

---

# *LABORATORY 6*

## *Zynq Design*

---

*Custom AXI-Slave*

*Adding Custom IP Cores*

*Hardware - Software*

# Adding Custom IP to the System

## 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.

## 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
- Modify the functionality of the custom IP block
- Add the custom peripheral to your design
- Add pin location constraints
- Add block memory to the system

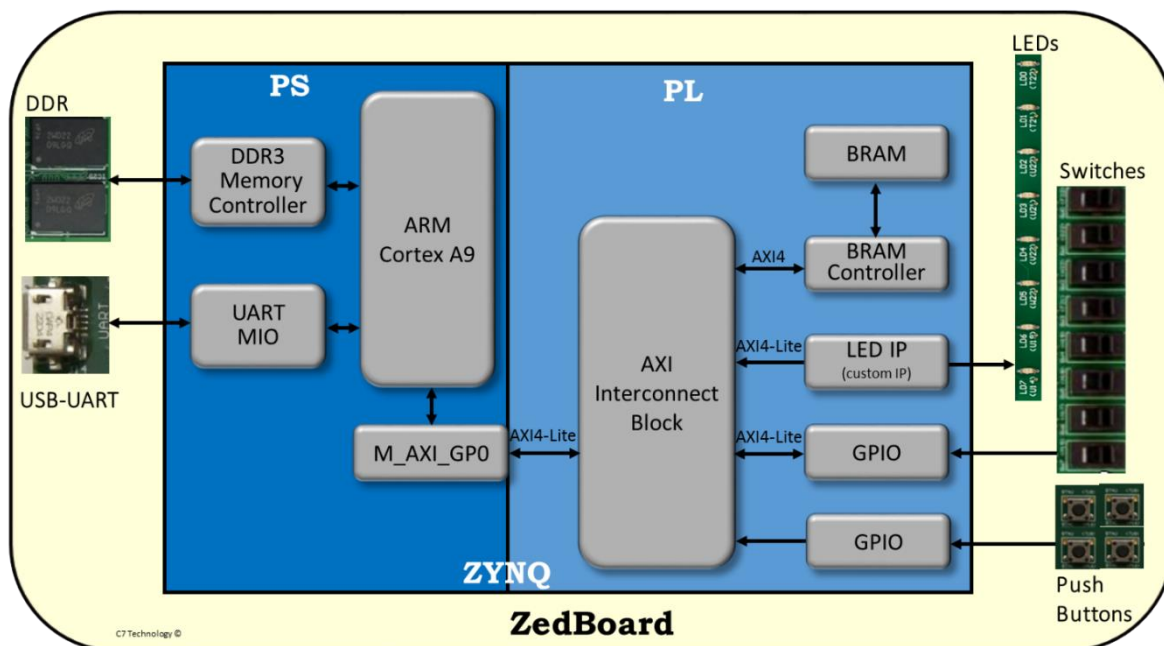
## Procedure

This lab comprises 4 primary steps: You will use a peripheral template to create a custom peripheral IP, package the IP using the IP Packager utility and import, add and connect the IP into a block design.

## Design Description

You will extend the **Lab 3** hardware design by creating and adding an AXI custom peripheral (refer to LED\_IP in figure below) to a system that will control the LEDs on the ZedBoard.

**Note:** in LAB3 you used an AXI GPIO block to control the LEDs, in this Lab that block will be replaced by your own AXI based LED controller block.

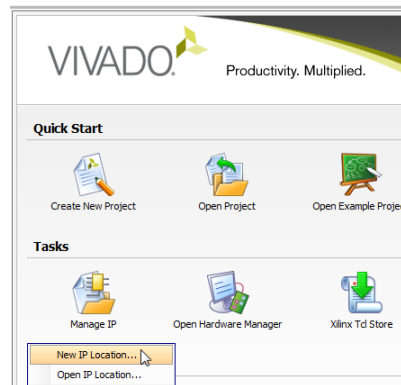


## Create a Custom IP

## Step 1

**1-1. Objective:** Use the provided *axi\_lite* slave peripheral template and the custom IP VHDL source code to create a custom IP.

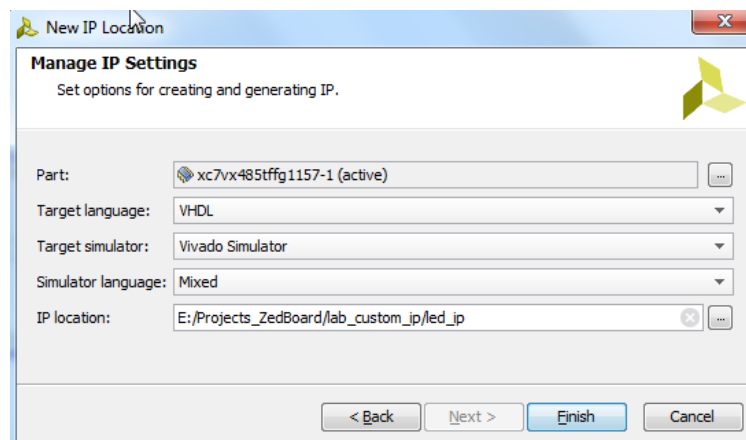
**1-1-1.** Open *Vivado* and select **Manage IP-> New IP Location**.



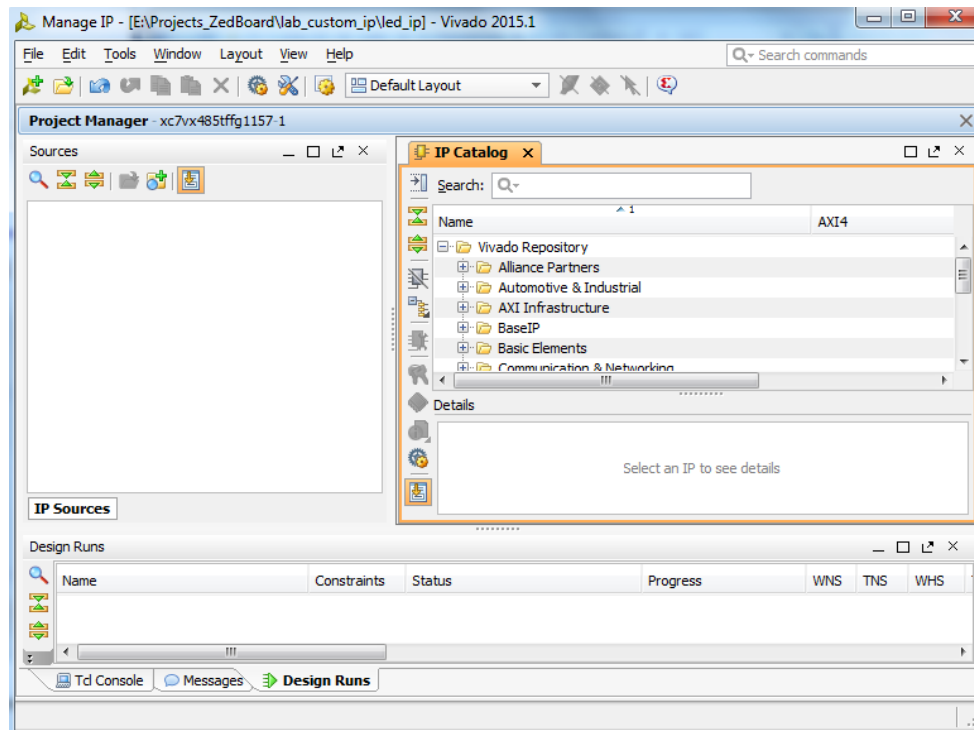
**1-1-2.** Click *Next* in the *New IP Location* window.

Select *VHDL* as the *Target Language*, *Mixed* as the *Simulator language*, and for the *IP location* type *c:\.....\SoC\_School\lab\_custom\_IP\led\_ip* and click **Finish** (leave other settings as defaults and click *OK* if prompted to create the directory). A Virtex 7 part is chosen by default for this project, but later compatibility for other devices will be added to the packaged IP.

**Note:** As *IP Location* select the directory that you will use for this lab. It is a good idea to create a subdirectory just for this IP.

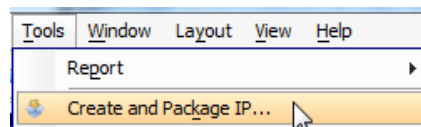


**1-1-3.** After clicking *Finish* the *Vivado Manage IP GUI* will open. This window looks a little different from the window we have been working with so far (Vivado GUI).



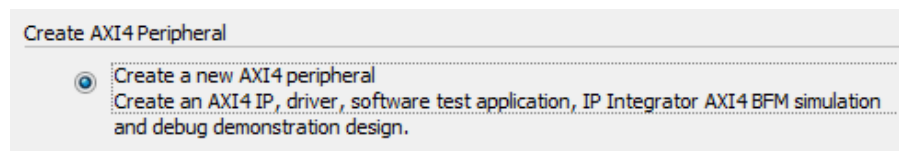
### 1-2. **Objective:** Run the Create and Package IP Wizard

1-2-1. Select **Tools > Create and Package IP**.



1-2-2. In the new window, click **Next**.

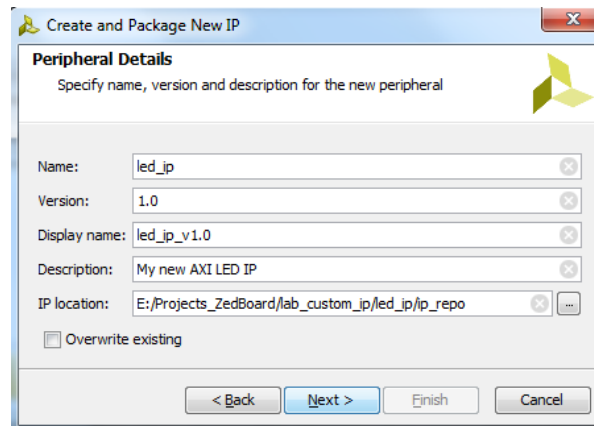
1-2-3. Select **Create AXI4 peripheral**, specify the **IP Definition location** as `{labs}\led_ip` and click **Next**.



