Joint ICTP-IAEA School on Systems-on-Chip based on FPGA for Scientific Instrumentation and Reconfigurable Computing

Introduction to AXI – Custom IP

Cristian Sisterna

Universidad Nacional San Juan

Argentina

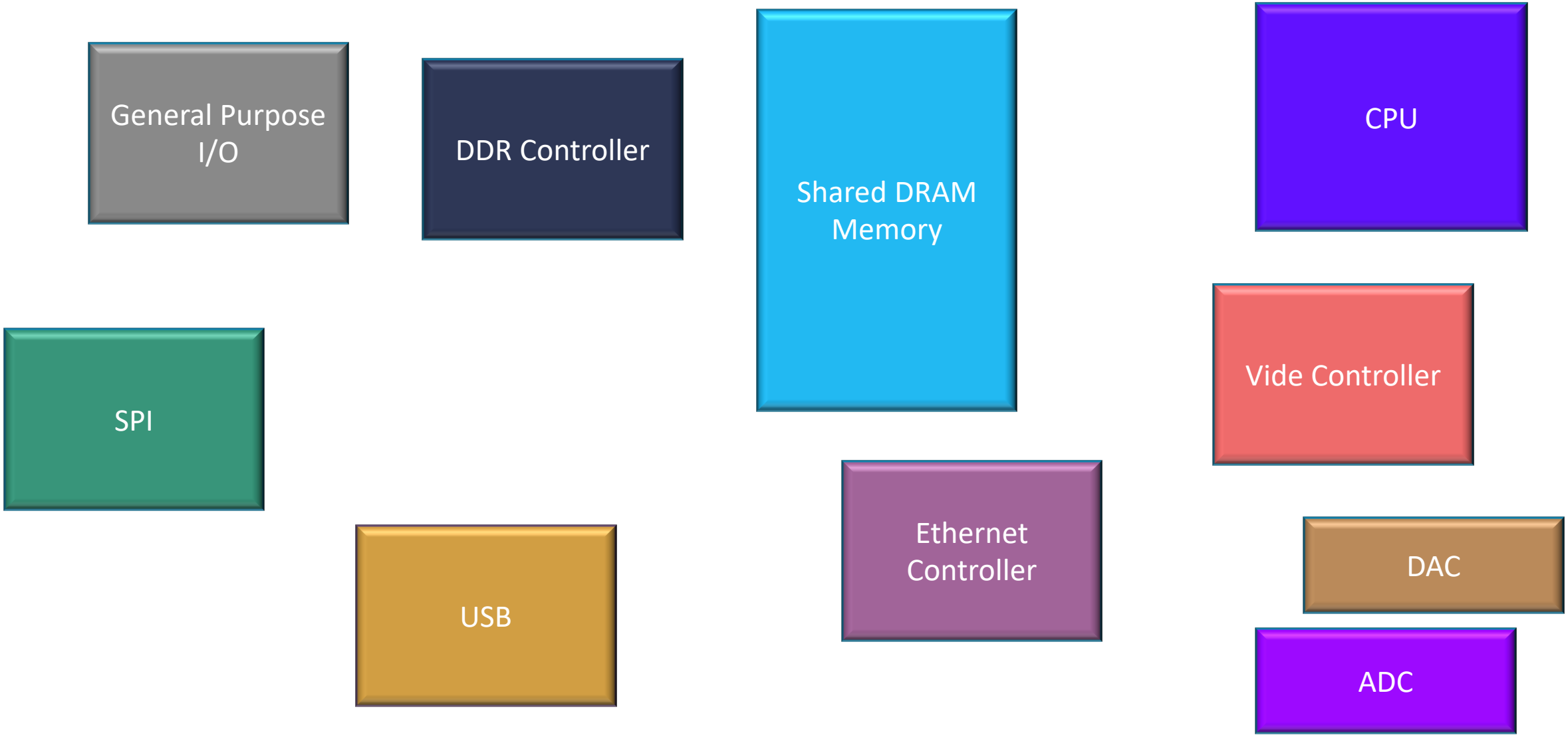
Agenda

- 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

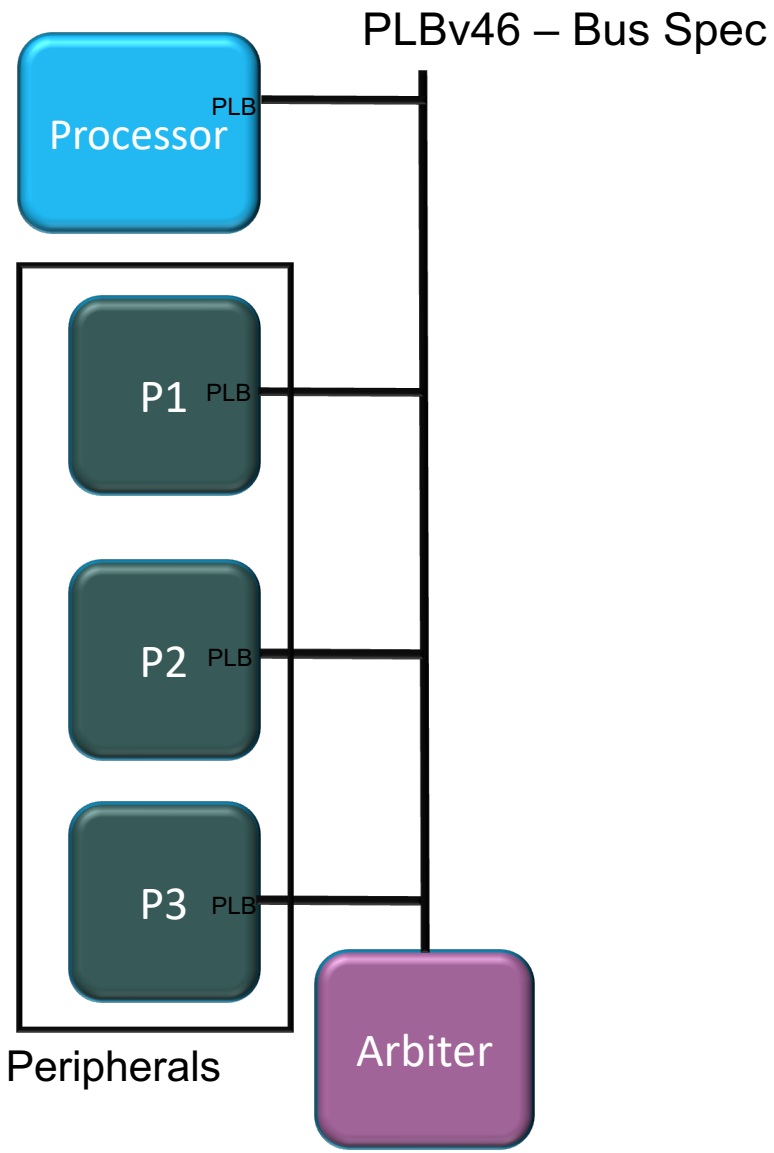
Need to Understand Device's Connectivity

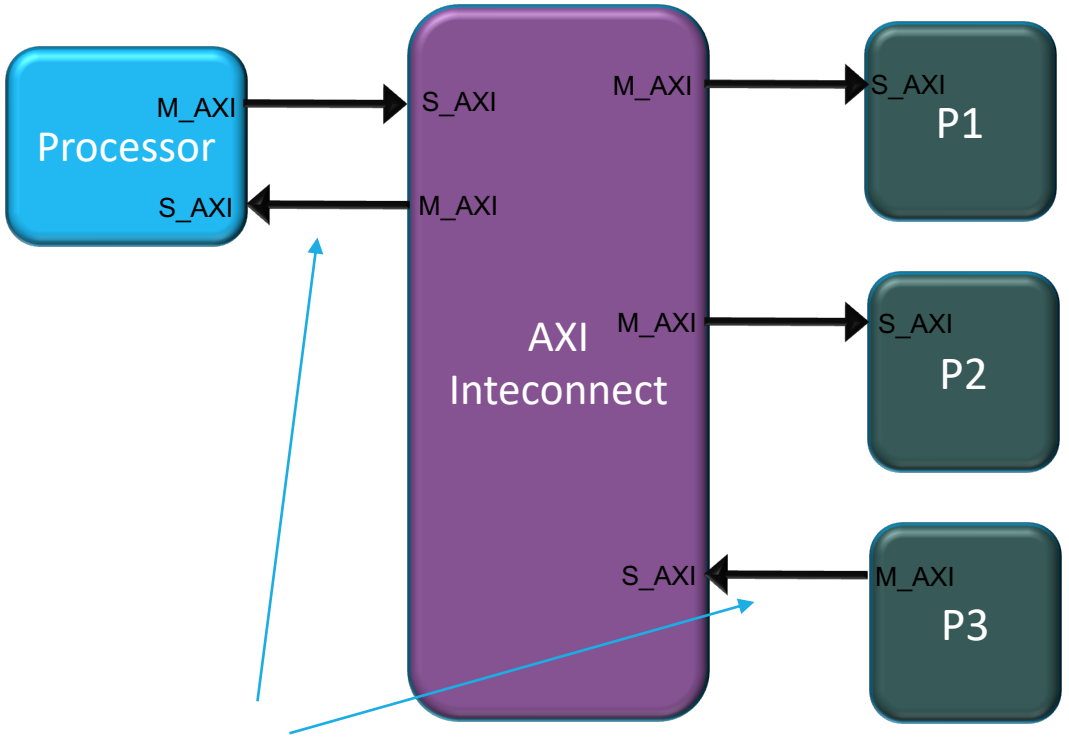
- There is a need to get familiar with the way that different devices communicate each other in an Embedded System like a Zynq based system
- Learning and understanding the communication among devices will facilitate the design of Zynq based systems
- All the devices in a Zynq system communicate each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





AXI4 Defines a Point to Point Master/Slave Interface

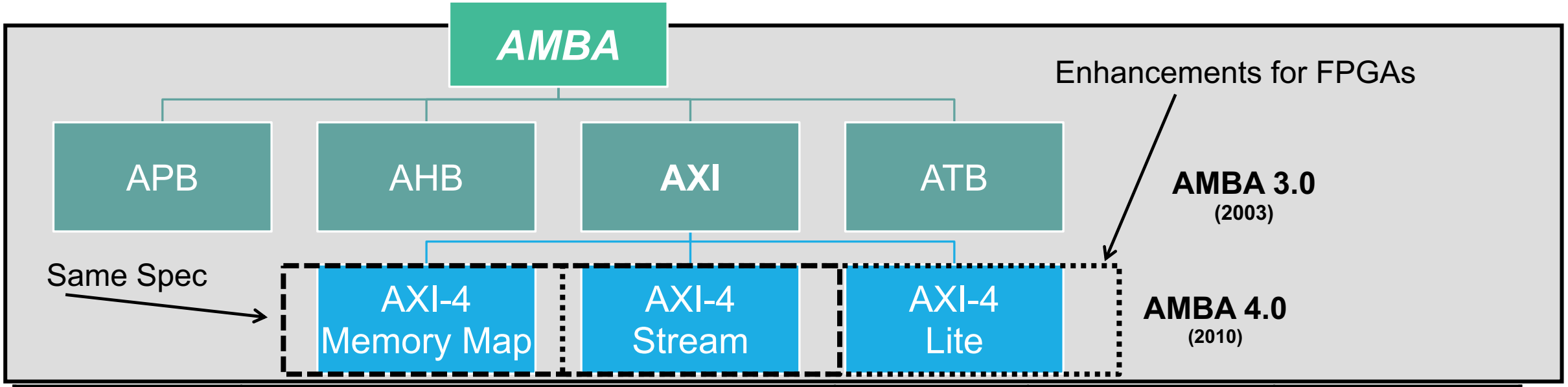
Connectivity -> Standard

- **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 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

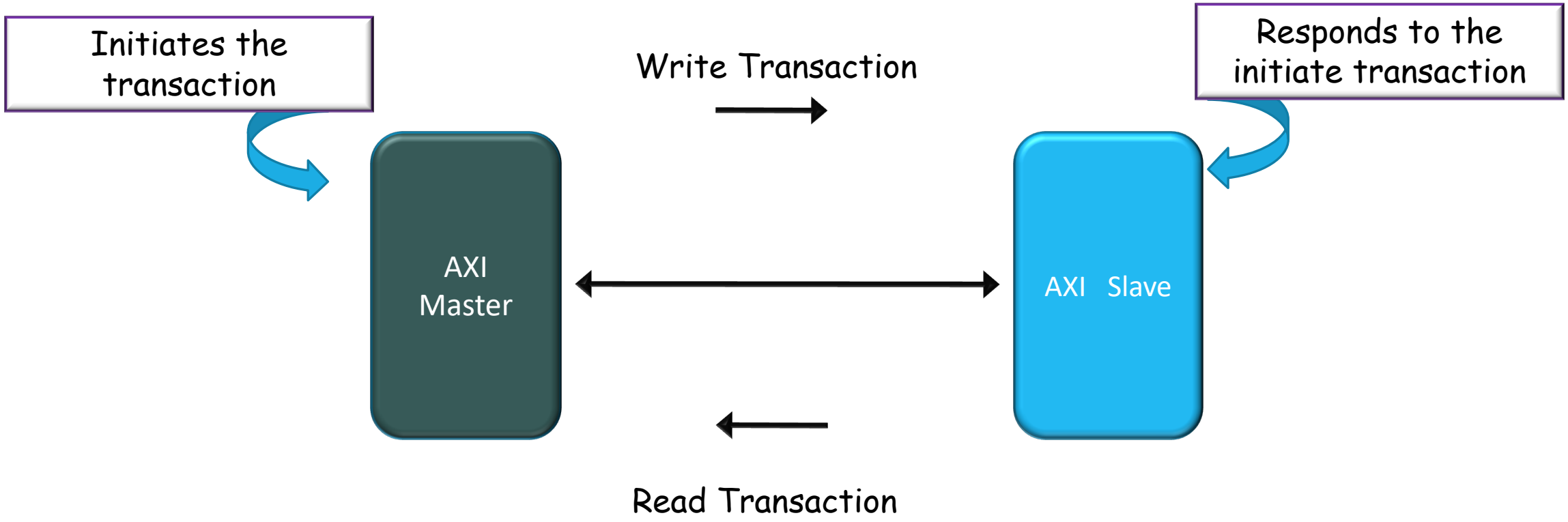
Transaction

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

Burst

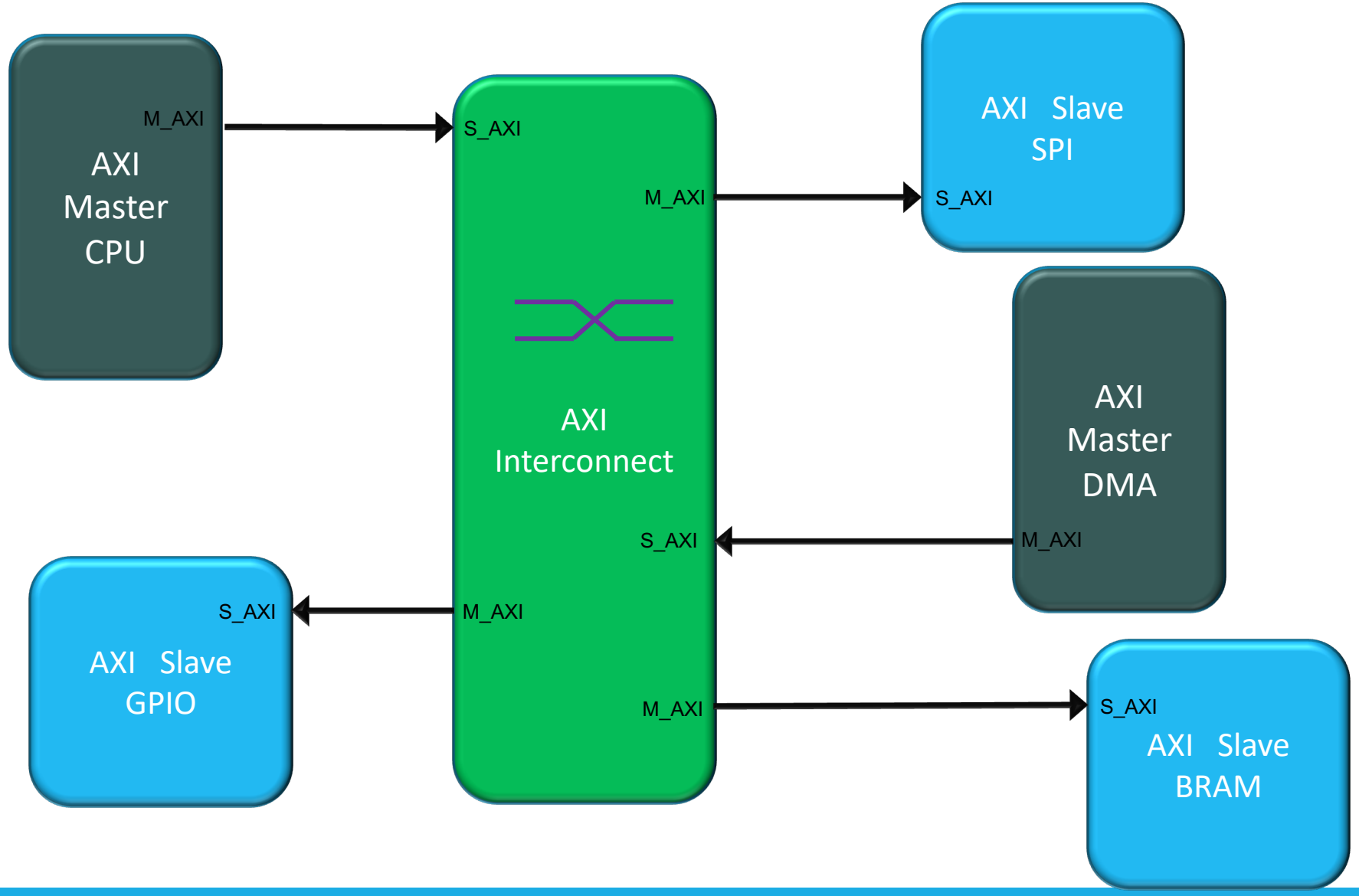
- Transaction that consists of more than one transfer

