Lab_4: Matrix Multiplication module

Introduction

Matrix multiplication is a popular kernel in high-performance scientific computing, gaming, and even machine learning workloads. Companies like NVIDIA now build GPU hardware that excels at the task of performing matrix multiplication. In contemporary usage, matrix multiplication hardware has even made it into the core of the Google Tensor Processing Unit (TPU). FPGAs are also competent at matrix multiplication, particularly from the perspective of energy efficiency.

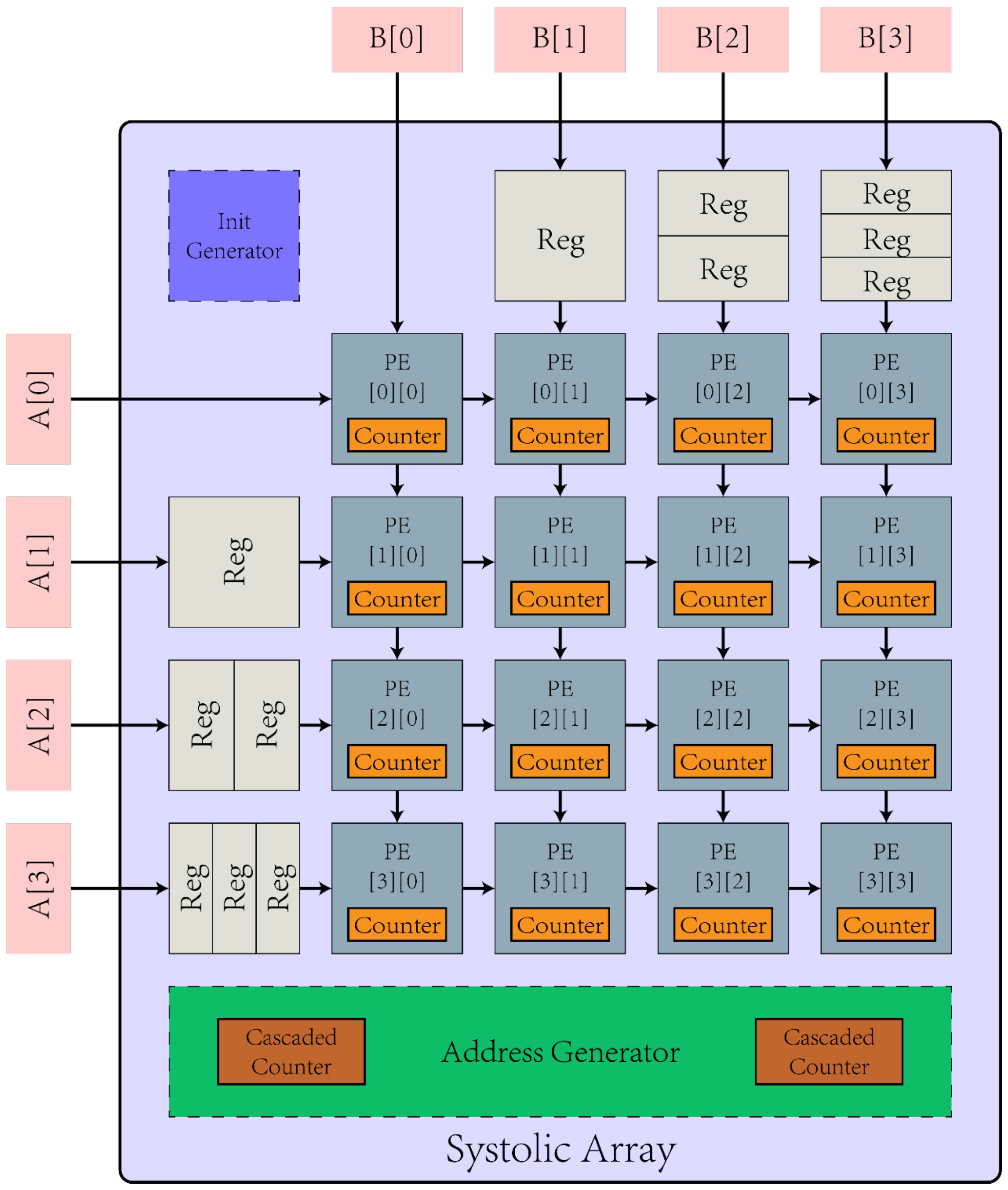
Objective

This lab's objective is to design a hardware module targeting FPGAs that multiply two matrices in a systolic fashion. A systolic array is a 2D grid of simple computing elements connected in nearest neighbor fashion. Dataflow through the array proceeds in a systolic fashion (one hop at a time), new elements injected into the array from the top, and left flanks per cycle. The co-ordination of data injection is crucial for correct evaluation of the computation.

Design Description

We show a high-level picture of the systolic array below. The **systolic core, the PEs(Processing Elements), and the counter** are the three key building blocks of your design. We have given the design code templates in the **Lab4_handin.zip**. Please download and study it as soon as possible.

The dashed box (*Init Generator* and *Address Generator*) represent functional modules in the block diagram, but they are not extracted as an entity in the code templates. You can fill it directly to the position of the dotted box (*Systolic Array*).



Specific tasks

