Digital Systems Design

Term Project: CNN Accelerator

Electrical and Computer Engineering

Seoul National University

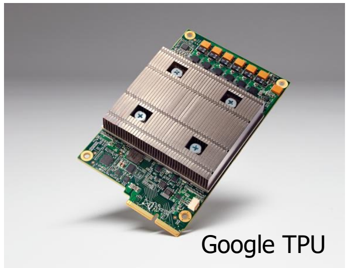
Table of Contents

- Introduction
- Background
    - What is Convolutional Neural Network?
- Workload Specification
    - CNN Model Architecture
- Module-wise description
    - Interface Protocols
    - VDMA Engine
    - Accelerator Modules
- System Overview
- Submission Guideline
- Miscellaneous

Introduction

Accelerators

- How to design accelerators?
  - 1. Select the target workload
  - 2. Analyze the bottleneck
  - 3. Optimize the bottleneck in a hardware-friendly way



Google TPU

NVIDIA GPU
(Titan V)

FPGA (Xilinx Virtex-7)

- Our Workload?
  - **Convolutional Neural Network (CNN) Inference**

Project Objectives

- Implement your own **CNN accelerator** on the **FPGA** board
  - Only for **inference**; not for training

- Everything will be given…
  - Pretrained CNN Model
  - Dataset
  - End-to-end Framework
    - The interface between the host CPU and the FPGA board
    - Xilinx IPs (Interconnects, DMA Engine, MIG, etc)
    - Simple testbench
      - □ Note that the testbench is imperfect (Does not strictly follow the actual control flow)
      - □ **You should revise testbench for precise evaluation**

- All you need to do is…
  - Implementing CNN compute primitives into hardware modules

Project Directory

- Directory Structure
  - build: Vivado project directory
  - src : Verilog source codes
  - docs : Documentation of the project
  - test : Python codes that run on the host CPU

- Refer to the ***README.md*** file in your GitHub repository for more information

Project Roadmap

- 1st Week (Nov. 20th ~ Nov. 26th)
  - Understand the workload/end-to-end framework
  - Roughly design your accelerator
  - Share your ideas with teammates

- 2nd Week (Nov. 27th ~ Dec. 3rd)
  - Implement your accelerator, focusing on **the functionality**
  - Debug your implementation

- 3rd Week (Dec. 4th ~ Dec. 10th)
  - Optimize your accelerator, regarding **the performance**
  - Debug your optimizations

- 4th Week (Dec. 11th ~ Dec. 17th)
  - Find the optimal design point

Remarks

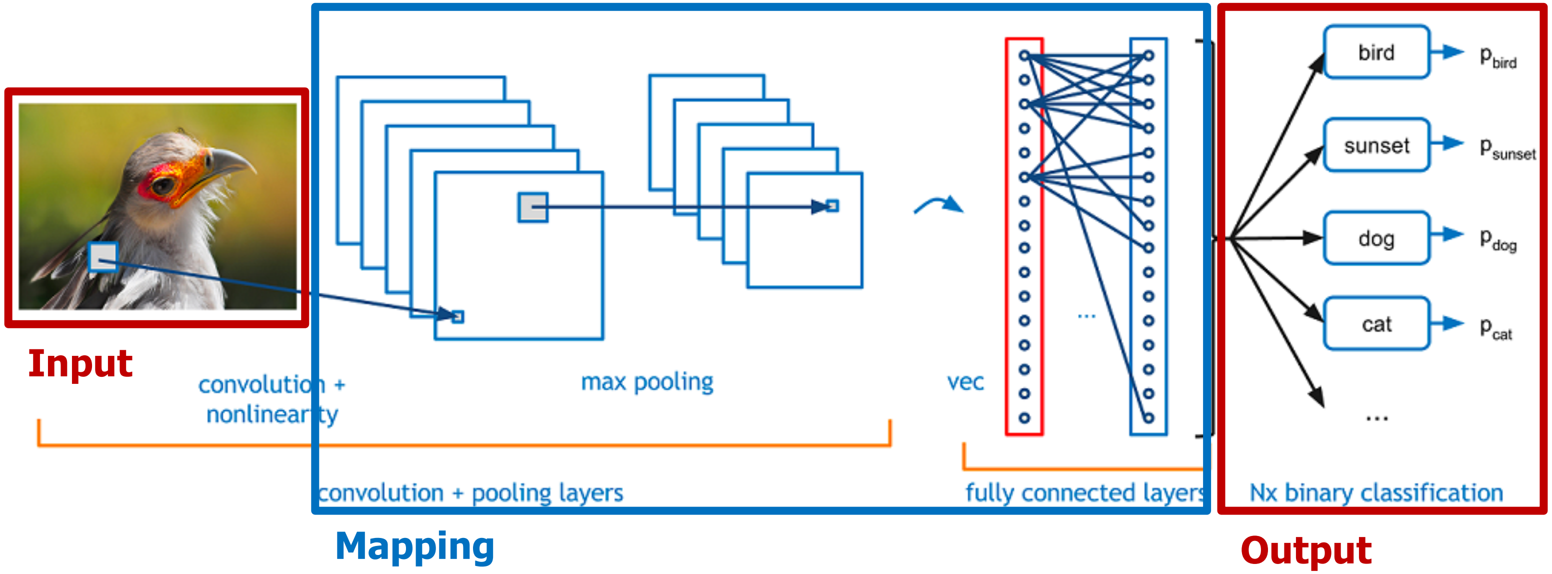
- This is not a Deep Learning course
  - We don’t expect you to know every little detail of deep learning
  - Understanding ***the underlying algorithm of CNN*** is enough

- **DO NOT COPY**
  - **You will fail the course if you cheat**
  - There are too many valid HDL codes for a single hardware
    - Even if you share the high-level algorithm, the implementation will be very different
  - We will use **a strict plagiarism detector**
    - We already have all the source codes from the last two years

Background

Neural Network

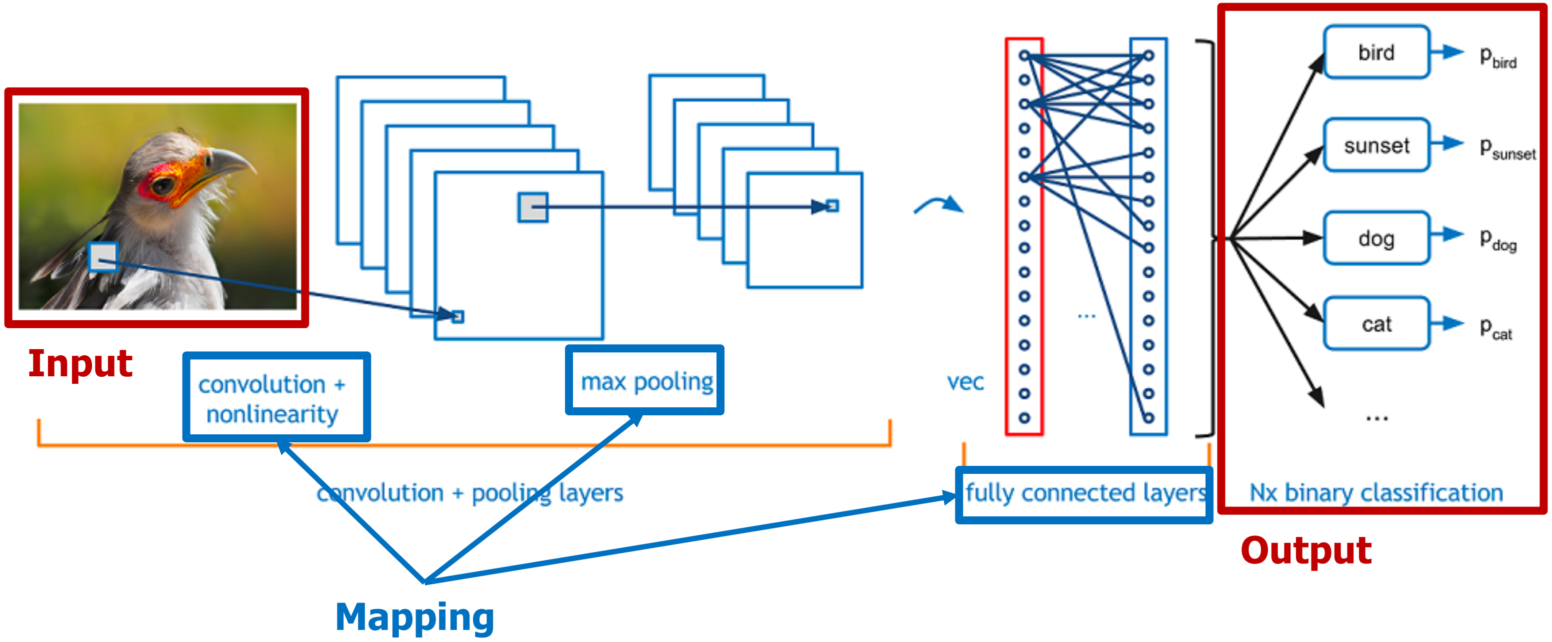
- Neural Network is a **Function Approximator**
  - ***A mapping from a given input to desired output can be learned***
  - e.g. Image Classification
    - Input: Image, Output: Classification Results



source: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

Convolutional Neural Network (CNN)

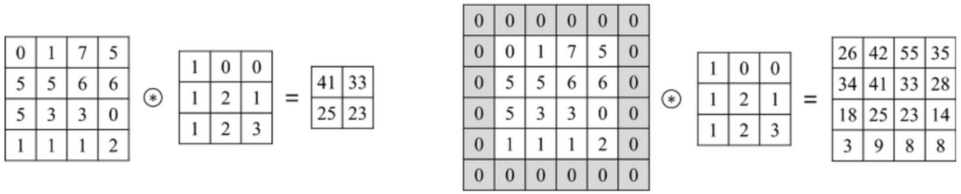
- CNN Model Architecture
  - Typically consists of three layers
    - Convolution, Pool, Fully-connected
  - **Each layer is essentially a different compute primitive**



source: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

Convolution Layer

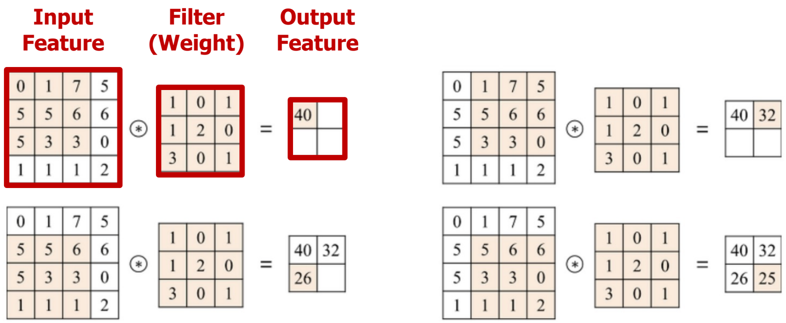
- Role: Extract the features from input data
- Parameter
  - Filter: weight matrix to multiply with the feature matrix
- Hyperparameter
  - Stride: How many elements should the filter jump?
  - Padding: Enlarge the size of input features
    - To make the size of input and output similar or the same
- **Compute Primitive**
  - **Convolution**
  - **ReLU: filter out negative values**



$$\begin{bmatrix}0&1&7&5\\5&5&6&6\\5&3&3&0\\1&1&1&2\end{bmatrix} \circledast \begin{bmatrix}1&0&0\\1&2&1\\1&2&3\end{bmatrix} = \begin{bmatrix}41&33\\25&23\end{bmatrix}$$

$$\begin{bmatrix}0&0&0&0&0&0\\0&0&1&7&5&0\\0&5&5&6&6&0\\0&5&3&3&0&0\\0&1&1&1&2&0\\0&0&0&0&0&0\end{bmatrix} \circledast \begin{bmatrix}1&0&0\\1&2&1\\1&2&3\end{bmatrix} = \begin{bmatrix}26&42&55&35\\34&41&33&28\\18&25&23&14\\3&9&8&8\end{bmatrix}$$

The effect of padding
(left: no padding, right: zero padding)



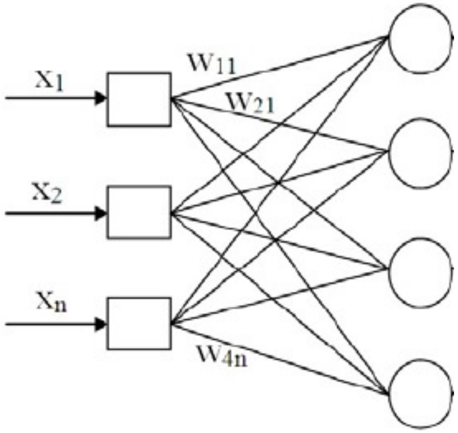
$$\begin{bmatrix}0&1&7&5\\5&5&6&6\\5&3&3&0\\1&1&1&2\end{bmatrix} \circledast \begin{bmatrix}1&0&1\\1&2&0\\3&0&1\end{bmatrix} = \begin{bmatrix}40&\\&\end{bmatrix} \qquad \begin{bmatrix}0&1&7&5\\5&5&6&6\\5&3&3&0\\1&1&1&2\end{bmatrix} \circledast \begin{bmatrix}1&0&1\\1&2&0\\3&0&1\end{bmatrix} = \begin{bmatrix}40&32\\&\end{bmatrix}$$

$$\begin{bmatrix}0&1&7&5\\5&5&6&6\\5&3&3&0\\1&1&1&2\end{bmatrix} \circledast \begin{bmatrix}1&0&1\\1&2&0\\3&0&1\end{bmatrix} = \begin{bmatrix}40&32\\26&\end{bmatrix} \qquad \begin{bmatrix}0&1&7&5\\5&5&6&6\\5&3&3&0\\1&1&1&2\end{bmatrix} \circledast \begin{bmatrix}1&0&1\\1&2&0\\3&0&1\end{bmatrix} = \begin{bmatrix}40&32\\26&25\end{bmatrix}$$

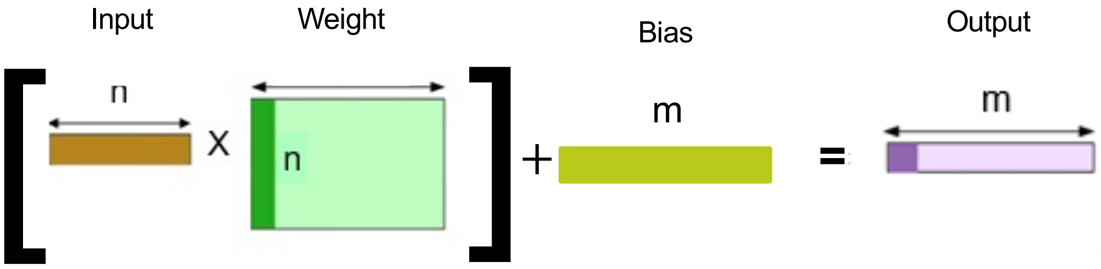
Example of convolution computation
(stride_size=1, padding=0)

Fully-Connected (FC) Layer

- Role: Classify the given input
- Parameter
  - Weight, Bias
- **Compute Primitive**
  - **Matrix-vector Multiplication**
  - **vector-vector Addition**
  - **ReLU**



Structure of FC layer [1]

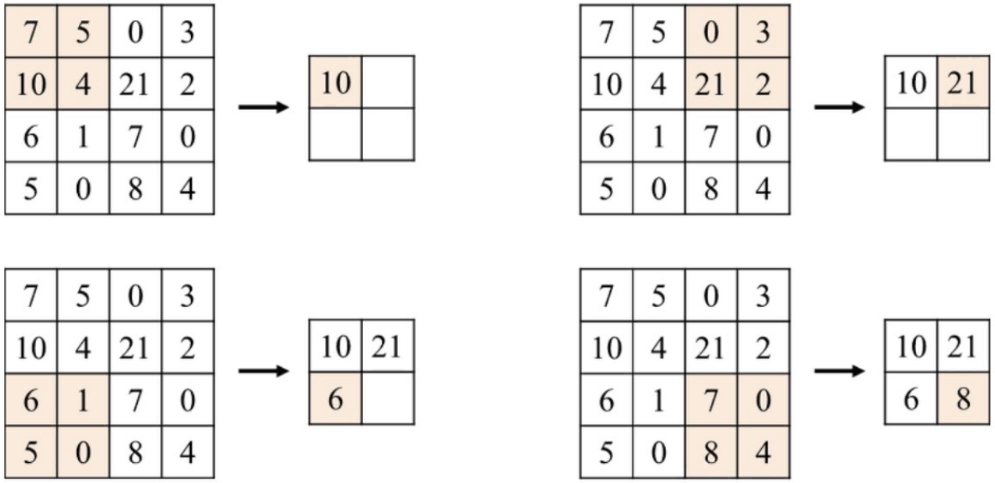


Operation of FC layer[2]

[1] http://needjarvis.tistory.com/182
[2] https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

Pooling Layer

- Role: Prevent overfitting
- Hyperparameter
  - Window: Small portion of input features from which we select the output element
  - Stride: How many elements should the window jump?
  - Policy: How do we select the output element?
    - Max, Min, Average, etc
    - We will be using ***Max pooling***
- **Compute Primitive**
  - Max



7	5	0	3
10	4	21	2
6	1	7	0
5	0	8	4

→

10	

7	5	0	3
10	4	21	2
6	1	7	0
5	0	8	4

→

10	21

7	5	0	3
10	4	21	2
6	1	7	0
5	0	8	4

→

10	21
6	

7	5	0	3
10	4	21	2
6	1	7	0
5	0	8	4

→

10	21
6	8

Example of pooling computation
(window_size=2x2, stride_size=2)

Appendix

- What is Deep Learning?
  - https://www.nature.com/articles/nature14539

- Convolutional Layer
  - http://cs231n.github.io/convolutional-networks/#conv
- Fully-connected Layer
  - http://cs231n.github.io/convolutional-networks/#fc
- Pooling Layer
  - http://cs231n.github.io/convolutional-networks/#pool

Workload Specification

Training Setup

- Dataset
  - CIFAR-10

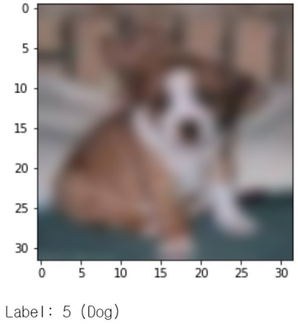
- CNN Model Architecture
  - VGGNet variant with…
    - 6 Conv Layers
    - 4 Pooling Layers
    - 3 FC Layers

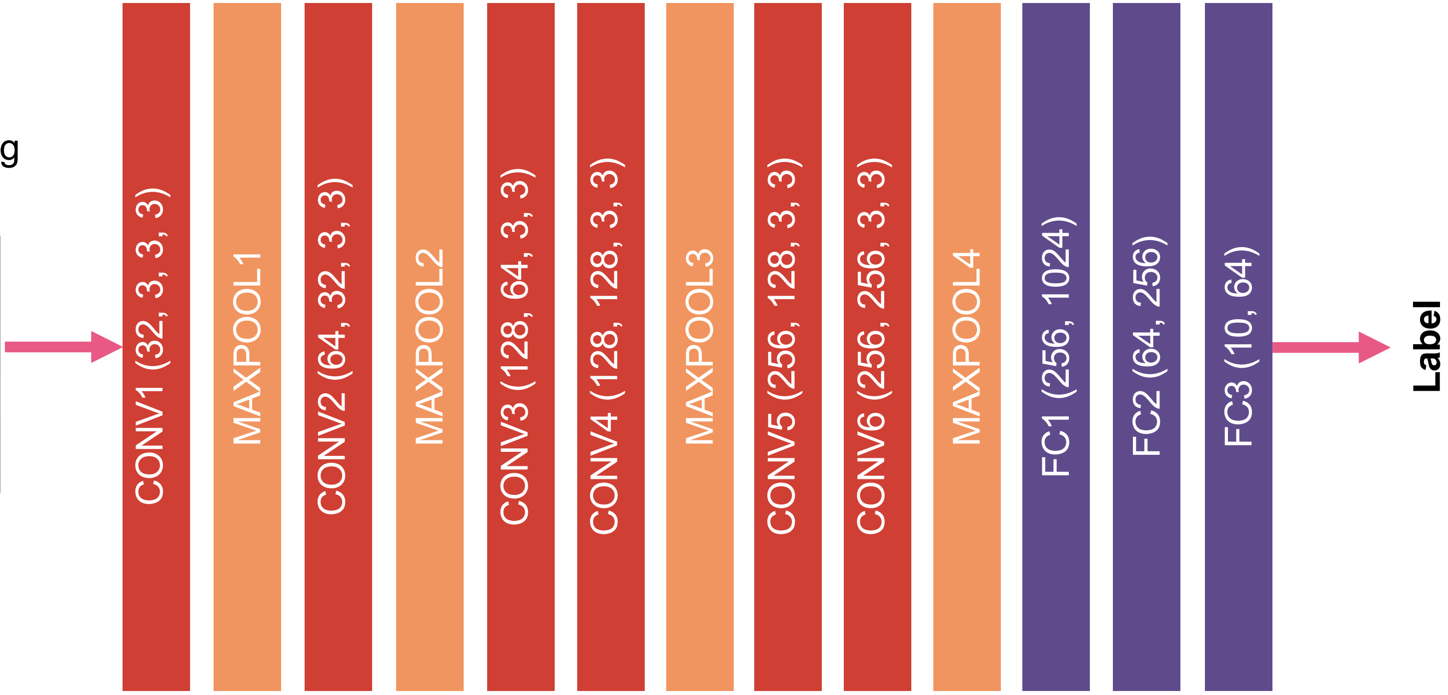
- Trained with PyTorch Framework
  - Post-training quantization applied

- We **already prepared** your **CNN Model with quantized parameters**

CNN Model Architecture

Input size per img
3x32x32





CONV1 (32, 3, 3, 3)

MAXPOOL1

CONV2 (64, 32, 3, 3)

MAXPOOL2

CONV3 (128, 64, 3, 3)

CONV4 (128, 128, 3, 3)

MAXPOOL3

CONV5 (256, 128, 3, 3)

CONV6 (256, 256, 3, 3)

MAXPOOL4

FC1 (256, 1024)

FC2 (64, 256)

FC3 (10, 64)

Label

(Out-ch, In-ch, Row, Col)

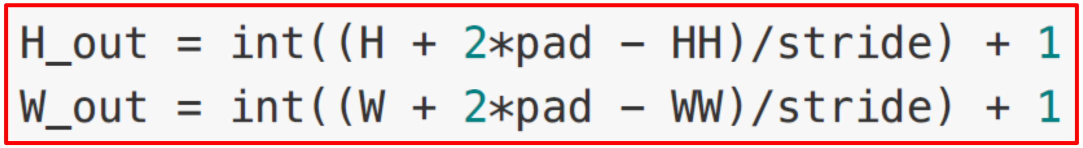
(Out, In)

Input Features (Hyperparameters)

- Input Features: (N, C, H, W)
  - N : Number of images
  - C : Number of channels of images
  - H : Height of the image
  - W: Width of the image
  - Example: Images (1000, 3, 32, 32)
    - There are a total of 1000 images
    - Each image consists of 3 channels (R, G, B)
    - Each channel of the image is represented with a 32x32 matrix

Convolution Layer (Hyperparameters)

- Filters: (F, C, HH, WW)
  - F : Number of “output” channels
  - C : Number of “input” channels
  - HH : Height of the filter
  - WW : Width of the filter
  - Example
    - Filter: (32, 3, 3, 3)
    - The filter consists of 3 input channels and 32 output channels
    - Each filter is a 3x3 matrix
- Bias: 1D array (F)
- Zero-Padding: 1, Stride: 1
  - **The size of the input does not change**, while # of input data changes!



```
H_out = int((H + 2*pad - HH)/stride) + 1
W_out = int((W + 2*pad - WW)/stride) + 1
```

Pooling Layer (Hyperparameters)

- Pooling Policy: Max Pooling

- Window Size: 2, Stride: 2
  - 2x2 Pooling Window
  - The size of row/col is reduced to half

Fully-connected Layer (Hyperparameters)

- Weights: (out_size, in_size)

- Remarks
  - The input features to the FC Layers should be flattened to 1D vector