1-2-4. Fill in the details for the IP

**Name:** `led_ip`

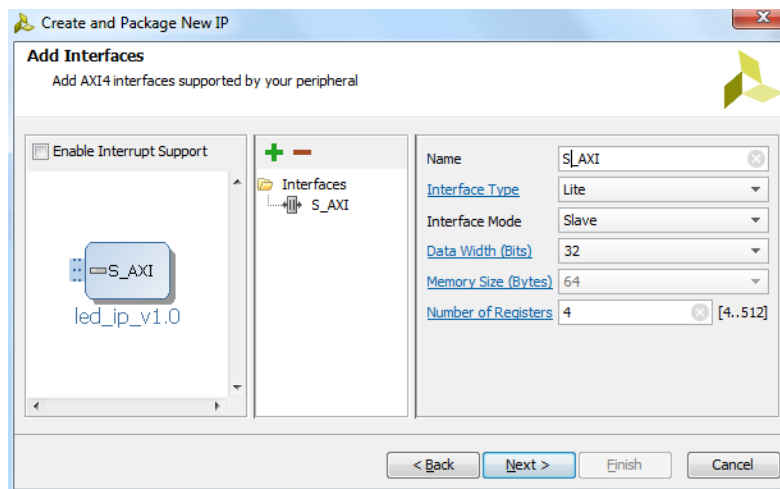
**Display Name:** `led_ip_v1.0`



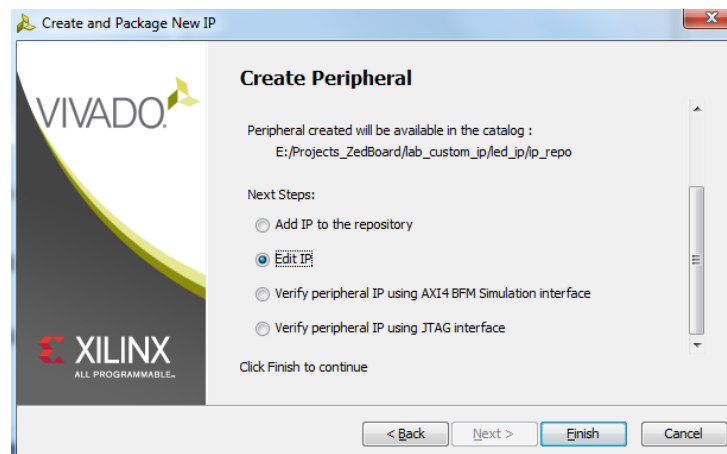
1-2-5. Click **Next**.

1-2-6. Change the **Name** of the interface to **S\_AXI** (see figure below).

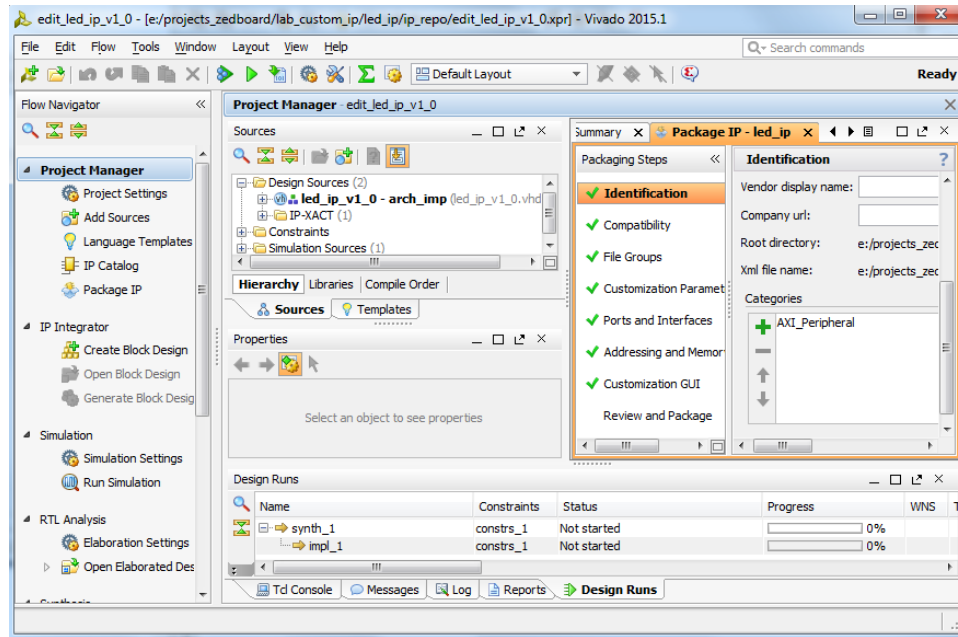
1-2-7. Leave the other settings as default, **Lite interface**, **Slave mode**, **Data Width 32**, **Registers 4**, and click **Next**.



1-2-8. In the **Create and Package New IP** window, select **Edit IP** and click **Finish**.



1-2-9. A new **Vivado IP** project is created based on the **led\_ip** IP configurations.



### 1-3. **Objective:** Create an interface to the ZedBoard LEDs

In the sources panel of the Project Manager window double-click to open the just created (by the tool) **led\_ip\_v1\_0.vhd** file.

This file is an automatically created HDL customized code with the AXI interface selected in the previous step (1-2-7). The top level file contains a module which implements the AXI interface logic, and an example design code to write to and read from the number of specified registers.

On the other side, you will create the respective output ports to be connected to the LEDs, this will be done the top level of the design (following the steps detailed below).

- 1-3-1.** In the **led\_ip\_v1\_0.vhd** file, scroll down to ~line 8 (~, indicates approximately). You should find a space provided for user *generic* parameter(s). Add the following line of code (since it is needed 8 bits to control the 8 LEDs, we define this generic constant):

```
LED_WIDTH : integer      := 8;
```

Go to line ~19 (below the comment “ - - users to add ports here - - ”) and add the line:

```
led : out std_logic_vector(LED_WIDTH-1 downto 0);
```

```

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

```

1-3-2. Insert the following at line ~54:

**LED\_WIDTH : integer :=8,**

And insert the following at line ~59

**led: out std\_logic\_vector(LED\_WIDTH-1 downto 0),**

```

52 component led_ip_v1_0_S_AXI is
53     generic (
54         LED_WIDTH : integer := 8;
55         C_S_AXI_DATA_WIDTH : integer := 32;
56         C_S_AXI_ADDR_WIDTH : integer := 4
57     );
58     port (
59         led : out std_logic_vector(LED_WIDTH-1 downto 0);
60         S_AXI_ACLK : in std_logic;
61         S_AXI_ARESETN : in std_logic;

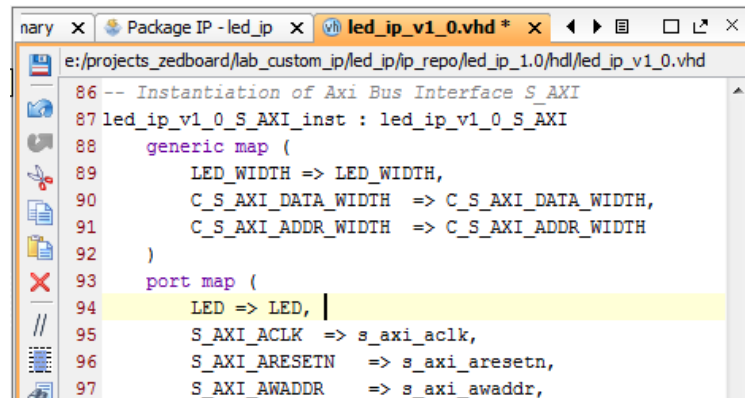
```

Insert the following at line ~89

**LED\_WIDTH => LED\_WIDTH,**

Insert the following at line ~ 94

**LED => LED,**



```

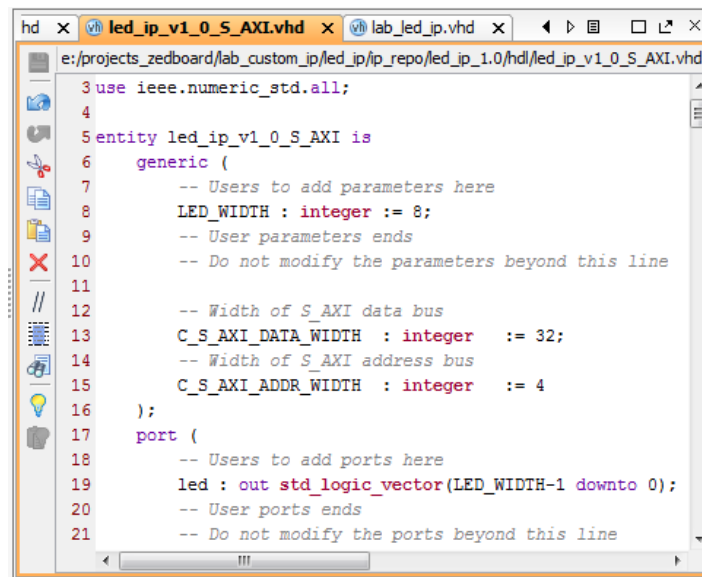
86 -- Instantiation of Axi Bus Interface S_AXI
87 led_ip_v1_0_S_AXI_inst : led_ip_v1_0_S_AXI
88     generic map (
89         LED_WIDTH => LED_WIDTH,
90         C_S_AXI_DATA_WIDTH => C_S_AXI_DATA_WIDTH,
91         C_S_AXI_ADDR_WIDTH => C_S_AXI_ADDR_WIDTH
92     )
93     port map (
94         LED => LED,
95         S_AXI_ACLK => s_axi_aclk,
96         S_AXI_ARESETN => s_axi_aresetn,
97         S_AXI_AWADDR => s_axi_awaddr,

```

1-3-3. Save the file, **File > Save File**.

1-3-4. In the sources view, expand the tree components for **led\_ip\_v1\_0**. Find and open the file **led\_ip\_v1\_0\_S\_AXI.vhd**.

