EE3220: System-On-Chip Design

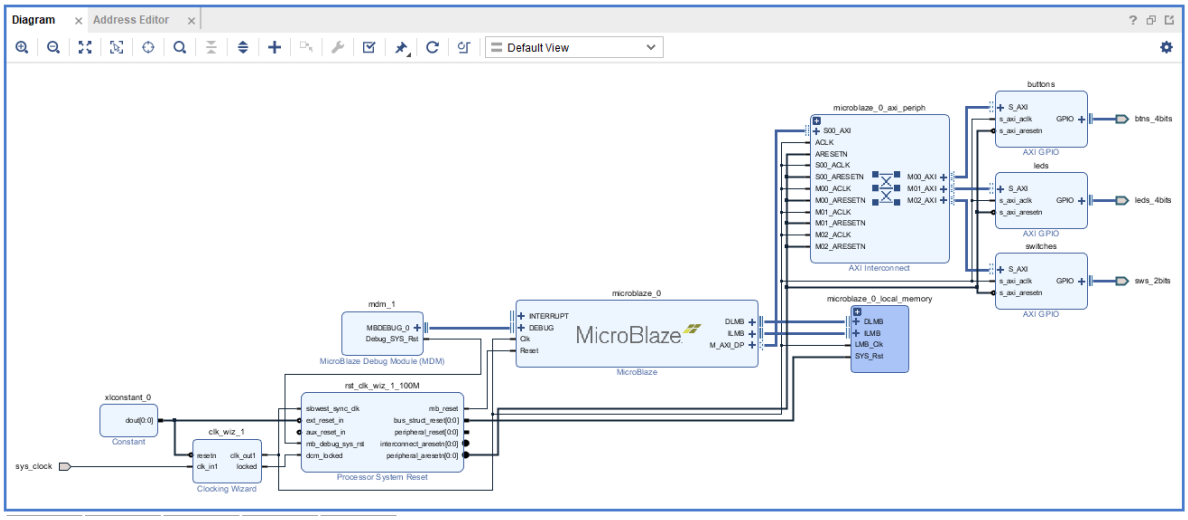
Lab 3: Designing Embedded System with MicroBlaze

Objectives

At the end of this lab, student 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 added in this lab will be LEDs, switches and buttons (only LEDs will be used in the program). We will add the components (APIs) of the peripherals to the Microblaze and then write a C program to control them. The programmable logic will be built in Vivado hlx while the FPGA and the code programming will be achieved in Vitis IDE. Students should control the LEDs on the FPGA by writing a C program for the MicroBlaze to execute. For the online nature of this lab, only the LED will be controlled from the program (no button/switch). 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 instantiate a MicroBlaze soft processor core and later add some peripherals to it i.e. LEDs, switches and buttons. We will use the block design approach in the Vivado and add the IPs for the Microblaze and the peripherals. The whole system will be synthesized, implemented and 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:

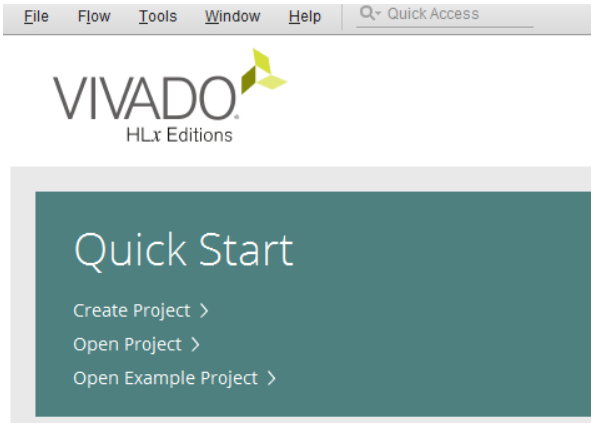
1. Add the Pynq board to Vivado for the board recognition.

Since Vivado does not contain the PYNQ board by default, we need to add the board files to the Vivado directory to the 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.

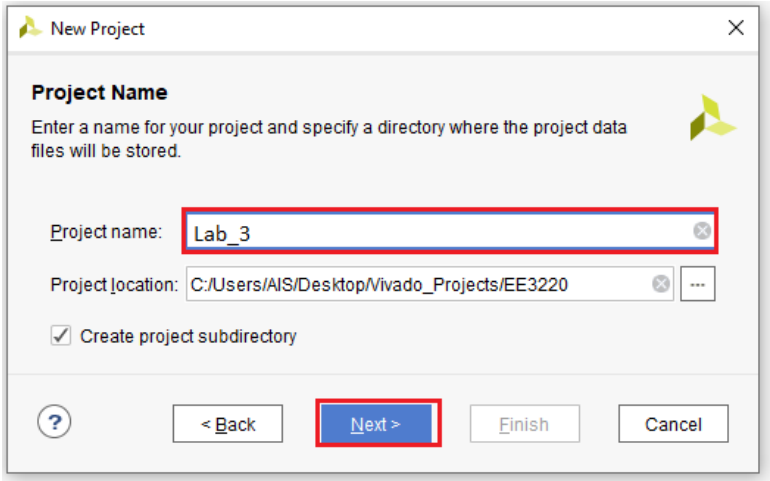
- Go to canvas to download and extract the PYNQ z2 board files.
- Copy the extracted folder to the location given below
  <Xilinx installation directory>\Vivado<version>\data\boards\board_files)

2. Create new Vivado project.

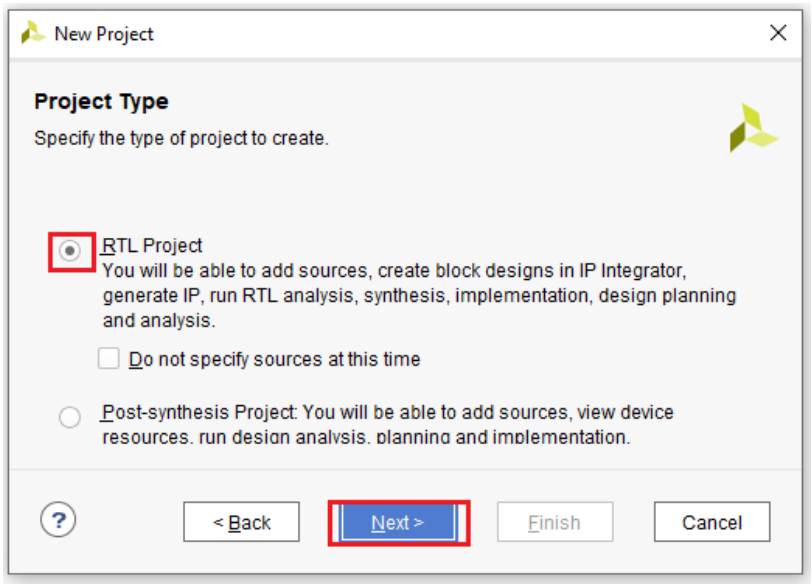
- Open Vivado, click **Create Project**, click next.



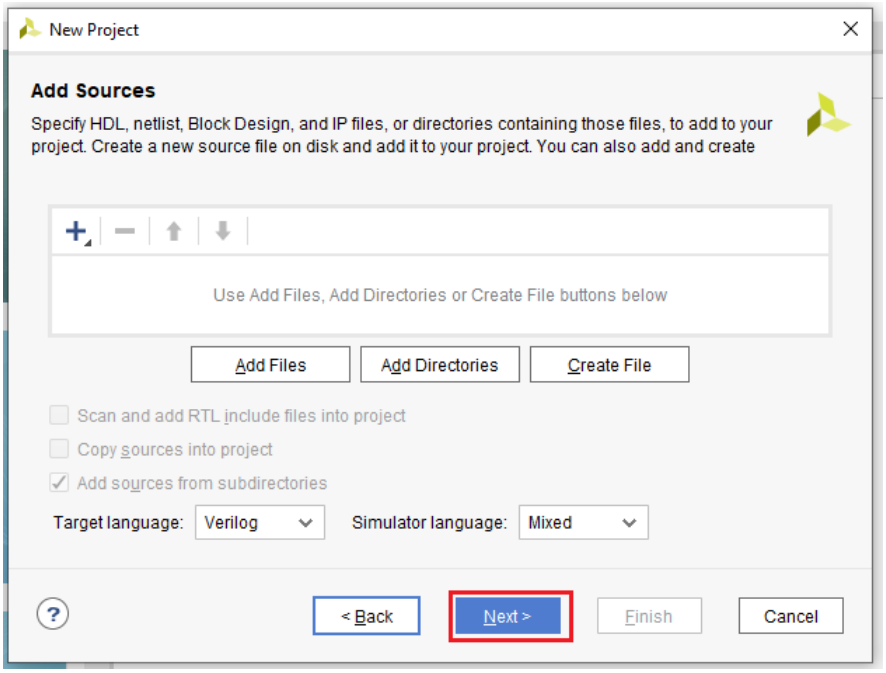
- Choose a location for your project and specify the project name, then click **Next**.



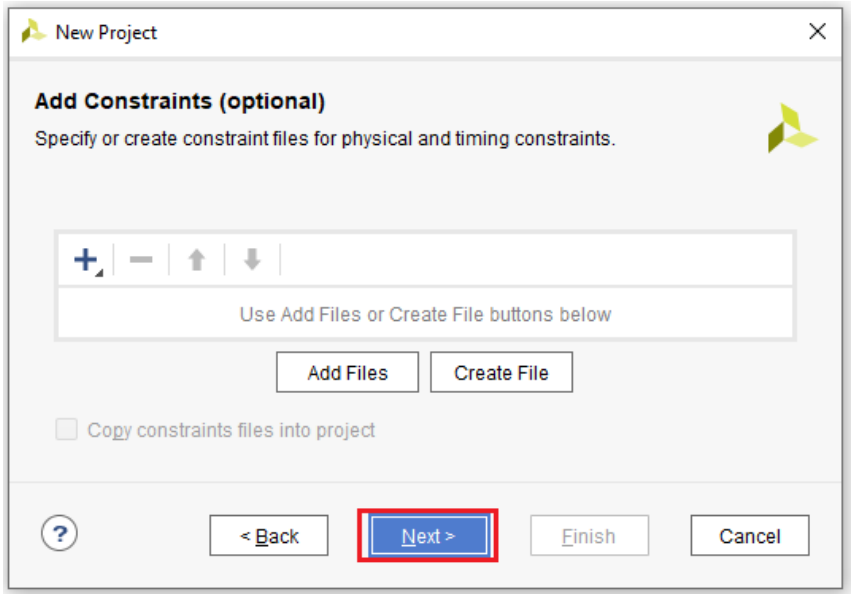
- Choose **RTL Project**, then click next



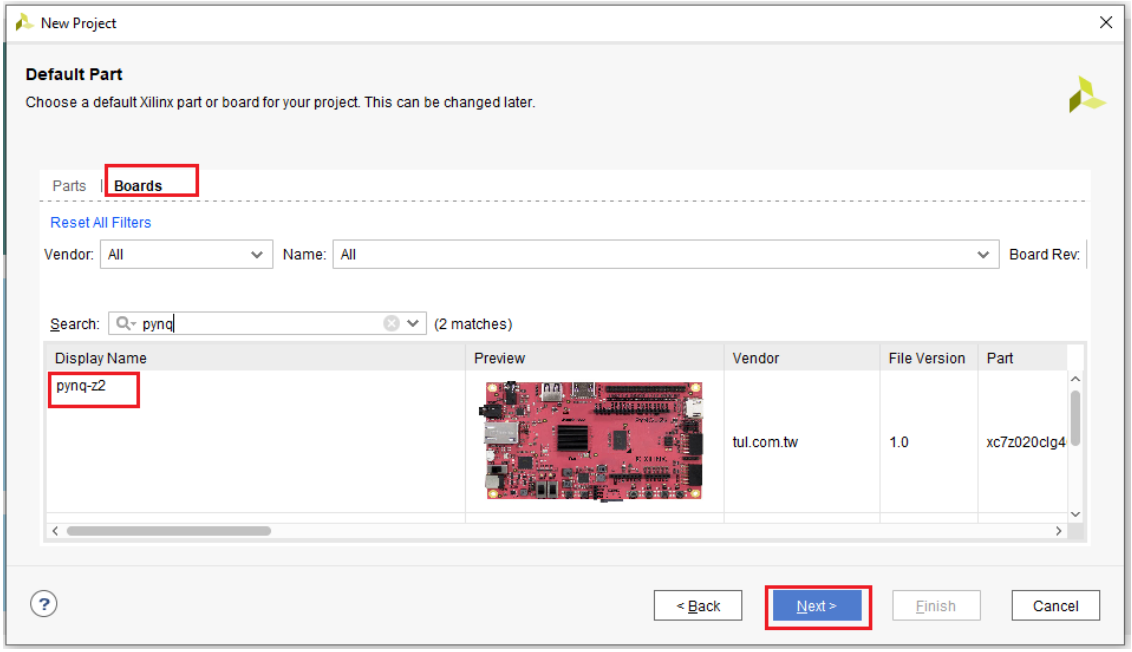
- Click **Next** to create an empty project with no source code



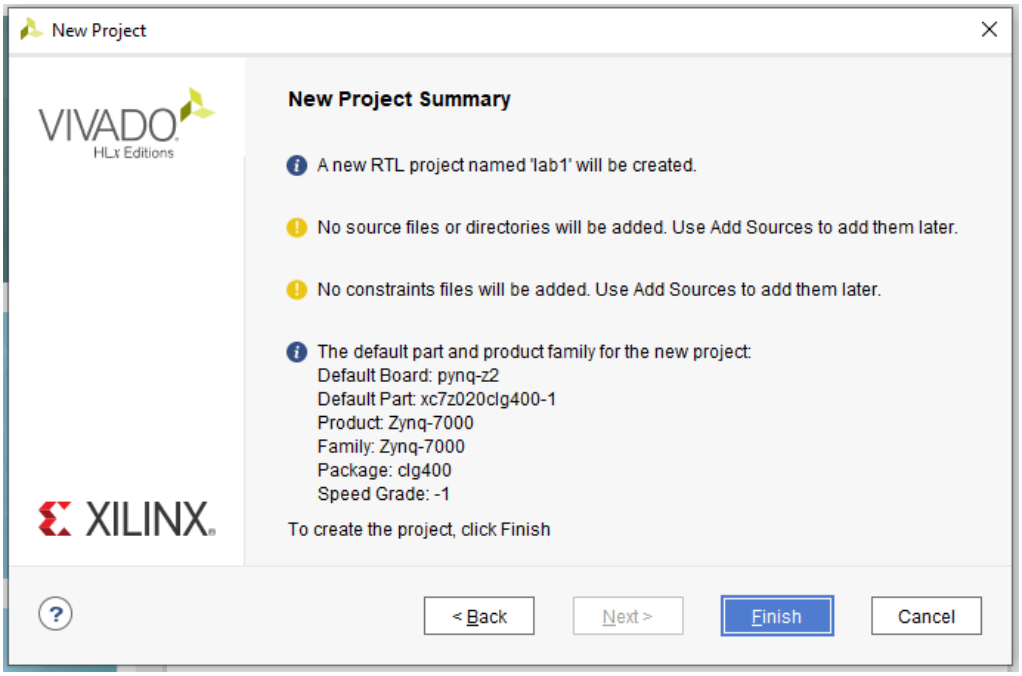
- Click **Next** to continue without adding any constraint



