

HW#5 Domain-Specific Accelerator



Chun-Jen Tsai
NYCU
12/23/2022

Homework Goal

- ❑ In this homework, you will integrate a domain-specific accelerator (DSA) to Aquila
- ❑ Your tasks:
 - You will integrate an accelerator IP into the Aquila SoC
 - An floating-point inner product IP will be used as an example here, but you are free to propose your own DSA
- ❑ You should upload your report to E3 by 1/8, 23:55.

Xilinx IP Catalog

The screenshot displays the Xilinx Vivado 2022.1 IP Catalog interface. The left sidebar shows the 'PROJECT MANAGER' section with 'IP Catalog' highlighted. A red circle and arrow point to it with the text 'Click this to show the IP catalog!'. The main window shows the 'PROJECT MANAGER - aquila_mpd' with a list of sources. Two sources are highlighted with red boxes and arrows: 'Clock_Generator: clk_wiz_0 (clk_wiz_0.xci)' and 'MIG: mig_7series_0 (mig_7series_0.xci)', with the text 'IPs inserted from the IP Catalog' above them. The right pane shows the 'IP Catalog' with the 'Floating Point' category selected. The 'AXI4-Stream' interface is highlighted with a red circle and the text 'bus interface of the IP'. The bottom pane shows the 'Tcl Console' with messages about refreshing IP repositories.

Flow Navigator

PROJECT MANAGER - aquila_mpd

Sources

- > Clock_Generator: clk_wiz_0 (clk_wiz_0.xci)
- > Aquila_SoC: aquila_top (aquila_top.v) (6)
- UART: uart (uart.v)
- > synchronizer: cdc_sync (cdc_sync.v) (10)
- Memory_Arbitrer: mem_arbitrer (mem_arbitrer.v)
- > MIG: mig_7series_0 (mig_7series_0.xci)
- > soc_tb (soc_tb.v) (2)
- > Memory File (2)
- > Configuration Files (1)

Hierarchy IP Sources Libraries Compile Order

Source File Properties

soc_top.v

Enabled

General Properties

Tcl Console

Messages Log Reports Design Runs

INFO: [IP_Flow 19-234] Refreshing IP repositories

INFO: [IP_Flow 19-1704] No user IP repositories specified

Type a Tcl command here

Project Summary IP Catalog

Cores Interfaces

Search: Q-

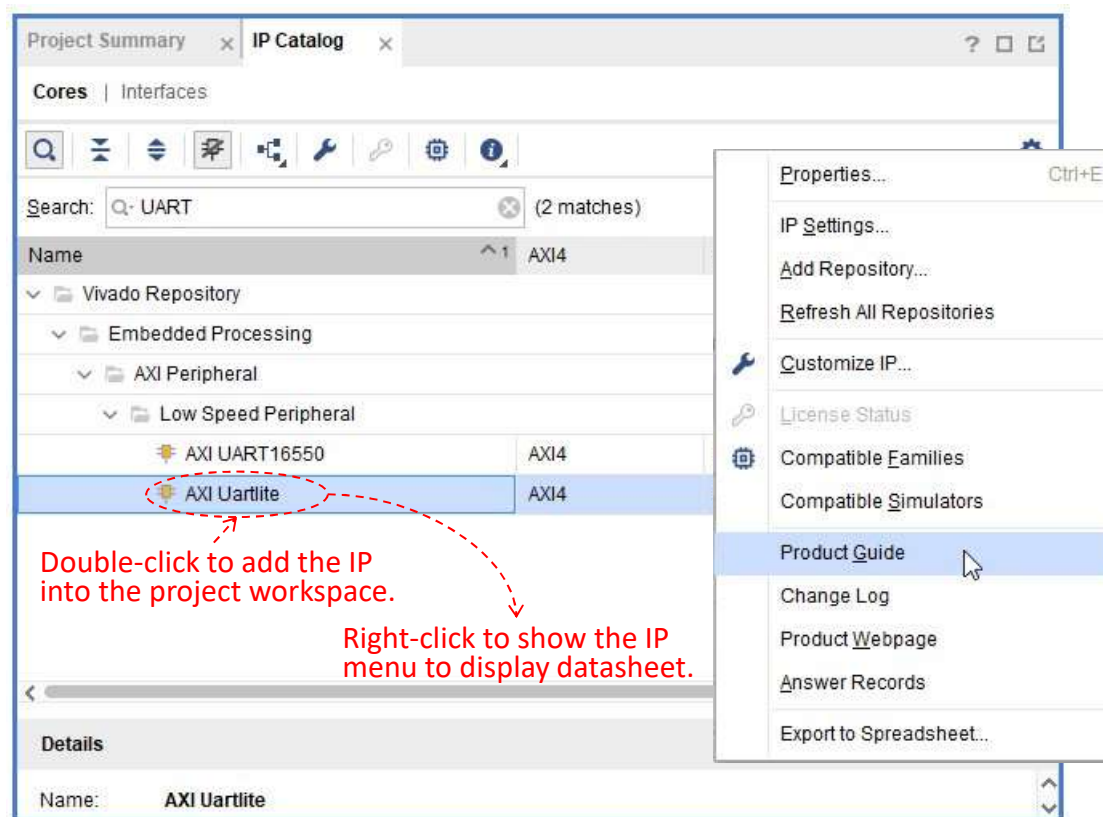
Name	AXI4	Status	License	VLNV
Math Functions				
Adders & Subtractors				
Conversions				
CORDIC				
Dividers				
Floating Point				
Floating-point	AXI4-Stream	Production	Included	xilinx.co
Multipliers				
Square Root				
Trig Functions				

