

# Advanced Workshop on Modern FPGA- Based Technology for Scientific Computing



---

## LABORATORY

### *Custom AXI-Slave IP Core*

---

Prepared by  
Cristian Sisterna  
ICTP-MLAB

---

# Creating a Custom IP Core

---

## Introduction

This lab guides you through the process of creating and adding a custom peripheral Intellectual Property (IP) block to a processor system by using the **Vivado IP Packager**.

The **Vivado Design Suite** provides the necessary tools to develop your own IP by helping you out to quickly turn designs, algorithms and small systems into reusable IP.

**Vivado IP Integrator** is an IP-centric design tool, part of the **Vivado Design Suite**, that will create either a simple or complex a digital system, by adding IPs from different sources, such as Xilinx IP, third-party IP and end-user Custom IP.

## Objectives

After completing this lab, you will be able to:

- Use the IP Packager feature of Vivado to create a custom peripheral Intellectual Property (IP) block
- Create a Custom IP which features an AXI-LITE interface for communication between the Processing System (PS) and the Programmable Logic (PL)
  - Use the Wizard to generate an AXI slave IP Core
  - Set the number of registers of the Core
- Deliver packaged custom IP to an end-user in a repository directory
- Create a Zynq system, adding the AXI Custom IP Core
- Modify and update the functionality of a custom IP
  - Edit the VHDL code for the custom IP Core
  - Update the Custom IP Core and repack it
- Add pin location constraints
- Export the hardware to SDK and create an Application Project
- Read and Write the IP Core's registers from the 'C' code (SDK)
- Generate the .elf file and download it to the ZedBoard

## Procedure

Three different approaches will be presented in this lab. Each has its own purpose and utility.

- **Case 1:** the simplest one, useful for small piece of code. The IP code is written within the code generated by the IP Creator tool.

- **Case 2:** the IP code is written in its own .vhd file. This component (.vhd component) is then instantiated in the the code generated by the IP Creator tool. The IP has no registers and its logic is very simple.
- **Case 3:** this case will go into details when the IP generated is more complicated, having registers to configure the functionality of the IP.

## Brief Introduction to PWM

Pulse Width Modulation or PWM is a technique for supplying electrical power to a load that has a relatively slow response. The supply signal consists of a train of voltage pulses such that the width of individual pulses controls the effective voltage level to the load. Both AC and DC signals can be simulated with PWM. In these notes we will describe the use of PWM on an Arduino for controlling LEDs and DC motors.

The PWM pulse train acts like a DC signal when devices that receive the signal have an electromechanical response time that is slower than the frequency of the pulses. For a DC motor, the energy storage in the motor windings effectively smooths out the energy bursts delivered by the input pulses so that the motor experiences a lesser or greater electrical power input depending on the widths of the pulses. For an LED, using PWM causes the light to be turned on and off at frequency that our eyes can detect. We simply perceive the light as brighter or dimmer depending on the widths of the pulses in the PWM output. Figure 1 shows a voltage signal comprised of pulses of duration  $\tau_o$  that repeat every  $\tau_c$  units of time. The output of a PWM channel is either  $V_s$  volts during the pulse or zero volts otherwise. If this signal is supplied as input to a device that has a response time much larger than  $\tau_c$ , the device will experience the signal as an approximately DC input with an effective voltage of

$$V_{eff} = V_s \frac{\tau_o}{\tau_c} \quad (1)$$

The ratio  $\tau_o/\tau_c$  is called the *duty cycle* of the square wave pulses. The effective DC



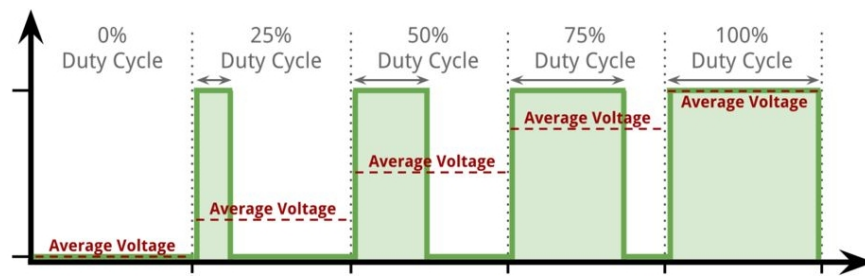
Figure 1: Nomenclature for definition of PWM duty cycle.

voltage supplied to the load is controlled by adjusting the duty cycle.<sup>1</sup>

So for large values of  $\tau_o$ , the longer would be the duration of the pulse in ON state, hence, the brighter would be the LED. Therefore, changing the duty cycle value the brightness of the LED would change from almost OFF to fully ON (fully bright).

<sup>1</sup> "Basic Pulse Width Modulation", G. Recktenwald.

In Figure 2 it can be seen the different values that  $V_{eff}$  would have for different duty cycle values.



---

## *Simple VHDL code for describing a PWM*

---

Below you will find a simple VHDL describing the behavior of a PWM module. This version of the code, we will see a more advanced version in case 3, has an input, `duty_cycle`, that will be used to set the value of the PWM duty cycle. This value will be set by the 'C' application code by writing to a register. Remember that all the peripherals in the Vivado environment has a specific memory address range to which the processor will write/read, and each peripheral has a register associated at each memory location.

We will use this VHDL code complete in case 2. For case 1, we will use part of the code, mainly the **`pwm_pr`** process.

It may come to your attention the 'strange' names used for the clock and reset signals. These names, `S_AXI_ACLK` and `S_AXI_ARESETN`, are actually the name used by the AXI Bus for the clock and reset signals. It's advisable to use the same name in the component that we create to facilitate the task of the synthesizer to recognize and associate the clock and reset signals.

This file can be found underneath the following directory

`/.../ictp_labs/.../lab_custom_ip/vhdl_src_case2/`

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

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

    generic (
        dc_bits : integer := 16);          -- number of bits for the duty cycle
    port (
        -- clock & reset signals
        S_AXI_ACLK      : in  std_logic;    -- AXI clock
        S_AXI_ARESETN   : in  std_logic;    -- AXI reset, active 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;

-----
-- 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;