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

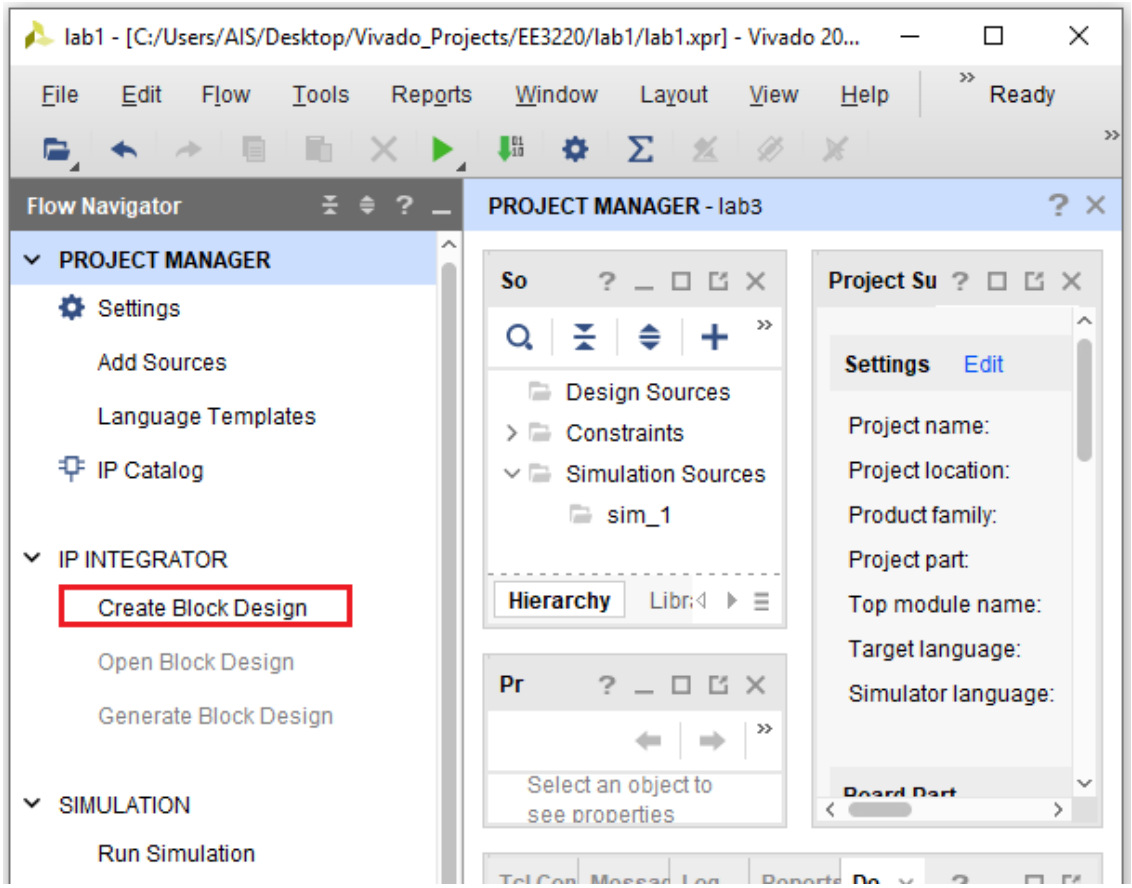


- Cross check the summary and click **Finish** to create the project

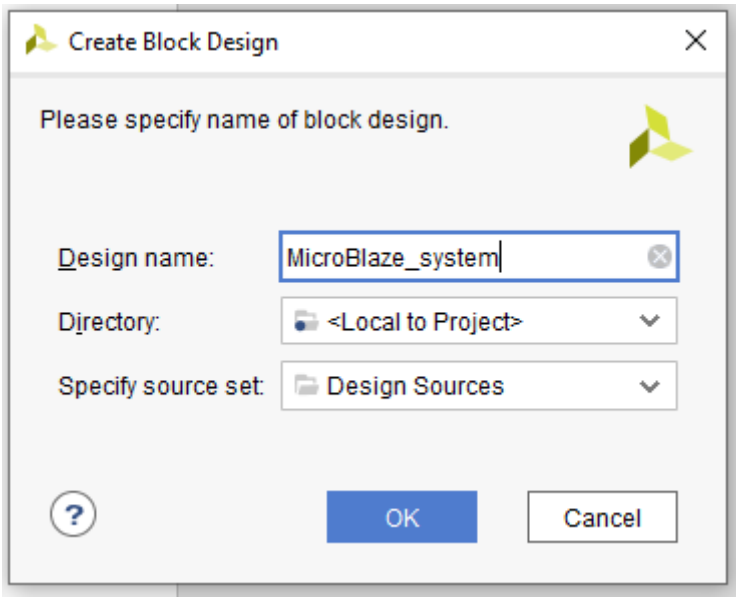


2. Instantiate the Microblaze core using the block design

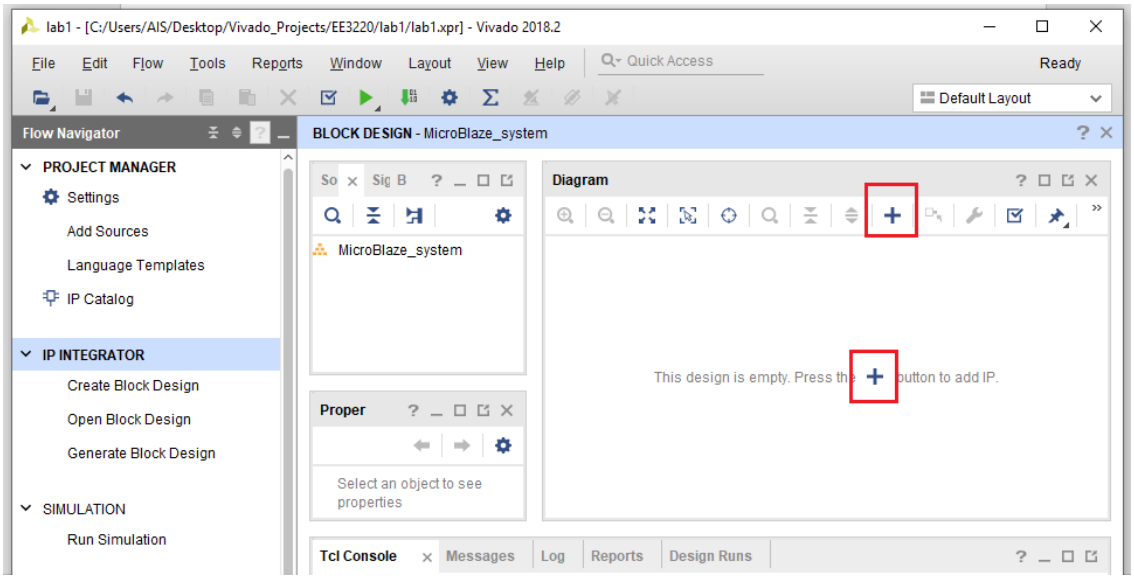
- In flow navigator, click **Create Block Design**, then click ok



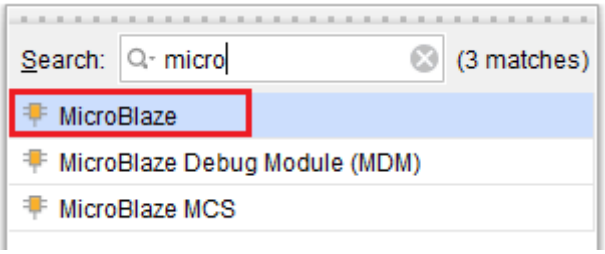
- Specify the design name and click **OK**



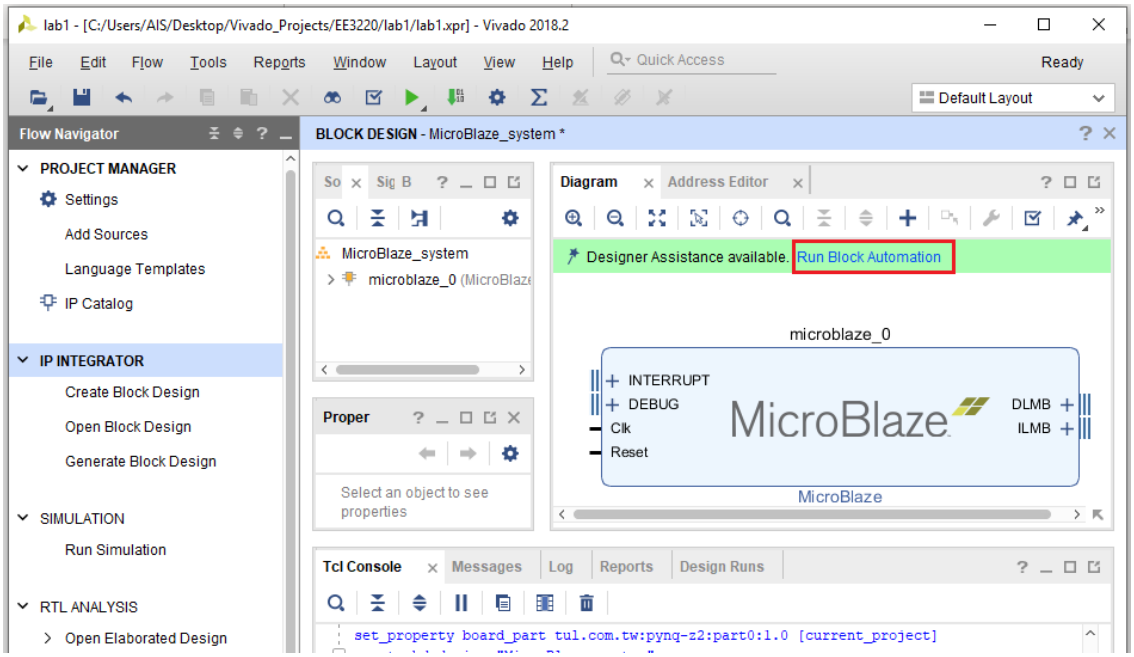
- Click the **"+"** icon to add IP



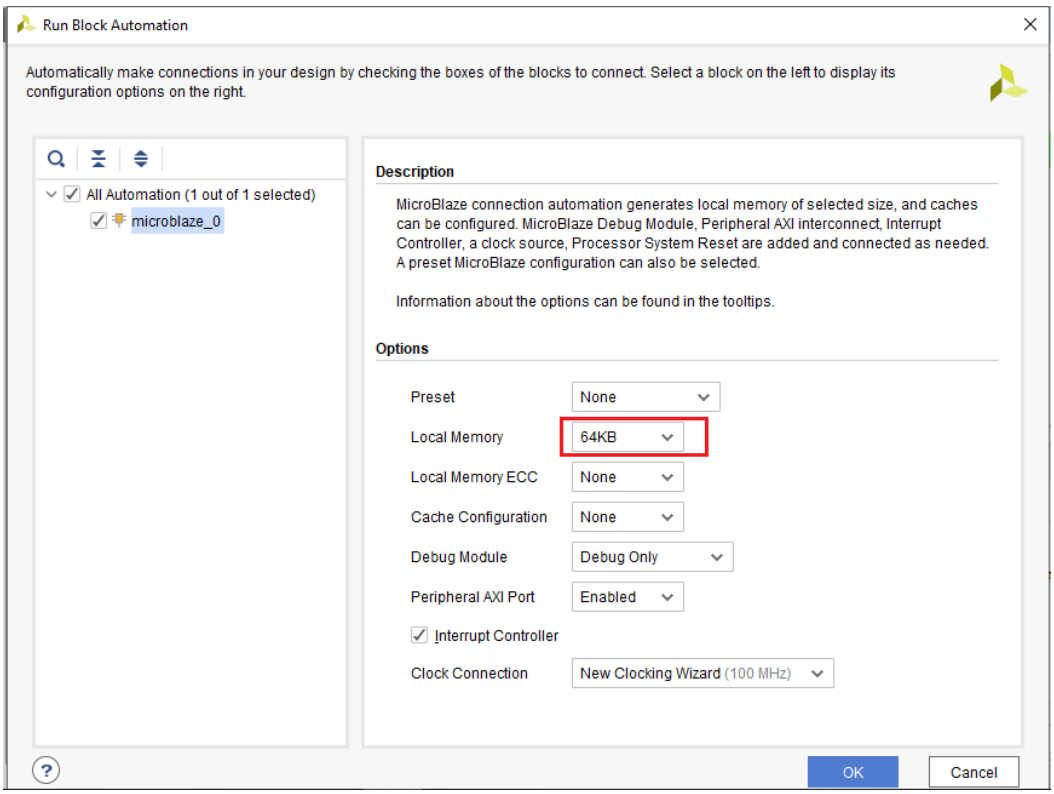
- Search and double click **MicroBlaze**, the MicroBlaze will now be added



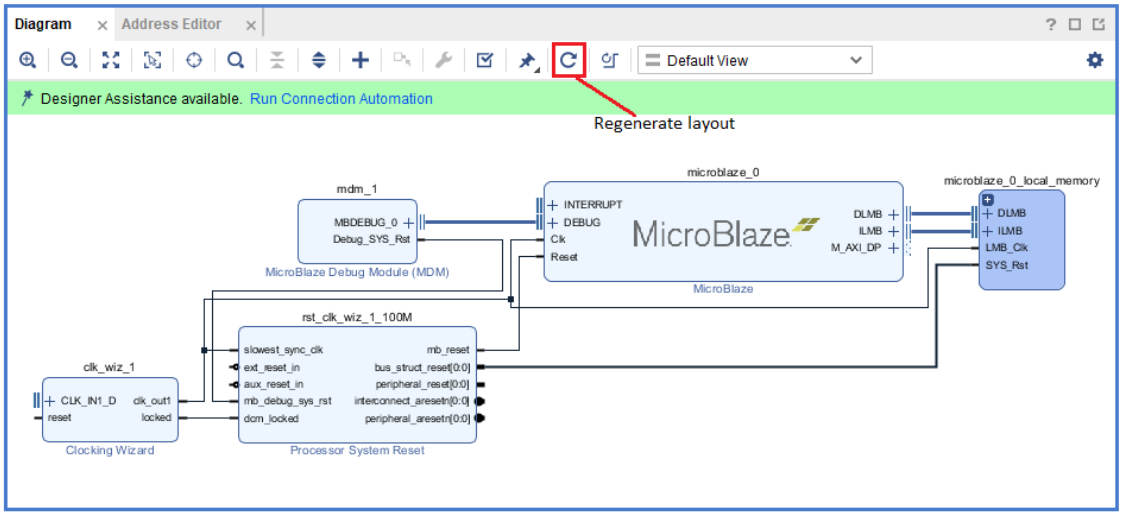
- Click **Run Block Automation**, change memory to 64KB and leave all the other setting as default.



