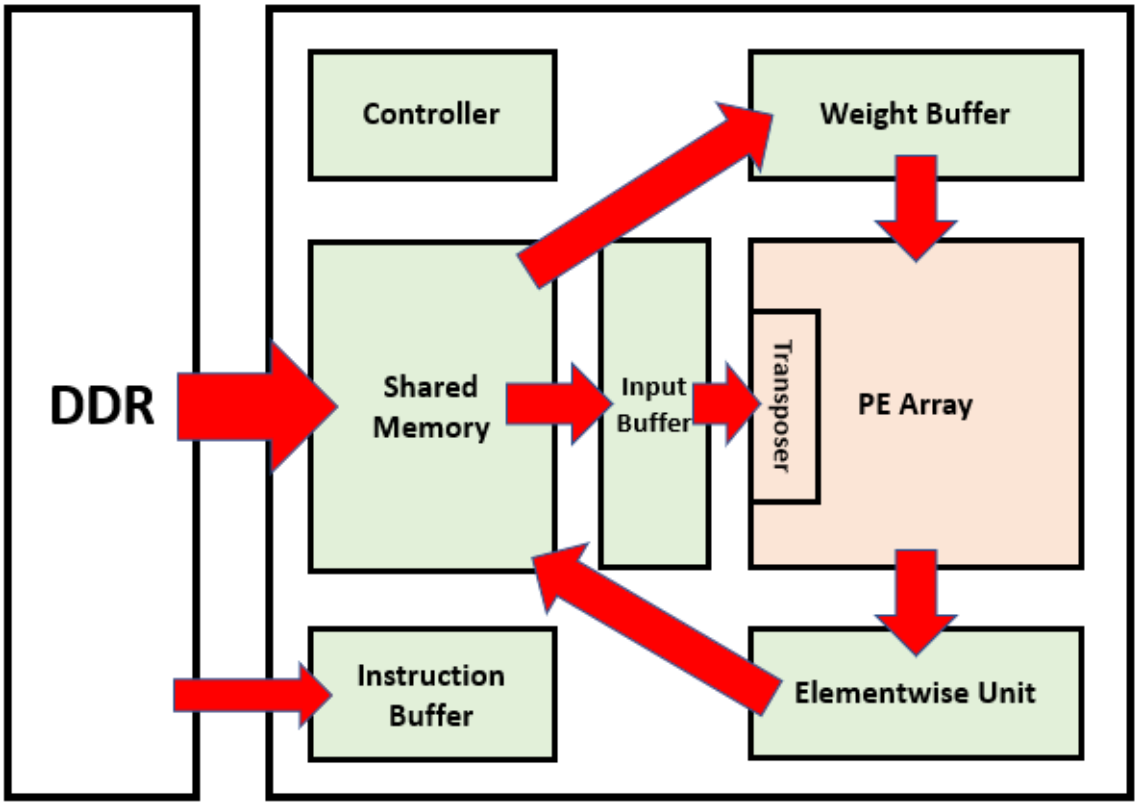
Lab2实验说明

本次实验的要求是设计一个推理加速器（称之为tiny-TPU），其主要功能为计算矩阵乘以及简单的激活函数。TPU简介：张量处理单元 - 维基百科，自由的百科全书 (wikipedia.org)

硬件结构总览

本次实验中，我们把TPU结构简化为下图：



其中，Shared Memory 中存放运算数据，Introduction Buffer中存放指令，在模块运行初期，Shared Memory 和Introduction Buffer需要从DDR中搬运数据和指令。为了简化实验，同学们可以使用verilog中的$readmemh模拟从DDR中取数的过程。除了Shared Memory和Weight Buffer，整个模块中还有两块存储模块：Input Buffer和Weight Buffer，分别存储的是用于PE Array计算所需的input activation和weight。

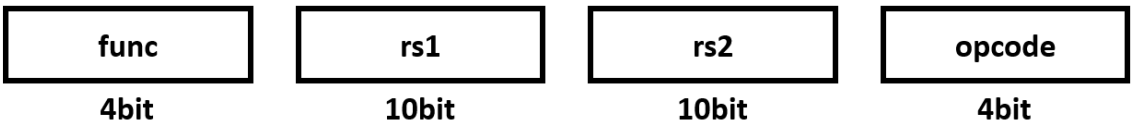
此次实验的计算模块包括：PE Array和Elementwise Unit。其中PE Array是tiny-TPU中计算矩阵和矩阵乘的模块，其具体实现拟为4x4的systolic array（systolic array简介：脉动阵列 - 因Google TPU获得新生 - 知乎 (zhihu.com)），dataflow类型是output-stationary；Elementwise Unit处理激活函数等功能。