-----
-- EOF
-----

```

VHDL code that describes the behavior of a PWM system

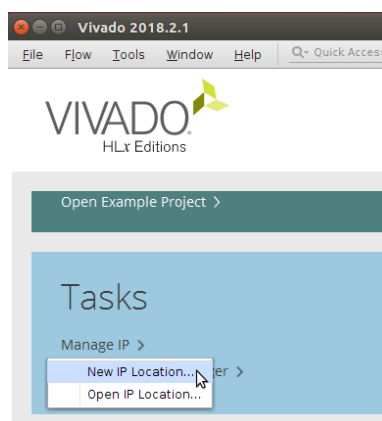
---

## Creating a Custom IP Core – Case 1

---

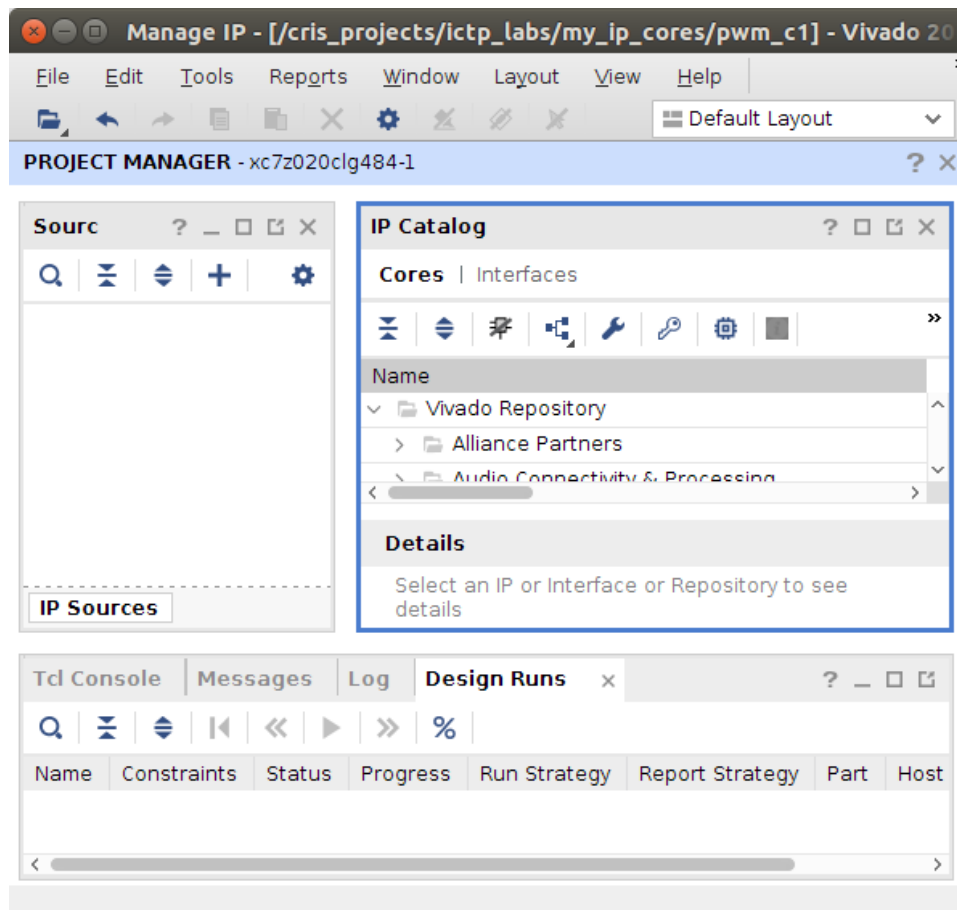
**Objective** Modify the VHDL AXI interface created by IP Creator by inserting a *process* of the PWM VHDL code.

1. Open **Vivado** and in the Tasks menu select, **Manage IP**→ **New IP Location**.



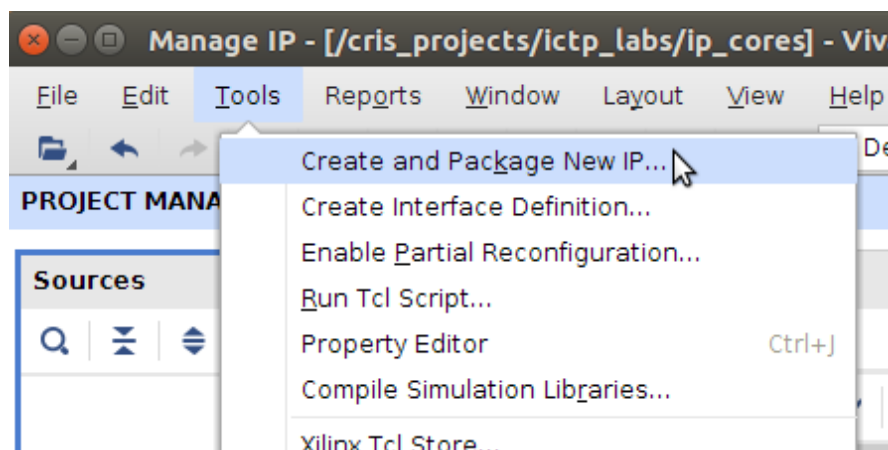
2. Click **Next** in the **New IP Location** window, **Manage IP Settings** configuration. Select:
  - The default part, xc7z020clg484-1, as **Part**
  - **VHDL** as the **Target Language**
  - **Vivado Simulator** as **Target Simulation**
  - **Mixed** as the **Simulator language**
  - For the **IP location** type: c:/.../ictp\_labs/my\_ip\_cores/pwm\_c1
3. Click **Finish** (leave other settings as defaults). Click **OK** if prompted to create the directory.





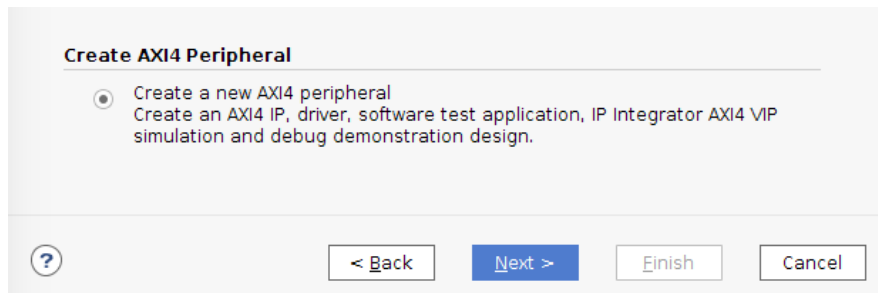
**Note:** Vivado will try to add the 'ip\_repo' subdirectory to the IP location that you set, just delete 'ip\_repo' from the path .

4. After clicking **Finish** the **Vivado Manage IP** GUI will open. This window looks a little different from the window we have been working with so far (Vivado GUI). Next step is to use the tool that enable the creation and packaging of the IP. Select **Tools** -> **Create and Package New IP**.



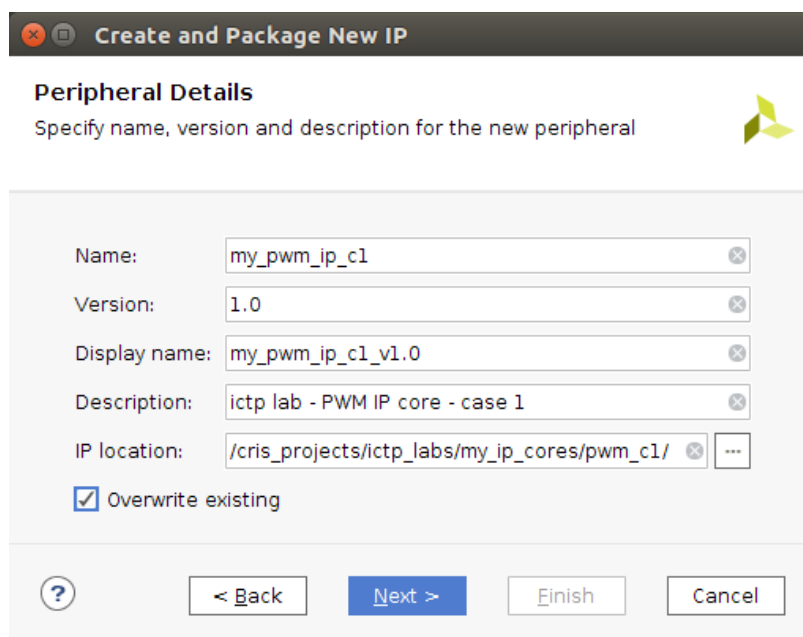
5. In the new window, **Create and Package New IP**, click **Next**.

6. Then, select **Create a new AXI4 peripheral** in the **Create AXI4 Peripheral** section. Click **Next**.

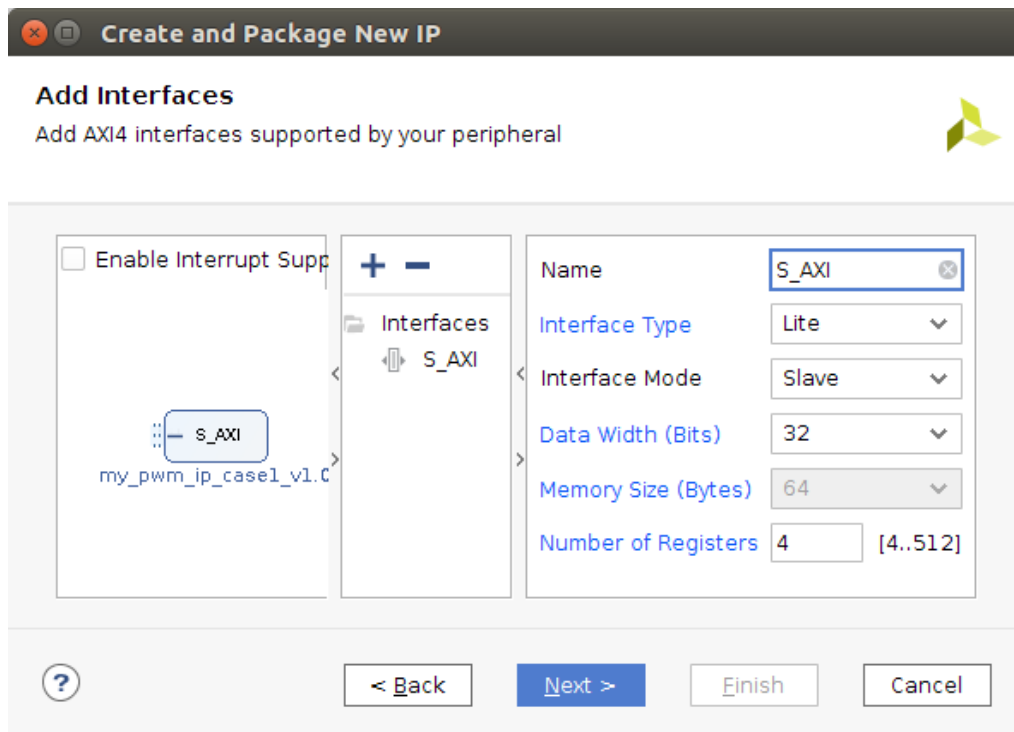


7. In the **Peripheral Details** window should be specified some information regarding the IP peripheral to be created. Below you will find some guides of the requested information:

- **Name:** name of the IP: my\_pwm\_ip\_case1
- **Version:** IP version: 1.0
- **Display Name:** the name of the IP that will be shown in the IP catalog
- **Description:** the IP description to share with an end =user of the IP:  
ictp lab - PWM IP core - case 1
- **IP Location:** to add the IP repository location to the IP Packager:  
c:/... /ictp\_labs/my\_ip\_cores/pwm\_c1/
- **Overwrite existing:** check this box to override an existing repository



8. Click **Next**.
9. On the **Add Interfaces** window, we will configure the name and some parameters of the AXI interface that to be created. Change the **Name** of the interface to **S\_AXI** (see figure below). Leave the other settings with their default values: **Lite** as **Interface Type**, **Slave** as **Interface Mode**, **Data Width 32** as **Data Width (Bits)**, and **4** as **Number of Registers**.



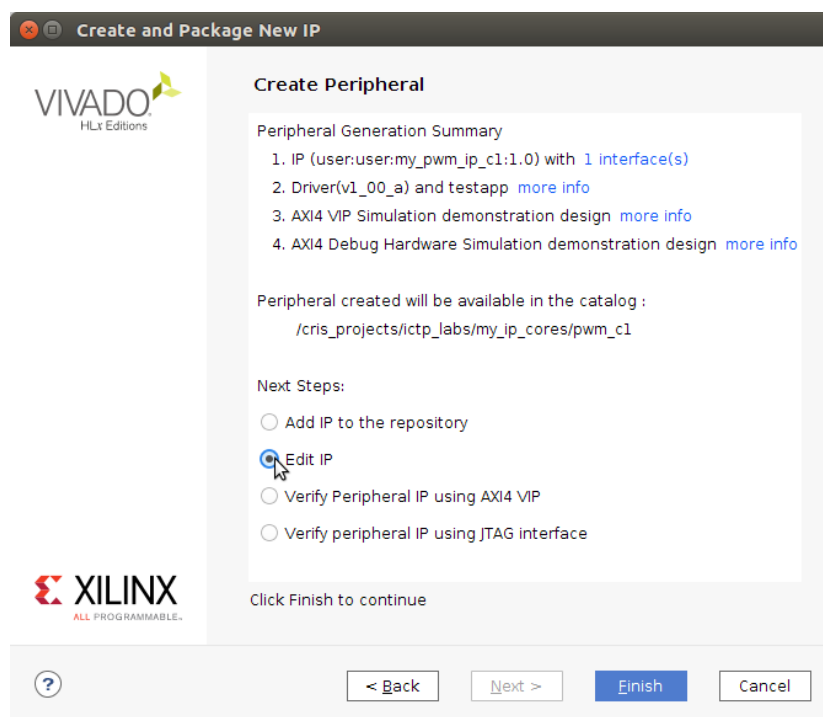
Note that some of the parameters are hyperlinked, clicking on the link you get a quick description of the parameter. Let's review some of these parameters and their values:

- **Interface Type:** For memory mapped interfaces **AXI4 FULL** allows burst of up to 256 data transfer cycles with a single address phase. **AXI4-Stream** allows a continuous data transfers. When coupled with other peripherals, such as DMA controllers, these **AXI4** protocols can be essential for meeting throughput requirements. **AXI4 LITE**, on the other hand, is a simpler protocol that satisfies the minimum hardware requirements of the AXI bus. This means that there will be fewer signals and state to worry about when designing your custom IP. It is a single transaction memory-mapped interface. For this particular custom IP, the **AXI4-LITE** interface is selected.
- **Interface Mode:** Since this IP is going to get commands by the processor (ARM processor in the PS), this IP will act as a Slave.
- **Data Width (bits):** We will keep the bus at the default width: 32.

■ **Number of registers:** This option will affect the generated Slave AXI code. With four registers, the data transferred from Master to Slave will be stored in 4 unique registers. The 4 least significant address bits are used to multiplex between these registers. The register's addresses will be b'0000 for the first register, b'0100 for the second register, b'1000 for the third register and b'1100 for the fourth register. The last two bits are always "b'00" due to 32 bits width alignment.

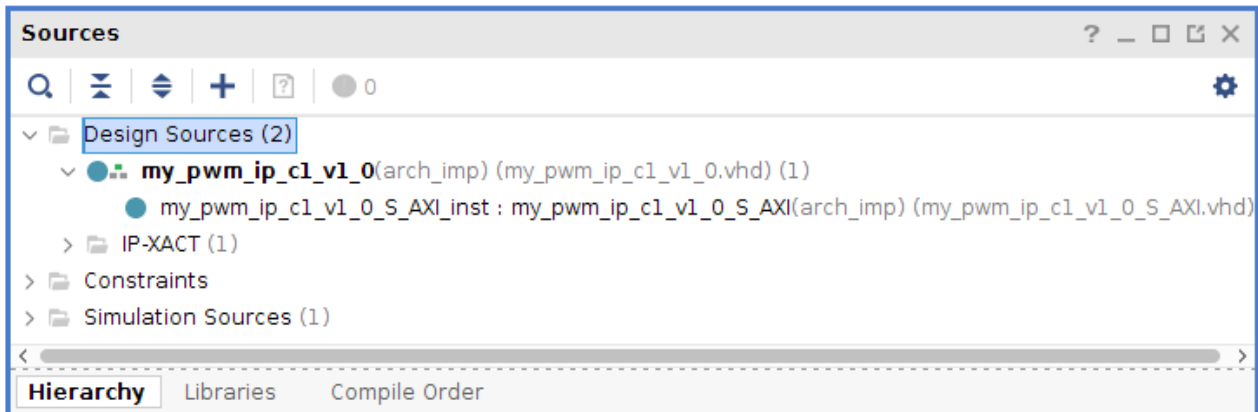
10. Click **Next**.

11. In the **Create Peripheral** window, first part details a summary of the IP to be created. The part below, shows the different options to be done as next steps. At this point we are not going to 'Add IP to the repository' because we have not finished the IP yet, the same reason for the bottom two options, therefore, select **Edit IP**, which will open the IP in the IP Packager. Then, click **Finish**.



12. The Vivado IP Packager opens up in a new **Vivado IP** project. Let's expand now the **Sources** pane.

## PROJECT MANAGER - edit\_my\_pwm\_ip\_c1\_v1\_0



Underneath **Design Sources** you will find two .vhd files. Both files are automatically created by the tool. The first file,

`my_pwm_ip_c1_v1_0.vhd`

is a simple VHDL file in which is instantiated the other VHDL file

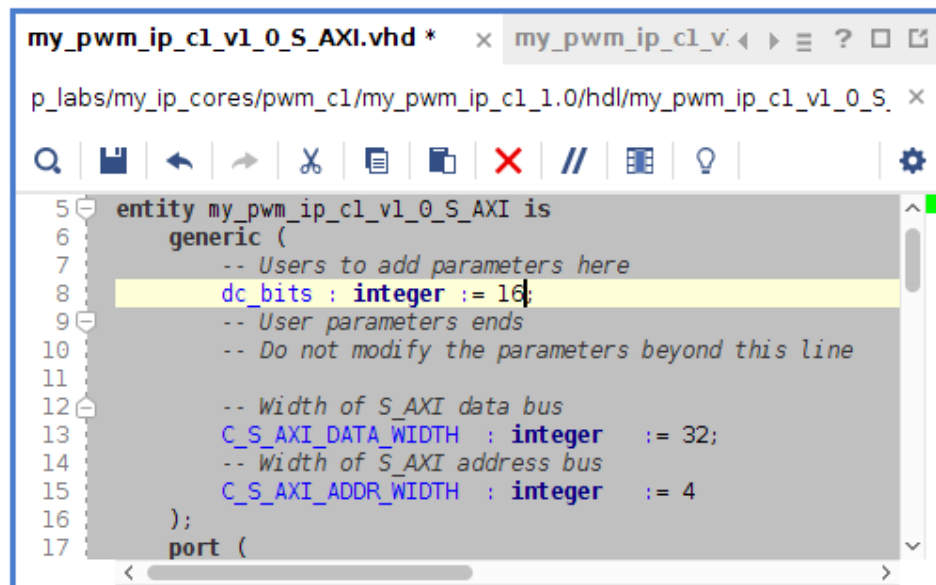
`my_pwm_ip_c1_0_S_AXI.vhd`

This second 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. For instance, you will find the process which is doing the reading of data through the AXI bus, other process which does the writing of data, other that generates the control signals, etc.

13. Now, we are going to modify the `my_pwm_ip_c1_0_S_AXI.vhd` file. Double click over the file name to open to edit it. The main modifications to this will be:

- add the generic, **dc\_bits**, to the generic part of the file
- add the **pmw** output to the entity ports
- add the **pwm\_pr** process
- declare **duty\_cycle** as internal signal
- assign the value of **slv\_reg0** to **duty\_cycle** signal
- make the **register 0** as the one that hold the value for the duty cycle

**13.2 Adding the generic:** from the PWM VHDL file (page 7) copy the generic declaration, `dc_bits : integer := 16;` and paste it in the space provided for user's generic parameter(s) in the `my_pwm_ip_c1_0_S_AXI.vhd` file. That is ~line 8 of the file (~line , indicates approximately that line number).



```

my_pwm_ip_c1_v1_0_S_AXI.vhd * x my_pwm_ip_c1_v1_0.vhd
p_labs/my_ip_cores/pwm_c1/my_pwm_ip_c1_1.0/hdl/my_pwm_ip_c1_v1_0_S_
5 entity my_pwm_ip_c1_v1_0_S_AXI is
6     generic (
7         -- Users to add parameters here
8         dc_bits : integer := 16;
9         -- User parameters ends
10        -- Do not modify the parameters beyond this line
11
12        -- Width of S_AXI data bus
13        C_S_AXI_DATA_WIDTH : integer := 32;
14        -- Width of S_AXI address bus
15        C_S_AXI_ADDR_WIDTH : integer := 4
16    );
17    port (

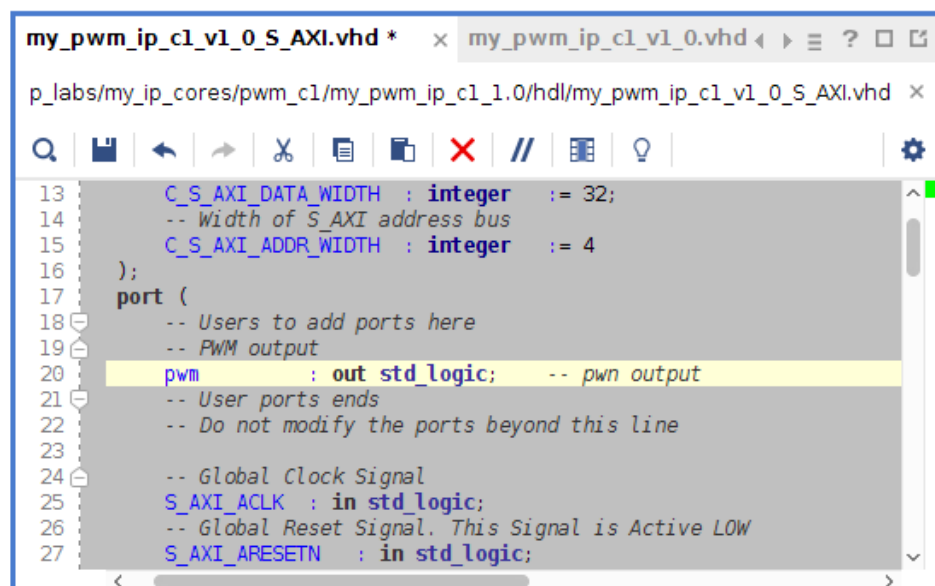
```

**13.3 Adding the pmw output to the entity ports:** from the PWM VHDL file (page 7 of this document) copy the declaration of **pwm** as an output port.

