ADIUVO
ENGINEERING AND TRAINING, LTD.

Professional PYNQ™

Adam Taylor

Agenda

- Who am I
- Introduction PYNQ™
  - What is PYNQ
  - How can it benefit your project
  - PYNQ Framework
  - Overlay Creation
- PYNQ Lab

ADIUVO
ENGINEERING AND TRAINING, LTD.

Your Instructor

Adam Taylor is a world recognized expert in design and development of embedded systems and FPGAs for several end applications. Throughout his career, Adam has used FPGAs to implement a wide variety of solutions, from RADAR to safety critical control systems (SIL4) and satellite systems. He also had interesting stops in image processing and cryptography along the way.

Adam is Chartered Engineer, Senior Member of the IEEE, Fellow of the Institute of Engineering and Technology.

He is the owner of the engineering and consultancy company, Adiuvo Engineering and Training, which develops embedded solutions for high reliability, mission critical and space applications. Current projects include ESA Plato, Lunar Gateway, Generic Space Imager, UKSA TreeView and several other clients across the world.

Requirements

To build along with this class, you will need to the following:

- AMD Vitis™ 2023.1 – This includes the AMD Vivado™ 2023.1 link
- PYNQ™ v3.0.1 link
- Avnet ZUBoard-1CG and Power Supply

Objective

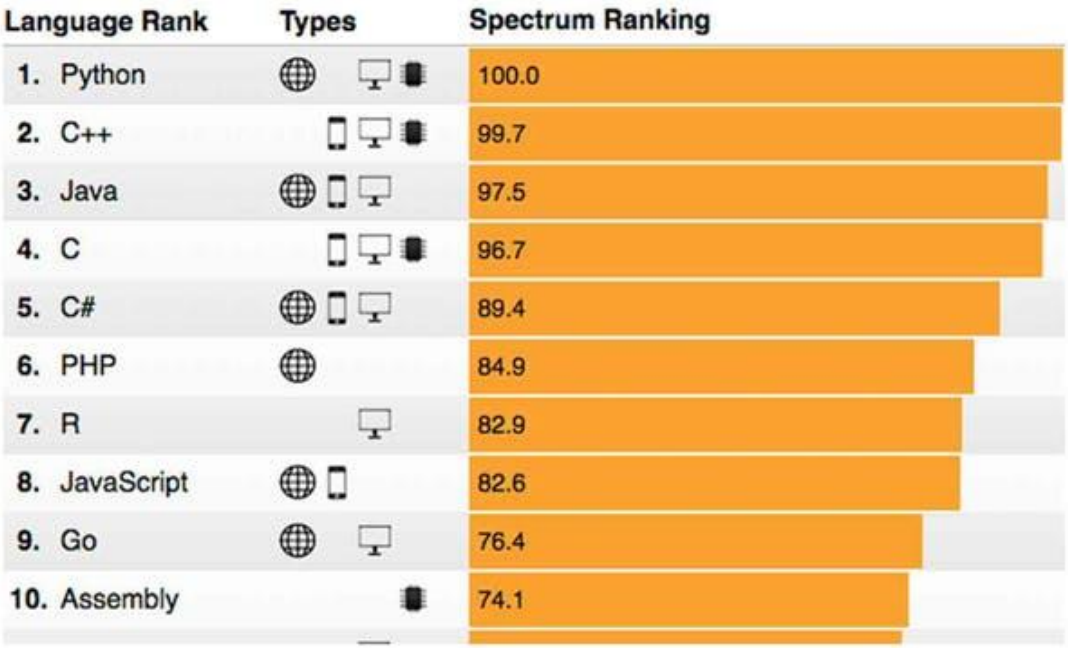
- Introduce PYNQ™ to professional developers
- Introduce the PYNQ framework and its key aspects
- Demonstrate how PYNQ can be used to test you HDL development

What is PYNQ™ – Introduction to the PYNQ Framework

ADIUVO
ENGINEERING AND TRAINING, LTD.

What is PYNQ™?

- PYNQ is an open source project started by AMD, which fuses the productivity of Python with the acceleration provided by programmable logic within the AMD Zynq™ 7000 SoCs, Zynq UltraScale+™ MPSoCs and Zynq UltraScale+™ RFSoCs



- Hosted at PYNQ.io

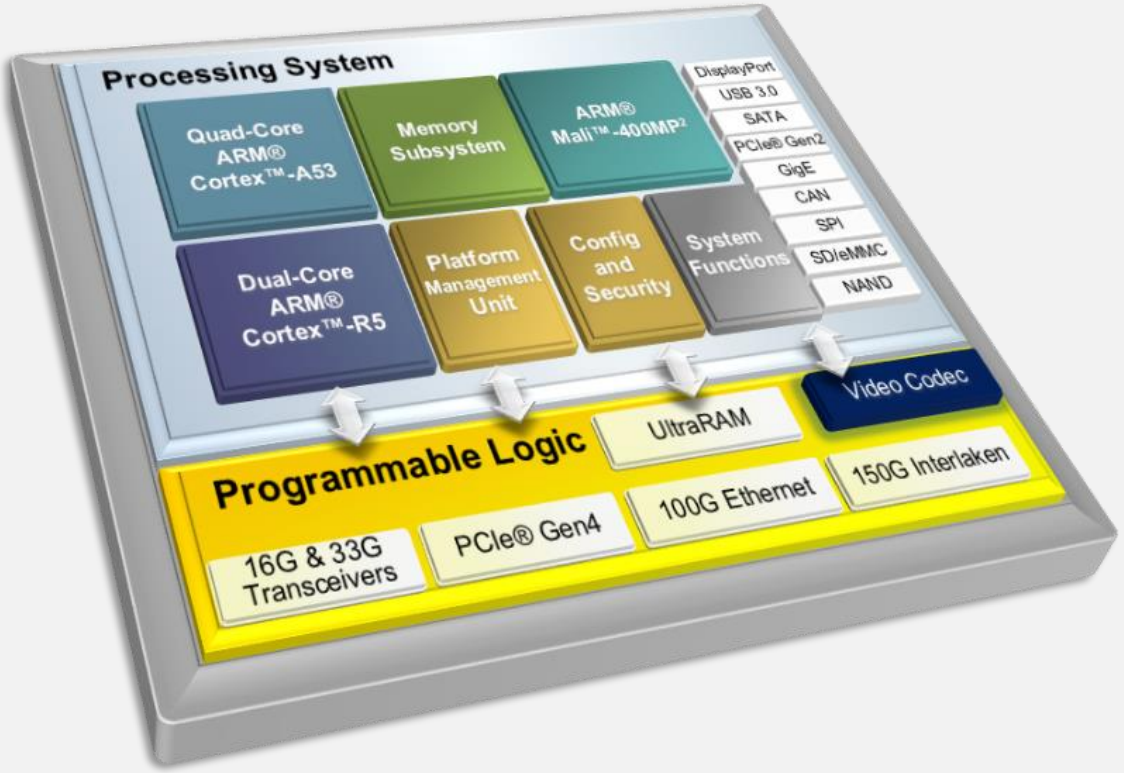
Heterogeneous SoC for Image Processing

Processing System (PS)