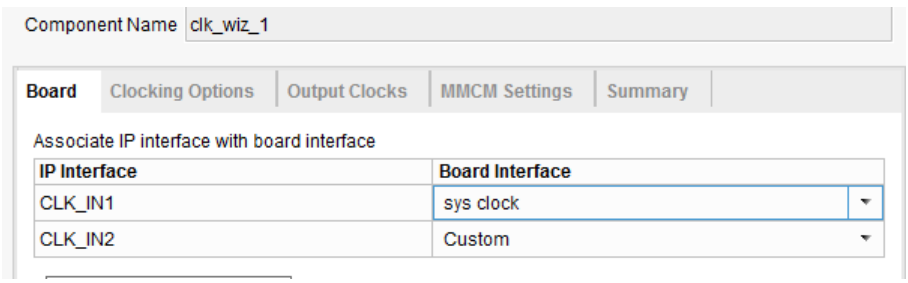
- Click **OK**, the block run automation adds and connects all the necessary modules required for running the MicroBlaze



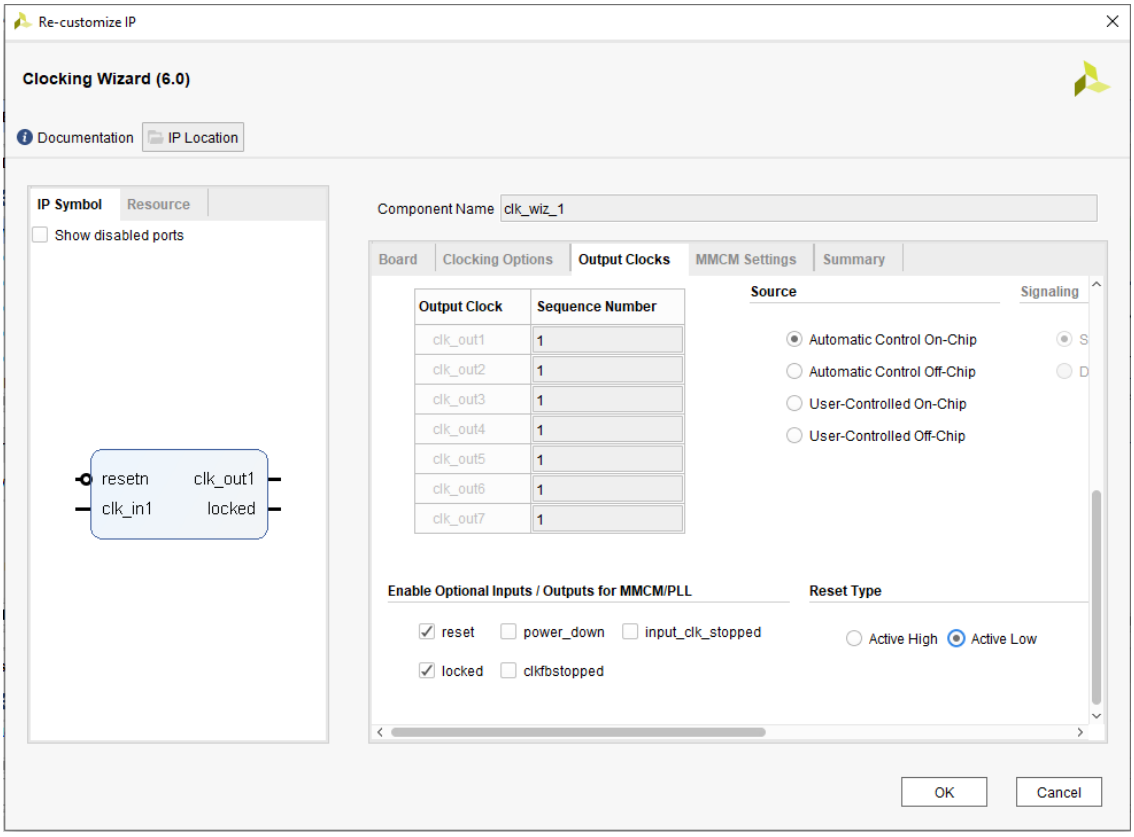
- Click Regenerate layout (a circular arrow like a refresh sign in the block design menu)



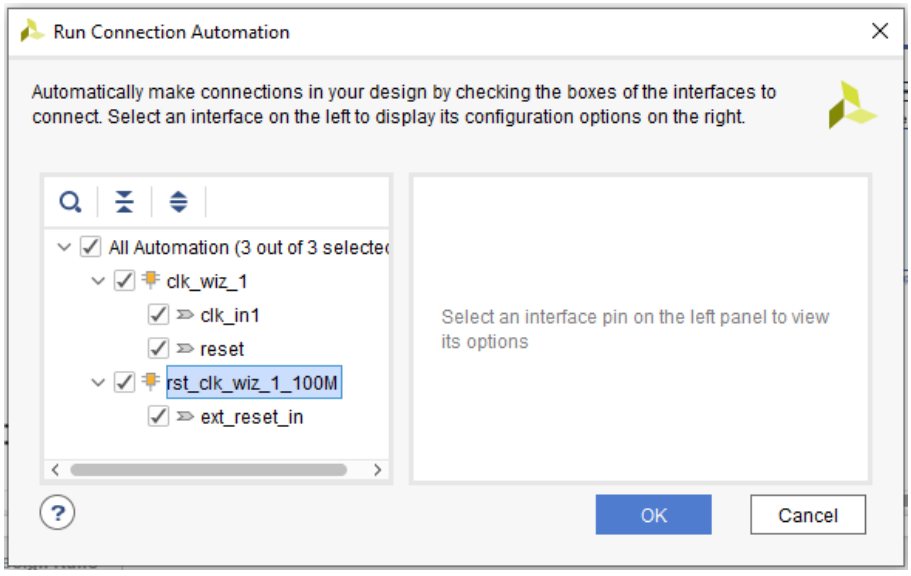
- Double click clocking wizard IP and Choose sys_clock as the source of clk_in1.



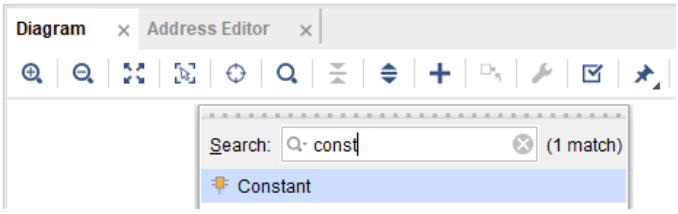
- Select Output Clocks menu and at the bottom change reset type to **Active Low**

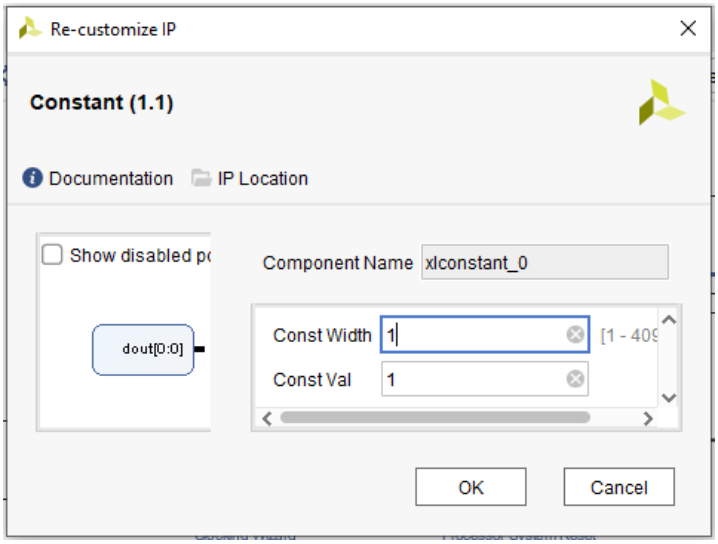


- Run connection automation

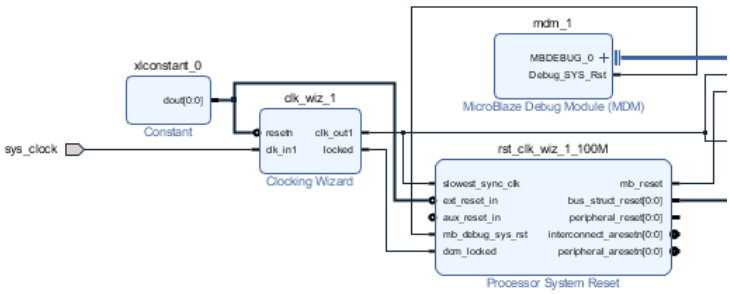


- Delete the **reset_rtl** and the wire connecting the **resetn** of the clock and the **ext_reset_in** of the Processor System Reset component.
- Click the **'+'** tab to add a constant IP, double click the IP and set its value to **1**.





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

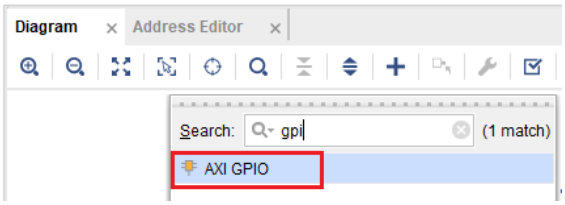


Check Point 1:

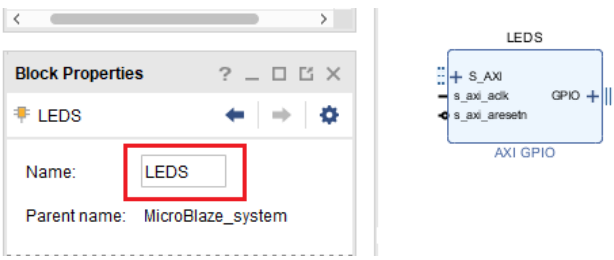
Take the snapshot of the current block design. Proceed to the next tasks.

2. Instantiate the Peripherals (LEDS, switches, buttons)

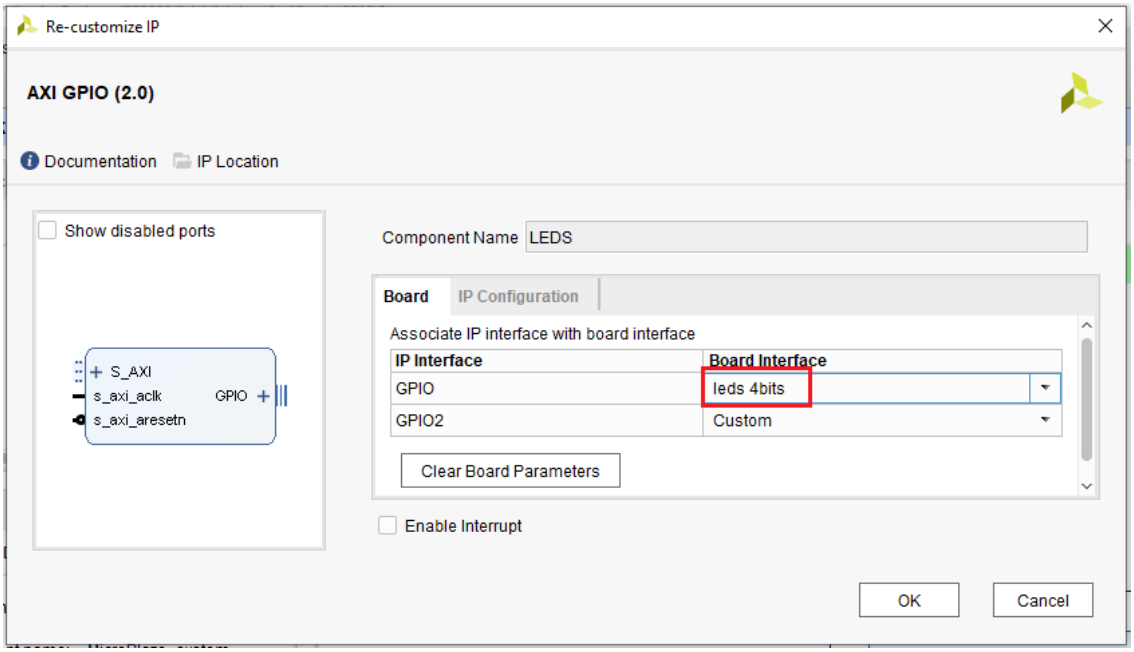
- Click add IP tab (**'+'**) from the block design menu, type **gpio** and select AXI GPIO by double clicking.



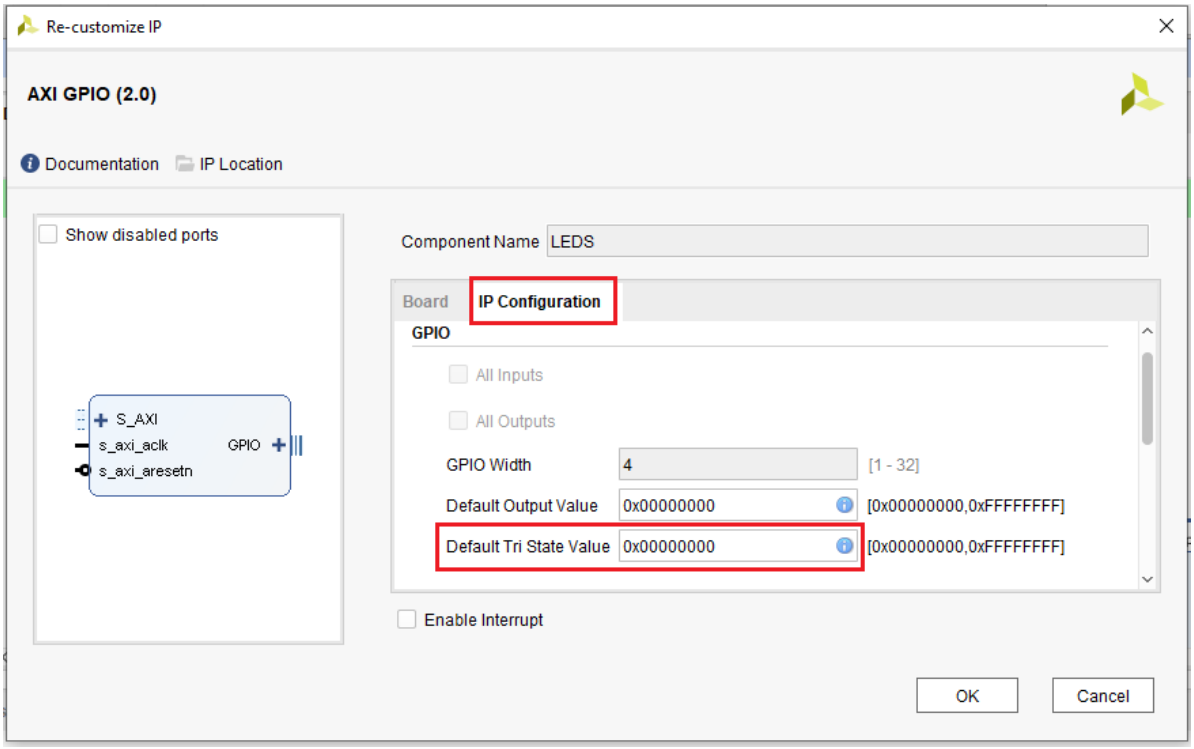
- Select the **AXI GPIO** IP, from the **Block Properties** at the left hand side, rename the GPIO to LEDS.