```
-- PWM output
```

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

and paste it in ~line 19 of the AXI VHDL file (just below the comment “-- User to add ports here”).



```

my_pwm_ip_c1_v1_0_S_AXI.vhd * x my_pwm_ip_c1_v1_0.vhd
p_labs/my_ip_cores/pwm_c1/my_pwm_ip_c1_1.0/hdl/my_pwm_ip_c1_v1_0_S_AXI.vhd
13 C_S_AXI_DATA_WIDTH : integer := 32;
14 -- Width of S_AXI address bus
15 C_S_AXI_ADDR_WIDTH : integer := 4
16 );
17 port (
18     -- Users to add ports here
19     -- PWM output
20     pwm : out std_logic; -- pwn output
21     -- User ports ends
22     -- Do not modify the ports beyond this line
23
24     -- Global Clock Signal
25     S_AXI_ACLK : in std_logic;
26     -- Global Reset Signal. This Signal is Active LOW
27     S_AXI_ARESETN : in std_logic;

```

**13.4 Adding the pwm\_pr process:** from the **pwm\_simle.vhd** file copy the **pwm\_pr process**, and paste it from the ~line 387. That is, below of the comment line “-- Add user logic here”.

```

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 clock
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;

```

The red line, in ~line 396, means there is an error in that line. In this case the error is due to the fact that the **'duty\_cycle'** signal is not declared in this VHDL file. Which is correct. Thus, we are going to declare that signal.

**13.5** Declare **duty\_signal** as a signal, in the declarative part of the architecture, ~ line 121:

```

121 --
122 signal duty_cycle: std_logic_vector(dc_bits-1 downto 0);
123
124 begin

```

**13.6** Now, we are going to associate **register0** with **duty\_cycle**. Thus, the value that will be written into **'register0'** will be the value of the **duty\_cycle**. Remember 'register 0' will be write/read from the 'C' application program that you'll write. At ~line 389 (below '- - Add user logic here') write the following VHDL statement.

```

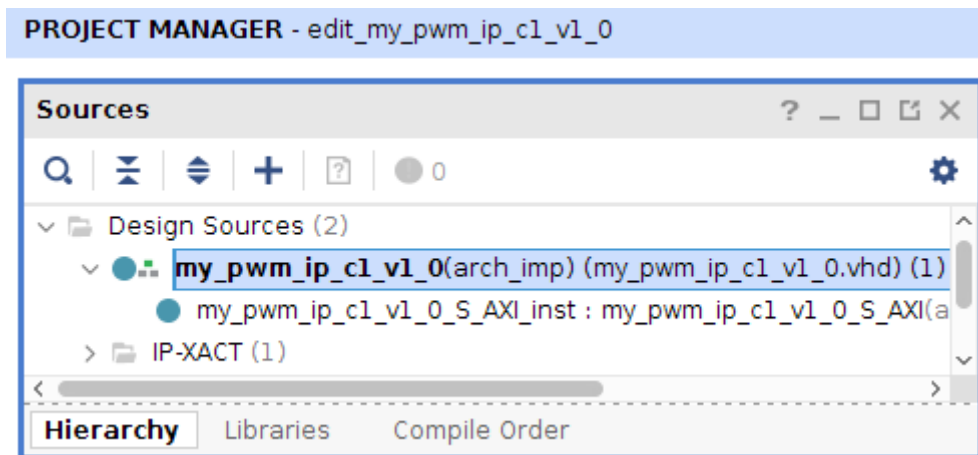
389 -- Add user logic here
390 duty_cycle <= slv_reg0(dc_bits-1 downto 0);
391 pwm_pr : process (S_AXI_ACLK, S_AXI_ARESETN) is

```

*Note 1: in all the AXI interface VHDL files, the 'register 0' is called 'slv\_reg0'.*

Note 2: when importing the process the signal `duty_cycle` could have been replaced directly by `slv_reg0`, the final result it would be the same.

14. In the next step, we will modify the top level VHDL file. Going back to the **Sources** pane, double click over the file `my_pwm_ip_c1_v1_0.vhd`.



15. In ~line 8, below the comment `-- Users to add parameters here`, add the following line:

```
dc_bits : integer := 16;
```

16. In ~line 19 add the **pwm** output statement:

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

17. Since we have modified the `my_pwm_ip_c1_v1_0_S_AXI` component (13.2, 13.3) we need to do the some modifications in the component declaration as well as in the component instantiation in the `my_pwm_ip_c1_v1_0.vhd` file :

- 17.1 In ~line 55, below the **generic** keyword, add the generic declaration:

```
dc_bits : integer := 16;
```

- 17.2 In ~line 61, below the **port** keyword, add the **pwm** output port.

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



```

49
50 architecture arch_imp of my_pwm_ip_c1_v1_0 is
51
52     -- component declaration
53     component my_pwm_ip_c1_v1_0_S_AXI is
54         generic (
55             dc_bits           : integer := 16;
56             C_S_AXI_DATA_WIDTH : integer := 32;
57             C_S_AXI_ADDR_WIDTH : integer := 4
58         );
59         port (
60             -- PWM output
61             pwm           : out std_logic;    -- pwm output
62             S_AXI_ACLK    : in std_logic;
63             S_AXI_ARESETN : in std_logic;
64             S_AXI_AWADDR  : in std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);
65             S_AXI_AWPROT  : in std_logic_vector(2 downto 0);
66             S_AXI_AWVALID : in std_logic;

```

18. Next, we are going to modify the component instantiation (that begin at ~line 88).

18.1 In ~line 91, below the keyword **generic map**, add the following statement:

```
dc_bits => dc_bits,
```

18.2 In ~line 96, below keyword **port map**, add the following statement:

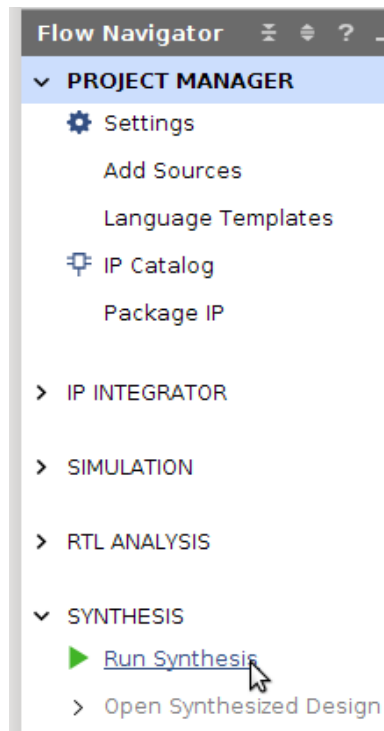
```
pwm => pwm,
```

