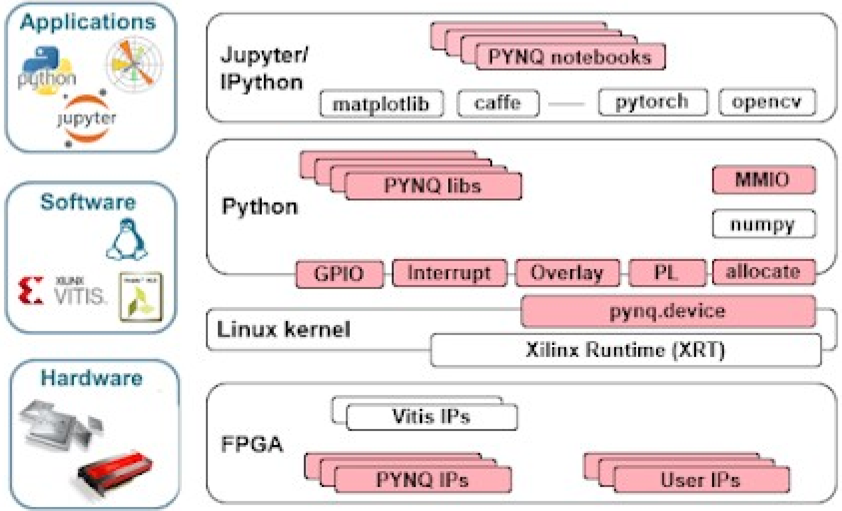
Introduction to PYNQ board

Digital Systems M, Module 2
Matteo Poggi, Università di Bologna

Introduction to PYNQ board

PYNQ is an open-source project from Xilinx that further brings FPGA programming to a higher level

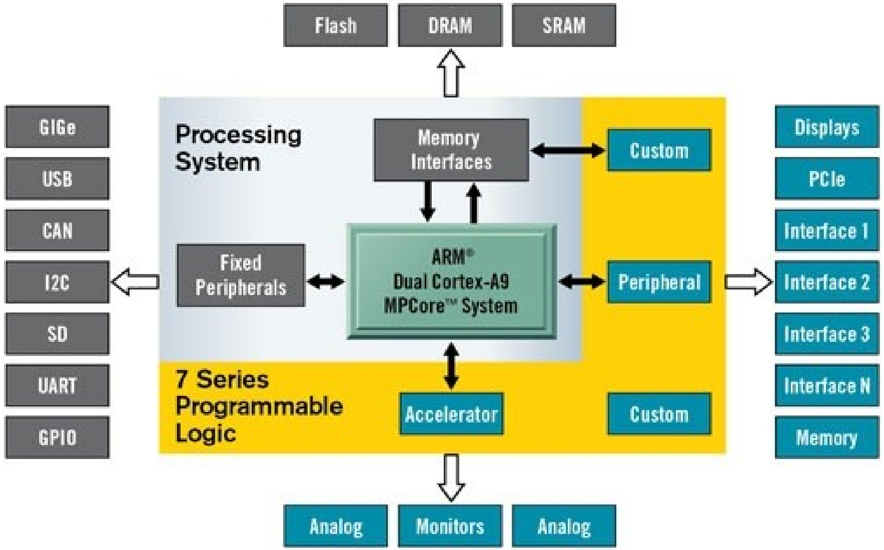
Designed for Python language and libraries, the PYNQ board brings together the benefits of programmable logic and microprocessors to build more capable and exciting digital systems



Introduction to PYNQ board

PYNQ boards are very similar to ZYNQ ones

They are made by a dual-core ARM Cortex-A9 processor (often referred to as **Processing System**, PS) and an FPGA (**Programmable Logic**, PL)



The PS can demand the execution of some functions to the PL by means of calls to specific hardware libraries, called **Overlays**

Overlays can both accelerate a software application or customize the hardware platform for a particular application

Introduction to PYNQ board

A software programmer can use an overlay in a similar way to a software library. It is particularly appealing when dealing with problems for which an FPGA can provide extremely good acceleration (e.g., image processing)

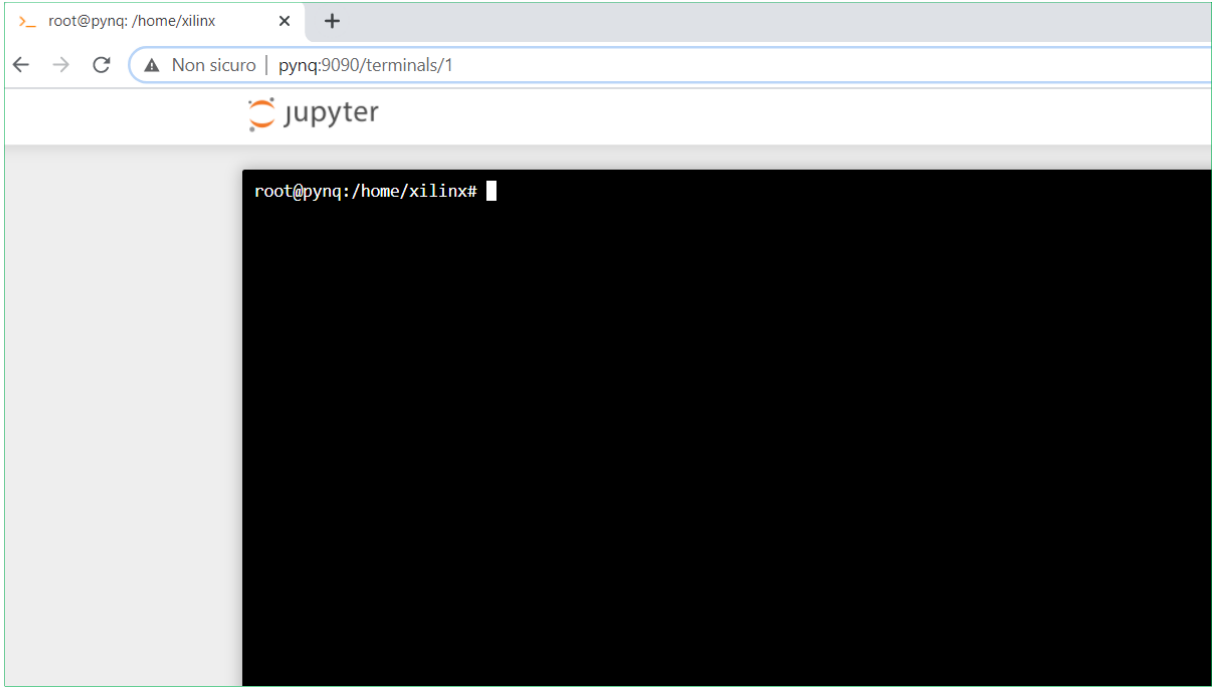
Overlays can be loaded to the FPGA dynamically just as software libraries and called when needed

