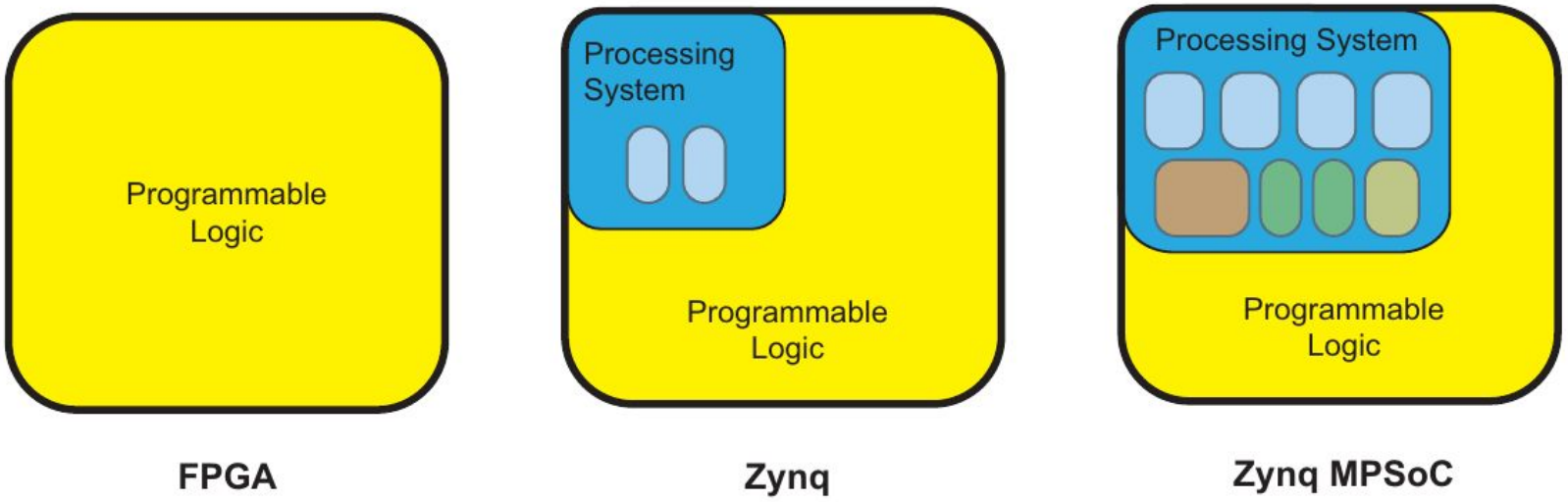
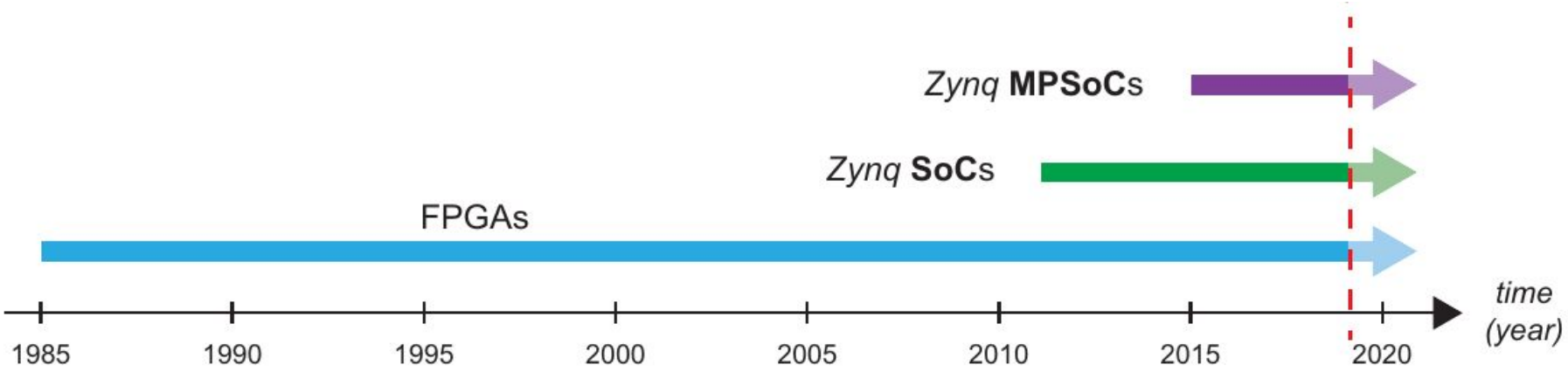
The Zynq and PYNQ

Manuel Rios
https://github.com/mriosrivas/pynq.git

High level comparison of FPGAs, Zynq, and Zynq MPSoC

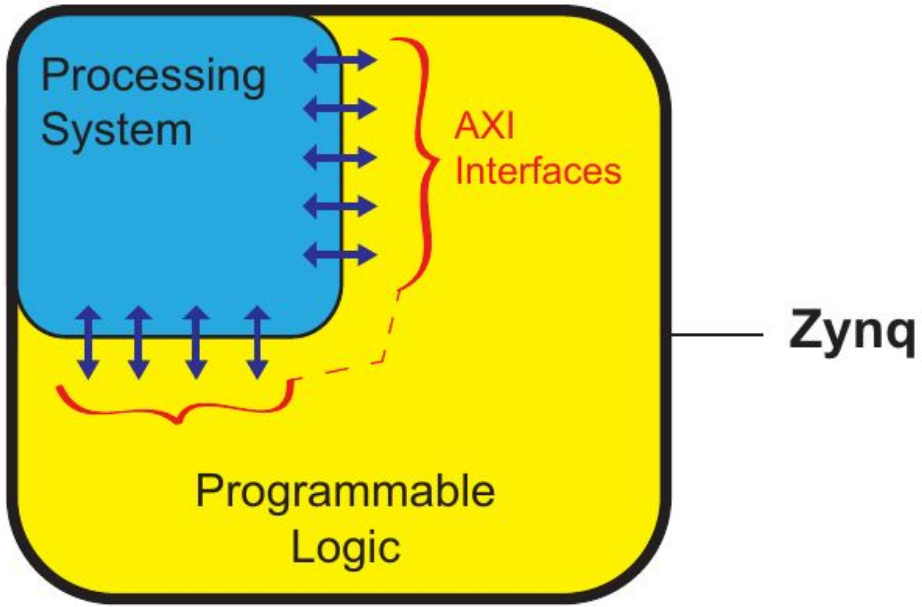


Timeline of FPGA, Zynq and Zynq MPSoC introductions



The Zynq-7000 SoC

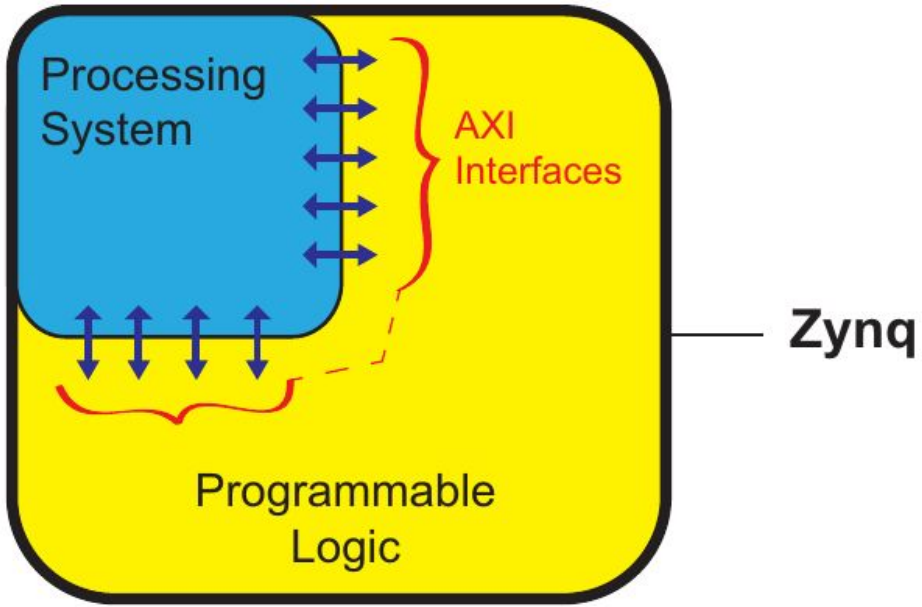
The Zynq-7000 SoC was the first SoC device released by Xilinx, combining FPGA-based PL with an Arm-based PS.



The Zynq-7000 SoC

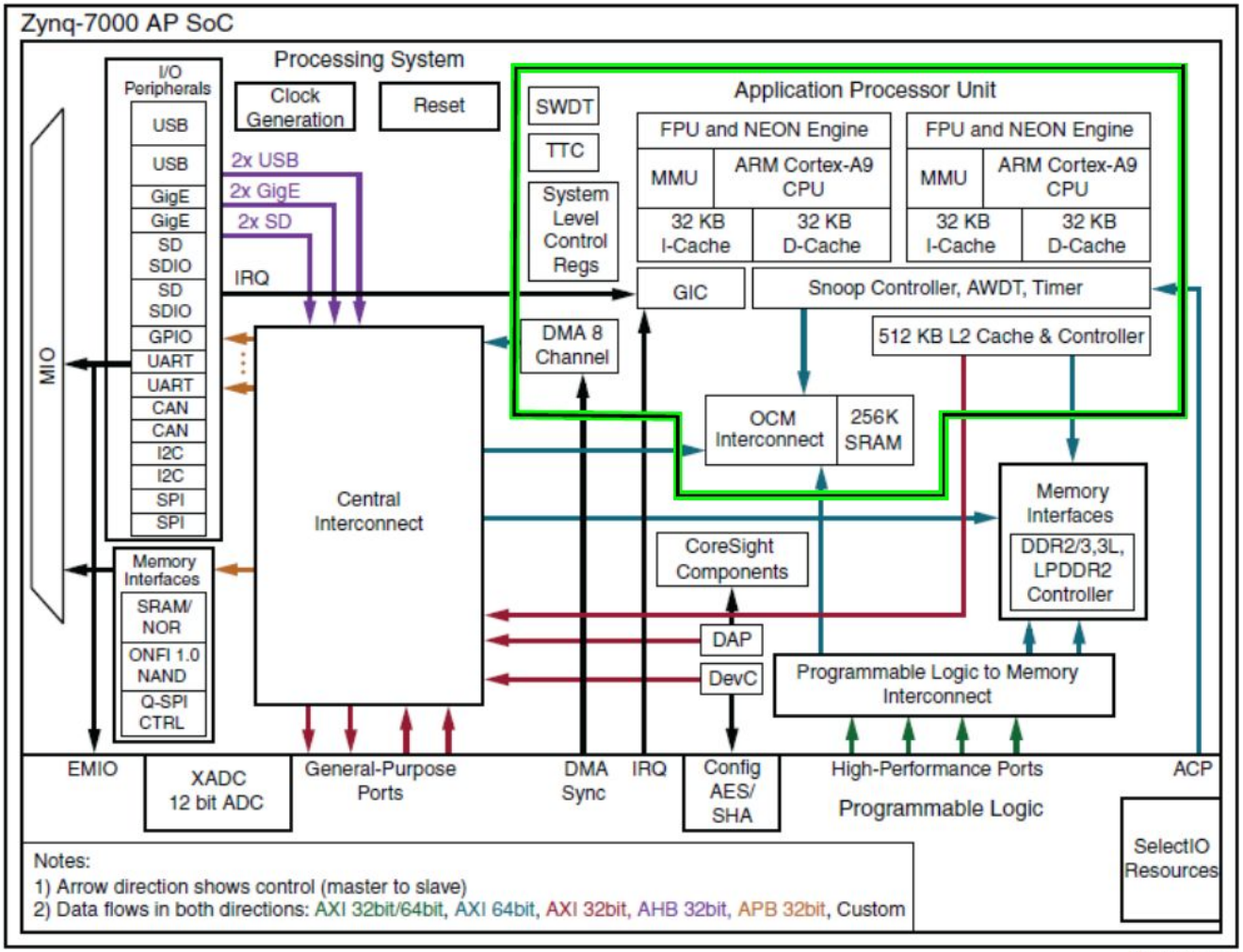
Has two parts: the PS and the PL, with a set of interconnections between them.

The interconnections are based on the Advanced eXtensible Interface (AXI) standard, an on-chip communications standard developed by Arm.



Zynq Processing System

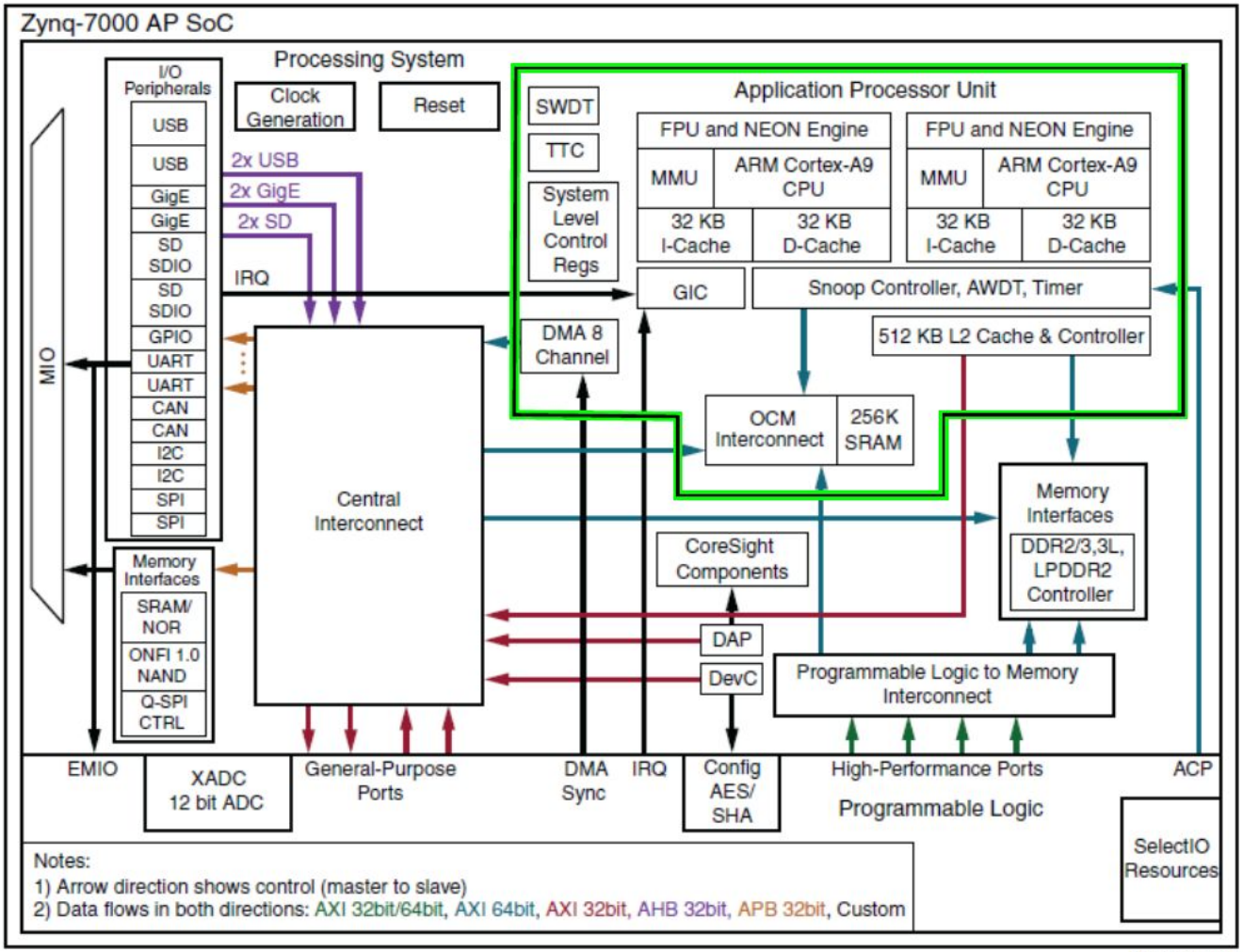
Application Processor Unit (APU)



