Joint ICTP-IAEA School on Detector Signal Processing and Machine Learning for Scientific Instrumentation and Reconfigurable Computing

The Abdus Salam International Centre for Theoretical Physics

IAEA International Atomic Energy Agency

unesco United Nations Educational, Scientific and Cultural Organization

Peripheral Interfaces & IP Integration

Cristian Sisterna

Senior Associate, ICTP-MLAB

Professor at Universidad Nacional San Juan- Argentina

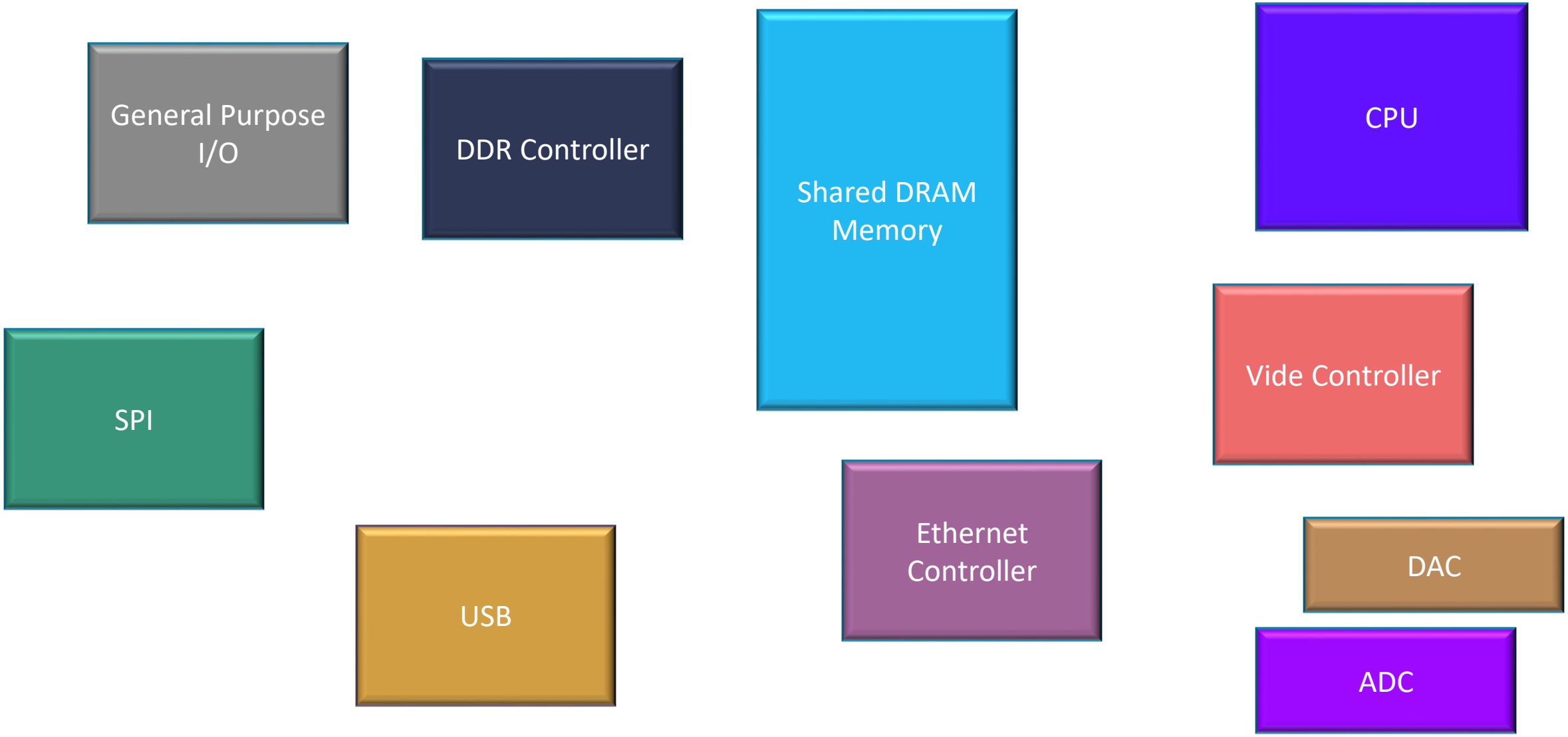
Agenda

- 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
- Design and Implementation of a Custom IP Core

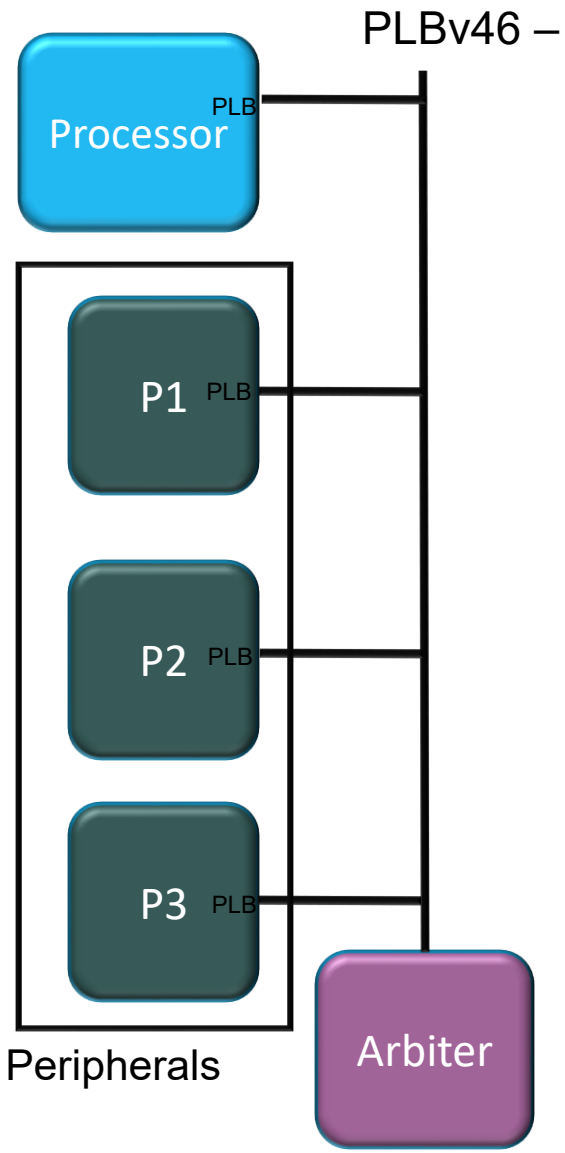
Need to Understand Device's Connectivity

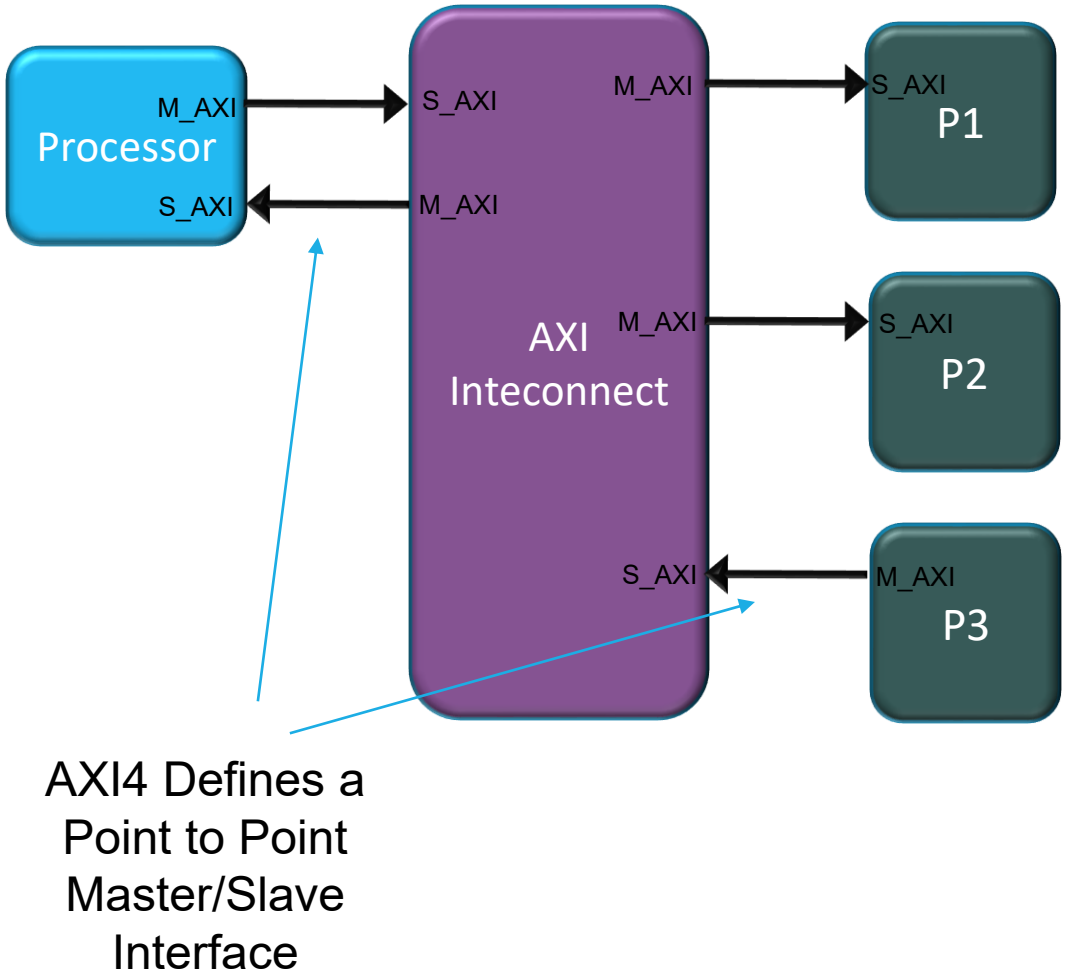
- There is a need to get familiar with the way that different devices communicate each other in an Embedded System like a Zynq based system
- Learning and understanding the communication among devices will facilitate the design of Zynq based systems
- All the devices in a Zynq system communicate each 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





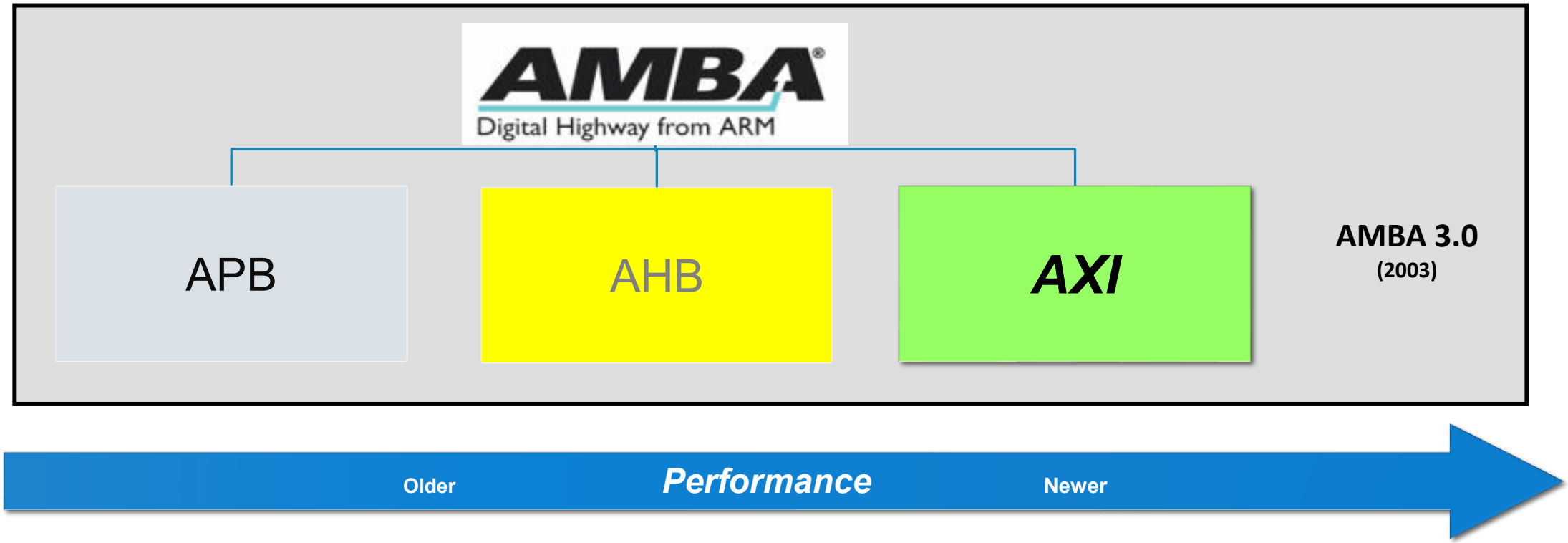
Connectivity -> Standard

- **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 SoPC Interfaces

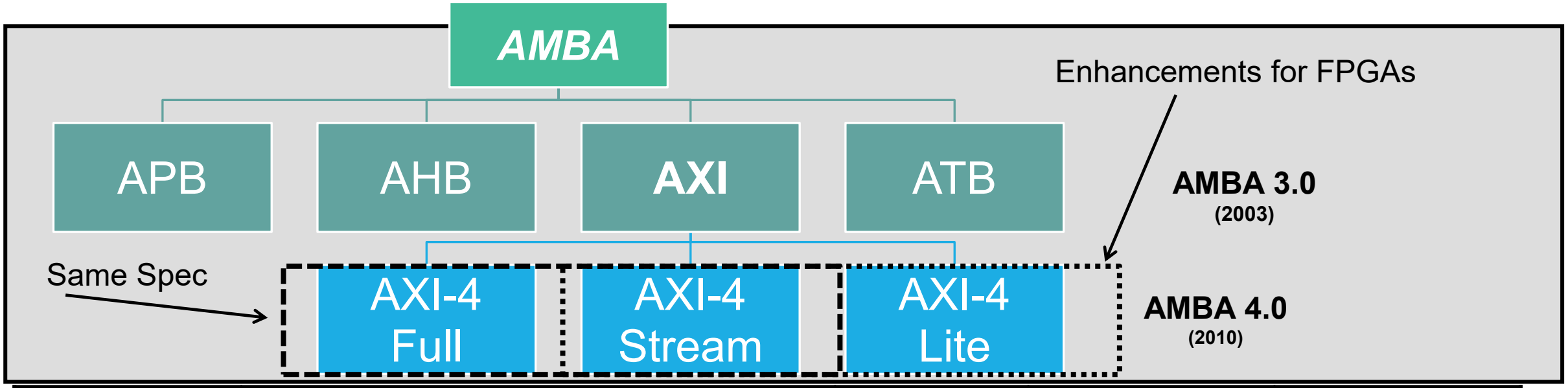
- **Core Connect (IBM)**
  - PLB/OPB (Power PC-FPGA bus interface)
- **WishBone**
  - OpenCore Cores
- **AXI**
  - ARM standard (more to come . . .)

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 Full	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 Interconnect

AXI is an interconnect system used to tie processors to peripherals

- **AXI Full:** 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 – 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 as master or as 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

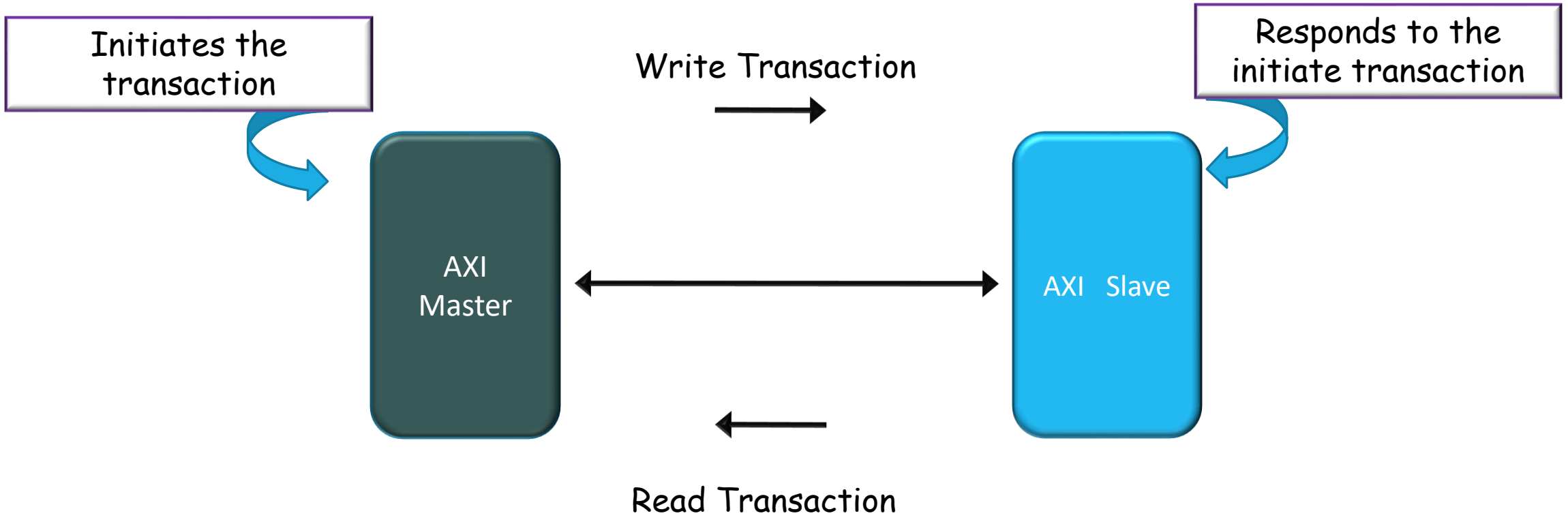
Transaction

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

Burst

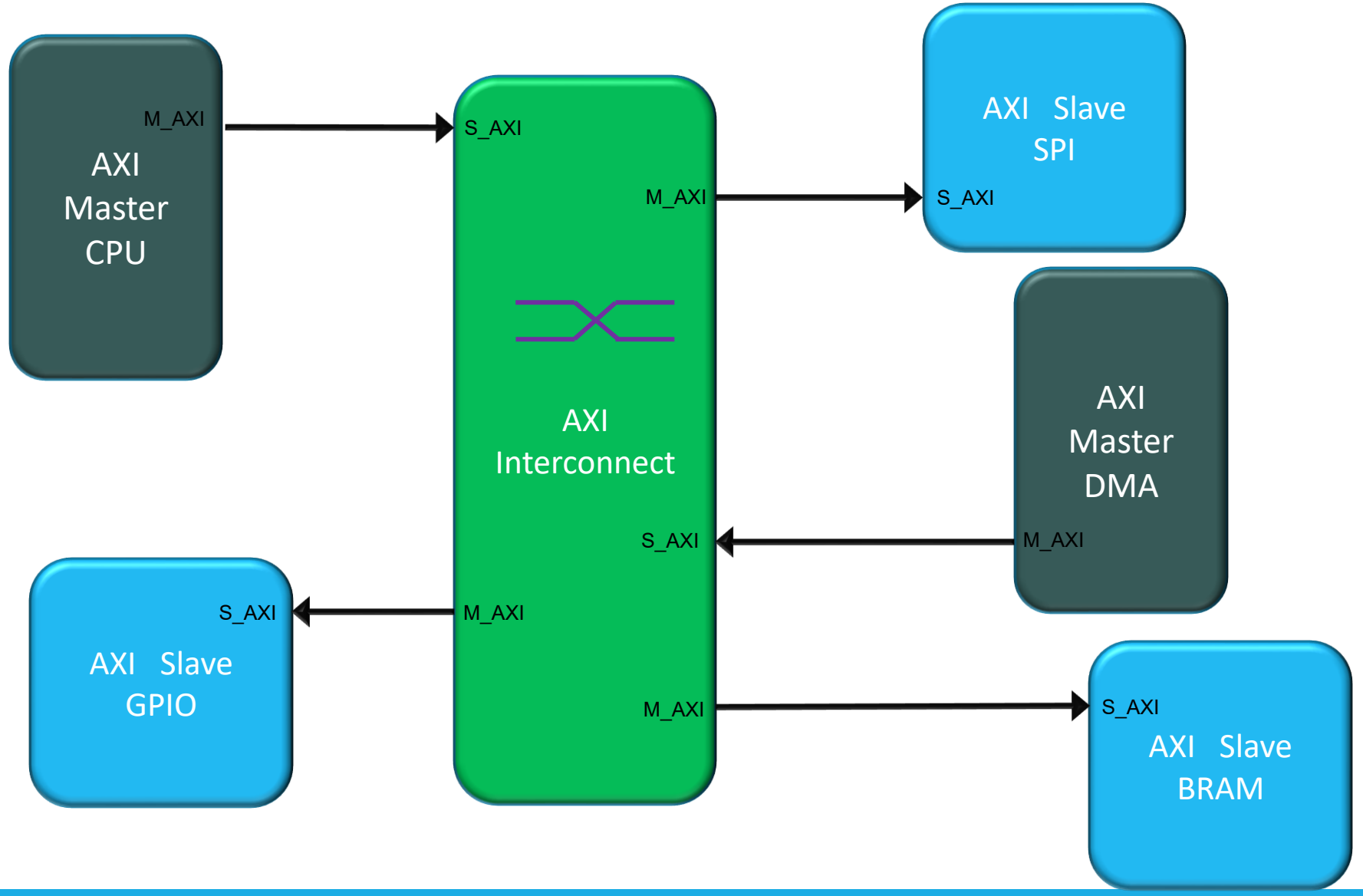
- Transaction that consists of more than one transfer

