PYNQ™

Introduction to Overlays

XILINX®

Outline

- Overlay Concept
- External devices support
- base Overlay
- Python programmer’s view

FPGA overlays – hardware libraries

> **Overlays are generic FPGA designs that target multiple users with new design abstractions and tools**

> **Overlay characteristics**

- Post-bitstream programmable via software APIs
- Typically optimized for given application domains
- Encourages the use of open source tools & fast compilation
- Enables productivity by re-using pre-optimized designs
- Makes benefits of FPGAs accessible to new users

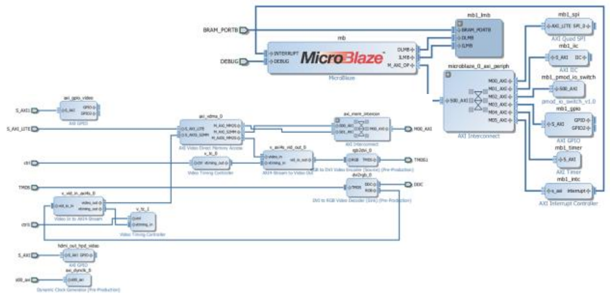
XILINX.

Anatomy of an overlay IP subsystem

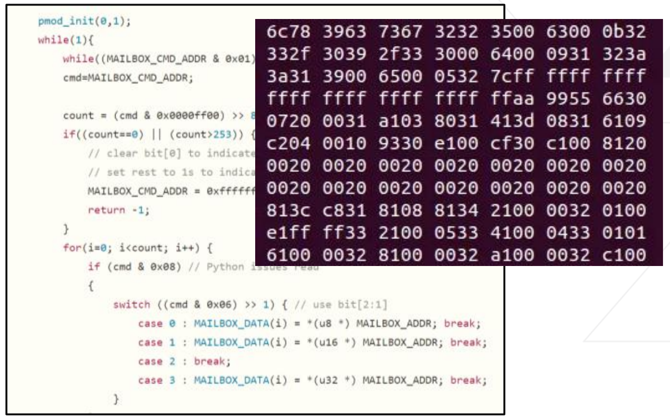
- **Designed to be immediately reused by anyone**
  - or re-purposed elsewhere by “person skilled in the art” (PSITA)
- **Comprises**
  - Programmable FPGA IP core
  - FPGA bitstream
  - C code to expose programmable functionality
  - Python-to-C bindings
  - Python library with API
  - Protocol
  - Jupyter notebook examples

XILINX

FPGA overlays – hardware libraries



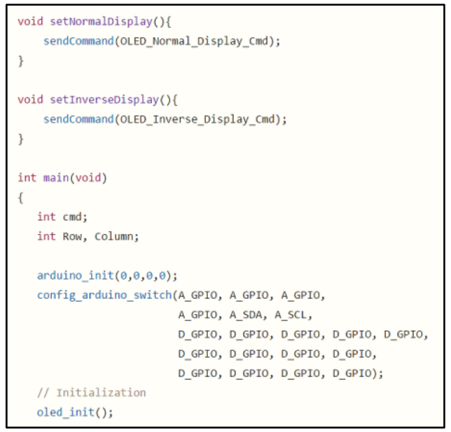
Step 1:
Create an FPGA design for a class of related applications



```
pmod_init(0,1);
while(1){
    while((MAILBOX_CMD_ADDR & 0x01)
    cmd=MAILBOX_CMD_ADDR;

    count = (cmd & 0x0000ff00) >> 8
    if((count==0) || (count>253)) {
        // clear bit[0] to indicate
        // set rest to 1s to indica
        MAILBOX_CMD_ADDR = 0xfffff
        return -1;
    }
    for(i=0; i<count; i++) {
        if (cmd & 0x08) // Python issues read
        {
            switch ((cmd & 0x06) >> 1) { // use bit[2:1]
                case 0 : MAILBOX_DATA(i) = *(u8 *) MAILBOX_ADDR; break;
                case 1 : MAILBOX_DATA(i) = *(u16 *) MAILBOX_ADDR; break;
                case 2 : break;
                case 3 : MAILBOX_DATA(i) = *(u32 *) MAILBOX_ADDR; break;
            }
```

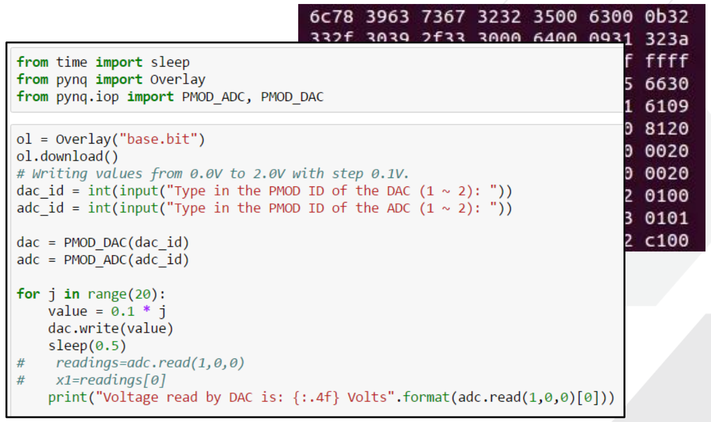
```
6c78 3963 7367 3232 3500 6300 0b32
332f 3039 2f33 3000 6400 0931 323a
3a31 3900 6500 0532 7cff ffff ffff
ffff ffff ffff ffff ffaa 9955 6630
0720 0031 a103 8031 413d 0831 6109
c204 0010 9330 e100 cf30 c100 8120
0020 0020 0020 0020 0020 0020 0020
0020 0020 0020 0020 0020 0020 0020
813c c831 8108 8134 2100 0032 0100
e1ff ff33 2100 0533 4100 0433 0101
6100 0032 8100 0032 a100 0032 c100
```

Step 2:
Export the bitstream and a C API for programming the design



```
void setNormalDisplay(){
    sendCommand(OLED_Normal_Display_Cmd);
}

void setInverseDisplay(){
    sendCommand(OLED_Inverse_Display_Cmd);
}

int main(void)
{
   int cmd;
   int Row, Column;

   arduino_init(0,0,0,0);
   config_arduino_switch(A_GPIO, A_GPIO, A_GPIO,
                         A_GPIO, A_SDA, A_SCL,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO,
                         D_GPIO, D_GPIO, D_GPIO, D_GPIO);

   // Initialization
   oled_init();
```

Step 3:
Wrap the C API to create a Python library



```
6c78 3963 7367 3232 3500 6300 0b32
332f 3039 2f33 3000 6400 0931 323a
                             f ffff
                             5 6630
                             1 6109
                             0 8120
                             0 0020
                             0 0020
                             2 0100
                             3 0101
                             2 c100
```

```python
from time import sleep
from pynq import Overlay
from pynq.iop import PMOD_ADC, PMOD_DAC

ol = Overlay("base.bit")
ol.download()
# Writing values from 0.0V to 2.0V with step 0.1V.
dac_id = int(input("Type in the PMOD ID of the DAC (1 ~ 2): "))
adc_id = int(input("Type in the PMOD ID of the ADC (1 ~ 2): "))

dac = PMOD_DAC(dac_id)
adc = PMOD_ADC(adc_id)

for j in range(20):
    value = 0.1 * j
    dac.write(value)
    sleep(0.5)
#    readings=adc.read(1,0,0)
#    x1=readings[0]
    print("Voltage read by DAC is: {:.4f} Volts".format(adc.read(1,0,0)[0]))
```

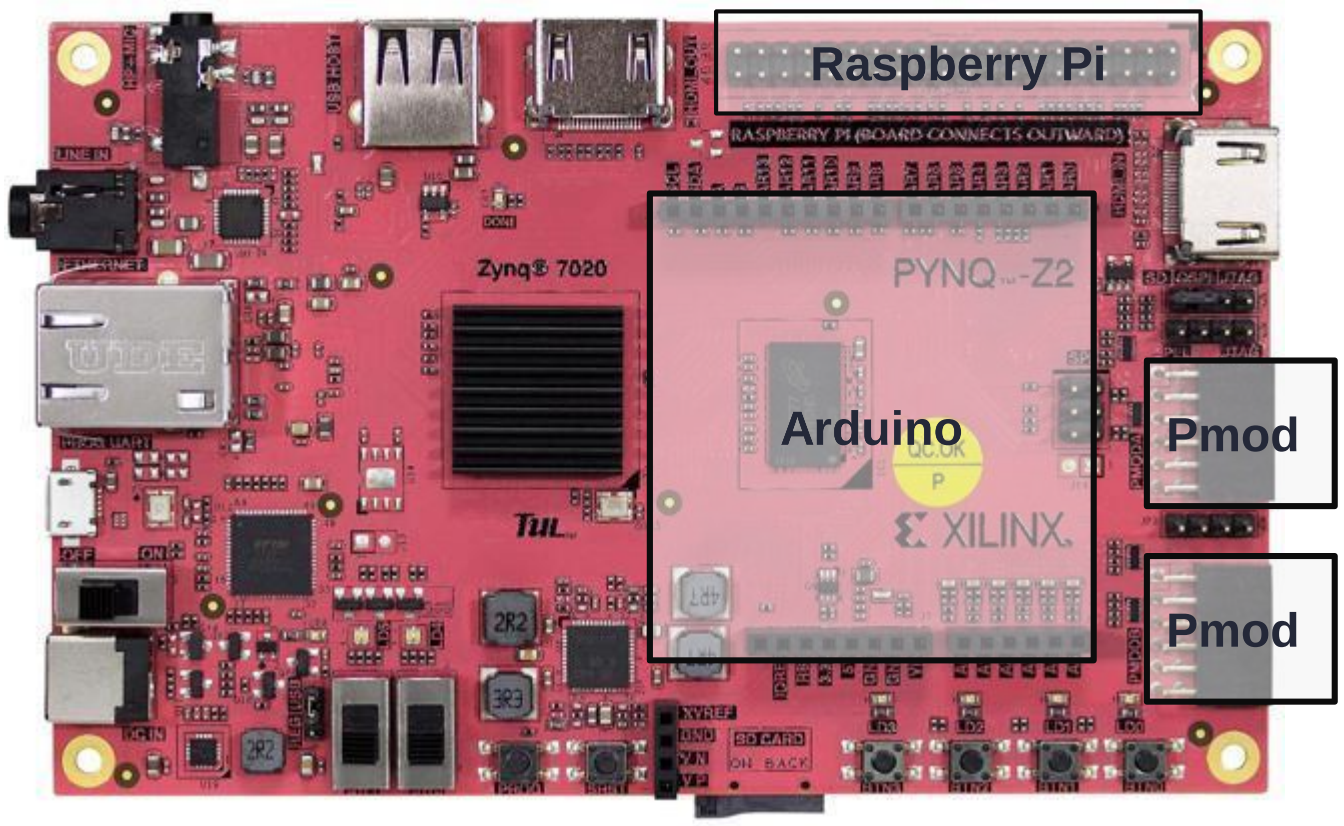
Step 4:
Import the bitstream and the library in your Python scripts and program

