Tiny TPU

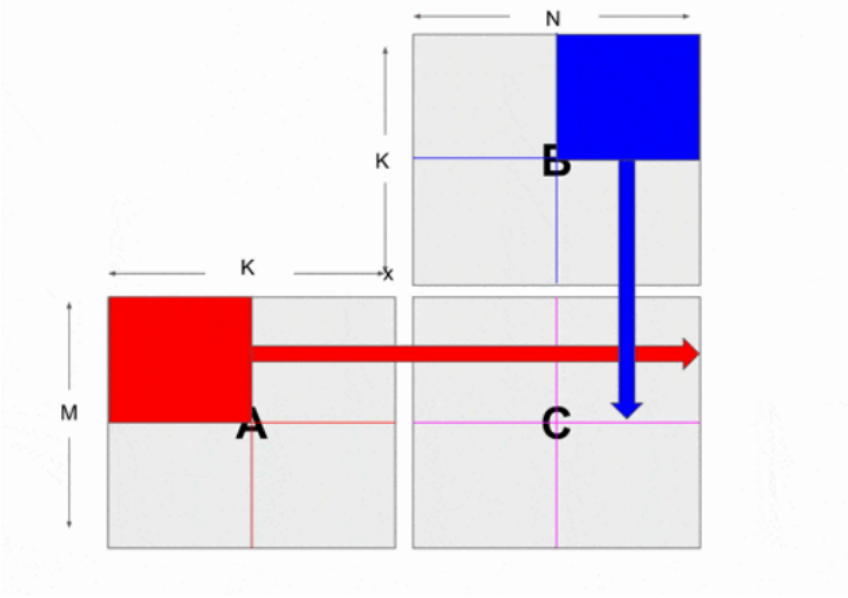
1 Introduction

1.1 Why we need TPU

Tensor Process Unit is an application-specific integrated circuit(ASIC) designed for AI acceleration by Google, especially for neural networks.

The most common type of computation in neural networks is **matrix multiplication**. Matrix multiplications are mainly accelerated by *general matrix multiply* method. GeMM introduces a good method called **Tiling**, which first divide the output matrix into small tiles, then perform matrix multiplication on each tile, as is shown in the graph blow. Tiling achieves great trade-off between memory locality and parallelism.

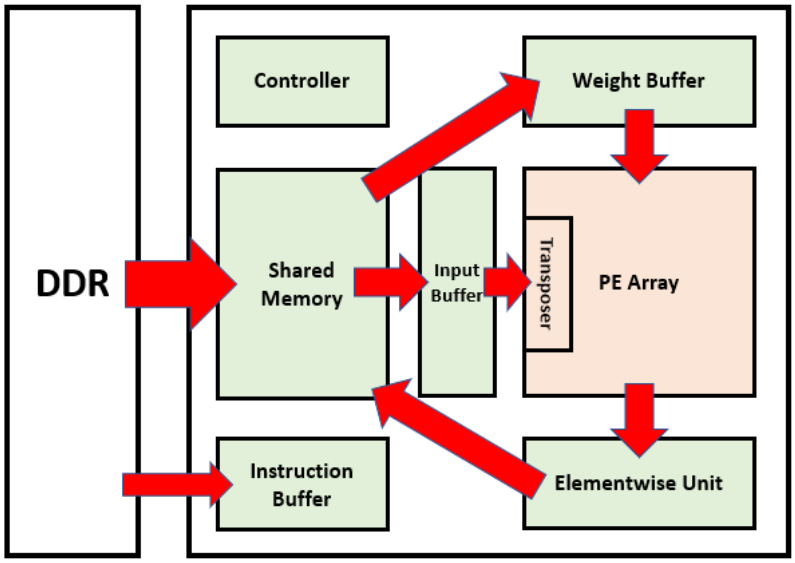
The critical factor of tiling method is the speed of computation of a single tile. As a result of this, TPU was designed.



https://spatial-lang.org/gemm

1.2 What is Tiny TPU

Tiny TPU is a simple implementation of tensor process unit. It could compute the a 4×4 matrix at a time and compute relu activation function. The architecture of Tiny TPU is shown below, which involves 7 parts: (1)Controller and (2)Instruction Buffer for control logic; (3)Shared Memory, (4)Input Buffer and (5)Weight Buffer for the acceleration of memory access; (6)PE Array and(7) Element-wise Unit for matrix multiplication and activation respectively. Tiny TPU is written by a *SystemVerilog*.



The Systolic-Array-based PE Array is the core of Tiny TPU. The dataflow of it is *output stationary*. The principle of systolic array is introduced in the next section.

My implementation has some *disadvantages*. First, it doesn't involve a DRAM, and I just use `$readmemb` in *SystemVerilog* to simulate DRAM access. Second, the controller is a simple Finite State Machine(FSM) and could only execute instruction sequentially. Moreover, this implementation is barely synthesized and simulated but never runs on a real FPGA. The logic of this implementation is correct but further work must be done to make it work in the real world.

2 Systolic Array

Systolic Array uses a pipeline approach to process elements. There are four pictures in Appendix to help understand the principle of it. If you never heard of it before, this blog will do great help.

