Lab 1: Zynq-SoC Design Flow Tutorial

Description

- This Lab serves as the back-bone of all the steps/processes that you will need through out the semester. Please make sure to do it fully and completely understand it.
- Through out the following Labs, specific parts of this Lab might be mentioned for your reference.
- In this Lab, you will learn about:
  - Importing user-created IP Repository
  - Creating your own IP Packages from VHDL designs that you write
  - Adding AXI hardware to your design to utilize software testing
  - Testing your hardware system using C testing code and VHDL testbench (as in the Pre-Lab)
  - Connecting a serial terminal to the PYNQ-Z1 board for communication.
  - Using Integrated Logic Analyzer (ILA) to probe different system signals during runtime.

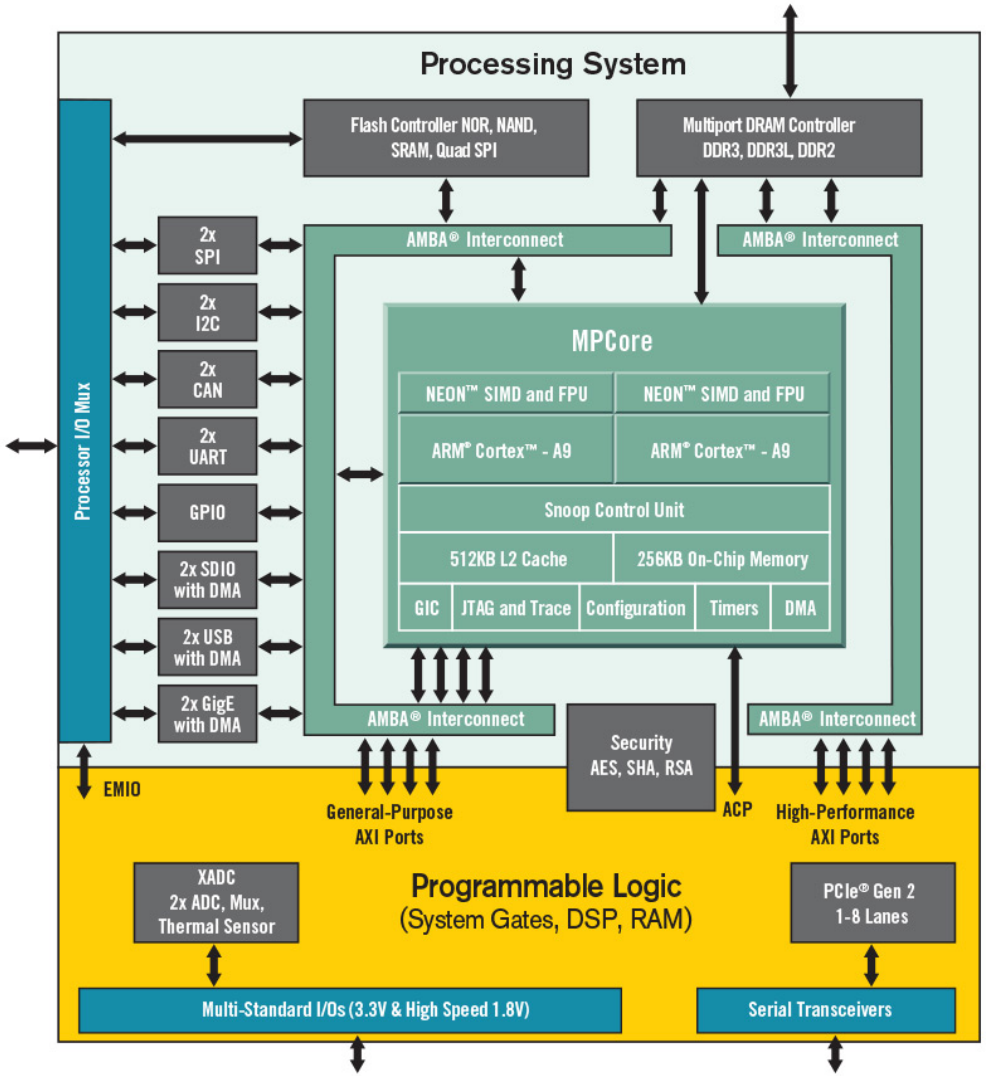
Xilinx Zynq SoC

Documentation Resources

- Zynq SoC [TRM - UG585]
- Vivado
  - Synthesis [UG901]
  - Constraints [UG903]
  - Implementation [UG904]
  - Simulation [UG900]
  - ILA [PG172]
  - Libraries [UG953]
- Embedded System Tools [UG1043]
- Petalinux [UG1144]

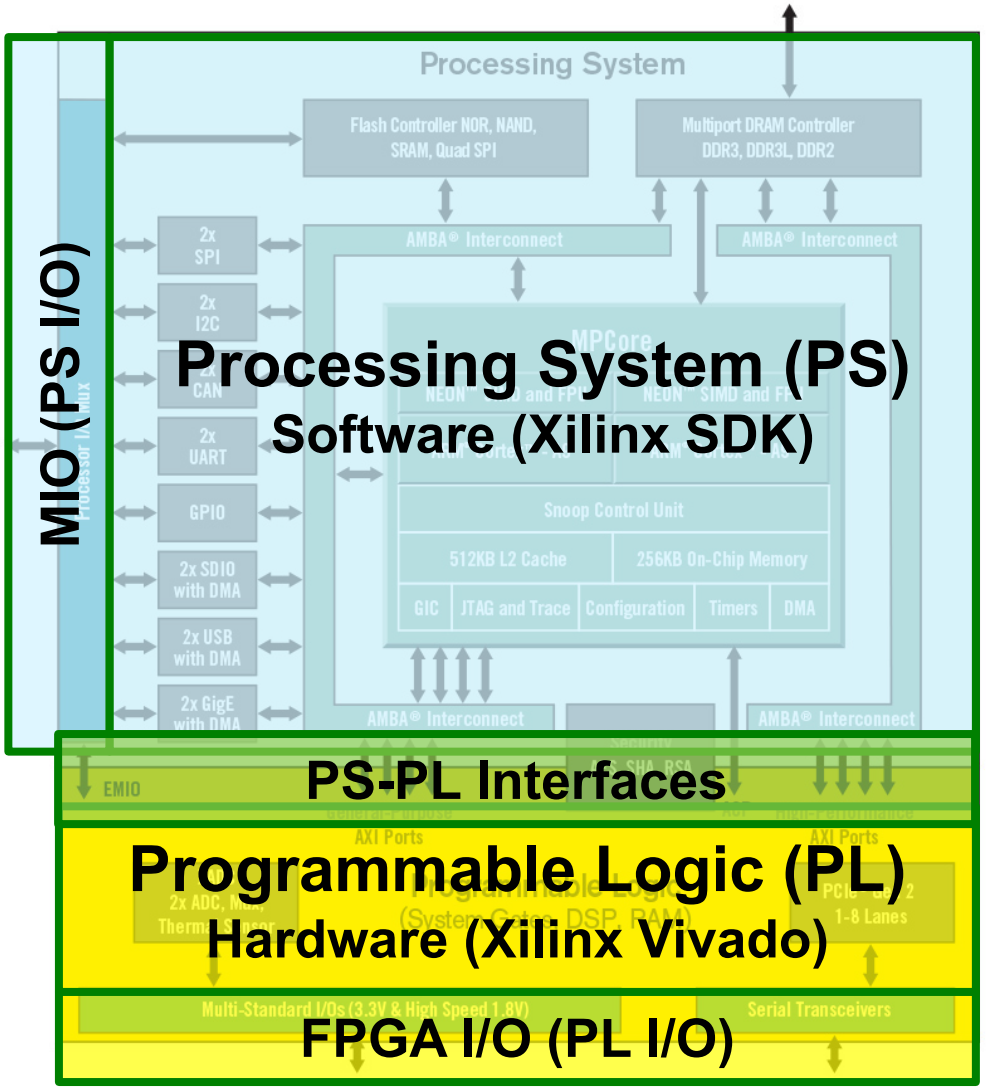
Note: These are just for reference. No need to look any of them now!

Zynq-7000 SoC Architecture Overview (1/2)



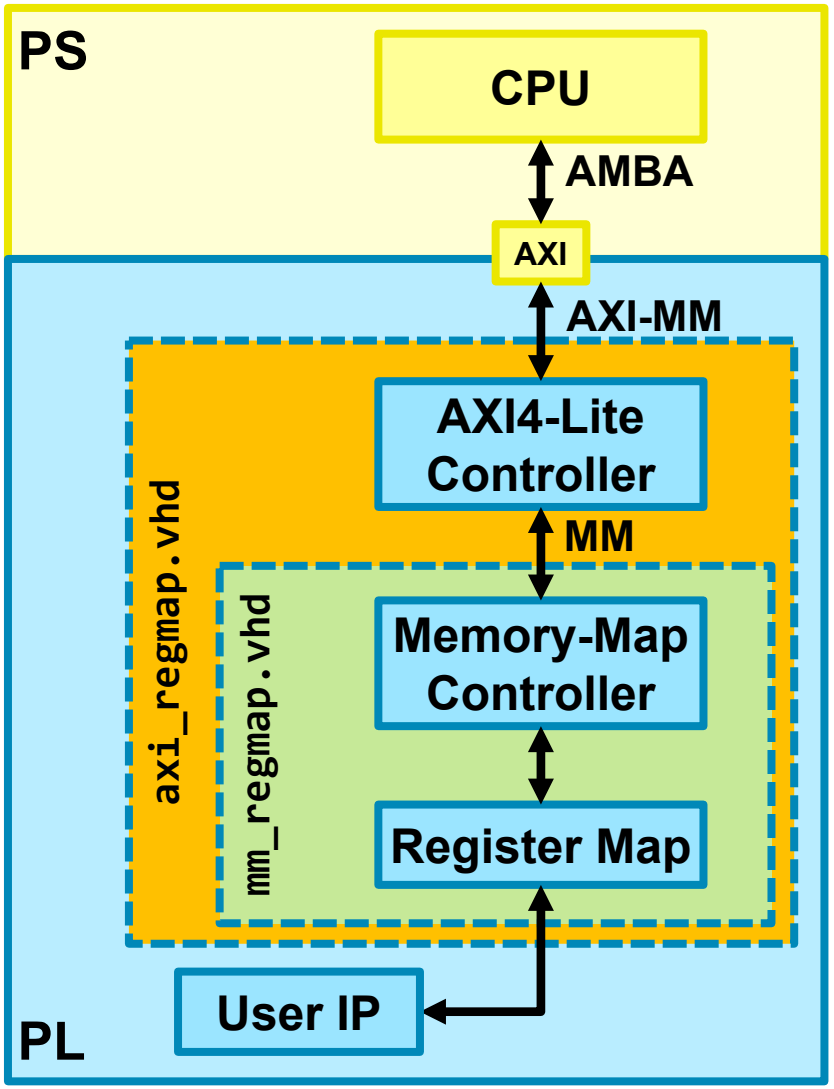
Zynq-7000 SoC Architecture Overview (2/2)

- Processing System (PS)
  - Contains general-purpose AXI ports with address space
  - Handles C programs to test Hardware (PL) Design
- Programmable Logic (PL)
  - FPGA Hardware
  - This is where the Configure Logic Blocks (CLBs) are



Memory-Mapped IP Cores

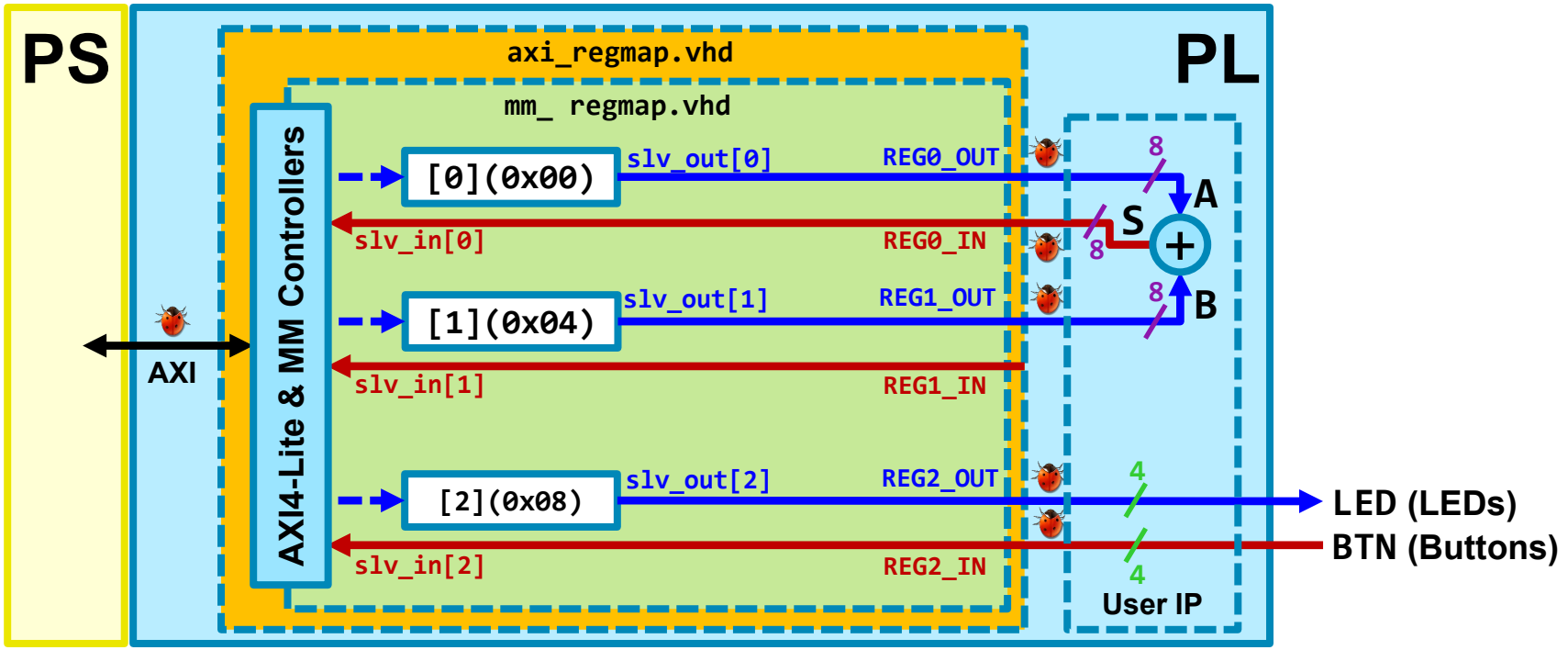
- PS
  - Contains general-purpose AXI ports with address space
  - CPU issues read/writes through AXI ports to peripherals in FPGA
- PL
  - Attach AXI-compatible IP core to AXI port to interface with CPU
  - AXI and memory-map controllers convert AXI transactions into register reads/writes
  - `regmap` IP enables users to connect custom IP with register map for general-purpose reads/writes



Example Design

Integrated Logic Analyzer (ILA)
Signals that will be probed

Register Map			
Index	**Offset**	**Read Access**	**Write Access**
[0]	0x00	S (adder output)	A (adder input)
[1]	0x04	Zero	B (adder input)
[2]	0x08	BTN	LED



Xilinx Vivado Tutorial

Part 1: Creating a new Vivado Project

Part 1: Create a new Vivado Project

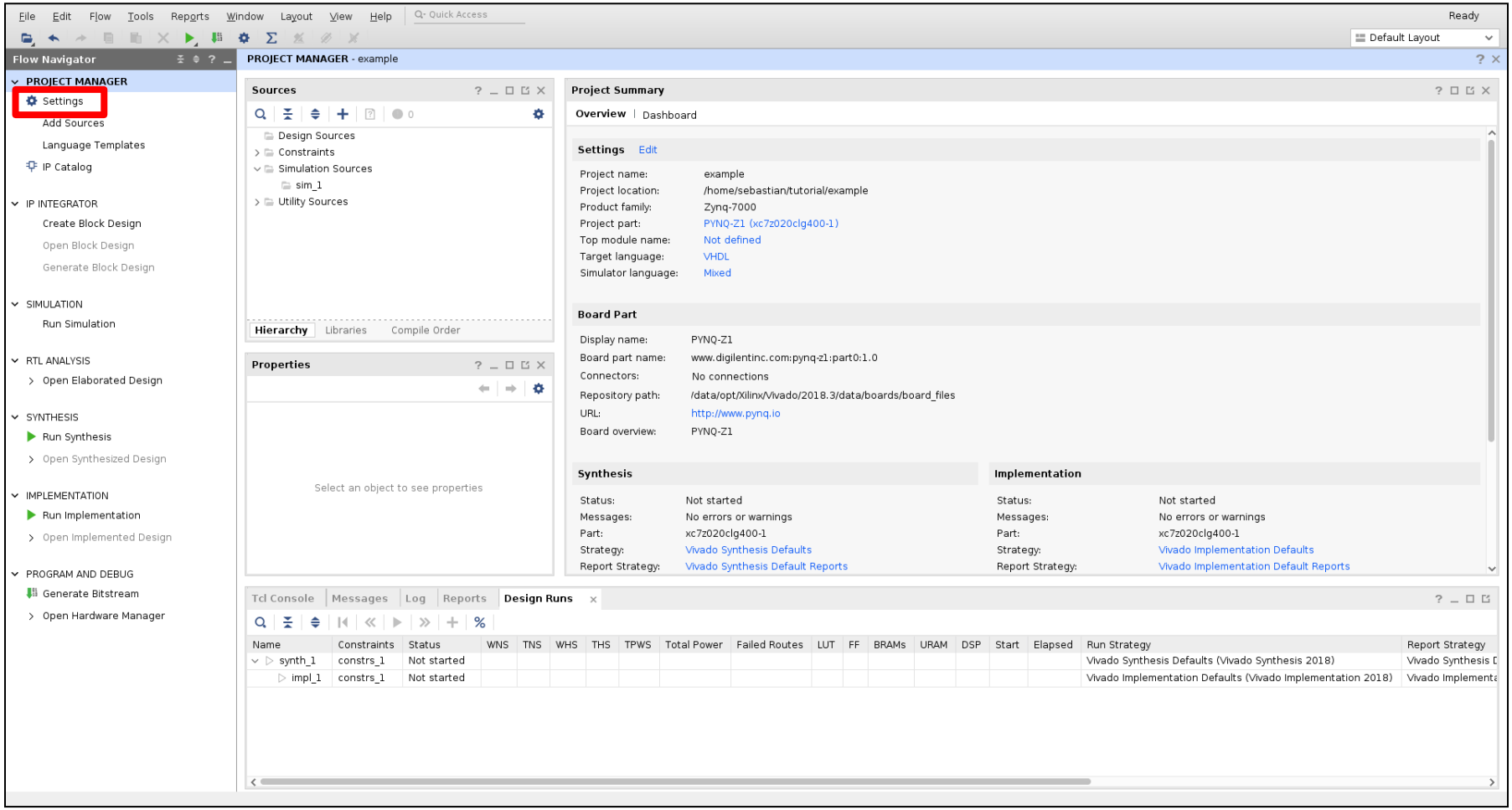
- Create a new project as per the Pre-Lab
  - The project should be named “Lab_1”.
    Note: Neither the project name, nor the project path should have spaces!
  - Project Type: RTL
  - Target Language: VHDL
  - Constraints File: “PYNQ-Z1_C.xdc” (from Pre-Lab files)
  - Boards: PYNQ-Z1

Note: These are just for reference. No need to look any of them now!

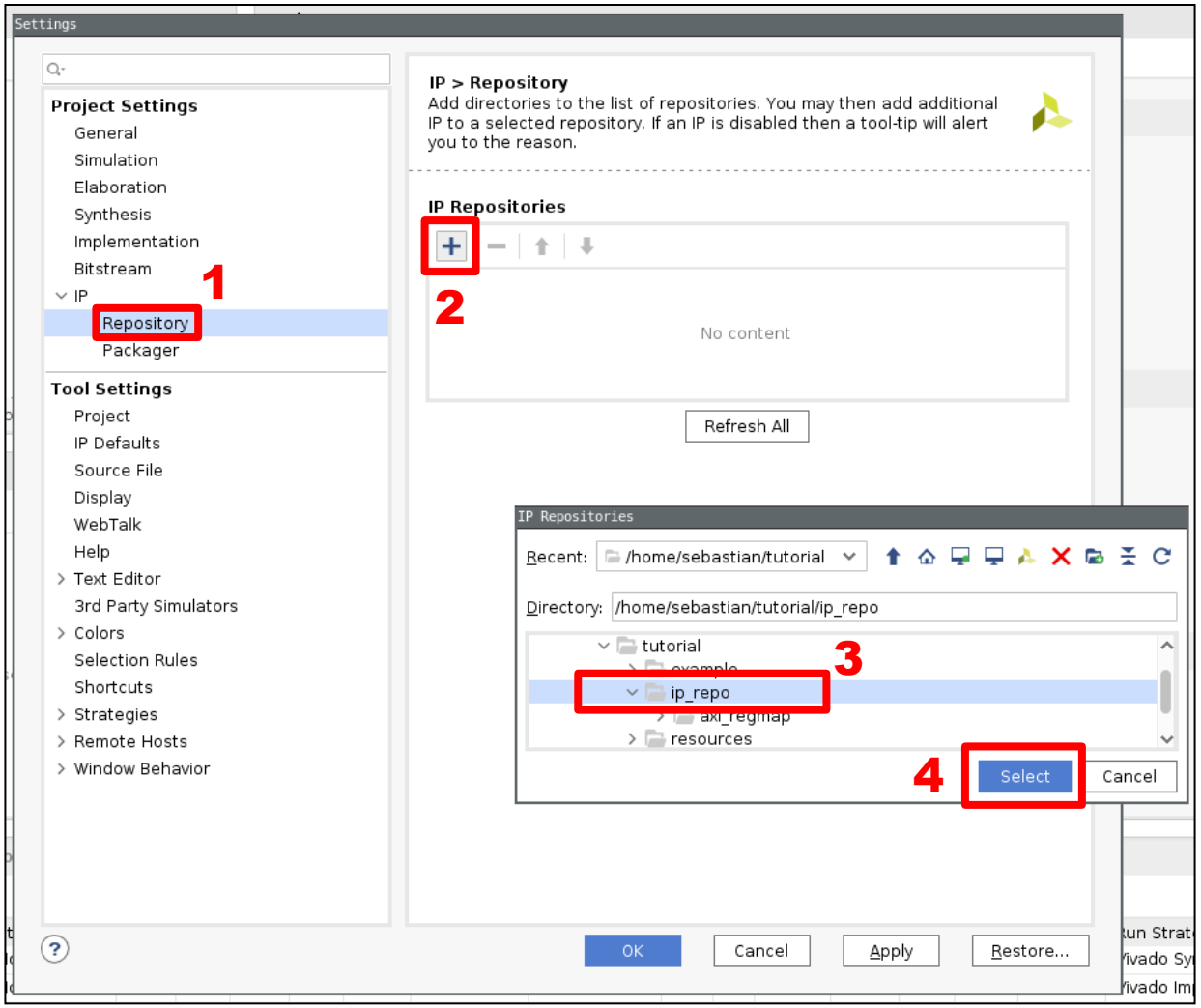
Part 2: Adding an IP Repository

An IP repository contains all packaged IP that can be used for system integration. We will add an IP repository that already contains the **axi_regmap** IP.

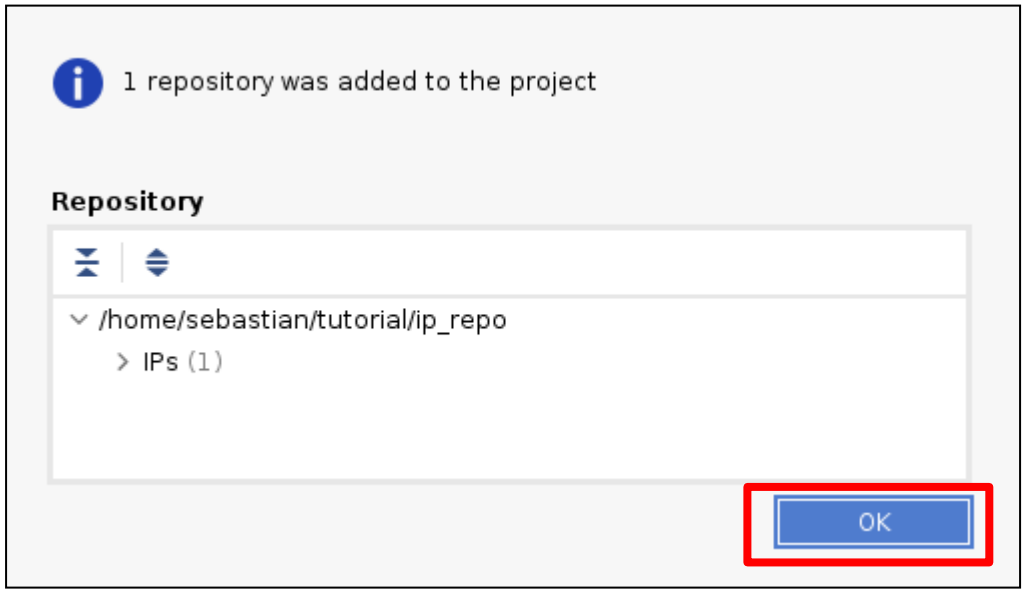
Click ***Settings***.



Click ***Repository***. Click ***+ (Add)***. Navigate to **`Lab_1_repo/ip_repo`**. Click ***Select***.

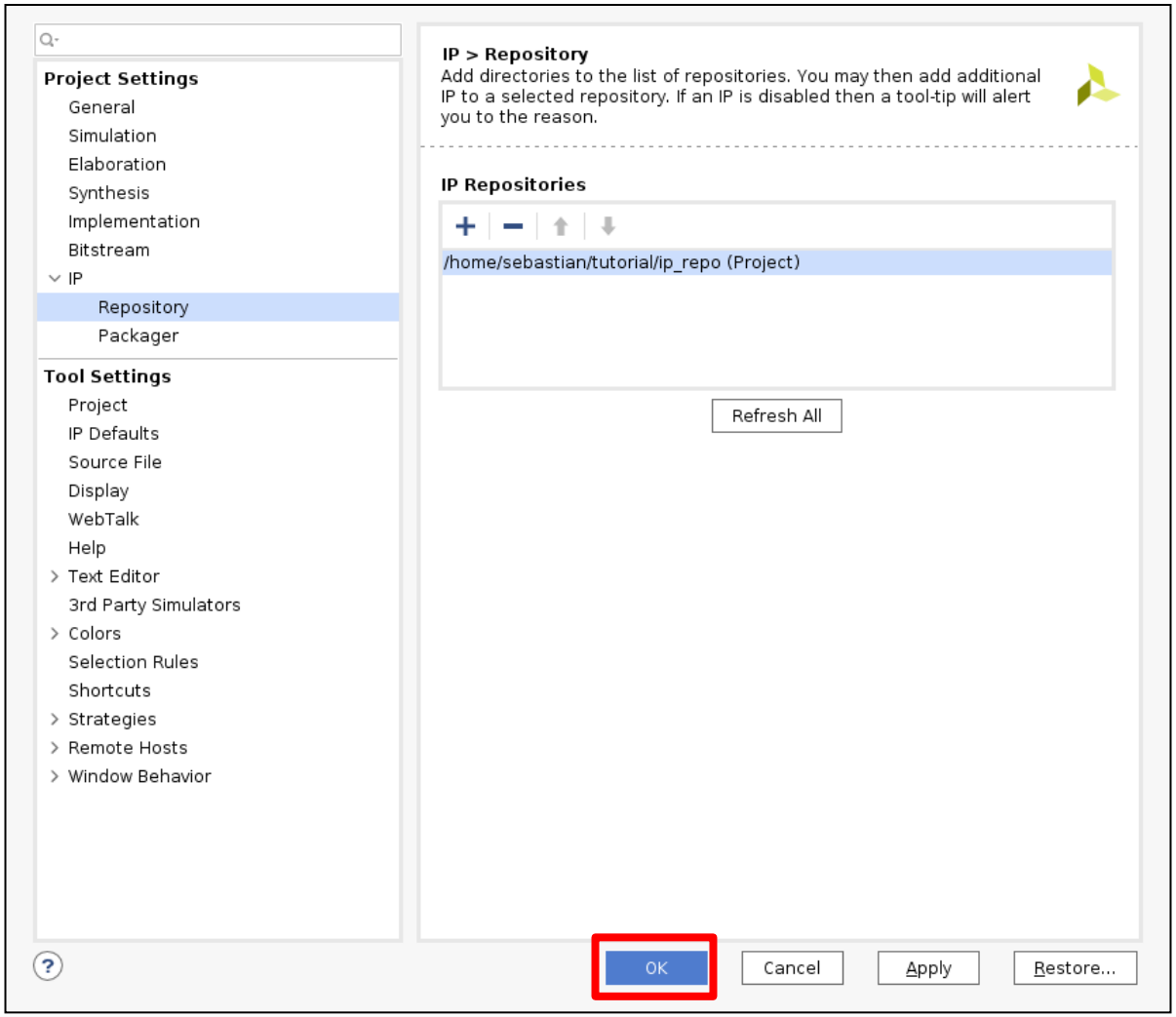


Click ***OK***.



.

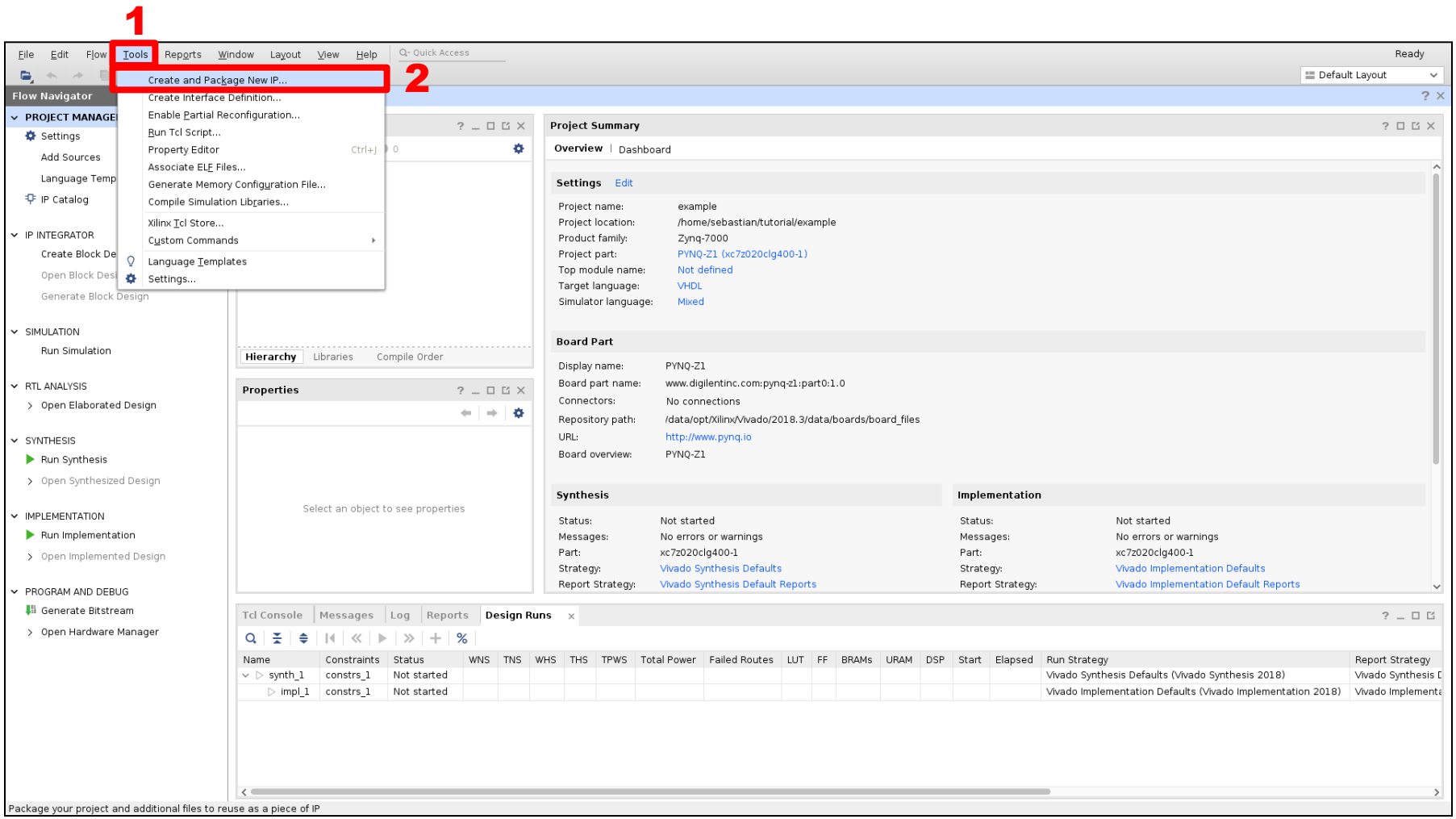
Click ***OK***.



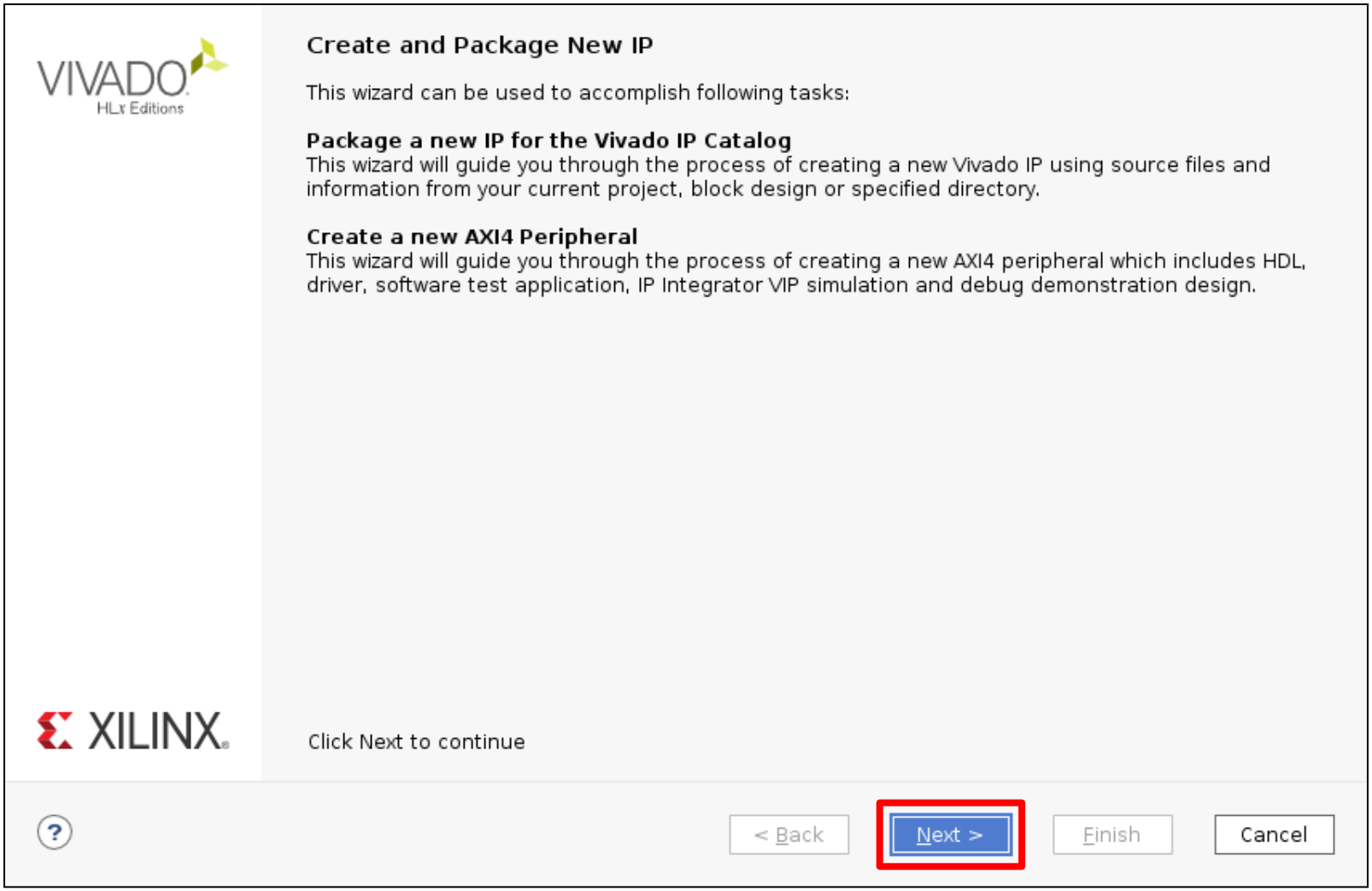
Part 3: Creating a new IP

Next, we will create an **adder** IP and package it using the Vivado IP Packager.

