Demonstration of PYNQ Framework on PYNQ Z2 Board

Prof. Sachin B Patkar

Indian Institute of Technology
Bombay

January 22, 2024

Table of Contents

Table of Contents

PYNQ Z2

- PYNQ stands for Python Productivity for Zynq
- It is an opensource project from Xilinx that makes it easier to use Xilinx platforms
- Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors to build more capable and exciting electronic systems
- PYNQ Z2 is mostly used in the projects of embedded systems
- The following are required to implement designs based on PYNQ
  1. The PYNQ-Z2 board featuring the ZYNQ XC7Z020-1CLG400C SoC
  2. The PYNQ image and 8 GB SD Card
  3. Micro-USB cable
  4. Ethernet Cable

PYNQ Z2 Board

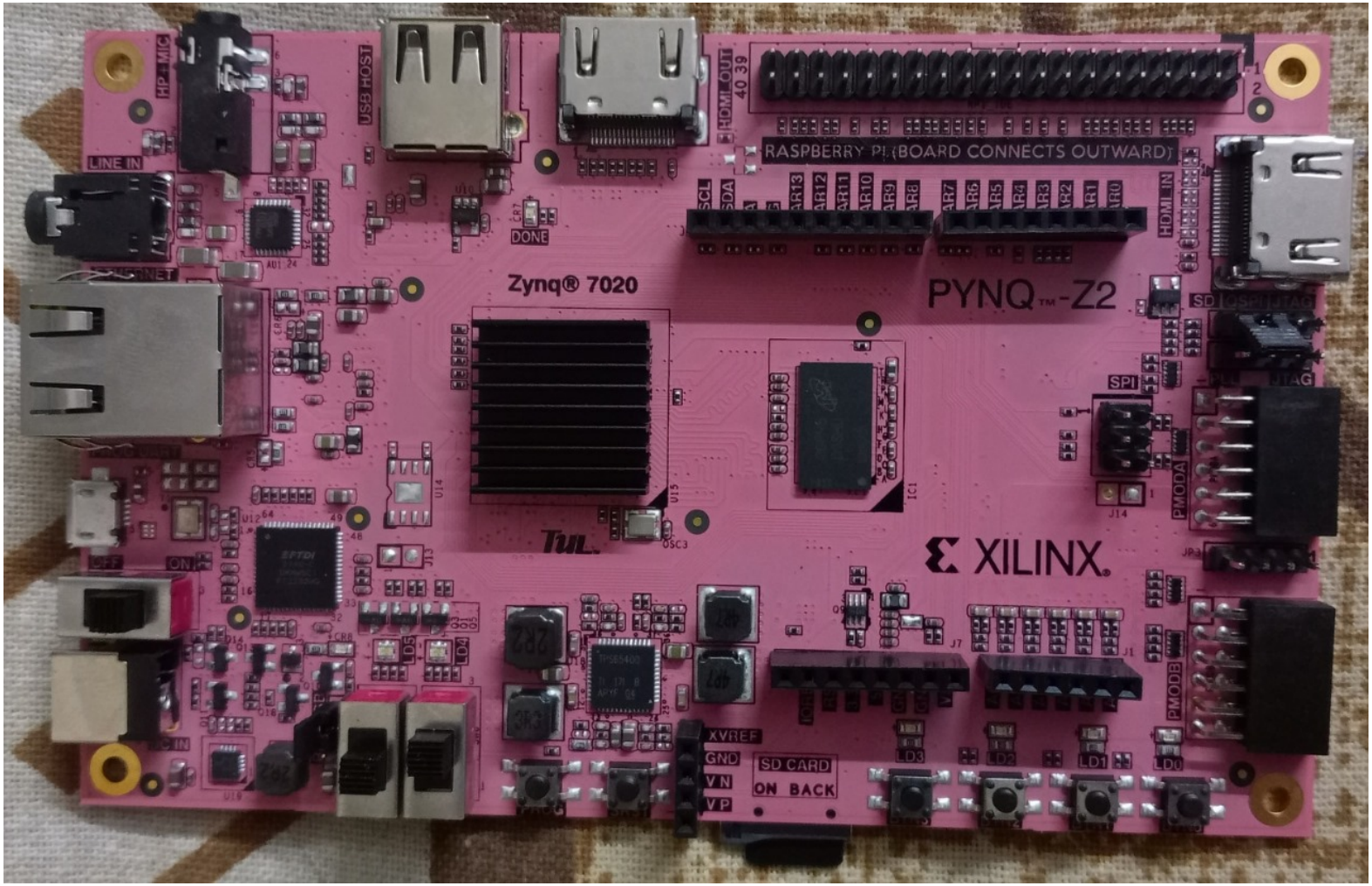

Figure: PYNQ Z2 Board

PYNQ Z2 Board Features

- Following are the features of the PYNQ Z2 board. Apart from the below IO interfaces, switches and LEDs are also available.

Feature	Description
FPGA	Zynq-7000 SoC XC7Z020-1CLG400C