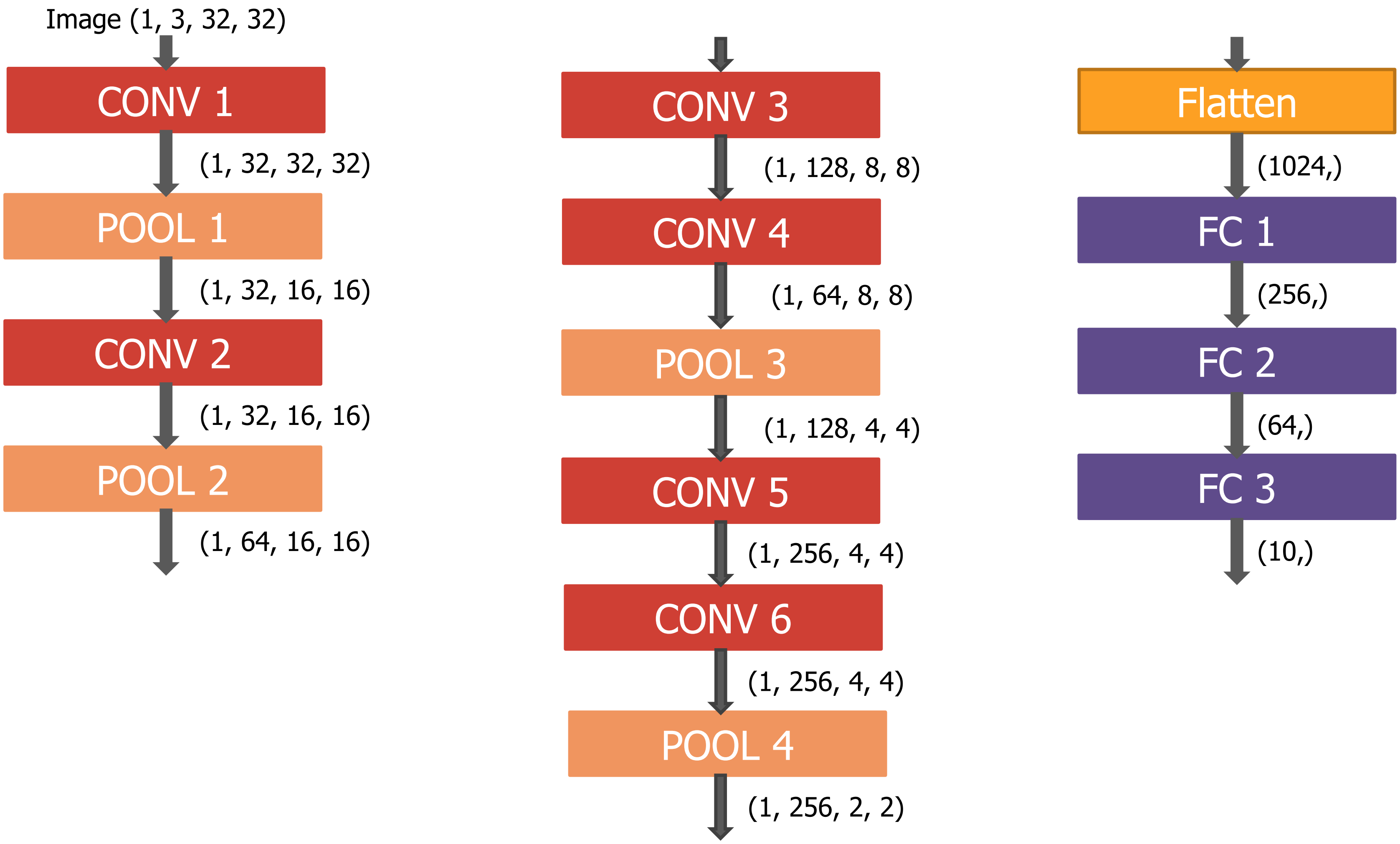
Data Layout

- You should know the order in which data is stored in the FPGA Memory

- Data Layout
  - e.g., NCHW Layout: W is the inner-most dimension

    => data in the W dimension is stored in consecutive memory addresses
  - Input Features: (N, C, H, W)
  - Convolution Filters: (F, C, HH, WW)
  - Convolution Bias: 1D
  - FC Weights: (out, in)
  - FC Bias: 1D

Overall Model



Module-wise Description

System Overview

- Hardware Modules
  - UART Module, Decoder
  - VDMA Engine
  - Conv Module, FC Module, Pooling Module
  - MIG

- We need a standard way to communicate between these modules
  - **Advanced Microcontroller Bus Architecture (AMBA) Protocol**

AMBA Protocols

- AMBA Protocols include…
  - AMBA AXI
  - AMBA AXI-Stream
  - AMBA AXI-Lite
  - AMBA APB
  - …

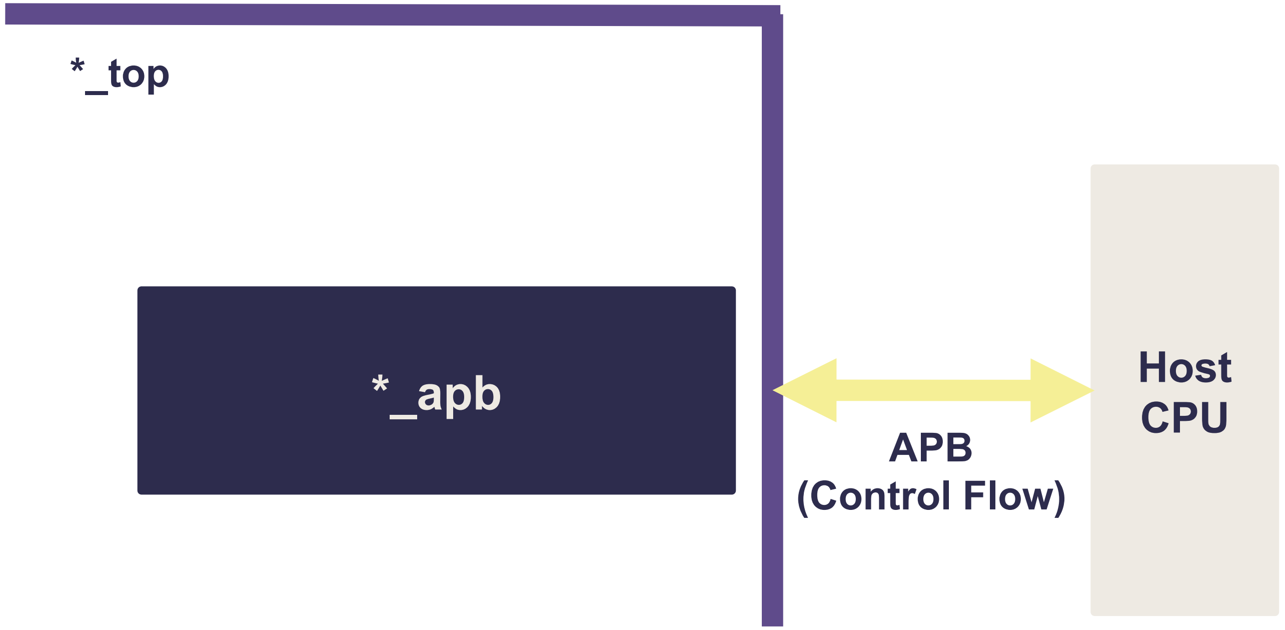
- Each protocol has different usage
  - A simple protocol to move small data

    vs. A complex protocol to move a large amount of data

- We will be using…
  - ***APB***
  - ***AXI***
  - ***AXI-Lite***
  - ***AXI-Stream***

Advanced Peripheral Bus (APB) Protocol

- APB protocol is for the Control Flow
  - *: conv, fc, pool



Advanced Peripheral Bus (APB) Protocol

- Host CPU controls the FPGA modules with APB
  - **Control registers** of Conv, FC, Pooling Modules **are memory-mapped**
  - **Performance counters** of Conv, FC, Pooling Modules **are memory-mapped**
  - You can program the control registers by read/write operations through the APB protocol

- Also useful for Debugging!
  - Add memory-mapped debug registers as you want
  - You can check the data in FPGA debug registers from the host CPU

- Reference
  - https://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/readings/ARM_AMBA3_APB.pdf

Advanced Peripheral Bus (APB) Protocol

- APB Interface (Ports)

APB input/output			
Input		**Output**	
Name	**Size**	**Name**	**Size**
PCLK	1-bit	PRDATA	32-bit
PRESETB	1-bit		
PADDR	32-bit		
PSEL	1-bit		
PENABLE	1-bit		
PWRITE	1-bit		
PWDATA	32-bit		

Advanced Extensible Interface (AXI) Protocols

- AXI
  - Basic version
  - **Memory-mapped**
    - Module with AXI interface ***can be accessed with the address***
  - Uses 5 channels for data transaction between master and slave
    - Write addr/control/data/resp, Read addr/data

- AXI-Lite
  - A light version of AXI that precludes some functionality (e.g. burst mode)
  - **Memory-mapped**

- **AXI-Stream**
  - Used for sending/receiving large data streams
  - **Not Memory-mapped**
    - **There is no explicit address**
  - Uses only one channel for data transaction between master and slave
    - Write data

AXI-Stream Protocol

- You should implement the AXI-Stream interface for your Conv/FC/Pooling Modules
  - Each acceleration module communicates with the VDMA engine to send/receive data to/from the FPGA DRAM
  - Master sends Data, while the slave receives data
    - Acceleration module (M) -> VDMA Engine (S)
    - VDMA Engine (S) -> Acceleration module (M)
  - Signals
    - *ACLK* and *ARESETn* are shared between master and slave
    - *TKEEP* and *TUSER* for the master should be set to {4{1'b1}}, 1'b0

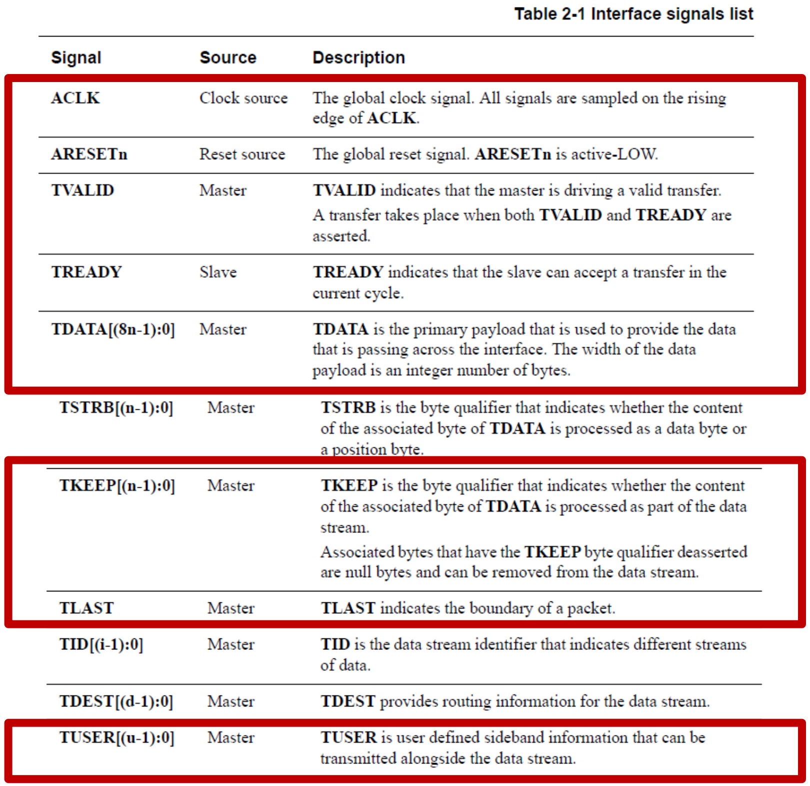


Table 2-1 Interface signals list

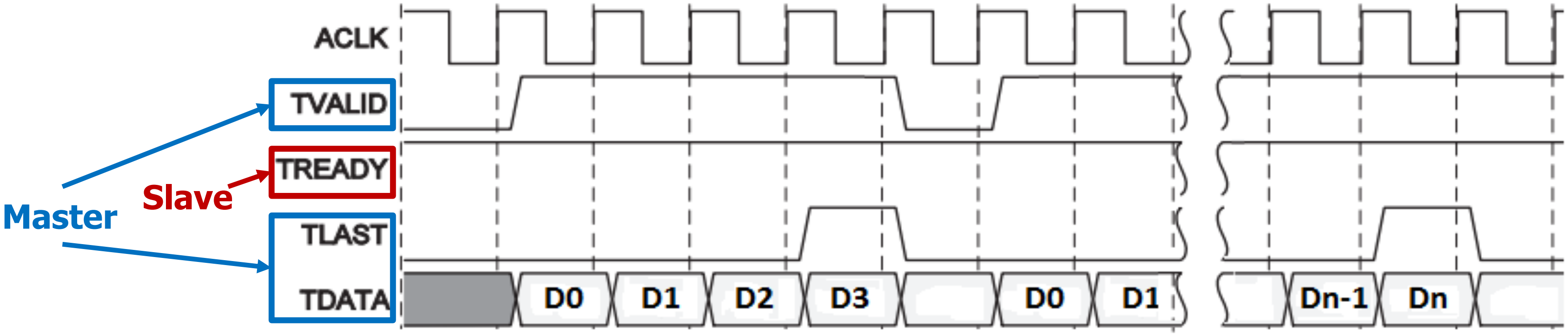
Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of **ACLK**.
ARESETn	Reset source	The global reset signal. **ARESETn** is active-LOW.
TVALID	Master	**TVALID** indicates that the master is driving a valid transfer. A transfer takes place when both **TVALID** and **TREADY** are asserted.
TREADY	Slave	**TREADY** indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	**TDATA** is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TSTRB[(n-1):0]	Master	**TSTRB** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as a data byte or a position byte.
TKEEP[(n-1):0]	Master	**TKEEP** is the byte qualifier that indicates whether the content of the associated byte of **TDATA** is processed as part of the data stream. Associated bytes that have the **TKEEP** byte qualifier deasserted are null bytes and can be removed from the data stream.
TLAST	Master	**TLAST** indicates the boundary of a packet.
TID[(i-1):0]	Master	**TID** is the data stream identifier that indicates different streams of data.
TDEST[(d-1):0]	Master	**TDEST** provides routing information for the data stream.
TUSER[(u-1):0]	Master	**TUSER** is user defined sideband information that can be transmitted alongside the data stream.

AXI-Stream Protocol

- AXI-Stream Interface (Ports)

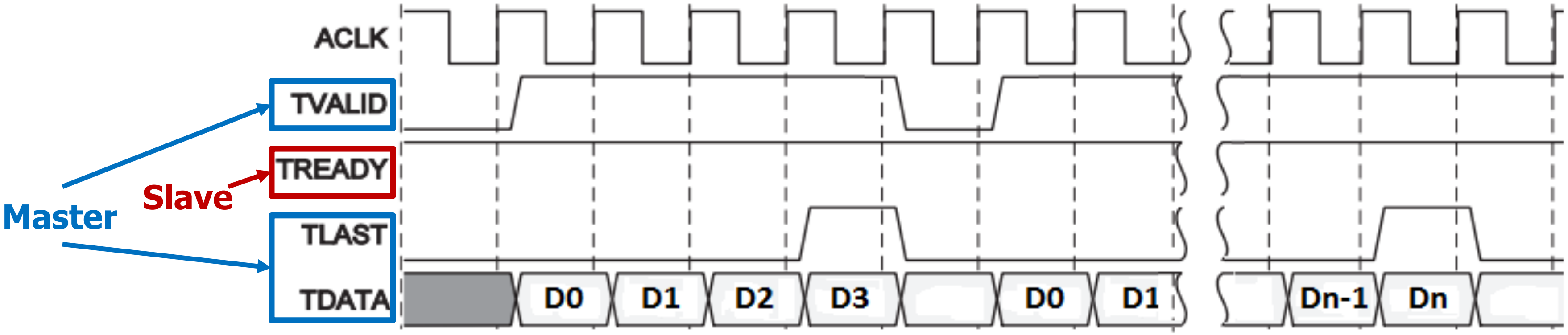
AXI-Stream input/output			
Input		**Output**	
Name	**Size**	**Name**	**Size**
M_AXIS_TREADY	1-bit	S_AXIS_TREADY	1-bit
S_AXIS_TDATA	32-bit	M_AXIS_TDATA	32-bit
S_AXIS_TVALID	1-bit	M_AXIS_TVALID	1-bit
S_AXIS_TLAST	1-bit	M_AXIS_TLAST	1-bit
S_AXIS_TKEEP	4-bit	M_AXIS_TKEEP	4-bit
S_AXIS_TUSER	1-bit	M_AXIS_TUSER	1-bit
AXIS_ACLK	1-bit		
AXIS_ARESETN	1-bit		

AXI-Stream Protocol



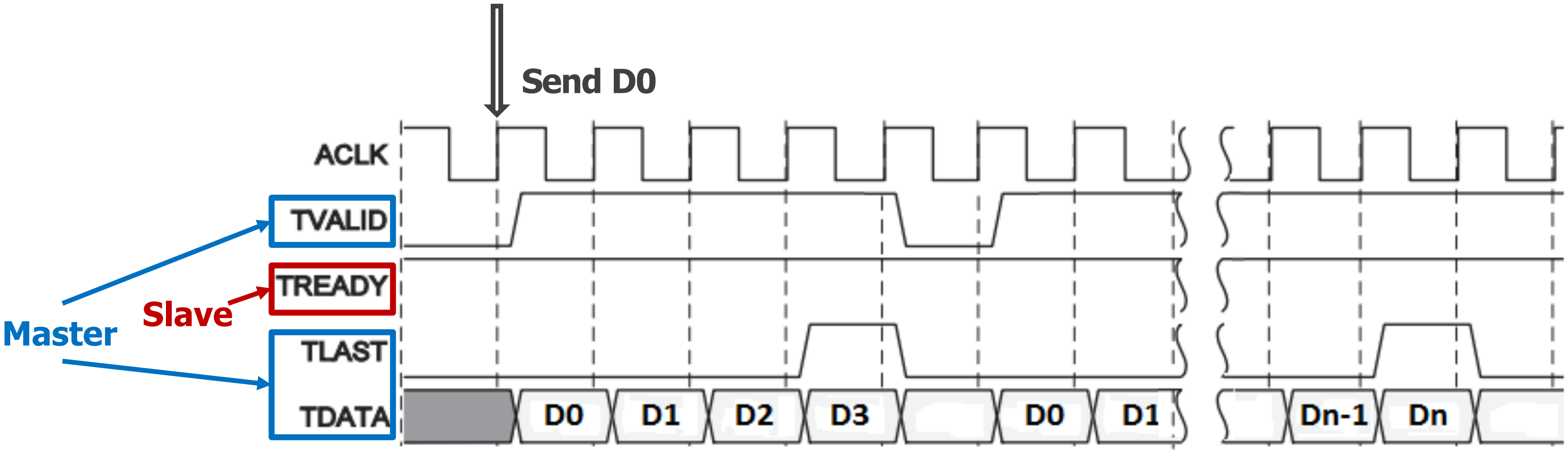
- When **the Slave** is ready to receive data,
  set **TREADY** high

AXI-Stream Protocol



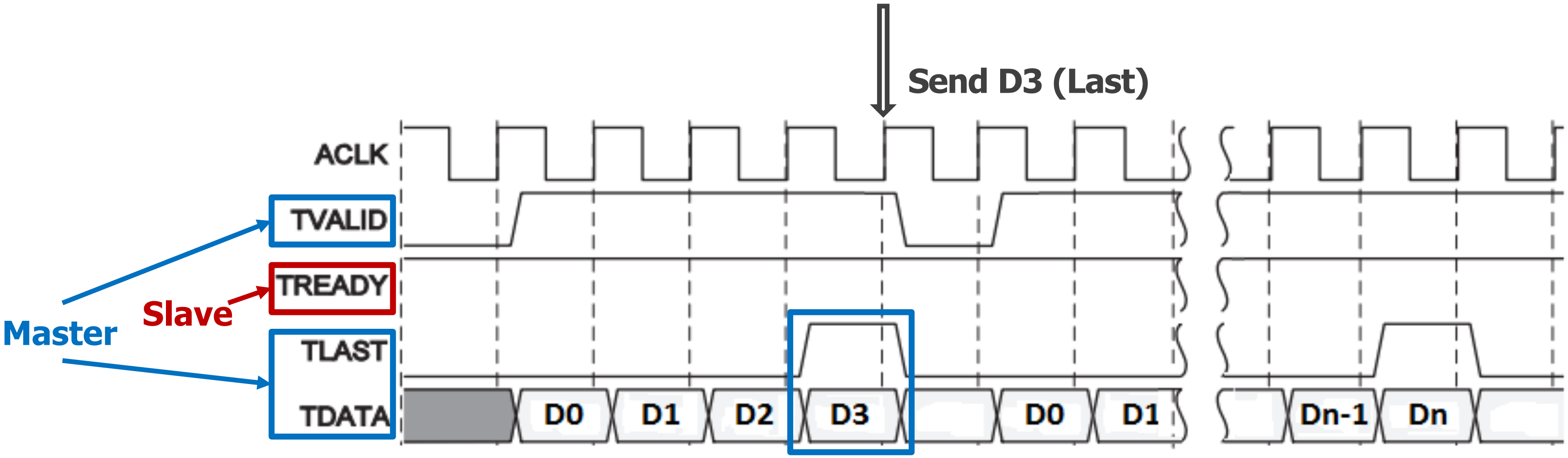
- When **the Master** is ready to send data, set **TVALID** and **TDATA**

AXI-Stream Protocol



- When **Slave TREADY** and **Master TVALID** are both **HIGH**, data transfer begins

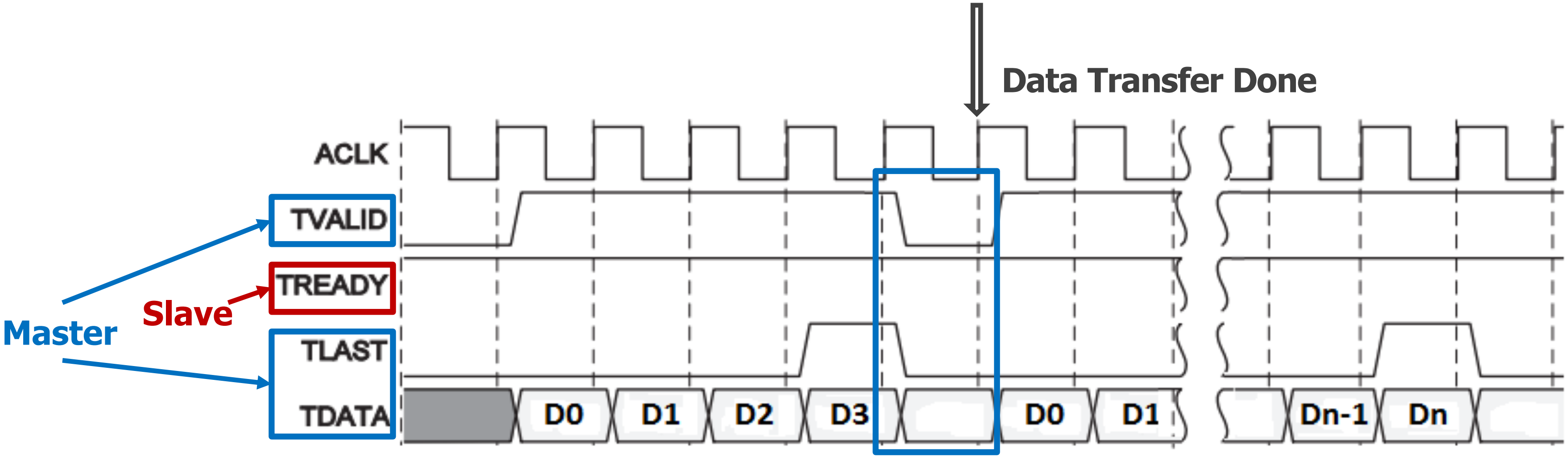
AXI-Stream Protocol



- When the last data is transferred,

  the Master sets **TLAST**

AXI-Stream Protocol



- When the data transfer ends, **the Master *resets* TLAST and TVALID**

AXI-Stream Protocol

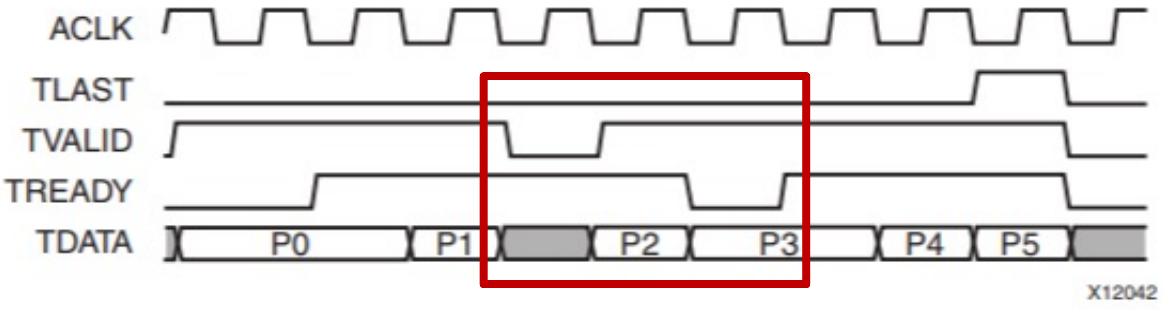


Figure 4-3: **AXI4-Stream Waveform**

- If **TVALID == 0** && **TREADY == 1**
  - Master does not have any data to send
- If **TAVLID == 1** && **TREADY == 0**
  - The Master has data to send, but the slave is not ready