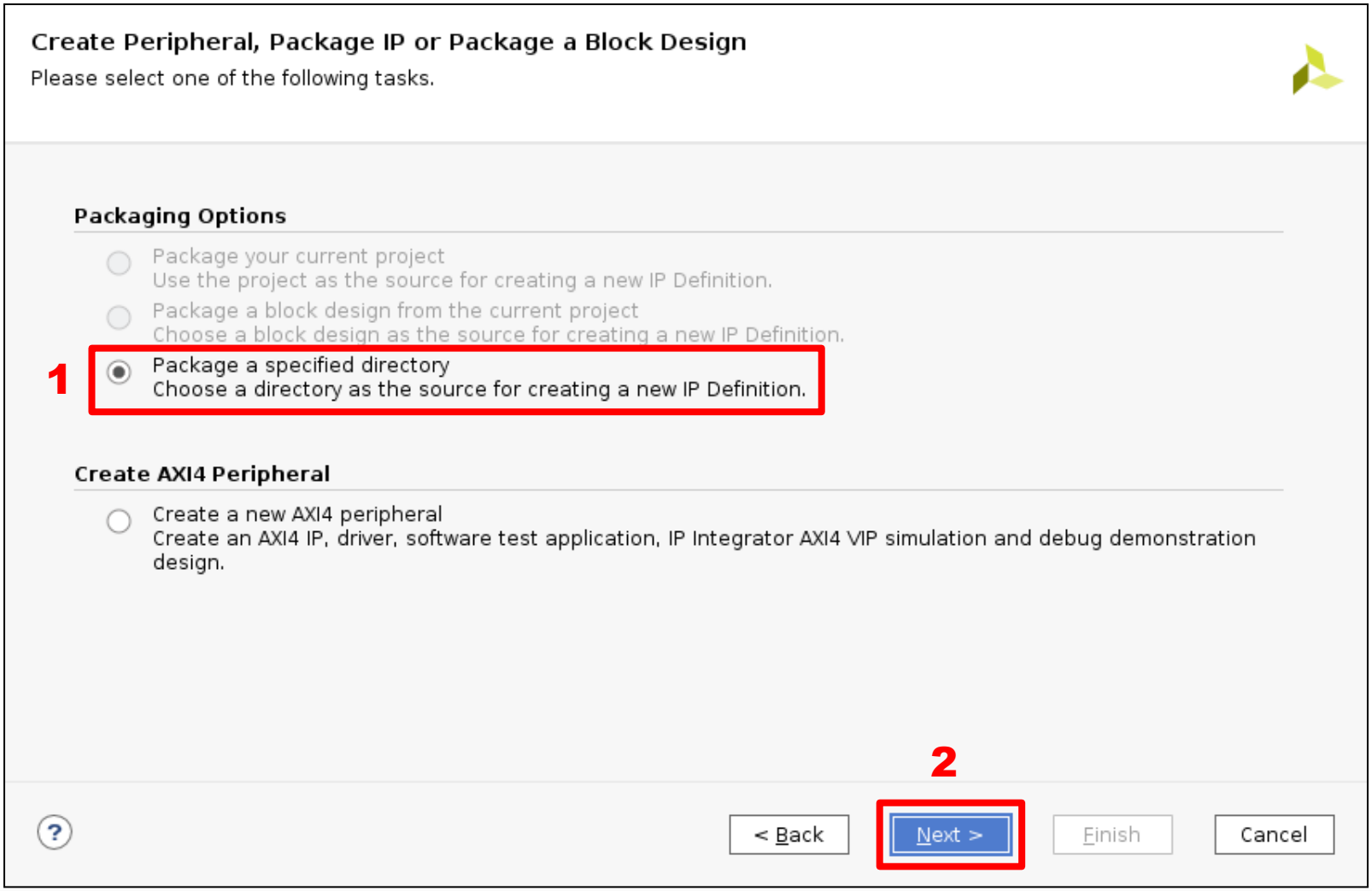
Click ***Tools***. Click ***Create and Package New IP…***.



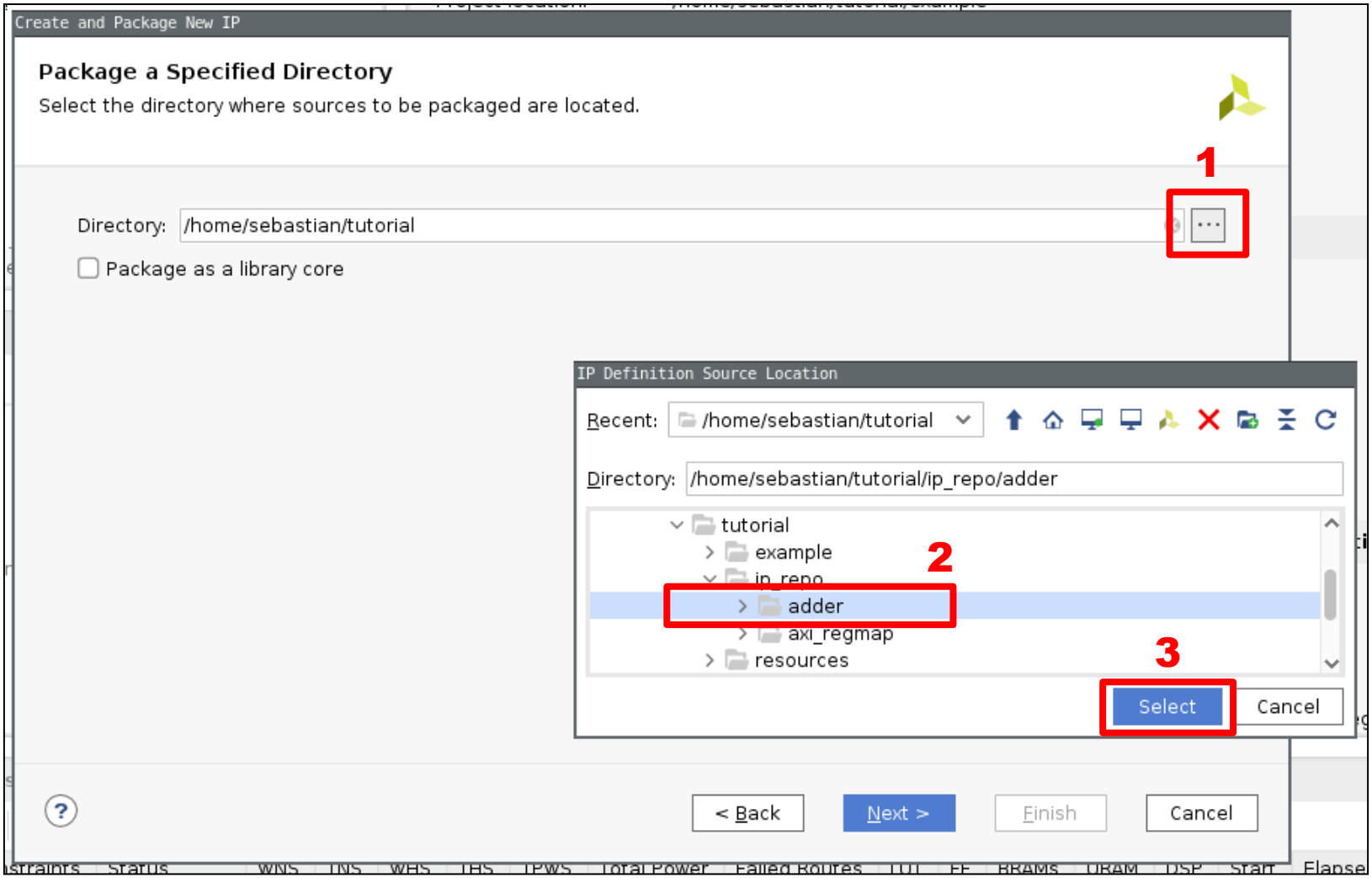
Click ***Next***.



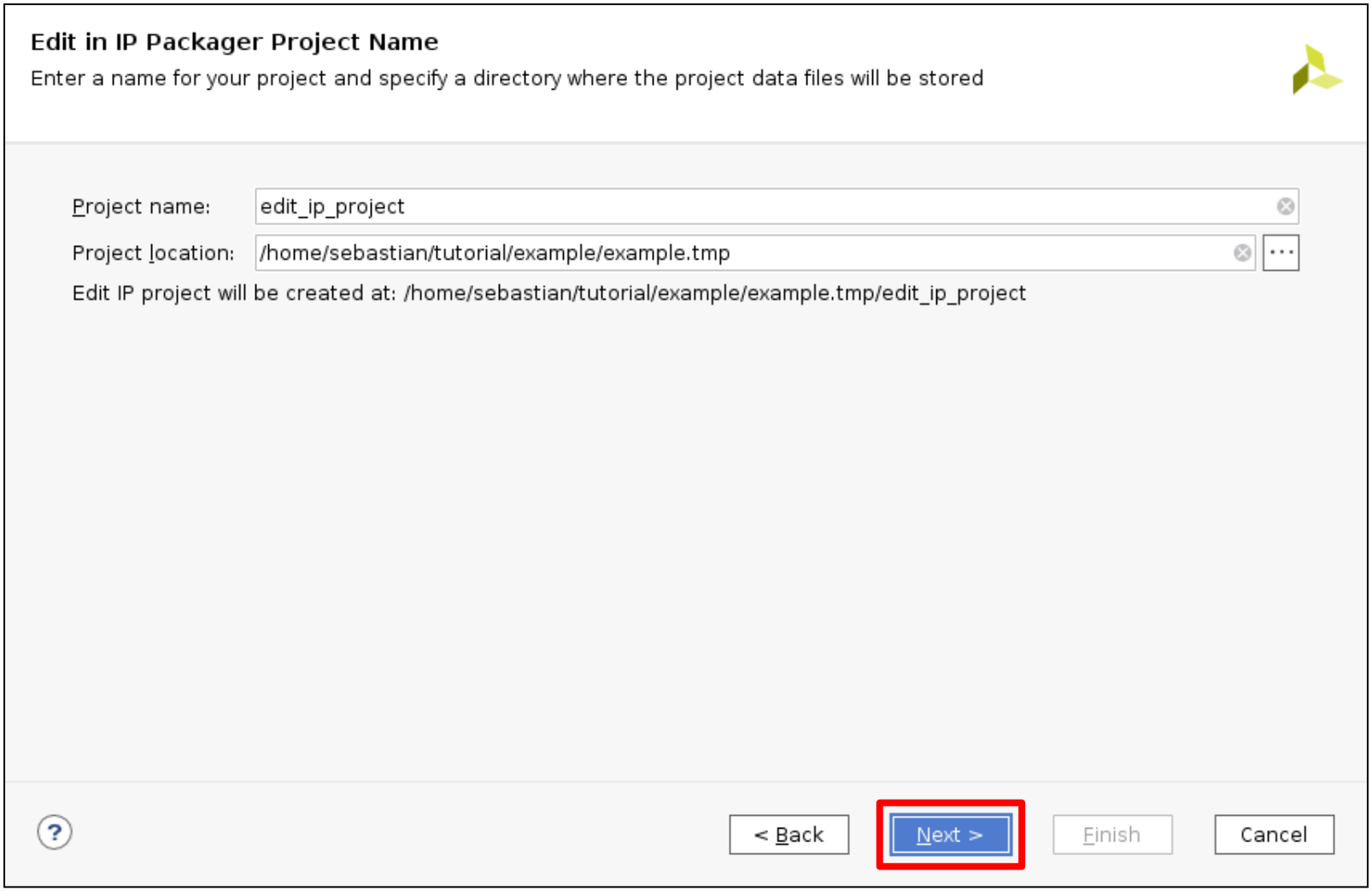
Select ***Package a specified directory***. Click ***Next***.



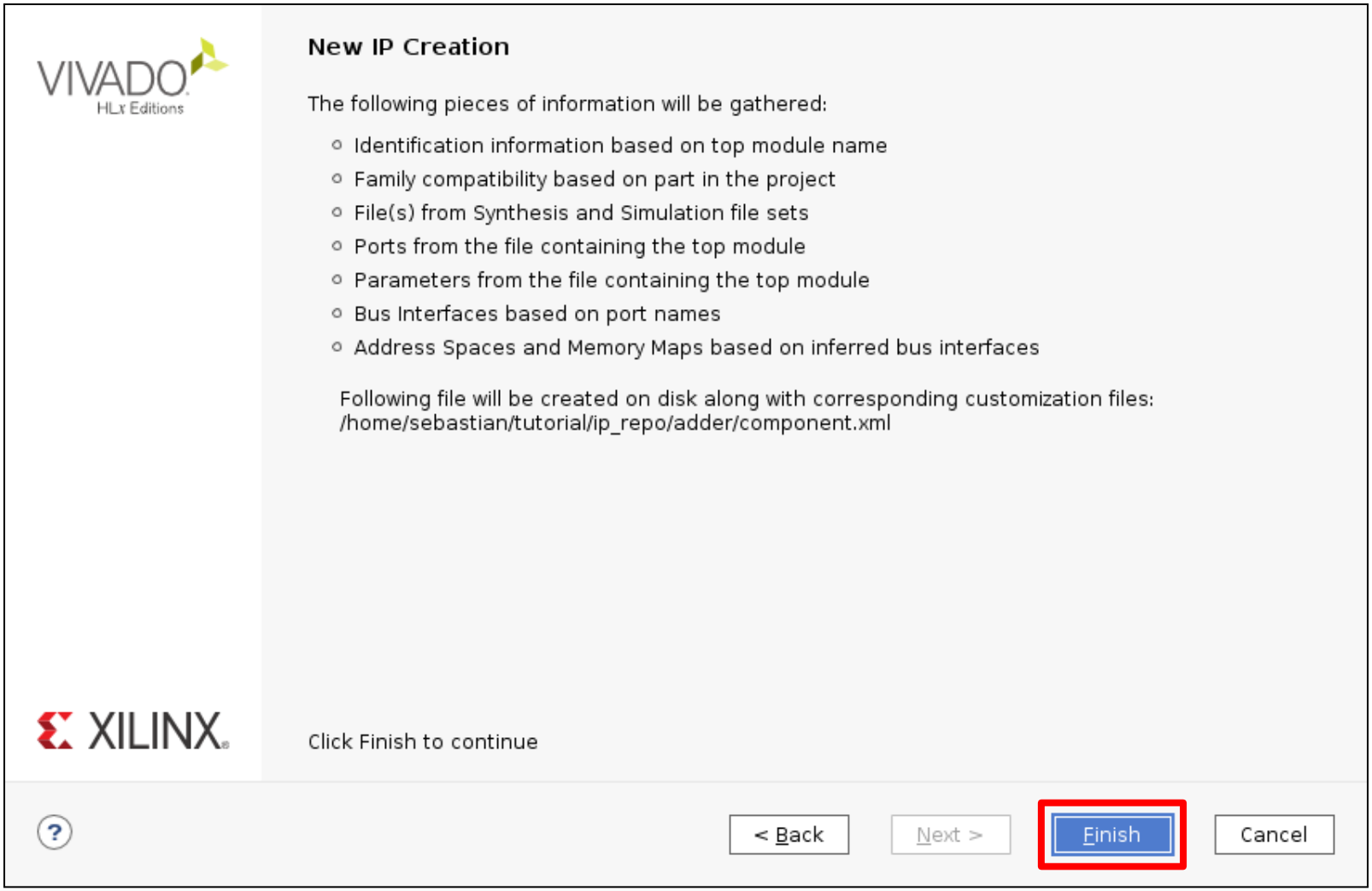
Click …. Navigate to **`Lab_1_repo/resources/adder`**. Click ***Select***.
Note: The above directory is different from the one in the picture.



Click ***Next***.

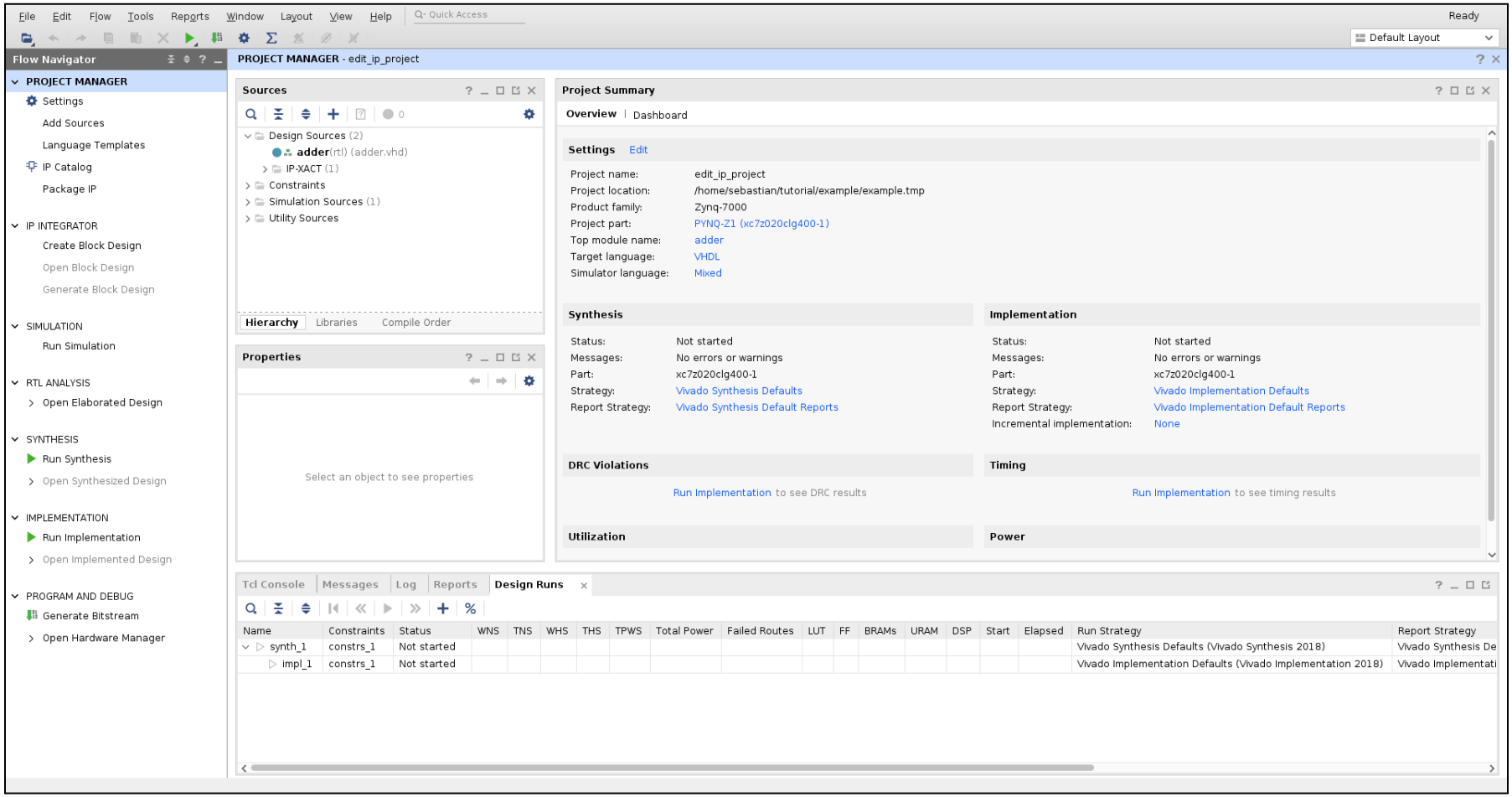


Click ***Finish***.



A new Vivado instance is created specifically for the **adder** IP. Use this instance to develop and package the **adder** IP. Continue.

Note: for future projects, you might want to leave this instance opened in case you need to edit and re-package your design.



adder.vhd. This RTL adds two inputs (**A** and **B**) represented as vectors of width **WIDTH** and outputs the sum (**S**) represented as a vector of width **WIDTH**. Continue.

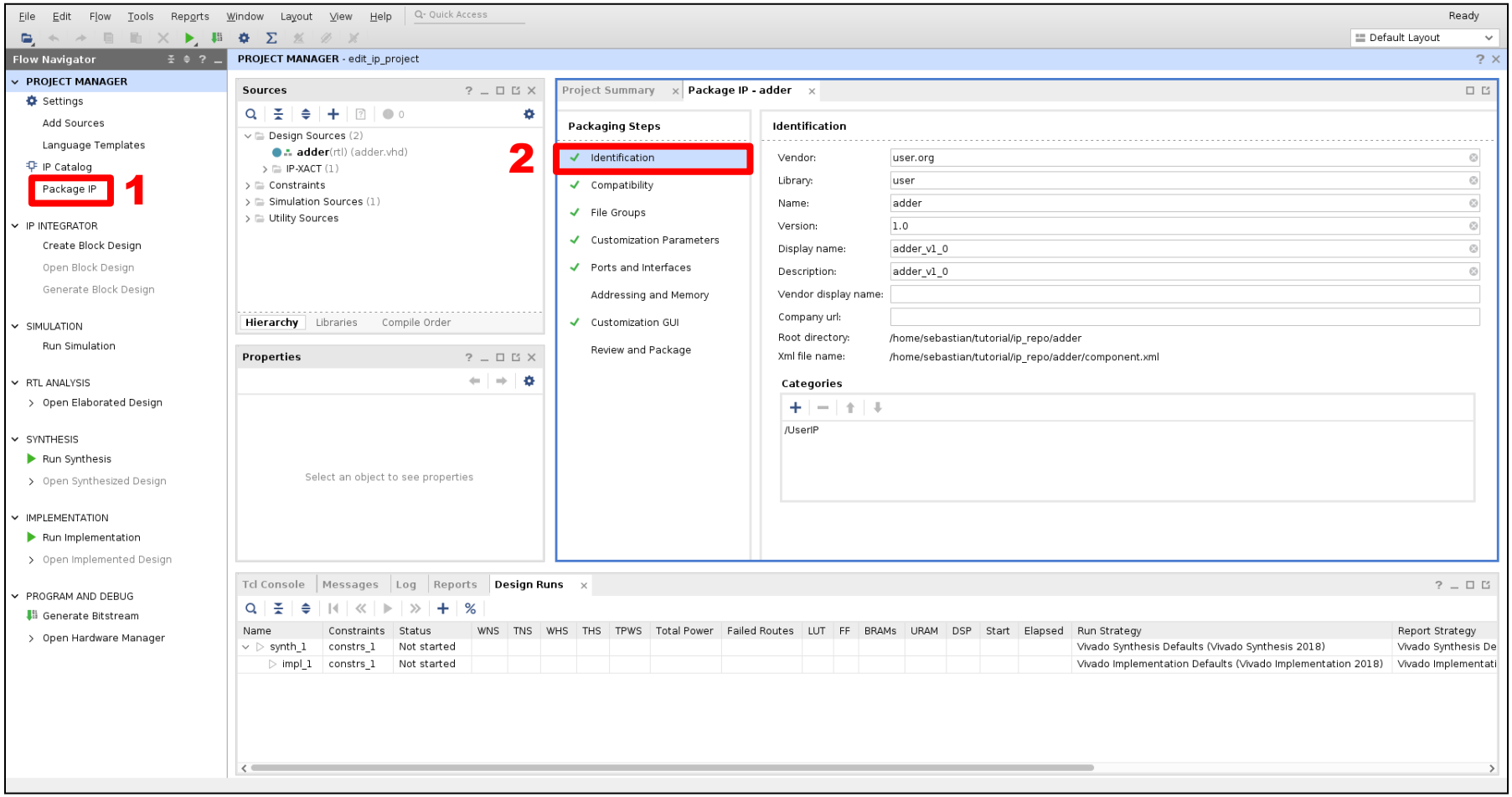
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity adder is
    generic (
        WIDTH : positive := 8
    );
    port (
        A     : in  std_logic_vector(WIDTH-1 downto 0);
        B     : in  std_logic_vector(WIDTH-1 downto 0);
        S     : out std_logic_vector(WIDTH-1 downto 0)
    );
end entity;

