第四届全国大学生集成电路创新创业大赛

电子科技大学校内选拔赛

项目技术文档

杯赛名称：Arm 杯

杯赛题目：智能口罩佩戴识别芯片及系统设计

队 伍 号：ASC125149

团队名称：旧的不去，芯的不来

摘要

本设计是一款用于口罩佩戴识别的前端 SoC 芯片，它将识别算法部署在 FPGA 上，在一定程度上克服了传统云端计算方案实时性差、规模受限的不足。

该系统旨在实现多目标人脸定位以及口罩佩戴判断结果显示的功能，其采用多特征尺度的 SSD 卷积神经网络算法，且在最优访存架构的基础上提出了一种卷积计算阵列的优化方案，并针对池化层的不规则性提出了一种多模式灵活池化算子结构。为了提高模型结果后处理的执行时效，本设计又提出了一种将冒泡排序、预测解码和非极大值抑制等功能集于一体的后处理算子，实现了后处理过程的全硬件化。在此基础上，引入摄像头和显示器件，配合软件控制，从而达到了图像实时捕获和结果显示的效果。

目录

引言

新冠肺炎疫情局势当下，病毒的传染速度和规模正在迅速增大，全民佩戴口罩行动刻不容缓。然而，在某些人力监管力度较低或人流量较大场所，很难保证口罩佩戴检查能够落实到每一个人，这成为了导致病毒扩散的安全隐患。这时，以人工智能技术为核心的口罩佩戴识别技术便应运而生，一个个口罩佩戴识别装置被部署在社会的每一个角落，它们能够通过摄像头实时监控人群的口罩佩戴情况，并对未佩戴口罩的人员施以警告。

纵观这些常见的口罩佩戴识别装置，大多都采用云计算的方案，即前端采集的影像数据被传送至云中心，由中心服务器计算得到结果后返回至前端显示。其识别算法虽然取得了较好的效果，但随着监控需求的增大，识别装置部署的范围和数量也在增加，前端采集数据量增加，对服务器的性能如传输带宽、计算能力要求便越高，其成本会随之增加；且作为即时交互系统，随着数据规模增大带来的传输时延增加，其实时性愈难以保证；在服务器资源有限的条件下，识别系统的扩展规模在一定程度上也会受到限制。因此，为了使系统更加可靠、更可扩展以及实时性更强，口罩佩戴识别系统有待向边缘计算发展。

本文阐述了一种应用于口罩佩戴识别的前端 SoC 芯片设计方案，它以 FPGA 为载体，将摄像头采集的影像数据通过神经网络硬件加速引擎计算后，将结果实时显示在前端屏幕上，在运行过程中，只需与服务器进行很少的数据交互（如上电重启后的参数初始化、非期望目标的云端上传等）。该设计在一定程度上克服了传统云计算识别系统实时性差、规模受限等不足，且经测试获得了良好的识别效果。

本设计可应用于任何由人力监管力度低或人流量较大导致口罩佩戴监督力度与精度不足的场所，如商场、地铁站、走廊、天桥、电梯等，该装置轻便、体积小，因而能够被安装在隐蔽的角落，当检测到人员未佩戴口罩时，它能够在第一时间将该人员面部图像传输至云服务器，以便进行人脸的识别与记录。由于采用了边缘计算，该装置可以在很大程度上节省数据的传输带宽、减轻云服务器负担，从而能够得到更大规模、更加灵活的部署，为本次防疫阻击战作出重要贡献。

第一章 功能介绍与智能算法选择

1.1 功能介绍

本次设计拟实现的功能为：

1.通过摄像头采集图像，在 FPGA 上经过智能算法处理，实时定位人脸，并标记出每个人脸是否佩戴口罩。

2.将标记后的图像实时显示在屏幕上，并将未佩戴口罩的人脸传送至云端服务器，以便进行人脸的识别。

1.2 智能算法选择

1.2.1 方案分析与选择

在本设计中，智能算法的选择与实现是关键，因此在此只介绍智能算法及其模型的方案选择，现拟定方案如下：

方案一：采用当前流行的 YOLO（You Only Look Once）算法，将佩戴口罩和未佩戴口罩划分为两个类别，配合 NMS（非极大值抑制）后处理，达到同时定位和分类的目的。

方案二：采用 SSD（Single Shot Detection）算法，同样配合 NMS 进行检测。

方案三：采用 MTCNN（Multi-task Convolutional Neural Network），即多任务卷积神经网络。该网络只适用于单类目标检测，因此还需采用相应的后处理算法，如支持向量机（SVM）等进行目标分类，以达到减少虚警和漏报的目的。

方案分析：

YOLO 虽然在目标检测领域有着突出的性能，但是其更偏向于多类目标检测的应用领域，对于本设计需求的两类目标检测，无疑是一种浪费，再加上 YOLO 的网络模型庞大，一旦将计算部署在 FPGA 上，在没有成熟的轻量化方案可利用时，时效会大大降低。

MTCNN 在单类目标检测与识别领域中具有网络结构简单、图片大小自适应的优点，MTCNN 采用了图像金字塔的思想，能够在不同尺度上对图片进行检测。其模型由 P-Net，R-Net 和 O-Net 三个网络级联构成，由于网络逐级加深，因此能够更大程度地提取目标的特征，从而达到目标识别的目的。但是该算法执行流程复杂，不适合硬件实现；且需要后处理算法辅助分类，增大了设计的难度。

SSD 采用多尺度特征图的方法，并且设有灵活比例和大小的预选框（anchor），因而能够适应大范围、高密集度的目标检测，非常适合本设计所应用的人脸检测。SSD 的主干网

络采用 VGG-16 的网络模型，规则且卷积核大小固定，因此易于硬件实现。并且在多方面搜集资料后，我们掌握了一种 SSD 模型轻量化的成果，这更为我们选择 SSD 算法提供了坚实依据。

综上考虑，本设计采用方案二。

1.2.2 智能算法介绍

SSD 相比于传统应用于目标检测的深度学习算法主要具有两个突出特点：多尺度特征映射和 anchor 多样化。前者指 SSD 网络将不同层的输出作为不同尺度的特征图，一并做出预测。这种做法克服了传统图像金字塔方法多循环、时效低的缺点，同时相比于单特征图的 CNN 算法能够适应更大范围的目标检测。后者指 SSD 的特征图每个单元都提供 4 个大小、比例不同的 anchor，这在一定程度上增强了该算法检测的精准性。

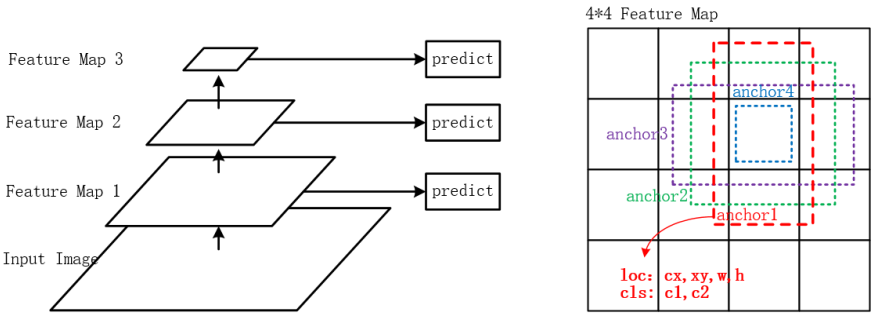


图 1 多尺度特征图映射(左) anchor 多样化(右)

本算法采用 33、17、9、5、3 五个特征图尺度，标记 Mask 和 No Mask 两个类别，由于每个特征图单元包含 4 个 anchor，每个 anchor 对应一个预测框，于是网络输出 5972 个预测框，每个预测框含有 4 个位置信息和两个置信度信息。

得到这些输出后，首先对每一个预测框的两个置信度取最大值（因为只有较大的值才能正确反映其类别），记录其类别后依据置信度阈值对置信度进行筛选。然后依照 anchor 的尺寸对位置信息解码，解码公式如下：

$$b^{cx} = 0.1\, d^{w} l^{cx} + d^{cx} \tag{1}$$

$$b^{cy} = 0.1\, d^{h} l^{cy} + d^{cy} \tag{2}$$

$$b^{w} = d^{w} e^{0.2 l^{w}} \tag{3}$$

$$b^{h} = d^{h} e^{0.2 l^{h}} \tag{4}$$

其中 cx、cy、w 和 h 分别代表中心 x 坐标、中心 y 坐标、宽度和长度。d、b 和 l 分别代表 anchor 值、实际值和网络预测输出值。

解码后的位置信息经过 NMS（详见 2.2.3 NMS 算子设计）处理后得到最终的人脸框坐标。 算法执行总流程如图 2 所示。

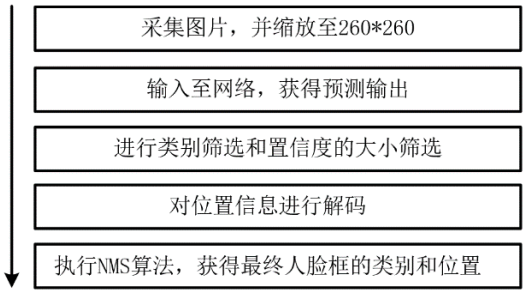


图 2 算法执行总流程

1.2.3 网络模型介绍

对 VGG-16 原始模型进行两方面修改：轻量化和全卷积化，使得其模型体积和参数数量大大降低，完整模型如图 3 所示（原图请见附件）。

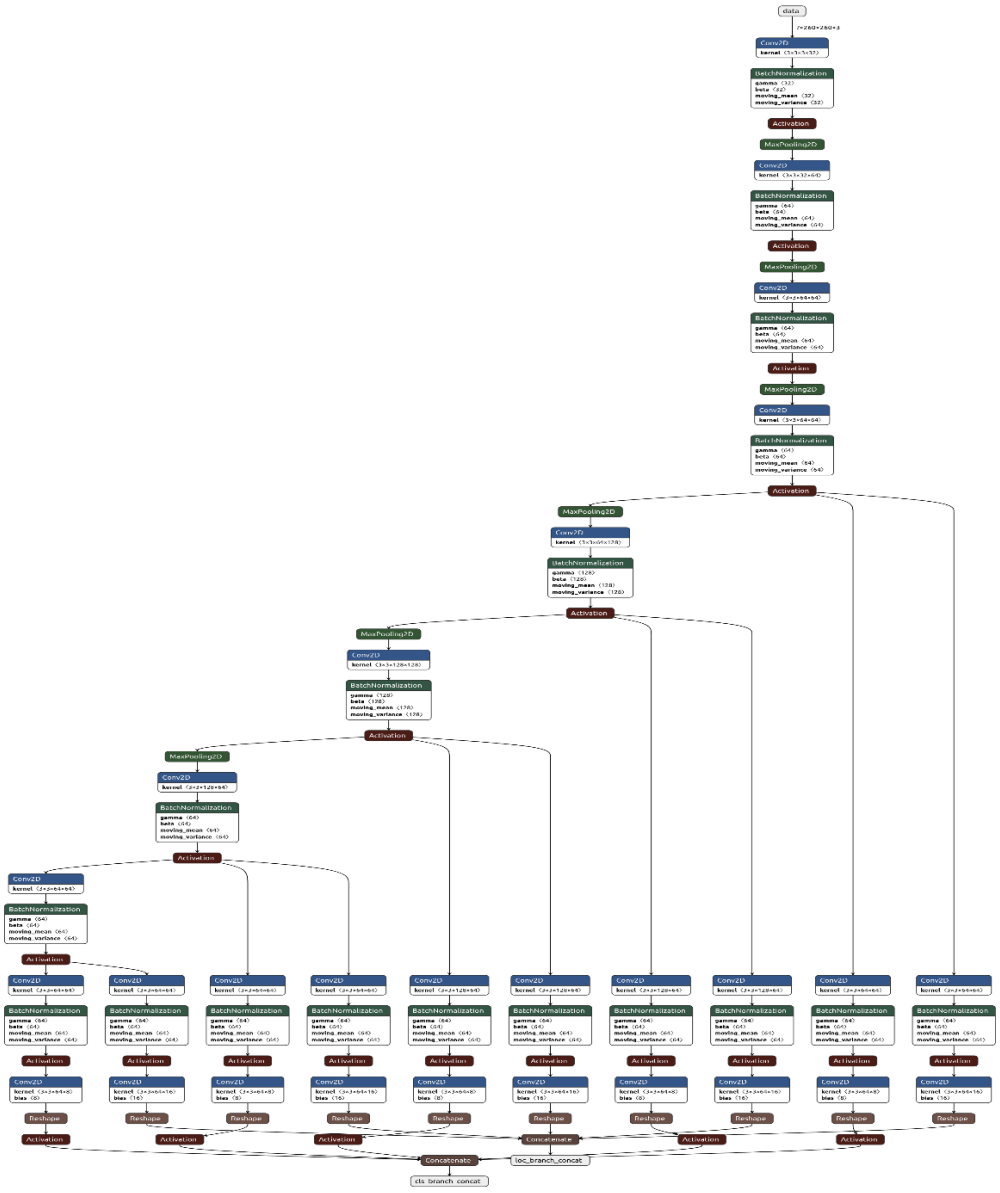


图 3 完整网络模型

该模型接收 260*260 RGB 格式图片输入。若把 padding、卷积、BN、池化等看作一大层，则该模型含有 8 个主干层和 20 个分支层，主干层用于生成不同尺度的特征图，分支层用于对不同尺度的特征图做出预测。