DS190_01_030713

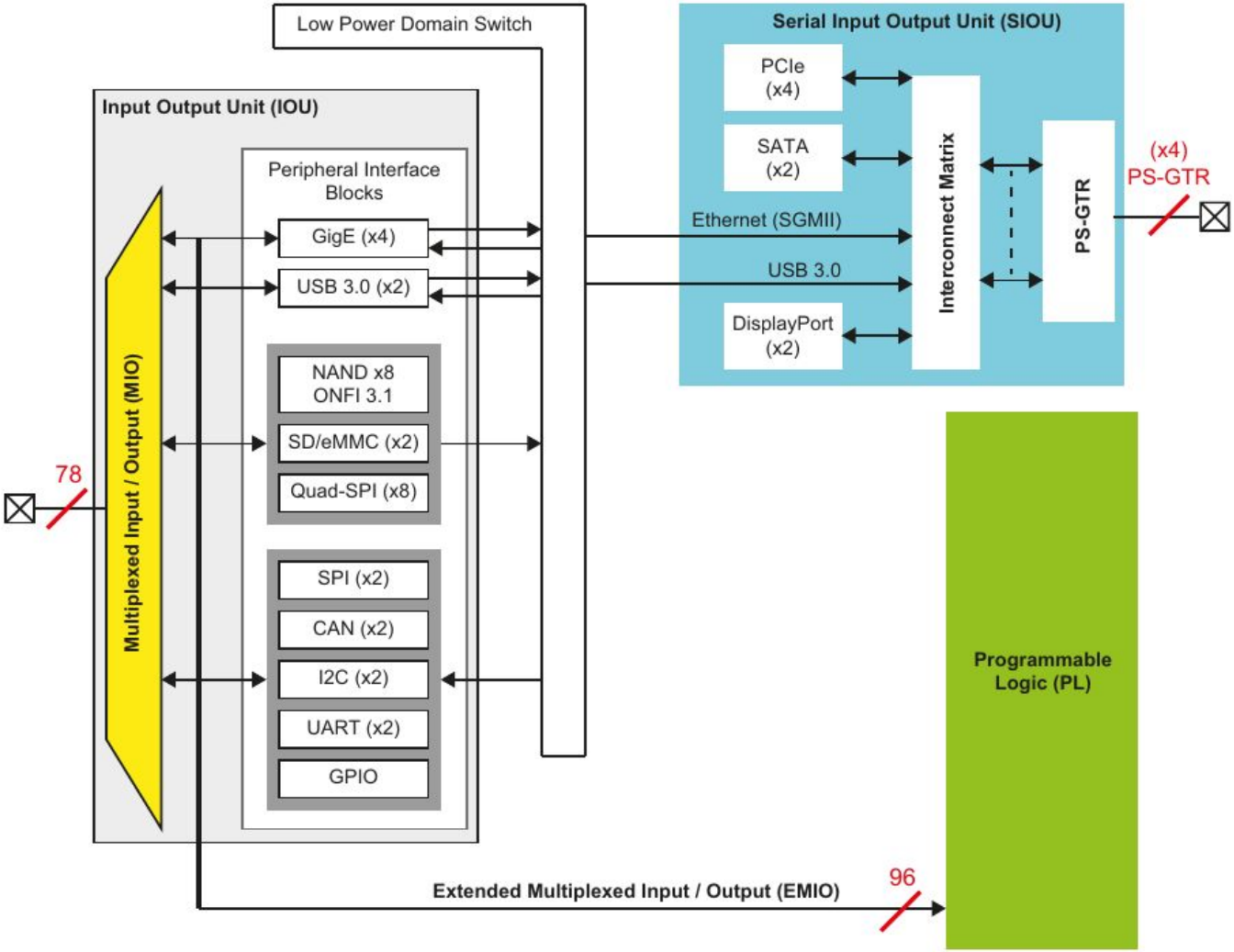
Zynq Processing System

Interconnects,Memory Interfaces and I/O Peripherals



Zynq Processing System

Block diagram of the MIO, EMIO, and SIOU connectivity in the Zynq MPSoC



Zynq Processing System

List of I/O Peripheral Interfaces

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

AXI Interfacing

AXI stands for **Advanced eXtensible Interface**, and the current version is AXI4, which is part of the ARM AMBA® 3.0 open standard.

Released in 1996, and now is described by ARM as “the de facto standard for on-chip communication”.

The AXI protocol in particular exhibits the following key features:

- address/control phases are separate from data phases
- byte strobes enable unaligned data transfers
- burst-based transactions possible with only start address issued
- read and write data channels are separate allowing low-cost Direct Memory Access(DMA)

Variations of AXI4

AXI4-Full:

The **high-performance interface**, suited for **memory mapped communication** allowing **bursts** of up to 256 data transfer cycles per address phase.

Variations of AXI4

AXI4-Lite:

A **light-weight** variant of the interface, used for **memory mapped single transactions**. This variant has the benefit of a smaller logic footprint with a **simplified interface**. This variation **does not support burst data** and only provides a **single data transfer** per transaction.

Variations of AXI4

AXI4-Stream:

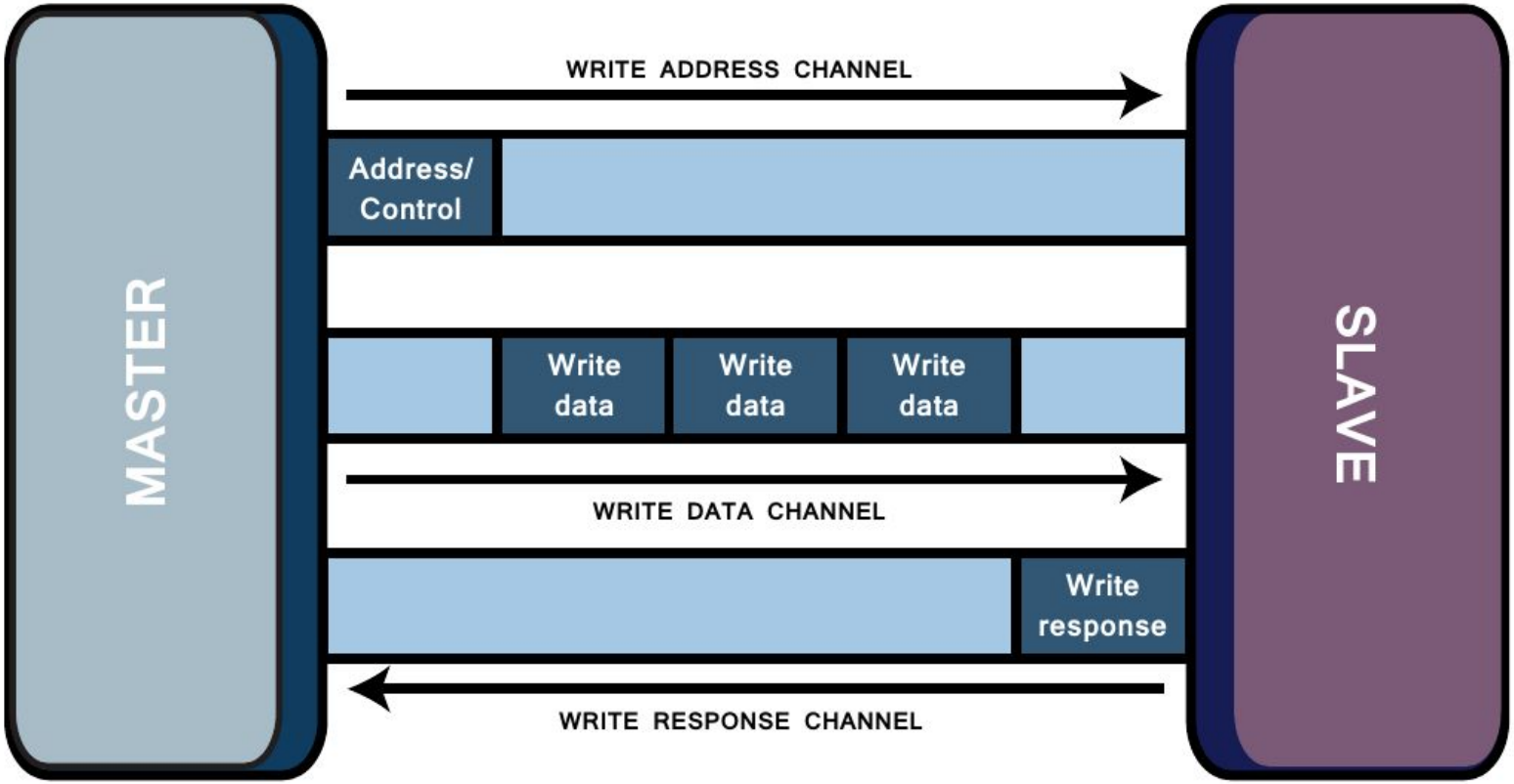
Allows for an **unlimited data burst size** since it is **not memory mapped** and a single channel is defined for transmission. Connection is from master to slave only, so if bidirectional transfers are required both peripherals must be of type master/slave.

AXI Architecture

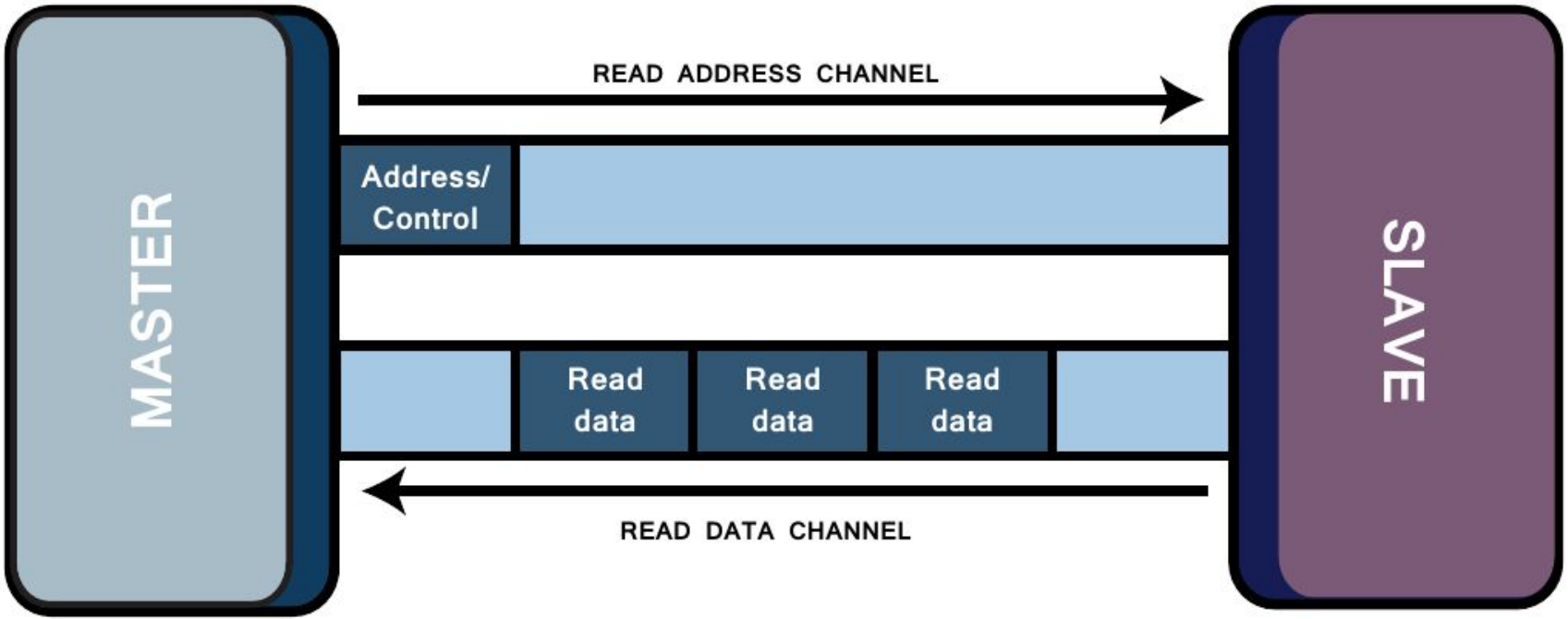
The AXI protocol features **burst-based transactions**, with each containing **address and control information** on the address channel.

Several **AXI masters** can be connected to several **AXI slaves** through an **AXI interconnect**.

