



Bridge of Life
Education

PYNQ Composable Pipeline & Partial Programming

2022/5/11



Outline

- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

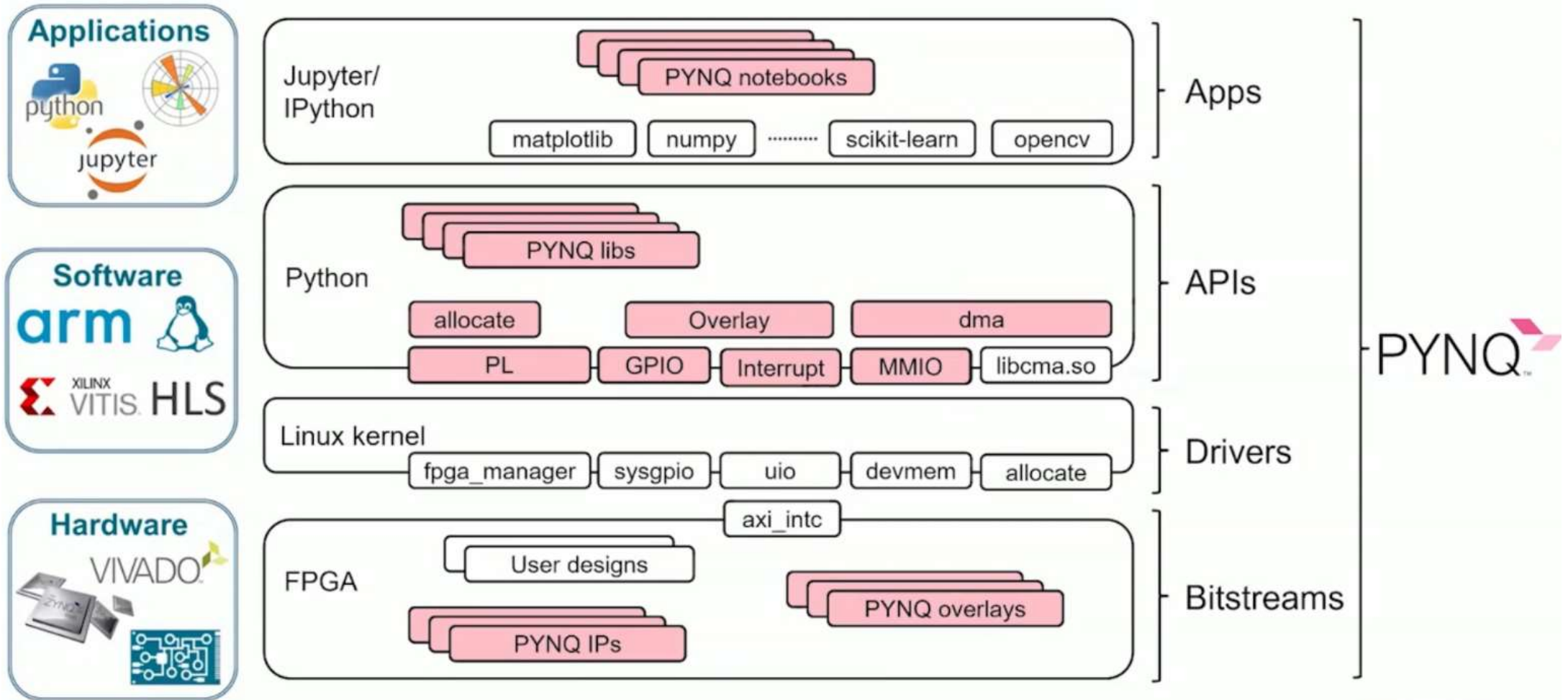
- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

PYNQ Introduction

- PYNQ is an open-source project from Xilinx
- PYNQ provides a Jupyter-based framework with Python APIs for using Xilinx platforms
- PYNQ supports Zynq and Zynq Ultrascale+, Zynq RFSoc, Alveo and AWS-F1 instances
- PYNQ enables architects, engineers and programmers who design embedded systems to use Zynq devices, without having to use ASIC-style design tools to design programmable logic circuits

PYNQ Introduction

Framework



PYNQ Background

Overlay

- Programmable logic circuits are presented as hardware libraries called overlays
- A software engineer can select the overlay that best matches their application
- Creating a new overlay still requires engineers with expertise in designing programmable logic circuits
- Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications

- **PYNQ**
 - PYNQ Introduction & Background
 - **PYNQ Overlays**
 - PYNQ Libraries
 - Overlay Design Methodology
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

PYNQ Overlays

Loading an Overlay

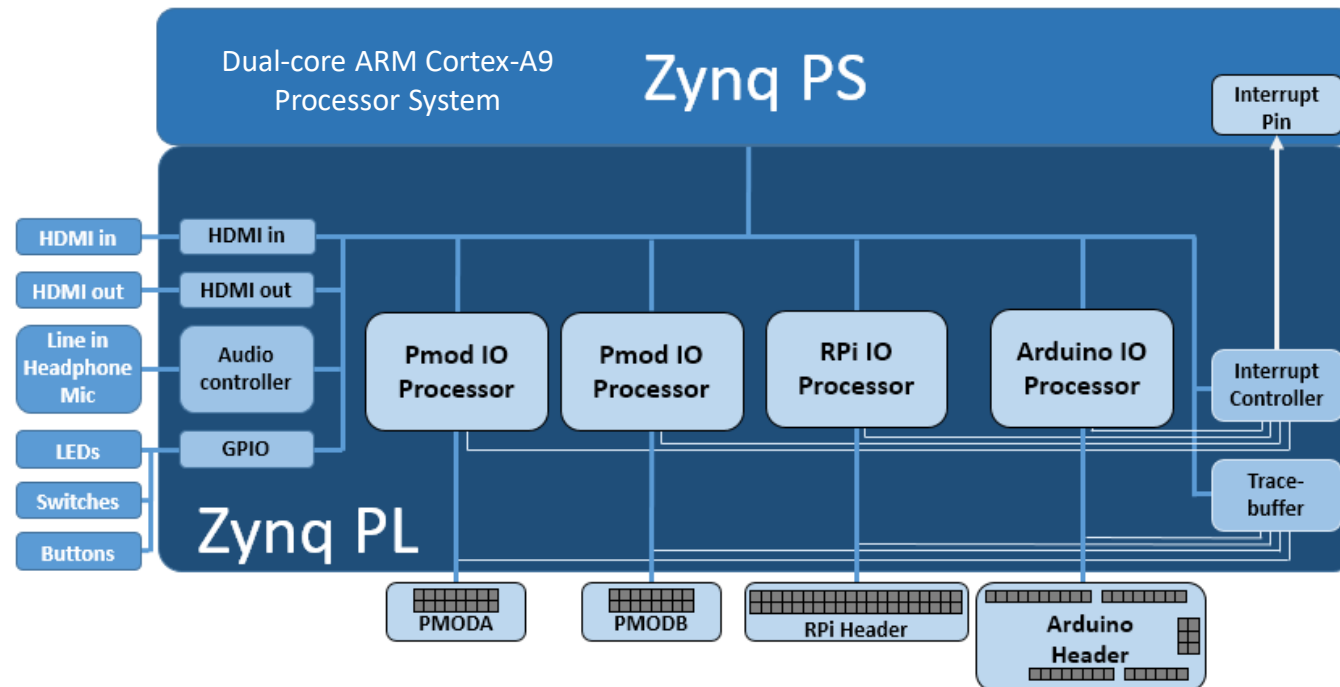
- By default, an overlay (bitstream) called base is downloaded into the Programmable Logic (PL) at boot time
- New overlays can be installed or copied to the board and can be loaded into the PL as the system is running
- An overlay usually includes:
 - A bitstream to configure the FPGA fabric
 - A Vivado design HWH file to determine the available IP
 - Python API that exposes the IPs as attributes
- The PYNQ Overlay class can be used to load an overlay

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


PYNQ Overlays

Base Overlay on PYNQ-Z2

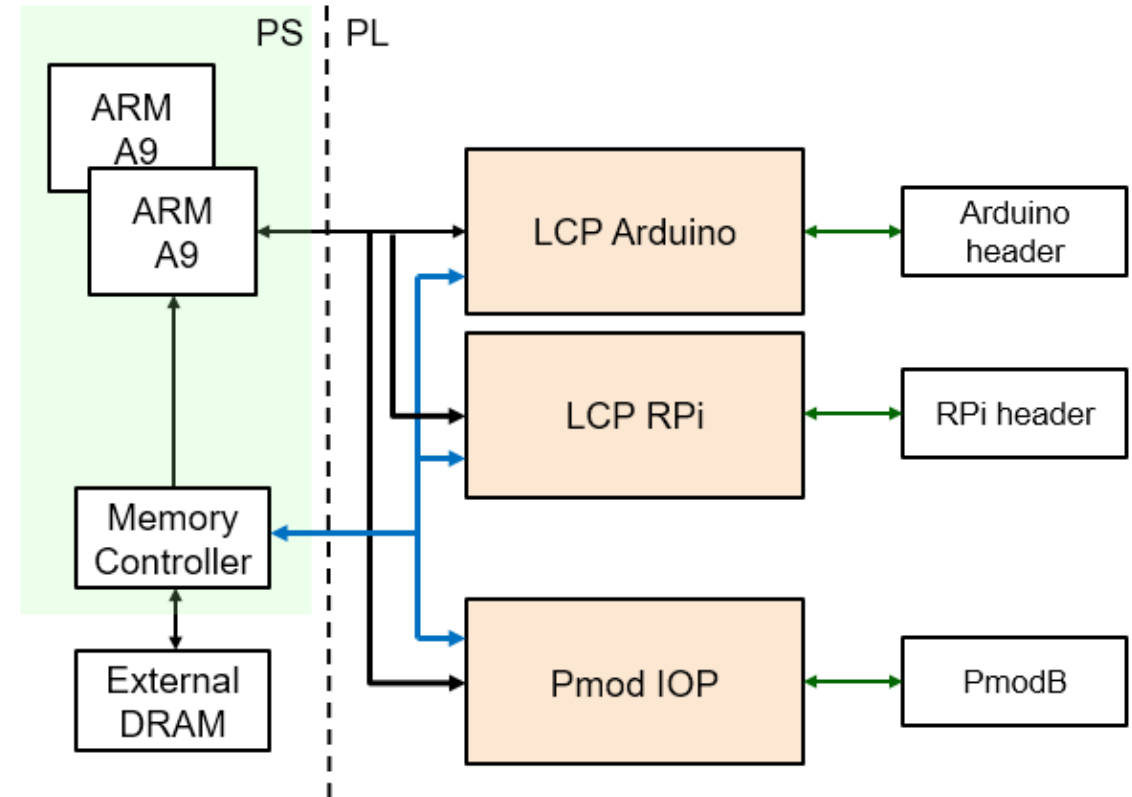
- The base overlay on PYNQ-Z2 includes the following hardware:
 - HDMI (Input and Output), Audio codec, User LEDs, Switches, Pushbuttons, 2x Pmod PYNQ MicroBlaze, Arduino PYNQ MicroBlaze, RPi (Raspberry Pi) PYNQ MicroBlaze, 4x Trace Analyzer (PMODA, PMODB, ARDUINO, RASPBERRYPI)



PYNQ Overlays

Logictools Overlay on PYNQ-Z2

- The logictools overlay can also has a trace analyzer to capture data from the IO interface for analysis and debug
- The PYNQ-Z2 logictools overlay has two instances of the logictools LCP (Logic Control Processor); one connected to the Arduino header, and the other connected to the RPi (Raspberry Pi) header
- The PYNQ-Z2 logictools overlay also includes a Pmod IOP connected to PmodB



- **PYNQ**
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - **PYNQ Libraries**
 - Overlay Design Methodology
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

PYNQ Libraries

Overlay

- The .bit file path can be provided as a relative, or absolute path. The Overlay class will also search the packages directory for installed packages, and download an overlay found in this location

```
from pynq import Overlay
```

```
base = Overlay("base.bit")    # bitstream implicitly downloaded to PL
```

```
base = Overlay("base.bit", download=False) # Overlay is instantiated, but bitstream is not downloaded to PL
```

```
base.download()              # Explicitly download bitstream to PL
```

```
base.is_loaded()             # Checks if a bitstream is loaded
```

```
base.reset()                 # Resets all the dictionaries kept in the overlay
```

```
base.load_ip_data(myIP, data) # Provides a function to write data to the memory space of an IP  
                               # data is assumed to be in binary format
```

PYNQ Libraries

Audio

- The Audio module provides methods to read audio from the input microphone, play audio to the output speaker, or read and write audio files

Initialization and playing wav

```
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
pAudio = base.audio
pAudio.set_volume(20)
pAudio.load("/home/xilinx/jupyter_notebooks/base/audio/recording_0.wav")

pAudio.play()
```

PYNQ Libraries

AxiGPIO

- The AxiGPIO class provides methods to read, write, and receive interrupts from external general purpose peripherals such as LEDs, buttons, switches

Initialization

```
from pynq import Overlay
from pynq.lib import AxiGPIO
ol = Overlay("base.bit")

led_ip = ol.ip_dict['leds_gpio']
switches_ip = ol.ip_dict['switches_gpio']
leds = AxiGPIO(led_ip).channel1
switches = AxiGPIO(switches_ip).channel1
```

Simple read and writes

```
mask = 0xffffffff
leds.write(0xf, mask)
switches.read()
```

PYNQ Libraries

Video

- The Video subpackage contains a collection of drivers for reading from the HDMI-In port, writing to the HDMI-Out port, transferring data, setting interrupts and manipulating video frames