architecture rtl of adder is
begin
    S <= std_logic_vector(unsigned(A) + unsigned(B));
end architecture;
```

Generics in VHDL become **Customization Parameters** in Vivado

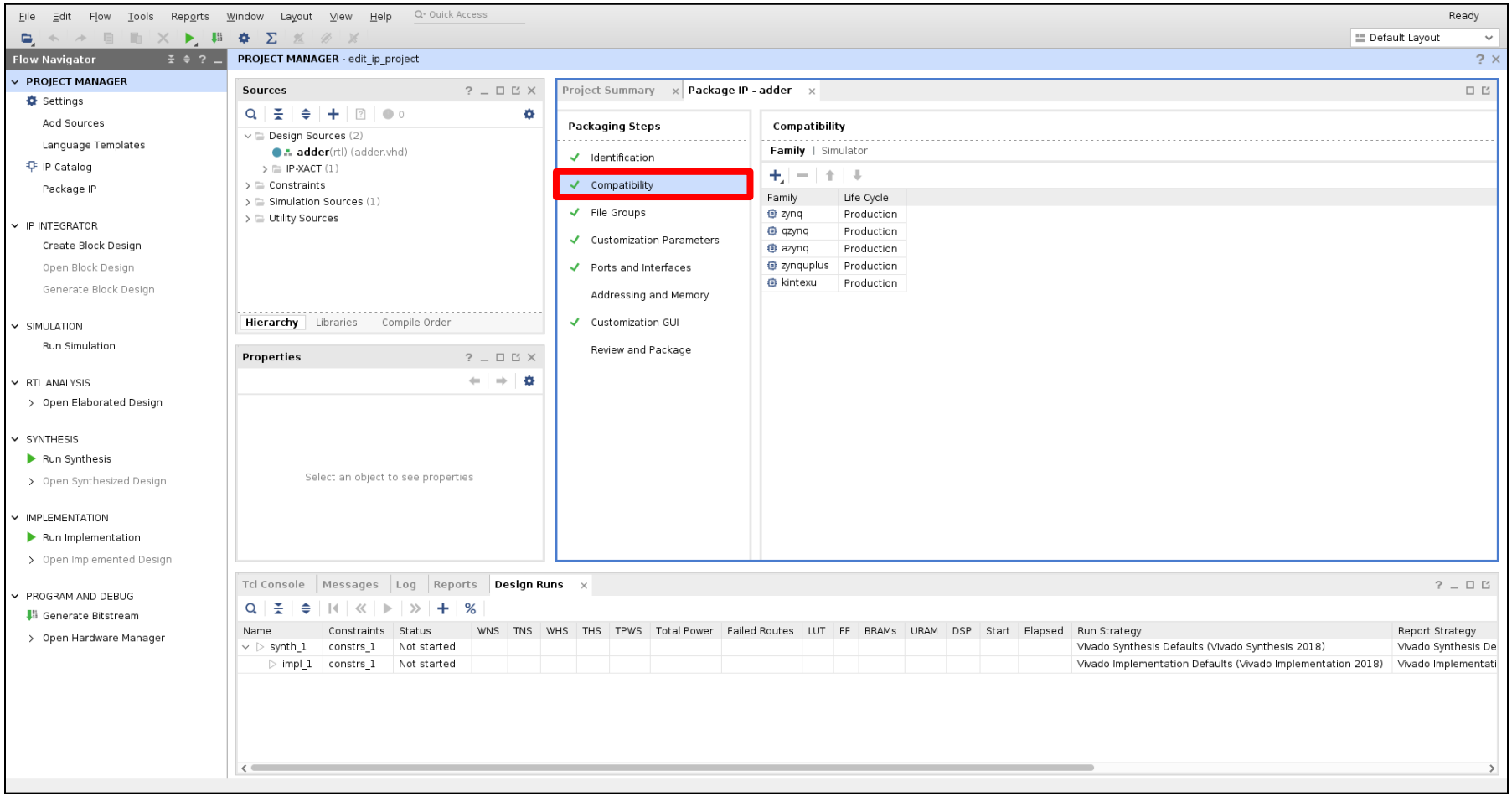
Ports in VHDL become **Ports and Interfaces** in Vivado

Click ***Package IP***. Click ***Identification***.

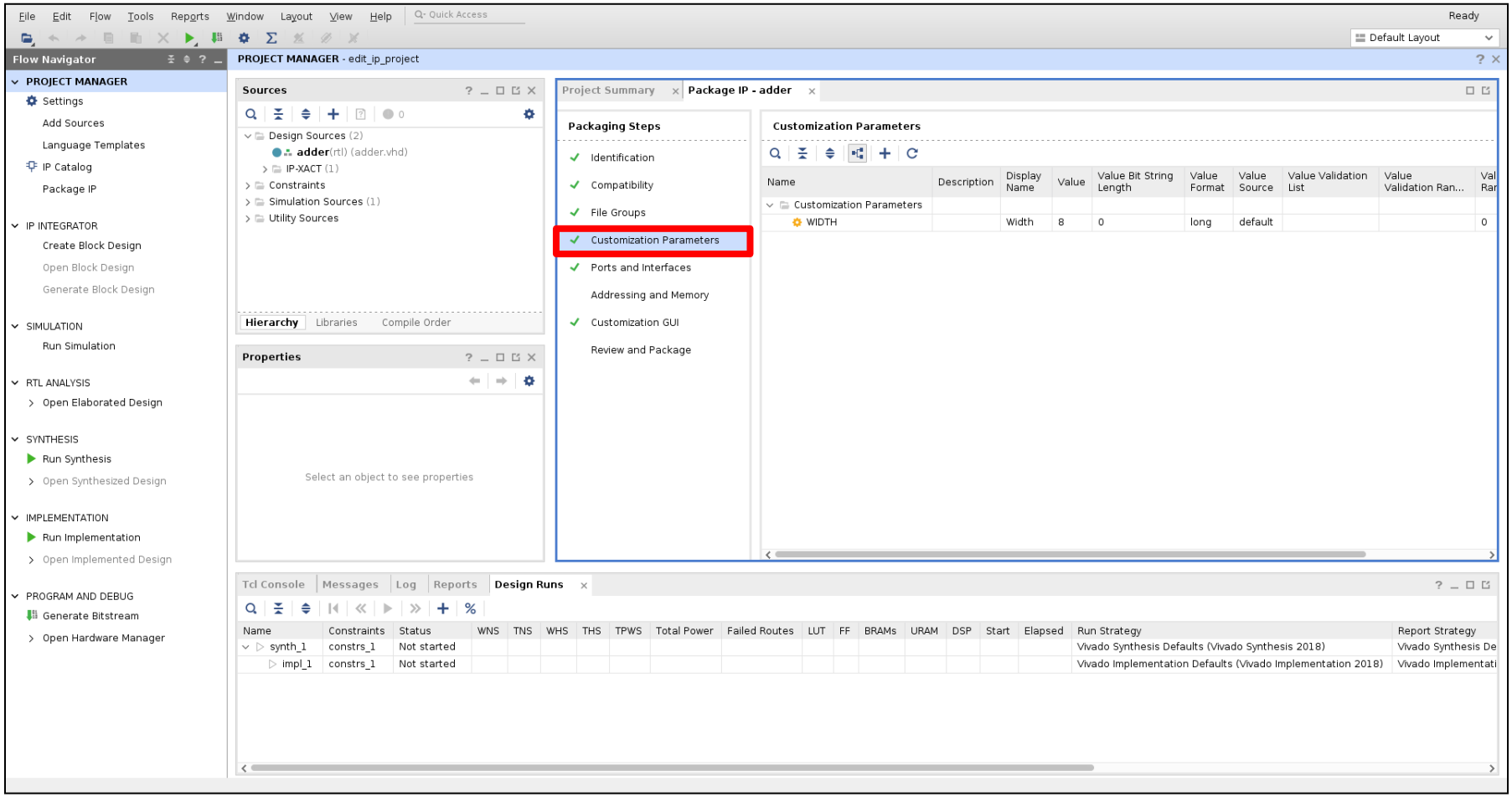
NOTE: Here you can specify the vendor, library, name, version, and description of your IP.



Click ***Compatibility***.
NOTE: Here you can specify SoC/FPGA devices supported by your IP.

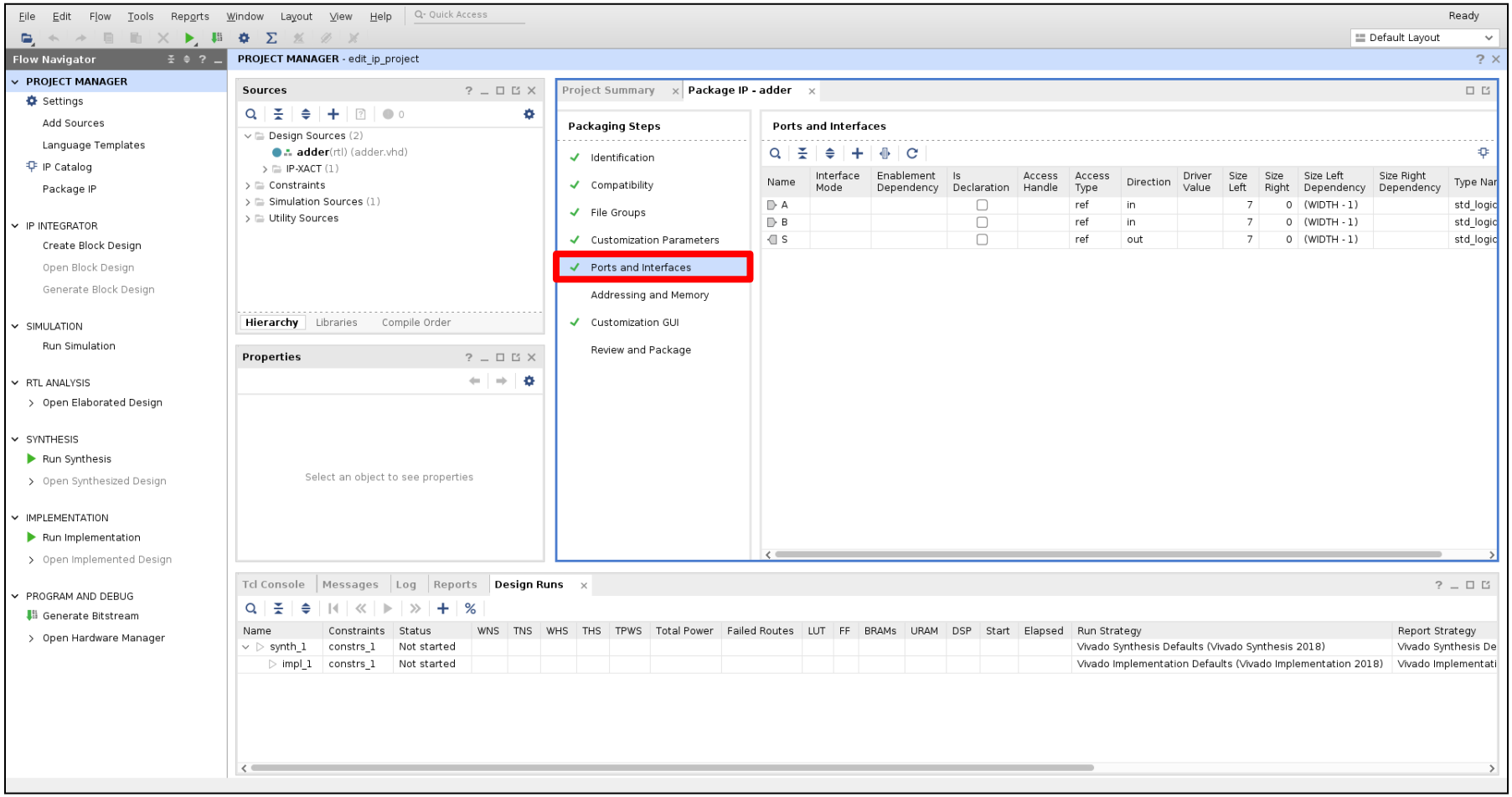


Click ***Customization Parameters***.
NOTE: Generics in VHDL are inferred as customization parameters (e.g., **WIDTH**).



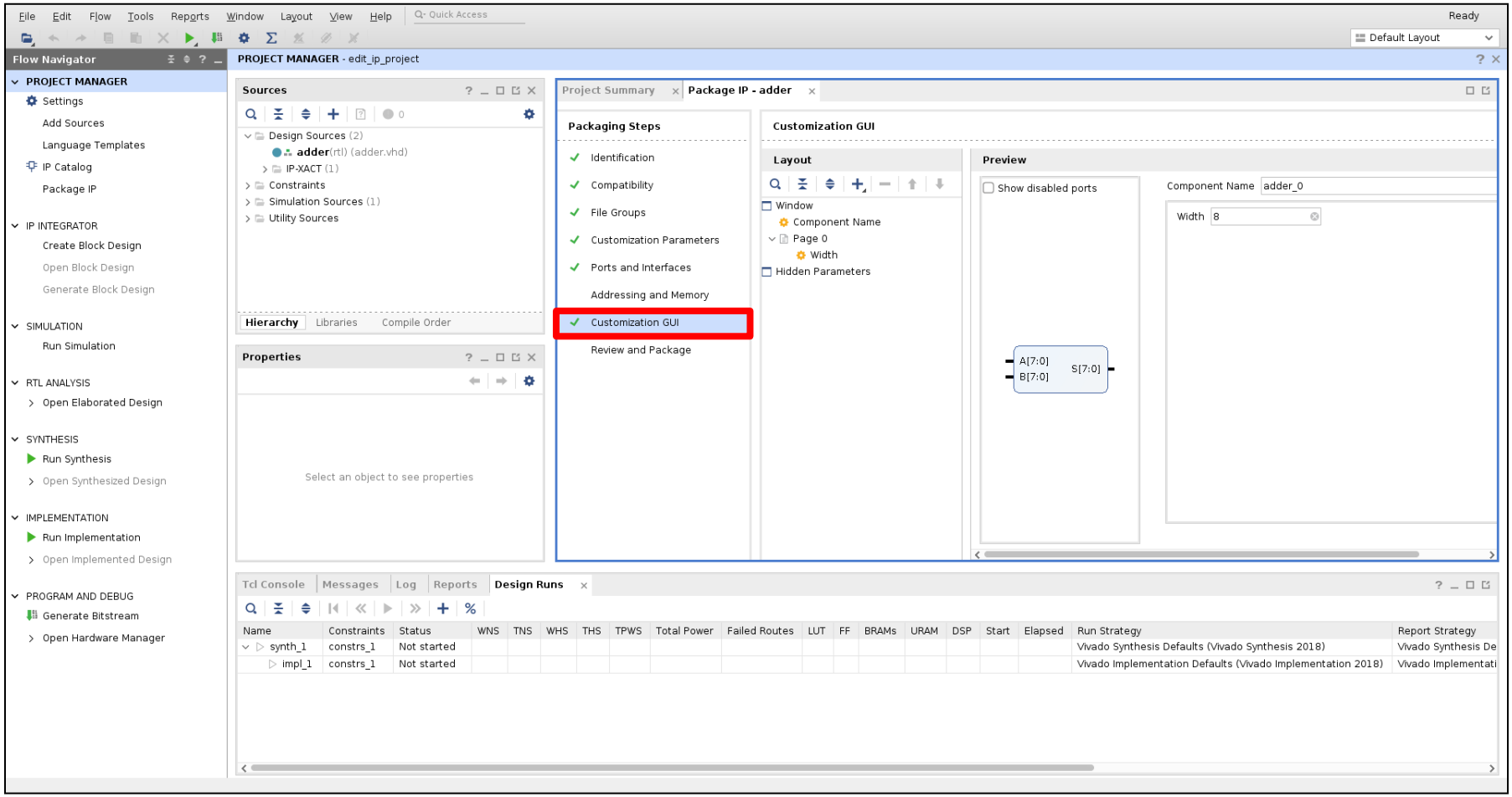
Click ***Ports and Interfaces***.

NOTE: Ports in VHDL are inferred as ports and interfaces (e.g., **A, B,** and **S**).

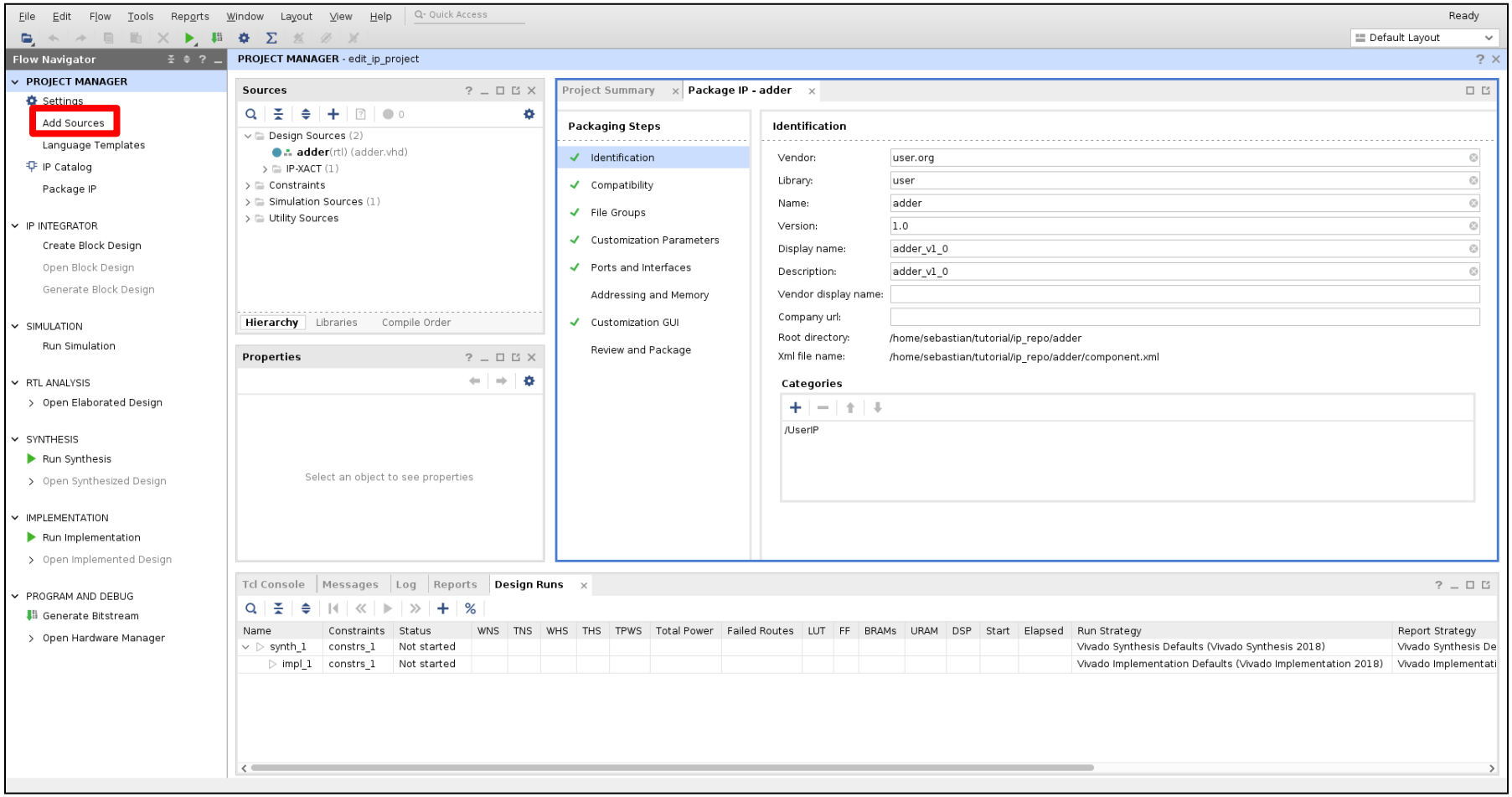


Click ***Customization GUI***.

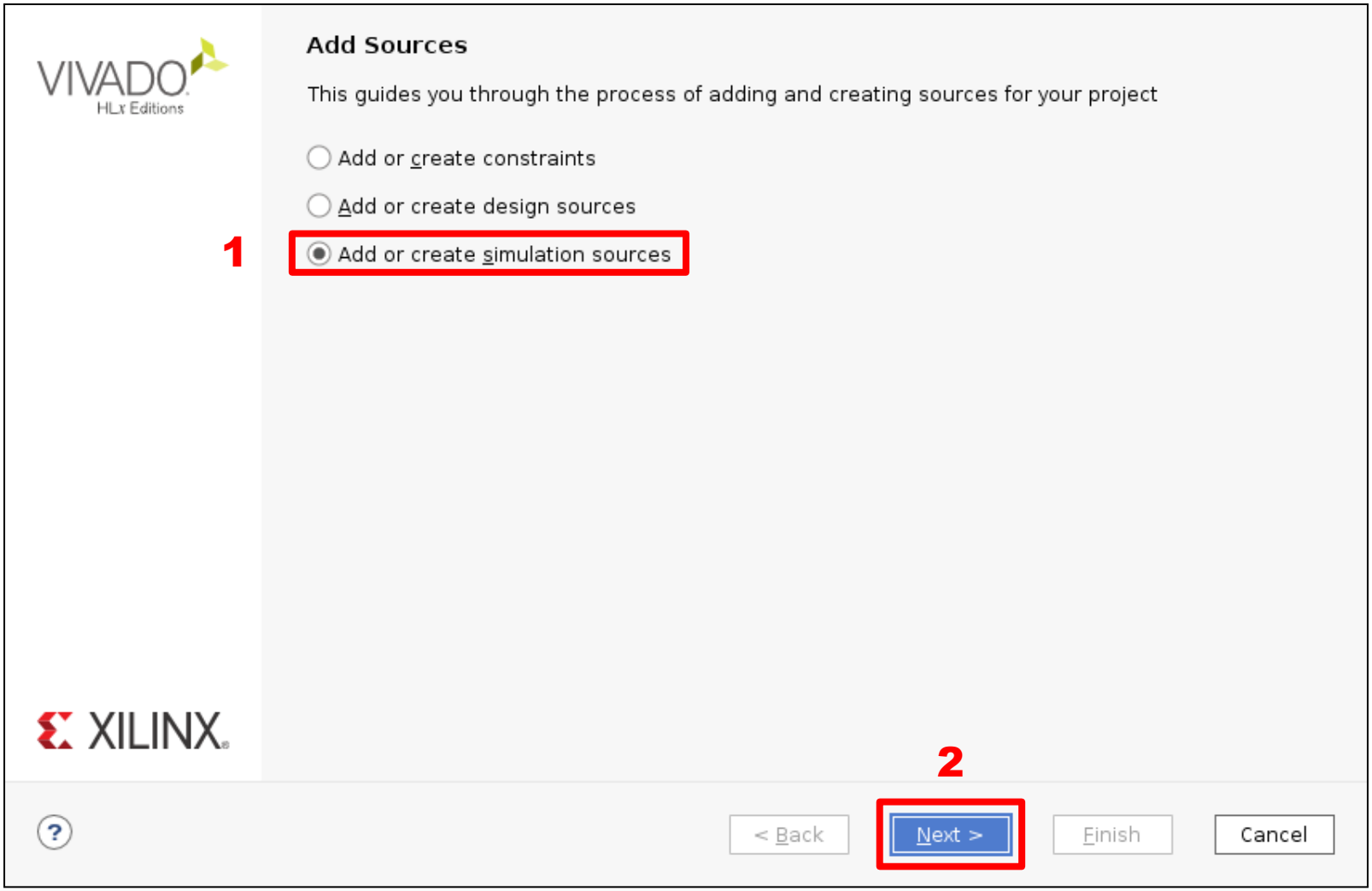
NOTE: This tab illustrates the customization GUI, which allows you to customize parameters of your IP in a block design.



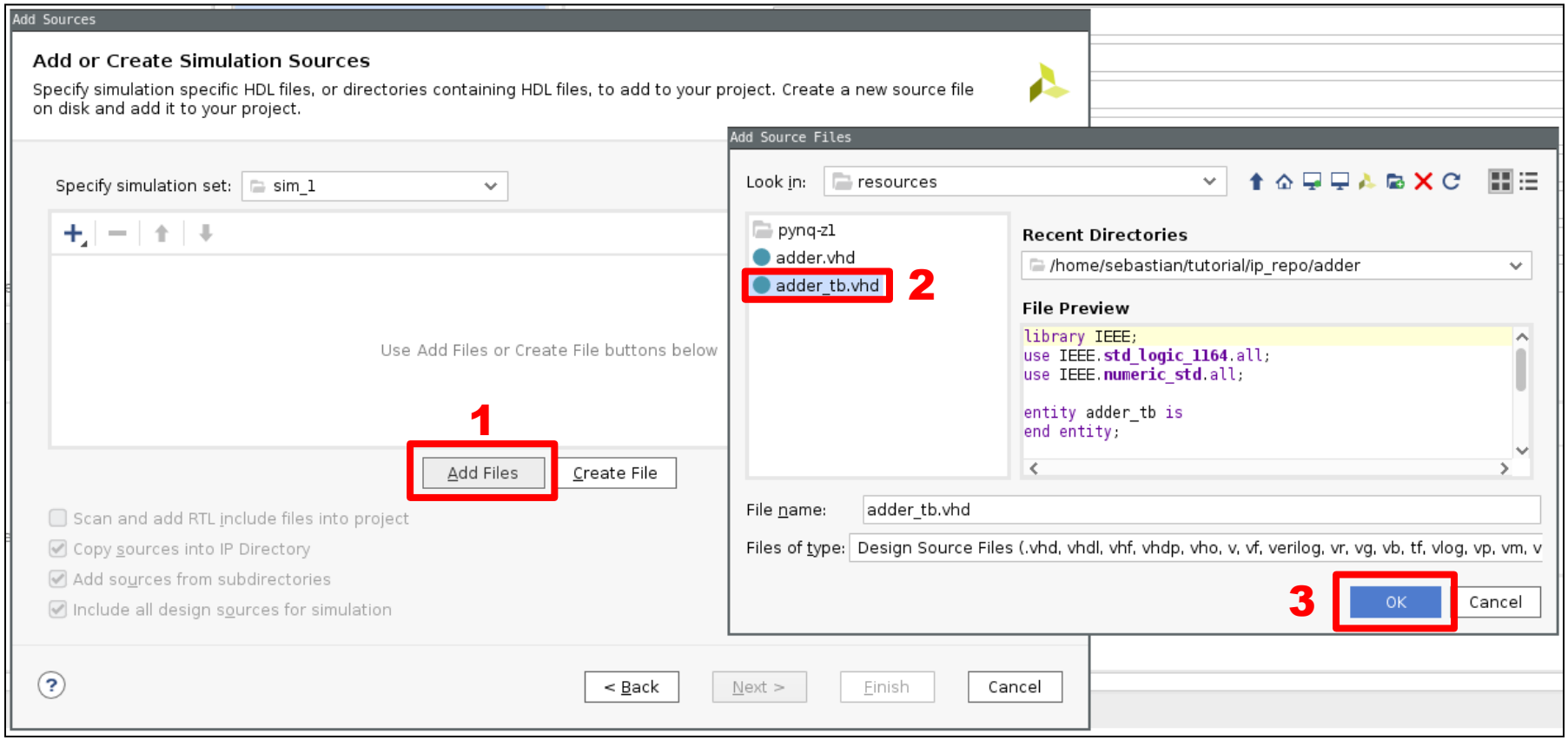
Time to simulate your design. Click ***Add Sources***.



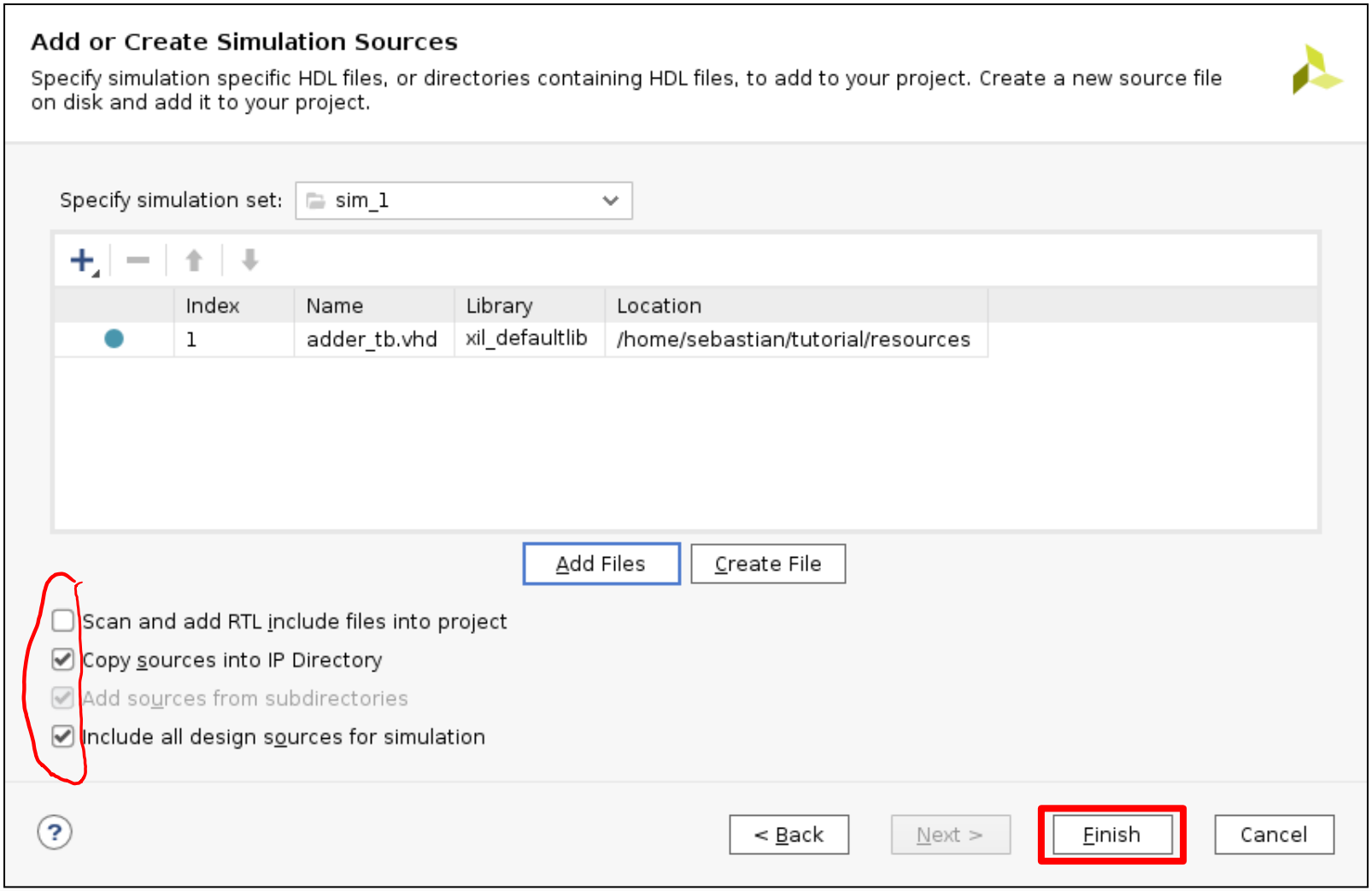
Select ***Add or create simulation sources***. Click ***Next***.



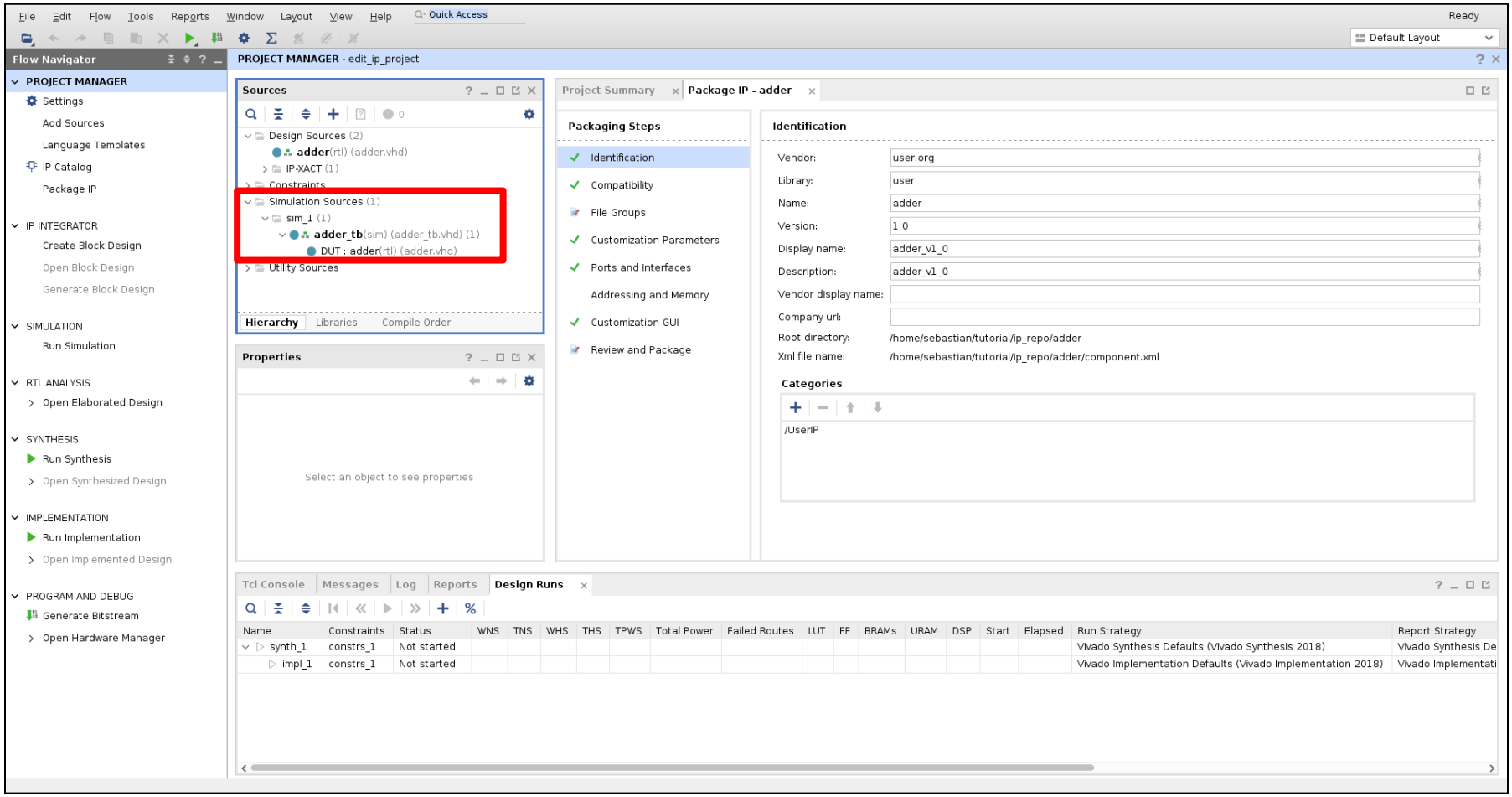
Click ***Add Files***. Navigate to **`Lab_1_repo/resources/adder_tb.vhd`**. Click ***OK***.



Click ***Finish***. Note the checked/unchecked boxes



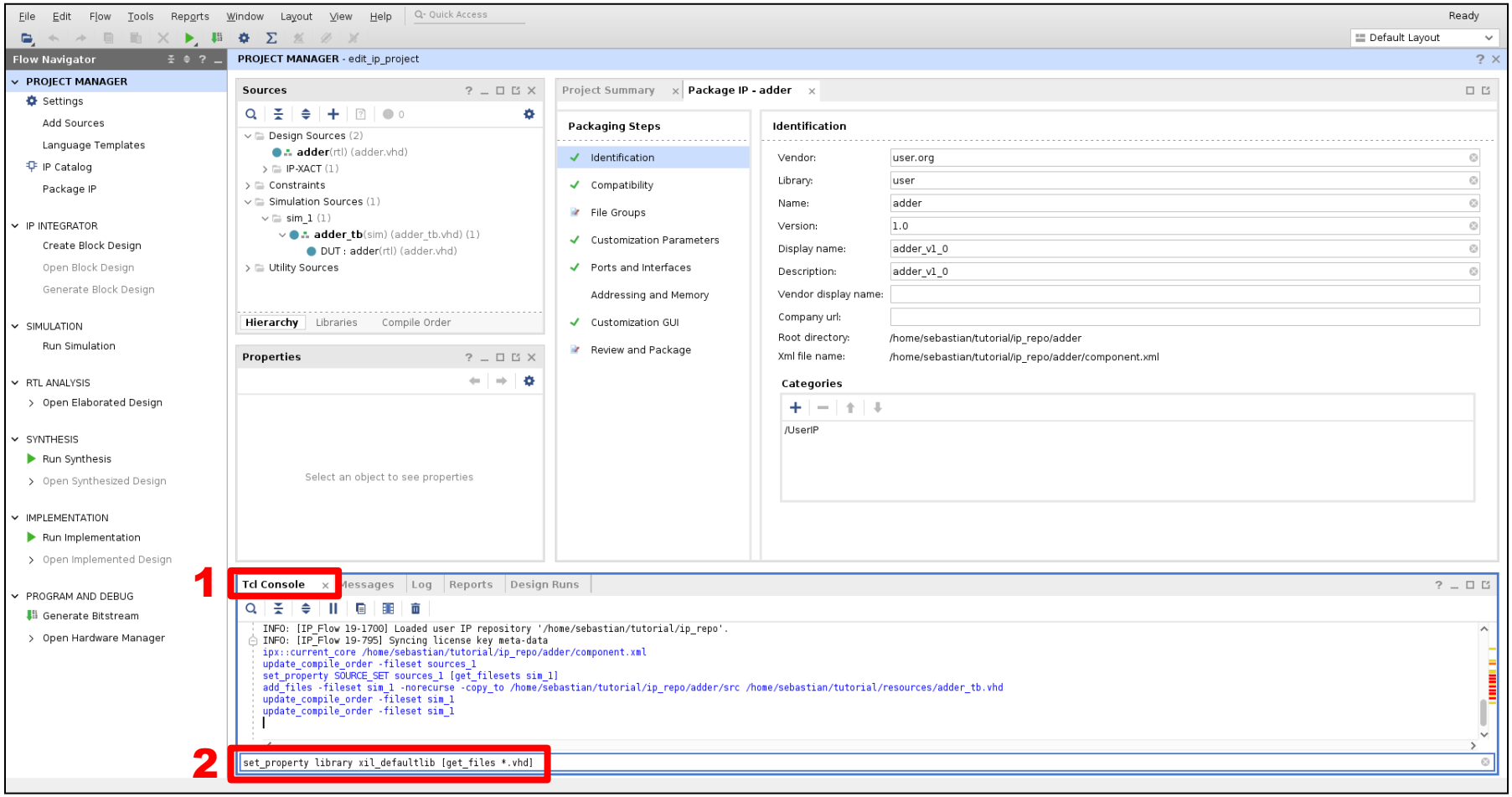
adder_tb.vhd has been added to the project as a simulation source.



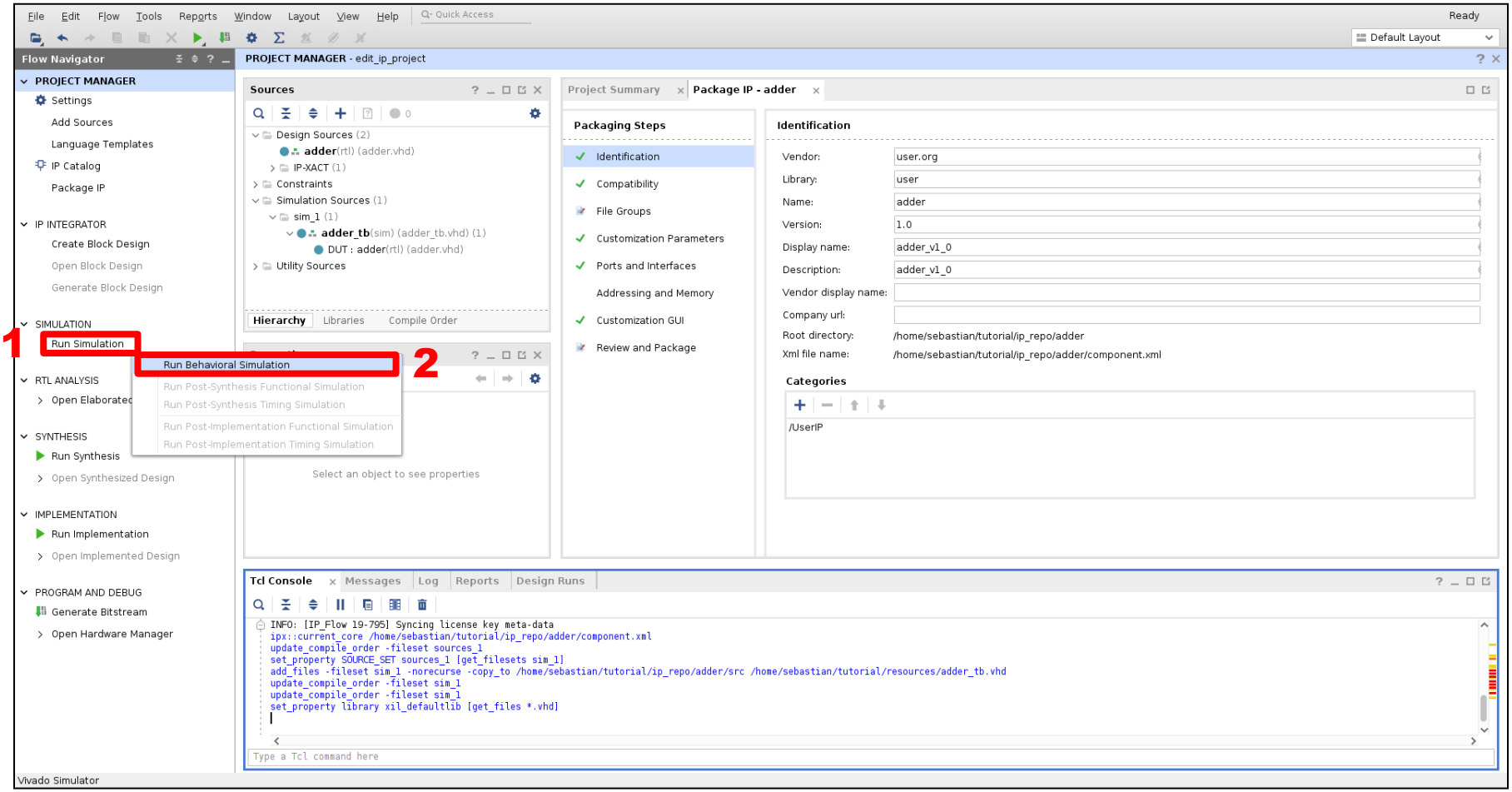
adder_tb.vhd. The main process assigns loop iterators to the adder inputs every 10 ns. Continue.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity adder_tb is
end entity;

architecture sim of adder_tb is
    component adder is
        generic (
            WIDTH : positive := 8
        );
        port (
            A     : in  std_logic_vector(WIDTH-1 downto 0);
            B     : in  std_logic_vector(WIDTH-1 downto 0);
            S     : out std_logic_vector(WIDTH-1 downto 0)
        );
    end component;

    constant WIDTH : positive := 4;
    signal A       : std_logic_vector(WIDTH-1 downto 0);
    signal B       : std_logic_vector(WIDTH-1 downto 0);
    signal S       : std_logic_vector(WIDTH-1 downto 0);
begin
    DUT: adder
        generic map (
            WIDTH => WIDTH
        )
        port map (
            A     => A,
            B     => B,
            S     => S
        );

    process
    begin
        for i in 0 to 2**WIDTH-1 loop
            for j in 0 to 2**WIDTH-1 loop
                A <= std_logic_vector(to_unsigned(i, WIDTH));
                B <= std_logic_vector(to_unsigned(j, WIDTH));
                wait for 10 ns;
            end loop;
        end loop;
        wait;
    end process;
end architecture;
```

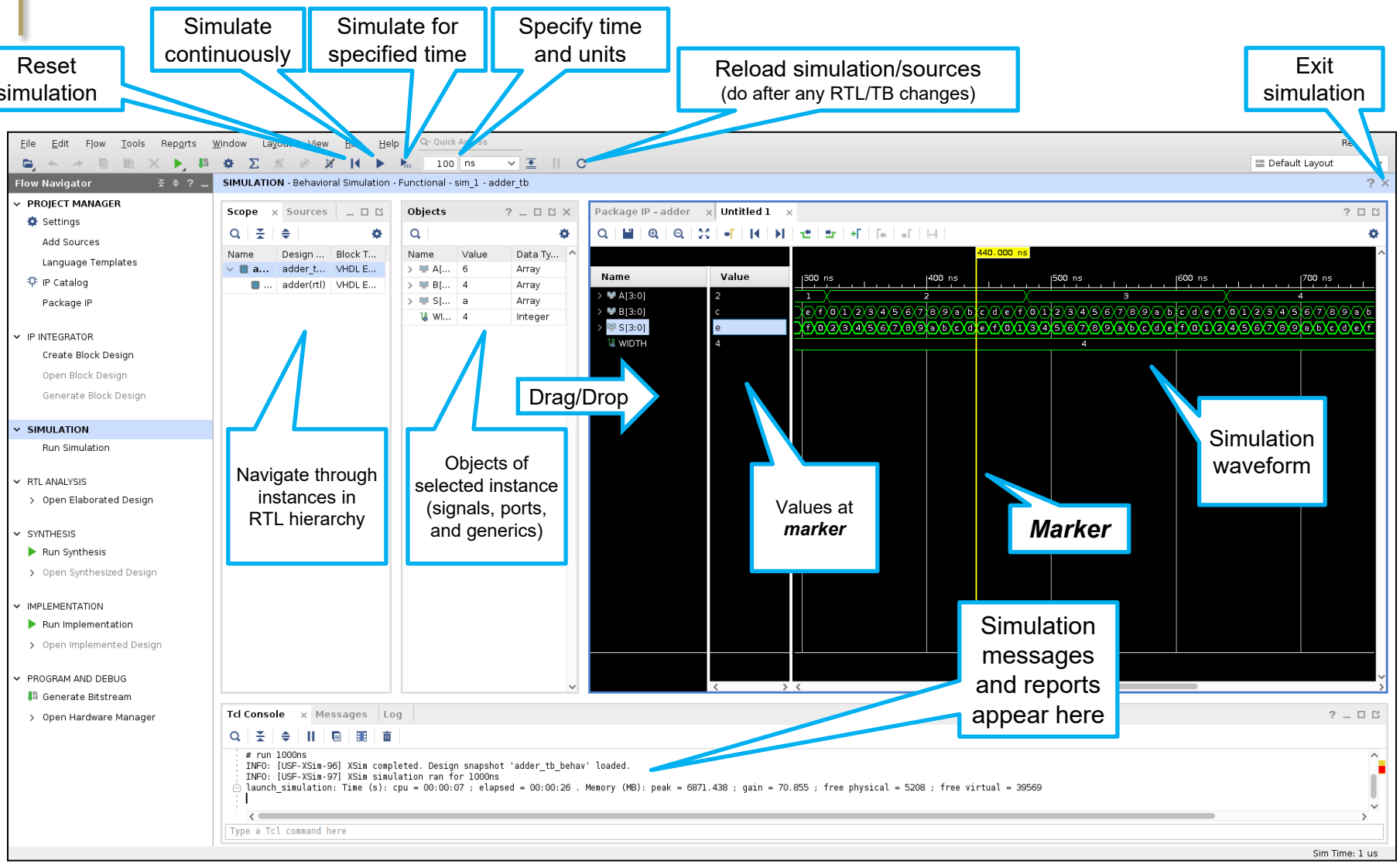
Click ***Tcl Console*** tab.
Enter Tcl command **`set_property library xil_defaultlib [get_files *.vhd]`**.



