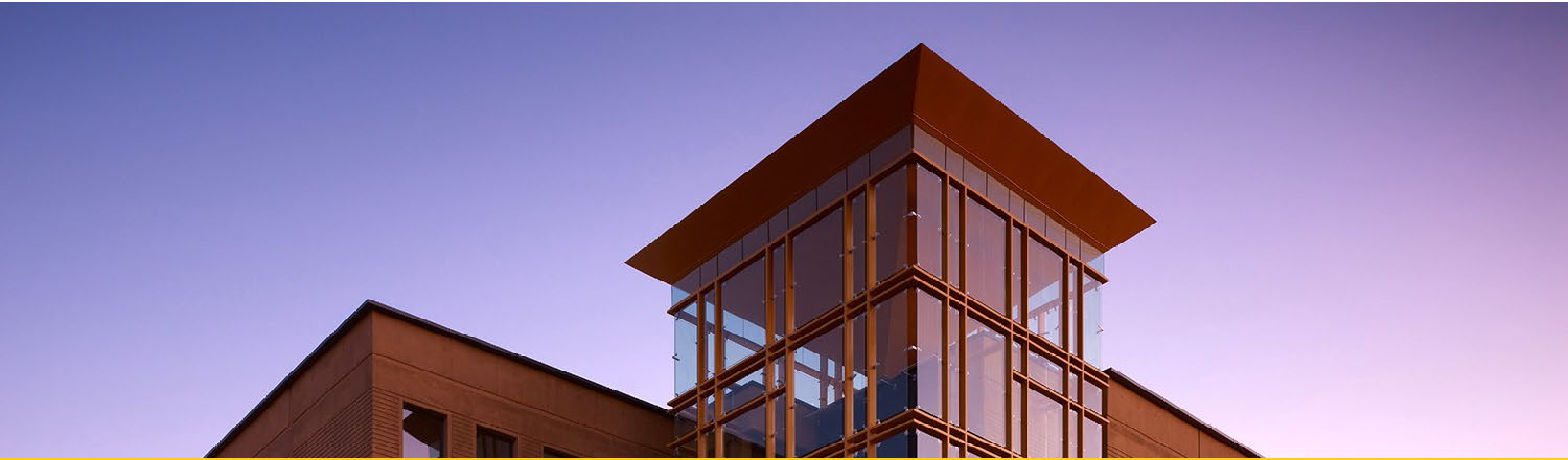
Lecture 4:
Reconfigurable Accelerators

Sitao Huang

sitaoh@uci.edu

January 27, 2022



Logistics

- **Homework 1** released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7

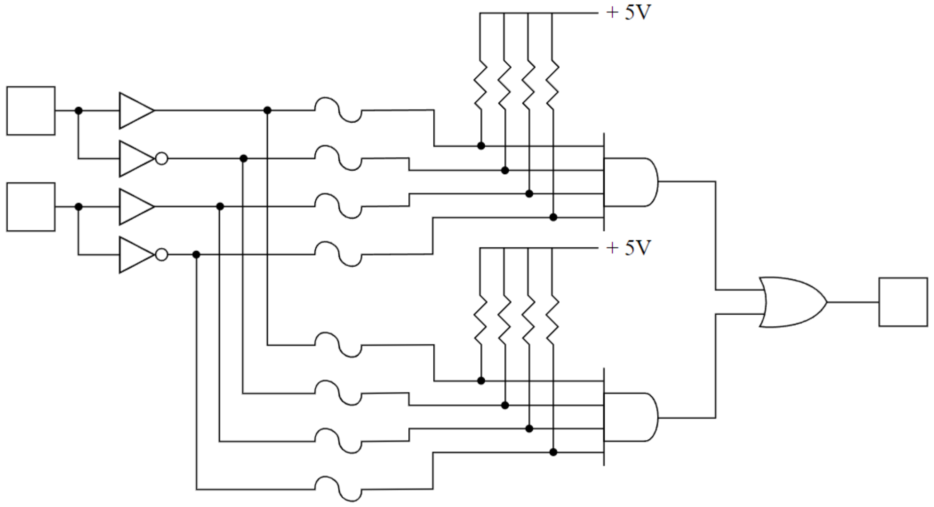
- **Midterm**: **February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)

- Project proposal due: February 14
  - Options: (a) literature review paper or (b) compiler + accelerator project

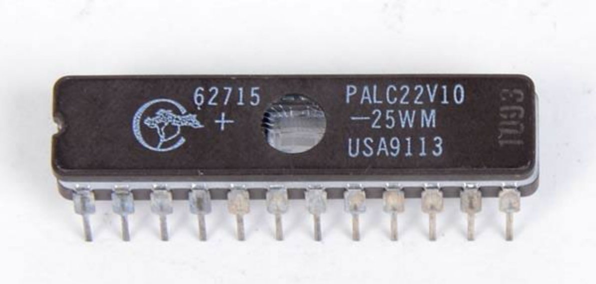
- In-person ***(hybrid)*** instruction starting next week (week 5)!
- First in-person class: ***February 1***

Programmable Logic Devices (PLD)

- Evolved from PALs (Programmable Array Logic), GALs (Generic Array Logic) and CPLD (Complex Programmable Logic Device)
- First generation devices (PALs) are one-time programmable; used to create combinational logical function
- Engineer writes logic expressions and defines I/Os on computer
- Software simplifies expression to SOP form and generates programming file
- Programming cable is used to set fuses / anti-fuses
- No memory / feedback elements limits applications
- Non-volatile (maintains design on power cycle)
- Using specialized machines, PAL devices were “field-programmable”
  - "One-time programmable" (OTP) devices
  - UV erasable devices
  - Flash erasable devices



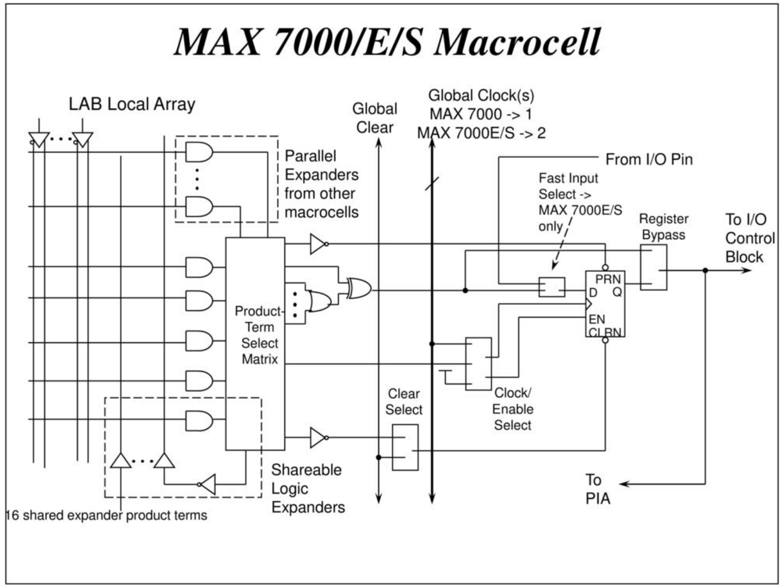
Simplified programmable logic device

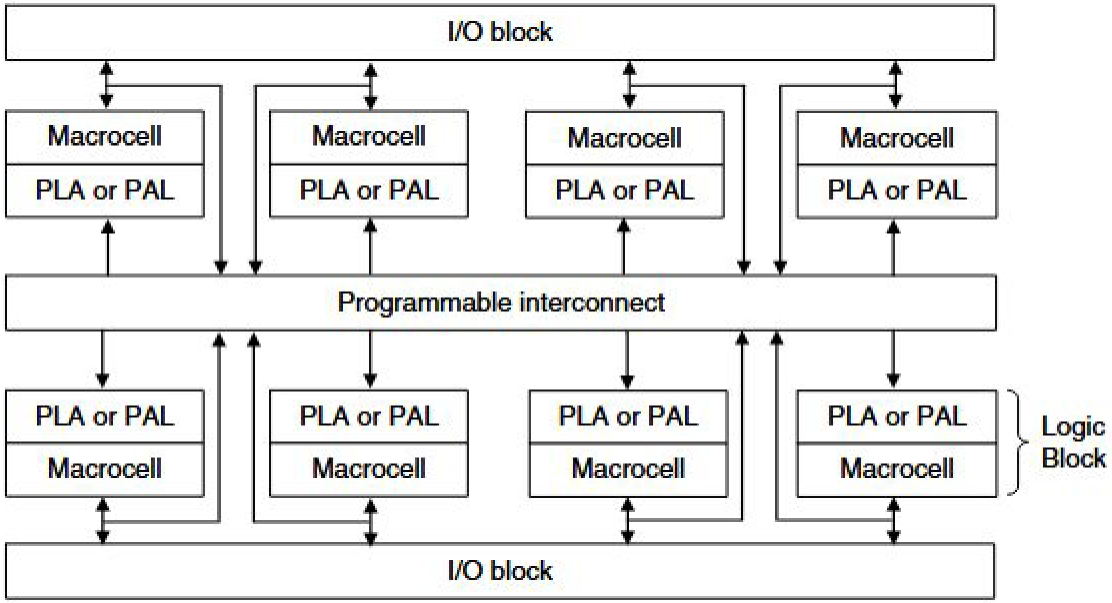


© Artisan Technology Group

Complex Programmable Logic Devices (CPLD)

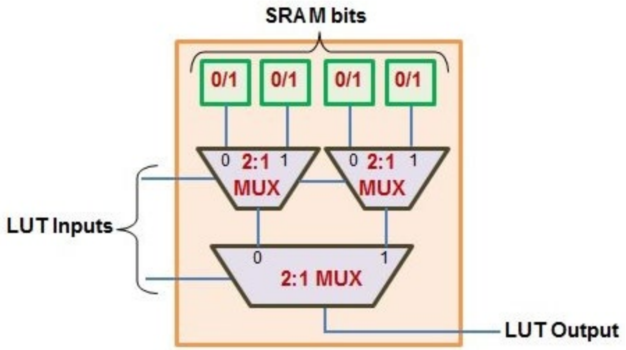
- CPLDs are one step more advanced than PAL/GAL
- combination of a fully programmable AND/OR array and a bank of macrocells
  - AND/OR array: reprogrammable, perform a multitude of logic functions
  - Macrocells: functional blocks, perform combinatorial or sequential logic
- Typically used for simple designs with small number of logic elements with limited internal routing (feedback)
- Predictable delay due to limited routing options, low density by today standards
- Non-volatile (maintains design on power cycle)

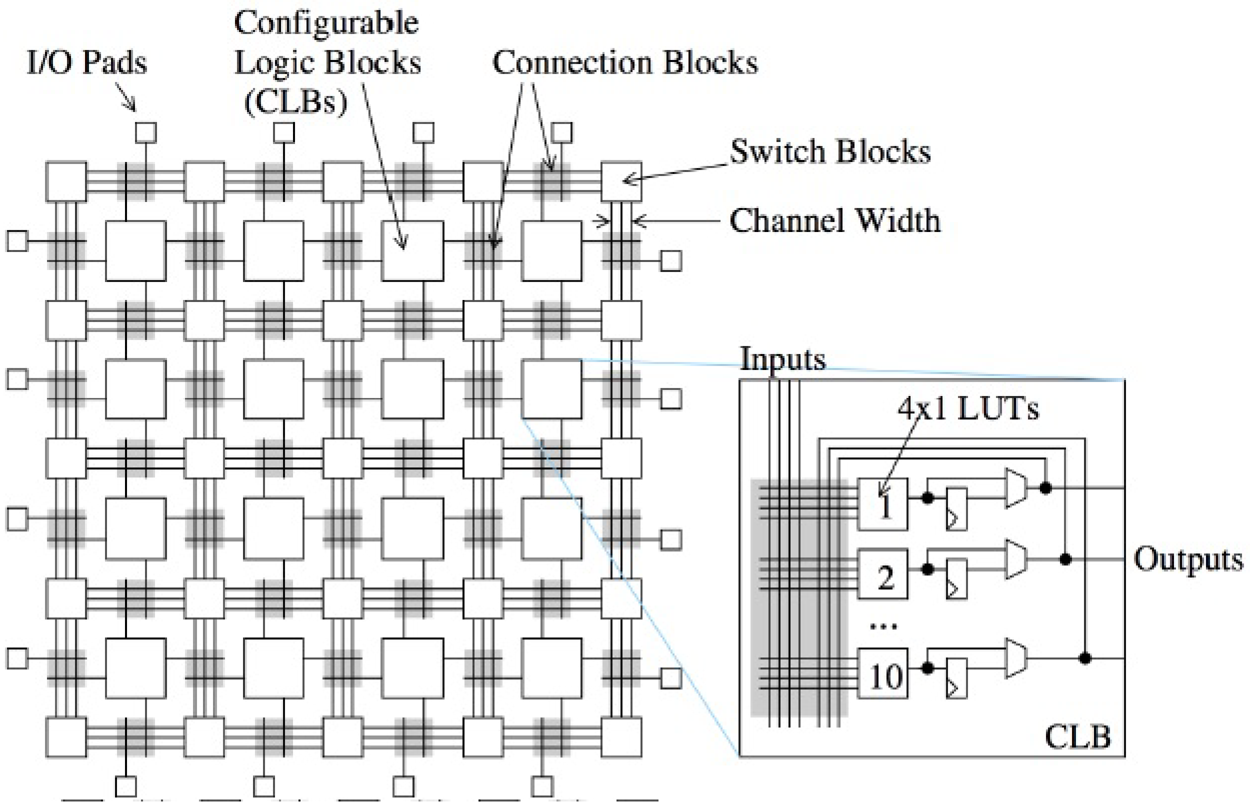




Field-Programmable Gate Array (FPGA)

- High-density programmable gate array
- Fully programmable through bit-stream configuration file
  - Programmable Logic
  - Programmable I/O
  - Programmable Interconnects (routing)
- Look-up table (LUT) based combinational logic
  - Can be implemented with SRAM (volatile)

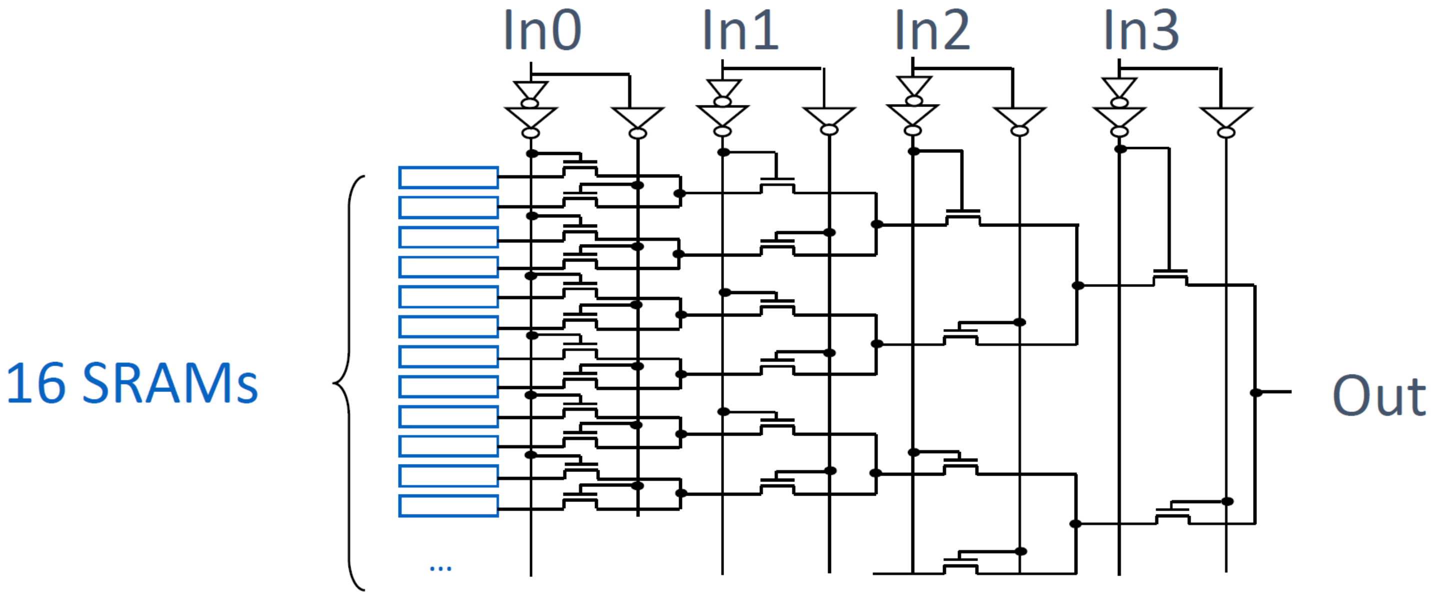




Field-Programmable: An electronic device or embedded system is said to be field-programmable or in-place programmable if its firmware can be modified “in the field”, without disassembling the device or returning it to its manufacturer. (Wikipedia)

