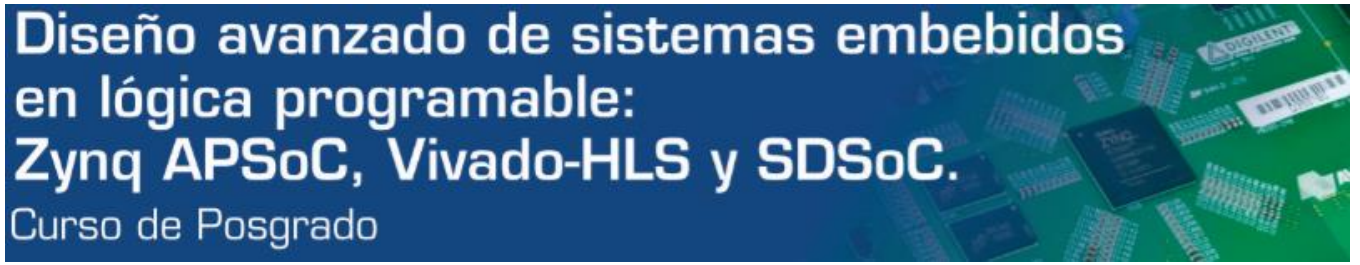


Introduction to AXI – Custom IP

Cristian Sisterna

Universidad Nacional de San Juan

Argentina

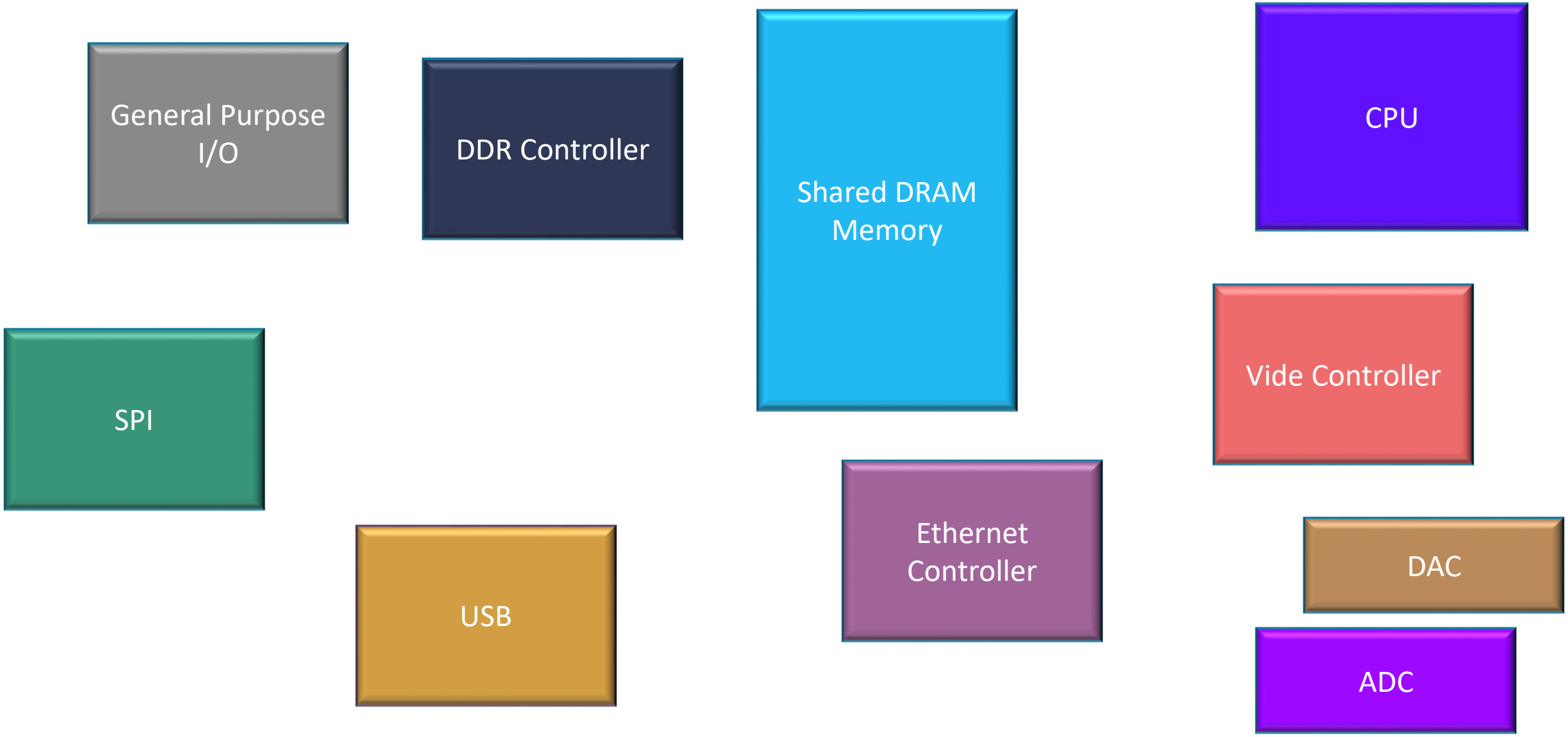
Agenda

- Describe the AXI4 transactions
- Summarize the AXI4 valid/ready acknowledgment model
- Discuss the AXI4 transactional modes of overlap and simultaneous operations
- Describe the operation of the AXI4 streaming protocol

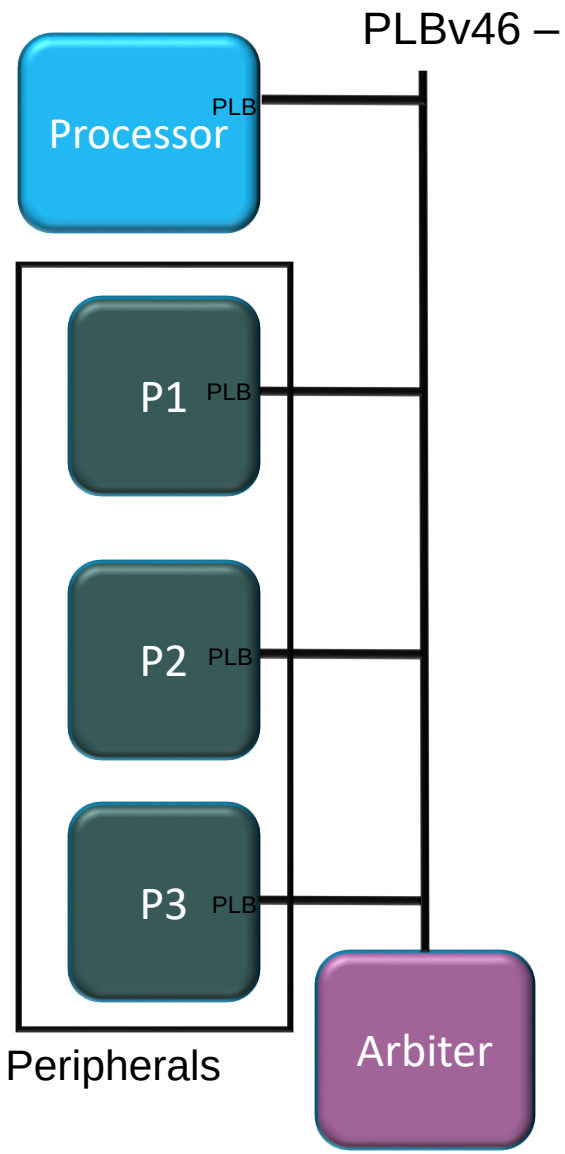
Need to Understand Device's Connectivity

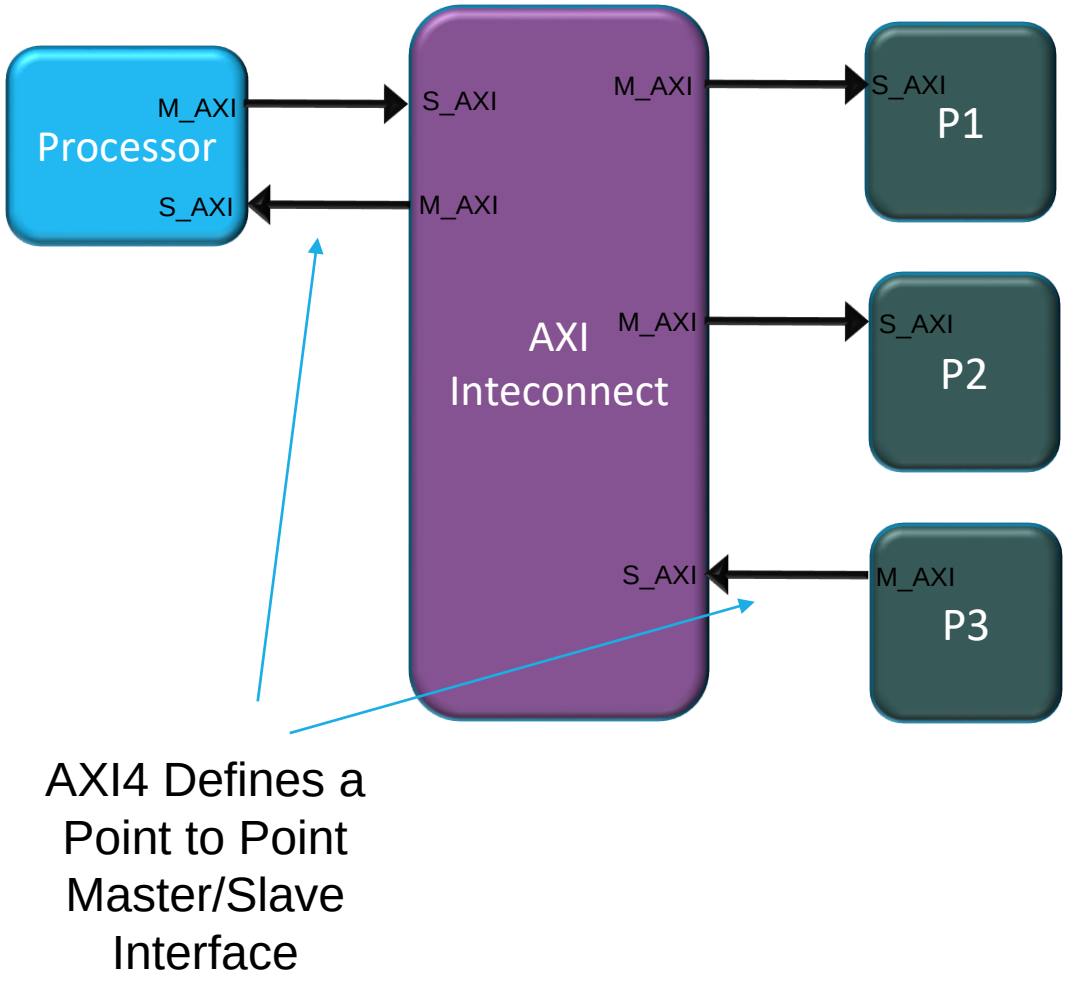
- There is a need to get familiar with the way that different devices communicate each other in an Embedded System like a Zynq based system
- Learning and understanding the communication among devices will facilitate the design of Zynq based systems
- All the devices in a Zynq system communicate each other based in a device interface standard developed by ARM, called AXI (ARM eXtended Interface):
  - AXI define a Point to Point Master/Slave Interface

Today's System-On-Chip



Interface Options





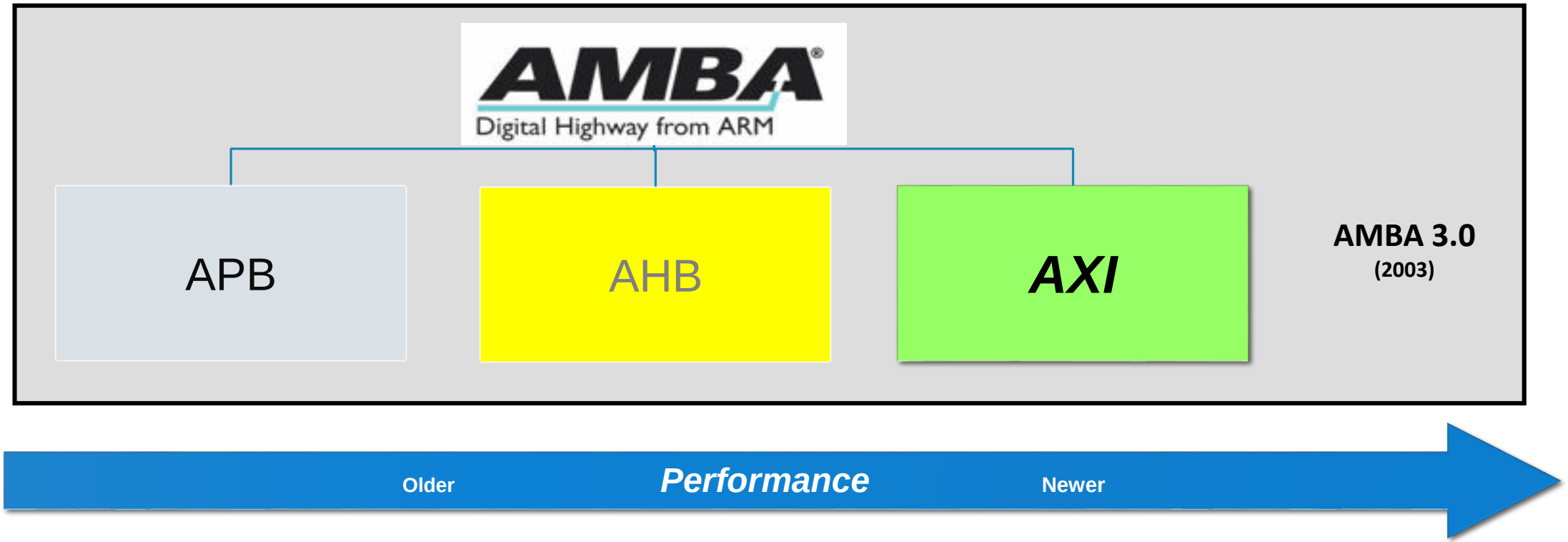
Connectivity -> Standard

- **A standard**
  - All units talk based on the same standard (same protocol, same language)
  - All units can easily talk to each other
- **Maintanence**
  - Design is easily maintained/updated
  - Facilitate debug tasks
- **Re-Use**
  - Developed cores can easily re-used in other systems

Common SoC Interfaces

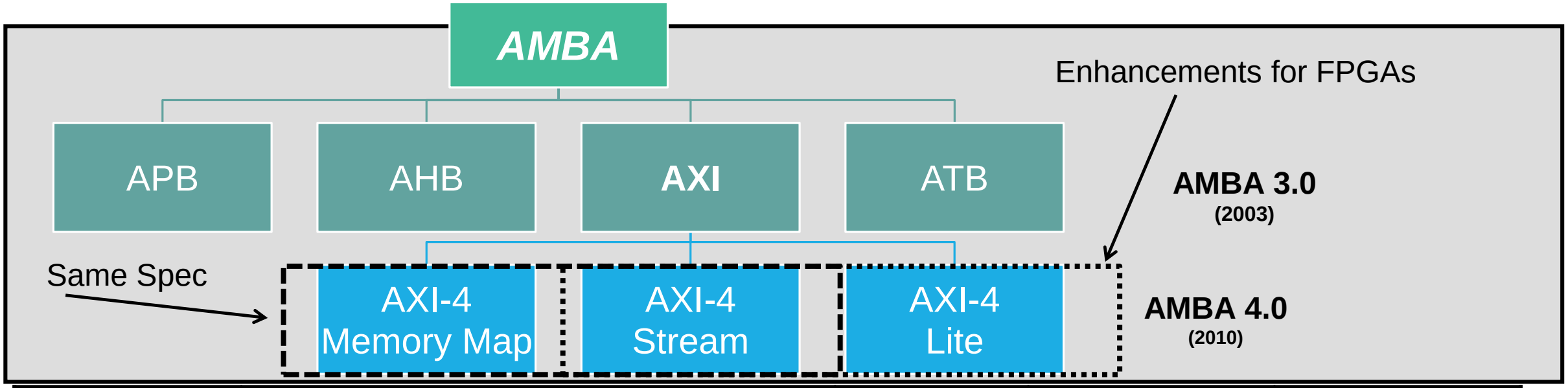
- **Core Connect (IBM)**
  - PLB/OPB (Power PC-FPGA bus interface)
- **WishBone**
  - OpenCore Cores
- **AXI**
  - ARM standard (more to come . . .)