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

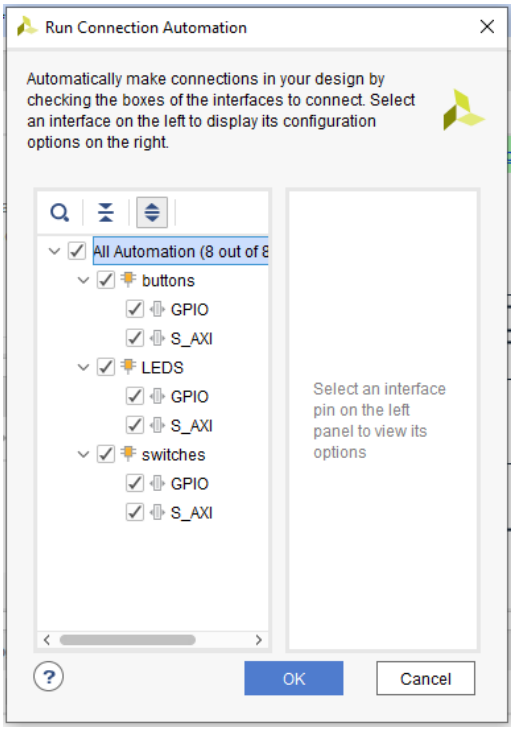


- Click IP configurations and change the **Default Tri state Value** to 0x00000000 (indicating OUTPUT), click OK

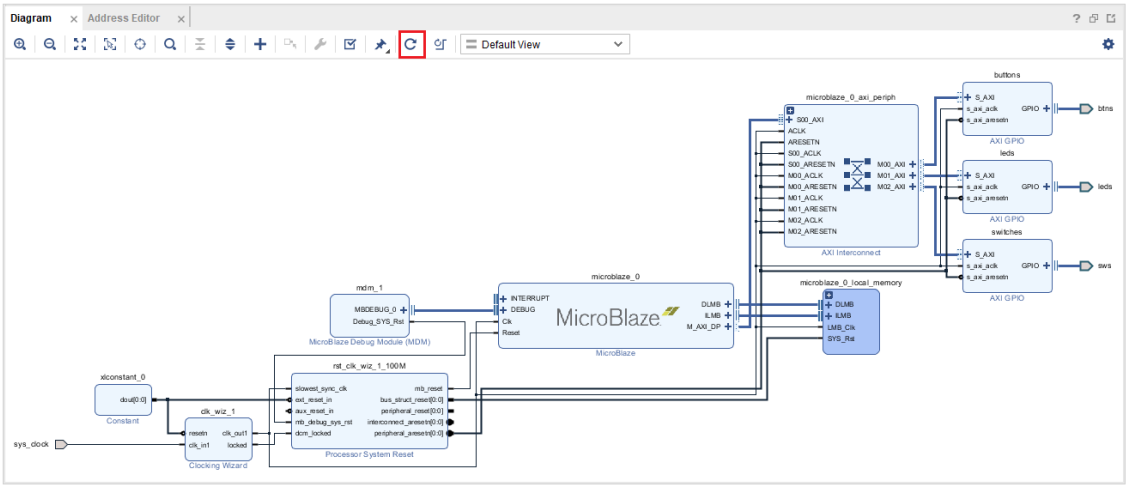


Repeat the above process to add and rename switches and buttons IP to switches and buttons respectively. However, under the Board interface of the GPIO now select sws 2bits and btns 4bits for the switches and buttons respectively. Leave the Default Tri State Value under IP configurations as 0xFFFFFFFF (indicating INPUT) for the buttons and switches.

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

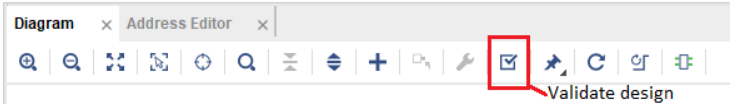


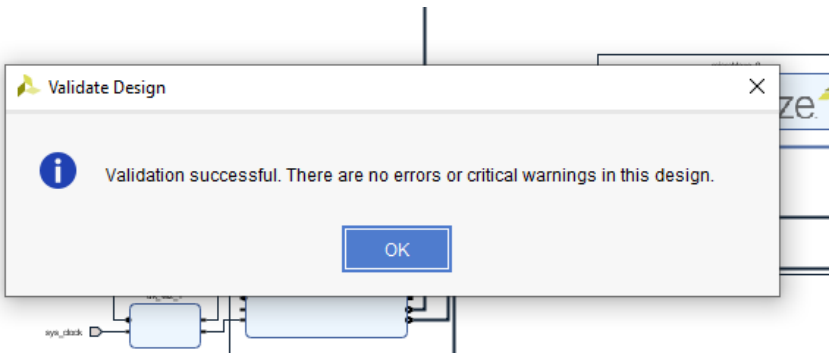
- Click regenerate layout



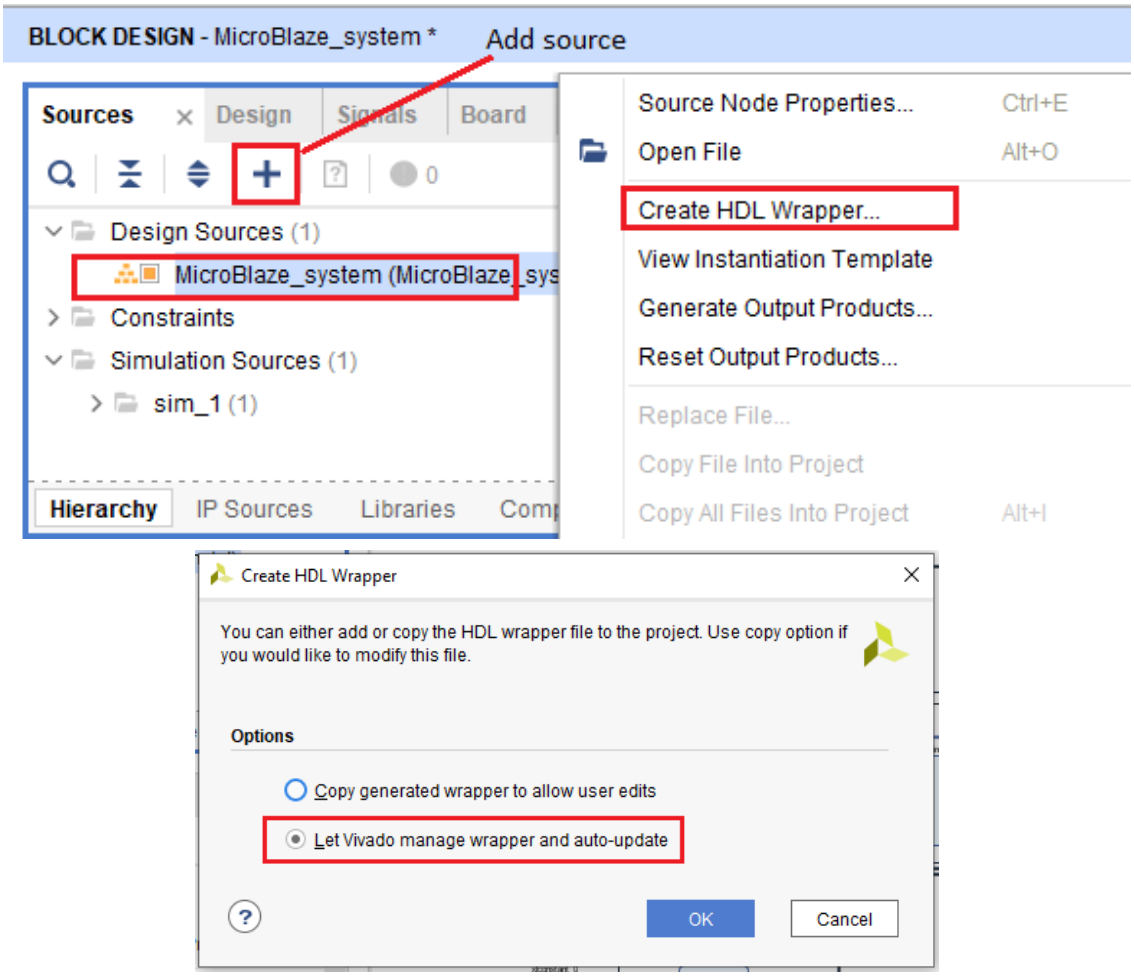
4. Validate the design and create bitstream

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

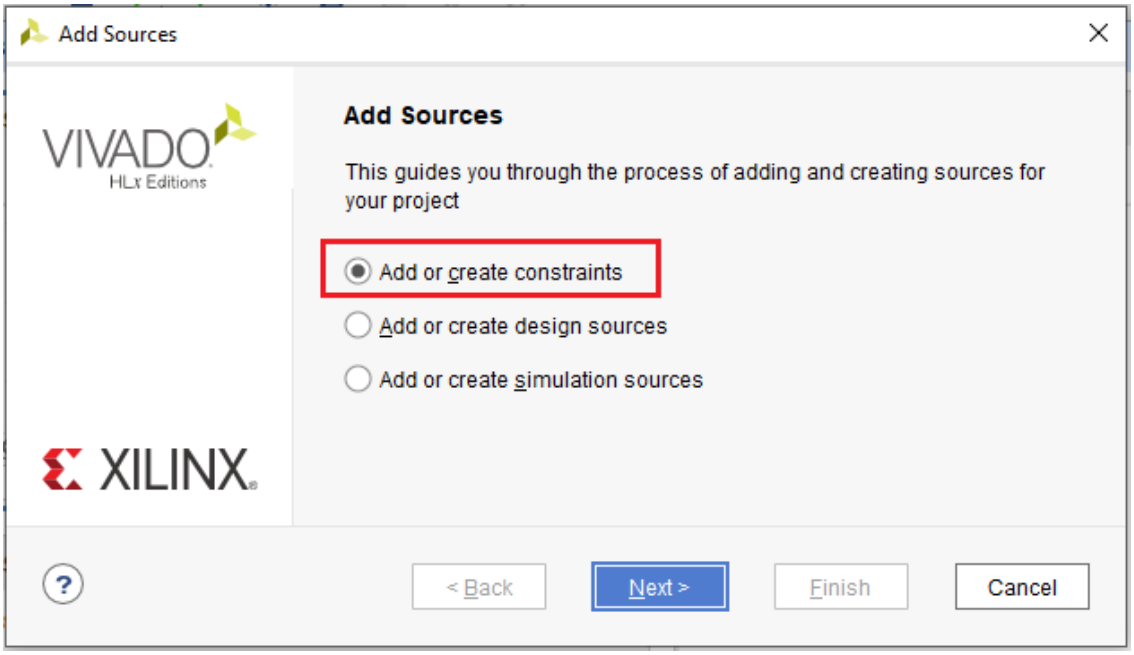




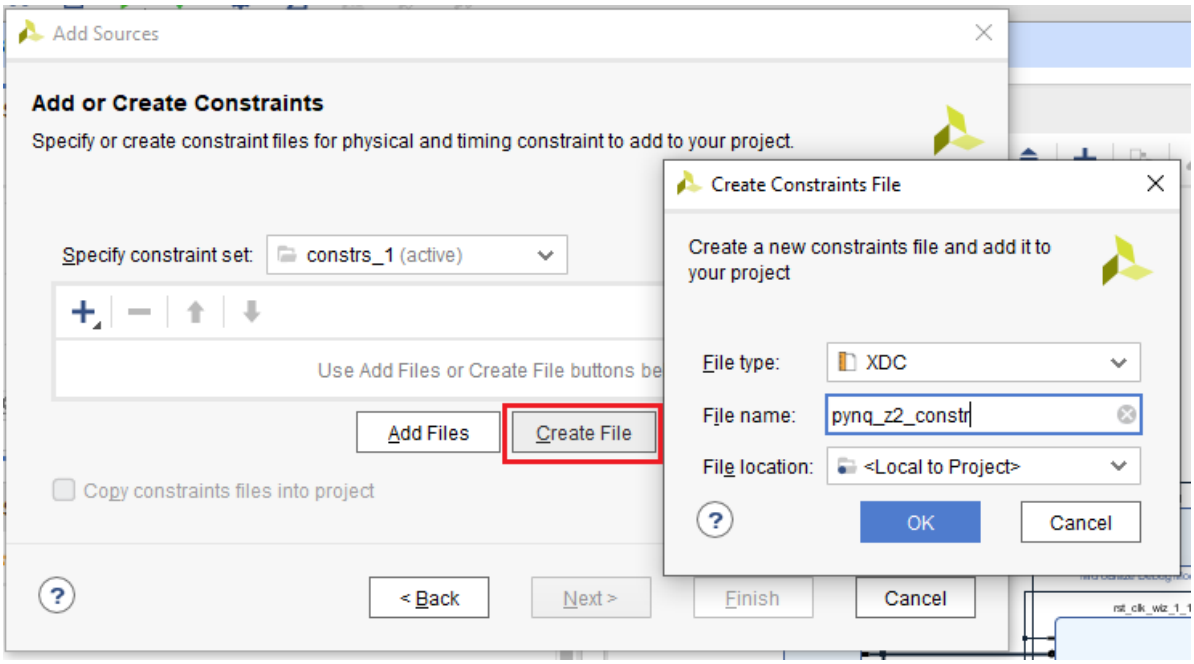
- On the sources panel, right click the block design, choose **Create HDL Wrapper**, click OK.
- Select **Let Vivado manage wrapper and auto-update**, click OK
- Choose add sources (**+**) sign under Sources to create a constraint file,



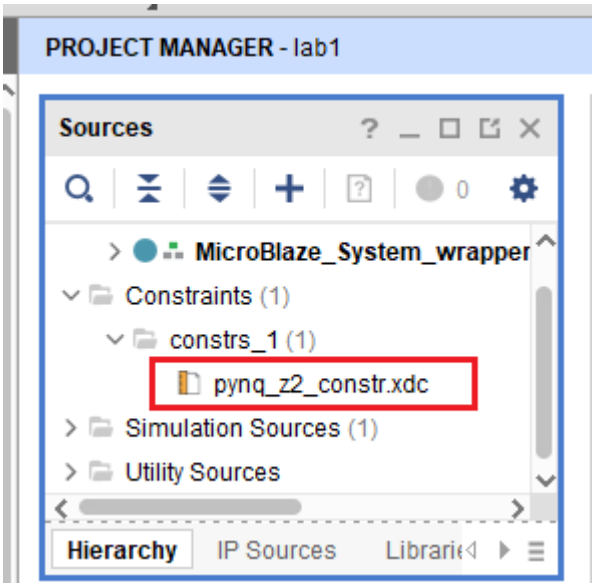
- Select **Add or create constraints**



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

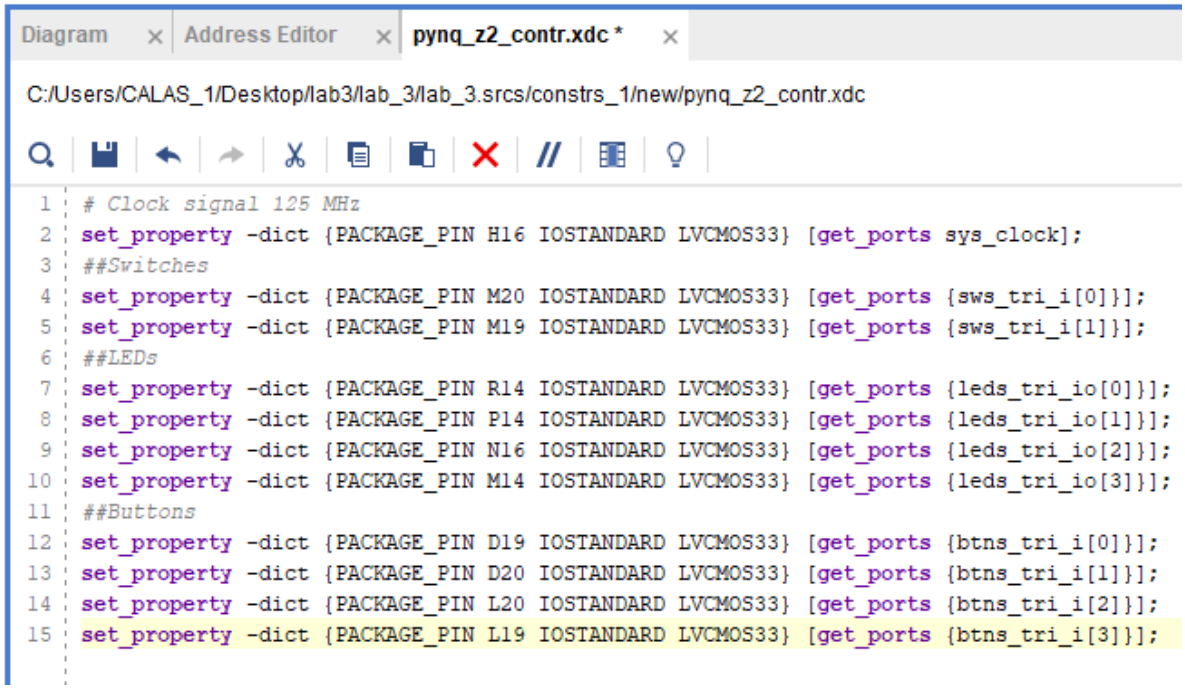


- Double click and open the constraint file you just created , i.e. pynq_z2_constr.xdc