AXI Architecture - AXI4 write channel



AXI Architecture - AXI4 read channel



Applications of Various AXI Interfaces

Interface	Industry	Example Applications
AXI4	Audio and Video/ Image Processing	Video Direct Memory Access
	Communications/ Networking	Ethernet VOIP Receiver, 3GPP LTE Channel Decoder
	Embedded Process-ing	AXI/PLBV46 Bridges (for backwards compatibil-ity), ChipScope AXI Monitor (for debugging/ embedded system diagnostics)

Applications of Various AXI Interfaces

Interface	Industry	Example Applications
AXI4-Lite	Audio and Video/ Image Processing	Deinterlacer, Gamma Correction, Image Edge Enhancement
	Automotive	Controller Area Network (CAN)
	Communications/ Networking	10 Gigabit Ethernet Media Access Controller, Digital Predistortion (DPD), Crest Factor Reduction (CFR)
	Embedded Processing	Hardware ICAP, BRAM Interface Controller, External Peripheral Controller

Applications of Various AXI Interfaces

Interface	Industry	Example Applications
AXI4-Stream	Audio and Video/ Image Processing	Streaming video input/output, image noise reduction
	Communications/ Networking	Encoders/decoders, interleavers/deinterleavers
	DSP	CORDIC, FFT, FIR Compiler
	Embedded Processing	Streaming FIFO, Ethernet peripheral,

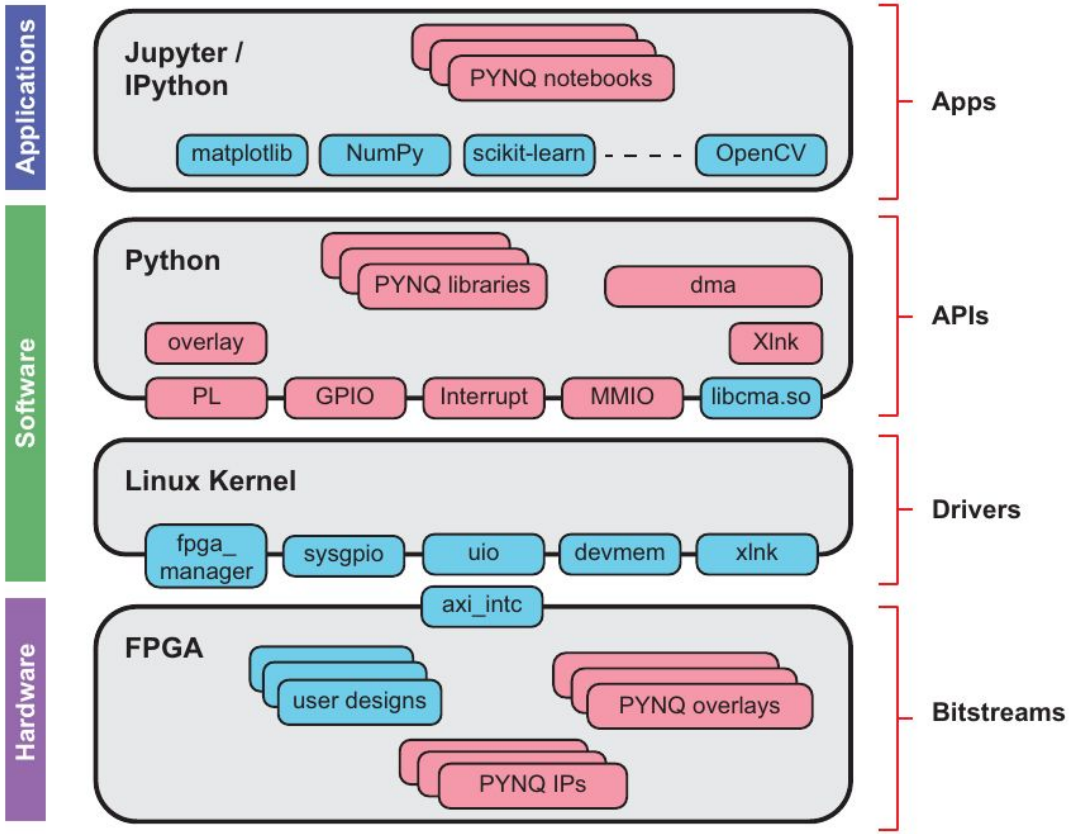
PYNQ (Python productivity for Zynq)

Open source framework for creating and using applications with Zynq and Zynq MPSoC devices.

The PYNQ framework **complements** the mainstream **Xilinx tools**, rather than replacing them.

PYNQ **accelerates the software/hardware co-design** aspects of Zynq-based embedded systems, and makes the task of interfacing between the PS and PL easier.

PYNQ Framework

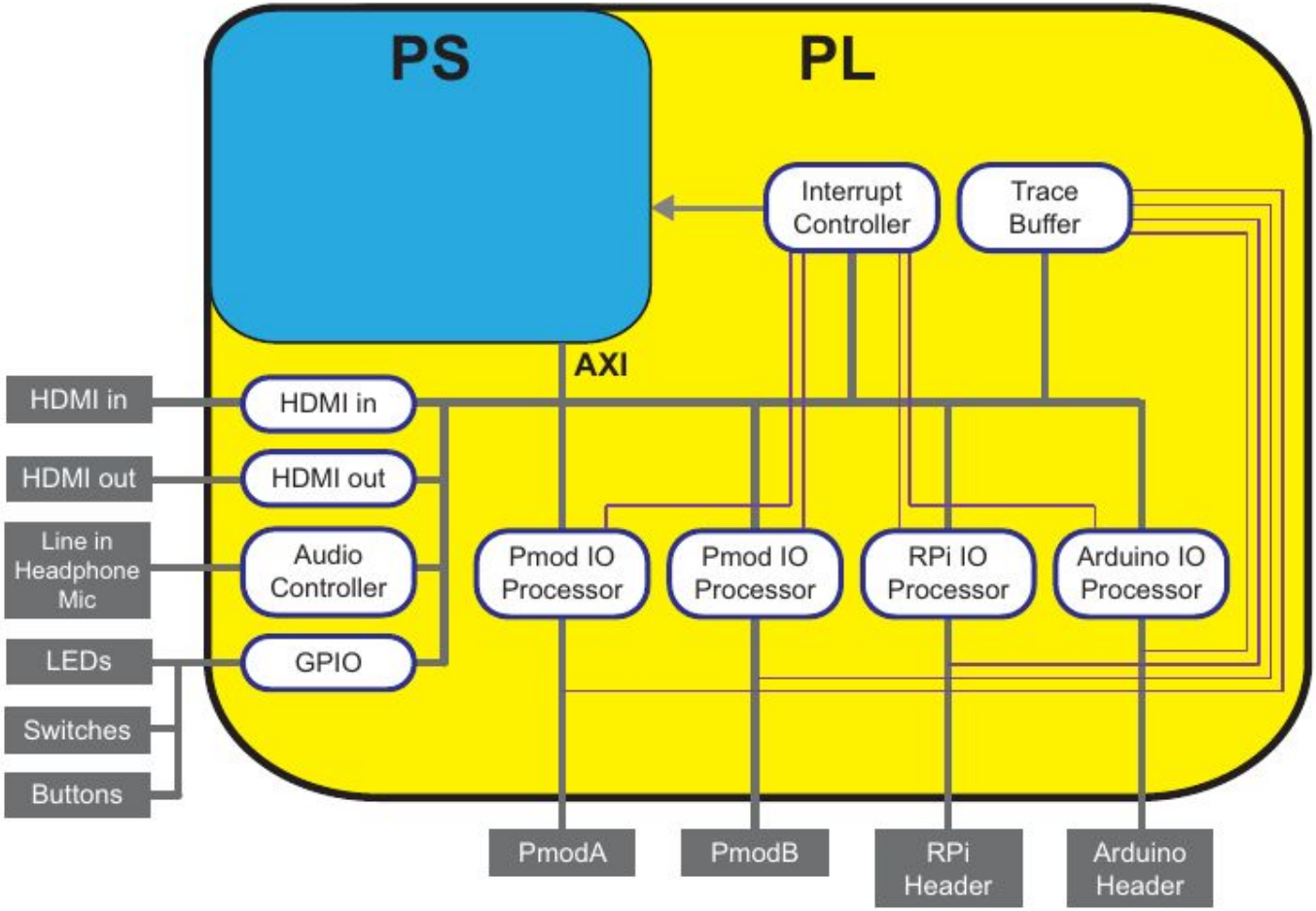


Overlays

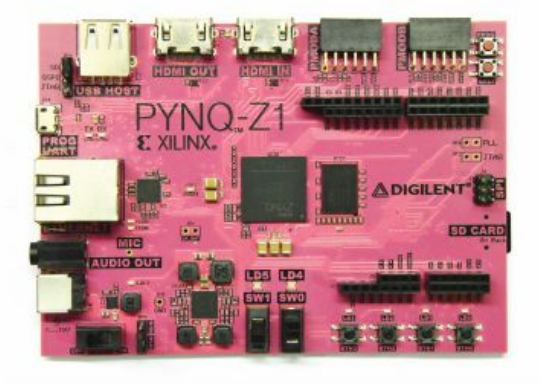
An overlay is a **complete hardware system** that will be programmed onto the PL, and it represents part of the hardware layer of the PYNQ framework.

The **uniqueness** of overlays (as opposed to regular bitstreams) is the ability to **interact with** these hardware designs from **Python code**, running in a Jupyter notebook on the PS.

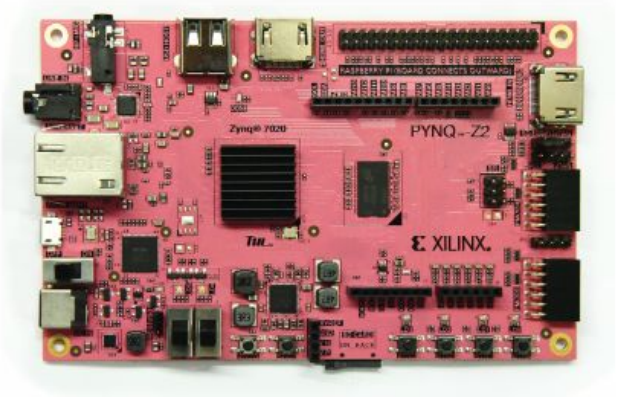
Overlays:
Base Overlay



Zynq boards



PYNQ-Z1



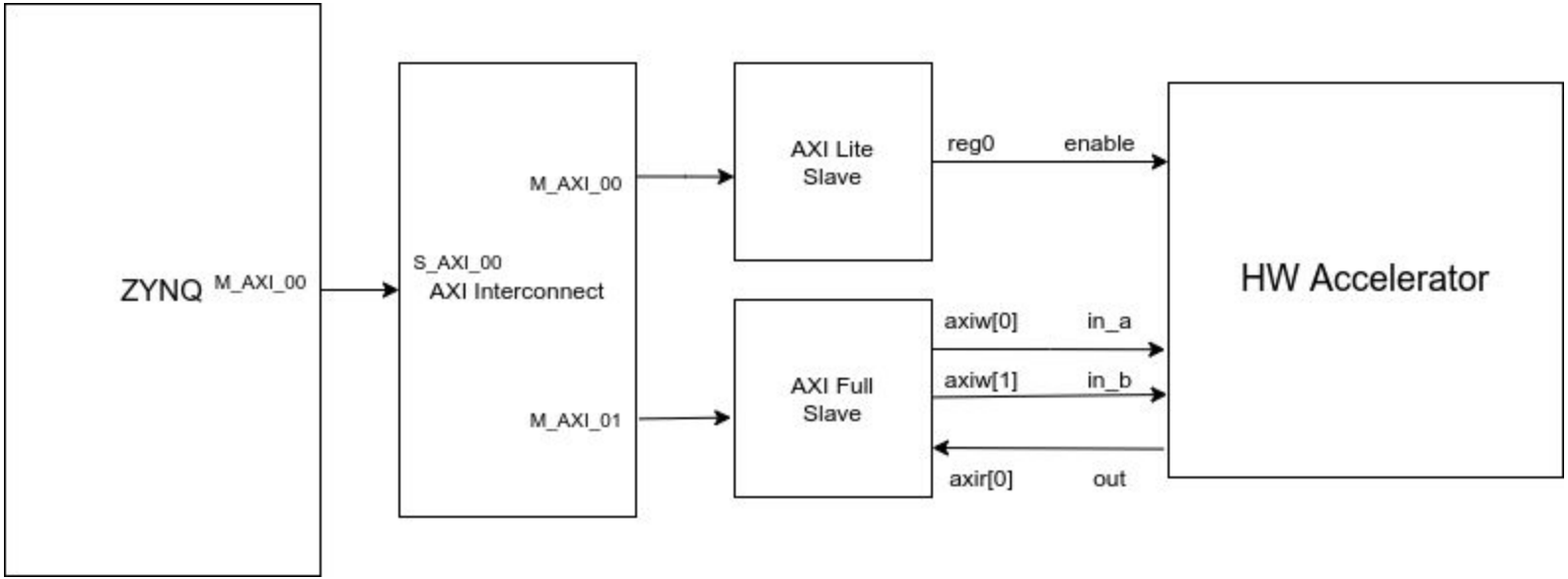
PYNQ-Z2

Zynq MPSoC boards

Ultra96

ZCU104

Architecture of an AXI Interface



How to Deploy an Accelerator with an AXI Interface

1. Design an accelerator in a HDL (Verilog/VHDL)
2. Package your design using Vivado
3. Design your Zynq system and add your own package
4. Create a HDL wrapper for your design
5. Generate bitstream

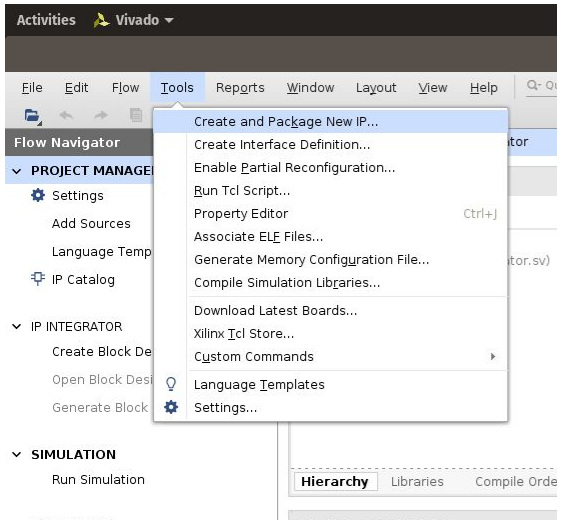
How to Deploy an Accelerator with an AXI Interface

1. Design an accelerator in a HDL (Verilog/VHDL)

```
module accelerator(
    input logic clk, reset, enable,
    input logic [WIDTH-1 : 0] in_a, in_b,
    output logic [WIDTH-1 : 0] out);

    parameter WIDTH = 32;

    logic [2*WIDTH-1 : 0] temp;

    always_ff@(posedge clk)
        begin
        if(~reset) temp <= 0;
        else if(reset && enable) temp <= in_a * in_b;
        end

    assign out = temp[WIDTH-1 : 0];

endmodule
```

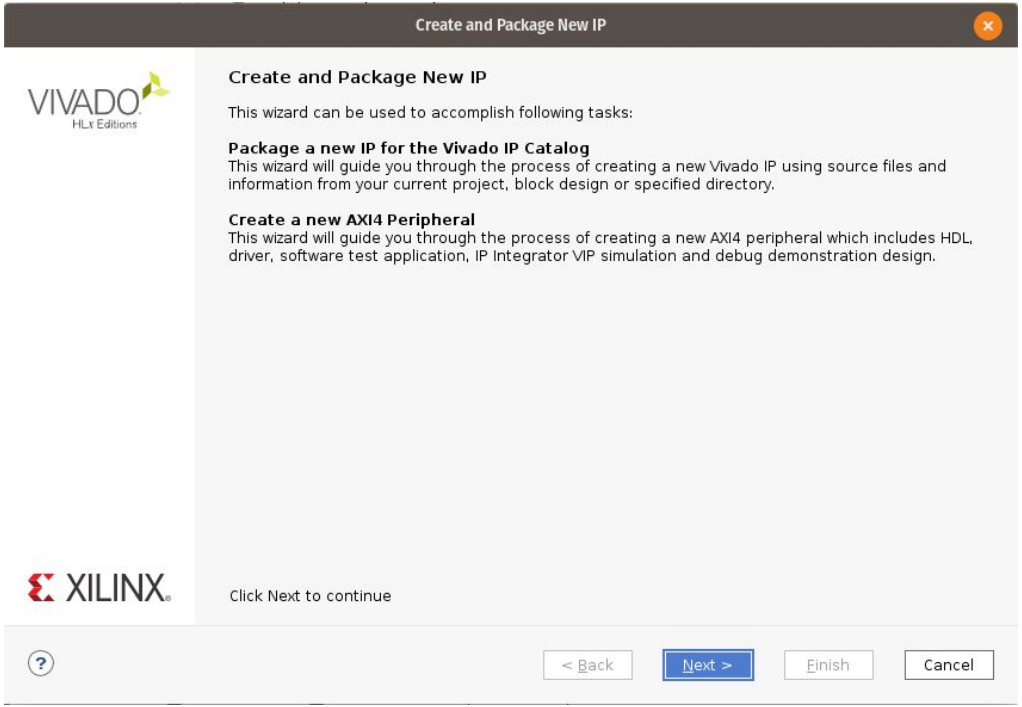
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