- Boots first
- Contains processors, fixed peripherals, clocks, memory, and memory controllers
- Dedicated silicon with software configurability

Programmable logic (PL)

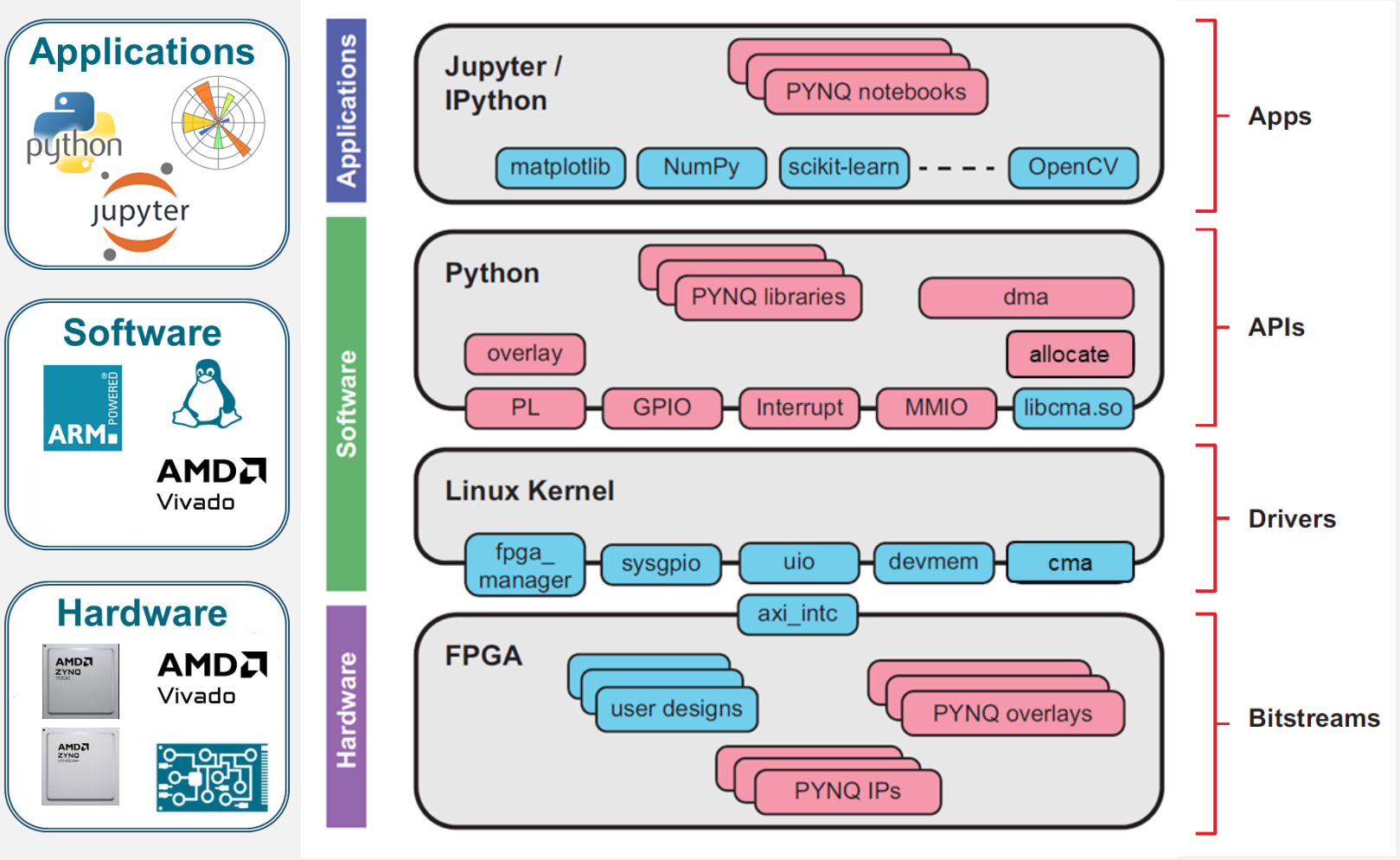
- Contains dedicated silicon resources: DSP48e, block RAM, high-speed serial, XADC, PCIe core, etc.
- Interconnects
- AXI-based
- Clocks and resets



Benefits of Heterogeneous SoC

Processor System	Programmable Logic
• High performance application processors and real time processors • Higher level algorithms can be implemented using industry standard frameworks – Embedded Linux & Real Time OS • Secure and safe system solution – Designed for ISO26262 / IEC61508 • Range of interfacing solutions – CAN, GigE, USB3, DisplayPort, etc. • Ability to accelerate functions using High Level Synthesis (C/C++)	• Any to any interfacing – MIPI DPHY is built in I/O cell • Ability to implement dedicated processing pipelines – significantly reduced DDR transactions • Large range of off the shelf video processing IP • Ability to use high-level synthesis and system level design tools • Deterministic and responsive solution

