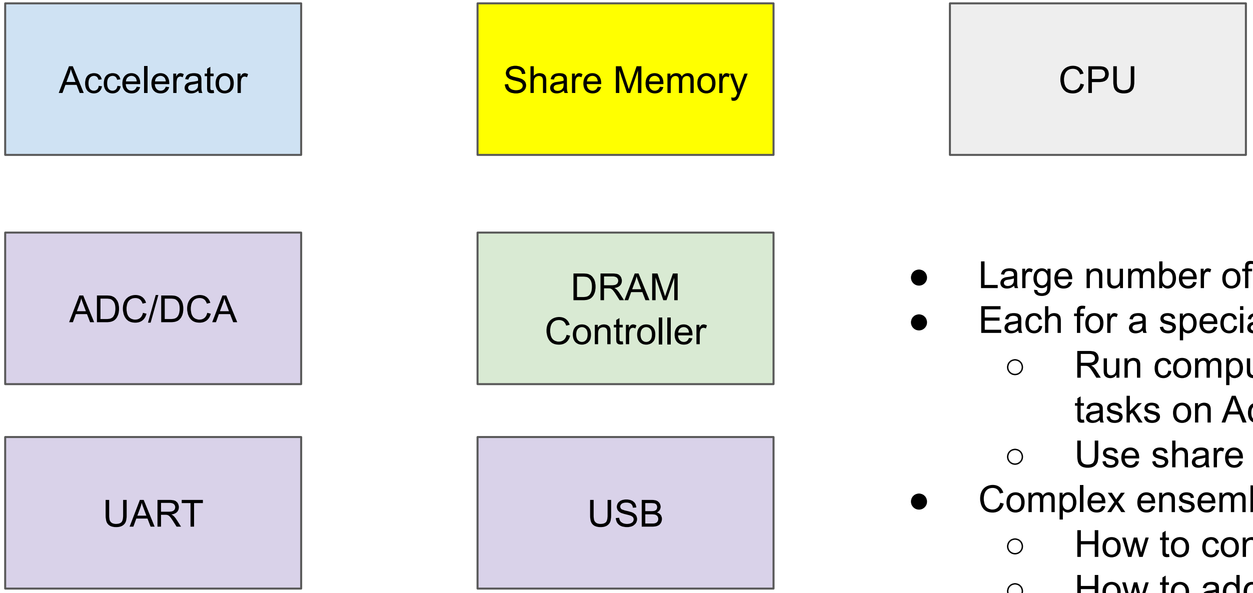
AXI Protocol Introduction

Speaker: Jia-Ming Lin

Outline

- What is AXI?
- AXI Interconnection
- AXI Stream Interface
- Labs
    - Using AXI Stream in HLS and PYNQ

Today’s System-On-Chips



Accelerator

Share Memory

CPU

ADC/DCA

DRAM Controller

UART

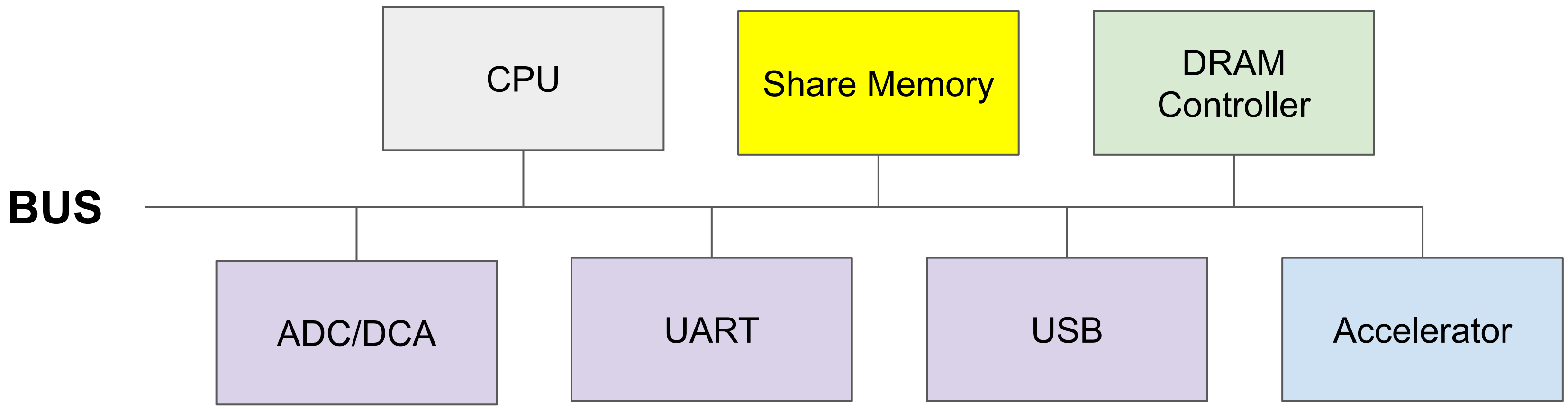
USB

- Large number of various modules
- Each for a special action, Example
  - Run computational intensive tasks on Accelerator
  - Use share memory to share data
- Complex ensemble of basic IP units
  - How to connect them together?
  - How to adding new modules?

