Lab2: 脉动阵列设计与实现

2019011112 无95 郑名宸

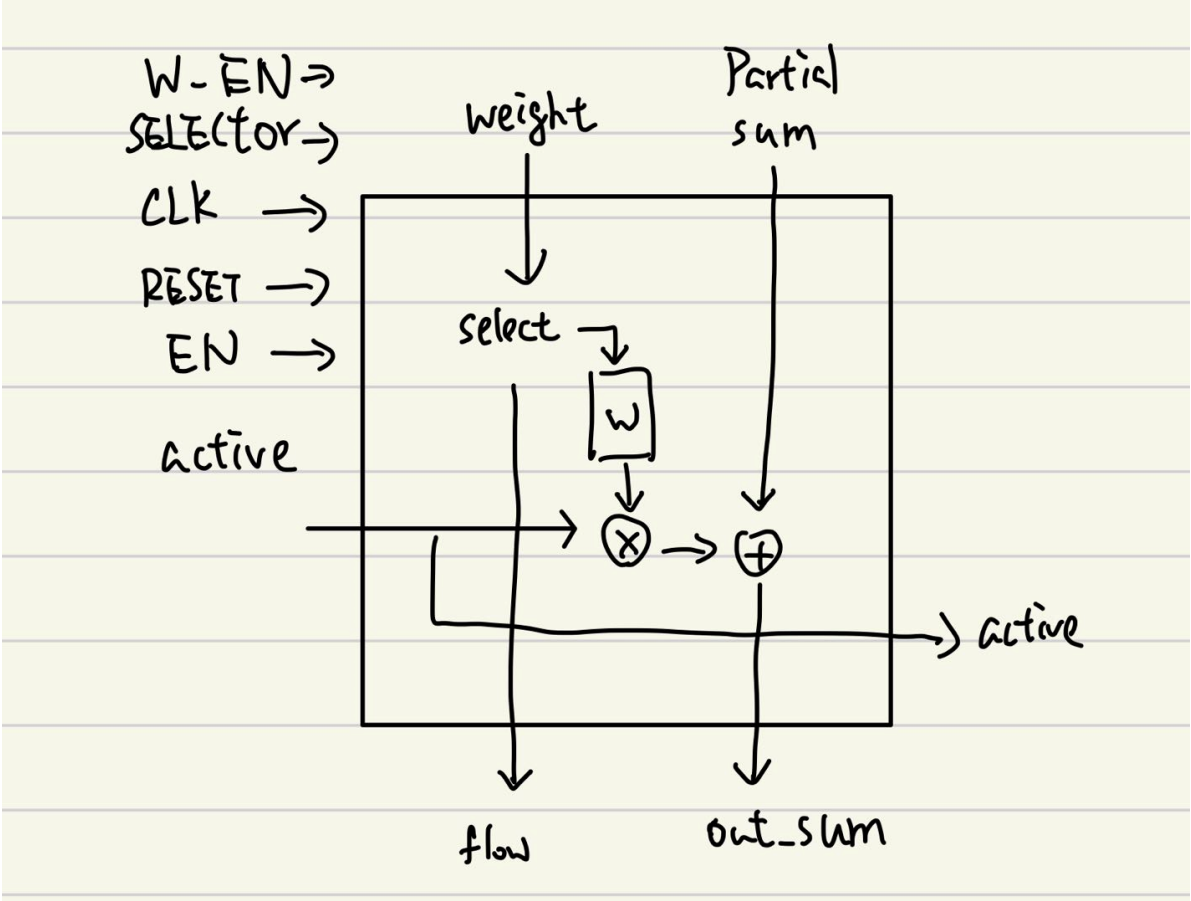
目录

1 单个PE的设计及验证

1.1 PE设计

设计图如下

```
有控制信号5个
reg RESET;
reg CLK;
reg EN;                          // enable signal for the accelerator; high
for active
reg SELECTOR;                    // weight select for read or use
reg W_EN;                        // enable weight to flow
数据流6个
input:
wire signed[7:0]active_left,
wire signed[15:0]in_sum,
wire signed[7:0]in_weight_above,
output:
reg signed[7:0]active_right,
reg signed[15:0]out_sum,
wire signed[7:0]out_weight_below
```



