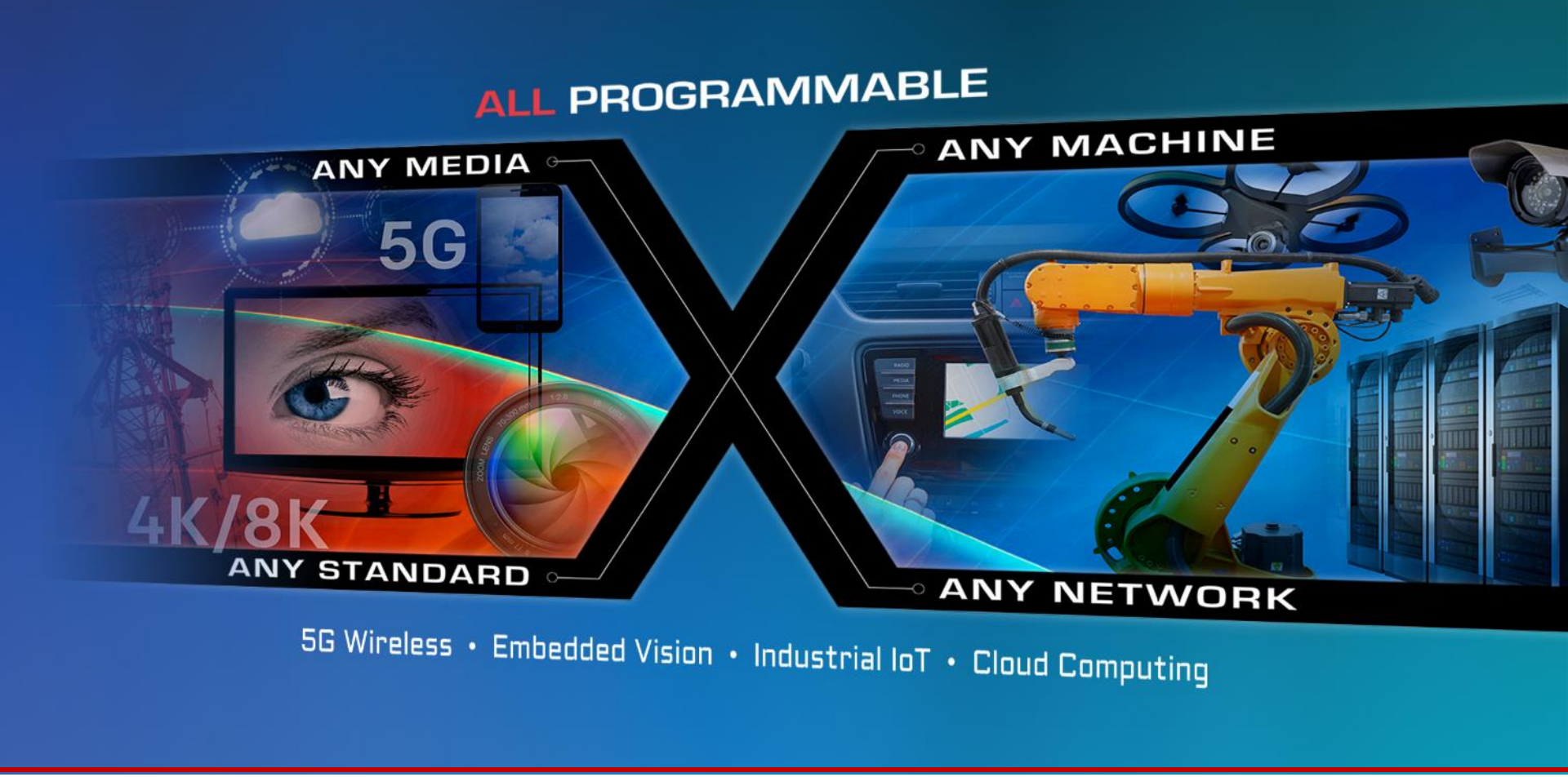


XILINX
ALL PROGRAMMABLE™

PYNQ™

Introduction to Overlays

XILINX
ALL PROGRAMMABLE™

Agenda

- Overlay Concept
- External devices support
- *base* Overlay
- Python programmer's view

FPGA overlays – hardware libraries

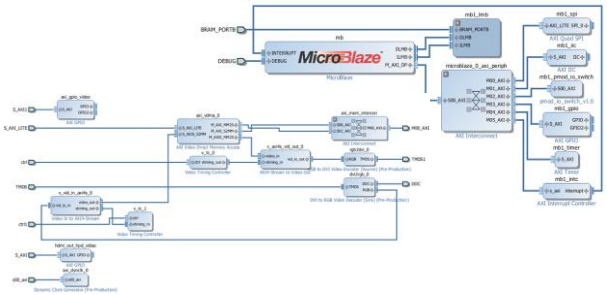
- Overlays are generic FPGA designs that target multiple users with new design abstractions and tools

- 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

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

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) >>
    if((count==0) || (count>253)) {
        // clear bit[0] to indicate
        // set rest to 1s to indic
        MAILBOX_CMD_ADDR = 0xffffff
        return -1;
    }
    for(i=0; i<count; i++) {
        if (cmd & 0x08) // Python
        {
            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, D_GPIO);
   // Initialization
   oled_init();
```

Step 3:

Wrap the C API to create a Python library

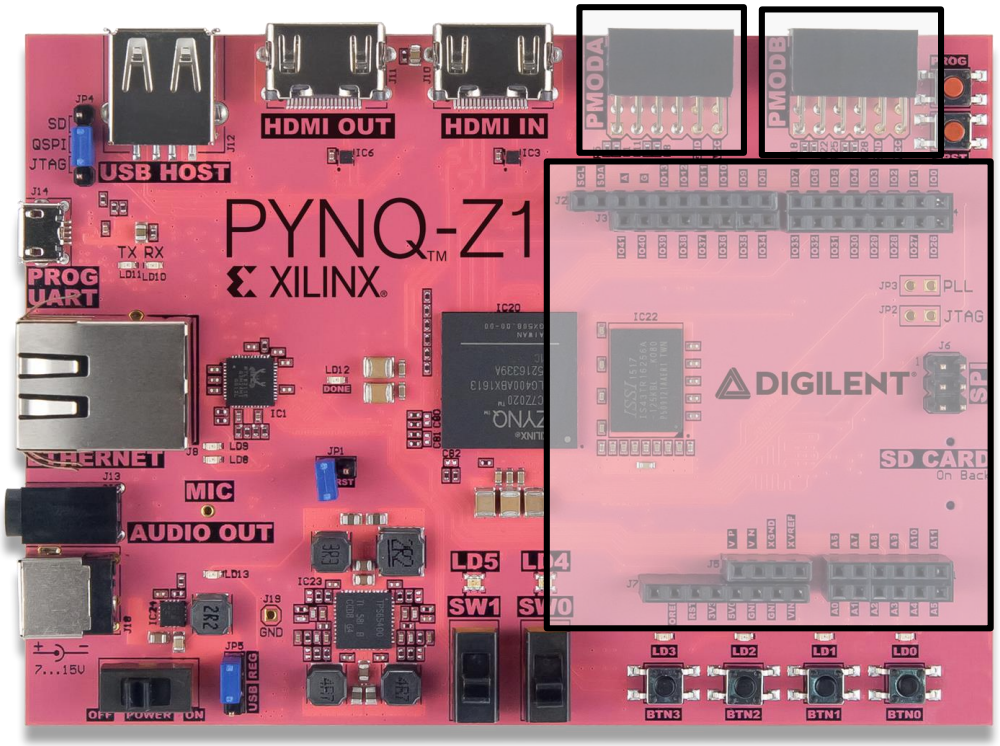
```
from time import sleep
from pynq import Overlay
from pynq.iop import PMOD_ADC, PMOD_DAC

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

Low-cost PYNQ-Z1: Pmod and Arduino Interfaces



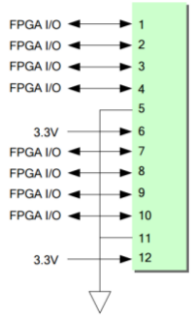
Pmods

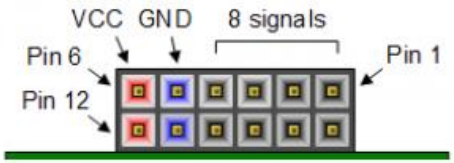
Arduino

Typically every Pmod or Arduino module requires a separate bitstream

Pmod: many physical & electrical instances

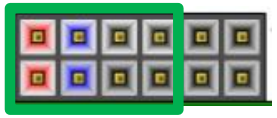
Pmod





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)