AXI Transactions / Master-Slave



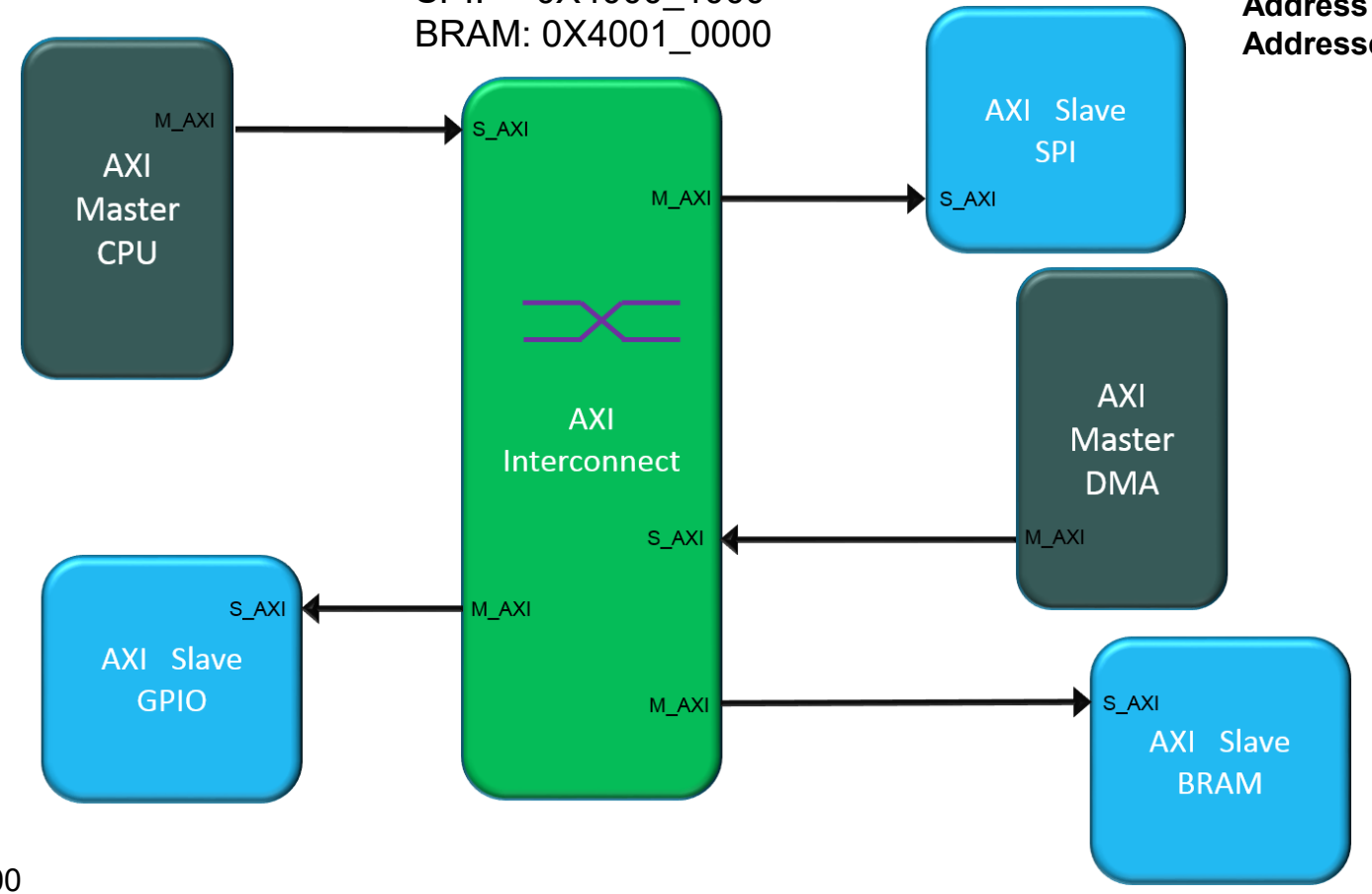
Transactions: transfer of data from one point on the hardware to another point

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

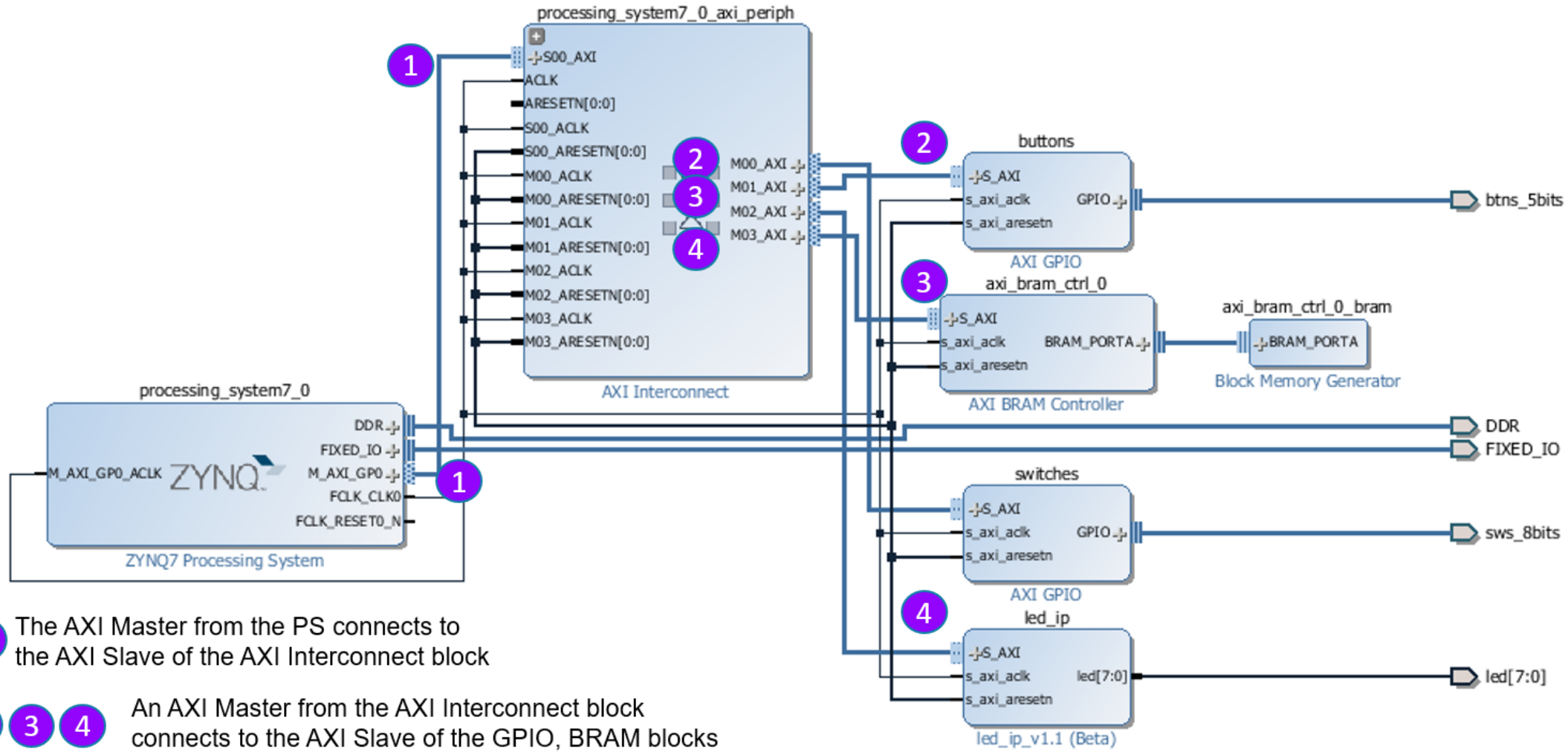
Address Range: 4K
Address Offset: 0X4000_0000
Addresses: 0X4000_0000 – 0X4000_0FFF

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 (pipelining), Input/Output FIFOs
- Clock Domains Transfer

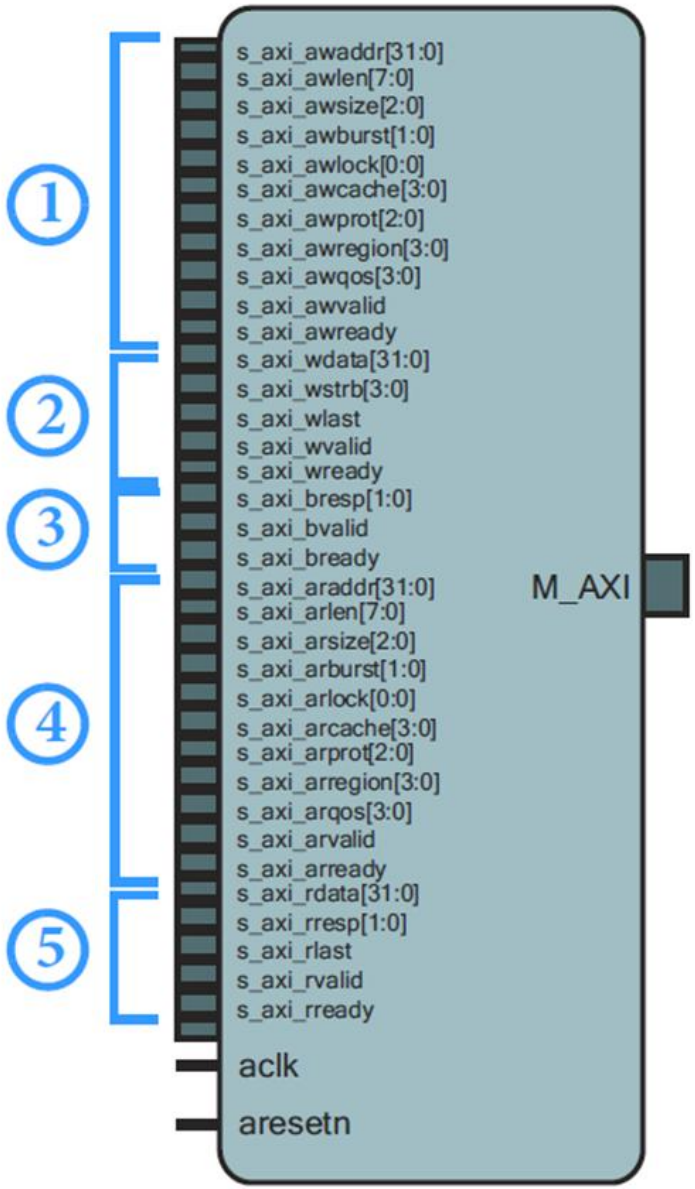
AXI Interface Example



1 The AXI Master from the PS connects to the AXI Slave of the AXI Interconnect block

2 3 4 An AXI Master from the AXI Interconnect block connects to the AXI Slave of the GPIO, BRAM blocks

AXI Slave Signals

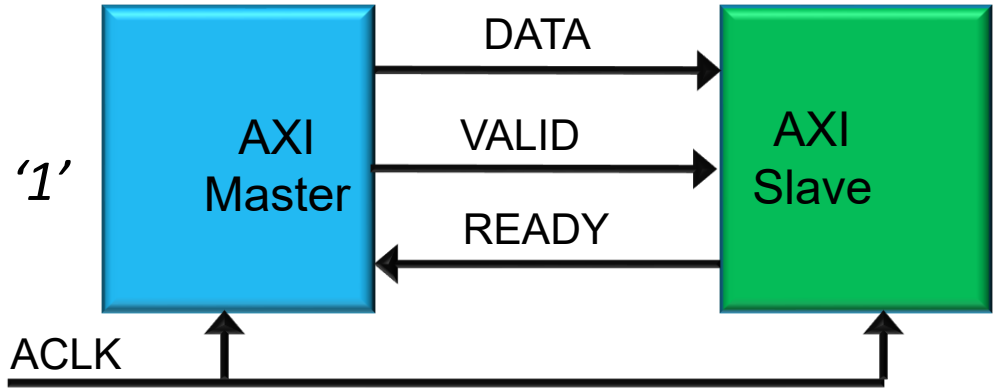


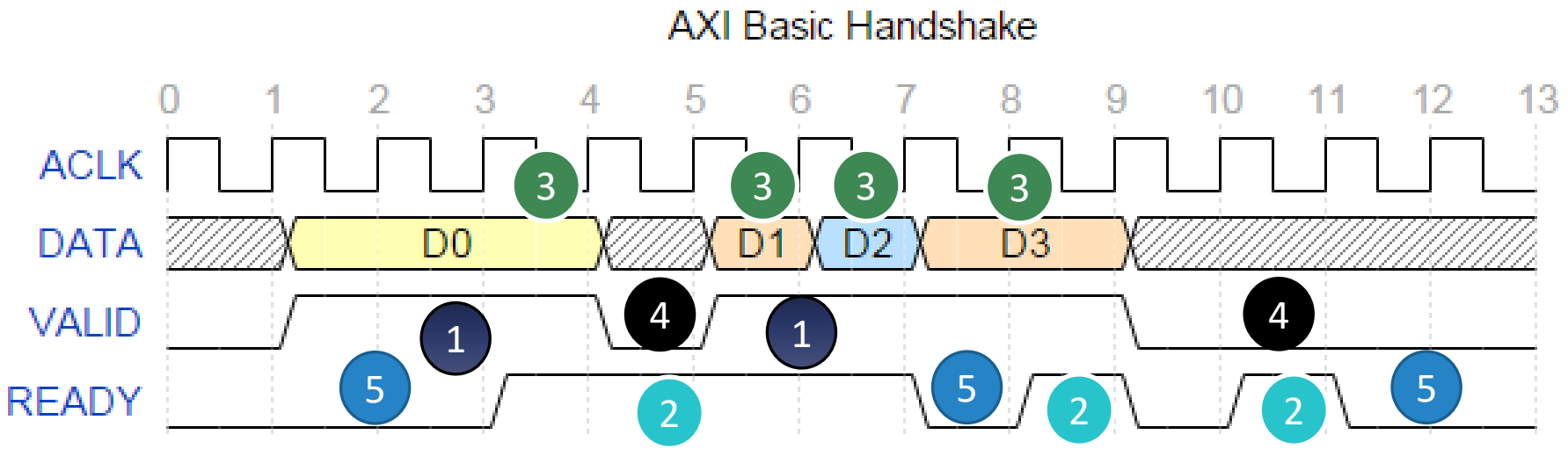
1. **Write Address Channel** — the signals contained within this channel are named in the format *s_axi_aw...*
2. **Write Data Channel** — the signals contained within this channel are named in the format *s_axi_w...*
3. **Write Response Channel** — the signals contained within this channel are named 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

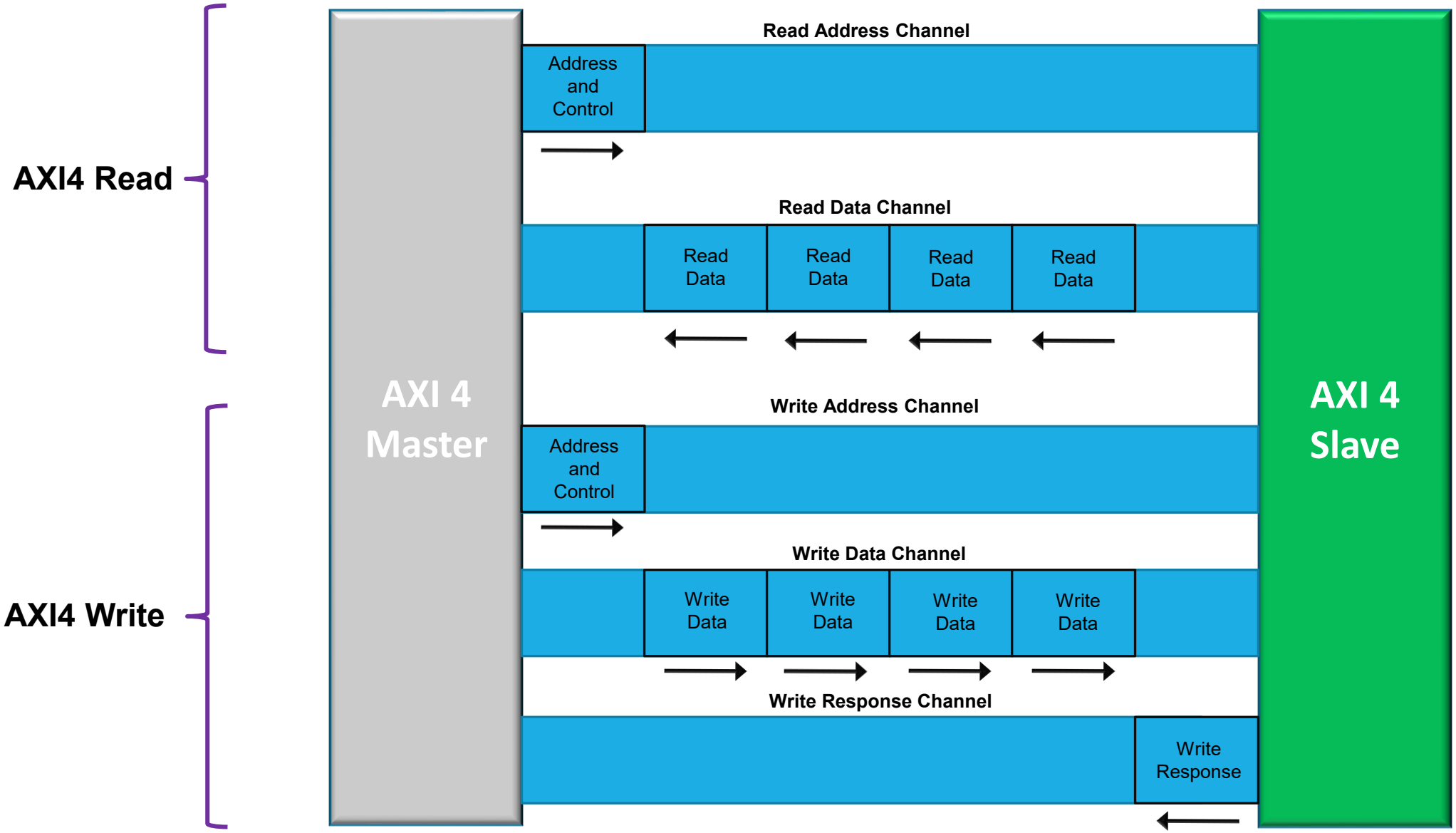
AXI Channels Use A Basic "VALID/READY" Handshake

