Python Productivity for Zynq

Parimal Patel
XUP Senior Systems Engineer

Platform evolution

Raspberry PI

Desktop Linux on ARM Microprocessors

Arduino

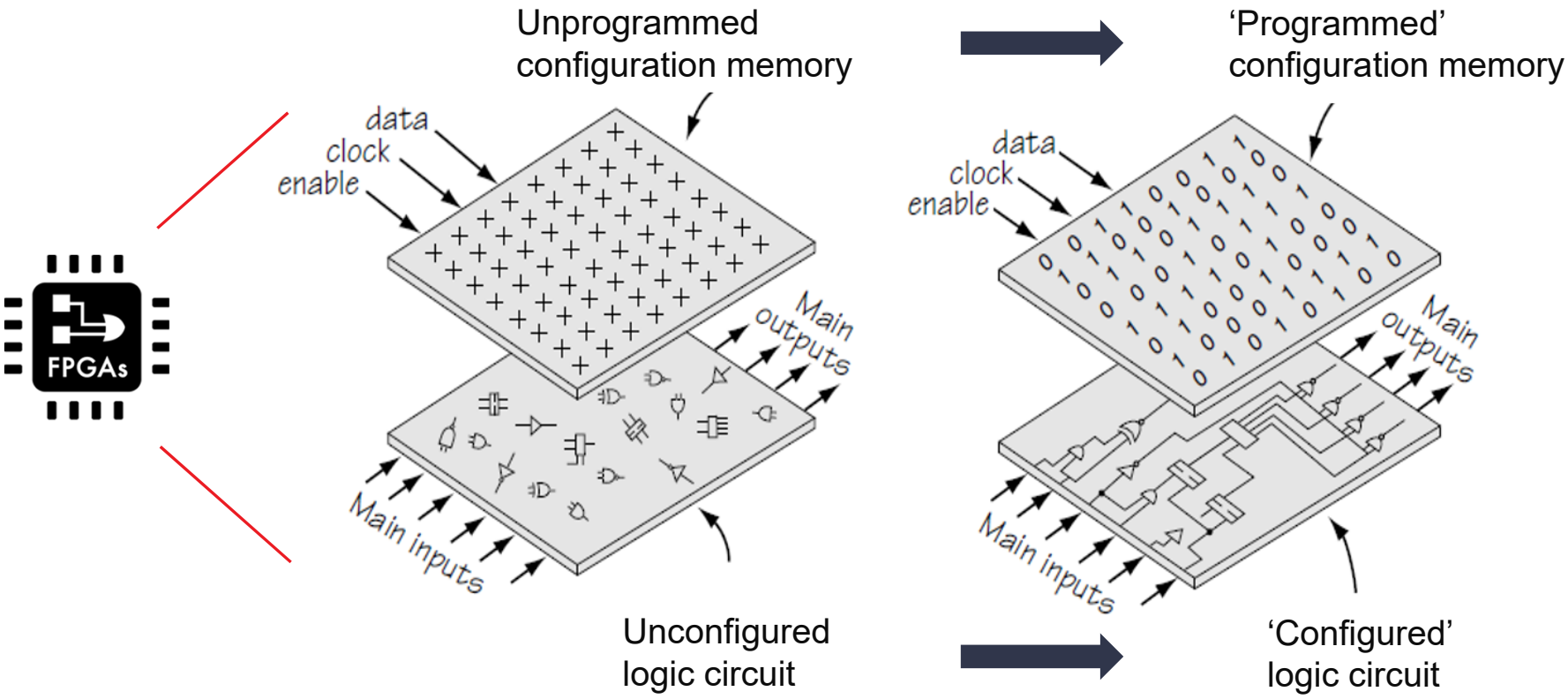
Low-level, ‘bit-banging’ microcontroller

FPGA

Fast, parallel, customizable logic

XILINX.

Field Programmable Gate Arrays (FPGAs)



XILINX.

Microprocessors, microcontrollers and FPGA logic, integrated with high bandwidth interconnect & IO

Platform integration

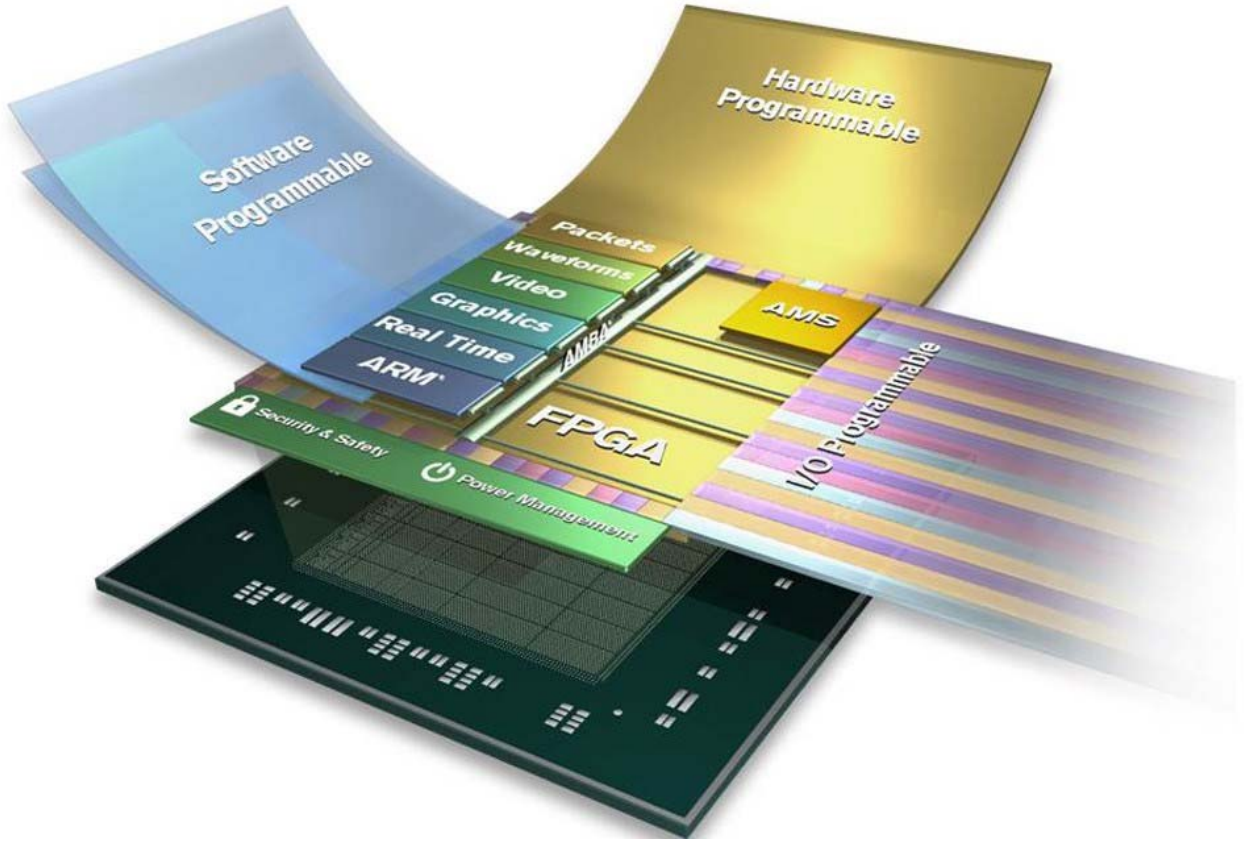
The Zynq/ZynqUltraScale+ family integrates

- ARM microprocessors
- Programmable logic (FPGA)
- High-speed IO
- Multiple soft micro-controllers (optional)

With high bandwidth connections between all the components

… and dynamic reconfiguration

Xilinx ZYNQ UltraScale+

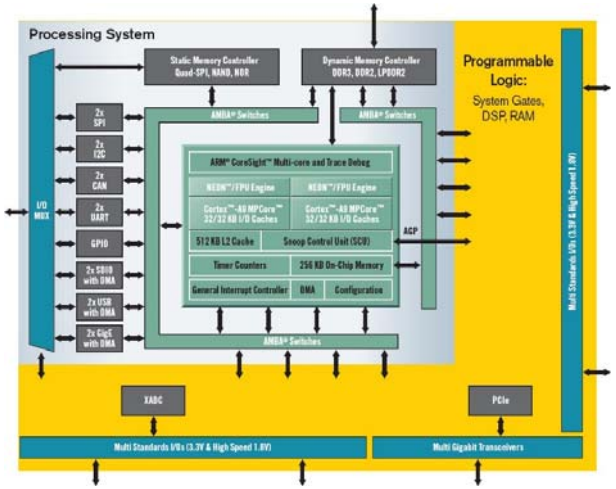


XILINX.

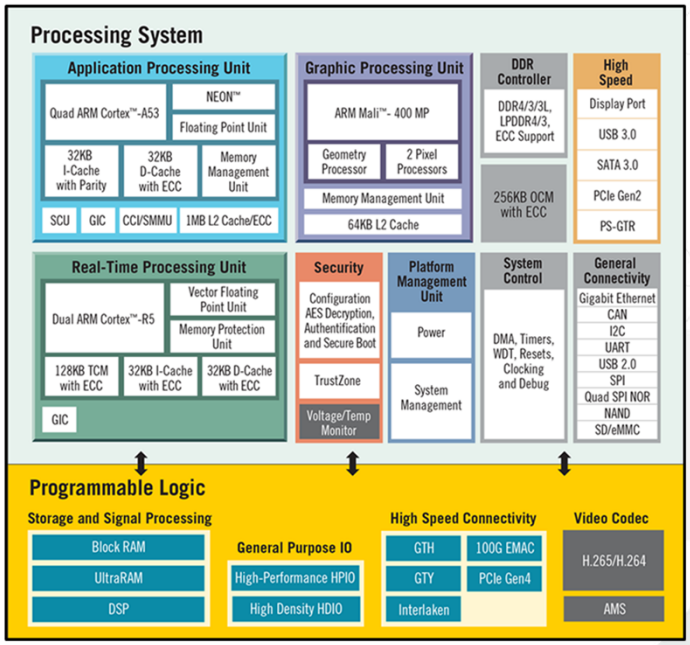
ZYNQ and ZYNQ UltraSCALE+

Best-in-class, All Programmable SoCs

ZYNQ 7000



ZYNQ UltraSCALE+



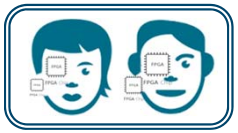
FPGAs and tightly-integrated CPUs enable entirely new opportunities

XILINX.

PYNQ Python Productivity on Zynq

Domain Experts
Software Engineers

Embedded software
Engineers



Hardware
Engineers

New users are not always hardware designers, or embedded systems designers

PYNQ

Enables more people to program Xilinx processing platforms, more productively

AND

Offers more rapid development for h/w designers and embedded s/w engineers

XILINX.

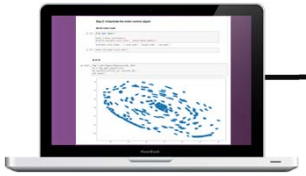
PYNQ Vision

> **Make Zynq so easy-to-use that programmers can access the benefits of FPGAs without learning advanced digital design skills**

Python Productivity for Zynq

Jupyter notebooks,
browser-based interface

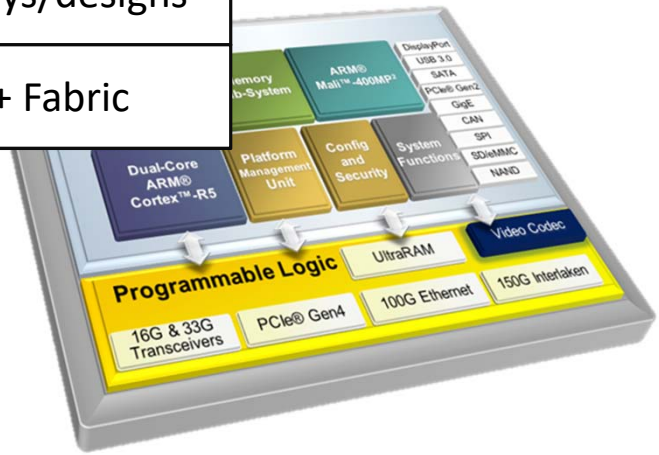
Most competitive, open
and advanced GUI



PYNQ enables JupyterLab
on Zynq and ZU+

Jupyter web server	
IPython kernel	
Ubuntu-based Linux	Overlays/designs
ARM A9 / A53	ZU+ Fabric

Hardware C-drivers wrapped in Python packages



XILINX.

PYNQ is a Framework

Applications

Jupyter / IPython

PYNQ notebooks

matplotlib NumPy scikit-learn - - - - OpenCV

Software

Python

PYNQ libraries

dma

overlay

Xlnk

PL GPIO Interrupt MMIO libcma.so

Linux Kernel

fpga_ manager sysgpio uio devmem xlnk

Hardware

axi_intc

FPGA

user designs

PYNQ overlays

PYNQ IPs

Apps

APIs

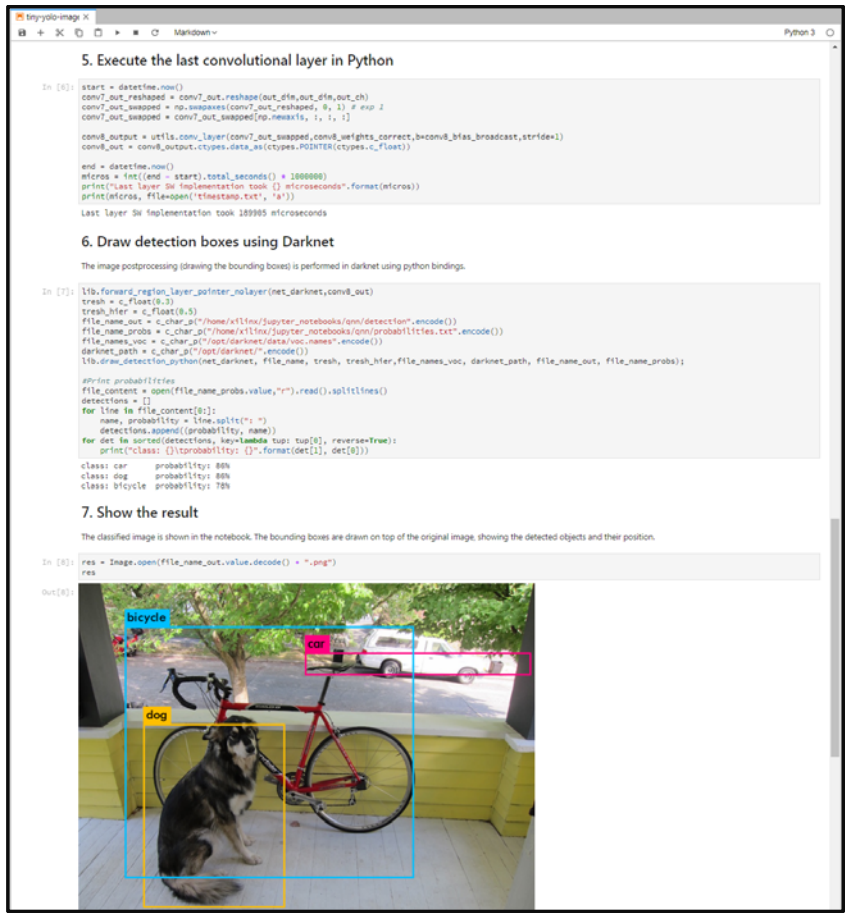
Drivers

Bitstreams

PYNQ

XILINX

Jupyter Notebooks … the engine of data science



jupyter

Open source browser-based, executable documents

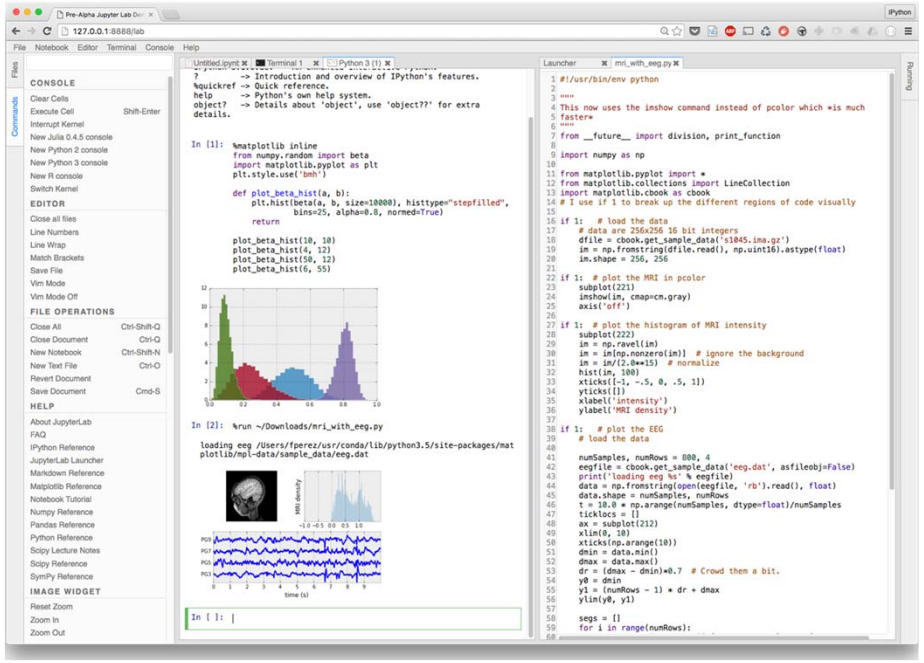
Live code, text, multimedia, graphics, equations, widgets …

1.7 million notebooks on GitHub

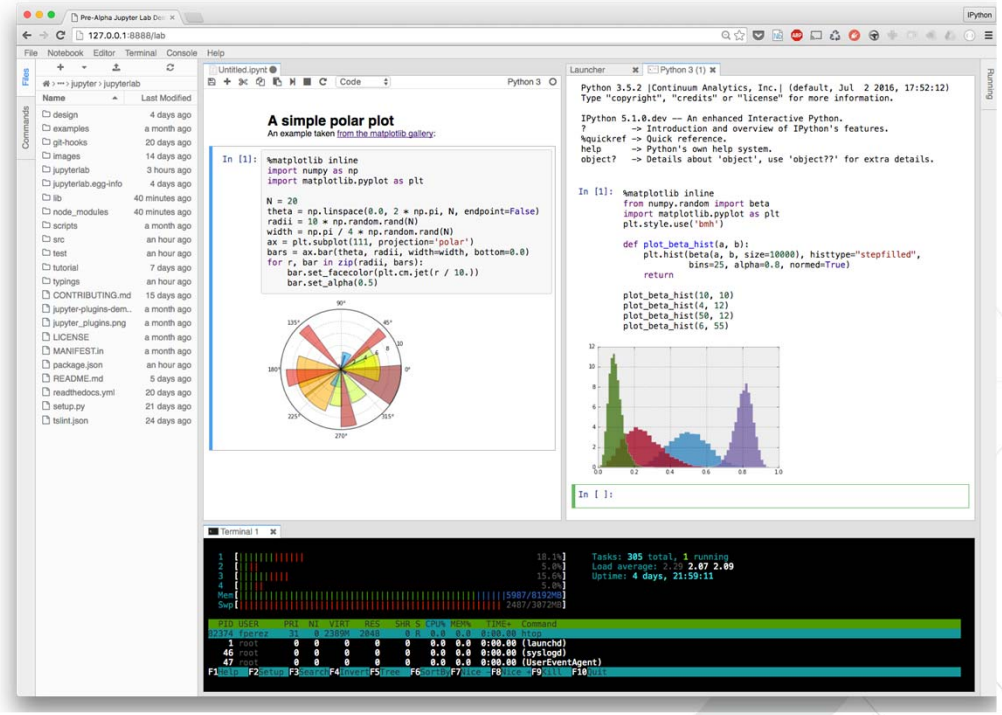
Taught to 1,000+ Berkeley data science students

XILINX.

JupyterLab: web-based IDE incl. Notebooks

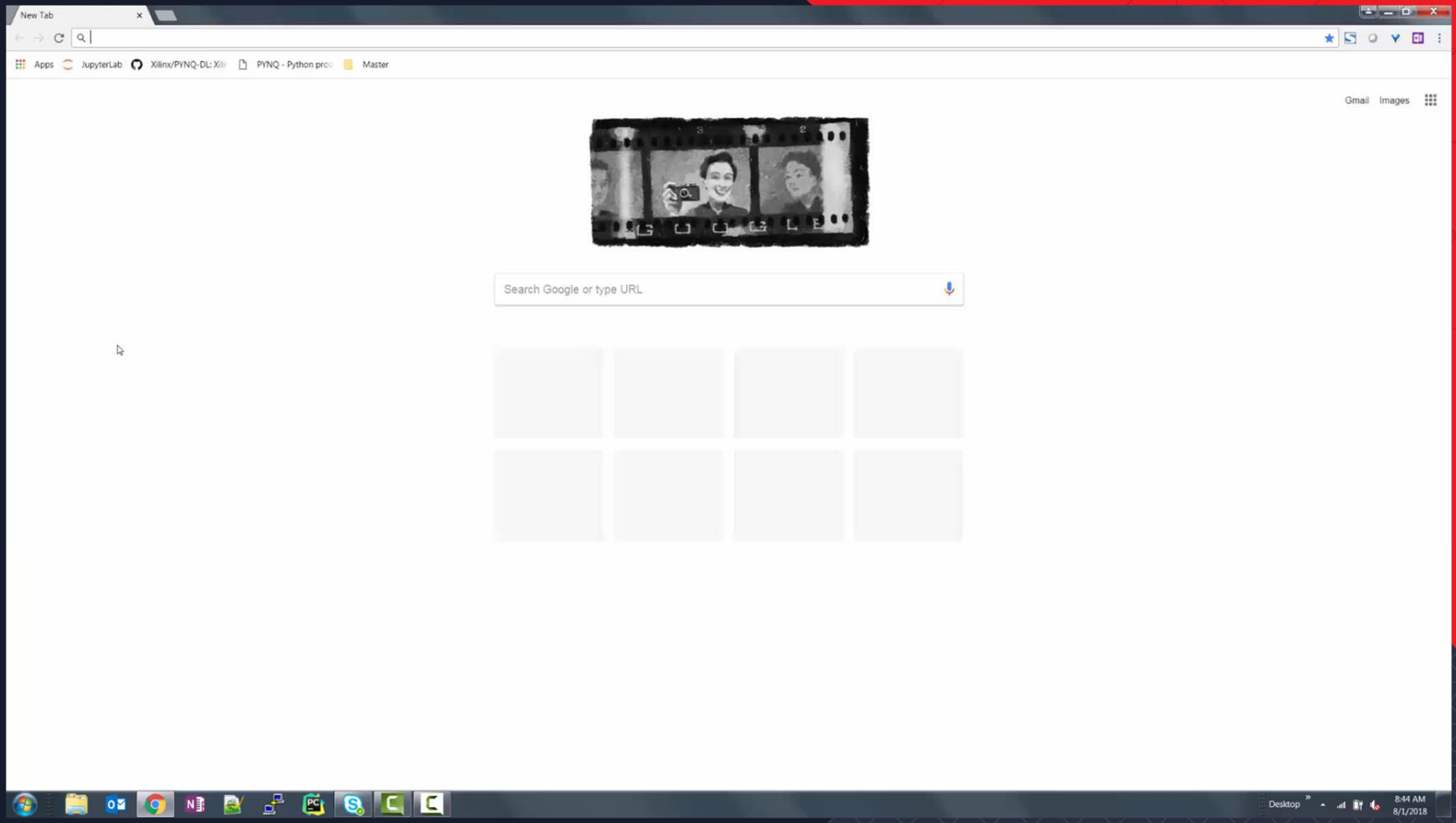


Jupyter Notebook is now one of many plug-ins within the JupyterLab integrated development environment



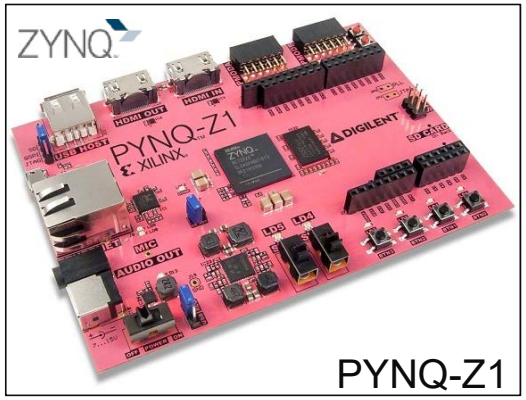
JupyterLab - an open-source, extensible IDE in a browser

XILINX.