Connectivity

- A standard
  - All modules talk based on that standard
  - All modules can talk easily to each other
- Maintenance
  - Design is easily maintained/updated and debugged
- Re-use
  - Units can be easily re-used in other design

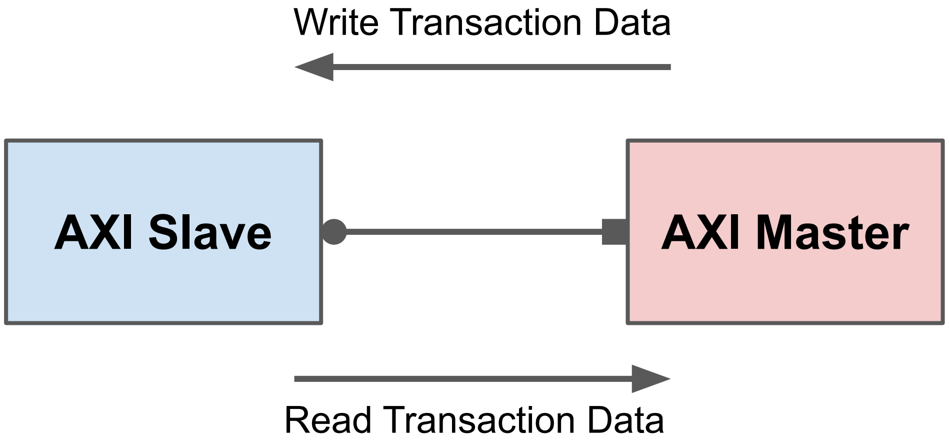
System-on-Chip Buses



- All modules are connected by the bus
- Through the bus
  - One module can talk to other modules
- Talking should obey a standard rule
- Famous SoC Buses
  - IBM Core-Connect, ARM AXI

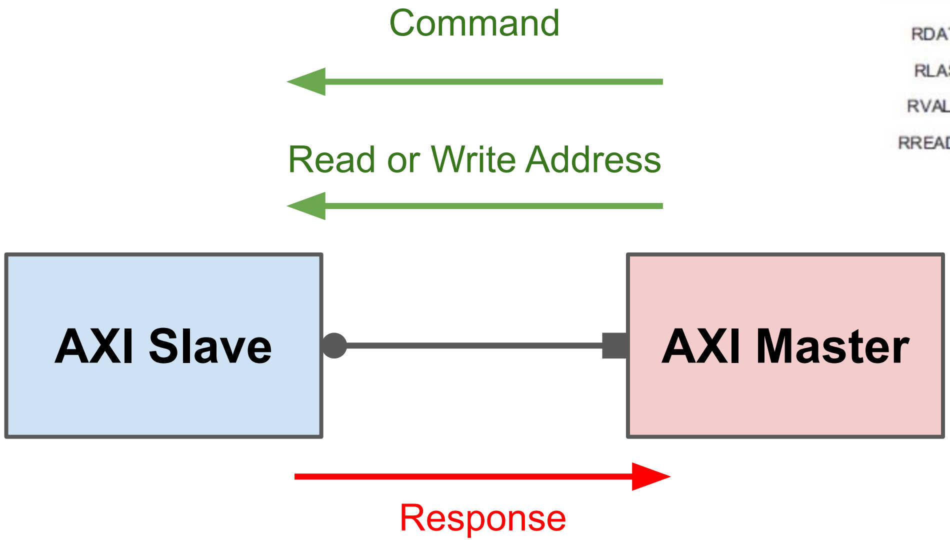
AXI Master / AXI Slave

- Transaction:
  - Transfer of data from one point in the hardware to another point
- Master: Initiates the transaction
- Slave: Response to the initiated transaction



- AXI Master: CPU
- AXI Slave: Memory
- CPU init tran to mem
- Memory response these reads and write
- Master rectangle
- Slave circle

