第 10 章 系统集成

止于至善 《礼记·大学》

本章重点介绍基于卷积神经网络（BNN）的交通标志识别系统在 ARM+FPGA 异构平台上的系统集成，实现交通标志识别加速目的。

第 10.1 节 道路交通标志识别系统

道路交通标志是现在交通重要设施[50]，用文字或符号传递引导、限制、警告或指示信息的道路设施。道路交通标志分为主标志和辅助标志两大类。主标志又分为警告标志、禁令标志、指示标志、指路标志、旅游区标志和道路施工安全标志六类，共有 300 多种。警告标志起警告作用，警告车辆、行人注意危险地点的标志。禁令标志起到禁止某种行为的作用，禁止或限制车辆、行人交通行为的标志。指令标志起指示作用，指示车辆、行人行进的标志。指路标志起指路作用，传递道路方向、地点、距离信息的标志。旅游区标志提供旅游景点方向、距离的标志。道路施工安全标志是通告道路施工区通行的标志，用以提醒车辆驾驶人和行人注意。辅助标志是在主标志无法完整表达或指示其内容时，为维护行车安全与交通畅通而设置的标志，附设在主标志下，起辅助说明作用。

在无人驾驶或辅助驾驶中，车辆能自动及时正确地识别道路交通标志是非常重要的。文献[40]在 PYNQ-Z1 开发板上使用 ARM 和 FPGA 协同设计实现了一个基于卷积神经网络的道路交通标志自动识别系统，实验表明该自动识别系统识别一张交通标志用时 49463 微秒，每秒可识别图片 20.22 张；而完全使用软件实现识别用时 812910 微秒，每秒可识别图片 1.23 张。前者识别速度是后者的近 17 倍。

本章主要介绍文献[40]的工作。

第 10.2 节 系统建模

本节介绍交通标志识别系统的自动机模型和 SysML 模型，以及系统资源约束。

10.2.1 系统概述

交通标志识别系统的输入为交通标志图片，经过分类器识别输出该图片所表示的标志含义。系统可识别限速、停止、禁止左转、注意行人等 43 种常用交通标志。交通标志识别系统共包括四部分：图片输入、图片预处理、二值神经网络（BNN）[51]分类决策和识别结果输出。图片输入用于从用户端获取要识别的标志图片。图片预处理将输入的图片格式转换成与分类器匹配的格式（系统采用 Cifar-10 数据库中的图片格式）。BNN 分类决策是系统重点部分，负责标志的识别。识别结果输出则将结果返回给用户。

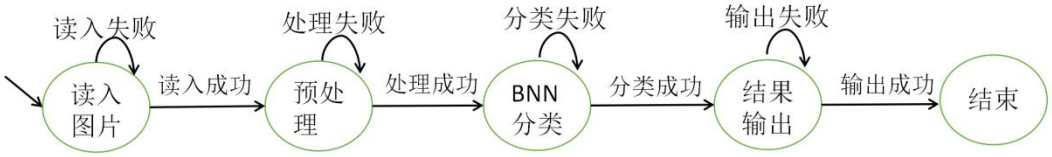

图 10-1 交通标志识别系统的功能有限自动机

在传统的图像识别中，特征提取和特征分类是分开的两个步骤；而使用 CNN 进行图像识别时，则无需专门设计特征提取算子，特征提取会在网络训练过程中自动学习。然而，卷积神经网络比传统图像识别分类方法需要更多的计算资源和更多的存储空间，大量的浮点数计算严重阻碍了其在移动设备上的应用。因此系统选用 BNN 分类器进行标志识别，将卷积神经网络二值化，使得计算主要在+1 和 -1 间进行，将大量数学计算变为位操作，降低了网络大小和计算量。

