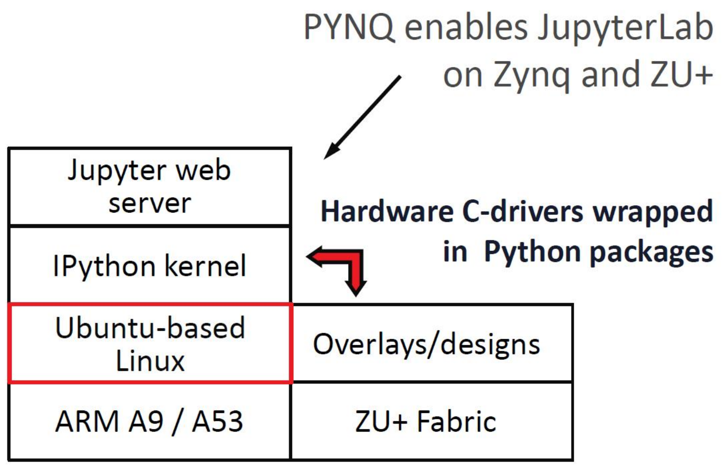
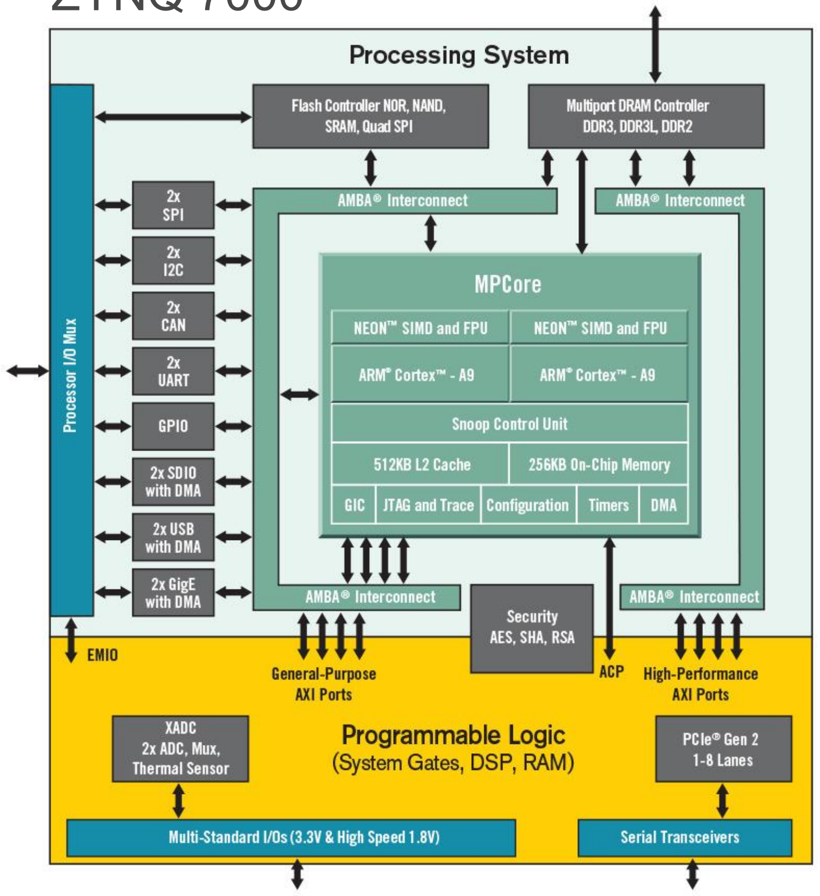
Tutorial: Acceleration with PYNQ and Vitis HLS

Basics: ZYNQ and PYN

- ZYNQ and ZYNQ UltraSCALE+
  - FPGA + tightly-integrated CPUs (Arm)
  - SoC (System on Chip)
- PYNQ: Python Productivity for ZYN
  - Python makes ZYNQ easy-to-use
  - framework