Details

Select an IP or Interface or Repository to see details

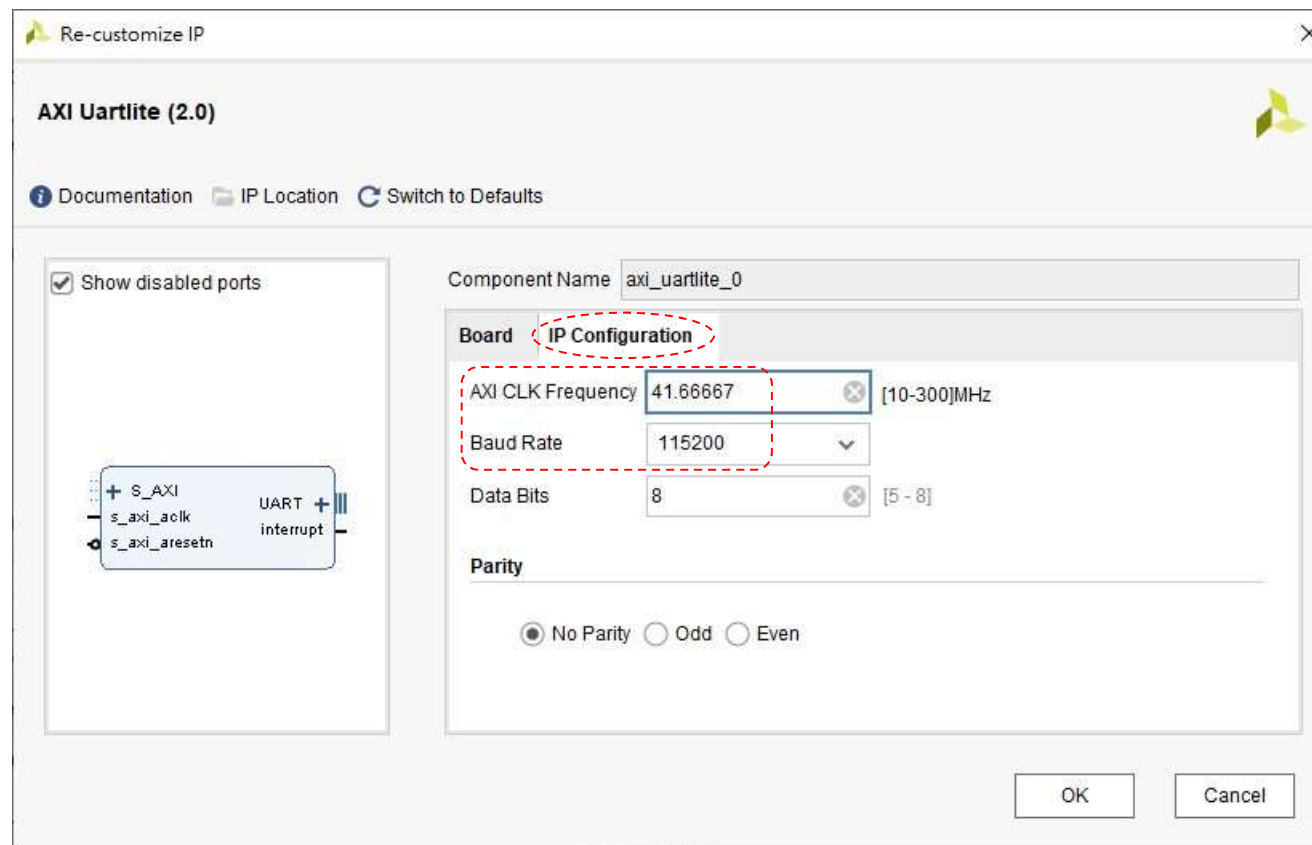
How to Add and IP into Aquila SoC

- ❑ As an example, we will use the AXI4 UART Lite IP to replace the default `uart.v` used in the Aquila SoC



Configure the IP

- ❑ As soon as you inserted the IP, a pop-up dialog shows for you to configure the IP:



Instantiation of the IP in Aquila

❑ Now, you can instantiate an `axi_uartlite_0`:

```
axi_uartlite_0 UART_Controller(  
  // AXI interface ports.  
  .s_axi_aclk(clk),  
  .s_axi_aresetn(~rst),  
  .s_axi_awaddr(axi_awaddr),  
  .s_axi_awvalid(axi_awvalid),  
  .s_axi_awready(axi_awready),  
  .s_axi_wdata(dev_din),  
  .s_axi_wstrb(dev_be),  
  .s_axi_wvalid(axi_wvalid),  
  .s_axi_wready(axi_wready),  
  .s_axi_bresp(axi_bresp),  
  .s_axi_bvalid(axi_bvalid),  
  .s_axi_bready(axi_bready),  
  .s_axi_araddr(axi_araddr),  
  .s_axi_arvalid(axi_arvalid),  
  .s_axi_arready(axi_arready),  
  .s_axi_rdata(axi_rdata),  
  .s_axi_rresp(axi_rresp),  
  .s_axi_rvalid(axi_rvalid),  
  .s_axi_rready(axi_rready),  
  
  // device-specific interface ports.  
  .interrupt(uart_interrupt),  
  .rx(uart_rx),  
  .tx(uart_tx)  
);
```

New AXI UART controller instance.

```
uart #(.BAUD(`SOC_CLK/`BAUD_RATE))  
