

EE3220 System-on-Chip Design

Lecture Note 9

SoC IP and General Purpose I/O (GPIO)

Overview

- Using IP in SoC Designs
 - IP Catalog and directory
 - IP device files
 - GP Interfaces
 - Adding IP to extend PS into PL
 - Bitstream generation
- GPIO
 - How do we make a program light up LEDs in response to a switch?
 - Basic Concepts
 - Port Circuitry
 - Alternate Functions
 - Peripheral Access In C
- PYNQ Arduino System and Grove I/O Kit

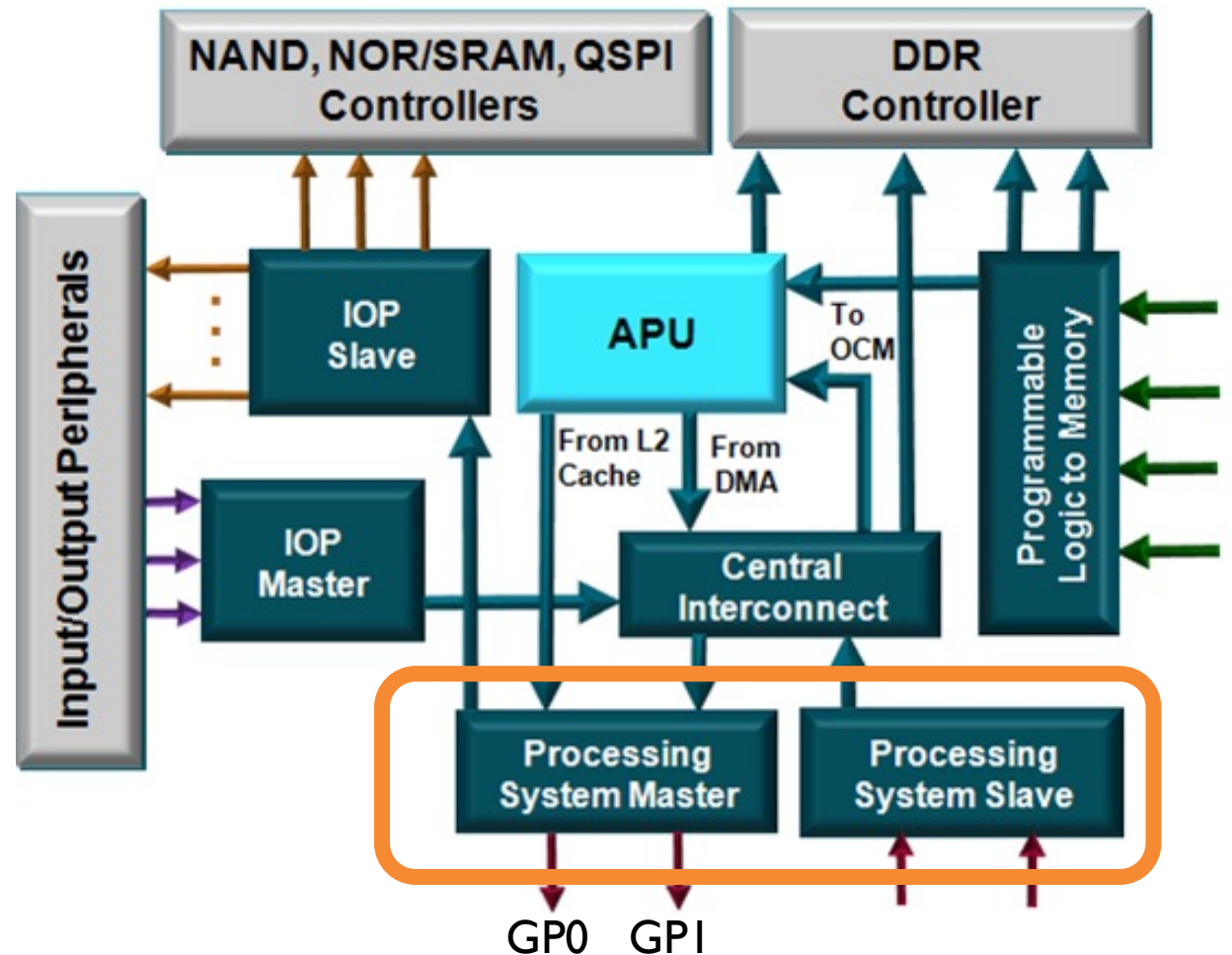
System-on-Chip Intellectual Property (IP)

The PS and the PL

- The Zynq-7000 AP SoC architecture consists of two major sections
 - PS: Processing system
 - Dual ARM Cortex-A9 processor based
 - Multiple peripherals
 - Hard silicon core
 - **PL: Programmable logic**
 - Shares the same 7 series programmable logic as
 - Artix™-based devices: Z-7010, Z-7015 and Z-7020 (high-range I/O banks only)
 - Kintex™-based devices: Z-7030, Z-7035, Z-7045, and Z-7100 (mix of high-range and high-performance I/O banks)
- This section focuses on the PL

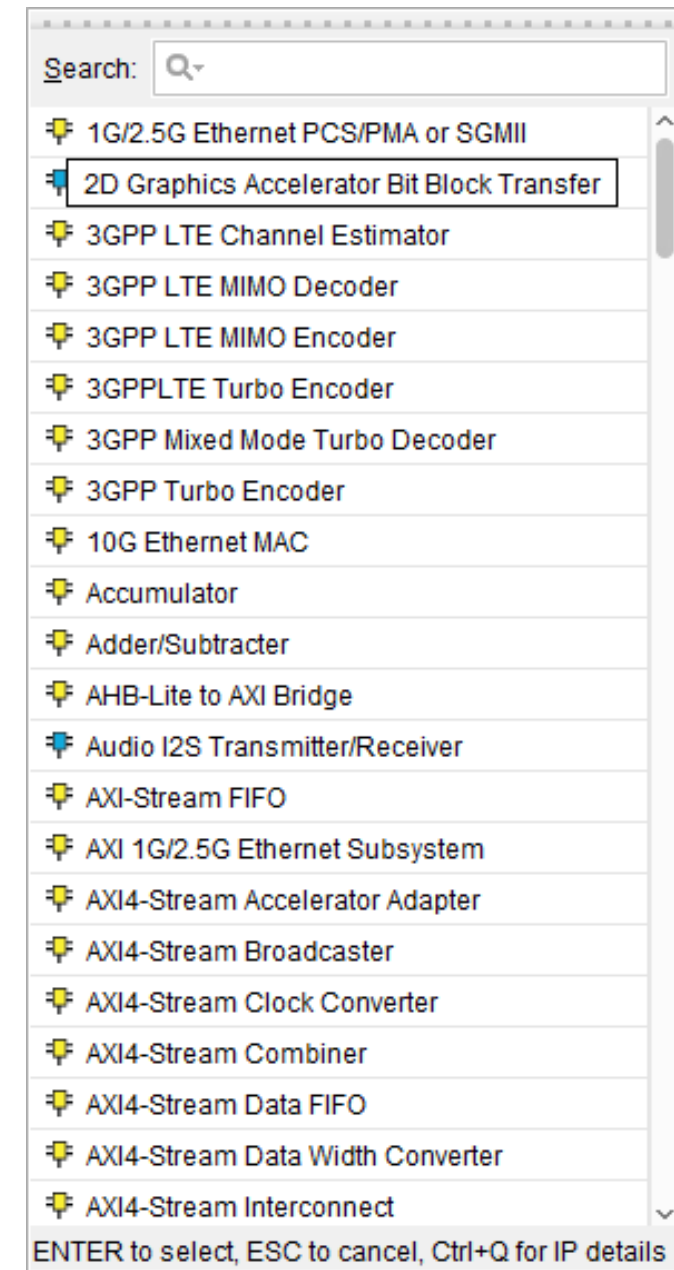
Communicating with PL

- Processing system master
 - Two ports from the processing system to programmable logic
 - Connects the CPU block to common peripherals through the central interconnect
- Processing system slave
 - Two ports from programmable logic to the processing system
- Slave PL peripherals address range
 - 4000_0000 and 7FFF_FFFF (connected to GP0) and
 - 8000_0000 and BFFF_FFFF (connected to GPI)



IP Catalog

- The IP Catalog contain a collection of IP that can be used to assemble the (embedded) system
- Supported by IPI
- Facilitates quick system construction
- Each IP block has its own configuration parameters
- Most of the IP is free, some require licenses
- Stored as source code in the install directory
 - Always synthesized with the latest tools
 - Some proprietary source code is encrypted
- Peripherals in the PS are always present and can be dynamically enabled or disabled through PS Configuration wizard

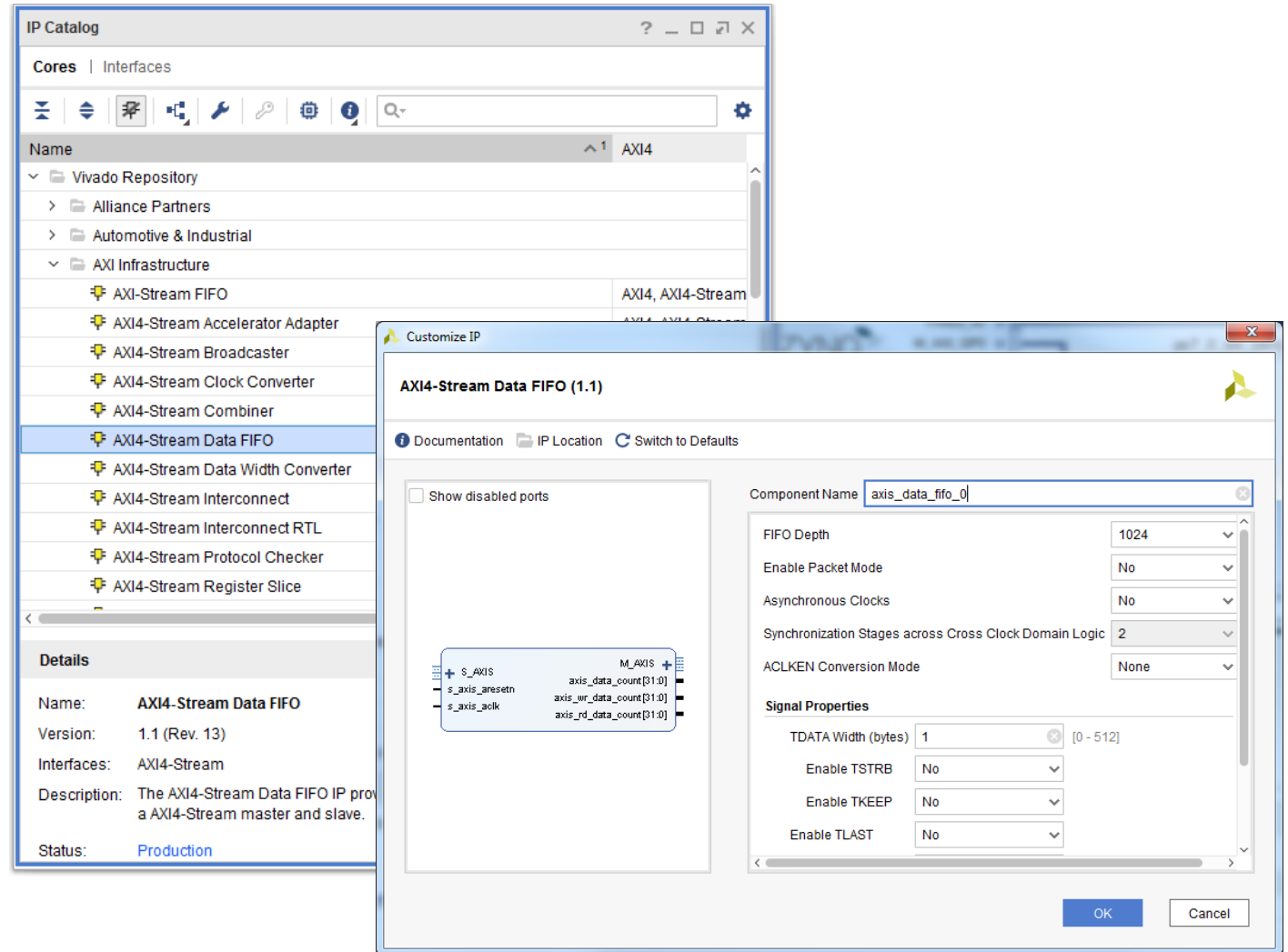


IP Peripherals - Included as Source (Free)

- Bus and bridge controllers
 - AXI to AXI connector
 - Local Memory Bus (LMB)
 - AXI Chip to Chip
 - AHB-Lite to AXI
 - AXI4-Lite to APB
 - AXI4 to AHB-Lite
- Debug cores
 - Integrated Logic Analyzer
- DMA and Timers
 - Watchdog, fixed interval
- Inter-processor communication
- External peripheral controller Memory and memory controller
- High-speed and low-speed communication peripherals
 - AXI 10/100 Ethernet MAC controller
 - Hard-core tri-mode Ethernet MAC
 - AXI IIC
 - AXI SPI
 - AXI UART
- Other cores
 - System monitor
 - Xilinx Analog-to-Digital Converter (XADC)
 - Clock generator, System reset module
 - interrupt controller
 - Traffic Generator, Performance monitor