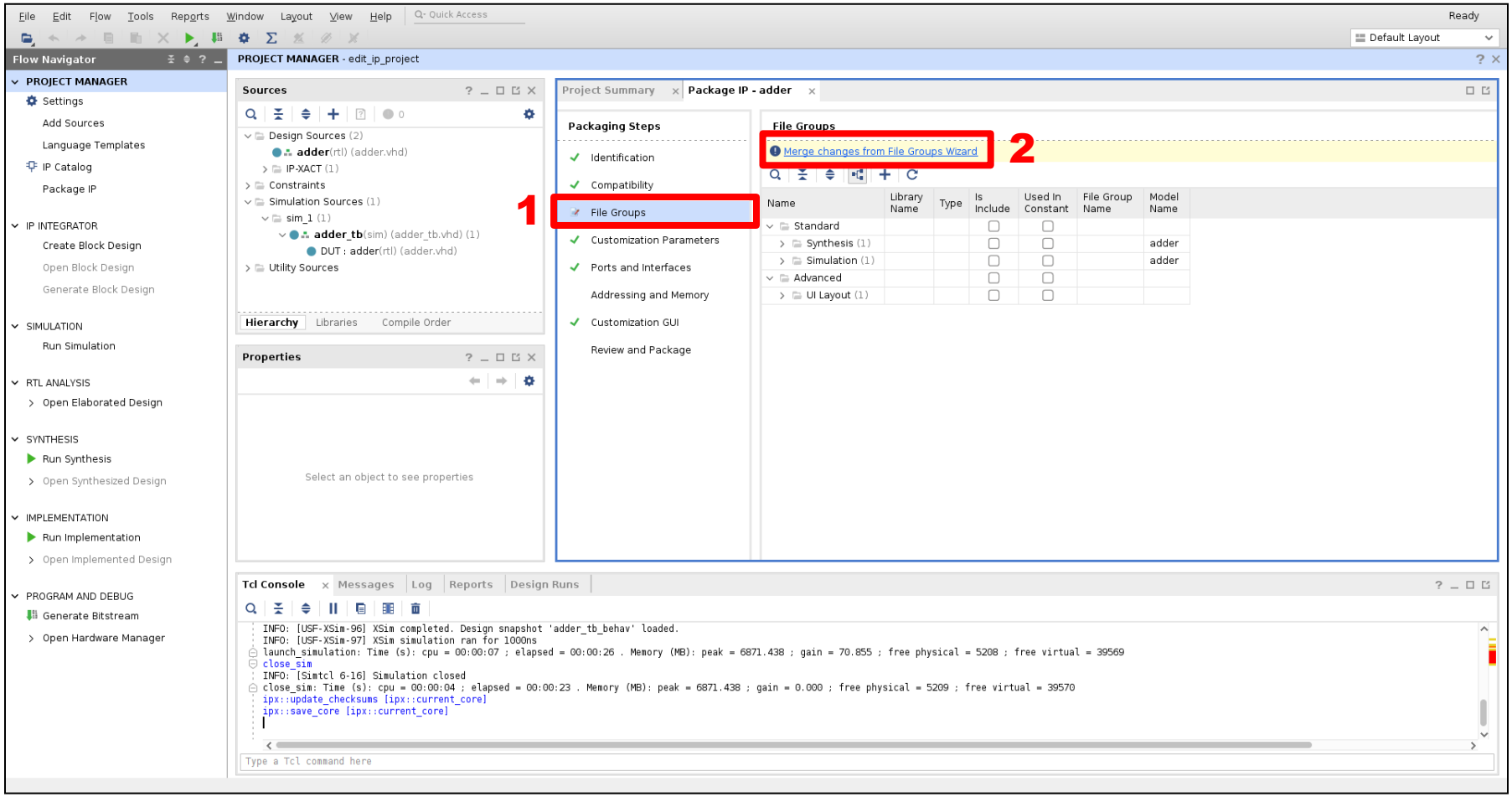
Click ***Run Simulation***. Click ***Behavioral Simulation***.



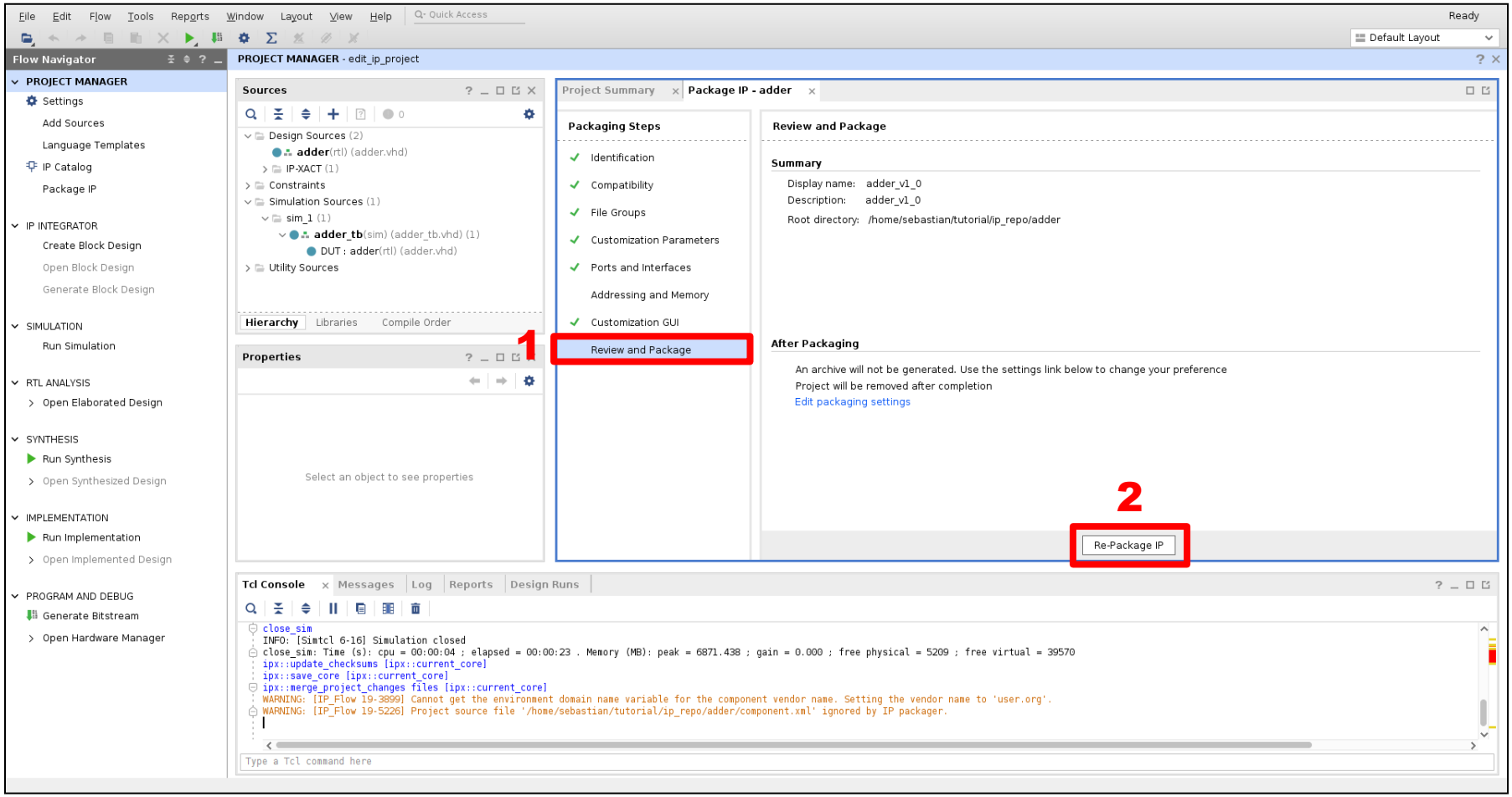
This is the Simulation GUI. Continue.



Click ***File Groups***. Click ***Merge changes from File Groups Wizards***.

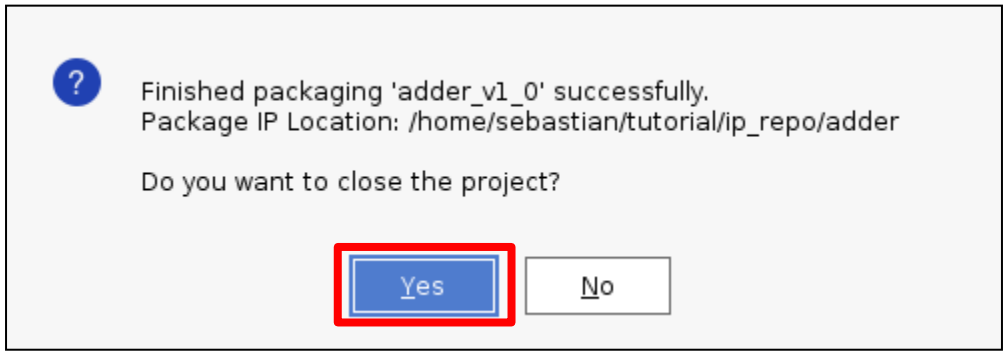


Click ***Review and Package***. Click ***Re-Package IP***.

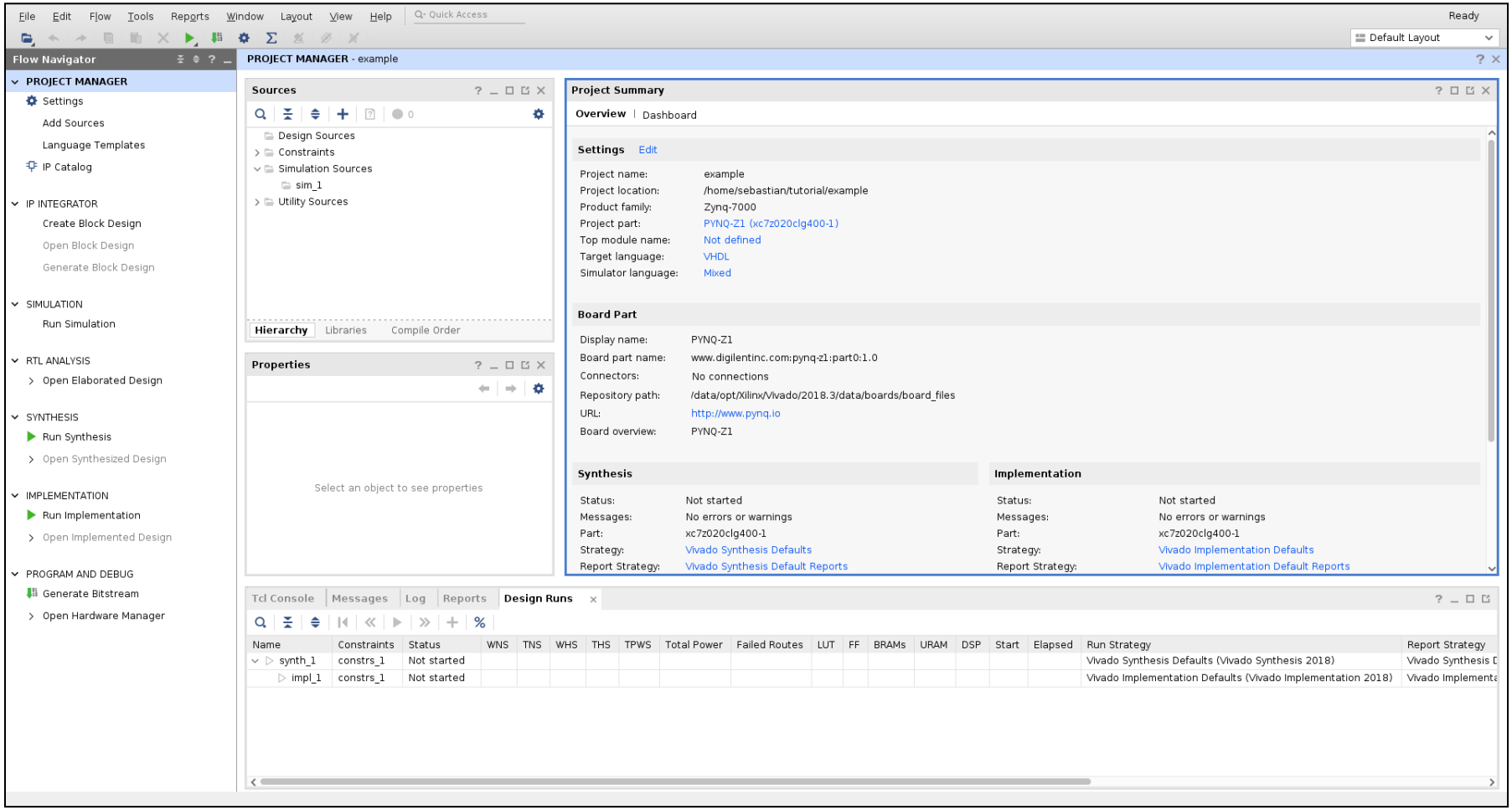


Click ***Yes***.

Note: for future projects, you might want to leave this instance opened in case you need to edit and re-package your design.



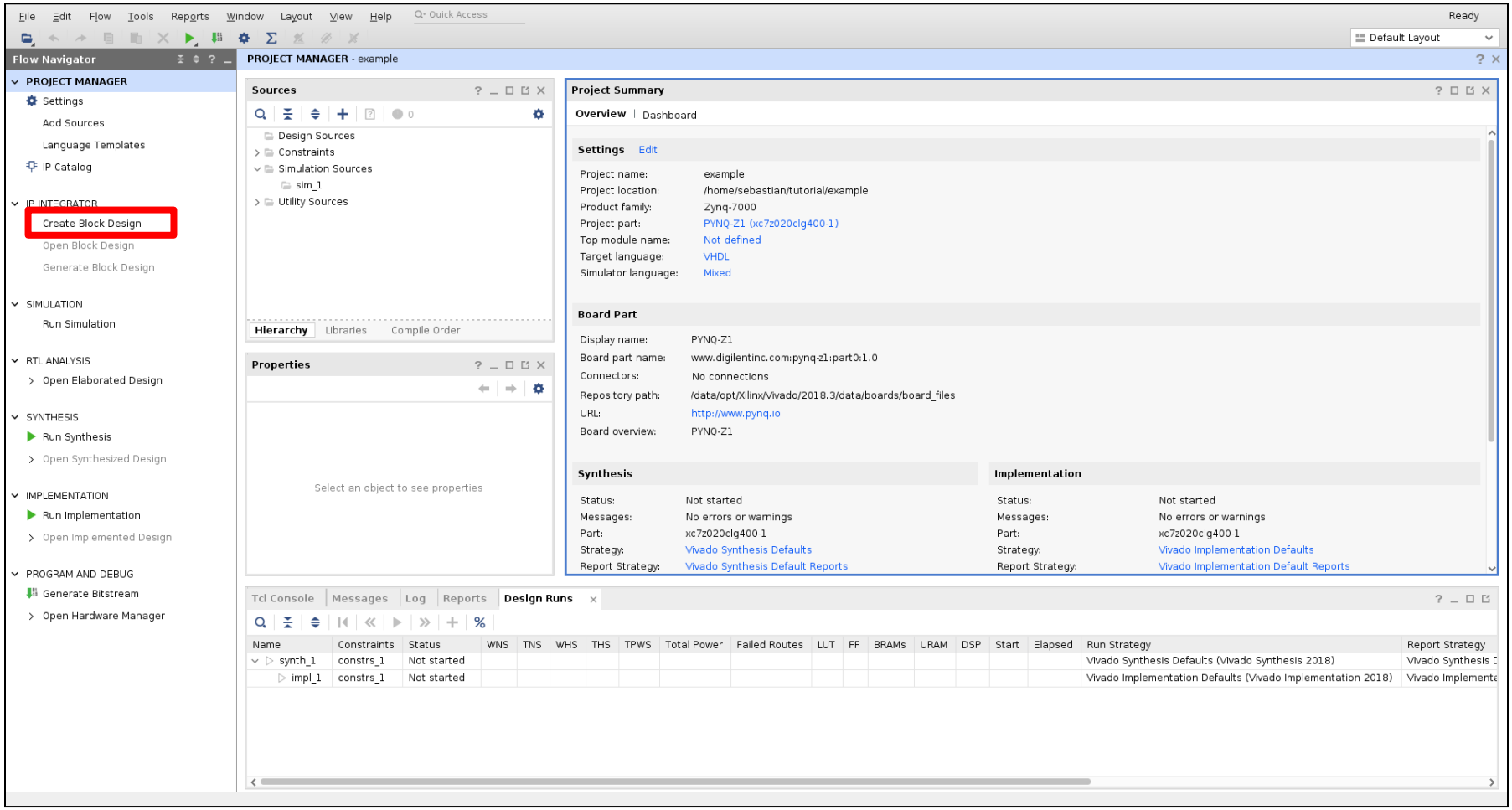
We return to the main Vivado instance. Continue.



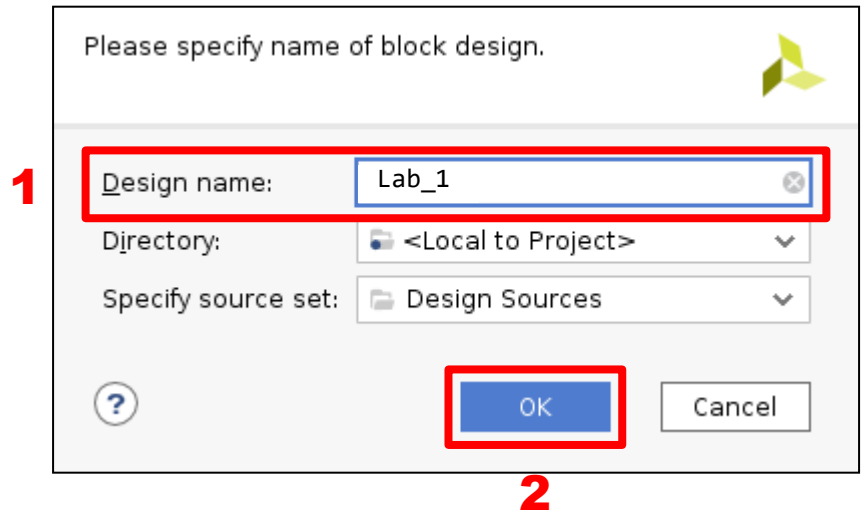
Part 4: Integrating a System

Next, we will integrate a system interfacing the **axi_regmap** and **adder** IP to the *Zynq PS*.

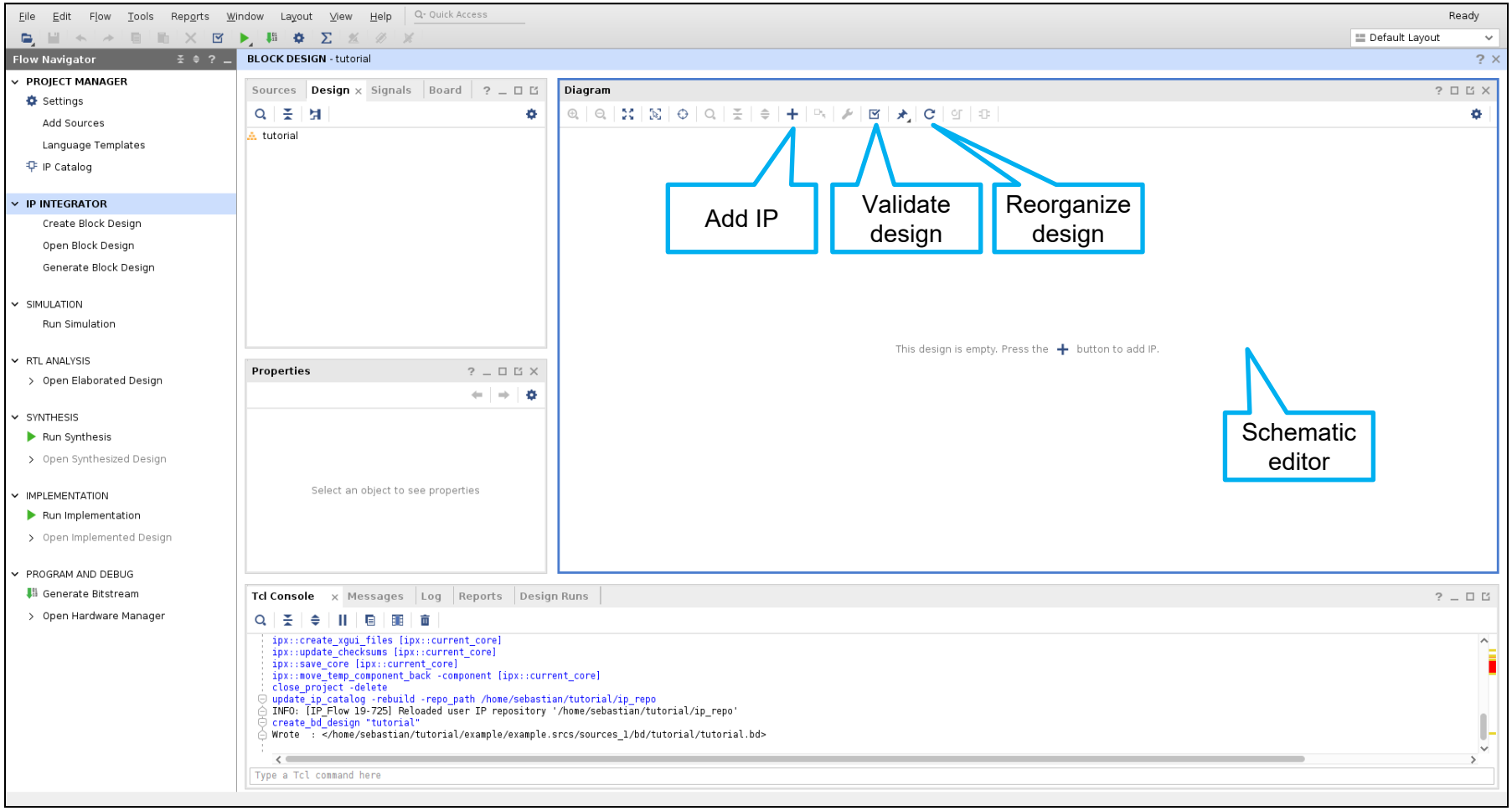
Click ***Create Block Design***.



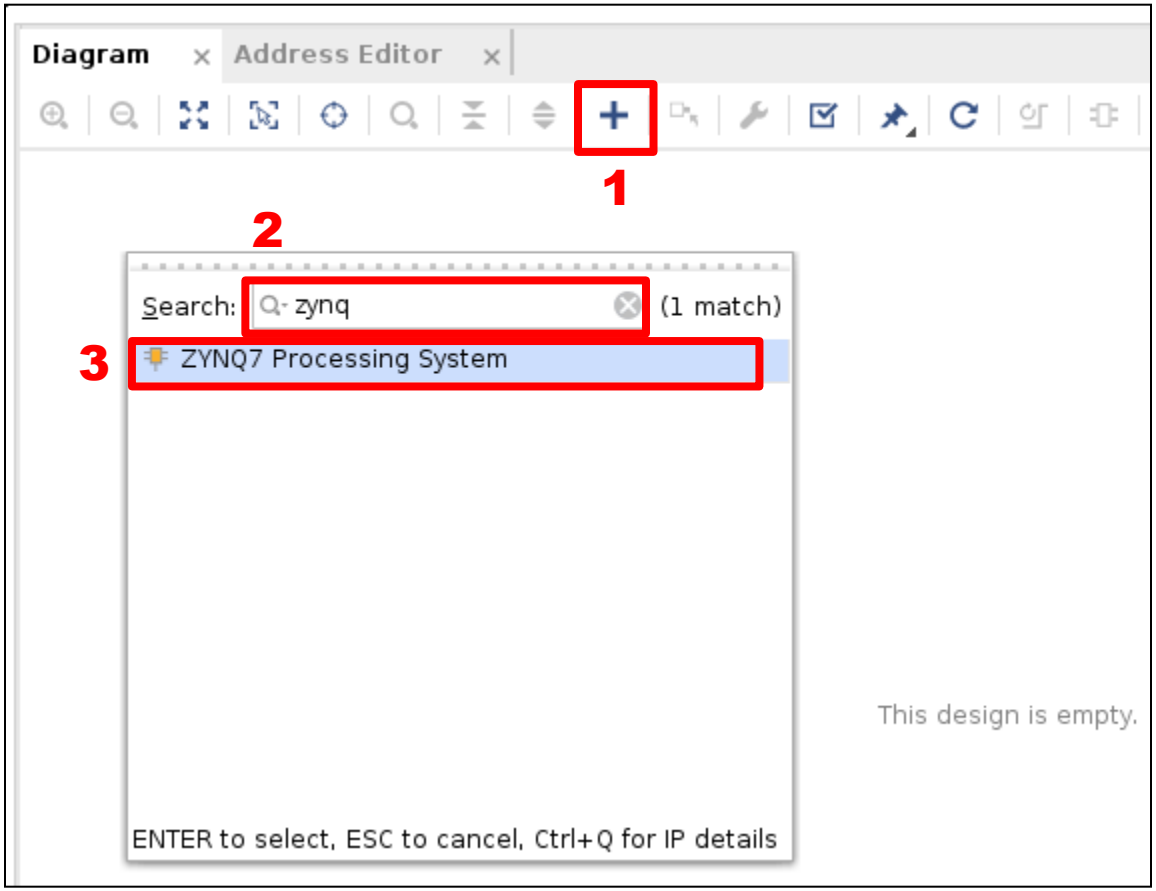
Enter design name **Lab_1**. Click ***OK***.



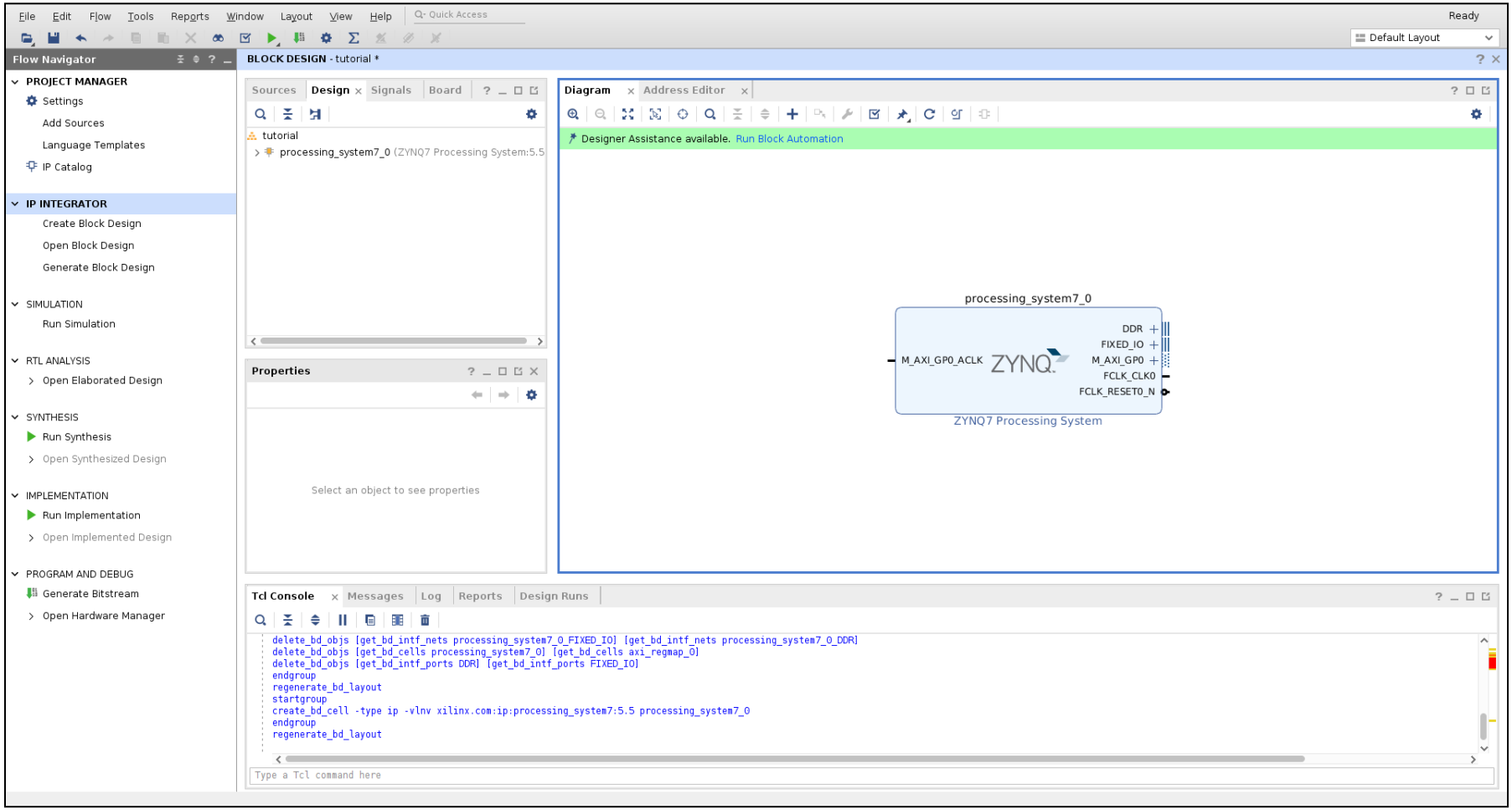
The block design **Lab_1** was created. The diagram (schematic editor) is shown. Continue.



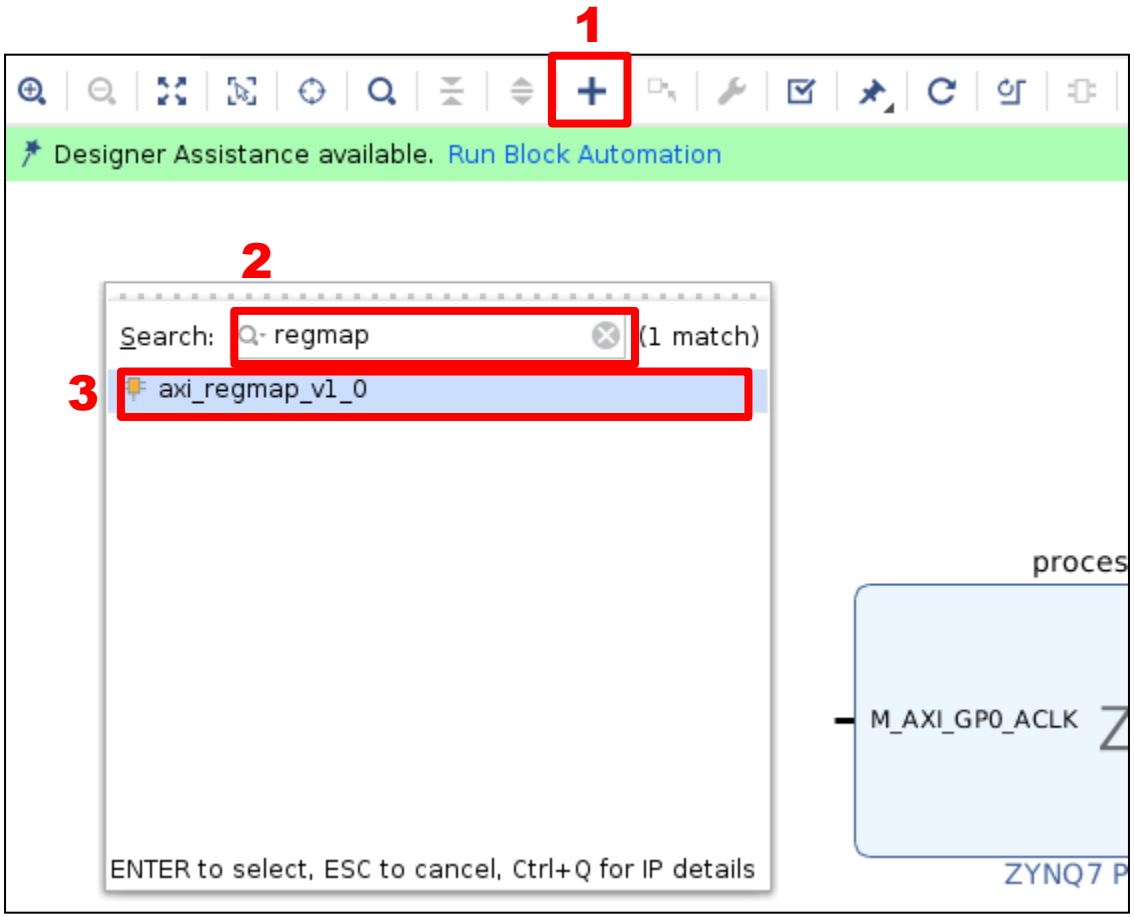
Click **+ *(Add IP)***. Search ***zynq***. Double-click ***ZYNQ7 Processing System***.



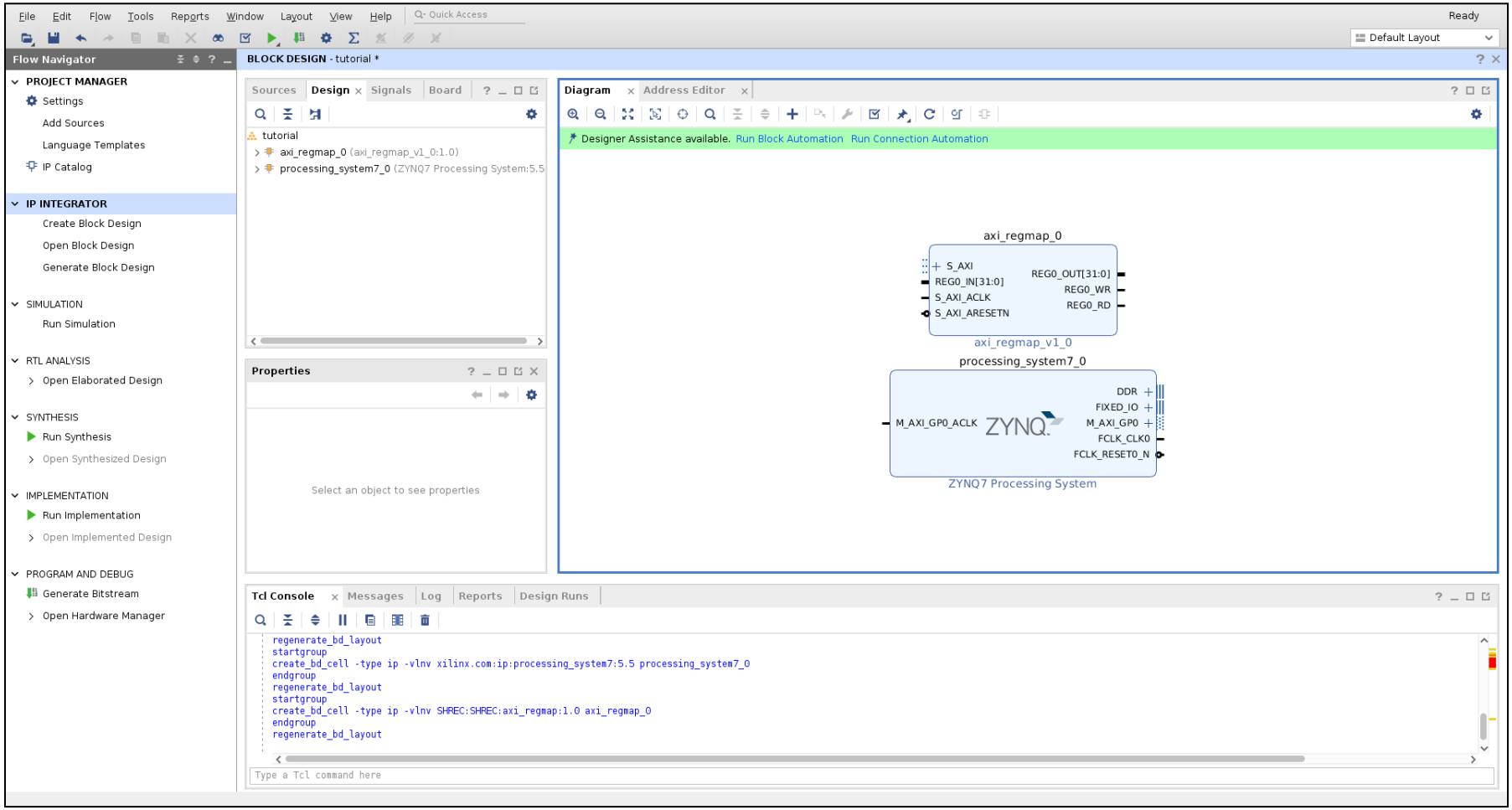
Wild ***ZYNQ7 Processing System*** appeared! The instance name is **processing_system_0**. Continue.



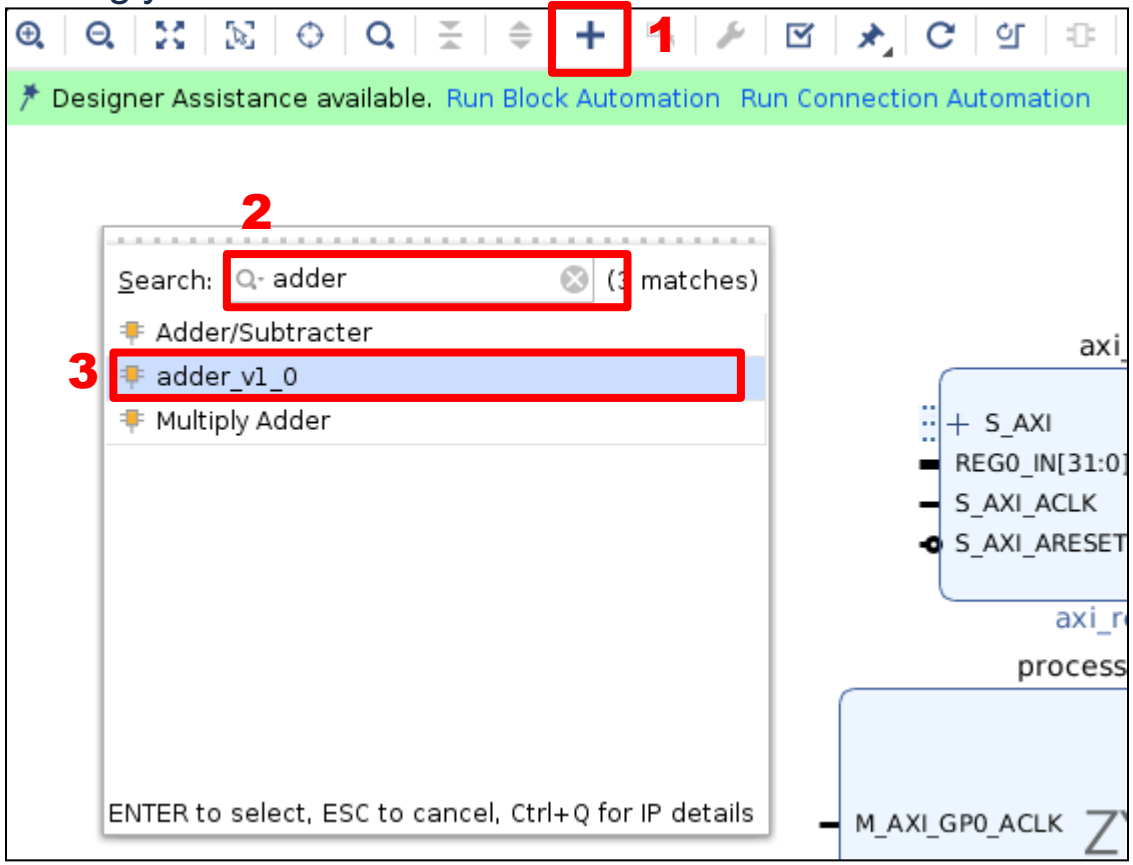
Click **+ *(Add IP)***. Search ***regmap***. Double-click ***axi_regmap_v1_0***.