Memory	1) 512 MB DDR3 2) 128MB Quad SPI Flash 3) Micro SD card connector
Clocks	1) One 125 MHz for PL 2) One 50 MHz for PS
Expansion ports	1) 2 PMOD ports 2) 1 Arduino shield connector 3) Raspberry Pi connector

Table: PYNQ Z2 board features

Table of Contents

PYNQ Z2 setup

1. Set the boot jumper to SD position and power jumper to USB
2. Insert Micro-SD card loaded with PYNQ image in the SD card slot
3. Connect the Micro USB cable and Ethernet cable
4. Turn on the board and check the boot sequence

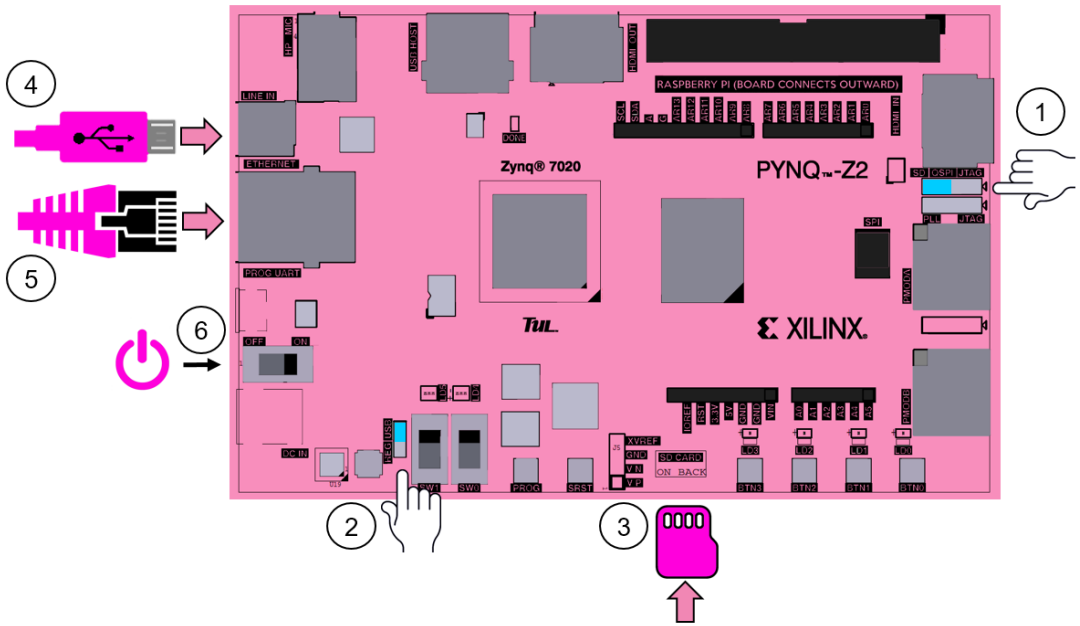

Figure: PYNQ Z2 board setup

PYNQ Framework

- The AMD-Xilinx Zynq All Programmable device is an SOC based on
  1. A dual-core ARM Cortex-A9 processor (Processing System or PS)
  2. FPGA fabric (referred to as Programmable Logic or PL)
- The PS subsystem includes a number of dedicated peripherals like
  1. Memory controllers
  2. USB
  3. UART, I2C, SPI and etc...
  4. can be extended with additional hardware IP in a PL Overlay