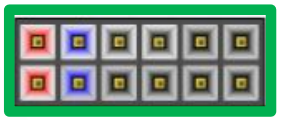
ADC



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

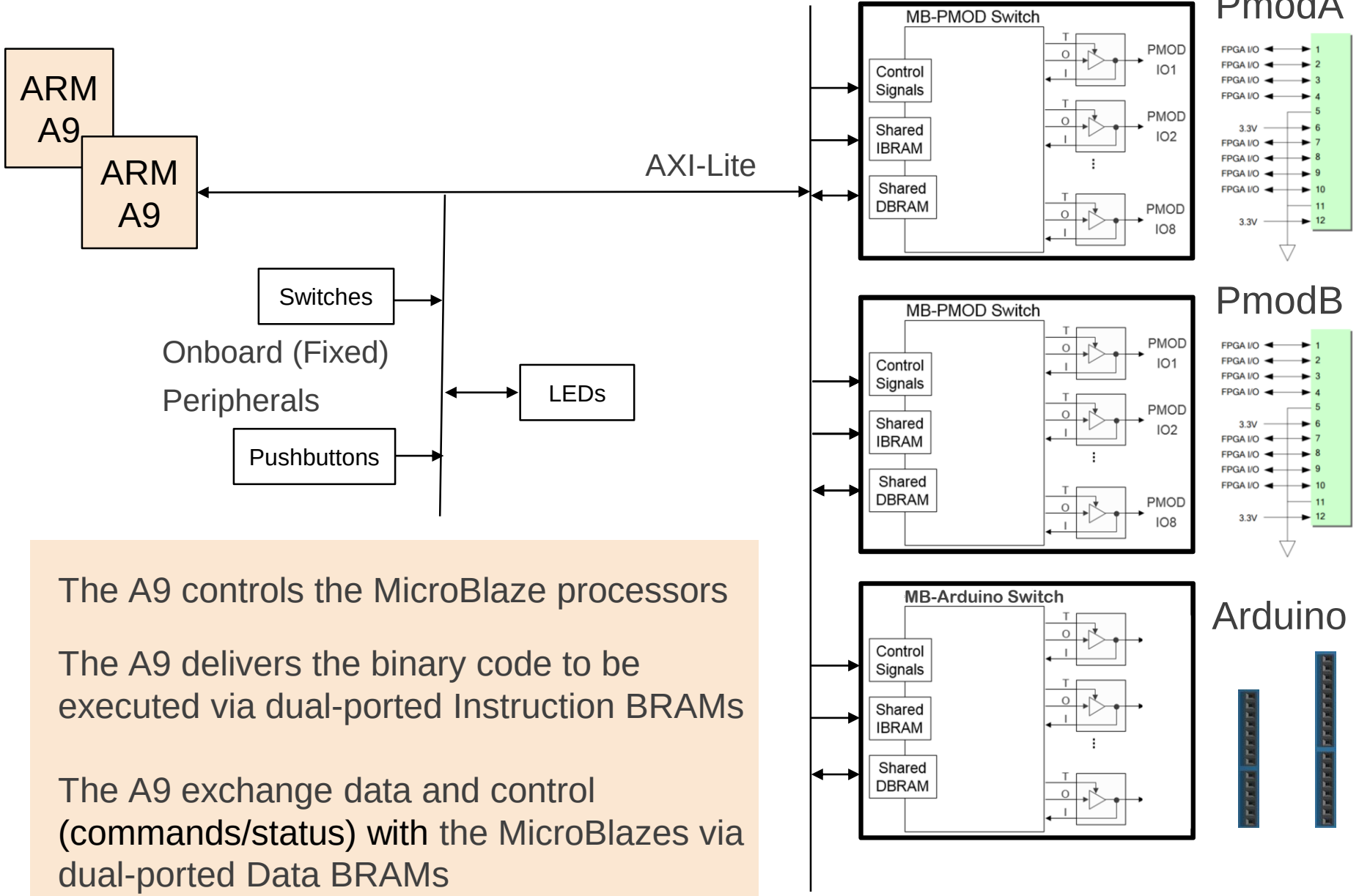


Pin	Signal	Description
1	~CS	Chip Select
2	MOSI	Master-Out-Slave-In
3	(NC)	Not Connected
4	SCLK	Serial Clock
5	GND	Power Supply Ground
6	VCC	Power Supply (3.3V/5V)

DAC

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

base Overlay MicroBlaze IO Processor (IOP)

