

EE3220: System-On-Chip Design

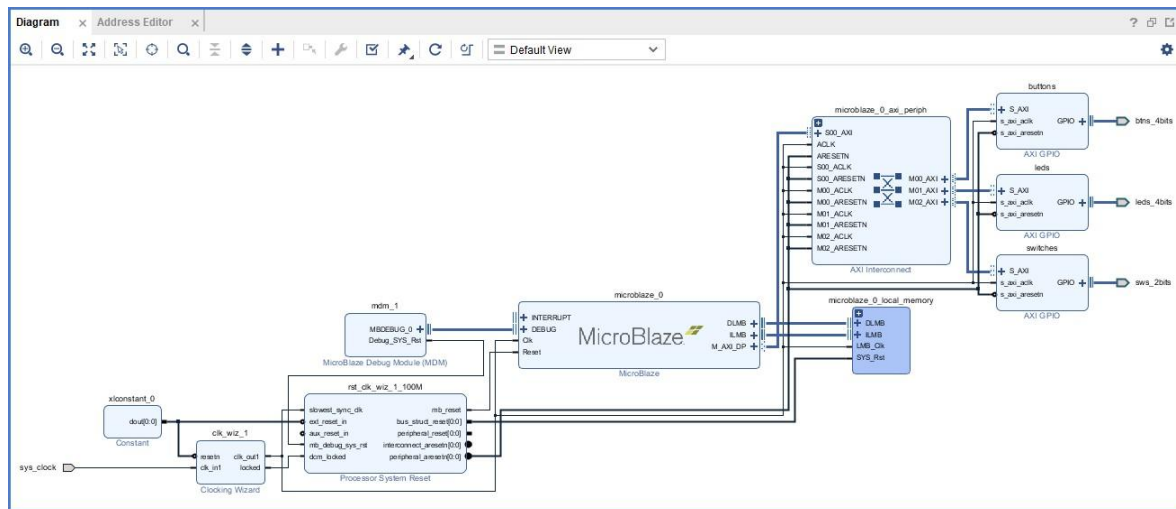
Lab 2: Designing Embedded Systems with MicroBlaze

Objectives

At the end of this lab, students should be able to:

- Use Vivado to instantiate a MicroBlaze soft processor core
- Add peripherals to the MicroBlaze i.e. GPIO LEDs, buttons, and switches
- Generate bitstream and export hardware for software development
- Write C program in Vitis IDE for the Microblaze
- Use the Vitis IDE to program the MicroBlaze core on the FPGA
- Control the LEDs from the C program written in the Vitis IDE

System overview:



In this lab, we are going to build an embedded system on FPGA based on MicroBlaze soft core processor. MicroBlaze is a soft processor designed by Xilinx for FPGAs. The processor can be preset as microcontroller, real time and high-performance application processor during its implementation in Vivado IDE. The MicroBlaze here is a microcontroller that will execute the C codes written for controlling the peripherals. The peripherals to be implemented and used in this lab are LEDs, switches and buttons. We will add connect the templates (APIs) of the peripherals to the Microblaze and then write a C program to control them. The programmable logic (hardware) will be built in Vivado hlx while the code (software) programming will be achieved in Vitis IDE. Using the Vitis IDE, the C code will built and finally be deployed (programmed) into the FGPA together with the hardware design (FPGA bitstream). Students should control the LEDs on the FPGA using the written C program executed by the MicroBlaze. There will be four check points whose evidence of achievements must be provided in the lab report to be submitted later. Take photos and snapshots of the check point achievement and add in your lab report.

Part 1: Embedded System Design and Implementation in Vivado

In this section, students will use Vivado hlx to create a hardware design. We will instantiate a MicroBlaze soft processor core and later add some peripherals (i.e. LEDs, switches and buttons) that connects to the MicroBlaze. We will use the block design approach in the Vivado and add the IPs of the Microblaze and the peripherals. The whole system will be synthesized and implemented. The bitstream will finally be created. The design will be exported to Vitis IDE for the software development and programming of the FPGA.

Follow the steps below to achieve part 1:

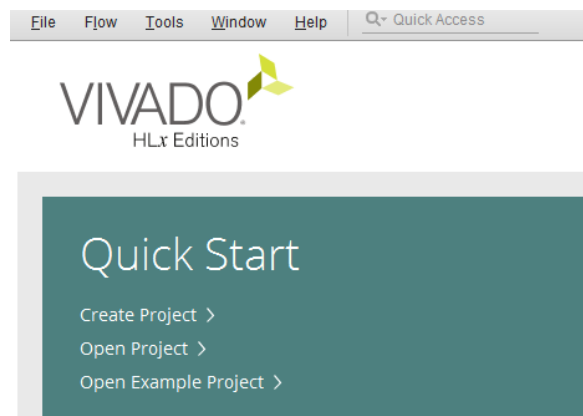
1. Add the Pynq board to Vivado for the board recognition

Since Vivado does not contain the PYNQ board by default, we need to add the board files to the Vivado directory to use the PYNQ board on Vivado. Skip this step if you use the computers in our lab, the board has already been added for you.

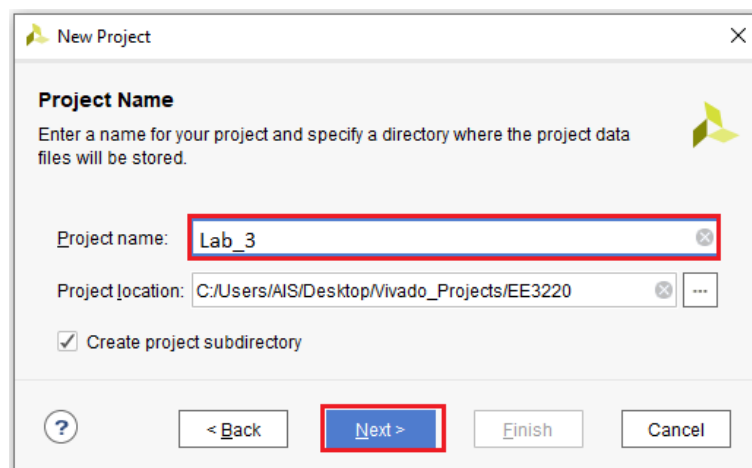
- Go to canvas to download and extract the PYNQ z2 board files.
- Copy the extracted folder to the location given below
<Xilinx installation directory>\Vivado<version>\data\boards\board_files)
[Note: You can alternatively download the files from the link (<https://d2m32eurp10079.cloudfront.net/Download/pynq-z2.zip>)]

2. Create a new Vivado project.

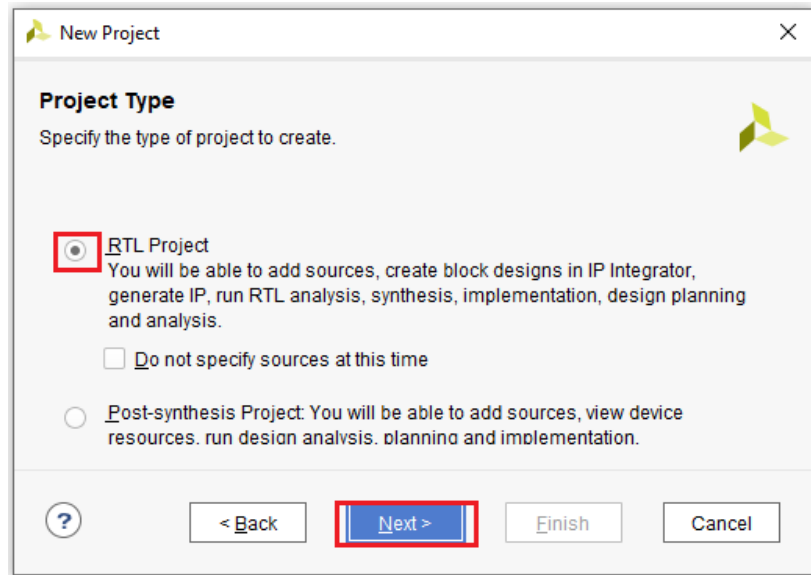
- Open Vivado, click **Create Project**, and click **Next**.



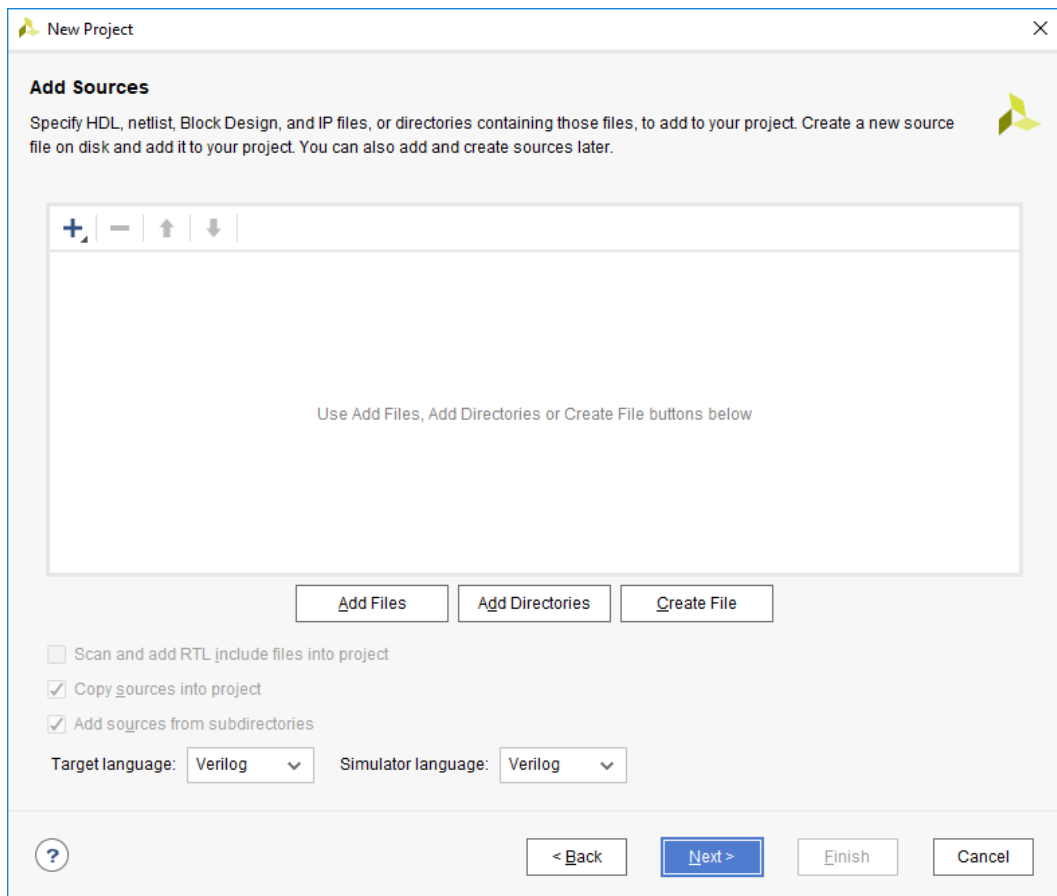
- Choose a location for your project and specify the project name, then click **Next**.



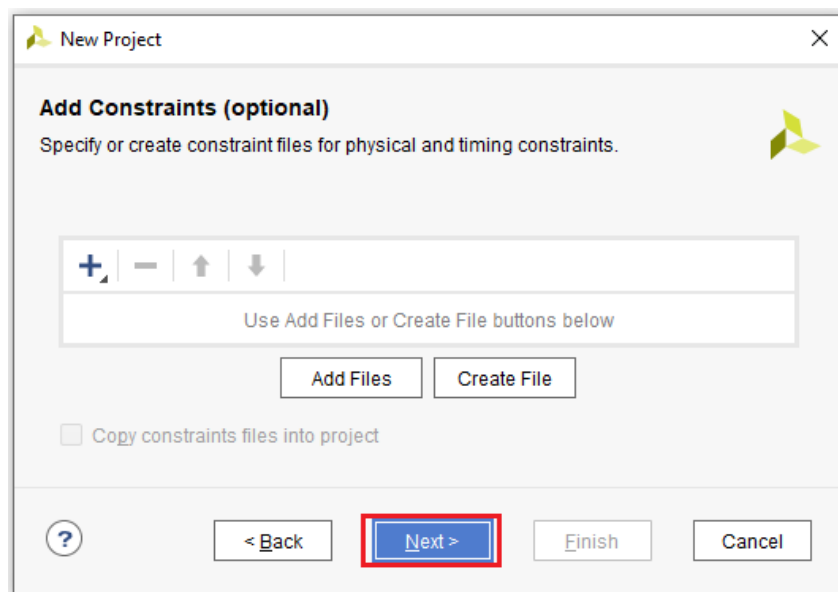
- ♦ Choose **RTL Project**, then click next



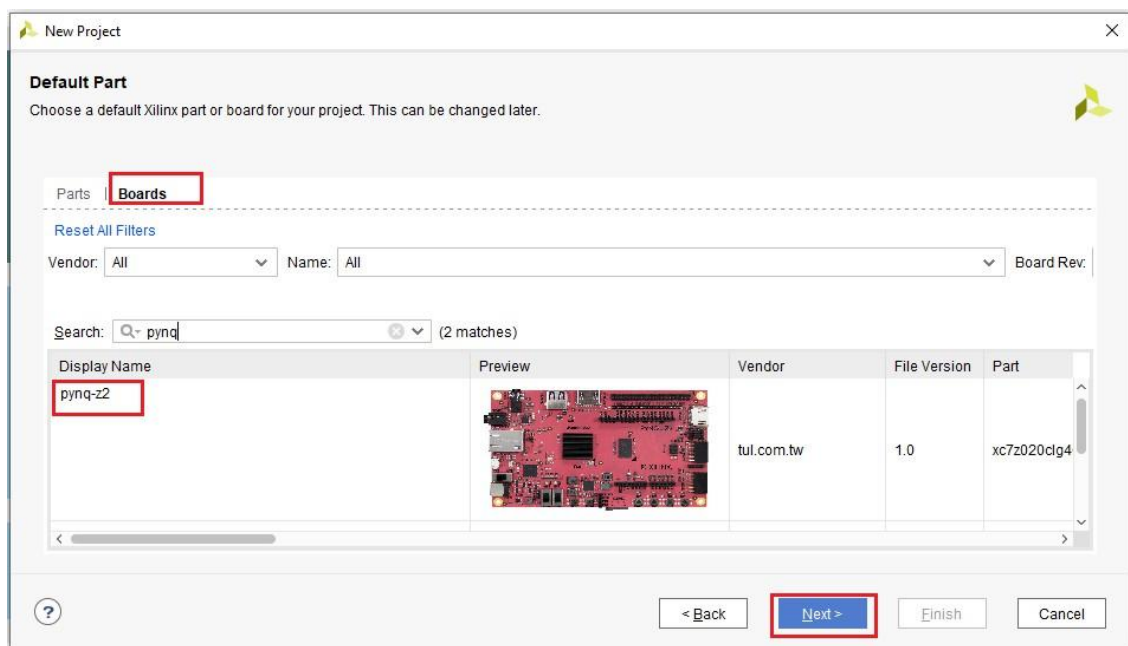
- ♦ Select **Verilog** as Target language and Simulator language. Click **Next** to create an empty project with no source code.