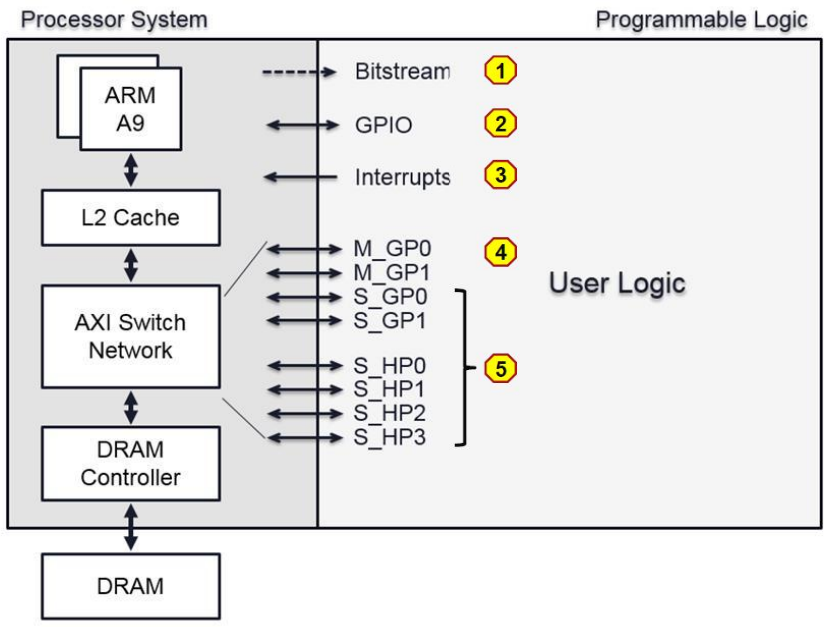


ZYNQ 7000

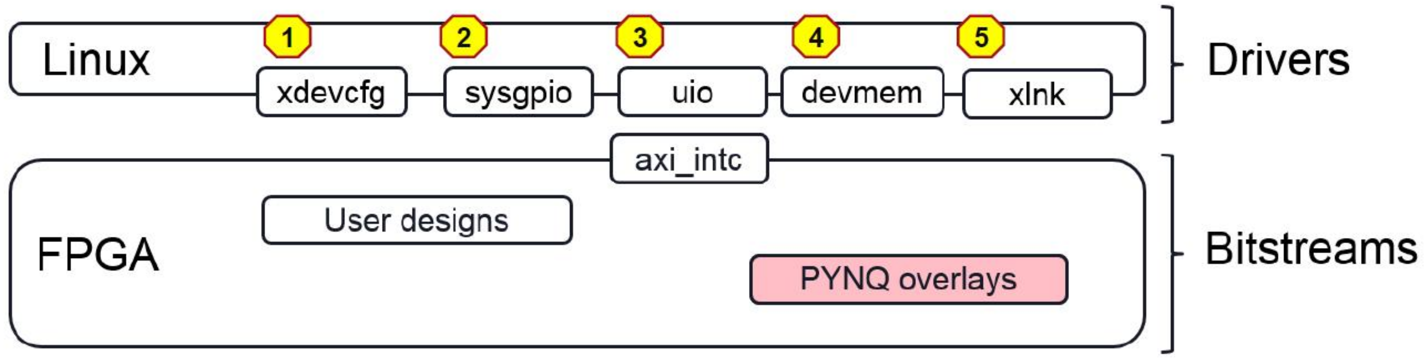


PS-PL Interfaces

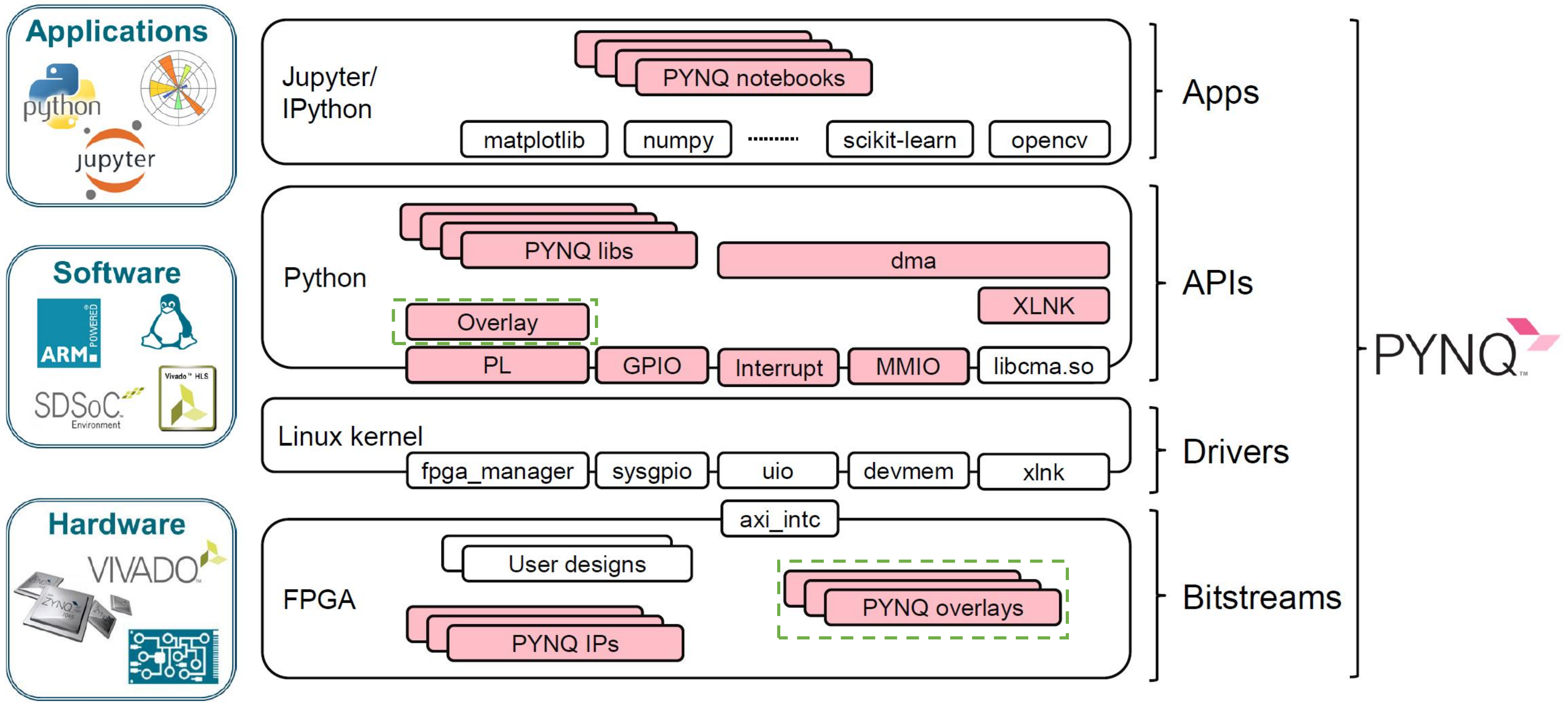
- ZYNQ PS-PL model
  - PS: Processor System (ARM)
  - PL: Programming Logic (FPGA)
  - reference: [PS/PL Interfaces]



- PYNQ provides Linux drivers for PS-PL interfaces



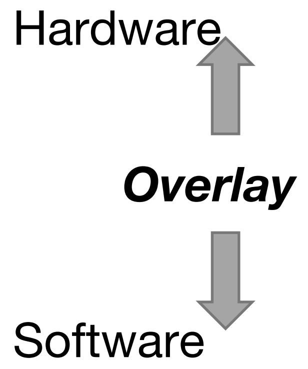
PYNQ Framework



Overlays are a bridging abstraction, which wraps hardware libraries (*IP*s) and enable software developers to access them
For more information, vitis the PYNQ Docs: PYNQ Overlays