1.3 模型简化

1.3.1 卷积层与 BN 层合并

BN（Batch Normalization），是一种在神经网络中为了提高训练效率，对网络某一层输入数据的每个 Batch 进行正则化，从而控制其均值和方差的手段，一般作用在激活函数之前。它的处理过程由式(5)(6)给出：

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{5}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{6}$$

其中，i表示当前 Batch 的第i组数据，x_i为卷积的计算结果，$\hat{x}_i$为对x_i归一化后得到的值，μ_B和σ_B^2为当前 Batch 的均值和方差；y_i为对$\hat{x}_i$进行分布修正后的值，其值直接被送至激活函数，γ和β为修正参数，由网络训练得到。

一旦网络训练结束，μ_B，σ_B^2，γ和β都为确定的常数，这时卷积层和 BN 层可以总体地表示为（由于引入了 BN 层，卷积层舍去了 bias 参数）：

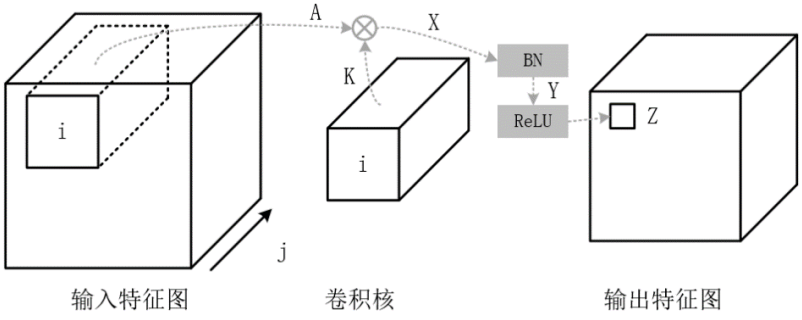


图 4 卷积层和 BN 层

$$X = \sum_j \sum_i A_{ij} K_{ij} \tag{7}$$

$$Y = \gamma \frac{X - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta = \frac{\gamma X}{\sqrt{\sigma_B^2 + \epsilon}} + \left(-\frac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \right) \tag{8}$$

将式(7)代入式(8)可得：

$$Y = \frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} \sum_j \sum_i A_{ij} K_{ij} + \left(-\frac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \right) \tag{9}$$

$$Y = \sum_j \sum_i A_{ij}\left(\frac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} K_{ij}\right) + \left(-\frac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta\right) \tag{10}$$

令：

$$\begin{cases} K_{ij}^* = \dfrac{\gamma}{\sqrt{\sigma_B^2 + \epsilon}} K_{ij} \\ B^* = -\dfrac{\gamma \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \end{cases} \tag{11}$$

则：

$$Y = \sum_j \sum_i A_{ij} K_{ij}^* + B^* \tag{12}$$

式(12)展示了使用一个参数为K_{ij}^*和B^*的新卷积层代替原始卷积层和 BN 层，新卷积层的参数的计算方法由式(11)给出。通过合并卷积层和池化层，简化了网络结构，减少了网络参数，在一定程度上降低了硬件设计难度，节省了推理时间和硬件资源。

1.3.2 Sigmoid 函数简化

Sigmoid 函数是神经网络中常用的激活函数，其表达式如下：

$$y = \frac{1}{1 + e^{-x}} \tag{13}$$

在本设计采用的算法中，需要使用 Sigmoid 函数激活置信度预测值。然而，Sigmoid 函数的硬件实现方法非常复杂，如 ROM 查找表法、泰勒级数逼近法、cordic 算法等，即使使用 Xilinx 的 cordic IP 核，也会造成较多资源的占用

经观察发现，置信度的作用仅体现在被比较和筛选，因此，只要保证其大小关系不发生变化即可，而不需对其精确定量。Sigmoid 函数是单调递增函数，因此其输入 x 和输出 y 是唯一映射的，假设置信度阈值为 y_thresh，则其对应的 x_thresh 便被唯一确定；且对于多个置信度 y1-ym，其对应的 x1-xm 大小关系不变。因此可以直接省去 sigmoid 函数，将原本的 y_thresh 替换为 x_thresh。值得注意的是，由于网络中间结果采用定点数量化，因此需要保证其表示范围大于 x_thresh，当输出溢出时，可直接判定为保留该置信度。

通过在 Pytorch 框架上进行多次验证，我们得出 y_thresh 为 0.5 时推理效果较好，于是 x_thresh 可设置为 0，这样原本复杂的 Sigmoid 函数便可直接被替换为简单的正负判断。

第二章 硬件设计

2.1 SoC 总览

2.1.1 SoC 总体架构

我们设计的 SoC 由如下几个主要模块构成，ACC、DMA、CAMERA、LCD 等模块为自主编写所得，其余模块由 CMSDK 生成或提供。

- Cortex-M3 Core
- Bus Matrix
- Single Channel DMA
- ITCM(Store new program code from KEIL for debugging)
- DTCM(Stack and Heap)
- UART
- GPIO(LED and Switch)
- LCD
- CAMERA
- ACC
- Timer

所有外设通过 AHB3lite 总线接入 Bus Matrix 实现与核的通信，SoC 总体结构如下。

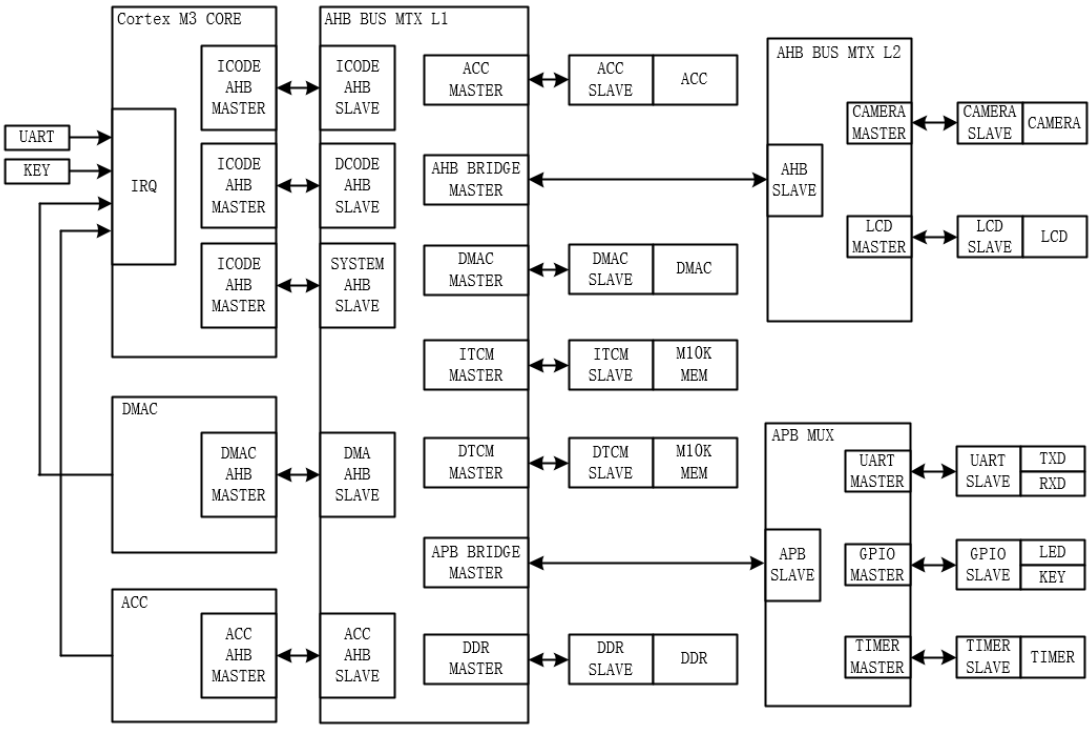


图 5 SoC 总体架构

下图展示了 AHB BUS MATRIX1 的互联关系。

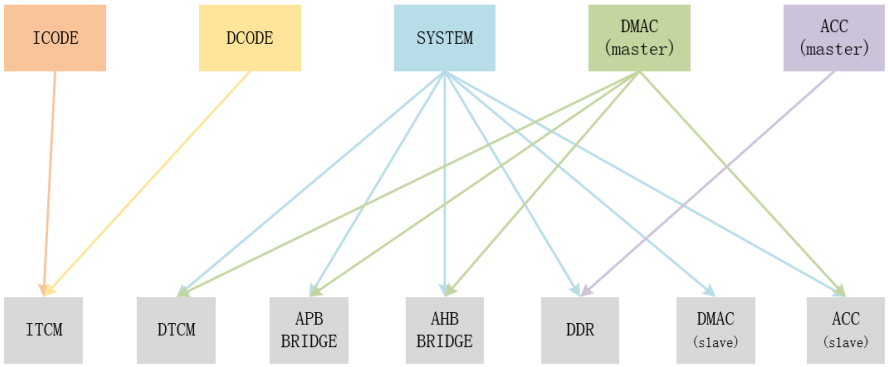


图 6　AHB BUS MATRIX1 互联关系

Cortex-M3 总共使用了 9 个 IRQ 中断：

- 四个按键中断，由 FPGA 开发板上 PUSH BUTTON 经过消抖与上升沿检测得到；
- 卷积计算结束（conv_done），卷积池化算子的计算结束标志；未检测到目标（no_face_detected），标志着本次推理未检测到人脸；结果可读（you_can_rd_result），标志着推理的结束，告知 CPU 结果寄存器结果可读。
- DMA 中断，由于 Bus Matrix 为静态仲裁，需要用 WFI 指令配合 DMA，使核主动交出总线控制权，因此 DMA 工作完成后需要用中断唤醒核；
- UART 中断，UART 接收模块没有缓冲区，因此使用中断。

