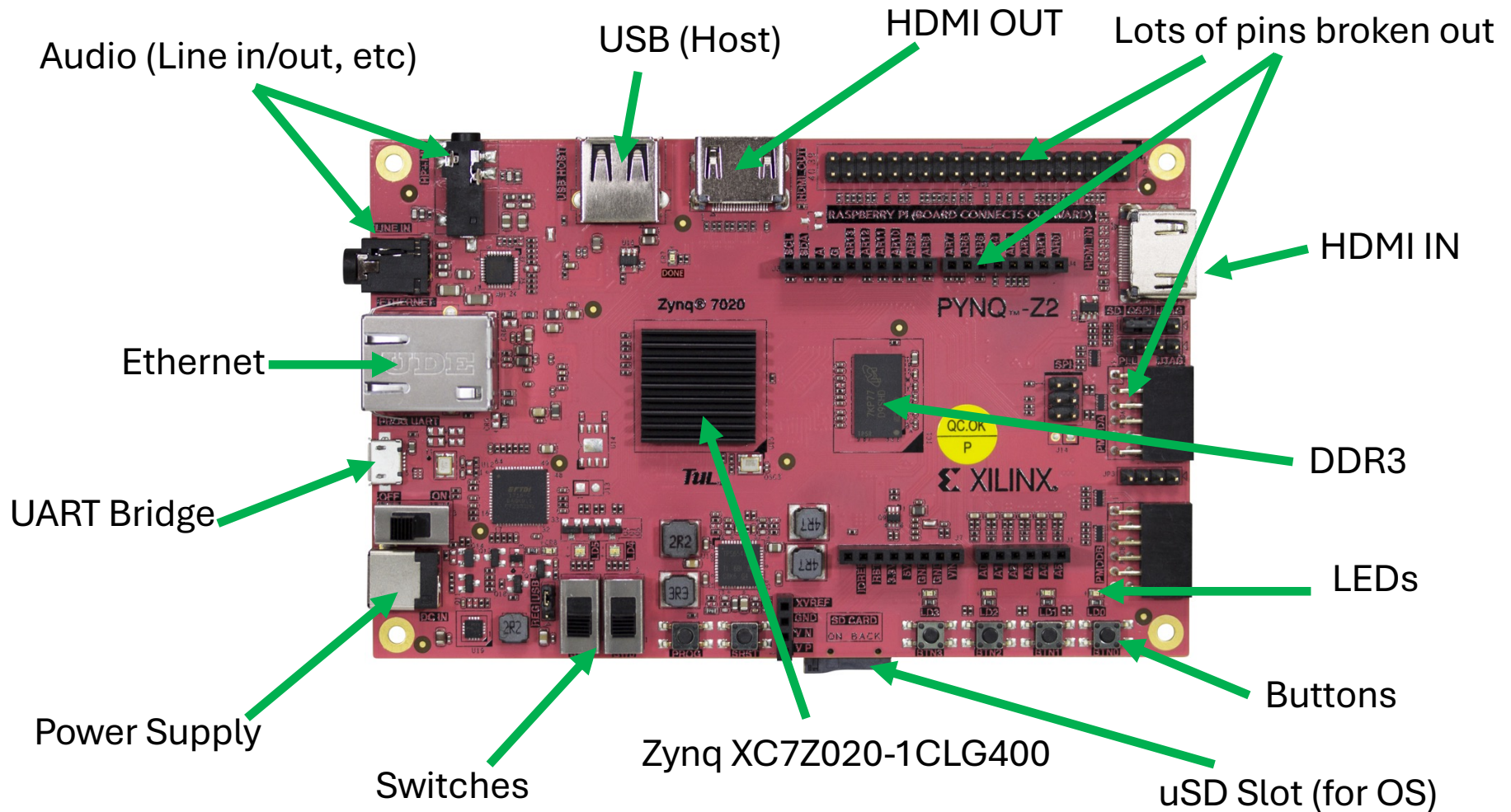


6.S965

Digital Systems Laboratory II

Lecture 4:
Zynq Architecture

Some Stuff on the Pynq Z2 Board

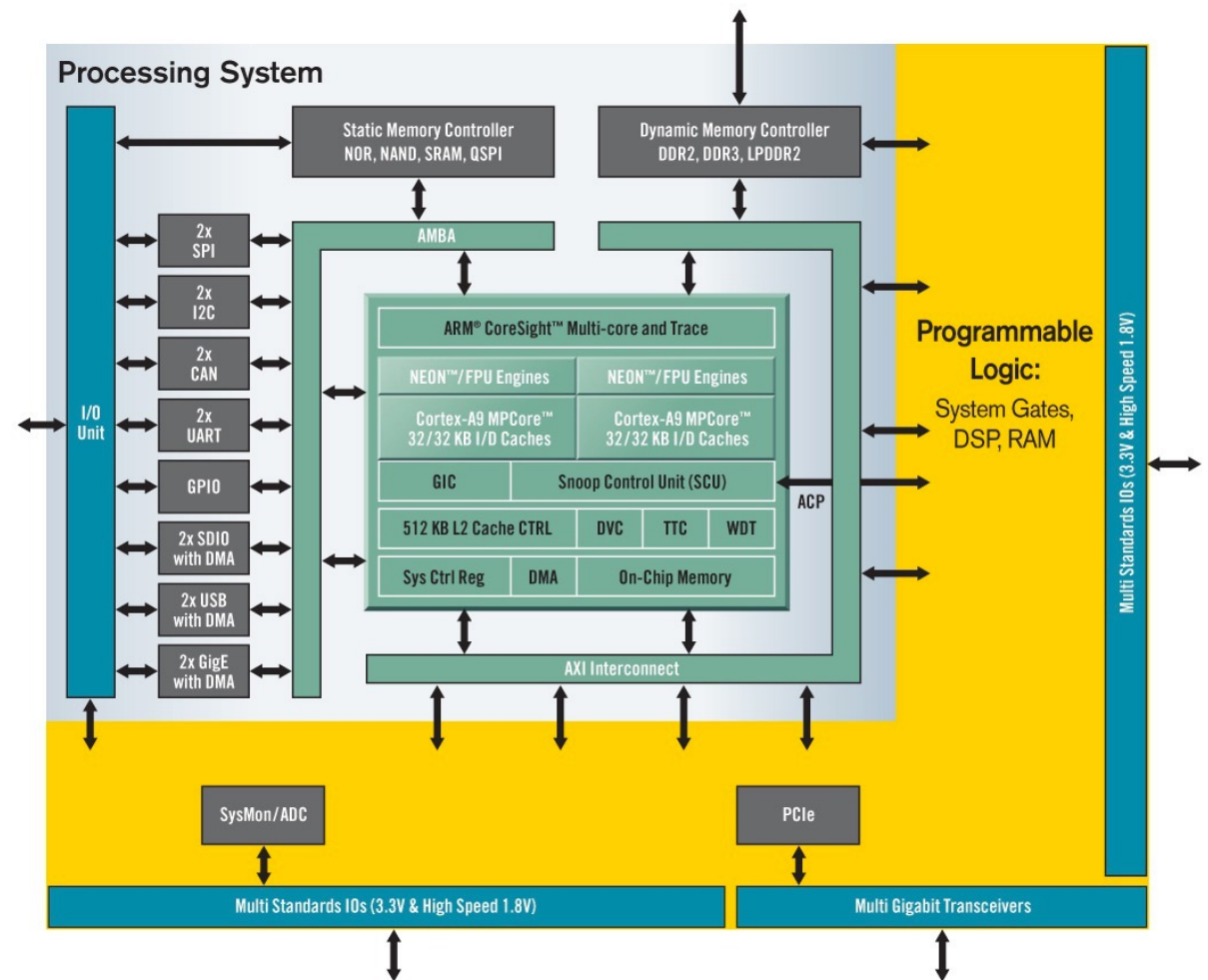


How Can You Work With it?

- The Zynq XC7Z020-1CLG400 has almost twice the amount of “classic” FPGA material as the Spartan 7 boards used in 6.205
 - 13,300 Logic Cells
 - 630 KByte of BRAM
 - 220 DSP slices
 - On-chip analog-to-digital converters on both
 - Four Clock management tiles
- Also has two ARM 9 Cores

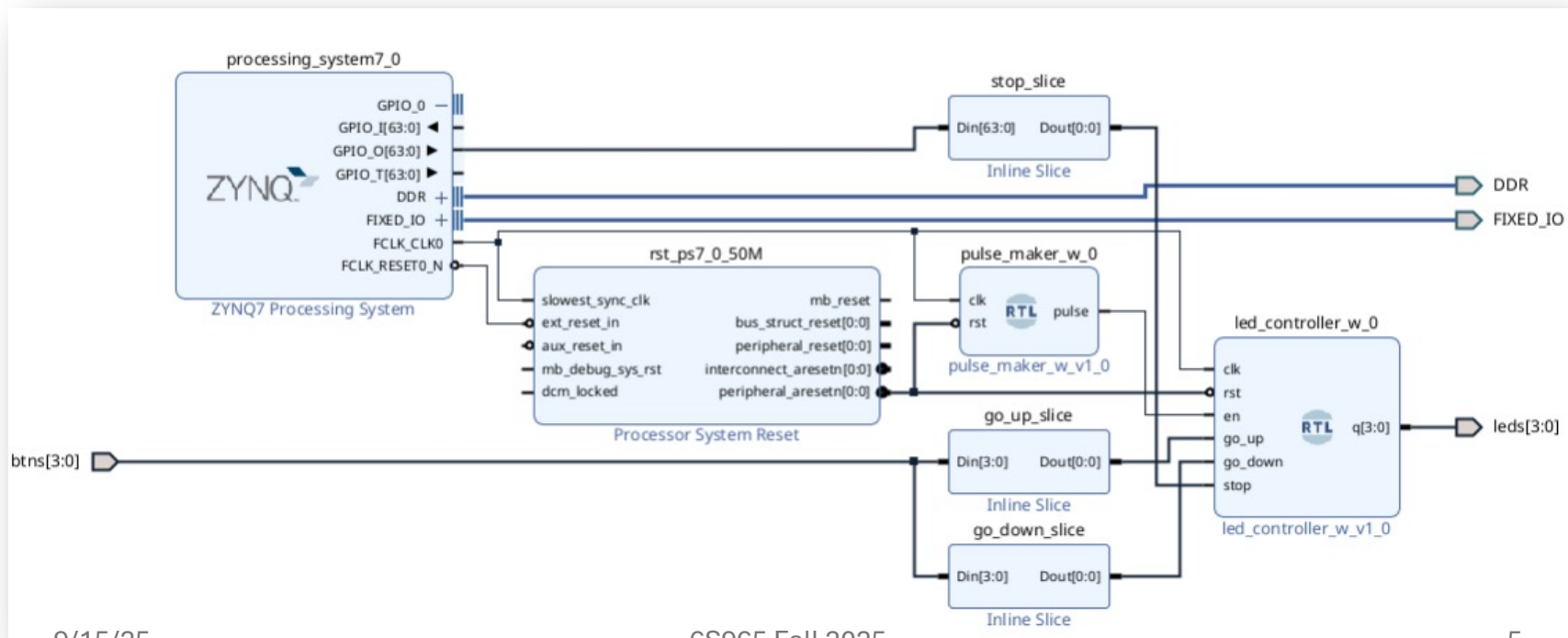
Generic Zynq Architecture

- Processing System (PS)
- Programmable Logic (PL)
- Both can be manipulated

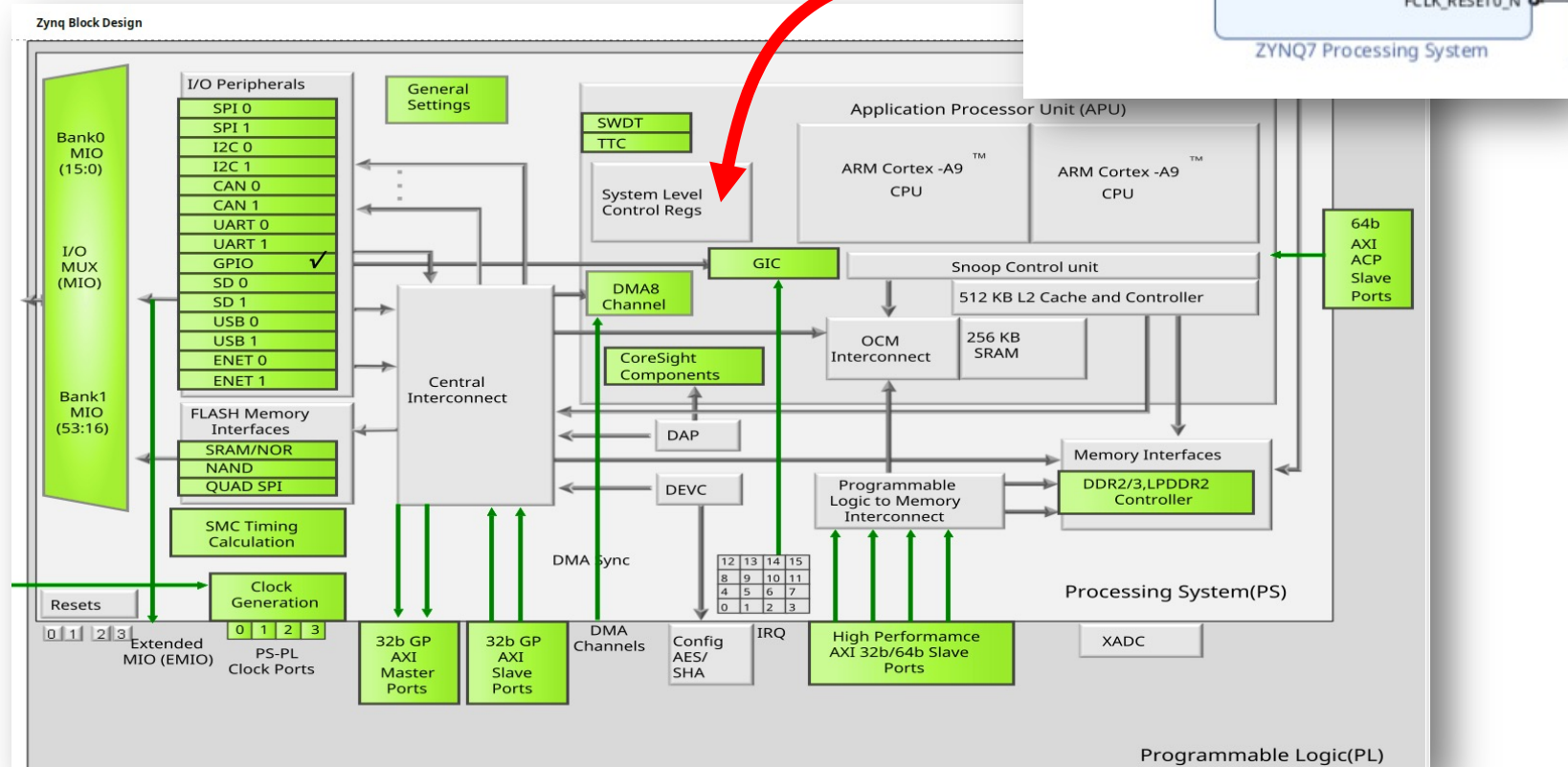


PL and PS

- When you're designing a system, there are a lot of things to control
- You can write Verilog, instantiate IP, and also configure the processing cores



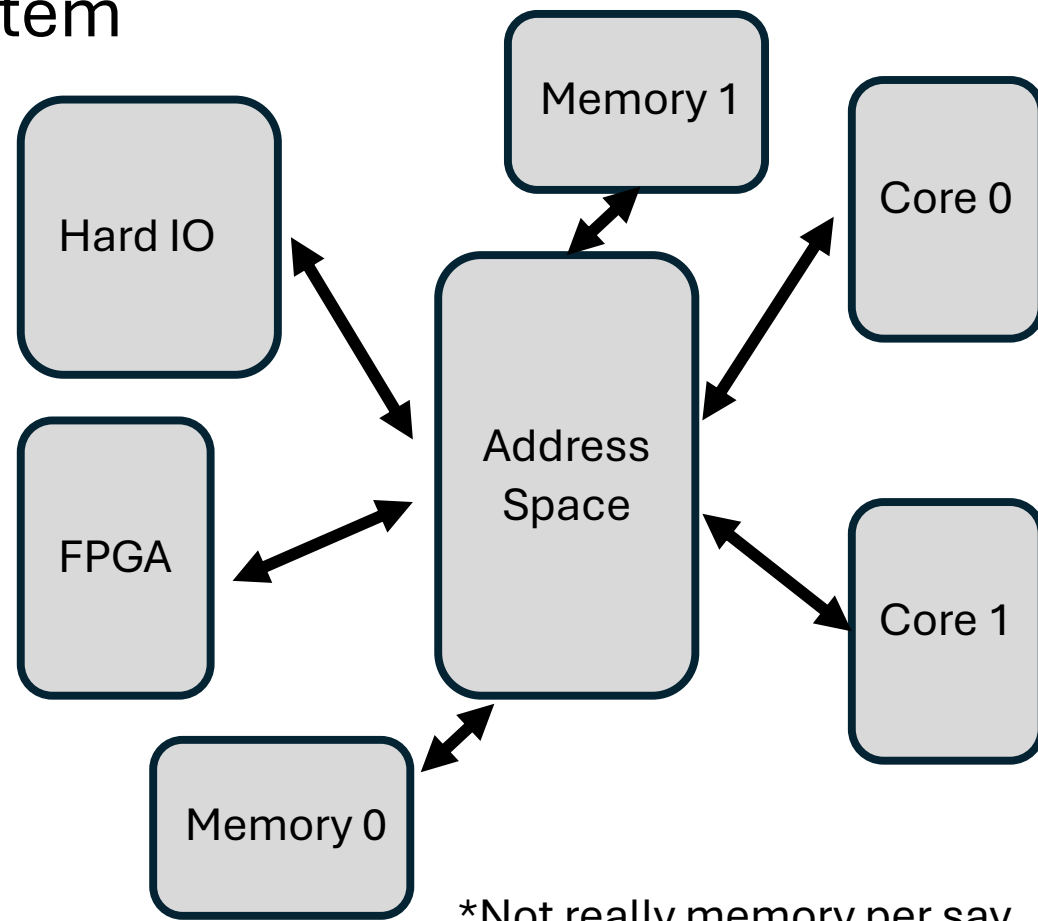
Zynq Chip



- These changes happen outside the FPGA portion

Addresses are at the Center of It All

- You have processors and you have circuits you build, and they all share information through an addressing system



*Not really memory per say...

Processing Cores are ARM

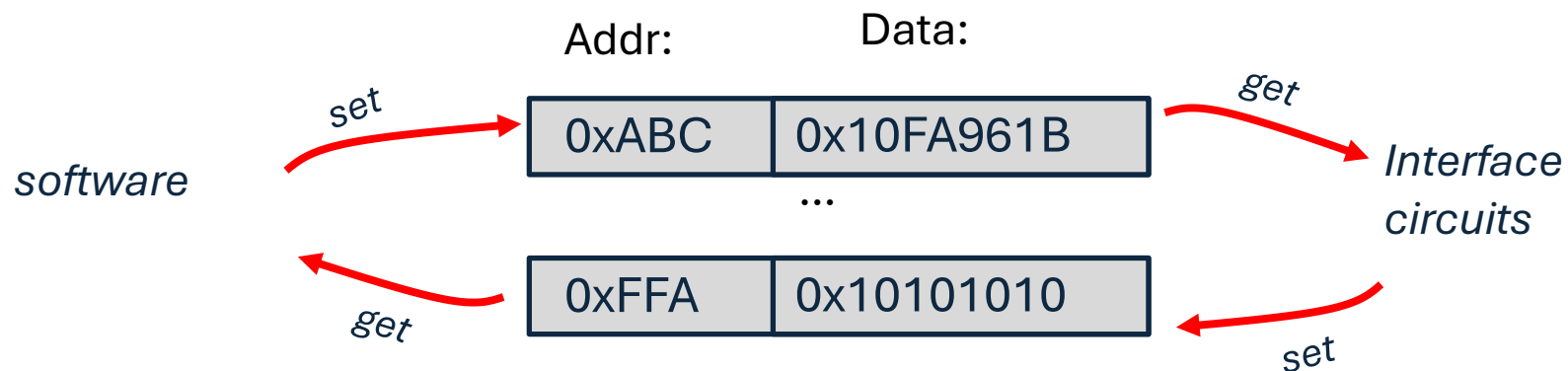
- ARM-A9 on the Pynq board (32 bit, two cores)
- ARM-A53 on the RFSoc4x2 board (64 bit, four cores), also two 32 bit ARM-R5 cores.
- ARM stands for?
- Advanced RISC Machines
- RISC stands for?
- Reduced Instruction Set Computing

Everything is Memory-Mapped

- Unlike CISC/x86 or other family processors, RISC is all about reducing the instruction set.
- In x86 memory is accessed with certain instructions and interfaces accessed with *different instructions*.
- In RISC, that's not the case...everything is accessed through LW or SW or whatever, etc...
- Everything outside the processor in RISC is seen as existing in an address space

Memory-Mapped-Input-Output (MMIO)

- In addition to having pointers take the address of variables in code that refer to memory, code will have certain addresses that are interfaces of the computer to the outside world
- Call this Memory-Mapped-Input-Output (MMIO)
- Certain addresses act like little mailboxes to set or get values from software to hardware/vice versa



MMIO Example...

```
void app_main(){
    int * temp_sensor = 0x30000004; //set pointer to a known address value
    int * heater = 0x30000008; //set pointer to a known address value
    //The two addresses above come from datasheet of processor!
    while(1){//run forever
        //check temperature...
        if (*temp_sensor < 60){ //get value...less than 60?
            *heater = 1; //set value to 1 (let it warm)
        }else{
            *heater = 0; //set value to 0 (let it cool)
        }
    }
}
```

Address:	Value:
0x30000004:	0x00000023
0x30000008:	0x00000001
...	
...	
...	
0x3fc93f58:	0x42016554
0x3fc93f5c:	0x30000004
0x3fc93f60:	0x30000008
0x3fc93f64:	0x00000000

**temp_sensor*

*Gets access to
this value*

**heater*

*Gets access to
this value*

temp_sensor

heater

L01-11

MMIO Example...

```
void app_main(){
    int * temp_sensor = 0x30000004; //set pointer to a known address value
    int * heater = 0x30000008; //set pointer to a known address value
    //The two addresses above come from datasheet of processor!
    while(1){//run forever
        //check temperature...
        if (*temp_sensor <60){ //get value...less than 60?
            *heater = 1; //set value to 1 (let it warm)
        }else{
            *heater = 0; //set value to 0 (let it cool)
        }
    }
}
```



*Thermometer circuit
will be writing values
to this memory spot*

Address:		Value:
0x30000004:		0x00000023
0x30000008:		0x00000001
...	...	
...	...	
...	...	
0x3fc93f58:		0x42016554
0x3fc93f5c:		0x30000004
0x3fc93f60:		0x30000008
0x3fc93f64:		0x00000000

*Heater circuit will be
reading values from
this memory spot to
know what to do*

L01-12

Everything Acts Like this on Zynq

- On a normal hard processor, the designers would pre-assign what IO/interfaces get assigned into each address location.
- The Zynq SOC is more of a mix, since it is reconfigurable has a lot more flexibility in that regard. In fact, one of the things the hardware handoff file contains is the memory-map addressing for a particular implementation after you've built!