PYNQ works with Vivado

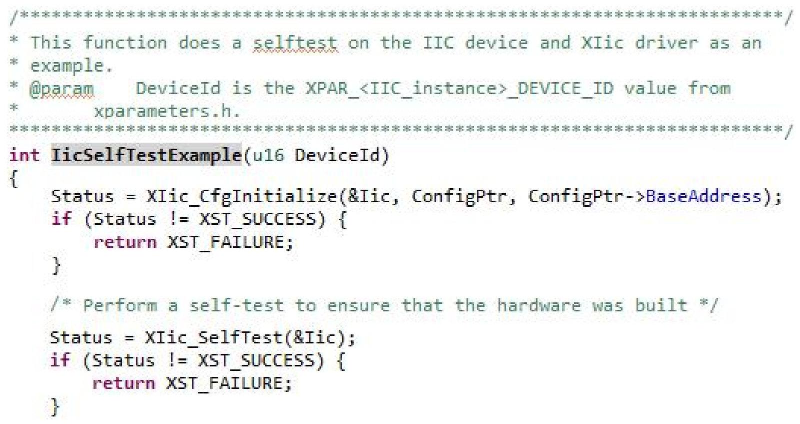
- PYNQ passes the Vivado metadata file to the target platform (ZYNQ)
  - IPs in the bitstream
  - can be exported from Vivado
- ZYNQ parses the Vivado metadate to:
  - set Zynq clock frequencies automatically
  - assign drivers for every IP
- PYNQ create a Python dictionary for the IPs in the bitstream
  - enables bitstream metadata to be queried and modified in Python at time



PYNQ Benefits

- PYNQ makes Zynq accessible to non-traditional customers
- PYNQ delivers open source benefits
  - github.com/Xilinx/PYNQ
- PYNQ enables highly-productive
  - prototyping \ debug \ evaluation
  - Typically, debug of IP uses C/C++ with SDK
    - PYNQ enables Python to execute on target.
    - More convenient!
- PYNQ documantation is good.
  - pynq.readthedocs.io
- “PYNQ makes FPGAs FUN again!”

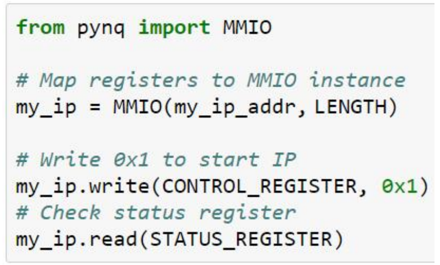
- Vitis pynq.io for more information!



```
/******************************************************************************/
* This function does a selftest on the IIC device and XIic driver as an
* example.
* @param    DeviceId is the XPAR_<IIC_instance>_DEVICE_ID value from
*           xparameters.h.
*******************************************************************************/
int IicSelfTestExample(u16 DeviceId)
{
    Status = XIic_CfgInitialize(&Iic, ConfigPtr, ConfigPtr->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* Perform a self-test to ensure that the hardware was built */

    Status = XIic_SelfTest(&Iic);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
```



```
from pynq import MMIO

# Map registers to MMIO instance
my_ip = MMIO(my_ip_addr, LENGTH)

# Write 0x1 to start IP
my_ip.write(CONTROL_REGISTER, 0x1)
# Check status register
my_ip.read(STATUS_REGISTER)
```

A Simple Example: Adder

- We’ll show the PYNQ acceleration flow with a simple adder kernel.

- The tutorial generally has the following four steps:
  a. *Design the adder kernel with Vitis HLS
  b. *Generate the Overlay with Vivado
  c. Write the Overly onto the board
  d. Evaluate through PYNQ

  *Steps with * doesn’t require a board aside*

Reference: [PYNQ Docs: Overlay Design Methodology]

Create HLS project

```
open_project adder
set_top add
add_files adder_srcs/adder.h
add_files adder_srcs/adder.cpp
add_files –tb adder_srcs/adder_tb.cpp
add_files –tb adder_srcs/adder.cpp
open_solution "solution1" –flow_target vivado
set_part {xc7z020–clg400–1}
create_clock –period 10 –name default
csim_design
csynth_design
export_design –rtl verilog –format ip_catalog
```

- Open Vitis HLS
- Create New Project
  - adder
  - add source files: adder.[hpp/cpp]
  - add test bench files: adder[_tb].cpp
  - set period (clock cycle)
  - choose the **board** *(xc7z020clg400-1)*
- Set top function
- Run C-Simulation \ C-Synthesis \ Export RTL