Look-Up Table (LUT) Based Combinational Logic

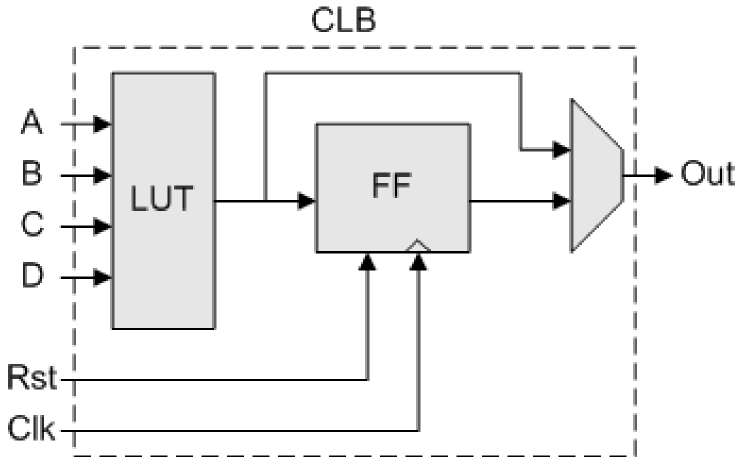
- FPGAs use LUTs (look-up tables) to express combinational logic
- Can be implemented with SRAM for high performance (this means FPGA is volatile, must be programmed on boot-up)
- E.g., 2:1 MUX can be directly implemented in single 4-input LUT (1 unused input)
- 4 input LUT is simply a 16x1 RAM

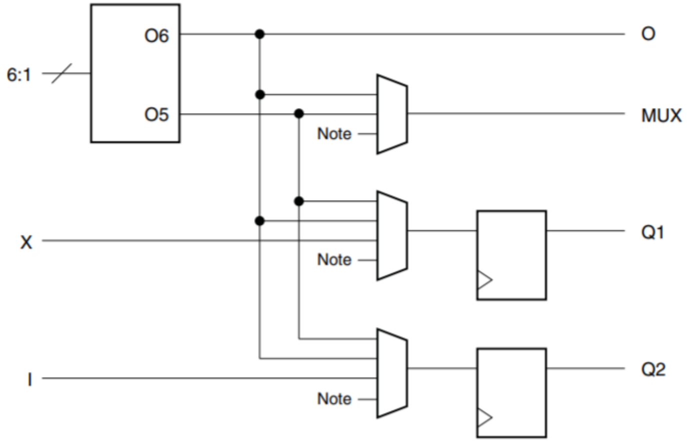


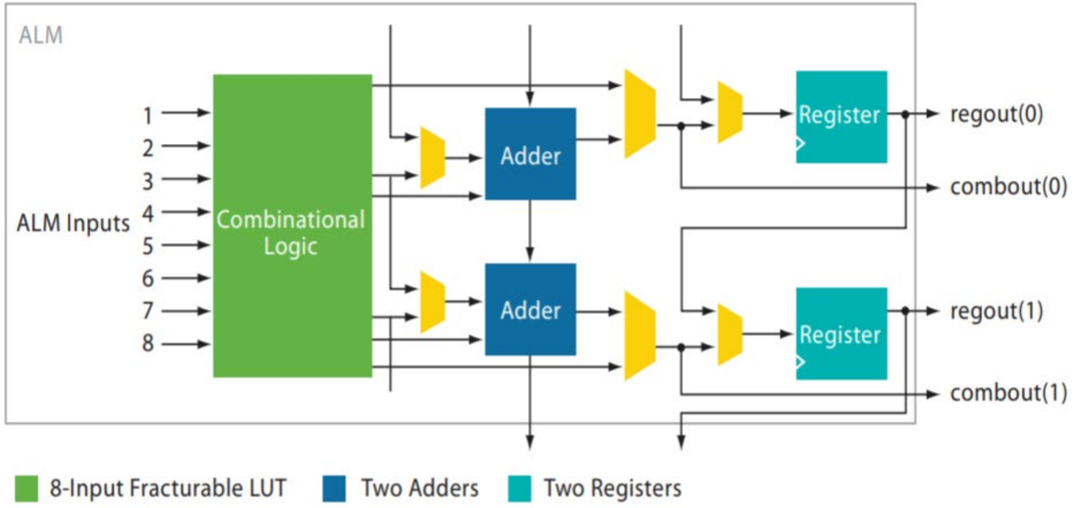
Out = f (in0, in1, in2, in3)

Configurable Logic Block (or Logic Element)

- Contains LUTs, flip-flops, MUX, etc.







Altera (Intel) Adaptive Logic Module (ALM)

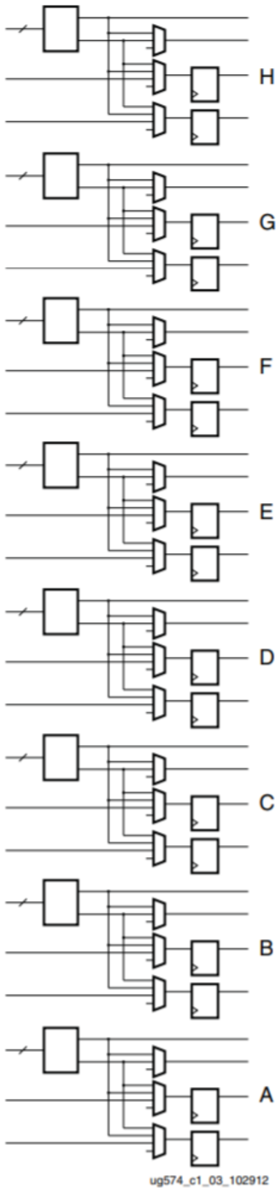
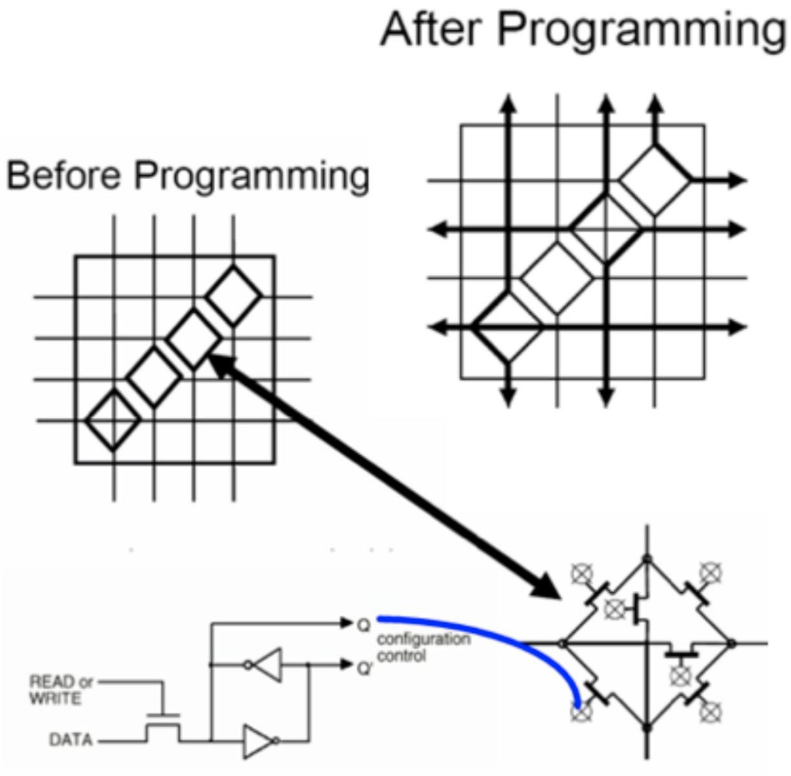


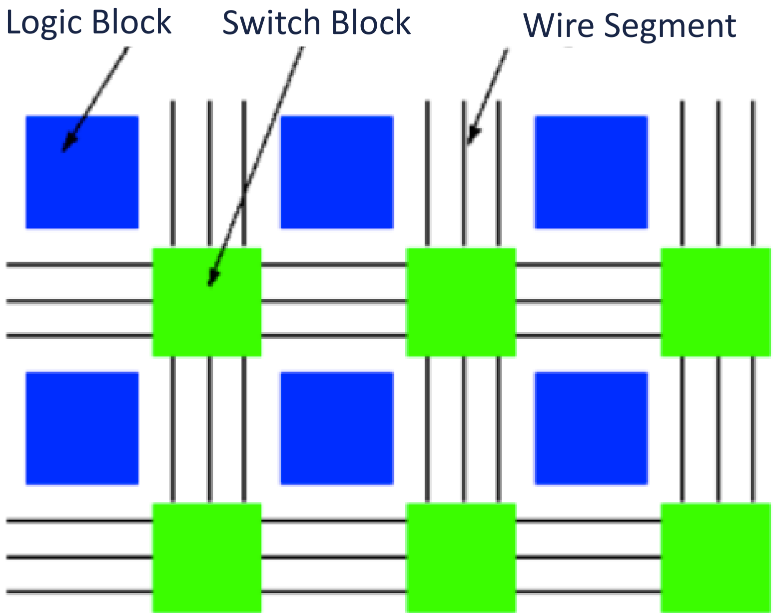
Figure 1-3: **LUTs and Storage Elements in One Slice**

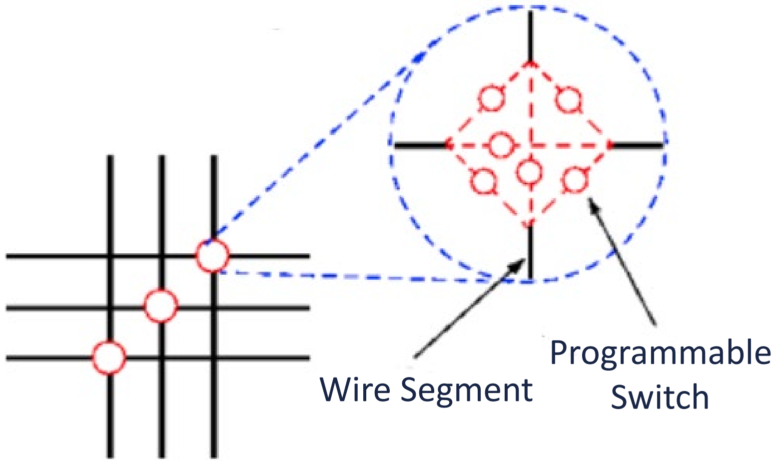
Xilinx UltraScale CLB Slice

Programmable Routing

- Between rows and columns of logic blocks are wiring channels
- Programmable: a logic block pin can be connected to one of many wiring tracks through a programmable switch



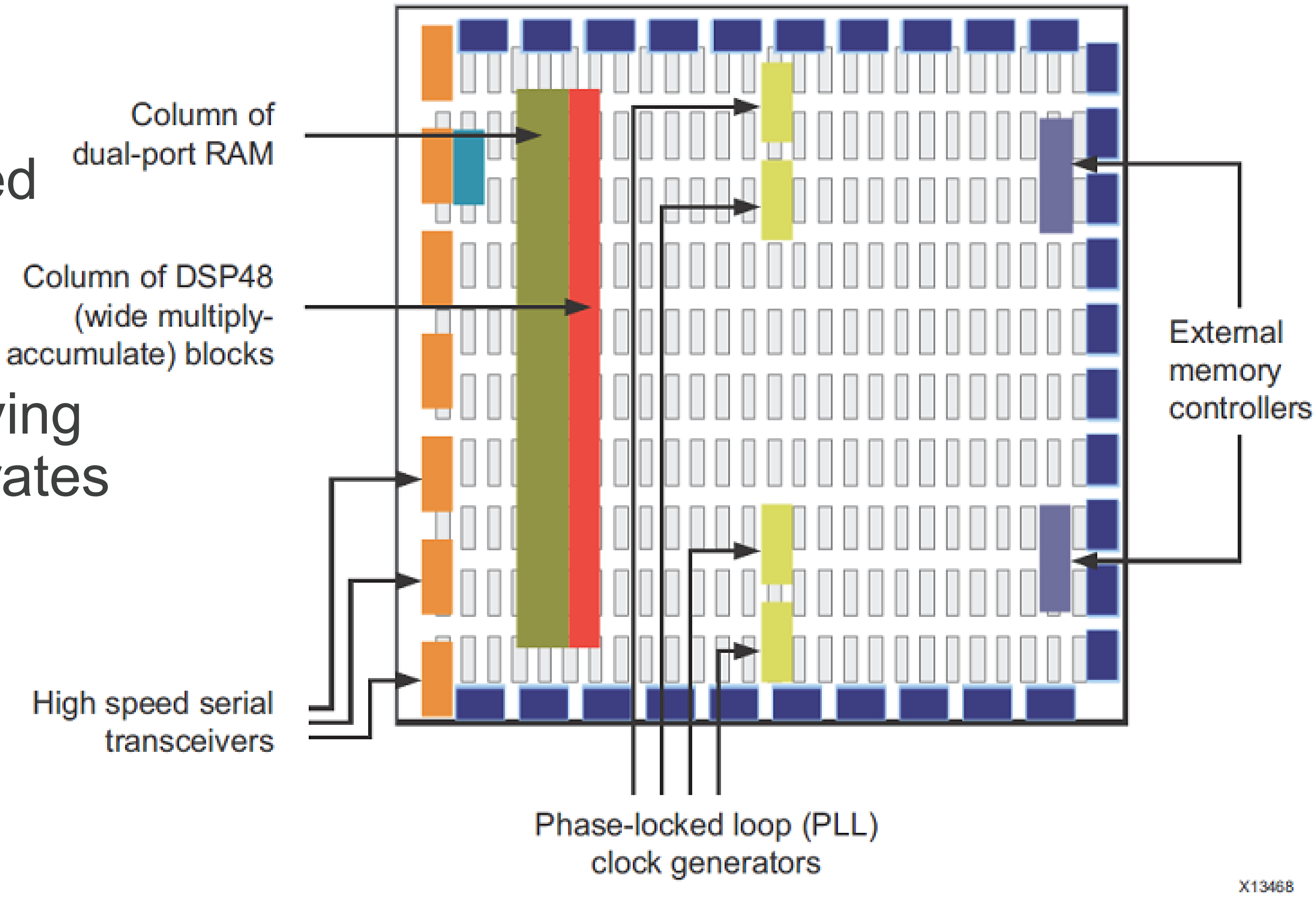




Other Hardware Resources in FPGAs

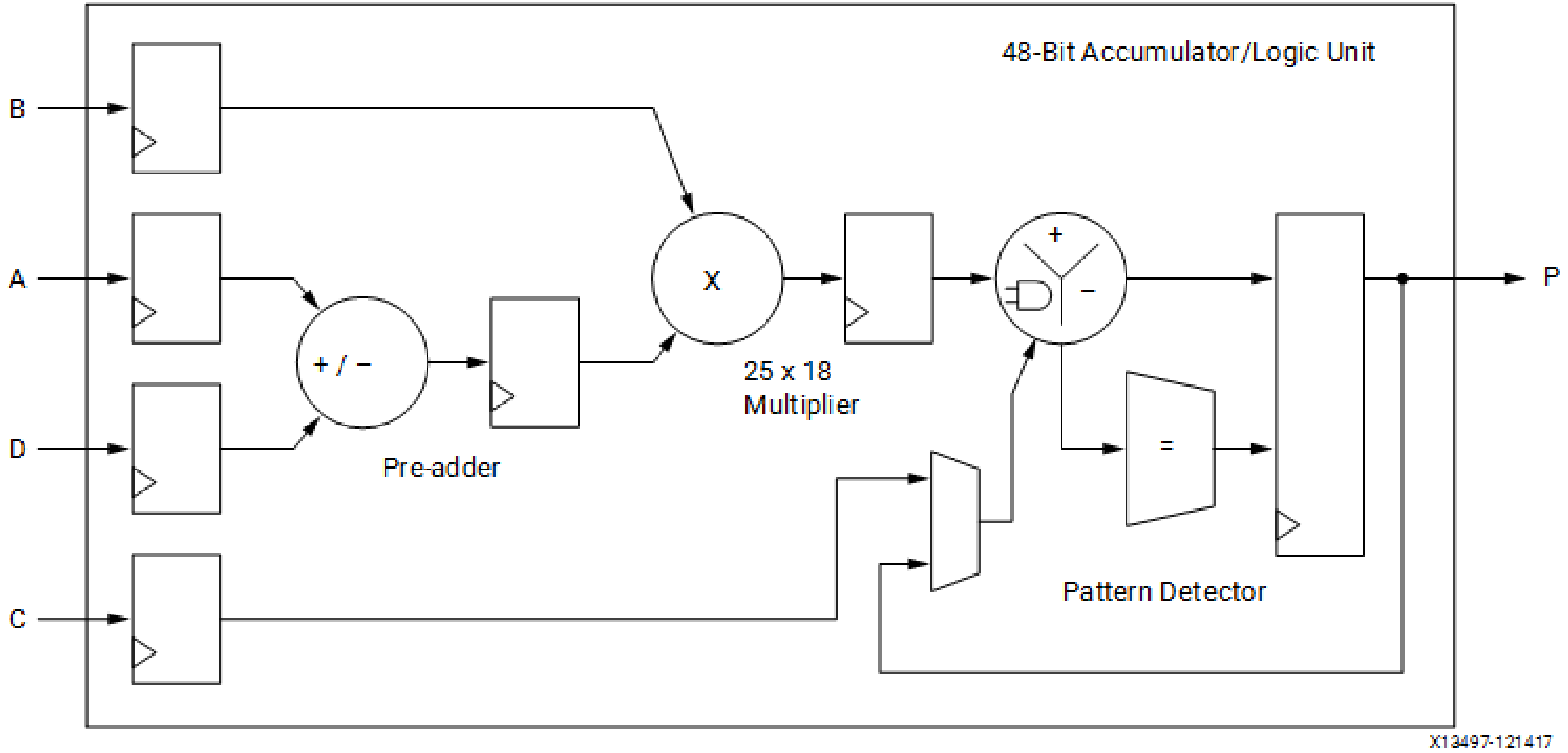
FPGAs have additional components

- High-speed serial transceivers
- Embedded memories for distributed data storage
- DSP blocks
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- Off-chip memory controllers
- PCIe controllers
- …



