第四届全国大学生嵌入式芯片与系统设计竞赛

芯片设计赛道

芯片设计及评估报告

作品名称（必填）：基于智能计算的图像识别系统

队伍编号（必填）：1377

参赛队名（必填）：chenbochenbochen

日期：2021 年 10 月 2 日

一、设计规划书（总体方案）

（一）设计场景及需求

嵌入式系统一直以其广泛的应用范围为人所熟悉，软硬件集成类单片机形态是它最初的形式，它将机械和电气部件完全嵌入到受控制部件内部，满足工业电气设备、数控机床设备、自动化机械产线设备等控制操作要求。

对于神经网络加速方面，谷歌的 TPU，寒武纪的 NPU 以及 MIT 的 EYERISS 都是目前热门的解决方案，他们拥有极高的灵活度，可以实现不同的神经网络推理任务，功耗与性能双高，一般以云计算的方式提供给边缘平台。

神经网络中的图像识别任务一直是嵌入式系统的盲区，由于图像识别任务的高算力与储存需求，市面上的图像识别任务一般由云计算承担，能够负担起图像识别任务的嵌入式系统也往往成本高昂，难以满足生产实践中的成本要求。

经过调查分析，图像识别任务一般采用 CNN 神经网络（卷积神经网络）进行实现，CNN 神经网络的大部分计算消耗在反复的乘加运算和有符号数比较运算上，针对这两种运算进行优化，能够极大提高边缘设备的神经网络计算能力。

（二）设计目的及关键指标

为了解决如今嵌入式系统处理图像识别任务方向的缺失，我们实现了一个通用的嵌入式处理图像识别任务的软硬件开发平台(嵌入式硬件系统+神经网络开发套件），全新的针对 CNN 神经网络运算指令集以及其硬件实现，使得此软硬件系统可以支持任意以乘加运算为主体的神经网络加速任务。

我们的设计包括：指令集，用以实现指令集的微架构，编译器，SOC 设计。

指令集的关键指标：

1. 指令集需要对于传统的 CNN 神经网络完备，能够独立处理绝大部分运算而不需要 MCU 参与，能够独立运行全连接，卷积，池化，ReLU 运算。使得大部分时间可以脱离 MCU 进行运算，与 MCU 进行充分解耦，使得移植 IP 更加方便。
2. 精简，减轻译码逻辑负担，同时方便编译器进行编译，可以使硬件的预期频率更高。

IP 微架构的关键指标：

1. 对于指令集的完整实现。
2. 充分合理的 retiming，使得 IP 可以跑在更高的频率，便于移植到其他平台。
3. 对面积的把握，由于产品定位于低端嵌入式设备的神经网络加速器 IP，更小的面积会具有更大的竞争力。

编译器的关键指标：

封装更多的深度学习算子，方便开发者对我们的作品进行软件开发。

SOC 的关键指标：

1. 使用开源的充分验证的 MCU，免去 MCU 的验证工作，并且免去 IP 授权费，增加产品的商业价值。
2. SOC 在满足功能的前提下面积应尽可能小，增加产品的商业价值。

（三）设计内容简述

指令集：

1. 一套针对 CNN 神经网络优化的累加器型指令集，可以实现卷积，全连接，池化，ReLU 激活等基于神经网络算子（详见附录三）

指令集结构类型	优点	缺点
堆栈型	是一种表示计算的简单模型；指令短小。	堆栈不能被随机访问，从而很难生成有效代码。同时，由于堆栈是瓶颈，所以很难被高效地实现。
累加器型	减小了机器的内部状态；指令短小。	由于累加器是唯一的暂存器。
寄存器型	是代码生成最一般的模型。	所有操作数均需命名，且显式表示，因而指令比较长。

表 1：指令集架构类型

2. 基于自制指令集的一套回归测试集，在此测试集的基础上对微架构进行功能仿真

微架构：

1. 利用新兴敏捷硬件描述语言 Chisel 联合 verilog 对自制指令集的完整实现，对 Chisel 生成 verilog→利用 ASIC 设计工具进行仿真的流程进行探索。
2. 计算模块十级流水，充分提高运行频率。
    1. 利用开源 SoC PicoRV32，此 SoC 多次流片，并且频率极高。
    2. 使用八位定点数进行运算，对比浮点数推理，面积频率都有明显降低。

编译器：

1. 一套和指令集一一对应的汇编语法，完整的汇编器，加速 IP 开发者的开发速度。
2. 封装了部分深度学习算子。

二、算法实现说明

由于指令集式加速器灵活度很高，可以实现任意基于乘加与比较的神经网络，

我们为了演示效果，制作了一些列 Demo 级别的软件系统，其中包含单隐层神经网络实现的的手写数字识别，卷积运算，下面以深层 BP 神经网络实现的图像识别进行举例具体说明如何将神经网络算法部署到我们的设计中。

（一）算法原理概述

推理算法：

BP 神经网络是一种按误差反向传播(简称误差反传)训练的多层前馈网络，其算法称为 BP 算法，它的基本思想是梯度下降法，利用梯度搜索技术，以期使网络的实际输出值和期望输出值的误差均方差为最小。

基本 BP 算法包括信号的前向传播和误差的反向传播两个过程。即计算误差输出时按从输入到输出的方向进行，而调整权值和阈值则从输出到输入的方向进行。正向传播时，输入信号通过隐含层作用于输出节点，经过非线性变换，产生输出信号，若实际输出与期望输出不相符，则转入误差的反向传播过程。误差反传是将输出误差通过隐含层向输入层逐层反传，并将误差分摊给各层所有单元，以从各层获得的误差信号作为调整各单元权值的依据。通过调整输入节点与隐层节点的联接强度和隐层节点与输出节点的联接强度以及阈值，使误差沿梯度方向下降，经过反复学习训练，确定与最小误差相对应的网络参数(权值和阈值)，训练即告停止。此时经过训练的神经网络即能对类似样本的输入信息，自行处理输出误差最小的经过非线形转换的信息。

量化算法：

在深度学习中，量化指的是使用更少的 bit 来存储原本以浮点数存储的 tensor，以及使用更少的 bit 来完成原本以浮点数完成的计算。这么做的好处主要有如下几点：

1. 更少的模型体积，接近 4 倍的减少；

2. 可以更快的计算，由于更少的内存访问和更快的 int8 计算，可以快 2~4 倍。

Pytorch 对量化的支持目前有三种方式：

1. Post Training Dynamic Quantization，模型训练完毕后的动态量化；

2. Post Training Static Quantization，模型训练完毕后的静态量化；

3. QAT（Quantization Aware Training），模型训练中开启量化。

我们采用后静态量化，相对比其他两个，静态量化可以保证在整个推理过程中始终采用 int8 类型，减少数据传送开销。静态量化通过添加一个 observer 对输入输出进行预处理后对每一层添加 scale、zero 值，对放缩指数进行反复变换，始终保持输入在 int8 范围，同时通过 Min-Max 来计算出最合适的 scale 和 zero。

(二) 算法框架简述

本部分主要介绍采用的算法在所提出的设计场景下计算流程是怎样的，包括输入输出的数据结构、算法的流程图等。

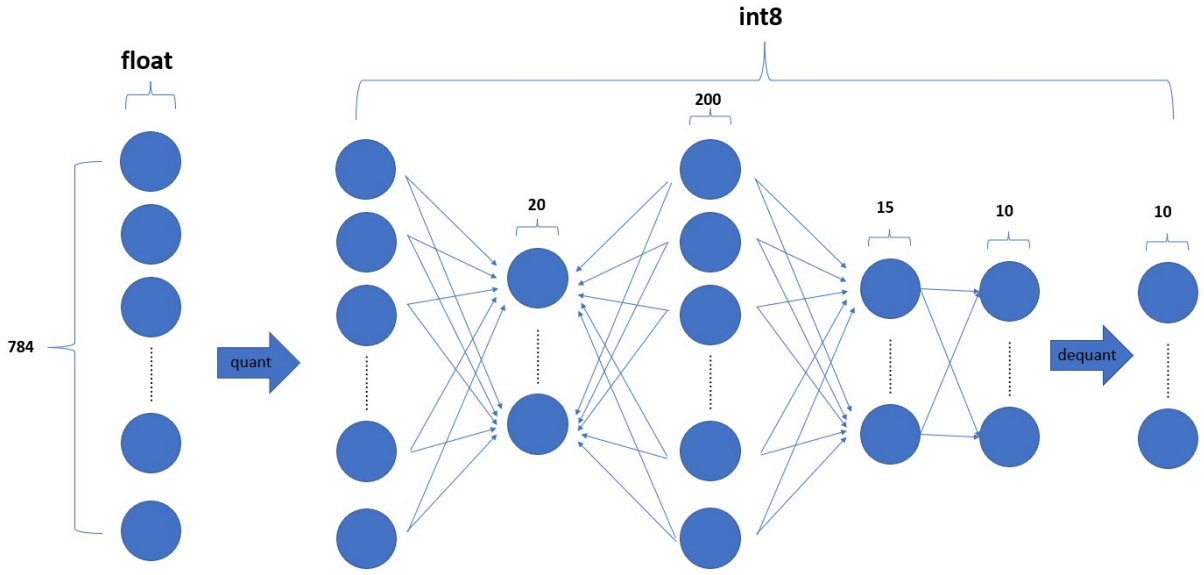


图 1：BP 神经网络

（1）首先我们将一张 28*28 的图片进行展开成为 784 的线性数据，然后通过 quant 对 784 个 float 进行第一次量化成为 784 个 int8 数据

（2）通过 784*20 个参数将 784 神经元推理到 20 神经元

（3）将 20 神经元扩展到 200 神经元

（4）将 200 神经元推理至 15 神经元

（5）将 15 神经元推理至 10 神经元

（6）根据最后一层的 scale 和 zero 将神经元的结果反量化回 float 类型，此时 10 个数中最大的值对应的就是推理出的结果

（三） 算法实现方案

本部分介绍具体的算法实现（如人工神经网络的具体参数等）方案。

首先我们给出 float 和 int8 的映射关系，用 f 来表示 float，i 来表示 int8，

$$f = S\ (\ i - Z\ ) \tag{1}$$

$$i = round(\ \frac{i}{S}\ +\ Z\ ) \tag{2}$$

$$S = \frac{i_{max} - i_{min}}{f_{max} - f_{min}} \tag{3}$$

$$Z = round(i_{max} - \frac{r_{max}}{S}) \tag{4}$$

矩阵运算的量化：

假设 f_1 和 f_2 是两个 N × N 的矩阵，f_3 是 f_1 和 f_2 相乘后的矩阵，

$$f_3^{i,k} = \sum_{j=1}^{N} f_1^{i,j} f_2^{j,k} \tag{5}$$

$$S_3\left(i_3^{i,k} - Z_3\right) = \sum_{j=1}^{N} S_1\left(i_1^{i,j} - Z_1\right) S_2\left(i_2^{j,k} - Z_2\right) \tag{6}$$

