

## 目 录

CV 方向要求 .....	2
需要完成的功能: .....	2
完成要求 .....	2
<b>基础知识 .....</b>	<b>3</b>
Overlay 架构 .....	3
静态 Static IP 调用 .....	3
动态 DFX Overlay 调用 .....	4
使用 Python 调整 IP 参数 .....	6
Webcam 输入 (VDMA 的使用) .....	6
Python+Composable Pipeline 协同设计.....	8
<b>Composable Pipeline 结构 .....</b>	<b>11</b>
<b>参考资料及链接 .....</b>	<b>14</b>
Composable Pipeline 简介.....	14
官方参考资料 .....	14
Composable Pipeline 的 GitHub .....	14

## CV 方向要求

### CV主题任务

#### ▸ 描述

- 完成一项软硬协同设计，需具备以下列出功能中的任意一项：
  - 手势识别、目标检测、人脸检测、数字识别、图像增强

#### ▸ 要求

- 使用PYNQ作为软件框架，使用PYNQ-Z2作为开发板
- 必须使用到Composable Pipeline 作为 Overlay，进行合理的负载划分
- 建议在Composable Pipeline 中放置自定义算子，完成核心算法的硬件加速
- 设计具备良好的可视化效果

#### ▸ 参考示例

- [Composable Pipeline仓库](#)
- [2021 Summer School CV方向](#)
- [Github Vitis Vision Library](#)

### 需要完成的功能：

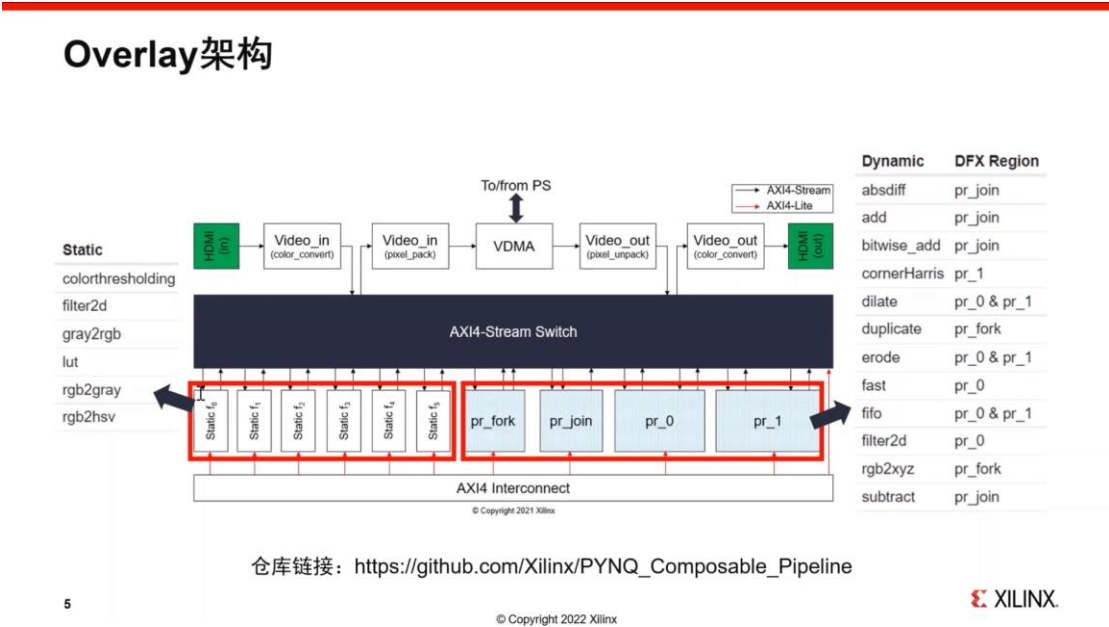
手势识别、**目标检测**、**人脸检测**、数字识别、图像增强。  
**完成软硬件协同设计**

### 完成要求

- **使用 PYNQ 以及 PYNQ-Z2;**
- **使用 Composable Pipeline 作为 Overlay;**
- 在 Composable Pipeline 中放置自定义算子,完成核心算法的硬件加速,可以修改 static 区域,也可以修改 DFX 区域;
- 也可以导入 VDMA 之后,进行 Python 的处理;
- 良好的可视化效果;
- 在 GitHub Vitis Vision Library 中有 Xilinx 优化过的 IP 可以进行使用。

基础知识

Overlay 架构



6 个 static 静态算子都是常见的，在旁边的动态区可以放置动态重构的算子，可以进行自定义配置和切换。VDMA 是 PS 和 PL 之间一个交互的方式，AXI4-Stream Switch 类似一个交换机。DFX 在左边浅蓝色的动态区。

已有Overlay

```
cpipe.c_dict
▼ composable:
  ► video/hdmi_in/color_convert [loaded]:
  ► video/hdmi_in/pixel_pack [loaded]:
  ► video/hdmi_out/pixel_unpack [loaded]:
  ► video/hdmi_out/color_convert [loaded]:
  ► video/composable/rgb2gray_accel [loaded]:
  ► video/composable/gray2rgb_accel [loaded]:
  ► video/composable/rgb2hsv_accel [loaded]:
  ► video/composable/filter2d_accel [loaded]:
  ► video/composable/lut_accel [loaded]:
  ► video/composable/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/absdiff_accel [unloaded]:
  ► pr_join/bitwise_and_accel [unloaded]:
  ► pr_join/add_accel [unloaded]:
  ► pr_join/subtract_accel [unloaded]:
  ► pr_fork/duplicate_accel [unloaded]:
  ► pr_fork/rgb2xyz_accel [unloaded]:

[6]: cpipe.dfx_dict
[6]: ▼ dfx_dict:
  ► pr_0:
    ► decoupler: "/dfx_decouplers/dfx_decoupler_pr_0"
    ► gpio:
      ► decouple: 0
      ► status: 1
  ► ip:
    ► cv_dfx_4_pr_pr_0_dilate_erode_partial.bit:
      ► pr_0/dilate_accel:
        ► bitstream: "/home/xilinx/jupyter_notebooks/composable-pipeline/overlay/cv_dfx_4_pr_pr_0_dilate_erode_partial.bit"
        ► interface: [ ] 2 items
          0: "/pr_0/stream_in1"
          1: "/pr_0/stream_out1"
        ► modtype: "dilate_accel"
      ► pr_0/erode_accel:
      ► cv_dfx_4_pr_pr_0_fast_fifo_partial.bit:
      ► cv_dfx_4_pr_pr_0_filter2d_fifo_partial.bit:
  ► pr_1:
  ► pr_fork:
  ► pr_join:
```

可以通过在 Jupyter 中输入最上面那一条命令来查看 DFX 的有关信息。

静态 Static IP 调用

下面是静态区调用 composable pipeline。

## 1. static ip调用

- ▶ 目标：实现rgb，hsv与灰度图相互转化，2d滤波等简单功能
- ▶ 第一步：下载Overlay
- ▶ 第二步：获取功能函数ip

```
video_in_in = cpipe.video.hdm_in.color_convert
video_in_out = cpipe.video.hdm_in.pixel_pack

filter2d = cpipe.video.composable.filter2d_accel
rgb2gray = cpipe.video.composable.rgb2gray_accel
gray2rgb = cpipe.video.composable.gray2rgb_accel
rgb2hsv = cpipe.video.composable.rgb2hsv_accel
colorthr = cpipe.video.composable.colorthresholding_accel
lut = cpipe.video.composable.lut_accel
```

- ▶ 第三步：先实现rgb图与灰度图的转化

```
video_pipeline = [video_in_in, rgb2gray, video_in_out]
cpipe.compose(video_pipeline)
cpipe.graph
```

12

© Copyright 2022 Xilinx

XILINX

## 1. static ip调用

- ▶ 第四步：替换ip，将rgb2gray替换为rgb2hsv

```
cpipe.replace([rgb2gray, rgb2hsv])
cpipe.graph
```

- ▶ 第五步：删除ip，将rgb2hsv这个ip删除

```
cpipe.remove([rgb2hsv])
cpipe.graph
```

- ▶ 第六步：插入ip，将filter2d和gray2rgb插入，其中数字为插入该ip的位置

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

cpipe.insert([gray2rgb],3)
cpipe.graph
```

- ▶ 第七步：停止pipeline，否则将会影响下次使用

```
video.stop()
```

13

© Copyright 2022 Xilinx

XILINX

## 动态 DFX Overlay 调用

下面是 DFX Overlay 的调用

## 2. DFX Overlay调用

- ▶ 目标：调用DFX区域的Overlay，并实现有分支（并行）的pipeline
- ▶ 第一步：下载Overlay

```
from pynq.lib.video import *
from composable_pipeline import ComposableOverlay
from composable_pipeline.libs import *

ol = ComposableOverlay("../overlay/cv_dfx_4_pr.bit")

cpipe = ol.composable
video = HDMIVideo(ol)
video.start()
```

## 2. DFX Overlay调用

- 第二步：载入函数dilate\_accel和erode\_accel的ip

```
cpipe.c_dict.loaded
cpipe.loadIP([cpipe.pr_1.dilate_accel])
cpipe.c_dict.loaded
```

- 第三步：搭建pipeline

```
video_in_in = cpipe.video.hdmi_in.color_convert
video_in_out = cpipe.video.hdmi_in.pixel_pack

dilate = cpipe.pr_1.dilate_accel
cpipe.compose([video_in_in, dilate, video_in_out])

cpipe.graph
```

## 2. DFX Overlay调用

- 第四步：搭建有分支的pipeline

```
cpipe.loadIP(['pr_fork/duplicate_accel', 'pr_join/subtract_accel', 'pr_0/filter2d_accel'])
filter2d = cpipe.video.composable.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 = 'gaussian_blur'

filter2d_d.sigma = 12
filter2d_d.kernel_type = 'gaussian_blur'
video_pipeline = [video_in_in, filter2d, duplicate, [[filter2d_d, [1]], subtract, video_in_out]
cpipe.compose(video_pipeline)

cpipe.graph
```

- 第五步：停止pipeline，否则将会影响下次使用

```
video.stop()
```

## 使用 Python 调整 IP 参数

### 3. 用python调整IP的参数

- ▶ 用python构建可视化的组件调整IP参数

```
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')

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

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

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

19

© Copyright 2022 Xilinx

XILINX

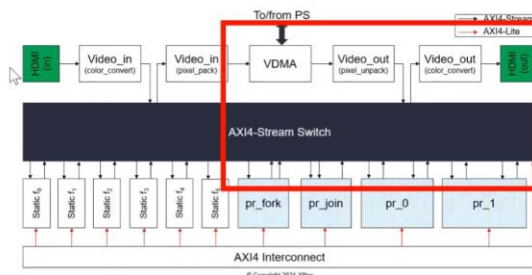
如上图所示，可以利用 Python 对算子的类型和参数阈值进行自定义，并设计一些可视化的组件。

以上是使用 HDMI 进行输入，下面是使用 Webcam 进行输入，也就是调用 VDMA 进行处理，从 video\_out\_in 输入，从 video\_out\_out 输出。

### Webcam 输入（VDMA 的使用）

#### 使用Webcam作为输入时的数据通路

- ▶ 从这个例子中可以发现，我们先把Webcam的输入加载到了板卡上的VDMA中，然后通过Video\_out\_in(pixel unpack)和Video\_out\_out(color convert)连接 Composable Overlay，与前几个例子中在Video\_in处连接pipeline不同。具体数据通路如下图所示：



22

© Copyright 2022 Xilinx

XILINX

## Webcam作为输入

### 第一步：下载Overlay

```
from pynq.lib.video import *
from composable_pipeline import ComposableOverlay
from composable_pipeline.libs import *
from composable_pipeline.video import *
from ipywidgets import widgets, interact, FloatSlider, IntSlider

ol = ComposableOverlay("../overlay/cv_dfx_4_pr.bit")

cpipe = ol.composable
```

## Webcam作为输入

### 第二步：搭建Webcam作为输入时的数据通路

```
video_in = Webcam()

hdm_i_out = ol.video.hdm_i_out
hdm_i_out.configure(video_in.mode, PIXEL_BGR)

video_in.start()
hdm_i_out.start();
video_in.tie(hdm_i_out)
```

## Webcam作为输入

### 第三步：搭建pipeline

```
video_out_in = cpipe.video.hdm_i_out.pixel_unpack
video_out_out = cpipe.video.hdm_i_out.color_convert
```

在载入算法将会用到的ip之前，注意要先将Webcam的输入暂停

```
video_in.pause()
cpipe.loadIP(['pr_1/cornerHarris_accel', 'pr_fork/duplicate_accel', 'pr_join/add_accel'])
```

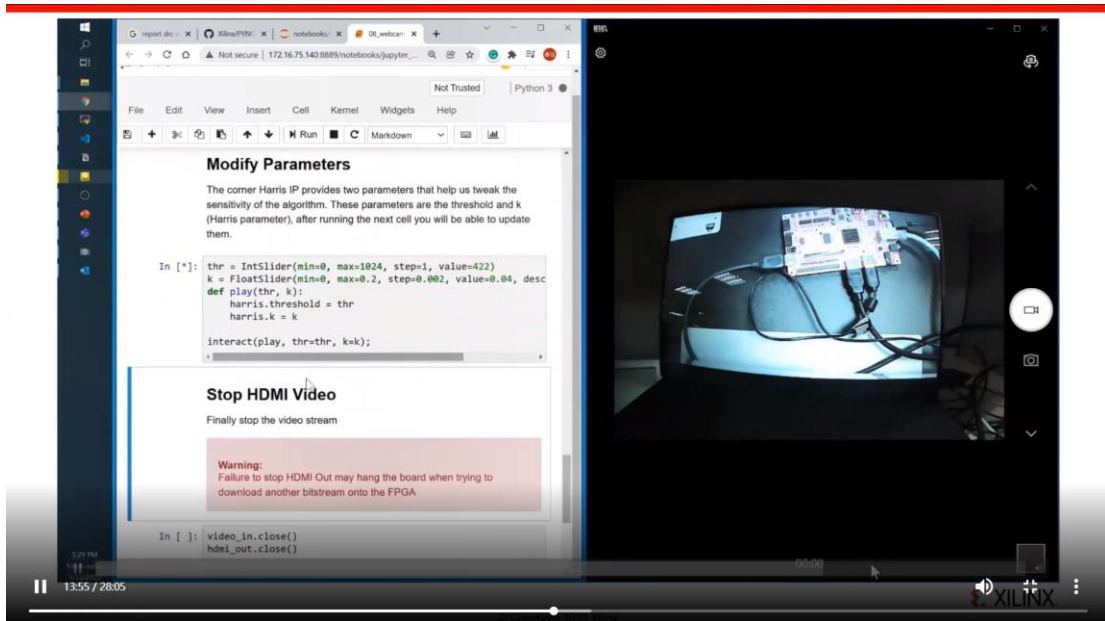
在load完成后，再重新启动

```
video_in.tie(hdm_i_out)
harris = cpipe.pr_1.cornerHarris_accel
duplicate = cpipe.pr_fork.duplicate_accel
add = cpipe.pr_join.add_accel
r2g = cpipe.video.composable.rgb2gray_accel
g2r = cpipe.video.composable.gray2rgb_accel
video_pipeline = [video_out_in, duplicate, [[r2g, harris, g2r], [1]], add, video_out_out]

cpipe.compose(video_pipeline)

cpipe.graph
```



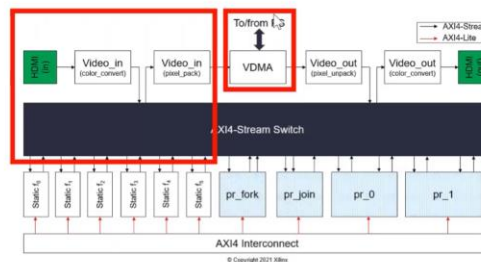


## Python+Composable Pipeline 协同设计

小恐龙游戏完成自动目标检测和避障。

### python结合已有ip进行开发时的数据通路

- 在本例中，我们先通过Video\_in\_in(color convert)和Video\_in\_out(pixel unpack)连接Composable Overlay，然后python通过VDMA访问其输出的数据进行处理，如下图所示：



在 PS 侧，通过 VDMA 将数据导入，而后进行处理，前半段的 look up table 和膨胀侵蚀的 PL 操作，就是在原来的通路中完成的，而后再进入 video\_in(pixel\_pack)，再进入 python 中进行一些操作，例如 opencv 等，再回到原始路径中去，原始路径就是 video\_out(pixel\_unpack)之后的路径。



## 小恐龙外挂

### 第一步：下载Overlay

```
from composable_pipeline import ComposableOverlay
from time import sleep
import numpy as np
import matplotlib.pyplot as plt
import serial

ol = ComposableOverlay("./overlay/cv_dfx_4_pr.bit")
cpipe = ol.composable
```

## 小恐龙外挂

### 第二步：调用算法所需的ip

```
cpipe.loadIP([cpipe.pr_0.erode_accel, cpipe.pr_0.dilate_accel, cpipe.pr_1.erode_accel, cpipe.pr_1.dilate_accel])
```

### 第三步：搭建数据通路

```
hdm_i_in = ol.video.hdm_i_in
hdm_i_out = ol.video.hdm_i_out
hdm_i_in.configure()
hdm_i_out.configure(hdm_i_in.mode)
print(hdm_i_in.mode)
print(hdm_i_out.mode)
hdm_i_in.start()
hdm_i_out.start()
hdm_i_in.tie(hdm_i_out)

VideoMode: width=1280 height=720 bpp=24
VideoMode: width=1280 height=720 bpp=24
```

载入 overlay，而后搭建数据通路，调用膨胀和侵蚀的 IP，而后进行一些配置，如 1280\*720 的配置。

## 小恐龙外挂

### 第四步：搭建pipeline

```
video_in_in = cpipe.video.hdm_i_in.color_convert
video_in_out = cpipe.video.hdm_i_in.pixel_pack
lut = cpipe.video.composable.lut_accel
erode_0 = cpipe.pr_0.erode_accel
dilate_0 = cpipe.pr_0.dilate_accel
erode_1 = cpipe.pr_1.erode_accel
dilate_1 = cpipe.pr_1.dilate_accel

lut.kernel_type = 'binary_threshold'

cpipe.compose([video_in_in, lut, dilate_0, dilate_1, erode_0, erode_1, video_in_out])
```

先把需要用到的算子进行实例化，而后设置 look up table 的类型，而后用 cpipe.compose 搭建一个通路，里面是膨胀侵蚀等操作和获得一个 LUT 查找表。以上就完成了 PL 部分的搭建，也就是把 composable pipeline 通路搭建完毕了。

下面是 PS 侧功能，也就是判断黑色障碍物

## 小恐龙外挂

- ▶ 第五步：使用python对composable pipeline的输出图像进行进一步处理，根据处理后的图像判断小恐龙是否应该起跳

```
class DinoBot:
    def __init__(self, hdmi_in):
        self.hdmi_in = hdmi_in
        self.serial = serial.Serial('/dev/ttyUSB0', 115200)
        if self.serial.isOpen():
            print("serial open success")
        else:
            print("serial open failed")
    def start(self):
        self.serial.write(b'press')
        sleep(0.1)
        self.serial.write(b'release')
        while(True):
            try:
                frame = self.hdmi_in.readframe()
                img_slice = frame[330:450, 320:350]
                if((img_slice==255).all()):
                    self.serial.write(b'release')
                else:
                    self.serial.write(b'press')
            except KeyboardInterrupt:
                break
```

首先用到 hdmi.readframe 封装来读取一帧画面，在 Python 程序的 frame 中，就是选出一个框范围，截图出一个 slice 出来，通过串口输出一些空格键的指令。判断框内是否为黑色，如果是，则跳跃，否则是正常行走。

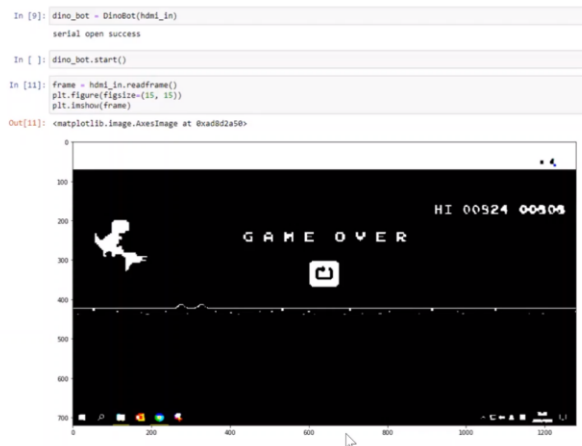
下面就是一个最终的结果了：

## 小恐龙外挂

- ▶ 第六步：启动pipeline

- ▶ 第七步：停止pipeline，否则将会影响下次使用

```
hdmi_in.close()
hdmi_out.close()
```

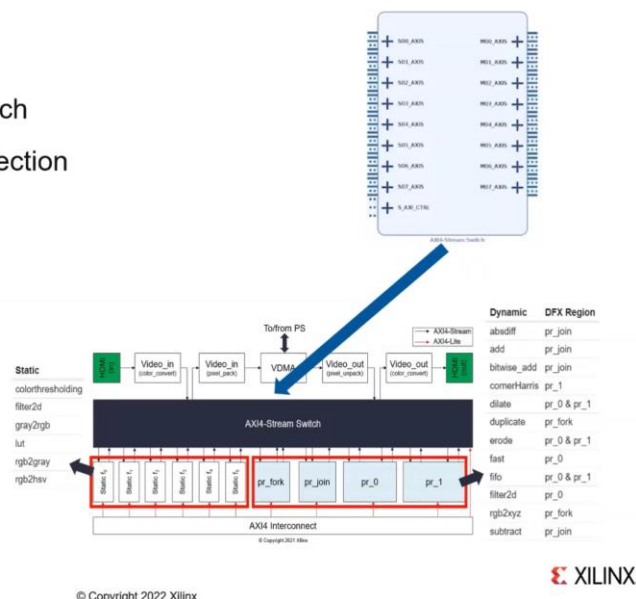


还有一些有趣的应用，比如机械臂识别抓取物品。

## Composable Pipeline 结构

### Composable

- ▶ Based on AXI4-Stream Switch
- ▶ Runtime point to point connection
- ▶ One design, multiple paths
- ▶ Limited to 16 ports
  - 16! Combinations
  - Use 15/16 now
  - 6 static + 7 dynamic + 2 video



Composable 就是可组合的意思，核心就是 AXI4-stream，支持 16 输入 16 输出的接口的配置，所以就有 16 中配置的方式，可以从任何一个端口连接到对面的端口去，并且是可以在运行时进行配置。因此我们可以将这些 IP 进行动态的调整。

### Dynamic

- ▶ Dynamic Function eXchange (DFX)
  - Swap functions on-the-fly
- ▶ Augments the overlay functionality
- ▶ Xilinx silicon ability to be dynamically reconfigured

Vivado Design Suite User Guide

Dynamic Function eXchange

UG089 v0021.0 December 17, 2021

Vivado Design Suite Tutorial

Dynamic Function eXchange

UG047 v0021.0 December 17, 2021

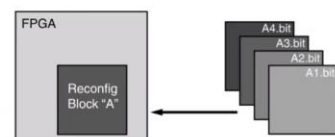


Figure 1-1: Basic Premise of Dynamic Function eXchange

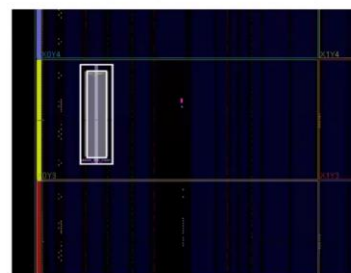


Figure 1: Pblock for the test, count Reconfigurable Partition

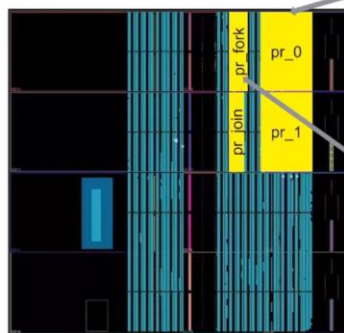
另外一个重要的功能就是 DFX (动态函数交换)，可以动态配置 IP，节省 FPGA 资源和空间，在结构图中的右下角部分就是 DFX 部分，里面有很多的 IP，在不用的时候配置在里面会占用 FPGA 资源，需要的时候再下载入 DFX 中即可。

优点是，当 FPGA 片上资源不够的时候，先把一些小的比特流文件下到设计中去，做一个运行时的硬件级别的重构。

## The composable Pipeline

Static	Dynamic	pr_0	pr_1	pr_fork	pr_join
colorThresh	absdiff				✓
filter2d	add				✓
gray2rgb	bitwise_and				✓
lut	cornerHarris		✓		
rgb2gray	dilate	✓	✓		
rgb2hsv	duplicate			✓	
	erode	✓	✓		
	fast	✓			
	fifo	✓	✓		
	filter2d	✓			
	rgb2xyz			✓	
	subtract				✓

函数的PR占用情况



所指定的4个DFX区域

1x full bitstream and 11x partial bitstreams

Pblock Properties			
Physical Resource Estimates			
Site Type	Available	Used	% Used
Slice LUTs	8000	3364	42.05
LUT as Logic	8000	3298	41.23
LUT as Memory	2800	66	2.36
Slice Registers	16000	4474	27.96
Register as Flip Flop	16000	4474	27.96
Register as Latch	16000	0	0
F7 Muxes	4000	20	0.50
F8 Muxes	2000	0	0
Slice	2000	1248	62.40

Pblock Properties			
Physical Resource Estimates			
Site Type	Available	Used	% Used
Slice LUTs	1600	1006	62.88
LUT as Logic	1600	842	58.88
LUT as Memory	600	64	10.67
Slice Registers	3200	1302	40.69
Register as Flip Flop	3200	1302	40.69
Register as Latch	3200	0	0
F7 Muxes	800	0	0
F8 Muxes	400	0	0
Slice	400	358	89.50

上图是 composable pipeline 在 vivado device 的一个界面，图中可以看出他分出来 4 个动态可重构区域，就是图中黄色的 4 个 pr 区域，这 4 个区域具有 DFX 的性质

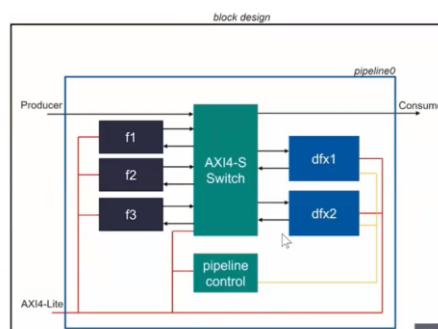
在上图中同样可以看出，在 pr\_0 中已经配置好了资源，比如大的设计分配了 8000 个 look up table，小的设计就分配 1600 个 LUT。而我们常用的这些函数就可以占据这些区域进行动态配置。

上图中的左侧函数部分，当我们需要使用这些函数的时候，将这个函数编成一个 partial 的比特流下载到对应的 pr 区去，而不会对外面的这些设计产生影响。

最终完整的 composable pipeline 设计就是 1 个完整的比特流文件，并含有 11 个 partial 比特流文件（图中左侧的函数的比特流文件）。

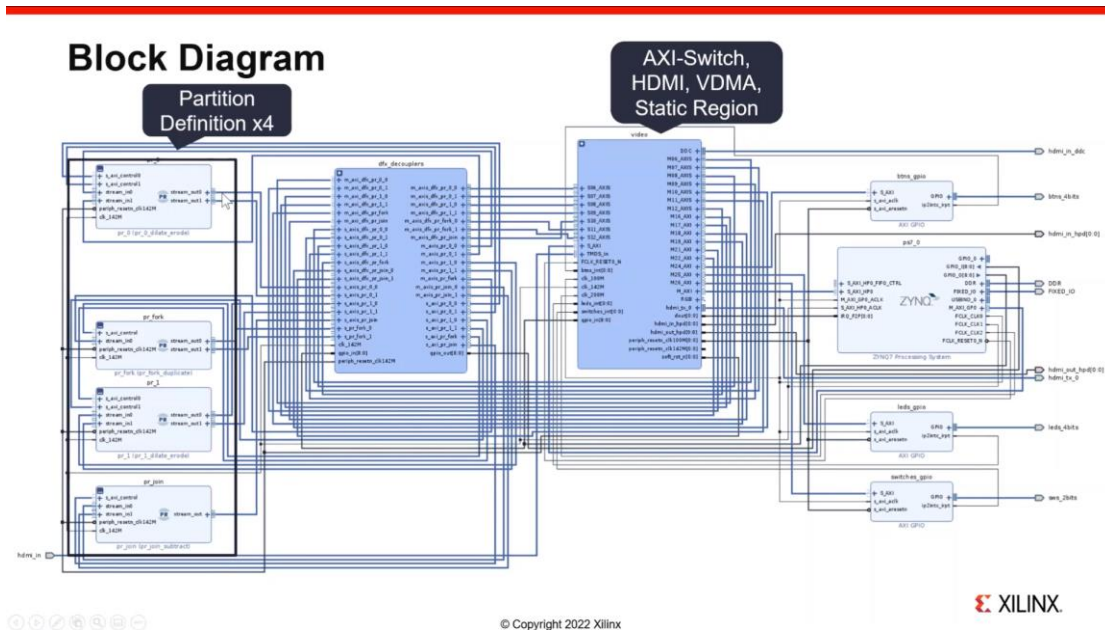
## Block Design

- ▶ AXI4-Stream Switch
- ▶ Pipeline Control
  - Soft reset
  - DFX control
- ▶ Static Functions
- ▶ DFX Regions
  - Add reconfigurable modules



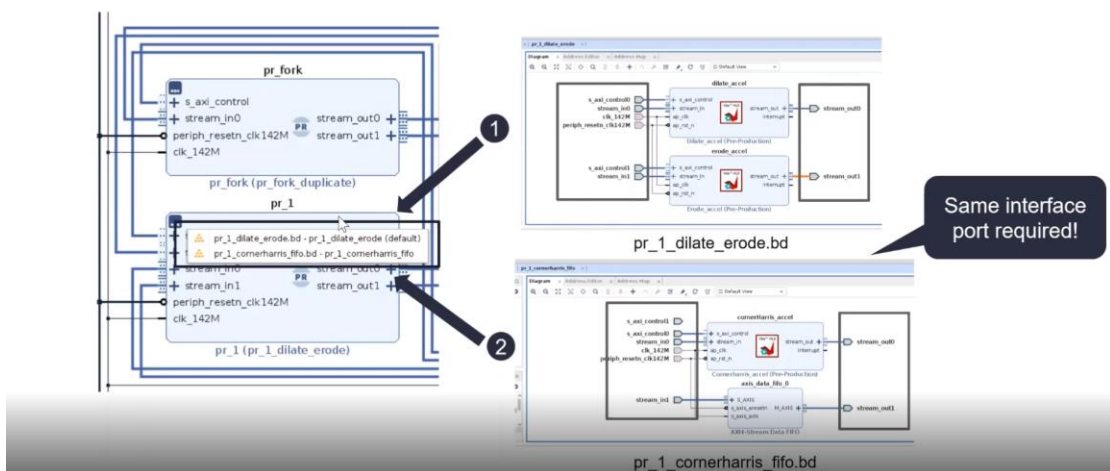
上图就是 composable pipeline 的一个整体设计情况，有静态区域和动态区域，也有 pipeline control 的函数，用胶水逻辑来粘合整个工程。

从 block design 的角度来看的话，如下图：



最左边就是 DFX 区域 4 个，是动态可重构的。其余部分，都是静态的区域。下面来具体讲一下 DFX 的配置，如下图，拿 pr\_fork 和 pr\_1 进行举例。

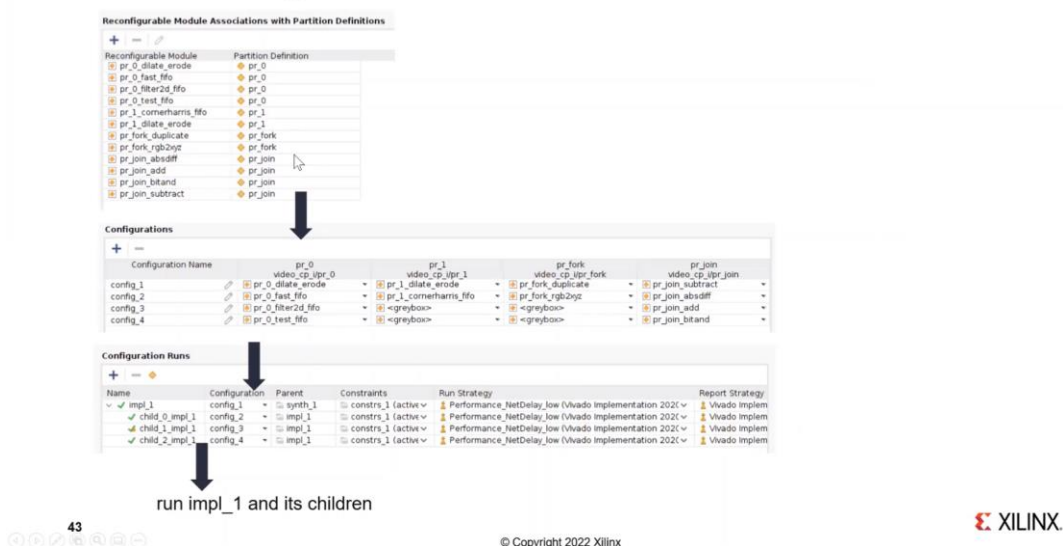
## Reconfigurable Modules



比如希望 pr\_1 支持两个算子，但是可能放不下，此时可以设置 DFX 属性，在 Block Diagram 中进行设置。点开可以看见这两个例子一个是膨胀侵蚀和 harris，看起来差不多，接口也完全相同，都是 AXI\_Control 和 stream 的接口，可以看 Harris 中，由于有一个 FIFO 所以 AXI\_Control 没有用到被悬空。

所以我们在进行动态设计的时候，只需要保证接口区域完全一致，在 Diagram 内部怎样改动都是可以的，只要资源可以承载的下。

## Add Reconfigurable Modules



如上图所示，在加载这些函数的时候，我们就可以去定义一下 pr\_0、pr\_1、pr\_fork、pr\_join 上的每一个函数。配置在 configuration 里面。可以分成几个 implement 去实现，然后生成比特流。

## 参考资料及链接

### Composable Pipeline 简介

[https://mp.weixin.qq.com/mp/homepage?\\_\\_biz=MzU0Mzk1MzM5NQ==&hid=9&sn=5d2ac6e24b255385b9062628824f5383&scene=18](https://mp.weixin.qq.com/mp/homepage?__biz=MzU0Mzk1MzM5NQ==&hid=9&sn=5d2ac6e24b255385b9062628824f5383&scene=18)

### 官方参考资料

1. PYNQ 官方社区: <http://www.pynq.io/community.html>
2. PYNQ 官方文档: <https://pynq.readthedocs.io/en/latest/index.html>
3. Xilinx 中文学习资料与开源设计: <https://github.com/xupsh>
4. PYNQ 开源社区微信公众号的菜单栏索引，包含丰富的项目案例与分享。

### Composable Pipeline 的 GitHub

[https://github.com/Xilinx/PYNQ\\_Composable\\_Pipeline](https://github.com/Xilinx/PYNQ_Composable_Pipeline)