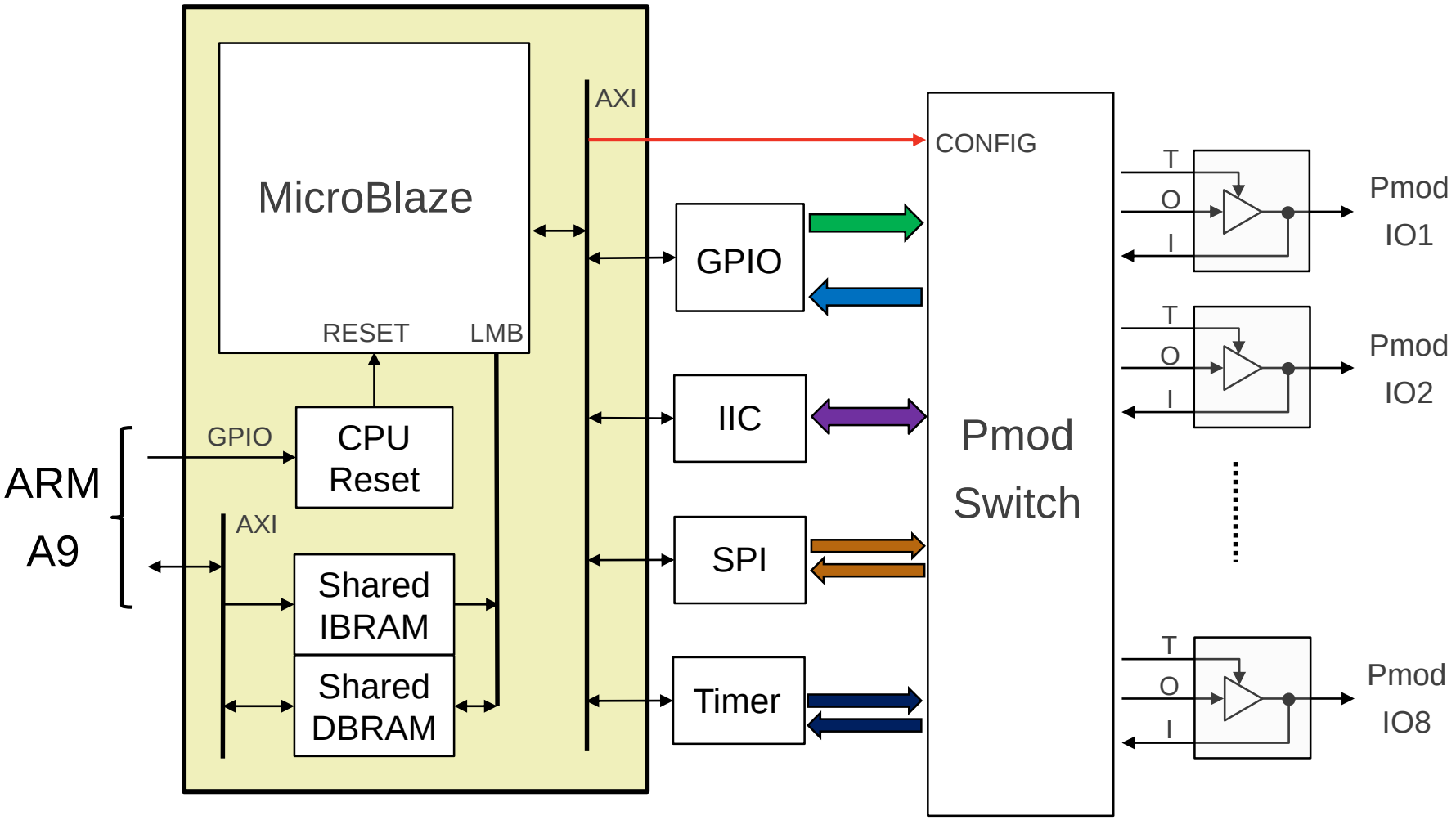


The A9 controls the MicroBlaze processors

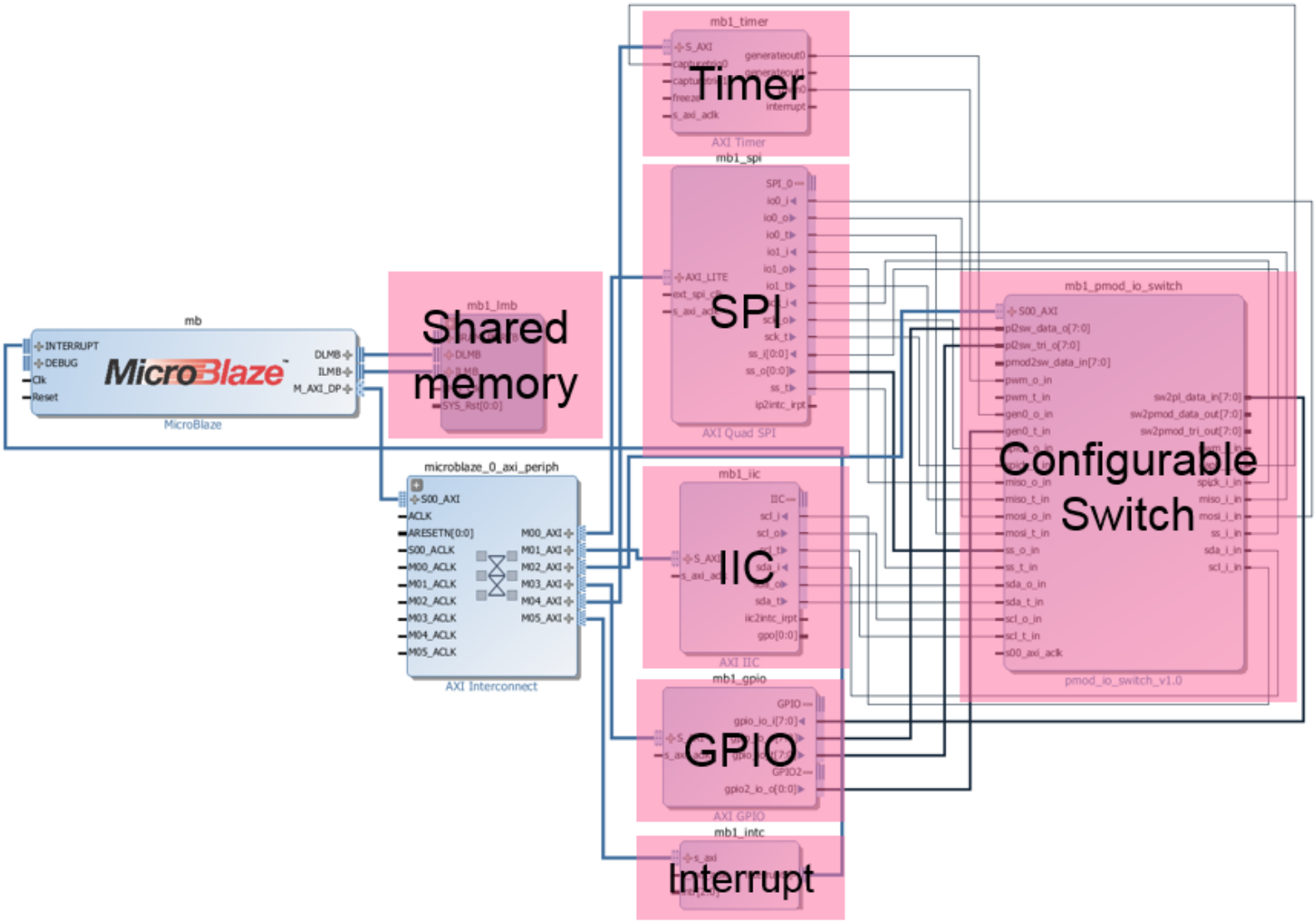
The A9 delivers the binary code to be executed via dual-ported Instruction BRAMs

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

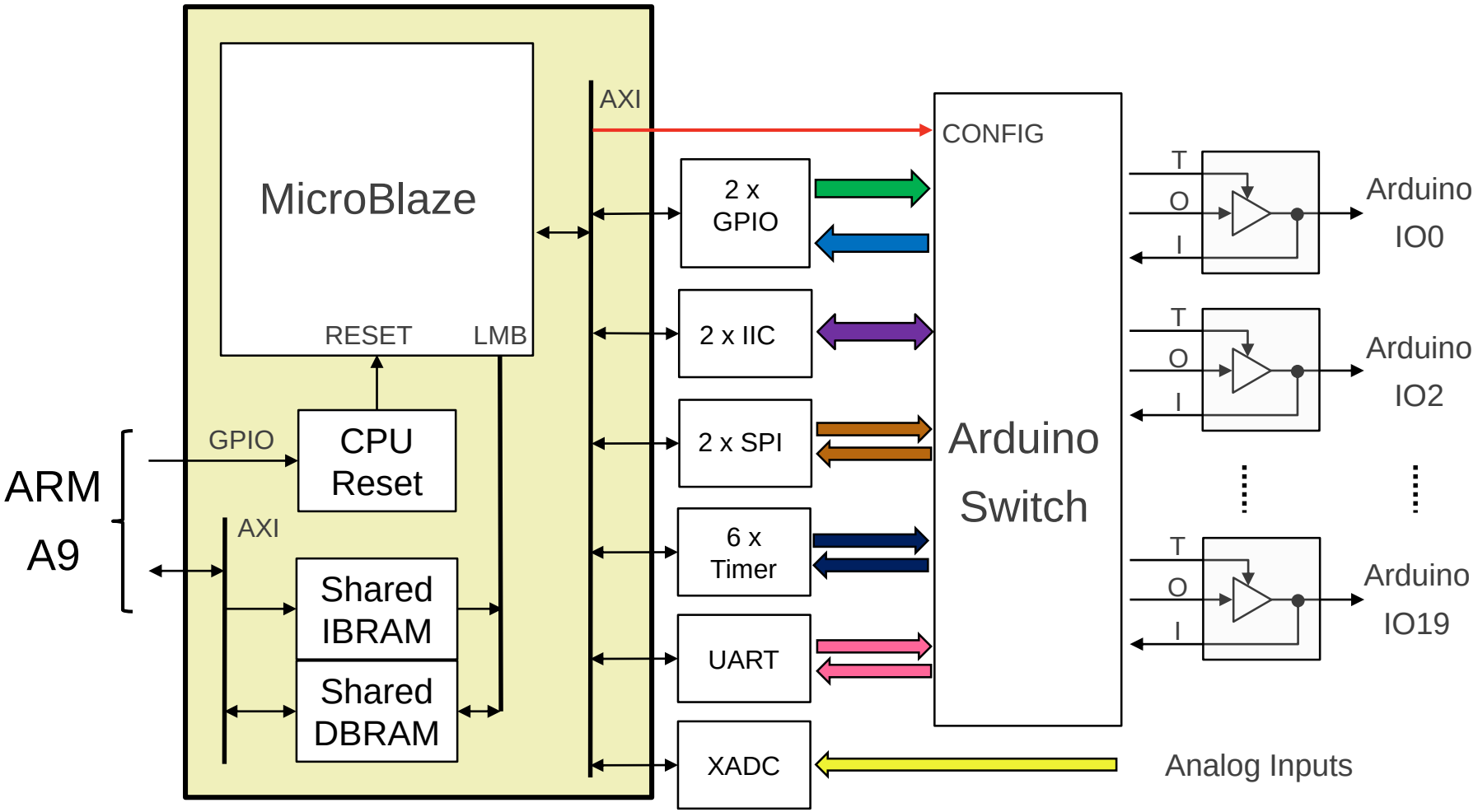
Pmod IO Processor



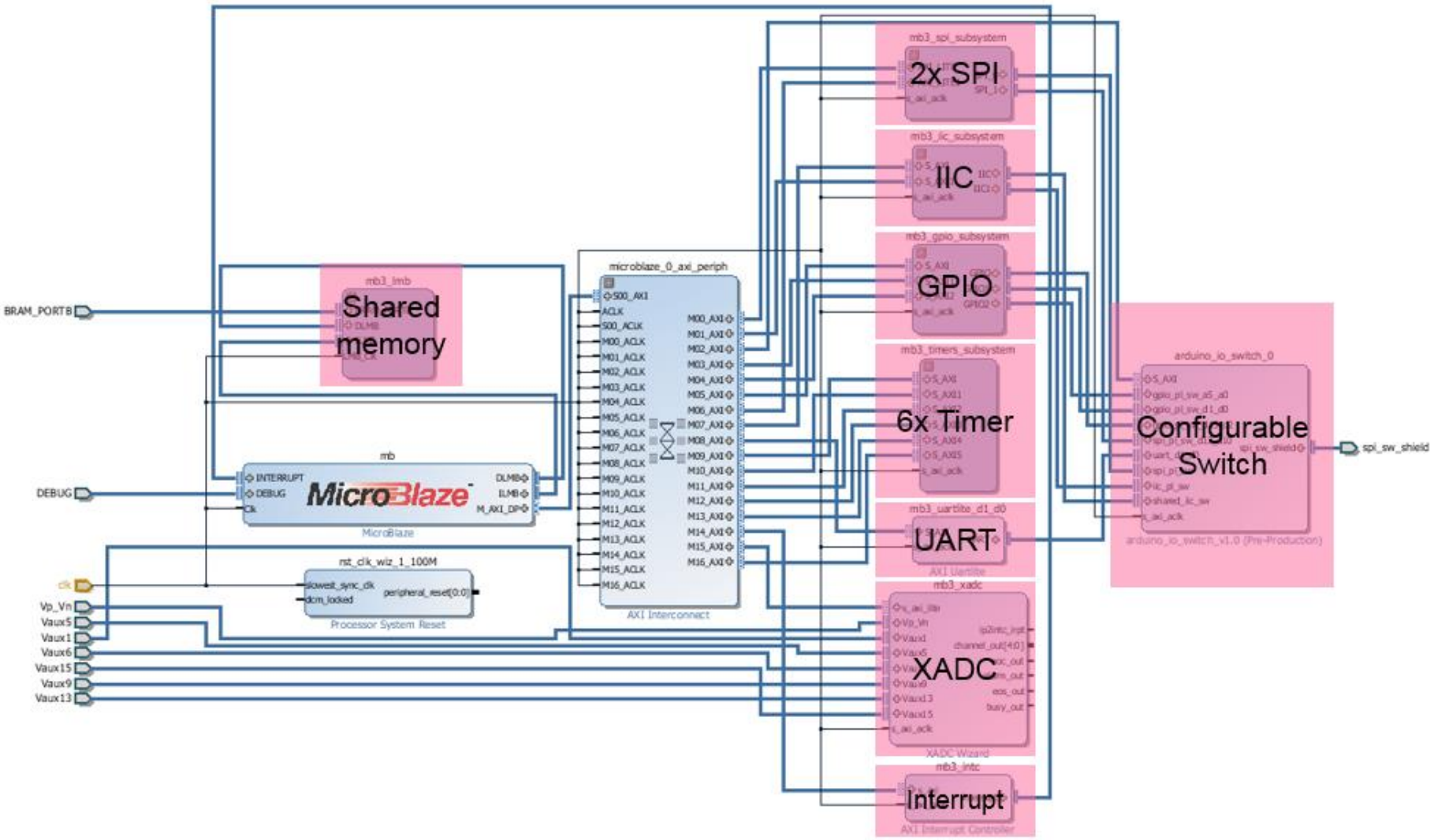
Pmod IOP



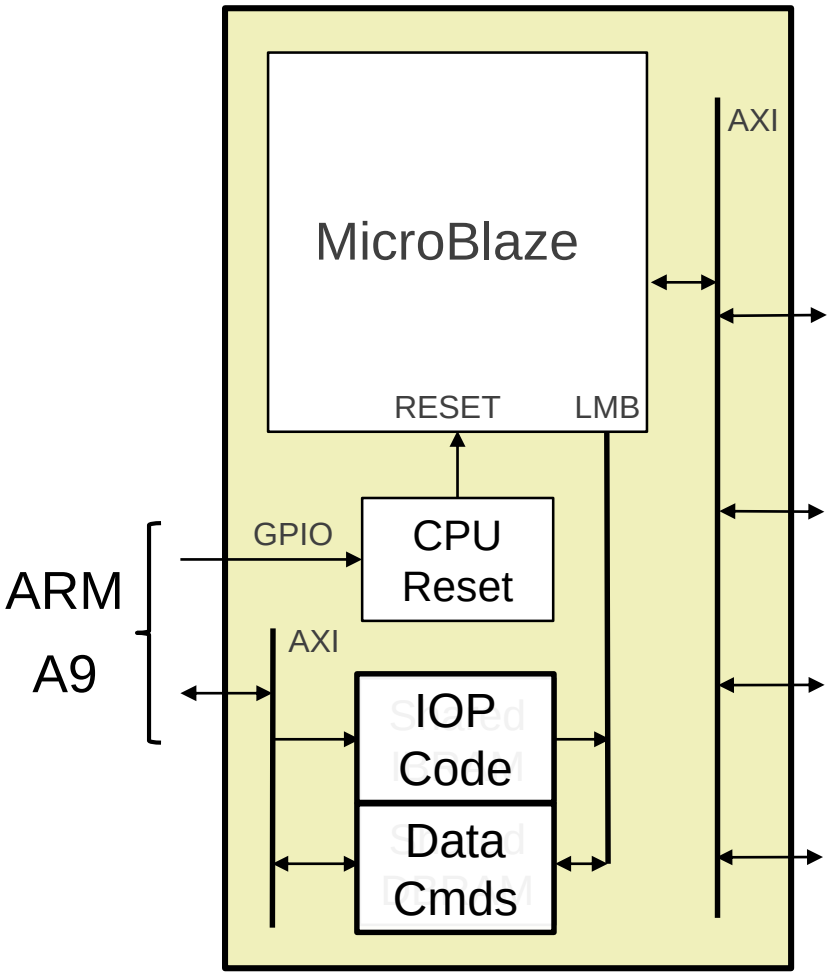
Arduino IO Processor



Arduino IOP



Soft Processor Subsystem (SPS)



The A9 controls the MicroBlaze processors

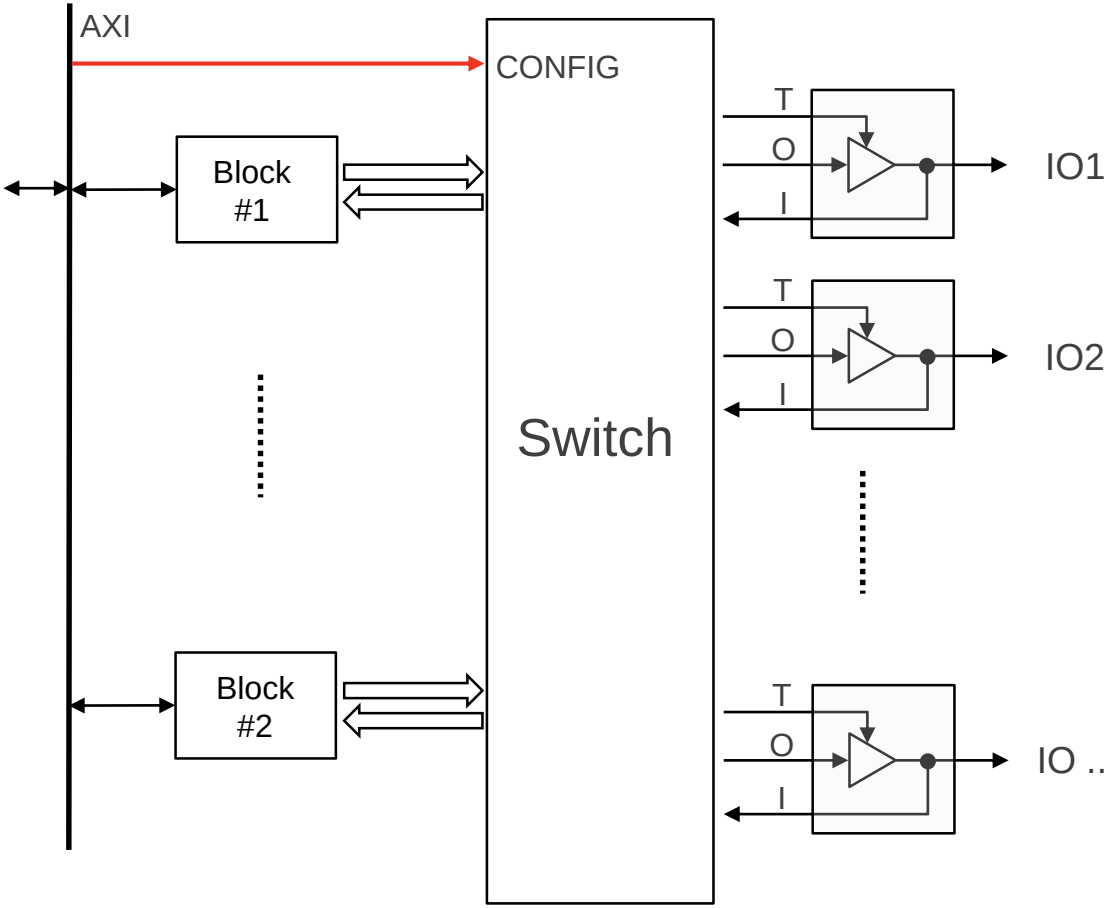
The A9 delivers the binary code to be executed via dual-ported Instruction BRAMs