```

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

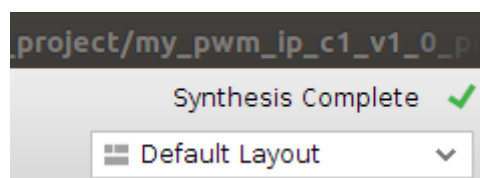
```

19. If there are no errors, no red marks on the right side of the VHDL code window, then the next step is to save all the files.
20. Then, click **Run Synthesis**. This step is going to check the design synthesizes correctly before packaging the IP. Save the project if prompted.



Important: It is strongly advisable to simulate and verify the functionality of the design before proceeding with the next steps. We will not do that for this lab.

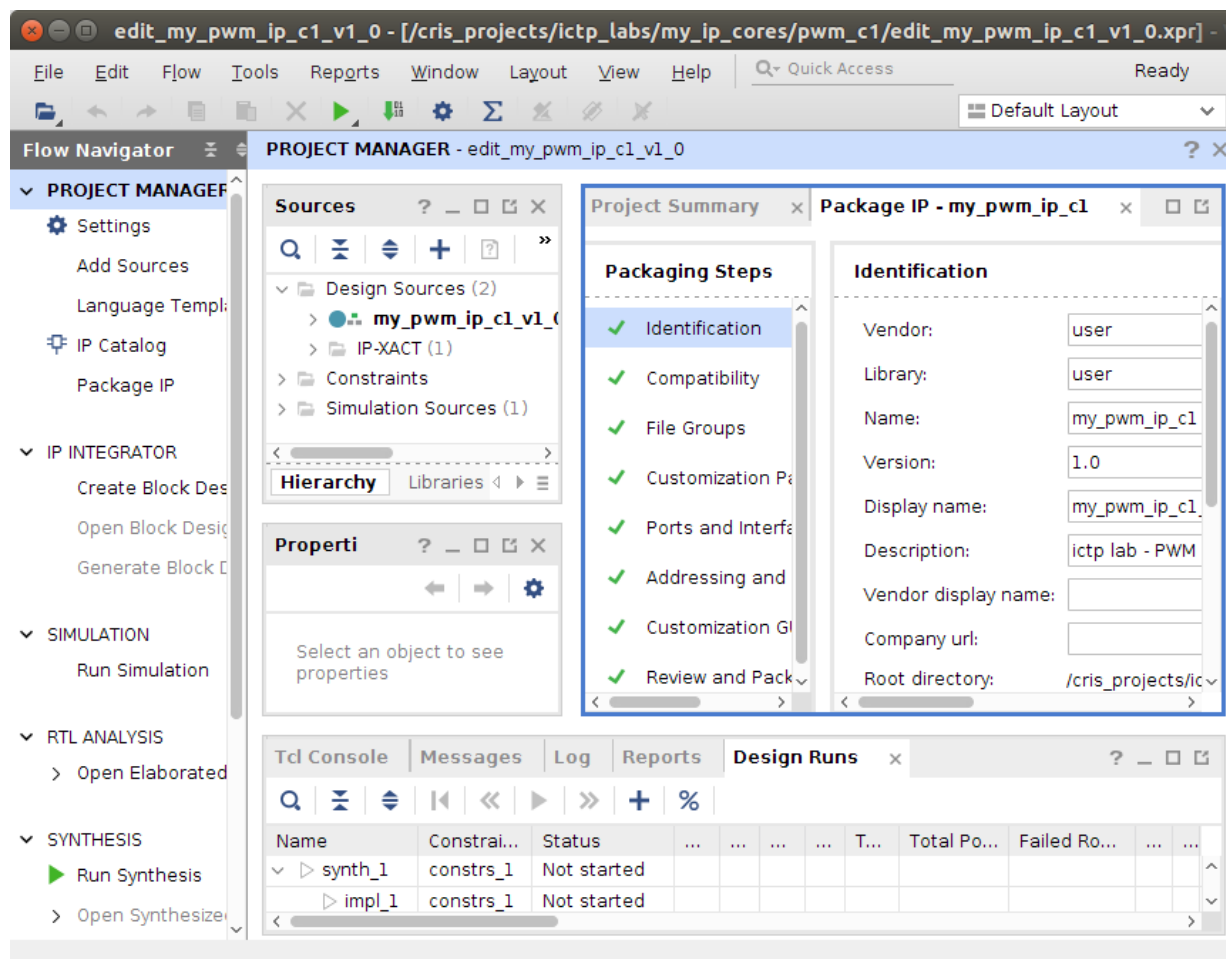
21. Look for the **Synthesis Complete** message on the top right corner of Vivado.



22. After synthesis is finished, check the **Messages** tab for any errors and correct them if necessary.
23. Do not run **Implementation** if prompted.
24. Let's now click on the **Package IP** tab to carry out a series of configurations. This project is created according to the default settings of the **my\_pwm\_ip\_c1** IP project.

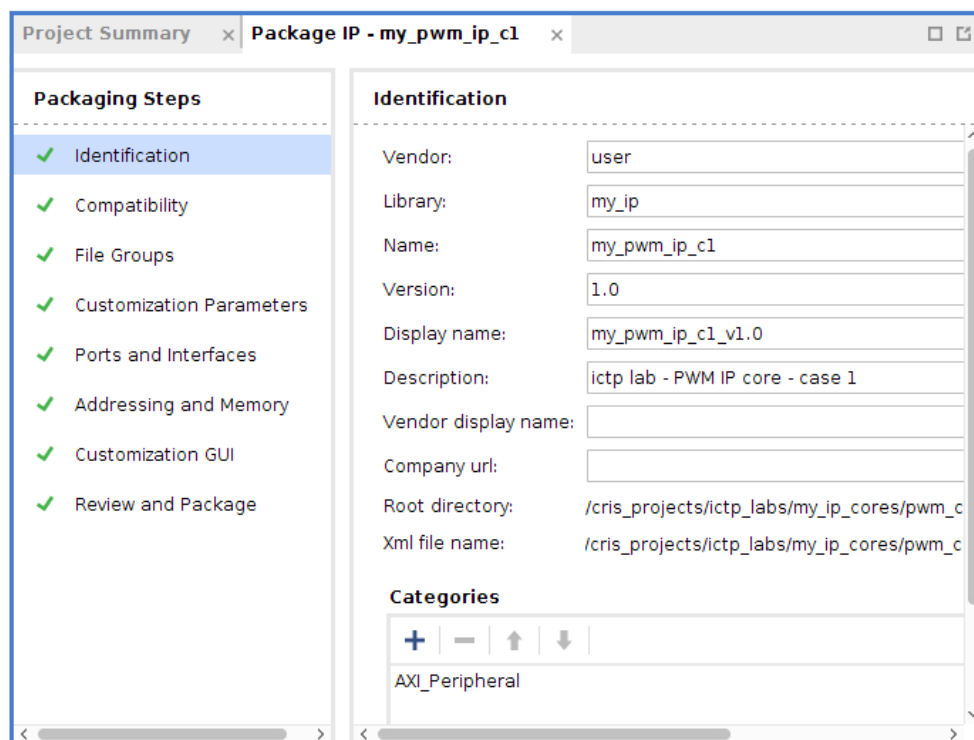
*Note: In case that the Package IP tab is not available, in the Source window select and expand IP-XACT, then double click on component.xml.*

The **Package IP** tab shows different options that describes the IP to be edited.

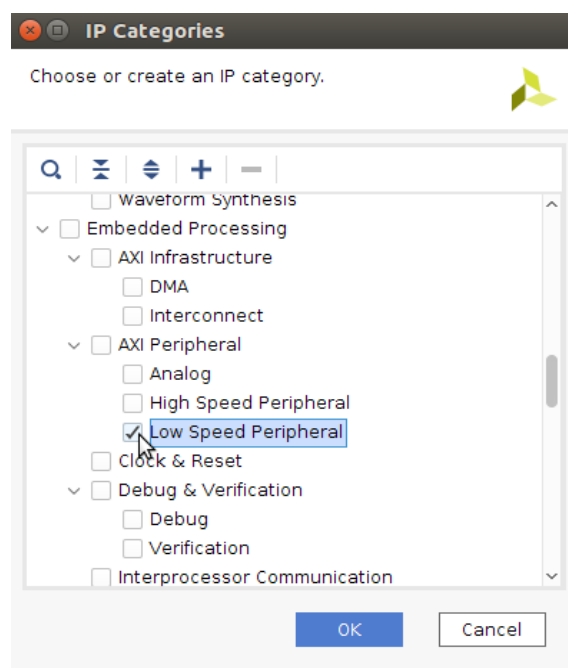


Note: the value showed by default in the previous figure, like Vendor: user, can be changed from Project Manager → Settings → IP → Packager.

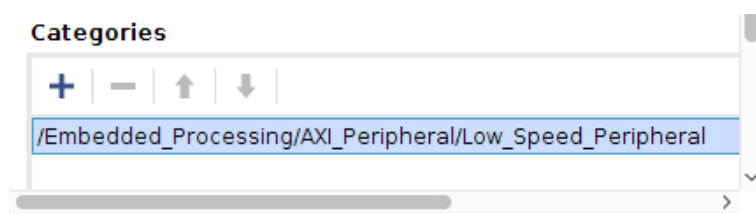
**24.1** The **Identification** page is populated by default with the values and names that we previously introduced. The **Vendor**, **Library**, **Name** and **Version** (aka **VLNV**) must be unique per each IP, since they identify the IP in the Vivado IP catalog. Change the values to something similar shown in the figure below.



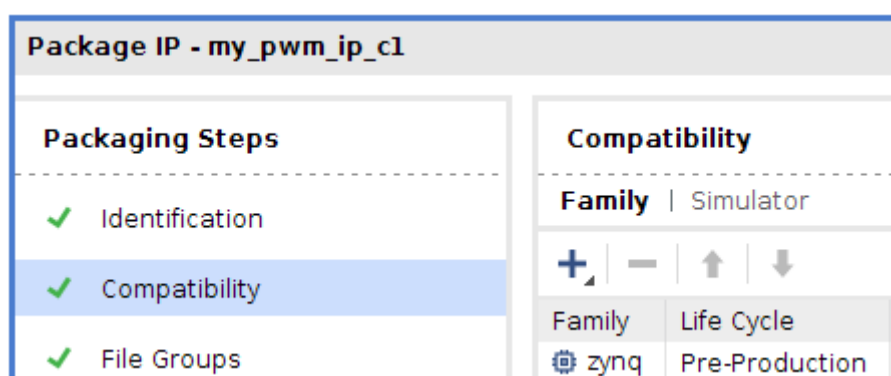
In the bottom part of **Identification** there is a configuration related to the category of the IP. There are different categories of IPs, separately by their functionalities. Let's change the category of our IP from **AXI\_Peripheral** (the one assigned by default) to **Low Speed Peripheral**. So, click on the + icon to open the **IP Categories** window. Then, uncheck **AXI Peripheral** and check **Low Speed Peripheral**.



After clicking **OK**, the **Categories** information should be updated to **Low Speed Peripheral**.



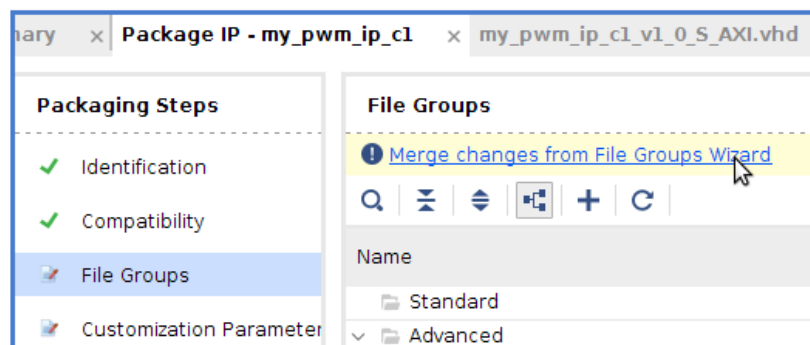
**24.2** The **Compatibility** page configures the specific Xilinx FPGA part compatible with the Custom IP. The family shown in here is inherited from the device selected for the IP project at the beginning of this IP creation. Hence, it should show the Zynq family. We select the **Zynq** family since we will be using this IP on the ZedBoard (that has a Zynq SoC-FPGA device).



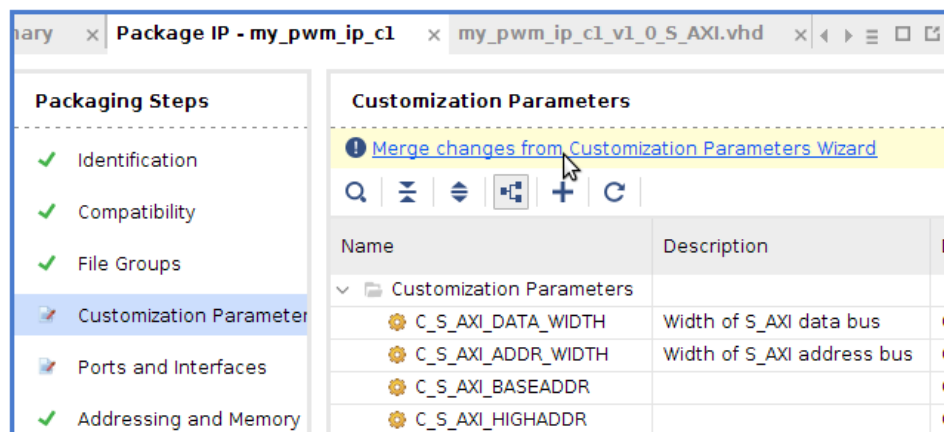
In case you have a board with a different device, just click on the **+** icon and select **Add family explicitly**, then select the desired SoPC family.

The **Life Cycle** of the IP can be set to different values, you can get the different options by right click on the **Life Cycle** field. This values will be updated by the designer of the IP as the IP goes into the different stages of its development. **Beta**, **Pre-Production** and **Production** are the ones advisable. Whereas that setting the property to **Discontinued**, **Hidden**, **Removed** or **Superseded** ensures that the IP does not appear in the IP Catalog.

**24.3** The **File Groups** page provides a list of the file that make up the **Custom IP**. Each file is grouped into specific groups that define the behavior of the file and its use. The files are grouped in the synthesis files, simulation files, software drivers files, layout and block diagram files. These files are automatically created by the IP Packager. Since there has been some changes in the files, there is a message saying **"Merge changes from File Groups Wizard"** to update the files with the changes made to the AXI IP VHDL code.



**24.4 Customization Parameters** page lists all the parameters (generics) of the Custom IP. These parameters can be used to customize the VHDL top source IP file. Since the VHDL code we added to the Custom IP, `pwm_simple.vhd`, has a generic (parameter). `dc_bits`, the message “**Merge changes from Customization Parameters Wizard**” comes up.



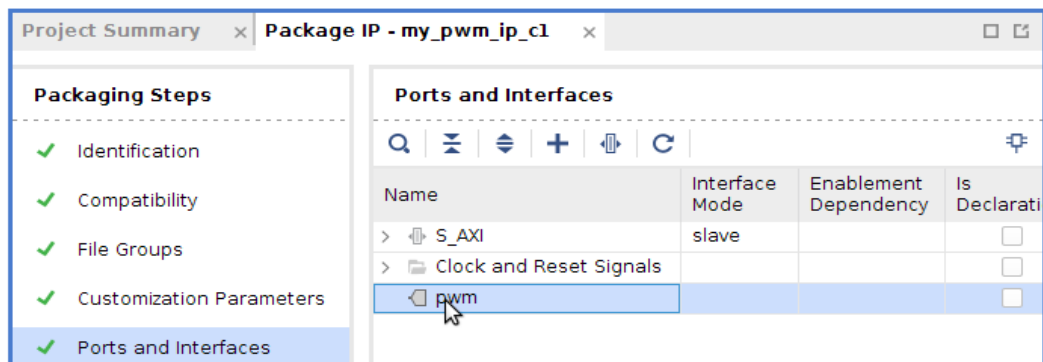
Single click over it and as result the generic `dc_bits` (declared in the VHDL file) should come up underneath the **Hidden Parameters** folder.

Name	Description	Display Name	Value
Customization Parameters			
C_S_AXI_DATA_WIDTH	Width of S_...	C S AXI DATA WIDTH	32
C_S_AXI_ADDR_WIDTH	Width of S_...	C S AXI ADDR WIDTH	4
C_S_AXI_BASEADDR		C S AXI BASEADDR	0xFF
C_S_AXI_HIGHADDR		C S AXI HIGHADDR	0x00
Hidden Parameters			
dc_bits		Dc Bits	16

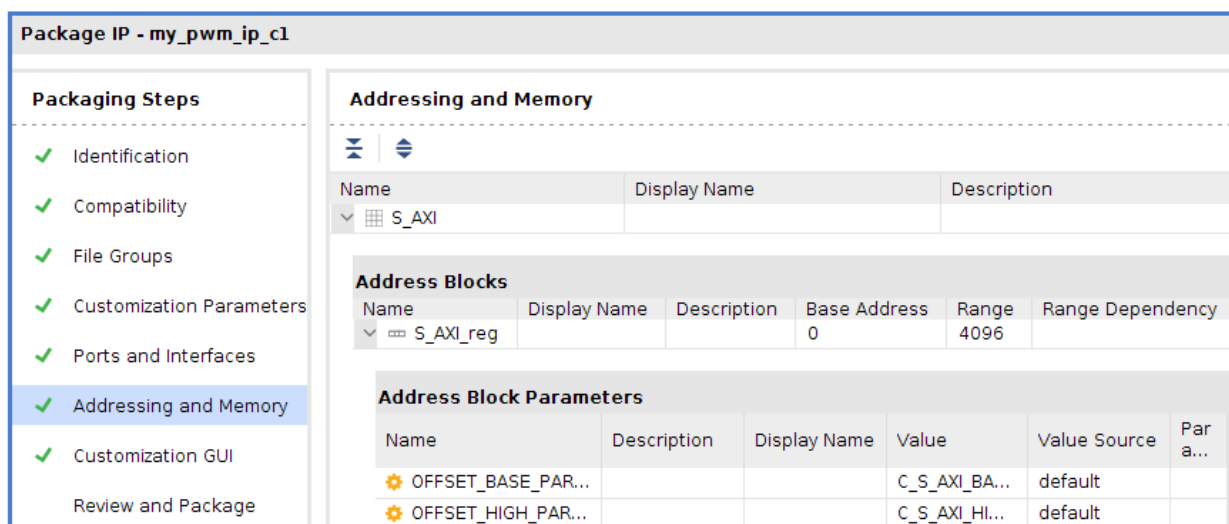
The **Customization Parameters** will be showed in the **Custom IP Customization GUI** and can be modified by the user. **Hidden Parameters** have the option to be shown or not (this is shown in the Customization GUI section, 24.7).

Note: the *Custom IP Customization GUI* is a GUI that can be accessed once the Custom IP has been added to the project system in the Vivado IP Integrator.

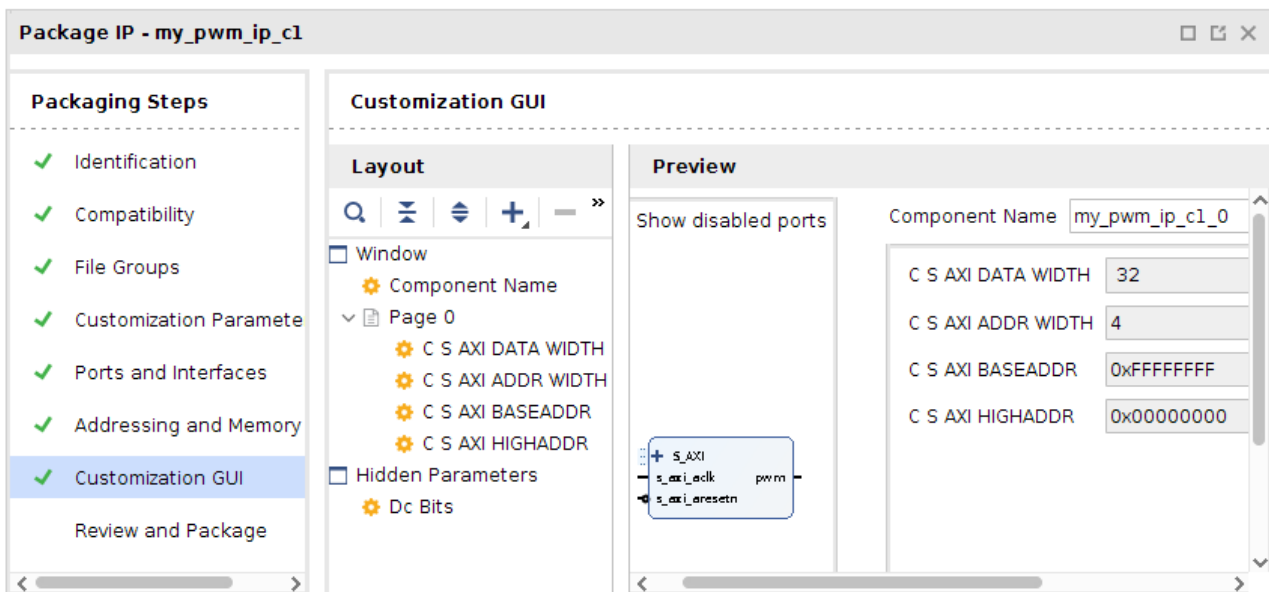
**24.5 The Ports and Interfaces** page provides a listing of ports and interfaces of the Custom IP. They are listed in a table format that contains several columns of information. There is nothing to do in this section, but check that the **pwm** output port is in shown in the list. Remember that **pwm** is the output port of the Custom IP, besides all the AXI I/O interface signals.



**24.6 The IP Addressing and Memory** page details the memory range assigned to the Custom IP. There is no need of changing in this section.



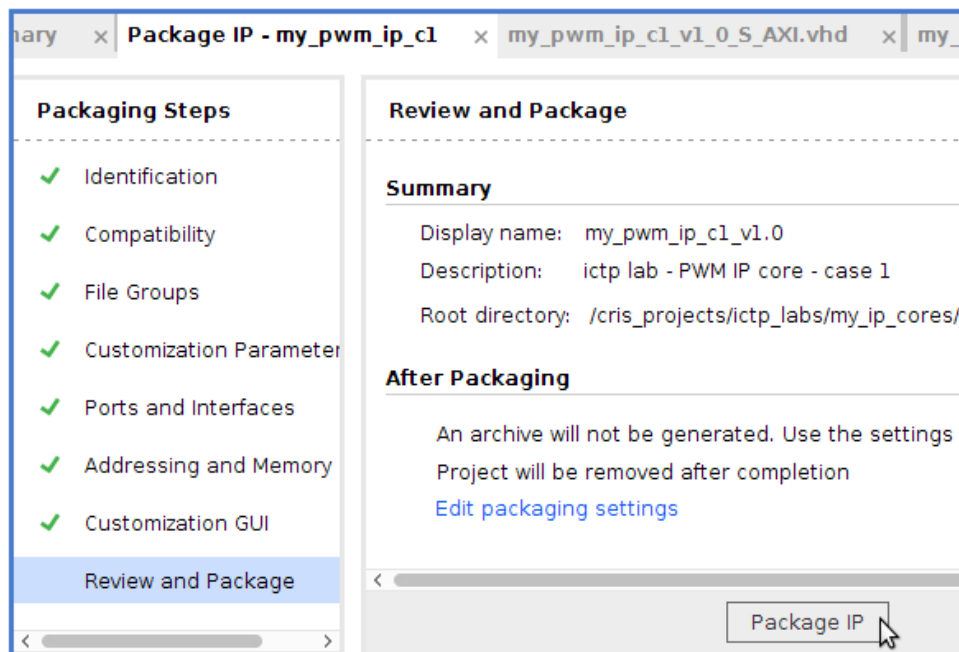
**24.7 The Customization GUI** page provides an environment for customizing the GUI interface of the Custom IP and how the symbolic representation of the **Custom IP** will look like. This step is useful when the **Custom IP** has several parameters that can be configured and we want to make sure that the final user can change their default values. Initially, the **Customization GUI** generates a layout with all the viewable parameters displayed on a single page of the GUI.



For this **Custom IP** there is no need of customizing the GUI. However, if in a future you want to have the IP parameters (generics) be available to customize the IP, you need to do a right click on the **Hidden Parameter, Dc Bits** in this case, select **Edit Parameter**, and in the **Edit IP Parameter** window check **Visible in Customization GUI**. Then the **dc\_bits** parameter will be available to be modified as needed by the final user.