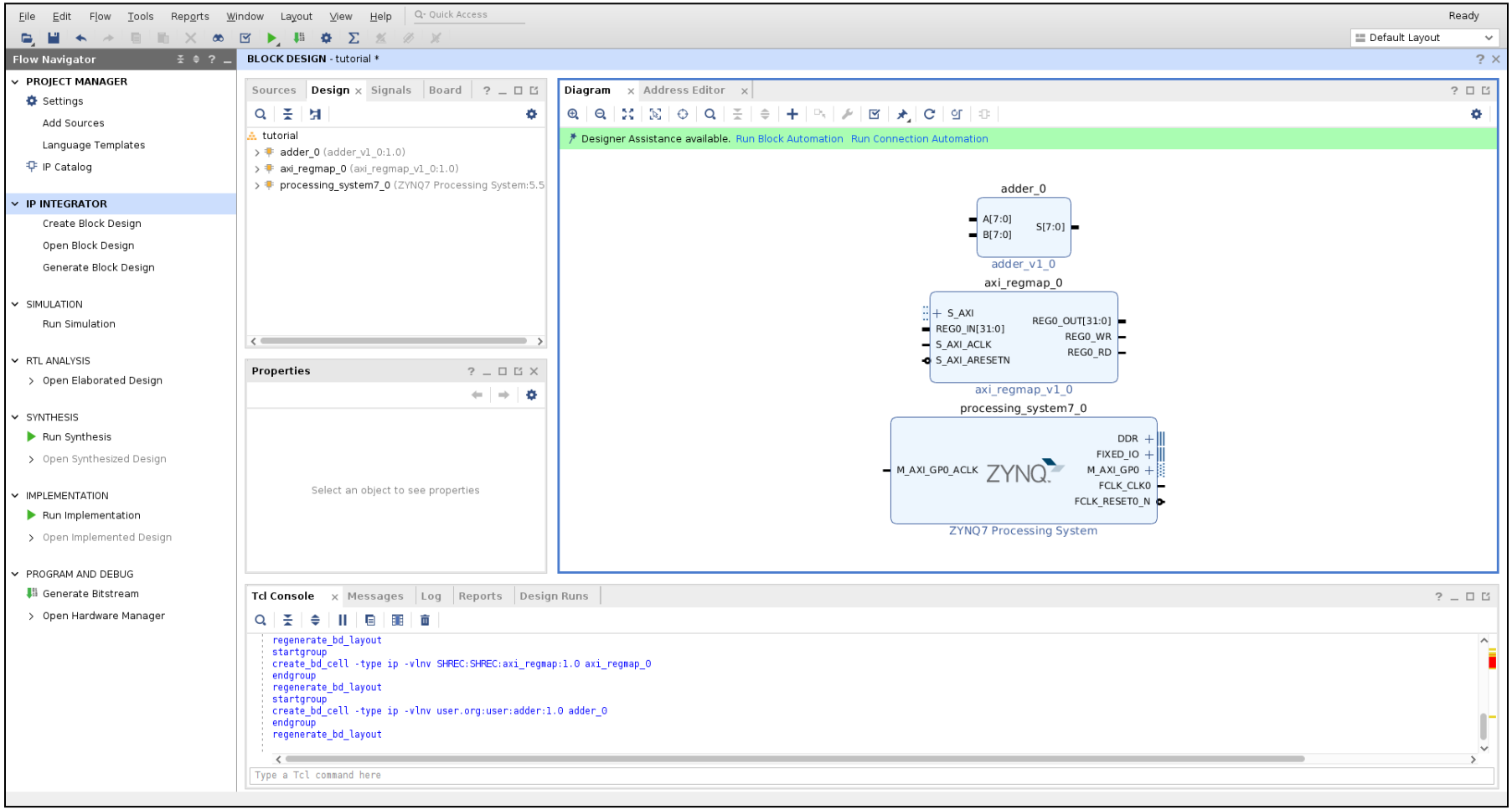
Wild ***axi_regmap_v1_0*** appeared! The instance name is **axi_regmap_0**. Continue.



Click **+ *(Add IP)***. Search ***adder***. Double-click ***adder_v1_0***.
Note 1: if you can't find it, click IP Catalog -> right click in the opened window -> refresh
Note 2: you can also switch to the **Sources** tab, click on **adder.vhd** and drag it to the Diagram window. (if you did it this way, you can skip slides 83-91. These are to edit the VHDL source of an IP added through **+ (Add IP)** or **Note 1.** To edit a module added via **Note 2**, you just double click on the design file (.vhd), edit it, then the block diagram will be updated accordingly.

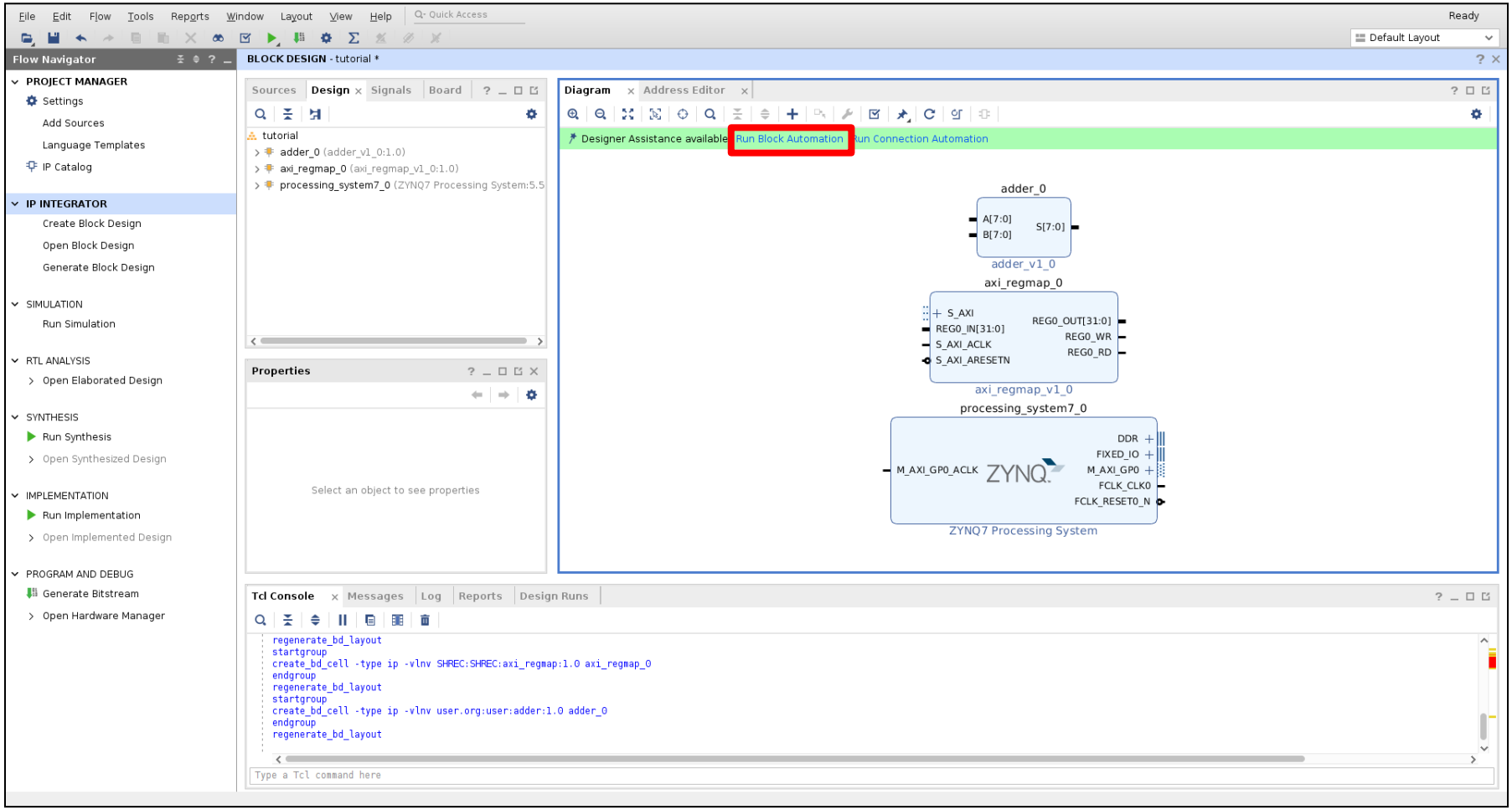


Wild ***adder_v1_0*** appeared! The instance name is **adder_0**. Continue.

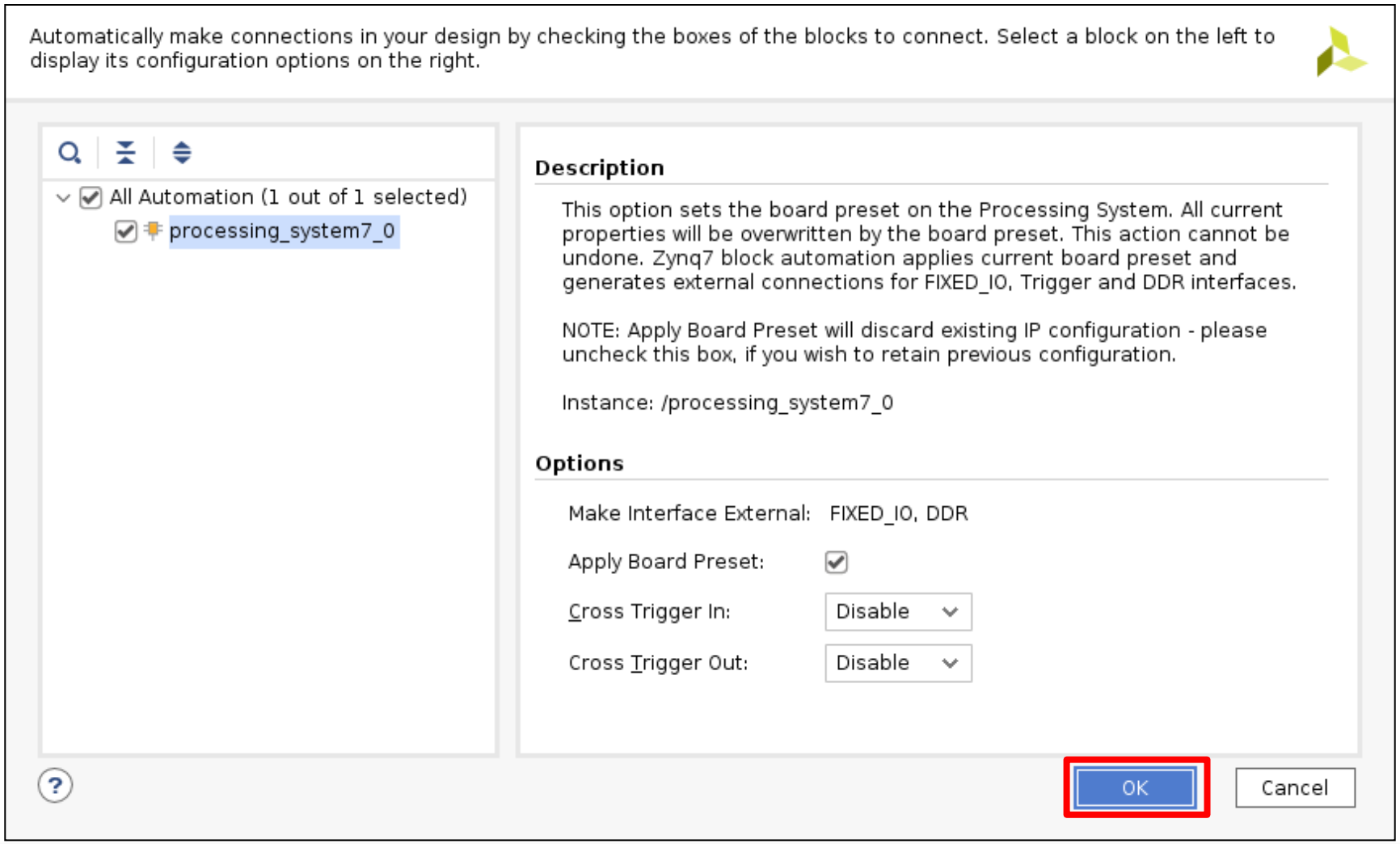


Click ***Run Block Automation***.

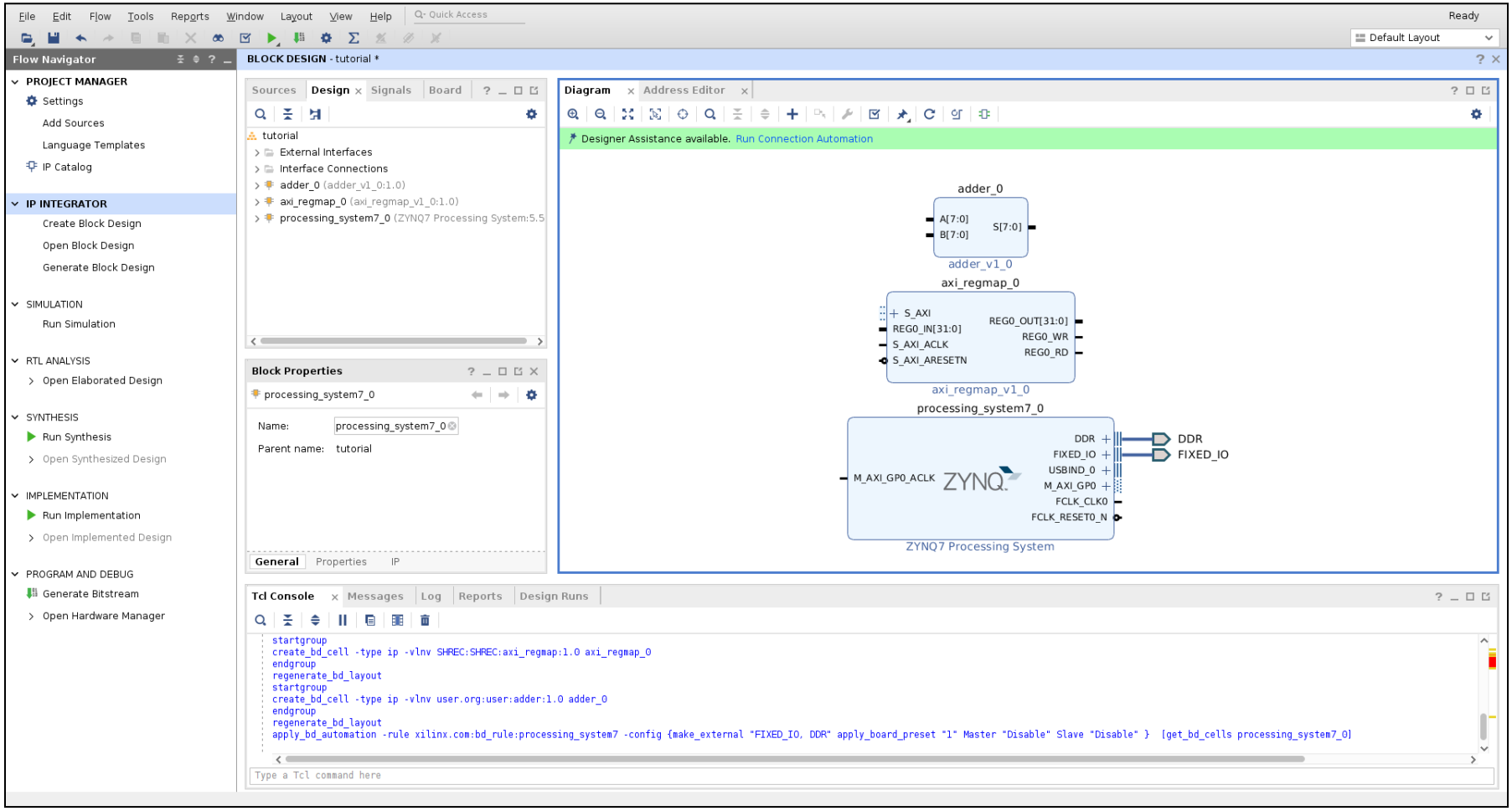
NOTE: This process will configure the **processing_system_0** to the board configuration specified by the PYNQ-Z1 board files.



Click ***OK***.

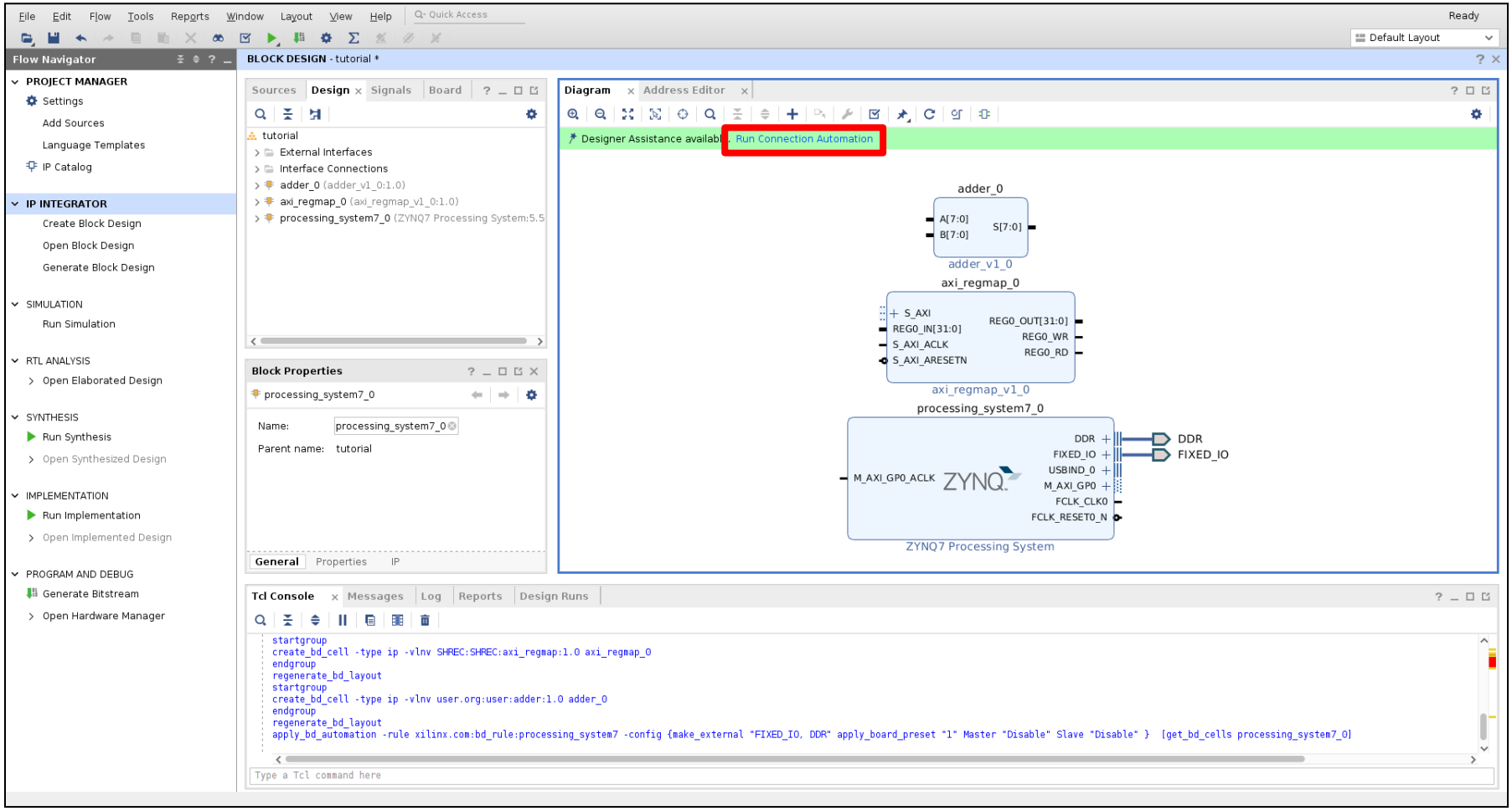


The **processing_system_0** IP has been configured. Continue.

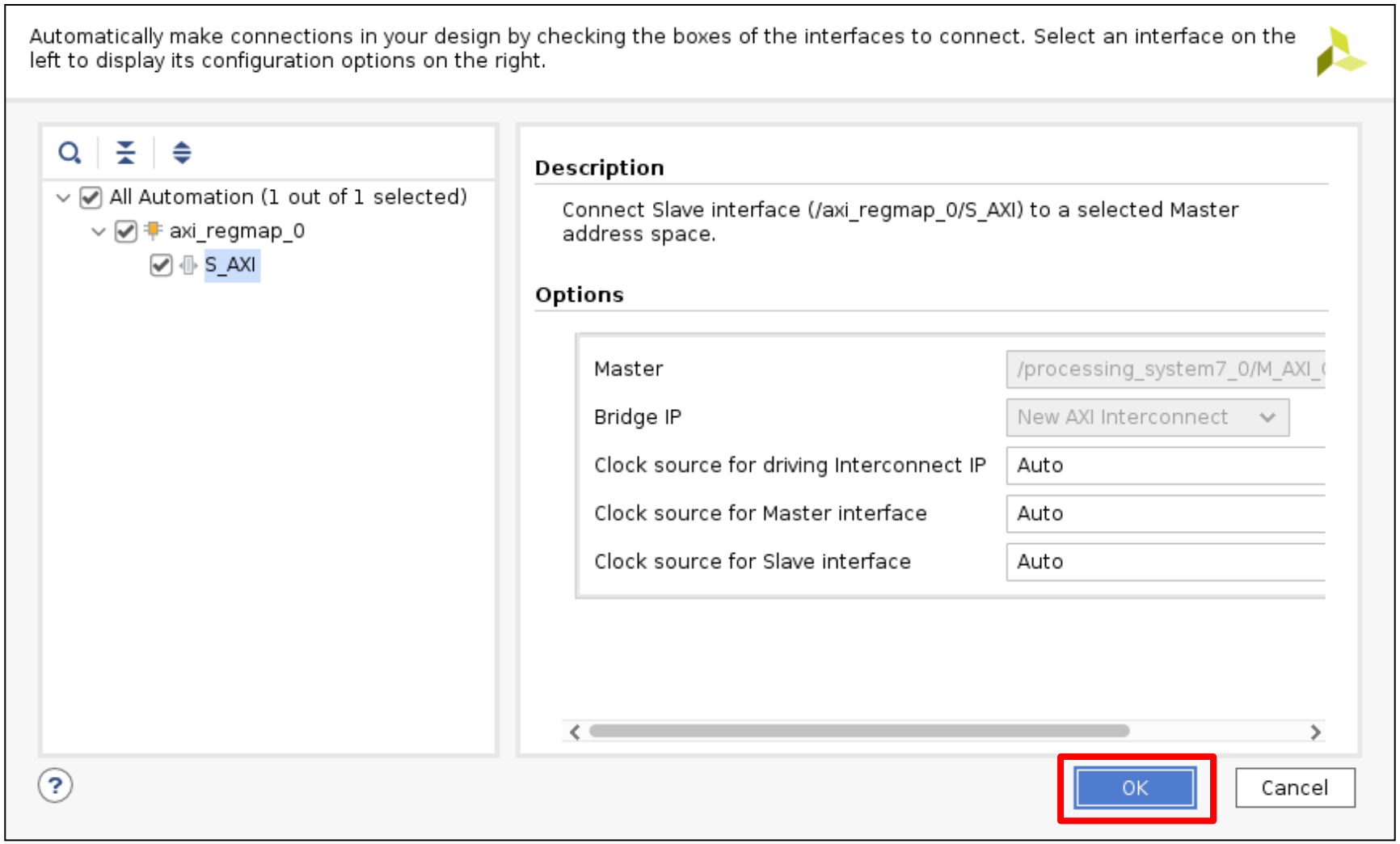


Click ***Run Connection Automation***.

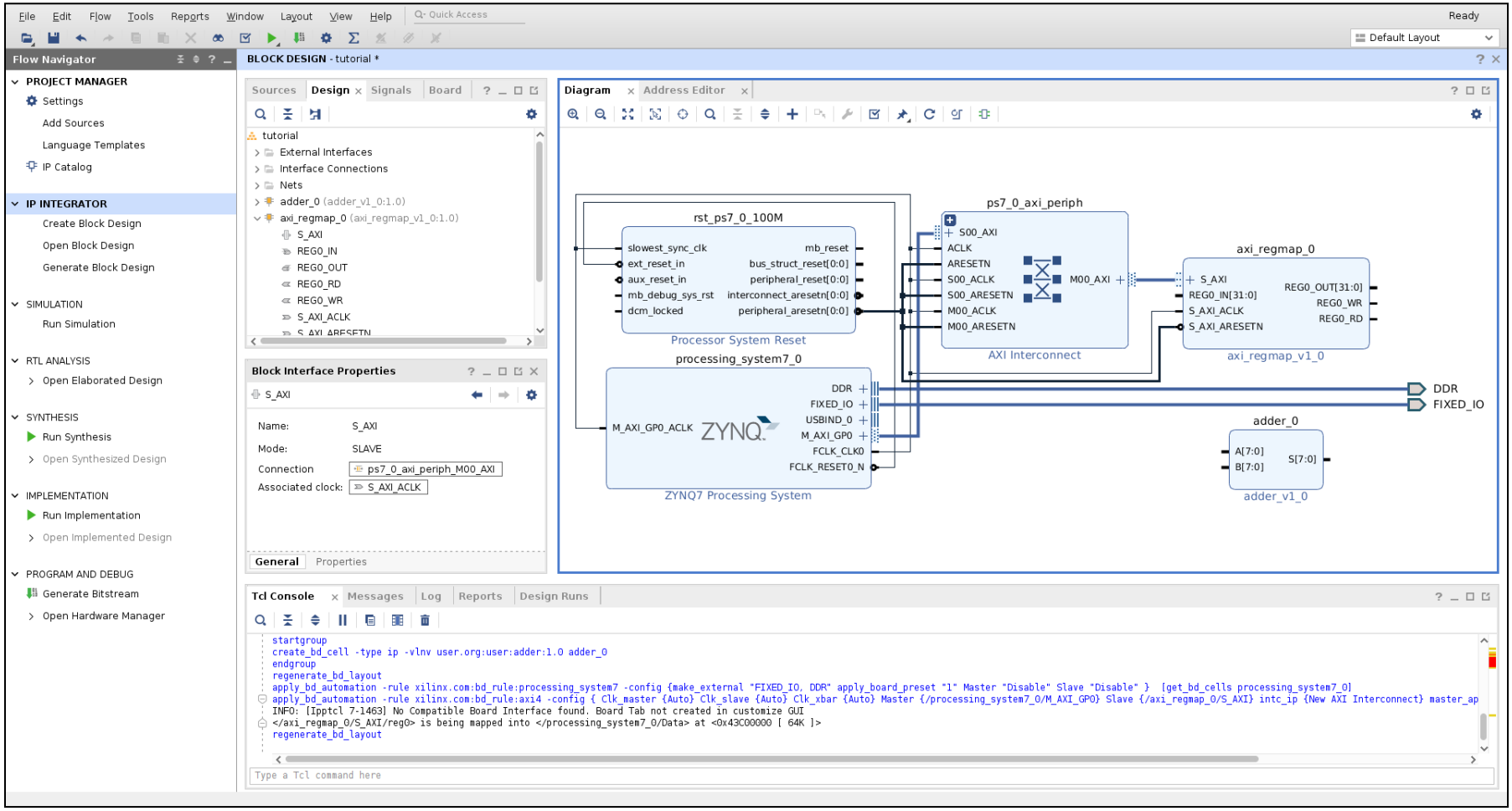
NOTE: This process will automatically instantiate an AXI connect between the **processing_system_0** and **axi_regmap_0** IP.



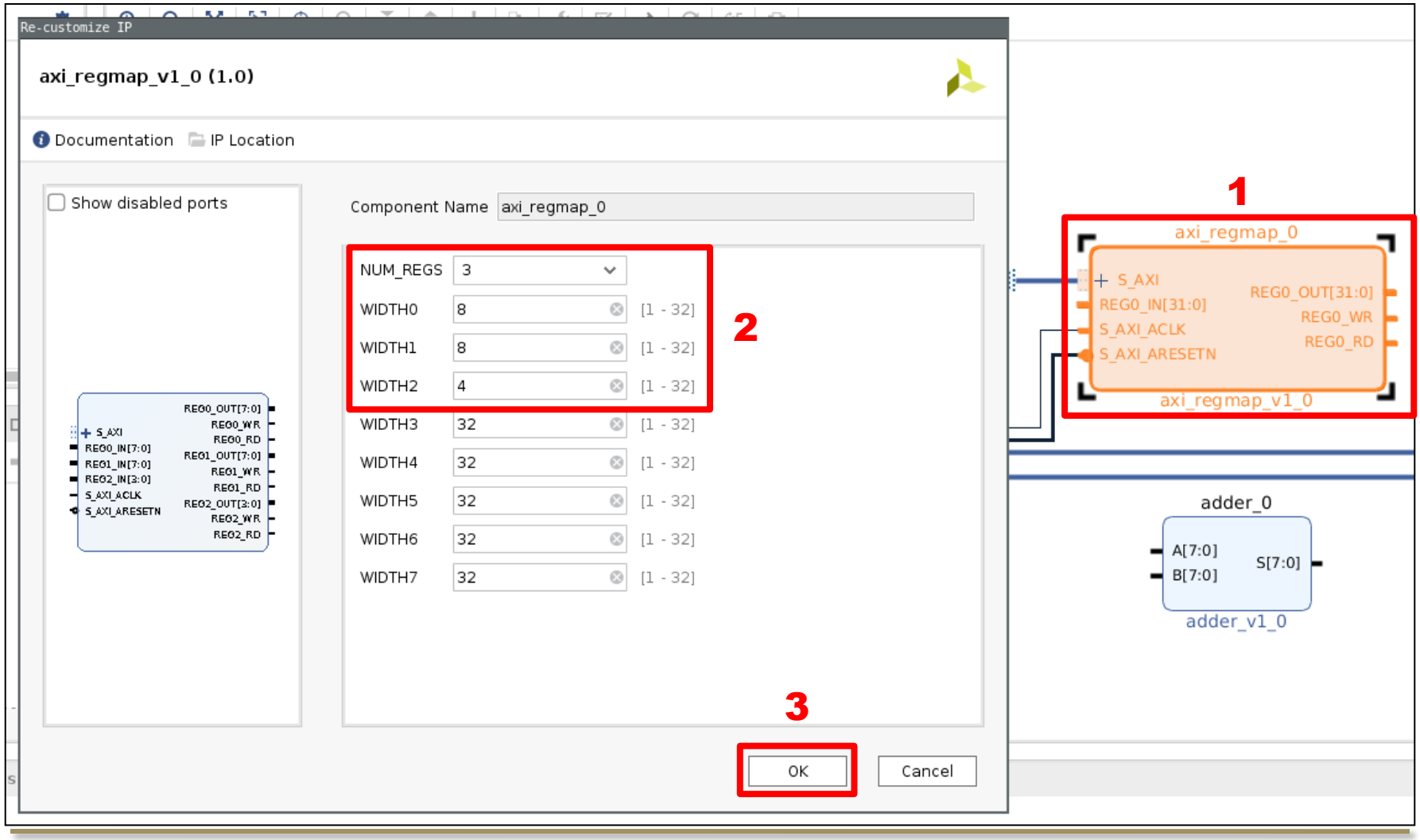
Click ***OK***.



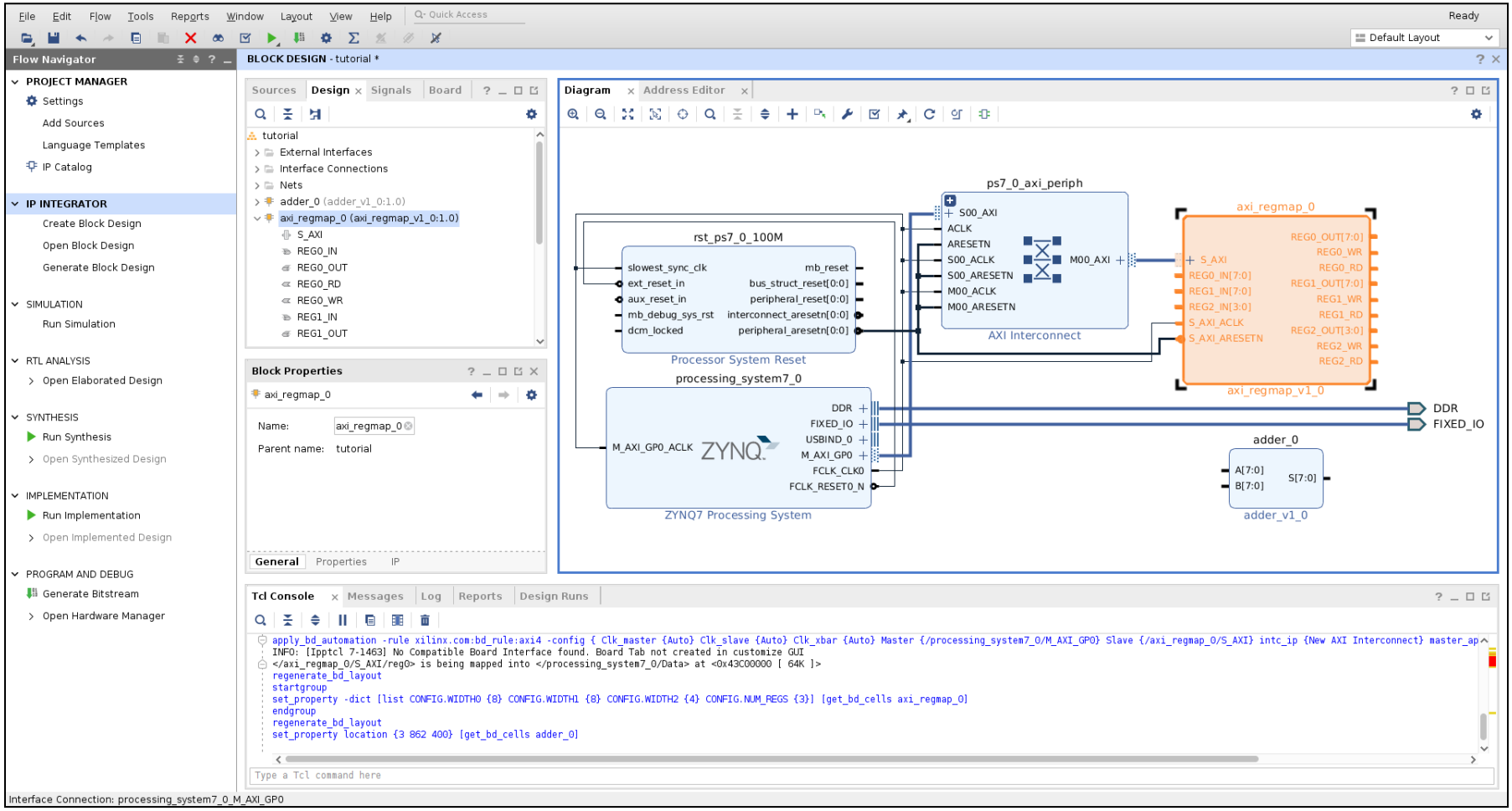
An AXI connection between the **processing_system_0** and **axi_regmap_0** IP was made. Supporting IP (***AXI Interconnect*** and ***Processor System Reset***) were also instantiated. Continue.