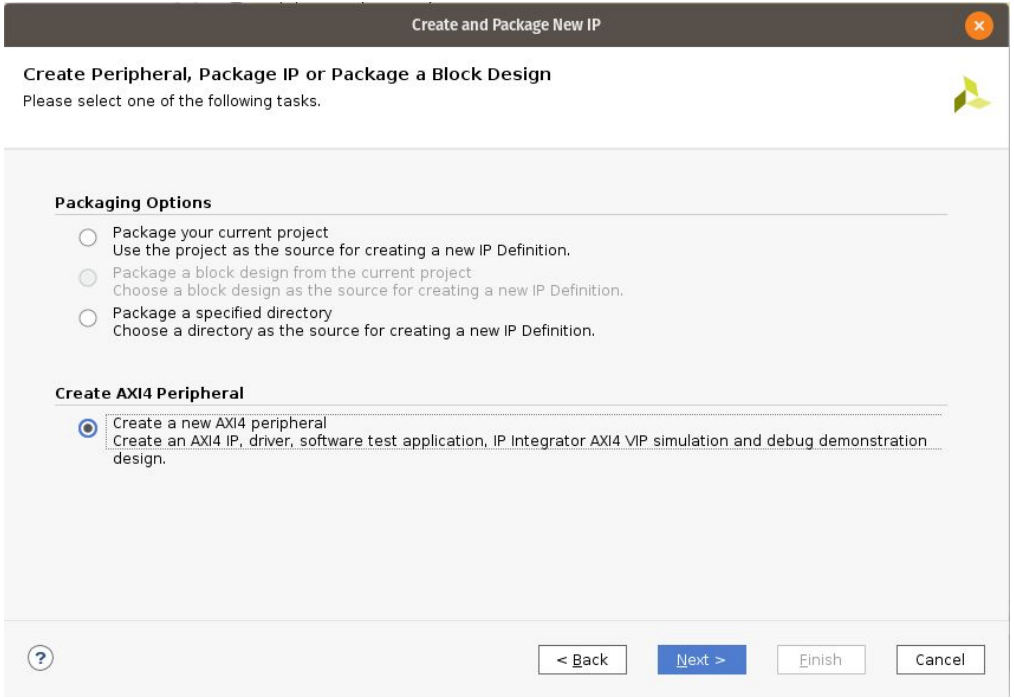
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



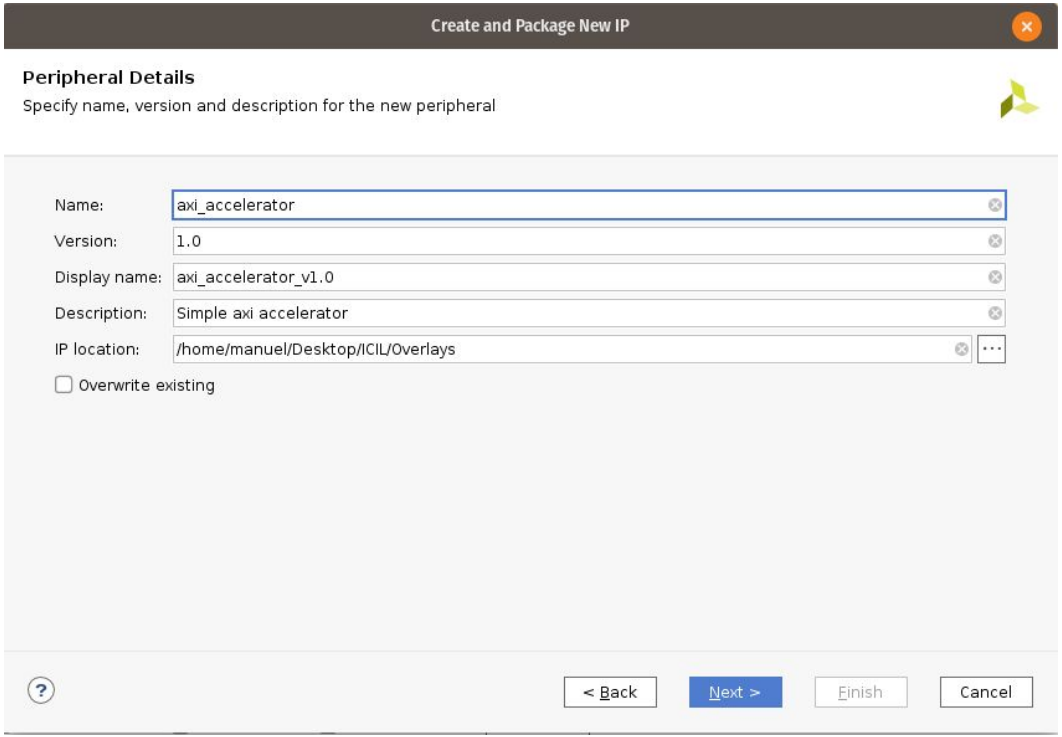
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



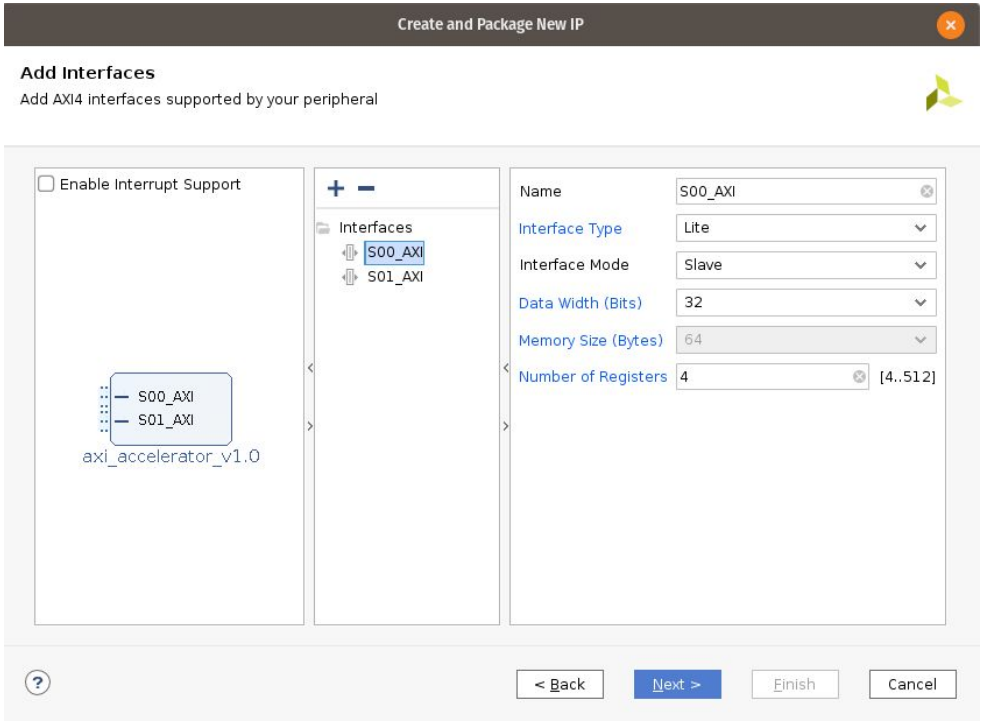
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



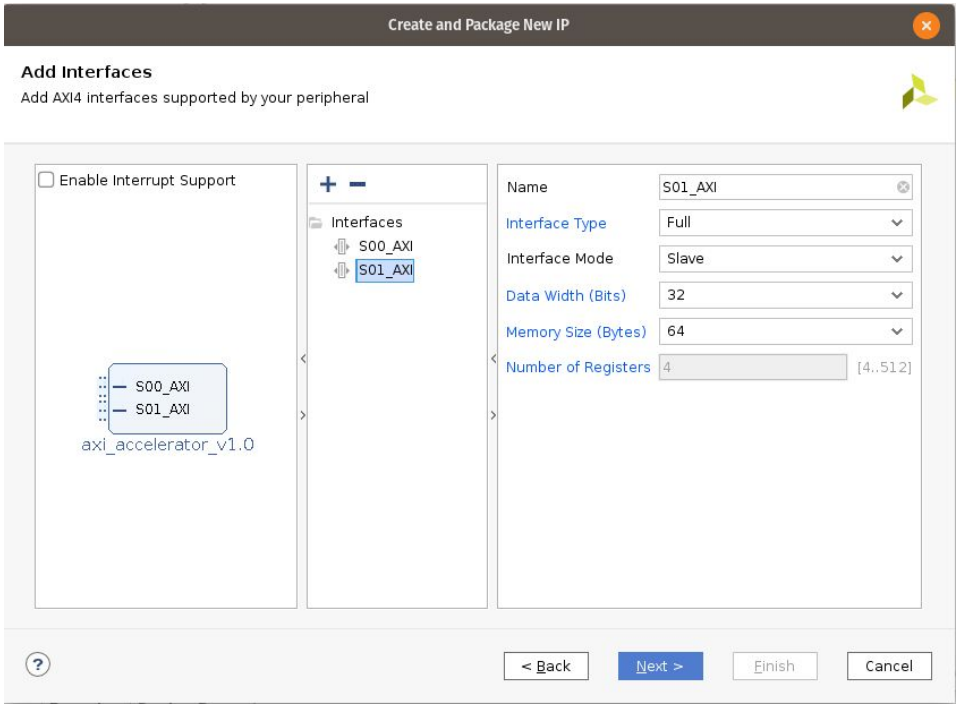
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



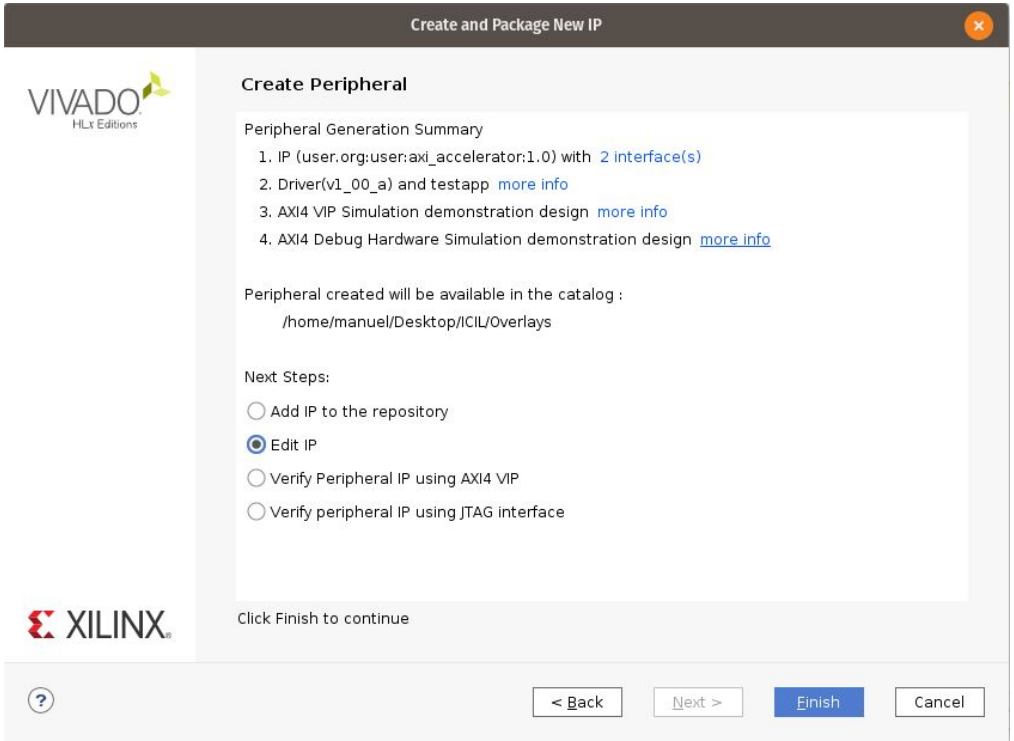
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



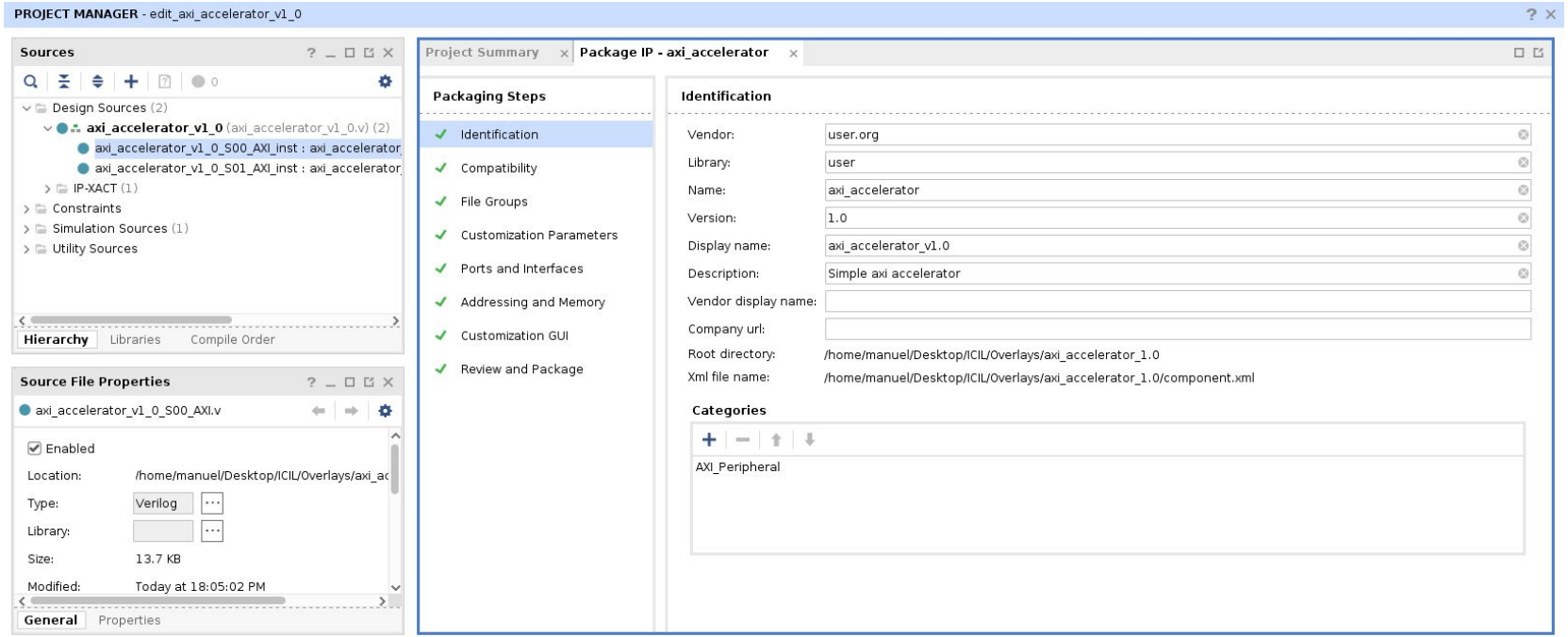
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S00_AXI.v (AXI Lite I*nterface)

```
104     //------------------------------------------------
105     //-- Signals for user logic register space example
106     //------------------------------------------------
107     //-- Number of Slave Registers 4
108     reg [C_S_AXI_DATA_WIDTH-1:0]     slv_reg0;
109     reg [C_S_AXI_DATA_WIDTH-1:0]     slv_reg1;
110     reg [C_S_AXI_DATA_WIDTH-1:0]     slv_reg2;
111     reg [C_S_AXI_DATA_WIDTH-1:0]     slv_reg3;
112     wire     slv_reg_rden;
113     wire     slv_reg_wren;
114     reg [C_S_AXI_DATA_WIDTH-1:0]     reg_data_out;
115     integer  byte_index;
116     reg  aw_en;
117
```

Register to map enable signal

How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S00_AXI.v (AXI Lite I*nterface)

```
401     // Add user logic here
402
403     always @(posedge S_AXI_ACLK)
404        enable_acc <= |slv_reg0; //Enable if slv_reg0 != 0
405
406     // User logic ends
```

Assignment of register to enable signal. This signal needs to be external.