**24.8 Review and Package** section provides both, a summary of the Custom IP and what Vivado will do after packaging is completed. Click on **Package IP**. One important point is that if any changes occur to any of the packaging steps of the custom IP, the custom IP must be repackaged to take effect the changes.

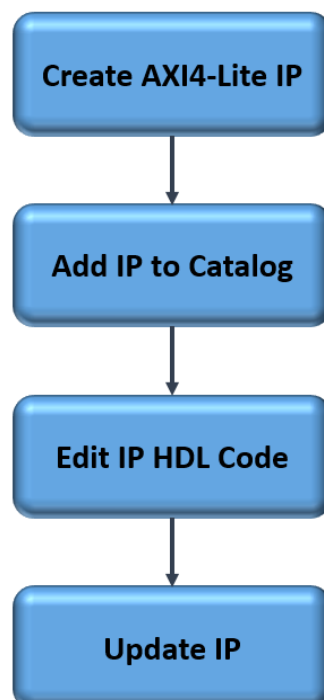




24.9 A successful message of packaging should come up.

24.10 Click on **Yes** to close the IP Core just generated.

24.11 With all these tasks the Custom IP has been created. A resume of all the executed tasks are shown in the flow diagram in the following figure.



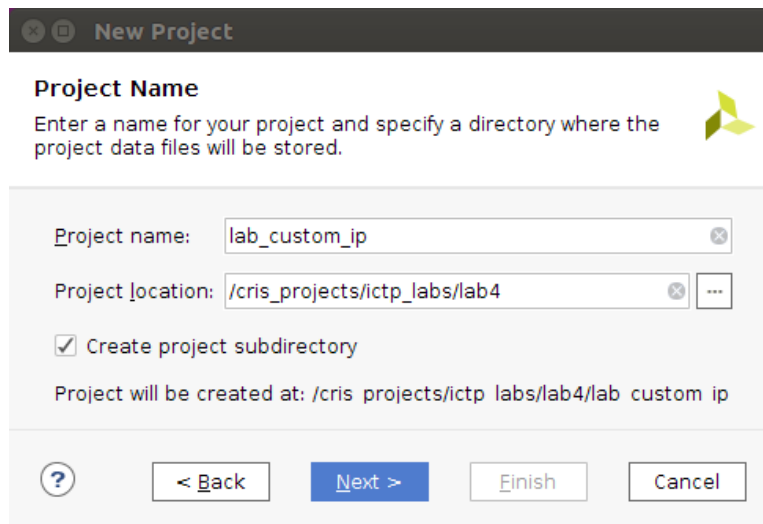
---

## Adding the Repository to Vivado – Creating a Vivado Project with the PWM IP

---

**Objective** Create a new Vivado project and adding the repository directory, where the PWM IP lies, to the Vivado IP integrator tool.

25. Let's create a new Vivado project in which we are going to add the just created. Open **Vivado** and select **Create Project**. Enter **lab\_custom\_ip** as the project name. Make sure that the **Create Project Subdirectory** option is checked, the project directory path is `c:/.../ictp_labs/lab4/` and click **Next**.



**New Project**


**Project Name**  
Enter a name for your project and specify a directory where the project data files will be stored.

Project name:

Project location:

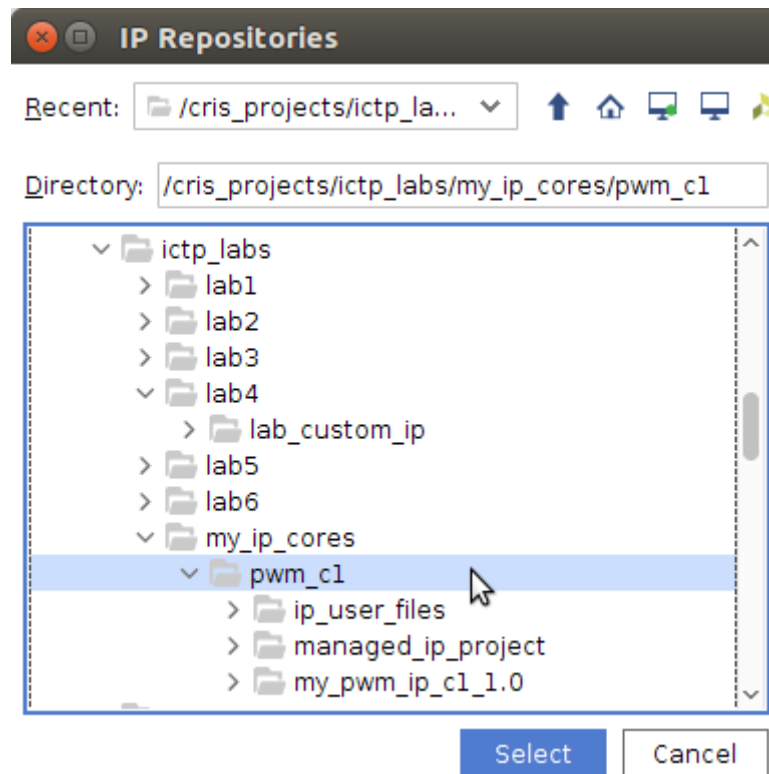
☒ Create project subdirectory

Project will be created at: /cris\_projects/ictp\_labs/lab4/lab\_custom\_ip

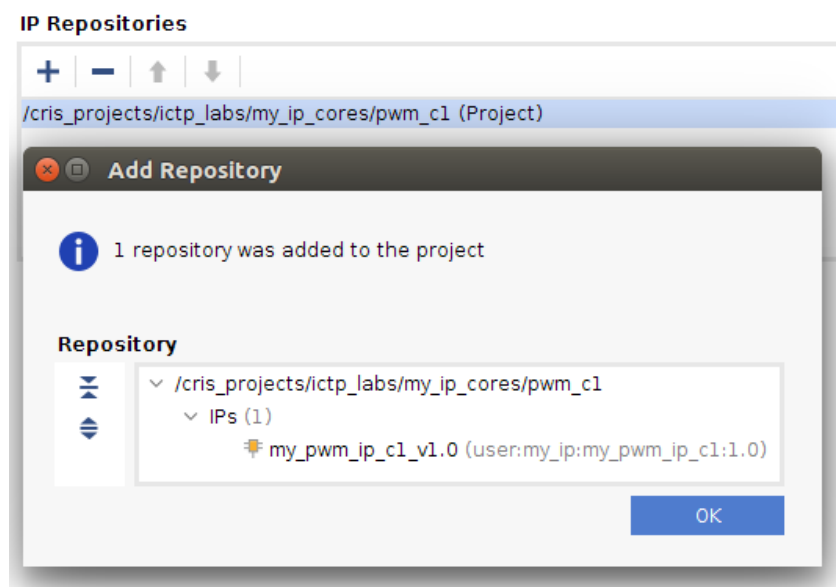


26. In the **Project Type** window, select **RTL Project**. Click **Next**.
27. Click **Next** in the **Add Sources** and in the **Add Constraint** windows.
28. In the **Default Part** window select **Boards**, then **ZedBoard**. Then **Next** and **Finish**.
29. In the **Flow Navigator** pane click **Settings** in the **Project Manager** section. Then, select **IP > Repository** in the left pane of the **Project Settings** form.

In this step we are going to add the directory where our PWM IP lies. Click the **+** icon, then browse to the main directory used to create the IP.



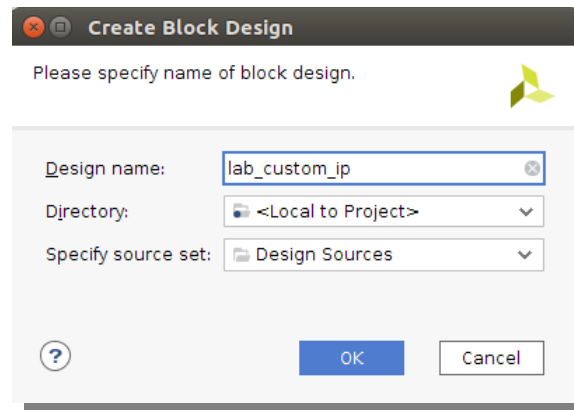
30. After clicking **Select**, a window will come up stating that *'1 repository was added to the project'* and the **IP Repositories** pane should be updated with the new repository directory.



31. Click **OK** in the **Add Repository** window. Our **pwm\_c1** IP has been added to the Vivado IP Manager.

32. Then click **OK** in the **Settings** window.

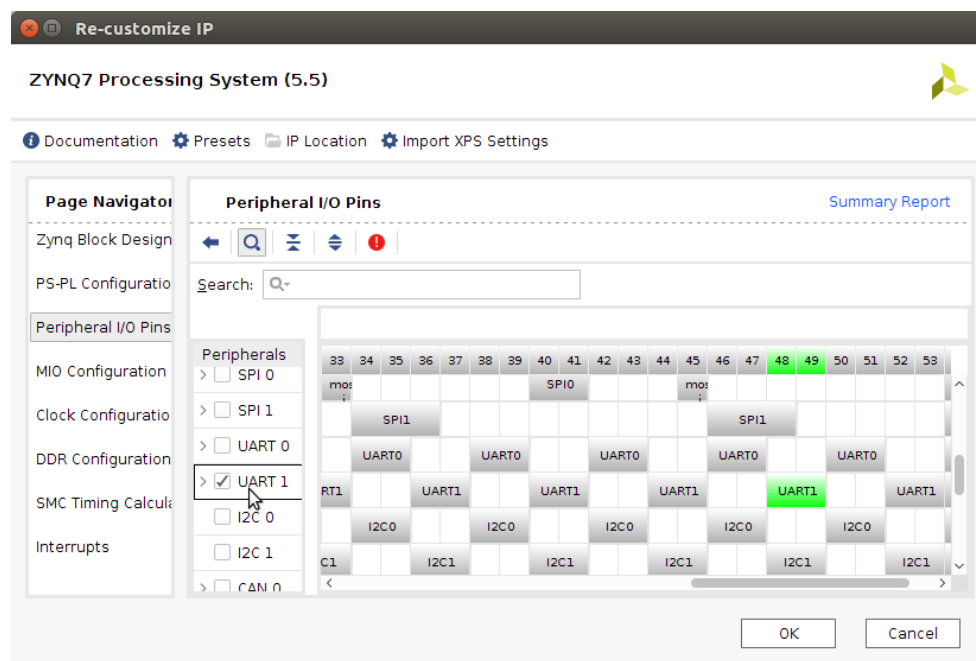
33. Let's now create the project itself. Click **Create Block Design** in the **Flow Navigator** pane. Change the **Design Name** to **lab\_custom\_ip**.



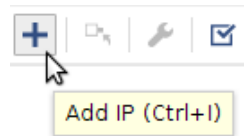
34. Add a **Zynq Processing System IP** to the **Block Design** canvas.

35. Click on **Run Block Automation**. In the upcoming window check **Apply Board Presets**. Click **Ok**.

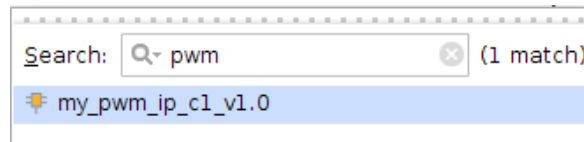
36. We are going to set the **UART1** peripheral for the communication between the Zynq and the PC. Double-click on the **Zynq Processing System** block. In the **Re-customize IP** window go to **Peripheral I/O Pins**, and select **UART1**. Uncheck all the other peripherals. Then click **OK** to close this window and go back to the block design.



37. In the block design, let's now add our IP. Click on **Add IP** icon.



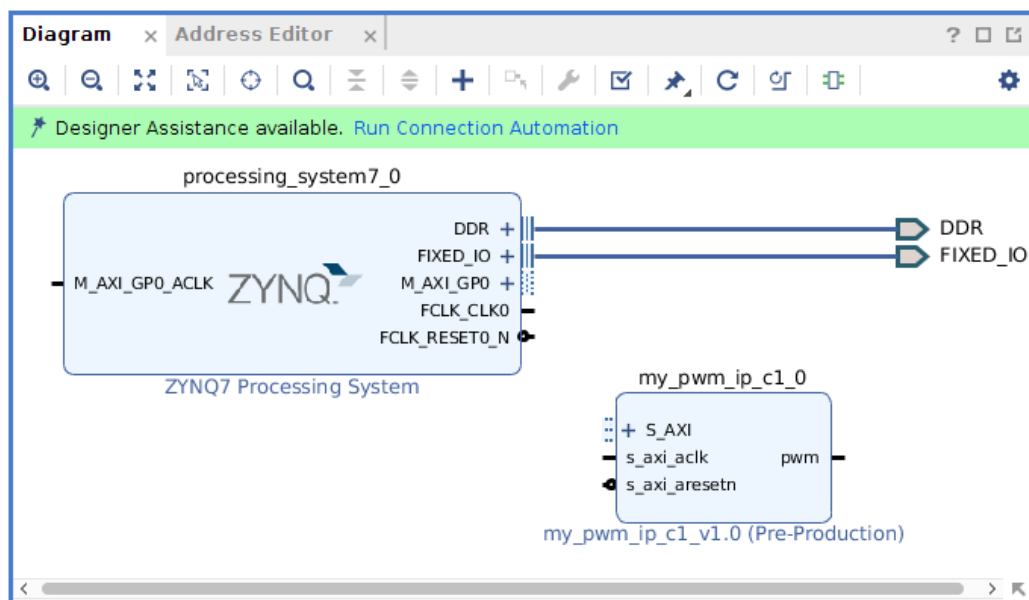
38. In the **Vivado IP Manager** window, type **pwm** in the search option.



39. Double click in the **pwm** IP.

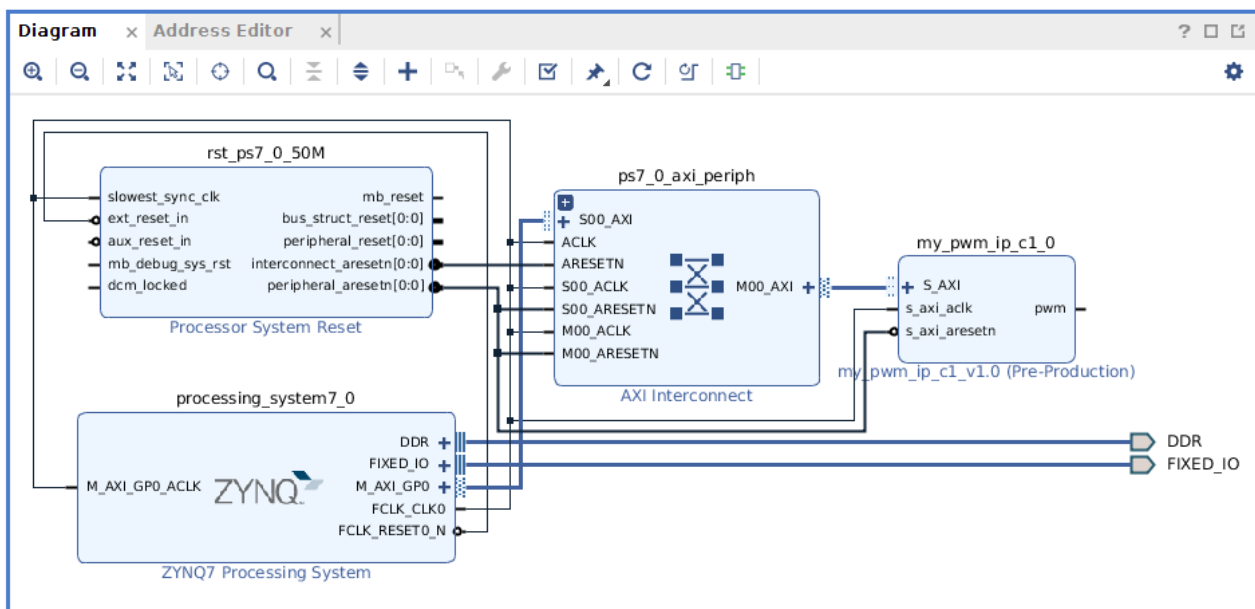
Note: if the **pwm** IP does not show up in the search, it means you have missed some of the previous steps.

40. You should have something like this:

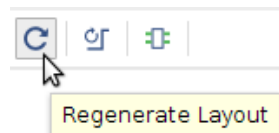


41. Click on **Run Connection Automation**. In the Run Connection Automation leave the default options as they are. Click **Ok**.

42. The block diagram should look like something like this:



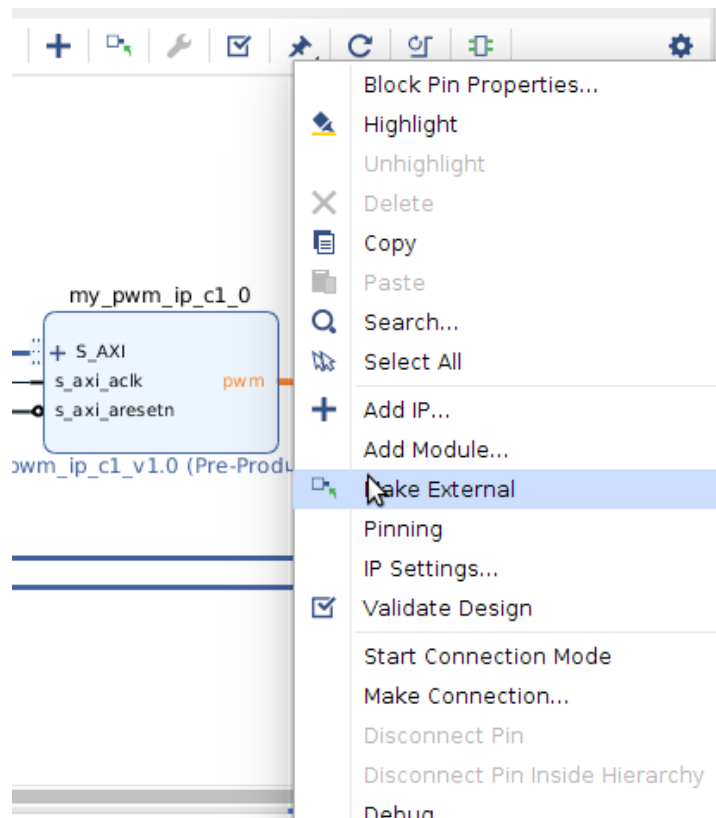
Remember to use the regenerate layout tool to get a better view of the block diagram.



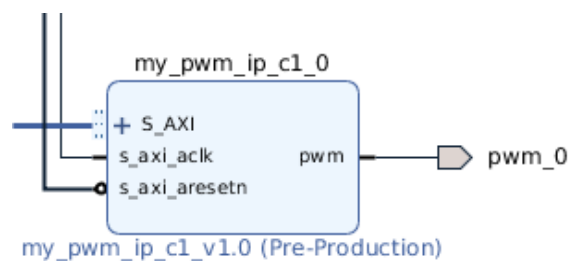
43. Check the **Address Editor** tab to find out the addresses assigned to the IP.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [ 1 G ])					
my_pwm_ip_c1_0	S_AXI	S_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF

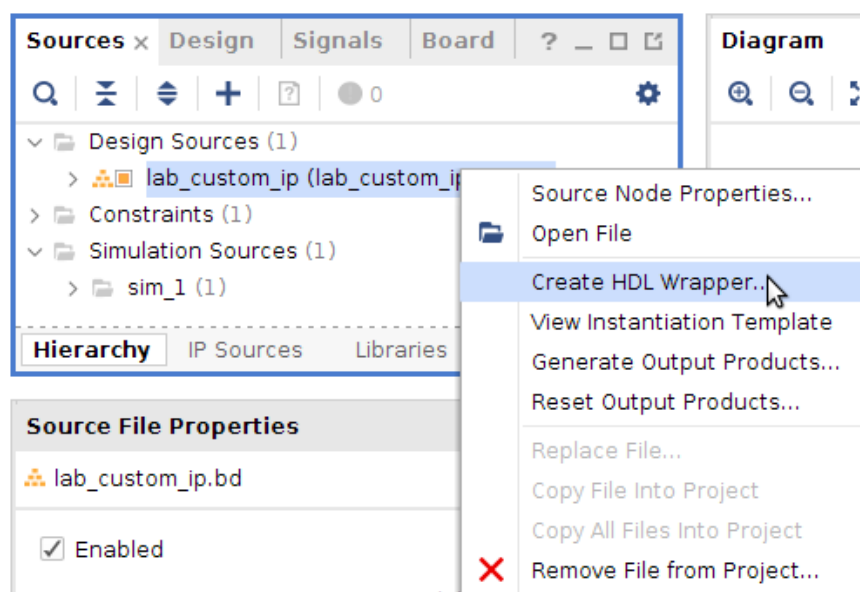
44. Let's now make the **pwm** output as an external output (going outside the Zynq). Select the **pwm** output on the **PWM IP** instance, by clicking on the **pwm** pin (the color of the port should change to light brown). Then, do right-click on the **pwm** output line and select **Make External**.



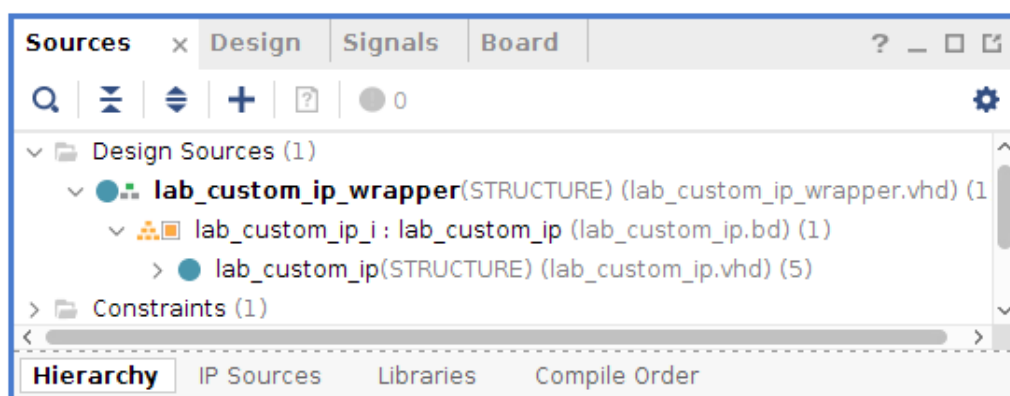
45. The output port is set for **pwm** and it's called **pwm\_0**.



46. Let's now create the VHDL wrapper. Go to the Sources pane, right-click on **lab\_custom\_ip(lab\_custom\_ip.bd)** and first select **Generate Output Products** and in the upcoming window select **Generate**. Then, again do right-click on **lab\_custom\_ip(lab\_custom\_ip.bd)**, this time select **Create HDL Wrapper**. Leave the option 'Let Vivado manage wrapper and auto-update', and click **OK**.



47. The Source pane should now be updated.



48. In order to control a specific LED of the ZedBoard with the **pwm\_0** output, it's necessary to associate **pwm\_0** output with the I/O pin ties to the specific LED. Reviewing the ZedBoard User Guide, we find that **LED0** is associated to pin **T22**.

### 2.7.3 User LEDs

The ZedBoard has eight user LEDs, LD0 – LD7. A logic high from the Zynq-7000 AP SoC I/O causes the LED to turn on. LED's are sourced from 3.3V banks through 390Ω resistors.

Table 14 - LED Connections

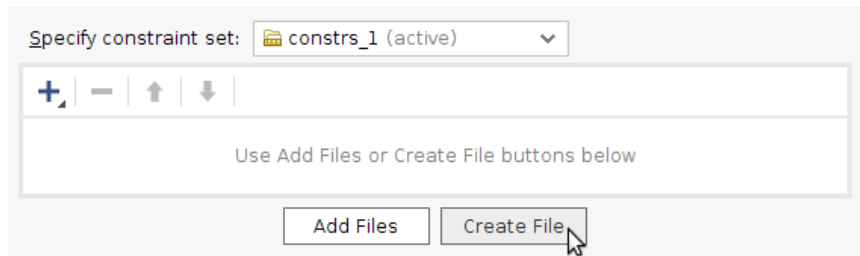
Signal Name	Subsection	Zynq pin
LD0	PL	T22
LD1	PL	T21
LD2	PL	U22
LD3	PL	U21
LD4	PL	V22
LD5	PL	W22
LD6	PL	U19
LD7	PL	U14
LD9	PS	D5 (MIO7)



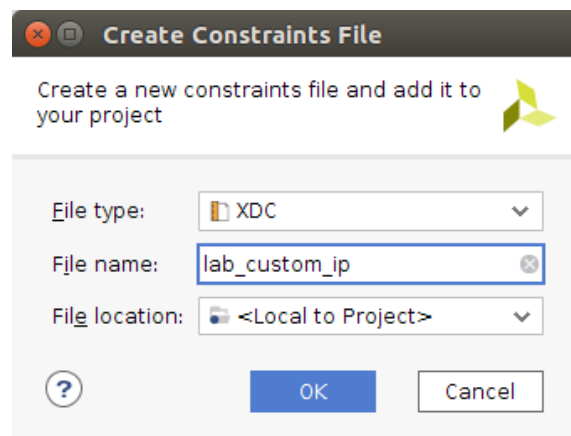
49. We are going to create a constraint file, called **Xilinx Constraint** file, to associate pin T22 with the **pwm\_0** output.

49.1 Go to the **Sources** pane, right-click and select **Add Sources**. Then select **Add or create constraints**. Click **Next**.

49.2 In the next window, select **Create File**.

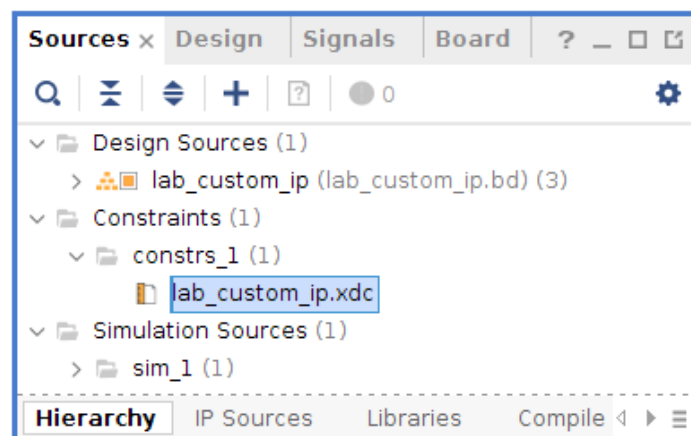


49.3 In the **Create Constraints File** window, type **lab\_custom\_ip** as File name.



49.4 Click **OK**, then **Finish**.

49.5 In the **Source** pane the **lab\_custom\_ip.xcd** file should be underneath **Constraints**.

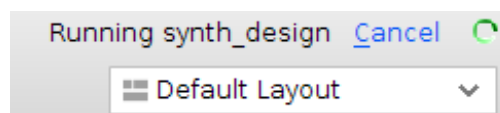


**49.6** Double-click on the lab\_custom\_ip.xdc file to open to edit it. Type in the following lines:

```
# ----- 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 #
```

**49.7** Save the file.

**50.** Let's do **Run Synthesis** by clicking on the **Run Synthesis** task in the **Flow Navigator** pane. Check on the upper right corner whether the tool is running.

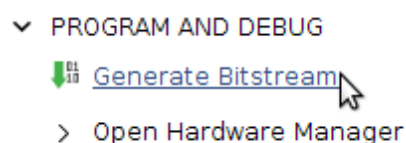


**51.** Once synthesis is successfully done, do **Run Implementation**.

**52.** In the **Implementation Completed** window select **Open Implemented Design** and click **OK**.

**53.** Once the implemented design opens, it can be seen in the **Device** tab how the logic is been placed and routed in the Zynq device.

**54.** Last step in the Vivado environment is to generate the respective bitstream. Do **Generate Bitstream** from the **Flow Navigator** pane.



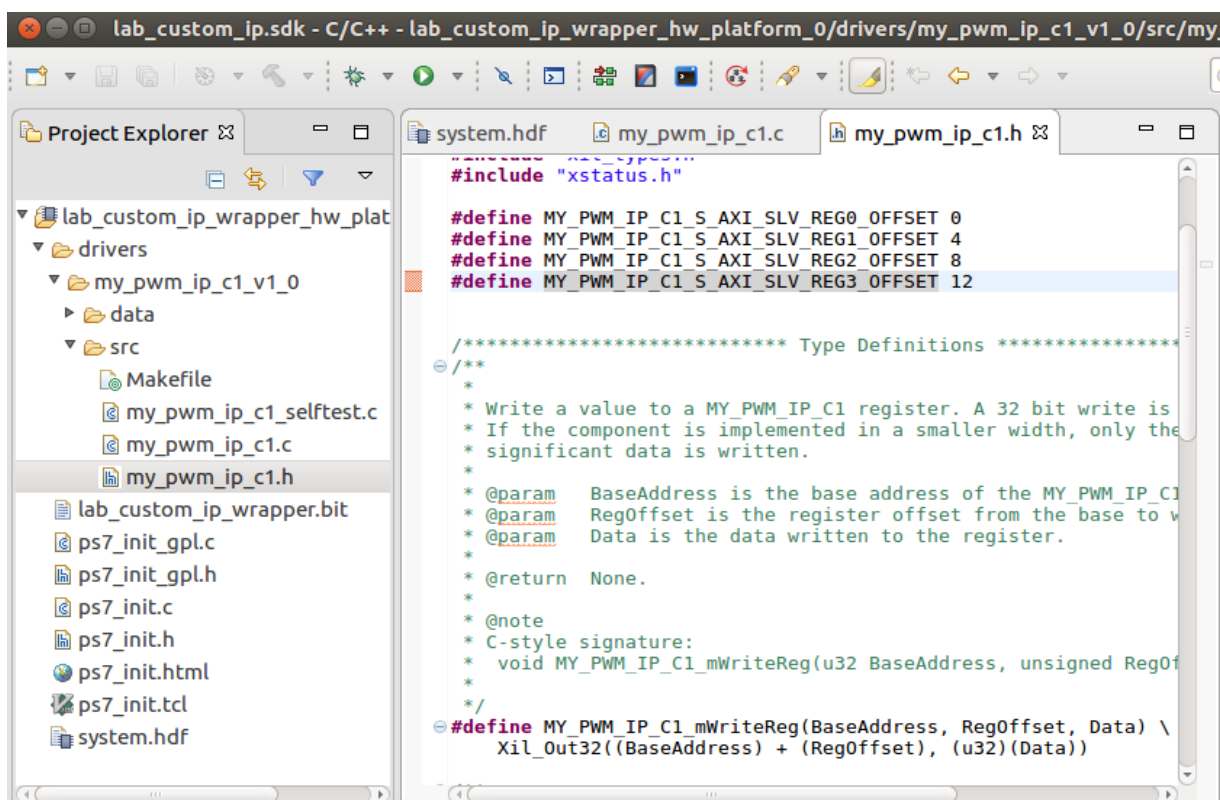
**55.** After finishing generating the bitstream file, close the message window and do **File > Export > Export Hardware**.

**56.** In the **Export Hardware** window check **Include Bitstream** option and click **OK**.

## Create an Application Project for the PWM IP in SDK

**Objective** Create a new SDK project and write the 'C' code to control the PWM duty cycle.

1. Do **File > Launch SDK**, and click **OK** in the **Launch SDK** window (leave the options as default).
2. When SDK opens, it will show the hardware files generated from Vivado environment and to be used in SDK. Among these files, there are the driver files, that hold the 'C' functions to be used to read and write to/from the memory (registers) locations. Let's open the header file, **my\_pwm\_ip\_c1.h**, underneath the **drivers→my\_pwm\_ip\_c1\_v1\_0→data→src** folder. The driver code was generated automatically when the IP template was created. The driver includes high level functions which can be called from the user application and it implements the low level functionality used to control the peripheral.



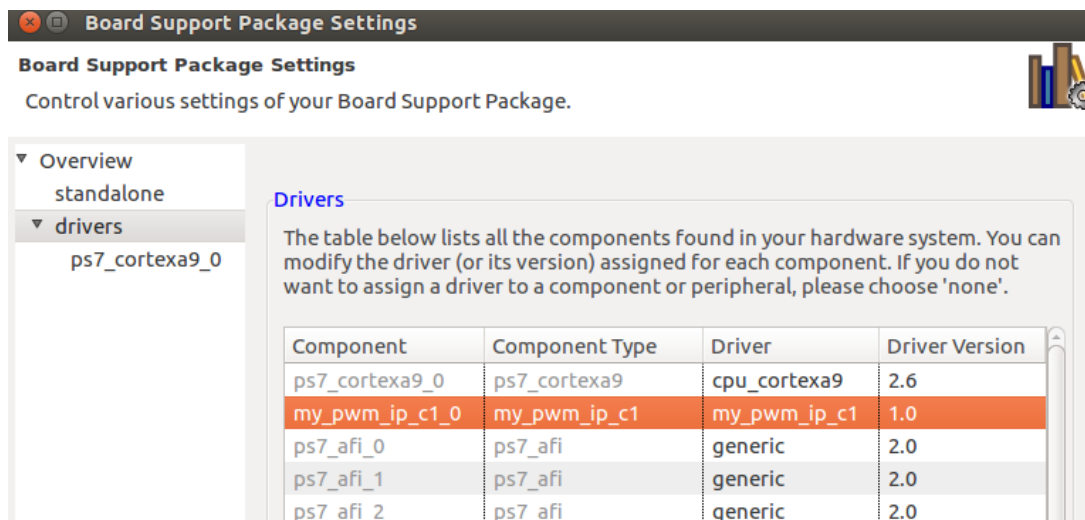
3. In the first lines, there are definitions of the offset for each register, whereas in the middle of the file you can find the function **MY\_PWM\_IP\_C1\_mWriteReg**, that we will use to write the register of the **PWM IP** (register `slv_reg0`).

For this driver, you can see the macros are aliases to the lower level functions **Xil\_Out32()** and **Xil\_In32()**. The macros in this file make up the higher level API of the **pwm\_ip** driver. If you are writing your own driver for your own IP, you will need to use low level functions like these to read and write from your IP as required. The low level hardware access functions are wrapped in your driver making it easier to use your IP in an Application project.

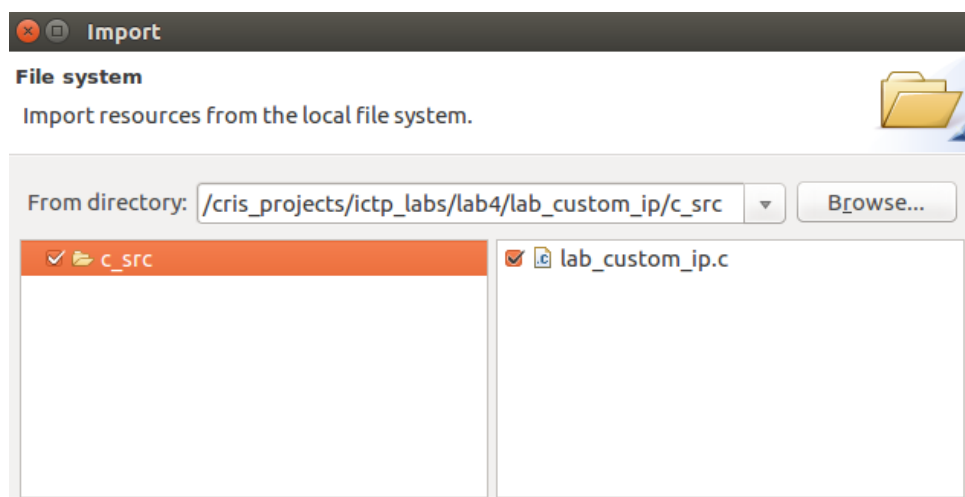
4. In SDK, select **File -> New -> Application Project**.
5. Enter **lab\_custom\_ip** as the **Project Name**. Leave all the other options with the default value. Then click **Next**.

The screenshot shows the 'New Project' dialog box in an IDE. The title bar says 'New Project'. Below the title bar, it says 'Application Project' and 'Create a managed make application project.' There is a folder icon with a 'G' on the right. The 'Project name' field contains 'lab\_custom\_ip'. The 'Use default location' checkbox is checked. The 'Location' field shows a path: '/cris\_projects/ictp\_labs/lab4/lab\_custom\_ip/lab\_custom\_ip'. There is a 'Browse...' button next to it. The 'Choose file system' dropdown is set to 'default'. The 'OS Platform' dropdown is set to 'standalone'. Under 'Target Hardware', the 'Hardware Platform' dropdown is set to 'lab\_custom\_ip\_wrapper\_hw\_platform\_0' with a 'New...' button next to it. The 'Processor' dropdown is set to 'ps7\_cortexa9\_0'. Under 'Target Software', the 'Language' section has 'C' selected with a radio button and 'C++' with an unselected radio button. The 'Compiler' dropdown is set to '32-bit'. The 'Hypervisor Guest' dropdown is set to 'N/A'. The 'Board Support Package' section has 'Create New' selected with a radio button and 'lab\_custom\_ip\_bsp' in the text field. There is also an unselected radio button for 'Use existing'. At the bottom, there is a help icon, and four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

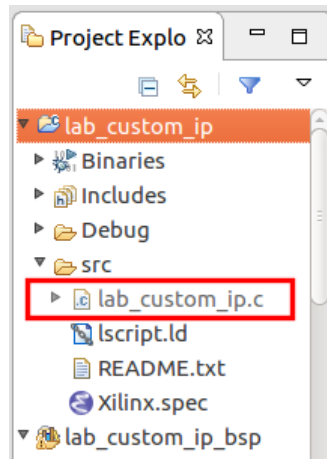
6. In the **Template** window, select **Empty Application** and click **Finish**.
7. Before going to write the 'C' code we need to be sure that the driver generated by Vivado is also available in the Board Support Package.
  - 7.1 In the **Project Explorer** pane, highlight the **lab\_custom\_ip\_bsp**, then right-click and select **Board Support Package Settings**.
  - 7.2 In the **Board Support Package Settings** window, select **Board Support Package Settings** vers (under **Overview**) and check that the component **my\_pwm\_ip\_c1\_0** be assigned the driver **my\_pwm\_ip\_c1**.



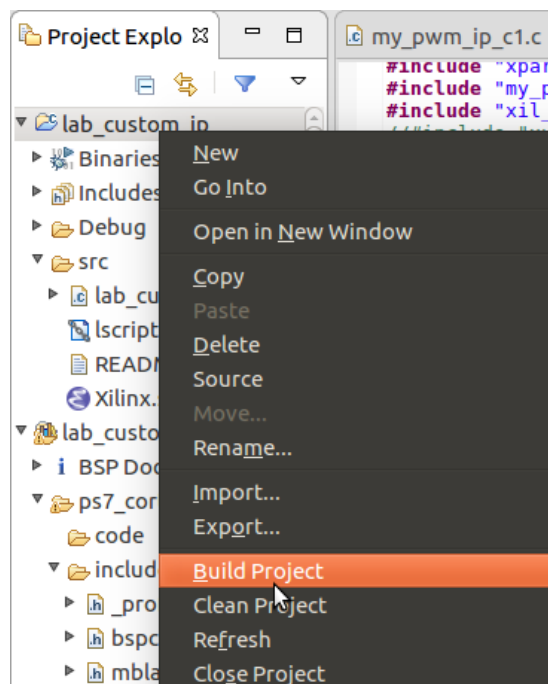
8. In the Project Explorer pane, most left pane, expand **lab\_custom\_ip**. Highlight the **src** folder, then right-click and select **Import**.
9. In the **Select** window, expand **General** category and double-click on **File System**.
10. Now browse to the directory **C:/.../ictp\_labs/lab4/c\_src** folder and click **OK**.



11. Check the `c_src` folder to be able to import the `lab_custom_ip.c` file.
12. Click **Finish**.
13. The `lab_custom_ip.c` file should be shown now in the project.



14. Take a look at the 'c' file. You need to complete the sentence to write the duty cycle value to the register 0.  
Hint: use the drivers created by the tool.
15. After completing the previous step, do right click on `lab_custom_ip` project and select **Build Project**.



16. Check for any error message.

---

## Download and Verify

---

### Objectives

Connect the board with the two micro-usb cables and power it ON. Set up the serial communication, configure the PL and execute the application in the PS.

1. Connect the two micro USB cables between the ZedBoard and the PC.
2. Power-on the board.
3. Configure a serial communication software (Tera Term, Putty, etc.).
4. Configure the FPGA by either selecting **Xilinx Tools -> Program FPGA** or by pressing the icon. Be sure to select the right **Hardware Platform** (in this case: **lab\_custom\_ip\_wrapper\_hw\_platform\_0**), and the right bitstream (**lab\_custom\_ip\_wrapper.bit**).
5. Click **Program** and wait for the blue LED on the ZedBoard to turn on, as a sign of successfully programmed the device.
6. Next, select **lab\_custom\_ip** in **Project Explorer** pane, right-click and select **Run As > Launch on Hardware (GDB)** to download the application, execute the **ps7\_init**, and run the **lab\_custom\_ip.elf**.
7. Check the **LED0** on the **ZedBoard**, it will go from full brightness to almost turn-off. This dimming is due to that the 'C' for-loop in which the value of the PWM duty cycle is changing after certain delay.
8. Try changing the 'C' code to play with the LED dimming.
9. You have successfully completed this part of the lab, you can now power off the board.

## Pros and Cons Case 1

---

### Pros:

- Very easy and quick to insert the VHDL code in the AXI interface module.

### Cons:

- Useful for very simple VHDL/Verilog code
- VHDL/Verilog code is not portable



## Creating a Custom IP Core – Case 2

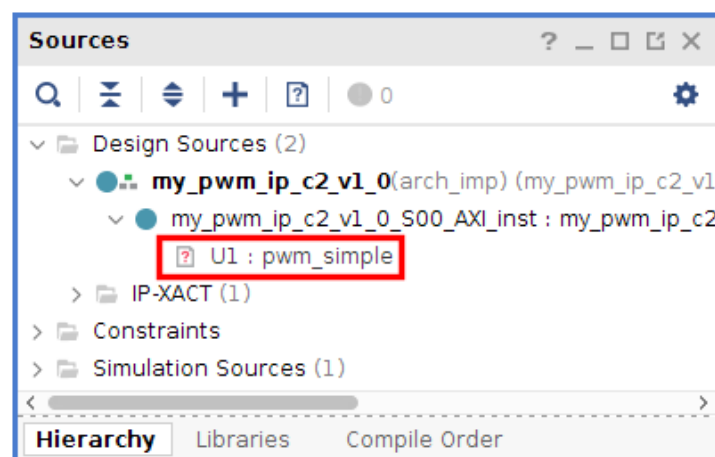
**Objective** Create IP, Vivado project and SDK application of a PWM IP created using a different approach: instantiating the IP in the VHDL AXI interface component.

1. Please follow all the steps detailed in Creating a Custom IP Core - Case 1, until you reach the point 13.4 (Adding the **pwm\_pr process**). In all the names/directories, etc.; that are needed for the IP creation, replace *case1* or *c1* for *case2* or *c2*.
2. Once you reach the point 13.4, you will 'include' the **pwm\_simple** component (see page 7) in the AXI interface VHDL component by using a component instantiation.

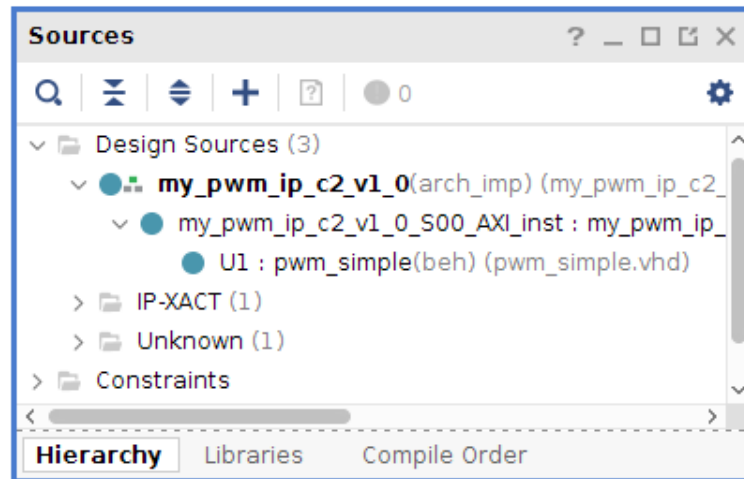
2.1. In the `my_pwm_ip_c1_0_S_AXI.vhd` file, copy and paste, at ~line 387, the following lines of code