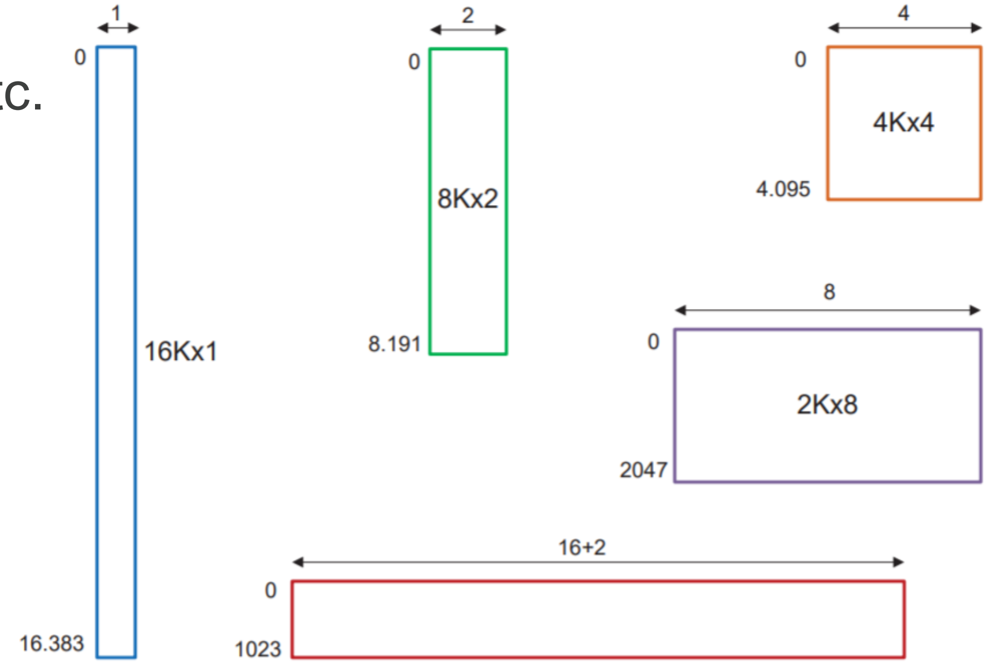
DSP48 Block in a Xilinx FPGA

- Can be used to implement "$P = B \times (A + D) + C$" or "$P += B \times (A + D)$"

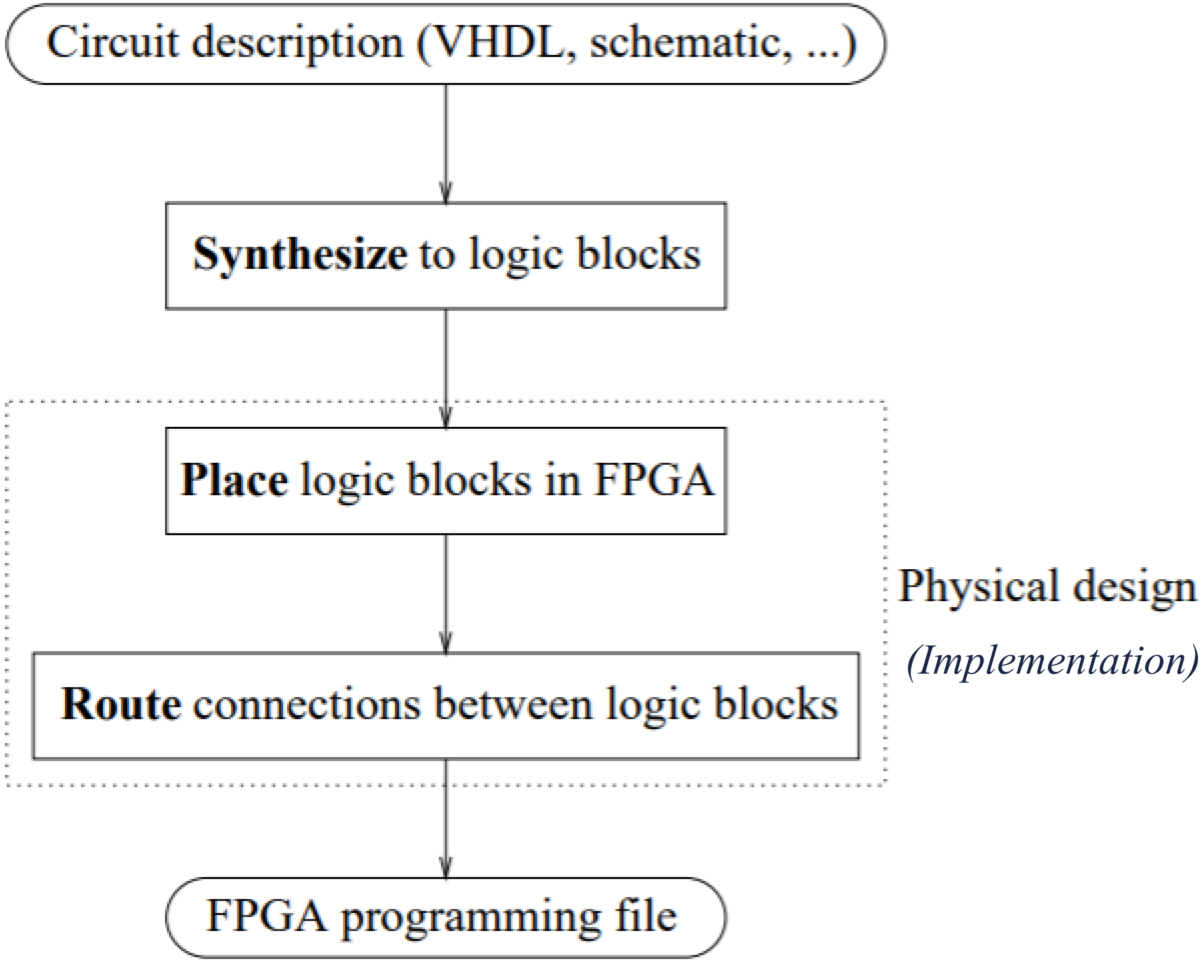


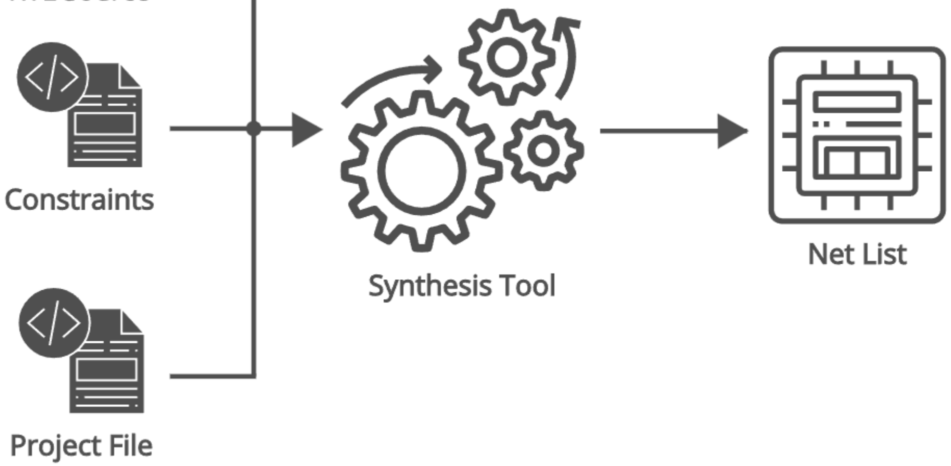
Block RAM (BRAM)

- BRAM is a dual-port RAM module instantiated into the FPGA fabric
- Can hold either 18k or 36k bits
  - Can be configured into different sizes
- Multiple configuration options
  - True dual-port, simple dual-port, single-port, etc.
- Two independent ports
  - Individual address, clock, write enable, clock enable, etc.
  - Independent widths for each port



FPGA Synthesis





Hardware Description Language (HDL)

- Specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits
- Two major languages: Verilog and VHDL
- Different types of description: dataflow, behavioral, and structural

Dataflow

```
module example1 ( e , a, b, do, dl,
d2, d3);
  input e, a, b;
  output do, dl, d2, d3;
  assign dO = ( e & ~a & ~b); //00
  assign dl = (e & ~a & b);   //01
  assign d2 = (e & a & ~b);   //10
  assign d3 = ( e & a & b);   //11
endmodu1e
```

Behavioral

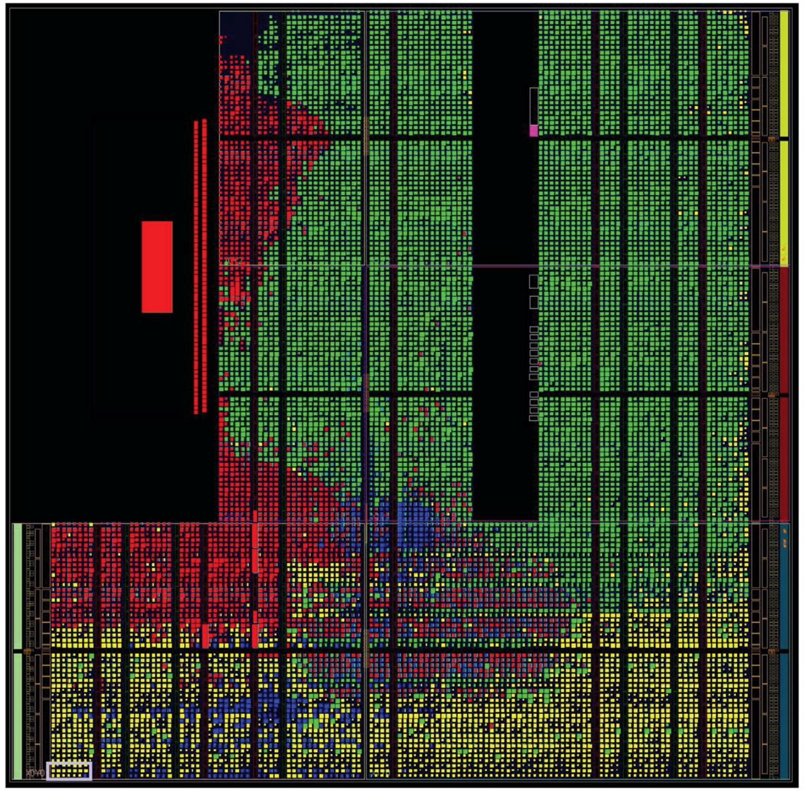
```
module example2 (e, i, d);
output [3:0] d;
input [l:0]i;
input e;
reg [3:0] d;
  always @ (i or e) begin
      if (e==l) begin
        case (i)
           0: d = 4'b 0001;
           1: d = 4'b 0010;
           2: d = 4'b 0100;
           3: d = 4'b 1000;
           default d = 4'b xxxx;
       endcase
      end
      else
        d = 4'b0000;
  end
endmodule
```

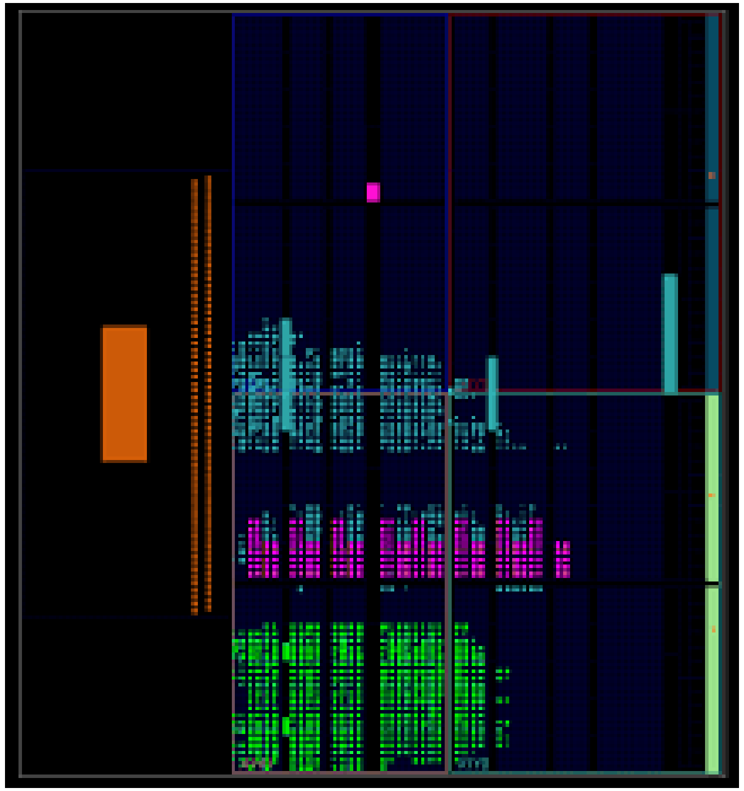
Structural

```
module build_xor (a, b, c);
 input a, b;
 output c;
 wire c, a_not, b_not;
   not a_inv (a_not, a);
   not b_inv (b_not, b);
   and a1 (x, a_not, b);
   and a2 (y, b_not, a);
   or out (c, x, y);
endmodule
```

Configuring the FPGA

- Programming an FPGA: download a *bitstream* (not a program, but a configuration) to an FPGA
- All FPGA components (logic blocks, interconnects, etc.) are configured using the bitstream so that they implement the correct functionalities