How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S00_AXI.v (AXI Lite I*nterface)

```
17          // Users to add ports here
18          output reg enable_acc,
19
```

enable signal (as output) made external

How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S01_AXI.v (AXI Full I*nterface)

```
29      // Users to add ports here
30      input wire enable_acc,
31
```

enable signal (as input) made external

How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S01_AXI.v (AXI Full I*nterface)

```
613     // Add user logic here
614     reg [C_S_AXI_DATA_WIDTH-1 :0] local_mem [15:0];
615     reg [C_S_AXI_DATA_WIDTH-1:0] out_reg;
616     wire [C_S_AXI_DATA_WIDTH-1:0] out_wire;
617
618     accelerator accelerator(
619     .clk(S_AXI_ACLK),
620     .reset(S_AXI_ARESETN),
621     .enable(enable_acc),
622     .in_a(local_mem[0]),
623     .in_b(local_mem[1]),
624     .out(out_wire));
625
626
627     always @(posedge S_AXI_ACLK)
628         if ( axi_wready && S_AXI_WVALID )
629           local_mem[axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]]<=S_AXI_WDATA;
630
631     always @(posedge S_AXI_ACLK)
632         out_reg <= out_wire;
633
```

Memory mapped inputs

How to Deploy an Accelerator with an AXI Interface

Let's look at file *axi_accelerator_v1_0_S01_AXI.v (AXI Full I*nterface)

```
613     // Add user logic here
614     reg [C_S_AXI_DATA_WIDTH-1 :0] local_mem [15:0];
615     reg [C_S_AXI_DATA_WIDTH-1:0] out_reg;
616     wire [C_S_AXI_DATA_WIDTH-1:0] out_wire;
617
618     accelerator accelerator(
619     .clk(S_AXI_ACLK),
620     .reset(S_AXI_ARESETN),
621     .enable(enable_acc),
622     .in_a(local_mem[0]),
623     .in_b(local_mem[1]),
624     .out(out_wire));
625
626
627     always @(posedge S_AXI_ACLK)
628         if ( axi_wready && S_AXI_WVALID )
629           local_mem[axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]]<=S_AXI_WDATA;
630
631     always @(posedge S_AXI_ACLK)
632         out_reg <= out_wire;
633
```

Memory mapped outputs

How to Deploy an Accelerator with an AXI Interface

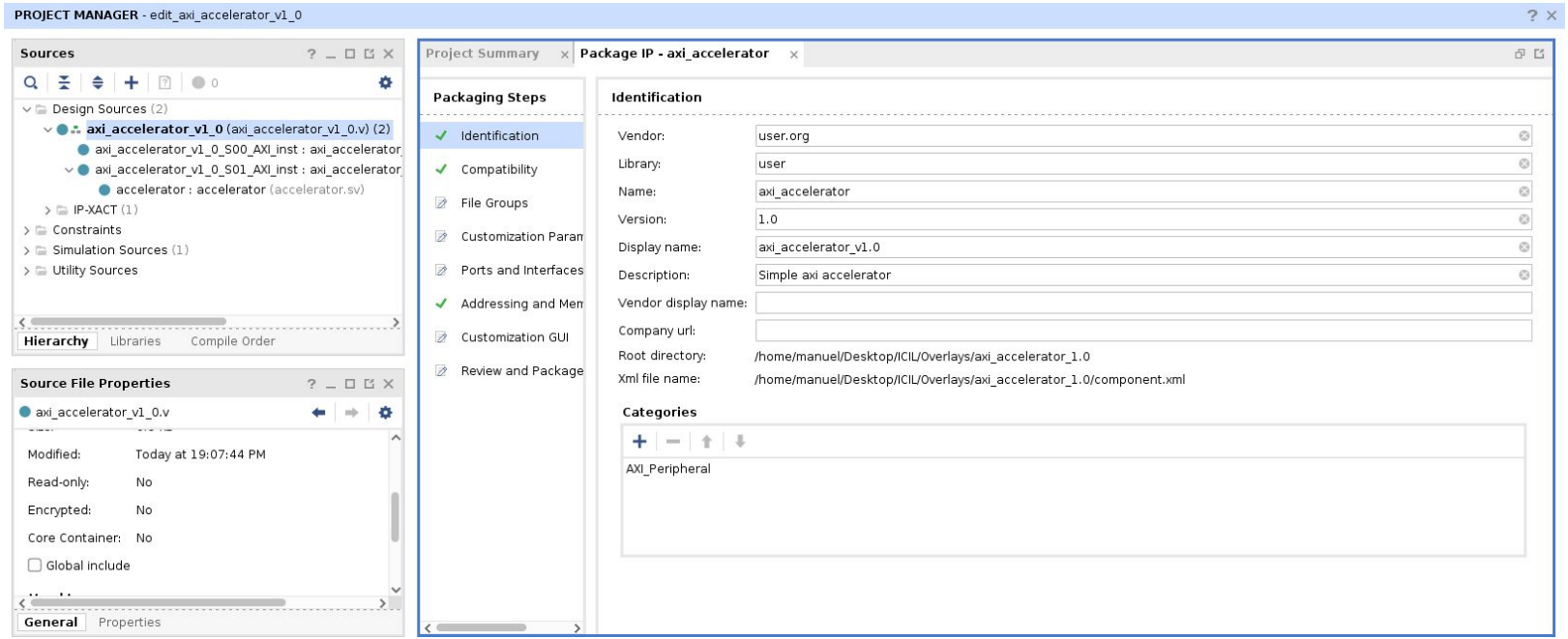
Let's look at file *axi_accelerator_v1_0_S01_AXI.v (AXI Full I*nterface)

```
613     // Add user logic here
614     reg [C_S_AXI_DATA_WIDTH-1 :0] local_mem [15:0];
615     reg [C_S_AXI_DATA_WIDTH-1:0] out_reg;
616     wire [C_S_AXI_DATA_WIDTH-1:0] out_wire;
617
618     accelerator accelerator(
619     .clk(S_AXI_ACLK),
620     .reset(S_AXI_ARESETN),
621     .enable(enable_acc),
622     .in_a(local_mem[0]),
623     .in_b(local_mem[1]),
624     .out(out_wire));
625
626
627     always @(posedge S_AXI_ACLK)
628         if ( axi_wready && S_AXI_WVALID )
629           local_mem[axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]]<=S_AXI_WDATA;
630
631     always @(posedge S_AXI_ACLK)
632         out_reg <= out_wire;
633
```

Instantiation of hardware accelerator.

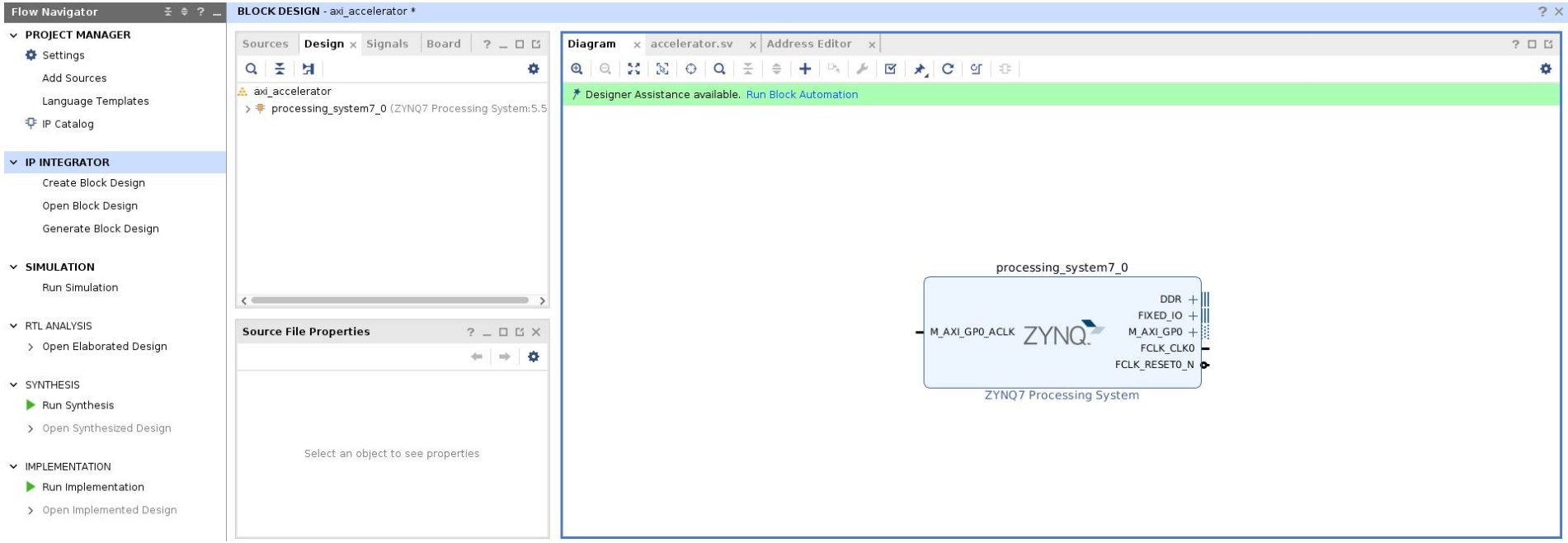
How to Deploy an Accelerator with an AXI Interface

2. Package your design using Vivado



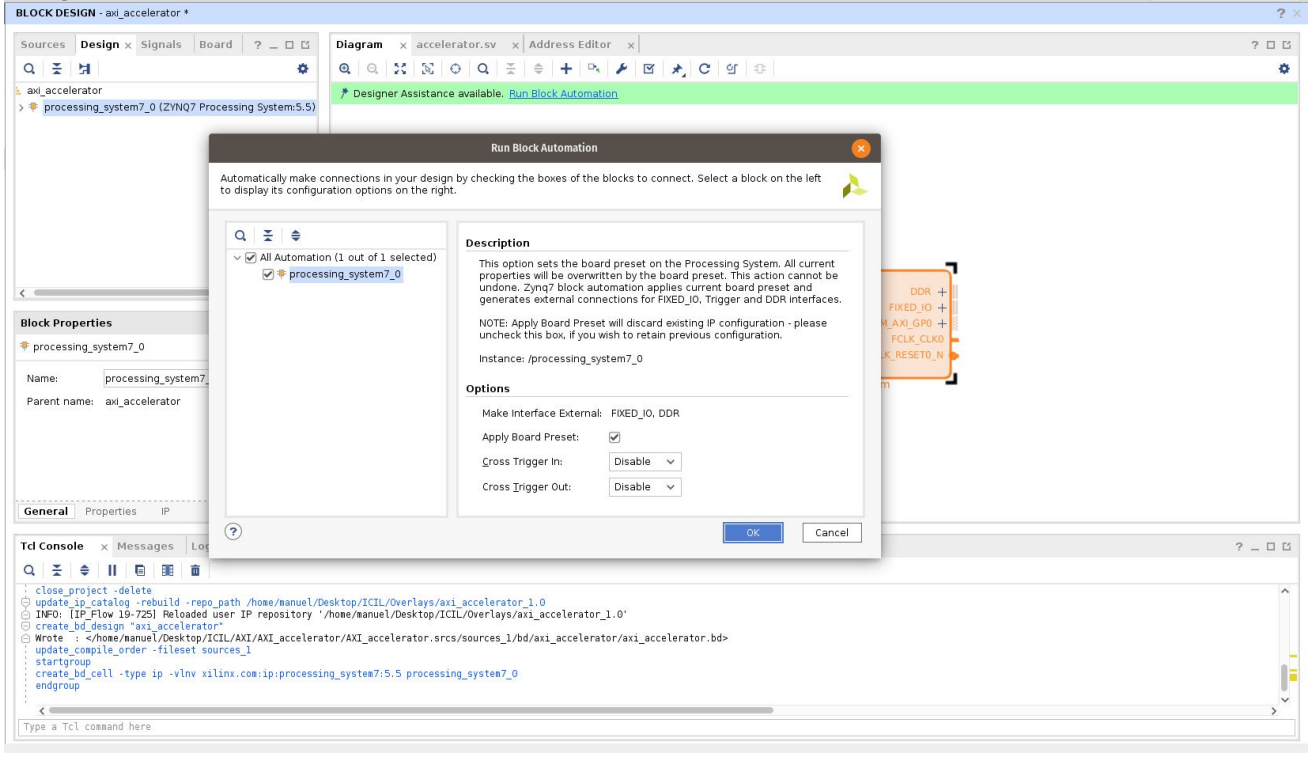
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



