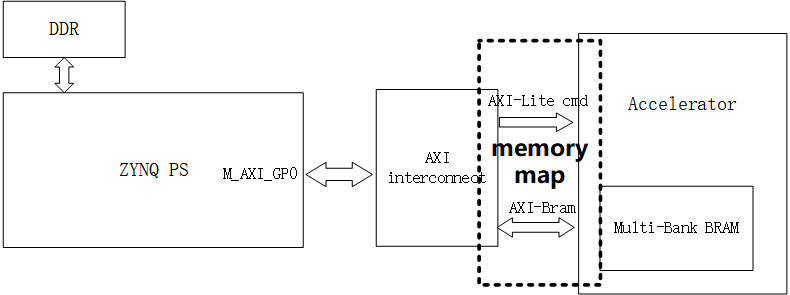
# Conv Accelerator Specification

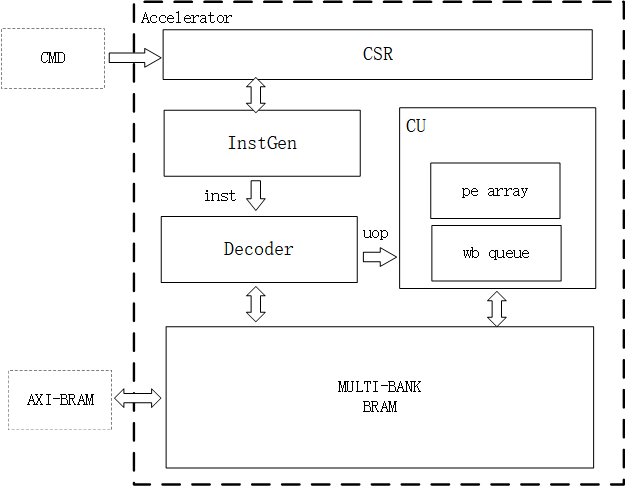
## Host-Device架构

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. 通过PS配置PL-CSR，PL-device硬件根据CSR状态生成一系列指令和uops
2. 计算单元（CU）根据uop指示，执行具体的数据计算。

## CSR规定

与RISC-V类似，CSR用于保存控制运行和标识状态的全局变量。

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中读写CSR，进而控制整个加速器的运行。

首先，分析一个torch conv1d算子：

nn.Conv1d(in\_channel=256, out\_channel=512, kernel\_size=3, stride=1, padding=1, bias=True)

可以看到，对于一个卷积算子来说，其卷积核至少有：

* in\_channel
* out\_channel
* kernel\_size
* stride
* padding
* bias

这6个参数。

除去卷积核参数，还需要传入**特征图首地址**，**特征图大小**这些参数。

如果考虑将每次conv以及激活函数ReLU融合实现，那么还需要一个信号用于选择是否需要融合ReLU层，以实现在device端硬件加速完成ReLU激活函数。

总结如下：

将其每个layer参数实现为每个32bit的CSR：

|  |  |  |  |
| --- | --- | --- | --- |
| **csr addr** | **csr name** | **function or definition** | **AXI访问权限** |
| 0 | feature\_baseaddr | 特征图首地址(soc内存映射地址) | RW |
| 1 | feature\_chin | 特征图通道数（在物理内存中，特征图数据依次按通道数，宽，高排列） | RW |
| 2 | feature\_width | 特征图宽度 | RW |
| 3 | feature\_height（conv1d） | 特征图高度（若高度为1，则退化为conv1d） | RW |
| 4 | kernel\_num(chout) | 卷积核数量（输出通道数） | RW |
| 5 | kernel\_size | 卷积核尺寸（仅支持conv1d或者方形卷积核的conv2d） | RW |
| 6 | stride | 卷积步长（1，2，……） | RW |
| 7 | padding | 边缘填充（0，1，……）（暂时不支持硬件运行时padding，此选项无效） | RW |
| 8 | has\_bias | 是否含有偏置（0 false；1 true）偏置的值存放在卷积核权重的最后一个 | RW |
| 9 | npu\_control | npu控制寄存器 | RW |
| 10 | npu\_state | npu状态寄存器 | R(写无效) |
| 11 | uop\_num | 当前conv layer需要的微操作次数 | R(写无效) |
| 12 | uop\_id | 当前的微操作id | R(写无效) |
| 13 | output\_baseaddr | 当前layer写回的首地址（soc内存映射地址） | RW |

npu\_control bits 定义

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 | … | 3 | 2 | 1 | 0 |  |
| Reserved | … | 停止运算并冲刷流水线 | 暂停 | 启动当前layer运算 | Reserved |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | … | 3 | 2 | 1 | 0 |
| Reserved | … | 停止运算并冲刷流水线 | 暂停 | 启动当前layer运算 | Reserved |

npu\_state bits 定义

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | … | 3 | 2 | 1 | 0 |
| Reserved | … | 异常 | 计算完成 | PE busy | IDLE空闲 |