2.1.2 Memory Map

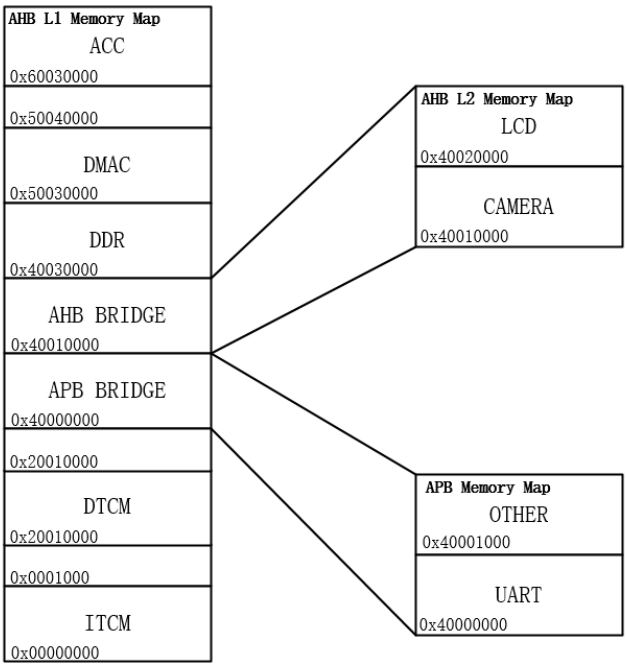


图 7　Memory Map

2.2 神经网络加速器（ACC）设计

2.2.1 ACC 结构总览

（a）ACC 结构组成

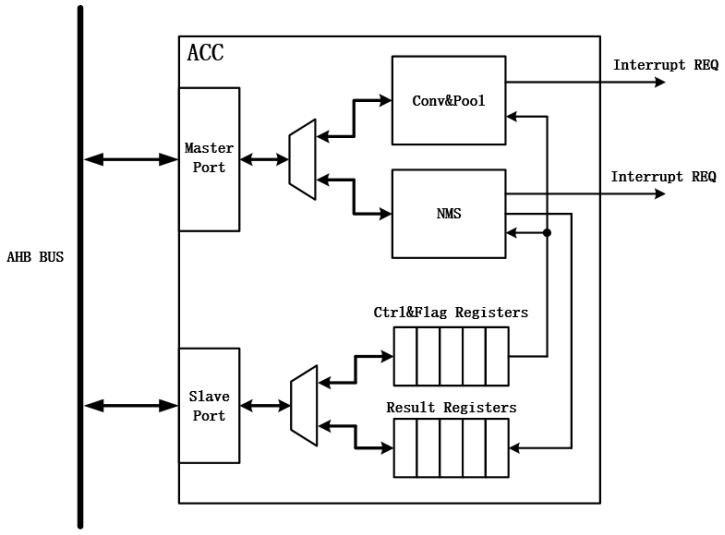


图 8 ACC 结构组成

ACC 设有 master 和 slave 两个接口，均遵循 AHB 总线协议。Master 接口与卷积池化（Conv&Pool）算子和非极大值抑制（NMS）算子两个运算单元相连；Slave 接口与 ACC 内部两个寄存器堆相连，它们分别是控制&标志（Ctrl&Flag）寄存器堆和结果（Results）寄存器堆，前者存放 ACC 的配置信息与状态标志，后者存放推理得到的最终结果。

另外，ACC 的两个运算单元能够分别产生一个中断请求，用以指示当前卷积层或 NMS 运算的结束。

（b）控制寄存器堆

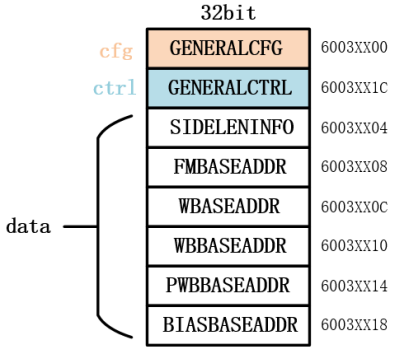


图 9 控制寄存器堆名称与地址映射

该寄存器堆包括 8 个 32 位寄存器，其中 1 个配置寄存器，1 个控制寄存器，6 个数据寄存器。其中配置寄存器在每一层卷积运算时起到配置卷积关键控制信息的作用，控制寄存器用于使能一些计数器、设置标志以及发出运算启动信号，数据寄存器用于存放当前运算访问 DDR 的基地址以及输入输出特征图边长信息等。各寄存器含义参照下表：

寄存器名称	位	含义
GENERALCFG	6-0	输入通道数-1
	12-7	一次 pass 中激活的输出通道数-1。硬件允许的最大并行输出通道数为 64，但某些层的输出通道为 128，则需要两个 pass 完成该层运算
	18-13	Δx,缓存窗边长
	19	是否预写回。在网络分支处，卷积的结果经 relu 后预写回（pre-write-back）作为网络分支的输入，其再经过池化后写会(write-back)作为网络主干上下一卷积层的输入
	20	是否处于网络最底层。在网络最低层的卷积运算比较特殊，它们的结果没有经过 ReLU 激活，因此使用该位来表示当前运算正处在这些层
	22-21	00：不是前三层；01：第一层；11：第二层；10：第三层。前三层输入特征图较大，因此被分割为多个缓存窗，因此其池化 padding 方式和输出 padding 方式都比较复杂，需要单独表示
	23	输出是否 padding。输出的 padding 即下一层卷积运算前的 padding
	27-24	n，表示缓存窗个数。输入特征图边长=n*Δx
	28	运算需要的 pass 数-1
	29	是否进行池化
	30	0：进行卷积池化运算；1：进行 NMS 运算
	31	保留
GENERALCTRL	0	卷积运算启动信号
	1	卷积计数器(m_FSM)使能信号
	2	特殊标志
	3	NMS 运算启动信号
	4	CPU 告知 ACC 读结果完毕
SIDELENINFO		输入特征图，输出特征图的边长信息
FMBASEADDR		输入特征图在 DDR 的基地址
WBASEADDR		权重在 DDR 的基地址
WBBASEADDR		输出写回 DDR 的基地址
PWBBASEADDR		输出预写回 DDR 的基地址
BIASBASEADDR		ACC 内部 Bias ROM 基地址

表 1 控制寄存器堆明细

（c）结果寄存器堆

ACC 结果寄存器堆提供五个寄存单元用于寄存 NMS 运算结果。每个寄存单元包括 6 个只读寄存器：is_valid（用于指示该单元的信息是否有效）、category（表示是否佩戴口罩）、xmin，ymin，xmax，ymax（人脸框位置信息）。

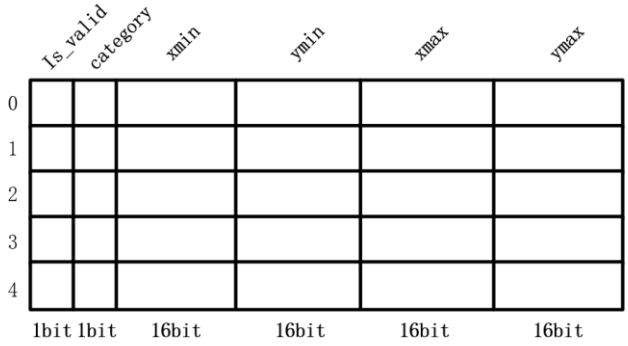


图 10 结果寄存器堆

2.2.2 卷积池化算子设计

（a）卷积池化算子结构与功能

卷积池化算子结构如图 4 所示。卷积池化算子能够在一次运算中将输入特征图中相邻两个通道上的一个窗口（缓存窗）读入计算阵列，配合权重进行卷积计算、池化运算，并将运算结果（ReLU 之前、ReLU 之后或者池化后）经过 padding 后写回至 DDR 相应的区域。当计算遍历完输入特征图的所有缓存窗和通道，整层运算结束，产生中断请求。

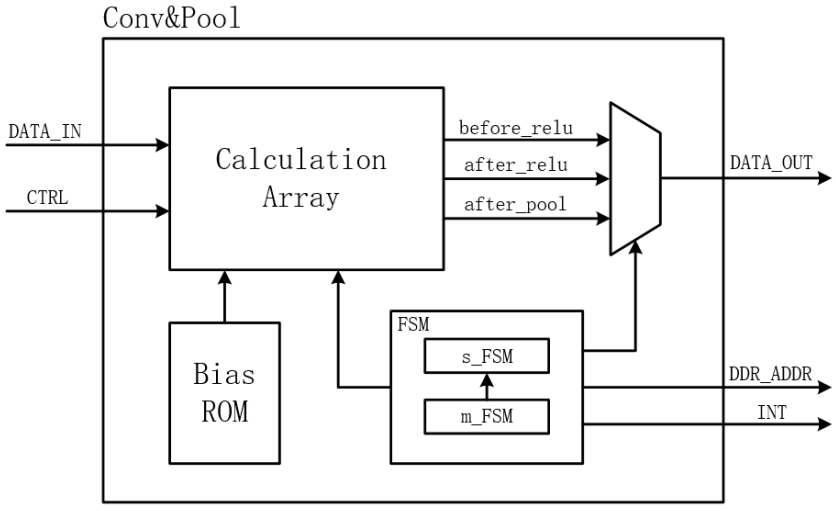


图 11 卷积池化算子

其中，DATA_IN 为算子的数据输入总线，DATA_OUT 为数据输出总线，CTRL 即控制寄存器堆表示的控制信息，Bias ROM 用于存放卷积的偏移量参数。

（b）卷积计算阵列的访存优化及设计

[1]针对使用片外存储方式的 CNN 硬件加速引擎提出了其访存次数的下界条件，并给出了满足该条件的最优访存架构。本设计采用权重与特征图片外存储的方式，使用同样的方法对网络模型进行了计算分析，在考虑 FPGA 内部 RAM 和 DSP 资源数量的前提下，给出了一种访存次数优化方案以及相应的硬件计算阵列架构。

采用[1]中的分析方法，将卷积运算表示成矩阵相乘形式：

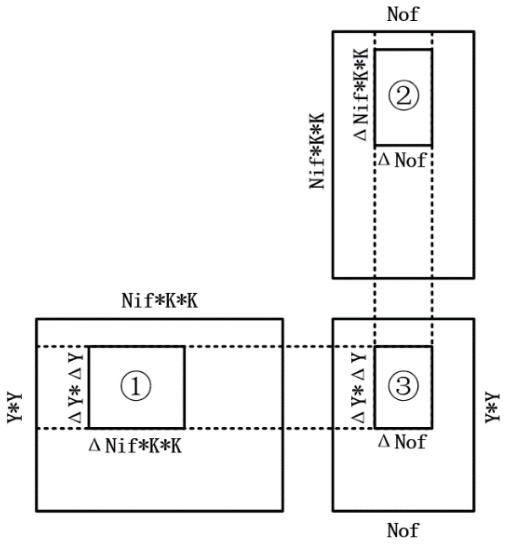


图 12 矩阵相乘表示的卷积运算

其中各变量含义如下：

N_{if}	输入特征图通道数
N_{of}	输出特征图通道数
K	卷积核边长
Y	输出特征图边长
ΔN_{if}	单次访存中获取的输入特征图通道数
ΔN_{of}	单次访存中获取的输出特征图通道数
ΔY	单次访存所得数据对应的输出特征图区域的边长

在一次访存中，获取输入特征图的①区域和权重矩阵的②区域，进行计算后，得到对应输出特征图③区域的一个部分和，该计算过程称为一轮计算。

在本模型中，卷积核大小$K = 3$,因此输出特征图中边长为ΔY的区域对应输入特征图中区域的边长为$\Delta Y + 2$，可得单次访存量为：

$$U_{once} = (\Delta Y + 2)^2 \Delta N_{if} + K^2 \Delta N_{of} \Delta N_{if} \tag{14}$$

每得到一个完整的输出特征图，需要访存的次数为：

$$T = \frac{Y^2 N_{if}}{\Delta Y^2 \Delta N_{if}} \frac{N_{of} N_{if}}{\Delta N_{of} \Delta N_{if}} \tag{15}$$

总访存量为：

$$U_{total} = U_{once} T = \left[\frac{(\Delta Y + 2)^2}{\Delta N_{of}} + K^2 \right] \frac{Y^2 N_{if}^2 N_{of}}{\Delta Y^2 \Delta N_{if}} \tag{16}$$

可以看出，在其他参数不变的情况下，随着单次访存中获取的输入特征图通道数ΔN_{if}和输出特征图通道数ΔN_{of}的增大，总访存量减小。然而ΔN_{if}和ΔN_{of}的增大会受到片内存储容量和硬件并行计算能力的限制。

本设计拟定了两个计算方案：

方案一：采用输出通道全并行计算且输入单通道读取的方式，即$\Delta N_{of} = N_{of}$，$\Delta N_{if} = 1$。这种情况下，为了满足输出通道数最大（$N_{of} = 128$）的卷积层，应将逻辑资源的并行度设置为 128，即设置 128 个独立的 MAC 单元。虽然资源数量允许，但是对于大部分$N_{of} \leq 64$的卷积层，这种方案会极大降低资源的利用率，造成资源的浪费。

方案二：将ΔN_{of}限制在 64 以下，但是令$\Delta N_{if} = 2$，并对单次访存中读取的所有输出通道和两个输入通道进行并行计算，即由方案一的 128*1 转换为 64*2，这样在采用同等数量的 MAC 单元的条件下，提高了资源的利用率。