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*



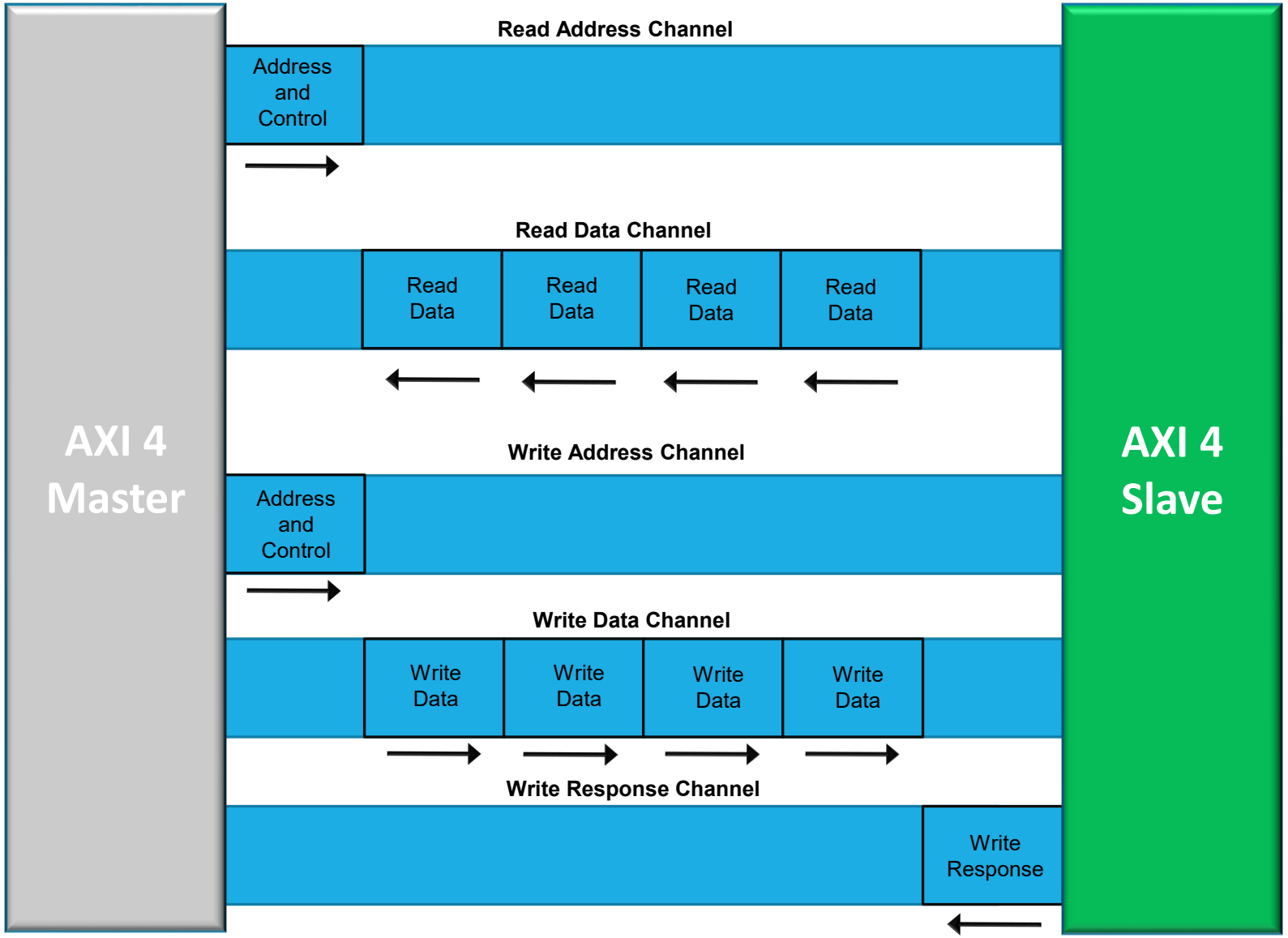


AXI Channels



AXI4 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

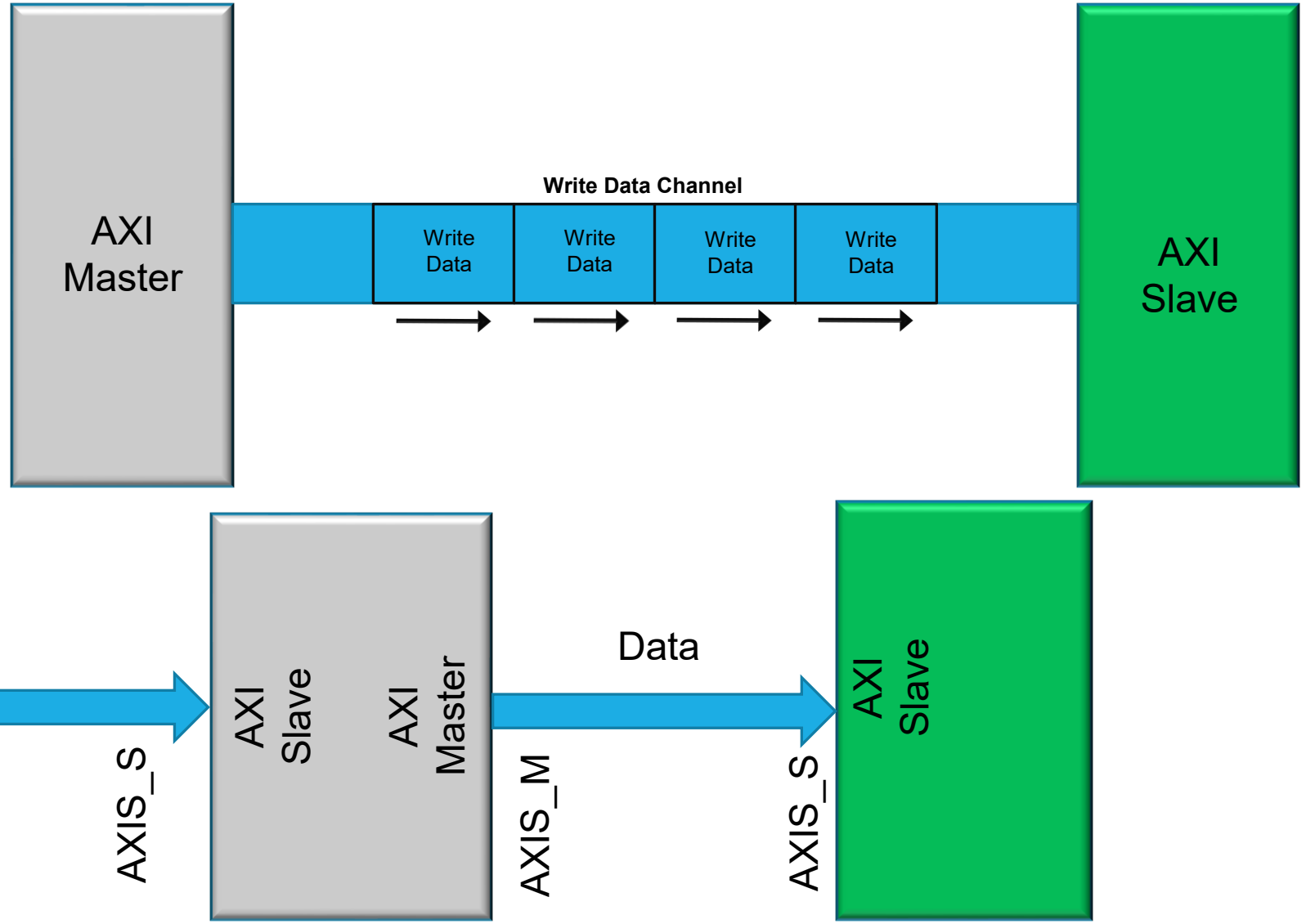


AXI4 (Full)

- 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



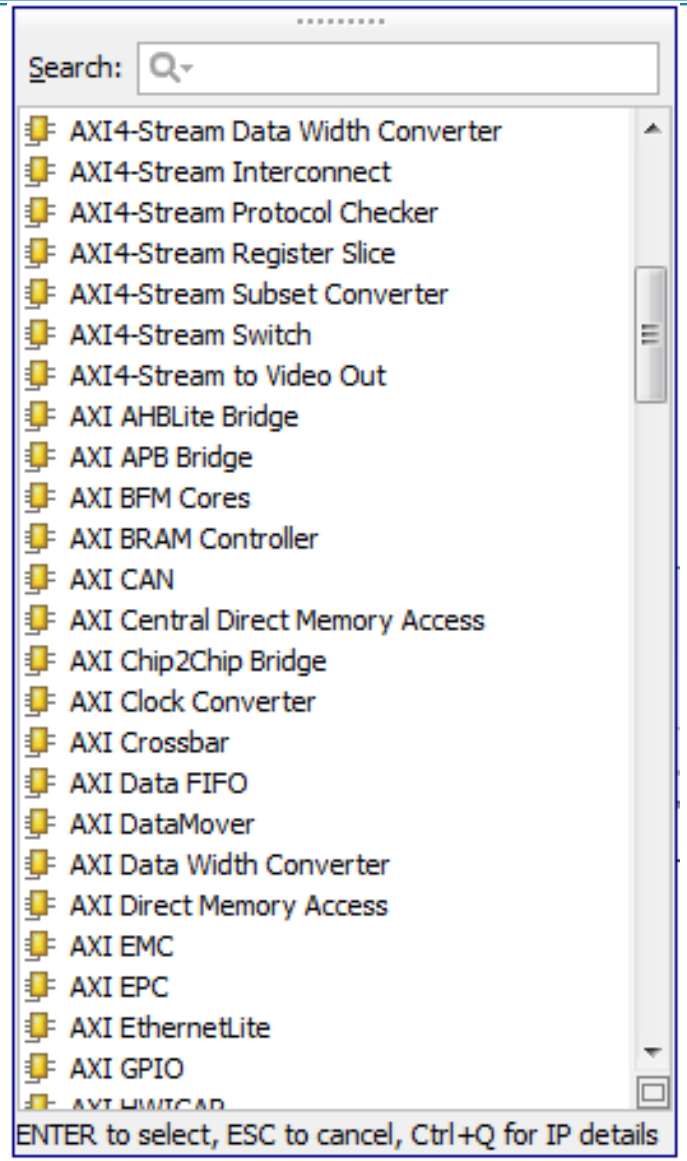
Custom AXI IP Cores

Different Soft IP Cores

Soft IP Cores		
	Pros	Cons
HDL (hardware description language)	End user can modify it	Vendor will not support if IP is modified
Encrypted HDL	Configurable using parameters	Customization is limited to the available parameters
	Sported by the vendor	
Gate-Level Netlist	High performance	Customization is limited to the available parameters
Synthesis , Place and Route are controlled by the end user		

IP Catalog Main Features

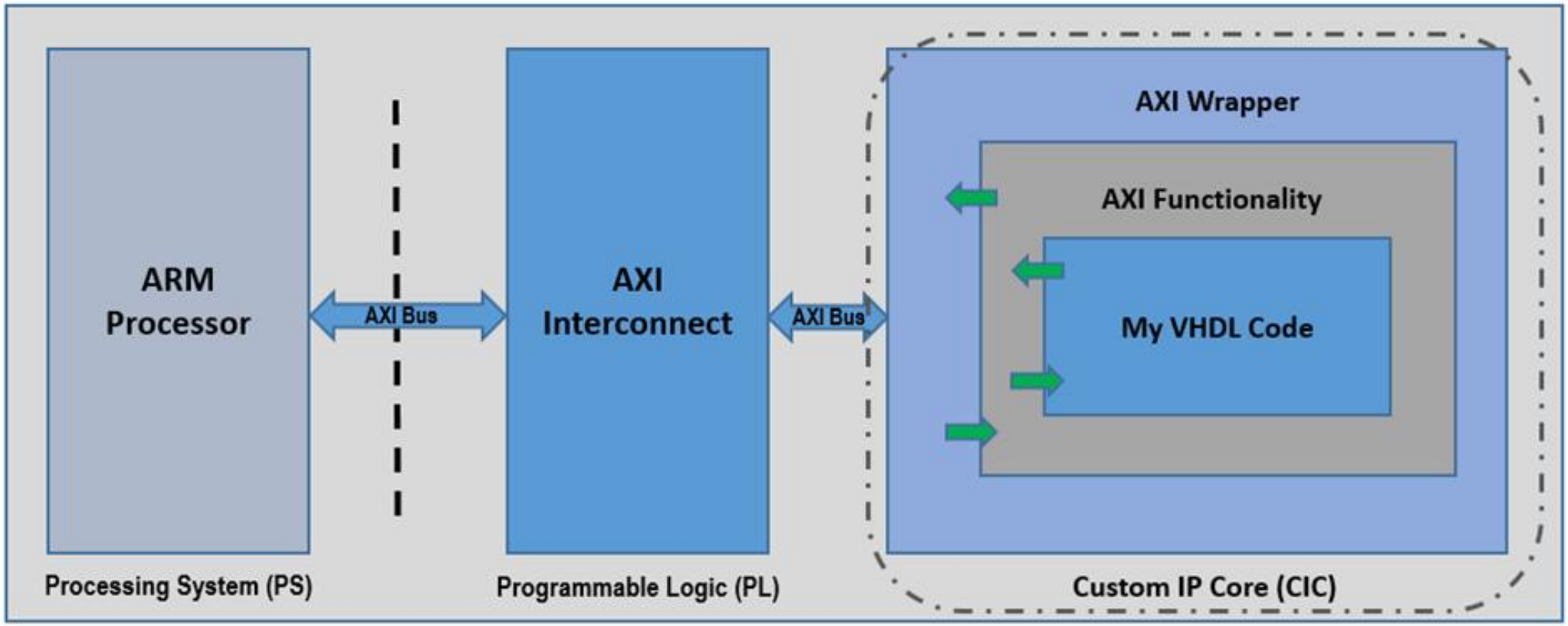
- Consistent, easy access
- Support for multiple physical locations, including shared network drives
- Access to the latest version of Xilinx-delivered IP
- Access to IP customization and generation using the Vivado IDE
- IP example designs
- Catalog filter options that let you filter by Supported Output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

My IP Generic Block Diagram



Simplest Case – Case 1

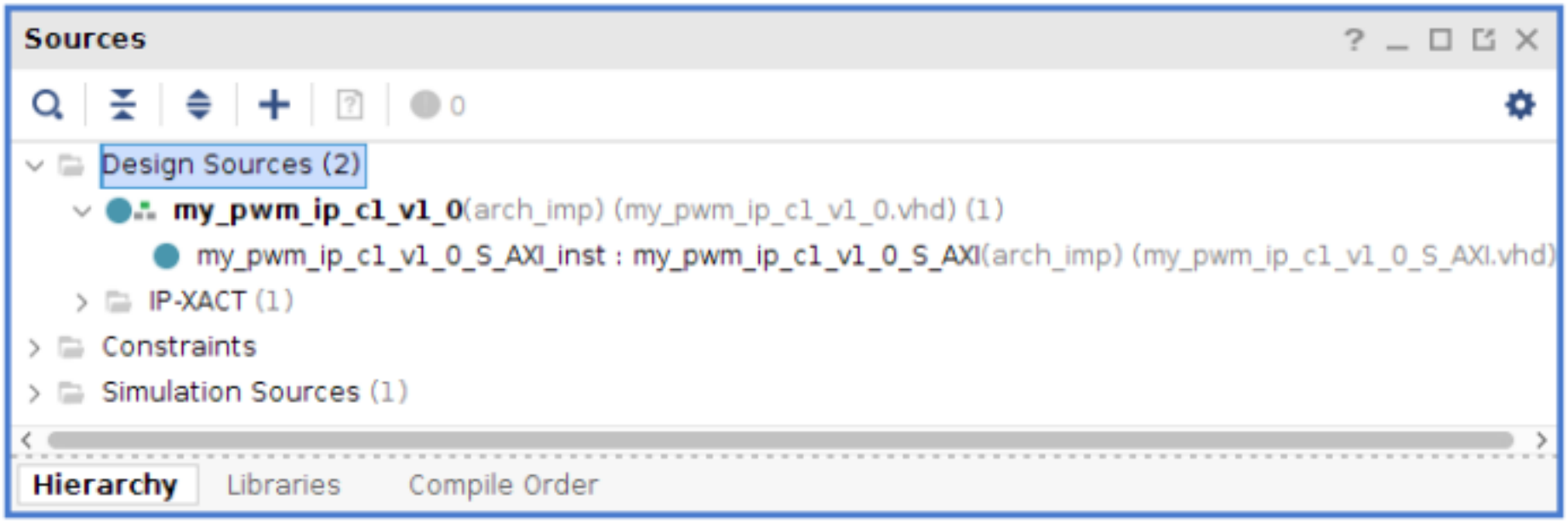
```
-------------------------------------------------------------------------------
-- entity declaration
-------------------------------------------------------------------------------
entity pwm_simple is

  generic (
    dc_bits : integer := 16);           -- number of bits for the duty cycle
  port (
    -- clock & reset signals
    S_AXI_ACLK       : in  std_logic;   -- AXI clock
    S_AXI_ARESETN    : in  std_logic;   -- AXI reset, active lov
    -- control input signal
    duty_cycle       : in  std_logic_vector(31 downto 0);
    -- PWM output
    pwm              : out std_logic     -- pwn output
    );
end entity pwm_simple;
```

1. **W**rite the functional VHDL code of the function you want to implement. In this example it is a simple PWM function

```
architecture beh of pwm_simple is
begin
  pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
     variable counter : unsigned(dc_bits-1 downto 0); -- count clocks tick
  begin  -- process pwm_pr
    if (S_AXI_ARESETN = '0') then
      counter   := (others => '0');
      pwm       <= '0';
    elsif (rising_edge(S_AXI_ACLK)) then
      counter := counter + 1;
      if (counter < unsigned(duty_cycle(dc_bits-1 downto 0))) then
        pwm <= '1';
      else
        pwm <= '0';
      end if;
    end if;
  end process pwm_pr;
end architecture beh;
```

Simplest Case – Case 1

2. **O**pen the IP Packager, and create a new IP Core (pick a good name according the functionality)



my_pwm_ip_c1_0_S_AXI_inst.vhd: This VHDL file is the one where we will instantiate the functionality described in the VHDL code.

my_pwm_ip_c1_v1_0.vhd: This VHDL file is the wrapper between our VHDL code and the AXI bus interface.

Simplest Case – Case 1

3. In the VHDL code of the '*my_pwm_ip_c1_v1_0_s_AXI.vhd*' file, where it says 'Add user logic here' add, the main ***process*** of the *pwm_simple* VHDL code.

slv_reg0 is the first writing register (address), through which the **duty_cycle** value is gotten (more details in the next slide)

```
388         -- Add user logic here
389         duty_cycle <= slv_reg0(dc_bits-1 downto 0);
390         pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
391            variable counter : unsigned(dc_bits-1 downto 0); -- count clocks tick
392         begin  -- process pwm_pr
393           if (S_AXI_ARESETN = '0') then
394             counter   := (others => '0');
395             pwm       <= '0';
396           elsif (rising_edge(S_AXI_ACLK)) then
397             counter := counter + 1;
398             if (counter < unsigned(duty_cycle(dc_bits-1 downto 0))) then
399               pwm <= '1';
400             else
401               pwm <= '0';
402             end if;
403           end if;
404         end process pwm_pr;
405         -- User logic ends
```

Simplest Case – Case 1

slv_reg0 write decoding

```
-- Implement memory mapped register select and write logic generation
-- The write data is accepted and written to memory mapped registers when
-- axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write strobes are used to
-- select byte enables of slave registers while writing.
-- These registers are cleared when reset (active low) is applied.
-- Slave register write enable is asserted when valid address and data are available
-- and the slave is ready to accept the write address and write data.
slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and S_AXI_AWVALID ;

process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
      slv_reg2 <= (others => '0');
      slv_reg3 <= (others => '0');
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      if (slv_reg_wren = '1') then
        case loc_addr is
          when b"00" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 0
                slv_reg0(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
              end if;
            end loop;
          when b"01" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 1
                slv_reg1(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
```

Simplest Case – Case 1

4. In the entity of the file, add the generic: '***dc_bits***' and the output: '***pwm***'

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity my_pwm_ip_c1_v1_0_S_AXI is
6       generic (
7           -- Users to add parameters here
8           dc_bits : integer := 16;
9           -- User parameters ends
10          -- Do not modify the parameters beyond this line
11
12          -- Width of S_AXI data bus
13          C_S_AXI_DATA_WIDTH  : integer   := 32;
14          -- Width of S_AXI address bus
15          C_S_AXI_ADDR_WIDTH  : integer   := 4
16      );
17      port (
18          -- Users to add ports here
19          -- PWM output
20          pwm         : out std_logic;    -- pwn output
21          -- User ports ends
22          -- Do not modify the ports beyond this line
```

Simplest Case – Case 1

5. In the file *my_pwm_ip_c1_v1_0.vhd*, in the entity, add the generic: '***dc_bits***' and the output: '***pwm***'

```
5     entity my_pwm_ip_c1_v1_0 is
6         generic (
7             -- Users to add parameters here
8             dc_bits : integer := 16;
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            -- PWM output
20            pwm : out std_logic;
21            -- User ports ends
22            -- Do not modify the ports beyond this line
```

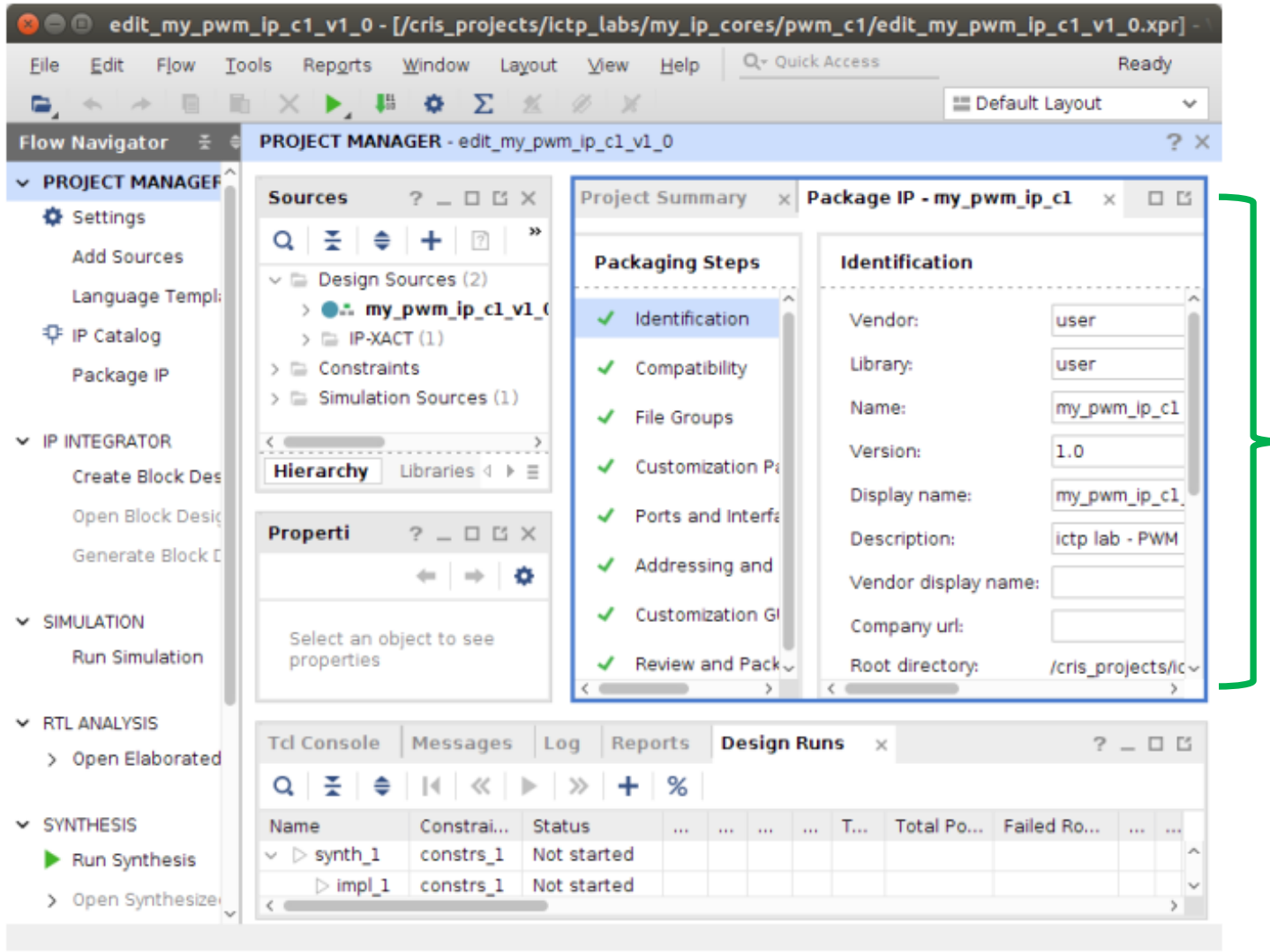
Simplest Case – Case 1

6. In the file ***my_pwm_ip_c1_v1_0.vhd***, in the architecture, in the component declaration and in the component instantiation (of the component ***my_pwm_ip_c1_v1_0_S_AXI)***, add the generic '**dc_bits**" and the **pwm** output port.