- Design a systolic matrix multiplication module in the file *systolic.vhd* to instantiate a 2D grid of PEs defined in *pe.vhd*. You will need to use **generate statements** in VHDL to construct the array of parametric dimensions. Intermediate signals are also needed to help wire-up the systolic array correctly. The systolic core will also instantiate the *Init Generator* and *Address Generator* (which call *Cascaded Counter* coded in *counter.vhd*) to generate the control signals required to operate logic within the array.
- Implement the PE module *pe.vhd* that performs the multiply-accumulate operation. A clocked MAC module computes $out\_sum = out\_sum + in\_a \times in\_b$ on a stream

of input ports *in_a* and *in_b* to generate a stream of output port *out_sum*. An *init* signal will be provided to controls when the accumulation starts. PE module is the most basic computing unit in this system.

- Implement a cascaded counter module *counter.vhd* to generate the address of input data. We should increase the higher digit's counter for a cascaded counter when the lower digit's counter is rolling over. Here, the higher digit's counter of the cascaded counter should have a *count_enable* control port named *enable_row_count* in the given code(Lab4_handin.zip).

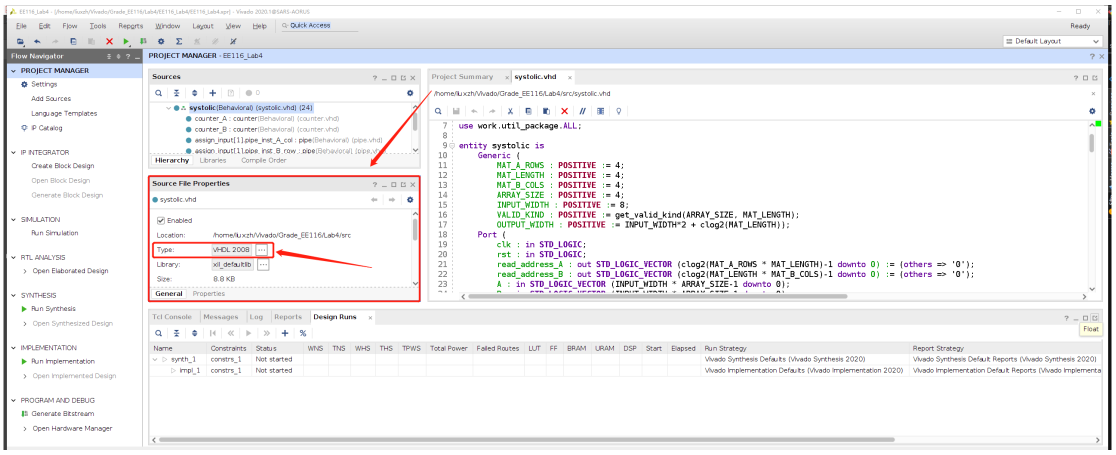
- Implement a shift register FIFO to control the cycle of data arrival. In the calculation of systolic arrays, timing is essential. This requires you to control the arrival time of each data to get the correct result. The first-in-first-out shift register is a cascade of a series of registers, which accepts one data in each cycle and simultaneously outputs one data. By adjusting the FIFO's depth, you can control the number of cycles that data takes from entering to reading out.