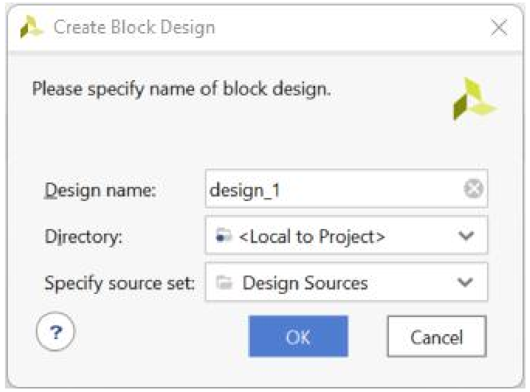
Create Project in Vivado

- Create project and name it scalar_add*_proj*
    - choose do not specify sources
    - select pynq-z2 **board**
- Add *HLS* IP
    - Click Tools > Settings > IP > Repository > "+"
    - choose your HLS project directory

```
vivado -mode tcl
create_project scalar_add_proj ./scalar_add_proj -part xc7z020clg400-1
set_property board_part tul.com.tw:pynq-z2:part0:1.0 [current_project]
set_property  ip_repo_paths path_to_hls_project [current_project]
update_ip_catalog
```

Create Block Design

- Click “Create Block Design” under “IP Integrator” in Project Manager
- Click “+”, add “ZYNQ7 Processing System”

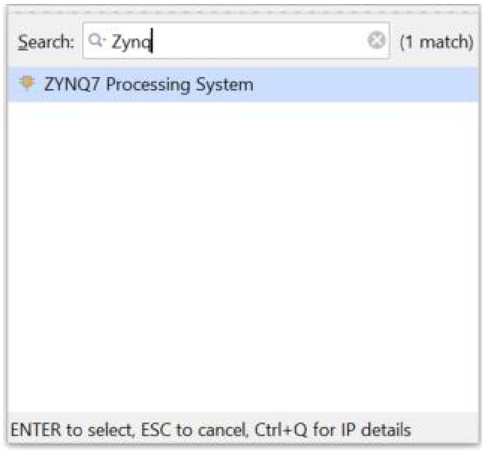

Create Block Design

Please specify name of block design.

Design name: design_1

Directory: <Local to Project>

Specify source set: Design Sources


Search: Zynq (1 match)

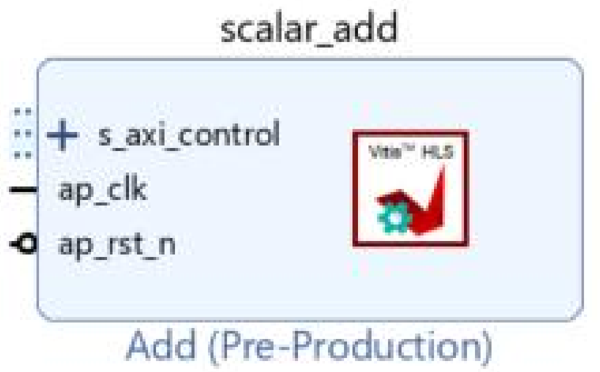
ZYNQ7 Processing System

ENTER to select, ESC to cancel, Ctrl+Q for IP details

```
create_bd_design "design_1"
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5 processing_system7_0
```

HLS IP and AXI DMA IP

- Add the *HLS* IP “Add”
  - rename it to “scalar_add”



```
create_bd_cell -type ip -vlnv xilinx.com:hls:add:1.0 add_0
set_property name scalar_add [get_bd_cells add_0]
```