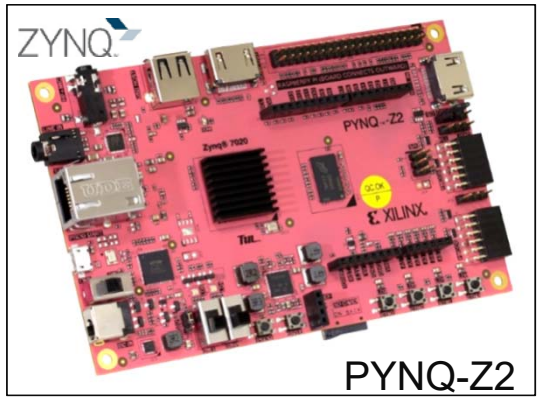

New Tab
Apps
JupyterLab
Xilinx/PYNQ-DL: Xili
PYNQ - Python proc
Master
Gmail
Images
Search Google or type URL
Desktop
8:44 AM
8/1/2018

PYNQ-enabled boards

- **Python productivity for Zynq**
  - Open source
  - Build image for other Zynq boards
- **Downloadable SD card image**
  - Zynq 7000
    - PYNQ-Z1 (Digilent)
    - PYNQ-Z2 (TUL)
  - Zynq Ultrascale+
    - ZCU104 (Xilinx)



PYNQ-Z1



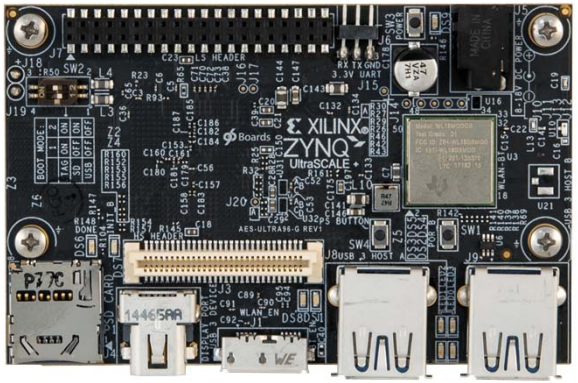
PYNQ-Z2

ZCU104

XILINX

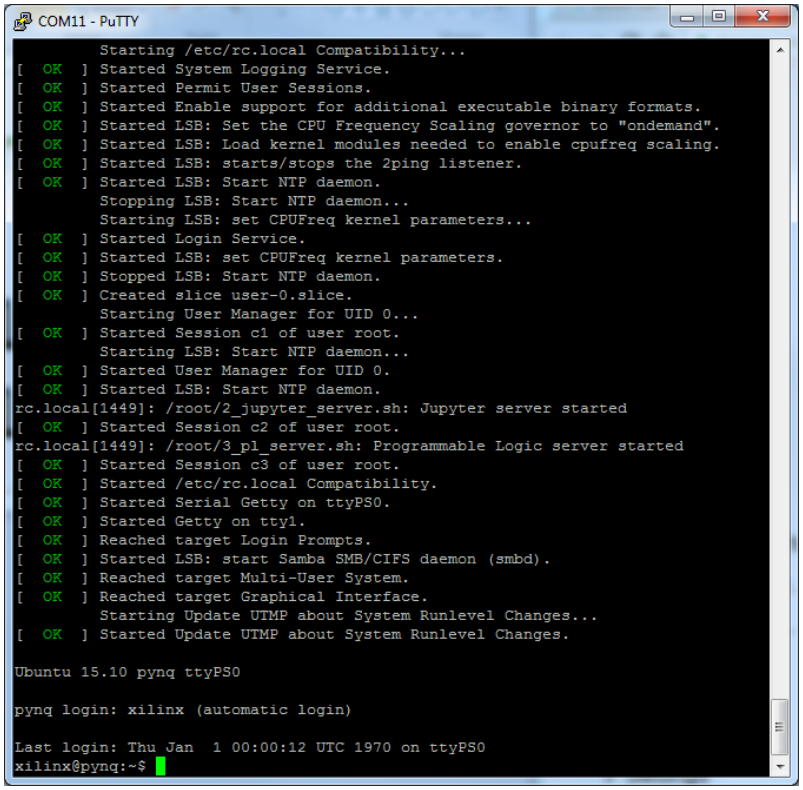
PYNQ-enabled board from our partners …

ULTRA96 Board



AVNET®
Reach Further™

Start using PYNQ out-of-the-box



- **PYNQ delivered as downloadable SD card image**
  - Linux preconfigured
- **Additional packages and drivers pre-installed**
  - USB peripheral drivers: webcams, wifi modules …
- **PYNQ is for Zynq**
  - PYNQ image is portable to other Zynq boards

Start using Zynq out of the box ✓

Simplify downloading bitstreams to PL

- **PYNQ 'Overlay' class**
  - Simplifies downloading bitstream
  - two lines of code
  - No Xilinx tools required
- **Maintain many bitstreams on the SD card**
  - E.g. multiple different demos
- **Can execute Python in browser, or from command line**

```
from pynq import Overlay

ol = Overlay('gray.bit')
```

Simply and fast way to configure Programmable Logic ✓

Simplify IP debug and prototyping

- **Debug of IP typically uses C/C++**
- **SDK tools used to:**
  - Compile test application
  - Download application to board
  - Step through code
- **PYNQ MMIO class allows peek/poke of IP registers from Python**
  - Python executes directly on the board
  - No offline compilation, download loop
  - No SDK tools

C code – compile, debug from host

```
/*****************************************************************************/
* This function does a selftest on the IIC device and XIic driver as an
* example.
* @param    DeviceId is the XPAR_<IIC_instance>_DEVICE_ID value from
*       xparameters.h.
*****************************************************************************/
int IicSelfTestExample(u16 DeviceId)
{
    Status = XIic_CfgInitialize(&Iic, ConfigPtr, ConfigPtr->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* Perform a self-test to ensure that the hardware was built */
    Status = XIic_SelfTest(&Iic);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
```

Python executes directly on target

```
from pynq import MMIO

# Map registers to MMIO instance
my_ip = MMIO(my_ip_addr, LENGTH)

# Write 0x1 to start IP
my_ip.write(CONTROL_REGISTER, 0x1)
# Check status register
my_ip.read(STATUS_REGISTER)
```

Rapid testing and prototyping ✓

pynq.io

Home | Get Started | PYNQ-Z1 Board | Community | Source Code | Support

Community Projects

Tutorials and other resources

Example Notebooks

PYNQ community

Tutorials and other resources

Ultra96 Facial Recognition Deadbolt Using PYNQ
Julian Bartolone

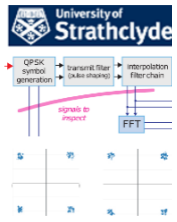
PYNQ Tutorial workshop

HLS filter example
(FPGA Developer)

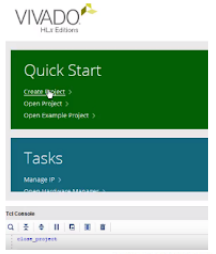
PYNQ HLS tutorial
(Dustin Richmond)

Accelerate FIR function
(FPGA Developer)

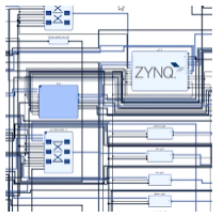
PYNQ RFSoC
University Strathclyde
QPSK demo on ZCU111



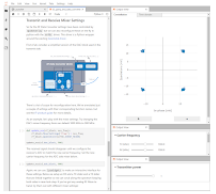
Video: Control custom IP using GPIO



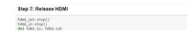
Video: Add existing IP to a PYNQ overlay



PYNQ



PYNQ



XILINX.

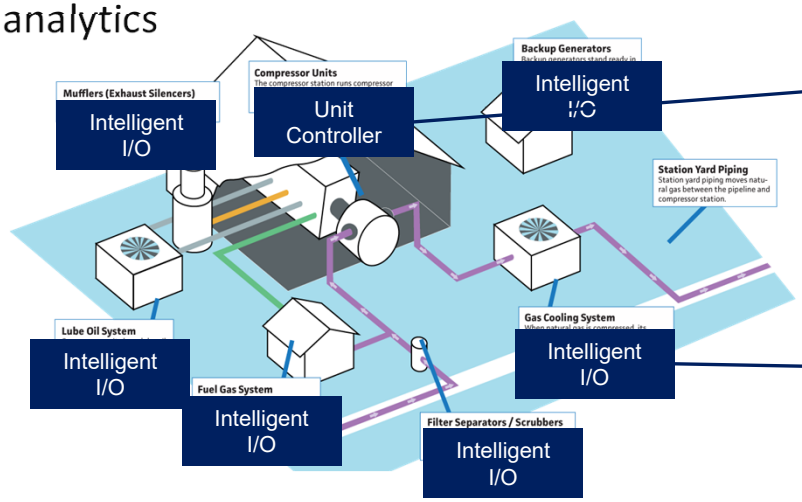
Case Study - Industrial Controls IIoT

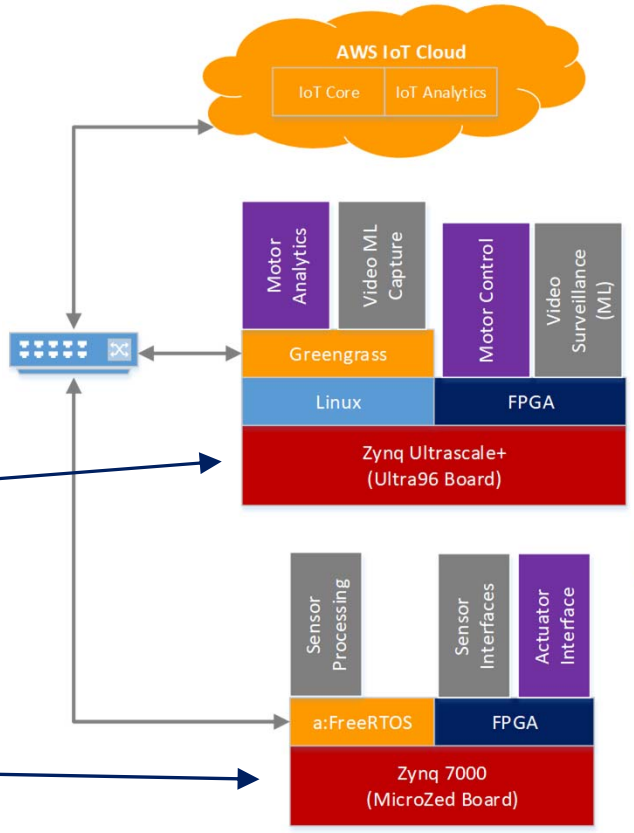
Zynq 7000 w/ AWS FreeRTOS

- Targets limited resource devices
- Often bridge physical & digital domains

Zynq US+ w/ Linux & Greengrass

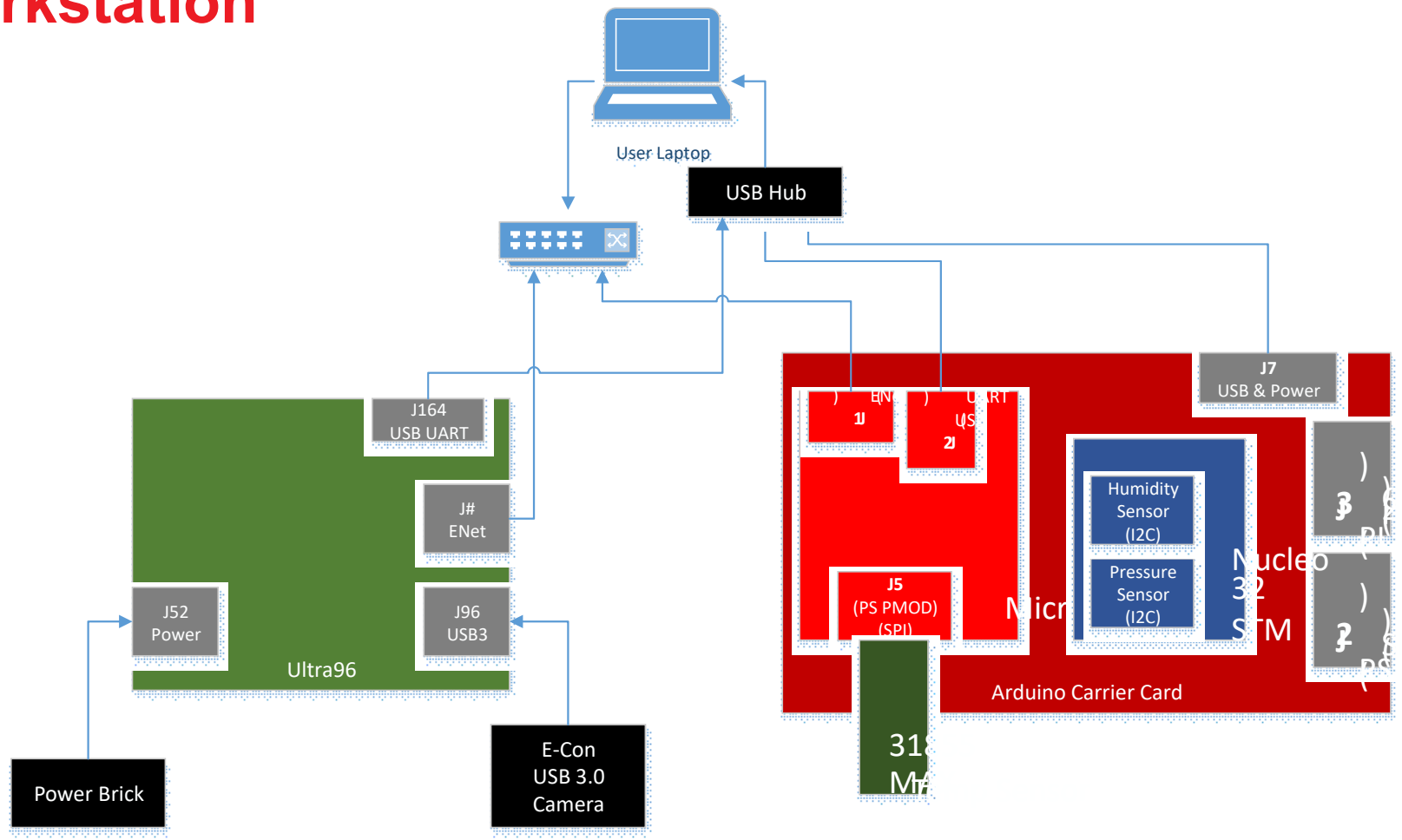
- Targets more capable embedded devices
- Brings together control action & local analytics





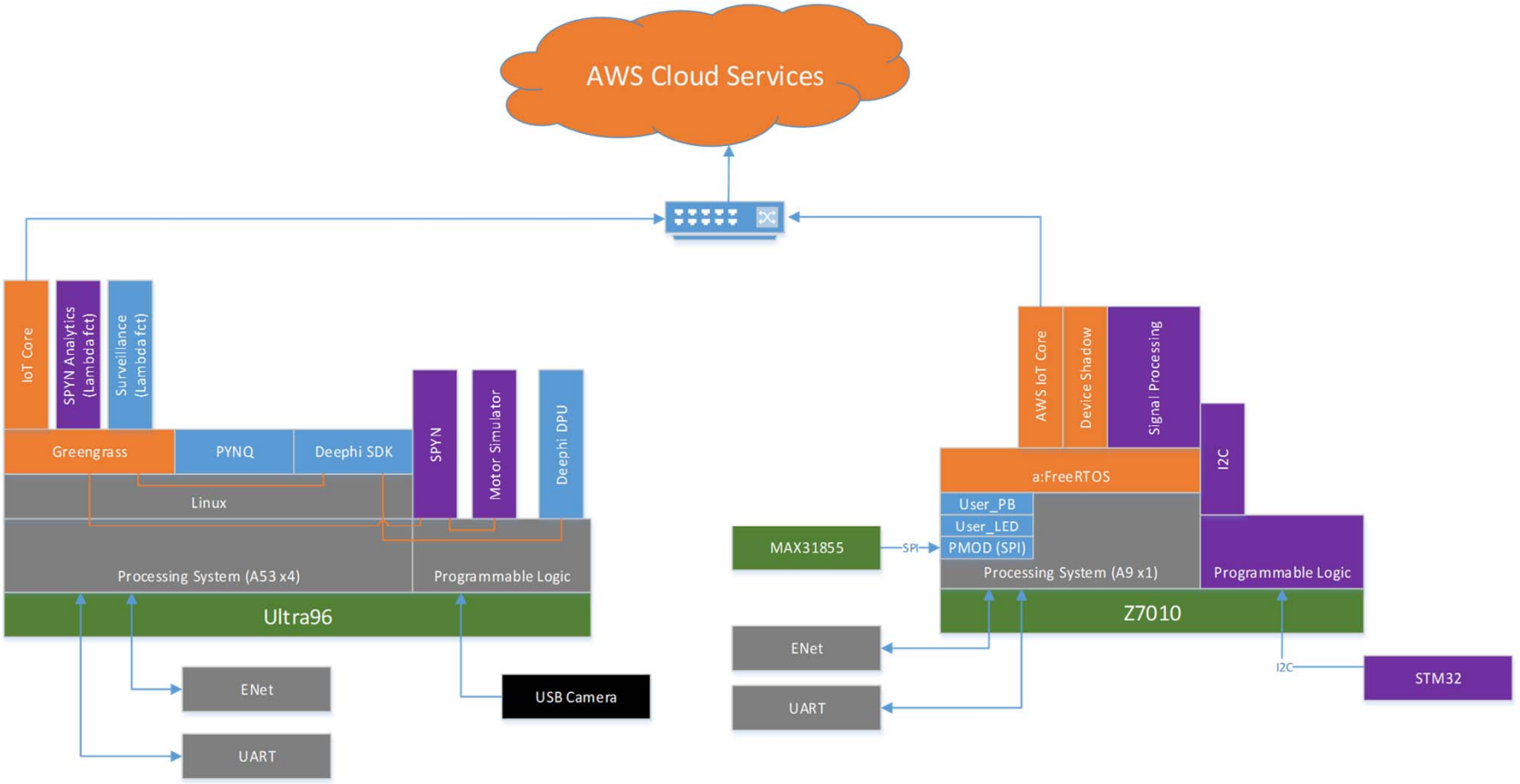
XILINX.

Workstation



XILINX.

Configuration



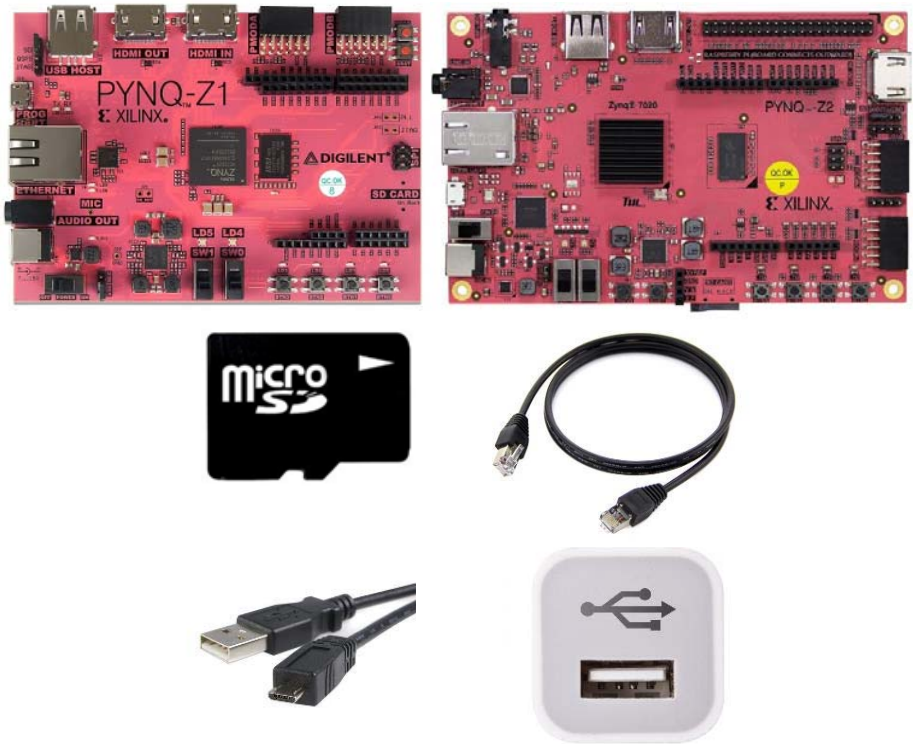
XILINX.