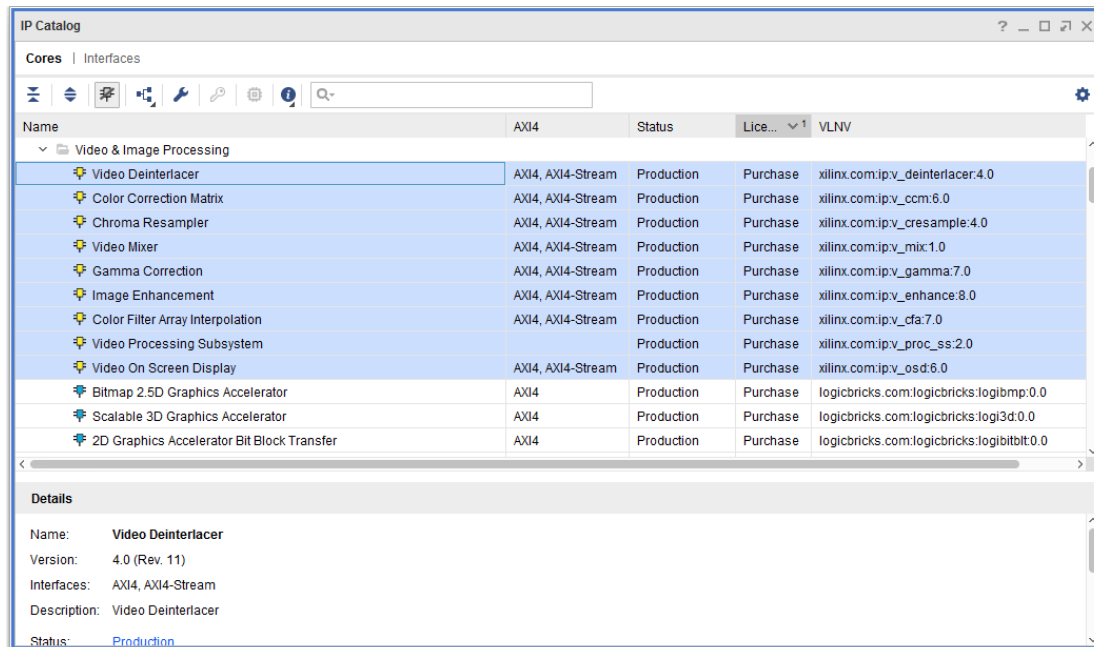
Vivado IP Catalog

- Integrated IP Support
 - Instant access to IP customization
 - Vivado IP GUI look and feel
 - Support for Vivado synthesis and implementation
 - Selectable IP output products
 - Full Tcl support



IP Cores Included as Evaluation

- Xilinx IP
- 3rd Party IP

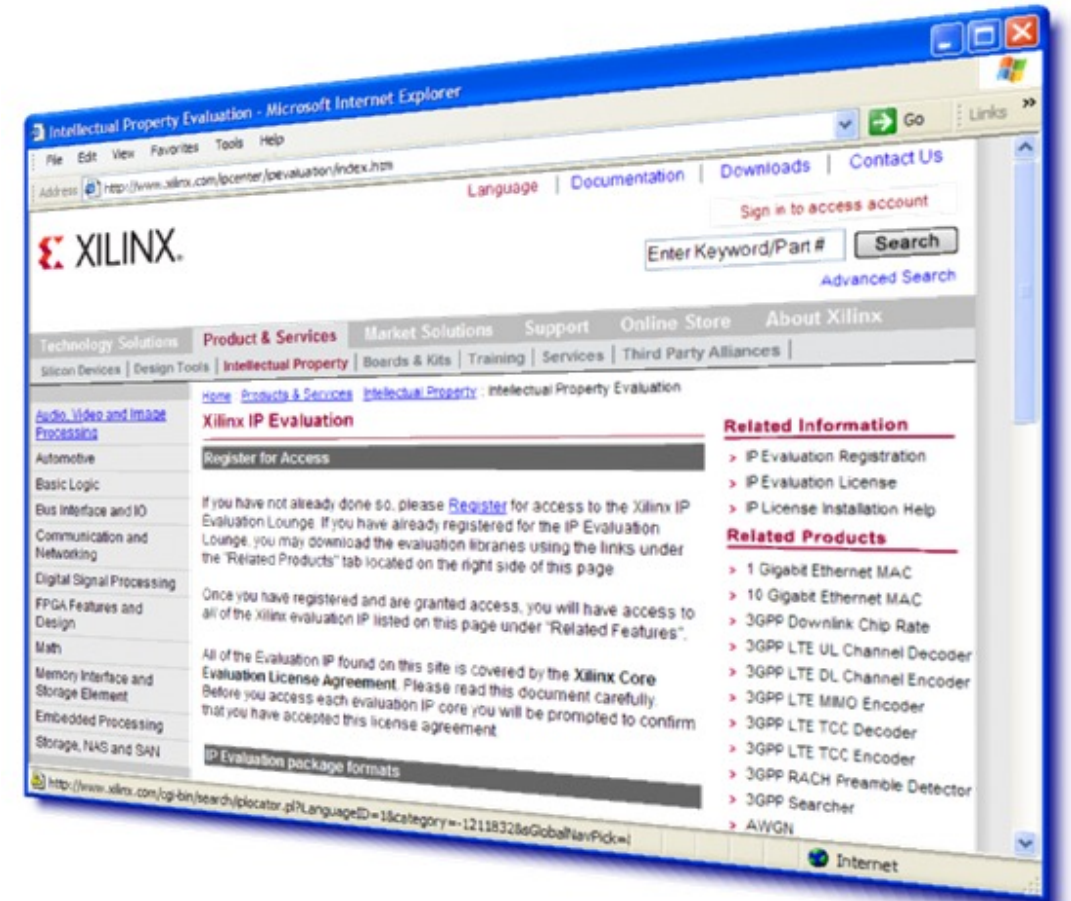


The screenshot shows the Xilinx IP Catalog interface. It features a search bar and a list of IP cores. The table below represents the data shown in the catalog.

Name	AXI4	Status	Lic...	VLNV
Video & Image Processing				
Video Deinterlacer	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_deinterlacer:4.0
Color Correction Matrix	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_ccm:6.0
Chroma Resampler	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_cresample:4.0
Video Mixer	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_mix:1.0
Gamma Correction	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_gamma:7.0
Image Enhancement	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_enhance:8.0
Color Filter Array Interpolation	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_cfa:7.0
Video Processing Subsystem		Production	Purchase	xilinx.com:ip_v_proc_ss:2.0
Video On Screen Display	AXI4, AXI4-Stream	Production	Purchase	xilinx.com:ip_v_osd:6.0
Bitmap 2.5D Graphics Accelerator	AXI4	Production	Purchase	logicbricks.com:logicbricks:logibmp:0.0
Scalable 3D Graphics Accelerator	AXI4	Production	Purchase	logicbricks.com:logicbricks:logi3d:0.0
2D Graphics Accelerator Bit Block Transfer	AXI4	Production	Purchase	logicbricks.com:logicbricks:logibitblt:0.0

Details

Name: Video Deinterlacer
Version: 4.0 (Rev. 11)
Interfaces: AXI4, AXI4-Stream
Description: Video Deinterlacer
Status: Production



Xilinx developed, delivered, and supported Evaluation IP installs with a 90-day evaluation license

IP Cores

- Right click to
 - Add/customize
 - IP Settings
 - Check License
 - Determine compatibility
 - Datasheet
 - Change Log
 - Product Webpage
 - Answer record

The screenshot displays the Vivado IP Catalog interface. The top tabs are 'IP Catalog', 'Diagram', and 'Address Editor'. Below the tabs, there's a search bar with 'pci' entered. The main list shows the hierarchy: 'Vivado Repository' > 'Standard Bus Interfaces' > 'PCI'. Two IP cores are listed: '32-bit Initiator/Target for PCI (7-Series)' and '64-bit Initiator/Target for PCI (7-Series)'. The first one is selected. A right-click context menu is open over the selected IP, showing options: 'Properties...' (Ctrl+E), 'IP Settings...', 'Add Repository...', 'Refresh All Repositories', 'Customize IP...' (with a wrench icon), 'License Status' (with a key icon), 'Compatible Families' (with a gear icon), 'Data Sheet', 'Change Log', 'Product Webpage', 'Answer Records', and 'Export to Spreadsheet...'. Below the list, the 'Details' section for the selected IP is visible, showing fields like Name, Version (5.0 (Rev. 8)), Description, Status (Production), License (Purchase), Change Log (View Change Log), Vendor (Xilinx, Inc.), VLNV (xilinx.com:ip:pci32:5.0), and Repository (C:/Xilinx/Vivado/2017.1/data/ip).

IP Core Information

- The size of each core is available in the data sheet
- For example, the axi_timer_v2_00_a data sheet contains the following table:

Table 2-2: Performance and Resource Utilization: Artix-7 FPGA (XC7A355TDIE) and Zynq-7000 Devices

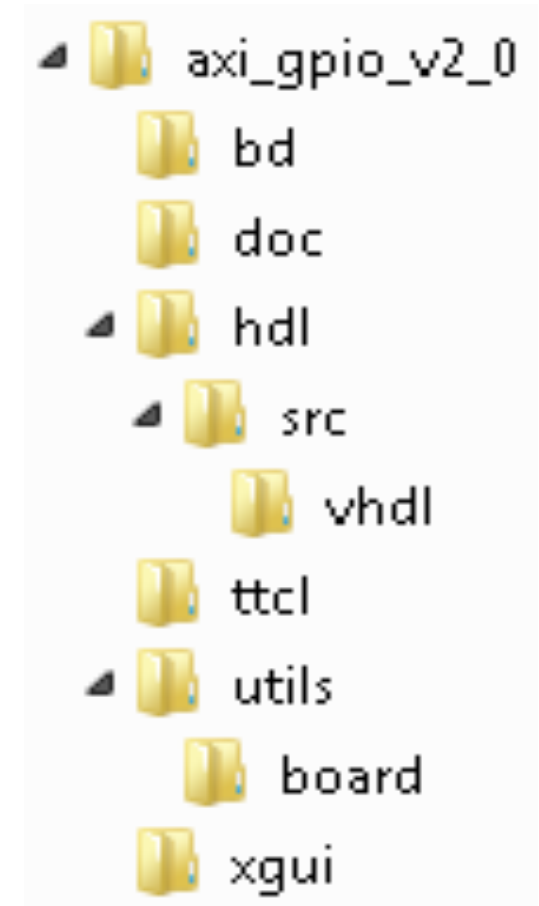
Parameter Values		Device Resources		
Width of Timer/Counter	Enable Timer2	Slices	Flip-Flops	LUTs
8	False	49	53	96
16	False	61	69	120
32	False	84	101	181
8	True	50	74	123
16	True	74	106	161
32	True	97	170	256

Data sheet provided for each core (right-click on core in IP catalog to access)

IP Directory - Peripheral Storage

- IP Core directory (located in the project directory)
 - {component}.xml
 - MyProcessorIPLib directory (user defined)
 - Repository Directory listed using
Project → **Project Options** → **Device and Repository Search** tab
 - %XILINX_INSTALL%\Vivado\{Version}\data\ip

User peripherals can be located in the project directory or a peripheral repository



IP Core files

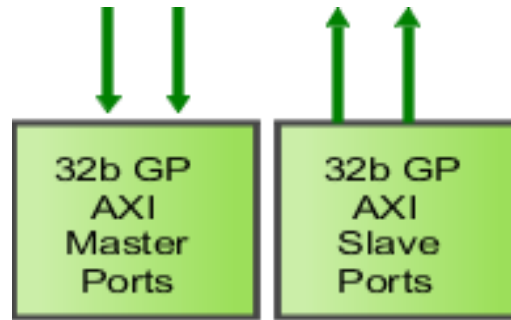
- component.xml
 - XML format
 - Top level folder
 - Provides ports description, parameters and options for IP
 - Links to source files
- xgui folder
 - .tcl file for IPI GUI

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component xmlns:xilinx="http://www.xilinx.com" xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <spirit:vendor>xilinx.com</spirit:vendor>
  <spirit:library>XUP</spirit:library>
  <spirit:name>led_ip</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>S_AXI</spirit:name>
      <spirit:busType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm" spirit:version="1.0"/>
      <spirit:abstractionType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm_rtl" spirit:version="1.0"/>
      <spirit:slave>
        <spirit:memoryMapRef spirit:memoryMapRef="S_AXI"/>
      </spirit:slave>
      <spirit:portMaps>
        <spirit:portMap>
          <spirit:logicalPort>
            <spirit:name>AWADDR</spirit:name>
          </spirit:logicalPort>
          <spirit:physicalPort>
            <spirit:name>s_axi_awaddr</spirit:name>
          </spirit:physicalPort>
        </spirit:portMap>
      </spirit:portMaps>
    </spirit:busInterface>
  </spirit:busInterfaces>

```

General-Purpose (GP) Ports

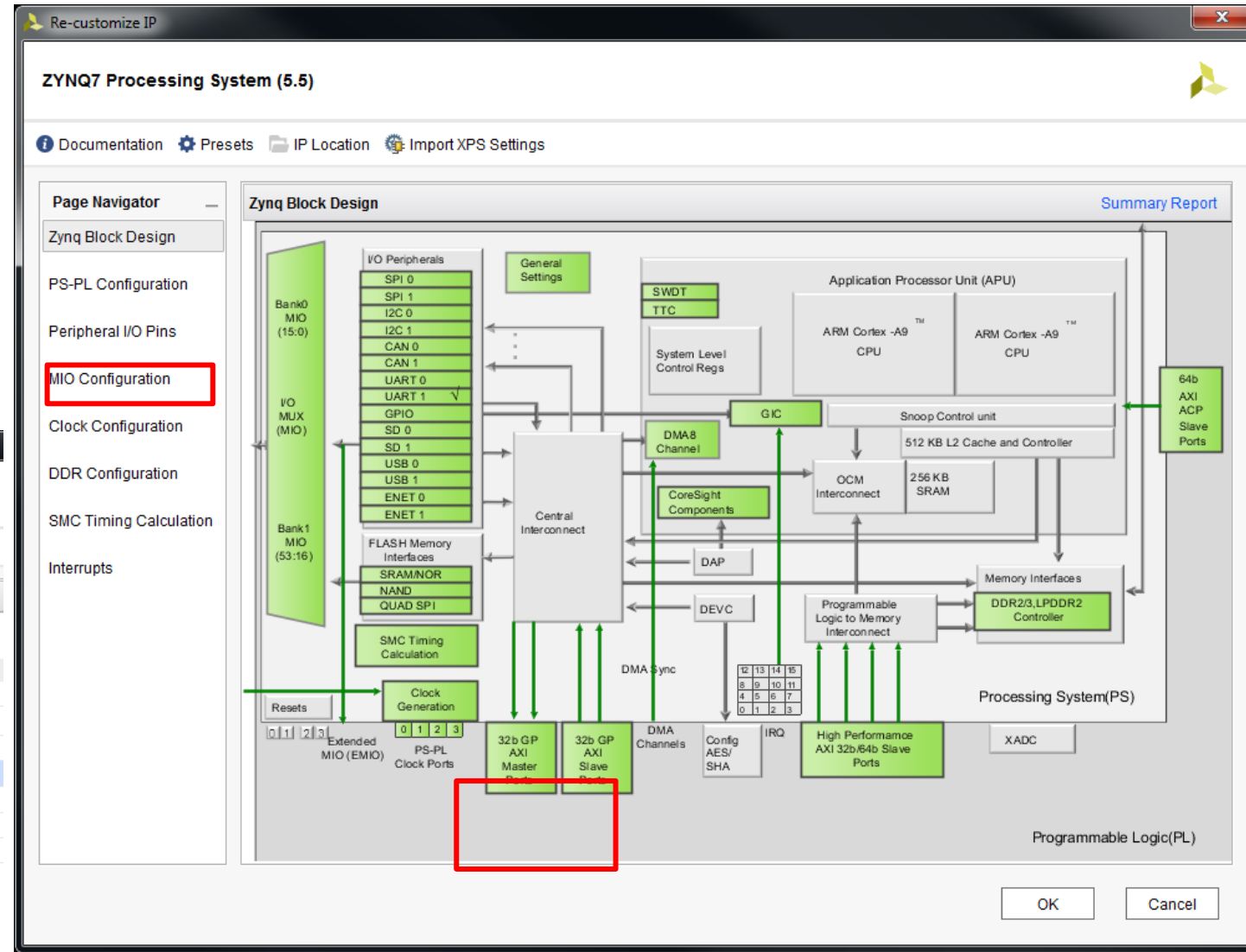
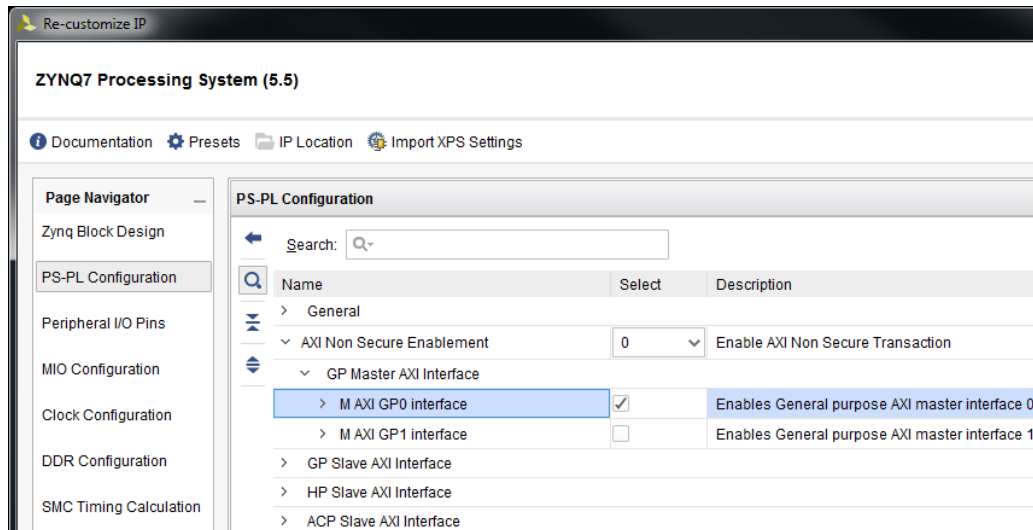
- By default, GP Slave and Master ports are disabled



- Enable GP Master and/or Slave ports depending on whether a slave or a master peripheral is going to be added in PL
- axi_interconnect block is required to connect IP to a port with different protocols
 - Automatically convert Protocols
 - Can be automatically added when using Block Automation in IPI

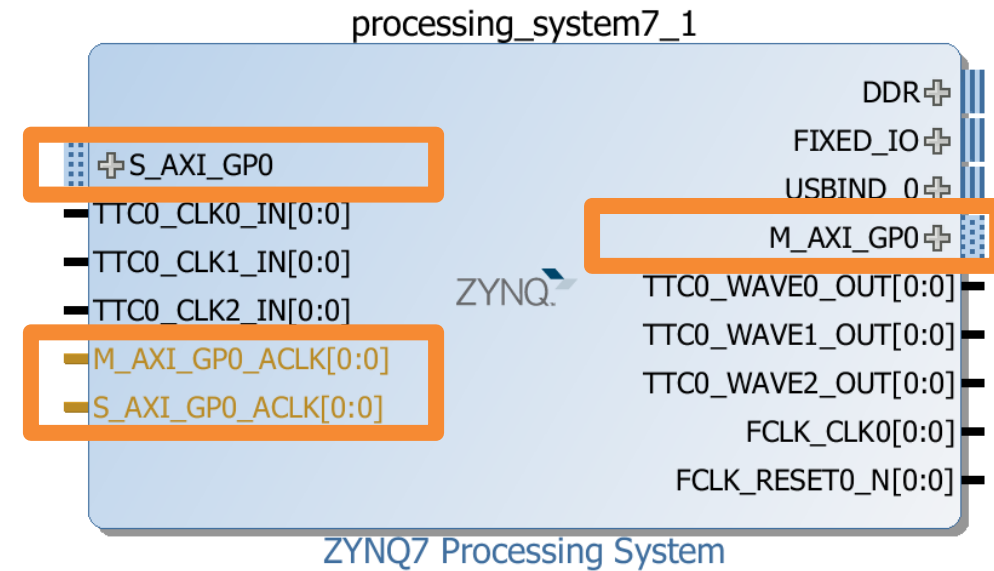
Configuring GP Ports

- Click on the menu or green GP Blocks to configure



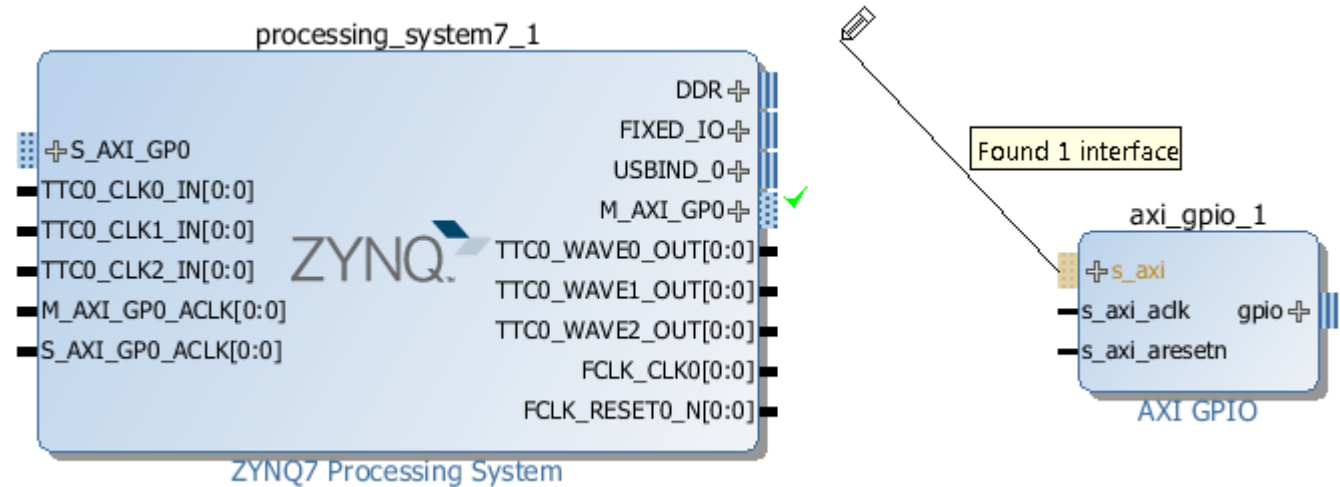
Add IP in the PL

- Configure GP ports from PS Customization GUI
- PS-PL Configuration
 - E.g. Enable M_AXI_GP0/I or S_AXI_GP0/I
- Ports will then be available on Zynq Block Diagram
- Connect the added IP to the appropriate port
- Assign address to the added IP, if unmapped
- Configure the IP, if necessary
- Make external connections, if needed
 - Add external ports/interfaces if the added IP is interacting with external devices



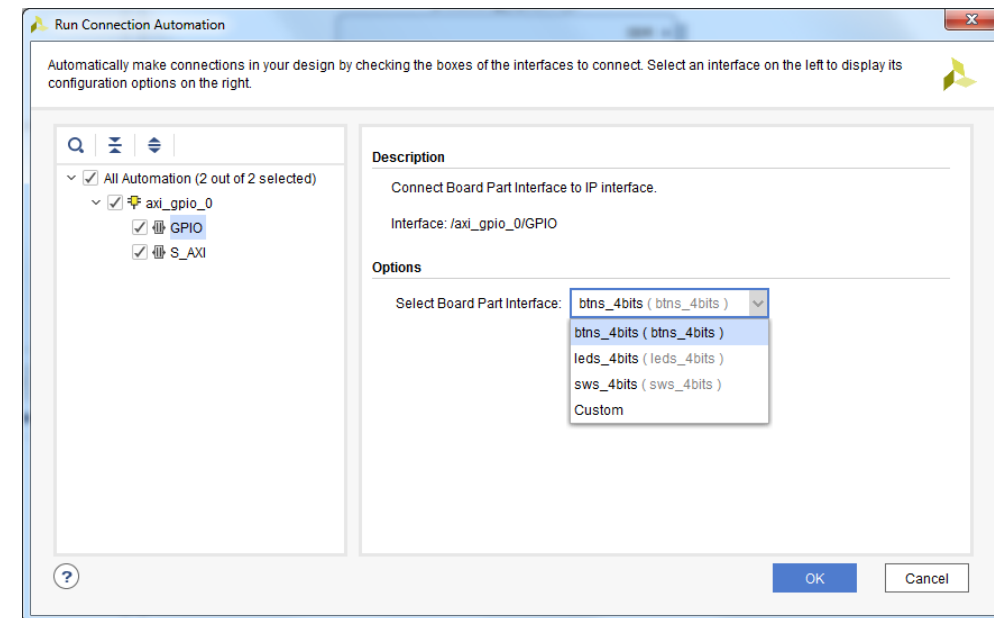
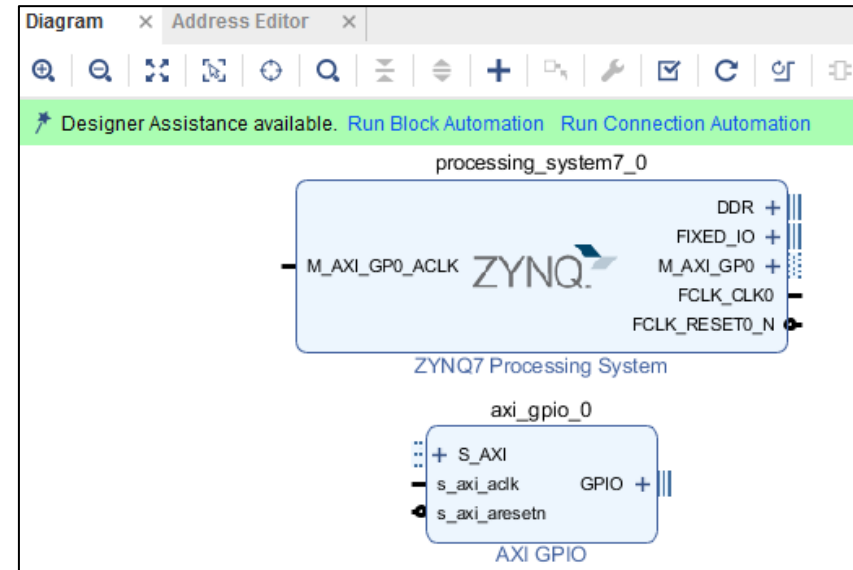
Connecting IP

- Add IP from the IP Catalog
- Click and drag to find connections
- Valid connections Highlighted
- Designer Assistance, Connection automation
 - If Board Support available, IP can be connected to external pins
- Or manually create and connect (external) ports



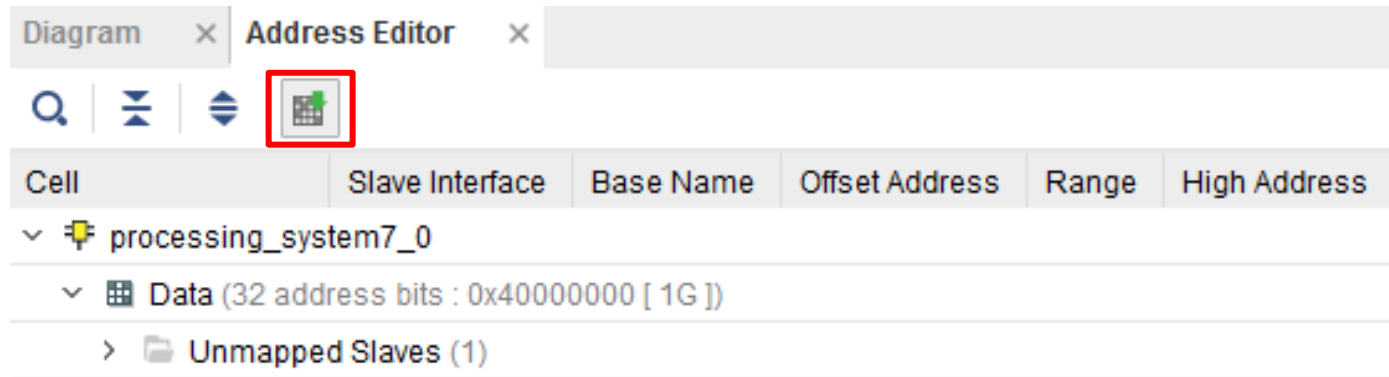
Designer assistance; Block Automation, Connection Automation

- Block, Connection
- Can automatically connect IP blocks
- Automatically insert required blocks
- E.g. Add BRAM; Automation will insert and connect BRAM controller and Reset logic
- If Board Support is available, IP can automatically be connected to top level ports



Assign Addresses

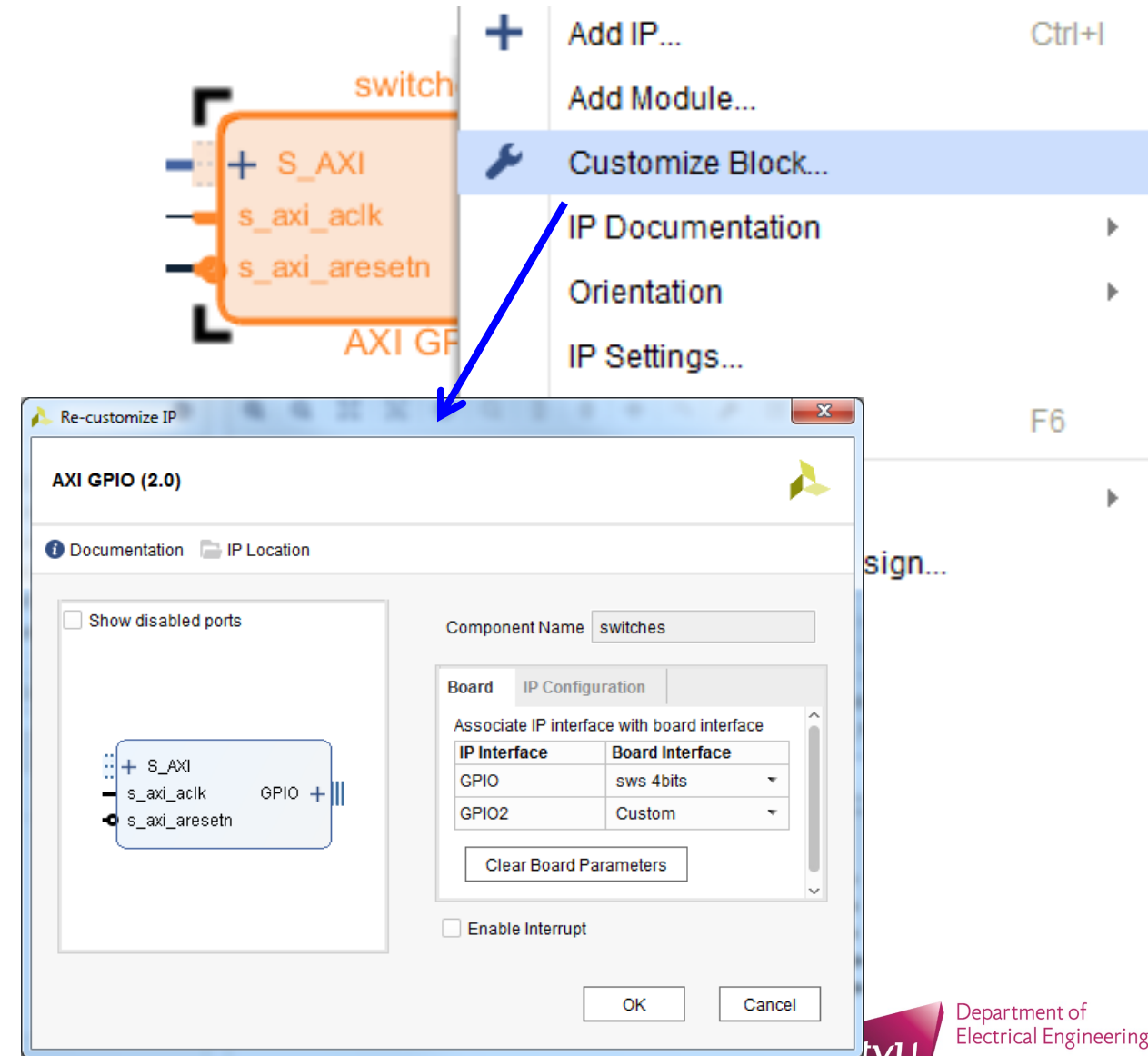
- Peripherals in the Zynq™ AP SoC PS have fixed addresses and do not appear in the address map when an IP is added to the system
- For PS peripherals Click on the Auto Assign Addresses button



- The address will be generated and show the generated addresses of the added IP
- The fixed addresses of the configured peripherals of the PS

Parameterize IP Instances

- Double-click or right click the instance and select Customize Block to open the configurable parameters dialog box (refer to the datasheet if needed)
- Default values are shown
 - Customize the parameters that you want



Extending the IP Catalog

- IP Packager
 - Packages into IP Integrator Format
- Specify repository (local/global)
- IP can then be used in IP Integrator
- More on IP Packager later

The screenshot shows the 'Package IP - axi_lite_slave' window in the IP Packager tool. The left sidebar lists the steps in the IP packaging process, with 'IP Identification' currently selected and highlighted in orange. The main area displays the 'IP Identification' tab, which includes a status bar at the top indicating '0 errors', '0 warnings', and '0 info messages'. Below this, a message states: 'Fill in and modify the information fields below that will be used to identify your IP. Set the categories in which your IP will appear in the IP Catalog by modifying the Taxonomy.' The 'Identification' section contains several input fields: 'Vendor' (xilinx.com), 'Library' (user), 'Name' (axi_lite_slave), 'Version' (1.0), 'Display Name' (axi_lite_slave_v1_0), 'Description' (axi_lite_slave_v1_0), 'Vendor Display Name' (empty), 'Company Url' (empty), 'Categories' (/Basic_Elements), 'Root Directory' (c:/xup/embedded/labs/led_ip), and 'Xml File Name' (c:/xup/embedded/labs/led_ip/component.xml). At the bottom, there is a checkbox for 'Show advanced information' which is currently unchecked.

Bitstream Generation

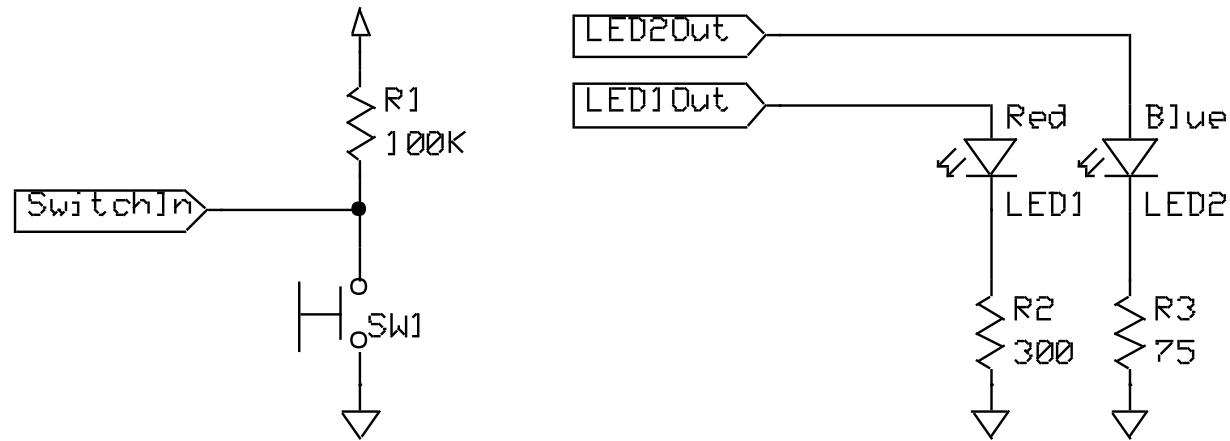
- After defining the system hardware, the next step is to create hardware netlists if the system hardware has logic in PL
- A HDL wrapper for the block diagram must be generated
 - Additional logic can be added to the HDL, or the Processor system can be used as a sub block in a HDL design
- The design and block diagram must be open before synthesise and implementation can be carried out
- If the system contains hardware in the PL, the bitstream must be generated
- The PL (FPGA) must be programmed before application can be downloaded and executed

Key points – SoC IP Integration

- PS functionality can be extended by instantiating peripherals in PL
- Adding IP in PL involves
 - Enabling interface(s) in PS
 - Selecting IP from the IP Catalog and configuring IP for desired functionality
 - Connecting the (PL) IP to the PS using IP Integrator
 - Assigning address
 - Connecting IP ports to ports of other peripherals and/or to external pins
- HDL Wrapper is needed for IP Integrator Block
- Bitstream must be generated when PL has any IP
- The FPGA must be programmed with the generated hardware bitstream before an application can be run

Digital Interfacing

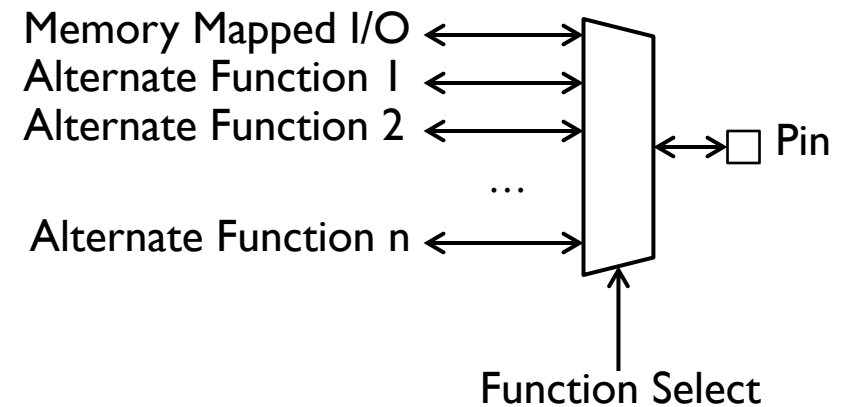
I/O Basic Concepts



- Goal: light either LED1 or LED2 based on switch SW1 position
- GPIO = General-purpose input and output (digital)
 - Input: program can determine if input signal is a 1 or a 0
 - Output: program can set output to 1 or 0
- Can use this to interface with external devices
 - Input: switch
 - Output: LEDs

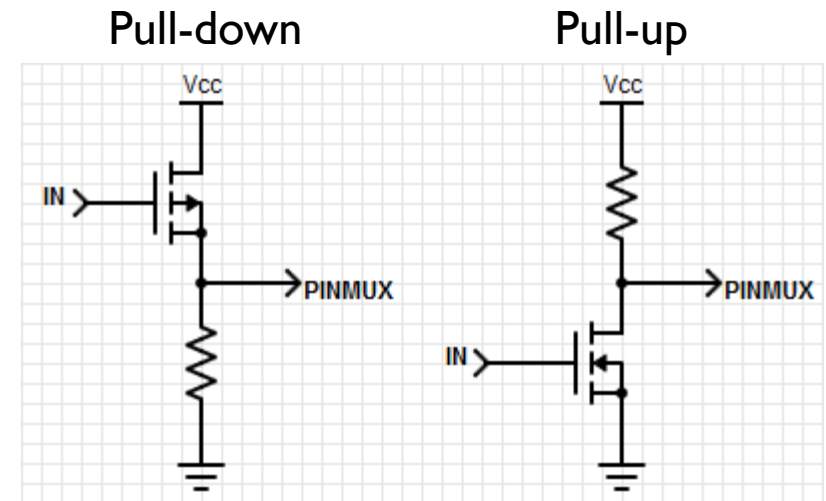
GPIO Alternative Functions

- Pins may have different features
- To enable an alternative function, set up the appropriate register
- May also have analogue paths for ADC / DAC etc.
- Advantages:
 - Saves space on the package
 - Improves flexibility



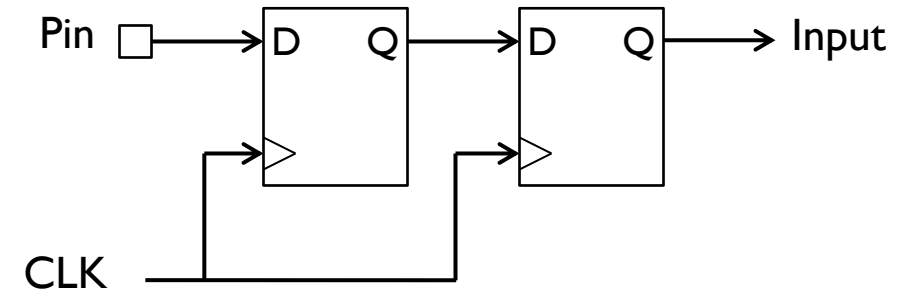
Pull-Up & Pull-Down Resistors

- Ensure a known value on the output if a pin is left floating
- In our example, we want the switch SW1 to pull the pin to ground, so we enable the pull-up
- The pin value is:
 - High when SW1 is not pressed
 - Low when SW1 is pressed



Input Synchronisation

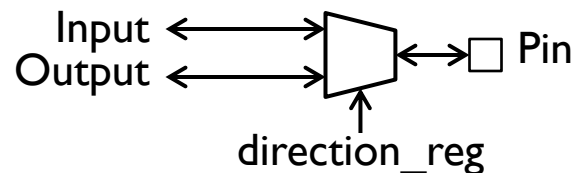
- External signals are asynchronous to internal clock
- If an external signal changes at the same time as the clock, a flip-flop can enter a metastable state indefinitely
- Solution – synchronise the input signals with the clock
- This is done for us by hardware, no need to worry!



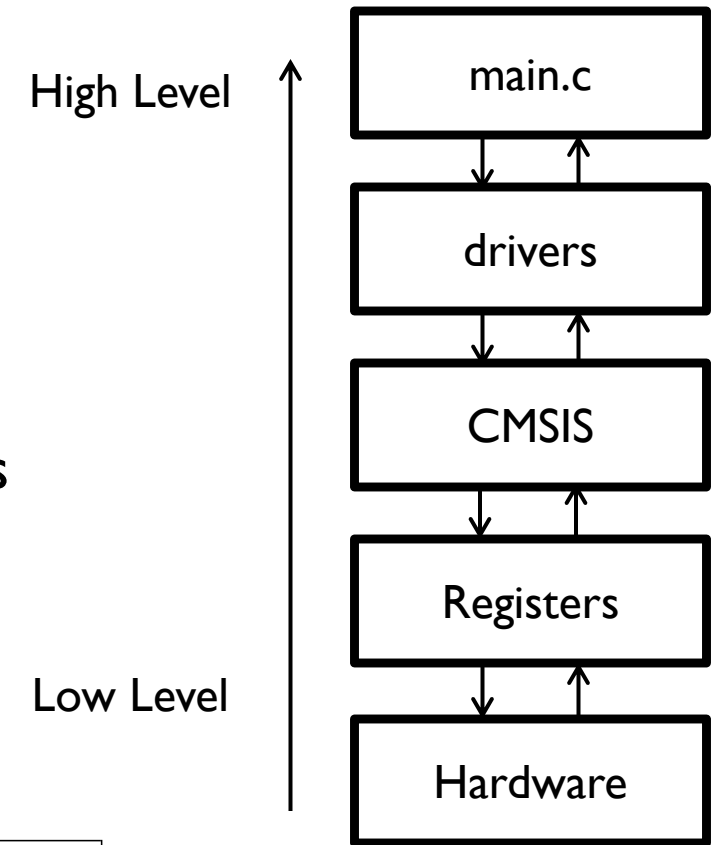
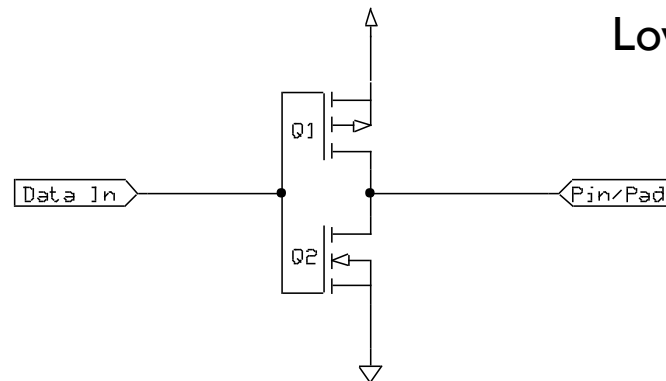
Code Structure

- Main code talks to the drivers, producing easy to read and understand code
 - `gpio_set_mode(P2_5, Output)`
- Drivers utilise CMSIS library and group relevant actions
 - `port_struct->direction_reg = output`
- CMSIS transforms memory mapped registers into C structs
 - `#define PORT0 ((struct PORT*)0x2000030)`

- Registers directly control hardware



- Hardware drives IO pins physically



Drivers Layer: How It Works

```
void gpio_set(Pin pin, int value)
```

- 1) `mask = 1 << pin index`
- 2) `tmp = port_struct->data_reg & ~mask`
- 3) `tmp |= value << pin index`
- 4) `port_struct->data_reg = tmp`

- e.g. `gpio_set(P2_5, 1)` with `PORT_DATA_REGISTER = 0b01010101`
 1. Create a mask for the bit we want to set (`0b00100000`)
 2. Invert the mask (`0b11011111`) to select all the other bits in the port data register, and save the status of the other bits (`tmp = 0b01010101`)
 3. Move the new value of the bit into position, and or it with the new register value (`tmp = 0b01110101`)
 4. Write the new data register value out to the port (`PORT_DATA_REGISTER = 0b01110101`)

Drivers Layer: How It Works

```
int gpio_get(Pin pin)
```

- 1) `mask = 1 << pin index`
- 2) `tmp = port_struct->data_reg & mask`
- 3) `tmp >>= pin index`
- 4) `return tmp`

- e.g. `gpio_get(P2_5)` with `PORT_DATA_REGISTER = 0b01110101`
 1. Create a mask for the bit we want to get (`0b00100000`)
 2. Select the bit in the port data register based on the mask (`tmp = 0b00100000`)
 3. Bitshift the value to produce a one or zero (`tmp = 0b00000001`)
 4. Return the value of the pin back to the user

C Interface: GPIO Configuration

```
/*! This enum describes the directional setup of a GPIO pin. */
typedef enum {
    Reset,    //!< Resets the pin-mode to the default value.
    Input,    //!< Sets the pin as an input with no pull-up or pull-down.
    Output,   //!< Sets the pin as a low impedance output.
    PullUp,   //!< Enables the internal pull-up resistor and sets as input.
    PullDown  //!< Enables the internal pull-down resistor and sets as input.
} PinMode;

/*! \brief Configures the output mode of a GPIO pin.
 *   Used to set the GPIO as an input, output, and configure the
 *   possible pull-up or pull-down resistors.
 *   \param pin    Pin to set.
 *   \param mode    New output mode of the pin.
 */
void gpio_set_mode(Pin pin, PinMode mode);
```

C Interface: Reading and Writing

```
/*! \brief Sets a pin to the specified logic level.  
 * \param pin    Pin to set.  
 * \param value  New logic level of the pin (0 is low, otherwise high).  
 */
```

```
void gpio_set(Pin pin, int value);
```

```
/*! \brief Get the current logic level of a GPIO pin.  
 * \param pin    Pin to read.  
 * \return The logic level of the GPIO pin (0 if low, 1 if high).  
 */
```

```
int gpio_get(Pin pin);
```

Pseudocode for Program

```
Make LED1 and LED2 outputs
Make switch an input with a pull-up resistor
do forever {
    if switch is not pressed {
        Turn off LED1
        Turn on LED2
    } else {
        Turn off LED2
        Turn on LED1
    }
}
```

C Code

```
gpio_set_mode(P_LED1, Output); // Set LED pins to outputs
gpio_set_mode(P_LED2, Output);
gpio_set_mode(P_SW, Pullup); // Switch pin to resistive pull-up

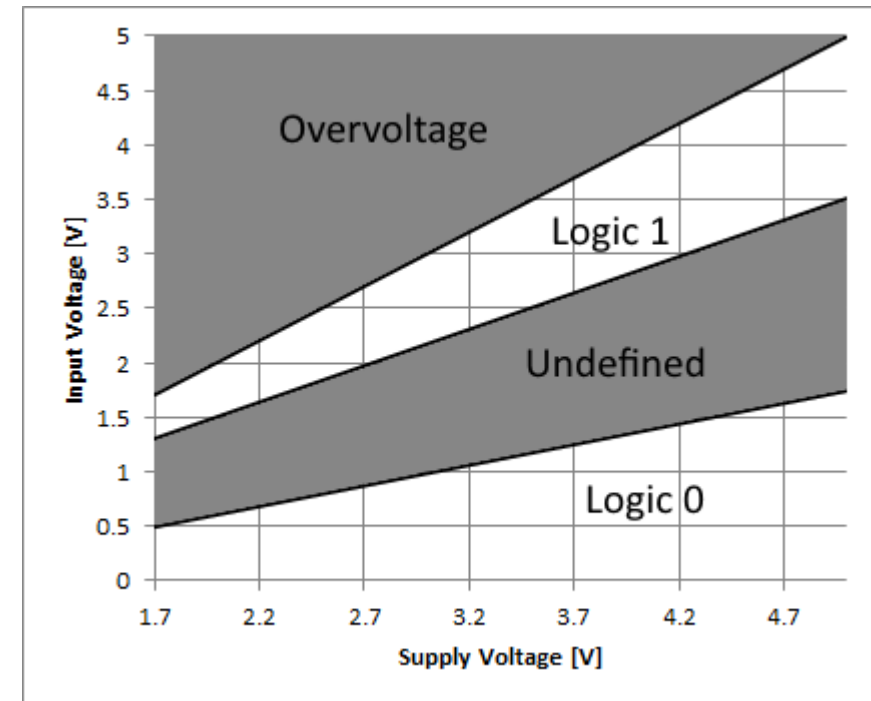
while (1) {
    if (gpio_get(P_SW)) {
        // Switch is not pressed (active low), turn LED1 off and LED2 on.
        gpio_set(P_LED1, 0);
        gpio_set(P_LED2, 1);
    } else {
        // Switch is pressed, turn LED2 off and LED1 on.
        gpio_set(P_LED2, 0);
        gpio_set(P_LED1, 1);
    }
}
```

Interfacing

Inputs and Outputs, Ones and Zeros, Voltages and Currents

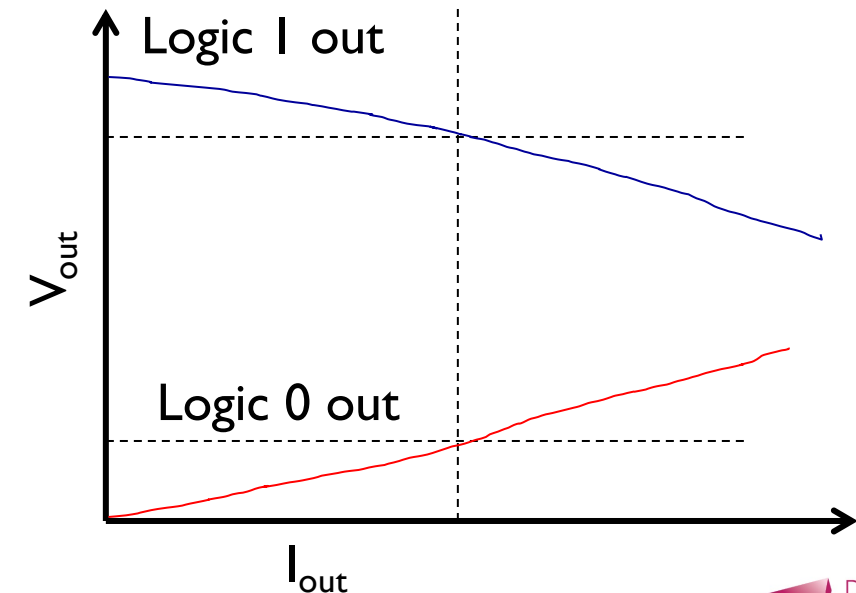
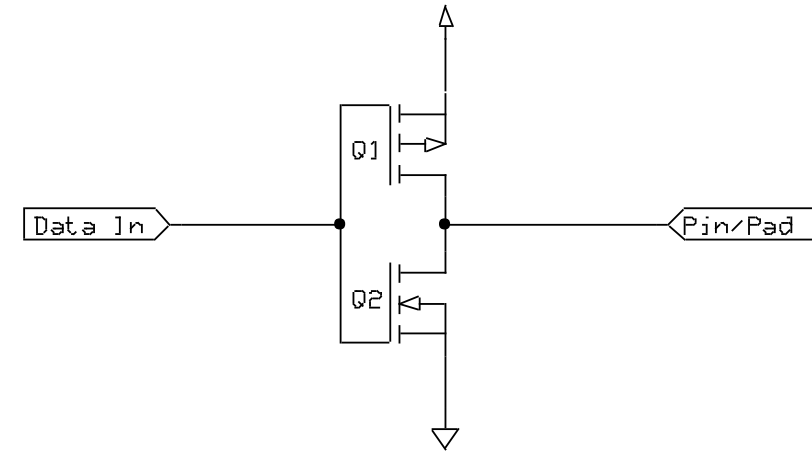
Inputs: What's a One? A Zero?

- Input signal's value is determined by voltage
- Input threshold voltages depend on supply voltage VDD
- Exceeding VDD or GND may damage chip



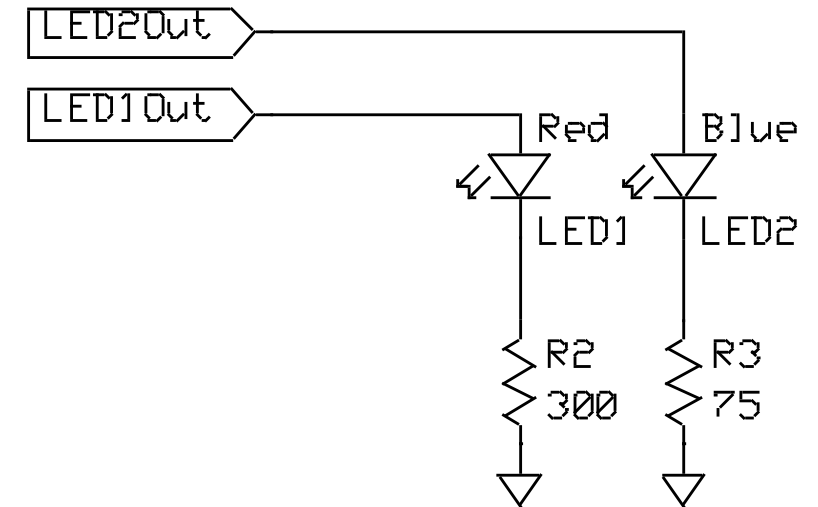
Outputs: What's a One? A Zero?

- Nominal output voltages
 - 1: $V_{DD} - 0.5\text{ V}$ to V_{DD}
 - 0: 0 to 0.5 V
- Note: Output voltage depends on current drawn by load on pin
 - Need to consider source-to-drain resistance in the transistor
 - Above values only specified when current $< 5\text{ mA}$ (18 mA for high-drive pads) and $V_{DD} > 2.7\text{ V}$



Output Example: Driving LEDs

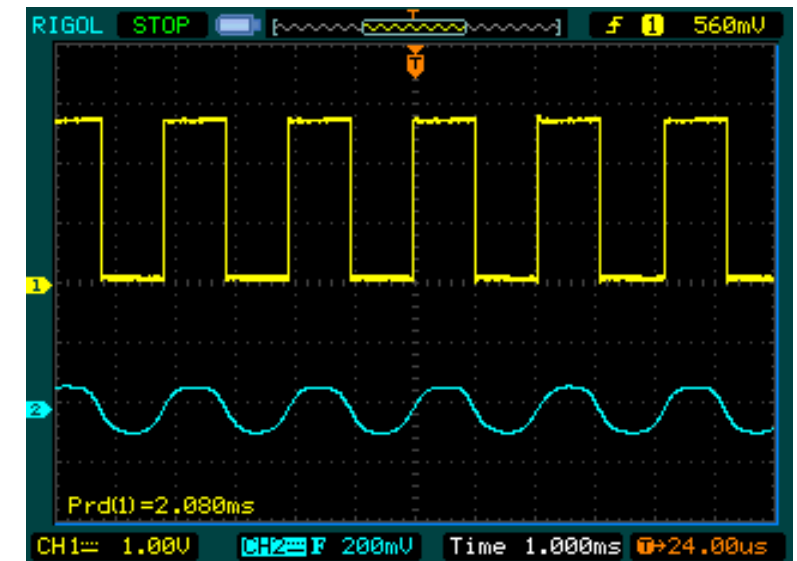
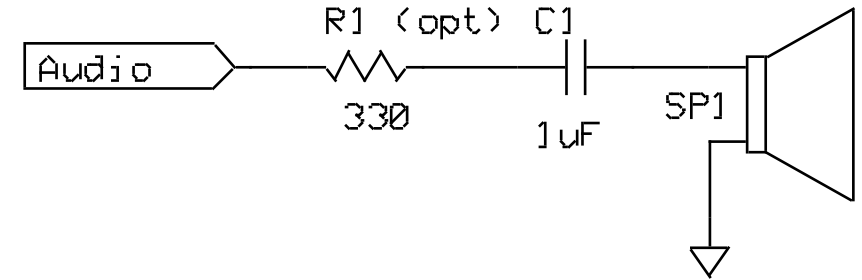
- Need to limit current to a value which is safe for both LED and MCU port driver
- Use current-limiting resistor
 - $R = (V_{DD} - V_{LED}) / I_{LED}$
- Set $I_{LED} = 4 \text{ mA}$
- V_{LED} depends on type of LED (mainly color)
 - Red: $\sim 1.8 \text{ V}$
 - Blue: $\sim 2.7 \text{ V}$
- Solve for R given $V_{DD} = \sim 3.0 \text{ V}$
 - Red: 300Ω
 - Blue: 75Ω
- Demonstration code in Basic Light Switching Example



Output Example: Driving a Speaker

- Create a square wave with a GPIO output
- Use capacitor to block DC value
- Use resistor to reduce volume if needed

```
void beep(void) {  
    unsigned int period = 20;  
    while (1) {  
        gpio_toggle(P_SPEAKER);  
        delay_ms(period/2);  
    }  
}
```



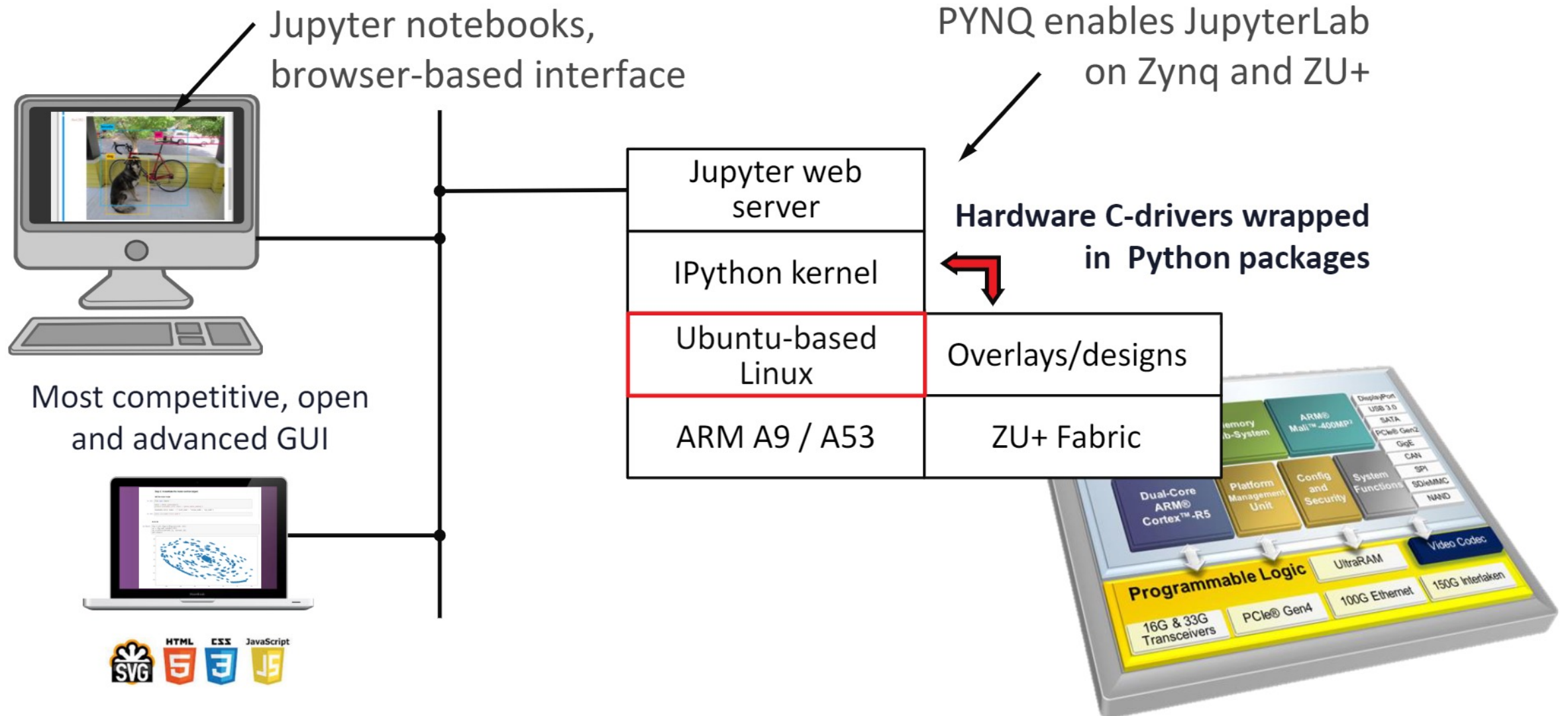
Analog Interfacing

Why It's Needed

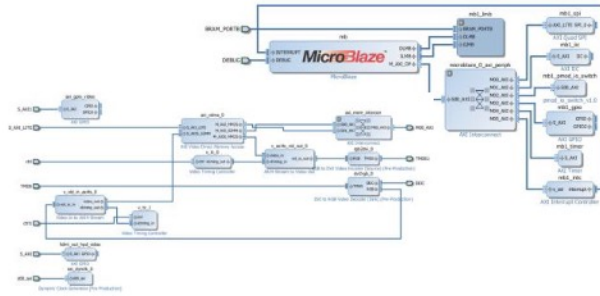
- Embedded systems often need to measure values of physical parameters
- These parameters are usually continuous (analog) and not in a digital form which computers (which operate on discrete data values) can process
- Temperature
 - Thermometer (do you have a fever?)
 - Thermostat for building, fridge, freezer
 - Car engine controller
 - Chemical reaction monitor
 - Safety (e.g. microprocessor processor thermal management)
- Light (or infrared or ultraviolet) intensity
 - Digital camera
 - IR remote control receiver
 - Tanning bed
 - UV monitor
- Rotary position
 - Wind gauge
 - Knobs
- Pressure
 - Blood pressure monitor
 - Altimeter
 - Car engine controller
 - Scuba dive computer
 - Tsunami detector
- Acceleration
 - Air bag controller
 - Vehicle stability
 - Video game remote
- Mechanical strain
- Other
 - Touch screen controller
 - EKG, EEG
 - Breathalyzer

PYNQ SoC Extensible I/O Systems

PYNQ: Python Productivity for Zynq



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) == 0){
        cmd=MAILBOX_CMD_ADDR;

        count = (cmd & 0x0000ff00) >> 8;
        if((count==0) || (count>253)) {
            // clear bit[0] to indicate
            // set rest to 1s to indicate
            MAILBOX_CMD_ADDR = 0xffffffff;
            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	7c7f	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 pyngq import Overlay
from pyngq.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)

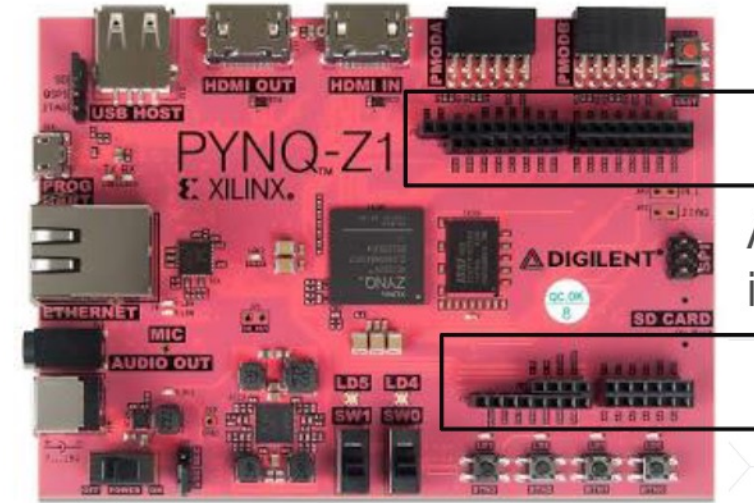
for j in range(20):
    value = 0.1 * j
    dac.write(value)
    sleep(0.5)
    # readings=adc.read(1,0,0)
    # x1=readings[0]
    print("Voltage read by DAC is: {:.4f} volts".format(adc.read(1,0,0)[0]))
```

6c78	3963	7367	3232	3500	6300	0b32
332f	3039	2f33	3000	6400	0931	323a
f	ffff					
5	6630					
1	6109					
9	8120					
9	0020					
9	0020					
2	0100					
3	0101					
2	c100					

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

Arduino Interface

- > Wide range of off-the-shelf Arduino shields
- > **Arduino interface specification**
 - >> 6 Analog Inputs
 - >> 14 Digital pins
 - UART, PWM, Timer, SPI, interrupts
 - >> Dedicated SPI, I2C
- > **On PYNQ-Z1/Z2 header connected to FPGA pins**
 - >> Interface is built in Overlay
 - >> Can breadboard to these pins

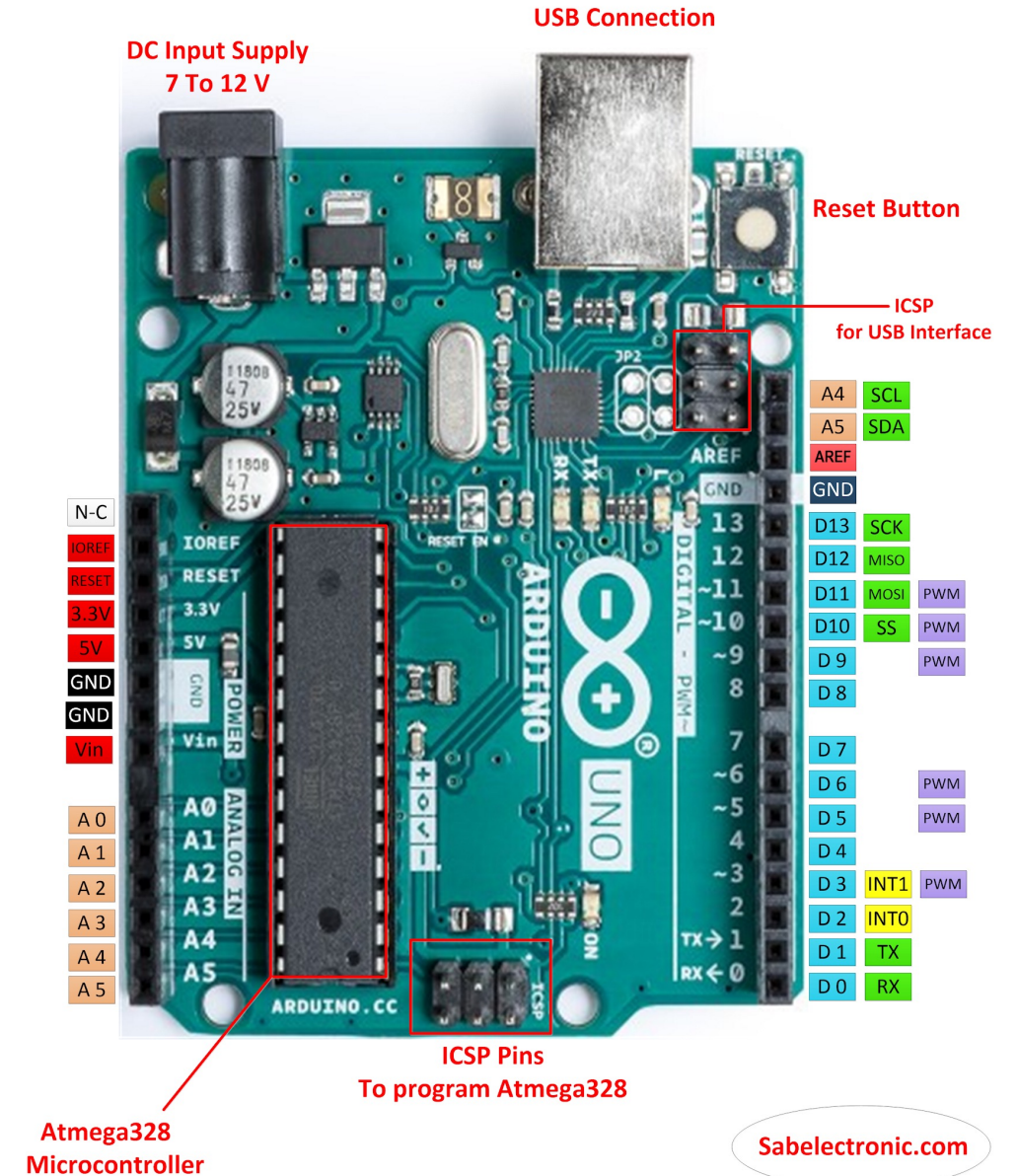
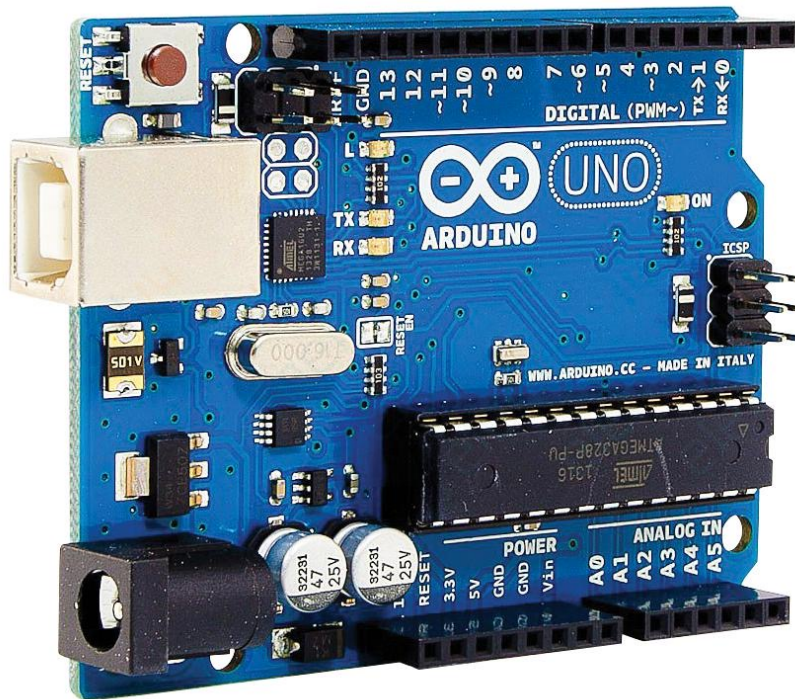


Arduino interface



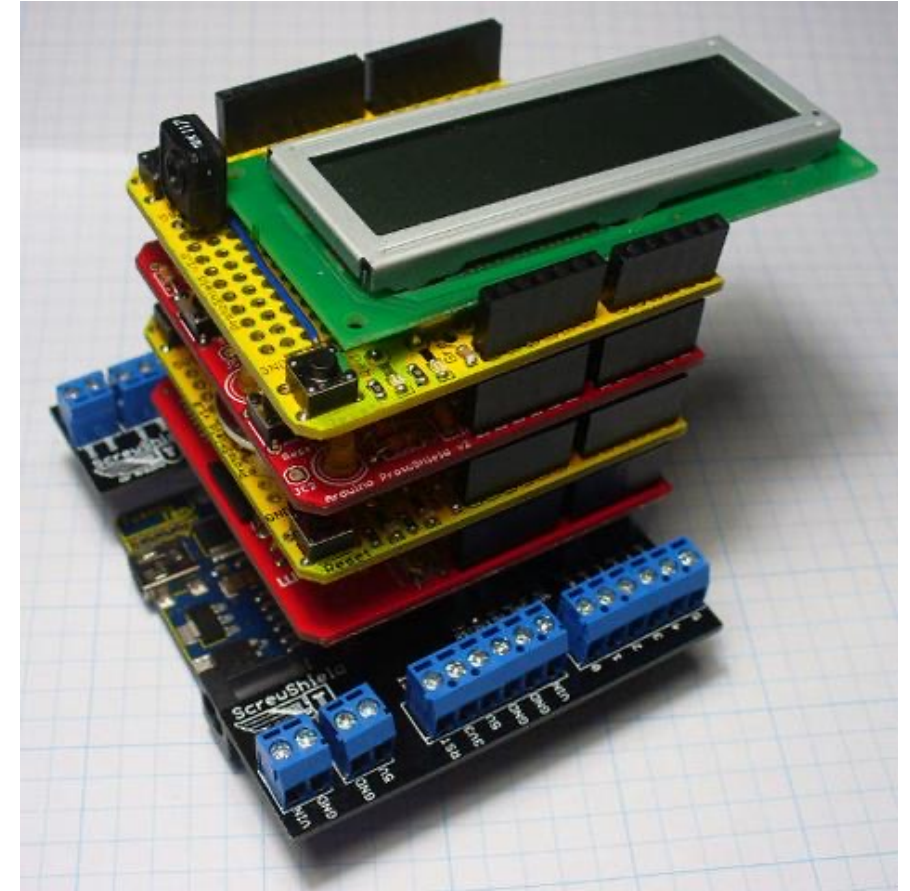
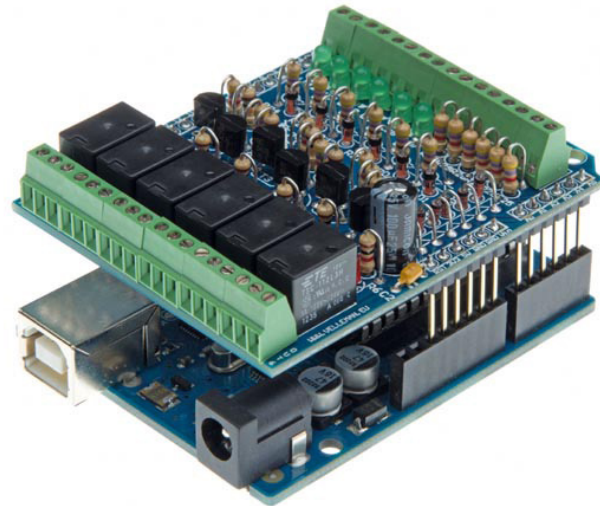
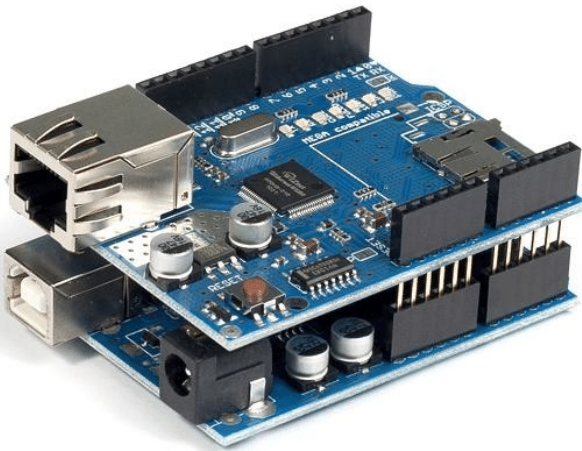
Arduino Shield System

- Arduino is a beginner friendly MCU
- Use it in other EE courses
- Become a standard in the STEM education



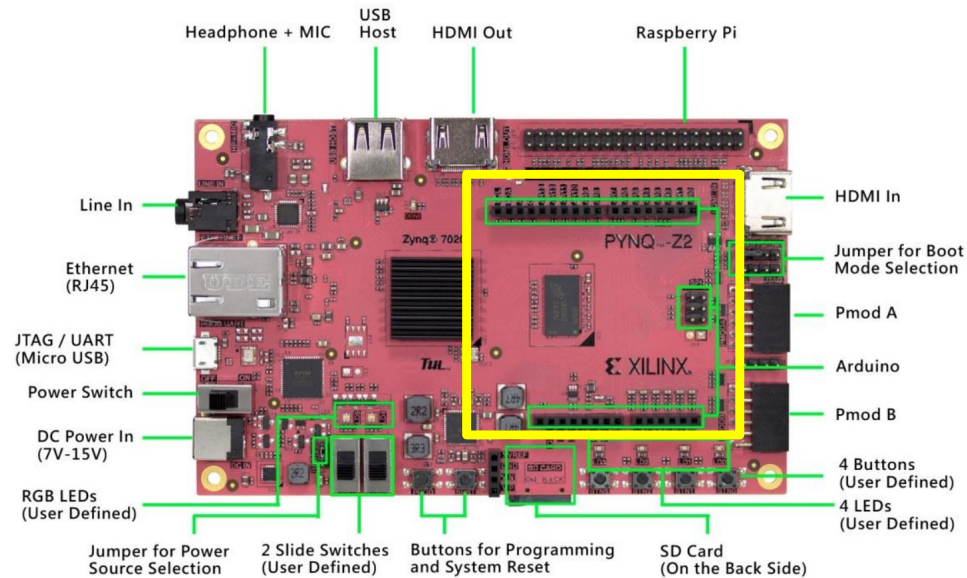
Arduino Shield System

- Standardized Shield with different function
- E.G. Sensor, internet, display, motor driver...

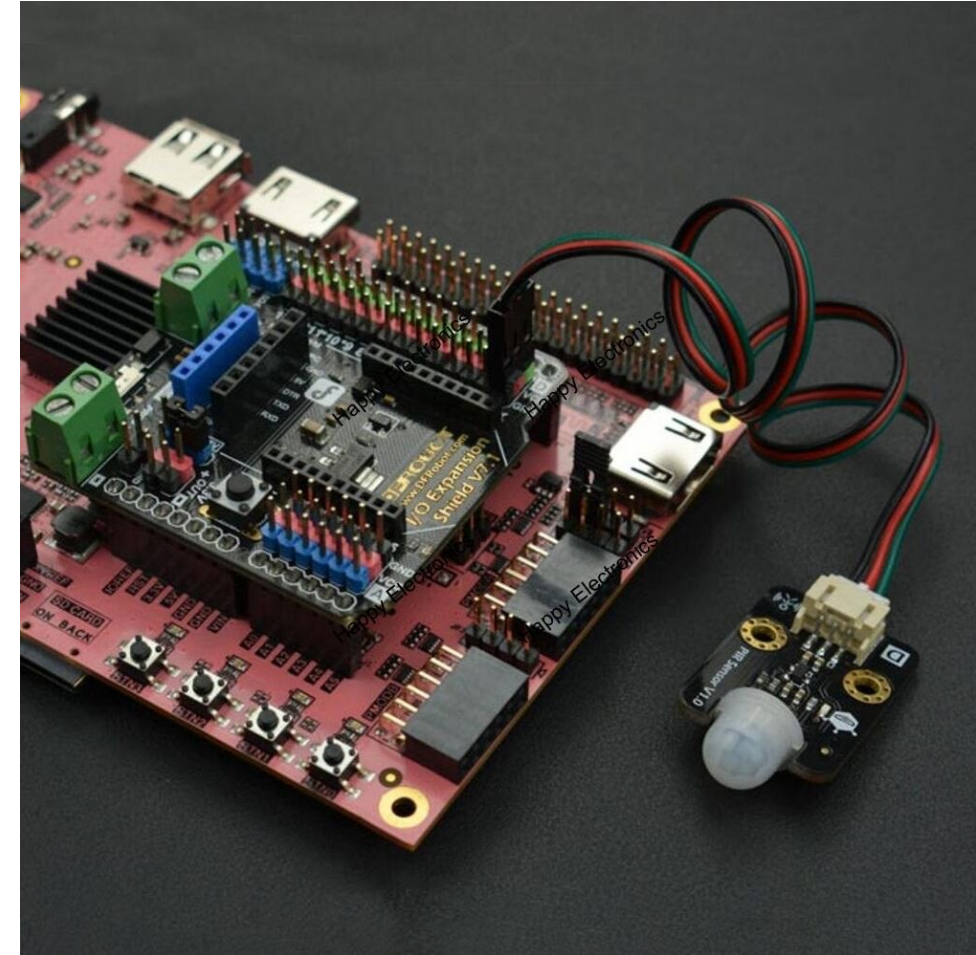


PYNQ Arduino Shield System

- Can be used to connect to Arduino compatible shields to PL
- Can also be used as general purpose pins

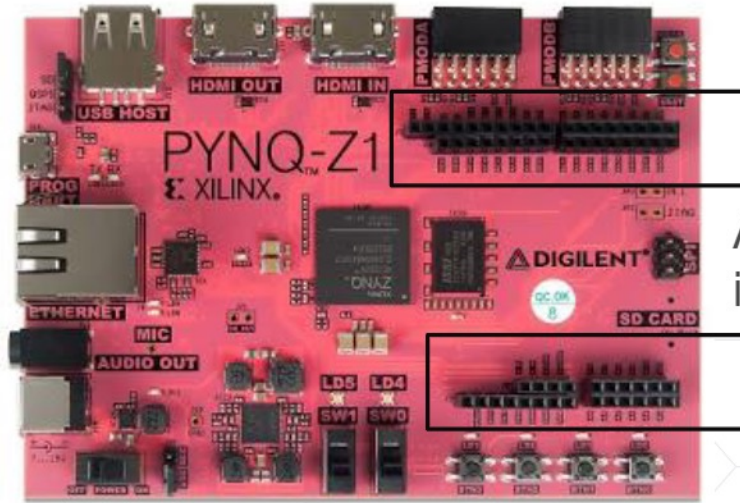


<http://www.pynq.io/>



Arduino Interface

- > Wide range of off-the-shelf Arduino shields
- > **Arduino interface specification**
 - >> 6 Analog Inputs
 - >> 14 Digital pins
 - UART, PWM, Timer, SPI, interrupts
 - >> Dedicated SPI, I2C
- > **On PYNQ-Z1/Z2 header connected to FPGA pins**
 - >> Interface is built in Overlay
 - >> Can breadboard to these pins



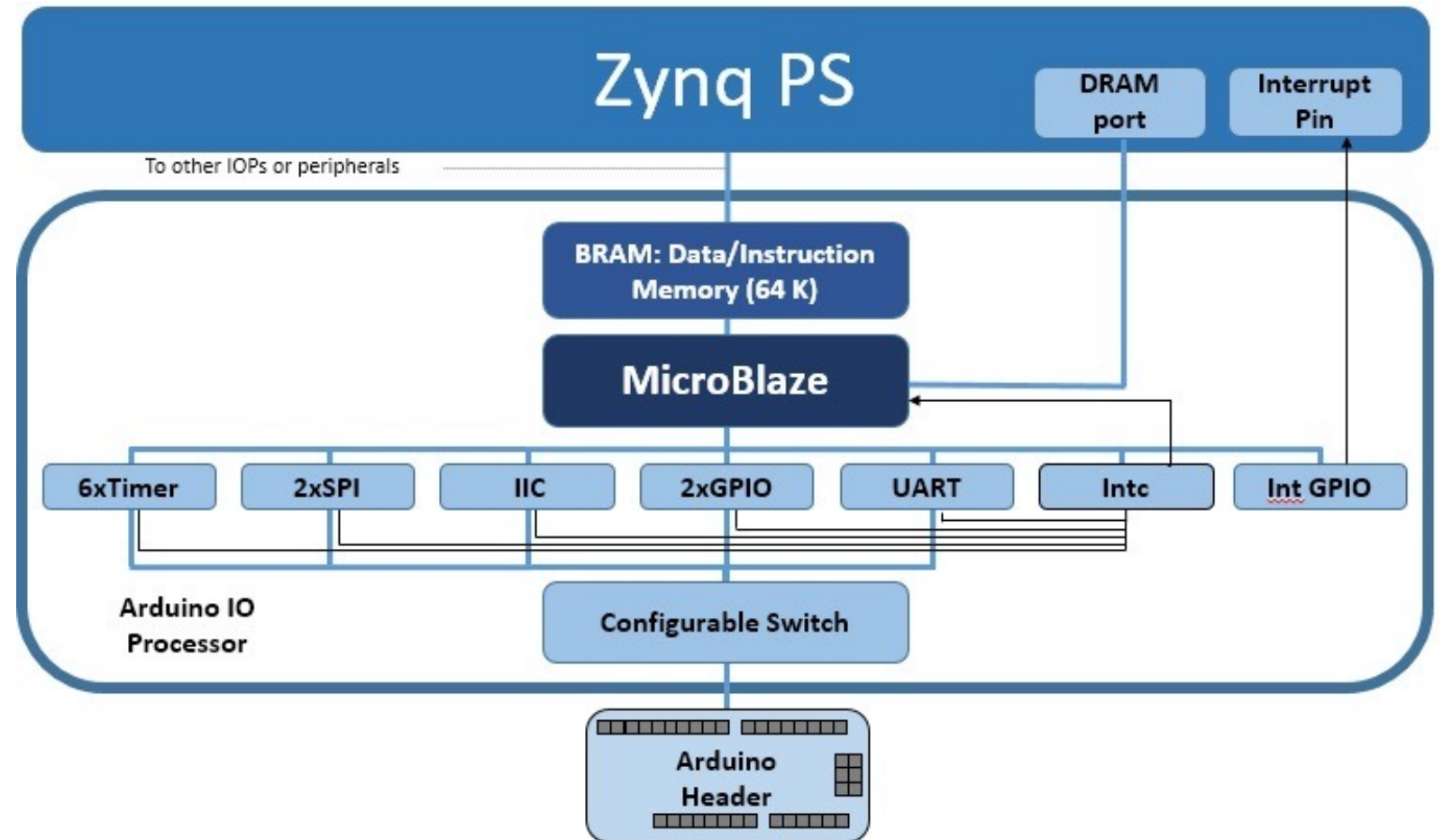
Arduino interface



RANDOMNERDTUTORIALS.COM

Arduino PYNQ MicroBlaze

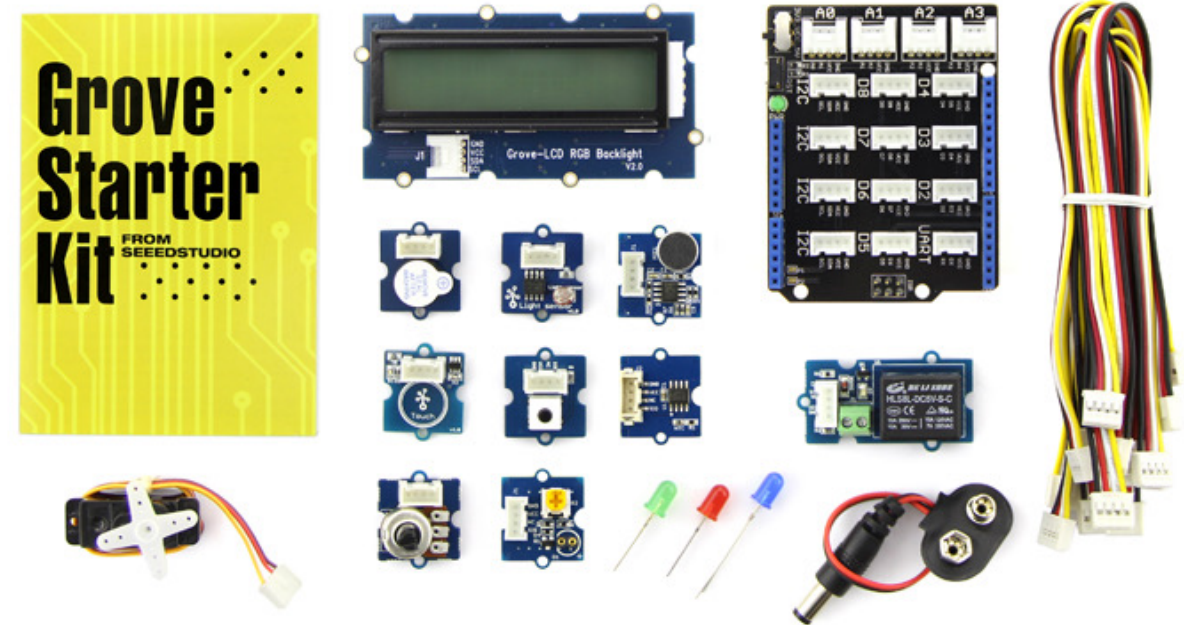
Peripheral	Pins
UART	D0, D1
I2C	SCL, SDA
SPI*	D10 - D13
PWM	D3, D5, D6, D9, D10, D11
Timer	D3 - D6 and D8 - D11



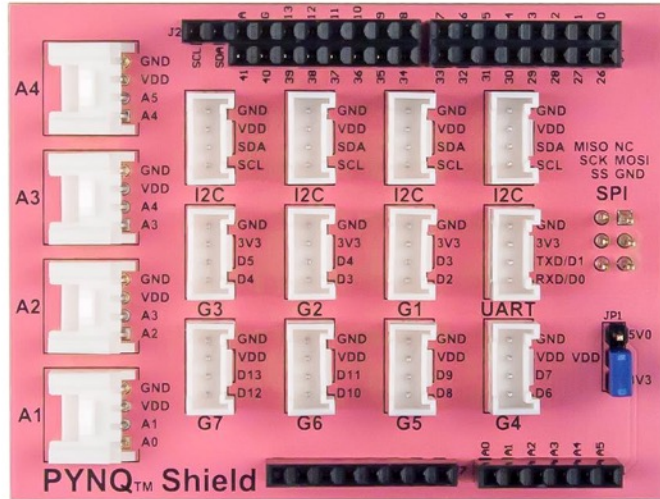
Grove Kit

- Open-source, modulated, and ready-to-use toolset
- Building block approach to assemble electronics
- Simplifies the learning process significantly
- Consists of a Base Shield and various modules with standardized connectors.

<https://www.seeedstudio.com/category/Grove-c-1003.html>

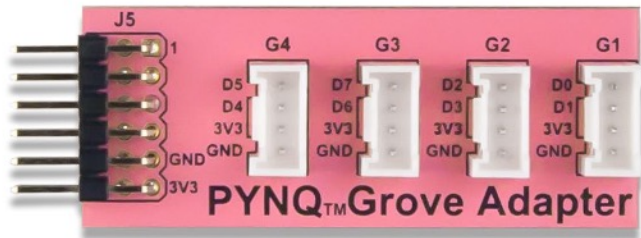


Low-cost PYNQ Shield & Pmod Grove Adapter



PYNQ Shield:

- 4 x Analog ports
- 4 x I2C ports
- 3 x 3.3V GPIO ports
- 1 x UART
- 4 x 3.3/5V switchable GPIO ports
- 1 x SPI header
- 1 x 16-pin GPIO header (inner header)



PYNQ Grove Adapter :

- 4 independent sockets for Grove modules
- Pmod compatible
- Solderless breadboard compatible
- Open-source design

Grove: Wide range low-cost sensors, actuators, ...

Environmental Monitoring

Have you ever wanted to get your daily weather report based on data from your garden instead of obtaining a more generic report from your TV or mobile phone? Sensors



Grove - Digital Light Sensor

Grove - Light Sensor

Grove - Temperature and Humidity Sensor

Grove - Barometer Sensor

Grove - Dust Sensor

Motion Sensing

Sensors in this category enable your microcontroller to detect motion, location and direction. You can make the movement of your microcontroller understandable in three dimensional spaces



Grove - 3-Axis Digital Compass

Grove - 3-Axis Digital Accelerometer ($\pm 1.5g$)

Grove - 3-Axis Digital Gyro

Grove - Collision Sensor

Grove - 3-Axis Analog Accelerometer

Wireless Communication

Communicating without wires is a cool feature that can spice up your project. Modules in this category arm your microcontroller with wireless communication ability such as RF, Bluetooth, etc.



Grove - 315MHz Simple RF Link Kit

Grove - Serial RF Pro

Grove - GPS

Grove - 125KHz RFID Reader

Grove - Serial Bluetooth

User Interface

Modules in this, our largest, category, let you interface with your microcontroller via input modules, such as touch pads, joysticks or your voice. Or you can choose output modules.



Grove - Solid State Relay

Grove - OLED Display 128*64

Grove - Serial LCD

Grove - LED Socket Kit

Grove - Button

Physical Monitoring

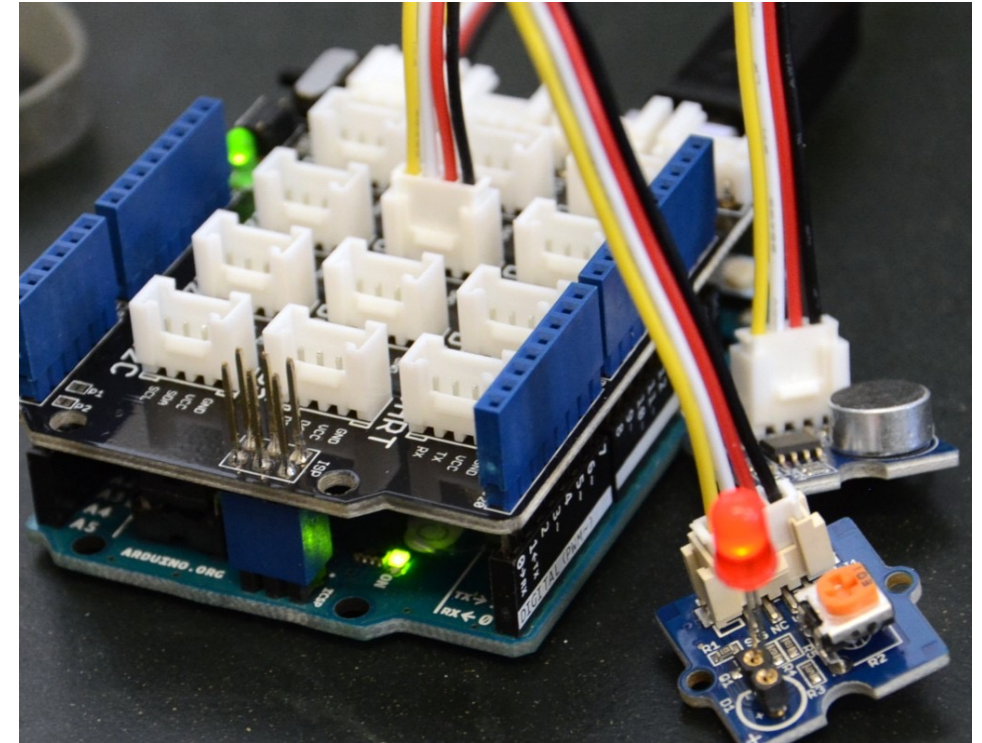
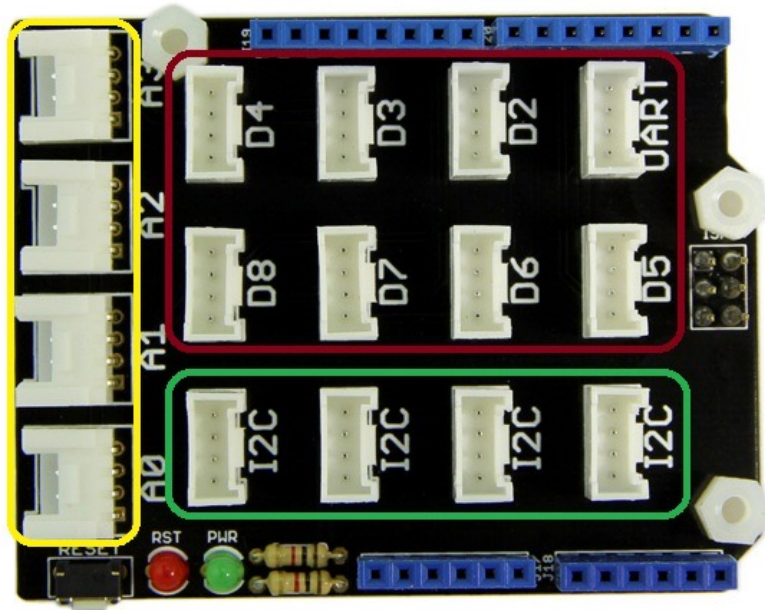
Scientists understand the world around us in physical dimensions. Modules in this category are designed to help you analyze the physical world. Measure your heart rate, ...



www.seeedstudio.com/wiki/Grove_System

Standardize and Modular

- Plug and Play with different port
- Digital / Analog / I2C



Summary

- Using IP in SoC Designs
 - Identify the IP supplied as part of Xilinx Vivado Platform
 - Describe how to add hardware to an existing Vivado project
 - Explain how IP is added to extend processing system functionality
- GPIO
 - How do we make a program light up LEDs in response to a switch?
 - Basic Concepts
 - Port Circuitry
 - Alternate Functions
 - Peripheral Access In C
- PYNQ Arduino System and Grove I/O Kit