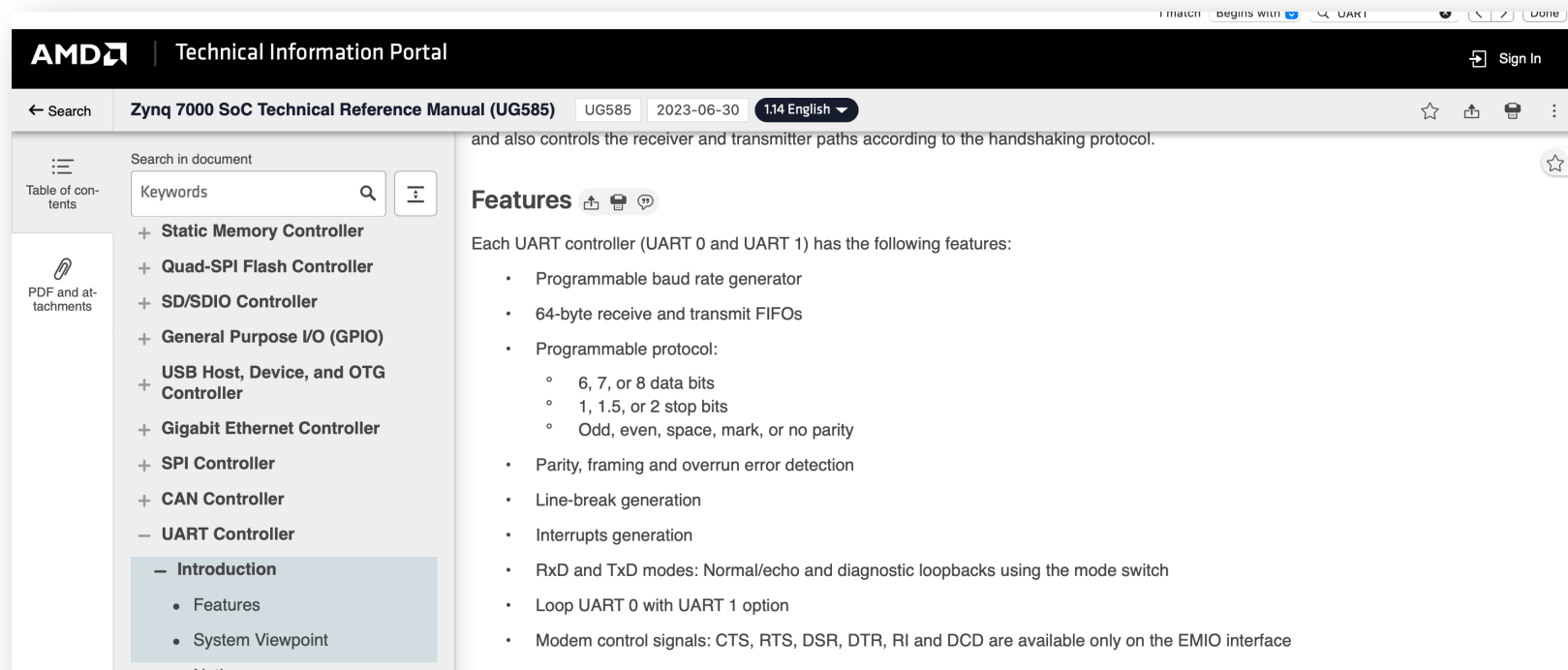
Open up the 2800 lines of the .hwh file to see...

```
1152 <PARAMETER NAME="PCW_TTC0_CLK0_PERIPHERAL_FREQMHZ" VALUE="133.333333"/>
1153 <PARAMETER NAME="
1154 <PARAMETER NAME="
1155 <PARAMETER NAME="
1156 <PARAMETER NAME="
1157 <PARAMETER NAME="
1158 <PARAMETER NAME="
1159 <PARAMETER NAME="
1160 <PARAMETER NAME="
1161 <PARAMETER NAME="
1162 <PARAMETER NAME="
1163 <PARAMETER NAME="
1164 <PARAMETER NAME="
1165 <PARAMETER NAME="
1166 <PARAMETER NAME="
1167 <PARAMETER NAME="PCW_TTC1_CLK1_PERIPHERAL_DIVISOR0" VALUE="1"/>
1168 <PARAMETER NAME="PCW_TTC1_CLK1_PERIPHERAL_FREQMHZ" VALUE="133.333333"/>
1169 <PARAMETER NAME="PCW_TTC1_CLK2_PERIPHERAL_CLKSRC" VALUE="CPU_1X"/>
1170 <PARAMETER NAME="PCW_TTC1_CLK2_PERIPHERAL_DIVISOR0" VALUE="1"/>
1171 <PARAMETER NAME="PCW_TTC1_CLK2_PERIPHERAL_FREQMHZ" VALUE="133.333333"/>
1172 <PARAMETER NAME="PCW_TTC1_HIGHADDR" VALUE="0xE0105FFF"/>
1173 <PARAMETER NAME="PCW_TTC1_PERIPHERAL_ENABLE" VALUE="0"/>
1174 <PARAMETER NAME="PCW_TTC1_TTC1_IO" VALUE="&lt;Select>"/>
1175 <PARAMETER NAME="PCW_TTC1_PERIPHERAL_FREQMHZ" VALUE="50"/>
1176 <PARAMETER NAME="PCW_UART0_BASEADDR" VALUE="0xE0000000"/>
1177 <PARAMETER NAME="PCW_UART0_BAUD_RATE" VALUE="115200"/>
1178 <PARAMETER NAME="PCW_UART0_GRP_FULL_ENABLE" VALUE="0"/>
1179 <PARAMETER NAME="PCW_UART0_GRP_FULL_IO" VALUE="&lt;Select>"/>
1180 <PARAMETER NAME="PCW_UART0_HIGHADDR" VALUE="0xE0000FFF"/>
1181 <PARAMETER NAME="PCW_UART0_PERIPHERAL_ENABLE" VALUE="1"/>
1182 <PARAMETER NAME="PCW_UART0_UART0_IO" VALUE="MIO 14 .. 15"/>
1183 <PARAMETER NAME="PCW_UART1_BASEADDR" VALUE="0xE0010000"/>
1184 <PARAMETER NAME="PCW_UART1_BAUD_RATE" VALUE="115200"/>
1185 <PARAMETER NAME="PCW_UART1_GRP_FULL_ENABLE" VALUE="0"/>
1186 <PARAMETER NAME="PCW_UART1_GRP_FULL_IO" VALUE="&lt;Select>"/>
1187 <PARAMETER NAME="PCW_UART1_HIGHADDR" VALUE="0xE0010FFF"/>
1188 <PARAMETER NAME="PCW_UART1_PERIPHERAL_ENABLE" VALUE="0"/>
1189 <PARAMETER NAME="PCW_UART1_UART1_IO" VALUE="&lt;Select>"/>
```

- All the interactions with UART Bus 0 happen from 0xE0000000 to 0xE0000FFF

Then you'd proceed to manual

- Look up the address space of UART busses



Keep Reading...

- See an interesting diagram in the docs explaining how it works...and click on it so you can see the unblurred version....

AMD | Technical Information Portal

Search Zynq 7000 SoC Technical Reference Manual (UG585) UG585 2023-06-30 1.14 English

Search in document
Keywords

- Features
- System Viewpoint
- + Notices
- + Functional Description
- + Programming Guide
- + System Functions
- + I/O Interface
- + I2C Controller
- + Programmable Logic Description
- + Programmable Logic Design Guide
- + Programmable Logic Test and Debug
- + Power Management

PDF and attachments

System Viewpoint

The system viewpoint diagram for the UART controllers is shown in [This Figure](#).

Figure: UART System Viewpoint

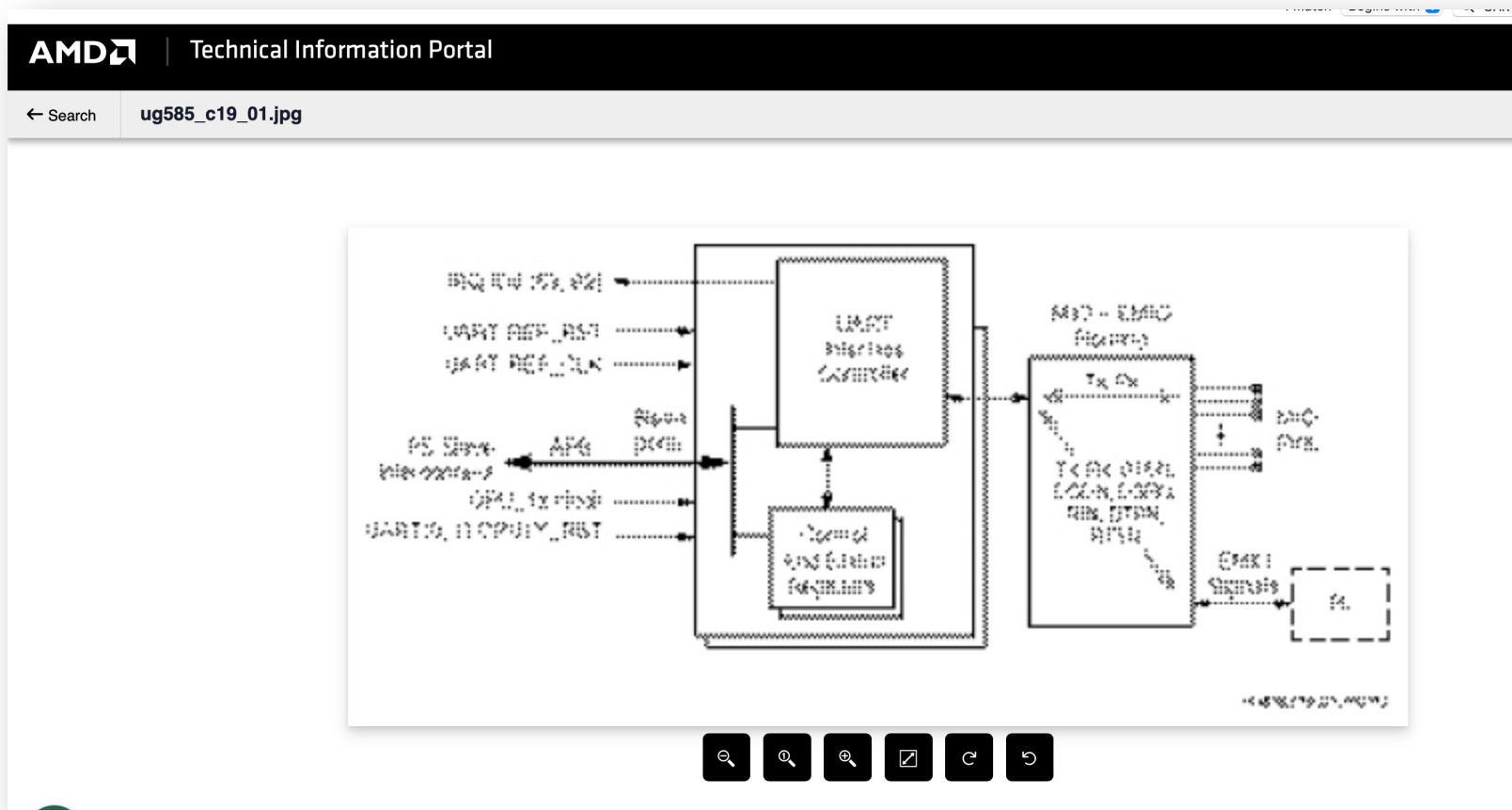
The diagram illustrates the UART System Viewpoint. It shows a central block labeled 'UART Controller' which is connected to various external components. On the left, there are inputs for 'UART RX', 'UART TX', and 'UART CTS'. These are connected to the 'UART Controller' block. The 'UART Controller' block is also connected to a 'General Purpose I/O' block. On the right, there are outputs for 'UART RX', 'UART TX', and 'UART CTS'. These are connected to the 'UART Controller' block. The 'UART Controller' block is also connected to a 'General Purpose I/O' block. The diagram shows the internal structure of the UART controller, including the 'UART Controller' block and the 'General Purpose I/O' block. The diagram is a block diagram showing the internal components and connections of the UART controller.

The slcr register set (refer to section [SLCR Registers](#)) includes control bits for the UART clocks, resets and MIO-EMIO signal mapping. Software accesses the UART controller registers using the APB 32-bit slave interface attached to the PS AXI interconnect. The IRQ from each controller is connected to the PS interrupt controller and routed to the PL.

9/15/25 © 2025 Advanced Micro Devices, Inc. 6S965 Fall 2025 16

Extract Meaning from this...

- Thank you Xilinx



<https://docs.amd.com/viewer/attachment/mxcNFn1EFZjLI1eShoEn5w/oeiYFdxDVPZU5ktSckeTug-mxcNFn1EFZjLI1eShoEn5w>

Jokes* Aside...

- Further Down the page are details about addresses to read/write to to configure the UART0 and then where in that address space, the In and out FIFO will live

Configure Controller Functions

Example: Configure Controller Functions

This example configures the character frame, the baud rate, the FIFO trigger levels, the Rx timeout mechanism, and enables the controller. All of these steps are necessary after a reset, but not necessary between enabling and disabling the controller.

1. Configure UART character frame . Write 0x0000_0020 into the uart.mode_reg0:
 - a. Disables clock pre-divisor, UART_REF_CLK/8: [CLKS] = 0
 - b. Selects 8-bit character length: [CHRL] = 00
 - c. Selects no parity: [PAR] = 100
 - d. Selects 1 stop bit: [NBSTOP] = 00
 - e. Selects normal channel mode (Mode Switch): [CHMODE] = 00
2. Configure the Baud Rate . Write to three registers: uart.Control_reg0, uart.Baud_rate_gen_reg0, and uart.Baud_rate_divider_reg0. Examples for the calculated CD and BDIV values are shown in table [Table: UART Parameter Value Examples](#). The baud rate generator is described in section [Baud Rate Generator](#)

<https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/Configure-Controller-Functions>

**is it really a joke?*

And for “Custom” modules...

- Same thing... Here is the disp_interface I wrote for lab 2:

```
<MODULE CURRENTVERSION= 2 FULLNAME= /disp_interface_0 HWDVERSION= 1.0 INSTANCE= disp_interface_0 IPTYPE= PERIPHERAL  
ISS="PERIPHERAL" MODTYPE="disp_interface" ULNV="user.org:user:disp_interface:1.0">  
  <DOCUMENTS/>  
  <ADDRESSBLOCKS>  
    <ADDRESSBLOCK ACCESS="" INTERFACE="S00_AXI" NAME="S00_AXI_reg" RANGE="4096" USAGE="register"/>  
  </ADDRESSBLOCKS>  
  <PARAMETERS>  
    <PARAMETER NAME="C_S00_AXI_ADDR_WIDTH" VALUE="5"/>  
    <PARAMETER NAME="C_S00_AXI_DATA_WIDTH" VALUE="32"/>  
    <PARAMETER NAME="Component_Name" VALUE="design_1_disp_interface_0_0"/>  
    <PARAMETER NAME="EDK_IPTYPE" VALUE="PERIPHERAL"/>  
    <PARAMETER NAME="C_S00_AXI_BASEADDR" VALUE="0x43C00000"/>  
    <PARAMETER NAME="C_S00_AXI_HIGHADDR" VALUE="0x43C0FFFF"/>  
  </PARAMETERS>  
  <PORTS>  
    <PORT DIR="I" LEFT="3" NAME="btn" RIGHT="0" SIGIS="undef" SIGNAME="External_Ports_btns">  
      <CONNECTIONS>  
        <CONNECTION INSTANCE="External_Ports" PORT="btns"/>  
      </CONNECTIONS>
```

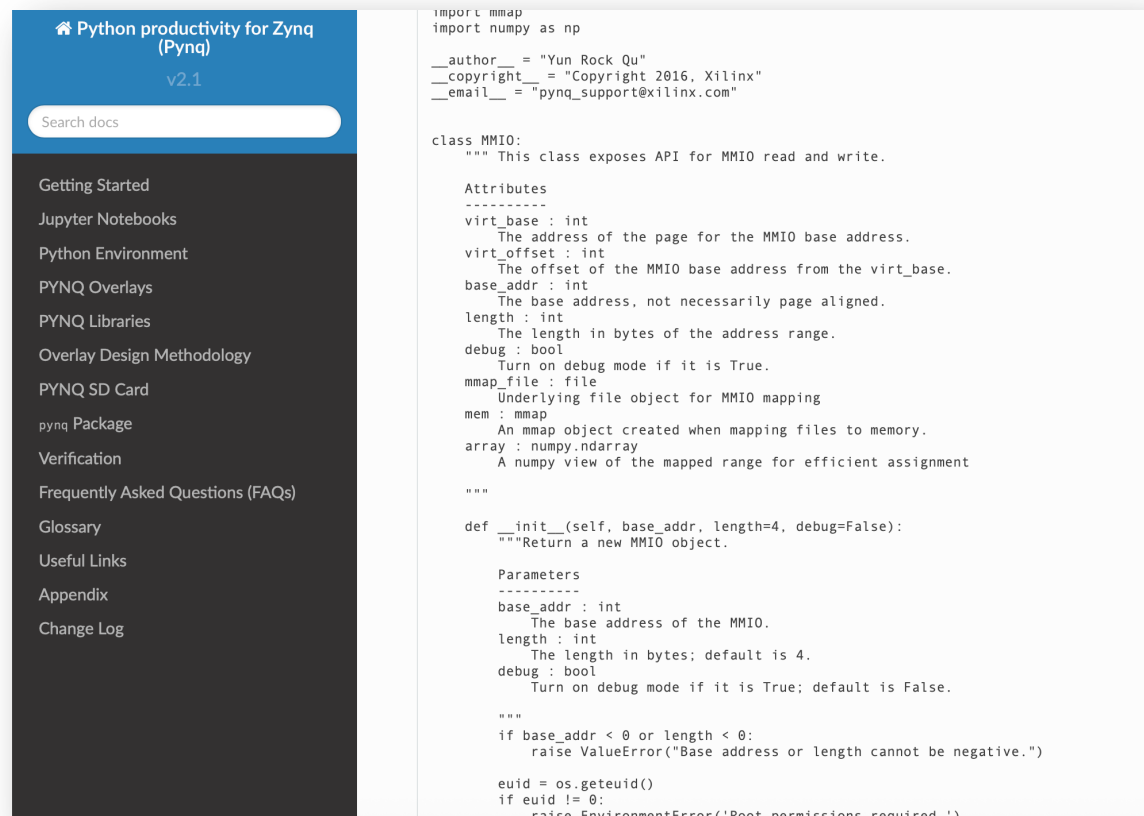
Then From the software side...

```
j5 = ol.disp_interface_0 #find the AXI MMIO module which we can talk to (name of IP)
#Now it is time to interface with the j5 IP:
# registers are four bytes in size, but address space is byte addressable! Keep in mind!
j5.write(0x08,0) #write 0 to address location 0x08 (command type)..
j5.write(0x0C,5) #write "5" to address location 0x0C (should show up on green LEDs due to slice
j5.write(0x10,2) #write 2 to address location 0x10
d = j5.read(0x04) # should read the value of all four push buttons (for test)
print(d) #print output (hopefully buttons)
d = j5.read(0x00) # read deadbeef hopefully (hard-coded in your mmio)
print(hex(d))
```

- Read/write to addresses that refer to the module you made

Look at Source of Pynq (or C that it uses underneath)

- See how it handles it with the underlying calls.



The image shows a side-by-side comparison. On the left is a screenshot of the Pynq documentation website. The header is blue with the text 'Python productivity for Zynq (Pynq) v2.1'. Below the header is a search bar and a list of navigation links: 'Getting Started', 'Jupyter Notebooks', 'Python Environment', 'PYNQ Overlays', 'PYNQ Libraries', 'Overlay Design Methodology', 'PYNQ SD Card', 'pynq Package', 'Verification', 'Frequently Asked Questions (FAQs)', 'Glossary', 'Useful Links', 'Appendix', and 'Change Log'. On the right is a screenshot of the source code for the Pynq library, specifically the 'MMIO' class. The code is in Python and shows imports for 'mmap' and 'numpy', version information, and the definition of the 'MMIO' class with its attributes and initialization method.

```
import mmap
import numpy as np

__author__ = "Yun Rock Qu"
__copyright__ = "Copyright 2016, Xilinx"
__email__ = "pynq_support@xilinx.com"

class MMIO:
    """ This class exposes API for MMIO read and write.

    Attributes
    -----
    virt_base : int
        The address of the page for the MMIO base address.
    virt_offset : int
        The offset of the MMIO base address from the virt_base.
    base_addr : int
        The base address, not necessarily page aligned.
    length : int
        The length in bytes of the address range.
    debug : bool
        Turn on debug mode if it is True.
    mmap_file : file
        Underlying file object for MMIO mapping
    mem : mmap
        An mmap object created when mapping files to memory.
    array : numpy.ndarray
        A numpy view of the mapped range for efficient assignment
    """

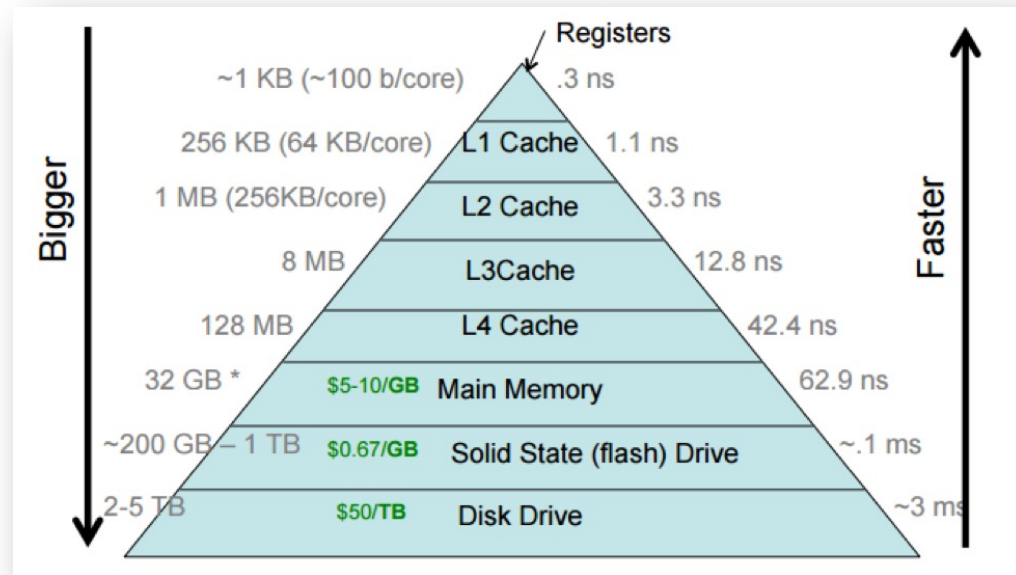
    def __init__(self, base_addr, length=4, debug=False):
        """Return a new MMIO object.

        Parameters
        -----
        base_addr : int
            The base address of the MMIO.
        length : int
            The length in bytes; default is 4.
        debug : bool
            Turn on debug mode if it is True; default is False.
        """
        if base_addr < 0 or length < 0:
            raise ValueError("Base address or length cannot be negative.")

        euid = os.geteuid()
        if euid != 0:
            raise EnvironmentError('Root permissions required.')
```

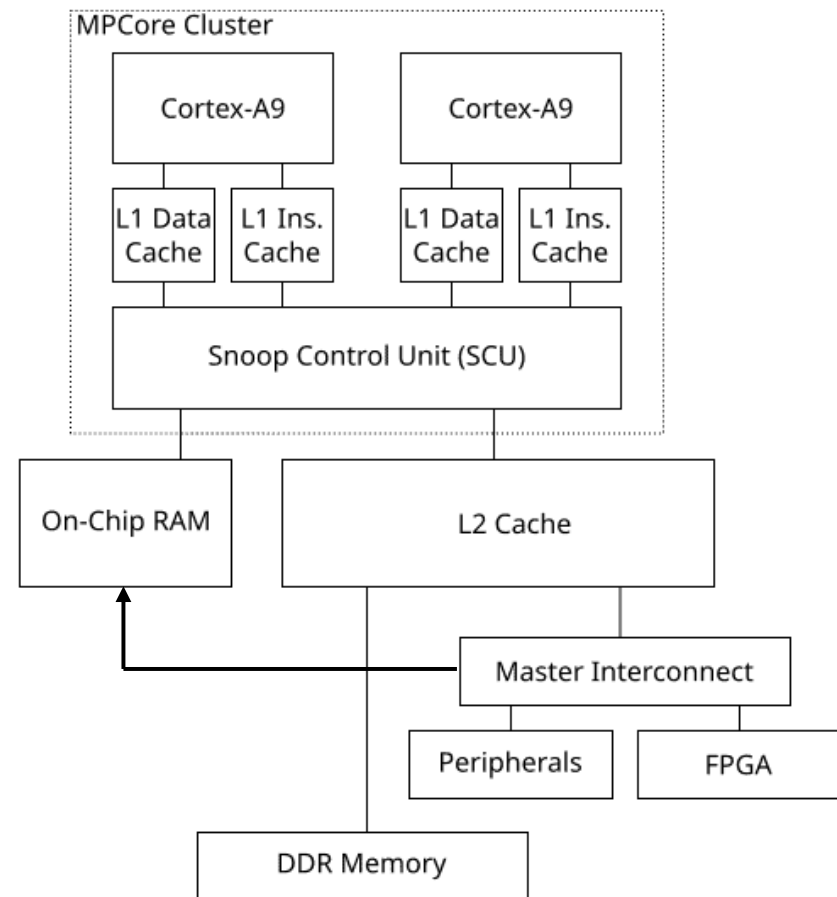
The Address Space is a Delicate Illusion

- Almost all modern compute use a hierarchy of memory layers to facilitate quick, effective access to data
- Our The Zynq SOC is no different.
- And this goes for even MMIO stuff
- And it is complicated



Memory Layout in Zynq Series

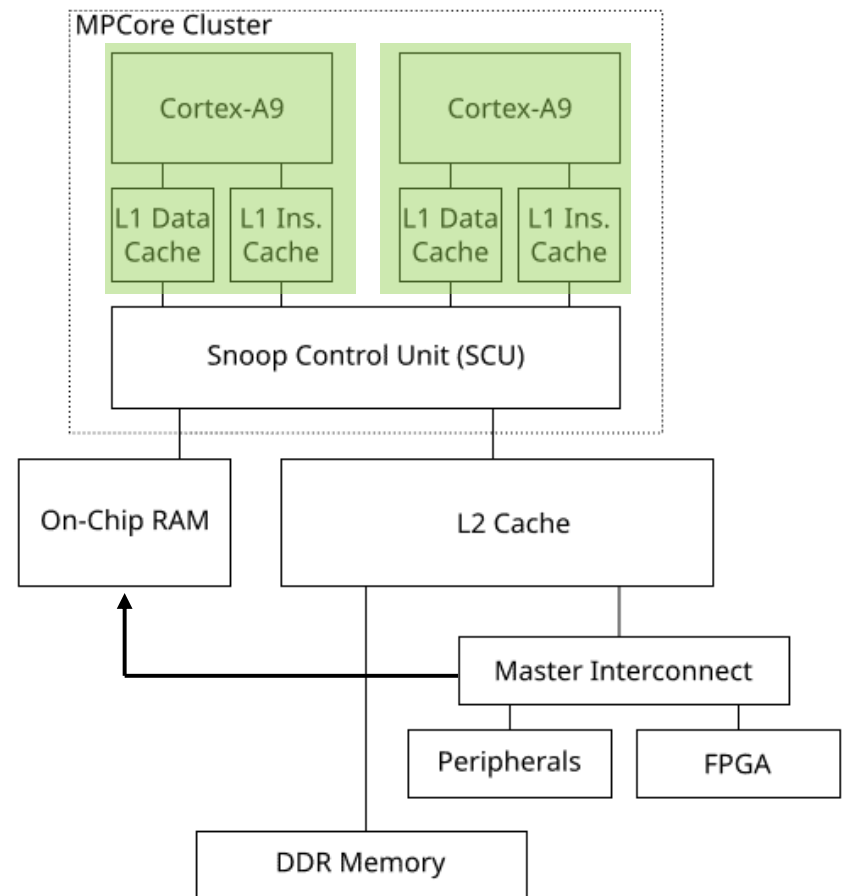
- Cores each have their own L1 Cache
- Below that everything is shared (L2 Cache, On-Chip RAM, everything else appropriately)



<https://www.jblopen.com/zynq-benchmarks/>

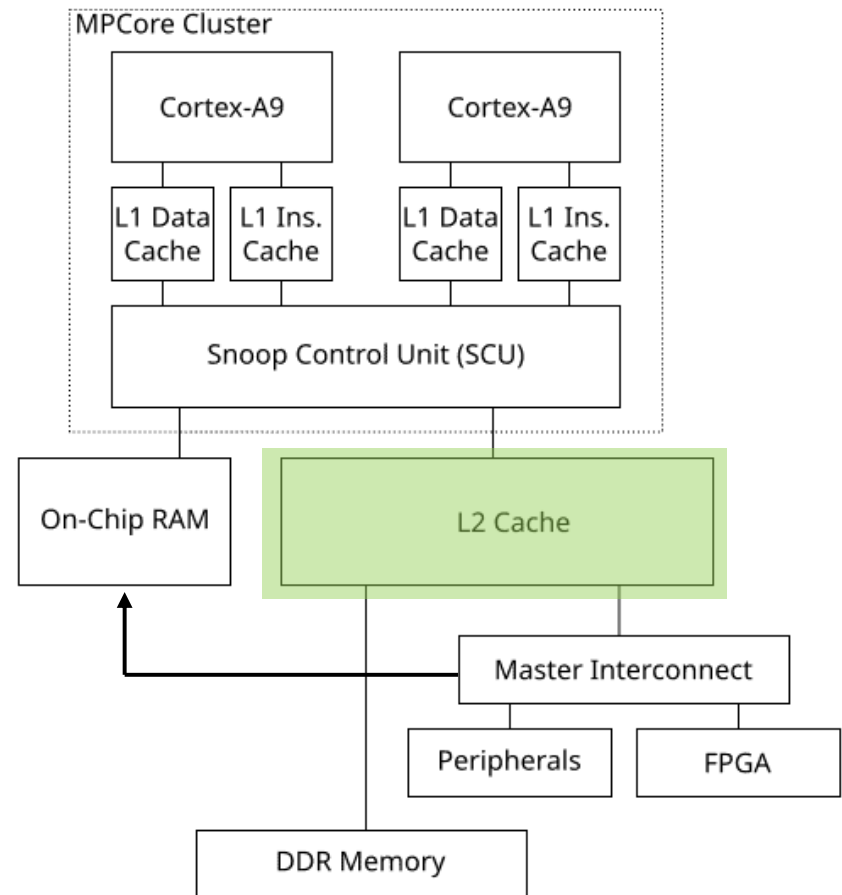
L1 Data and Instruction Caches

- Each core has a pair of L1 caches
- Works like an L1 cache normally
does...effectively a temporary clone of relevant memory regions for the cores to have access to.