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

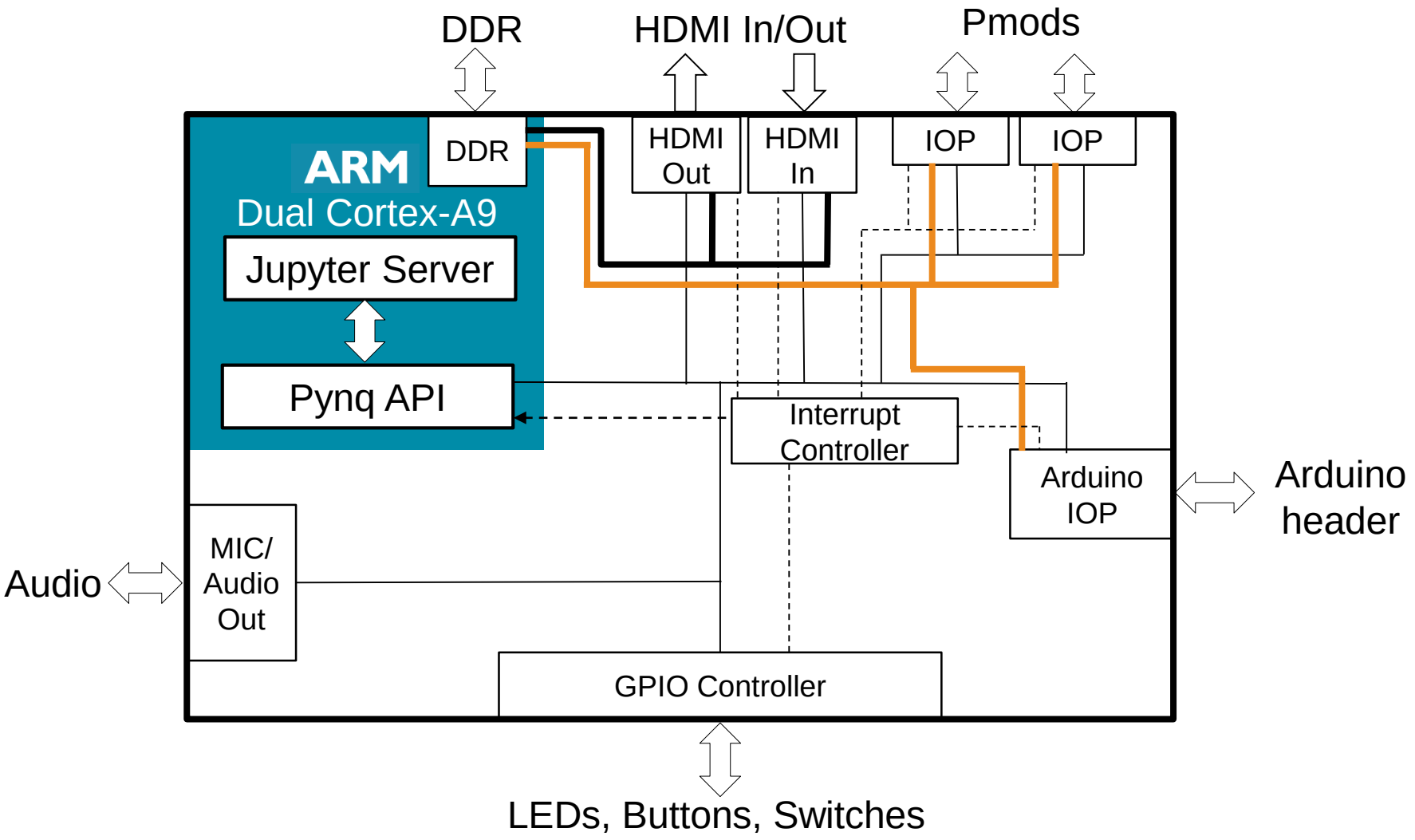
IO Switch (IOS)



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

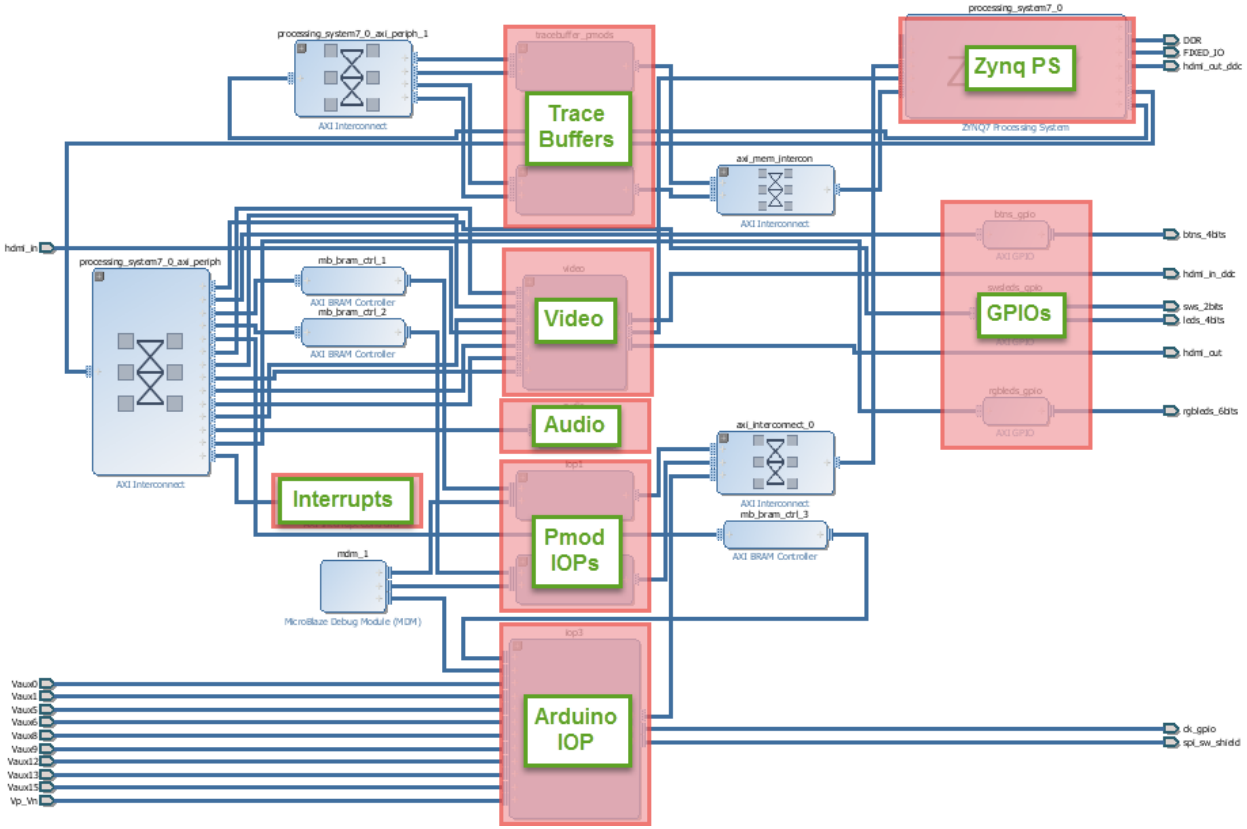
The rest of the *base* Overlay

Complete *base* Overlay (base.bit)