```
52      -- component declaration
53      component my_pwm_ip_c1_v1_0_S_AXI is
54          generic (
55          dc_bits             : integer    := 16;
56          C_S_AXI_DATA_WIDTH  : integer    := 32;
57          C_S_AXI_ADDR_WIDTH  : integer    := 4
58          );
59          port (
60          -- PWM output
61          pwm             : out std_logic;     -- pwm output
62          S_AXI_ACLK      : in std_logic;
```

```
88      -- Instantiation of Axi Bus Interface S_AXI
89      my_pwm_ip_c1_v1_0_S_AXI_inst : my_pwm_ip_c1_v1_0_S_AXI
90          generic map (
91              dc_bits             => dc_bits,
92              C_S_AXI_DATA_WIDTH  => C_S_AXI_DATA_WIDTH,
93              C_S_AXI_ADDR_WIDTH  => C_S_AXI_ADDR_WIDTH
94          )
95          port map (
96              pwm             => pwm,
97              S_AXI_ACLK      => s_axi_aclk,
```

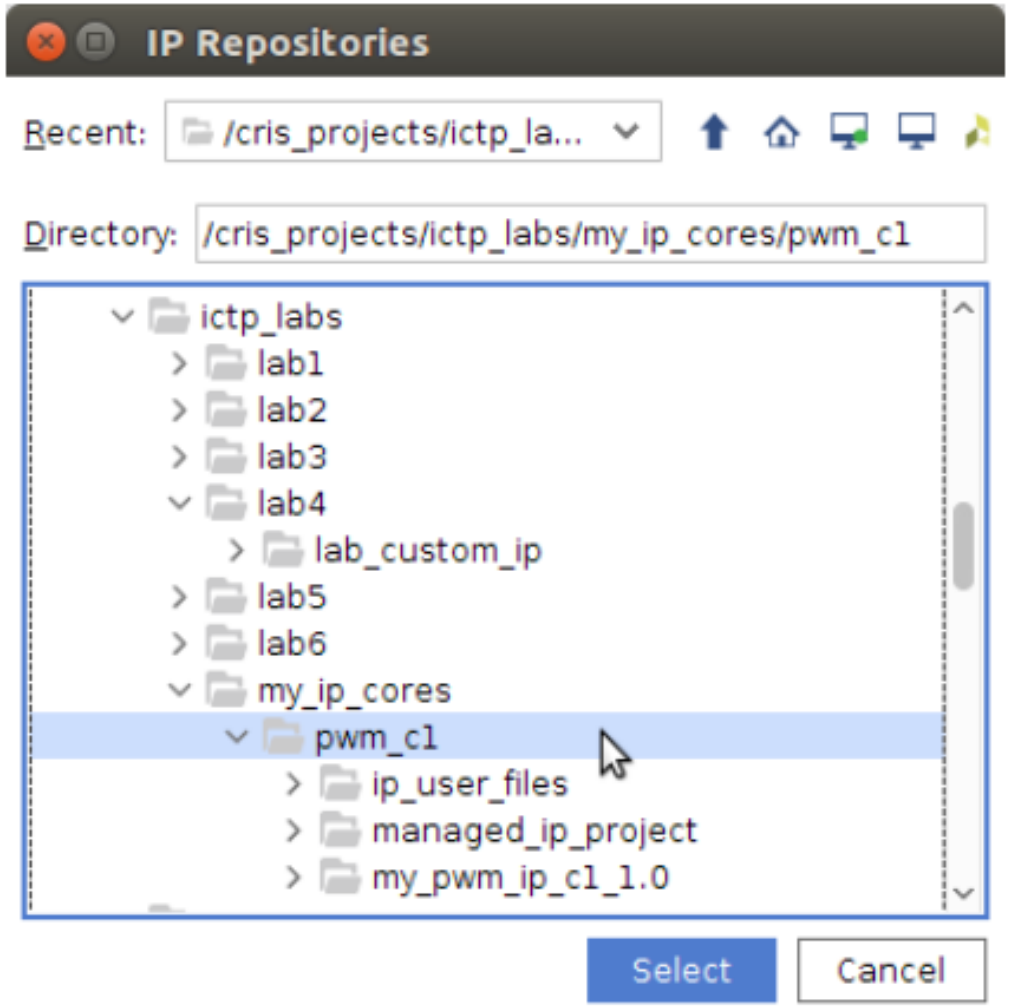
Simplest Case – Case 1



7. **T**hen, in the IP Packager, fill in the requested information in the Package IP tab.

8. **C**lose the IP Packager.

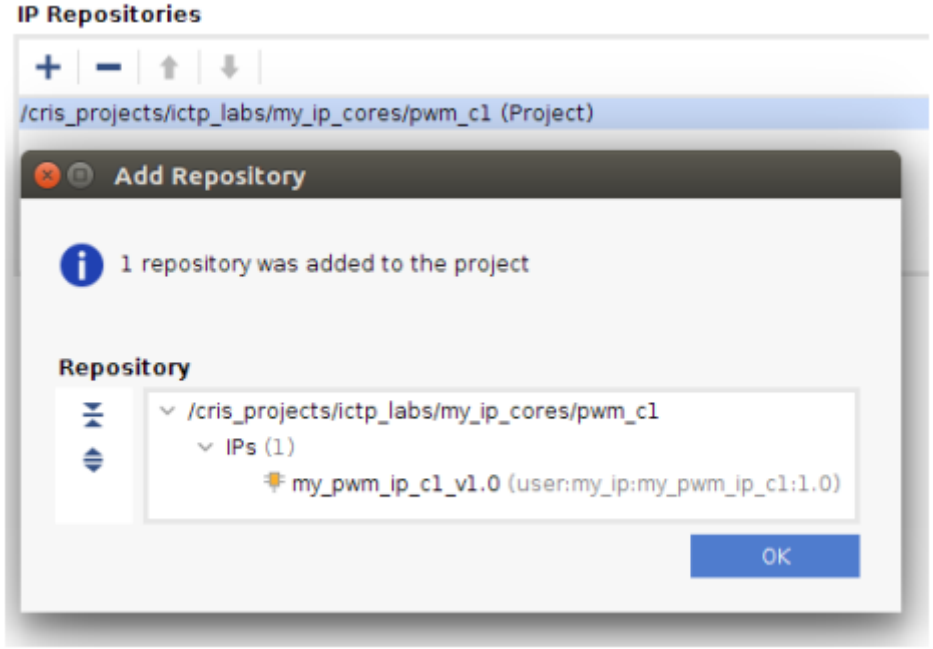
Using My IP in Vivado



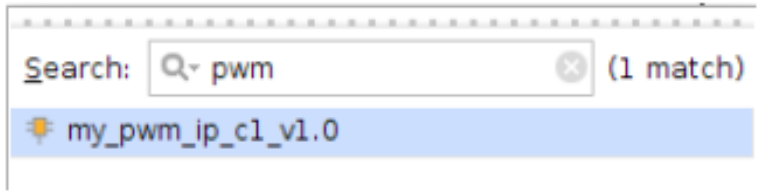
9. In the Vivado Project Manager, go to **Project Settings -> IP Repository.**

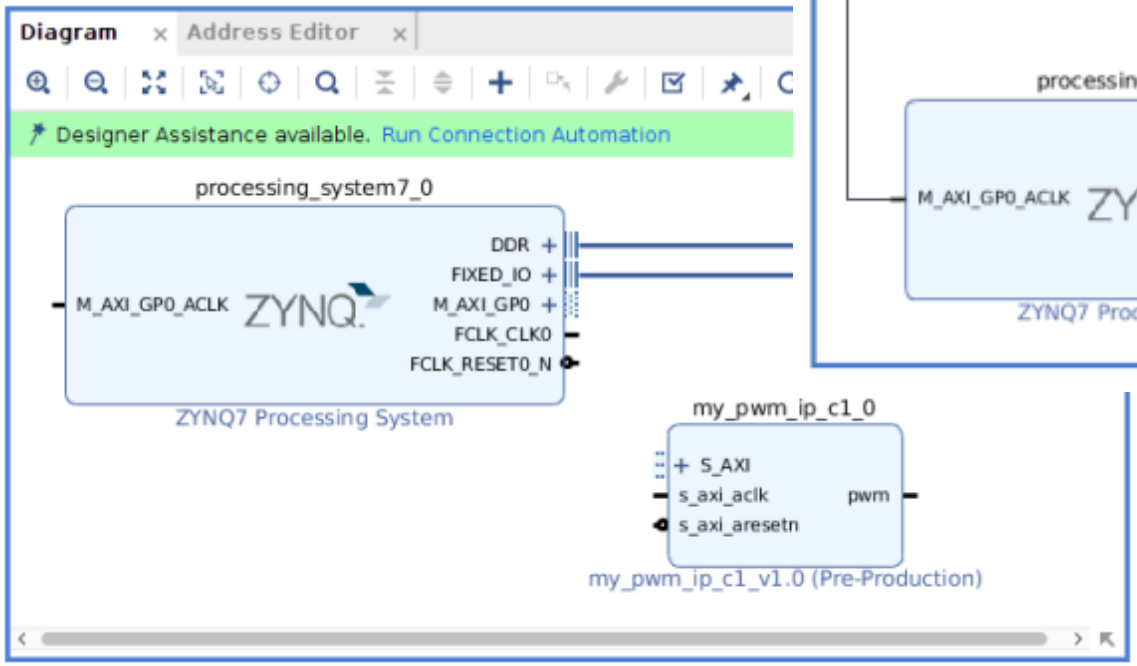
10. **C**lick on the **+** sign to add the repository (directory) where you created the IP Core.

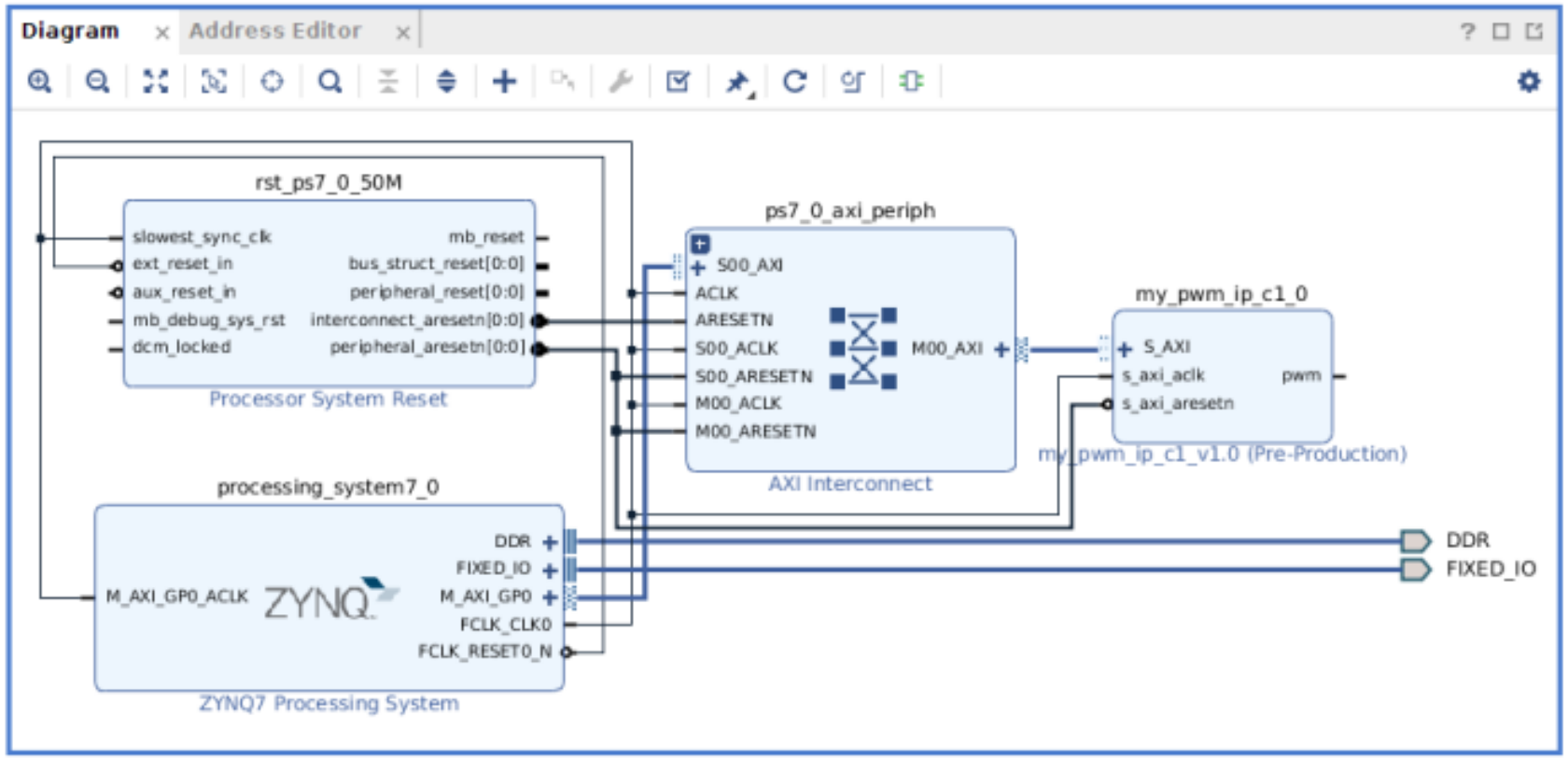
11. **C**lick Ok. Then the IP Core Will be visible in the Vivado IP Library.

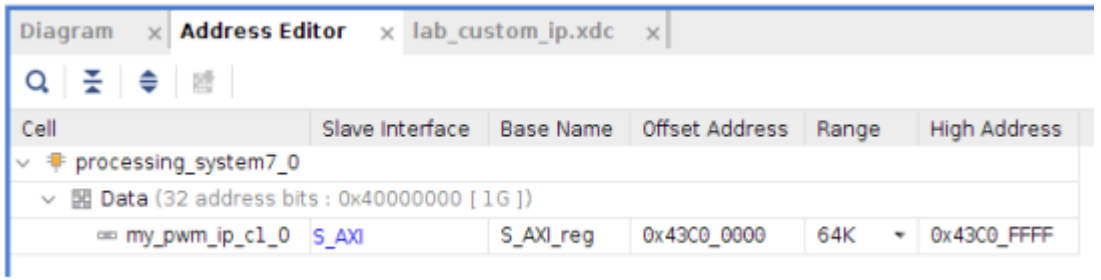


Using My IP in Vivado

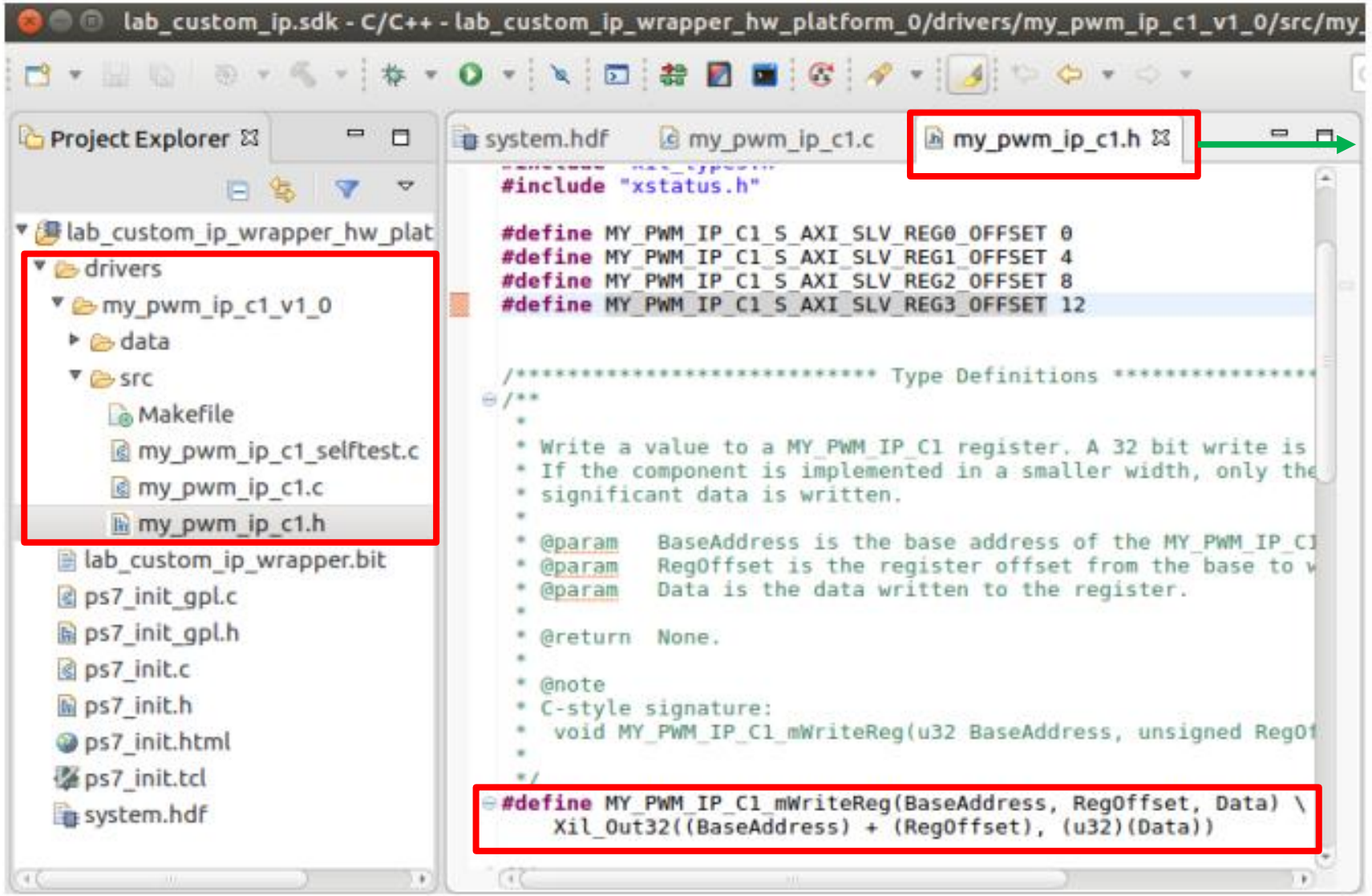








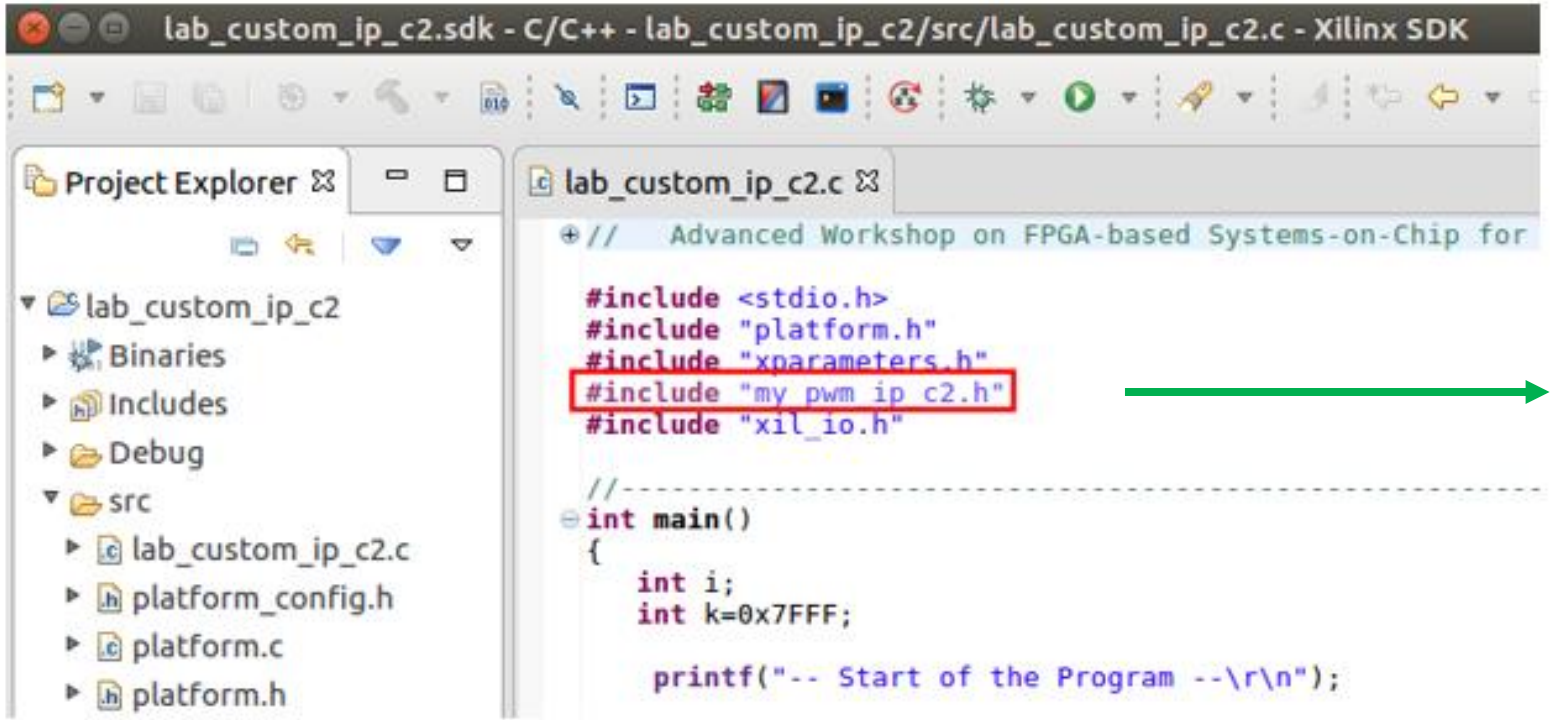
Using My IP in Vitis



.h file of the created IP Core

write function

Using My IP in Vitis



It is necessary to include the .h file in the .c

My IP – Case 2

In this case, the VHDL Code, the ‘component’, is instantiated in the created IP Core file: *my_pwm_ip_c2_0_S_AXI_inst.vhd* (this name can be any).

This case is very similar to ‘Case 1’, the only difference is that the PWM functionality is implemented by instantiating the ‘pwm_simple’ component, instead of writing the PWM process like in Case 1.

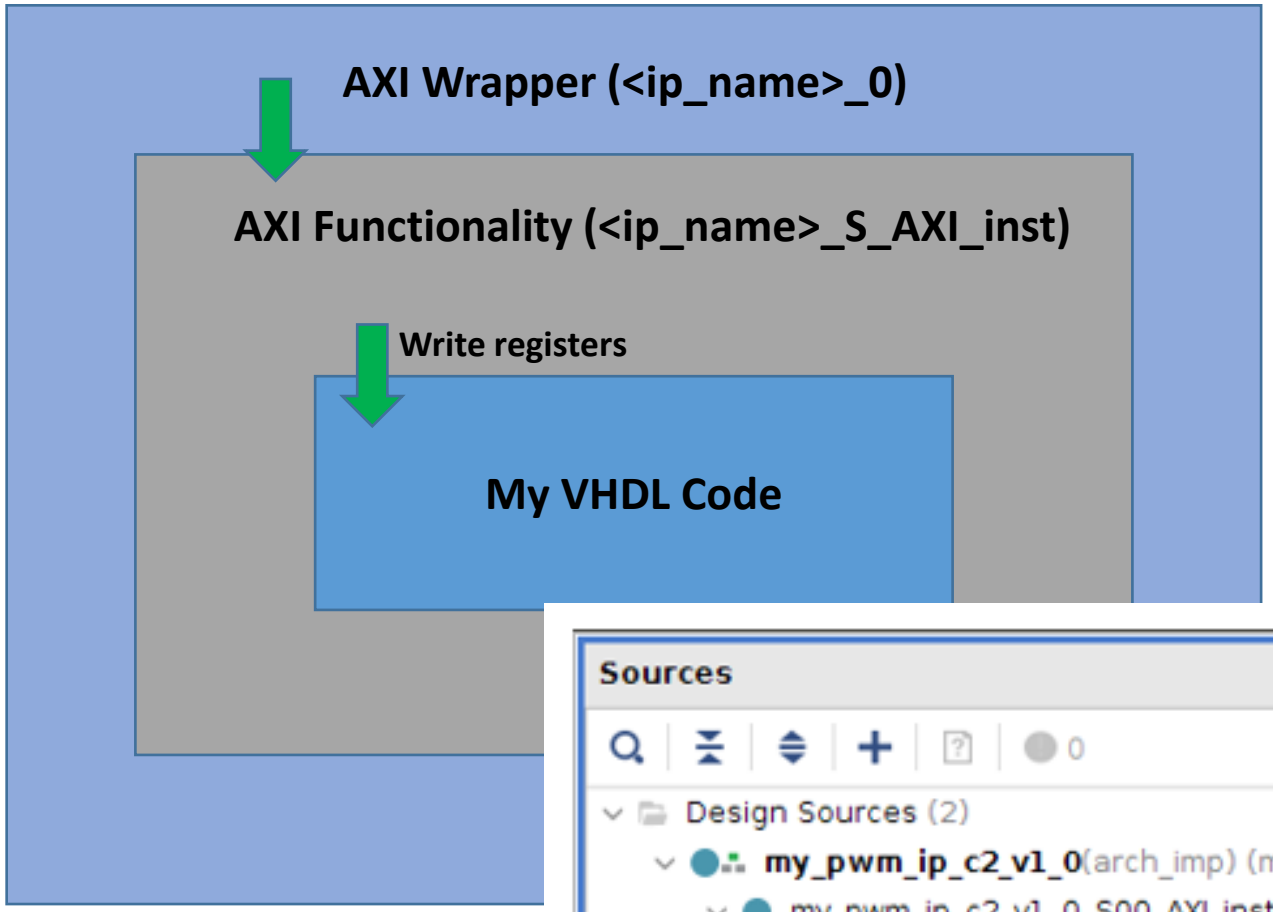
So, all the steps explained for Case 1 have to be follow, except the ***process***

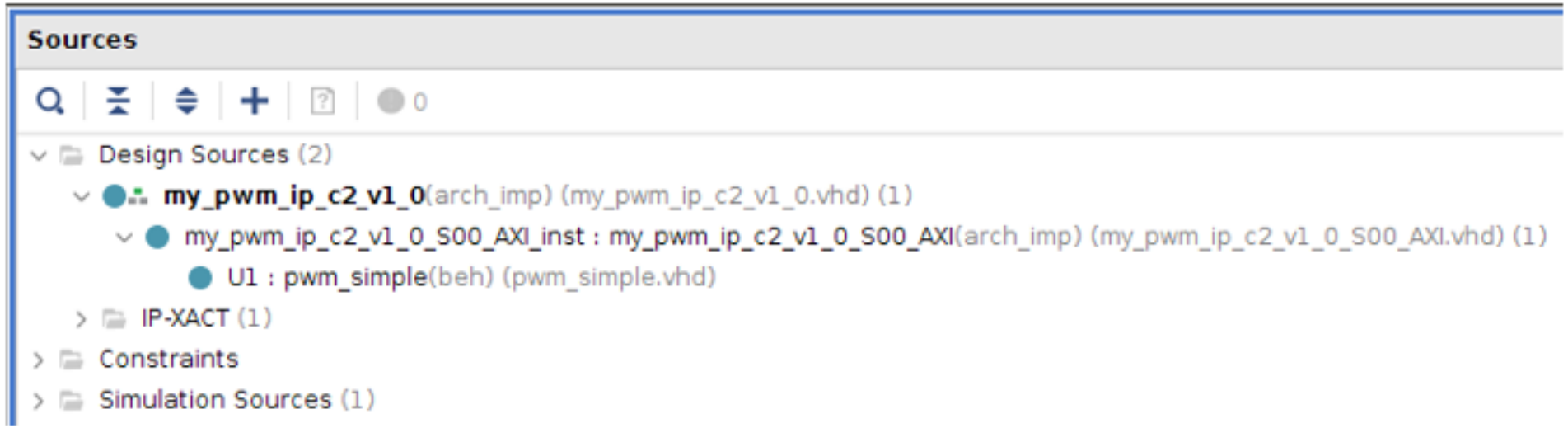
.

The ‘component’ is instantiated in the m*y_pwm_ip_c2_0_S_AXI_inst.vhd* file.