通过整理得：

$$i_3^{i,k} = \frac{S_1 S_2}{S_3} \sum_{j=1}^{N} \left(i_1^{i,j} - Z_1\right)\left(i_2^{j,k} - Z_2\right) + Z_3 \tag{7}$$

仔细观察（7）式可以发现，除了$\frac{S_1 S_2}{S_3}$，其他都是定点整数运算。那如何把$\frac{S_1 S_2}{S_3}$也变成定点运算呢？这里要用到一个 trick。假设 $M = \frac{S_1 S_2}{S_3}$，由于 M 通常都是(0, 1) 之间的实数 (这是通过大量实验统计出来的)，因此可以表示成 M= 2-nM0，其中 M0 是一个定点实数。注意，定点数并不一定是整数，所谓定点，指的是小数点的位置是固定的，即小数位数是固定的。因此，如果存在 M=2-n M0，那我们就可以通过 M0 的 bit 位移操作实现 2-n M0，这样整个过程就都在定点上计算了。

那么如何找到 M0 和 n 的值呢，我们通过下面的算法来计算：

```
def multiply(n, M, P):
    result = M * P
    Mo = int(round(2 ** n * M)) # 这里不一定要四舍五入截断，因为 python 定点数不好表示才这样处理

    approx_result = (Mo * P) >> n
    print("n=%d, Mo=%d, approx=%f, error=%f"%\
          (n, Mo, approx_result, result-approx_result))

    for n in range(1, 16):
    multiply(n, M, P)
```

在 error 小于 1 时我们认为误差已经收敛。

加法的量化：

通常在神经网络的推理中公式为：

$$a = \sum_i^N w_i x_i + b \tag{8}$$

通过公式（1）（2）我们可以得到：

$$S_a(i_a - Z_a) = \sum_i^N S_w(i_w - Z_w) S_x(i_x - Z_x) + S_b(i_b - Z_b) \tag{9}$$

$$i_a = \frac{S_w S_x}{S_a}\sum_i^N (i_w - Z_w)(i_x - Z_x) + \frac{S_b}{S_a}(i_b - Z_b) + Z_a \quad (10)$$

这里面非整数的部分就只有$\frac{S_w S_x}{S_a}$、$\frac{S_b}{S_a}$，因此接下来需要把这部分也变成定点运算。

对于 bias，由于$\sum_i^N (\mathrm{i}_w - Z_w)(\mathrm{i}_x - Z_x)$ 的结果通常用 int32 的整数存储，因此 bias 通常也量化到 int32。这里我们可以直接用$S_w S_x$来代替S_b，由于S_w、S_x都是对应 8 个 bit 的缩放比例，因此$S_w S_x$ 最多就放缩到 16 个 bit，用 32bit 来存放 bias 绰绰有余，而Z_b 则直接记为 0。

三、系统架构设计

(一) 芯片架构图

SOC 架构：

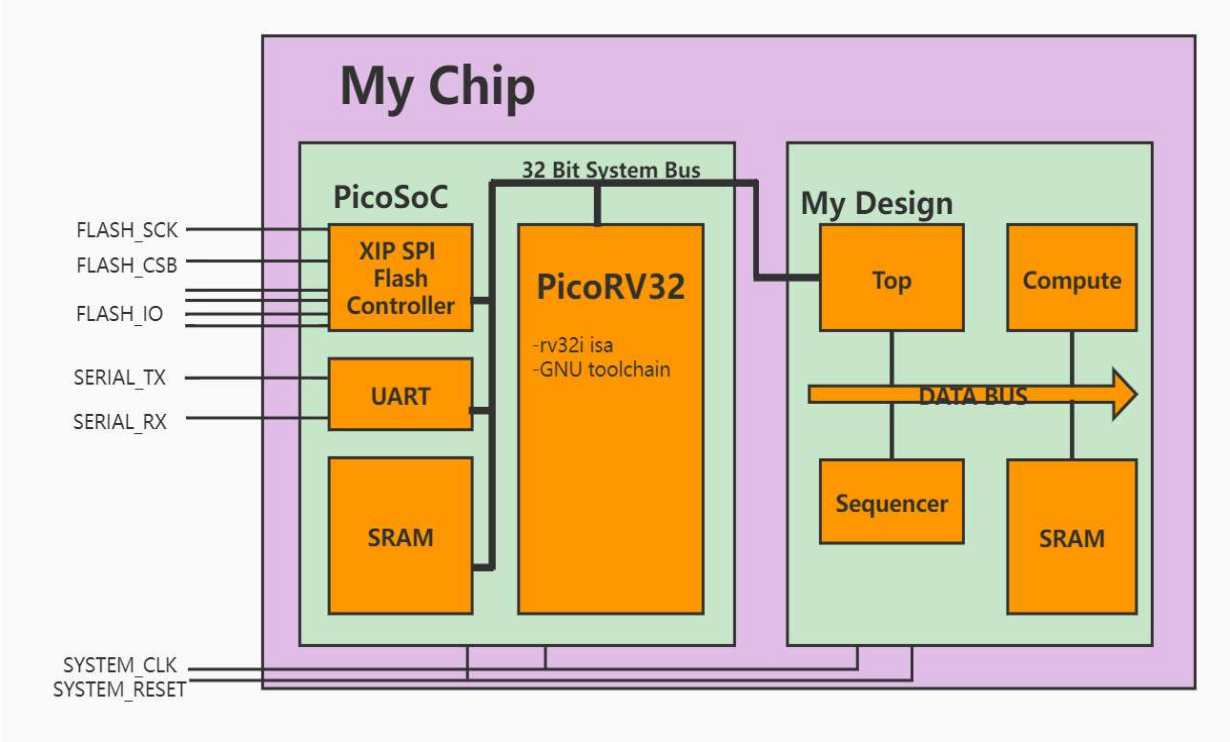


图 2：SoC 整体架构

PicoSoC 使用了基于 PicoRV32 内核实现的开源 SoC，有多次流片经历，以极低的面积与在 FPGA 上可达 500MHz 的表现所闻名，其中包含 PicoRV32 核

心，UART 控制器，SPI Flash 控制器，默认为 1kB 的 SRAM 储存空间。（具体信息可见 PicoSoC.pdf 与 PicoRV32.pdf，后文不再赘述）

My Design 是我们实现的神经网络加速器，里面分别包含 Top，Sequencer，Compute，Memory 模块，分别实现与 PicoRV32 的数据交换（使用 32 位 System Bus）/访存仲裁，指令的派遣，指令的计算，SRAM 的控制。

本部分以框图的形式介绍芯片架构的构成（主要展示子模块交互关系），顶层接口信号说明，包括哪些主要子模块，简述其功能、接口。

（二）Top 模块说明

1．Top 模块详细介绍

Top 为 sequencer 和 compute 提供访存仲裁服务并且支持与外部交互

外部数据传递

这里是加速器 IP 与 PicoRV32 进行数据传递，利用 32 位 System Bus 实现对加速器 IP 主存的交互以及对加速器的控制，包括启动加速器，暂停加速器，访问加速器运行状态。

主存仲裁

主存仲裁在加速器中，可能会有三种对主存的访问请求，Sequencer 请求，Compute 请求，Top 请求（如上，外部对主存的读写）。由于 Top 和 Sequencer 与 Compute 存在顺序关系，若先处理 Sequencer 或 Top 请求，可能会引起死锁，于是构造一个优先级仲裁，Compute > Sequencer & Top，Top 和 Sequencer 不存在互斥关系，轮流进行访存，防止一方一直抢占主存，请求将由请求方一直保持到请求完成，经过一些测试用例表明，这样的仲裁在一些大部分测试用例中都可以得到较优越的性能表现，访存仲裁状态机如下。

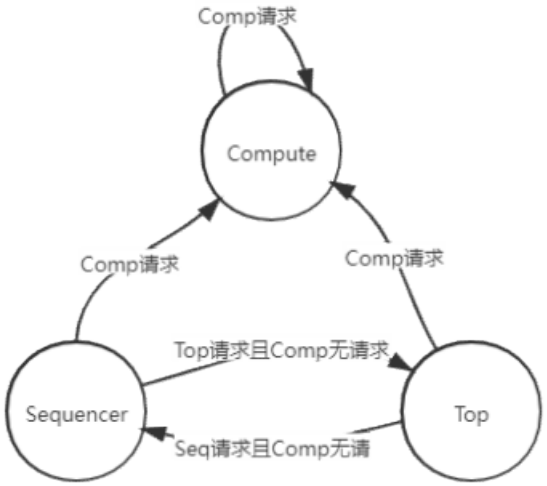


图 3：主存访存状态机

2．Top 模块接口定义

信号名	信号位宽	方向	信号功能描述	备注
clock	1bit	输入	时钟输入	
resetn	1bit	输入	复位输入	低有效
valid	1bit	Pico 输入	事务请求	Pico System Bus
ready	1bit	输出至 Pico	请求完成	Pico System Bus
wstrb	4bit	Pico 输入	写掩码	Pico System Bus
addr	32bit	Pico 输入	地址	Pico System Bus
wdata	32bit	Pico 输入	写数据	Pico System Bus
rdata	32bit	输出至 Pico	读数据	Pico System Bus