PYNQ provides a Python interface to allow overlays in the **PL** to be controlled from Python running in the **PS**. While overlays are written by hardware specialists, software developers can call them via Python transparently and **without requiring** deep hardware design knowledge

Getting started

Introduction to PYNQ board

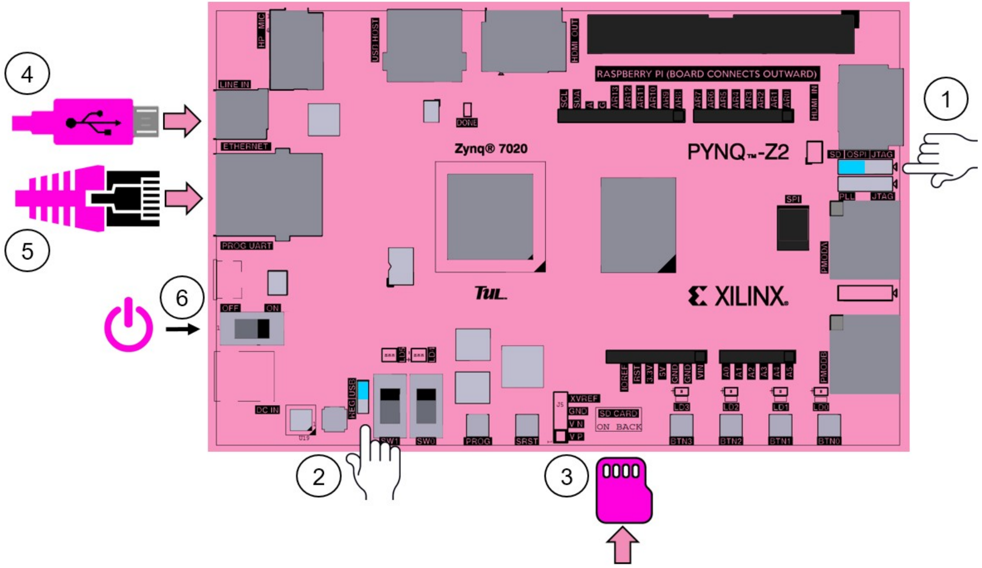
The PYNQ board can be easily configured and accessed from a standard PC and a web browser. Let's see how



Introduction to PYNQ board

The board comes with an SD card (3) containing a custom Linux distribution (actually, PYNQ v2.5)

- It can be powered with a regular power supply (6) or via **USB** (4). The modality is selected by a jumper (1)
- To access the board, we can either connect it (6) to a router or to our own PC
- A switch (2) allows to power it up (wait for all leds to light up)



Introduction to PYNQ board

The standard IP assigned to the board is 192.168.2.99

If the board is connected via ethernet to a router, let's browse to

http://pynq:9090

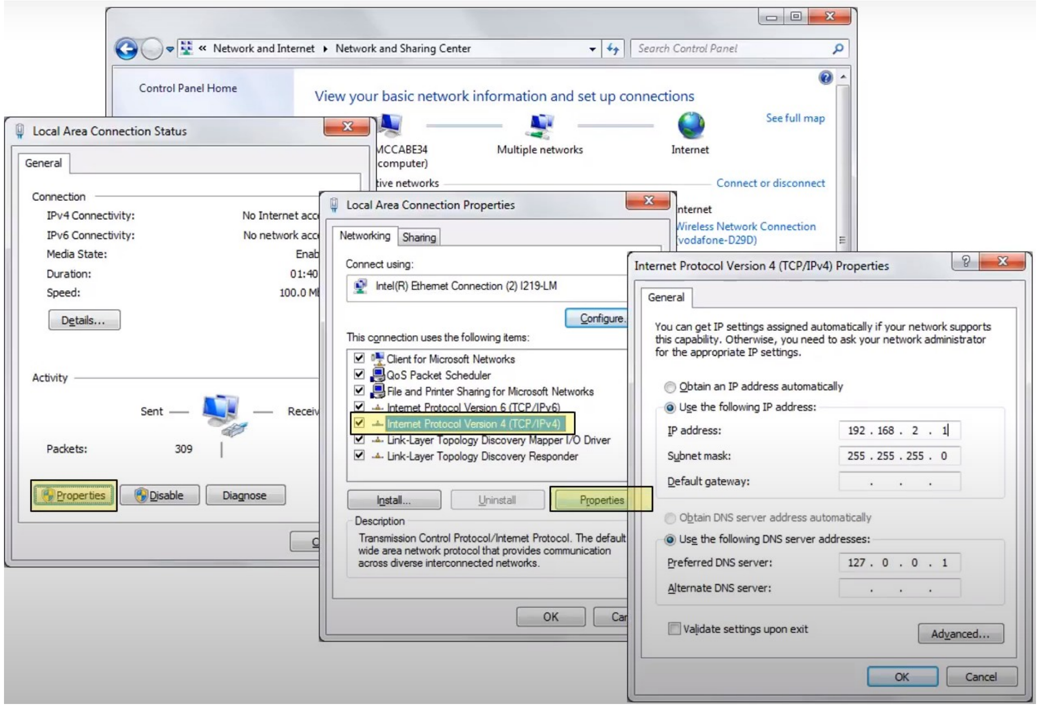
Otherwise, a **static IP** in the same subnetwork (192.168.2.x) is required for our laptop (see next slide). Then, let's browse to

http://192.168.2.99:9090

In both cases, the password to access the board is **xilinx**

Introduction to PYNQ board

Windows:

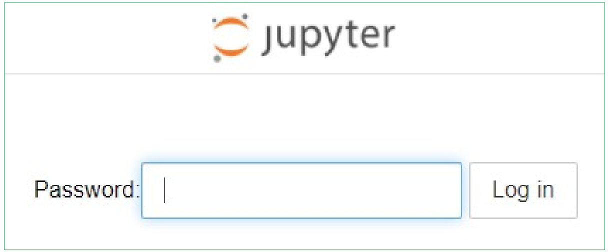


Linux:

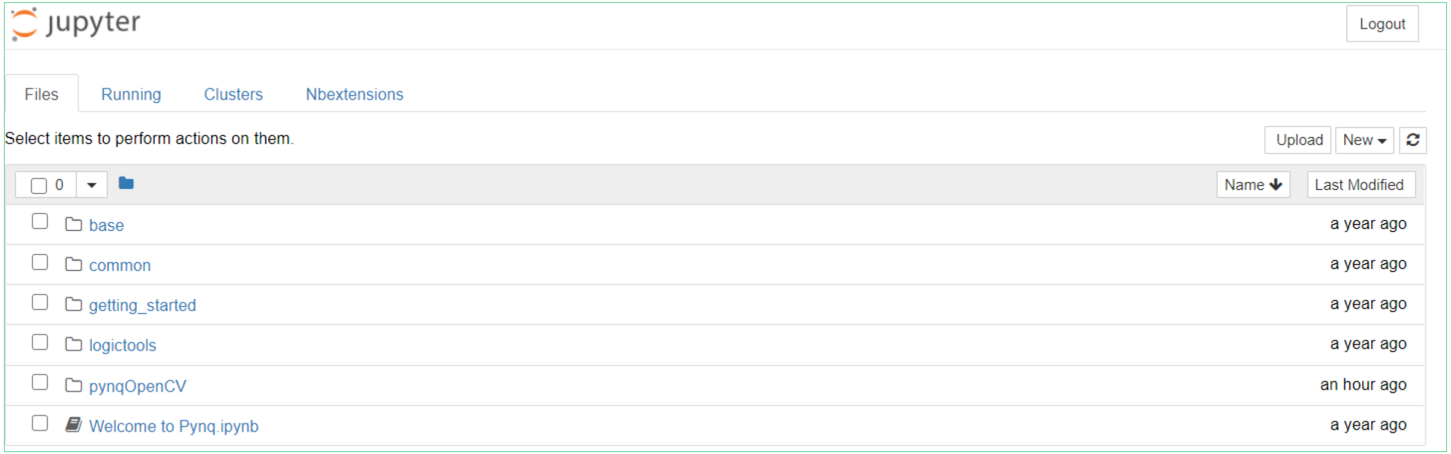
ifconfig eth0 192.168.2.1 netmask 255.255.255.0 up

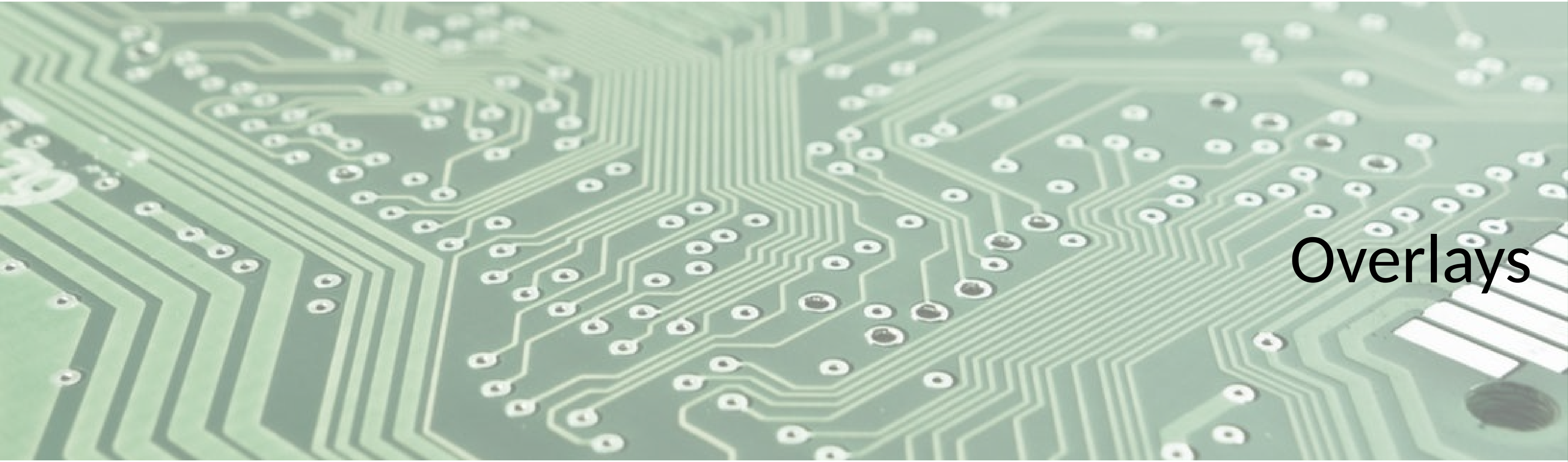
Introduction to PYNQ board

By browsing to the previous links, we should reach this page…



… and then be ready to play!