    => Data movement stalls, wait until the slave becomes ready

AXI-Stream Protocol

- Exercise
  - The AXI-Stream interface is fundamentally similar to the AXI-Lite interface
  - You can implement a toy example with an AXI-Stream interface (Lab 9)

- Reference (**Highly Recommended**)
  - https://developer.arm.com/documentation/ihi0051/a/

VDMA Engine

- Direct Memory Access (DMA)
  - A mechanism that allows certain hardware subsystems to access main system memory without CPU intervention
  - The opposite of Programmed IO (PIO)

- AXI VDMA (Video Direct Memory Access) Engine
  - Provides a high bandwidth DMA for video applications
  - Provides high-performance direct memory access between FPGA memory and **AXI4-Stream** type target peripherals

VDMA Engine

- How to use VDMA Engine
  - **The host CPU must program the control registers in the VDMA Engine**
    - Recap) ***The control registers are memory-mapped***

1. *VDMACR (control)* register: set the control
   (i.e., MM2S or S2MM, the number of frames)
2. *START_ADDRESS* register: set the base memory address to send/receive data
3. *FRMDLY_STRIDE* register: set the stride
   (if you don’t need this feature, set it as the same as the HSIZE)
4. *HSIZE* register: set the HSIZE (line size in Bytes)
5. *VSIZE* register: set the VSIZE (# of lines)

- Once all these control registers are set, VDMA will begin the data transfer between **FPGA memory** and **the Acceleration Module**

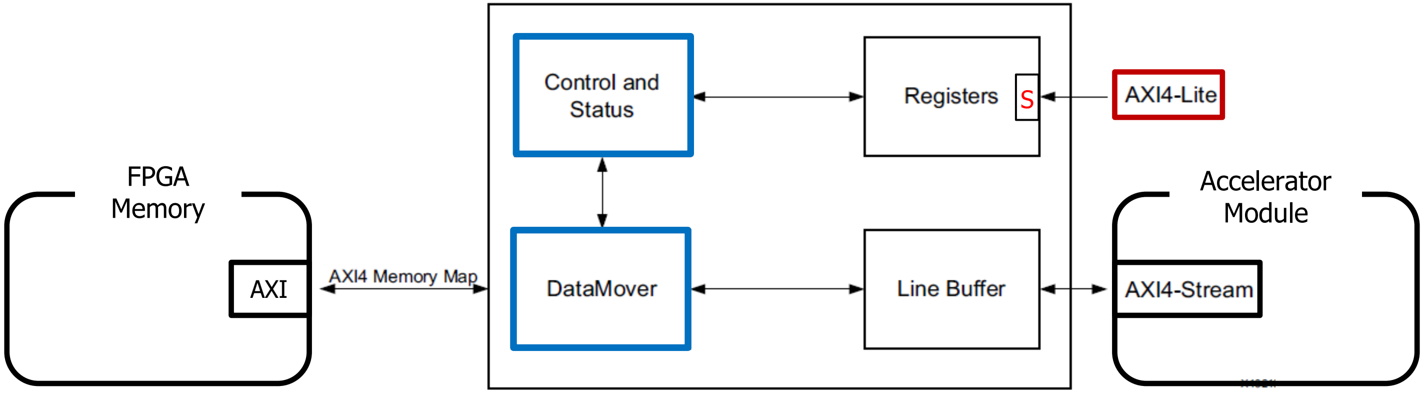
VDMA Engine

- Memory Map
  - Each VDMA Engine control registers are memory-mapped
  - Address = VDMA Engine Base Address + Register address offset

Register type	Address Offset	
	MM2S	S2MM
VDMACR (control)	0x00	0x30
START_ADDRESS	0x5C	0xAC
FRMDLY_STRIDE	0x58	0xA8
HSIZE	0x54	0xA4
VSIZE	0x50	0xA0

VDMA Engine

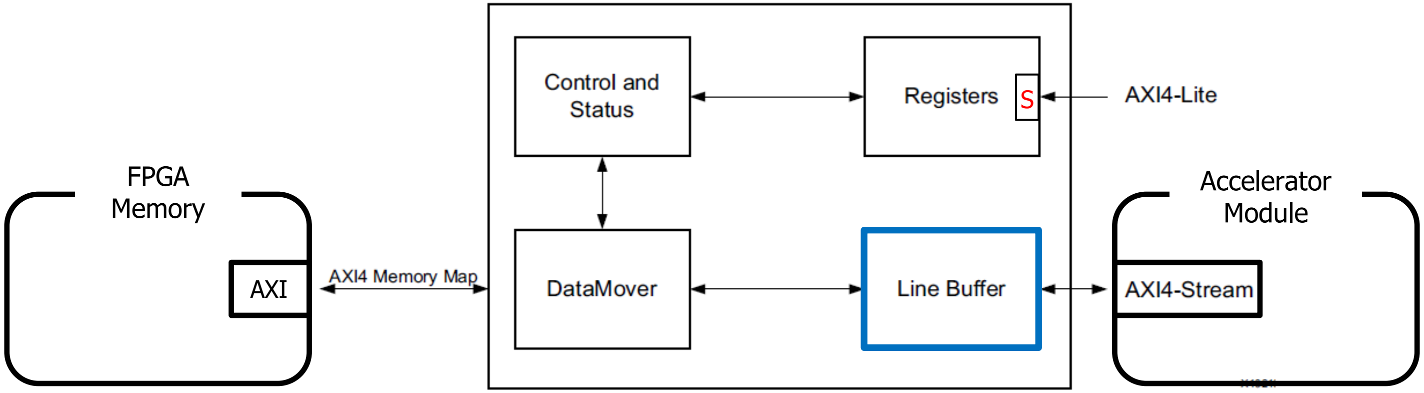
- We program VDMA Engine control registers through AXI4-Lite Interface
- Once all the control registers are set…
  - The **Control/Status** logic generates appropriate commands
  - The **DataMover** executes Read/Write commands to initiate data transfer



AXI VDMA Block Diagram

VDMA Engine

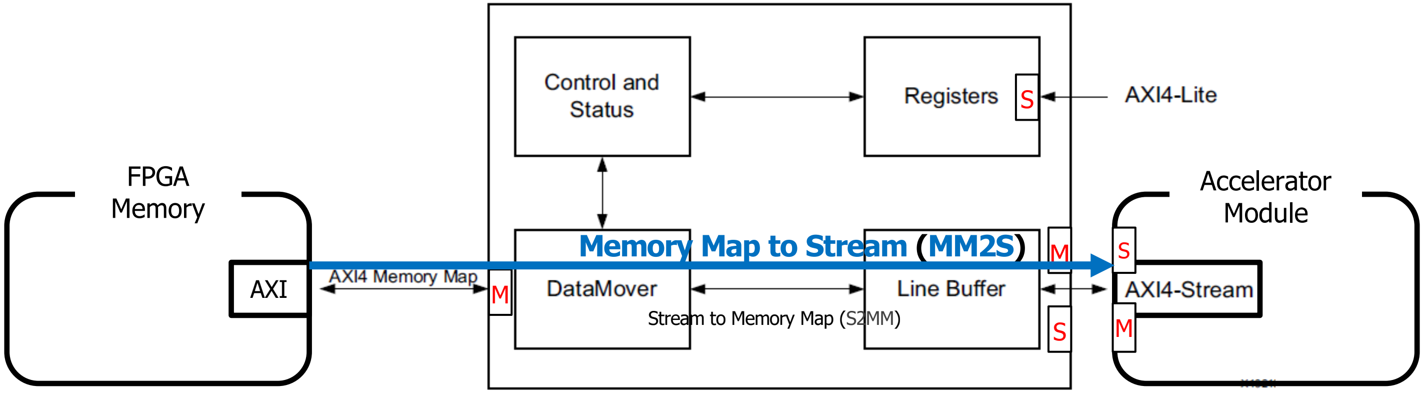
- **Line Buffer**
  - Asynchronous internal buffer to temporarily hold the data
  - VDMA Engine stalls data transfer if the slave is not ready



AXI VDMA Block Diagram

VDMA Engine

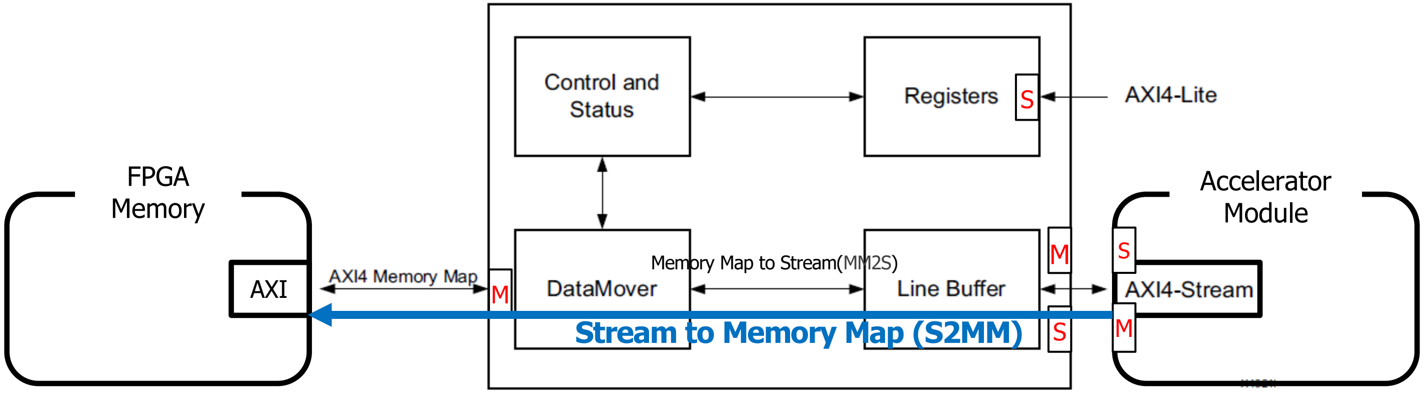
- **Read** data from FPGA memory to the Accelerator Module
  - Interface
    - FPGA Memory ⇔ VDMA Engine: AXI Interface (Memory-mapped)
    - VDMA Engine ⇔ Accelerator Module: AXI-Stream Interface (Stream)
  - Data Flow: FPGA Memory => Accelerator Module (MM2S)
    - FPGA Memory (S_AXI) => VDMA Engine (M_AXI)
      => VDMA Engine (M_AXIS) => Accelerator Module (S_AXIS)



AXI VDMA Block Diagram

VDMA Engine

- **Write** data from the Accelerator Module to FPGA Memory
  - Interface
    - FPGA Memory ⇔ VDMA Engine: AXI Interface (Memory-mapped)
    - VDMA Engine ⇔ Accelerator Module: AXI-Stream Interface (Stream)
  - Data Flow: Accelerator Module => FPGA Memory (S2MM)
    - Accelerator Module (M_AXIS) => VDMA Engine (S_AXIS)
      => VDMA Engine (M_AXI) => FPGA Memory (S_AXI)



AXI VDMA Block Diagram

VDMA Engine

- Summary
  - A set of control registers should be set by the host CPU via AXI-Lite interface to use the VDMA Engine
  - Data transfer directions
    - FPGA Memory => Accelerator Module (MM2S)
    - Accelerator Module => FPGA Memory (S2MM)
  - The unit of data transfer in VDMA is a frame
    - Frame = HSIZE (line size) x VSIZE (# of lines)

Conv Module

- Interface (Ports)
  - APB
  - AXI-Stream

- Internal Module
  - Control Unit
  - Compute Unit
    - **2D Systolic Array (Mandatory)**
      - □ MULT: 8-bit
      - □ ACCUMULATION: bitwidth should be wide enough to provide precise result
    - Quantization Unit
    - ReLU Unit
  - Memory Unit
    - BRAM bandwidth should be large enough to populate 2D Systolic Array without any stall

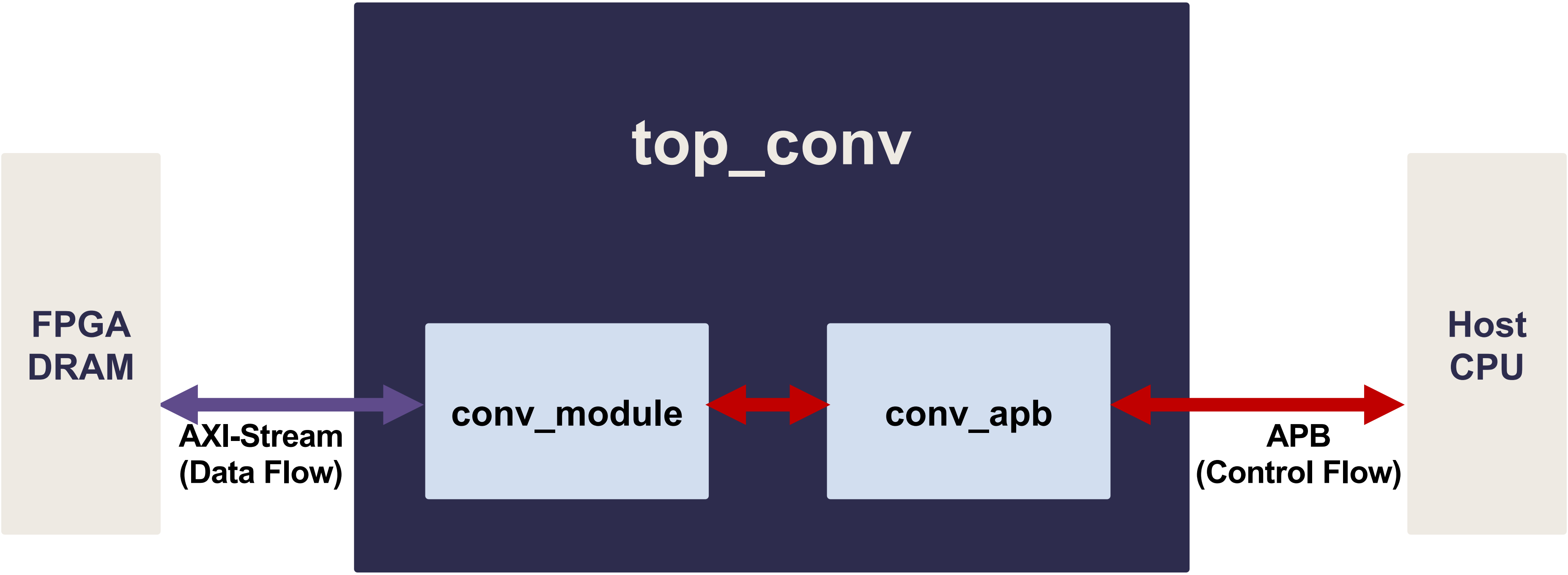
Conv Module

- APB Interface
  - *conv_start* register
    - Host CPU set *conv_start* register via APB protocol
    - The conv module begins operation whenever the *conv_start* register is set
  - *conv_done* register
    - Host CPU busy poll *conv_done* register via APB protocol
    - The conv module sets the *conv_done* register whenever the operation ends
  - You can freely add more memory-mapped registers using the APB protocol

- AXI-Stream
  - I/O ports are given, and unnecessary pins are fixed to default
  - When Pooling Module is a **Master** (send data to VDMA Engine)
    - set M_AXIS_TDATA, M_AXIS_TLAST, M_AXIS_TVALID, etc
  - When Pooling Module is a **Slave** (receive data from VDMA Engine)
    - set S_AXIS_TREADY

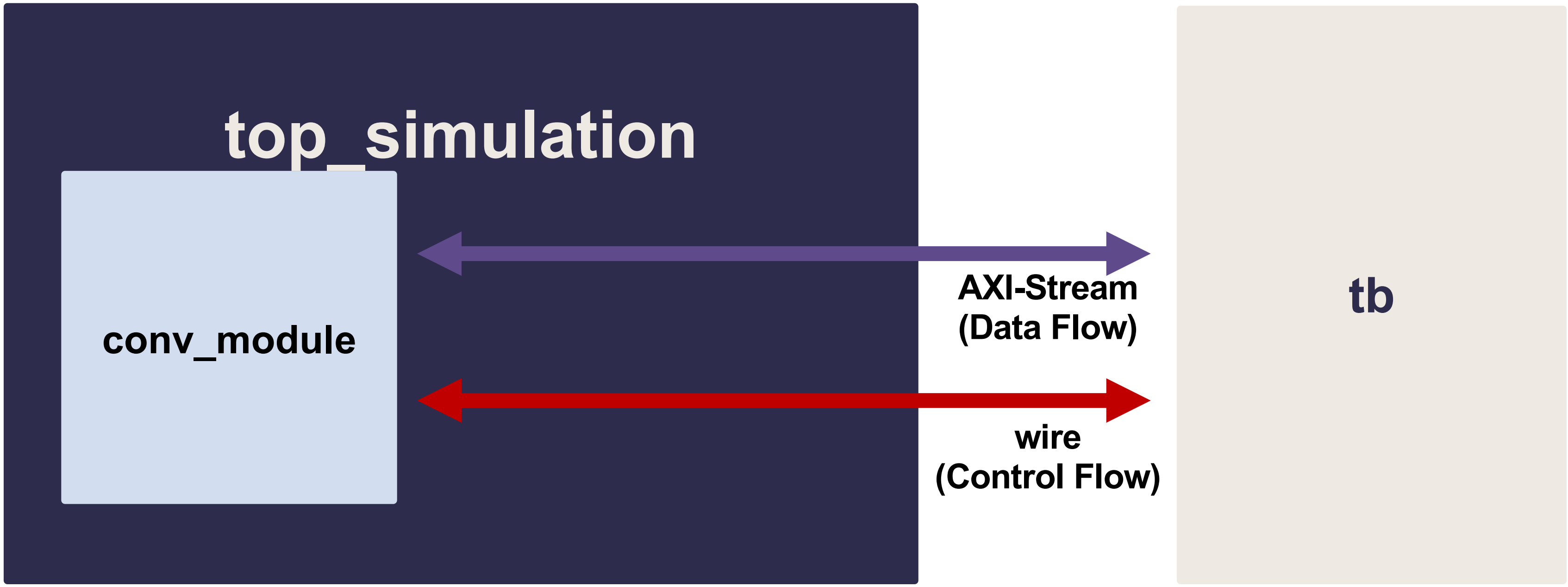
Conv Module

- Block Diagram



Conv Module

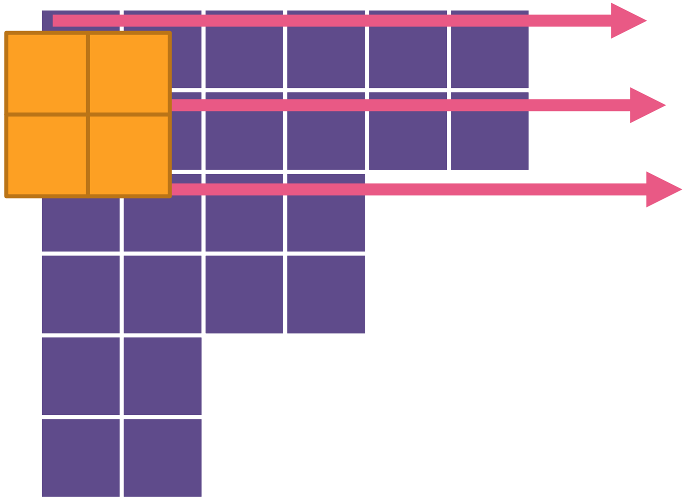
- Testbench
  - Conv module is initiated only after the VDMA Engine transfers all the data
  - This does not strictly match the behavior of the python codes
    - Python code initiates Conv module, then programs the VDMA Engine
  - **Make your testbench and the python codes coherent**



Pooling Module

- Interface (Ports)
  - APB
  - AXI-Stream

- Internal Module
  - Control Unit
  - Compute Unit
    - MAX Unit
  - Memory Unit



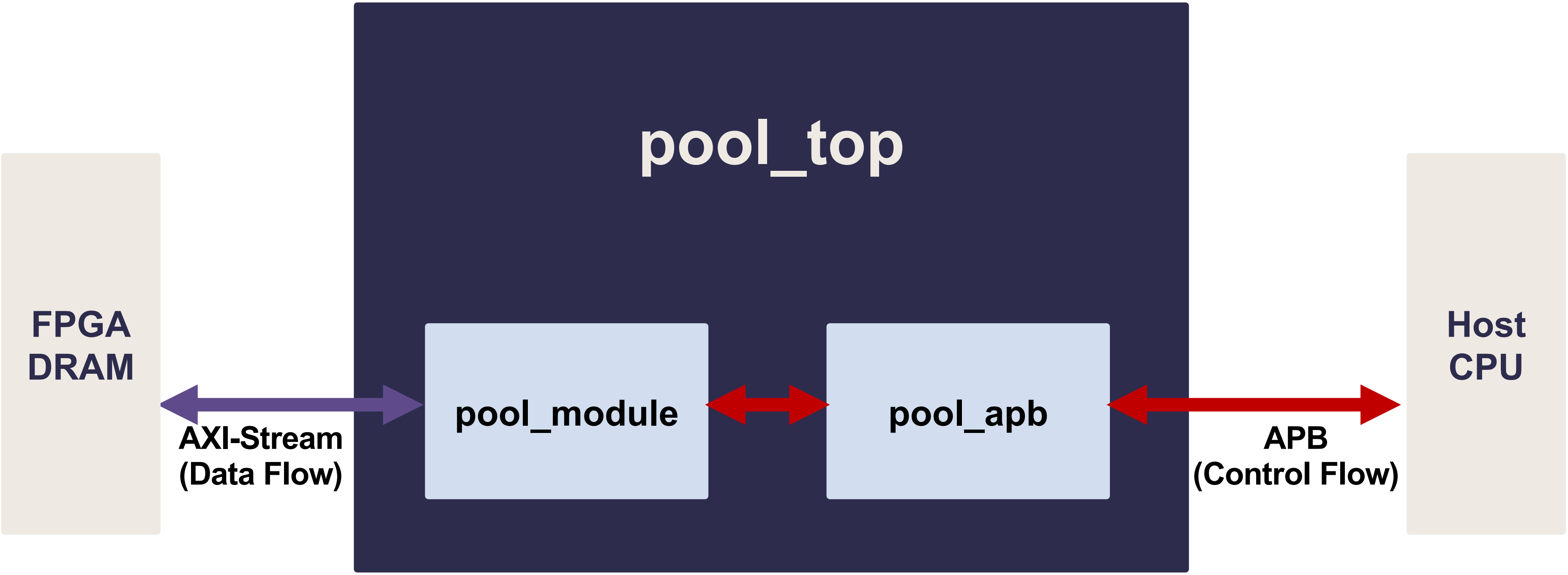
Pooling Module

- APB Interface
  - *pool_start* register
    - Host CPU set *pool_start* register via APB protocol
    - The pooling module begins operation whenever the *pool_start* register is set
  - *pool_done* register
    - Host CPU busy poll *pool_done* register via APB protocol
    - The pooling module sets the *pool_done* register whenever the operation ends
  - You can freely add more memory-mapped registers using the APB protocol

- AXI-Stream
  - I/O ports are given, and unnecessary pins are fixed to default
  - When Pooling Module is a **Master** (send data to VDMA Engine)
    - set M_AXIS_TDATA, M_AXIS_TLAST, M_AXIS_TVALID, etc
  - When Pooling Module is a **Slave** (receive data from VDMA Engine)
    - set S_AXIS_TREADY

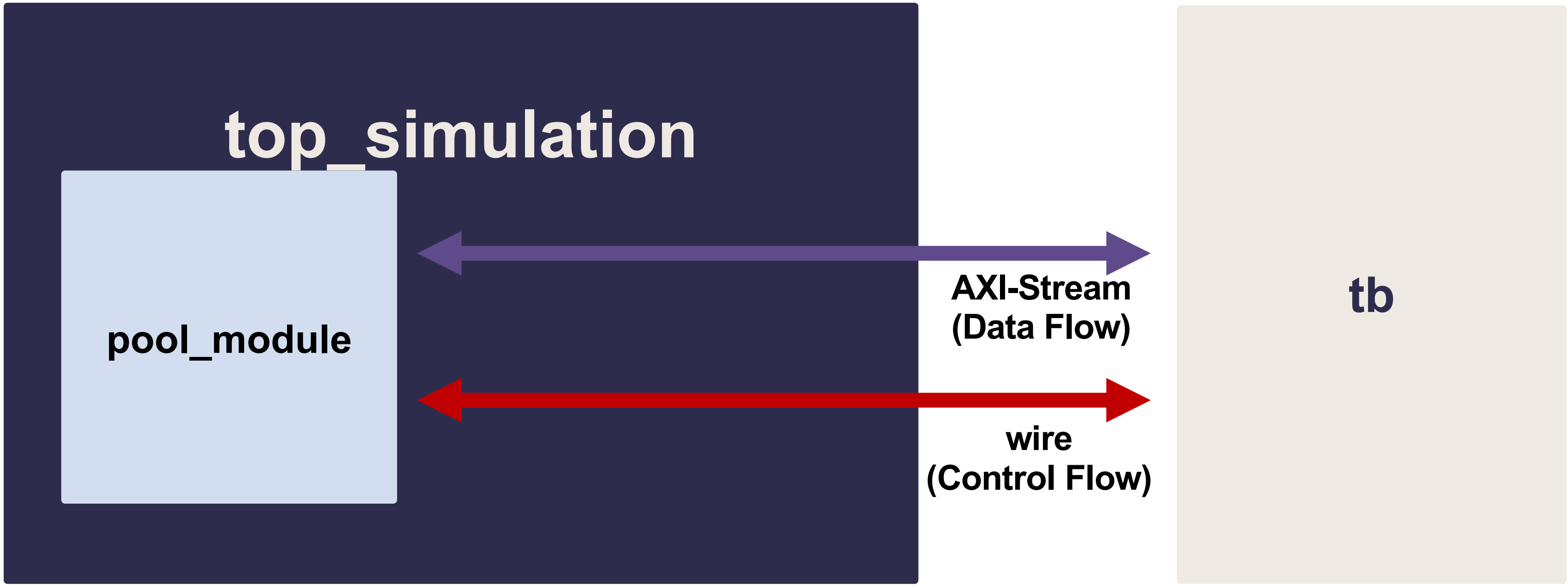
Pooling Module

- Block Diagram



Pooling Module

- Testbench
  - Pooling module is initiated only after the VDMA Engine transfers all the data
  - This does not strictly match the behavior of the python codes
    - Default Python code initiates Pooling module, then programs the VDMA Engine
  - **Make your testbench and the python codes coherent**