## 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 signed [`DATA\_RANGE] x,

    input signed [`DATA\_RANGE] weight,

    input in\_valid,

    input flush,

    input out\_en,

    input calc\_bias,

    input calc\_relu,

    //////////////////

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

    output reg out\_valid\_r,

    output reg illegal\_uop,

    //////////////////

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **flush** | **in\_valid** | **calc\_bias** | **calc\_relu** | **out\_en** |
| 1 | x | x | x | x |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |

## 计算单元CU设计

计算单元CU承接指令解码器decoder派遣的微操作uop，并从BRAM取出操作数，打包分发给PE阵列（pe\_array）完成计算。

由于PE\_NUM个pe单元会在同一个周期返回多个计算结果，因此在CU模块中需要实现一个写回队列（wb\_queue）用于将并行写回的多个result在多个周期内串行返回给上一级模块，并最终串行地写回特征图缓存BRAM内。

由于并串模块是一个周期内接受多个数据，但数据需要多个周期才能完成发送。这一步可能出现前一步的数据还没发送完成，但是PE阵列又送来新数据的情况。因此，在这个环节需要引入反压设计（back pressure）。

为了尽可能避免无意义的阻塞，CU的反压设计思路如下：对于PE而言，反压是不可见的，反压仅仅对于CU以及更高层的模块可见。反压的源头是wb\_queue中的有效数据非空。反压将会向CU的上一层传递，最终阻塞decoder向CU发送out\_en信号。也就是说，如果前一阶段的数据滞留在wbq中还未完成发送，后续的MAC计算、bias计算和RELU计算可以正常进行，只不过不允许从PE阵列输出；一旦后续的wbq清空，则解除反压，decoder允许发出out\_en信号，从而允许PE阵列输出结果。

CU的端口声明如下：

module cu (

    // pe input ports

    input signed [`DATA\_RANGE] kernel\_data  [`PE\_NUM-1:0],

    input signed [`DATA\_RANGE] feature\_data [`PE\_NUM-1:0],

    input [`PE\_NUM-1:0] in\_valid  ,

    input [`PE\_NUM-1:0] out\_en    ,

    input [`PE\_NUM-1:0] calc\_bias ,

    input [`PE\_NUM-1:0] calc\_relu ,

    input flush,

    // output ports

    output logic signed [`DATA\_RANGE] result\_out,

    output logic result\_out\_valid,

    output logic illegal\_uop,

    output logic wb\_busy,

    //////////////////////

    input wire clk,

    input wire rst\_n

);

## decoder模块设计

decoder模块承接instgen模块产生的单次conv操作指令，将其解析为一系列的微操作uops，并且生成BRAM的PORTB访存地址，用于准备uop和对应的操作数，传递给CU完成计算。

其端口声明如下：

module decoder (

    // instgen port

    input [`FRAM\_ADDR\_RANGE] feature\_baseaddr,

    input [`KRAM\_ADDR\_RANGE] kernel\_baseaddr,

    input [`DATA\_RANGE] feature\_chin,

    input [`DATA\_RANGE] feature\_chout,

    input [`DATA\_RANGE] feature\_width,

    input [`DATA\_RANGE] kernel\_sizeh,

    input [`DATA\_RANGE] kernel\_sizew,

    input has\_bias,

    input has\_relu,

    input [`FRAM\_ADDR\_RANGE] output\_baseaddr,

    input inst\_valid,

    output decoder\_ready,

    // cu port

    output reg [`PE\_NUM-1:0] in\_valid  ,

    output reg [`PE\_NUM-1:0] out\_en    ,

    output reg [`PE\_NUM-1:0] calc\_bias ,

    output reg [`PE\_NUM-1:0] calc\_relu ,

    output reg               flush     ,

    input wb\_busy,

    // BRAM port

    output logic [`FRAM\_ADDR\_RANGE] fram\_addr,

    output logic [`KRAM\_ADDR\_RANGE] kram\_addr,

    //////////////////////

    input wire clk,

    input wire rst\_n

);