1-3-5. In this file it is also necessary to add the **LED\_WIDTH** parameter and the **LED** signal. This should be done at lines ~7 and ~18.



```

3 use ieee.numeric_std.all;
4
5 entity led_ip_v1_0_S_AXI is
6     generic (
7         -- Users to add parameters here
8         LED_WIDTH : integer := 8;
9         -- User parameters ends
10        -- Do not modify the parameters beyond this line
11
12        -- Width of S_AXI data bus
13        C_S_AXI_DATA_WIDTH : integer := 32;
14        -- Width of S_AXI address bus
15        C_S_AXI_ADDR_WIDTH : integer := 4
16    );
17    port (
18        -- Users to add ports here
19        led : out std_logic_vector(LED_WIDTH-1 downto 0);
20        -- User ports ends
21        -- Do not modify the ports beyond this line

```

1-3-6. Scroll down to ~line 382 and insert the following code. This line of code represents the instantiation of the component **lab\_led\_ip**. This component will be added in the following steps, and it describes, in VHDL, the logic to control the LEDs.

```

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

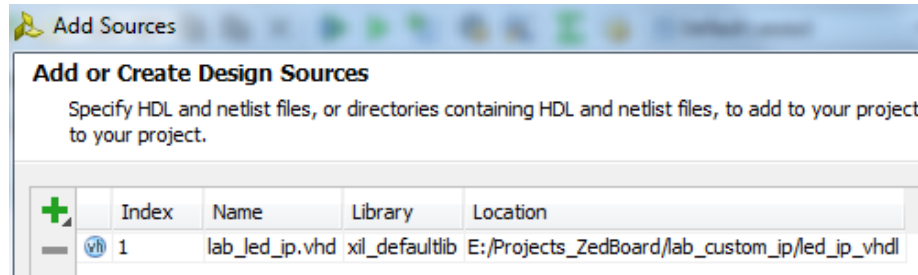
```

**Note:** for other future design, you should use here the name of the entity of the VHDL describing the functionality needed.

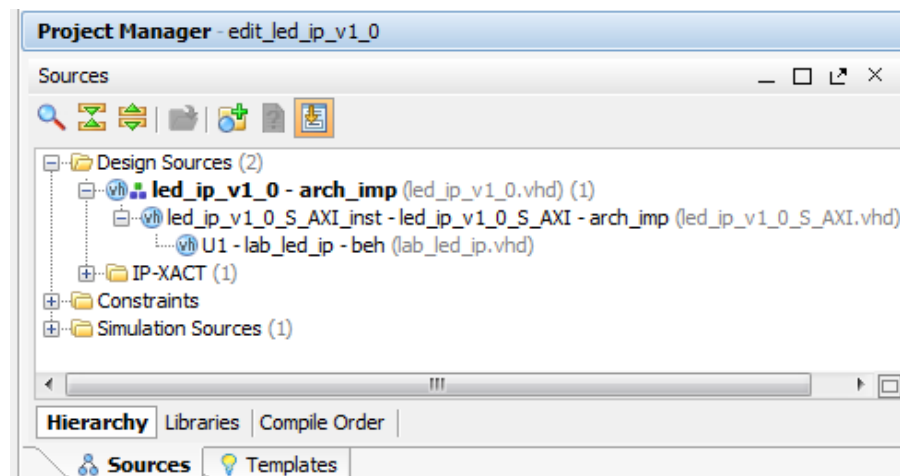


**1-3-7.** Save the file by selecting **File > Save File**.

**1-3-8.** Click on the *Add Sources* in the Flow Navigator pane, select *Add or Create Design Sources*, browse to `:\.....\SoC_School\lab_custom_IP\lab6_VHDL_File\`, select the **lab\_led\_ip.vhd** file and click **OK**, and then click **Finish** to add the file.



**1-3-9.** If all the processes detailed in the previous steps have been correctly implemented; in the *Sources* pane the hierarchy of the design should be similar to the one showed in the following figure:



You can do a double click over the **lab\_led\_ip.vhd** file to open it and to see the logic described in the VHDL code.

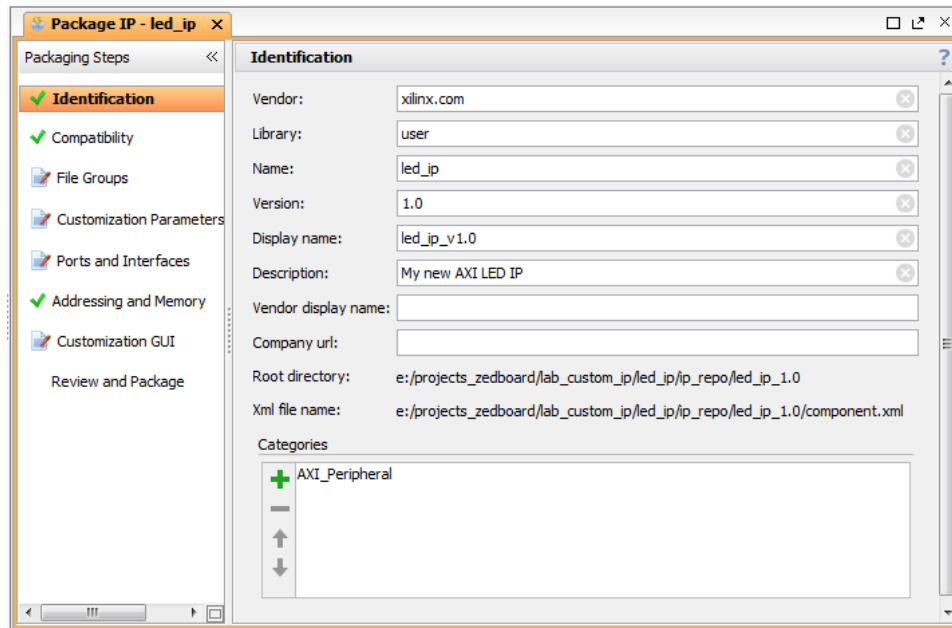
**1-3-10.** Then, click **Run Synthesis** and **Save** if prompted. This step is to check the design synthesizes correctly before packaging the IP.


Important: If VHDL code has been written by you, it is strongly advised to simulate and verify the functionality of the design before proceeding with the next steps.

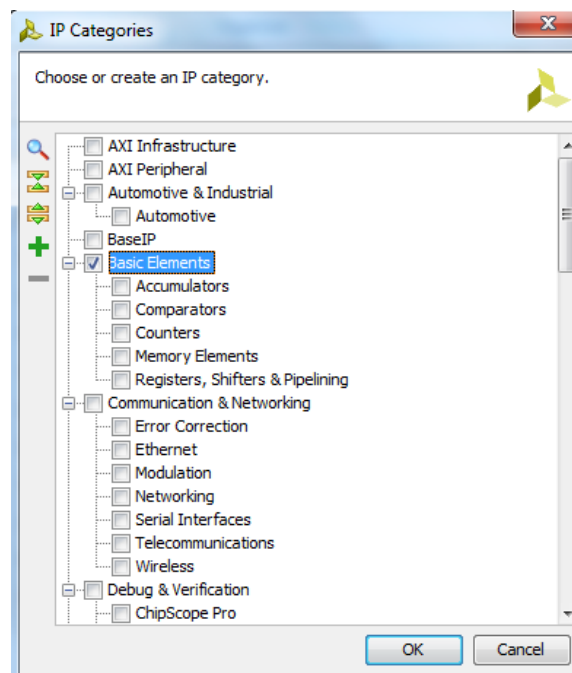
**1-3-11.** After synthesis is finished, check the **Messages** tab for any errors and correct them if necessary, before moving to the next step.

## 1-4. Package the IP

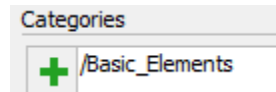
**1-4-1.** Click on the **Package IP – led\_ip** tab. Change the *Vendor* name to any name you want.



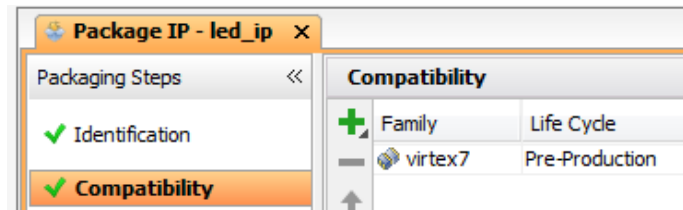
- 1-4-2.** In order to the custom IP, that you are creating, appears in the Vivado IP catalog in a particular category, the IP must be configured to be part of an specific category. To change which categories the IP will appear, click the  icon in the *Categories* window. This opens the *Choose IP Categories* window.
- 1-4-3.** For the purpose of this exercise, uncheck the **AXI Peripheral** box and check the **Basic Elements** and click **OK**.




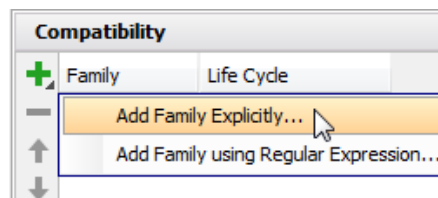
- 1-4-4.** The Categories information should be updated to `/Basic_Elements`.



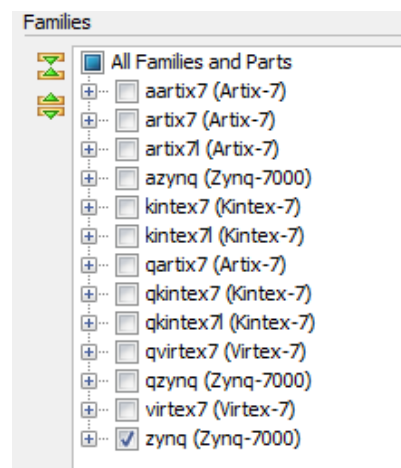
- 1-4-5.** Next, from the option on the right side of the pane, select **Compatibility**. This shows the different Xilinx FPGA Families that the IP supports. The value shown in here is inherited from the device selected for the IP project, at the beginning of this IP creation.



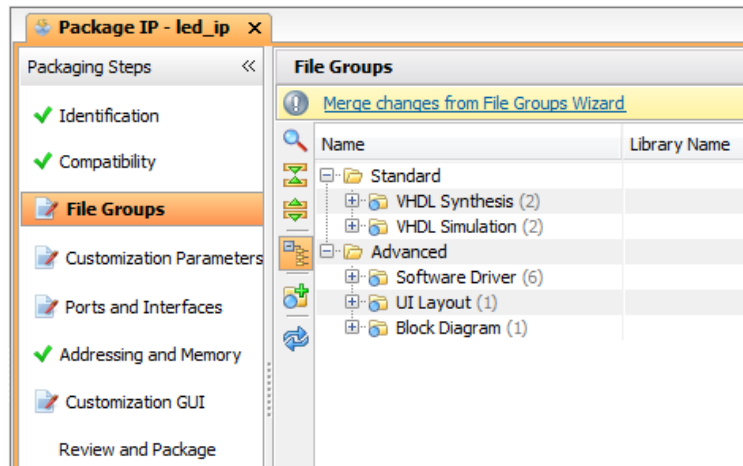
- 1-4-6.** Click in the  icon and select *Add Family Explicitly* (in Vivado versions previous to 2015, the + icon does not exist, so you need to place the mouse over the Family column, press right button and then select 'Compatibility' to get the list of available devices).