FC Module

- Interface (Ports)
  - APB
  - AXI-Stream

- Internal Module
  - Control Unit
  - Compute Unit
    - Any valid unit that can perform FC computation
      - □ MULT: 8-bit
      - □ ACCUMULATION: bitwidth should be wide enough to provide precise result
    - Quantization Unit
    - ReLU Unit
  - Memory Unit

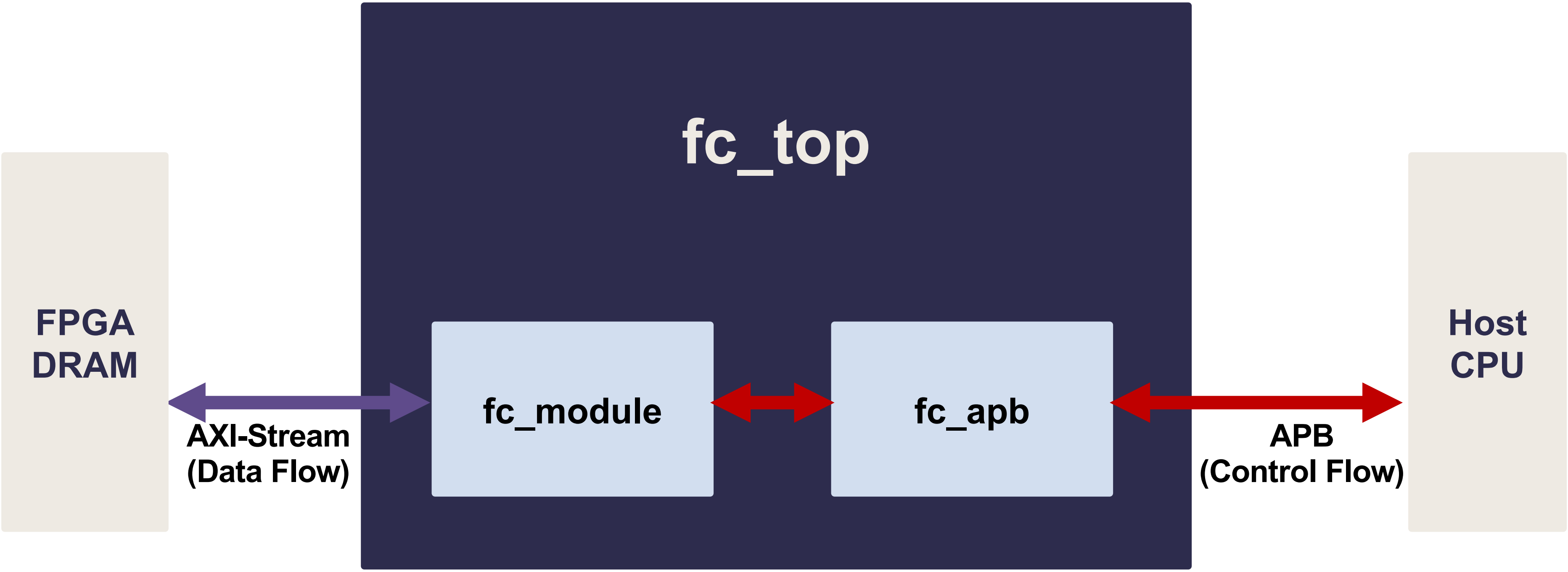
FC Module

- APB Interface
  - *fc_start* register
    - Host CPU set *fc_start* register via APB protocol
    - The fc module begins operation whenever the *fc_start* register is set
  - *fc_done* register
    - Host CPU busy poll *fc_done* register via APB protocol
    - The fc module sets the *fc_done* register whenever the operation ends
  - You can freely add more memory-mapped registers using the APB protocol

- AXI-Stream
  - I/O ports are given, and unnecessary pins are fixed to default
  - When Pooling Module is a **Master** (send data to VDMA Engine)
    - set M_AXIS_TDATA, M_AXIS_TLAST, M_AXIS_TVALID, etc
  - When Pooling Module is a **Slave** (receive data from VDMA Engine)
    - set S_AXIS_TREADY

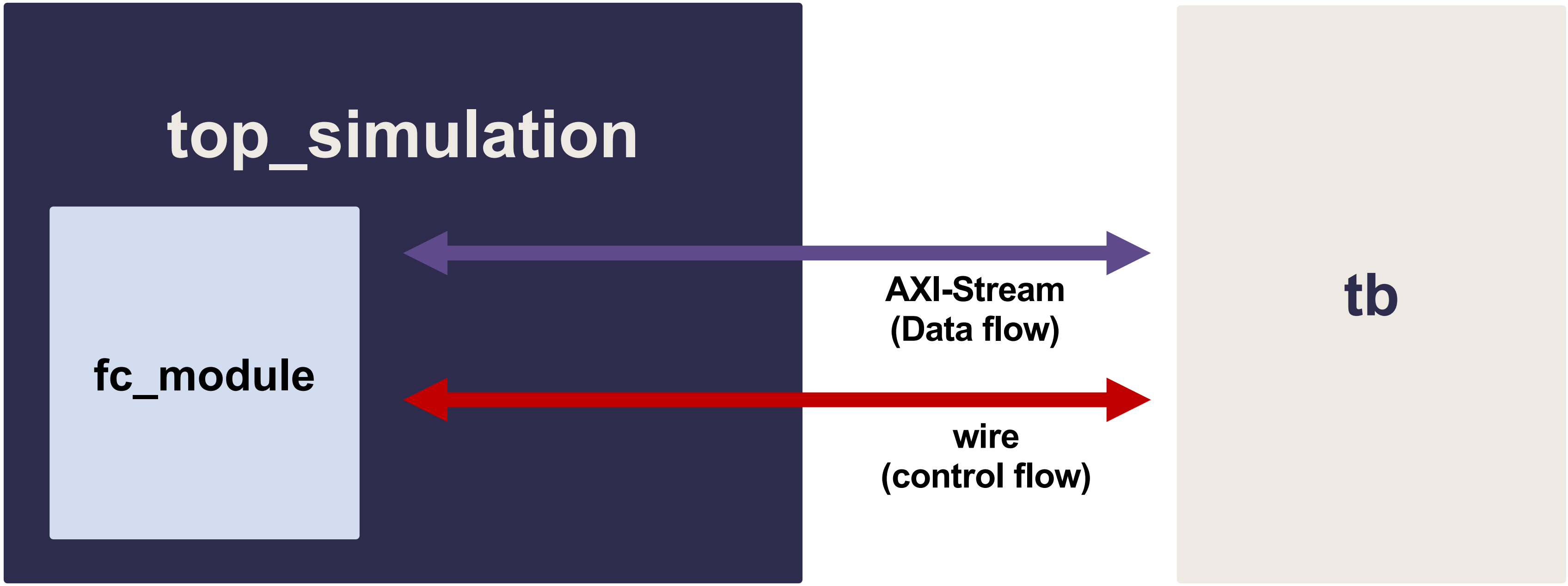
FC Module

- Block Diagram



FC Module

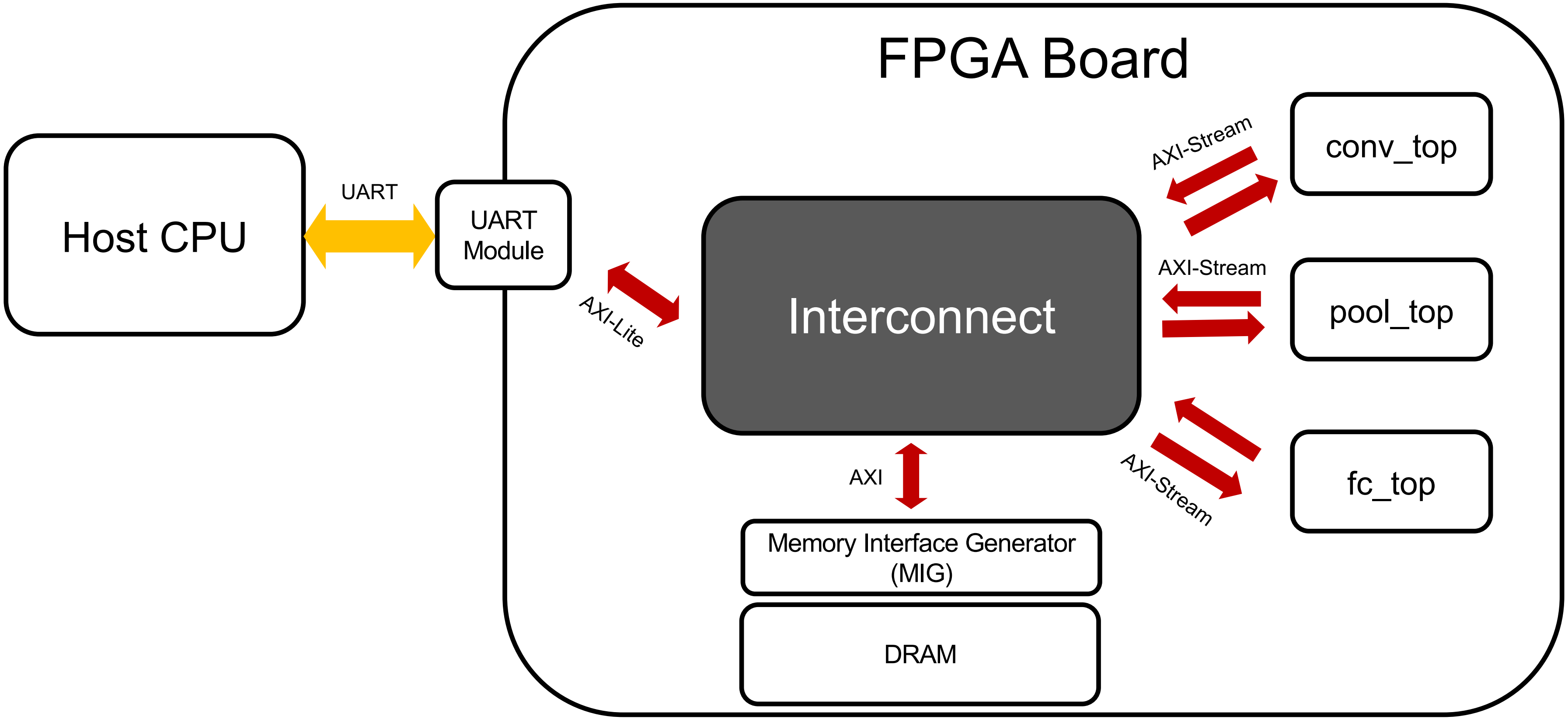
- Testbench
  - FC module is initiated only after the VDMA Engine transfers all the data
  - This does not strictly match the behavior of the python codes
    - Default Python code initiates FC module, then programs the VDMA Engine
  - **Make your testbench and the python codes coherent**



System Overview

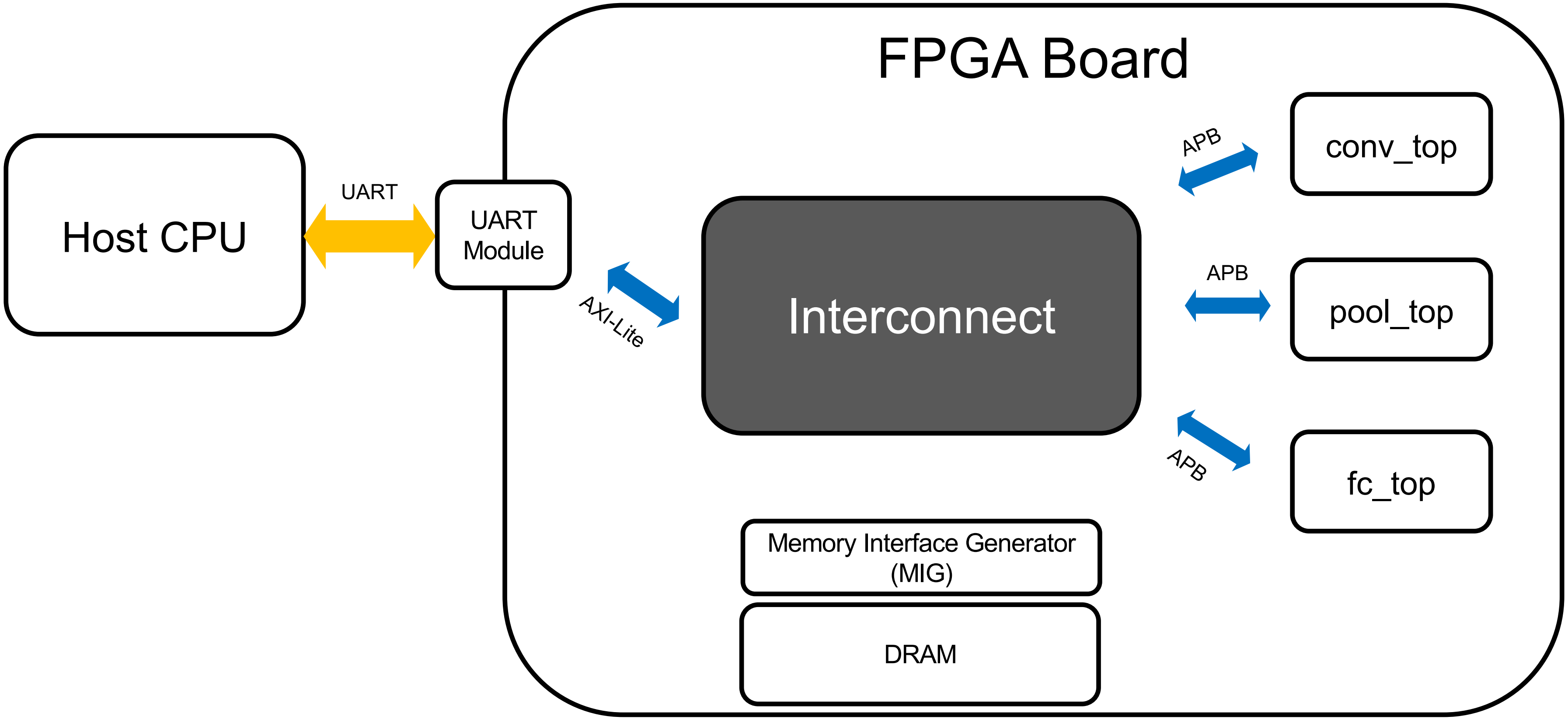
System Overview

- CNN Accelerator System (Control Flow)

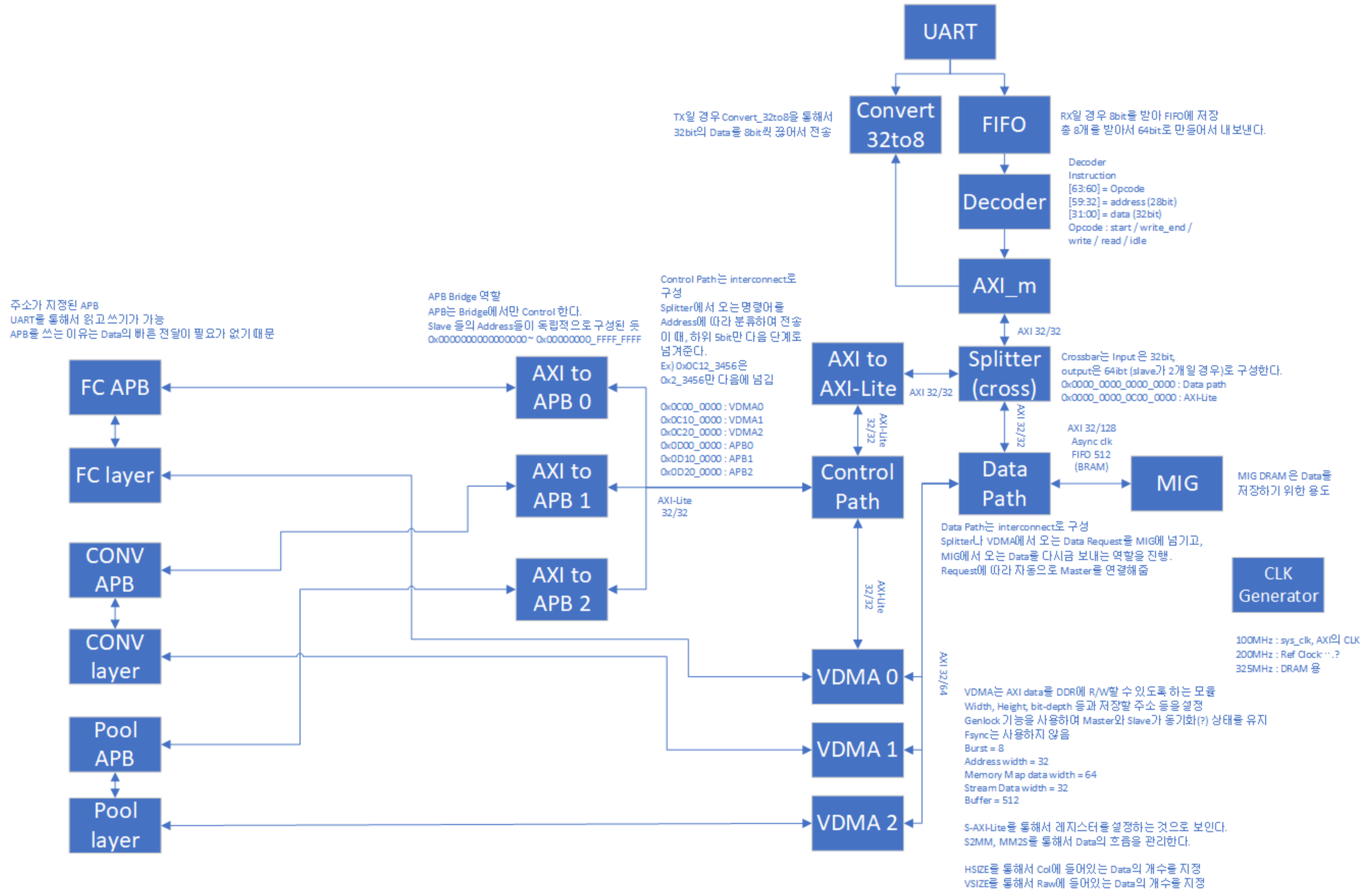


System Overview

- CNN Accelerator System (Control Flow)



System Overview



TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write_end /
write / read / idle

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 들의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Control Path는 interconnect로
구성
Splitter에서 오는 명령어를
Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로
넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

Crossbar는 Input은 32bit,
output은 64ibt (slave가 2개일 경우)로 구성한다.
0x0000_0000_0000_0000 : Data path
0x0000_0000_0C00_0000 : AXI-Lite

MIG DRAM은 Data를
저장하기 위한 용도

Data Path는 interconnect로 구성
Splitter나 VDMA에서 오는 Data Request를 MIG에 넘기고,
MIG에서 오는 Data를 다시금 보내는 역할을 진행.
Request에 따라 자동으로 Master를 연결해줌

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock….?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

Memory Map

- **Each hardware module has its own address space, exposed to the host CPU**

DRAM:	0x0000_0000 ~ 0x0800_0000
VDMA0 Control Registers:	0x0C00_0000 ~ 0x0C10_0000
VDMA1 Control Registers:	0x0C10_0000 ~ 0x0C20_0000
VDMA2 Control Registers:	0x0C20_0000 ~ 0x0C30_0000
APB0 Control Registers:	0x0D00_0000 ~ 0x0D10_0000
APB1 Control Registers:	0x0D10_0000 ~ 0x0D20_0000
APB2 Control Registers:	0x0D20_0000 ~ 0x0D30_0000

- Instructions from the host CPU are **decoded** by the ***decode module***, then **arbitrated** by the ***interconnects***

Instructions

- We define a bunch of instructions to run the FPGA accelerator from the host CPU
  - 64-bit Instruction
    - [63:60] = Opcode (Read: 5, Write: 4)
    - [59:32] = Address
    - [31:00] = Data

- Instructions are essentially **Read/Write requests with varying memory-mapped addresses**
  - Instruction 1: Write data from the host CPU to the FPGA DRAM
  - Instruction 2: Program the Accelerator Module
  - Instruction 3: Program the VDMA Engine
  - Instruction 4: Read the result from the FPGA DRAM to the host CPU
  - …

Instruction 1 (Python Code)

- **Write data from the host CPU to the FPGA DRAM**
  - Data: Input Images, CNN Model parameters

- single_layer_test.ipynb
  - Pseudo Code

```
…
SU.su_set_conv_w({‘BASE_ADDR’: 0x0200_0000}, “~.npy”)
```

- scale_uart.py
  - Pseudo Code

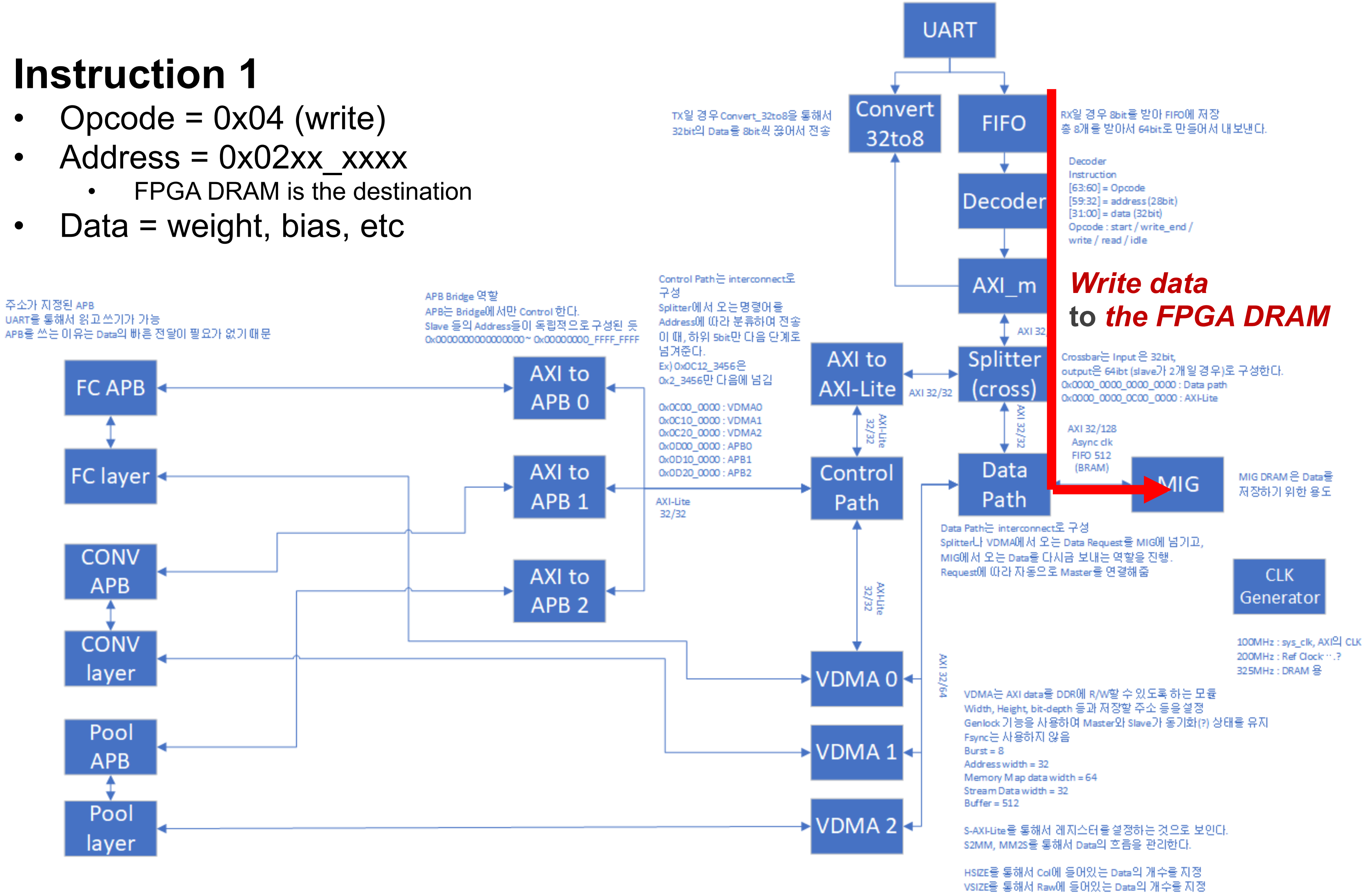
```
def su_set_conv_w(self, weight_info, f_name):
    …
    self.su_write_data_trans(base_addr + (cout – 4), val)
    …
```

```
Opcode:   Write
Address:  0x02xx_xxxx
Data:     val
```

