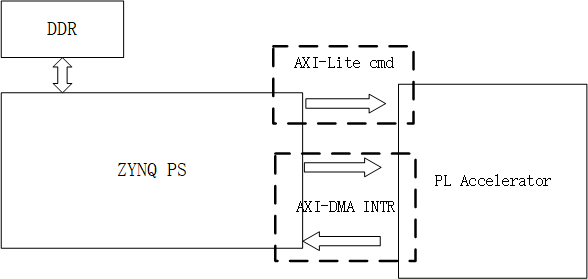
# conv accelerator specification

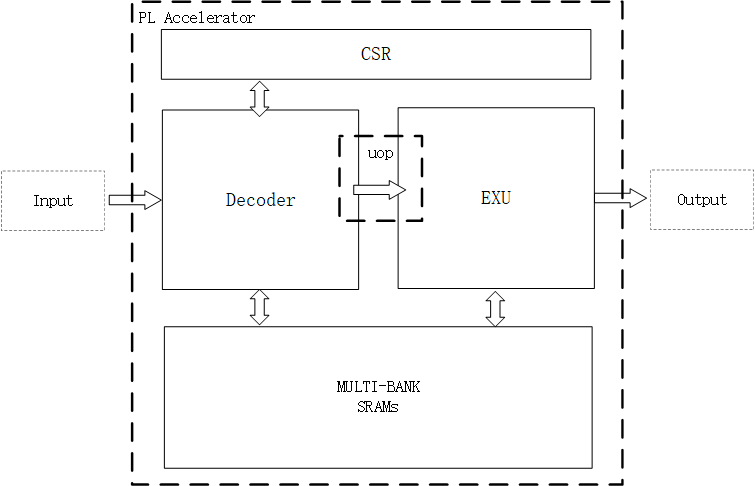
## Host-Device架构

采用PS的 ZYNQ ARM A9 硬核作为Host（主机），自己设计PL加速器作为Device。

SoC规划大致如图：



下面是PL device的模块图：



## 加速器接口定义

本项目作为一个硕士毕业设计，首先需要考虑的是实现成本（可实现性），因此在控制接口的设置上，尽可能精简。

加速器的控制接口仅仅实现能够支持通用CONV运算的最小子集即可。

基于这种考虑，暂定为基于**控制和状态寄存器（CSR）**操作的模式：

定义一系列的CSR用于给conv-npu内部的控制与状态寄存器编程；Device端仅仅根据当前内部CSR寄存器状态进行对应的操作。

**设计目标：**需要支持一个pytorch conv算子的运算。

在微架构设计上，采用类似CISC处理器的思想，在加速器的顶层只需要关心pytorch conv算子的参数（如input\_scale, in\_channel, out\_channel, kernel\_size, stride, padding等）

Device的解码器decoder将在运行时进行译码，将一个单层conv layer的调用，转变为一系列单次卷积核大小的乘累加（MAC）并行计算。单次kernel size大小的SIMD卷积运算视为Device的一次微操作（uop）。

因此在设计实现上，大致可分为两级：

1. 配置加速器的CSR，Decoder根据CSR状态对后端硬件进行特定操作（issue uops）
2. 后端功能部件（EXU）根据uop指示，执行具体的数据计算。

## CSR规定

CSR用于保存控制整个加速器运行和标识当前状态的全局变量。

CSR设计为AXI-LITE slave内存映射寄存器，可以参考vivado Tools内自带的AXI IP例程RTL实现方案。最终在Vitis中进行嵌入式软件开发时，只需调用Xilinx官方BSP自带的Xil\_Out函数即可完成向PL device中写入CSR，进而控制整个加速器的运行。

## 访存延迟隐藏方案

访存延迟（latency）是GPGPU以及NPU设计中不可回避的问题。在本项目的加速器架构设计中，为简化实现，每一块BRAM bank均采用ping-pong buffer设计（或者称为A-B slot设计）。

其中每一块bram bank均有相同的两份，当A被用于计算时，B可被用于立即同时写入下一次conv调用所需要的数据。如此通过slot-A和slot-B来回切换工作，最终实现隐藏访存延迟的目的。

经过分析，使用这种设计模式，BRAM开销在可接受范围内。这是因为conv算子天然的权重复用特性，单个conv卷积核的权重数据量很小。

## 最小计算单元PE设计

基本思想：控制和计算分离，指令前后端解耦。

最基础的计算单元为PE，其输入输出端口如下：

module pe (

    input [`DATA\_RANGE] x,

    input [`DATA\_RANGE] weight,

    input in\_valid,

    input flush,

    input out\_en,

    input calc\_bias,

    output reg [`DATA\_RANGE] result\_r,

    output reg out\_valid\_r,

    //////////////////

    input clk,

    input rst\_n

);

PE作为设计中的基本计算单元，仅仅根据输入控制信号完成冲刷(FLUSH)、基本乘累加（MAC）、计算偏置（calc\_bias）以及输出（out\_en）功能。对于CONV1D计算采用输出固定模式（OS-MODE）。

其功能定义如下：

rst\_n异步复位优先级最高，一旦有效则立即复位pe内部寄存器。

其次，按照如下逻辑进行信号处理：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **flush** | **in\_valid** | **calc\_bias** | **out\_en** | **执行结果 1cycle** |
| 1 | x | x | x | pe寄存器冲刷 |
| 0 | 0 | 0 | 0 | nop，维持寄存器不变 |
| 0 | 0 | 0 | 1 | 将out\_valid\_r拉高 |
| 0 | 0 | 1 | 0 | 非法uop，触发异常 |
| 0 | 0 | 1 | 1 | 非法uop，触发异常 |
| 0 | 1 | 0 | 0 | 基本乘累加MAC操作 |
| 0 | 1 | 0 | 1 | 完成最后一次MAC并将out\_valid\_r拉高 |
| 0 | 1 | 1 | 0 | result\_r加本周期的weight（偏置） |
| 0 | 1 | 1 | 1 | result\_r加本周期的weight（偏置），并将out\_valid\_r拉高 |