在上述端口CU PORT中：

**in\_valid\out\_en\calc\_bias\calc\_relu\flush**

可以视为一组uop信号，它们共同决定了CU的计算数据流。

为了保证uop控制信号与来自BRAM的数据同步，上述uop信号均使用reg延迟一拍输出。

即：BRAM PORT中的fram\_addr和kram\_addr比uop信号早一个周期给出，这样当BRAM在下一拍输出数据时，恰好与uop信号完成同步。

## 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；~~

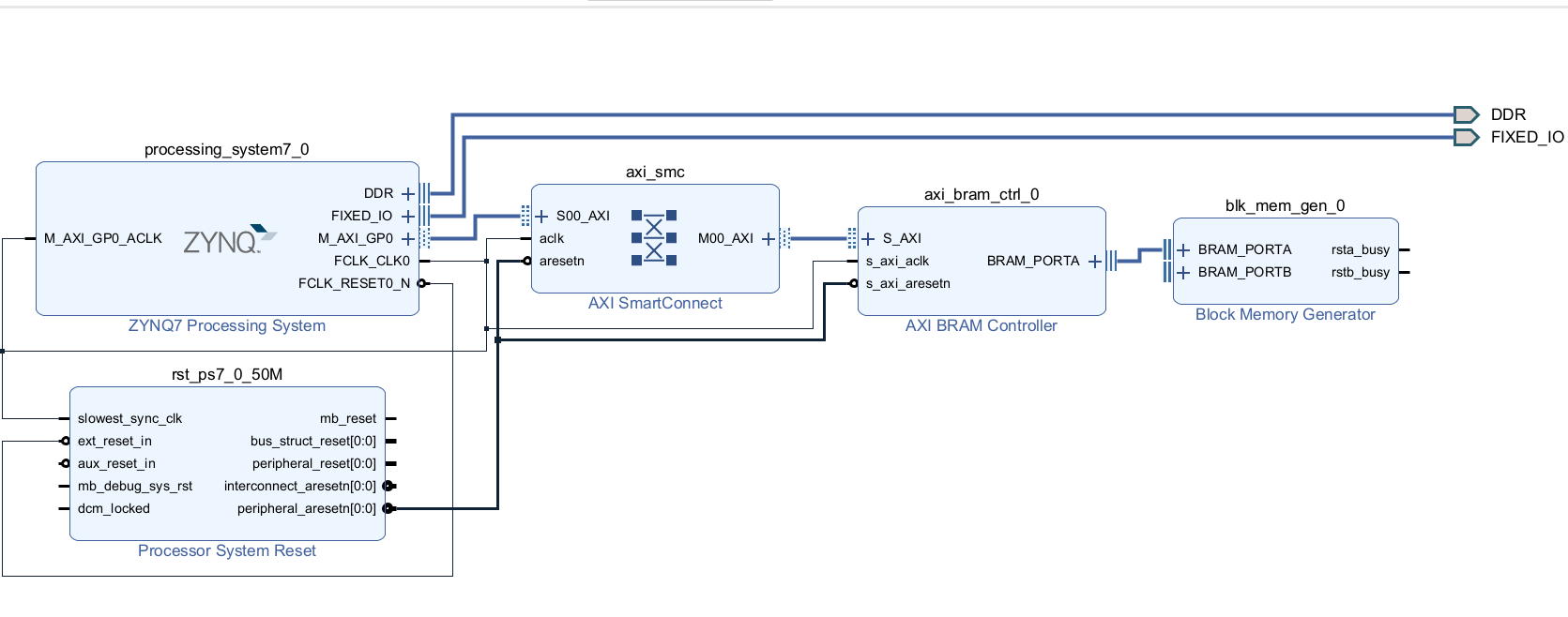
~~写CSR，通告Device本次卷积的计算参数，例如channel数，kernel\_size，padding，stride等；~~