External interfacing with the Base Overlay

XILINX

PYNQ

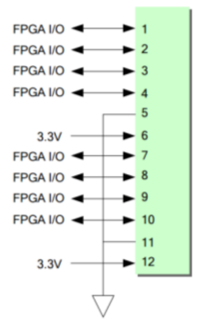
Low-cost PYNQ boards: Pmod, RPi, Arduino Interfaces

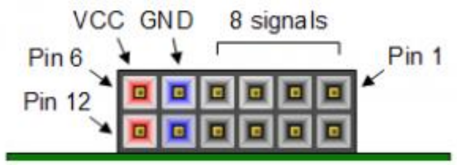


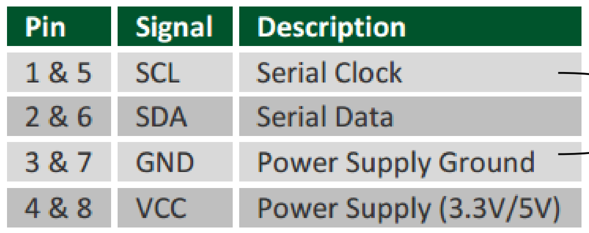
Typically every new Pmod, Raspberry Pi, or Arduino module requires a new design/seperate bitstream

Pmod: many physical & electrical instances

Pmod

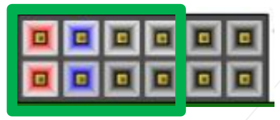


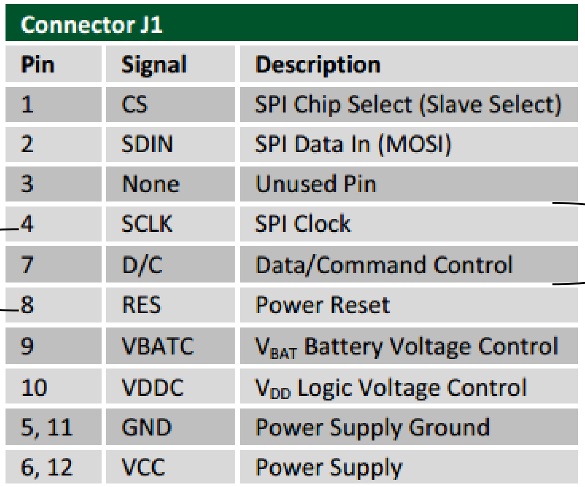




Pin	Signal	Description
1 & 5	SCL	Serial Clock
2 & 6	SDA	Serial Data
3 & 7	GND	Power Supply Ground
4 & 8	VCC	Power Supply (3.3V/5V)

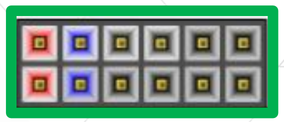
ADC

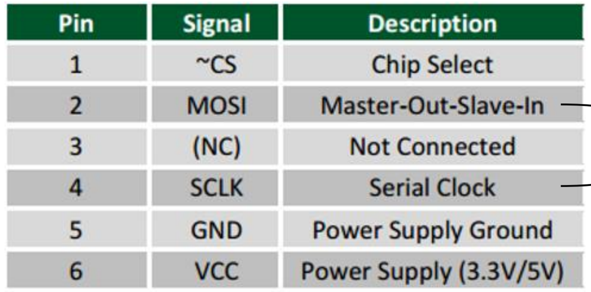




Connector J1		
Pin	**Signal**	**Description**
1	CS	SPI Chip Select (Slave Select)
2	SDIN	SPI Data In (MOSI)
3	None	Unused Pin
4	SCLK	SPI Clock
7	D/C	Data/Command Control
8	RES	Power Reset
9	VBATC	V_{BAT} Battery Voltage Control
10	VDDC	V_{DD} Logic Voltage Control
5, 11	GND	Power Supply Ground
6, 12	VCC	Power Supply

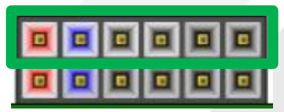
OLED





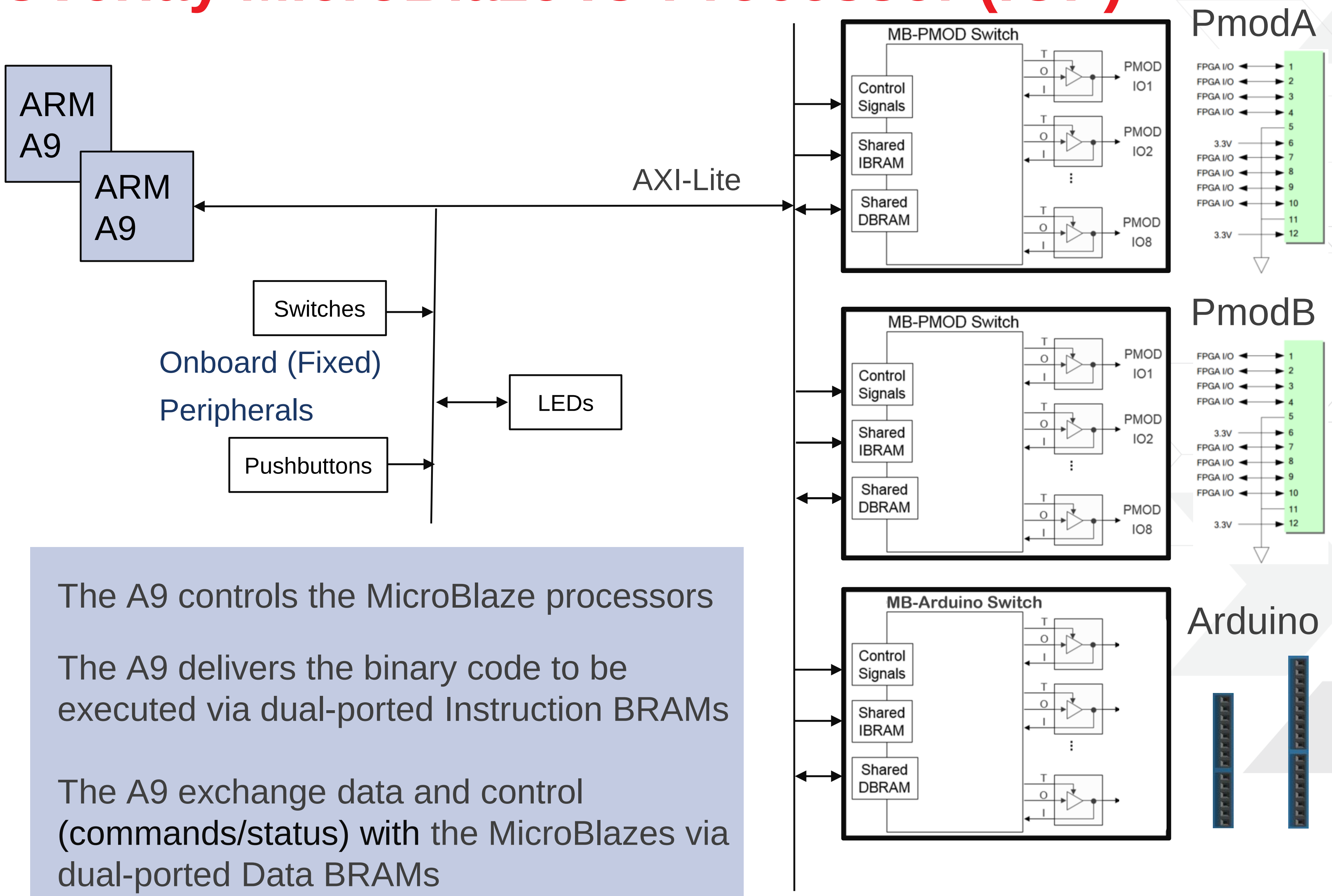
Pin	Signal	Description
1	~CS	Chip Select
2	MOSI	Master-Out-Slave-In
3	(NC)	Not Connected
4	SCLK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

DAC



What if we could handle all Pmod instances with a single bitstream?

base Overlay MicroBlaze IO Processor (IOP)