除此之外，还需要Controller模块进行整个硬件系统的调度。不同模块之间的连接关系已经用红色箭头表示。

ISA定义

机器码格式定义

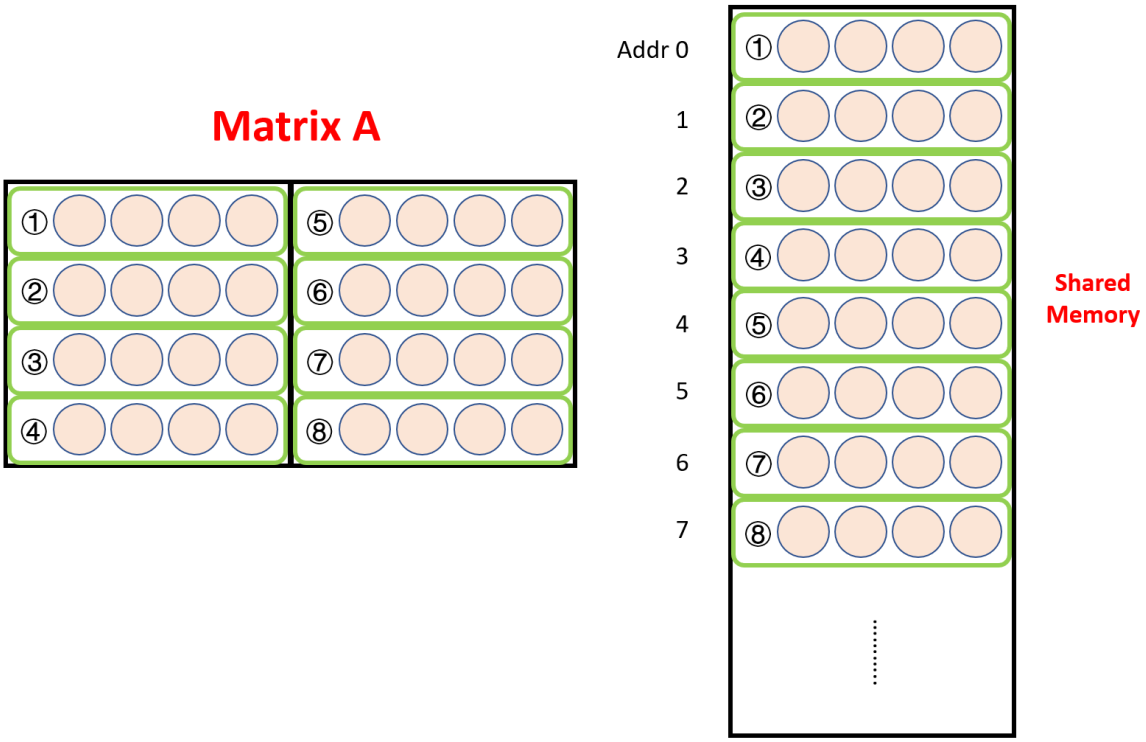
本设计中的指令机器码统一长度是28bit，其结构如下：



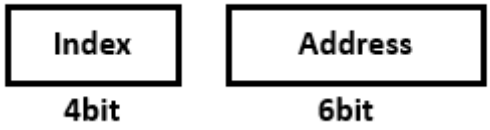
其中opcode是操作码，用于区别不同指令的功能；rs1和rs2表示操作数地址或者操作结果存储地址；func是该条指令除基础功能外的冗余功能。

地址空间定义

我们的设计中，PE Array是4x4的systolic array，一次可以计算得到两个4x4矩阵乘累加的结果。实际编程时，我们会按照课程中所讲的分tile方式来计算大矩阵乘，如两个8x8的矩阵乘可以分为2个4x4的矩阵乘来完成，为了简单起见，我们约定矩阵的size控制在4的倍数。为了配合分tile的这种计算方式，在存储器中矩阵的存放也是每4个元素为一簇来存的。下面这个例子会详细说明：



如上图矩阵A的尺寸为4x8，每个元素是用int32表示。由于计算方式是分tile的，因此存储之前会把该矩阵分为左右两个4x4的矩阵以便计算使用。把该矩阵存到memory时，会按照图中绿色框所示把元素按行分簇存储，具体存放视右图，在存储器中不同元素簇在存储器中的地址依次递增，对存储器读写时，在给定Addr后，存储器就会把该Addr下的一簇元素并行读出/写入（所以每个存储器的读写位宽为4x32=128bit）。该项目中的每个存储器存储方式一致。



机器码中的rs1和rs2都是10bit，把它拆解开看也分为两部分：1）Index部分是为了选择不同的memory块（如下表所示）；2）Address是当前memory块中要选择的数据块的首地址。由于计算是基于tile的，因此读写数据也是分tile的，因此当硬件系统拿到rs1或rs2中的Address后，会默认读取/写入当前Addr至Addr+3中的4簇元素（也就是4x4tile的数据）。