```
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
  );
```

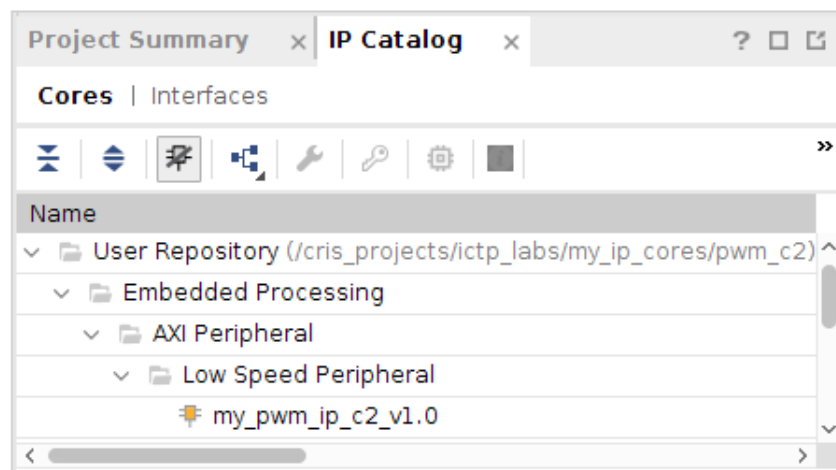
3. Then, keep working from point 14 to point 18. Remember to replace *case1* or *c1* for *case2* or *c2* as needed.
4. At this point you should have a red question mark regarding the **pwm\_simple** file in the **Source** pane.



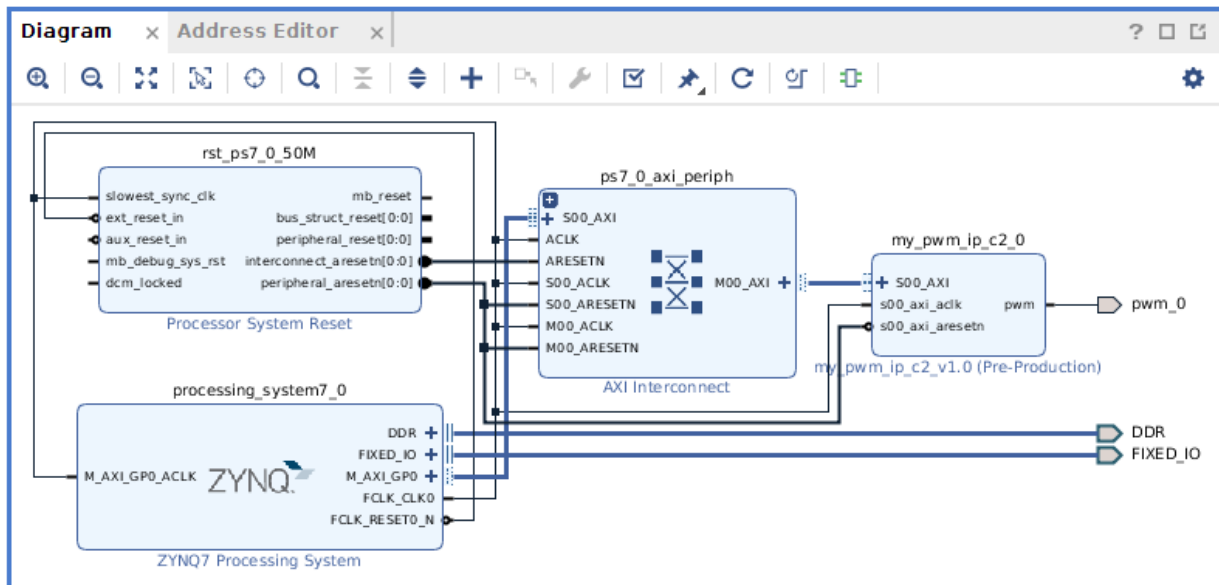
5. This is due to the fact that *we do need* to import the **pwm\_simple** VHDL component file in our project. The compiler is looking for that component, it does not find it, then it generates the error message. So, go to the **Sources** pane, right click and select **Add Sources**. Then select **Add or Create Design Sources**. Click **Next**. Select **Add Files**. Browse to `c:/../lab4/rtl_src` and select **pwm\_simple.vhd**.
6. Once you finish adding the file the red question mark should disappear and now the hierarchy should be complete.



7. Let's keep working now from point 20 until the end of that section (point 24.11).
  8. Next, you also need to add this new IP to the Vivado Repository and then create the respective Vivado project. Hence, please follow all the steps explained in the section Adding the Repository to Vivado - Creating a Vivado Project with the PWM IP . For the Vivado project use as project name: **pwm\_custom\_ip\_c2**.
- 8.1. After finishing adding the IP to the repository, you should see something like this:



8.2. After finishing creating the block diagram, you should have something like this:



9. After completing previous step (step 8) , let's create the SDK project as it was explained in Create an Application Project for the PWM IP in SDK . Use as SDK project name: **pwm\_custom\_ip\_c2**.

9.1. You should be able to use the same 'C' code used before, **lab\_custom\_ip.c**. It can be find in **c:/../ictp\_labs/lab4/c\_src/**. However, you will need to use the right definitions for the base address, the register offset, etc. This details can be obtained from the **my\_pwm\_ip\_c2.h** and the **xparameters.h** files.

10. After completing step 8, you have to configure the FPGA and run the application as it was explained in Download and Verify. Check for the brightness of **LED0**.

## Pros and Cons Case 2

### Pros:

- Portability of the code. The component instantiated in the AXI Interface can be reused in any other project.
- Useful for complicated / advanced systems.

### Cons:

- Still this code is not fully controllable by the processor

---

## Creating a Custom IP Core – Case 3

---

**Objective** Create IP, Vivado project and SDK application of a PWM IP. The PWM IP have 5 registers, some of them reading registers, some of them writing registers.

The VHDL code for the PWM is a code the make the PWM configurable by the processor. That's one of the reason of having several registers. Besides of being configurable other register, for instance, report the status of the IP at the moment of being read the register.

Reviewing how the IP Cores communicate with the ARM Processor, remember that is done by an AXI Bus, and for the Processor the IP Core is seeing as a memory location. The memory locations used to access to the IP Core are better known as registers. Hence, the IP Cores have a set of registers, which can be used for different purposes. The amount of registers to be used in an IP Core is going to depend on the IP Cores itself. We have seen in the previous examples that only one register was necessary for the case 1 and case 2. However, in this case, case 3, there will be used more registers. These registers are writable and readable registers, so the ARM processor can send and get data from the IP Core. The amount of registers are defined by the designer of the IP Core. In this case there were defined 5 registers, therefore there will be fives memory addresses associated to each register. The registers inside the wrapper have a name defined by the tool as it's `slv_reg $n$` , where  $n$  is the register number.

<code>slv_reg0</code>
<code>slv_reg1</code>
<code>slv_reg2</code>
<code>slv_reg3</code>
<code>slv_reg4</code>

However, in the VHDL code of IP Core, the name of the registers can be any. In this particular case the registers in the wrapper are associated with the following input or output register in the VHDL Code.

<u>slv_reg0</u>	reg0_control	in
<u>slv_reg1</u>	reg1_status	out
<u>slv_reg2</u>	reg2_pwm_dc_value	in
<u>slv_reg3</u>	reg3_ip_version	out
<u>slv_reg4</u>	reg4_pwm_dc_value	out

Below is the heading of the VHDL in which the different functions of the registers is detailed.