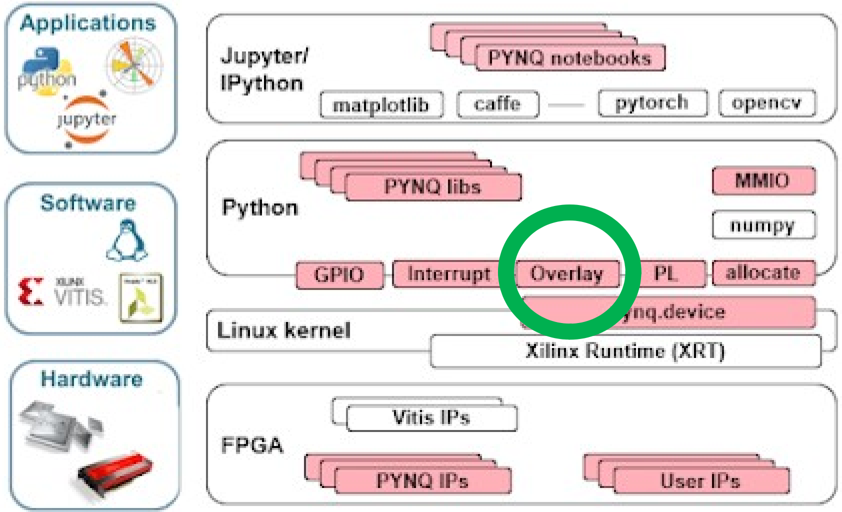
Overlays

Introduction to PYNQ board

Overlays allows for easy and transparent use of low-level IPs at Python level

An overlay **loads** a bitstream and give access to its functionalities

This is done by means of **drivers** objects



Example: call a custom IP, **scalar_add**, written in HLS to be run on Python
(more examples at https://github.com/PeterOgden/overlay_tutorial)

Let's consider a simple HLS function that returns the sum of two 32-bit integers

```
void add(int a, int b, int& c) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

After designing this IP, we can load it on Python side by means of an Overlay

Introduction to PYNQ board

From the **pynq** package (pre-installed on PYNQ SD image), we can import Overlay and create an overlay object from the bitstream containing the custom IP, then we can retrieve the specific IP

```
In [1]: from pynq import Overlay

        overlay = Overlay('/home/xilinx/tutorial_1.bit')
```

```
In [3]: add_ip = overlay.scalar_add
        help(add_ip)
```

The **help** function print the IP object extracted by the overlay

Introduction to PYNQ board

The IP core is loaded by a DefaultIP object, a driver exposing **read** and **write** APIs

According to the documentation generated by HLS, to use the IP core we need to write the two arguments to offset *0x10* and 0x18, then read the result back from *0x20*

```
In [4]:  add_ip.write(0x10, 4)
         add_ip.write(0x18, 5)
         add_ip.read(0x20)

Out[4]:  9
```

```
class DefaultIP(builtins.object)
 |  Driver for an IP without a more specific driver
 |
 |  This driver wraps an MMIO device and provides a base class
 |  for more specific drivers written later. It also provides
 |  access to GPIO outputs and interrupts inputs via attributes. More specific
 |  drivers should inherit from `DefaultIP` and include a
 |  `bindto` entry containing all of the IP that the driver
 |  should bind to. Subclasses meeting these requirements will
 |  automatically be registered.
 |
 |  Attributes
 |  ----------
 |  mmio : pynq.MMIO
 |      Underlying MMIO driver for the device
 |  _interrupts : dict
 |      Subset of the PL.interrupt_pins related to this IP
 |  _gpio : dict
 |      Subset of the PL.gpio_dict related to this IP
 |
 |  Methods defined here:
 |
 |  __init__(self, description)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  read(self, offset=0)
 |      Read from the MMIO device
 |
 |      Parameters
 |      ----------
 |      offset : int
 |          Address to read
 |
 |  write(self, offset, value)
 |      Write to the MMIO device
 |
 |      Parameters
 |      ----------
 |      offset : int
 |          Address to write to
 |      value : int or bytes
 |          Data to write
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Introduction to PYNQ board

The default driver is not user-friendly. A better solution would be a custom IP driver exposing a single **add** function. We can do this by extending the DefaultIP class

```
This driver wraps an MMIO device and provides a base class
for more specific drivers written later. It also provides
access to GPIO outputs and interrupts inputs via attributes. More specific
drivers should inherit from `DefaultIP` and include a
`bindto` entry containing all of the IP that the driver
should bind to. Subclasses meeting these requirements will
automatically be registered.
```

In [5]:
```
from pynq import DefaultIP

class AddDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:add:1.0']

    def add(self, a, b):
        self.write(0x10, a)
        self.write(0x18, b)
        return self.read(0x20)
```

Introduction to PYNQ board

Calling **help** on the overlay using the DefaultIP driver:

```
class Overlay(pynq.pl.Bitstream)
 |  Default documentation for overlay /home/xilinx/tutorial_1.bit. The following
 |  attributes are available on this overlay:
 |
 |  IP Blocks
 |  ----------
 |  scalar_add           : pynq.overlay.DefaultIP
 |
```

Calling **help** on the overlay using a custom driver:

```
class Overlay(pynq.pl.Bitstream)
 |  Default documentation for overlay /home/xilinx/tutorial_1.bit. The following
 |  attributes are available on this overlay:
 |
 |  IP Blocks
 |  ----------
 |  scalar_add           : __main__.AddDriver
 |
```

Introduction to PYNQ board

Now, we can access to the IP core and use a single **add** API, made available by our custom driver