```
385         -- Add user logic here
386         U1: entity work.pwm_simple   -- pwm_simple component instantiation
387           generic map (
388             dc_bits => dc_bits)
389           port map(
390             S_AXI_ACLK     => S_AXI_ACLK,
391             S_AXI_ARESETN  => S_AXI_ARESETN,
392             duty_cycle     => slv_reg0,
393             pwm            => pwm);
394         -- User logic ends
395
396     end arch_imp;
```

My IP – Case 2


AXI Wrapper (<ip_name>_0)
AXI Functionality (<ip_name>_S_AXI_inst)
Write registers
My VHDL Code


Sources
0
Design Sources (2)
my_pwm_ip_c2_v1_0(arch_imp) (my_pwm_ip_c2_v1_0.vhd) (1)
my_pwm_ip_c2_v1_0_S00_AXI_inst : my_pwm_ip_c2_v1_0_S00_AXI(arch_imp) (my_pwm_ip_c2_v1_0_S00_AXI.vhd) (1)
U1 : pwm_simple(beh) (pwm_simple.vhd)
IP-XACT (1)
Constraints
Simulation Sources (1)

My IP – Case 2

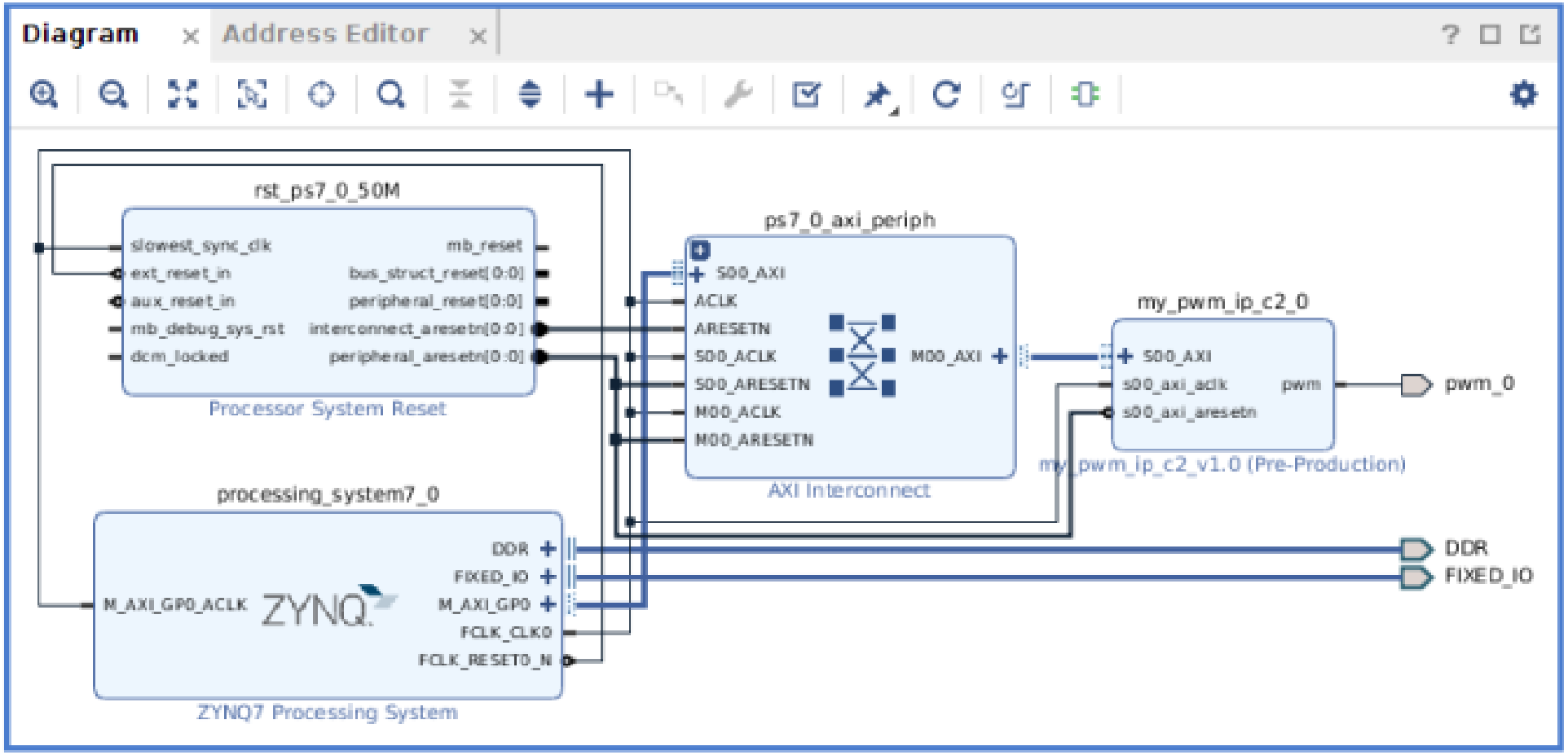
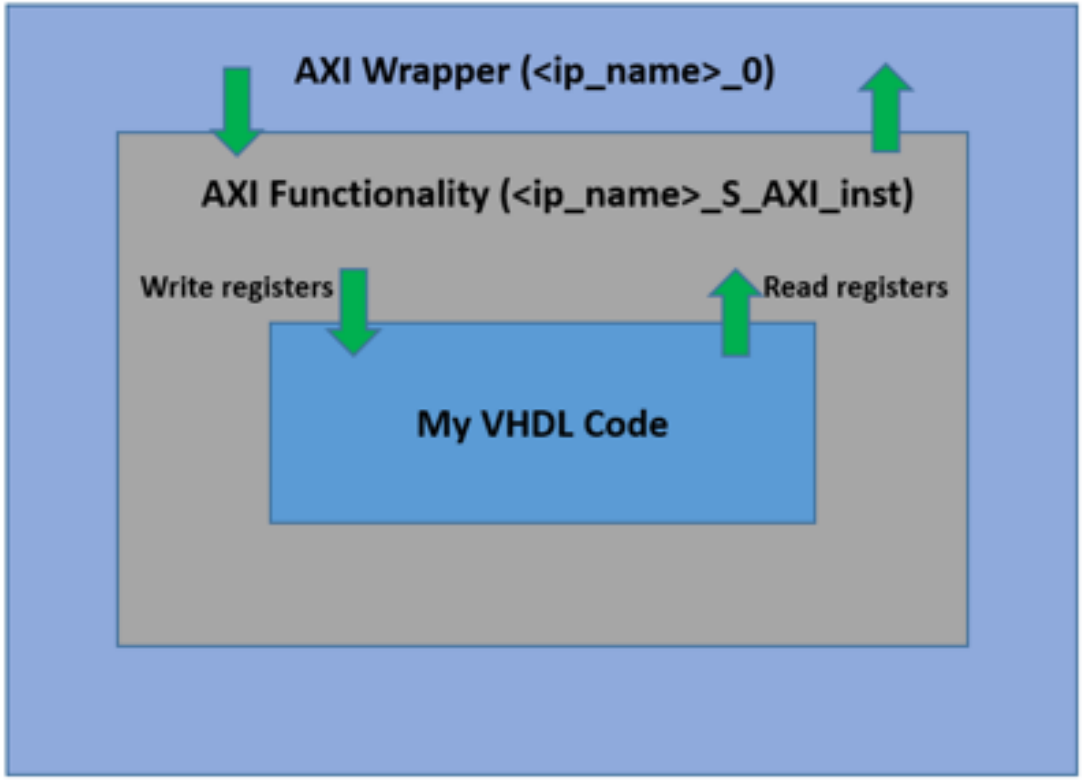


Diagram
Address Editor
rst_ps7_0_50M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
ps7_0_axi_periph
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M00_AXI
AXI Interconnect
my_pwm_ip_c2_0
S00_AXI
s00_axi_aclk
s00_axi_aresetn
pwm
my_pwm_ip_c2_v1.0 (Pre-Production)
pwm_0
processing_system7_0
M_AXI_GP0_ACLK
ZYNQ
DDR
FIXED_IO
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
DDR
FIXED_IO

My IP – Case 3

The PWM VHDL has more input and output signals that we would like to be controlled by the PS.
The PWM IP Core's registers, defined by the HW designer, now has: .
2 registers to write to, 3 registers to read from.

slv_reg0	reg0_control	in
slv_reg1	reg1_status	out
slv_reg2	reg2_pwm_dc_value	in
slv_reg3	reg3_ip_version	out
slv_reg4	reg4_pwm_dc_value	out



My IP – Case 3 - VHDL

```
12  ----------------------------------------------------------------------------------
13  -- Description: generation of the PWM signal using Rd/Wr registers that
14  -- will be Wr/Rd through the AXI bus.
15  --
16  -- The following register are defined for this PWM IP:
17  --
18  -- ----------------       Register 0: Control Register   ----------------------
19  -- bit31 | ... |   bit 4   |   bit 3   |   bit2   |   bit 1   |   bit 0   |
20  --                clear        Enable    invert PWM   enable    sw reset_n
21  --               interrupt    interrupt   output      disable
22  --
23  -- ----------------       Register 1: Status Register    ----------------------
24  -- bit31 | ... | ... ... ... ... ... ... ... ...   |  bit 1   |   bit 0   |
25  --                                                    interrupt   PWM output
26  --                                                      request    value
27  --
28  -- ----------------------       Register 2     -------------------------------
29  -- Writable register: ARM will write into this register the PWM (duty cycle) value
30  --
31  -- ----------------------       Register 3     -------------------------------
32  -- Readable register: hold the current version of the PWM IP module
33  --
34  -- ----------------------       Register 4     -------------------------------
35  -- Readable register: copy of Register 2, that can be read by the ARM
36  --
37  --  ----------------------       Outputs     -------------------------------
38  -- pwm: which is the PWM value, '0' or '1'
39  -- int_pwm: which generate an int request (goes to '1') on the falling edge
40  -- of the pwm ouptut.
41  ----------------------------------------------------------------------------------
42  --
43  ----------------------------------------------------------------------------------
44  -- Revisions  :
45  -- Date        Version  Author  Description
46  -- 2018-06-12  1.0      cristian  Created
47  ----------------------------------------------------------------------------------
```

My IP – Case 3 – VHDL Code - Entity

```
------------------------------------------------------------------------------
-- entity declaration
------------------------------------------------------------------------------
entity pwm_complete is

  generic (
    dc_bits : integer := 32);           -- number of bits for the duty cycle

  port (
    -- clock & reset signals
    S_AXI_ACLK          : in  std_logic;  -- AXI clock
    S_AXI_ARESETN       : in  std_logic;  -- AXI async reset, active low

    -- registers
    reg0_control        : in  std_logic_vector(31 downto 0);
    reg1_status         : out std_logic_vector(31 downto 0);
    reg2_pwm_dc_value   : in  std_logic_vector(31 downto 0);
    reg3_ip_version     : out std_logic_vector(31 downto 0);
    reg4_pwm_dc_value   : out std_logic_vector(31 downto 0);

    -- PWM output
    pwm                 : out std_logic;          -- pwn output;

    -- Int request output
    pwm_int_req         : out std_logic
    );
end entity pwm_complete;
```

My IP – Case 3 – VHDL Code - Architecture

```
------------------------------------------------------------------------------
-- architecture
------------------------------------------------------------------------------
architecture beh of pwm_complete is

  -- PWM IP version constant declaration
  constant pwm_version_ctt : std_logic_vector(31 downto 0) := X"00010001";  -- V 1.1

  -- alias declaration for the different bits of the control register
  alias soft_reset_bit_n: std_logic is reg0_control(0);  -- sw reset initialized by
                                                         -- PS7, active low
  alias enable_bit      : std_logic is reg0_control(1);  -- enable the whole PWM module
  alias pwm_invert_bit  : std_logic is reg0_control(2);  -- invert the PWM output when '1'
  alias enable_int_bit  : std_logic is reg0_control(3);  -- enable int when '1'
  alias clear_int_bit   : std_logic is reg0_control(4);  -- clear int request
  alias duty_cycle_reg  : std_logic_vector(31 downto 0) is reg2_pwm_dc_value(31 downto 0);  -- initial duty cycle value