综上所述，本设计采用方案二，并提出了一种卷积计算阵列，如图 13 所示。

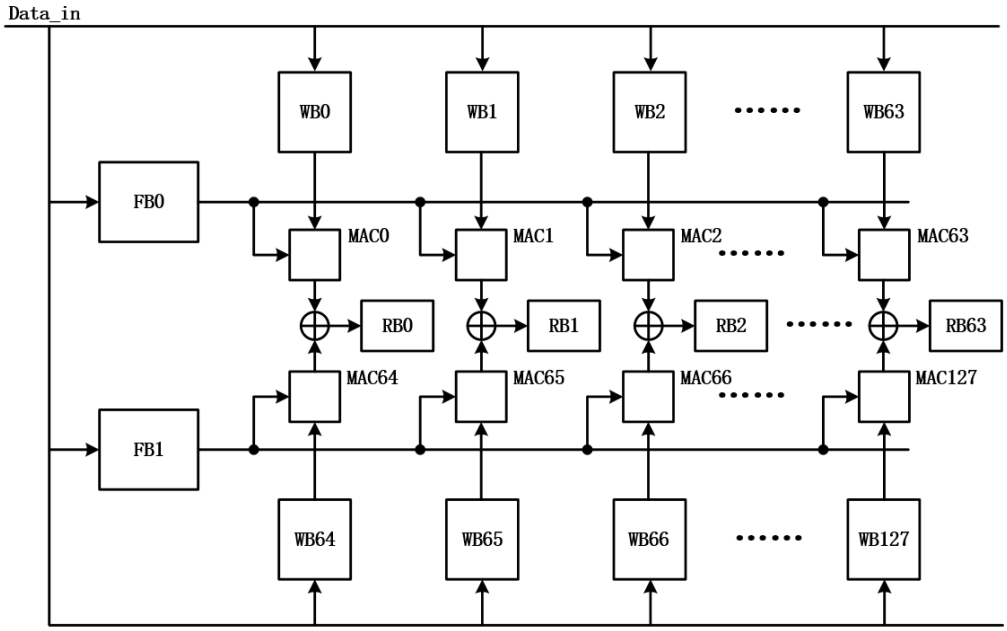


图 13 卷积计算阵列

在一轮计算中，ACC 通过其 Master 接口将输入特征图的一个缓存窗读入 FB0 和 FB1 两个特征图缓存，将权重数据读入权重缓存 WB0-WBx、WB64-WB(64+x)（x 视该层激活的输出通道的并行度而定），将卷积计算的结果存放到结果缓存 RB0-RBx。

图 14 展示了缓存窗和权重在 Buffer 中的存储方式。其中ΔX表示缓存窗边长。在存储过程中，输入缓存窗的宽、高和通道，卷积核的个数都由独立的计数器跟进，这些计数器通过组合逻辑计算生成 DDR 的读地址。

值得注意的是，为了保证多个缓存窗经卷积后的结果能够不重叠、不遗漏地组成完整的结果特征图，多个缓存窗之间应当有 K-1 的重叠。

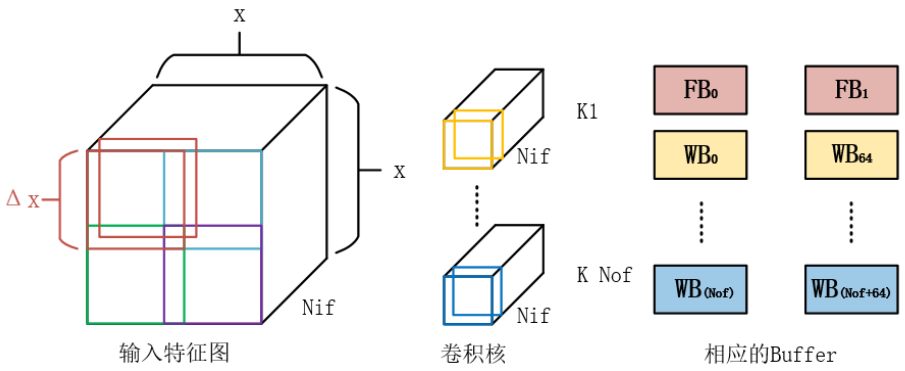


图 14 缓存窗和权重在 Buffer 中的存储方式（Nof 全并行）

图 15 展示了一轮卷积计算的大致过程。在该过程中，当前卷积窗在缓存窗中的位置（宽、高）和当前正在计算单元在卷积窗中的位置（宽、高）都由独立的计数器跟进，这些计数器通过组合逻辑计算生成 FBx 和 WBx 的读地址。数据从 FB 和 WB 串行输出，进入乘累加单元，K*K 个时钟周期后，得到两个输入通道上的部分和结果，经相加后写入 RB，写入地址同样由上述计数器生成。

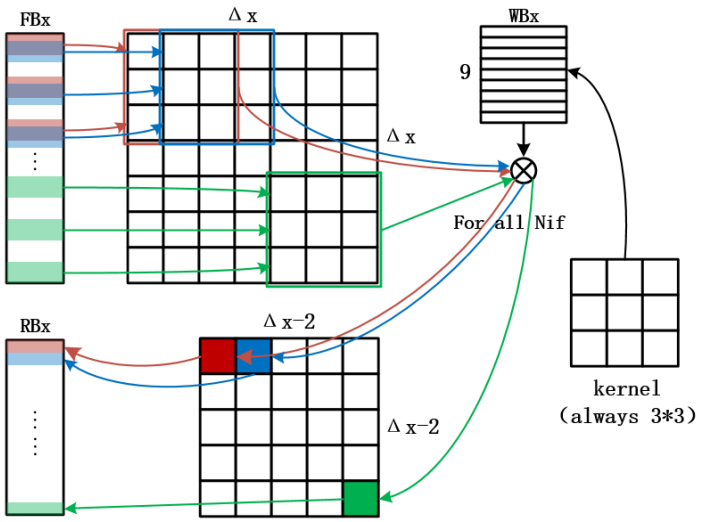


图 15 卷积计算的大致过程

（c）灵活池化

本设计采用的网络模型在池化方式上具有很大的不规则性，主要表现在：池化出现的随机性（即并非每次卷积都跟随池化）；池化 padding 的随机性（即某些特征图经卷积后边长为奇数，需要 padding）；划分缓存窗带来的 padding 的多样化（即划分缓存窗对池化 padding 方式具有不确定的影响）。针对上述难题，本设计通过增加控制信息，设计复杂的 FSM 以及定义多种 padding 方式设计出了一种能够适应多种网络结构、多种 padding 方式的灵活的池化方案。

为了提高资源利用率，卷积与池化采用了 RB 分时复用的方式，即在卷积运算时，将卷积计算结果写入 RB 中；在池化运算时，池化算子从 RB 中读出数据，经 padding 和池化后重新写回 RB。其硬件结构如图 16 所示。

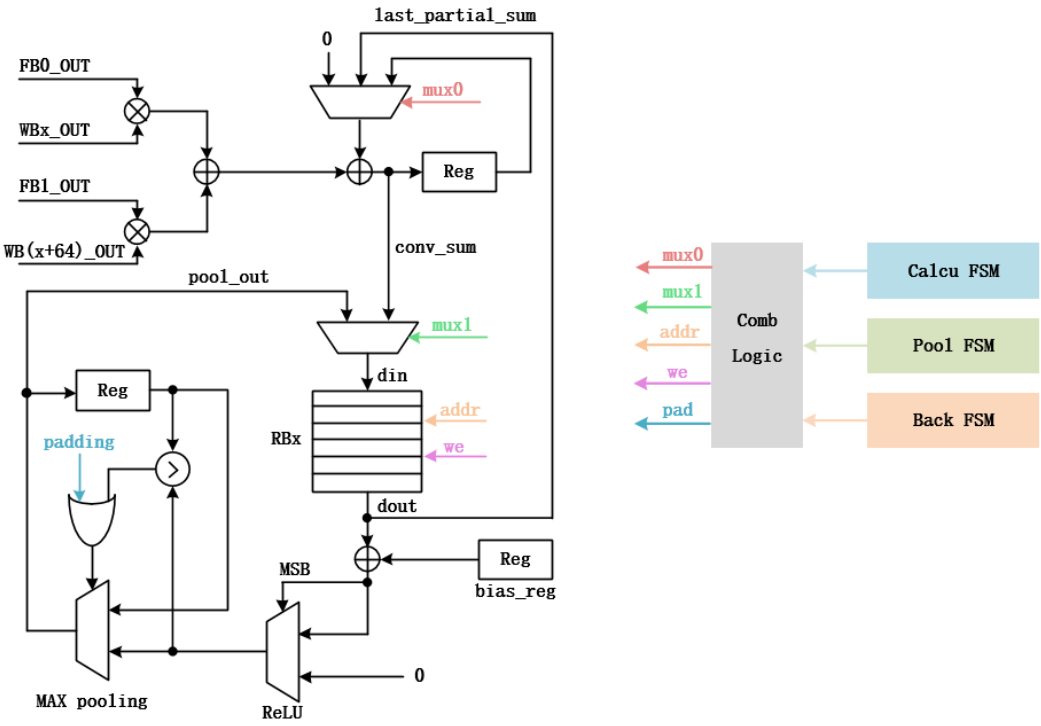


图 16 池化对应的硬件结构

其中 FB_OUT 和 WB_OUT 分别为卷积计算阵列中 FB 和 WB 的输出端。值得注意的是，在一轮计算之后，RB 中存放的只是对应输入特征图两个通道上的部分和，因此，当下一轮的计算开始时，乘累加应该在上一轮计算结果的基础上进行，为此特地提供图中 last_partial_sum 数据通路。当遍历所有输入特征图通道，得到完整和时，便告知 FSM 进入池化状态，同时在下一次卷积计算时乘累加器清零。

在池化过程中，使用一个寄存器来寄存当前池化窗的最大值，当计数器告知硬件当前正处在 padding 状态时，该寄存器值保持不变，达到了 padding 的效果。当完成一个池化窗口的运算后，将池化结果写回至 RB。池化过程可通过图 17 表示。

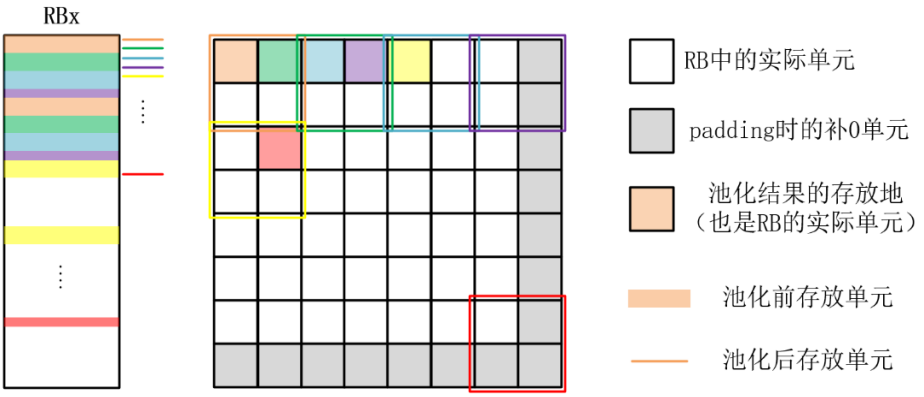


图 17 池化过程

当不划分缓存窗时，卷积和池化均以整个输入特征图为单位进行，这时对于某些卷积结果边长为奇数的层，只需进行半边式的 padding（如图 17）即可；然而在将输入特征图划分为多个缓存窗之后，卷积结果也被相应划分为多个区域（成为卷积结果窗），假设每个缓存窗对应的卷积结果边长为ΔZ，完整的结果图边长为Z。

当ΔZ为偶数时，Z也为偶数，这时每个卷积结果窗都不需要 padding，如图 18 所示。

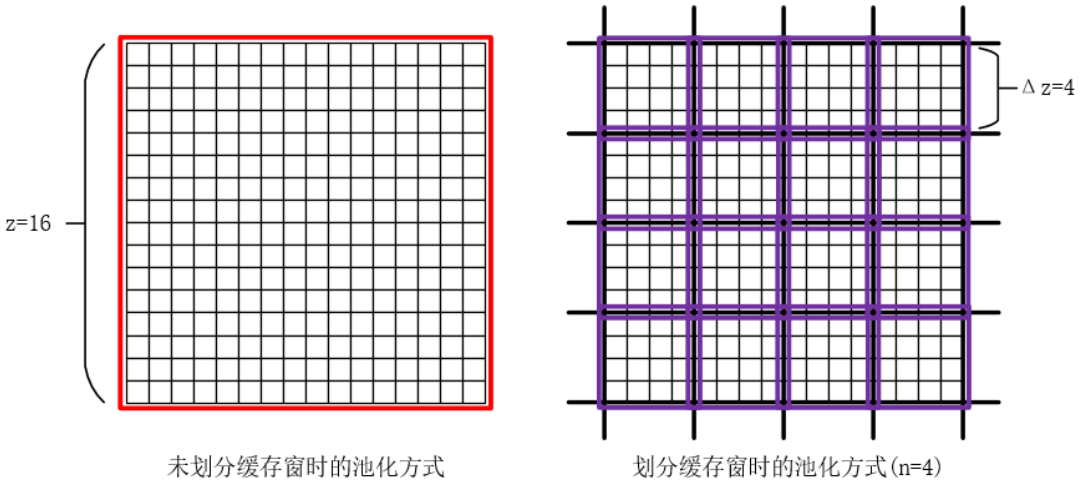


图 18 ΔZ为偶数时的池化方式

当ΔZ为奇数时，无论Z是否为偶数，在每个缓存窗对应的结果窗的边缘，缓存窗的划分都将原本连续的池化过程打断。这时如果每个卷积结果窗都采用相同的池化方式，则会造成池化结果的冗余和缺漏。因此本设计中定义了 4 中特殊的 padding 方式：00 型、01 型、10 型和 11 型，如图 19 所示。

采用这种定义，对同一个输入特征图得到的不同的卷积结果窗进行不同方式的池化，保证了池化结果的完整性。然而这种方式会造成结果窗某些信息的丢失，在缓存窗数量不多的情况下，考虑 CNN 网络的高容错性能，该误差基本可以忽略。

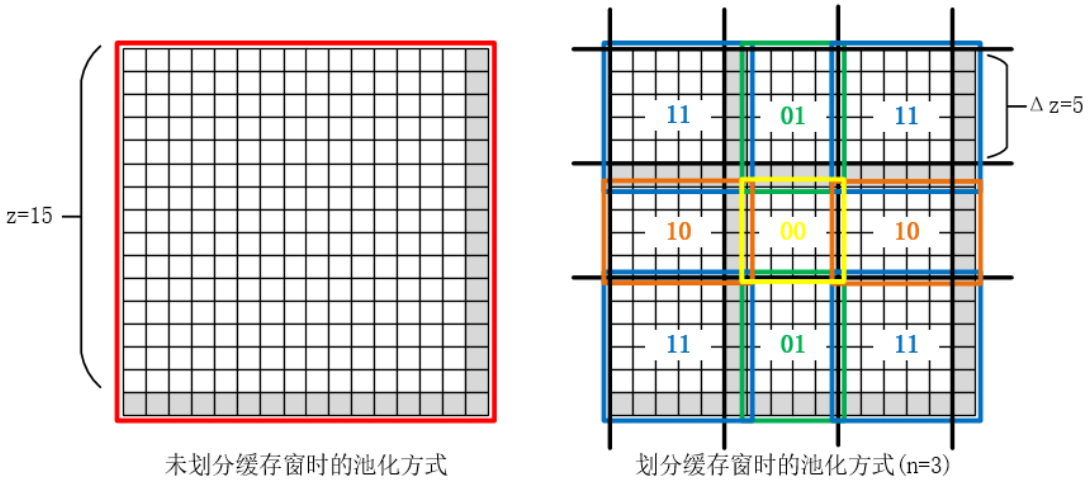


图 19 ΔZ为奇数时的池化方式

为了实现这种多方式池化，本设计中为卷积池化算子引入 pad_style 这一控制信息，并对池化状态的内部计数器和用于生成 RB 访问地址的组合逻辑进行了补充，使其能够对不同的 pad_style 产生不同的池化行为。

（d）FSM 设计

卷积池化算子的运行由两个 Master_FSM 和 Slave_FSM 共同跟进。前者包括 p_Nif、p_nw、p_nh 和 p_k 四个嵌套的计数器，分别用于指示当前计算所处的输入通道、缓存窗在输入特征图中的位置以及当前处在的 pass（见表 1）；后者包括 conv_FSM 和 pool_FSM 两个并行的计数器，conv_FSM 用于指示当前的卷积状态，分别为 idle、wr_fb（写 FB）、wr_wb（写 WB）和 calcu（卷积计算），pool_FSM 用于指示当前的池化状态，分别为 idle、pre_back（预写回）、pool（池化）和 back（写回）。

上述 FSM 的时序图与状态转换图如图 20-22。

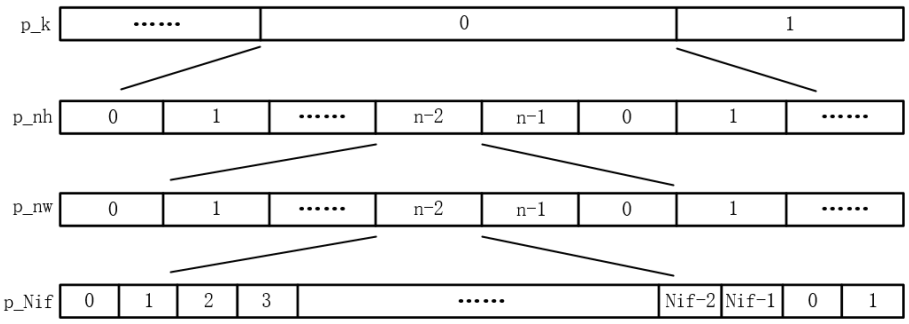


图 20 Master_FSM 时序图

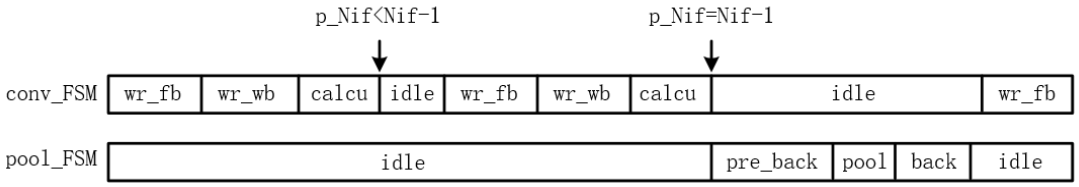


图 21 Slave_FSM 时序图

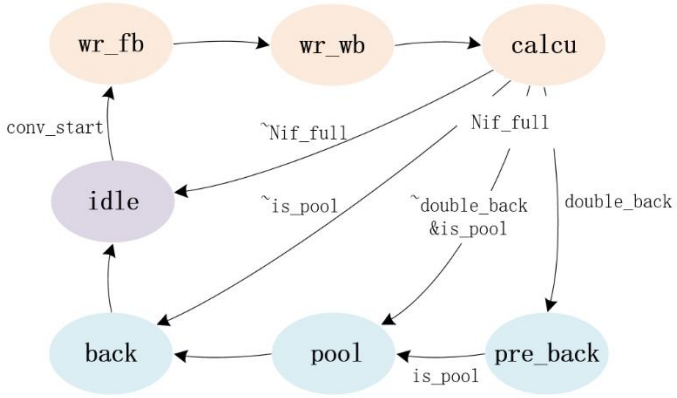


图 22 Slave_FSM 状态转换图

图 15 中 conv_start 为卷积池化算子的启动信号，Nif_full 指示当前已计算至输入特征图的最后一个通道，double_back 表示当前层需预写回，is_pool 表示当前层需池化。

2.2.3 NMS 算子设计

（a）NMS 算法介绍

NMS（Non Maximum Suppression）即非极大值抑制，是应用于目标检测与识别领域的深度学习常见的后处理算法。当得到多个目标定位预测框时，可能出现多个预测框表示同一个目标的情况，这时通过不断地选出最优框和剔除重叠框，就可以筛选出所有能够唯一表示每个目标的预测框。其一般执行流程如下：

①筛选出置信度大于阈值的所有预测框，并进行坐标解码

②对上述预测框按照置信度排序

③选出置信度最大的预测框作为最优框

④计算其余每一个预测框与最优框的交并比（IoU）, IoU 大于阈值的记为重叠框

⑤记录最优框至结果，然后剔除最优框和所有的重叠框

⑥如果剩余预测框数量小于等于 1，结束；否则，跳转至③

⑦return 结果

图 23 展示了 NMS 的执行效果。

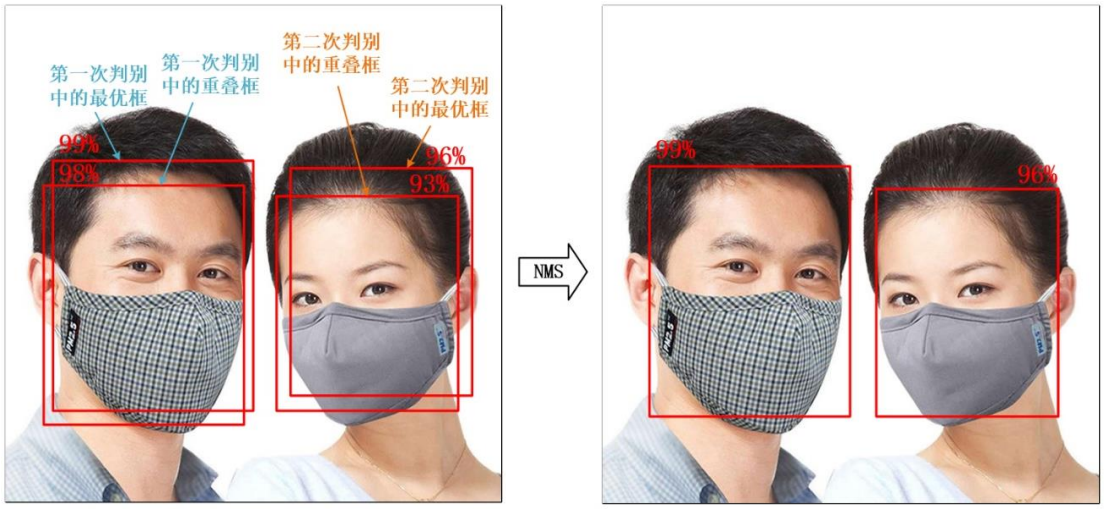


图 23 NMS 执行效果图

（b）NMS 算子结构与功能

本设计中采用的 NMS 总流程分为四个状态：置信度筛选、置信度排序、预测框解码和非极大值抑制，四个状态分时运行，每个状态内部是并行或流水线执行的。

置信度筛选	从 DDR 中读取网络计算得到的原始置信度矩阵[5972,2]（5972 为数量，2 表示 mask 和 no mask 两个类别）
	通过纵向取最大值得到[5972,1]矩阵，同时利用置信度阈值筛选该矩阵，获得置信度数组 cls[m,1]以及相应的类别数组 class[m,1]和索引数组 indx[m,1]（m 为筛选后得到的元素数量）
	将 cls、class 和 indx 分别存入 cls_RAM 和 indx_RAM
置信度排序	对 cls 和 class 同时按照 cls 中置信度的大小进行冒泡排序
预测框解码	从 indx_RAM 中读出 indx 数组，作为地址从 DDR 中读取原始的预测框偏移量矩阵
	对照标准 anchor 反算得出预测框归一化的坐标矩阵 loc（包括 xmin，ymin，xmax 和 ymax 四个位置信息）和每个预测框的归一化面积 area
	将 loc 中每个预测框的 xmin，ymin，xmax，ymax 和 area，连同 indx 存入 main_RAM 中
非极大值抑制	通过不断地读写 main_RAM、计算和比较，最终得到 NMS 结果，并将其写入结果寄存器堆中

表 2 NMS 硬件执行流程

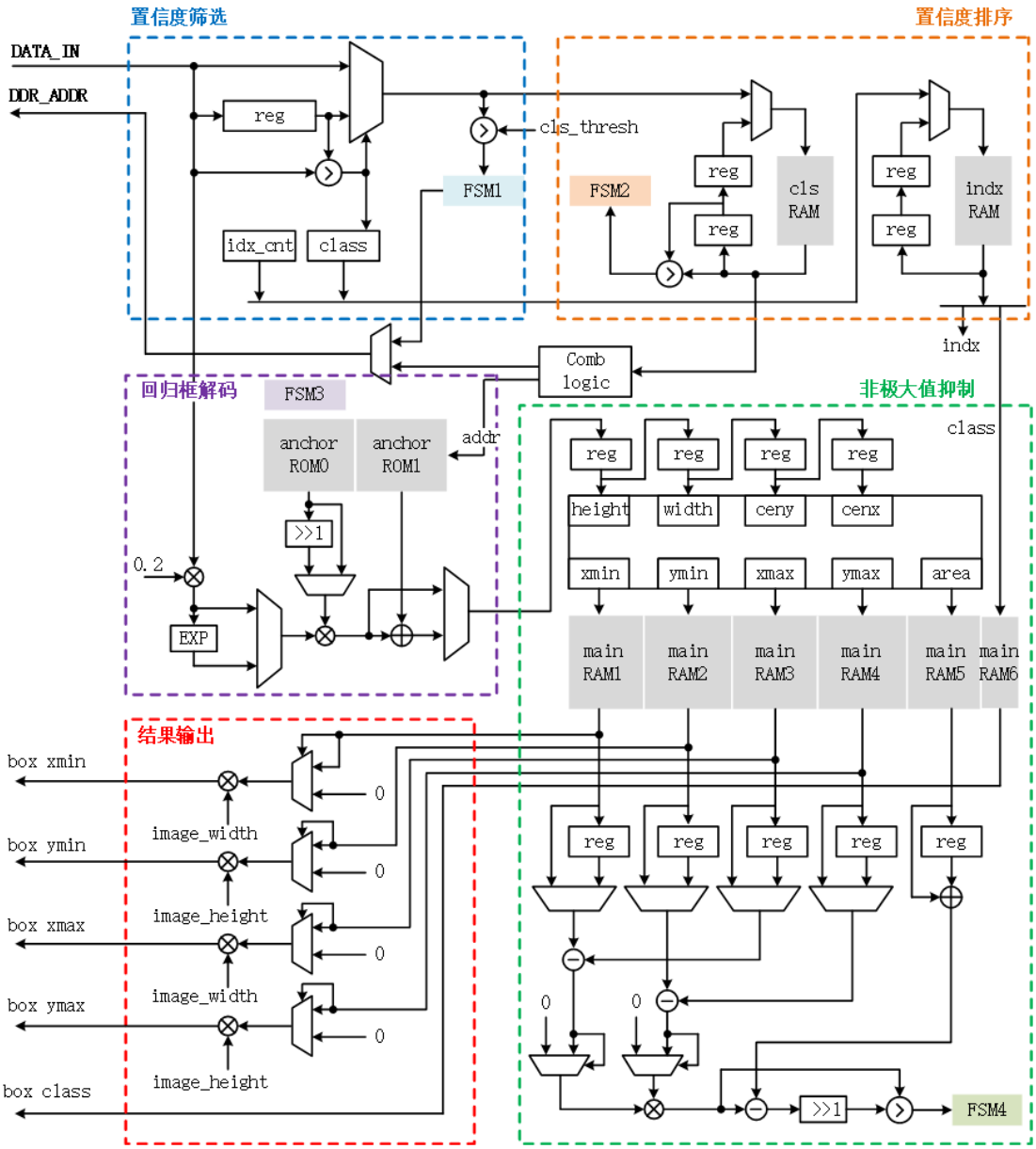


图 24 NMS 算子结构图

（c）硬件冒泡排序实现

硬件冒泡排序包括两个状态：读 cls RAM 和写 cls RAM。初始时为读 cls RAM 状态，该 RAM 的地址按 1 递增，读出的数据被放入一级寄存器 reg1，因此 RAM 的输出端 dout 和 reg1 中总是两个相邻地址的置信度值，如图 24。当比较器检测到 reg1 小于 dout 时，FSM 进入写 cls RAM 状态，这时原本递增的地址保持不变并被禁用，在原本地址基础上生成的交换地址被激活。两个时钟周期后，相邻的两个置信度被交换地址，FSM 返回到读 cls RAM 状态，递增地址重新被激活，比较器继续检测下一组相邻的置信度。由于状态的转换需要延迟一个时钟周期，这里增加了二级寄存器 reg2 来匹配时序。

使用一个计数器 round_cnt 来记录当前的轮次，当读 cls RAM 的地址达到 round_cnt 时，地址清零，开启下一轮的比较，同时 round_cnt 自减一，如此往复，直到 round_cnt 等于 1，产生 sort_done 信号，标志着冒泡排序的结束。

值得注意的是，由于 indx RAM 中存放的 class 和 indx 代表着 cls RAM 中每一个置信度的属性，因此为了保证它们始终保持对应，在对 cls 冒泡排序时，对 indx RAM 也实施相同的操作，即两个 RAM 共用一个地址和写使能信号。

（d）硬件 NMS 实现

软件实现 NMS 伪代码如下：

```
pick = []    #定义数组用于存放 NMS 结果
while box.length > 0:
#box 初始为[m,4]数组，4 代表预测框的 4 个位置信息，m 个框已按照置信度从大到小排序
    pick.append(box[0])      #先把最优框的保留下来
    xmin=box[:,0]      #得到每个框的左上角 x 坐标
    ymin=box[:,1]      #得到每个框的左上角 y 坐标
    xmax=box[:,2]      #得到每个框的右下角 x 坐标
    ymax=box[:,3]      #得到每个框的右下角 y 坐标

    overlap_xmin = max(xmin[0], xmin[1:])   #得到最优框与其他每个框重叠区的左上角 x 坐标
    overlap_ymin = max(ymin[0], ymin[1:])   #得到最优框与其他每个框重叠区的左上角 y 坐标
    overlap_xmax = min(xmax[0], xmax[1:])   #得到最优框与其他每个框重叠区的右下角 x 坐标
    overlap_ymax = min(ymax[0], ymax[1:])   #得到最优框与其他每个框重叠区的右下角 y 坐标

    overlap_w = np.maximum(0, overlap_xmax - overlap_xmin)   #重叠区宽
    overlap_h = np.maximum(0, overlap_ymax - overlap_ymin)   #重叠区高
    overlap_area = overlap_w * overlap_h   #重叠区面积
    overlap_ratio = overlap_area / (area[0] + area[1:] - overlap_area)   #计算 IoU

    need_to_be_deleted = (box[0] , box[where(overlap_ratio>threshold)+1])
    box = box.delete(need_to_be_deleted)   #删除最优框和重叠框
```

本设计中硬件 NMS 的执行状态如图 25 所示。

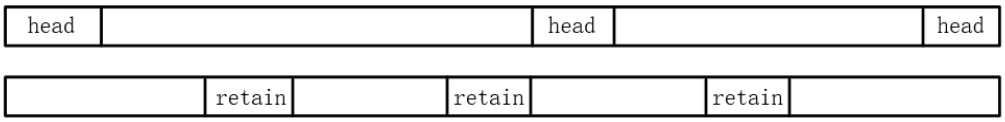


图 25 硬件 NMS 执行状态

在每一轮检测开始时，FSM 进入 head 状态，如图 24，从 main RAM 中读出的位置（面积）信息被存放至寄存器中，作为最优框；同时，这些信息连同 class 被送至 NMS 算子外部 ACC 的结果寄存器堆中。一个时钟周期后退出 head 状态，依次读取 main RAM 中其他框的信息，其坐标、面积的计算采用二级并行流水线。值得注意的是，这里采用 IoU

阈值为 0.5，只需进行简单的移位和比较便可实现，避免了不必要的除法运算。当检测到 IoU 小于阈值时，FSM 进入 retain 状态，将此时 main RAM 的输出信息重新写回至指定地址处，意为保留。一个周期后退出 retain 状态，继续检测 IoU。当有多组信息需要保留时，其写回地址依次递增，这样在一轮检测执行结束时，新的 box 数组便自然诞生，其长度便为上一轮检测时最后一次写回的地址值。

如此反复，当检测到新的数组长度等于 0 时，发出 nms_done 信号，标志着 NMS 结束；当检测到新的数组长度为 1 时，首先进入 head 状态，然后发出 nms_done 信号。

2.2.4 适应模型的 DDR 分区与地址生成

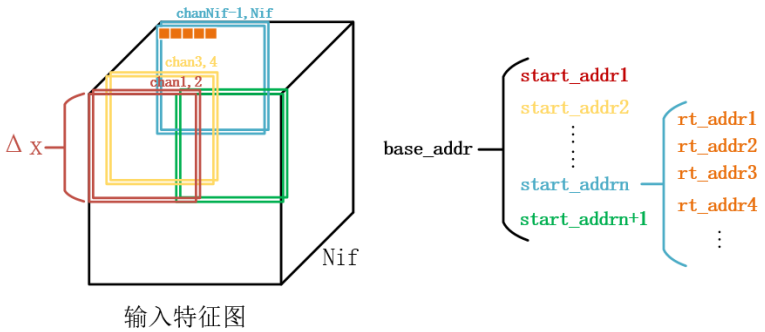
本设计采用的网络模型分为主干和分支两部分，多个分支理论上可以并行运行，但是由于卷积池化算子同一时刻只能计算一个卷积池化层，所以包括分支在内的 28 个卷积池化层必须分时顺序进行，因此定义合适的网络的执行顺序是必要的。

本设计中定义网络分支优先的执行顺序，即在网络分支处，先执行两个分支线路，当分支网络执行完毕，得到最终结果后，返回执行主干线路。基于这种定义，现为 DDR 划分区域如表 3：

名称	包含子区域	作用
W（weights）	28	存储 28 个卷积层的权重
IN（image_in）		存储输入图片
T（trunk）	2	在网络主干上用于“乒乓”存储中间结果
B（branch）	2	在网络分支上用于“乒乓”存储中间结果
R（results）	10	存储网络计算结果，即置信度和坐标信息

表 3 DDR 分区

每个子区域都对应一个基地址。在卷积池化计算时，需要得知权重、输入特征图以及计算结果存放的区域，这时只需 CPU 在当前层运算前向 ACC 的 6 个数据寄存器写入相应区域的基地址，作为整层运算的基地址。由于一层的运算分多个轮次进行，每轮计算的 DDR 访问起始地址都在基地址的基础上，通过 p_k、p_nh、p_nw 和 p_Nif 四个计数器生成。而访存时的实时地址都在起始地址的基础上由 s_FSM 状态内部的计数器生成。



输入特征图

图 26 DDR 访问地址生成

2.3 DMAC 设计

2.3.1 DMAC 结构

DMAC（Direct Memory Access Controller），是嵌入式系统硬件的重要组成部分，它能够在 CPU 不参与传输的条件下，将大量数据从一个 Slave 设备搬运至另一个 Slave 设备。本设计中 DMAC 的结构如图 27。

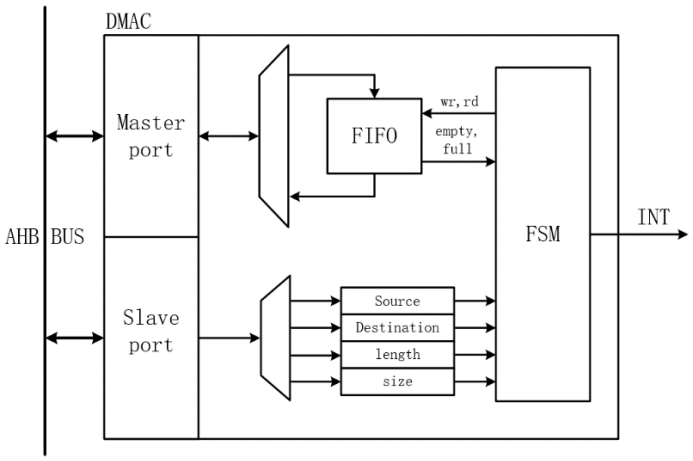


图 27 DMAC 结构

DMAC 含有 Master 和 Slave 两个接口，均与 AHB 总线相连。内部设有 Source、Destination、Length 和 Size 四个寄存器，分别用于存放 DMA 传输时的源 slave 初始地址、目的 slave 初始地址、传输长度和传输位宽。

2.3.3 DMAC 执行流程

当需要进行 DMA 传输时，CPU 通过 DMAC 的 slave 接口配置四个寄存器，然后向 FSM 发出 start 信号，同时释放总线。DMA 传输分为读（写 FIFO）和写（读 FIFO）两个过程，当 FIFO 写满时，FSM 接收到 full 标志，开始将 FIFO 的数据向另一个 slave 写入，直到接收到表示 FIFO 空的 empty 标志，一轮读写完毕。经过多轮读写后，DMA 传输结束，向 CPU 发出 INT 中断请求信号，使 CPU 重新获取总线占有权。

第三章 软件设计

Cortex-M3 软件主要用于系统流程控制和 ACC 运行控制。

3.1 ACC 控制流程

在 ACC 初始化完成之后，首先利用 ACC 的 conv_done 中断进行写 ACCC 控制&标志寄存器堆从而对各层网络运算信息进行配置，然后利用 ACC 的 you_can_rd_result 中断运行结果的读取。