UART(  
  .clk(clk),  
  .rst(rst),  
  
  .EN(dev_strobe & uart_sel),  
  .ADDR(dev_addr[3:2]),  
  .WR(dev_we),  
  .BE(dev_be),  
  .DATAI(dev_din),  
  .DATAO(uart_dout),  
  .READY(uart_ready),  
  
  .RXD(uart_rx),  
  .TXD(uart_tx)  
);
```

Old UART controller instance.

The Aquila Device Interface

- ❑ The Aquila core uses a simple memory-mapped I/O interface to talk to the external devices:

```
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 bus: enable both burst and single-beat data transfer
 - Lite bus: 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
 - A `core2axi_if.v` is available on E3 as an example of such bus bridge
 - `core2axi_if` only converts Aquila interface to AXI Lite bus

Core2AXI Module

```
module core2axi_if #( parameter XLEN = 32, parameter AXI_ADDR_LEN = 8)
(
    input                clk_i,
    input                rst_i,

    // Aquila M_DEVICE master interface signals.
    input                S_DEVICE_strobe_i,
    input [XLEN-1 : 0]   S_DEVICE_addr_i,
    input                S_DEVICE_rw_i,
    input [XLEN/8-1 : 0] S_DEVICE_byte_enable_i,
    input [XLEN-1 : 0]   S_DEVICE_data_i,
    output               S_DEVICE_data_ready_o,
    output [XLEN-1 : 0]  S_DEVICE_data_o,

    // Converted AXI master interface signals.
    output reg [AXI_ADDR_LEN-1:0] m_axi_awaddr, // Master write address signals.
    output reg                m_axi_awvalid, // Master write addr/ctrl is valid.
    input                    m_axi_awready, // Slave ready for write command.
    output [XLEN-1 : 0]       m_axi_wdata, // Master write data signals.
    output [XLEN/8 - 1 : 0]    m_axi_wstrb, // Master byte select signals.
    output reg                m_axi_wvalid, // Master write data is valid.
    input                    m_axi_wready, // Slave ready to receive write data.
    input [1 : 0]             m_axi_bresp, // Slave write-op response signal.
    input                    m_axi_bvalid, // Slave write-op response is valid.
    output reg                m_axi_bready, // Master ready for write response.
    output reg [AXI_ADDR_LEN-1:0] m_axi_araddr, // Master read address signals.
    output reg                m_axi_arvalid, // Master read addr/ctrl is valid.
    input                    m_axi_arready, // Slave is ready for read command.
    input [XLEN - 1 : 0]       m_axi_rdata, // Slave read data signals.
    input [1 : 0]             m_axi_rresp, // Slave read-op response signal
    input                    m_axi_rvalid, // Slave read response is valid.
    output reg                m_axi_rready // Master ready for read response.
);
```

Create a Workspace Using AXI UART

- ❑ Use the `aquila_dram` workspace as a starting point
- ❑ Download `soc_top.v` and `core2axi_if.v` from E3
 - Replace the old `soc_top.v` by the new one
 - Add `core2axi_if.v` to the source tree
- ❑ Insert the AXI UART Lite IP as shown in previous slides
- ❑ Now, you can build the bitstream and test your workspace

diff
meld

Suggested DSA

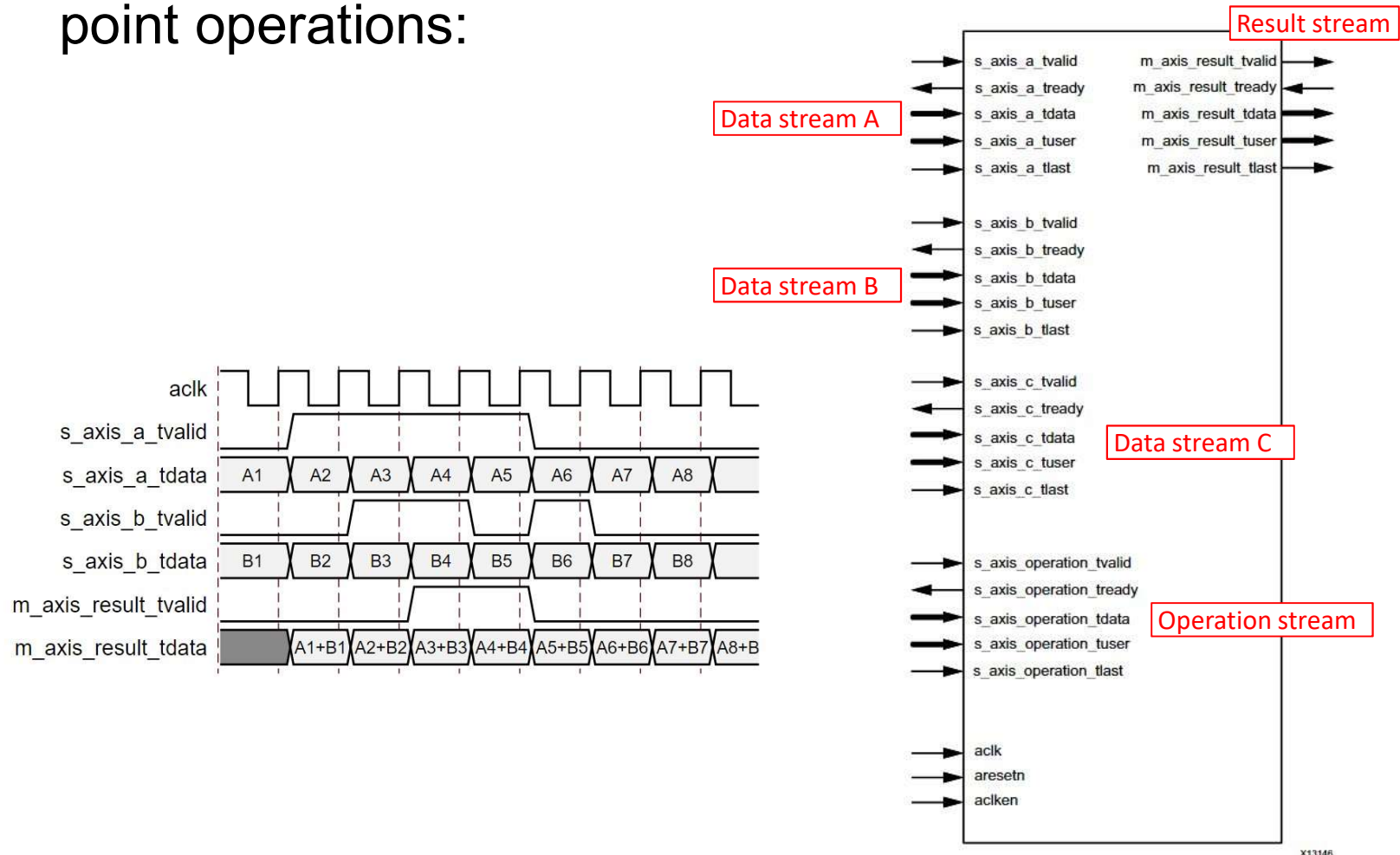
- ❑ For AI computing using convolutional neural network (CNN), the most crucial operation is inner-product:

$$\sum_{i=1}^n c_i x_i$$

- ❑ Since Aquila has no floating-point hardware, computing inner-product using soft-fp is expensive
- ❑ Use AXI Floating-Point IP to accelerate inner-product
 - The IP is design to perform a sequence of FP computations

AXI Floating-Point IP Interface

- ❑ The IP is designed to handle a sequence of floating-point operations:

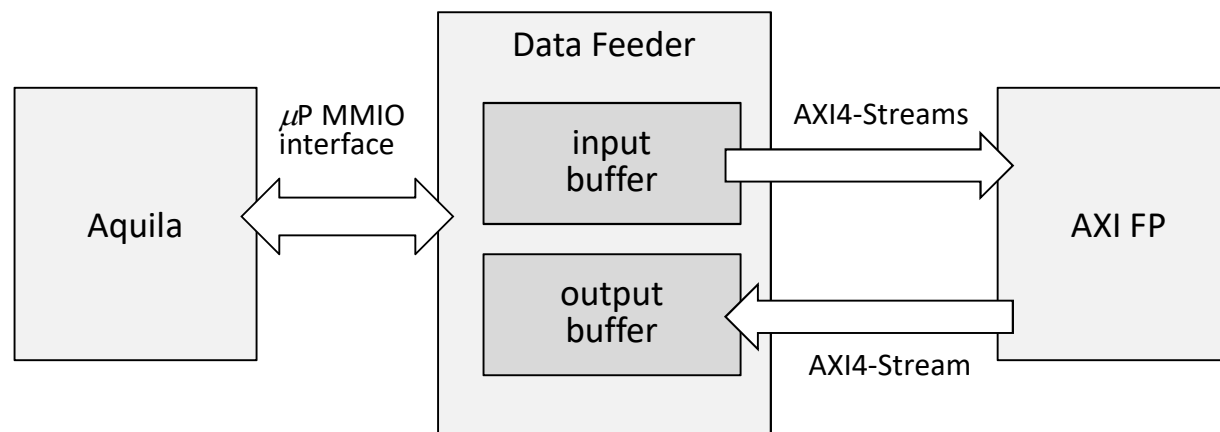


For HW#5

- ❑ Write a C code to compute the inner product
 - Generate two vectors of random floating point numbers
 - Set the vector length n to 8, 64, 128, ... etc.
 - Compute and print the inner-product
- ❑ Integrate the AXI Floating-Point IP into Aquila, and design an logic to feed the two vectors to the IP to compute the inner-product
- ❑ Compare the performance between the C and the HW implementations

Interface between Aquila and HW FP

- ❑ You can use the blocking mode interface of the AXI FP module, or design a logic to feed data into the FP HW to use the non-blocking mode interface



Comments on the Homework

- ❑ The key point of this homework is to learn how to integrate an AXI accelerator to speed up computations
- ❑ For the HW FP 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 separately
- ❑ 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
 - TA will build and make sure your code actually runs