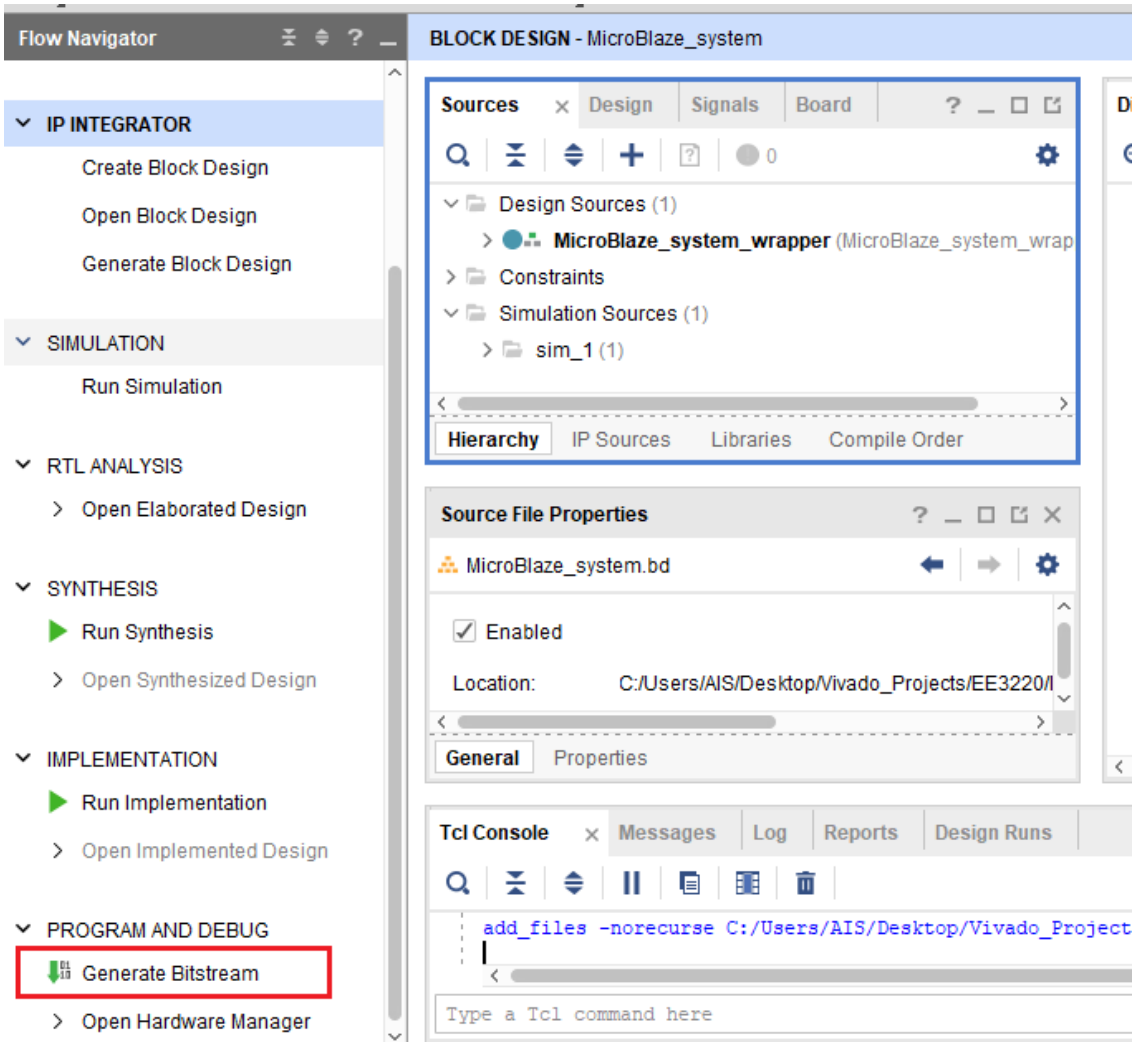


- Enter the constraints we provide you below by copying or typing

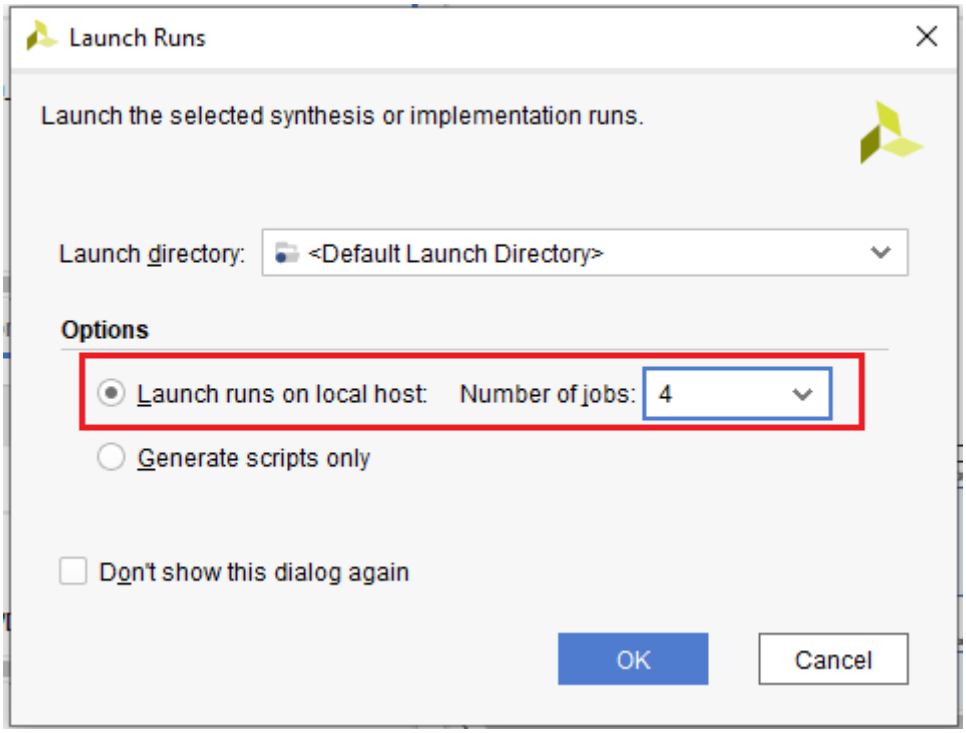
```
# Clock signal 125 MHz
set_property -dict {PACKAGE_PIN H16 IOSTANDARD LVCMOS33} [get_ports sys_clock];
##Switches
set_property -dict {PACKAGE_PIN M20 IOSTANDARD LVCMOS33} [get_ports
{sws_tri_i[0]}];
set_property -dict {PACKAGE_PIN M19 IOSTANDARD LVCMOS33} [get_ports
{sws_tri_i[1]}];
##LEDs
set_property -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]}];
```



- In the Flow Navigator, click generate bitstream

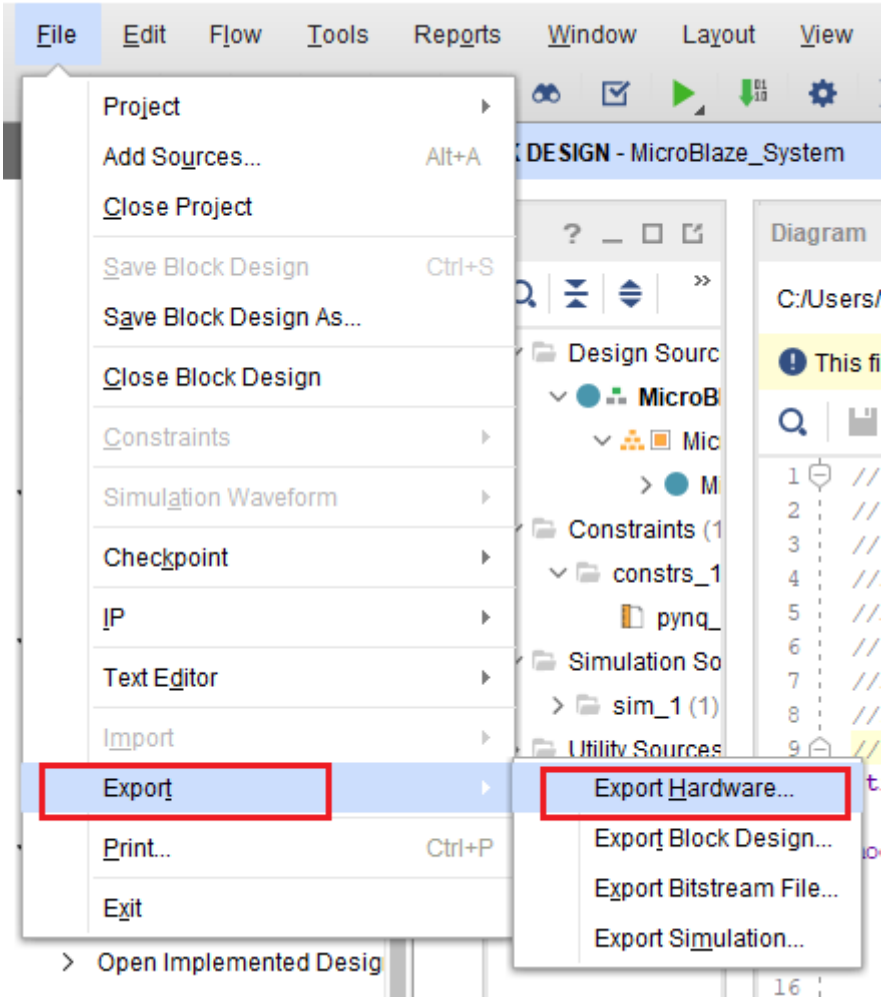


- Select **Launch runs on local host**, choose the **Number of jobs** and click OK

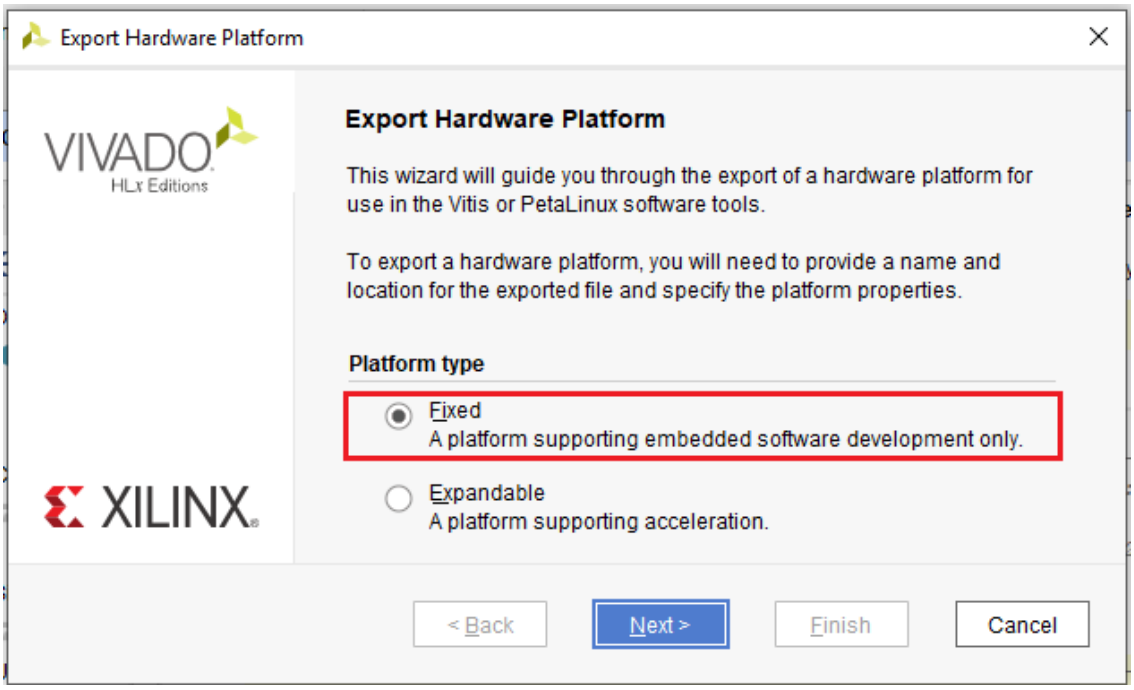


5. Export hardware after bitstream file is successfully generated.

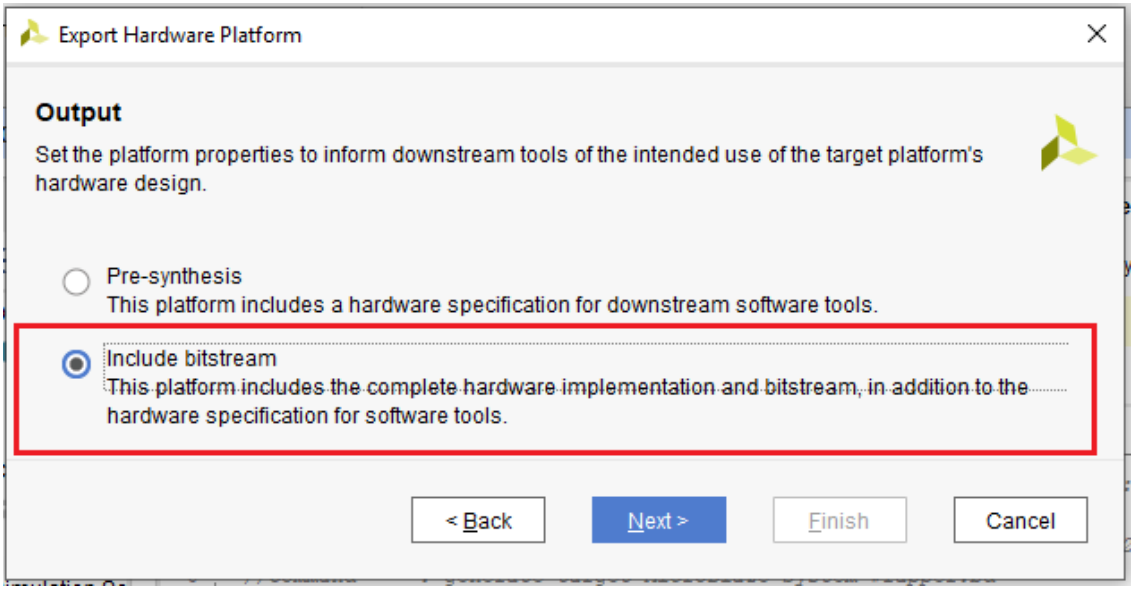
- Go to File, under Export, click **Export Hardware**