3.1.1 配置寄存器

ACC 主要包含两个运算单元，即卷积池化算子和 NMS 算子，卷积池化算子可分为 28 层，NMS 算子可看作为一层，而每一层网络所对应的控制寄存器内容不同，如果给每一层网络设以单独的寄存器，那将会造成资源的浪费，这并不是我们所希望看到的。因此我们采用软件来实现每层网络寄存器的配置，由于每层网络运行结束后都会产生 conv_done 中断，所以可以在 conv_done 中断程序中完成对下一层网络的配置。流程图如下：

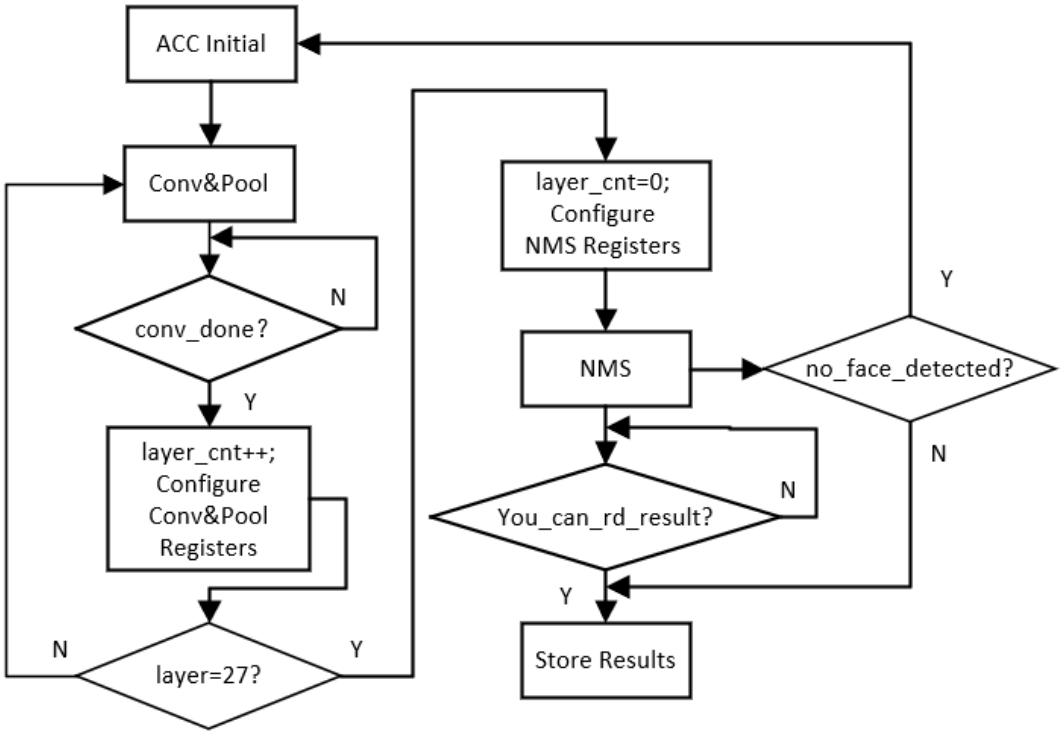


图 28 ACC 软件控制流程

具体实现方法如下：

- 设置变量 layer_cnt，每当 conv_done 中断来临时，layer_cnt 进行加 1 操作，用以区分不同网络层。

- 在 ACC 初始化函数完成卷积池化算子第 1 层网络的配置，并初始化 layer_cnt 为 0。同样的，当 no_face_detected 中断来临时，相应的中断服务程序也将进行此配置。
- 卷积池化算子的第 1 层网络运算结束产生 conv_done 中断进入相应的中断服务程序，根据 layer_cnt 以配置第 2 层网络运算所需的信息。以此类推。
- 当卷积池化算子第 28 层运算结束进入 conv_done 中断服务程序时，则进行 NMS 算子运算所需的信息配置。并将 layer_cnt 置为 0。

3.1.2 运行结果读取

当 NMS 算子运算结束后，整个 ACC 运算也就结束了，运算的结果存放在结果寄存器堆中，而 ACC 运算结束并将结果写入结果寄存器堆时会产生中断 you_can_rd_result，因此我们可以利用此中断完成结果的读取。由前面 ACC 硬件部分可知 ACC 每运算一次可检测最多五张人脸，因此每次运算结束会产生五组数据，每组数据包含 is_valid、category 以及人脸框的坐标，当 is_valid 为 1，标志此组信息有效，则读取坐标值；若 is_valid 为 0，标志信息无效，则不读取相应数据。ISVALID 是一个 8 位的寄存器，我们只需要关注第五位信息即可，每一位代表一个人脸框的信息有效与否。具体流程见图 29，图中所示流程为一个人脸框坐标的读取过程，不同的人脸框需要一一根据 ISVALID 寄存器第五位来判断是否读取。

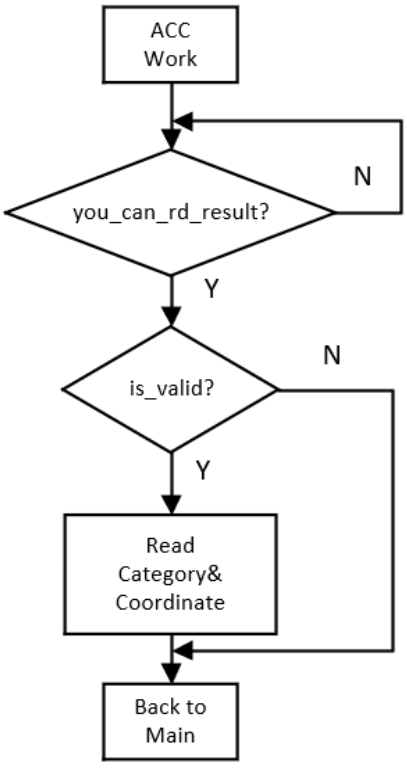


图 29　ACC 结果读取

3.2 系统工作流程

SoC 工作流程如图 30 所示：

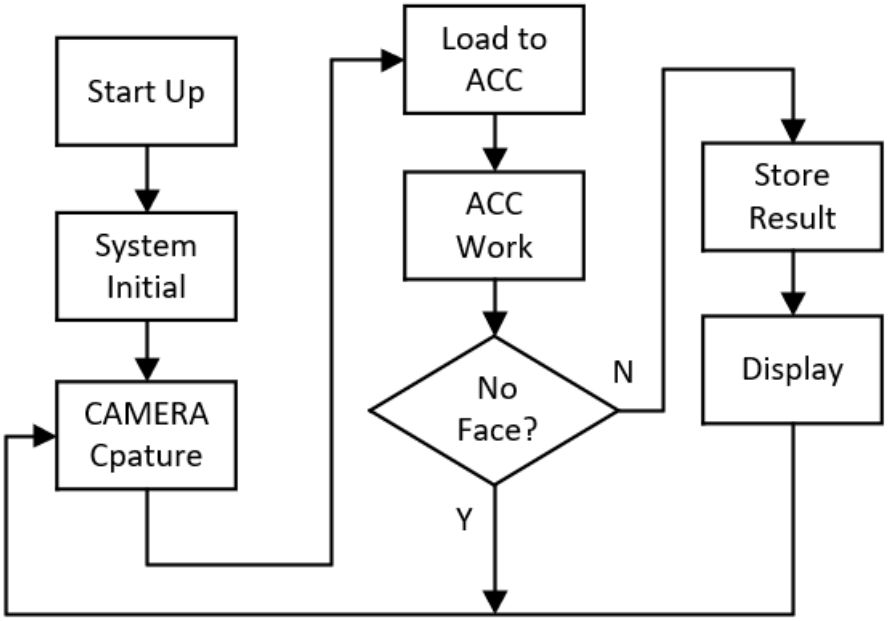


图 30　SoC 工作流程

软件流程主要步骤如下：

1） 系统初始化。对 CAMERA、LCD 显示屏等外设以及中断进行初始化。

2） 将摄像头捕获的照片传进 ACC，ACC 开始工作。

3） ACC 工作结束，则对有效信息进行读取，并根据读取的坐标值将人脸框标记在摄像头捕获的照片上，用不同颜色的框来标记戴口罩者和未戴口罩者。

4） 继续从摄像头捕获照片传进 ACC，实现实时监测。

第四章 仿真与测试

4.1 仿真

4.1.1 ACC 仿真

（a）推理时间计算与仿真验证

使用脚本对推理总时间进行估算，得出在 50MHz 的系统时钟条件下，单帧推理时间为 0.382s 左右。之后使用 vivado 的 testbench 对第 24 层计算进行仿真，测得的计算时间与脚本得出的结果基本一致。

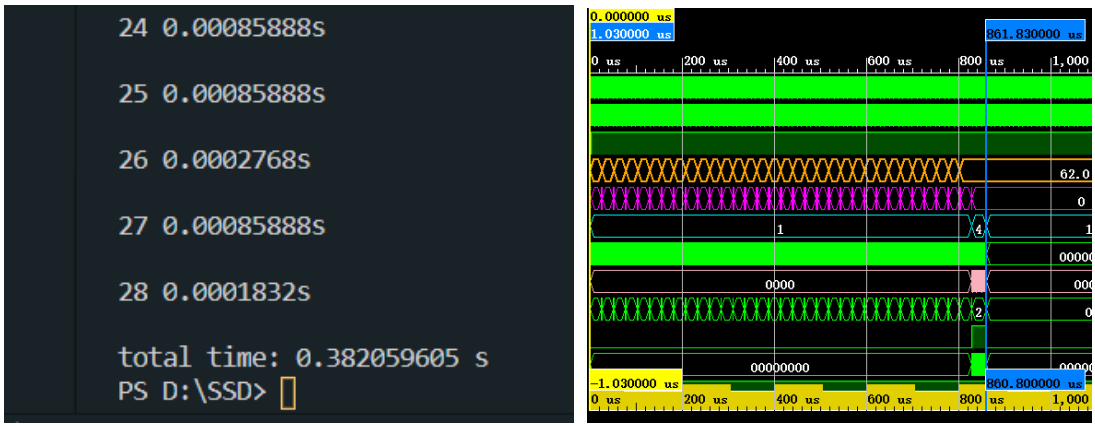


图 31 脚本计算（左）和 testbench 仿真第 24 层（右）

（b）卷积池化算子仿真

图 32 展示了卷积池化的状态机仿真，见 p_Nif、conv_status、pool_status 三个状态机。HADDRM 为 ACC 访问 DDR 的 AHB 总线地址。图 33 展示了第 24 卷积池化层的仿真，其中，conv_out_bus 为写回的结果数据，cnt11-13 代表当前正在返回的数据在输出特征图中所处的列、行和通道。图 34 展示了在 Pytorch 框架下推理得到的网络输出值，将其与 conv_ out_bus 对比可以发现，两者基本一致。（仿真/推理文件请见附件）