AXI is Part of ARM's AMBA



AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced Extensible Interface

AXI is Part of AMBA



Interface	Features	Burst	Data Width	Applications
AXI4	Traditional Address/Data Burst (single address, multiple data)	Up to 256	32 to 1024 bits	Embedded, Memory
AXI4-Stream	Data-Only, Burst	Unlimited	Any Number	DSP, Video, Communications
AXI4-Lite	Traditional Address/Data—No Burst (single address, single data)	1	32 or 64 bits	Small Control Logic, FSM

AXI – Vocabulary

Channel

- Independent collection of AXI signals associated to a VALID signal

Interface

- Collection of one or more channels that expose an IP core's connecting a master to a slave
- Each IP core may have multiple interfaces

Bus

- Multiple-bit signal (not an interface or channel)

Transfer

- *Single clock cycle* where information is communicated, qualified by a VALID handshake

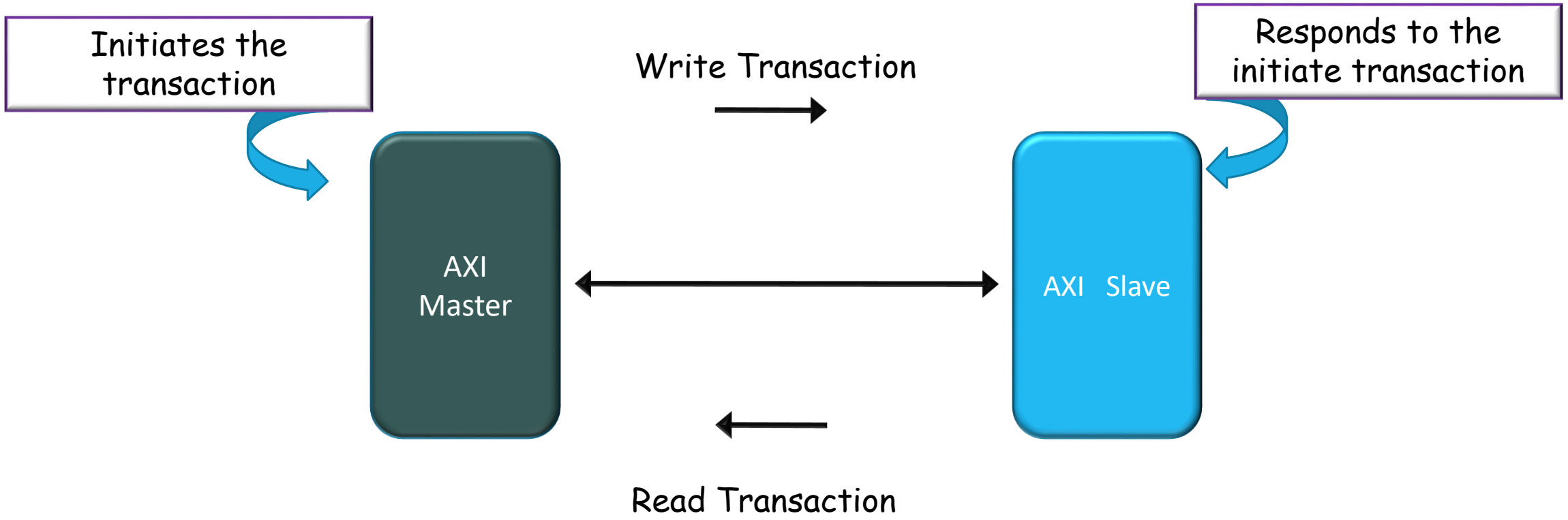
Transaction

- Complete communication operation across a channel, composed of a one or more transfers

Burst

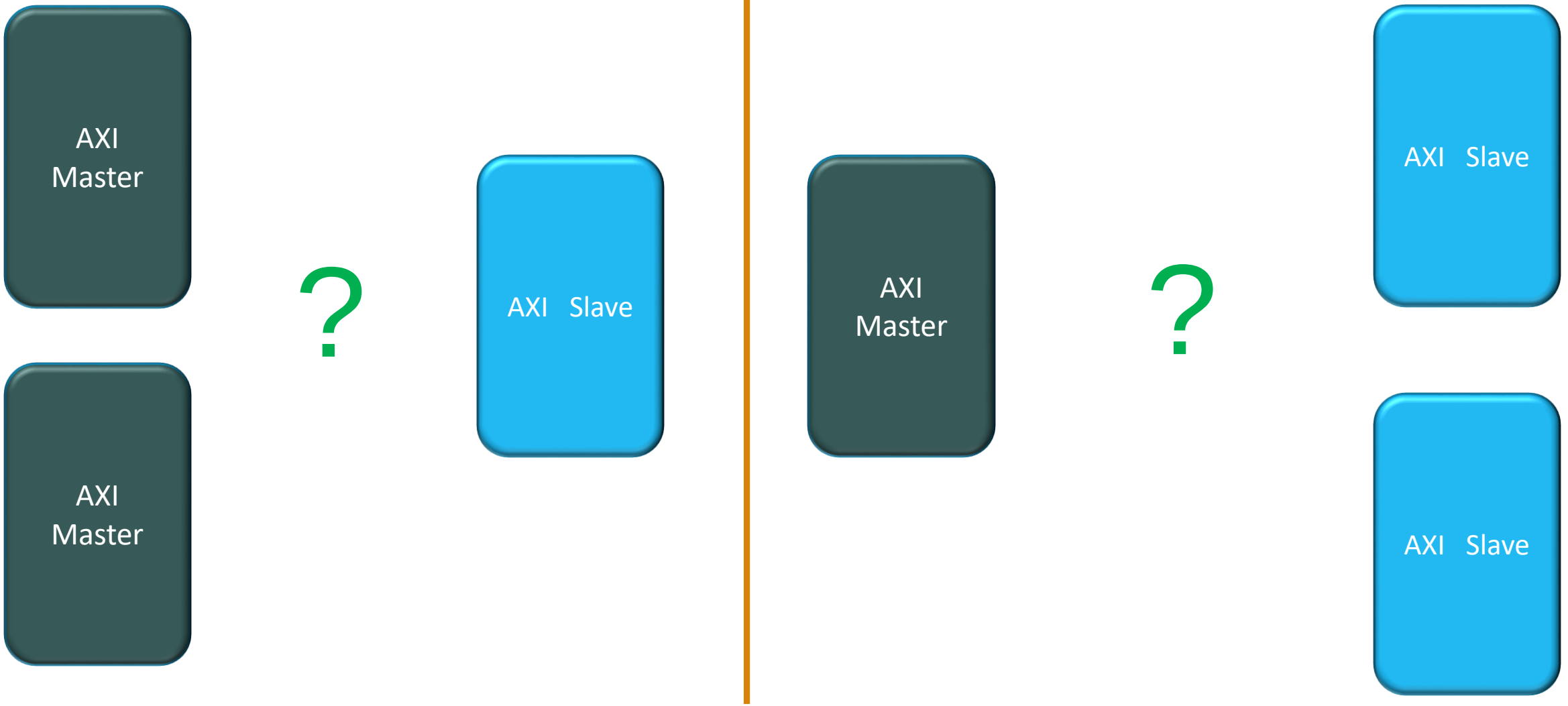
- Transaction that consists of more than one transfer

AXI Transactions / Master-Slave



Transactions: transfer of data from one point on the hardware to another point

More than One-to-One

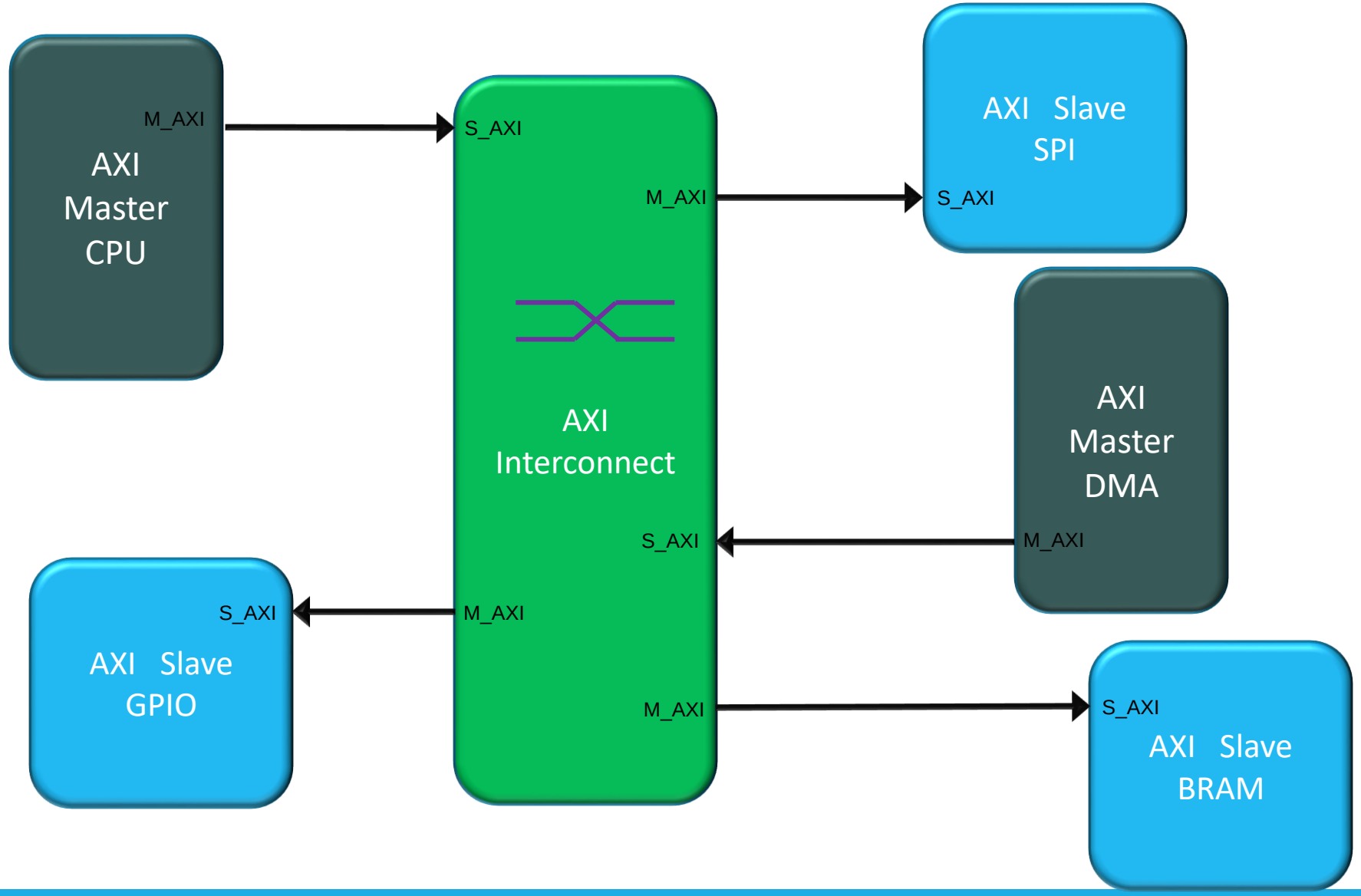


AXI Interconnect

AXI is an interconnect system used to tie processors to peripherals

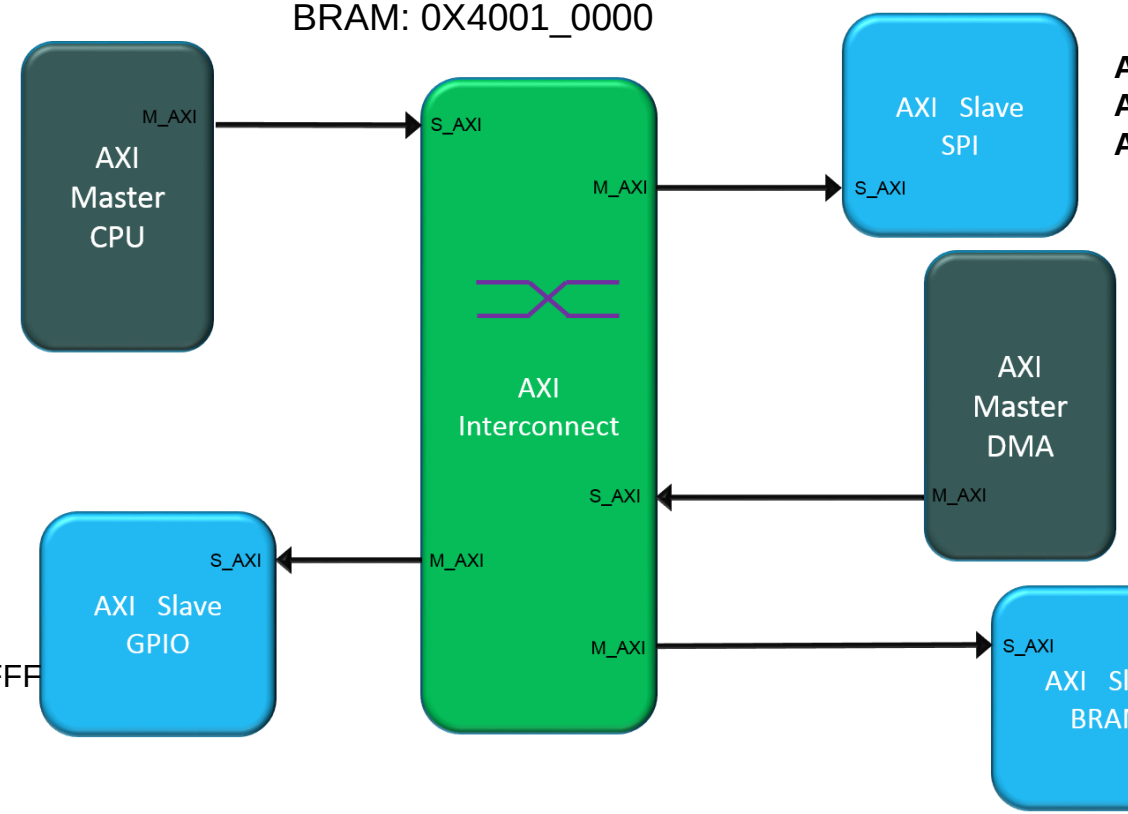
- **AXI Full memory map**: Full performance bursting interconnect
- **AXI Lite**: Lower performance non bursting interconnect (saves programmable logic resources)
- **AXI Streaming**: Non-addressed packet based or raw interface

AXI Interconnect



AXI Interconnect – Addressing & Decoding

Address Decoding Table
GPIO: 0X4000_0000
SPI: 0X4000_1000
BRAM: 0X4001_0000



Address Range: 4K
Address Offset: 0X4000_1000
Addresses: 0X4000_0000 – 0X4000_1FFF

Address Range: 4K
Address Offset: 0X4000_0000
Addresses: 0X4000_0000 – 0X4000_0FFF

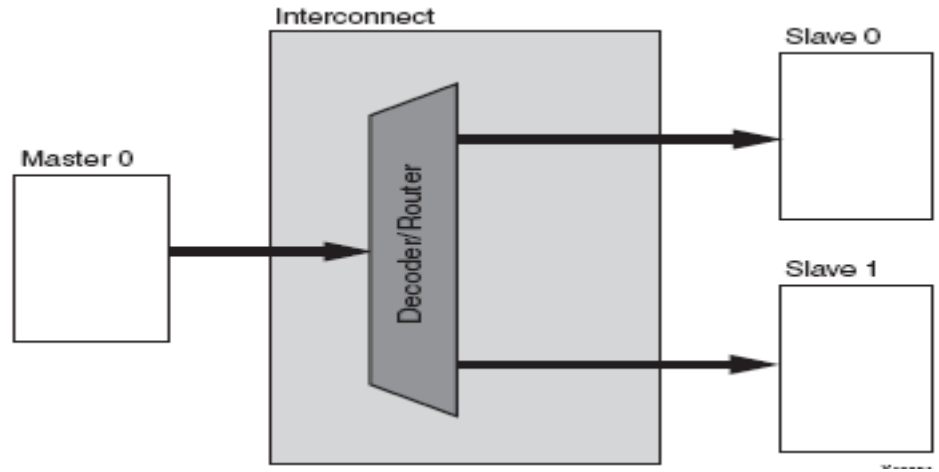
Address Range: 64K
Address Offset: 0X4001_0000
Addresses: 0X4001_0000 – 0X4001_FFFF

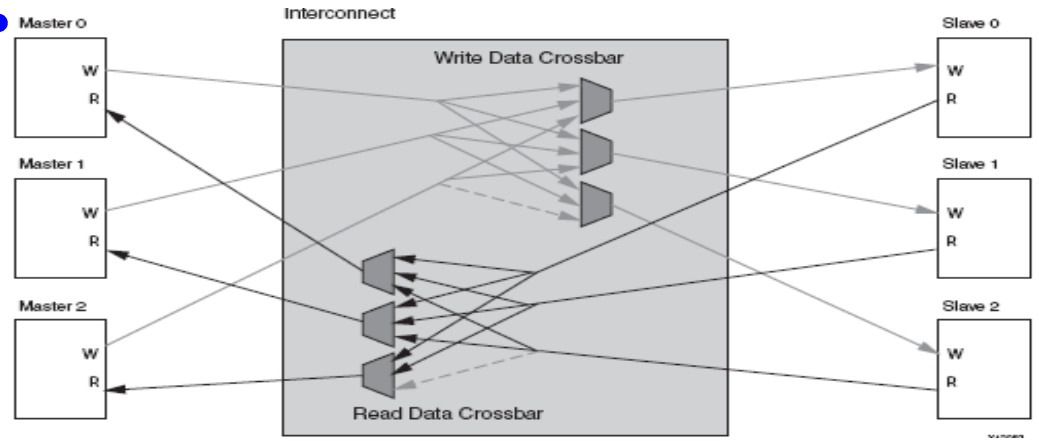
AXI Interconnect Main Features

- Different Number of (up to 16)
  - Slave Ports
  - Master Ports
- Data Width Conversion
- Conversion from AXI3 to AXI4
- Register Slices, Input/Output FIFOs
- Clock Domains Transfer

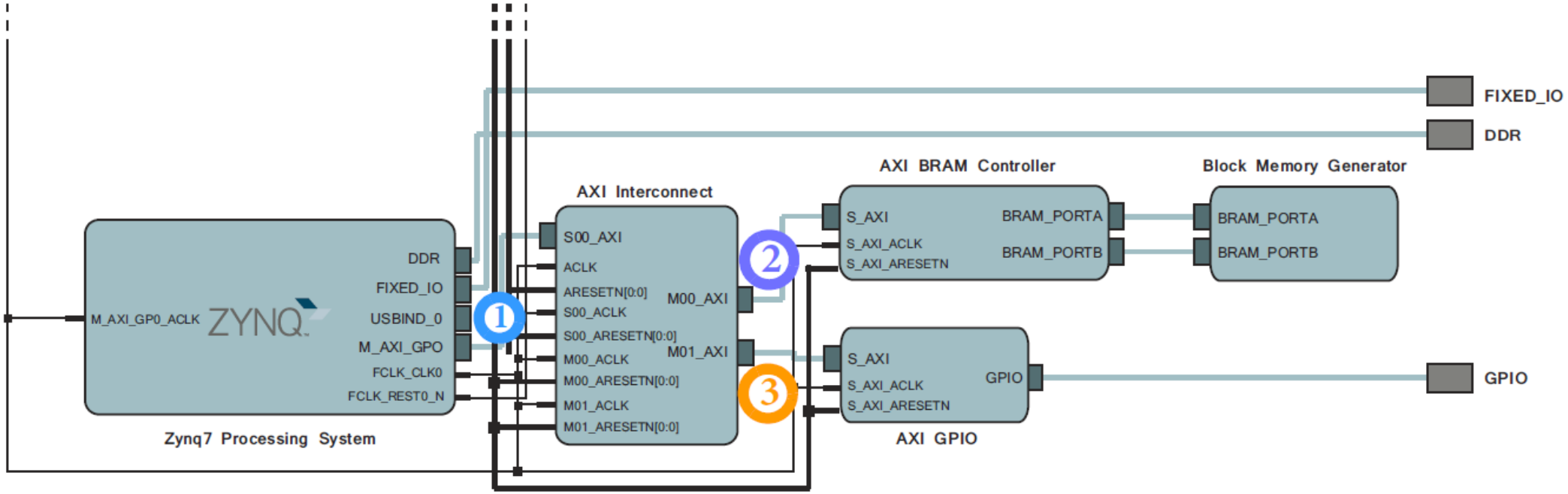
AXI Interconnect

- axi_interconnect component
  - Highly configurable
    - Pass Through
    - Conversion Only
    - N-to-1 Interconnect
    - 1-to-N Interconnect
    - N-to-M Interconnect – full crossbar
    - N-to-M Interconnect – shared bus structure
- **Decoupled master and slave interfaces**
- **Xilinx provides three configurable**
  - AXI4 Lite Slave
  - AXI4 Lite Master
  - AXI4 Slave Burst
- **Xilinx AXI Reference Guide(UG761)**