  -- internal signal declarations
  signal reset_n        : std_logic;                     -- global reset (hw and sw)
  signal pwm_i          : std_logic;                     -- internal pwm generation
  signal pwm_dly        : std_logic;                     -- one clock delayed version of pwm_i
  signal pwm_out_i      : std_logic;                     -- internal pwm ouptut
  signal int_req_bit_i  : std_logic;                     -- internal int request signal
```

My IP – Case 3 - Arch

```
-- assign version number to version register
reg3_ip_version <= pwm_version_ctt;

-- update status reg to be read by the ARM
reg1_status  <= ((1) => int_req_bit_i,  -- int request bit
                 (0) => pwm_out_i,      -- current pwm output value
                 others => '0');

-- assign current duty cycle to read register
reg4_pwm_dc_value <= duty_cycle_reg;      -- current value of duty cycle to be read

-- reset = hw_reset or sf_reset
reset_n <=  S_AXI_ARESETN and soft_reset_bit_n;

---&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&---
--------------------------------------------------------------------------------
-- duty cycle process
-- count clock cycles until reach the value in 'alias' duty_cycle (reg2)
--------------------------------------------------------------------------------
pwm_pr : process (S_AXI_ACLK, reset_n) is
 variable counter : unsigned(dc_bits-1 downto 0);      -- count clocks tick
begin  --
  if (reset_n = '0') then
    counter        := (others => '0');
    pwm_i          <= '0';
   -- duty_cycle_reg <= 0X"0000FF00";
  elsif (rising_edge(S_AXI_ACLK)) then
    if (enable_bit = '1')  then
      counter := counter + 1;
      if (counter < unsigned(duty_cycle_reg)) then
        pwm_i <= '1';
      else
        pwm_i <= '0';
      end if;
    end if;
  end if;
end process pwm_pr;
```

My IP – Case 3 – VHDL Code - Architecture

```
-- invert PWM output when required
pwm_out_i     <= not pwm_i when (pwm_invert_bit = '1') else pwm_i;
pwm           <= pwm_out_i;              -- entity output
-- pwm_value_bit <= pwm_out_i;            -- status register bit 0


---&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
------------------------------------------------------------------------------
-- the following two process are related to interrupt generation
-- negative edge detection for pwm_i to generate an interrupt request
-- the interrupt request is cleared by the software by writing '1' to the
-- int clear bit in the control register
------------------------------------------------------------------------------
int_pwm_dly_pr: process (S_AXI_ACLK, reset_n)is
begin
  if reset_n='0' then
    pwm_dly <= '0';
  elsif rising_edge(S_AXI_ACLK) then
    if (enable_int_bit='1') then
      pwm_dly <= pwm_i;
    end if;
  end if;
end process int_pwm_dly_pr;
```

```
------------------------------------------------------------------------------
-- int_request_bit goes to '1' until clear_int_bit is '1'
-- negative edge detection for pwm_i to generate an interrupt request
-- the interrupt request is cleared by the software by writing '1' to the
-- int clear bit in the control register
------------------------------------------------------------------------------
int_req_pr: process (S_AXI_ACLK, reset_n) is
begin
  if (reset_n = '0') then
    int_req_bit_i <= '0';
  elsif (rising_edge(S_AXI_ACLK)) then
    if (clear_int_bit='1') then
      int_req_bit_i <= '0';
    elsif ((pwm_i='0') and (pwm_dly='1')) then  -- neg edge detection
      int_req_bit_i <= '1';
    end if;
  end if;
end process int_req_pr;
------------------------------------------------------------------------------

pwm_int_req <= int_req_bit_i;          -- output from this module
--int_req_bit <= int_req_bit_i;          -- status register bit 1

------------------------------------------------------------------------------

end architecture beh;
```

My IP – Case 3 – Instantiation & VHDL Code

```
-- Add user logic here
-- slv_reg0 associated with pwm control register 0
reg0_control_i <= slv_reg0;
-- slv reg2 (32 bits) associated with duty cycle register 2 (16 bits)
reg2_duty_cycle_i <= slv_reg2;

-- pwm_complete component instantiation
--U1: entity work.pwm_complete
U1: pwm_complete
    generic map (
        dc_bits => dc_bits )
    port map (
        S_AXI_ACLK        => S_AXI_ACLK,
        S_AXI_ARESETN     => S_AXI_ARESETN,
        reg0_control      => reg0_control_i,
        reg1_status       => reg1_status_i,
        reg2_pwm_dc_value => reg2_duty_cycle_i,
        reg3_ip_version   => reg3_ip_version_i,
        reg4_pwm_dc_value => reg4_dc_value_i,
        pwm               => pwm,
        pwm_int_req       => pwm_int_req
        );
```

slv_reg0 and **slv_reg2** are writable registers, so, it is similar to the previous cases.

slv_reg1, slv_reg3 and **slv_reg4** are readable, so, we will assign the values from registers reg1_status_i, reg3_ip_versión_i, and reg4_dc_value_i, to them (details in the next slide)

My IP – Case 3 – Instantiation & VHDL Code

```
-- Implement memory mapped register select and read logic generation
-- Slave register read enable is asserted when valid address is available
-- and the slave is ready to accept the read address.
slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

process (slv_reg0, reg1_status_i, slv_reg2, reg3_ip_version_i, reg4_dc_value_i, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
      when b"000" =>
        reg_data_out <= slv_reg0;
      when b"001" =>
        reg_data_out <= reg1_status_i;      ---slv_reg1
      when b"010" =>
        reg_data_out <= slv_reg2;
      when b"011" =>
        reg_data_out <= reg3_ip_version_i;  --slv_reg3
      when b"100" =>
        reg_data_out <= reg4_dc_value_i;    --slv_reg4
      when others =>
        reg_data_out  <= (others => '0');
    end case;
end process;
```

slv_reg1, slv_reg3 and **slv_reg4** are readable, so, we Will assign the values from registers reg1_status_i, reg3_ip_versión_i, and reg4_dc_value_i, to them.

My IP – Case 3

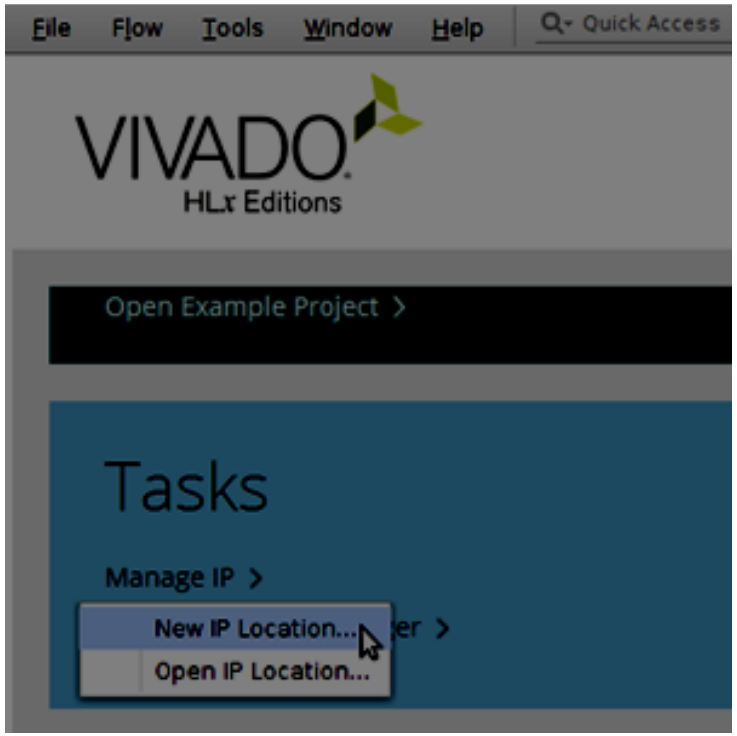
The following steps are similar to the Case 1.

- ✓ Complete the information requested by IP Packager
- ✓ Close IP Packager
- ✓ In the Vivado project, go to Poject Manager -> Settings -> IP Repository, and there add the directory where the IP Core lays.
- ✓ The core should be available in the Vivado Cores Library
- ✓ When the Vivado project is exported to Vitis, the .h header file is created, and it contains the Read and Write functions to be used to have access to the read and write registers of the IP Core.

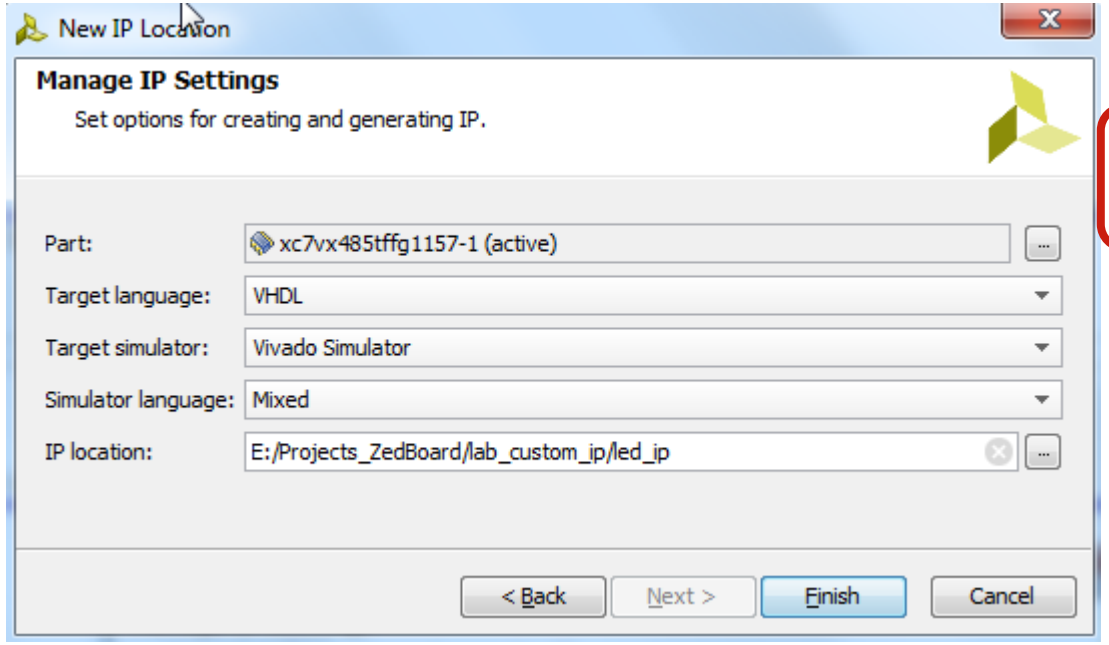
Apendix

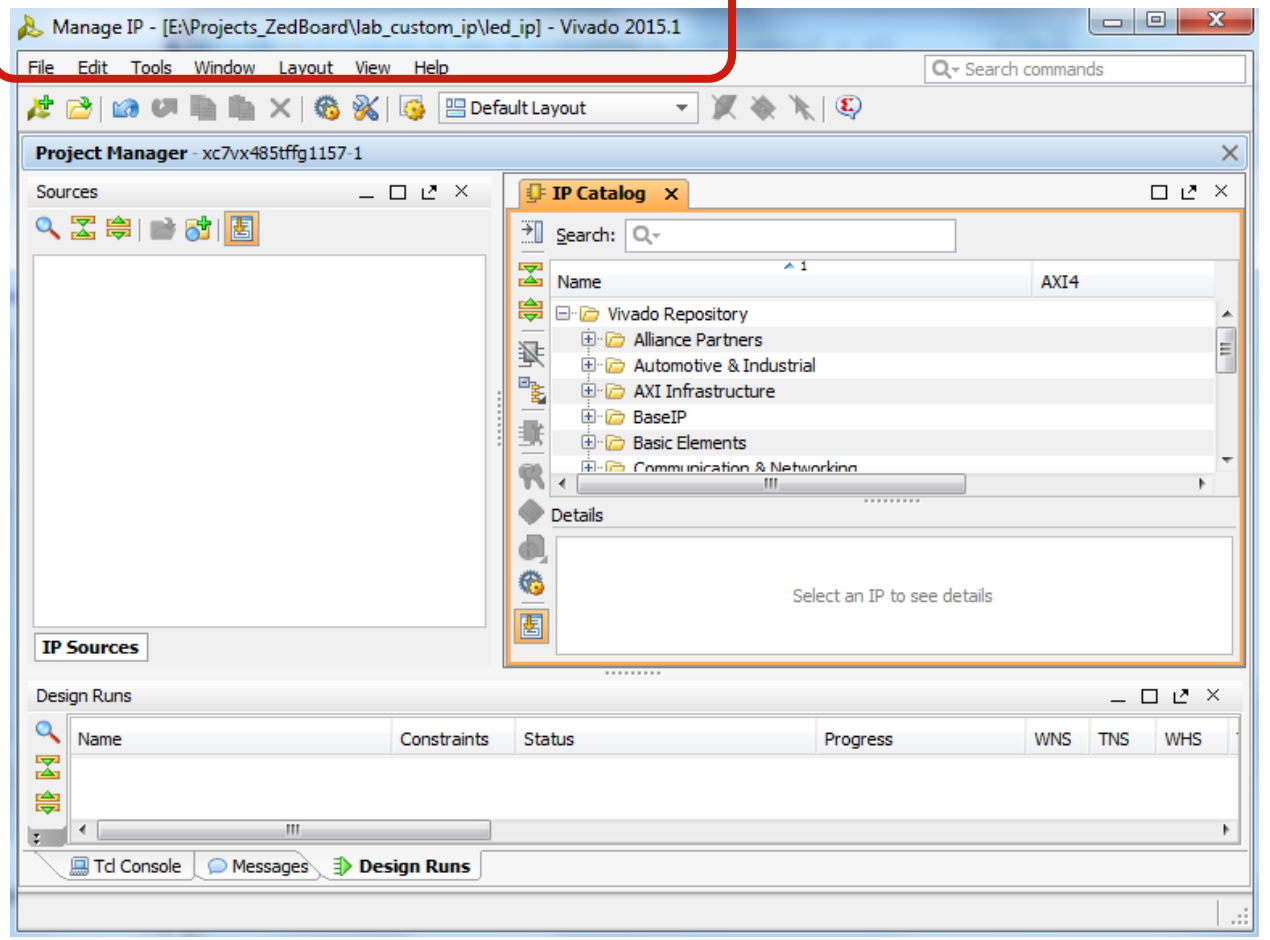
IP Manager

- **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

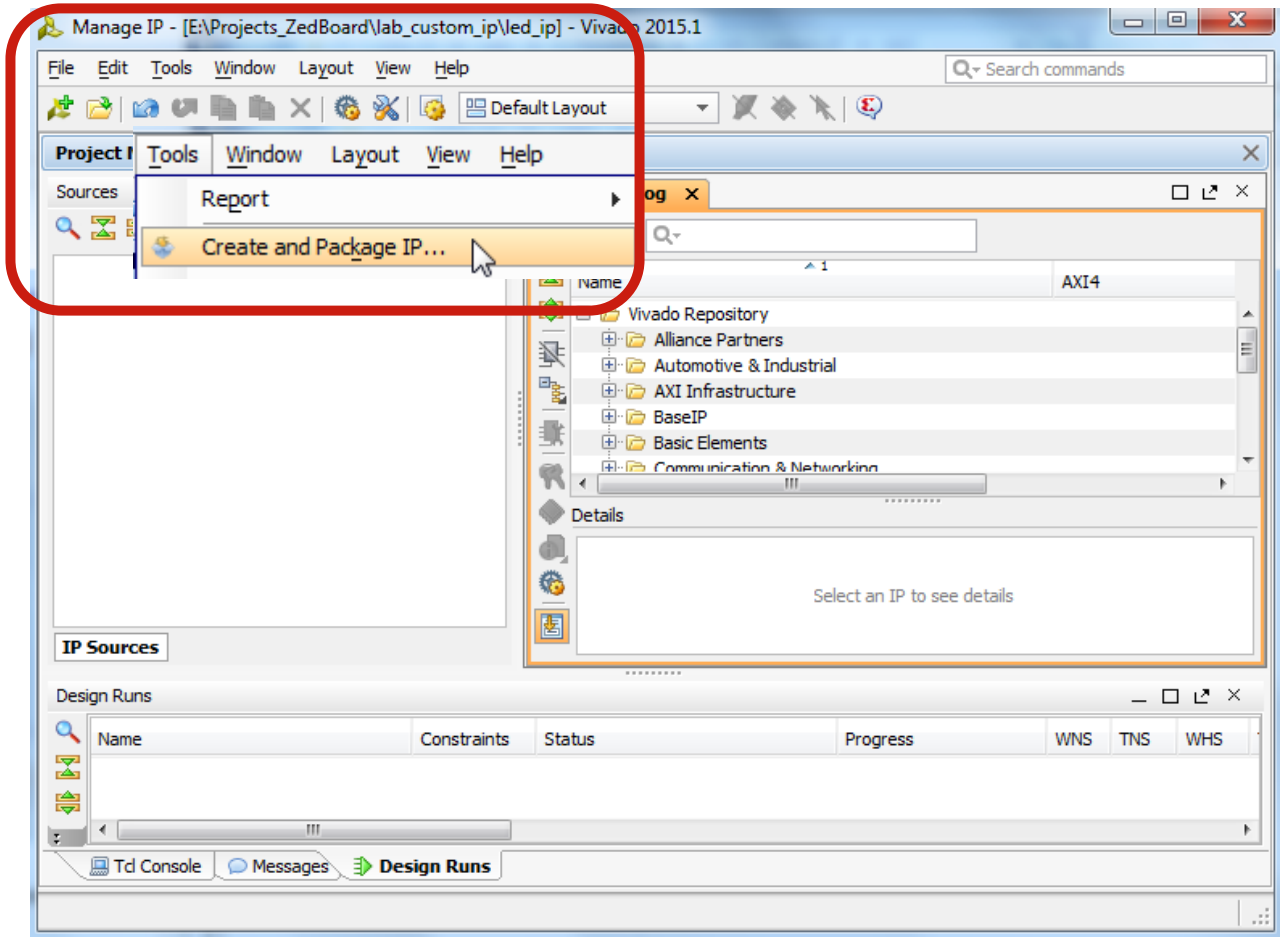


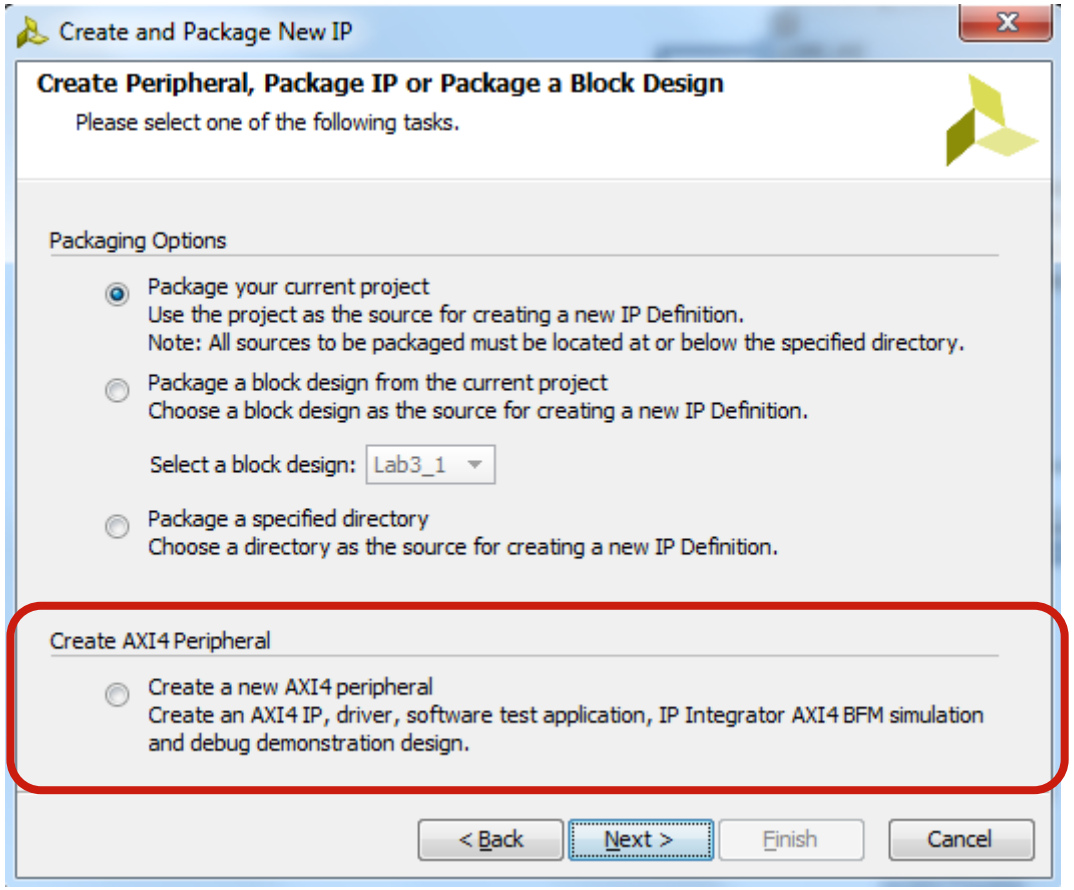
Create Custom AXI4 IP



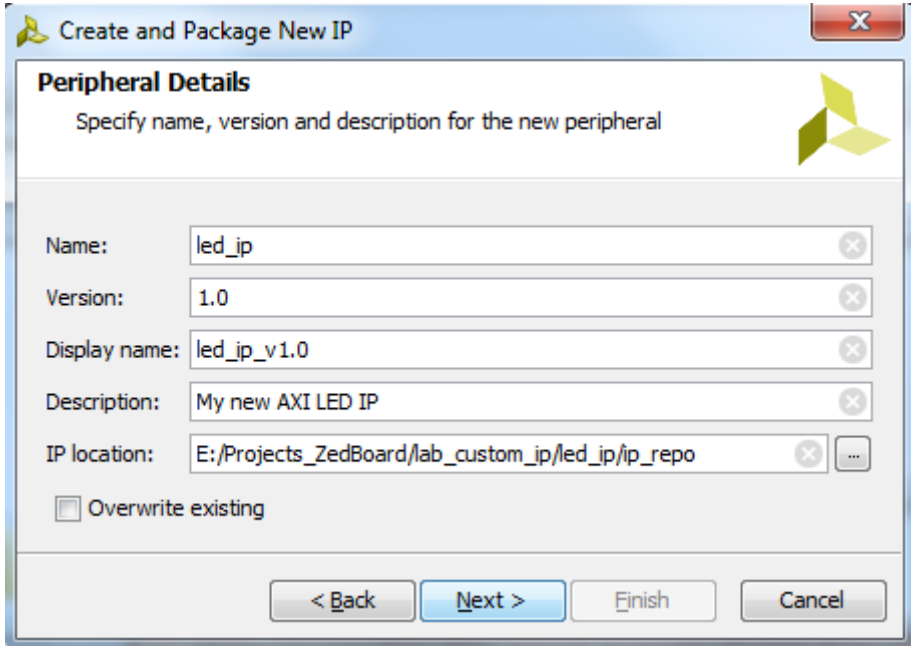


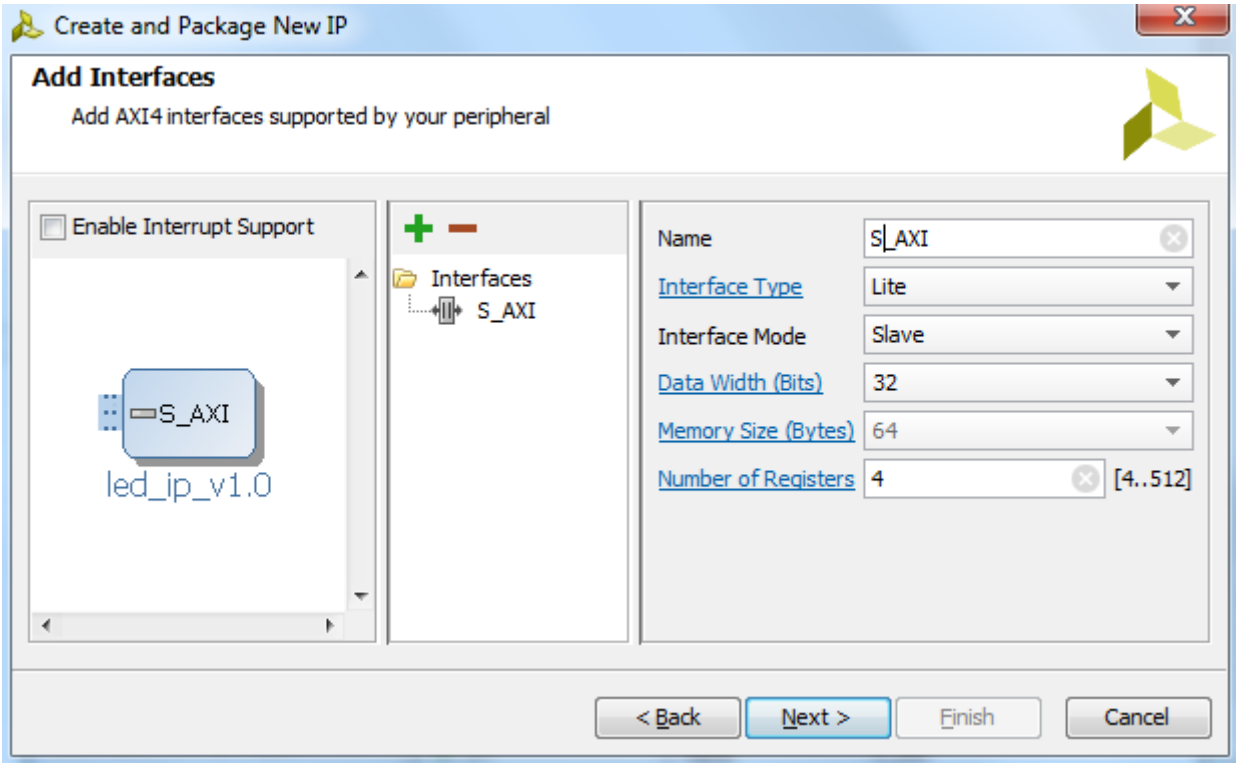
Create Custom AXI4 IP



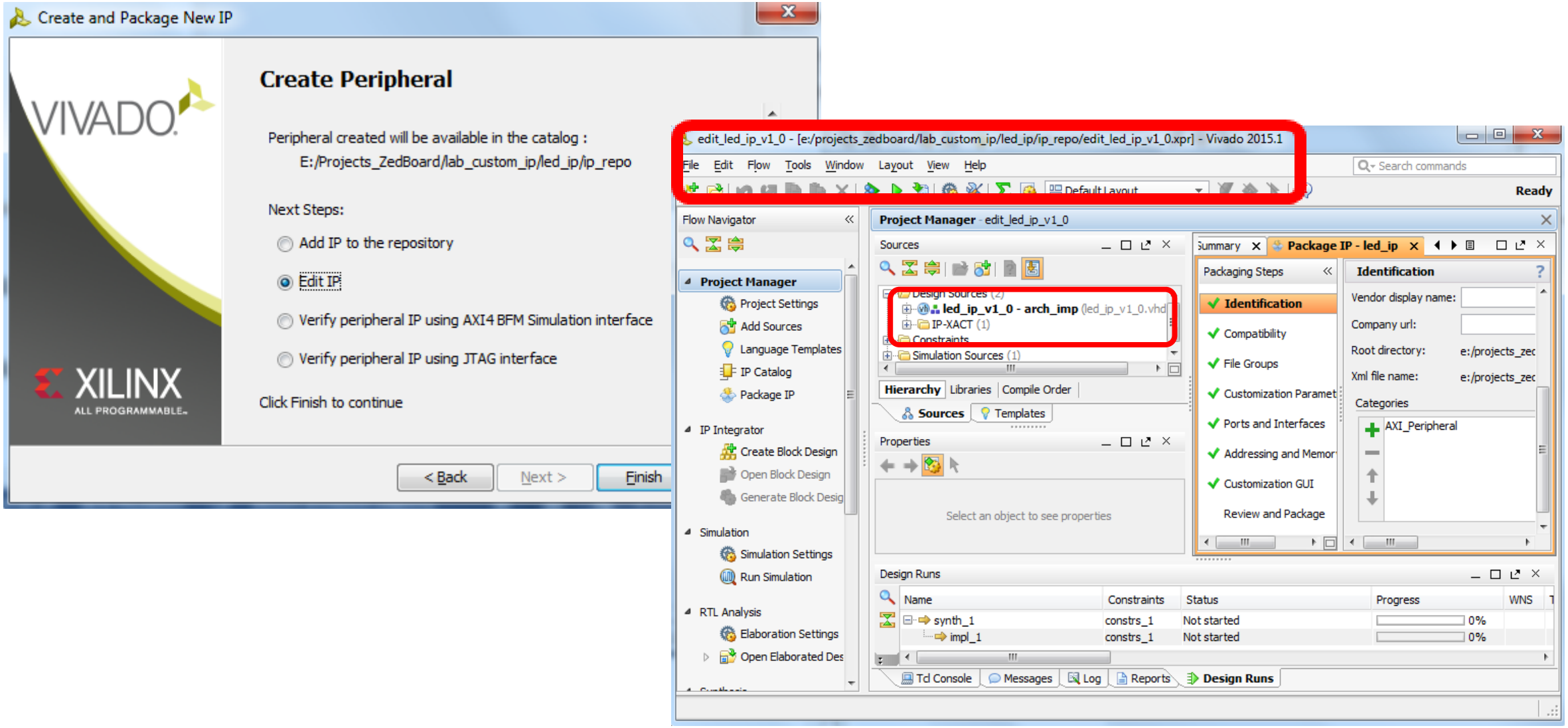


Create Custom AXI4 IP

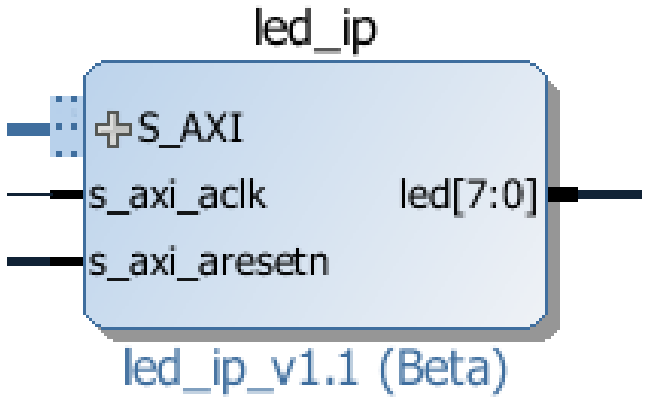




Edit Created Custom AXI4 IP



Edit Created Custom AXI4 IP



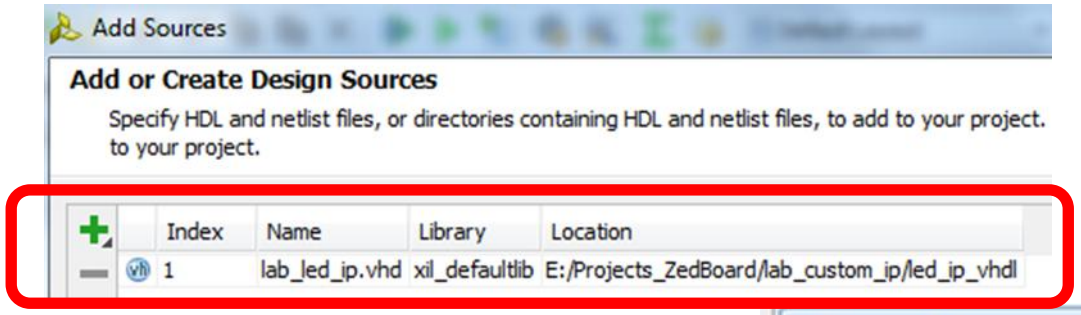
led_ip_v1_0.vhd | led_ip_v1_0_S_AXI.vhd | lab_l

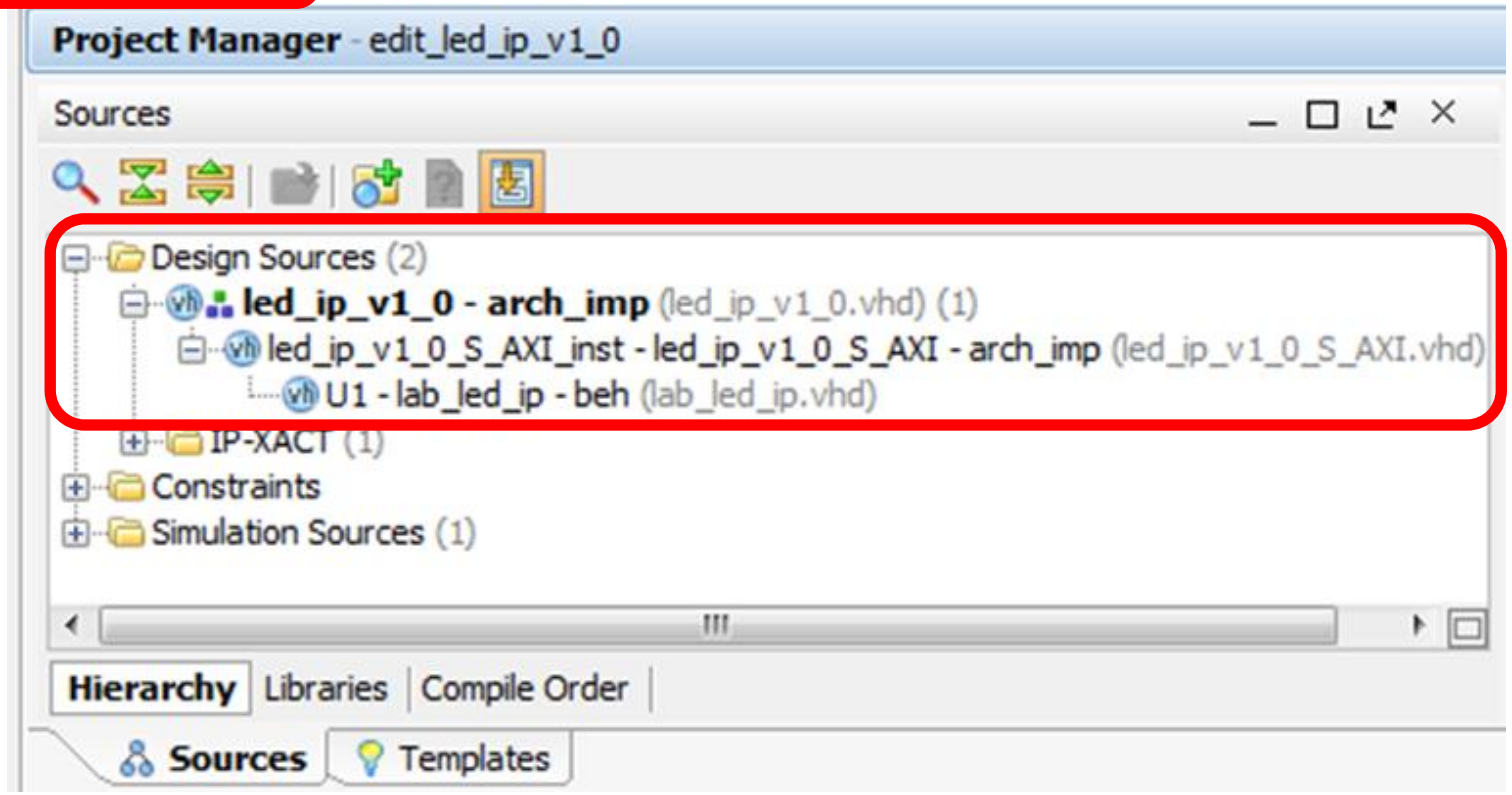
e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0/hdl/led_ip_v1_0.vhd