~~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1个make\_layer：**

BLOCK1:

conv1 数据：4\*360个点：共46080bit

权重：16个 3\*1\*4，单个卷积核384bit

identity 数据：4\*360个点：共46080bit

权重：16个 1\*1\*4，单个卷积核128bit

conv2 数据：16\*360个点：共184320bit（约23KB）

权重：16个3\*1\*16 单个卷积核1536bit

BLOCK2:

conv1 数据：16\*360 共184320bit（约23KB）

权重：16个3\*1\*16 单个卷积核1536bit

identity 空

conv2 数据：16\*360 共184320bit（约23KB）

权重：16个3\*1\*16 单个卷积核1536bit

**第2个make\_layer：**

BLOCK1:

conv1 数据：16\*360 共184320bit（约23KB）

权重：32个 3\*1\*16，单个卷积核1536bit

identity 数据：16\*360 共184320bit（约23KB）

权重：32个 1\*1\*16，单个卷积核512bit

conv2 数据：32\*180个点：共184320bit（约23KB）

权重：32个3\*1\*32 单个卷积核3072bit

BLOCK2:

conv1 数据：32\*180个点：共184320bit（约23KB）

权重：32个3\*1\*32 单个卷积核3072bit

identity 空

conv2 数据：32\*180个点：共184320bit（约23KB）

权重：32个3\*1\*32 单个卷积核3072bit

**第3个make\_layer：**

BLOCK1:

conv1 数据：32\*180个点：共184320bit（约23KB）

权重：64个 3\*1\*32，单个卷积核3072bit

identity 数据：32\*180个点：共184320bit（约23KB）

权重：64个 1\*1\*32，单个卷积核1024bit

conv2 数据：64\*90个点：共184320bit（约23KB）

权重：64个3\*1\*64 单个卷积核6144bit（约0.8KB）

BLOCK2:

conv1 数据：64\*90个点：共184320bit（约23KB）

权重：64个3\*1\*64 单个卷积核6144bit（约0.8KB）

identity 空

conv2 数据：64\*90个点：共184320bit（约23KB）

权重：64个3\*1\*64 单个卷积核6144bit（约0.8KB）

**第4个make\_layer：**

BLOCK1:

conv1 数据：64\*90个点：共184320bit（约23KB）

权重：128个 3\*1\*64，单个卷积核6144bit（约0.8KB）

identity 数据：64\*90个点：共184320bit（约23KB）

权重：128个 1\*1\*64，单个卷积核2048bit

conv2 数据：128\*45个点：共184320bit（约23KB）

权重：128个3\*1\*128 单个卷积核12288bit（约1.5KB）

BLOCK2:

conv1 数据：128\*45个点：共184320bit（约23KB）

权重：128个3\*1\*128 单个卷积核12288bit（约1.5KB）

identity 空

conv2 数据：128\*45个点：共184320bit（约23KB）

权重：128个3\*1\*128 单个卷积核12288bit（约1.5KB）

**第5个make\_layer：**

BLOCK1:

conv1 数据：128\*45个点：共184320bit（约23KB）

权重：256个 3\*1\*128，单个卷积核12288bit（约1.5KB）

identity 数据：128\*45个点：共184320bit（约23KB）

权重：256个 1\*1\*128，单个卷积核4096bit

conv2 数据：256\*23个点：共188416bit（约23KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

BLOCK2:

conv1 数据：256\*23个点：共188416bit（约23KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 空

conv2 数据：256\*23个点：共188416bit（约23KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

**第6个make\_layer：**

BLOCK1:

conv1 数据：256\*23个点：共188416bit（约23KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 数据：256\*23个点：共188416bit（约23KB）

权重：256个 1\*1\*256，单个卷积核8192bit

conv2 数据：256\*12个点：共98304bit（约12KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

BLOCK2:

conv1 数据：256\*12个点：共98304bit（约12KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 空

conv2 数据：256\*12个点：共98304bit（约12KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

**第7个make\_layer：**

BLOCK1:

conv1 数据：256\*12个点：共98304bit（约12KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 数据：256\*12个点：共98304bit（约12KB）

权重：256个 1\*1\*256，单个卷积核8192bit

conv2 数据：256\*6个点：共49152bit（约6KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

BLOCK2:

conv1 数据：256\*6个点：共49152bit（约6KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 空

conv2 数据：256\*6个点：共49152bit（约6KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

**第8个make\_layer：**

BLOCK1:

conv1 数据：256\*6个点：共49152bit（约6KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 数据：256\*6个点：共49152bit（约6KB）

权重：256个 1\*1\*256，单个卷积核8192bit

conv2 数据：256\*3个点：共24576bit（约3KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