Initialization

```
from pynq import Overlay
from pynq.lib.video import *
ol = Overlay("base.bit")

base = Overlay('base.bit')
hdmi_in = base.video.hdmi_in
hdmi_out = base.video.hdmi_out
```

PYNQ Libraries

Video

Configuration, execution and taking the unmodified input stream and passing it directly to the output

```
hdmi_in.configure()  
hdmi_out.configure(hdmi_in.mode)  
  
hdmi_in.start()  
hdmi_out.start()  
  
frame = hdmi_in.readframe()  
...  
hdmi_out.writeframe(frame)
```


- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - **Overlay Design Methodology**
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

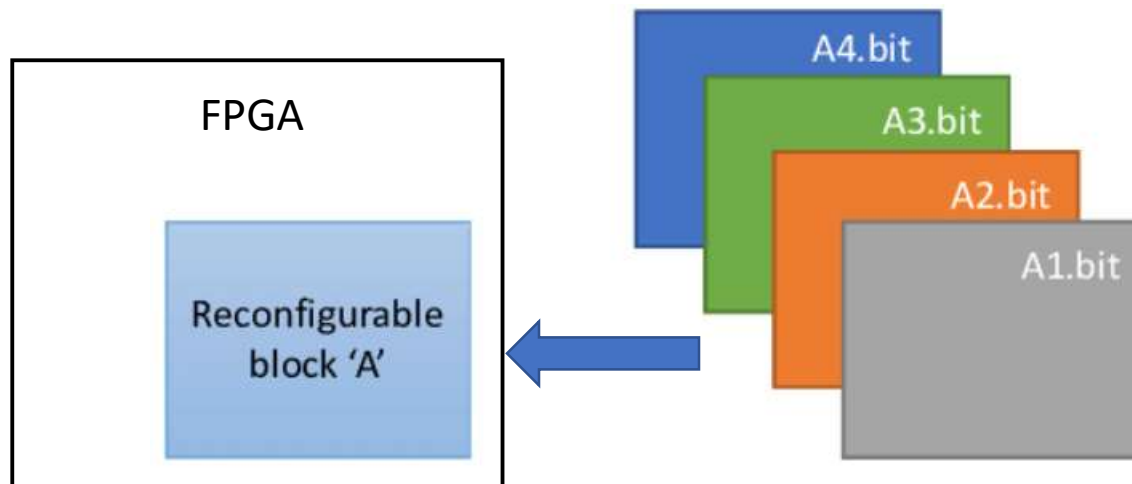
Overlay Design Methodology

- Overlay design is a specialized task for hardware engineers
 - The Xilinx Vivado software is used to create a Zynq design. A bitstream or binary file (.bit file) will be generated
 - The HWH (hardware handoff) file is automatically generated from the Vivado IP Integrator block design and it is used by PYNQ to automatically identify the Zynq system configuration, IP including versions, interrupts, resets, and other control signals

Overlay Design Methodology

Partial Reconfiguration

- The partial bitstreams are managed by the overlay class. It is always recommended to use the .hwh file along with the .bit for the overlay class
 - If the Vivado project is configured as a partial reconfiguration project, the .hwh file for the full bitstream will not contain any information inside a partial region
 - The .hwh file only provides the information on the interfaces connecting to the partial region
 - The complete information on the partial regions are revealed by the .hwh files of the partial bitstreams



In the function implemented in reconfigurable block 'A' is modified by switching between partial bitstreams, A1.bit, A2.bit, A3.bit and A4.bit

Overlay Design Methodology

Partial Reconfiguration

Loading Full Bitstream and download the full bitstream again

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

```
overlay.download() # To download the full bitstream again
```

Loading Partial Bitstream

```
overlay.block_0.download('rm_0_partial.bit') # The first way, using the download() method of the  
# DefaultHierarchy class
```

```
overlay.block_0.download('rm_1_partial.bit') # Load different reconfiguration module
```

```
overlay.pr_download('block_0', 'rm_0_partial.bit') # The second way, using pr_download() method of  
# the Overlay class
```

```
overlay.pr_download('block_0', 'rm_1_partial.bit') # Load different reconfiguration module
```

Overlay Design Methodology

Overlay Tutorial

- This tutorial is primarily designed to demonstrate two points, walking through the process of interacting with a new IP, developing a driver
 - Developing a Single IP
 - Creating a Driver

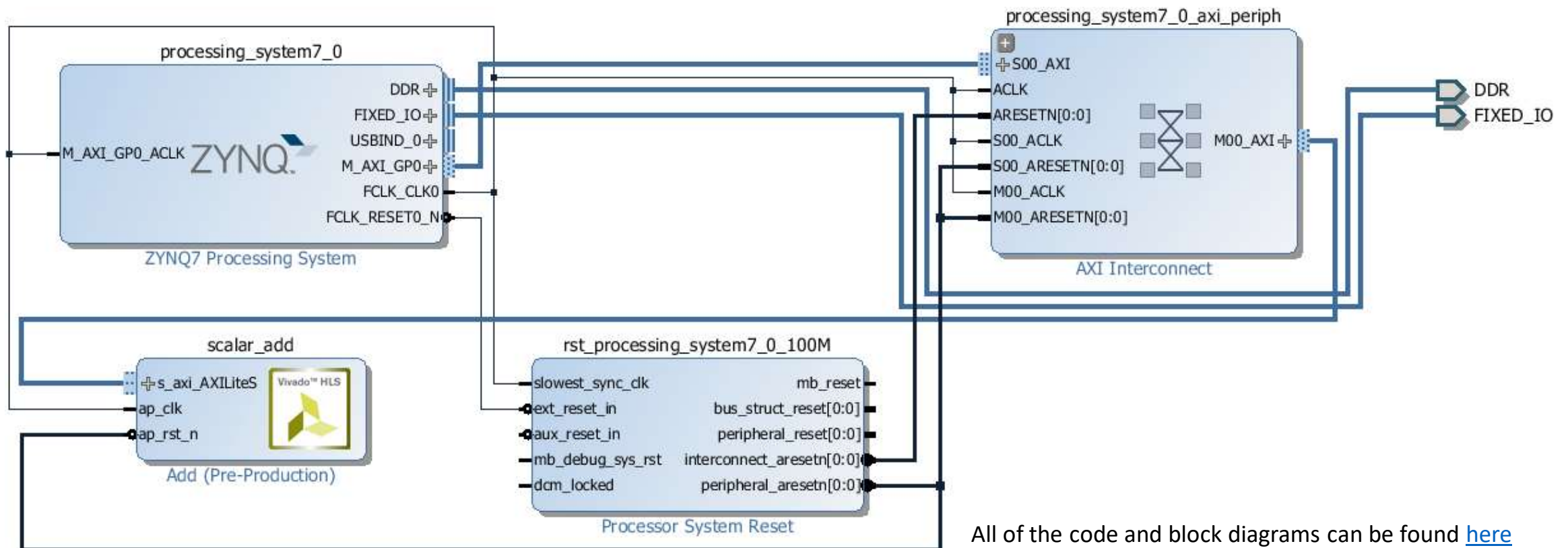
This IP was developed using HLS and adds two 32-bit integers together

```
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;  
}
```

Overlay Design Methodology

Overlay Tutorial - Developing a Single IP

- With a block diagram consisting solely of the HLS IP and required glue logic to connect it to the ZYNQ7 IP



All of the code and block diagrams can be found [here](#)

Overlay Design Methodology

Overlay Tutorial - Developing a Single IP

- Load the overlay containing the IP
- Accessing the `scalar_add` attribute will create a driver for the IP
- By providing the HWH file, we can also expose the register map associated with IP

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

```
add_ip = overlay.scalar_add
```

```
add_ip.register_map
```

```
RegisterMap {  
    a = Register(a=0),  
    b = Register(b=0),  
    c = Register(c=0),  
    c_ctrl = Register(c_ap_vld=1, RESERVED=0)  
}
```

Overlay Design Methodology

Overlay Tutorial - Developing a Single IP

- We can interact with the IP using the register map directly
- Alternatively by reading the driver source code generated by HLS we can determine that offsets we need to write the two arguments are at offsets **0x10** and **0x18** and the result can be read back from **0x20**.

```
add_ip.register_map.a = 3  
add_ip.register_map.b = 4  
add_ip.register_map.c
```

```
Register(c=7
```

```
)
```

```
add_ip.write(0x10, 4)  
add_ip.write(0x18, 5)  
add_ip.read(0x20)
```

```
9
```