- Simulate your module for functional correctness by using the testbench written by yourselves. Ensure that the generated result matches the expected golden result. Also, ensure that the source code can be synthesized.

Note

The behavior described in this part is only a hint for the final implementation, not a restriction. If the material description conflicts with the given code (Lab4_handin.zip), the given code shall prevail.

The type of all given code files must be switched to VHDL 2008 in Vivado as shown in figure below.



Sources: systolic(Behavioral) (systolic.vhd) (24)
- counter_A : counter(Behavioral) (counter.vhd)
- counter_B : counter(Behavioral) (counter.vhd)
- assign_input[1].pipe_inst_A_col : pipe(Behavioral) (pipe.vhd)
- assign_input[1].pipe_inst_B_row : pipe(Behavioral) (pipe.vhd)

Source File Properties: systolic.vhd

Enabled	✓
Location:	/home/liu xzh/Vivado/Grade_EE116/Lab4/src
Type:	VHDL 2008
Library:	xil_defaultlib
Size:	8.8 KB

/home/liu xzh/Vivado/Grade_EE116/Lab4/src/systolic.vhd

```
use work.util_package.ALL;

entity systolic is
    Generic (
        MAT_A_ROWS : POSITIVE := 4;
        MAT_LENGTH : POSITIVE := 4;
        MAT_B_COLS : POSITIVE := 4;
        ARRAY_SIZE : POSITIVE := 4;
        INPUT_WIDTH : POSITIVE := 8;
        VALID_KIND : POSITIVE := get_valid_kind(ARRAY_SIZE, MAT_LENGTH);
        OUTPUT_WIDTH : POSITIVE := INPUT_WIDTH*2 + clog2(MAT_LENGTH));
    Port (
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        read_address_A : out STD_LOGIC_VECTOR (clog2(MAT_A_ROWS * MAT_LENGTH)-1 downto 0) := (others => '0');
        read_address_B : out STD_LOGIC_VECTOR (clog2(MAT_LENGTH * MAT_B_COLS)-1 downto 0) := (others => '0');
        A : in STD_LOGIC_VECTOR (INPUT_WIDTH * ARRAY_SIZE-1 downto 0);
```

Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy	Report Strategy
synth_1	constrs_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2020)	Vivado Synthesis Default Reports (Vivado Synthesis 2020)
impl_1	constrs_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2020)	Vivado Implementation Default Reports (Vivado Implementa

Systolic Core

The core computation for matrix multiplication is performed by a 2D systolic array with parameterizable design size $ARRAY\_SIZE \times ARRAY\_SIZE$. The left column and top row of the array will stream input matrices $matrix\_A$ and $matrix\_B$. The result matrix $matrix\_D$

is read from the 2D pe array in-place. For this lab, there is no readout circuit for sending $matrix\_D$ out of the array.

Building the systolic engine

The matrix $matrix\_A$ and $matrix\_B$ will use VHDL custom data type 2D array construct, as the provided testbench shows. Each value of these matrix has a configurable bit-width $INPUT\_WIDTH$. In each cycle, the testbench will read the address output by the systolic array (by output port $read\_address\_A$ and $read\_address\_B$), find the corresponding data, and transfer it to the systolic array through the input port A and B. When connecting the external ports to the 2D grid of processing elements, you need to connect columns of $A[N-1:0]$ and $B[N-1:0]$ to the corresponding lanes of the design. For example, for a 4×4 systolic dimension, $A[0]$ will connect to $PE[0][0]$, $A[1]$ will connect to $PE[1][0]$ as shown in the Figure above. Do not forget to add a suitable FIFO between the input port and PE.