Getting Started

XILINX

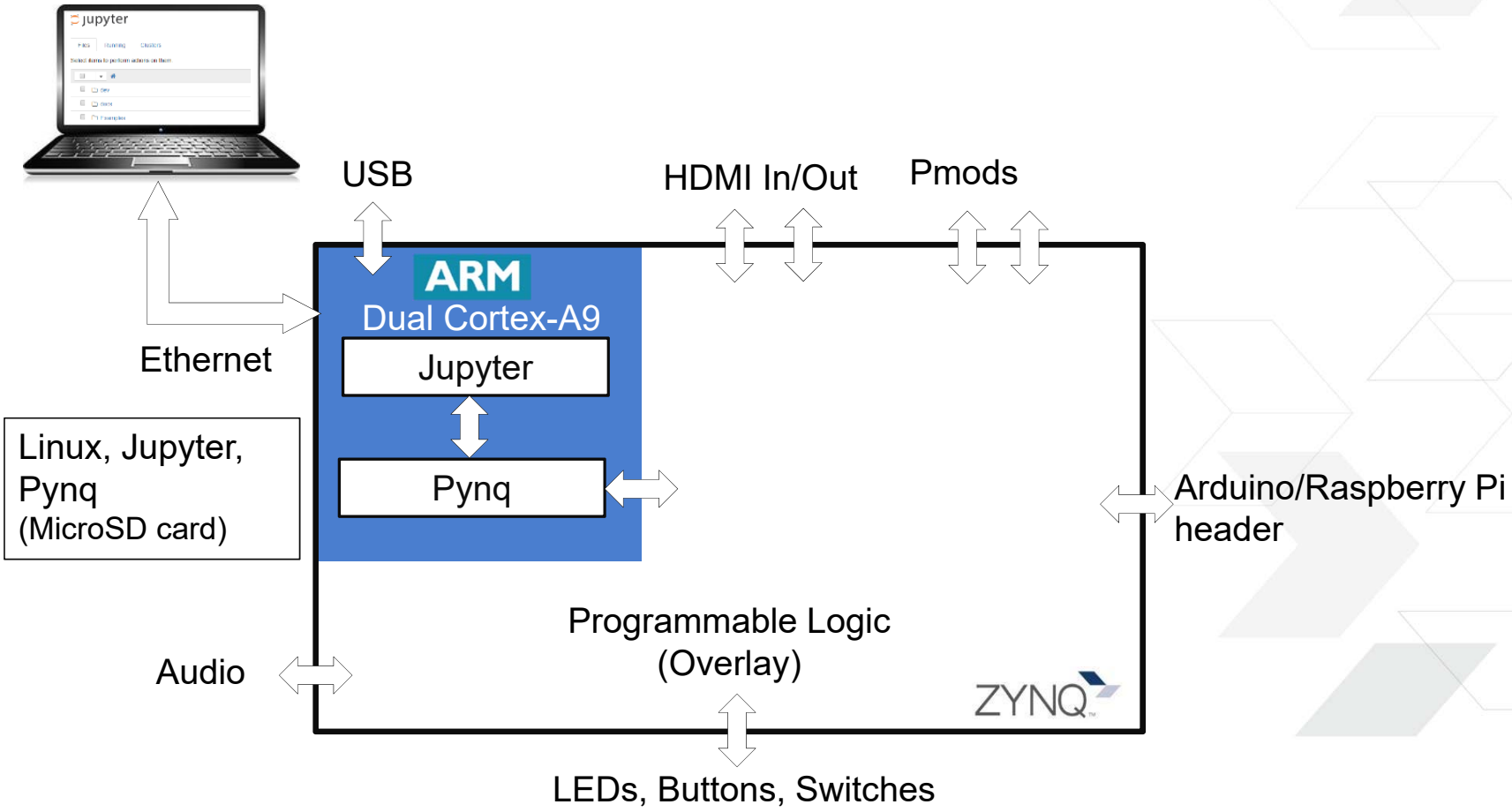
PYNQ

Prerequisites



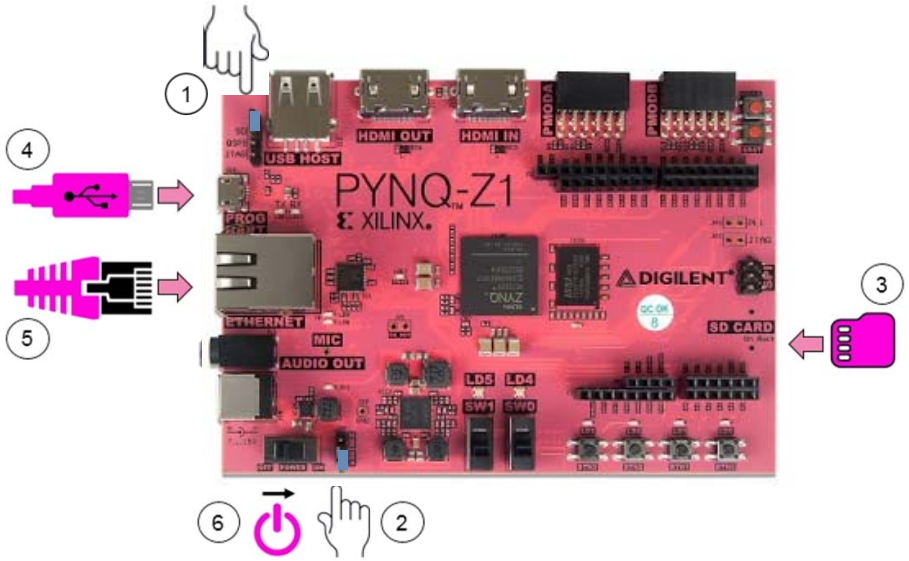
*Instructions to download and prepare SD card:
http://pynq.readthedocs.io/en/latest/getting_started.html

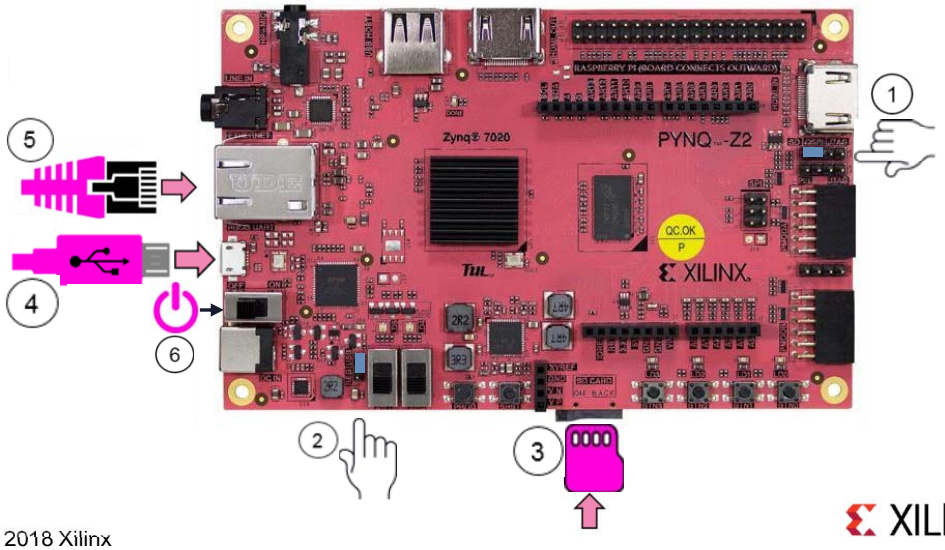
Pynq overview (PYNQ-Z1/PYNQ-Z2)



Connecting to the board

1. Configure board to boot from SD Card
2. Set Jumper to power from USB
3. Insert SD Card
4. Connect USB cable
5. Connect Ethernet cable to PC or to a Switch/Router
6. Power On

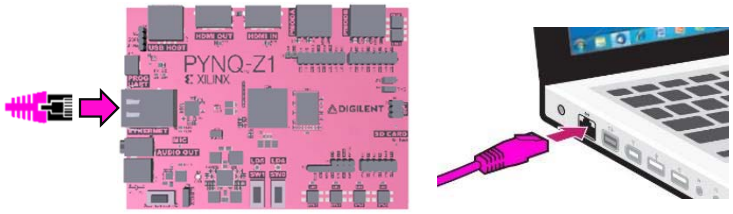




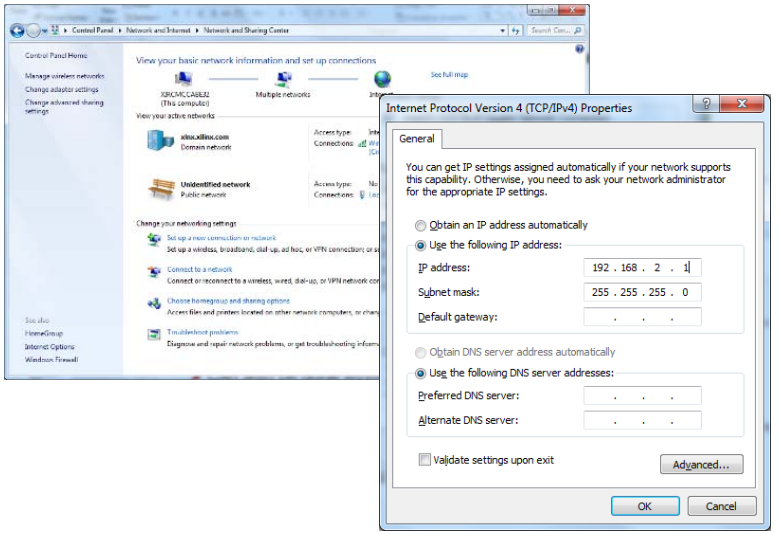
XILINX

Connecting to the board – Direct connection

- **Connect board directly to Ethernet port on PC**
  - USB to Ethernet adapter if no Ethernet port available
- **Board IP will default to 192.168.2.99**
- **Manually specify static IP for PC**
  - Must be in same range as board:
    - E.g. 192.168.2.1
  - No internet access unless you bridge network connections



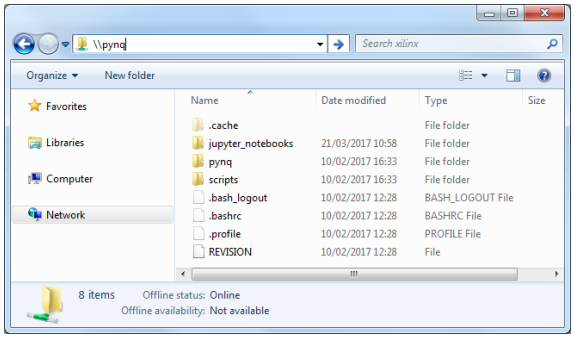
Connect board directly to PC

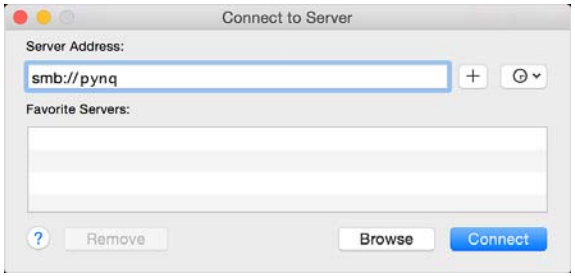


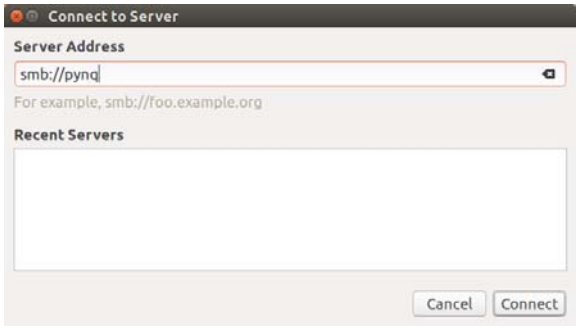
XILINX.

Samba share

- **Board can also be accessed as a shared drive**
  - Windows: \\192.168.2.99\xilinx
  - Linux: smb://192.168.2.99/xilinx
  - MAC OS: smb://pynq/xilinx
    - Hit Command+K to bring up the 'Connect to Server' window
- **Log-in**
  - Username = xilinx
  - Password = xilinx
- **Copy files easily between PC and board**



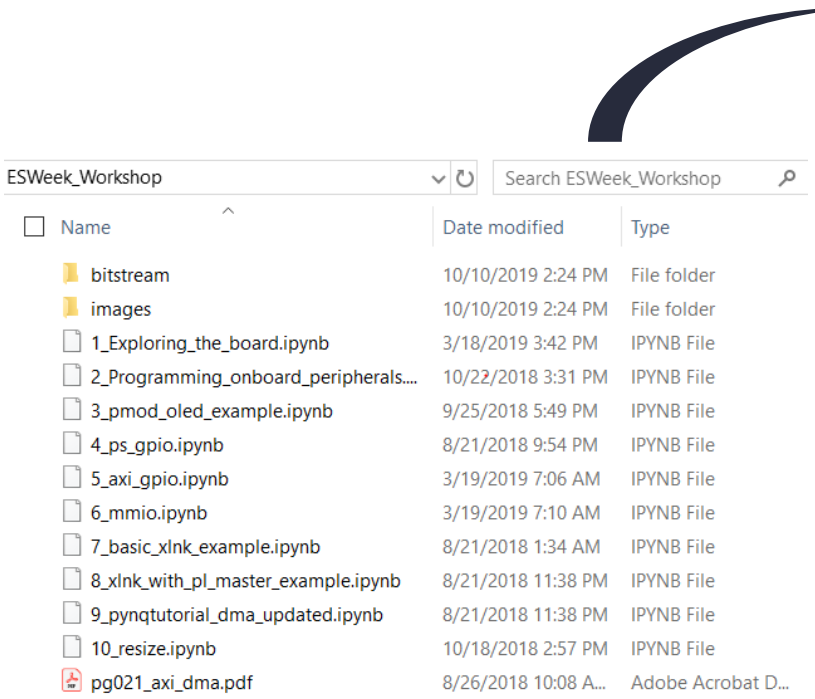




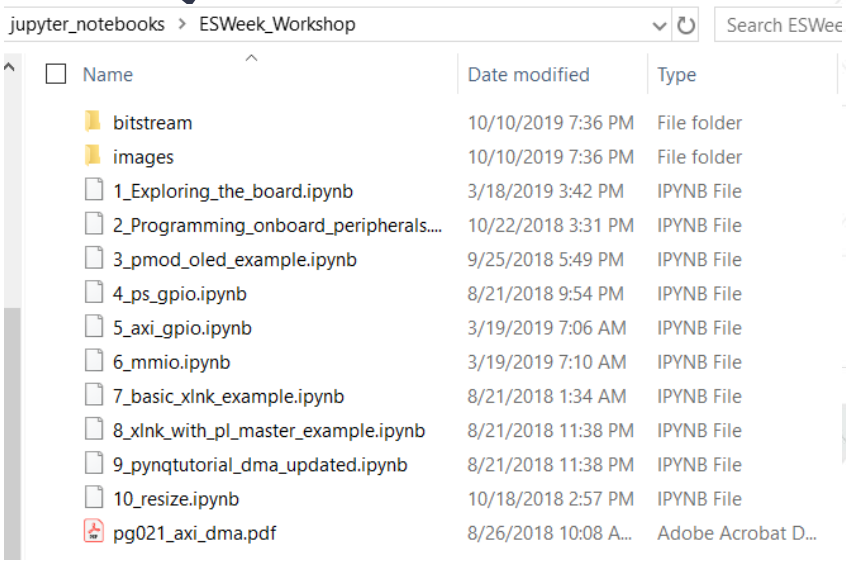
XILINX.

Introduction to Jupyter notebooks

> **Copy workshop files to the board: \\192.168.2.99\xilinx\jupyter_notebooks**



From laptop

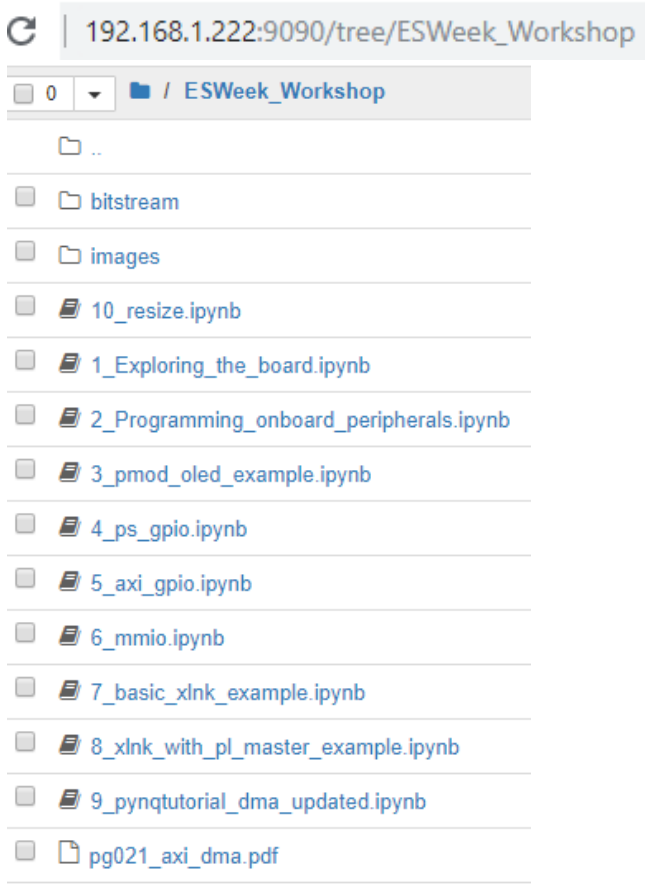


To the board

XILINX.

Log in to Jupyter portal

- **Open a browser***
  - Chrome (preferred)
- **Browse to: http://*192.168.2.99*:*9090***
- **password = xilinx**
- **Browse to *ESWeek_Workshop* folder**



*http://jupyter-notebook.readthedocs.io/en/latest/notebook.html#browser-compatibility

The board doesn't have a realtime clock. The first time a board is used, the date and time of the system may be too far out of sync, and cause some browser (e.g. FireFox) to refuse setting a cookie which prevents log-in to the board. Chrome does not have this issue. To resolve this issue, update the date on the board. In a terminal execute: sudo date +%Y%m%d -s "20180920 "
"20180920" is YYYYMMDD

Lab exercises: Session 1

- **Exploring the board**
  - Getting CPU information
  - Getting network status
- **Programming on-board peripherals**
  - Controlling on-board LEDs
  - Interacting with buttons, switches, and LEDs

Overlays and IOPs

XILINX

PYNQ

FPGA overlays – hardware libraries

> **Overlays are generic FPGA designs that target multiple users with new design abstractions and tools**

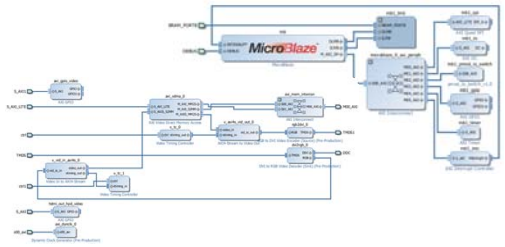
> **Overlay characteristics**

- Post-bitstream programmable via software APIs
- Typically optimized for given application domains
- Encourages the use of open source tools & fast compilation
- Enables productivity by re-using pre-optimized designs
- Makes benefits of FPGAs accessible to new users

Anatomy of an overlay IP subsystem

- **Designed to be immediately reused by anyone**
  - or re-purposed elsewhere by “person skilled in the art” (PSITA)
- **Comprises**
  - Programmable FPGA IP core
  - FPGA bitstream
  - C code to expose programmable functionality
  - Python-to-C bindings
  - Python library with API
  - Protocol
  - Jupyter notebook examples

FPGA overlays – hardware libraries



Step 1:
Create an FPGA design for a class of related applications

```
pmod_init(0,1);
while(1){
    while((MAILBOX_CMD_ADDR & 0x01
    cmd=MAILBOX_CMD_ADDR;

    count = (cmd & 0x0000ff00) >>
    if((count==0) || (count>253))
        // clear bit[0] to indicat
        // set rest to 1s to indic
        MAILBOX_CMD_ADDR = 0xffff
        return -1;
    }
    for(i=0; i<count; i++) {
        if (cmd & 0x08) // Python
        {
            switch ((cmd & 0x06) >> 1) { // use bit[2:1]
                case 0 : MAILBOX_DATA(i) = *(u8 *) MAILBOX_ADDR; break;
                case 1 : MAILBOX_DATA(i) = *(u16 *) MAILBOX_ADDR; break;
                case 2 : break;
                case 3 : MAILBOX_DATA(i) = *(u32 *) MAILBOX_ADDR; break;
            }
```

```
6c78 3963 7367 3232 3500 6300 0b32
332f 3039 2f33 3000 6400 0931 323a
3a31 3900 6500 0532 7cff ffff ffff
ffff ffff ffff ffff ffaa 9955 6630
0720 0031 a103 8031 413d 0831 6109
c204 0010 9330 e100 cf30 c100 8120
0020 0020 0020 0020 0020 0020 0020
0020 0020 0020 0020 0020 0020 0020
813c c831 8108 8134 2100 0032 0100
e1ff ff33 2100 0533 4100 0433 0101
6100 0032 8100 0032 a100 0032 c100
```

Step 2:
Export the bitstream and a C API for programming the design

```
void setNormalDisplay(){
    sendCommand(OLED_Normal_Display_Cmd);
}

void setInverseDisplay(){
    sendCommand(OLED_Inverse_Display_Cmd);
}

int main(void)
{
   int cmd;
   int Row, Column;

   arduino_init(0,0,0,0);
   config_arduino_switch(A_GPIO, A_GPIO, A_GPIO,
                         A_GPIO, A_SDA, A_SCL,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO, D_GPIO,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO);

   // Initialization
   oled_init();
```

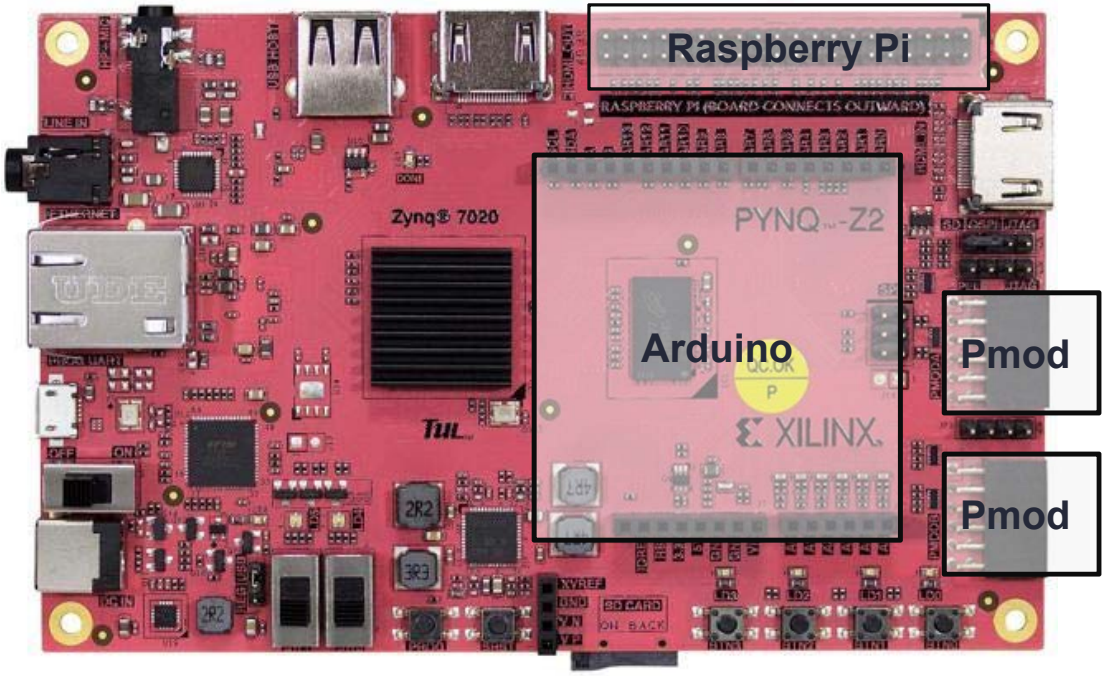
Step 3:
Wrap the C API to create a Python library

```
from time import sleep
from pynq import Overlay
from pynq.iop import PMOD_ADC, PMOD_DAC

ol = Overlay("base.bit")
ol.download()
# Writing values from 0.0V to 2.0V with step 0.1V.
dac_id = int(input("Type in the PMOD ID of the DAC (1 ~ 2): "))
adc_id = int(input("Type in the PMOD ID of the ADC (1 ~ 2): "))

dac = PMOD_DAC(dac_id)
adc = PMOD_ADC(adc_id)

for j in range(20):
    value = 0.1 * j
    dac.write(value)
    sleep(0.5)
#    readings=adc.read(1,0,0)
#    x1=readings[0]
    print("Voltage read by DAC is: {:.4f} Volts".format(adc.read(1,0,0)[0]))
```

Step 4:
Import the bitstream and the library in your Python scripts and program

XILINX.