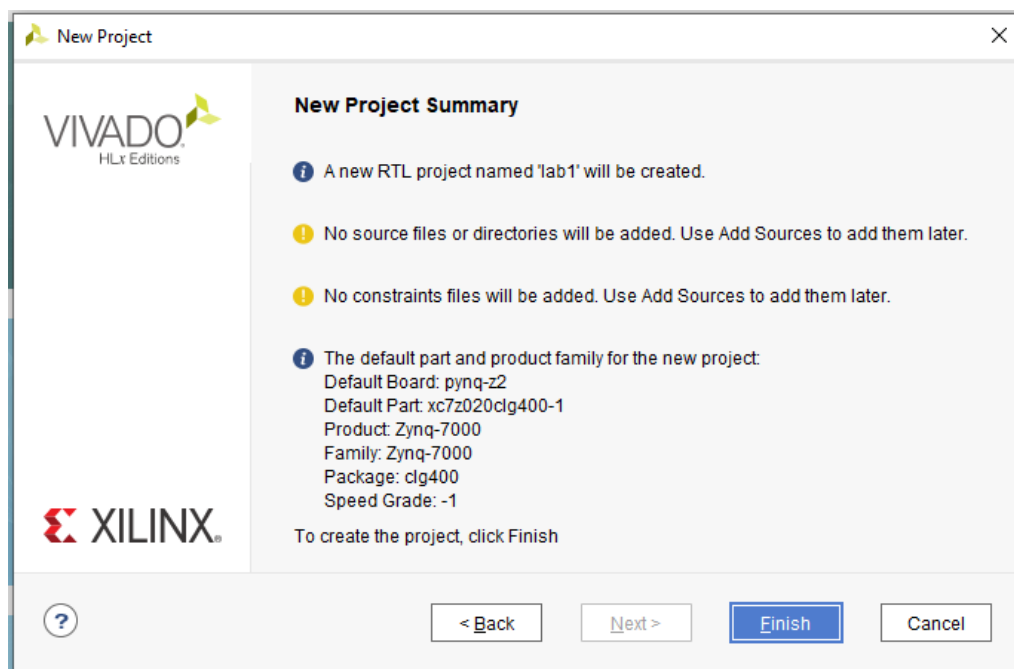
- ♦ Click **Next** to continue without adding any constraint



- Choose **Boards** tab, and search PYNQ. Then choose **PYNQ-Z2** and click next

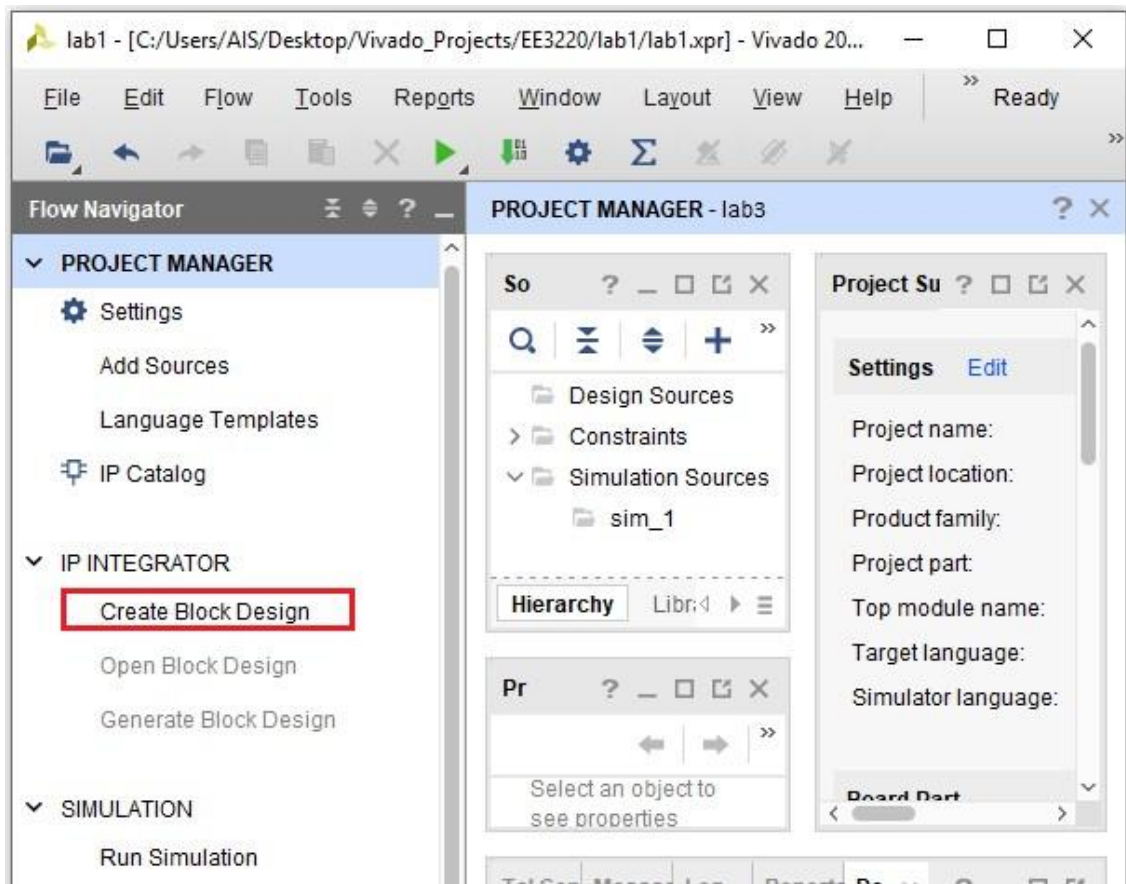


- Cross check the summary and click **Finish** to create the project

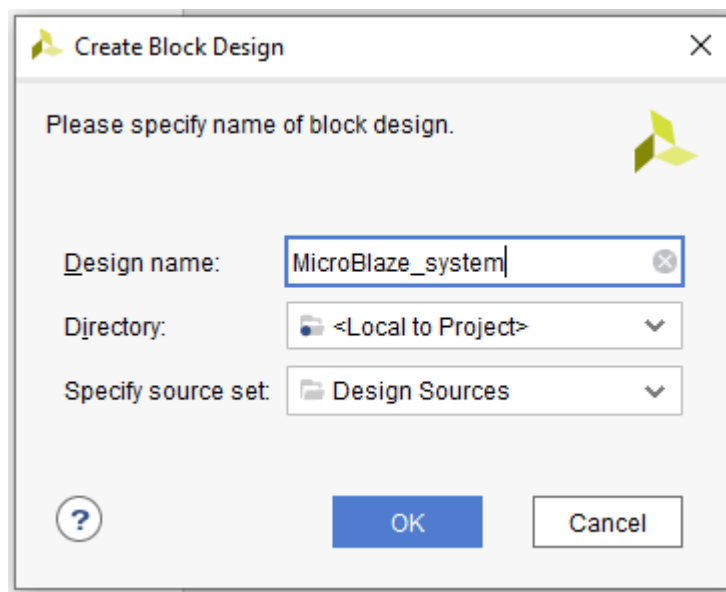


3. Instantiate the Microblaze core using the block design

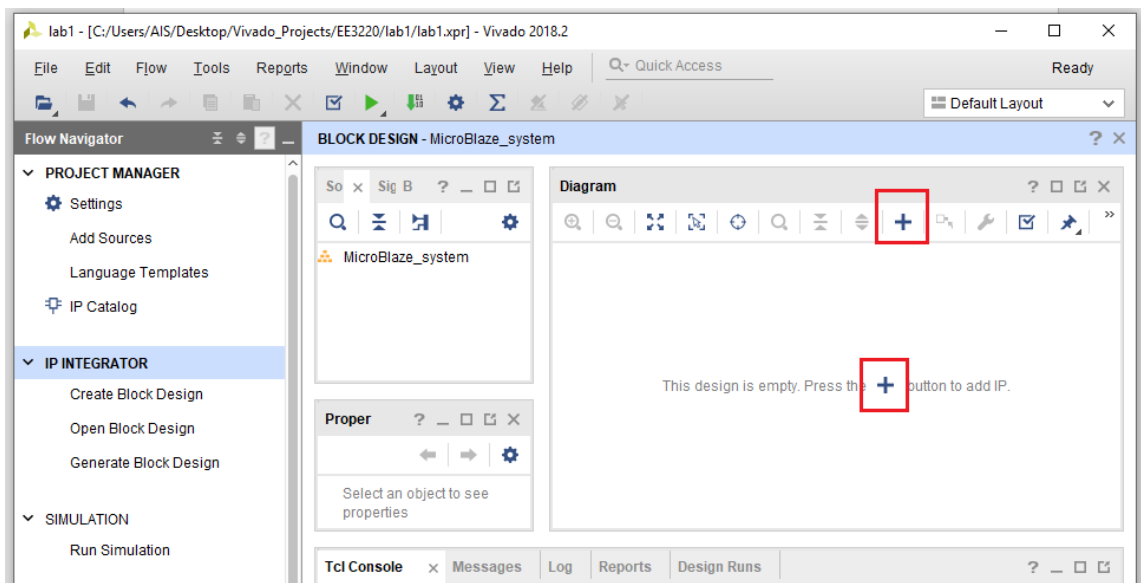
- ◆ In the **Flow Navigator** Window, click the **Create Block Design** button, then click **OK**



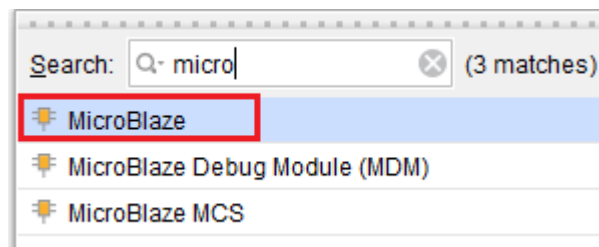
- ◆ Specify the design name and click **OK**



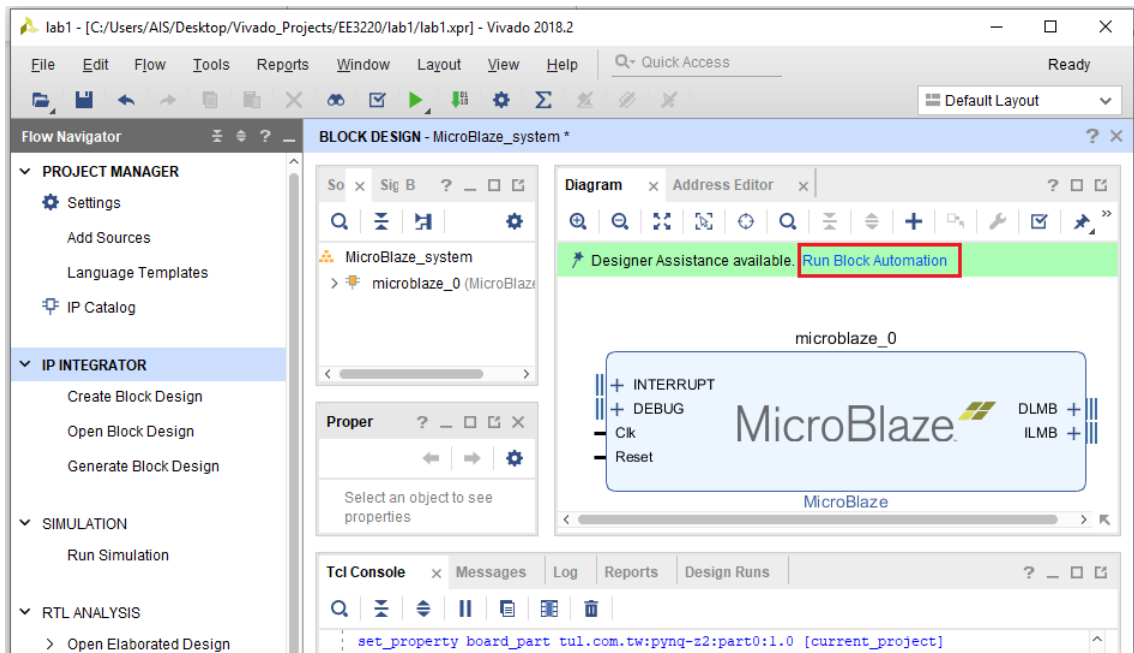
- ◆ Click the “+” icon to add IP



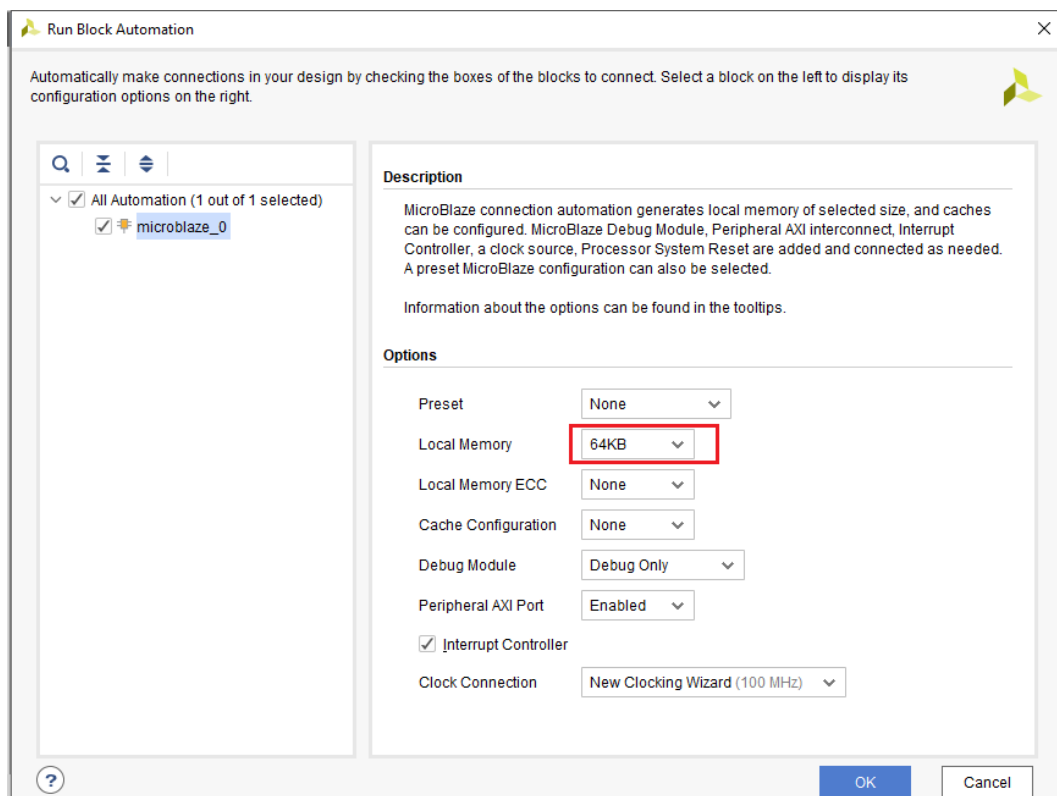
- Search and double click **MicroBlaze**, the MicroBlaze will now be added