We must then connect the PEs in a systolic fashion to each other and include data forwarding logic to send the inputs to the next PEs along the row and column. You must use the VHDL to **generate** a construct to create a parametric 2D grid of PEs. Ensure that you define and declare custom data type 3D signals to help stitch horizontal and vertical traffic lanes. Each PE will store a partial product as the result of its multiply-accumulate operation. For our 4x4 systolic array, this will mean we will have 16 partial products at a point in time: One partial product for each PE. Once our four rows of A and four columns of B have been streamed in, distributed to the PE in systolic fashion, and streamed out, the resulting D output will be available "in-place" across all PEs. The results are exposed as a custom data type 3D signal and transfer to the STD_LOGIC_VECTOR output port at the top-level of *systolic.vhd*. The testbench will read data from each PE in parallel via the output port and determine which array these data should be stored in.

Streaming Matrices

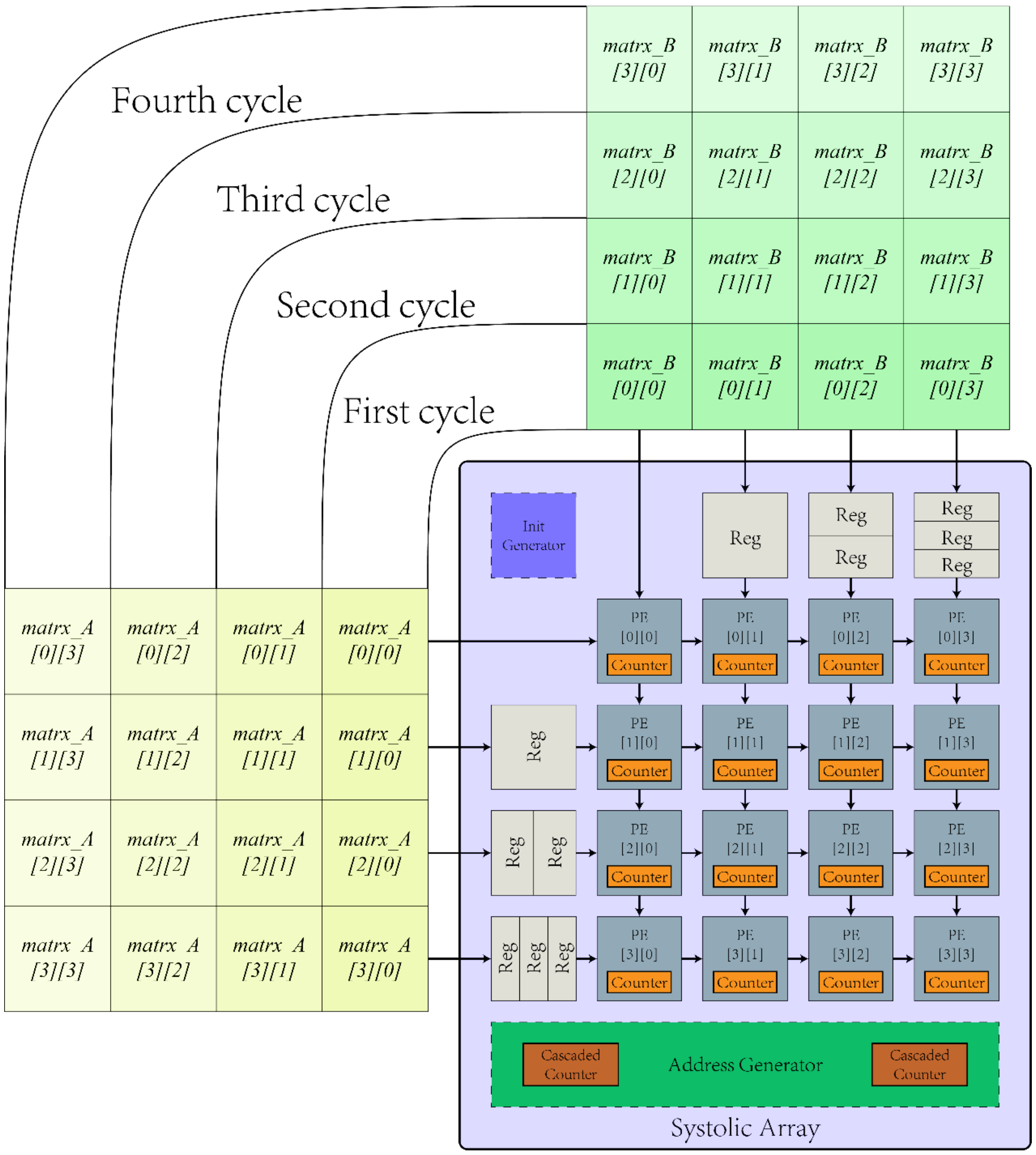
So, how exactly are $matrix\_A$ and $matrix\_B$ streamed? This depends on **the size of the systolic array** and **the size of the matrices**. Here we assume that the size of the input matrix ($matrix\_A$ and $matrix\_B$) is $M \times M$ $(\mathrm{MAT\_A\_ROWS} = \mathrm{MAT\_LENTH} \quad = \mathrm{MAT\_B\_COLS} = \mathrm{M})$ the size of the systolic array is $N \times N$ $(\mathrm{ARRAY\_SIZE} = \mathrm{N})$, and give two examples