- 1-4-7.** Select the **Zynq** family since we will be using this IP on the ZedBoard, and click **OK**.

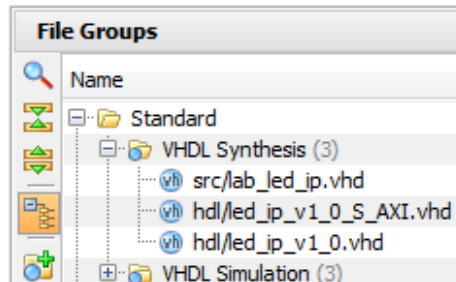


- 1-4-8.** In this package IP window, you can also customize the address space and add memory address space by using the **IP Addressing and Memory** option. For this particular case, we won't make any changes.
- 1-4-9.** Click on **IP File Groups** and click *Merge changes from IP File Groups Wizard*.

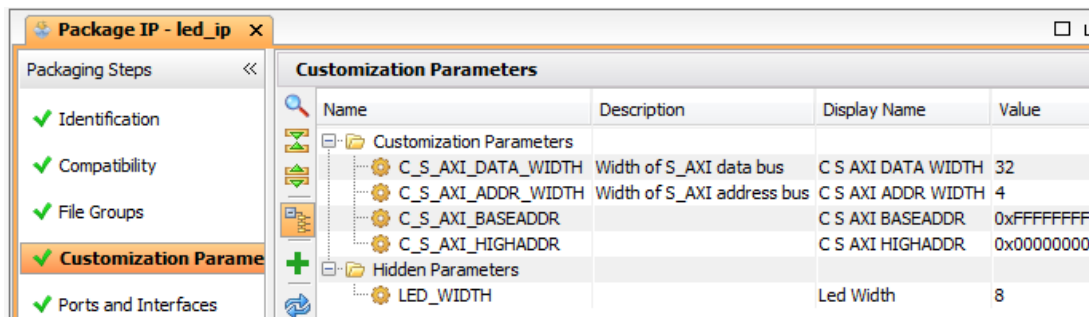


This step is to update the *IP Packager* with the changes that were made to the IP and the *lab\_led\_ip.vhd* file that was added to the project.

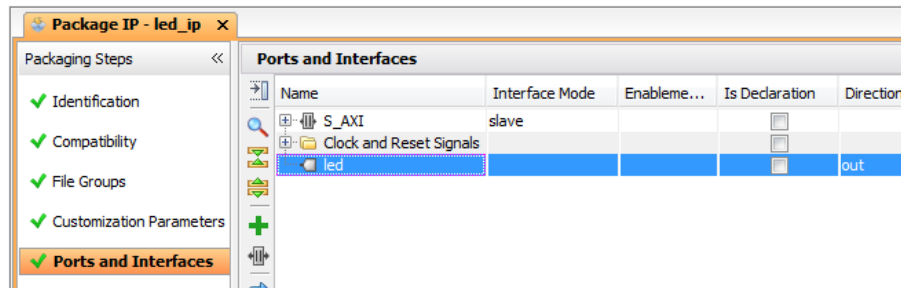
**1-4-10.** Expand *VHDL Synthesis* and notice **lab\_led\_ip.vhd** has been included.



**1-4-11.** Click on *Customization Parameters* and again click on *Merge changes from IP Customization Parameters Wizard*.



**1-4-12.** In the *Port and Interfaces* option, the *LED* port should show up.



**1-4-13.** Select *Review and Package*, and notice the path where the IP will be created.

**1-4-14.** Click **Re-Package IP**. The project will be closed when complete.

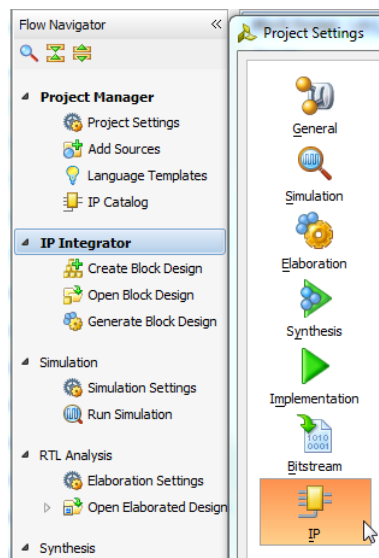
**1-4-15.** In the original *Vivado* window click **File > Close Project**.

**1-4-16.** The IP has been created and imported to the available IP in Vivado environment. In the next steps we will see how it will be used.

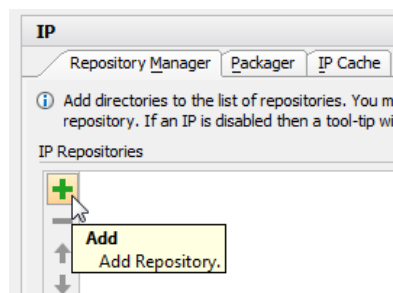
## Add Custom IP, BRAM and .xdc to a Project

## Step 2

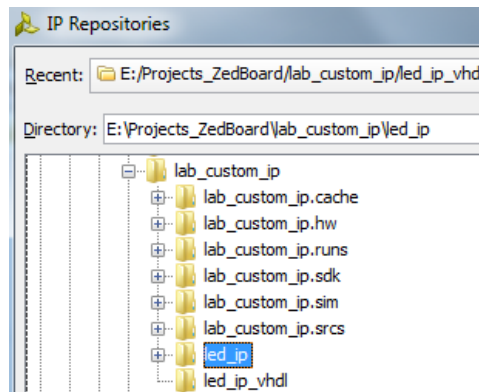
- 2-1. Objective:** Open the project you have done for *lab3\_1* and save the project as *lab\_custom\_ip*. Set *Project Settings* to point to the created IP repository. Add *led\_ip* to the design and connect to the *AXI4Lite* interconnect in the IPI. Make internal and external port connections. Establish the LED port as external FPGA pins.
- 2-1-1.** Start *Vivado* and open *lab3 (lab\_gpio\_in)*. Select *File > Save Project As*. Enter *lab\_custom\_ip* as the project name. Make sure that the *Create Project Subdirectory* option is checked, the project directory path is *c:\.....\SoC\_School\lab\_custom\_IP\* and click **OK**.
- 2-1-2.** Click *Project Settings* in the *Flow Navigator* pane. Select *IP* in the left pane of the *Project Settings* form.



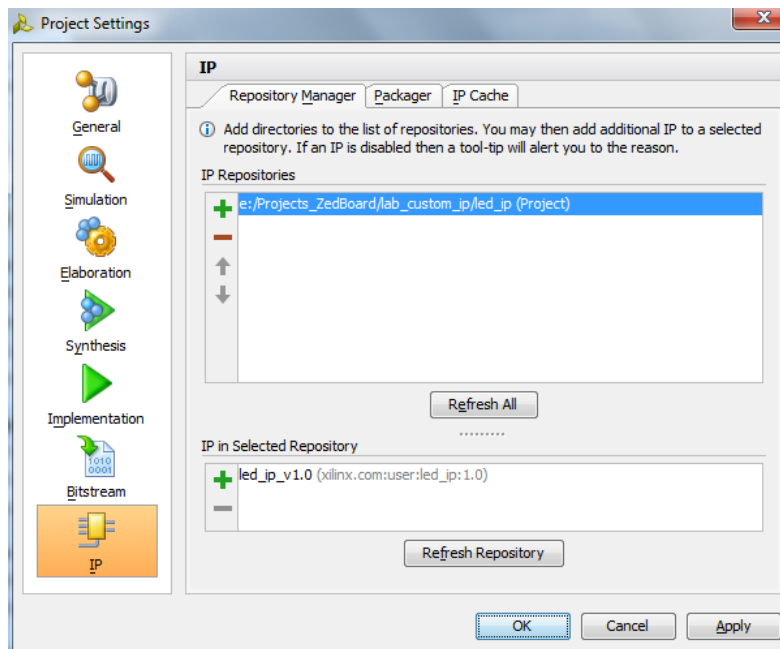
- 2-1-3.** Click on the **Add Repository...** button.



- 2-1-4.** Browse to *c:\.....\SoC\_School\lab\_custom\_IP\led\_ip* directory and click *Select* it.

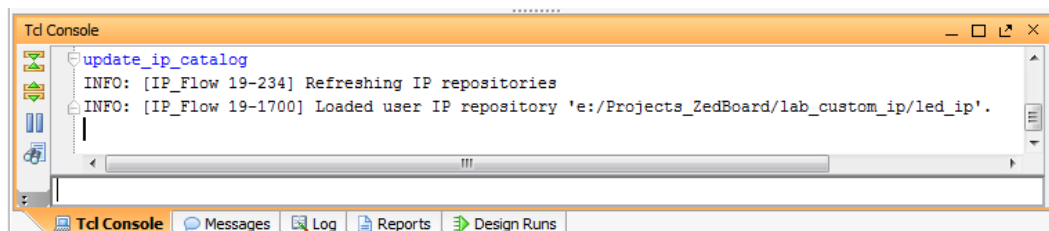


2-1-5. The **led\_ip\_v1\_0** IP should appear the IP in the Selected Repository window.




2-1-6. Click **OK** to finish the process.

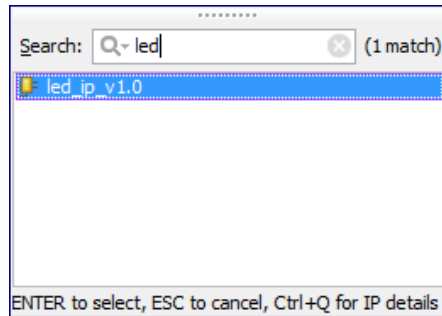
2-1-7. In the *Tcl Console* of Vivado main GUI, the following message should appear if the process was successfully completed.



2-2. **Objective:** Add **led\_ip** to the design and connect to the AXI4Lite interconnect in the IPI. Make internal and external port connections. Establish the LED port as external FPGA pins.