AXI Transactions / Master-Slave



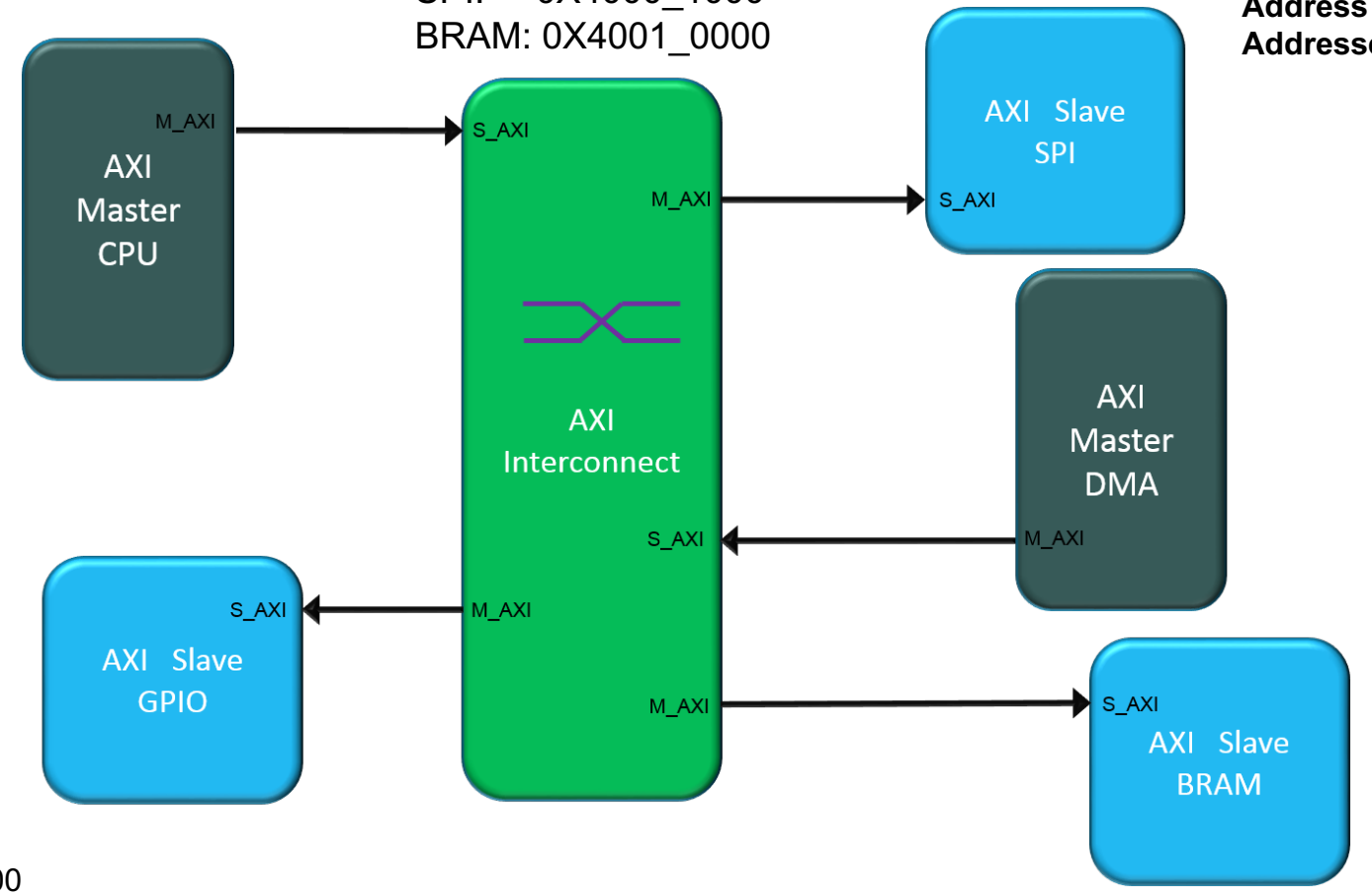
Transactions: transfer of data from one point on the hardware to another point

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

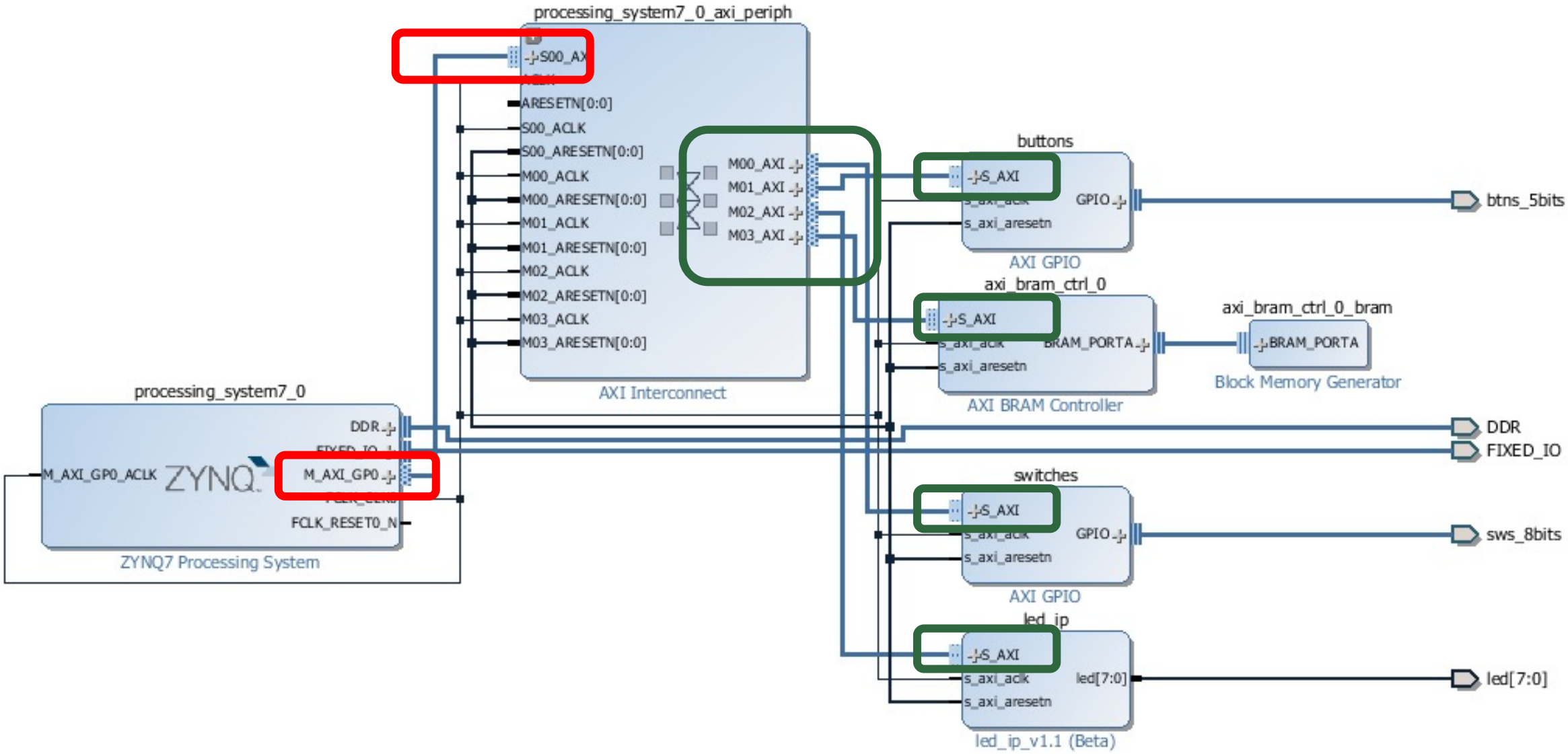
Address Range: 4K
Address Offset: 0X4000_0000
Addresses: 0X4000_0000 – 0X4000_0FFF

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

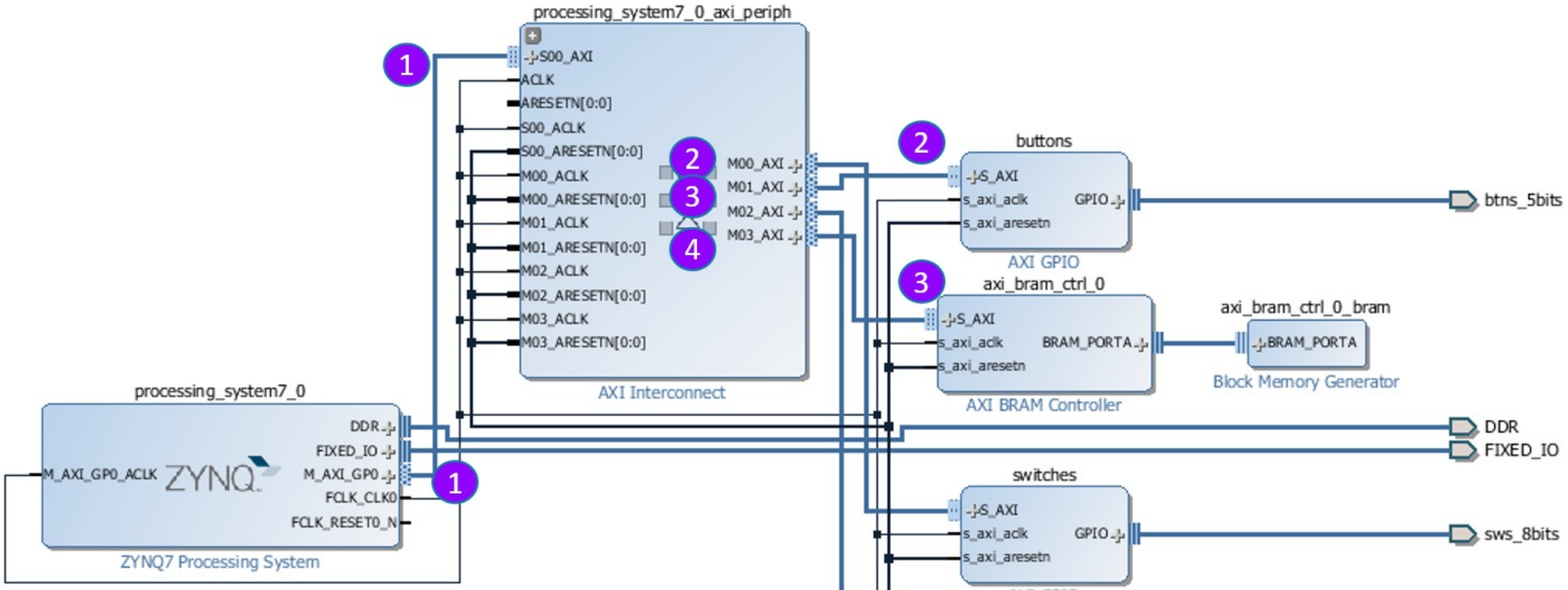
AXI Interconnect Main Features

- 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



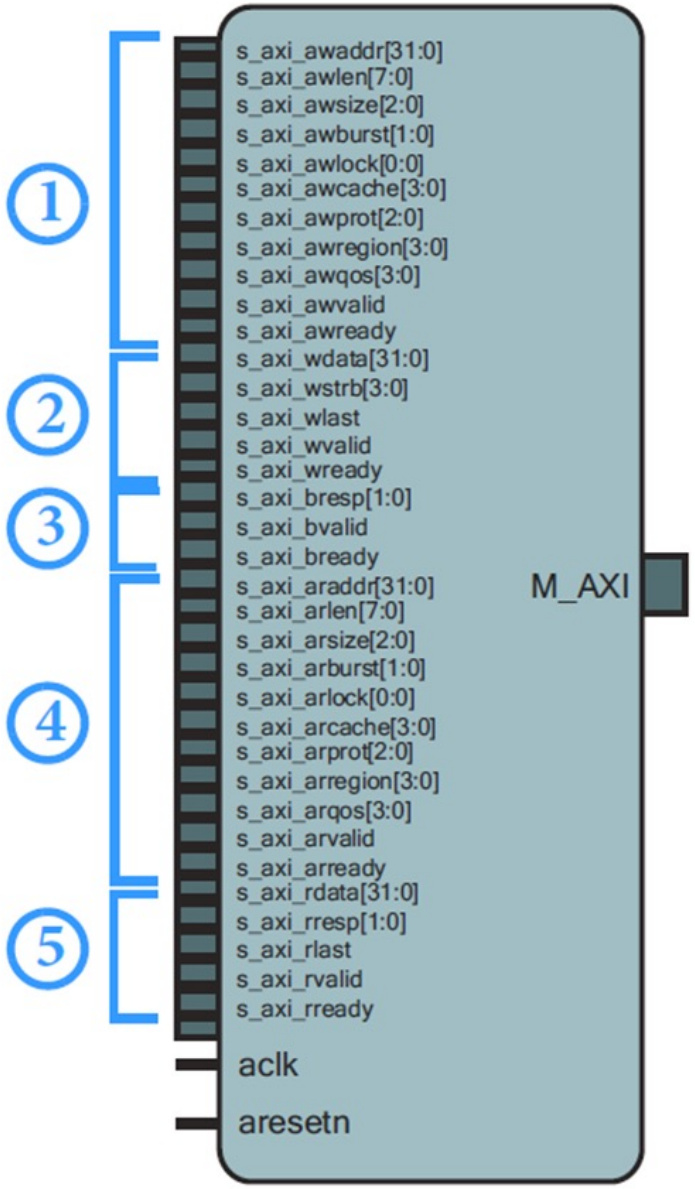
AXI Interface Example



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

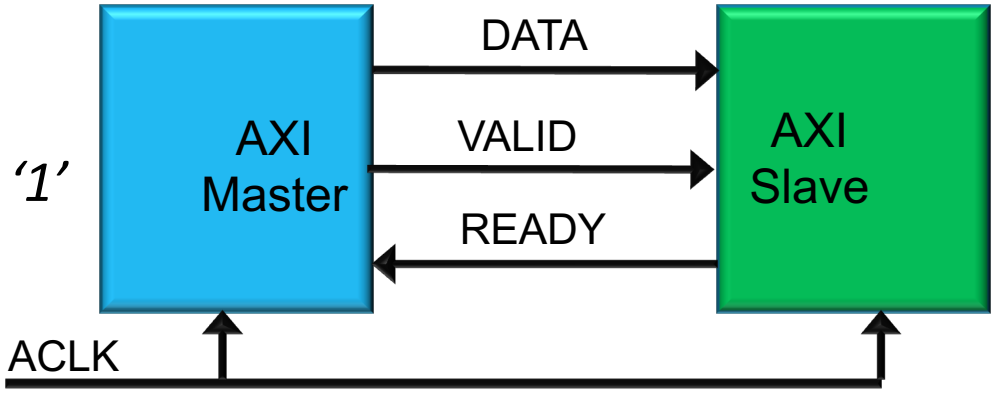


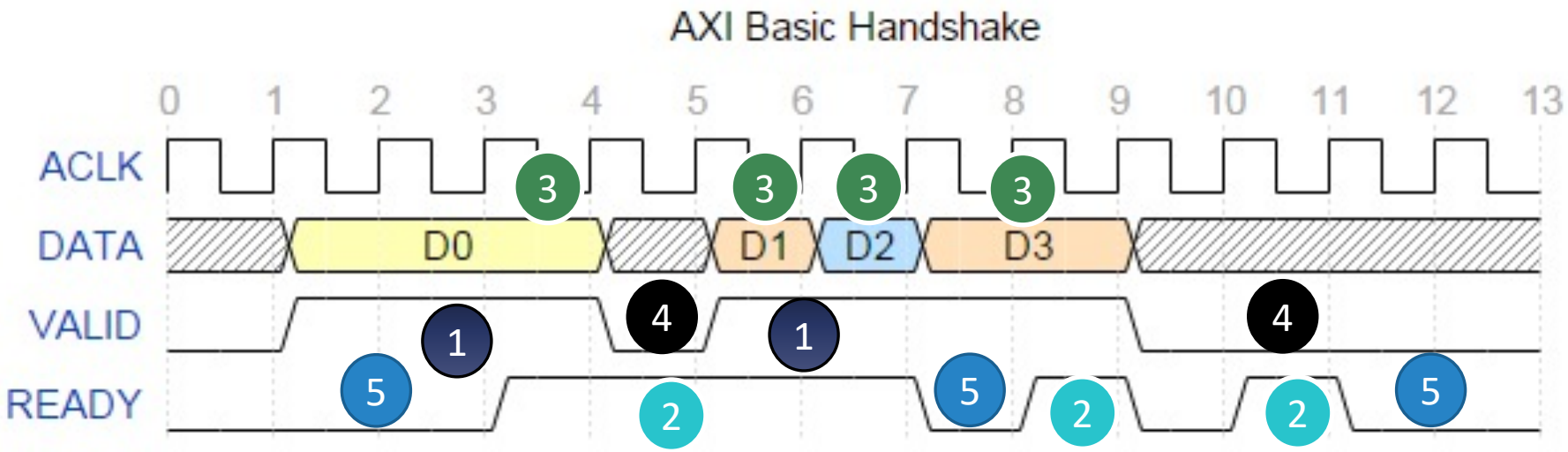
1. **Write Address Channel** — the signals contained within this channel are named in the format *s_axi_aw...*
2. **Write Data Channel** — the signals contained within this channel are named in the format *s_axi_w...*
3. **Write Response Channel** — the signals contained within this channel are named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