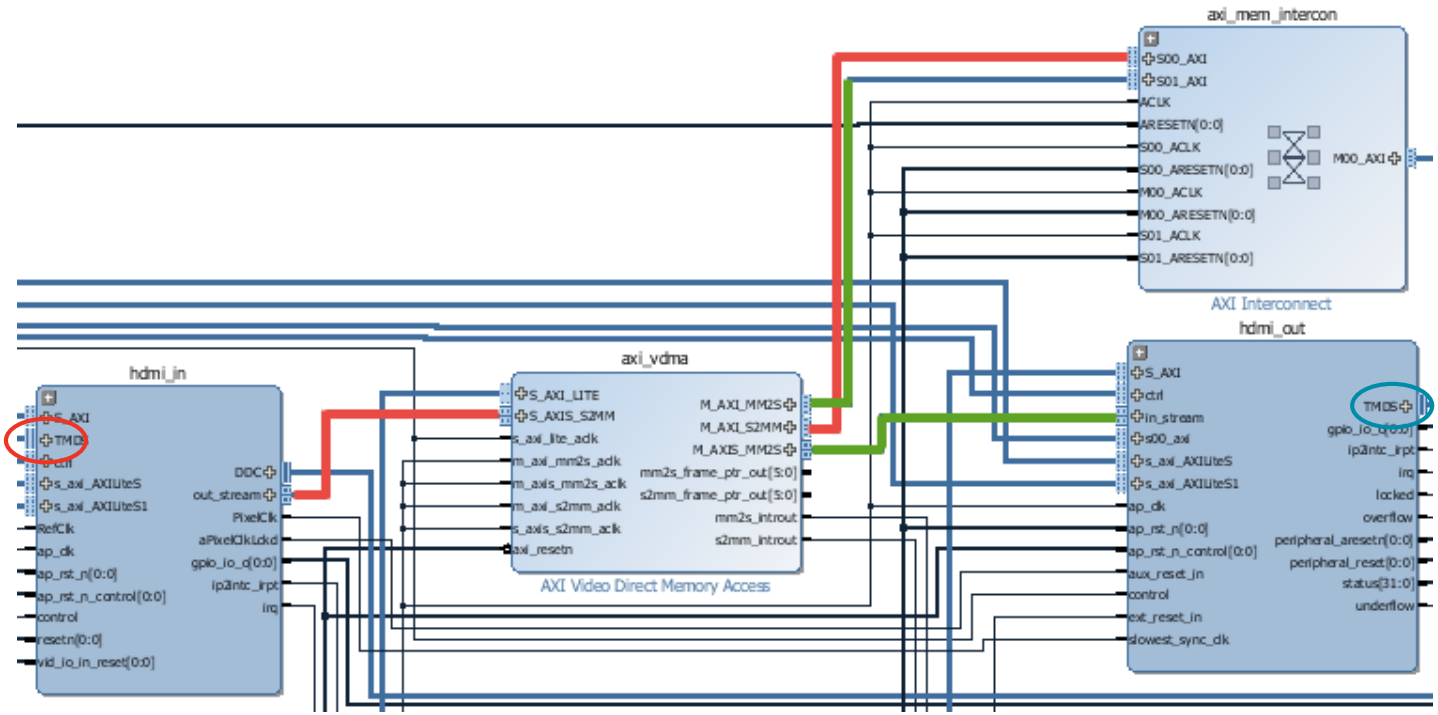
base Overlay – Vivado Interface View

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



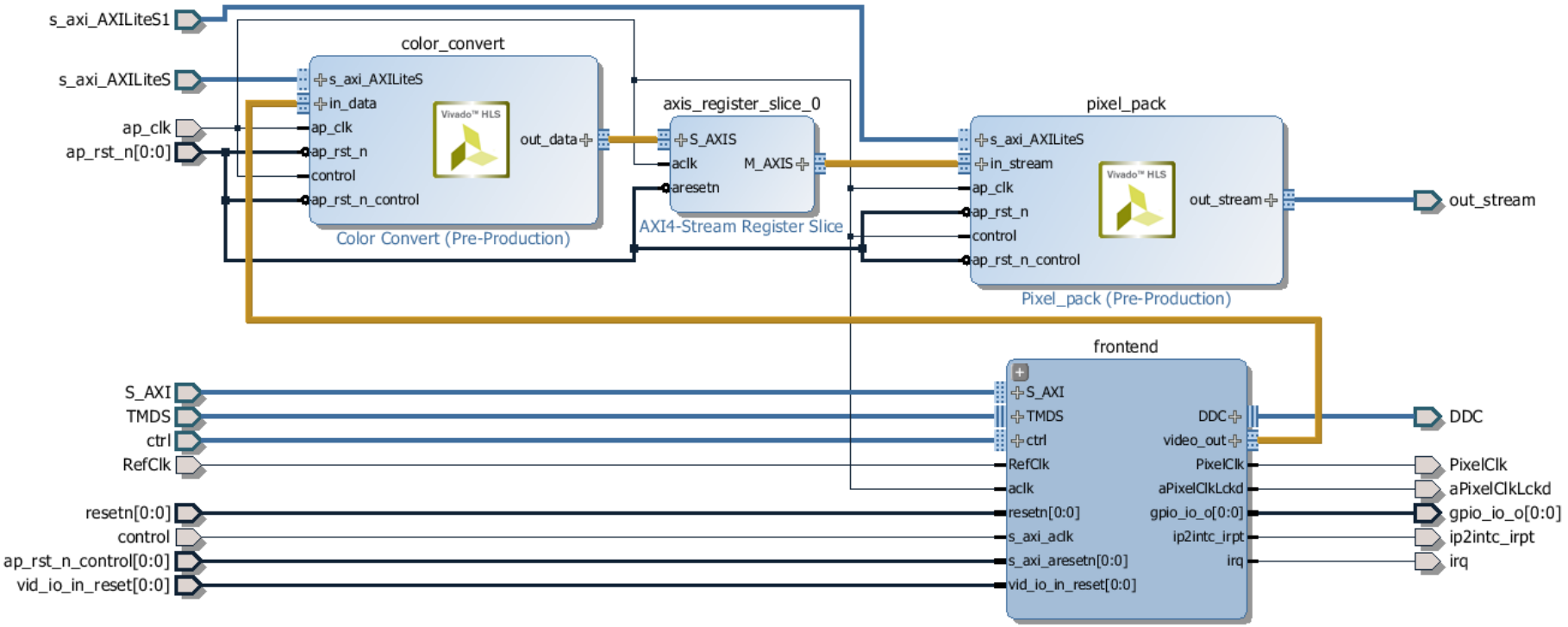
Video

- 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



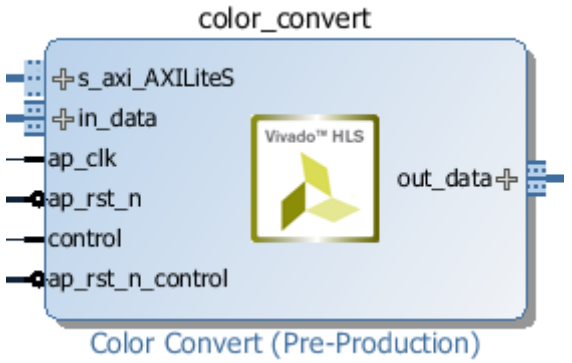
Video In path

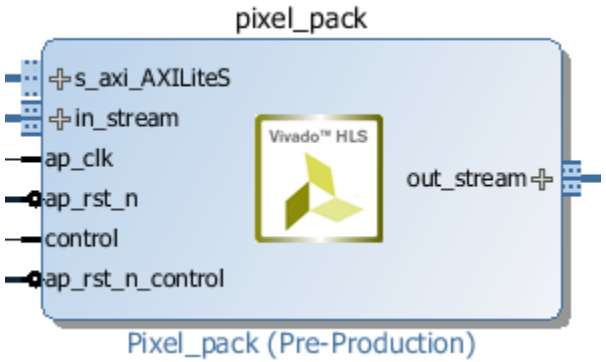
- HDMI_in (TMDS) -> frontend -> color_convert -> axis_register_slice -> pixel_clock -> axis_vdma -> HP0 (PS7)
  - The frontend module wraps all of the clock and timing logic