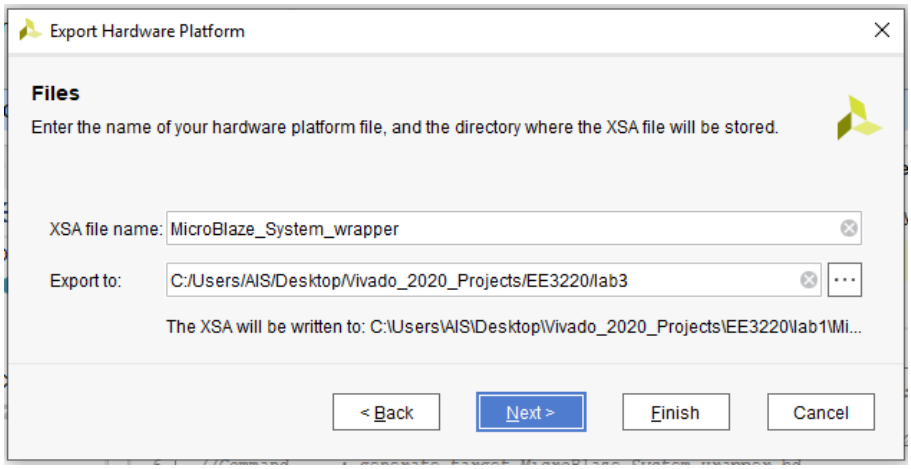
- Select **Fixed** option and click **Next**

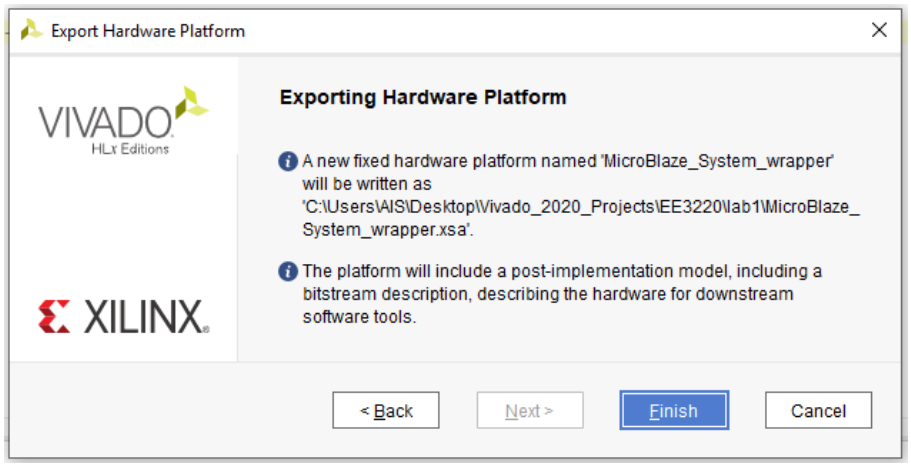


- Choose **Include bitstream** and click next



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





Check Point 2:

Take some snapshots as evidence of achieving the check point. Proceed to the next tasks

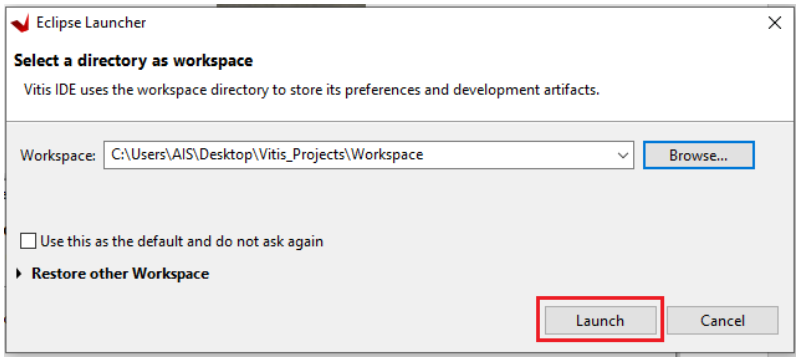
Part 2: Software Development using Vitis

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

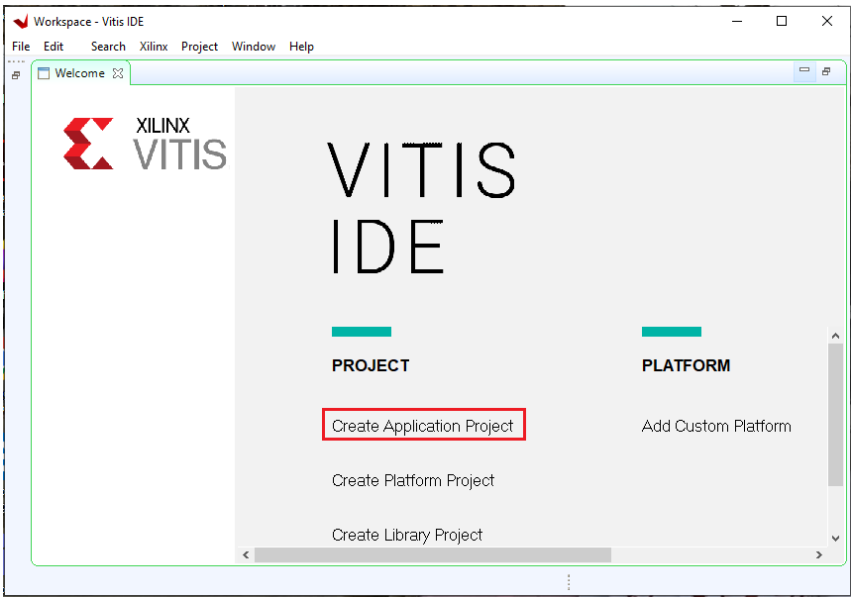
Follow the following steps to achieve this task:

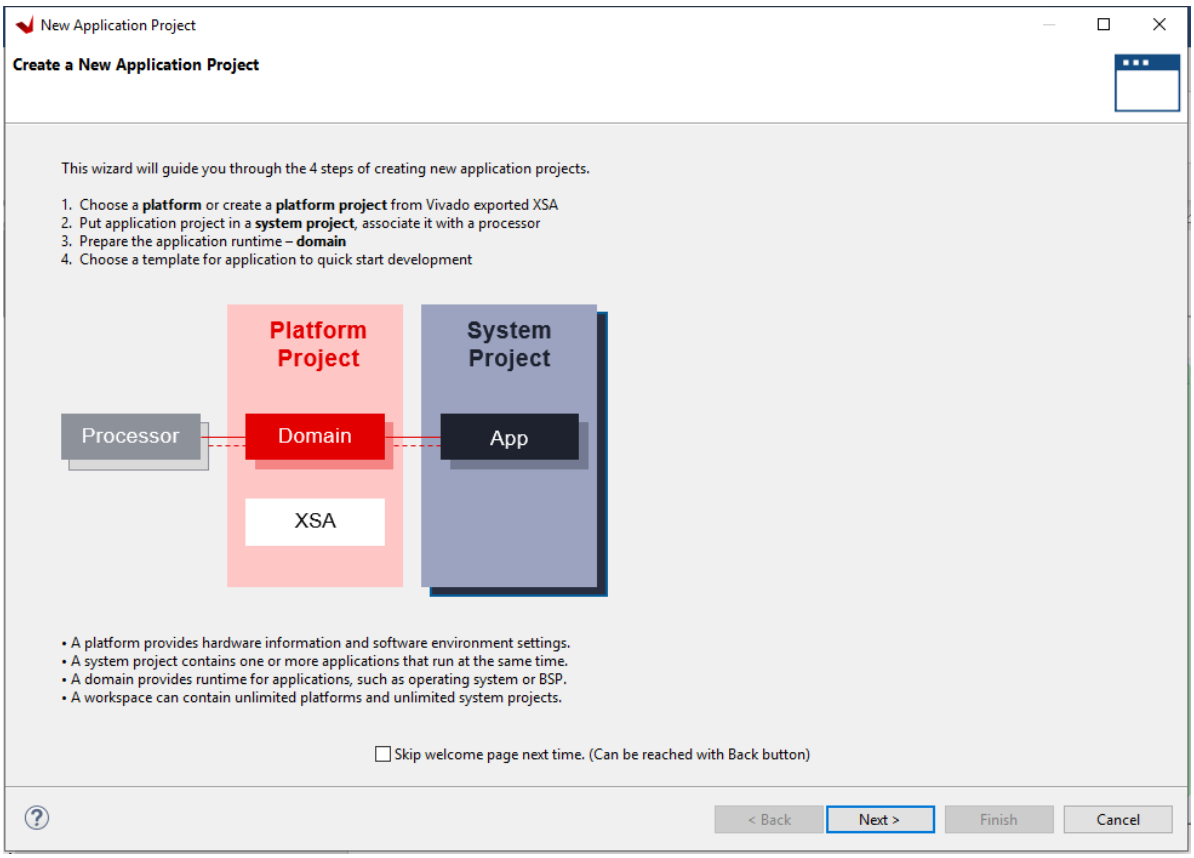
1. Create new Vitis application project

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

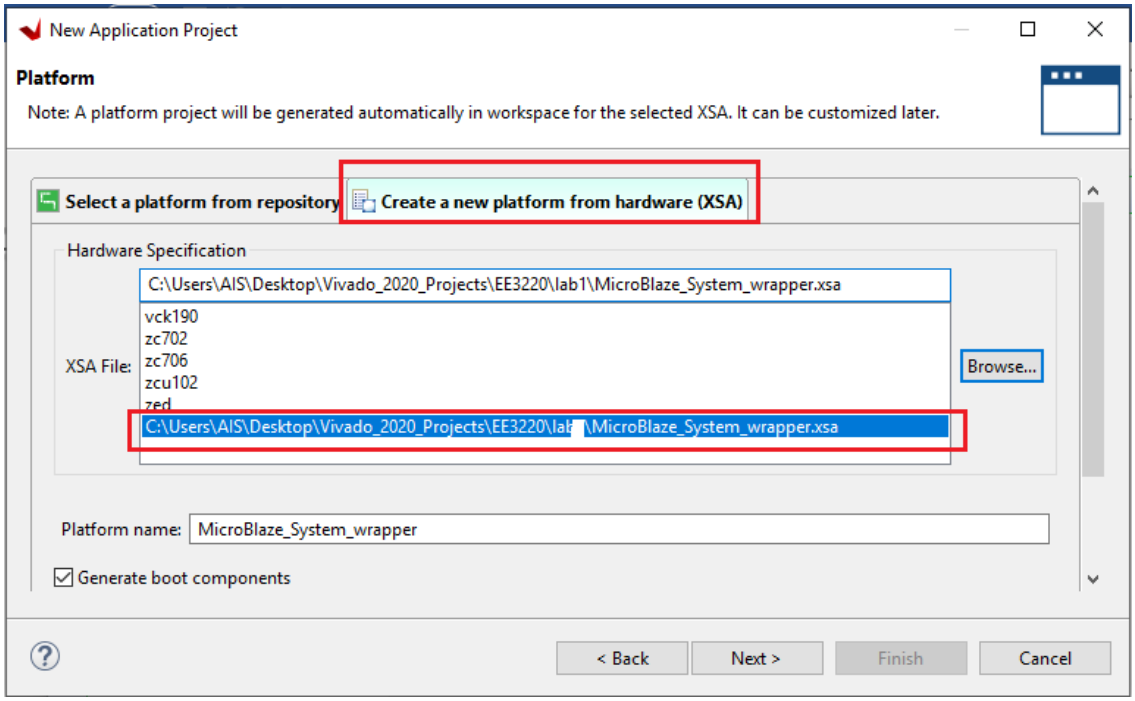


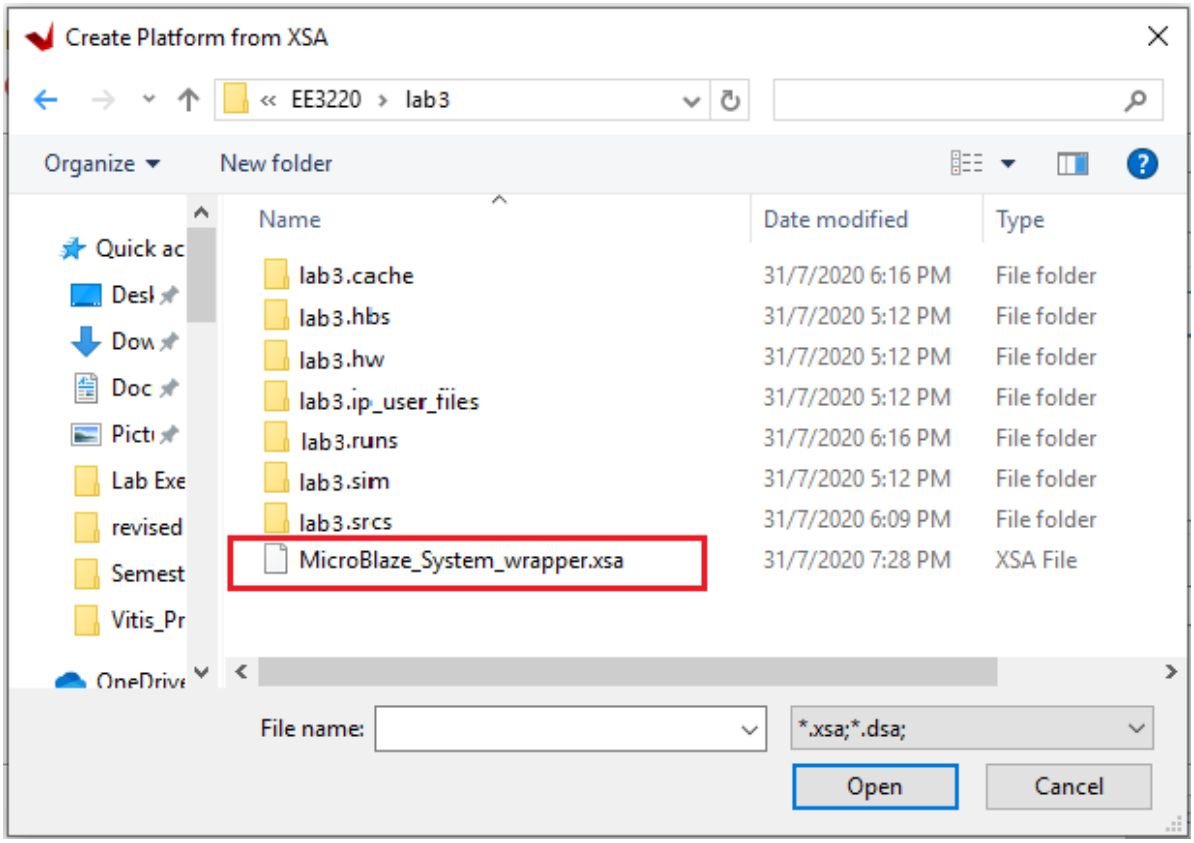
- Choose **Create application project** and click Next