AXI Interface Example

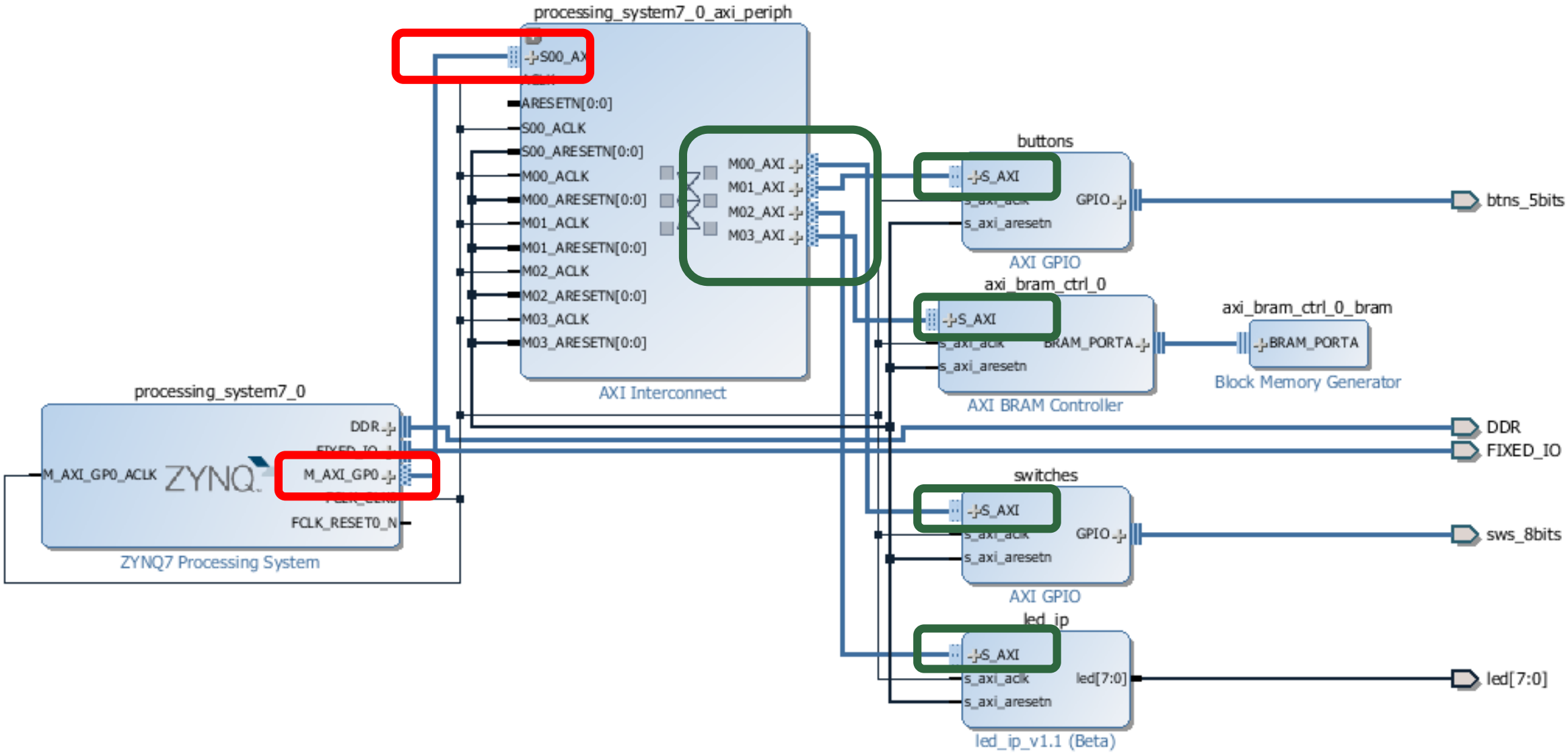


1. The AXI master signal from the Zynq processing system connects to the AXI slave port of the AXI Interconnect block

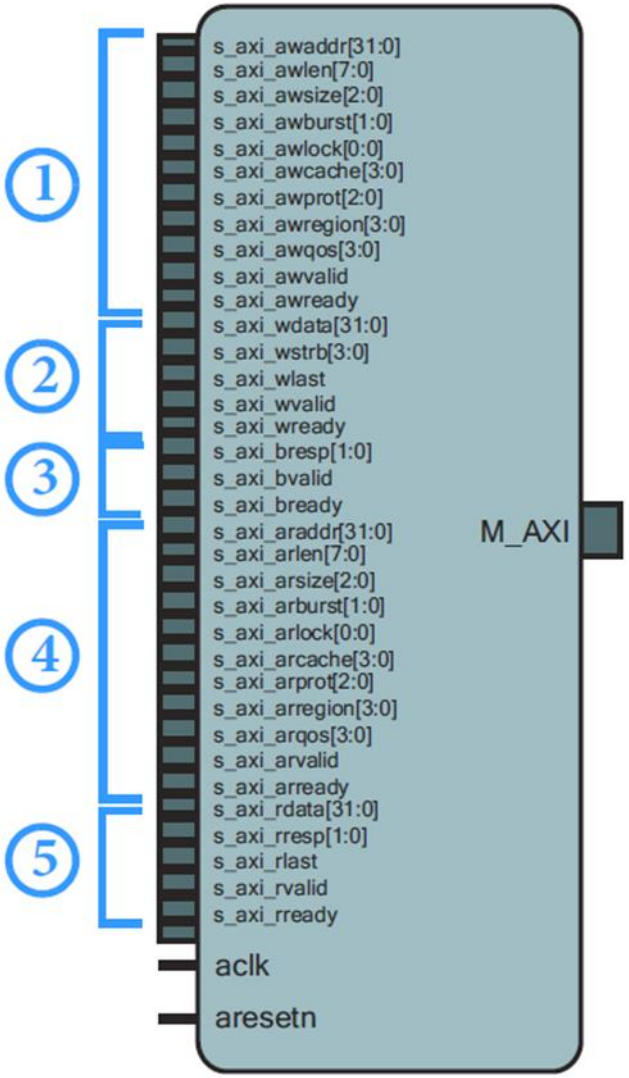
2. An AXI master signal from the AXI Interconnect connects to the AXI slave port of the BRAM controller

3. An AXI master signal from the AXI Interconnect connects to the AXI slave port of the GPIO instance

AXI Interface Example



AXI Slave Signals



① • **Write Address Channel** — the signals contained within this channel are named in the format *s_axi_aw...*

② • **Write Data Channel** — the signals contained within this channel are named in the format *s_axi_w...*

③ • **Write Response Channel** — the signals contained within this channel are named in the format *s_axi_b...*

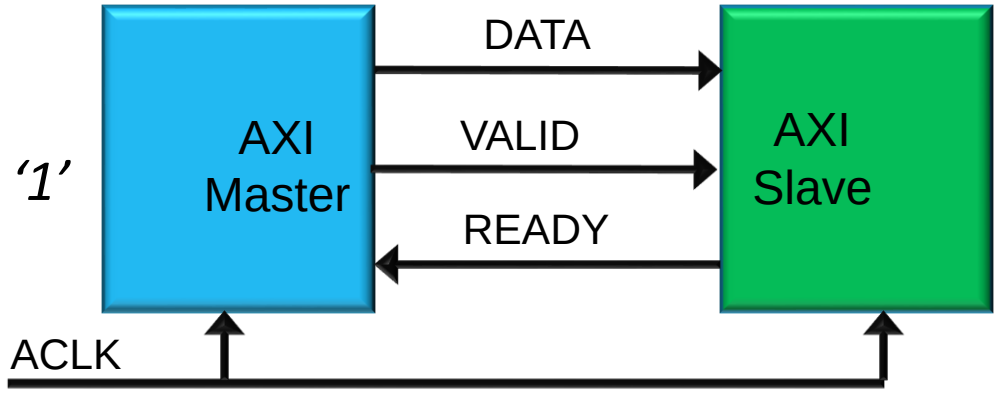
④ • **Read Address Channel** — the signals contained within this channel are named in the format *s_axi_ar...*

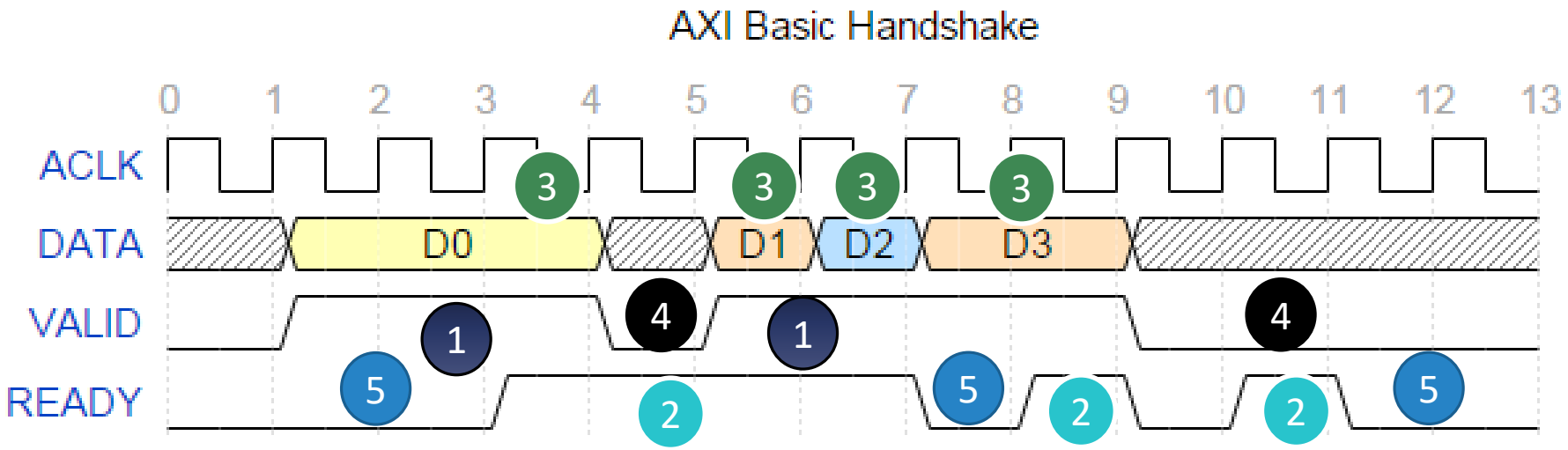
⑤ • **Read Data Channel** — the signals contained within this channel are named in the format *s_axi_r...*

Basic AXI Rd/Wr Process

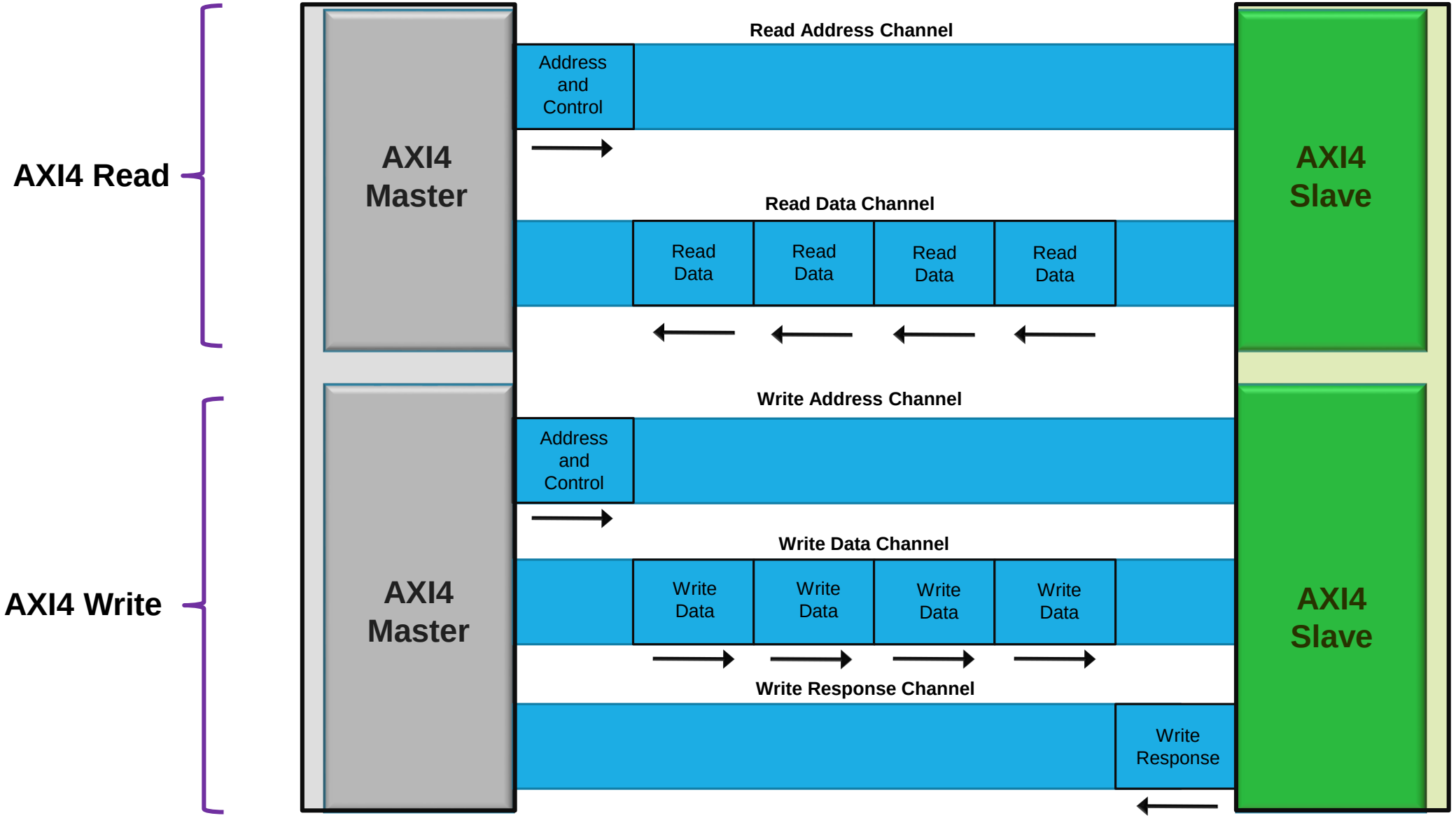
AXI Channels Use A Basic "VALID/READY" Handshake

1. *Master asserts and hold VALID when data is available*
2. *Slave asserts READY if able to accept data*
3. *Data and other signals transferred when VALID and READY = '1'*
4. *Master sends next DATA/other signals or deasserts VALID*
5. *Slave deasserts READY if no longer able to accept data*

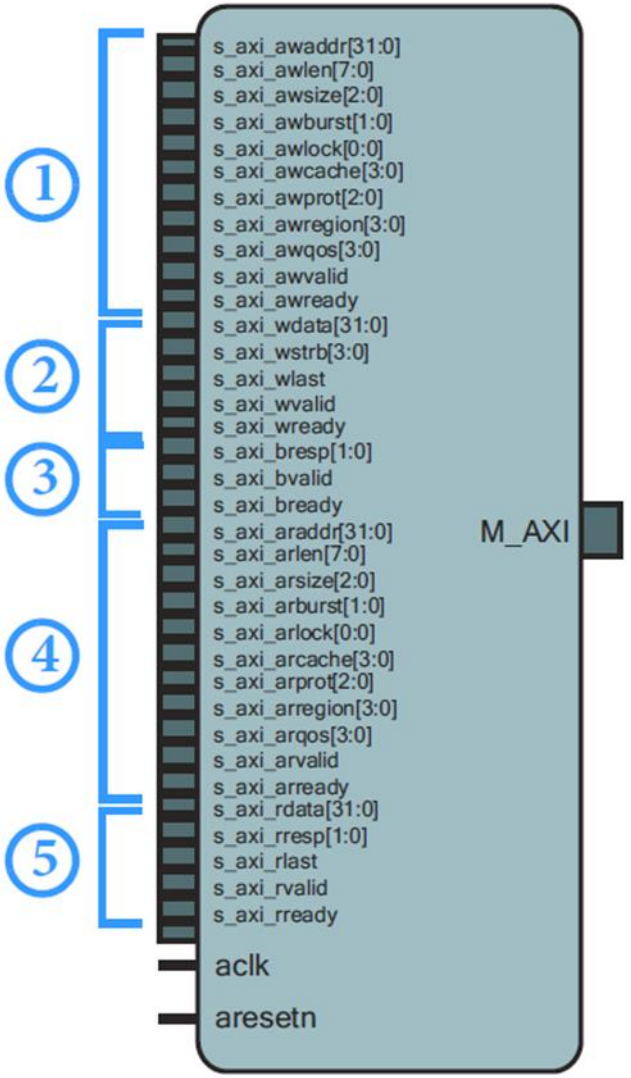




AXI Channels (AXI4 and AXI Lite)



