Advanced Workshop on FPGA-based Systems-On-Chip for Scientific Instrumentation and Reconfigurable Computing

ICTP

Introduction to AXI – Custom IP

Cristian Sisterna

Universidad Nacional San Juan

Argentina

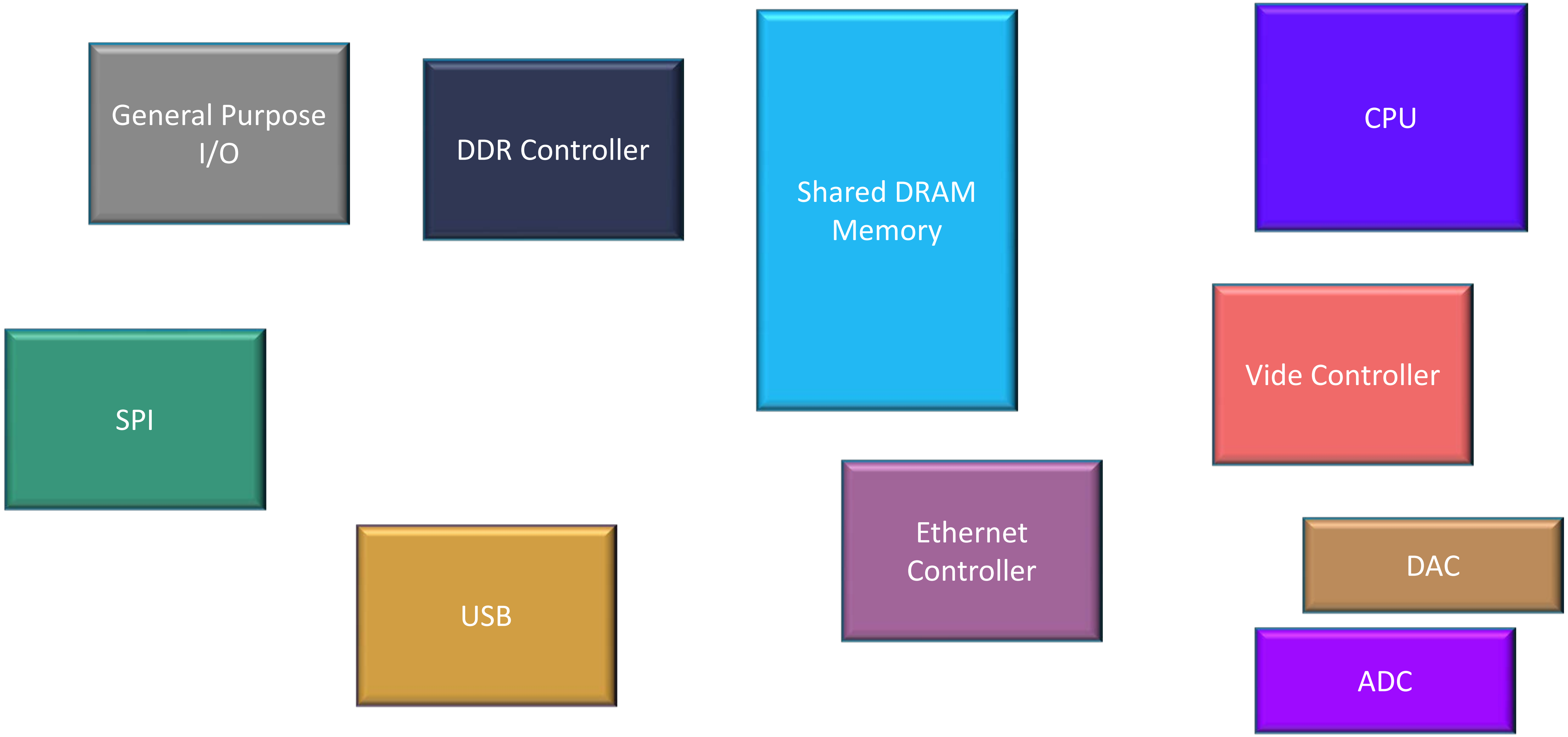
Agenda

- 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

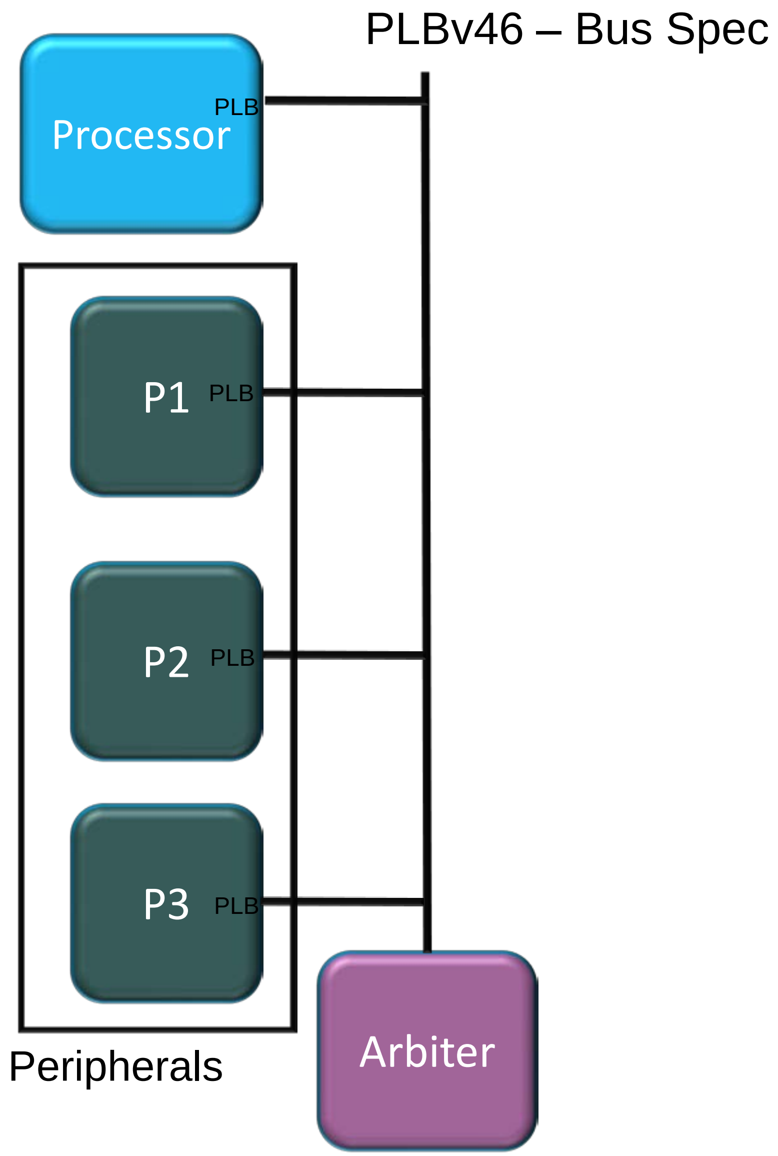
Need to Understand Device's Connectivity

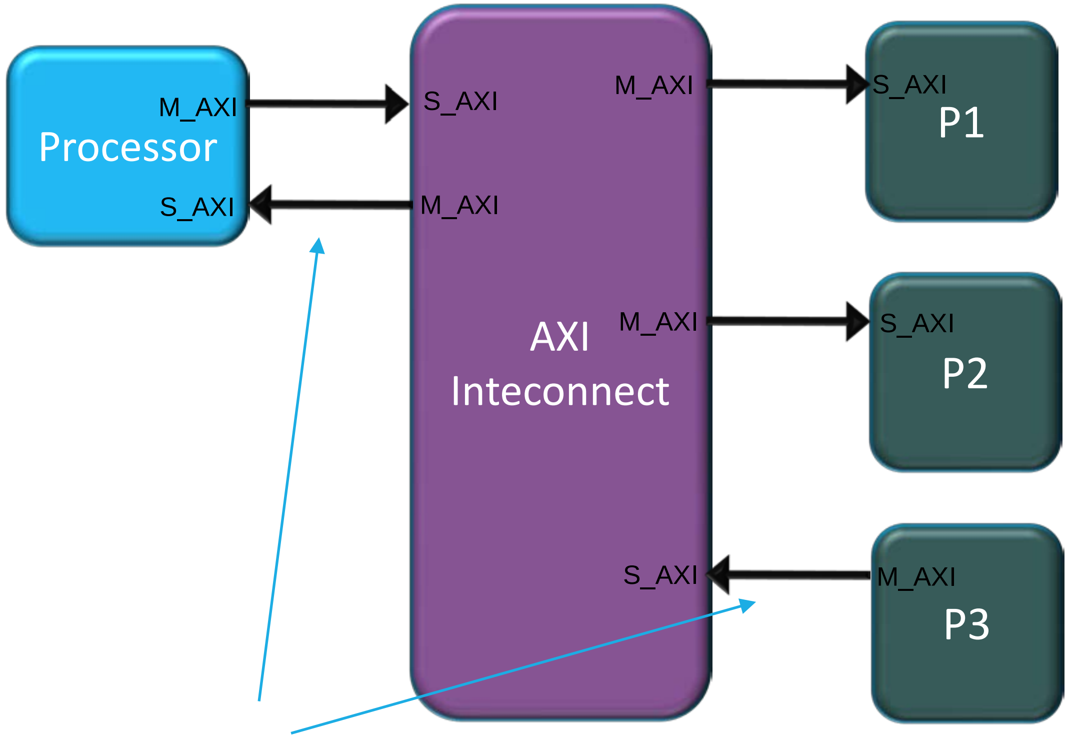
- There is a need to get familiar with the way that different devices communicate each other in an Embedded System like a Zynq based system
- Learning and understanding the communication among devices will facilitate the design of Zynq based systems
- All the devices in a Zynq system communicate each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

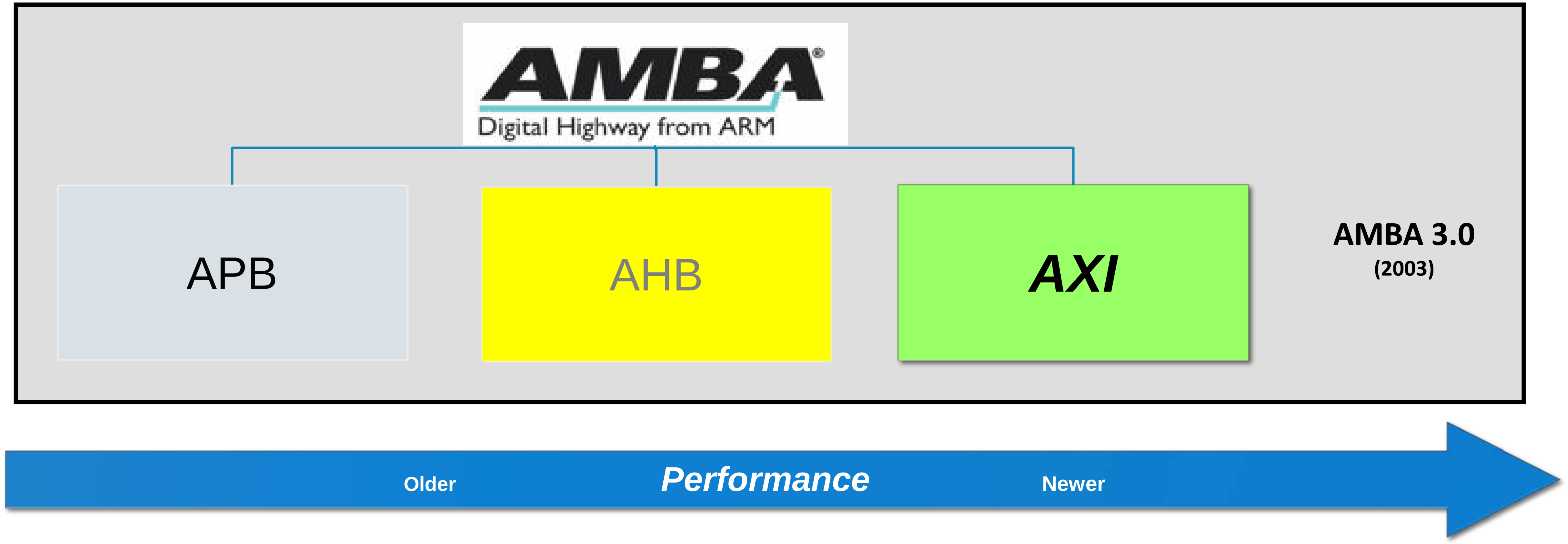
Connectivity -> Standard

- **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 SoC Interfaces

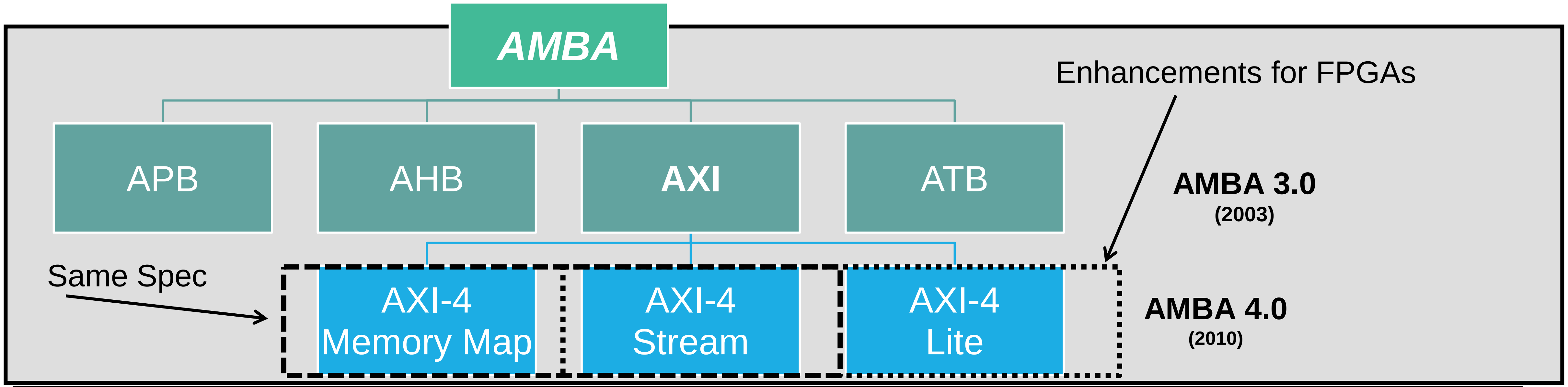
- **Core Connect (IBM)**
  - PLB/OPB (Power PC-FPGA bus interface)
- **WishBone**
  - OpenCore Cores
- **AXI**
  - ARM standard (more to come . . .)

AXI is Part of ARM's AMBA



AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced Extensible Interface

AXI is Part of AMBA



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

AXI Interconnect

AXI is an interconnect system used to tie processors to peripherals

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

AXI – 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

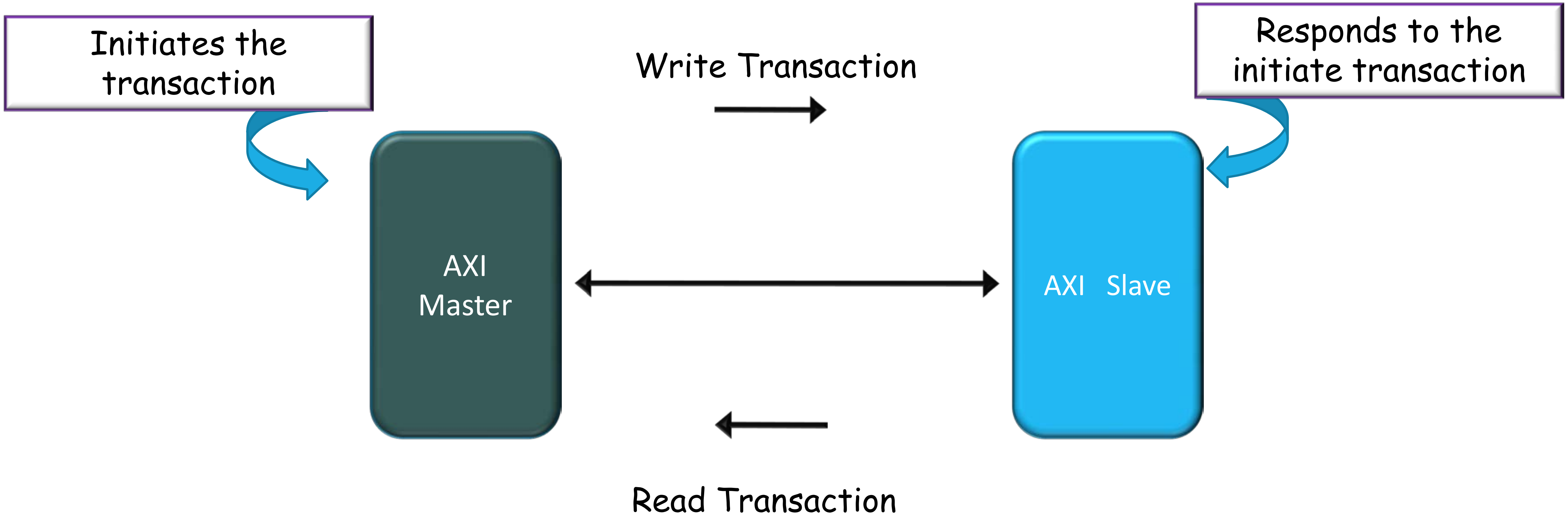
Transaction

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

Burst

- Transaction that consists of more than one transfer

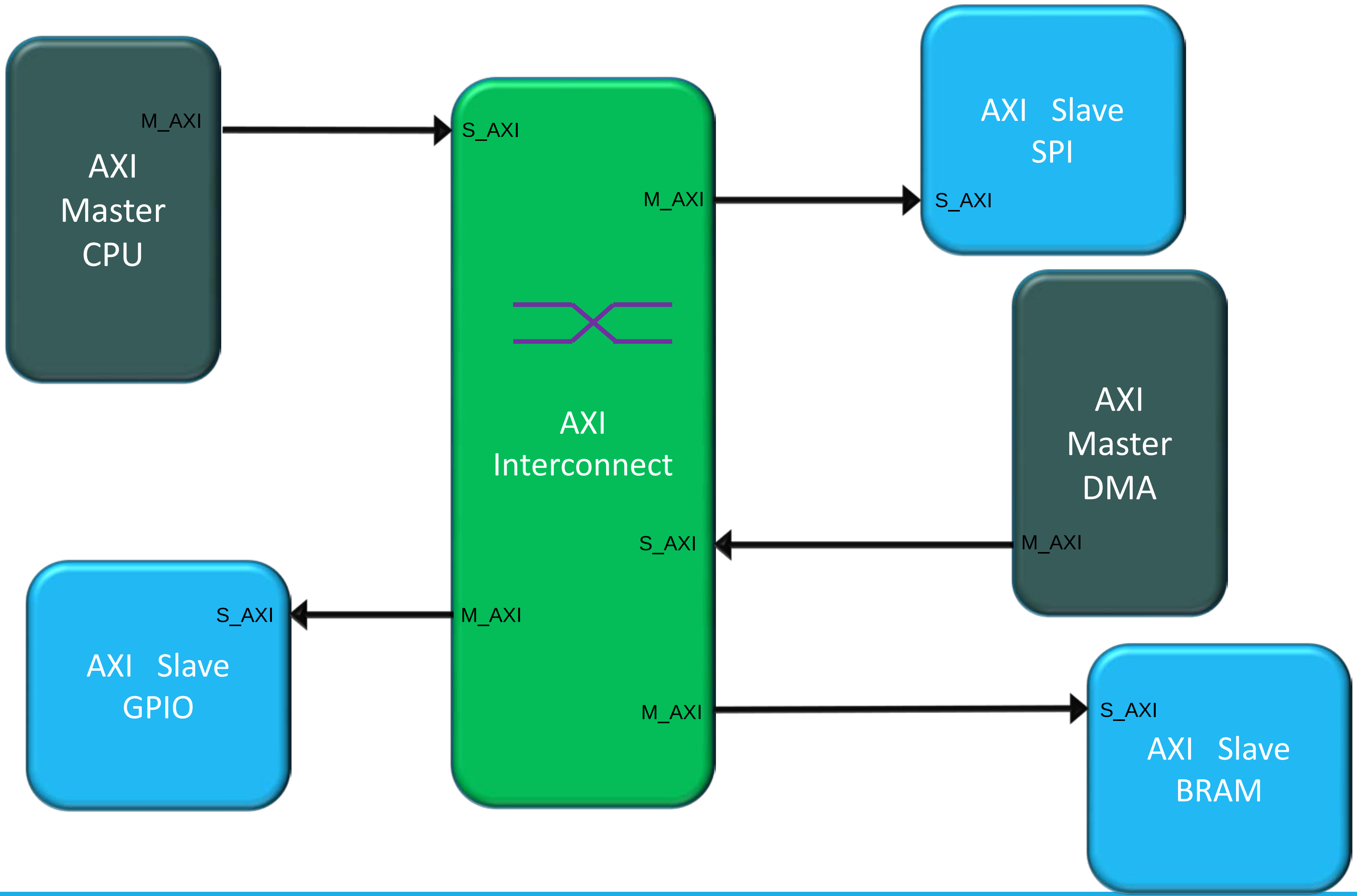
AXI Transactions / Master-Slave



Initiates the transaction
Write Transaction
Responds to the initiate transaction
AXI Master
AXI Slave
Read Transaction

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

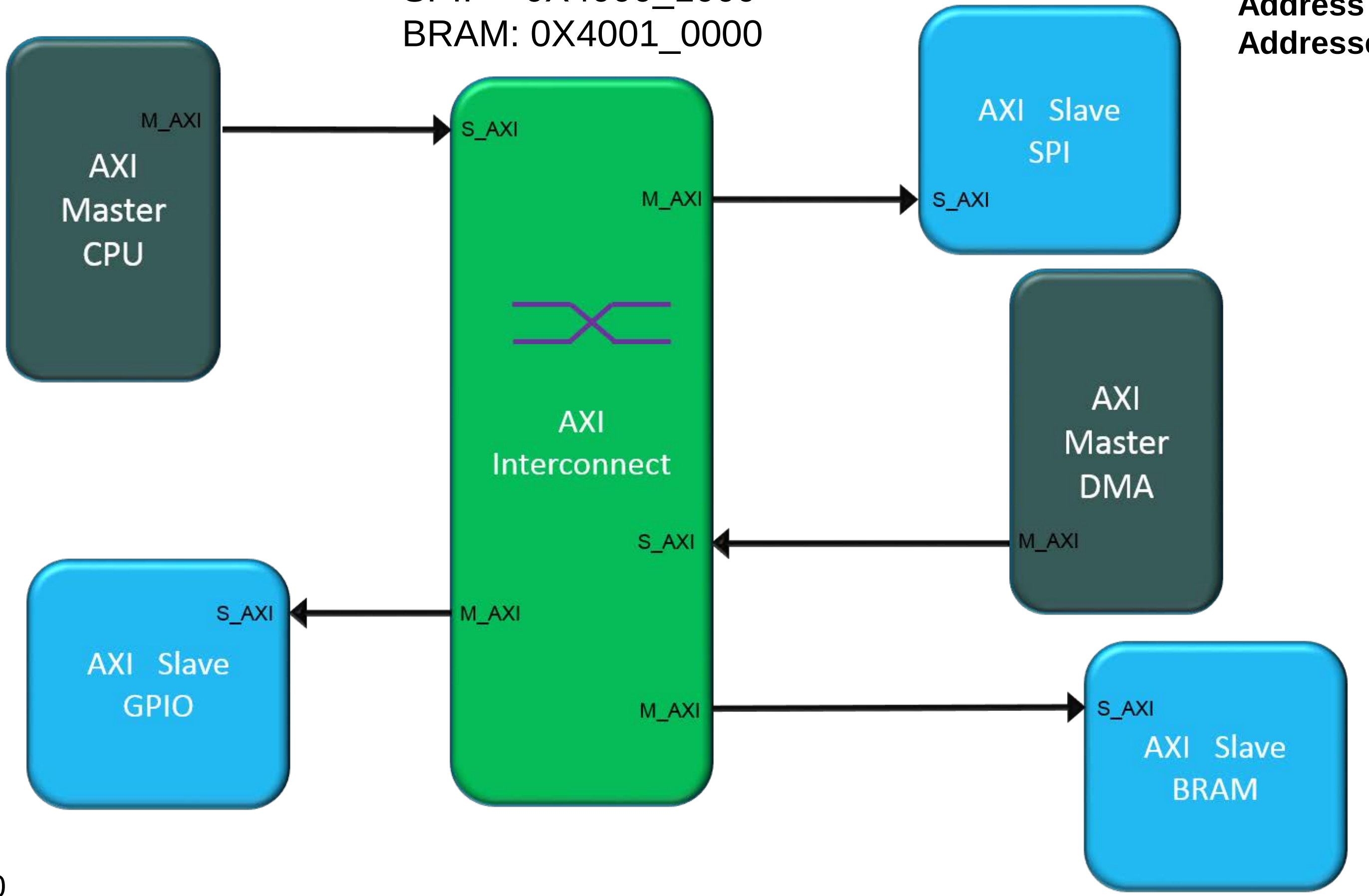
AXI Interconnect



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

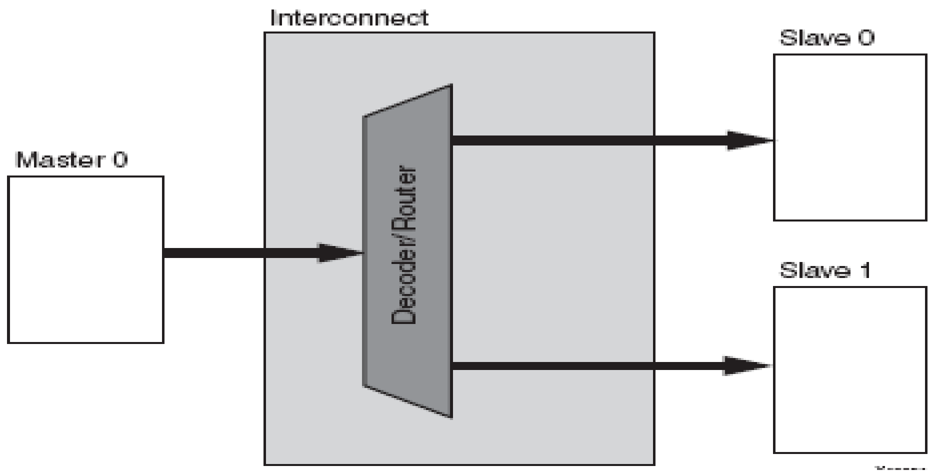
Address Range: 64K
Address Offset: 0X4001_0000
Addresses: 0X4001_0000 – 0X4001_FFFF

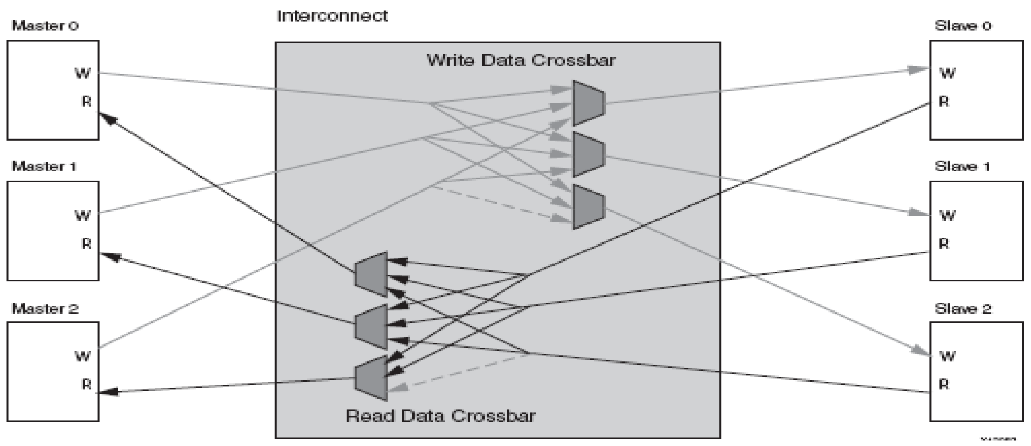
AXI Interconnect Main Features

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

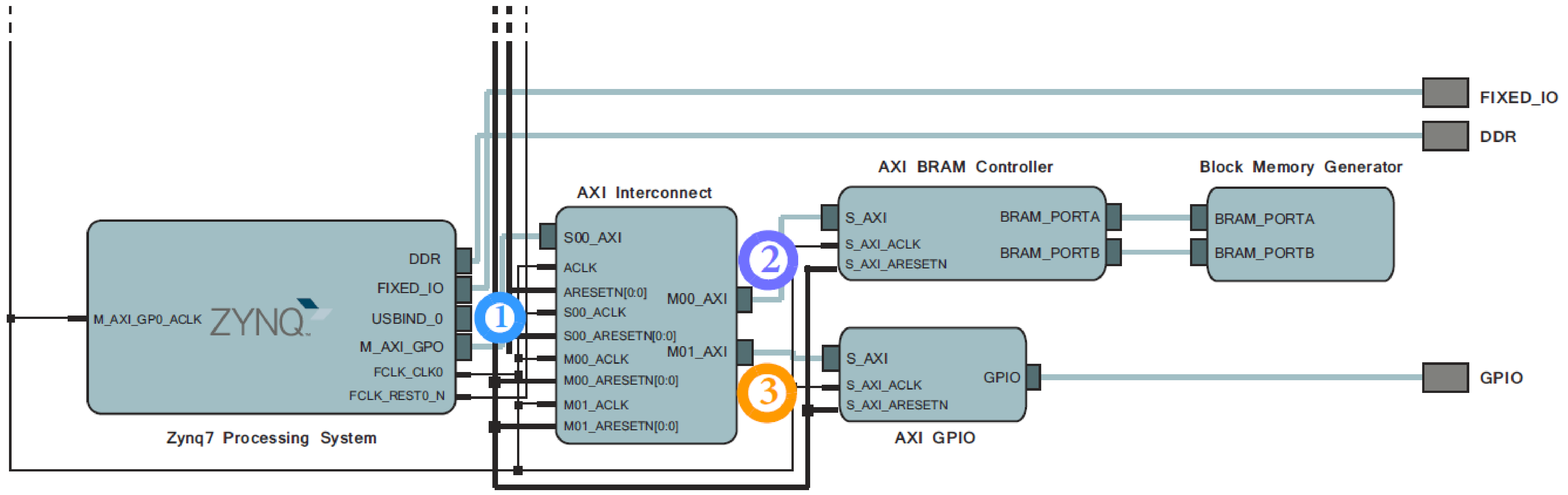
AXI Interconnect

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





AXI Interface Example

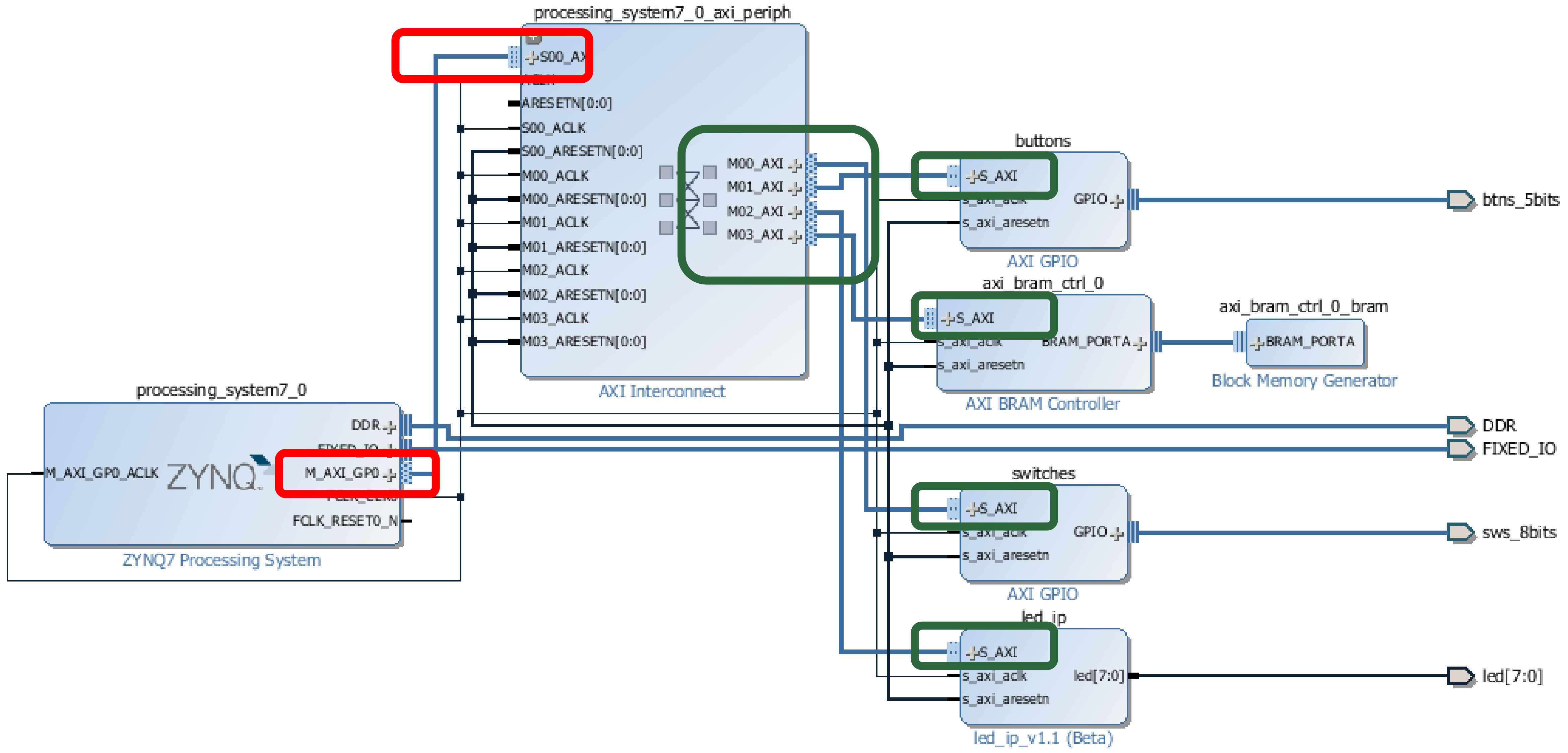


1. The AXI master signal from the Zynq processing system connects to the AXI slave port of the AXI Interconnect block

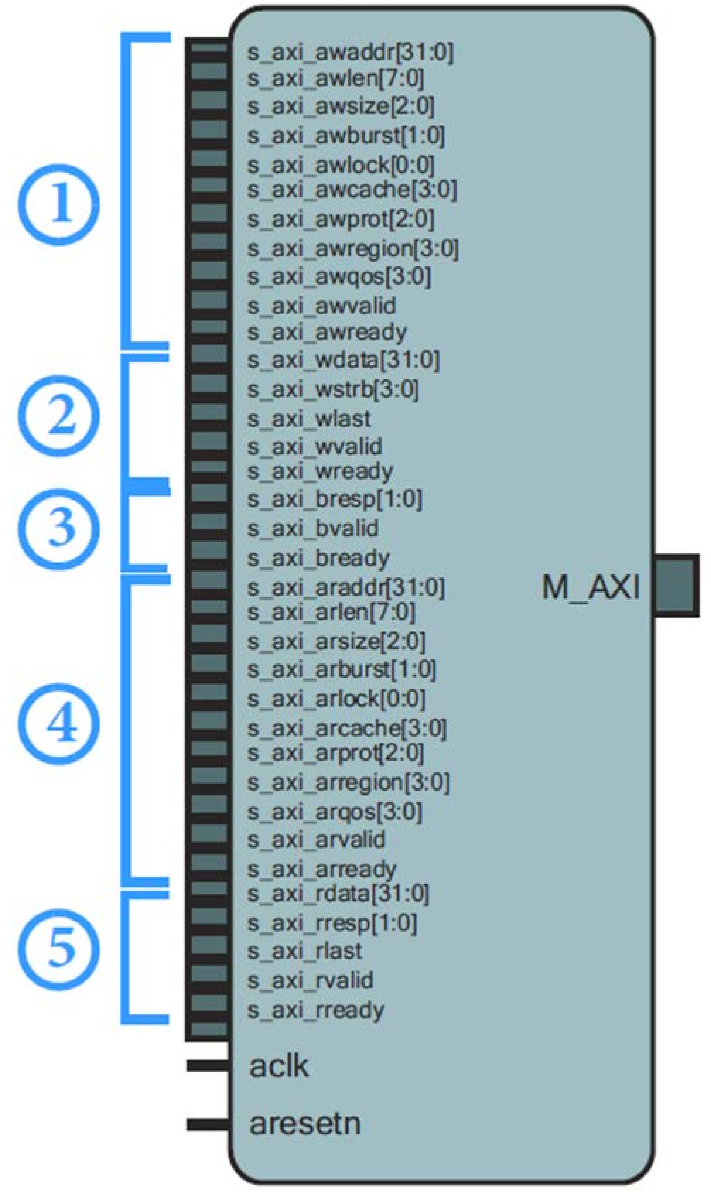
2. An AXI master signal from the AXI Interconnect connects to the AXI slave port of the BRAM controller

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

AXI Interface Example



AXI Slave Signals



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

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

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

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

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

Basic AXI Rd/Wr Process

AXI Channels Use A Basic “VALID/READY” Handshake

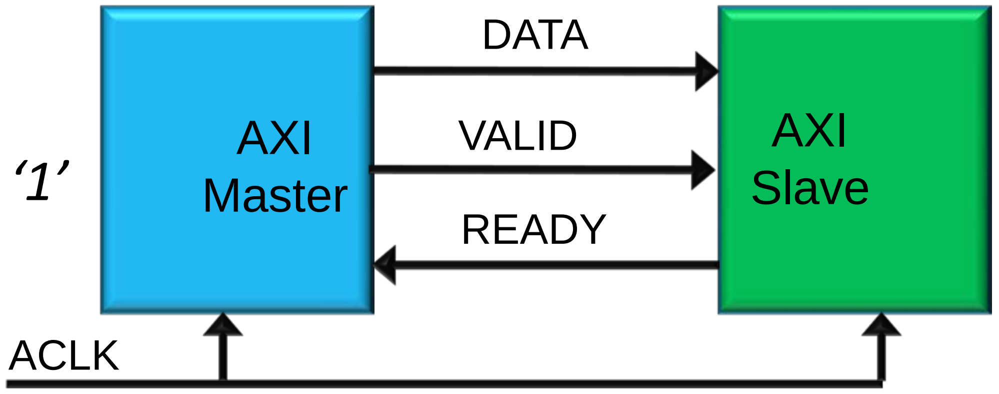
1 *Master asserts and hold VALID when data is available*

2 *Slave asserts READY if able to accept data*

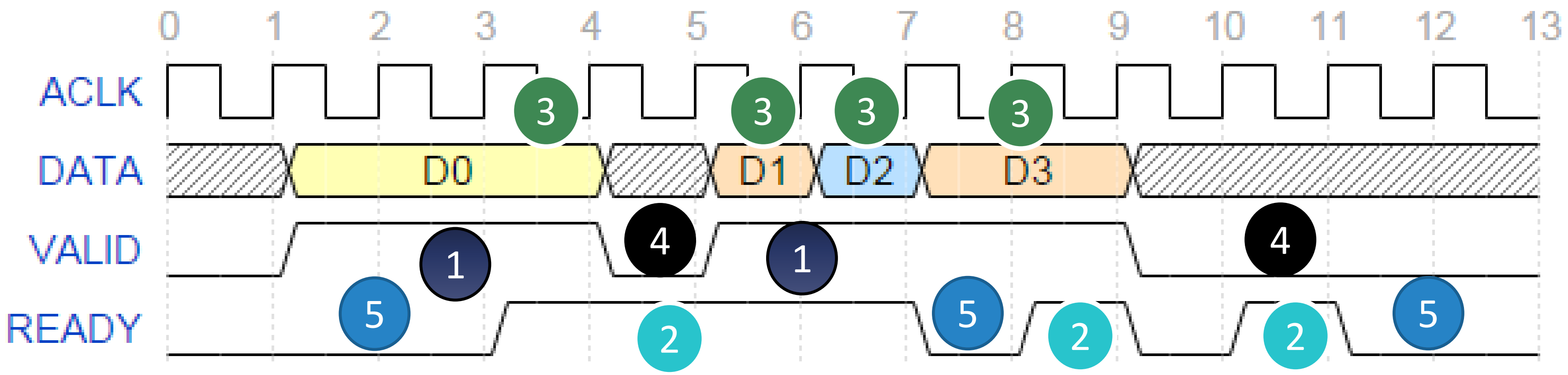
3 *Data and other signals transferred when VALID and READY = ‘1’*

4 *Master sends next DATA/other signals or deasserts VALID*

5 *Slave deasserts READY if no longer able to accept data*

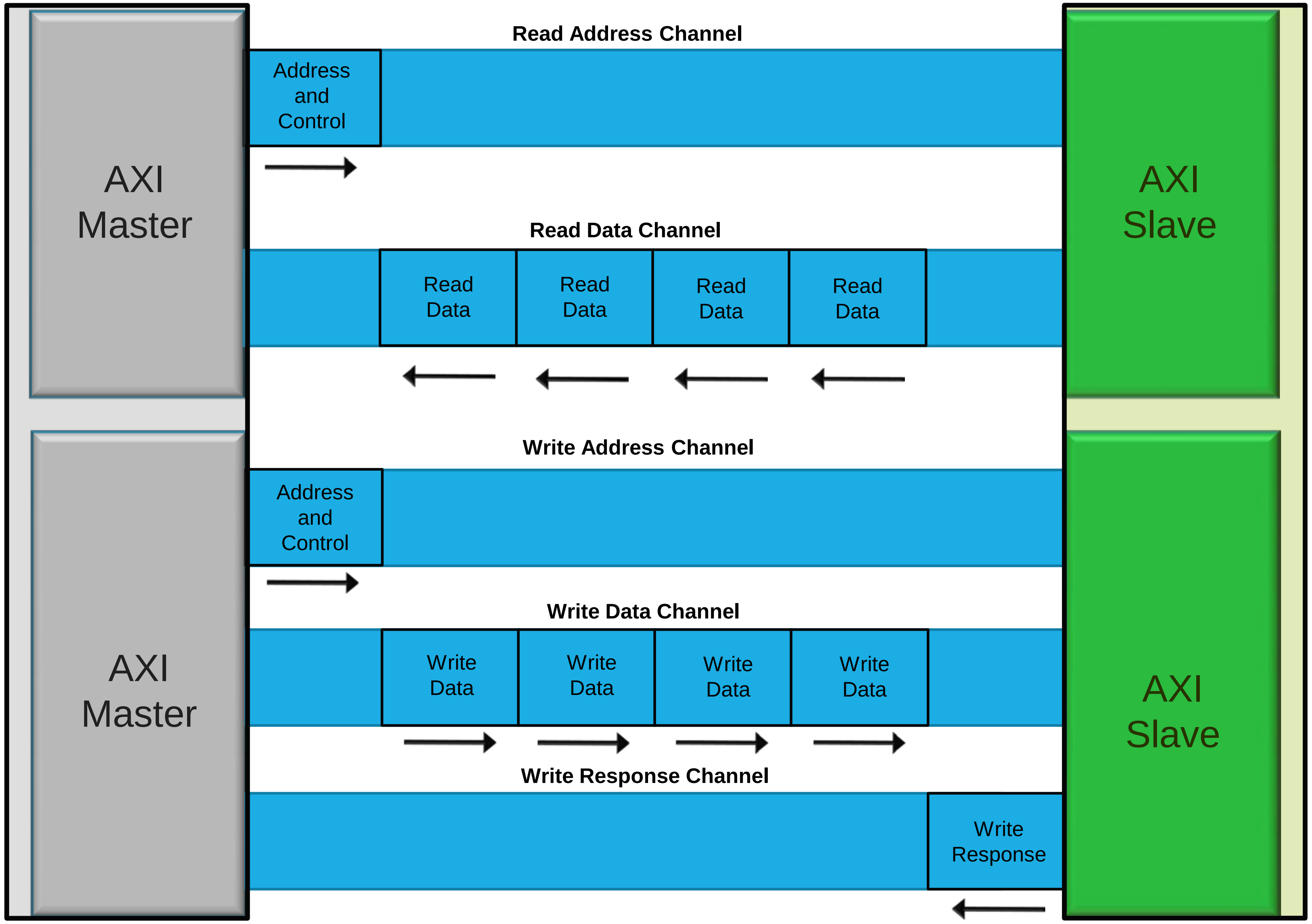


AXI Basic Handshake

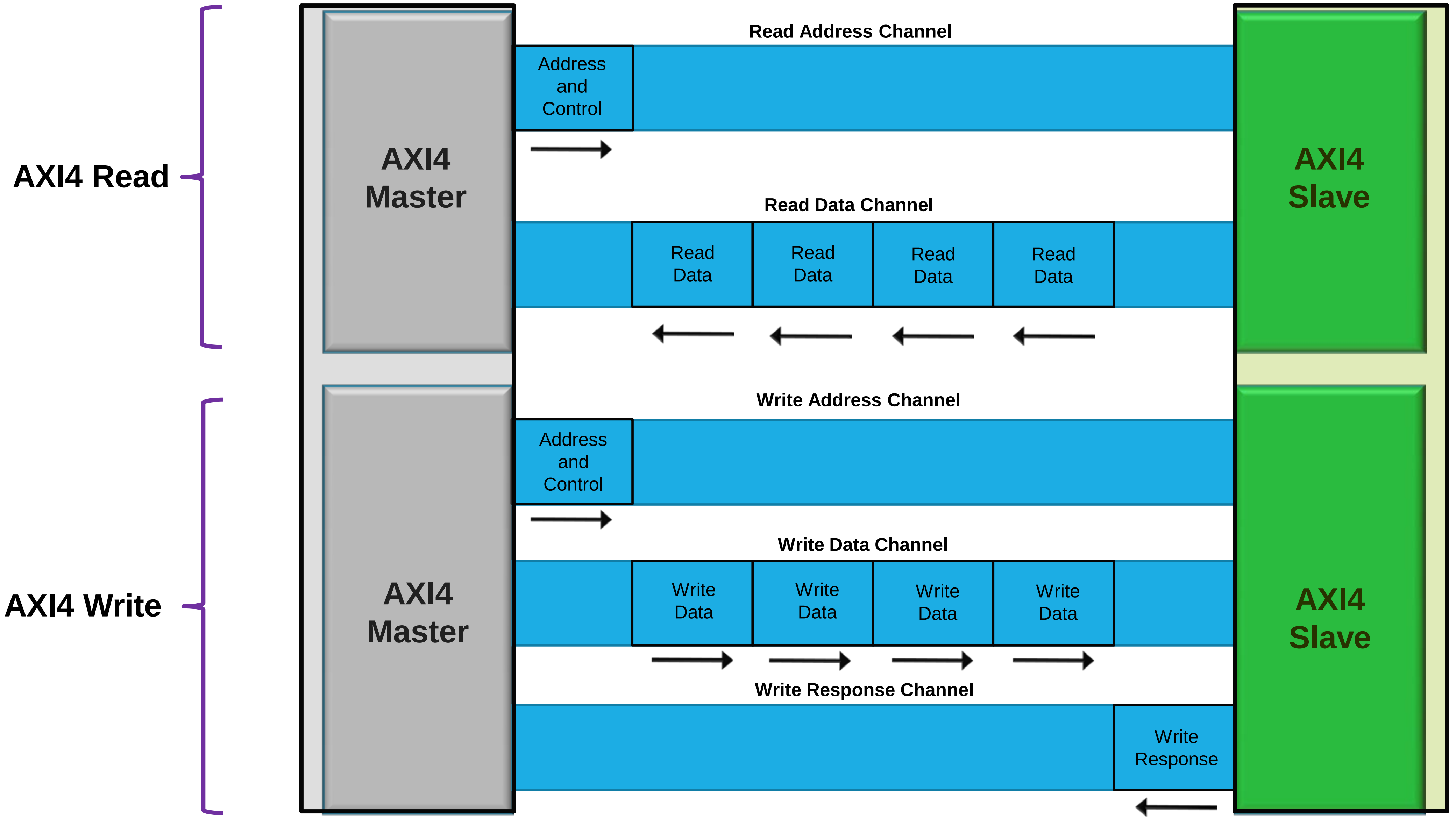


AXI4 Lite

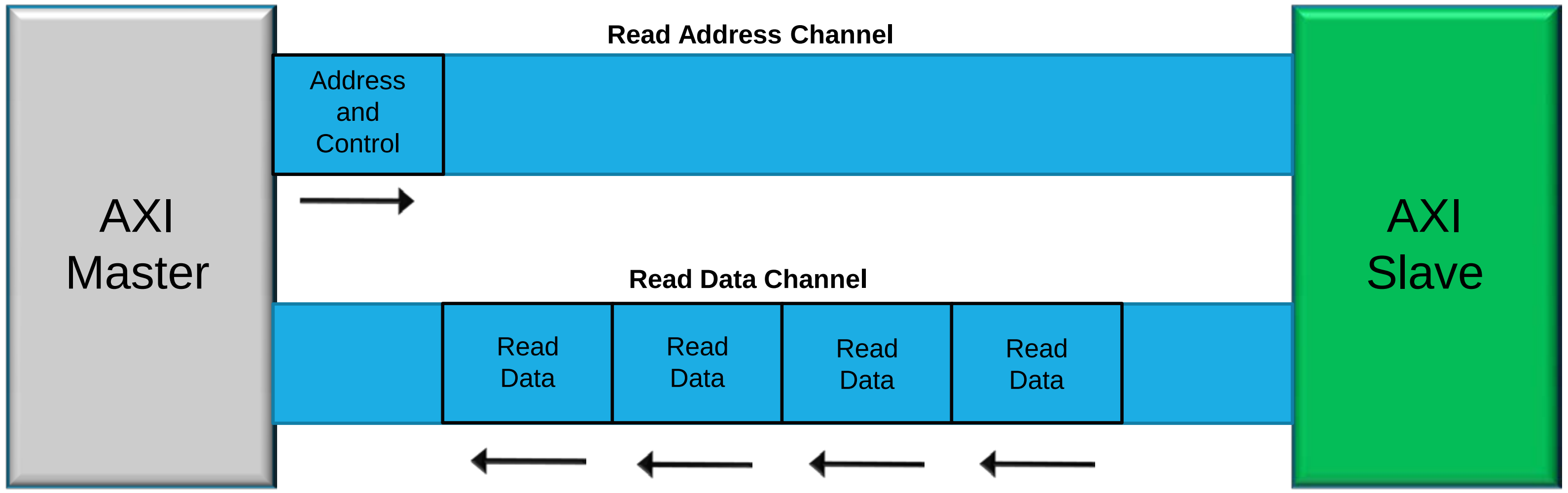
- No Burst
- Single address, single data
- Data Width 32 or 64 bits (Xilinx IP only support 32)
- Very small size
- The AXI Interconnect is automatically generated



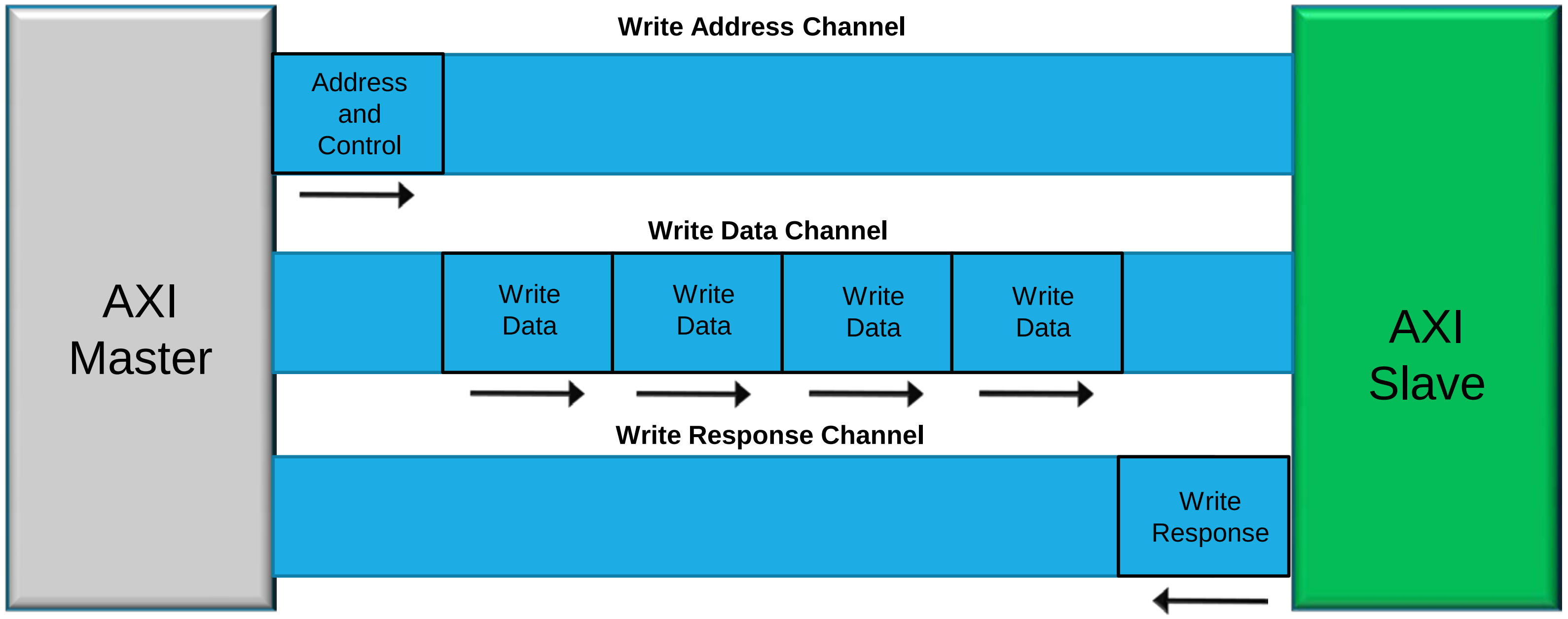
AXI Channels (AXI4 and AXI Lite)



AXI4 Lite Read

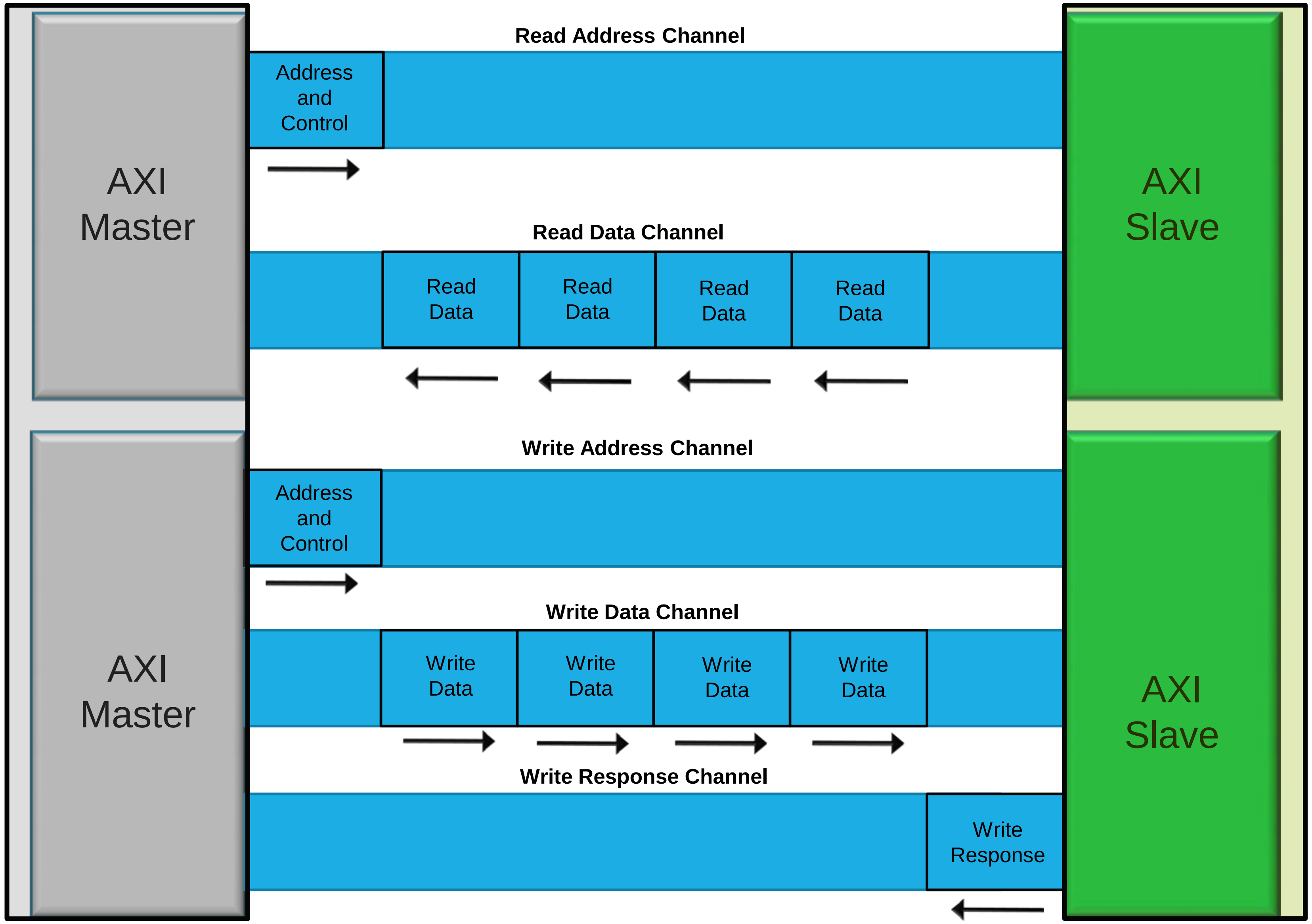


AXI4 Lite Write



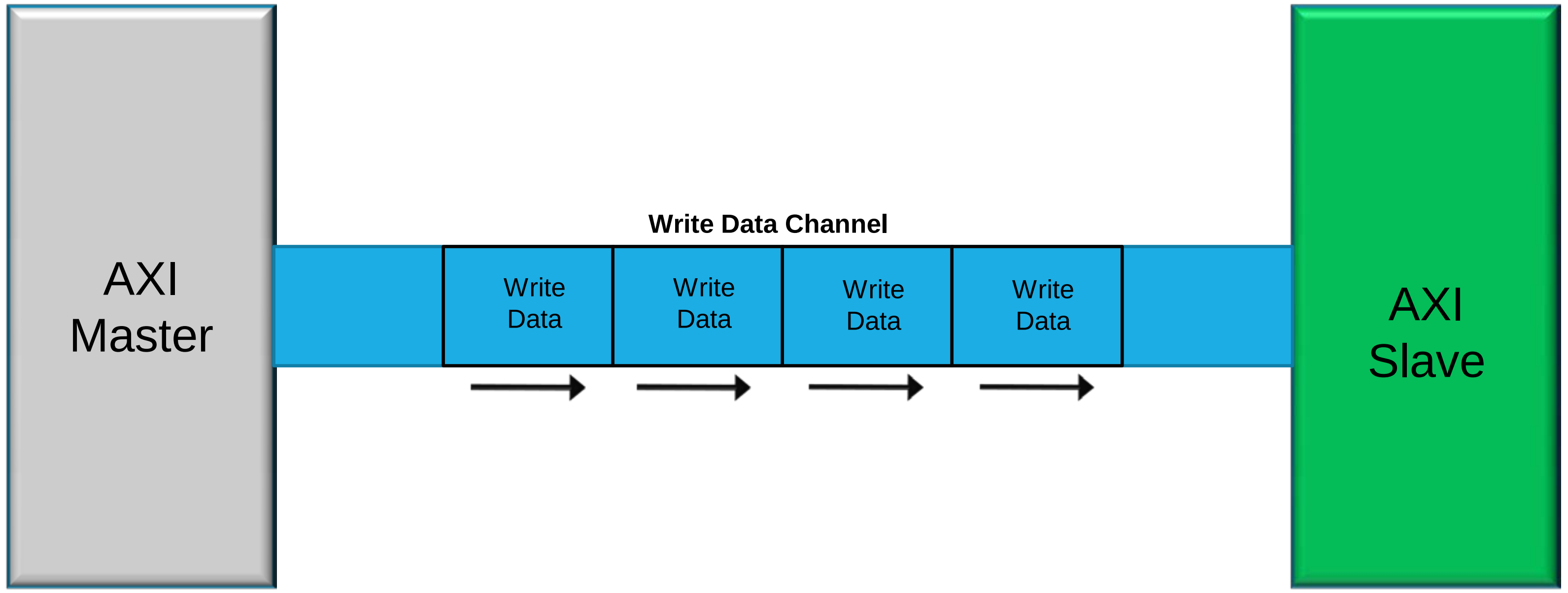
AXI4 (Full)

- 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

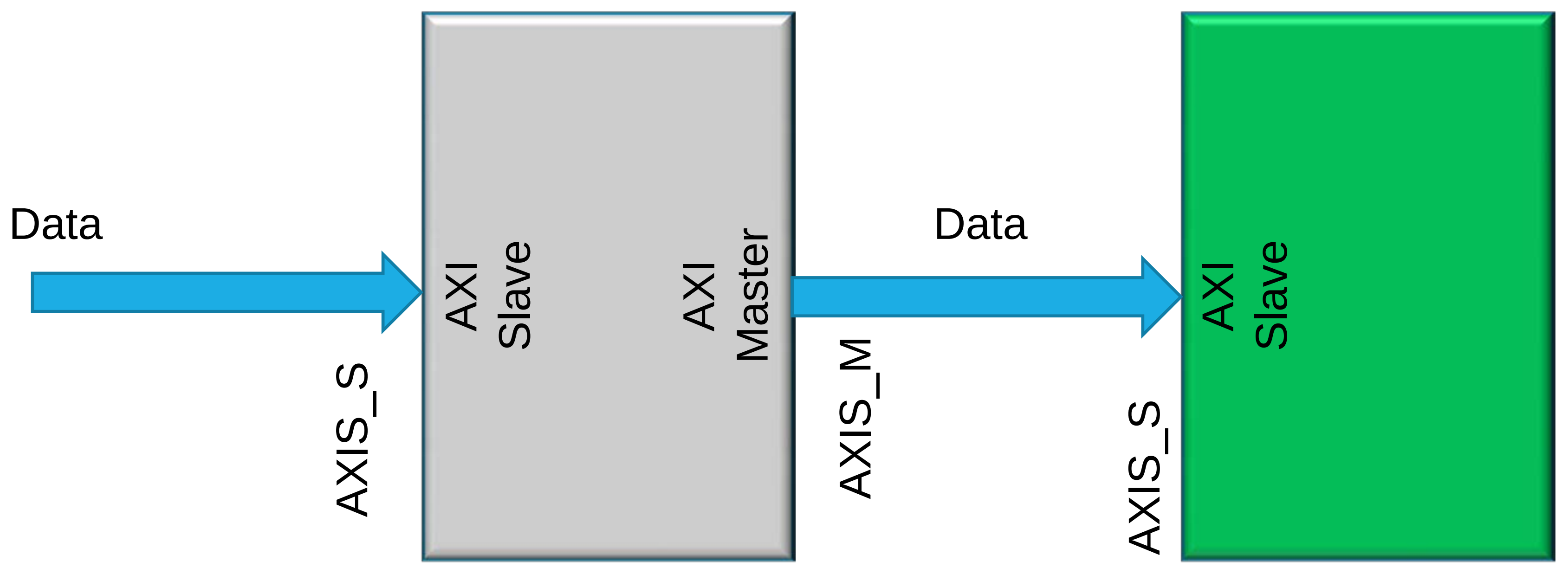


AXI4 Stream

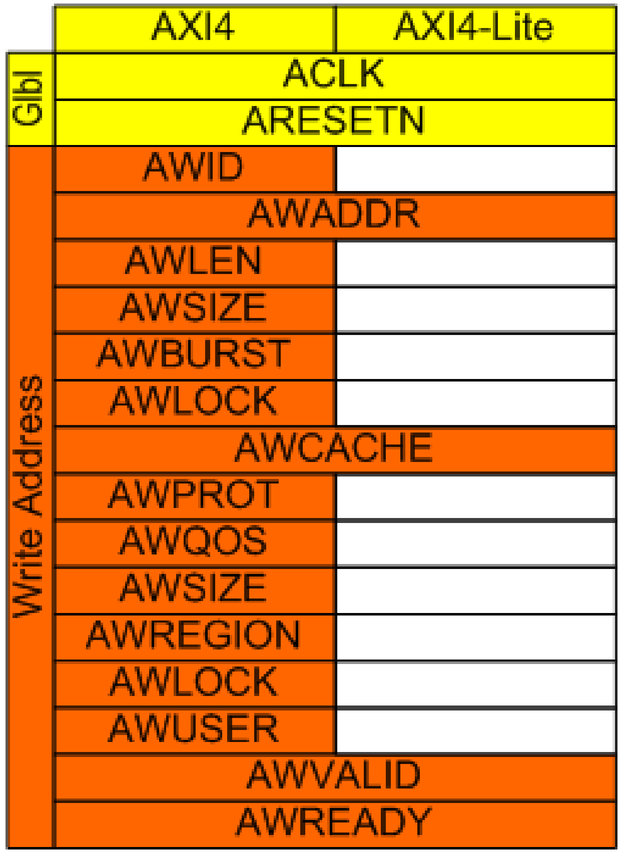
- No address channel, no read and write, always just Master to Slave
  - Just an AXI4 Write Channel
- Unlimited burst length
- Supports sparse, continuous, aligned, unaligned streams



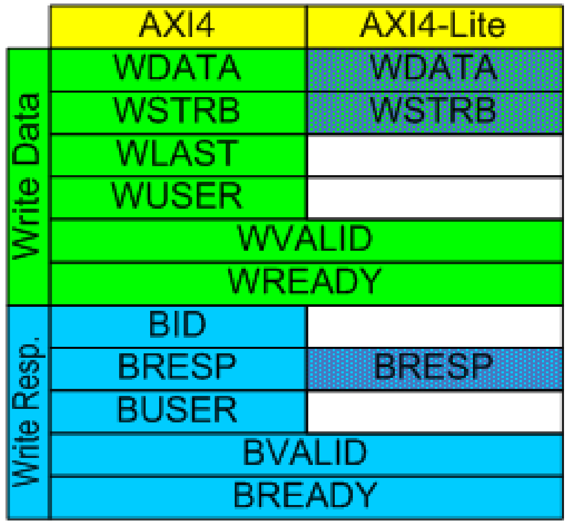
AXI Stream



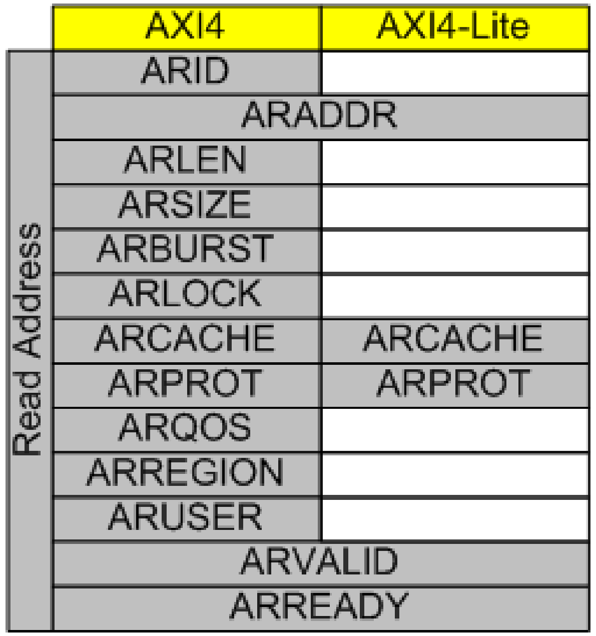
AXI4 – AXI Lite: Signals Available



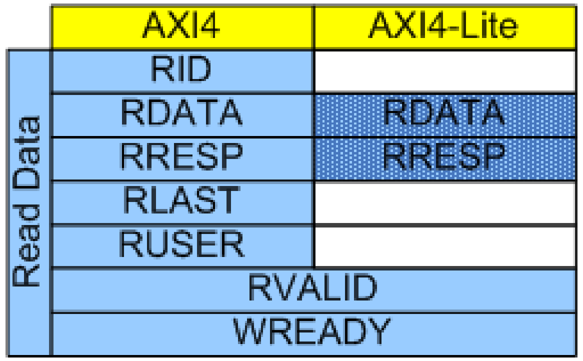
	AXI4	AXI4-Lite
Glbl	ACLK	ACLK
	ARESETN	ARESETN
Write Address	AWID	
	AWADDR	AWADDR
	AWLEN	
	AWSIZE	
	AWBURST	
	AWLOCK	
	AWCACHE	AWCACHE
	AWPROT	
	AWQOS	
	AWSIZE	
	AWREGION	
	AWLOCK	
	AWUSER	
	AWVALID	AWVALID
	AWREADY	AWREADY



	AXI4	AXI4-Lite
Write Data	WDATA	WDATA
	WSTRB	WSTRB
	WLAST	
	WUSER	
	WVALID	WVALID
	WREADY	WREADY
Write Resp.	BID	
	BRESP	BRESP
	BUSER	
	BVALID	BVALID
	BREADY	BREADY



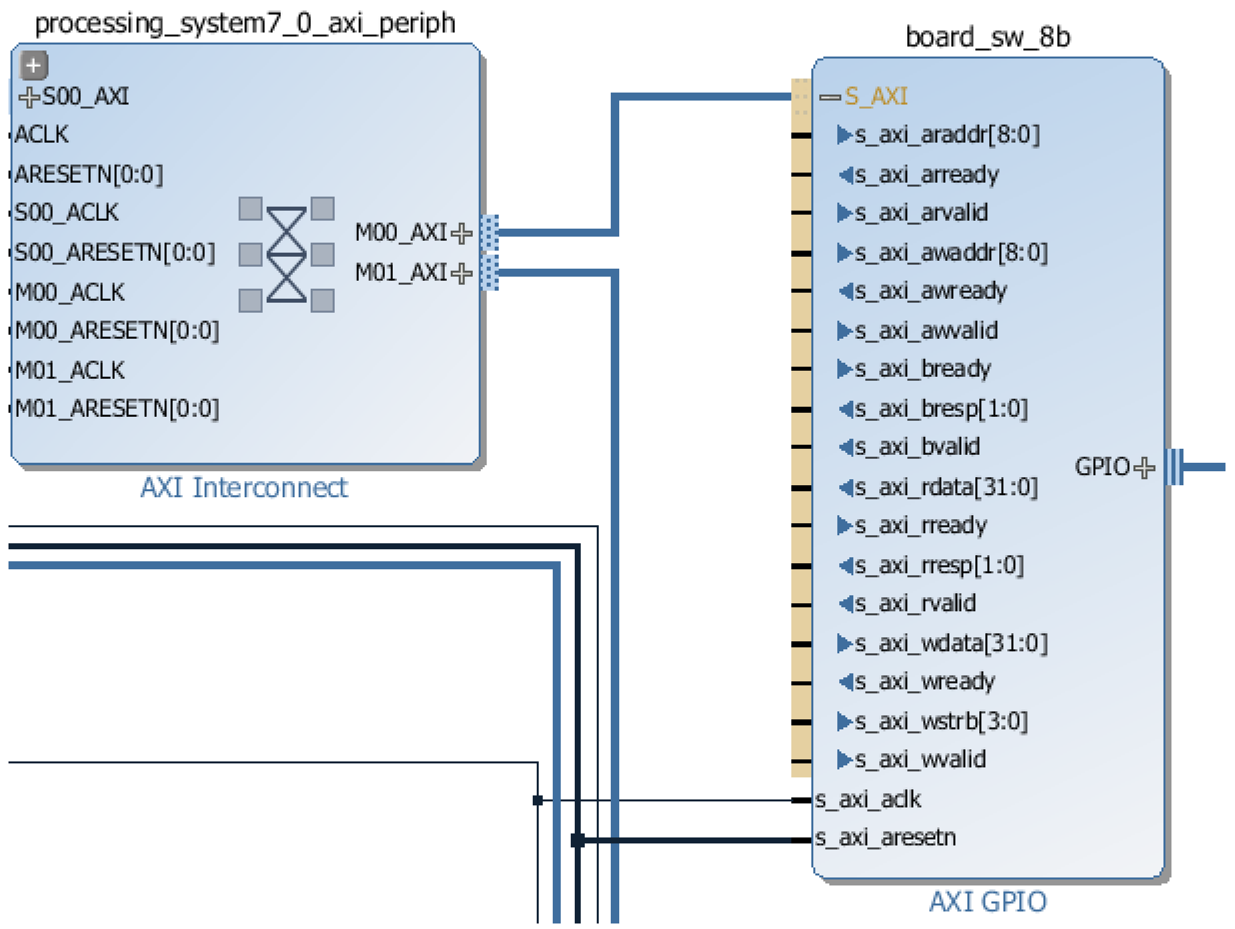
	AXI4	AXI4-Lite
Read Address	ARID	
	ARADDR	ARADDR
	ARLEN	
	ARSIZE	
	ARBURST	
	ARLOCK	
	ARCACHE	ARCACHE
	ARPROT	ARPROT
	ARQOS	
	ARREGION	
	ARUSER	
	ARVALID	ARVALID
	ARREADY	ARREADY



	AXI4	AXI4-Lite
Read Data	RID	
	RDATA	RDATA
	RRESP	RRESP
	RLAST	
	RUSER	
	RVALID	RVALID
	WREADY	WREADY

AXI4-Lite Custom IP
The VHDL Underneath

AXI4-Lite Signal Names

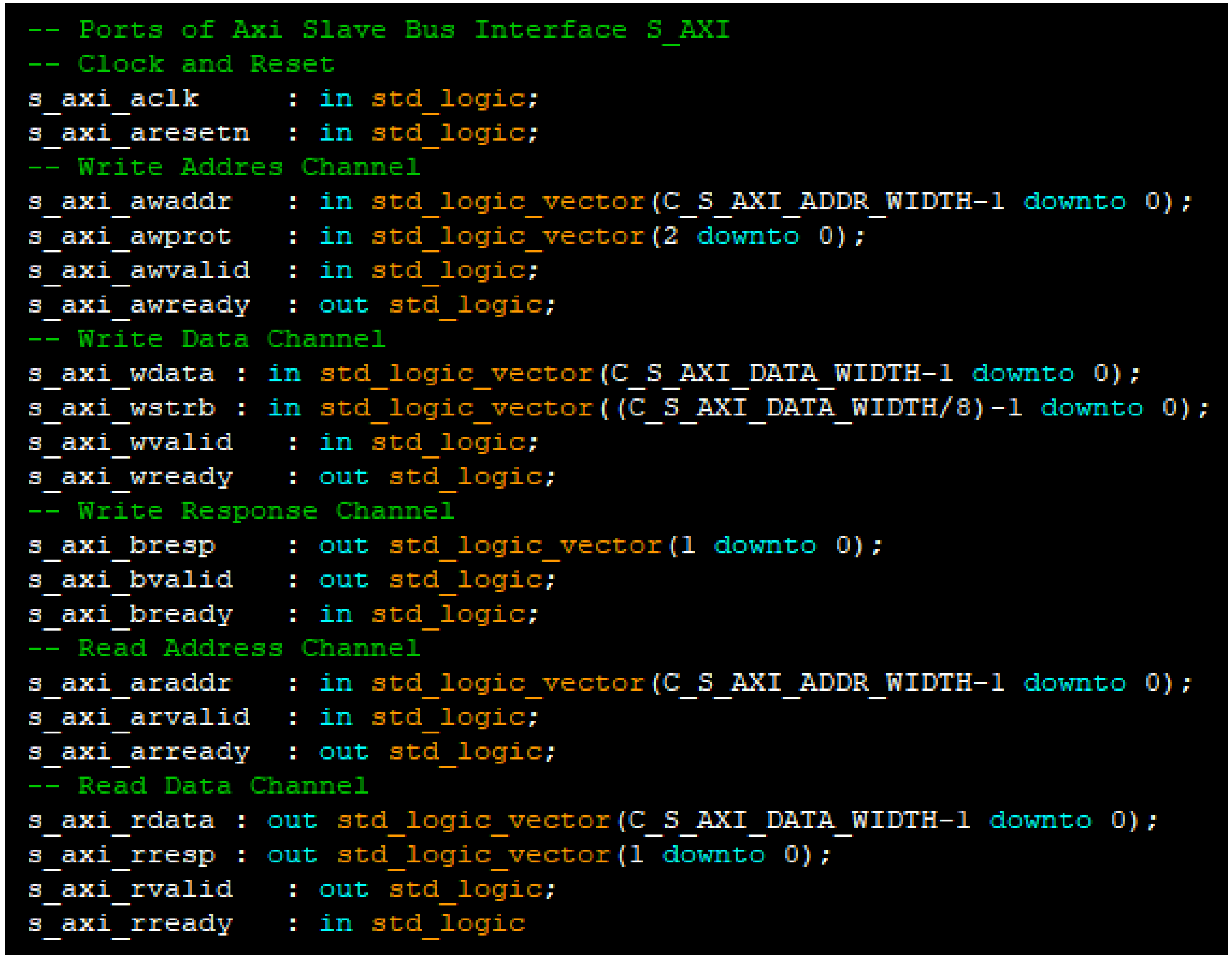


AXI4-Lite Signal Names

- 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

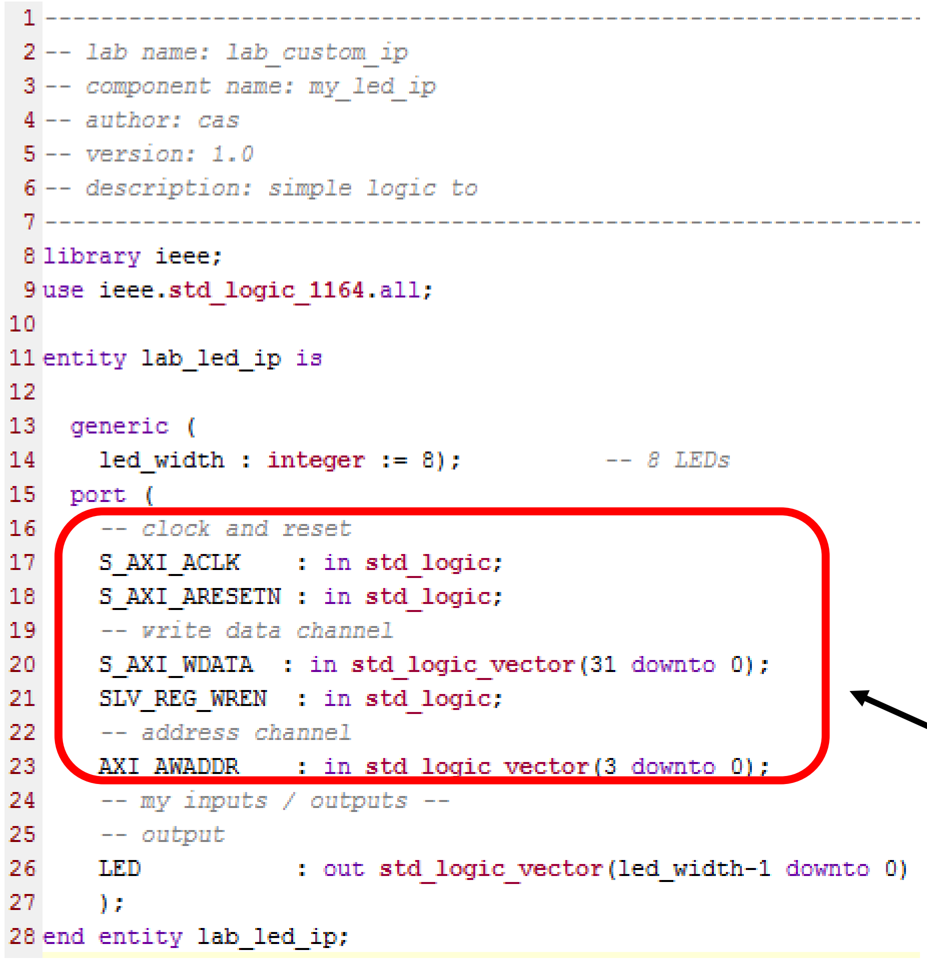
- In previous versions of the Xilinx design flow (where PLB and OPB peripherals were typically used) it was necessary for each IP peripheral connected to the processor to individually decode all transactions that were presented by a master on the bus (“multi-drop”). it was the responsibility of each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

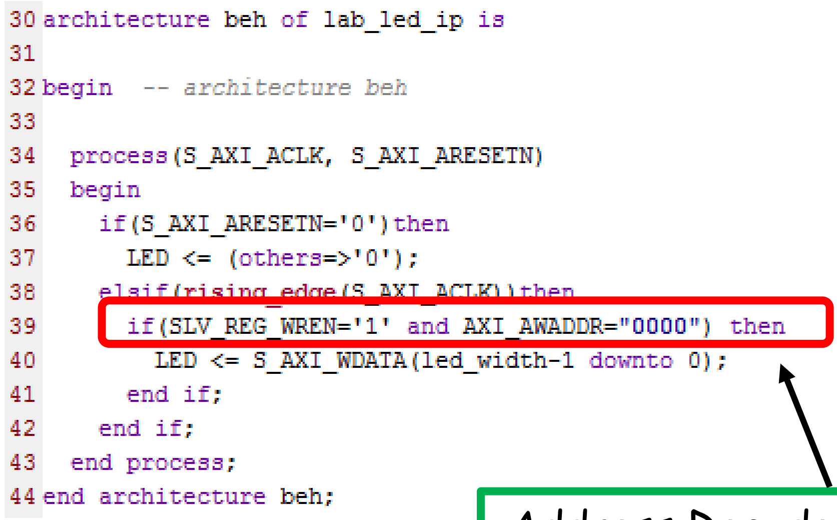
- **The designer merely needs to decode enough of the incoming address bus to determine which of the registers in the slave IP should be read or written**

My VHDL Code – Address Decoding



```vhdl
---------------------------------------------------------------
-- lab name: lab_custom_ip
-- component name: my_led_ip
-- author: cas
-- version: 1.0
-- description: simple logic to
---------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity lab_led_ip is

  generic (
    led_width : integer := 8);          -- 8 LEDs
  port (
    -- clock and reset
    S_AXI_ACLK    : in std_logic;
    S_AXI_ARESETN : in std_logic;
    -- write data channel
    S_AXI_WDATA   : in std_logic_vector(31 downto 0);
    SLV_REG_WREN  : in std_logic;
    -- address channel
    AXI_AWADDR    : in std_logic_vector(3 downto 0);
    -- my inputs / outputs --
    -- output
    LED           : out std_logic_vector(led_width-1 downto 0)
    );
end entity lab_led_ip;
```

AXI4-Lite IP

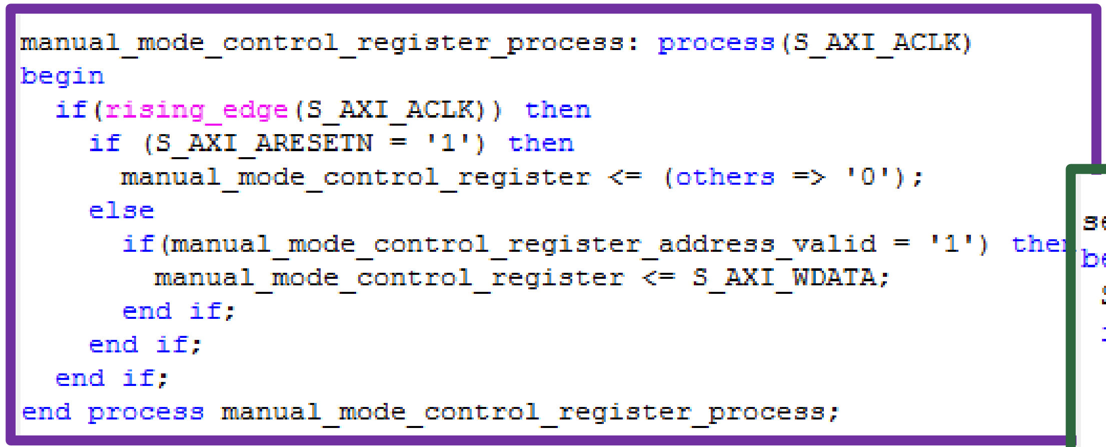


```vhdl
architecture beh of lab_led_ip is

begin  -- architecture beh

  process(S_AXI_ACLK, S_AXI_ARESETN)
  begin
    if(S_AXI_ARESETN='0')then
      LED <= (others=>'0');
    elsif(rising_edge(S_AXI_ACLK))then
      if(SLV_REG_WREN='1' and AXI_AWADDR="0000") then
        LED <= S_AXI_WDATA(led_width-1 downto 0);
      end if;
    end if;
  end process;
end architecture beh;
```

Address Decode & Write Enable

AXI4-Lite – Implementing Addressable Registers

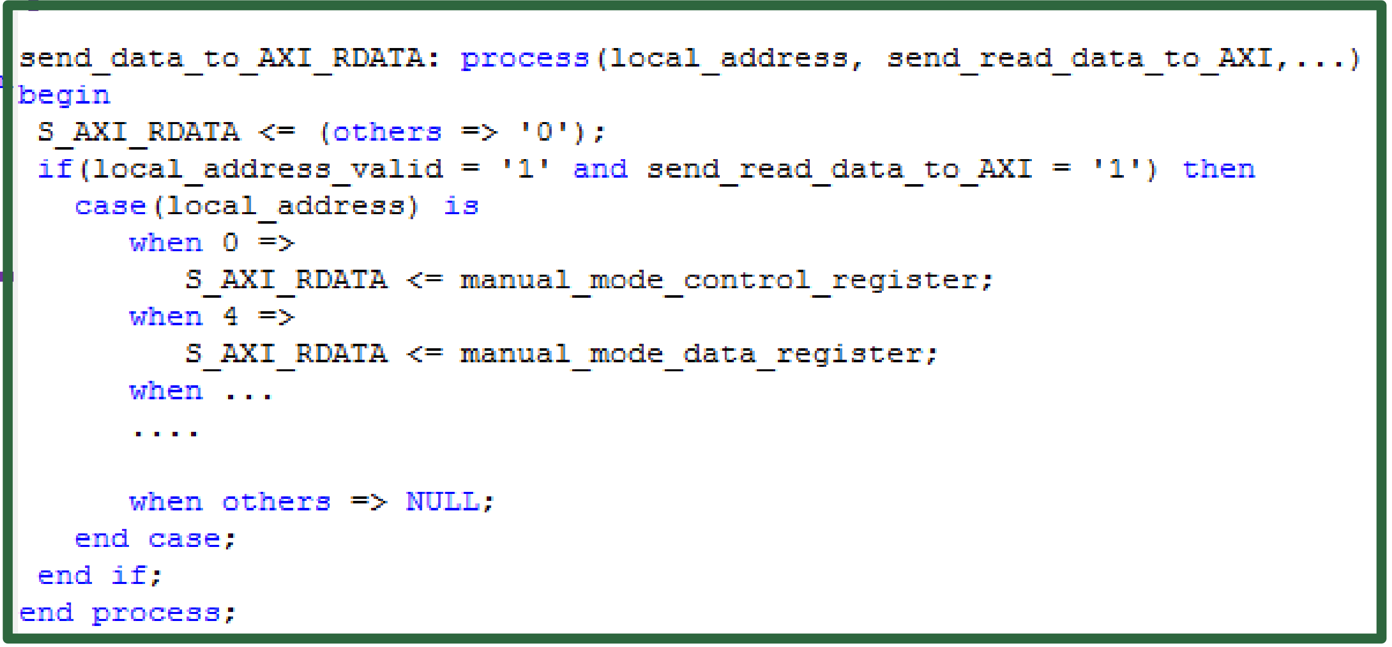
- Using the address decoding scheme above, it is extremely simple to implement registers in VHDL which can receive data values written by a master on the AXI4-lite interconnect. The following extract of code shows how an individual register can be quickly and easily implemented (in this case mapped to 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') the
        manual_mode_control_register <= S_AXI_WDATA;
      end if;
    end if;
  end if;
end process manual_mode_control_register_process;
```

WriteTransaction

Read Transaction

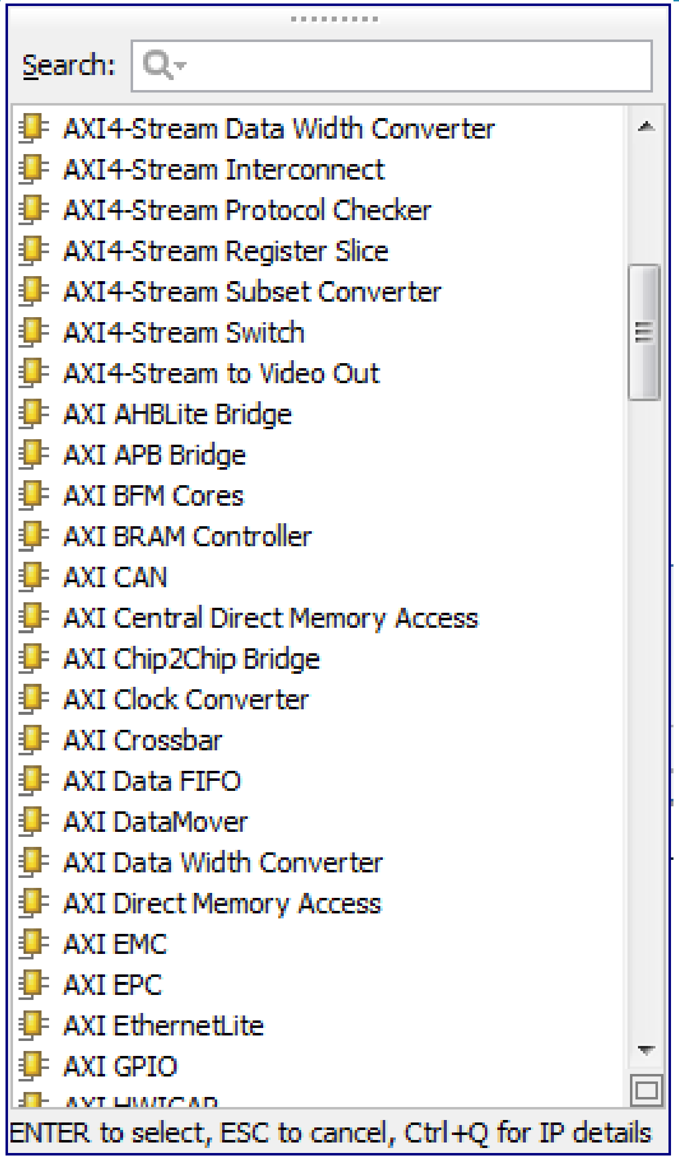


```
send_data_to_AXI_RDATA: process(local_address, send_read_data_to_AXI,...)
begin
 S_AXI_RDATA <= (others => '0');
 if(local_address_valid = '1' and send_read_data_to_AXI = '1') then
   case(local_address) is
      when 0 =>
         S_AXI_RDATA <= manual_mode_control_register;
      when 4 =>
   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

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

Custom AXI IPs

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



Search:

AXI4-Stream Data Width Converter
AXI4-Stream Interconnect
AXI4-Stream Protocol Checker
AXI4-Stream Register Slice
AXI4-Stream Subset Converter
AXI4-Stream Switch
AXI4-Stream to Video Out
AXI AHBLite Bridge
AXI APB Bridge
AXI BFM Cores
AXI BRAM Controller
AXI CAN
AXI Central Direct Memory Access
AXI Chip2Chip Bridge
AXI Clock Converter
AXI Crossbar
AXI Data FIFO
AXI DataMover
AXI Data Width Converter
AXI Direct Memory Access
AXI EMC
AXI EPC
AXI EthernetLite
AXI GPIO

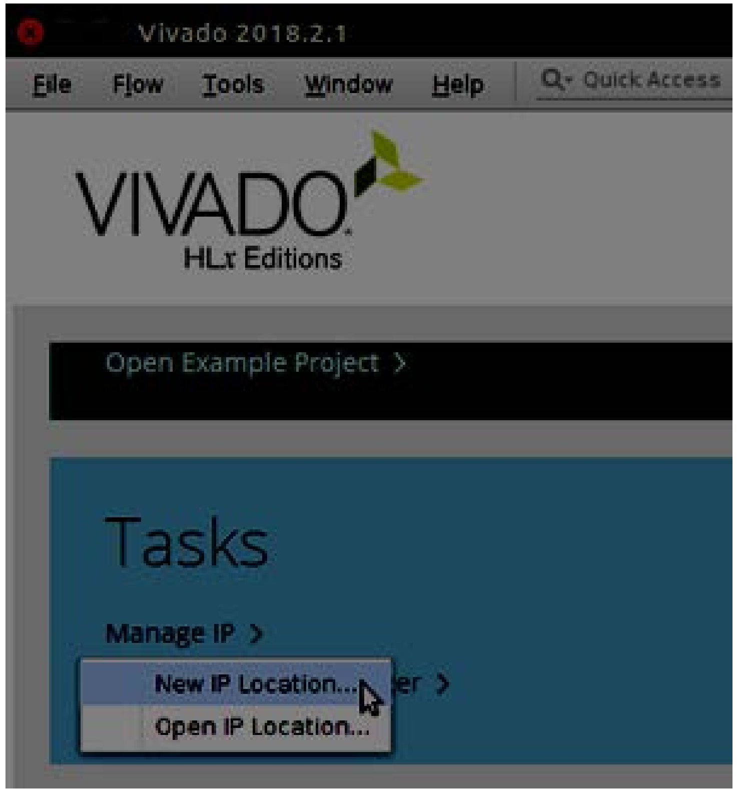
ENTER to select, ESC to cancel, Ctrl+Q for IP details

IP Packager

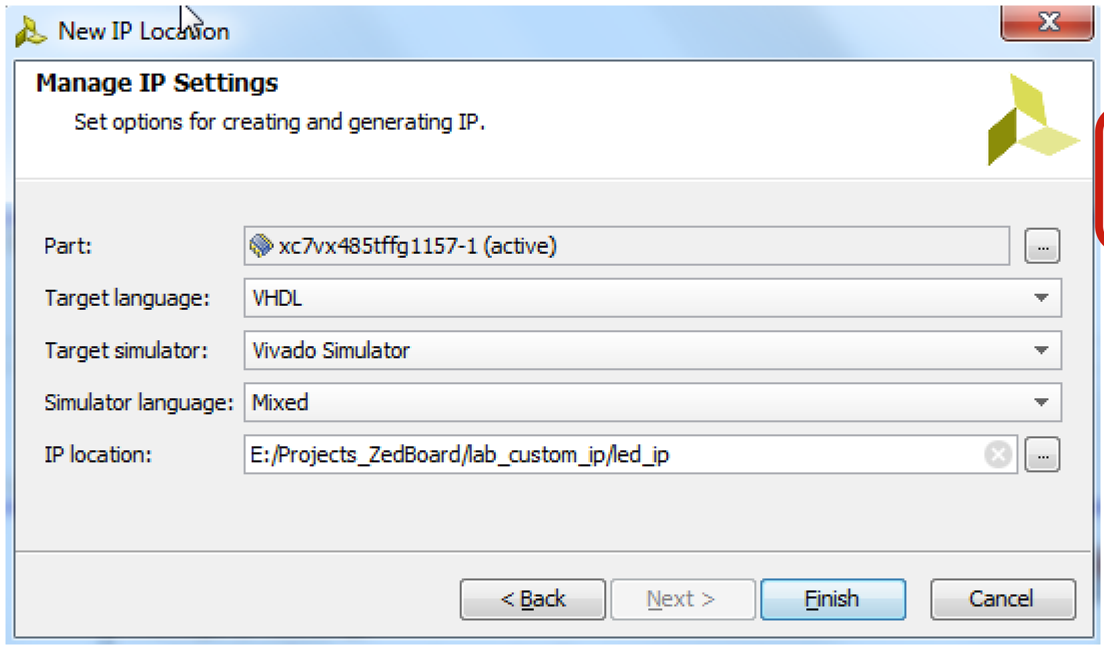
- ❑ The IP Packager allows a core to be packaged and included in the IP Catalog, or for distribution
- ❑ IP-XACT Industry Standard (IEEE) XML format to describe IP using meta-data
  - o Ports
  - o Interfaces
  - o Configurable Parameters
  - o Files, documentation
- o IP-XACT only describes high level 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

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

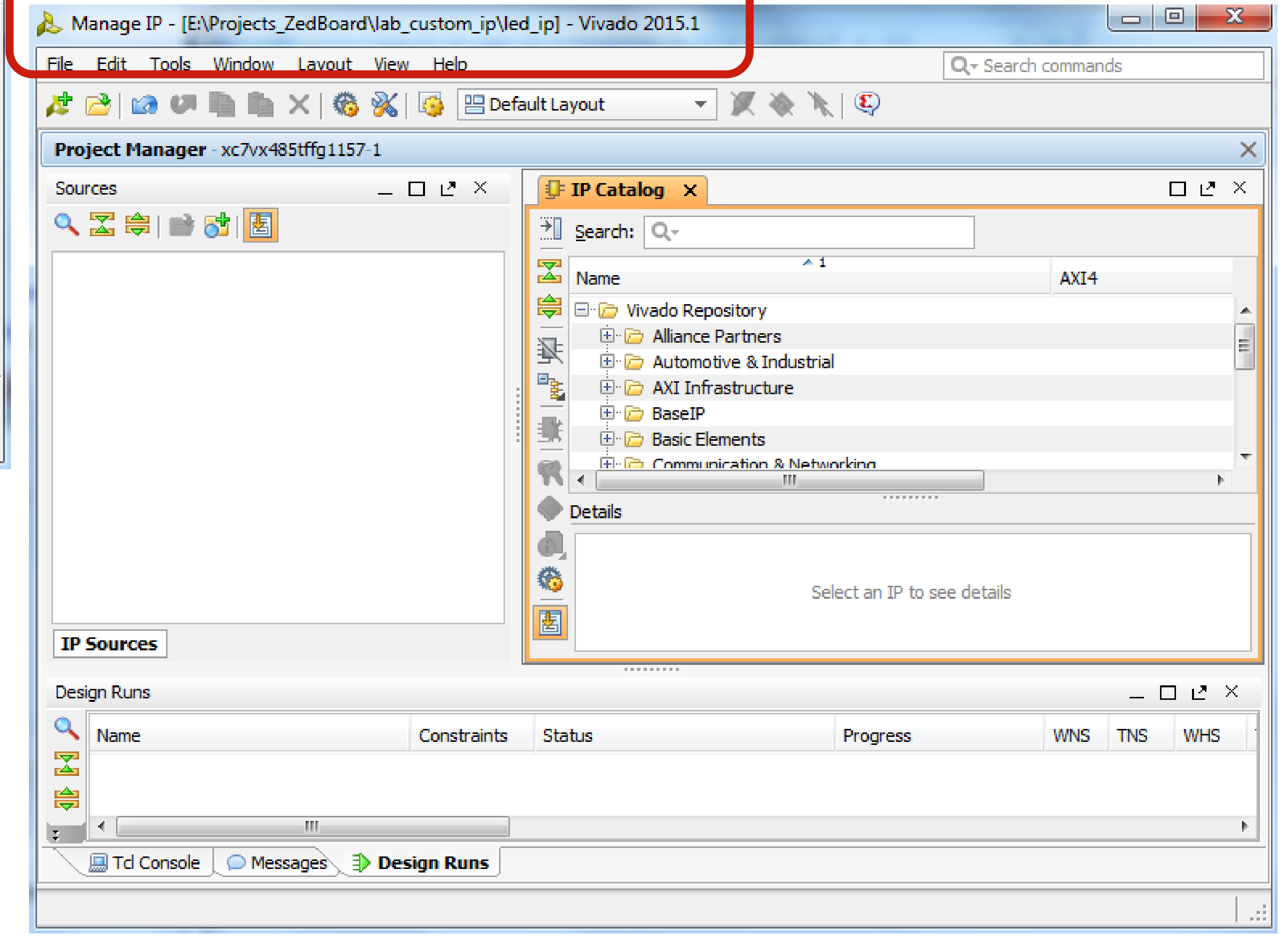


New IP Location

Manage IP Settings

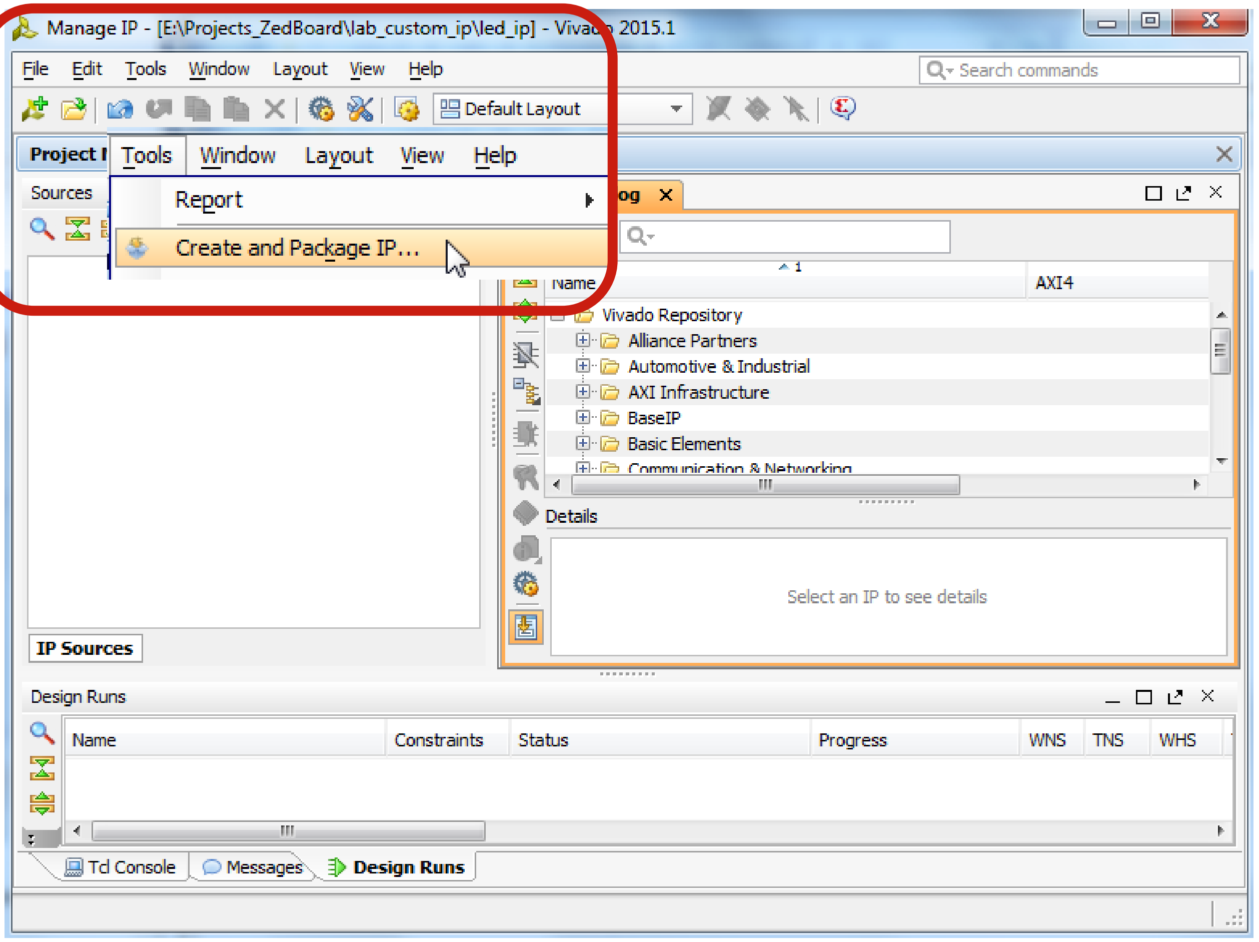
Set options for creating and generating IP.

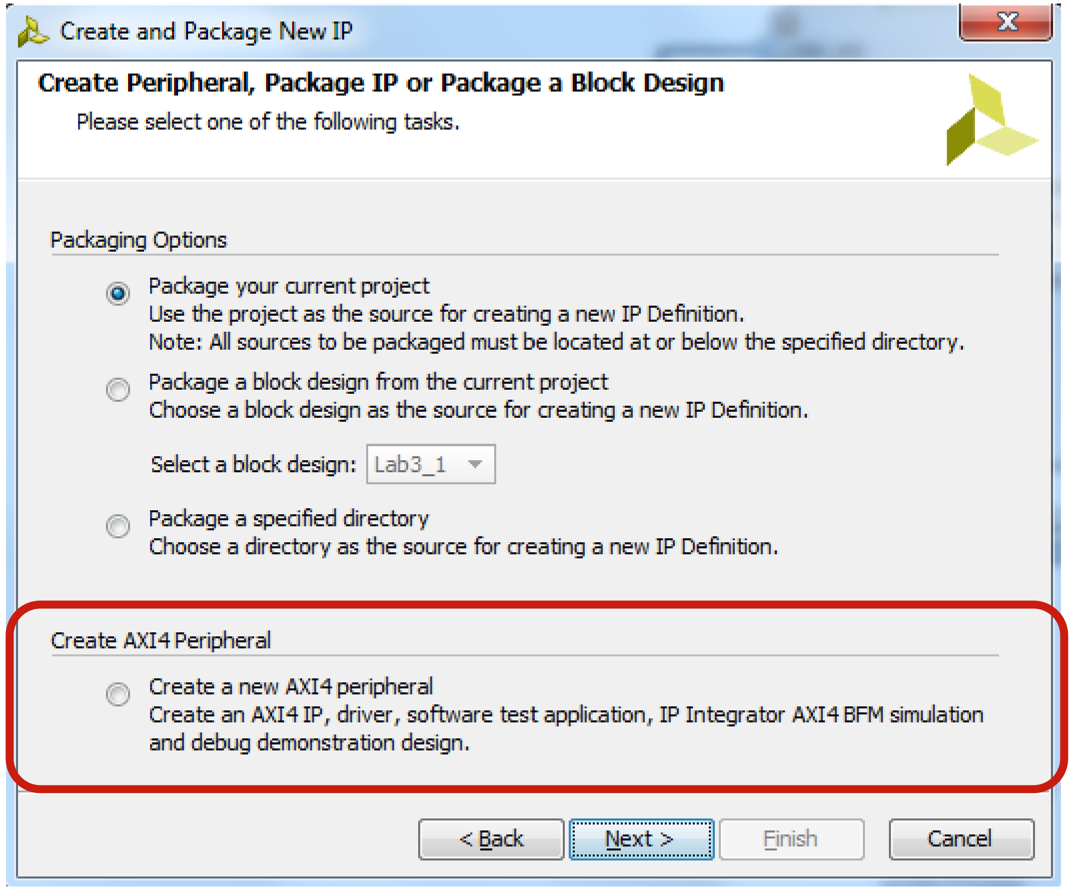
Part:	xc7vx485tffg1157-1 (active)
Target language:	VHDL
Target simulator:	Vivado Simulator
Simulator language:	Mixed
IP location:	E:/Projects_ZedBoard/lab_custom_ip/led_ip



Manage IP - [E:\Projects_ZedBoard\lab_custom_ip\led_ip] - Vivado 2015.1

Create Custom AXI4 IP


Manage IP - [E:\Projects_ZedBoard\lab_custom_ip\led_ip] - Vivado 2015.1
File Edit Tools Window Layout View Help
Default Layout
Search commands
Project I Tools Window Layout View Help
Sources
Report
Create and Package IP...
Name
AXI4
Vivado Repository
Alliance Partners
Automotive & Industrial
AXI Infrastructure
BaseIP
Basic Elements
Communication & Networking
Details
Select an IP to see details
IP Sources
Design Runs
Name Constraints Status Progress WNS TNS WHS
Tcl Console Messages Design Runs


Create and Package New IP

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

Packaging Options

[✓ Package your current project]
Use the project as the source for creating a new IP Definition.
Note: All sources to be packaged must be located at or below the specified directory.

Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.

Select a block design: Lab3_1

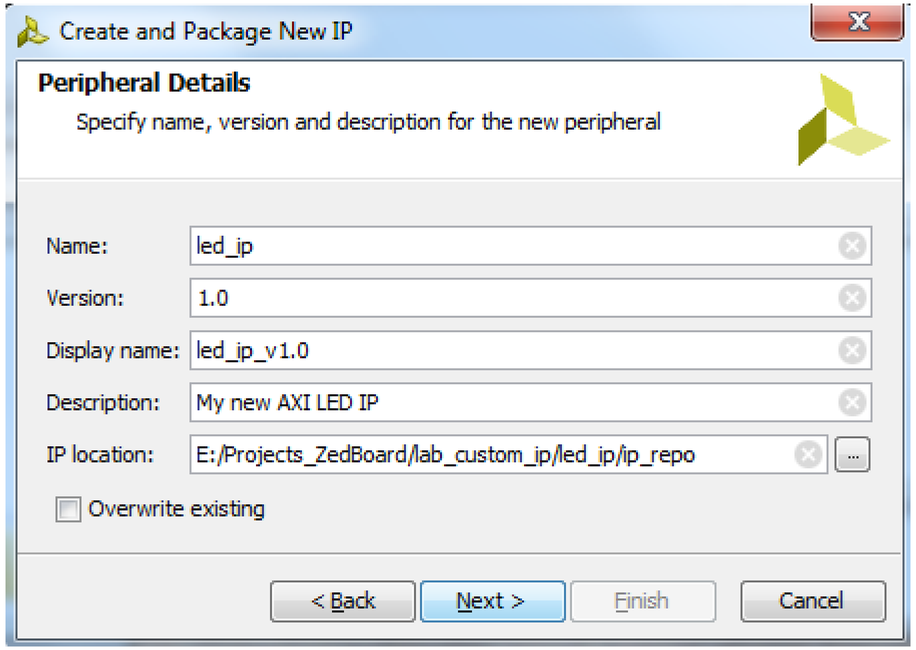
Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 BFM simulation and debug demonstration design.

< Back | Next > | Finish | Cancel

Create Custom AXI4 IP



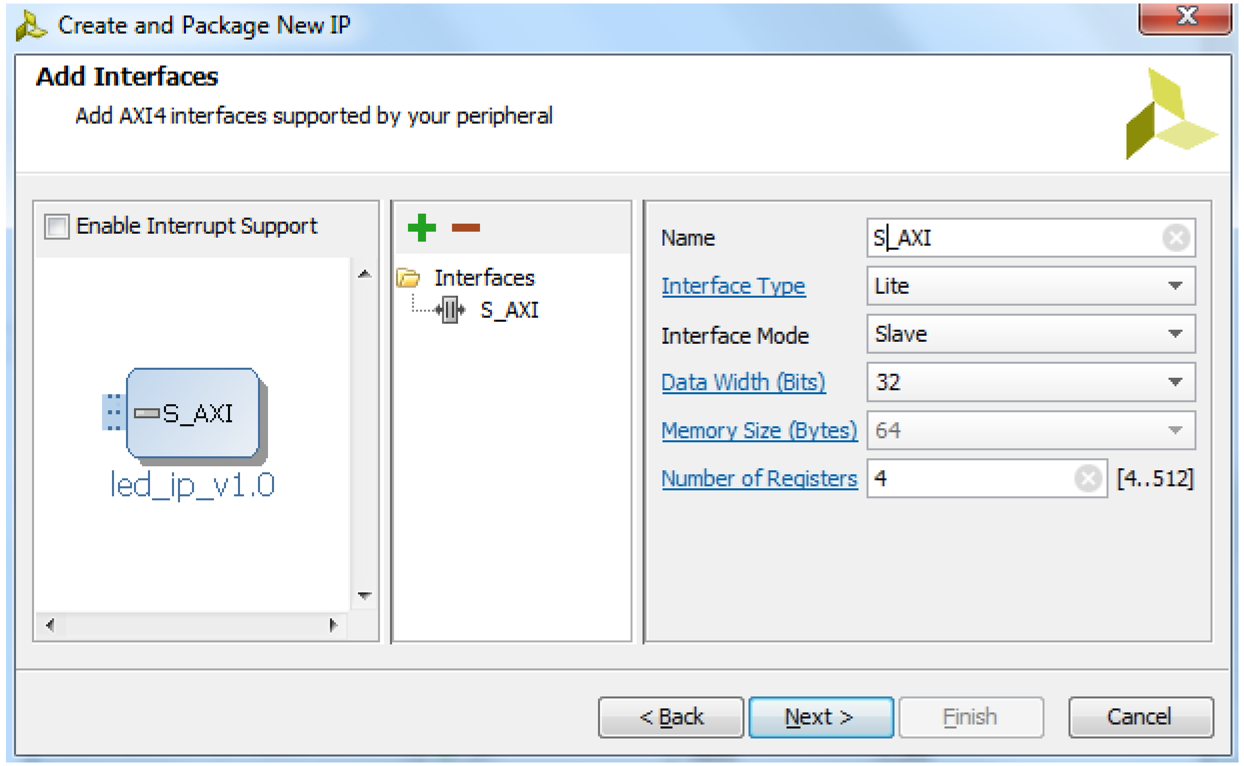
Create and Package New IP

Peripheral Details

Specify name, version and description for the new peripheral

Name:	led_ip
Version:	1.0
Display name:	led_ip_v1.0
Description:	My new AXI LED IP
IP location:	E:/Projects_ZedBoard/lab_custom_ip/led_ip/ip_repo

☐ Overwrite existing



Create and Package New IP

Add Interfaces

Add AXI4 interfaces supported by your peripheral

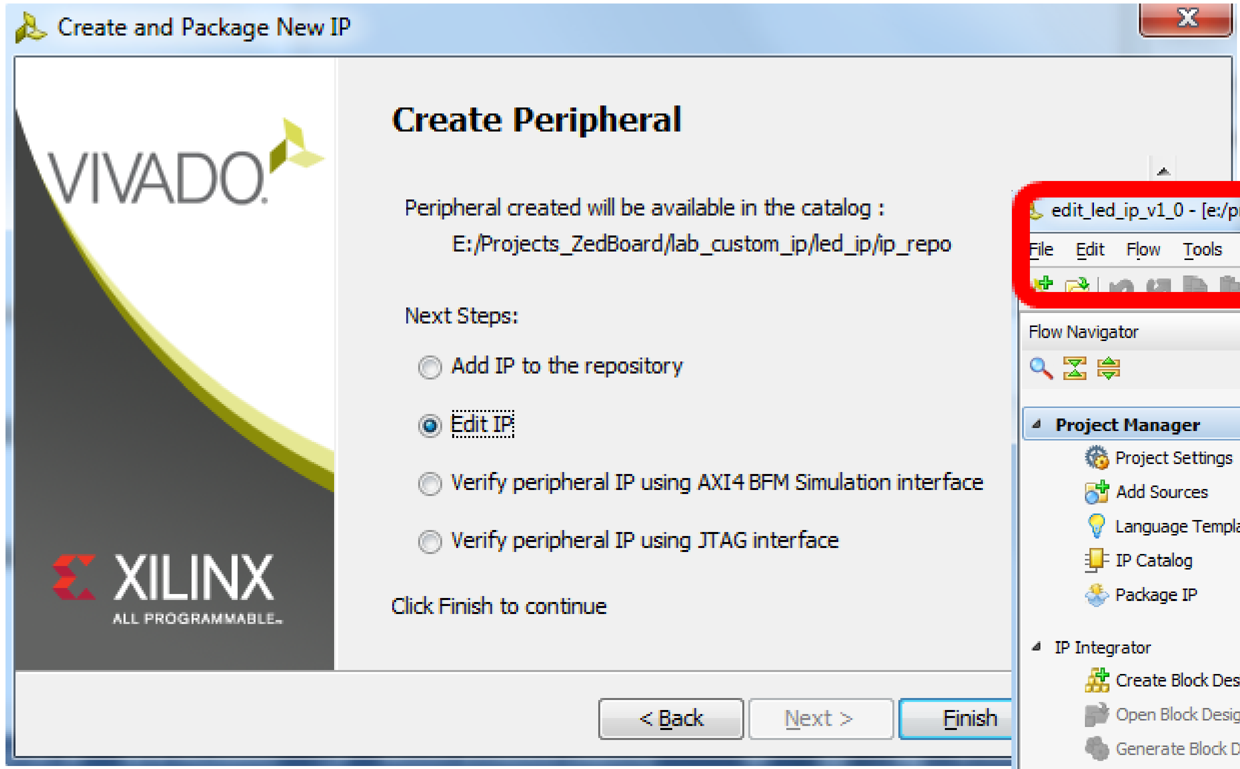
☐ Enable Interrupt Support

Interfaces
S_AXI

S_AXI
led_ip_v1.0

Name	S_AXI
Interface Type	Lite
Interface Mode	Slave
Data Width (Bits)	32
Memory Size (Bytes)	64
Number of Registers	4 [4..512]

Edit Created Custom AXI4 IP



Create and Package New IP

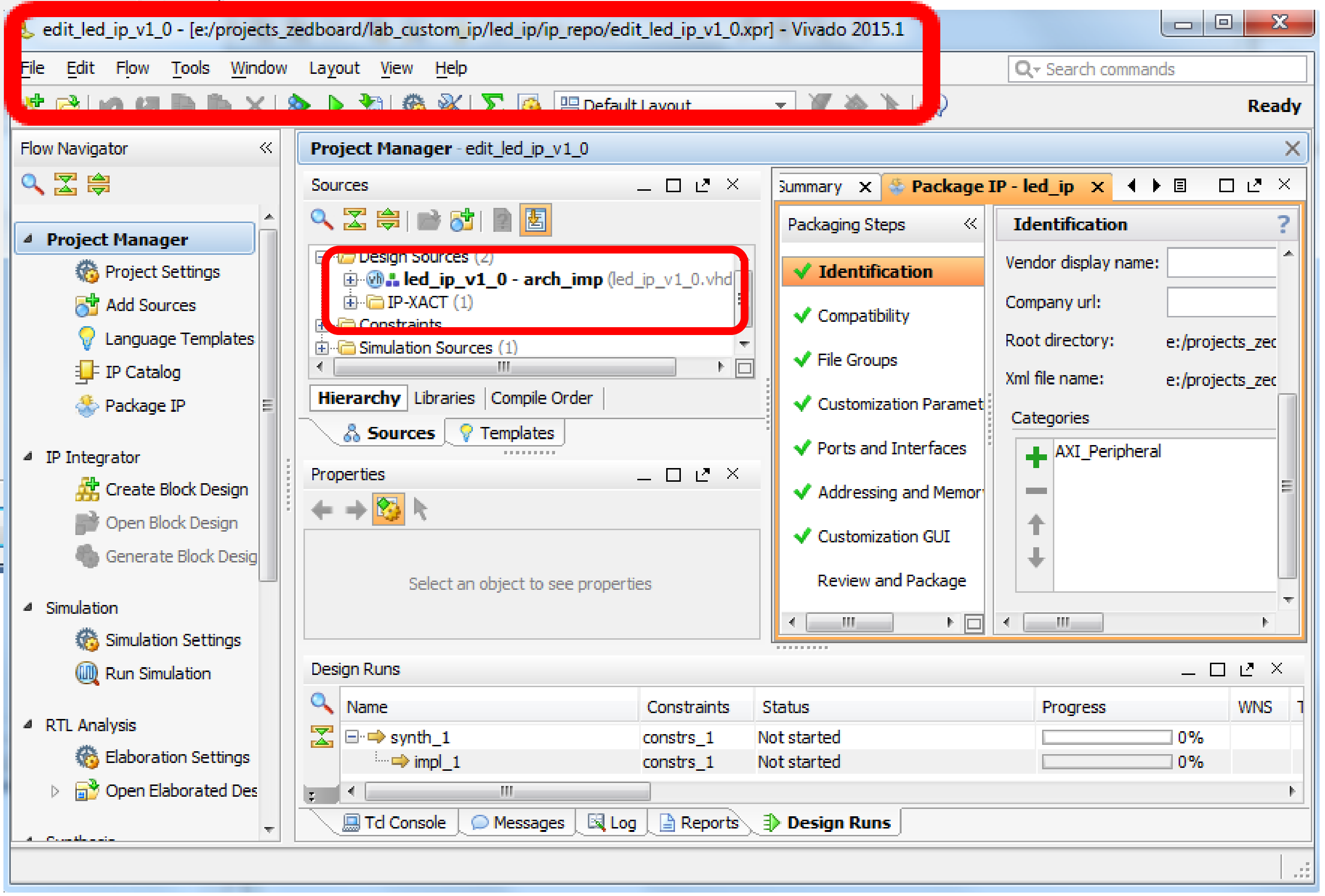
Create Peripheral

Peripheral created will be available in the catalog :
E:/Projects_ZedBoard/lab_custom_ip/led_ip/ip_repo

Next Steps:

- Add IP to the repository
- [✓ Edit IP]
- Verify peripheral IP using AXI4 BFM Simulation interface
- Verify peripheral IP using JTAG interface

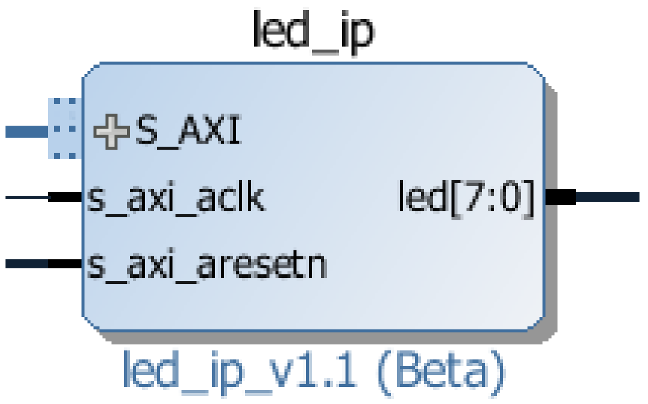
Click Finish to continue

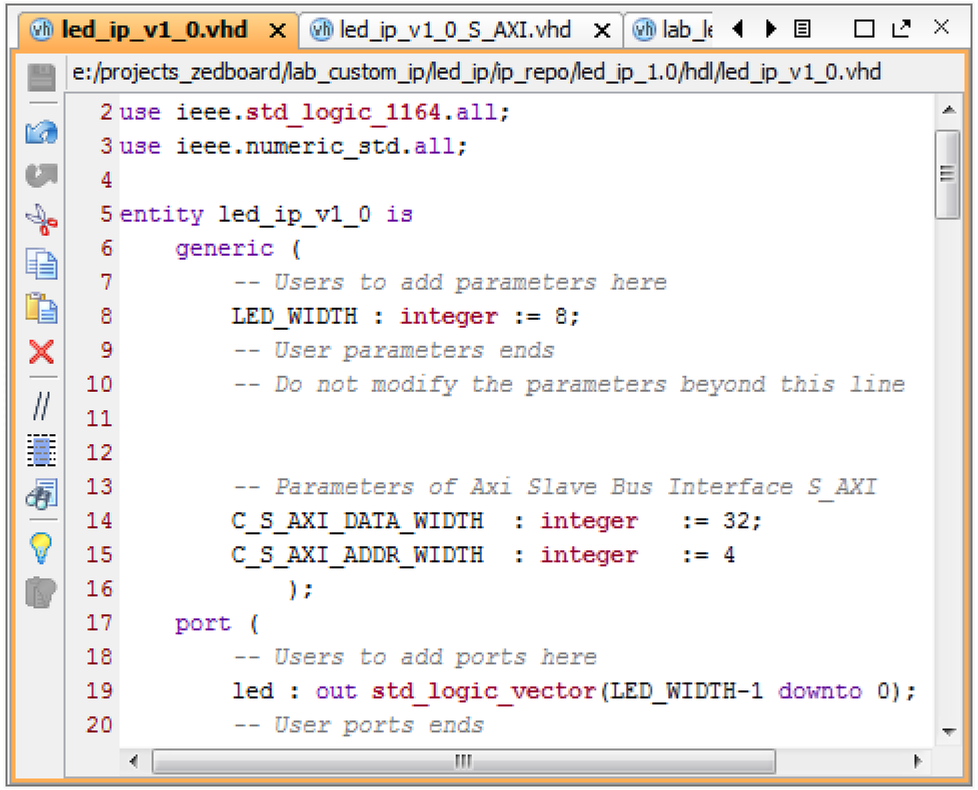


edit_led_ip_v1_0 - [e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/edit_led_ip_v1_0.xpr] - Vivado 2015.1

Name	Constraints	Status	Progress	WNS
synth_1	constrs_1	Not started	0%	
impl_1	constrs_1	Not started	0%	

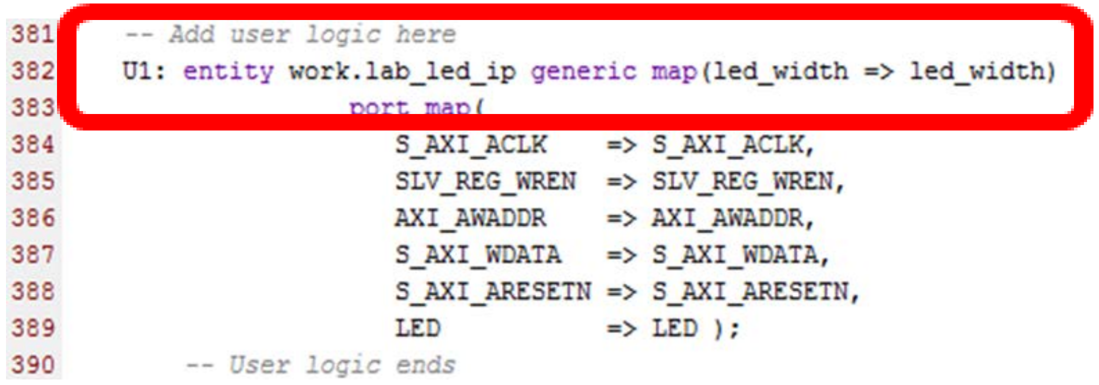
Edit Created Custom AXI4 IP





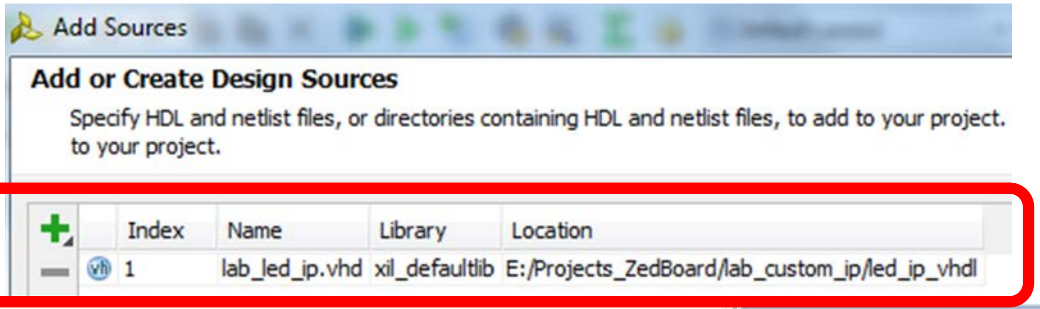
e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0/hdl/led_ip_v1_0.vhd

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity led_ip_v1_0 is
    generic (
        -- Users to add parameters here
        LED_WIDTH : integer := 8;
        -- User parameters ends
        -- Do not modify the parameters beyond this line


        -- Parameters of Axi Slave Bus Interface S_AXI
        C_S_AXI_DATA_WIDTH  : integer   := 32;
        C_S_AXI_ADDR_WIDTH  : integer   := 4
            );
    port (
        -- Users to add ports here
        led : out std_logic_vector(LED_WIDTH-1 downto 0);
        -- User ports ends
```



```
    -- Add user logic here
    U1: entity work.lab_led_ip generic map(led_width => led_width)
                port map(
                        S_AXI_ACLK    => S_AXI_ACLK,
                        SLV_REG_WREN  => SLV_REG_WREN,
                        AXI_AWADDR    => AXI_AWADDR,
                        S_AXI_WDATA   => S_AXI_WDATA,
                        S_AXI_ARESETN => S_AXI_ARESETN,
                        LED           => LED );
    -- User logic ends
```

Hierarchy of My IP

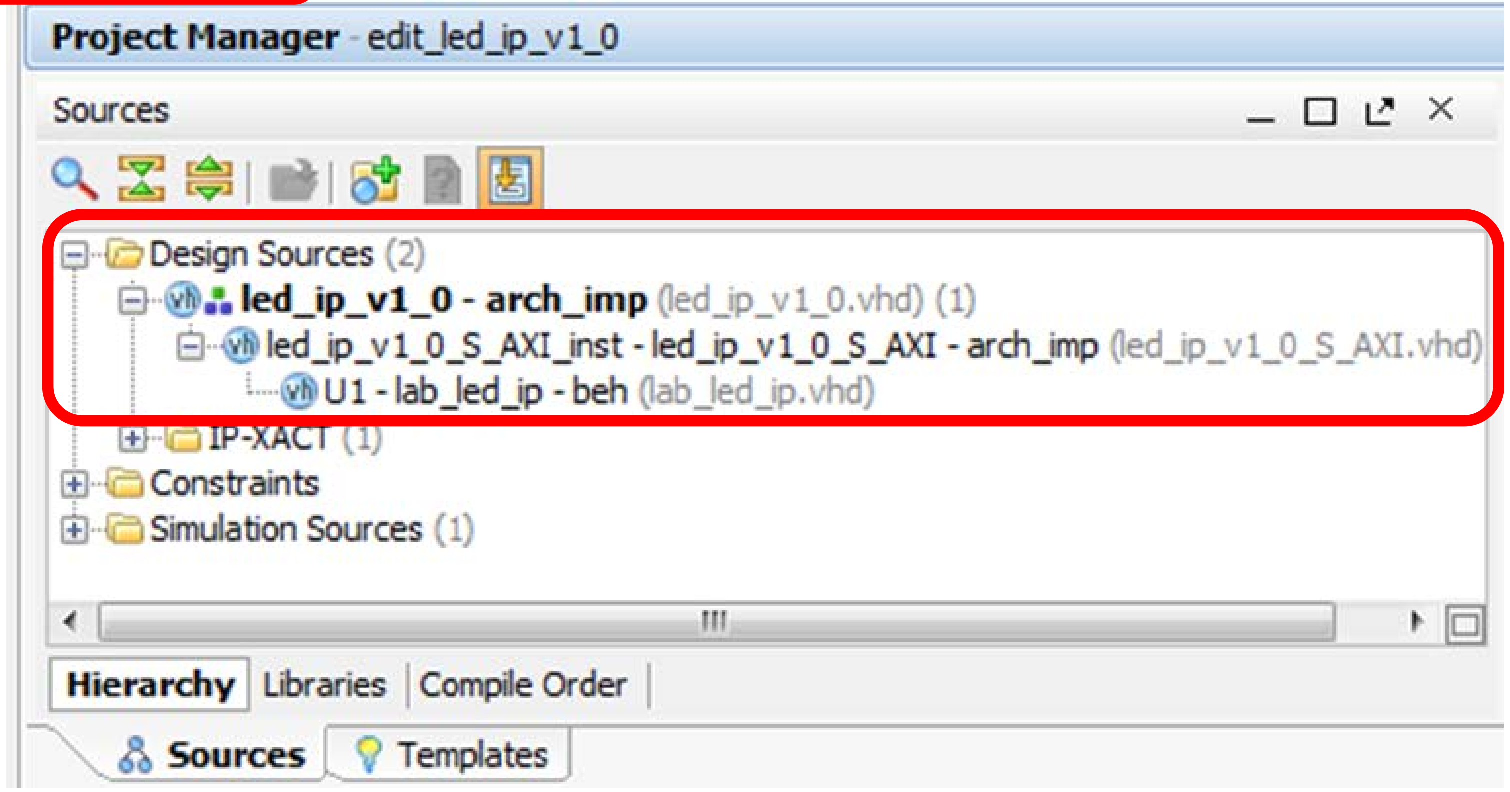


Add Sources

Add or Create Design Sources

Specify HDL and netlist files, or directories containing HDL and netlist files, to add to your project.
to your project.

Index	Name	Library	Location
1	lab_led_ip.vhd	xil_defaultlib	E:/Projects_ZedBoard/lab_custom_ip/led_ip_vhdl

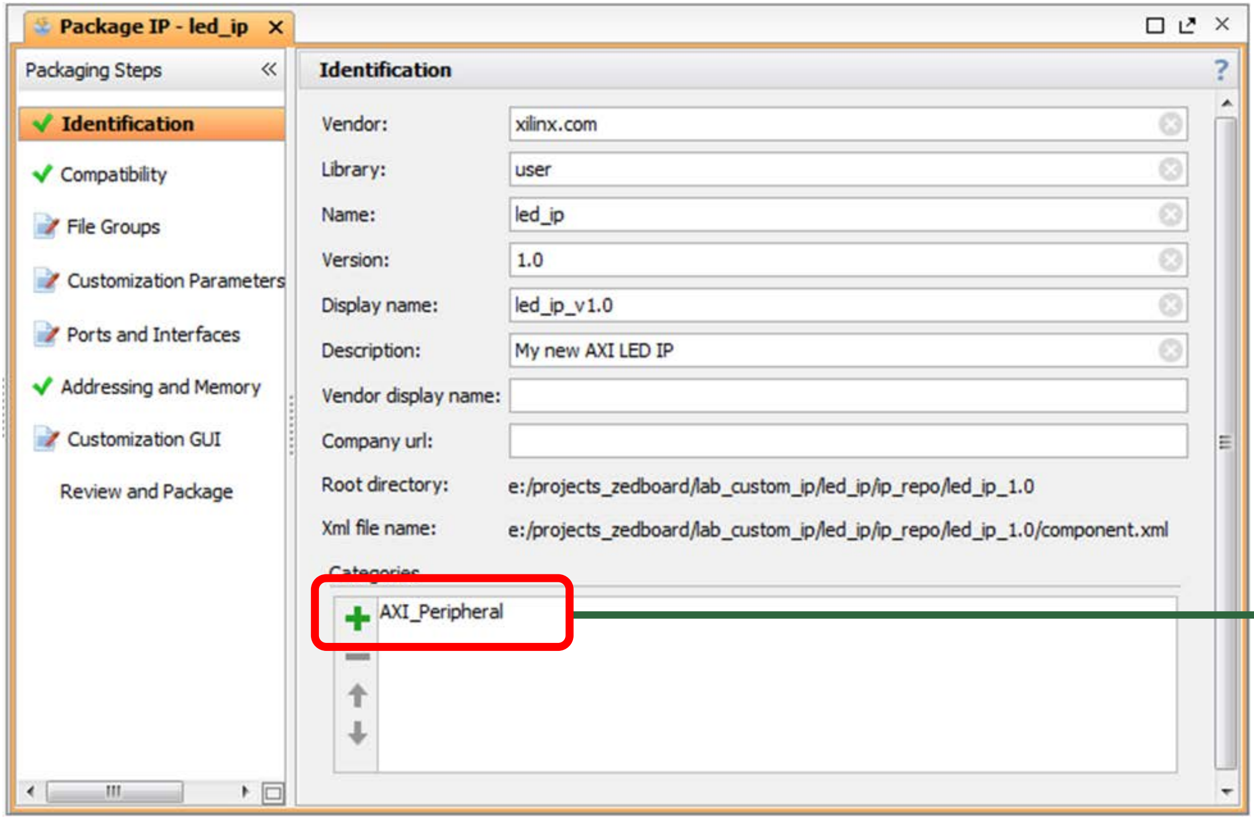


Project Manager - edit_led_ip_v1_0

Sources

```
Design Sources (2)
  led_ip_v1_0 - arch_imp (led_ip_v1_0.vhd) (1)
    led_ip_v1_0_S_AXI_inst - led_ip_v1_0_S_AXI - arch_imp (led_ip_v1_0_S_AXI.vhd)
      U1 - lab_led_ip - beh (lab_led_ip.vhd)
  IP-XACT (1)
Constraints
Simulation Sources (1)
```

Package the IP



Package IP - led_ip

Packaging Steps

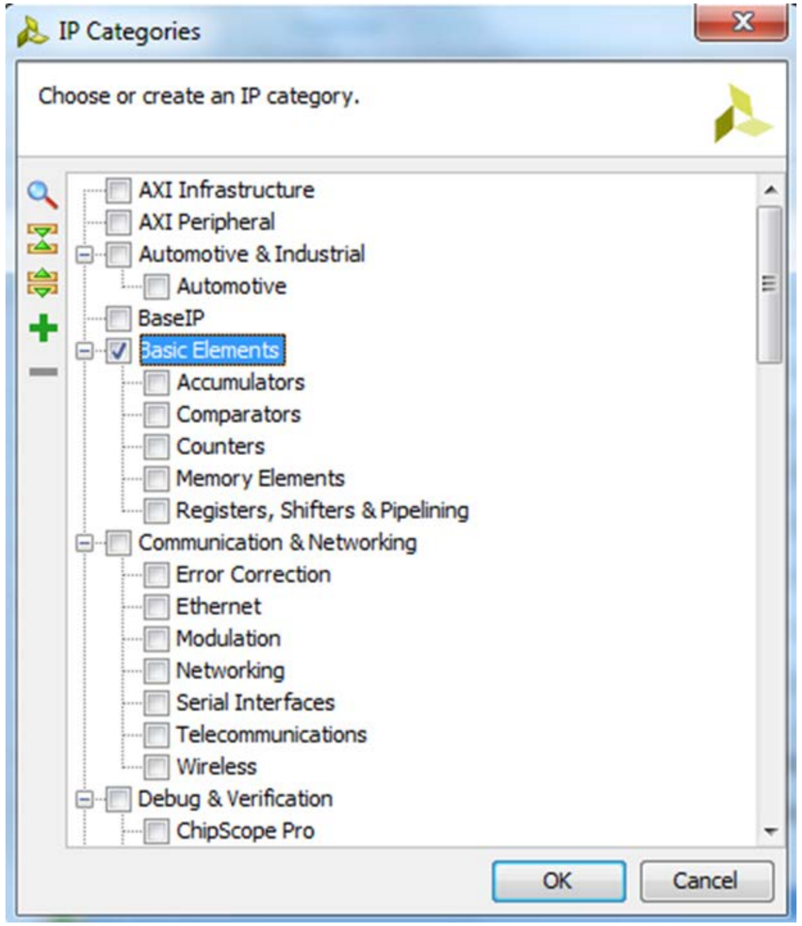
- ✓ Identification
- ✓ Compatibility
- File Groups
- Customization Parameters
- Ports and Interfaces
- ✓ Addressing and Memory
- Customization GUI
- Review and Package

Identification

Vendor:	xilinx.com
Library:	user
Name:	led_ip
Version:	1.0
Display name:	led_ip_v1.0
Description:	My new AXI LED IP
Vendor display name:	
Company url:	
Root directory:	e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0
Xml file name:	e:/projects_zedboard/lab_custom_ip/led_ip/ip_repo/led_ip_1.0/component.xml

Categories

AXI_Peripheral

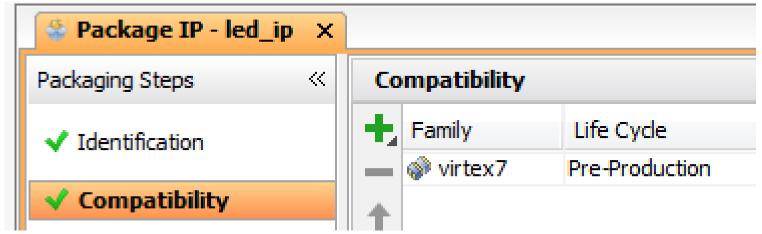


IP Categories

Choose or create an IP category.

- AXI Infrastructure
- AXI Peripheral
- Automotive & Industrial
  - Automotive
- BaseIP
- ✓ Basic Elements
  - Accumulators
  - Comparators
  - Counters
  - Memory Elements
  - Registers, Shifters & Pipelining
- Communication & Networking
  - Error Correction
  - Ethernet
  - Modulation
  - Networking
  - Serial Interfaces
  - Telecommunications
  - Wireless
- Debug & Verification
  - ChipScope Pro

Compatibility of My IP



Package IP - led_ip

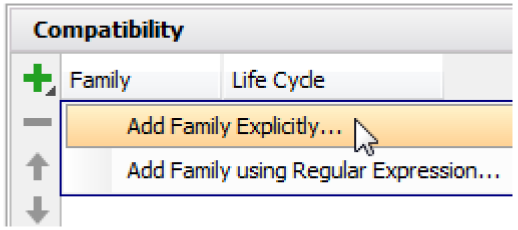
Packaging Steps

✓ Identification

✓ Compatibility

Compatibility

Family	Life Cycle
virtex7	Pre-Production

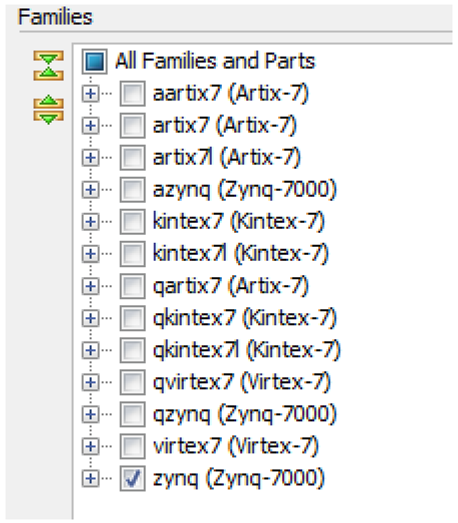


Compatibility

Family	Life Cycle

Add Family Explicitly…

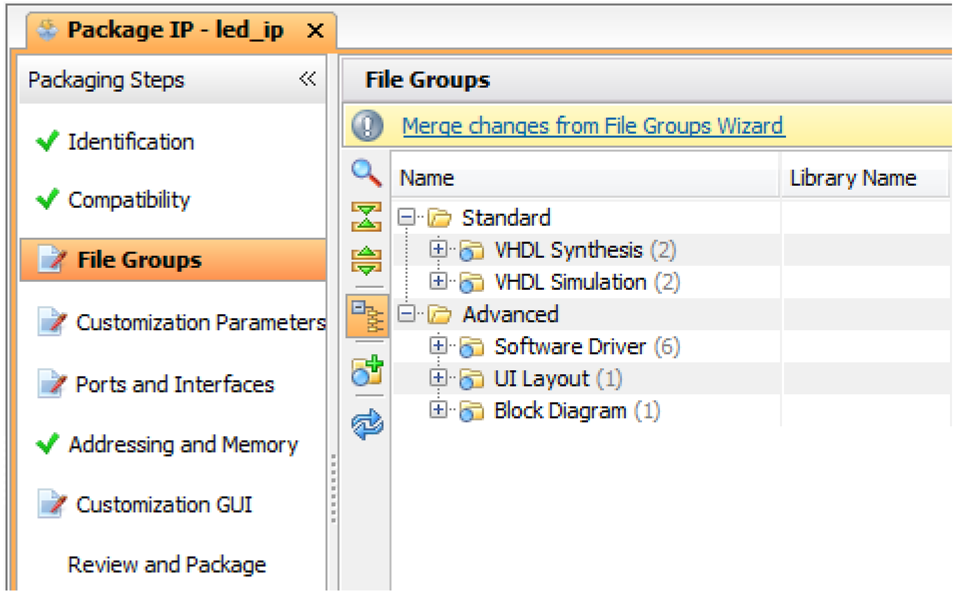
Add Family using Regular Expression…



Families

- All Families and Parts
  - [ ] aartix7 (Artix-7)
  - [ ] artix7 (Artix-7)
  - [ ] artix7l (Artix-7)
  - [ ] azynq (Zynq-7000)
  - [ ] kintex7 (Kintex-7)
  - [ ] kintex7l (Kintex-7)
  - [ ] qartix7 (Artix-7)
  - [ ] qkintex7 (Kintex-7)
  - [ ] qkintex7l (Kintex-7)
  - [ ] qvirtex7 (Virtex-7)
  - [ ] qzynq (Zynq-7000)
  - [ ] virtex7 (Virtex-7)
  - [✓] zynq (Zynq-7000)

Updating Generated Files



Package IP - led_ip

Packaging Steps

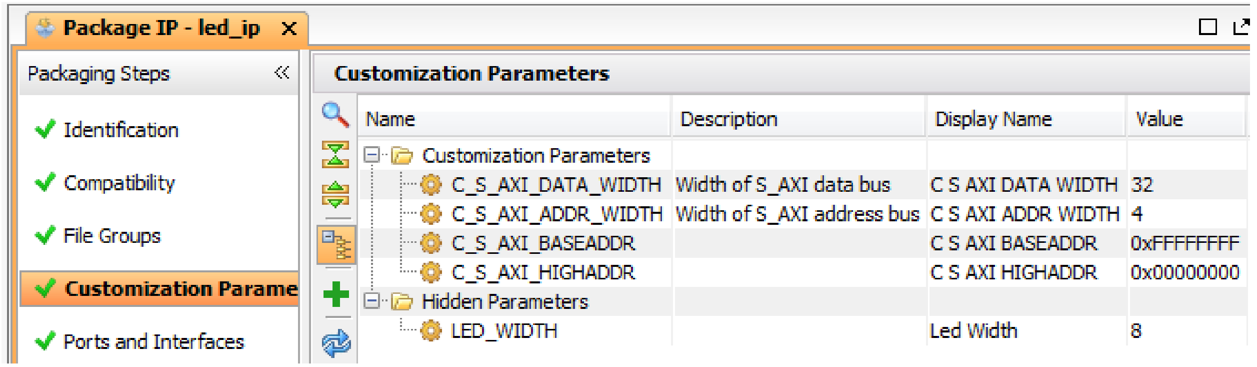
- ✓ Identification
- ✓ Compatibility
- File Groups
- Customization Parameters
- Ports and Interfaces
- ✓ Addressing and Memory
- Customization GUI
- Review and Package

File Groups

Merge changes from File Groups Wizard

Name	Library Name
Standard	
VHDL Synthesis (2)	
VHDL Simulation (2)	
Advanced	
Software Driver (6)	
UI Layout (1)	
Block Diagram (1)	

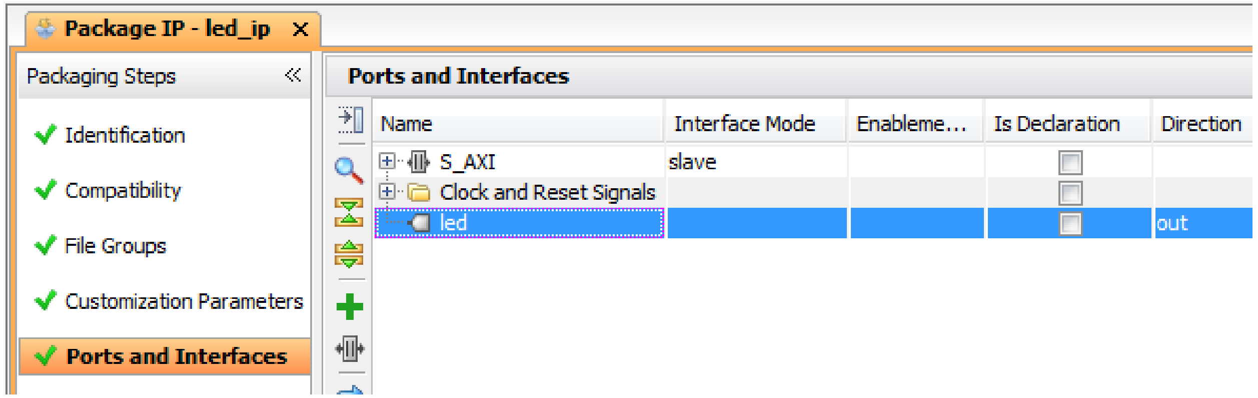
Checking Parameters and I/O Ports


Package IP - led_ip

Packaging Steps
- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ **Customization Parame**
- ✓ Ports and Interfaces

Customization Parameters

Name	Description	Display Name	Value
Customization Parameters			
C_S_AXI_DATA_WIDTH	Width of S_AXI data bus	C S AXI DATA WIDTH	32
C_S_AXI_ADDR_WIDTH	Width of S_AXI address bus	C S AXI ADDR WIDTH	4
C_S_AXI_BASEADDR		C S AXI BASEADDR	0xFFFFFFFF
C_S_AXI_HIGHADDR		C S AXI HIGHADDR	0x00000000
Hidden Parameters			
LED_WIDTH		Led Width	8


Package IP - led_ip

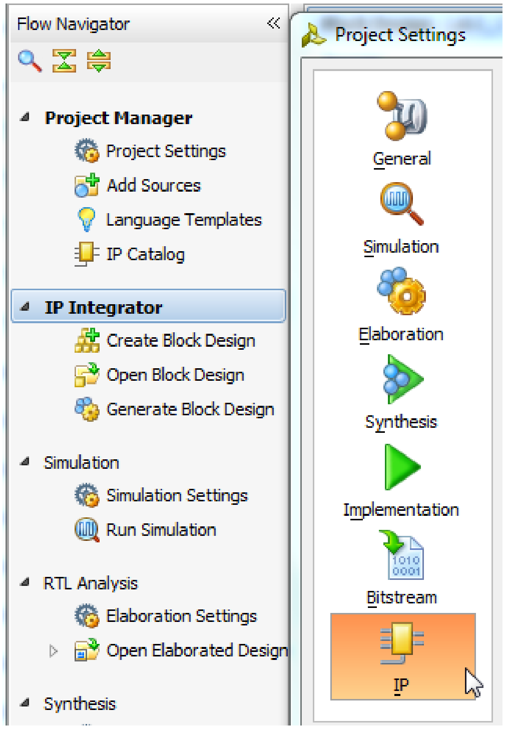
Packaging Steps
- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- ✓ Customization Parameters
- ✓ **Ports and Interfaces**

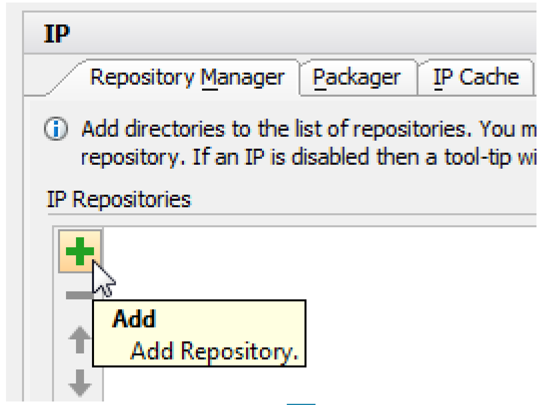
Ports and Interfaces

Name	Interface Mode	Enableme...	Is Declaration	Direction
S_AXI	slave		☐	
Clock and Reset Signals			☐	
led			☐	out

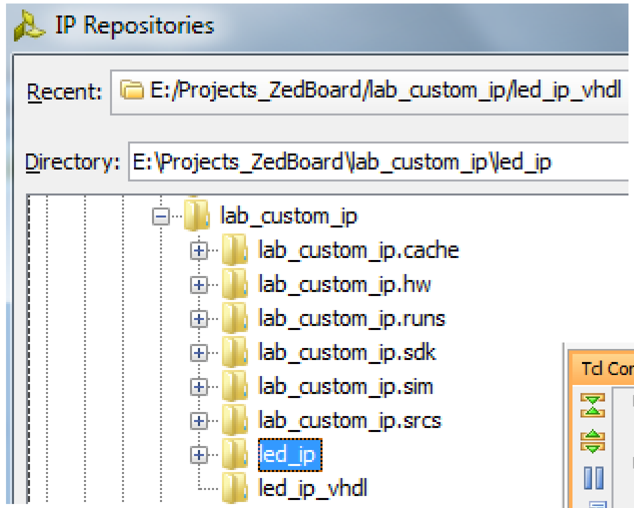
(This ends the Works on the edit_ip environment)

Add My IP to the Repository



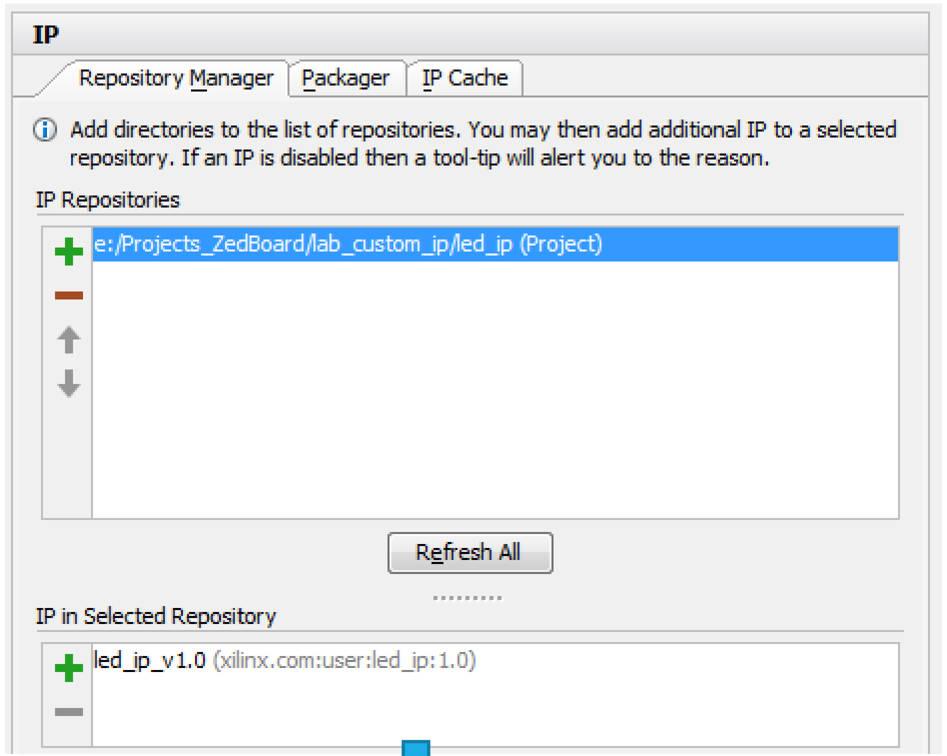


Add directories to the list of repositories. You m
repository. If an IP is disabled then a tool-tip wi



Recent: E:/Projects_ZedBoard/lab_custom_ip/led_ip_vhdl

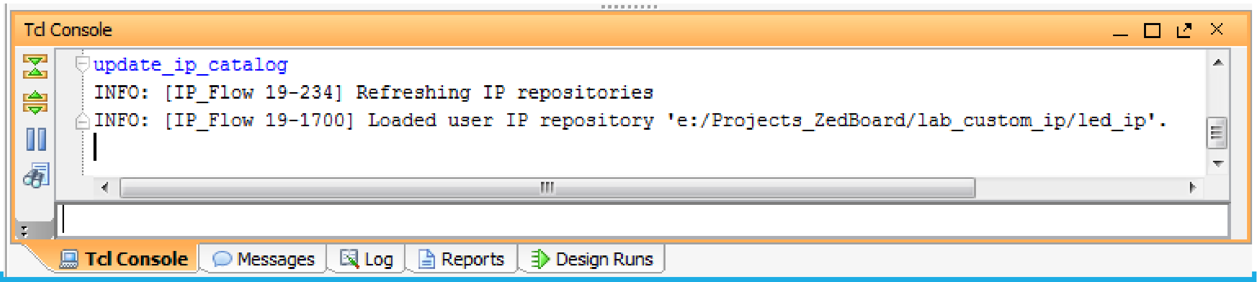
Directory: E:\Projects_ZedBoard\lab_custom_ip\led_ip



Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

e:/Projects_ZedBoard/lab_custom_ip/led_ip (Project)

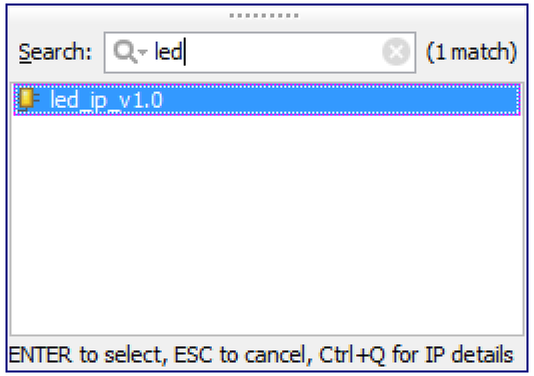
led_ip_v1.0 (xilinx.com:user:led_ip:1.0)



```
update_ip_catalog
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1700] Loaded user IP repository 'e:/Projects_ZedBoard/lab_custom_ip/led_ip'.
```

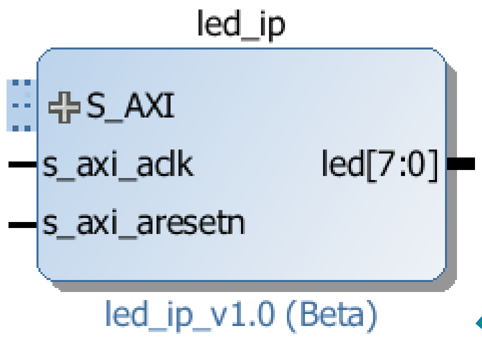
These steps shold be done in the Vivado Environment

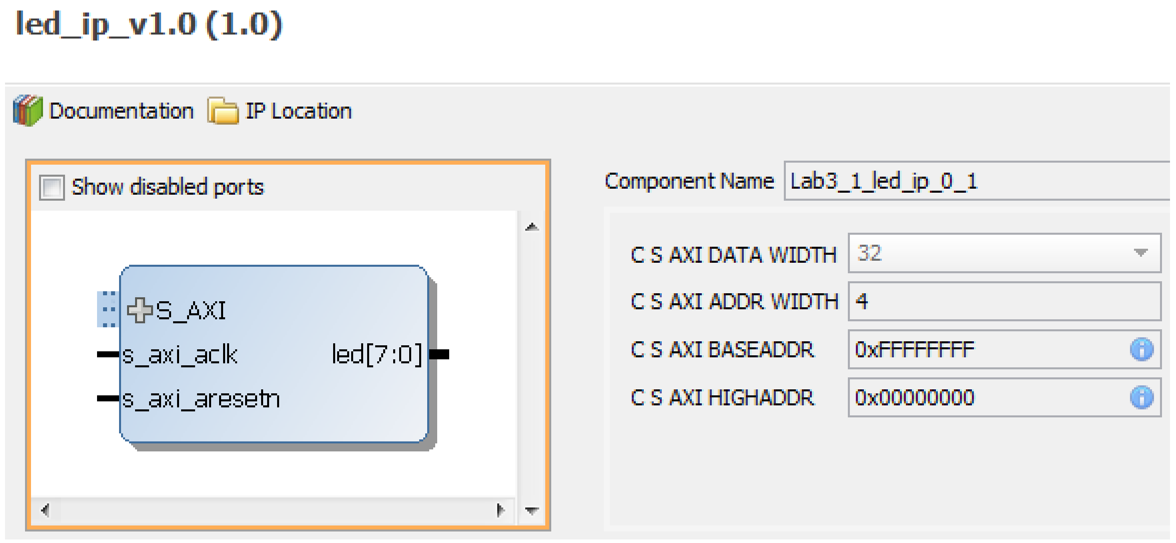
led_ip Now Available in the IP List


Search: led (1 match)

led_ip_v1.0

ENTER to select, ESC to cancel, Ctrl+Q for IP details




led_ip_v1.0 (1.0)

Documentation IP Location

Show disabled ports

Component Name: Lab3_1_led_ip_0_1

C S AXI DATA WIDTH: 32

C S AXI ADDR WIDTH: 4

C S AXI BASEADDR: 0xFFFFFFFF

C S AXI HIGHADDR: 0x00000000

Files created

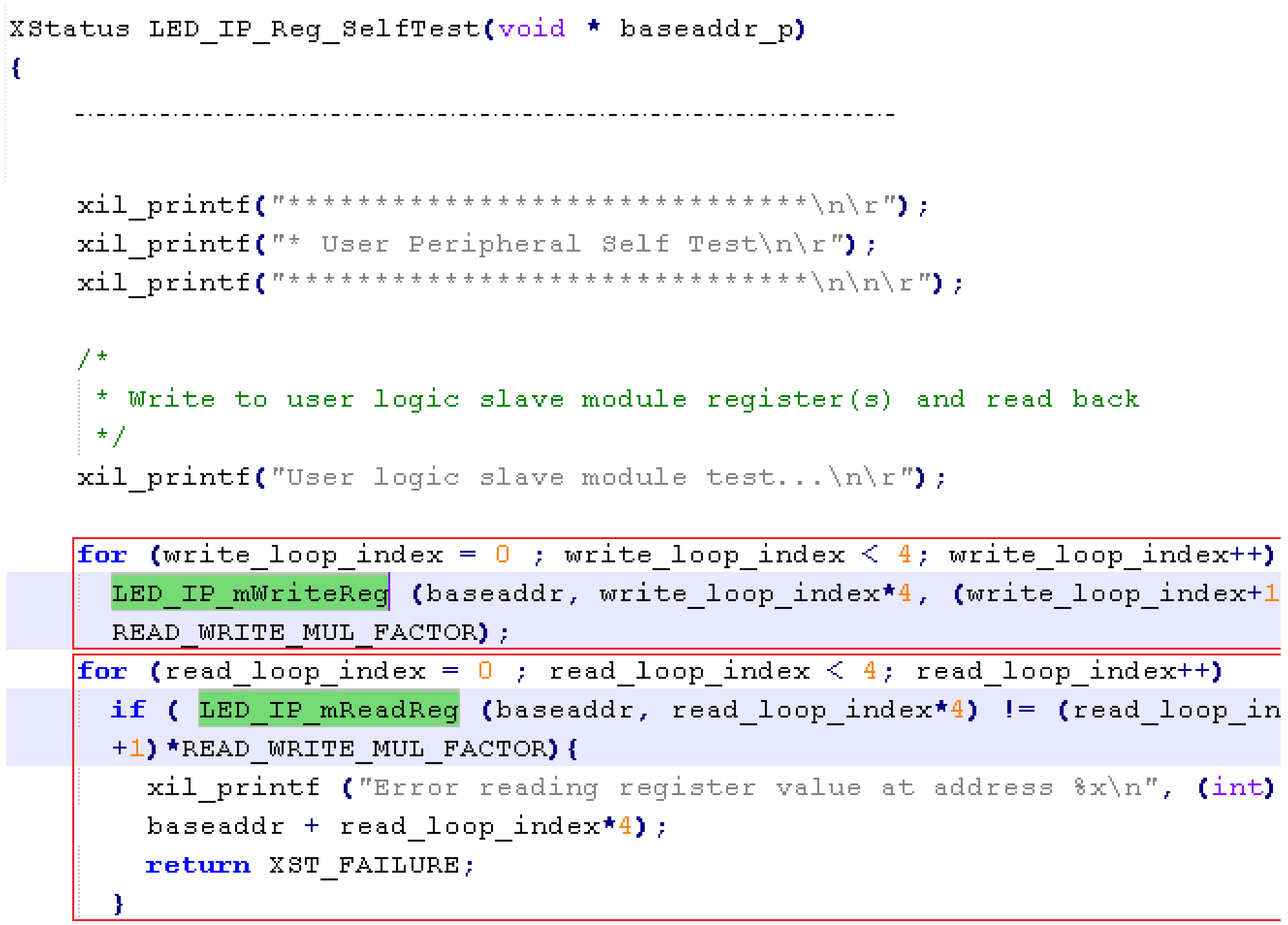
component.xml
- IP XACT description

.bd
- Block Diagram tcl file

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

hdl
- Verilog/VHDL source

xgui
- GUI tcl file



```
XStatus LED_IP_Reg_SelfTest(void * baseaddr_p)
{

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

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

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

Steps for Custom IP - Summary

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

Lab Custom IP

Lab Custom IP

Basic PWM Functionality

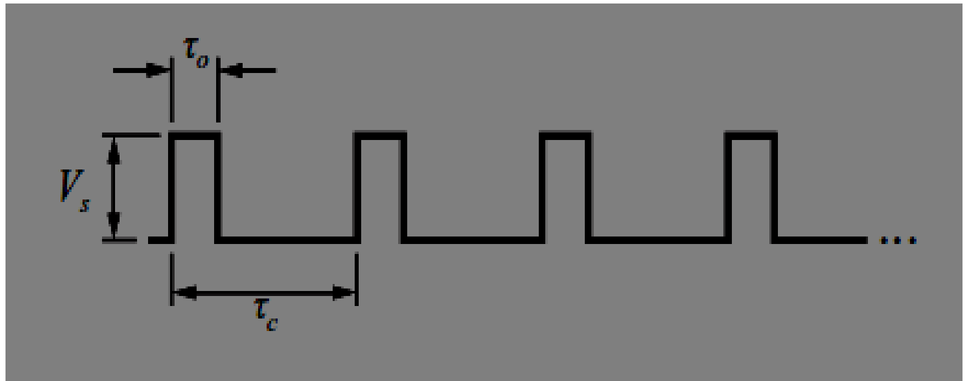
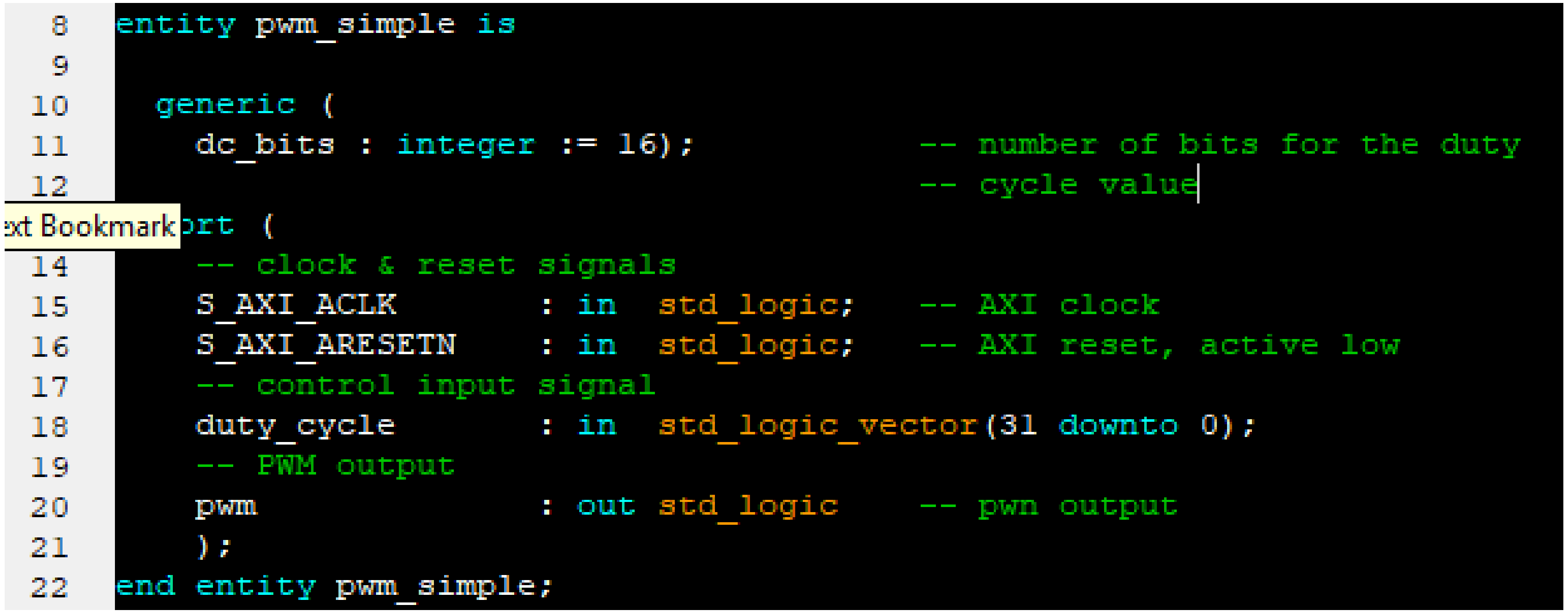


Figure 1: Nomenclature for definition of PWM duty cycle.

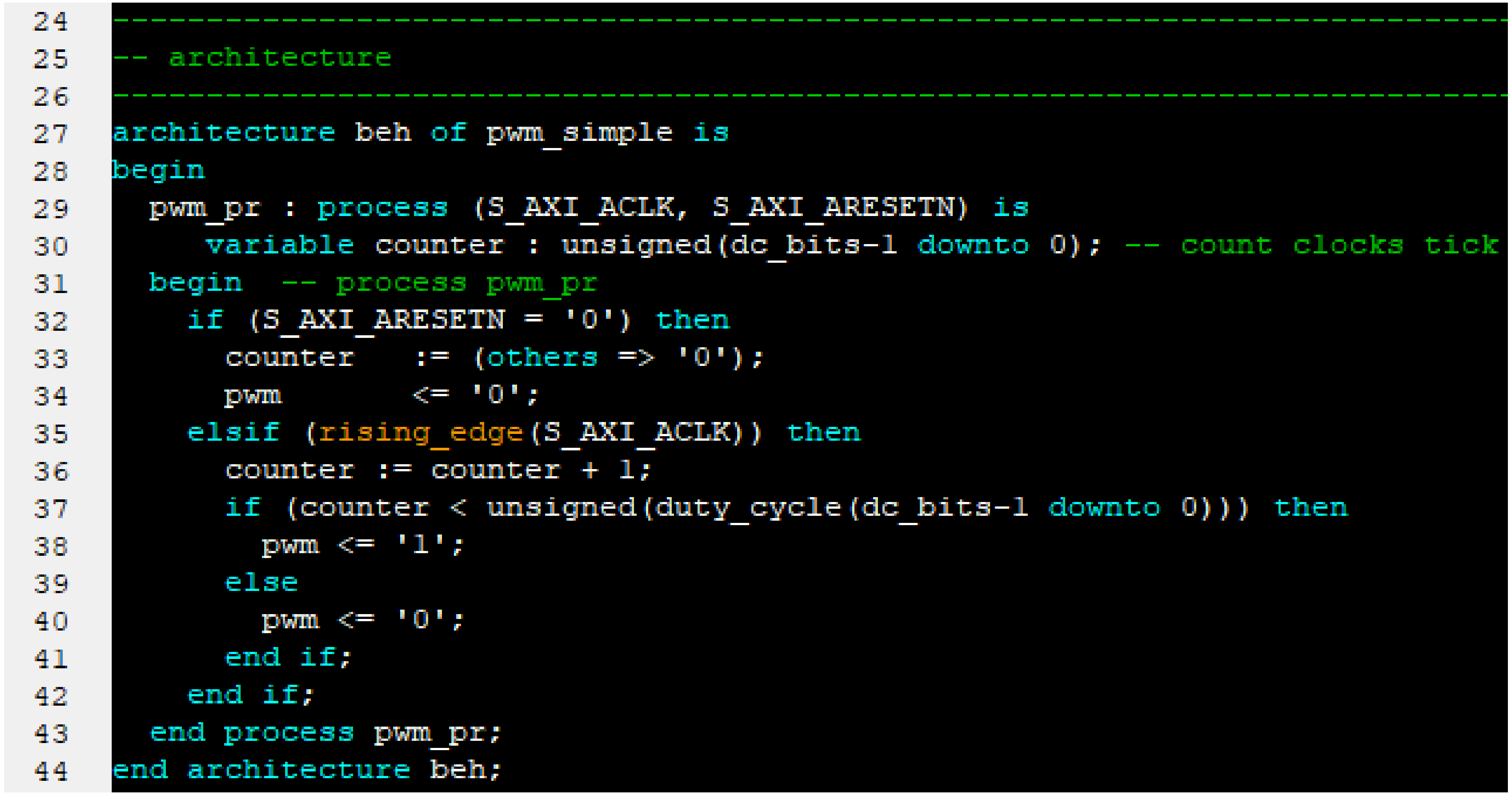
$$Veff = Vs\frac{\tau o}{\tau c} \quad (1)$$

VHDL Code for PWM Simple



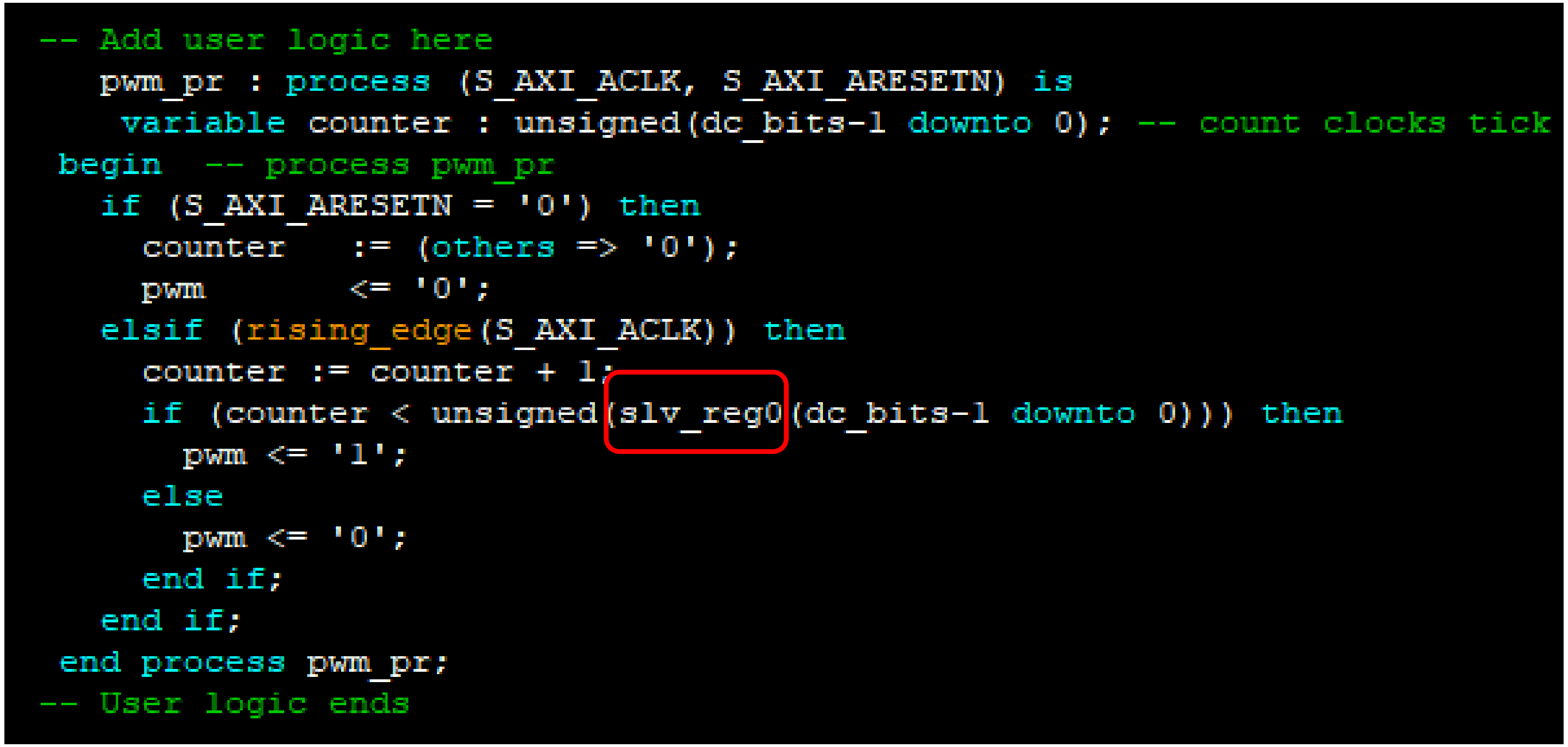
```vhdl
entity pwm_simple is

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

VHDL Code for PWM Simple



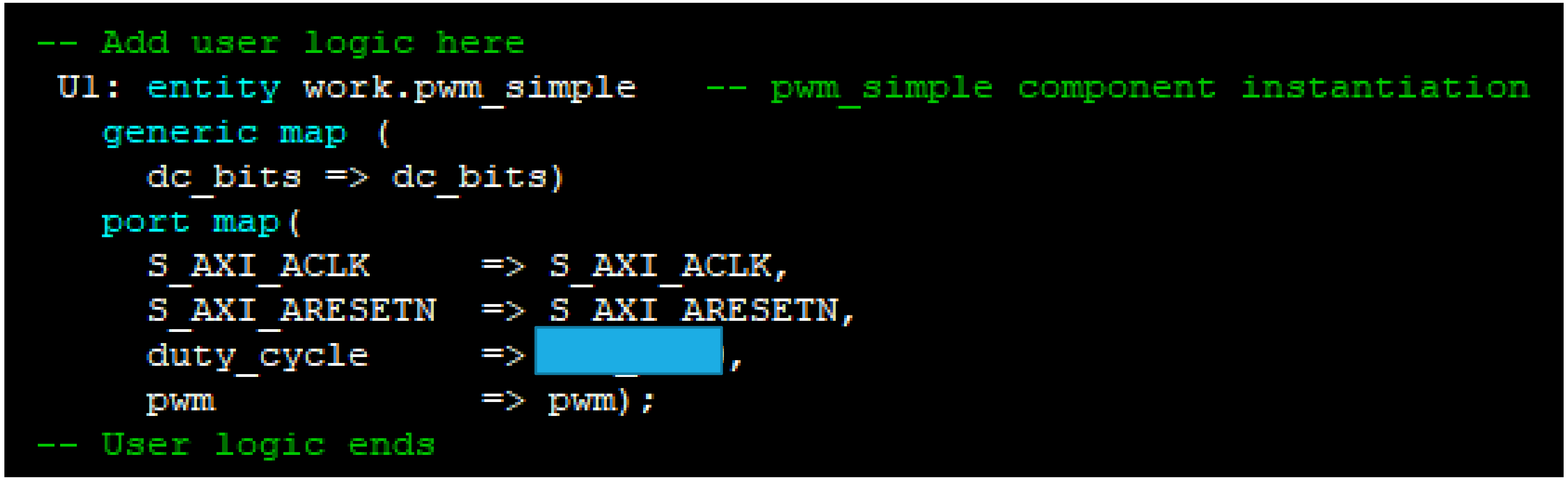
```vhdl
---------------------------------------------------------------------------
-- architecture
---------------------------------------------------------------------------
architecture beh of pwm_simple is
begin
  pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
     variable counter : unsigned(dc_bits-1 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;
```

PWM IP Core - Case 1



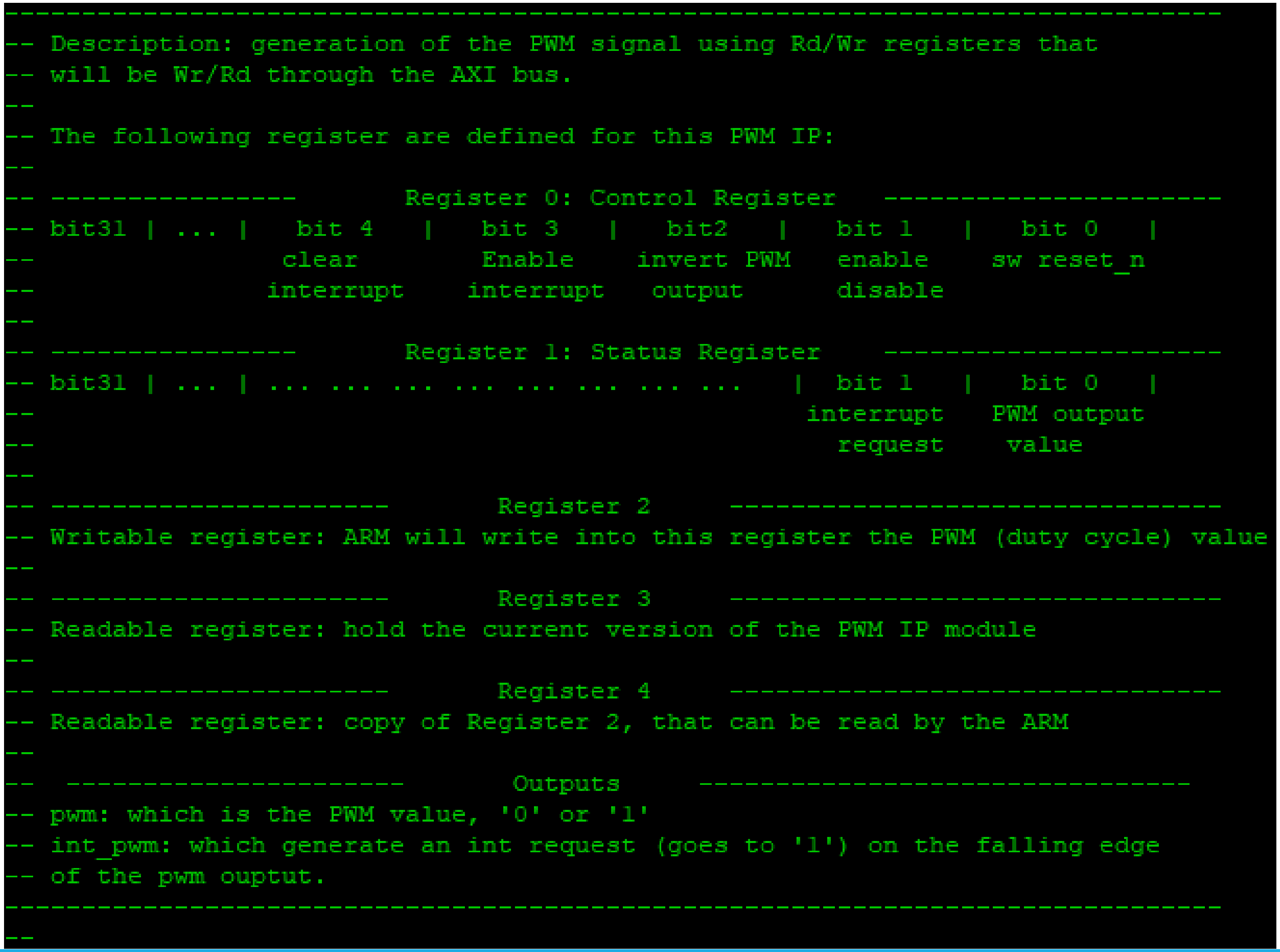
```vhdl
-- Add user logic here
   pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is
    variable counter : unsigned(dc_bits-1 downto 0); -- count clocks tick
 begin  -- process pwm_pr
   if (S_AXI_ARESETN = '0') then
     counter   := (others => '0');
     pwm       <= '0';
   elsif (rising_edge(S_AXI_ACLK)) 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



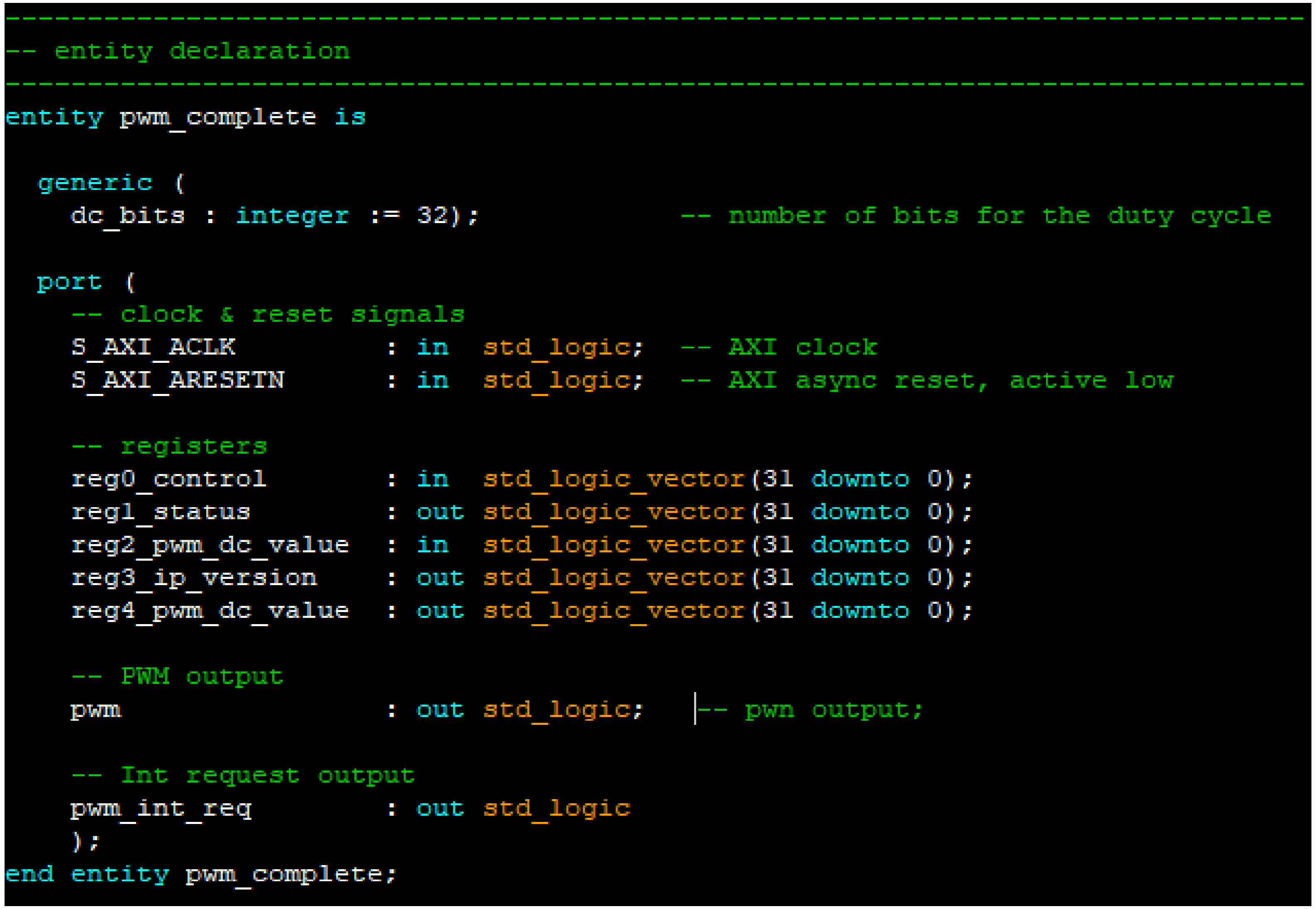
```
-- Add user logic here
 U1: entity work.pwm_simple    -- pwm_simple component instantiation
   generic map (
     dc_bits => dc_bits)
   port map(
     S_AXI_ACLK      => S_AXI_ACLK,
     S_AXI_ARESETN   => S_AXI_ARESETN,
     duty_cycle      => [redacted],
     pwm             => pwm);
-- User logic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

VHDL code for PWM Complete (1)



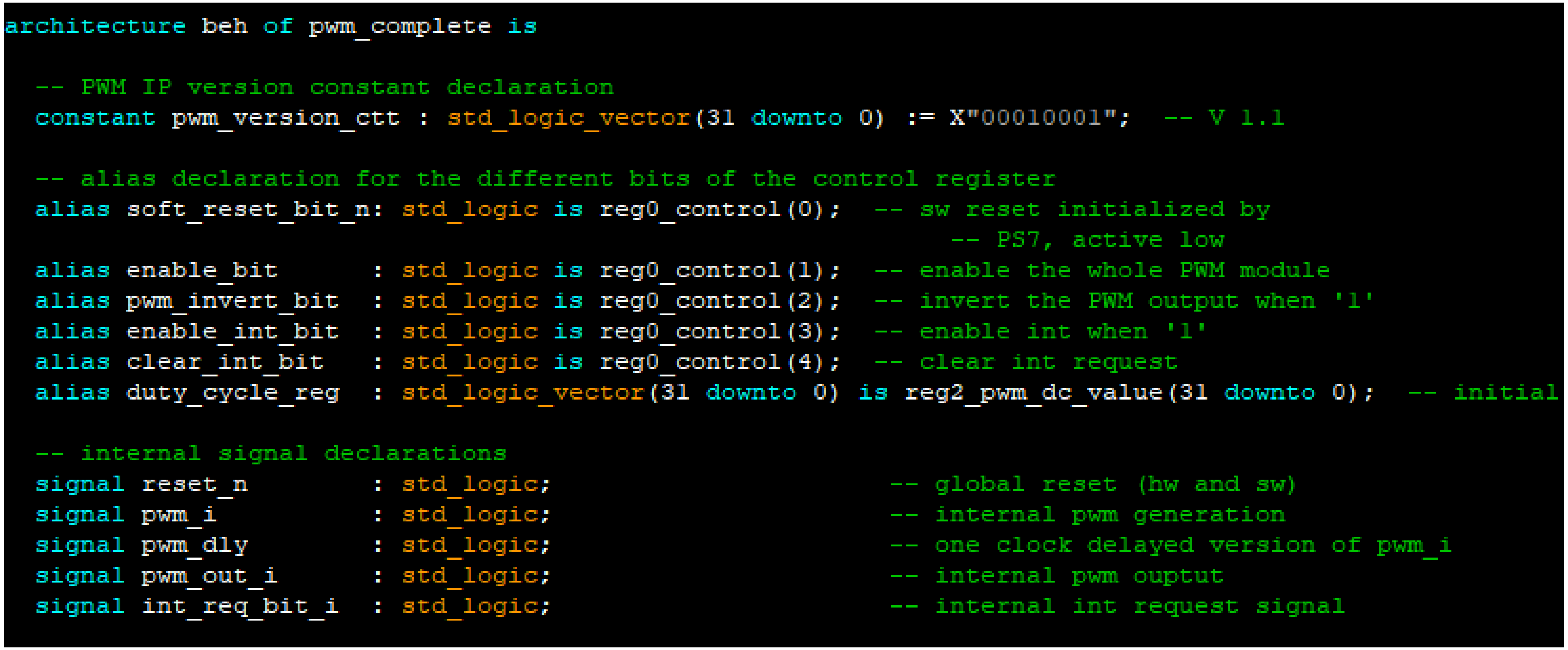
```
-------------------------------------------------------------------------------
-- Description: generation of the PWM signal using Rd/Wr registers that
-- will be Wr/Rd through the AXI bus.
--
-- The following register are defined for this PWM IP:
--
-- ----------------       Register 0: Control 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)



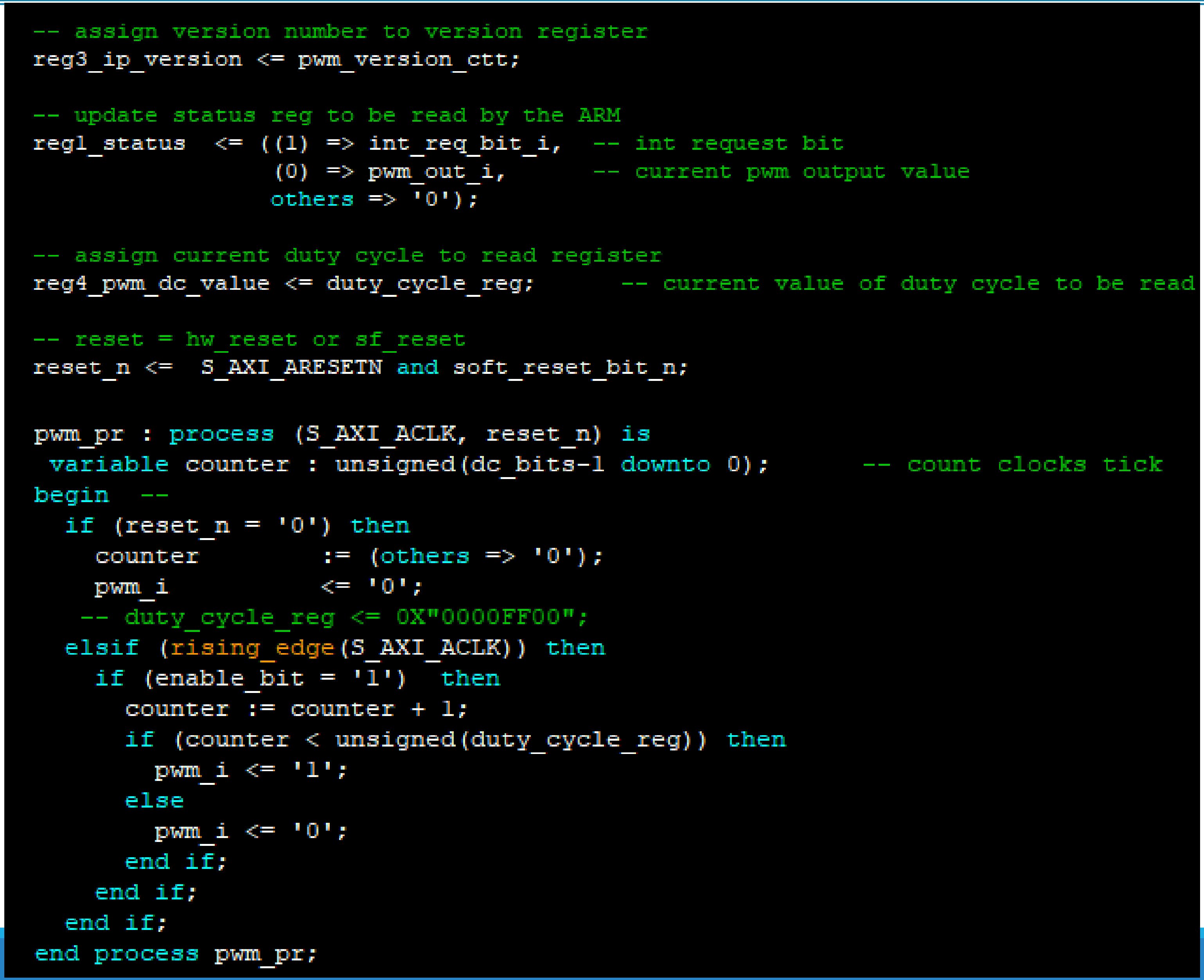
```
-------------------------------------------------------------------------------
-- entity declaration
-------------------------------------------------------------------------------
entity pwm_complete is

  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)



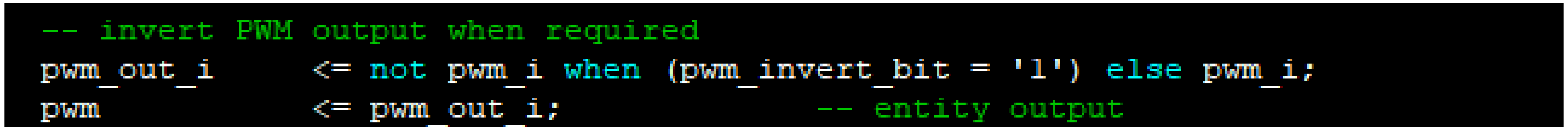
```vhdl
architecture beh of pwm_complete is

  -- 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)

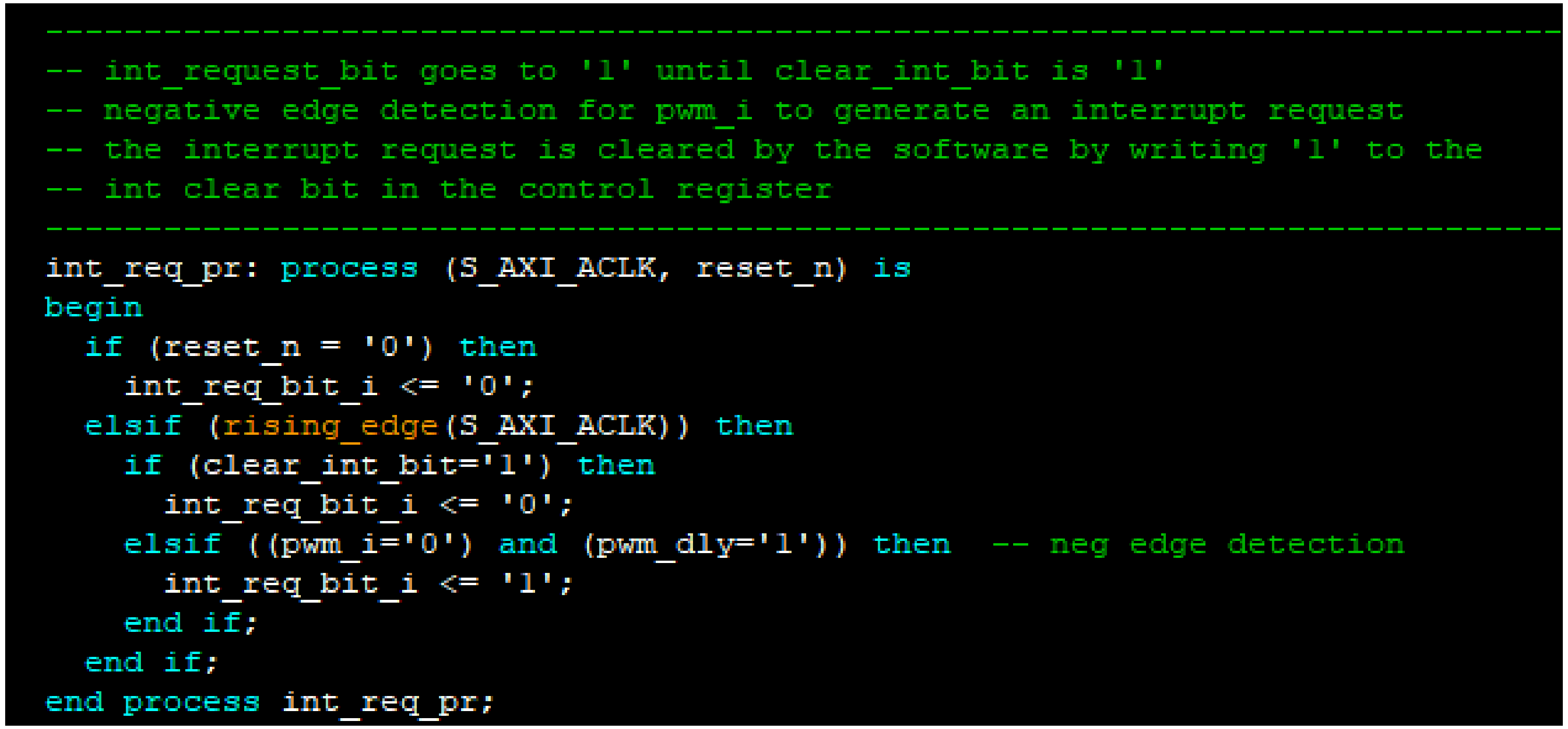


```vhdl
-- assign version number to version register
reg3_ip_version <= pwm_version_ctt;

-- 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)

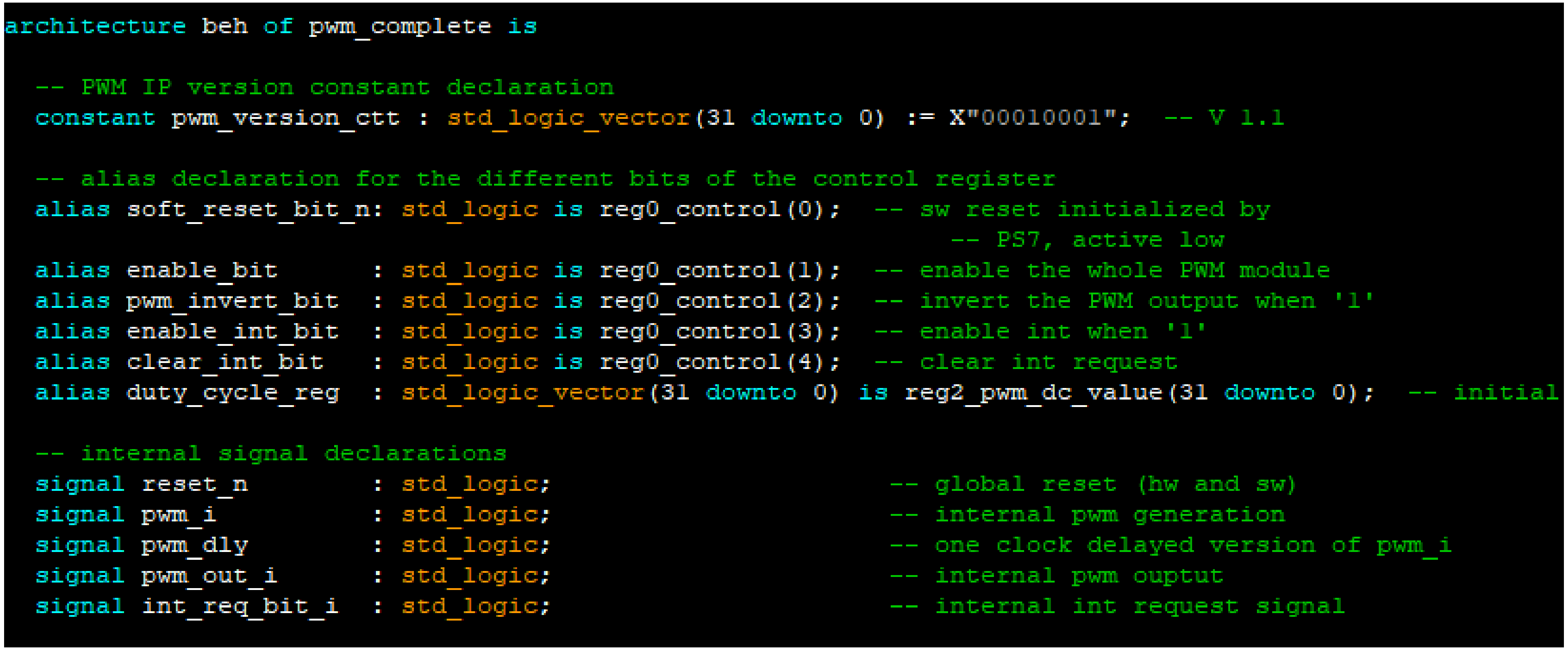


```
-- invert PWM output when required
pwm_out_i     <= not pwm_i when (pwm_invert_bit = '1') else pwm_i;
pwm           <= pwm_out_i;             -- entity output
```



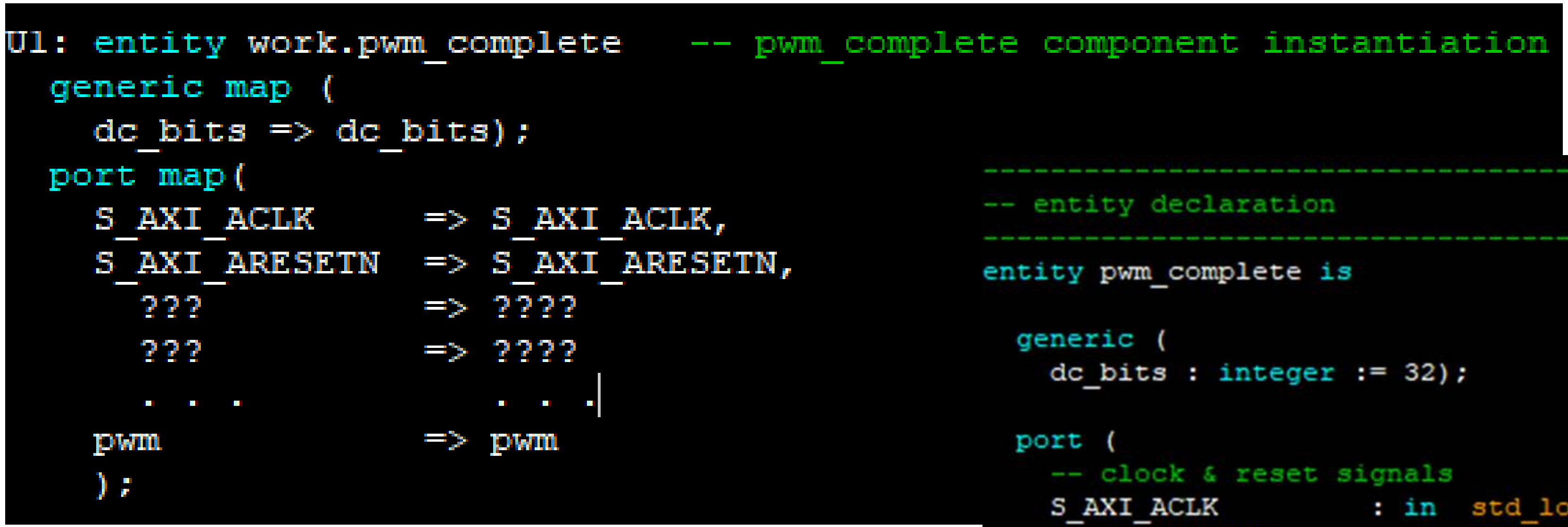
```
-----------------------------------------------------------------------------
-- int_request_bit goes to '1' until clear_int_bit is '1'
-- negative edge detection for pwm_i to generate an interrupt request
-- the interrupt request is cleared by the software by writing '1' to the
-- int clear bit 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)

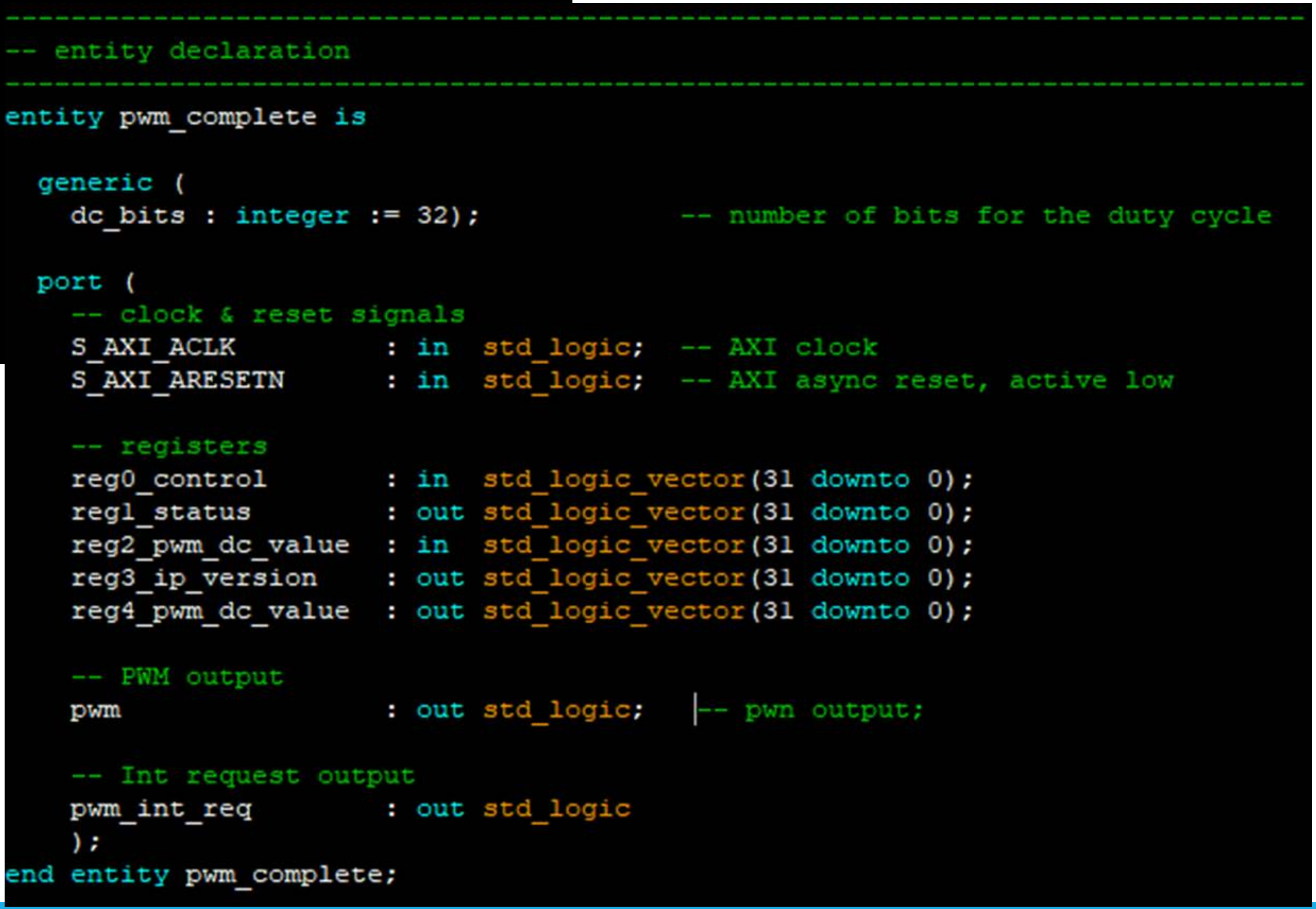


```
architecture beh of pwm_complete is

  -- 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



```
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
    );
```



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