1.2 单个PE验证

testbench如附件pe_testbench.v

仿真结果如下：

可以看到下面四部分验证结果都正确

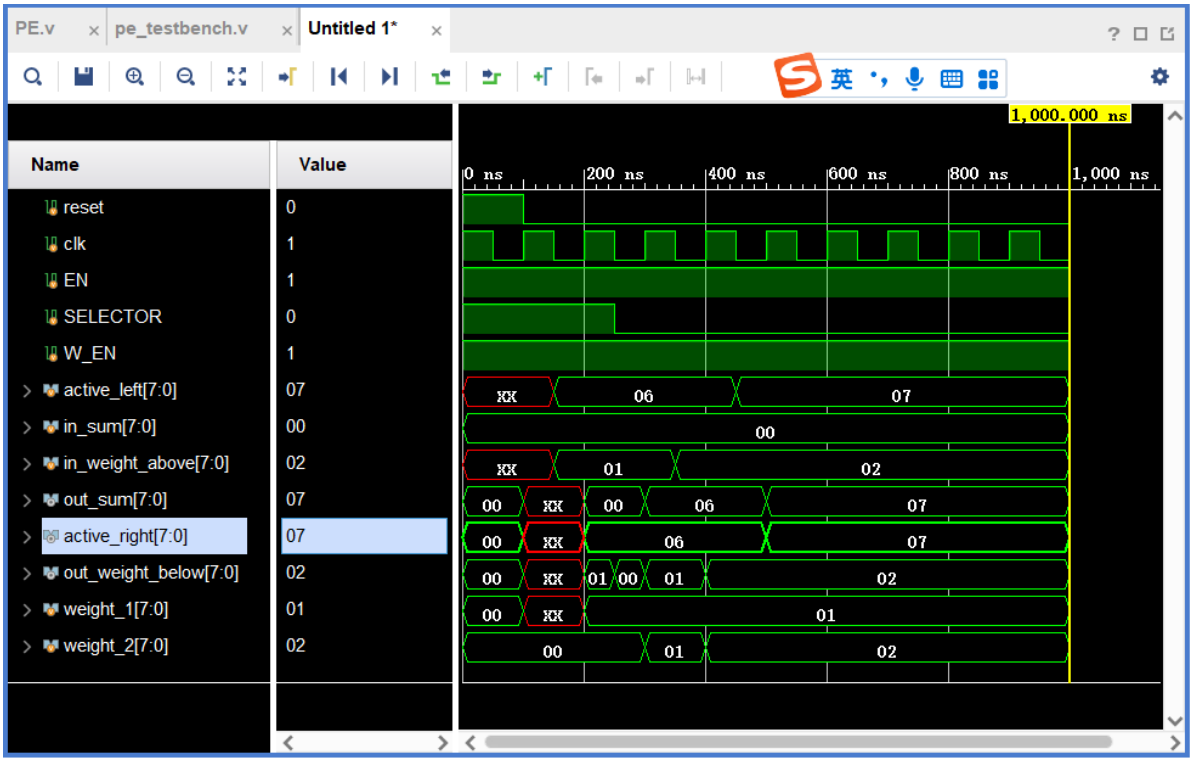
1）计算一个权重与激活值的乘法结果，

2）部分和的累加与存储，

3）权重的写入、互换，

4）权重与多个输入计算时，输入和部分和的传播是否正确。

可以看到改变selector可以切换weight的register，且数据流流动正常



2 Systolic 阵列的设计及验证

2.1 PE行的设计

要特别设计的是最左边的PE,当gi==0，输入是active_left,其余的都是接在左边PE的active_right，有代码如下

```
genvar gi;
generate
    for(gi = 0; gi < 16; gi = gi + 1)   //16 PE
    begin:label
        // some reg/wire variables for each PE
        // .......
        if(gi == 0)begin
            PE PE_unit(
            .CLK(CLK),
            .RESET(RESET),
            .EN(EN),
            .SELECTOR(SELECTOR),
```

```
            .W_EN(W_EN),
            // .....
            .active_left(active_left),
            .active_right(active_right[7:0]),
            .in_sum(in_sum[15:0]),
            .out_sum(out_sum[15:0]),
            .in_weight_above(in_weight_above[7:0]),
            .out_weight_below(out_weight_below[7:0])
            );
        end
        else begin
            PE PE_unit(
            .CLK(CLK),
            .RESET(RESET),
            .EN(EN),
            .SELECTOR(SELECTOR),
            .W_EN(W_EN),
            // .....
            .active_left(active_right[gi*8-1:(gi-1)*8]),
            .active_right(active_right[(gi+1)*8-1:gi*8]),
            .in_sum(in_sum[(gi+1)*16-1:gi*16]),
            .out_sum(out_sum[(gi+1)*16-1:gi*16]),
            .in_weight_above(in_weight_above[(gi+1)*8-1:gi*8]),
            .out_weight_below(out_weight_below[(gi+1)*8-1:gi*8])
            );
        end

    end
```