AXI Slave - Channels



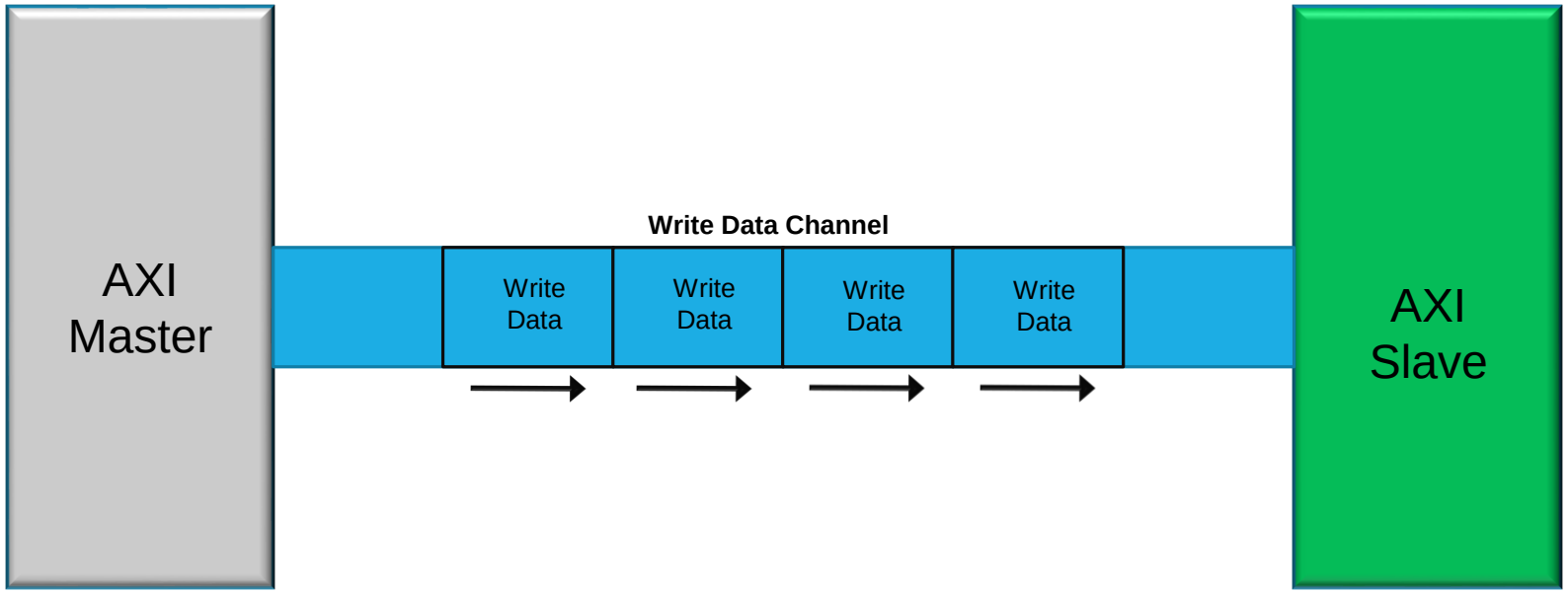
1. **Write Address Channel** — the signals contained within this channel are named in the format *s_axi_aw...*
2. **Write Data Channel** — the signals contained within this channel are named in the format *s_axi_w...*
3. **Write Response Channel** — the signals contained within this channel are named in the format *s_axi_b...*
4. **Read Address Channel** — the signals contained within this channel are named in the format *s_axi_ar...*
5. **Read Data Channel** — the signals contained within this channel are named in the format *s_axi_r...*

(Full) AXI4

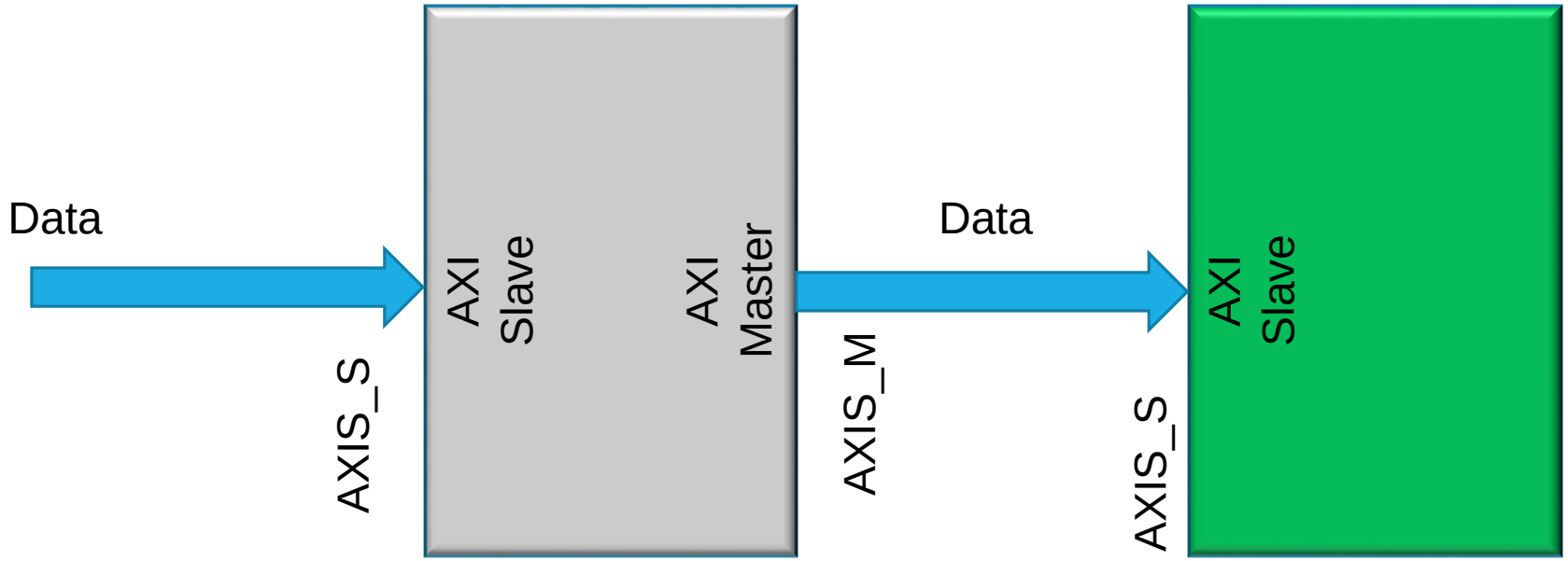
- Sometimes called "*Full AXI*" or "*AXI Memory Mapped*"
- Single address multiple data
  - Burst up to 256 data
- Data Width parameterizable
  - 32, 64, 128, 256, 512, 1024 bits

AXI4 Stream

- No address channel, no read and write, always just Master to Slave
  - Just an AXI4 Write Channel
- Unlimited burst length
- Supports sparse, continuous, aligned, unaligned streams



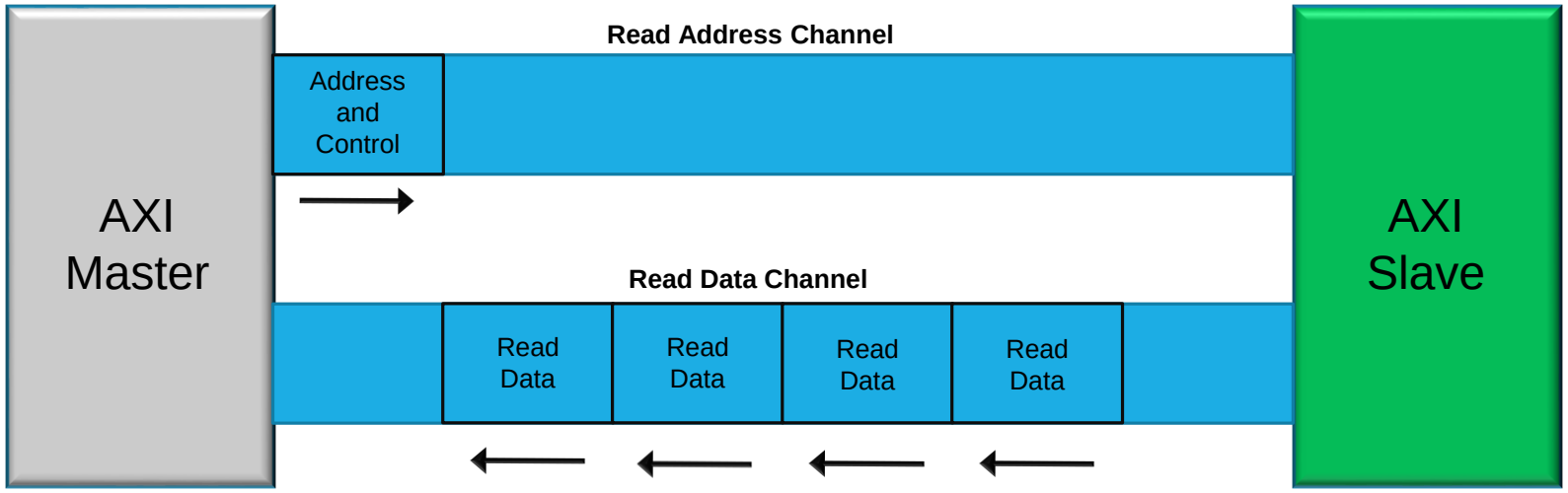
AXI Stream



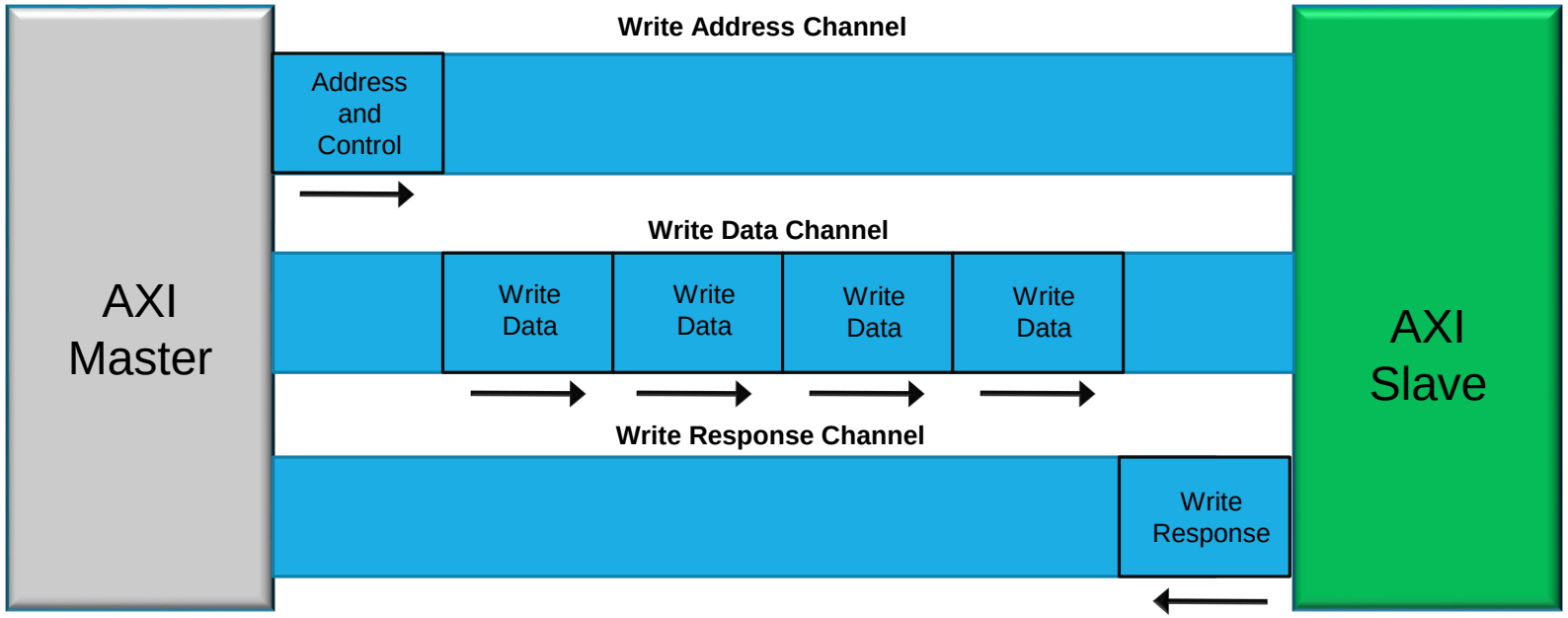
AXI4 Lite

- No Burst
- Single address, single data
- Data Width 32 or 64 bits (Xilinx IP only support 32)
- Very small size
- The AXI Interconnect is automatically generated

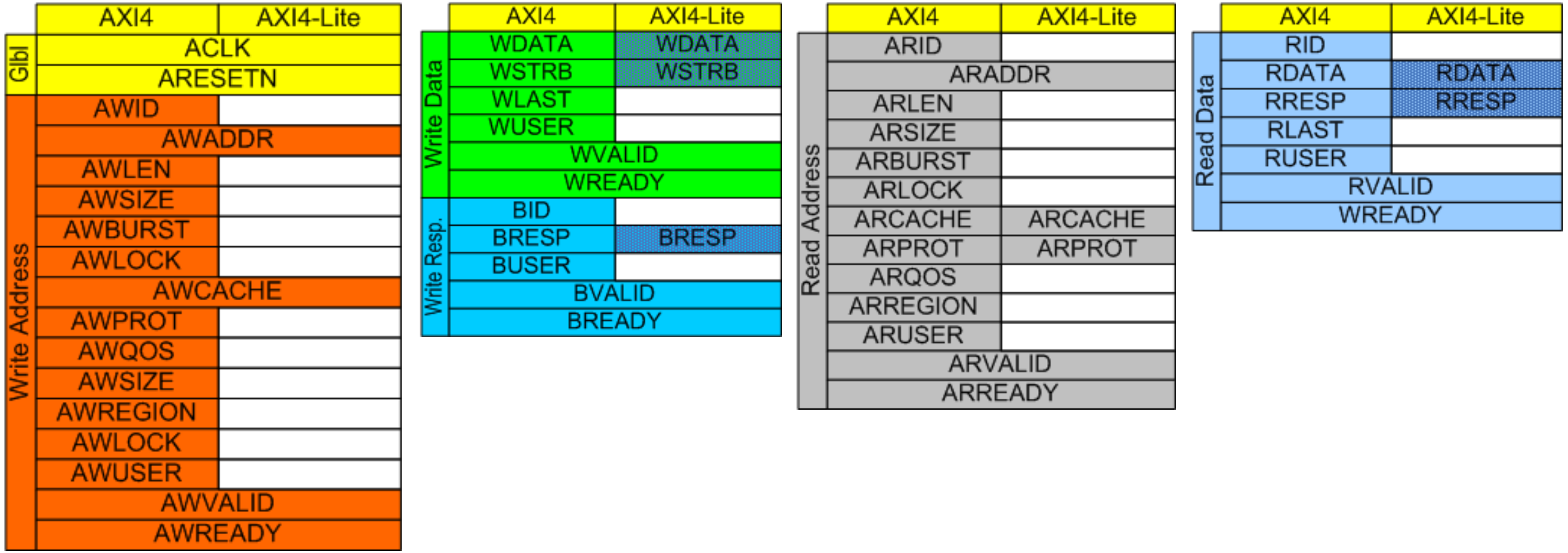
AXI4 Lite Read



AXI4 Lite Write

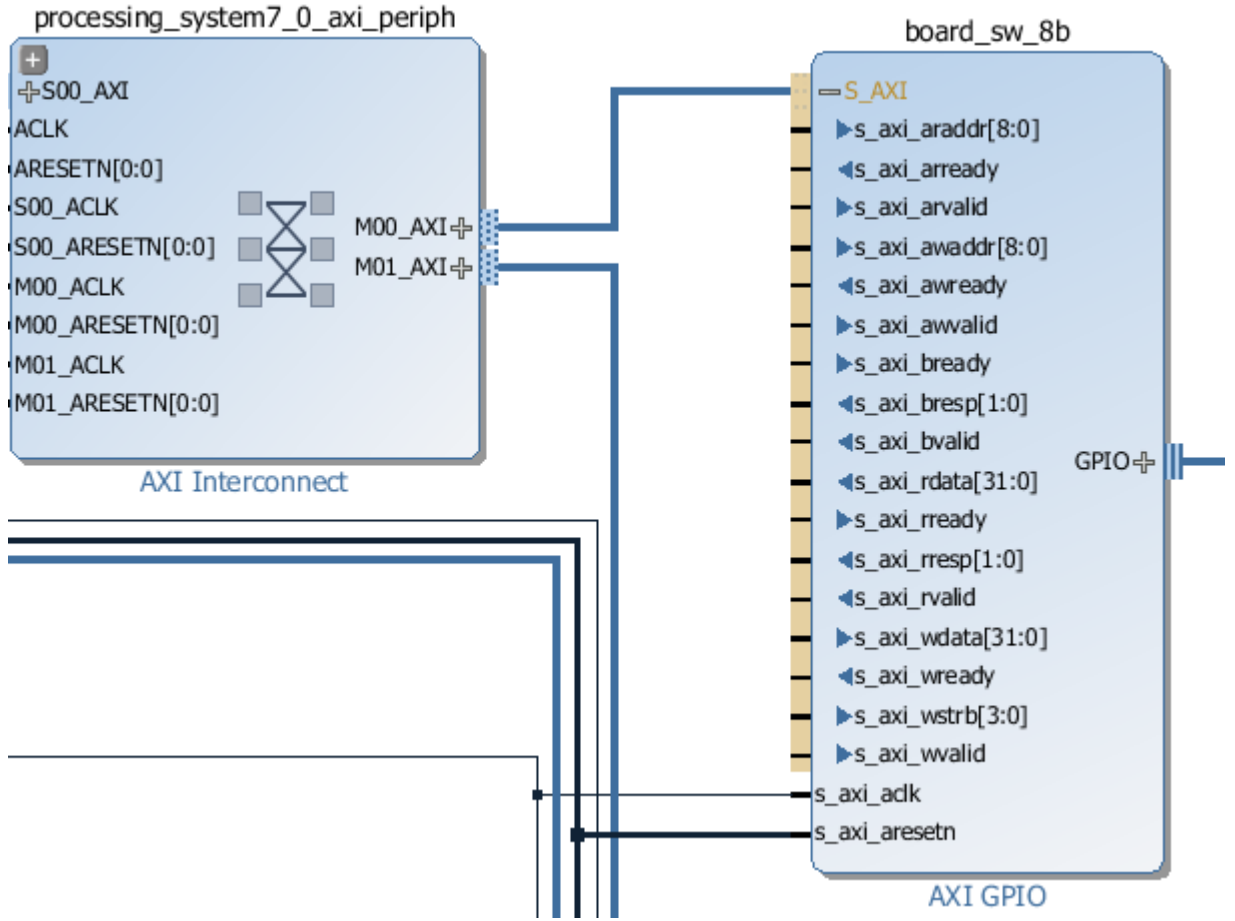


AXI4 – AXI Lite: Signals Available



AXI4-Lite Custom IP The VHDL Underneath

AXI4-Lite Signal Names



AXI4-Lite Signal Names

- During the creation of a Xilinx IP block, the Vivado tools can be used to map each AXI signal onto the signal name that the designer used when creating the IP
- However in order to make the life of the designer much easier, the signal names shown here are recommended when designing a custom AXI slave in VHDL
- Using these signal names will allow the Vivado design tools to automatically detect the signal names during the “create and package IP” step (described later on).

```
-- Clock and Reset
S_AXI_ACLK        : in std_logic;
S_AXI_ARESETN     : in std_logic;
-- Write Address Channel
S_AXI_AWADDR      : in  std_logic_vector(31 downto 0);
S_AXI_AWVALID     : in  std_logic;
S_AXI_AWREADY     : out std_logic;
-- Write Data Channel
S_AXI_WDATA       : in  std_logic_vector(31 downto 0);
S_AXI_WSTRB       : in  std_logic_vector(3 downto 0);
S_AXI_WVALID      : in  std_logic;
S_AXI_WREADY      : out std_logic;
-- Read Address Channel
S_AXI_ARADDR      : in  std_logic_vector(31 downto 0);
S_AXI_ARVALID     : in  std_logic;
S_AXI_ARREADY     : out std_logic;
-- Read Data Channel
S_AXI_RDATA       : out std_logic_vector(31 downto 0);
S_AXI_RRESP       : out std_logic_vector(1 downto 0);
S_AXI_RVALID      : out std_logic;
S_AXI_RREADY      : in  std_logic;
-- Write Response Channel
S_AXI_BRESP       : out std_logic_vector(1 downto 0);
S_AXI_BVALID      : out std_logic;
S_AXI_BREADY      : in  std_logic;
```