3．Top 模块接口时序

接口采用阻塞式设计，单次事务未完成不允许发起下一次事务。

读事务

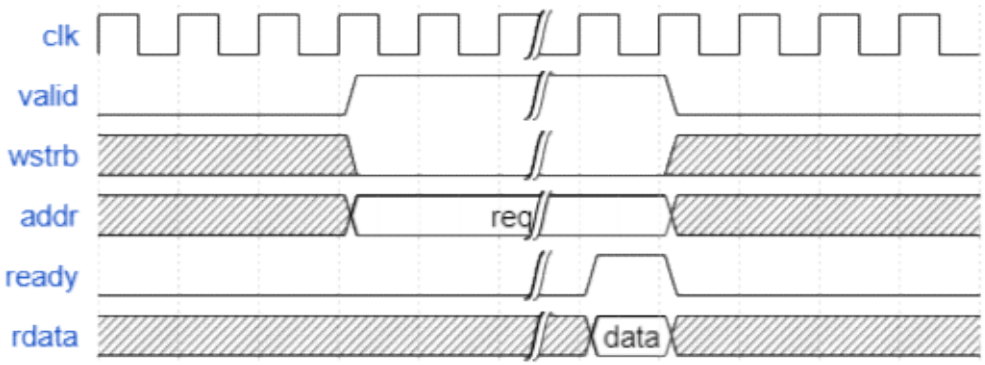


图 4：读事务时序图

wstrb 保持为 0 时 master 将 valid 拉高即为一次读事务，要求 valid 与 addr 一直有效直到 ready 有效，ready 有效一周期，即完成读事务，rdata 中数据即为读反馈数据。

写事务

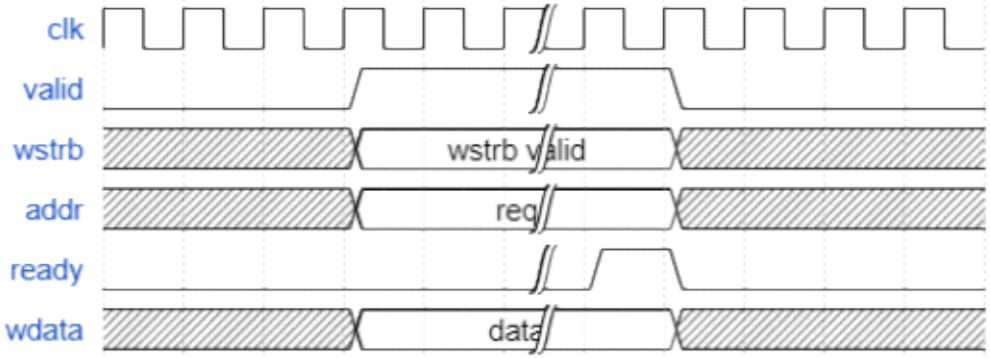


图 5：写事务时序图

wstrb 保持不为 0 时 master 将 valid 拉高即为一次写事务，wstrb 为写掩码，要求 valid，addr，wdata，wstrb 一直有效直到 ready 有效，ready 有效一周期，即完成写事务，表示写完成。

（三）Sequencer 模块说明

1．Sequencer 模块详细介绍

Sequencer 负责计算指令的派遣与跳转指令的执行。

Sequencer 执行跳转指令

由于神经网络运算是确定性的，不存在分支跳转，可以只实现调用和返回指令，方便程序员进行编程，为了实现 CALL (调用)，RETURN (返回)，Sequencer 实现了硬件堆栈隐式执行跳转指令，CALL 指令通过压栈当前地址，并改变 PC 为目标地址；RETURN 指令通过出栈栈顶地址，并跳转至栈顶地址进行实现，可以完成单周期跳转。

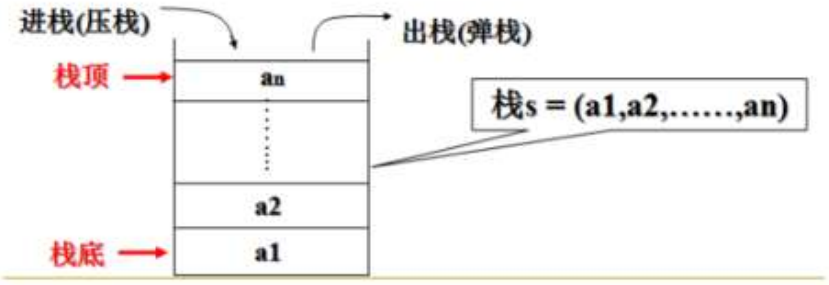


图 6：堆栈示意图

Sequencer 计算指令派遣

由于神经网络运算是确定性的，不存在分支跳转，指令流的流向是可预测的，可以利用一个指令队列预取指，在空闲时进行取指令，繁忙时释放主存，使主存的利用率更加充分。

指令派遣分为两种情况，普通指令和连续指令，通过 Sequencer 内建 mini decoder 进行判断。指令派遣逻辑图如下：

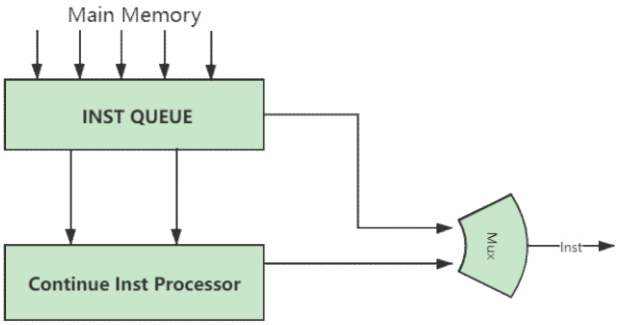


图 7：Sequencer 指令派遣

指令队列从主存中取指令，先判断是否是连续指令。如果是，则由 Continue Inst Processor 进行指令发生，产生连续指令输出至 Compute，否则直接将指令队列中的数据传递至 Compute。

2．Sequencer 模块接口定义

信号名	信号位宽	方向	信号功能描述	备注
clock	1bit	输入	时钟输入	
reset	1bit	输入	复位输入	高有效
start	1bit	Top 输入	开始运行	
addr	16bit	Top 输入	从 addr 开始运行	
busy	1bit	输出至 Top	正在派遣指令	
smem_valid	1bit	输出至 Top	访存 valid	访存接口
smem_ready	1bit	Top 输入	访存 ready	访存接口
smem_addr	16bit	输出至 Top	访存地址	访存接口
smem_data	1bit	输出至 Top	读数据	访存接口
comp_valid	1bit	输出至 Compute	指令 valid	指令接口
comp_ready	1bit	Compute 输入	指令 ready	指令接口
comp_insn	32bit	输出至 Compute	指令	指令接口