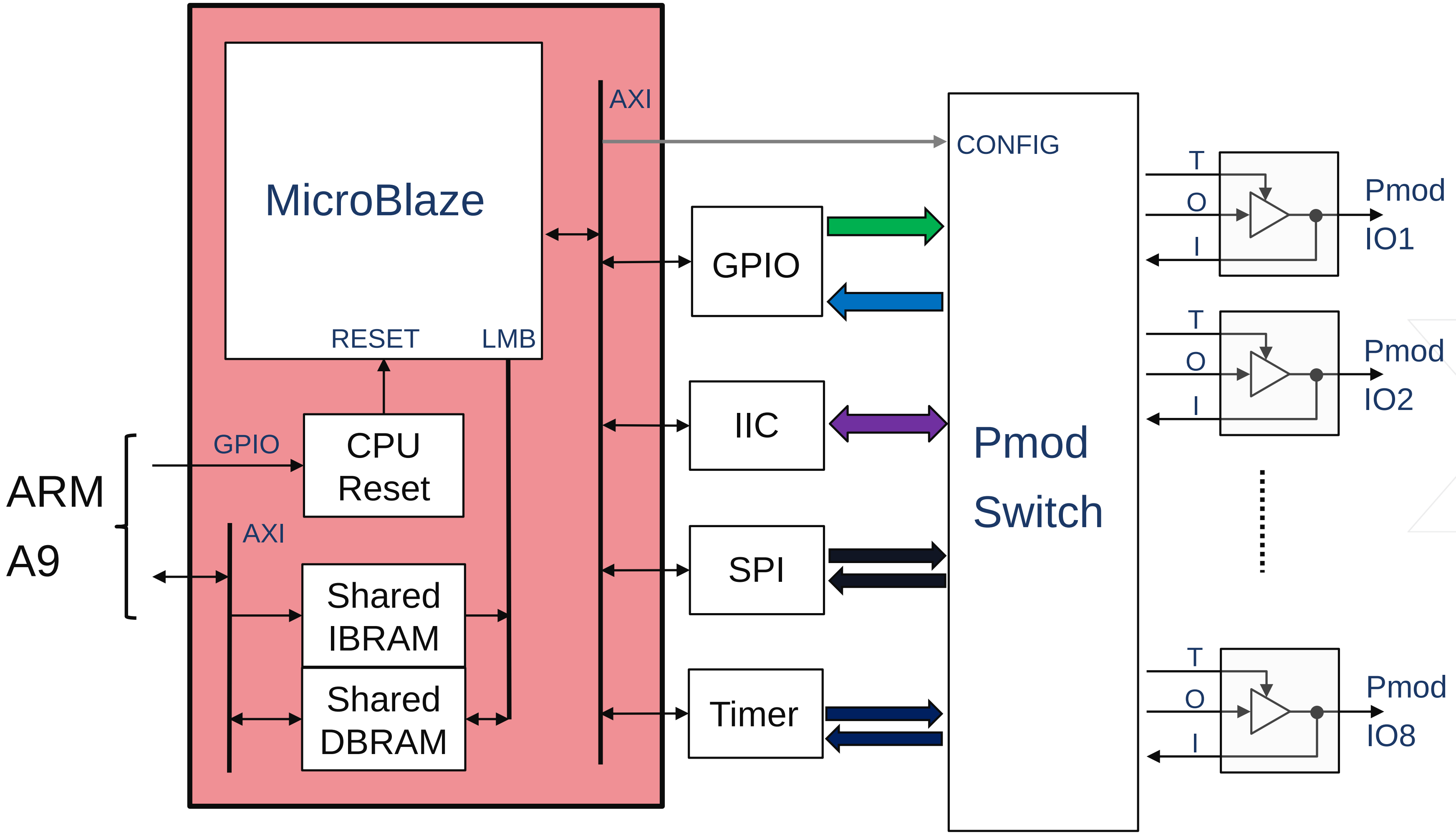


The A9 controls the MicroBlaze processors

The A9 delivers the binary code to be executed via dual-ported Instruction BRAMs

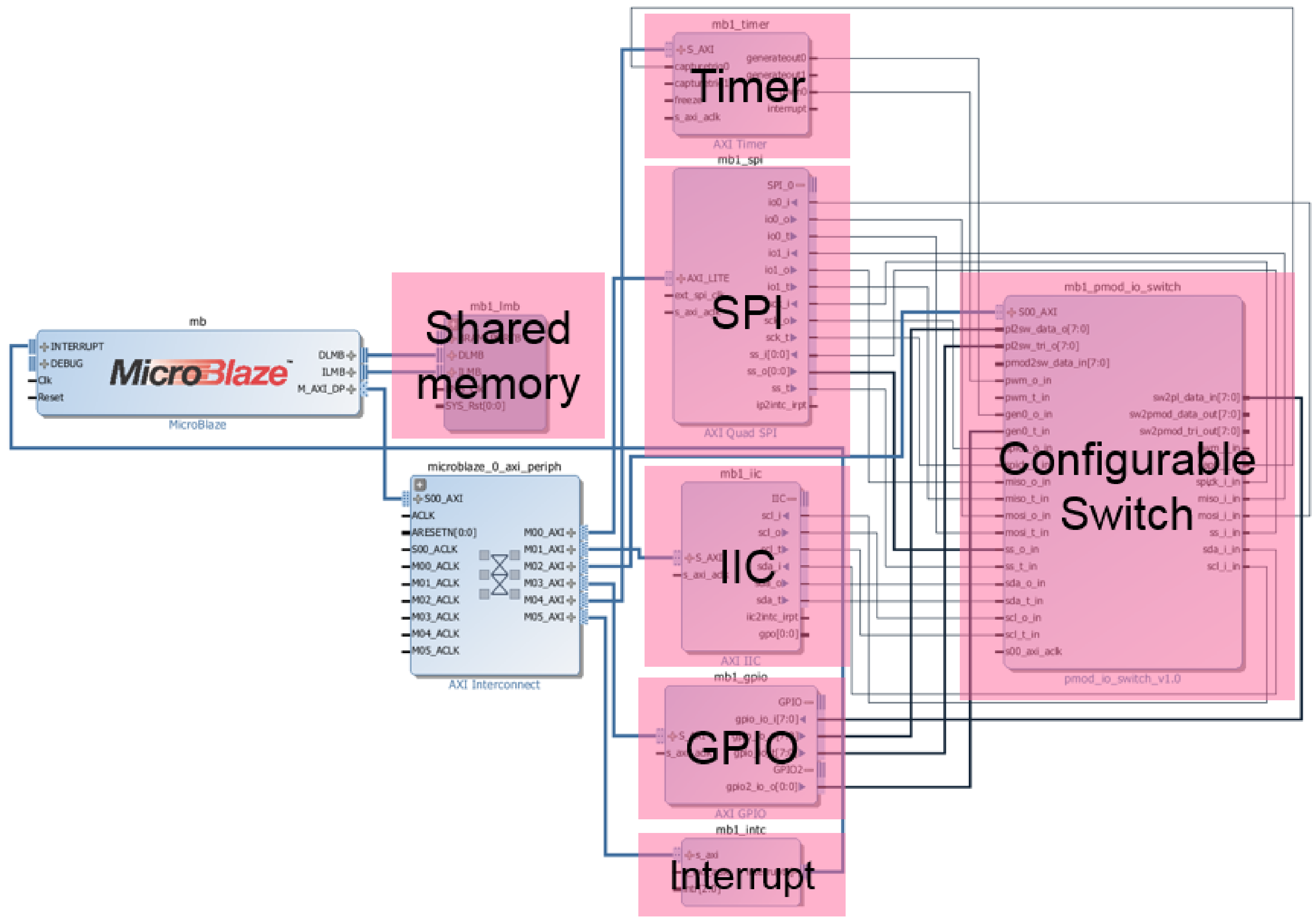
The A9 exchange data and control (commands/status) with the MicroBlazes via dual-ported Data BRAMs