AXI4-Lite Address Decoding

- In previous versions of the Xilinx design flow (where PLB and OPB peripherals were typically used) it was necessary for each IP peripheral connected to the processor to individually decode all transactions that were presented by a master on the bus (“multi-drop”). it was the responsibility of each peripheral to accept or reject each bus transaction depending on the address that was placed on the address bus.

- With AXI4-lite, the interconnect does not use a multi-drop architecture, but uses a scheme where each transaction from the master(s) is specifically routed to a single slave IP depending on the address provided by the master.

- This premise permits a completely different design methodology to be adopted by the creator of a slave IP, in that any transactions which reach the slave’s interface ports are already known to be destined for that peripheral.

- The designer merely needs to decode enough of the incoming address bus to determine which of the registers in the slave IP should be read or written

My VHDL Code – Address Decoding

```
1  ----------------------------------------------------------------------
2  -- lab name: lab_custom_ip
3  -- component name: my_led_ip
4  -- author: cas
5  -- version: 1.0
6  -- description: simple logic to
7  ----------------------------------------------------------------------
8  library ieee;
9  use ieee.std_logic_1164.all;
10
11 entity lab_led_ip is
12
13   generic (
14     led_width : integer := 8);          -- 8 LEDs
15   port (
16     -- clock and reset
17     S_AXI_ACLK    : in std_logic;
18     S_AXI_ARESETN : in std_logic;
19     -- write data channel
20     S_AXI_WDATA   : in std_logic_vector(31 downto 0);
21     SLV_REG_WREN  : in std_logic;
22     -- address channel
23     AXI_AWADDR    : in std_logic_vector(3 downto 0);
24     -- my inputs / outputs --
25     -- output
26     LED           : out std_logic_vector(led_width-1 downto 0)
27     );
28 end entity lab_led_ip;
```

AXI4-Lite IP

```
30 architecture beh of lab_led_ip is
31
32 begin  -- architecture beh
33
34   process(S_AXI_ACLK, S_AXI_ARESETN)
35   begin
36     if(S_AXI_ARESETN='0')then
37       LED <= (others=>'0');
38     elsif(rising_edge(S_AXI_ACLK))then
39       if(SLV_REG_WREN='1' and AXI_AWADDR="0000") then
40         LED <= S_AXI_WDATA(led_width-1 downto 0);
41       end if;
42     end if;
43   end process;
44 end architecture beh;
```

Address Decode & Write Enable

AXI4-Lite Address Decoding – VHDL Example

```
local_address <= to_integer(unsigned(S_AXI_AWADDR(31 downto 0);

address_deco : process(local_address)
begin
   manual_mode_control_register_address_valid <= '0';
   manual_mode_data_register_address_valid    <= '0';
   servo_position_register_address_valid      <= (others => '0');
   low_endstop_register_address_valid         <= (others => '0');
   high_endstop_register_address_valid        <= (others => '0');
   case(local_address) is
      when 0 =>
           manual_mode_data_register_address_valid <= '1';
      when 4 =>
           manual_mode_control_register_address_valid <= '1';
      when 128 to 255 =>
           servo_position_register_address_valid <= '1';
      when 256 to 280 =>
           low_endstop_register_address_valid <= '1';
      when 281 to 511 =>
           high_endstop_register_address_valid <= '1';
      when others =>
           NULL;
   end case;
end process;
```

AXI4-Lite – Implementing Addressable Registers

- Using the address decoding scheme above, it is extremely simple to implement registers in VHDL which can receive data values written by a master on the AXI4-lite interconnect. The following extract of code shows how an individual register can be quickly and easily implemented (in this case mapped to BASEADDR + 0x00, as has been coded in the previous VHDL snippet).

```
manual_mode_control_register_process: process(S_AXI_ACLK)
begin
  if(rising_edge(S_AXI_ACLK)) then
    if (S_AXI_ARESETN = '1') then
      manual_mode_control_register <= (others => '0');
    else
      if(manual_mode_control_register_address_valid = '1') then
        manual_mode_control_register <= S_AXI_WDATA;
      end if;
    end if;
  end if;
end process manual_mode_control_register_process;
```

WriteTransaction

Read Transaction

```
send_data_to_AXI_RDATA: process(local_address, send_read_data_to_AXI,...)
begin
 S_AXI_RDATA <= (others => '0');
 if(local_address_valid = '1' and send_read_data_to_AXI = '1') then
   case(local_address) is
      when 0 =>
         S_AXI_RDATA <= manual_mode_control_register;
      when 4 =>
         S_AXI_RDATA <= manual_mode_data_register;
      when ...
      ....

      when others => NULL;
   end case;
 end if;
end process;
```

AXI4-Lite – Controlling AXI Transactions

- Usually there is a need to implement some logic to control the AXI transactions.
- This can be achieved by the use of a finite state machine. Here, it is an example of a (simplified) state machine, showing the implementation of some of the states, and how a read transaction might be handled in the design.
- The example is not designed to cover all of the states required to implement read and write transactions, but should help to illustrate a style of coding suitable for creating the FSM.

```
state_machine_update : process (S_AXI_ACLK)
    begin
        if S_AXI_ACLK'event and S_AXI_ACLK = '1' then
            if Local_Reset = '1' then
                current_state <= reset;
            else
                current_state <= next_state;
            end if;
        end if;
    end process;

state_machine_decisions : process (current_state, combined_S_AXI_AWVALID_S_AXI_ARVALID, S_AXI_ARVALID,
S_AXI_RREADY, S_AXI_AWVALID, S_AXI_WVALID, S_AXI_BREADY, local_address, ...(signals removed)... )
begin
case current_state is
        when reset =>
                next_state <= idle;
        when idle =>
                next_state <= idle;
                case combined_S_AXI_AWVALID_S_AXI_ARVALID is
                        when "01" => next_state <= read_transaction_in_progress;
                        when "10" => next_state <= write_transaction_in_progress;
                        when others => NULL;
                end case;

        when read_transaction_in_progress =>
                next_state <= read_transaction_in_progress;
                S_AXI_ARREADY <= S_AXI_ARVALID;
                S_AXI_RVALID <= '1';
                S_AXI_RRESP <= "00";
                if S_AXI_RREADY = '1' then
                    next_state <= complete;
                end if;

                case (local_address) is
                        when 0 => S_AXI_RDATA <= manual_mode_control_register;
                        when 4 => S_AXI_RDATA <= manual_mode_data_register;
                        when others =>
                            if (local_address >=8 and local_address < (8+((NUMBER_OF_SERVOS-1)*4))) then
                               S_AXI_RDATA <= servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4) &
                                              servo_position_register_array((local_address-8)/4);
                            else
                                S_AXI_RDATA <= (others => '0');
                            end if;
                    end case;
        ... {additional code removed from here} ...
        when complete =>
                case combined_S_AXI_AWVALID_S_AXI_ARVALID is
                        when "00" => next_state <= idle;
                        when others => next_state <= complete;
                end case;


        when others => next_state <= reset;
end case;
end process;
```

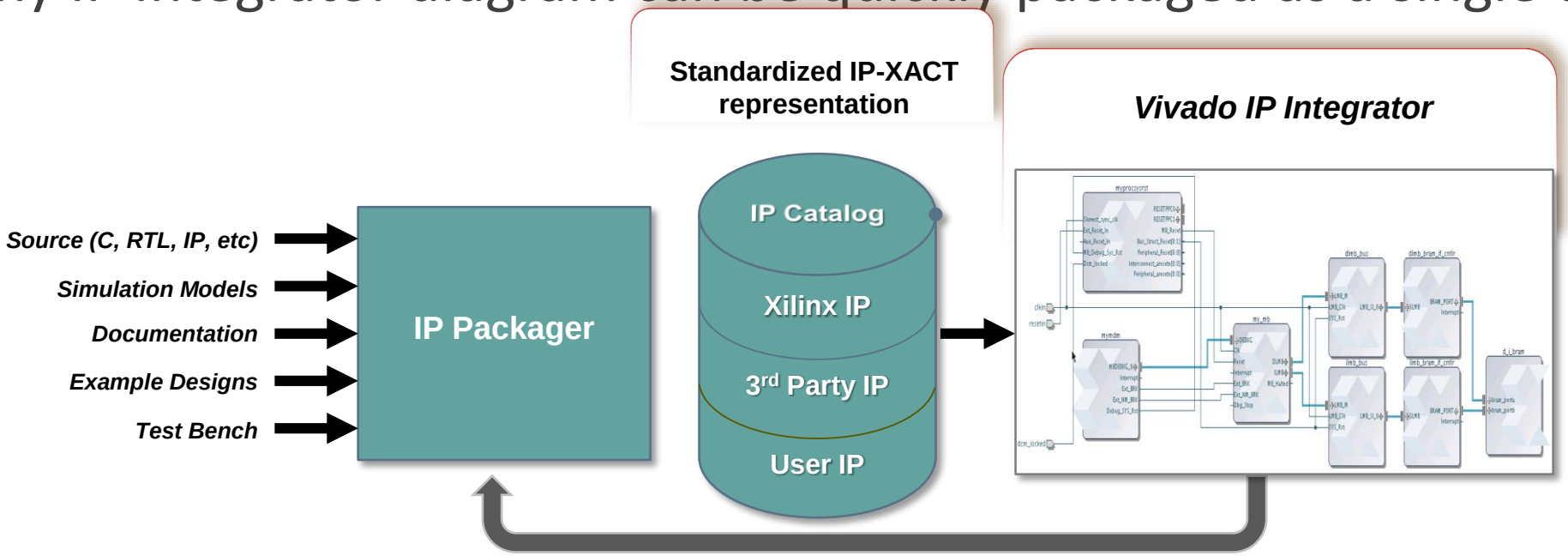
Custom AXI IPs
Use of MGPO & MGP1

Reusing Your IP

IP from many sources can be packaged and made available in Vivado

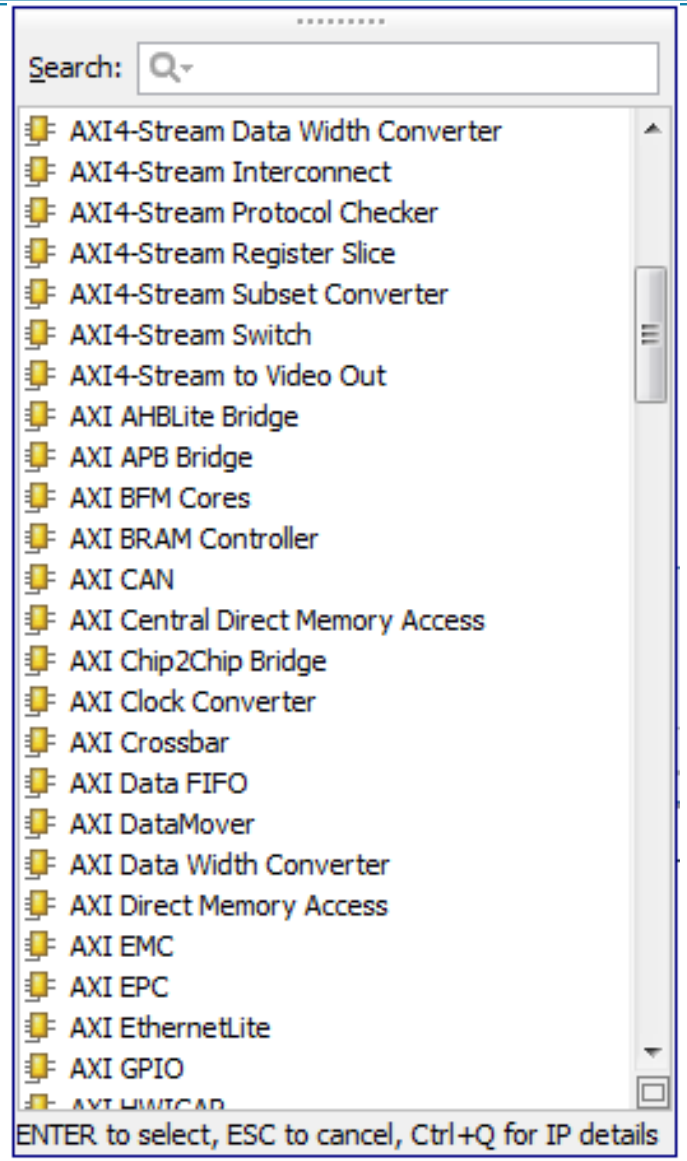
All IP available in the Vivado IP Catalog can be used to create IP Integrator designs

Any IP Integrator diagram can be quickly packaged as a single complex IP



IP Catalog Main Features

- Consistent, easy access
- Support for multiple physical locations, including shared network drives
- Access to the latest version of Xilinx-delivered IP
- Access to IP customization and generation using the Vivado IDE
- IP example designs
- Catalog filter options that let you filter by Supported Output Products, Supported Interfaces, Licensing, Provider, or Status



IP Packager

- The IP Packager allows a core to be packaged and included in the IP Catalog, or for distribution
- IP-XACT
- Complete set of files include
  - Source code, Constraints, Test Benches (simulation files), documentation
- IP Packager can be run from Vivado on the current project, or on a specified directory

IP-XACT

- Industry Standard (IEEE) XML format to describe IP using meta-data
  - Ports
  - Interfaces
  - Configurable Parameters
  - Files, documentation

- IP-XACT only describes high level information about IP, not low level description, so does not replace HDL or Software.
- Enables automatic connection, configuration and integration
- Enables integration of 3rd Party IP
  - (And Export of your own IP)

Transfer Data from ARM Bus to PL Logic

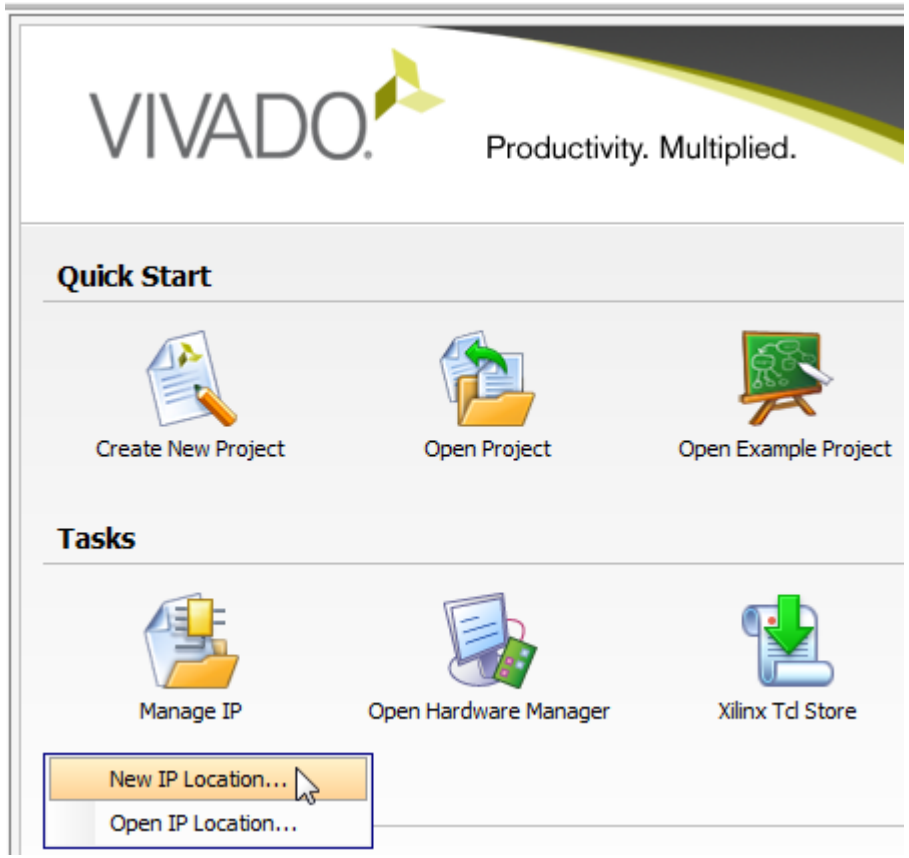
- Use of MGPO & MGP1 to transfer data from the PS to a AXI Slave in the PL
- Write transaction to PL through MGP0
- Read transaction from PL through MGP1