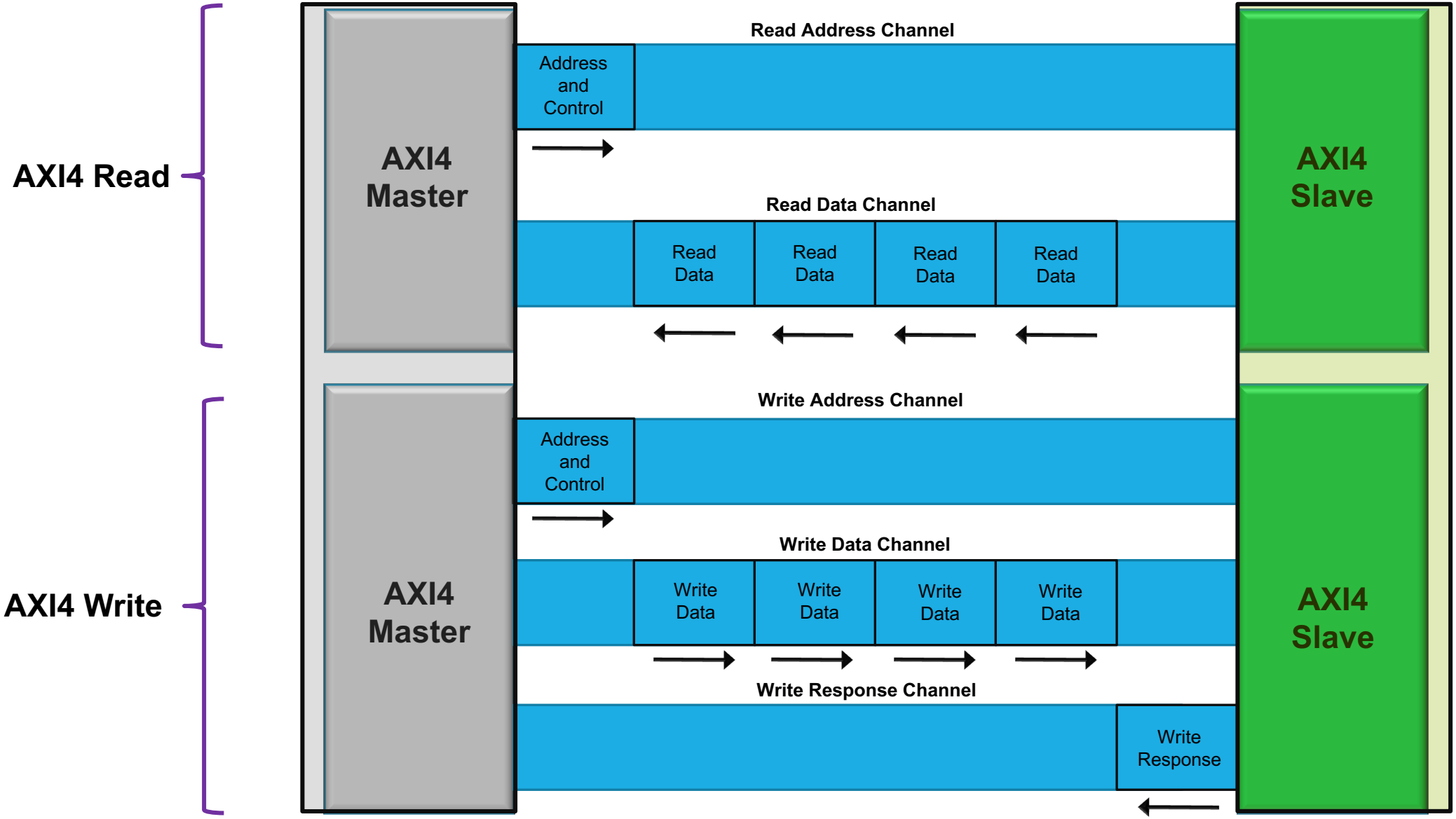




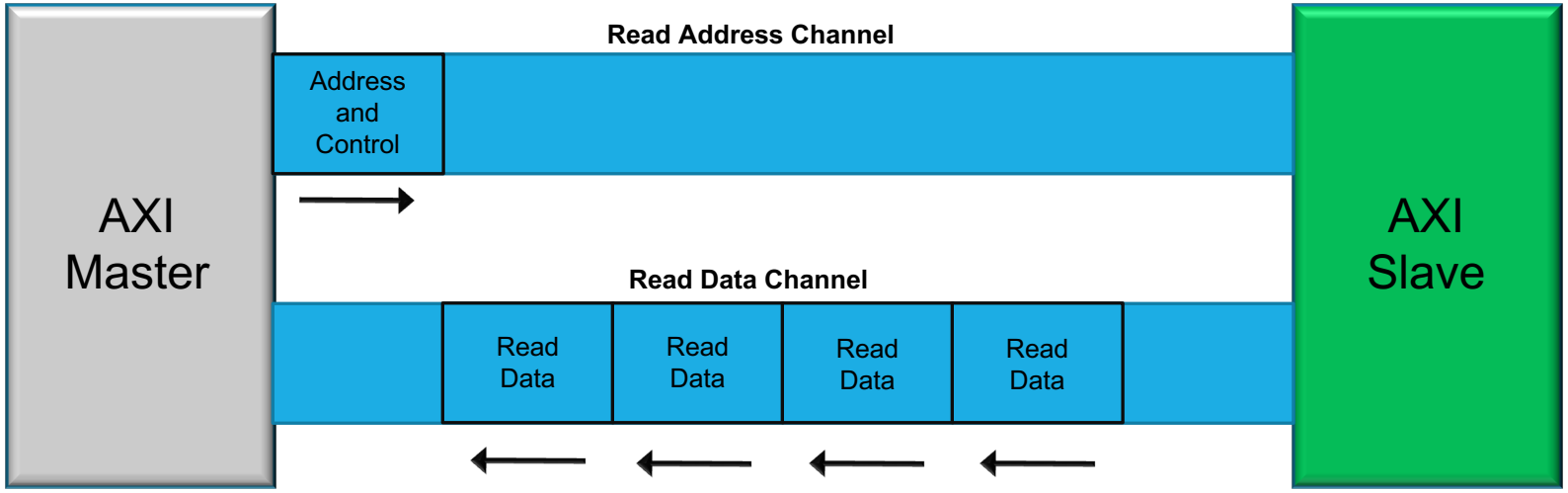
AXI4 Lite

- 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

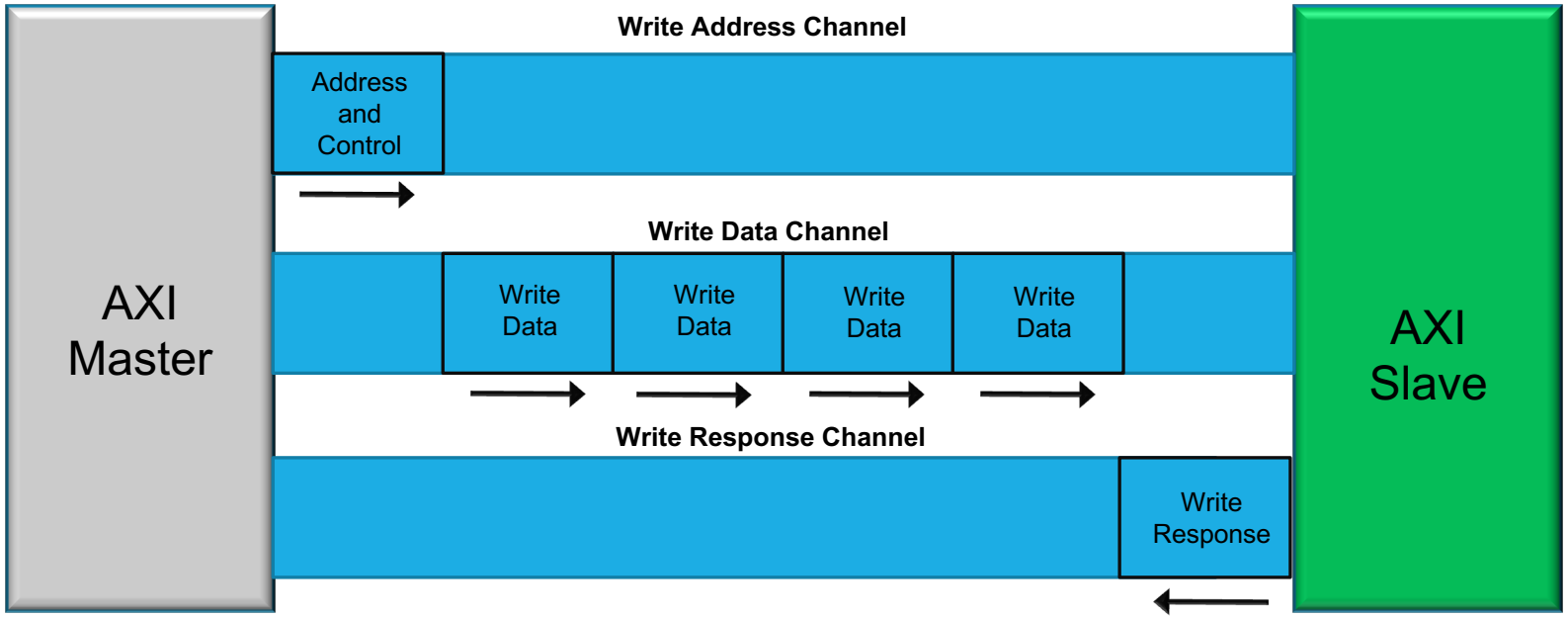
AXI Channels



AXI4 Lite Read



AXI4 Lite Write

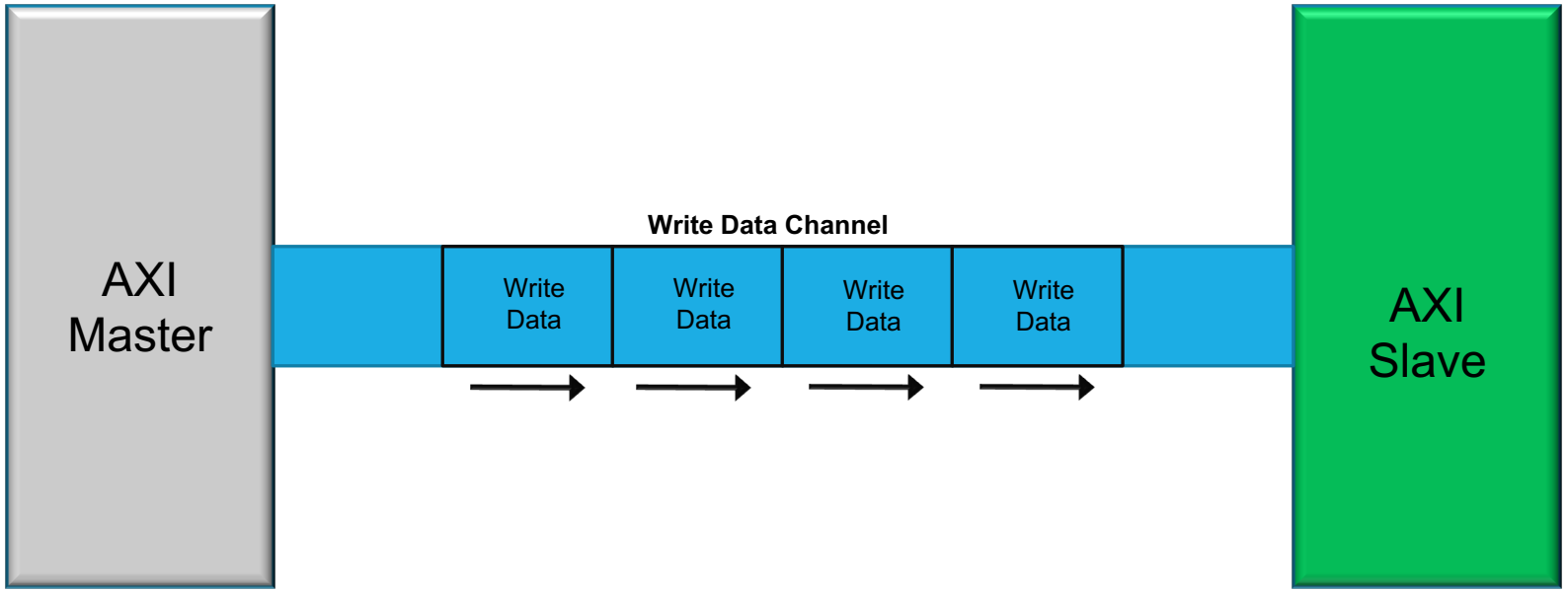


AXI4 (Full)

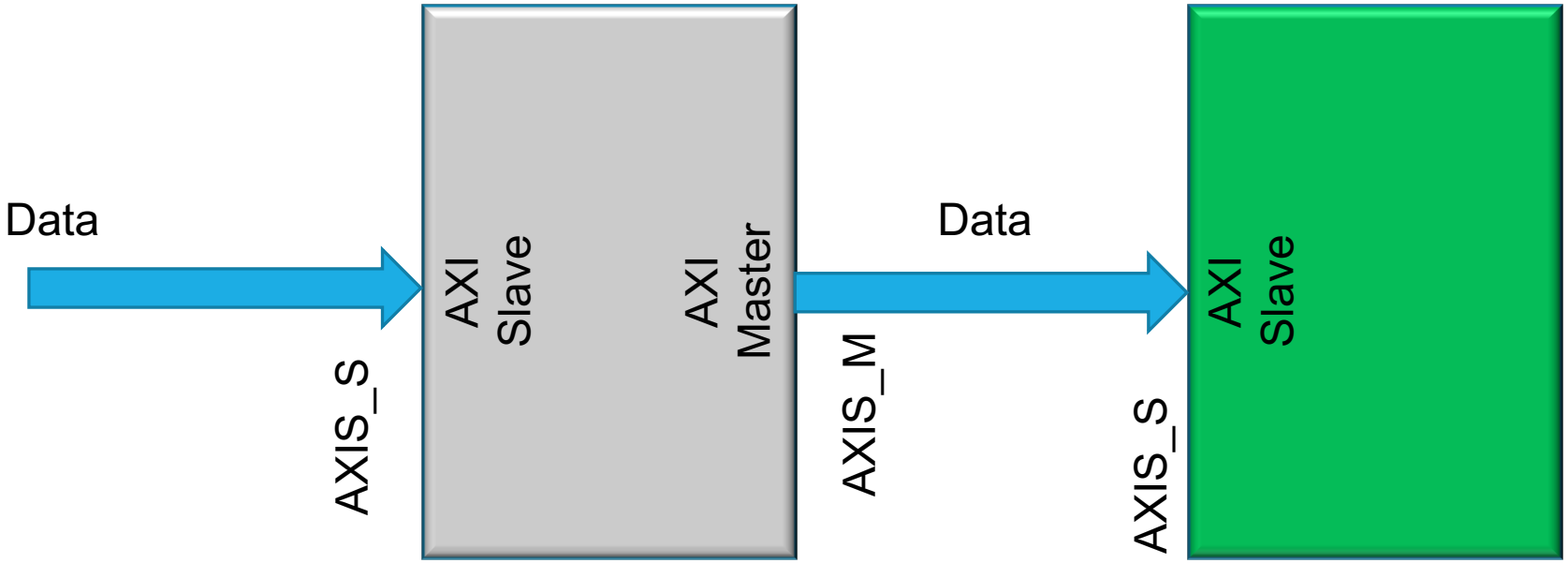
- Sometimes called "*Full AXI*" or "*AXI Memory Mapped*"
- Single address multiple data
  - Burst up to 256 data
- Data Width parameterizable
  - 32, 64, 128, 256, 512, 1024 bits

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



AXI Stream



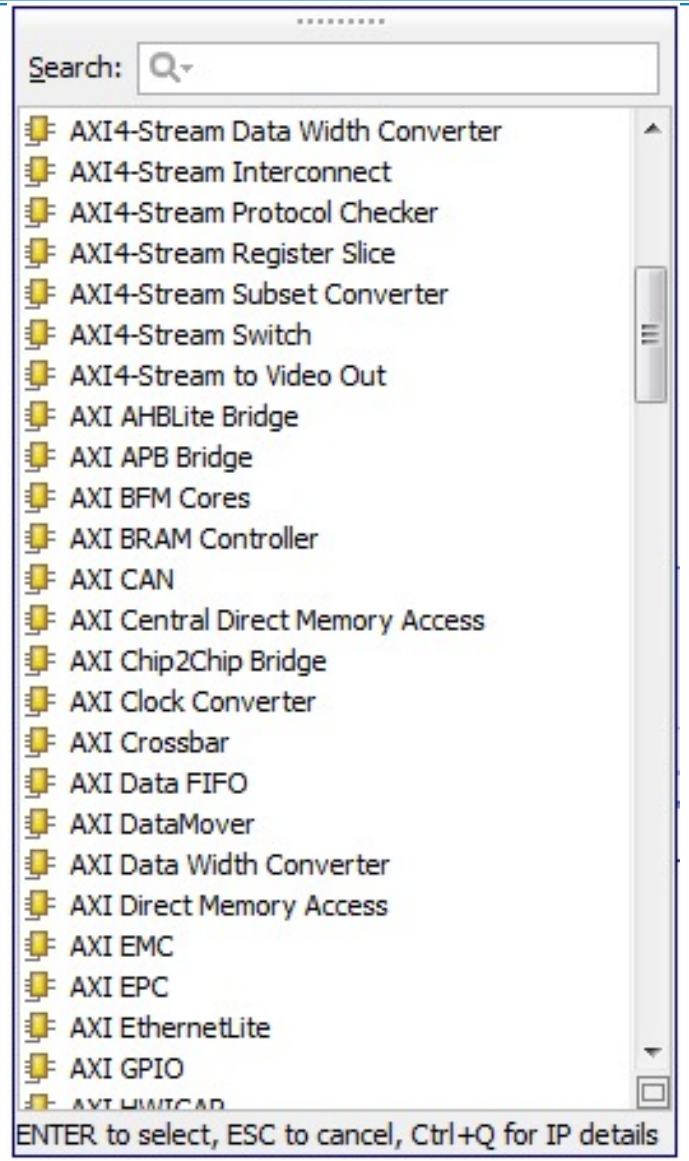
Custom AXI IPs

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

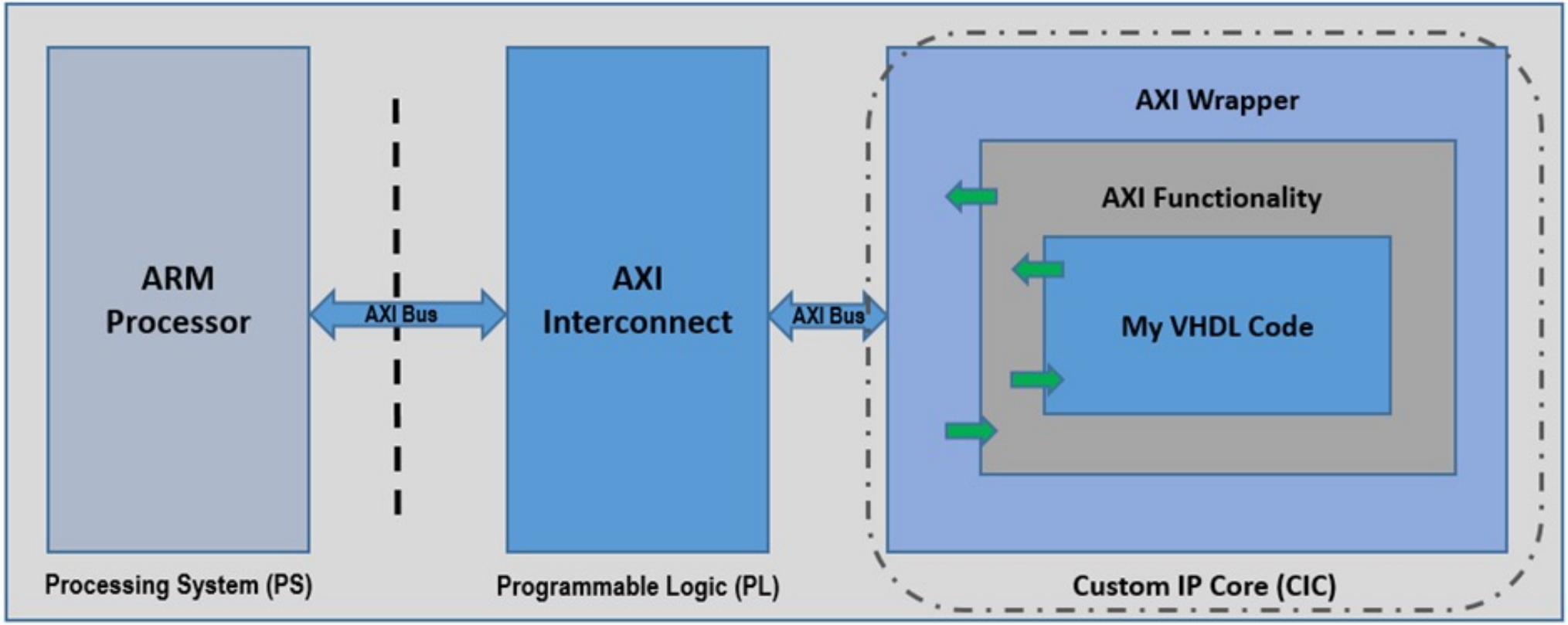
- Consistent, easy access
- Support for multiple physical locations, including shared network drives
- Access to the latest version of Xilinx-delivered IP
- Access to IP customization and generation using the Vivado IDE
- IP example designs
- Catalog filter options that let you filter by Supported Output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

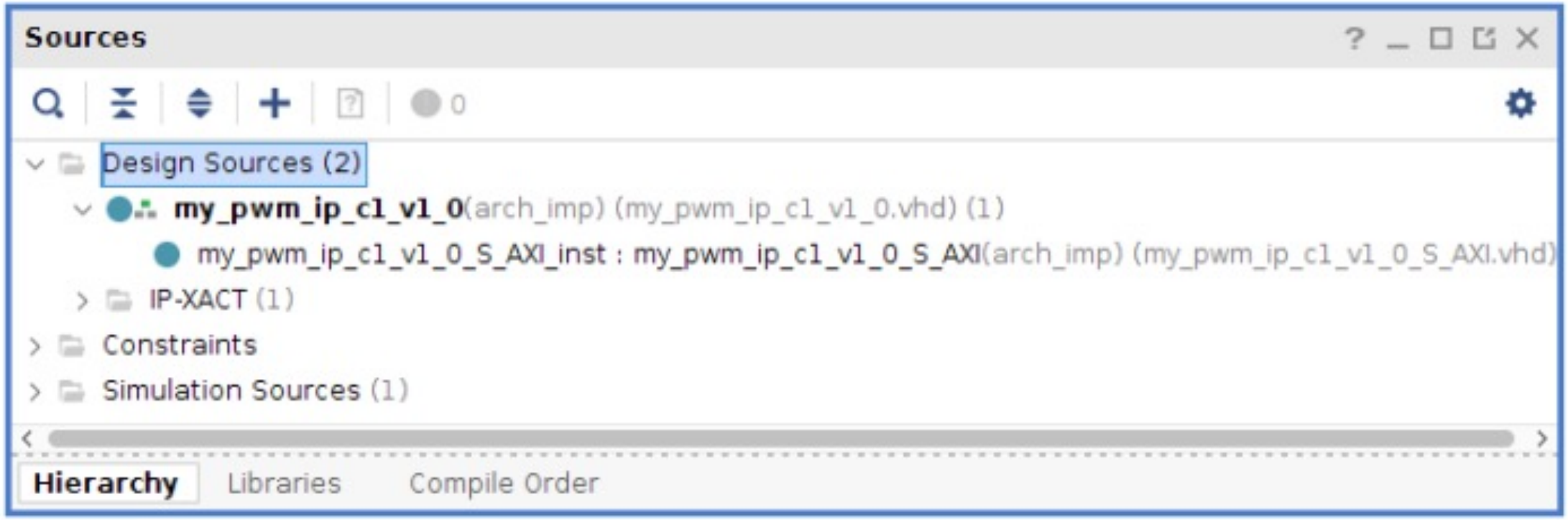

ARM
Processor
AXI Bus
AXI
Interconnect
AXI Bus
AXI Wrapper
AXI Functionality
My VHDL Code
Processing System (PS)
Programmable Logic (PL)
Custom IP Core (CIC)

Simple Case – Case 1

```
entity pwm_simple is

  generic (
    dc_bits : integer := 16);            -- number of bits f
  port (
    -- clock & reset signals
    S_AXI_ACLK        : in  std_logic;   -- AXI clock
    S_AXI_ARESETN     : in  std_logic;   -- AXI reset, active
    -- control input signal
    duty_cycle        : in  std_logic_vector(31 d
    -- PWM output
    pwm               : out std_logic          --
    );
end entity pwm_simple;
```

```
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;
```

Simple Case – Case 1



my_pwm_ip_c1_0_S_AXI.vhd: This VHDL file is the one that has the AXI Lite interface.
You can open this file and try to understand the code by mainly reading the comments. etc.

Simple Case – Case 1

We need to modify the `my_pwm_ip_c1_0_S_AXI.vhd` file:

1. add the generic, **dc_bits**, to the generic part of the file
2. add the **pmw** output to the entity ports
3. add the **pwm_pr process**
4. declare **duty_cycle** as internal signal
5. assign the value of **slv_reg0** to **duty_cycle** signal
6. make the **register 0** as the one that hold the value for the duty cycle

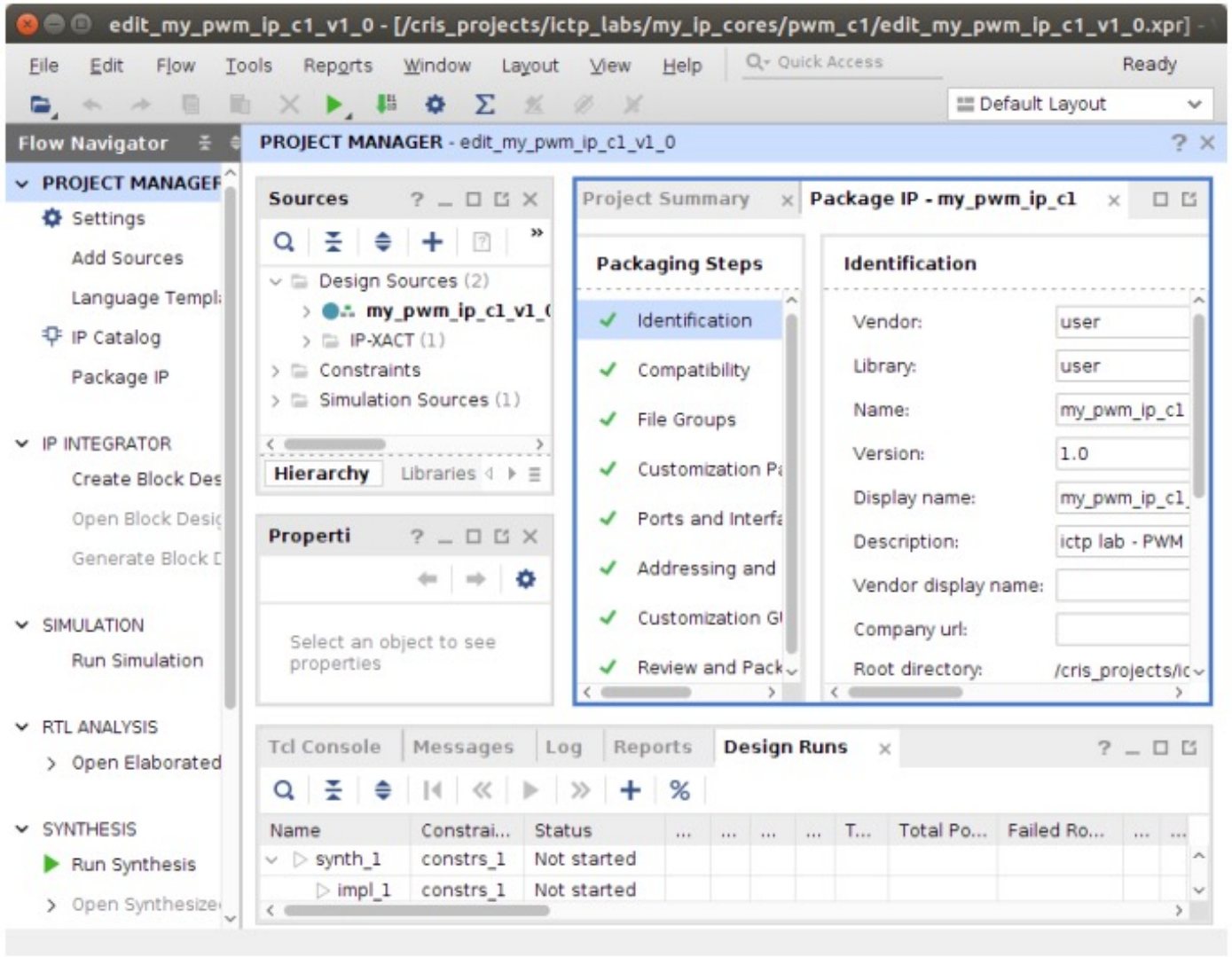
Simple Case – Case 1

my_pwm_ip_cl_v1_0_S_AXI.vhd * | my_pwm_ip_cl_v1_0.vhd

/ictp_labs/my_ip_cores/pwm_cl/my_pwm_ip_cl_1.0/hdl/my_pwm_ip_cl_v1_0_S_AXI.vhd

```
386
387     -- Add user logic here
388       pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
389        variable counter : unsigned(dc_bits-1 downto 0); -- count cloc
390     begin  -- process pwm_pr
391       if (S_AXI_ARESETN = '0') then
392         counter   := (others => '0');
393         pwm       <= '0';
394       elsif (rising_edge(S_AXI_ACLK)) then
395         counter := counter + 1
396         if (counter < unsigned(duty_cycle(dc_bits-1 downto 0))) then
397           pwm <= '1';
398         else
399           pwm <= '0';
400         end if;
401       end if;
402     end process pwm_pr;
403     -- User logic ends
404
405 end arch_imp;
```

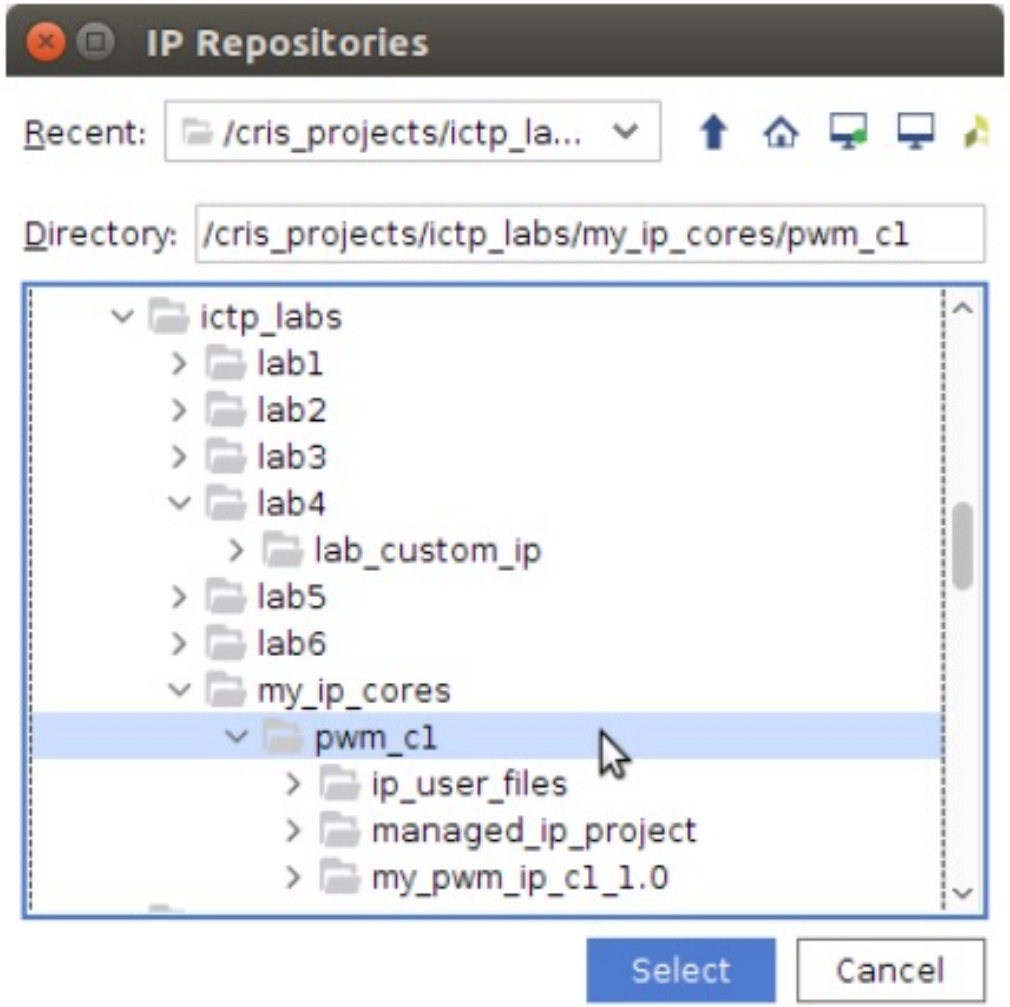
Simple Case – Case 1

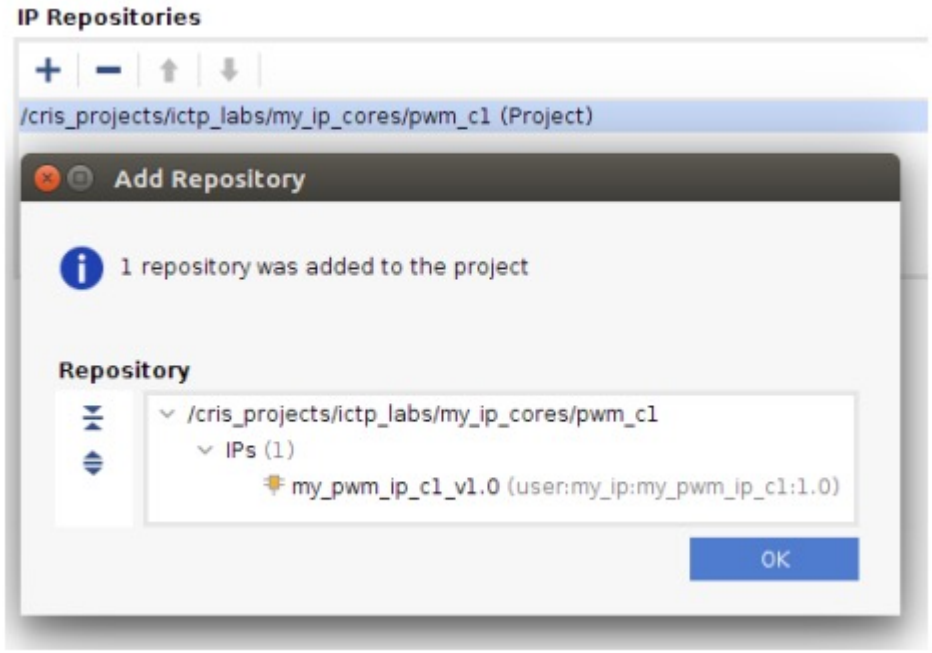


Summary

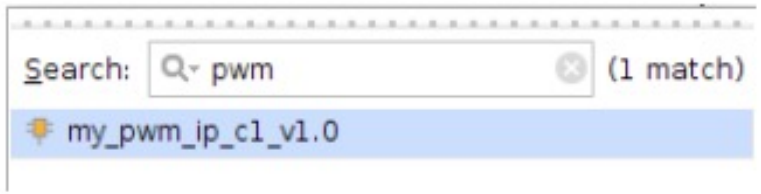
Create AXI4-Lite IP
Add IP to Catalog
Edit IP HDL Code
Update IP

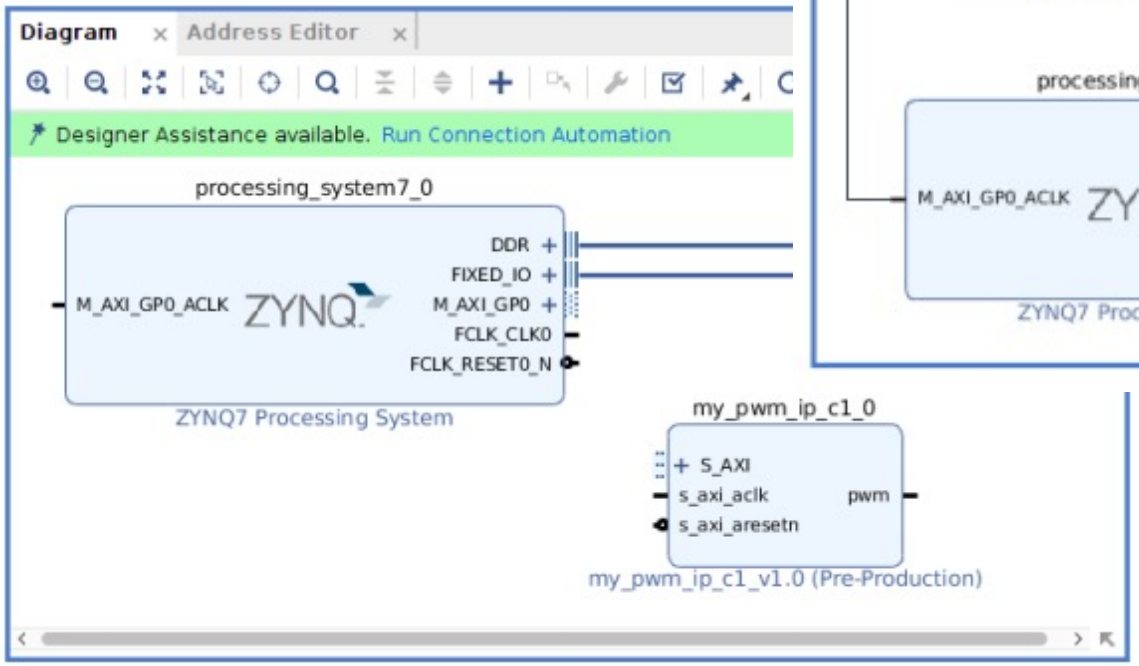
Using My IP in Vivado

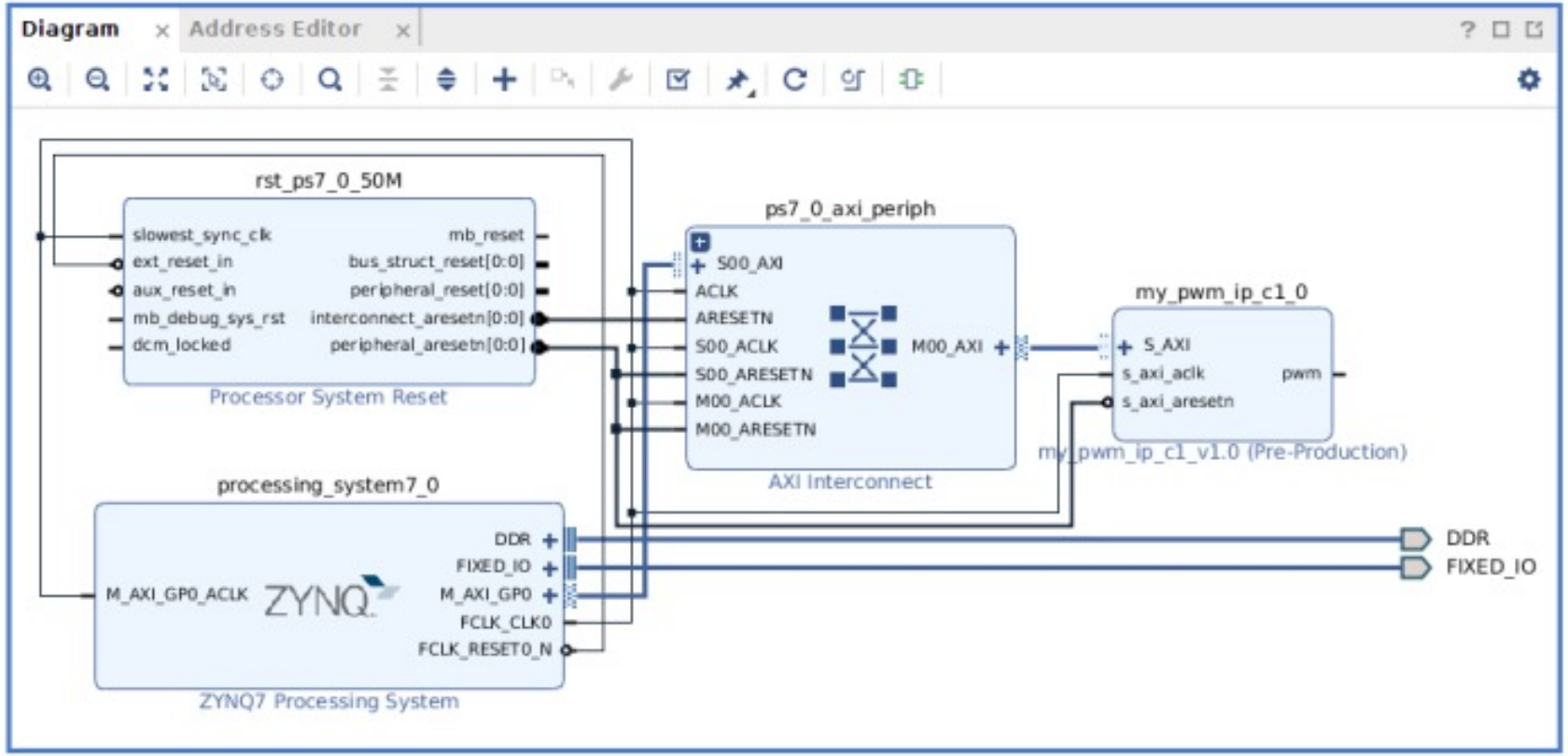




Using My IP in Vivado







Constraint File – LED Pinout

```
# -------- Constraint File for the Custom IP Lab -------- #
# ---- the pwm_0 output will be associated with LED0 ---- #
set_property PACKAGE_PIN T22 [get_ports pwm_0]
set_property IOSTANDARD LVCMOS33 [get_ports pwm_0]
# EOF #
```

Using My IP in SDK

lab_custom_ip.sdk - C/C++ - lab_custom_ip_wrapper_hw_platform_0/drivers/my_pwm_ip_c1_v1_0/src/my_

Project Explorer

- lab_custom_ip_wrapper_hw_plat
  - drivers
    - my_pwm_ip_c1_v1_0
      - data
      - src
        - Makefile
        - my_pwm_ip_c1_selftest.c
        - my_pwm_ip_c1.c
        - my_pwm_ip_c1.h
  - lab_custom_ip_wrapper.bit
  - ps7_init_gpl.c
  - ps7_init_gpl.h
  - ps7_init.c
  - ps7_init.h
  - ps7_init.html
  - ps7_init.tcl
  - system.hdf

system.hdf | my_pwm_ip_c1.c | my_pwm_ip_c1.h

```
#include "xstatus.h"

#define MY_PWM_IP_C1_S_AXI_SLV_REG0_OFFSET 0
#define MY_PWM_IP_C1_S_AXI_SLV_REG1_OFFSET 4
#define MY_PWM_IP_C1_S_AXI_SLV_REG2_OFFSET 8
#define MY_PWM_IP_C1_S_AXI_SLV_REG3_OFFSET 12


/************************** Type Definitions ***************
/**
 *
 * Write a value to a MY_PWM_IP_C1 register. A 32 bit write is
 * If the component is implemented in a smaller width, only the
 * significant data is written.
 *
 * @param   BaseAddress is the base address of the MY_PWM_IP_C1
 * @param   RegOffset is the register offset from the base to w
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *  void MY_PWM_IP_C1_mWriteReg(u32 BaseAddress, unsigned RegOf
 *
 */
#define MY_PWM_IP_C1_mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
```

Using My IP in SDK

lab_custom_ip_c2.sdk - C/C++ - lab_custom_ip_c2/src/lab_custom_ip_c2.c - Xilinx SDK

Project Explorer

- lab_custom_ip_c2
  - Binaries
  - Includes
  - Debug
  - src
    - lab_custom_ip_c2.c
    - platform_config.h
    - platform.c
    - platform.h

lab_custom_ip_c2.c

```
//   Advanced Workshop on FPGA-based Systems-on-Chip for

#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "my_pwm_ip_c2.h"
#include "xil_io.h"

//----------------------------------------------------------
int main()
{
    int i;
    int k=0x7FFF;

     printf("-- Start of the Program --\r\n");
```

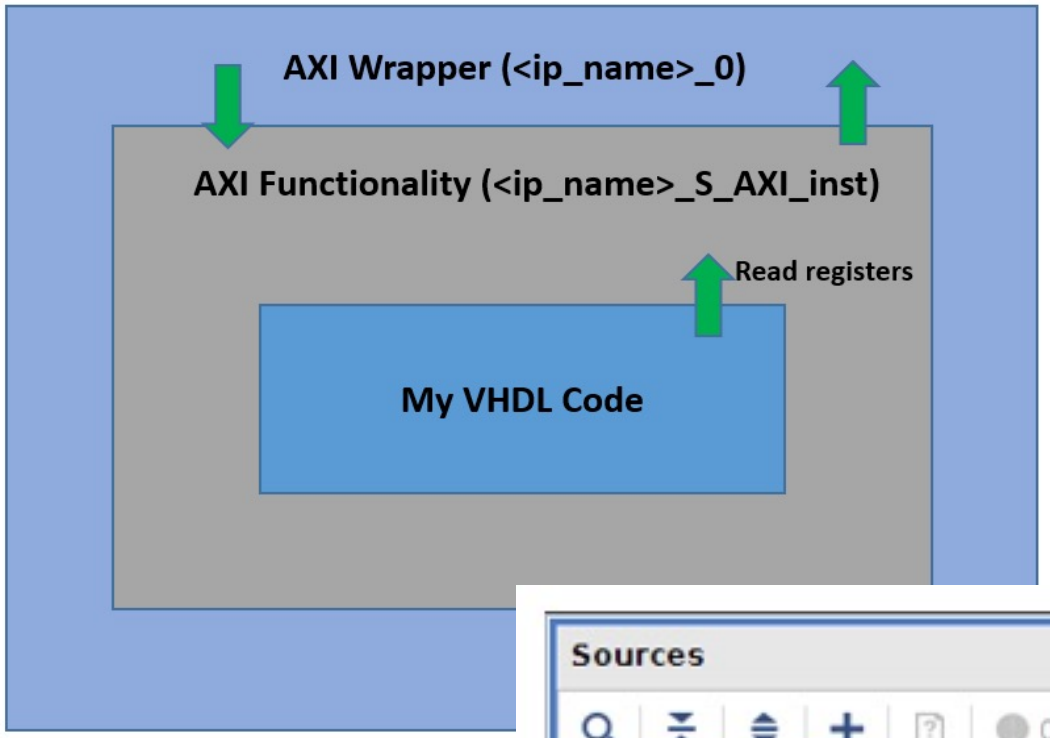
My IP – Case 2

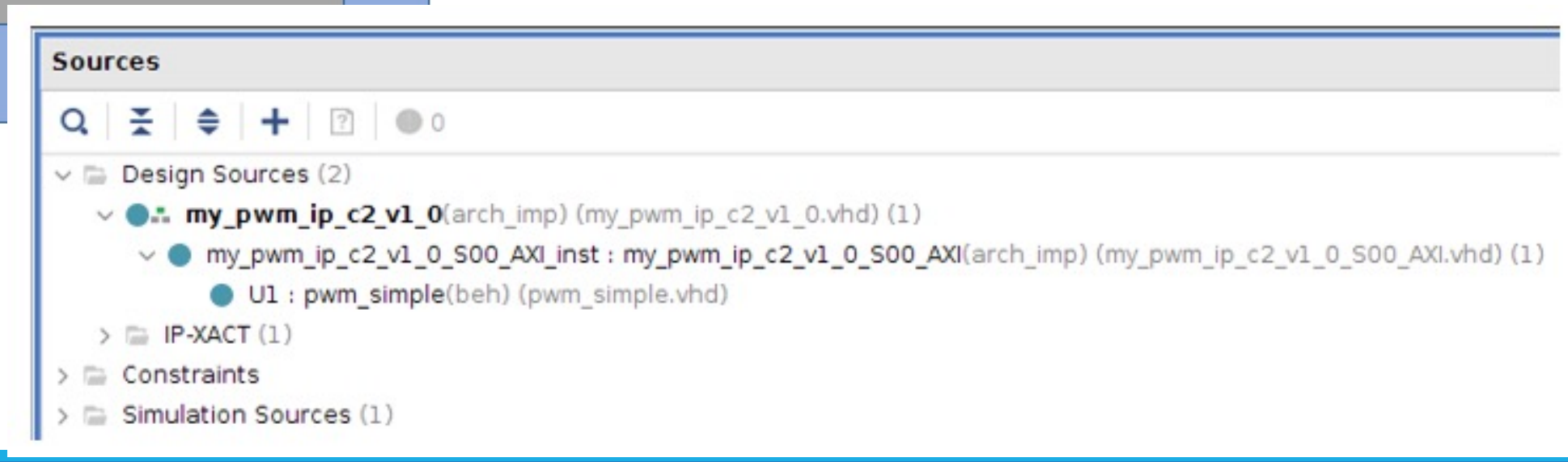
This is the VHDL instantiation statement that is necessary to write in the my*_pwm_ip_c1_0_S_AXI_inst.vhd* file

```
U1: entity work.pwm_simple   -- pwm_simple component instantiation
  generic map (
    dc_bits => dc_bits)
  port map(
    S_AXI_ACLK      => S_AXI_ACLK,
    S_AXI_ARESETN   => S_AXI_ARESETN,
    duty_cycle      => slv_reg0,
    pwm             => pwm
    );
```

```
entity pwm_simple is

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

My IP – Case 2


AXI Wrapper (<ip_name>_0)
AXI Functionality (<ip_name>_S_AXI_inst)
Read 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

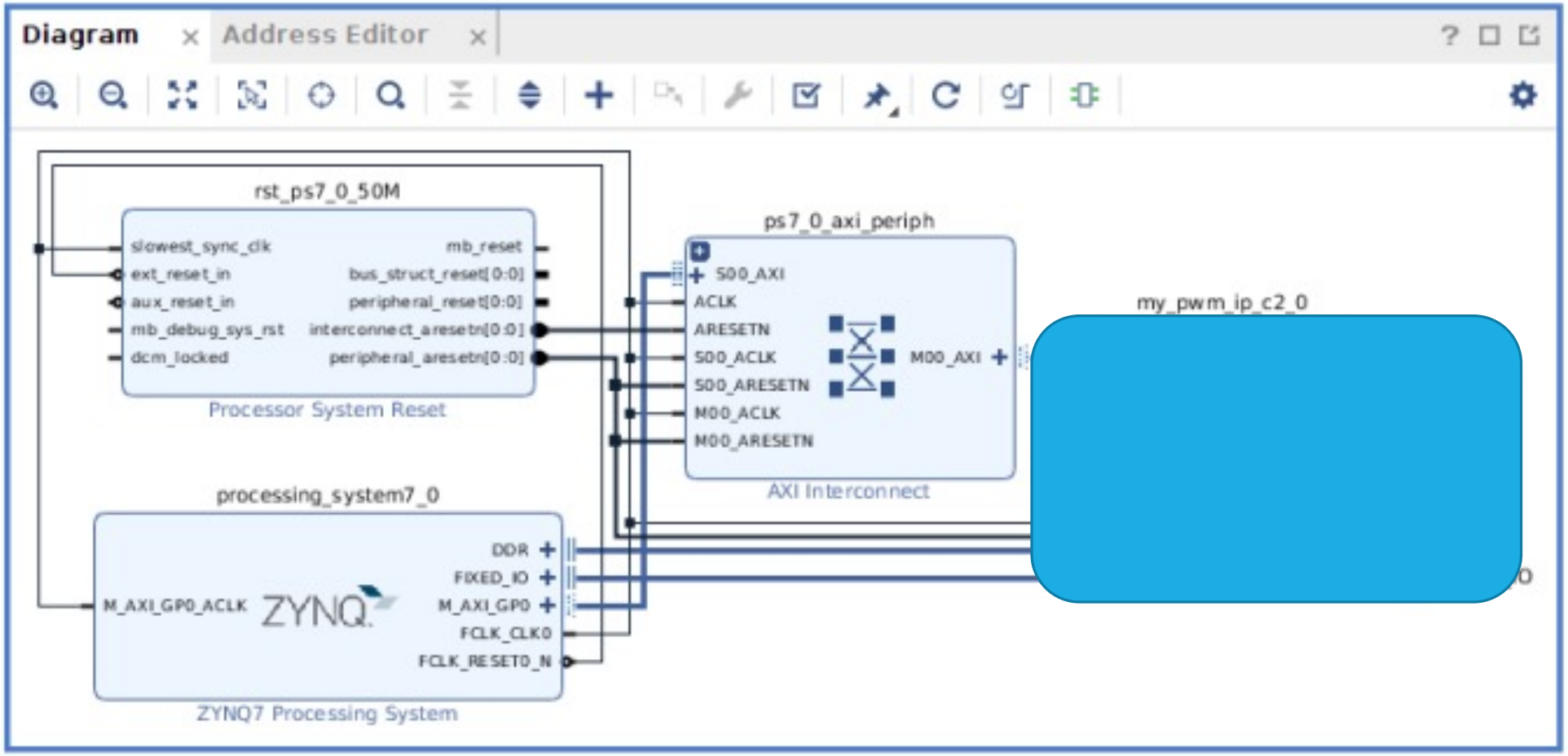


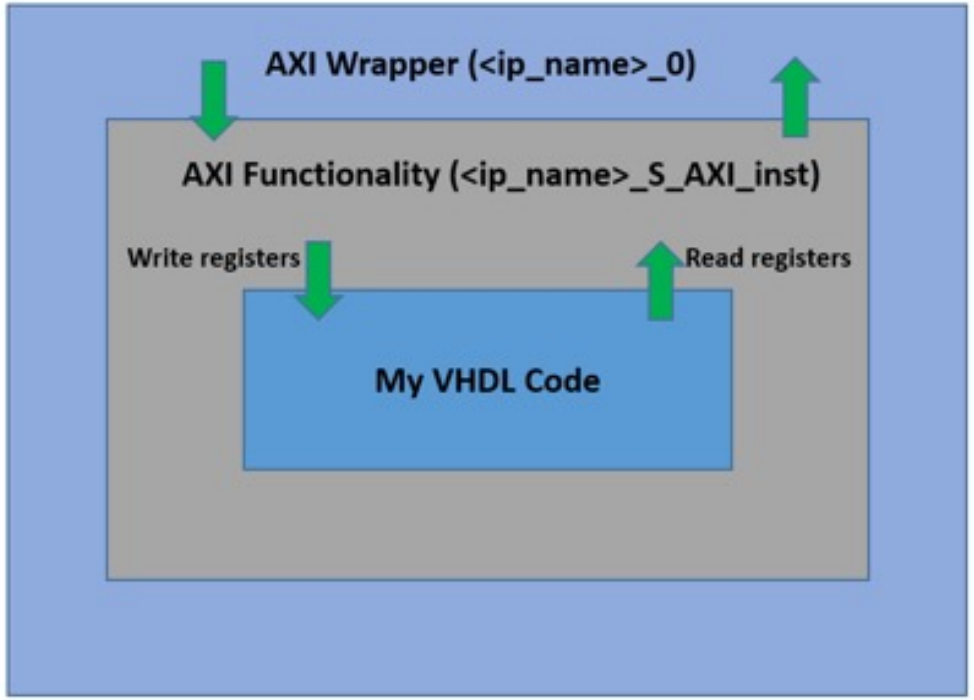
Diagram
Address Editor
rst_ps7_0_50M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
ps7_0_axi_periph
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M00_AXI
AXI Interconnect
my_pwm_ip_c2_0
processing_system7_0
M_AXI_GP0_ACLK
ZYNQ
DDR
FIXED_IO
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System

My IP – Case 3

```
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  --
```

My IP – Case 3

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



Lab Custom IP

Lab Custom IP

Basic PWM Functionality

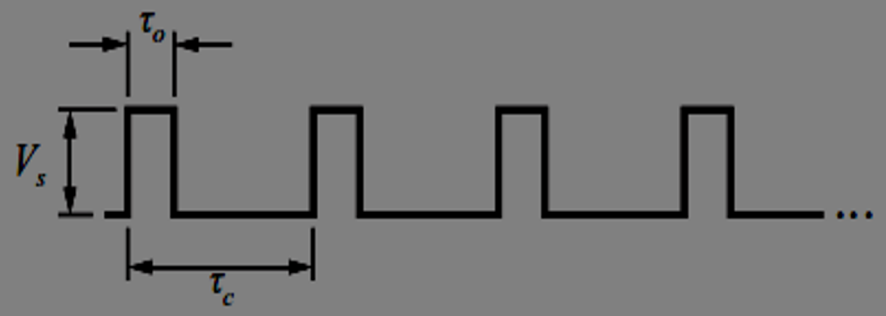


Figure 1: Nomenclature for definition of PWM duty cycle.

$$Veff = Vs\frac{\tau o}{\tau c} \quad (1)$$

VHDL Code for PWM Simple

```
8   entity pwm_simple is
9
10    generic (
11      dc_bits : integer := 16);            -- number of bits for the duty
12                                           -- cycle value
ext Bookmark ort (
14      -- clock & reset signals
15      S_AXI_ACLK        : in  std_logic;   -- AXI clock
16      S_AXI_ARESETN     : in  std_logic;   -- AXI reset, active low
17      -- control input signal
18      duty_cycle        : in  std_logic_vector(31 downto 0);
19      -- PWM output
20      pwm               : out std_logic    -- pwn output
21      );
22  end entity pwm_simple;
```

VHDL Code for PWM Simple

```
24  ------------------------------------------------------------------------
25  -- architecture
26  ------------------------------------------------------------------------
27  architecture beh of pwm_simple is
28  begin
29    pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
30       variable counter : unsigned(dc_bits-1 downto 0); -- count clocks tick
31    begin  -- process pwm_pr
32      if (S_AXI_ARESETN = '0') then
33        counter   := (others => '0');
34        pwm       <= '0';
35      elsif (rising_edge(S_AXI_ACLK)) then
36        counter := counter + 1;
37        if (counter < unsigned(duty_cycle(dc_bits-1 downto 0))) then
38          pwm <= '1';
39        else
40          pwm <= '0';
41        end if;
42      end if;
43    end process pwm_pr;
44  end architecture beh;
```

PWM IP Core - Case 1

```
-- Add user logic here
   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(slv_reg0(dc_bits-1 downto 0))) then
       pwm <= '1';
     else
       pwm <= '0';
     end if;
   end if;
 end process pwm_pr;
-- User logic ends
```

PWM IP Core - Case2

```
-- Add user logic here
 U1: entity work.pwm_simple    -- pwm_simple component instantiation
   generic map (
     dc_bits => dc_bits)
   port map(
     S_AXI_ACLK      => S_AXI_ACLK,
     S_AXI_ARESETN   => S_AXI_ARESETN,
     duty_cycle      =>          ,
     pwm             => pwm);
-- User logic ends
```

VHDL code for PWM Complete (1)

```
-------------------------------------------------------------------------------
-- Description: generation of the PWM signal using Rd/Wr registers that
-- will be Wr/Rd through the AXI bus.
--
-- The following register are defined for this PWM IP:
--
-- ----------------       Register 0: Control Register   ----------------------
-- bit31 | ... |   bit 4   |   bit 3   |   bit2   |   bit 1   |   bit 0   |
--                clear        Enable    invert PWM   enable    sw reset_n
--              interrupt    interrupt   output      disable
--
-- ----------------       Register 1: Status Register    ----------------------
-- bit31 | ... | ... ... ... ... ... ... ... ...   |  bit 1   |   bit 0   |
--                                                  interrupt   PWM output
--                                                    request    value
--
-- ----------------------       Register 2     --------------------------------
-- Writable register: ARM will write into this register the PWM (duty cycle) value
--
-- ----------------------       Register 3     --------------------------------
-- Readable register: hold the current version of the PWM IP module
--
-- ----------------------       Register 4     --------------------------------
-- Readable register: copy of Register 2, that can be read by the ARM
--
--  ----------------------       Outputs     -------------------------------
-- pwm: which is the PWM value, '0' or '1'
-- int_pwm: which generate an int request (goes to '1') on the falling edge
-- of the pwm ouptut.
-------------------------------------------------------------------------------
--
```

VHDL code for PWM Complete (2)

```
-------------------------------------------------------------------------------
-- 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;
```

VHDL code for PWM Complete (3)

```
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

  -- 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
```

VHDL code for PWM Complete (4)

```
-- 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;

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;
```

VHDL code for PWM Complete (5)

```
-- 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
```

```
-----------------------------------------------------------------------
-- 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;
```

VHDL code for PWM Complete (3)