2.2 PE行的验证

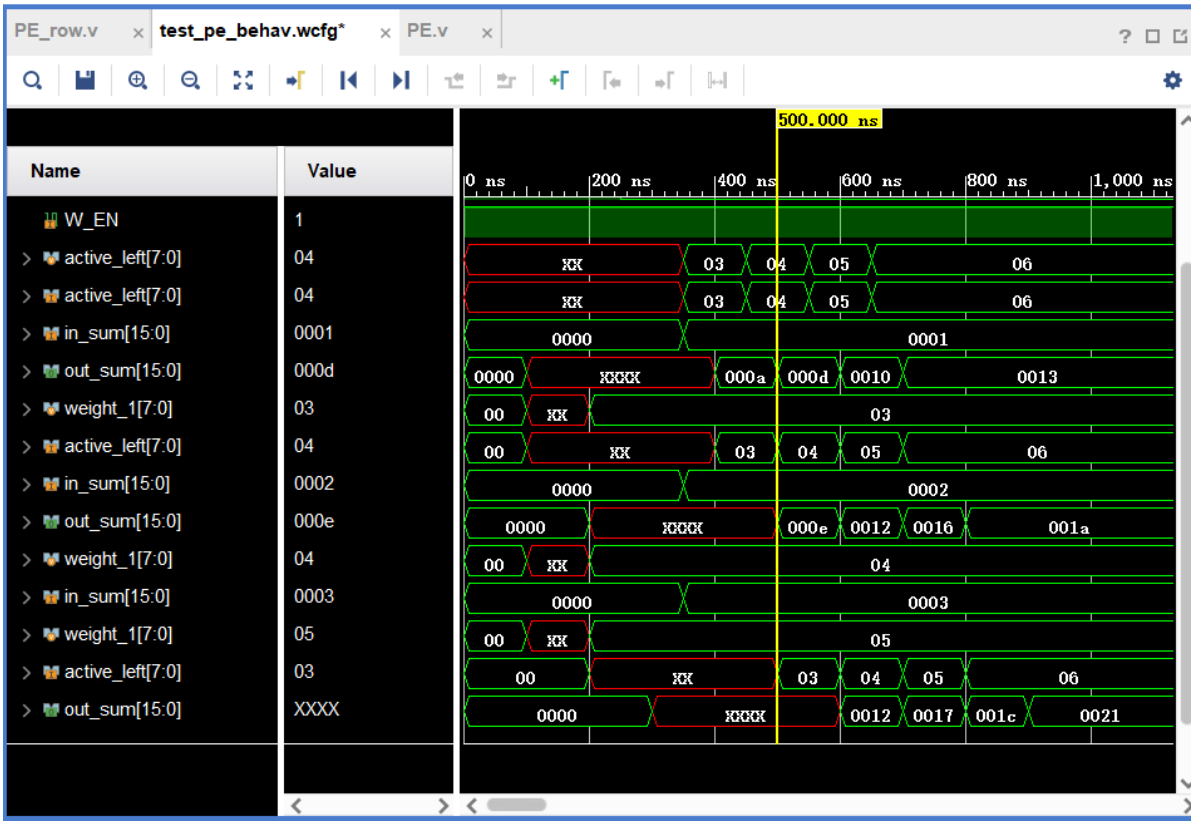
testbench如附件pe_row_testbench.v

1）一个输入与一行权重按照脉动阵列的方式多个周期的脉动计算，以及输出的传递；

2）多个输入与一行权重的脉动阵列结果，以及输出的脉动传递；

3）同时进行权重值的向下传递，与使用另一权重值的计算。

仿真



2.3 PE阵列设计

要特别设计的是最上边的PE row,当gi==0，输入是in_sum,in_weight_above，其余的都是接在上边边PE_row的out_sum和out_weight_below，有代码如下

```
wire [num1*8*num2-1:0]out_weight_below;
wire [num1*16*num2-1:0]out_sum;
// generate of every PE row
genvar gi;
generate
    for(gi = 0; gi < num1; gi = gi + 1)   //16 row
    begin:label
        // some reg/wire variables for each row
        // .......
        if(gi == 0)begin
            PE_row #(.num(num2))PE_row_unit(
            .CLK(CLK),
            .RESET(RESET),
            .EN(EN),
            .SELECTOR(SELECTOR),
            .W_EN(W_EN),
            // .....
            .active_left(active_left[7:0]),
            .in_sum(in_sum),
            .out_sum(out_sum[num2*16-1:0]),
            .in_weight_above(in_weight_above),
            .out_weight_below(out_weight_below[num2*8-1:0])
            );
        end
        else begin
            PE_row #(.num(num2))PE_row_unit(
            .CLK(CLK),
            .RESET(RESET),
```

```
            .EN(EN),
            .SELECTOR(SELECTOR),
            .W_EN(W_EN),
            // .....
            .active_left(active_left[(gi+1)*8-1:gi*8]),
            .in_sum(out_sum[num2*16*gi-1:num2*16*(gi-1)]),
            .out_sum(out_sum[num2*16*(gi+1)-1:num2*16*gi]),
            .in_weight_above(out_weight_below[num2*8*gi-1:num2*8*(gi-1)]),
            .out_weight_below(out_weight_below[num2*8*(gi+1)-1:num2*8*gi])
            );
        end

    end
```