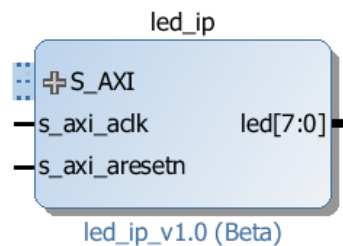
2-2-1. Click **Open Block Design** under **IP Integrator** in the **Flow Navigator** pane

- 2-2-2.** Click the **Add IP** icon  and search for **led\_ip\_v1\_0** in the catalog by typing “led” in the search field.



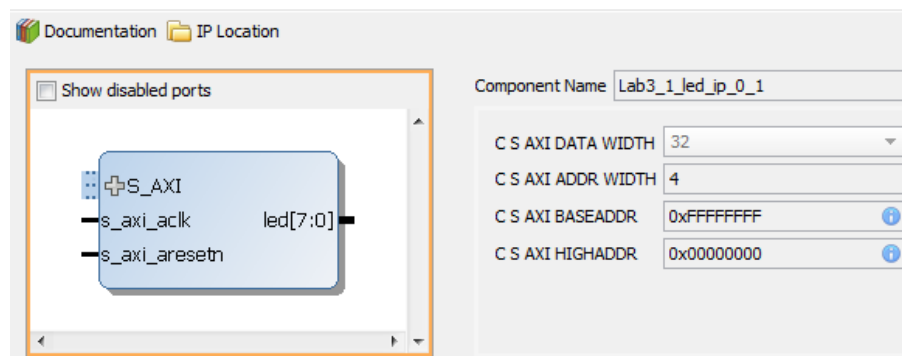
- 2-2-3.** Double-click **led\_ip\_v1\_0** to add the core of the design.

- 2-2-4.** Select the IP in the block diagram and change the instance name in the properties view from **led\_ip\_0** to **led\_ip**.

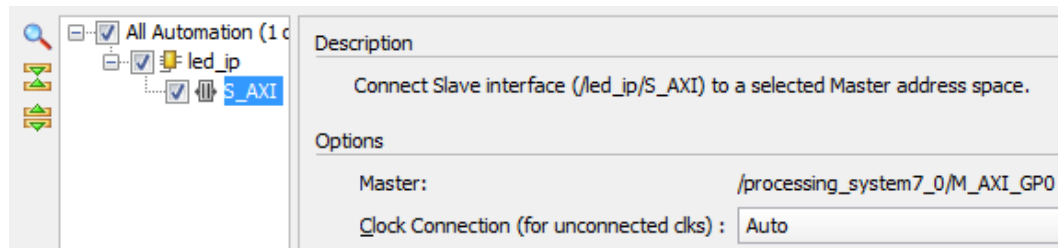



- 2-2-5.** Double click on the **led\_ip\_v1.0** block to open the configuration properties.

**led\_ip\_v1.0 (1.0)**



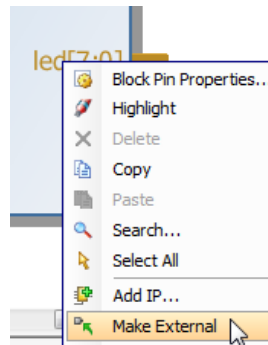
- 2-2-6.** Click on **Run Connection Automation**, select **/led\_ip/S\_AXI** and click **OK** to automatically make the connection from the AXI Interconnect to the IP.



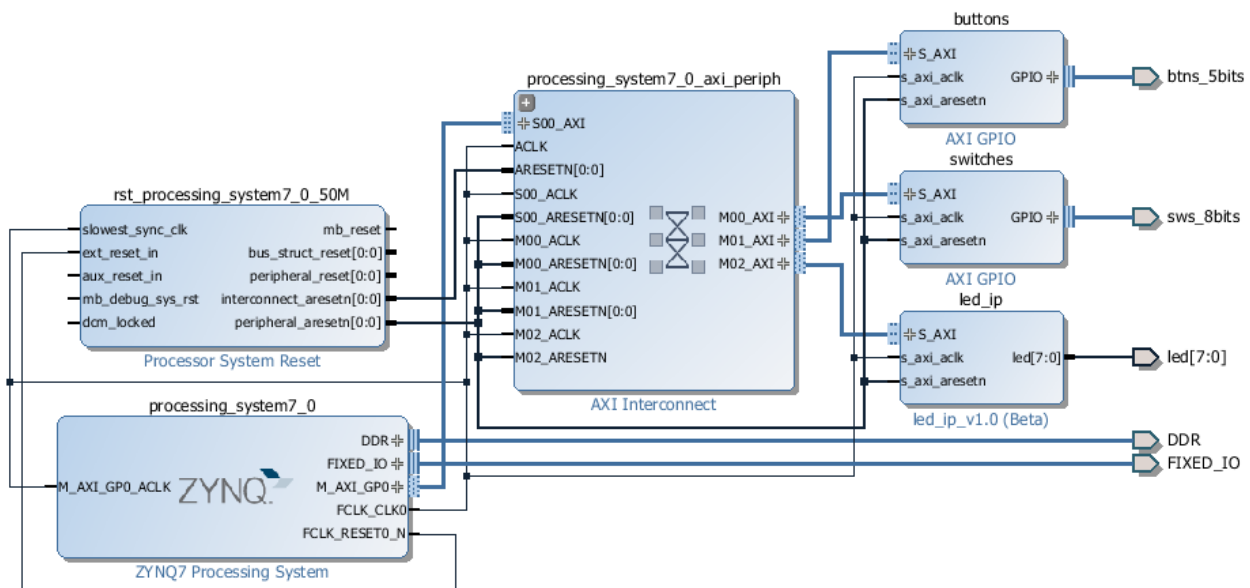
Click the regenerate button (  ) to redraw the diagram.



**2-2-7.** Select the **LED** port on the `led_ip` instance, by clicking on its pin (the color of the port should be light brown), then right-click and select **Make External**.




**2-2-8.** Now, the whole system should look like similar to the following:

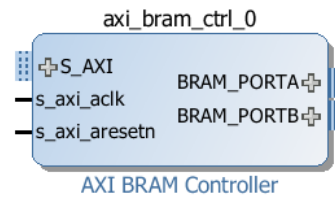
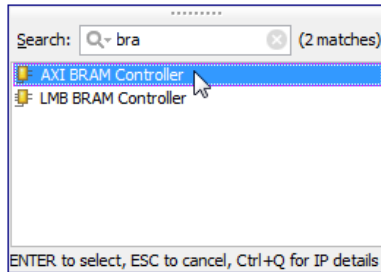


**2-2-9.** Select the *Address Editor* tab and verify that an address has been assigned to `led_ip`.

Cell	Slave Inte...	Base N...	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [ 1G ])					
switches	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
buttons	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
led_ip	S_AXI	S_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF

## 2-3. Objective: Add BRAM to the design

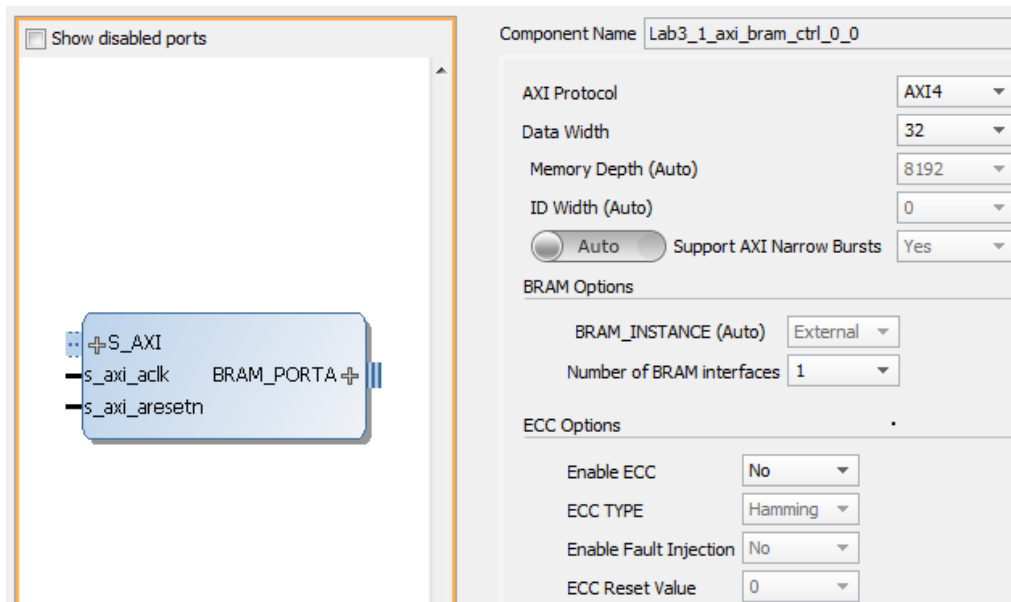
**2-3-1.** In the **Block Diagram**, click the **Add IP** icon  and search for BRAM and add one instance of the **AXI BRAM Controller**.



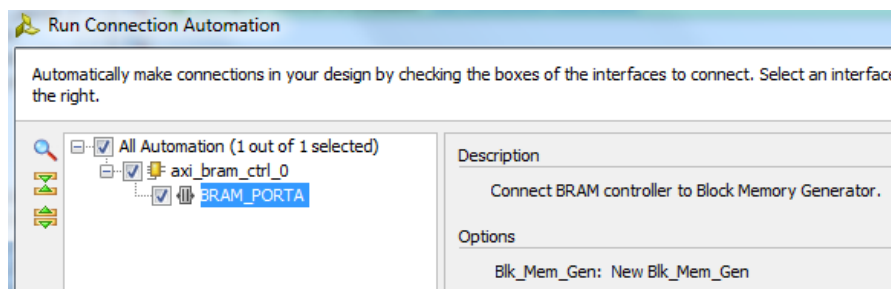
**2-3-2.** Run *Connection Automation* on *axi\_bram\_ctrl\_0/S\_AXI* (only on S\_AXI) and click **OK** when prompted to connect it to the **M\_AXI\_GP0 Master**.

**2-3-3.** Double click on the block *axi\_bram\_ctrl\_0* to customize it. Change the number of BRAM interfaces from 2 to 1 and click **OK**.

Notice that the AXI Protocol being used is AXI4 instead of AXI4Lite since BRAM can provide higher bandwidth and the controller can support burst transactions.