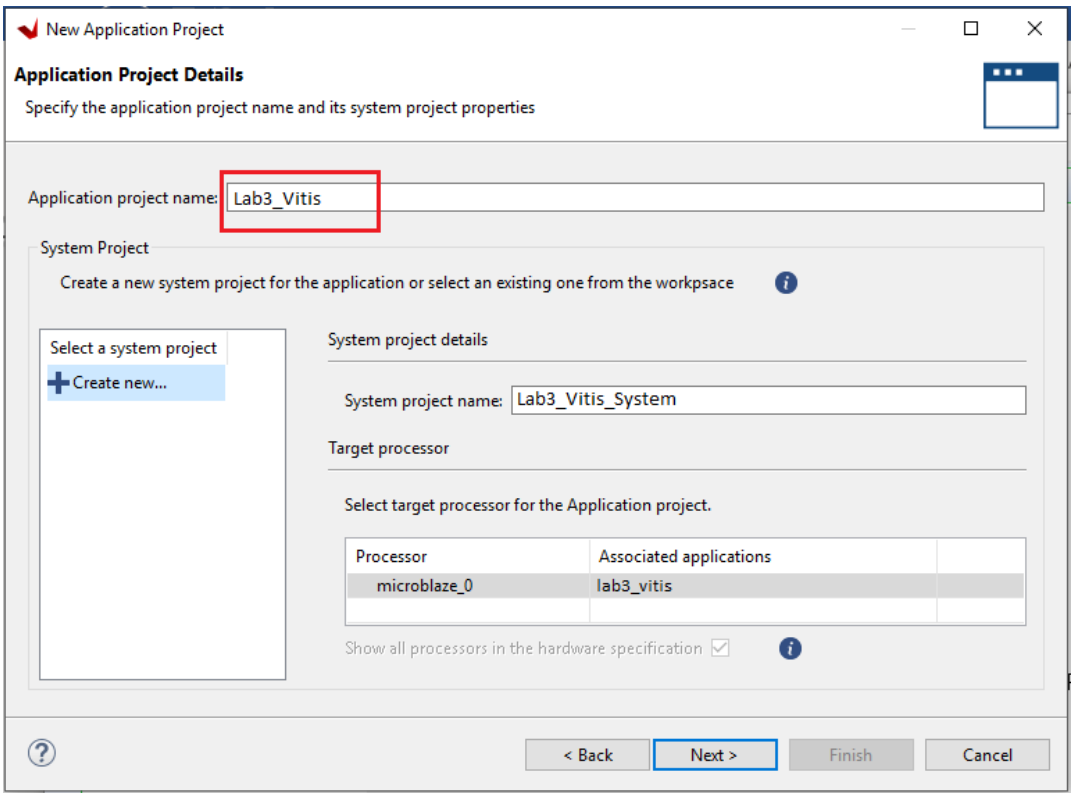


- Choose **Create a new platform from hardware(xsa)**, browse and go to the lab3 project folder and select the **MicroBlaze_System_wrapper.xsa**. Click Next and leave other setting as default, then, Finish.

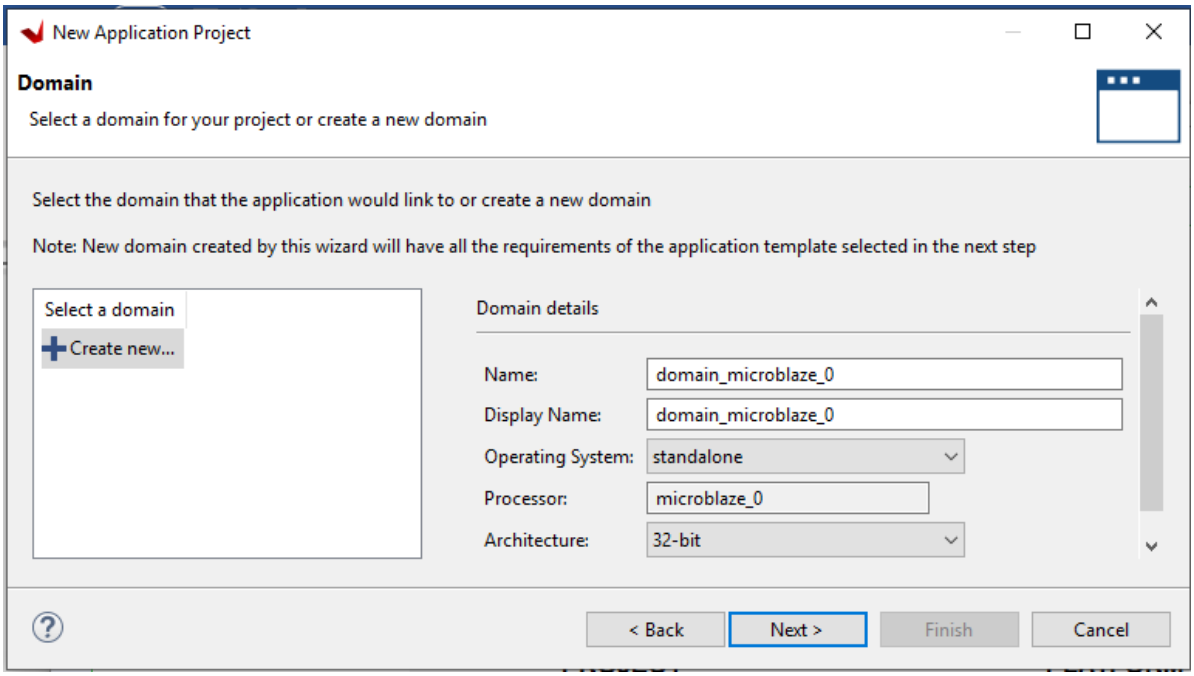




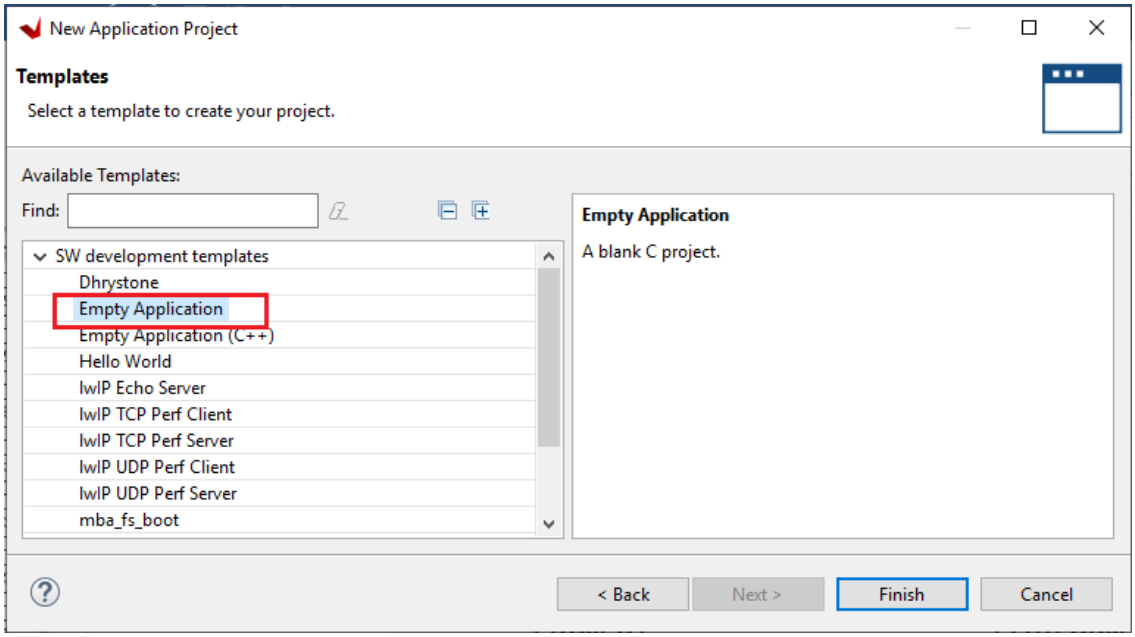
- Specify the project name (lab3_vitis), click **Next**.

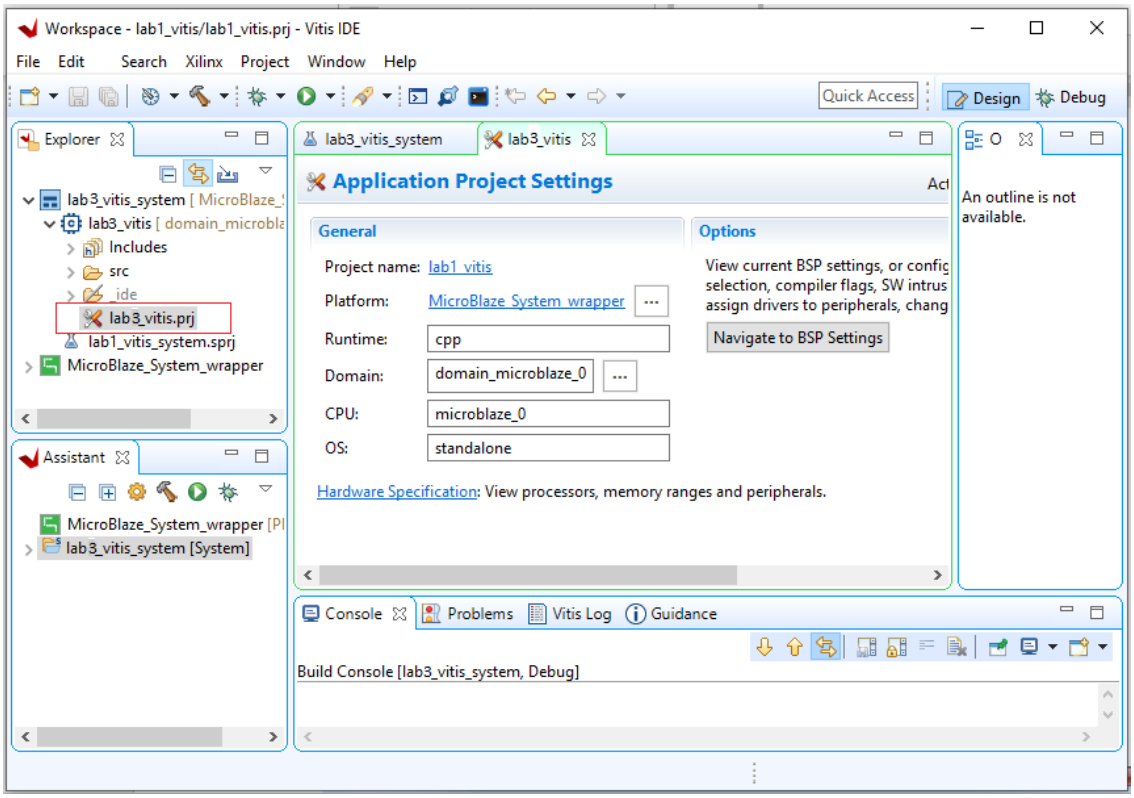


- Click **Next**



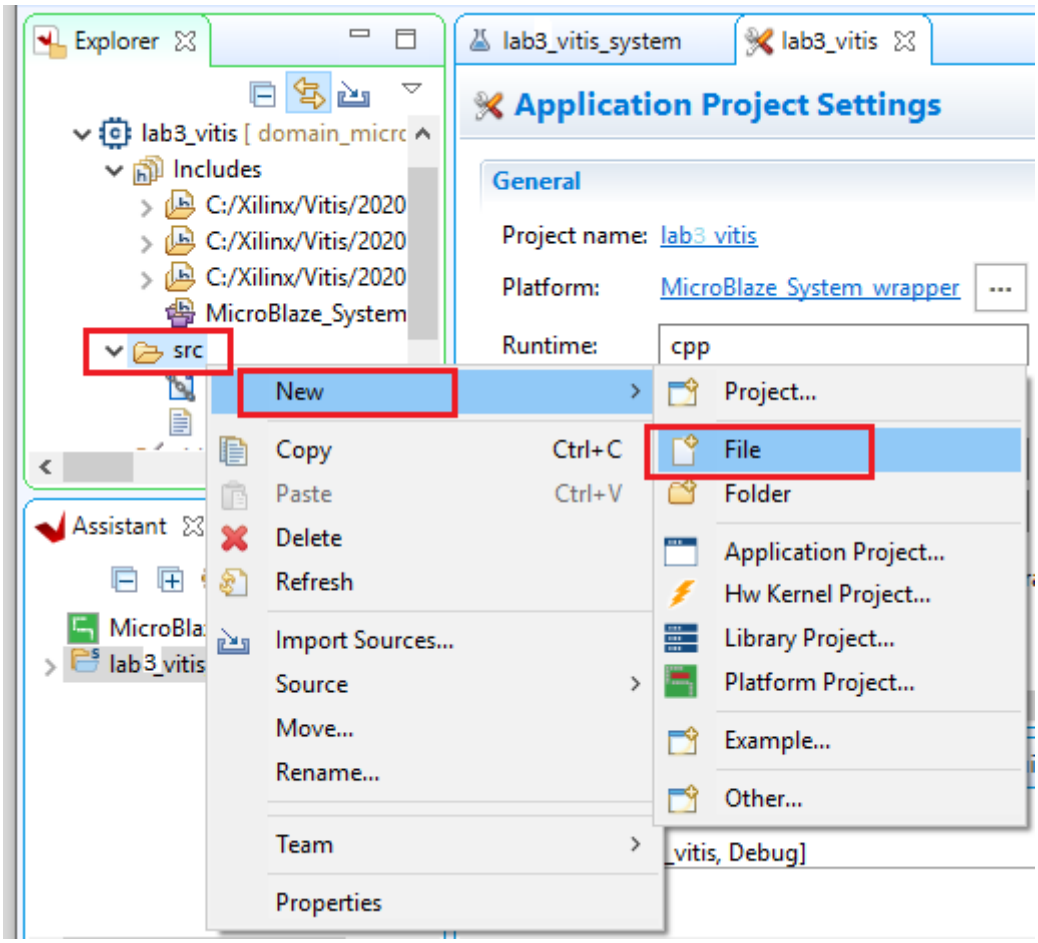
- Choose **Empty Application** and click Finish. Our new application project will be created