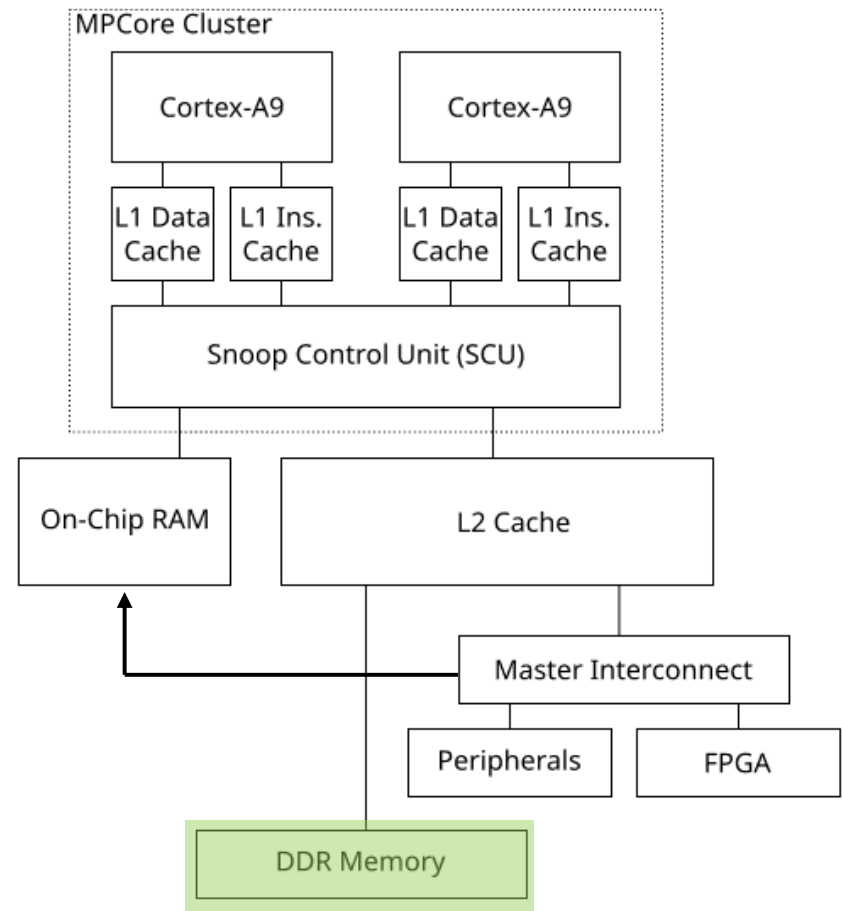
L2 Cache

- There is a single L2 shared cache
- L1 draws from it



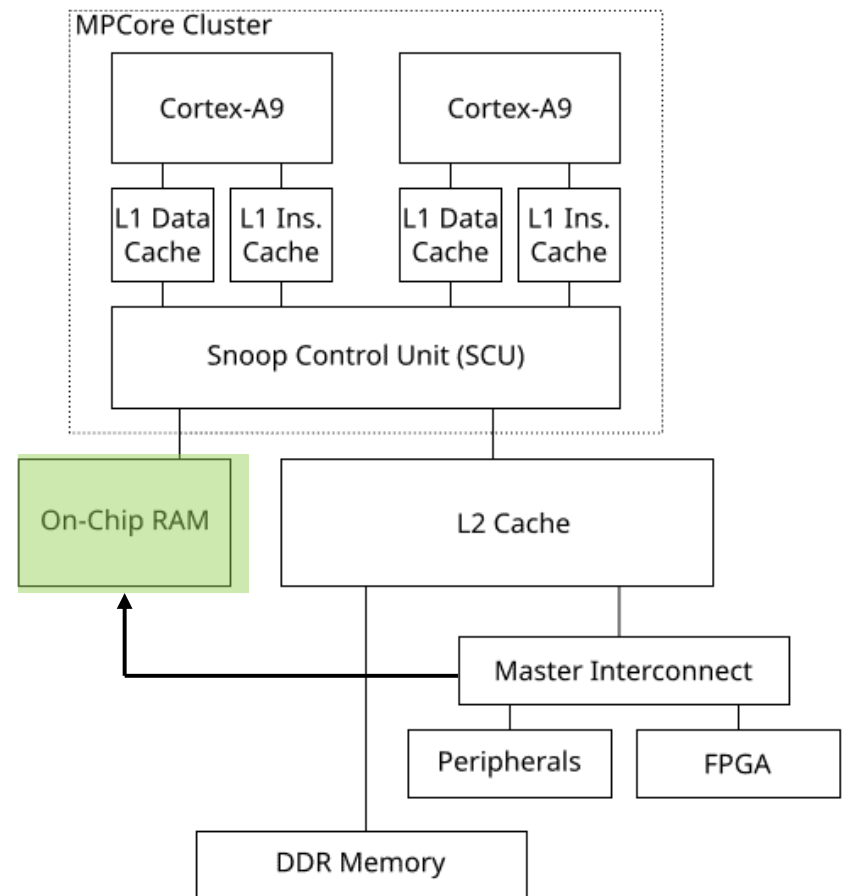
DDR Memory

- Huge amount (512 MB or more) of off-chip DDR2 or DDR3
- “Global” repository of almost all the memory space (not necessarily entire address space)
- More on that later



On-Chip RAM/Memory (OCR/OCM)

- There is ~256 KB of On-Chip RAM
- Separate piece of memory on chip with fixed address space:
 - 192kB at 0x0000_0000
 - 64kB at 0xFFFC_0000
- As fast as a cache but not used as a cache!



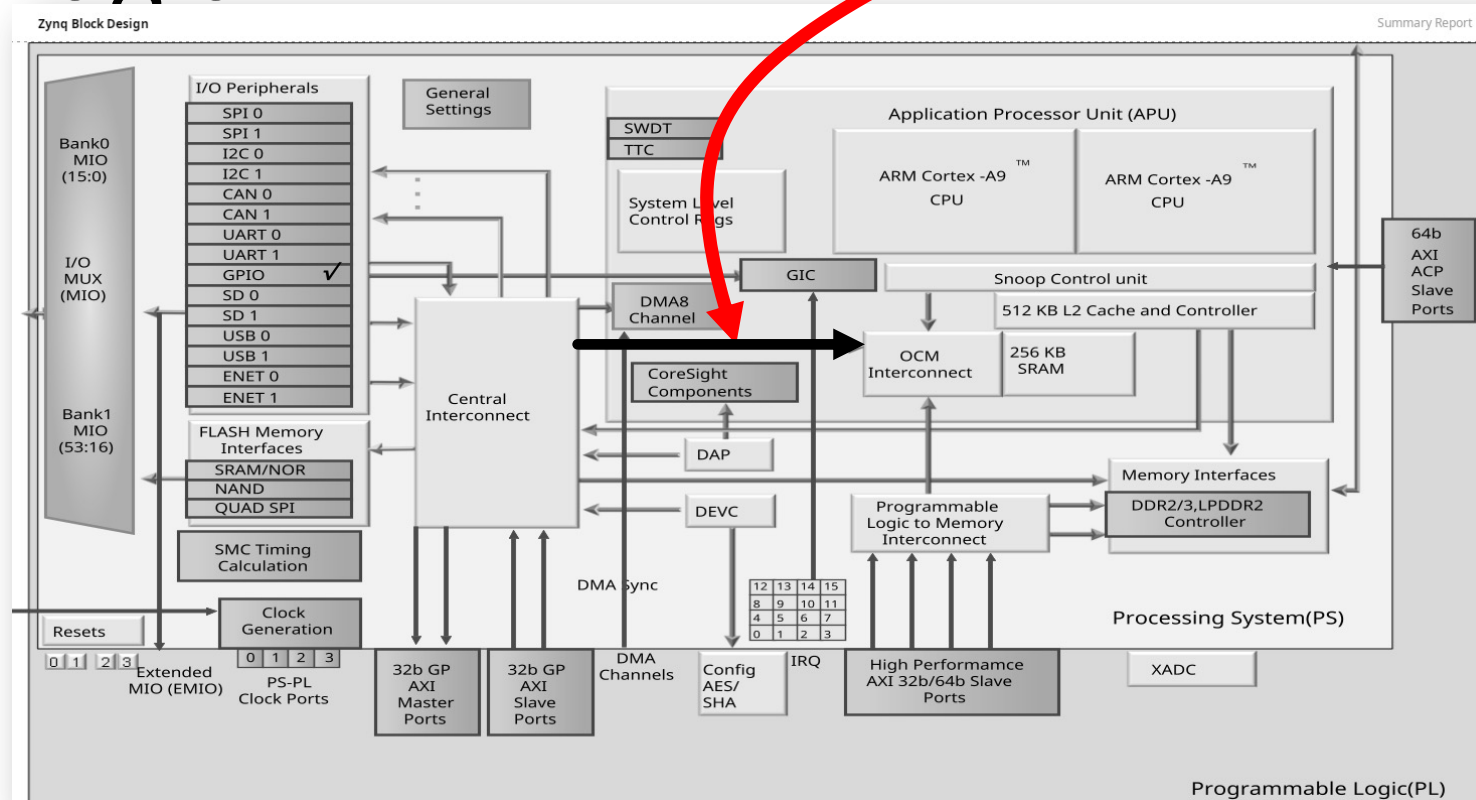
<https://www.jblopen.com/zynq-benchmarks/>

OCM vs. Cache

- Cache represents a moving target of regions of the ultimate address space (stuff from DRAM, stuff from IO, etc...)
- The OCM is a fixed global address space that you can directly address (both from with the PS and from the PL)

Zynq Block Diagram

Direct connection
from Central
Interconnect to OCM

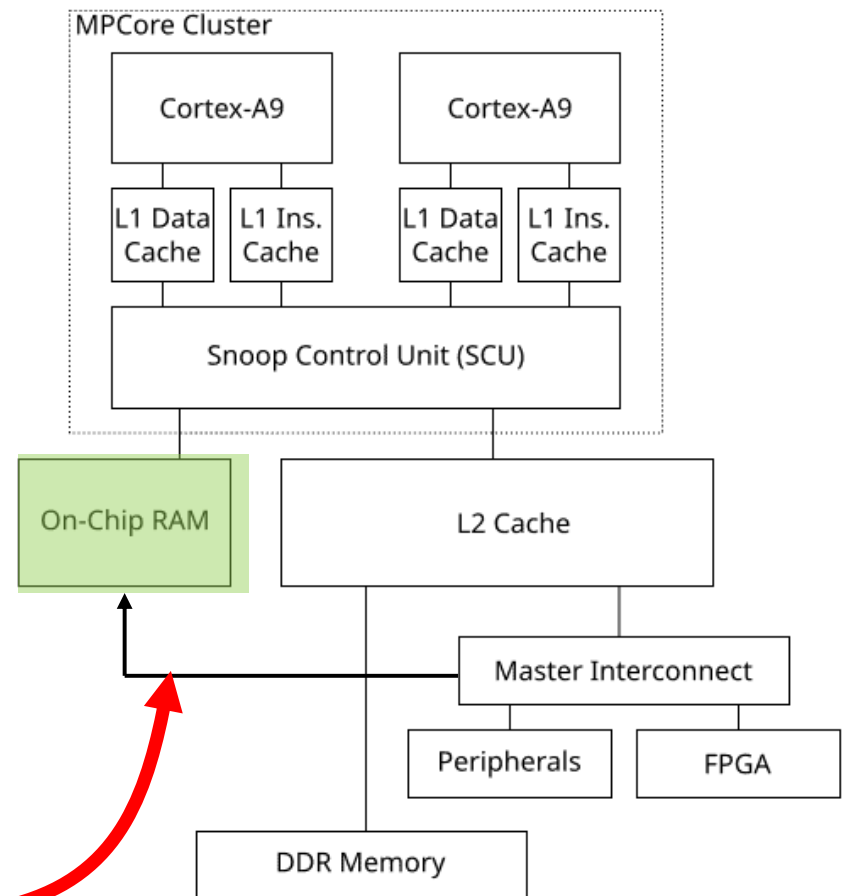


- These changes happen outside the FPGA portion

On-Chip RAM/Memory (OCR/OCM)

- Why might OCM be useful?
- Sensitive, low-latency information can be conveyed between the FPGA and processor without cacheing, etc...

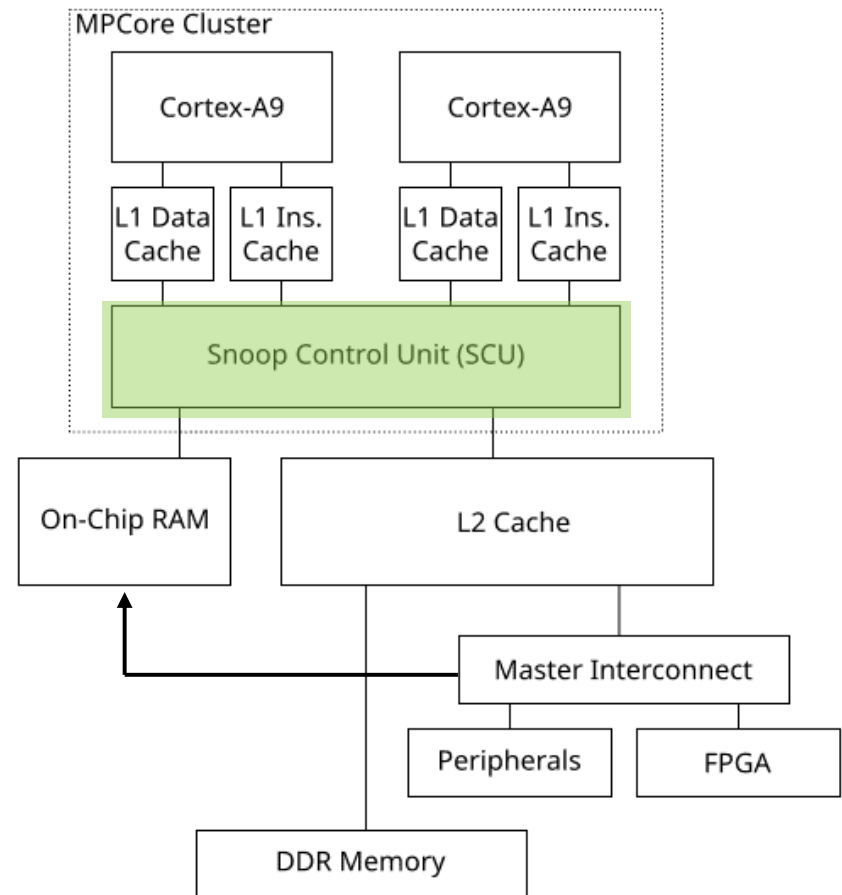
Direct connection
from Central
Interconnect to OCM



<https://www.jblopen.com/zynq-benchmarks/>

Snoopy Cache

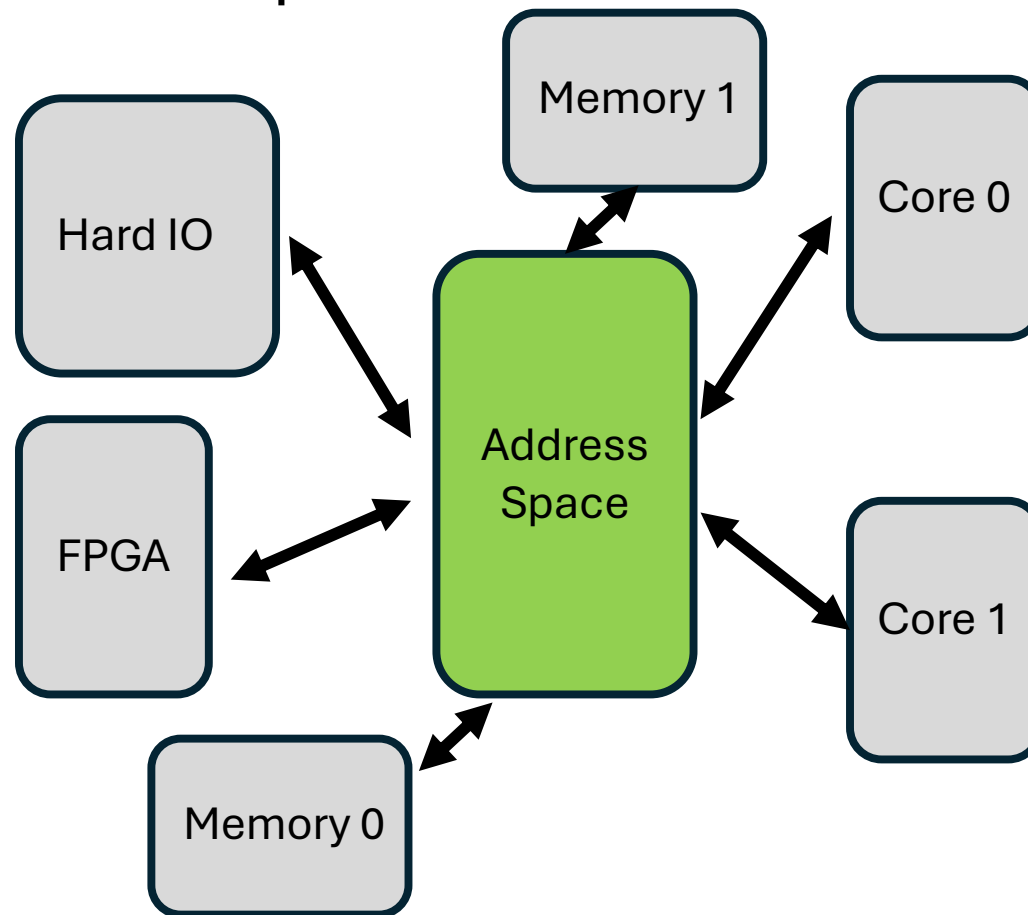
- The Snoop Control Unit is in charge of keeping the multiple L1 caches and the greater L2, OC, DDR, etc... Synchronized
- Complicated piece of hardware
- Further Reading:
 - https://en.wikipedia.org/wiki/Bus_snooping



<https://www.jblopen.com/zynq-benchmarks/>

Snoop Control Unit

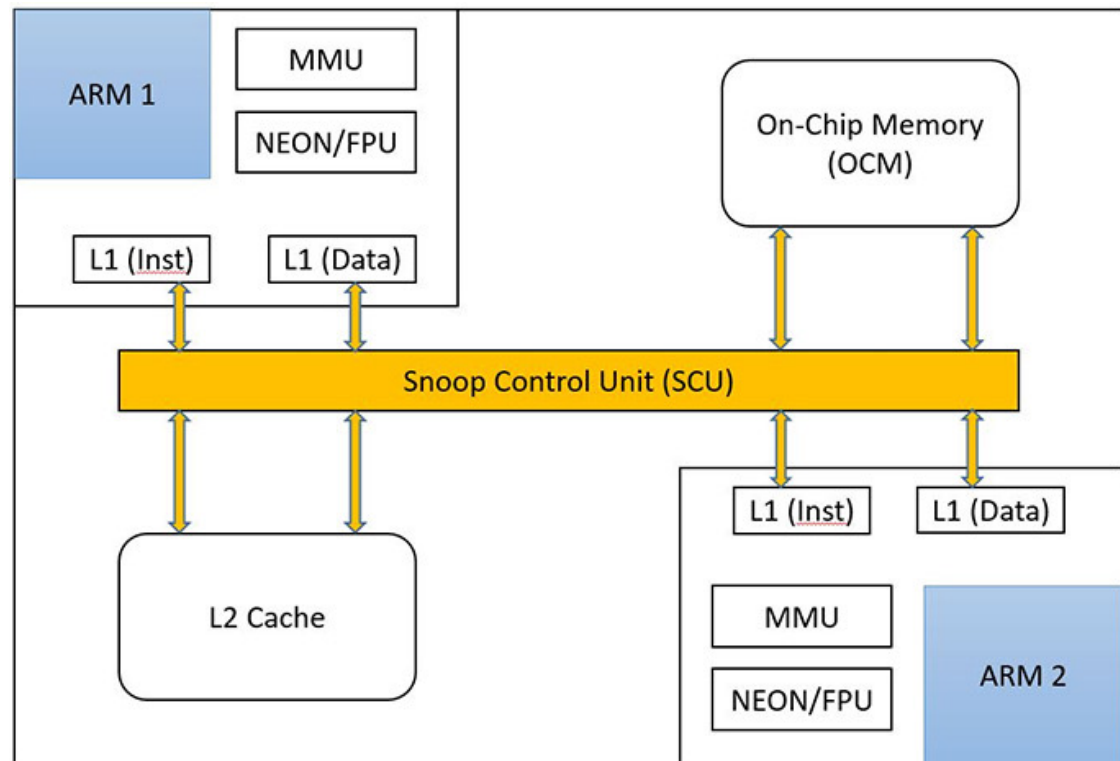
- Critical in maintaining the illusion of unified memory/address space



*Not really memory per say...

SCU is at center of it

- Anything going to processor has to go through the SCU



<https://www.aldec.com/en/company/blog/144--introduction-to-zynq-architecture>

Data Between PS and PL

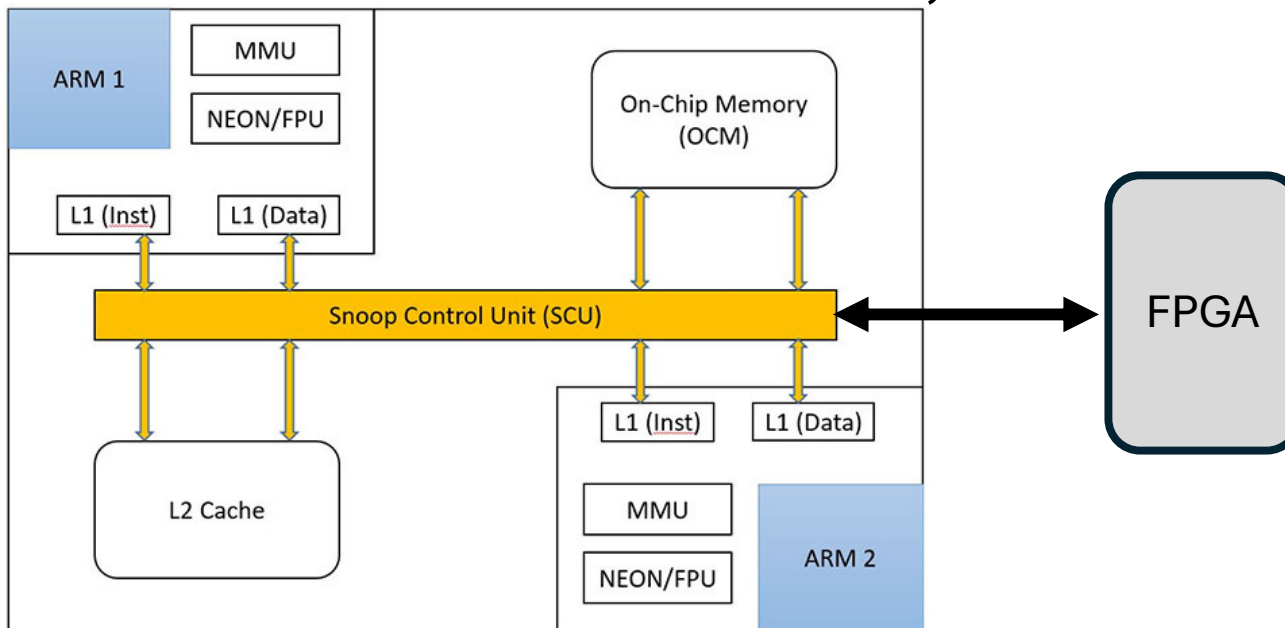
- Because of MMIO, for the most part, data moves between these two entities through the L2 cache and then through the Snoop Control Unit
- You want the FPGA to see correct memory cache values just like the
- One exception is the OCM, but that is relatively small

Two Other “Better” Ways...

- It may be desirable to link more closely to the processor than through regular channels
 - Accelerator Coherency Port (ACP)
- You may need to move massive amounts of data into or out of memory and not want to go through caches arbitration and things.
 - Direct Memory Access (DMA)

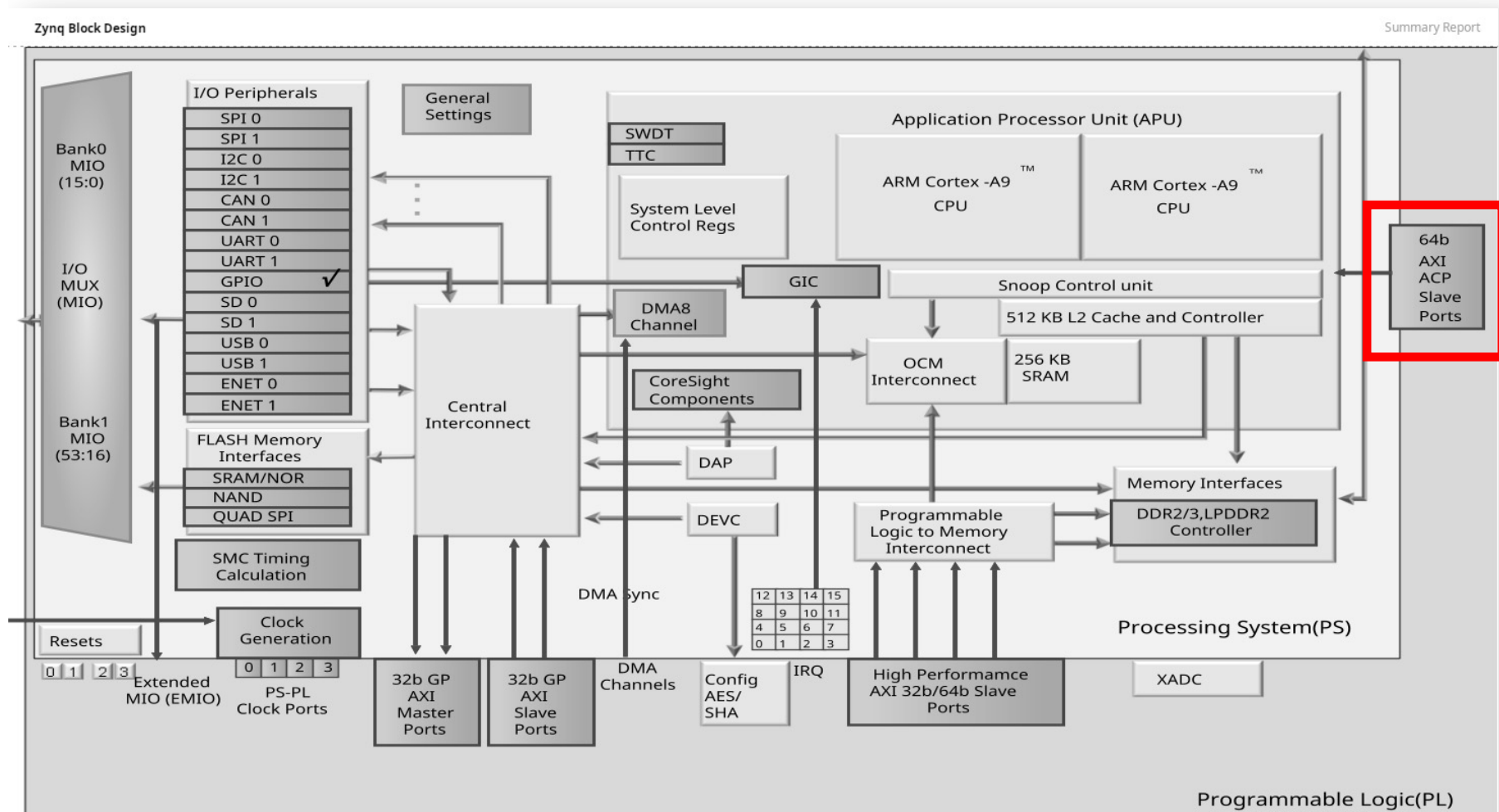
Accelerator Coherency Port (ACP)

- There is one Accelerator Coherency Port
- Direct Interface to SCU from the FPGA
- Allows quick, small-size interfacing between Processors and FPGA fabric, if needed.



<https://www.aldec.com/en/company/blog/144--introduction-to-zynq-architecture>

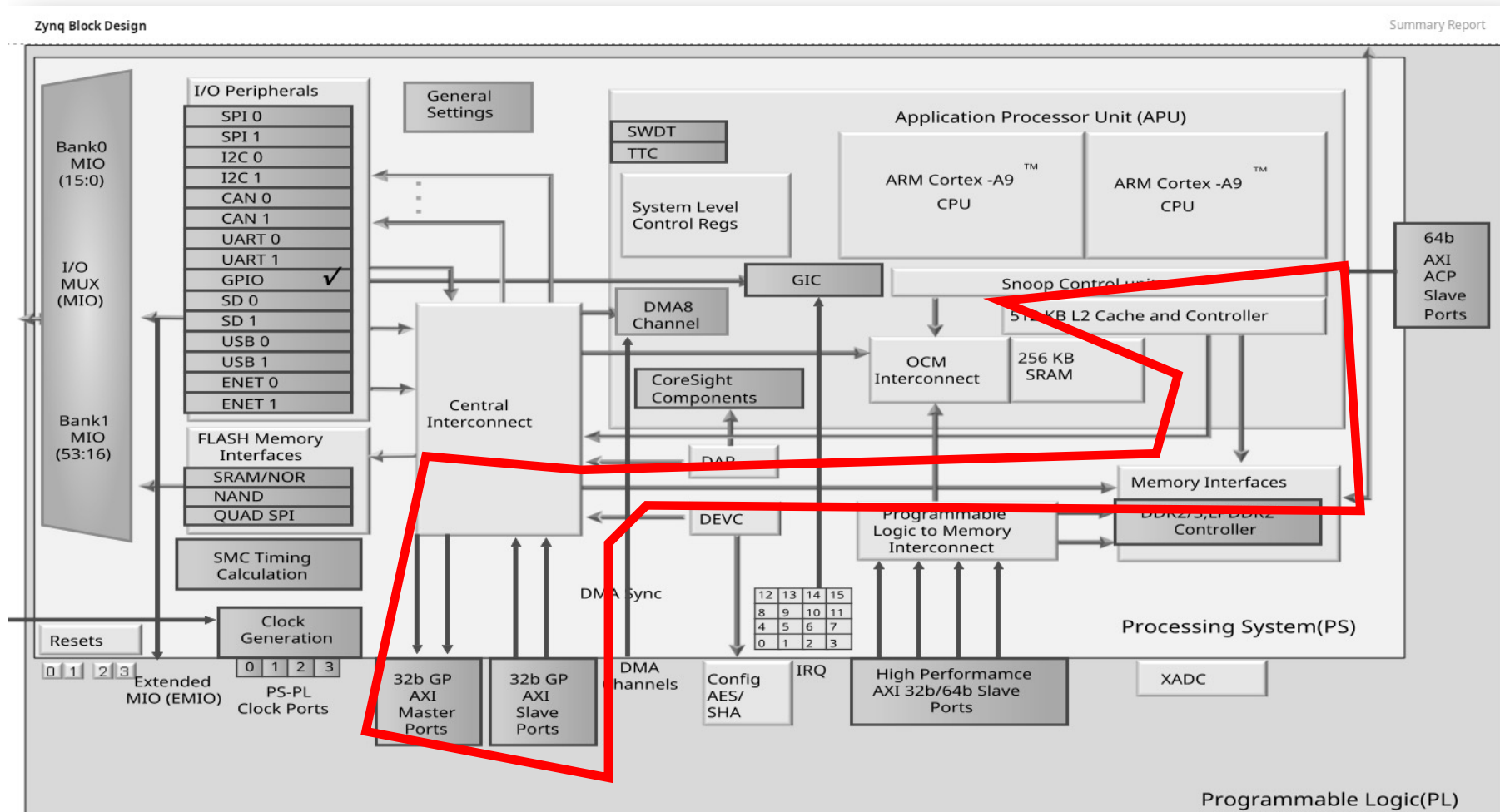
ACP (right side)



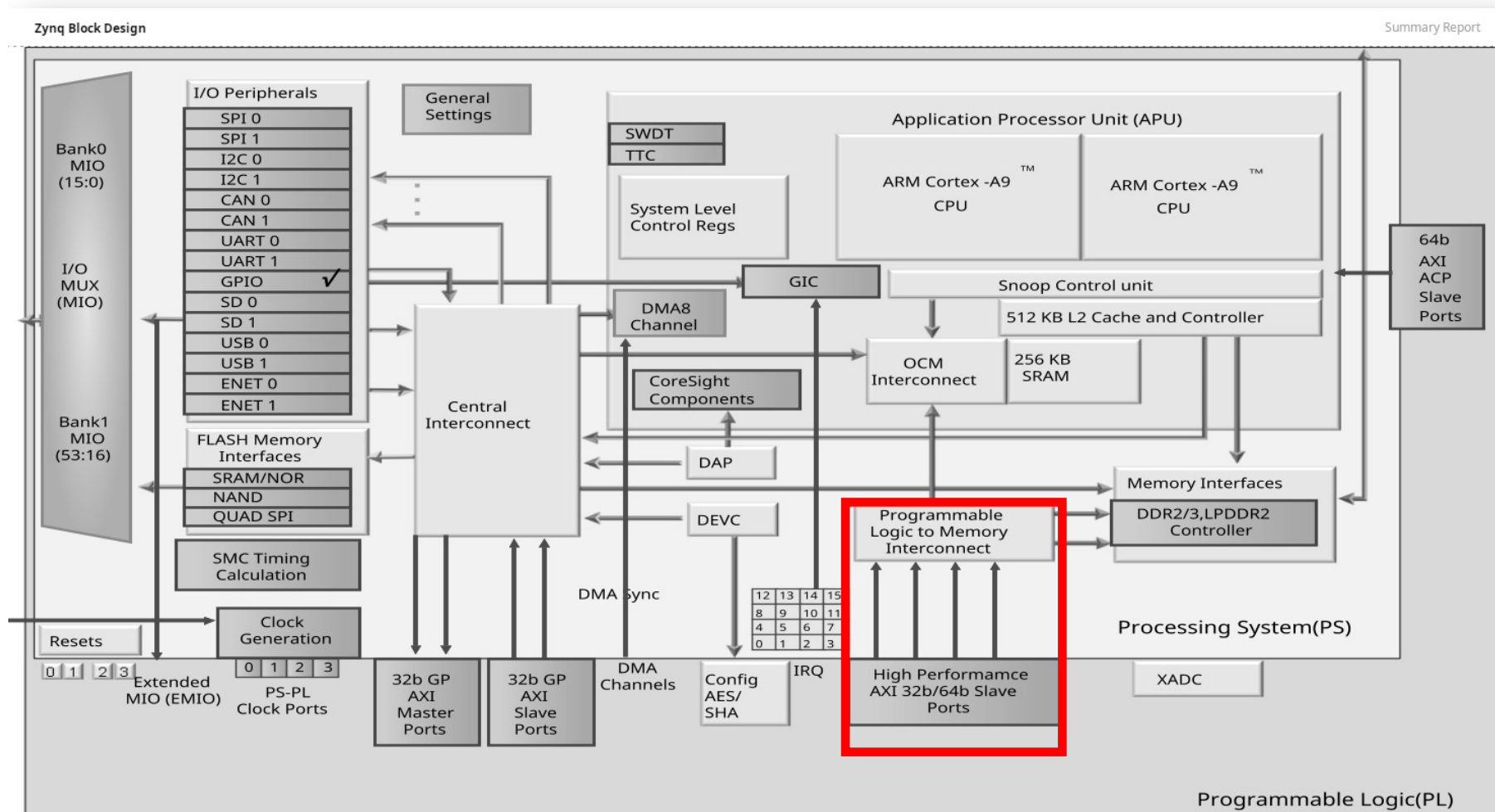
Two Other “Better” Ways...

- It may be desirable to link more closely to the processor than through regular channels
 - Accelerator Coherency Port (ACP)
- You may need to move massive amounts of data into or out of memory and not want to go through caches arbitration and things.
 - Direct Memory Access (DMA)

Conventional Memory Access



Direct Memory Access (DMA)

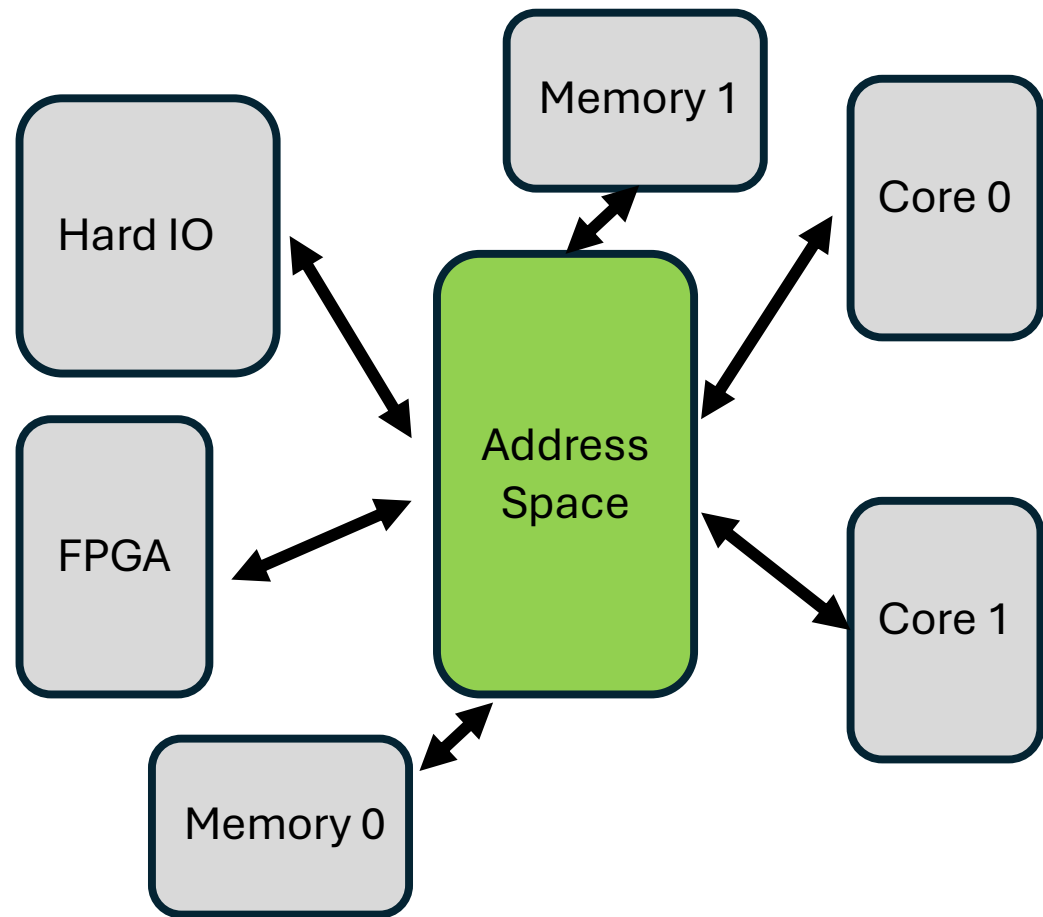


Direct Memory Access (DMA)

- Instead of having the FPGA interface through layers of caches (which can be slow), the DRAM Memory controller can be Accessed Directly from the FPGA.
- If used correctly, this can happen simultaneously with the processor running, provided it isn't having cache misses and going to DRAM
- Allows actual Memory-Mapped Linkage of information between PS and PL
- Can facilitate massive amounts of data (100's of MBs at very high speeds when done in bursts)

Illusion of Continuous Address Space

- Every piece in the entire system can talk and send messages back and forth using a consistent and global address scheme

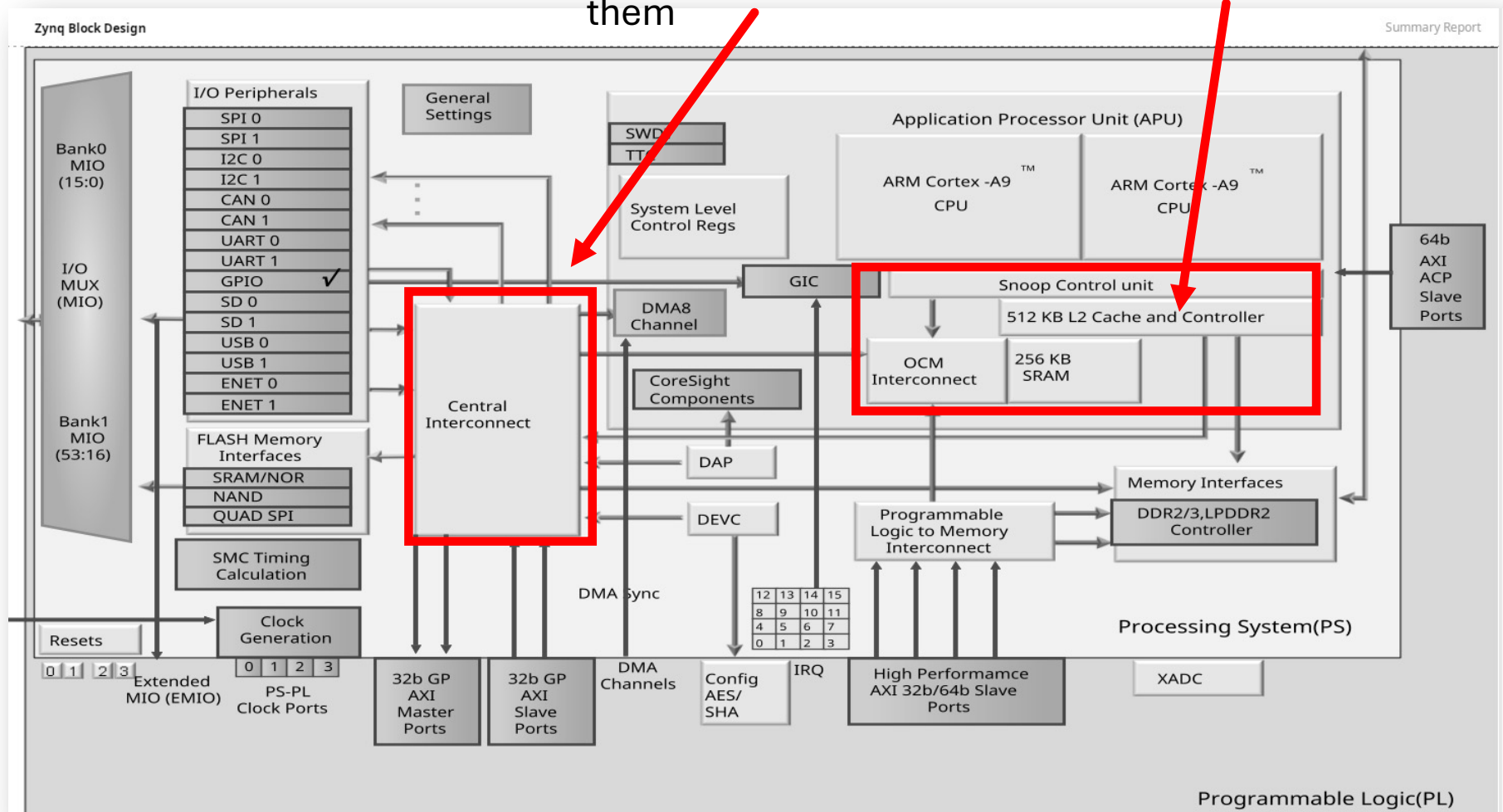


*Not really memory per say...

Zynq Block Diagram

Central Interconnect
handles address requests
between processor and
FPGA, external IO, etc...
Determines where to route
them

SCU and Cache
controllers know how to
direct address space
request to the correct
resources



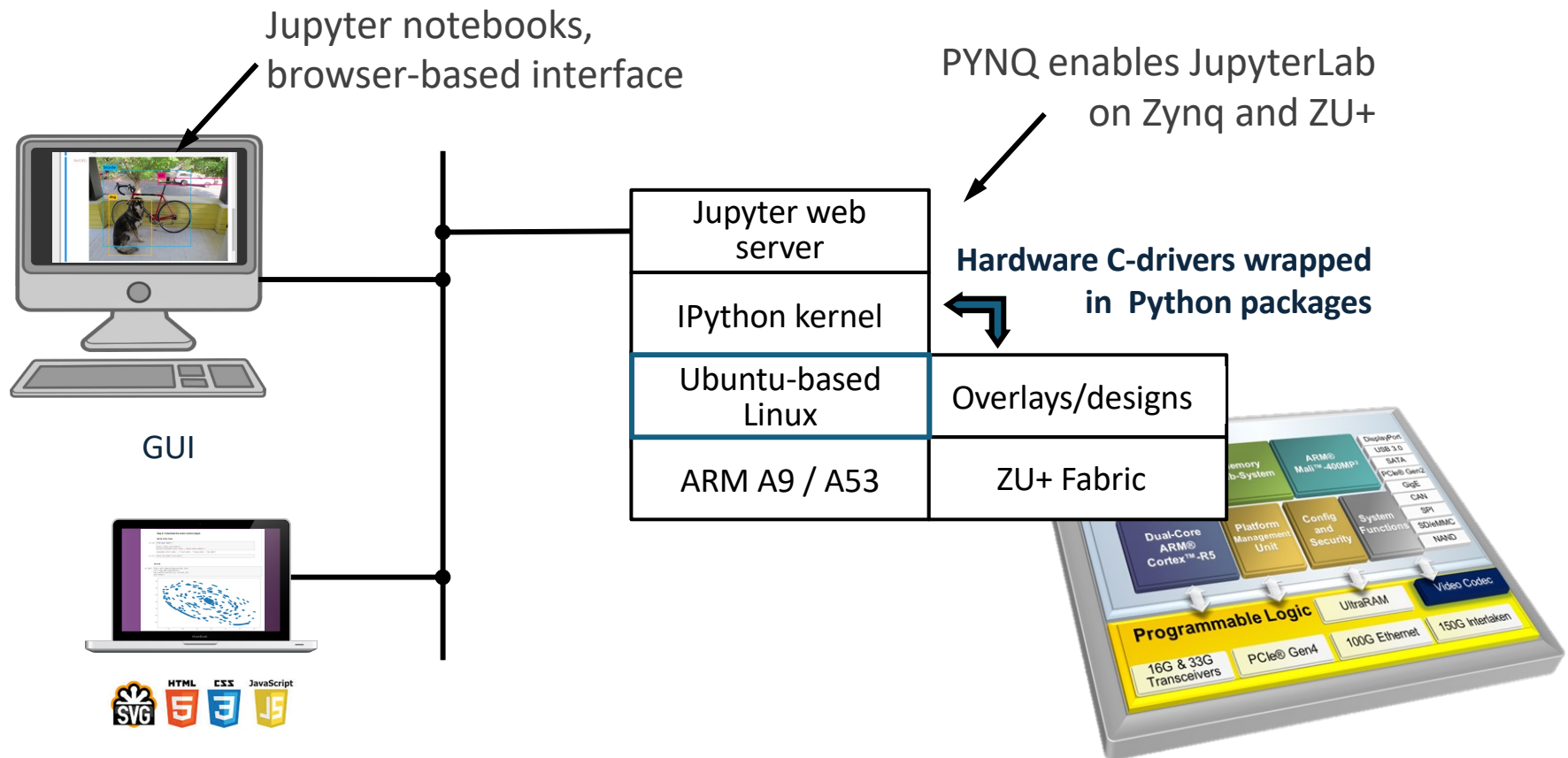
Address Space Handling

- All of this address space handling between the PS and the PL is accomplished with the Advanced Microcontroller Bus Architecture (**AMBA**)'s Advanced eXtensible Interface (or **AXI**) protocol
- We'll go over/review that in class this upcoming Wednesday (hopefully)

On Top of Low Level

Skipping AXI for a moment.

Python for Zynq...Pynq



Taken from some Xilinx talk I went to...

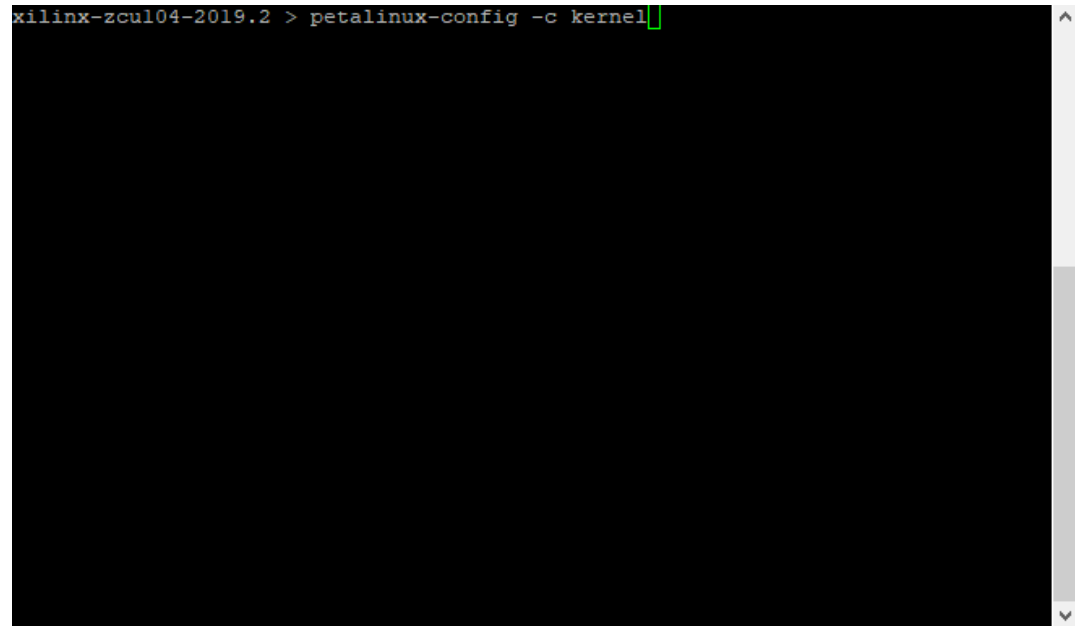
Yocto



- Yocto is a project dating back >10 years...focus of it is to build linux for embedded systems applications
- With Yocto you can basically build images of linux distributions targeted at small, particular processors (such as the ARM cores on the Zynq chip)
- Yocto is installed on your computer (kinda like any tool) and then you build for other systems...just like how we build for our FPGA with Vivado.

PetaLinux

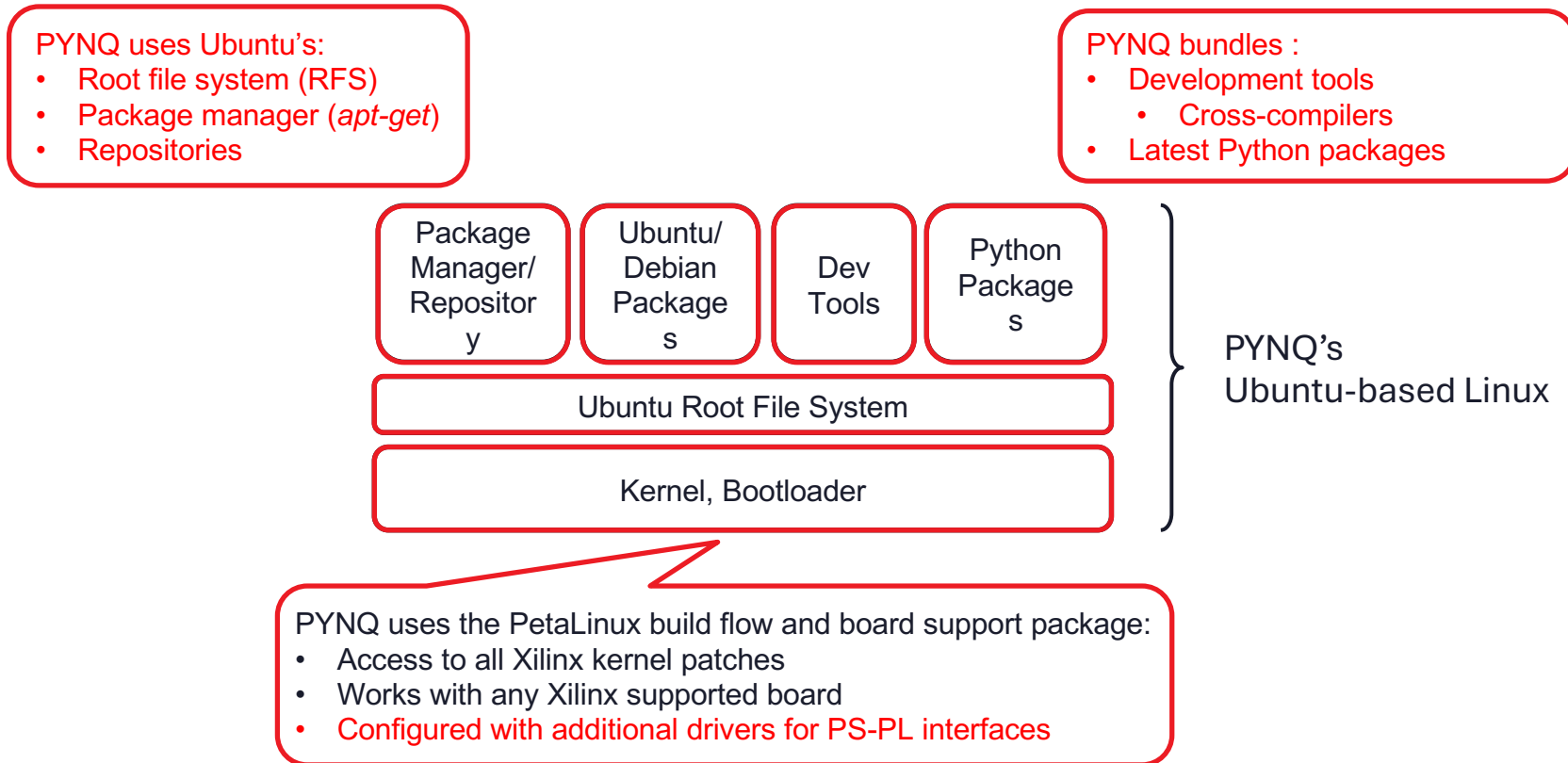
- AMD/Xilinx took Yocto, added some stuff on top intended to streamline these tools for their chips and architectures specifically and called it PetaLinux

A terminal window with a black background and white text. The prompt is 'xilinx-zcu104-2019.2 >'. The command 'petalinux-config -c kernel' is entered, followed by a green cursor. The terminal has a scrollbar on the right side.

```
xilinx-zcu104-2019.2 > petalinux-config -c kernel
```

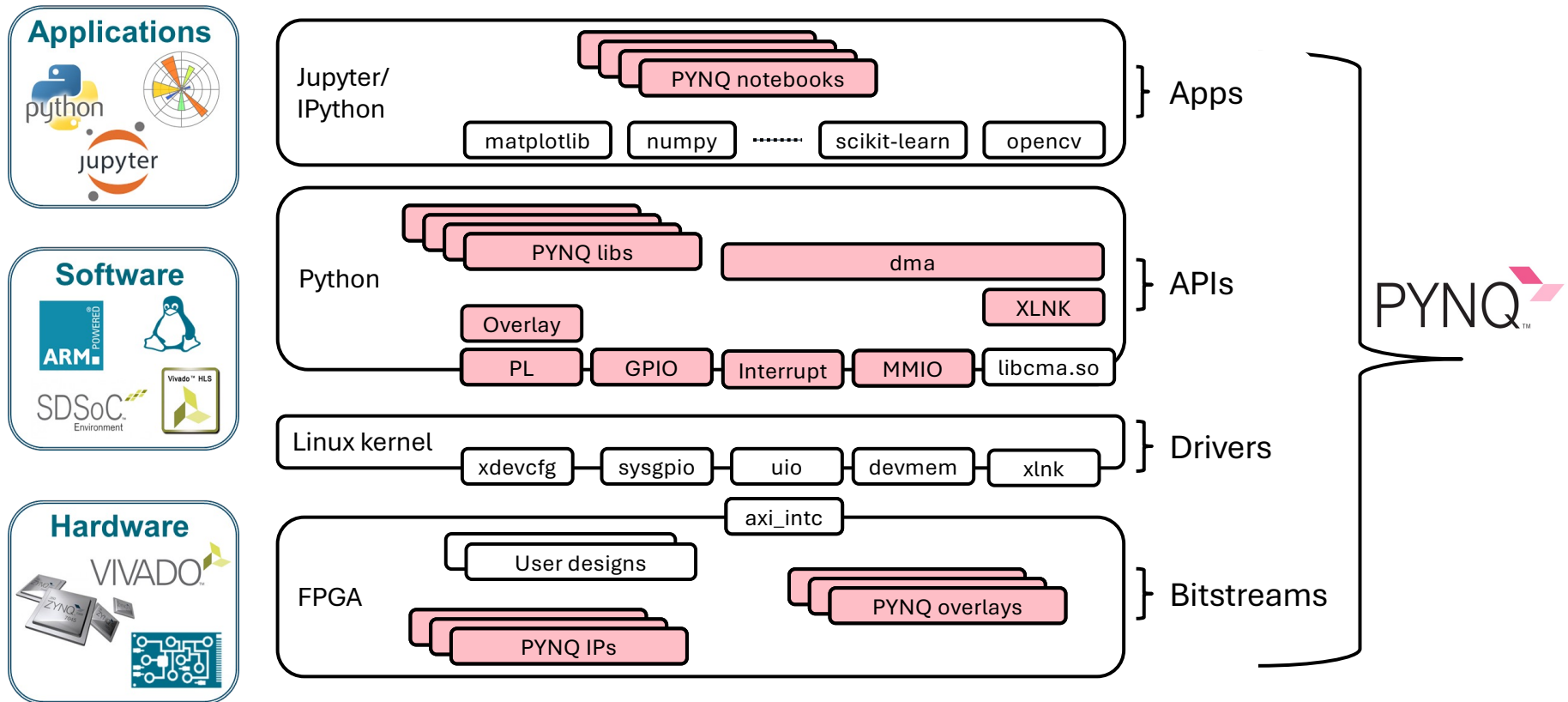
<https://discuss.pynq.io/t/deploying-pynq-and-jupyter-with-petalinux/677>

PYNQ uses an Ubuntu based Linux



Taken from some Xilinx talk I went to...

PYNQ Framework

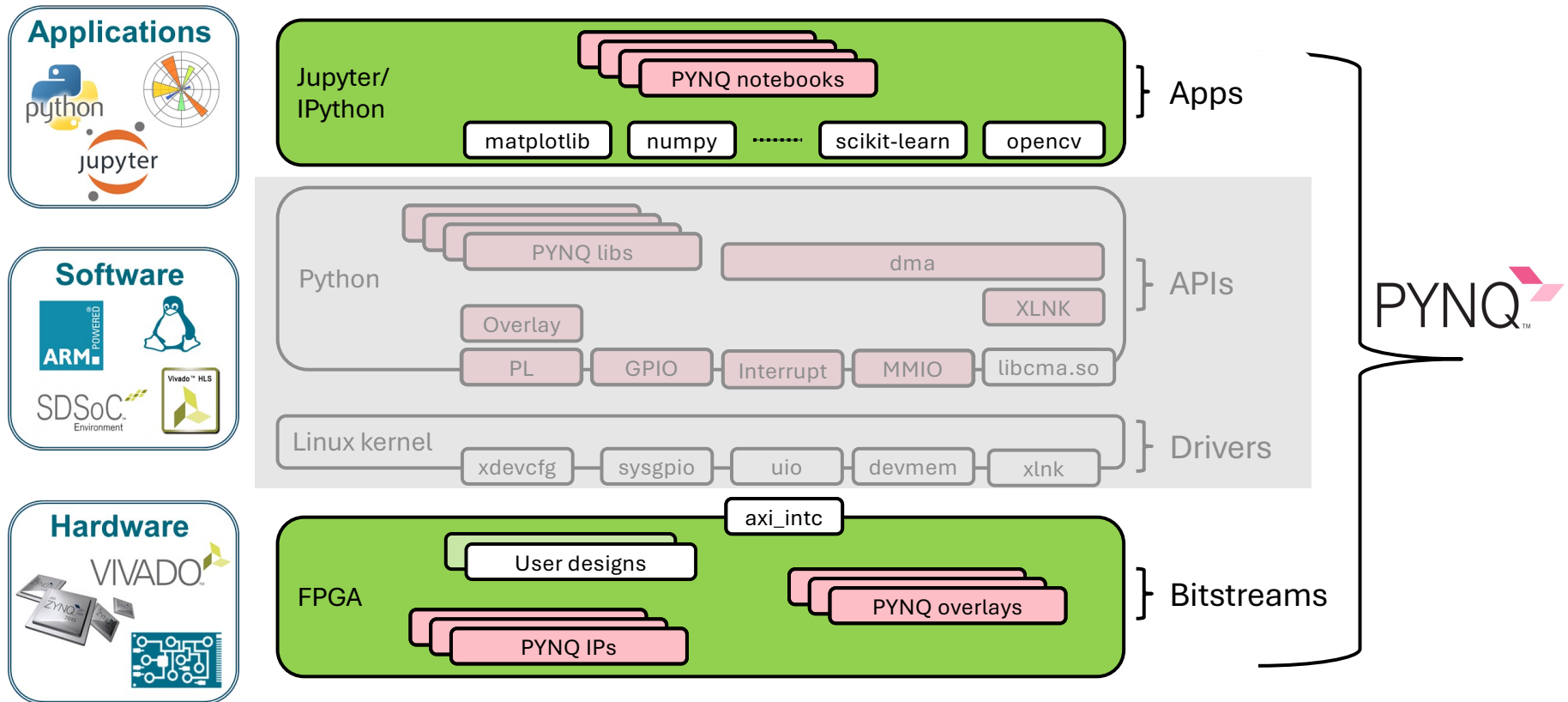


Taken from some Xilinx talk I went to...

Pynq Compromises

- With the Pynq framework you're basically starting with a pre-built Yocto/Petalinux implementation that people have already designed for you.
- To get the most out of a chip, one may want to go and do their own custom version and build and then make an image.
- You can 100% build your own PYNQ image from scratch or with modifications:
 - https://pynq.readthedocs.io/en/latest/pynq_sd_card.html

We're largely ignoring middle part

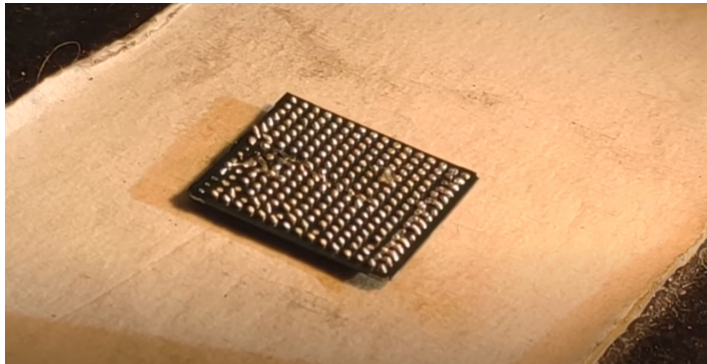


Taken from some Xilinx talk I went to...

Physical Pinout of Pynq

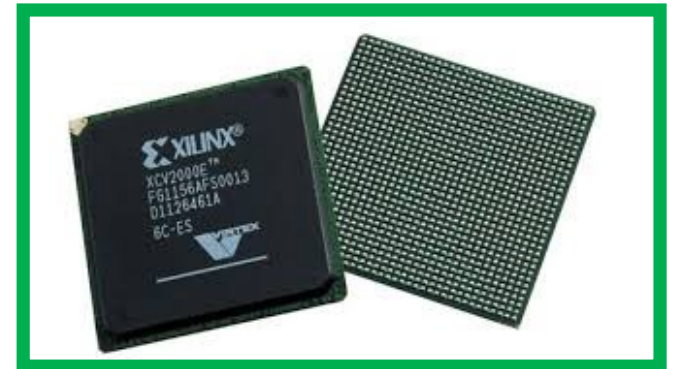
ZYNQ 7020 is a chip like any other chip

- Zynq package is a ball grid array (all pins are underneath)
- One of the most unforgiving packages out there...



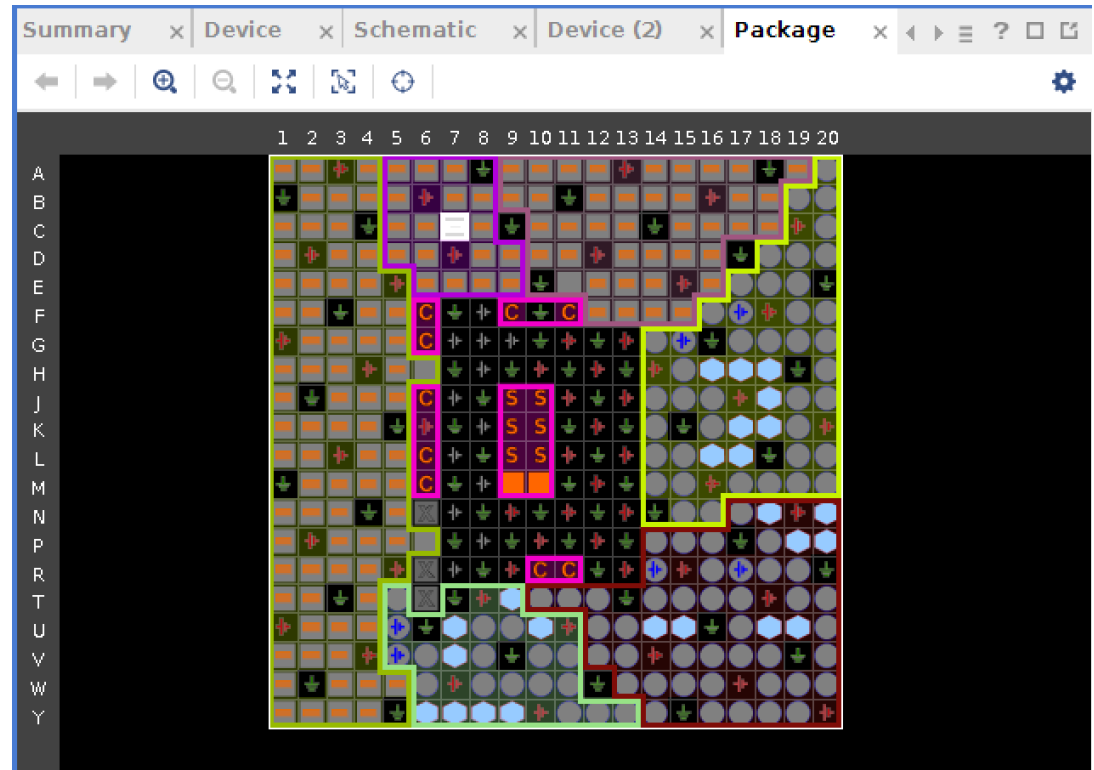
Still from video of somebody “reballing” an Xilinx chip

<https://www.youtube.com/watch?v=DVTxHx0z-wo>



Assigning Pins

Once design is synthesized you can specify where to route (we'll not do this much since much of this has been decided ahead of time with the PYNQ board's PCB layout, but if you were designing with the chip from scratch this would be part of process



- Pinout file can be found here:
- <https://www.xilinx.com/content/dam/xilinx/support/packagefiles/z7/packages/xc7z020clg400pkg.txt>

400 Pins Listed Out

- Some pins connect to the PL part of chip
- Some pins connect to the PS part of chip.
- Just how it goes...

9/15/25

Device/Package xc7z020clg400 9/18/2012 09:51:09

Pin	Pin Name	Memory	Byte	Group	Bank	VCCAUX	Group	Super	Logic	Region	I/O Type	No-Connect
R11	DONE_0	NA			0	NA		NA			CONFIG	NA
M9	DXP_0	NA			0	NA		NA			CONFIG	NA
J10	GNDADC_0	NA			0	NA		NA			CONFIG	NA
J9	VCCADC_0	NA			0	NA		NA			CONFIG	NA
L9	VREFP_0	NA			0	NA		NA			CONFIG	NA
L10	VN_0	NA			0	NA		NA			CONFIG	NA
F11	VCCBATT_0	NA			0	NA		NA			CONFIG	NA
F9	TCK_0	NA			0	NA		NA			CONFIG	NA
M10	DXN_0	NA			0	NA		NA			CONFIG	NA
K10	VREFN_0	NA			0	NA		NA			CONFIG	NA
K9	VP_0	NA			0	NA		NA			CONFIG	NA
F10	RSVDGND	NA			0	NA		NA			CONFIG	NA
N6	RSVDVCC3	NA			0	NA		NA			CONFIG	NA
R6	RSVDVCC2	NA			0	NA		NA			CONFIG	NA
R10	INIT_B_0	NA			0	NA		NA			CONFIG	NA
G6	TDI_0	NA			0	NA		NA			CONFIG	NA
F6	TDO_0	NA			0	NA		NA			CONFIG	NA
T6	RSVDVCC1	NA			0	NA		NA			CONFIG	NA
M6	CFGBVS_0	NA			0	NA		NA			CONFIG	NA
L6	PROGRAM_B_0	NA			0	NA		NA			CONFIG	NA
J6	TMS_0	NA			0	NA		NA			CONFIG	NA
V5	IO_L6N_T0_VREF_13	0			13	NA		NA			HR	7Z010
U7	IO_L11P_T1_SRCC_13	1			13	NA		NA			HR	7Z010
V7	IO_L11N_T1_SRCC_13	1			13	NA		NA			HR	7Z010
T9	IO_L12P_T1_MRCC_13	1			13	NA		NA			HR	7Z010
U10	IO_L12N_T1_MRCC_13	1			13	NA		NA			HR	7Z010
Y7	IO_L13P_T2_MRCC_13	2			13	NA		NA			HR	7Z010
Y6	IO_L13N_T2_MRCC_13	2			13	NA		NA			HR	7Z010
Y9	IO_L14P_T2_SRCC_13	2			13	NA		NA			HR	7Z010
Y8	IO_L14N_T2_SRCC_13	2			13	NA		NA			HR	7Z010
V8	IO_L15P_T2_DQS_13	2			13	NA		NA			HR	7Z010
W8	IO_L15N_T2_DQS_13	2			13	NA		NA			HR	7Z010
U9	IO_L16P_T2_13	2			13	NA		NA			HR	7Z010
J19	IO_L10N_T1_AD11N_35	1			35	NA		NA			HR	NA
L16	IO_L11P_T1_SRCC_35	1			35	NA		NA			HR	NA
L17	IO_L11N_T1_SRCC_35	1			35	NA		NA			HR	NA
K17	IO_L12P_T1_MRCC_35	1			35	NA		NA			HR	NA
K18	IO_L12N_T1_MRCC_35	1			35	NA		NA			HR	NA
H16	IO_L13P_T2_MRCC_35	2			35	NA		NA			HR	NA
H17	IO_L13N_T2_MRCC_35	2			35	NA		NA			HR	NA
J18	IO_L14P_T2_AD4P_SRCC_35	2			35	NA		NA			HR	NA
H18	IO_L14N_T2_AD4N_SRCC_35	2			35	NA		NA			HR	NA
F19	IO_L15P_T2_DQS_AD12P_35	2			35	NA		NA			HR	NA
F20	IO_L15N_T2_DQS_AD12N_35	2			35	NA		NA			HR	NA
G17	IO_L16P_T2_35	2			35	NA		NA			HR	NA
G18	IO_L16N_T2_35	2			35	NA		NA			HR	NA
J20	IO_L17P_T2_AD5P_35	2			35	NA		NA			HR	NA
H20	IO_L17N_T2_AD5N_35	2			35	NA		NA			HR	NA
G19	IO_L18P_T2_AD13P_35	2			35	NA		NA			HR	NA
G20	IO_L18N_T2_AD13N_35	2			35	NA		NA			HR	NA
H15	IO_L19P_T3_35	3			35	NA		NA			HR	NA
G15	IO_L19N_T3_VREF_35	3			35	NA		NA			HR	NA
K14	IO_L20P_T3_AD6P_35	3			35	NA		NA			HR	NA
J14	IO_L20N_T3_AD6N_35	3			35	NA		NA			HR	NA
N15	IO_L21P_T3_DQS_AD14P_35	3			35	NA		NA			HR	NA
N16	IO_L21N_T3_DQS_AD14N_35	3			35	NA		NA			HR	NA
L14	IO_L22P_T3_AD7P_35	3			35	NA		NA			HR	NA
L15	IO_L22N_T3_AD7N_35	3			35	NA		NA			HR	NA
M14	IO_L23P_T3_35	3			35	NA		NA			HR	NA
M15	IO_L23N_T3_35	3			35	NA		NA			HR	NA
K16	IO_L24P_T3_AD15P_35	3			35	NA		NA			HR	NA
J16	IO_L24N_T3_AD15N_35	3			35	NA		NA			HR	NA
J15	IO_25_35	NA			35	NA		NA			HR	NA
E7	PS_CLK_500	NA			500	NA		NA			MIO	NA
E11	PS_MIO_VREF_501	NA			501	NA		NA			MIO	NA
C7	PS_POR_B_500	NA			500	NA		NA			MIO	NA
C8	PS_MIO15_500	NA			500	NA		NA			MIO	NA
E14	PS_MIO17_501	NA			501	NA		NA			MIO	NA
D10	PS_MIO19_501	NA			501	NA		NA			MIO	NA
F14	PS_MIO21_501	NA			501	NA		NA			MIO	NA
D11	PS_MIO23_501	NA			501	NA		NA			MIO	NA
F15	PS_MIO25_501	NA			501	NA		NA			MIO	NA
D13	PS_MIO27_501	NA			501	NA		NA			MIO	NA
C13	PS_MIO29_501	NA			501	NA		NA			MIO	NA
E16	PS_MIO31_501	NA			501	NA		NA			MIO	NA
D15	PS_MIO33_501	NA			501	NA		NA			MIO	NA

Aside...The RFSoc is Bigger

- Go to this site (<https://www.xilinx.com/support/package-pinout-files/zynq-ultrascale-plus-pkgs.html>) and use the non-functional sort tools to find the pin file for the xczu48
- You'll see that it is a 1156 pin BGA

Showing 1 - 24 of 24

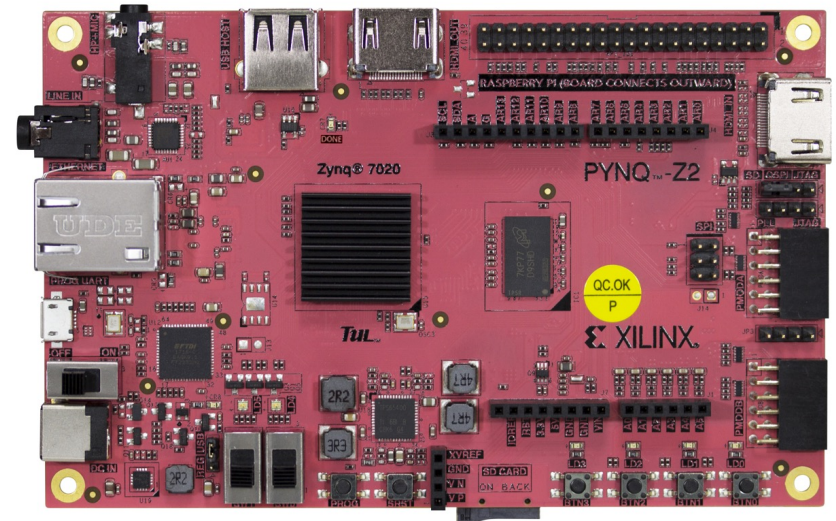
Sort By: Featured

Download Table

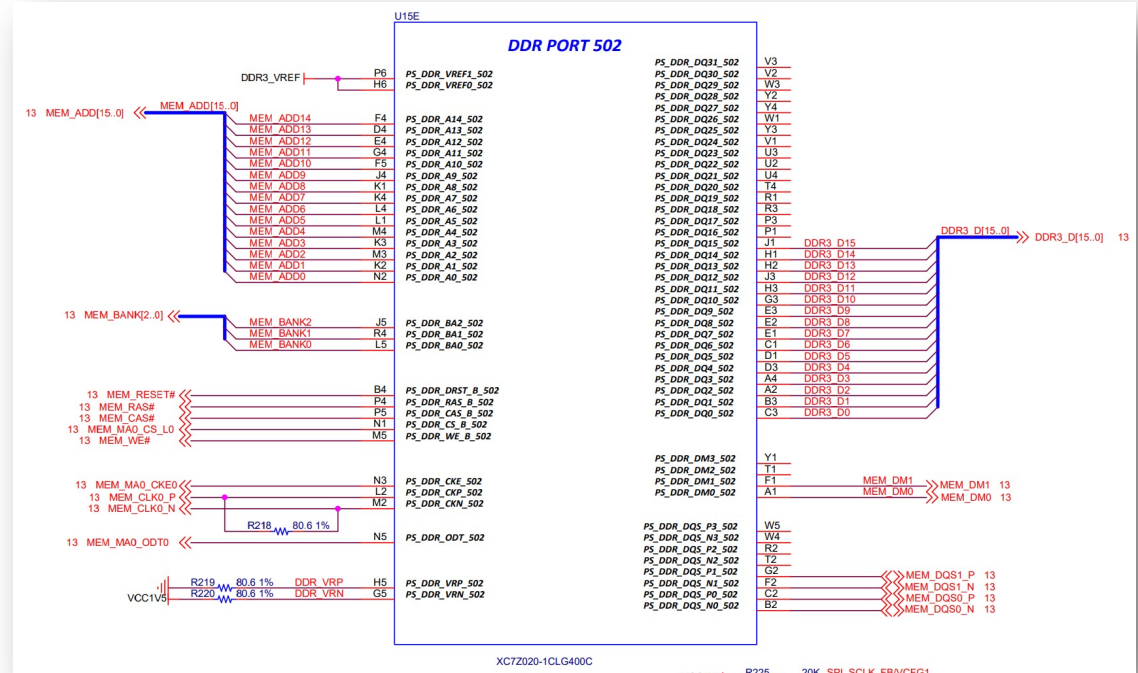
Compare	Mfr Part #	Quantity Available	Price	Series	Package	Product Status	Architecture	Core Processor	Flash Size	RAM Size	Peripherals	Connectivity	Speed
<input type="checkbox"/>	<div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></div><div><div><div><div></div><div></div></div><div><div></div><div></div></div></div></</div></div></div></div>												

Now, the Pynq Z2 board made some choices for us

- If you were the engineer laying out the chip/board from scratch you would also need to make these decision.
- Some decisions have very little wiggle room, others do.