Memory block	Shared Memory	Input Buffer	Weight Buffer
Index	0001	0010	0100

汇编指令集

基于Shared Memory的架构的运算流程大致可以抽象为三步：1）Load；2）Compute；3）Store。基于这种抽象，我们可以把系统的汇编指令集定义为以下3条：

1) mv rs1 rs2
2) preload rs1 rs2
3) compute rs1 rs2

接下来我们来展开讲这几条指令的定义：

mv rs1 rs2

含义：从rs1取数，把数搬运的rs2地址上。

机器码：0001 rs1 rs2 0001

preload rs1 rs2

含义：从rs1取数，预装载到PE array中，等compute指令算完后，将数据结果存至rs2。

机器码：

0001 rs1 rs2 0010 (func=0001，compute指令算完将数据存回rs2之前，需要对结果进行ReLU)

0010 rs1 rs2 0010 (func=0010，compute指令算完将数据存回rs2之前，不需要任何激活函数)

compute rs1 rs2

含义：矩阵A和B在systolic array相乘，A的地址为rs1，B的地址为rs2。

机器码：0001 rs1 rs2 0100

使用这三条汇编指令，即可完成矩阵乘加运算，具体案例见***附录***。

模块功能和接口定义

各个memory、buffer模块

负责数据存储，读写位宽为4*32bit，具体接口如下：

```
module weight_buffer#(parameter depth = 8)
(
    output reg  [127:0] q,
    input  wire         clk,
    input  wire         reset,
    input  wire         ren,
    input  wire         wen,
    input  wire [5:0]   a,
    input  wire [127:0] d
    );
```

PE Array

dataflow类型是output stationary，功能等参考：[脉动阵列 - 因Google TPU获得新生 - 知乎 (zhihu.com)]

注意事项：

1) weight或input从buffer中读出时，4个元素是一起输出buffer的，为了使数据进入systolic array的顺序符合计算要求，需要对其中3个元素插入不同数量buffer；
2) 从systolic array出来的计算结果也需要时序对齐，因此在systolic array的output处也需要插入buffer保证时序；
3) 由于矩阵tile是按照“行”存在各级memory中，你会发现input buffer出来的元素需要转置后才能进入PE array，因此也需要在PE Array中添加transposer模块。

接口如下：

```
module PE_array#(parameter num = 4)
(
    // interface to system
    input wire clk,
    input wire reset,
    input wire c_en,                                // compute enable
    input wire p_en,                                // preload enable
    // interface to PE row .....

    input wire signed[31:0]in_weight[num-1:0],             // wire from weight buffer direction

    input wire signed[31:0]in_input[num-1:0],
                // wire from input buffer direction
    output signed[31:0]result[num-1:0],

    output compute_finished

    );
```

Elementwise Unit

从PE Array中出来的4个元素经过buffer后是时序对齐的，进入Elementwise Unit后根据指令中的func码进行不同的计算操作，最终并行储存到preload指令中定义的rs2地址中（本设计为Shared Memory中的地址）。

Controller

tiny-TPU的具体运行方式是：取指令，译码，执行指令，取指令，译码……

这个功能最简单最直接的实现方法就是FSM，当然进一步优化的空间也很大，比如采用多级流水线的设计方式等。我们为同学们实现了基于FSM的简单Controller，详情见***附录***，大家可以根据自己的思路实现Controller，或者直接复用我们提供的模块。