AXI Master / AXI Slave



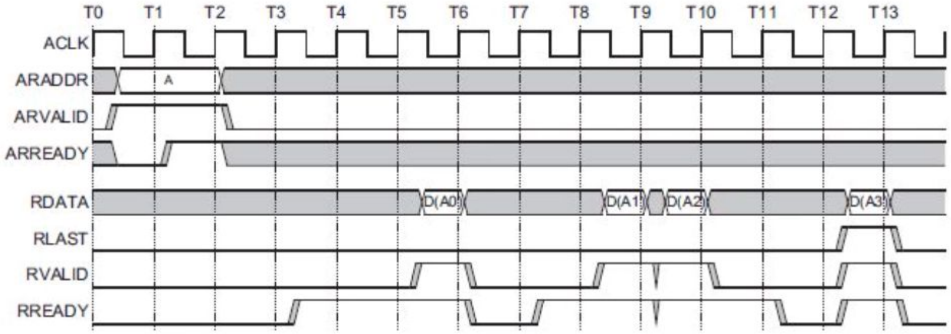
Command

Read or Write Address

AXI Slave

AXI Master

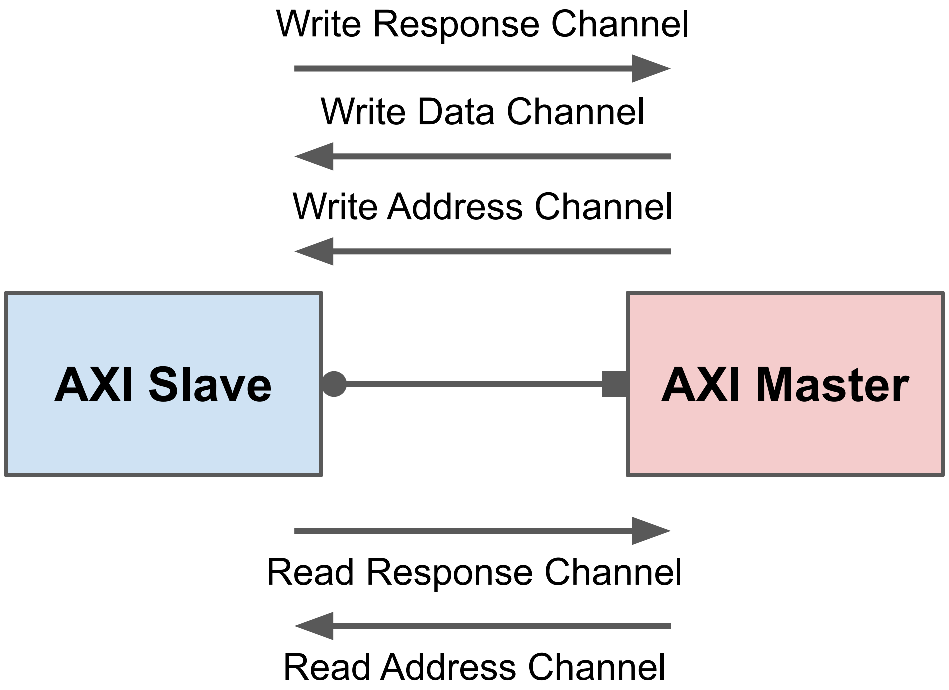
Response



AXI – Burst Read

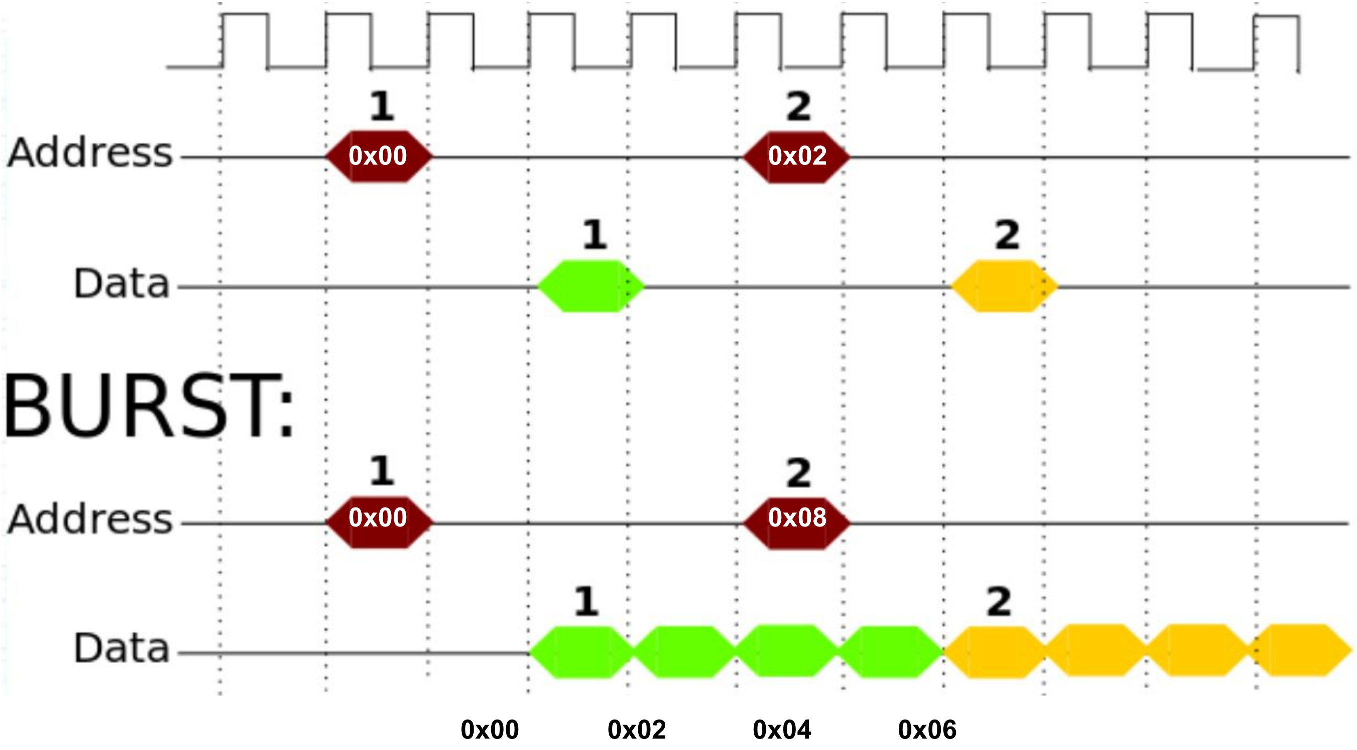
- ARVALID: initiate handshake with ARADDR is provided
- ARREADY: the memory is ready to transfer data
- RDATA: Data channel
- RLAST: the last data transfer

The 5 Channels of AXI Interface



- Each channel contains a set of signals
- Example: Write
  - Initiate: Master puts memory address on address channel
  - Data transfer: Master puts data on Data Channel
  - When complete, slave response results to master through Response Channel.