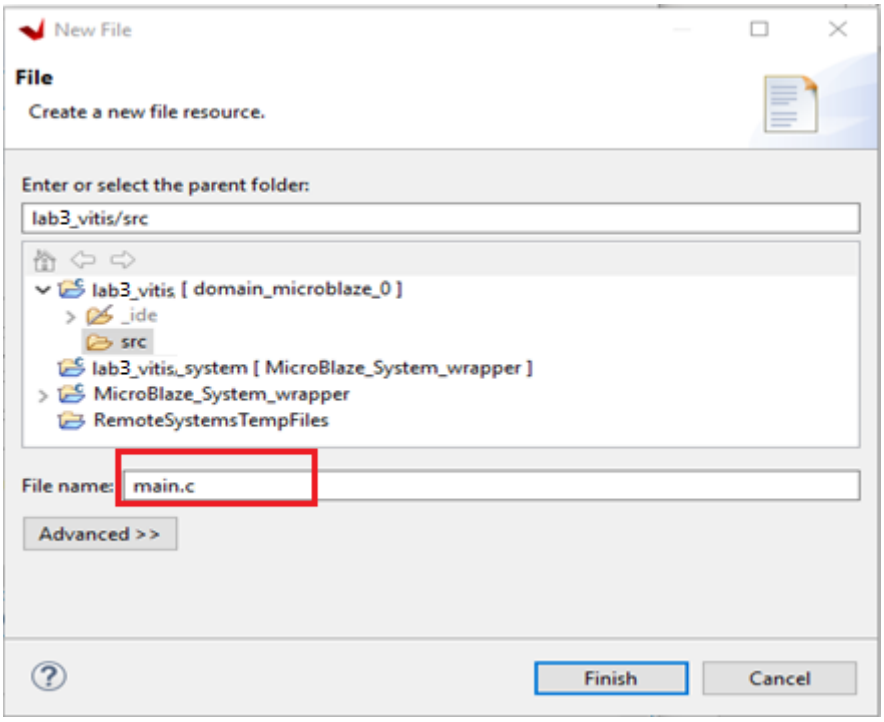


2. Create new source code file (main.c)

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

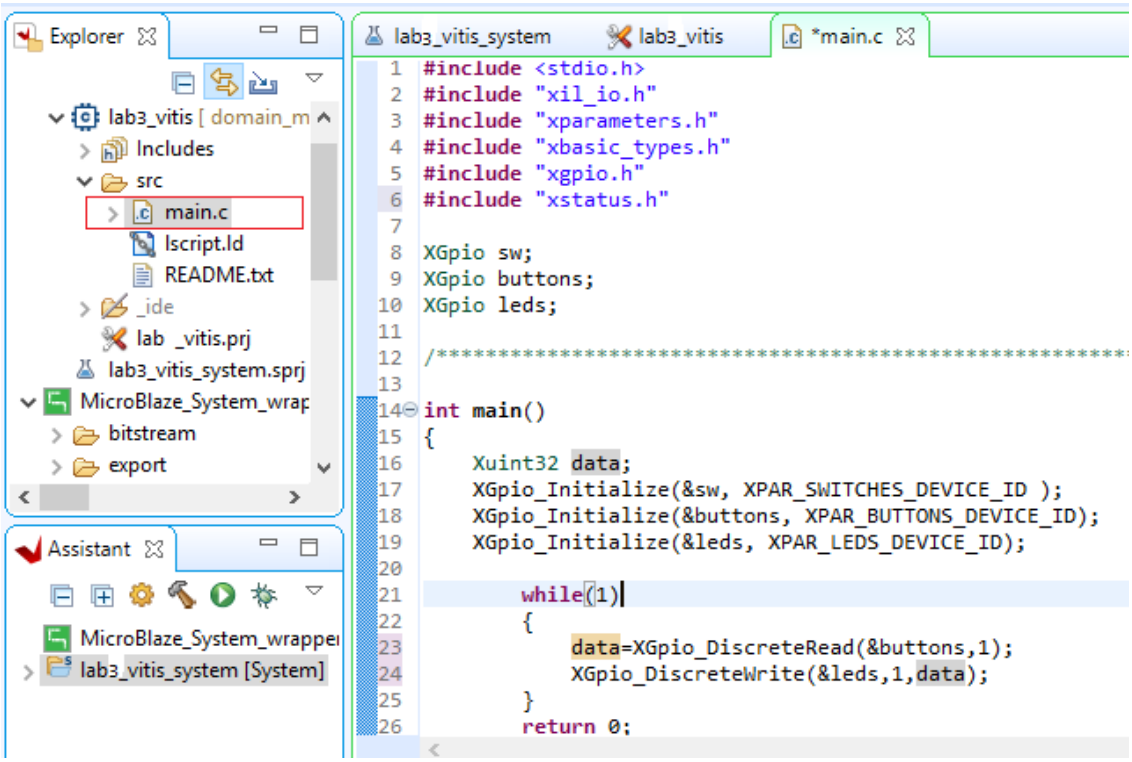


- Name the source file as **main.c**, click finish. A new source file will be created

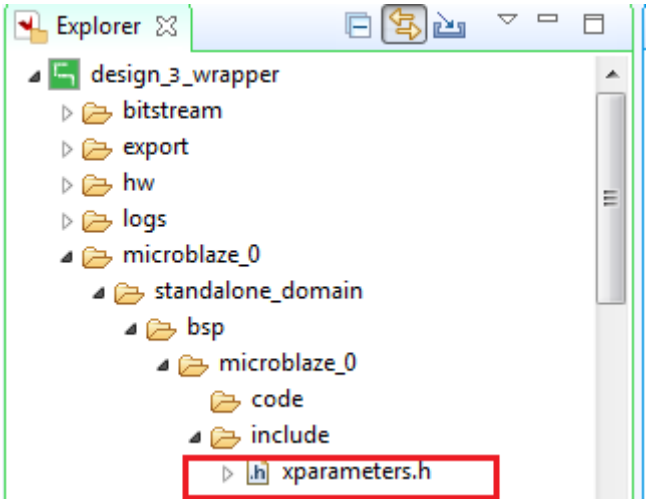


3. Write a program to control the LEDs, build project and program the FPGA.

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

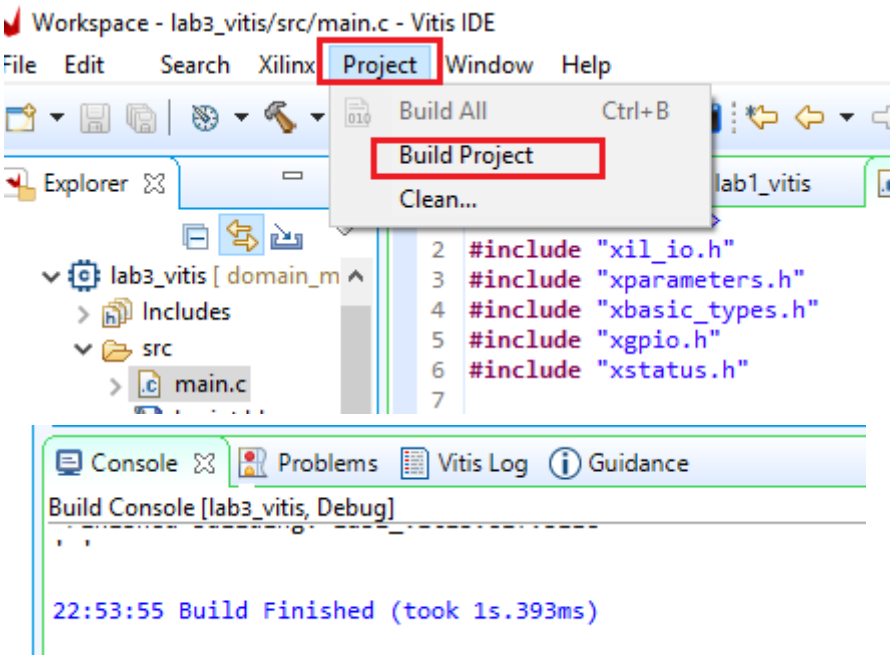


- To understand the definitions used for the GPIO peripherals, check the **xparameter.h**
- XGpio_DiscreteWrite() is used to write on the leds (GPIO output). The XGpio_DiscreteRead() could be used to read buttons or switches in case you need to do it in the future.

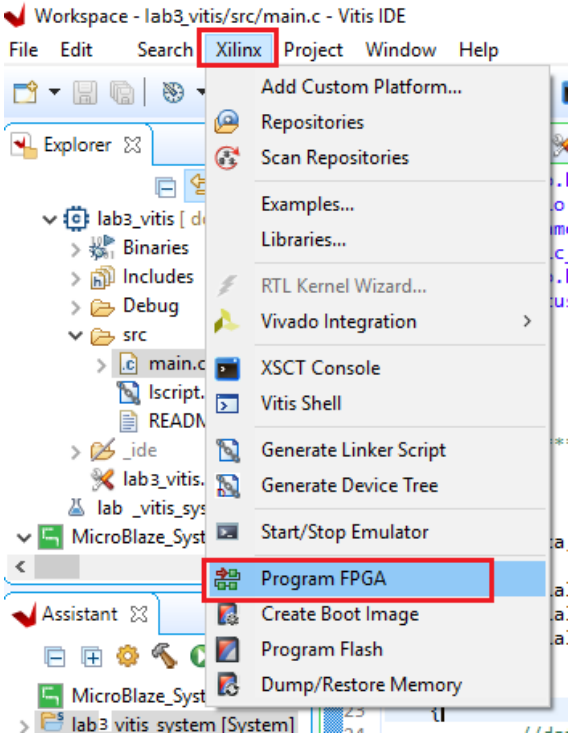



```
#include <stdio.h>
#include "xil_io.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "xstatus.h"
#include "sleep.h"

XGpio leds;
/*****************************************************************************/
int main()
{
     Xuint32 data;
     XGpio_Initialize(&leds, XPAR_LEDS_DEVICE_ID);

     while(1)
     {
       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.



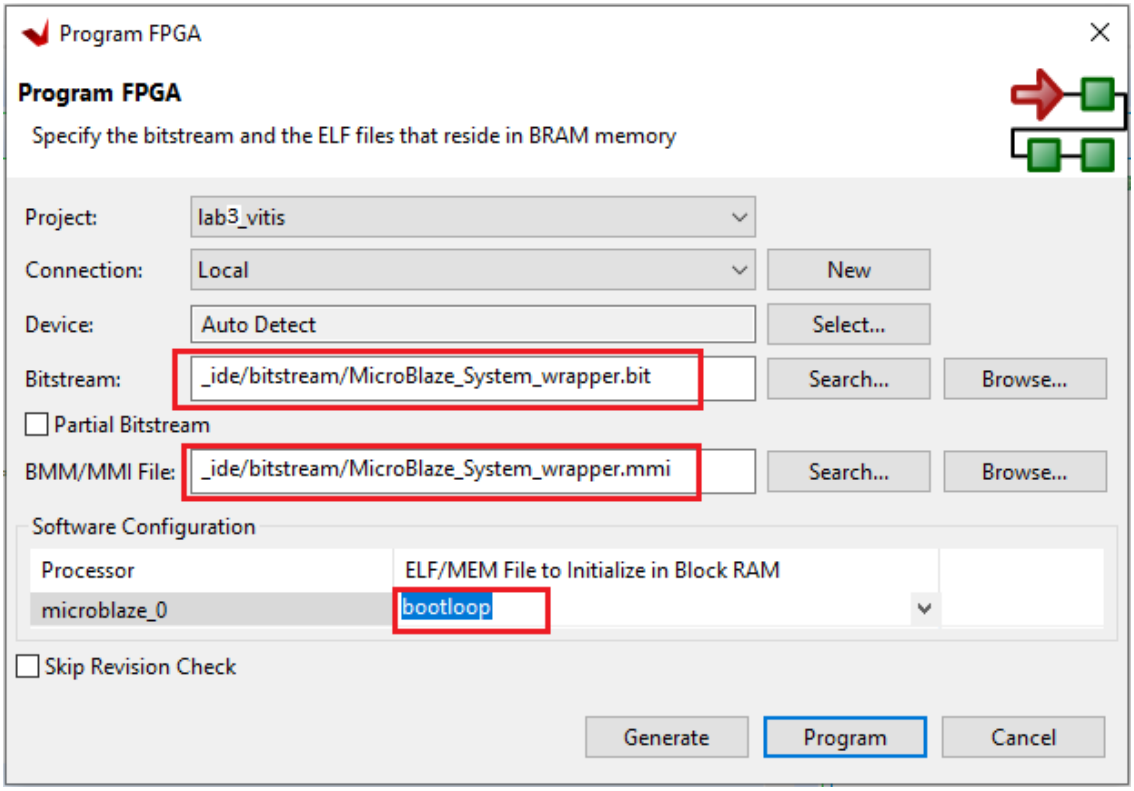
- Browse the project folder for the Bitstream file and the .mmi file. You may need to change the search file type to mmi to be able to see the .mmi file. if you use default location, the files are located at:

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

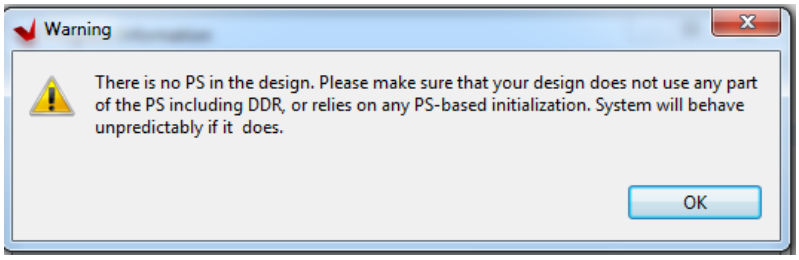
  .mmi file: < C:\Users\EEUsers\Workspace\lab3_vitis_ide\bitsream\microBlaze_System_wrapper.mmi >

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

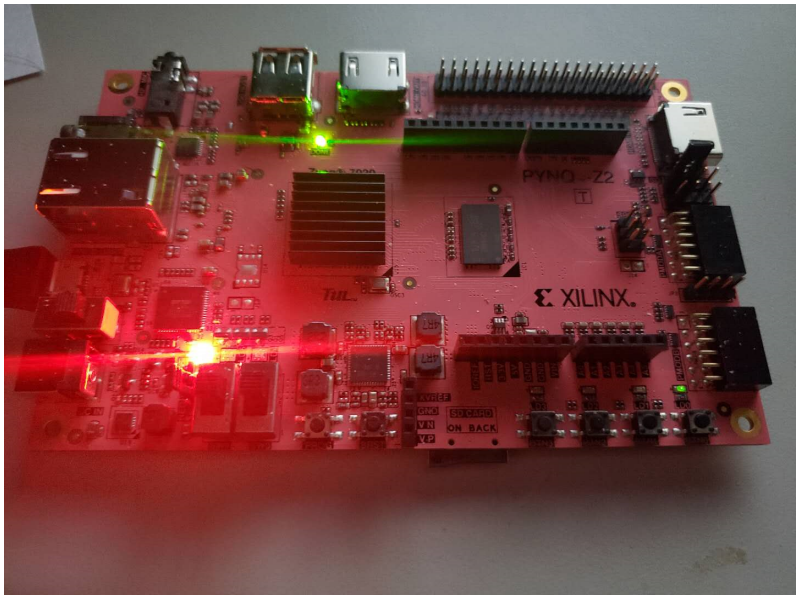
- For the elf/mem file, you should choose another your project's elf instead of the **bootloop** there, go to the project folder and select "<project folder> \Debug\lab3_vitis.elf".



- 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 2 and then 3 LEDs separately. Take snapshots for your lab report.

Check Point 4:

Modify the program to show a binary counting from 1 to 4 (**0001** to **0100**) 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 500ms delay or below i.e. **usleep(500000)**. The statement below gives you an example of a for loop.]

```
for( i=0; i <= 0x008FFFFF; i++) {;}
```

~ END ~