Reading Materials

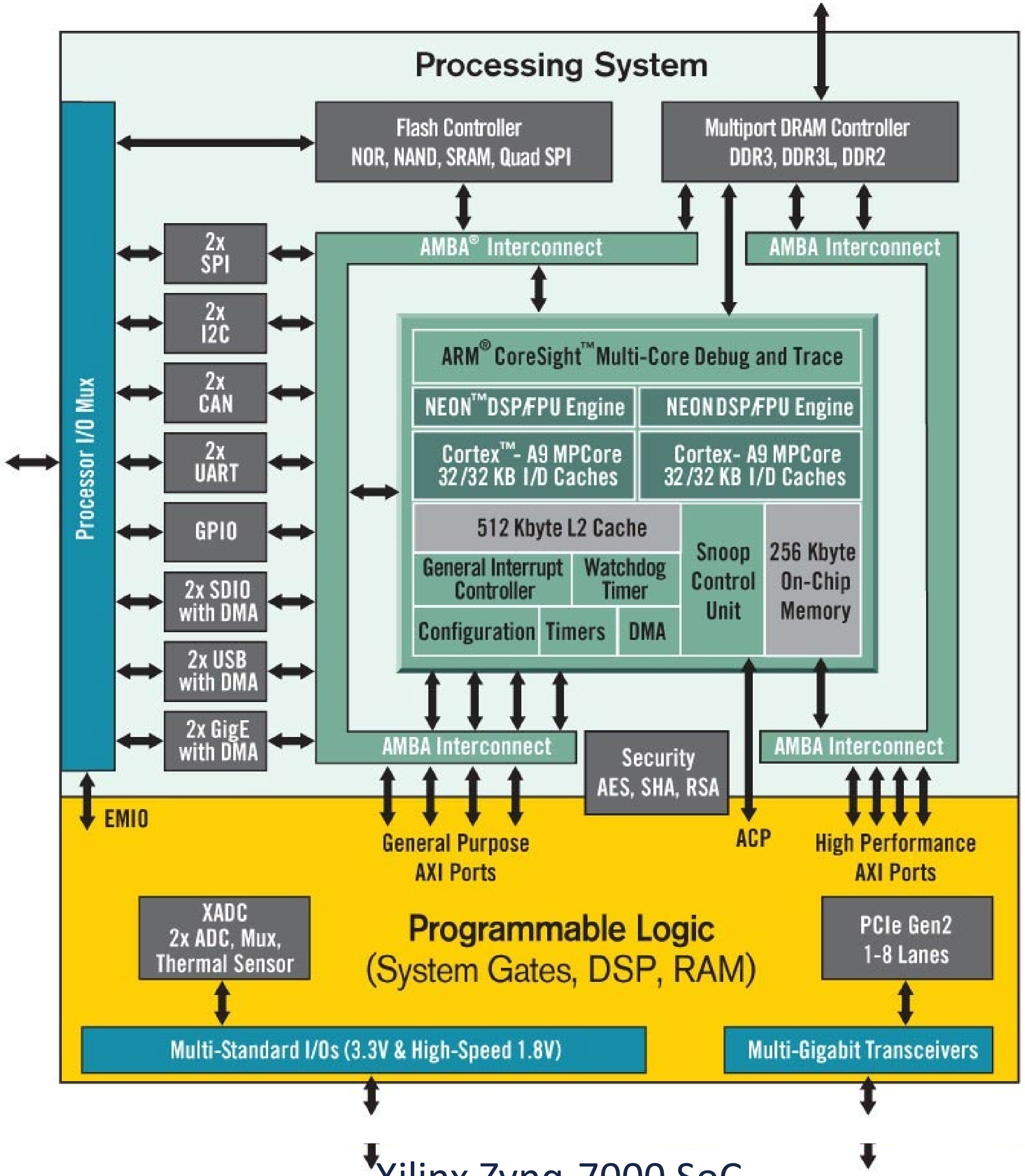
- Andrew Boutros and Vaughn Betz, FPGA Architecture: Principles and Progression. 2021. [PDF]
- FPGA Architecture Basics. https://www.rapidwright.io/docs/FPGA_Architecture.html

Xilinx Zynq SoC Overview

- SoC can be divided into two parts:
  - ***PS*** : Processing System (ARM CPU)
  - ***PL***: Programmable Logic (FPGA)

Note (for Xilinx Zynq-7000 SoC on the right):

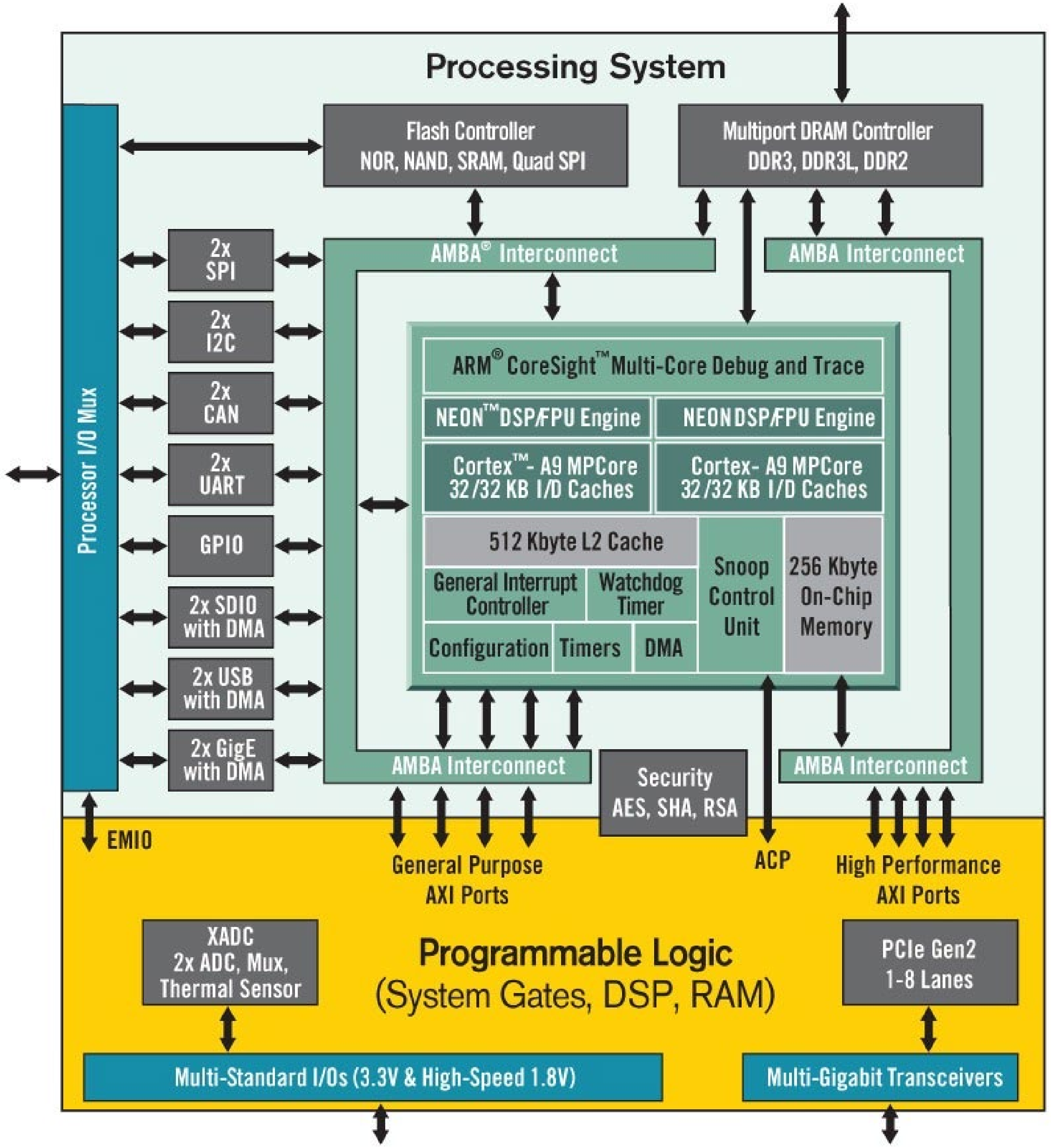
- No dedicated video controller or GPU
- Peripherals can be connected to PS or PL



Xilinx Zynq-7000 SoC

Xilinx Zynq SoC Overview

- PS: Dual Core ARM A9
  - 1-2 GOPS
- Floating point support
  - ARM NEON support as well
- Up to 1GHz Clock
- L2 Cache
  - Unified 512KB
- AXI High performance interconnects
- 256KB on-chip memory scratchpad
- PL: FPGA
  - 85K Logic cells + 220 DSP slices
  - 10-100 GOPS!



Xilinx Zynq-7000 SoC

Zynq SoC Boot Flow

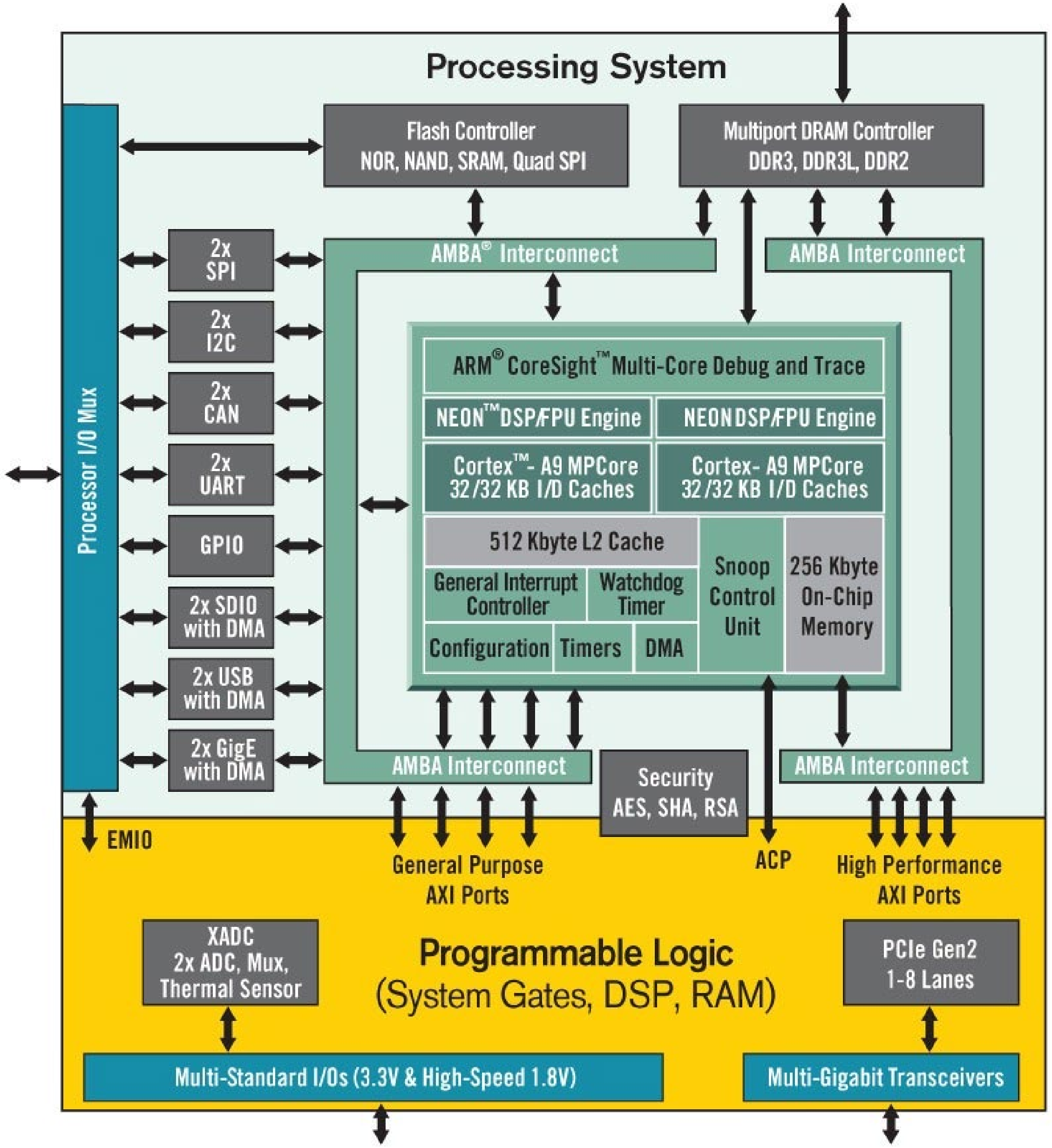
Multi-Stage

1. Run from ROM
   a) Copy FSBL from boot device to OCM (on-chip memory)
2. First Stage Boot Loader (FSBL)
   a) Load Uboot from boot device to DDR
   b) Program PL
   c) Initiate PS boot
3. Uboot
   a) Load linux kernel to DDR
   b) Device tree init
   c) FPGA init
4. OS Boot
   a) Linux Boot

More Info:
http://www.wiki.xilinx.com/Getting+Started
http://xillybus.com/tutorials/device-tree-zynq-1

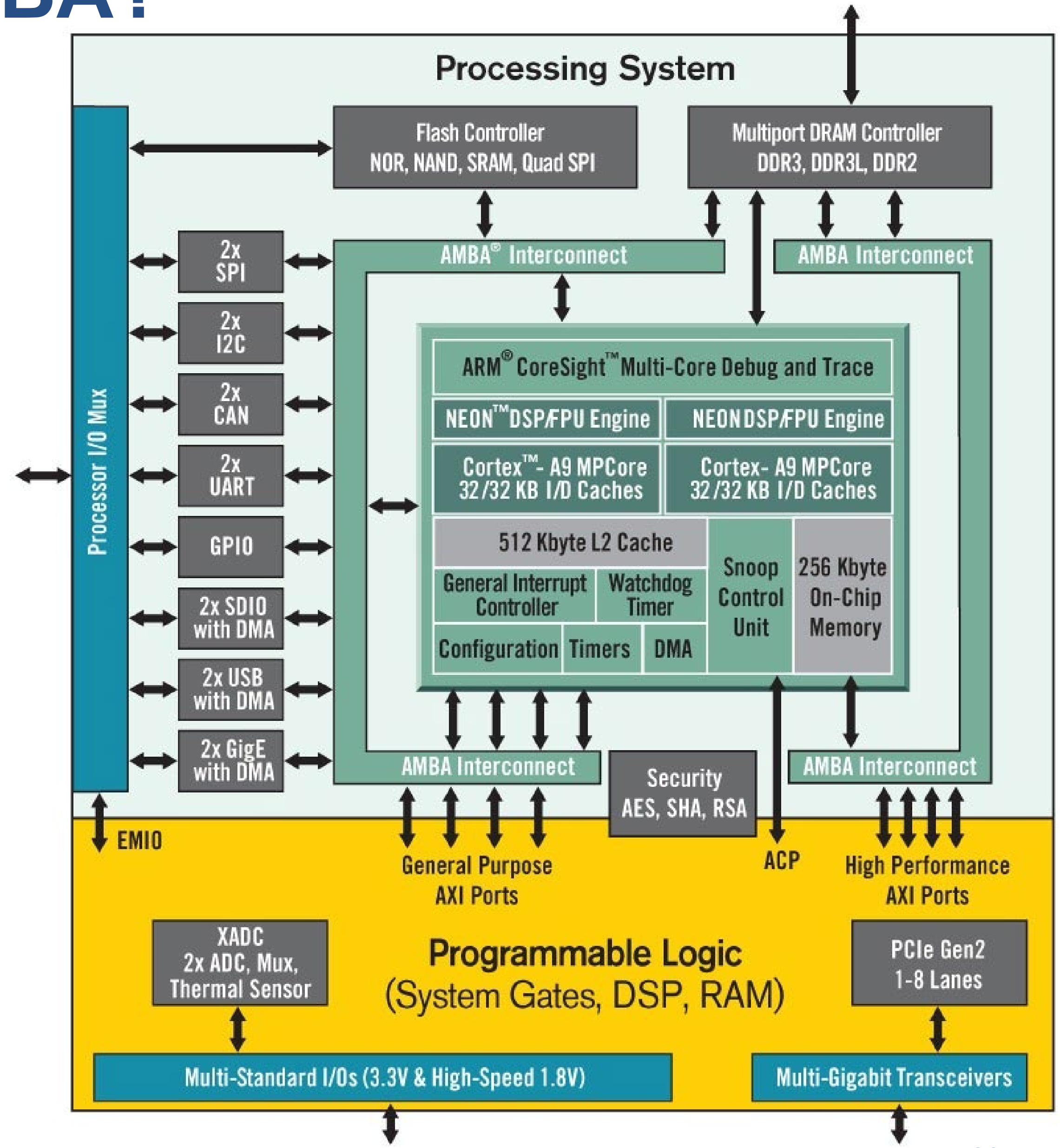
Xilinx Zynq SoC: Interrupts

- Several interrupts available for PS-PL interface
- 16 peripheral interrupts available from PL to PS
  - Use for accelerator dev
- PS to PL interrupts exists as well
  - Read up via manual



Xilinx Zynq SoC: What is AXI/AMBA?