Burst AXI Transactions: When Access to Consecutive Data



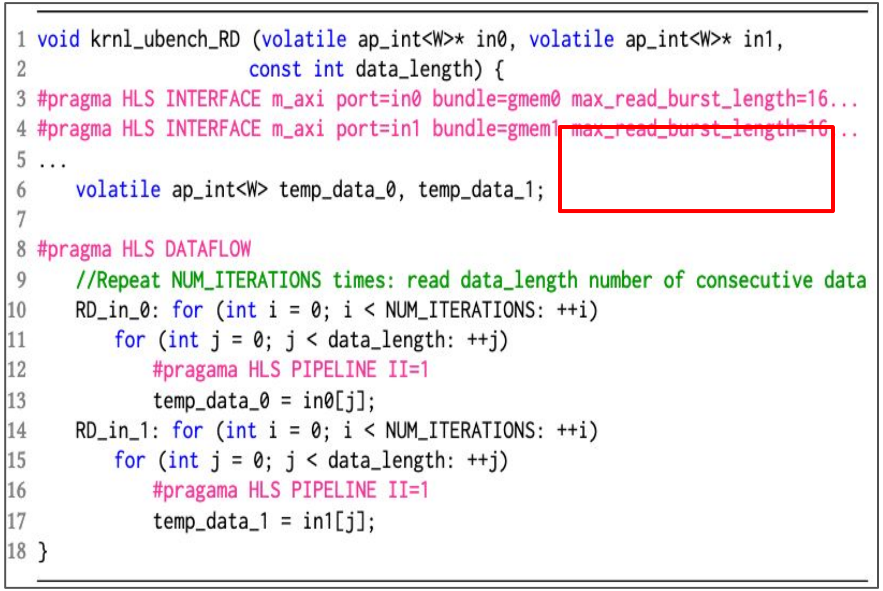
Reduce the handshake overhead

Burst Length:
AXI Signal “arlen”

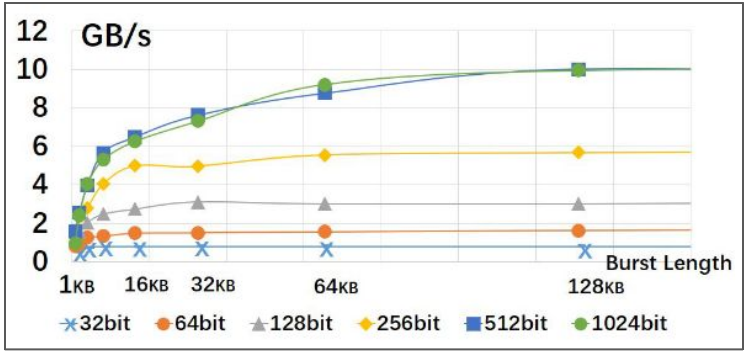
http://www.googoolia.com/wp/wp-content/uploads/2014/04/designing_with_axi_in_vivado_part_01.pdf

Burst AXI Transaction: When Access to Consecutive Data

Implementation in HLS

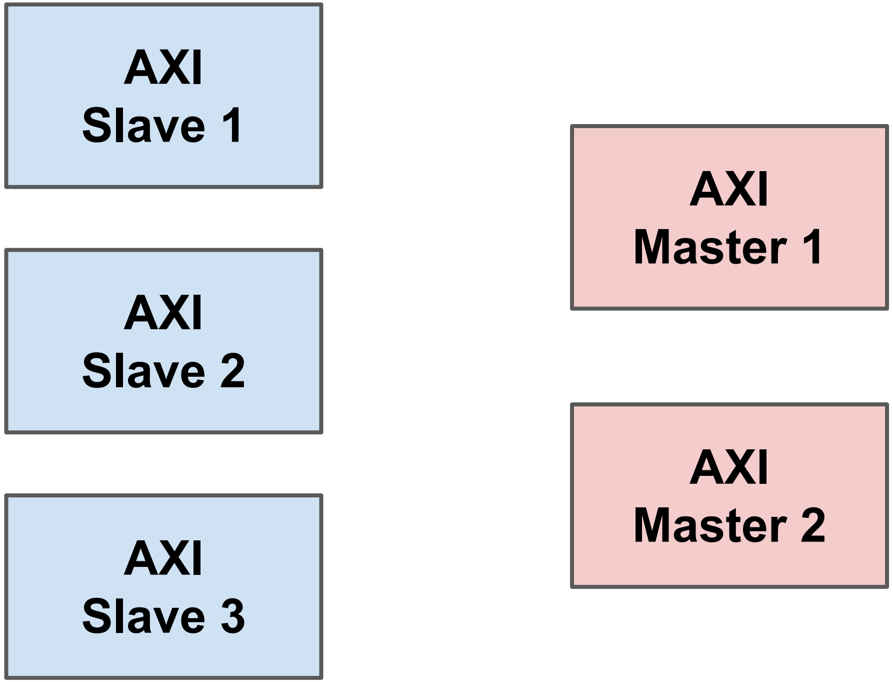


```
void krnl_ubench_RD (volatile ap_int<W>* in0, volatile ap_int<W>* in1,
                     const int data_length) {
#pragma HLS INTERFACE m_axi port=in0 bundle=gmem0 max_read_burst_length=16...
#pragma HLS INTERFACE m_axi port=in1 bundle=gmem1 max_read_burst_length=16...
...
    volatile ap_int<W> temp_data_0, temp_data_1;

#pragma HLS DATAFLOW
    //Repeat NUM_ITERATIONS times: read data_length number of consecutive data
    RD_in_0: for (int i = 0; i < NUM_ITERATIONS: ++i)
        for (int j = 0; j < data_length: ++j)
            #pragama HLS PIPELINE II=1
            temp_data_0 = in0[j];
    RD_in_1: for (int i = 0; i < NUM_ITERATIONS: ++i)
        for (int j = 0; j < data_length: ++j)
            #pragama HLS PIPELINE II=1
            temp_data_1 = in1[j];
}
```