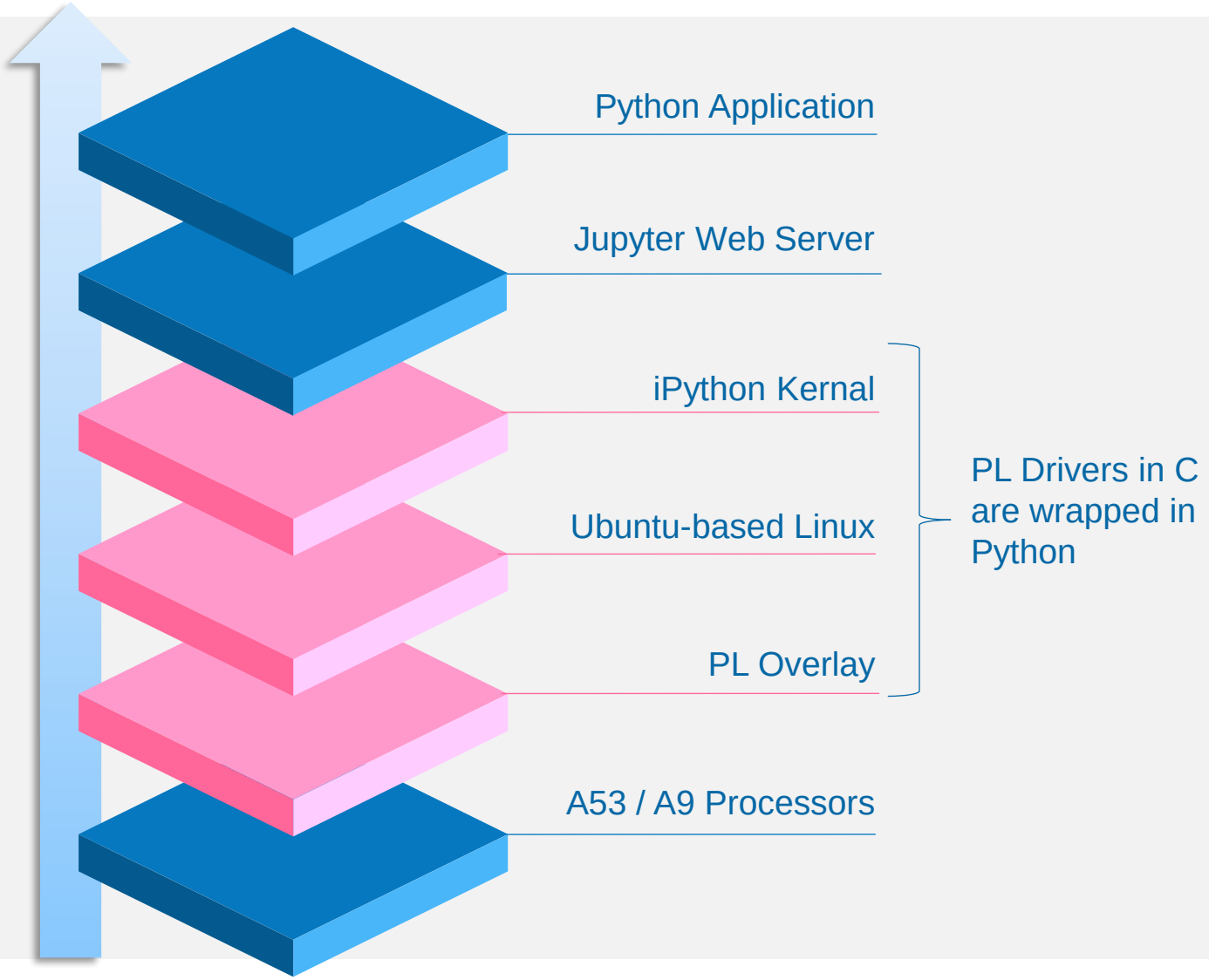
PYNQ™ Framework



AMD PYNQ

PYNQ™ Architecture

- PYNQ is built upon the AMD Petalinux flow
- Standard way to create PYNQ image for a custom board is via Petalinux BSP
- Uniform across all PYNQ-enabled boards



Overlays

- **Overlays are the design loaded into the programmable logic**

- Can be custom created or accessed via the PYNQ.IO community
- Range of Overlays in the community including

Machine Learning | Image Processing | RISC-V | Kalman Filter

- Base Overlay is the initial overlay which is created with the PYNQ™ image

Why PYNQ™?

ADIUVO
ENGINEERING AND TRAINING, LTD.

Opens performance of programmable logic to all types of developers

Frees Python programmers from the sequential software world and opens up PL acceleration

Don’t need to be a design expert

Growing community of users and overlays

Additional Benefits

ADIUVO
ENGINEERING AND TRAINING, LTD.

Responsive
Leverage the parallel processing capability provided by the PL

Deterministic
Creates processing pipeline competing for fewer shared resources

Power Efficient
Less off chip transactions to or from DDR memory, dedicated hardware implementation

Fast
Significant acceleration – 10x to 100x

Why should I learn PYNQ™?

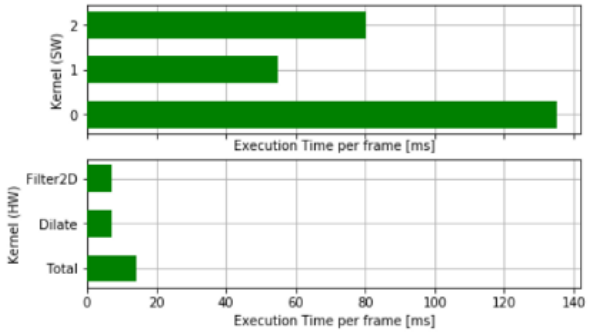
Tight coupling of processing system (PS) and programmable logic (PL) in the AMD Zynq™ 7000 SoCs and Zynq UltraScale+™ MPSoCs create a system which is:

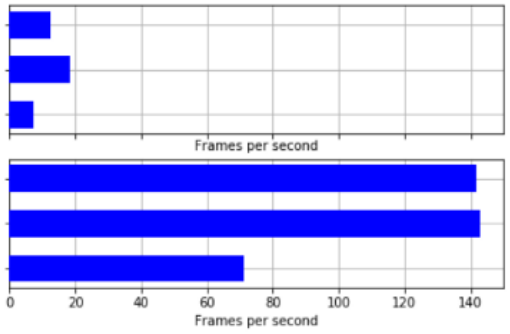
- Responsive – Leverage the parallel processing capability provided by the PL
- Deterministic – Creates processing pipeline competing for fewer shared resources
- Power Efficient – Less off chip transactions to or from DDR memory, dedicated hardware implementation is more efficient than discrete solutions.

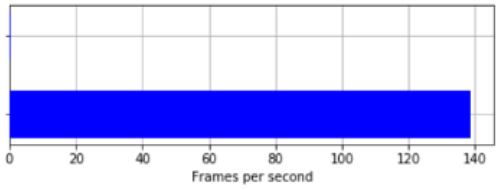
PYNQ frees Python programmers from the sequential software world and opens up the acceleration of programmable logic without the need to be a digital designer.

Why should I learn PYNQ™?

- Simple example of PYNQ in an image processing application
  - Image Filtering
    - SW < 20 Frames per Second
    - HW > 60 Frames per Second
  - Optical Flow
    - SW < 1 Frame per Second
    - HW > 120 Frames per Second