BLOCK2:

conv1 数据：256\*3个点：共24576bit（约3KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

identity 空

conv2 数据：256\*3个点：共24576bit（约3KB）

权重：256个3\*1\*256 单个卷积核24576bit（约3KB）

**YOLO-detector**

**最后分支（256\*3输入）**

**mixed layer：**

conv1 数据：256\*3个点：共24576bit（约3KB）

权重：256个 1\*1\*256，单个卷积核8192bit

conv2 数据：256\*3个点：共24576bit（约3KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv3 数据：512\*3个点：共49152bit（约6KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

**OUTPUT LAYER:**

conv1 数据：256\*3个点：共24576bit（约3KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv2 数据：512\*3个点：共49152bit（约6KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

conv3 数据：256\*3个点：共24576bit（约3KB）

权重：12个1\*1\*256 单个卷积核8192bit

**输出张量：12\*3个点**

**倒数第二分支（256\*6 concat upsample(256\*3) = 512\*6）**

**mixed layer：**

conv1 数据：512\*6个点：共98304bit（约12KB）

权重：256个 1\*1\*512，单个卷积核16384bit(2KB)

conv2 数据：256\*6个点：共49152bit（约6KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv3 数据：512\*6个点：共98304bit（约12KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

**OUTPUT LAYER:**

conv1 数据：256\*6个点：共49152bit（约6KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv2 数据：512\*6个点：共98304bit（约12KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

conv3 数据：256\*6个点：共49152bit（约6KB）

权重：12个1\*1\*256 单个卷积核8192bit

**输出张量：12\*6个点**

**倒数第三分支（256\*12 concat upsample(256\*6) = 512\*12）**

**mixed layer：**

conv1 数据：512\*12个点：共196608bit（约24.576KB）

权重：256个 1\*1\*512，单个卷积核16384bit(2KB)

conv2 数据：256\*12个点：共98304bit（约12KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv3 数据：512\*12个点：共196608bit（约24.576KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

**OUTPUT LAYER:**

conv1 数据：256\*12个点：共98304bit（约12KB）

权重：512个3\*1\*256 单个卷积核24576bit（约3KB）

conv2 数据：512\*12个点：共196608bit（约24.576KB）

权重：256个1\*1\*512 单个卷积核16384bit（约2KB）

conv3 数据：256\*12个点：共98304bit（约12KB）

权重：12个1\*1\*256 单个卷积核8192bit

**输出张量：12\*12个点**

**统计结果，最大的特征图为196608bit，即24576Byte；（需要32KB X 4块）**

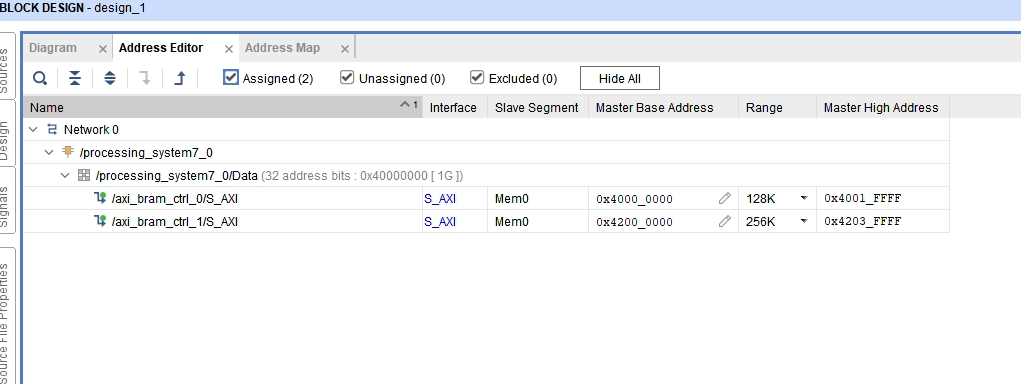
**若特征图BRAM位宽为32bit，则需要Depth为8192的4块。**

**最大的单个卷积核为24576bit，即3072Byte。（需要4KB X 64块）**

**若卷积核权重BRAM位宽为32bit，则需要Depth为1024的64块。**

## BRAM缓存spec

为了减小BRAM地址映射占用的AXI总线地址空间大小，将4块32KB的放在一起，接入一个AXI-BRAM controller；将另外的64个4KB的BRAM放在一起，接入第二个AXI-BRAM controller。地址划分大致如下（AXI总线内存映射基地址可按需修改）：