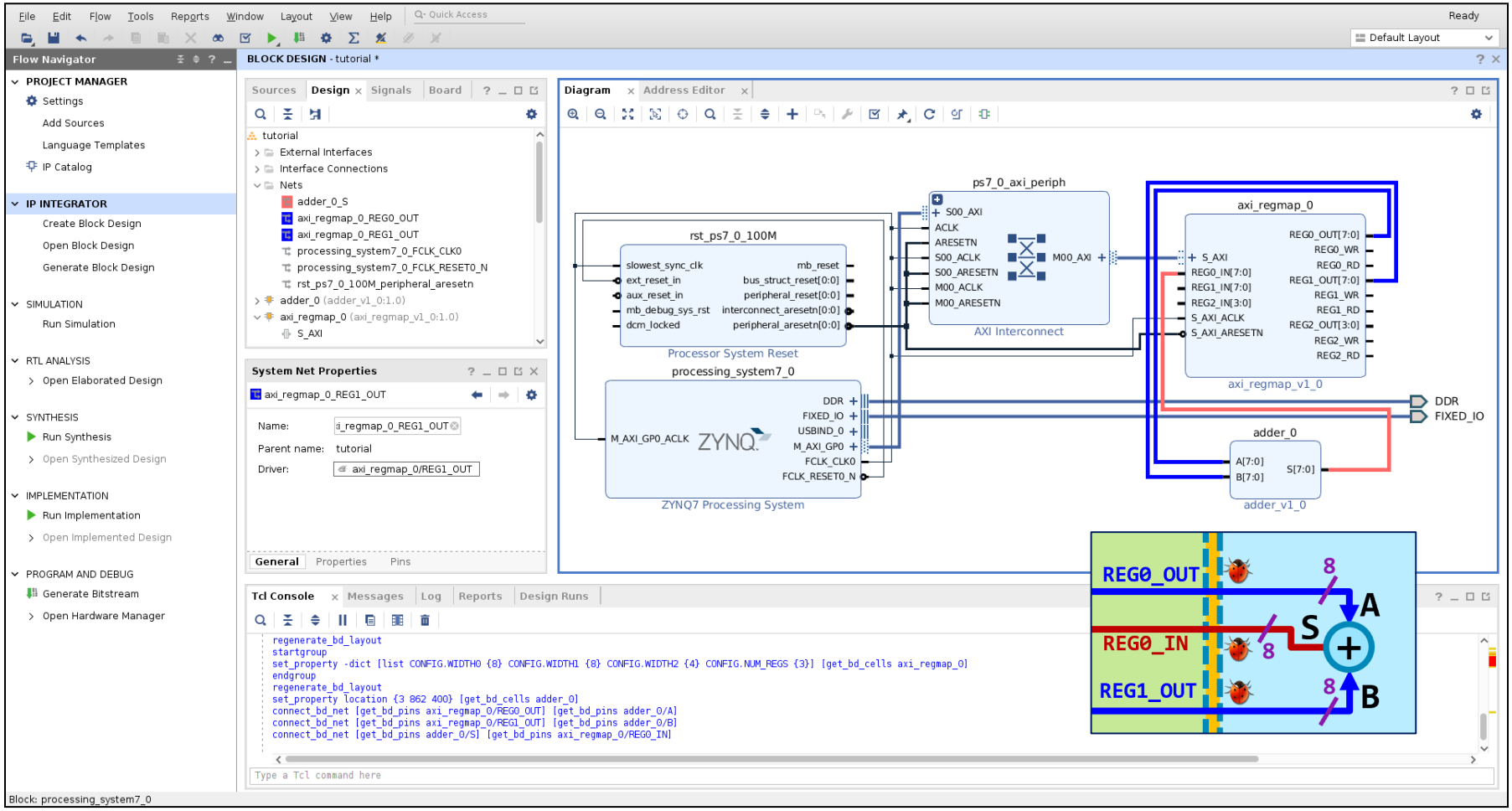
Double-click **axi_regmap_0**.
Set parameters ***NUM_REGS*** → **3**, ***WIDTH0*** → **8**, ***WIDTH1*** → **8**, and ***WIDTH2*** → **4**.
Click ***OK***.



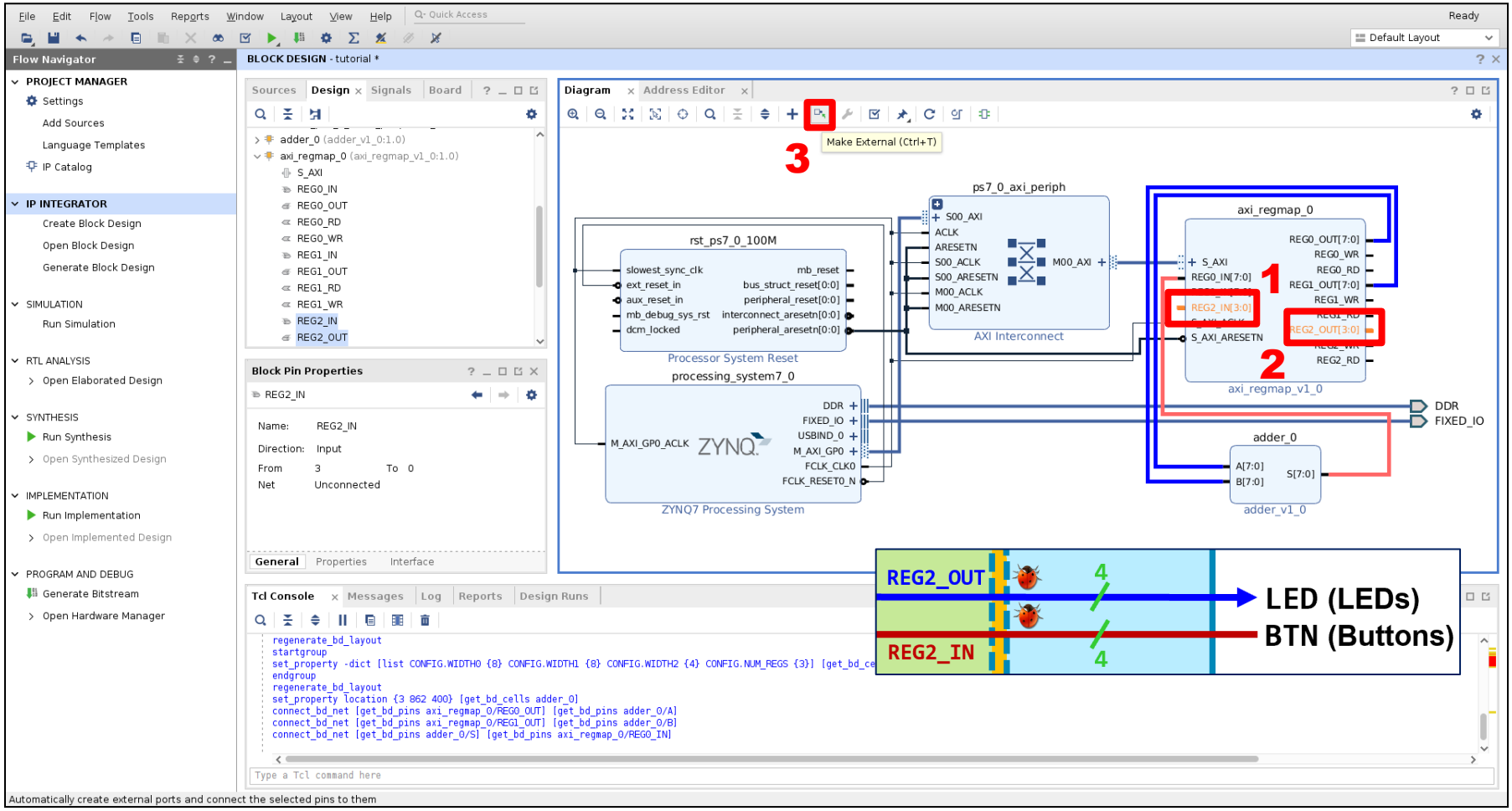
The **axi_regmap_0** IP block has changed. Continue.



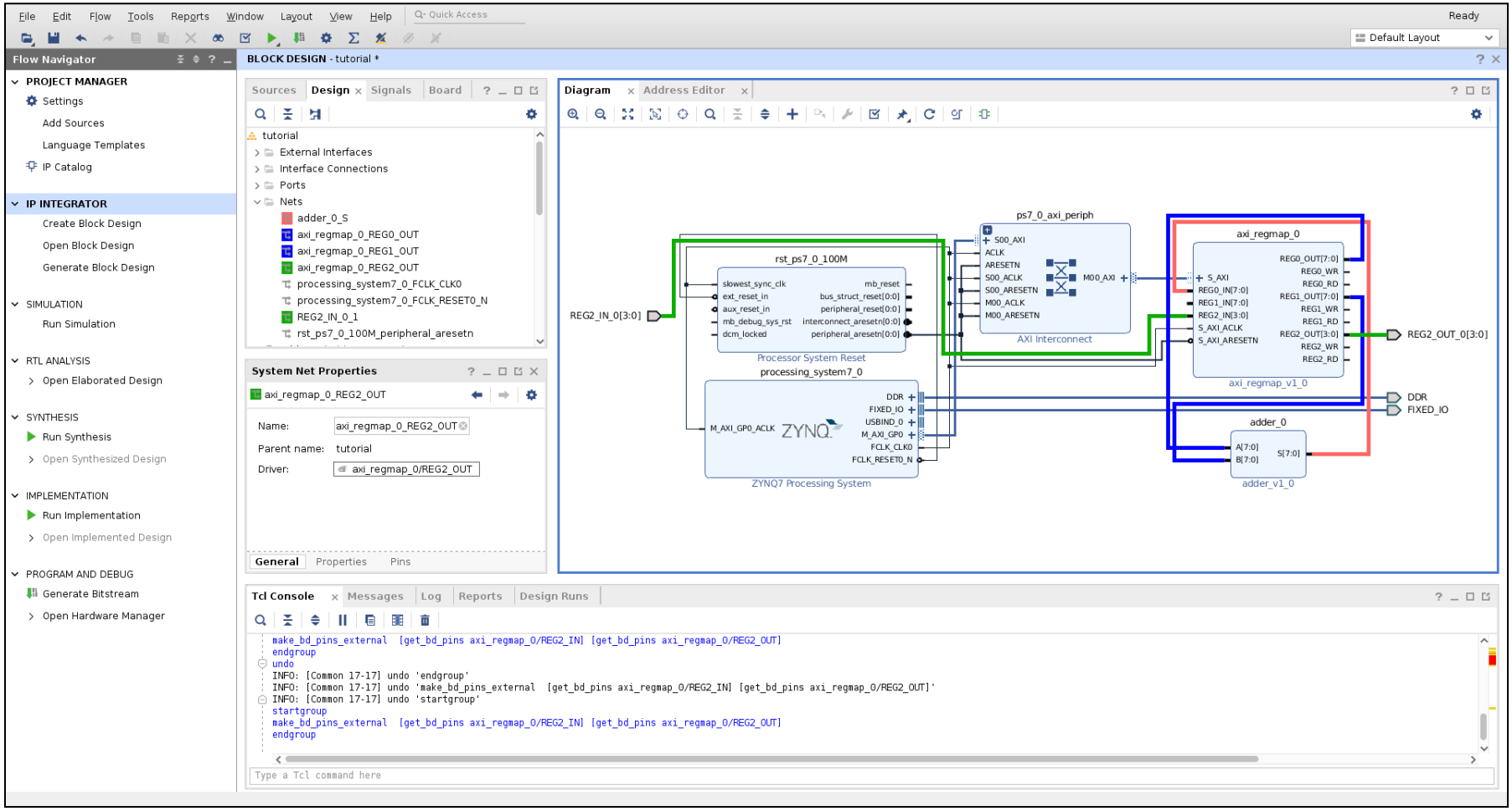
Make the following connections (Hold-click from one pin to another).
REG0_OUT → A (adder), REG1_OUT → B (adder), and **S (adder) → REG0_IN**



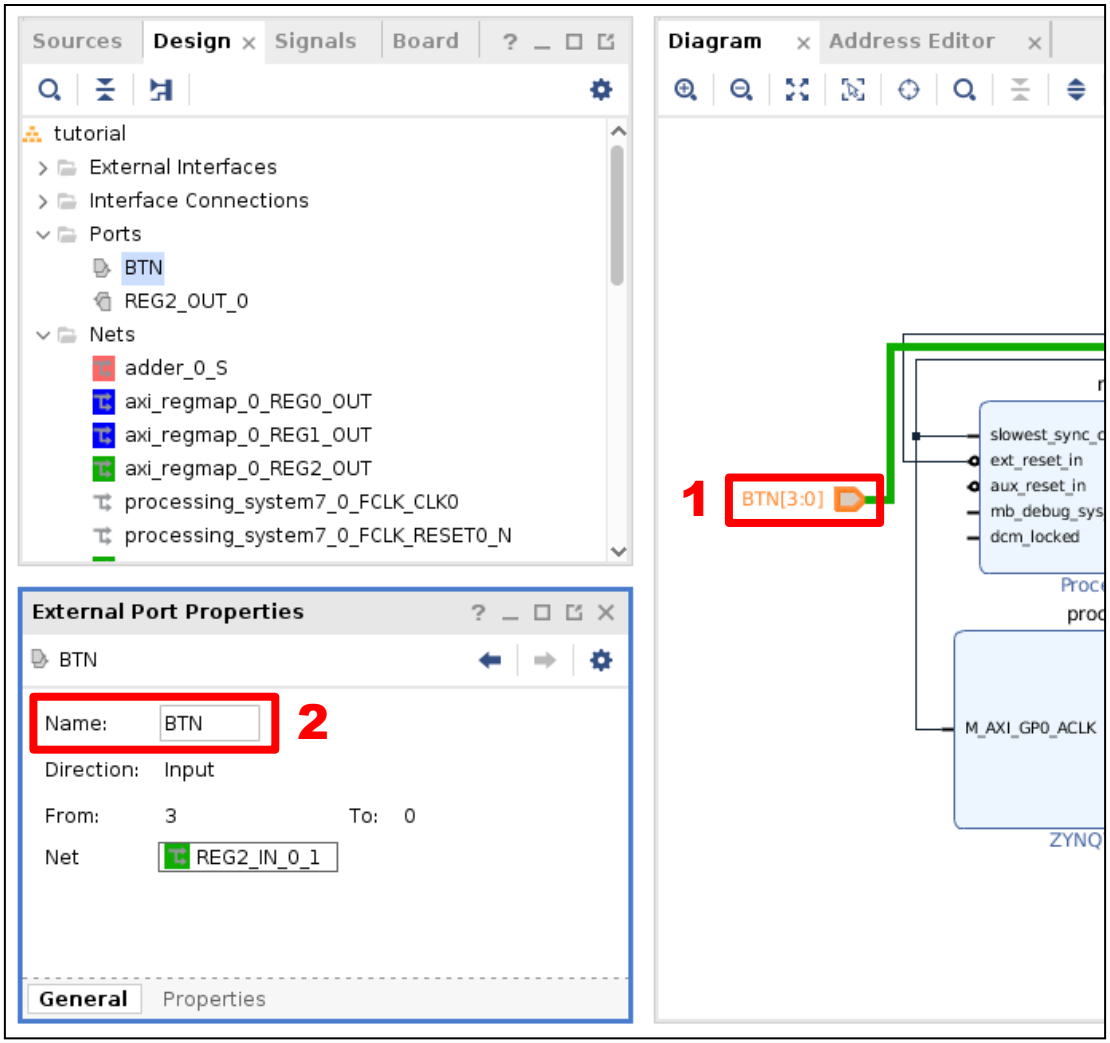
Control-click **REG2_IN** and **REG2_OUT**. Click ***Make External***.



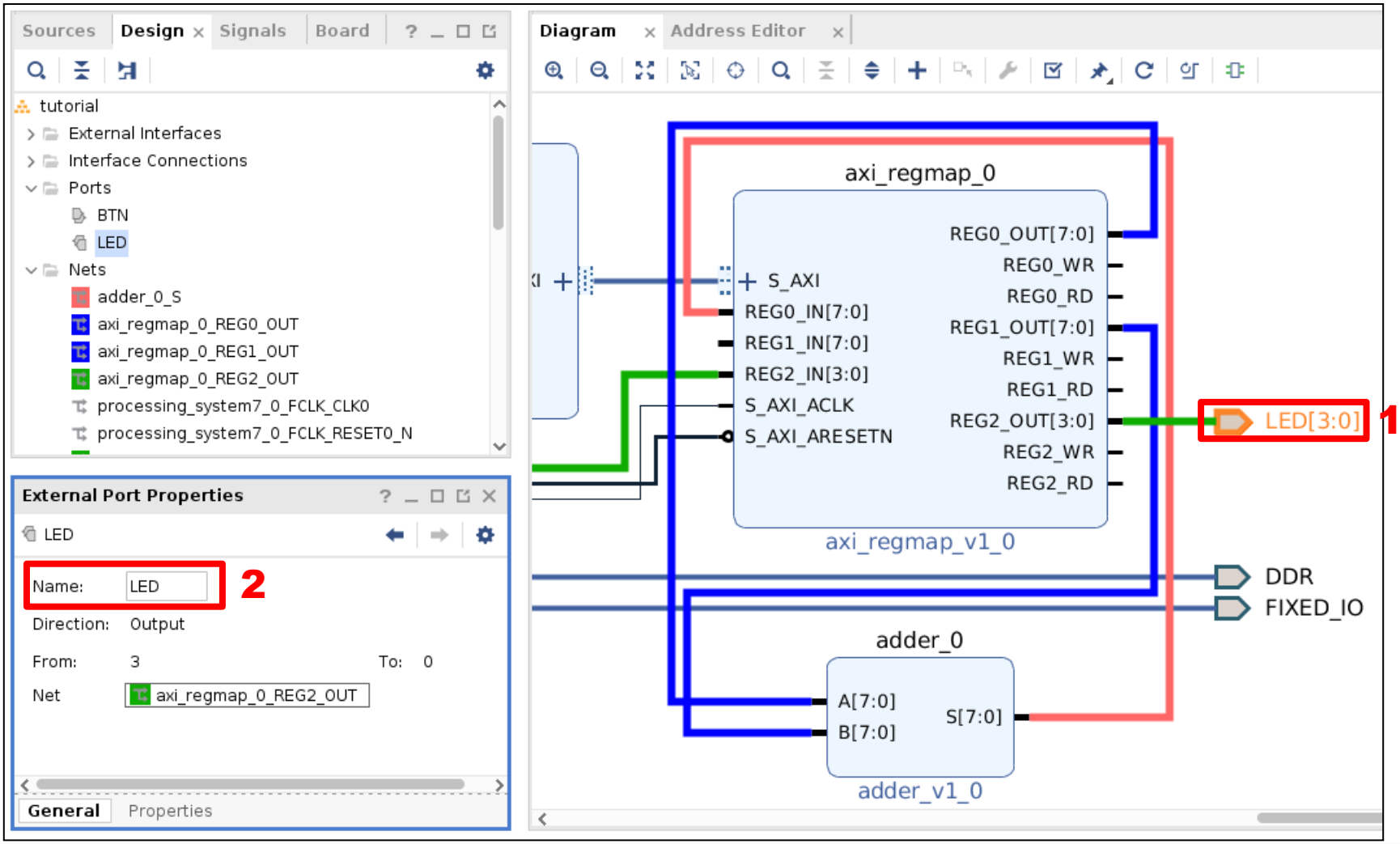
This is the current design. Continue.



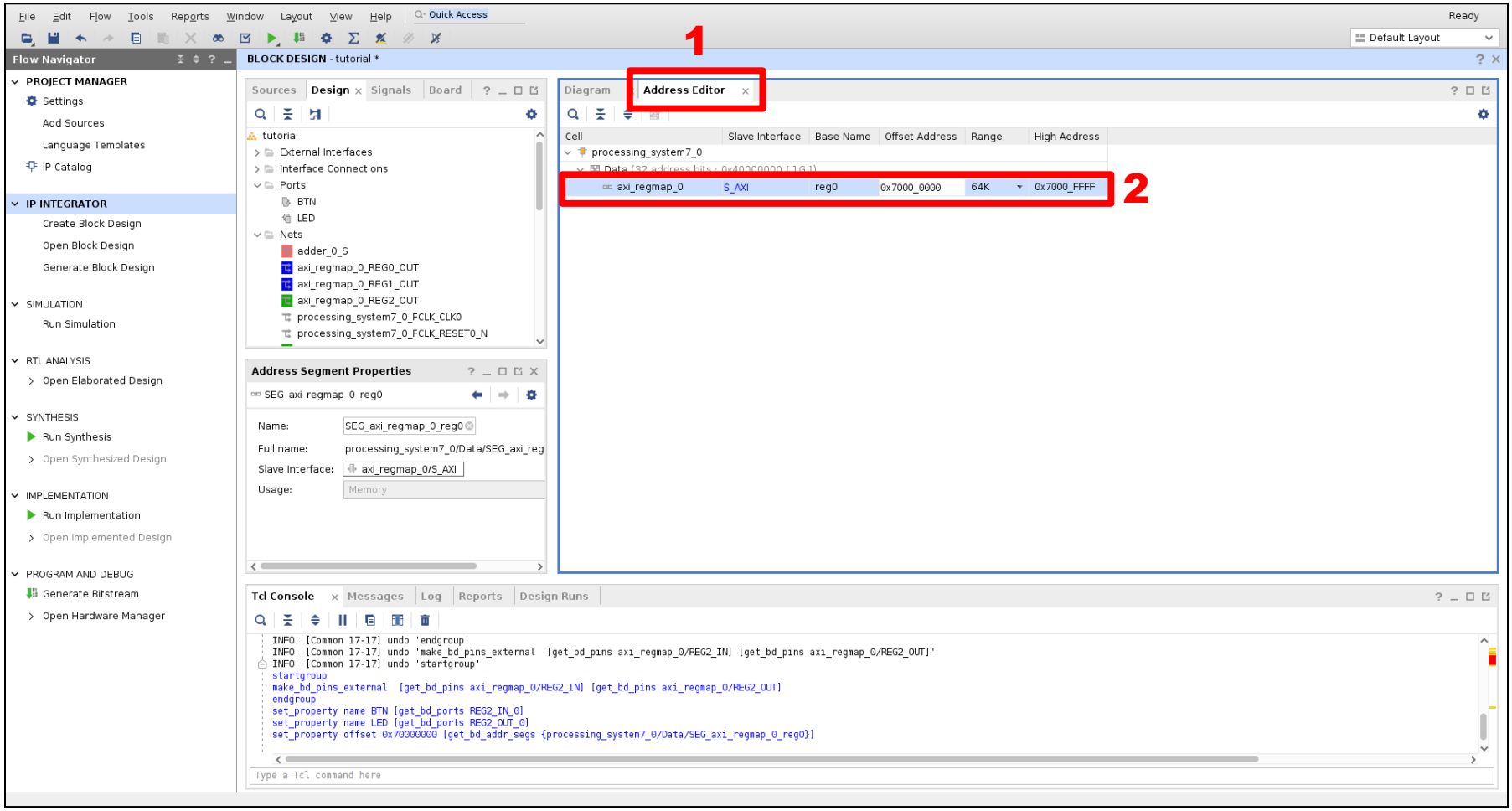
Click **REG2_IN_0[3:0]** (appears as **BTN[3:0]** in the fig). Rename external port to **btn**. **NOTE**: Although VHDL is case insensitive, the XDC (constraints file) is. Make sure that the port name here matches the case in your XDC file.



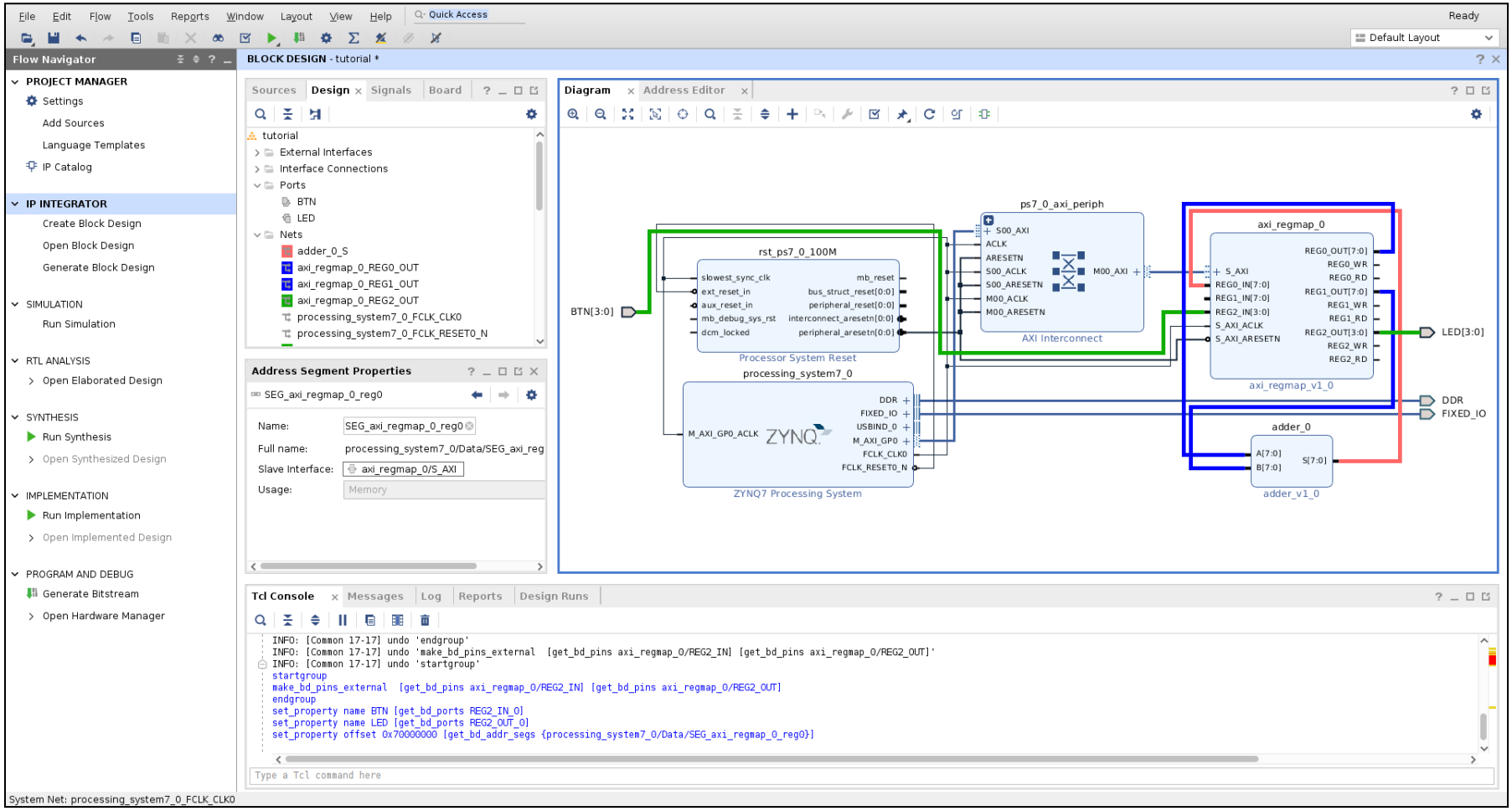
Click **REG2_OUT_0[3:0]** (appears as **LED[3:0]** in the pic). Rename external port to **led**. **NOTE**: Although VHDL is case insensitive, the XDC (constraints file) is. Make sure that the port name here matches the case in your XDC file.



Click ***Address Editor***. Enter offset address **`0x7000_0000`** for **axi_regmap_0**.



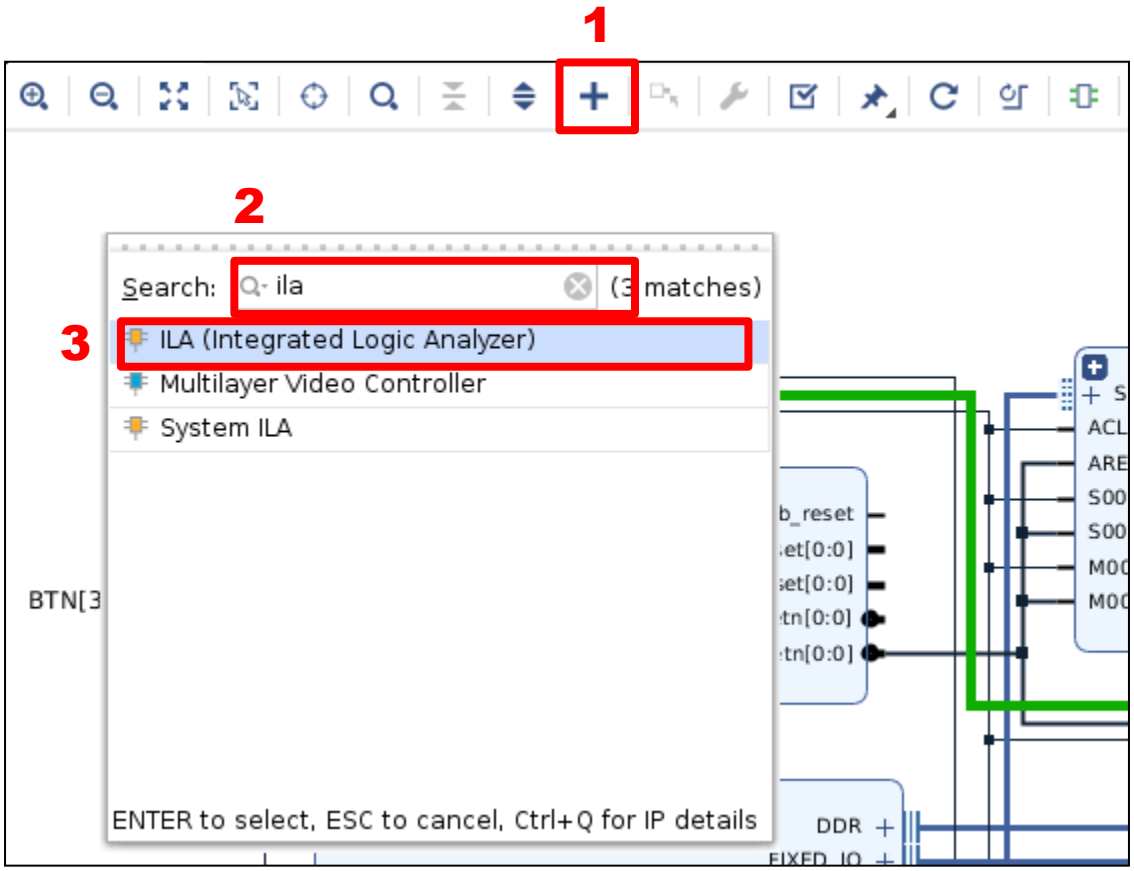
This is the current design. Continue.



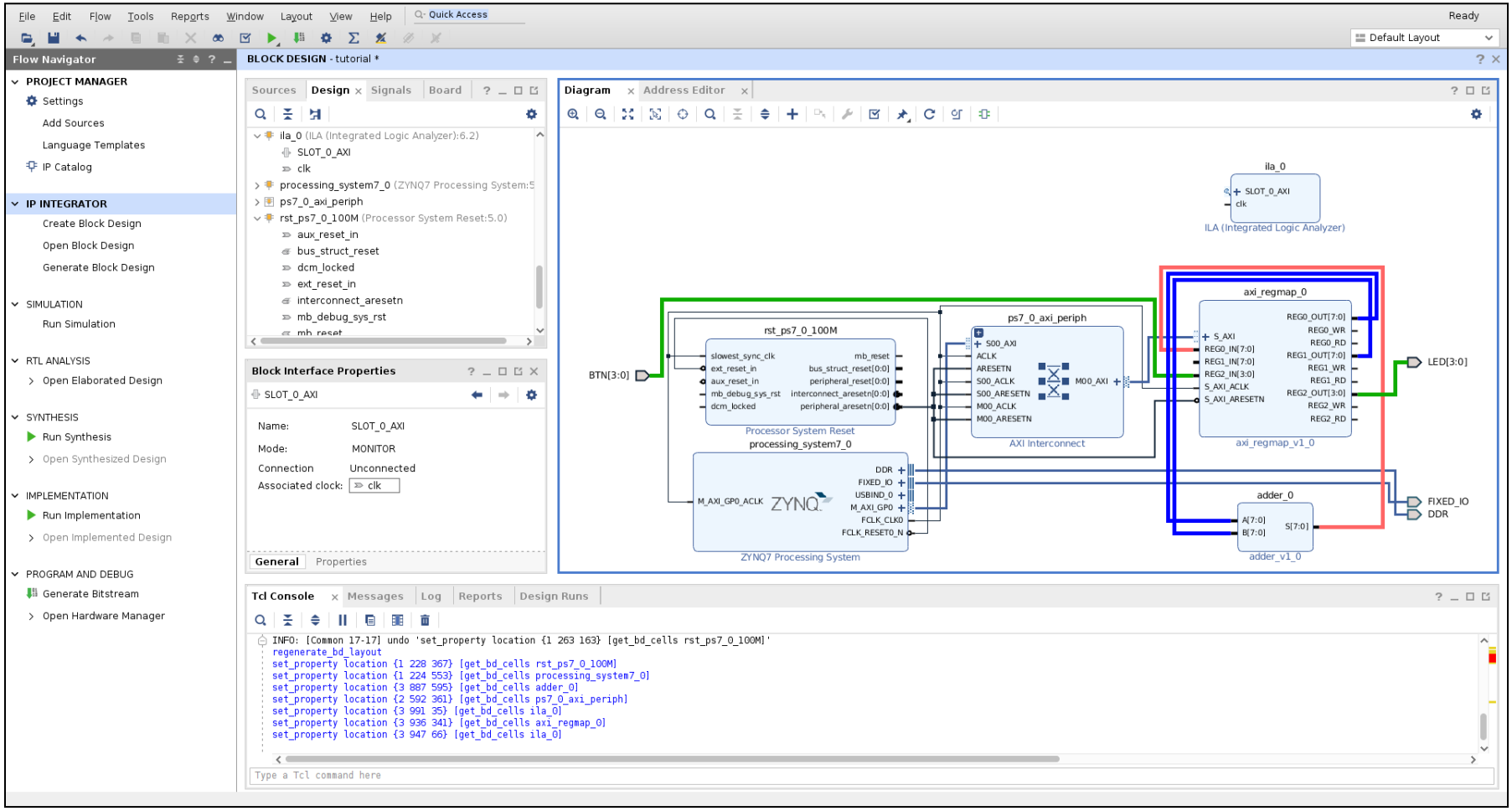
Part 5: Adding ILA

Next, we will add an Integrated Logic Analyzer (ILA) to probe the signals needed.

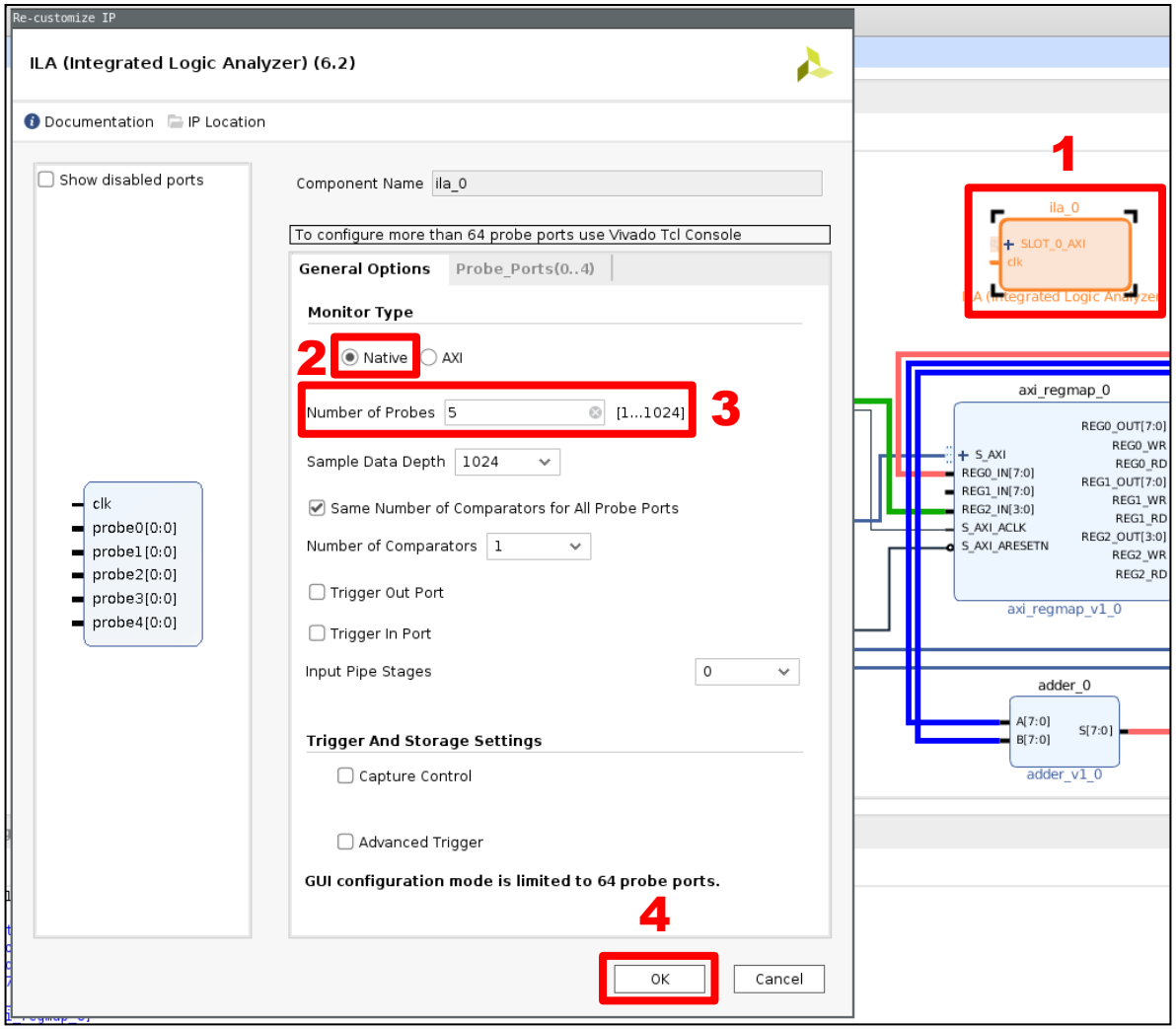
Click **+ *(Add IP)***. Search ***ila***. Double-click ***ILA (Integrated Logic Analyzer)***.
NOTE: The ILA is a useful debug tool for analyzing internal signals while the device and design are running.