Overlay

Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can be used to accelerate a software application, or to customize the hardware platform for a particular application

PYNQ Series Framework

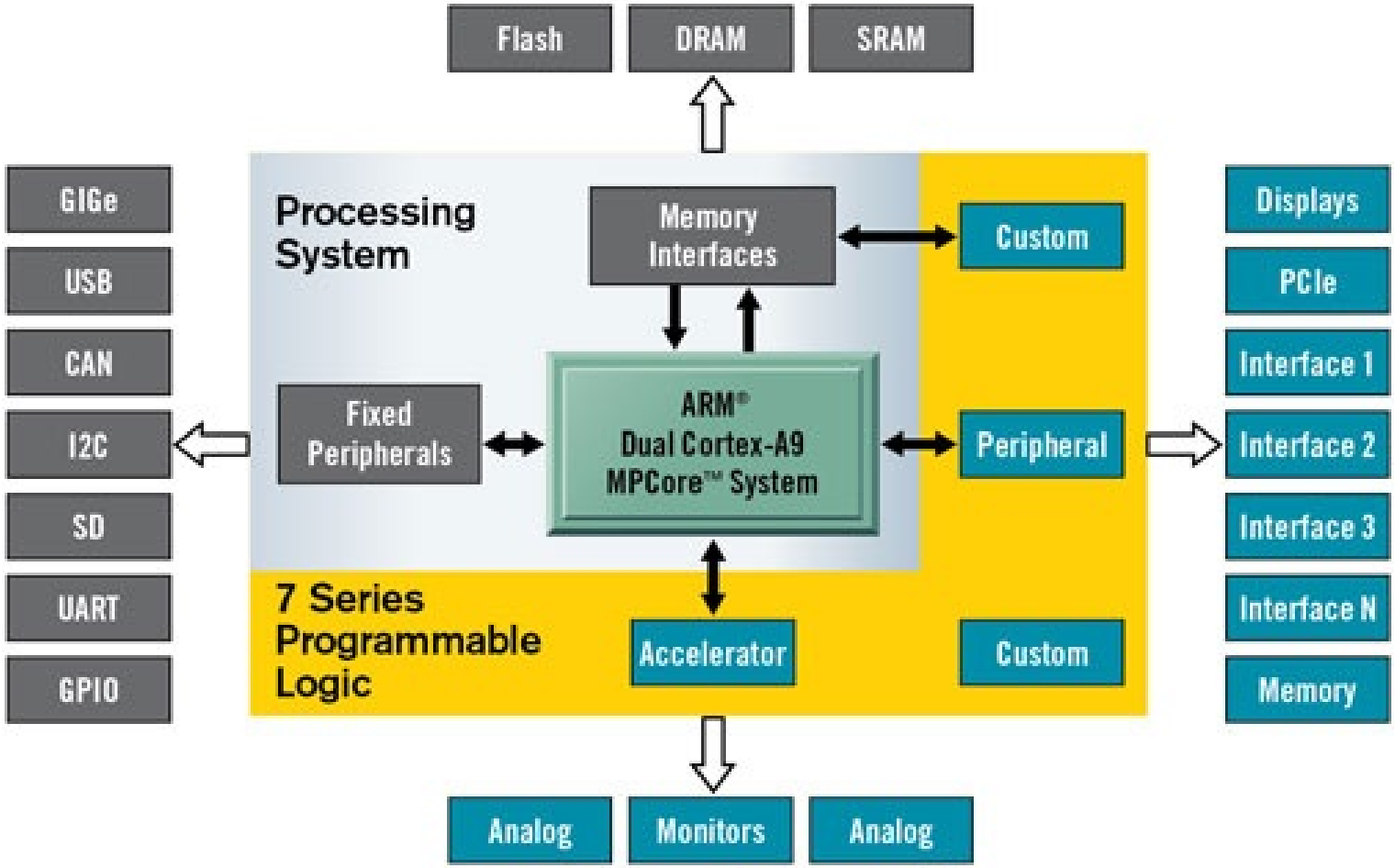

Figure: PYNQ Framework

PYNQ Framework

- PYNQ provides a Python interface to allow overlays in the PL to be controlled from Python running in the PS
- PYNQ overlays are created by hardware designers, and wrapped with this PYNQ Python API
- Software developers can then use the Python interface to program and control specialized hardware overlays without needing to design an overlay themselves
- This is analogous to software libraries created by expert developers which are then used by many other software developers working at the application level

Table of Contents

AXI Interfaces

- Since PYNQ is a Xilinx opensource project, AXI interfaces are being supported to allow communication between PS and PL
- AXI4 (Full), AXI4Lite and AXI4Stream can be used by the PS to control PL
- The Zynq has 9 AXI interfaces between the PS and the PL
  1. 4 AXI Master HP ports
  2. 2 AXI Master GP ports
  3. 2 AXI SIave GP ports
  4. 1 AXI Master ACP port
- There are four pynq classes that are used to manage data movement between the Zynq PS and PL interfaces
  1. General Purpose Input/Output
  2. Memory Mapped I/O
  3. Memory Allocation
  4. Direct Memory Access

AXI Interfaces

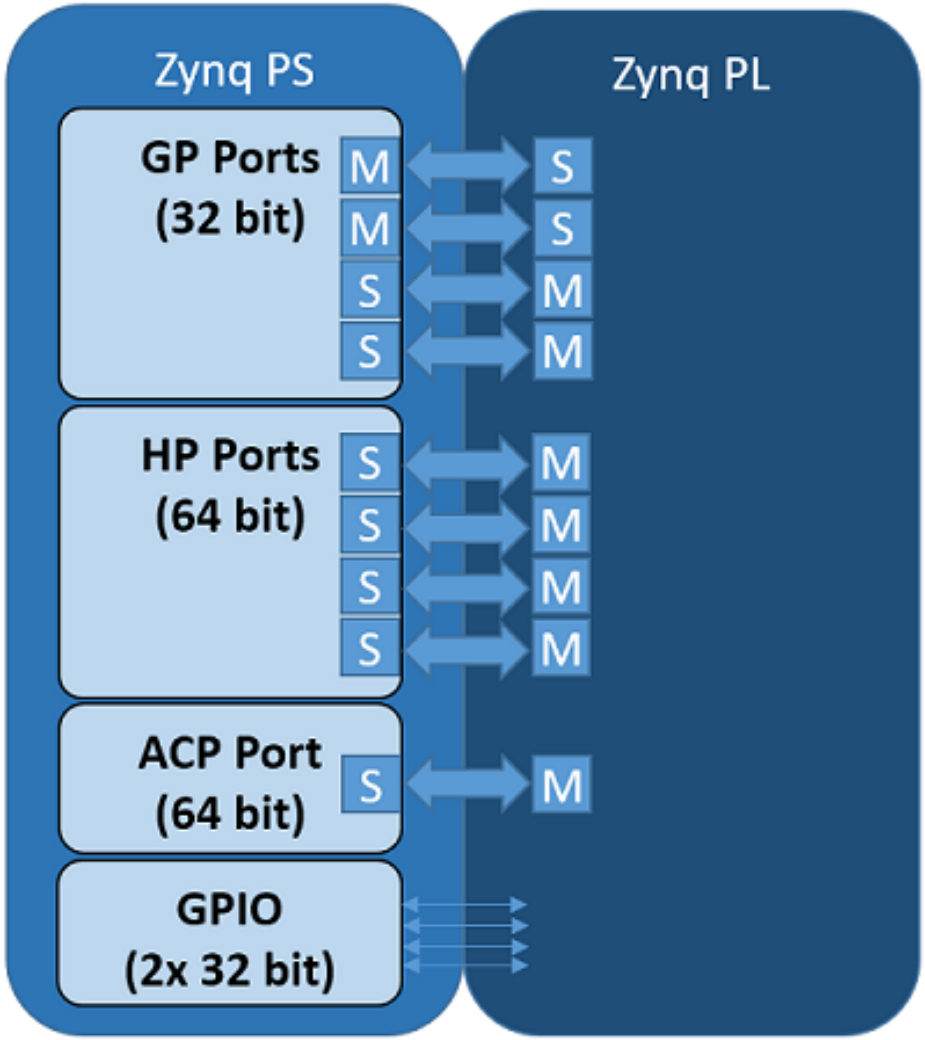

Figure: Zynq AXI Interfaces

HLS based design using PYNQ

- High Level Synthesis tools like Vivado HLS can be used to synthesize larger and compex designs and export the RTL as IP
- After exporting the IP, this IP can be used and block design is created in Vivado based on the interface we used.
- If the design is created only based on AXI4Lite, Zynq 7000 SoC IP and our design IP are to be connected and wrapper has to be created
- If the design accepts inputs in the form of stream, AXI4Stream interface will be used and a DMA has to be used to provide inputs in streaming fashion
- After validating the block design, HDL wrapper has to be created and bitstream file needs to be generated.

Example Design using AXI4Lite

- In the following block design (QR decomposition), the inputs to the IP are given using AXI4Lite (memory mapped I/O interface)

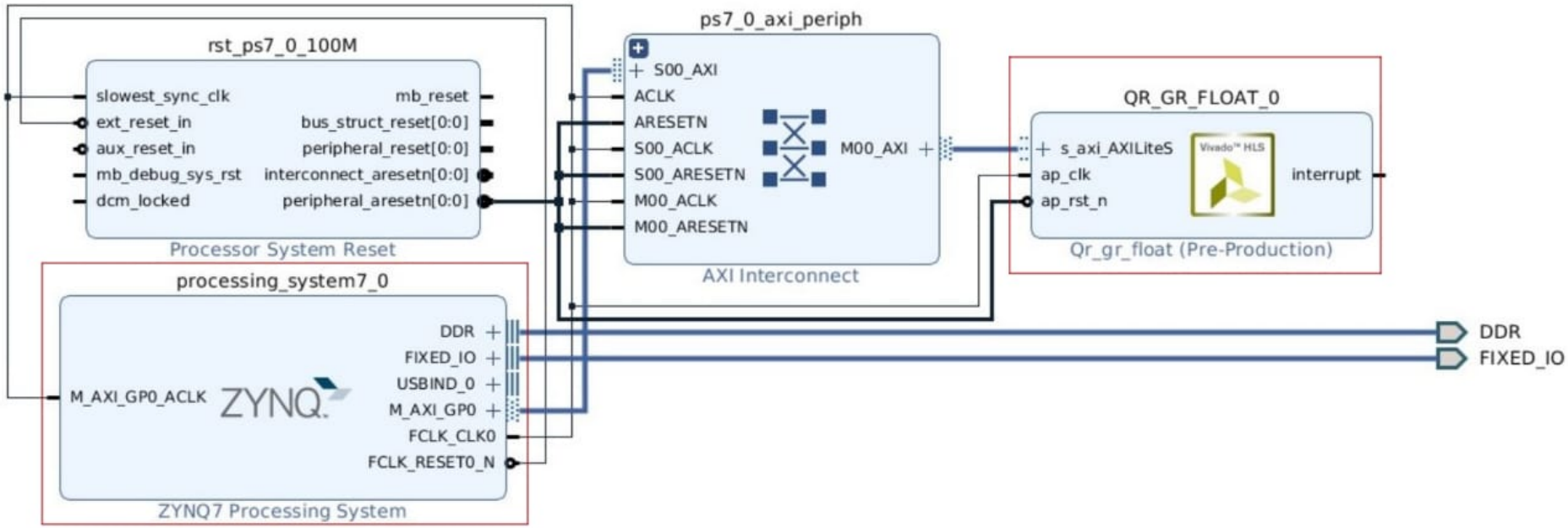

Figure: Block design for AXI4Lite interface

Example Design using AXI4Stream

- In the following block design (Digital Correlator), the inputs to the IP are given using AXI4Stream and other control signals to the IP using AXI4Lite (memory mapped I/O interface)

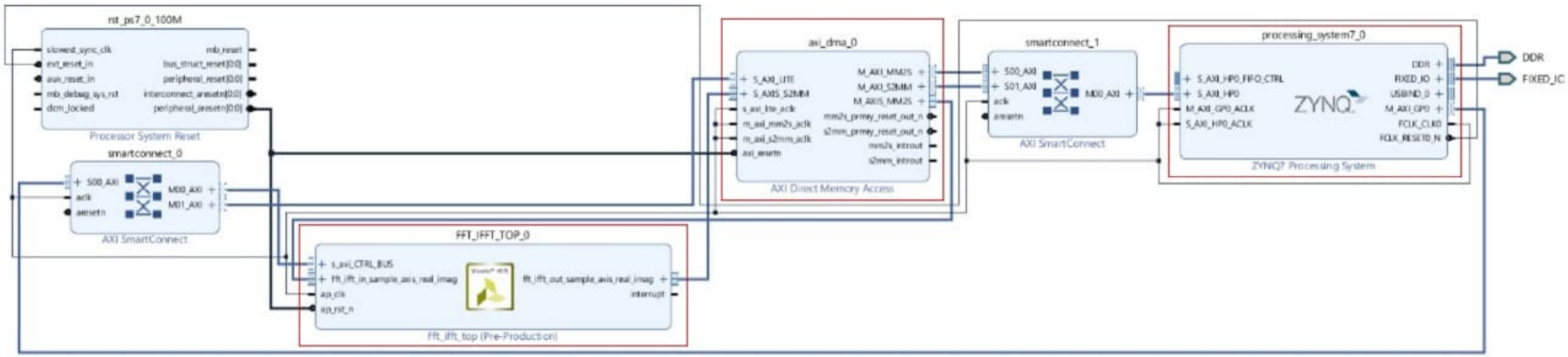

Figure: Block design for AXI4Stream interface

Table of Contents

Jupyter Notebook for PYNQ

- After PYNQ is setup, to test our design, we can open the jupyter notebook using the following ip address **192.168.2.99**
- The default password to this notebook is **xilinx**
- Our design can be tested using a python code where we create the overlay and provide inputs to our IP using the four classes mentioned before.
- For AXI4Lite signals, we provide inputs using MMIO and for AXI4Stream signals, we provide inputs using DMA

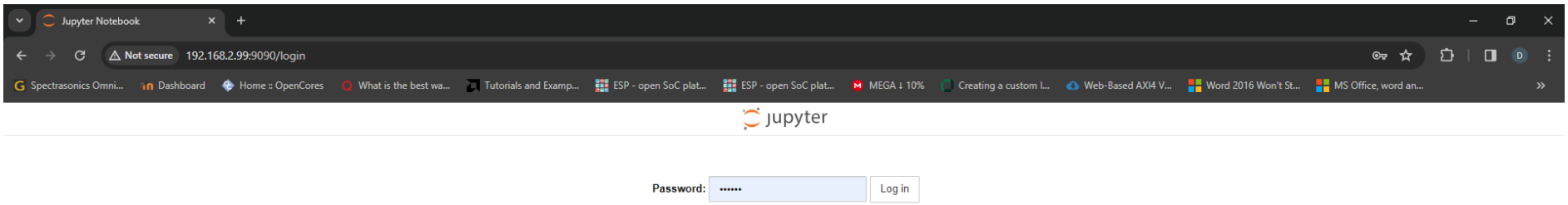

Figure: Jupyter Notebook

Files required for Testing Design on PYNQ

- The following files need to be uploaded in the jupyter notebook and python code has to be written which provide inputs and control signals to our design
  1. Bitstream file from Vivado (.bit file)
  2. Hardware handoff (.hwh file)
  3. Block design (.tcl file)
- The following code snippet shows how to create overlay and use our designed IP in the jupyter notebook

```
from pynq import Overlay
from pynq import MMIO
import numpy as np
import time
import struct
overlay = Overlay("./QR_GR_FLOAT.bit")
```