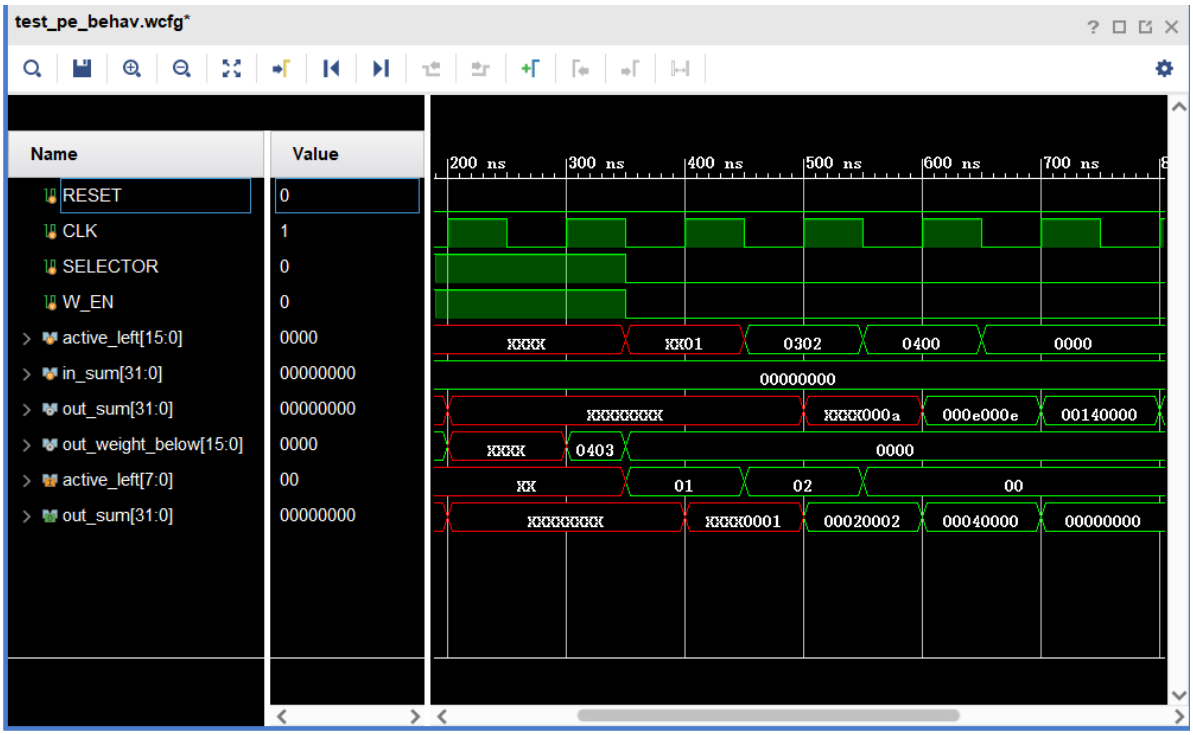
2.4 PE 阵列的验证

testbench如附件pe_array_testbench.v

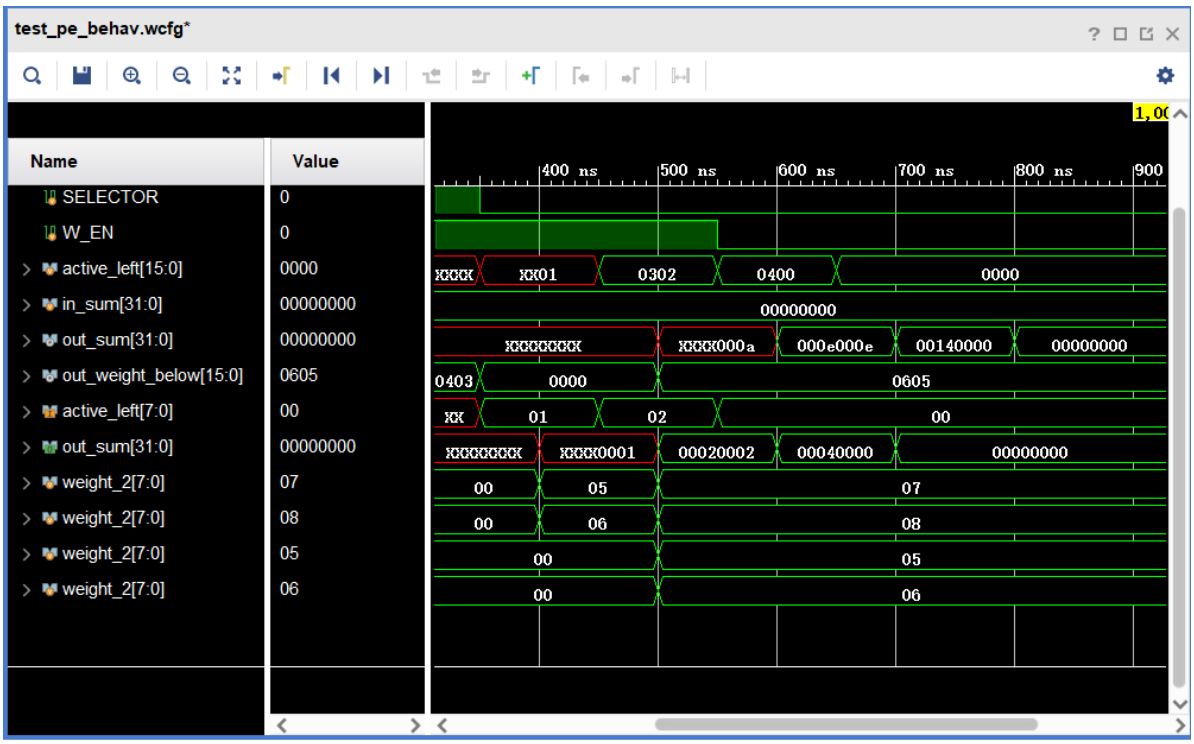
为了方便验证，将阵列设为2*2来进行验证

1）2x2矩阵权重值的初始化；

2）验证矩阵乘AxB=C，其中A,B均为2x2的矩阵；

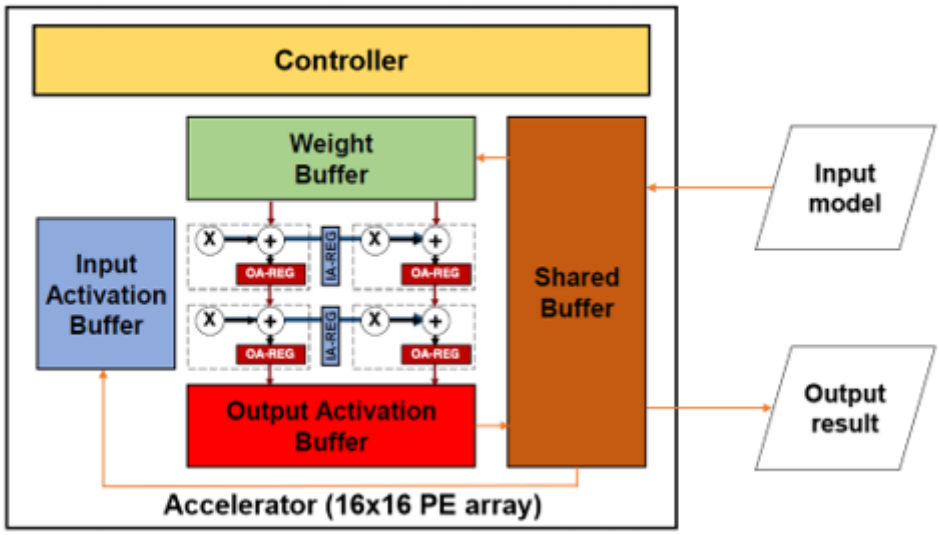


3）验证在计算的同时，进行另一矩阵D的初始化，初始化完使用W_EN让weight流动停下来方便检查。



3 支持单层运算的加速器整体架构的设计及验证

3.1 整体架构的设计



整体架构如上图，具体设计如下

```
buffer分配:
使用sram作为buffer，原本想用fifo作为buffer，但是readmemb使用sram比较方便
因为buffer总共是32kb，我将容量均分，
weight-buffer，input Activation Buffer，output activation buffer各10kb
```

具体的状态机如下面的controller state，我采用大循环和小循环状态来控制，大循环是从外面的DRAM读入数据到share buffer，接着计算，小循环是重复计算直到当前载入share buffer的数据都算完了，接着跳回大循环。

```
controller状态:
STATE:
IDLE        //初始idling的状态加上变量初始化
```