$\boldsymbol{M = N}$ case

For simplicity, first, let us assume $M = N = 4,$ and the matrix dimensions match system size.

The matrix $matrix\_A$ will be streamed from the left of the 4x4 systolic array in a row-wise fashion. Each row of the systolic array will receive a streaming row of matrix A. Thus, we will read four rows of $matrix\_A$ in parallel with some offset. The matrix $matrix\_B$ will be streamed from the top of the 4x4 systolic array in a column-wise fashion. Each column of the systolic array will receive a streaming column of matrix $matrix\_B$. Thus, we will read four columns of B in parallel with some cycle.

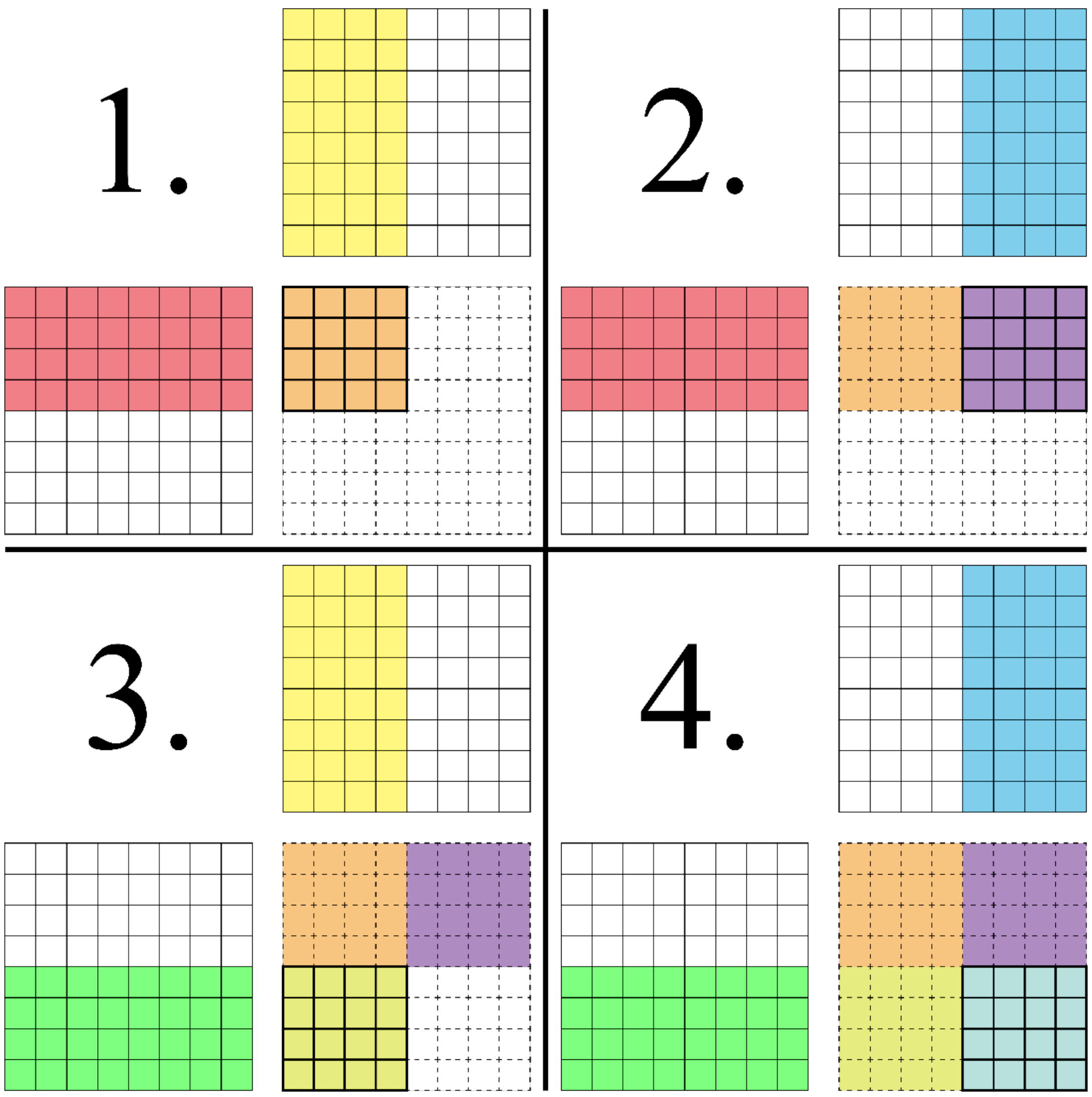
The streaming process is shown below.