Low-cost PYNQ boards: Pmod, RPi, Arduino Interfaces



Typically every new Pmod, Raspberry Pi, or Arduino module requires a new design/seperate bitstream

Grove: Wide range low-cost sensors, actuators, etc

Environmental Monitoring

Have you ever wanted to get your daily weather report based on data from your garden instead of obtaining a more generic report from your TV or mobile phone? Sensors

Grove - Digital Light Sensor | Grove - Light Sensor | Grove - Temperature and | Grove - Barometer Sensor | Grove - Dust Sensor

Motion Sensing

Sensors in this category enable your microcontroller to detect motion, location and direction. You can make the movement of your microcontroller understandable in three dimensional spaces

Grove - 3-Axis Digital Compass | Grove - 3-Axis Digital Accelerometer(±1.5g) | Grove - 3-Axis Digital Gyro | Grove - Collision Sensor | Grove - 3-Axis Analog Accelerometer

Wireless Communication

Communicating without wires is a cool feature that can spice up your project. Modules in this category arm your microcontroller with wireless communication ability such as RF, Bluetooth, etc.

Grove - 315MHz Simple RF Link Kit | Grove - Serial RF Pro | Grove - GPS | Grove - 125KHz RFID Reader | Grove - Serial Bluetooth

User Interface

Modules in this, our largest, category, let you interface with your microcontroller via input modules, such as touch pads, joysticks or your voice. Or you can choose output modules,

Grove - Solid State Relay | Grove - OLED Display 128*64 | Grove - Serial LCD | Grove - LED Socket Kit | Grove - Button

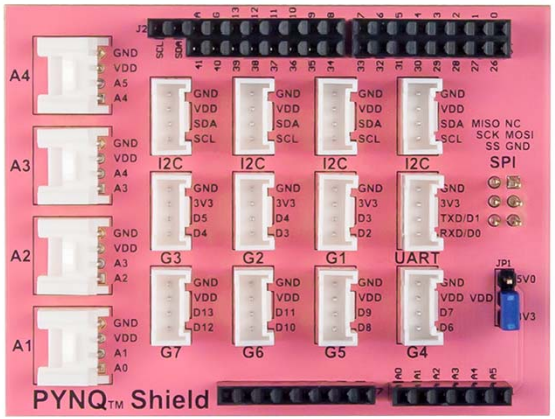
Physical Monitoring

Scientists understand the world around us in physical dimensions. Modules in this category are designed to help you analyze the physical world. Measure your heart rate,

Grove - Water Sensor | Grove - Magnetic Switch | Grove - Alcohol Sensor | Grove - RTC | Grove - Differential Amplifier

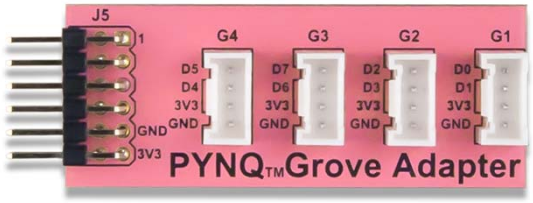
www.seeedstudio.com/wiki/Grove_System

Low-cost PYNQ Shield & Pmod Grove Adapter



PYNQ Shield:

- 4 x Analog ports
- 4 x I2C ports
- 3 x 3.3V GPIO ports
- 1 x UART
- 4 x 3.3/5V switchable GPIO ports
- 1 x SPI header
- 1 x 16-pin GPIO header (inner header)



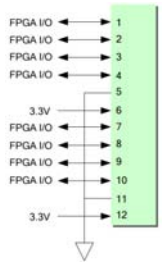
PYNQ Grove Adapter :

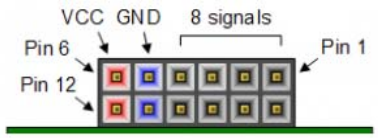
- 4 independent sockets for Grove modules
- Pmod compatible
- Solderless breadboard compatible
- Open-source design

XILINX.

Pmod: many physical & electrical instances

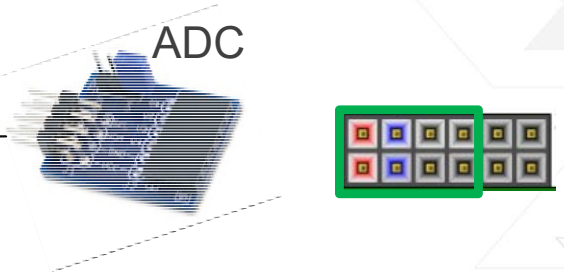
Pmod





Pin	Signal	Description
1 & 5	SCL	Serial Clock
2 & 6	SDA	Serial Data
3 & 7	GND	Power Supply Ground
4 & 8	VCC	Power Supply (3.3V/5V)

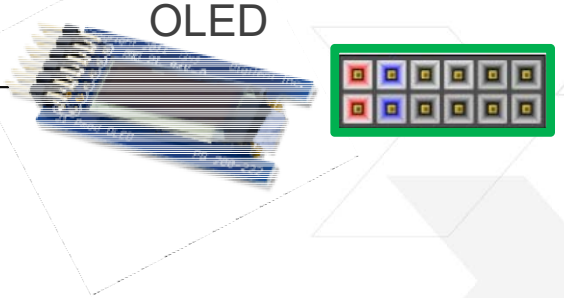
ADC



Connector J1

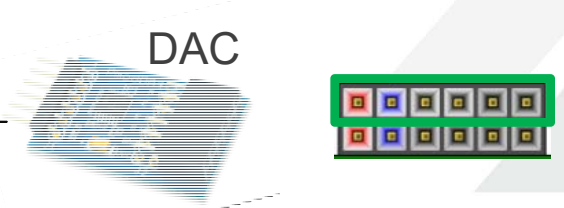
Pin	Signal	Description
1	CS	SPI Chip Select (Slave Select)
2	SDIN	SPI Data In (MOSI)
3	None	Unused Pin
4	SCLK	SPI Clock
7	D/C	Data/Command Control
8	RES	Power Reset
9	VBATC	V_{BAT} Battery Voltage Control
10	VDDC	V_{DD} Logic Voltage Control
5, 11	GND	Power Supply Ground
6, 12	VCC	Power Supply

OLED



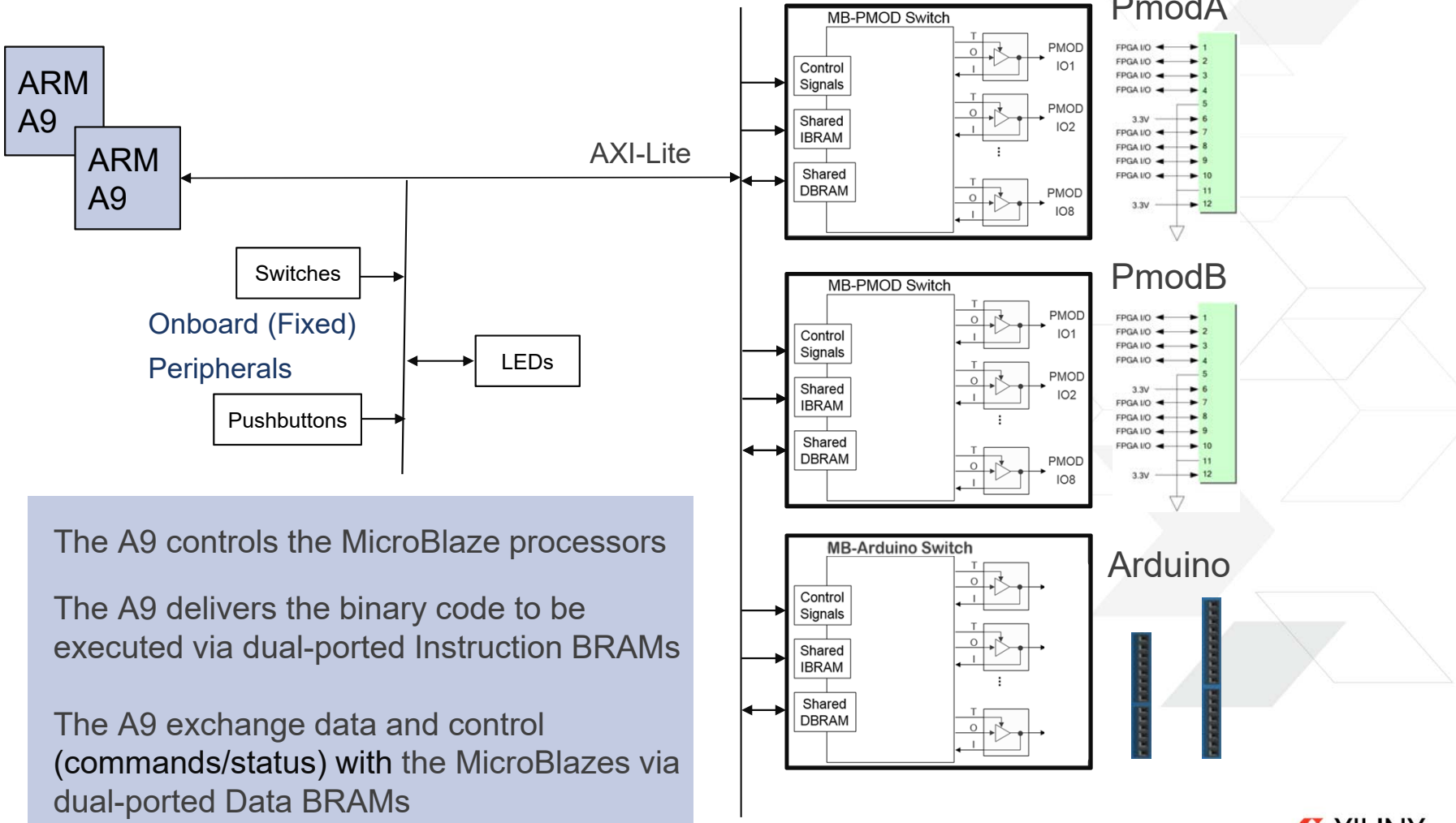
Pin	Signal	Description
1	~CS	Chip Select
2	MOSI	Master-Out-Slave-In
3	(NC)	Not Connected
4	SCLK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

DAC



What if we could handle all Pmod instances with a single bitstream?

base Overlay MicroBlaze IO Processor (IOP)

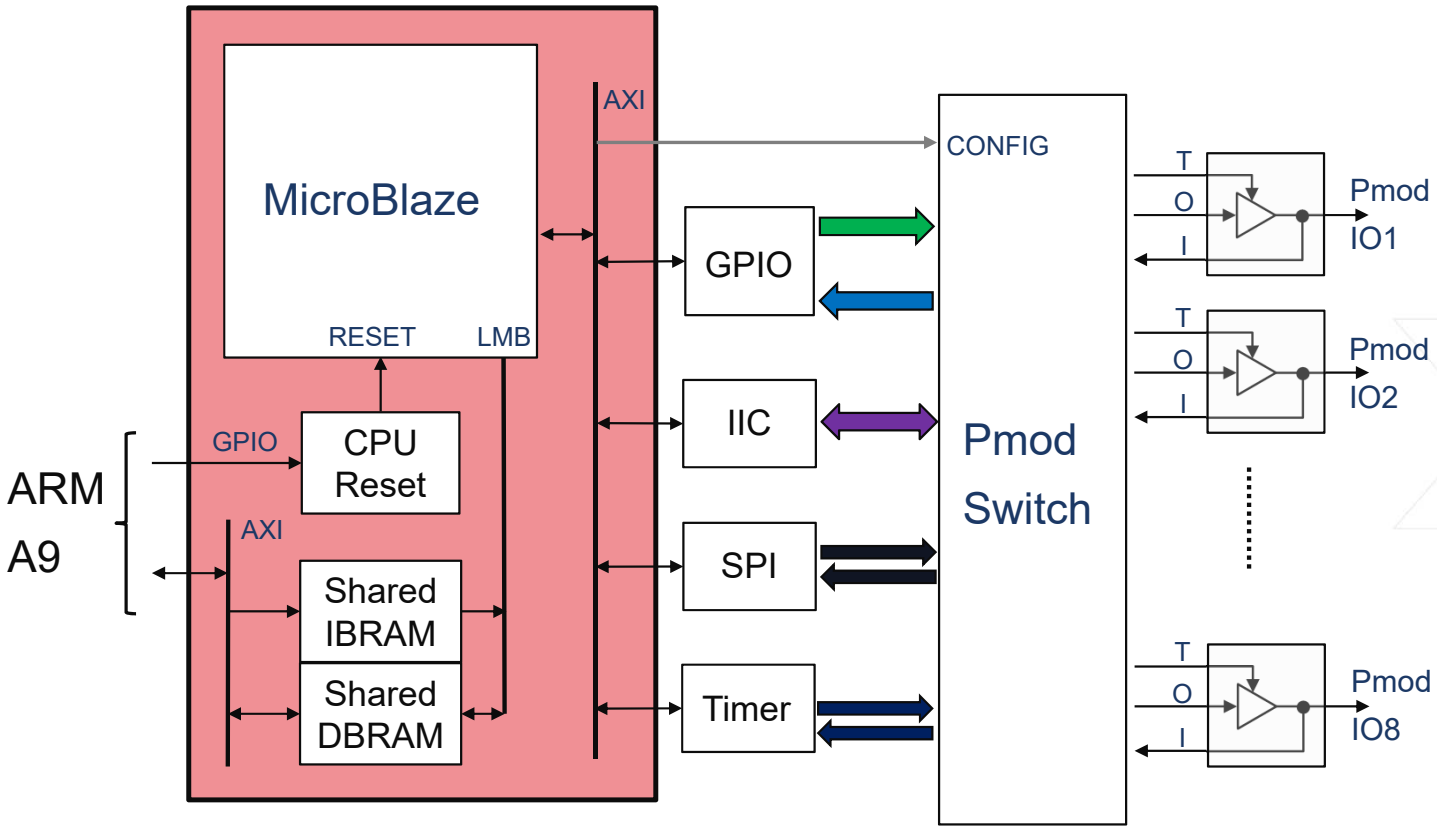


The A9 controls the MicroBlaze processors

The A9 delivers the binary code to be executed via dual-ported Instruction BRAMs

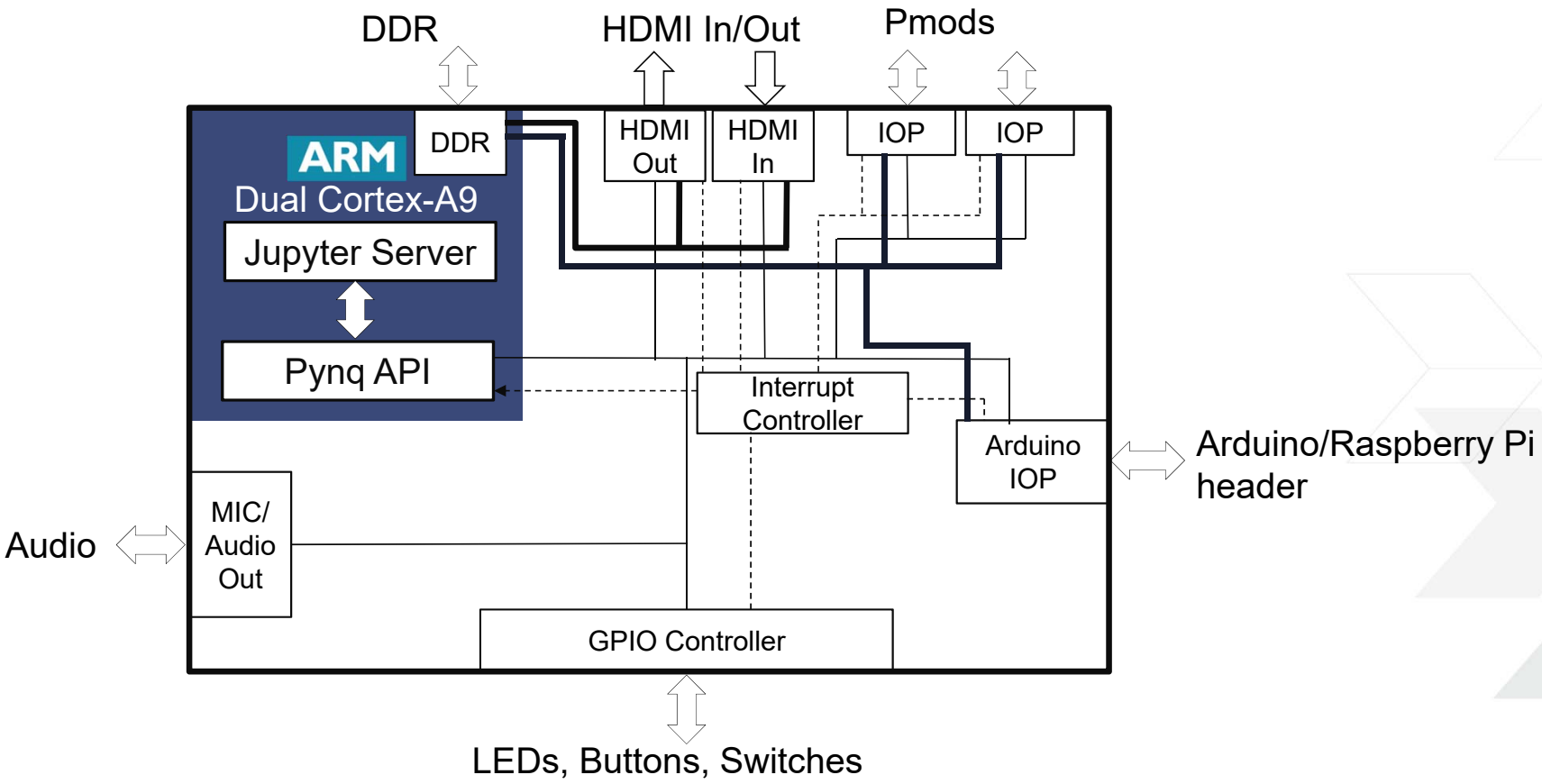
The A9 exchange data and control (commands/status) with the MicroBlazes via dual-ported Data BRAMs

Configure IO Processor for Pmod



XILINX.

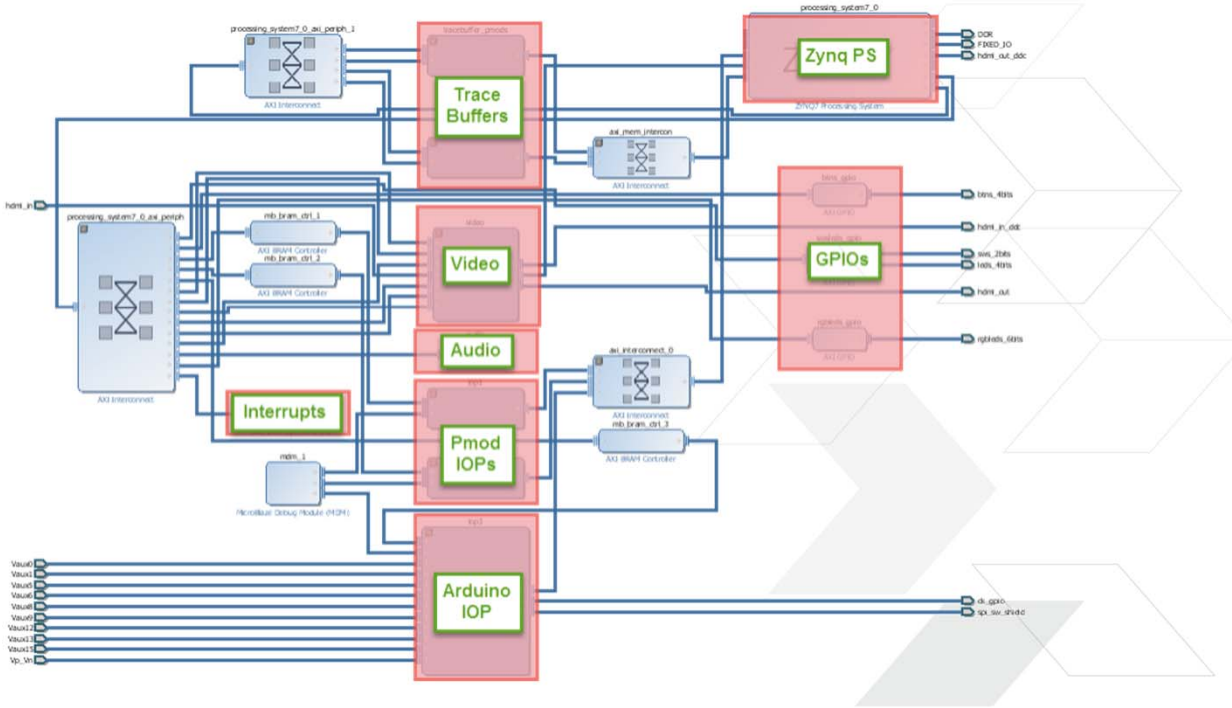
Complete *base* Overlay (base.bit)



XILINX.