```
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
```

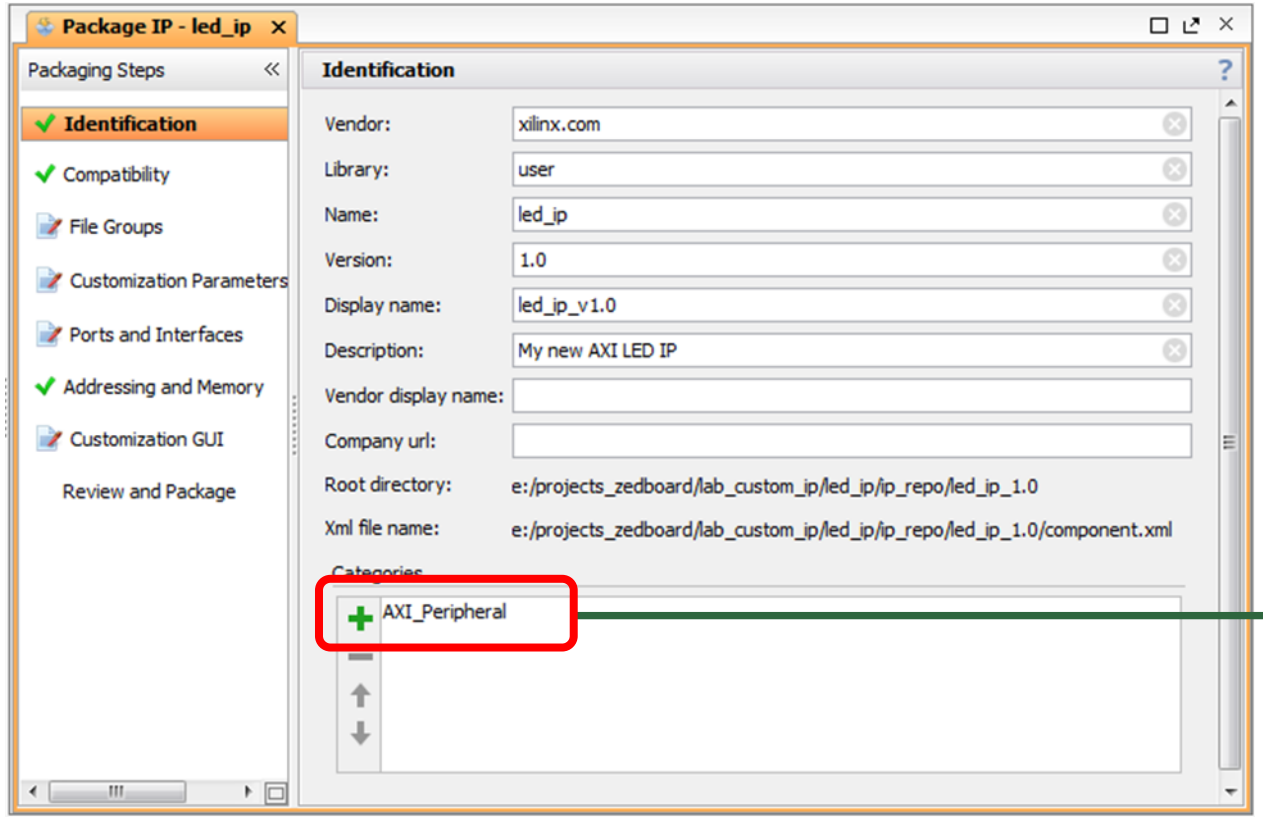
```
381     -- Add user logic here
382     U1: entity work.lab_led_ip generic map(led_width => led_width)
383                     port map(
384                         S_AXI_ACLK    => S_AXI_ACLK,
385                         SLV_REG_WREN  => SLV_REG_WREN,
386                         AXI_AWADDR    =>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