Color conversion and pixel packing

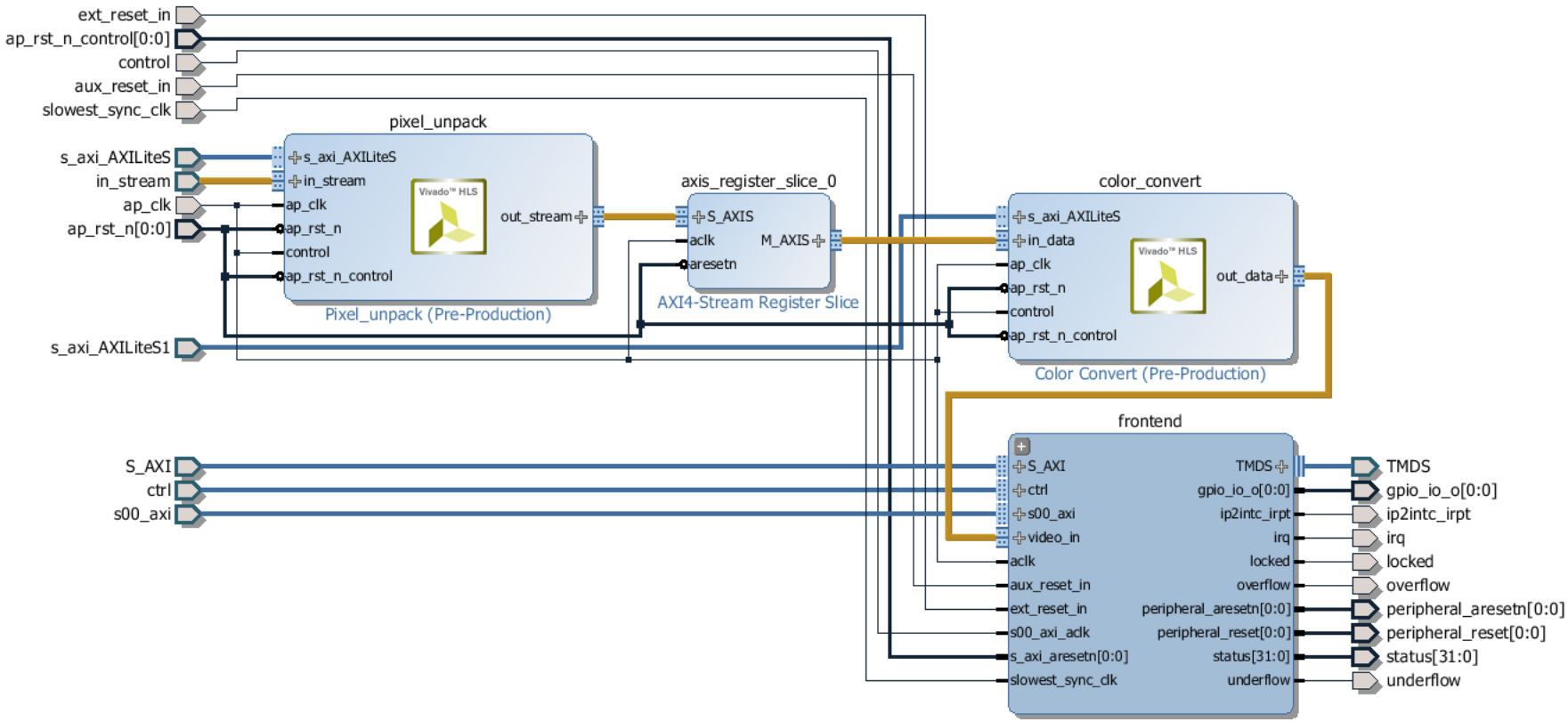
- color_convert
  - Transform the input signal into different color spaces
- pixel_pack
  - Convert between 8/24/32-bit
- Default: BGR (24-bit)
  - RGB (24-bit)
  - RGBA (32-bit)
  - BGR (24-bit)
  - YCbCr (24-bit)
  - Grayscale (8-bit)
- HLS source available





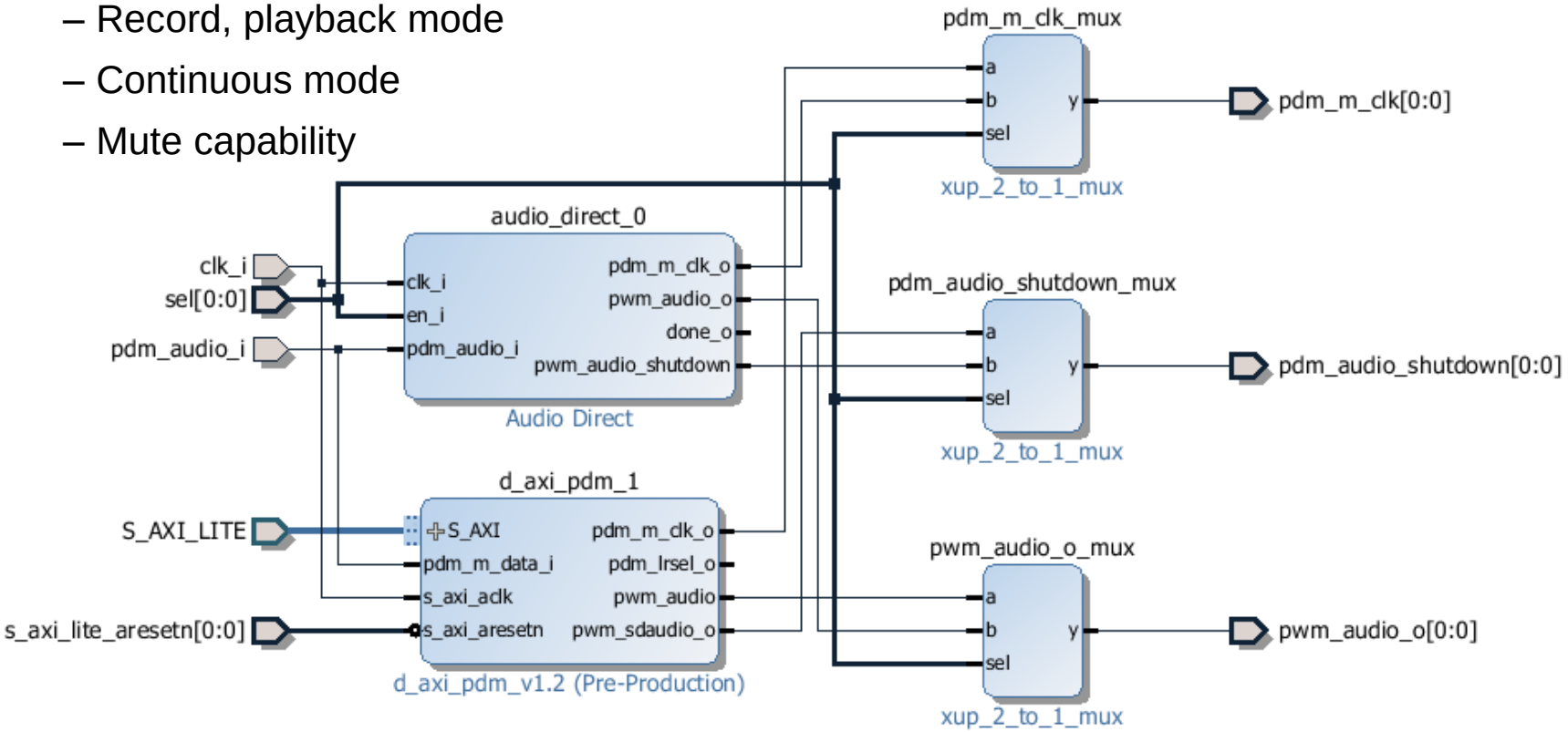
Video Out path

- 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



Audio

- Audio controller
  - MIC in (PDM)
  - Line Out (PWM)
  - Record, playback mode
  - Continuous mode
  - Mute capability