3．Sequencer 模块接口时序

Sequencer 的唤起

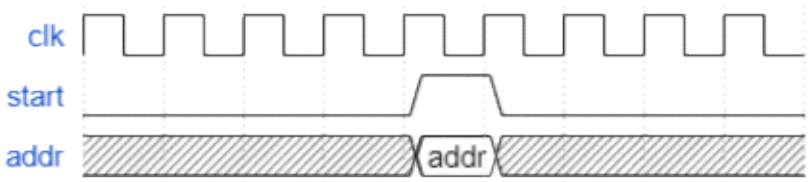


图 8：Sequencer 唤起的时序

start 有效时，置 addr，Sequencer 便开始以 addr 为起始地址顺序从 Memory 中取指令执行。

Sequencer 访存

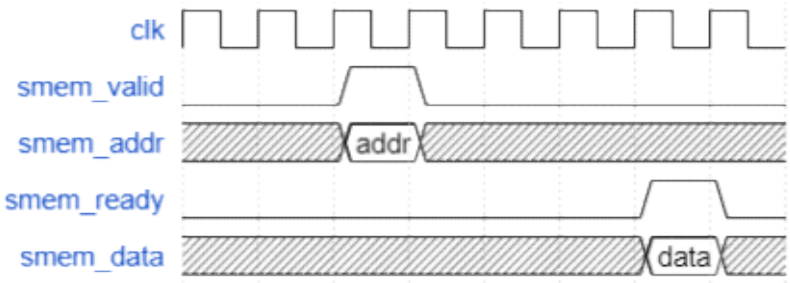


图 9：Sequencer 访存的时序

valid 保持一周期，发起一次读事务，ready 被拉高则结束一次读事务，data 为读数据。

Sequencer 指令派遣

指令派遣分为两种情况，普通指令与连续指令，连续指令包括 Execute，ContinueLoad 指令。（参考附录三）

- 普通指令

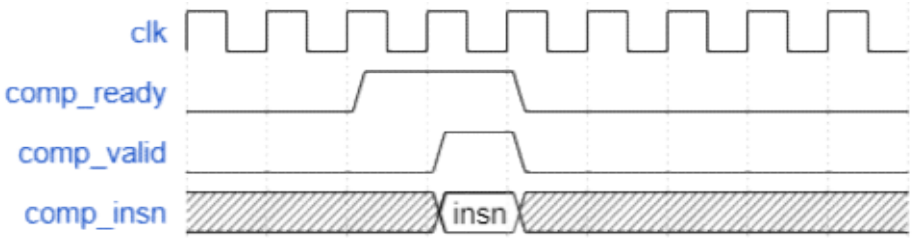


图 10：Sequencer 普通指令时序

- 连续指令

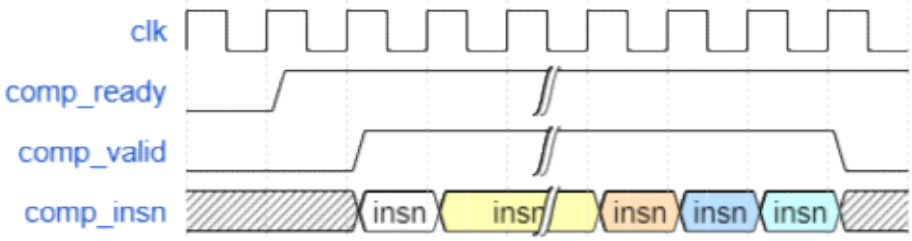
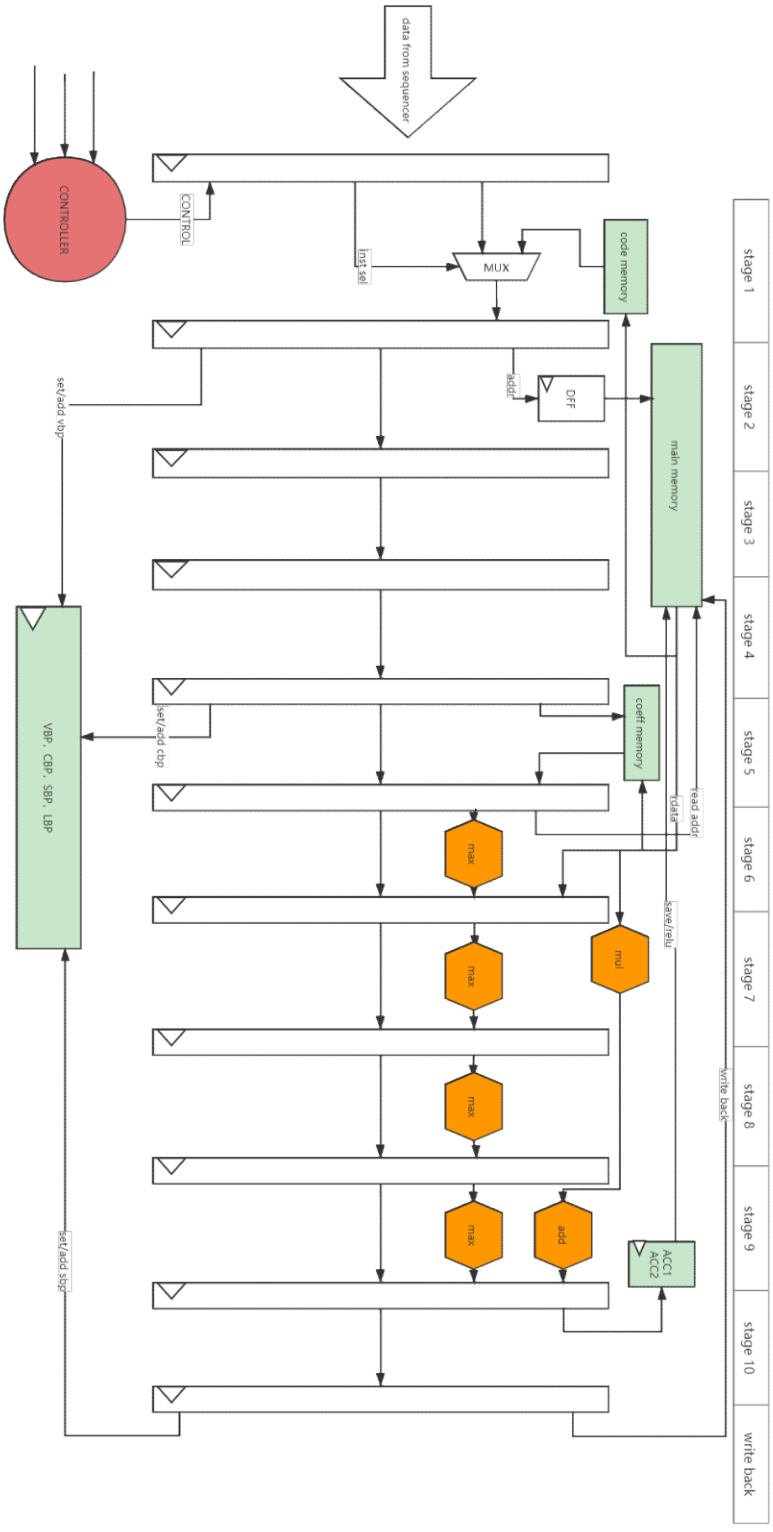


图 11：Sequencer 连续指令时序

（四）Compute 模块说明

1．Compute 模块详细介绍

Compute 是计算指令的执行单元，负责处理除了跳转以外的所有指令，利



用静态十级流水线进行处理，原理图如下：

图 12：Compute 原理图

Compute 实现了指令集(详见附录三)中所定义的两个累加器 ACC0，ACC1；代码专用储存器 Code Memory，参数专用储存器 Coeff Memory0，Coeff Memory1，基址寄存器 SBP，VBP，LBP，CBP，并且利用十级流水线以达到更高的频率。

流水线冒险处理机制

由于流水线深度较深，可能会出现的冒险较多，出于频率考虑，采用最简单的解决冒险机制，分别如下：

Code Memory 和 Coeff Memory0/1 的写后读冒险采用插入同步指令（Sync)，具体做法为在 LoadCode/LoadCoeff0/LoadCoeff1 后插入 3 条 Sync 指令。

多条指令对主存占用产生的结构冒险，在指令集中，大多指令都会存在访存操作，这里采用集中式仲裁的结构，在流水线前端对结构冒险进行预测并且阻塞流水线（Stall）。

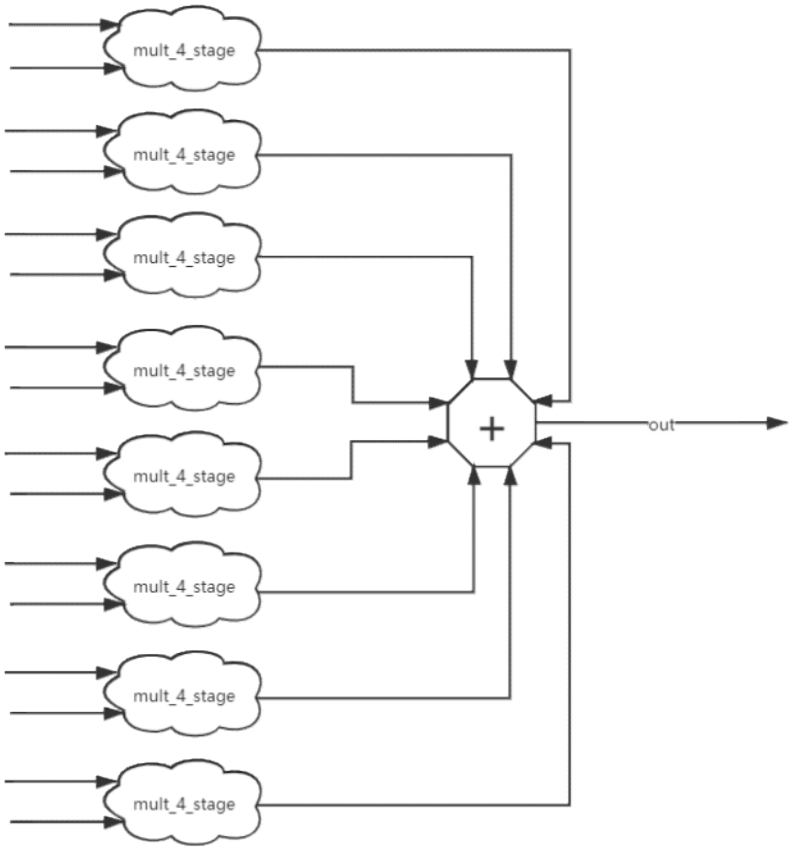
MAX 操作需要对 ACC0 进行访问所造成的写后读冒险，同上一条，在流水线前端进行预测并且阻塞流水线。

虽然使用了阻塞式的结构会影响性能，但是通过合理的软硬协同开发，事实证明，大部分时间可以实现 IPC（Instruction Per Cycle）接近 1 的性能指标。（详见仿真部分）

计算模块

Compute 所做的算数运算主要有 Int8 的乘法与 8 个 Int8 的比较运算，为

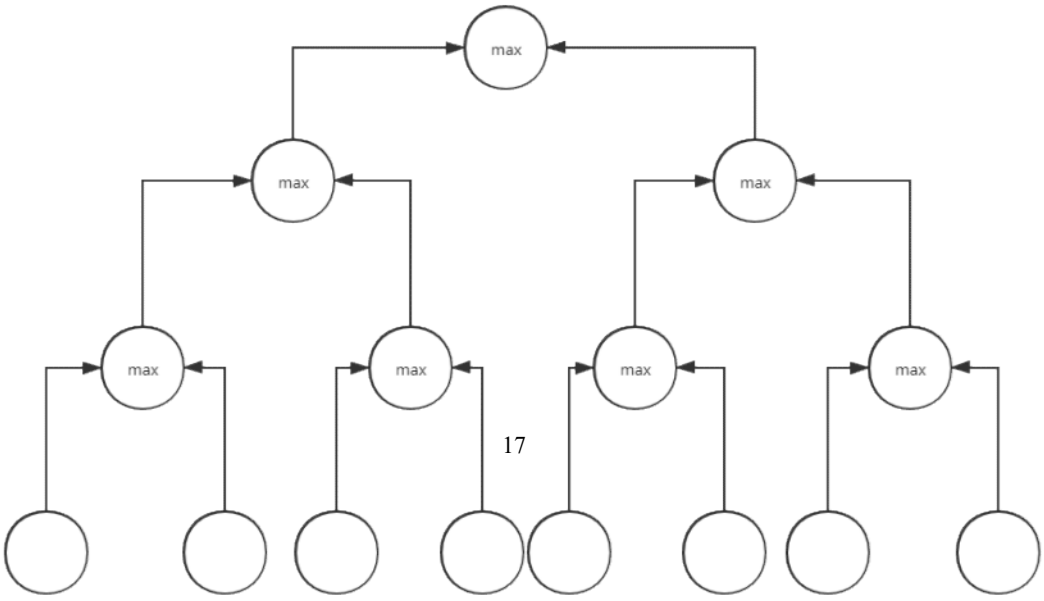
了提高流水线的频率，上述运算都采用了流水线结构，比较运算使用树状结构，



乘法通过调用 DesignWare 的 DW02_mult_4_stage IP 来实现。结果可以打达到单周期实现 2 次 8 个 Int8 的乘累加运算和 1 次 8 个 int8 中取最大运算。

图 13：乘加运算示意图

图 14：有符号数比较示意图



2．Compute 模块接口定义

信号名	信号位宽	方向	信号功能描述	备注
clock	1bit	输入	时钟输入	
reset	1bit	输入	复位输入	高有效
busy	1bit	输出至 Top	正在计算	
cmd_valid	1bit	Sequencer 输入	指令 valid	指令接口
cmd_ready	1bit	输出至 Sequencer	指令 ready	指令接口
cmd_insn	32bit	Sequencer 输入	指令	指令接口
mem_ren	1bit	输出至 Top	读有效	访存接口
mem_wen	8bit	输出至 Top	写有效	访存接口
mem_addr	16bit	输出至 Top	访存地址	访存接口
mem_wdata	64bit	输出至 Top	写数据	访存接口
mem_rdata	64bit	Top 输入	读数据	访存接口
tick_simd	1bit	输出至 Top	simd 指令	性能测试接口
tick_nosimd	1bit	输出至 Top	非 simd 指令	性能测试接口