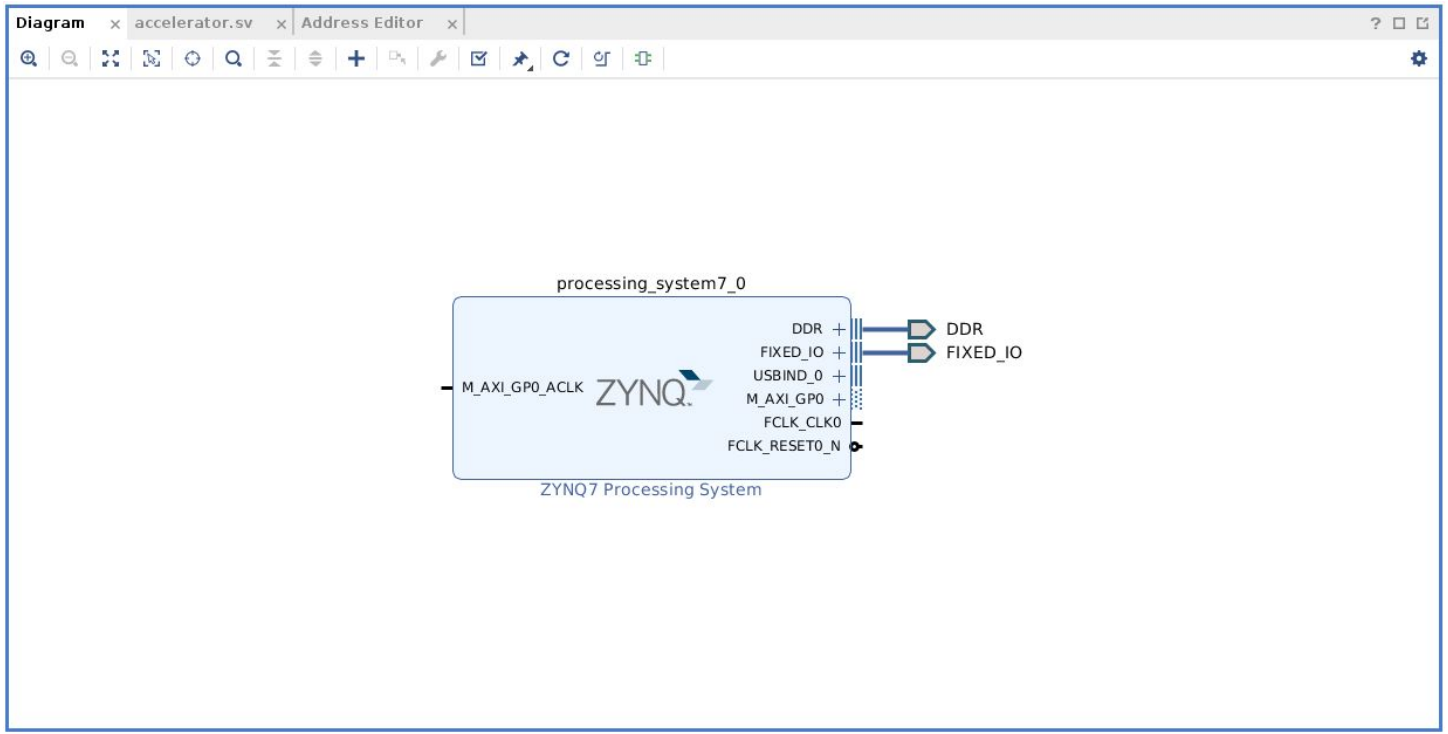
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



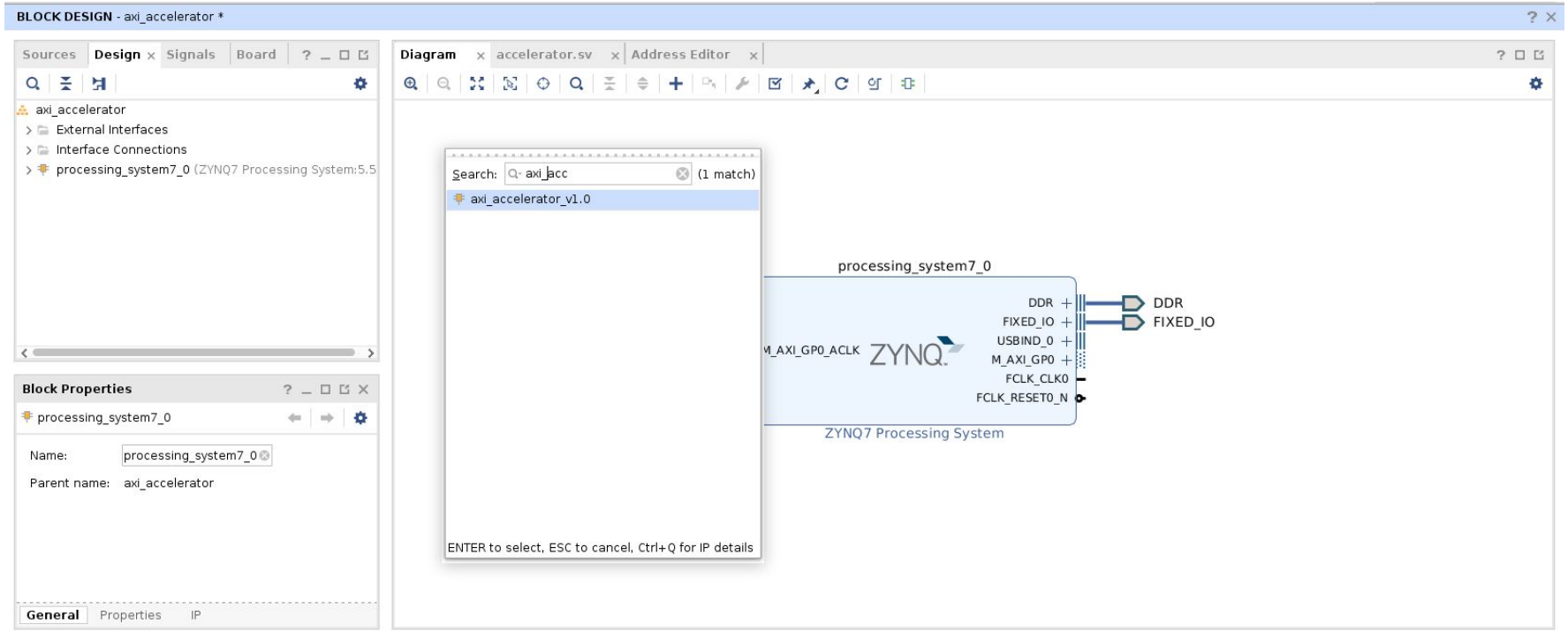
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



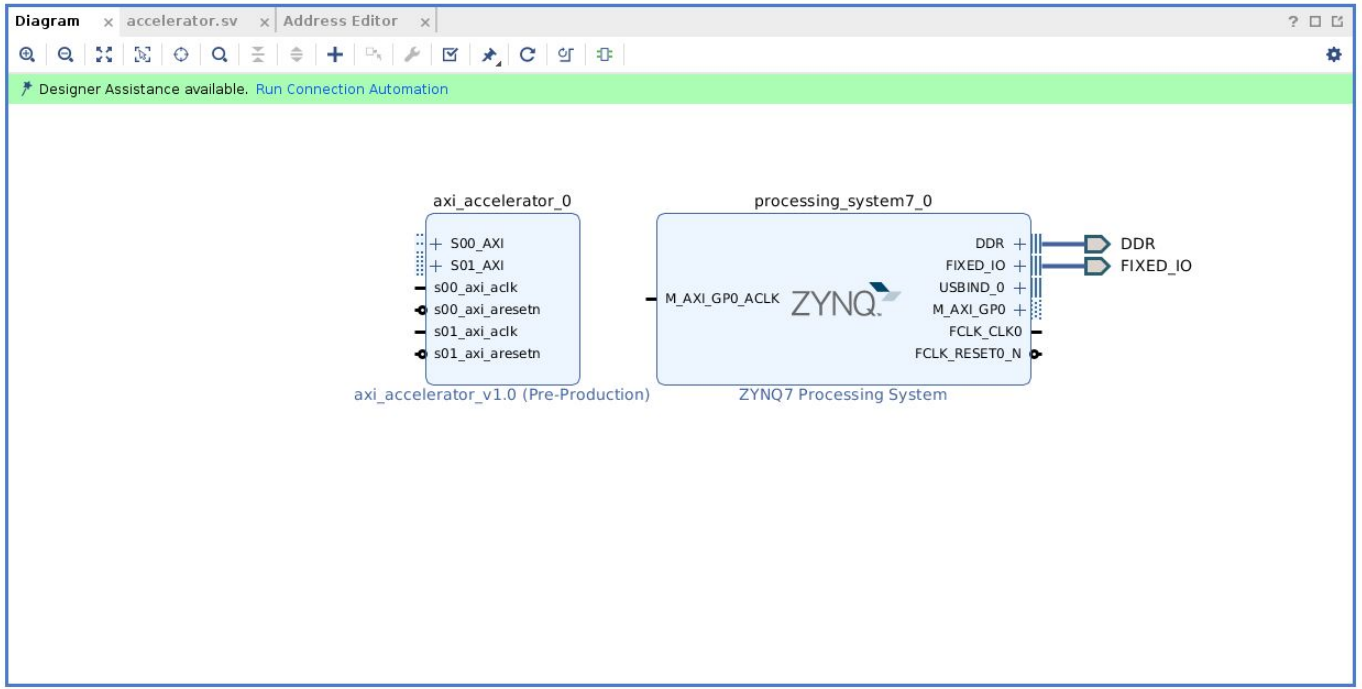
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



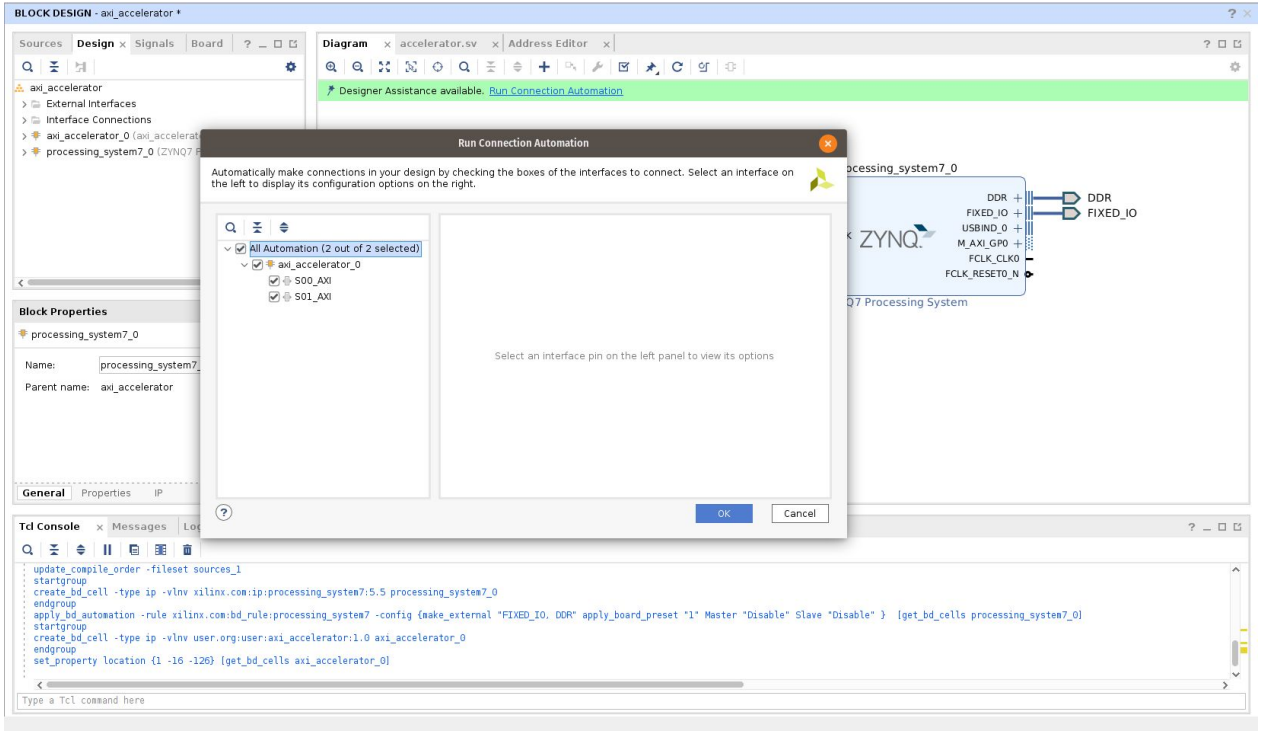
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



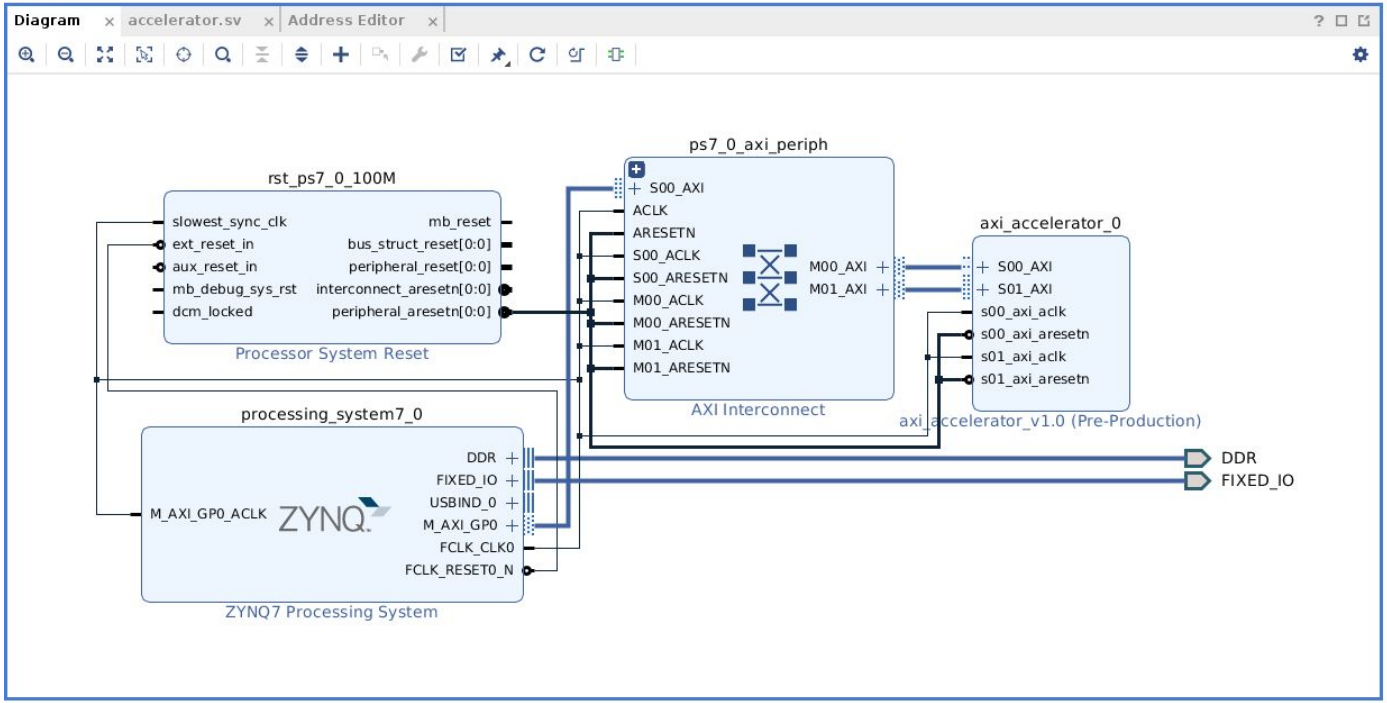
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