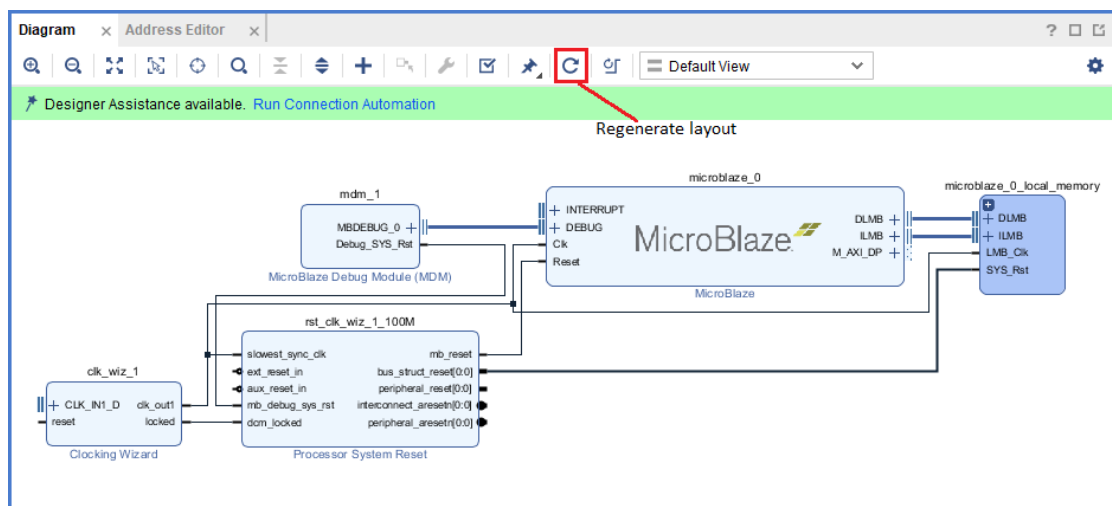
- Click **Run Block Automation**, change the memory to 64KB, and leave all the other settings as default.



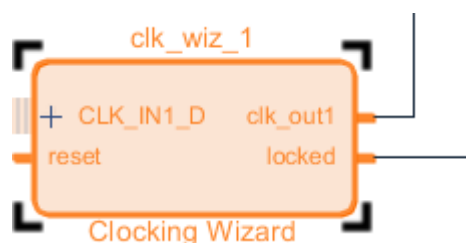
- Click **OK**. The run block automation adds and connects all the necessary modules required for running the MicroBlaze



- Click Regenerate layout (a circular arrow like a refresh sign in the block design menu)



- Double click **clocking wizard** IP and choose **sys_clock** as the source of **clk_in1**.



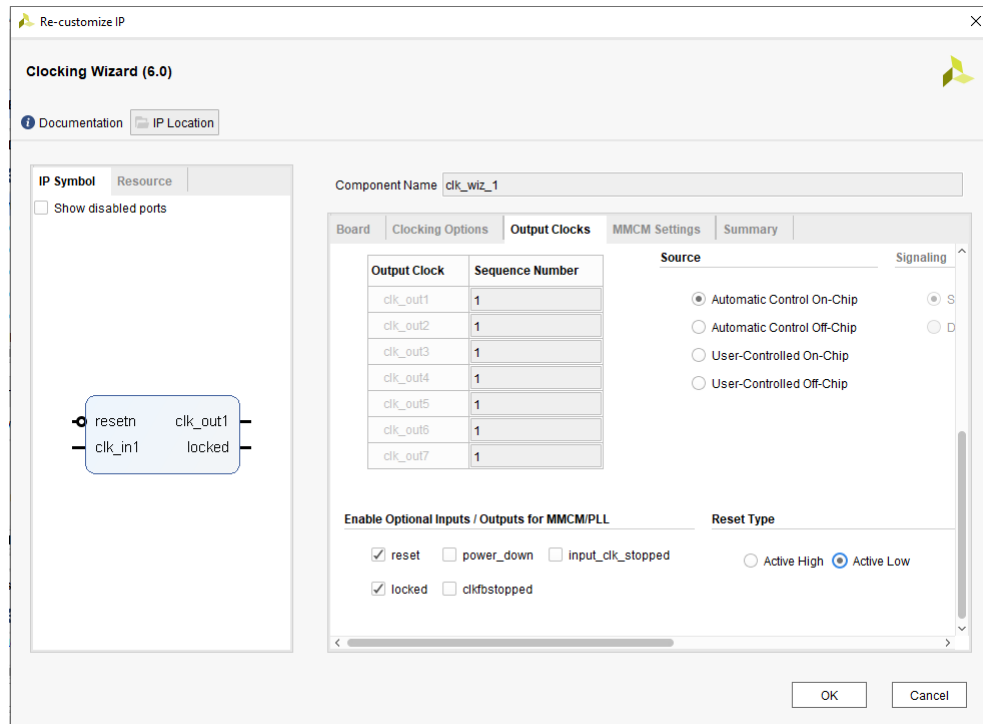
Component Name: clk_wiz_1

Board: Clocking Options Output Clocks MMCM Settings Summary

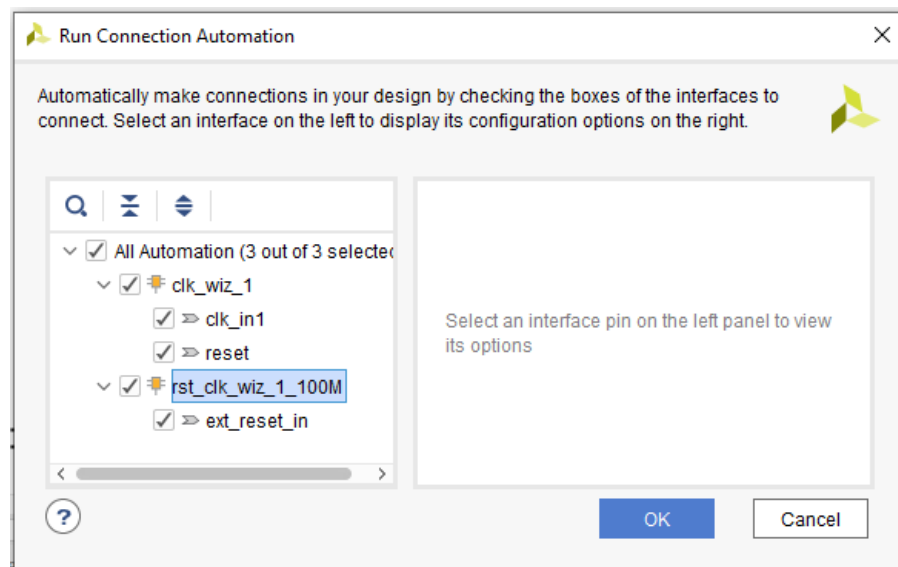
Associate IP interface with board interface

IP Interface	Board Interface
CLK_IN1	sys clock
CLK_IN2	Custom

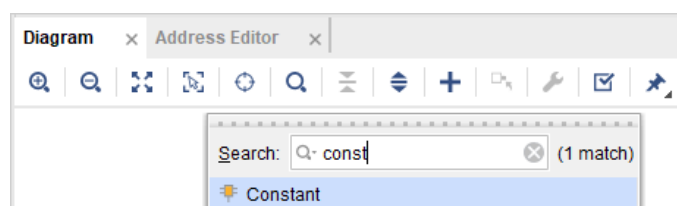
- ♦ Select the **Output Clocks** menu, scroll down to the bottom and change the reset type to **Active Low**.
- ♦ Click OK

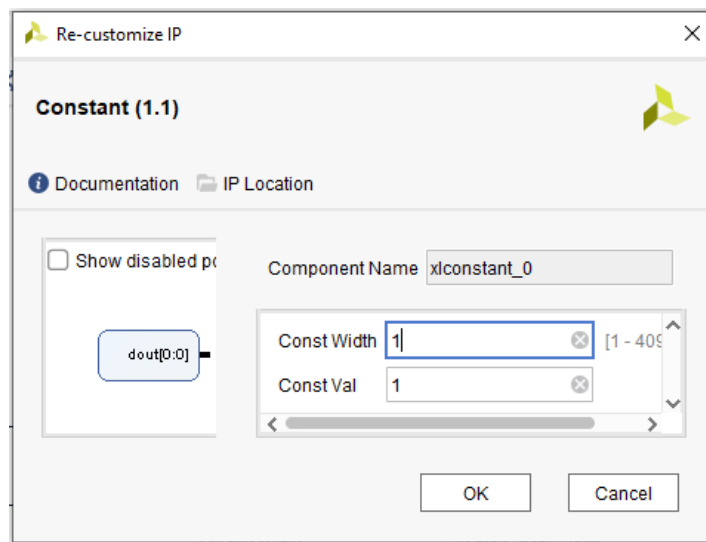


- ♦ Run connection automation

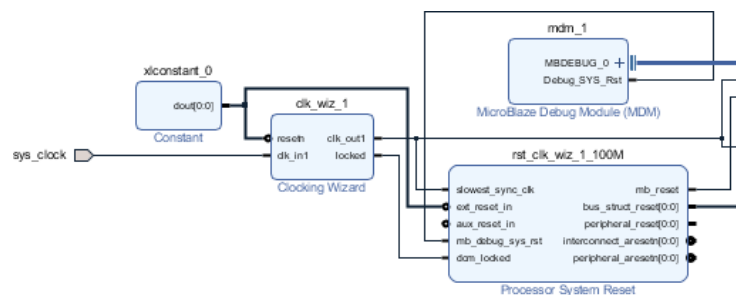


- ♦ Delete the **reset_rtl** and the wire connecting the **resetn** of the clock and the **ext_reset_in** of the Processor System Reset component.
- ♦ Click the **+** tab to add a constant IP, double click the IP and set its value to **1**.





- Connect the **resetn** of the clocking Wizard and **ext_reset_in** of the Processor System Reset to the constant

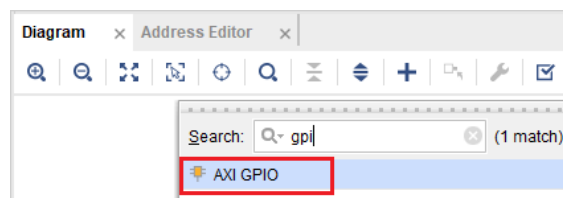


Check Point 1:

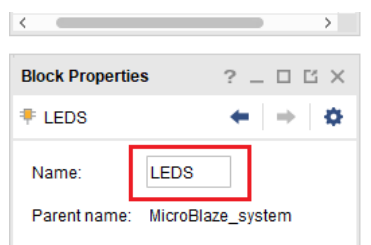
Show your work to your lab supervisor. The supervisor will inspect your work and tick the checkpoint mark for you on the checkpoint sheets. Take some snapshots as evidence of achieving the checkpoint. Proceed to the next tasks

4. Instantiate the peripherals (LEDs, switches, buttons)

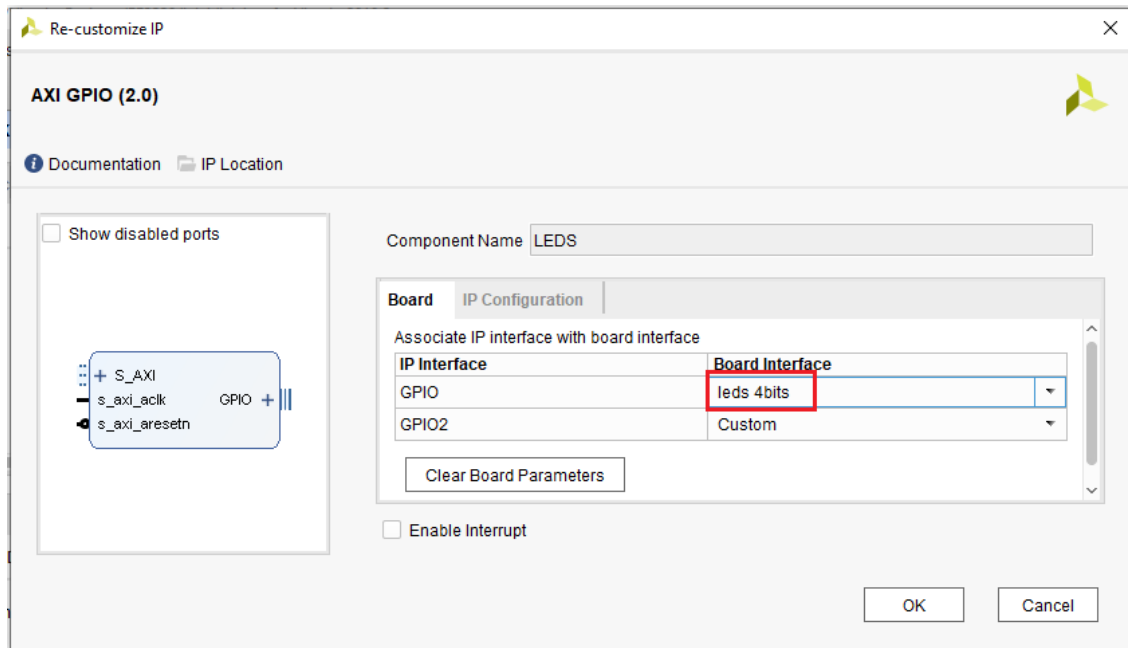
- Click add IP tab (+) from the block design menu, type **gpio** and select AXI GPIO by double-clicking.