3．Compute 模块接口时序

Compute 指令接口

指令接口的时序与 Sequencer 指令接口时序相同，参考前文。

Compute 访存接口

读事务

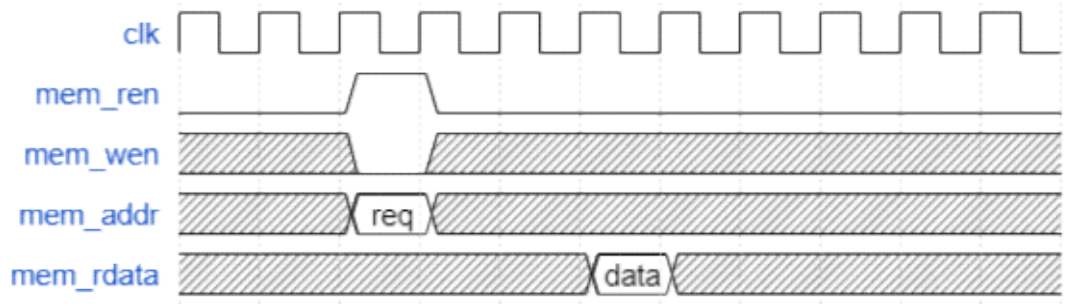


图 15：读事务时序图

由于 Compute 访存的优先级最高，所以不会产生访存阻塞，读事务会在第三个周期返回结果。

写事务

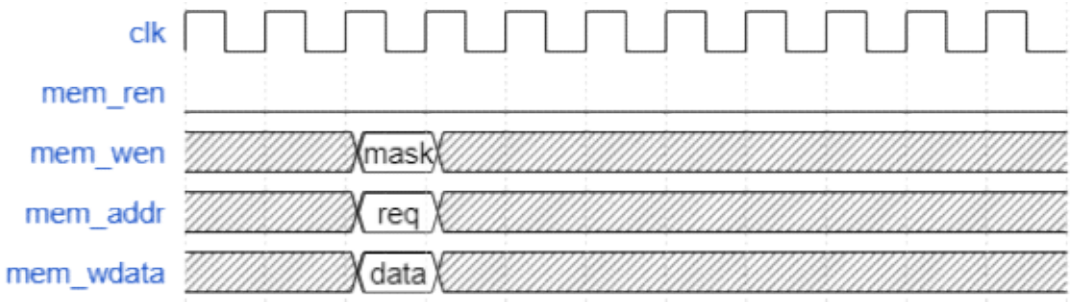


图 16：写事务时序图

与读事务类似，会在 3 个周期后写完成。

四、系统工作模式

（一）工作模式

上传网络模型

将网络模型通过 32bit System Bus 从 Pico 上传到加速器，如果网络模型较小，代码+数据小于 128kB，那么只需要上电时上传一次，后续只需要上传数据即可。

上传数据

对 Pico 进行编程的程序员与对神经网络加速器 IP 进行编程的程序员进行协商，规定神经网络加速器 IP 的程序运行起点与输入数据入口，Pico 先对外设拍摄出的图片进行预处理，预处理为 int8 的大小为 28x28 的灰度图，然后上传至 IP，等待 IP 进行下一步处理。

执行指令

规定神经网络加速器 IP 的执行指令的起始地址，顺序执行，进行推理后将结果存放在与对 Pico 编程的程序员协商的数据出口，等待取数据。

下载结果与结果处理

Pico 通过 32bit System Bus 从 IP 下载回 Pico，进行后续处理，在我们完成的 Demo 级软件系统 (图像识别系统) 中，后续处理为找出 8bit 数据的 Index of Max，最终得到识别结果。

（二）工作流程（图）介绍

我们的 SoC 有极强的灵活性与可配置性，以下是我们推荐的开发/运行流程。

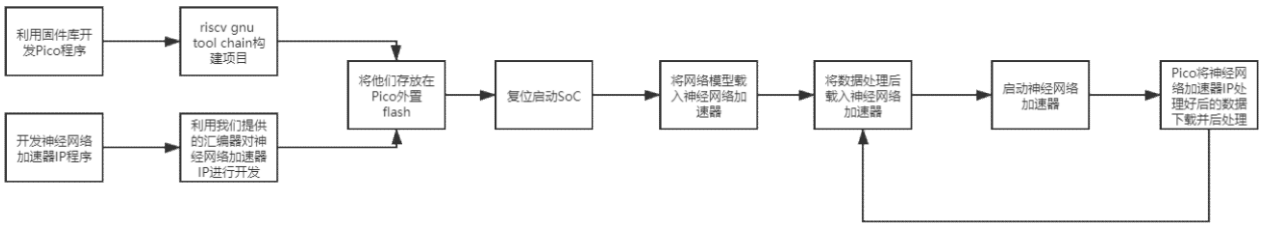


图 17：开发/运行流程

各工作模式下的仿真波形

上传模型与数据

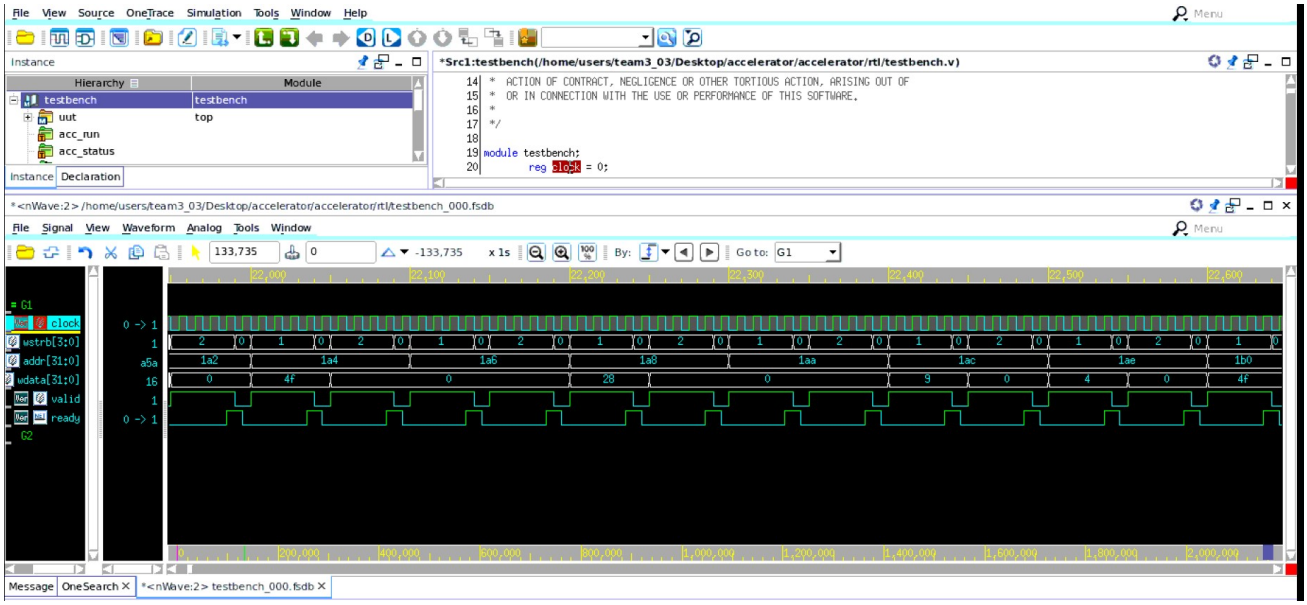


图 18：上传模型与数据波形

如图是 Pico 通过 System Bus 将神经网络模型与输入数据传入神经网络加速器 IP。

进行计算

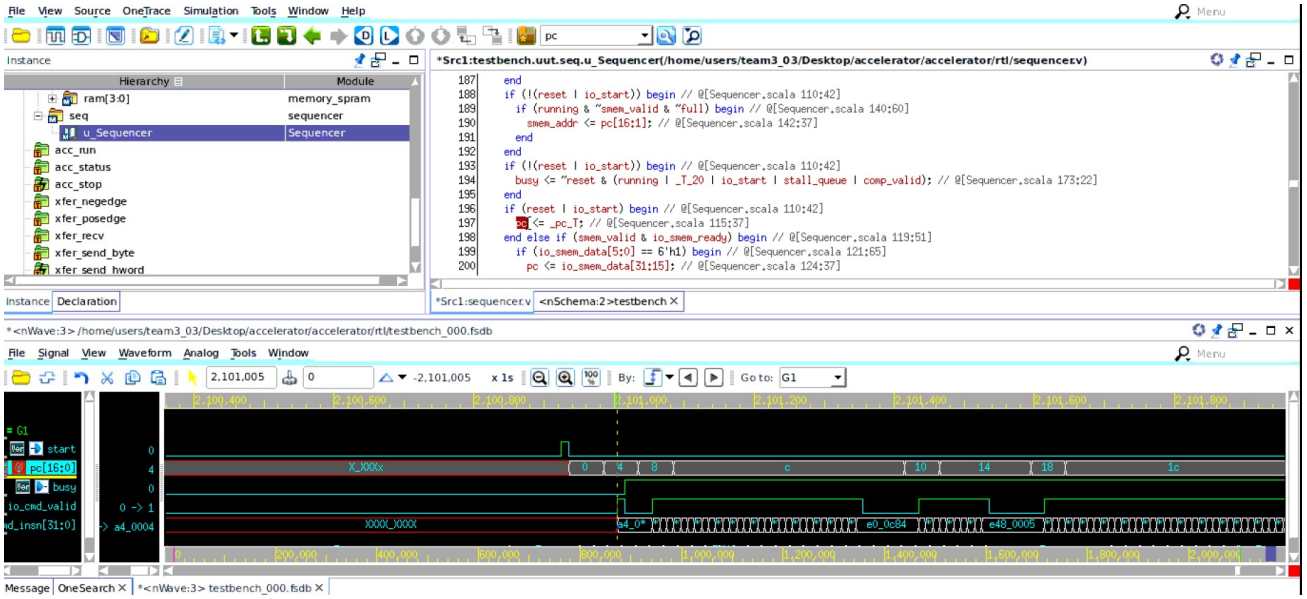


图 19：单个指令派遣波形

在 Pico 通过寄存器配置神经网络加速器 IP 进行运算后，start 信号有效 1 一周期，若干周期后，Sequencer 向 Compute 传递指令，PC（程序计数器）不断按每个指令 4 字节递增。

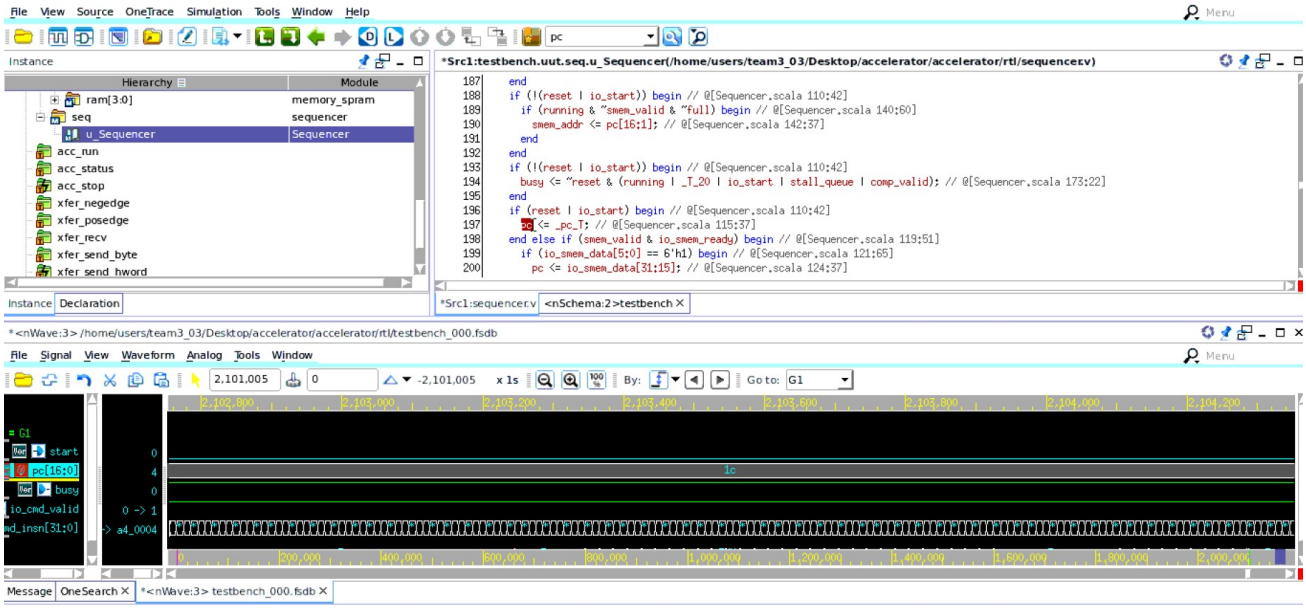


图 20：连续指令派遣波形

连续指令派遣，ContinueLoad，Execute 指令单个指令多次派遣，PC 不改变，insn 不断改变。

下载数据

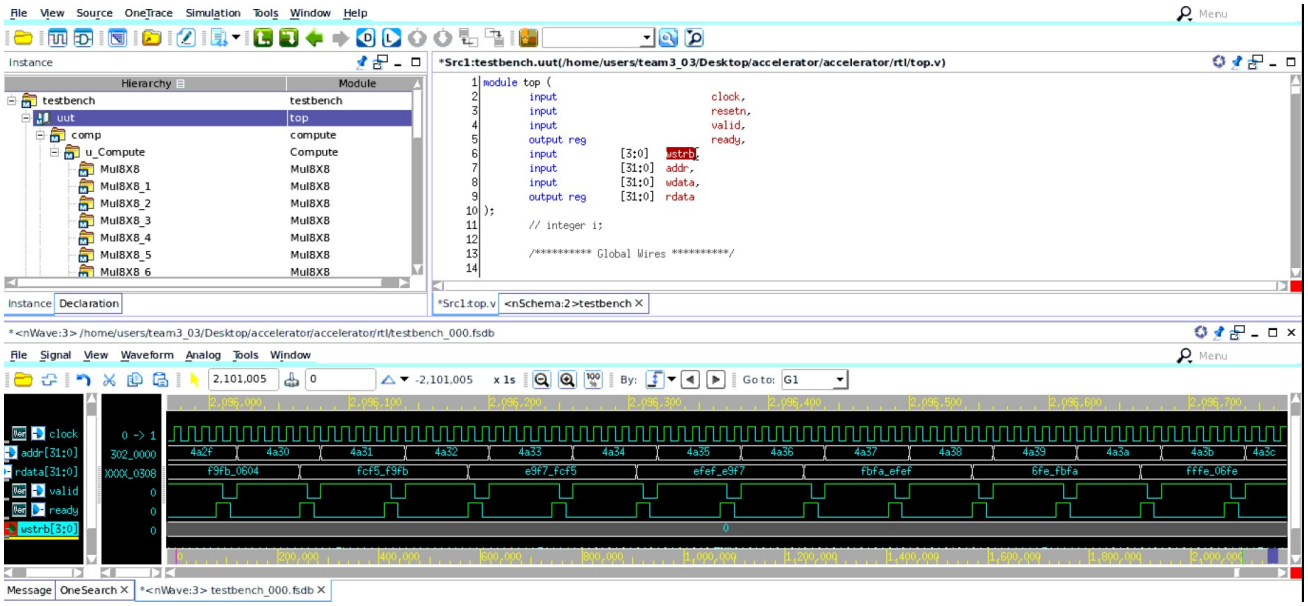


图 21：下载数据的波形

Pico 通过 System Bus 从神经网络加速器 IP 不断下载输出结果。

五、寄存器列表

（一）SoC 地址映射

Address Range	Description
0x00000000 .. 0x00FFFFFF	Pico 内嵌 SRAM
0x01000000 .. 0x01FFFFFF	外置 Flash
0x02000000 .. 0x02000003	SPI Flash 配置寄存器
0x02000004 .. 0x02000007	UART 时钟分频计数器
0x02000008 .. 0x0200000B	UART TX RX
0x03000000 .. 0x0301FFFF	神经网络加速器 128kB 主存主存
0x03020000	神经网络加速器控制寄存器

（二）神经网络加速器 IP 控制寄存器（可写不可读）

0x03020000	Bit31~Bit16	Bit15~Bit62	Bit1	Bit0
功能	配合 Bit0，指令起始地址	Reserved	停止运行	开始运行

（三）神经网络加速器 IP 状态寄存器（可读不可写）

Bit31~Bit0	功能
0xFFFFFFFF	busy
0x00000000	Idle
0x03020000	status

（四）SPI 配置寄存器（Copy From PicoSoC）

Bit(s)	Description
31	MEMIO Enable (reset=1, set to 0 to bit bang SPI commands)
30:23	Reserved (read 0)
22	DDR Enable bit (reset=0)
21	QSPI Enable bit (reset=0)
20	CRM Enable bit (reset=0)
19:16	Read latency (dummy) cycles (reset=8)
15:12	Reserved (read 0)
11:8	IO Output enable bits in bit bang mode
7:6	Reserved (read 0)
5	Chip select (CS) line in bit bang mode
4	Serial clock line in bit bang mode
3:0	IO data bits in bit bang mode

六、作品分析与评估

```
Number of clock cycles:                  4668
Number of SIMD-instructions:              794
Number of non-SIMD-instructions:         3874

Avg. ops per cycle:              3.55
Avg. simd ops per cycle:         2.72

Avg. insn per cycle:             1.00
Avg. simd insn per cycle:        0.17
```

图 22：Benchmark

运行手写数字识别任务，fasion MNIST 训练集下利用 Torch 训练/fasion MNIST 验证集验证,数据仅需要使用 torch.nn.quant 函数进行定点化处理即可。