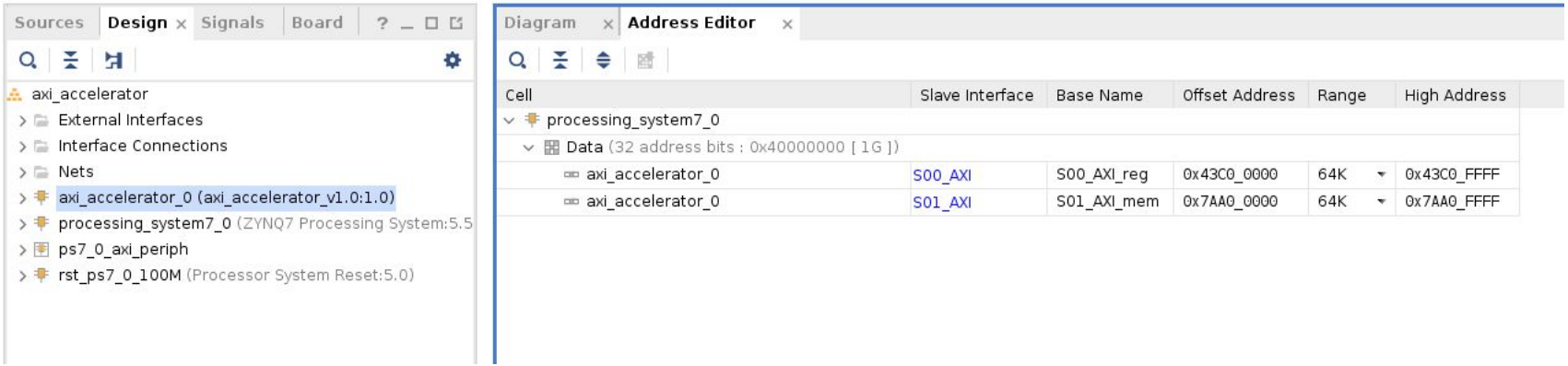
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



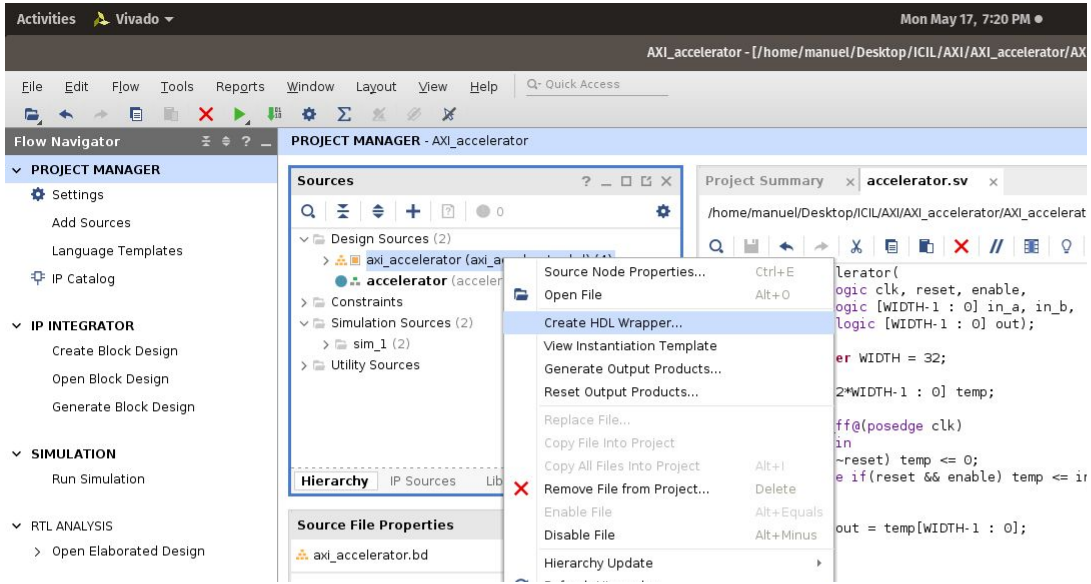
How to Deploy an Accelerator with an AXI Interface

3. Design your Zynq system and add your own package



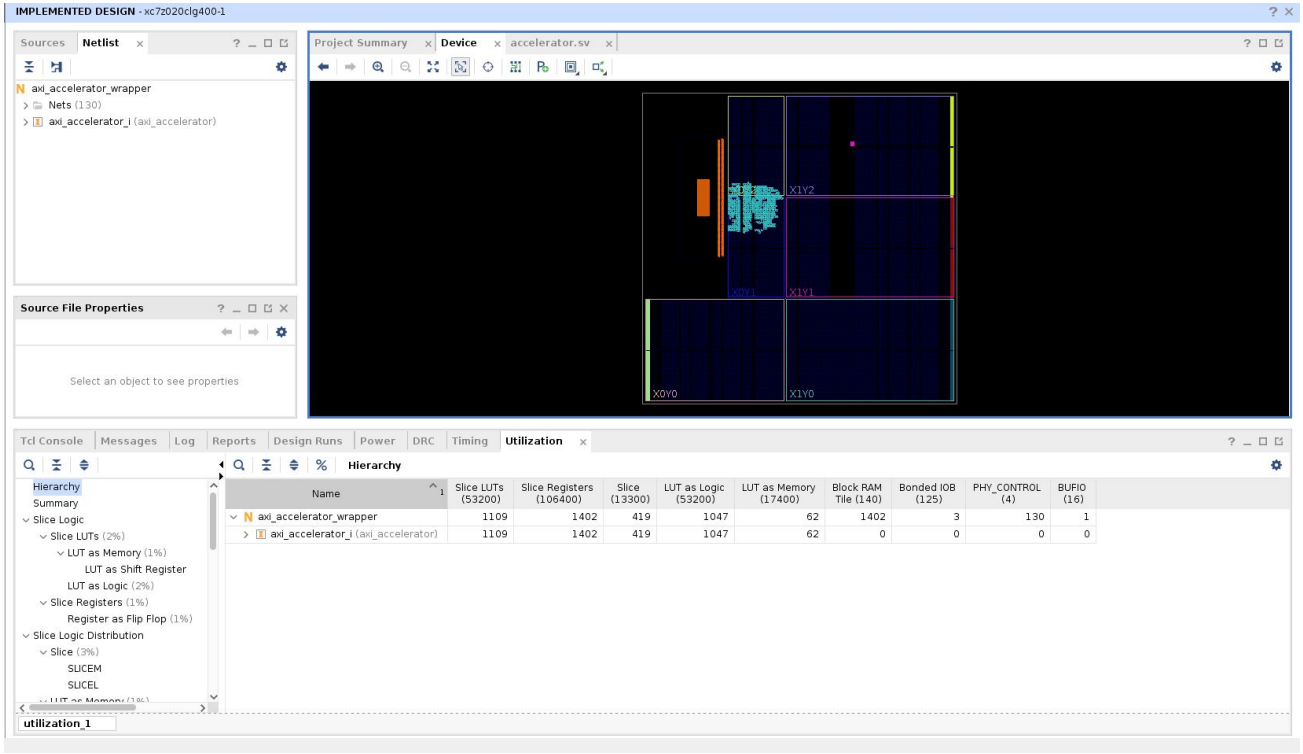
How to Deploy an Accelerator with an AXI Interface

4. Create a HDL wrapper for your design



How to Deploy an Accelerator with an AXI Interface

5. Generate bitstream



How to Use your Accelerator with PYNQ Framework

1. Save bitstream file (*.bit) and Hardware Handoff file(*.hwh) inside the overlay folder of your board.
2. Create a Jupyter Notebook and load the bitstream file.
3. Transfer data between software and hardware.

How to Use your Accelerator with PYNQ Framework

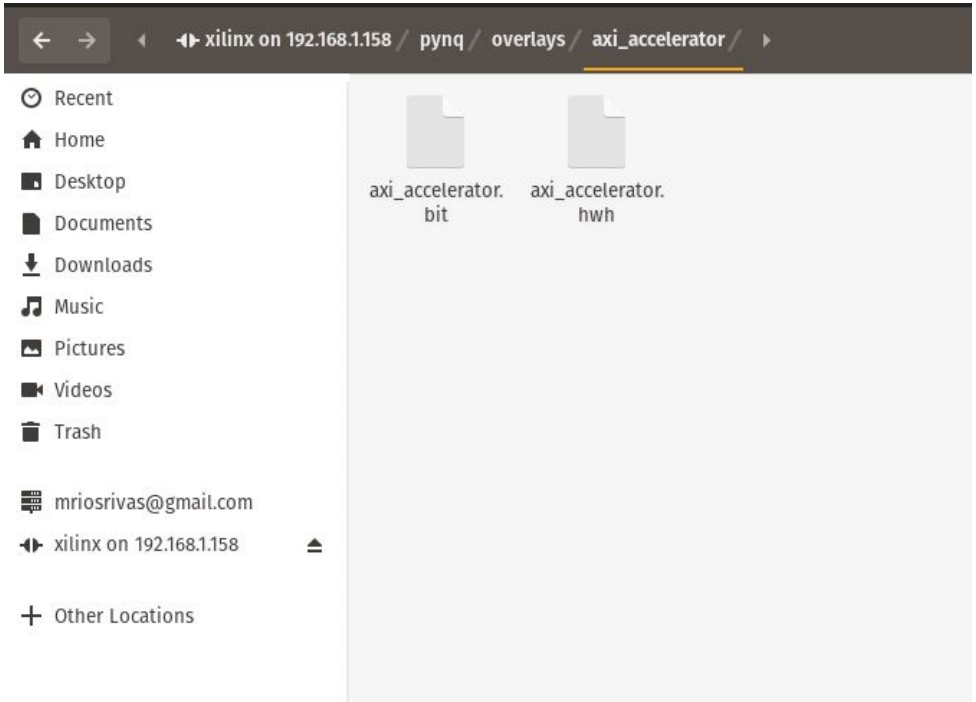
1. Save bitstream file (*.bit) and Hardware Handoff file(*.hwh) inside the overlay folder of your board.

File	Location
axi_accelerator.hwh	$PROJECT_PATH/srcs/sources_1/bd/axi_accelerator/hw_handoff
axi_accelerator_wrapper.bit	$PROJECT_PATH/runs/impl_1

Note: When copying the files rename both with the same name. The script *get_hardware.py* might be useful.

How to Use your Accelerator with PYNQ Framework

1. Save bitstream file (*.bit) and Hardware Handoff file(*.hwh) inside the overlay folder.



How to Use your Accelerator with PYNQ Framework

2. Create a Jupyter Notebook and load the bitstream file.

In [1]:
```
from pynq import Overlay
overlay = Overlay('/home/xilinx/pynq/overlays/axi_accelerator/axi_accelerator.bit')
```

Check if your overlay was correctly generated

In [2]:
```
overlay?
```

You should be able to see some information like this:

```
IP Blocks
----------
axi_accelerator_0/S00_AXI : pynq.overlay.DefaultIP
axi_accelerator_0/S01_AXI : pynq.overlay.DefaultIP
processing_system7_0 : pynq.overlay.DefaultIP
```

How to Use your Accelerator with PYNQ Framework

3. Transfer data between software and hardware.

In Python you will be able to access each AXI interfase as different objects

In [3]:
```
axi_lite = overlay.axi_accelerator_0.S00_AXI
axi_full = overlay.axi_accelerator_0.S01_AXI
```

First we set our enable signal to zero. Remember that the `enable` signal es mapped to `slv_reg0` from the AXI Lite interconnect.

In [4]:
```
axi_lite.write(0,0)
```

Now we set our memory mapped values with the AXI Full interconnect.

In [5]:
```
axi_full.write(0,25) #input in_a is mapped to address 0
axi_full.write(4,2) #input in_b is mapped to address 4
```

We enable the accelerator to start the calculation

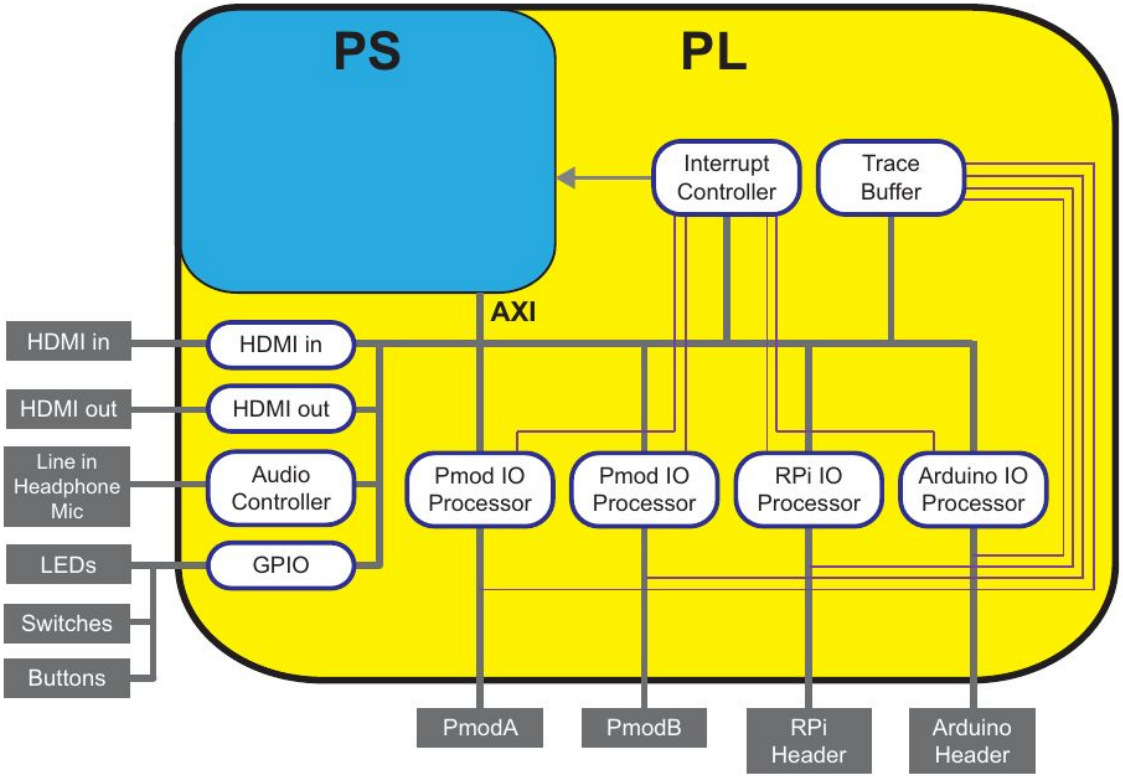
In [6]:
```
axi_lite.write(0,1)
```

Now we check the correct result in the AXI Full interconnection at address zero.