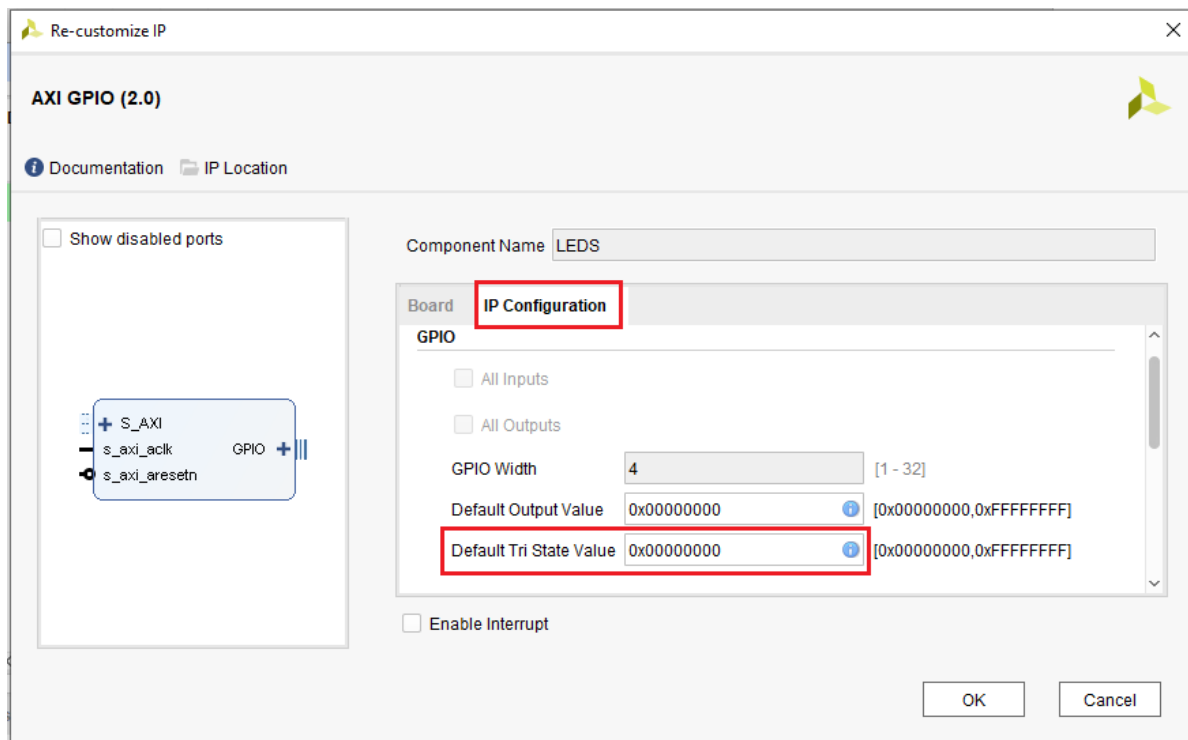
- Select the **AXI GPIO** IP, rename the GPIO to LEDs from the **Block Properties** on the left-hand side.



- ◆ Double click the AXI GPIO, under **Board Interface** of GPIO select **leds 4bits** and click **OK**

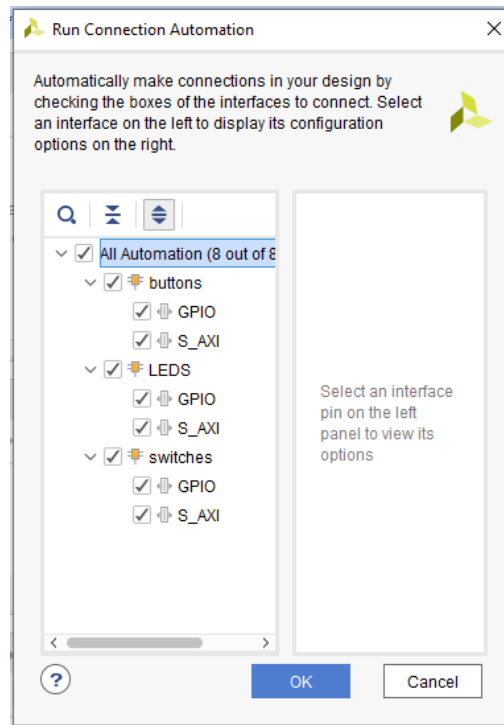


- ◆ Click IP configurations and change the **Default Tri state Value** to **0x00000000** (indicating OUTPUT), click OK

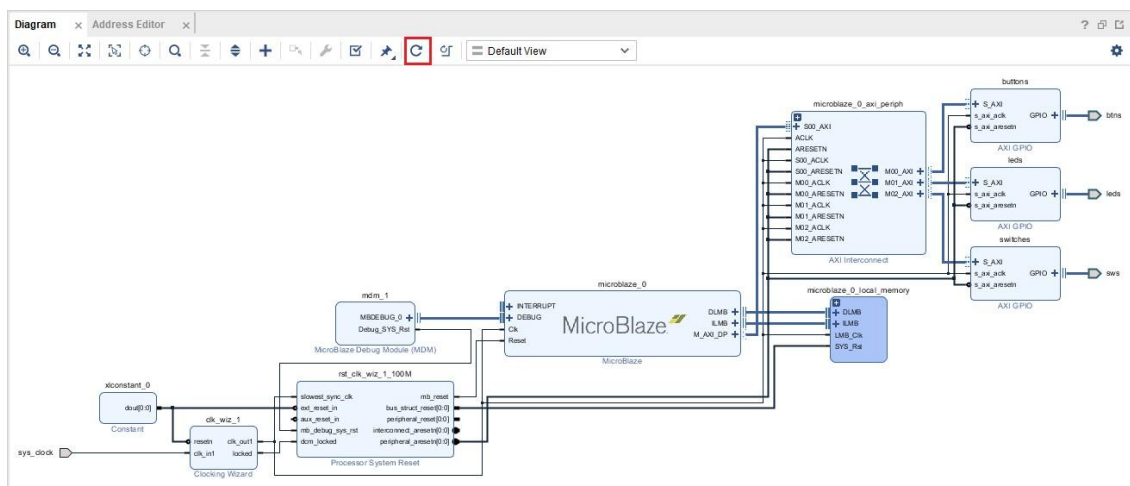


Repeat the above procedure to add the AXI GPIO for the switch and the button IPs. Rename one of the AXI GPIO to **switch** and the other to **button**. Under the **Board Interfaces** of the GPIOs, now select **sws 2bits** and **btns 4bits** for the switch and the button respectively. Leave the Default Tri State Value under IP configurations as **0xFFFFFFFF** (indicating INPUT) for the buttons and switches.

- Click **Run connection automation**, select all the peripherals and click OK.

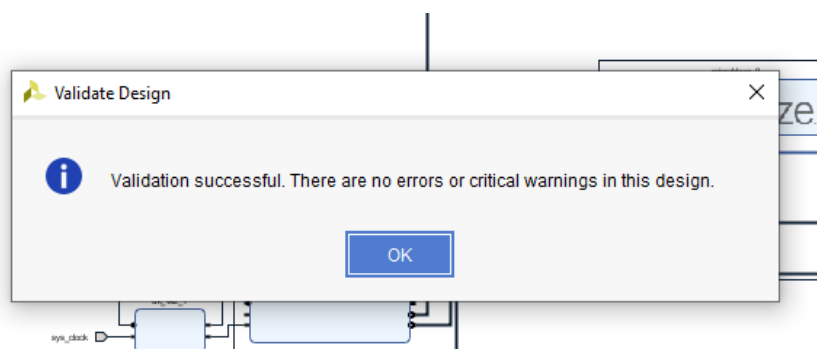


- Click regenerate layout

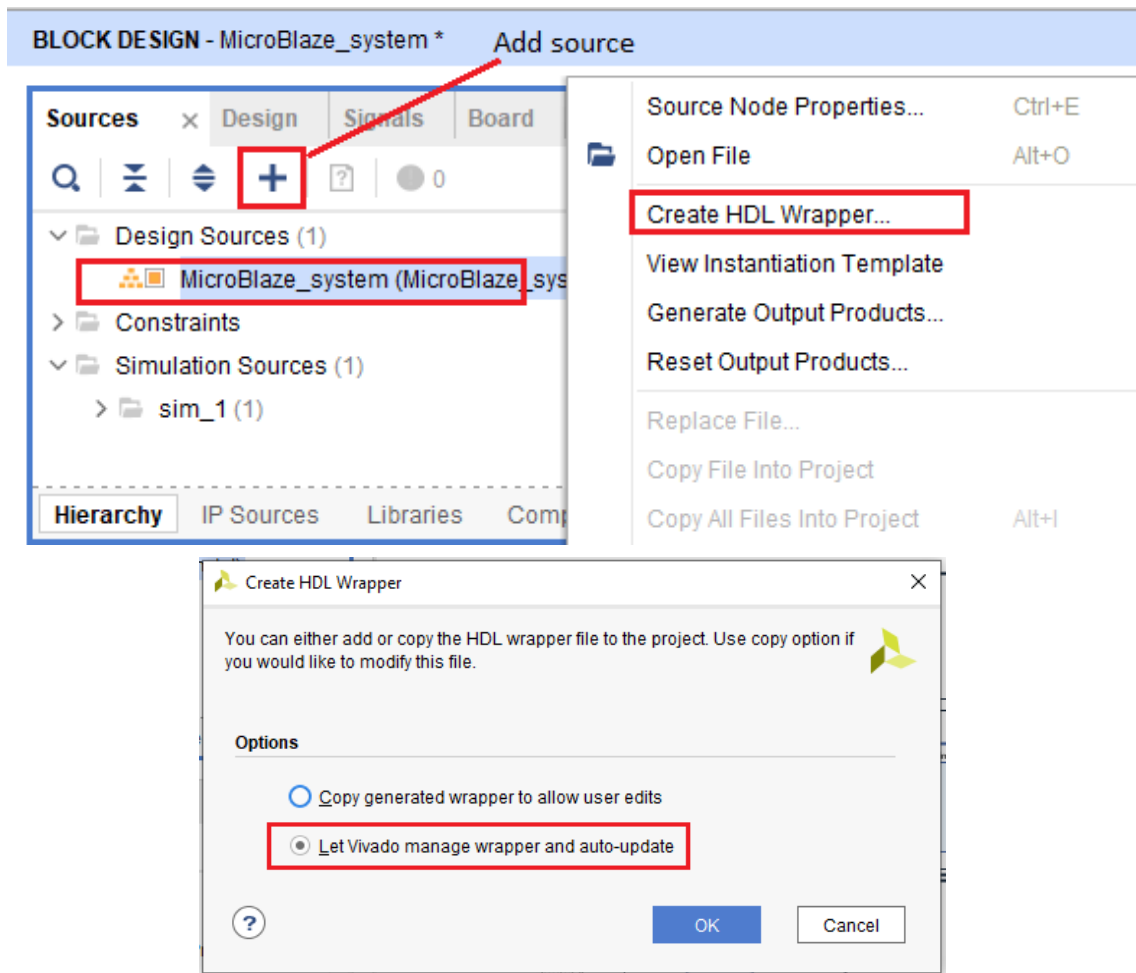


5. Validate the design and create a bitstream

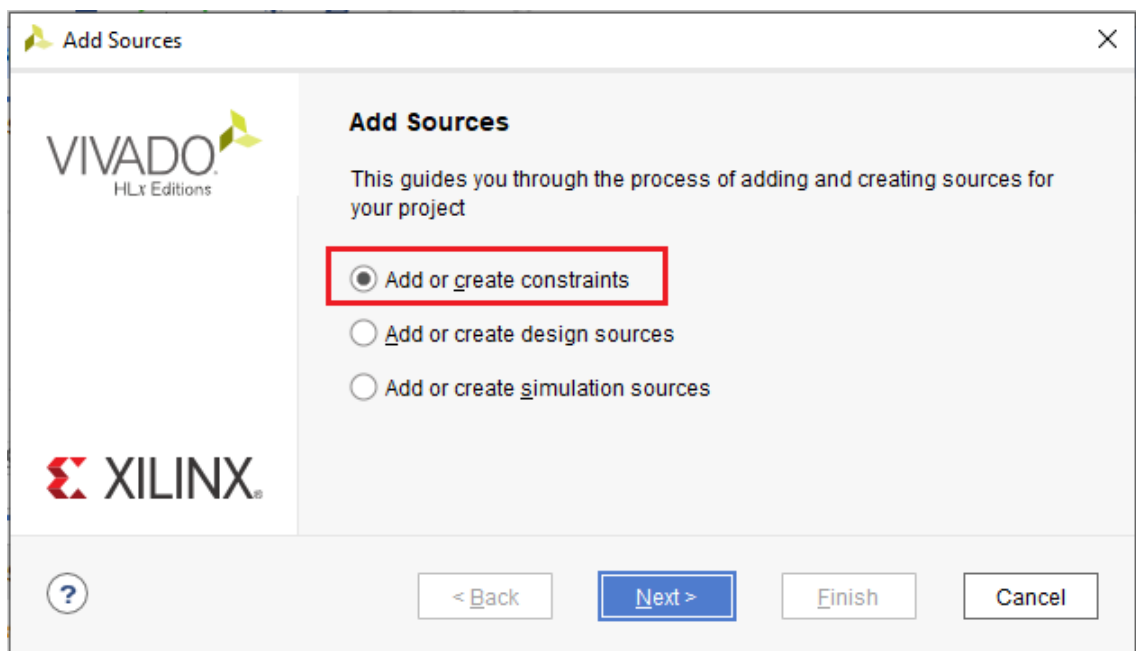
- Click the **Validate design** tab (a tick icon in the block design menu). After the validation, a dialog box shows the validation is successful. Click **OK**



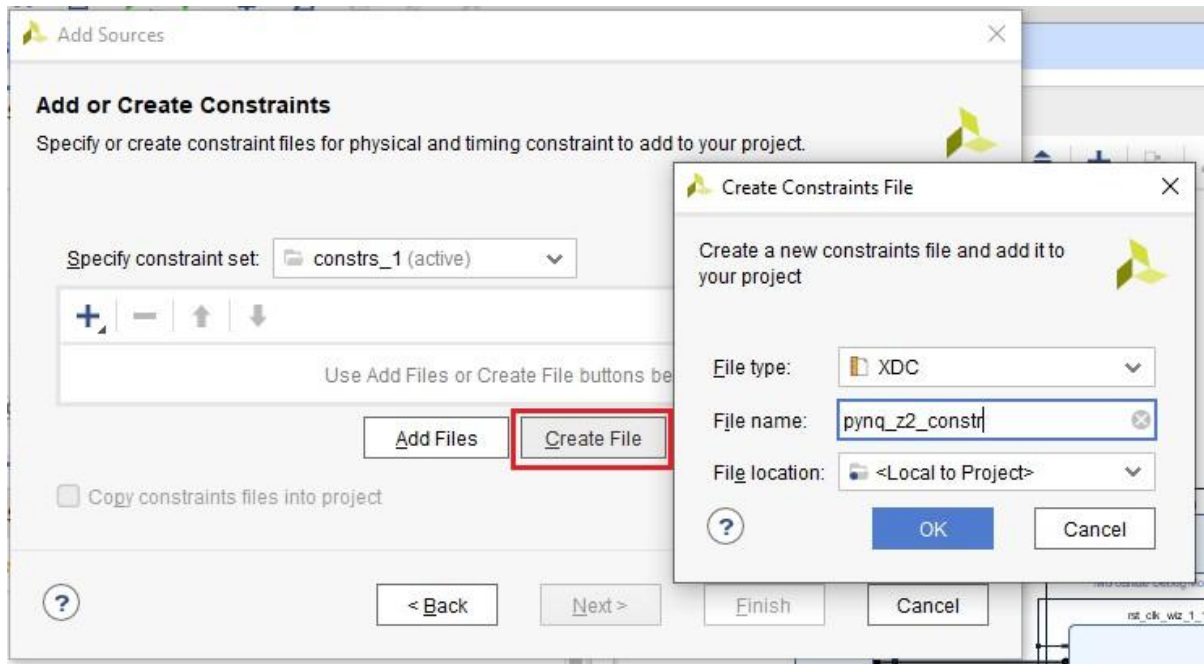
- ◆ On the sources panel, right-click the block design and click **Create HDL Wrapper**. Click OK.
- ◆ Select **Let Vivado manage wrapper and auto-update**, click OK
- ◆ Choose add sources (+) sign under **Sources** to create a constraint file,