Configure IO Processor for Pmod



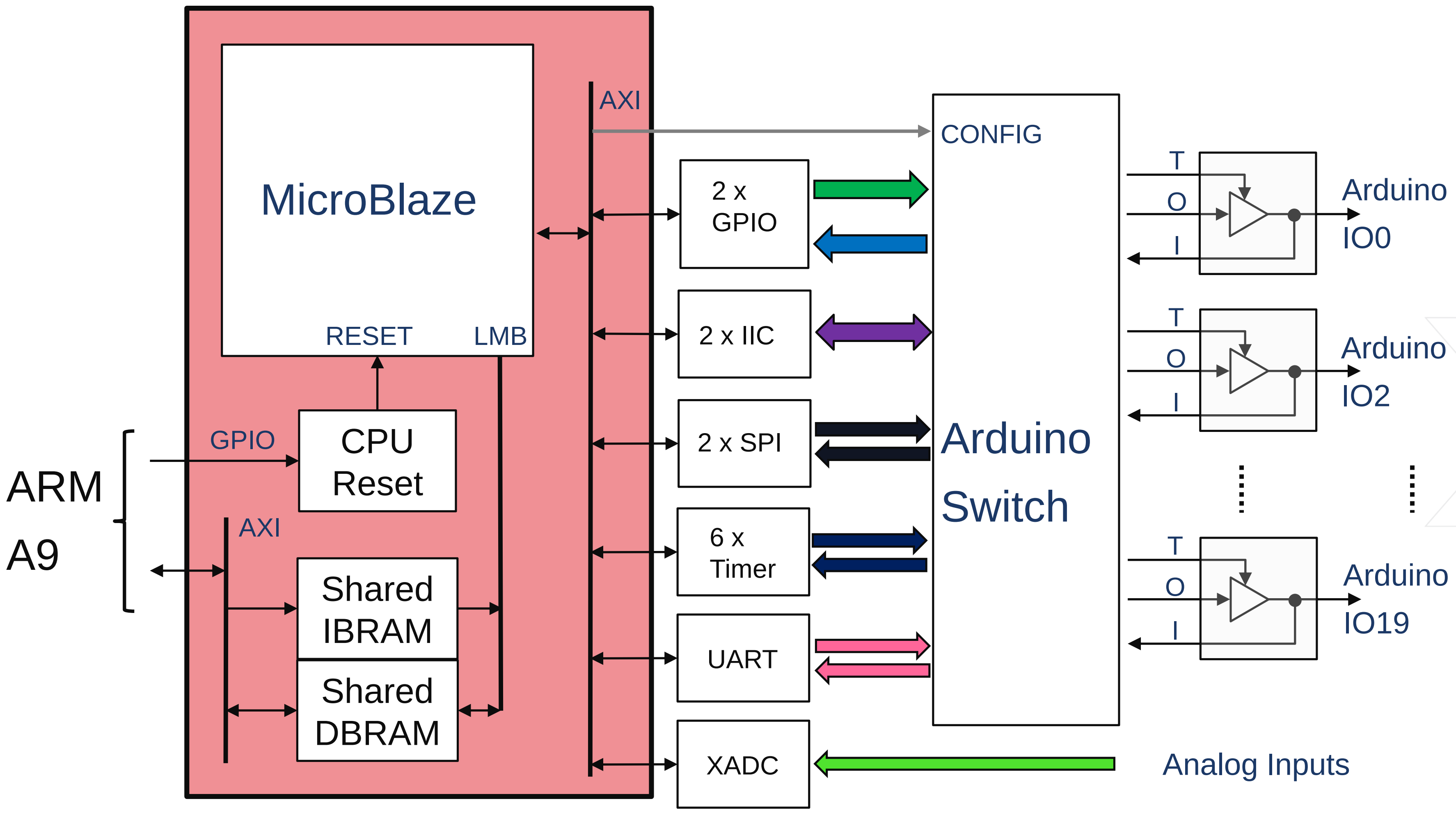
XILINX

Pmod IOP

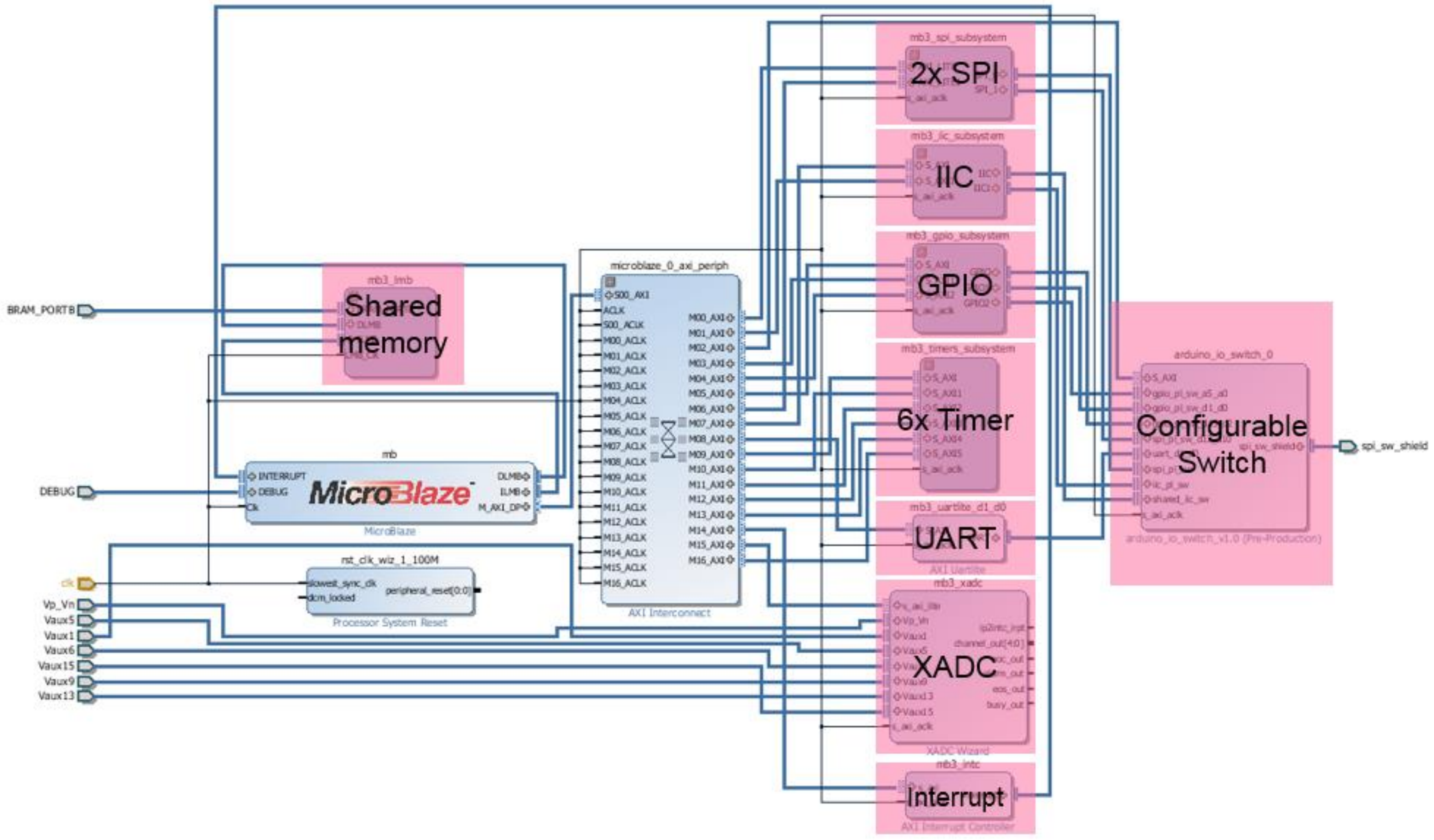


mb
MicroBlaze
INTERRUPT
DEBUG
Clk
Reset
DLMB
ILMB
M_AXI_DP
Shared memory
mb1_lmb
DLMB
ILMB
SYS_Rst[0:0]
microblaze_0_axi_periph
S00_AXI
ACLK
ARESETN[0:0]
S00_ACLK
M00_ACLK
M01_ACLK
M02_ACLK
M03_ACLK
M04_ACLK
M05_ACLK
M00_AXI
M01_AXI
M02_AXI
M03_AXI
M04_AXI
M05_AXI
AXI Interconnect
mb1_timer
Timer
S_AXI
generateout0
generateout1
interrupt
freeze
s_axi_aclk
AXI Timer
mb1_spi
SPI
SPI_0
io0_i
io0_o
io0_t
io1_i
io1_o
io1_t
AXI_LITE
ext_spi_clk
s_axi_aclk
sck_t
ss_i[0:0]
ss_o[0:0]
ss_t
ip2intc_irpt
AXI Quad SPI
mb1_iic
IIC
scl_i
scl_o
S_AXI
sda_t
iic2intc_irpt
gpo[0:0]
AXI IIC
mb1_gpio
GPIO
gpio_io_i[7:0]
GPIO2
gpio2_io_o[0:0]
AXI GPIO
mb1_intc
Interrupt
mb1_pmod_io_switch
Configurable Switch
S00_AXI
pl2sw_data_o[7:0]
pl2sw_tri_o[7:0]
pmod2sw_data_in[7:0]
pwm_o_in
pwm_t_in
gen0_o_in
gen0_t_in
miso_o_in
miso_t_in
mosi_o_in
mosi_t_in
ss_o_in
ss_t_in
sda_o_in
sda_t_in
scl_o_in
scl_t_in
s00_axi_aclk
sw2pl_data_in[7:0]
sw2pmod_data_out[7:0]
sw2pmod_tri_out[7:0]
miso_i_in
mosi_i_in
ss_i_in
sda_i_in
scl_i_in
pmod_io_switch_v1.0

Configure IO Processor for Arduino Shield

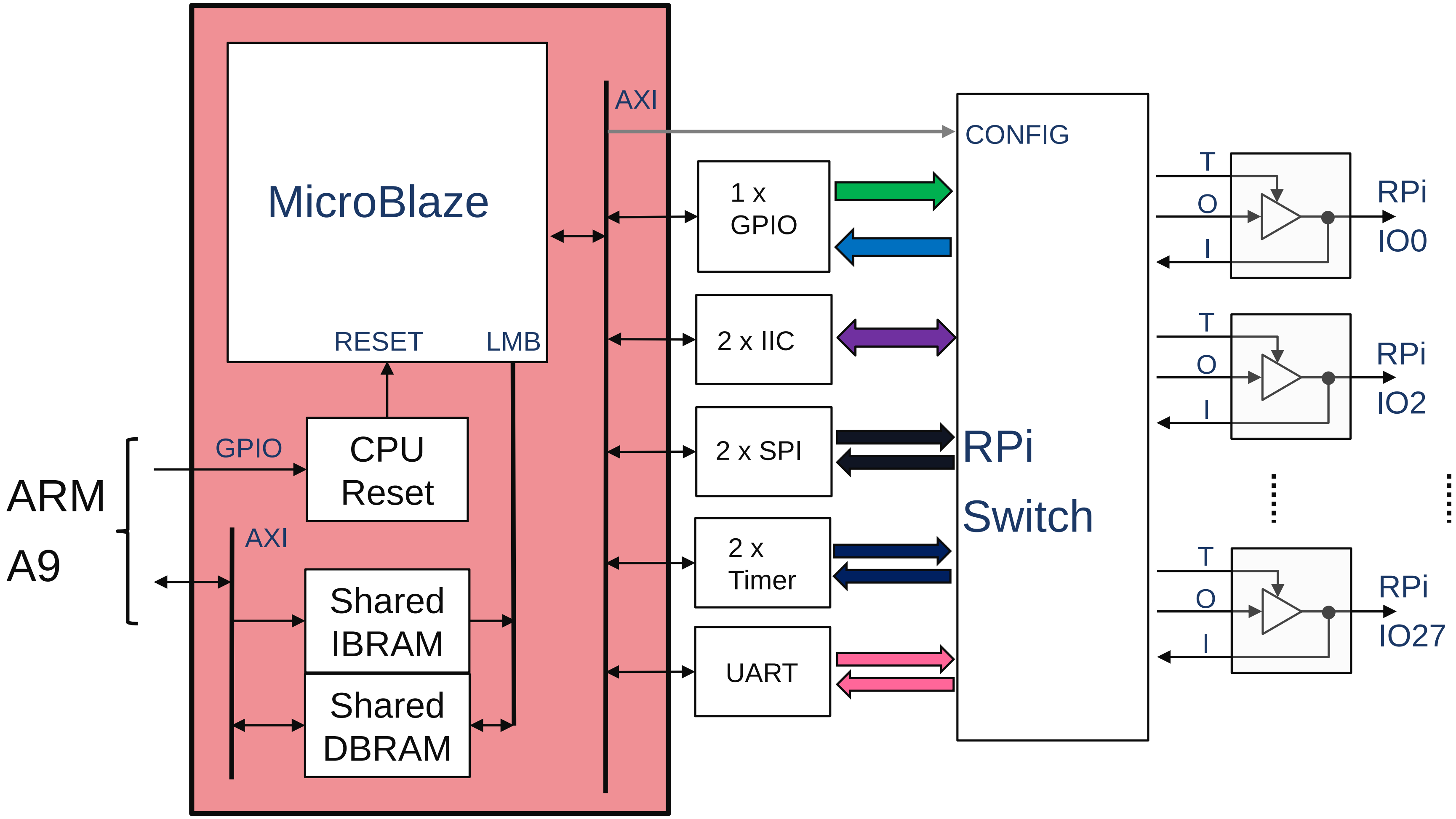


Arduino IOP

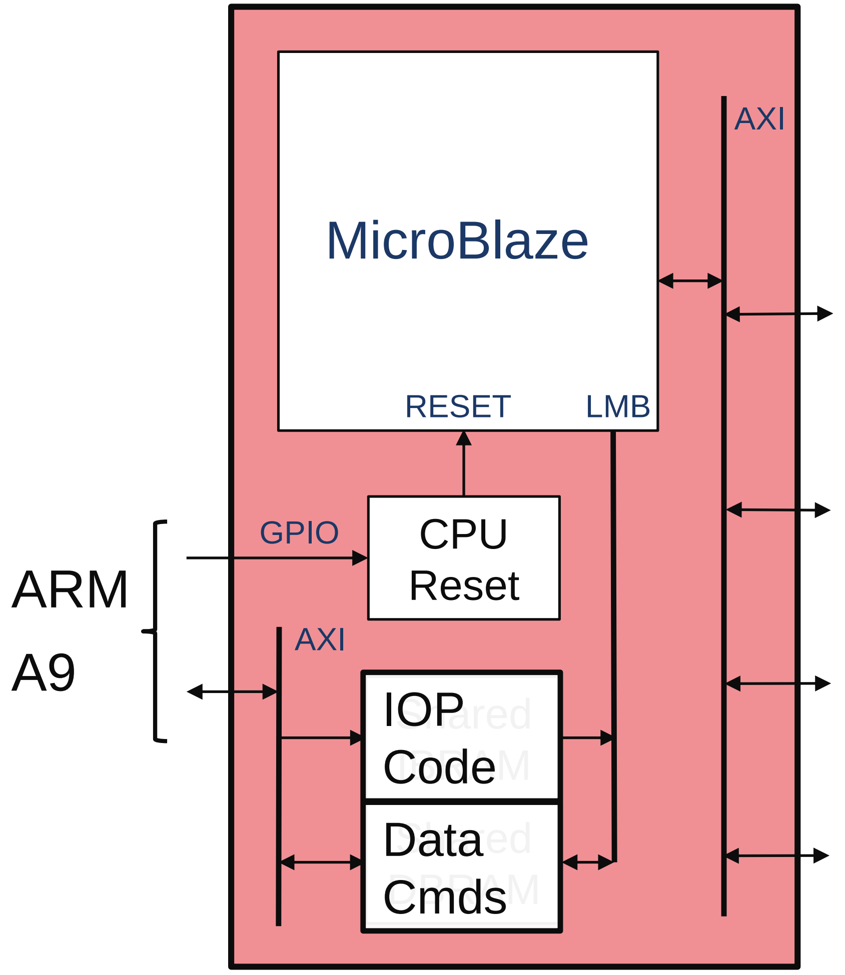


BRAM_PORTB
DEBUG
clk
Vp_Vn
Vaux5
Vaux1
Vaux6
Vaux15
Vaux9
Vaux13
mb3_lmb
Shared memory
mb
INTERRUPT
DEBUG
Clk
DLMB
ILMB
M_AXI_DP
MicroBlaze
rst_clk_wiz_1_100M
slowest_sync_clk
dcm_locked
peripheral_reset[0:0]
Processor System Reset
microblaze_0_axi_periph
S00_AXI
ACLK
AXI Interconnect
mb3_spi_subsystem
2x SPI
mb3_iic_subsystem
IIC
mb3_gpio_subsystem
GPIO
mb3_timers_subsystem
6x Timer
mb3_uartlite_d1_d0
UART
AXI Uartlite
mb3_xadc
XADC
XADC Wizard
mb3_intc
Interrupt
AXI Interrupt Controller
arduino_io_switch_0
Configurable Switch
arduino_io_switch_v1.0 (Pre-Production)
spi_sw_shield

Configure IO Processor for Raspberry Pi Shield



Soft Processor Subsystem (SPS)