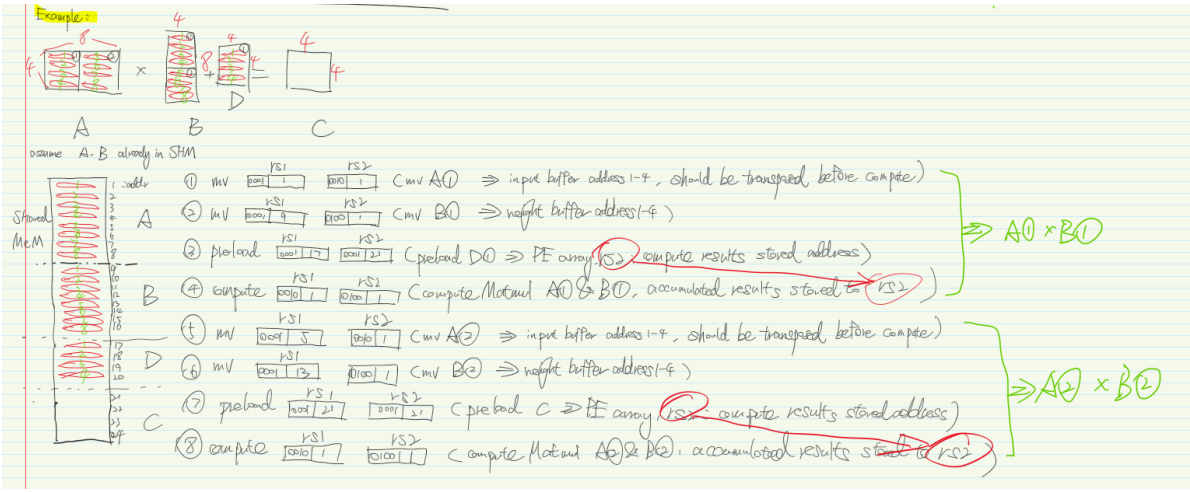
同学们的任务

1) 完成PE Array的RTL设计，PE Array模块需验证通过；
2) 理解整个tiny-TPU的运行机制，将PE Array接入系统中，按照附录中提供的参考汇编代码写一份小程序，使得tiny-TPU执行矩阵乘加的功能；（我们提供模块也许存在bug，如果需要复用，需自行debug）
3) 需要提交：PE Array的源码以及testbench、实验报告一份。

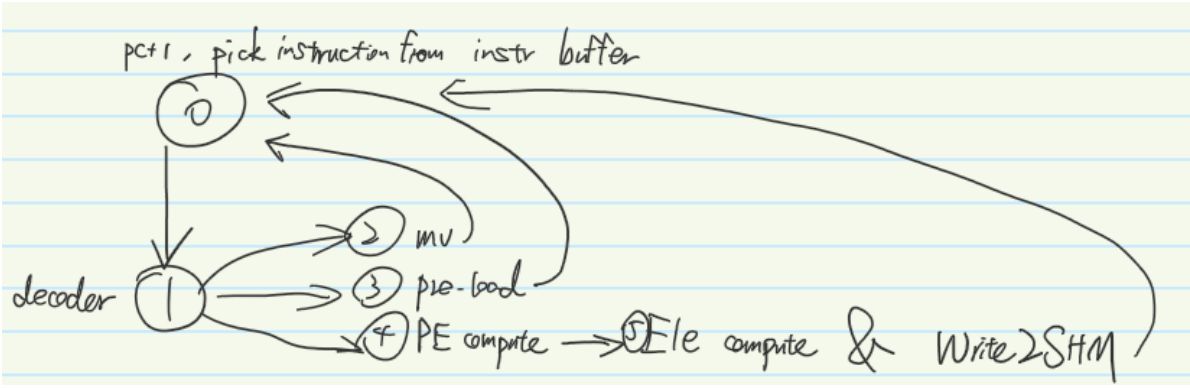
附录

1）为了简单起见，可以把矩阵的size控制在4的倍数。考虑到实际运算中存在不规则大小的矩阵乘，同学们也可以自行完成不规则大小的矩阵乘的指令及架构设计。

使用汇编指令完成AxB+D的矩阵乘加运算，A大小为4x8，B为8x4，D大小为4x4：



2）基于FSM的Controller设计：



我们总结出，基于“取指令，译码，执行指令”的运行方式可以用上图中的FSM实现，其中由于我们的指令有三条，因此译码之后，状态有三个分支。

Controller中维护了这个状态机，这个状态机会根据现有的现有的状态对不同模块的en信号进行赋值，下面有两个例子：1）当在②mv状态时，需要两个memory进行数据传输，因此状态机会将需要读数据的memory的ren（read enable）置为1，会将需要写数据的memory的wen（write enable）置为1；2）在④preload状态下，需要将数据先从相应memory中读出，数据读出的同时需要让数据依次进到PE Array中，因此整个状态的前半段需要把相应memory的ren置为1，后半段需要把memory的ren置为0、PE Array的pen(pre-load enable)置为1。

状态机的状态跳转是根据不同模块的状态计数器实现的。当状态计数器到指定值，则表示当前模块任务完成，跳转到下一个状态。下面有两个例子：1）当在②mv状态时，需要两个memory进行数据传输，第一个memory读数据期间每个周期会将它对应的状态计数器加一，第二个memory写数据期间每个周期会将它对应的状态计数器加一，当两个状态计数器的值到达指定值，则状态从②mv跳转至下一个状态0 pick instruction；2）在④preload状态下，需要将数据先从相应memory中读出，需要进行读操作的memory进行读动作期间每个周期会将它对应的状态计数器加一，PE Array进行preload期间每个周期会将它对应的状态计数器加一，状态机监视到两个状态计数器都达到指定值，则状态从③preload跳转至下一个状态0 pick instruction。