10.2.2 SysML 建模

依据 10.2.1 节的系统概述，使用 SysML 对系统进行建模，系统模块定义图如图 10-3 所示。

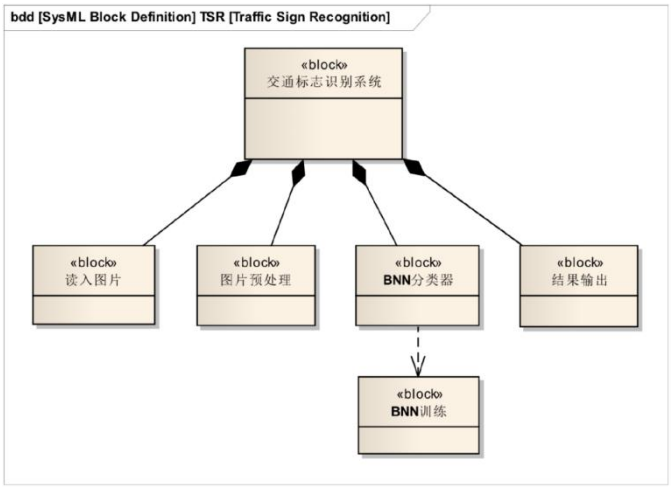

图 10-2 交通标志识别系统模块定义图

模块定义图定义了该系统由 4 个模块组成，其中 BNN 分类器通过事先的 BNN 训练得到。

序列图（图 10-3）则展示了各个对象之间的时序交互关系。

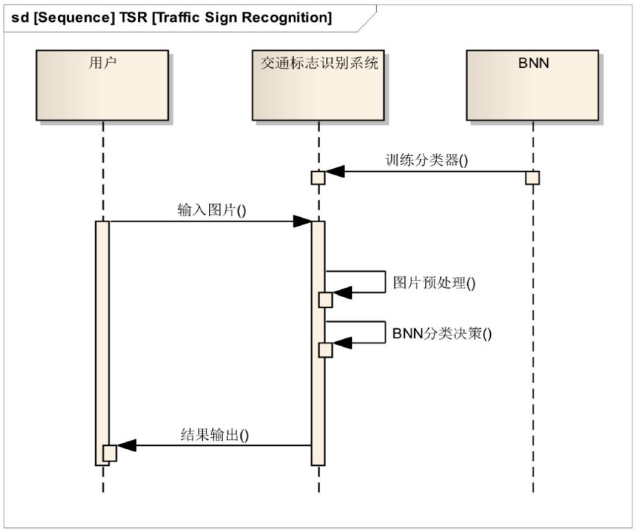

图 10-3 交通标志识别系统序列图

10.2.3 系统资源约束

当交通标志识别系统被用于辅助驾驶或自动驾驶时，由于车辆在运动中速度较快，需要及时识别标志进行相应操作，对交通标志识别系统的时间性能要求很高。因此，使用 FPGA 对交通标志识别系统进行硬件加速。

系统采用 PYNQ-Z1 开发板进行开发，该开发板上集成了 ZYNQ-7020 SoC 器件，还给用户提供了丰富的硬件外设接口，具体参数如表 10-1 所示。

从表中可以看出该设备中查找表 LUT 数量为 53200 个，因此系统要求硬件部分所使用的 LUT 数量不超过 53200 个。

表 10-1 PYNQ-Z1 开发板参数

参数	配置
尺寸	87mm x 122mm
处理器	双核 ARM Cortex A9
FPGA	53200 个 LUT 130 万个可重配置门电路
内存	512M DDR3/FLASH
存储	支持 Micro SD 卡

接口	HDMI 输入输出接口 音频输入输入接口 千兆以太网接口 USB OTG 接口 Arduino 和 Pmod 等
其他	LEDx6、按键 x4、开关 x2

第 10.3 节 任务属性

本节介绍道路交通标志识别系统的软件时间属性和硬件时间属性，以及 FPGA 查找表属性指标，最后使用第 5 章多目标划分方法给出软硬件划分结果。

10.3.1 任务提取

系统中图片输入、预处理和识别结果输出这三个过程操作十分简单，使用软件就可以很容易实现，因此只需考虑将较为复杂、耗时较长的 BNN 分类决策部分采用硬件加速。

首先将 BNN 分类器整体使用硬件实现，对 C++代码使用 Vivado HLS 软件进行综合。BNN 分类器整体的综合结果如图 10-4 所示。

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	-	-	-	-
Instance	43	52	69569	65232
Memory	189	-	3104	416
Multiplexer	-	-	-	6939
Register	-	-	319	-
Total	232	52	72992	72589
Available	280	220	106400	53200
Utilization (%)	82	23	68	136

图 10-4 BNN 硬件综合结果

从综合结果可知所需 LUT 数量为 72589 个，使用率为 136%，超过了资源数量 53200（如果换成 LUT 数量更多的开发板，会增加实现成本）。因此需要将 BNN 进行任务划分，选择其中部分任务使用硬件实现。

系统所用 BNN 层次结构如图 10-5 所示，包括 6 个卷积层、2 个池化层和 3 个全连接层。其中第 1、2、4、5、7、8 层为卷积层，用来进行特征提取；第 3、6 层池化层对输入的特征图进行压缩，一方面使特征图变小，简化网络计算复杂度，另一方面进行特征压缩，提取主要特征；因此卷积层和池化层相连即可完成一次主要特征提取。最后三层全连接层则连接所有的特征，计算每类别对应的得分。我们根据层次功能，把每次特征提取看做一个任务，分别称为特征提取 1、特征提取 2 和特征提取 3，再将最后 3 个全连接层看做一个整体，整个 BNN 分类器被分成了四个任务。将对这四个任务进行软硬件划分。