## 一次典型的conv计算过程

~~写CSR给出下一次写操作的信息，通告Device即将写入当前conv layer权重BRAM；~~

~~启动DMA写通道，将权重数据写入PE\_NUM个权重缓存BRAM内；~~

~~写CSR给出下一次写操作的信息，通告Device即将写入一部分特征图数据；~~

~~启动DMA写通道，将特征图数据写入feature map缓存BRAM内；~~

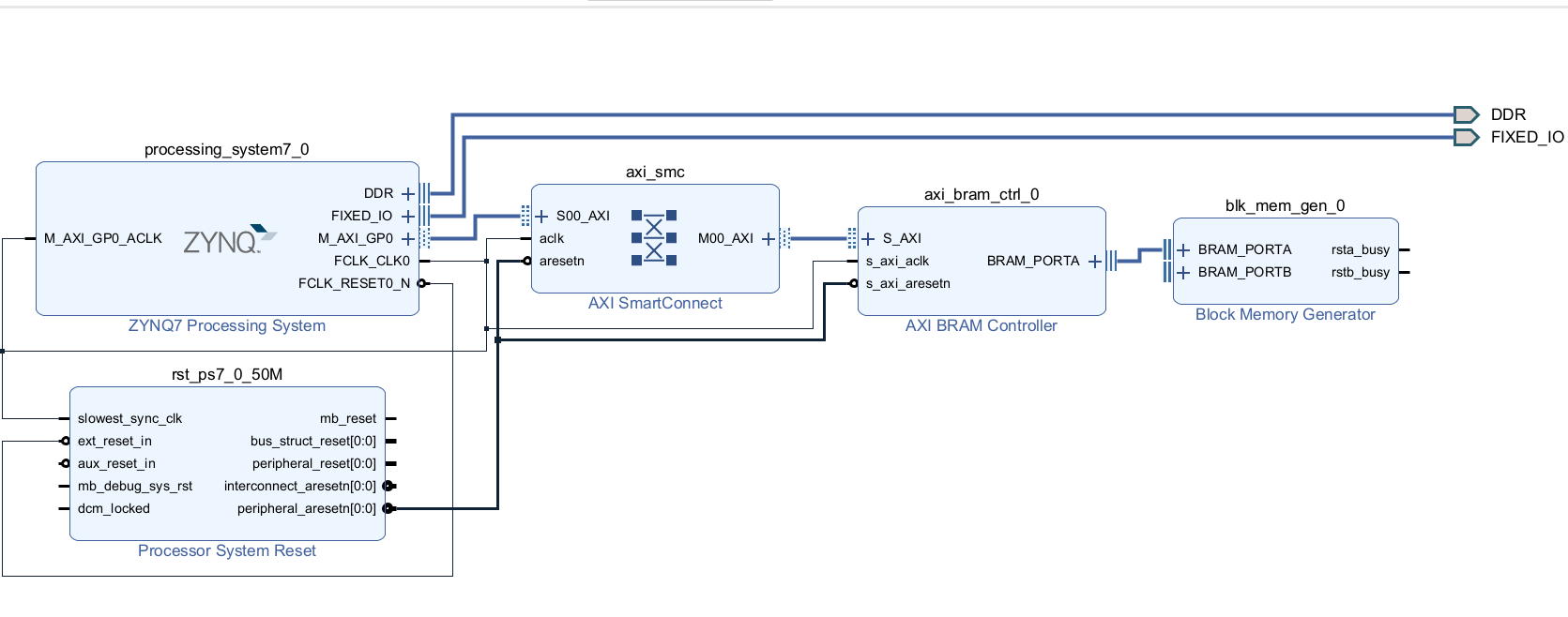
~~写CSR，通告Device本次卷积的计算参数，例如channel数，kernel\_size，padding，stride等；~~

~~写CSR，通告Device启动计算，同时开启DMA的返回数据接受通道。~~

~~在等待计算任务完成的同时，和上面类似，启动下一层卷积的权重和特征图输入DMA写入。~~

现在，考虑将所有加速器的BRAM挂载到AXI互联总线，并且用双端口模式例化。

其中PORT A连接到AXI-BRAM controller；另一个端口PORT B预留给device使用。



如图所示：

这样所有的BRAM都将有memory map内存映射。也就是说，从主机cpu访问这些BRAM可以像访问DDR内存一样，直接通过地址访问。

## BRAM bank设置

假定PE\_NUM为32：

再加上一个输入特征图缓存和一个输出特征图缓存

则共有34块BRAM，其中前32块为权重缓存，后2块为特征图缓存。

根据conv算子的特性可知，前面的32块容量较小，后两块容量需求较大。

再考虑到Ping-pong buffer的设计（用于流水化执行，隐藏访存延迟），每一块BRAM都需要一个相同大小的复制作为ping-pong缓冲。因此实际需要68块这样的BRAM.

下面需要根据算法的要求，合理地规划这68块BRAM占用的大小：

## 算法数据流分析

原始输入数据：1\*360个样本点（设每个样本32bit）则有11520bit

第一层权重：4个 3\*1\*1 每个卷积核参数量为3word，即单个卷积核96bit

（这一层参数不合适，加速效率太低，可能需要删改）

第二层conv数据：4\*360个点：共46080bit

权重：16个 3\*1\*4，单个卷积核384bit

（也不太合适）

第三层conv数据：16\*360个点：共184320bit（约23KB）

权重：16个3\*1\*16 单个卷积核1536bit