为了支持单口的AXI-BRAM controller到多bank BRAM的访问，需要一个硬件**片选产生模块**，用于将AXI-BRAM controller的单个BRAM读写端口映射到多个实体BRAM上。

### BRAM\_BANK\_MUX模块

bram\_bank\_mux模块用于将多bank交织的SRAM访问地址映射到正确的端口。在本工程中，使用了参数化的设计方法，以便于在feature sram和kernel weight sram复用相同的代码，其端口如下：

module bram\_bank\_mux #(

    parameter ADDR\_WIDTH = 32,

    parameter DATA\_WIDTH = 32,

    parameter BANK\_NUM = 4,

    parameter BANK\_ADDR\_WIDTH = ADDR\_WIDTH - $clog2(BANK\_NUM)

)(

    // AXI BRAM controller SLAVE ports

    input  logic [ADDR\_WIDTH-1:0] addr,

    input  logic [DATA\_WIDTH-1:0] wdata,

    input  logic                  we,

    input  logic                  en,

    output logic [DATA\_WIDTH-1:0] rdata,

    // BRAM BANK ports

    output logic [BANK\_ADDR\_WIDTH-1:0] bram\_addr [BANK\_NUM-1:0],

    output logic [DATA\_WIDTH-1:0] bram\_wdata [BANK\_NUM-1:0],

    output logic [BANK\_NUM-1:0] bram\_we,

    output logic [BANK\_NUM-1:0] bram\_en,

    input  logic [DATA\_WIDTH-1:0] bram\_rdata [BANK\_NUM-1:0]

);

## 通用conv计算要解决的问题

1：内存编址的维度顺序

[ch, h, w] 还是 [h, w, ch]

这直接决定了下面的问题2和问题3的难易程度。若使用[ch, h, w]，问题3是非常容易解决的，但是问题2会变得非常麻烦；若使用[h, w, ch]，则问题2会相对简单，但是问题3会变麻烦。

经过权衡，选择[ch, h, w]数据排列方式。这样当kernel\_num > PE\_NUM时，仅仅需要将分多次计算输出的特征图首尾相接即可。

2：当feature size > feature bram时如何拆分？

需要沿着[h,w]维度切片，分多次计算。

需要注意的是，一旦采用了[ch, h, w]数据内存排布格式，则沿[h，w]进行特征图切分将会变得略有繁琐，涉及到不连续的内存重排。但考虑到将DDR的数据拷贝到内存映射BRAM本来也是逐点拷贝的，这一步的开销尚可接受。

3：当kernel num > PE\_NUM 时如何拆分？

见问题1的回答。

4：当kernel size > kernel bram时如何拆分？

由于conv运算的本身特性，每一个卷积核的尺寸均非常小，因此这种情况几乎不可能出现。

5：通用的stride如何实现？

通过上述spec文档中的CSR-INST-UOP架构，每次计算一个kernel size大小的立方体，每次传入正确的像素首地址即可解决stride通用性。

~~6：通用的padding如何实现？~~

~~通过设计一套虚拟地址，在uop层面均使用虚拟地址，当访问BRAM时进行虚实地址转译，若发现虚地址在padding区域内，则直接将对应的操作数reg赋值为0；若转译地址属于真实BRAM地址，则返回BRAM输出的数据作为操作数。~~

上述实现需要非2幂次除数的除法运算和多周期的数值比较，硬件实现代价过大，因此PL加速器端不考虑padding。默认传入的特征图已经进行了padding。

7：如何实现conv1d和conv2d的兼容？

通过将kernel\_sizeof\_height参数设置为1，将conv2d退化为1d。

除此以外，decoder模块在生成一次微操作的BRAM地址时已经考虑了conv2d与conv1d

feature和kernel的偏移地址差异。

8：为什么此加速器的限制如此大，用起来如此麻烦？

因为没有专用的编译器！我们的CU模块理论上支持任意的数据流计算，如果我们将CU的uop控制接口暴露给加速器顶层，供HOST设备直接调用，理论上可以支持任意的conv计算，但是这将数据流控制的复杂度从硬件转移给了软件，我们在有限的时间内无法实现适配本硬件的编译器。因此，退而求其次，我们将原本可以实现的基于指令的细粒度数据流控制，转变为由控制和状态寄存器（CSR）控制的若干种固定操作模式，变成了一种粗粒度的并行计算加速器。