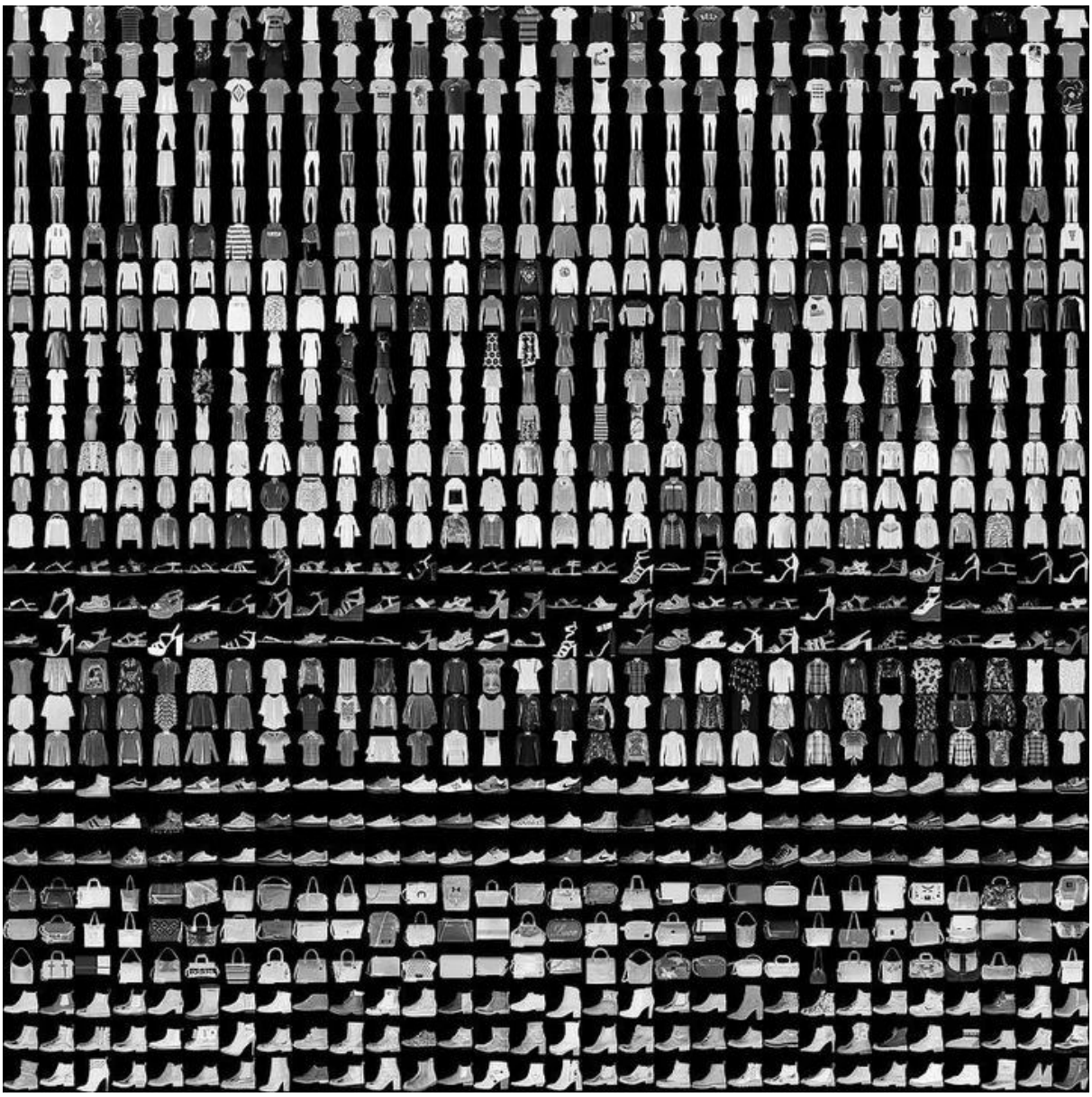


图 23：Fasion MNIST

性能与吞吐率

平均单周期可进行 2.72 次 int8 乘法运算，在 400MHz 的频率下，可以达到 856FPS/S，可以达到实际应用需求，乘法效率远远大于 PicoRV32，与之对比的是 72Cycle/单次乘法。

识别精度

由于指令集式的架构，和网络模型充分解耦，我们采用 BP 网络对手写数字任务进行识别，得到了不错的结果，和 torch 仿真基本一致。

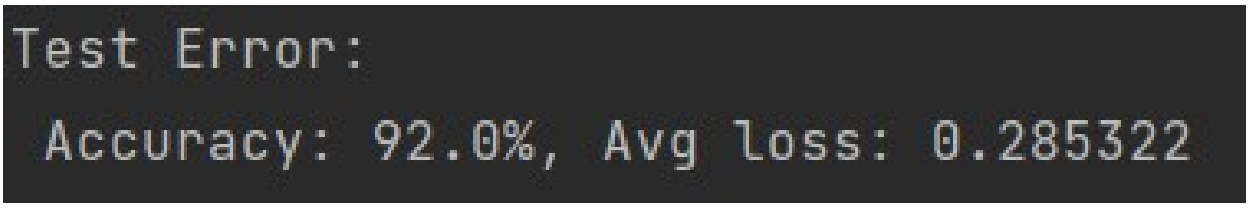


图 24：torch 软件模型识别精度

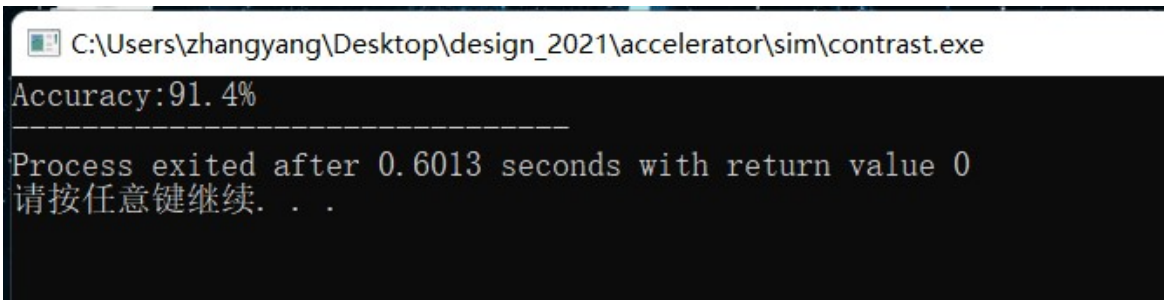


图 25：神经网络加速器 IP 识别精度

可以看到接近 torch 软件模型，并且可以达到实际生产环境所需的准确率。（经过大量实验表明，人类对 fasion MNIST 数据集的识别精度也在百分之 90 左右）

场景适配能力

由于神经网络加速器的极高的灵活性，通过性能测试推断，在 400MHz 的工作频率下，可以对任何 1Gops 以下规模的神经网络达到不错的识别吞吐率，在这种性能下，可以实现大多数低精度图片的识别。（识别精度由网络模型决定，会比 Torch 的浮点模型低 2%左右）

技术方案的特色与创新之处

指令集式的神经网络加速器灵活性极高，软硬联调难度大，对计算机体系结构知识要求高，以此获得的是对多种场景的适配和对高速发展的神经网络结构的适应，目前国内外芯片公司（如谷歌，寒武纪）设计的芯片均为指令集架构，中科院陈云霁老师所著的《智能计算系统》有这样的描述：“为了使一个芯片能支持深度学习算法中不同的算子（或层），业界一般有硬连线和指令集两种方案。

硬连线方案为每个神经网络层提供专门的硬件控制逻辑。这种方案无法支持新的神经网络层。而指令集方案则是把各种神经网络层或算子拆分成一些基本的操作，每个操作由一条指令来完成。如果指令集覆盖了各种深度学习算子的最大公约数，新的神经网络层就能由这些指令拼接组合出来（类似于我们可以用加减乘除和跳转拼接组合出所有科学计算）。因此，指令集方案就能兼顾能效和通用性。”

本项目大量使用开源工具，指令集参考 SymbolicEDA 社区开源的神经网络加速器处理器 MARLANN 的指令集。硬件 IP 部分 80~90%以新型敏捷 RTL 硬件描述语言 Chisel 进行开发，Chisel 项目全部开源，在 Chisel 编码过程中借鉴伯克利 Rocket Chip 实现参数化，函数式与面向对象的硬件编程，技术难度较高，国外 Chisel 构建的项目多次流片，在国内 Chisel 的流片项目较少，基本是基于 FPGA 实现，据了解大型项目只有前段时间国科大开源的香山处理器，在 RV 开源社区引起不小轰动。本次参赛作品也是一次对 Chisel+verilog 联合开发，进行 ASIC Flow 的一次尝试。主控方面使用开源的 PicoRV32，此项目由 verilog 构建，并经过多次流片。使用开源仿真工具及波形查看器 iverilog+gtkwave。尝试 Clifford woolf 老师开源的 yosys 及 ICE Storm 进行 FPGA Flow 的综合与布局布线。

使用八位定点进行推理，八位定点已经成为 AI 加速器行业的事实标准，但是仍然属于新兴技术，在精度下降少量的情况下获得不错的面积下降和性能提升。

七、加分方向实现及完成情况说明

（一）方向 1 完成情况说明

基本完成，已部署至 Lattice FPGA，但限于图片识别任务需要摄像头支持，这涉及到更多的我们不了解的知识，暂时没有展示性。

（二）方向 2 完成情况说明

考虑后续完成。

（三）方向 3 完成情况说明

暂未考虑。

八、参赛感想（不限字数）

附录一：交付文件清单

文件序号	文件名	文件简介	备注
0	asm	汇编器，对.asm 文件进行翻译	
1	doc	设计文档	
2	lib	设计中用的库包括 sram	
3	rtl		
3-1	cpmpute.v	核心计算单元	
3-2	dump.tcl	生成波形脚本	
3-3	Makefile	仿真脚本	
3-4	mem256×32	Picorv32 的 sram	
3-5	mem512×32	Codemem 的 sram	
3-6	mem512×64	Coeffmem 的 sram	
3-7	mem16384×16	Mainmem 的 sram	
3-8	memory.v	主存控制器	

3-9	picorv32.v	cpu 核心	
3-10	picosoc.v	包含 picorv32 和其他外设	
3-11	sequencer.v	指令发射和控制模块	
3-12	simpleuart.v	uart 外设	
3-13	Soctop.v	整个芯片设计顶层	
3-14	spiflash.v	spi 接口的 flash	
3-15	spiflash_tb.v	spiflash 测试环境	
3-16	spimemio.v	spiflash 控制单元	
3-17	testbench.f	加速器仿真文件列表	
3-18	testbench.v	加速器仿真环境	
3-19	top.v	加速器顶层文件	
4	running	测试运行的文件需要放在这里	
5	script	识别精度的测试脚本	
6	sim	用来进行对比标准的行为模型	
7	spyglass	代码检查报告	
8	testcase	进行功能验证的文件	对应每一条汇编指令
9	urgReport	代码覆盖率的报告	

附录二：参考文献

[1]GitHub - SymbioticEDA/MARLANN: Multiply-Accumulate and Rectified-Linear Accelerator for Neural Networks (fastgit.org)

[2]李玲、李威、郭崎、杜子东，智能计算系统 [M].北京： 机械工业出版社

[3]胡伟武、苏孟豪、王焕东、汪文祥、章隆兵，计算机体系结构基础[M].北京： 机械工业出版社

[4]GitHub - cliffordwolf/picorv32: PicoRV32 - A Size-Optimized RISC-V CPU (fastgit.org)

[5] Jacob B, Kligys S, Chen B, et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference [C]// CVPR, 2018.

[6]GitHub - chipsalliance/chisel3: Chisel 3: A Modern Hardware Design Language (fastgit.org)

[7]神经网络量化入门--基本原理 - jermmyhsu - 博客园 (cnblogs.com)

[8]神经网络量化入门--后训练量化 - jermmyhsu - 博客园 (cnblogs.com)

[9] Yu-Hsin Chen,Tushar Krishna,Joel S,Emer,Vivienne Sze,et al.Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks [J]. IEEE Journal of Solid-State Circuits, Jan. 2017: 127 - 138.

[10] [1]龚成，卢冶，代素蓉,等. 一种超低损失的深度神经网络量化压缩方法[J].软件学报，32(8):17.

[11]www.imm.dtu.dk/~masca/chisel-book.pdf(电子书)

[12](11条消息) 第零章 序 Chisel教程汇总 iChthyosaur的博客-CSDN博客 chisel教程

[13]GitHub - chipsalliance/rocket-chip: Rocket Chip Generator (fastgit.org)

[14]包云岗. 处理器芯片开源设计与敏捷开发方法思考与实践[J]. 中国计算机学会通讯(CCCF)，2019.