Overlay Design Methodology

Overlay Tutorial - Creating a Driver

- We want to create an IP-specific driver exposing a single **add** function to call the accelerator. Custom drivers are created by inheriting from **DefaultIP** and adding a **bindto** class attribute consisting of the IP types the driver should bind to

```
from pynq import DefaultIP

class AddDriver(DefaultIP):
    def __init__(self, description):          # The description is a dictionary containing the address map
        super().__init__(description=description) # and any interrupts and GPIO pins connected to the IP

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

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

Overlay Design Methodology

Overlay Tutorial - Creating a Driver

- We reload the overlay again our new driver is bound to the IP
- Our custom driver with an **add** function is created

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

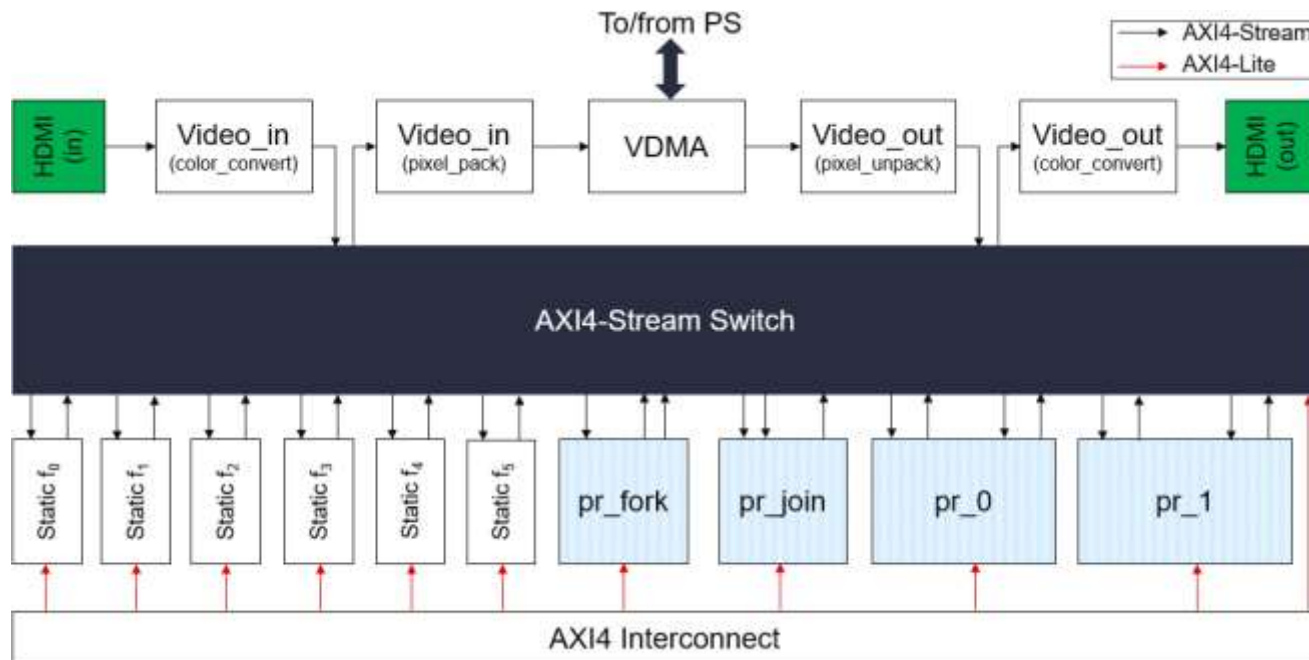
```
overlay.scalar_add.add(15,20)
```

35

- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- **PYNQ Composable Overlays**
 - **A Composable Video Pipeline**
 - Composable Overlay Methodology
 - Default Paths
- Tutorial
 - Composable Video Pipeline

A Composable Video Pipeline

- To demonstrate the benefits of the composable overlay, we are providing a composable video pipeline. An overview is shown below
 - The most common functions are implemented in the static region, these account for 6 functions. The composable overlay also provides 12 dynamic functions implemented across 4 DFX regions, note that pr_0 and pr_1 provide pairs of functions



Static IP	Dynamic IP	DFX Region		
colorthresholding	absdiff	pr_join	erode	pr_0 & pr_1
filter2d	add	pr_join	fast	pr_0
gray2rgb	bitwise_add	pr_join	fifo	pr_0 & pr_1
lut	cornerHarris	pr_1	filter2d	pr_0
rgb2gray	dilate	pr_0 & pr_1	rgb2xyz	pr_fork
rgb2hsv	duplicate	pr_fork	subtract	pr_join

- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- **PYNQ Composable Overlays**
 - A Composable Video Pipeline
 - **Composable Overlay Methodology**
 - Default Paths
- Tutorial
 - Composable Video Pipeline

Composable Overlay Methodology

- There are two key characteristics of a composable overlay
 - Uses at least one AXI4-Stream Switch – configured to use control register routing
 - Wraps the composable logic into a hierarchy
- AXI4-Stream Switch
 - The AXI4-Stream Switch provides configurable routing between managers and subordinates. It supports up to 16 managers and 16 subordinates
 - With the control register routing enabled, an AXI4-Lite interface is used to configure the routing table
 - The Python driver to manage the AXI4-Stream Switch is `pynq_composable.switch.StreamSwitch`

Composable Overlay Methodology

- Hierarchy
 - In Vivado IP Integrator, you can create a hierarchical block in a diagram to group a set of IP. Many IP in the Vivado catalog are implemented as hierarchical IP or also called subsystems
 - In the case of the composable overlays, all the logic associated to the composable portion must be inside of a hierarchy
- Dynamic Function eXchange (DFX)
 - DFX is the ability to dynamically modify blocks of logic by downloading partial bit files while the remaining logic continues to operate without interruption
 - Use DFX, you should include a DFX Decoupler IP for each partial Reconfigurable Partition (RP)
 - The [pynq_composable.composable.Composable](#) driver can control the DFX Decoupler IP via an AXI GPIO. This AXI GPIO must be included in the same hierarchy as the AXI4-Stream Switch

Composable Overlay Methodology

- Application Programming Interface (API)
 - StreamSwitch Driver
 - The `pynq_composable.switch.StreamSwitch` driver, this is the lowest level API and it requires intimate knowledge of the design and how the manager and subordinate interfaces are connected to the AXI4-Stream Switch
 - Composable Driver
 - The `pynq_composable.composable.Composable` driver provides an out-of-the-box experience with any composable overlay. The hardware will be automatically discovered, this process takes a few seconds the first time, and expose to the users
 - Pipeline App
 - Three basic methods for this API: `.start()`, `.play()` and `.stop()`. These methods, in combination with widgets or a dashboard should convey all the application functionality
 - One such example of this high level API is the `pynq_composable.apps.PipelineApp`, this is the parent class of all the applications supported by the Composable Video Pipeline.

- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- **PYNQ Composable Overlays**
 - A Composable Video Pipeline
 - Composable Overlay Methodology
 - **Default Paths**
- Tutorial
 - Composable Video Pipeline

Default Paths

- In a composable overlay, a default path specifies the nodes that are source and sink in a design
- Default paths are specified in a unique json file with the name **<overlay_name>_paths.json**. This file must be placed next to the overlay. The structure of this dictionary is as follows:
 - The first level key indicates the hierarchy
 - The second level key provides an arbitrary name for the path
 - The third level key defines the ci (Subordinate) and pi (Manager) interfaces on the AXI4-Stream Switch that the path is connected to

- PYNQ
 - PYNQ Introduction & Background
 - PYNQ Overlays
 - PYNQ Libraries
 - Overlay Design Methodology
- PYNQ Composable Overlays
 - A Composable Video Pipeline
 - pynq_composable Package
 - Composable Overlay Methodology
 - Default Paths
- **Tutorial**
 - **Composable Video Pipeline**

Composable Video Pipeline

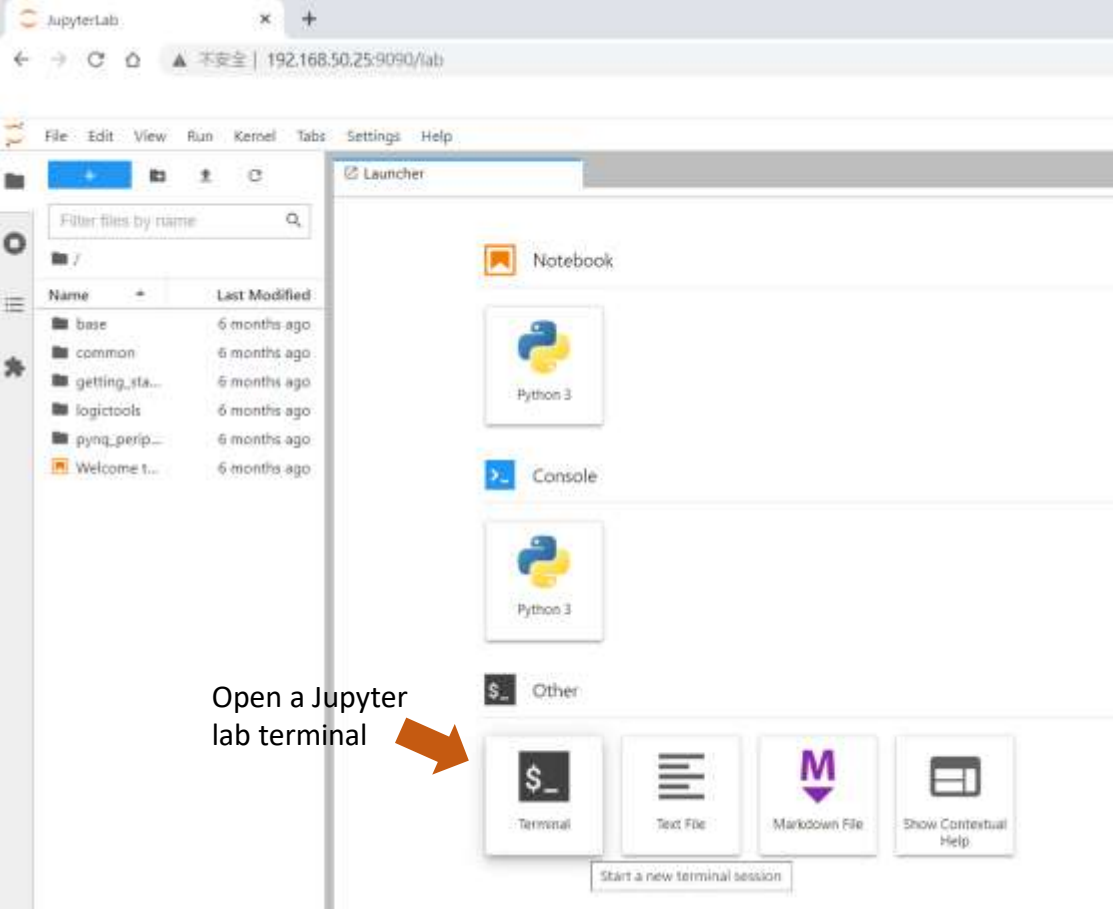
Installation on PYNQ-Z2 Board

- The PYNQ-Z2 board needs pynq 2.7 up
 - <https://github.com/Xilinx/PYNQ/releases>
- Check pynq version

```
root@pynq:/# pynq -v  
PYNQ version 2.7.0  
Git Id: 285d1457e64c076bbb39844afd54b38f075ad2c7
```

Composable Video Pipeline

Installation on PYNQ-Z2 Board



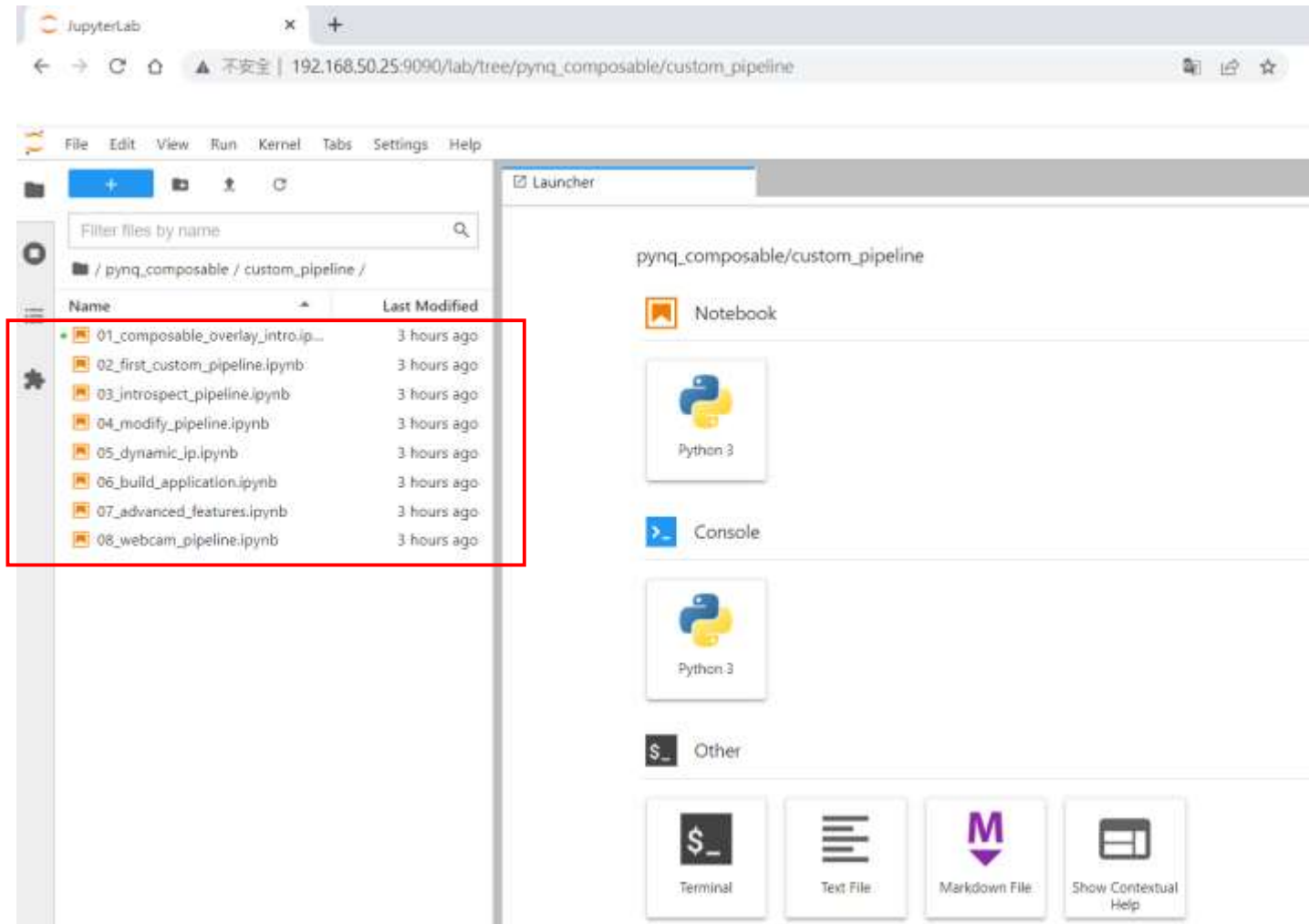
Open a Jupyter lab terminal

```
root@pynq:/# cd /home/xilinx/
root@pynq:/home/xilinx# git clone https://github.com/Xilinx/PYNQ_Composable_Pipeline
Cloning into 'PYNQ_Composable_Pipeline'...
remote: Enumerating objects: 1729, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (38/38), done.
remote: Total 1729 (delta 9), reused 14 (delta 9), pack-reused 1682
Receiving objects: 100% (1729/1729), 5.09 MiB | 1.09 MiB/s, done.
Resolving deltas: 100% (1073/1073), done.
root@pynq:/home/xilinx# python3 -m pip install PYNQ_Composable_Pipeline/ --no-build-isolation
Processing ./PYNQ_Composable_Pipeline
  Preparing wheel metadata ... done
Requirement already satisfied: pynq>=2.7.0 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq-composable==1.0.2) (2.7.0)
Requirement already satisfied: graphviz>=0.17 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq-composable==1.0.2) (0.20)
Requirement already satisfied: cffi in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq>=2.7.0->pynq-composable==1.0.2) (1.14.5)
Requirement already satisfied: setuptools>=24.2.0 in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq>=2.7.0->pynq-composable==1.0.2) (44.0.0)
Requirement already satisfied: pandas in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq>=2.7.0->pynq-composable==1.0.2) (1.3.3)
Requirement already satisfied: numpy in /usr/local/share/pynq-venv/lib/python3.8/site-packages (from pynq>=2.7.0->pynq-composable==1.0.2) (1.20.3)
Requirement already satisfied: pycparser in /usr/lib/python3/dist-packages (from cffi->pynq>=2.7.0->pynq-composable==1.0.2) (2.19)
Requirement already satisfied: pytz>=2017.3 in /usr/lib/python3/dist-packages (from pandas->pynq>=2.7.0->pynq-composable==1.0.2) (2019.3)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/lib/python3/dist-packages (from pandas->pynq>=2.7.0->pynq-composable==1.0.2) (2.7.3)
Building wheels for collected packages: pynq-composable
  Building wheel for pynq-composable (PEP 517) ... done
  Created wheel for pynq-composable: filename=pynq_composable-1.0.2-py3-none-any.whl size=4751290 sha256=987bed85836ea126a7e892a6c822eab76aaa75ca0e853b245131701097793775
  Stored in directory: /tmp/pip-ephem-wheel-cache-r2hvm7vm/wheels/9c/2d/75/8b14d19327ce7d9f1cd7a43a93a1c365eae70005b14757204a
Successfully built pynq-composable
Installing collected packages: pynq-composable
  Attempting uninstall: pynq-composable
    Found existing installation: pynq-composable 1.0.2
    Uninstalling pynq-composable-1.0.2:
      Successfully uninstalled pynq-composable-1.0.2
Successfully installed pynq-composable-1.0.2
root@pynq:/home/xilinx# pynq-get-notebooks pynq_composable -p $PYNQ_JUPYTER_NOTEBOOKS
Delivering notebooks '/home/xilinx/jupyter_notebooks/pynq_composable'...
root@pynq:/home/xilinx# ls jupyter_notebooks/pynq_composable/
01_get_started.ipynb 02_vision_intro.ipynb 03_download_video.ipynb applications custom_pipeline img
root@pynq:/home/xilinx#
```

Note: The composable video package packages are pre-installed, please skip executed commands in your lab

Composable Video Pipeline

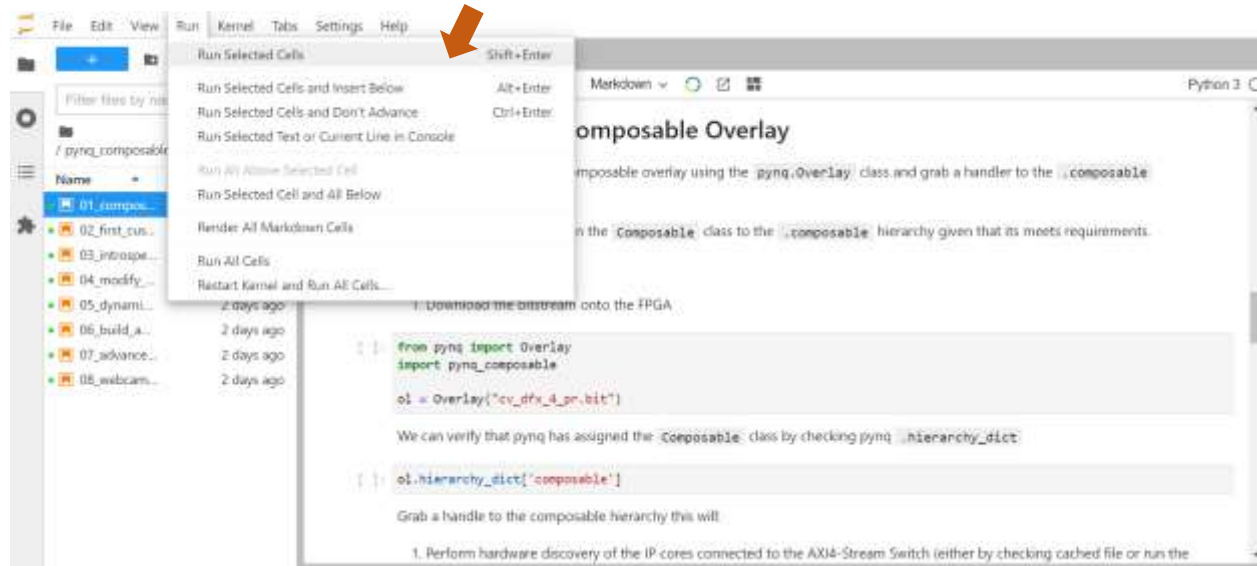
Custom Pipeline Exercises



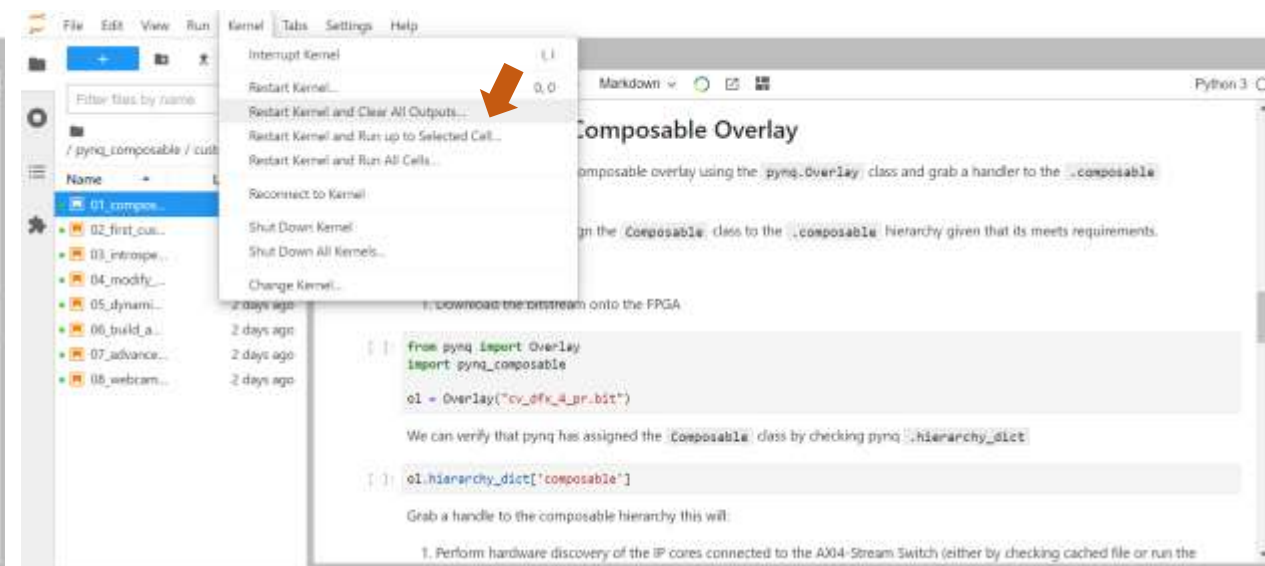
Composable Video Pipeline

Run Jupyter Notebooks

Click “Run Selected Cells” to execute Cell by Cell



Click “Restart Kernel and Clear All Outputs” to reset all Cells’ execution



Composable Video Pipeline

Exception of Run Jupyter Notebooks

Click “Run All Cells” results **Kernel died** in 02~06 and 08 tutorials, because of the unpredictable execution between PS and PL

The image is a composite of three screenshots from a Jupyter Notebook environment. The top-left screenshot shows the 'Run' menu with 'Run All Cells' highlighted by an orange arrow. The top-right screenshot shows the notebook content with a 'Download Composable Overlay' section. The bottom screenshot shows a 'Kernel Restarting' dialog box with the message 'The kernel for pynq_composable/custom_pipeline/02_first_custom_pipeline.ipynb appears to have died. It will restart automatically.' and an 'Ok' button. Below the dialog box, a red warning box states 'Warning: Failure to connect HDMI cable to a valid video source and screen may cause the notebook to hang'.

You need (1) power off -> power on
or (2) SSH to PYNQ and execute reboot

Composable Video Pipeline

01_composable_overlay_intro.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/01_composable_overlay_intro.ipynb

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

ol = Overlay("cv_dfx_4_pr.bit")
```

```
[2]: ol.hierarchy_dict['composable']
```

```
[3]: cpipe = ol.composable
```

All of the IP cores available to compose our pipeline

```
[4]: cpipe.c_dict
```

```
[4]: ▼ composable:
```

- ▶ rgb2gray_accel [loaded]:
- ▶ gray2rgb_accel [loaded]:
- ▶ rgb2hsv_accel [loaded]:
- ▶ filter2d_accel [loaded]:
- ▶ lut_accel [loaded]:
- ▶ colorthresholding_accel [loaded]:
- ▶ pr_0/fast_accel [unloaded]:
- ▶ pr_0/axis_data_fifo_0 [unloaded]:
- ▶ pr_0/erode_accel [unloaded]:
- ▶ pr_0/dilate_accel [unloaded]:
- ▶ pr_0/filter2d_accel [unloaded]:
- ▶ pr_0/axis_data_fifo_1 [unloaded]:

- ▶ pr_1/cornerHarris_accel [unloaded]:
- ▶ pr_1/axis_data_fifo_0 [unloaded]:
- ▶ pr_1/dilate_accel [unloaded]:
- ▶ pr_1/erode_accel [unloaded]:
- ▶ pr_join/bitwise_and_accel [unloaded]:
- ▶ pr_join/absdiff_accel [unloaded]:
- ▶ pr_join/subtract_accel [unloaded]:
- ▶ pr_join/add_accel [unloaded]:
- ▶ pr_fork/rgb2xyz_accel [unloaded]:
- ▶ pr_fork/duplicate_accel [unloaded]:
- ▶ hdmi_source_in [loaded][default]:
- ▶ hdmi_source_out [loaded][default]:
- ▶ hdmi_sink_in [loaded][default]:
- ▶ hdmi_sink_out [loaded][default]:

Composable Video Pipeline

01_composable_overlay_intro.ipynb

Filter by loaded, unloaded and default IP cores using the .loaded, .unloaded and .default attributes

```
[5]: cpipe.c_dict.loaded
```

```
[5]: ▼ composable:
    ▶ rgb2gray_accel [loaded]:
    ▶ gray2rgb_accel [loaded]:
    ▶ rgb2hsv_accel [loaded]:
    ▶ filter2d_accel [loaded]:
    ▶ lut_accel [loaded]:
    ▶ colorthresholding_accel [loaded]:
    ▶ hdmi_source_in [loaded][default]:
    ▶ hdmi_source_out [loaded][default]:
    ▶ hdmi_sink_in [loaded][default]:
    ▶ hdmi_sink_out [loaded][default]:
```

```
[6]: cpipe.c_dict.unloaded
```

```
[6]: ▼ composable:
    ▶ pr_0/fast_accel [unloaded]:
    ▶ pr_0/axis_data_fifo_0 [unloaded]:
    ▶ pr_0/erode_accel [unloaded]:
    ▶ pr_0/dilate_accel [unloaded]:
    ▶ pr_0/filter2d_accel [unloaded]:
    ▶ pr_0/axis_data_fifo_1 [unloaded]:
    ▶ pr_1/cornerHarris_accel [unloaded]:
    ▶ pr_1/axis_data_fifo_0 [unloaded]:
    ▶ pr_1/dilate_accel [unloaded]:
    ▶ pr_1/erode_accel [unloaded]:
    ▶ pr_join/bitwise_and_accel [unloaded]:
    ▶ pr_join/absdiff_accel [unloaded]:
    ▶ pr_join/subtract_accel [unloaded]:
    ▶ pr_join/add_accel [unloaded]:
    ▶ pr_fork/rgb2xyz_accel [unloaded]:
    ▶ pr_fork/duplicate_accel [unloaded]:
```

```
[7]: cpipe.c_dict.default
```

```
[7]: ▼ composable:
    ▶ hdmi_source_in [loaded][default]:
    ▶ hdmi_source_out [loaded][default]:
    ▶ hdmi_sink_in [loaded][default]:
    ▶ hdmi_sink_out [loaded][default]:
```

Composable Video Pipeline

01_composable_overlay_intro.ipynb

- If you expand an entry on this dictionary, you will see
 - decoupler: name of the decoupler that handles the DFX regions
 - gpio: PS GPIO pins that enable decouple and status of said decoupler
 - rm: (reconfigurable module) dictionary of partial bitstreams and IP cores contained in them

All of the DFX regions available in the composable overlay

```
[8]: cpipe.dfx_dict
```

```
[8]: ▼ dfx_dict:
      ▼ pr_0:
        decoupler: "/composable/dfx_decouplers/dfx_decoupler_pr_0"
        decouple: 0
        status: 4
      ▼ rm:
        ► cv_dfx_4_pr_composable_pr_0_fast_fifo_partial.bit:
        ► cv_dfx_4_pr_composable_pr_0_dilate_erode_partial.bit:
        ► cv_dfx_4_pr_composable_pr_0_filter2d_fifo_partial.bit:
      ▼ pr_1:
        decoupler: "/composable/dfx_decouplers/dfx_decoupler_pr_1"
        decouple: 1
        status: 5
        ► rm:
      ▼ pr_join:
        decoupler: "/composable/dfx_decouplers/dfx_decoupler_pr_join"
        decouple: 2
        status: 6
        ► rm:
      ▼ pr_fork:
        decoupler: "/composable/dfx_decouplers/dfx_decoupler_pr_fork"
        decouple: 3
        status: 7
        ► rm:
```

Composable Video Pipeline

02_first_custom_pipeline.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/02_first_custom_pipeline.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

      ol = Overlay("cv_dfx_4_pr.bit")

      cpipe = ol.composable
```

Start HDMI Video

Get `VideoStream` object and start video

Warning:

Failure to connect HDMI cables to a valid video source and screen may cause the notebook to hang

```
[2]: video = VideoStream(ol)
      video.start()
```



Composable Video Pipeline

02_first_custom_pipeline.ipynb

Let us Compose

Grab a handler to the LUT IP object

```
[3]: lut = cpipe.lut_accel
```

Let us read the documentation on the method `.compose`

```
[4]: cpipe.compose?
```

Signature: `cpipe.compose(cle_list: list) -> None`

Docstring:

Configure design to implement required dataflow pipeline

Parameters

`cle_list` : list

list of the composable IP objects

Examples:

`[a, b, c, d]` yields

.. code-block:: none

`-> a -> b -> c -> d ->`

`[a, b, [[c,d],[e]], f, g]` yields

.. code-block:: none

```
      -> c -> d -
     /
-> a -> b      \ f -> g ->
     \          /
      ---> e ----
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

This method expect a list with the IP object, based on this list the pipeline will be configured on our FPGA. After you run the next cell the video stream on your monitor should change,

Composable Video Pipeline

02_first_custom_pipeline.ipynb

```
[5]: video_pipeline = [cpipe.hdm_source_in, lut, cpipe.hdm_source_out]  
    cpipe.compose(video_pipeline)
```



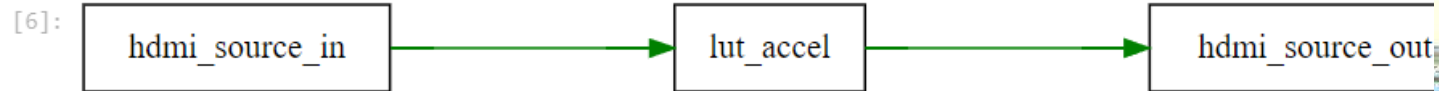
Composable Video Pipeline

02_first_custom_pipeline.ipynb

Visualize the Pipeline

We can visualize the implemented pipeline with the `.graph` attribute. This allows to quickly verify the pipeline

```
[6]: cpipe.graph
```



Play with the LUT IP

The LUT is one of the IP available on the static region of the composable overlay, this IP allows further runtime configuration with predefined kernels

The next cell will change the kernel type of the LUT IP every second, you will be able to watch the change on the output video

```
[7]: import time
for i in xvLut:
    lut.kernel_type = i
    time.sleep(0.6)
```



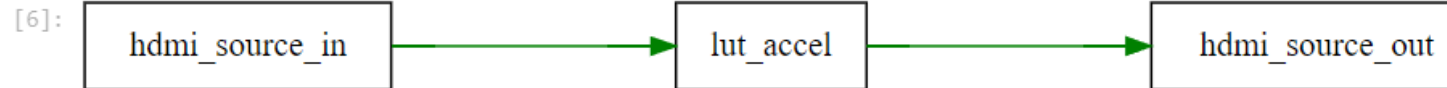
Composable Video Pipeline

02_first_custom_pipeline.ipynb

Visualize the Pipeline

We can visualize the implemented pipeline with the `.graph` attribute. This allows to quickly verify the pipeline

```
[6]: cpipe.graph
```

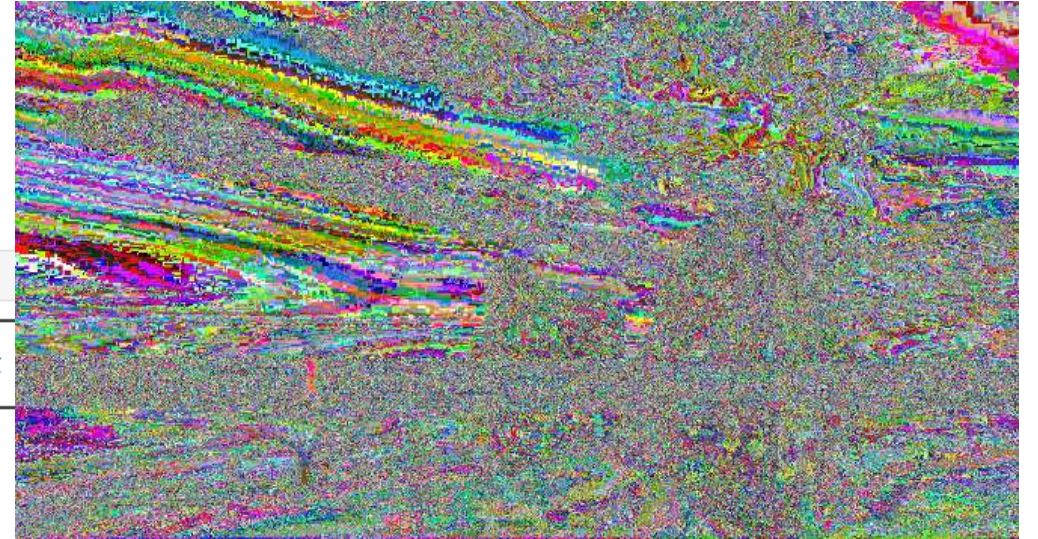


Play with the LUT IP

The LUT is one of the IP available on the static region of the composable overlay, this IP allows further runtime configuration with predefined kernels

The next cell will change the kernel type of the LUT IP every second, you will be able to watch the change on the output video

```
[7]: import time
for i in xvlut:
    lut.kernel_type = i
    time.sleep(0.6)
```



Composable Video Pipeline

03_introspect_pipeline.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/03_introspect_pipeline.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

      ol = Overlay("cv_dfx_4_pr.bit")

      cpipe = ol.composable
```

Start HDMI Video

Get `VideoStream` object and start video

Warning:

Failure to connect HDMI cables to a valid video source and screen may cause the notebook to hang

```
[2]: video = VideoStream(ol)
      video.start()
```



Composable Video Pipeline

03_introspect_pipeline.ipynb

Let us Compose

First we need to grab handlers to the IP objects to simplify the notebook

```
[3]: filter2d = cpipe.filter2d_accel  
     rgb2gray = cpipe.rgb2gray_accel  
     gray2rgb = cpipe.gray2rgb_accel  
     rgb2hsv = cpipe.rgb2hsv_accel  
     colorthr = cpipe.colorthresholding_accel  
     lut = cpipe.lut_accel
```

This method expect a list with the IP object, based on this list the pipeline will be configured on our FPGA. After you run the next cell the video stream on your monitor should change,

```
[4]: video_pipeline = [cpipe.hdmi_source_in, lut, rgb2hsv, rgb2gray, gray2rgb, cpipe.hdmi_source_out]  
     cpipe.compose(video_pipeline)
```



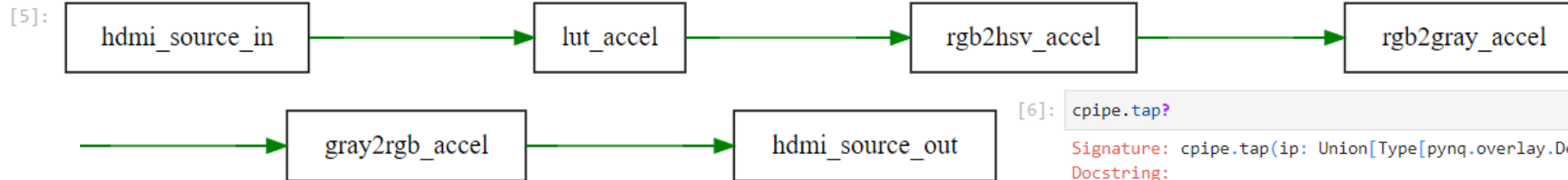
Composable Video Pipeline

03_introspect_pipeline.ipynb

Visualize the Pipeline

We can visualize the implemented pipeline with the `.graph` attribute. This allows to quickly verify the pipeline

```
[5]: cpipe.graph
```



```
[6]: cpipe.tap?
```

Signature: `cpipe.tap(ip: Union[Type[pynq.overlay.DefaultIP], int] = None) -> None`

Docstring:

Observe the output of an IP object in the current pipeline

Tap into the output of any of the IP cores in the current pipeline

Note that tap is not supported in a branch

You can tap by passing the IP name or the index of the IP in the list.

Note that tap does not modify the attribute `current_pipeline`

Parameters

ip:

Either an IP object in the current pipeline to be tapped or
index of IP object in the current pipeline to be tapped

Examples:

```
tap(cpipe.pr_1.dilate)
```

```
tap(6)
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

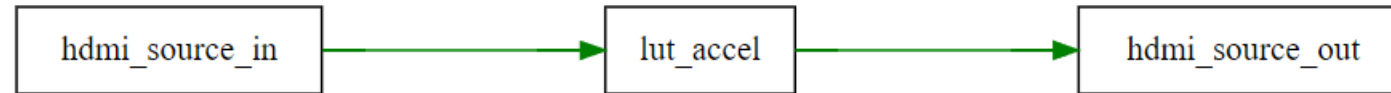
Composable Video Pipeline

03_introspect_pipeline.ipynb

```
[7]: cpipe.tap(lut)
```

```
cpipe.graph
```

```
[7]:
```



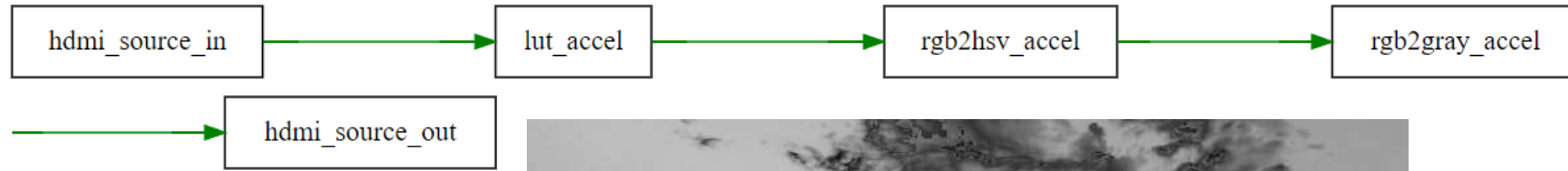
Composable Video Pipeline

03_introspect_pipeline.ipynb

```
[8]: cpipe.tap(3)
```

```
cpipe.graph
```

```
[8]:
```



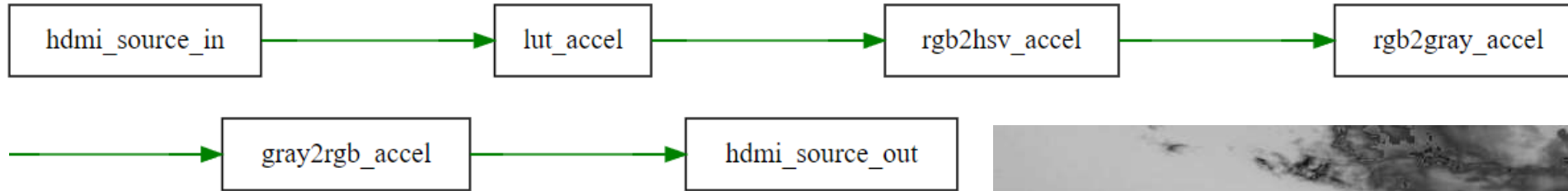
Composable Video Pipeline

03_introspect_pipeline.ipynb

```
[9]: cpipe.untap()
```

```
cpipe.graph
```

```
[9]:
```



Composable Video Pipeline

04_modify_pipeline.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/04_modify_pipeline.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

      ol = Overlay("cv_dfx_4_pr.bit")

      cpipe = ol.composable
```

Start HDMI Video

Get `VideoStream` object and start video

Warning:

Failure to connect HDMI cables to a valid video source and screen may cause the notebook to hang

```
[2]: video = VideoStream(ol)
      video.start()
```



Composable Video Pipeline

04_modify_pipeline.ipynb

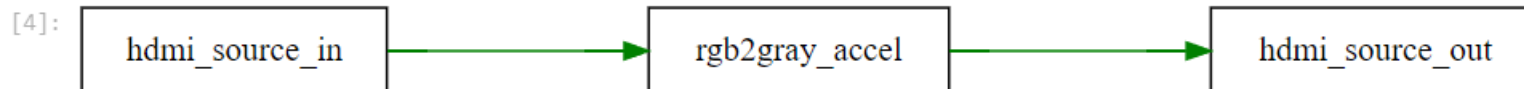
Let us Compose

First we need to grab handlers to the IP objects to simplify the notebook

```
[3]: filter2d = cpipe.filter2d_accel  
     rgb2gray = cpipe.rgb2gray_accel  
     gray2rgb = cpipe.gray2rgb_accel  
     rgb2hsv = cpipe.rgb2hsv_accel  
     colorthr = cpipe.colorthresholding_accel  
     lut = cpipe.lut_accel
```

We will start with a simple pipeline that converts from RGB color space to Grayscale color space

```
[4]: video_pipeline = [cpipe.hdmi_source_in, rgb2gray, cpipe.hdmi_source_out]  
     cpipe.compose(video_pipeline)  
     cpipe.graph
```



Composable Video Pipeline

04_modify_pipeline.ipynb

Replace IP object

We can replace the `rgb2gray` IP object with the `rgb2hsv` easily using the `.replace` method. This method takes a tuple with the IP object to be replaced and the new IP object.

```
[5]: cpipe.replace?
```

Signature: `cpipe.replace(replaceip: tuple) -> None`

Docstring:

Replace an IP object in the current pipeline

Parameters

`replaceip: tuple`

Tuple of two items.

First: IP object to be replaced

Second: new IP object

Examples:

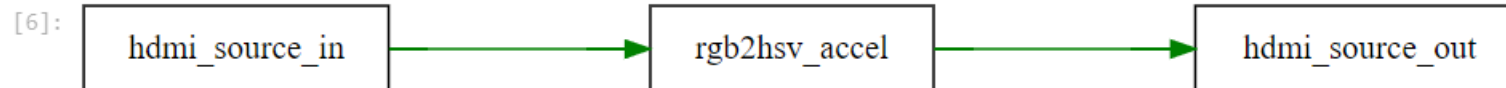
```
(cpipe.pr_0.erode, cpipe.pr_1.dilate)
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

```
[6]: cpipe.replace((rgb2gray, rgb2hsv))
```

```
cpipe.graph
```



Composable Video Pipeline

04_modify_pipeline.ipynb

Remove IP object

To visualize the RGB color space we can simply remove the `rgb2hsv` IP object from the composable pipeline using the `.remove` method. This method gets a list of IP object to be removed as argument

```
[7]: cpipe.remove?
```

Signature: `cpipe.remove(iplist: list = None) -> None`

Docstring:

Remove IP object from the current pipeline

Parameters

`iplist: list`

List of IP to be removed from the current pipeline

Examples:

```
[cpipe.pr_0.erode]
```

```
[cpipe.pr_1.filter2d, cpipe.pr_fork.duplicate]
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

```
[8]: cpipe.remove([rgb2hsv])
```

```
cpipe.graph
```

```
[8]:
```

```
hdmi_source_in
```



```
hdmi_source_out
```



Composable Video Pipeline

04_modify_pipeline.ipynb

Insert IP objects

The `.insert` method allows you to insert an IP object or list of IP object into a given index within the current pipeline

```
[9]: cpipe.insert?
```

Signature: `cpipe.insert(iptuple: tuple) -> None`

Docstring:

Insert a new IP or list of IP into current pipeline

Parameters

iptuple: tuple

 Tuple of two items.

 First: list of IP to be inserted

 Second: index

Examples:

```
    ([cpipe.pr_0.erode], 3)
```

```
    ([cpipe.pr_1.filter2d, cpipe.pr_fork.duplicate], 2)
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

```
[10]: cpipe.insert([filter2d, lut], 1)
```

```
cpipe.graph
```

```
[10]:
```



Composable Video Pipeline

04_modify_pipeline.ipynb

```
[11]: filter2d.kernel_type = xvF2d.sharpen
```

Insert the gray2rgb IP after the LUT IP

```
[12]: cpipe.insert([gray2rgb], 3))
```

```
cpipe.graph
```



Composable Video Pipeline

05_dynamic_ip.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/05_dynamic_ip.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

      ol = Overlay("cv_dfx_4_pr.bit")

      cpipe = ol.composable
```

Start HDMI Video

Get `VideoStream` object and start video

Warning:

Failure to connect HDMI cables to a valid video source and screen may cause the notebook to hang

```
[2]: video = VideoStream(ol)
      video.start()
```



Composable Video Pipeline

05_dynamic_ip.ipynb

Load Dynamic IP

The Composable Overlay provides DFX regions where IP can be loaded dynamically to bring new functionality. If we want to load an IP within a DFX region, the `.loadIP` method is used.

Let us start by looking at the `.c_dict` to see what IP cores are loaded

```
[3]: cpipe.c_dict.loaded
```

```
[3]: ▼ composable:
      ▶ rgb2gray_accel [loaded]:
      ▶ gray2rgb_accel [loaded]:
      ▶ rgb2hsv_accel [loaded]:
      ▶ filter2d_accel [loaded]:
      ▶ lut_accel [loaded]:
      ▶ colorthresholding_accel [loaded]:
      ▶ hdmi_source_in [loaded][default]:
      ▶ hdmi_source_out [loaded][default]:
      ▶ hdmi_sink_in [loaded][default]:
      ▶ hdmi_sink_out [loaded][default]:
```

The documentation of `.loadIP` specify that IP can be loaded using the full name or the IP object

```
[4]: cpipe.loadIP?
```

Signature: `cpipe.loadIP(dfx_list: list) -> None`

Docstring:

Download dfx IP onto the corresponding partial regions

Parameters

`dfx_list: list`

List of IP to be downloaded onto the dfx regions. The list can contain either a string with the fullname or the IP object

Examples:

```
[cpipe.pr_0.fast_accel, cpipe.pr_1.dilate_accel]
['pr_0/fast_accel', 'pr_1/dilate_accel']
```

File: `/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py`

Type: `method`

Composable Video Pipeline

05_dynamic_ip.ipynb

```
[5]: cpipe.loadIP([cpipe.pr_1.dilate_accel])
```

Examine the `.c_dict` again and verify that `dilate_accel` and `erode_accel` are indeed loaded, both are in the same DFX region

```
[6]: cpipe.c_dict.loaded
```

```
[6]: ▼ composable:  
    ▶ rgb2gray_accel [loaded]:  
    ▶ gray2rgb_accel [loaded]:  
    ▶ rgb2hsv_accel [loaded]:  
    ▶ filter2d_accel [loaded]:  
    ▶ lut_accel [loaded]:  
    ▶ colorthresholding_accel [loaded]:  
    ▶ pr_1/dilate_accel [loaded]:  
    ▶ pr_1/erode_accel [loaded]:  
    ▶ hdmi_source_in [loaded][default]:  
    ▶ hdmi_source_out [loaded][default]:  
    ▶ hdmi_sink_in [loaded][default]:  
    ▶ hdmi_sink_out [loaded][default]:
```

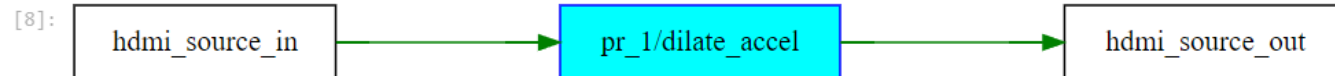
Let us Compose

Grab handlers to the dilate IP and compose

```
[7]: dilate = cpipe.pr_1.dilate_accel
```

```
[8]: cpipe.compose([cpipe.hdmi_source_in, dilate, cpipe.hdmi_source_out])
```

```
cpipe.graph
```



Composable Video Pipeline

05 dynamic ip.ipynb

Branched Pipeline

In this part of the notebook, we will bring new functionality into the four DFX regions to compose the [Difference of Gaussians](#) application that was also introduced in the previous session.

Load dynamic IP, grab handlers and set up default values

```
[9]: cpipe.loadIP(['pr_fork/duplicate_accel', 'pr_join/subtract_accel', 'pr_0/filter2d_accel'])
```

Grab handlers and configure filter2D with the gaussian filter

```
[10]: filter2d = cpipe.filter2d_accel
duplicate = cpipe.pr_fork.duplicate_accel
subtract = cpipe.pr_join.subtract_accel
fifo = cpipe.pr_0.axis_data_fifo_1
filter2d_d = cpipe.pr_0.filter2d_accel

filter2d.sigma = 0.3
filter2d.kernel_type = xvF2d.gaussian_blur

filter2d_d.sigma = 12
filter2d_d.kernel_type = xvF2d.gaussian_blur
```

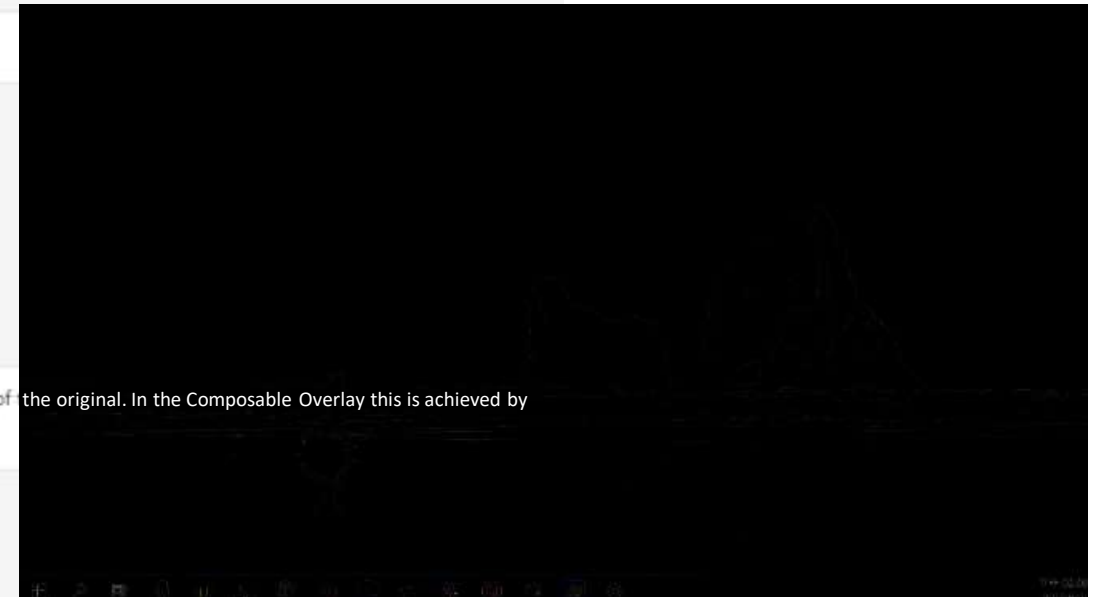
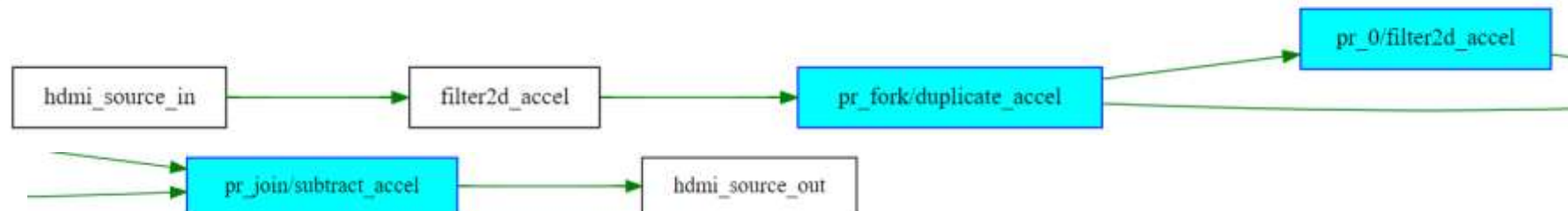
The Difference of Gaussians is realized by subtracting one Gaussian blurred version of an original image from another less blurred version of the original. In the Composable Overlay this is achieved by branching the pipeline, which is expressed as a list of a list.

```
[11]: video_pipeline = [cpipe.hdmi_source_in, filter2d, duplicate, [[filter2d_d], [1]], subtract, cpipe.hdmi_source_out]
```

```
cpipe.compose(video_pipeline)
```

```
cpipe.graph
```

```
[11]:
```



Composable Video Pipeline

05_dynamic_ip.ipynb

Conflicting Dynamic IP

Note that IP within the DFX regions are often mutually exclusive (some partial bitstreams support multiple IP within the DFX region), this means that they cannot be loaded at the same time. The `.loadIP` will raise an exception in these cases, try it by yourself running the following cell

```
[12]: cpipe.loadIP(['pr_fork/duplicate_accel', 'pr_fork/rgb2xyz_accel'])
```

```
-----
SystemError                                Traceback (most recent call last)
<ipython-input-12-4a38f0c32419> in <module>
----> 1 cpipe.loadIP(['pr_fork/duplicate_accel', 'pr_fork/rgb2xyz_accel'])

/usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/composable.py in loadIP(self, dfx_list)
   538         self._c_dict[fullpath]['loaded']
   539         elif bit_dict[pr]['bitstream'] != bitname:
--> 540             raise SystemError("\{'}\' and \{'}\' bitstreams cannot"
   541                               " be loaded into the same DFX "
   542                               " region \{'}\' at the same time"

SystemError: 'cv_dfx_4_pr_composable_pr_fork_duplicate_partial.bit' and 'cv_dfx_4_pr_composable_pr_fork_rgb2xyz_partial.bit' bitstreams cannot be loaded into the same DFX region 'pr_fork' at the same time
```

Info! Use the `dfx_dict` attribute to identify which IP are mutually exclusive

Composable Video Pipeline

06_build_application.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/06_build_application.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

      ol = Overlay("cv_dfx_4_pr.bit")

      cpipe = ol.composable
```

Start HDMI Video

Get `VideoStream` object and start video

Warning:

Failure to connect HDMI cables to a valid video source and screen may cause the notebook to hang

```
[2]: video = VideoStream(ol)
      video.start()
```



Composable Video Pipeline

06_build_application.ipynb

Compose Pipeline

First we need to load the partial bitstreams to bring `fast` and `cornerHarris` functionality

```
[3]: cpipe.loadIP(['pr_0/fast_accel', 'pr_1/cornerHarris_accel'])
```

Grab handlers, compose the pipeline and visualize it

```
[4]: rgb2gray = cpipe.rgb2gray_accel  
gray2rgb = cpipe.gray2rgb_accel  
fast = cpipe.pr_0.fast_accel  
harr = cpipe.pr_1.cornerHarris_accel  
  
cpipe.compose([cpipe.hdmi_source_in, rgb2gray, fast, gray2rgb, cpipe.hdmi_source_out])
```

```
cpipe.graph
```



Composable Video Pipeline

06_build_application.ipynb

Build the Application

In the following cells we will define some useful functions to help us change the functionality of the application

Declare the `threshold` and `K` values as `IntSlider` and `FloatSlider` respectively

```
[5]: from ipywidgets import widgets, IntSlider, FloatSlider, interact

thr = IntSlider(min=0, max=255, step=1, value=20)
k_harris = FloatSlider(min=0, max=0.2, step=0.002, value=0.04, description='K')
```

Declare `swap` function that enables to change between `fast` and `cornerHarris`

```
[6]: algorithm = 'Fast'
def swap():
    global algorithm
    global thr
    if algorithm == 'Fast':
        cpipe.replace((fast, harr))
        algorithm = 'Harris'
        thr.max = 1024
        thr.value = 422
    else:
        cpipe.replace((harr, fast))
        algorithm = 'Fast'
        thr.max = 255
        thr.value = 20
```

```
[7]: def app(new_algorithm, threshold, k):
    global thr
    global k_harris
    if new_algorithm != algorithm:
        swap()
    elif new_algorithm == 'Fast':
        fast.threshold = threshold
        k_harris.disabled = True
    else:
        harr.threshold = threshold
        k_harris.disabled = False
        harr.k = k
```

Composable Video Pipeline

06_build_application.ipynb

Run the Application

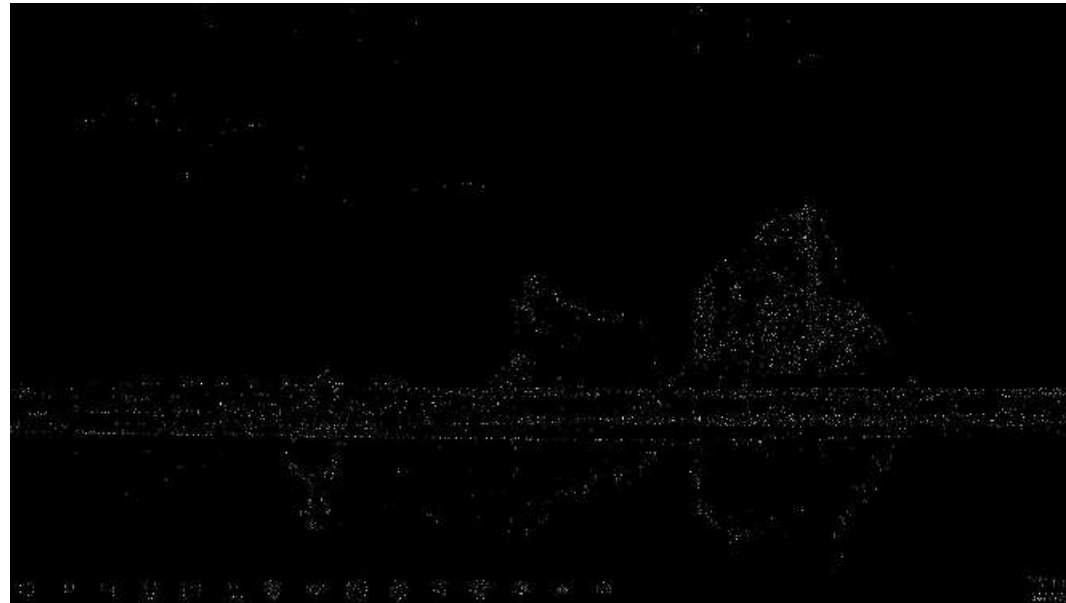
Finally we can use `interact`, which automatically creates user interface (UI) controls, to run our application. The first argument is the function we want to call and the following are the argument to such function.

```
[8]: interact(app, new_algorithm=['Fast','Harris'], threshold=thr, k=k_harris);
```

new_algorit... Fast ▼

threshold 20

K 0.04



Composable Video Pipeline

06_build_application.ipynb

Run the Application

Finally we can use `interact`, which automatically creates user interface (UI) controls, to run our application. The first argument is the function we want to call and the following are the argument to such function.

```
[8]: interact(app, new_algorithm=['Fast','Harris'], threshold=thr, k=k_harris);
```

new_algorit... Harris ▼

threshold 422

K 0.04



Composable Video Pipeline

07_advanced_features.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/07_advanced_features.ipynb

This attribute will enable debug mode in the graph

```
[1]: from pynq import Overlay
      from pynq.lib.video import *
      from pynq_composable import *

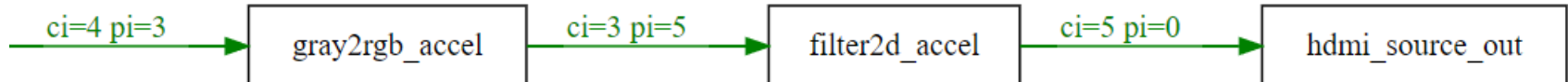
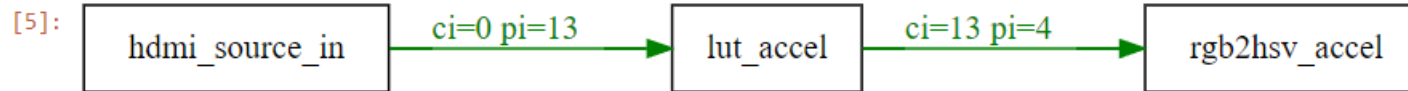
      ol = Overlay("cv_dfx_4_pr.bit")
      cpipe = ol.composable
```

```
[2]: cpipe._graph_debug = True
```

```
[3]: filter2d = cpipe.filter2d_accel
      rgb2hsv = cpipe.rgb2hsv_accel
      gray2rgb = cpipe.gray2rgb_accel
      lut = cpipe.lut_accel
```

```
[4]: video_pipeline = [cpipe.hdmi_source_in, lut, rgb2hsv, gray2rgb, filter2d, cpipe.hdmi_source_out]
      cpipe.compose(video_pipeline)
```

```
[5]: cpipe.graph
```

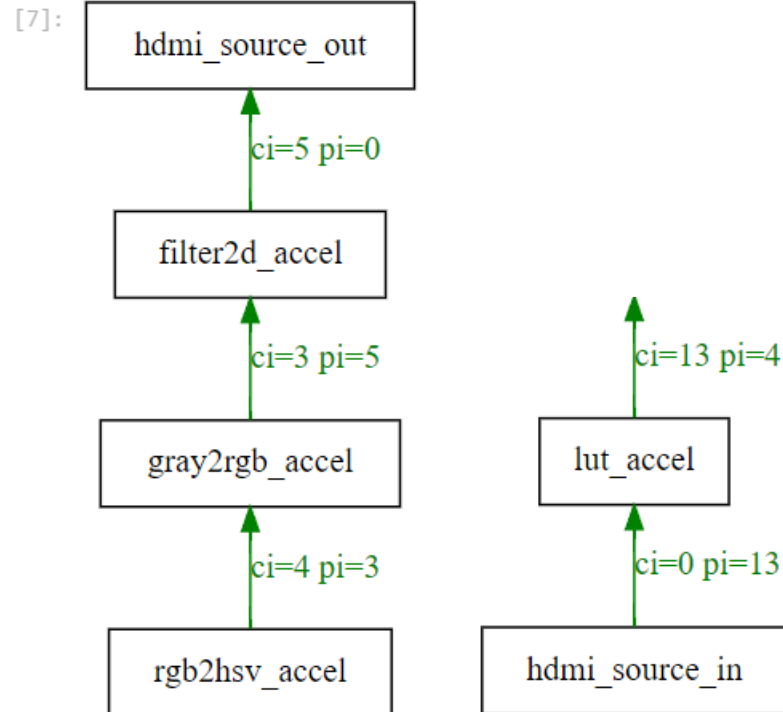


Composable Video Pipeline

07_advanced_features.ipynb

```
[6]: cpipe.graph.graph_attr['rankdir'] = 'BT'
```

```
[7]: cpipe.graph
```



```
[8]: cpipe.axis_switch?
```

Type: StreamSwitch
String form: <pynq_composable.switch.StreamSwitch object at 0xb5070f58>
File: /usr/local/share/pynq-venv/lib/python3.8/site-packages/pynq_composable/switch.py
Docstring:
AXI4-Stream Switch python driver

This class provides the driver to control an AXI4-Stream Switch which uses the AXI4-Lite interfaces to specify the routing table. This routing mode requires that there is precisely only one path between producer and consumer. When attempting to map the same consumer interface to multiple producer interfaces, only the lowest consumer interface is able to access the consumer interface. Unused producer interfaces are automatically disabled by the logic provided in this driver

Composable Video Pipeline

07_advanced_features.ipynb

```
[9]: cpipe.axis_switch.pi?
```

```
Type:      property
String form: <property object at 0xaf2ea870>
Docstring:
AXI4-Stream Switch configuration
```

Configure the AXI4-Stream Switch given a numpy array
Each element in the array controls a consumer interface selection.
If more than one element in the array is set to the same consumer interface, then the lower producer interface wins.

Parameters

conf_array : numpy array (dtype=np.int64)

An array with the mapping of consumer to producer interfaces
The index in the array is the producer interface and
the value is the consumer interface slot
The length of the array can vary from 1 to max slots
Use negative values to indicate that a producer is disabled

For instance, given this input [-1, 2, 1, 0]

Consumer 2 will be routed to Producer 1

Consumer 1 will be routed to Producer 2

Consumer 0 will be routed to Producer 3

Producer 0 is disabled

You can check the current AXI4-Stream Switch configuration using the `.pi` attribute.

Note, the index in the array is the producer interface number and the value is the consumer interface number

```
[10]: cpipe.axis_switch.pi
```

```
[10]: array([      5,      1, 2147483648,      4,      13,
           3, 2147483648, 2147483648, 2147483648, 2147483648,
           2147483648, 2147483648, 2147483648,      0, 2147483648],
        dtype=int64)
```

Composable Video Pipeline

08_webcam_pipeline.ipynb

- http://PYNQ_IP:Port/lab/tree/pynq_composable/custom_pipeline/08_webcam_pipeline.ipynb

Download Composable Overlay

Import the pynq video libraries as well as Composable class and the drivers for the IP.

Download the Composable Overlay using `pynq.Overlay` and grab a handler to the `composable` hierarchy

```
[1]: from pynq import Overlay
from pynq.lib.video import *
from pynq_composable import *
from ipywidgets import widgets, interact, FloatSlider, IntSlider
from pynq.ps import CPU_ARCH, ZYNQ_ARCH

ol = Overlay("cv_dfx_4_pr.bit")

cpipe = ol.composable
```

Configure Webcam

Configure the Webcam and with `VideoStream` class, and start the video

Warning:

Failure to connect HDMI output cable to an screen may cause the notebook to hang

```
[2]: video = VideoStream(ol, source=VSource.OpenCV)

video.start()
```



Composable Video Pipeline

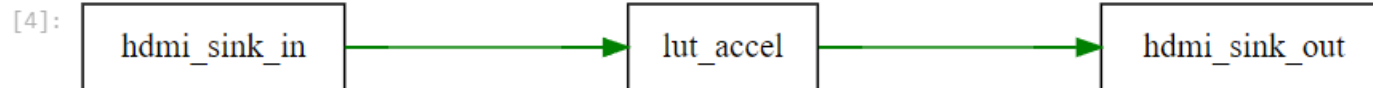
08_webcam_pipeline.ipynb

Compose Simple Pipeline

Grab handlers to LUT and compose

```
[3]: lut = cpipe.lut_accel  
lut.kernel_type = xvLut.negative
```

```
[4]: cpipe.compose([cpipe.hdmi_sink_in, lut, cpipe.hdmi_sink_out])  
cpipe.graph
```



Composable Video Pipeline

08_webcam_pipeline.ipynb

Compose Complex Pipeline

Warning:

Failure to pause the VideoStream for Zynq-7000 devices before using `.loadIP` may cause the notebook to hang

In this part of the notebook, we will bring new functionality into the DFX regions to compose a corner detect application.

Load dynamic IP, grab handlers and set up default values

```
[5]: if CPU_ARCH != ZYNQ_ARCH:
      video.pause()

      cpipe.loadIP(['pr_0/fast_accel', 'pr_fork/duplicate_accel', 'pr_join/add_accel'])
```

```
[6]: #Resume Webcam stream
      if CPU_ARCH != ZYNQ_ARCH:
          video.start()
```

Grab handler to functions

```
[7]: fast = cpipe.pr_0.fast_accel
      duplicate = cpipe.pr_fork.duplicate_accel
      add = cpipe.pr_join.add_accel
      r2g = cpipe.rgb2gray_accel
      g2r = cpipe.gray2rgb_accel
```

The Corner Detect is realized by adding (masking) the output of the Fast algorithm to the original image. In the Composable Overlay this is achieved by branching the pipeline, which is expressed as a list of a list.

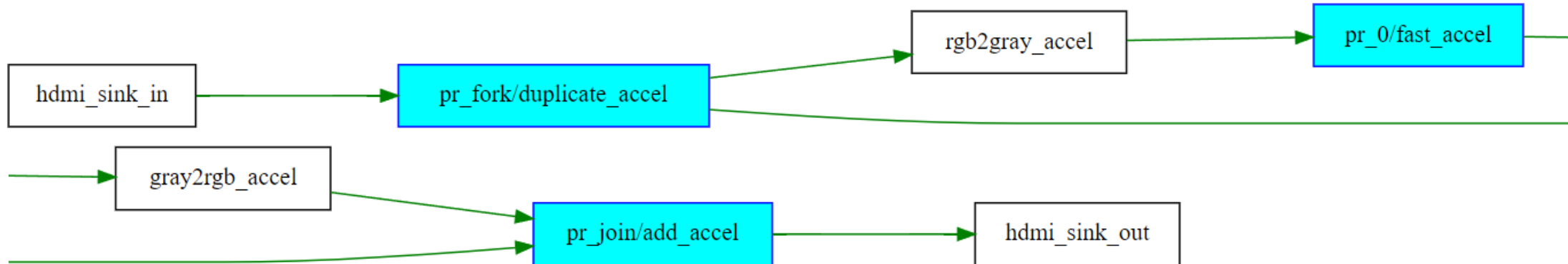
Composable Video Pipeline

08_webcam_pipeline.ipynb

```
[8]: video_pipeline = [cpipe.hdmi_sink_in, duplicate, [[r2g, fast, g2r], [1]], add, cpipe.hdmi_sink_out]  
      cpipe.compose(video_pipeline)  
      cpipe.graph
```



[8]:



Composable Video Pipeline

08_webcam_pipeline.ipynb

Modify Parameters

The corner Harris IP provides two parameters that help us tweak the sensitivity of the algorithm. These parameters are the threshold and k (Harris parameter), after running the next cell you will be able to update them.

```
[9]: thr = IntSlider(min=0, max=255, step=1, value=25)
def play(thr):
    fast.threshold = thr
interact(play, thr=thr);
```

thr  25

Composable Video Pipeline

Build Sources on Vivado Server

- The build process provides a way to modify Composable Video Pipeline project. After finishing build source (run **make**)
 - Manually move generated bitstream and hwh files to PYNQ-Z2 Board
 - Test overlay using Jupyter ipynb files
- The build process is scripted using a Makefile, when you run **make** the build process will do the following steps
 - Vision IP will be generated
 - PYNQ HLS IP will be generated
 - The Vivado project is created along with the IP design
 - The bitstream generation is launched
 - The bitstreams and hwh files are copied to the **overlay** folder
 - Files are versioned