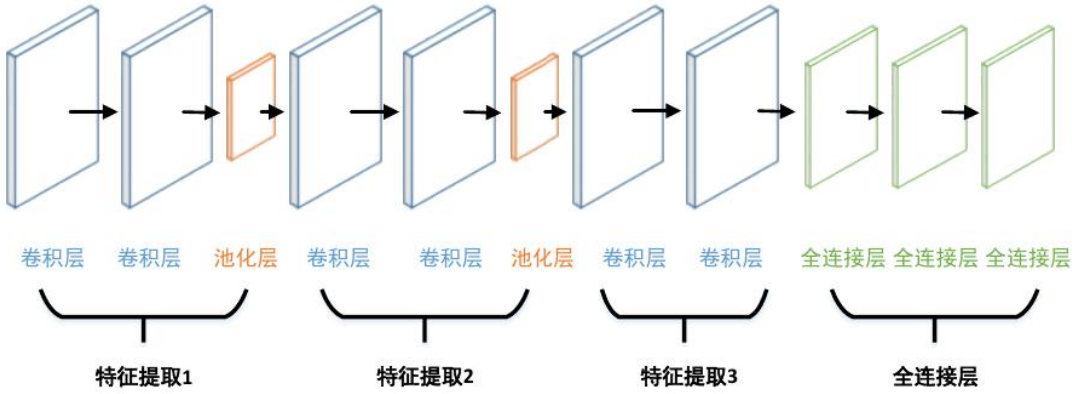

图 10-5 BNN 层次结构图

10.3.2 任务性能获取

任务软件执行时间属性获取方式为：将任务使用 C++实现，并加上时间戳，运行代码，获取该任务的一次软件执行时间，多次运行结果取平均值得到最终任务的软件执行时间。运行环境对软件的执行时间有着很大影响，考虑到最终系统集成后软件部分在 ARM 核中运行，为了保证数据准确有效并具有参考价值，此处的运行环境与系统集成后环境一致，即任务的软件执行时间为该任务在 ARM 核中运行所需时间。经执行计算得到四个任务所需软件执行时间分别为：T_1：534910 μs，T_2：240385 μs，T_3：32629 μs，T_4：9467 μs。

任务硬件执行时间属性获取方式为：将任务在 Vivado HLS 中对 C++代码进行综合，从综合结果报告中可得到每一部分的硬件执行时间。通过估算得到四个任务所需硬件执行时间分别为：T_1：7293 μs，T_2：4018 μs，T_3：514 μs，T_4：377 μs。

在系统中，任务所占硬件面积以所占用 FPGA 中的 LUT 数量为度量单位。分

析 Vivado HLS 综合结果则可以得到估算的每一层所用 LUT 资源数。72589 个 LUT 约有 6000 个用于存储、复用器或总线控制等，其他 LUT 则用于 BNN 分类器的主体部分。该部分每个函数调用实例所占用 LUT 情况图（部分）如图 10-6 所示。

⊟ Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
DoCompute_Block_pro_U0	DoCompute_Block_pro	0	0	68	89
DoCompute_Block_pro_1_U0	DoCompute_Block_pro_1	0	4	287	71
DoCompute_Block_pro_2_U0	DoCompute_Block_pro_2	0	0	64	89
DoCompute_Block_pro_3_U0	DoCompute_Block_pro_3	0	4	287	71
DoCompute_Block_pro_4_U0	DoCompute_Block_pro_4	0	4	287	71
DoCompute_entry34512_U0	DoCompute_entry34512	0	0	3	92
Mem2Stream_Batch_U0	Mem2Stream_Batch	0	0	567	918
Stream2Mem_Batch_U0	Stream2Mem_Batch	0	0	557	959
StreamingConvolution_U0	StreamingConvolution	8	4	1159	1473
StreamingConvolution_1_U0	StreamingConvolution_1	0	4	1649	1518
StreamingConvolution_2_U0	StreamingConvolution_2	0	0	1050	1549
StreamingConvolution_3_U0	StreamingConvolution_3	0	0	3299	1510
StreamingConvolution_4_U0	StreamingConvolution_4	0	0	2037	1513
StreamingConvolution_5_U0	StreamingConvolution_5	16	4	1262	1457
StreamingDataWidthCo_U0	StreamingDataWidthCo	0	0	221	303
StreamingDataWidthCo_1_U0	StreamingDataWidthCo_1	0	0	67	155
StreamingDataWidthCo_10_U0	StreamingDataWidthCo_10	0	0	161	239
StreamingDataWidthCo_11_U0	StreamingDataWidthCo_11	0	0	161	239
StreamingDataWidthCo_12_U0	StreamingDataWidthCo_12	0	0	354	239

图 10-6 LUT 详细占用情况（部分）

图中前 6 个函数调用实例（DoCompute_*）用于接口定义，接下来两个函数 Mem2Stream_*和 Stream2Mem_*用于数据流转换，这两部分不属于 BNN 的任何一层，因此不计算在内。根据剩下的函数调用情况可以得到每一层、每一个任务的 LUT 使用数量。例如，StreamingConvolution_U0 在第 1-6 个卷积层中被调用，则第 1 个卷积层中 LUT 数据计入 1473，在第 2 个卷积层中 LUT 数据计入 1518，第 3 个卷积层中 LUT 数据计入 1549，等等；StreamingDataWidthCo_U0 在全链接层被调用，每调用一次就会产生 LUT 使用，因而把这些 LUT 数统计到全连接层中。然后根据层数与任务的对应关系计算出每个任务所使用的 LUT 数量。