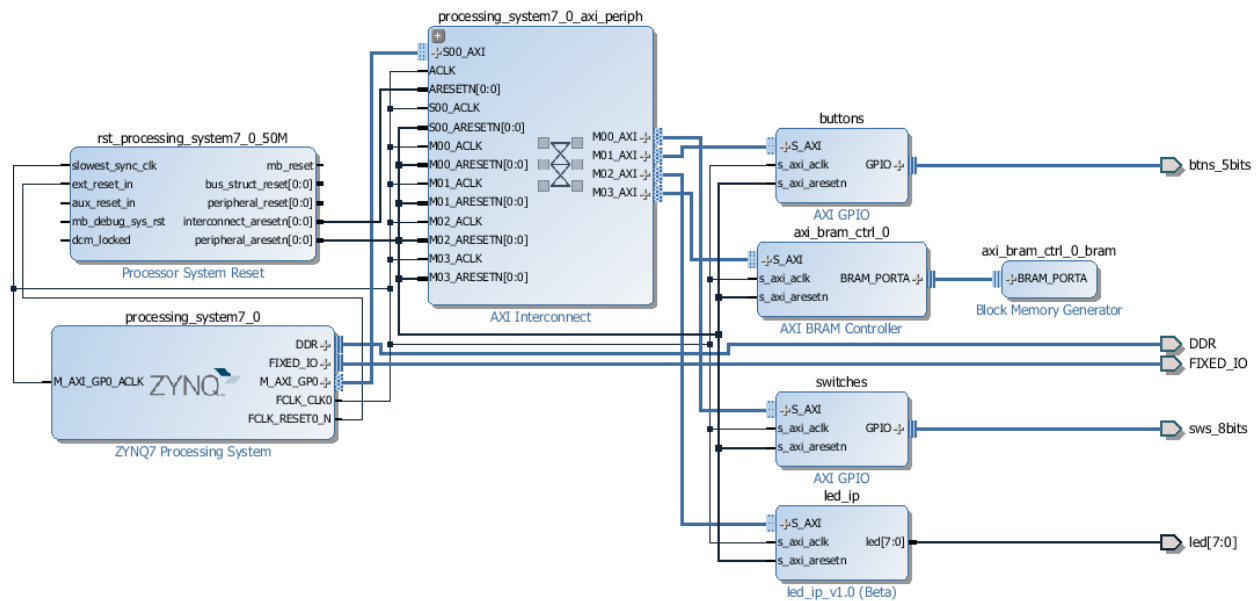
**2-3-4.** Click on *Run Connection Automation* to add and connect a **Block Memory Generator** by selecting *axi\_bram\_ctrl\_0/BRAM\_PORTA* and click **OK** (This could be added manually).



A BRAM block memory is added to the diagram.

- 2-3-5.** Validate the design to ensure there are no errors (F6), and click the regenerate button (🔄) to redraw the diagram.

The design should look similar to the figure below.



- 2-3-6.** In the Address editor, check that the range address for the BRAM block has been added.

Cell	Slave Inte...	Base N...	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [ 1G ])					
switches	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
buttons	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
led_ip	S_AXI	S_AXI_reg	0x43C0_0000	64K	0x43C0_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF

- 2-3-7.** Press **F6** to validate the design one last time.

## 2-4. **Objective:** Update the top-level wrapper and add the provided lab3\_\*.xdc constraints file.

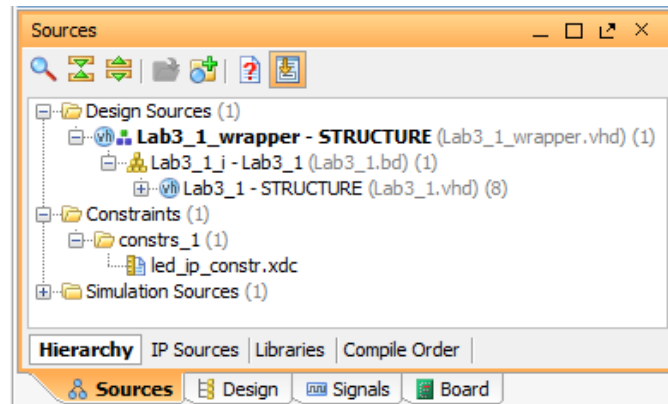
- 2-4-1.** Click **Add Sources** in the **Flow Navigator** pane, select **Add or Create Constraints**, and click **Next**.

- 2-4-2.** Click the **Add Files** button, browse to the `c:\.....\SoC_School\lab_custom_IP\sources\lab6_Constraint_File` folder, select **led\_ip\_constr.xdc**.

- 2-4-3.** Click **Finish** to add the file.

- 2-4-4.** Expand Constraints folder in the **Sources** pane, and double click the **led\_ip\_constr.xdc** file entry to see its content. This file contains the pin locations and IO standards for the LEDs on the ZedBoard.

The pin association between the Zynq device and the LEDs is obtained from the *ZedBoard Reference Manual*.



**2-4-5.** Right click on *lab3\_1.bd* and select **Generate output products**.

**2-4-6.** In the *Flow Navigator* pane Click on **Generate Bitstream** and click **Yes** if prompted to save the Block Diagram, and click **Yes** again if prompted to launch synthesis and implementation. Click **Ok** when prompted to *Open the Implemented Design*.

---

## Export to SDK and create Application Project

---

## Step 3

### 3-1. **Objective:** Export the hardware along with the generated bitstream to SDK.

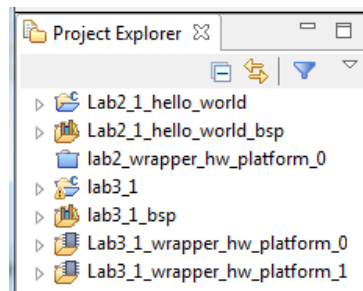
To Export the hardware, the block diagram must be open and the Implemented design must be open.

**3-1-1.** To export the hardware, both the block diagram and the implemented design must be open.

**3-1-2.** Click **File > Export > Export Hardware** (include the bitstream). If prompted, click **Ok** to overwrite existing exported file. Then click **File > Launch SDK** and click **OK**.

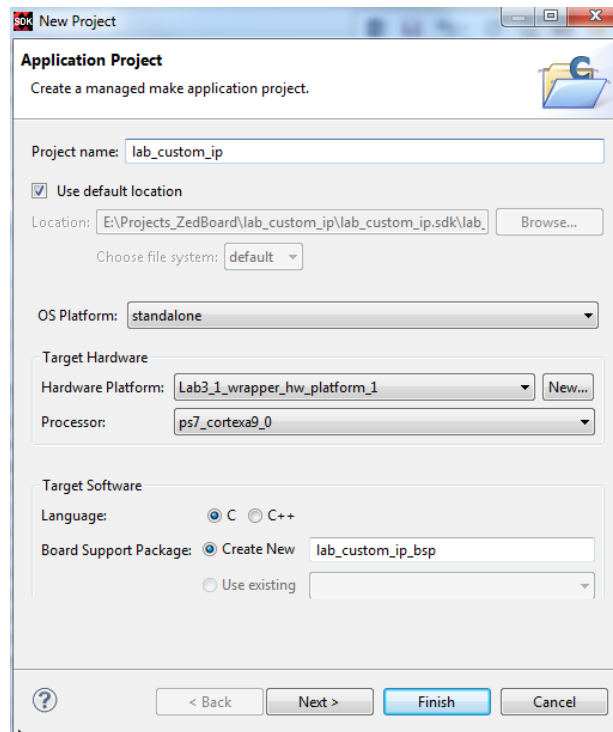
### 3-2. **Objective:** Create a new application project

**3-2-1.** In the **SDK** main window there would be a similar hardware platform to the one just created. For instance, the platform from previous lab named *lab3\_1\_wrapper\_hw\_platform\_0*, and the new one named *lab3\_1\_wrapper\_hw\_platform\_1*. This is due to the fact that the block design created for the *lab\_custom\_ip* project was based on the *lab3\_1* block design.

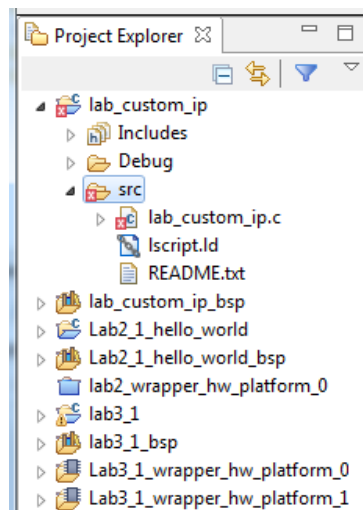


**3-2-2.** Select **File > New > Application Project**.

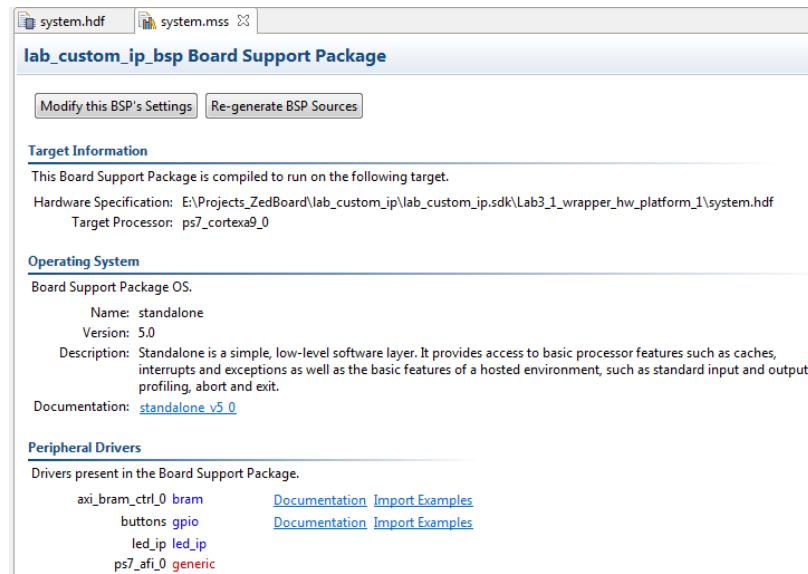
**3-2-3.** Enter **lab\_custom\_ip** as the *Project Name*. For *Hardware Platform* be sure to select the new platform. And for *Board Support Package*, choose **Create New lab\_custom\_ip\_bsp**.