AXI Interconnect

Connecting Masters and Slaves



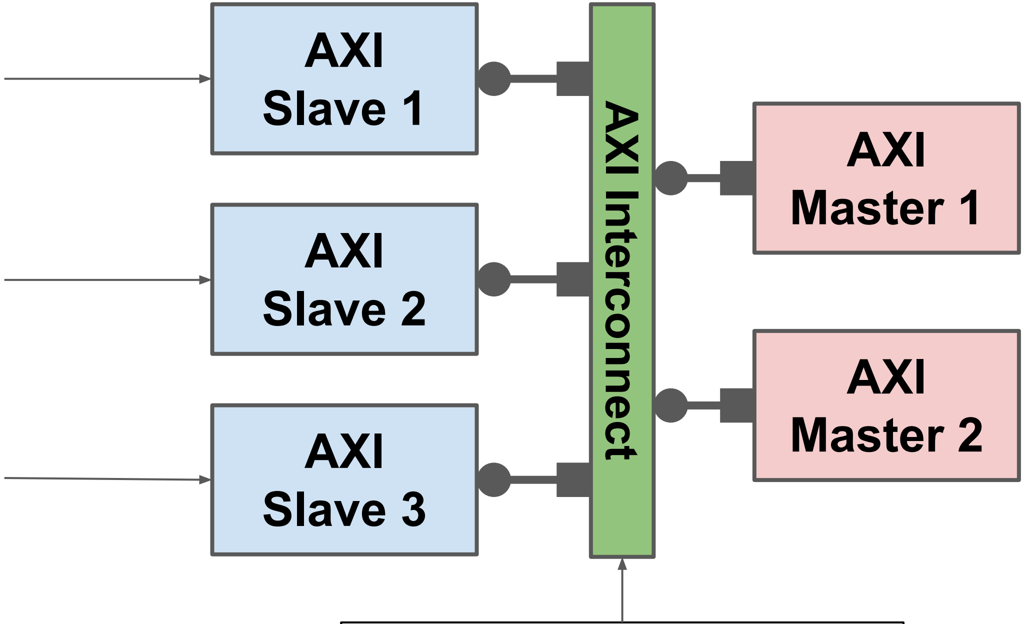
- What if we have multiple Masters and Slaves?
- Each Master should be able to initiate transaction to each Slave
- How to connect them?

AXI Interconnect: Connecting Masters and Slaves

Address Range: 4K
Address Offset: 0x40000000
Address: 0x40000000 - 0x40000FFF

Address Range: 4K
Address Offset: 0x40001000
Address: 0x40001000 - 0x40001FFF

Address Range: 64K
Address Offset: 0x40010000
Address: 0x40010000 - 0x4001FFFF



Address Decoding Table
Slave 1: 0x40000000 - 0x40000FFF
Slave 2: 0x40001000 - 0x40001FFF
Slave 3: 0x40010000 - 0x4001FFFF

AXI Interconnection: Flexibility

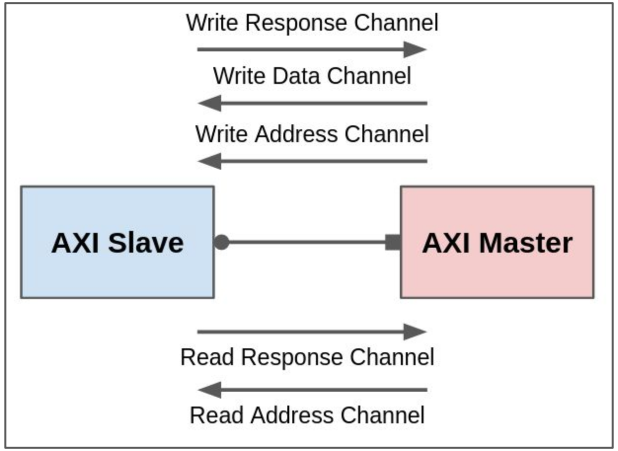
- Different number of Masters and Slaves Ports
- Width Conversion
  - 32 bits convert to 16 bits
- AXI3 to AXI4
- Clock Domain Transformation

AXI Stream Interface

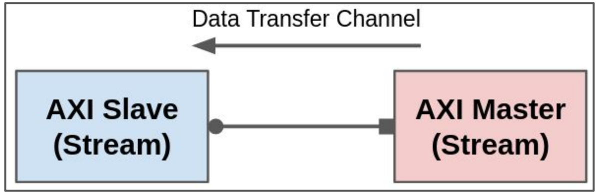
Two Types of AXI Interfaces

- Memory Mapped
  - Read / Write transactions contain destination address
- Streaming
  - One AXI channel