Instruction 1 (FPGA Flow)

Instruction 1

- Opcode = 0x04 (write)
- Address = 0x02xx_xxxx
  - FPGA DRAM is the destination
- Data = weight, bias, etc



Write data
to ***the FPGA DRAM***

TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write_end /
write / read / idle

Control Path는 interconnect로
구성
Splitter에서 오는 명령어를
Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로
넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 등의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Crossbar는 Input은 32bit,
output은 64ibt (slave가 2개일 경우)로 구성한다.
0x0000_0000_0000_0000 : Data path
0x0000_0000_0C00_0000 : AXI-Lite

MIG DRAM은 Data를
저장하기 위한 용도

Data Path는 interconnect로 구성
Splitter나 VDMA에서 오는 Data Request를 MIG에 넘기고,
MIG에서 오는 Data을 다시금 보내는 역할을 진행.
Request에 따라 자동으로 Master를 연결해줌

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock….?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

Instruction 2 (Python Code)

- **Program the Accelerator Module**
  - Data: APB Commands

- single_layer_test.ipynb
  - Pseudo Code

```
FC_BASE_ADDR = 0x0D00_0000
…
SU.su_fc_control(I, F, W, B, R, VDMA0_BASE_ADDR, FC_BASE_ADDR)
```

- scale_uart.py
  - Pseudo Code

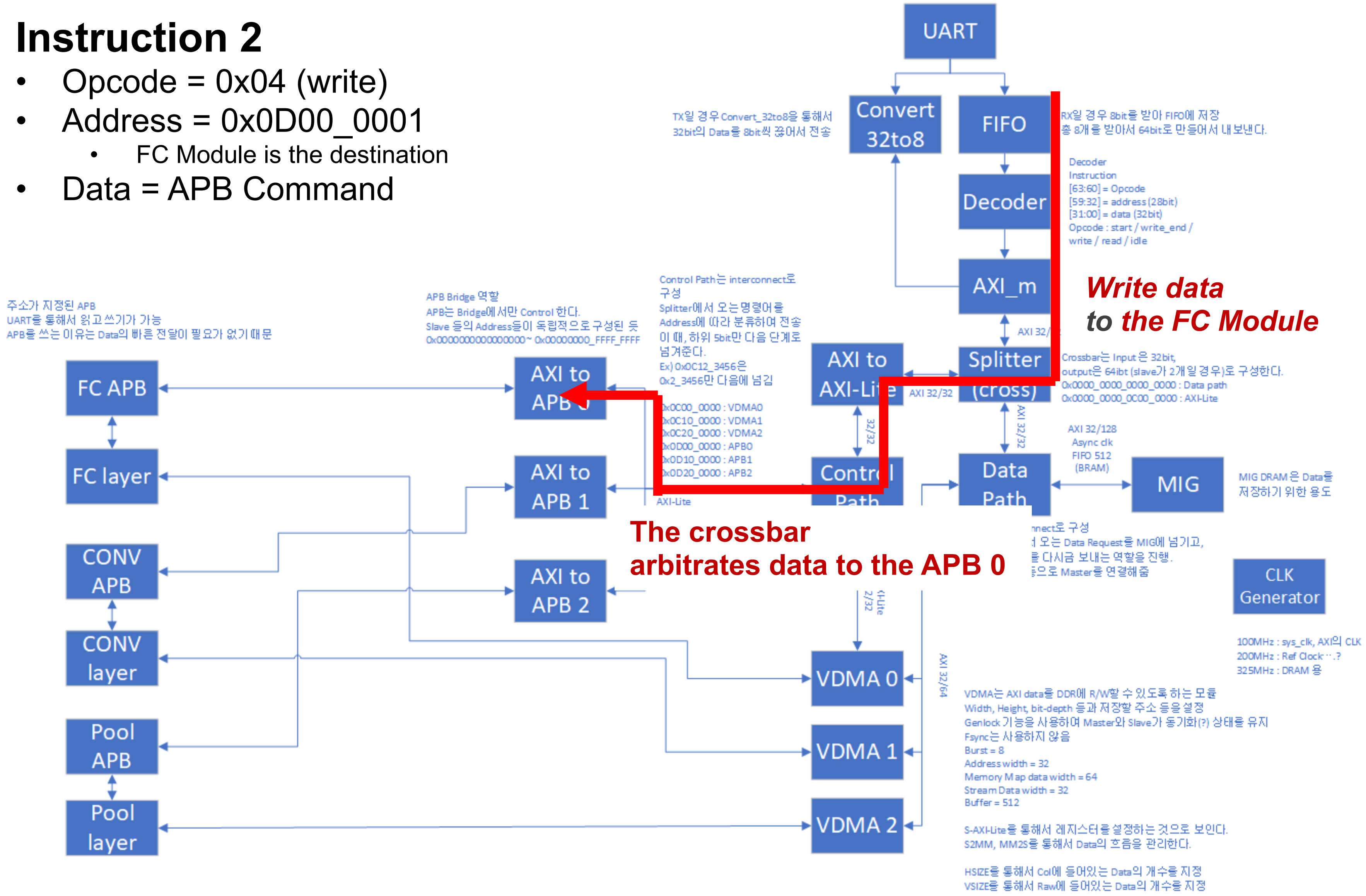
```
def su_fc_control(self, F, W, B, R, vaddr, caddr):
    …
    self.su_write_data(caddr + 0x00, 0x1)
    …
```

Opcode: Write
Address: 0x0D00_0001
Data: 0x1

Instruction 2 (FPGA Flow)

Instruction 2

- Opcode = 0x04 (write)
- Address = 0x0D00_0001
  - FC Module is the destination
- Data = APB Command



TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write_end /
write / read / idle

Write data
to ***the FC Module***

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 등의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Control Path는 interconnect로
구성
Splitter에서 오는 명령어를
Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로
넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

Crossbar는 Input은 32bit,
output은 64ibt (slave가 2개일 경우)로 구성한다.
0x0000_0000_0000_0000 : Data path
0x0000_0000_0C00_0000 : AXI-Lite

AXI 32/128
Async clk
FIFO 512
(BRAM)

MIG DRAM은 Data를
저장하기 위한 용도

The crossbar
arbitrates data to the APB 0

nnect로 구성
ㅓ 오는 Data Request를 MIG에 넘기고,
를 다시금 보내는 역할을 진행.
등으로 Master를 연결해줌

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock … .?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

Instruction 3 (Python Code)

- **Program the VDMA Engine**
  - Data: VDMA Control Values

- single_layer_test.ipynb
  - Pseudo Code

```
VDMA0_BASE_ADDR = 0x0C00_0000
…
SU.su_fc_control(I, F, W, B, R, VDMA0_BASE_ADDR, FC_BASE_ADDR)
```

- scale_uart.py
  - Pseudo Code

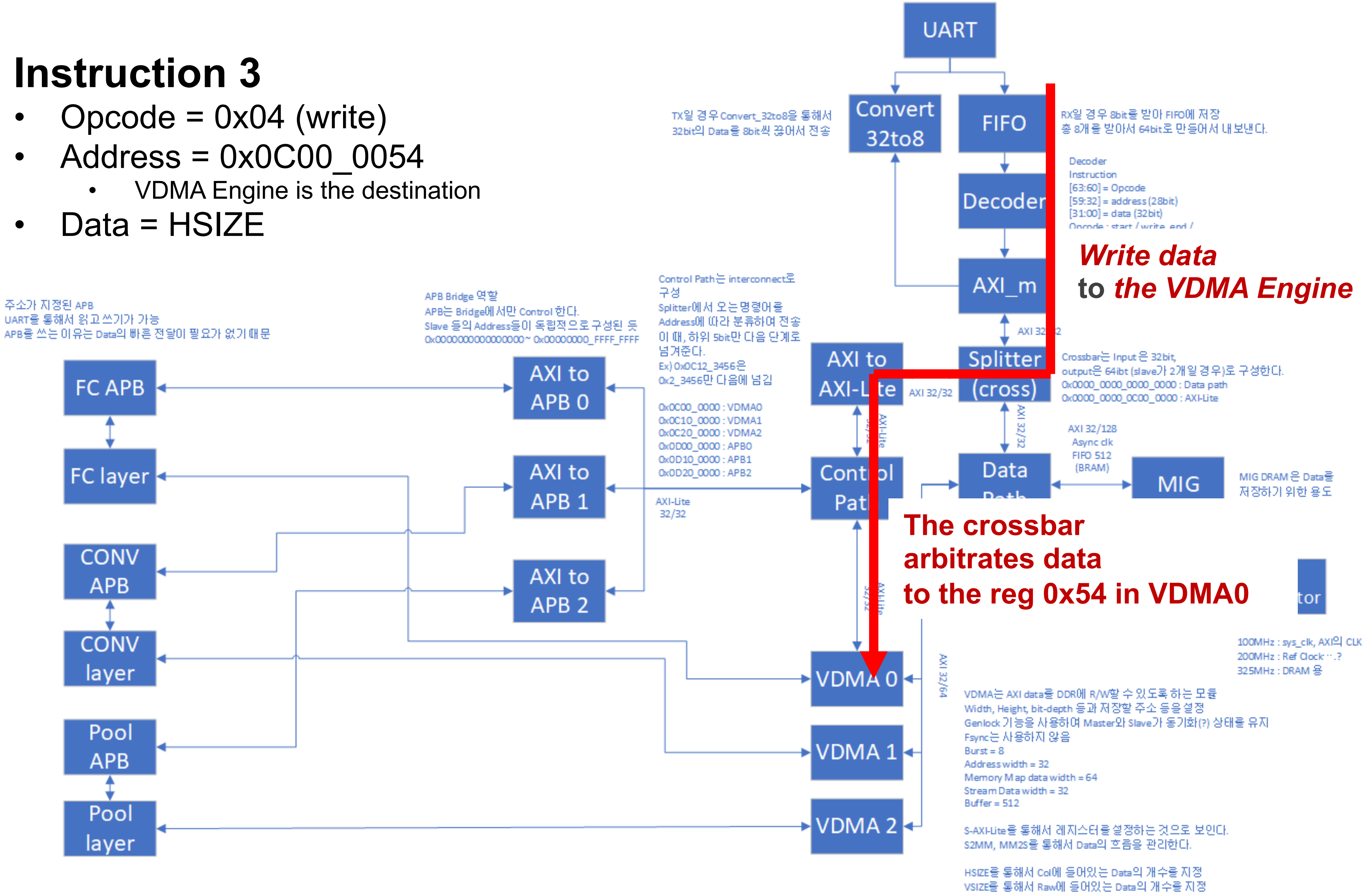
```
def su_fc_control(self, F, W, B, R, vaddr, caddr):
    …
    self.su_write_data(vaddr + 0x54, F[‘HSIZE’])
    …
```

Opcode: Write
Address: 0x0C00_0054
Data: F[‘HSIZE’]

Instruction 3 (FPGA Flow)

Instruction 3

- Opcode = 0x04 (write)
- Address = 0x0C00_0054
  - VDMA Engine is the destination
- Data = HSIZE



TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write, end /

Write data
to *the VDMA Engine*

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 등의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Control Path는 interconnect로 구성
Splitter에서 오는 명령어를 Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로 넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

Crossbar는 Input은 32bit,
output은 64ibt (slave가 2개일 경우)로 구성한다.
0x0000_0000_0000_0000 : Data path
0x0000_0000_0C00_0000 : AXI-Lite

MIG DRAM은 Data를
저장하기 위한 용도

The crossbar
arbitrates data
to the reg 0x54 in VDMA0

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock….?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

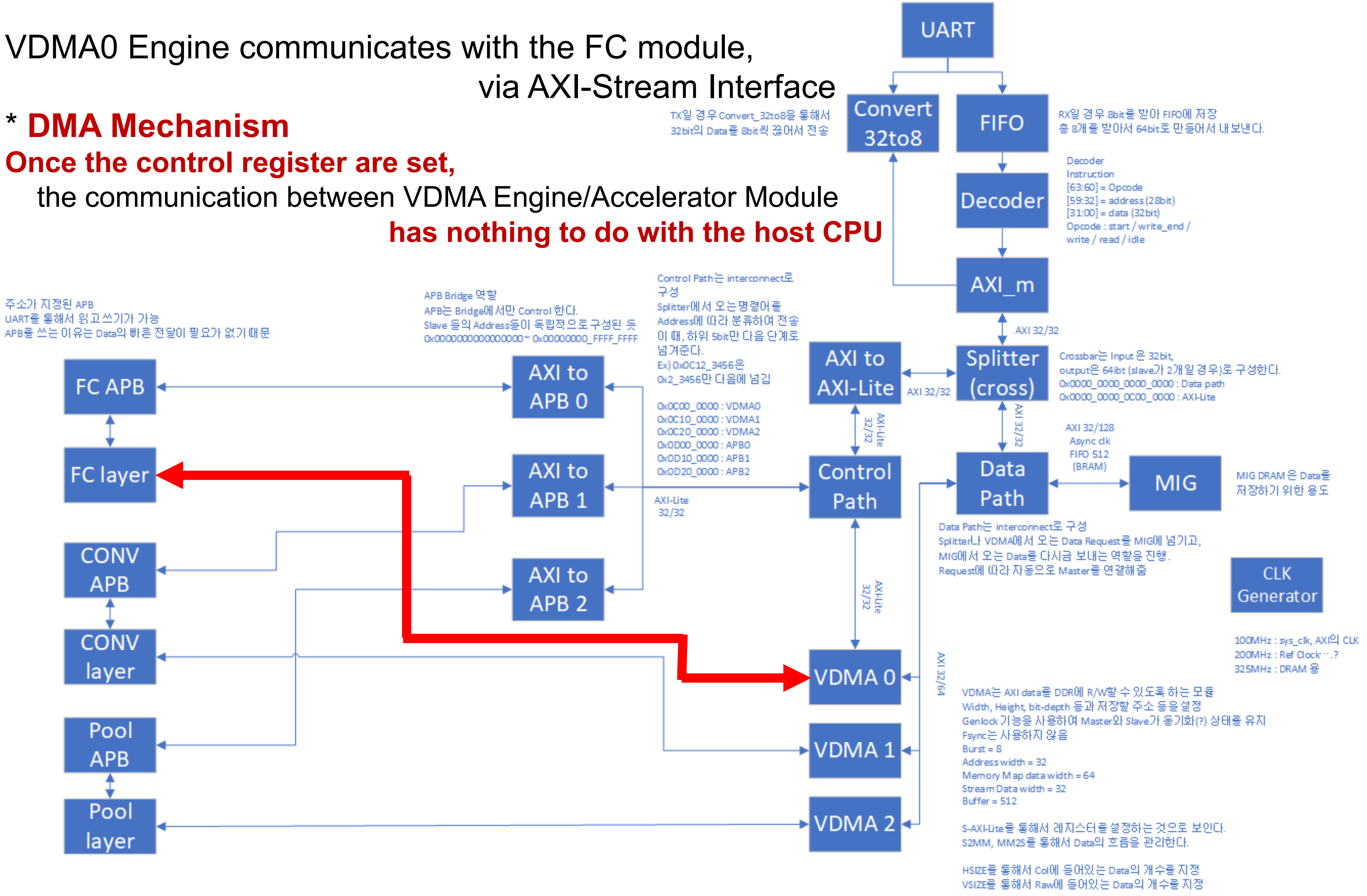
HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

VDMA Mechanism (FPGA Flow)

VDMA0 Engine communicates with the FC module,
via AXI-Stream Interface

* **DMA Mechanism**

Once the control register are set,
the communication between VDMA Engine/Accelerator Module
has nothing to do with the host CPU



TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write_end /
write / read / idle

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 등의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Control Path는 interconnect로
구성
Splitter에서 오는 명령어를
Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로
넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

Crossbar는 Input은 32bit,
output은 64ibt (slave가 2개일 경우)로 구성한다.
0x0000_0000_0000_0000 : Data path
0x0000_0000_0C00_0000 : AXI-Lite

MIG DRAM은 Data를
저장하기 위한 용도

Data Path는 interconnect로 구성
Splitter나 VDMA에서 오는 Data Request를 MIG에 넘기고,
MIG에서 오는 Data를 다시금 보내는 역할을 진행.
Request에 따라 자동으로 Master를 연결해줌

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock….?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

Instruction 4 (Python Code)

- **Read the inference result from the FPGA DRAM to the Host CPU**

- single_layer_test.ipynb
  - Pseudo Code

```
…
SU.su_check_result("~.txt", 0x06A0_0000)
```

- scale_uart.py
  - Pseudo Code

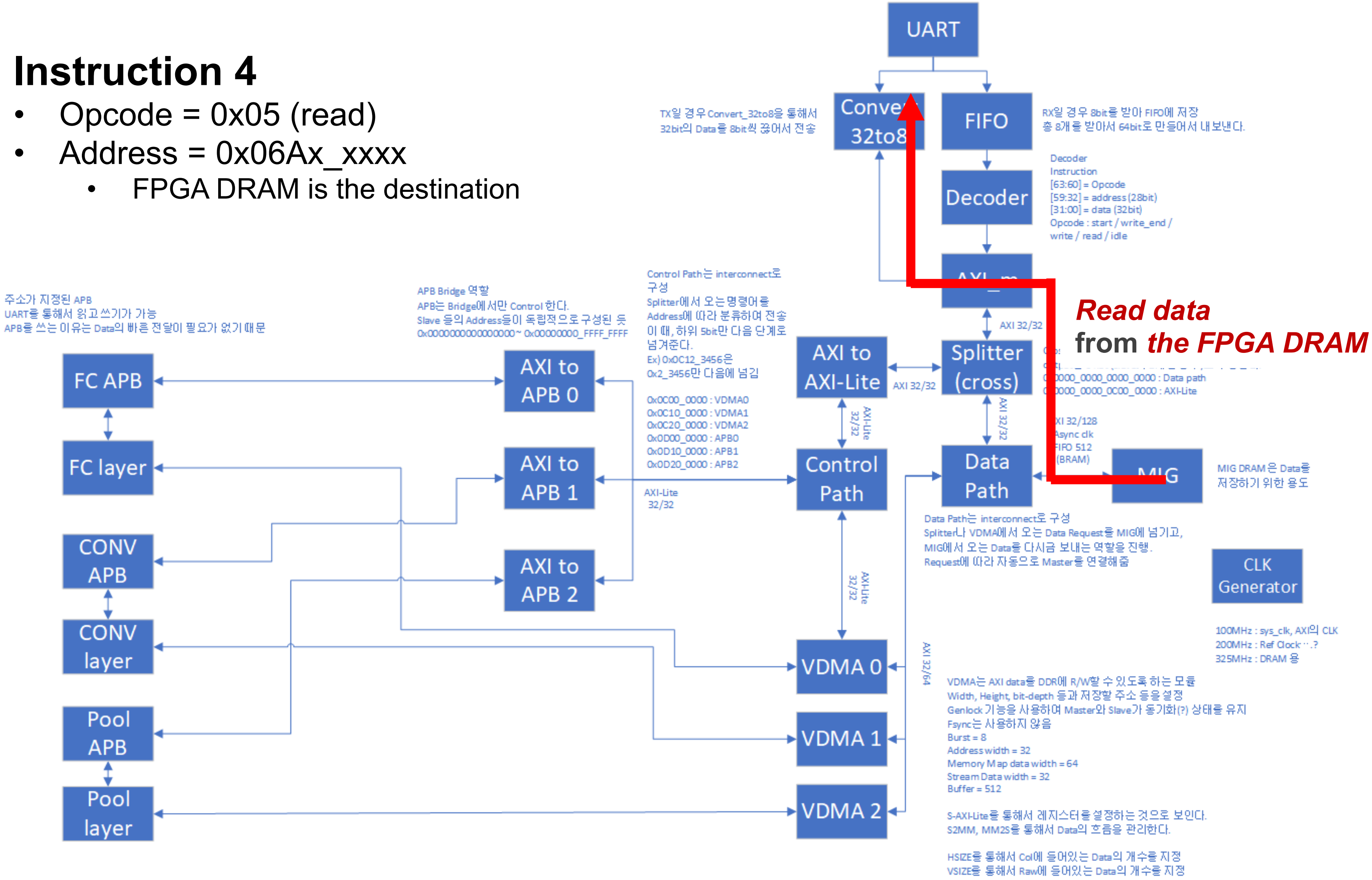
```
def check_result(self, file_name, base_addr):
    …
    data_test = self.su_read_data(base_addr + count * 4)   ←
    …
```

Opcode: Read
Address: 0x06Ax_xxxx

Instruction 4 (FPGA Flow)

Instruction 4

- Opcode = 0x05 (read)
- Address = 0x06Ax_xxxx
  - FPGA DRAM is the destination



Read data
from *the FPGA DRAM*

TX일 경우 Convert_32to8을 통해서
32bit의 Data를 8bit씩 끊어서 전송

RX일 경우 8bit를 받아 FIFO에 저장
총 8개를 받아서 64bit로 만들어서 내보낸다.

Decoder
Instruction
[63:60] = Opcode
[59:32] = address (28bit)
[31:00] = data (32bit)
Opcode : start / write_end /
write / read / idle

주소가 지정된 APB
UART를 통해서 읽고 쓰기가 가능
APB를 쓰는 이유는 Data의 빠른 전달이 필요가 없기 때문

APB Bridge 역할
APB는 Bridge에서만 Control 한다.
Slave 들의 Address들이 독립적으로 구성된 듯
0x0000000000000000~ 0x00000000_FFFF_FFFF

Control Path는 interconnect로
구성
Splitter에서 오는 명령어를
Address에 따라 분류하여 전송
이 때, 하위 5bit만 다음 단계로
넘겨준다.
Ex) 0x0C12_3456은
0x2_3456만 다음에 넘김

0x0C00_0000 : VDMA0
0x0C10_0000 : VDMA1
0x0C20_0000 : VDMA2
0x0D00_0000 : APB0
0x0D10_0000 : APB1
0x0D20_0000 : APB2

0000_0000_0000_0000 : Data path
0000_0000_0C00_0000 : AXI-Lite

XI 32/128
Async clk
FIFO 512
(BRAM)

MIG DRAM은 Data를
저장하기 위한 용도

Data Path는 interconnect로 구성
Splitter나 VDMA에서 오는 Data Request를 MIG에 넘기고,
MIG에서 오는 Data를 다시금 보내는 역할을 진행.
Request에 따라 자동으로 Master를 연결해줌

100MHz : sys_clk, AXI의 CLK
200MHz : Ref Clock….?
325MHz : DRAM 용

VDMA는 AXI data를 DDR에 R/W할 수 있도록 하는 모듈
Width, Height, bit-depth 등과 저장할 주소 등을 설정
Genlock 기능을 사용하여 Master와 Slave가 동기화(?) 상태를 유지
Fsync는 사용하지 않음
Burst = 8
Address width = 32
Memory Map data width = 64
Stream Data width = 32
Buffer = 512

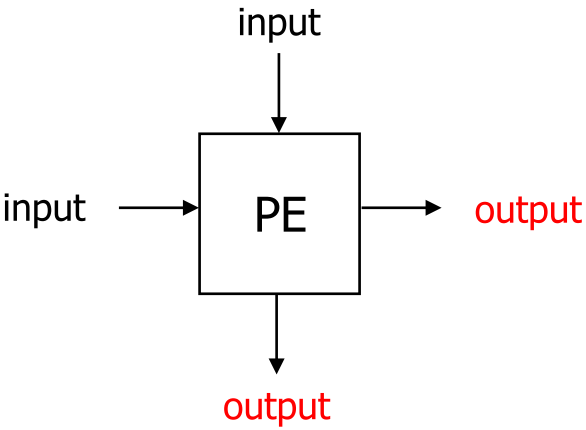
S-AXI-Lite를 통해서 레지스터를 설정하는 것으로 보인다.
S2MM, MM2S를 통해서 Data의 흐름을 관리한다.