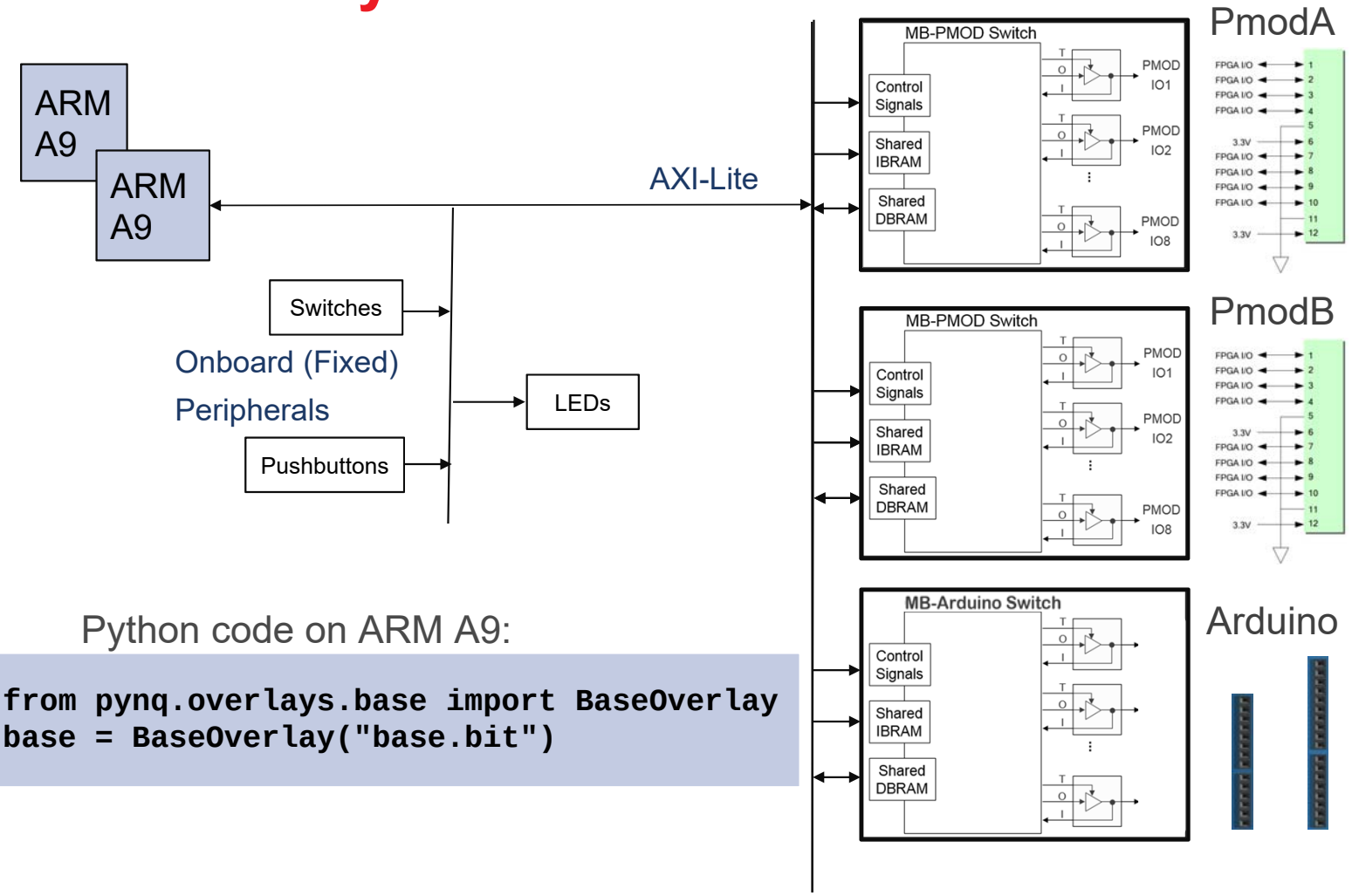
base Overlay – Vivado Interface View

- PL design of the base Overlay
- Standard FPGA design flow used
- Vivado IPI
- Interface to Python
- Memory Map
- Open source
- Components are re-useable



XILINX.

Load *base* overlay on PL



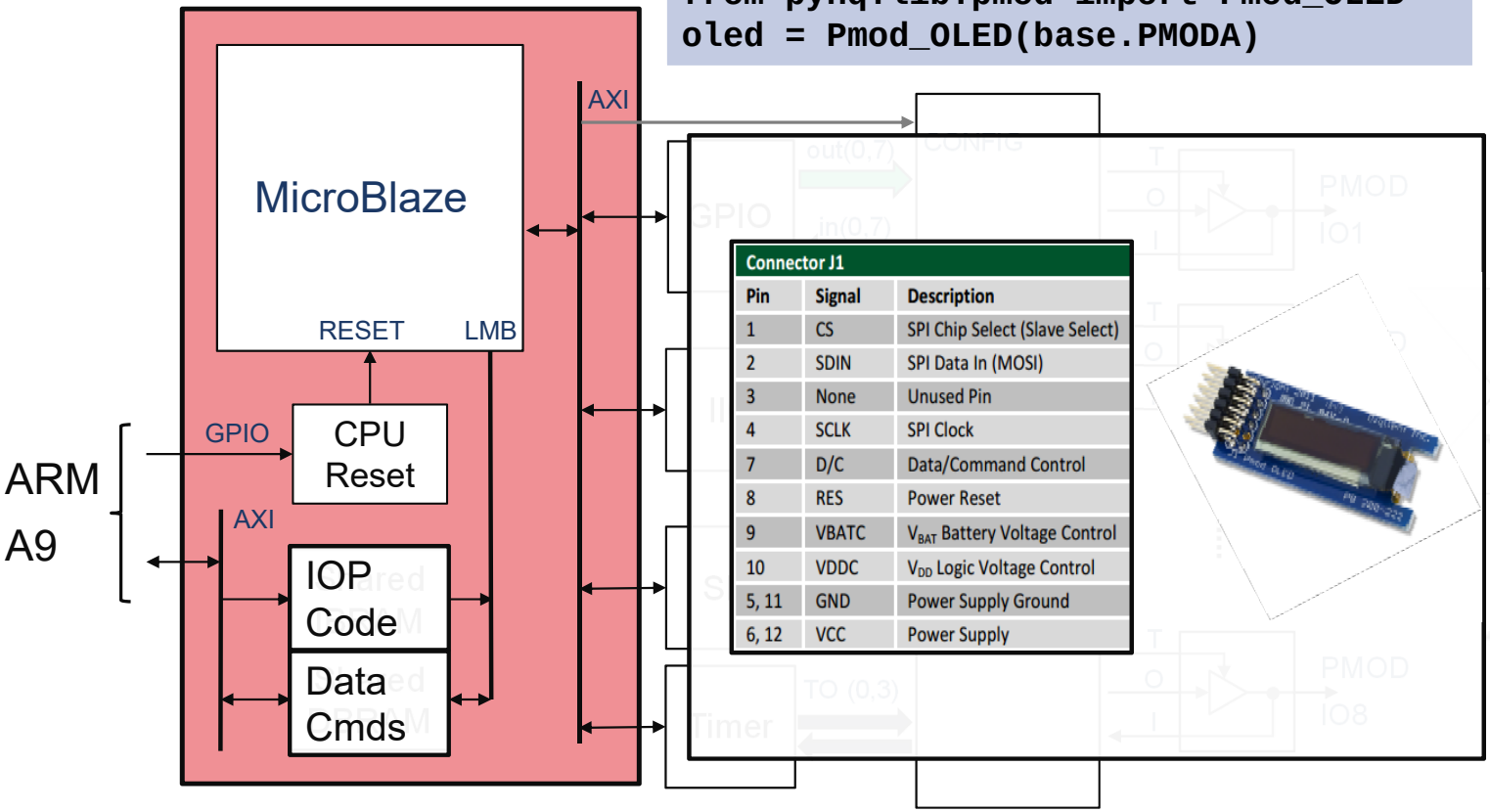
Python code on ARM A9:

```
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
```


XILINX.

Configure IO Processor

```
from pynq.lib.pmod import Pmod_OLED
oled = Pmod_OLED(base.PMODA)
```



Connector J1		
Pin	Signal	Description
1	CS	SPI Chip Select (Slave Select)
2	SDIN	SPI Data In (MOSI)
3	None	Unused Pin
4	SCLK	SPI Clock
7	D/C	Data/Command Control
8	RES	Power Reset
9	VBATC	V_{BAT} Battery Voltage Control
10	VDDC	V_{DD} Logic Voltage Control
5, 11	GND	Power Supply Ground
6, 12	VCC	Power Supply


```
oled.write("1 2 3 4 5 6")
```

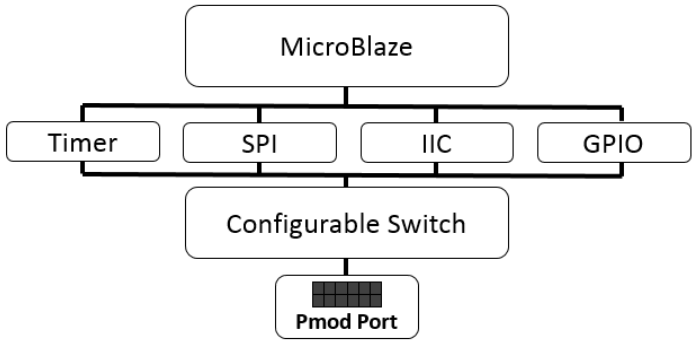

XILINX.

Lab exercises: Session 2

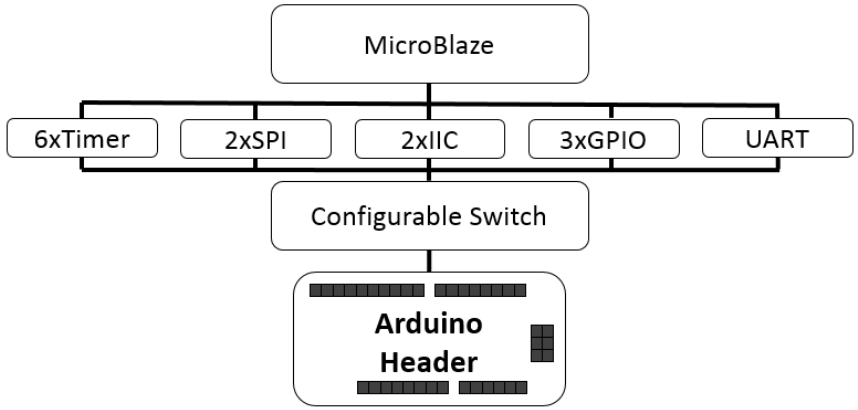
> **Pmod OLED**

IOP software flow

- **Pmod IOP/Arduino IOP/ RPi IOP**
  - Same MicroBlaze & instruction/data memory
  - Same configurable switch
  - Supports wide range of peripherals
- **The process for building software is the same**



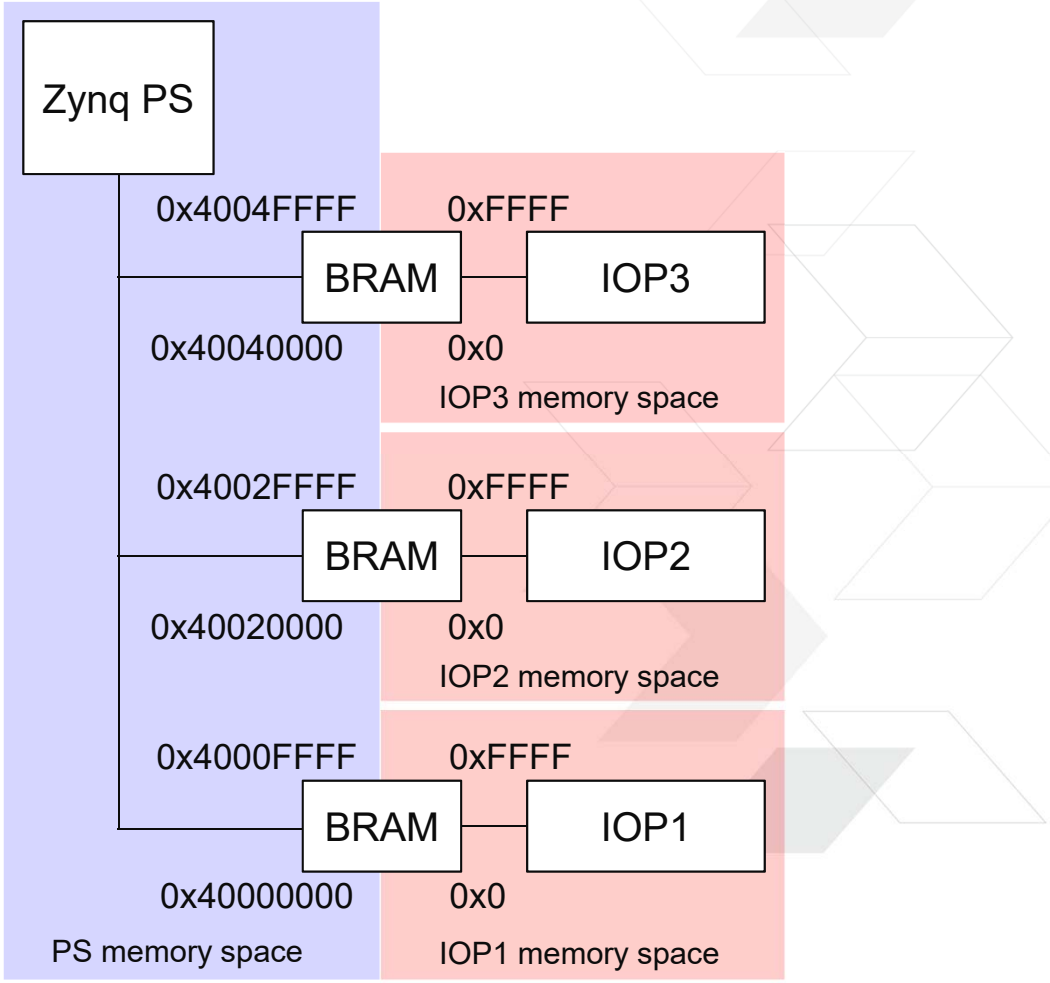
Pmod IOP



Arduino IOP

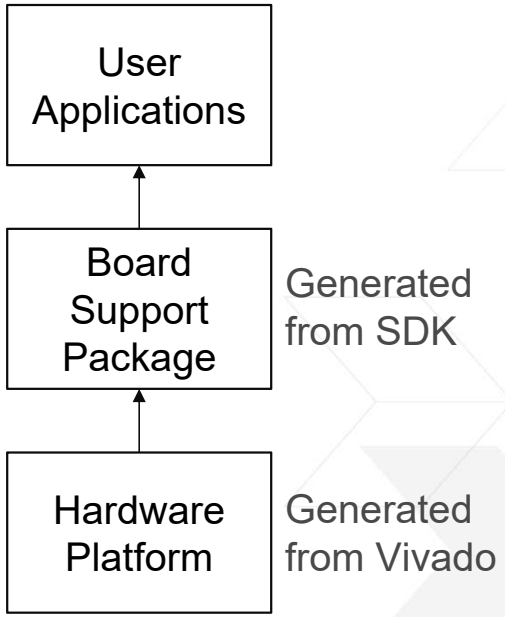
Building an IOP executable

- **IOP instruction/data memory accessible from IOP and PS**
- **From the PS perspective:**
  - Each IOP memory has different location in PS memory map
- **From the IOP perspective:**
  - Each IOP has a consistent memory map
  - Code for an IOP can be compiled for any IOP (of the same type)
    - E.g. Pmod IOP executable will run on other Pmod IOPs, not on an Arduino IOP
  - The same executable can be run on any IOP (of the same type)
- **PS/Python can load program, and share data with IOP**



Writing software

- **Standard MicroBlaze software design**
  - Xilinx SDK
  - gcc/make flow
- **“Hardware Platform” required**
  - Generated by Vivado
  - Available pre-compiled in Pynq repository
- **“Board Support Package” required**
  - Requires Hardware Platform
  - Generated by SDK



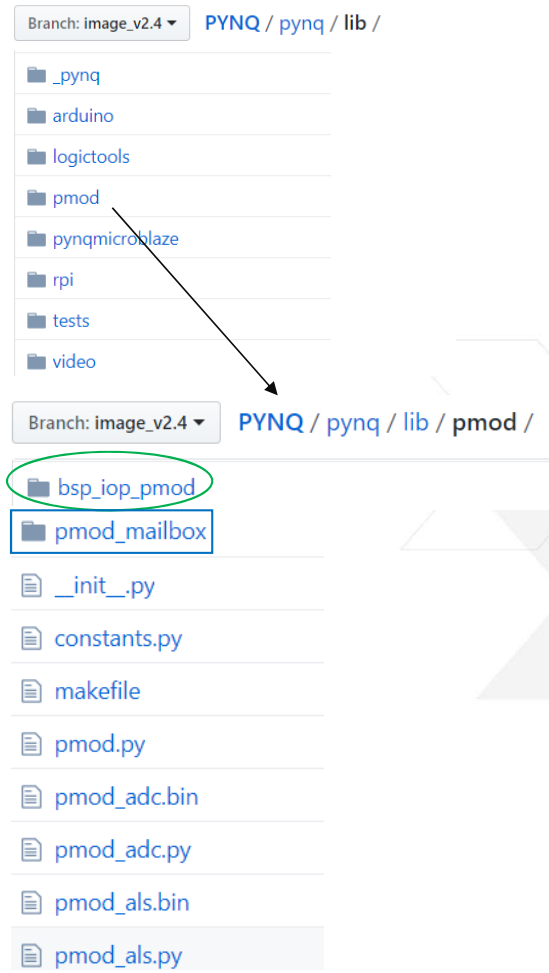
Example projects (GitHub)

- **Source code and projects available on GitHub for a range of peripherals**
  - Grove and Pmod
  - Some Arduino shield examples
  - Can be used as starting point for a new project
- **API available**
  - IIC, SPI, GPIO, Configurable switch
    - Simple low level API's; Read( ), Write( )
- ***Make* flow to build IOP projects available**

XILINX.

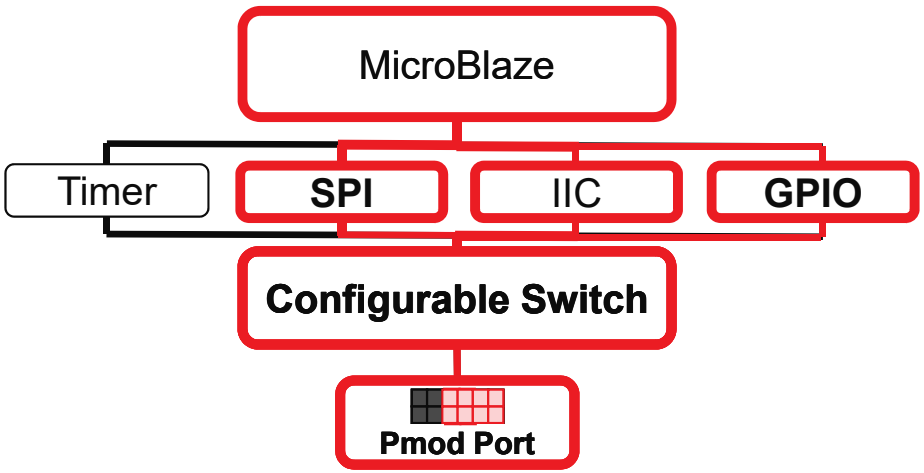
Software directory (GitHub)

- **Various software projects grouped according to interface and overlay related reside under ./pynq/lib/**
  - Arduino, logictools, Pmod, pynqmicroblaze, rpi, video
- **Under each group reside related software projects, bsp, makefile, bin (binary executable files), and Python class file**
- **mailbox**
  - Enables data and command/status exchanges between AP and IOP



Configurable Switch

> **Allows peripherals with different interfaces to be used in the same overlay without needing a new FPGA design**



Configurable Switch

- **Common API for all types of interfaces**
  - Pmod, Arduino, Raspberry Pi
  - xio_switch.h, xio_switch.c
    - config_io_switch(), set_pin(), init_io_switch()
- **Pin types can be**
  - GPIO, I2C, SPI, Timer, UART
- **Call open_device()**
  - Calls set_pin()
    - Pin 3 is configured as SCL
    - Pin 2 is configured as SDA

```
void config_io_switch(int num_of_pins);
void set_pin(int pin_number, u8 pin_type);
void init_io_switch(void);
```

```
device = i2c_open(3, 2);
```

```
i2c i2c_open(unsigned int sda, unsigned int scl){
    if (last_sda != -1) set_pin(last_sda, GPIO);
    if (last_scl != -1) set_pin(last_scl, GPIO);
    last_sda = sda;
    last_scl = scl;
    set_pin(scl, SCL0);
    set_pin(sda, SDA0);
    return i2c_open_device(XPAR_IO_SWITCH_0_I2C0_BASEADDR);
}
```

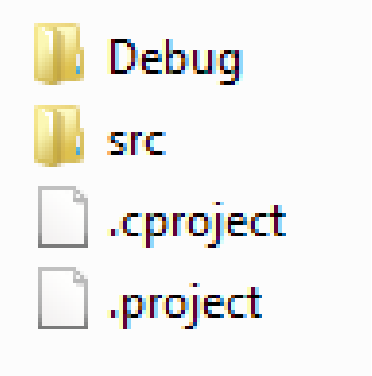
Branch: image_v2.4 ▾ PYNQ / boards / sw_repo / pynqmb / src /

- Makefile
- circular_buffer.c
- circular_buffer.h
- gpio.c
- gpio.h
- i2c.c
- i2c.h
- spi.c
- spi.h
- timer.c
- timer.h
- uart.c
- uart.h

```
enum io_configuration {
    GPIO      = 0x00,
    UART0_TX  = 0x02,
    UART0_RX  = 0x03,
    SPICLK0   = 0x04,
    MISO0     = 0x05,
    MOSI0     = 0x06,
    SS0       = 0x07,
    SPICLK1   = 0x08,
    MISO1     = 0x09,
    MOSI1     = 0x0A,
    SS1       = 0x0B,
    SDA0      = 0x0C,
    SCL0      = 0x0D,
    SDA1      = 0x0E,
    SCL1      = 0x0F,
    PWM0      = 0x10,
    PWM1      = 0x11,
    PWM2      = 0x12,
    PWM3      = 0x13,
    PWM4      = 0x14,
    PWM5      = 0x15,
    TIMER_G0  = 0x18,
    TIMER_G1  = 0x19,
    TIMER_G2  = 0x1A,
    TIMER_G3  = 0x1B,
    TIMER_G4  = 0x1C,
    TIMER_G5  = 0x1D,
    TIMER_G6  = 0x1E,
    TIMER_G7  = 0x1F,
    UART1_TX  = 0x22,
    UART1_RX  = 0x23,
    TIMER_IC0 = 0x38,
    TIMER_IC1 = 0x39,
    TIMER_IC2 = 0x3A,
    TIMER_IC3 = 0x3B,
    TIMER_IC4 = 0x3C,
    TIMER_IC5 = 0x3D,
    TIMER_IC6 = 0x3E,
    TIMER_IC7 = 0x3F,
};
```

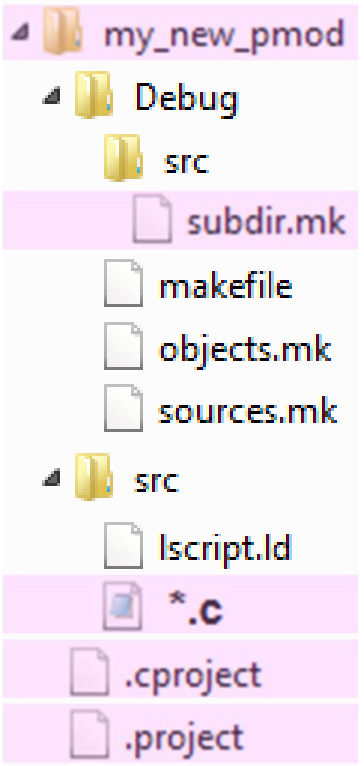
IOP Project

- **Xilinx SDK project files**
  - .cproject, .project
  - Not essential, but allow project to be imported back into SDK
- **src/**
  - Contains C source code, and linker script
- **Debug/**
  - makefile to build IOP project as seen previously
  - Other project files (includes objects, sources, directories, build settings)



Creating your own IOP program

- **Recommended to start with existing project**
- **Copy project folder and rename**
  - E.g. pmod_als -> my_new_pmod
- **Find and replace project name in the following files:**
  - E.g. pmod_als -> my_new_pmod
    - .project, .cproject
    - Debug/makefile
    - Debug/src/subdir.mk
      - Add any other new source files to this file
- **Modify/Replace existing .c/.h source file in src/**



MicroBlaze magic!

```
In [1]: from pynq.overlays.base import BaseOverlay
        base = BaseOverlay('base.bit')

In [2]: %%microblaze base.PMODA
        #include <i2c.h>
        #include <pmod_grove.h>

        int adc_read() {
            i2c device = i2c_open(PMOD_G4_B, PMOD_G4_A);
            unsigned char buf[2];
            buf[0] = 0;
            i2c_write(device, 0x50, buf, 1);
            i2c_read(device, 0x50, buf, 2);
            return ((buf[0] & 0xF) << 8) | buf[1];
        }

In [3]: adc_read()

Out[3]: 2178
```

IPython “magics”

Compile Microblaze on ARM

Bind C to Python?