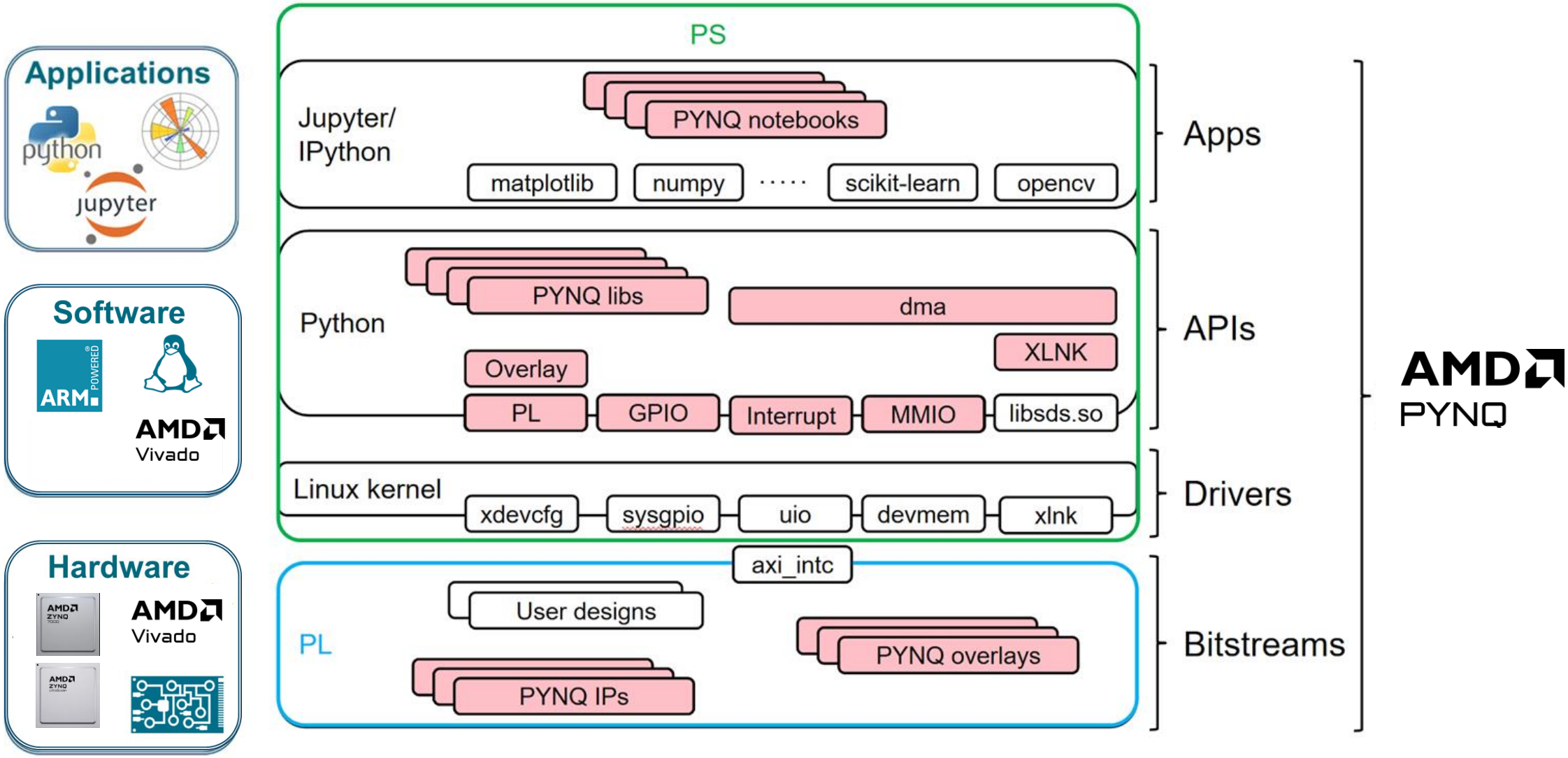
Use in Development

PYNQ™ enables developers to rapidly test and verify IP cores and algorithms on hardware.

- Leverage PYNQ libraries to easily communicate with IP modules which use AXI4 / AXI Lite Interfaces.
  - AXIS Interfaces can be easily accessed using Direct Memory Access.
- Verify interfaces which use AXI – demonstrate expected outcomes from register settings on the IP core being verified
- Verify implementation works at intended frequency

PYNQ™ Framework: interfacing Python with AMD SoC

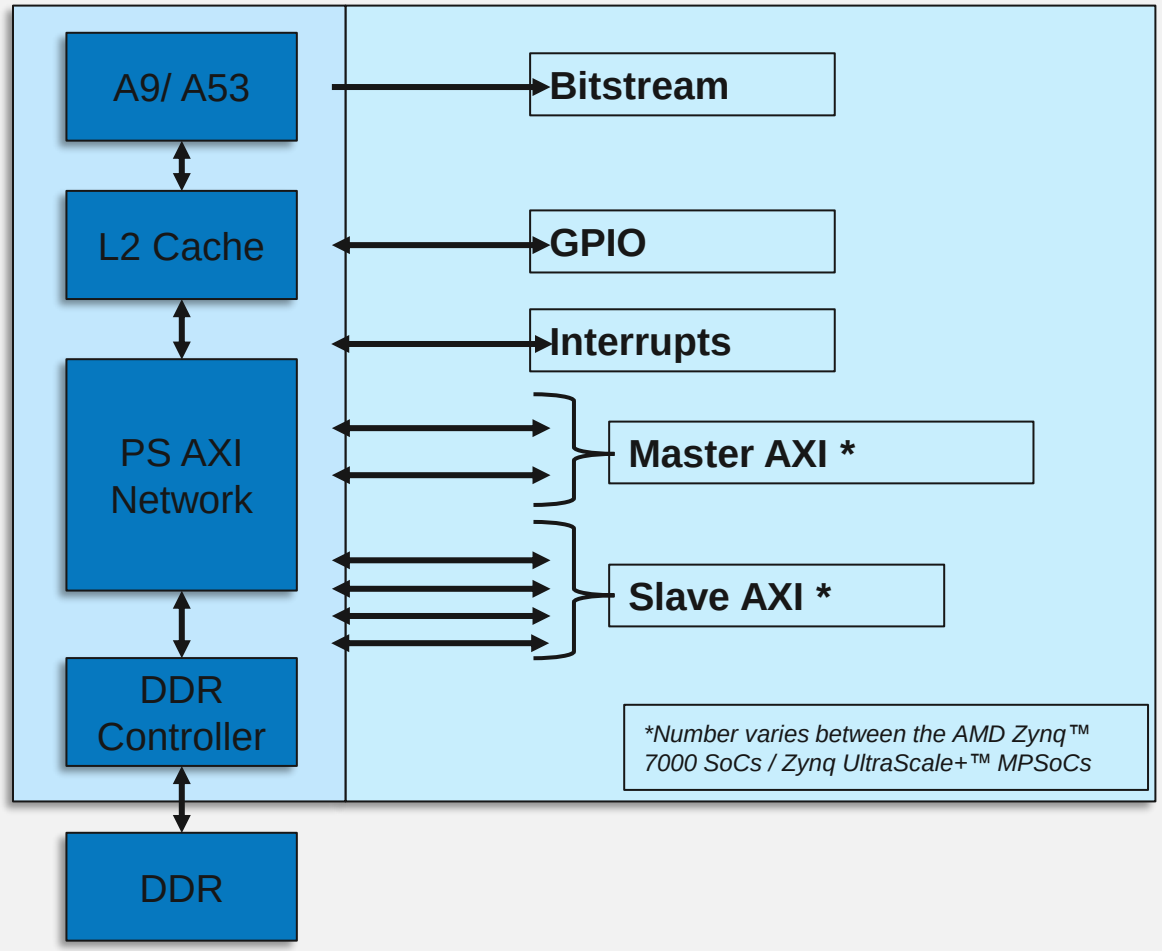
ADIUVO ENGINEERING AND TRAINING, LTD.



PYNQ™ Components

- To enhance user productivity, PYNQ combines:
  - Jupyter Notebooks
  - PYNQ Package
  - PYNQ Libs
  - PYNQ Classes
  - PYNQ IP
  - PYNQ Overlays

Working with the PL



- **Bitstream:** Configures the programmable logic for the desired application.
- → PYNQ™ uses the **fpga_manager** framework

- **GPIO:** Provides simple IO in both directions.
- → PYNQ uses the **sysgpio** driver

- **Interrupts:** Support interrupt generation from the programmable logic to the processing system.
- → PYNQ uses the **Userspace IO** driver

- **Master AXI Interfaces:** Used to transfer data between the PS to the PL when the PS is the initiator of the transaction.