```
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

  -- 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
```

PWM IP Core – Case 3

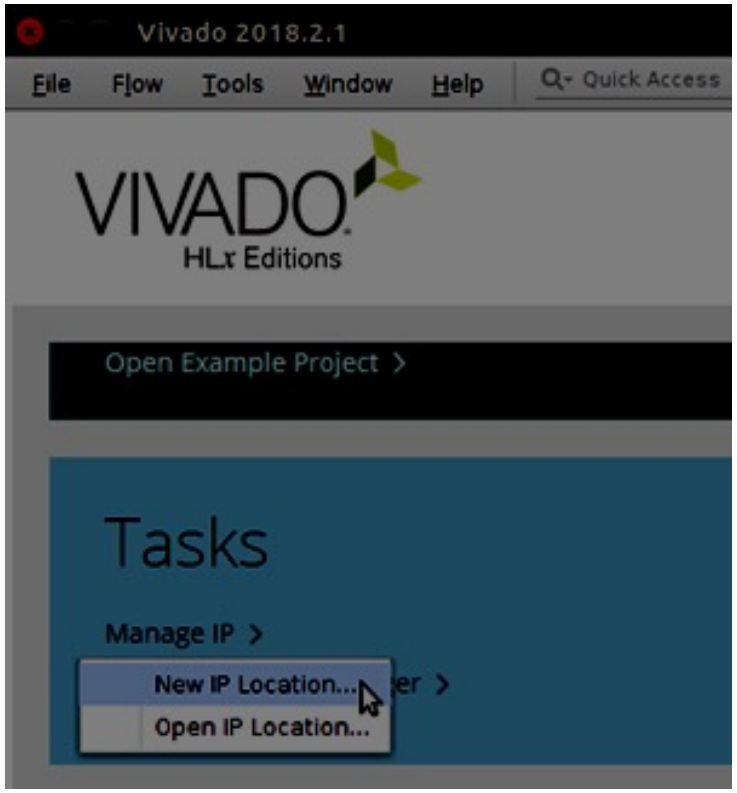
```vhdl
U1: entity work.pwm_complete   -- pwm_complete component instantiation
  generic map (
    dc_bits => dc_bits);
  port map(
    S_AXI_ACLK      => S_AXI_ACLK,
    S_AXI_ARESETN   => S_AXI_ARESETN,
      ???           => ????
      ???           => ????
      . . .           . . .
    pwm             => pwm
    );
```

```vhdl
-------------------------------------------------------------------------------
-- 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;
```

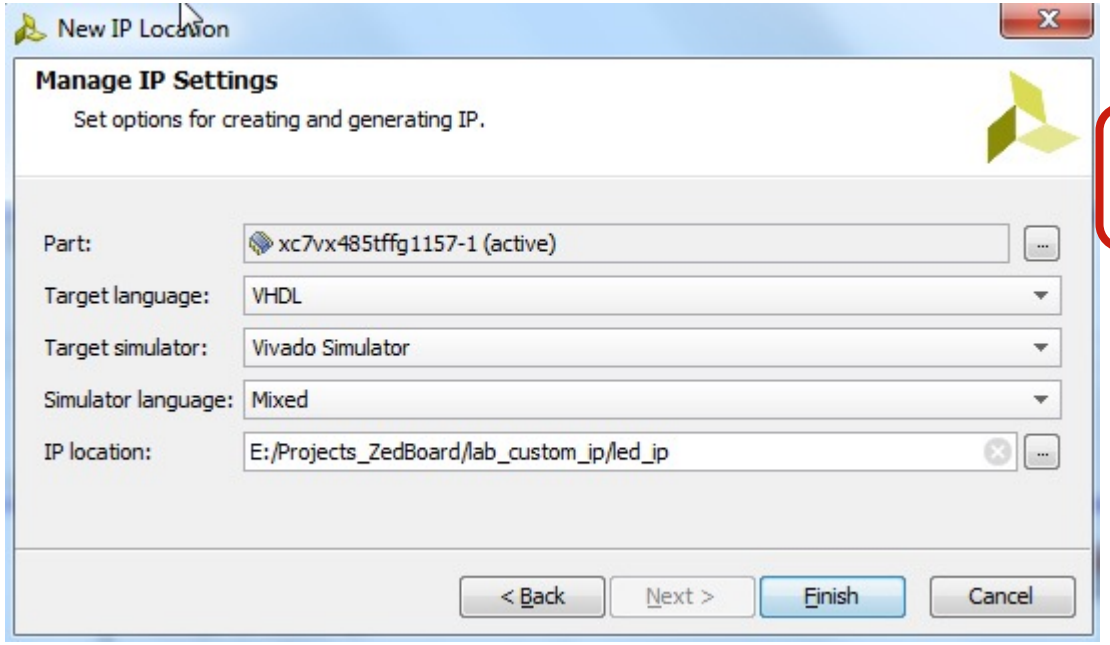
Apendix

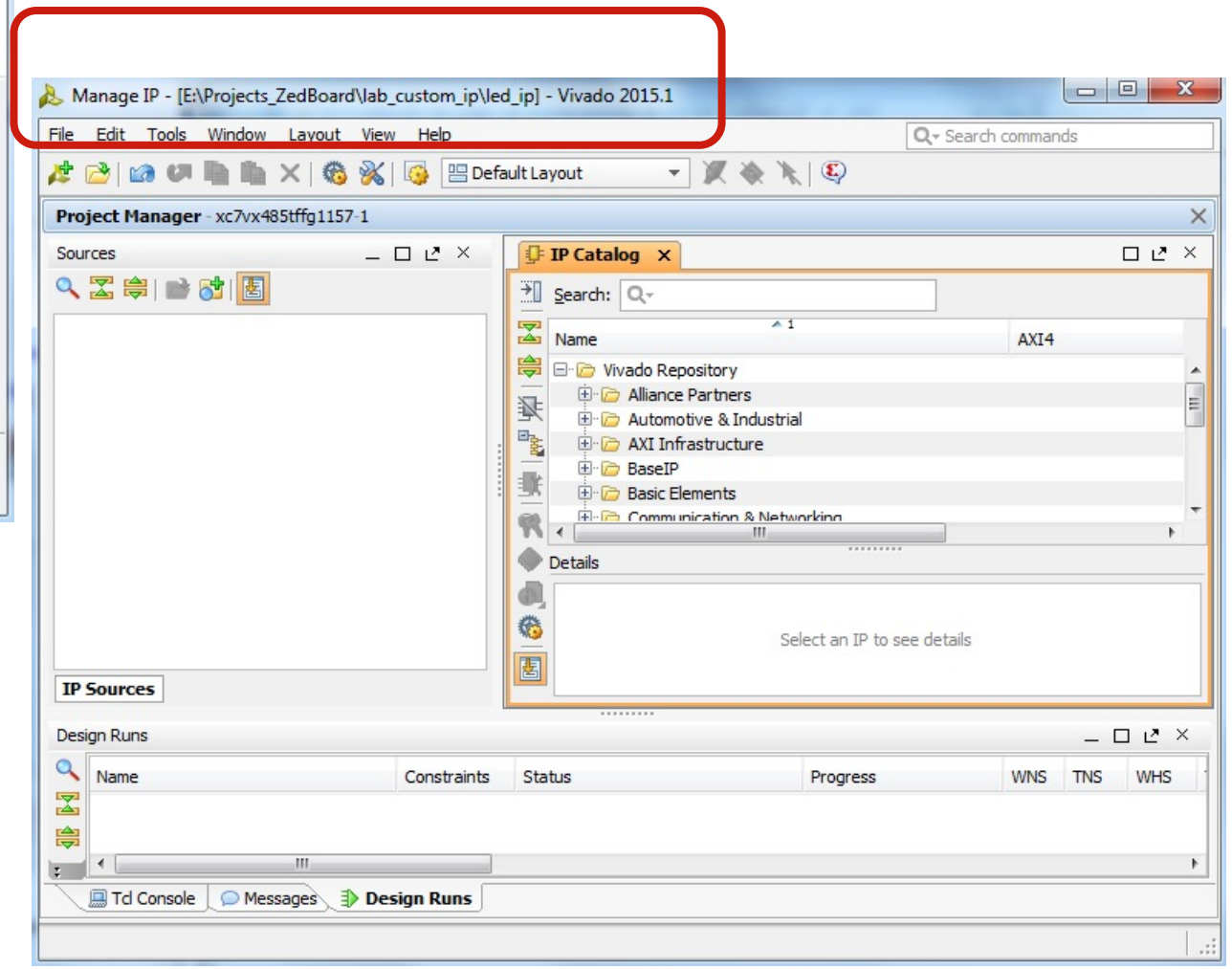
IP Manager

- **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

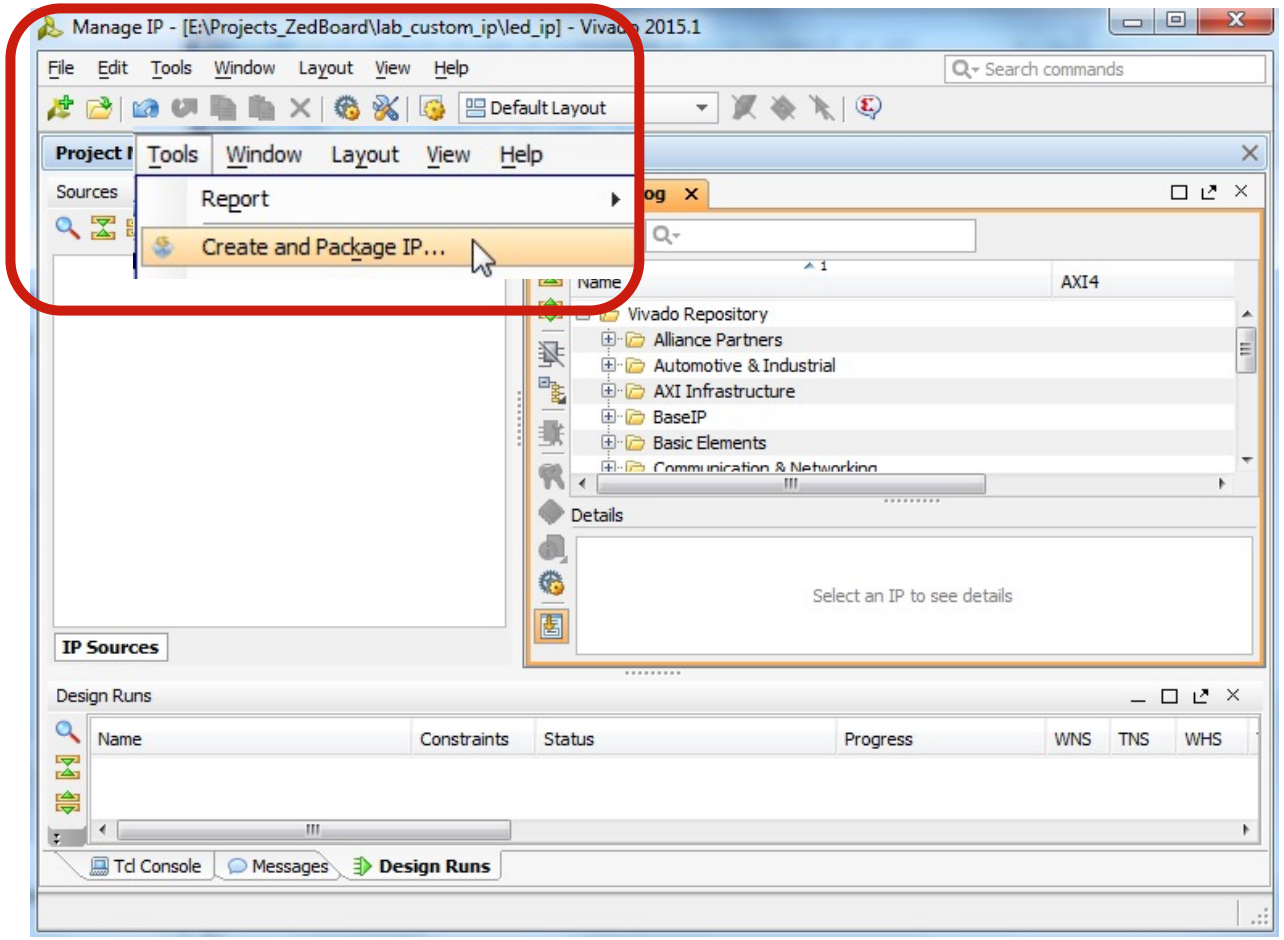


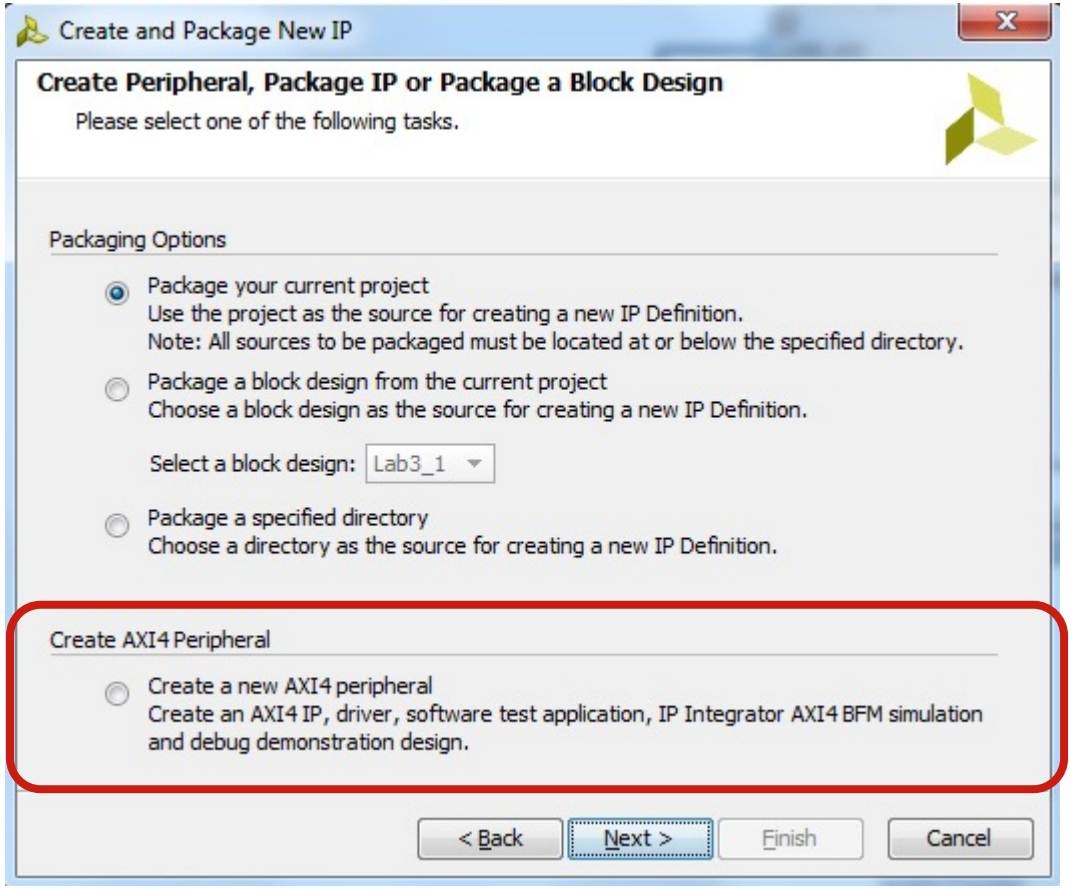
Create Custom AXI4 IP



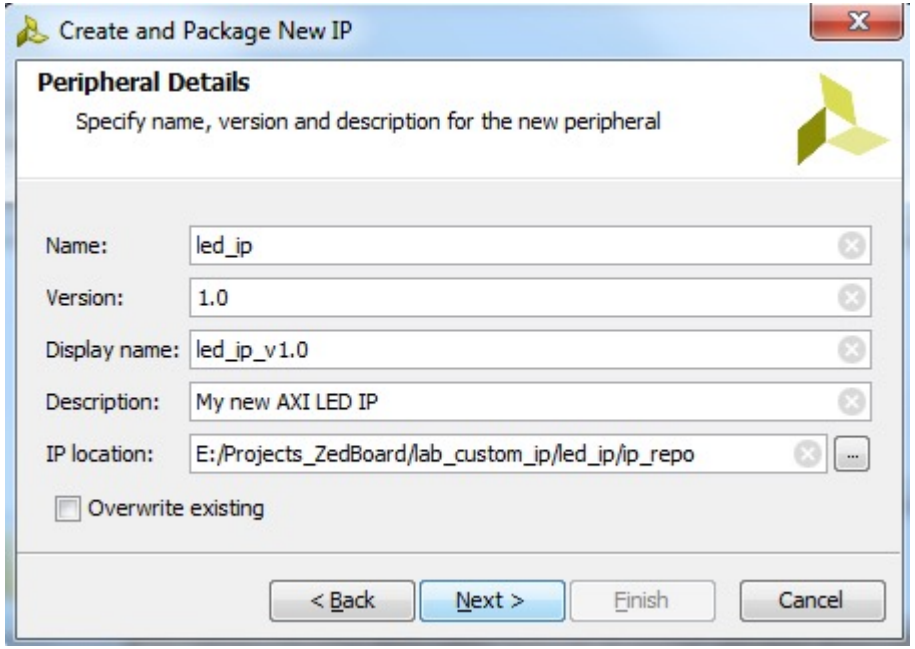


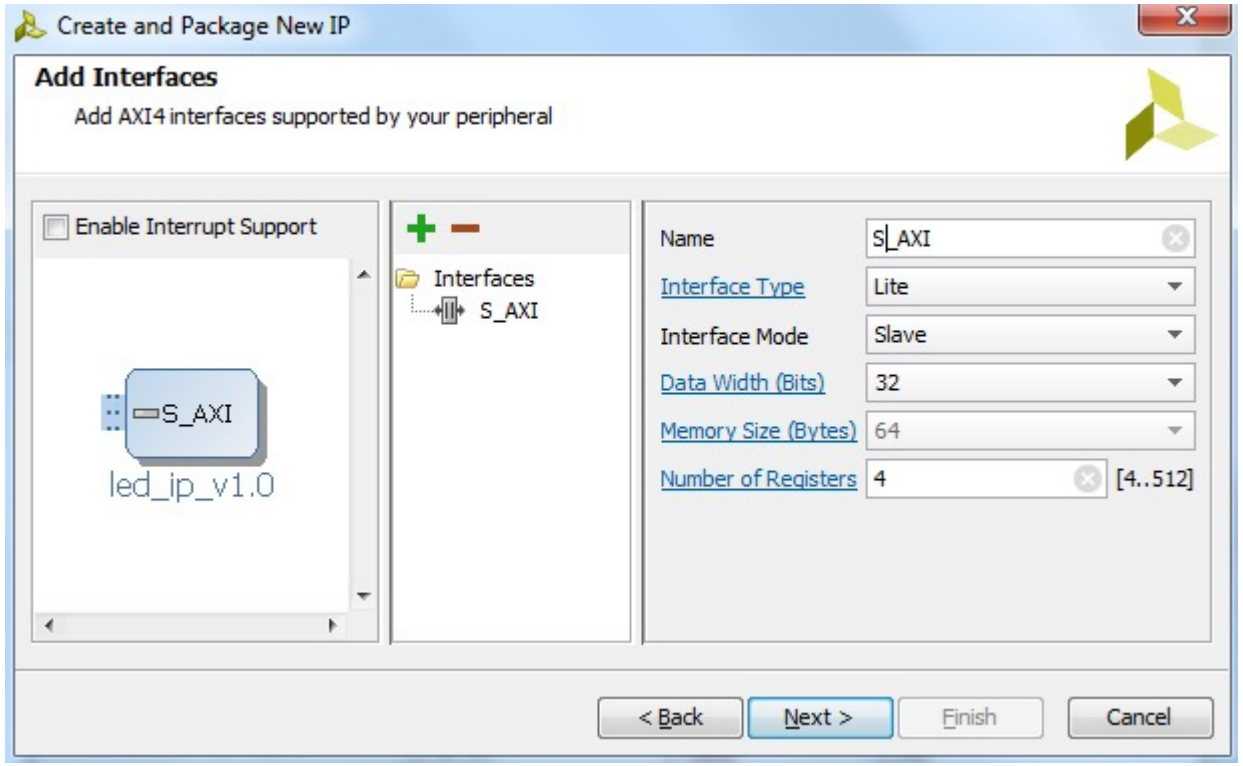
Create Custom AXI4 IP



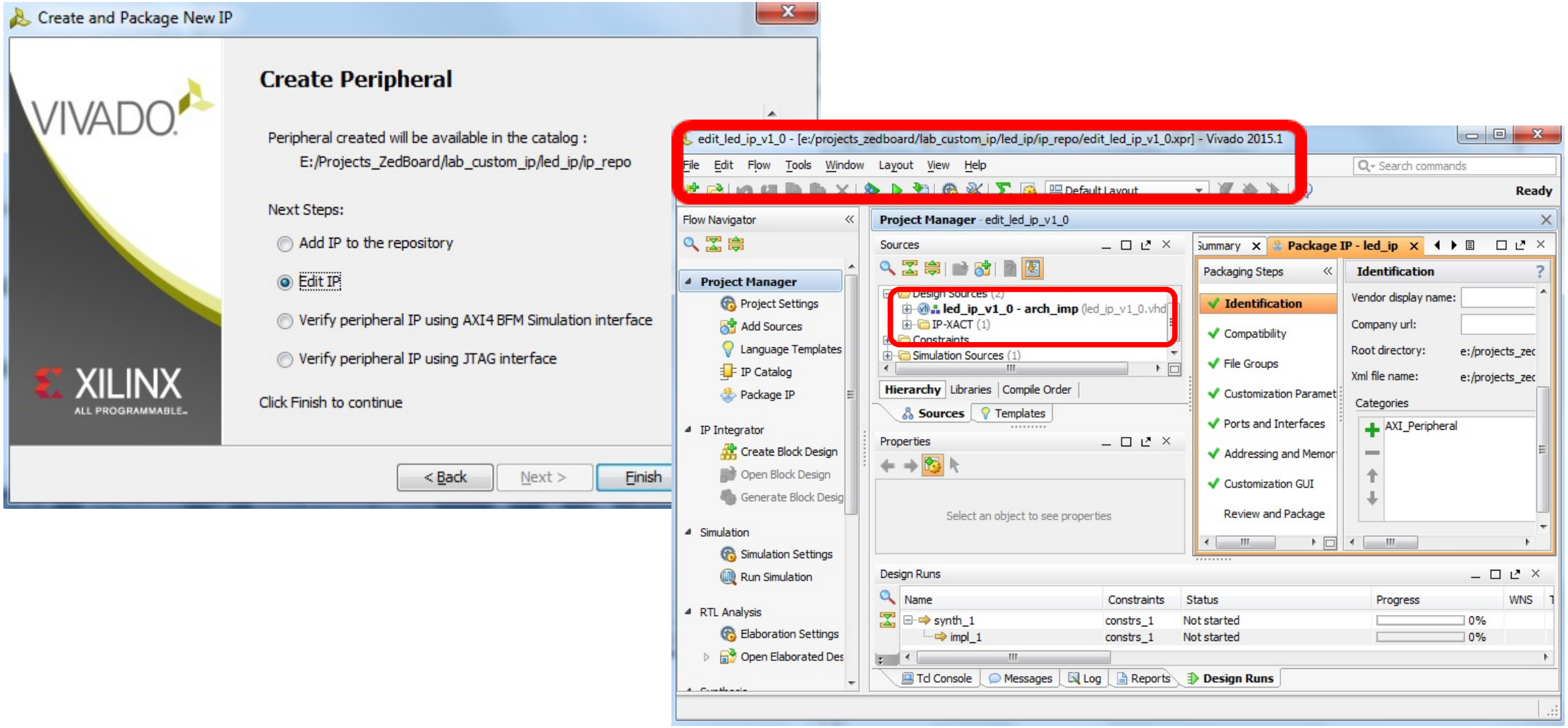


Create Custom AXI4 IP

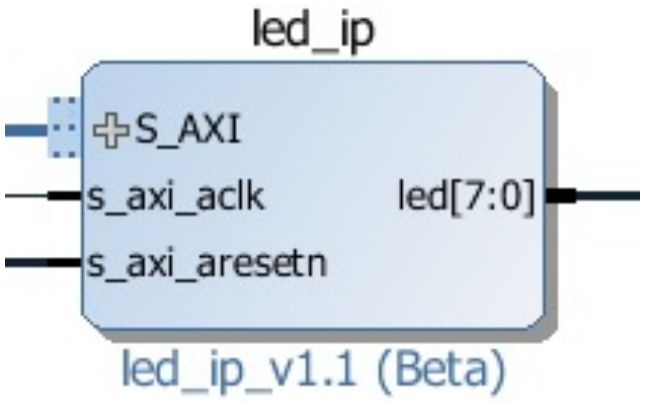




Edit Created Custom AXI4 IP



Edit Created Custom AXI4 IP



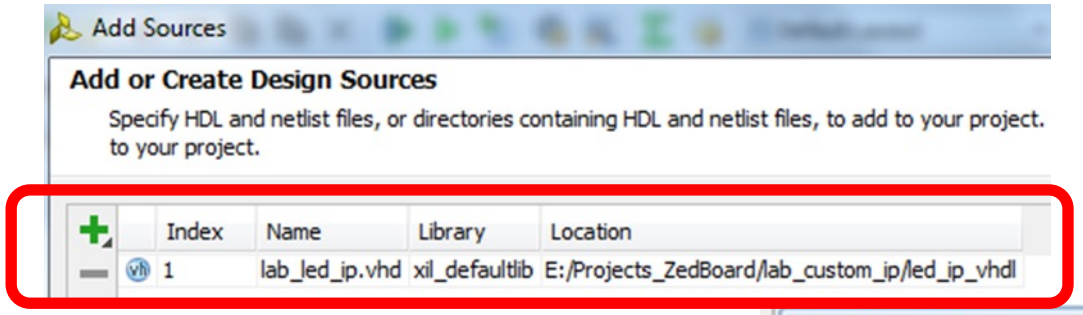
led_ip_v1_0.vhd | led_ip_v1_0_S_AXI.vhd | lab_le

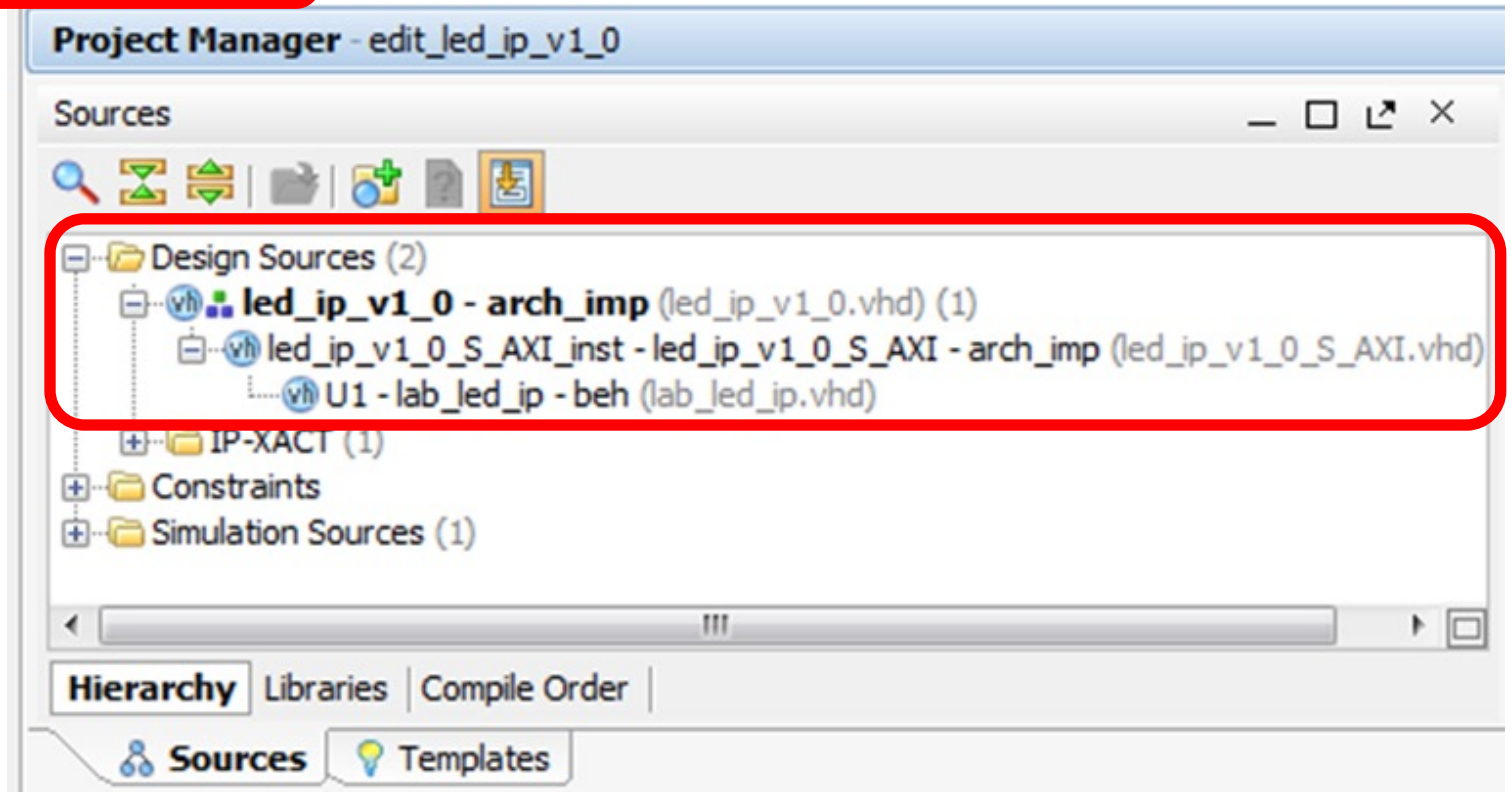
e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0/hdl/led_ip_v1_0.vhd

```
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
```

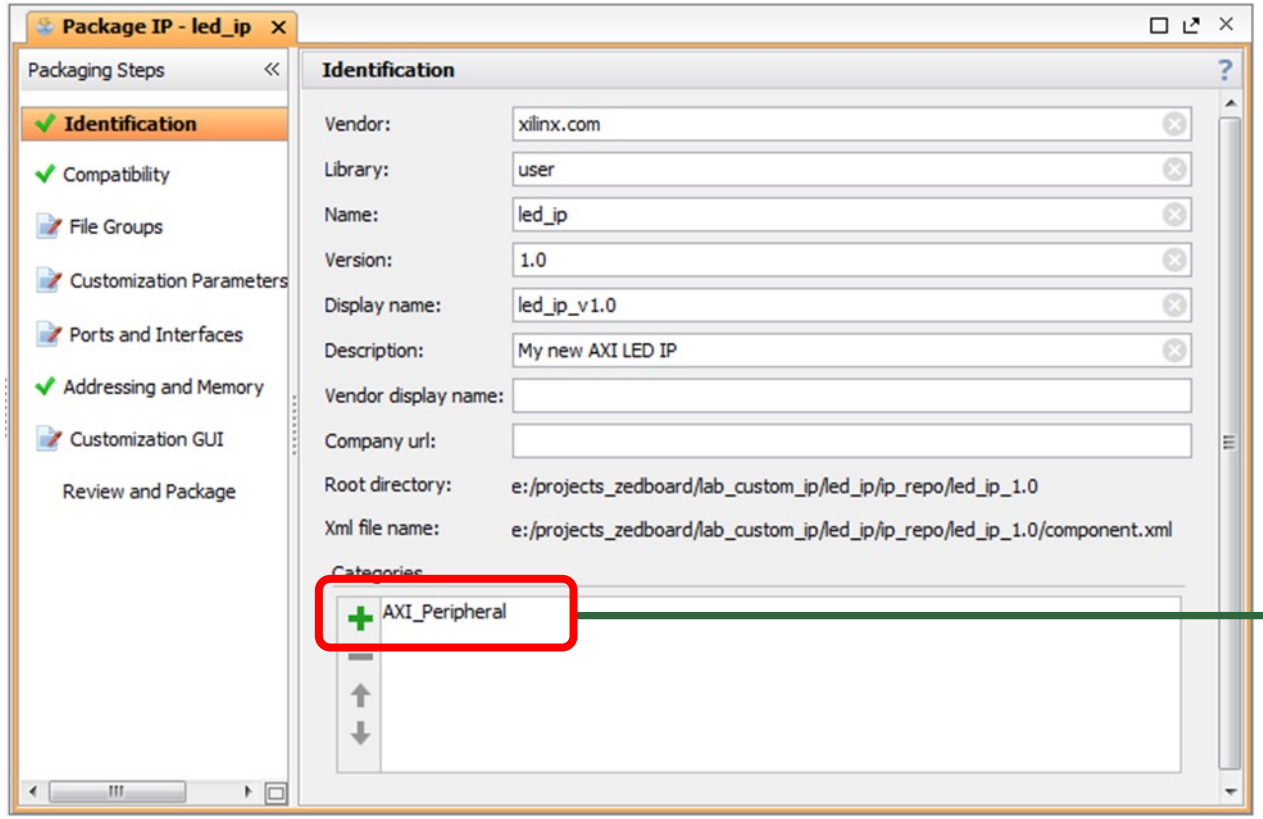
```
381     -- Add user logic here
382     U1: entity work.lab_led_ip generic map(led_width => led_width)
383                 port map(
384                     S_AXI_ACLK    => S_AXI_ACLK,
385                     SLV_REG_WREN  => SLV_REG_WREN,
386                     AXI_AWADDR    => AXI_AWADDR,
387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

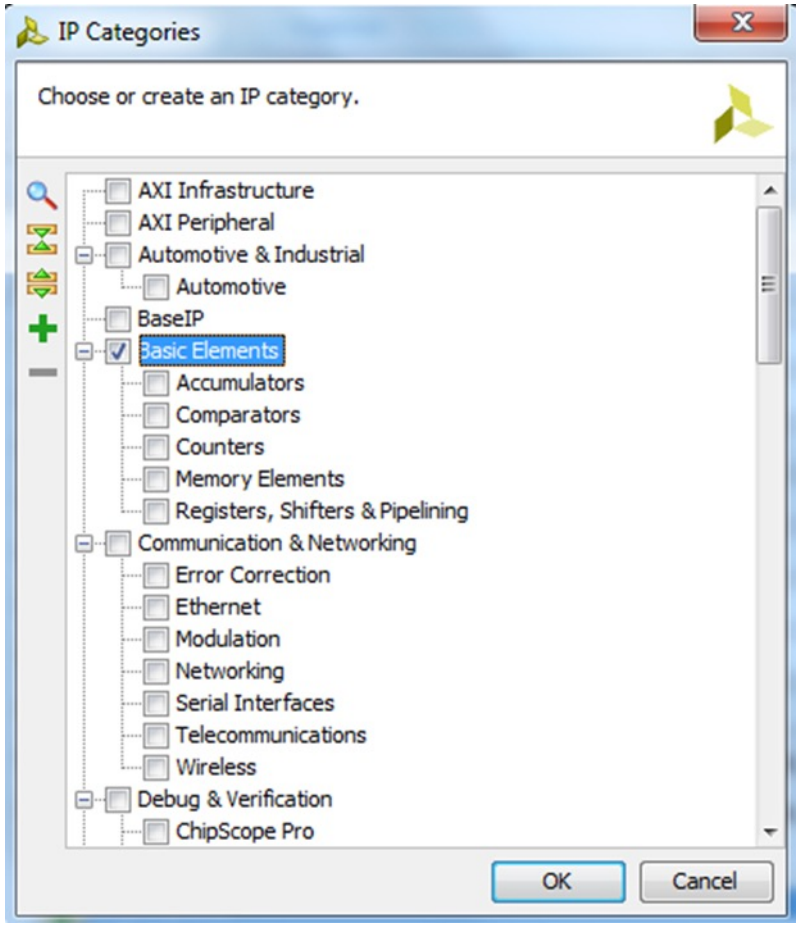
Hierarchy of My IP



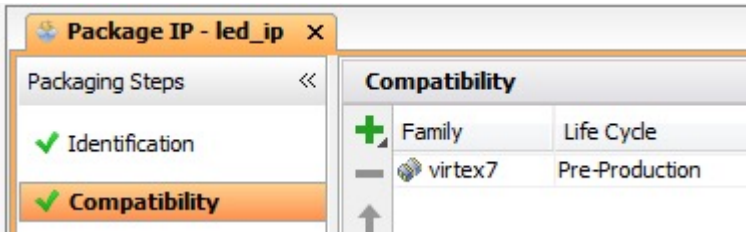


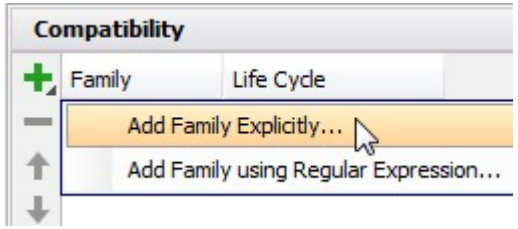
Package the IP

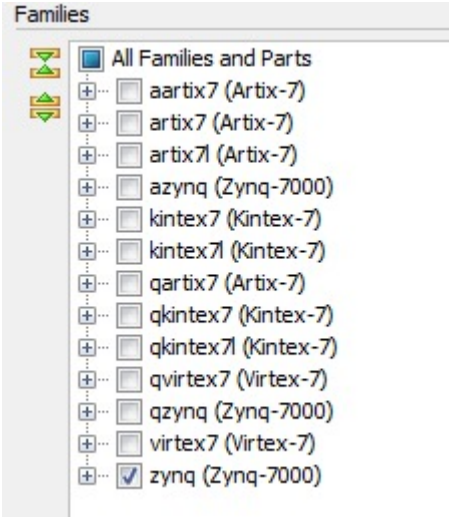




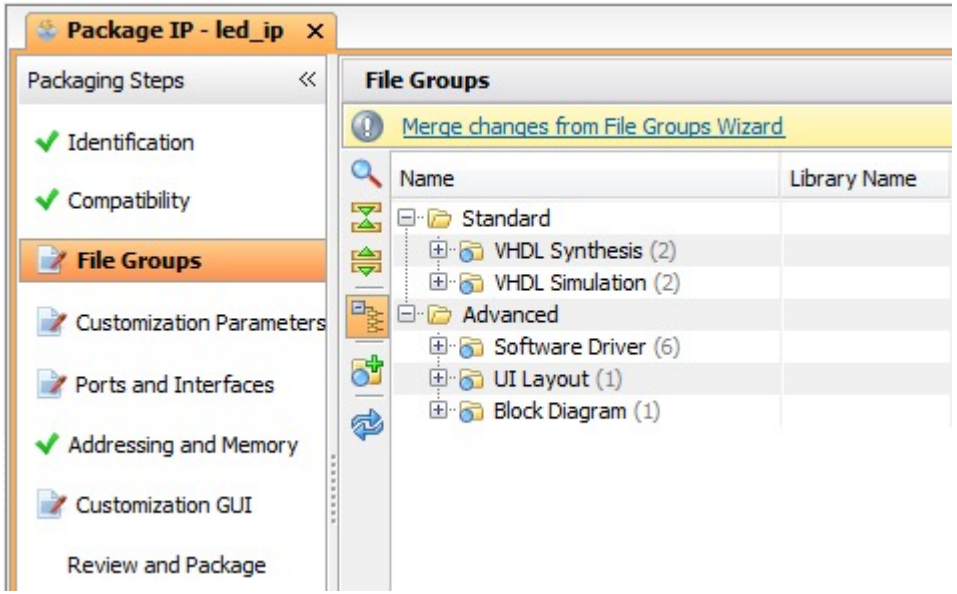
Compatibility of My IP



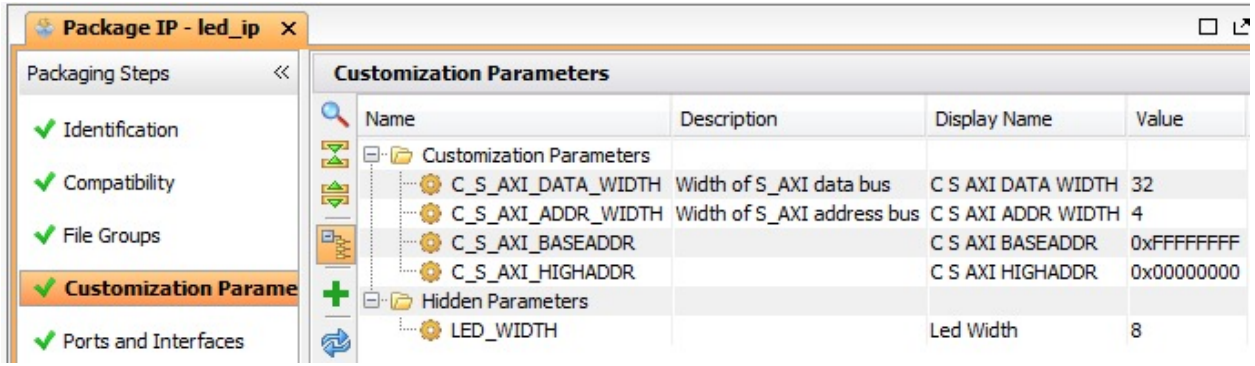


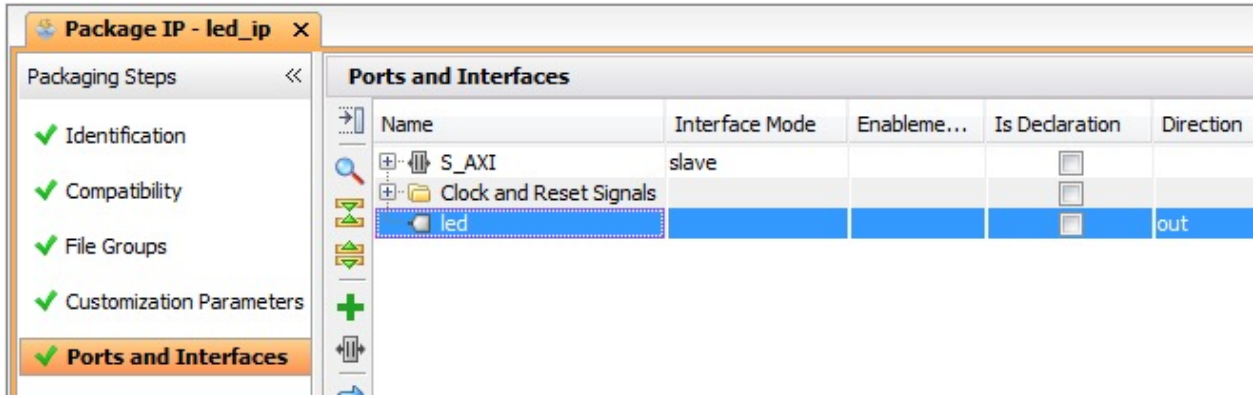


Updating Generated Files



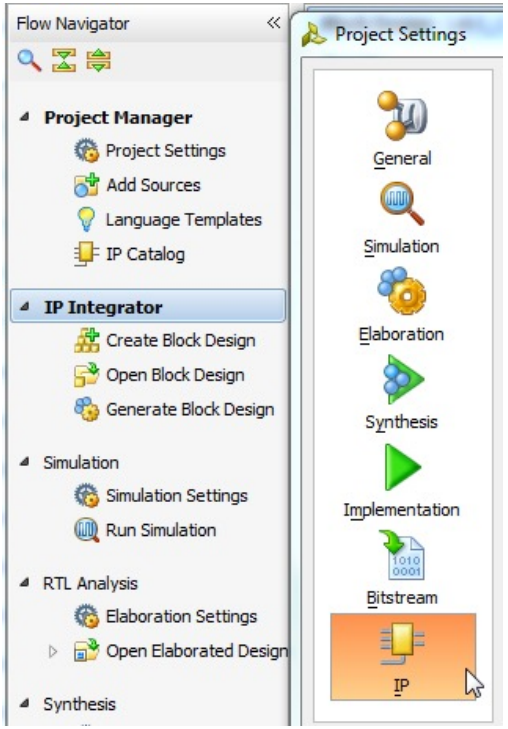
Checking Parameters and I/O Ports

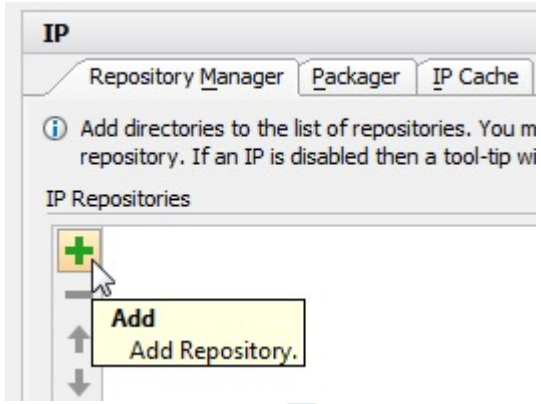


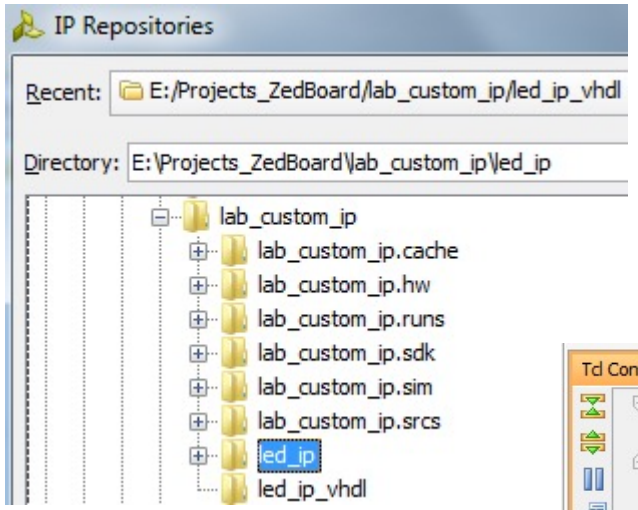


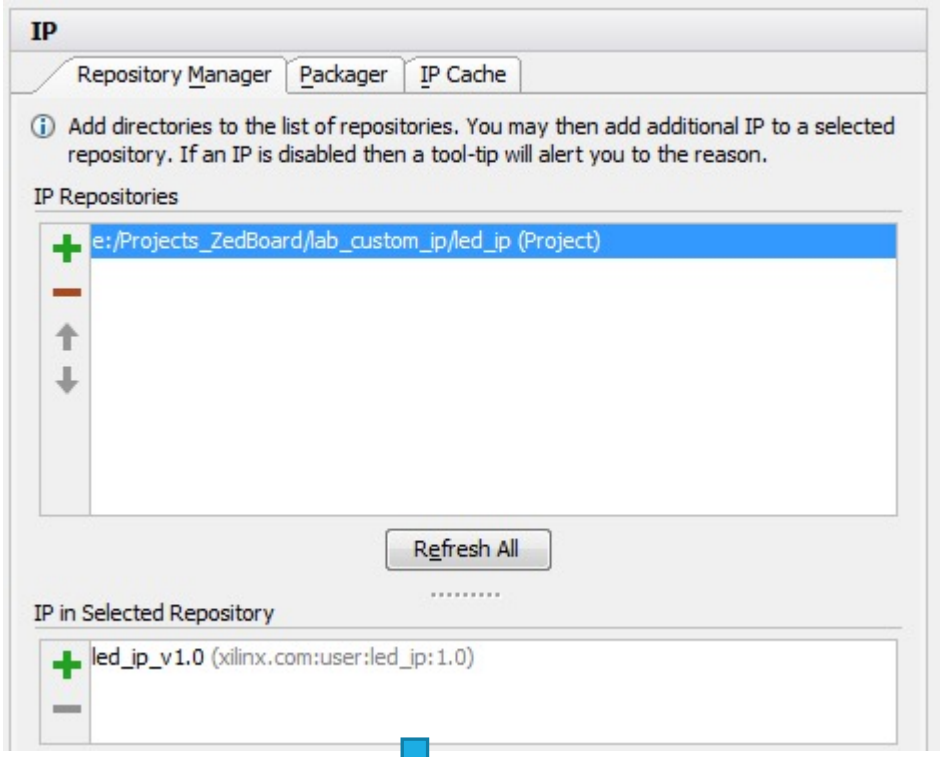
(This ends the Works on the edit_ip environment)

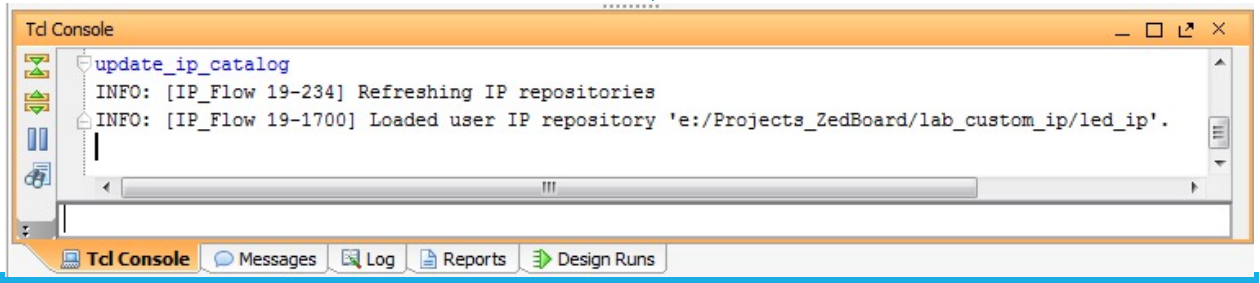
Add My IP to the Repository





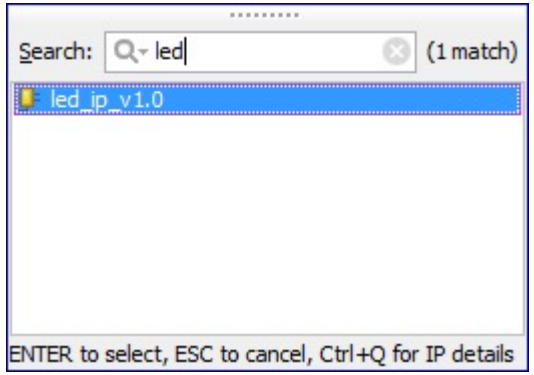


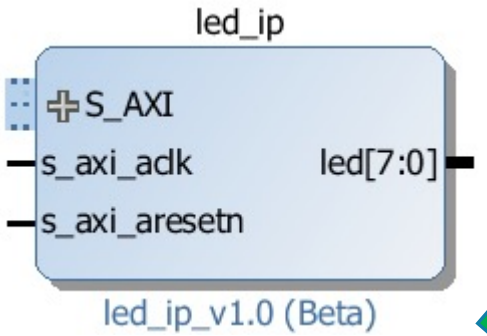


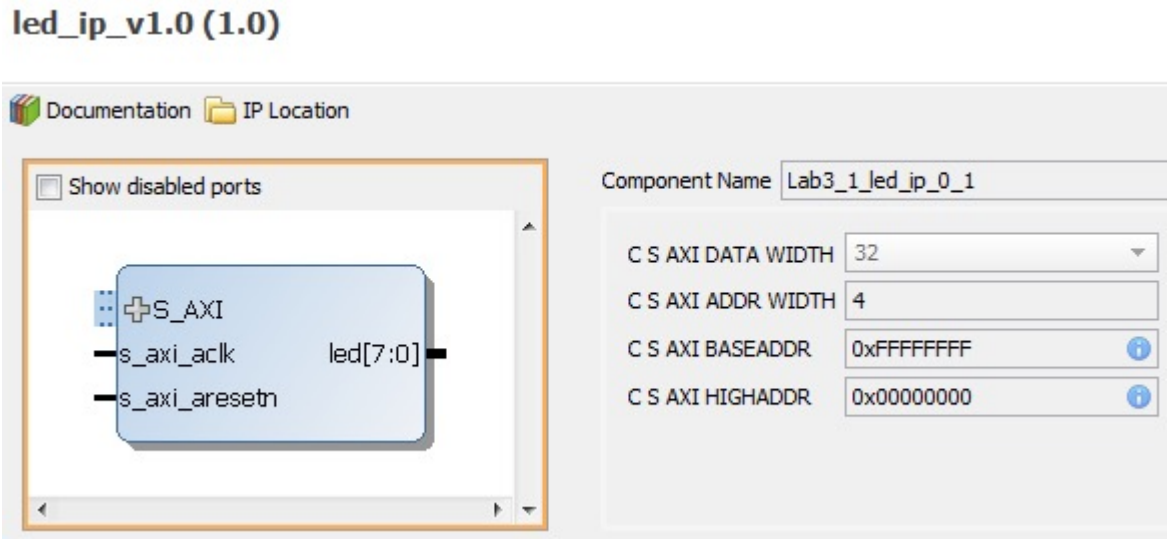


These steps shold be done in the Vivado Environment

led_ip Now Available in the IP List







Files created

D:\ictp_labs\lab_custom_ip\lab_custom_ip\lab_custom_ip.sdk\lab_custom_ip_wrapper_hw_platform_0\drivers\my_pwm_ip_c1_v1_0\src

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;
      }
```

Drivers: Write Register

D:\ictp_labs\lab_custom_ip\lab_custom_ip\lab_custom_ip.sdk\lab_custom_ip_wrapper_hw_platform_0\drivers\my_pwm_ip_c1_v1_0\src\my_pwm_ip_c1.h

```
/************************** Type Definitions **************************/
/**
 *
 * Write a value to a MY_PWM_IP_C1 register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param   BaseAddress is the base address of the MY_PWM_IP_C1device.
 * @param   RegOffset is the register offset from the base to write to.
 * @param   Data is the data written to the register.
 *
 * @return  None.
 *
 * @note
 * C-style signature:
 *  void MY_PWM_IP_C1_mWriteReg(u32 BaseAddress, unsigned RegOffset, u32 Data)
 *
 */
#define MY_PWM_IP_C1_mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))
```

Drivers: Read Register

D:\ictp_labs\lab_custom_ip\lab_custom_ip\lab_custom_ip.sdk\lab_custom_ip_wrapper_hw_platform_0\drivers\my_pwm_ip_c1_v1_0\src\my_pwm_ip_c1.h

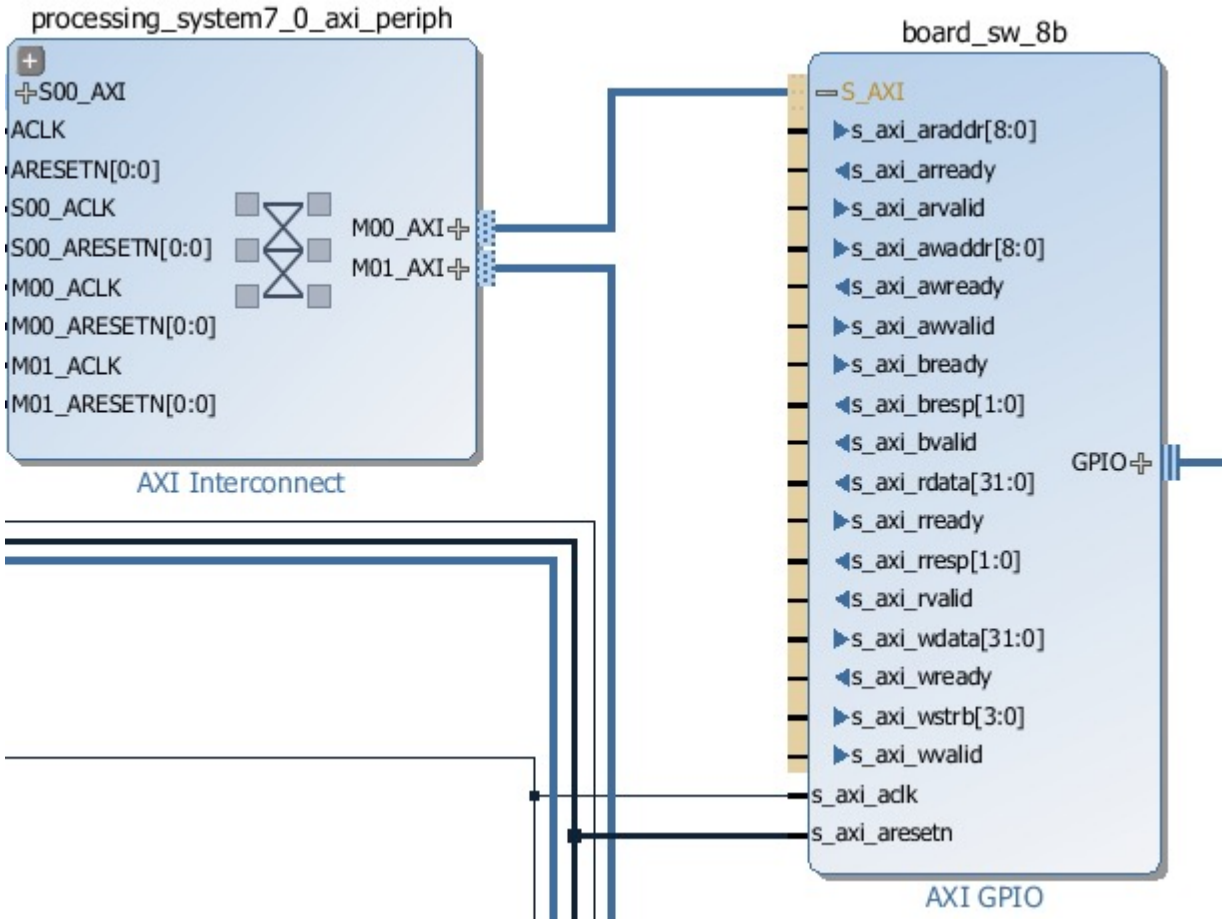
```
/**
 *
 * Read a value from a MY_PWM_IP_C1 register. A 32 bit read is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is read from the register. The most significant data
 * will be read as 0.
 *
 * @param   BaseAddress is the base address of the MY_PWM_IP_C1 device.
 * @param   RegOffset is the register offset from the base to write to.
 *
 * @return  Data is the data from the register.
 *
 * @note
 * C-style signature:
 *  u32 MY_PWM_IP_C1_mReadReg(u32 BaseAddress, unsigned RegOffset)
 *
 */
#define MY_PWM_IP_C1_mReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))
```

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

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;
```

Write Transaction

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;
```