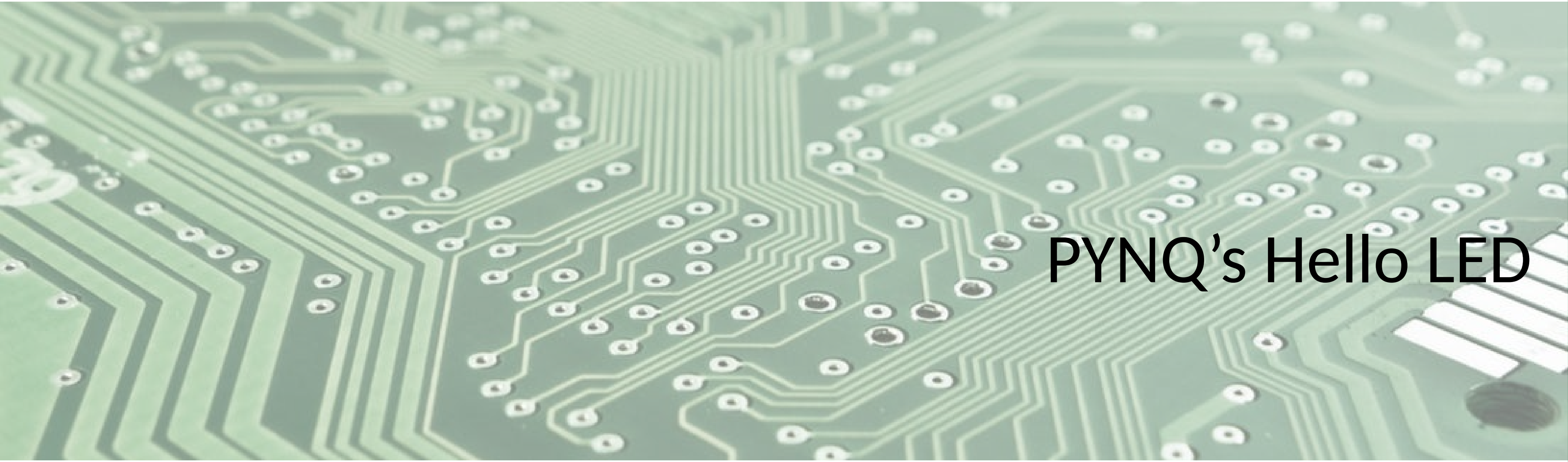
```
In [7]:  overlay.scalar_add.add(15,20)

Out[7]:  35
```

With the overlays mechanism we can easily reuse IPs, implement IP Hierarchies, custom overlays and more

Introduction to PYNQ board

The pynq library includes a number of drivers as part of the **pynq.lib** package. These include

- AXI GPIO
- AXI DMA (simple mode only)
- AXI VDMA
- AXI Interrupt Controller (internal use)
- Pynq-Z1 Audio IP
- Pynq-Z1 HDMI IP
- Color convert IP
- Pixel format conversion
- HDMI input and output frontends
- Pynq Microblaze program loading
- …



PYNQ’s Hello LED

Introduction to PYNQ board

All the peripherals on the PYNQ board are connected to PL. This means controllers must be implemented in an overlay before these peripherals can be used. The base overlay contains controllers for all the peripherals.

(Full tutorial at https://pynq.readthedocs.io/en/v1.3/5_programming_onboard.html)

By loading the base overlay, we can import specific classes for each peripherals from the **pynq.overlays.base** module

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

We can easily instantiate objects and manipulate the corresponding peripherals. For instance, we can access to a single LED and turn it **on/off**

```
led0 = base.leds[0] #Corresponds to LED LD0
led1 = base.leds[1] #Corresponds to LED LD1
led2 = base.leds[2] #Corresponds to LED LD2
led3 = base.leds[3] #Corresponds to LED LD3
```

```
led0.on()
```

```
led0.off()
```

We can also change the current status of a LED with the **toggle()** method

```
for i in range(20):
    led0.toggle()
    time.sleep(0.1)
```

Introduction to PYNQ board

We can use **switches** or **buttons** available on the board to turn **on/off** the LEDS

We first get access to LEDS/switches/buttons available

```
MAX_LEDS = 4
MAX_SWITCHES = 2
MAX_BUTTONS = 4

leds = [0] * MAX_LEDS
switches = [0] * MAX_SWITCHES
buttons = [0] * MAX_BUTTONS

for i in range(MAX_LEDS):
    leds[i] = base.leds[i]
for i in range(MAX_SWITCHES):
    switches[i] = base.switches[i]
for i in range(MAX_BUTTONS):
    buttons[i] = base.buttons[i]
```

Introduction to PYNQ board

Let's first reset the LEDs status by implementing a simple **clear_LEDs** function

```
In [7]:  # Helper function to clear LEDs
         def clear_LEDs(LED_nos=list(range(MAX_LEDS))):
             """Clear LEDS LD3-0 or the LEDs whose numbers appear in the list"""
             for i in LED_nos:
                 leds[i].off()

         clear_LEDs()
```

Then, we can implement the following policy:

- If Switch(0) is on, LED(2) and LED(0) will be on
- If Switch(1) is on, LED(3) and LED(1) will be on

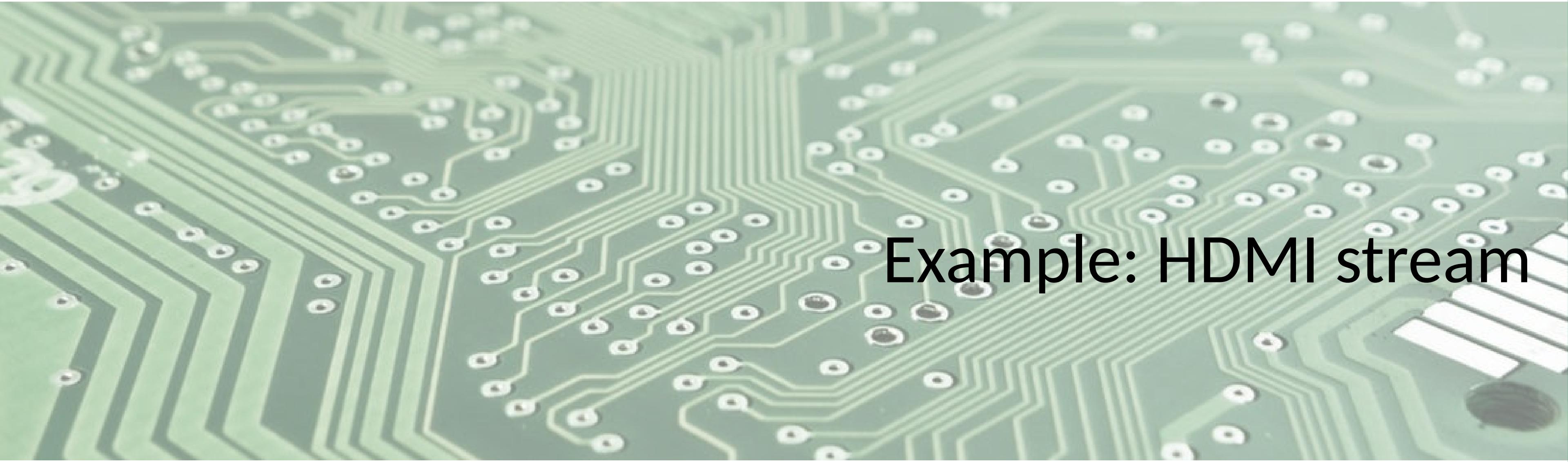
```
In [8]:  clear_LEDs()

         for i in range(MAX_LEDS):
             if switches[i%2].read():
                 leds[i].on()
             else:
                 leds[i].off()
```

Let’s try with another policy:

- Toogle LED(i) if Button(i) is pressed while Switch(0) is switched on

```
In [9]: import time

        clear_LEDs()

        while switches[0].read():
            for i in range(MAX_LEDS):
                if buttons[i].read():
                    leds[i].toggle()
                    time.sleep(.1)

        clear_LEDs()
```

Example: HDMI stream



Introduction to PYNQ board

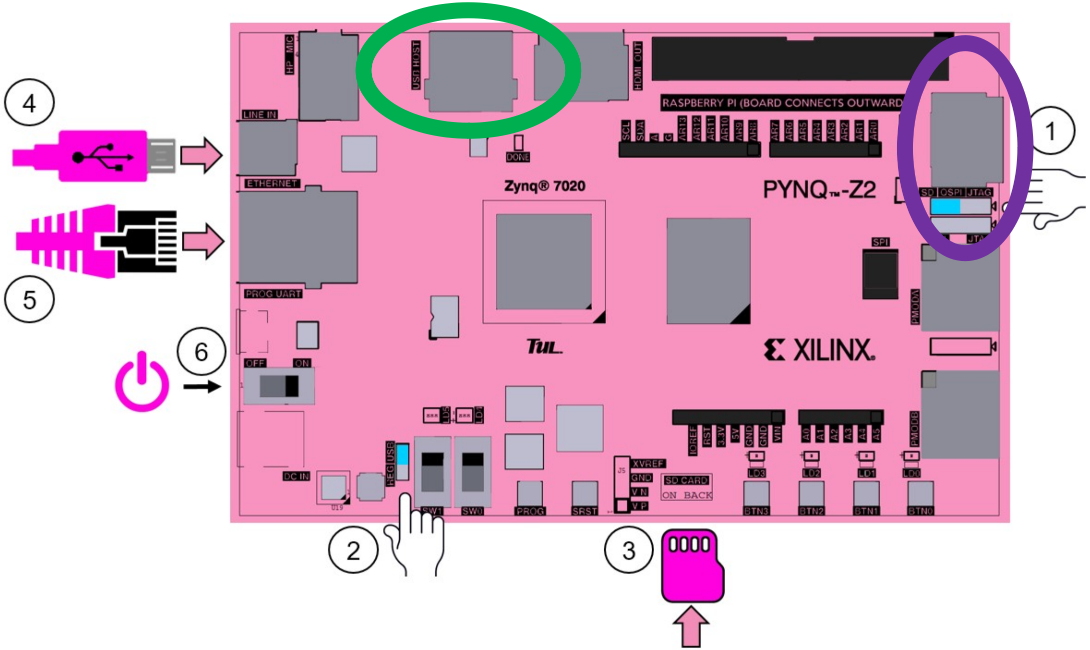
Let’s start with a simple pipeline to acquire an input video stream from a and show it into a monitor over HDMI ports

Source:
http://pynq:9090/notebooks/getting_started/5_base_overlay_video.ipynb

(other examples – not dealing with image processing – are at http://pynq:9090/notebooks/getting_started/)

Introduction to PYNQ board

The PYNQ board is equipped with 2x HDMI ports, respectively dedicated to handle input and output streams



Introduction to PYNQ board

The PYNQ board provides drivers to handle the HDMI ports

We can simply load the pynq.lib.video module

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

        base = BaseOverlay("base.bit")
        hdmi_in = base.video.hdmi_in
        hdmi_out = base.video.hdmi_out
```

Introduction to PYNQ board

We can easily configure the in/out HDMI handlers with few lines:

```
In [2]: hdmi_in.configure()
        hdmi_out.configure(hdmi_in.mode)

        hdmi_in.start()
        hdmi_out.start()
```

At first, we can just replicate the input stream on the output HDMI port with the **tie** method

```
In [3]: hdmi_in.tie(hdmi_out)
```

Equivalently, we can use the **readframe** and **writeframe** methods, respectively on the input and outputports, to grab a frame and forward it towards the output HDMI port

In [4]:
```
import time

numframes = 600
start = time.time()

for _ in range(numframes):
    f = hdmi_in.readframe()
    hdmi_out.writeframe(f)

end = time.time()
print("Frames per second:  " + str(numframes / (end - start)))
```

```
Frames per second:  34.12674016266714
```

This allows to **process** the frames before showing them on screen

```
In [5]: import cv2
        import numpy as np

        numframes = 10
        grayscale = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                               dtype=np.uint8)
        result = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                            dtype=np.uint8)

        start = time.time()

        for _ in range(numframes):
            inframe = hdmi_in.readframe()
            cv2.cvtColor(inframe,cv2.COLOR_BGR2GRAY,dst=grayscale)
            inframe.freebuffer()
            cv2.Laplacian(grayscale, cv2.CV_8U, dst=result)

            outframe = hdmi_out.newframe()
            cv2.cvtColor(result, cv2.COLOR_GRAY2BGR,dst=outframe)
            hdmi_out.writeframe(outframe)

        end = time.time()
        print("Frames per second:  " + str(numframes / (end - start)))

        Frames per second:  3.5044367097311837
```

Introduction to PYNQ board

Don’t forget to close the stream interfaces once we are done