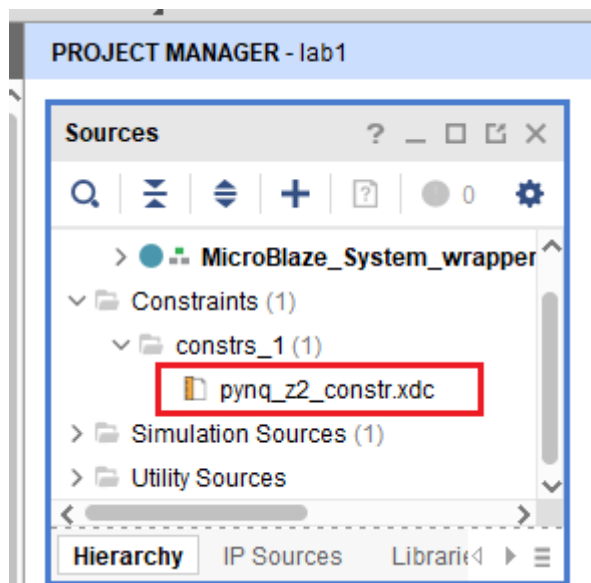
- ◆ Select **Add or create constraints**



- ◆ Click **Create File** tab, specify constraint file name and click OK



- ◆ Double-click and open the constraint file you just created , i.e. pynq_z2_constr.xdc



- ◆ Enter the constraints we provide you below by copying or typing

```

# Clock signal 125 MHz
set_property -dict {PACKAGE_PIN H16 IOSTANDARD LVCMOS33} [get_ports sys_clock];
##Switches
set_property -dict {PACKAGE_PIN M20 IOSTANDARD LVCMOS33} [get_ports {sws_tri_i[0]}};
set_property -dict {PACKAGE_PIN M19 IOSTANDARD LVCMOS33} [get_ports {sws_tri_i[1]}};
##LEDs
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[0]}};
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[1]}};
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[2]}};
set_property -dict {PACKAGE_PIN M14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[3]}};
##Buttons
set_property -dict {PACKAGE_PIN D19 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[0]}};
set_property -dict {PACKAGE_PIN D20 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[1]}};
set_property -dict {PACKAGE_PIN L20 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[2]}};
set_property -dict {PACKAGE_PIN L19 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[3]}};

```

Diagram x Address Editor x pynq_z2_contr.xdc *

C:/Users/CALAS_1/Desktop/lab3/lab_3/lab_3.srcs/constrs_1/new/pynq_z2_contr.xdc

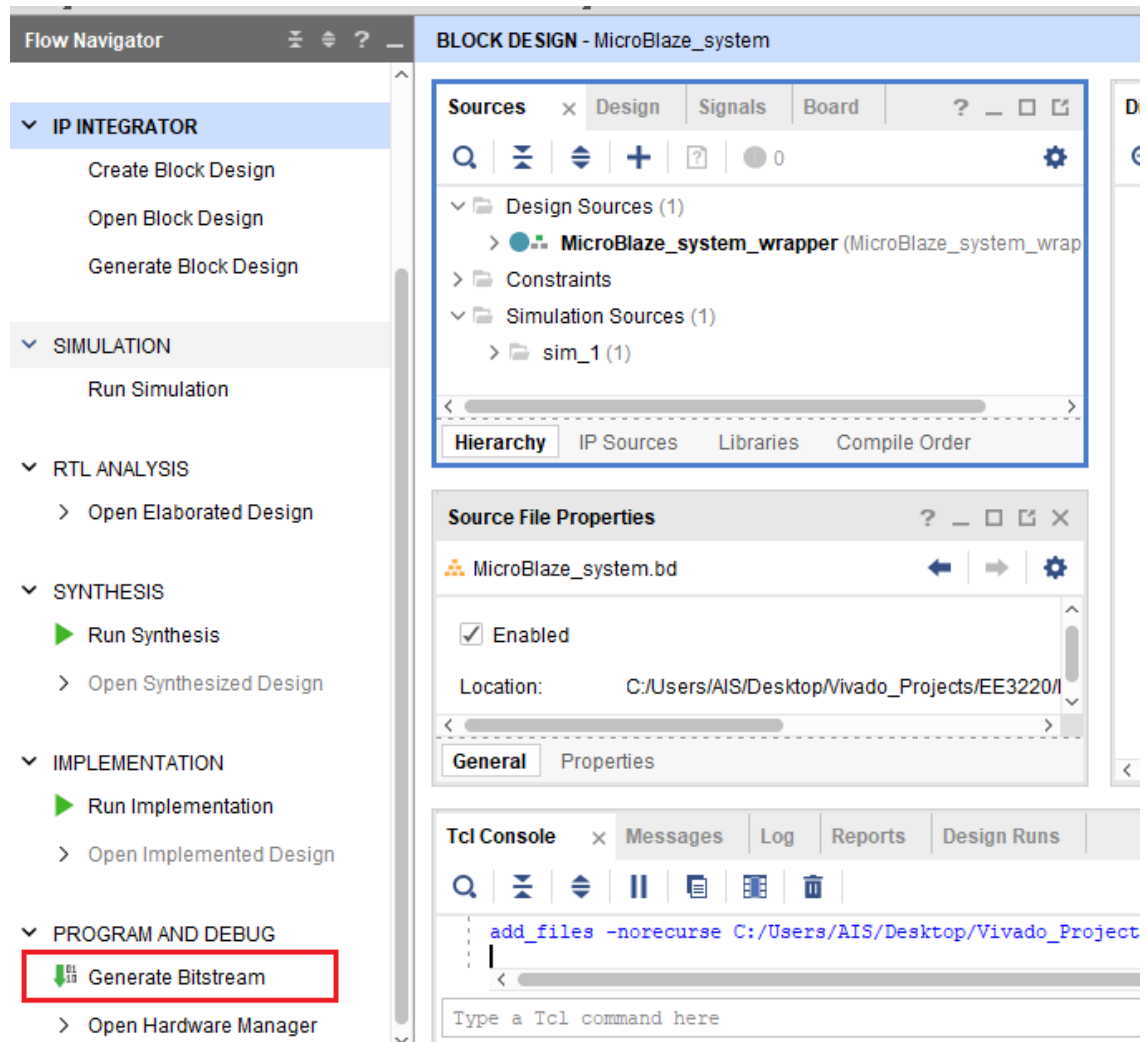
Q [Icons]

```

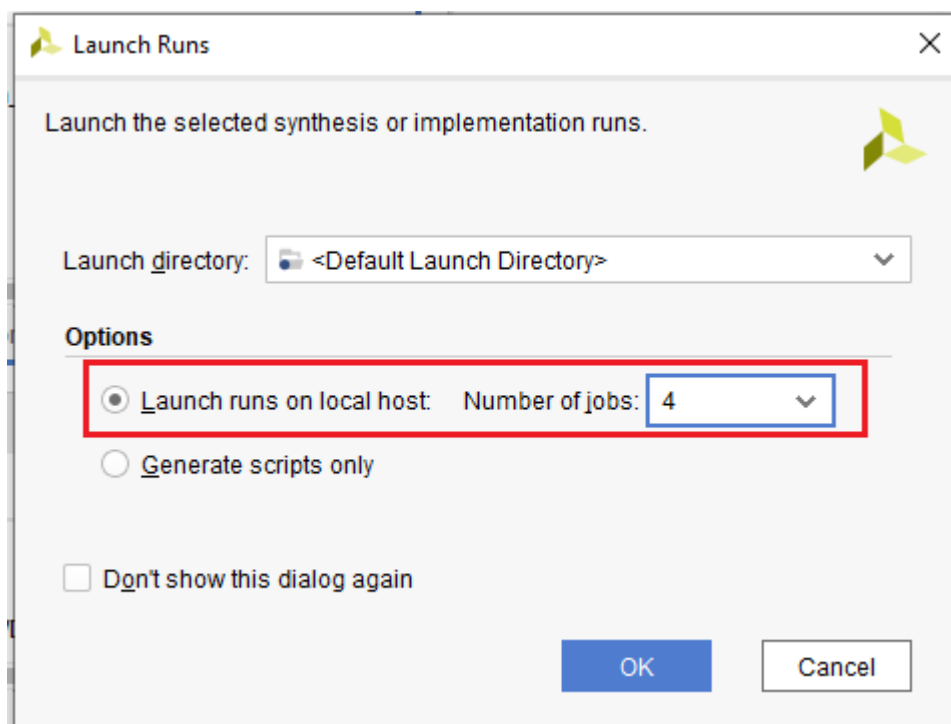
1 | # Clock signal 125 MHz
2 | set_property -dict {PACKAGE_PIN H16 IOSTANDARD LVCMOS33} [get_ports sys_clock];
3 | ##Switches
4 | set_property -dict {PACKAGE_PIN M20 IOSTANDARD LVCMOS33} [get_ports {sws_tri_i[0]}};
5 | set_property -dict {PACKAGE_PIN M19 IOSTANDARD LVCMOS33} [get_ports {sws_tri_i[1]}};
6 | ##LEDs
7 | set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[0]}};
8 | set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[1]}};
9 | set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[2]}};
10 | set_property -dict {PACKAGE_PIN M14 IOSTANDARD LVCMOS33} [get_ports {leds_tri_io[3]}};
11 | ##Buttons
12 | set_property -dict {PACKAGE_PIN D19 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[0]}};
13 | set_property -dict {PACKAGE_PIN D20 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[1]}};
14 | set_property -dict {PACKAGE_PIN L20 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[2]}};
15 | set_property -dict {PACKAGE_PIN L19 IOSTANDARD LVCMOS33} [get_ports {btns_tri_i[3]}};

```

- In the Flow Navigator, click **Generate Bitstream**

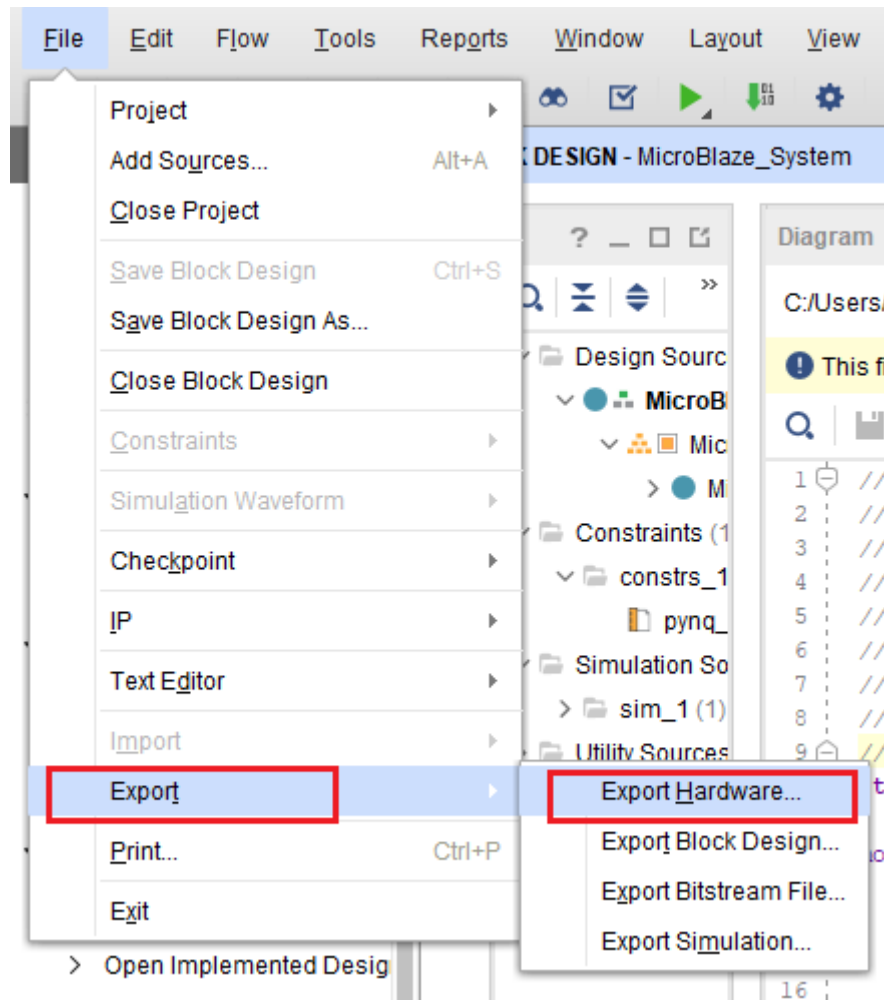


- Select **Launch runs on local host**, choose the **Number of jobs** and click OK

