

# HW#5 Domain-Specific Accelerator



Chun-Jen Tsai  
NYCU  
12/06/2024

# Homework Goal

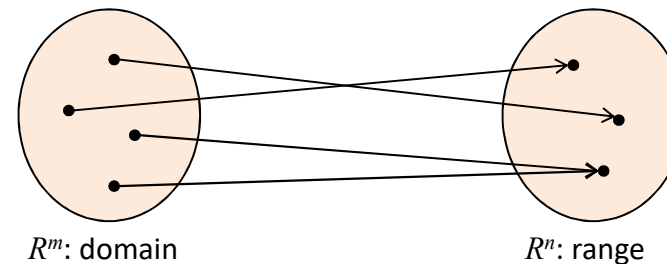
---

- ❑ In this homework, you must integrate a domain-specific accelerator (DSA) to Aquila to improve the speed of a CNN neural network
- ❑ Your tasks:
  - Add a **vector** floating-point HW IP by Xilinx into the Aquila SoC
  - Use the IP to accelerate the computing speed of a neural network application
- ❑ You should upload report & code to E3 by 1/6, 17:00.

# Neural Network as Computers

- All computing systems are used to compute functions:

$$f: R^m \rightarrow R^n$$



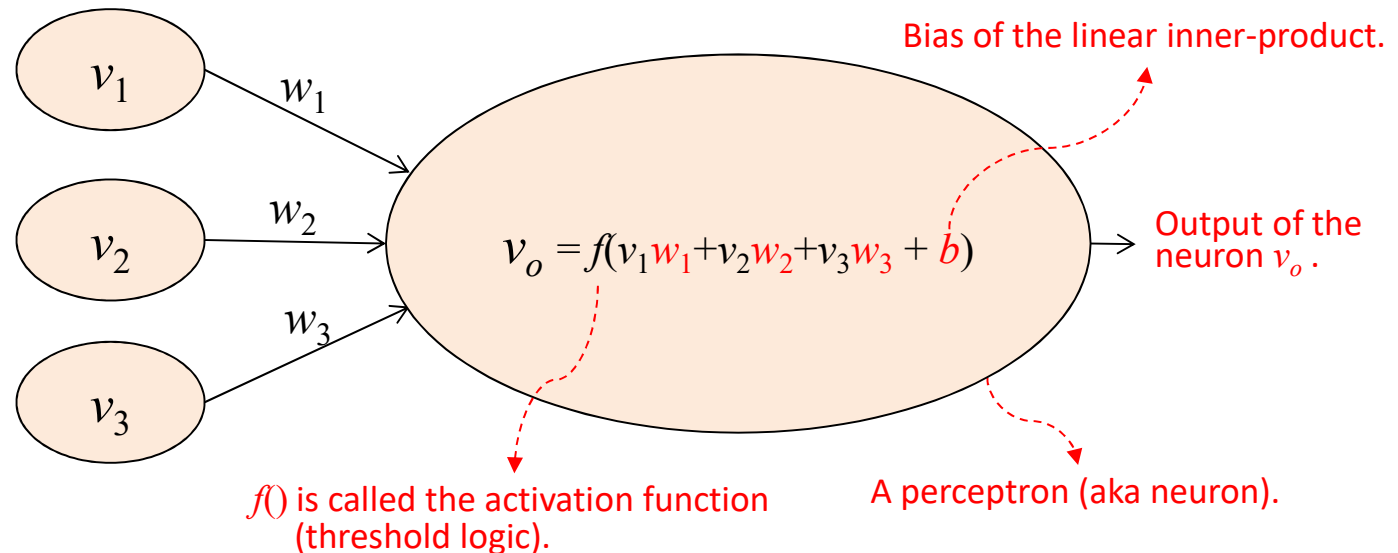
- In 1957, Kolmogorov proved that any continuous function can be decomposed into  $(2m+1)+n$  different  $R^m \rightarrow R$  linear functions

---

† A. N. Kolmogorov, "On the representation of continuous function of many variables by superpositions of continuous functions of one variable and addition," *Doklady Akademii Nauk USSR*, 114(5):953-956, 1957.

# Basic Neural Network Components

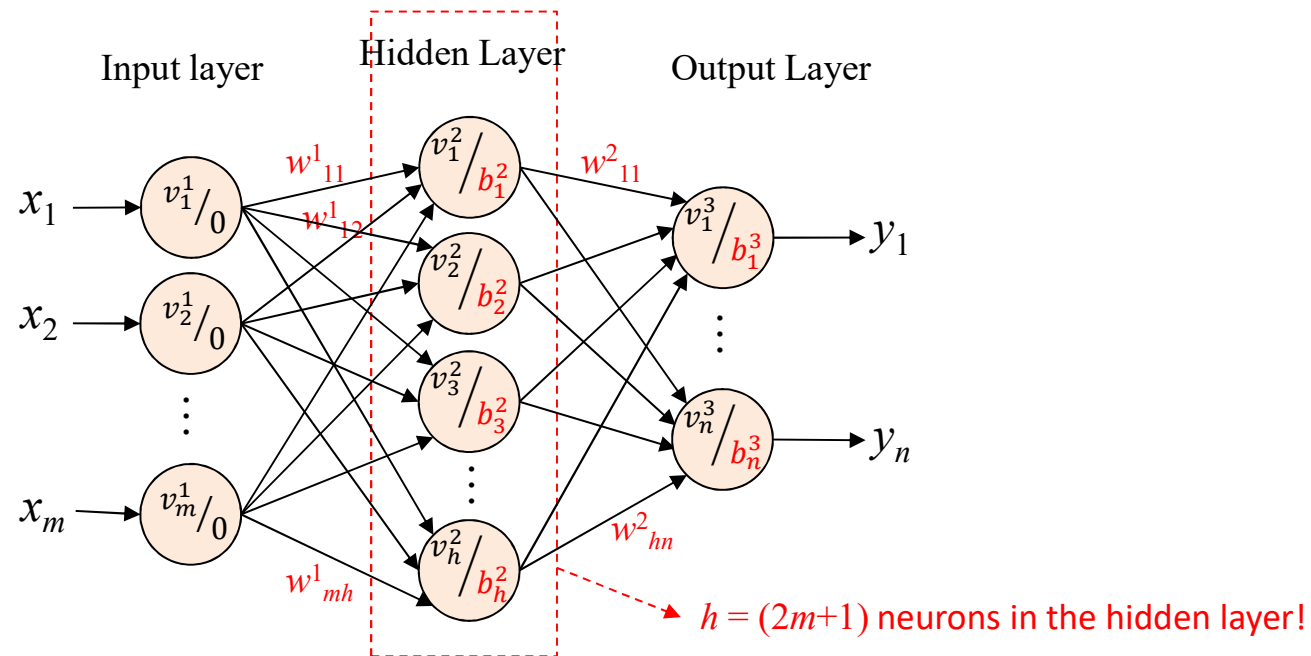
- ❑ W. McCulloch and W. Pitts proposed a computing element, threshold logic, for Artificial Neural Network (ANN) in 1943:



† W. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. **5** (4), 1943, pp. 115–133.

# A Universal Neural Network

- By A. N. Kolmogorov, a continuous function  $f: R^m \rightarrow R^n$  can be computed using 3 layers of perceptrons

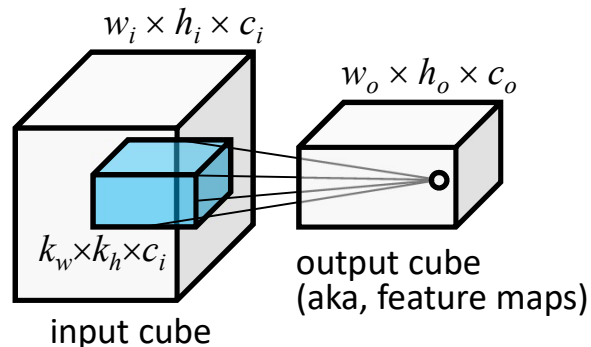


- The “program” of a neural network is in the weights,  $\{w_{ji}^k, b_i^k\}$
- This architecture is called Multi-Level Perceptrons (MLP)

# Different Neural Network Layer Types

## ❑ Convolutional Layer

- Computes 3D dot product btw. the input data & the kernel
- Each neuron in the output cube of **the same channel** shares the same kernel



## ❑ Avg-Pooling Layer

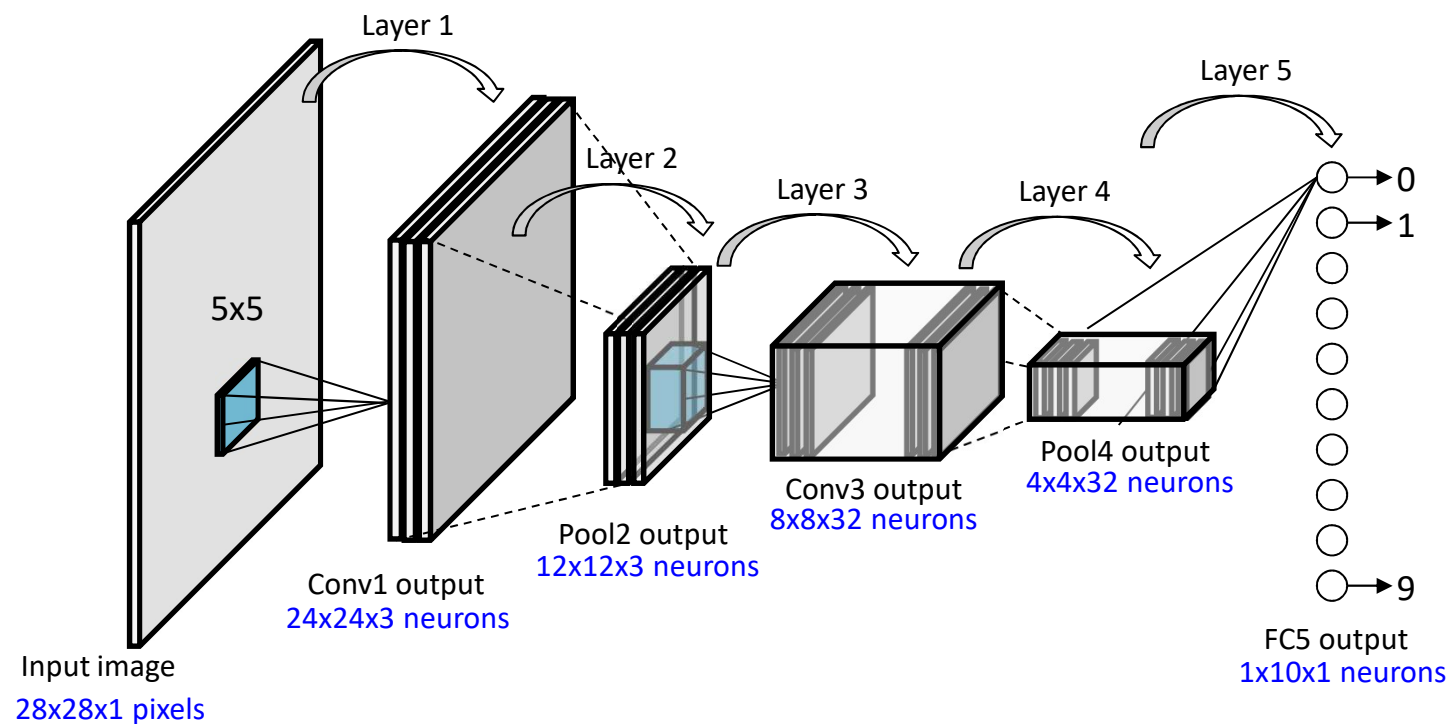
- Reduce the width & height of the input feature maps by averaging the neuron values in a small  $m \times n$  rectangles of the same channels

## ❑ Fully-Connected Layer

- As in the MLP

# Convolutional Neural Network (CNN)

- The CNN model used in this HW



# Hand-Written Character Recognition

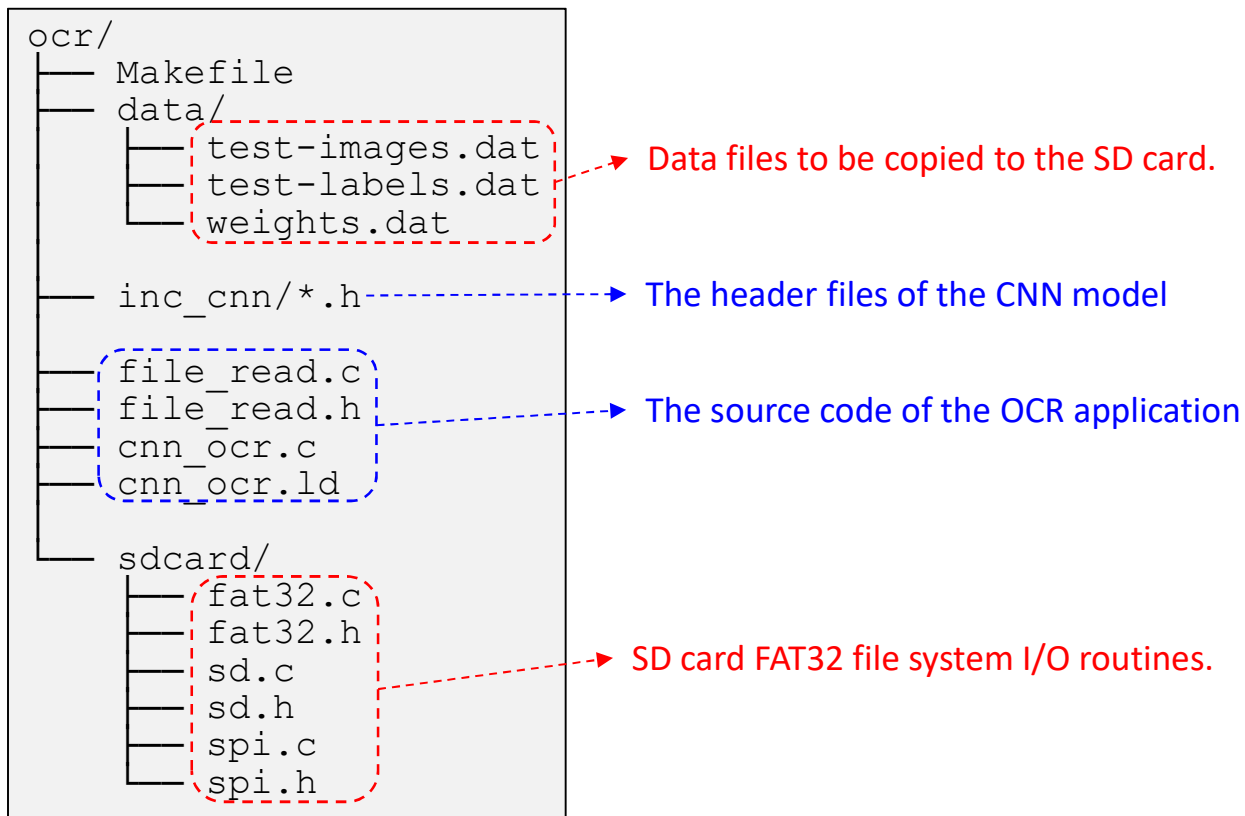
- ❑ In this HW, we use a 5-layer CNN for hand-written character recognition
- ❑ The neural network weights is trained using the MNIST dataset of 60,000 images:
  - Each image has 28×28 pixels





# The HW/SW for HW#5

- ❑ You should download from E3 the CNN program, `cnn_ocr.tgz`.
- ❑ The source tree of the `cnn_ocr` program:



# The HW Workspace for HW#5

- ❑ We have added an AXI SPIO controller for SD card:

IPs inserted from the Vivado IP Catalog

**Project Summary**

Overview | Dashboard

**Settings** Edit

Project name: aquila\_mpd  
Project location: R:/temp/aquila\_dram/aquila\_mpd  
Product family: Artix-7  
Project part: xc7a100tcsq324-1  
Top module name: soc\_top  
Target language: Verilog  
Simulator language: Mixed

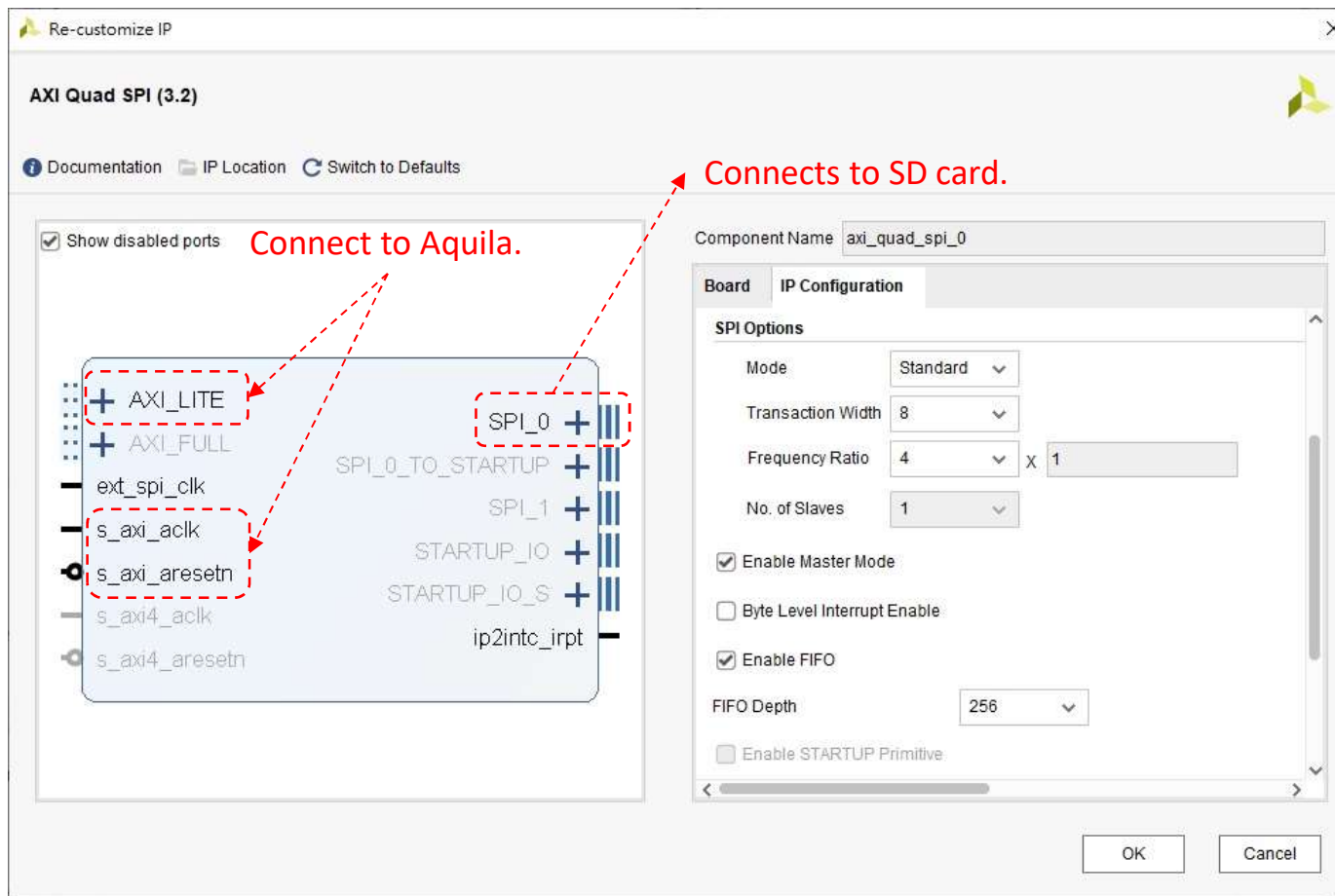
Synthesis	Implementation	Sum
Status: <span style="color: green;">✔</span> Complete	Status: <span style="color: green;">✔</span> Compl	
Messages: <span style="color: yellow;">!</span> 935 warnings	Messages: <span style="color: yellow;">!</span> 20 war	
Active run: synth_1	Active run: impl_1	
Part: xc7a100tcsq324-1	Part: xc7a10	

**Design Runs**

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology
✓ synth_1 (active)	constrs_1	synth_design Complete!									
✓ impl_1	constrs_1	write_bitstream Complete!	1.274	0.000	0.009	0.000	10.908	0.000	0.914	0	22 Warn

# AXI Quad SPI Controller

- ❑ Double-click the IP opens the parameter box:



# Instantiation of the IP in Aquila

## ❑ Instantiation of the SPI Controller:

```
// -----  
// SPI controller  
// -----  
// This controller connects to the PMOD microSD module in  
// the JD connector of the Arty A7-100T.  
//  
axi_quad_spi_0 SD_Card_Controller(  
  
    // Interface ports to the Aquila SoC.  
    .s_axi_aclk(clk),  
    .s_axi_aresetn(~rst),  
    .s_axi_awaddr(axi_awaddr),  
    .s_axi_awvalid(axi_awvalid), // master signals write addr/ctrl valid.  
    .s_axi_awready(axi_awready), // slave ready to fetch write address.  
    .s_axi_wdata(axi_wdata), // write data to the slave.  
    .s_axi_wstrb(axi_wstrb), // byte select signal for write operation.  
    .s_axi_wvalid(axi_wvalid), // master signals write data is valid.  
    .s_axi_wready(axi_wready), // slave ready to accept the write data.  
    .s_axi_araddr(axi_araddr),  
    .s_axi_arready(axi_arready), // slave ready to fetch read address.  
    .s_axi_arvalid(axi_arvalid), // master signals read addr/ctrl valid.  
    .s_axi_bready(axi_bready), // master is ready to accept the response.  
    .s_axi_bresp(axi_bresp), // reponse code from the slave.  
    .s_axi_bvalid(axi_bvalid), // slave has sent the respond signal.  
    .s_axi_rdata(axi_rdata), // read data from the slave.  
    .s_axi_rready(axi_rready), // master is ready to accept the read data.  
    .s_axi_rresp(axi_rresp), // slave sent read response.  
    .s_axi_rvalid(axi_rvalid), // slave signals read data ready.  
  
    // Interface ports to the SD Card.  
    .ext_spi_clk(clk),  
    .io0_o(spi_mosi),  
    .io1_i(spi_miso),  
    .sck_o(spi_sck),  
    .ss_o(spi_ss)  
);
```



# The Aquila Device Interface

- ❑ The Aquila core uses a simple memory-mapped I/O interface to talk to the external devices
  - A bridge is required for Aquila to talk to the AXI SPI controller

```
aquila_top Aquila_SoC
(
    .clk_i(clk), .rst_i(rst), .base_addr_i(32'b0),

    // External instruction memory ports.
    .M_IMEM_strobe_o(IMEM_strobe),
    .M_IMEM_data_i(IMEM_data),

    // External data memory ports.
    .M_DMEM_strobe_o(DMEM_strobe),
    .M_DMEM_data_i(DMEM_rd_data),

    // I/O device ports.
    .M_DEVICE_strobe_o(dev_strobe),      // Issue read/write requests.
    .M_DEVICE_addr_o(dev_addr),          // Target device address.
    .M_DEVICE_rw_o(dev_we),              // Read or write?
    .M_DEVICE_byte_enable_o(dev_be),     // Byte-select signal.
    .M_DEVICE_data_o(dev_din),           // Data input to the device.
    .M_DEVICE_data_ready_i(dev_ready),   // Is device ready?
    .M_DEVICE_data_i(dev_dout)           // Data output from the device.
);
```

# Bridging the IP Interface

---

- ❑ Most IPs in the Xilinx IP Catalog use the AXI bus interfaces to communicate with other IPs:
  - AXI Full: enable both burst and single-beat data transfer
  - AXI Lite: enable single-beat data transfer
  - AXI Stream: enable burst-only data transfer
- ❑ We must convert the Aquila interface bus signals to the AXI bus signals for IP integration
  - The module `core2axi_if.v` is used for Aquila to connect to any IP that supports AXI Lite bus interface.

# Running the CNN\_OCR Program

---

1. First, copy the data files to the SD card  
→ make sure the SD card is formatted as a **FAT32** disk
2. Insert the SD card into the SD card pmod module
3. Insert the pmod module into the socket JD on Arty
4. Then, build `cnn_ocr.elf` by typing “make”
5. Synthesize the Aquila SoC and configure the FPGA
6. Load and run `cnn_ocr.elf`

# Output of the CNN\_OCR Program

- ❑ CNN\_OCR make inferences on a sequence of images:

```
=====
Copyright (c) 2019-2024, EISL@NYCU, Hsinchu, Taiwan.
The Aquila SoC is ready.
Waiting for an ELF file to be sent from the UART ...

Program entry point at 0x80003A18, size = 0xE65C.
-----

(1) Reading the test images, labels, and neural weights.
It took 529 msec to read files from the SD card.

(2) Perform the hand-written digits recognition test.
Here, we use a 5-layer CNN neural network model.
Begin computing ... tested 20 images. The accuracy is 95.00%

It took 21102 msec to perform the test.

-----
Program exit with a status code 0
Press <reset> on the FPGA board to reboot the cpu ...
```



# Key Hotspot of the Program

---

- ❑ There are two functions that are good candidates of hardware acceleration:
  1. `conv_3d()` in `convolution_layer.h`
  2. `fully_connected_layer_forward_propagation()` in `fully_connected_layer.h`
- ❑ For the HW:
  - The second function computes 1D dot-product is much easier to accelerate, but give less speedup
  - The first function, which computes 3D dot-product, is the most time consuming hotspot, but harder to turn into hardware
- ❑ Both circuits can use the 1D dot-product IP from Xilinx

# Inner-Product IP in Xilinx IP Catalog

The screenshot displays the Xilinx Vivado 2024.1 interface. On the left, the 'Flow Navigator' pane shows the 'IP Catalog' option under the 'PROJECT MANAGER' section, circled in red. A red arrow points to it with the text 'Click this to show the IP catalog!'. The main workspace is divided into several panes. The 'Sources' pane shows the project hierarchy, including 'soc\_top' and its components. The 'IP Properties' pane shows the 'Floating-point' IP core with version '7.1 (Rev. 18)'. The 'IP Catalog' pane on the right shows the search results for 'Floating-point' IP, with the 'Floating-point' core selected and circled in red. A red arrow points to it with the text 'We want to insert this floating-point IP that can compute inner-product.' The 'Details' pane shows the 'Floating-point' core's interfaces, including 'AXI4-Stream', which is also circled in red. A red arrow points to it with the text 'bus interface of the IP'. The 'Design Runs' pane at the bottom shows the status of the synthesis and implementation, with 'synth\_design Complete!' and 'write\_bitstream Complete!'.

Flow Navigator

- PROJECT MANAGER
  - Settings
  - Add Sources
  - Language Templates
  - IP Catalog
- IP INTEGRATOR
  - Create Block Design
- SIMULATION
  - Run Simulation
- RTL ANALYSIS
  - Run Linter
  - Open Elaborated Design
- SYNTHESIS
  - Run Synthesis
  - Open Synthesized Design
- IMPLEMENTATION
  - Run Implementation

PROJECT MANAGER - aquila\_mpd

Sources

- Verilog Header (1)
- soc\_top (soc\_top.v) (8)
  - Clock\_Generator: clk\_wiz\_0 (clk\_wiz\_0.xci)
  - Aquila\_SoC: aquila\_top (aquila\_top.v) (6)
  - UART: uart (uart.v)
  - Core2AXI\_0: core2axi\_if (core2axi\_if.v)
  - SD\_Card\_Controller: axi\_quad\_spi\_0 (axi\_quad\_spi\_0.xci)
  - synchronizer: cdc\_sync (cdc\_sync.v) (10)
  - Memory\_Arbitrer: mem\_arbitrer (mem\_arbitrer.v)
  - MIG: mig\_7series\_0 (mig\_7series\_0.xci)

Hierarchy IP Sources Libraries Compile Order

IP Properties

Floating-point

Version: 7.1 (Rev. 18)

Project Summary IP Catalog

Cores | Interfaces

Search: Q:

Name AXI4 Stat

- Math Functions
  - Adders & Subtracters
  - Conversions
  - CORDIC
  - Dividers
  - Floating Point
    - Floating-point

AXI4-Stream

Details

Name: Floating-point

Version: 7.1 (Rev. 18)

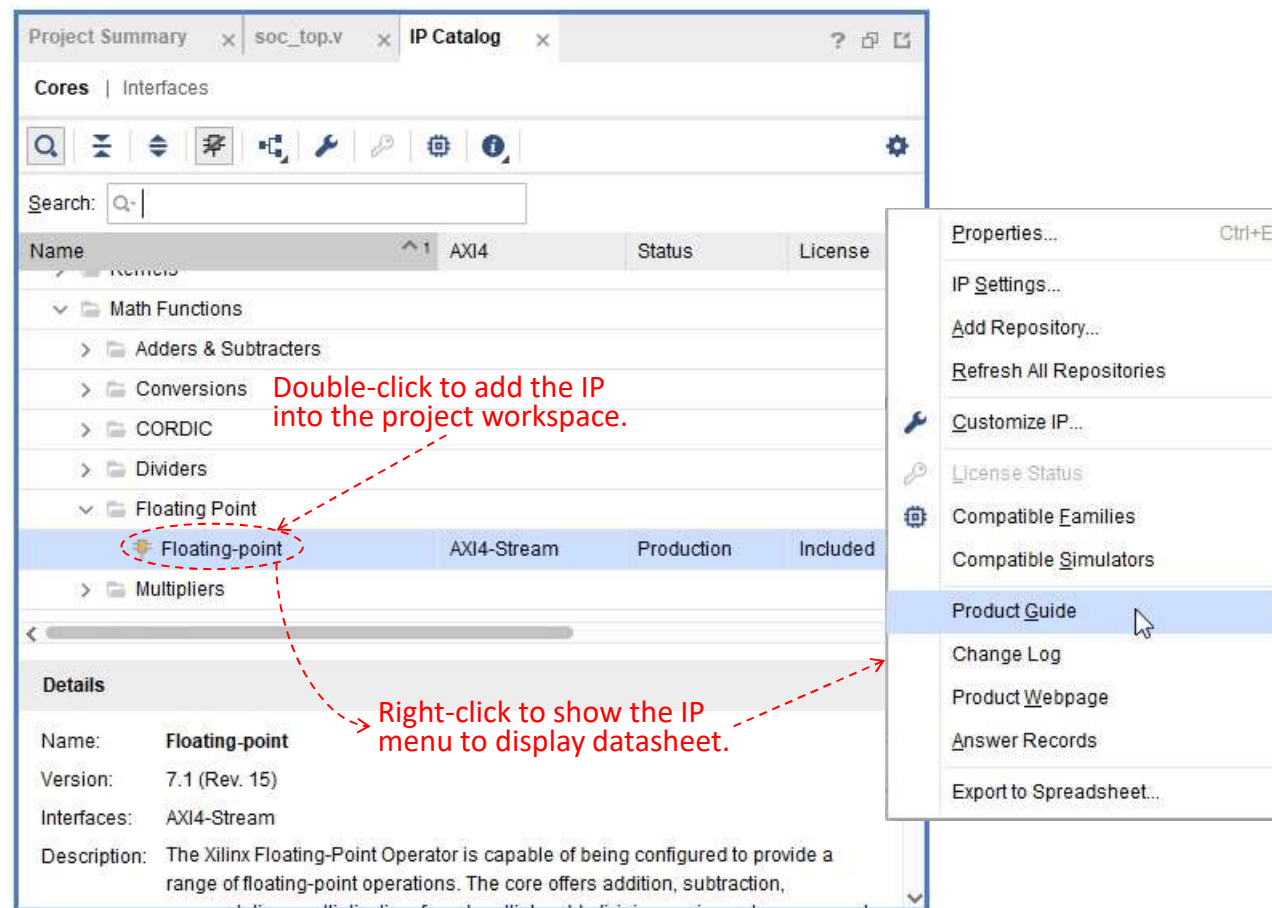
Interfaces: AXI4-Stream

Tcl Console Messages Log Reports Design Runs

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology
synth_1 (active)	constrs_1	synth_design Complete!									
impl_1	constrs_1	write_bitstream Complete!	1.274	0.000	0.009	0.000	10.908	0.000	0.914	0	22 Warn

# How to Add an IP into Aquila SoC

- Simply double-clicking the IP to add it to the workspace:

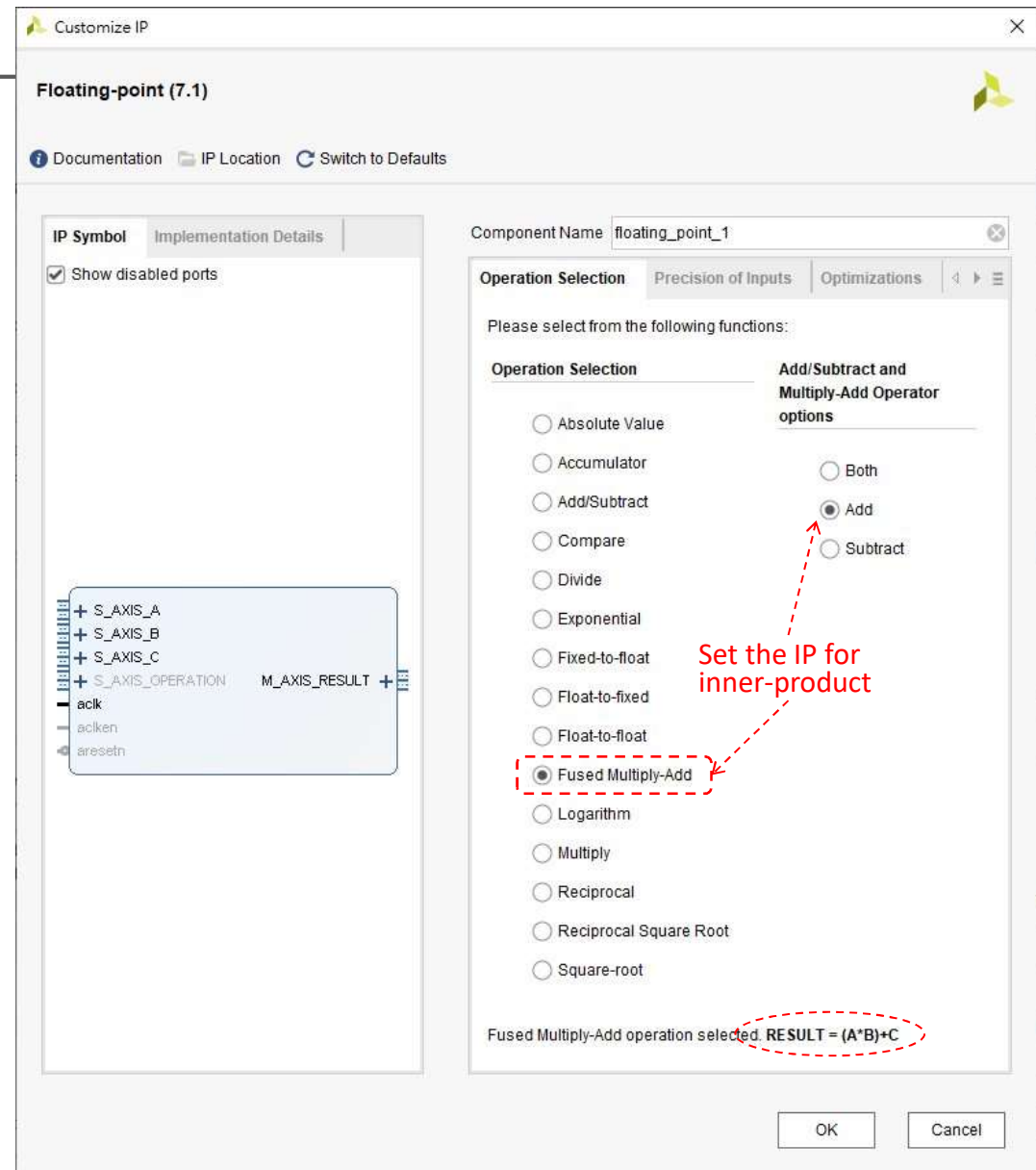


# Configure the IP

- ❑ Double-clicking the IP, a pop-up dialog will show
  - Set the IP to do Fused Multiply-Add
- ❑ Default parameters:
  - Blocking mode
  - 32-bit precision
  - Latency: 17 cycles

Can be reduced to 2 without timing error on Arty. However, the latency is not an issue for this HW.

The bottleneck is somewhere else.



# Inserting the IP from the TCL Script

- ❑ For HW submission, you should modify the TCL script to insert the floating-point IP into the workspace
  - To insert a fused multiply-add floating-point IP into the workspace, you can add the following code to `build.tcl`:

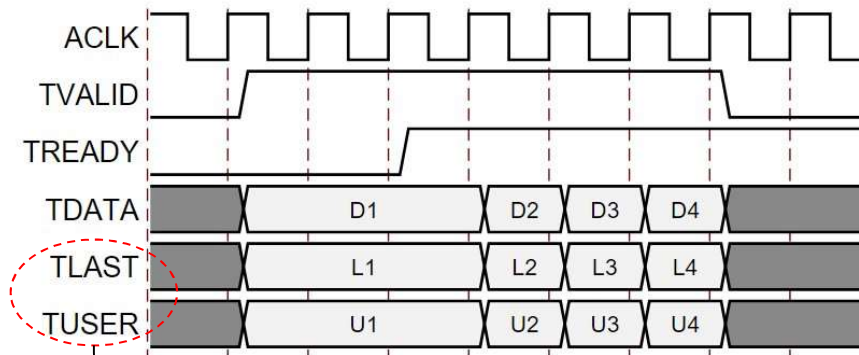
```
# Adding an Floating-Point IP
create_ip -name floating_point -vendor xilinx.com -library ip -module_name
floating_point_0
set_property -dict [ list \
    CONFIG.OPERATION_TYPE {FMA} \
    CONFIG.ADD_SUB_VALUE {Add}] [get_ips floating_point_0]
generate_target all [get_files ${proj_name}/${proj_name}.srcs/sources_1/ip/
floating_point_0/floating_point_0.xci]
```

- The parameter options can be listed using the following TCL command in the TCL console window of Vivado:

```
report_property [get_ips floating_point_0]
```

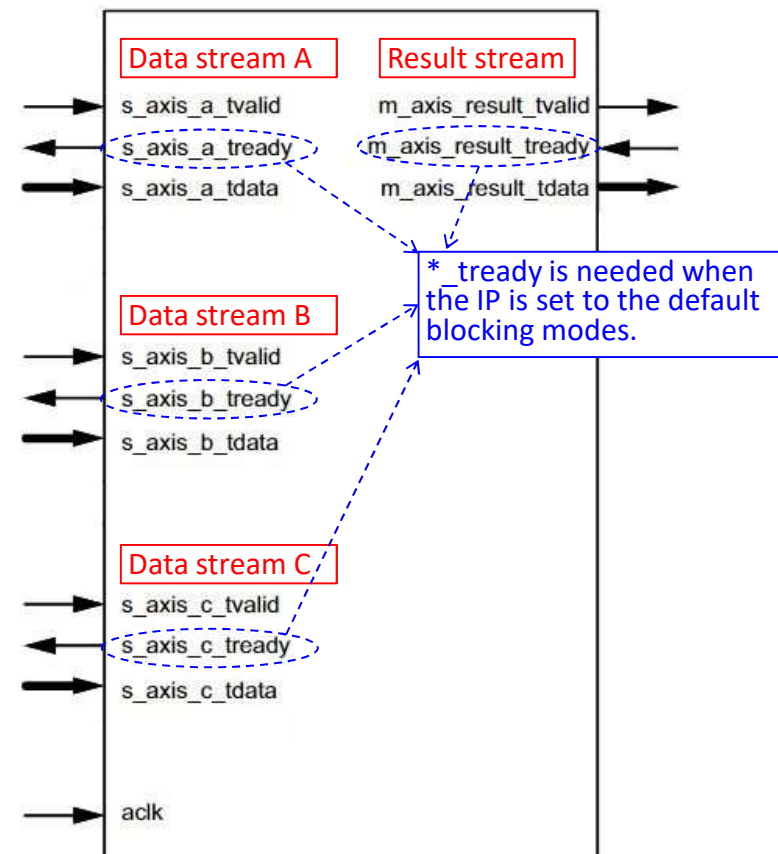
# AXI Floating-Point IP Interface

- ❑ The floating-point IP is designed to handle a sequence of floating-point operations
- ❑ AXI stream bus is used for I/O ports:



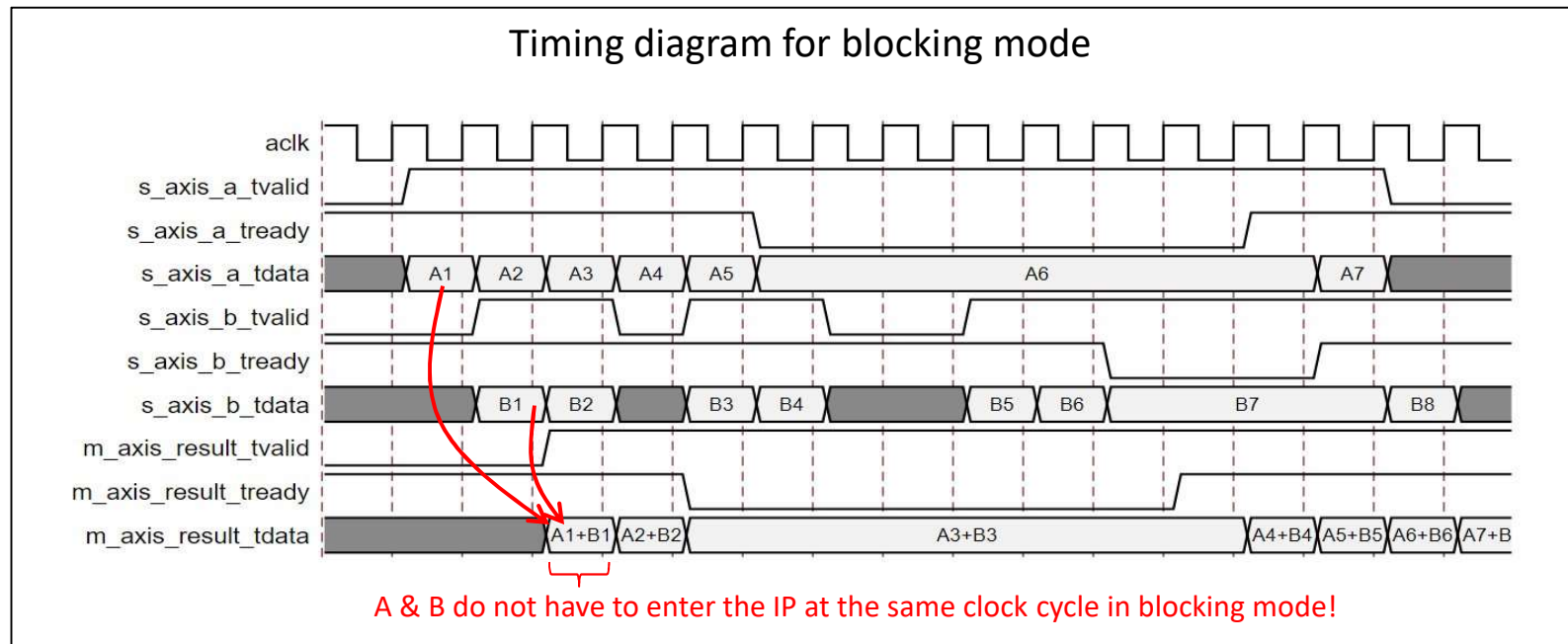
TLAST and TUSER are optional for the floating-point IP.

TLAST can be used to reset the accumulator when the IP is set to the "fused multiply-add" operation type, which is nice but not necessary.



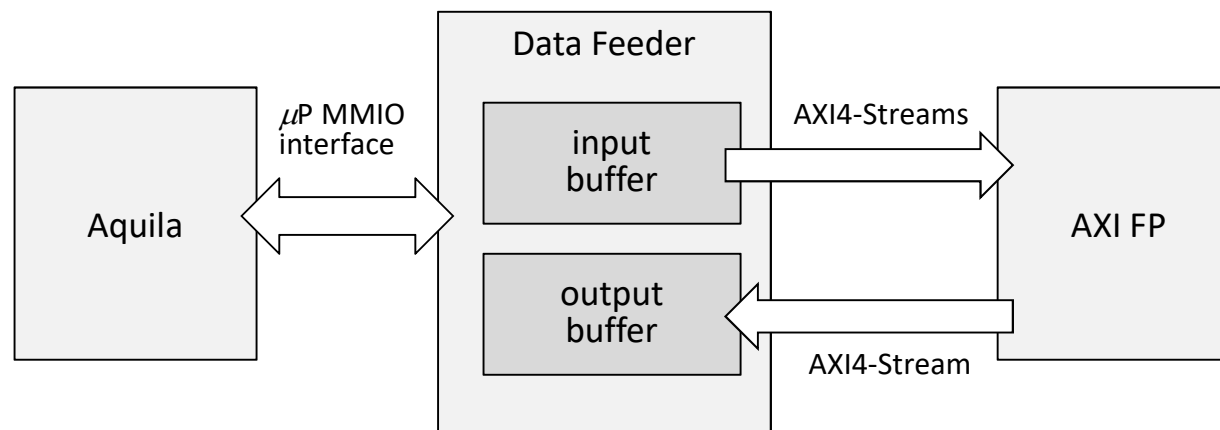
# Blocking vs Non-Blocking

- ❑ Two operation modes: blocking & non-blocking
  - Either mode is fine for this HW, but blocking mode requires more complex controls:



# Interface between Aquila and HW FP

- ❑ You can design a data feeder module between the Aquila and the AXI FP:
  - On the Aquila-side, use `memcpy()` to copy the vectors into the input buffer, and read the final value from the output buffer
  - On the AXI FP side, use AXI4-stream bus to transmit the data
  - A good example of the  $\mu$ P MMIO interface is in `clint.v`





# Examples of conv\_3d() HW Interface

- ❑ The SW code shall be replaced by HW trigger code:

```
void conv_3d(uint64_t o, convolutional_layer *entry, float_t *pa)
{
```

Code to setup pointers & indices!

```
for (uint64_t inc = 0; inc < in_.depth_; inc++) {
```

Code to setup pointers & indices!

```
for (uint64_t y = 0; y < out_.height_; y++) {
```

```
for (uint64_t x = 0; x < out_.width_; x++) {
```

```
float_t *ppw = pw, sum = (float_t) 0;
```

```
uint64_t wx = 0, widx = 0;
```

```
for (uint64_t wyx = 0; wyx < weight_.height_ * weight_.width_; wyx++) {
```

```
sum += *ppw++ * ppi[widx];
```

```
wx++, widx++;
```

```
if (wx == weight_.width_) {
```

```
wx = 0;
```

```
widx += const1;
```

```
}
```

```
pa[idx++] += sum;
```

```
ppi += w_stride_;
```

```
}
```

```
ppi += const2;
```

```
}
```

```
}
```

```
}
```

Read soc\_top.v, line 251 ~ 260.

```
volatile int *trigger_hw = (int *) 0xC4000000;
```

```
*trigger_hw = 1;
```

```
while (*trigger_hw) /* busy waiting */;
```

# Some Shortcuts to This HW

---

- ❑ To reduce the memory overhead, you can
  - Store the entire weight data into on-chip BRAM at the beginning of the program
    - This is not feasible for large CNN models, but ours is small
  - Feature maps of the previous layer can also fit into the BRAM
- ❑ In the CNN model, the paddings are 0 and the strides are 1, these constant can be used to simplify the functions to be made into accelerators

# Comments on the Homework

---

- ❑ For the accelerated system, the bottleneck is in the feeding of data streams to the AXI FP IP
  - You should measure the time spent on data feeding, and the time spent for computations, respectively
- ❑ The key point of this homework is to learn how to integrate an AXI accelerator to speed up computations
- ❑ You only need to write a detail report for this HW; there will be no demo
- ❑ You still need to upload your code to E3
  - TAs will verify that your code actually runs