- AMBA: Advanced Microcontroller Bus Architecture
  - Protocol
  - Open standard, on-chip interconnect for SoC
- AXI:
  - Advanced eXtensible Interface
  - Very common AMBA interface
- Why:
  - Flexible
  - IP reuse
  - Ease of use



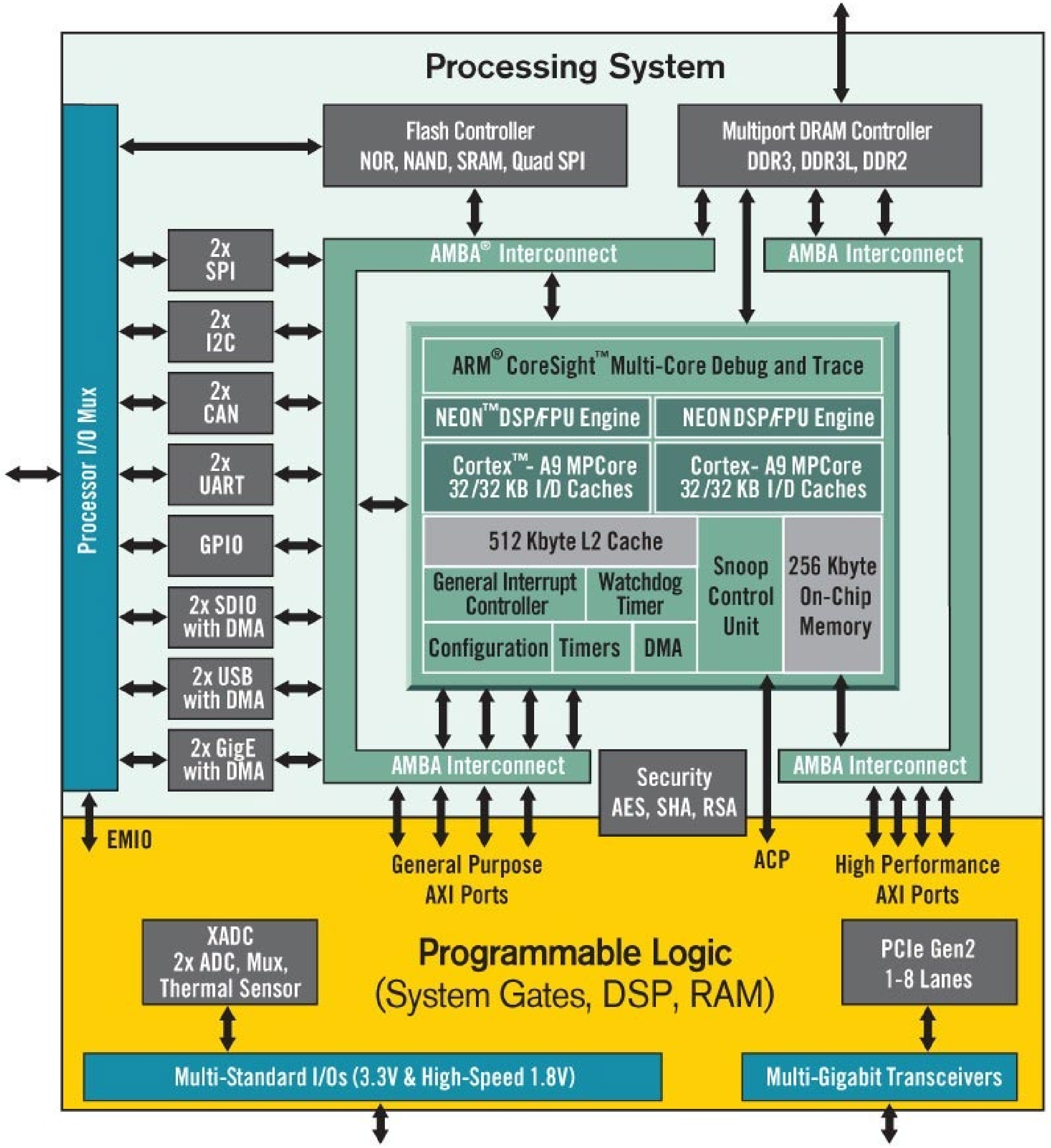
Xilinx Zynq SoC: AXI

AXI Interfaces:

- AXI4 ***Mainly for Data Movement***
  - High performance
  - Memory mapped
- AXI4-Lite ***Mainly for Control and Status***
  - Low throughput
  - Memory mapped
- AXI4-Stream ***Mainly for Data Movement***
  - High performance
  - Streaming Data

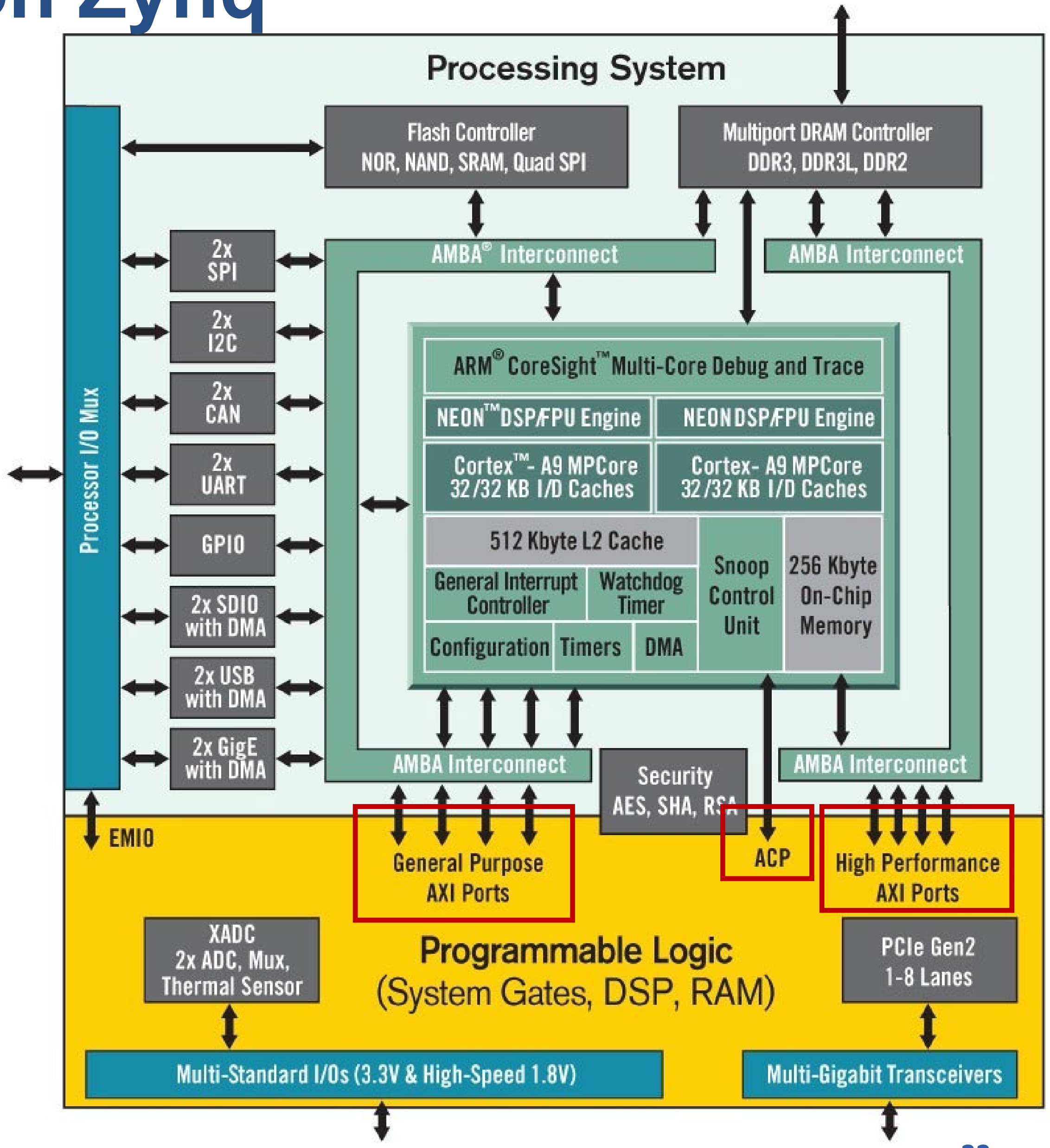
More Info:

http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

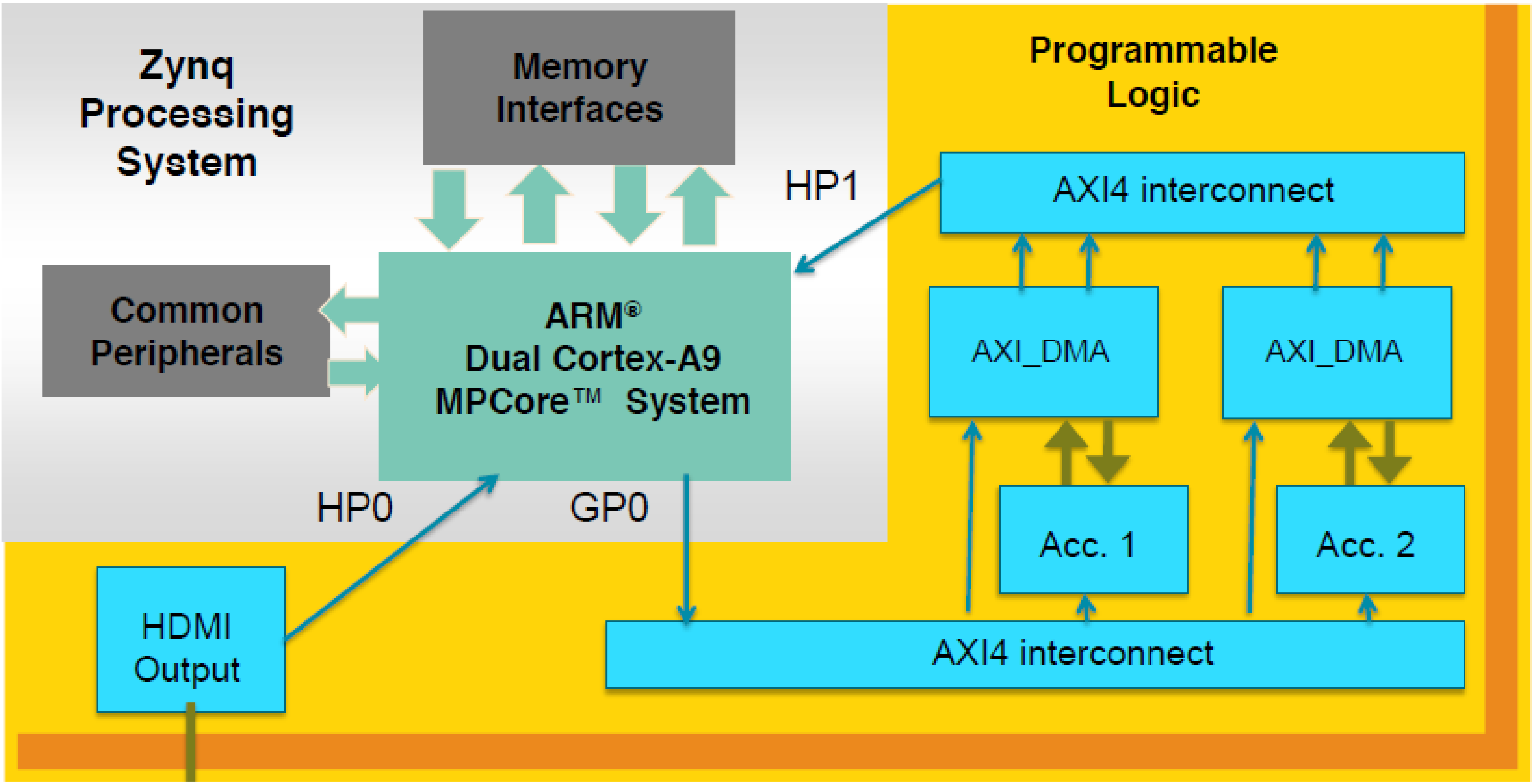


Xilinx Zynq SoC: AXI Interfaces on Zynq

- HP (High Performance): ***Mainly for burst data movement***
  - 4x64bit Slave
  - ***High bandwidth*** access to external memory
    - @150MHz, bandwidth = 9.6GB/s
  - Large Data bursts
- GP (General Purpose): ***Mainly for Control and Status***
  - 2x32bit Slave
    - PL to PS peripherals
  - 2x32bit Master
    - PS to PL access
- ACP (Accelerator Coherency Port): ***for cache-coherent data access***
  - 1x64bit Slave
    - PL accesses processor L2 cache
  - Hardware coherence

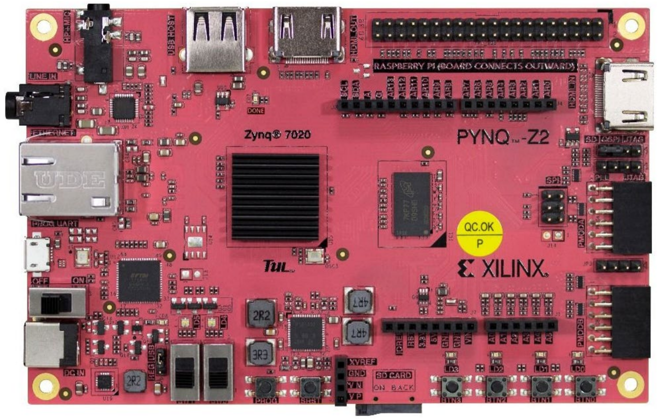


Xilinx Zynq SoC: Example



PYNQ Boards

- PYNQ-Z2 contains a Zynq-7000 SoC
  - Zynq SoC: ZYNQ XC7Z020-1CLG400C
    - 650MHz dual-core Cortex-A9 processor
    - Programmable logic equivalent to Artix-7 FPGA
      - 13,300 logic slices
      - 630 KB of fast block RAM
      - 220 DSP slices
  - Memory
    - 512MB DDR3 with 16-bit bus @ 1050Mbps
    - 16MB Quad-SPI Flash
    - MicroSD Slot
  - USB and Ethernet
    - Gigabit Ethernet PHY
    - Micro USB-JTAG Programming circuitry
    - Micro USB-UART bridge
    - USB 2.0 OTG PHY (supports host only)



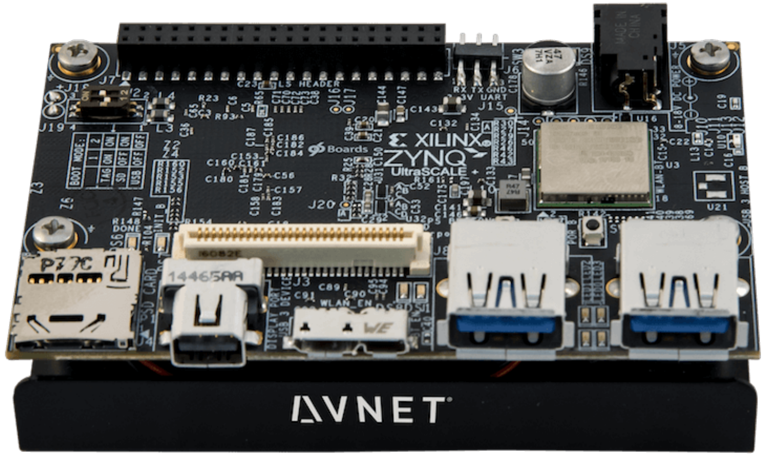
- Audio and Video (input/output)
- Switches, Push-buttons and LEDs
- Expansion Connectors
  - Two standard Pmod ports
  - 16 total FPGA I/O
  - Arduino connector
  - Raspberry Pi connector

Supports PYNQ programming environment

PYNQ-Z2 specs: https://www.tul.com.tw/images/PYNQ-Z2_PA_v2_pp_20201209_STD.pdf
PYNQ-Z2 user manual: https://dpoauwgwqsy2x.cloudfront.net/Download/PYNQ_Z2_User_Manual_v1.1.pdf
PYNQ-Z2 setup guide: https://pynq.readthedocs.io/en/v2.6.1/getting_started/pynq_z2_setup.html