Overlay Design Methodology

XILINX

PYNQ

Outline

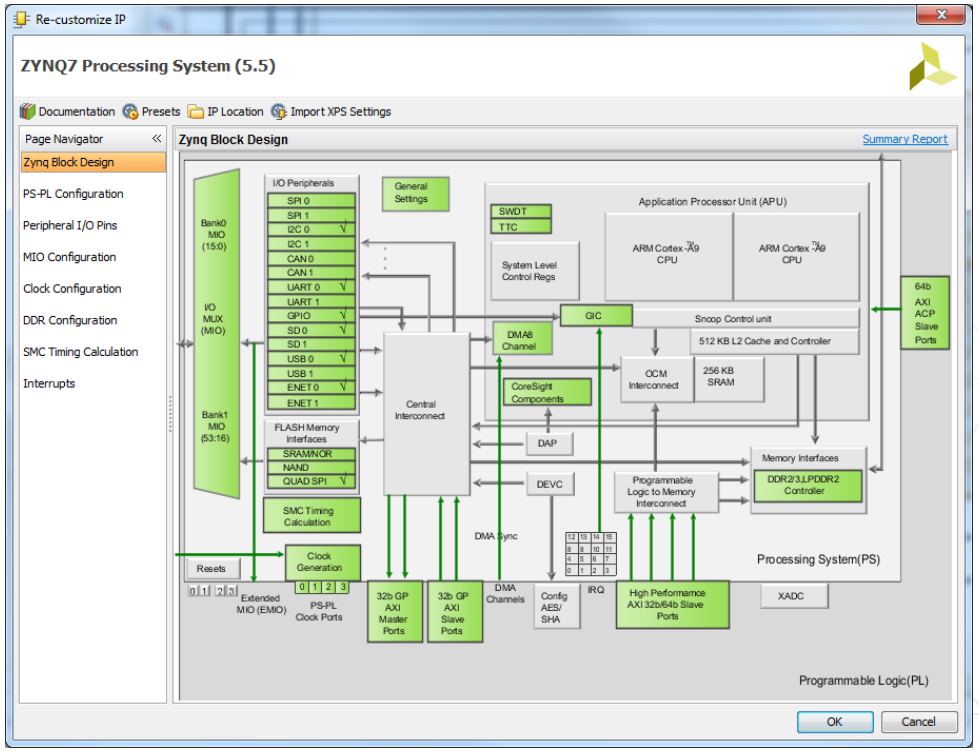
- *Overlay design considerations*
- Zynq PS-PL interfaces
- Pynq interface classes
- Overlay API

Overlay design considerations

- **Zynq design**
  - Zynq PS configuration settings
  - Zynq architecture & interfaces between PS-PL
  - PL design – reusable IP
  - Overlay programmability
- **Python API**
  - C integration
- **Soft processors**
  - Software design for any IOPs or other soft processor

Zynq base configuration

- **PYNQ image used to boot board**
  - PS is configured during boot
  - Default overlay loaded
  - Some PS settings can be changed at runtime
    - E.g. PS to PL clocks
- **PS settings**
  - SD, Ethernet, USB, UART, DDR3 (Used by PYNQ)
  - Used in some overlays: Flash, GPIO (Interrupts), IIC (Video)
  - Interrupts enabled

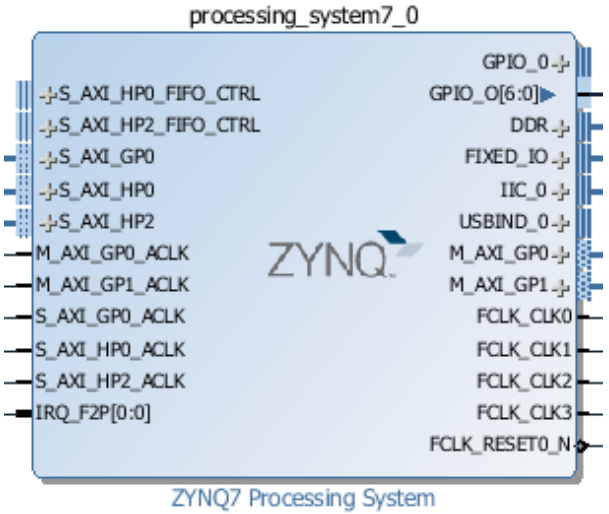


Zynq PS settings

XILINX.

Designing and using overlays

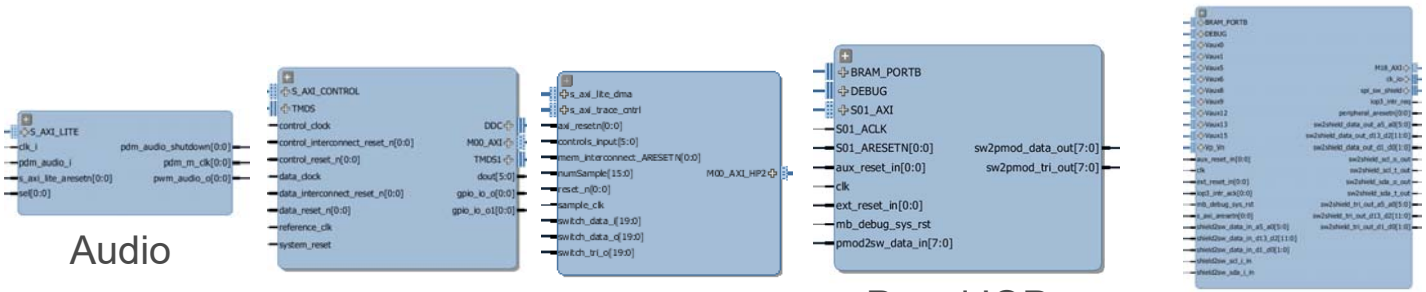
- **Zynq design**
  - Retain existing “PYNQ” MIO peripherals
    - SD, Ethernet, USB, UART, DDR3
    - May be possible to extend with EMIO
  - Use any of available PS-PL interfaces
    - Do not need to be enabled in default (boot) overlay
  - Modify PS to PL clocks as required
- **Downloading new overlays**
  - The PL design (bitstream) will be downloaded by default when overlay is instantiated
  - PS clock settings to be updated from overlay Tcl




```
Overlay("base.bit“)
```

PL Design

- **Standard FPGA/Zynq design**
- **V2.4: PYNQ supports Vivado: 2018.3**
  - IP versions and drivers depend on Vivado version
    - PYNQ drivers also depend on version
  - Other versions may work, but are untested (IP usually have minor revisions between versions)
- **PYNQ reusable blocks**
  - Audio, Video, Tracebuffer, IOPs



XILINX.

Overlay Programmability

- **End-user programmability and flexibility valued over logic utilisation**
  - Create Programmable IP and overlays that can be reused for many applications and domains
- **IOPs support many peripherals from the same overlay design**
- **Performance vs flexibility**
  - For example; Neural Network Design
    - Weights can be “hard coded” into bitstream for higher performance/resource utilisation
    - However, each application may have a specific network that would need to be rebuilt
  - Neural Network Overlay
    - Weights can be programmed into BRAM at runtime for flexibility
    - Many networks can run on the same overlay

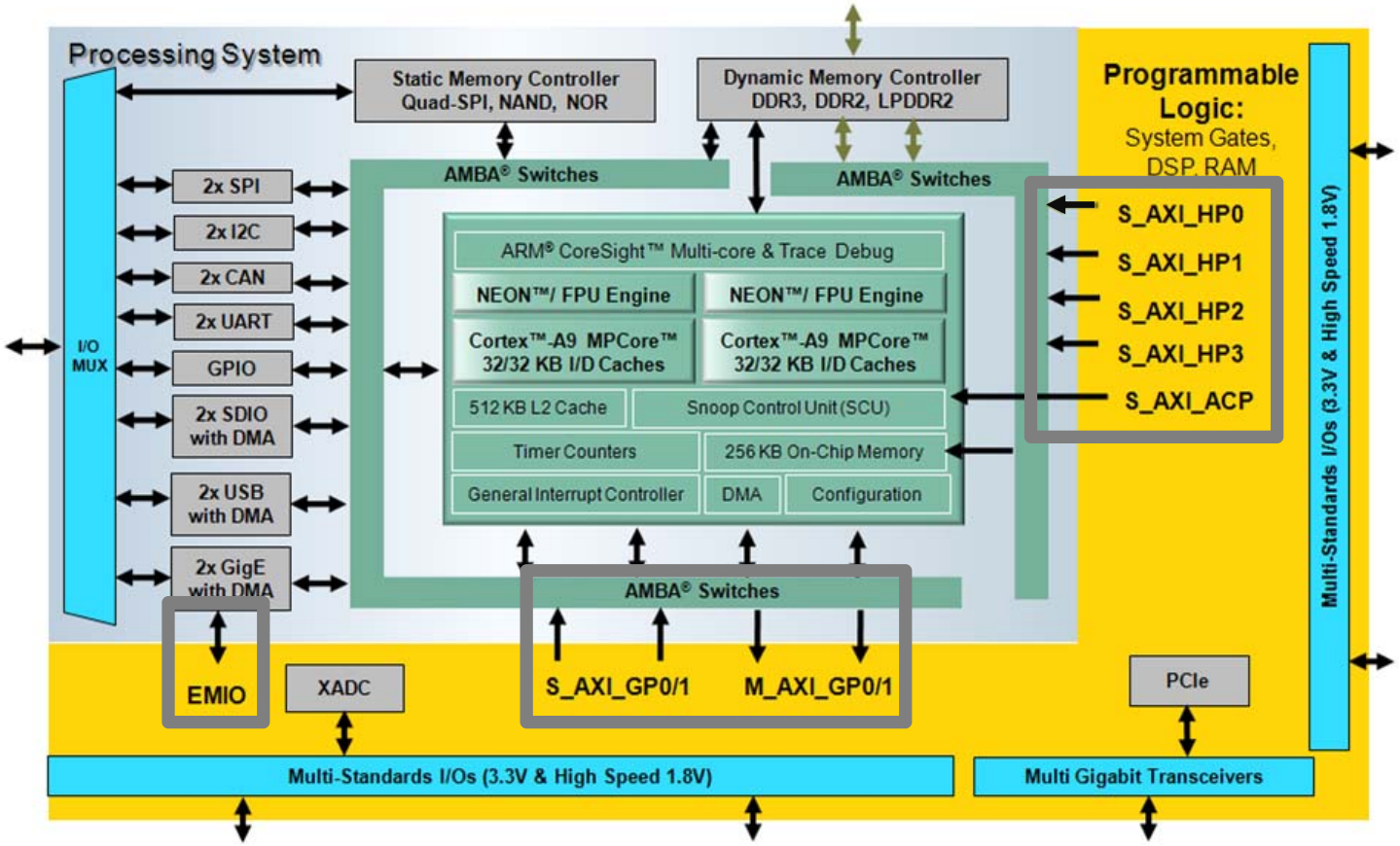
Overlay Tcl/hwh

- **Bitstream + Tcl/hwh must be provided to load an overlay**
  - Default location: \\pynq\xilinx\pynq\overlays\<overlay>\
- **Must generate Tcl from Vivado IP Integrator Block Diagram**
  - "Standardizes" TCL scripts
  - File > Export > Export Block Design
  - write_bd_tcl command
- **OR use automatically generated hwh file from <proj_dir>/<proj>.srcs/sources_1/bd/<design_name>/hw_handoff**
- **Tcl/hwh and bitstream file names should match**
- **Tcl/hwh files provide information about the system**
  - List of IP in the design
  - GPIO information, used to connect IOP resets/interrupts
  - PS to PL clock setting, used to dynamically update clocks

Outline

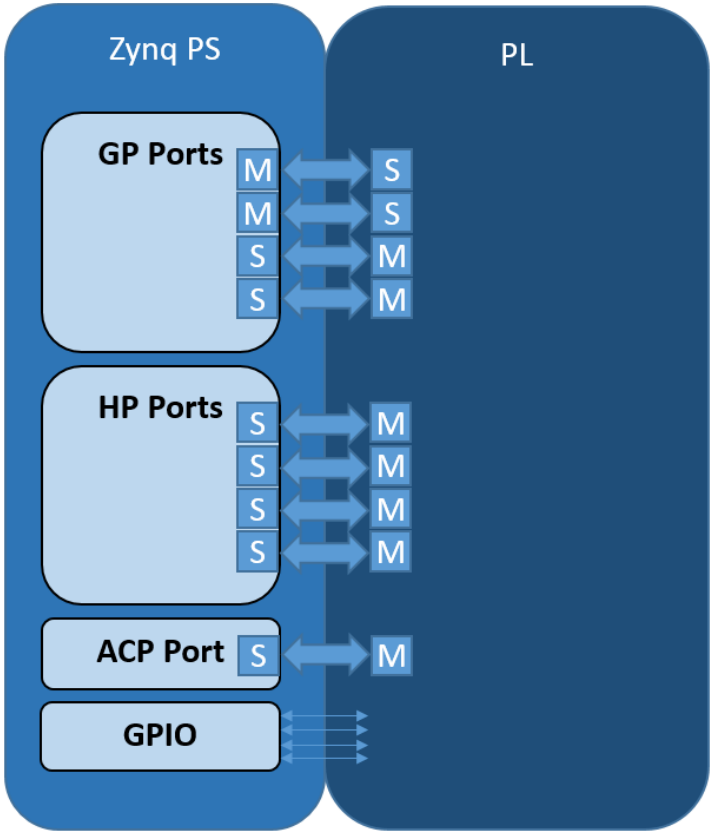
- Overlay design considerations
- *Zynq PS-PL interfaces*
- Pynq interface classes
- Overlay API

Zynq PS-PL interfaces



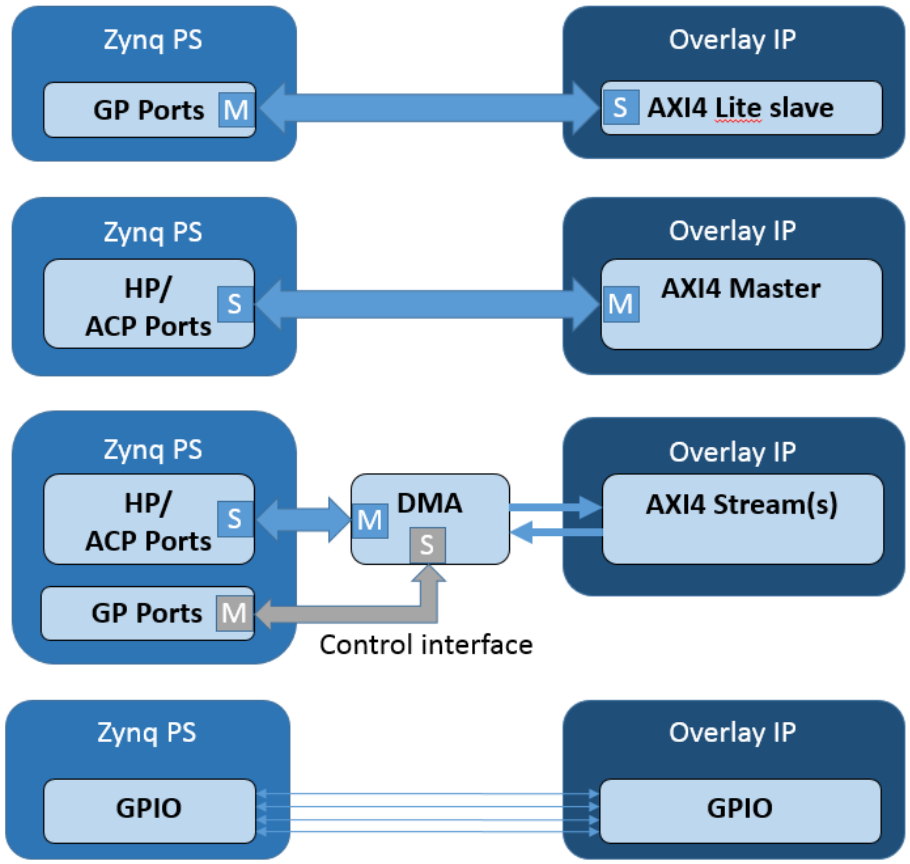
XILINX.

PS-PL interfaces types



- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)
- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access
- **GPIO (EMIO)**
  - Wires (64)

IP interfaces in PL



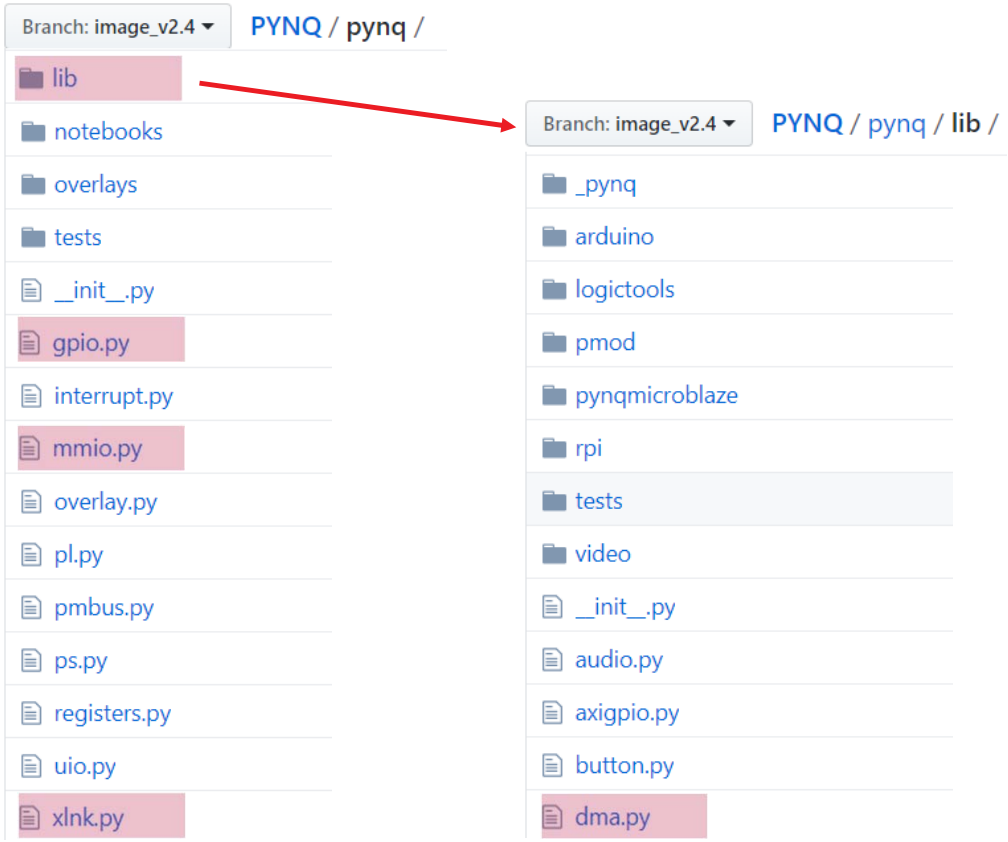
- **AXI (Lite) Slave IP**
  - General Purpose ports
  - Typically lower performance IP
- **AXI Master**
  - AXI HP/ACP
  - Typically higher performance IP
- **AXI Stream**
  - Via DMA
  - HP/ACP ports for data path
  - GP slave for control
- **GPIO**
  - Control type data

Outline

- Overlay design considerations
- Zynq PS-PL interfaces
- *Pynq interface classes*
- Overlay API

Pynq interface classes – run on PS

- **MMIO (pynq.mmio)**
  - Memory Mapped Input Output
  - Register based memory mapped transactions
- **Xlnk (pynq.xlnk)**
  - Memory allocation (used in DMA or for AXI Master peripheral)
- **DMA (pynq.lib.dma)**
  - Direct Memory Access
  - Offload memory transfers from main CPU
- **GPIO (pynq.gpio)**
  - Read/Write GPIO wires



MMIO Class – memory mapped IO

- **Import MMIO**
- **Define memory mapped region**
  - BASE_ADDRESS: starting location
  - ARRAY_SIZE: length of accessible memory (optional, default 4 bytes)
- **Read and Write 32-bit values**
  - ADDRESS_OFFSET: Offset from BASE_ADDRESS, should be multiples of 4
  - Need to ensure the memory can be read/written

```
from pynq import MMIO

BASE_ADDRESS = 0x40000000
ARRAY_SIZE = 1024

mmio = MMIO(BASE_ADDRESS, ARRAY_SIZE)

data = 0x12345678
ADDRESS_OFFSET = 0x10

mmio.write(ADDRESS_OFFSET, data)

result = mmio.read(ADDRESS_OFFSET)

print(hex(result))
> 0x12345678
```

XInk Class

- **Memory managed by Linux**
  - Virtual
- **Memory must be allocated before PL Master can access it**
  - PL needs the physical address of a memory buffer
- **XInk can allocate (contiguous) memory buffers (using NumPy)**
  - Maps virtual and physical addresses
  - Contiguous memory is more efficient/allows simpler DMA logic
  - Array can be specified as NumPy data type, and size/shape
- **Once buffer is allocated a DMA can be used to transfer data between PS/PL**
- **DMA class uses XInk for memory allocation**

Xlnk example

- **Import Xlnk**
- **Allocate contiguous buffer**
  - cma_array()
  - Returns Linux virtual address
- **Get Physical Address**
  - `.physical_address`
  - Can be used by PL to access DDR (Linux) memory
- **Read/write buffer**

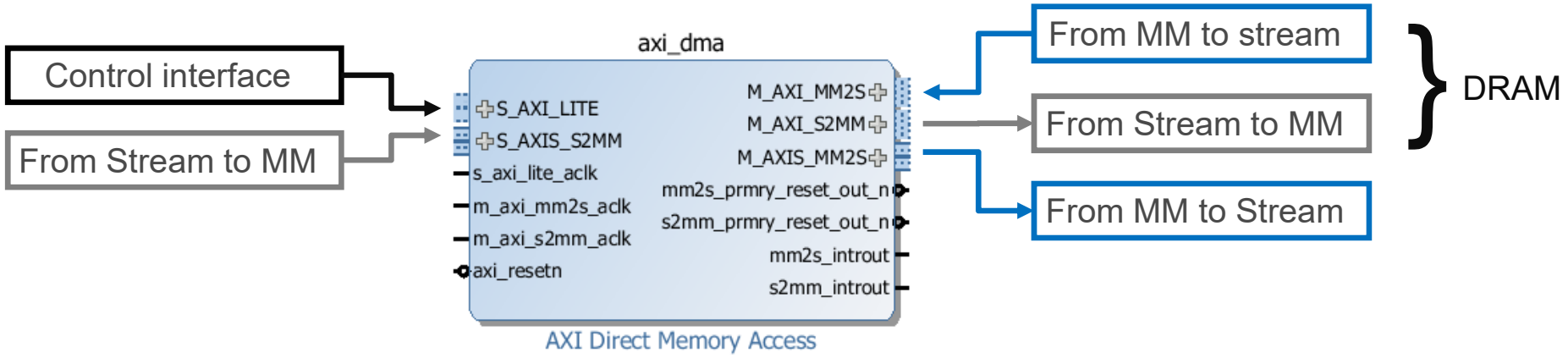
```
MEMORY_SIZE = 10
from pynq import Xlnk
import numpy as np
xlnk = Xlnk()

input_buffer = xlnk.cma_array(shape=(10,),
dtype=np.float32)
phy_addr = input_buffer.physical_address

for i in range(10):
    input_buffer[i] = i
print(input_buffer)
```

DMA Class

- **Direct memory access**
  - Transfer data between memories directly
    - PS - PL
  - Bypasses CPU
    - Doesn't waste CPU cycles on data transfer
  - Speed up memory transfers with burst transactions
- **Xilinx *AXI Direct Memory Access* IP block supported in PYNQ**
  - Read and Write Paths from PL to DDR and DDR to PL
  - Memory Mapped to Stream
  - Stream to Memory Mapped
- **Needs to stream to/from an allocated memory buffer**
  - PYNQ DMA class inherits from xlnk for memory allocation

AXI Direct Memory Access IP

- **AXI Lite control interface (AXI GP port)**
- **Memory mapped interface (AXI interface, HP/ACP ports)**
- **AXI Stream interface (AXI stream accelerator)**
- **Transfer between streams and memory mapped locations**
  - Paths from PL to DRAM and DRAM to PL
  - Memory Mapped to Stream (MM2S)/Stream to Memory Mapped (S2MM)



XILINX.