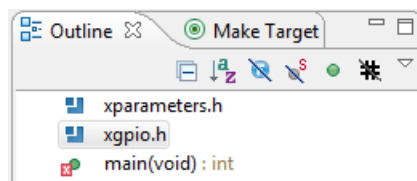
- 3-2-4. Click **Next**, and select *Empty Application* and click **Finish**.
- 3-2-5. Expand **lab\_custom\_ip** in the project view and right-click in the *src* folder and select **Import**.
- 3-2-6. Expand **General** category and double-click on **File System**.
- 3-2-7. Browse to `c:\.....\SoC_School\lab_custom_IP\sources\lab6_C_Code` folder and click OK.
- 3-2-8. Select **lab\_custom\_ip.c** and click **Finish** to add the file to the project (ignore any errors for now).



- 3-2-9. Expand **lab\_custom\_ip\_bsp** and open the **system.mss**
- 3-2-10. Click on **Documentation** link corresponding to **buttons** peripheral under the Peripheral Drivers section to open the documentation in a default browser window. Since the `led_ip` we created is very similar to GPIO, we look at the mentioned documentation.



- 3-2-11.** View the various C and Header files associated with the GPIO by clicking **Files** at the top of the page.
- 3-2-12.** Double-click on **labcustom\_ip.c** in the Project Explorer view to open the file. This will populate the **Outline** tab.
- 3-2-13.** Double click on **xgpio.h** in the *Outline* view and review the contents of the file to see the available function calls for the GPIO.



The following steps must be performed in your software application to enable reading from the GPIO: **1) Initialize the GPIO, 2) Set data direction, and 3) Read the data.** Following a resume of each of the necessary functions:

**XGpio\_Initialize (XGpio \*InstancePtr, u16 DeviceId)**

**InstancePtr** is a pointer to an XGpio instance. The memory the pointer references must be pre-allocated by the caller, that is why in this example in particular there are two instances of XGpio defined: dip (for the switches) and push (for the buttons). Further calls to manipulate the component through the XGpio API must be made with this pointer.

**DeviceId** is the unique id of the device controlled by this XGpio component. Passing in a device id associates the generic XGpio instance to a specific device, as chosen by the caller or application developer. The DeviceID number can be found in the xparameters.h file.

**XGpio\_SetDataDirection (XGpio \* InstancePtr, unsigned Channel, u32 DirectionMask)**

**InstancePtr** is a pointer to the XGpio instance to be worked on.

**Channel** contains the channel of the GPIO (1 or 2) to operate on.

**DirectionMask** is a bitmask specifying which bits are inputs and which are outputs. Bits set to 0 are output and bits set to 1 are input.

**XGpio\_DiscreteRead(XGpio \*InstancePtr, unsigned channel)**

**InstancePtr** is a pointer to the XGpio instance to be worked on.

**Channel** contains the channel of the GPIO (1 or 2) to operate on.

*Note: a quick way to access to the definition of a function is to place the mouse over the function name and then press <CTRL> and left click of the mouse.*

- 3-2-14.** Open the header file **xparameters.h** by double-clicking on **xparameters.h** in the **Outline** tab.

The xparameters.h file contains the address map for peripherals in the system. This file is generated from the hardware platform description in Vivado. Find the following **#define** used to identify the **dip switch** peripheral:

```
#define XPAR_SWITCHES_DEVICE_ID 1          //Note: the ID # may be different
```

Notice the other **#define XPAR\_BUTTONS\_\*** statements in this section for the push buttons peripheral.

- 3-2-15.** Modify line 14 of lab\_custom\_ip.c to use the definition (macro) of the switches, in the *XGpio\_Initialize* function.

```
XGpio_Initialize(&dip, XPAR_DIP_DEVICE_ID); // Modify this
XGpio_SetDataDirection(&dip, 1, 0xffffffff);
```

- 3-2-16.** Do the same for the buttons; find the macro (**#define**) for the *buttons* peripheral in xparameters.h, and modify line 17. Save the file.

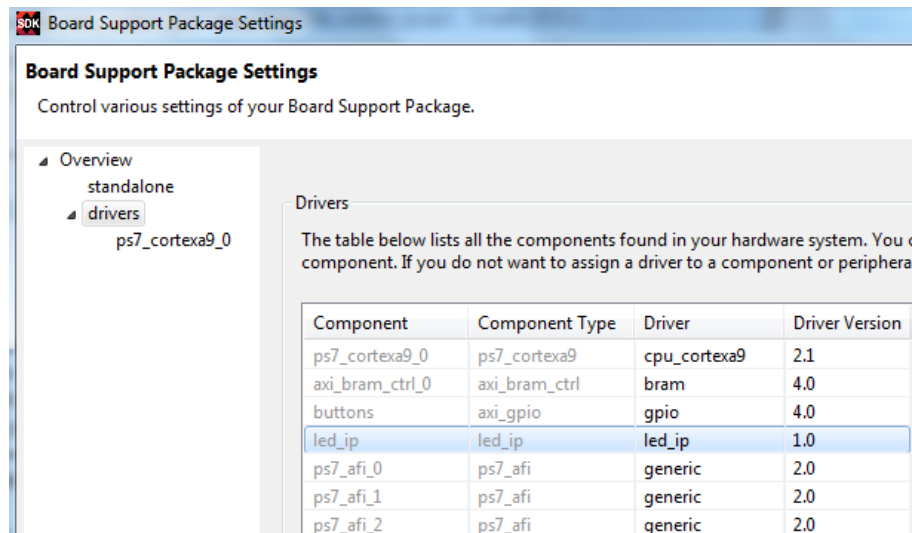
```
XGpio_Initialize(&push, XPAR_PUSH_DEVICE_ID); // Modify this
XGpio_SetDataDirection(&push, 1, 0xffffffff);
```

The project will be rebuilt. If there are any errors, check and fix your code. Your C code will eventually read the value of the switches and output it to the **led\_ip** (this last process is explained in the next steps).

### **3-3. Assign the led\_ip driver from the *driver* directory to the led\_ip instance.**

- 3-3-1.** Select **lab\_custom\_bsp** in the Project Explorer view, right-click, and select **Board Support Package Settings**.
- 3-3-2.** Select *drivers* on the left (under *Overview*)
- 3-3-3.** If the **led\_ip** driver has not already been selected, select *Generic* under the *Driver* column for *led\_ip* to access the dropdown menu. From the dropdown menu, select **led\_ip**, and click **OK**.





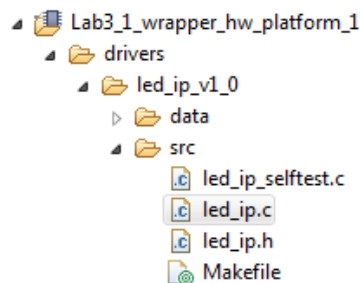
### 3-4. Examine the Driver code

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.

- 3-4-1.** To find the driver code you can do it in two ways. One way is to browse using windows explorer to:

{lab\_directory}\led\_ip\ip\_repo\led\_ip\_1.0\drivers\led\_ip\_v1\_0\src

The other way is browsing in the Project Explorer pane: **see lab3\_1 !!**



Open the *led\_ip.c* file. It only includes the header file, *led\_ip.h*, for the *led\_ip*.

- 3-4-2.** Close *led\_ip.c* and open the header file *led\_ip.h*. In the beginning there is a short description on the functionality of the functions defined in this .h file:

```
/**
 *
 * Write a value to a LED_IP register. A 32 bit write is performed.
 * If the component is implemented in a smaller width, only the least
 * significant data is written.
 *
 * @param BaseAddress is the base address of the LED_IP device.
 * @param RegOffset is the register offset from the base to write to.
 * @param Data is the data written to the register.
 *
 * @return None.
 *
 * @note
```

```
* C-style signature:
* void LED_IP_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset, u32 Data)
**/
```

There are two important macros:

```
LED_IP_mWriteReg( ... )
```

```
LED_IP_mReadReg( ... )
```

defined as follow:

```
#define LED_IP_mWriteReg(BaseAddress, RegOffset, Data) \
    Xil_Out32((BaseAddress) + (RegOffset), (u32)(Data))

#define LED_IP_mReadReg(BaseAddress, RegOffset) \
    Xil_In32((BaseAddress) + (RegOffset))
```

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 *led\_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.

### 3-5. Modify your C code to echo the dip switch settings on the LEDs by using the *led\_ip* driver API macros.

3-5-1. Include the header file:

```
#include "led_ip.h"
```

3-5-2. Include the function to write to the IP (insert before the *for* loop):

```
LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);
```

Remember that the hardware address for a peripheral (e.g. the macro `XPAR_LED_IP_S_AXI_BASEADDR` in the line above) can be found in *xparameters.h*

```
#include "xparameters.h"
#include "xgpio.h"
#include "led_ip.h"

//=====

int main (void)
{
    XGpio dip, push;
    int i, psb_check, dip_check;

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

    XGpio_Initialize(&dip, XPAR_SWITCHES_DEVICE_ID);
    XGpio_SetDataDirection(&dip, 1, 0xffffffff);

    XGpio_Initialize(&push, XPAR_BUTTONS_DEVICE_ID);
    XGpio_SetDataDirection(&push, 1, 0xffffffff);


    while (1)
    {
        psb_check = XGpio_DiscreteRead(&push, 1);
        xil_printf("Push Buttons Status %x\r\n", psb_check);
        dip_check = XGpio_DiscreteRead(&dip, 1);
        xil_printf("DIP Switch Status %x\r\n", dip_check);

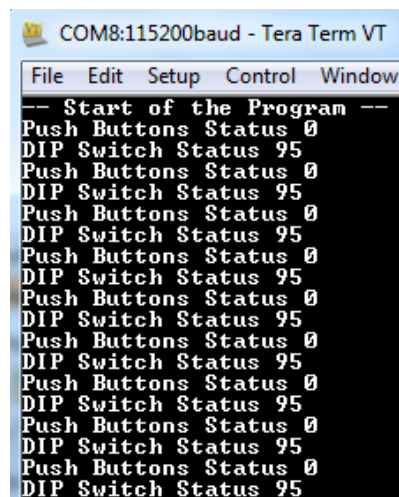
        // output dip switches value on LED_ip device
        LED_IP_mWriteReg(XPAR_LED_IP_S_AXI_BASEADDR, 0, dip_check);
        for (i=0; i<9999999; i++);
    }
}
```

**3-5-3.** Save the file and the program will be compiled again.

## Verify in Hardware

### Step 4

- 4-1. Objective: Connect the board with the two micro-usb cables and power it ON. Establish the serial communication, configure the PL and run the program in PS.**
- 4-1-1.** Follow the indications in previous labs to power the ZedBoard ON and connect the two micro USB cables.
- 4-1-2.** Configure the serial communication software (Tera Term, Putty, etc.) as it was done in previous labs.
- 4-1-3.** 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: *lab3\_1\_wrapper\_hw\_platform\_1*), and the right bitstream.
- 4-1-4.** Wait for the blue LED on the ZedBoard to turn on, as signal of successfully programed the device.
- 4-1-5.** Next, select **lab\_custom\_ip** in *Project Explorer*, 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*.
- 4-1-6.** Flip the DIP switches on the ZedBoard and verify that the LEDs light change according to the switch positions. Check that you see the right values of the DIP switches and the Push button in the serial terminal software.