→ PYNQ uses **devmem**

- **Slave AXI Interfaces:** Used to transfer data between the PS and PL when the PL is the initiator of the transaction.
- → PYNQ uses **CMA**

PYNQ™ Libraries

PYNQ provides several libraries which provide support for management of the processor and allow access to the low-level hardware including

IP Cores – Audio, AXI GPIO, AXI IIC, DMA, Logic Tools, Video

IOP – Arduino, Grove, RPI, PMOD

PYNQ MicroBlaze – AMD MicroBlaze™ Subsystem RPC and Library

PS / PL Interface – Interrupt, MMIO, PS GPIO, Xrt

PS Control – PMBus

PL Control – Overlay, PL and Bitstream Classes

PYNQ™ Package

- To access **xdevcfg, sysgpio, Userspace IO, devmem** and **xlnk**, we can use the PYNQ package.
- The PYNQ package contains much more than just this to get us working effectively with the PL

These packages can be grouped in to four groups:

1. Foundational Modules
2. Data Movement Modules
3. Additional Modules
4. Sub Packages

Latest information available at https://pynq.readthedocs.io

Fundamental Packages

pynq.ps - Facilitates management of the Processing System (PS) and PS/PL interface.

pynq.pl - Facilitates management of the Programmable Logic (PL).

pynq.overlay - Manages the state, drivers, and contents of overlays.

pynq.bitstream - Instantiates class for PL bitstream (full/partial).

pynq.devicetree - Instantiates the device tree segment class.

Data Movement Modules

pynq.mmio - Implements PYNQ™ Memory Mapped IO (MMIO) API

pynq.gpio - Implements PYNQ General-Purpose IO (GPIO) by wrapping the Linux Sysfs API

pynq.allocate - Implements Contiguous Memory Allocation for PYNQ DMA and PL accessible buffers

pynq.buffer - Implements a buffer class for DMA engines and accelerators

Additional Modules

pynq.interrupt - Leverages PYNQ™ asyncio

pynq.pmbus - PYNQ class for reading power measurements from PMBus

pynq.uio - Interacts directly with a UIO device

pynq.registers - Allows users to access registers easily

Pynq.utils – Functions to assist installation and testing

Sub-packages

pynq.lib - Contains sub-packages with drivers for PMOD, Arduino and Logictools PYNQ™ Libraries, and drivers for various communication controllers (GPIO, DMA, Video, Audio, etc.)

pynq.pl_server - Contains sub-packages for PL server to work across multiple devices. It also includes the overlay metadata parsers (e.g., tcl, hwh)

Overlays

Overlays are the design loaded into the programmable logic

Can be custom created or accessed via the PYNQ.IO community

Range of Overlays in the community including:

- Machine Learning
- Image Processing
- RISC-V
- Kalman filter

A board's optional Base Overlay provides access to various I/O on that board

Download and work with new overlays as required

You can also create your own, as we will see

Adding Overlays

There is an increasing amount of overlays available in the community. Most are shared and publicized at PYNQ.io

- Not a central repository but provides links to GitHub repos of PYNQ™ overlays

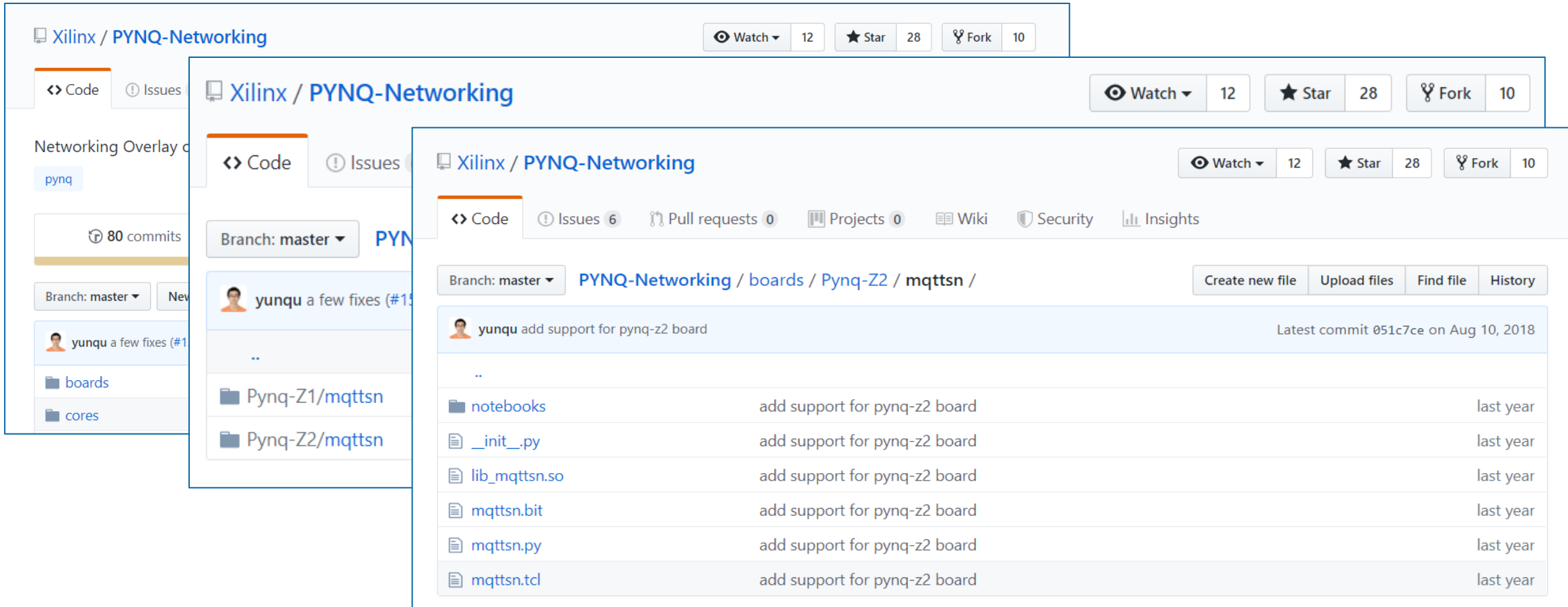
Installation is simple via the PYNQ terminal window

Before we install it, we need to check that it is suitable for our PYNQ board

We do this by checking the Boards directory on the GitHub Repo

Boards Directory

Board directory contains overlays and Notebooks for supported PYNQ™ boards.



Installing an Overlay

Make sure you have the right version of PYNQ™ for the Overlay

Make sure you install any pre-requisites first indicated in the Overlay Readme on GitHub

```
xilinx@pynq:~$ sudo pip3 install --upgrade git+https://github.com/Xilinx/PYNQ-ComputerVision.git
Collecting git+https://github.com/Xilinx/PYNQ-ComputerVision.git
  Cloning https://github.com/Xilinx/PYNQ-ComputerVision.git to /tmp/pip-pjzlsesj-build
Requirement already up-to-date: pynq>=2.3 in /usr/local/lib/python3.6/dist-packages (from pynq-c
v==2.3)
Installing collected packages: pynq-cv
  Running setup.py install for pynq-cv ... done
Successfully installed pynq-cv-2.3
xilinx@pynq:~$
```

Successful Installation

- Once successfully installed, you will see a new folder at the top level
- Under that folder, you will see the example notebooks provided to use that overlay

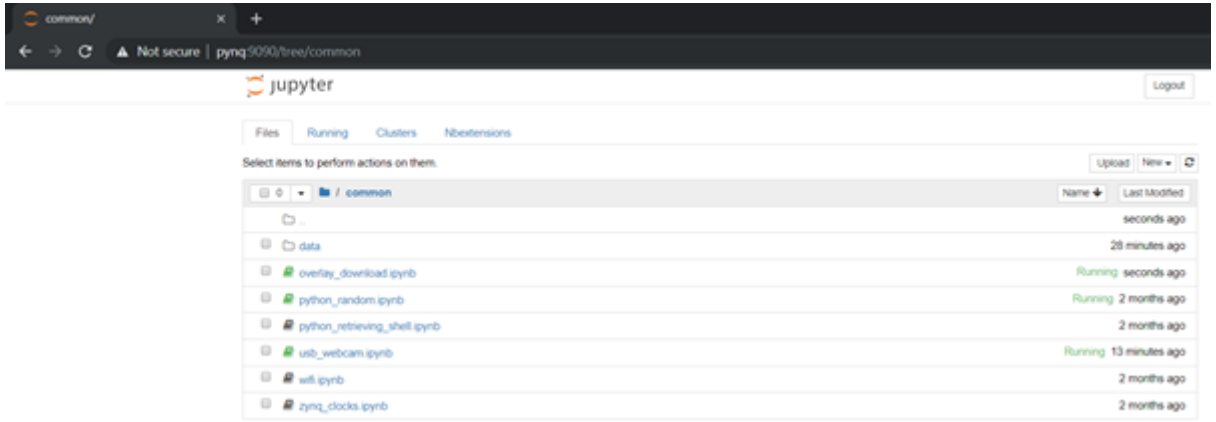
New Overlay Directory

Creating PYNQ™ Applications

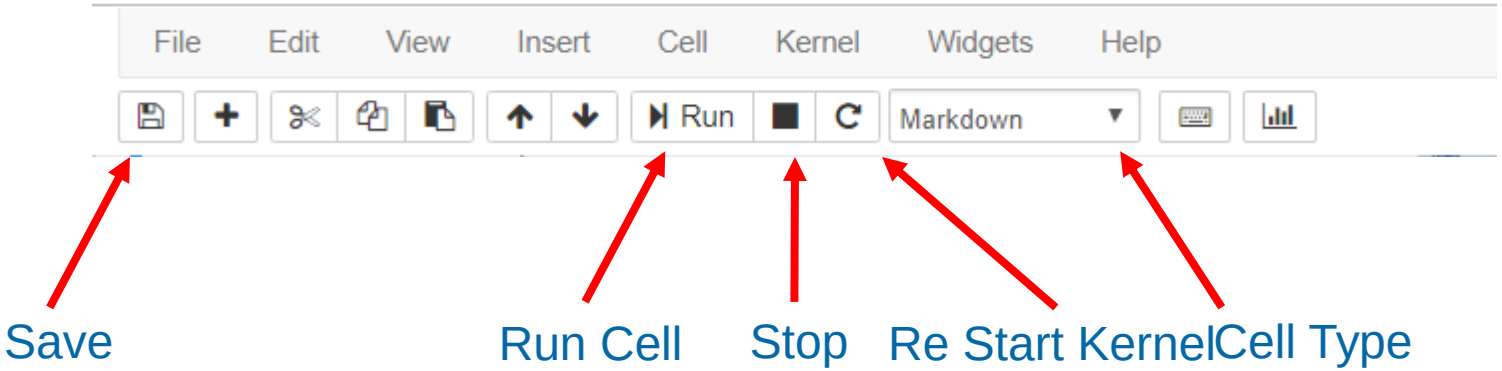
PYNQ uses web-based development in Jupyter notebooks

Can connect the Avnet MicroZed board over Ethernet (Z1, Z2, ZCU104, ZCU111) or USB-Ethernet / WIFI (Ultra96)

Access by going to **PYNQ:9090** in a browser on same network as PYNQ board



Working With Jupyter Notebooks



When Kernel first starts showing cell has not run

```
In [ ]: from pynq.overlays.base import BaseOverlay

        overlay = BaseOverlay('base.bit')
        trace_analyzer = overlay.trace_pmoda
```

Asterix shows cell currently running

```
In [*]: from pynq.overlays.base import BaseOverlay

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

Cell Execution Complete Number increments to show how often it was run

```
In [1]: from pynq.overlays.base import BaseOverlay

        overlay = BaseOverlay('base.bit')
        trace_analyzer = overlay.trace_pmoda
```

Working With Jupyter Notebooks

Not secure | pynq:9090/notebooks/base/trace/trace_analyzer_i2c.ipynb

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

Run Markdown

Step 1: Downloading the overlay

The trace analyzer for all the interfaces are exposed from the base overlay.

In [1]:
```
from pynq.overlays.base import BaseOverlay

overlay = BaseOverlay('base.bit')
trace_analyzer = overlay.trace_pmoda
```

Step 2: Setup the trace analyzer

The I2C controller on PMODA interface runs at 100KHz, it is recommended to oversample the transactions (capturing more than 2 samples per logic level). Hence here we set the sampling frequency to 400KHz.

It takes little time to capture the samples, therefore we set the number of analyzer samples to the maximum (64K samples).

In [2]:
```
trace_analyzer.setup(frequency_mhz=0.4, num_analyzer_samples=65535)
```

Step 3: Prepare the I2C sensor

We plug the PMOD TMP2 sensor to PMODA interface on the board.

In [3]:
```
from pynq.lib.pmod import Pmod_TMP2

sensor = Pmod_TMP2(overlay.PMODA)
```

Step 4: Run the trace analyzer

Start the run the trace analyzer, followed by the sensor reading the temperature values in degree C.

In [4]:
```
trace_analyzer.run()
reading = sensor.read()
trace_analyzer.stop()
```

Working with Jupyter Notebooks

Running Notebooks – Opening a notebook starts the notebook running
Can shutdown either in the notebook or here

Working with Hardware

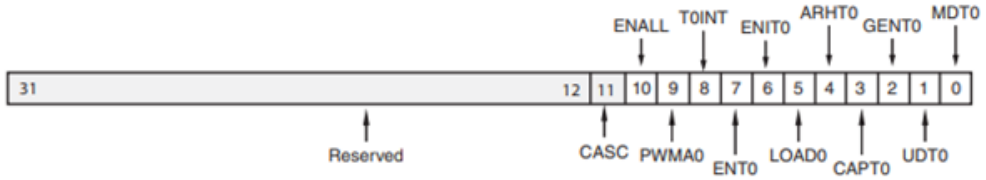
- PYNQ™ provides a range of features that assist with hardware development.
  - Automatic assignment of MMIO drivers for IPs without drivers (e.g. custom IP)
  - Automatic configuration of the clock from the design metadata
  - Comprehensive dictionary from metadata to work with the IP

```
IP Blocks
----------
axi_vdma_0          : pynq.lib.video.dma.AxiVDMA
v_tpg_0             : pynq.overlay.DefaultIP
color_convert_2_0   : pynq.lib.video.pipeline.ColorConverter
pixel_pack_2_0      : pynq.lib.video.pipeline.PixelPacker
axi_intc_0          : pynq.overlay.DefaultIP
zynq_ultra_ps_e_0   : pynq.overlay.DefaultIP
```

Working with Hardware

PYNQ™ is aware of Register Mappings for IP.

This saves looking up information in product guides and datasheets.




```
ol.axi_timer_0.register_map

RegisterMap {
  TCSR0 = Register(MDT0=0, UDT0=0, GENT0=0, CAPT0=0, ARHT0=0, LOAD0=0, ENIT0=0, ENT0=0, T0INT=0, PWMA0=0, ENALL=0, CASC=0),
  TLR0 = Register(TCLR0=0),
  TCR0 = Register(TCR0=0),
  TCSR1 = Register(MDT1=0, UDT1=0, GENT1=0, CAPT1=0, ARHT1=0, LOAD1=0, ENIT1=0, ENT1=0, T1INT=0, PWMA1=0, ENALL=0),
  TLR1 = Register(TCLR1=0),
  TCR1 = Register(TCR1=0)
}
```

```
In [29]: ol.axi_timer_0.register_map.TCR0.
Out[29]: ol.axi_timer_0.register_map.TCR0.address
         ol.axi_timer_0.register_map.TCR0.count
         ol.axi_timer_0.register_map.TCR0.create_subclass
         ol.axi_timer_0.register_map.TCR0.debug
         ol.axi_timer_0.register_map.TCR0.TCR0
         ol.axi_timer_0.register_map.TCR0.width
In [26]:
Out[26]:
```

Working with Hardware

Often we need to be able to work with interrupts in your hardware.

PYNQ™ makes this very simple by leveraging asyncio.

List of interrupts in a design can be obtained from the dictionary.

```
In [4]: ol.interrupt_pins

Out[4]: {'axi_vdma_0/s2mm_introut': {'controller': 'axi_intc_0',
          'index': 0,
          'fullpath': 'axi_vdma_0/s2mm_introut'},
         'xlconcat_0/In0': {'controller': 'axi_intc_0',
          'index': 0,
          'fullpath': 'xlconcat_0/In0'},
         'v_tpg_0/interrupt': {'controller': 'axi_intc_0',
          'index': 1,
          'fullpath': 'v_tpg_0/interrupt'},
         'xlconcat_0/In1': {'controller': 'axi_intc_0',
          'index': 1,
          'fullpath': 'xlconcat_0/In1'}}
```

IP Dictionary

The AMD Vivado™ software exports the hardware description metadata files. Python parses these files and constructs the hardware objects.

This can be very useful for supporting and developing your Python Application.