DMA example

- Setup DMAs
- Allocate memory buffers
- Start DMA transactions

Create DMA instances

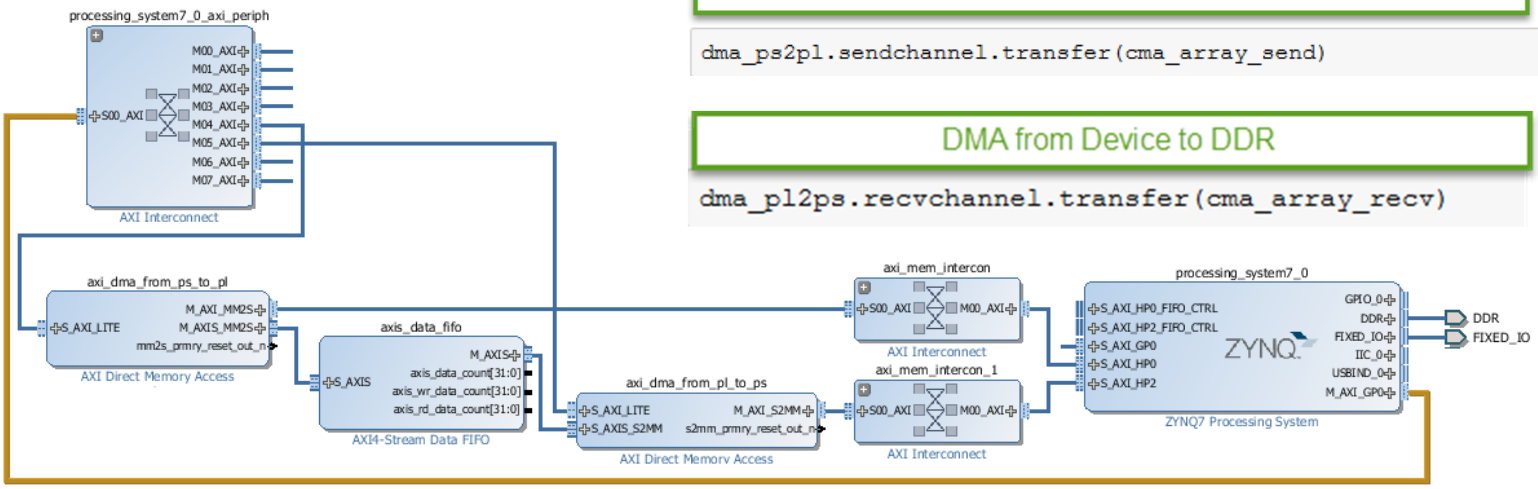
```
from pynq.lib import DMA

dma_ps2pl_description = overlay.ip_dict['axi_dma_from_ps_to_pl']
dma_ps2pl = DMA(dma_ps2pl_description)

dma_pl2ps_description = overlay.ip_dict['axi_dma_from_pl_to_ps']
dma_pl2ps = DMA(dma_pl2ps_description)
```

DMA from DDR to Device

```
dma_ps2pl.sendchannel.transfer(cma_array_send)
```

DMA from Device to DDR

```
dma_pl2ps.recvchannel.transfer(cma_array_recv)
```



XILINX.

GPIO Class

- **Up to 64 GPIO wires from PS**
  - Tri-state
- **Accessed from Linux**
- **Setup GPIO instance**
- **Map to GPIO pin**
  - Translated pin numbers to Linux GPIO number
  - get_gpio_pin()
- **Read/Write**
- **Most appropriate for simple control**
  - Set, reset, start, done …

```
from pynq import GPIO

btn_gpio = GPIO.get_gpio_pin(0)
led_gpio = GPIO.get_gpio_pin(1)

ps_btn = GPIO(btn_gpio, 'in')
ps_led = GPIO(led_gpio, 'out')

ps_btn.read()

ps_led.write()
```

Python Wrapper

- **Standard Python code**
- **Interface classes used**
  - MMIO, Xlnk, DMA, GPIO
- **Check PYNQ repository for examples**

```
class LED(object):
    """This class controls the onboard leds.

    Attributes
    ----------
    _impl : object
        An object with appropriate LED methods
    """
    def __init__(self, device):
        """Create a new LED object.
        Parameters
        ----------
        device : object
            An object with appropriate LED methods:
            on, off, toggle and len
        """
        methods = ['on', 'off', 'toggle']
        if all(m in dir(device) for m in methods):
            self._impl = device
        else:
            raise TypeError("'device' must contain LED methods: " +
                            str(methods))
    def toggle(self):
        """Toggle led on/off."""
        self._impl.toggle()
    def on(self):
        """Turn on led."""
        self._impl.on()
    def off(self):
        """Turn off led."""
        self._impl.off()
```

Outline

- Overlay design considerations
- Zynq PS-PL interfaces
- Pynq interface classes
- *Overlay API*

Python Overlay API

- **PYNQ Overlay class supports basic overlay functionality**
  - Requires overlay Tcl/hwh
  - Allows download, and overlay discovery (ip_dict)
  - Assigns default drivers to IP
    - Read() and write() access to IP address space

```
from pynq import Overlay
overlay = Overlay("pynqtutorial.bit")
```

```
help(overlay)
```

```
class Overlay(pynq.pl.Bitstream)
 |  Default documentation for overlay pynqtutorial.bit.
 |  attributes are available on this overlay:
 |
 |  IP Blocks
 |  ----------
 |  axi_dma_from_pl_to_ps : pynq.lib.dma.DMA
 |  axi_dma_from_ps_to_pl : pynq.lib.dma.DMA
 |  btns_gpio             : pynq.lib.axigpio.AxiGPIO
 |  mb_bram_ctrl_1        : pynq.overlay.DefaultIP
 |  mb_bram_ctrl_2        : pynq.overlay.DefaultIP
 |  rgbleds_gpio          : pynq.lib.axigpio.AxiGPIO
 |  swsleds_gpio          : pynq.lib.axigpio.AxiGPIO
 |  system_interrupts     : pynq.overlay.DefaultIP
```

```
overlay.rgbleds_gpio.write(0, 3)
overlay.swsleds_gpio.read()
```

Custom Overlay API

- **Custom Overlay API**
  - Allows automatic assignment of custom drivers
    - Overwrite default drivers
  - No need to import additional drivers
  - Easier for software developers to use overlays
- **Overlay "attributes" available from overlay instance**
  - E.g. base.PMODA
- **help()**
  - Discover information about the overlay

```
from pynq.overlays.base import BaseOverlay

base = BaseOverlay("base.bit")
```

```
help(base)
```

```
Help on BaseOverlay in module pynq.overlays.base.base ob

class BaseOverlay(pynq.overlay.Overlay)
 |  The Base overlay for the Pynq-Z1
 |
 |  This overlay is designed to interact with all of the
 |  and external interfaces of the Pynq-Z1 board. It exp
 |  attributes:
 |
 |  Attributes
 |  ----------
 |  iop1 : IOP
 |       IO processor connected to the PMODA interface
 |  iop2 : IOP
 |       IO processor connected to the PMODB interface
 |  iop3 : IOP
 |       IO processor connected to the Arduino/ChipKit i
 |  trace_pmoda : pynq.logictools.TraceAnalyzer
 |      Trace analyzer block on PMODA interface, control
```

Python overlay API example: base overlay

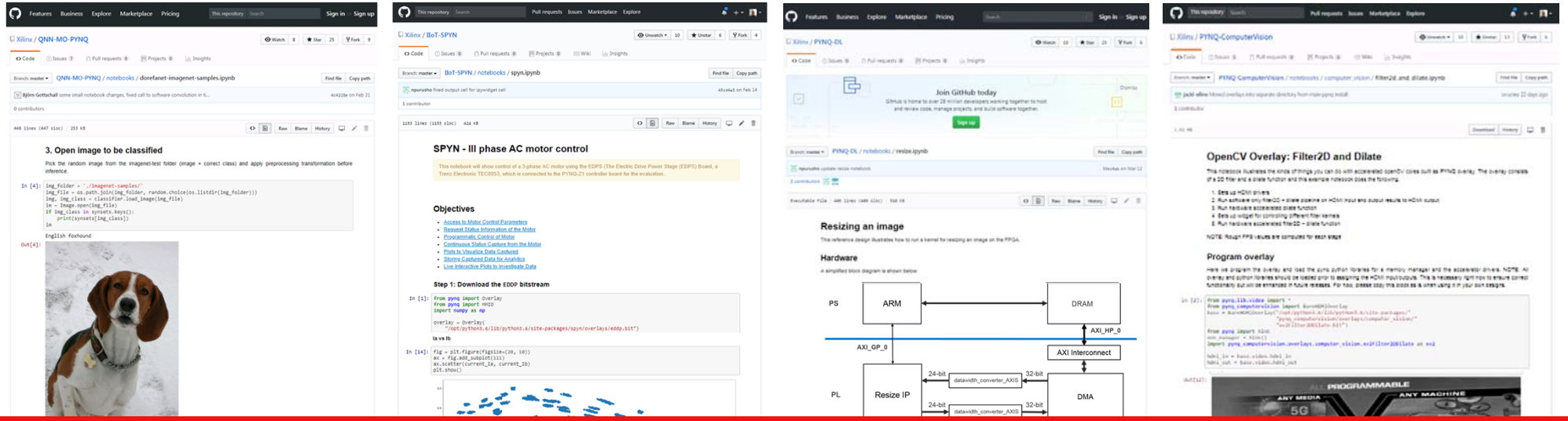
- **Custom Python wrapper provided with overlay**
- **Inherits from pynq.Overlay class**
- **Define custom drivers, interface types, IOP types, etc.**
- **If no driver is specified, a *DefaultIP* driver is assigned**
  - Read(), write() to IP address space (base on MMIO)

```
class BaseOverlay(pynq.Overlay):

    self.iop_pmoda.mbtype = "Pmod"
    self.iop_arduino.mbtype = "Arduino”
    self.iop_rpi.mbtype = “Rpi”


    self.leds =
         self.leds_gpio.channel1
    self.switches =
         self.switches_gpio.channel1

    self.leds.setlength(4)
    self.switches.setlength(2)

    self.leds.setdirection("out")
    self.switches.setdirection("in")
```

xilinx\pynq\boards\Pynq-Z2\base\base.py

Software-style packaging & distribution of designs

Enabled by new *hybrid packages*



Download a design from GitHub with a single Python command:

```
pip install git+https://github.com/Xilinx/pynq-helloworld.git
```


XILINX.

Overlays and Demo

- Digital Overlay
- Boolean Overlay
- PYNQ Infrastructure for HLS Debug and Analysis
- Partial Reconfiguration Overlay

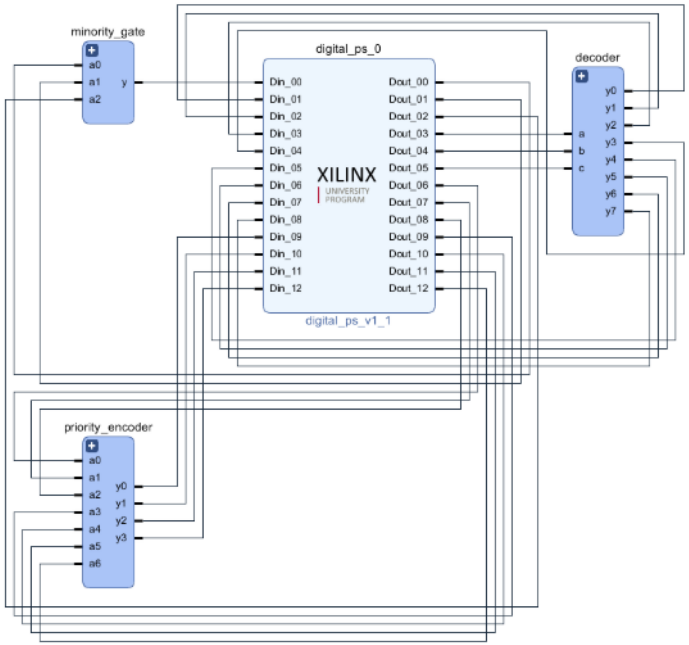
Digital Overlay

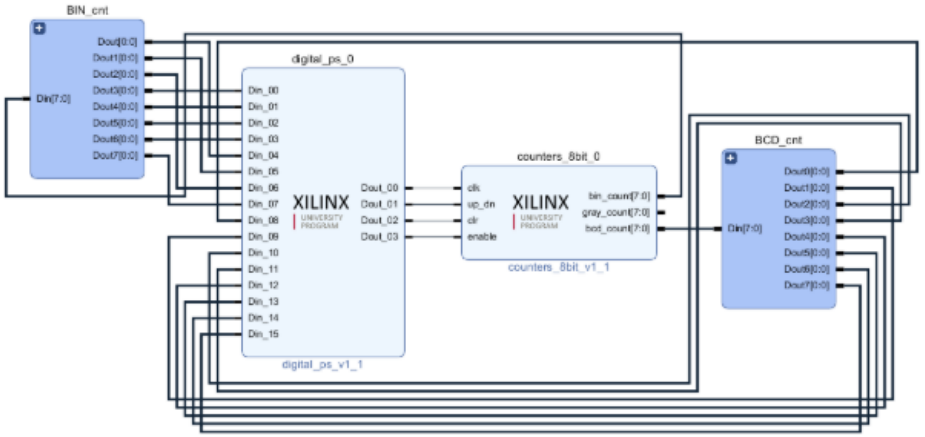
Notebook example showing how to use simple APIs to validate the circuitry built using Vivado IPI blocks

The available methods are:

- bit_write(bit#, value)
- bit_read(bit#)
- clk(bit#)
- bus_write(end_index, start_index, value)
- bus_read(end_index, start_index)

This example shows a *minority gate*, a *3-to-8 decoder*, and a *priority_encoder*

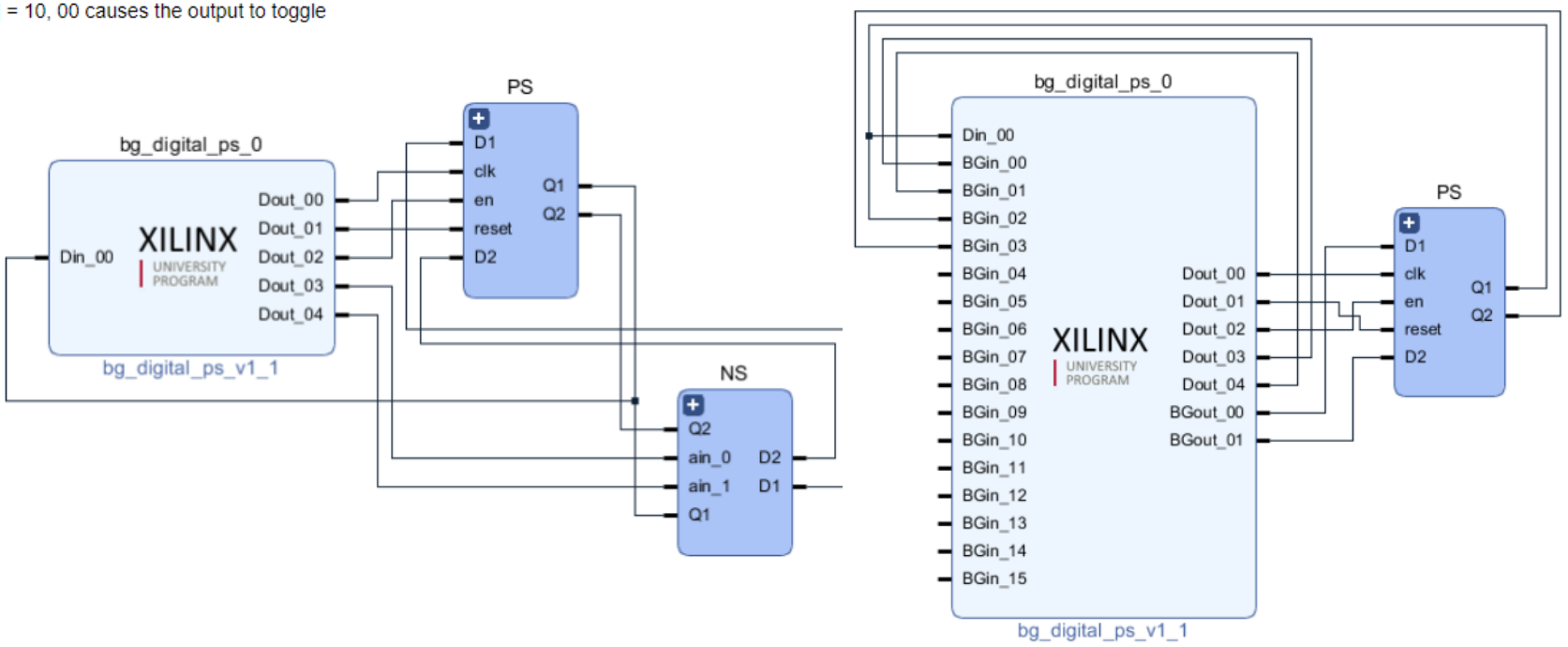




XILINX.