$M > N$ case

Let us now consider the case where $M > N$. We will restrict our discussion to cases where M is a multiple of N.

Since $M > N$, it is clear that the entire matrix cannot merely be streamed to generate the multiplied output in one-shot. Instead, we break the input into batches of size $M \times N$ for $matrix\_A$ and $N \times M$ for $matrix\_B$ and stream those sets of rows/columns into the array. Here, computation will proceed in four phases, as shown below. So, in this case, there is no difference in the calculation; we only need to adjust the way of obtaining input data (by given the correct address).



Cascaded counters and $valid\_D$

The cascaded counters will help us choose the correct patch number for each matrix when $M > N$. Hence, the two cascaded counter instances in the systolic design will generate four counts: $slice\_cntr\_A$, $pixel\_cntr\_A$, $slice\_cntr\_B$, and $pixel\_cntr\_B$. When $M = N$, the slice counter for the A and B will be held constant at 0 because we do not need to calculate the result matrix in blocks. You should combine these counts into two addresses. Read the `send_input` process in the given testbench code to understand the addressing logic.

Your systolic design should also output a STD_LOGIC_VECTOR $valid\_D$ which could transfer to the 2D array of n-bit signal, signaling the testbench to read D for partial matrix products and store them in the partial array. When we use a systolic array for continuous operations, sometimes we should record multiple outputs in the same cycle, and they belong to different matrix multiplication operations. At this time, $valid\_D$ is not only used to distinguish whether the output is legal, but also should be used to distinguish whether the output belongs to the same matrix multiplication. Each n bit of the $valid\_D$ signal corresponds to the matrix product being ready at the corresponding PE and belongs to witch multiplication

($valid\_D[0 \times N + 0 :+ n] \rightarrow PE[0][0]$, $valid\_D[0 \times N + 2 :+ n] \rightarrow PE[0][2]$). The specific element of D is not valid when the corresponding position of $valid\_D$ is 0. Furthermore, here, we agree that after the reset state ends (rst from '1' to '0') and starts to work, the $valid\_D$ of the corresponding position should be 1 when the D at each position can be output for the first time. Moreover, it should be 2 when the D at each position can be output for the second time and so on.

Another point is that systolic does not need to know which matrix it is calculating after reset, nor does it need to do any special processing. Of course, the testbench needs to consider this.

