado HLS 上运行复杂条件判断、跳转或需要动态的内存分配的程序，FPGA 则会失去其优势，甚至综合出来的 RTL 硬件描述程序会不满足时序要求，无法完成预期功能。

车牌的区域定位与车牌字符的分割任务，无法在 FPGA 用简单的腐蚀膨胀二值化等形态学处理完成，所以我们使用更灵活的 ARM 端实现这一部分的功能。下面是使用 ARM 处理器图像处理的框架图。

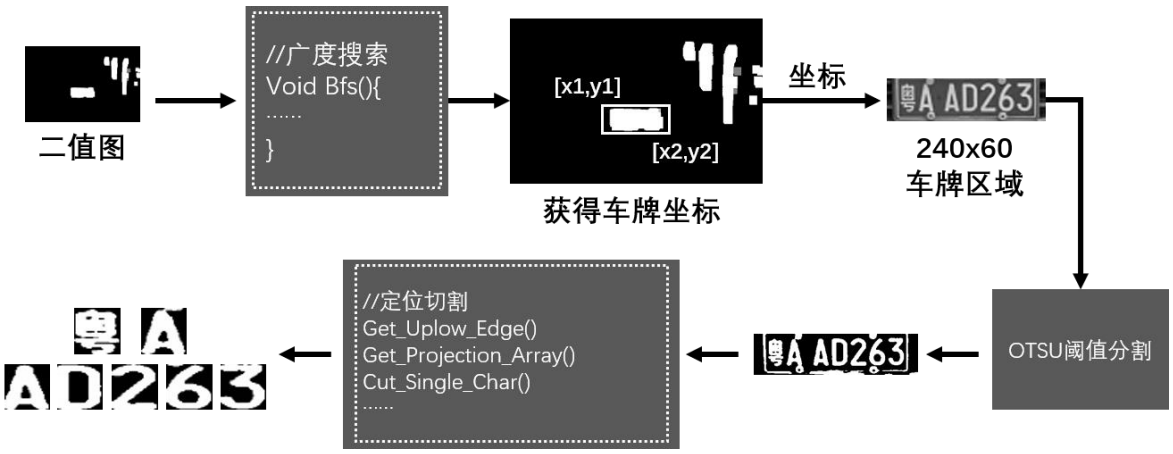

图 4-13 ARM 端图像处理框图

3.2.2.1 车牌区域定位

我们使用广度搜索，得到二值化图像中各个连通域最小外接矩形的位置坐标，并且通过车牌长宽比等形态特征去除不合要求的区域，筛选得到车牌位置坐标。

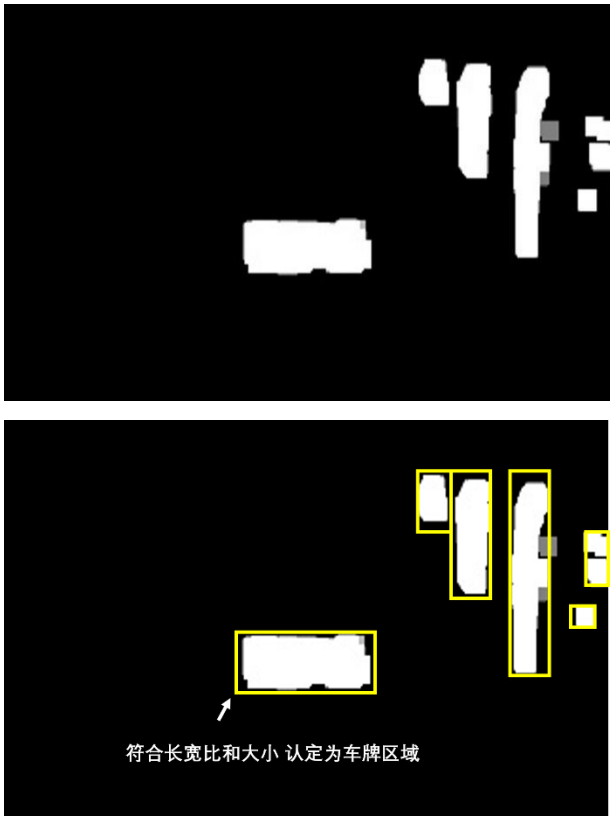

图 4-14 车牌区域筛选

3.2.2.2 车牌字符分割

获得车牌区域坐标后，我们开始车牌字符的分割。

由于车牌的上下边框和铆钉会对后续的字符分割与字符识别带来干扰，因此必须对车牌区域进行精定位以去除上下边框和铆钉。首先，我们将输入车牌区域图像进行灰度化和 OTSU 二值化处理。由于在车牌区域中，字符区在水平方向黑白跳变次数较多，而上下边框和铆钉区域在水平方向像素跳变次数较少，因此我们可以从车牌区域中间依次向上向下遍历每一行像素，并且设定阈值，

记录黑白跳变次数。当跳变次数低于阈值时，即认为达到车牌字符的边界，从而实现车牌精定位。示意图如下：

图 4-15 车牌字符分割

在去除上下边框和铆钉之后，我们使用基于最大间隔和垂直投影的方法进行车牌字符分割。根据车牌形态特征，我们注意到第二个字符和第三个字符之间的间隔最大，因此可以根据此特征找到第二个字符的左边界和第三个字符的右边界，从而实现车牌所有字符的分割。首先我们对二值化车牌进行垂直投影，白色像素点记为 1，黑色像素点记为 0，得到一维投影数组。将一维投影数组中宽度过小且高度过小的值置为零，从而去除车牌区域中的部分白色噪点和第二个字符与第三个字符之间的分隔点。然后遍历该数组，找到最大间隔，从而确定第二个字符左边界和第三个字符右边界，并且可进一步确定所有字符左右边界，实现字符分割。垂直投影示意图如下。

图 4-16 垂直投影示意图

下面我们通过一个实例来展示整个车牌定位与字符分割的流程。如下图所示。

图 4-17 PS 端完整流程图

3.3 基于高层次综合的 LENET 神经网络推理实现

3.3.1 使用 Tensoflow 训练 LENET 模型

我们在 Tensorflow 上将 LeNet 搭建好，并在网上找到单一车牌字符的训练集，进行训练和测试。最终识别成功率约为 9%。其中形状较为简单的数字与字母识别正确率较高，较为复杂的汉字识别正确率较低。

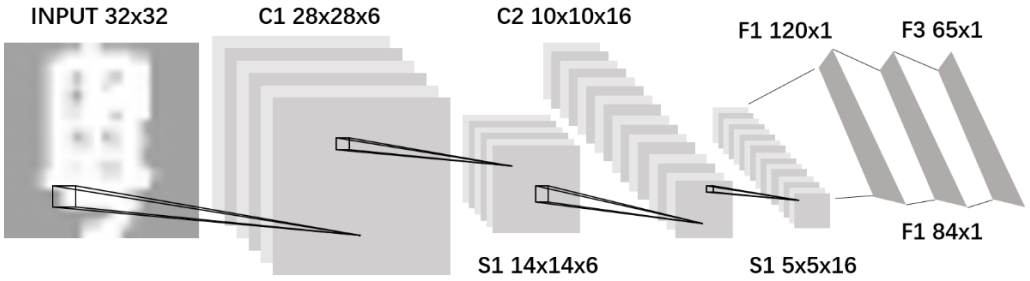

图 4-18 LeNet 结构图

图 4-19 训练数据

我们将训练好的模型权重以文本格式导出，在 Vivado HLS 中以 include 头文件方式实现权重的导入。

3.3.2 基于权重量化的神经网络压缩

量化权重和神经元是模型压缩的常用方法。在常见的深度学习开发框架中，神经网络的权重和神经元常常用浮点数来表示。近年来，有许多学者开始尝试对权重与神经元进行量化，即用低精度的定点数代替浮点数表示权重与神经元。一方面，使用更少位数的定点数来表示权重和神经元能够减少神经网络推理系统的内存带宽；另一方面，使用更简单的定点数表示方式能够减少运算所带来的硬件开销。同时，许多研究表明，数据量化对神经网络推理带来的精度下降往往在人们的接受范围之内。

数据量化的位数直接影响了模型识别效果。如果数据量化位数过少，则许多原本不为零的神经元直接被量化成零，导致神经元“失活”，且另一些神经元量化的结果会与原来的值相差较大，在后续的计算中误差被放大，导致识别结果不正确。如果数据量化的位数过大，则使得内存需求增大，计算开销增大，失去了量化的意义。量化位数常常需要通过实验得到，也常常需要在牺牲识别精度和牺牲内存空间二者之间做出折中的选择。

我们在 FPGA 上使用 Vivado HLS 实现神经网络的推理，使用的是 FPGA 端的硬件资源，包括 BRAM（Block Ram），DSP48，LUT（Look-up Table）等。FPGA 的资源有限，使用数据量化非常必要。若合适地以低精度的数据类型表示权重，能很好地降低 FPGA 的资源利用率，亦能大大地提高计算速度。

Vivado HLS 提供的任意精度定点数据类型，为权重的量化提供了很大的便利。在 LeNet 的实现过程中，我们通过大量实验，决定将 float32 的浮点数据类型权重量化为 12bit 的定点数据类型，其中 6 位为整数部分，6 位为小数部分。在减少了 6%的权重大小同时，仅仅下降了不到一个百分点，我们的识别精度由原来的 9%下降至 9%，依旧很好地完成了识别任务。

	Float32	16bit	**12bit**	8bit
识别正确率	**97.4%**	97.2%	**96.8%**	92.4%

表格 4-1 浮点位数及正确率

3.3.3 基于合理组织循环顺序的神经网络加速

神经网络的推理需要进行大量的运算，而 FPGA 的结构非常适合并行化任务。所以使用 FPGA 进行神经网络推理计算，从而实现并行加速，是一个切实可行的方案。然而在推理过程中，数据有着读写的先后依赖关系，因此需要合理地组织计算顺序才能达到最大并行度、最大计算速率和数据吞吐量。

下面我们以实现一个卷积层为例来展示我们如何通过合理地组织计算顺序来提高并行度。卷积计算本质上是一个六重循环，通过相乘累计的方式计算出卷积结果。要合理地组织计算顺序，先要合理地组织循环顺序。同时，由于较大的数组在 HLS 中会综合成 BRAM，存储在 BRAM 中的数据只能通过单端口或双端口进行数据访问，这将会限制并行计算能力。所以合理地将数组的维度进行划分也是实现高度并行化的关键。合理组织循环顺序，合理对数组维度进行划分之后，便可以添加 pipeline 或 unroll 的 HLS directives，实现计算并行化。

我们先实现一个最原始的卷积函数，并逐步对函数进行修改，体现合理组织循环顺序的过程。

IN_CH	OUT_CH	ROW_IN	COL_IN	ROWS	COLS	K
6	16	14	14	10	10	5

表 4-2 卷积函数中的常量参数

Algorithm 1

float *In[IN_CH][ROW_IN][COL_IN];*
float *Out[OUT_CH][ROW][COL];*
float *W[OUT_CH][IN_CH][K][K];*

1 Output_Channel: **for** *cho=0 to OUT_CH: {*
2 Input_Channel: **for** *chi=0 to IN_CH: {*
3 ROWS: **for** *r=0 to ROW: {*
4 COLS: **for** *c=0 to COL: {*
5 Kernel_Row: **for** *kr=0 to K: {*
6 Kernel_Col: **for** *kc=0 to K: {*
7 *Out[cho][r][c]+=In[chi][r+kr][c+kc]*W[cho][chi][kr][kc];*
8 *} } } } } }*

图 4-20 算法 1

算法 1 没有进行任何优化，没有优化流水线，也没有对循环进行展开。在不添加任何优化时，循环顺序并不影响卷积计算所需要的时间。结果表明，该算法只占用了很少的资源，但是时间开销很大。综合后所得到的报告如下。

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Output_Channel	2037344	2037344	127334	-	-	16	no
+ Input_Channel	127332	127332	21222	-	-	6	no
++ Row	21220	21220	2122	-	-	10	no
+++ Col	2120	2120	212	-	-	10	no
++++ Kernel_Row	210	210	42	-	-	5	no
+++++ Kernel_Col	40	40	8	-	-	5	no

图 4-21 算法 1 的 Latency 和 Intervel

在此基础上，我们可以添加优化，改变循环顺序，进行并行加速。虽然并行加速必定会带来更多硬件资源的消耗，但在资源相对充足的条件下，这是些消耗是允许的。我们将标号为 Output Channel 的 for 语句与标号为 Input Channel 的 for 语句移至最内层，并且对输入输出层相应的维度进行全部划分。算法如下：

Algorithm 2

```
float In[IN_CH][ROW_IN][COL_IN];
float Out[OUT_CH][ROW][COL];
float W[OUT_CH][IN_CH][K][K];

#pragma HLS array_partition variable=In complete dim=1
#pragma HLS array_partition variable=Out complete dim=1
#pragma HLS array_partition variable=W complete dim=1
#pragma HLS array_partition variable=W complete dim=2

1 ROWS: for r=0 to ROW: {
2 COLS: for c=0 to COL: {
3 Kernel_Row: for kr=0 to K: {
4 Kernel_Col: for kc=0 to K: {
5 #pragmas HLS pipeline
5 Output_Channel: for cho=0 to OUT_CH: {
7 Input_Channel: for chi=0 to IN_CH: {
9 Out[cho][r][c]+=In[chi][r+kr][c+kc]*W[cho][chi][kr][kc];
10 } } } } } }
```

图 4-22 算法 2

再次综合，得到的报告显示：仅仅改变循环的顺序和加了两行优化条件，卷积层的运行速度就比原来快了 102 倍。可见，在添加优化后，合理地组织循环顺序，可以大幅度地提升程序的运行速度。

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Kernel_Row_Kernel_Col	20000	20000	9	8	1	2500	yes

图 4-23 算法 2 的 Latency 和 Intervel

但算法 2 依仍有优化的空间。观察综合过程中所发出的警告，我们发现：部分数据在运算过程中出现了读写依赖，使得程序无法进一步进行优化，无法进一步提升并行度。具体产生的数据依赖如下：当进入第一次循环时(r=c=kr=kc=0)，Out [0][0][0]这一元素先是被读取，然后再被写入。而进入第二次循环时(r=c=kr=0, kc=1)，Out [0][0][0]依旧要被读取和写入。即：相邻两次循环需要对同一块内存进行读写，这样程序无法完全地实现流水化。如图所示：

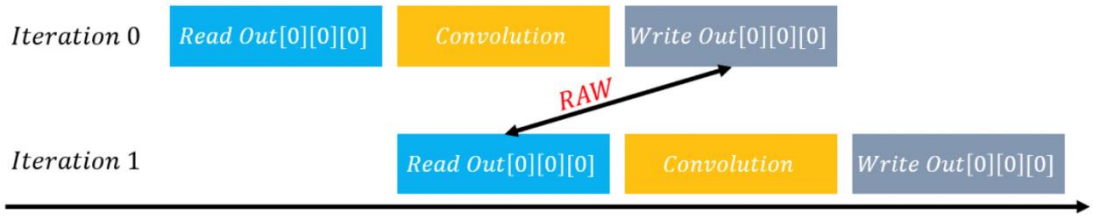

图 4-24 for 循环中存在 RAW（Read after Write）读写依赖

如果我们再次更改循环顺序，将标号为 Kernel_Row 的循环和标号为 Kernel_Col 的循环移至最外层，则可消除上述所提到的 RAW 读写依赖关系，如图所示：

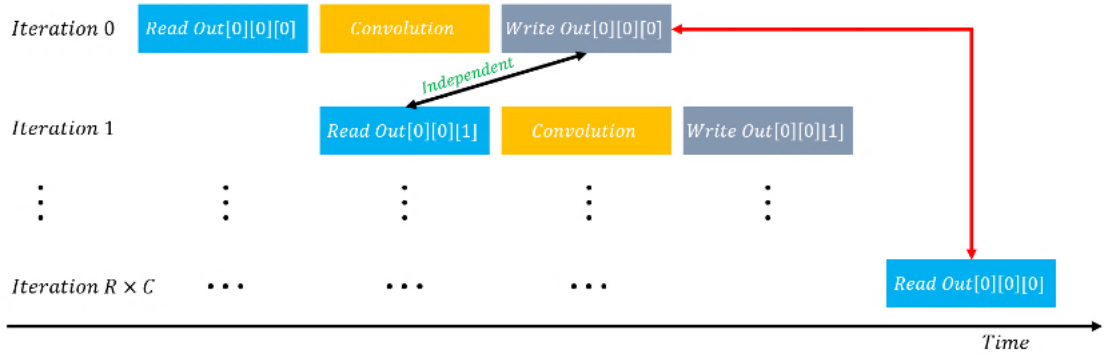

图 4-25 for 循环中的 RAW（Read after Write）读写依赖消失了

将循环顺序再次改变后，我们得到了卷积函数最终的版本。如下图的算法 3。

Algorithm 3

```
float In[IN_CH][ROW_IN][COL_IN];
float Out[OUT_CH][ROW][COL];
float W[OUT_CH][IN_CH][K][K];

#pragma HLS array_partition variable=In complete dim=1
#pragma HLS array_partition variable=Out complete dim=1
#pragma HLS array_partition variable=W complete dim=1
#pragma HLS array_partition variable=W complete dim=2

1 Kernel_Row: for kr=0 to K: {
2 Kernel_Col: for kc=0 to K: {
3 ROWS: for r=0 to ROW: {
4 COLS: for c=0 to COL: {
5 #pragmas HLS pipeline
5 Output_Channel: for cho=0 to OUT_CH: {
7 Input_Channel: for chi=0 to IN_CH: {
9 Out[cho][r][c]+=In[chi][r+kr][c+kc]*W[cho][chi][kr][kc];
10 } } } } } }
```

图 4-26 算法 3

最终，该卷积函数的综合报告显示：相比起算法 1，算法 3 的速度提升了 812 倍。仅需 25us 就能完成一次卷积运算。可见，合理组织循环顺序，添加合理的优化条件，能对性能带来质的飞跃。

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Kernel_Row_Row_Col	2507	2507	9	1	1	2500	yes

图 4-27 算法 3 的的 Latency 和 Intervel

根据以上原理，我们可以对池化层和全连接层进行类似的优化。但是，由于并行优化需要大量消耗资源，而图像处理部分所留下的资源已不支持我们进行全体优化了，我们只能退而求其次进行部分优化。经过部分优化后，LeNet 神经网络推理所占用的资源如下：

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1908	-
FIFO	-	-	-	-	-
Instance	31	4	6096	10331	-
Memory	14	-	48	29	0
Multiplexer	-	-	-	672	-
Register	-	-	769	-	-
Total	45	4	6913	12940	0
Available	280	220	106400	53200	0
Utilization (%)	16	1	6	24	0

图 4-28 部分优化的资源占用率

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
2935322	2978666	29.353 ms	29.787 ms	2935322	2978666	none

图 4-29 部分优化的 Latency 和 Intervel

最后，我们将各个子模块拼接起来，经过仿真，综合，布局布线，最终生成硬件平台的 xsa 描述文件与比特流。整个项目综合并布局布线后，占用的资源利用率如下：

LUT %	FF %	BRAM %	URAM %	DSP %	LUTRA...	IO %	GT %	BUFG %	MMCM %
0.00	**0.00**	**0.0**	**0.00**	**0.00**	**0.00**	**22.50**	**0.00**	**0.00**	**0.00**
78.80	45.03	41.1	0.00	42.27	9.25	23.00	0.00	21.88	25.00

图 4-30 全项目的资源占用情况

其中除了 LUT（Look-Up Table）占用率较高外（主要是二值化处理与 LeNet 神经网络占用了大量的资源），其他的资源都有较多的剩余。可见本系统“轻量”的特点。

4 验证与测试

4.1 硬件连线

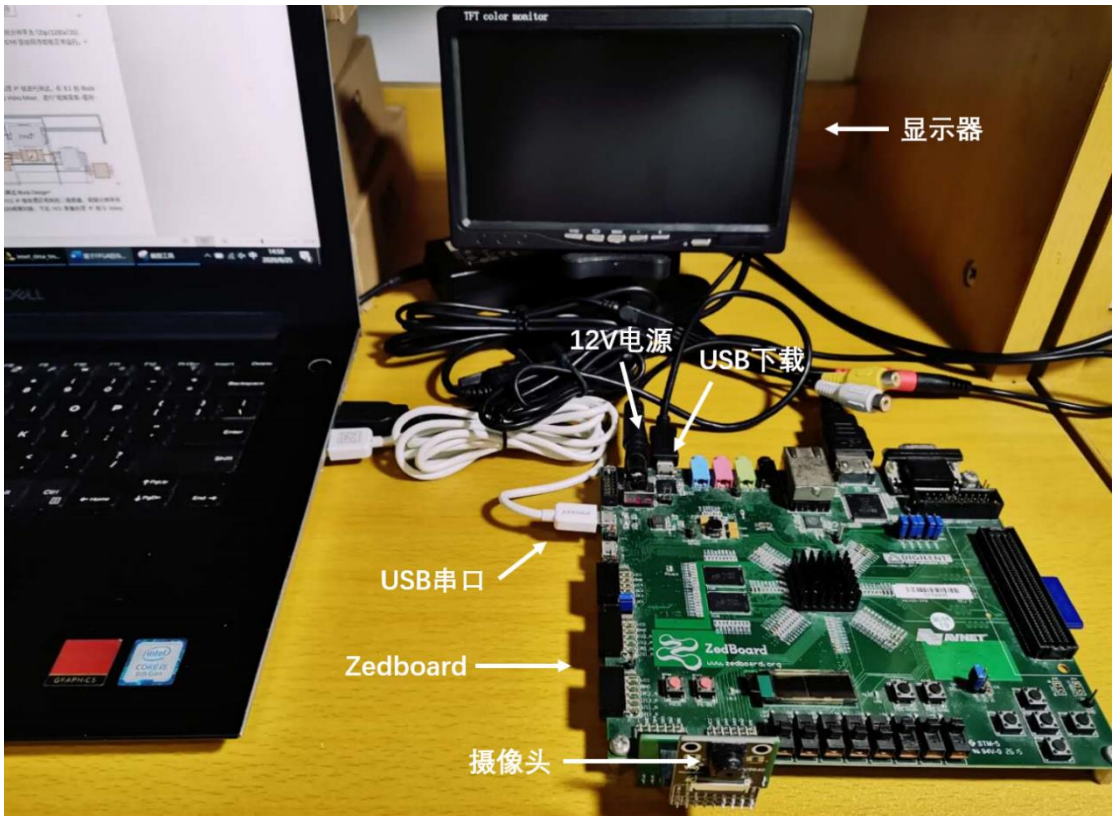

图 5-1 硬件连线图

开始测试之前，先完成硬件连线。将摄像头 PMOD 接头插入 Zedboard 的 PMOD A B 接口中，将显示器的 HDMI 接头与 Zedboard 连接，将 Zedboard 上的两个 USB 接口与电脑连接，其中一个作为 USB 用作串口输出，另外一个用作 USB 下载比特流对 Zedboard 进行烧录。最后将 Zedboard 与显示器分别和两个 12V 电源适配器连接。打开 Zedboard 的开关，即可烧录程序。打开显示器开关，即可进行视频显示。

4.2 视频采集-缓存-显示验证测试

为了验证摄像头与显示器能正常工作，我们先去除项目中的其它功能模块，仅保留摄像头驱动 IP，VDMA 视频缓存 IP 和 HDMI 驱动 IP，以便于测试“采集-缓存-显示”这一最基本的功能。

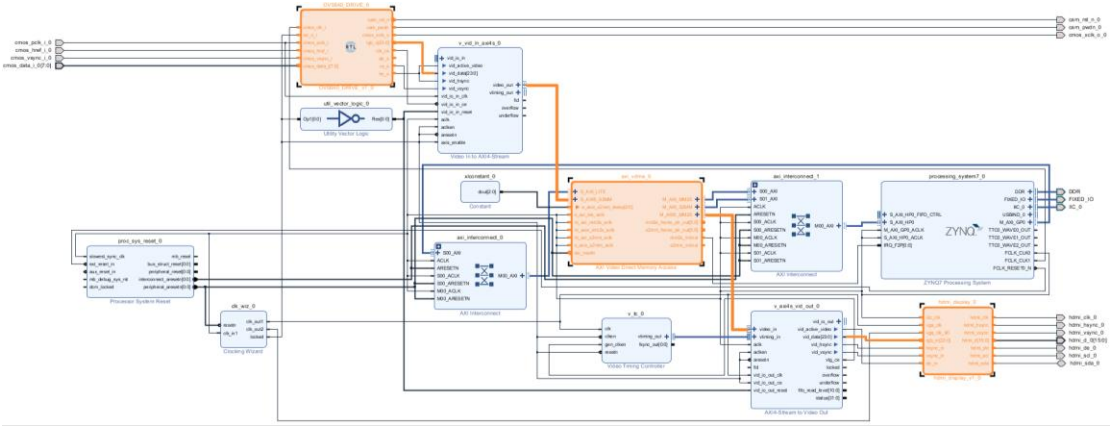

图 5-2 视频采集-缓存-显示验证测试 Block Design

将比特流烧录入 Zedboard 后，显示器能实时地显示摄像头所采集的视频流。视频分辨率为 720p（1280x720），视频帧数稳定在 25 帧。可见摄像头和显示器能正常工作，摄像头驱动程序和 HDMI 驱动程序都能正常运行。

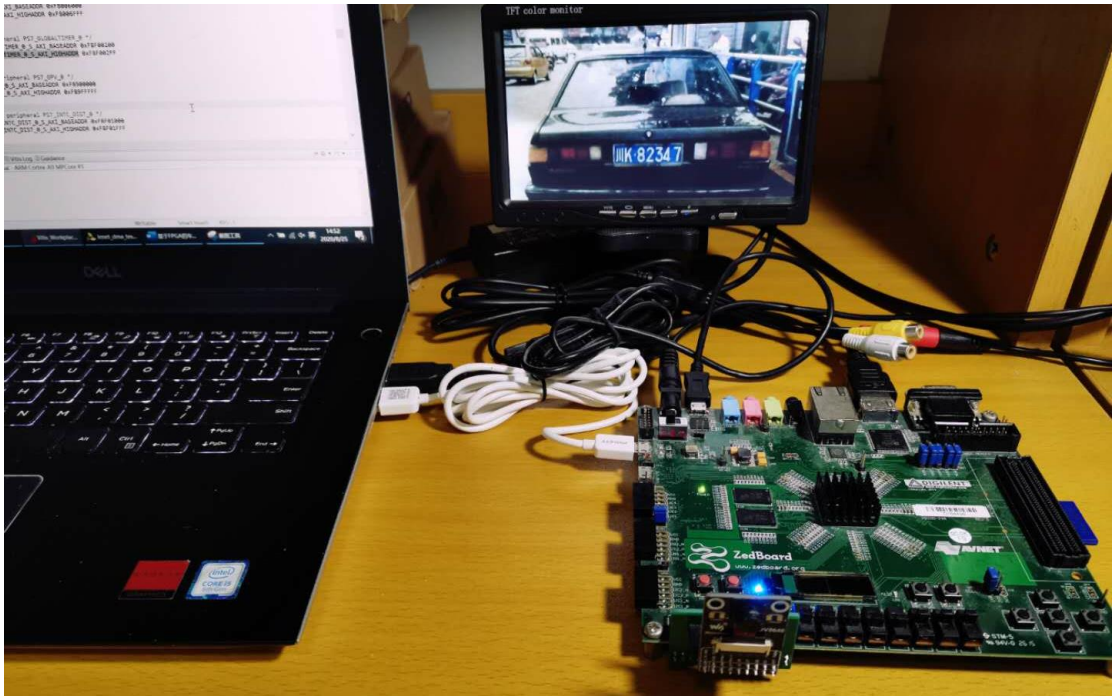

图 5-3 测试运行

4.3 视频采集-缓存-处理-显示测试

确保摄像头与显示器能够正常工作后，接下来我们对二值化处理程序进行测试。在上一个测试的 Block Design 的基础上，我们添加 HLS 图像处理 IP 核，和负责切换视频流的 Video Mixer，进行“视频采集-缓存-处理-显示”的测试。

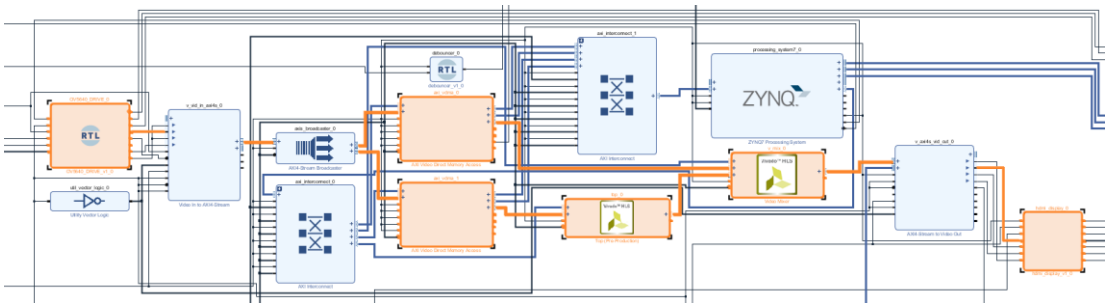

图 5-4 视频采集-缓存-处理-显示测试 Block Design

将比特流烧录入 Zedboard 后，显示器能够实时地显示经 HLS IP 核处理后得到的二值图像。视频分辨率和帧数与一致。通过按下按键，可实现处理前和处理后的视频切换。可见 HLS 图像处理 IP 核与 Video Mixer 皆能正常工作。

图 5-5 按键切换效果示意图

4.4 车牌区域检测效果测试

我们开始对在 ARM 端实现的广度搜索算法进行测试。我们将含有车牌的图片在摄像头前以一定速率移动，使用广度搜索检测出来的的车牌区域限定框能够无延时地与车牌区域一同移动。可见广度搜索寻找车牌区域的算法速度较快，准确度较高，车牌区域检测效果较好。

图 5-6 广度搜索测试

4.5 车牌字符切割效果测试

确保车牌区域检测无误后，我们开始对车牌的上下界进行定位并将车牌切割为单个字符。为了观察字符切割效果，我们将切割后的字符通过串口逐个逐个像素地输出至电脑的串口调试助手，并使用 Python 将接收到的字符数组转成 32x32 的灰度图，结果如下。可见算法能够正确地完成车牌字符切割。

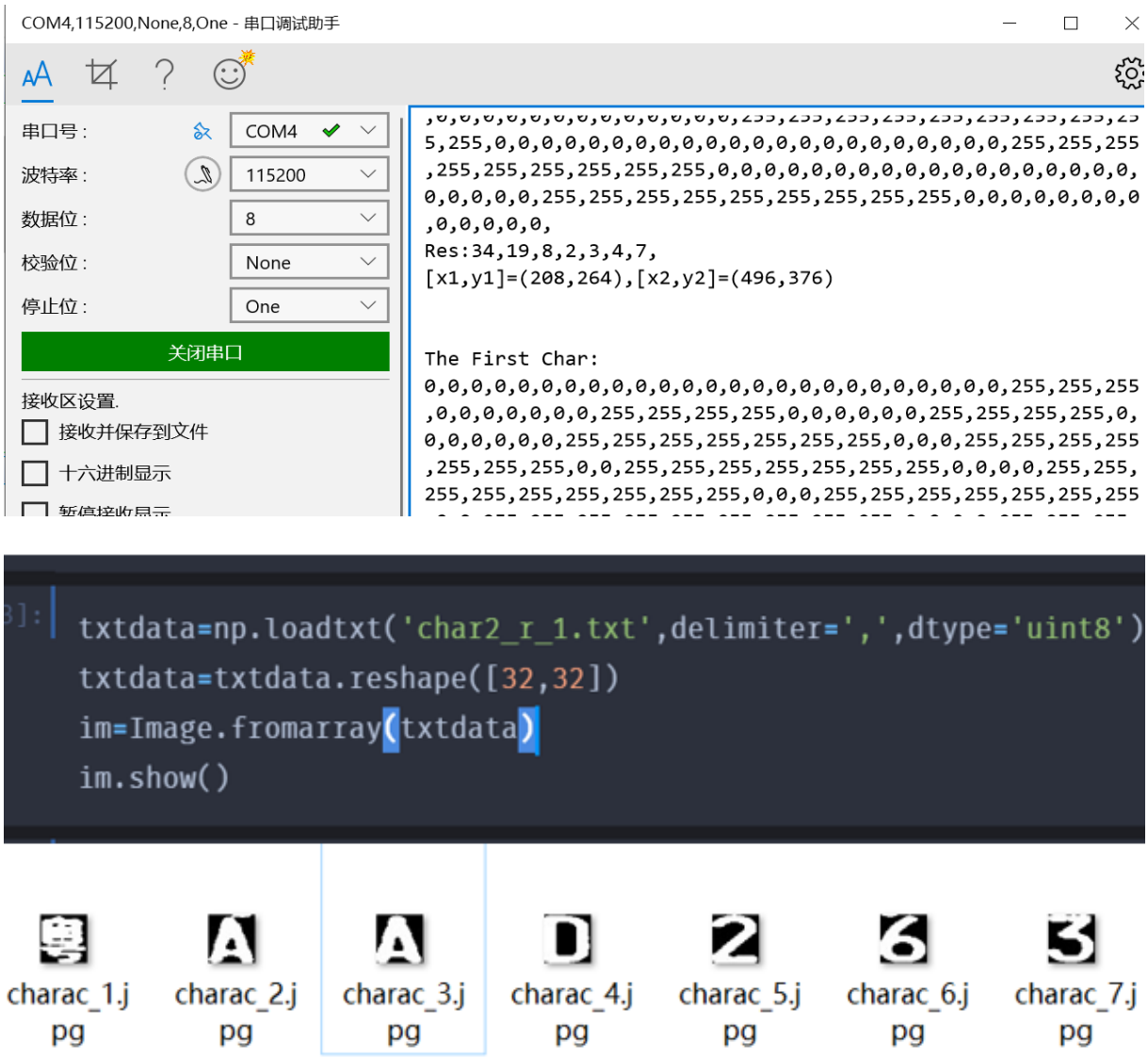

图 5-7 字符切割测试

4.6 车牌字符识别正确率测试

将切割得到的字符传送至 LeNet 识别之前，我们要先验证 LeNet IP 核能够正常完成识别功能。

负责车牌字符识别的 LeNet IP 核较为独立，不依赖于先前的图像处理过程，PS 端仅需将测试数据通过 DMA 传送至 LeNet IP 核，等待其识别完毕后，获取识别结果，将识别结果与正确结果对比，即可验证 LeNet IP 核的识别的正确性。因此，我们单独构建 Zynq-DMA-LeNet-Zynq 这一数据环路，对 LeNet IP 核进行验证。

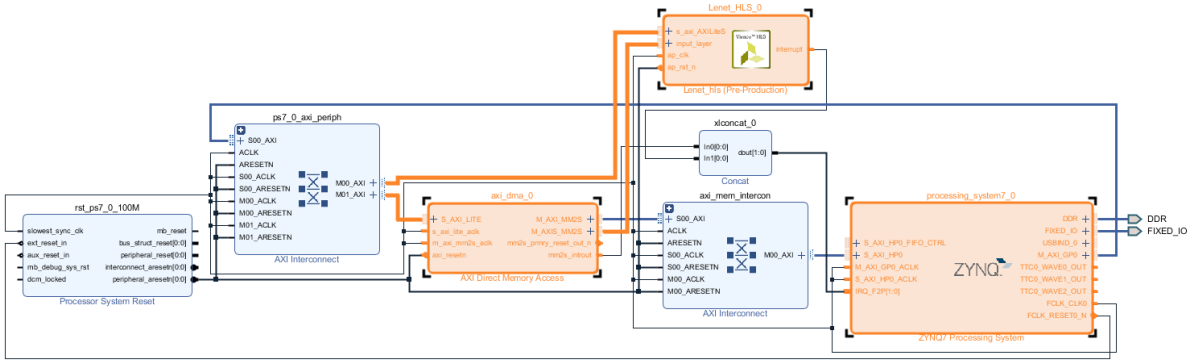

图 5-8 数据环路

我们通过串口助手进行测试，测试结果如下。识别结果与正确结果移植，证明 LeNet IP 核能够正常工作。

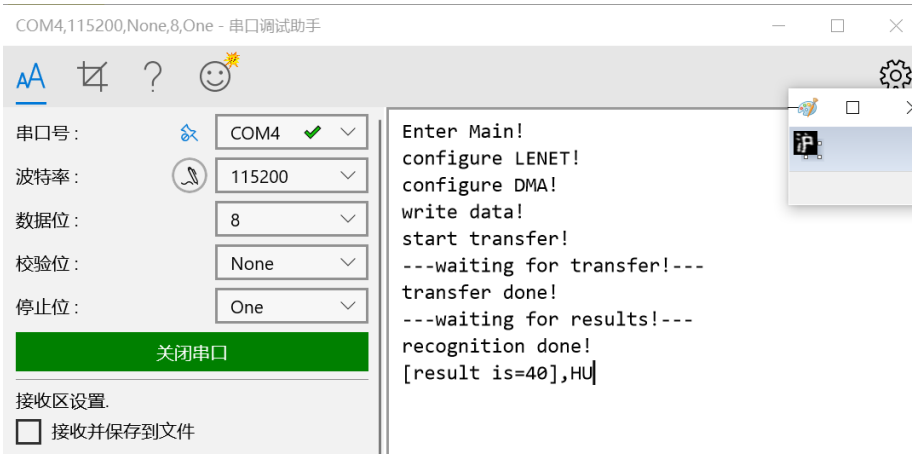

图 5-9 车牌识别正确性测试

4.7 整体测试

系统的各个功能模块测试完后，我们进行最终的系统整体测试，整体的 Block Design 如下。

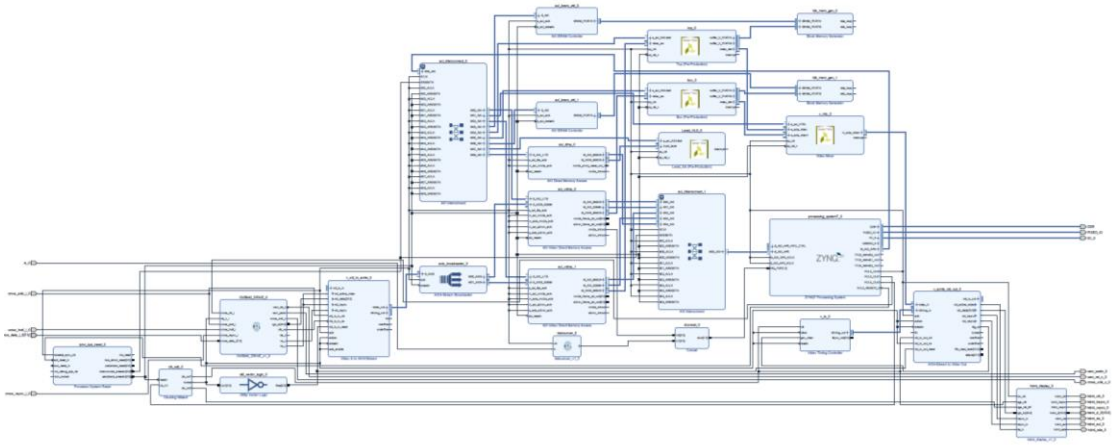

图 5-10 整体 Block Design

保持硬件接线不变，将 Zedboard 复位后，将整体系统工程的比特流烧录至 Zedboard 后，可见 Zedboard 上蓝灯亮起，说明烧录成功。可见 OLED 显示“Enter Main“，说明 PS 端程序进入主函数。电脑的串口输出如下，表明各个功能模块的初始化设置成功。

图 5-11 初始化成功

烧录程序后，显示器默认显示经过处理后的视频流。此时正在运行的功能模块有：摄像头视频采集，PL 端 HLS IP 核图像处理，PS 端部分的图像处理（广度搜索获得车牌区域坐标），与 HDMI 显示器显示。

运行结果如下图所示。处理后的二值图像经过广度搜索后筛选出的坐标通过串口在电脑上不断地输出显示，坐标所限定的 Bounding Box 在显示屏上亦随着车牌而移动。

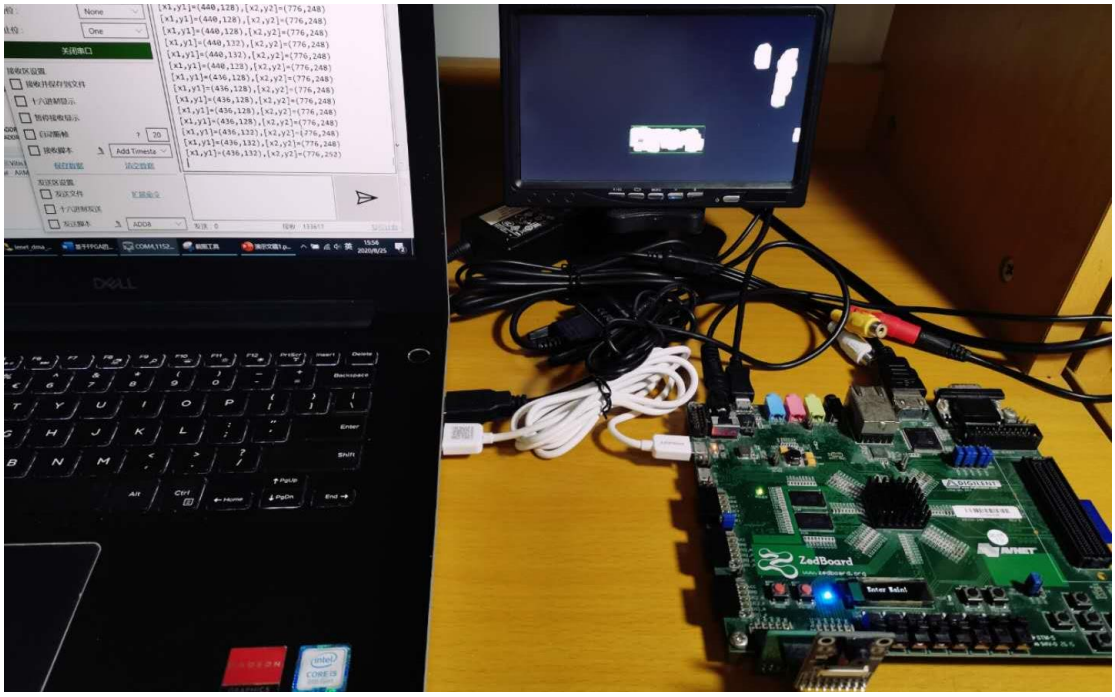

图 5-12 车牌定位

按下视频切换按钮，显示器显示处理前的视频流。此时正在运行的功能模块有：摄像头视频采集，PL 端 HLS IP 核图像处理，PS 端完整的图像处理（广度搜索获得车牌区域坐标，车牌上下界定位，字符切割），车牌字符识别，OLED 显示车牌识别结果，与 HDMI 显示器显示。

运行结果如下图所示。显示器显示处理前的图像，Bounding Box 将车牌区域框起，电脑串口助手不断显示 Bounding Box 的坐标与车牌字符识别结果，OLED 亦将识别结果不断地打印出来。

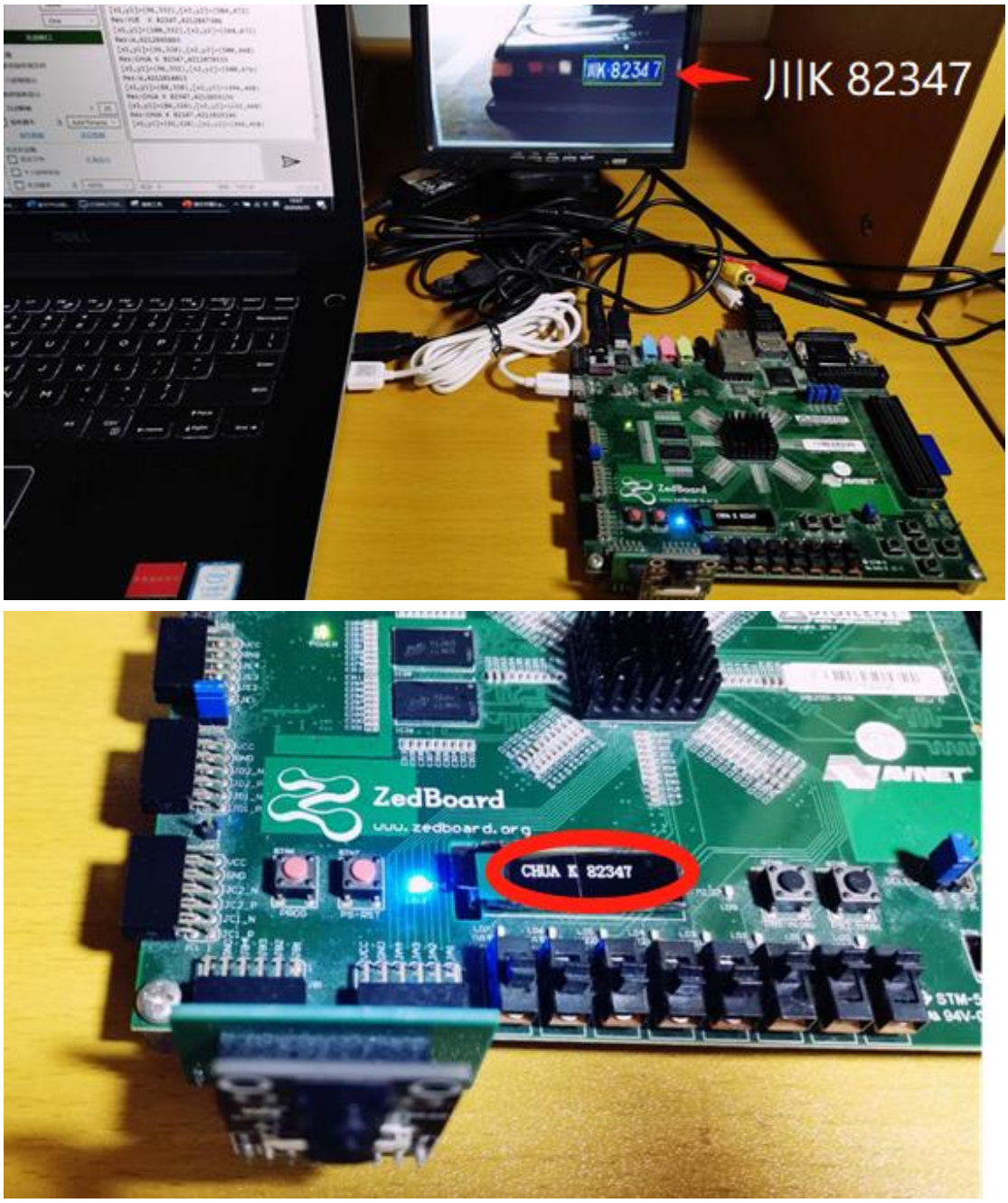

图 5-13 车牌识别结果测试

使用更多的车牌图片测试，亦能实现检测，定位，识别的功能，且结果准确。鉴于篇幅所限，下面仅放出 2 张车牌与 OLED 所显示的识别结果。可见系统皆能将车牌字符正确地识别。

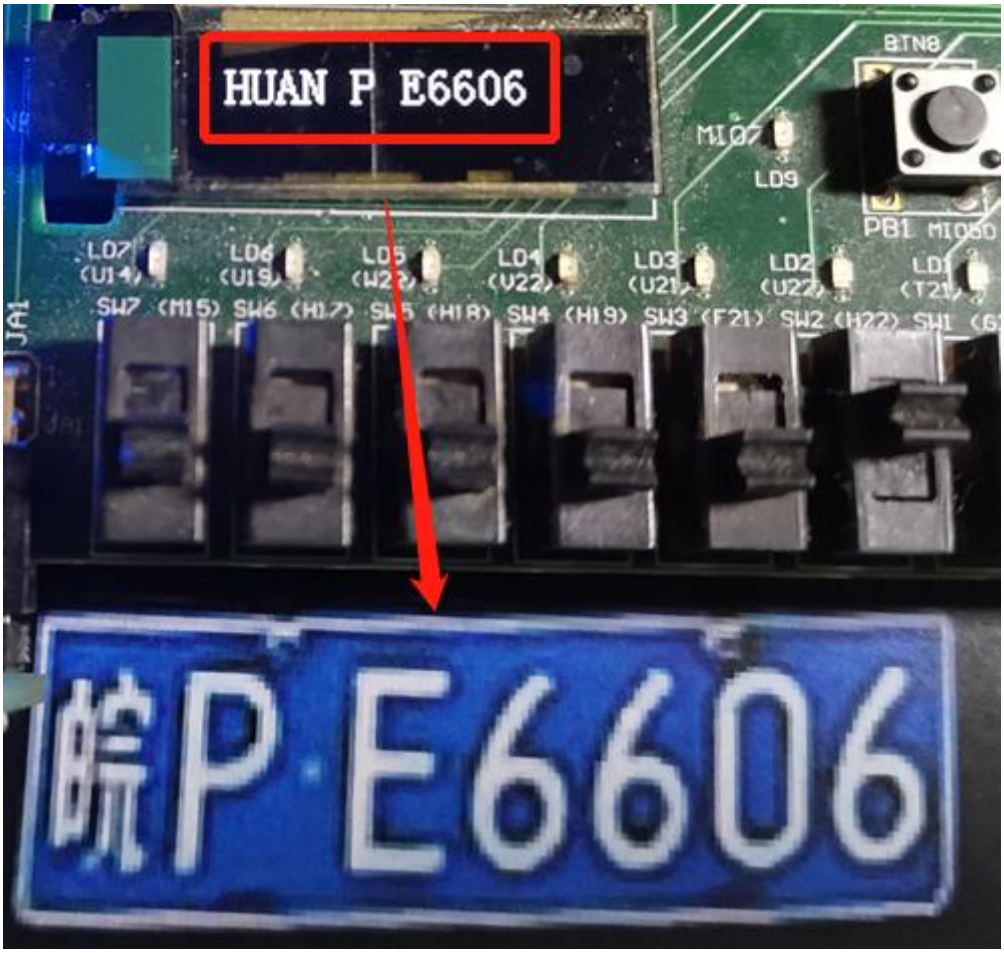

图 5-14 车牌识别结果测试

至此，系统的整体测试完毕。

5 项目创新点

5.1 使用了 FPGA 搭建车牌识别系统

传统的图像处理算法与神经网络推理往往会在 CPU 或者是 GPU 上进行实现。而高性能的 CPU 或 GPU 较为昂贵且功耗较高。在 FPGA 上进行图像处理与神经网络推理的难度往往会比在 CPU 或 GPU 实现高，但它的功耗低，而且速度能与 GPU 相当，特别适用于轻量级的系统。我们使用 FPGA 实现车牌识别系统，是对未来实现更加强大的系统的展望。

5.2 采用轻量级图像处理算法完成车牌定位与字符分割

我们采取了基于色彩分割的二值化处理方法，基于广度搜索的车牌区域定位方法，与基于垂直分割的字符分割完成了车牌照片的图像处理，没有使用复杂的定位分割方法，计算量较小，实现较简单且效果较好，满足项目的轻量级，低功耗的定位要求。

5.3 采用了 FPGA+ARM 的方式完成图像处理算法

单独使用 FPGA 或 ARM 进行图像处理各自都有优缺点。我们令 FPGA 和 ARM 相互配合，发挥处理器各自的优势，完成车牌图像处理的功能，使得 Zynq 系列芯片的优势最大化。

5.4 采用高层次综合的方法实现 LeNet 神经网络的推理

使用 HLS 进行神经网络推理具有开创性。神经网络的卷积层，池化层与全连接层的推理计算并没有现成的库函数可调用。使用 HLS 手动实现卷积层，池化层，全连接层，并使用数据量化与并行加速的方式进行优化，需要经过复杂的多次的测试与验证。这也是本项目最大的亮点。

6 总结与期望

使用 Zedboard 实现车牌识别系统的过程中，我们遇到了诸多困难。基于 FPGA 的图像处理+神经网络的现成项目较少，可供参考的资源十分有限；Zedboard 开发板可用硬件资源较少，对进行更高层次的优化提出了挑战；HLS 的图像处理可用库函数较少，我们需自行实现其他函数，以完成更加复杂的功能；基于 HLS 的神经网络推理需要手动实现，使神经网络的复杂度受到一定限制……但是，我们在克服了诸多困难，依旧完成了该系统，并且取得了较好的成果。

当然，该车牌识别系统依旧具有很大的提升空间。在借助更加强大的开发工具和硬件平台下，我们能逐步完善当前项目的不足之处。2020 年 6 月 Xilinx 公司推出 Vitis 2020，同时推出 Vitis Vision 和 Vitis AI，为实现更复杂的图像处理，推理更深层次的神经网络提供了更高效易用的开发工具；使用功能更强大的开发板，如 Digilent 推出的基于 Zynq Ultrasacle 芯片的 Genesys ZU 开发板，能得到更好的性能。若使用 Vitis2020 软件平台与 Genesys ZU 硬件平台，我们的车牌识别系统的性能将获得显著提升，功能也会更加完善。

虽然本项目只是车牌识别这一较为普遍的边缘智能应用，但通过打通算法设计-软硬件任务分配-硬件实现与优化的研发链条，我们已经掌握了一整套方法和工具，为后续开展自动驾驶相关的目标识别与跟踪等更高复杂度深度学习相关应用奠定了扎实的基础。我们所探索的高层次综合方案，也为深度学习算法在 FPGA 上的便捷实现提供了切实可行的实现路径。