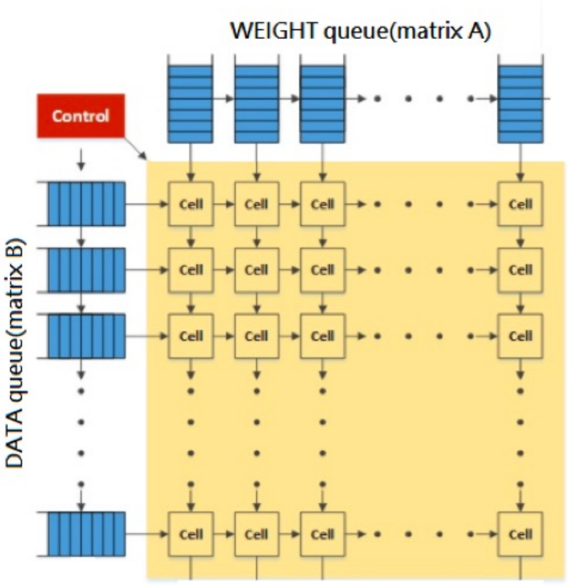
HSIZE를 통해서 Col에 들어있는 Data의 개수를 지정
VSIZE를 통해서 Raw에 들어있는 Data의 개수를 지정

Systolic Array

Systolic Array

- A set of tightly coupled processing elements (PE)
  - Each PE is responsible for computing a partial result of the entire operation
  - Input, Output to the PE is given from the neighboring PEs
- Advantages
  - Maximize data reuse opporunity
  - Simple control logic





Google TPU: example systolic-array application

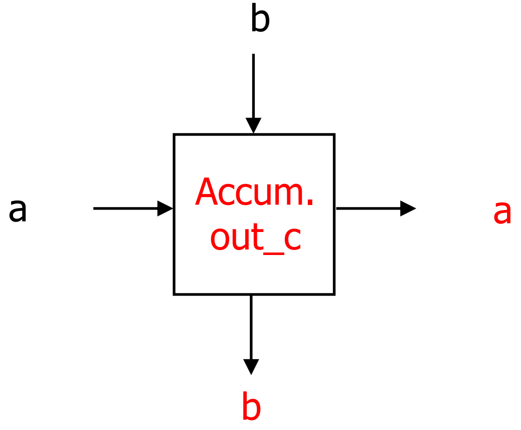
2D Systolic Array (Example)

- Matrix-Matrix multiplication
  - Let's multiply two 3x3 matrices A and B
- Example PE Pseudo code (Output Stationary)

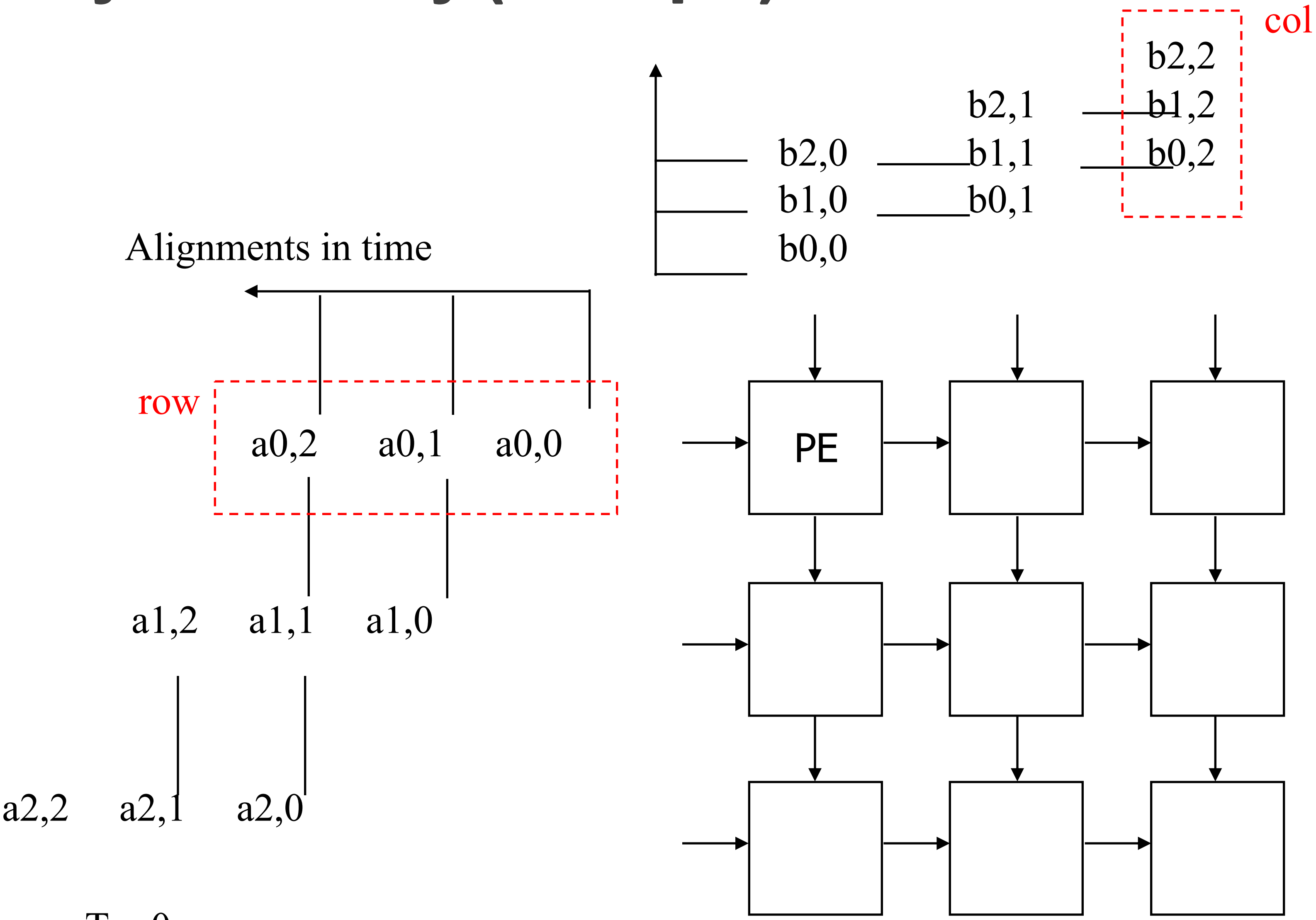
```
module pe(clk,in_a,in_b,out_a,out_b,out_c);
  input wire reset,clk;
  input wire [data_size-1:0] in_a,in_b;
  output reg [2*data_size:0] out_c;
  output reg [data_size-1:0] out_a,out_b;

  always @(posedge clk) begin
    out_c <= out_c + in_a*in_b;
    out_a <= in_a;
    out_b <= in_b;
  end
endmodule
```

Resize the bitwidth
so that no overflow occurs

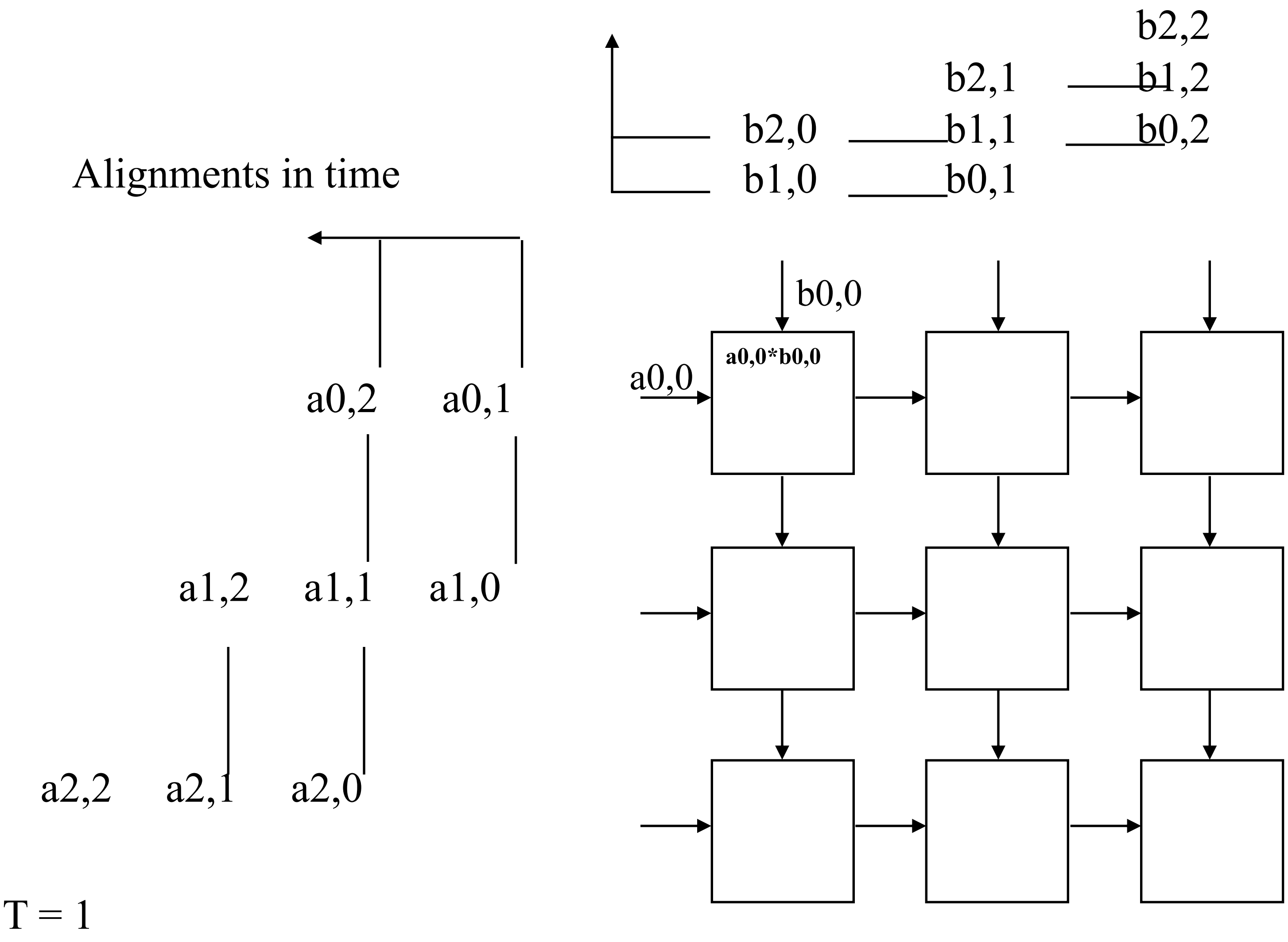


2D Systolic Array (Example)

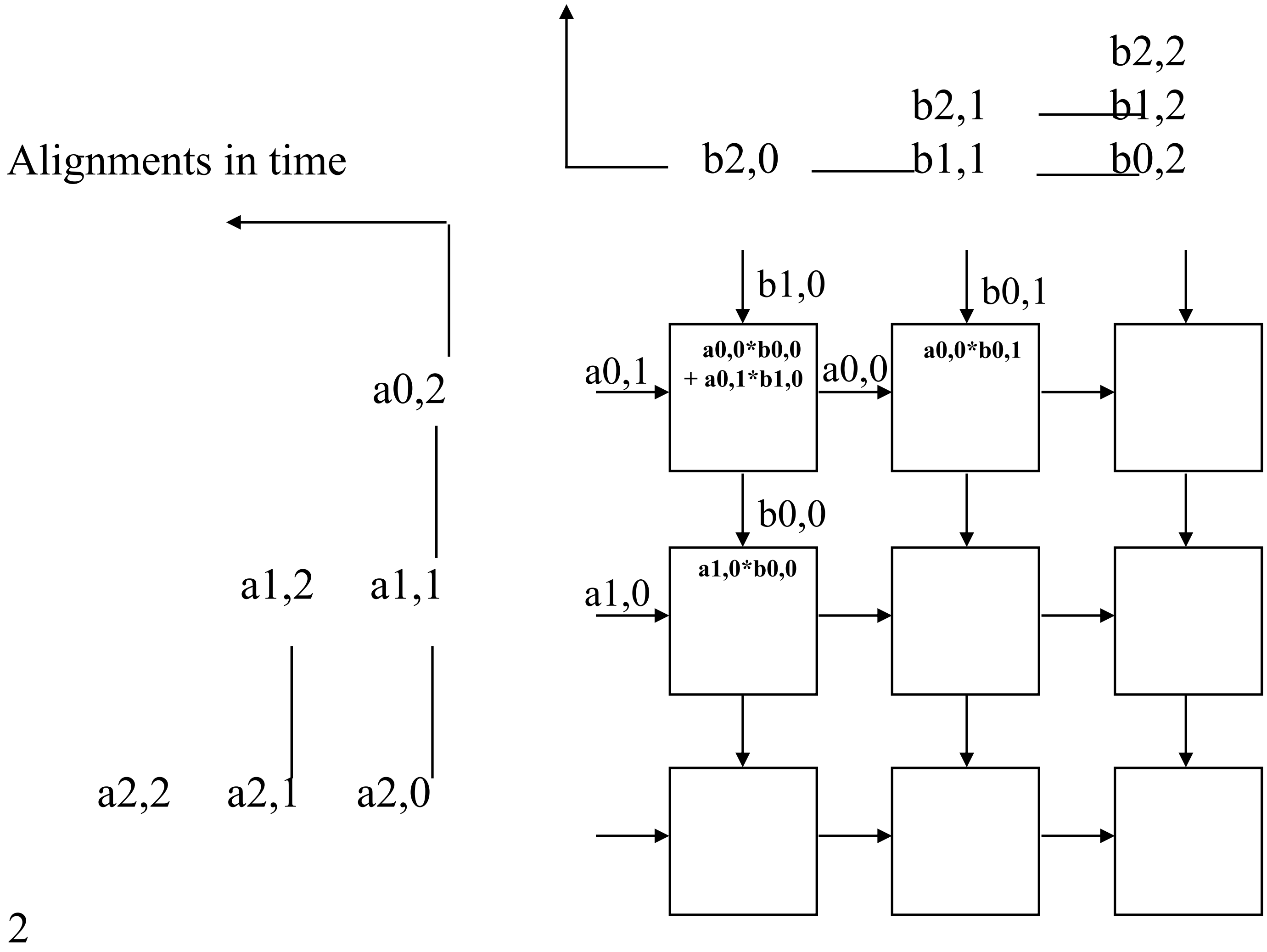


T = 0

2D Systolic Array (Example)

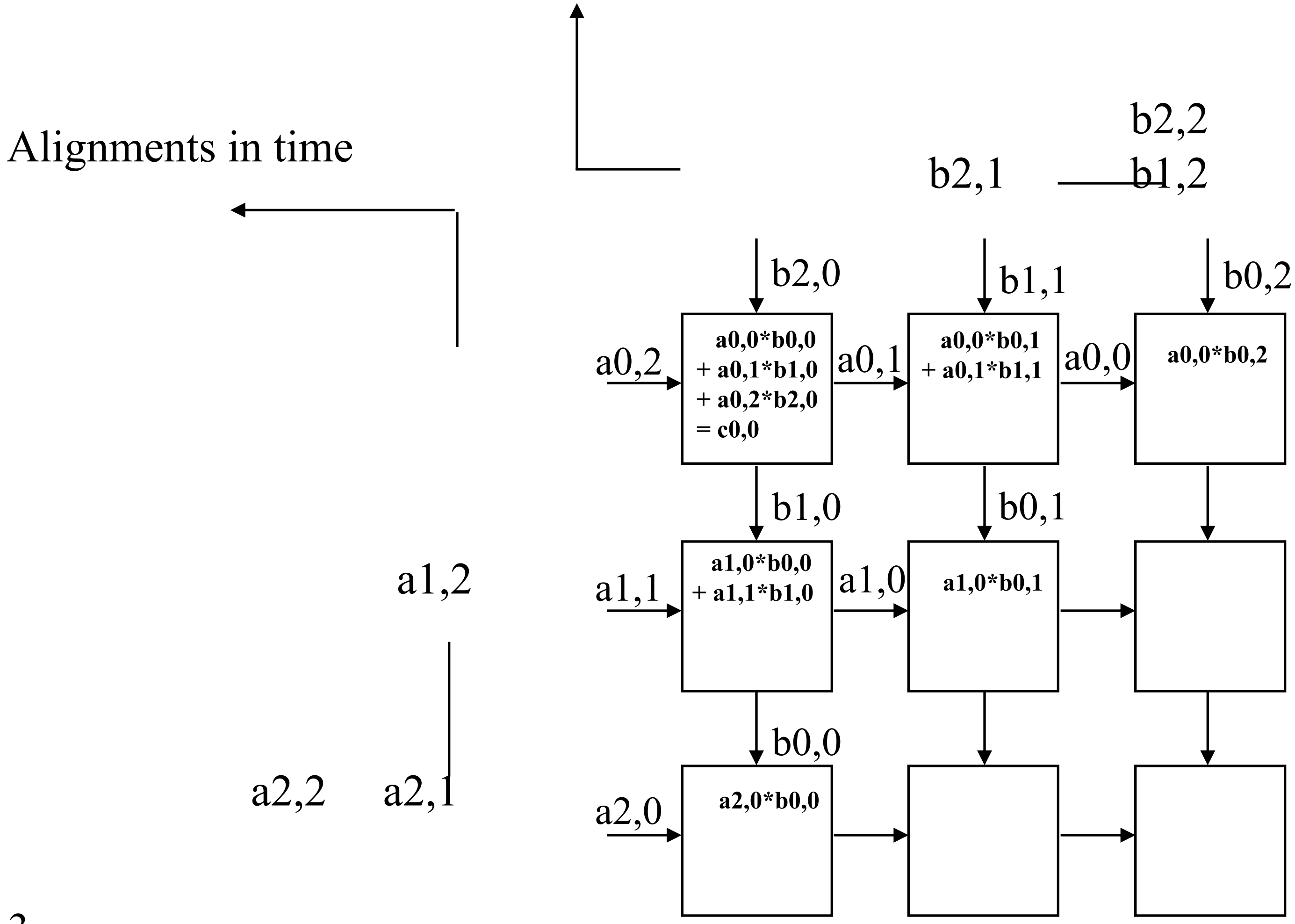


2D Systolic Array (Example)



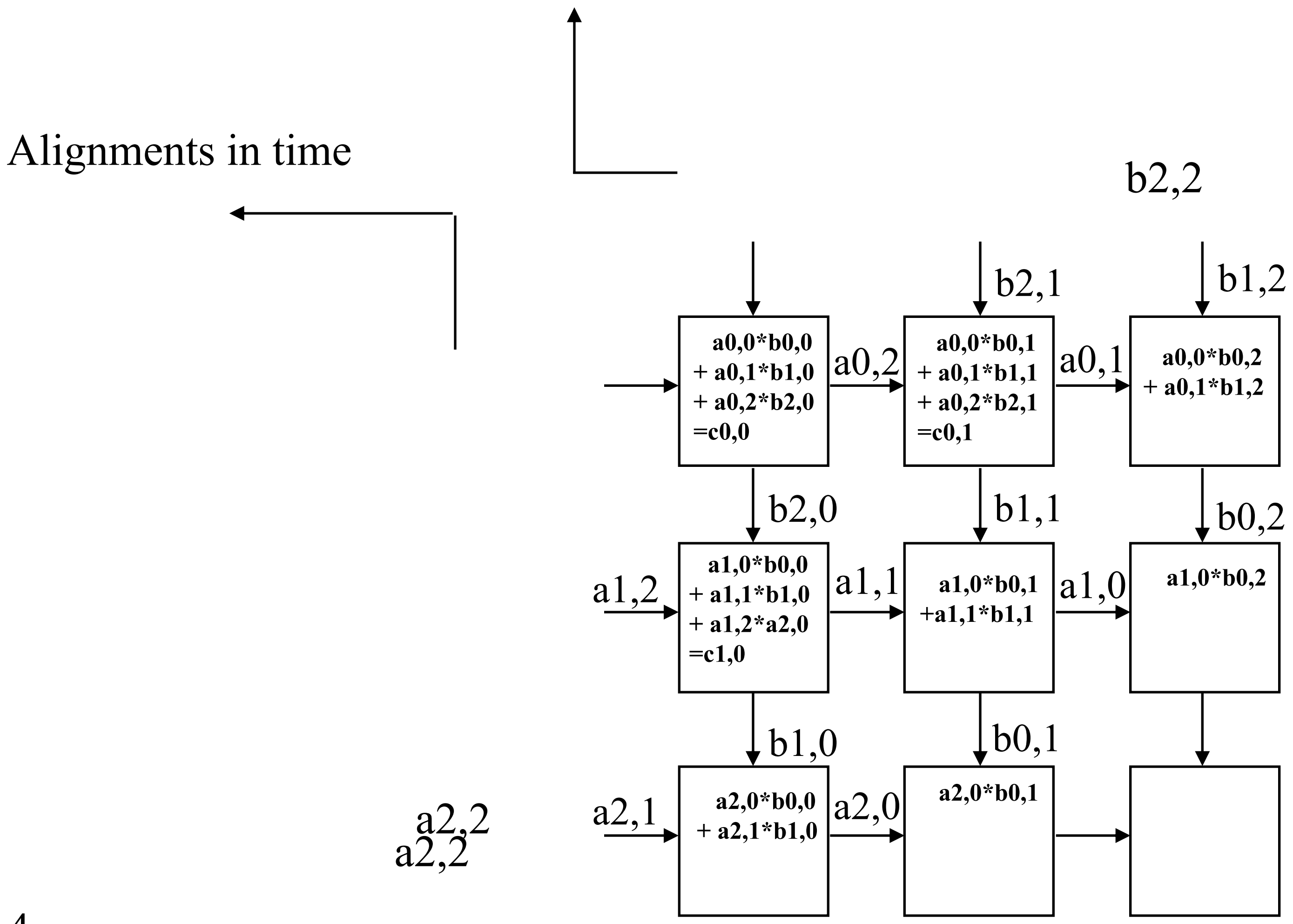
T = 2

2D Systolic Array (Example)



T = 3

2D Systolic Array (Example)



Alignments in time

b2,2

b2,1

b1,2

a0,0*b0,0
+ a0,1*b1,0
+ a0,2*b2,0
=c0,0

a0,2

a0,0*b0,1
+ a0,1*b1,1
+ a0,2*b2,1
=c0,1

a0,1

a0,0*b0,2
+ a0,1*b1,2

b2,0

b1,1

b0,2

a1,2

a1,0*b0,0
+ a1,1*b1,0
+ a1,2*a2,0
=c1,0

a1,1

a1,0*b0,1
+a1,1*b1,1

a1,0

a1,0*b0,2

b1,0

b0,1

a2,2

a2,2

a2,1

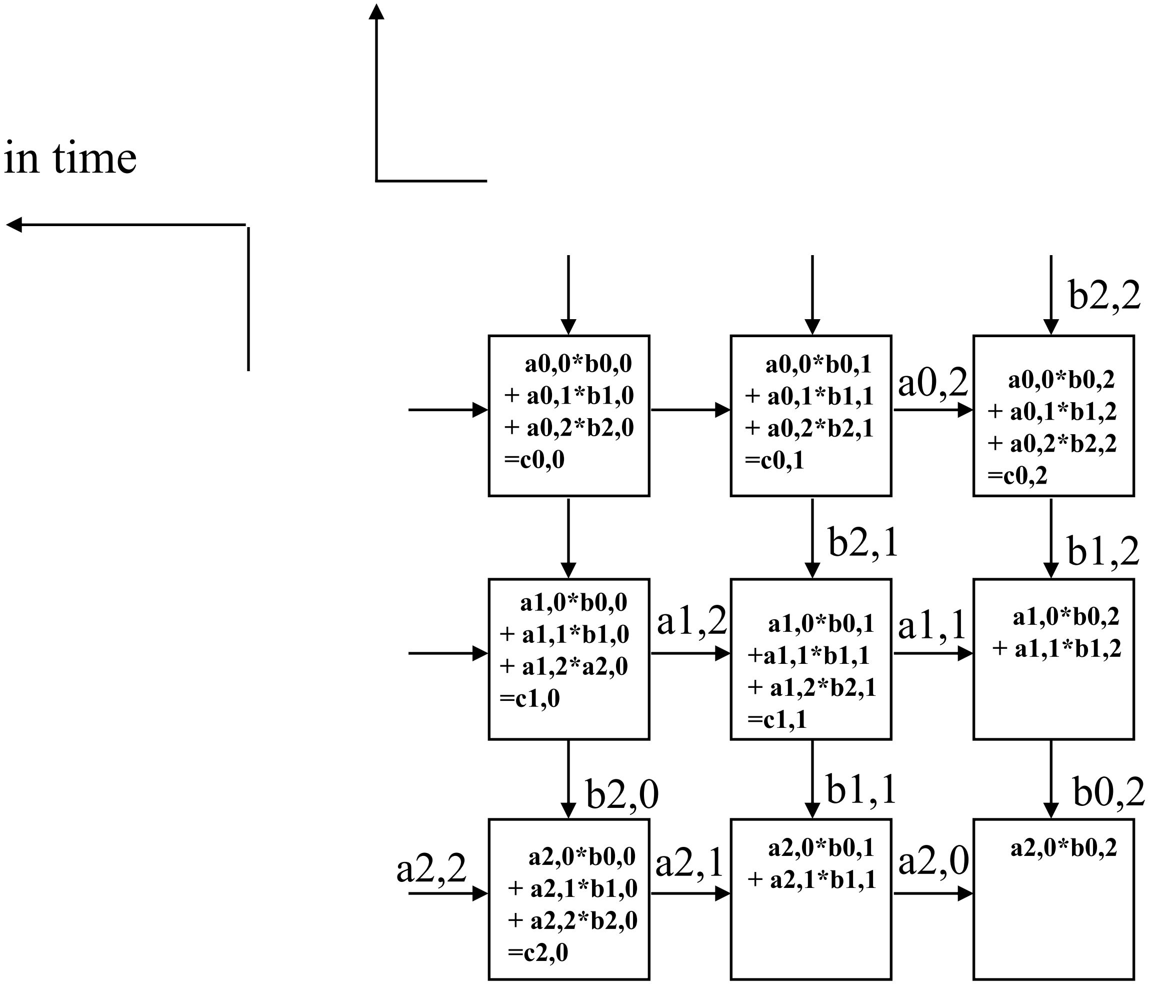
a2,0*b0,0
+ a2,1*b1,0

a2,0

a2,0*b0,1

T = 4

2D Systolic Array (Example)

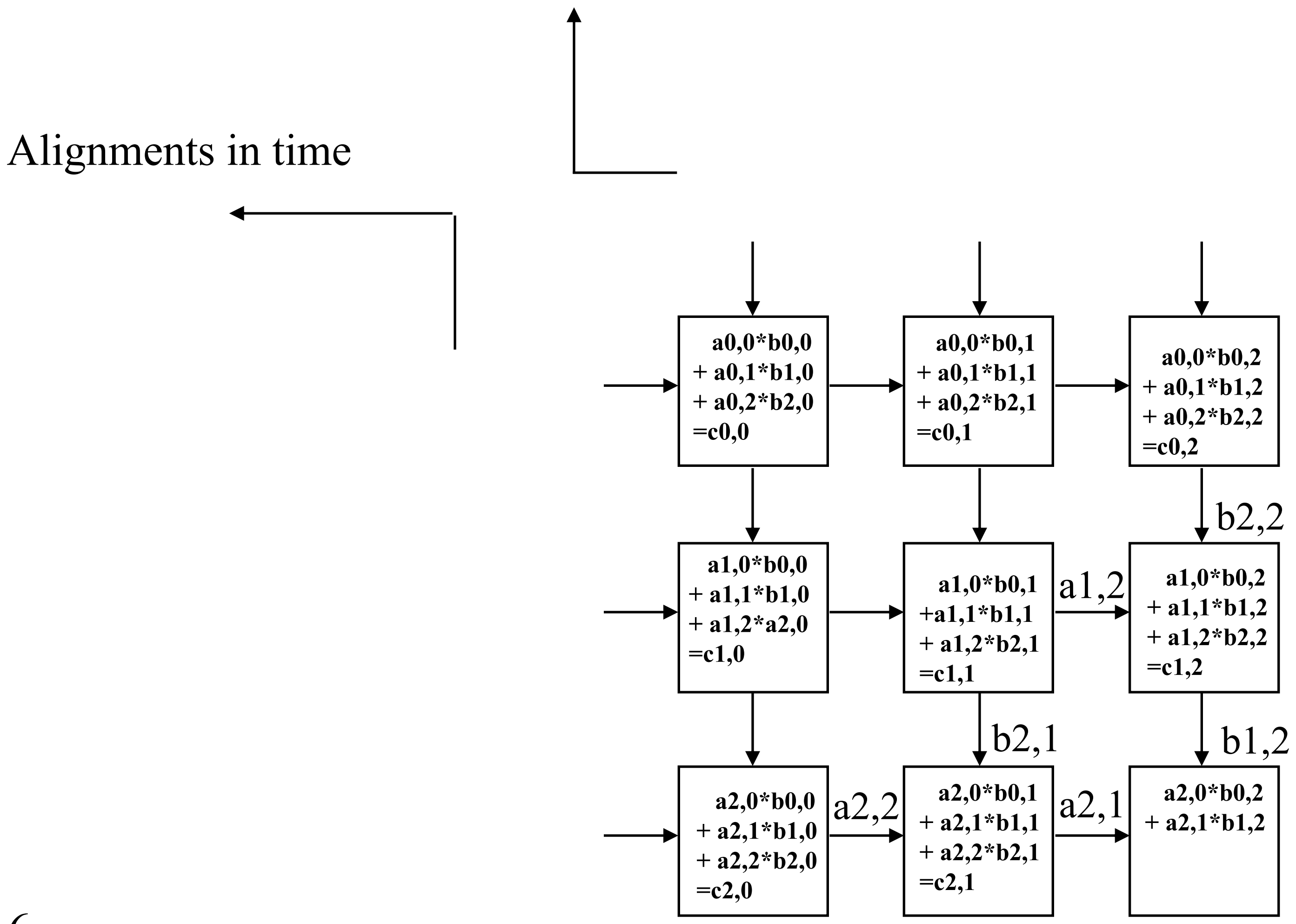


Alignments in time

a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0 =c0,0	a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1 =c0,1	a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2 =c0,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0 =c1,0	a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1 =c1,1	a1,0*b0,2 + a1,1*b1,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0 =c2,0	a2,0*b0,1 + a2,1*b1,1	a2,0*b0,2

T = 5

2D Systolic Array (Example)



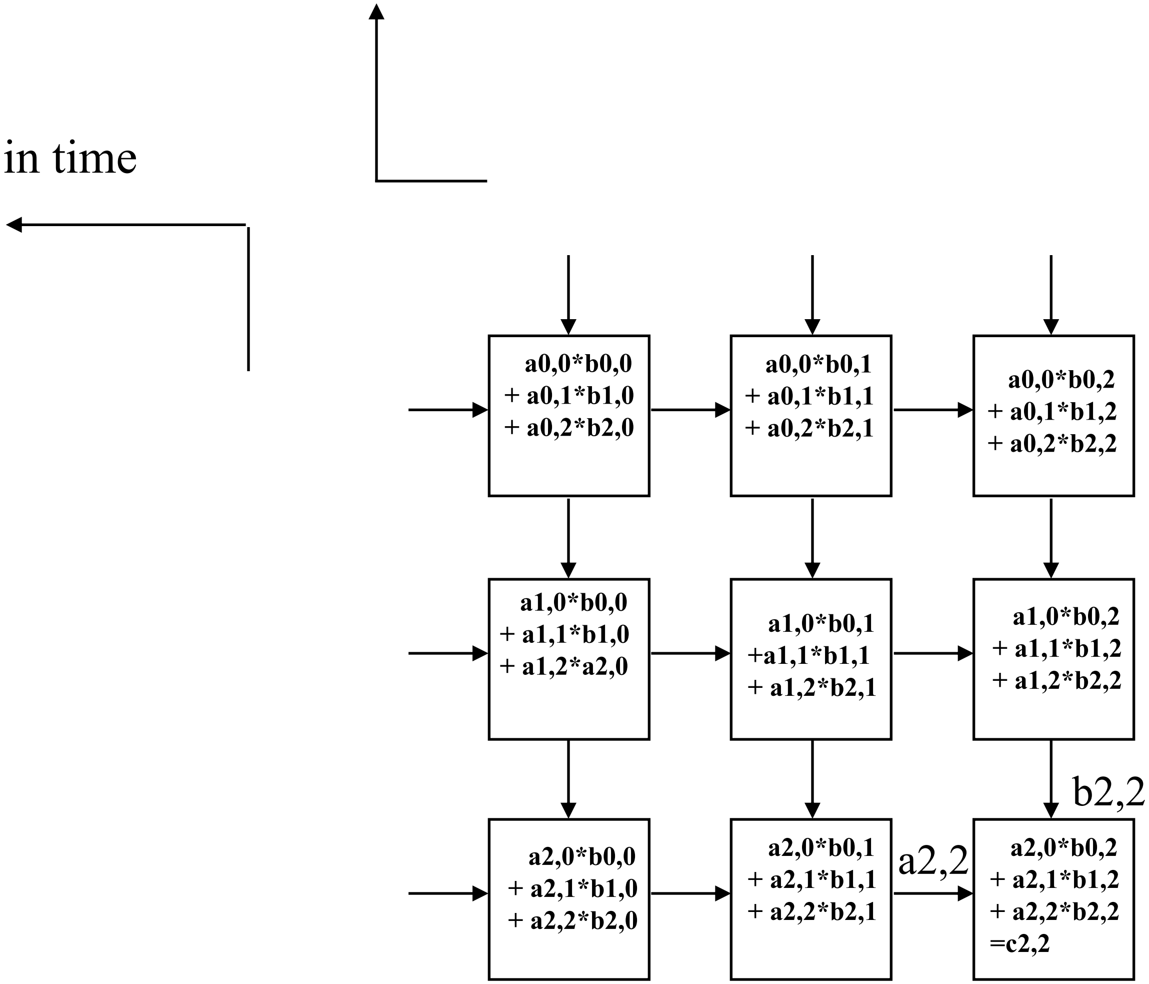
Alignments in time

a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0 =c0,0	a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1 =c0,1	a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2 =c0,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0 =c1,0	a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1 =c1,1	a1,0*b0,2 + a1,1*b1,2 + a1,2*b2,2 =c1,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0 =c2,0	a2,0*b0,1 + a2,1*b1,1 + a2,2*b2,1 =c2,1	a2,0*b0,2 + a2,1*b1,2

T = 6

2D Systolic Array (Example)

Alignments in time



a0,0*b0,0 + a0,1*b1,0 + a0,2*b2,0	a0,0*b0,1 + a0,1*b1,1 + a0,2*b2,1	a0,0*b0,2 + a0,1*b1,2 + a0,2*b2,2
a1,0*b0,0 + a1,1*b1,0 + a1,2*a2,0	a1,0*b0,1 +a1,1*b1,1 + a1,2*b2,1	a1,0*b0,2 + a1,1*b1,2 + a1,2*b2,2
a2,0*b0,0 + a2,1*b1,0 + a2,2*b2,0	a2,0*b0,1 + a2,1*b1,1 + a2,2*b2,1	a2,0*b0,2 + a2,1*b1,2 + a2,2*b2,2 =c2,2

Done

Remarks

- There can be various PE Designs
  - Which data stays inside the PE until the end of the computation?
    - Partial Sum (Output Stationary)
    - Input Feature (Input Stationary)
    - Wight (Weight Stationary)
  - Depending on the hyperparameters, the optimal PE design may differ
    - Find the best dataflow!

- Be aware that **the inputs are fed in a skewed manner!**
  - From the previous example, $a_{0,0}$, $a_{1,0}$, $a_{2,0}$ are not fed in the same cycle

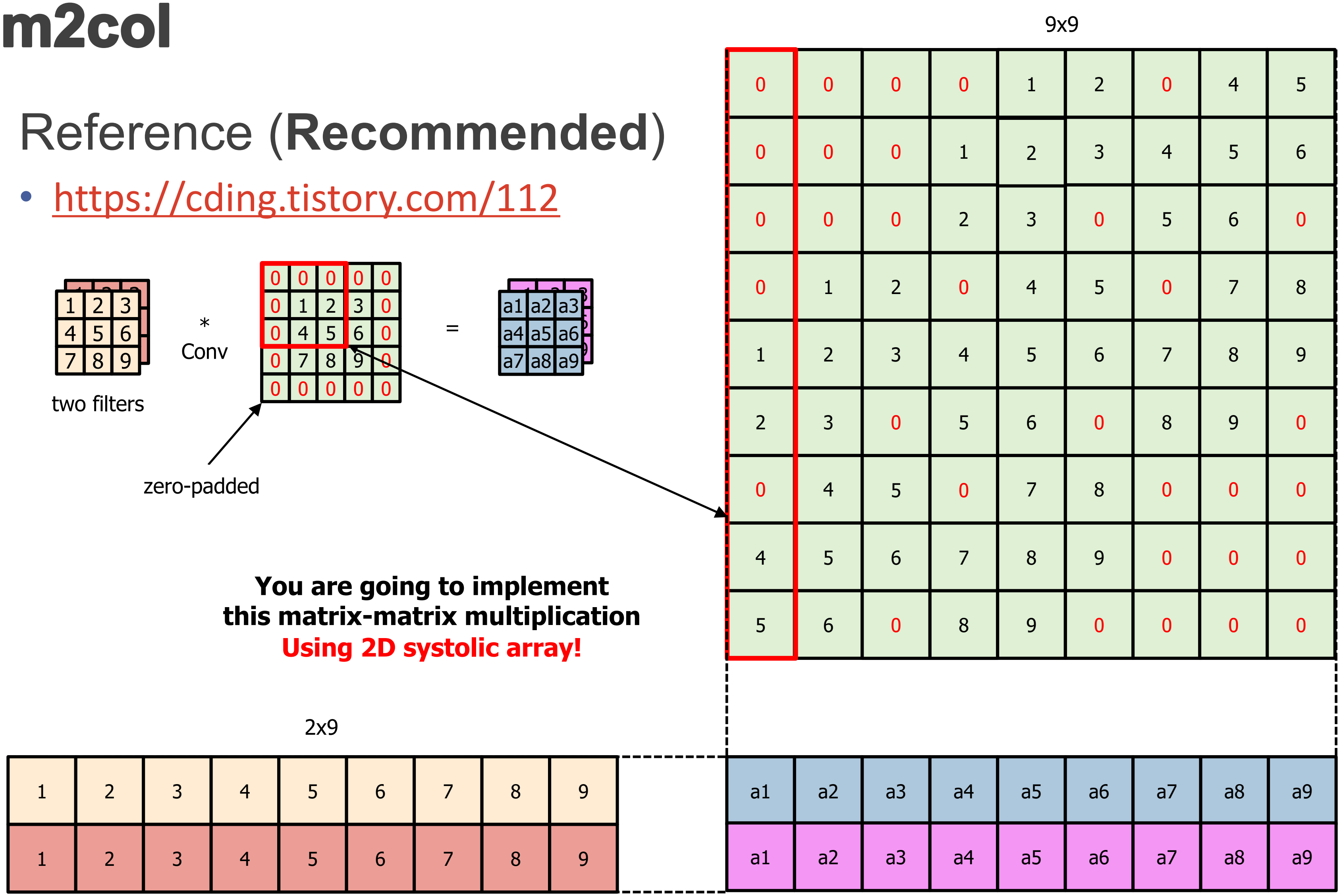
Im2col

- How can we convert **convolution operation** into **Matrix-Matrix multiplication**?
  - **Im2col Algorithm**

- We will implement our conv module with a 2D systolic array
  - Without Im2col algorithm, the systolic array will be useless

Im2col

- Reference (**Recommended**)
  - https://cding.tistory.com/112



You are going to implement this matrix-matrix multiplication
Using 2D systolic array!

9x9

0	0	0	0	1	2	0	4	5
0	0	0	1	2	3	4	5	6
0	0	0	2	3	0	5	6	0
0	1	2	0	4	5	0	7	8
1	2	3	4	5	6	7	8	9
2	3	0	5	6	0	8	9	0
0	4	5	0	7	8	0	0	0
4	5	6	7	8	9	0	0	0
5	6	0	8	9	0	0	0	0

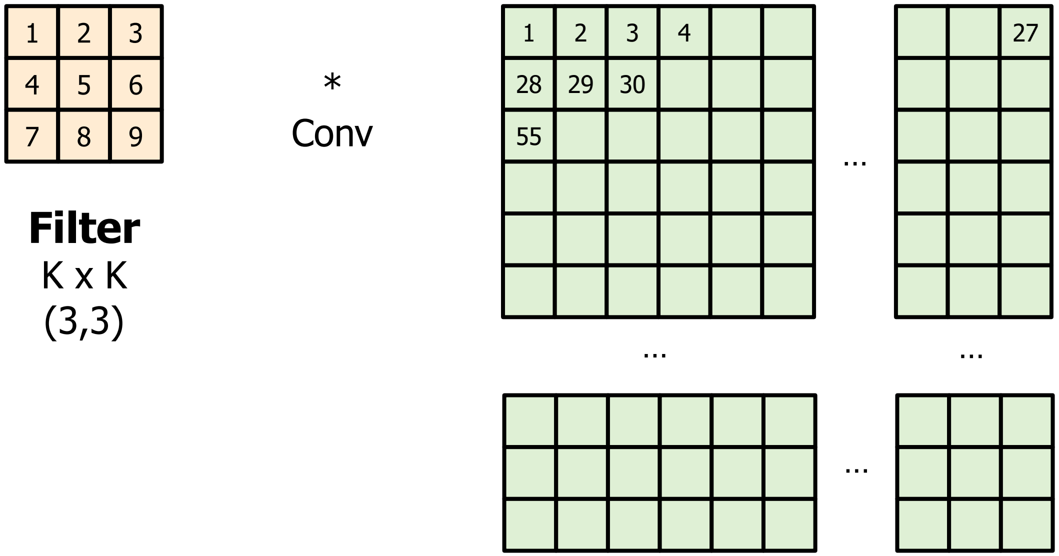
2x9

1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9

a1	a2	a3	a4	a5	a6	a7	a8	a9
a1	a2	a3	a4	a5	a6	a7	a8	a9

Im2col (Example)

- Simple 2D Convolution



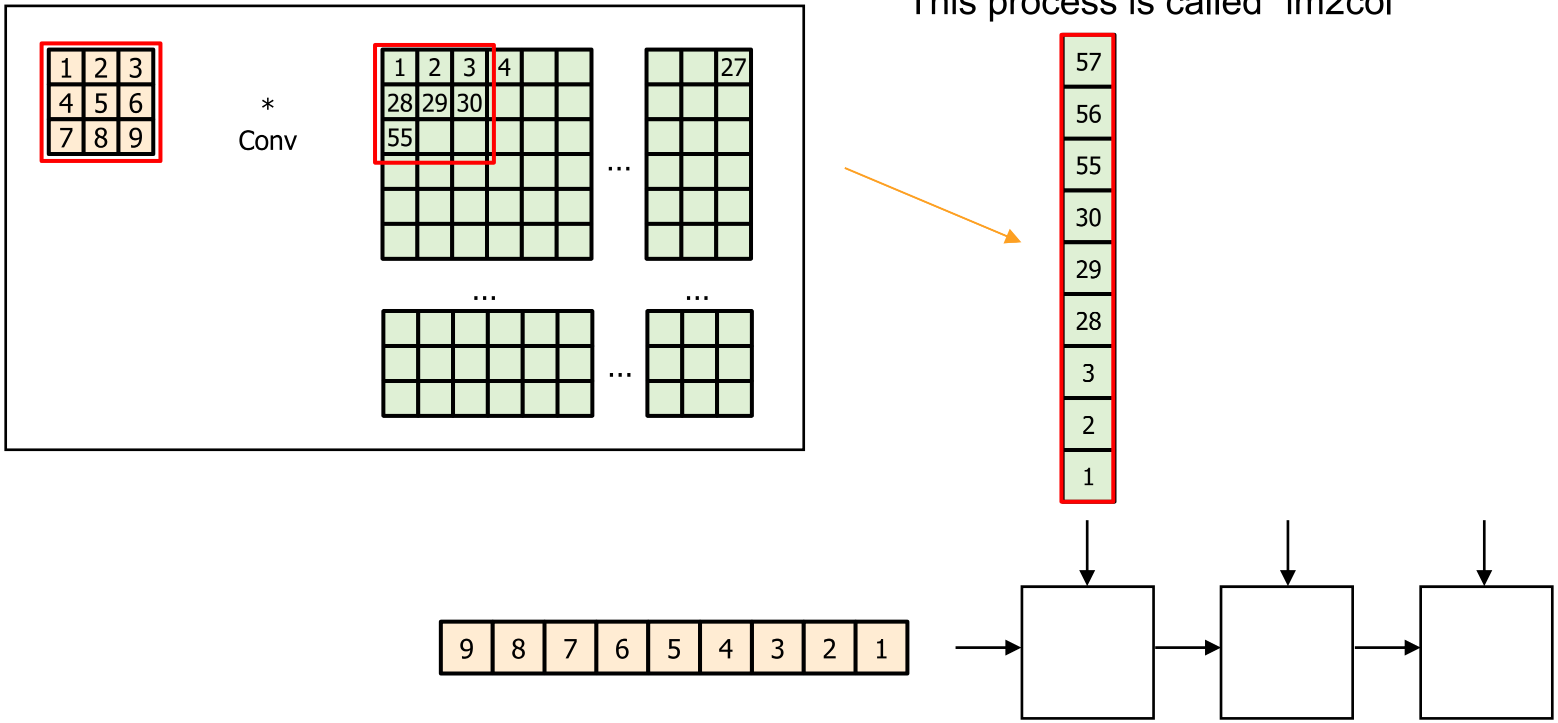
Filter
K x K
(3,3)

Image
H x W
(27,27)

Im2col (Example)

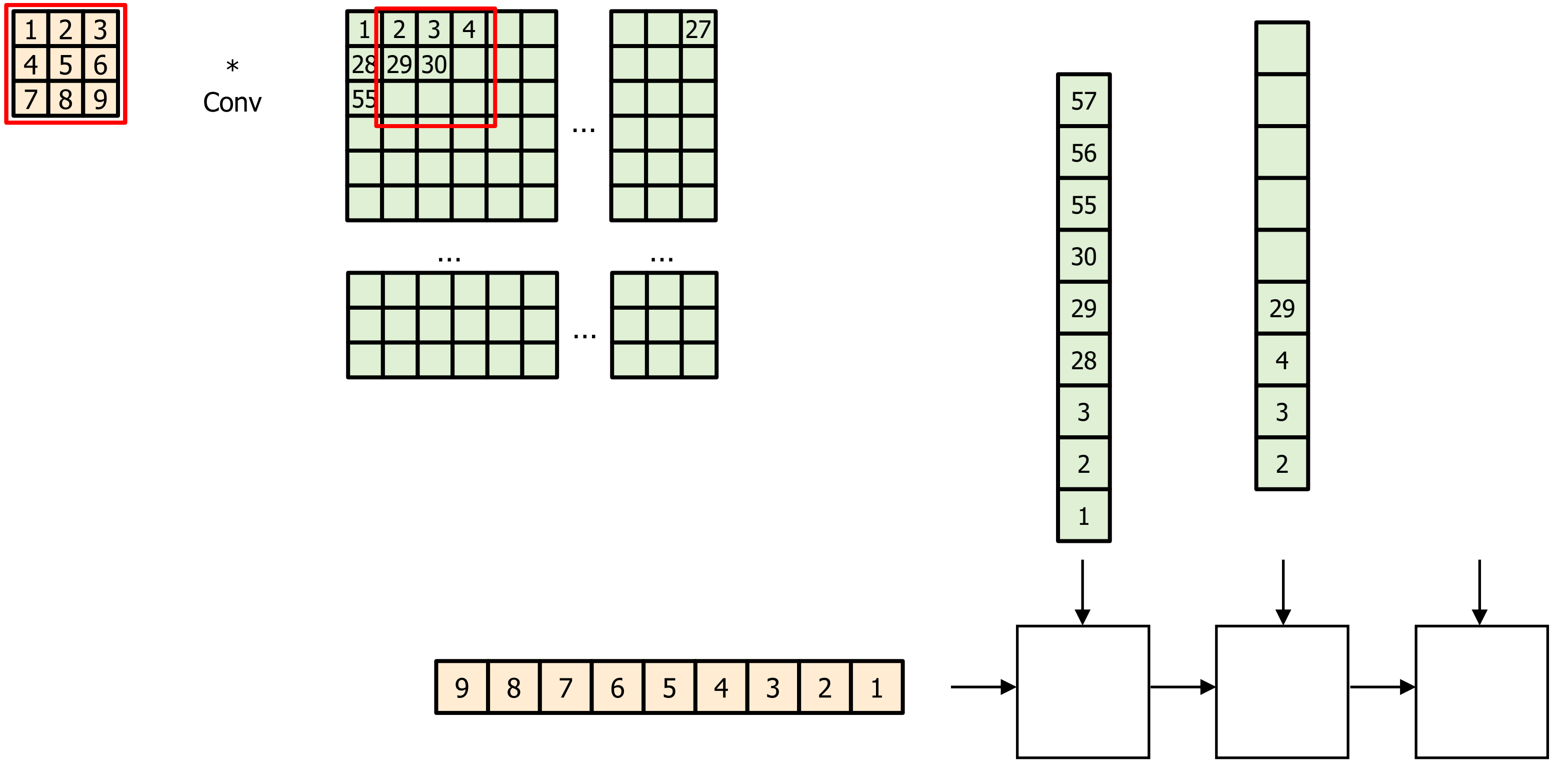
- Simple 2D Convolution
  - Convert a portion of 3x3 image into a column of transformed input feature matrix