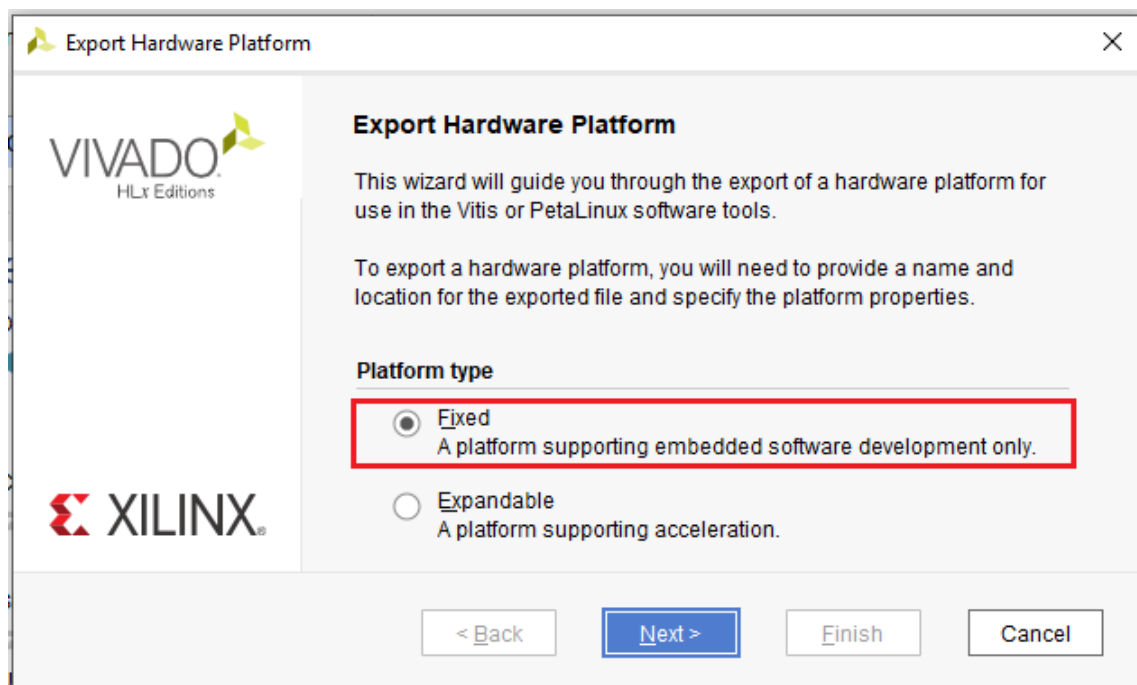


6. Export hardware after the bitstream file is successfully generated

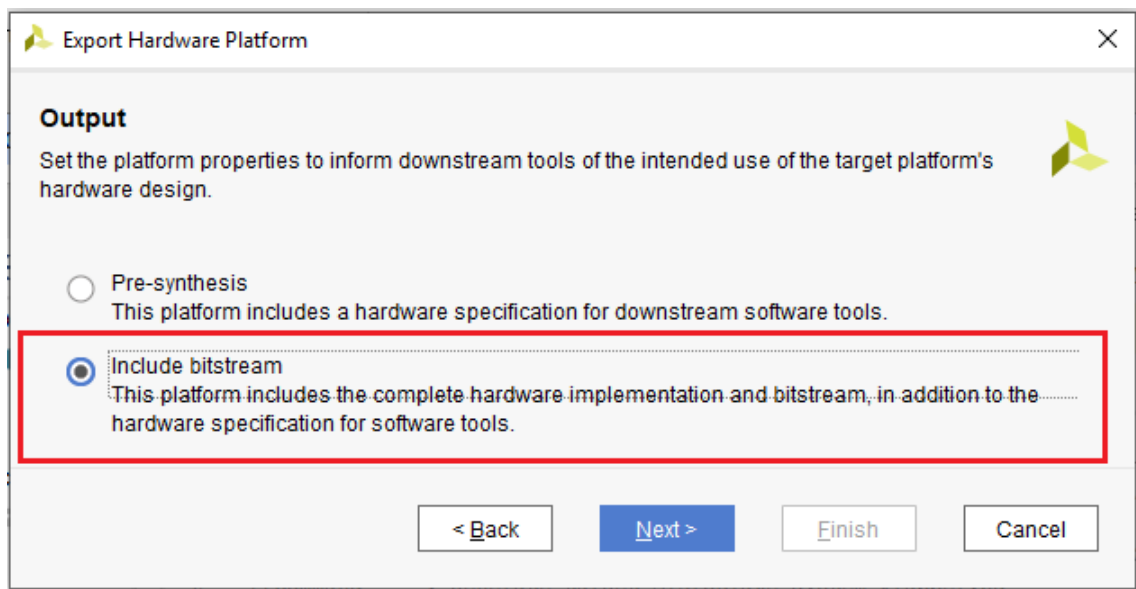
- ♦ Go to File, under Export, click **Export Hardware**



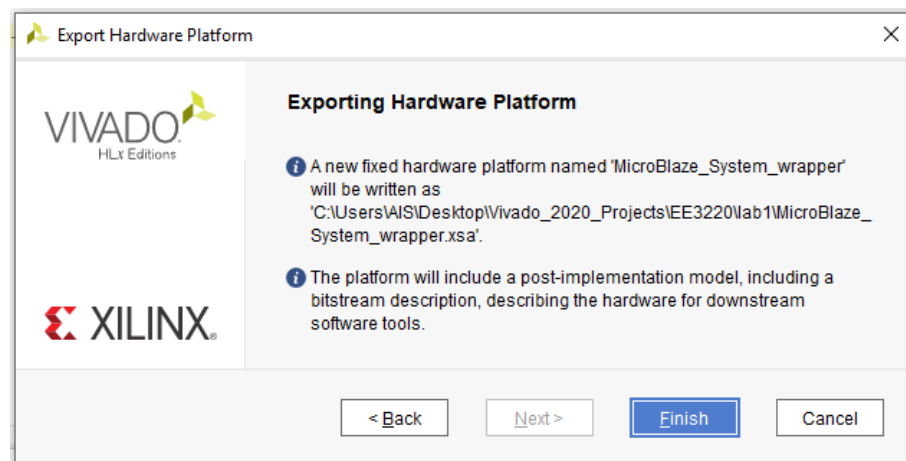
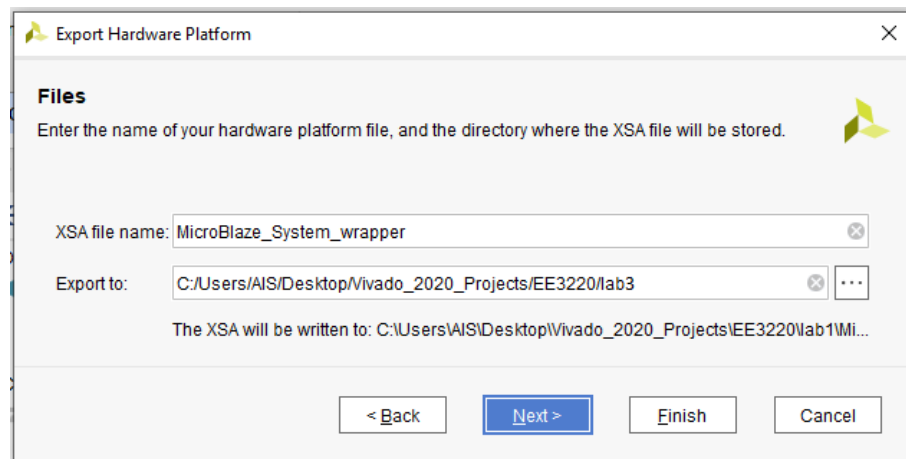
- ♦ Select **Fixed** option and click **Next**



- ♦ Choose **Include bitstream** and click next



- Choose the lab3 project directory where the XSA file will be stored.
- Click **Next** and then **Finish**



Check Point 2:

Show your work to your lab supervisor. The supervisor will inspect your work and tick the checkpoint mark for you on the checkpoint sheets. Take some snapshots as evidence of achieving the checkpoint. Proceed to the next tasks

Part 2: Software Development using Vitis

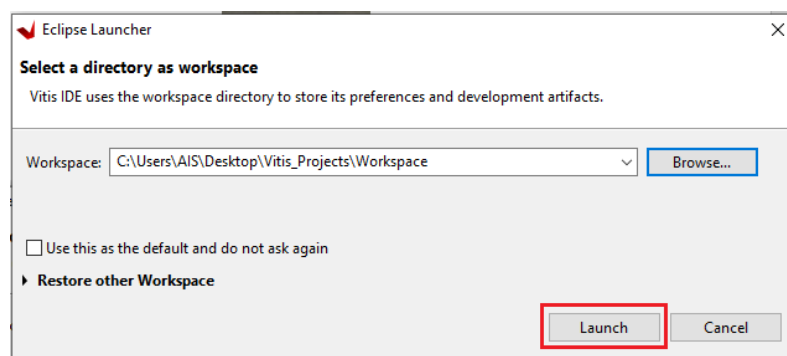
Now we will use a new tool called **Vitis**, introduced after Vivado 2019.2 version to develop our software and program the MicroBlaze on the FPGA.



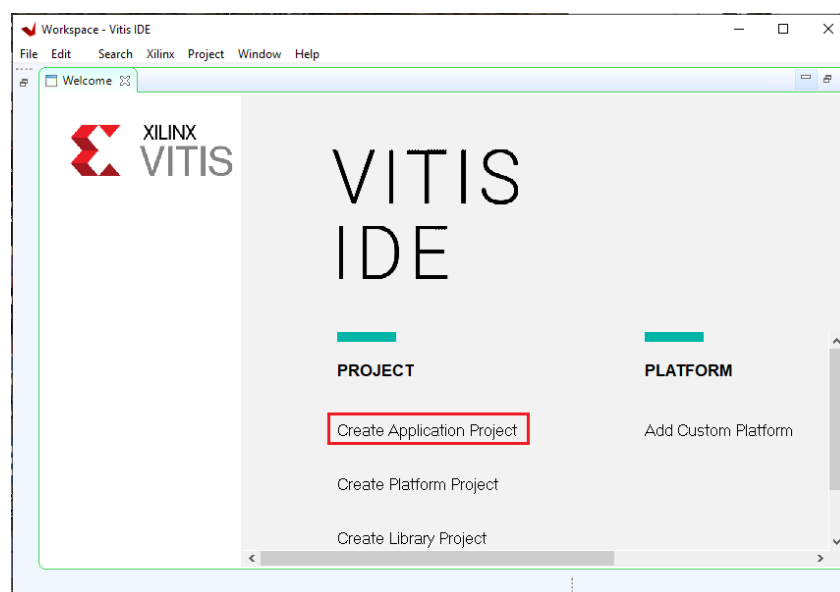
Follow the following steps to achieve this task:

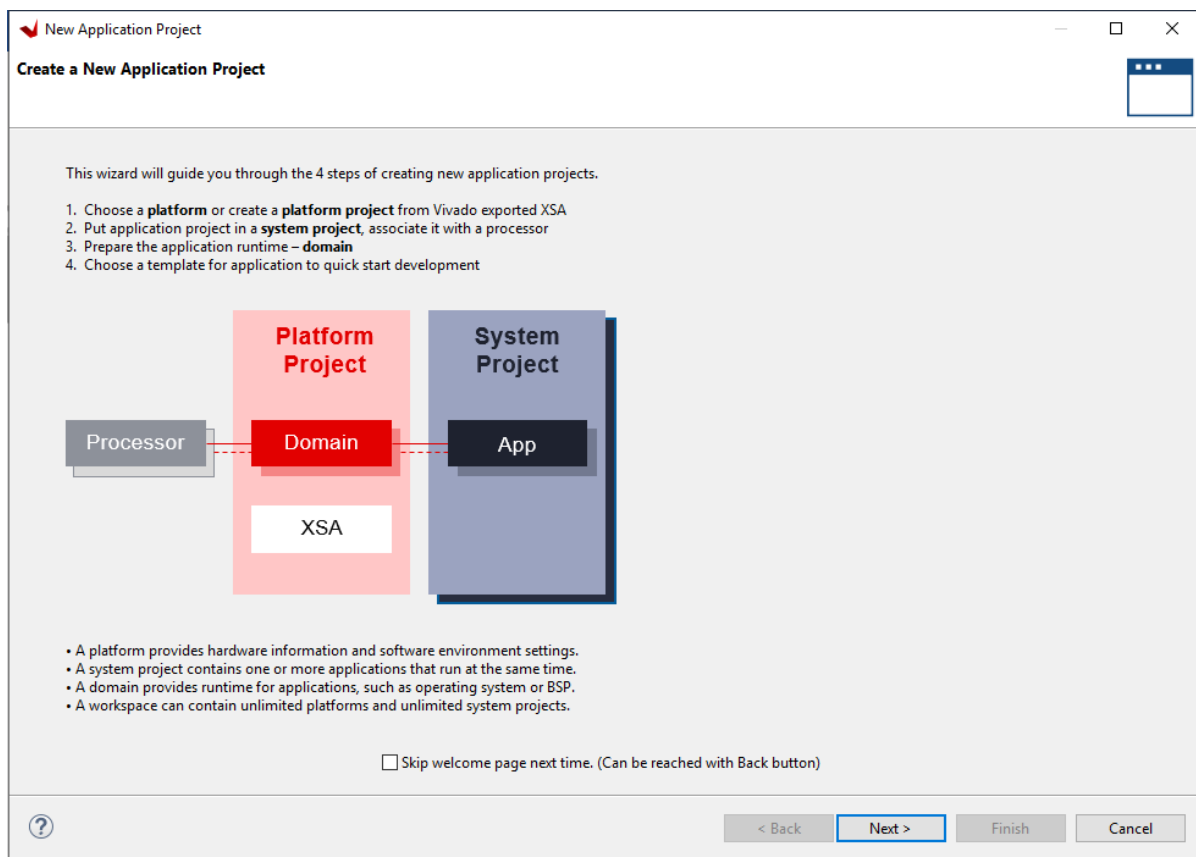
1. Create a new Vitis application project

- Open Vitis, choose a folder as your workspace location, and then click launch.