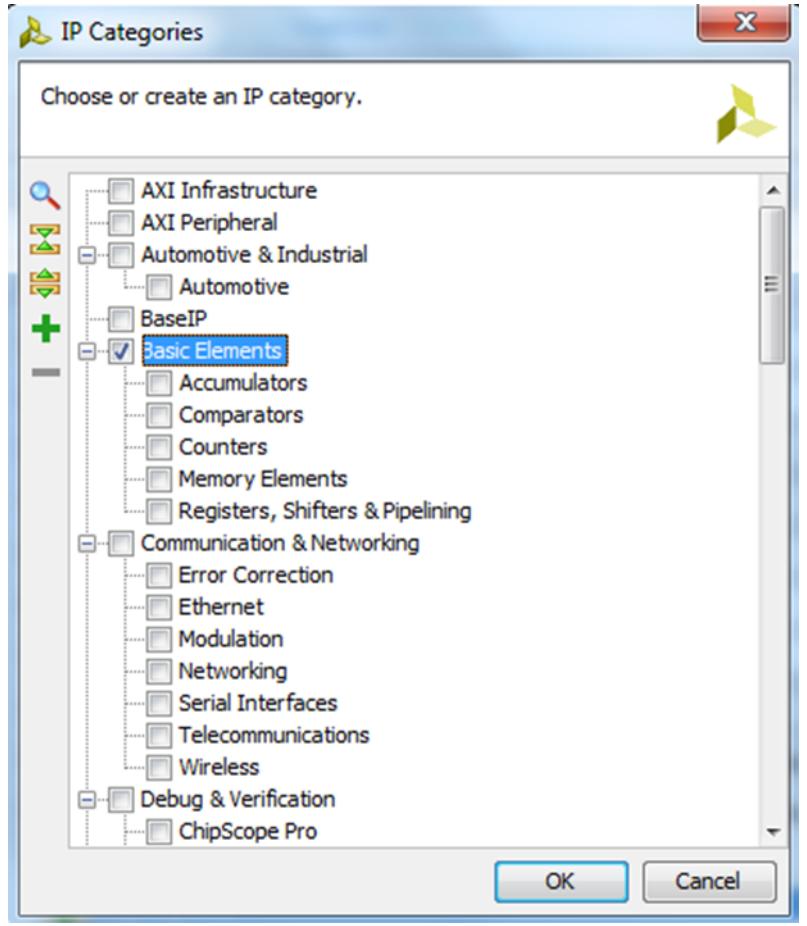
Hierarchy of My IP



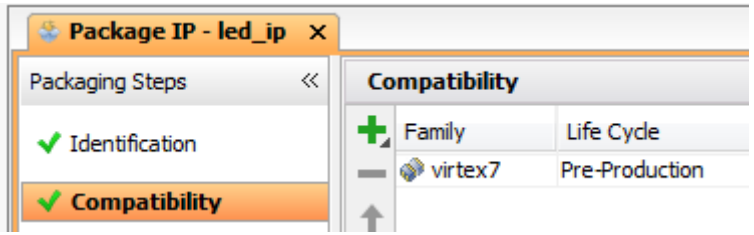


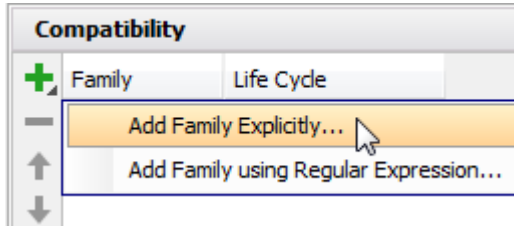
Package the IP

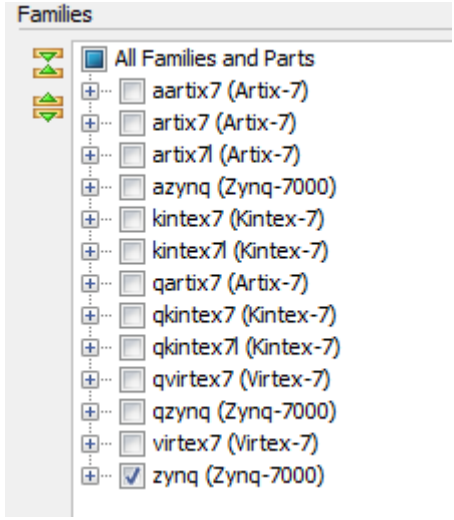




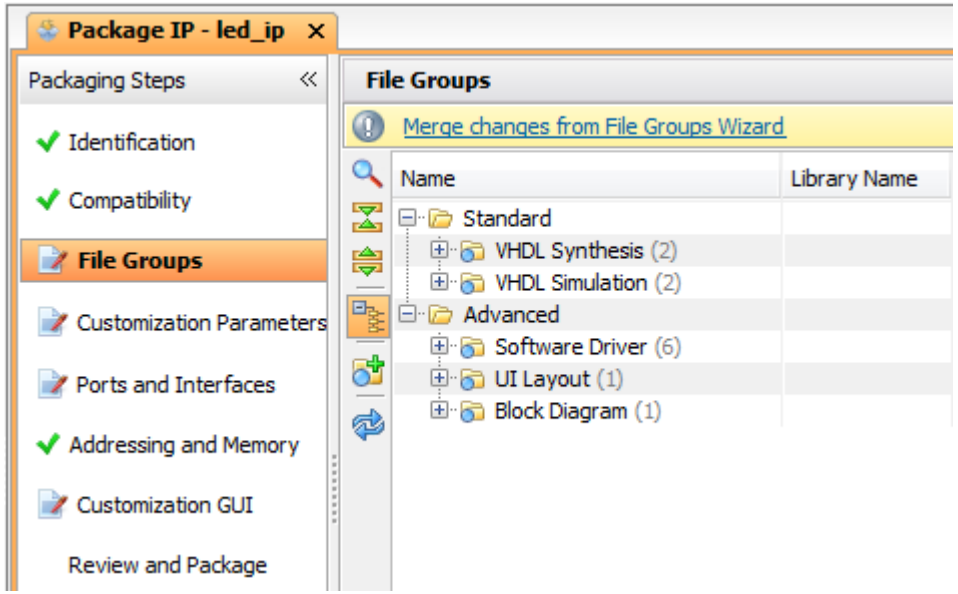
Compatibility of My IP



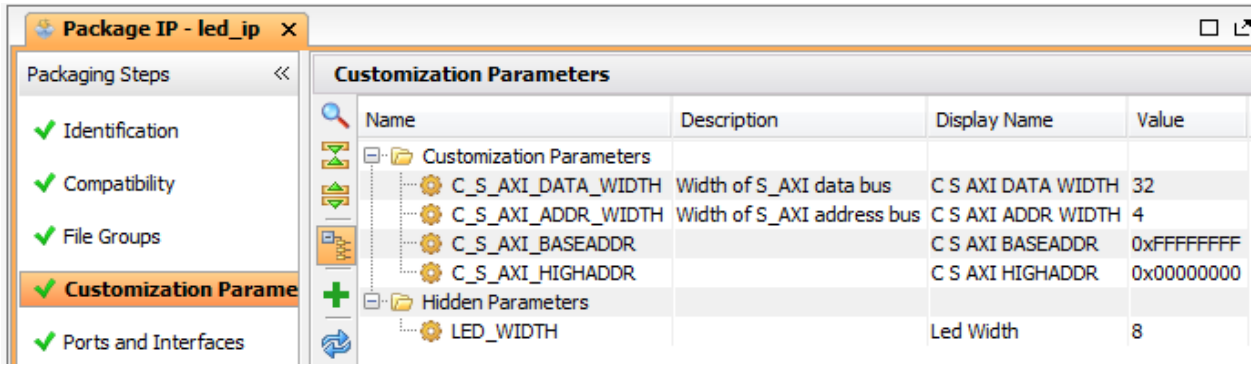


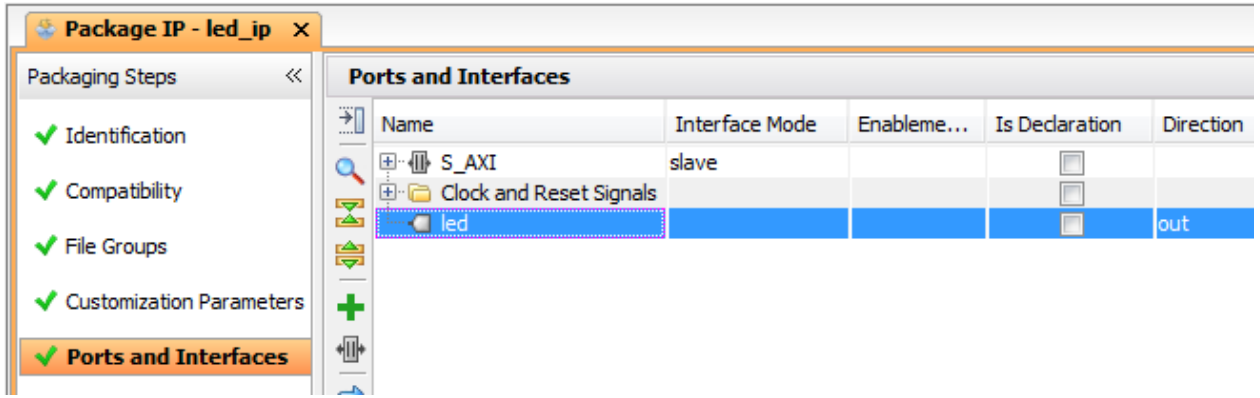


Updating Generated Files



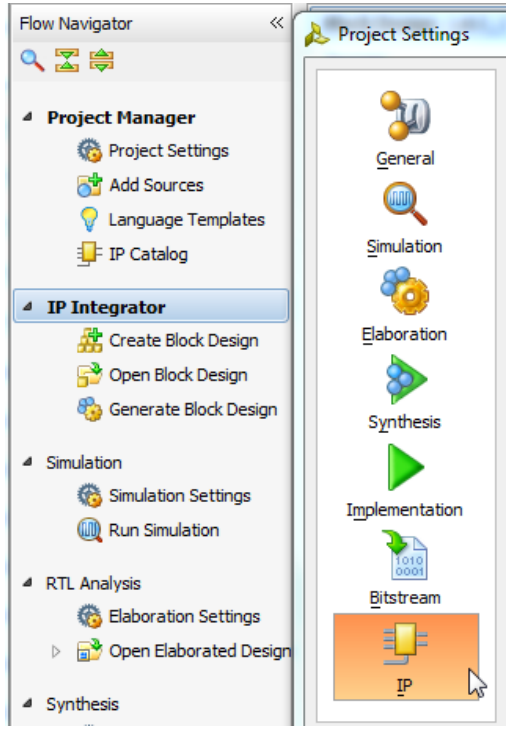
Checking Parameters and I/O Ports

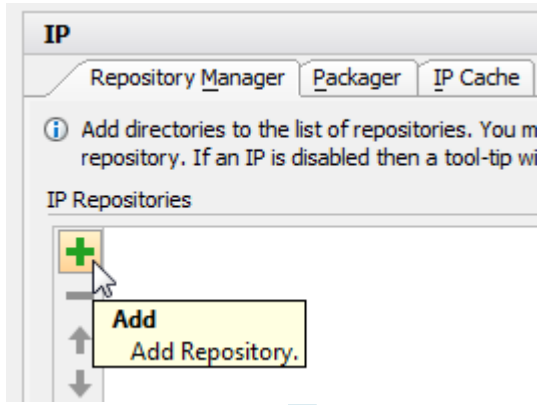


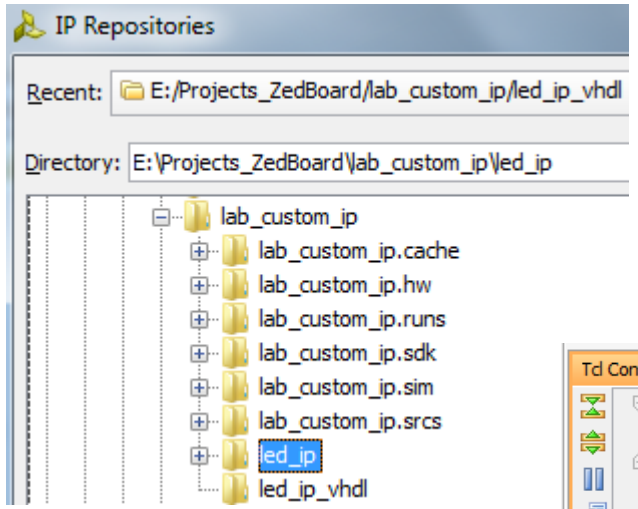


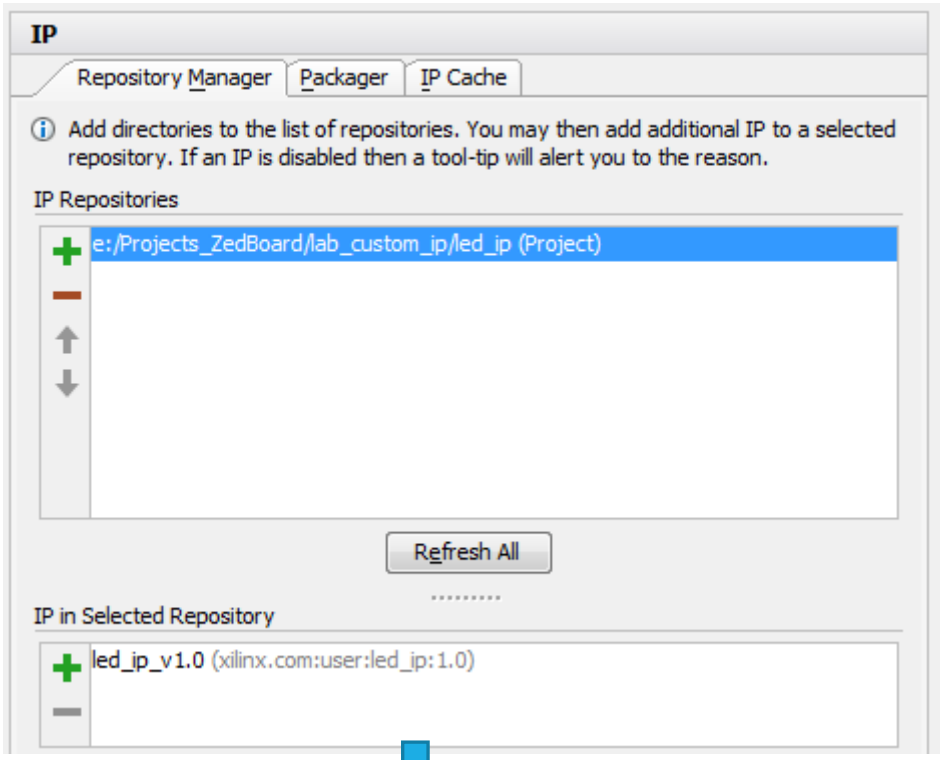
(This ends the Works on the edit_ip environment)

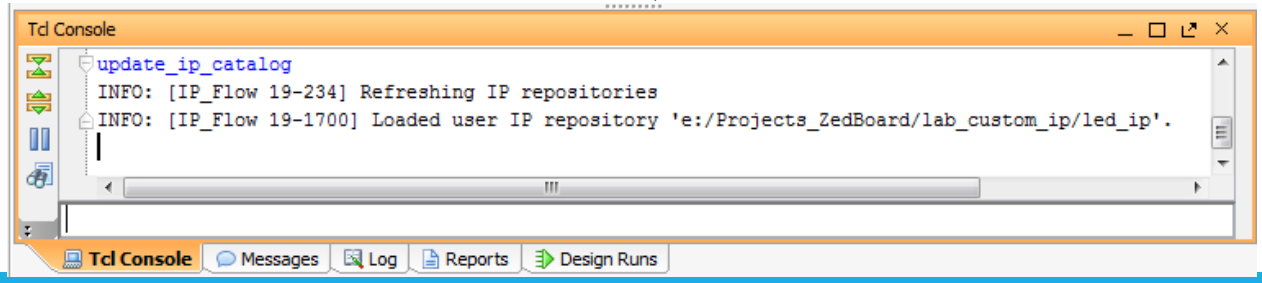
Add My IP to the Repository





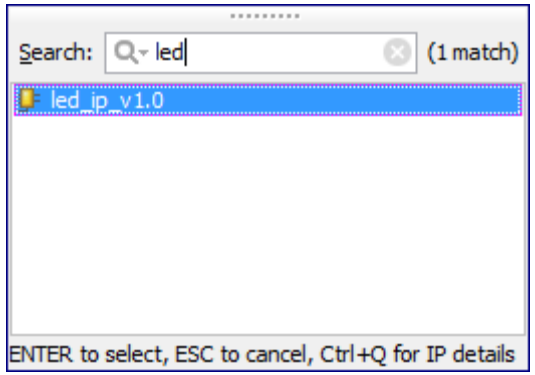


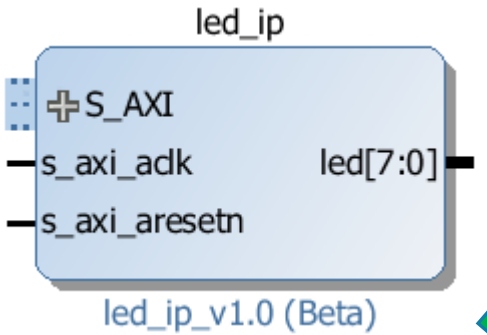




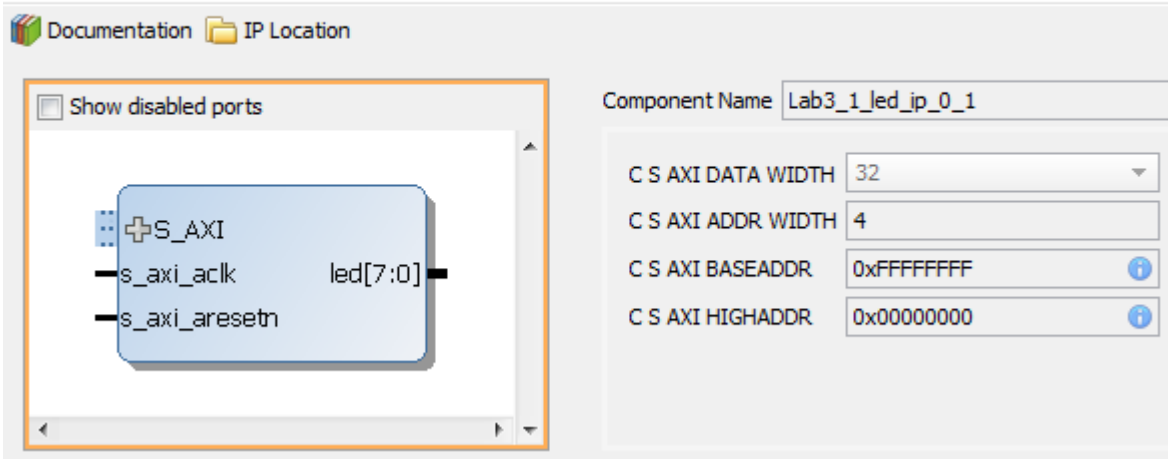
These steps shold be done in the Vivado Environment

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

hdl
- Verilog/VHDL source

xgui
- GUI tcl file

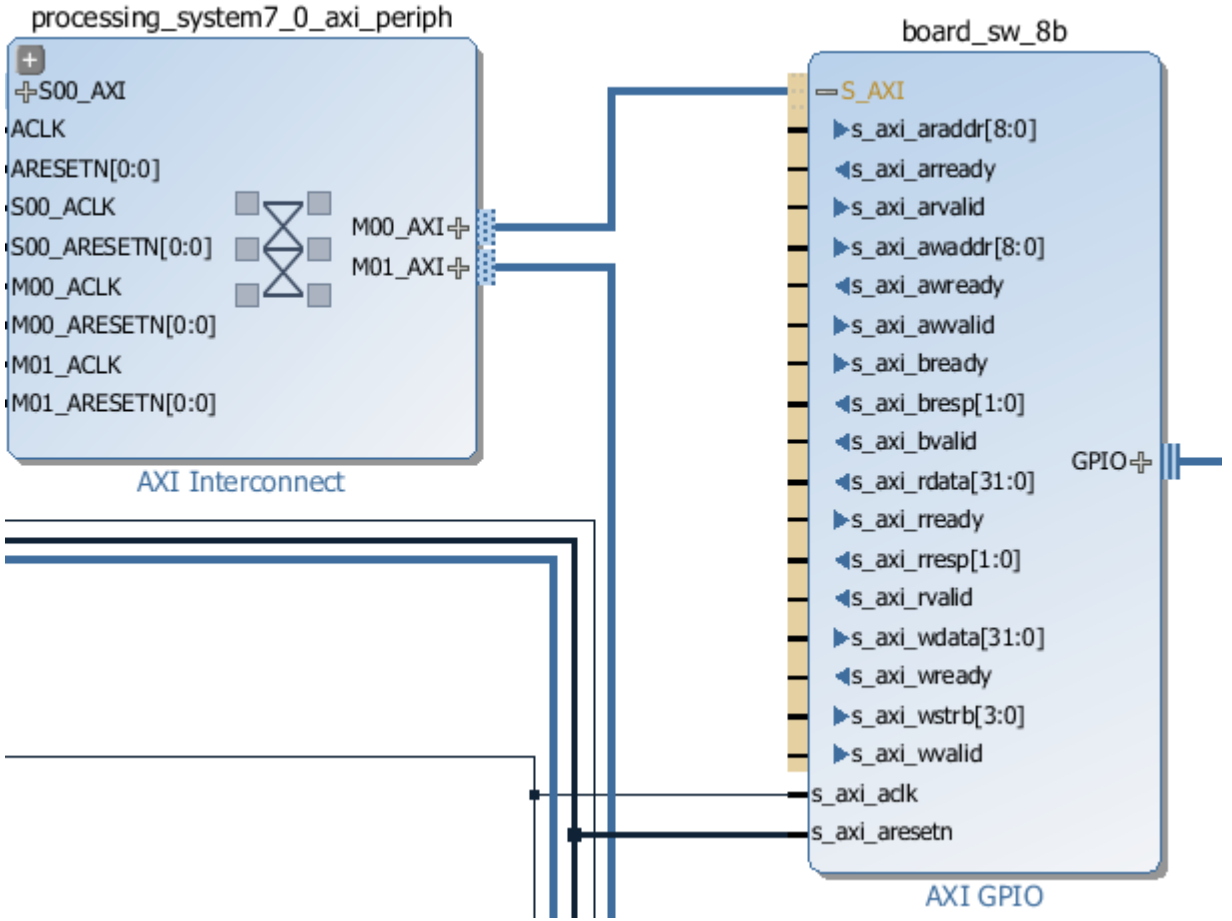
```
XStatus LED_IP_Reg_SelfTest(void * baseaddr_p)
{
    ------------------------------------------------------------

    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 Vitis. Creating a Application and BSP project
  - Write the “C” code to Wr/Rd the IP Cores registers
  - Edit Space

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).

```
-- Ports of Axi Slave Bus Interface S_AXI
-- Clock and Reset
s_axi_aclk     : in std_logic;
s_axi_aresetn  : in std_logic;
-- Write Addres Channel
s_axi_awaddr   : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
s_axi_awprot   : in std_logic_vector(2 downto 0);
s_axi_awvalid  : in std_logic;
s_axi_awready  : out std_logic;
-- Write Data Channel
s_axi_wdata : in std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
s_axi_wstrb : in std_logic_vector((C_S_AXI_DATA_WIDTH/8)-1 downto 0);
s_axi_wvalid   : in std_logic;
s_axi_wready   : out 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;
-- Read Address Channel
s_axi_araddr   : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
s_axi_arvalid  : in std_logic;
s_axi_arready  : out std_logic;
-- Read Data Channel
s_axi_rdata : out std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
s_axi_rresp : out std_logic_vector(1 downto 0);
s_axi_rvalid   : out std_logic;
s_axi_rready   : 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 – 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;
```