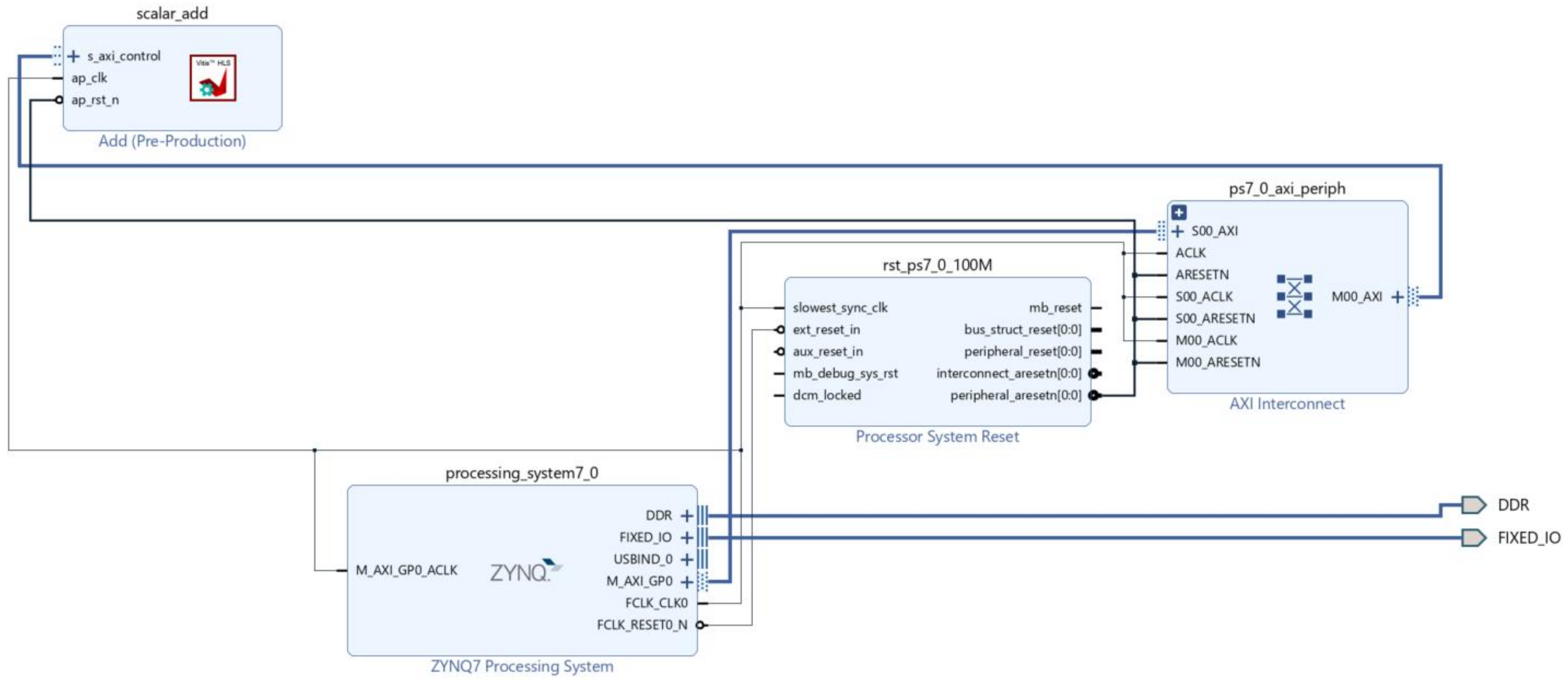
Automatic connections

- Click on **Run Block Automation**
- Click on **Run Connection Automation** and select all.

```
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config {make_external
"FIXED_IO, DDR" apply_board_preset "1" Master "Disable" Slave "Disable" }  [get_bd_cells
processing_system7_0]

apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Clk_slave {Auto}
Clk_xbar {Auto} Master {/processing_system7_0/M_AXI_GP0} Slave {/scalar_add/s_axi_control}
ddr_seg {Auto} intc_ip {New AXI Interconnect} master_apm {0}}  [get_bd_intf_pins
scalar_add/s_axi_control]
```

Automatic connections (2)



Generate bitstream

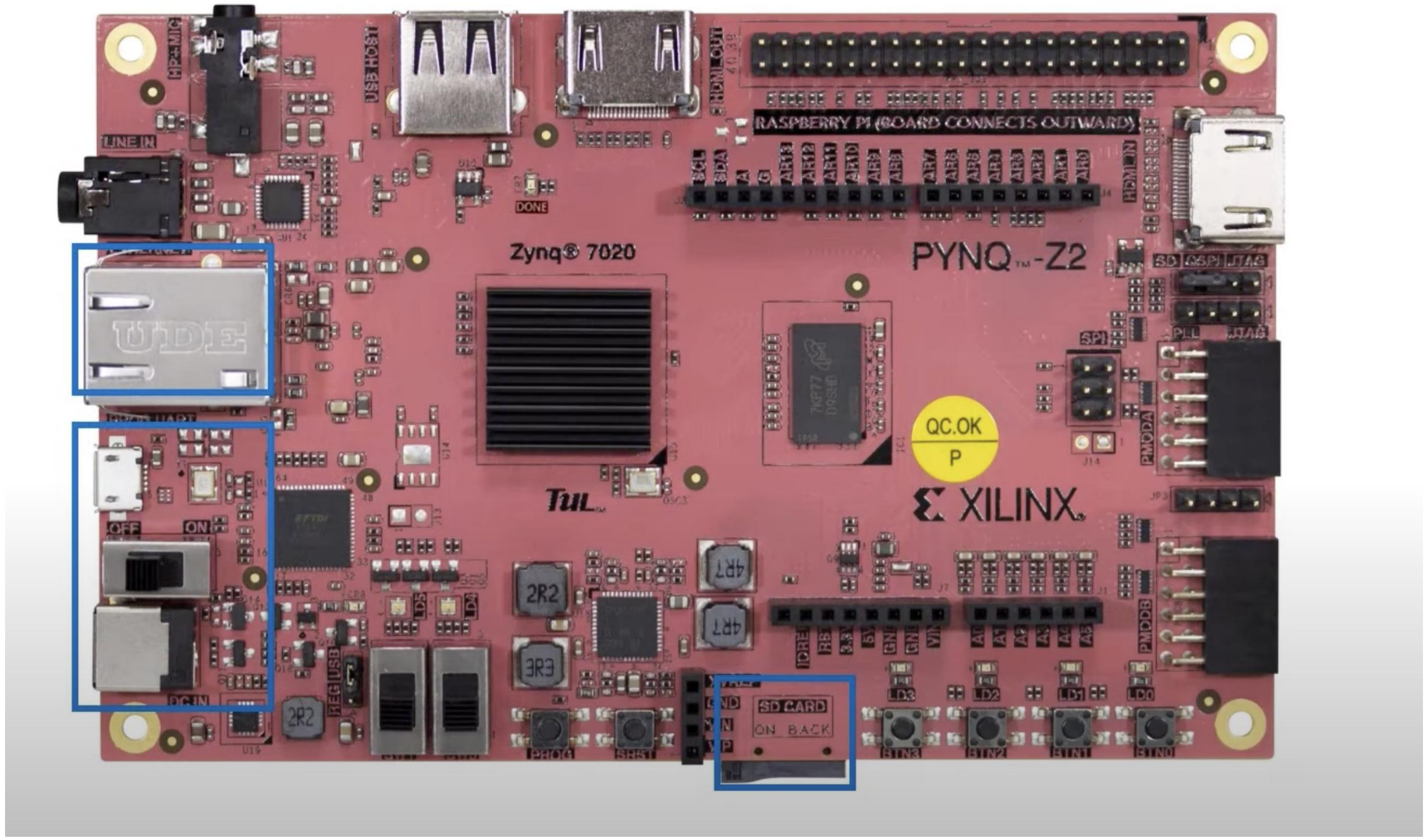
- Save your design **CTRL+S** or **File > Save Block Design**
- Validate your design: **Tools > Validate Design**
- In Sources, right click on **design_1**, and **Create HDL Wrapper**. Now you should have **design_1_wrapper.**
- Generate bitstream by clicking on **Generate Bitstream**

```
save_bd_design
validate_bd_design
make_wrapper -files
[get_files ./scalar_add_proj/scalar_add_proj.srcs/sources_1/bd/design_1/design_1.bd] –top
add_files -
norecurse ./scalar_add_proj/scalar_add_proj.gen/sources_1/bd/design_1/hdl/design_1_wrapper.v
launch_runs impl_1 -to_step write_bitstream -jobs 12
```

Implement and Export

- Get *design_1_wrapper.bit* from ./scalar_add_proj/scalar_add_proj.runs/impl_1
- Get *design_1.hwh* from ./scalar_add_proj/scalar_add_proj.gen/sources_1/bd/design_1/hw_handoff
- Rename them to scalar_add.[bit/hwh]
- Put them under directory scalar_add