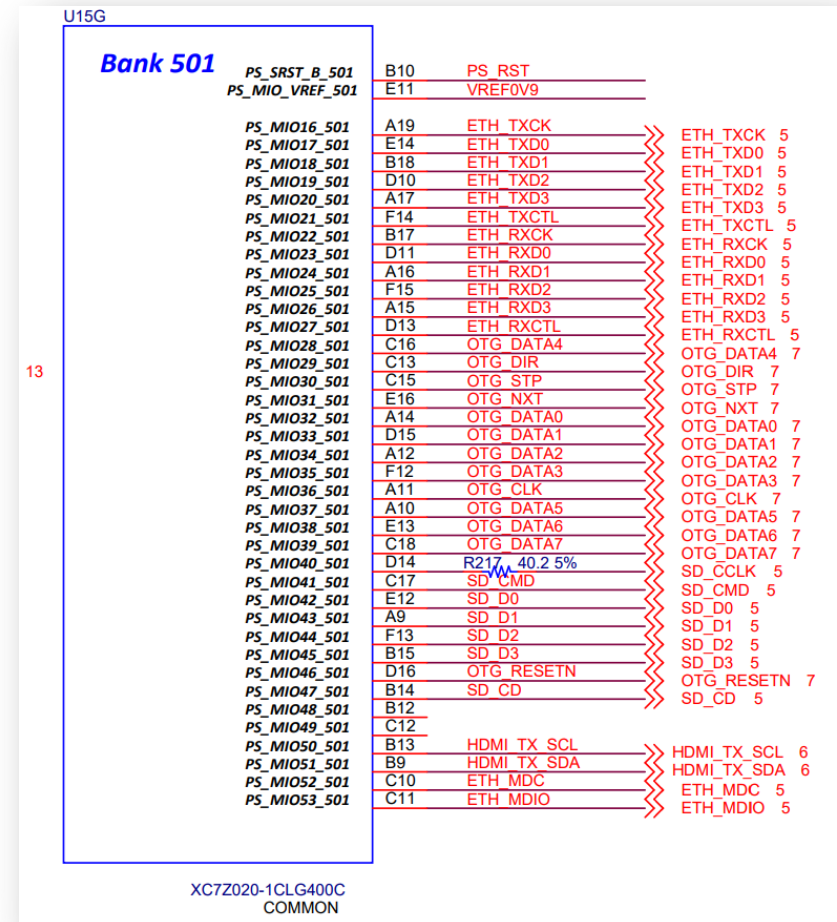
Schematic of PYNQ Z2 *Board*



- The 512 MB DRAM is routed to PS_... Pins of the Zynq chip.
- Meaning the DRAM is only accessible in the processing side
- There is no PL-only accessible DRAM

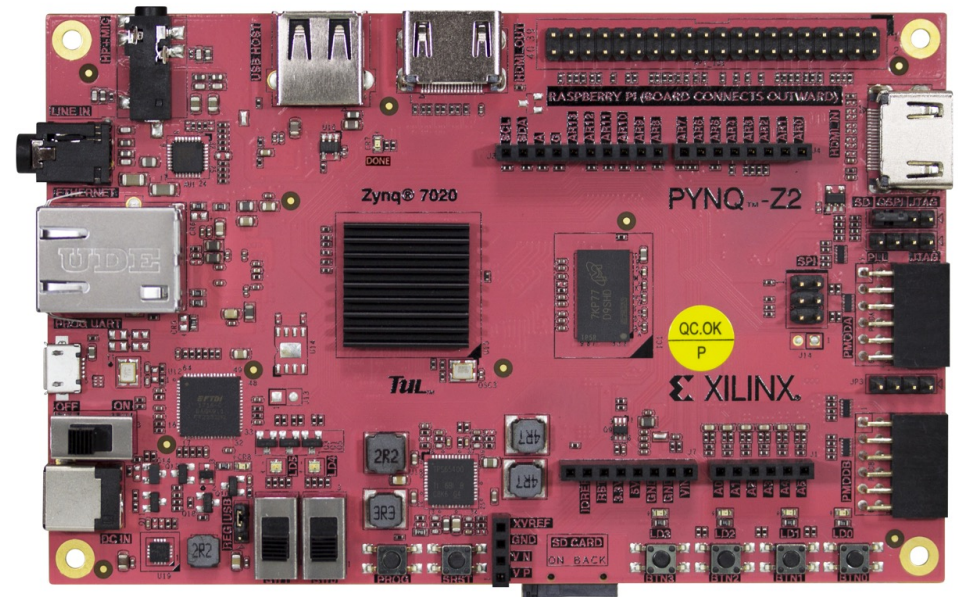
Schematic of PYNQ Z2 Board

- Ethernet, SD card, some HDMI control portions, OTG/USB are all also wired to PS_ pins
- That means those are not accessible via



Most other things on the board are actually wired to pins that are part of the PL (Programmable Logic)

- So pretty much everything else...
- All these the random pins, the audio, the HDMI in/out, buttons, etc...



List of I/O Peripherals for the PS:

- "Hard" IP cores exist on the **PS** that perform certain interfacing roles/protocols:
- These can be multiplexed out to many subsets of pins

I/O Interface	Description
SPI (x2)	Serial Peripheral Interface [10] <i>De facto standard for serial communications based on a 4-pin interface. Can be used either in master or slave mode.</i>
I2C (x2)	I ² C bus [14] <i>Compliant with the I2C bus specification, version 2. Supports master and slave modes.</i>
CAN (x2)	Controller Area Network <i>Bus interface controller compliant with ISO 118980-1, CAN 2.0A and CAN 2.0B standards.</i>
UART (x2)	Universal Asynchronous Receiver Transmitter <i>Low rate data modem interface for serial communication. Often used for Terminal connections to a host PC.</i>
GPIO	General Purpose Input/Output <i>There are 4 banks GPIO, each of 32 bits.</i>
SD (x2)	<i>For interfacing with SD card memory.</i>
USB (x2)	Universal Serial Bus <i>Compliant with USB 2.0, and can be used as a host, device, or flexibly ("on-the-go" or OTG mode, meaning that it can switch between host and device modes).</i>
GigE (x2)	Ethernet <i>Ethernet MAC peripheral, supporting 10Mbps, 100Mbps and 1Gbps modes.</i>

Taken from The Zynq Book

Using them

- In a normal microcontroller, you would simply activate a module, such as an SPI controller and connect it to some pins.
- The way the Pynq Z2 board is laid out you can't do that.
- In an effort to ensure flexibility for development, they connected most things and broke out most general IO from the PL side.

Assigning I/O pins to Hard IP

Here I double-clicked on the Zynq7 Processing IP Core

Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation Presets IP Location Import XPS Settings

Page Navigator

- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts

Peripheral I/O Pins

Search:

Peripherals

- ☐ SPI 0
- ☒ SPI 1
- ☐ UART 0
- ☒ UART 1
- ☐ I2C 0
- ☐ I2C 1
- ☒ CAN 0
- ☐ CAN 1
- ☐ TTC0
- ☐ TTC1
- ☐ SWDT
- ☐ PJTAG
- ☐ TPIU
- ☒ GPIO MIO
- ☐ GPIO EMIO

Bank 0 LVCMOS 3.3V Bank 1 LVCMOS 3.3V

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

SPI0 SPI1

UART0 UART1

I2C0 I2C1

CAN0 CAN1

TTC0 TTC1

SWDT

PJTAG

Trace

CAN 0 : [MIO 10-11]

OK Cancel

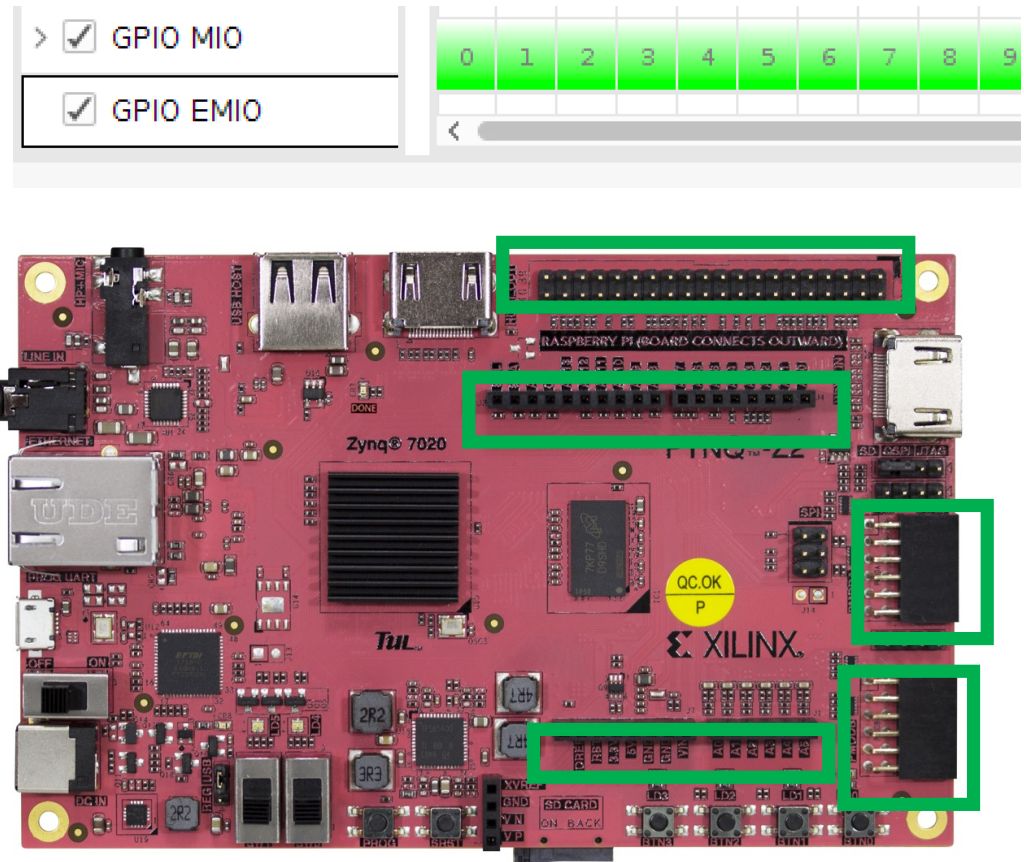
CAN and SPI can't share same pins!

UART is fine

GPIO

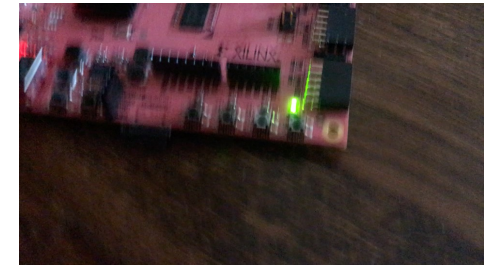
Linking to Outside World

- The I/O pins normally go to the outside world, but on our PYNQ board we need to extend them into the PL (which has its own actual physical output pins)
- Making the GPIO pins **EMIO** (Extended Multiplexed In/Out) puts them into the PL for further manipulation

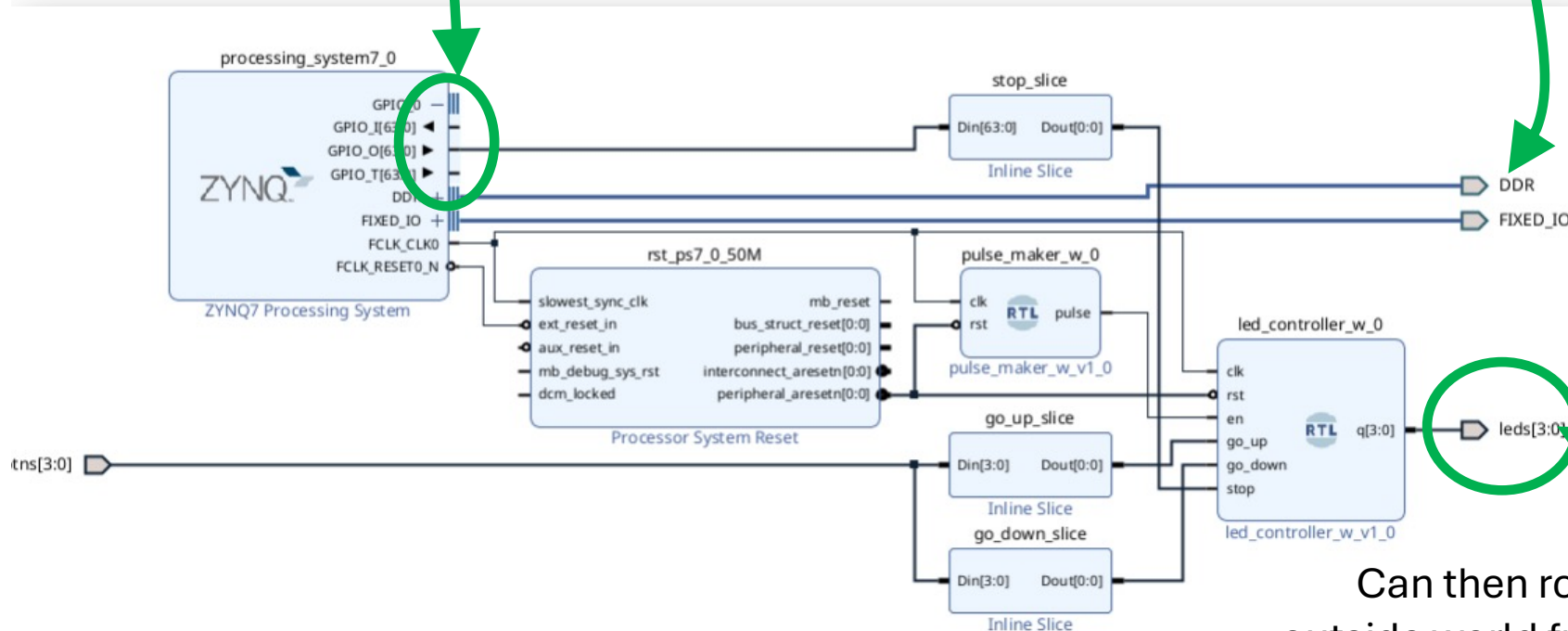


Week 1

We have specified the Zynq PS to route its IO pins out into the PL fabric and we can do what we want with them



These represent pins that come directly from the PS and interface with DRAM (DDR) and some hard-wired interfaces



Can then route into outside world from PL's bank of usable pins

Clicking on these things is really just a nice way to configure internal multiplexers

The screenshot shows the 'Re-customize IP' window for the ZYNQ7 Processing System (5.5). The 'Peripheral I/O Pins' tab is active, displaying a grid of pins for Bank 0 and Bank 1. The grid shows various peripherals mapped to specific pins. Annotations highlight that CAN and SPI cannot share the same pins, while UART is fine, and GPIO is also configured.

Peripheral I/O Pins Configuration:

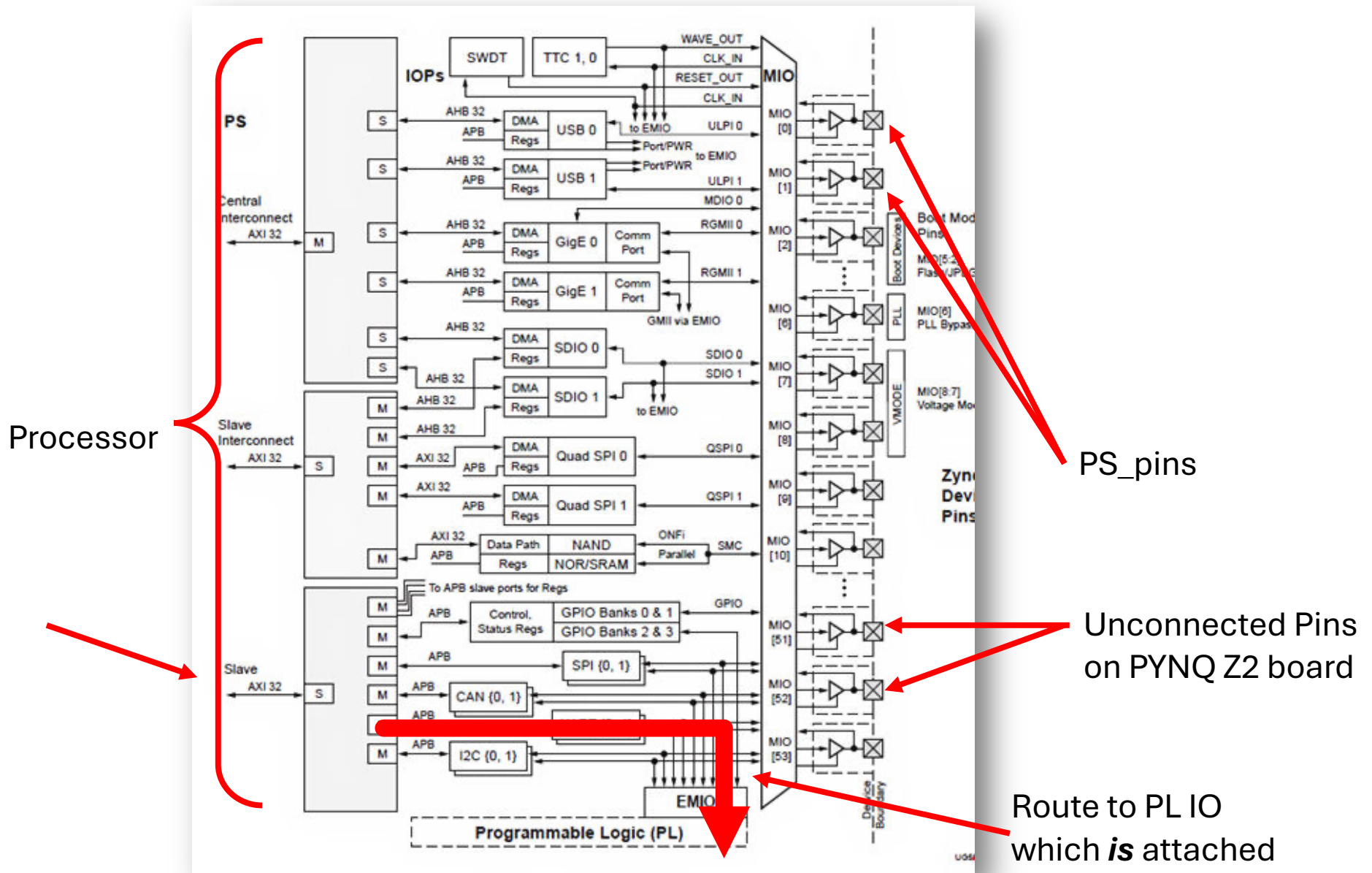
- Bank 0:** LVCMOS 3.3V
- Bank 1:** LVCMOS 3.3V

Peripherals:

- ☐ SPI 0
- ☒ SPI 1
- ☐ UART 0
- ☒ UART 1
- ☐ I2C 0
- ☐ I2C 1
- ☒ CAN 0
- ☐ CAN 1
- ☐ TTC0
- ☐ TTC1
- ☐ SWDT
- ☐ PJTAG
- ☐ TPIU
- ☒ GPIO MIO
- ☐ GPIO EMIO

Annotations:

- CAN and SPI can't share same pins!** (Blue arrow pointing to SPI1 and CAN0)
- UART is fine** (Blue arrow pointing to UART1)
- GPIO** (Blue arrow pointing to GPIO MIO)



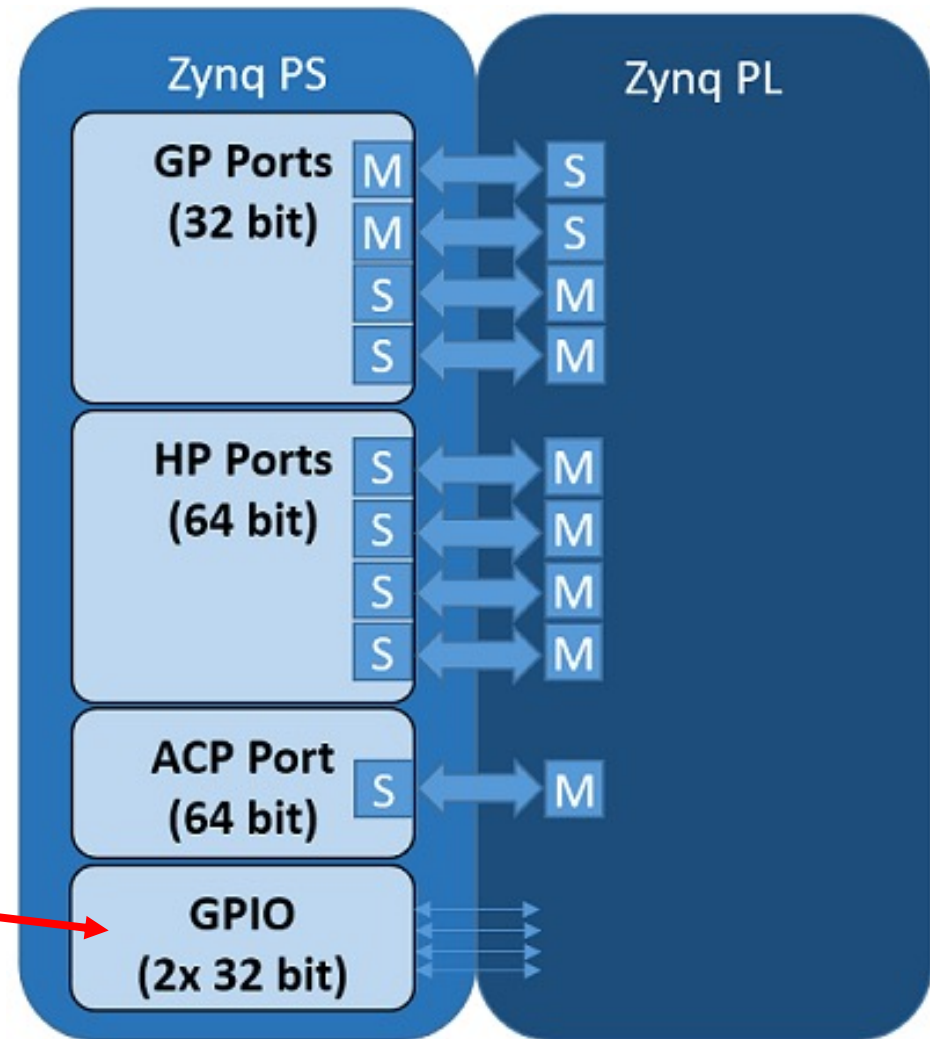
Taken from the *MicroZed Chronicles Blog/Xilinx Docs*

Other PL-PS Interconnects

Interface Between PS and PL

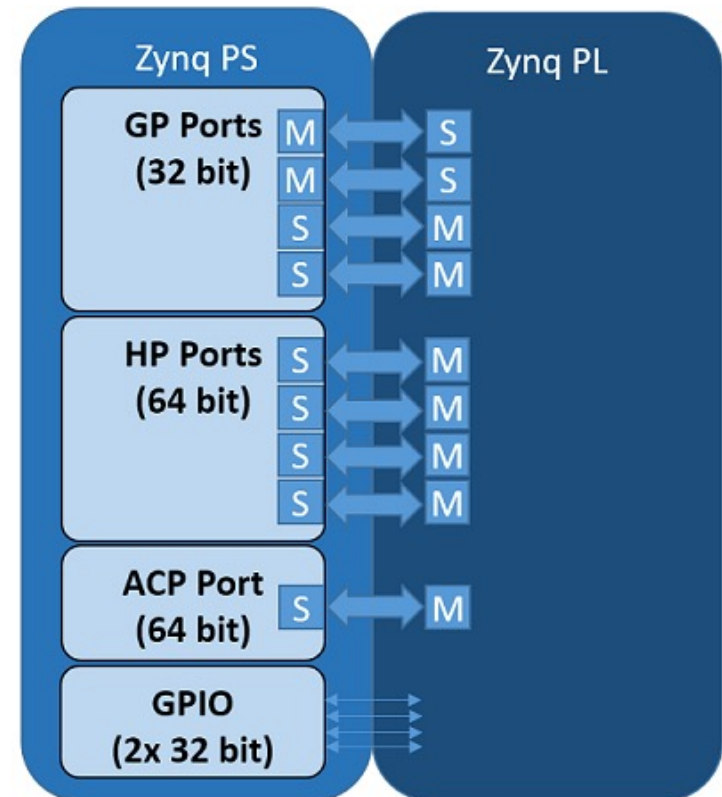
- Four Ways to Transfer Data from the PS to the PL
 - 64 bits of GPIO
 - 4 GP AXI Ports
 - 4 HP AXI Ports
 - 1 ACP Port

Just talked about this

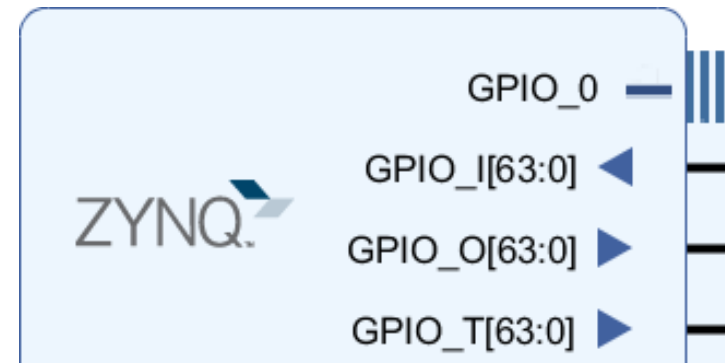


GPIO Pins

- **General Purpose Input Output**
- You can via software (writing to registers), control and be controlled by ~54 pins
- These are good for low-speed control, configuration, reset signals...things like that.