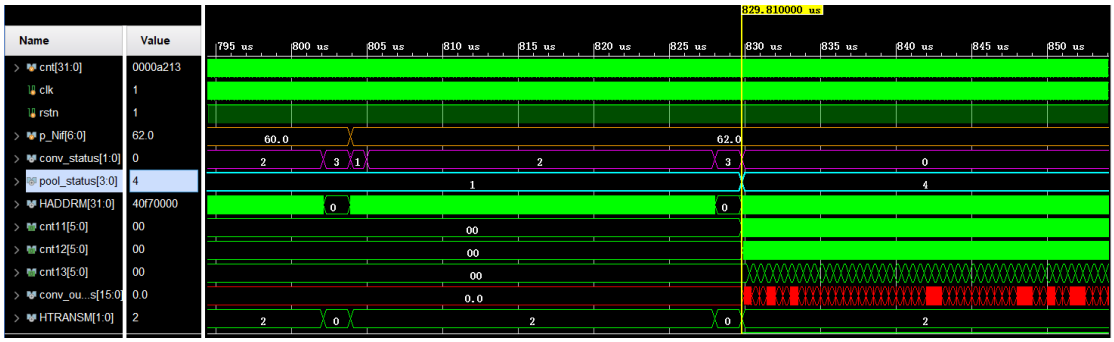


图 32 卷积池化算子仿真之状态机

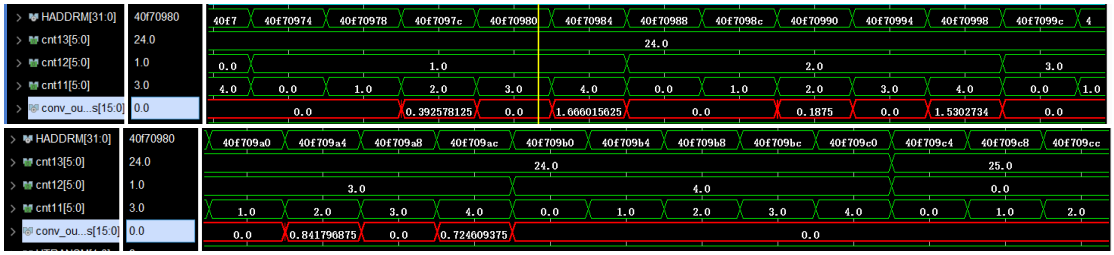


图 33 卷积池化算子仿真之计算结果

```
59    print(out[0,24,:,:])

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS

PS D:\SSD>  cd 'd:\SSD'; & 'D:\Anaconda3\envs\Pytorch\python.exe' 'c:\Use
s\debugpy\launcher' '57946' '--' 'd:\vivado_proj\Competition\ACC\ACC.sim\
tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.3862, 0.0000, 1.6535, 0.0000],
        [0.0000, 0.1721, 0.0000, 1.5083, 0.0000],
        [0.0000, 0.8311, 0.0000, 0.7154, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000]], dtype=torch.float64)
```

图 34 Pytoch 框架上推理计算结果

由于仿真所需时间较长，我们只对部分具有代表性的卷积池化层进行了 testbench 仿真，如数据量较大的前两层、缓存窗 padding 多样化的第三层、网络分支处的卷积池化层和网络底层等。经仿真发现，卷积池化算子每层计算结果与 Pytorch 上所得结果基本一致，能够正确产生 DDR 访问地址，且能够正确应对和产生所有 AHB lite 总线信号。

然而，尽管采取了 round 截取的方法，每层的硬件计算结果与 Pytorch 推理结果仍存在较小的误差。在之后的调试中，将视误差带来的影响大小对设计进行必要的调整。

（c）NMS 算子仿真

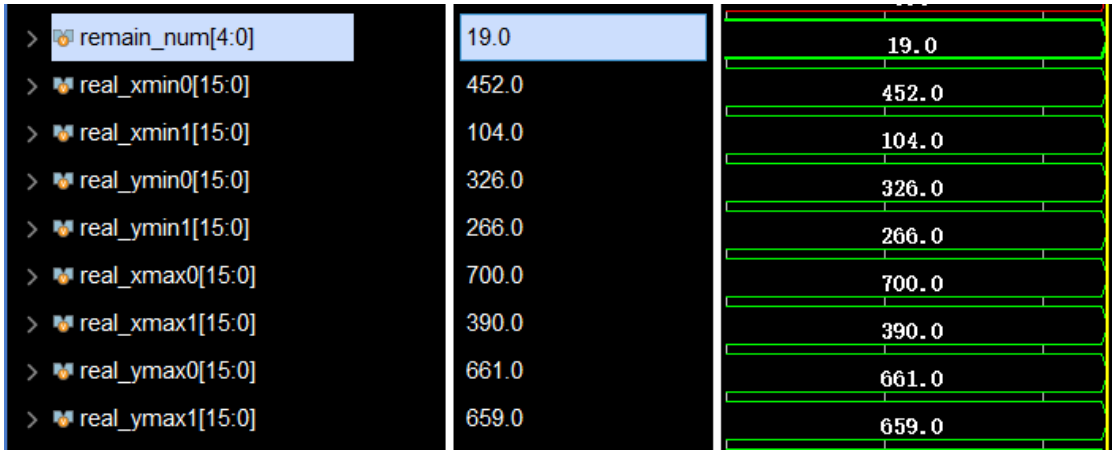


图 35 NMS 藩镇结果

```
[5952 5717 5948 5903 5889 5891 5902 5888 5890 5883 5900 5911 5901 5909
 5882 5908 5910 5880 5881]
width= 800
height= 800
xmin,ymin,xmax,ymax= [104, 266, 390, 660]
xmin,ymin,xmax,ymax= [452, 326, 700, 661]
PS D:\SSD>
```

图 36 Pytorch 上运行的 NMS 结果

4.1.2 DMAC 仿真

通过给 AHB 总线特定的信号以实现从首地址为 0x000F0000 的地址空间传输一个包含 20 个 32 位数的数据块到首地址为 0x000E0000 的地址空间。经 Vivado 仿真，DMA 工作正常。仿真波形如下：

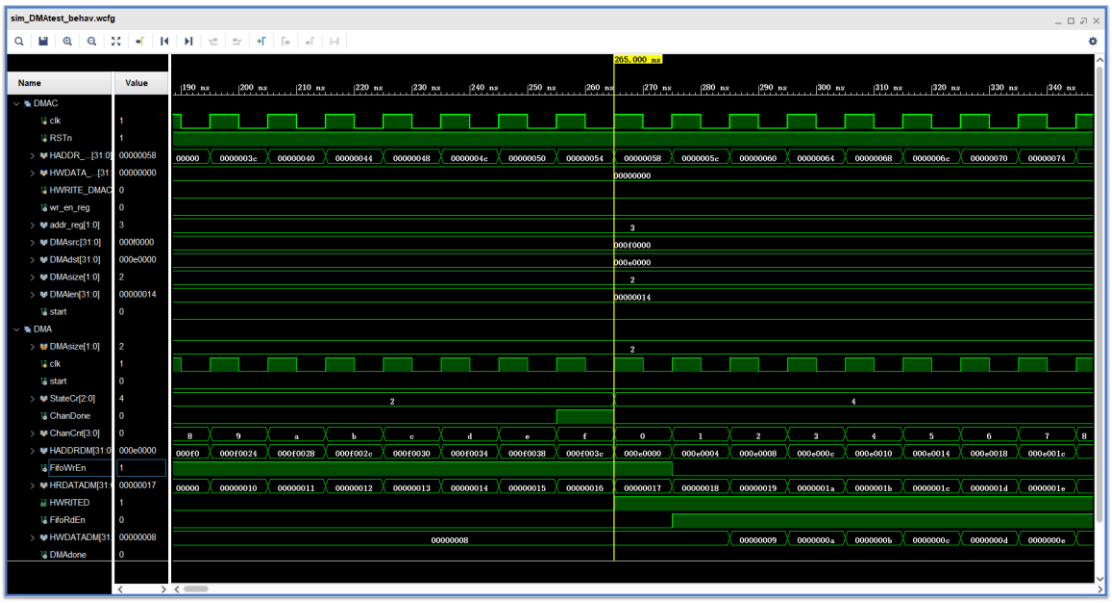


图 37 DMAC 仿真波形

4.2 测试

4.2.1 UART 测试

通过将 Retarget.c 程序移植到工程实现了 printf 和 scanf 函数重定向到 UART，从而可以配合 PC 端的 minicom 进行 UART 读写，编写 main 函数如下：

```c
#include "CortexM3.h"

int main(void) {
    printf("******************\n");
    printf("cortex-m3 startup!\n");
    printf("******************\n");
    uint32_t buf[10];
    while(1) {
        scanf("%s",buf);
        printf("Double the input string is : %s %s\n",buf,buf);
    }
    return 0;
}
```

图 38 UART 测试程序

实现的功能是在 SoC 启动时打印启动信息，并且将 uart 接收到的字符串重新发送两边，测试截图如下：

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Dec 23 2019, 02:06:26.
Port /dev/ttyS3

Press CTRL-A Z for help on special keys

******************
cortex-m3 startup!
******************
Double the input string is : 111111 111111
Double the input string is : asaaaaaa asaaaaaa
Double the input string is : 2222222 2222222
```

图 39 UART 测试结果

4.2.2 DDR 读取测试

编写汇编程序对 DDR 进行读写操作，从 DDR 首地址起写入 15 个 32 位数并将其一一读出，编写汇编程序如下：

```
Reset_Handler   PROC
                GLOBAL  Reset_Handler
                ENTRY

                LDR   R3, =0xF
READY           SUBS  R3,R3,#1
                BNE   READY

                LDR   R0, =0xA7A7A7A0
                LDR   R2, =0xF
                LDR   R1, =0x40030000           ; ddr3 addr
WRITE           STR   R0, [R1]
                ADDS  R0,R0,#1
                ADDS  R1,R1,#4
                SUBS  R2,R2,#1
                BNE   WRITE

                MOVS  R0, #0
                LDR   R2, =0xF
                LDR   R1, =0x40030000           ; ddr3 addr
READ            LDR   R0, [R1]
                ADDS  R1,R1,#4
                SUBS  R2,R2,#1
                BNE   READ

                ENDP

                ALIGN 4

                END
```

图 40　DDR 测试汇编程序

经测试，DDR 可正常工作，测试结果截图如下：

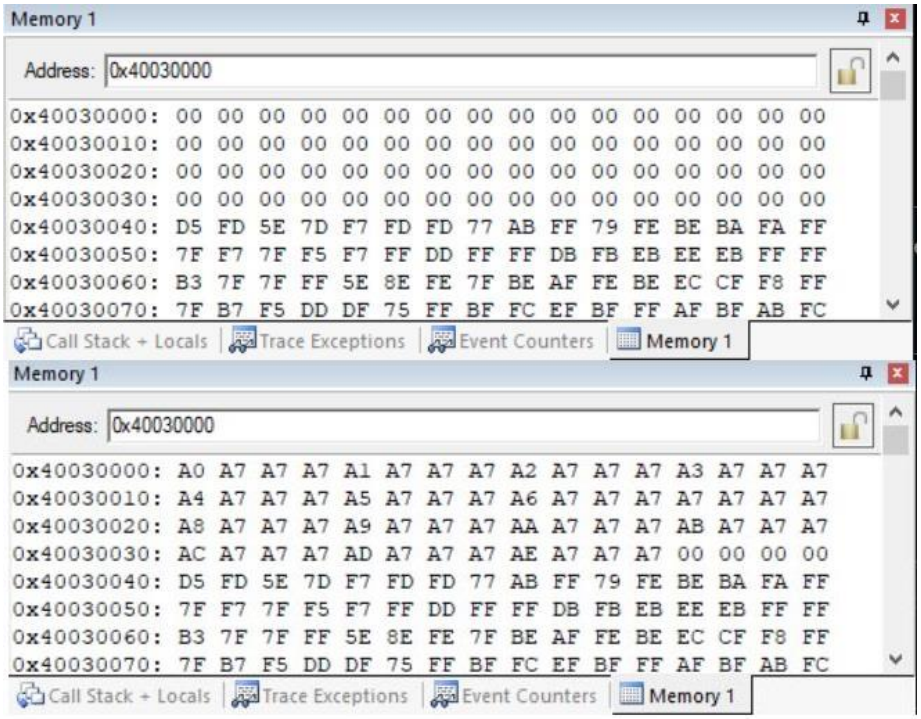


图 41　DDR 测试结果

第五章 总结

本设计提出了一种将口罩佩戴识别算法部署在前端的系统方案及其详细实现方法，由于系统结构复杂、包含组件较多，我们目前只完成了核心部分如硬件加速器（ACC）、SoC 架构、DDR 访问接口等组件的设计与编写。系统在 xc7a75 平台上获得了较高的资源利用率，其中 LUT 利用率 84.6%，BRAM 利用率 97.1%，DSP 利用率 87.2%。

我们使用 testbench 对 ACC 进行了仿真测试，并在将上述组件组装起来的基础上，编写相应的软件代码，完成了对 UART、DDR 的访问的实际测试。通过对 5 个具有代表性的卷积池化层的仿真，ACC 能够正确得出卷积池化和 NMS 运算的结果，正确率接近 100%（并非准确率），在 50MHz 时钟下推理总时间可达 0.38s；并且通过上板实测，CPU 能够正确访问 DDR、UART 和 ACC slave 和 DMAC slave。

在接下来的实测中，我们会首先测试 ACC 单层卷积池化和 NMS 计算，然后通过 PC 向 FPGA 发送一张图片，测试整个推理流程，最后增加摄像头和 LCD 模块，测试实时捕捉、检测及显示功能。希望我们能够在初赛中实现完整的系统功能、取得更好的成绩！