IP Packager

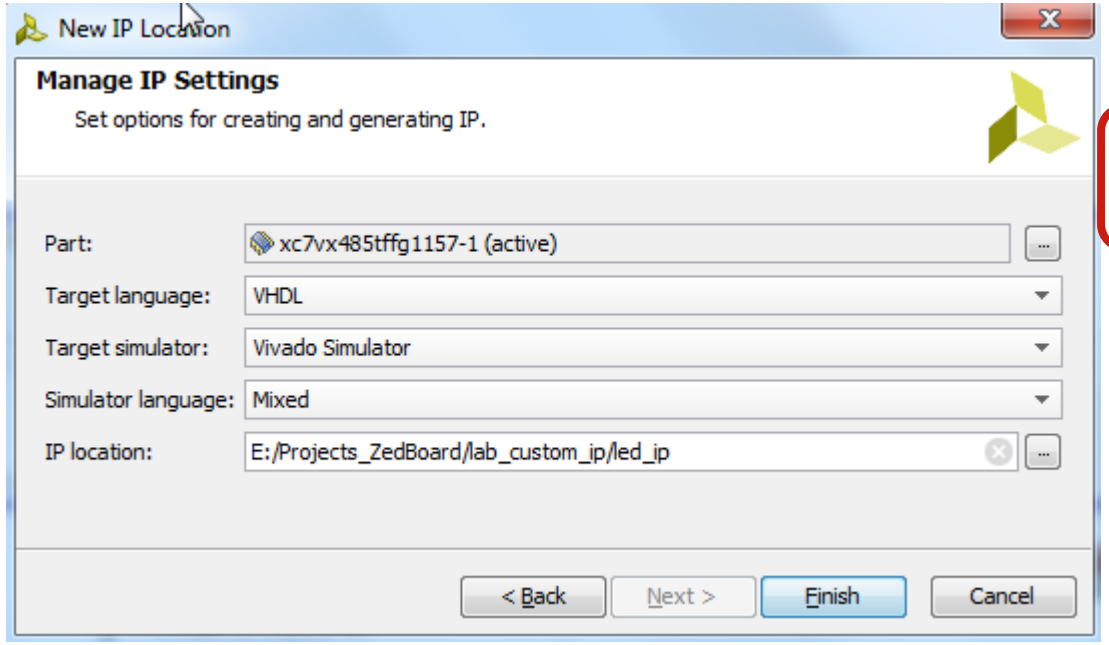
- The IP Packager allows a core to be packaged and included in the IP Catalog, or for distribution
- IP-XACT
- Complete set of files include
  - Source code, Constraints, Test Benches (simulation files), documentation
- IP Packager can be run from Vivado on the current project, or on a specified directory

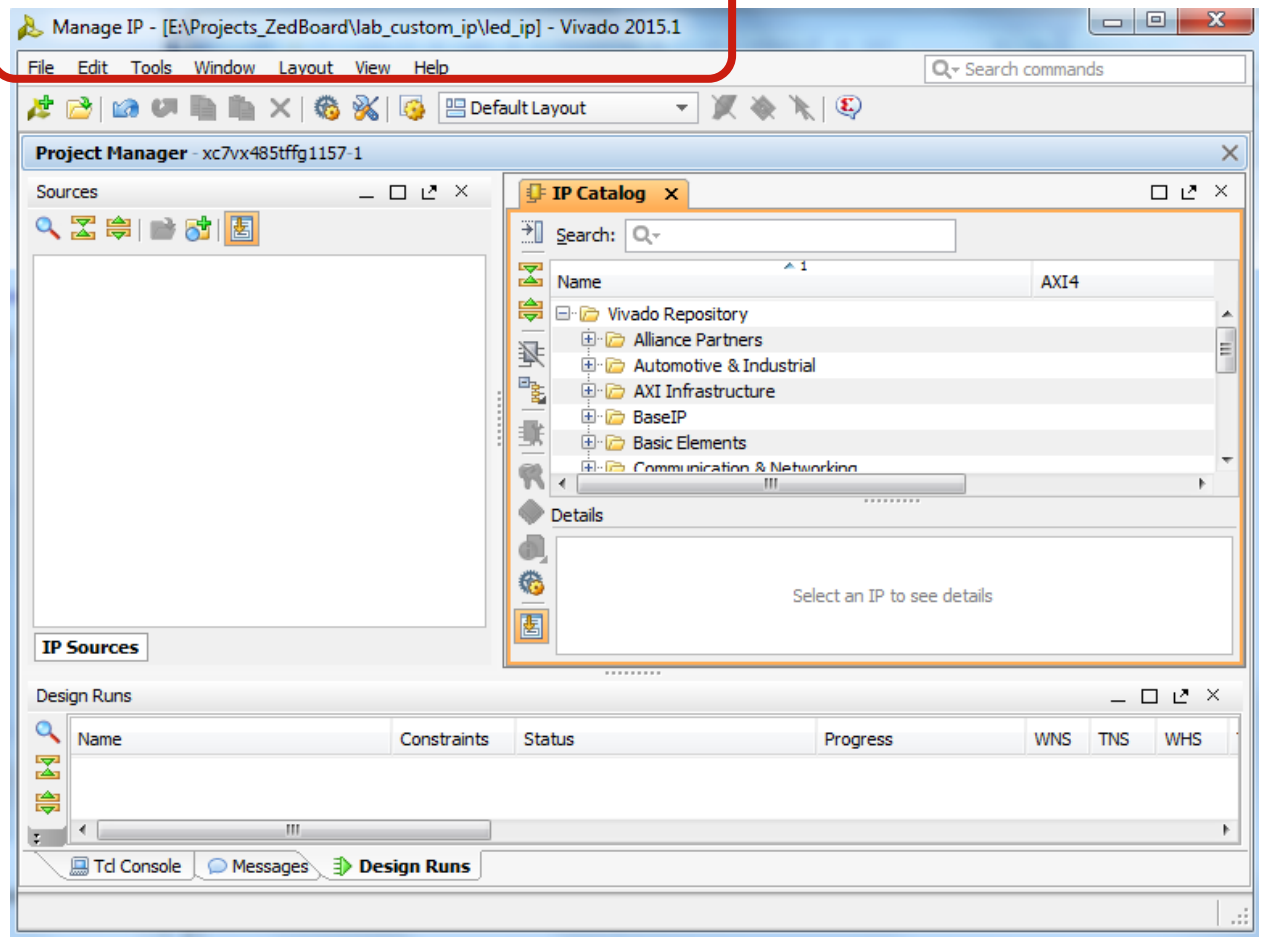
IP Manager

- **Create and Package IP Wizard**
- **Generates HDL template for**
  - Slave/Master
    - AXI Lite/Full/Stream
- **Optionally Generates**
  - Software Driver
    - Only for AXI Lite and Full slave interface
  - Test Software Application
  - AXI4 BFM Example