```
In [6]: ol.ip_dict
Out[6]: {'axi_vdma_0': {'type': 'xilinx.com:ip:axi_vdma:6.3',
          'mem_id': 'S_AXI_LITE',
          'memtype': 'REGISTER',
          'gpio': {},
          'interrupts': {'s2mm_introut': {'controller': 'axi_intc_0',
            'index': 0,
            'fullpath': 'axi_vdma_0/s2mm_introut'}},
          'parameters': {'C_S_AXI_LITE_ADDR_WIDTH': '9',
           'C_S_AXI_LITE_DATA_WIDTH': '32',
           'C_DLYTMR_RESOLUTION': '125',
           'C_PRMRY_IS_ACLK_ASYNC': '0',
           'C_ENABLE_VIDPRMTR_READS': '1',
           'C_DYNAMIC_RESOLUTION': '1',
           'C_NUM_FSTORES': '3',
           'C_USE_FSYNC': '1',
           'C_USE_MM2S_FSYNC': '0',
           'C_USE_S2MM_FSYNC': '2',
           'C_FLUSH_ON_FSYNC': '1',
           'C_INCLUDE_INTERNAL_GENLOCK': '1',
```

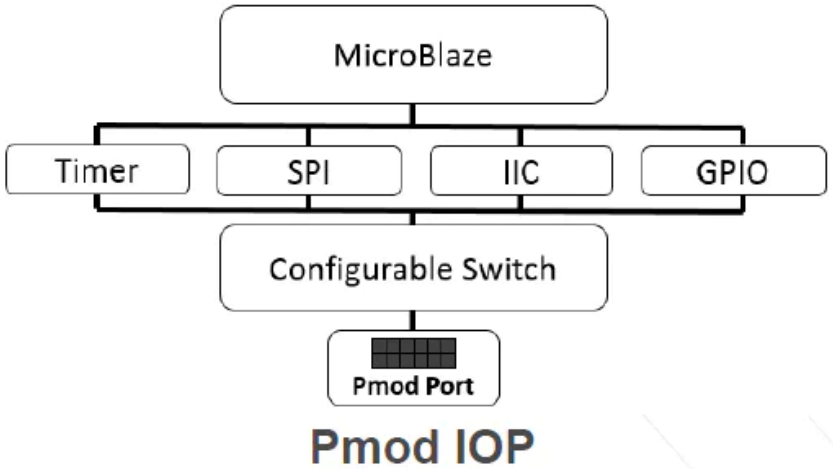
```
           'HAS_TLAST': '1'},
          'registers': {'mode': {'address_offset': 16,
            'size': 32,
            'access': 'write-only',
            'description': 'Data signal of mode',
            'fields': {'mode': {'bit_offset': 0,
              'bit_width': 32,
              'access': 'write-only',
              'description': 'Bit 31 to 0 Data signal of mode'}}},
          'alpha_V': {'address_offset': 24,
            'size': 32,
            'access': 'write-only',
            'description': 'Data signal of alpha_V',
            'fields': {'alpha_V': {'bit_offset': 0,
              'bit_width': 8,
              'access': 'write-only',
              'description': 'Bit 7 to 0 Data signal of alpha_V'},
```

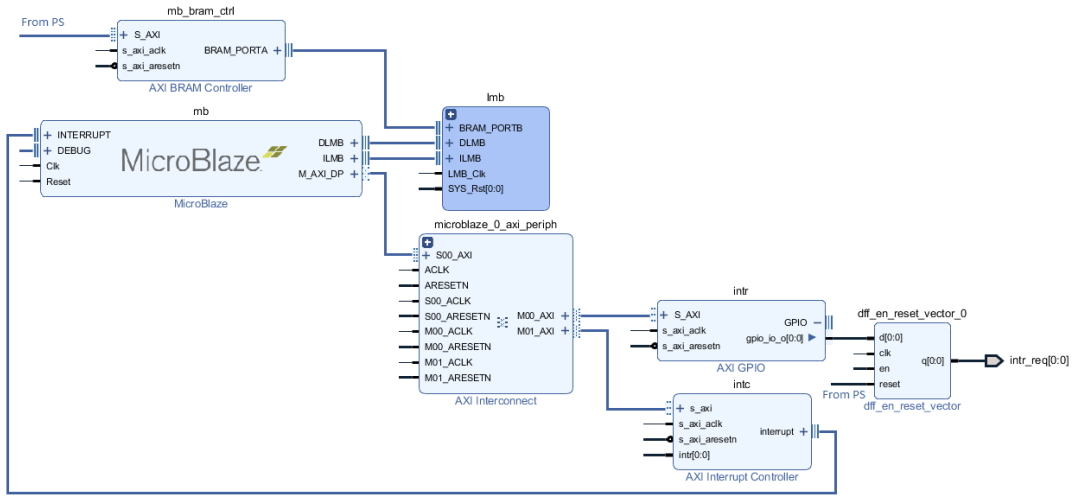
AMD MicroBlaze™ IO Processor

PYNQ™ provides ability to control MicroBlaze instances in the PL.

The MicroBlaze processor runs from block RAM in the programmable logic.

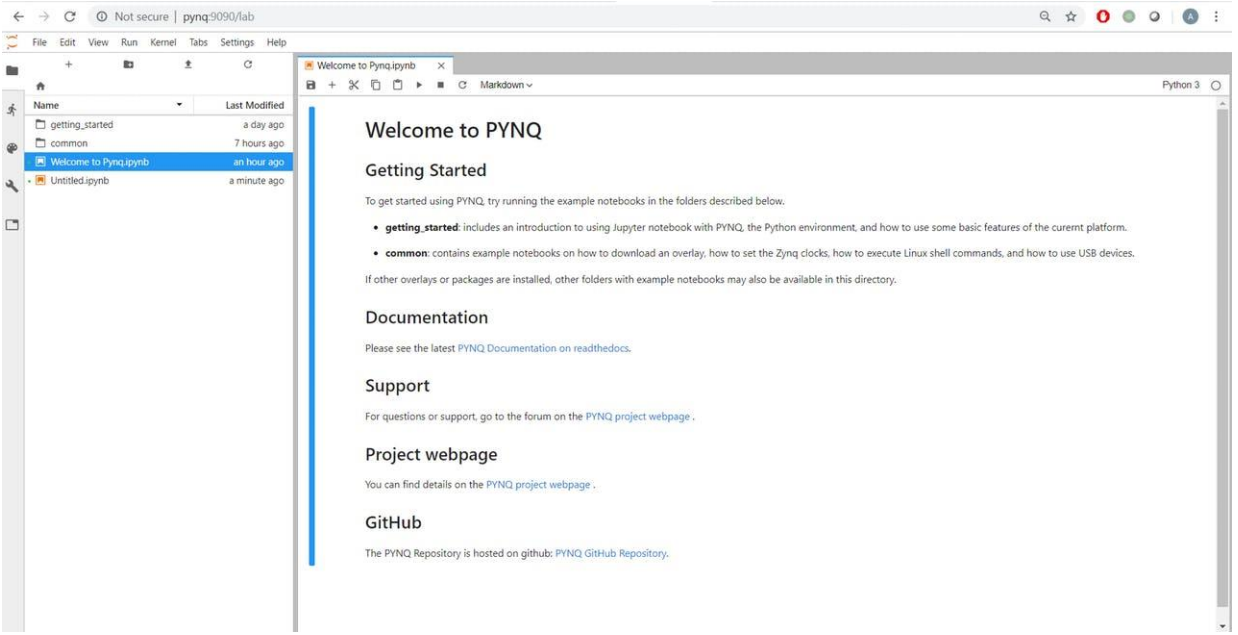
Application SW can be updated via Jupyter notebook.





Jupyter Labs

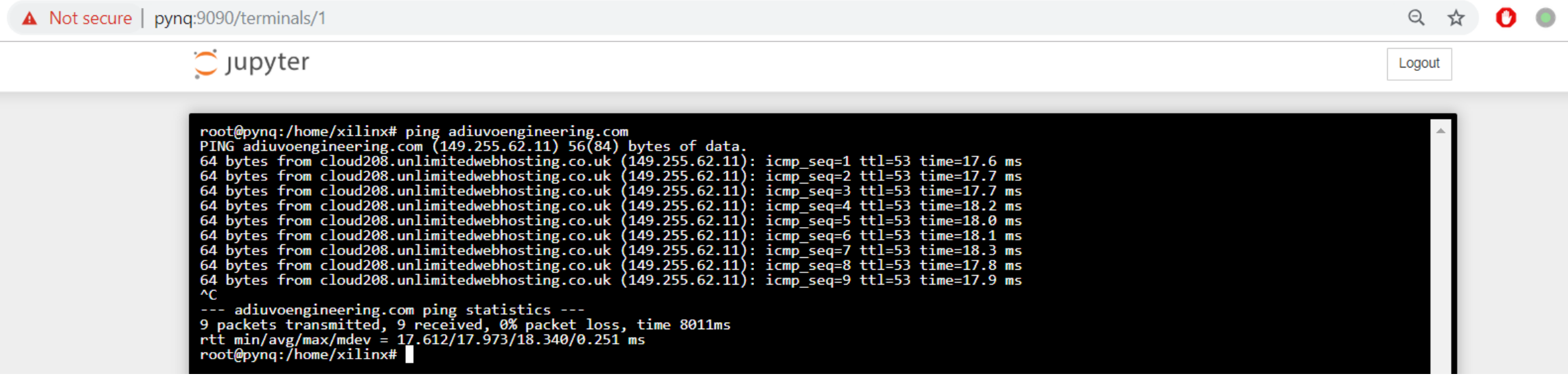
- Along with notebooks, we can also use Jupyter Labs
- Access via ***pynq:9090/lab***
- Jupyter lab, more customizable than the standard notebook interface and we can spin up notebooks, terminals and Python shells.



Terminal

We can open a terminal window in Jupyter notebook too

Very useful for adding overlays and doing standard terminal type applications



AMD sponsored this workshop, including engineering hours. AMD, and the AMD Arrow logo, MicroBlaze, PYNQ, UltraScale+, Vitis, Vivado, Zynq, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

ADIUVO
ENGINEERING AND TRAINING, LTD.

ADIUVO

ENGINEERING AND TRAINING, LTD.

www.adiuvoengineering.com

info@adiuvoengineering.com