Boolean Overlay

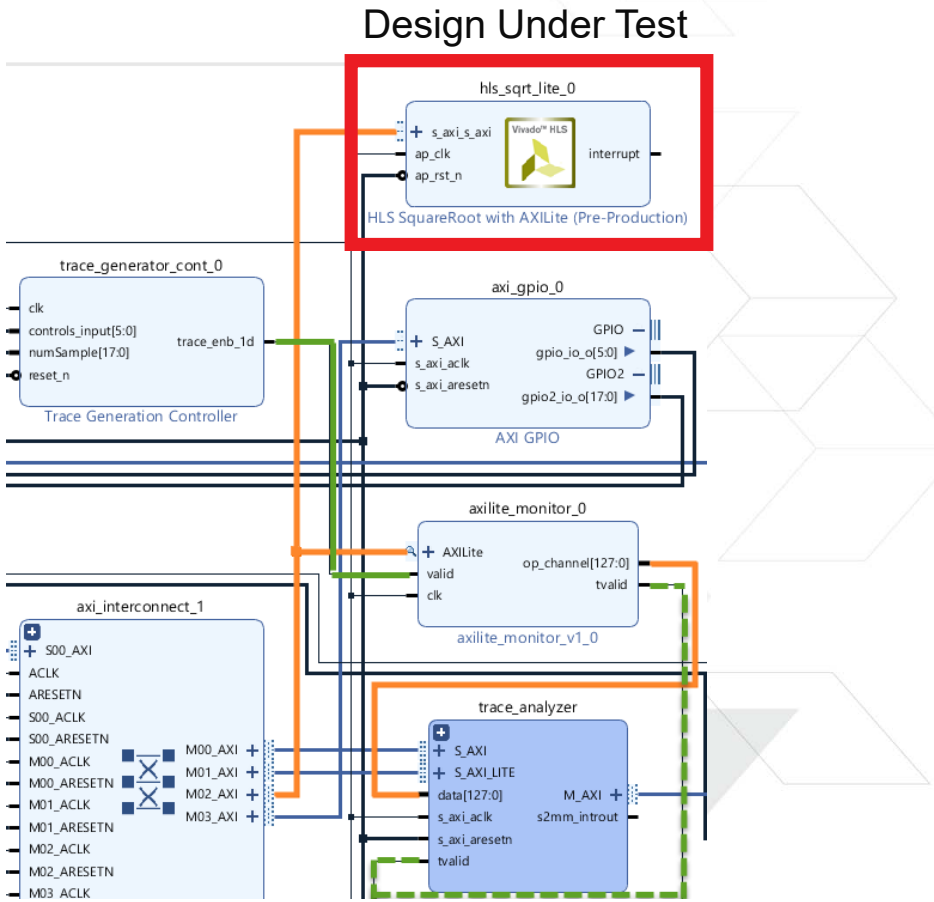
Design a sequence detector implementing a Moore state machine. The state machine has two inputs (ain[1:0]) and one output (yout). The output yout begins as 0 and remains a constant value unless one of the following input sequences occurs:
(i) The input sequence ain[1:0] = 01, 00 causes the output to become 0
(ii) The input sequence ain[1:0] = 11, 00 causes the output to become 1
(iii) The input sequence ain[1:0] = 10, 00 causes the output to toggle



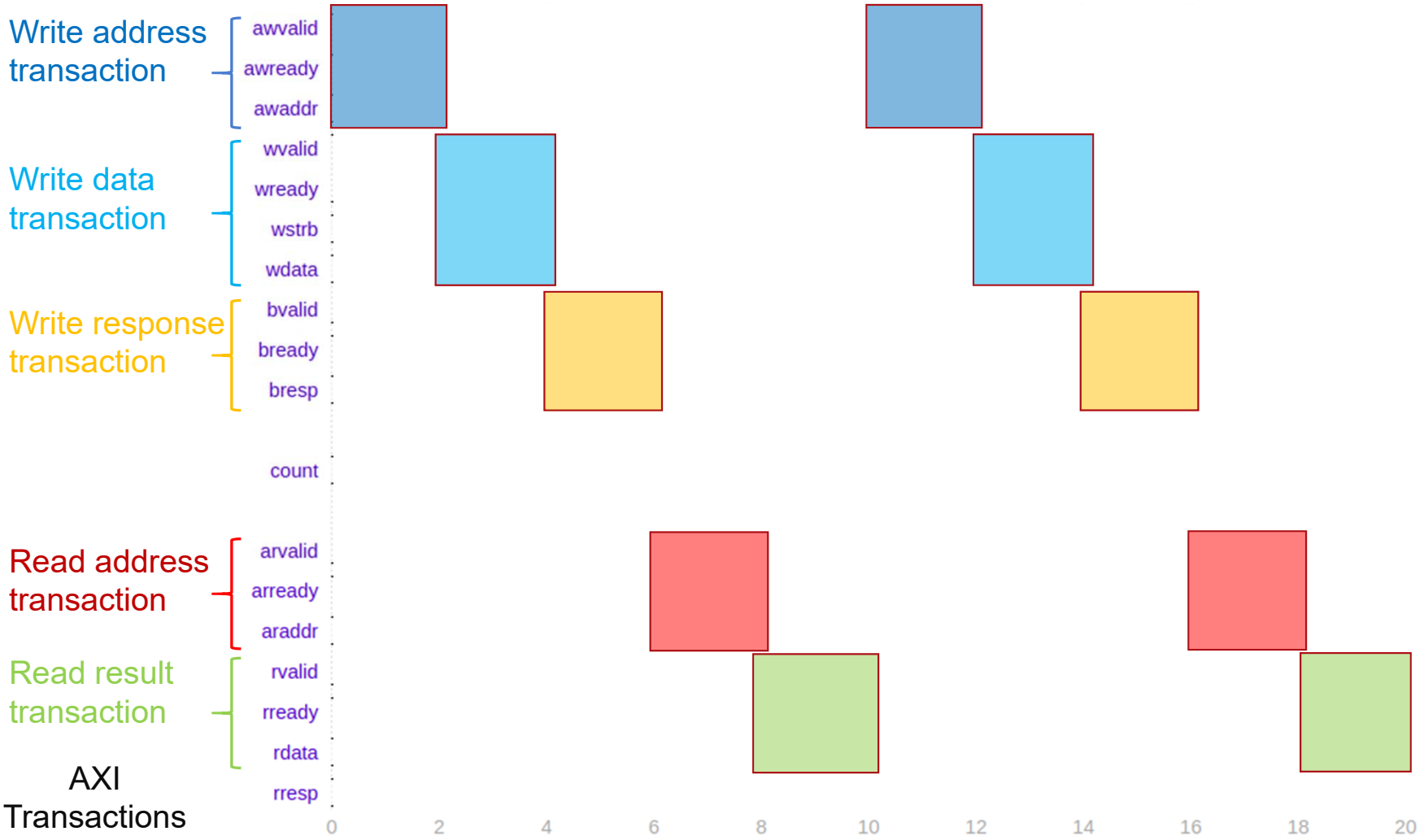
XILINX.

PYNQ Infrastructure for HLS Debug & Analysis

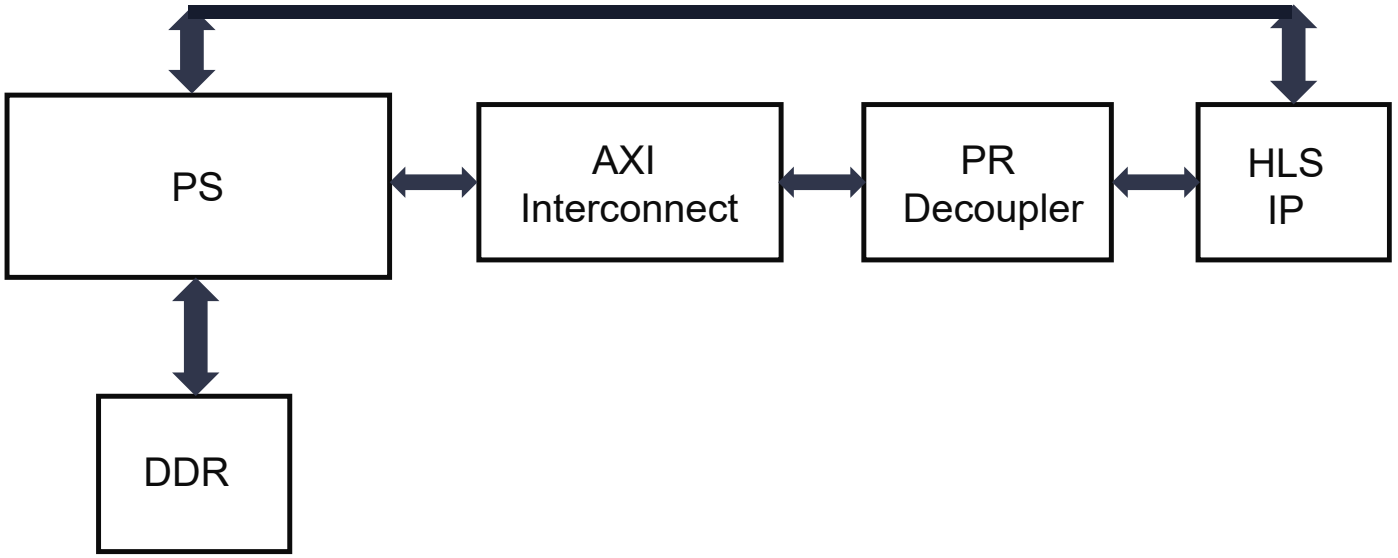
- **Making it easier to test and debug your hardware design from PYNQ**
- **Monitor signals at run time**
  - AXI interfaces
  - I/O pins
  - Internal signals
- **Capture data in Python**
- **Parse data interactively with WaveDrom**



Capture Trace data interactively



Partial Reconfiguration Overlay



PR AXI4

In [ ]:
```
from pynq.overlays.logictools.logictools_axi4_pr_trace import LogicToolsOverlay
from pynq import MMIO
from pynq.lib.logictools.compute import axi4_read_int,axi4_write_int
from pynq.lib.logictools.wavedrom_print import axi4_read_print,axi4_write_print
import numpy as np
from pynq import Xlnk
from pynq import GPIO
import math
```

In [ ]:
```
ol1=LogicToolsOverlay('sqrt.bit')
```

In [ ]:
```
xlnk = Xlnk()
length = 25

in_buffer = xlnk.cma_array(shape=(length,), dtype=np.int32) # input buffer
out_buffer = xlnk.cma_array(shape=(length,), dtype=np.int32) # output buffer
a=[i for i in range(length)]
for i in range(length):
    a[i]=int(math.pow(a[i],2))
np.copyto(in_buffer, a) # copy samples to inout buffer
```

In [ ]:
```
NUM_SAMPLES=length

trace_analyzer1=ol1.trace_ip_read
trace_analyzer1.setup(frequency_mhz=100, num_analyzer_samples=NUM_SAMPLES,fclk_index=0)

trace_analyzer2=ol1.trace_ip_write
trace_analyzer2.setup(frequency_mhz=100,num_analyzer_samples=NUM_SAMPLES,fclk_index=0)
```

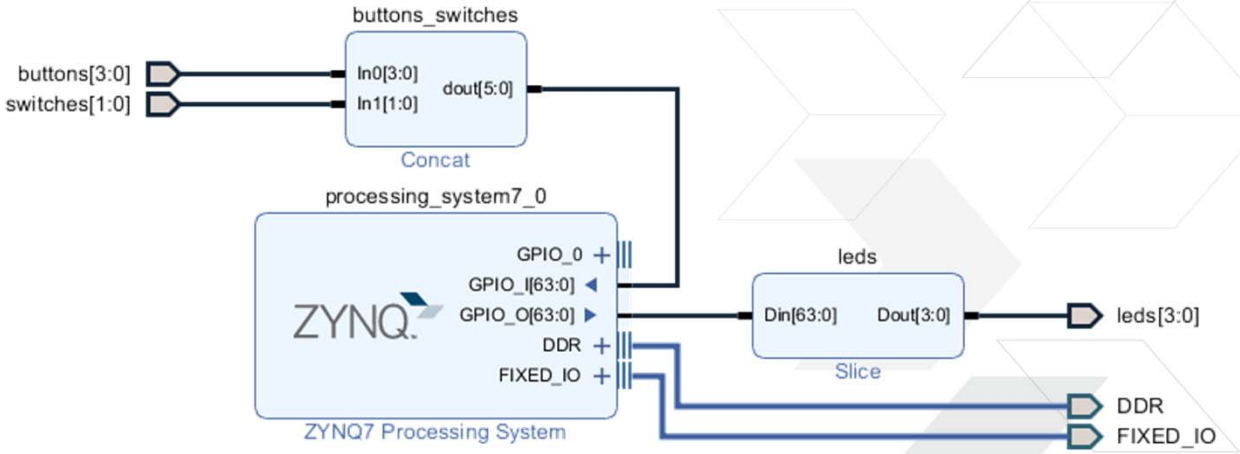
Lab exercises: Session 3

- PS GPIO
- AXI GPIO
- MMIO (GPIO)
- Memory allocation example using Xlnk
- Using memory from PL master
- DMA
- Image resizer

PS GPIO

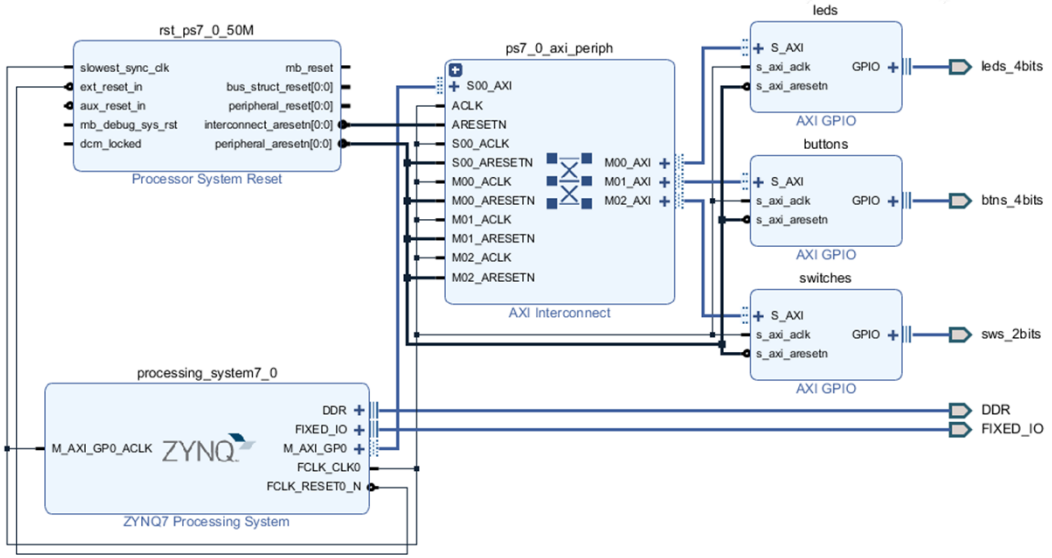
- **64 PS GPIO available**
  - Useful for debug/prototyping
- **Use PS GPIO class**
  - Instantiate GPIO pin using get_gpio_pin(n) method and direction
  - Use .write and .read methods to interact with the instantiated pin

- **GPIO 0:3 = push-buttons**
- **GPIO 4:5 = dip-switches**
- **GPIO 6:9 = LEDs**



AXI GPIO

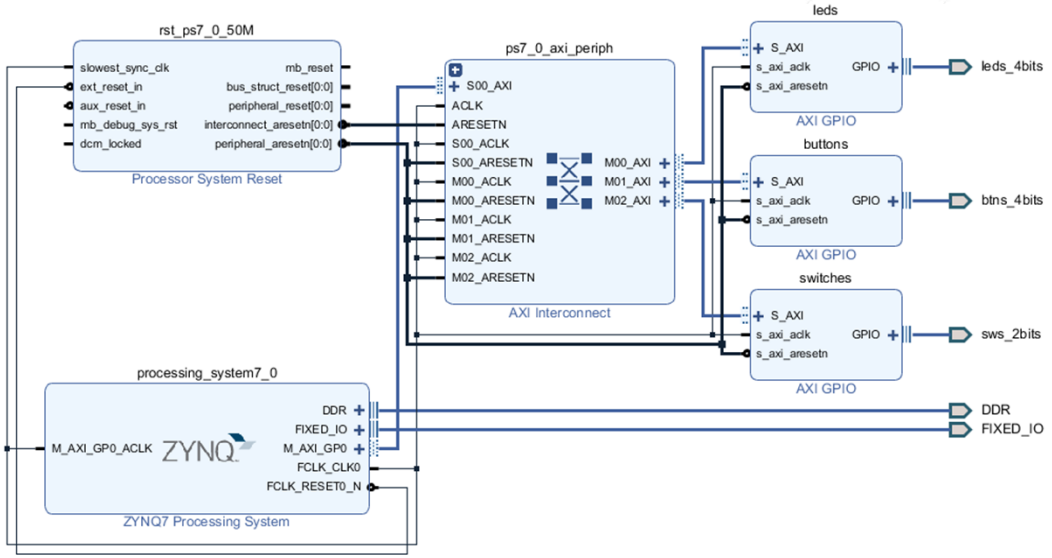
- **Similar to previous example**
  - Switches, buttons, LEDs
- **3x AXI GPIO controller**
- **Uses PYNQ AxiGPIO driver**
  - Address peripheral by slice
  - LED[0:3]



XILINX.

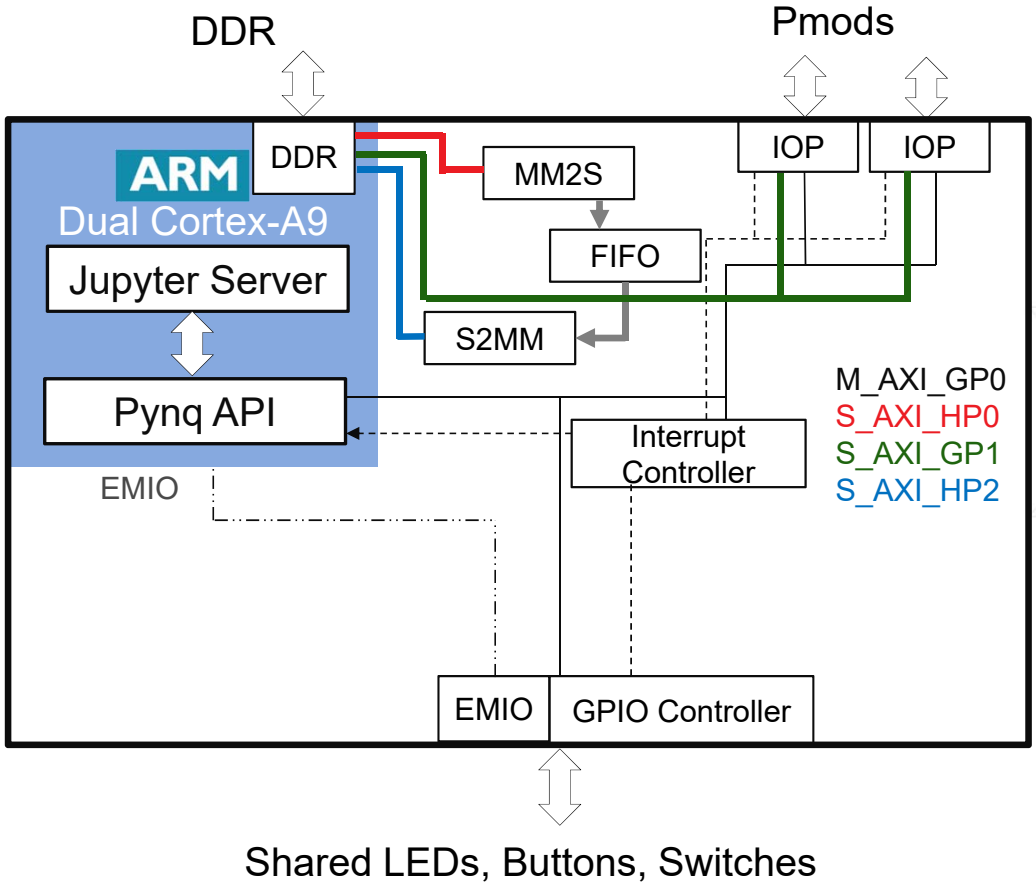
PYNQ MMIO

- Uses previous design
- AXI GPIO controller
- Read and write Register 0x0 directly
- Write tri-state register (0x4) to configure as inputs or outputs
- Build your own Python functions to control IP



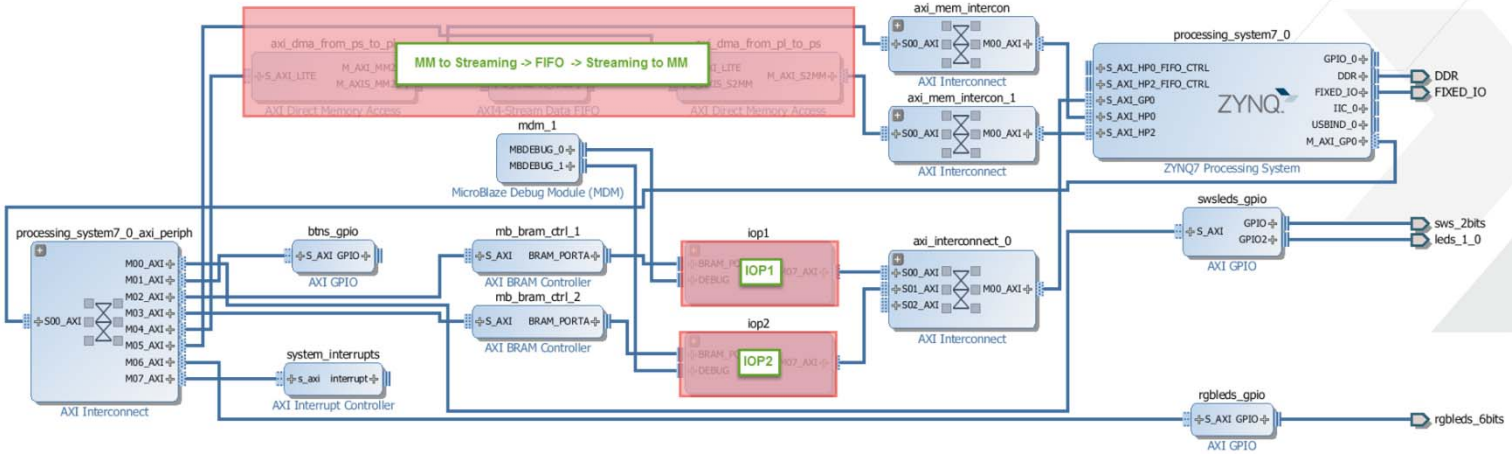
XILINX.

pynqtutorial Overlay (pynqtutorial.bit)



pynqtutorial Overlay – Interface View

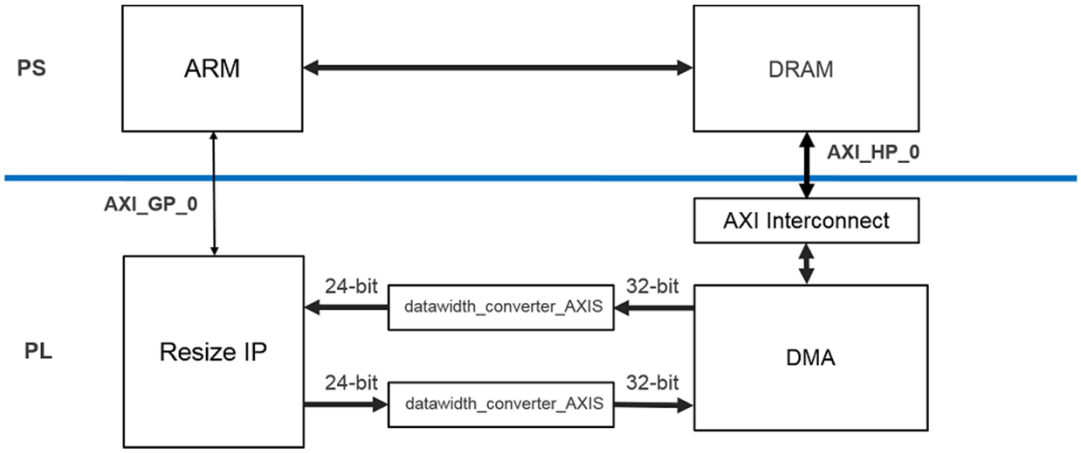
- **IP included**
  - IOP1 and IOP2
  - DMA -> FIFO -> DMA
- **Shared LEDs and buttons between EMIO and MIO ports of PS**
- **Allows testing of DMA interfaces to AXI streams**
- **Replace FIFO with custom stream IP**



XILINX.

Image resize

- Resizer IP in PL
- Uses DMA, and MMIO
- Resize an image in software
- Resize image in PL and compare



Summary

- **PYNQ is an open-source framework**
  - Outweighs programmability over performance
  - Distributed processing/co-processing using multiple soft processors
  - Supports various kinds of external devices
    - High-speed: video
    - Low-speed: audio, pmod, grove, Arduino, Raspberry-Pi
- **Overlay Design considerations**
  - PL, PS configuration, Programmability, Python interface
- **Zynq PS-PL interfaces**
  - HP/GP, EMIO GPIO
- **PYNQ interface classes**
  - MMIO, Xlnk, DMA, GPIO

Image 2.5 Released

- **Updates to PYNQ 2.4 release**
  - Productivity Additions
    - Updated to JupyterLab 1.1.3
    - JupyterLab extensions support added
    - Support for multiple memories using mem_dict entries
    - Support for Device Tree Overlays delivered with PL overlays
    - Support for custom PL device communication using the Device metaclass
  - Programmable Logic Updates
    - All bitstreams built using Vivado 2019.1
    - XRT Support added (beta)
  - Repository Updates
    - Jenkins CI added
    - Sdist support added (removing all binaries from the repository)
  - SDBuild Updates
    - Boot partition built on Petalinux 2019.1

Adaptable.
Intelligent.

XILINX

PYNQ