最终得到基于 BNN 的交通标志识别系统任务属性表如表 10-2 所示。

表 10-2 BNN 分类器任务属性表

任务	软件执行时间	硬件执行时间	硬件面积	备注
T_1	534,910 μs	7,293 μs	24,956 个 LUT	特征提取 1
T_2	240,385 μs	4,018 μs	21,566 个 LUT	特征提取 2
T_3	32,629 μs	514 μs	9,045 个 LUT	特征提取 3
T_4	9,467 μs	377 μs	4,639 个 LUT	全连接层

在目标开发板上，处理器核（ARM）与硬件部分（FPGA）通过高速 AXI 总线进行数据传输，由于传输速度很快，传输时间在微秒级别内无法记录，因此任务间通信时间忽略不计。最终得到图 10-7 所示 BNN 分类器的任务图，其中节点的三权值分别为软件执行时间、硬件执行时间和硬件 LUT 数。

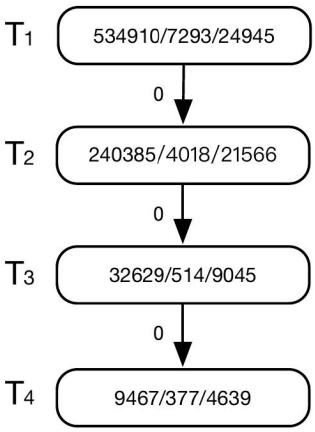

图 10-7 BNN 分类器任务图

10.3.3 软硬件划分

对 BNN 分类器中的任务进行软硬件划分。任务图由上一小节得到，预留 6000 个 LUT 用于存储、复用器及总线控制，则剩余的 47200 个 LUT 为系统硬件面积约束。使用第 5 章多目标划分方法对 BNN 分类器的四个任务进行软硬件划分：将时间作为总体极小化目标，约束条件是 LUT 数量不超过 47200 个。划分结果为硬件实现 T1 和 T2，软件实现为 T3 和 T4，LUT 数量使用了 46522 个，时间为 53407 μs。因此，BNN 分类器中特征提取 1 和特征提取 2 需要使用硬件进行加速，特征提取 3 和全连接层则由软件实现。虽然目标开发板提供了双核处理器，但由于几个任务顺序执行，所以此处并不涉及多个处理器核上的调度问题。再加上图片读入和预处理等任务，最终完整的系统软硬件划分结果如图 10-8 所示。

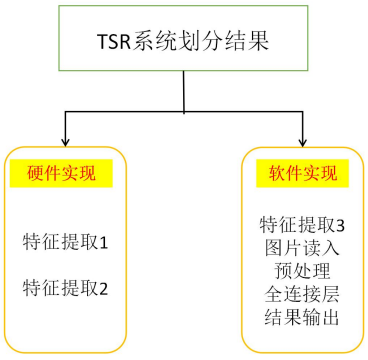

图 10-8 基于 BNN 的交通标志识别系统软硬件划分结果

第 10.4 节 软硬件综合

本节介绍在 Vivado HLS 平台上实现软硬件综合。软件综合使用 C++和 Python 语言实现，硬件综合在 Vivado HLS 平台上进行，只是要选取合适的开发板型号。利用测试文件对综合进行测试，并进行 C/RTL 的协同仿真，验证设计的功能正确性。

10.4.1 软件综合

软件综合需选取合适的语言对软件任务进行实现。首先，对于特征提取 3 和全连接层这两个任务，由于在之前获取任务属性时该部分已经使用 C++代码实现并进行过测试，为了提高代码复用性，减少工作量，此部分的软件实现沿用 C++实现。

接下来考虑其他三个软件任务即图片读入、预处理、识别结果输出的实现。Python 中的 Python Image Library 有着强大的图片处理能力，可以满足图片读入和预处理的需求，并且 Python 可以直接调用 C++语言，再考虑到系统实现所选取的 PYNQ-Z1 开发板顶层支持 Python 语言，因此 Python 是实现这三个任务的最佳语言。

10.4.2 硬件综合

为了用硬件实现特征提取 1 和特征提取 2 这两个任务，需要创建相应 IP 核，这一过程仍然在 Vivado HLS 平台中进行。

为了创建所需要的 IP 核，需要在 Vivado HLS 中创建一个工程，导入实现特征提取 1 和特征提取 2 功能的 C++代码并指定综合时的顶层函数，此处仍然复用之前综合时使用的代码，然后导入为该部分重新设计的测试文件，并选择开发板型号为“xc7z020clg400-1”（与目标开发板对应）。