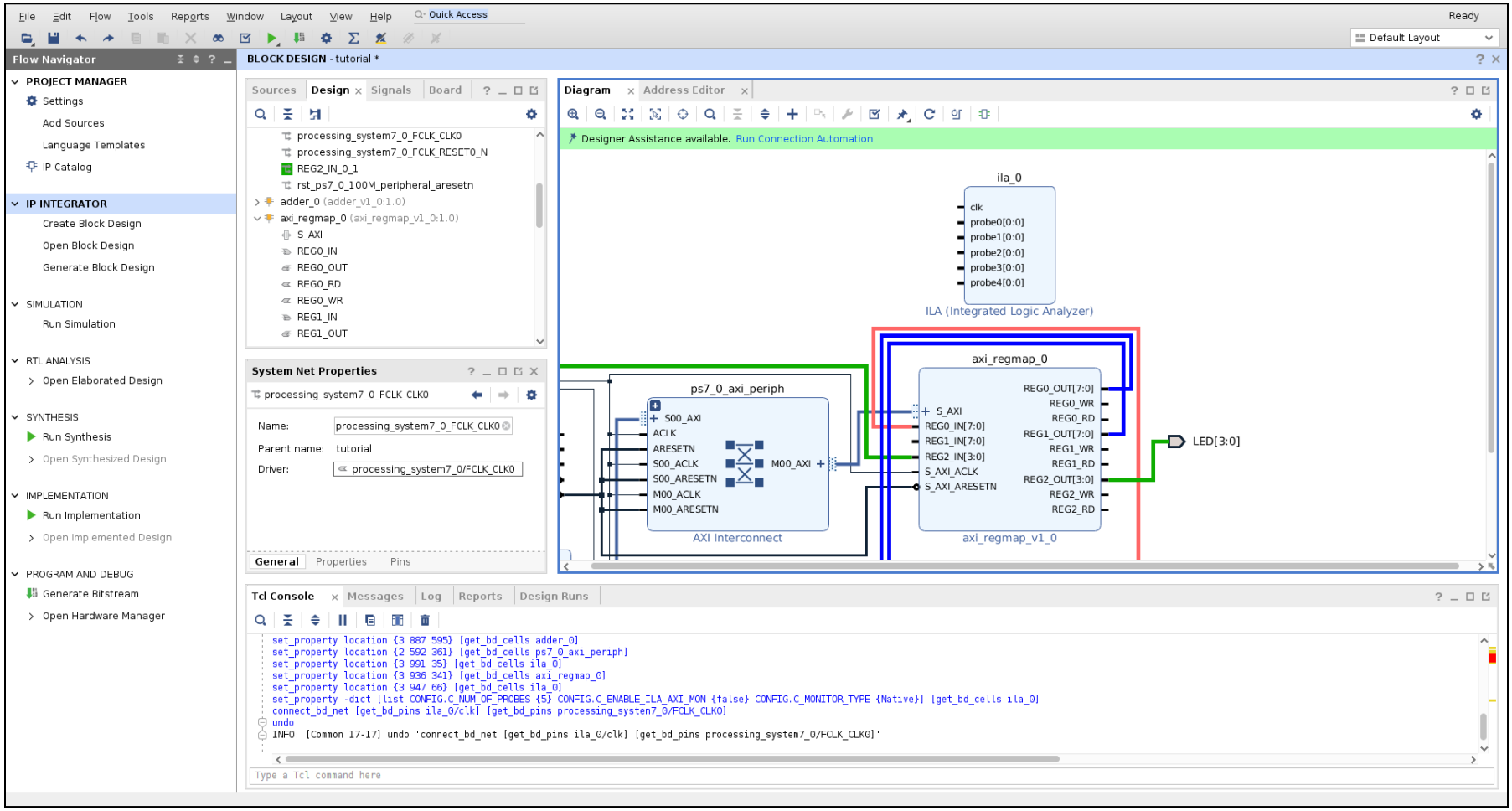
Wild ***ILA (Integrated Logic Analyzer)*** appeared! The instance name is **ila_0**. Continue.



Double-click **ila_0**. Set parameters ***Native*** and ***Number of Probes→5***. Click ***OK***.

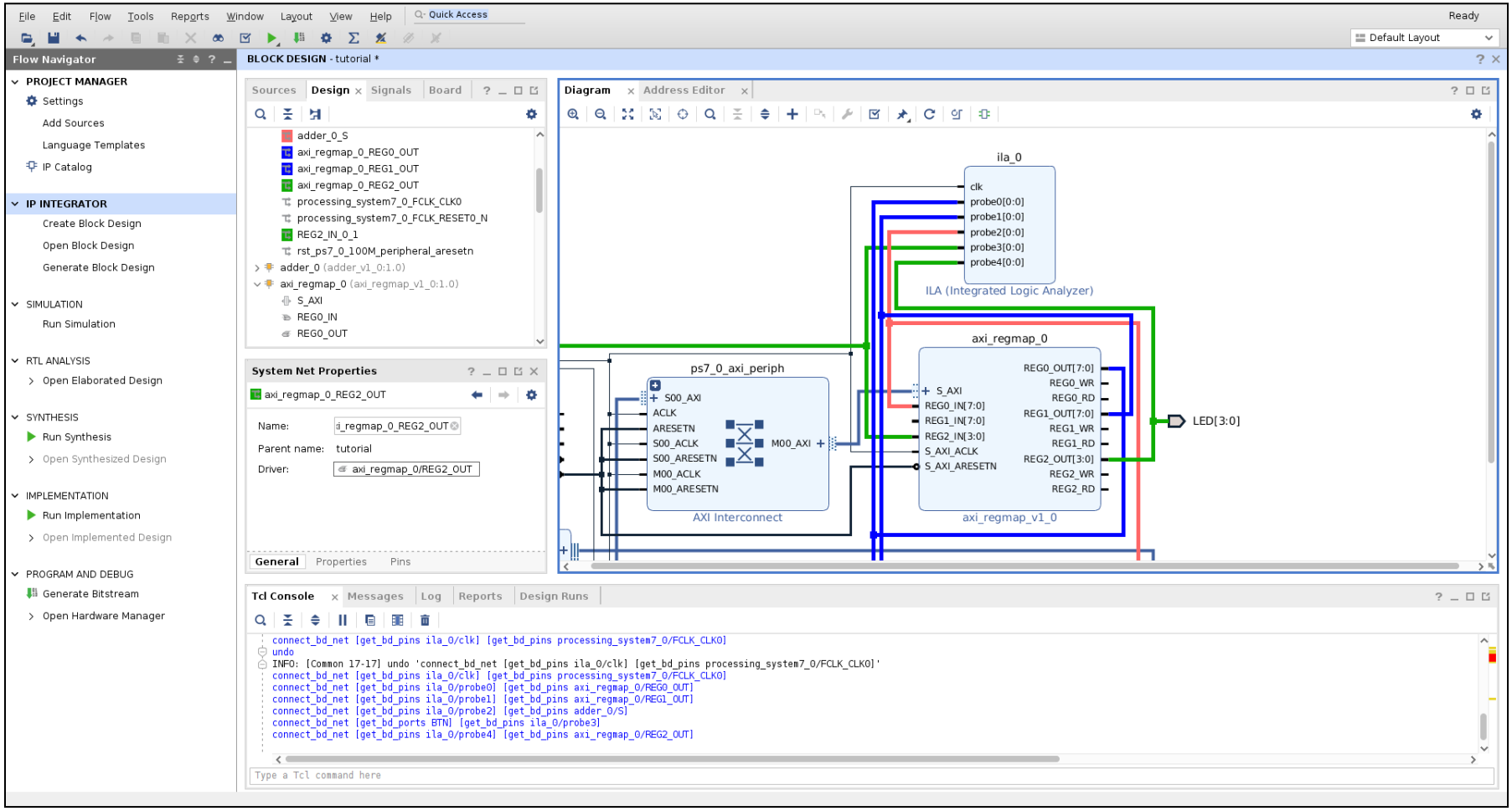


The **ila_0** IP block has changed. Continue.



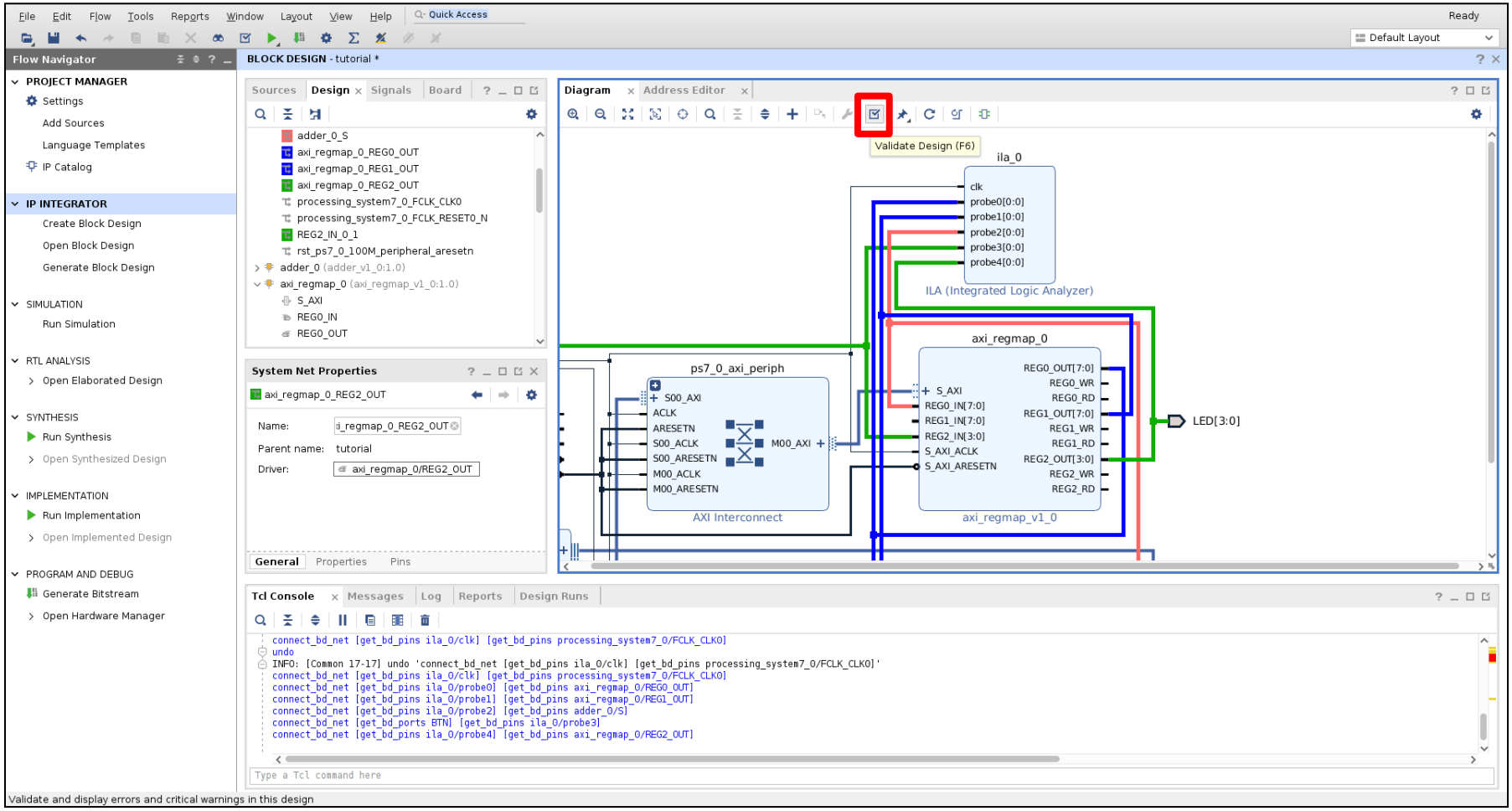
Connect the **ila_0 clk** port to the global clock signal (same as **S_AXI_ACLK** on **axi_regmap_0**). Connect the **ila_0** probes to the target nets.

NOTE: The order of probes does not matter; the probes are renamed to the connected net name.



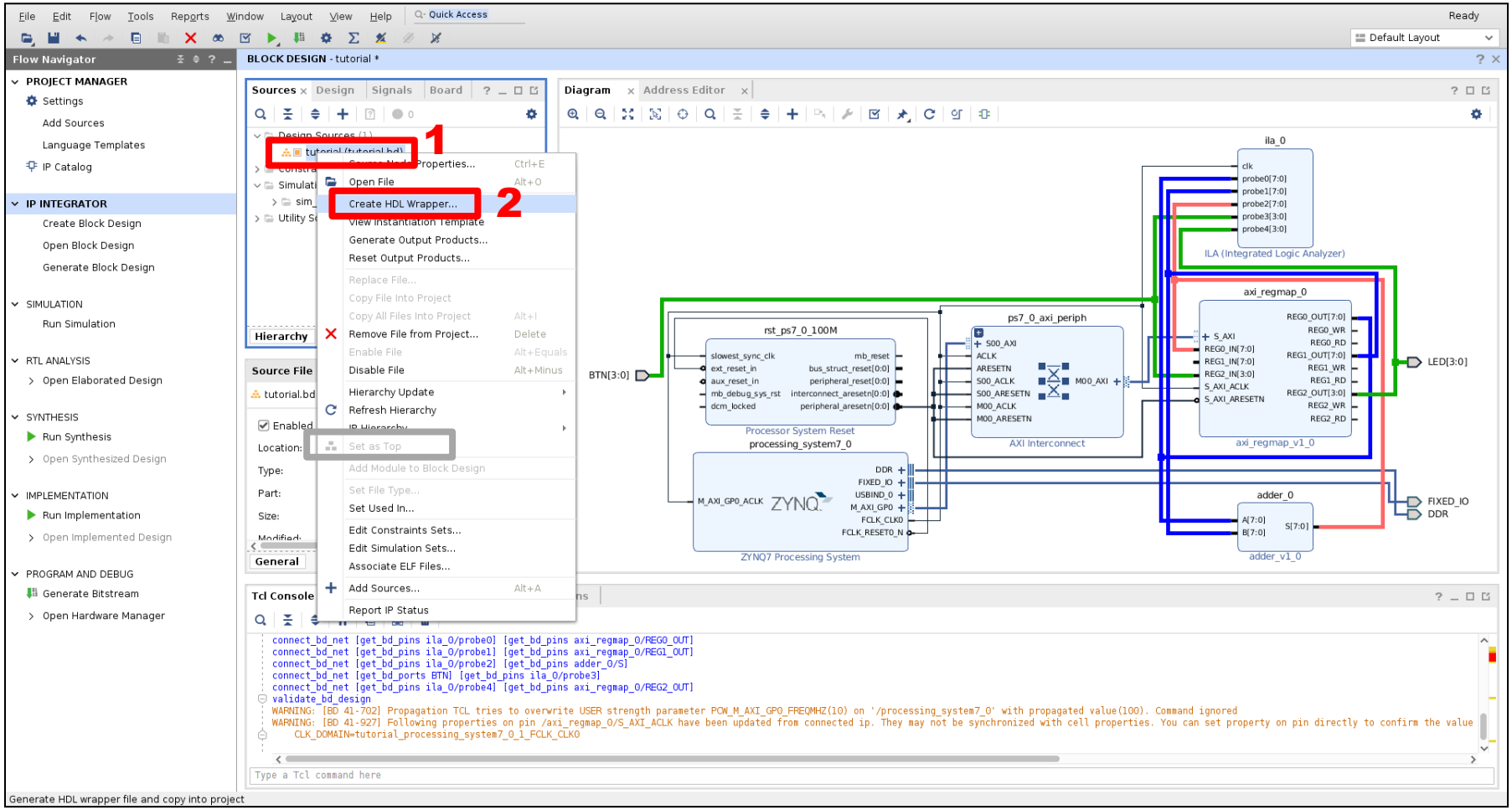
Click ***Validate Design***.

NOTE: This will report design errors and propagate parameters throughout the design.

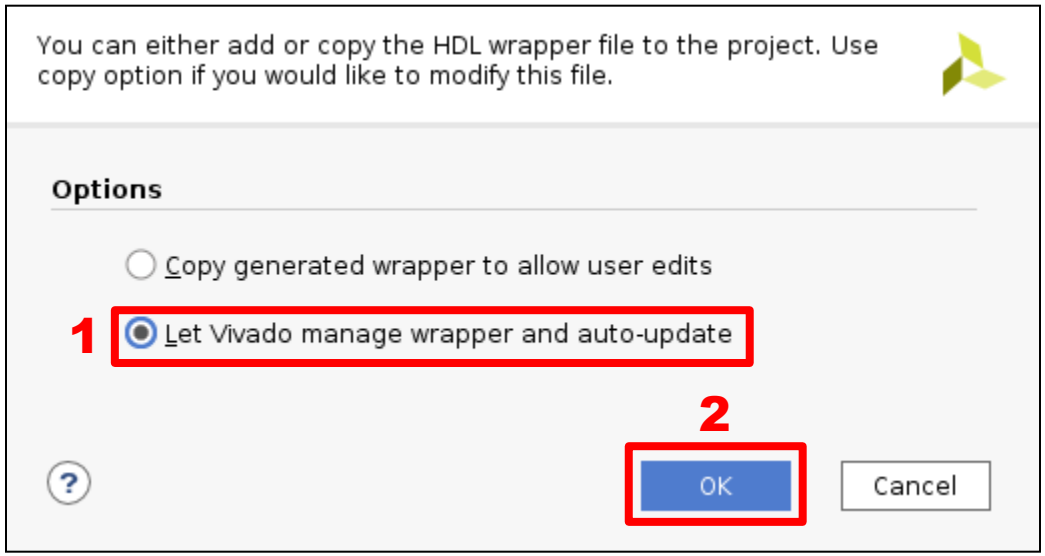


Click ***OK***.

Right-click the block design **Lab_1**. Click ***Create HDL Wrapper…***.
NOTE: This process will generate a VHDL wrapper for the block design. You can think of this as the top design that we will be synthesizing and downloading on the FPGA.
NOTE: make sure the block design is set as Top (the grey box in the picture below).



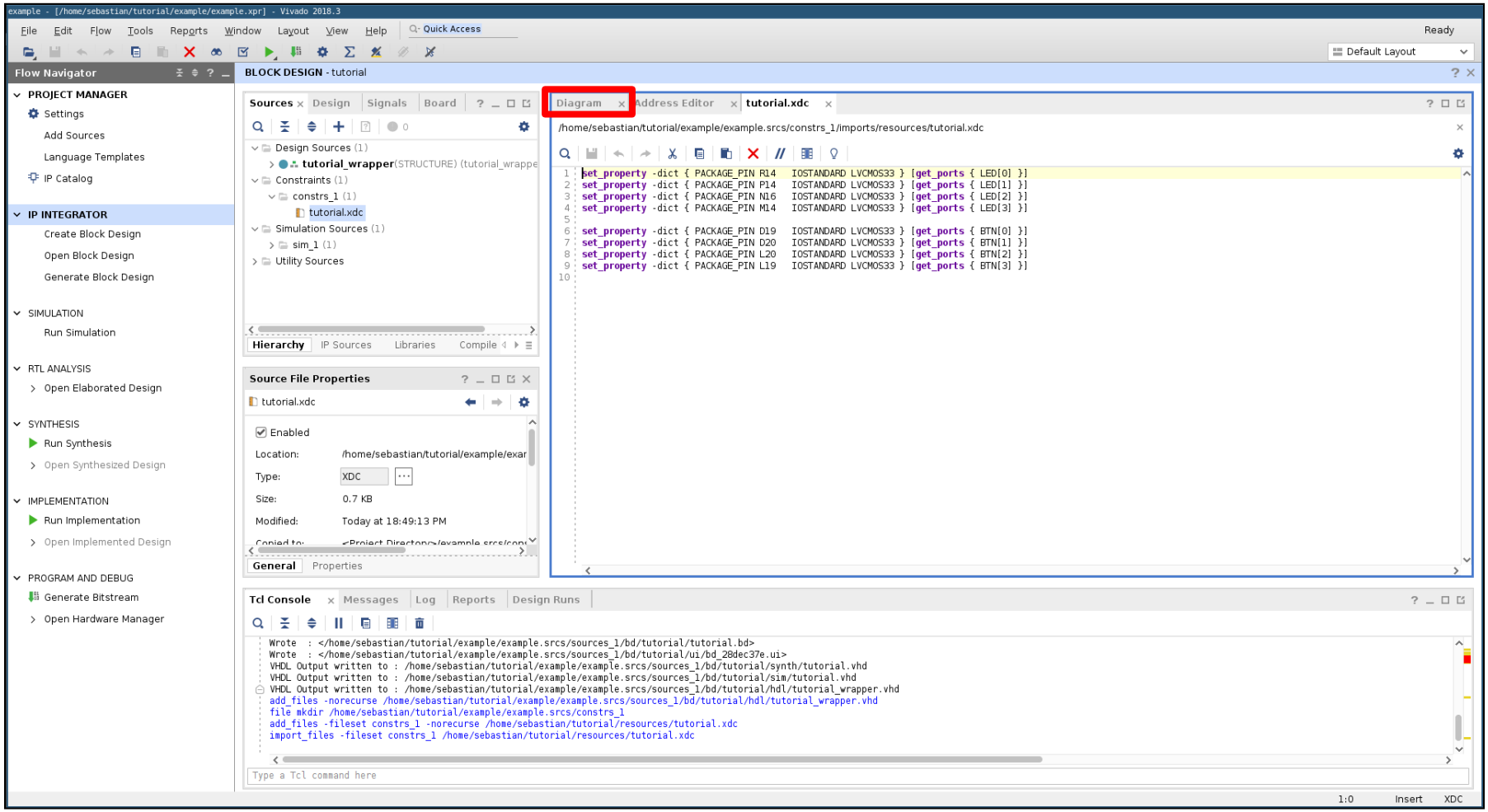
Select ***Let Vivado manage wrapper and auto-update***. Click ***OK***.



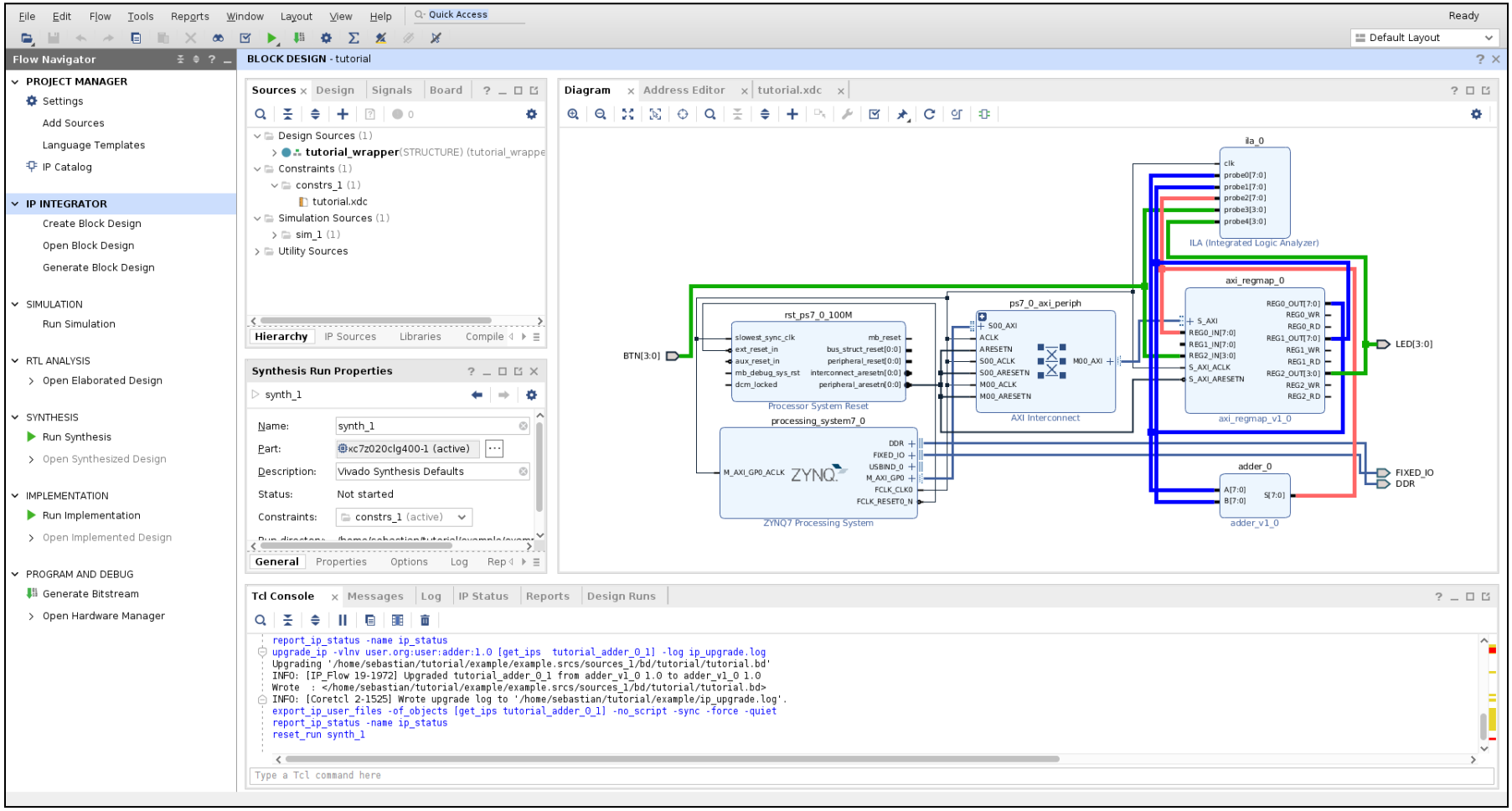
PYNQ-Z1_C.xdc. FPGA **package pins** and **IO standard (type and voltage)** are assigned to indexed bits of the **BTN and LED port vectors**. Continue.
Note: port names must match external ports of the block design.

```
set_property -dict { PACKAGE_PIN R14   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }]
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }]
set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 } [get_ports { LED[2] }]
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { LED[3] }]

set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports { BTN[0] }]
set_property -dict { PACKAGE_PIN D20   IOSTANDARD LVCMOS33 } [get_ports { BTN[1] }]
set_property -dict { PACKAGE_PIN L20   IOSTANDARD LVCMOS33 } [get_ports { BTN[2] }]
set_property -dict { PACKAGE_PIN L19   IOSTANDARD LVCMOS33 } [get_ports { BTN[3] }]
```

Click the ***Diagram*** tab.



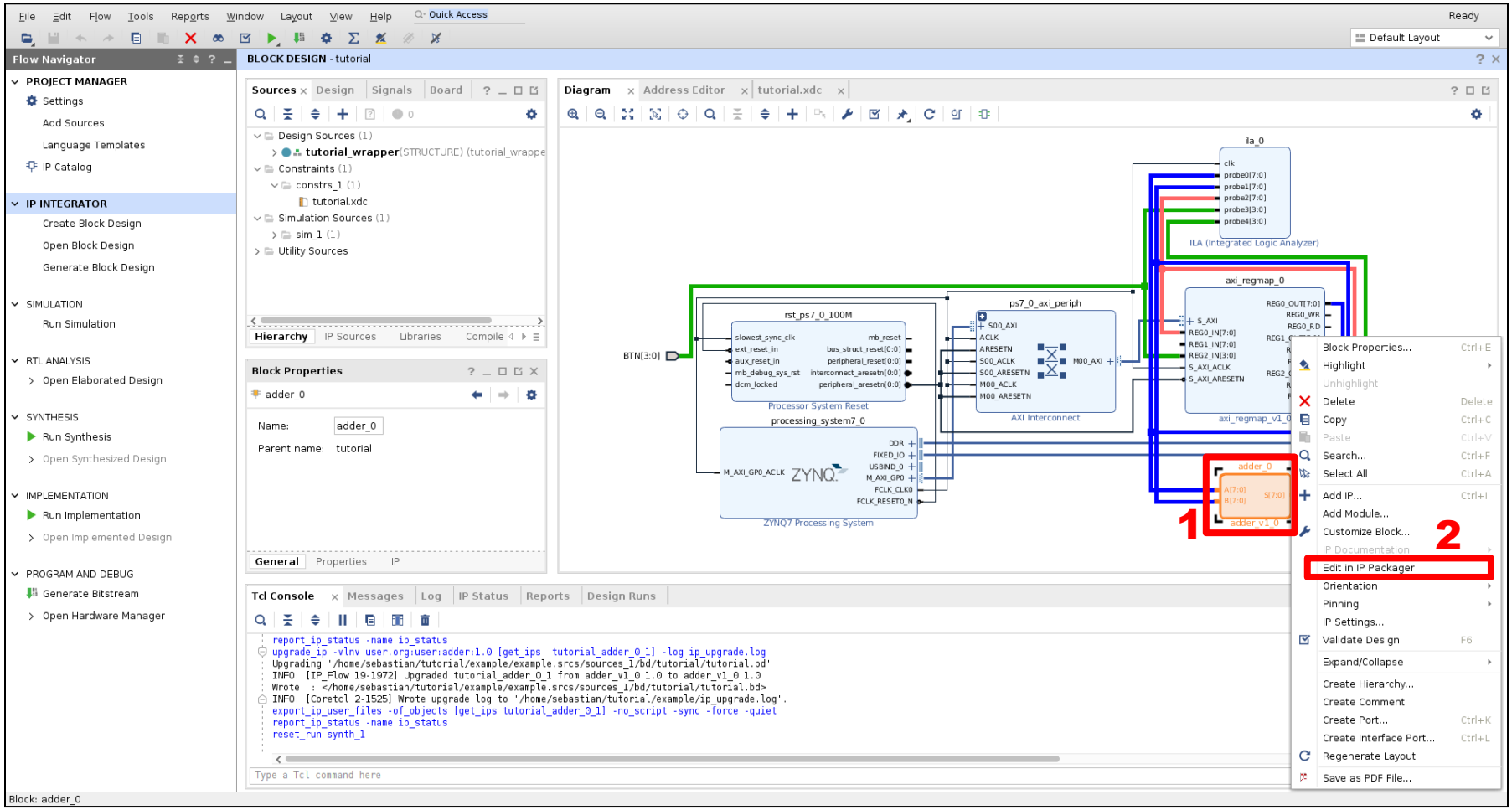
This is the current design. Continue.



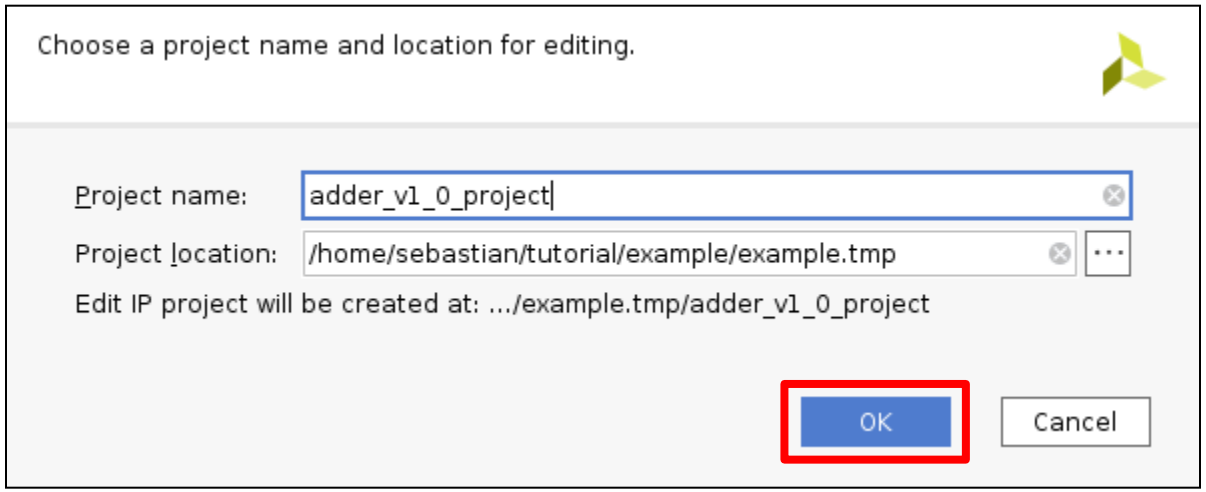
Part 5: Editing an IP

If the RTL (e.g., VHDL) of an IP must be modified, the Vivado IP packager must be used to re-package the IP. Let us suppose that the **adder** IP must be edited.

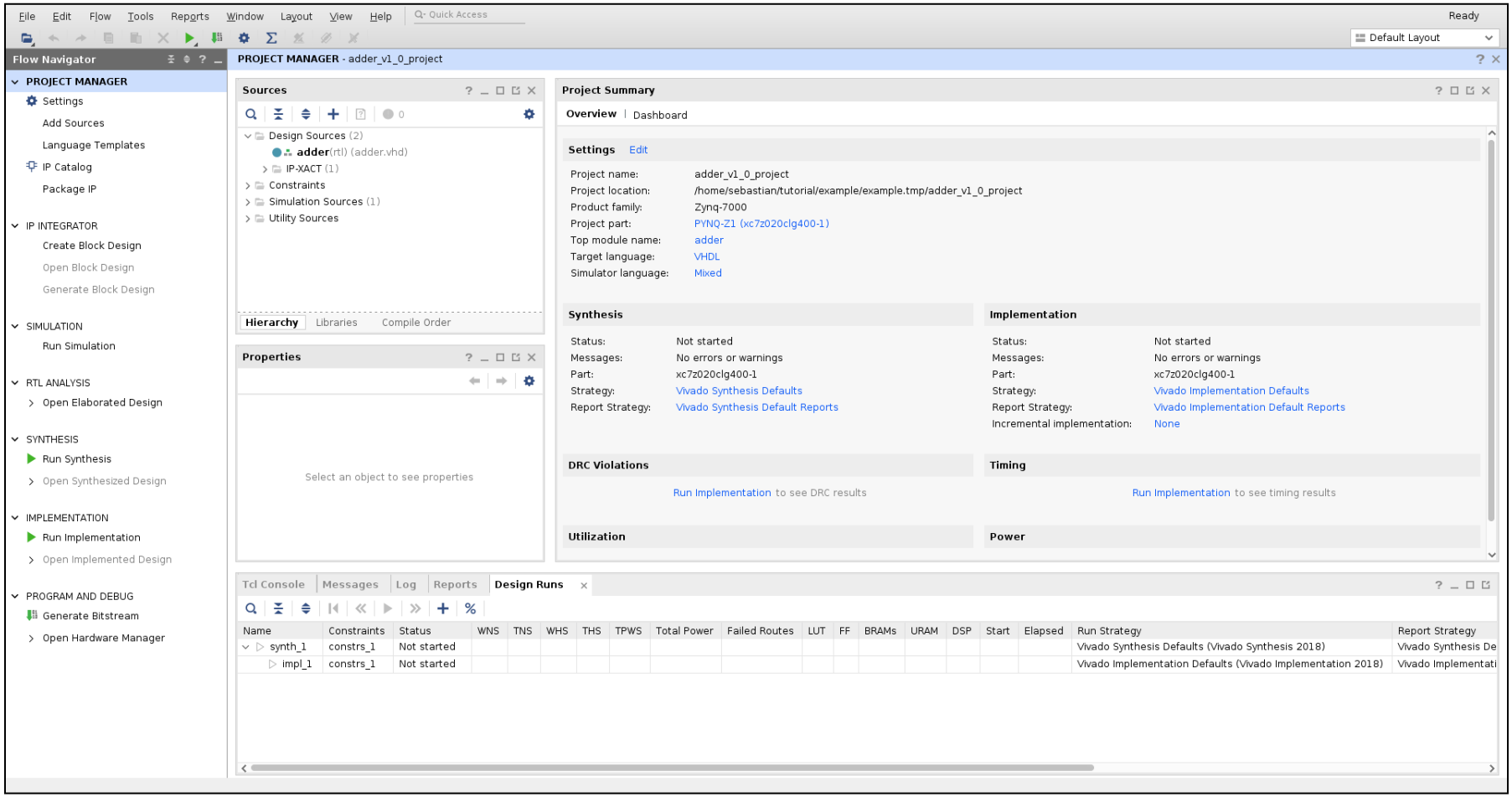
Right-click the **adder_0** IP block. Click ***Edit in IP Packager***.