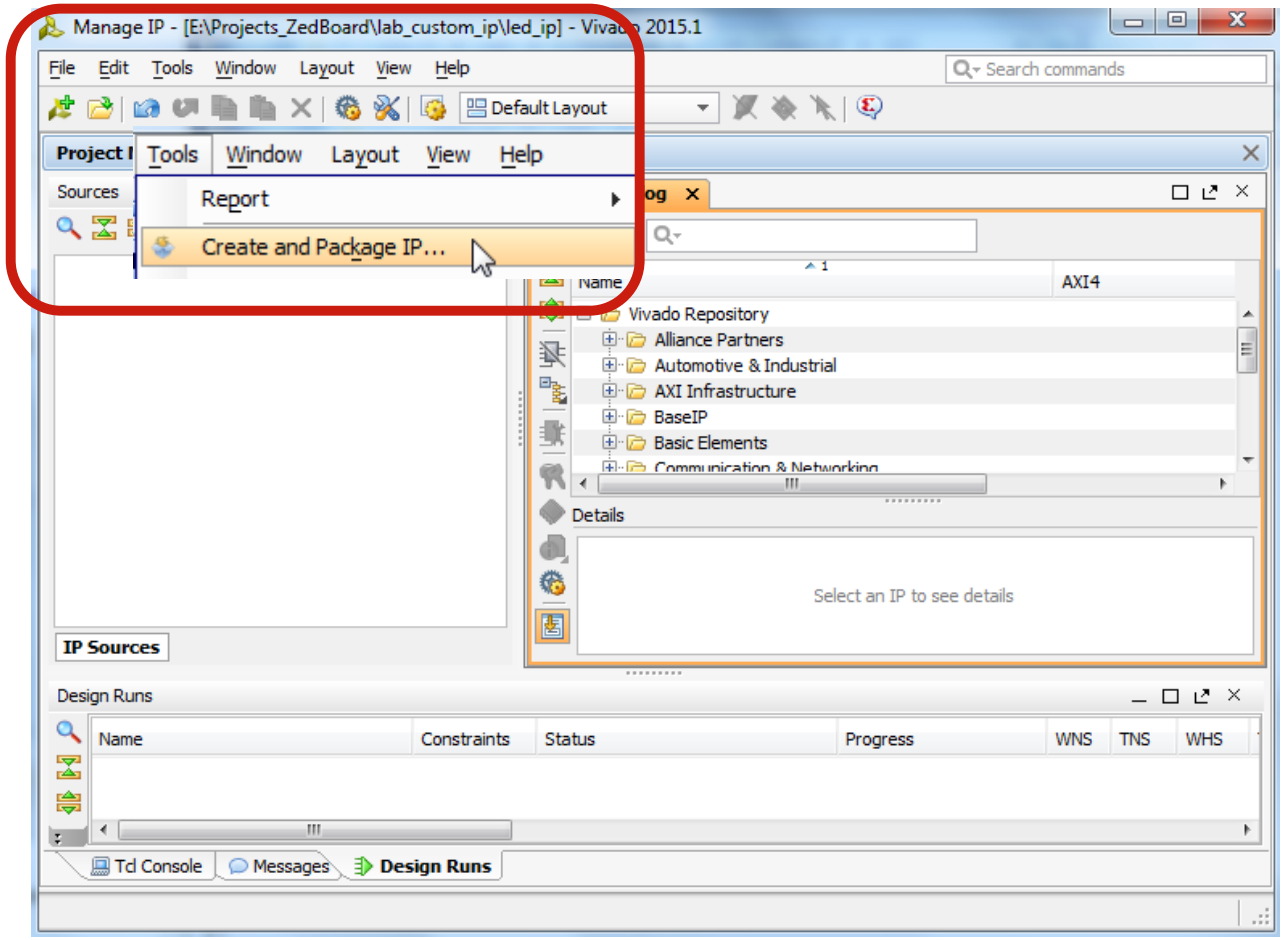


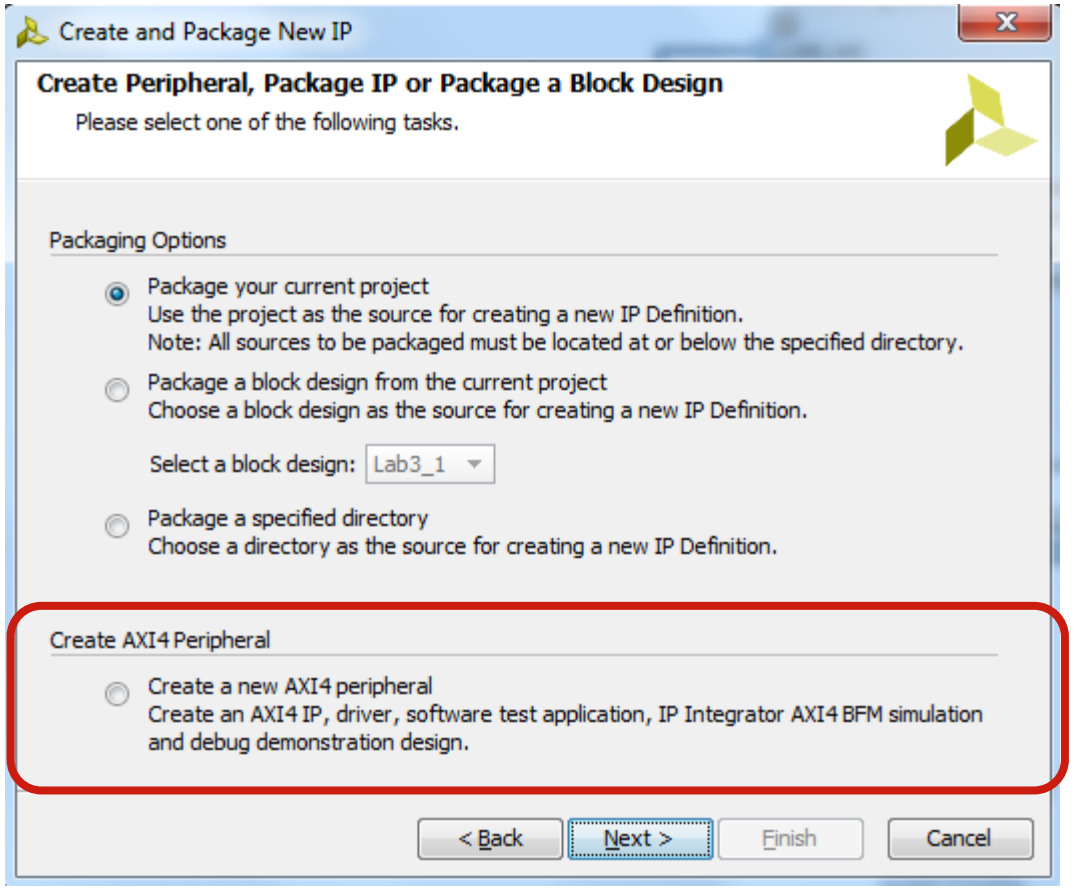
Create Custom AXI4 IP



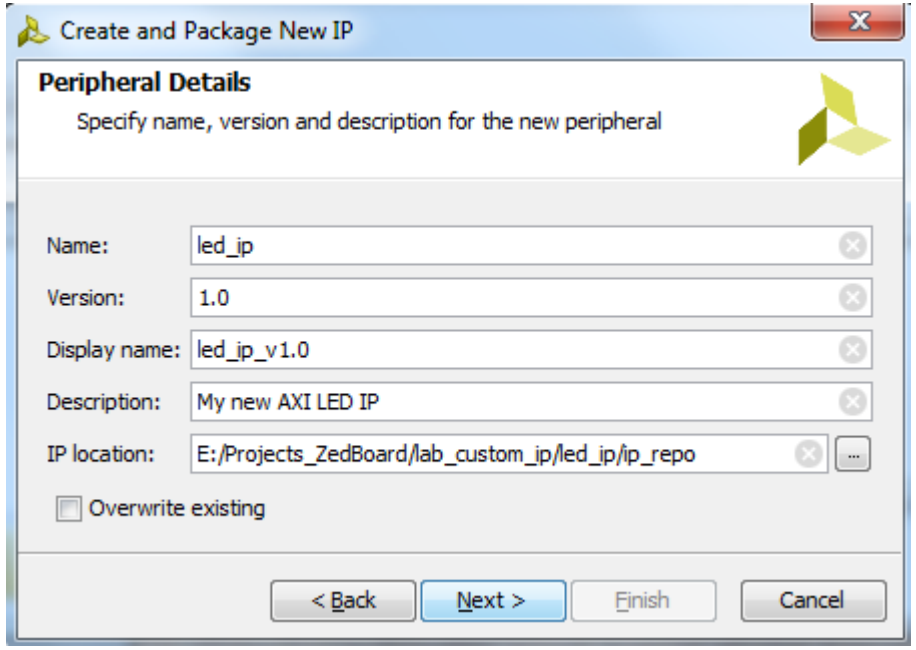


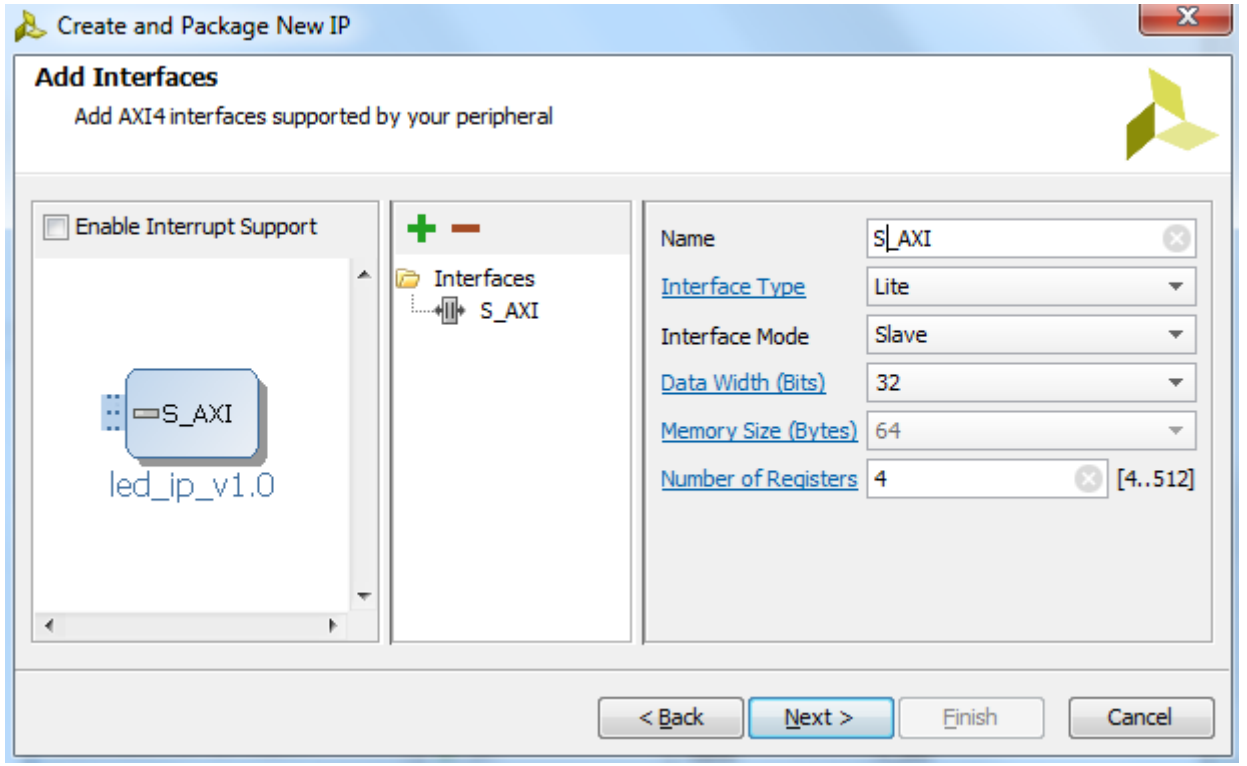
Create Custom AXI4 IP



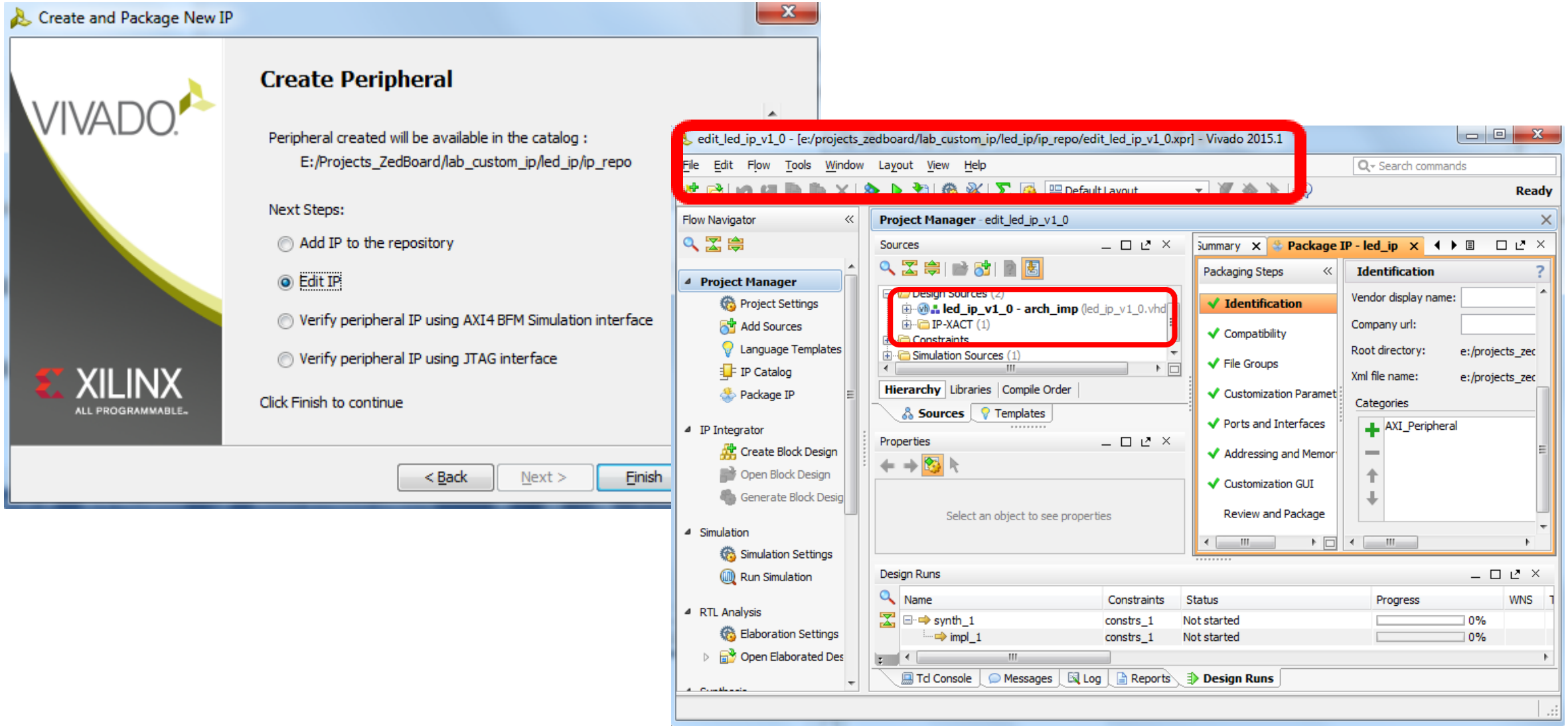


Create Custom AXI4 IP

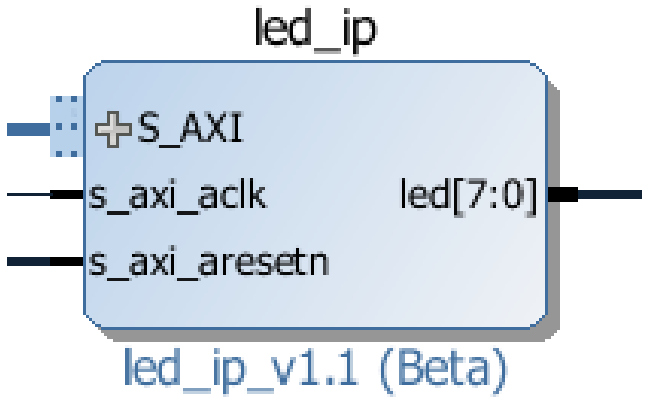




Edit Created Custom AXI4 IP



Edit Created Custom AXI4 IP



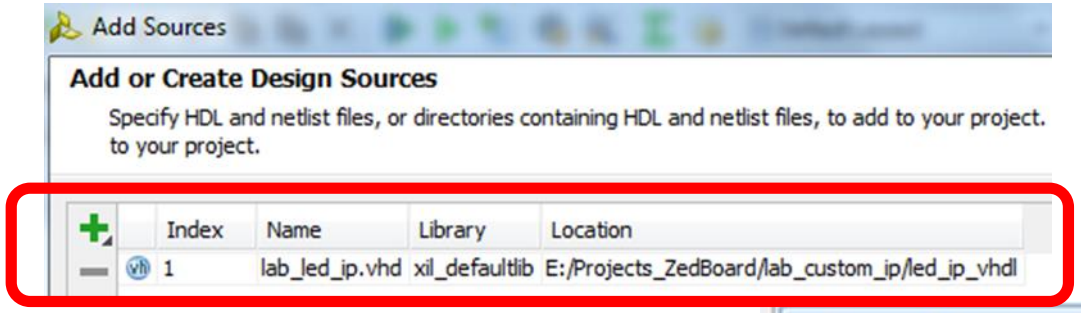
led_ip_v1_0.vhd | led_ip_v1_0_S_AXI.vhd | lab_l

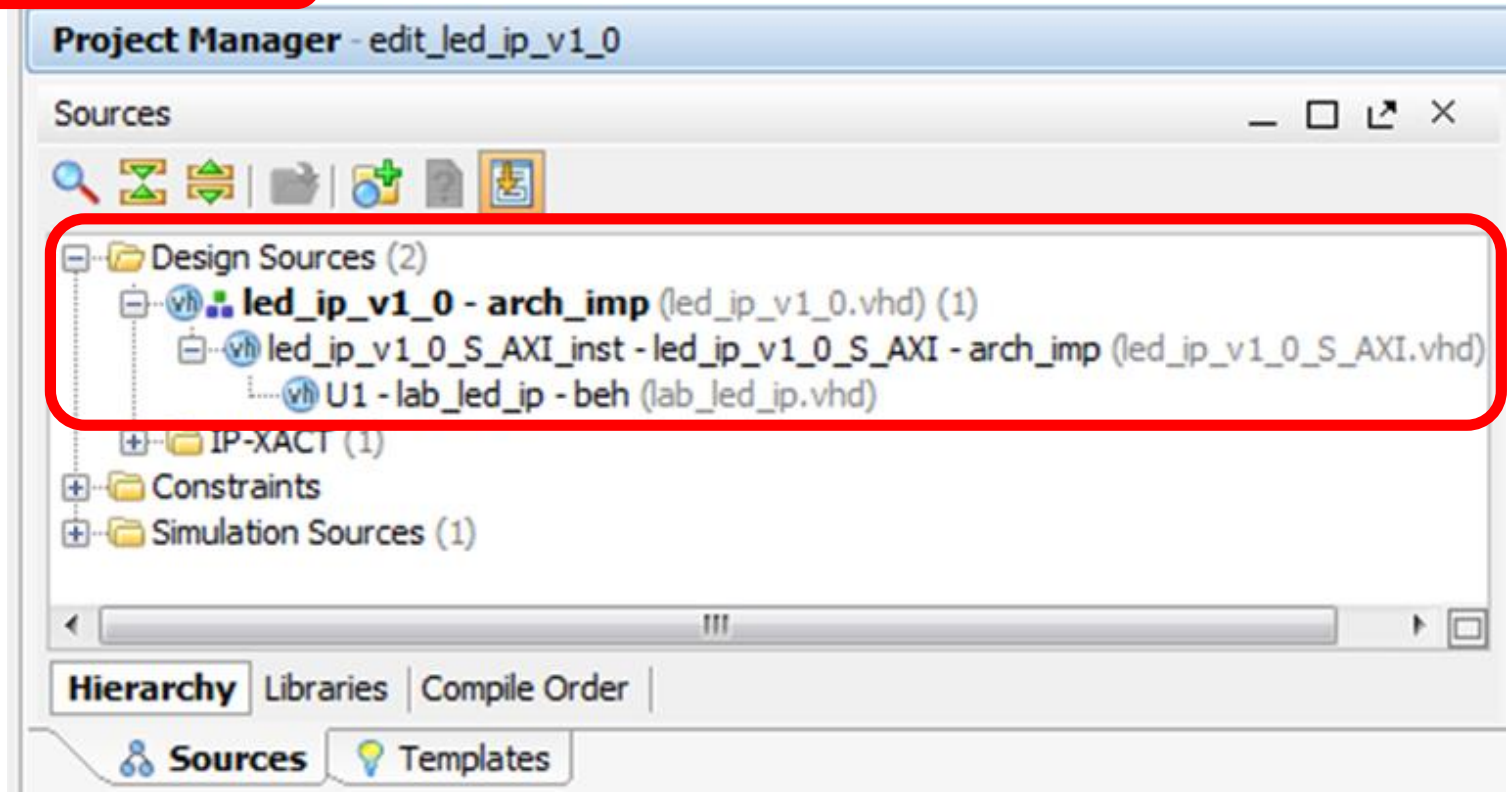
e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0/hdl/led_ip_v1_0.vhd

```
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity led_ip_v1_0 is
6      generic (
7          -- Users to add parameters here
8          LED_WIDTH : integer := 8;
9          -- User parameters ends
10         -- Do not modify the parameters beyond this line
11
12
13         -- Parameters of Axi Slave Bus Interface S_AXI
14         C_S_AXI_DATA_WIDTH  : integer   := 32;
15         C_S_AXI_ADDR_WIDTH  : integer   := 4
16             );
17     port (
18         -- Users to add ports here
19         led : out std_logic_vector(LED_WIDTH-1 downto 0);
20         -- User ports ends
```

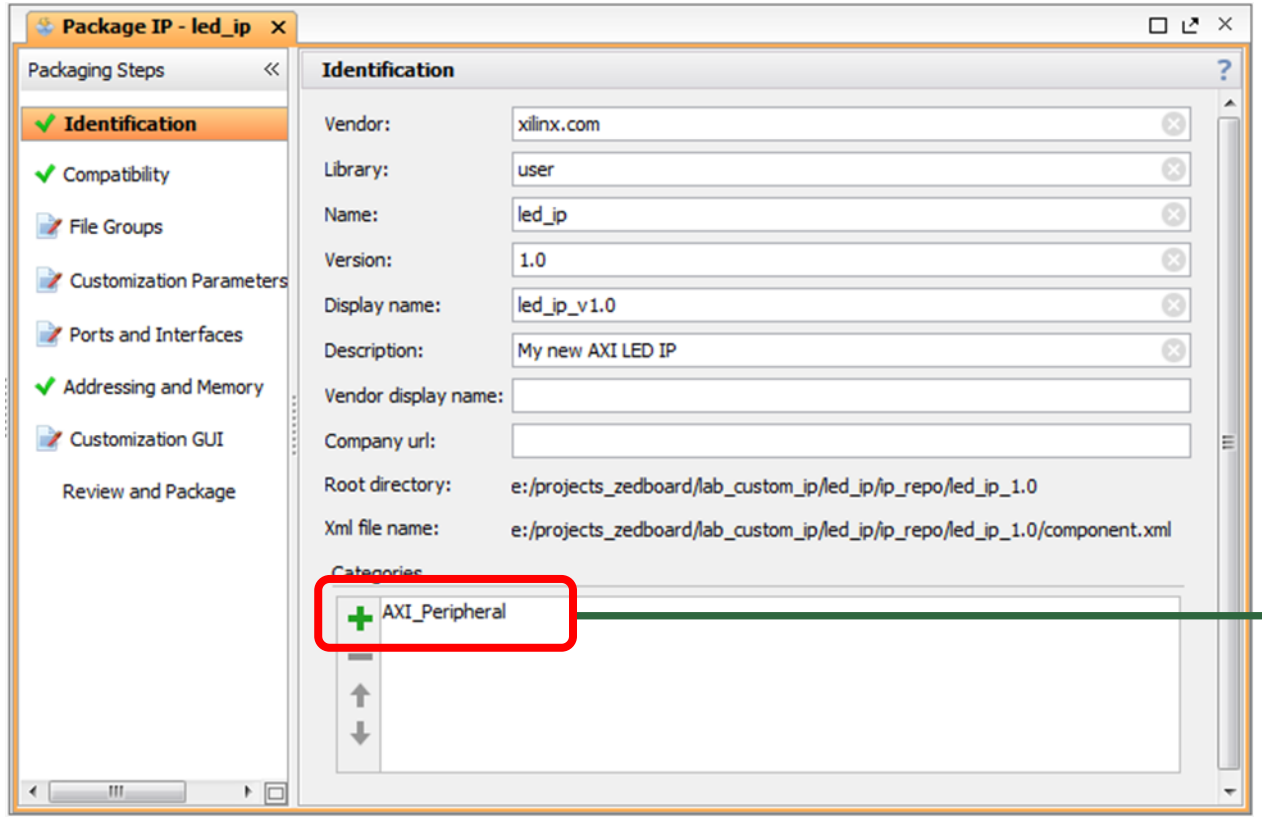
```
381     -- Add user logic here
382     U1: entity work.lab_led_ip generic map(led_width => led_width)
383                     port map(
384                         S_AXI_ACLK    => S_AXI_ACLK,
385                         SLV_REG_WREN  => SLV_REG_WREN,
386                         AXI_AWADDR    => AXI_AWADDR,
387                         S_AXI_WDATA   => S_AXI_WDATA,
388                         S_AXI_ARESETN => S_AXI_ARESETN,
389                         LED           => LED );
390         -- User logic ends
```