This process is called “im2col”



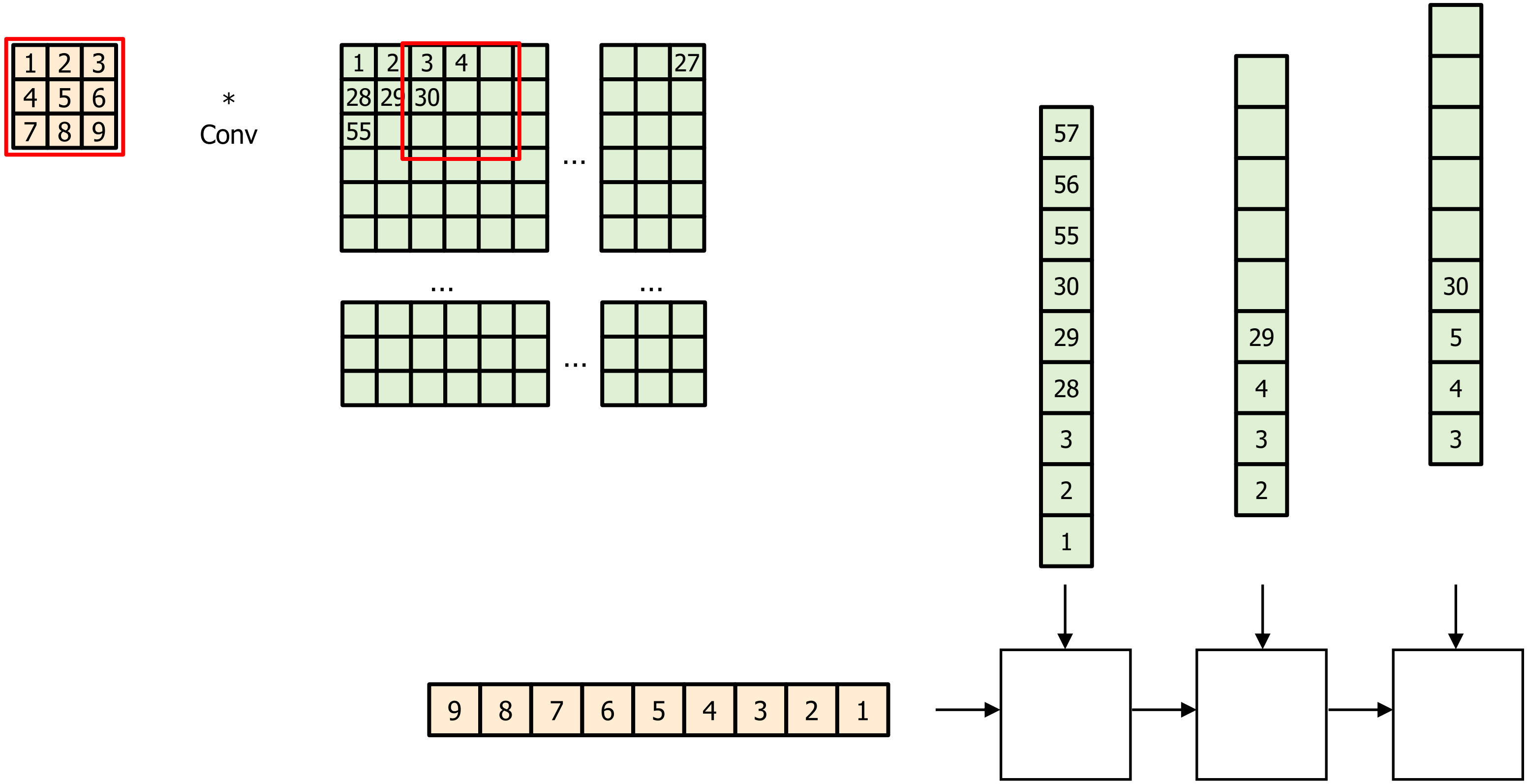
Im2col (Example)

- Simple 2D Convolution
  - Do the same thing for each sliding window



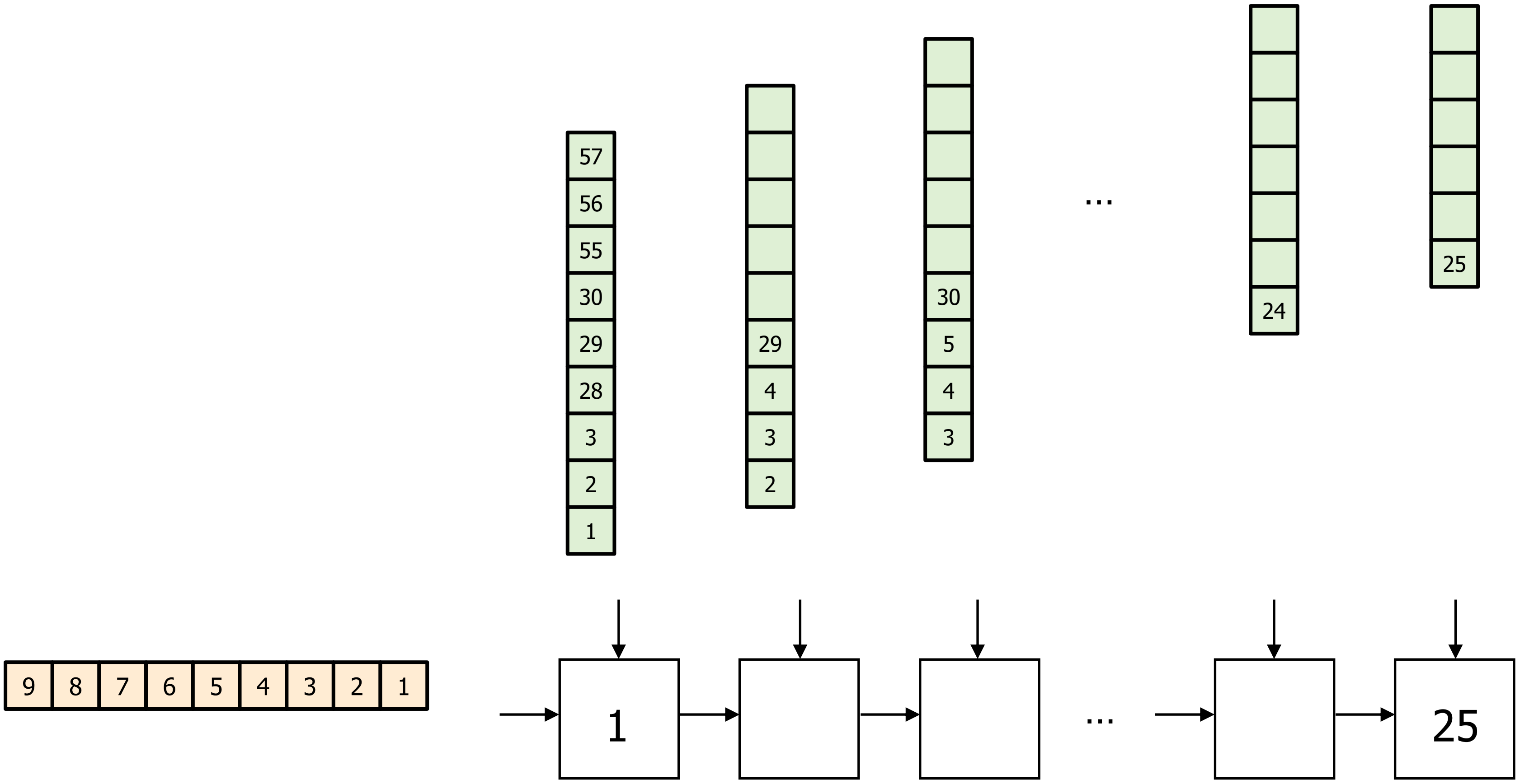
Im2col (Example)

- Simple 2D Convolution
  - Do the same thing for each sliding window



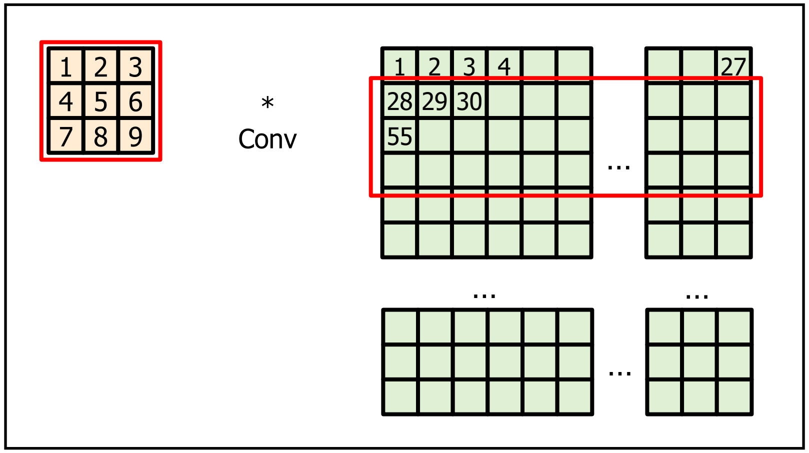
Im2col (Example)

- Simple 2D Convolution
  - After 34 cycles, the result of the 1st row is made



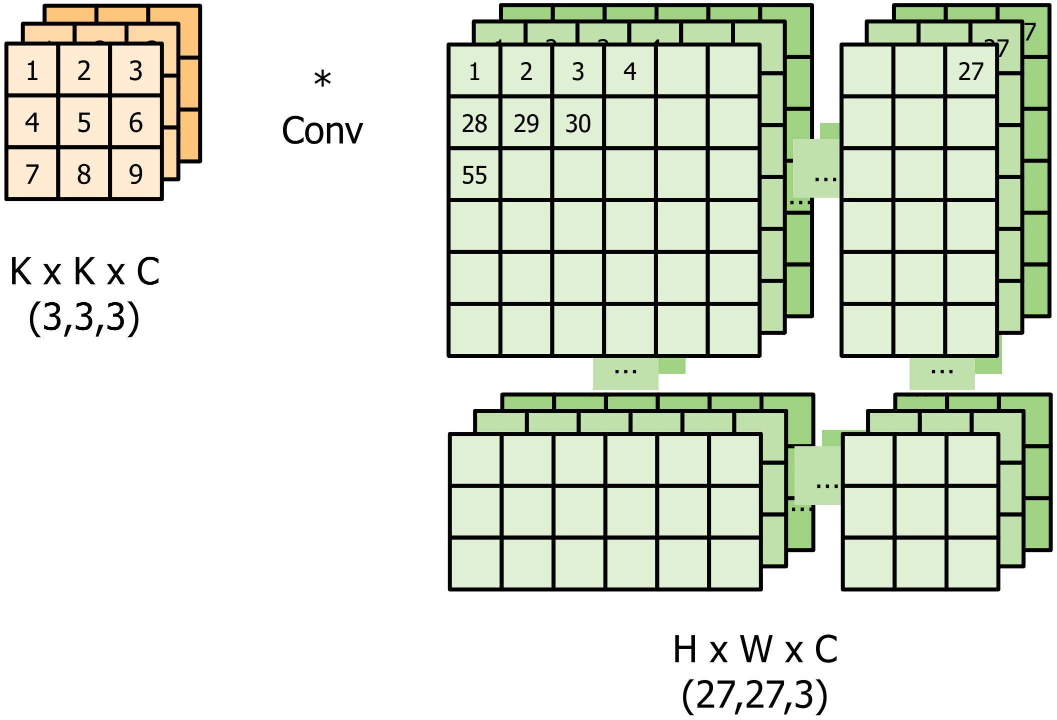
Im2col (Example)

- Simple 2D Convolution
  - Then, do the same things for the 2nd row



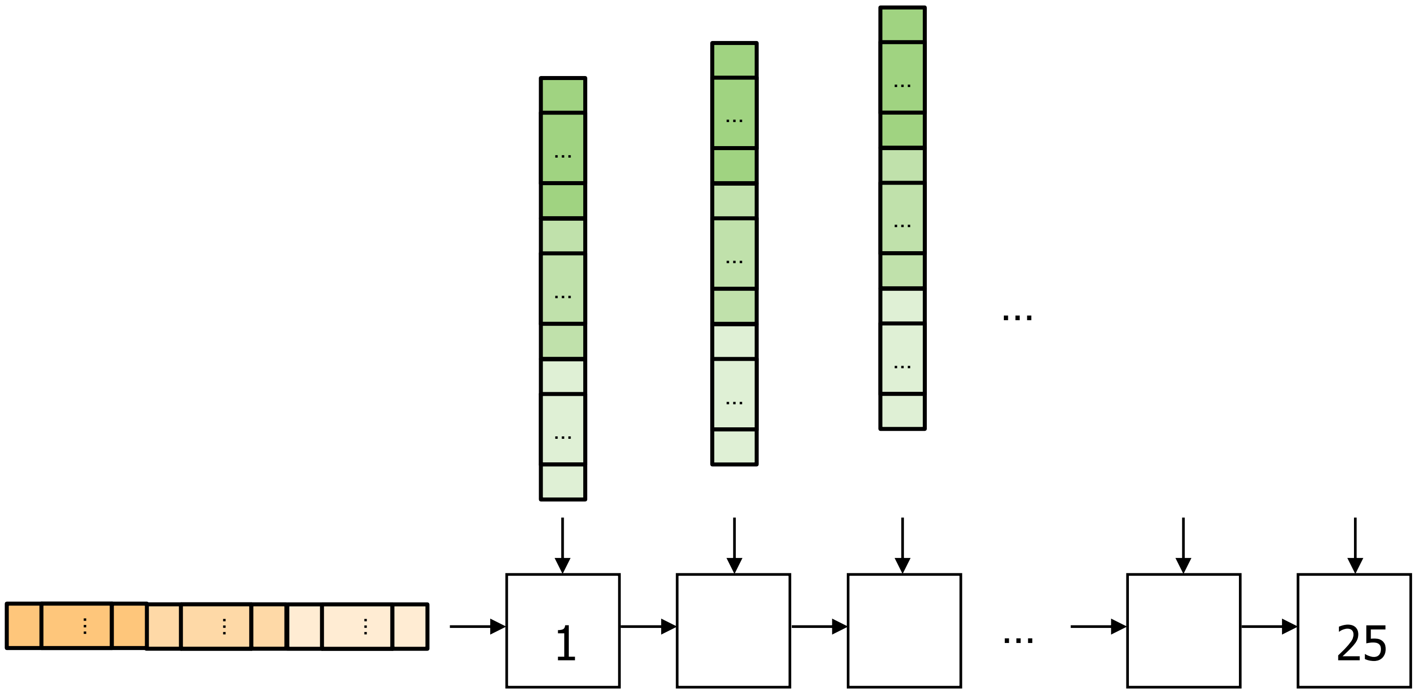
More Complex Example?

- **Convolution with multi-channel**



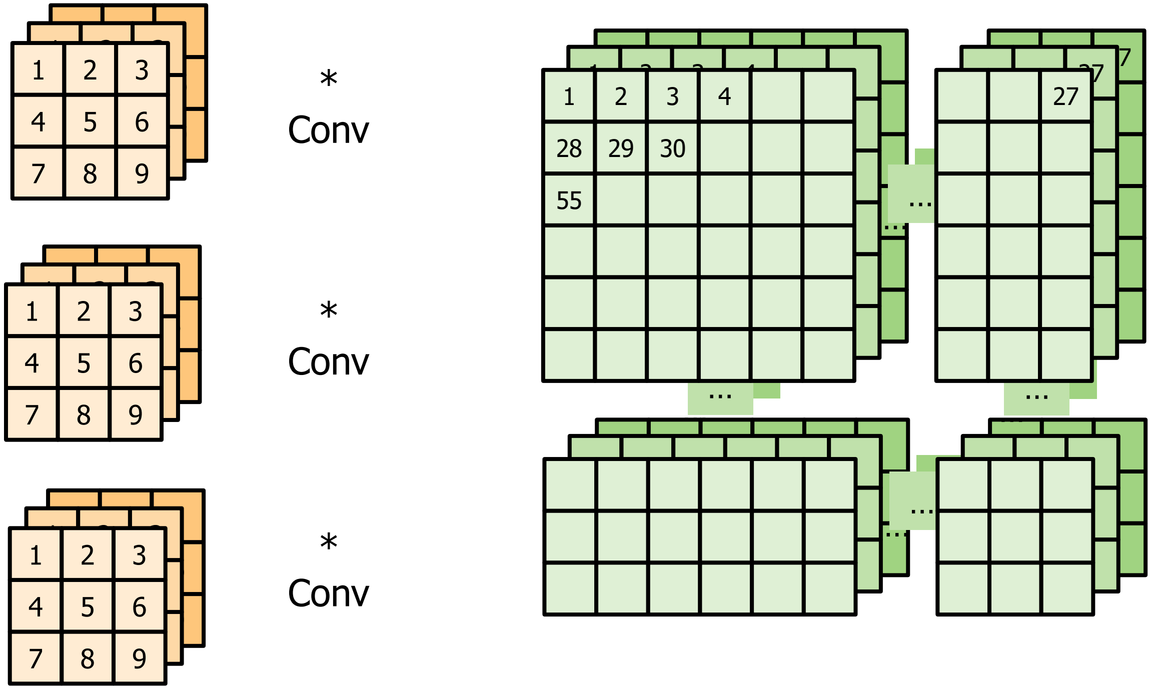
More Complex Example?

- Convolution with multi-channel
  - After applying Im2col on each channel, you will obtain multiple vectors
  - **Append these vectors alongside the column of the transformed input feature matrix**



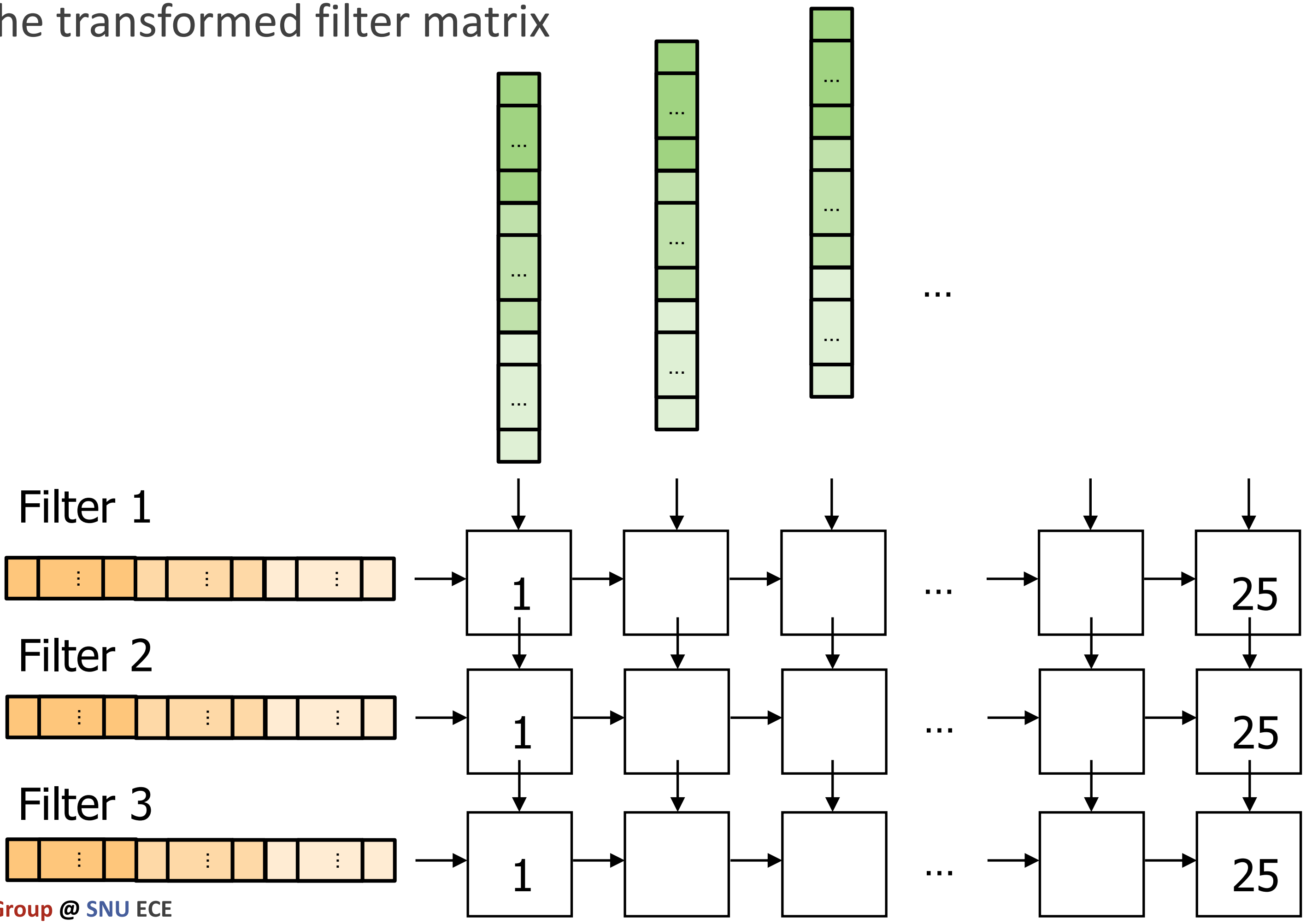
Even More Complex Example?

- **Convolution with multi-filter**



Even More Complex Example?

- Convolution with multi-filter
  - Nothing different for the input feature matrix
  - Extend the transformed filter matrix



Remarks

- Whenever you apply Im2col,
  take special care of **padded input features**
  - Padding will induce some corner cases

- 2D Systolic Array may not be large enough
  to handle the transformed matrix multiplication at once
  - You should tile the transformed matrix
  - **Im2col with tiling**... => Think carefully

- Resource Constraints
  - BRAM and LUTs are precious resources
  - Start with a small systolic array, first test functionality,
    then scale up for the better performance
    - Parameterizing your module will help

Other Optimizations

- You can always refer to the latest CNN accelerators
  - e.g., Eyriss (MIT), TPU (Google), etc

- Find the optimal design point,
  regarding the resource constraints

Submission Guideline

Submission Guideline

- Submission Deadline
  - **TBA**
  - **We will not accept any delayed submission**

- Grading Policy
  - Functionality
    - You will demonstrate your accelerator
  - Performance
    - We will measure it with a given performance counter (*clk_counter.v*)

- **DO NOT COPY**

Submission Guideline

- Submit to GitHub
  - Your entire project, including…
    - Python codes that you have revised (e.g. *scale_uart.py*)
    - Verilog codes (**.v*)
    - Bitstream (**.bit*)
    - Constraint files (**.xdc*)

  - Push everything to **the "main" branch** of your GitHub repository
  - Your final commit before the deadline will be assessed

Submission Guideline

- Submit to eTL
  - One report for each team, including…
    - A detailed explanation of your Verilog codes
      - Even if you haven’t finished the project, you should explain what you have done so far
      - e.g., We implemented the FC module and it passes the simulation & single_layer_test.ipynb
    - Block Diagram of your overall design (including FSMs)
    - Contribution of each member
  - One demo video for each team
    - Video of running the ipynb files (e.g. **.mp4*)
    - Less than 500MB
  - **One member should upload all the deliverables on the eTL**

- We will double-check your results with the source code in the GitHub

Miscellaneous

Tips

- Before you start anything, make sure that you have plans
  - Go over all the materials that we have provided
  - You should know…
    - How does the host CPU communicate with the FPGA board through python APIs
    - How to read/write on memory-mapped FPGA registers from the host CPU
    - What computation should be performed (exact specification of our CNN model)
  - Design your hardware algorithm
    - **We recommend you draw a lot of block diagrams!**
    - In what order should we read data from the FPGA Memory?
    - What should be the BRAM design to provide enough bandwidth to the compute unit?
    - …

Tips

- Debugging RTL codes is difficult and time-consuming
  - Synthesis/Implementation stages take around an hour on the laptop
  - **Follow the Verilog coding conventions**
    - Try to write easy-to-understand codes,
      so that you and your teammates can crosscheck the implementation
  - **Add debug registers through the APB protocol**
    - This will help you examine what is happening inside your hardware
  - **Revise the testbench**
    - **The given testbench is imperfect**
      - □ **Does not strictly follow the control flow of the python codes**
    - You should write your own testbench to test the corner cases
      - □ Corner cases may differ depending on your design

Python Codes

- The host CPU and the FPGA board will communicate via UART
  - Some useful APIs are defined in the ***scale_uart.py***
  - You can freely revise the codes to debug or optimize your design

- Python Environment
  - Refer to the materials from Lab 8
    - Use **Conda** to install packages
    - Package dependencies: jupyter, pyserial, numpy, matplotlib, **torch**, glob2
    - $ **conda install pip**
    - $ **pip install -r requirements.txt**

Python Codes

- We also provide python scripts to help you understand how our CNN model architecture looks like

- Refer to the ***cifar_pytorch.ipynb*** for more information
  - Also, take a look at ***layers_cifar10.py, setup_cifar10.py*** and ***bit_operation.py***