```
In [6]: hdmi_out.close()
        hdmi_in.close()
```

More options:

- Cachable frames
  - Frames are cached to speed-up software processing (ARM). This slows HDMI processing and can be disabled with *hdmi_in.cacheable_frames = False*
- Gray-scale
  - We can delegate conversion to HW with *hdmi_in.configure(PIXEL_GRAY)*
- Other color-spaces
  - *hdmi_in.colorspace = COLOR_IN_YCBCR*

Example: HW accelerated filtering

Introduction to PYNQ board

In this example, the previous acquisition/visualization pipeline is extended with **hardware accelerated** image processing

Source: https://github.com/Xilinx/PYNQ-ComputerVision.git

To run it, we first need to upgrade Cython and install the PYNQ-ComputerVision package

pip3 install --upgrade Cython

pip3 install git+https://github.com/Xilinx/PYNQ-ComputerVision.git

Introduction to PYNQ board

We now have a new folder, **pynqOpenCV**, containing our examples

base	a year ago
common	2 hours ago
getting_started	22 minutes ago
logictools	2 hours ago
pynqOpenCV	2 days ago
Welcome to Pynq.ipynb	2 months ago

Source: http://pynq:9090/notebooks/pynqOpenCV/filter2d.ipynb

Introduction to PYNQ board

As usual, we first import the Overlay module. This time, we load a **custom** bitstream containing the 2D filter module

We also load the **xlnk memory manager**, a python class in charge of allocating **continuous memory** that is more efficient for PL IPs. It is compatible with numpy arrays

```
In [1]: # Load filter2D + dilate overlay
        from pynq import Overlay
        bareHDMI = Overlay("/usr/local/lib/python3.6/dist-packages/"
                           "pynq_cv/overlays/xv2Filter2DDilate.bit")
        import pynq_cv.overlays.xv2Filter2DDilate as xv2

        # Load xlnk memory mangager
        from pynq import Xlnk
        Xlnk.set_allocator_library("/usr/local/lib/python3.6/dist-packages/"
                                   "pynq_cv/overlays/xv2Filter2DDilate.so")
        mem_manager = Xlnk()

        hdmi_in = bareHDMI.video.hdmi_in
        hdmi_out = bareHDMI.video.hdmi_out
```

Introduction to PYNQ board

After configuring the HDMI input and output ports, we can run some 2D filtering on software side (nothing new here)

```
In [4]: import numpy as np
        import time
        import cv2

        #Sobel Vertical filter
        kernelF = np.array([[1.0,0.0,-1.0],[2.0,0.0,-2.0],[1.0,0.0,-1.0]],np.float32)

        numframes = 20

        start = time.time()
        for _ in range(numframes):
            inframe = hdmi_in.readframe()
            outframe = hdmi_out.newframe()
            cv2.filter2D(inframe, -1, kernelF, dst=outframe)
            inframe.freebuffer()
            hdmi_out.writeframe(outframe)
        end = time.time()
        print("Frames per second:  " + str(numframes / (end - start)))

        Frames per second:  3.3585366809661044
```

Introduction to PYNQ board

We might define a set of filters and **interactively** choose the one we want to apply using **ipywidgets**

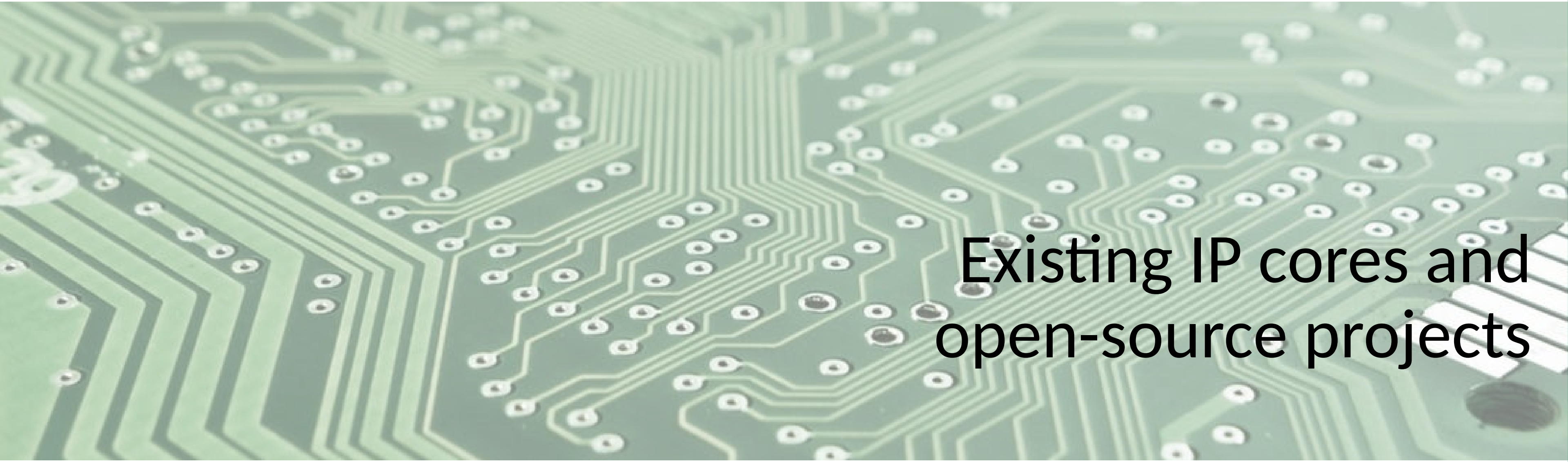
In [7]:
```
from ipywidgets import interact, interactive, fixed, interact_manual
from ipywidgets import IntSlider, FloatSlider
import ipywidgets as widgets