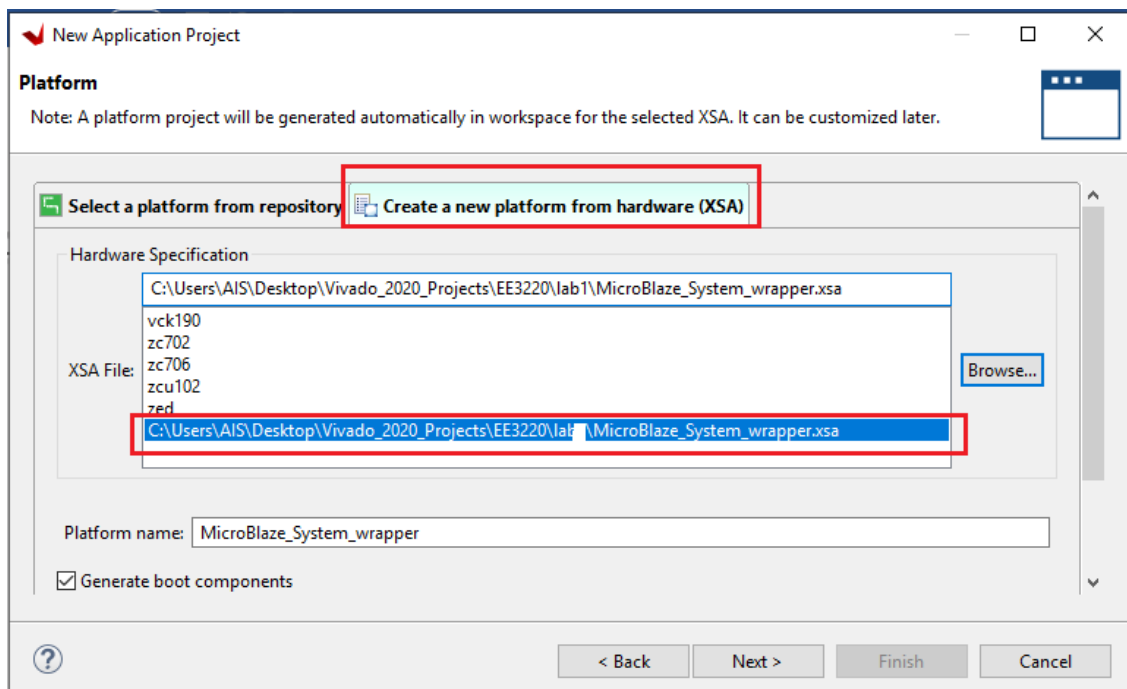


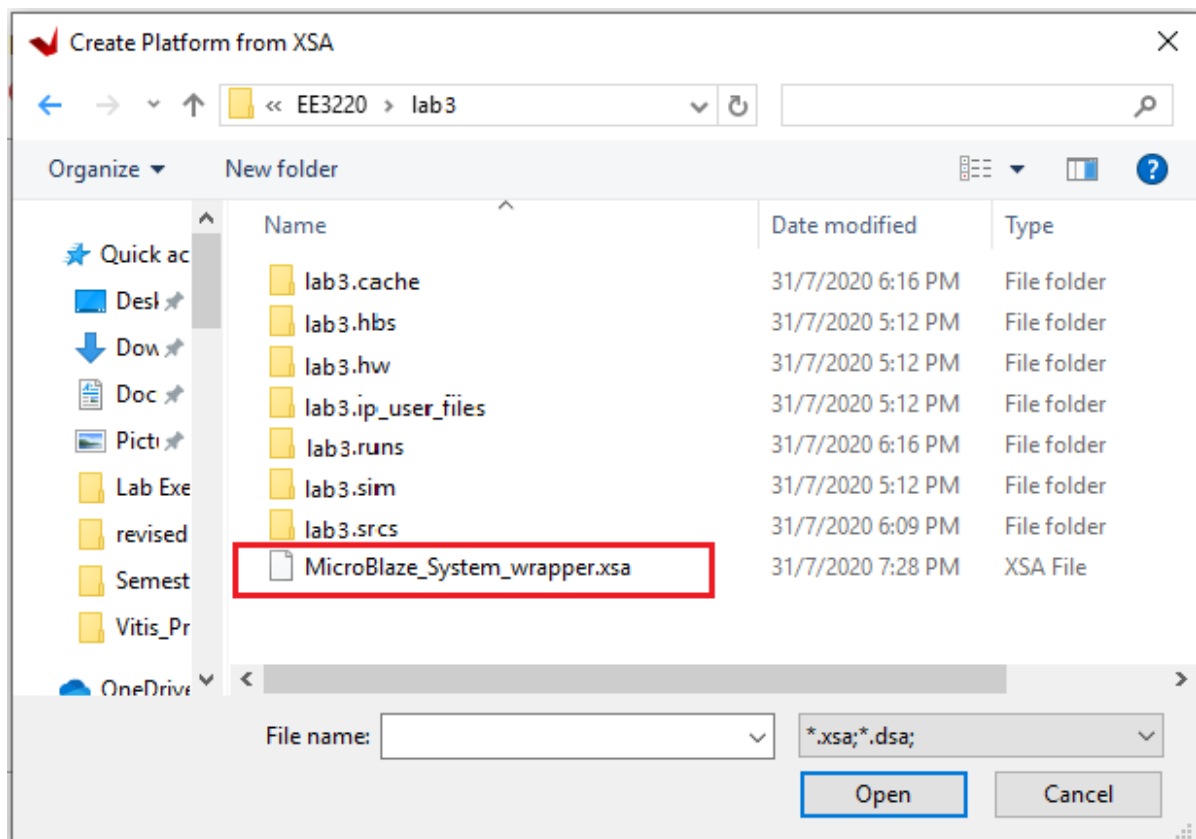
- Choose **Create application project** and click Next



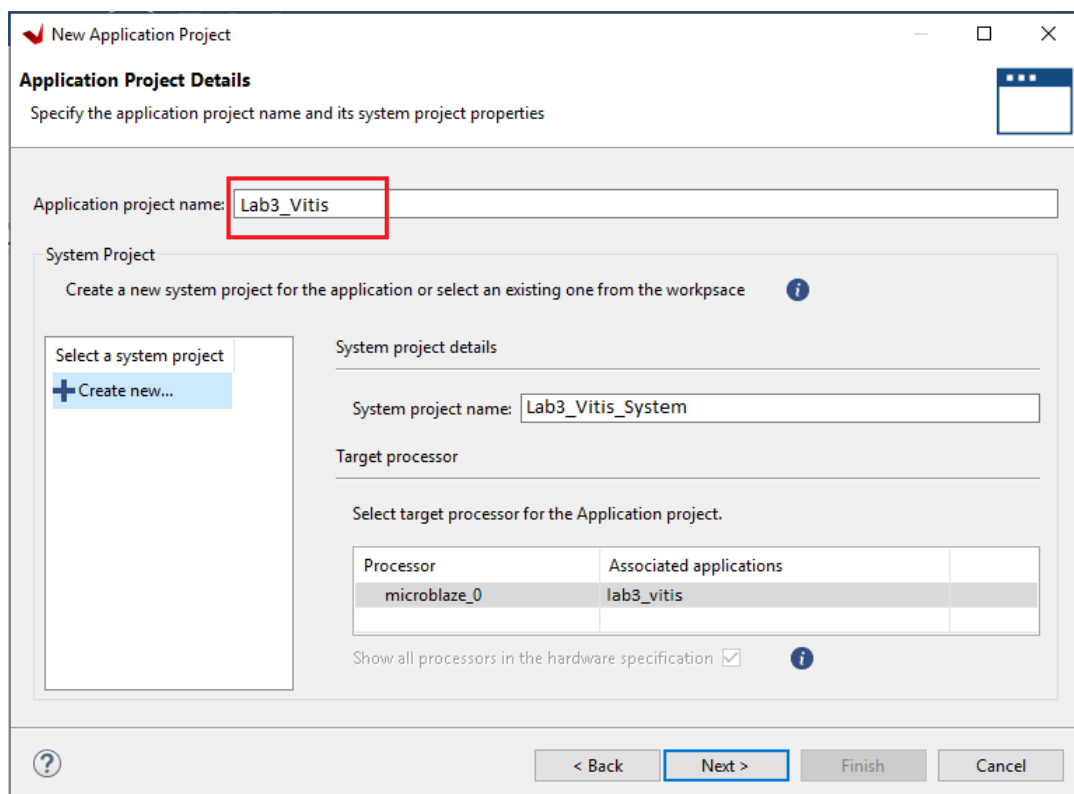


- ♦ Choose **Create a new platform from hardware(xsa)**, browse and go to the lab3 project folder to select the **MicroBlaze_System_wrapper.xsa**. Click Next and leave other settings as default, then, Finish.





- Specify the project name (lab3_vitis), click **Next**.



- Click **Next**

New Application Project

Domain

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain
+ Create new...

Domain details

Name: domain_microblaze_0

Display Name: domain_microblaze_0

Operating System: standalone

Processor: microblaze_0

Architecture: 32-bit

< Back Next > Finish Cancel

- ◆ Choose **Empty Application** and click Finish. Our new application project will be created

New Application Project

Templates

Select a template to create your project.

Available Templates:

Find:

SW development templates

Dhrystone

Empty Application

Empty Application (C++)

Hello World

lwIP Echo Server

lwIP TCP Perf Client

lwIP TCP Perf Server

lwIP UDP Perf Client

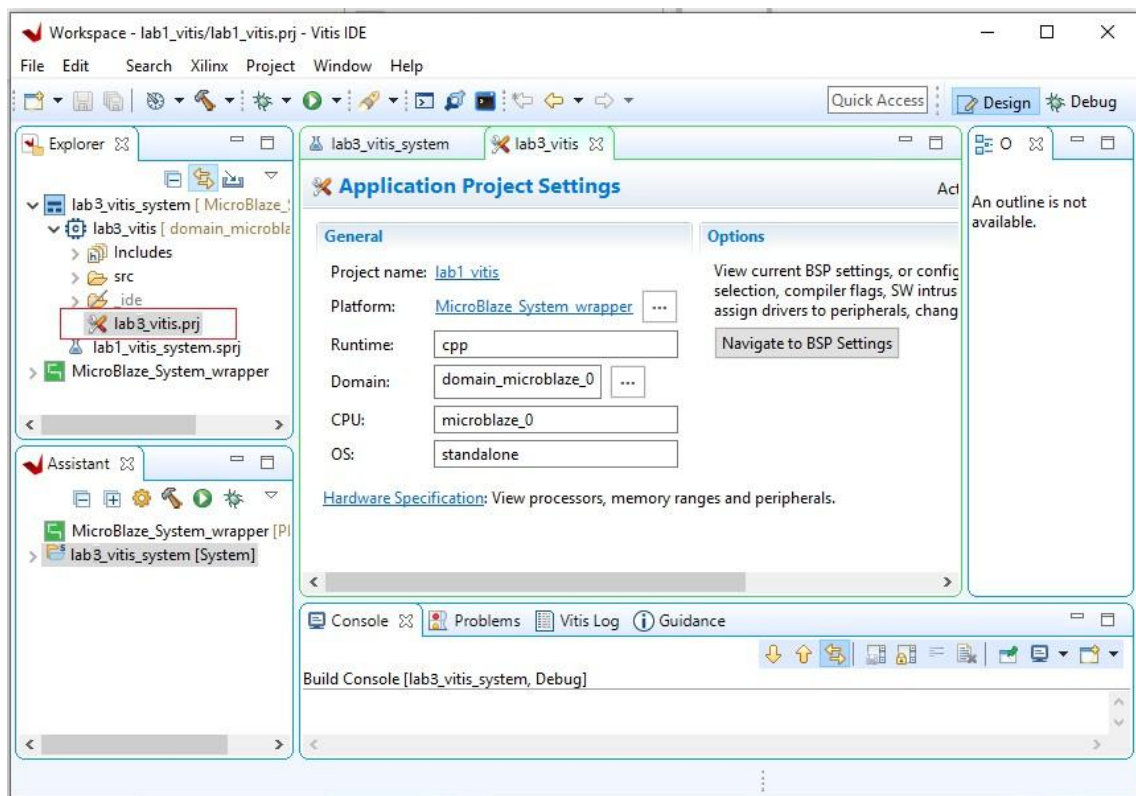
lwIP UDP Perf Server

mba_fs_boot

Empty Application

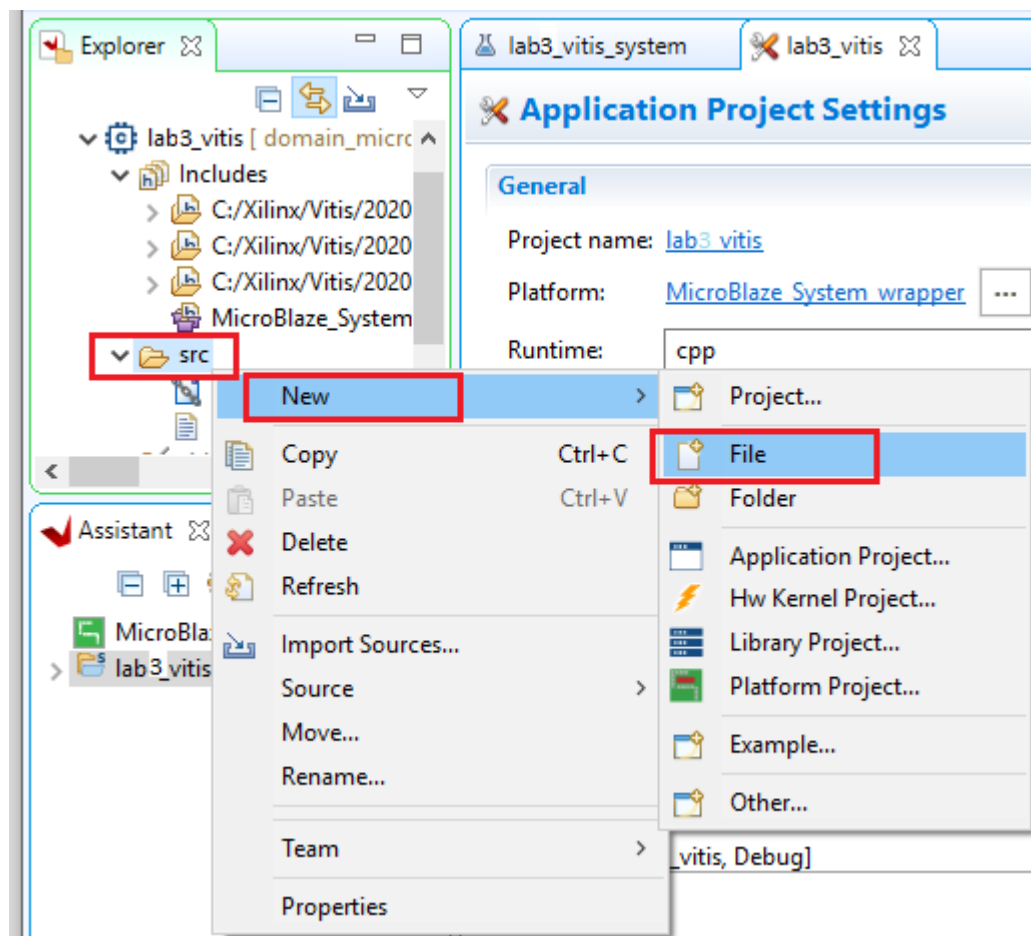
A blank C project.

< Back Next > Finish Cancel

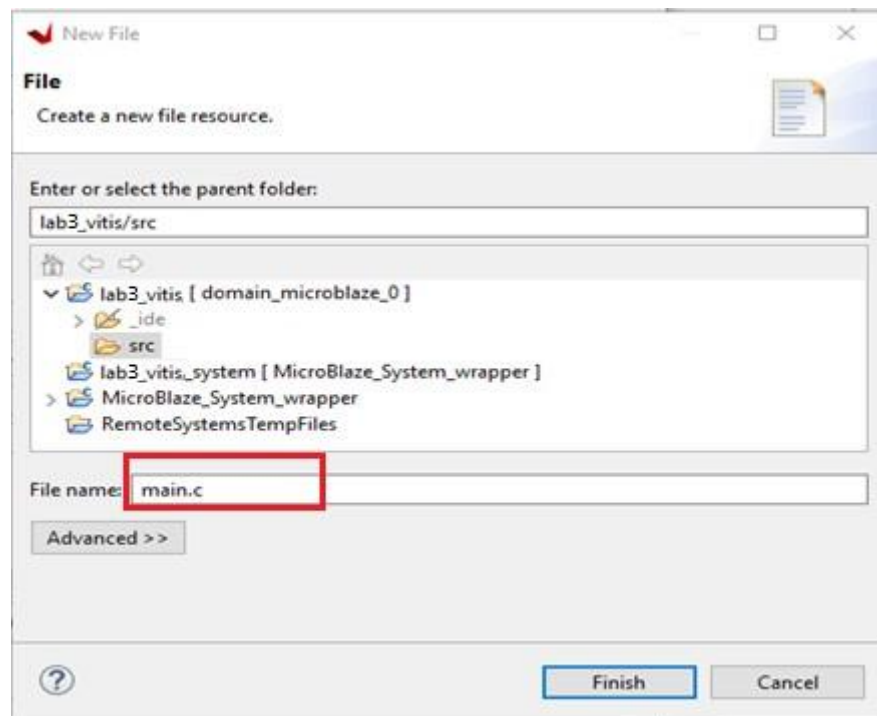


2. Create a new source code file (main.c)

- Right click **src** folder, select **New** and **File** to create a new source code file.