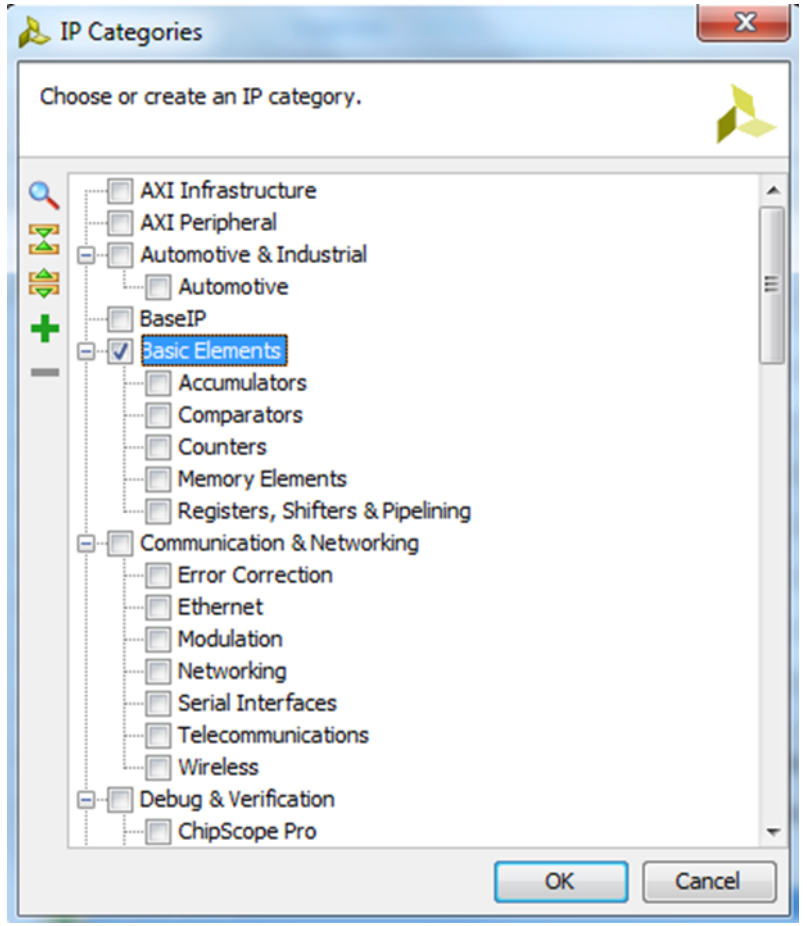
Hierarchy of My IP



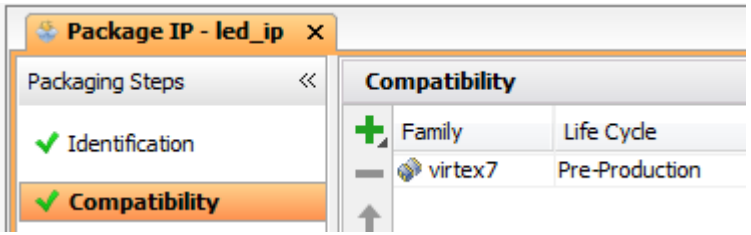


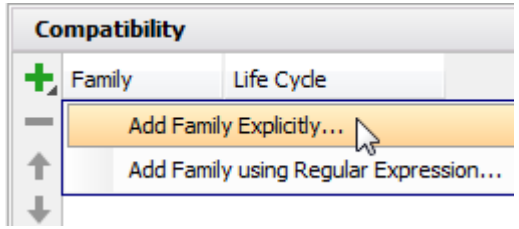
Package the IP

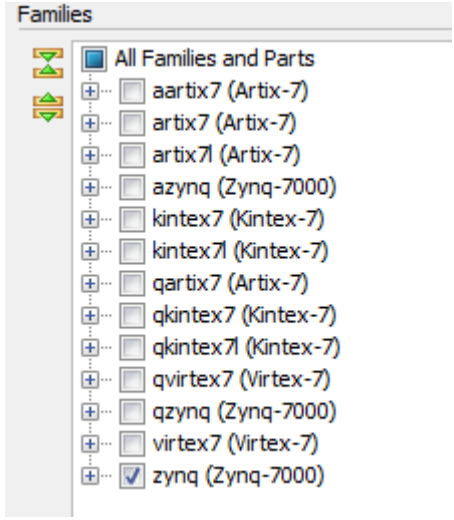




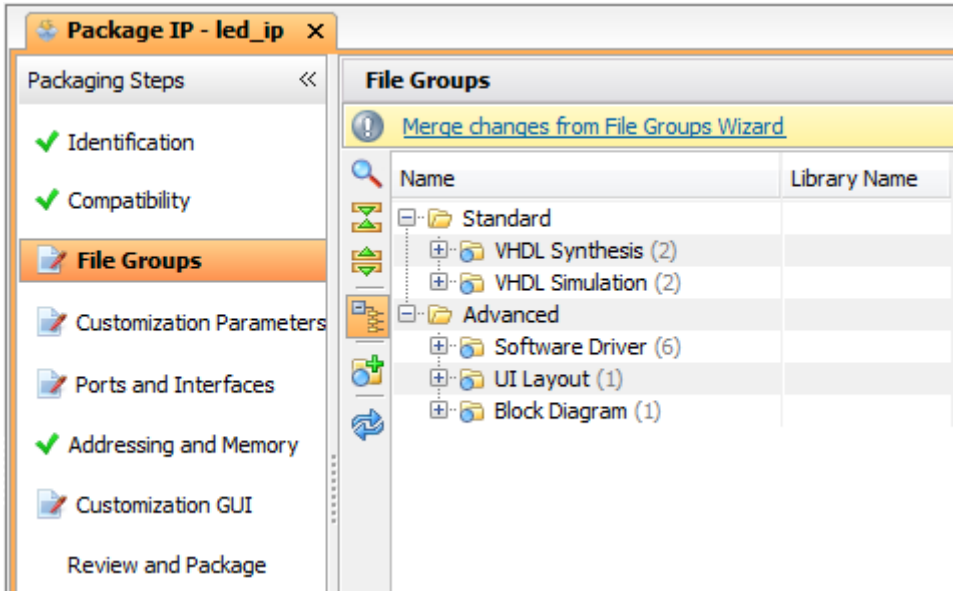
Compatibility of My IP



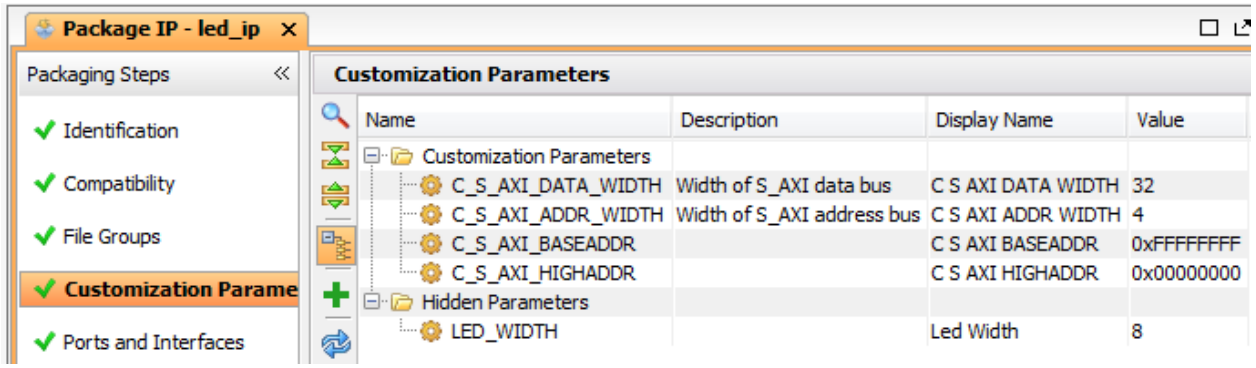


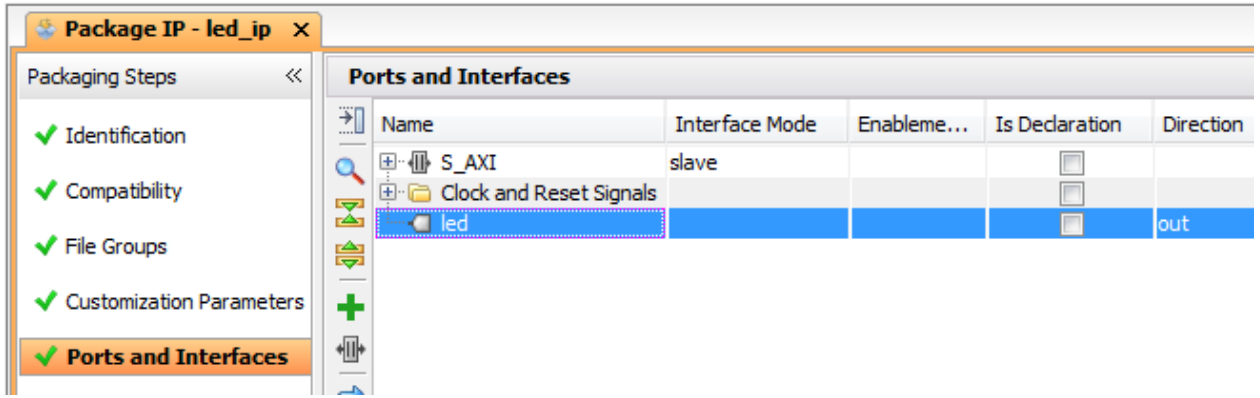


Updating Generated Files



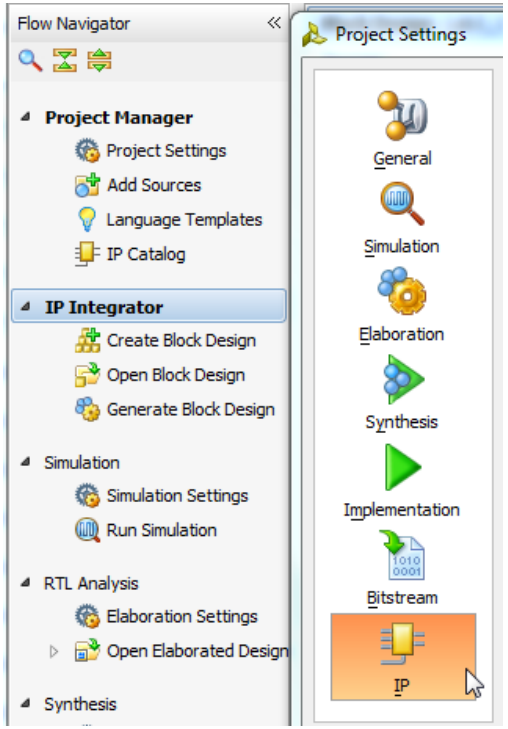
Checking Parameters and I/O Ports

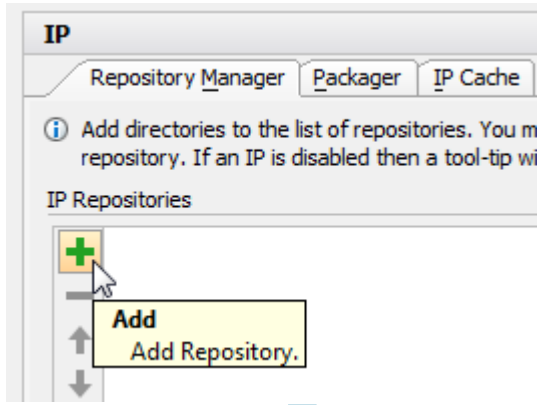


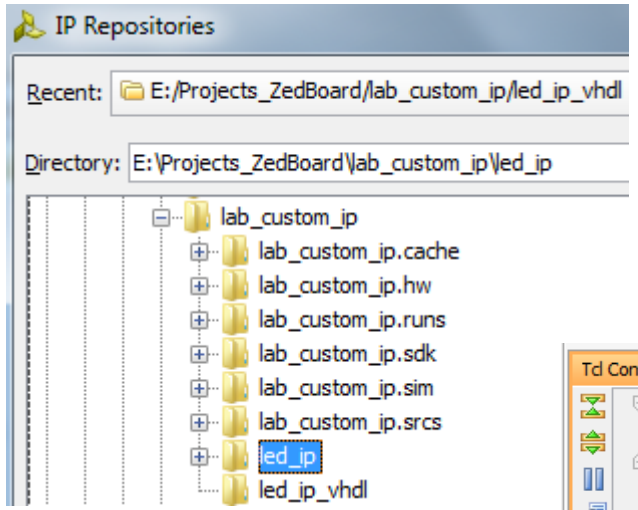


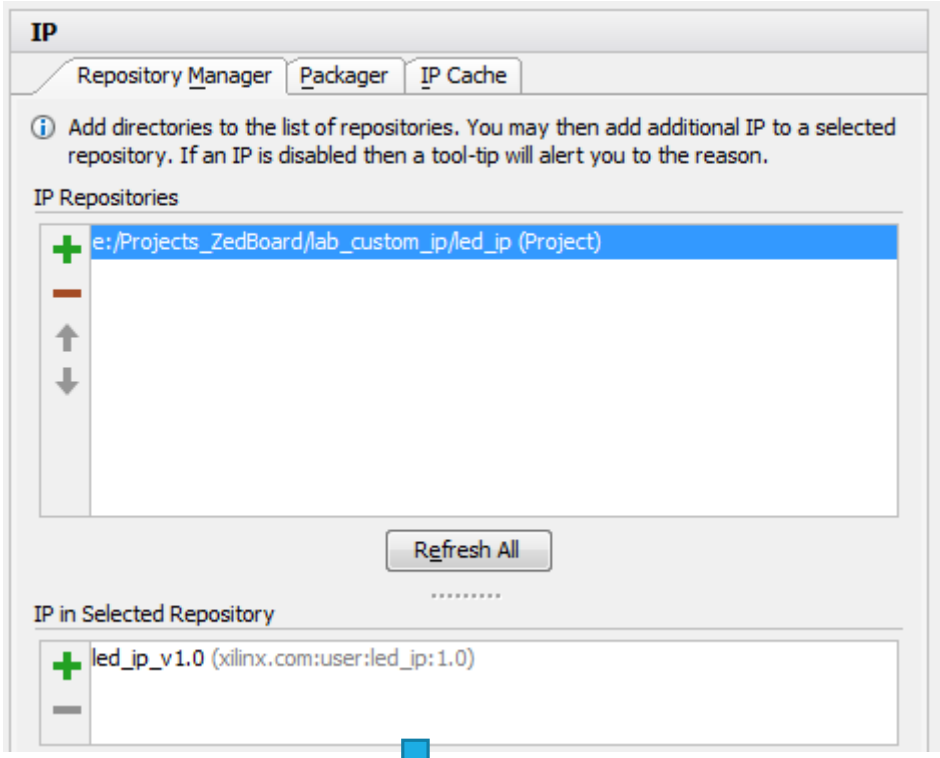
(This ends the Works on the edit_ip environment)

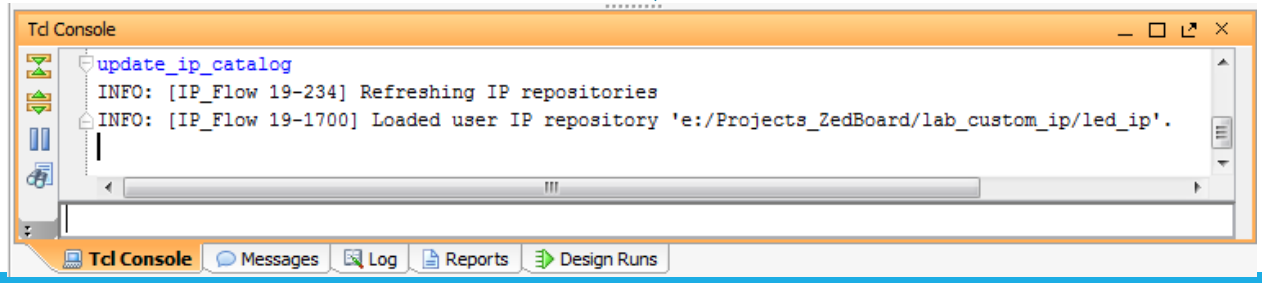
Add My IP to the Repository





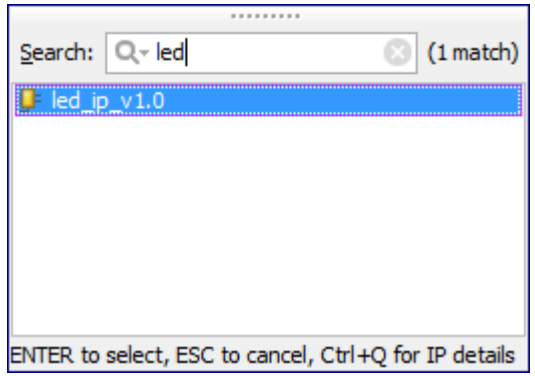


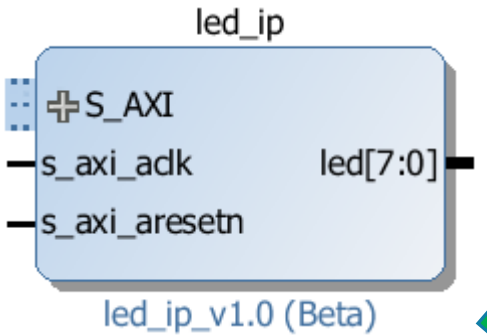




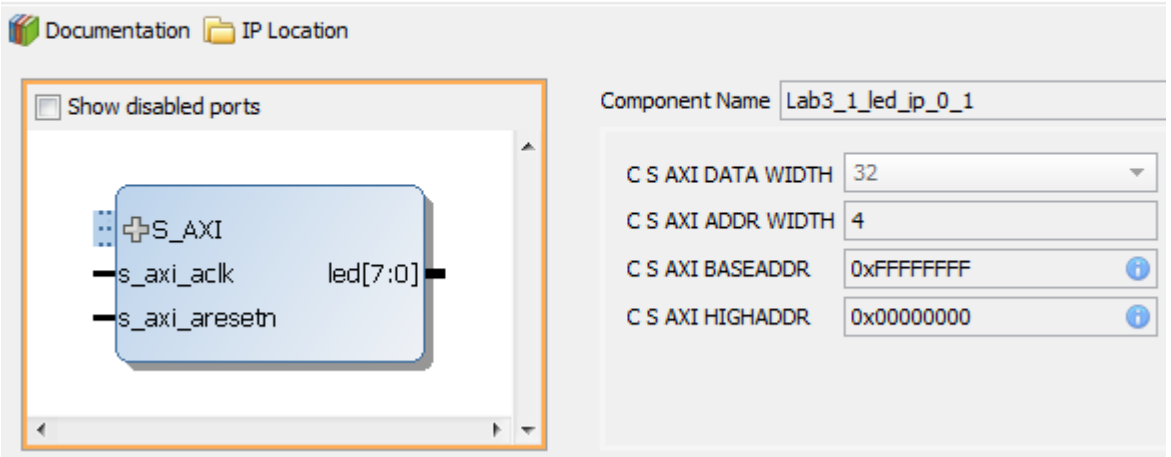
These steps shold be done in the Vivado Environment

led_ip Now Available in the IP List





led_ip_v1.0 (1.0)



Files created

component.xml
- IP XACT description

.bd
- Block Diagram tcl file

drivers
- SDK and software files (c code)
- Simple register/memory read/write functionality
- Simple SelfTest code

hdl
- Verilog/VHDL source

xgui
- GUI tcl file

```
XStatus LED_IP_Reg_SelfTest(void * baseaddr_p)
{
    ------------------------------------------------------------

    xil_printf("******************************\n\r");
    xil_printf("* User Peripheral Self Test\n\r");
    xil_printf("******************************\n\n\r");

    /*
     * Write to user logic slave module register(s) and read back
     */
    xil_printf("User logic slave module test...\n\r");

    for (write_loop_index = 0 ; write_loop_index < 4; write_loop_index++)
      LED_IP_mWriteReg (baseaddr, write_loop_index*4, (write_loop_index+1
      READ_WRITE_MUL_FACTOR);
    for (read_loop_index = 0 ; read_loop_index < 4; read_loop_index++)
      if ( LED_IP_mReadReg (baseaddr, read_loop_index*4) != (read_loop_in
      +1)*READ_WRITE_MUL_FACTOR){
        xil_printf ("Error reading register value at address %x\n", (int)
        baseaddr + read_loop_index*4);
        return XST_FAILURE;
      }
```

Steps for Custom IP - Summary

- **Create an AXI Slave/Master IP Core**
  - Use the Wizard to generate an AXI Slave/Master ‘device’
  - Set the number of registers
- **Building the Complete Zynq system**
  - Creating a Zynq based System
  - Adding the necessary Ips
  - Adding our custom AXI IP Core
  - Edit Address Space
- **Customize the IP Core**
  - File structure of the IP Cores
  - Edit the HDL generated by the wizard
  - Updating the IP Core and repack
  - Rebuild the system
- **Programming the device**
  - Open SDK. Creating a Application and BSP project
  - Write the “C” code to Wr/Rd the IP Cores registers
  - Edit Space