The A9 controls the MicroBlaze processors

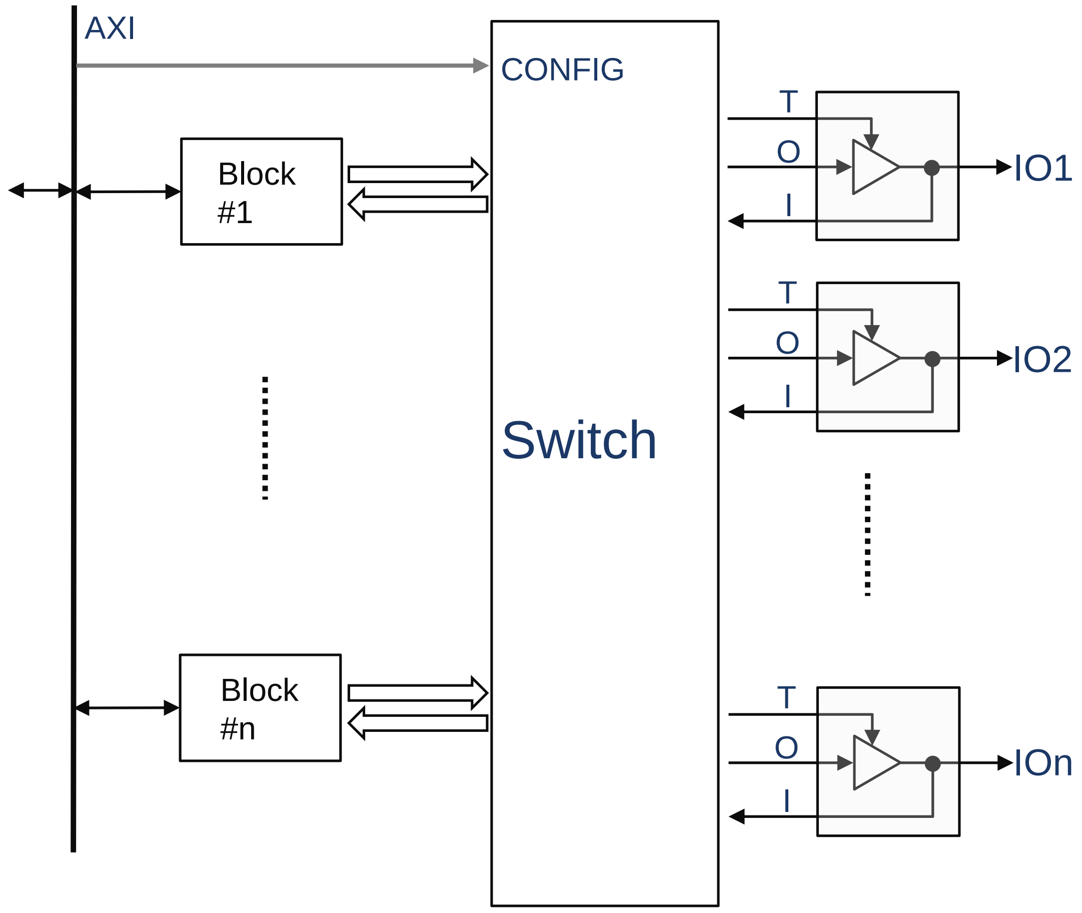
The A9 delivers the binary code to be executed via dual-ported Instruction BRAMs

The A9 exchange data and control (commands/status) with the MicroBlazes via dual-ported Data BRAMs

Multiple SPS units can be used to control subsystems in the PL fabric: e.g. IO interfaces, internal interfaces, data-path units and instrumentation

SPS can also be used for distributed processing

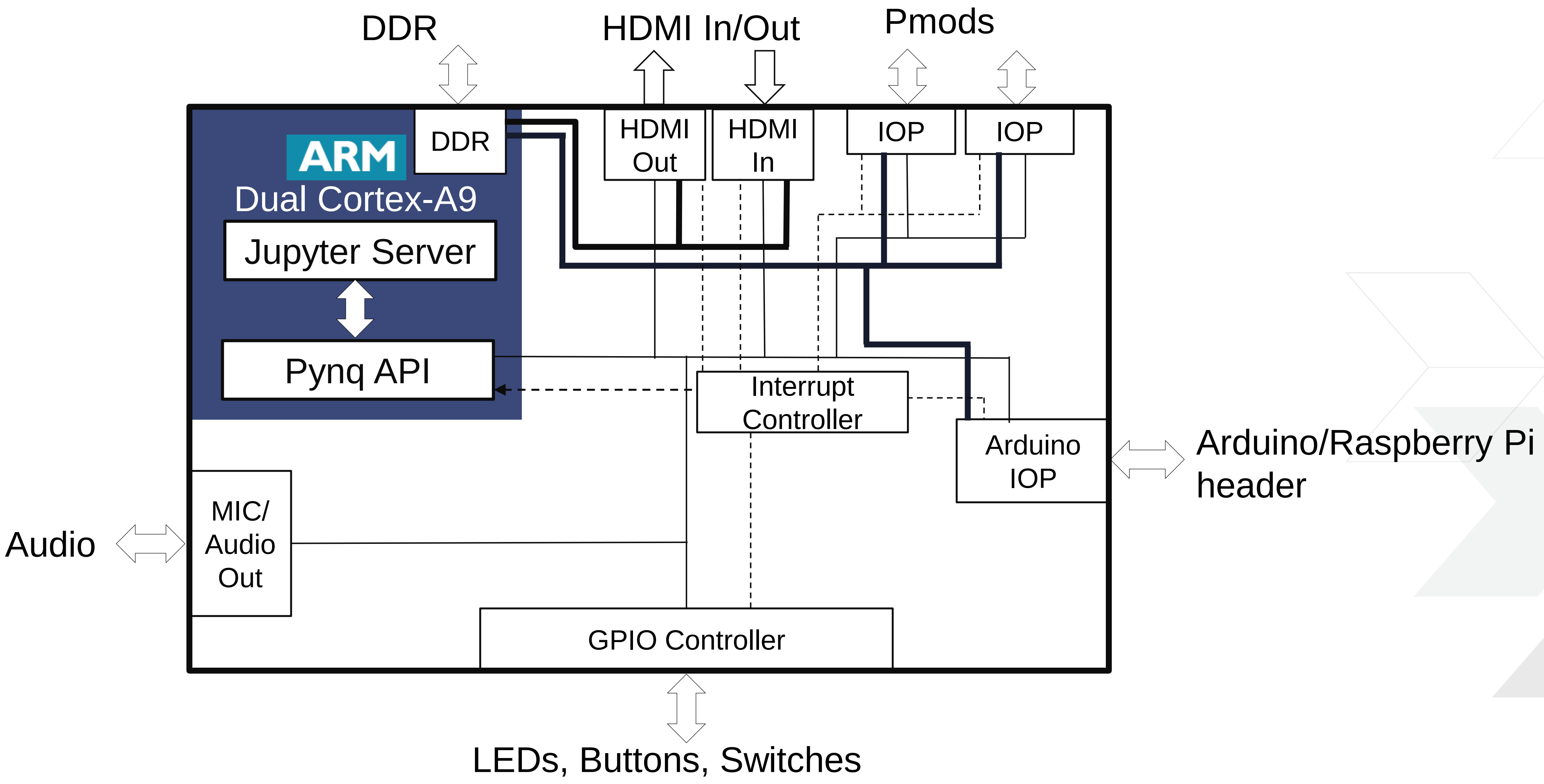
IO Switch (IOS)



The IO Switch
can be re-used to control
any external interface

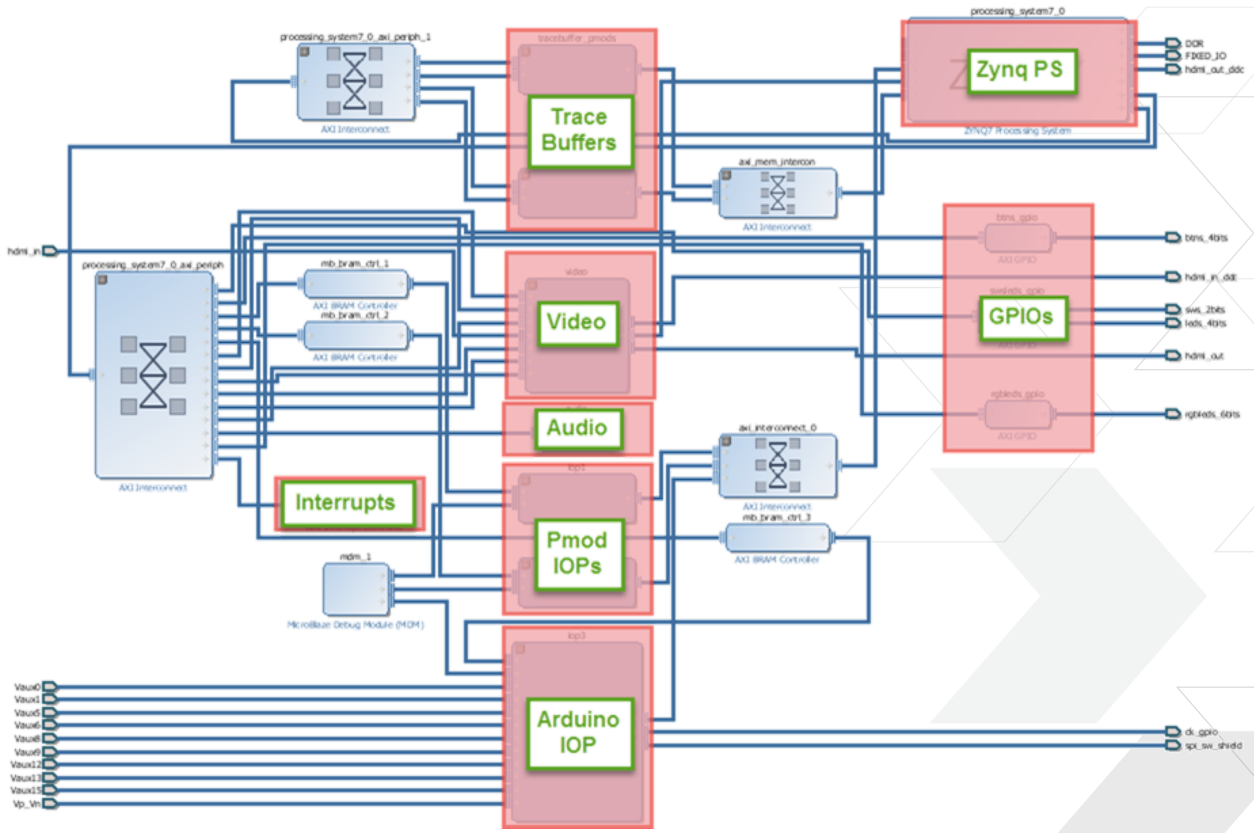
The rest of the *base* Overlay

Complete *base* Overlay (base.bit)



base Overlay – Vivado Interface View

- > PL design of the base Overlay
- > Standard FPGA design flow used
- > Vivado IPI
- > Interface to Python
- > Memory Map
- > Open source
- > Components are re-useable