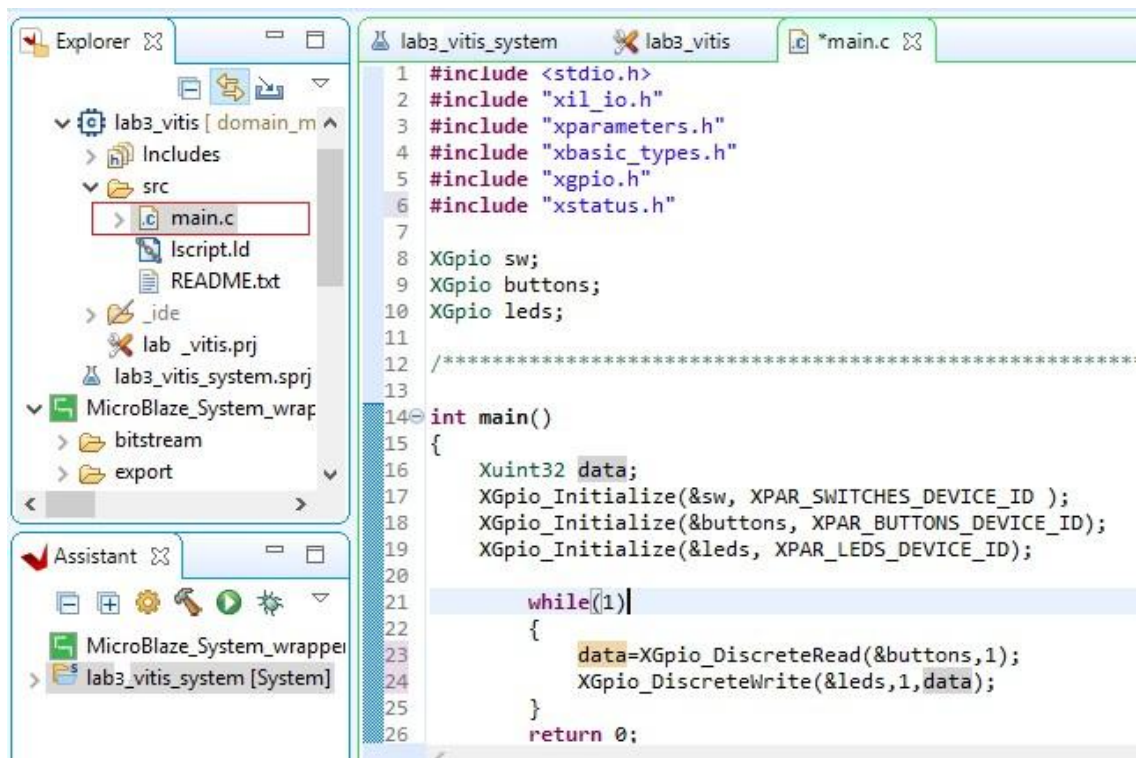


- Name the source file as **main.c**, click finish. A new source file will be created

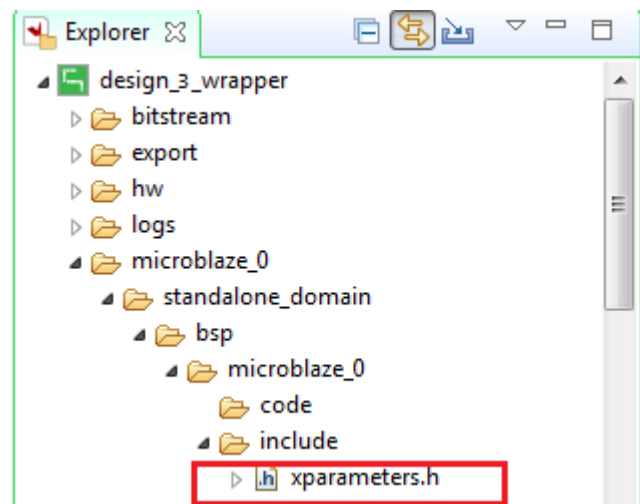


3. Write a C program to control the LEDs, build the project, and program the FPGA.

- Open the **main.c** file, copy and paste the source code provided



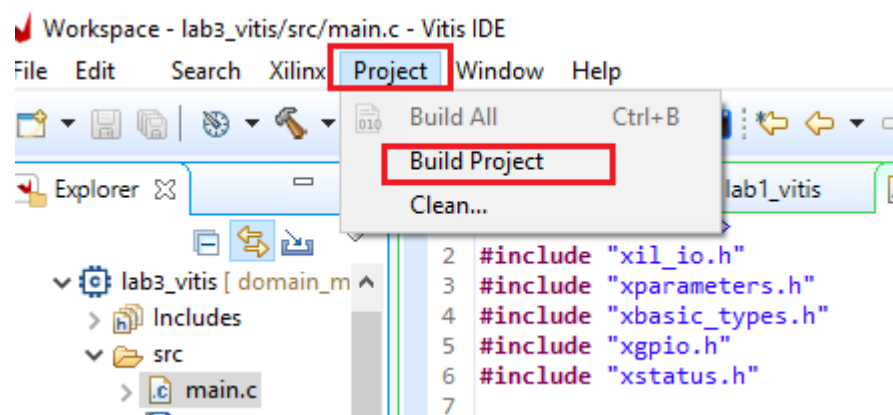
- To understand the definitions used for the GPIO peripherals, check the **xparameter.h**
- XGpio_DiscreteWrite() is used to write on the leds (GPIO output). The XGpio_DiscreteRead() could be used to read buttons or switches in case you need to do it in the future.

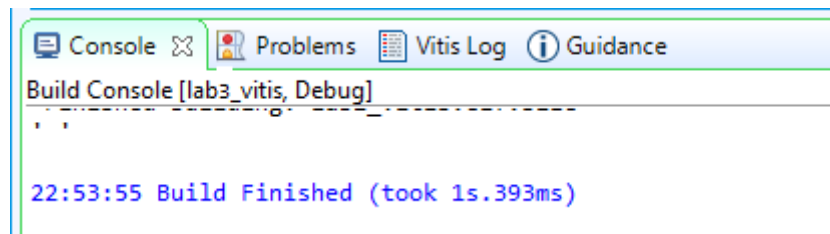


```
#include <stdio.h>
#include "xil_io.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xstatus.h"
#include "sleep.h"
XGpio leds;
XGpio switches;
XGpio buttons;
/*****
int main()
{
    xuint32 data;
    //Xuint32 i;
    XGpio_Initialize(&leds, XPAR_LEDS_DEVICE_ID);
    //XGpio_Initialize(&switches, XPAR_SWITCH_DEVICE_ID);
    //XGpio_Initialize(&buttons, XPAR_BUTTON_DEVICE_ID);

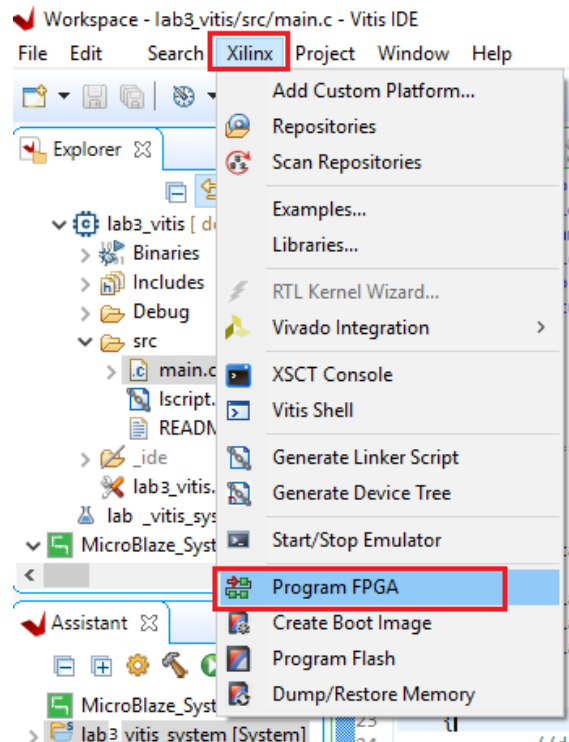
    while(1)
    {
        //data = XGpio_DiscreteRead(&buttons, 1);// can use this to read button
        data=0x01;
        XGpio_DiscreteWrite(&leds,1,data);
    }
    return 0;
}
```

- ♦ Open the **Project** menu and choose the **Build Project**





- ♦ Open the **Xilinx** menu, choose **Program FPGA**. You should also connect your board to your computer and turn it ON before you can program the board.



- ♦ Browse the project folder for the Bitstream file and the .mmi file. You may need to change the search file type to mmi to be able to see the .mmi file. if you use default location, the files are located at:

Bitsream: <

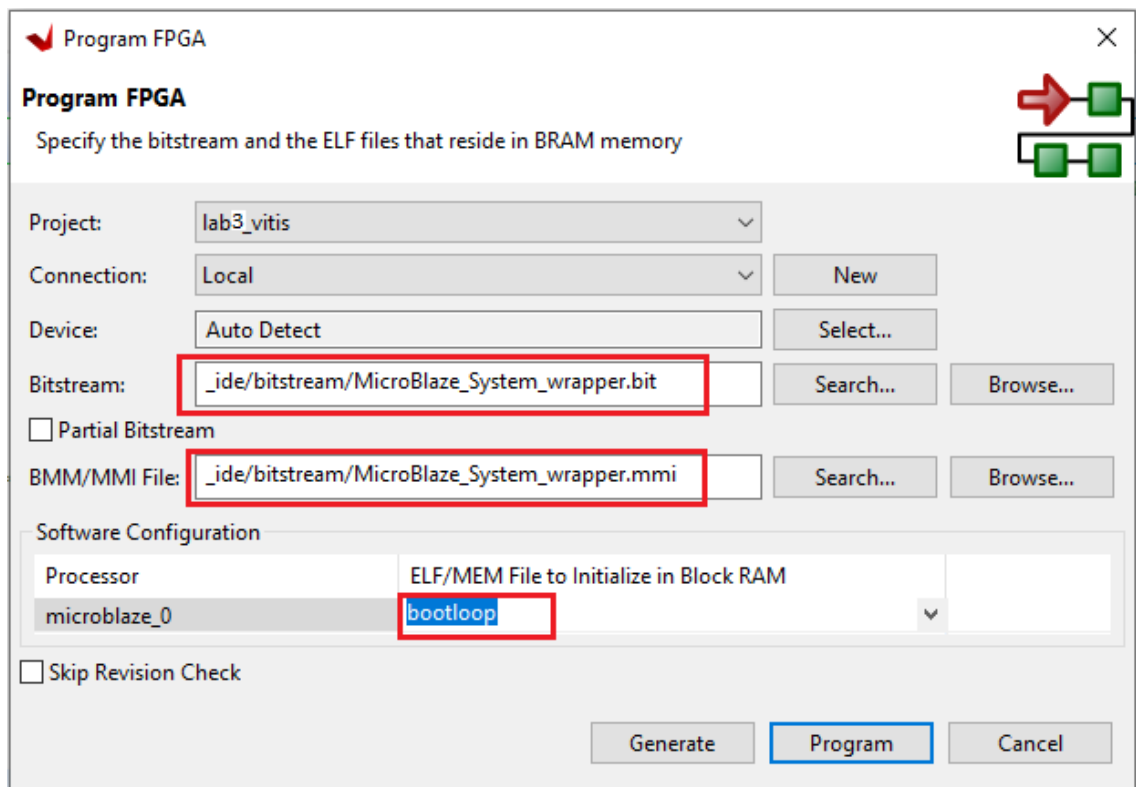
C:\Users\EEUsers\Workspace\lab3_vitis_ide\bitsream\microBlaze_System_wrapper.bit >

.mmi file: <

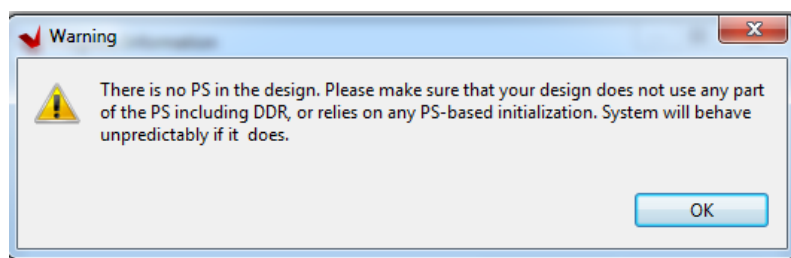
C:\Users\EEUsers\Workspace\lab3_vitis_ide\bitsream\microBlaze_System_wrapper.mmi >

.elf file: < C:\Users\EEUsers\Workspace\lab3_vitis\Debug\lab3_vitis.elf >

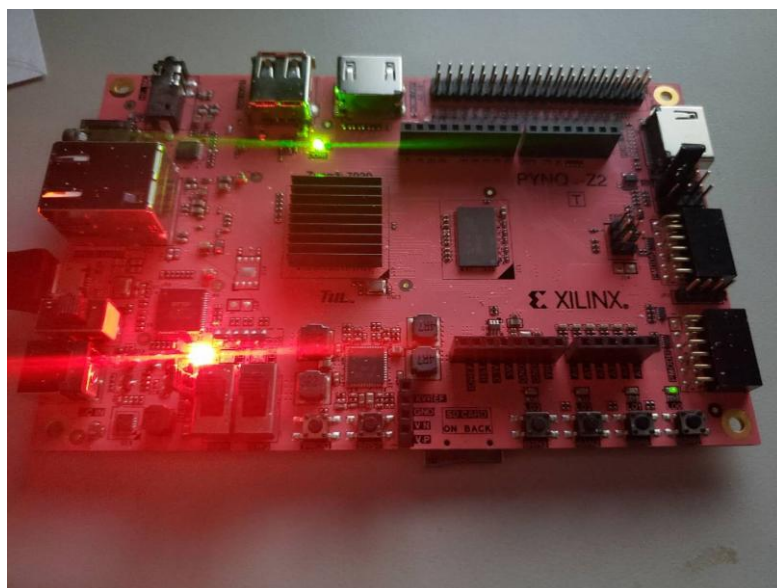
- ♦ For the elf/mem file, you should choose your project's elf instead of the **bootloop** already there, go to the project folder and select "<project folder> \Debug\lab3_vitis.elf".



- ◆ Wait and click OK.



- ◆ Now all are done. You can now see the leds turning on according to value set in the program



Check Point 3:

Modify the program to ON the LEDS corresponding to the switches and the buttons. If any button or switch is ON, the LED infront of it should ON. Show your work to the lab supervisor Take snapshots for your lab report.

Check Point 4:

Modify the program to show a binary counting from 1 to 7 (**0001** to **0111**) on the LEDs. Take snapshots or small video clip as your evidence of achieving this check point in the lab report.

[**Hint:** You can use a **for loop** and the **usleep(int delay)** function for the delay. The delay is written in microseconds. You are advised to use 250ms delay or below i.e. **usleep(250000)**. The statement below the Bonus is an example of a for loop.]

Bonus:

Use buttons to display a binary count of up to 8 on the LEDs. The binary number increments after every button press. Show your work to your lab supervisor.

[Hint: Check if the button is pressed and turn on the leds if true. Reset the count to zero when it reaches 8. You may use the for loop delay below for button debounce.]

```
for( i=0; i <= 0x00FFFFFF; i++) {};
```

~ END ~