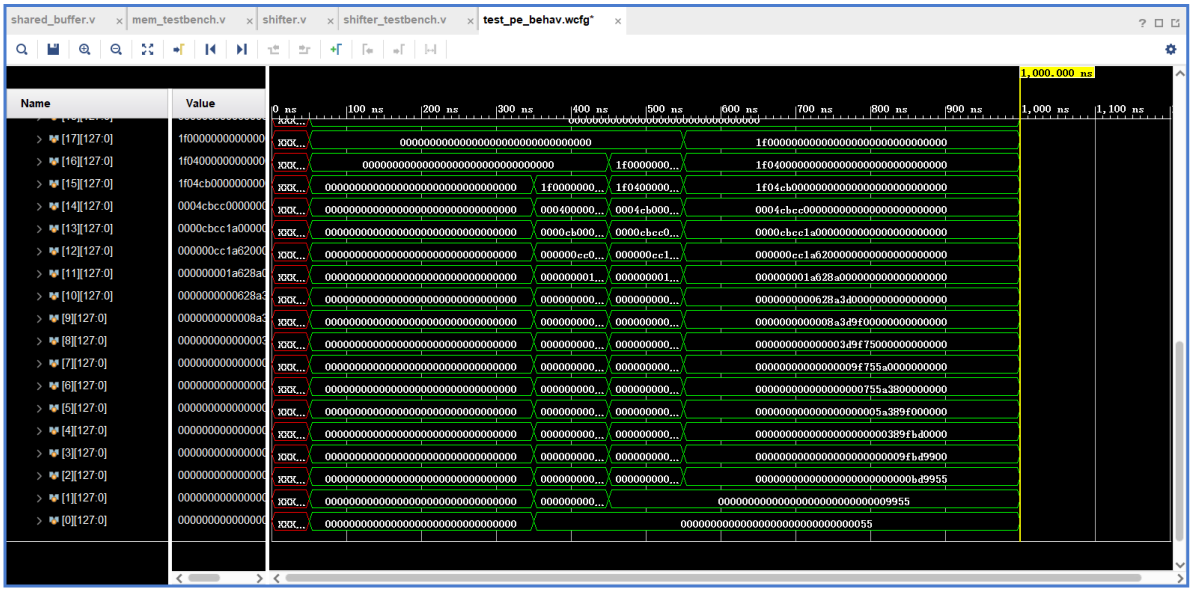
```
INPUTSW      //从外面dram载入weight到share buffer
INPUTSA      //从外面dram载入activate到share buffer
INPUTA       //share buffer 到 activate buffer
INPUTW       //share buffer 到 weight buffer，此时，可以将weight流入PE阵列
OUTPUT-BUFFER //将output输出到share buffer
CALCULATE    //脉动阵列计算，
             //此时weight 可以继续加载
INPUTS       //载入share buffer的状态
OUTPUTS      //将share buffer里面的output输出到.MEM
RETURN       //如果share buffer的数据没有用完，则回到状态input weight
             //如果share buffer的数据用完，则到回到状态IDLE，此时把output buffer的数据输出到module外
```

3.2 整体架构的验证

3.2.1 buffer的编写

首先是activate_buffer的编写，我以sram为基础将shifter的操作直接加在了里面，

仿真文件为shifter_testbench.v,下面尝试输入三行的数据，成功shift了



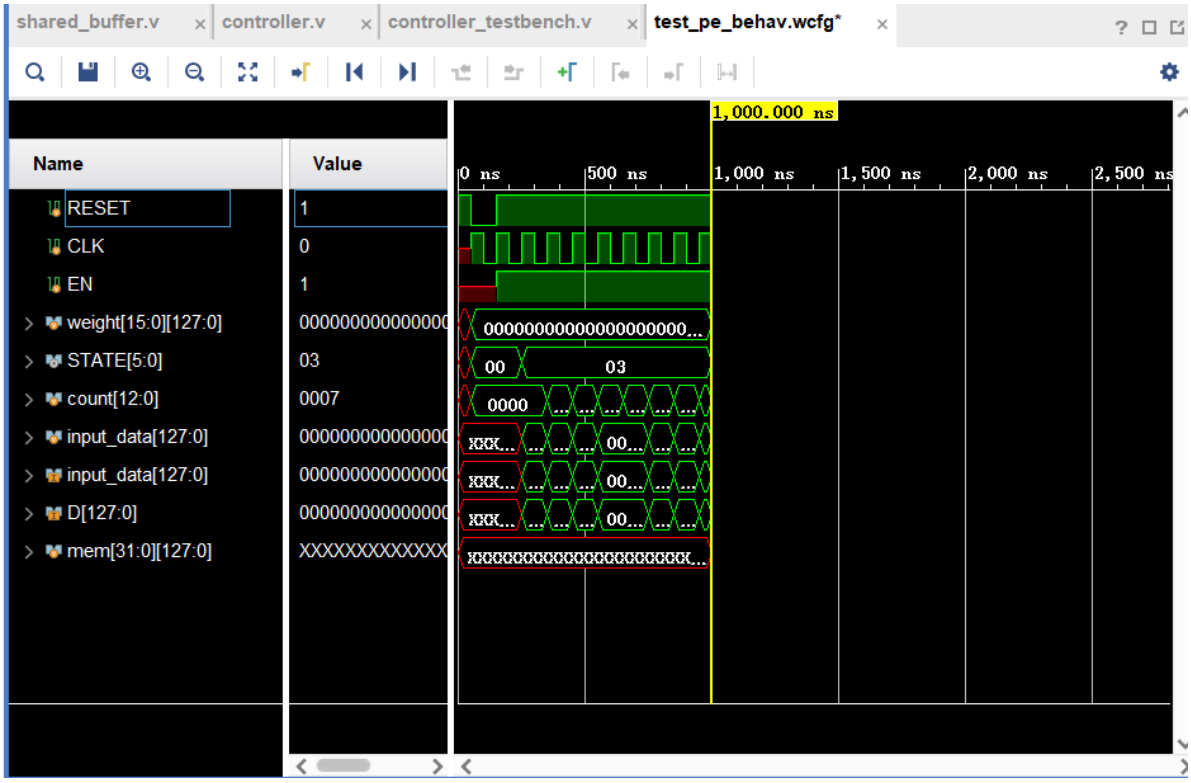
其他的buffer则是使用sram，因为上面等同测试过了，所以不再验证

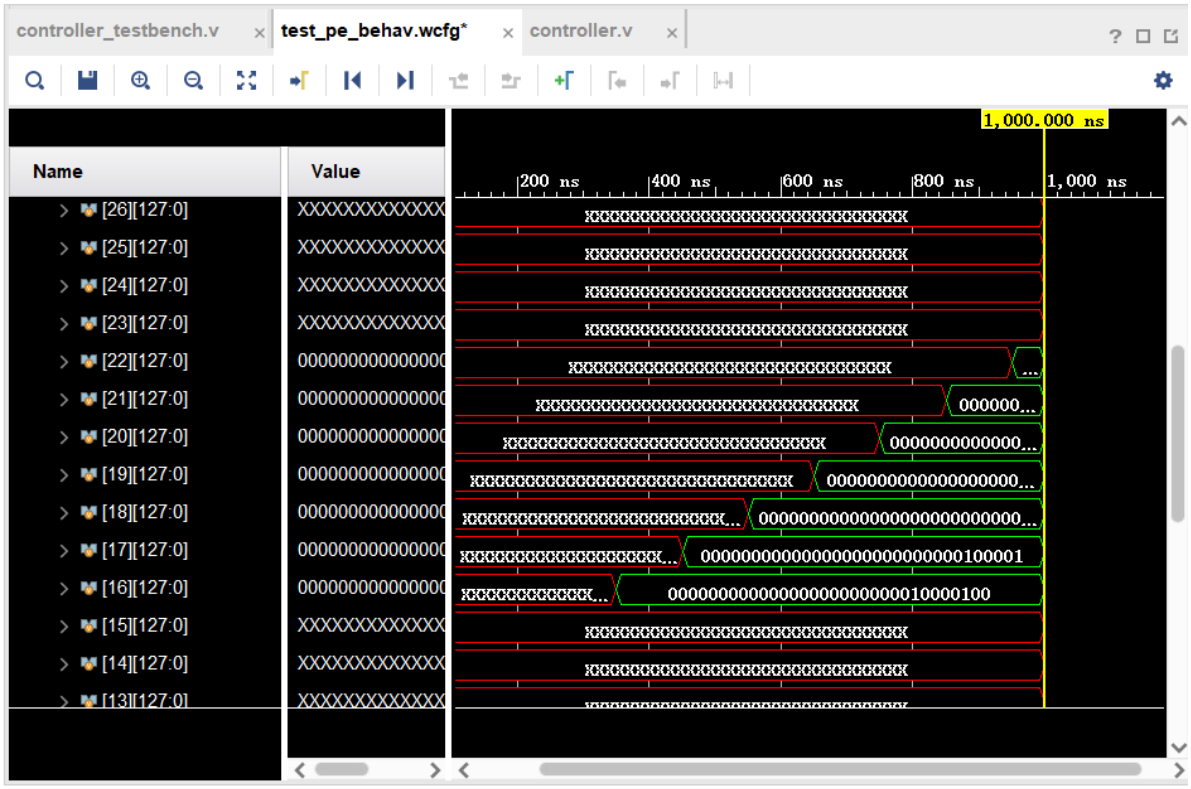
3.2.2 controller测试

1 状态INPUTSW和INPUTSA

首先是状态INPUTSW，通过testbench输入的地址，动态的调整activate和weight在share buffer的存入大小，下图可以看见，weight正常输入了share buffer的指定位置，同理inputsa输入activate进入share buffer也同理。

testbench见附件controller_testbench.v





2 INPUTW 和INPUTA状态

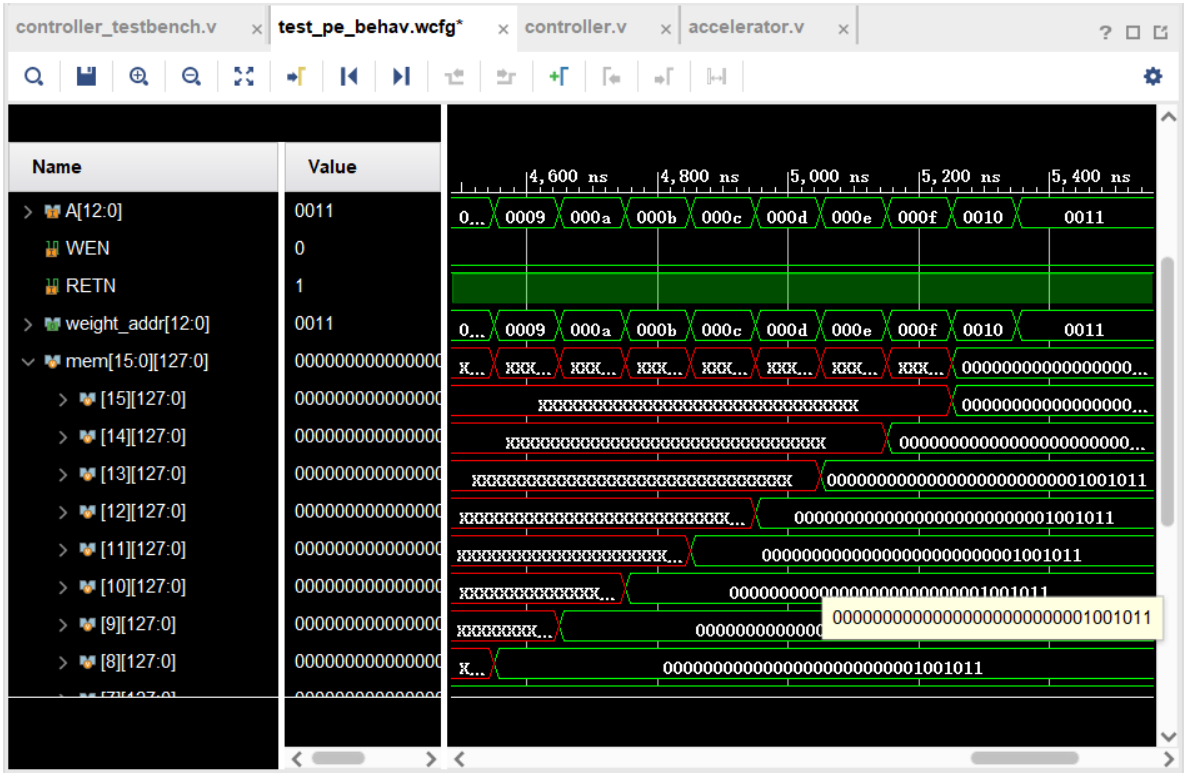
接着是share buffer到weight buffer和activate buffer的通路。