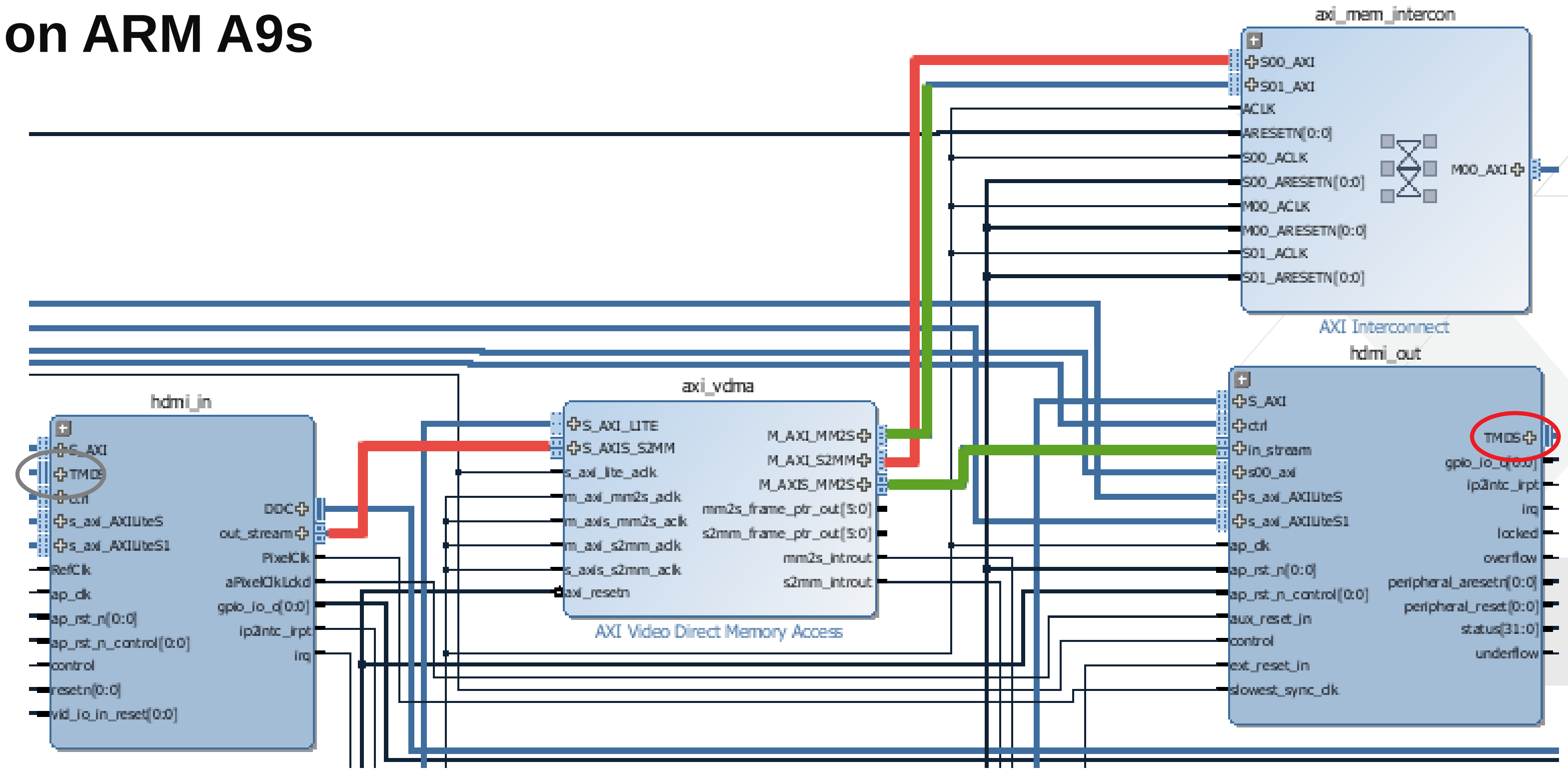
Video

> **HDMI_in, HDMI_out**

- >> Stream from HDMI_in to DRAM; stream from DRAM to HDMI_out
- >> 3 separate DRAM framebuffers available

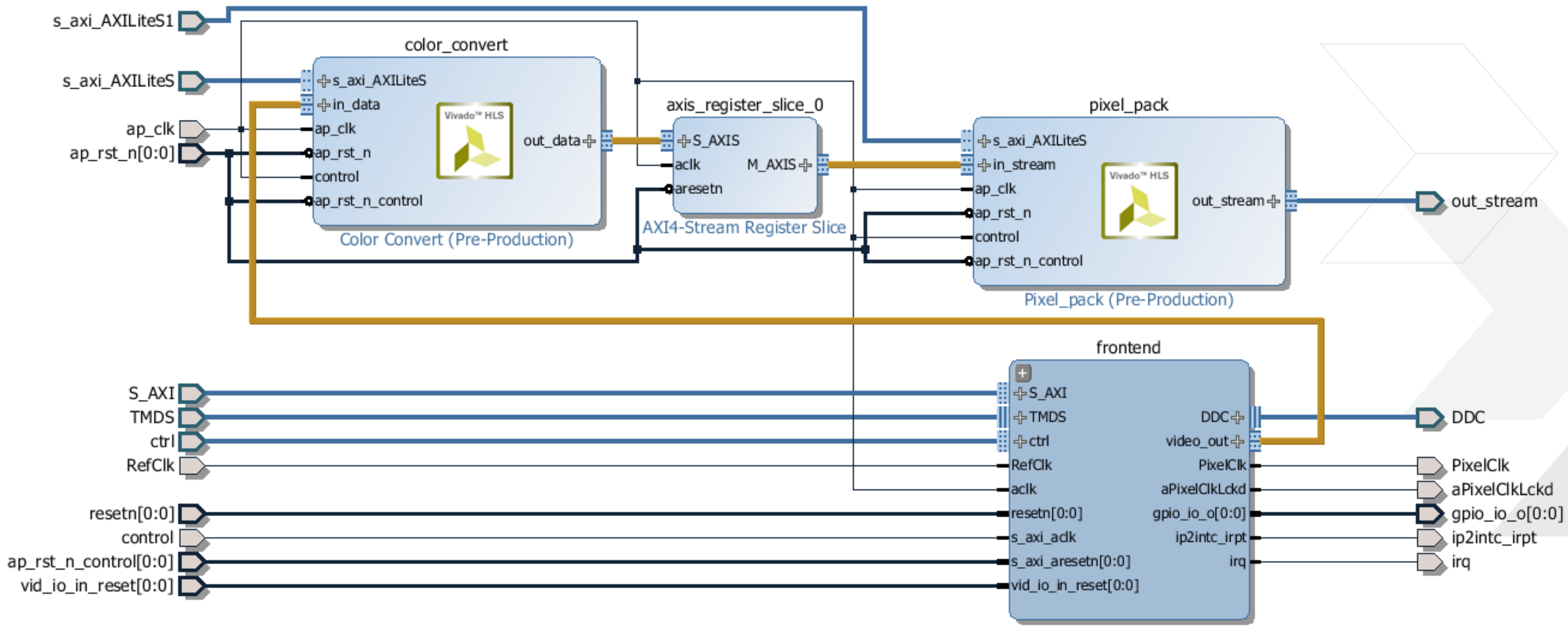
> **Image processing on ARM A9s**

- >> E.g. OpenCV



Video In path

> **HDMI_in (TMDS) -> frontend -> color_convert -> axis_register_slice -> pixel_clock -> axis_vdma -> HP0 (PS7)**
  >> The frontend module wraps all of the clock and timing logic



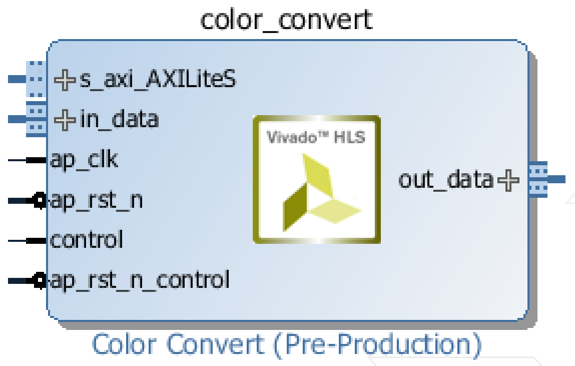
Color conversion and pixel packing

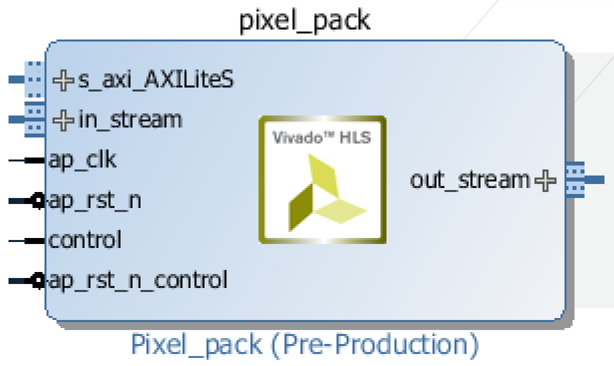
> **color_convert**
>> Transform the input signal into different color spaces

> **pixel_pack**
>> Convert between 8/24/32-bit

> **Default: BGR (24-bit)**
>> RGB (24-bit)
>> RGBA (32-bit)
>> BGR (24-bit)
>> YCbCr (24-bit)
>> Grayscale (8-bit)