Ultra96 Boards

- Ultra96 contains a Zynq UltraScale+ MPSoC
  - MPSoC: Xilinx Zynq UltraScale+ MPSoC ZU3EG A484
  - Memory: Micron 2GB LPDDR4 memory
  - Storage: Delkin 16GB microSD card + adaptor
  - Wireless: 802.11b/g/n Wi-Fi and Bluetooth 4.2
  - USB and Ethernet
    - 1x USB 3.0 Type Micro-B upstream port
    - 2x USB 3.0, 1x USB 2.0 Type A downstream ports
  - Display: Mini DisplayPort (MiniDP or mDP)
  - Expansion interface:
    - 40-pin 96Boards Low-speed expansion header
    - 60-pin 96Boards High speed expansion header



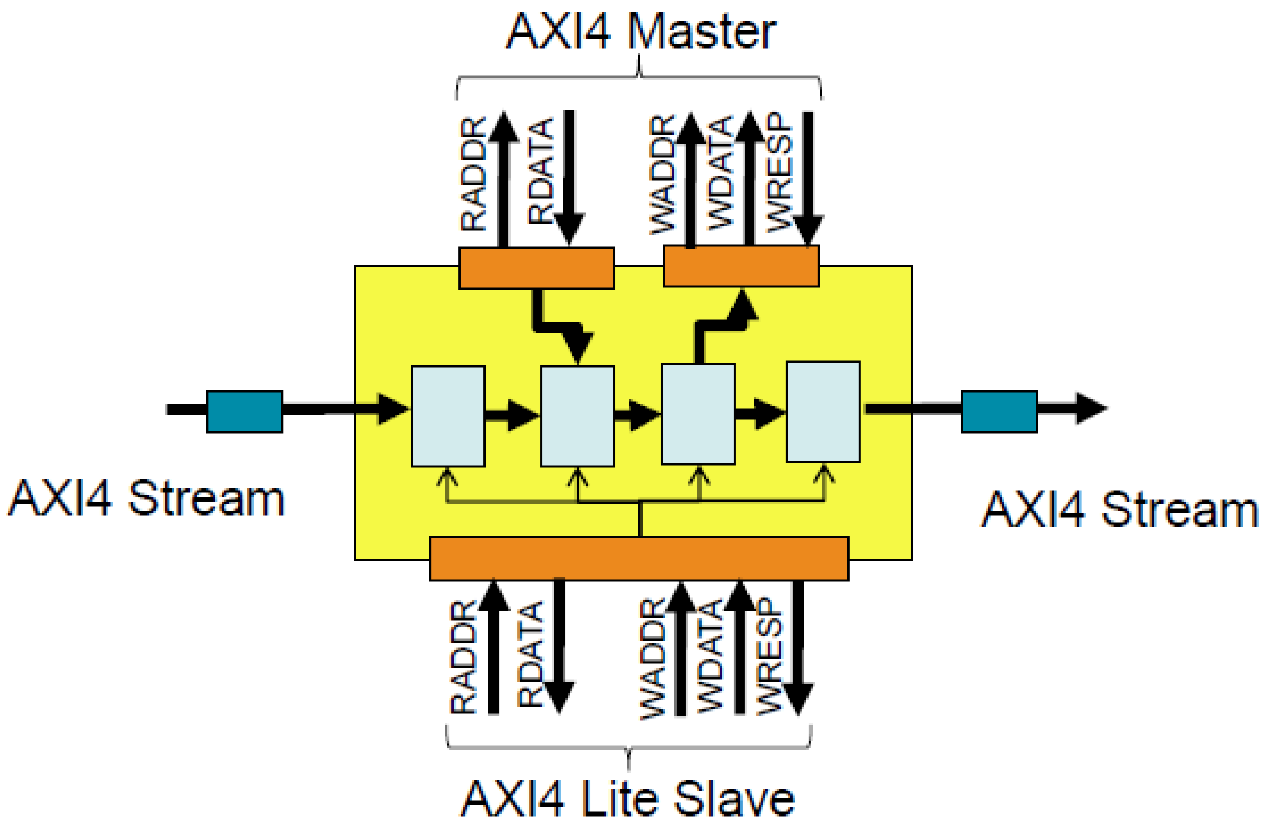
Supports PYNQ programming environment

Ultra96-v2 user's guide: https://www.element14.com/community/servlet/JiveServlet/downloadBody/92688-102-2-395912/Ultra96-V2-HW-User-Guide-v1_1.pdf
Ultra96-v2 PYNQ setup guide: https://ultra96-pynq.readthedocs.io/en/latest/getting_started.html

Zynq SoC: Accelerator Development Example

- Example:
  Design a vector add accelerator to perform
  c[i] = (a[i] + b[i])/N
  for given vectors
  - Variable size arrays
  - Performance and power
  - Arrays to initialized in software
  - N is constant

- Choose Interface:
  - How should I move data to FPGA?
  - Stream or memory-mapped?
- Latency calculations
  - Time to process/move data
- Resource Usage
  - How much parallelism is available?
- On-FPGA memory
  - Can all vectors be stored on FPGA?
- HW-SW co-design



Zynq SoC: Accelerator Development Example

- Simplify Example (sequential Vector Add)

```
for(i=0; i<N; i++)
    C[i] = A[i] + B[i]
```

- Basic Performance modeling/estimation:
  - Break down into smaller operations
  - Compute time per operation
  - Eg. Read latency = r, Write latency = w, floating point add latency = c

- Vector Add
  - Repeat N times
    - LOAD A[i]
    - LOAD B[i]
    - ADD C[i], A[i], B[i]
    - STORE C[i]

Total time = N*(2r + c +w)

Zynq SoC: Accelerator Development Example

Total time = N*(2r + c +w)

- What does this tell us?
  - How large does N need to be before it makes sense to accelerate this?
  - For each compute operation, 12 bytes of data is moved
  - How do I bring data in?
    - MEMORY < --- > PS < --- > PL
    - MEMORY < --- > PL

- Bandwidth vs Latency:
  - Bandwidth:
    How much data can you bring in per unit time.
  - Latency:
    How long does it take data to arrive.
- To perform K operations in parallel, you need K*12bytes
  - B/W needed = K*12*Frequency
  - Choose K wisely

Zynq SoC: Accelerator Development Example

- Is 100GOPS really possible?
  - Yes and No
- Assumption: Each op is floating point
  - 8 bytes per op
  - Total bandwidth needed 800Gb/s?

Data input throughput needed

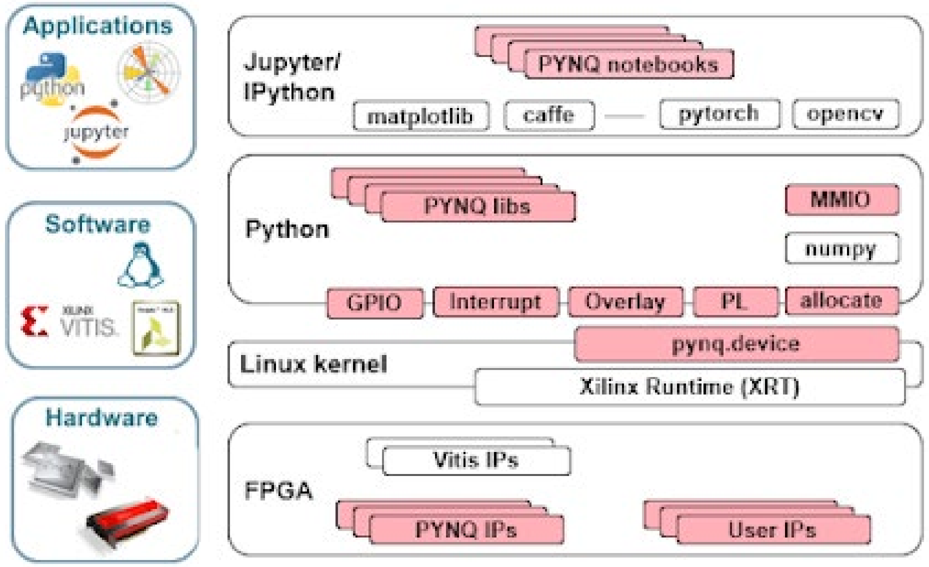
Compute throughput

Data output throughput needed

- Look to Data Reuse
  - On-Chip memory can be a limit
- Streaming/Systolic Design:
  - Smaller blocks, feeding into each other
- Be smart about bandwidth
  - Dedicated read and dedicated write channels may not be smart
    - Think about communication patterns
    - Do you need concurrent read and write?

What is PYNQ?

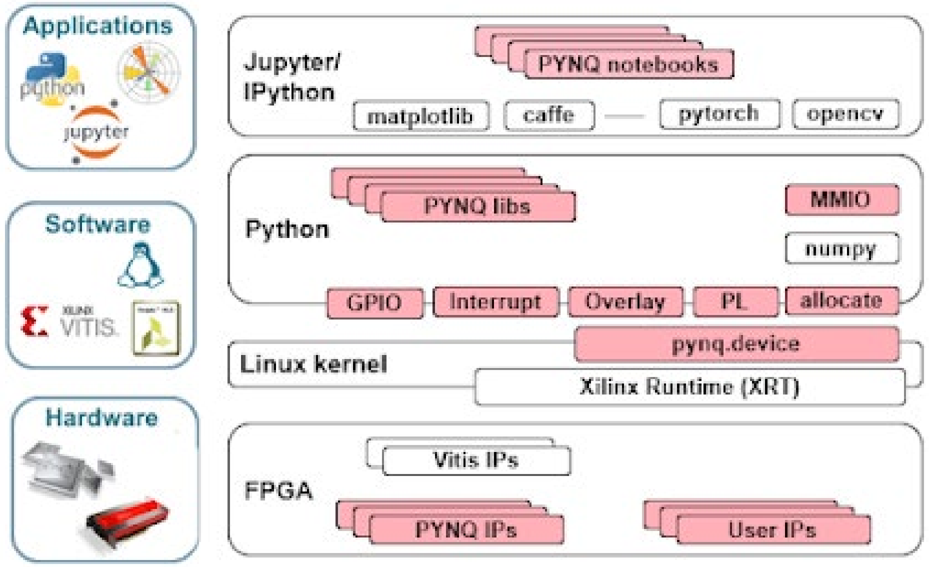
- An open-source project from Xilinx® that makes it easier to use Xilinx platforms
- ***Python***-based APIs and libraries
- Simplifies ***host*** programming
- Supports wide range of Xilinx devices:
    - Zynq, Zynq UltraScale+, Zynq RFSoC, MPSoC, Alevo, AWS-F1, etc.



What is PYNQ?

Key Technologies

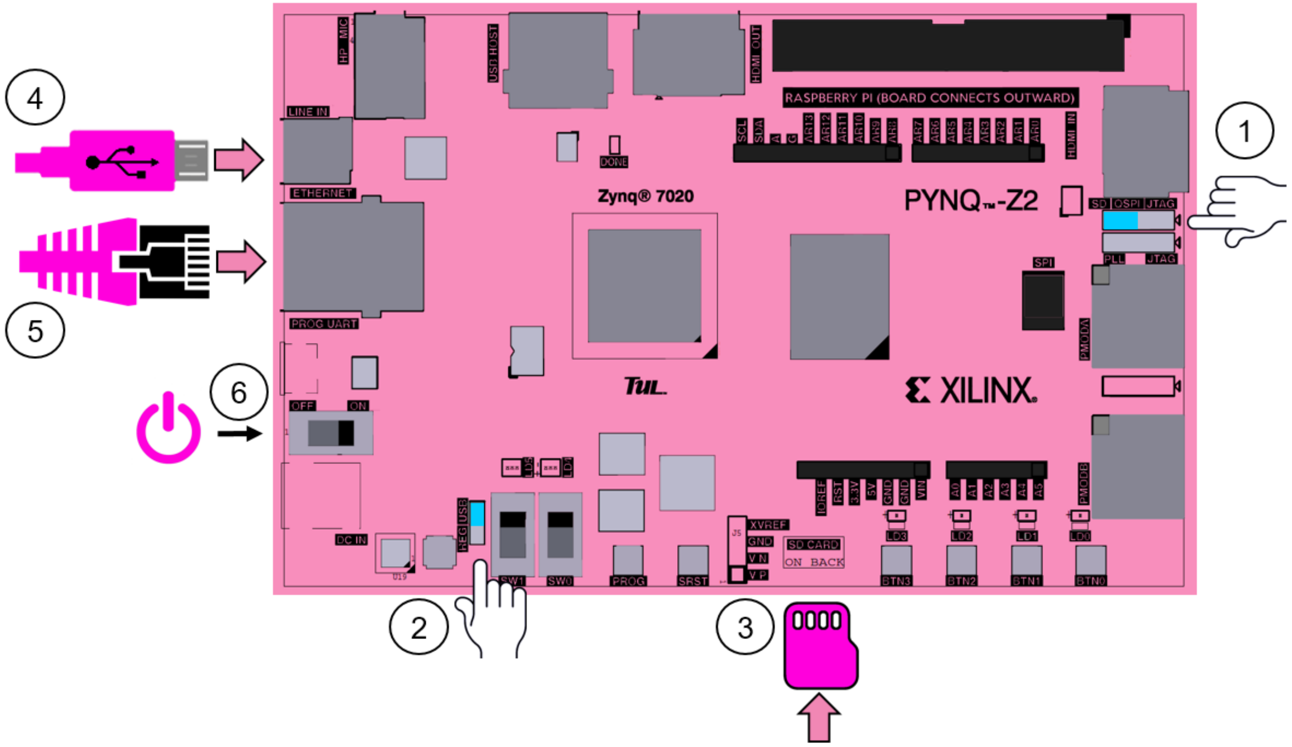
- Jupyter Notebook: a browser based interactive computing environment
- PYNQ enabled FPGA boards can be programmed in Jupyter notebook using Python (host programming)
- PYNQ is delivered in two forms:
  - Bootable Linux image for Zynq boards
  - Open-source Python package for Alveo and AWS-F1



Getting Started with PYNQ

Using PYNQ-Z2 board as example

1. Set the ***Boot*** jumper to the *SD* position (boot from the Micro-SD card)
2. To power the board from the micro-USB cable, set the **Power** jumper to the *USB* position. (You can also power the board from an external 12V power regulator by setting the jumper to *REG*.)
3. Insert the Micro SD card loaded with the PYNQ-Z2 image into the **Micro SD** card slot underneath the board
4. Connect the USB cable to your PC/Laptop, and to the **PROG - UART** Micro-USB port on the board
5. Connect the Ethernet port by following the instructions below
6. Turn on the PYNQ-Z2

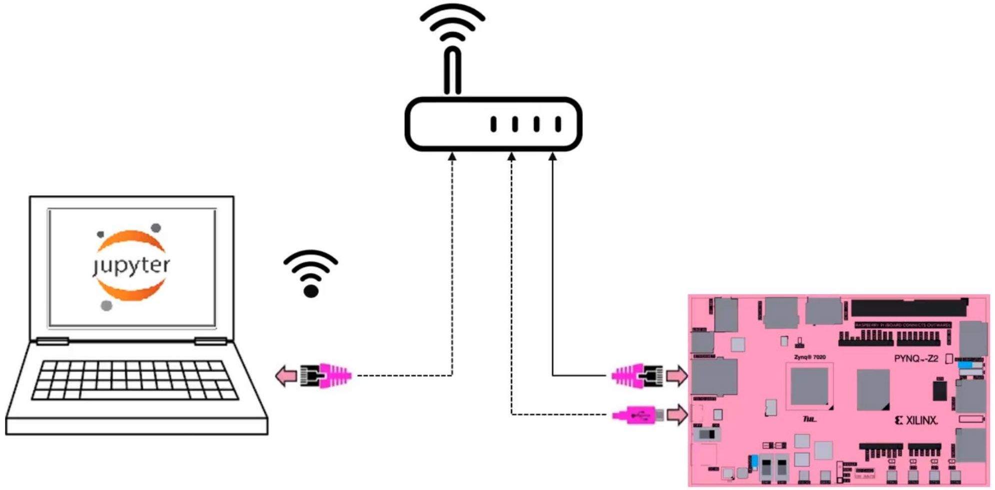


Getting Started with PYNQ

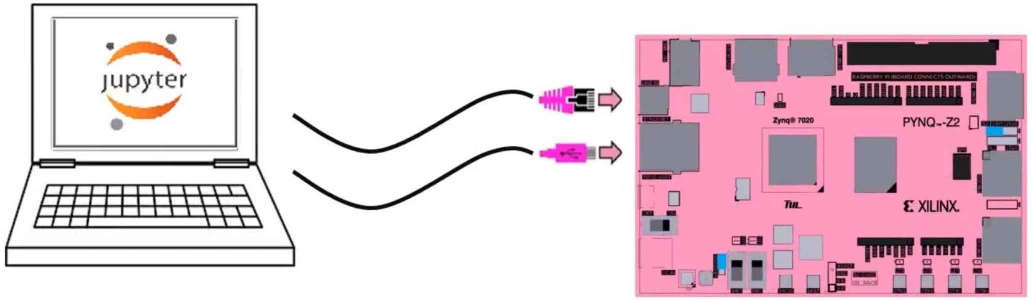
Network connection

Once your board is setup, you need to connect to it to start using Jupyter notebook