#Sobel Vertical filter
kernel_g = np.array([[1.0,0.0,-1.0],[2.0,0.0,-2.0],[1.0,0.0,-1.0]],np.float32)

def setKernelAndFilter3x3(kernelName):
    global kernel_g

    kernel_g = {
        'Laplacian high-pass': np.array([[0.0,1.0,0.0],[1.0,-4.0,1.0],
                                         [0.0,1.0,0.0]],np.float32),
        'Gaussian high-pass': np.array([[-0.0625,-0.125,-0.0625],
                                        [-0.125,0.75,-0.125],
                                        [-0.0625,-0.125,-0.0625]],np.float32),
        'Average blur':  np.ones((3,3),np.float32)/9.0,
        'Gaussian blur': np.array([[0.0625,0.125,0.0625],
                                   [0.125,0.25,0.125],
                                   [0.0625,0.125,0.0625]],np.float32),
        'Sobel ver': np.array([[1.0,0.0,-1.0],[2.0,0.0,-2.0],
                               [1.0,0.0,-1.0]],np.float32),
        'Sobel hor': np.array([[1.0,2.0,1.0],[0.0,0.0,0.0],
                               [-1.0,-2.0,-1.0]],np.float32)
    }.get(kernelName, np.ones((3,3),np.float32)/9.0)

interact(setKernelAndFilter3x3, kernelName
         = ['Sobel ver','Sobel hor','Laplacian high-pass','Gaussian high-pass','Average blur',
           'Gaussian blur',]);
```

Introduction to PYNQ board

Let's now call the 2D filter (implemented as IP core) through the Overlay

In order to allow kernel redefinition on the fly, subsequent function calls are run as **threads** thanks to the **Thread class**

```
In [8]: import numpy as np
        import cv2
        from threading import Thread

        def loop_hw2_app():
            global kernel_g

            numframes = 600

            start=time.time()
            for _ in range(numframes):
                outframe = hdmi_out.newframe()
                inframe = hdmi_in.readframe()
                xv2.filter2D(inframe, -1, kernel_g, dst=outframe, borderType=cv2.BORDER_CONSTANT)
                hdmi_out.writeframe(outframe)
                inframe.freebuffer()
            end=time.time()
            print("Frames per second:  " + str(numframes / (end - start)))

        t = Thread(target=loop_hw2_app)
        t.start()

        Frames per second:  43.98824715869524
```



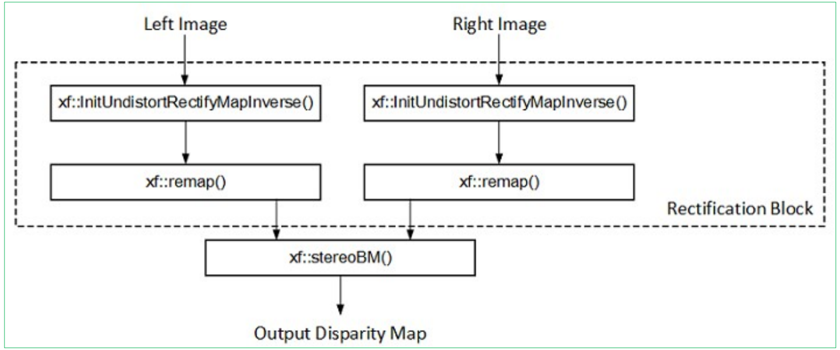
Existing IP cores and open-source projects

Introduction to PYNQ board

Xilinx provides plenty of IP cores implementing OpenCV functions!

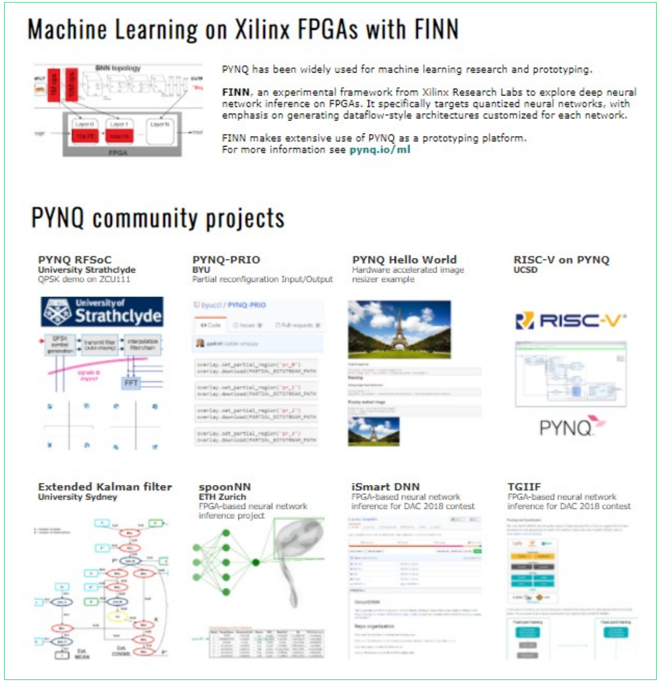
More than 60 kernels at
https://github.com/Xilinx/xfopencv
https://github.com/Xilinx/Vitis_Libraries/tree/master/vision

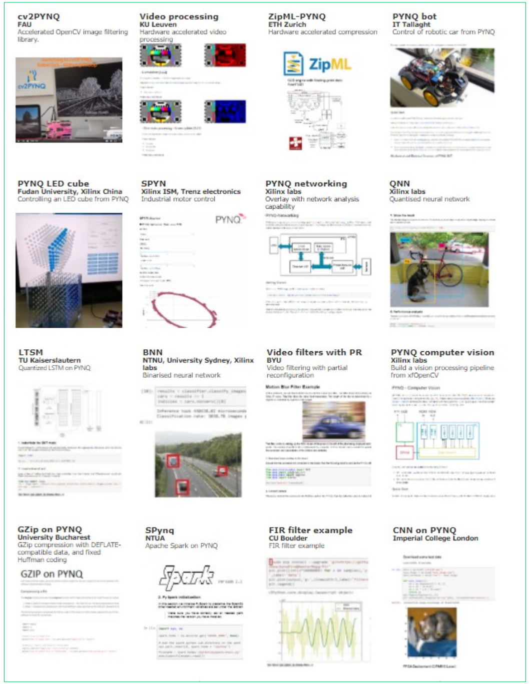
e.g. stereo matching pipeline



Introduction to PYNQ board

Finally, a list of PYNQ open-source projects is available at http://www.pynq.io/examples





References

- PYNQ: Python productivity - http://www.pynq.io/
- Python productivity for Zynq (Pynq) - https://pynq.readthedocs.io/en/latest/index.html
- PYNQ Computer Vision - https://github.com/Xilinx/PYNQ-ComputerVision