> **HLS source available**

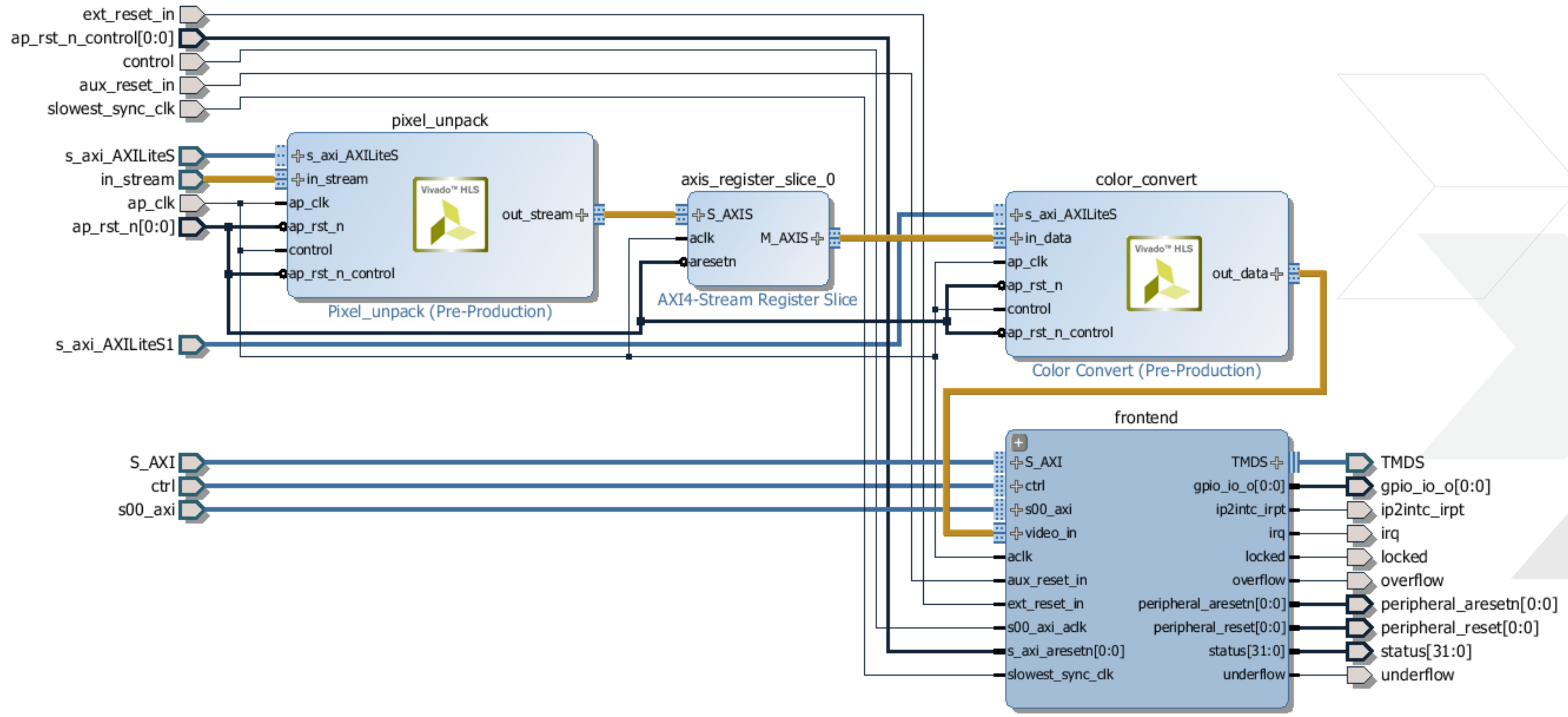




XILINX

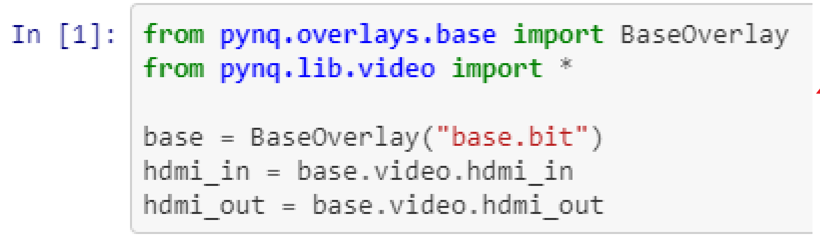
Video Out path

- HP0 (PS7) -> axi_vdma -> pixel_unpack -> axis_register_slice -> color_convert -> frontend -> HDMI_out (TMDS)
  - All sub-modules perform reverse operations of the HDMI IN block



XILINX.

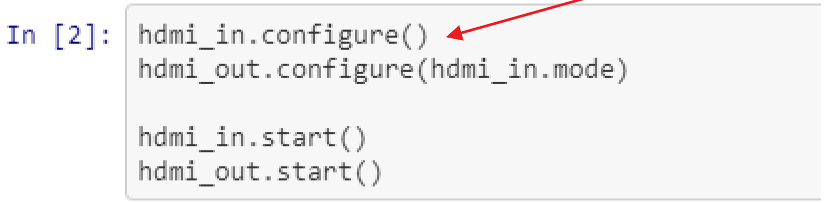
Video example



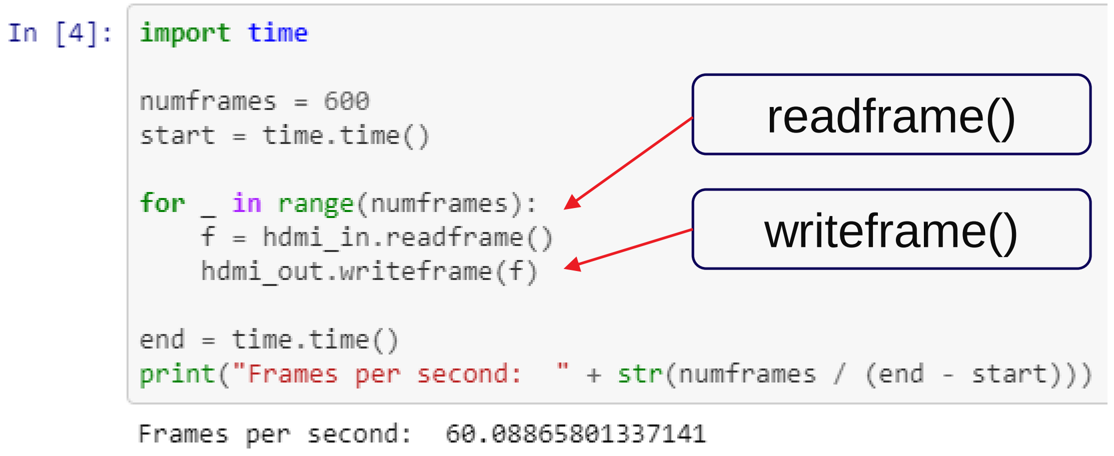
```
In [1]: from pynq.overlays.base import BaseOverlay
        from pynq.lib.video import *

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

Python imports, HDMI instances

Configure() – HDMI in/out resolution/colorspace



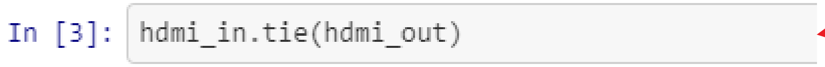
```
In [2]: hdmi_in.configure()
        hdmi_out.configure(hdmi_in.mode)

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



```
In [4]: import time

        numframes = 600
        start = time.time()

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

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

        Frames per second:   60.08865801337141
```

readframe()

writeframe()



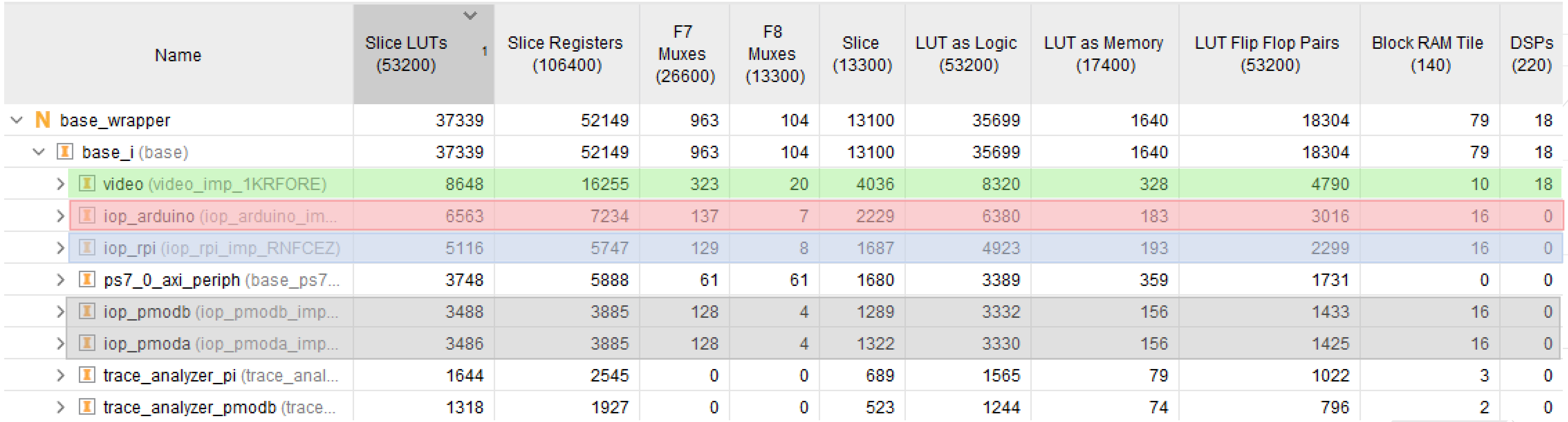
```
In [3]: hdmi_in.tie(hdmi_out)
```

Connect HDMI in to out

https://github.com/Xilinx/PYNQ/blob/master/boards/Pynq-Z1/base/notebooks/video/hdmi_introduction.ipynb

Base overlay resource utilization

- **Z-7020, LUTs resource utilization ~50%**
  - IOP (Pmod) ~ 6%
  - Video ~ 15%
  - IOP (Arduino) ~ 12%
  - IOP (RPi) ~ 10%