Memory Mapped

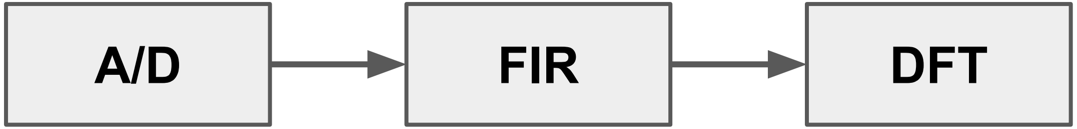


Streaming



Applications of AXI Stream

Signal Processing

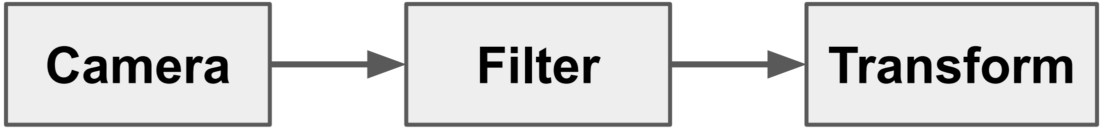


A/D

FIR

DFT

Image Processing



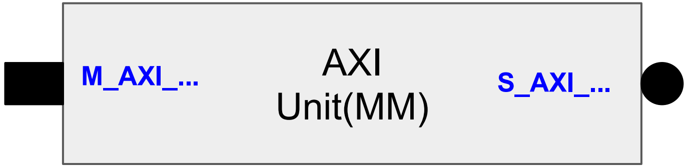
Camera

Filter

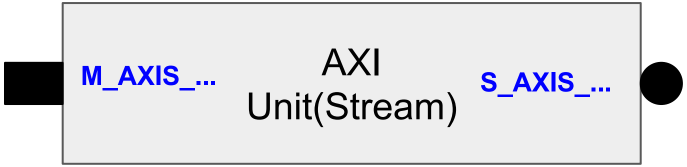
Transform

AXI Ports Naming Styles

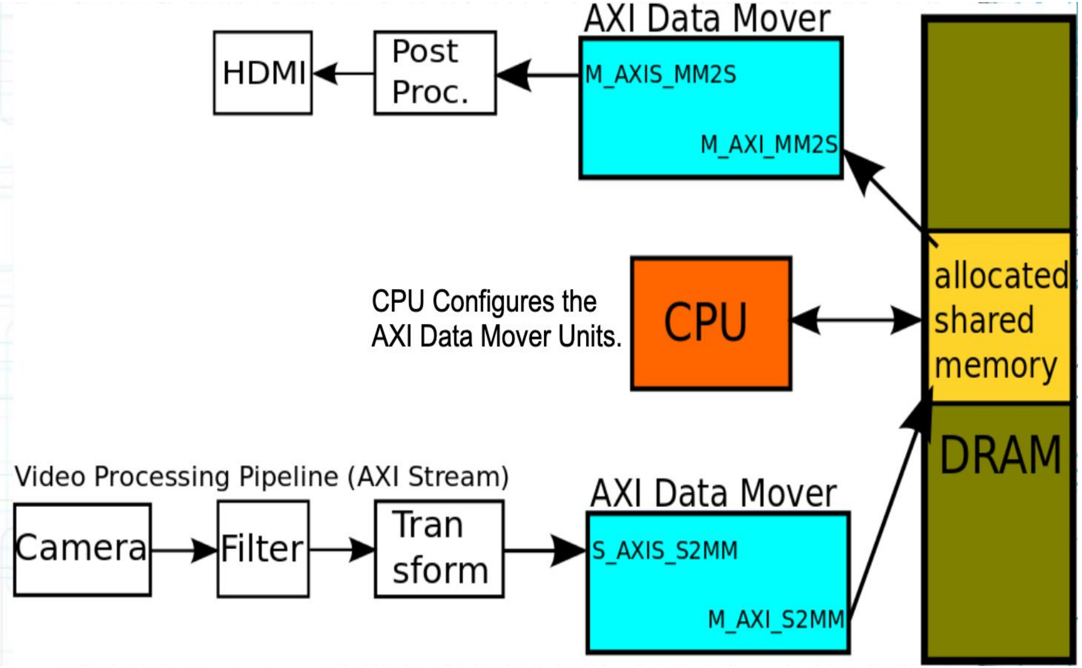
- Memory Mapped



- Streaming



AXI Data Mover



Data Mover

- Gets configured by the host CPU
- Interrupts when a transfer task is done
  - Example: a frame is transferred completely
- Gets triggered by the host CPU

Example:

- AXI Central DMA engine

http://www.googoolia.com/wp/wp-content/uploads/2014/04/axi_stream.pdf

Summary

- Different modules are connected by Buses, e.g. AXI interface
    - Connectivity: Standard, Maintenance, Re-use
- Types of AXI interface
    - Memory Mapped
        - Full AXI (Burst Capable)
        - AXI Lite (Single Beat)
    - Stream
- Next time
    - Text book(Parallel Programming for FPGA) chapter 3 CORDIC.

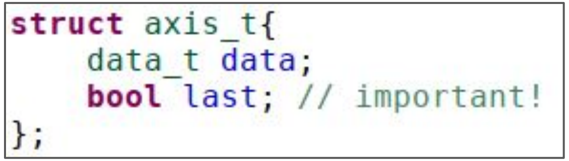
Labs

Prepare the HLS Project

- Download files, link
- Create a HLS project and import files
  - **fir.cpp**: top function
  - **fir.h**: header file
  - **fir_testbench.cpp**: testbench.
- We aims to using **AXI stream interface**, so leave N is small size(=21) for simplicity.
- You should further
  - Validate the results from Python and hardware design
  - Compare the performances of CPU and hardware design