附录三：指令集文档

指令集参考了 github 开源项目 MARLANN

这是一个累加器式的指令集，指令集中规定了：

- 两个累加器 ACC0，ACC1（32bit），用来执行乘累加运算的累加（对于高性能处理器，可以不限于两个累加器，采用多个累加器并行运算，充分向量化）
- 为实现基址变址寻址，规定了 VBP，LBP，SBP，CBP 共四个基址寄存器
- 为提升 IPC（Instruction Per Cycle），规定了代码专用内存 Code Memory（32bit 位宽，深度不限），参数专用内存 Coeff Memory0 与 Coeff Memory1（各 64bit 位宽，深度不限），分别对应 ACC0 与 ACC1
- 一个 32bit 宽的调用堆栈（下面简称堆栈），用于实现 CALL，RETURN 指令

指令集符号含义：

- MADDR：主存的变址
- CADDR：Code Memory 或 Coeff Memory0 与 Coeff Memory1 的变址
- LEN：连续指令的执行长度
- ARG：特殊参数，特别说明

完整指令集如下

31 15	14 6	5 Opcode 0	Name

31 15	14 6	5 Opcode 0	Name
00000	00000	000000 0	Sync
MADDR	00000	000001 1	Call
00000	00000	000010 2	Return
LEN	CADDR	000011 3	Execute
MADDR	CADDR	000100 4	LoadCode
MADDR	CADDR	000101 5	LoadCoeff0
MADDR	CADDR	000110 6	LoadCoeff1
LEN	00000	000111 7	ContinueLoad
MADDR	00000	001000 8	SetVBP
MADDR	00000	001001 9	AddVBP
MADDR	00000	001010 10	SetLBP
MADDR	00000	001011 11	AddLBP
MADDR	00000	001100 12	SetSBP
MADDR	00000	001101 13	AddSBP
00000	CADDR	001110 14	SetCBP
00000	CADDR	001111 15	AddCBP
MADDR	ARG	010000 16	Store

31 15	14 6	5 Opcode 0	Name
MADDR	ARG	010001 17	Store0
MADDR	ARG	010010 18	Store1
MADDR	ARG	010100 20	ReLU
MADDR	ARG	010101 21	ReLU0
MADDR	ARG	010110 22	ReLU1
MADDR	00000	011000 24	Save
MADDR	00000	011001 25	Save0
MADDR	00000	011010 26	Save1
MADDR	00000	011100 28	LdSet
MADDR	00000	011101 29	LdSet0
MADDR	00000	011110 30	Ldset1
MADDR	00000	100000 32	LdAdd
MADDR	00000	100001 33	LdAdd0
MADDR	00000	100010 34	LdAdd1
MADDR	CADDR	101000 40	MACC
MADDR	CADDR	101001 41	MMAX
MADDR	CADDR	101010 42	MACCZ

31 15	14 6	5 Opcode 0	Name
MADDR	CADDR	101011 43	MMAXZ
CADDR	CADDR	101101 45	MMAXN

指令含义

Sync：空指令，不做任何处理，插入 Sync 用于处理数据冒险。

Call：将此条 Call 下条指令地址压入堆栈，并跳转至主存的 MADDR 进行执行（MADDR 必须 4 字节对齐）

Return：将栈顶地址弹出堆栈，并跳转至此地址进行执行

Execute：从 Code Memory 的 CADDR 为起始，连续执行其中的 LEN 条指令

LoadCode：将 4 字节由主存的 MADDR 地址存入 Code Memory 的 CADDR 地址处（MADDR 必须 4 字节对齐）

LoadCoeff0：将 8 字节由主存的 MADDR 地址存入 Coeff Memory0 的 CADDR 地址处（MADDR 必须 2 字节对齐）

LoadCoeff1：将 8 字节由主存的 MADDR 地址存入 Coeff Memory1 的 CADDR 地址处（MADDR 必须 2 字节对齐）

ContinueLoad：此条指令必须紧接 LoadCode，LoadCoeff0 或 LoadCoeff1，表示连续执行上述指令 LEN 次，LoadCode，LoadCoeff0 或 LoadCoeff1 中的 MADDR 与 CADDR 依次向后递推

SetVBP：将 VBP 置为 MADDR

AddVBP：将 VBP 置为 VBP+MADDR

SetLBP：将 LBP 置为 MADDR

AddLBP：将 LBP 置为 LBP+MADDR

SetSBP：将 SBP 置为 MADDR

AddSBP：将 SBP 置为 SBP+MADDR

SetCBP：将 CBP 置为 MADDR

AddCBP：将 CBP 置为 MADDR

Store：将 ACC0 与 ACC1 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），分别将经过操作的 ACC0 中数据存入主存的 SBP+MADDR 地址处，经过操作的 ACC1 中的数据存入主存的 SBP+MADDR+1 地址处。

Store0：将 ACC0 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），将经过操作的 ACC0 中的数据存入主存的 SBP+MADDR 地址处。

Store1：将 ACC1 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），将经过操作的 ACC1 中的数据存入主存的 SBP+MADDR+1 地址处。

ReLU：将 ACC0 与 ACC1 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），若结果为负数，则置为 0，分别将经过操作的 ACC0 中数据存入主存的 SBP+MADDR 地址处,经过操作的 ACC1 中的数据存入主存的 SBP+MADDR+1 地址处。

ReLU0：将 ACC0 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），若结果为负数，则置为 0，将经过操作的 ACC0 中的数据存入主存的 SBP+MADDR 地址处。

ReLU1：将 ACC1 中数据算术右移 ARG 位，若结果超过 Int8 所能表示的范围，则置为最大（+127）或最小（-128），若结果为负数，则置为 0，将经过操作的 ACC1 中的数据存入主存的 SBP+MADDR+1 地址处。

Save：将 ACC0 中数据存入主存中的 SBP+MADDR 处，将 ACC1 中数据存入主存中的 SBP+MADDR+4 处

Save0：将 ACC0 中数据存入主存中的 SBP+MADDR 处

Save1：将 ACC1 中数据存入主存中的 SBP+MADDR+4 处

LdSet：将主存中 MADDR+LBP 处 32bit 的内容存入 ACC0，将主存中 MADDR+LBP+4 处 32bit 的内容存入 ACC1

LdSet0：将主存中 MADDR+LBP 处 32bit 的内容存入 ACC0

LdSet1：将主存中 MADDR+LBP+4 处 32bit 的内容存入 ACC1

LdAdd：将主存中 MADDR+LBP 处 32bit 的内容与 ACC0 相加存入 ACC0，将主存中 MADDR+LBP+4 处 32bit 的内容与 ACC1 相加存入 ACC1

LdAdd0：将主存中 MADDR+LBP 处 32bit 的内容与 ACC0 相加存入 ACC0

LdAdd1：将主存中 MADDR+LBP+4 处 32bit 的内容与 ACC1 相加存入 ACC1

MACC：将主存中 MADDR+VBP 处 8 字节数据每字节（视作 8 个 Int8）分别与 Coeff Memory0 中 CADDR+CBP 处 8 字节（视作 8 个 Int8）相乘后相加累加入 ACC0，将主存中 MADDR+VBP 处 8 字节数据每字节分别与 Coeff Memory1 中 CADDR+CBP 处 8 字节相乘后相加累加入 ACC1

MMAX：将主存中 MADDR+VBP 处 8 字节数据作有符号数比较，将最大值与 ACC0 比较，较大的存入 ACC0，若 Coeff Memory0 中对应字节为 0，则主存中对应数据不参与比较，将主存中 MADDR+VBP 处 8 字节数据每字节分别与 Coeff

Memory1 中 CADDR+CBP 处 8 字节相乘后相加累加入 ACC1。

MACCZ：将 ACC0 与 ACC1 置为 0，再执行 MACC

MMAXZ：将 ACC0 与 ACC1 置为 0，再执行 MMAX

MMAXN：将 ACC0 置为 Int32 最小值，将 ACC1 置为 0，再执行 MMAX

附录四：汇编文档

我们提供了一个简单的汇编语法与其汇编器的实现，方便对 IP 进行编程的开发者进行编程。

一段完整的汇编应有代码段和数据段的声明，格式为：

- .code <addr>
- .data <addr>

分别表示代码段与数据段会编译到主存的哪个部分，<addr>应小于 128K，要求代码段的起始地址 4 字节对齐。

同时可以定义常量，格式为.sym <lable> <addr>，用以指向程序中某个地址，可以利用.sym 语句与对上位机程序员协商输入输出数据所存放的地址，定义后可在汇编中如使用数字常量一样使用所定义的 lable。

代码段

代码段储存汇编指令，汇编指令与机器指令一一对应（机器指令详见附录三）

Sync

Call <maddr>

Return

Execute <caddr>, <len>

LoadCode <maddr>, <caddr>

LoadCoeff0 <maddr>, <caddr>

LoadCoeff1 <maddr>, <caddr>

ContinueLoad <arg>

SetVBP <maddr>

AddVBP <maddr>

SetLBP <maddr>

AddLBP <maddr>

SetSBP <maddr>

AddSBP <maddr>

SetCBP <caddr>

AddCBP <caddr>

Store <maddr>, <arg>

Store0 <maddr>, <arg>

Store1 <maddr>, <arg>

ReLU <maddr>, <arg>

ReLU0 <maddr>, <arg>

ReLU1 <maddr>, <arg>

Save <maddr>

Save0 <maddr>

Save1 <maddr>

LdSet <maddr>

LdSet0 <maddr>

LdSet1 <maddr>

LdAdd <maddr>

LdAdd0 <maddr>

LdAdd1 <maddr>

MACC <maddr>, <caddr>

MMAX <maddr>, <caddr>

MACCZ <maddr>, <caddr>

MMAXZ <maddr>, <caddr>

MMAXN <maddr>, <caddr>

汇编器内嵌一个四则运算解析式，(<maddr>,<caddr>,<arg>,<len>)都可以使用常量或四则运算，提高语法的表达能力。

数据段

数据段后应跟以若干 int8 常量，每个数字应处于(-128~+127)，每行个数应为 4 的倍数，不足则补 0，数据会被顺序编译至数据段所声明的地址后。