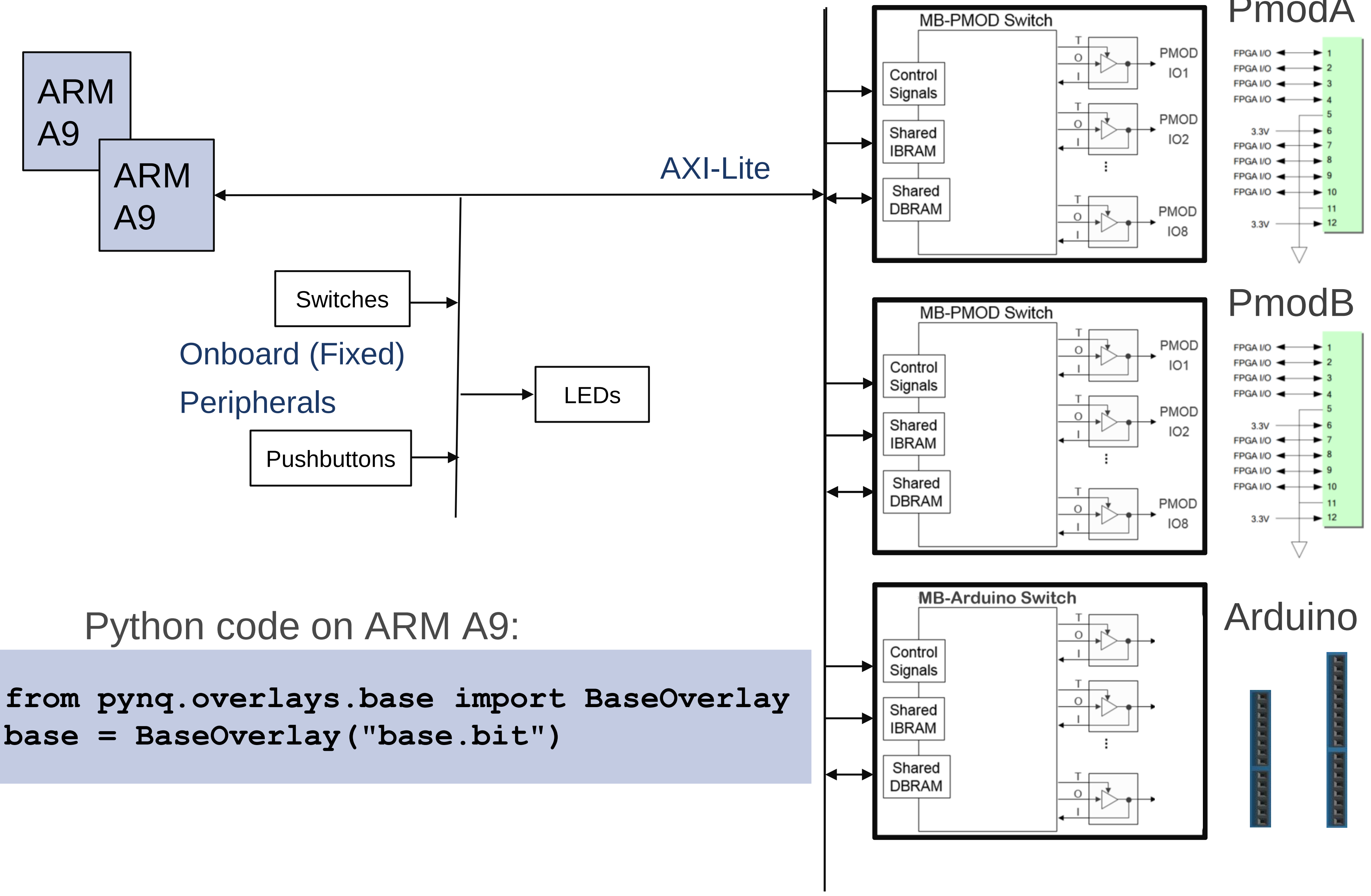


Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	DSPs (220)
base_wrapper	37339	52149	963	104	13100	35699	1640	18304	79	18
base_i (base)	37339	52149	963	104	13100	35699	1640	18304	79	18
video (video_imp_1KRFORE)	8648	16255	323	20	4036	8320	328	4790	10	18
iop_arduino (iop_arduino_im...	6563	7234	137	7	2229	6380	183	3016	16	0
iop_rpi (iop_rpi_imp_RNFCEZ)	5116	5747	129	8	1687	4923	193	2299	16	0
ps7_0_axi_periph (base_ps7...	3748	5888	61	61	1680	3389	359	1731	0	0
iop_pmodb (iop_pmodb_imp...	3488	3885	128	4	1289	3332	156	1433	16	0
iop_pmoda (iop_pmoda_imp...	3486	3885	128	4	1322	3330	156	1425	16	0
trace_analyzer_pi (trace_anal...	1644	2545	0	0	689	1565	79	1022	3	0
trace_analyzer_pmodb (trace...	1318	1927	0	0	523	1244	74	796	2	0

The cost of handling the interfaces is modest

The Python programmer’s view

Load *base* overlay on PL

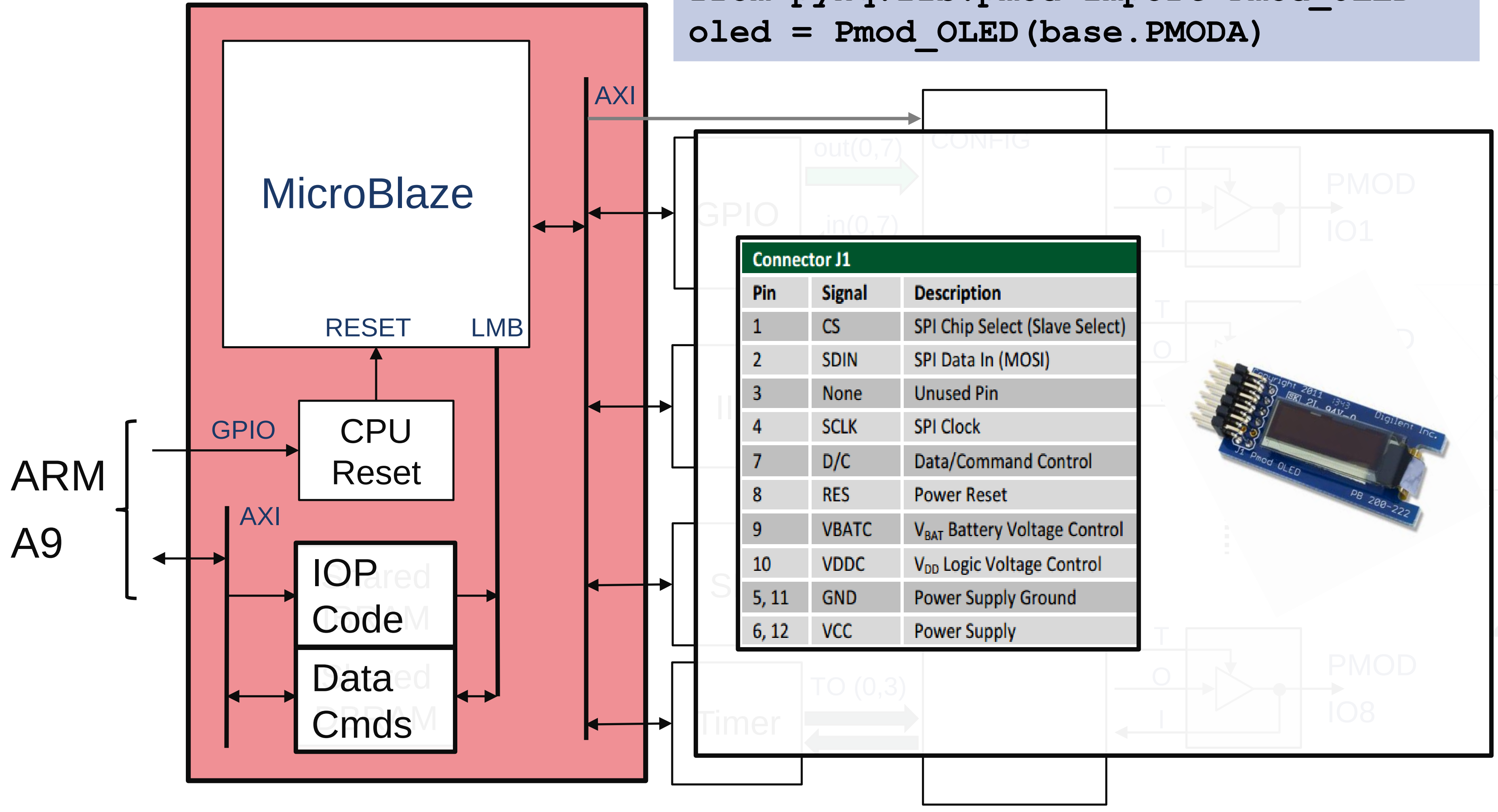


Python code on ARM A9:

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

Configure IO Processor

```
from pynq.lib.pmod import Pmod_OLED
oled = Pmod_OLED(base.PMODA)
```

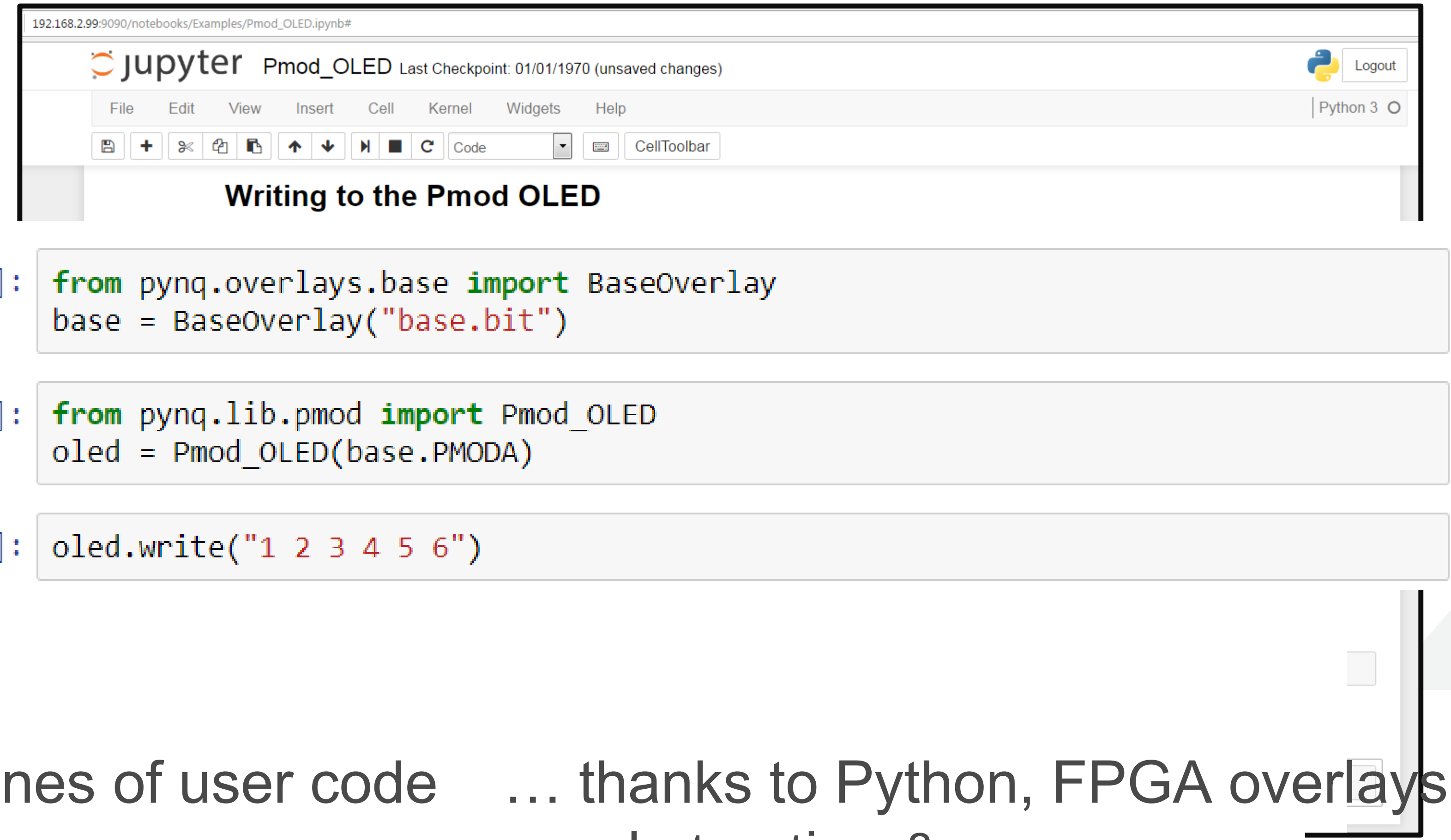


Connector J1

Pin	Signal	Description
1	CS	SPI Chip Select (Slave Select)
2	SDIN	SPI Data In (MOSI)
3	None	Unused Pin
4	SCLK	SPI Clock
7	D/C	Data/Command Control
8	RES	Power Reset
9	VBATC	V_{BAT} Battery Voltage Control
10	VDDC	V_{DD} Logic Voltage Control
5, 11	GND	Power Supply Ground
6, 12	VCC	Power Supply

```
oled.write("1 2 3 4 5 6")
```

Jupyter Notebook


192.168.2.99:9090/notebooks/Examples/Pmod_OLED.ipynb#
jupyter
Pmod_OLED Last Checkpoint: 01/01/1970 (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Python 3
Code
CellToolbar

Writing to the Pmod OLED

```
In [1]: from pynq.overlays.base import BaseOverlay
        base = BaseOverlay("base.bit")

In [2]: from pynq.lib.pmod import Pmod_OLED
        oled = Pmod_OLED(base.PMODA)

In [3]: oled.write("1 2 3 4 5 6")
```

5 lines of user code … thanks to Python, FPGA overlays, abstraction & re-use

Summary

- Overlay Concept
- *base* Overlay
- IOPs
- Using Overlays
- Labs
  - Grove temperature sensor
  - Pmod OLED
  - Grove LEDBar (optional)
  - Grove light sensor (optional)

Questions?

Adaptable.

Intelligent.

XILINX®

PYNQ™