Note: Setting the DIP switches and push buttons will change the results displayed.

## Target Code sections to the BRAM controller (Optional Step)

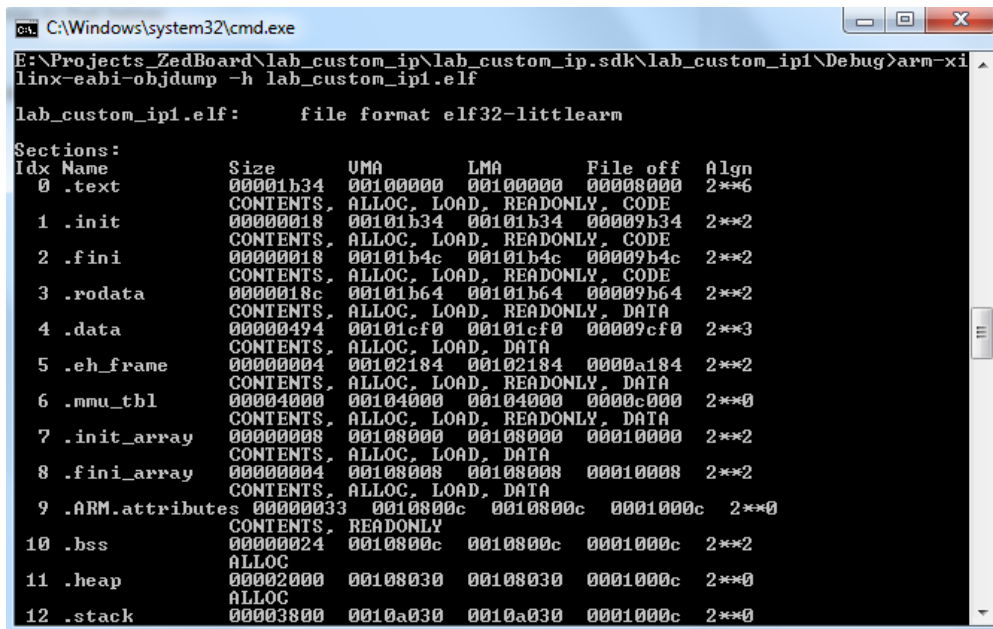
### Step 5

#### 6-1. **Objective:** Launch Shell and objdump lab4.elf and look at the sections it has created.

6-1-1. Launch the shell window from the SDK by selecting **Xilinx Tools > Launch Shell**.

6-1-2. Change the directory to `{current_dir}\Debug` using the `cd` command in the shell. You can determine your directory path and the current directory contents by using the `pwd` and `dir` commands.

Type `arm-xilinx-eabi-objdump -h lab_custom_ip.elf` at the prompt in the shell window. This command will list various sections of the program, along with the starting address and the size of each section. You should see results similar to that below:



```

C:\Windows\system32\cmd.exe
E:\Projects_ZedBoard\lab_custom_ip\lab_custom_ip.sdk\lab_custom_ip1\Debug>arm-xi
linx-eabi-objdump -h lab_custom_ip1.elf

lab_custom_ip1.elf:      file format elf32-littlearm

Sections:
Idx Name          Size      VMA           LMA       File off  Algn
 0 .text          00001b34  00100000  00100000  00008000  2**6
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .init          00000018  00101b34  00101b34  00009b34  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .fini          00000018  00101b4c  00101b4c  00009b4c  2**2
   CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .rodata        0000018c  00101b64  00101b64  00009b64  2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data          00000494  00101cf0  00101cf0  00009cf0  2**3
   CONTENTS, ALLOC, LOAD, DATA
 5 .eh_frame      00000004  00102184  00102184  0000a184  2**2
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .mmu_tbl       00004000  00104000  00104000  0000c000  2**0
   CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .init_array    00000008  00108000  00108000  00010000  2**2
   CONTENTS, ALLOC, LOAD, DATA
 8 .fini_array    00000004  00108008  00108008  00010008  2**2
   CONTENTS, ALLOC, LOAD, DATA
 9 .ARM.attributes 00000033  0010800c  0010800c  0001000c  2**0
   CONTENTS, READONLY
10 .bss           00000024  0010800c  0010800c  0001000c  2**2
   ALLOC
11 .heap          00002000  00108030  00108030  0001000c  2**0
   ALLOC
12 .stack         00003800  0010a030  0010a030  0001000c  2**0

```

At this point all the software is executed from the DDR3 memory. We are going to let the linker use the BRAM as well in the following steps.

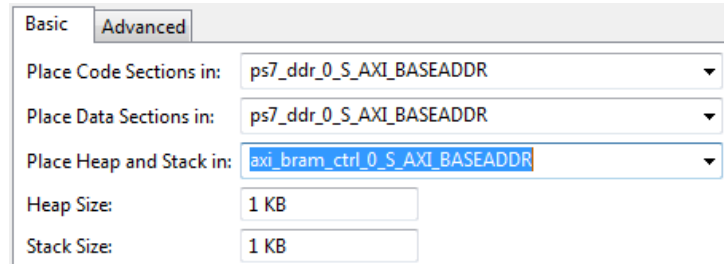
6-1-3. Right click on lab\_custom\_ip and click **Generate Linker Script**.

6-1-4. On the left pane it is detailed the Hardware Memory Map for this project

Hardware Memory Map		
Memory	Base Address	Size
axi_bram_ctrl_0_S_AXI_BASEADDR	0x40000000	8 KB
ps7_ddr_0_S_AXI_BASEADDR	0x00100000	511 MB
ps7_ram_0_S_AXI_BASEADDR	0x00000000	192 KB
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	~63.5 KB

- 6-1-5.** The right pane, it is the configuration section where it is defined how the code is split in the different memory available.

In the *Basic Tab* change the *Heap and Stack* sections to **axi\_bram\_ctrl\_0\_S\_AXI\_BASEADDR** memory, click **Generate**, and click **Yes** to overwrite.



That is to say, that the Heap and the Stack should be placed in the BRAM memory.

The program will compile again.

Once again type **arm-xilinx-eabi-objdump -h lab\_custom\_ip.elf** at the prompt in the shell window to list various sections of the program. You should see results similar to this:

```

C:\Windows\system32\cmd.exe
E:\Projects_ZedBoard\lab_custom_ip\lab_custom_ip.sdk\lab_custom_ip1\Debug>arm-xi
linx-eabi-objdump -h lab_custom_ip1.elf

lab_custom_ip1.elf:      file format elf32-littlearm

Sections:
Idx Name              Size      UMA      LMA      File off  Algn
 0 .text              00001b34  00100000  00100000  00008000  2**6
CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .init              00000018  00101b34  00101b34  00009b34  2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
 2 .fini              00000018  00101b4c  00101b4c  00009b4c  2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
 3 .rodata             0000018c  00101b64  00101b64  00009b64  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .data               00000494  00101cf0  00101cf0  00009cf0  2**3
CONTENTS, ALLOC, LOAD, DATA
 5 .eh_frame           00000004  00102184  00102184  0000a184  2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
 6 .mmu_tbl            00004000  00104000  00104000  0000c000  2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
 7 .init_array         00000008  00108000  00108000  00010000  2**2
CONTENTS, ALLOC, LOAD, DATA
 8 .fini_array         00000004  00108008  00108008  00010008  2**2
CONTENTS, ALLOC, LOAD, DATA
 9 .ARM.attributes     00000033  0010800c  0010800c  0001000c  2**0
CONTENTS, READONLY
10 .bss               00000024  0010800c  0010800c  0001000c  2**2
ALLOC
11 .heap              00000400  40000000  40000000  00018000  2**0
ALLOC
12 .stack             00001c00  40000400  40000400  00018000  2**0

```

As it can be seen, the *.heap* and *.stack* sections targeted to BRAM whereas the rest of the application is in DDR.

- 6-2. Objective:** Execute the **lab\_custom\_ip.elf** application and observe the application working even when various sections are in different memory.

- 6-2-1.** Select **lab\_custom\_ip** in *Project Explorer*, right-click and select **Run As > Launch on Hardware (GDB)** to download the application, execute **ps7\_init**, and execute **lab\_custom\_ip.elf**

Click **Yes** if prompted to stop the execution and run the new application.

Observe the serial terminal window as the program executes. Play with the dip switches and observe the LEDs. Notice that the system is very slow in displaying the message in the serial terminal and to change in the switches as the stack and heap are from a non-cached BRAM memory.

**6-2-2.** When finished, click on the **Terminate** button in the *Console* tab.

**6-2-3.** Exit SDK and Vivado.

**6-2-4.** Power OFF the board.

---

## Conclusion

---

Use SDK to define, develop, and integrate the software components of the embedded system. You can define a device driver interface for each of the peripherals and the processor. SDK imports an hdf file, creates a corresponding MSS file and lets you update the settings so you can develop the software side of the processor system. You can then develop and compile peripheral-specific functional software and generate the executable file from the compiled object code and libraries. If needed, you can also use a linker script to target various segments in various memories. When the application is too big to fit in the internal BRAM, you can download the application in external memory and then execute the program.

### Note

*This practical exercise is based in a Laboratory offered by the Xilinx University Program (XUP). It has been modified and updated by Cristian Sisterna.*