顶层函数代码为：

```
void BlackBoxJam(ap_uint<64> * in, ap_uint<128> * out_inter6, bool doInit, unsigned int targetLayer, unsigned int targetMem, unsigned int targetInd, ap_uint<64> val, unsigned int numReps) {

#pragma HLS RESOURCE variable=thresMem1 core=RAM_S2P_LUTRAM
#pragma HLS RESOURCE variable=thresMem2 core=RAM_S2P_LUTRAM
#pragma HLS RESOURCE variable=thresMem3 core=RAM_S2P_LUTRAM
```

```
    // pragmas for MLBP jam interface
    // signals to be mapped to the AXI Lite slave port
    #pragma HLS INTERFACE s_axilite port=return bundle=control
    #pragma HLS INTERFACE s_axilite port=doInit bundle=control
    #pragma HLS INTERFACE s_axilite port=targetLayer bundle=control
    #pragma HLS INTERFACE s_axilite port=targetMem bundle=control
    #pragma HLS INTERFACE s_axilite port=targetInd bundle=control
    #pragma HLS INTERFACE s_axilite port=val bundle=control
    #pragma HLS INTERFACE s_axilite port=numReps bundle=control
    // signals to be mapped to the AXI master port (hostmem)
    #pragma HLS INTERFACE m_axi offset=slave port=in bundle=hostmem depth=256
    #pragma HLS INTERFACE s_axilite port=in bundle=control
    #pragma HLS INTERFACE m_axi offset=slave port=out_inter6 bundle=hostmem
depth=256
    #pragma HLS INTERFACE s_axilite port=out_inter6 bundle=control

    // partition PE arrays
    #pragma HLS ARRAY_PARTITION variable=weightMem0 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=thresMem0 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=weightMem1 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=thresMem1 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=weightMem2 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=thresMem2 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=weightMem3 complete dim=1
    #pragma HLS ARRAY_PARTITION variable=thresMem3 complete dim=1

        if (doInit) {
            DoMemInit(targetLayer, targetMem, targetInd, val);
        } else {
            DoCompute(in, out_inter6, numReps);
        }
    }

    //特征提取 1，特征提取 2
    void DoCompute(ap_uint<64> * in, ap_uint<128> * out_inter6, const unsigned int
numReps) {
    #pragma HLS DATAFLOW

    stream<ap_uint<64> > inter0("DoCompute.inter0");
    stream<ap_uint<192> > inter0_1("DoCompute.inter0_1");
    stream<ap_uint<24> > inter0_2("DoCompute.inter0_2");
    #pragma HLS STREAM variable=inter0_2 depth=128
    stream<ap_uint<64> > inter1("DoCompute.inter1");
    #pragma HLS STREAM variable=inter1 depth=128
```

```
    stream<ap_uint<64> > inter2("DoCompute.inter2");
    stream<ap_uint<64> > inter3("DoCompute.inter3");
    #pragma HLS STREAM variable=inter3 depth=128
    stream<ap_uint<128> > inter4("DoCompute.inter4");
    #pragma HLS STREAM variable=inter4 depth=128
    stream<ap_uint<128> > inter5("DoCompute.inter5");
    stream<ap_uint<128> > inter6("DoCompute.inter6");
    #pragma HLS STREAM variable=inter6 depth=81

    const unsigned int inBits = 32*32*3*8;
    const unsigned int outBits = L6_MH*16;

    Mem2Stream_Batch<64, inBits/8>(in, inter0, numReps);
    StreamingDataWidthConverter_Batch<64, 192, (32*32*3*8) / 64>(inter0, inter0_1,
numReps);
    StreamingDataWidthConverter_Batch<192, 24, (32*32*3*8) / 192>(inter0_1,
inter0_2, numReps);
    StreamingFxdConvLayer_Batch<L0_K, L0_IFM_CH, L0_IFM_DIM, L0_OFM_CH,
L0_OFM_DIM, 8, 1, L0_SIMD, L0_PE, 24, 16, L0_WMEM, L0_TMEM>(inter0_2, inter1,
weightMem0, thresMem0, numReps);
    StreamingConvLayer_Batch<L1_K, L1_IFM_CH, L1_IFM_DIM, L1_OFM_CH, L1_OFM_DIM,
L1_SIMD, L1_PE, 16, L1_WMEM, L1_TMEM>(inter1, inter2, weightMem1, thresMem1,
numReps);
    StreamingMaxPool_Batch<L1_OFM_DIM, 2, L1_OFM_CH>(inter2, inter3, numReps);
    StreamingConvLayer_Batch<L2_K, L2_IFM_CH, L2_IFM_DIM, L2_OFM_CH, L2_OFM_DIM,
L2_SIMD, L2_PE, 16, L2_WMEM, L2_TMEM>(inter3, inter4, weightMem2, thresMem2,
numReps);
    StreamingConvLayer_Batch<L3_K, L3_IFM_CH, L3_IFM_DIM, L3_OFM_CH, L3_OFM_DIM,
L3_SIMD, L3_PE, 16, L3_WMEM, L3_TMEM>(inter4, inter5, weightMem3, thresMem3,
numReps);
    StreamingMaxPool_Batch<L3_OFM_DIM, 2, L3_OFM_CH>(inter5, inter6, numReps);

    Stream2Mem_Batch<128, 400>(inter6, out_inter6, numReps);
    }

    //特征提取 3，全连接层
    void Con_fcl(ap_uint<128> * in_inter6, ap_uint<64> * out, const unsigned int
numReps){
    stream<ap_uint<128> > inter6("DoCompute.inter6c");
    stream<ap_uint<256> > inter7("DoCompute.inter7");
    stream<ap_uint<256> > inter8("DoCompute.inter8");
    stream<ap_uint<64> > inter9("DoCompute.inter9");
    stream<ap_uint<64> > inter10("DoCompute.inter10");
    stream<ap_uint<64> > memOutStrm("DoCompute.memOutStrm");
```

```
    const unsigned int inBits = L6_MH*16;
    const unsigned int outBits = L8_MH*16;
    Mem2Stream_Batch<128, 400>(in_inter6, inter6, numReps);

    StreamingConvLayer_Batch<L4_K, L4_IFM_CH, L4_IFM_DIM, L4_OFM_CH, L4_OFM_DIM,
L4_SIMD, L4_PE, 16, L4_WMEM, L4_TMEM>(inter6, inter7, weightMem4, thresMem4,
numReps);
    StreamingConvLayer_Batch<L5_K, L5_IFM_CH, L5_IFM_DIM, L5_OFM_CH, L5_OFM_DIM,
L5_SIMD, L5_PE, 16, L5_WMEM, L5_TMEM>(inter7, inter8, weightMem5, thresMem5,
numReps);
    StreamingFCLayer_Batch<256, 64, L6_SIMD, L6_PE, 16, L6_MW, L6_MH, L6_WMEM,
L6_TMEM>(inter8, inter9, weightMem6, thresMem6, numReps);
    StreamingFCLayer_Batch<64, 64, L7_SIMD, L7_PE, 16, L7_MW, L7_MH, L7_WMEM,
L7_TMEM>(inter9, inter10, weightMem7, thresMem7, numReps);

    StreamingFCLayer_NoActivation_Batch<64, 64, L8_SIMD, L8_PE, 16, L8_MW, L8_MH,
L8_WMEM>(inter10, memOutStrm, weightMem8, numReps);

    Stream2Mem_Batch<64, outBits/8>(memOutStrm, out, numReps);
    }
```

注： Mem2Stream_Batch 与 Stream2Mem_Batch 出现在文件 dma.h 中，StreamingDataWidthConverter_Batch 出现在文件 streamtools.h 中。

创建工程之后利用测试文件进行 C++仿真，测得仿真结果与预期相符。之后进行 C 综合，同样可以得到资源利用情况，此部分综合结果如图 10-9 所示，此时 LUT 使用情况已满足数量限制，且 BRAM、DSP、FF 等资源的使用数量均有所减少。综合过程中还会自动生成 RTL 设计即相应的 VHDL 和 Verilog 代码；综合完成后可进行 C/RTL 协同仿真，验证设计的功能正确性；验证完成之后，即可将 RTL 封装成可重用 IP 核导出，随后可导入到 Vivado IP Catalog 中用于之后的实现。

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8
FIFO	-	-	-	-
Instance	30	24	24881	46151
Memory	32	-	1888	96
Multiplexer	-	-	-	3813
Register	-	-	319	-
Total	62	24	27088	50068
Available	280	220	106400	53200
Utilization (%)	22	10	25	94

图 10-9 硬件综合结果

第 10.5 节 系统实现

在 ZYNQ-7020 板子上进行道路交通标志自动识别系统实现。实验结果表明该系统能正确地识别交通标志的含义，识别速度比纯软件识别速度快近 17 倍。

10.5.1 实现环境

系统使用 PYNQ-Z1 开发板进行开发。PYNQ-Z1 由 Digilent 公司推出，基于 Xilinx ZYNQ-7020，并在其基础上添加了对 Python 的支持。ZYNQ-7020 是 Xilinx 公司推出的一款可扩展处理芯片，集成了双核 ARM 处理器和 FPGA 可编程逻辑器件，旨在为视频监视、汽车驾驶员辅助以及工厂自动化等高端嵌入式应用提供所需的处理与计算性能水平。而 Python 语言本身易学易用、扩展库丰富、讨论社区活跃，借助这些特性可以有效降低 ZYNQ 嵌入式系统的开发门槛。PYNQ 将 ARM 处理器与 FPGA 器件的底层交互逻辑完全封装起来，顶层封装使用 Python，使用时只需导入对应的模块名称即可导入对应的硬件模块，进行底层到上层数据的交互或者为系统提供硬件加速。对于 PYNQ 的开发者来说，ARM 上运行着一个 Linux 系统，Python 运行在该系统上，需要硬件加速的模块被抽象为若干加速 IP，开发者可以通过运行一些简单的 Python 脚本即可使用这些 IP 核。

10.5.2 实现过程

在软、硬件综合过程中已经实现了每一个任务的功能，但各个任务没有联系起来，系统功能还不能完整实现，在最后实现阶段要将每个任务模块集成起来，整个系统实现架构图如图 10-10 所示。

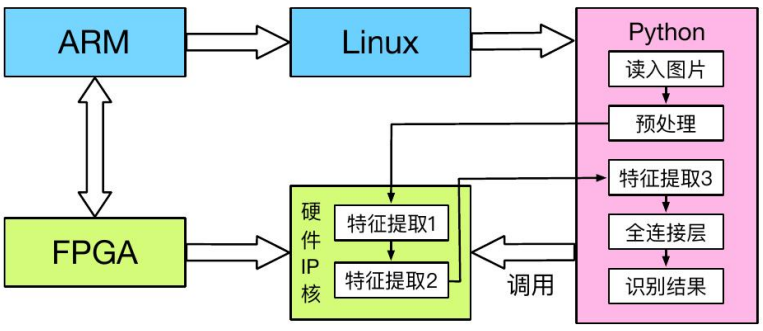

图 10-10 系统实现架构图

开发板上集成 ZYNQ-7020 设备，包括处理系统 PS 端：双核 ARM 处理器和处理逻辑 PL 端：FPGA。在 PS 端上运行着一个 Linux 系统，系统实现的顶层是运行在 Linux 系统上的 Python。最终系统运行时硬件部分在 FPGA 上运行，软件部分

则在 ARM 核中运行。

软件实现部分：：对于已经用 C++实现的代码，需先将其编译生成共享对象库（.so 文件），Python 便可以使用 CFFI（Foreign Function Interface for Python）直接进行调用。

硬件实现部分：首先要在 Vivado 集成设计环境中选择开发板创建新的工程，加入硬件部分所需 IP 核，在此过程中将之前在 Vivado HLS 中生成的 IP 核导入，IP 核的连接情况如图 10-11 所示。

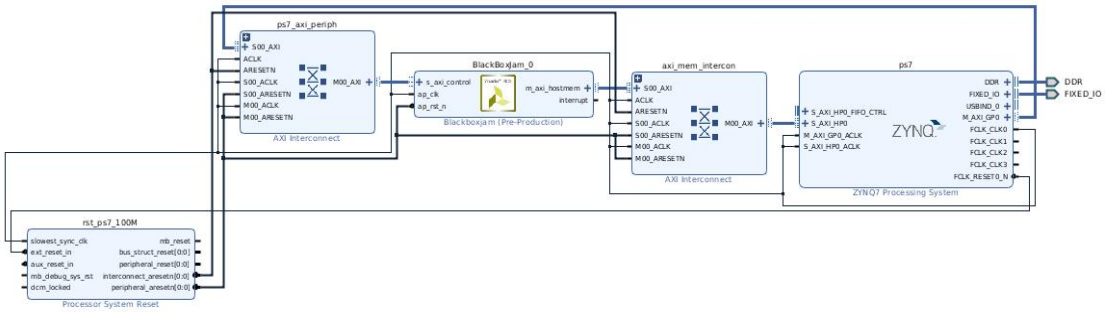

图 10-11 Vivado Block Design 中 IP 核连接

图中间的 IP 核是硬件综合后生成的 IP 核，最右边的为 ZYNQ7 Processing System IP 核，其余 IP 则用于 AXI 总线控制、内存访问和复位，这些 IP 核均由 Xilinx 提供。IP 核连接好之后可进行综合和布局布线，完成后生成二进制比特流文件。将其拷贝至开发板即可使用 Python 用 PYNQ 提供的 Overlay 类进行加载。

10.5.3 实现结果

本小节对系统实现结果进行验证。首先正确连接计算机和 PYNQ-Z1 开发板，随后在浏览器中通过开发板访问 Jupyter Notebook。Jupyter Notebook 是一个在线交互式计算环境，对 Python 有着很好的支持。在 Jupyter 中编辑 ipynb 文件调用我们写好的系统实现并运行就可以即时看到结果输出。

以一张含有停止标志的图片（图 10-12）为例，将其输入到交通标志识别系统中，对系统功能进行验证并计算执行时间。

图 10-12 测试用图片—停止标志

验证时的 Jupyter Notebook 界面如图 10-13 所示，对于输入的图片，所得到的识别结果为“Stop”，说明系统实现正确。整个识别过程用时 49463 微秒，每秒可识别图片 20.22 张。