PE

Your PE should accept two inputs: in_a and in_b at every clock cycle. It should perform multiply-accumulate operations on these inputs. Additionally, your PE should be capable of streaming out the registered versions of in_a and in_b. Your PE should reset all its output (out_sum, out_a, out_b) on the assertion of the rst signal. Whereas on the assertion of the $init$ signal, only out_sum should start accumulating again. Note: You should not make out_sum zero on init, but instead, you should start a new accumulation. For this lab, we will not shift out the final accumulated results, and they will stay-in-place within the out_sum register. It would be best if you generated a $valid\_sum$ from each PE when the result of the dot product is ready for inspection by the external testbench. Please note that the output port $valid\_sum$ of the PE is different from the output port $valid\_D$ of the top-level function systolic. It only needs to care about the situation inside the PE.

Counter

The cascaded counters indicate which entry of the row/column is being fed into the systolic array. A separate counter is provided for $matrix\_A$ and $matrix\_B$ matrices. For the $matrix\_A$ matrix shifted from the left, the pixel counter will loop through the different entries in a row. Different rows of the systolic array will be provided different rows of $matrix\_A$ with a cycle offset per row. For $M > N$, the slice counter of the $matrix\_A$ matrix will increment after the pixel counter has counted through its range. The slice counter indicates which of the (M/N) $N \times M$ slice of matrix $matrix\_A$ that will be fed into the systolic core. The roles are reversed for the counter for $matrix\_B$.

The cascaded counter accepts a $enable\_row\_count$ control signal to allow the cascaded counter to increment. This is necessary to realize the $(M/N)^2$ partial product calculations needed on the systolic array.

Testbench Operation

We will provide a testbench with some functions for you to debug your code. The testbench now has these features:

- A series of custom data types are defined. They can provide you with some references.

- Generate a clock signal. This function is implemented by process **`clock_generate`**.
- Generate a reset signal. This function is implemented by process **`rst_generate`**.
- Read data from the file and store them in the custom 3D memory $matrix\_A$ and $matrix\_B$. This function is implemented by process **`read_input`**. Note that you need to use signal $matrix\_in\_done\_sig$ to control the behavior of this process
- Retrieve data from custom 3D memory $matrix\_A$ and $matrix\_B$, and send them to the input port A and B of the systolic array according to the data address from output port $read\_address\_A$ and $read\_address\_B$ of the systolic array. This function is implemented by process **`send_input`**.
- Write results stored in custom 3D memory $matrix\_D$ to file. It would be best if you used signal $matrix\_out\_done\_sig$ to control the behavior of this process. Note that before you pull up the $matrix\_out\_done\_sig$, you should accept the result returned by the systolic array and fill the $matrix\_D$ in the correct way. This function is implemented by process **`save_result`**.
- Compare the results generated by the systolic array $(matrix\_D)$ with the reference results you generated in the test-bench $(reference\_D)$, and save the comparison results in a file. It would be best if you used signal $matrix\_out\_done\_sig$ to control the behavior of this process. Note that before you pull up the $matrix\_out\_done\_sig$, the $reference\_D$ should store the correct result. This function is implemented by process **`check_result`**.

You also need to complete the following tasks for the testbench:

- Calculate the reference result at the right time and put it in the right place.
- Receive the data output by the systolic array and place it in the correct place on each rising edge of the clock.

Task

1. Read the code given to understand the behavior logic that the top-level function should implement. If there is any part in the description of this material that conflicts with the given code, the code shall prevail and report to the TA on BB.
2. Implement the systolic algorithm and complete the VHDL codes. Note that your code should be able to adapt to different combinations of configurable parameters $\mathrm{MAT\_A\_ROWS}$, $\mathrm{MAT\_LENTH}$, $\mathrm{MAT\_B\_COLS}$, and $\mathrm{ARRAY\_SIZE}$. Only need to deal with the case where $\mathrm{MAT\_A\_ROWS}$ and $\mathrm{MAT\_B\_COLS}$ are integral multiples of $\mathrm{ARRAY\_SIZE}$. The more situations you can handle, the higher your score in this task.
3. Complete the testbench to process your systolic array correctly and output the correct result to the file. Note that your code should be able to adapt to different combinations of configurable parameters $\mathrm{MAT\_A\_ROWS}$, $\mathrm{MAT\_LENTH}$, $\mathrm{MAT\_B\_COLS}$, and

ARRAY_SIZE. Only need to deal with the case where MAT_A_ROWS and MAT_B_COLS are integral multiples of ARRAY_SIZE. The more situations you can handle, the higher your score in this task.

4. Please be aware that you can only add code where there are comments, but cannot delete or modify the original code when completing the above tasks. Any deletion or modification of the code will cause unexpected errors in the automatic scoring.

5. Your code should be synthesizable. Non-synthesizable statements are only allowed to appear in the testbench. Otherwise, points will be deducted.

Code format

1. Keywords (except data type) are **all lowercase**.
2. The library names and data types are **all capitalized**.
3. Naming must be **meaningful** (except for the prescribed name). There are no naming restrictions, and you can reasonably use uppercase letters and underscores to make your name more comfortable to read and understand.
4. Use **four Spaces** or **a Tab** to indent your code. Proper indentation makes your code easier to read and understand.

Attention

- **In this lab, the TA CAN do these:**
    - Explain experimental materials and experimental requirements.
    - Explain the scoring criteria and scoring methods (not including specific information about test cases).
    - Explain the algorithm.
    - Explain the provided code file and related information.
    - Explain the logs and error messages on the auto-scoring platform
- **In this lab, the TA CAN NOT do these:**
    - Operate your computer.
    - See your code.
    - See the error message on your computer.

Submit

1. Rename the testbench file to test_systolic.vhd.template and change the default value of the first eight generic ports to the form of PORT_NAME_VALUE, like this:

   MAT_A_ROWS : POSITIVE := MAT_A_ROWS_VALUE;

   Make sure that these default values only appear once in the testbench file as it will be simple replaced during the automatic scoring.

2. Compress the whole folder and **name the compressed package as follow the format**:

   Lab4_[YourNameInEnglish]_[YourStudentNumber].zip

3. You should submit:

   - Source codes
     - All of the source codes
   - Test bench codes
     - All of the test bench codes

4. **File Organization Schema in Package**:

```
Lab4_[YourName]_[YourStudentNumber].zip
      ├──── sim
      |     └──── test_systolic.vhd.template
      └──── src
            ├──── counter.vhd
            ├──── pe.vhd
            ├──── pipe.vhd
            ├──── systolic.vhd
            └──── util_package.vhd
```

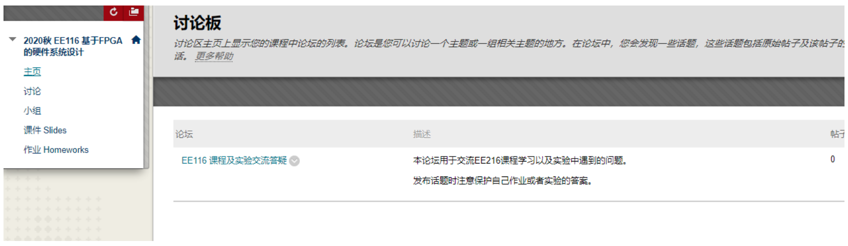
5. Submit the compressed package to the automatic scoring platform and wait for the scoring result. Note that it takes three and a half minutes to score; please be patient.

Deadline of Submit: 2020-11-11 15:00

- Submit on time, get all scores.
- Submission time does not exceed 24 hours of the deadline, get half of the score.
- Submitted more than 24 hours from the deadline, get no score.

Any Question?

Any questions on course or labs can be proposed in the Discussing Forum on BB.



2020秋 EE116 基于FPGA的硬件系统设计

- 主页
- 讨论
- 小组
- 课件 Slides
- 作业 Homeworks

讨论板

讨论区主页上显示您的课程中论坛的列表。论坛是您可以讨论一个主题或一组相关主题的地方。在论坛中，您会发现一些话题，这些话题包括原始帖子及该帖子的话。 更多帮助

论坛	描述	帖
EE116 课程及实验交流答疑	本论坛用于交流EE216课程学习以及实验中遇到的问题。 发布话题时注意保护自己作业或者实验的答案。	0

We recommend you subscribe to the forum to receive the newest topics on time.