Interrupts

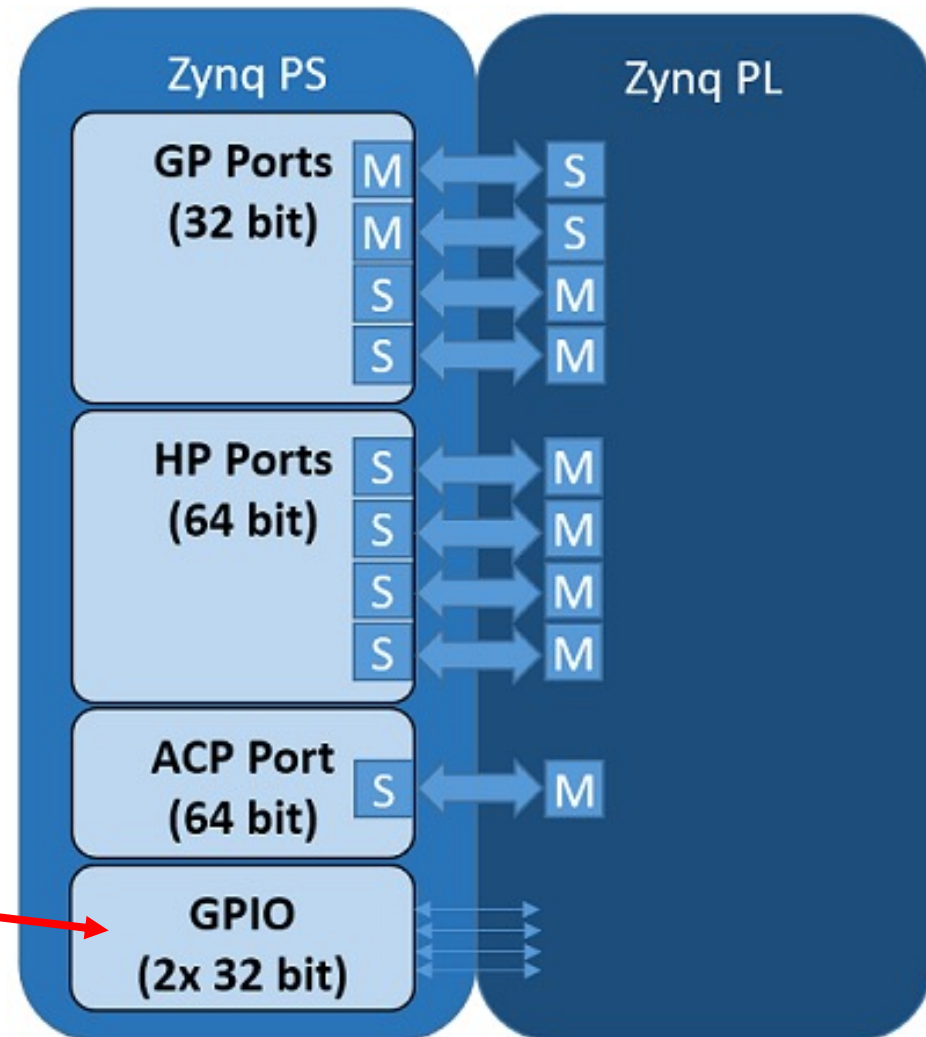


- The GPIO of the PS can be setup to have interrupts even when you are routing them “internally” into the PL Using EMIO.
- This means you can actually have the PL trigger Python processes to run by setting up the interrupts as well as some async programming on the Python side
- https://pynq.readthedocs.io/en/latest/pynq_libraries/interrupt.html
- https://pynq.readthedocs.io/en/latest/overlay_design_methodology/pynq_and_asyncio.html#pynq-and-asyncio

Interface Between PS and PL

- Four Ways to Transfer Data from the PS to the PL
 - 64 bits of GPIO
 - 4 GP AXI Ports
 - 4 HP AXI Ports
 - 1 ACP Port

Just talked about this

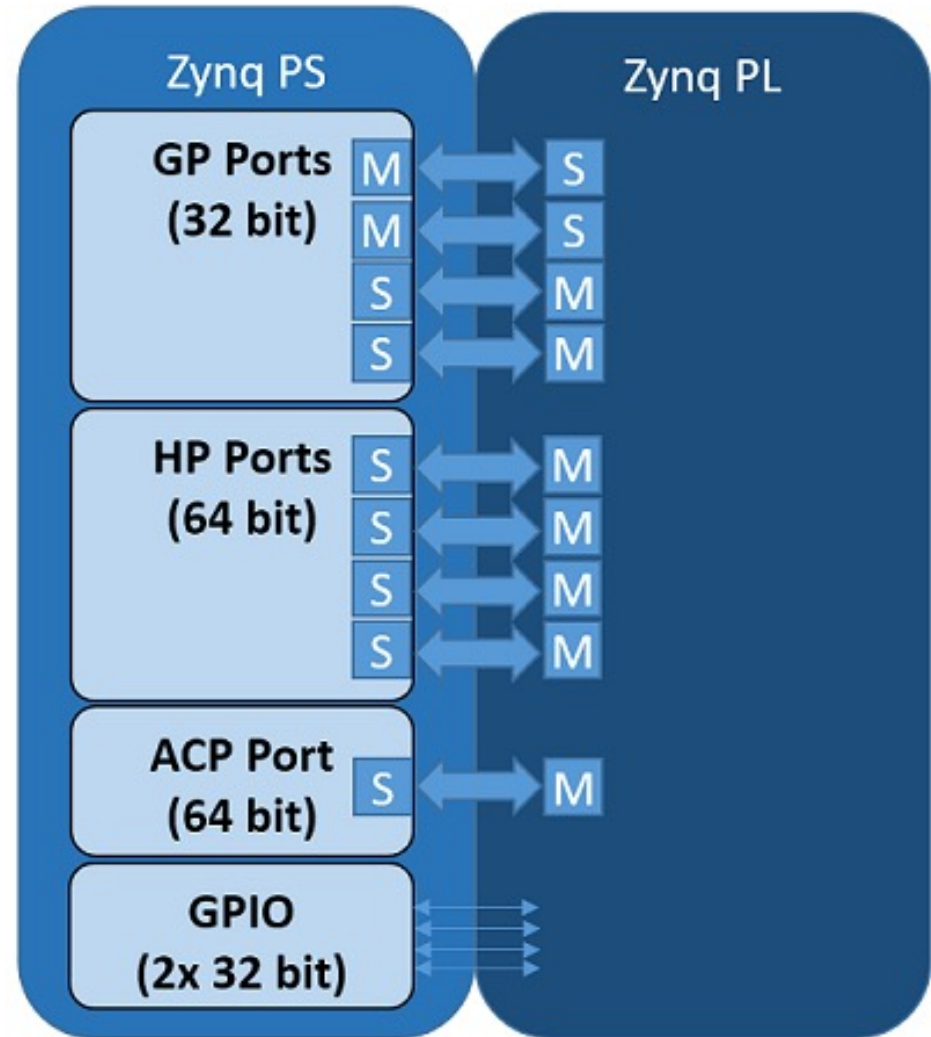


Master/Slave Terminology

- I've been a big fan of moving away from this terminology.
- For SPI, for example, instead of MOSI/MISO, do COPI/CIPO (controller/peripheral), etc...
- **However, all** of the AMD/Xilinx, use Master/Slave and **everything** has that M's and S's prepended, appended, etc..
- I'm going to just use their nomenclature so we don't have to constantly be mapping between alternate names.

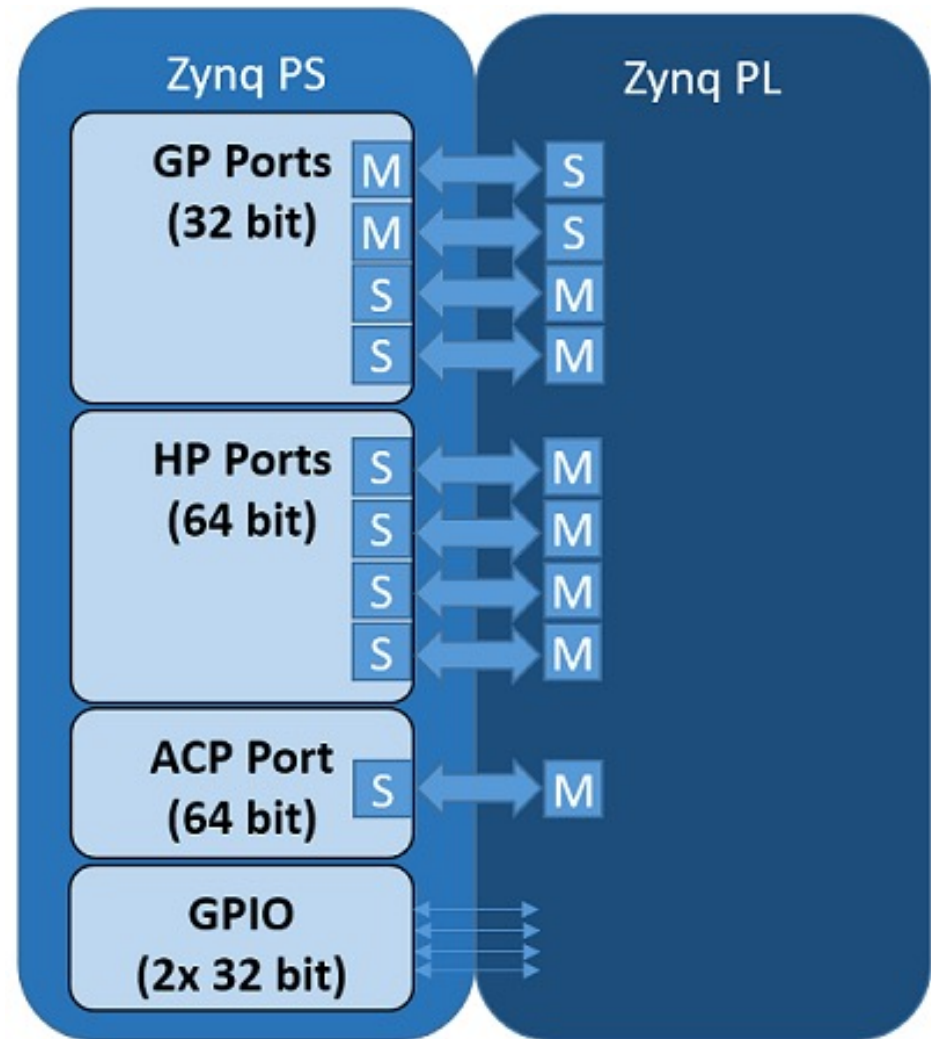
AXI Ports

- Parallel Busses of two different flavors that allow us to pretty quickly transfer data between the Processing System and the FPGA section using shared registers and some other stuff



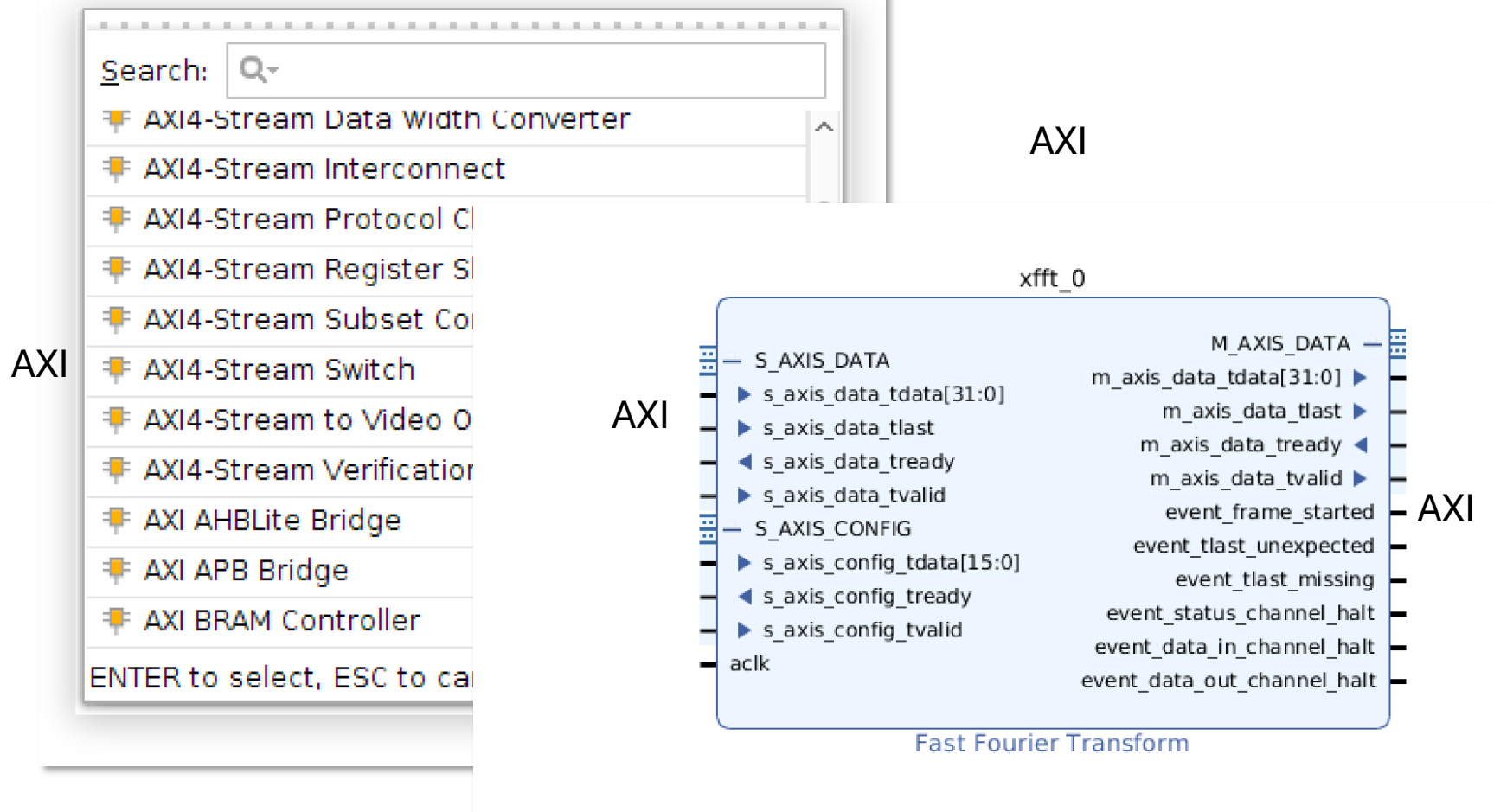
ACP Port

- Accelerator Coherency Port
- 64-bit wide bus that can transfer data from very quickly from PL fabric



AXI Everywhere

- There's lot of neat IP we can work with....if you wanted to implement a hardware accelerated Fast Fourier Transform you totally can...



Advanced Microcontroller Bus Architecture (AMBA)

- Version 1 released in 1996 by ARM
- 2003 saw release of **A**dvanced **eX**tensible Interface (AXI3)
- 2011 saw release of AXI4
- There are no royalties affiliated with AMBA/AXI so they're used a lot.
- It is a general, flexible, and relatively free* communication protocol for development

Three General Flavors of AXI4

- **AXI4 (Full AXI):** For memory-mapped links. Provides highest performance.
 1. Address is supplied
 2. Then a data burst transfer of up to 256 data words
- **AXI4 Lite:** A memory-mapped simplified link supporting only one data transfer per connection (no bursts). (also restricted to 32 bit addr/data)
 1. Address is supplied
 2. One data transfer
- **AXI4 Stream:** Meant high-speed streaming data
 - Can do burst transfers of unrestricted size
 - No addressing
 - Meant to stream data from one device to another quickly on its own direct connection

From the Zynq Book

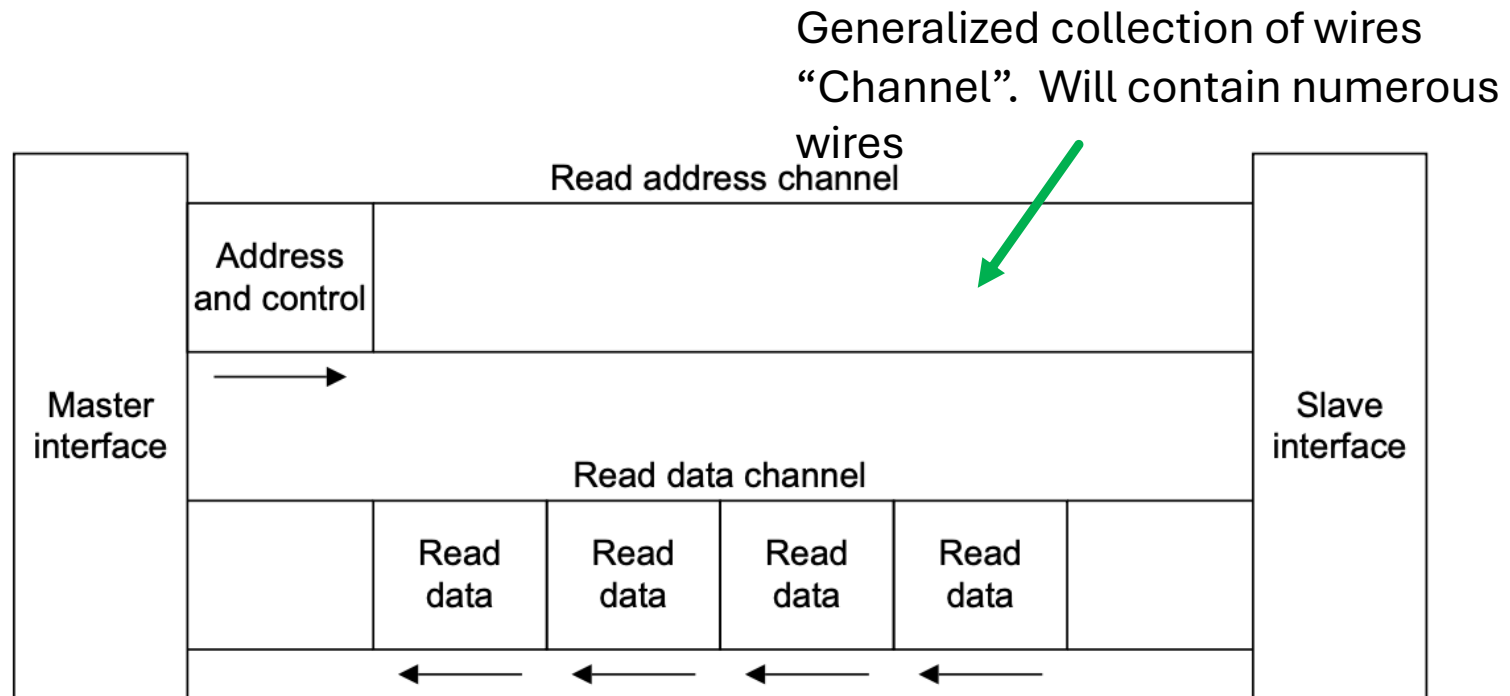
Memory Map?

- Memory mapped means an address is specified within the transaction by the master (read or write). This corresponds to an address in the system memory space.
- For **AXI4-Lite**, which supports a single data transfer per transaction, data is then written to, or read from, the specified address
- For **Full-AXI4** sending a burst, the address specified is for the first data word to be transferred, and the slave must then calculate the addresses for the data words that follow.
- **AXI-Stream** has no addressing so no memory mapping

AXI Idea

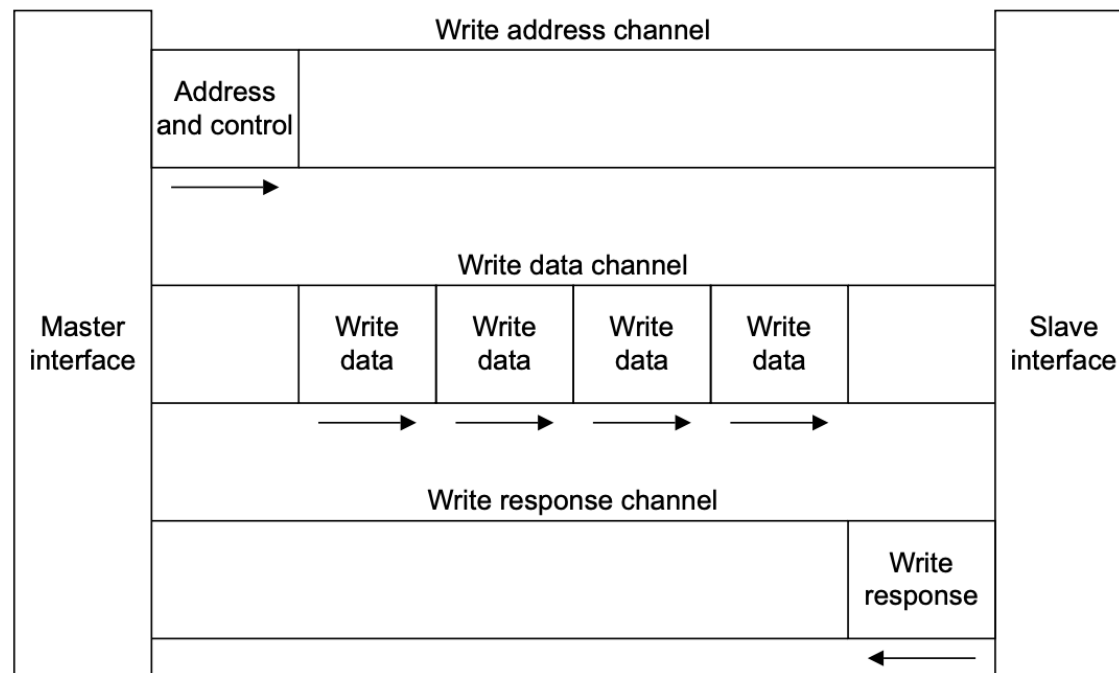
- Communication between two devices (Master and Slave) is carried out over multiple assigned “channels”
- Each channel has its own collection of wires which convey data, signals, etc.
- The channels can work somewhat independently, however in practice what one channel does is often the result of what a different one did previously
- Five Types of Channels (may have all or a subset):
 - Read Address: “AR” channel
 - Read Data: “R” channel
 - Write Address: “AW” channel
 - Write Data: “W” channel
 - Write Response: “B” channel

Read Wiring



Master initiates communication, Slave responds

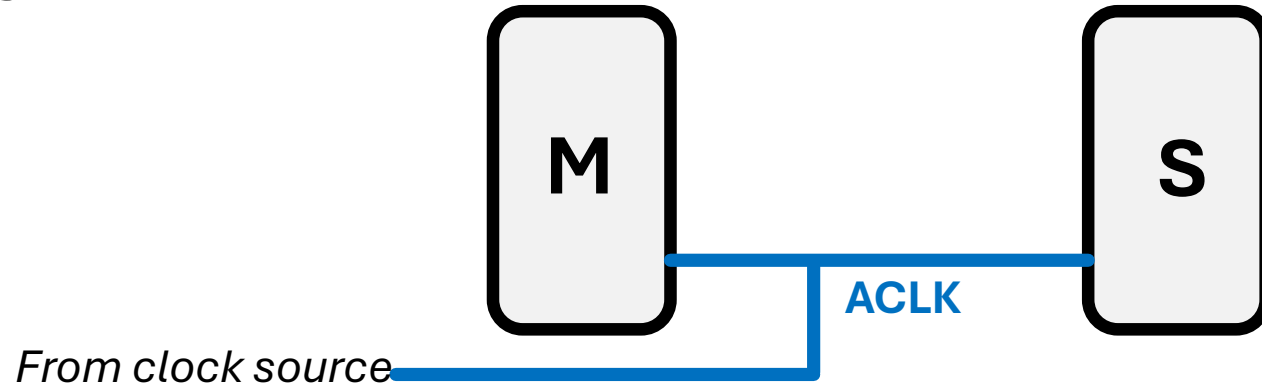
Write Wiring



Within Each Channel are wires:

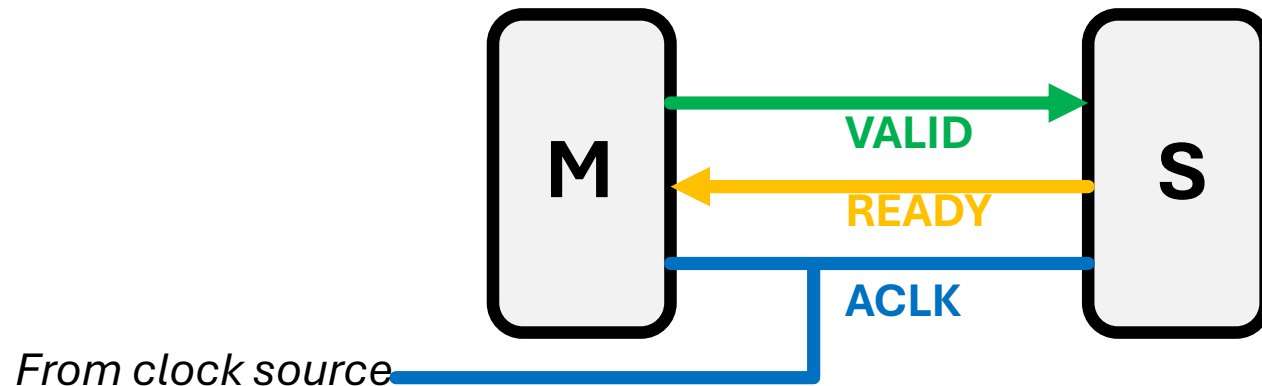
- These wires serve specific purposes.
- Some are universal to all channels, and others are specific

AXI Clock



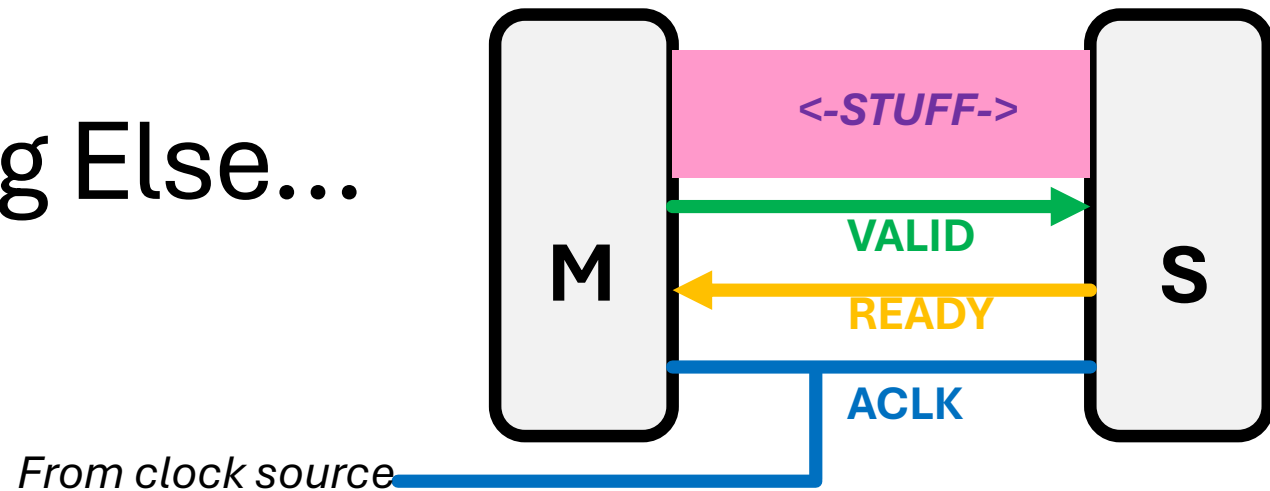
- Everything in system will run off of AXI clock usually called **ACLK** in documentation
- No combinatorial paths between inputs and outputs. Everything must be registered.
- All signals are sampled **on rising edge**
- AXI modules should also have Reset pins. AXI work ACTIVE LOW so the Reset pin is usually called **ARSTn** or **ARESETn**

Valid and Ready



- All of AXI uses the same handshake procedure:
- The source of a data generates a **VALID** signal
- The destination generates a **READY** signal
- Transfer of data only occurs when both are high
- Both Master and Slave Devices can therefore control the flow of their data as needed

Everything Else...