Figure: Creating Overlay

Python code for AXI4Lite Designs

- The following code snippet shows the register map and how to pass inputs to the design using base addresses.

```
overlay?
```

```
top_func = overlay.QR_GR_FLOAT_0
```

```
top_func.register_map
```

```
RegisterMap {
  CTRL = Register(AP_START=0, AP_DONE=0, AP_IDLE=1, AP_READY=0, RESERVED_1=0, AUTO_RESTART=0, RESERVED_2=0),
  GIER = Register(Enable=0, RESERVED=0),
  IP_IER = Register(CHAN0_INT_EN=0, CHAN1_INT_EN=0, RESERVED=0),
  IP_ISR = Register(CHAN0_INT_ST=0, CHAN1_INT_ST=0, RESERVED=0),
  Memory_A = Register(value=0),
  Memory_B = Register(value=0),
  dim = Register(dim=write-only),
  Memory_solve_vec = Register(value=0)
}
```

```
base_addr = 0x43C00000
addr_range = 0x10000
mmio = MMIO(base_addr, addr_range)
```

Figure: Register Map of IP

MMIO and Passing Inputs

```
mmio.write(0x40, struct.unpack('I', struct.pack('f', 3.0))[0])
mmio.write(0x44, struct.unpack('I', struct.pack('f', 2.0))[0])
mmio.write(0x48, struct.unpack('I', struct.pack('f', -1.0))[0])
mmio.write(0x4c, struct.unpack('I', struct.pack('f', -2.0))[0])
mmio.write(0x50, struct.unpack('I', struct.pack('f', 2.0))[0])
mmio.write(0x54, struct.unpack('I', struct.pack('f', 1.0))[0])
mmio.write(0x58, struct.unpack('I', struct.pack('f', 1.0))[0])
mmio.write(0x5c, struct.unpack('I', struct.pack('f', 1.0))[0])
mmio.write(0x60, struct.unpack('I', struct.pack('f', 1.0))[0])
mmio.write(0x80, struct.unpack('I', struct.pack('f', 6.0))[0])
mmio.write(0x84, struct.unpack('I', struct.pack('f', 3.0))[0])
mmio.write(0x88, struct.unpack('I', struct.pack('f', 4.0))[0])
mmio.write(0x90,3)
start_HW = time.time()
mmio.write(0x00,1)
stop_HW = time.time()
duration_HW = stop_HW - start_HW
```

Figure: Passing inputs to IP using MMIO

Python code for AXI4Stream Designs

- The following code snippets shows how to use DMA to send inputs and receive ouputs in the streaming fashion for the designed IP

```
base_addr = 0x43C00000
addr_range = 0x10000
mmio = MMIO(base_addr, addr_range)

overlay = Overlay('./ADC_CORRELATION.bit')

dma = overlay.axi_dma_0
dma_send = dma.sendchannel
dma_recv = dma.recvchannel
```

```
in_buffer = allocate(shape=(4096,),dtype=np.single)
out_buffer = allocate(shape=(2048,),dtype=np.single)

np.copyto(in_buffer,out)

print(len(out))
start_time = time.time()
dma_send.transfer(in_buffer)
dma_recv.transfer(out_buffer)
stop_time = time.time()
HW_Exec_Time = stop_time - start_time
y1 = list(out_buffer)
print(y1)
```

Figure: Sendchannel and RecvChannel for DMA

Conclusion

- Along with the above mentioned, PYNQ Z2 is also used for many other embedded applications.
- PYNQ Z2 is a smaller board while this PYNQ framework is supported by other bigger boards like ZCU104
- This enables us to prototype larger and complex designs using PYNQ framework

Thank You!