- Option A: Connect to a Network Router
  - Connect the Ethernet port on your board to a router/switch
  - Connect your computer to Ethernet or Wi-Fi on the router/switch
  - Browse to http://<board IP address>
- Option B: Connect to a Computer
  - Assign your computer a static IP address
  - Connect the board to your computer's Ethernet port
  - Browse to http://192.168.2.99



Option A: Connect to a Network Router



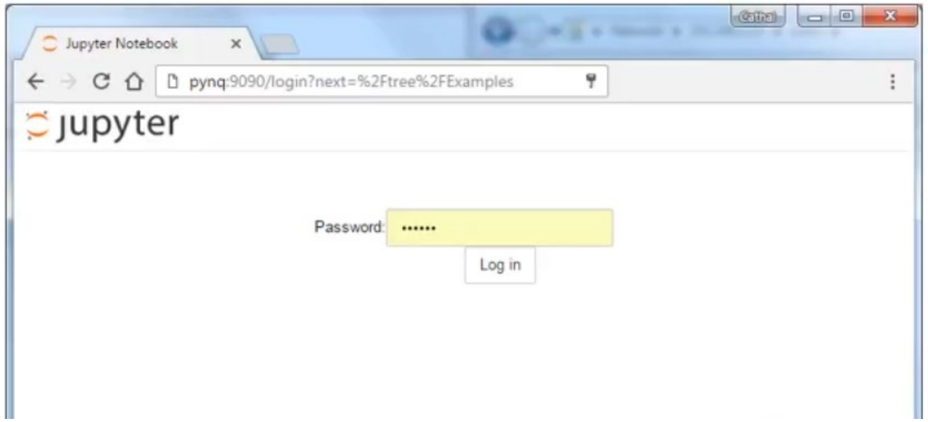
Option B: Connect to a Computer

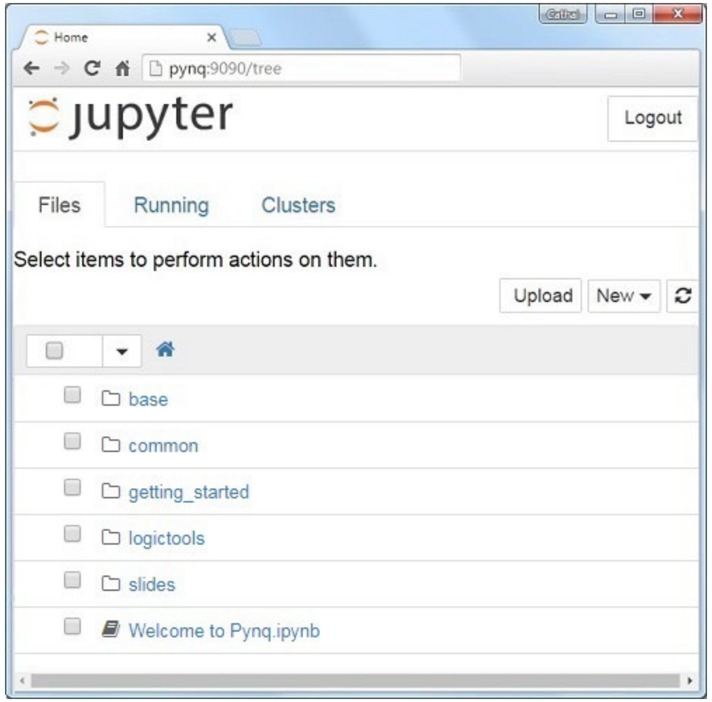
Getting Started with PYNQ

Connecting to Jupyter portal

After network connection is established, browse to board's corresponding IP address, then you can start to use the board using Jupyter notebook.

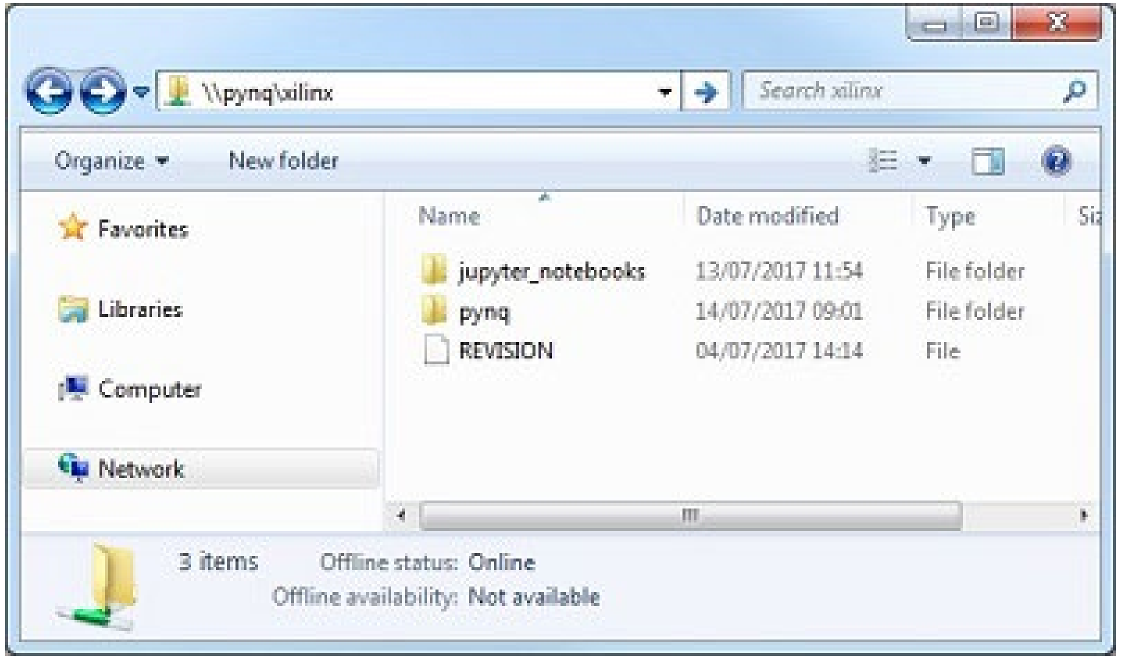
- For boards connected to network:
  - Browse to http://pynq:9090
- For boards connected to computer:
  - Browse to http://192.168.2.99:9090
- Default password: xilinx





Getting Started with PYNQ

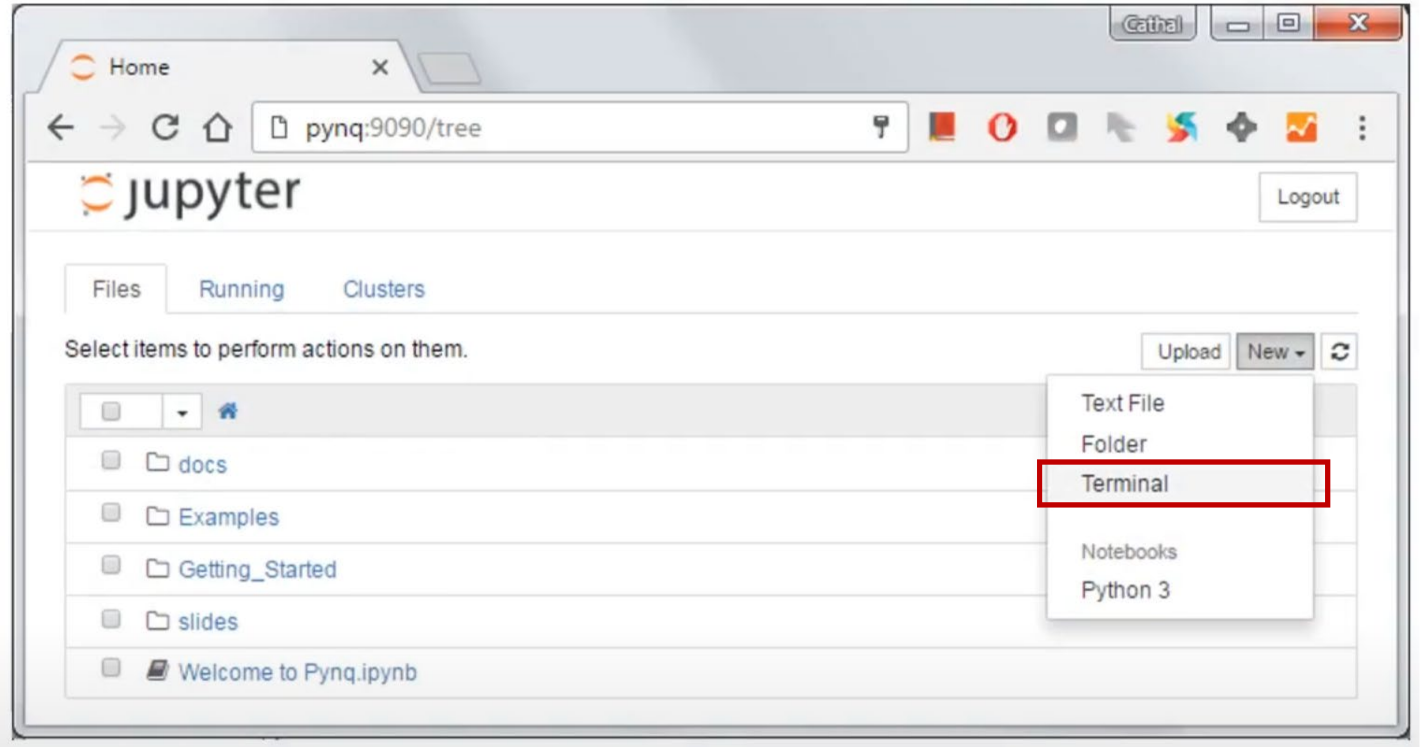
Connect via Samba

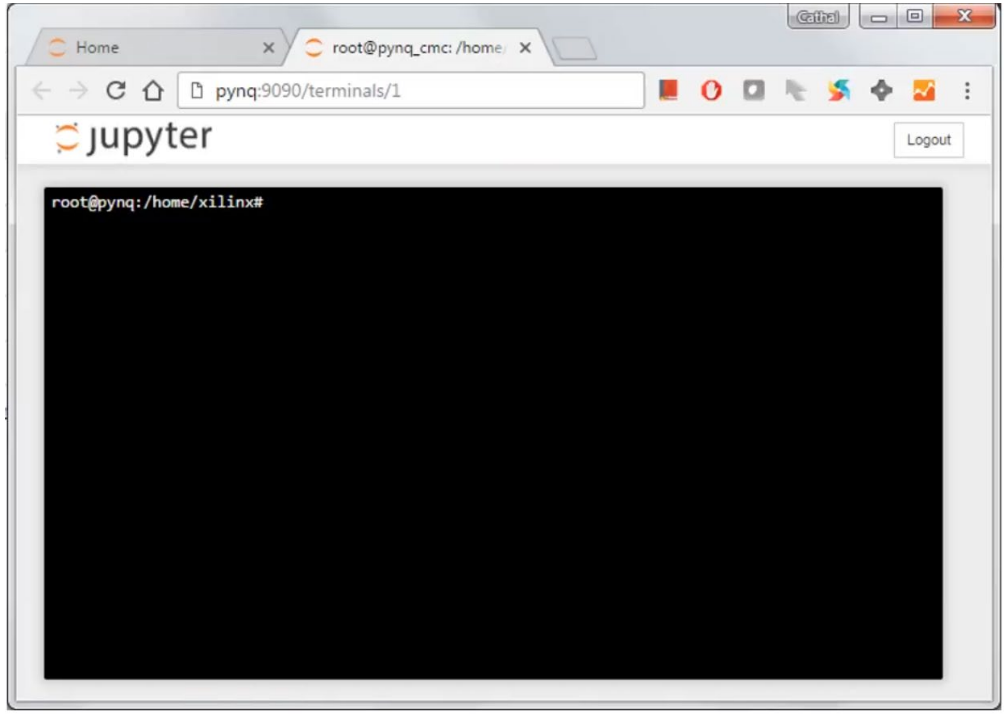


- Windows: [\\pynq\xilinx](\\pynq\xilinx)
- Mac or Linux: smb://pynq/xilinx

Getting Started with PYNQ

Jupyter terminal





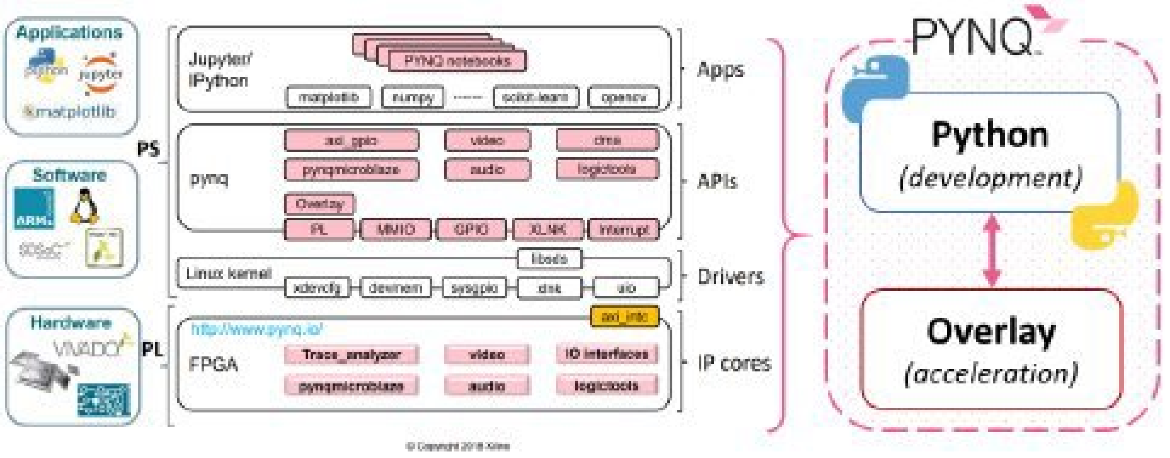
The PS runs a full Linux OS, so you can also ***ssh*** to the system:

```
ssh pynq@192.168.2.99
```

PYNQ Overlays

What are overlays?