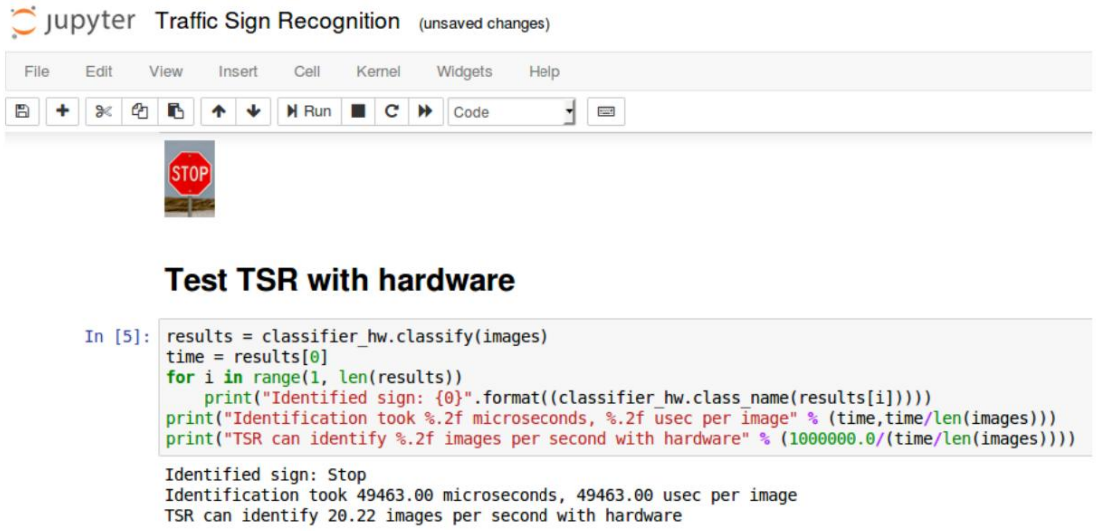

图 10-13 Jupyter Notebook 界面

为了说明软硬件协同设计对系统的加速情况，需进行对比实验，即所有任务都由软件实现，在 ARM 核上运行。实验结果如图 10-14 所示。可以看到系统完全使用软件实现识别用时 812910 微秒，每秒可识别图片 1.23 张。

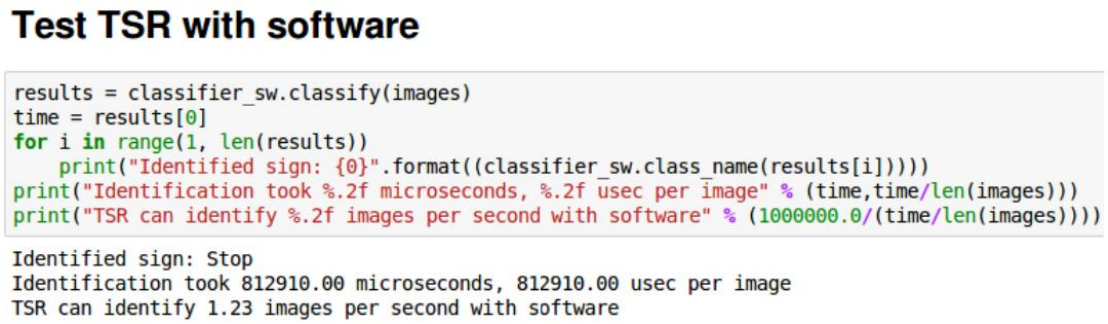

图 10-14 系统软件实现结果

再以“注意前方儿童”、“野生动物出没”、“50KM/h”几个标志作为输入进行了对比实验，结果如图 10-15 所示。

Test TSR with hardware

```
results = classifier_hw.classify(images)
time = results[0]
for i in range(1, len(results))
    print("Identified sign: {0}".format((classifier_hw.class_name(results[i]))))
print("Identification took %.2f microseconds, %.2f usec per image" % (time,time/len(images)))
print("TSR can identify %.2f images per second with hardware" % (1000000.0/(time/len(images))))
```

```
Identified sign: Children crossing ahead
Identified sign: Wild animals
Identified sign: 50 Km/h
Identification took 142478.00 microseconds, 47492.67 usec per image
TSR can identify 21.06 images per second with hardware
```

Test TSR with software

```
results = classifier_sw.classify(images)
time = results[0]
for i in range(1, len(results))
    print("Identified sign: {0}".format((classifier_sw.class_name(results[i]))))
print("Identification took %.2f microseconds, %.2f usec per image" % (time,time/len(images)))
print("TSR can identify %.2f images per second with software" % (1000000.0/(time/len(images))))
```

```
Identified sign: Children crossing ahead
Identified sign: Wild animals
Identified sign: 50 Km/h
Identification took 2459037.00 microseconds, 819679.00 usec per image
TSR can identify 1.22 images per second with software
```

图 10-15 TSR 实现对比结果

实验结果如表 10-3 所示，采取软硬件协同设计方法对基于 BNN 的交通标志识别系统进行开发、ARM 和 FPAG 协同工作时，相比于系统完全由软件实现，识别速度提升了约 17 倍。

表 10-3 软硬件实现系统速度对比

实现方式	识别单张图片用时	识别 3 张图片用时	平均每秒识别图片
ARM	812910 μs	245037 μs	1.22 张
ARM+FPGA	49463 μs	142478 μs	20.84 张

第 10.6 节 本章小结

本章以基于卷积神经网络的道路交通标志识别系统为例，介绍在 ARM+FPGA 异构系统平台上进行智能嵌入式系统的综合开发，将第 2 章至第 9 章的内容进行综合实践，为智能嵌入式系统设计和开发提供一个示范工程。将本教材所讲授的内容应用到智能嵌入式系统的综合开发是本教材的初衷。