Power, SD card, Ethernet

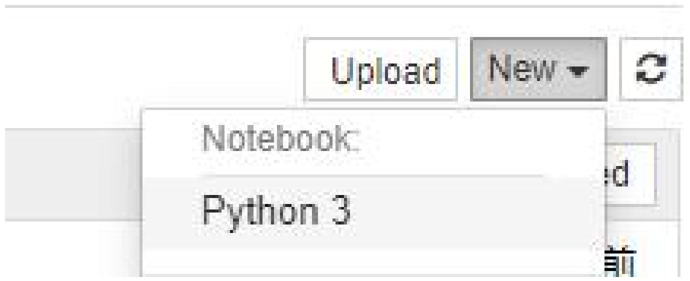


Prepare PYNQ Jupyter Notebook

- Change the Ethernet setting
  - Ipv4: 192.168.2.1 mask:255.255.255.0

- Move scalar_add file to pynq\overlays
  - windows: \\pynq\xilinx\pynq\overlays
  - mac\ubuntu: smb://192.168.2.99/xilinx /pynq/overlays
  - https://ag.montana.edu/it/support/smb-macs.html
  - Both username and password are xilinx

- Open the jupyter website, click New Python 3
  - http://192.168.2.99
  - Password is xilinx



Prepare PYNQ Jupyter Notebook

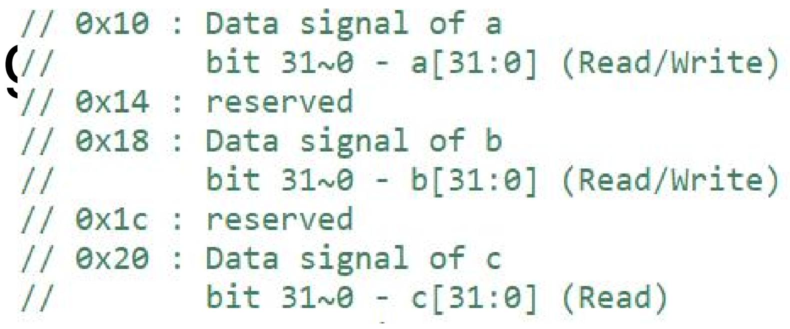
```
from pynq import Overlay

overlay = Overlay("/home/xilinx/pynq/overlays/scalar_add/scalar_add.bit")

add_ip = overlay.scalar_add
```

Interact with the Board

- Write IP port with the IP's SW/HW mapping

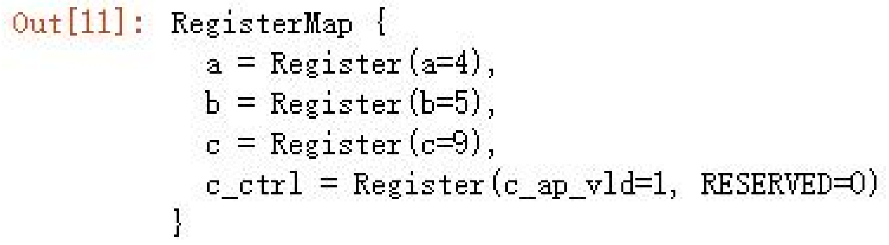


```
// 0x10 : Data signal of a
//        bit 31~0 - a[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of b
//        bit 31~0 - b[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of c
//        bit 31~0 - c[31:0] (Read)
```

add_ip.write(0x10, 4)
add_ip.write(0x18, 5)

- Check IP's register_map

add_ip.register_map



```
Out[11]: RegisterMap {
           a = Register(a=4),
           b = Register(b=5),
           c = Register(c=9),
           c_ctrl = Register(c_ap_vld=1, RESERVED=0)
         }
```

- Read result from IP port

add_ip.read(0x20)

Play with LEDs

- http://192.168.2.99:9090/notebooks/base/board/board_btns_leds.ipynb

- on, off, toggle

Next

- Feel free to explore Section *Overlay Design Methodology and* Tutorial
- . PYNQ community provides some good tutorials:
  - VSCode development
  - Xilinx's PYNQ Workshop
  - An Matmult example
- Reference:
  - ug892-vivado-design-flows-overview
  - ug994-vivado-ip-subsystems
  - ug835-vivado-tcl-commands