```

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 --      |      |      |      |      |      |      |      |
21 --      |      |      |      |      |      |      |      |
22 --      |      |      |      |      |      |      |      |
23 -- ----- Register 1: Status Register -----
24 -- bit31 | ... | ... | ... | ... | ... | bit 1 | bit 0 |
25 --      |      |      |      |      |      |      |      |
26 --      |      |      |      |      |      |      |      |
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 --

```

As it can be understood from the explanation in the VHDL comments, each register has a specific purpose and depending on the value that is write in the register the IP Core functionality can/could change. For instance, writing a '0' in bit 1 of the **Register 0**, it will cause that the PWM IP Core will not produce any output since the core is disable. Likewise, for instance, if we write a '1' in bit 2, the output of the PWM will be inverted.

**Register 1**, is a readable register that report to the ARM Processor some values that could be of interest.

In **register 2** the ARM Processor will write the value to be used by the VHDL code to generate the duty cycle.

**Register 3** is a register that can be read by the ARM Processor to find out which is the version of the PWM VHDL Code that is running.

**Register 4** holds the current value being used as duty cycle. It can be read by the ARM Processor.

In the VHDL code some aliases were defined to facilitate the understanding of the code.

Take a moment to try to understand the VHDL code.

1. Please follow all the steps detailed in Creating a Custom IP Core - Case 1, until you reach the point 13.4 (Adding the **pwm\_pr process**). In all the names/directories, etc.; that are needed for the IP creation, replace *case1* or *c1* for *case3* or *c3*.

*IMPORTANT 1: for this case you will need 5 registers, so when generate the AXI interface the amount of registers should be set to 5.*

*IMPORTANT 2: in this VHDL code there is one output more, therefore besides of the **pwm** output you have to declare the interrupt output in the same place that you do for the **pwm** output.*

*IMPORTANT 3: use the constraint file of case 2, but add the interrupt output.*

2. Once you reach the point 13.4, you will 'include' the **pwm\_complete** component in the AXI interface VHDL component by using a component instantiation.

2.1. In the `my_pwm_ip_c3_0_S_AXI.vhd` file, copy and paste, at ~line 387, the following lines of code

```
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
  );
```

In the instantiation statement only the writable registers are referred. On the other side the readable registers have to be first, associated with an internal register (signal) in the instantiation statement. This internal signal have to be declared in the declarative part of the architecture. Then in the reading process of the AXI wrapper associate the internal signal with the respective register. For instance, in the instantiation statement associate **reg1\_status** with

`reg1_status_internal`, then in ~line 357, replace `slv_reg1` with `reg1_status_internal`. Do the same for other readable register.

3. keep working from point 14. Remember to replace *case1* or *c1* for *case3* or *c3* as needed.
4. Use the 'C' code used in case 2 as base for the 'C' code for this code, but:
  - 4.1. When writing the 'C' code remember to configure first the IP `reg0_control` to be able to make the IP works.
  - 4.2. Write a very simple 'C' statements just to check the IP is working.
  - 4.3. Then use the different options in the control register to change the functionality of the IP Core.
  - 4.4. Read the registers that can be read, and print their value.

*Note: interrupt is not implemented in this case (see challenge below).*

---

## Challenge

---

Even though the VHDL code has an interrupt request output, for this lab it has not been tied to the interrupt input of the PS. However, as challenge you can connect the interrupt output to the interrupt input in the PS following the steps done in the Interrupt lab. Then, write the 'C' code that will attend the interrupt request from the PWM IP Core.



---

## Conclusion

---

There different cases have been implemented for generating a custom IP Core. The three have different advantages and disadvantages, you should be able to pick the one that fit your needs.