- “hardware libraries”, where IP instances become objects
- programmable/configurable FPGA designs
- Can be used in a similar way to a software library to run some functions on the FPGA fabric
- Can be loaded to FPGA dynamically, just like a software library



PYNQ Overlays:

- Provide a Python interface for controlling PL from Python running in the PS
- Created by hardware designers and wrapped with PYNQ Python API
- Each entry (*IPs* or *ports*) in the hardware becomes an ***object*** and has specific attributes and methods

  (e.g. LED IO ports can be accessed with `overlay.leds`)

An overlay usually includes:

- A *bitstream* to configure FPGA fabric
- A Vivado design *Tcl file* to determine the available IPs
- Python API that exposes the IPs as attributes (PYNQ library)

Roughly, overlay is:

bitstream + block design structure file + APIs

PYNQ Overlays

Loading an Overlay:

- The PYNQ `Overlay` class can be used to load an overlay

```
from pynq import Overlay
overlay = Overlay("base.bit")
```

- An overlay can be instantiated by specifying the bitstream file
- Overlay instantiation also downloads the bitstream to FPGA

Inspecting an Overlay:

- Once overlay is instantiated, `help()` method can be used to discover what is in an overlay

```
help(overlay)
```

- `help()` can also be used to get more information about a specific object in the overlay

```
help(overlay.leds)
```

- An example output from calling `help(base_overlay)`:

```
Help on BaseOverlay in module pynq.overlays.base.base object:

class BaseOverlay(pynq.overlay.Overlay)
 |  The Base overlay for the Pynq-Z1
 |
 |  This overlay is designed to interact with all of the on board peripherals
 |  and external interfaces of the Pynq-Z1 board. It exposes the following
 |  attributes:
 |
 |  Attributes
 |  ----------
 |  leds : AxiGPIO
 |       4-bit output GPIO for interacting with the green LEDs LD0-3
 |  buttons : AxiGPIO
 |       4-bit input GPIO for interacting with the buttons BTN0-3
 |  switches : AxiGPIO
 |       2-bit input GPIO for interacting with the switches SW0 and SW1
 |  rgbleds : [pynq.board.RGBLED]
 |       Wrapper for GPIO for LD4 and LD5 multicolour LEDs
 |  video : pynq.lib.video.HDMIWrapper
 |       HDMI input and output interfaces
 |  audio : pynq.lib.audio.Audio
 |       Headphone jack and on-board microphone
```

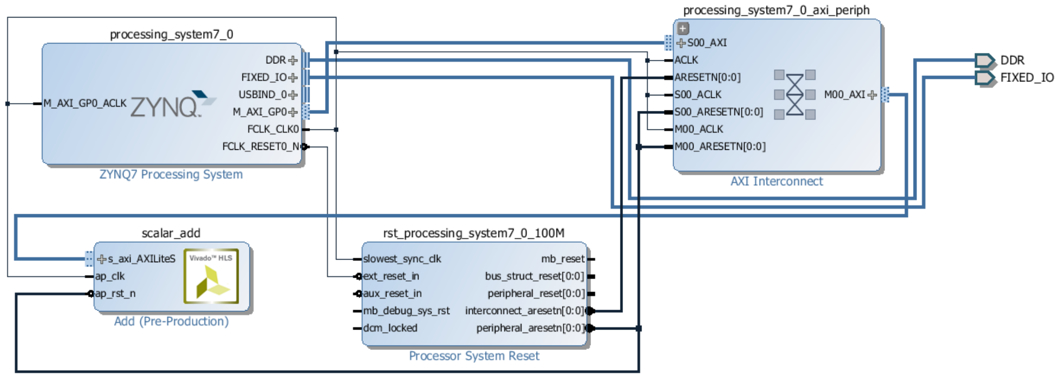
- An API can be used to control the object. For example, turning on LED0 on the board:

```
base_overlay.leds[0].toggle()
```

PYNQ Overlays

Example of Creating and Using Overlay

Assume we create an IP `scalar_add`, and create a block diagram (BD):



This block diagram consists of the IP and glue logic to connect to PS IP.

After synthesis, we get the *bitstream file* (say, hw.bit) and the *BD structure file* (tcl file or hwh file).
With these two files, we can wrap them with PYNQ Overlay class to create a PYNQ overlay (both files should be in the same directory):

```
from pynq import Overlay
overlay = Overlay('hw.bit')
```

Now we get the PYNQ overlay object "`overlay`".
(note that we get this overlay by bitstream + BD structure + PYNQ API)

Creating the overlay will automatically download the bitstream to FPGA

After creating the overlay, we can inspect the overlay. In Jupyter notebook, we can use a question mark to find out what is inside:

```
overlay?
```

All the entries in the overlay are accessible via attributes on the overlay class. For example, we can access `scalar_add` IP:

```
add_ip = overlay.scalar_add
```

We can also expose the register map associated with IP:

```
add_ip.register_map
```

It prints:

```
RegisterMap {
  a = Register(a=0),
  b = Register(b=0),
  c = Register(c=0),
  c_ctrl = Register(c_ap_vld=1, RESERVED=0)
}
```

We can also interact with the IP using the register map:

```
add_ip.register_map.a = 3
add_ip.register_map.b = 4
Print(add_ip.register_map.c)
```

Alternatively, by reading the driver source code generated by HLS we can determine the offsets we need to write the two arguments (they are all in the same memory space):

```
add_ip.write(0x10, 4)
add_ip.write(0x18, 5)
add_ip.read(0x20)
```

Reference: https://pynq.readthedocs.io/en/latest/overlay_design_methodology/overlay_tutorial.html

PYNQ Overlays – Prepare Input/Output Buffers

Allocate

- The `pynq.allocate` function is used to allocate memory that will be used by IP in the PL
- `pynq.allocate` function returns a `pynq.Buffer` object that is a sub-class of NumPy’s `ndarray` with additional properties and methods suited for use with the programmable logic
  - `device_address` is the address that should be passed to the programmable logic to access the buffer
  - `coherent` is True if the buffer is cache-coherent between the PS and PL
  - `flush` flushes a non-coherent or mirrored buffer ensuring that any changes by the PS are visible to the PL
  - `invalidate` invalidates a non-coherent or mirrored buffer ensuring any changes by the PL are visible to the PS
  - `sync_to_device` is an alias to flush
  - `sync_from_device` is an alias to invalidate

Example: Create a contiguous array of 5 32-bit unsigned integers

```
from pynq import allocate
input_buffer = allocate(shape=(5,), dtype='u4')
input_buffer[:] = range(5)
input_buffer.flush()
```

PYNQ Overlays – Running Accelerators

After creating the overlay and have data buffers ready, we can start the accelerator.

Running Accelerators

Start the kernel synchronously:

```
ol.my_kernel.call(input_buf, output_buf)
```

The call function has the same function signature as the top function in original HLS source code.

Alternatively, start the kernel in a non-blocking way:

```
handle = ol.my_kernel.start(input_buf, output_buf)
handle.wait()
```

Freeing designs (overlays):

```
ol.free()
```

Execution results can be collected by accessing output buffers.

NOTE: call, start, and wait are newly added since PYNQ 2.5.

For older version of PYNQ, to invoke the accelerator, we need to manually set the *ap_start* bit of the IP, and then wait for the *ap_start* signal.

The address of these bits can be found from the driver file generated by Vivado HLS.

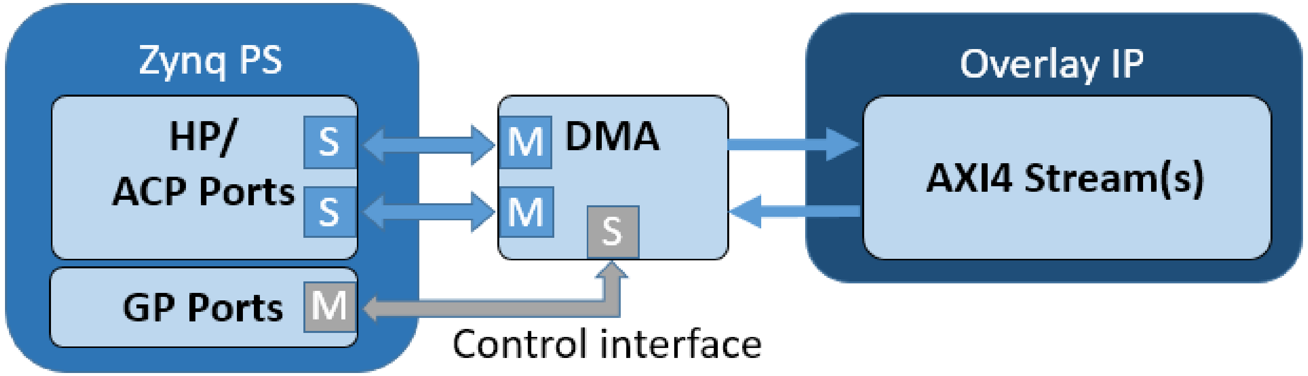
PYNQ Libraries

PYNQ provides a Python API for common peripherals and PL control

- Audio/Video
- GPIO devices (buttons, switches, LEDs, etc.)
- Headers and IO pins (e.g. Raspberry Pi header)
- PynqMicroBlaze subsystem
- Low-level PL control e.g. memory-mapped IO, memory allocation, overlay control, etc.

PYNQ Libraries - DMA

- DMA (direct memory access) can be used for high performance burst transfers between PS DRAM and the PL.
- PYNQ supports the AXI central DMA IP with the PYNQ DMA class.




```
overlay = Overlay('example.bit')
dma = overlay.axi_dma
# allocate arrays
input_buffer = allocate(shape=(5,), dtype=np.uint32)
output_buffer = allocate(shape=(5,), dtype=np.uint32)
# write some data to input array
for i in range(5):
   input_buffer[i] = i
```

```
# actual compute
...

# transfer data using DMA
dma.sendchannel.transfer(input_buffer)
dma.recvchannel.transfer(output_buffer)
dma.sendchannel.wait()
dma.recvchannel.wait()
```

PYNQ Libraries – PYNQ MicroBlaze Subsystem

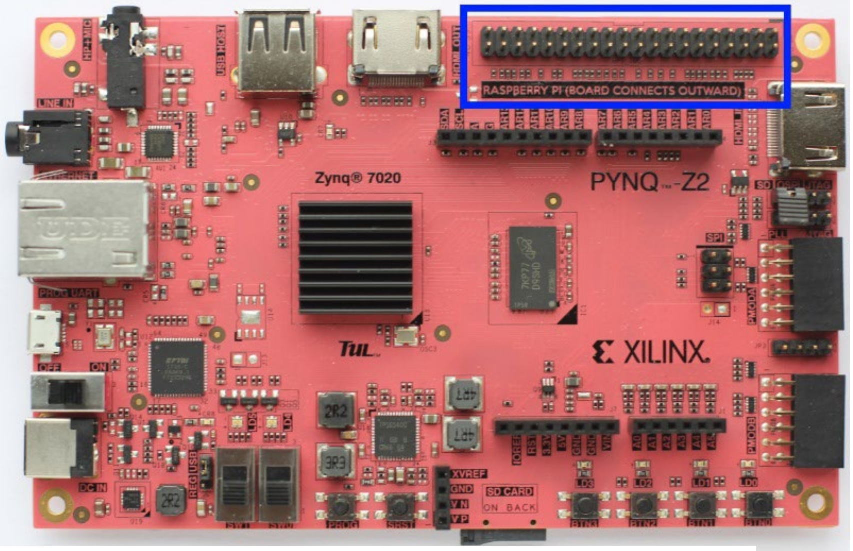
The PYNQ MicroBlaze subsystem allows loading of programs from Python, controlling executing by triggering the processor reset signal, reading and writing to shared data memory, and managing interrupts received from the subsystem.

- Each PYNQ MicroBlaze subsystem is contained within an IO Processor (IOP)
- An IOP defines a set of communication and behavioral controllers that are controlled by Python.
- Supported IOPs: Arduino, Grove, Pmod, Raspberry Pi

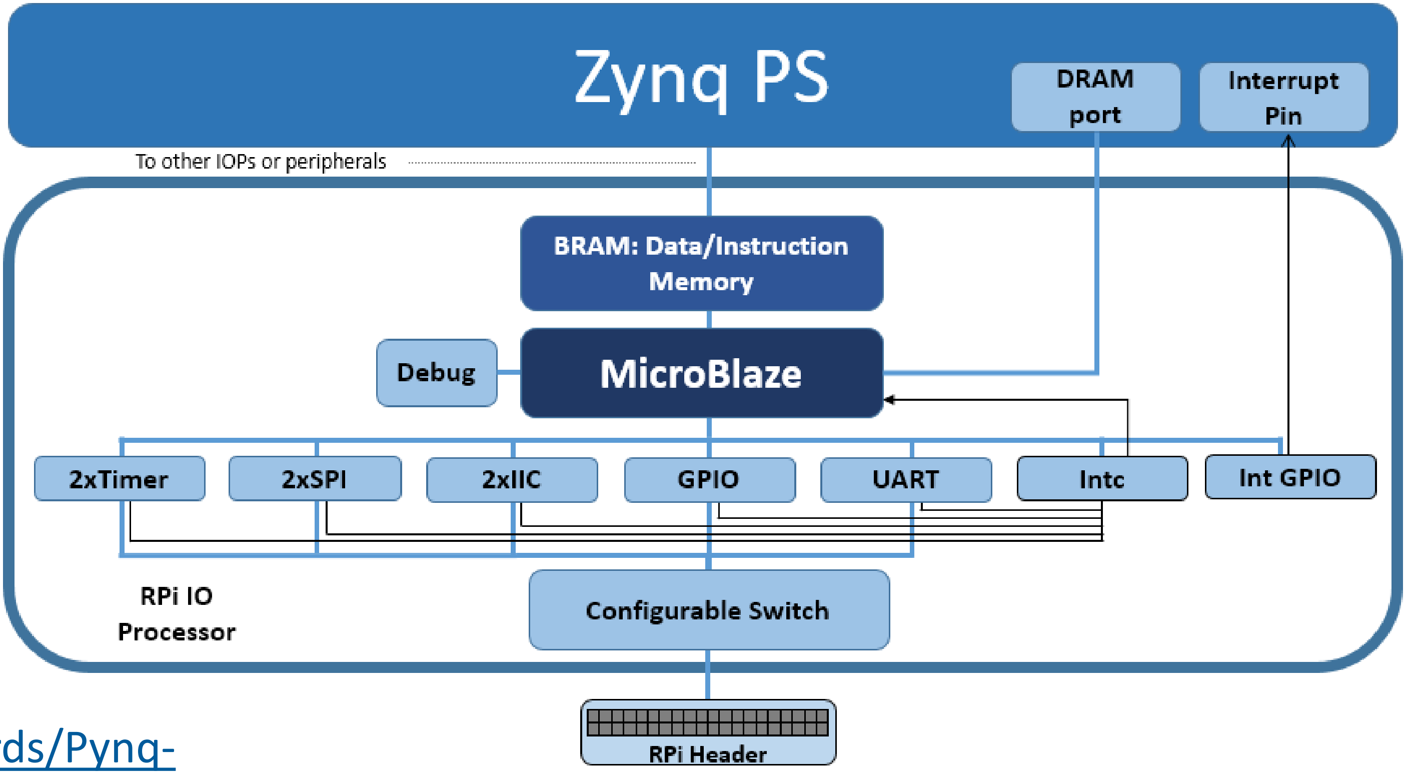
Example:

```
from pynq.overlays.base import BaseOverlay
from pynq.lib import PynqMicroblaze

base = BaseOverlay('base.bit')

mb = PynqMicroblaze(base.iop1.mb_info,  # iop1 corresponds to Pmod A connector
                    "/home/xilinx/pynq/lib/pmod/pmod_timer.bin")
mb.reset()
```

PYNQ Libraries – Raspberry Pi Header



- The rpi subpackage is a collection of drivers for controlling peripherals attached to a RPi (Raspberry Pi) interface.
- The RPi PYNQ MicroBlaze is available to control the RPi interface
- RPi PYNQ MicroBlaze has a PYNQ MicroBlaze Subsystem, a configurable switch, and the following AXI controllers:
  - 2x AXI I2C
  - 2x AXI SPI
  - 1x AXI GPIO
  - 2x AXI Timer
  - 1x AXI UART
  - AXI Interrupt controller
  - Interrupt GPIO
  - Configurable Switch



Example: Reading Values from Touch Keypad:
https://github.com/Xilinx/PYNQ/blob/master/boards/Pynq-Z2/base/notebooks/rpi/rpi_touchpad.ipynb

PYNQ on XRT Platforms

Besides Xilinx Zynq platforms, PYNQ also support XRT-based platforms such as Amazon's AWS F1 and Alveo for cloud and on-premise deployment.

(XRT: Xilinx Runtime Library)

Running Accelerators

Start the kernel synchronously:

```
ol.my_kernel.call(input_buf, output_buf)
```

The call function has the same function signature as the top function in original HLS source code.

Alternatively, start the kernel in a non-blocking way:

```
handle = ol.my_kernel.start(input_buf, output_buf)
handle.wait()
```

Freeing designs (overlays):

```
ol.free()
```

Efficient Scheduling of Multiple Kernels

start and **call** have an optional keyword parameter **waitfor** that can be used to create a dependency graph which is executed in the hardware.

```
handle = ol.vadd_1.start(input1, input2, output)
ol.vadd_1.call(input3, output, output, waitfor=(handle,))
```

Multiple FPGA Cards

PYNQ supports multiple accelerator cards in one server. It provides a **Device** class to designate which card should be used for given operations.

```
> for i in range(len(pynq.Device.devices)):
>   print("{}) {}".format(i, pynq.Device.devices[i].name))
0) xilinx_u200_xdma_201830_2
1) xilinx_u250_xdma_201830_2
2) xilinx_u250_xdma_201830_2
3) xilinx_u250_xdma_201830_2
```

EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu