PYNQ 开发教程

2019.9.24 17:30:51

版权声明

公司网址:
http://www.alinx.com

技术论坛:
http://www.heijin.org

官方旗舰店:
http://alinx.jd.com

邮箱:
avic@alinx.com.cn

电话:
021-67676997

传真:
021-37737073

ALINX 微信公众号:

文档修订记录:

版本	时间	描述
1.01		初始版本

我们承诺本教程并非一劳永逸，固守不变的文档。我们会根据论坛上大家的反馈意见，以及实际的开发实践经验积累不断的修正和优化教程

序

首先感谢大家购买芯驿电子科技（上海）有限公司出品的 ZYNQ 的开发板 AX7010、AX7020！ 您对我们和我们产品的支持和信任,给我们增添了永往直前的信心和勇气。

“播下一粒种子，收获一片森林”，更是芯驿电子科技（上海）有限公司的美好愿望，同时我们会在黑金动力社区 *http://www.heijin.org* 和大家一起讨论，一起学习，一起进步，一起成长。

目录

第一章 PYNQ 简介

1.1 PYNQ 能简化 ZYNQ 开发吗

PYNQ 主要目标是使用 **P**ython 开发 Z**ynq**，使设计者能快速体验基于 zynq 的嵌入式开发，按照 Xilinx 官方的说法：让不懂 FPGA、甚至不懂 C 语言的人来开发 ZYNQ。真是如此吗？我们来看一下 PYNQ 的核心部分是什么？

Overlays，这个用中文很难表述清楚，本质是 FPGA 的编译后的结果，是一个 bit 流文件，PYNQ 可以动态加载、卸载这些 bit 流，这样的话 FPGA 端的功能就可以根据 ARM 软件的需求动态改变。那么，如果不会开发 FPGA，这些 bit 流文件怎么来？如果在大公司，可以找 FPGA 开发人员开发，如果是规模一般的公司，那么开发者还是老老实实先把 FPGA 学扎实，关于多久能学好 FPGA 的问题，笔者理解是经过坚持不懈地练习，大概一年左右。

PYNQ 使用 python 为开发语言，迄今为止，C 或 C++是最常用的嵌入式编程语言，因为嵌入式经常和硬件底层打交道，虽然 python 可以提升开发效率，但是还是要有人为 Python 开发库，这些底层库基本都是 C 或 C++。

PYNQ 是基于浏览器的，使用一种叫 Jupyter notebook 的技术，这个不是 ZYNQ 特有的，普通的 PC 也可以。

总结一下：

- PYNQ= Python + ZYNQ
- FPGA 还是需要有人开发
- 还是要有嵌入式开发人员给 Python 提供底层接口
- 不能简化 ZYNQ 开发

1.2 为什么还要用 PYNQ 呢?

- 非常熟悉 Python，以前的算法都是 Python 上写好的，现在需要移植
- 确实不会 C 或 C++，也不会搭建交叉编译环境
- Xilinx 提供了一部分底层硬件的 Python 库，例如寄存器读写、DMA 使用，有了这些基本的操作，可以解决大部分对效率要求不高的嵌入式程序，用于调试还是很方便的
- PYNQ 属于锦上添花，在学校 ZYNQ 的同时，对于有 Python 基础，但是不想再系统学习嵌入式开发的人员来说是非常合适的工具

1.3 不会 python 能用 PYNQ 吗?

如果不会 python，这个真不行!

1.4 PYNQ 学习资源

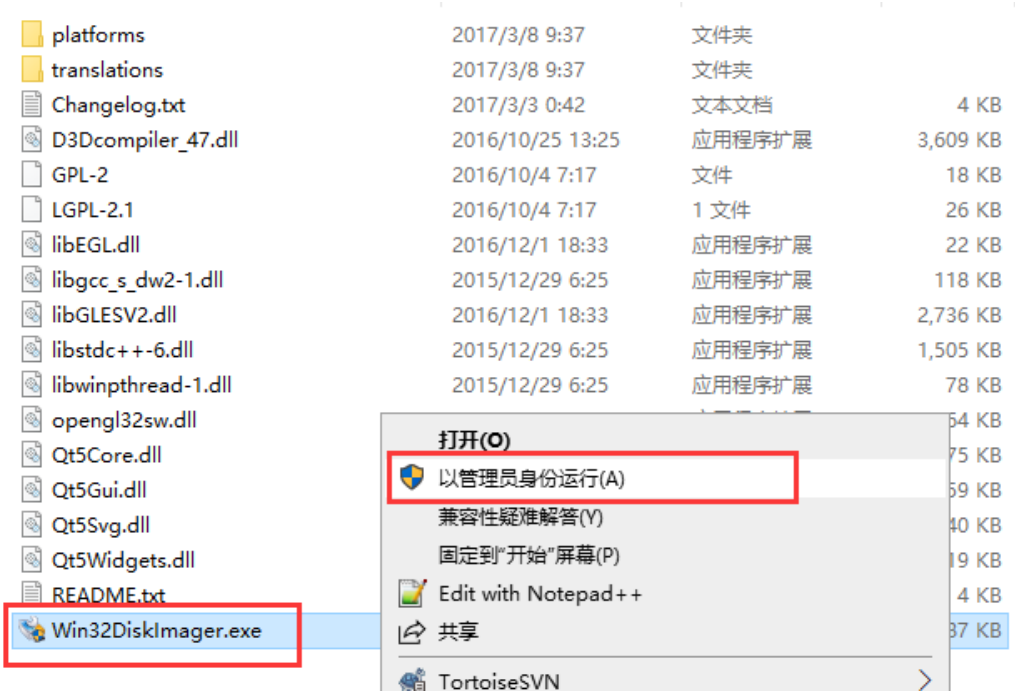
官网：http://www.pynq.io
Github: https://github.com/Xilinx/PYNQ

第二章 PYNQ 快速上手

2.1 PYNQ 镜像

AX7010/AX7020 出厂程序不支持 PYNQ，需要重新烧写镜像到 SD 卡，烧录方法如下（关于如何制作一个 PYNQ 镜像，可以参考官网相关教程，本教程不提供镜像制作方法）

1. 解压镜像压缩包 ax7010-2.4.zip 或 ax7020-2.4.zip，得到一个扩展名为 img 的文件
2. sd 卡（至少 8G，烧写后原有内容全部丢失）用读卡器插在电脑 USB 口，准备烧写
3. 下载 Win32DiskImager 镜像烧写工具，这个工具用途广泛，比如树莓派烧写镜像，然后以管理员权限运行

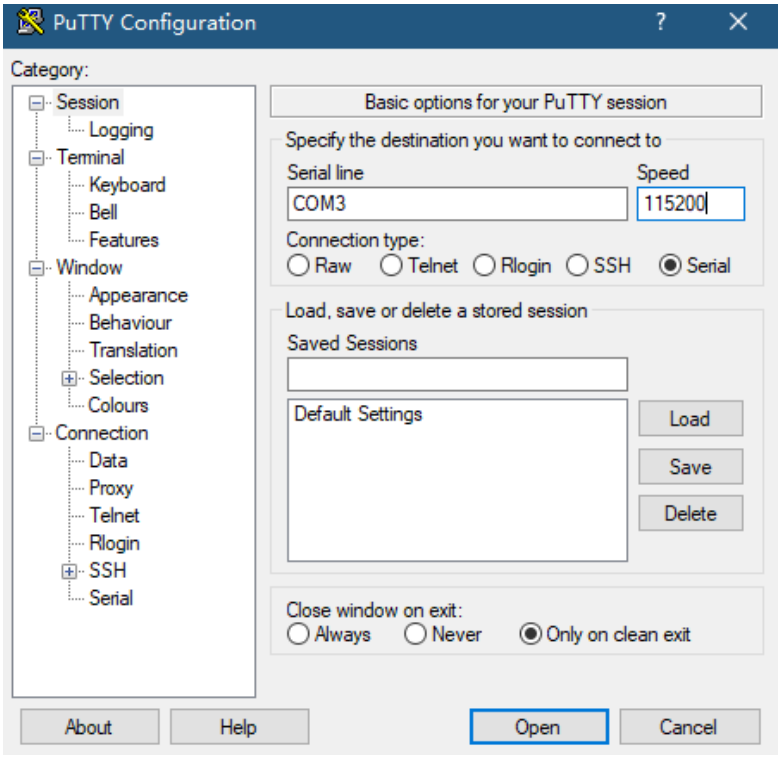


4. 选择镜像文件，再选择设备，注意要选 sd 卡所在盘符，然后点击写入

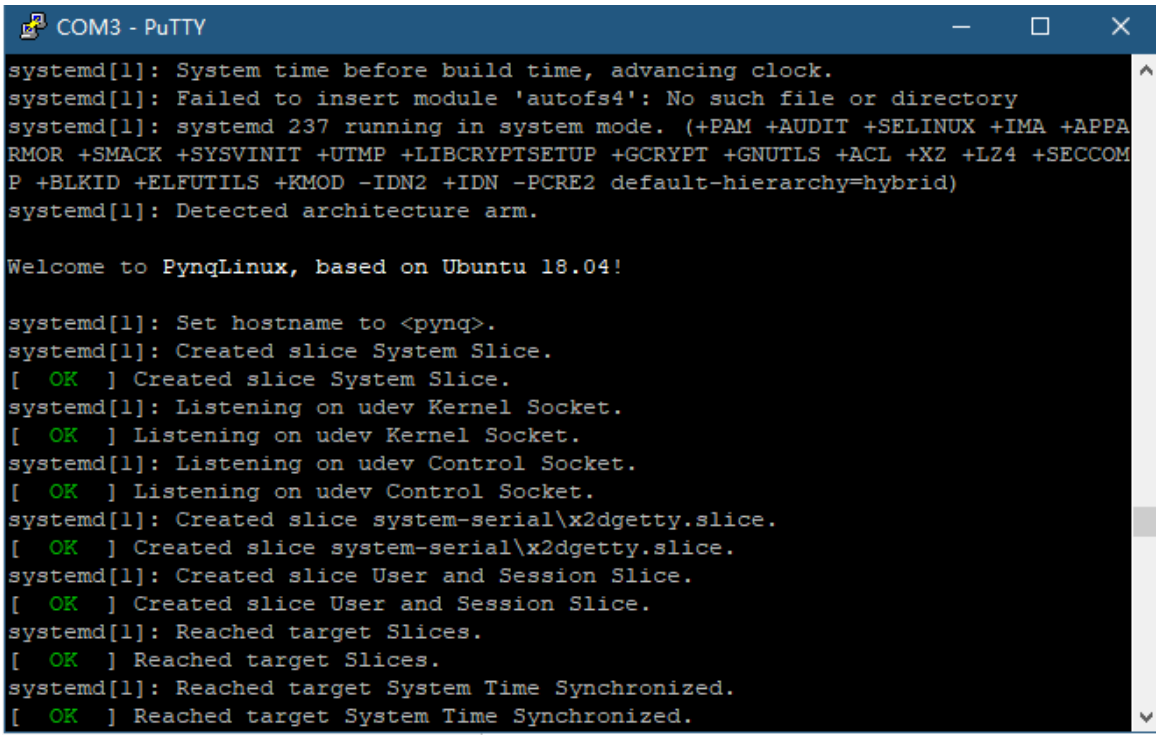
5. 写入完成以后的 sd 卡，被分为 2 个分区，第一分区为 FAT32，可以在 windows 下看到，第二分区为 EXT4，Windows 下不可见。

2.2 运行 PYNQ

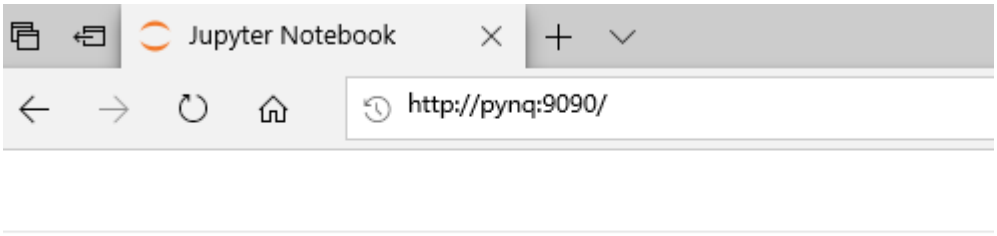
1. sd 卡插回开发板，确认开发板启动模式为 sd 卡模式
2. 连接开发板网口到路由器（最好支持 DHCP），如果路由器支持 DHCP，开发板上电后会自动获取 IP 地址，并可以用主机名 pynq 来访问，如果不支持 DHCP，开发板会启动静态 IP 地址 192.168.2.99
3. 连接开发板的 UART 口到电脑的 USB 口
4. 使用 Putty 为串口调试工具，使用方法参考基础教程



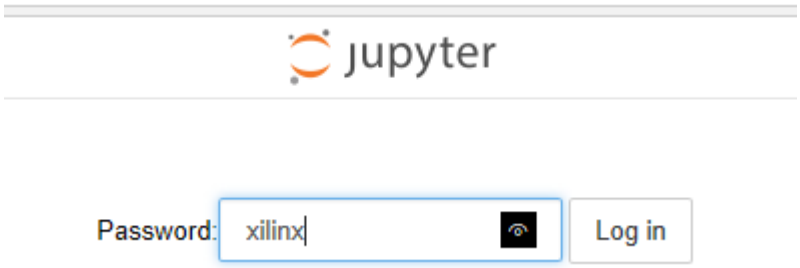
5. 上电运行，从打印信息可以看出跟文件系统是 Ubuntu 18.04



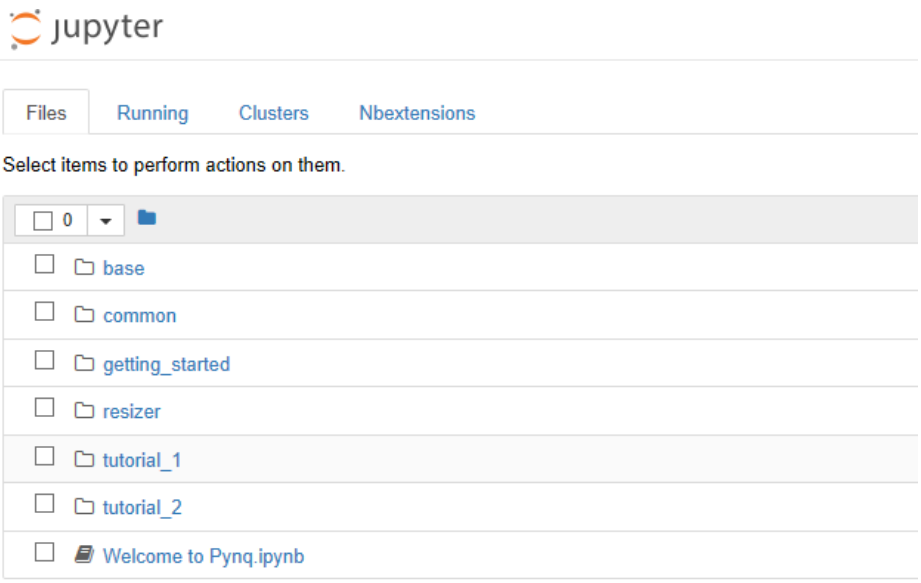
6. 如果路由器支持 DHCP，在浏览器里输入 http://pynq:9090,如果路由器不支持 DHCP，先修改电脑 IP 到 192.168.2.x 网段，然后浏览器输入 http://192.168.2.99:9090



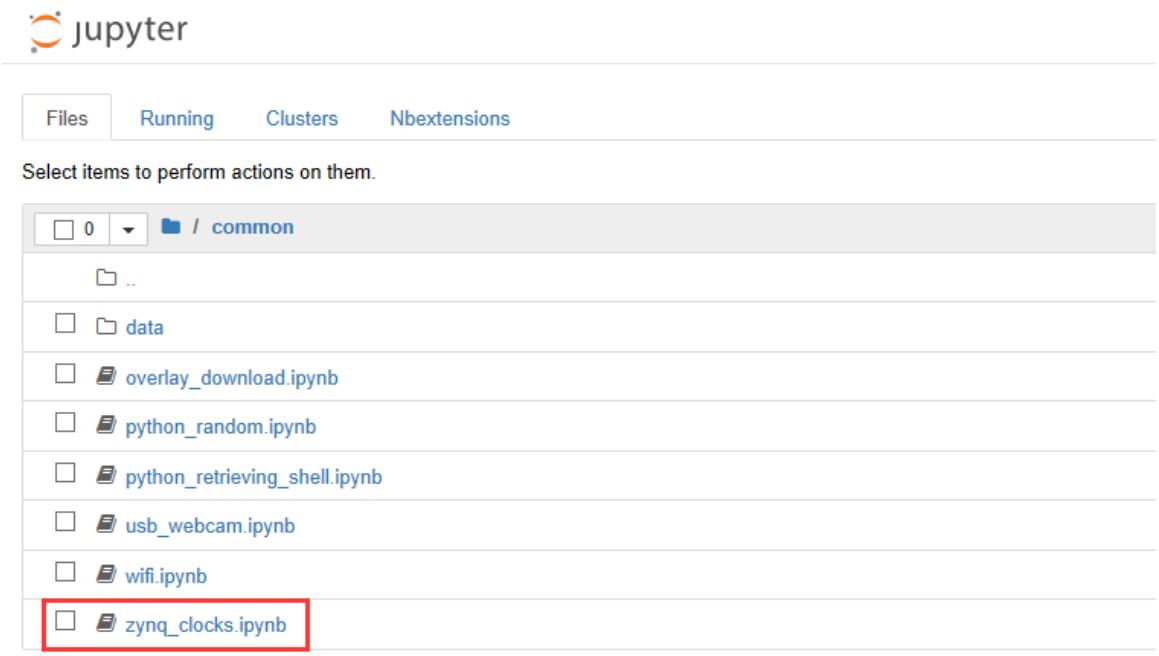
7. 输入登录密码：xilinx，点击”Log in”



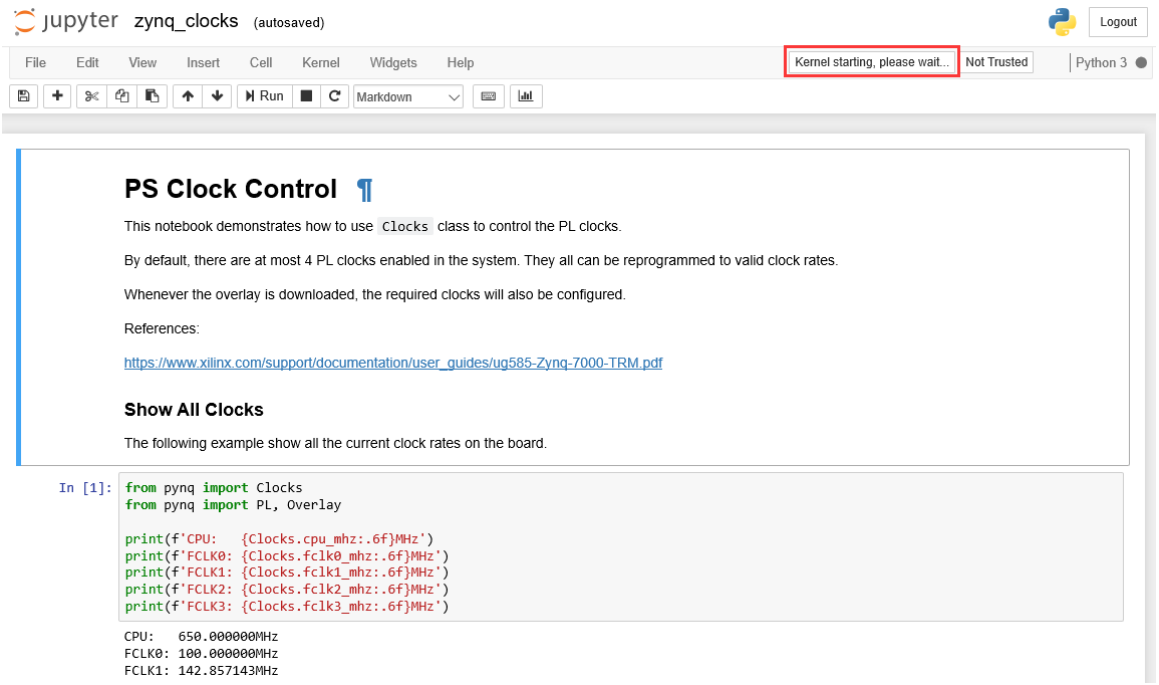
8. 可以看到一个类似文件管理器的界面



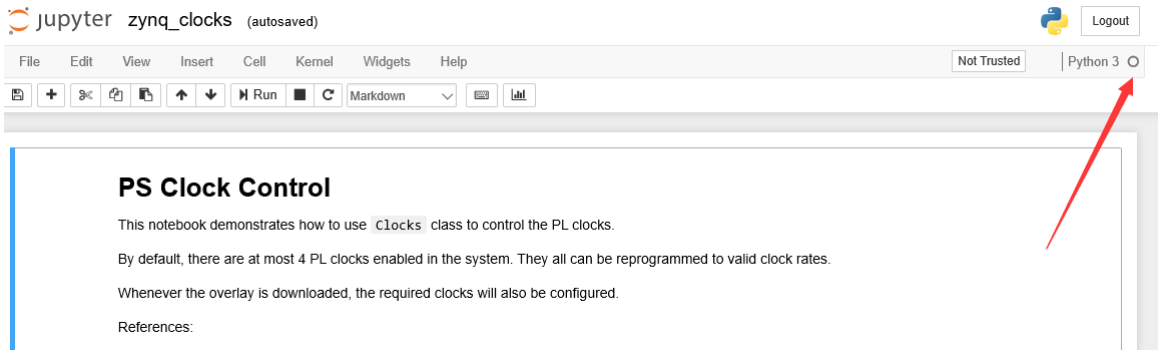
9. 先找到一个通俗易懂的例子运行一下，在 common 文件夹，点击“zynq_clocks.ipynb”



10. 打开后会有一个状态提示“Kernel starting，please wait…”，这个时候还不能点击“Run”

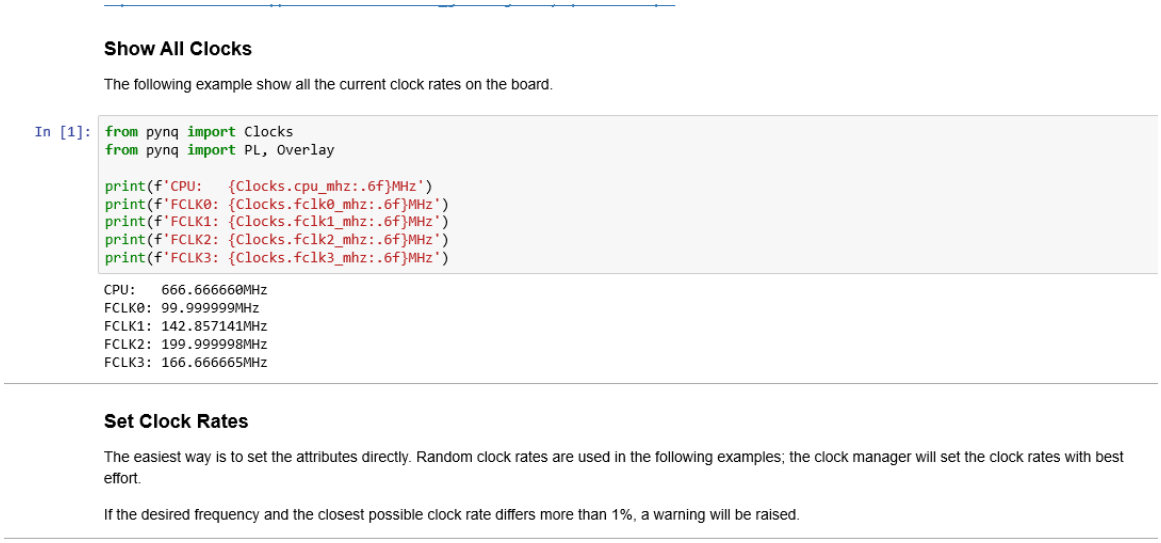


11. 等状态变为“Kernel idle”是可以点击“Run”

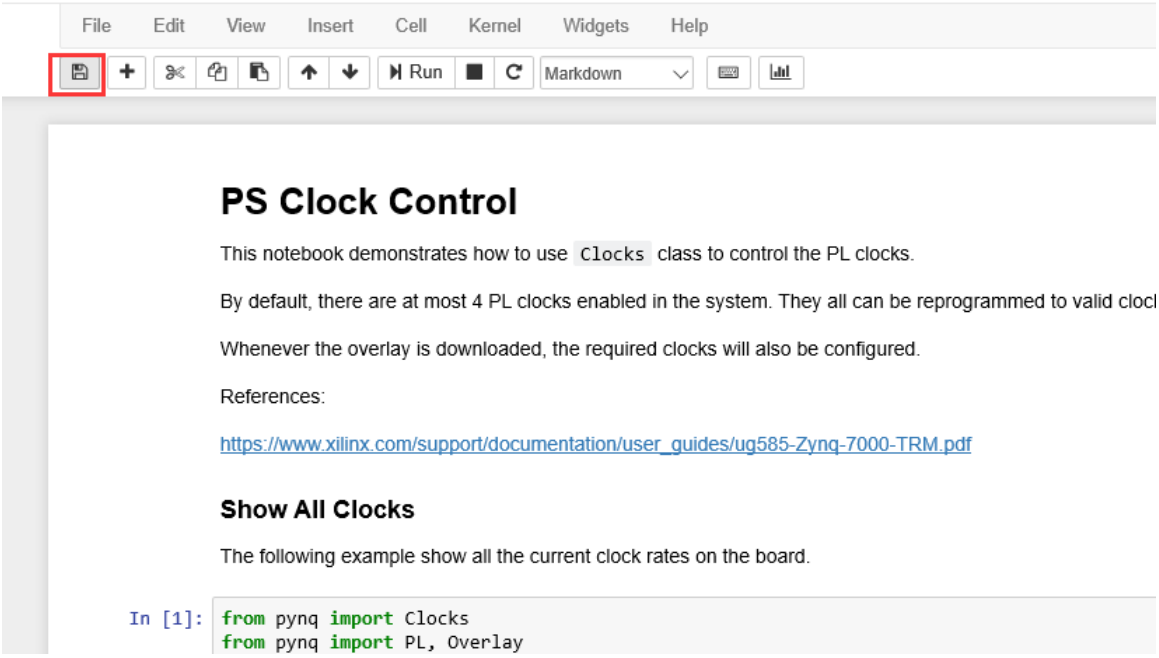


12. 用鼠标可以选择称为“cell”的运行单元，运行单元有 4 类，常用类为 Markdown 和 Code，顾名思义，Code 就是代码，Markdown 是文本，相当于注释，虽然“cell“都可以运行，但只有标记为”Code“的才有意义

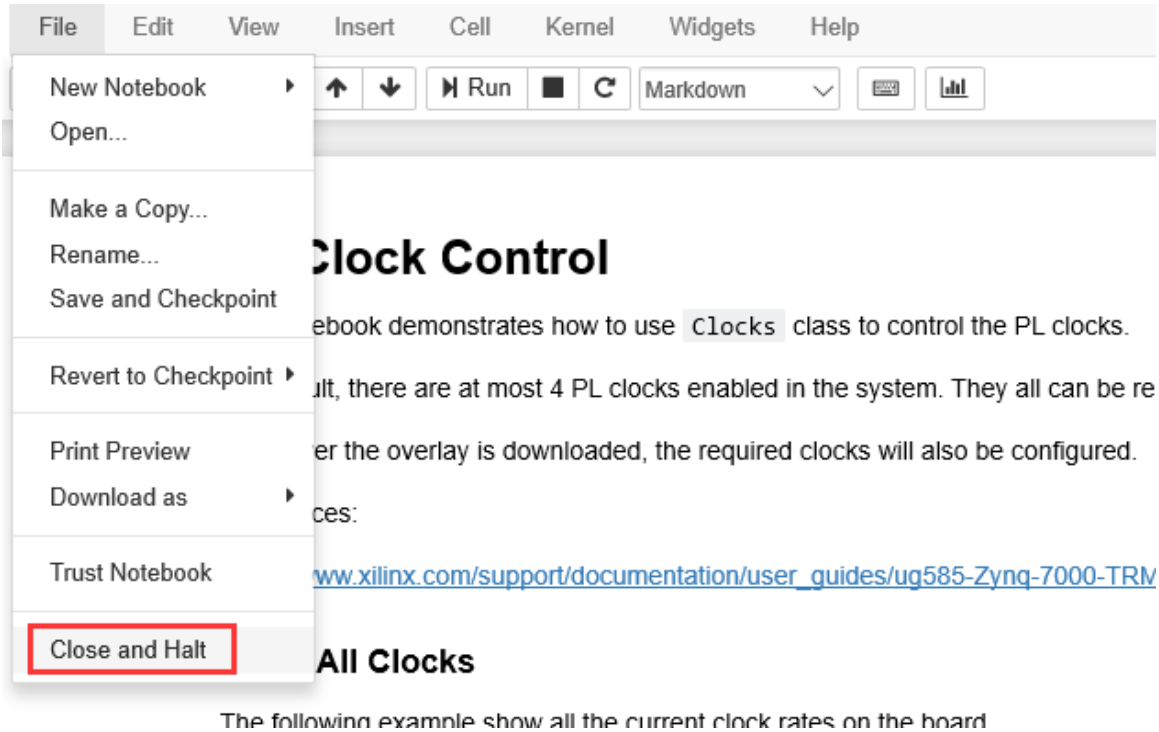
13. 一般我们从第一个 cell 开始运行，我们运行第一段代码后，在 cell 下面会给出打印结果，结果显示 cpu 主频为 666.666666Mhz。**特别要注意：每次点击 Run 后请等待状态变为“Kernel ilde“，再运行下一个 cell。**



14. 运行结束后，可以点击保存，当前程序和状态都保存起来



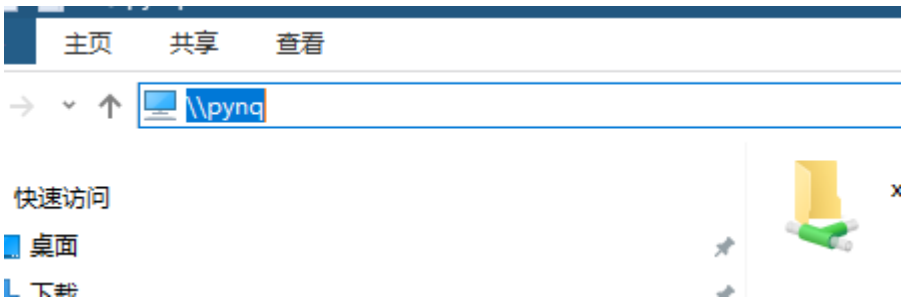
15. 最后可以点击 File –> Close and Halt 退出程序



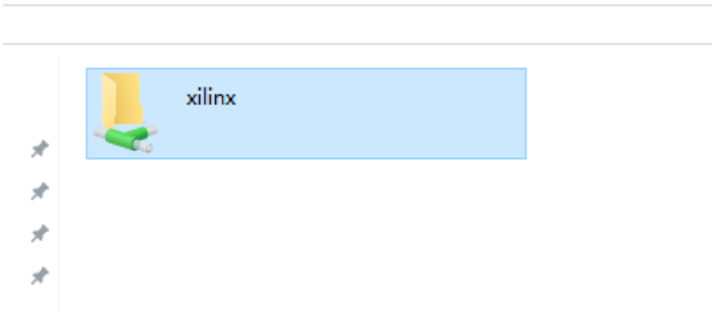
2.3 samba 文件共享

PYNQ 提供了了 samba 共享文件夹，方便上传和下载文件，使用方法如下：

1. 路由器支持 DHCP 时在资源管理器里输入\\pynq，不支持的输入\\192.168.2.99



2. 双击“xilinx“文件夹



3. 用户名和密码都输入“xilinx”

4. 可以看到开发板上的文件夹和文件

2.4 常见问题

2.4.1 使用 http://pynq:9090 无法访问

有些系统无法通过 hostname 访问，这个使用我们要用 ip 地址直接访问，在串口终端中输入命令 ifconfig，eth0 的 ip 地址就是 DHCP 分配到 ip 地址，如果没有地址，表明 DHCP 分配失败。在浏览器输入 http://192.168.1.212:9090 代替 http://pynq:9090

```
xilinx@pynq:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.212  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::2c85:45ff:fe36:320e  prefixlen 64  scopeid 0x20<link>
        ether 2e:85:45:36:32:0e  txqueuelen 1000  (Ethernet)
        RX packets 16397  bytes 1772321 (1.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9839  bytes 10055997 (10.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 29  base 0xb000

eth0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.99  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 2e:85:45:36:32:0e  txqueuelen 1000  (Ethernet)
        device interrupt 29  base 0xb000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1317  bytes 163650 (163.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1317  bytes 163650 (163.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

第三章 USB 摄像头边沿检测和人脸识别

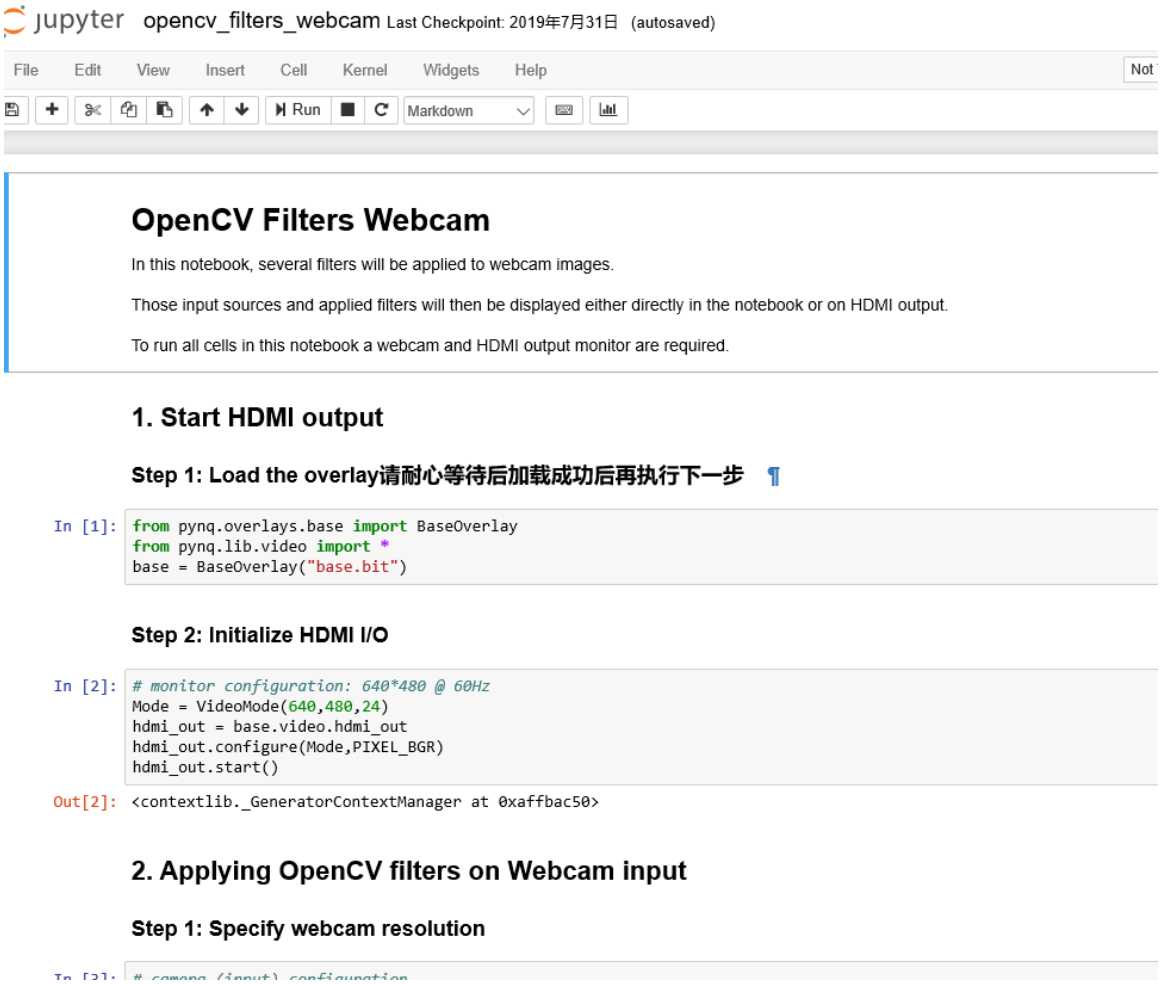
在 pynq 镜像里给出了很多自带例程，比较有代表的就是边沿检测和人脸识别，比较遗憾的是，这 2 个例程都没有使用 FPGA 做加速，是纯软件计算的，但是使用了 FPGA 的 HDMI 输出显示结果。这些例程都是 Xilinx 官方提供，芯驿电子做了一些本地化工作。

3.1 边沿检测

和前面一章一样，先登录 Jupyter Notebooks，然后在目录 base/video 中找到 opencv_filters_webcam.ipynb 这个 Notebook。

在运行这个例程前，先把 USB 摄像头（免驱型 USB 摄像头）。

在 Notebooks 强大注释功能下本教程显得特别苍白无力，每一步都可以有丰富的提示。Python 语言本身也不是本教程的重点，如果对 Python 语言不熟悉，先在电脑上加强练习。



需要解释有：如果程序中不使用 FPGA 资源，可以不加载 FPGA 的 bit 文件，如果使用 FPGA 资源，必须先加载 FPGA 的 bit 文件，本例程中 HDMI 显示是 FPGA 做的，必须加载 bit 文件。

通过本例程，我们可以看到支持 Python 常用的库，例如 opencv 等，如果需要安装其他库，可以 pip 命令安装。

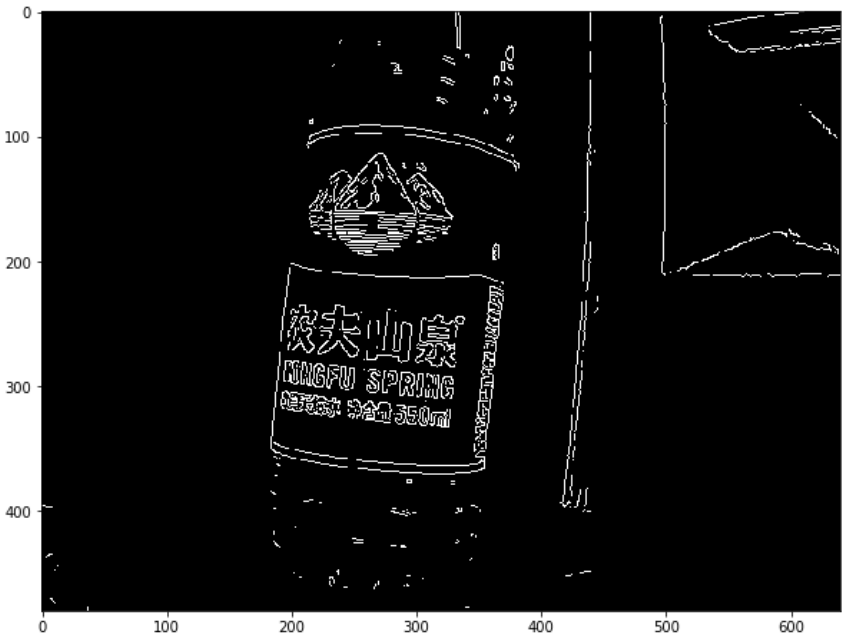
通过实验，可以看到边沿检测结果能直接在浏览器中显示出来

Step 6: Show results

Now use matplotlib to show filtered webcam input inside notebook.

```
: %matplotlib inline
from matplotlib import pyplot as plt
import numpy as np

frame_canny = cv2.Canny(frame_webcam, 100, 110)
plt.figure(1, figsize=(10, 10))
frame_vga = np.zeros((480,640,3)).astype(np.uint8)
frame_vga[:,:,0] = frame_canny
frame_vga[:,:,1] = frame_canny
frame_vga[:,:,2] = frame_canny
plt.imshow(frame_vga)
plt.show()
```



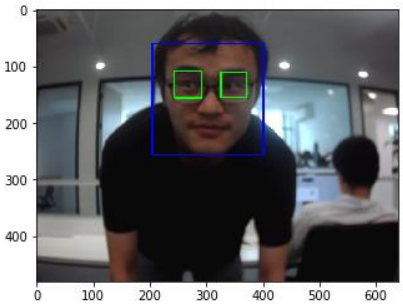
需要注意的是：涉及到 FPGA 的程序，每次都要完整运行，如果程序运行一半退出，可能导致下次运行异常，需要开发板断电重启。

3.2 人脸检测

人脸检测例程是 opencv_face_detect_webcam.ipynb 这个 Notebook。人脸检测程序主要检测图像中有没有人脸，然后框出人脸和眼睛，是直接调用 opencv 进行检测

Step 7: Now use matplotlib to show image inside notebook

In [9]:
```
# Output OpenCV results via matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import numpy as np
plt.imshow(np_frame[:,:,[2,1,0]])
plt.show()
```



Step 8: Release camera and HDMI

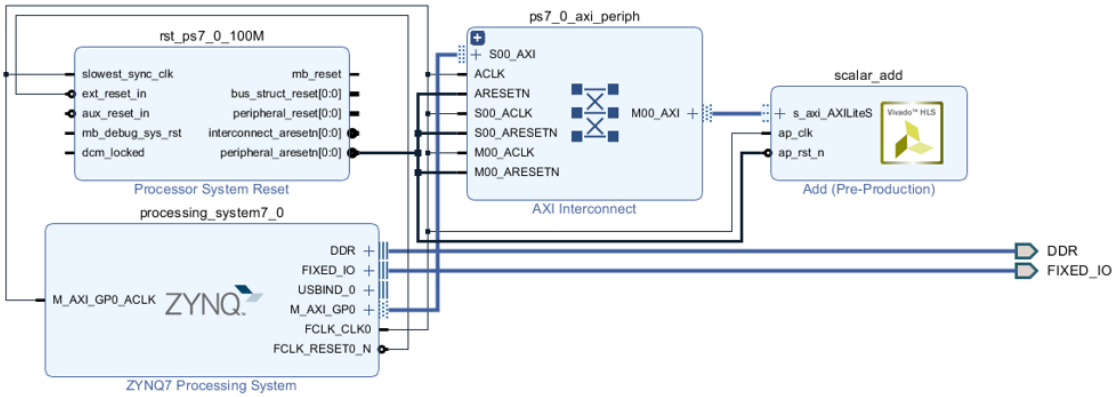
In [10]:
```
videoIn.release()
hdmi_out.stop()
del hdmi_out
```

第四章 基于寄存器操作自定义 Overlays

本章内容讲解如何把通用的 vivado 工程导入到 PYNQ，这是大家学习 PYNQ 梦寐以求的事情。

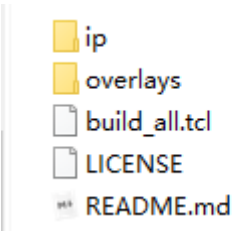
4.1 vivado 工程

如下图所示，vivado 工程里核心模块是一个自定义 IP，使用 HLS 编写，主要功能是完成 c = a + b，前面已经说过，xilinx 只提供了 python 寄存器、dma、vdma 底层封装，也就是说不是所有 vivado 工程都能直接导入到 PYNQ，只能是一些寄存器读写，dma 操作的工程。



hls 和 vivado 工程建立不是本教程关注的重点，如何使用 hls 和 vivado 可以参考其他教程，这里做个简单介绍。

在给的资料 overlay_tutorial_17_4 目录中有一下内容，ip 为 hls 编写的 ip 源码，overlays 为 overlays 的 vivado 工程和 tcl 文件。build_all.tcl 脚本为 hls 编译、vivado 工程编译脚本。



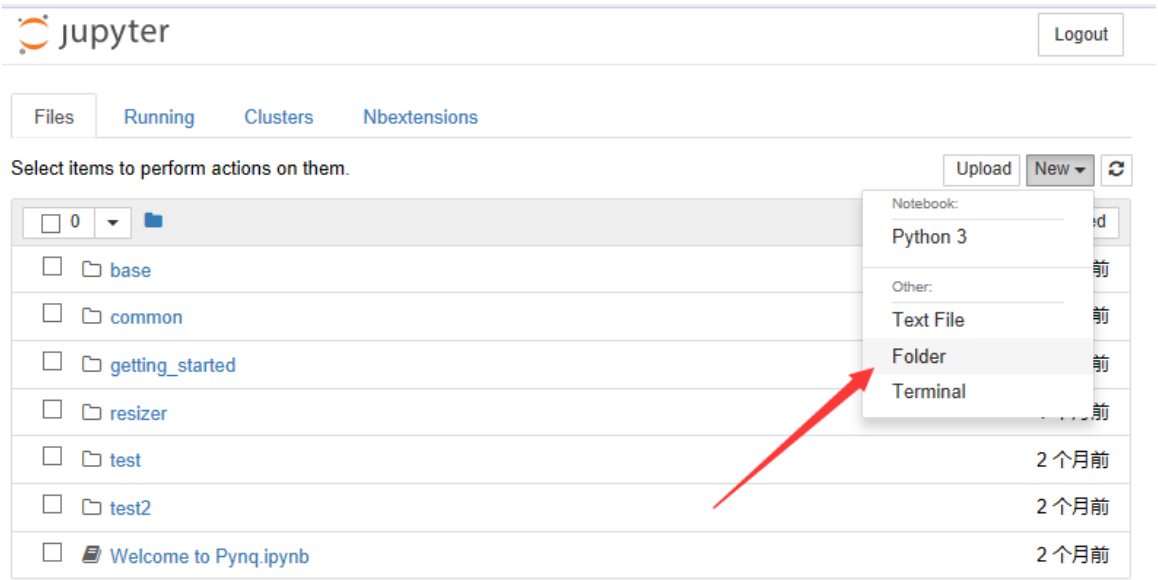
给的例程中已经编译完成，生成了 tutorial_1、tutorial_2、tutorial_1.bit、tutorial_1.hwh、tutorial_2、tutorial_2.bit、tutorial_2.hwh，如果要再次编译，需要先删除这些文件。tutorial_1.bit、tutorial_1.hwh 是本教程需要用到的 2 个文件、tutorial_2.bit、tutorial_2.hwh 是后续教程需要用到的文件。

对于 bit 文件我们已经非常熟悉了，hwh 文件前面也有概述，这里再说明一下，hwh 文件是描述硬件连接信息的 xml 文档，通过 hwh 文件可以获取到硬件设计中的 IP 信息、寄存器信息、中断信息，PYNQ 框架通过解析 hwh 文件自动匹配相关的 IP 驱动、自动获取 IP 的地址信息等。

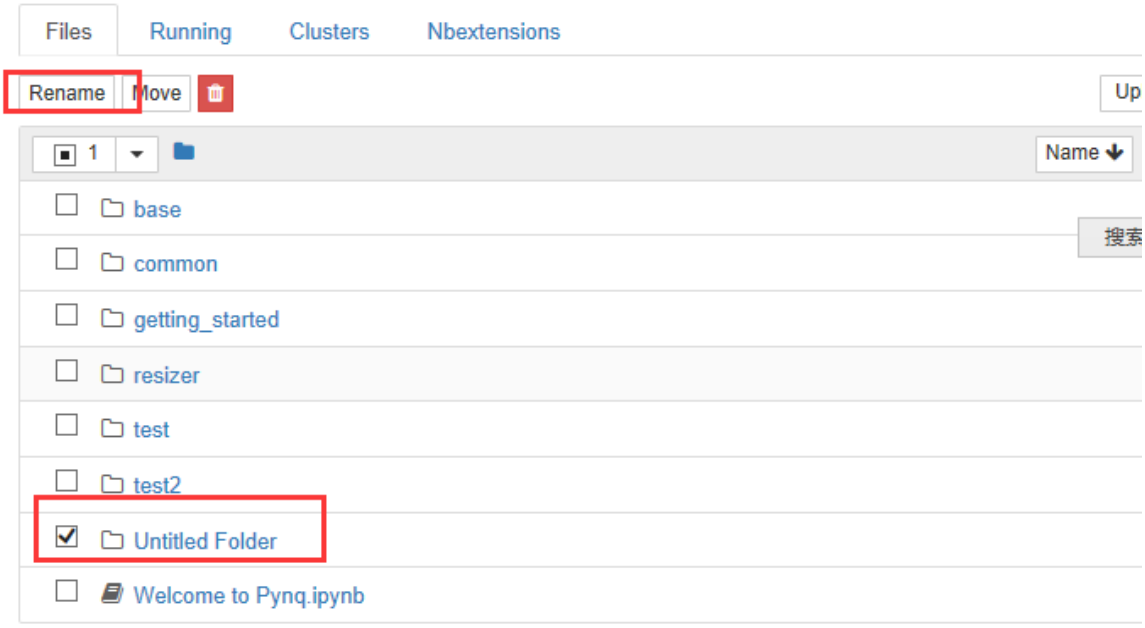
4.2 建立 botebook

1. 使用浏览器登录 PYNQ notebook

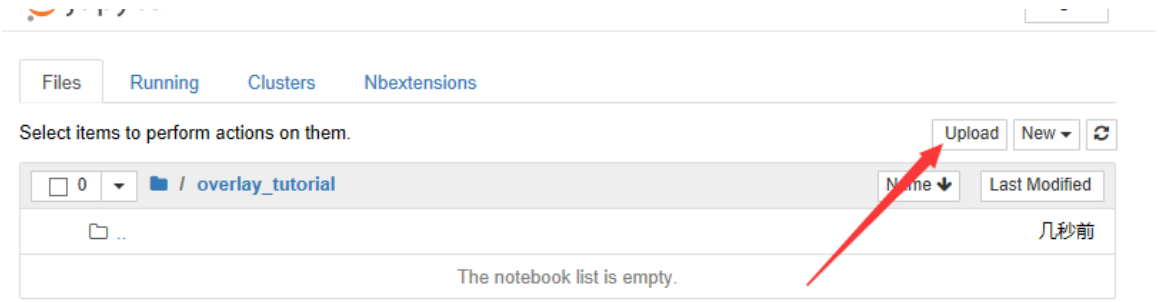
2. 使用“New -> Folder”创建一个新文件夹



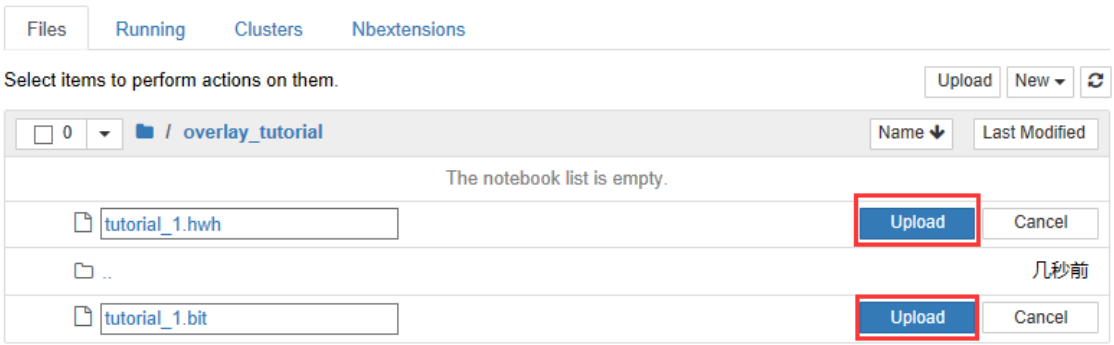
3. 重新命名为“overlay_tutorial”



4. 进入目录“overlay_tutorial”，点击“Upload”上传文件

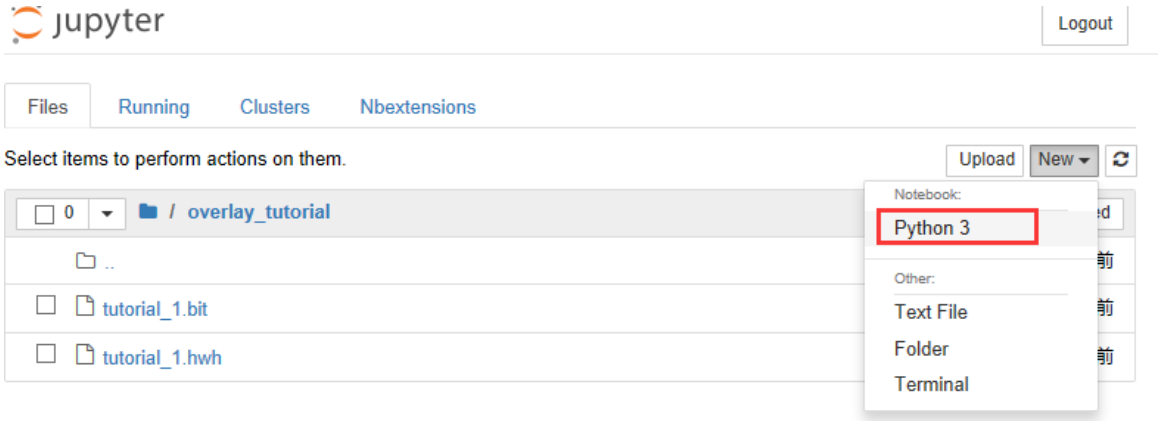


5. 把 tutorial_1.bit、tutorial_1.hwh 都选择上传，然后点击“Upload”

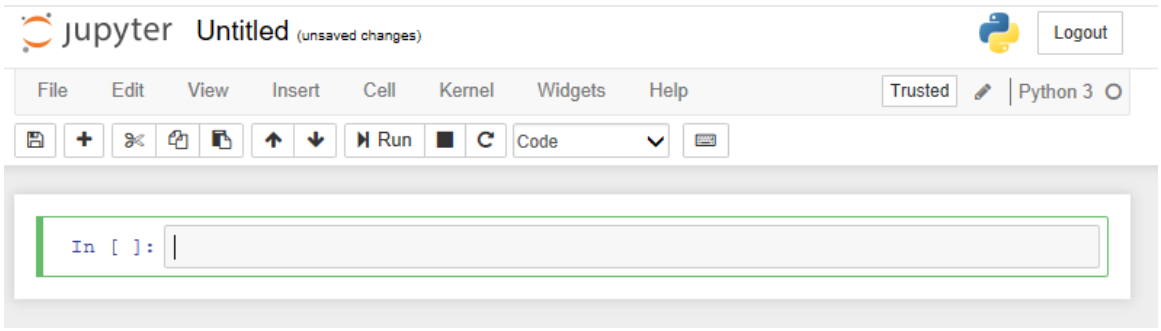


6. 上传完成

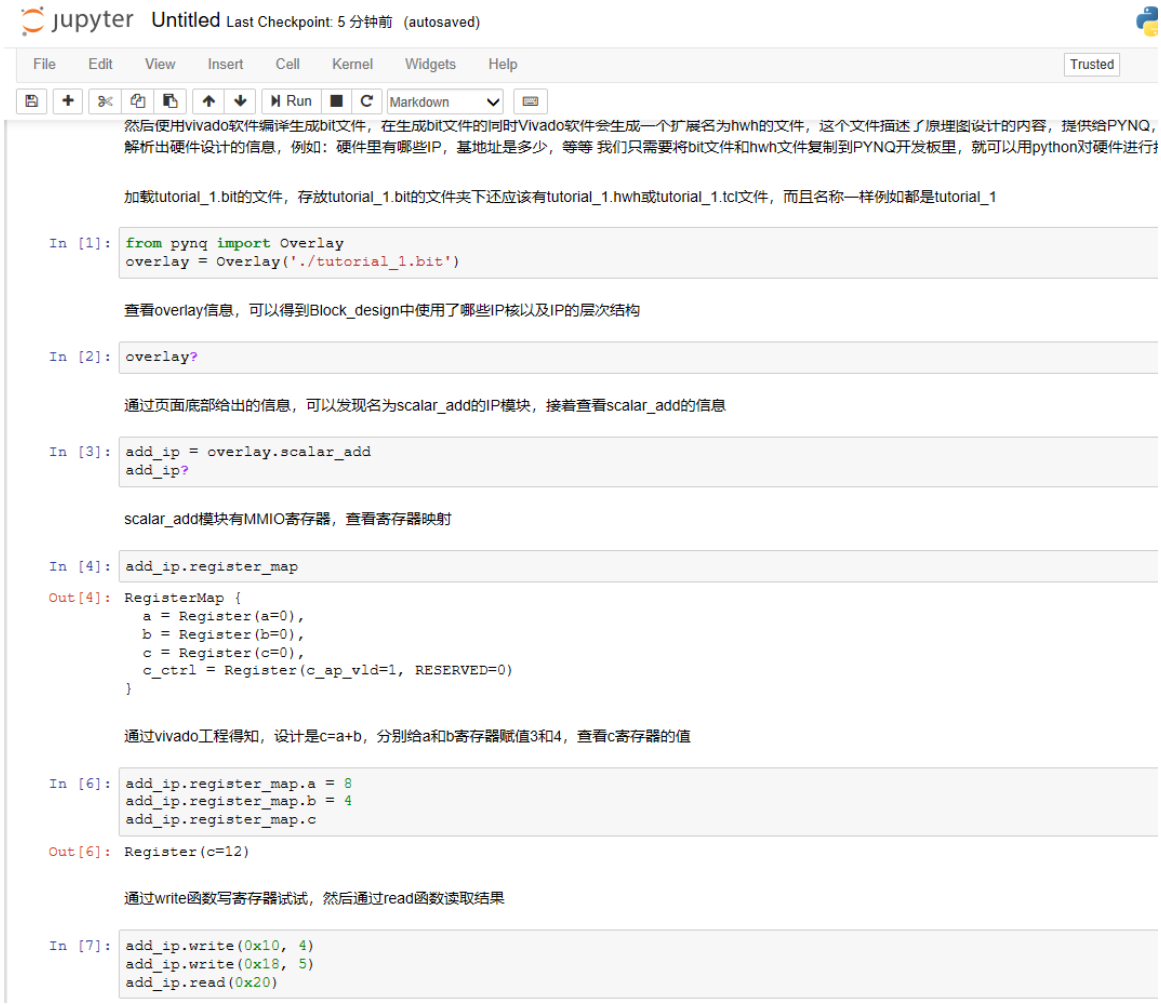
7. 新建一个 Notebook



8. 弹出 notebook 编辑界面，默认是一个 Code 类型的 cell，

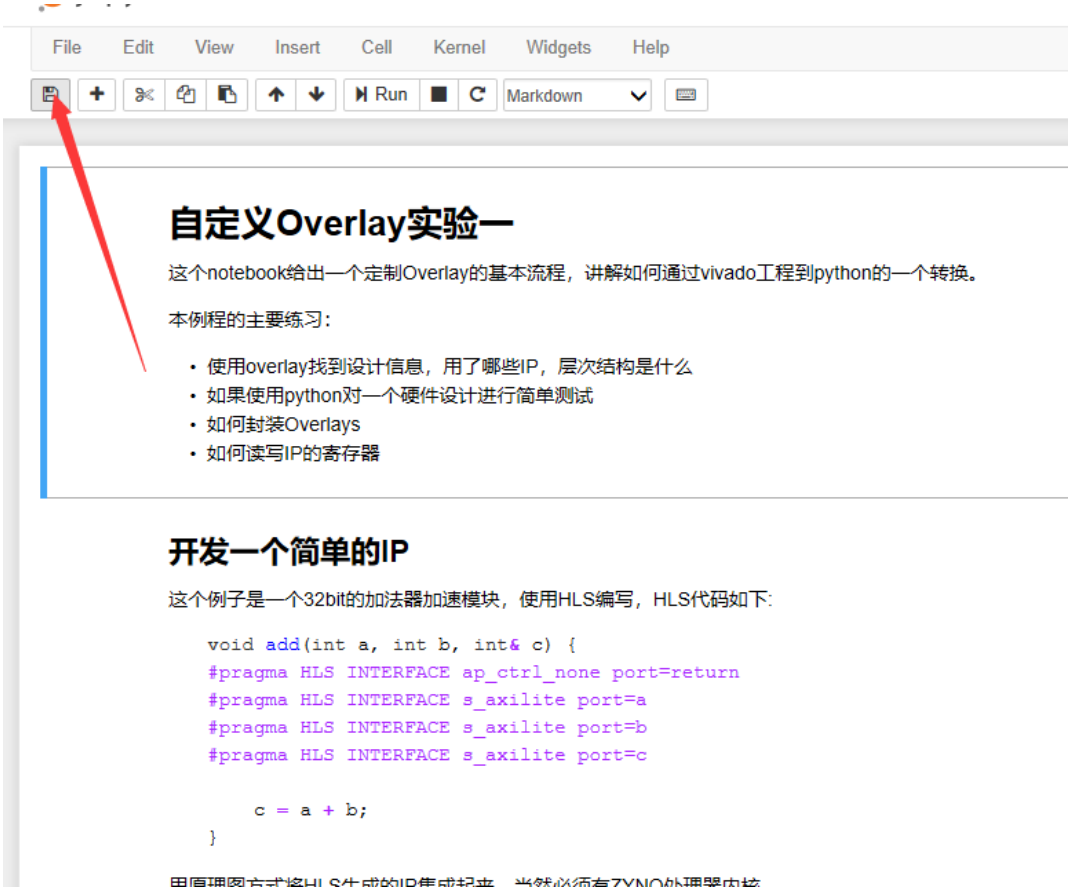


9. 开始编辑 Notebook，编辑过程比较简单，类似于 word 的使用，编辑过程不是教程重点

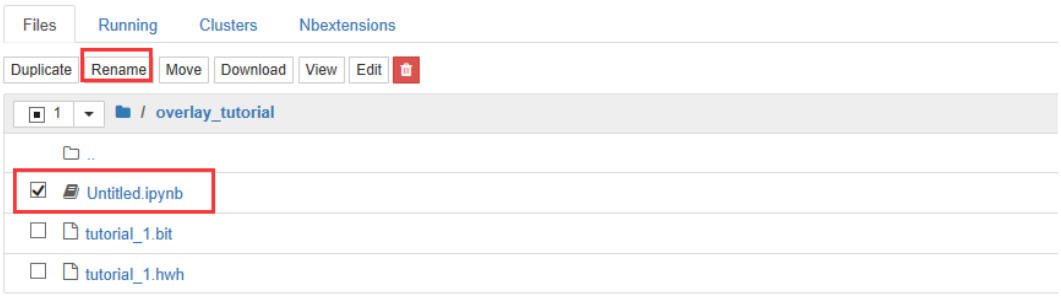


10. 由于 notebook 里有详细的注释、这部分 python 程序不再讲解

11. 最后不要忘了点击保存



12. 为了方便记忆，将刚才 botebook 修改为” tutorial_1.ipynb”



13. 通过 samba 访问 \\192.168.x.xxx\xilinx\jupyter_notebooks\overlay_tutorial，可以找到我们上传和新建的文件，samba 访问前面已经讲过，账号为 xilinx、密码为 xilinx。

第五章 含有 DMA 操作自定义 Overlays

通过前一章我们初步掌握了如何通过自定义的 vivado 工程来做 overlay，上一个例子中只有寄存器读写操作，大批量高速数据交互时往往会使用 DMA，所以含有 DMA 的 Overlay 就显得非常重要，Python 中 DMA 驱动是 Xilinx 官方提供，目前只支持 sample AXI DMA，不支持 SG 模式，也不支持中断。

本实验同样是一个基于 HLS 编写的流乘法器，就是一个数组都乘一个常数、得到另一个数组

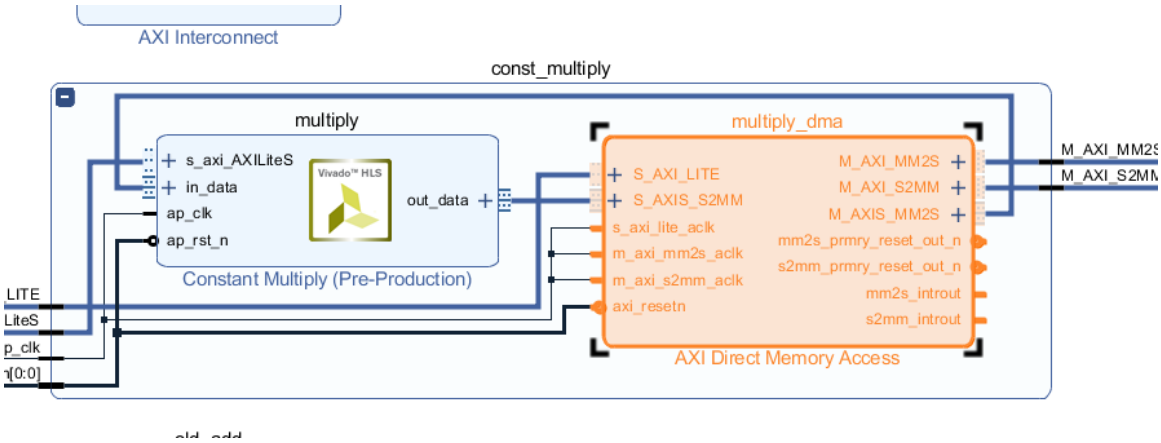
```
#include "ap_axi_sdata.h"
#include "ap_int.h"

typedef ap_axiu<32,1,1,1> stream_type;

void mult_constant(stream_type* in_data, stream_type* out_data, ap_int<32> constant) {
#pragma HLS INTERFACE s_axilite register port=constant
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=in_data
#pragma HLS INTERFACE axis port=out_data
    out_data->data = in_data->data * constant;
    out_data->dest = in_data->dest;
    out_data->id = in_data->id;
    out_data->keep = in_data->keep;
    out_data->last = in_data->last;
    out_data->strb = in_data->strb;
    out_data->user = in_data->user;

}
```

tutorial_2 就是本实验的 vivado 工程，我们可以看到 multiply 和 multiply_dma 组合成一个 const_multiply，这相当于一个子模块，在 PYNQ 中引用时要分出层次结构



就像代码中 dma = overlay.const_multiply.multiply_dma，先是引用 const_multiply，再引用 multiply_dma

```
: import pynq.lib.dma

  overlay = Overlay('./tutorial_2.bit')

  dma = overlay.const_multiply.multiply_dma
  multiply = overlay.const_multiply.multiply
```

5.1 DMA 如何操作

结合代码说明一下 DMA 操作

1. 首先导入 Xlnk 库，这个库里包含了 Xilinx 的 DMA 管理接口，
2. 使用 cma_array 分配 DMA 缓冲区
3. 使用 sendchannel.transfer 发起一个 DMA 发送操作
4. 使用 recvchannel.transfer 发起一个 DMA 接受操作
5. sendchannel.wait()等待发送完成
6. recvchannel.wait()等待接收完成

```
from pynq import Xlnk
import numpy as np

xlnk = Xlnk()
in_buffer = xlnk.cma_array(shape=(5,), dtype=np.uint32)
out_buffer = xlnk.cma_array(shape=(5,), dtype=np.uint32)

for i in range(5):
    in_buffer[i] = i

multiply.constant = 3
dma.sendchannel.transfer(in_buffer)
dma.recvchannel.transfer(out_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()

out_buffer
```

第六章 ADC 采集并显示波形

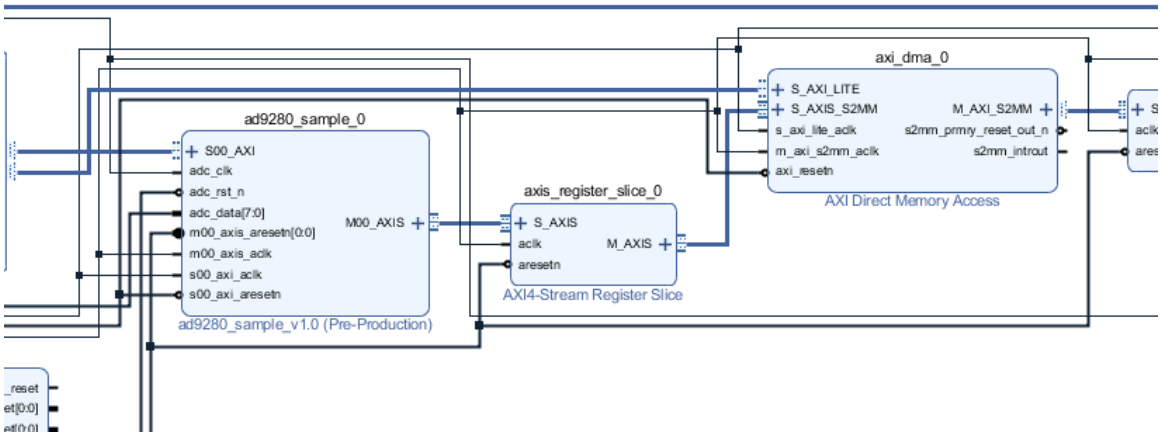
在前面的实验中我们学习了使用 PYNQ 操作 dma，但是不太直观，本实验通过采集 ADC 数据并显示出波形，用更直观的实例来说明 dma 的使用。

6.1 ADC 硬件

实验中使用模块为 AN108，一路 32M 采样率 ADC、一路 125M 采样率 DAC，实验中使用了其中的 ADC 通道。在 SDK 例程中我们已经使用过 AN108 模块，这里不做说明了。

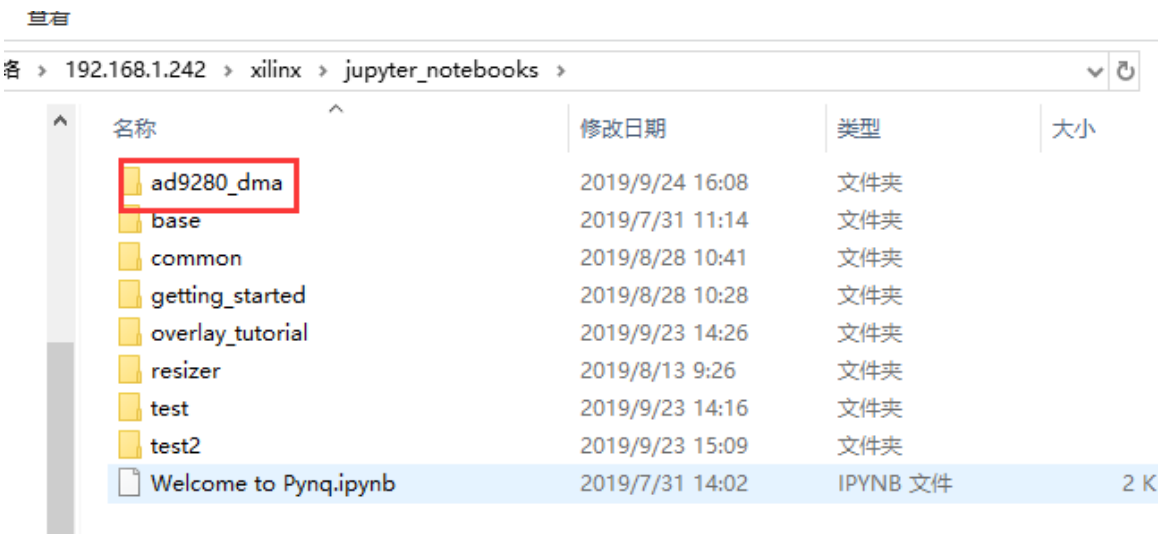
6.2 Vivado 工程

Vivado 工程来自 course_s2 的 ad9280_dma_hdmi，这里删除了 HDMI 显示部分，删除了 DMA 的中断连接，这里不再讲解这个 Vivado 工程。

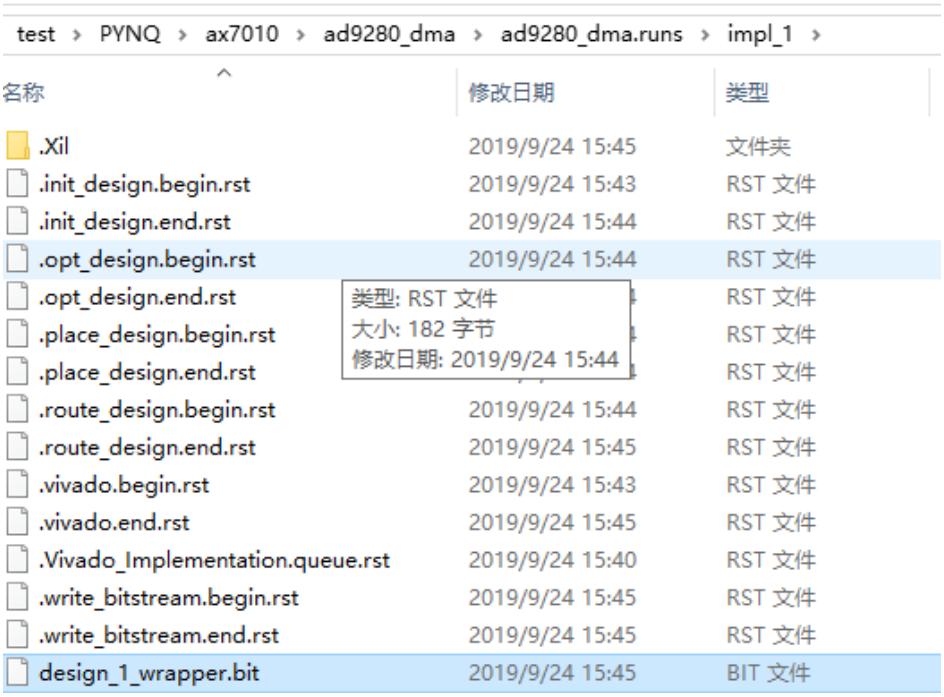


6.3 Notebook 文件

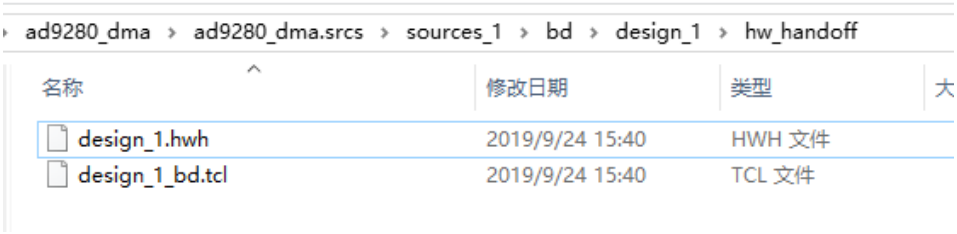
1. 通过 samba 访问 \\192.168.x.xxx\xilinx\jupyter_notebooks，新建一个文件夹 ad9280_dma 用于存储 bit 文件、hwh 文件和 jupyter_notebook 文件



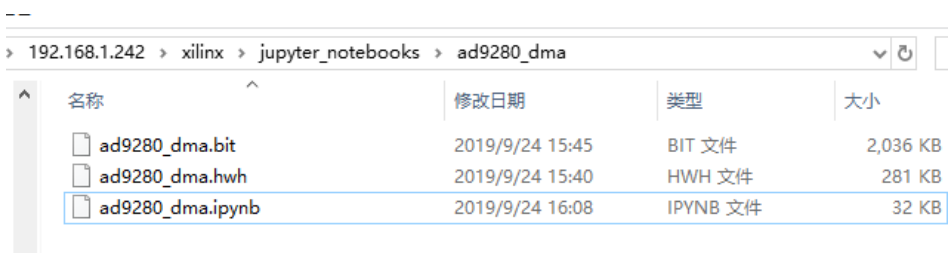
2. 将 design_1_wrapper.bit 文件复制到 \\192.168.x.xxx\xilinx\jupyter_notebooks\ad9280_dma，并重命名为 ad9280_dma.bit



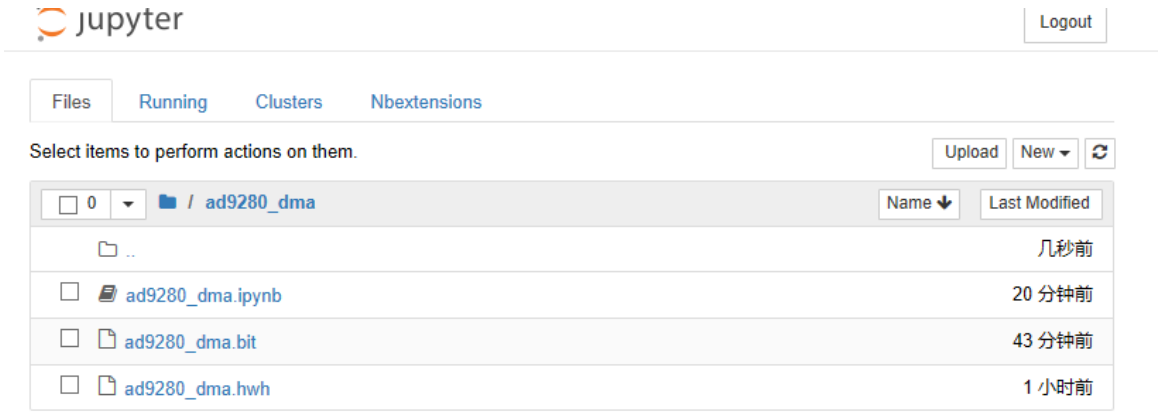
3. 将 design_1.hwh 复制到\\192.168.x.xxx\xilinx\jupyter_notebooks\ad9280_dma 并重命名为 ad9280_dma.hwh



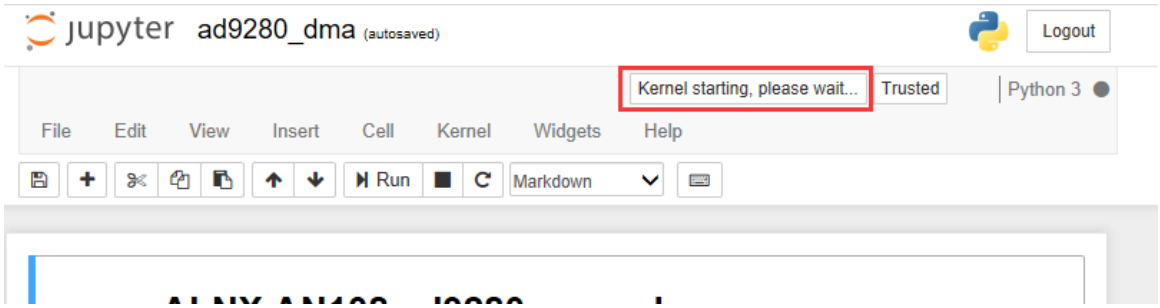
4. 如果不想自己再编辑 notebook，把 ad9280_dma.ipynb 文件复制到 \\192.168.x.xxx\xilinx\jupyter_notebooks\ad9280_dma



5. 浏览器登录打开 notebook



6. 等 Kernel start 完成才能运行



7. 100Khz 正弦波，运行结果

```
In [14]: #获取AD采集到的数据并显示
dma.recvchannel.transfer(adc_buffer)
adc_sample.write(0x04, 1024) #sample 1024 points
adc_sample.write(0x00, 0x01) #start sample
dma.recvchannel.wait()
#画图
fig1 = plt.figure()
ax1 = fig1.gca()
plt.plot(adc_buffer)
plt.cla
```

```
Out[14]: <function matplotlib.pyplot.cla>
```