Base Overlay Resource Utilization

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

IOP (Arduino) ~ 12%

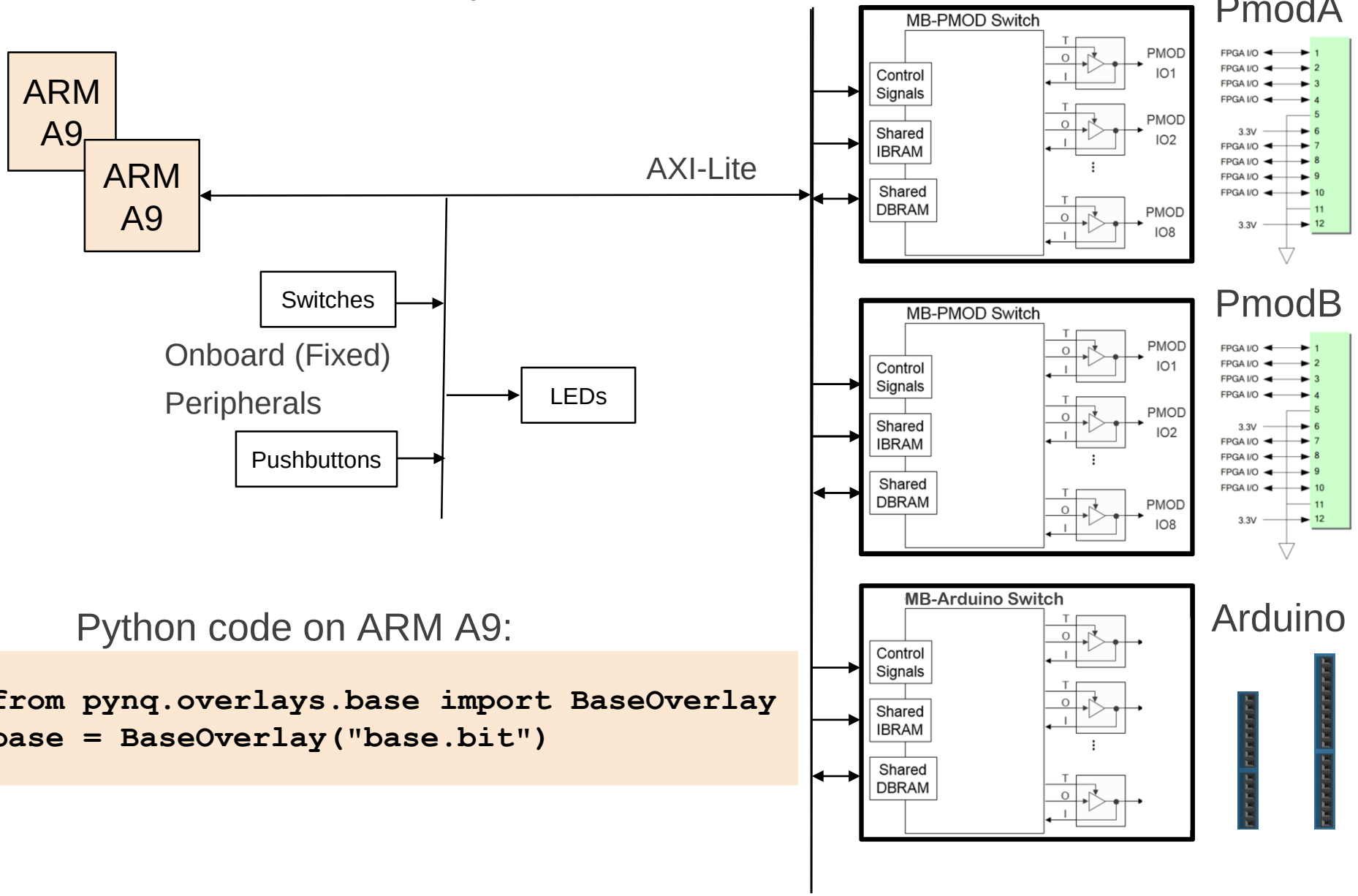
Audio ~ 2%

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)
top	26821	33503	797	121	10603	24628	2193	12697	58	18
system_i (system)	26821	33503	797	121	10603	24628	2193	12697	58	18
audio (audio_imp_P...	1238	639	64	32	440	534	704	255	0	0
audio_path_sel (syst...	0	0	0	0	0	0	0	0	0	0
axi_interconnect_0 (...	177	134	0	0	78	177	0	90	0	0
btns_gpio (system_b...	75	99	0	0	30	75	0	60	0	0
concat_interrupts (s...	0	0	0	0	0	0	0	0	0	0
iop1 (iop1_imp_2R...	3252	2996	123	2	1084	3095	157	1344	16	0
iop2 (iop2_imp_GME...	3255	2996	123	2	1157	3098	157	1336	16	0
iop3 (iop3_imp_17Q...	6389	6247	134	4	2346	6195	194	3103	16	0
video (video_imp_1E...	8322	15086	292	20	4157	7757	565	4468	10	18

The cost of handling all Pmod and Arduino instances is modest

The Python programmer's view

Load *base* overlay on PL

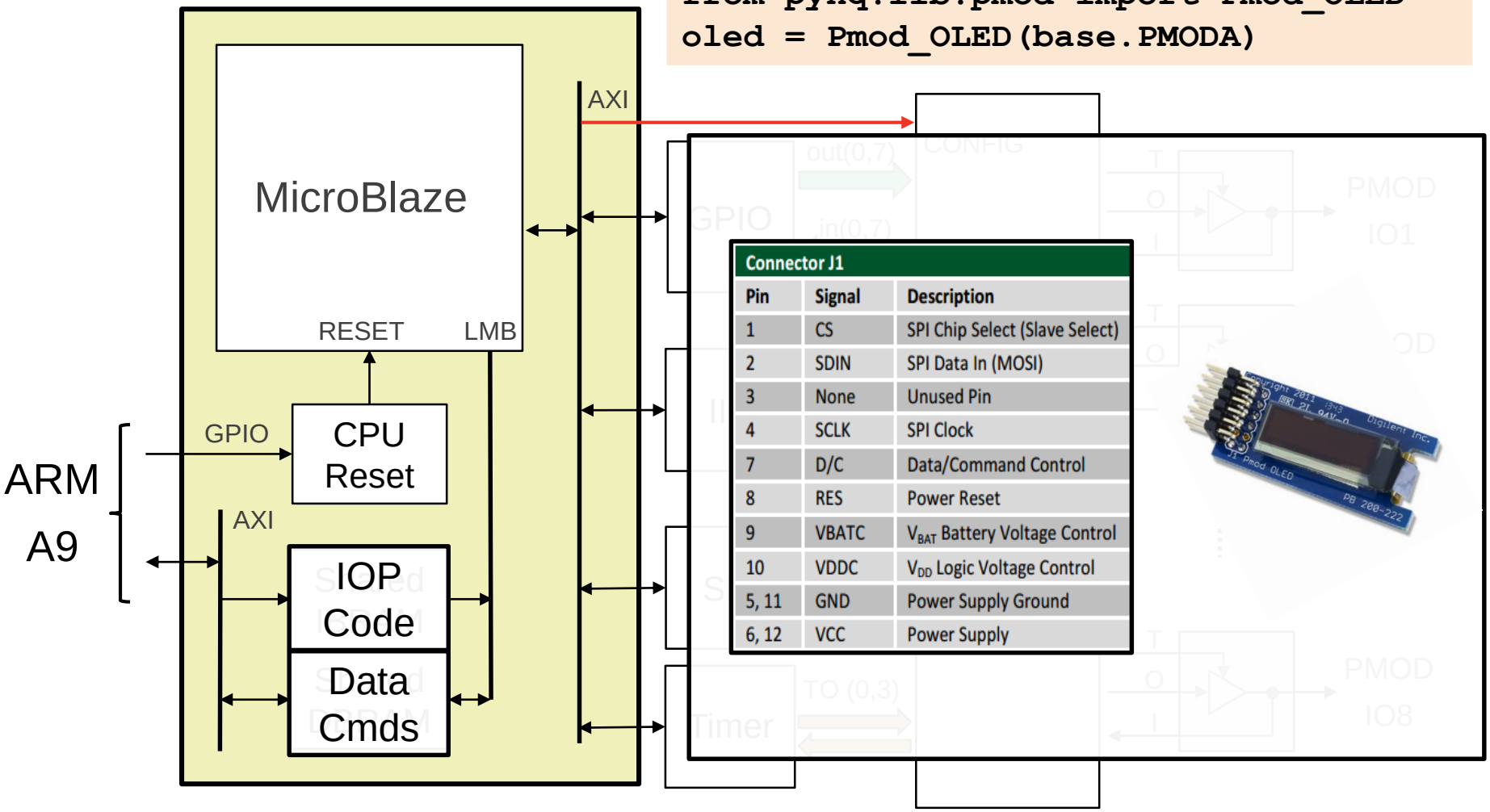


Python code on ARM A9:

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

Configure IO processor

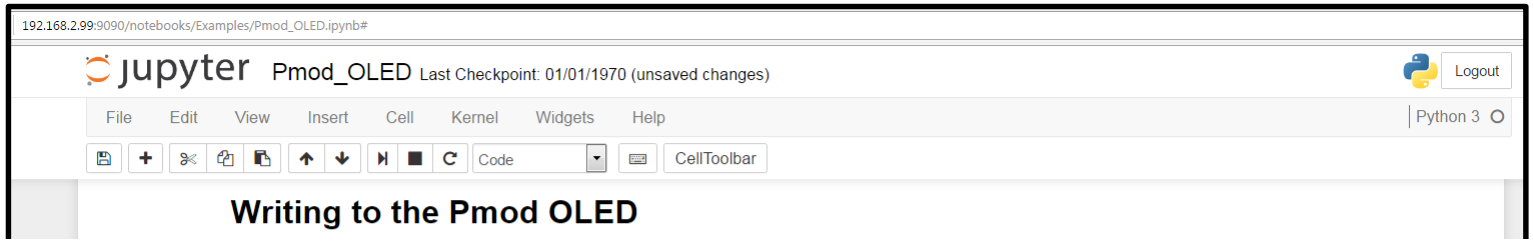
```
from pynq.lib.pmod import Pmod_OLED
oled = Pmod_OLED(base.PMODA)
```



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



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?