Composable Video Pipeline

make Sources on Vivado Server

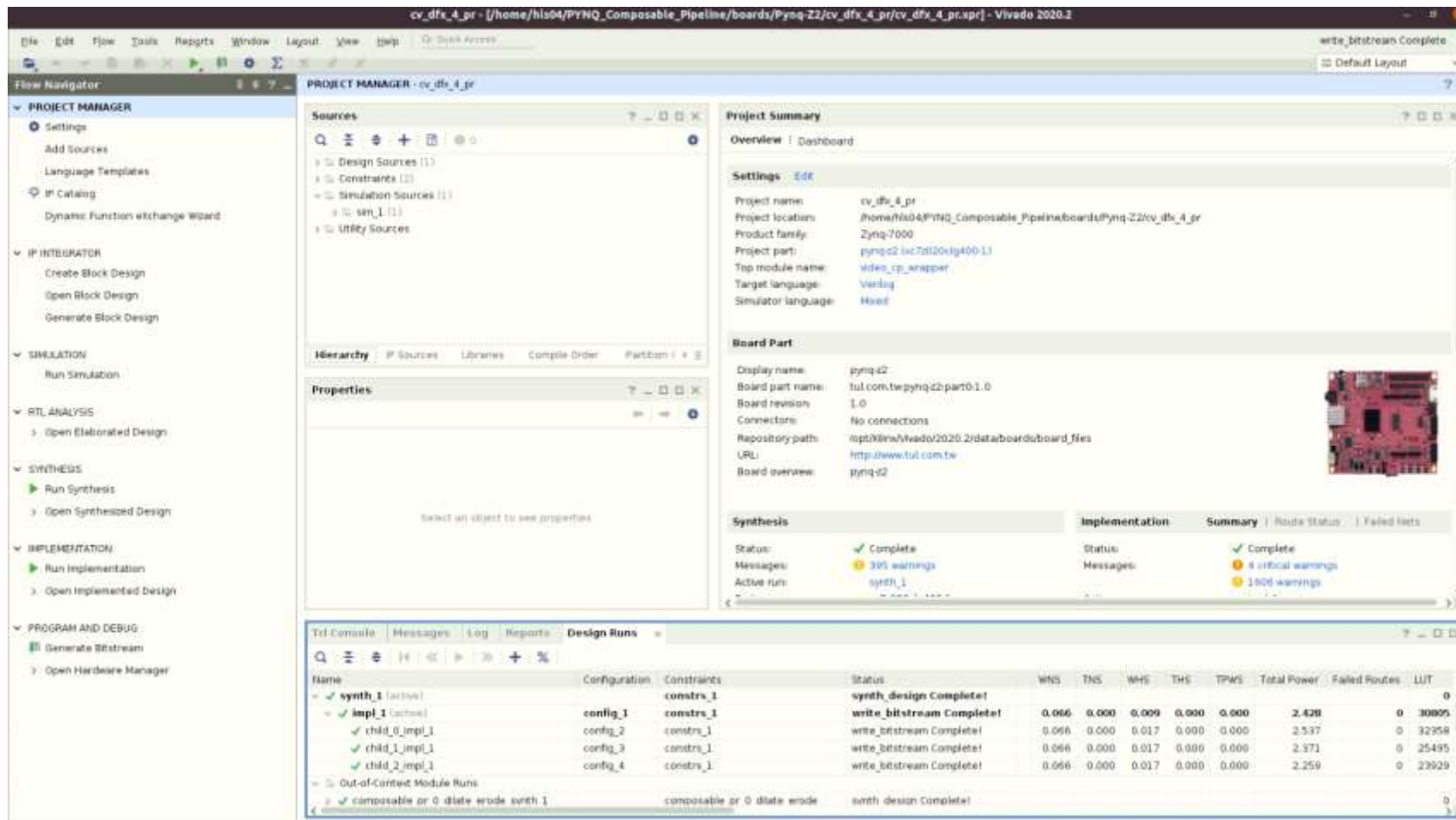
- Ubuntu 20.04 and Vivado 2020.2 (patched with [y2k22_patch](#))
- Install dependency and PYNQ-Z2 board files before executing **make**
 - `sudo apt install libc6-dev-i386 -y`
 - `unzip pynq-z2.zip -d /vivado_install_path/Vivado/2020.2/data/boards/board_files`

```
hls04@HLS04:~$ git clone https://github.com/Xilinx/PYNQ_Composable_Pipeline --recursive
Cloning into 'PYNQ_Composable_Pipeline'...
remote: Enumerating objects: 1729, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (38/38), done.
remote: Total 1729 (delta 9), reused 14 (delta 9), pack-reused 1682
Receiving objects: 100% (1729/1729), 5.09 MiB | 7.88 MiB/s, done.
Resolving deltas: 100% (1073/1073), done.
hls04@HLS04:~$
hls04@HLS04:~$ ls PYNQ_Composable_Pipeline/boards/Pynq-Z2/
cv_dfx_4_pr.tcl cv_dfx_4_pr.xdc default_paths.json ip LICENSE Makefile notebooks pinout.xdc README.md
hls04@HLS04:~$
hls04@HLS04:~$ cd PYNQ_Composable_Pipeline/boards/Pynq-Z2/
hls04@HLS04:~$ make
..... make process .....
hls04@HLS04:~$ ls
cv_dfx_4_pr cv_dfx_4_pr.xdc ip Makefile notebooks pinout.xdc vivado.jou
cv_dfx_4_pr.tcl default_paths.json LICENSE NA overlay README.md vivado.log
hls04@HLS04:~$
```


Composable Video Pipeline

Open Vivado Project

- Project file: /PYNQ_Composable_Pipeline/boards/Pynq-Z2/cv_dfx_4_pr/cv_dfx_4_pr.xpr



Composable Video Pipeline

Bitstream

```
hls04@HLS04:~/PYNQ_Composable_Pipeline/boards/Pynq-Z2/overlay$ ls -alF
```

```
total 13144
```

```
drwxrwxr-x 2 hls04 hls04 4096  17 23:28 ./
```

```
drwxrwxr-x 8 hls04 hls04 4096  18 10:14 ../
```

```
-rw-rw-r-- 1 hls04 hls04 4045676  17 23:28 cv_dfx_4_pr.bit
```

```
-rw-rw-r-- 1 hls04 hls04 1850480  17 23:28 cv_dfx_4_pr.hwh
```

Full Bitstream

```
-rw-rw-r-- 1 hls04 hls04 4869  17 23:28 cv_dfx_4_pr_composable.pkl
```

```
-rw-rw-r-- 1 hls04 hls04 975153  17 23:28 cv_dfx_4_pr_composable_pr_0_dilate_erode_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 64695  17 23:28 cv_dfx_4_pr_composable_pr_0_dilate_erode_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 975153  17 23:28 cv_dfx_4_pr_composable_pr_0_fast_fifo_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 54822  17 23:28 cv_dfx_4_pr_composable_pr_0_fast_fifo_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 975153  17 23:28 cv_dfx_4_pr_composable_pr_0_filter2d_fifo_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 55506  17 23:28 cv_dfx_4_pr_composable_pr_0_filter2d_fifo_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 1238433  17 23:28 cv_dfx_4_pr_composable_pr_1_cornerharris_fifo_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 56198  17 23:28 cv_dfx_4_pr_composable_pr_1_cornerharris_fifo_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 1238433  17 23:28 cv_dfx_4_pr_composable_pr_1_dilate_erode_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 64695  17 23:28 cv_dfx_4_pr_composable_pr_1_dilate_erode_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 359329  17 23:28 cv_dfx_4_pr_composable_pr_fork_duplicate_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 40697  17 23:28 cv_dfx_4_pr_composable_pr_fork_duplicate_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 359329  17 23:28 cv_dfx_4_pr_composable_pr_fork_rgb2xyz_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 34799  17 23:28 cv_dfx_4_pr_composable_pr_fork_rgb2xyz_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 209785  17 23:28 cv_dfx_4_pr_composable_pr_join_absdiff_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 40236  17 23:28 cv_dfx_4_pr_composable_pr_join_absdiff_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 209785  17 23:28 cv_dfx_4_pr_composable_pr_join_add_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 39480  17 23:28 cv_dfx_4_pr_composable_pr_join_add_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 209785  17 23:28 cv_dfx_4_pr_composable_pr_join_bitand_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 40762  17 23:28 cv_dfx_4_pr_composable_pr_join_bitand_partial.hwh
```

```
-rw-rw-r-- 1 hls04 hls04 209785  17 23:28 cv_dfx_4_pr_composable_pr_join_subtract_partial.bit
```

```
-rw-rw-r-- 1 hls04 hls04 40425  17 23:28 cv_dfx_4_pr_composable_pr_join_subtract_partial.hwh
```

Partial Bitstream

```
-rw-rw-r-- 1 hls04 hls04 650  17 23:28 cv_dfx_4_pr_paths.json
```

```
-rw-rw-r-- 1 hls04 hls04 2334  17 23:28 version.txt
```

Composable Video Pipeline

Default Paths

ci: (consumer interface) the output stream of this IP is connected to this consumer interface on the AXI4-Stream Switch

pi: (producer interface) the input stream of this IP is connected from this producer interface on the AXI4-Stream Switch

```
hls04@HLS04:~$ cat PYNQ_Composable_Pipeline/boards/Pynq-Z2/overlay/cv_dfx_4_pr_paths.json
```

```
{
  "composable": {
    "hdmi_source": {
      "ci": {
        "port": 0,
        "Description": "HDMI IN frontend PL path"
      },
      "pi": {
        "port": 0,
        "Description": "HDMI IN frontend PS path"
      }
    },
    "hdmi_sink": {
      "ci": {
        "port": 1,
        "Description": "HDMI OUT frontend PS path"
      },
      "pi": {
        "port": 1,
        "Description": "HDMI OUT frontend PL path"
      }
    }
  }
}
```

Composable Video Pipeline

Files are Versioned

```
hls04@HLS04:~$ cat PYNQ_Composable_Pipeline/boards/Pynq-Z2/overlay/version.txt
board = Pynq-Z2
git_id = 8e2cd7dba3de06a8e8fe66c61aa9829e13daf35e
date = 17 四月 2022
version = 1.0.2
---- md5sum                Files ----
3edff519d841c60d2d4fdd28b953ea4e overlay/cv_dfx_4_pr.bit
102a19da238e1f75b699cd26ddf06a2c overlay/cv_dfx_4_pr_composable_pr_0_dilate_erode_partial.bit
6622470892193fd9bd42a7a79194d57b overlay/cv_dfx_4_pr_composable_pr_0_dilate_erode_partial.hwh
310dc4f5973a681d67d364e04305f578 overlay/cv_dfx_4_pr_composable_pr_0_fast_fifo_partial.bit
da063c04449e1b3d677526454c144edd overlay/cv_dfx_4_pr_composable_pr_0_fast_fifo_partial.hwh
d76f7dfad034ba82c3e5803b620463bd overlay/cv_dfx_4_pr_composable_pr_0_filter2d_fifo_partial.bit
5cdc025105bd7b06bc0c8250e7d5caaa overlay/cv_dfx_4_pr_composable_pr_0_filter2d_fifo_partial.hwh
b0488361b2054d58a504a04d1fdfed50 overlay/cv_dfx_4_pr_composable_pr_1_cornerharris_fifo_partial.bit
761d814efe4f0421fcf8330aa99c77a4 overlay/cv_dfx_4_pr_composable_pr_1_cornerharris_fifo_partial.hwh
7dc46308a8a98a20398bd75cf75ec4bb overlay/cv_dfx_4_pr_composable_pr_1_dilate_erode_partial.bit
37e57d6e8a8744f4294fa9034f08a627 overlay/cv_dfx_4_pr_composable_pr_1_dilate_erode_partial.hwh
b968e79c03c2e42921aa39e980966fcd overlay/cv_dfx_4_pr_composable_pr_fork_duplicate_partial.bit
427eb663e187a0ee058c78b58efb803d overlay/cv_dfx_4_pr_composable_pr_fork_duplicate_partial.hwh
ed3b8cb90cdb558060b208ed74b36a8a overlay/cv_dfx_4_pr_composable_pr_fork_rgb2xyz_partial.bit
895b0d20fef6c317248f54717ce1f324 overlay/cv_dfx_4_pr_composable_pr_fork_rgb2xyz_partial.hwh
65ec1bc6fc445ce61f2bac2bc5f4d7b5 overlay/cv_dfx_4_pr_composable_pr_join_absdiff_partial.bit
7869ca144901c39b3fafac03fd0df117 overlay/cv_dfx_4_pr_composable_pr_join_absdiff_partial.hwh
1b7acc9a66191aba928de55cfbcc450c overlay/cv_dfx_4_pr_composable_pr_join_add_partial.bit
d18840a5dd01b861af2cf374a2d9e9f7 overlay/cv_dfx_4_pr_composable_pr_join_add_partial.hwh
372c0618cc6ca670a9d9f243866b3686 overlay/cv_dfx_4_pr_composable_pr_join_bitand_partial.bit
0ea5504fb7ddb7f544ceedc3a58e924c overlay/cv_dfx_4_pr_composable_pr_join_bitand_partial.hwh
1631af313b62b2266bb65b4610daf6e4 overlay/cv_dfx_4_pr_composable_pr_join_subtract_partial.bit
1a2cde807af63b836f2ec1908c0c540a overlay/cv_dfx_4_pr_composable_pr_join_subtract_partial.hwh
5bdc04faaf8663ddec5f741d14d42ecd overlay/cv_dfx_4_pr.hwh
```

References

- PYNQ: Python productivity for Xilinx platforms
 - <https://pynq.readthedocs.io/en/v2.7.0/index.html>
- PYNQ Composable Overlays
 - https://pynq-composable.readthedocs.io/en/latest/pynq_composable.html
- PYNQ_Composable_Pipeline
 - https://github.com/Xilinx/PYNQ_Composable_Pipeline
- The Composable Overlays Overview
 - <https://youtu.be/nKu8dVKDweg>

Resources

- PYNQ-Z2 Board Files
 - <https://dpoauwggwqsy2x.cloudfront.net/Download/pynq-z2.zip>
- PYNQ Community
 - <http://www.pynq.io/community.html>
- element14 Community
 - <https://community.element14.com/>