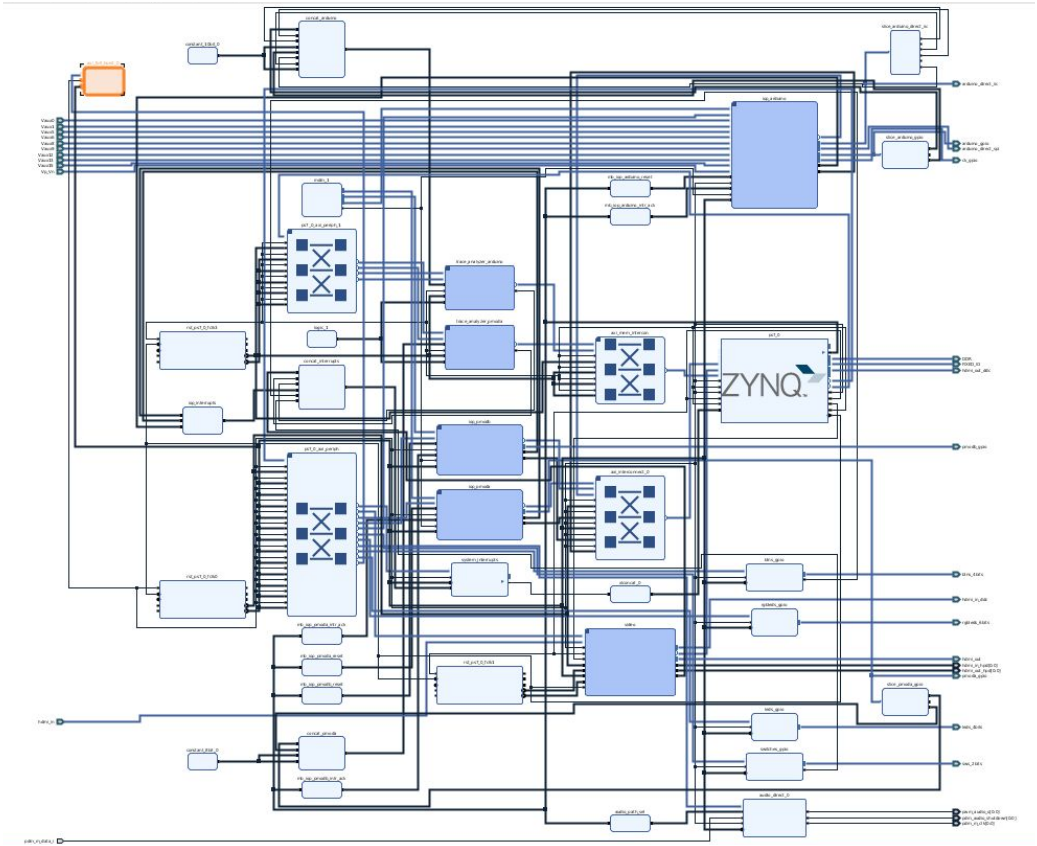
In [8]:
```
axi_full.read(0)
```

Out[8]: 50

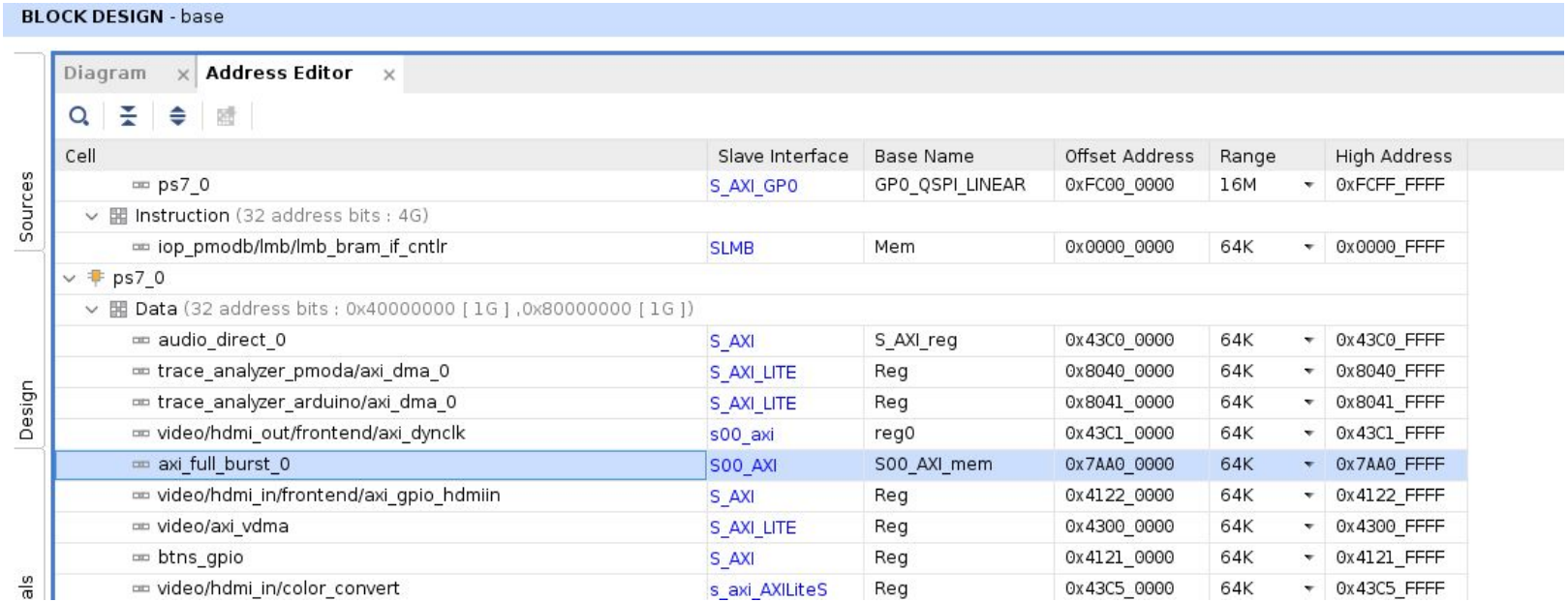
The Base Overlay



Modifying the Base Overlay



Modifying the Base Overlay



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
ps7_0	S_AXI_GP0	GP0_QSPI_LINEAR	0xFC00_0000	16M	0xFCFF_FFFF
Instruction (32 address bits : 4G)					
iop_pmodb/lmb/lmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	64K	0x0000_FFFF
ps7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
audio_direct_0	S_AXI	S_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF
trace_analyzer_pmoda/axi_dma_0	S_AXI_LITE	Reg	0x8040_0000	64K	0x8040_FFFF
trace_analyzer_arduino/axi_dma_0	S_AXI_LITE	Reg	0x8041_0000	64K	0x8041_FFFF
video/hdmi_out/frontend/axi_dynclk	s00_axi	reg0	0x43C1_0000	64K	0x43C1_FFFF
axi_full_burst_0	S00_AXI	S00_AXI_mem	0x7AA0_0000	64K	0x7AA0_FFFF
video/hdmi_in/frontend/axi_gpio_hdmiin	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
video/axi_vdma	S_AXI_LITE	Reg	0x4300_0000	64K	0x4300_FFFF
btns_gpio	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
video/hdmi_in/color_convert	s_axi_AXILiteS	Reg	0x43C5_0000	64K	0x43C5_FFFF

Modifying the Base Overlay

In [1]:
```
from pynq import Overlay
overlay = Overlay('/home/xilinx/pynq/overlays/base_modified/base_ver_1.bit')
```

Passing data by stream

To send data by stream we make use of the MMIO class from the PYNQ framework. The MMIO class allows a Python object to access addresses in the system memory mapped. In particular, registers and address space of peripherals in the PL can be accessed. In an overlay, peripherals connected to the AXI General Purpose ports will have their registers or address space mapped into the system memory map. With PYNQ, the register, or address space of an IP can be accessed from Python using the MMIO class.

In [6]:
```
from pynq import MMIO
import numpy as np
```

In [7]:
```
mmio = MMIO(0x7AA00000,0x10000) #(IP_BASE_ADDRESS, ADDRESS_RANGE)
```

To access the memory mapped data we can use the `array` method. For example, to write 16 different values we could do

In [8]:
```
mmio.array[0:16] = np.arange(0,16)
```

And to check that the values were stored

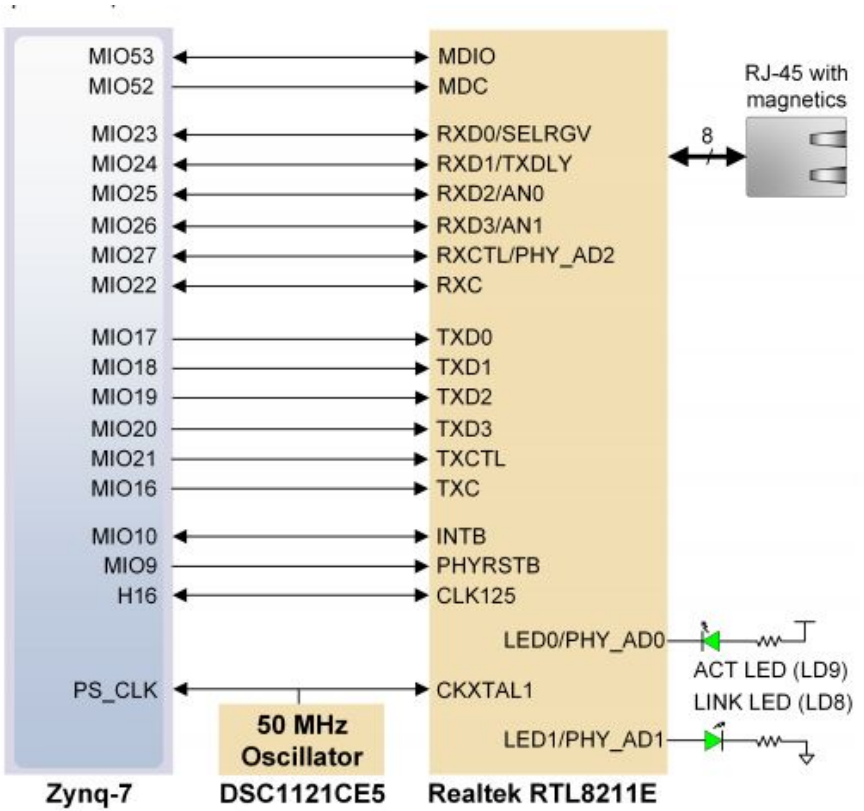
In [11]:
```
mmio.array.view(dtype='int32')
```

Out[11]:
```
array([ 5,  6,  7, ..., 18, 19, 20])
```

Ethernet Connection

The PYNQ-Z1 uses a Realtek RTL8211E-VL PHY to implement a 10/100/1000 Ethernet port for network connection.

We can use a network socket to connect to the Zynq.



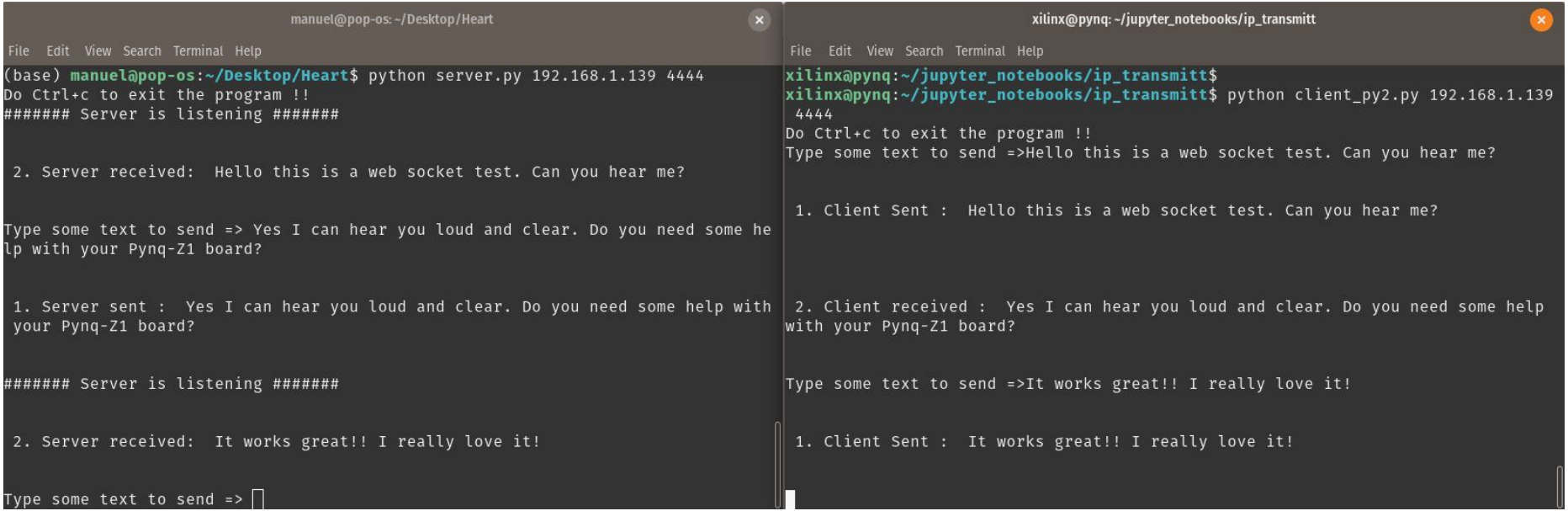
Web Socket - Server

```
1 import socket
2 import sys
3
4 if len(sys.argv) == 3:
5     # Get "IP address of Server" and also the "port number" from argument 1
  and argument 2
6     ip = sys.argv[1]
7     port = int(sys.argv[2])
8 else:
9     print "Run like : python3 server.py <arg1:server ip:this system IP
  192.168.1.6> <arg2:server port:4444 >"
10    exit(1)
11
12 # Create a UDP socket
13 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
14 # Bind the socket to the port
15 server_address = (ip, port)
16 s.bind(server_address)
17 print "Do Ctrl+c to exit the program !!"
18
19 while True:
20    print "####### Server is listening #######"
21    data, address = s.recvfrom(4096)
22    print "\n\n 2. Server received: ", data.decode('utf-8'), "\n\n"
23    send_data = raw_input("Type some text to send => ")
24    s.sendto(send_data.encode('utf-8'), address)
25    print "\n\n 1. Server sent : ", send_data,"\n\n"
```

Web Socket - Client

```
1 import socket
2 import sys
3
4 if len(sys.argv) == 3:
5     # Get "IP address of Server" and also the "port number" from argument 1
  and argument 2
6     ip = sys.argv[1]
7     port = int(sys.argv[2])
8 else:
9     "Run like : python3 client.py <arg1 server ip 192.168.1.102> <arg2 server
  port 4444 >"
10    exit(1)
11
12 # Create socket for server
13 s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, 0)
14 print "Do Ctrl+c to exit the program !!"
15
16 # Let's send data through UDP protocol
17 while True:
18    send_data = raw_input("Type some text to send =>");
19    s.sendto(send_data.encode('utf-8'), (ip, port))
20    print "\n\n 1. Client Sent : ", send_data, "\n\n"
21    data, address = s.recvfrom(4096)
22    print "\n\n 2. Client received : ", data.decode('utf-8'), "\n\n"
23 # close the socket
24 s.close()
```

Web Socket - Example



Other Types of Communication

- Another possibility is doing the transfers via the UART. There is one connected to the PS which is connected to the onboard FTDI chip. Its transfer rate is maximum 12MBaud/s.
- If you want to do transfers with the base overlay directly to PL, there are UARTs connected to the onboard microblaze on your PMOD and arduino headers.
- If you’re making your own overlay and you don’t want the processor coming between the communication, you can implement your own hardware module for UART, I2C, SPI, parallel etc.
- The Zynq7020 does not support PCIe.