- Everything else is information and depends on what is needed in situation. Could be:
 - Address
 - Data
 - Other specialized wires like:
 - STRB (used to specify which bytes in current data step are valid, sent by Master along with data payload to Slave)
 - RESP (sort of like a status)
 - LAST (sent to indicate the final data clock cycle of data in a burst)

Each channel has its own subset of “stuff” that goes along with those core signals shared by all

For example, the Write Data Channel (“W” channel)

Payload

Signal	Source	Description
WID	Master	Write ID tag. This signal is the ID tag of the write data transfer. Supported only in AXI3. See Transaction ID on page A5-77 .
WDATA	Master	Write data.
WSTRB	Master	Write strobes. This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits of the write data bus. See Write strobes on page A3-49 .
WLAST	Master	Write last. This signal indicates the last transfer in a write burst. See Write data channel on page A3-39 .
WUSER	Master	User signal. Optional User-defined signal in the write data channel. Supported only in AXI4. See User-defined signaling on page A8-100 .

CORE

WVALID	Master	Write valid. This signal indicates that valid write data and strobes are available. See Channel handshake signals on page A3-38 .
WREADY	Slave	Write ready. This signal indicates that the slave can accept the write data. See Channel handshake signals on page A3-38 .

Supplemental Stuff

The Read Data Channel:

Table A2-6 Read data channel signals

Signal	Source	Description
RID	Slave	Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave. See Transaction ID on page A5-77 .
RDATA	Slave	Read data.
RRESP	Slave	Read response. This signal indicates the status of the read transfer. See Read and write response structure on page A3-54 .
RLAST	Slave	Read last. This signal indicates the last transfer in a read burst. See Read data channel on page A3-39 .
RUSER	Slave	User signal. Optional User-defined signal in the read data channel. Supported only in AXI4. See User-defined signaling on page A8-100 .
RVALID	Slave	Read valid. This signal indicates that the channel is signaling the required read data. See Channel handshake signals on page A3-38 .
RREADY	Master	Read ready. This signal indicates that the master can accept the read data and response information. See Channel handshake signals on page A3-38 .

Payload

CORE

Supplemental Stuff

Read Address Chanel

Table A2-5 Read address channel signals

Payload

Signal	Source	Description
ARID	Master	Read address ID. This signal is the identification tag for the read address group of signals. See Transaction ID on page A5-77 .
ARADDR	Master	Read address. The read address gives the address of the first transfer in a read burst transaction. See Address structure on page A3-44 .
ARLEN	Master	Burst length. This signal indicates the exact number of transfers in a burst. This changes between AXI3 and AXI4. See Burst length on page A3-44 .
ARSIZE	Master	Burst size. This signal indicates the size of each transfer in the burst. See Burst size on page A3-45 .
ARBURST	Master	Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated. See Burst type on page A3-45 .
ARLOCK	Master	Lock type. This signal provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4. See Locked accesses on page A7-95 .
ARCACHE	Master	Memory type. This signal indicates how transactions are required to progress through a system. See Memory types on page A4-65 .
ARPROT	Master	Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. See Access permissions on page A4-71 .
ARQOS	Master	<i>Quality of Service</i> , QoS. QoS identifier sent for each read transaction. Implemented only in AXI4. See QoS signaling on page A8-98 .
ARREGION	Master	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4. See Multiple region signaling on page A8-99 .
ARUSER	Master	User signal. Optional User-defined signal in the read address channel. Supported only in AXI4. See User defined signaling on page A8-100 .
ARVALID	Master	Read address valid. This signal indicates that the channel is signaling valid read address and control information. See Channel handshake signals on page A3-38 .
ARREADY	Slave	Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals. See Channel handshake signals on page A3-38 .

CORE

Write Response

Table A2-4 Write response channel signals

	Signal	Source	Description
	BID	Slave	Response ID tag. This signal is the ID tag of the write response. See <i>Transaction ID</i> on page A5-77.
Payload	BRESP	Slave	Write response. This signal indicates the status of the write transaction. See <i>Read and write response structure</i> on page A3-54.
	BUSER	Slave	User signal. Optional User-defined signal in the write response channel. Supported only in AXI4. See <i>User-defined signaling</i> on page A8-100.
CORE	BVALID	Slave	Write response valid. This signal indicates that the channel is signaling a valid write response. See <i>Channel handshake signals</i> on page A3-38.
	BREADY	Master	Response ready. This signal indicates that the master can accept a write response. See <i>Channel handshake signals</i> on page A3-38.

Write Address Channel

Table A2-2 Write address channel signals

Payload

Signal	Source	Description
AWID	Master	Write address ID. This signal is the identification tag for the write address group of signals. See Transaction ID on page A5-77 .
AWADDR	Master	Write address. The write address gives the address of the first transfer in a write burst transaction. See Address structure on page A3-44 .
AWLEN	Master	Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address. This changes between AXI3 and AXI4. See Burst length on page A3-44 .
AWSIZE	Master	Burst size. This signal indicates the size of each transfer in the burst. See Burst size on page A3-45 .
AWBURST	Master	Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated. See Burst type on page A3-45 .
AWLOCK	Master	Lock type. Provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4. See Locked accesses on page A7-95 .
AWCACHE	Master	Memory type. This signal indicates how transactions are required to progress through a system. See Memory types on page A4-65 .
AWPROT	Master	Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. See Access permissions on page A4-71 .
AWQOS	Master	<i>Quality of Service</i> , QoS. The QoS identifier sent for each write transaction. Implemented only in AXI4. See QoS signaling on page A8-98 .
AWREGION	Master	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4. See Multiple region signaling on page A8-99 .
AWUSER	Master	User signal. Optional User-defined signal in the write address channel. Supported only in AXI4. See User-defined signaling on page A8-100 .

CORE

AWVALID	Master	Write address valid. This signal indicates that the channel is signaling valid write address and control information. See Channel handshake signals on page A3-38 .
AWREADY	Slave	Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals. See Channel handshake signals on page A3-38 .

Generalized Transaction

- All Channel Interactions follow same high-level structure

Sending One "beat" of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1's and 0's of info or 8 bytes for example...
Or it could be something else

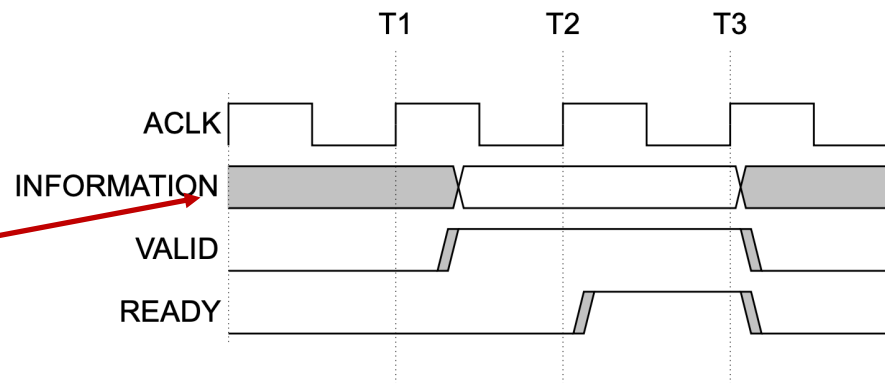


Figure A3-2 VALID before READY handshake

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Generalized Transaction

- All Channel Interactions follow same high-level structure

Sending One "beat" of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1's and 0's of info or 8 bytes for example...
Or it could be something else

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

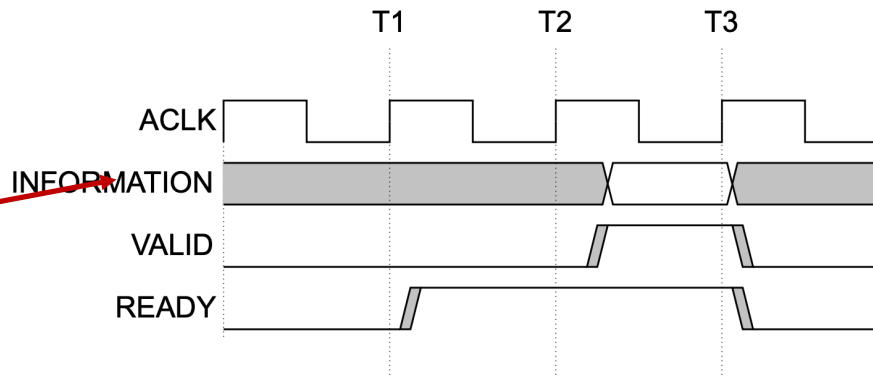


Figure A3-3 READY before VALID handshake

Generalized Transaction

- All Channel Interactions follow same high-level structure

Sending One "beat" of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1's and 0's of info or 8 bytes for example...
Or it could be something else

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

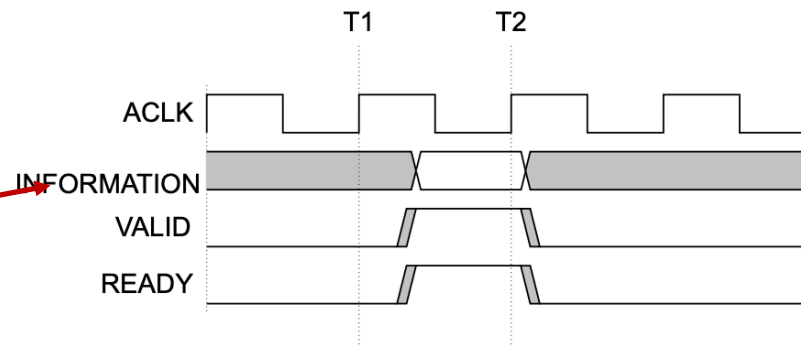


Figure A3-4 VALID with READY handshake

Other Things to Keep in Mind

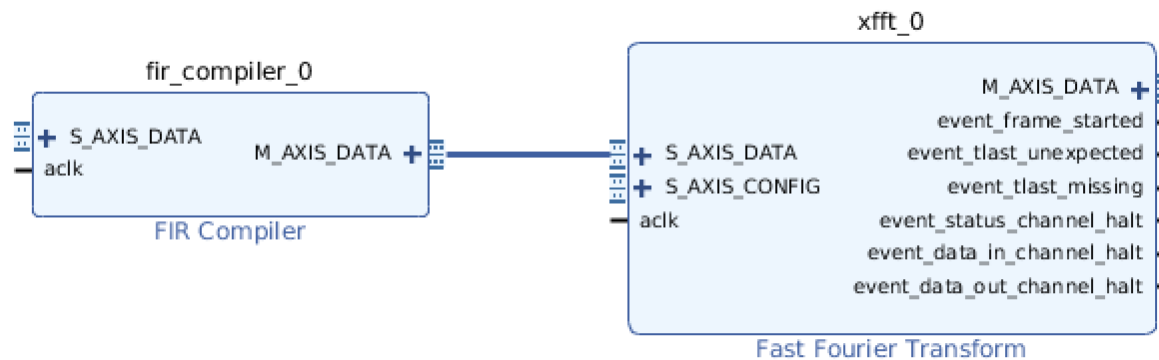
- the VALID signal of the AXI interface sending information must not be dependent on the READY signal of the AXI interface receiving that information
- an AXI interface that is receiving information can wait until it detects a VALID signal before it asserts its corresponding READY signal.
- Fail to Follow these rules and could have devices wait infinitely.
 - Like when two people keep going “no, after you at a door”

And Up to All Five AXI channels can come from one device

- While operating independently at their individual transaction level, they can then report to the larger module to provide overall interfaces
- Example:
 - The slave device receives address on write channel address
 - The write data channel then becomes active and knows where to point incoming data
 - The response channel then opens and does its thing
 - And so on
- Hierarchy of Control/Design

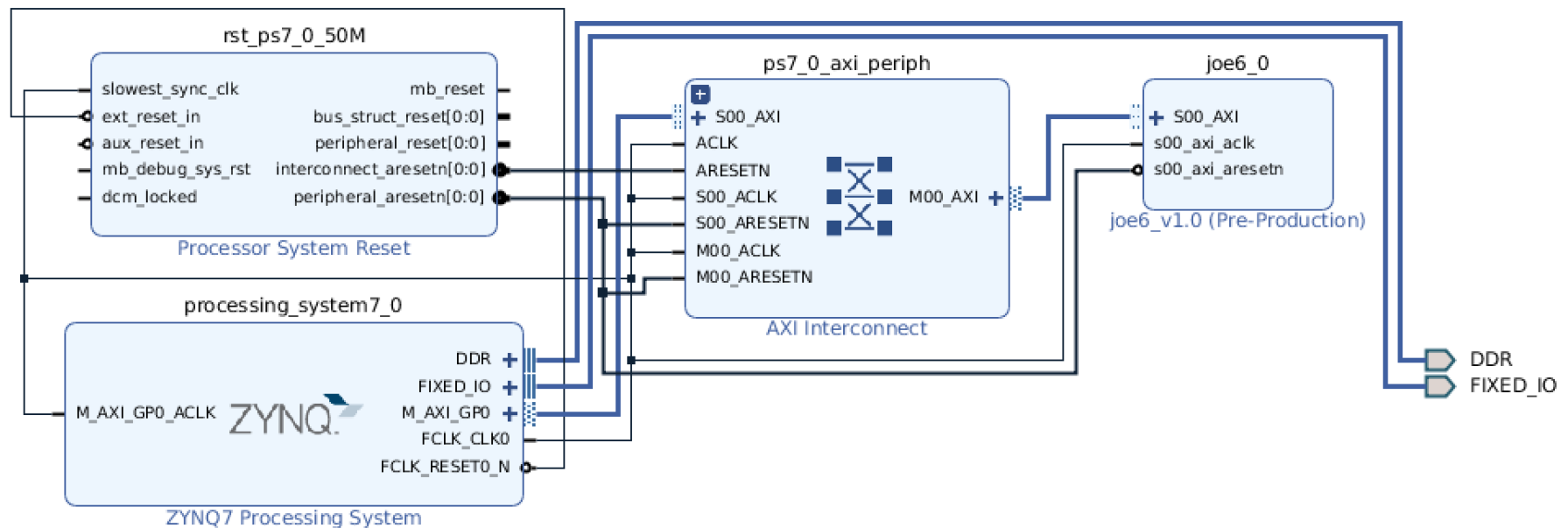
And you Can Use AXI to Interface with Tons of things!

Connecting a FIR (from a Xilinx IP) to the FFT module



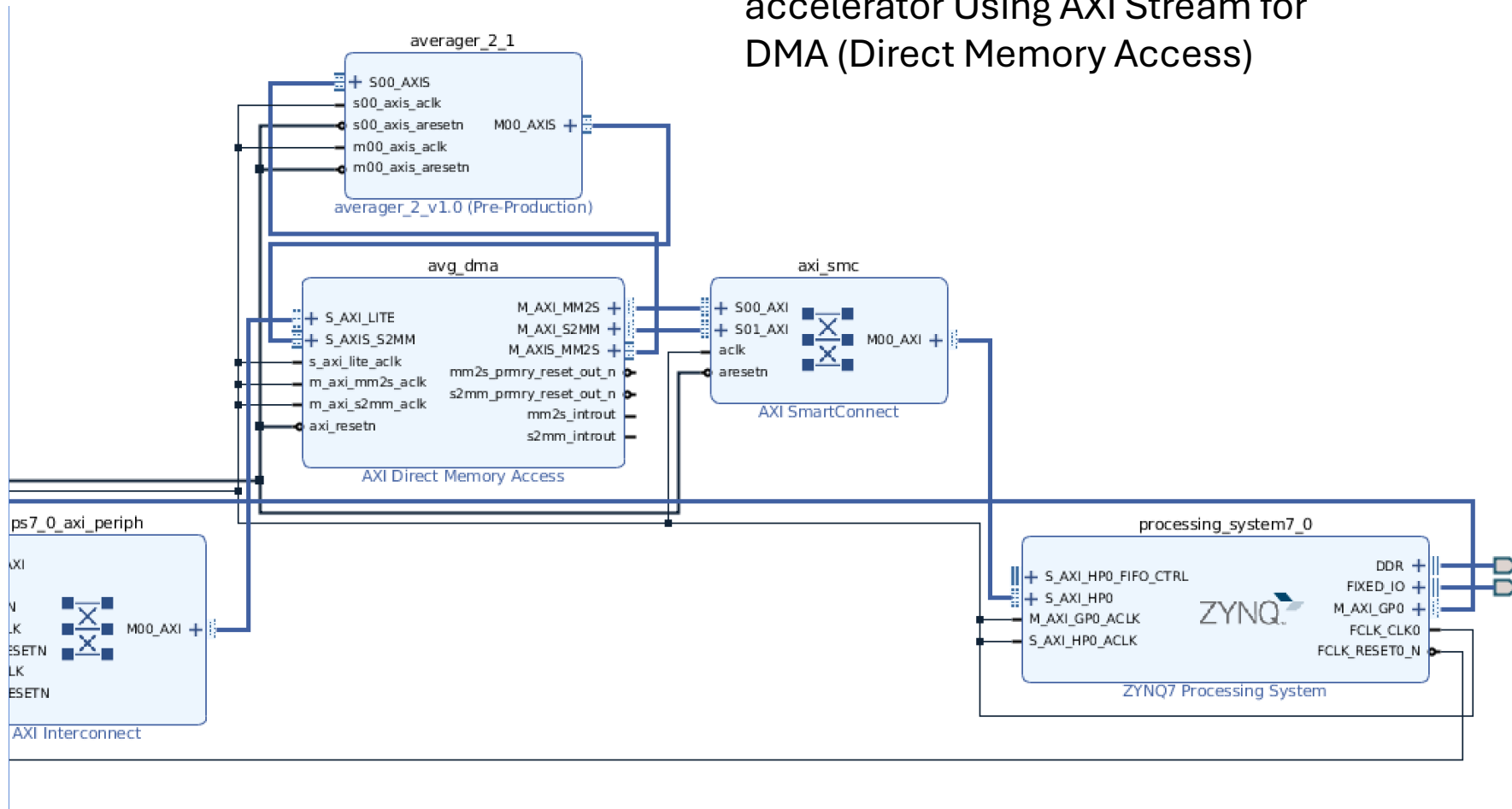
And you Can Use AXI to Interface with Tons of things!

Creating a AXI-controlled joe6 module that I can then call from Python



And you Can Use AXI to Interface with Tons of things!

A running-average hardware accelerator Using AXI Stream for DMA (Direct Memory Access)



The AXI Interfaces on the Zynq Enable PS to PL communication effectively

Interface Name	Interface Description	Master	Slave
M_AXI_GP0	General Purpose (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	General Purpose (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), cache coherent transaction	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs).	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Master/Slave refers to who controls/initiates comms on that bus that bus

From Zynq Book

General Purpose/Performance “GP” AXI Ports

- 32 bits in size
- Maximum flexibility
- Allow register access from:
 - PS to PL
 - PL to PS

High Performance “HP” AXI Ports

- Can be 32 or 64 bits wide (or variable between, but avoid)
- Maximum bandwidth access to external memory and on-chip-memory (OCM)
- When use all four HP ports at 64 bits, you can outpace ability to write to DDR and OCM bandwidths!
 - HP Ports : $4 * 64 \text{ bits} * 150 \text{ MHz} * 2 = \mathbf{9.6 \text{ GByte/sec}}$
 - external DDR: $1 * 32 \text{ bits} * 1066 \text{ MHz} * 2 = \mathbf{4.3 \text{ GByte/sec}}$
 - OCM : $64 \text{ bits} * 222 \text{ MHz} * 2 = \mathbf{3.5 \text{ GByte/sec}}$
- Optimized for large burst lengths

Taken from ECE699 lec 6 notes gm.edu

How it is Laid Out

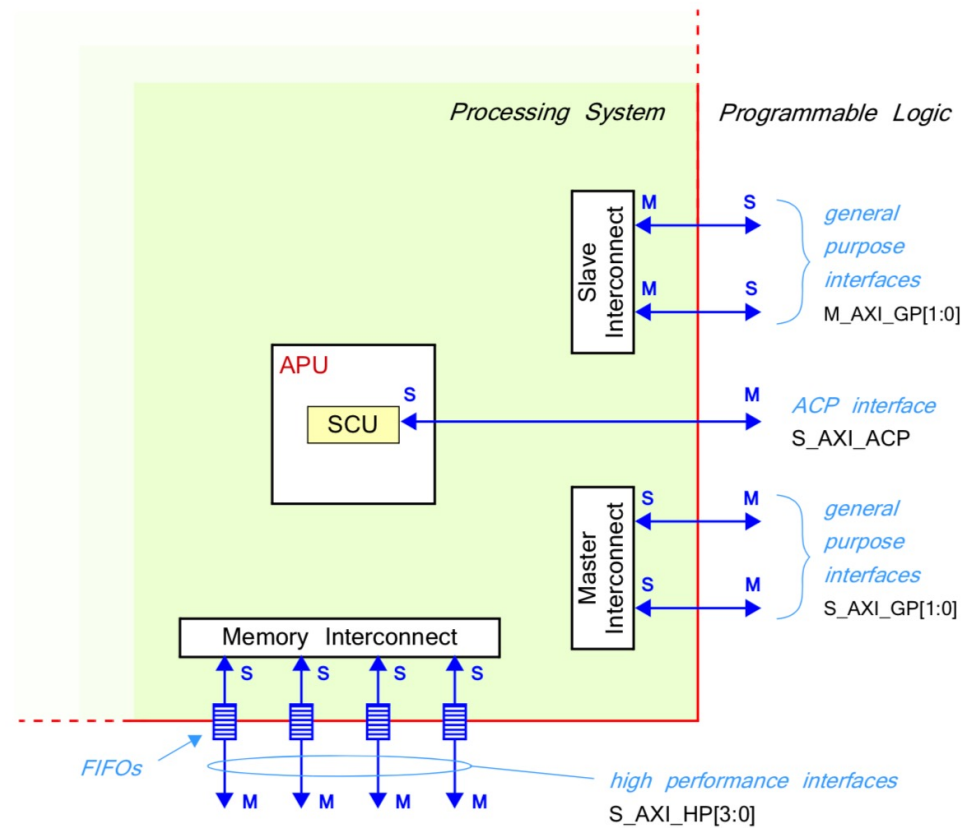


Figure 2.9: The structure of AXI interconnects and interfaces connecting the PS and PL

From The Zynq Book

Complexity

- In terms of wires and options, Full-AXI is the most complex
- AXI-LITE has a lot less options (single data beat so all the supplemental stuff that specifies burst characteristics gets skipped)
- AXI-STREAM has even less...basically a high-speed write channel (Few options), but often needs that extra TLAST signal

Full-AXI4



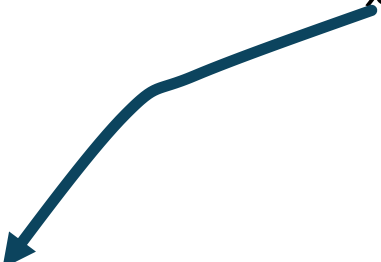
AXI-LITE



AXI-STREAM

Sources

This is the thing right here...the spec sheet/manual is surprisingly good!!



- **“AMBA® AXITM and ACETM Protocol Specification”, ARM 2011**
- **“The Zynq Book”, L.H. Crockett, R.A. Elliot, M.A. Enderwitz, and R.W. Stewart, University of Glasgow**
- **“Building Zynq Accelerators with Vivado High Level Synthesis”
Xilinx Technical Note**
- **Some material from ECE699 Spring 2016**
https://ece.gmu.edu/coursewebpages/ECE/ECE699_SW_HW/S16/

Crack open the AXI spec sheet with a few data sheets for some Xilinx IP cores (like the CORDIC, FFT, etc...) and you should be able to start making sense of it.