3 Modification of ISA

Specifying whether to use relu activation in *preload* instruction is neither natural nor convenient. The reasons are as follows. First, extra work must be done to save the *func* of *preload* instruction. Second, there are cases when there are no matrix needed to be preloaded and *preload* is used only for specifying output address.

As a result, I extend the functionality of *preload* and hand over the activation specifying work to *compute*. Since *mv*, *mv*, *preload*, and *compute* must work as a whole, this modification simplifies design and extends functionalities at the same time.

- preload rs1 rs2
  - 0001 rs1 rs2 0010: preload the matrix in r1
  - 0010 rs1 rs2 0010: preload 0 matrix
- compute rs1 rs2
  - 0001 rs1 rs2 0100: with relu
  - 0010 rs1 rs2 0100: without relu

4 Implementation

4.1 PE Array

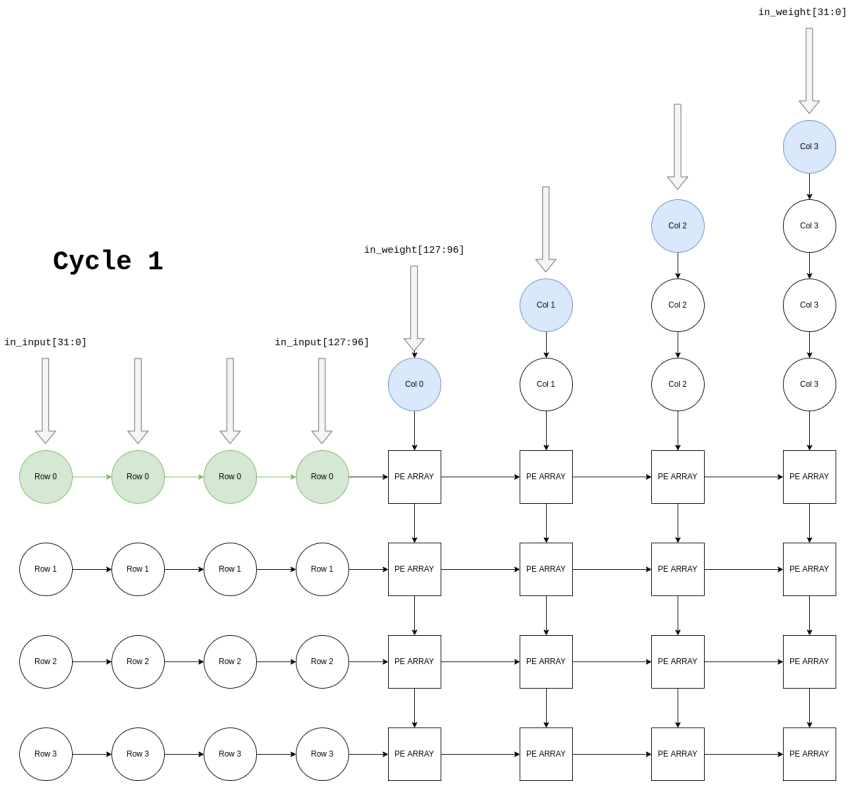
PE Array computes the product of two matrices, named as *Input* and *Weight* respectively. To be more specific, PE Array computes $Input \times Weight$. The inputs of PE Array are six signals: *in_weight*, *in_input*, *c_en* and *p_en*, as well as *clk* and *reset*. *in_input* is a 128 bit(4×32 bit) signal representing a line of *Input*, and *in_weight* is also a 128 bit signal representing a row of *Weight* with the same index. *c_en* and *p_en* are state enable signals representing *Compute* mode and *Preload* mode respectively.

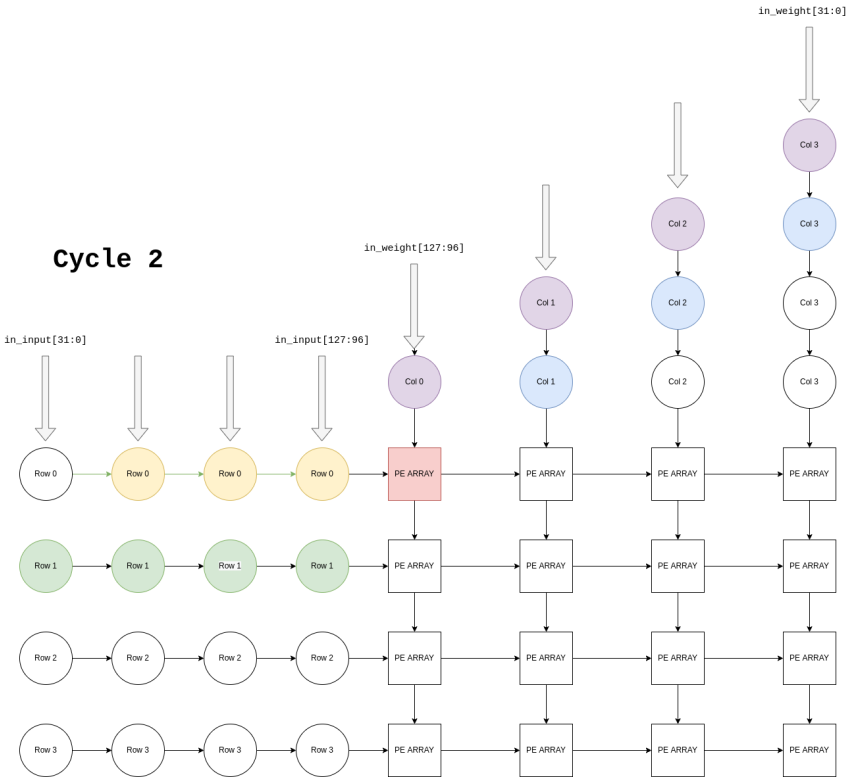
4.1.1 Compute State

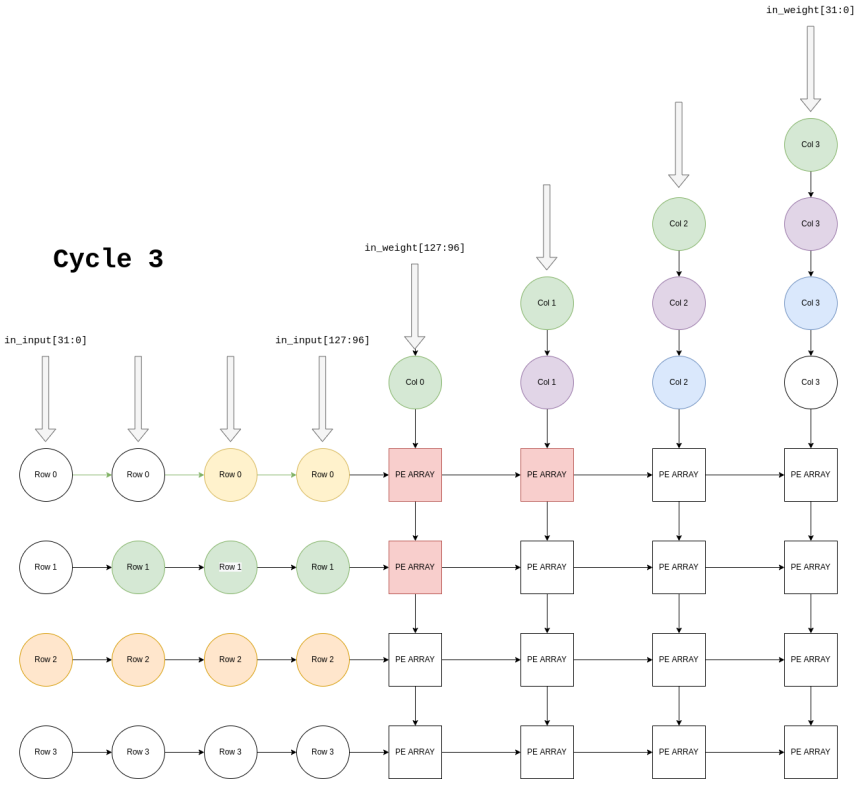
In each clock cycle, a row of *Input* and a row of *Weight* could be loaded into PE Array from input wires. Including the first cycle which reads out data from buffers, the two matrices are all loaded in PE Array in the first five cycles.

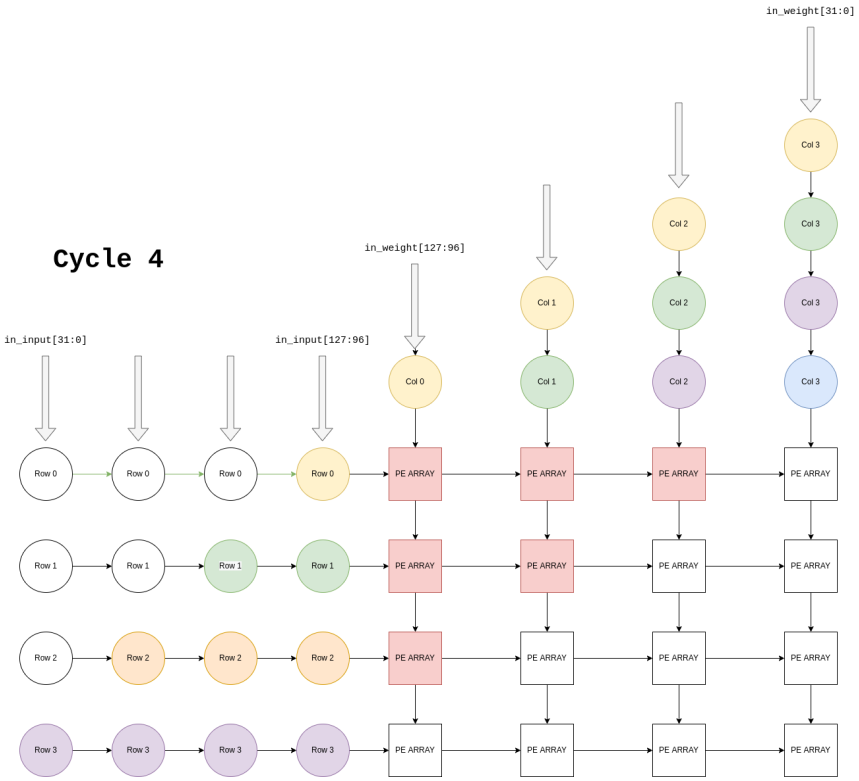
To support the systolic way of data loading, 8 groups of shift registers are used, 4 for *Input* and 4 for *Weight*. Because PE Array could not load a row of *Input* and its corresponding column of *Weight* at one time, the arrangement of shift registers are different for *Input* and *Weight*.

The arrangement and loading schema are shown in the following graphs. The circles with the same color represent a row of *Input* or *Weight*. Moreover, the computation and data loading could be done parallelly, so the first computation unit(PE) starts working in Cylce 2.




Cycle 2
in_weight[31:0]
in_weight[127:96]
in_input[31:0]
in_input[127:96]
Col 0
Col 1
Col 2
Col 3
Row 0
Row 1
Row 2
Row 3
PE ARRAY


Cycle 3
in_weight[31:0]
in_weight[127:96]
in_input[31:0]
in_input[127:96]
Col 0
Col 1
Col 2
Col 3
Row 0
Row 1
Row 2
Row 3
PE ARRAY



After Cycle 4, no input is needed. Each PE takes 1 cycle to do a multiplication. The multiplication of the two matrices ends after cycle 11.

In conclusion, the execution state of *compute* instruction takes 12 cycles in all. The main **drawback** of this implementation is the imbalanced I/O and computation. I/O and computation are done in parallel in the first five cycles, but in the later 7 cycles no I/O is involved.

There are two possible improvements:

1. do more than one multiplication in a clock cycle during the last 7 cycles
2. load the *Input* and *Weight* of next tile in the last four clock cycles

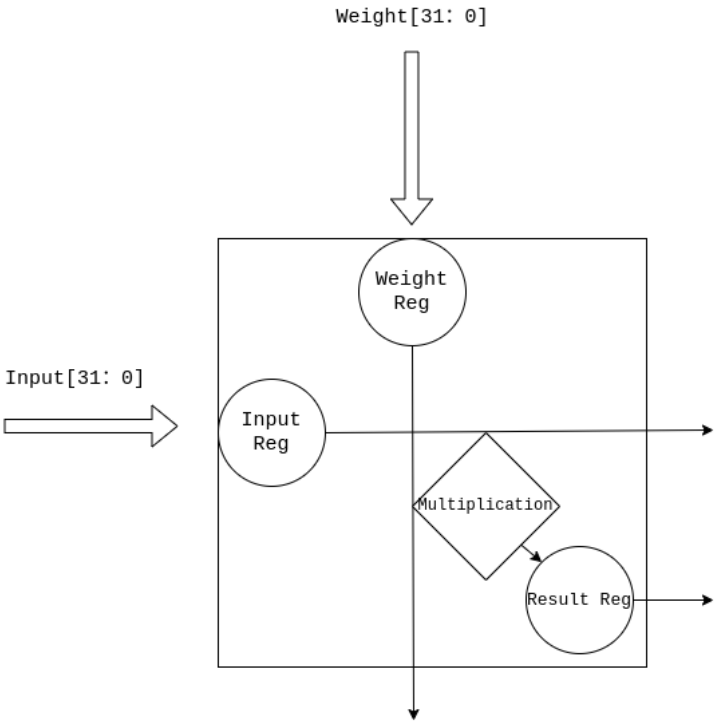
4.1.2 Preload

When *p_en* signal is enabled and *c_en* is disabled, the input wires of *Weight* is connected to the registers inside each PE units. Whether the *Weight* wires are connected to outer shift registers or those inside PE units is determined by *p_en* and *c_en*, using combinational logic.

The execution state of *preload* instruction takes 5 cycles in all. In the first cycle, the first row is read from shared memory. Each cycle in the later 4 cycles is to load a row to PE Array.

4.2 PE Unit

The architecture of PE unit is shown below, including three registers for inputs and accumulated result, and a multiplier.

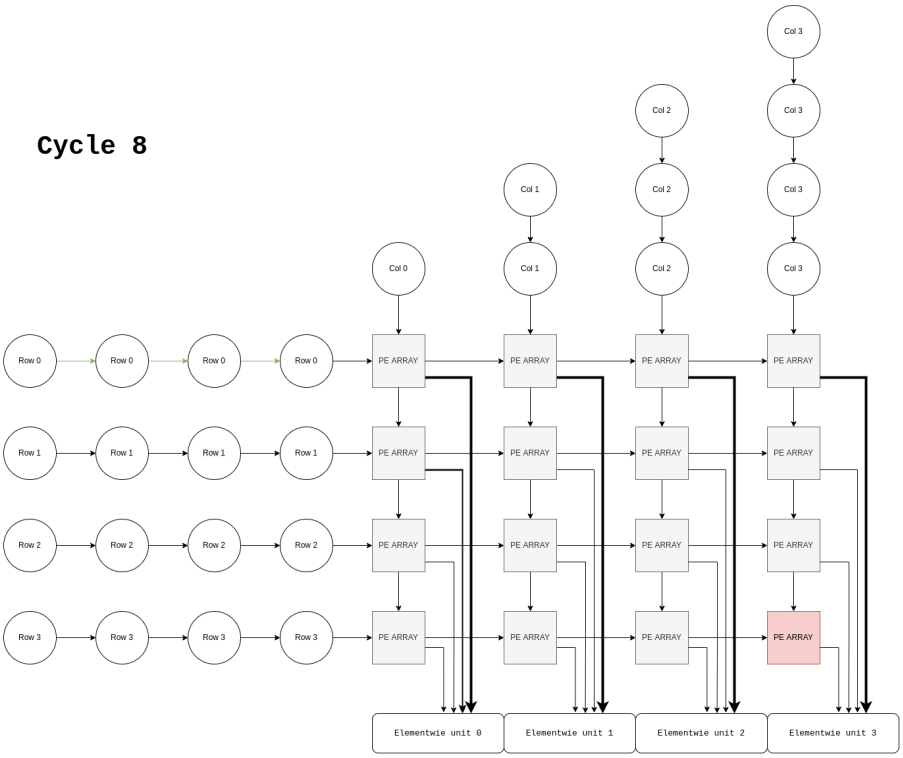


It's worthing noting that although *Input Reg* and *Weight Reg* are inside PE unit logically, they are defined in the PE Array in the concrete implementation, to make register shifting in an easier way.

4.3 Element-wise Unit

The element-wise unit could process 4 elements at a time, computing the relu activation function or not according to *func* code of *compute* instruction.

The method of element-wise unit fetching result from PE Array is shown below. The output wire of each PE result is connected with the input wire of the corresponding Element-wise Unit.



In this version of implementation, Element-wise unit fetching start in cycle 11. Actually it could start in cycle 8 in order to reduce the total clock cycles of this state from 4 to 1.

5 Test Bench

The test bench is in file `tb_top.sv`. The contents of *Instruction Buffer* and *Shared Memory* are in file `instructions.dat` and `shared_memory_contents.dat` respectively.

Don't forget to change the file path of `.dat` files in `tb_top.sv` before testing.

First, the test computes

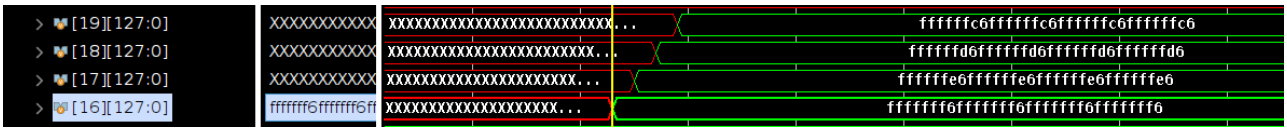
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -10 & -10 & -10 & -10 \\ -26 & -26 & -26 & -26 \\ -42 & -42 & -42 & -42 \\ -58 & -58 & -58 & -58 \end{bmatrix}$$

Then it computes

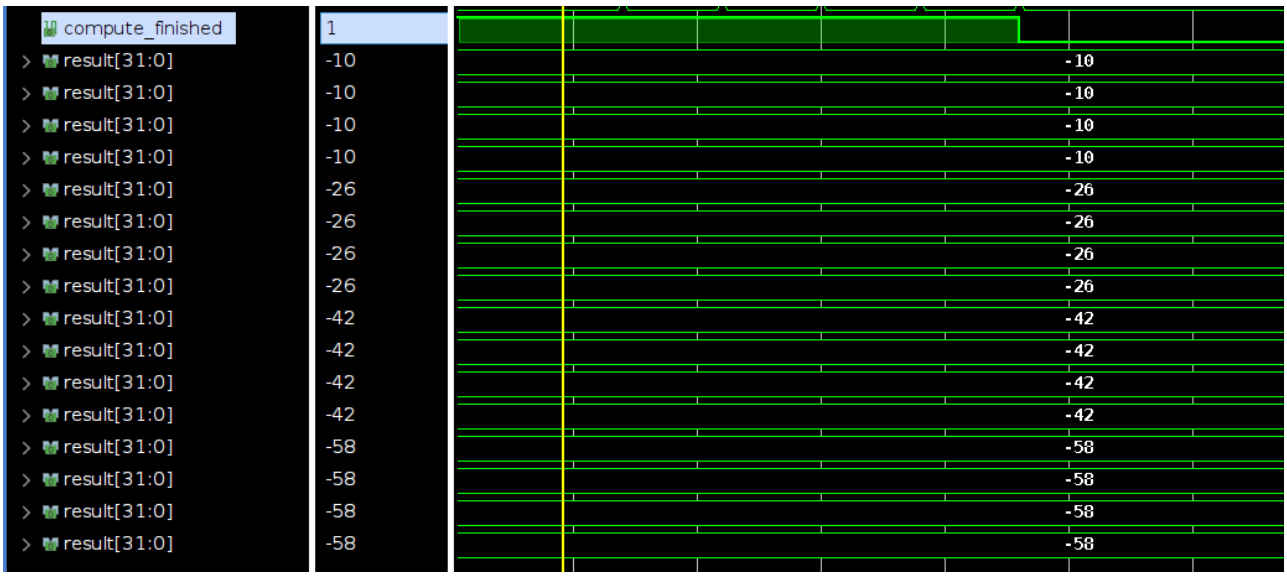
$$\begin{bmatrix} -10 & -10 & -10 & -10 \\ -26 & -26 & -26 & -26 \\ -42 & -42 & -42 & -42 \\ -58 & -58 & -58 & -58 \end{bmatrix} + relu(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{bmatrix})$$

After the first stage, the results in PE Array and Shared Memory are correct, which are shown below:

Memory

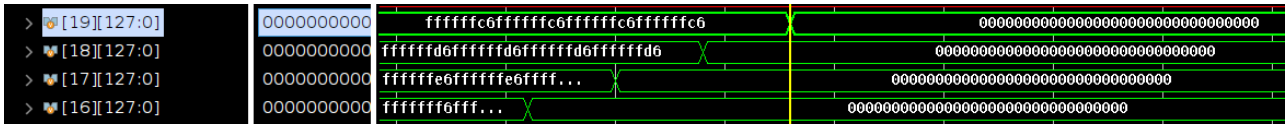


PE Array

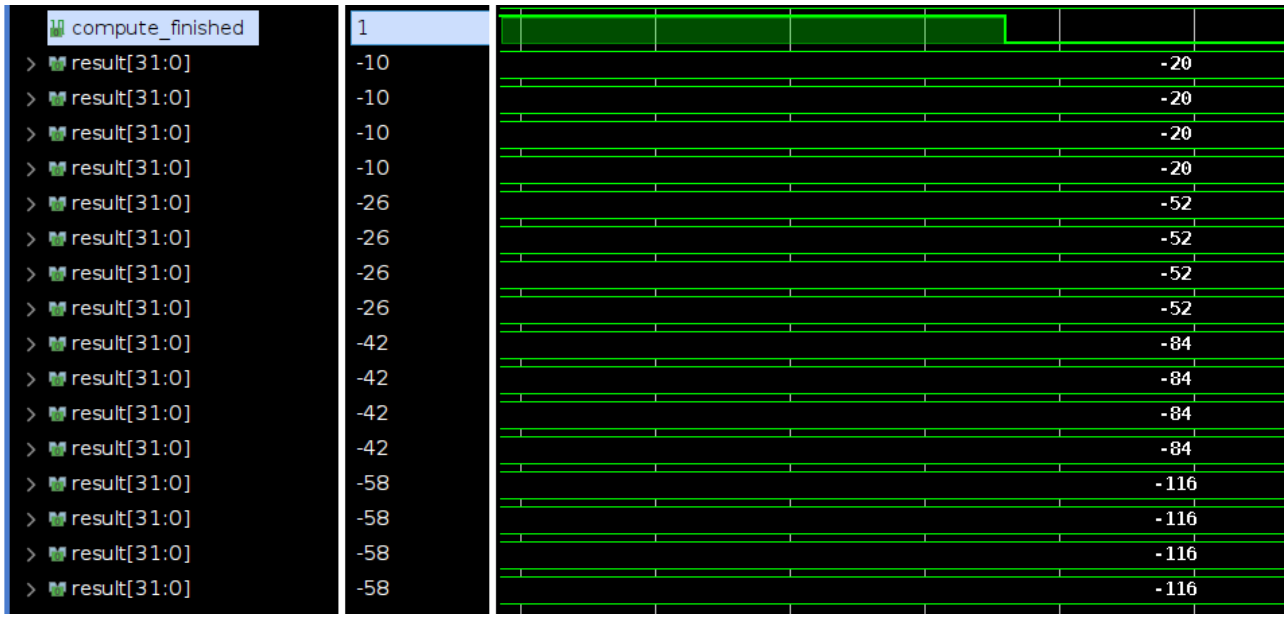


At the end of the second stage, results in PE Array and Shared Memory are shown below:

Memory



Shared Memory



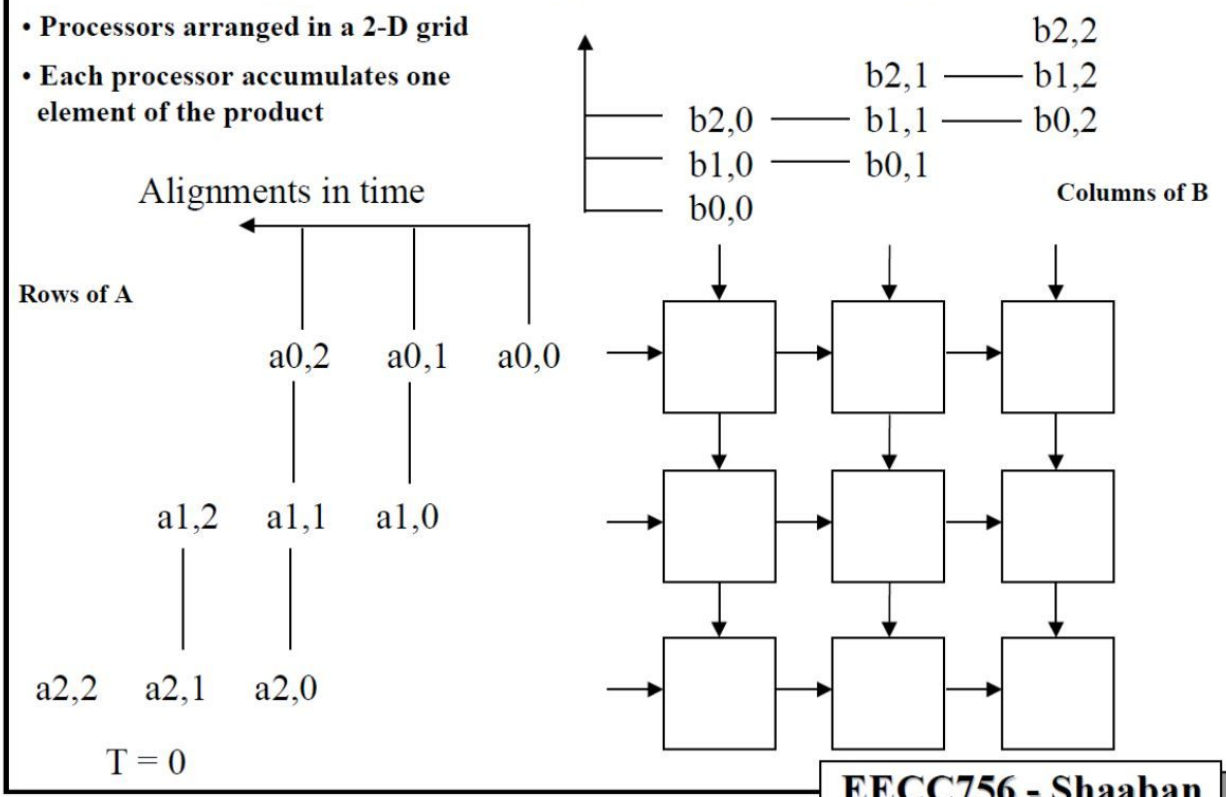
6 Conclusion

Through this project, I acquired many valuable knowledge about TPU design and principle of systolic array. Furthermore, I learned somewhat about Verilog RTL languag. The most important thing is the thinkings in hardware programming and algorithm-hardware co-design, which is totally different from software design patterns I have learned before.

7 Appendix

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

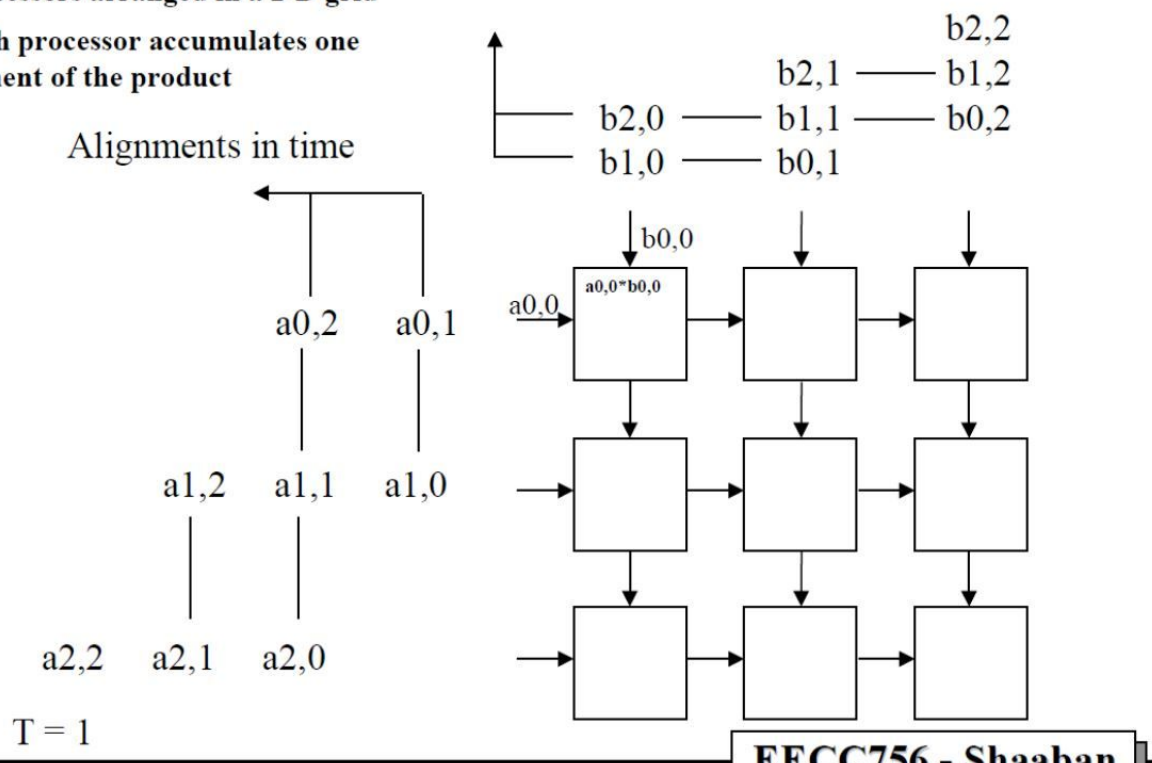
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

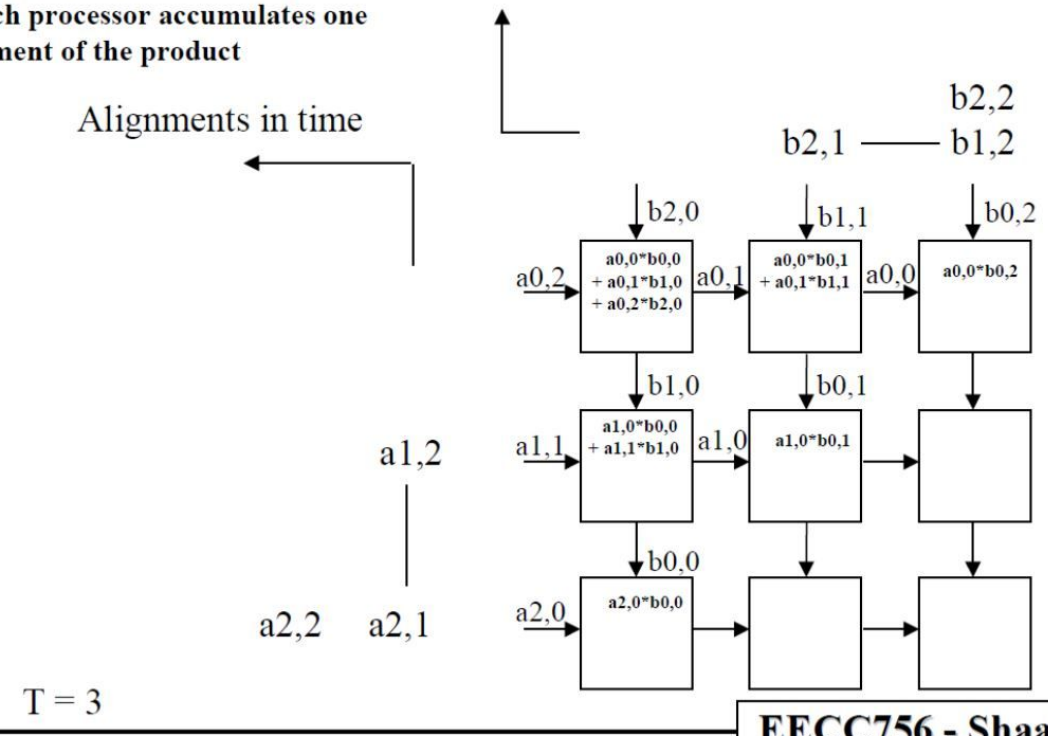
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

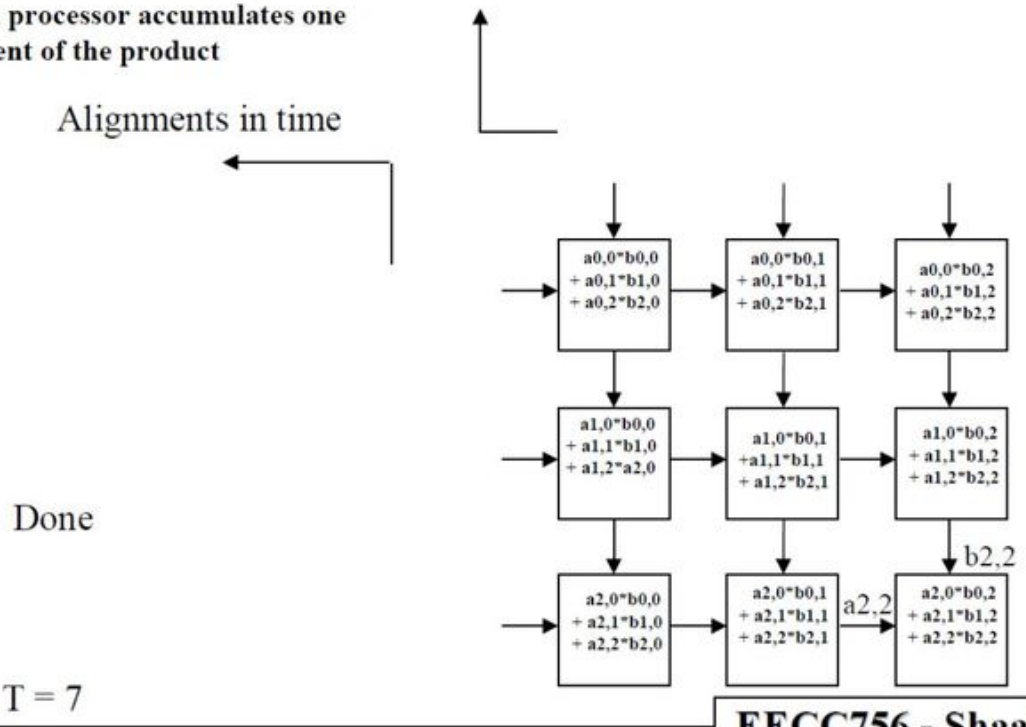
- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/

Systolic Array Example:
3x3 Systolic Array Matrix Multiplication

- **Processors arranged in a 2-D grid**
- **Each processor accumulates one element of the product**



EECC756 - Shaaban

Example source: http://www.cs.hmc.edu/courses/2001/spring/cs156/