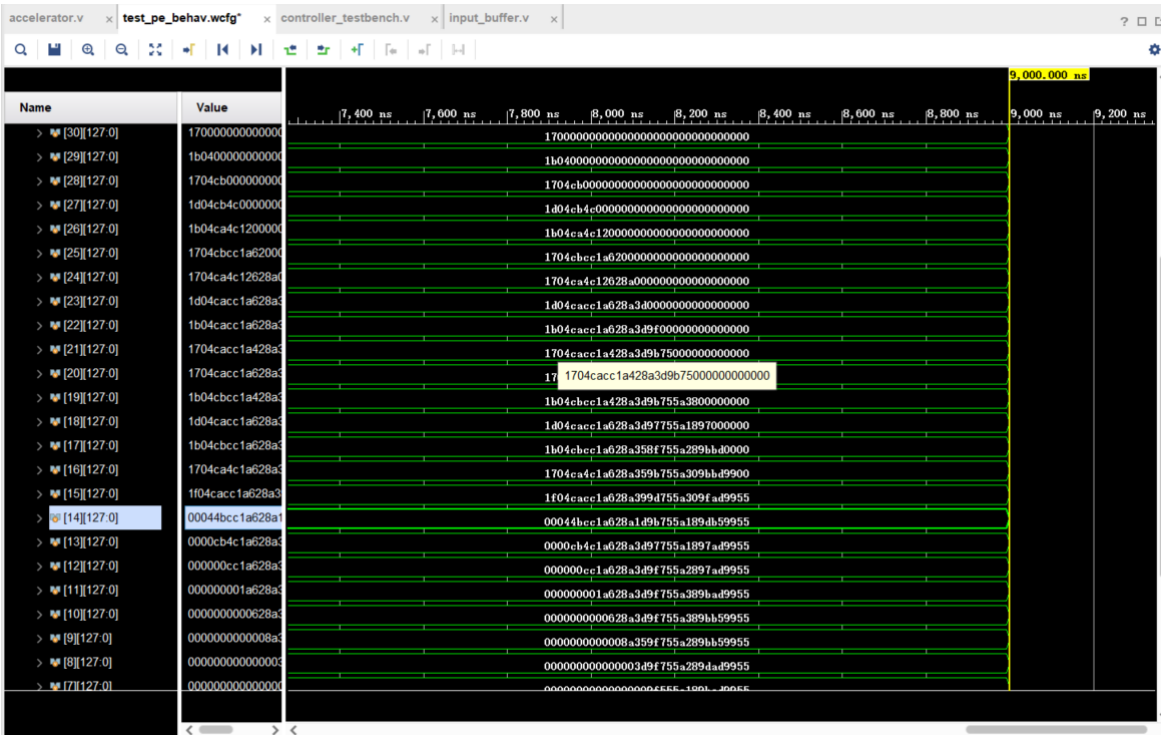
值得一提的是，当操作share buffer 到 input buffer时，可以同时将weight流入PE阵列

下图1可看见，成功从share到weight buffer

下图2可看见，成功从share到input buffer，并且成功完成shifter的操作

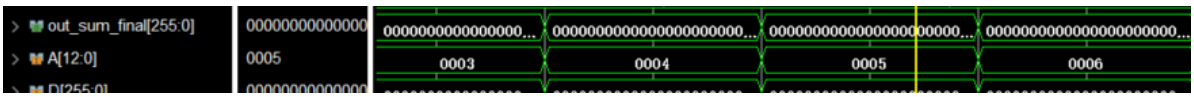
testbench见附件controller_testbench.v





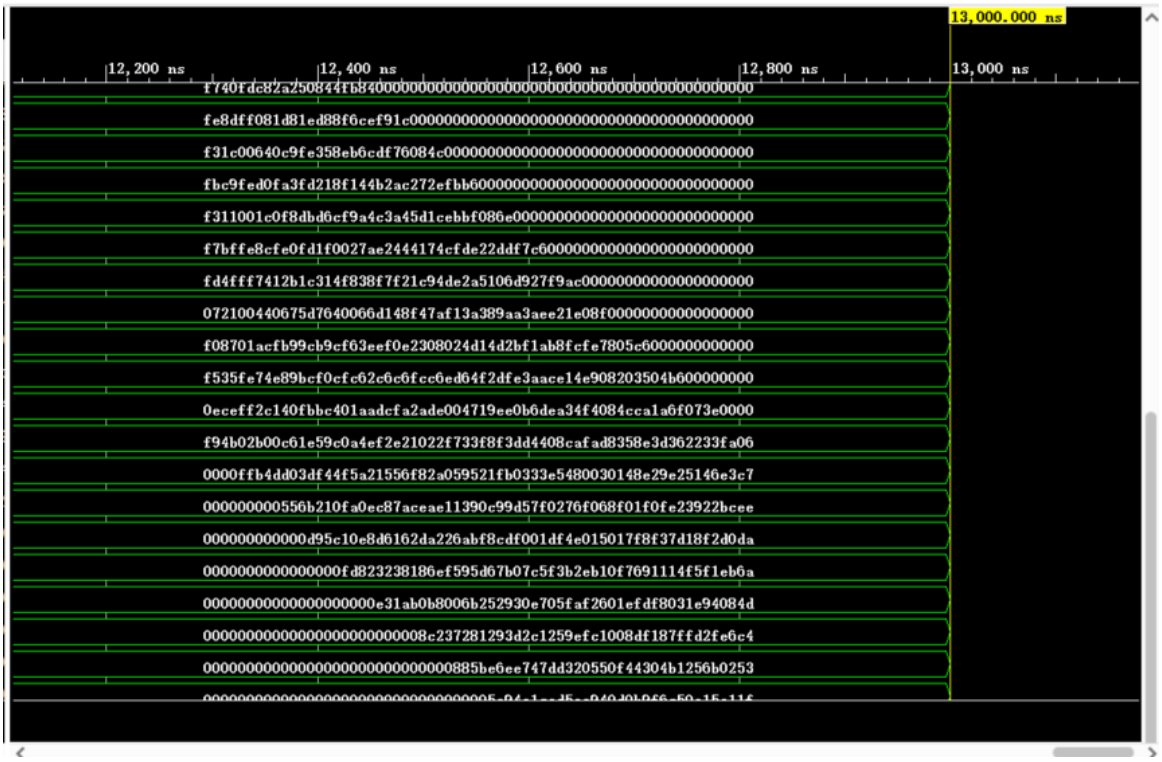
3 CALCULATE状态

此状态和上面测试pe array一样，这边确认out sum final有输出

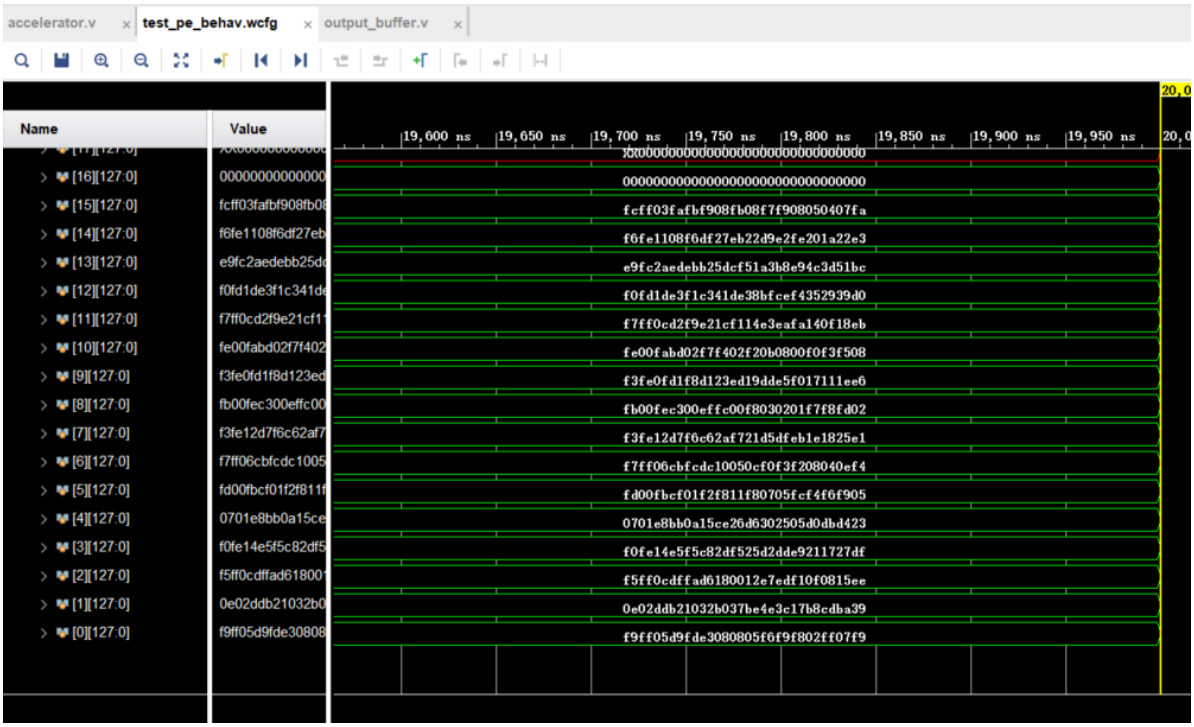


4 output

最后输出如下，之后再过个shifter即可将平行四边形的output矫正



我将逆shifter和output buffer融合后可以方便算出，此外我还将量化的取前8位的操作也融合在了output buffer中。输出变为下图



5 RETURN状态

重复的状态机

如果share buffer的数据没有用完，则回到状态input weight

如果share buffer的数据用完，则到回到状态input share buffer，此时把output buffer的数据输出到module外面

4 思考题

4.1 如何设计支持多层，对硬件架构有什么影响?

如果要支持多层的网络卷积，本质上卷积就是一种矩阵操作运算，如果采用的是这样的脉动阵列，我觉得是controller和buffer要适当的调整，因为不同网络层的特点是不一样的，比如对于super resolution这种任务，他的层一般是activation会特别大，但weight会比较小。对于classification的网络，一般weight量会特别大。因此如果某些需求特别极端，可能需要适当增加buffer大小。接着是如果要适应不同的层的需求，controller不能再是状态机的模式，可能需要变为指令的方式，用以适应不同的任务。

4.2. batchsize>1的时应该如何设计硬件架构?

如果batchsize大于1，等同是activate增加了，此时如果要保持同样的时间算完，可以增加脉动阵列的核数。然而我觉得这样并不本质，也可以使用同样的架构，但就是多算几次而已。因此我觉得现在的硬件架构是可以做的。

4.3 限于时间，目前的加速器主要由有限状态机控制器来完成控制。在实际的硬件中，需要使用指令对加速器进行控制。针对脉动阵列架构，我们需要设计怎样的指令集？对应到硬件，需要增加/修改哪些硬件模块?

如果需要改成由指令来对加速器进行控制，主要就是controller需要做改变。

首先是硬件模块上的改动，会像cpu一样，需要一个instruction memory来储存指令。

再来是指令集的设置。可以仿造现在的状态机，将状态转化为指令，但不同的是，我现在写的状态机的参数是写死的，像是一次载入share buffer的数据量是固定的，但如果是指令的方式，可以形成带参数的模式，方便来适应不同的任务。