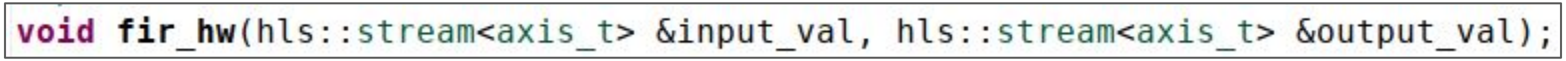
Using AXI Stream in HLS(1/2)

- AXIS package



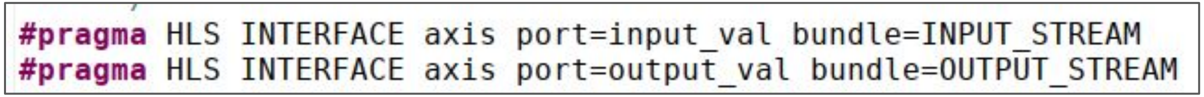
```
struct axis_t{
    data_t data;
    bool last; // important!
};
```

- Top function interfaces



```
void fir_hw(hls::stream<axis_t> &input_val, hls::stream<axis_t> &output_val);
```

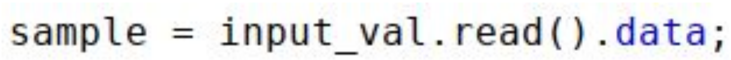
- HLS Pragma



```
#pragma HLS INTERFACE axis port=input_val bundle=INPUT_STREAM
#pragma HLS INTERFACE axis port=output_val bundle=OUTPUT_STREAM
```

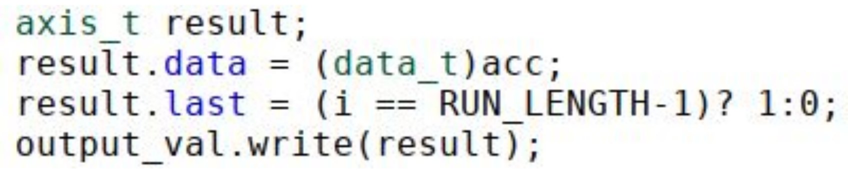
Using AXI Stream in HLS(1/2)

- AXIS read input data



```
sample = input_val.read().data;
```

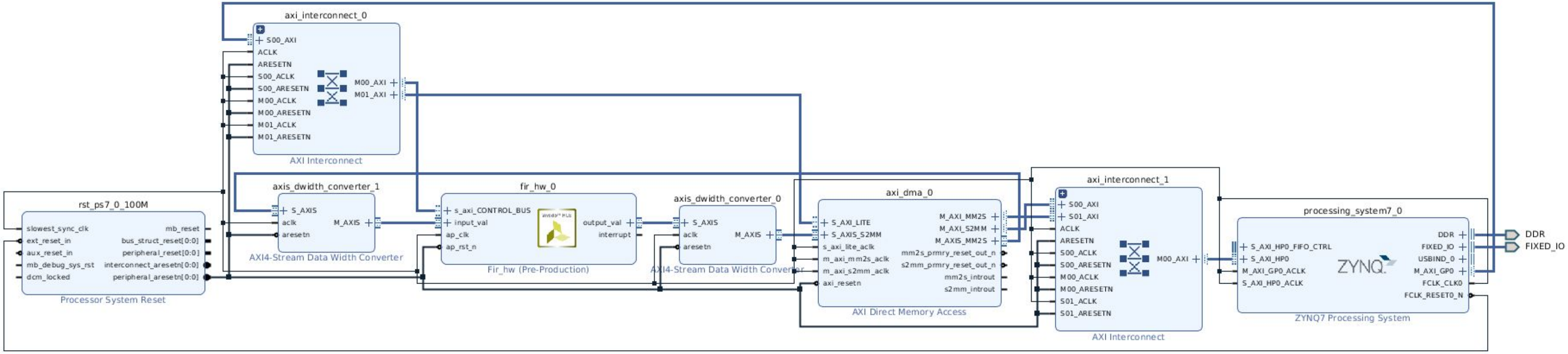
- AXIS write output data



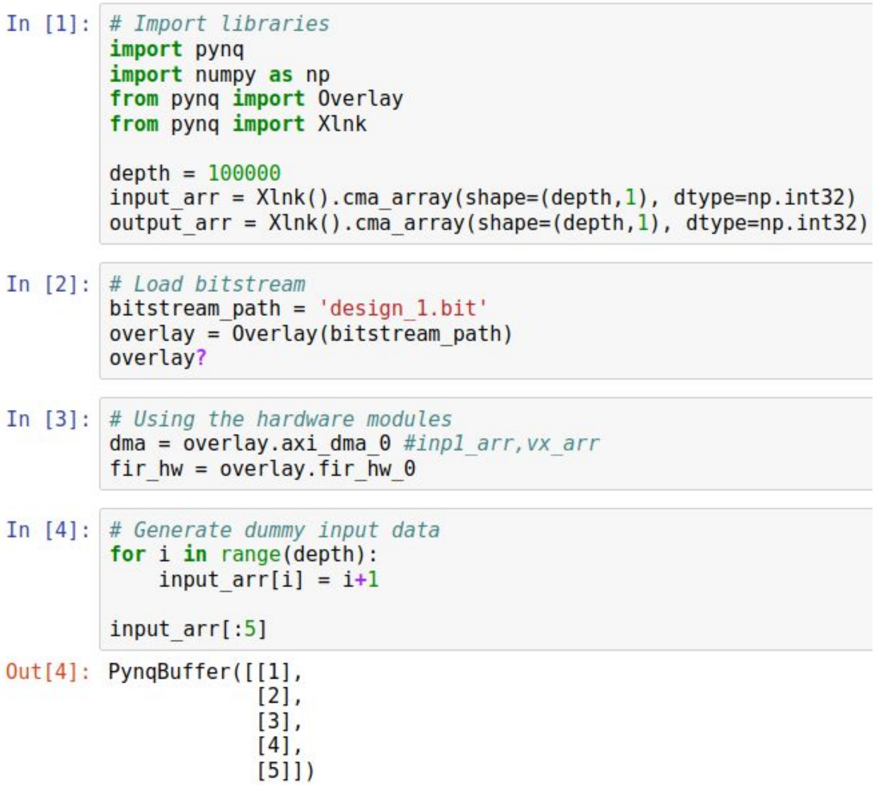
```
axis_t result;
result.data = (data_t)acc;
result.last = (i == RUN_LENGTH-1)? 1:0;
output_val.write(result);
```

Prepare Bitstream File

- Create a Vivado Project
- I will show how to construct the block diagram in live.



PYNQ Scripts

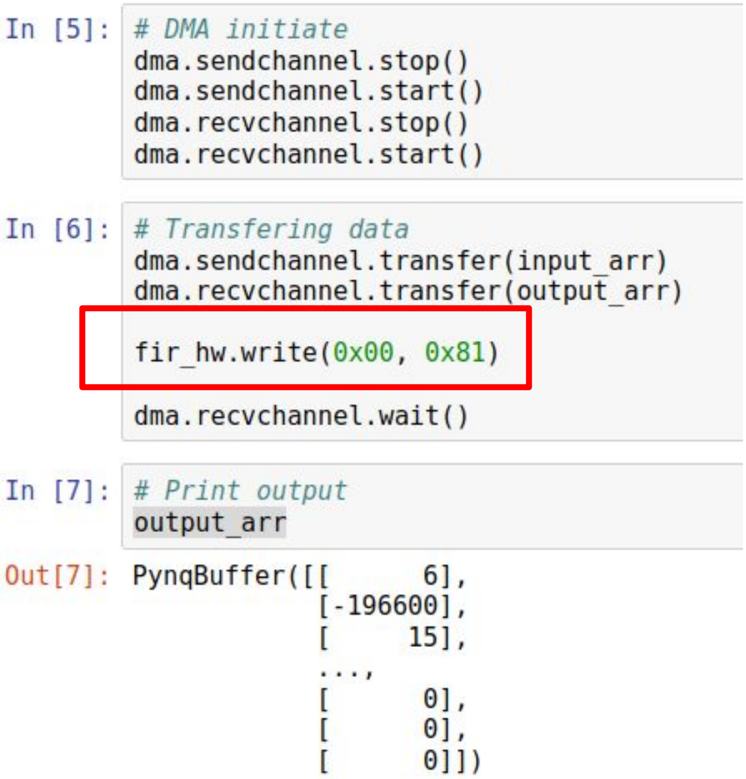


```
In [1]: # Import libraries
        import pynq
        import numpy as np
        from pynq import Overlay
        from pynq import Xlnk

        depth = 100000
        input_arr = Xlnk().cma_array(shape=(depth,1), dtype=np.int32)
        output_arr = Xlnk().cma_array(shape=(depth,1), dtype=np.int32)
```

```
In [2]: # Load bitstream
        bitstream_path = 'design_1.bit'
        overlay = Overlay(bitstream_path)
        overlay?
```

```
In [3]: # Using the hardware modules
        dma = overlay.axi_dma_0 #inp1_arr,vx_arr
        fir_hw = overlay.fir_hw_0
```

```
In [4]: # Generate dummy input data
        for i in range(depth):
            input_arr[i] = i+1

        input_arr[:5]

Out[4]: PynqBuffer([[1],
                    [2],
                    [3],
                    [4],
                    [5]])
```



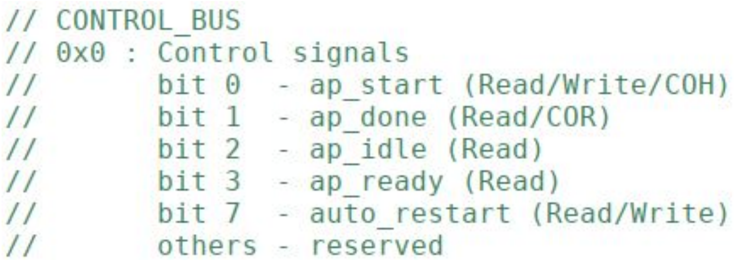
```
In [5]: # DMA initiate
        dma.sendchannel.stop()
        dma.sendchannel.start()
        dma.recvchannel.stop()
        dma.recvchannel.start()
```

```
In [6]: # Transfering data
        dma.sendchannel.transfer(input_arr)
        dma.recvchannel.transfer(output_arr)

        fir_hw.write(0x00, 0x81)

        dma.recvchannel.wait()
```

```
In [7]: # Print output
        output_arr

Out[7]: PynqBuffer([[      6],
                    [-196600],
                    [     15],
                    ...,
                    [      0],
                    [      0],
                    [      0]])
```

Trigger FIR IP to start

- The control bus of “**fir_hw**”



```
// CONTROL_BUS
// 0x0 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
```

- fir_hw.write(0x00, 0x81)
  - 0x81 means bit 0 and bit 7 are high