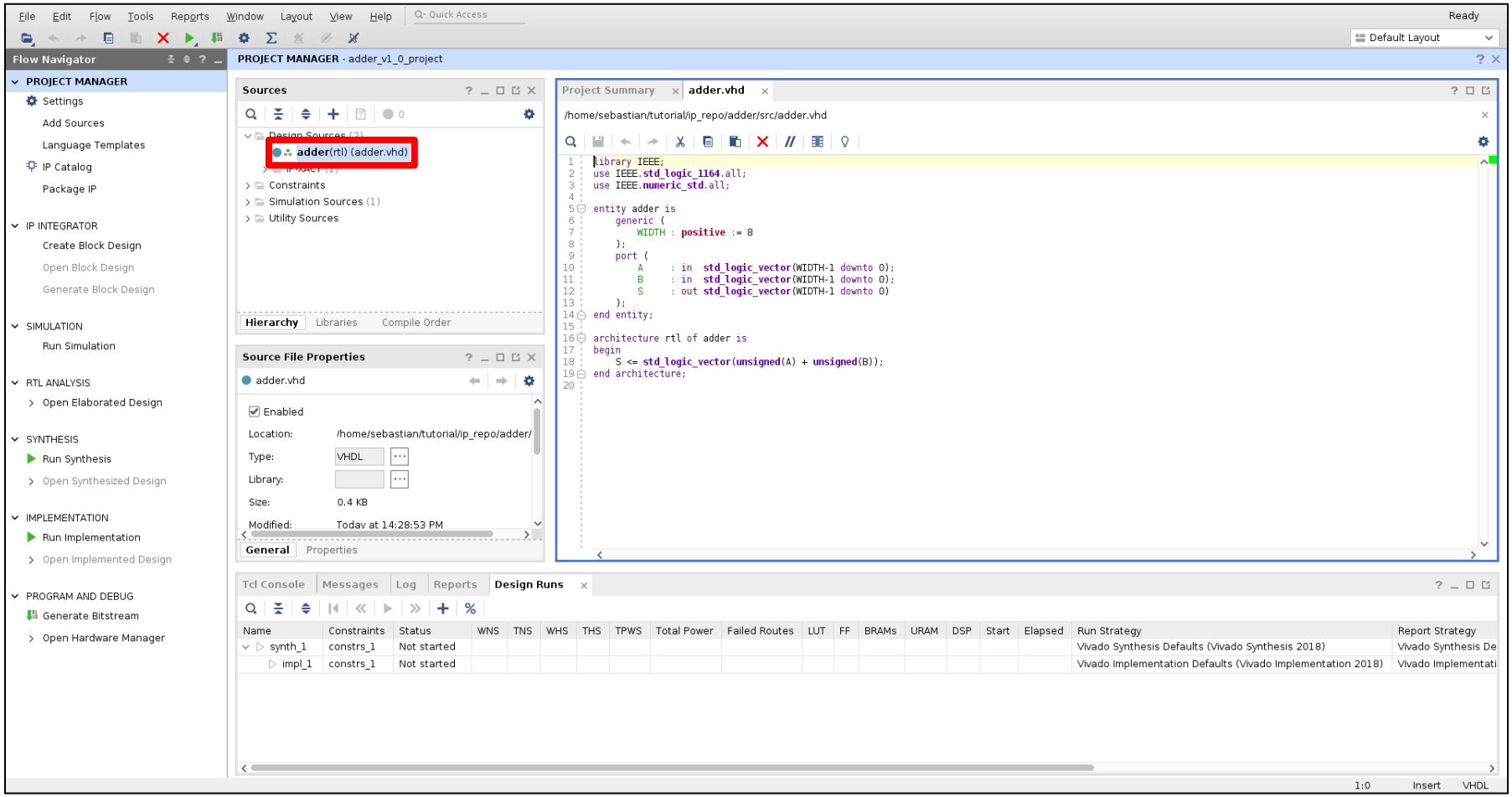
Click ***OK***.



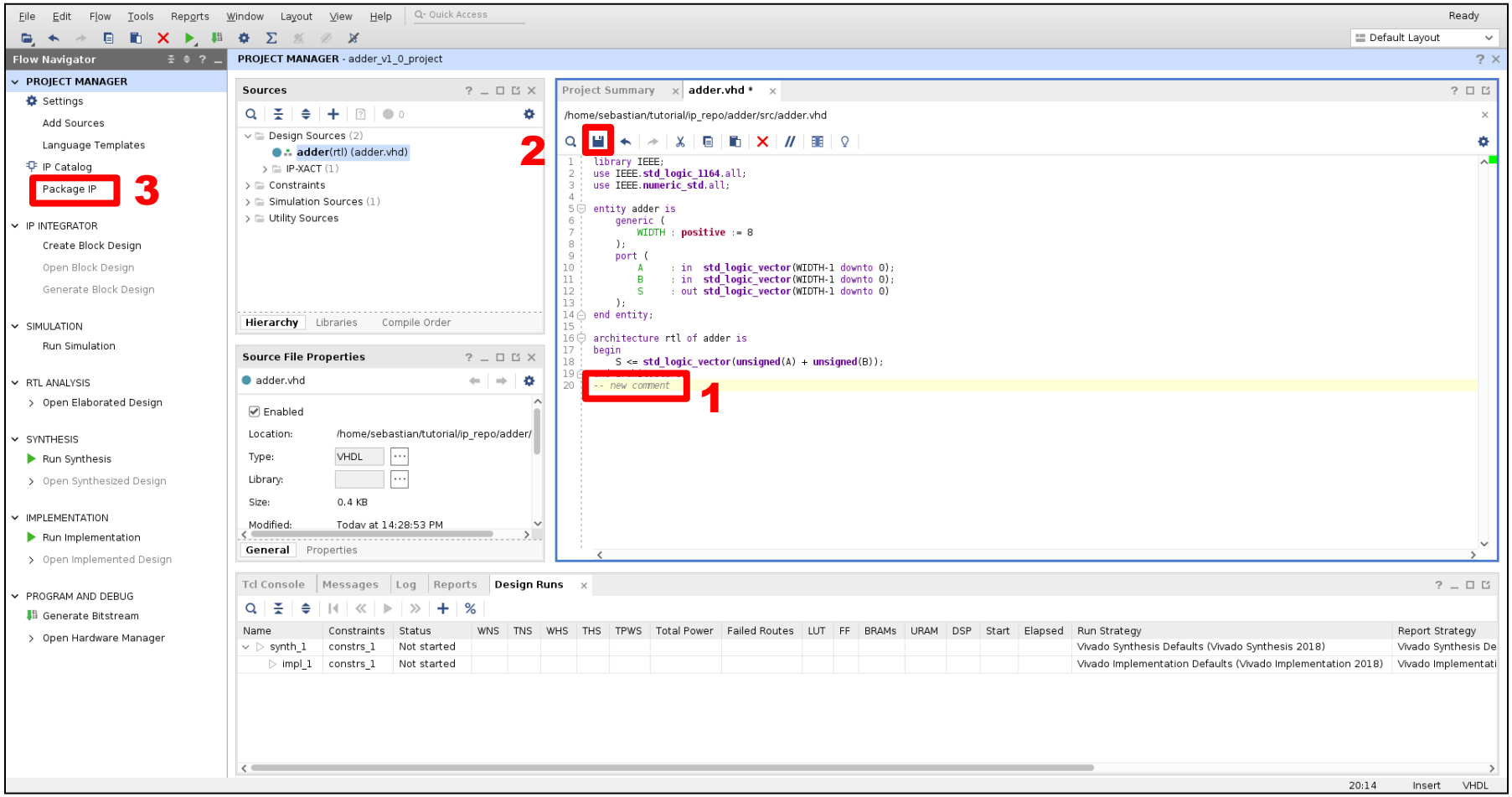
A new Vivado instance is created specifically for the **adder** IP. Use this instance to develop and package the **adder** IP. Continue.



Double-click **adder.vhd** to open it in the editor.

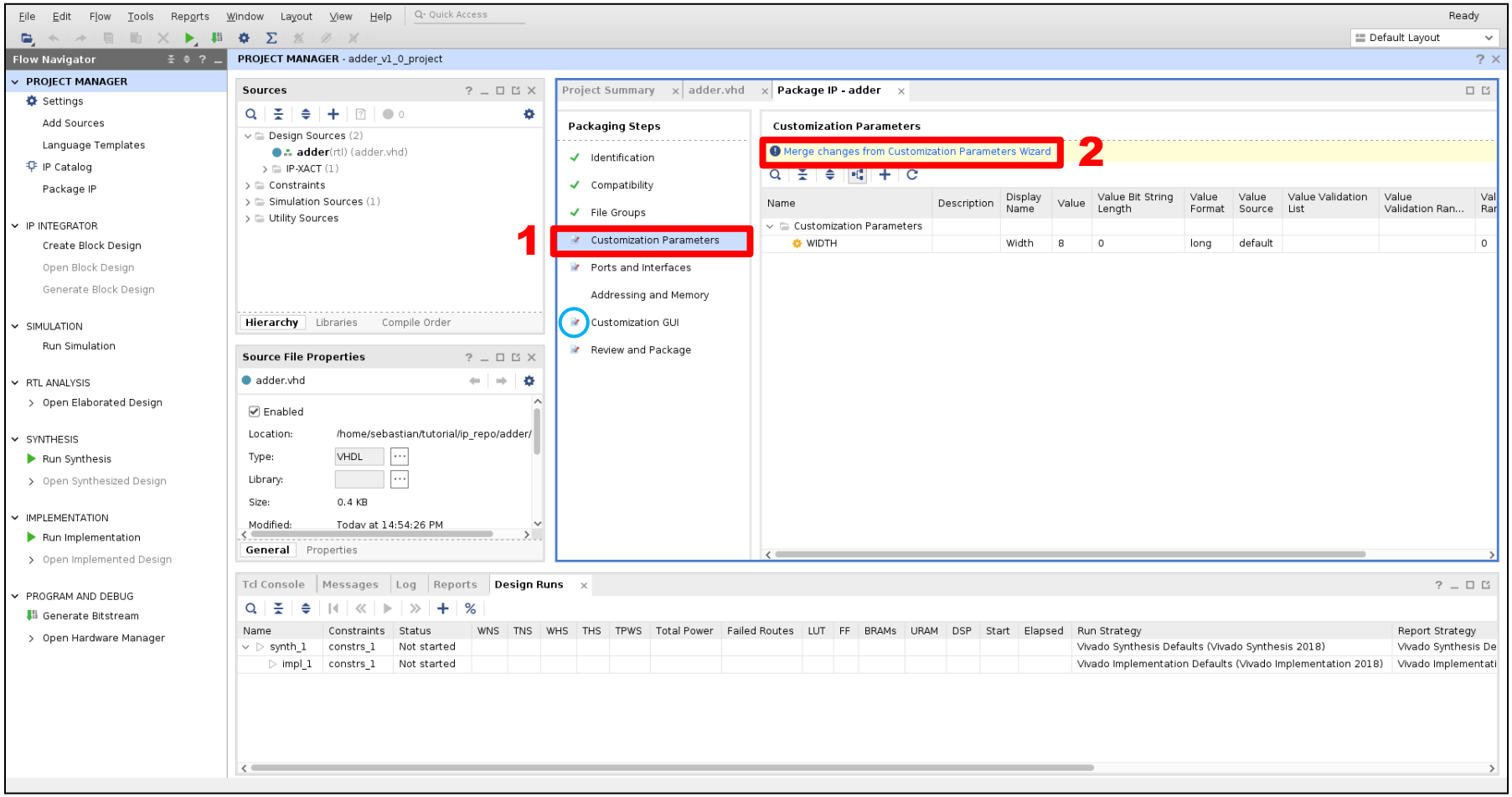


Add a commented line. Click ***Save***. Click ***Package IP***.

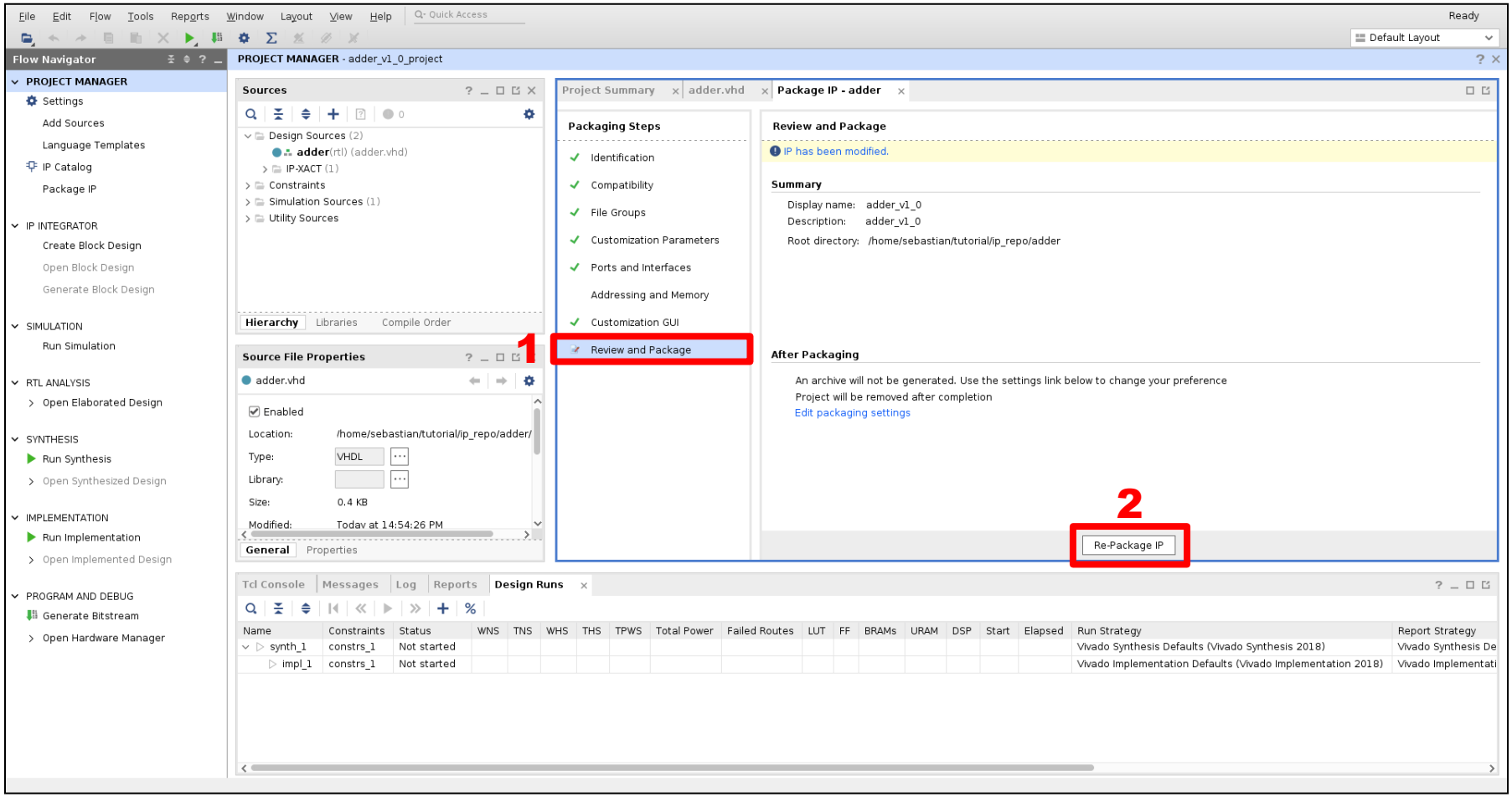


Click on any tabs containing the modified **symbol**. Click the yellow-highlighted link to merge changes.

NOTE: Often, clicking one link will merge all changes.

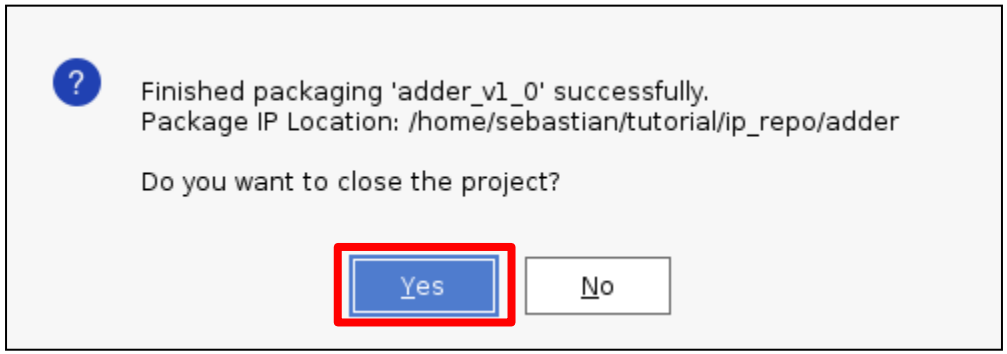


Click ***Review and Package***. Click ***Re-Package IP***.

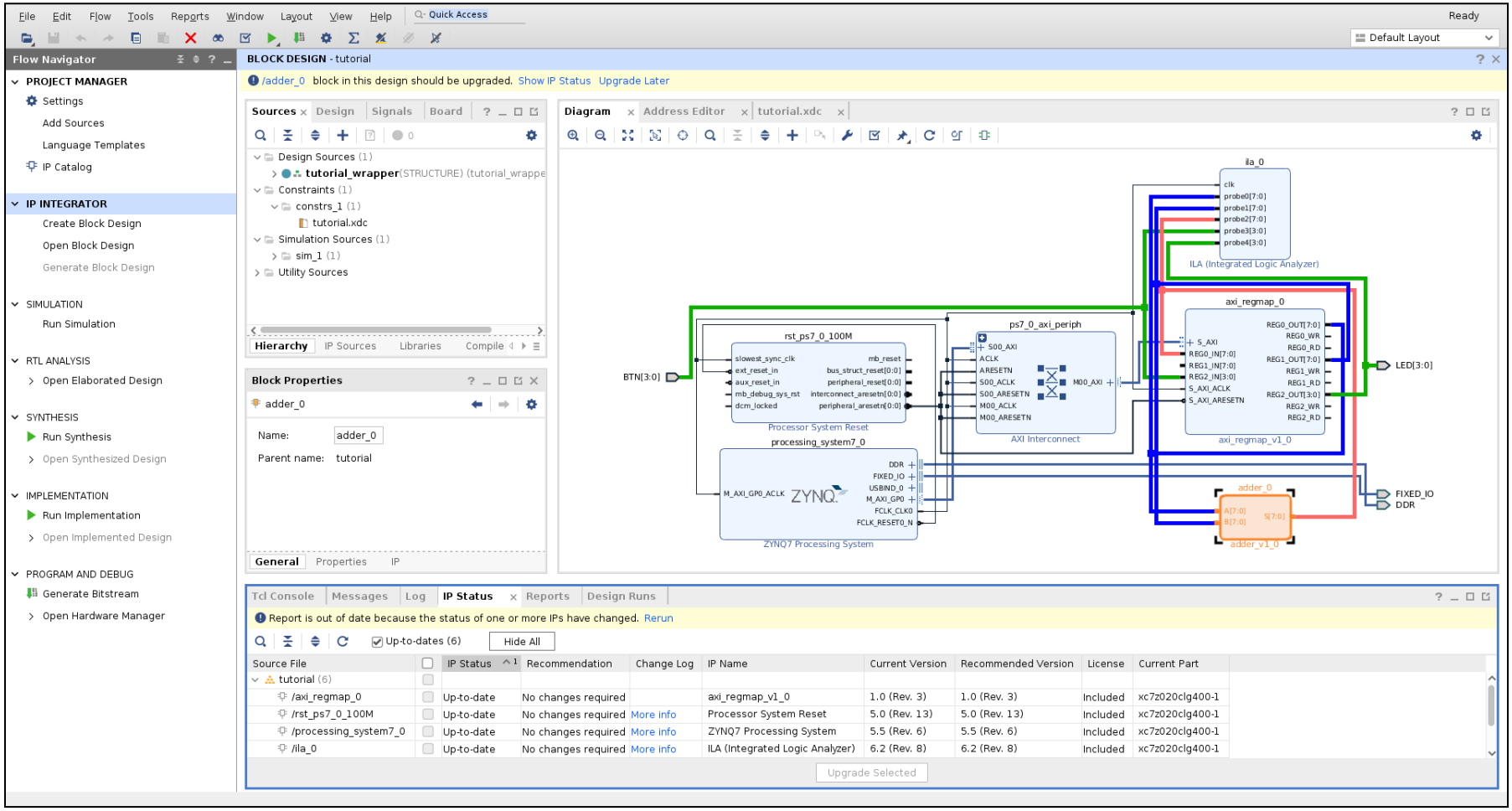


Click ***Yes***.

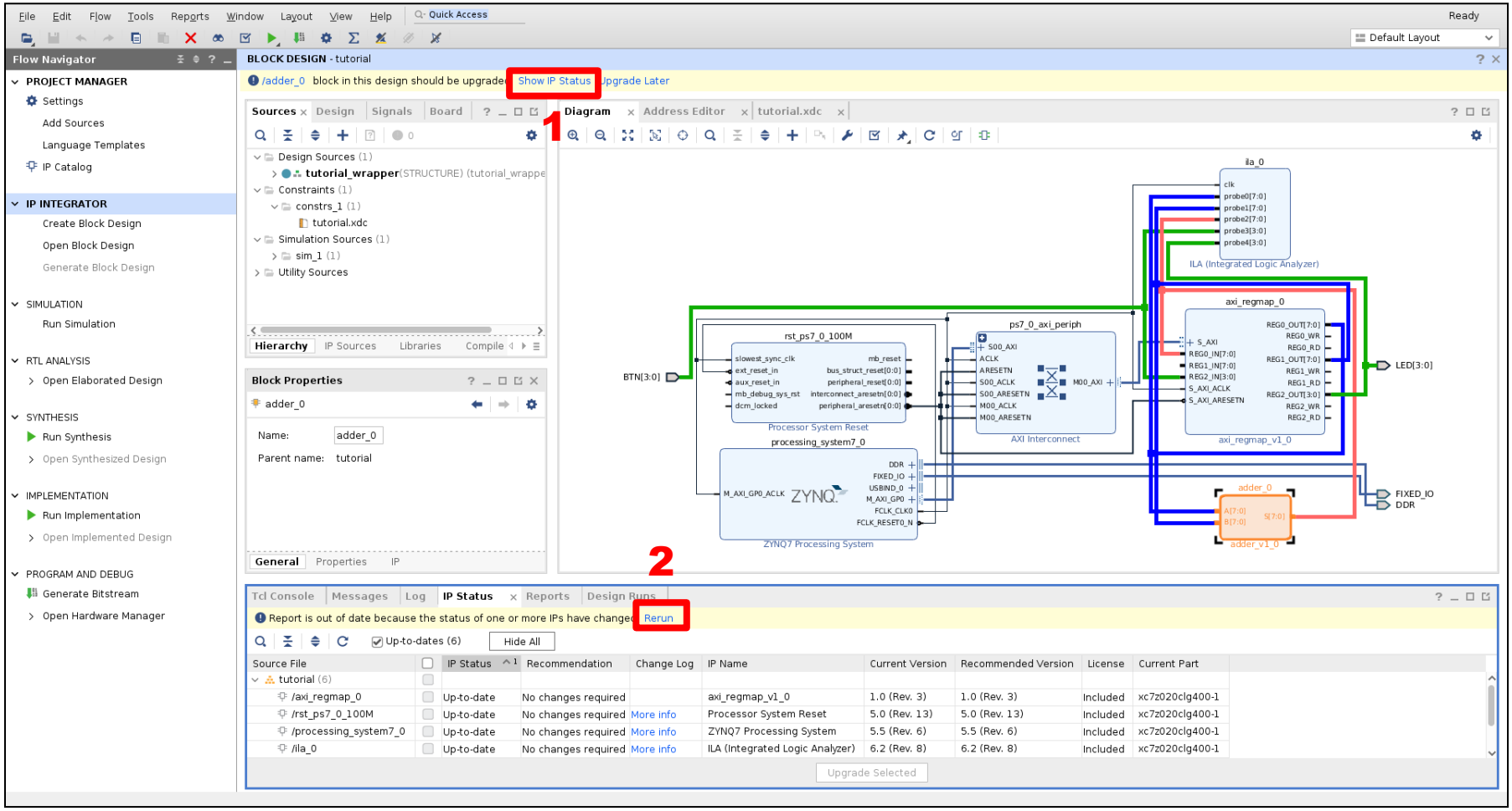
NOTE: You can optionally keep this Vivado instance open to continue developing your IP.



We return to the main Vivado instance. Continue.

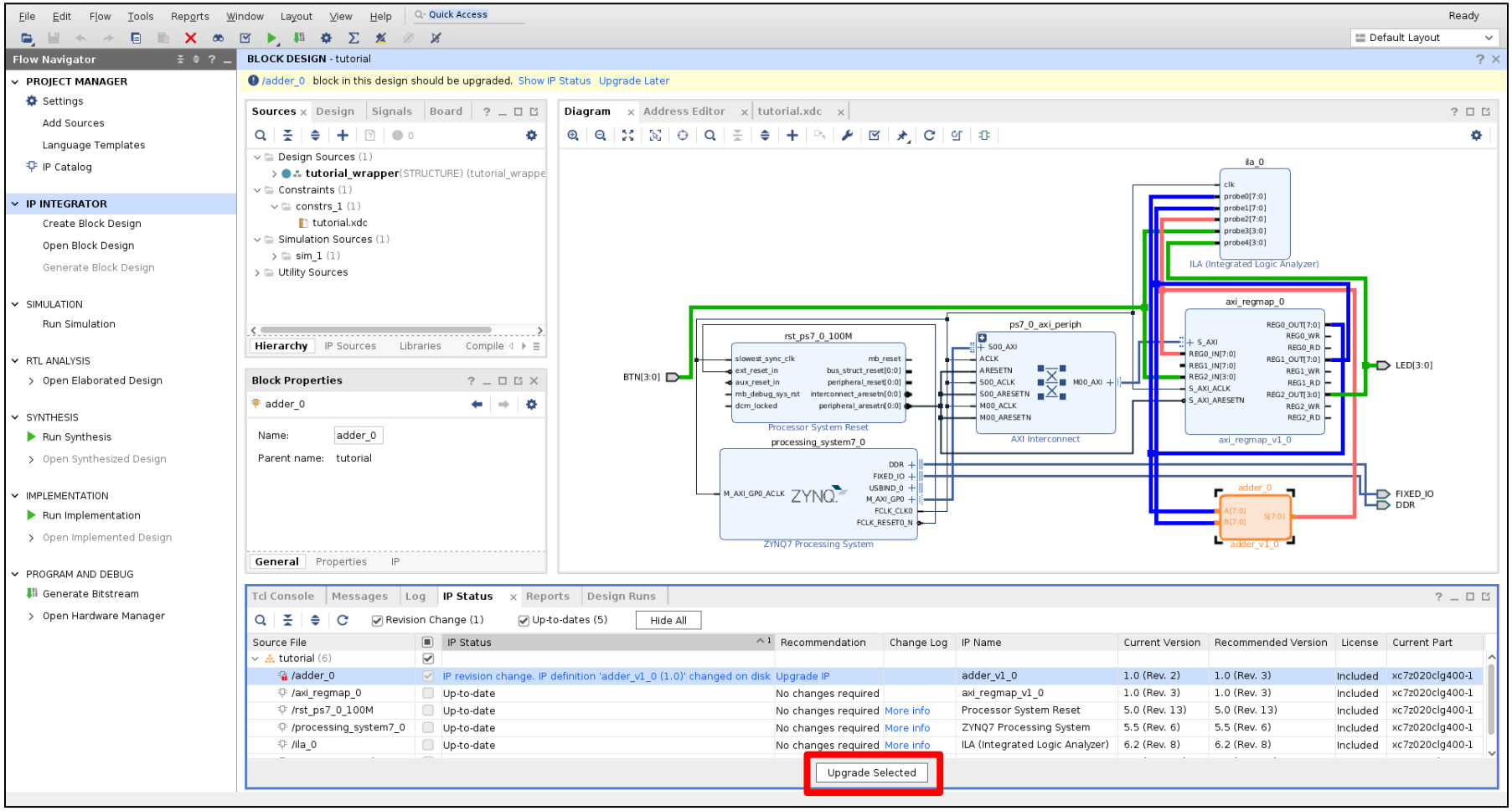


Click on the **adder_0** instance, then ***Show IP Status***. Click ***Rerun***.
Note: if you didn't see the **Rerun** option, continue to the next slide.

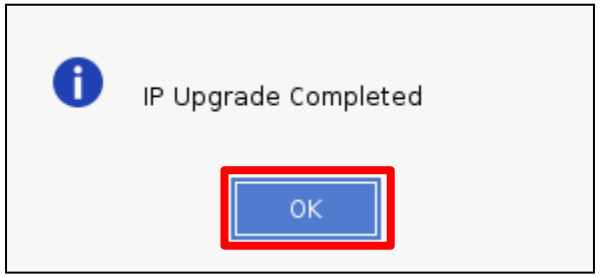


Click ***Upgrade Selected***.

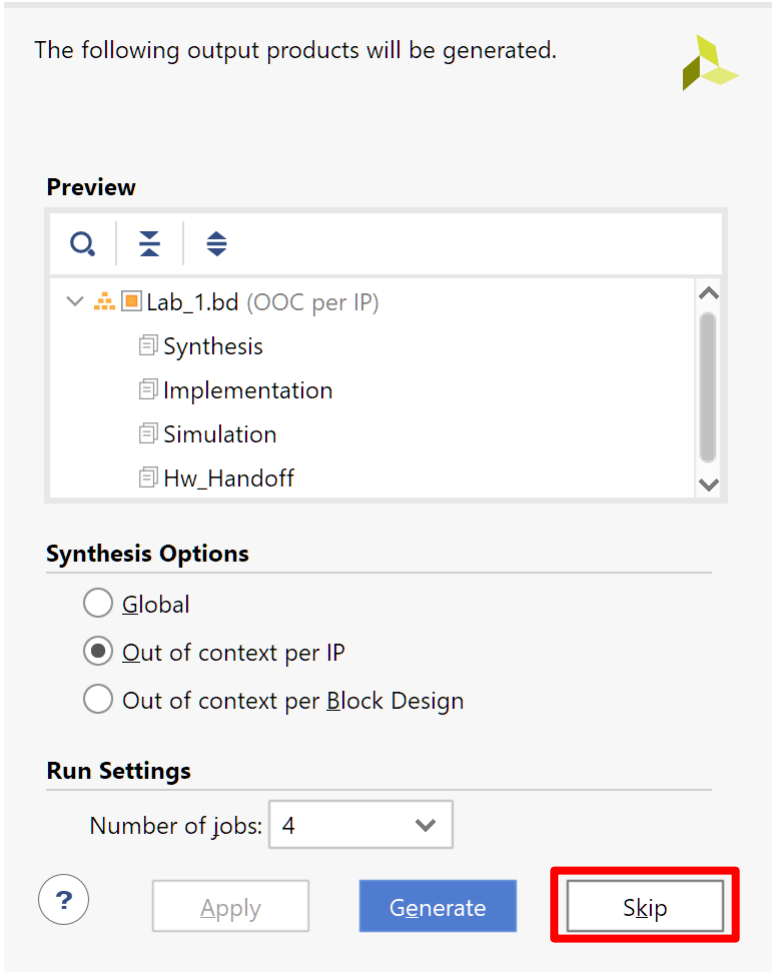
NOTE: All upgradeable IP are listed. All instances of a modified IP are upgraded.



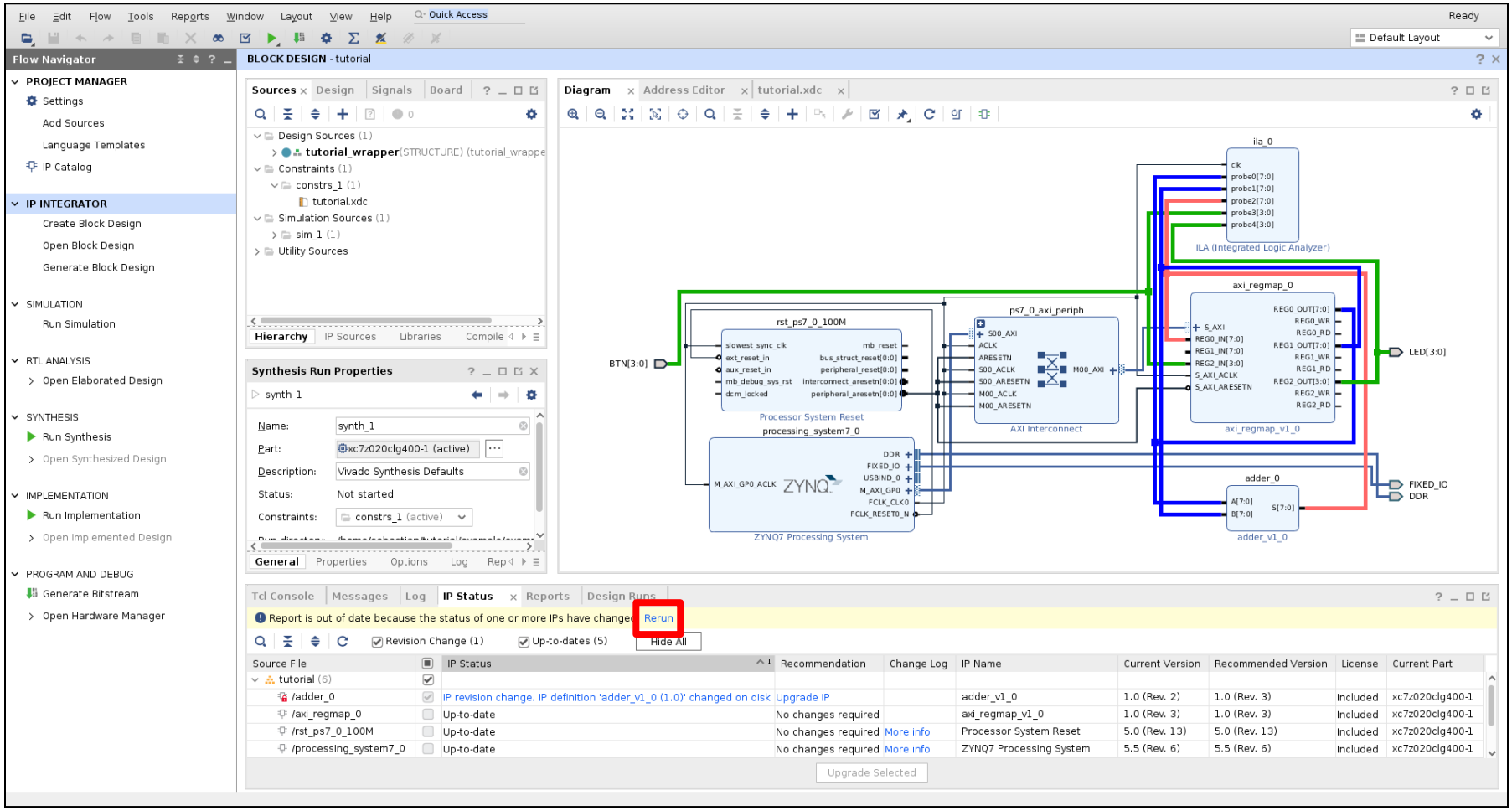
Click ***OK***.



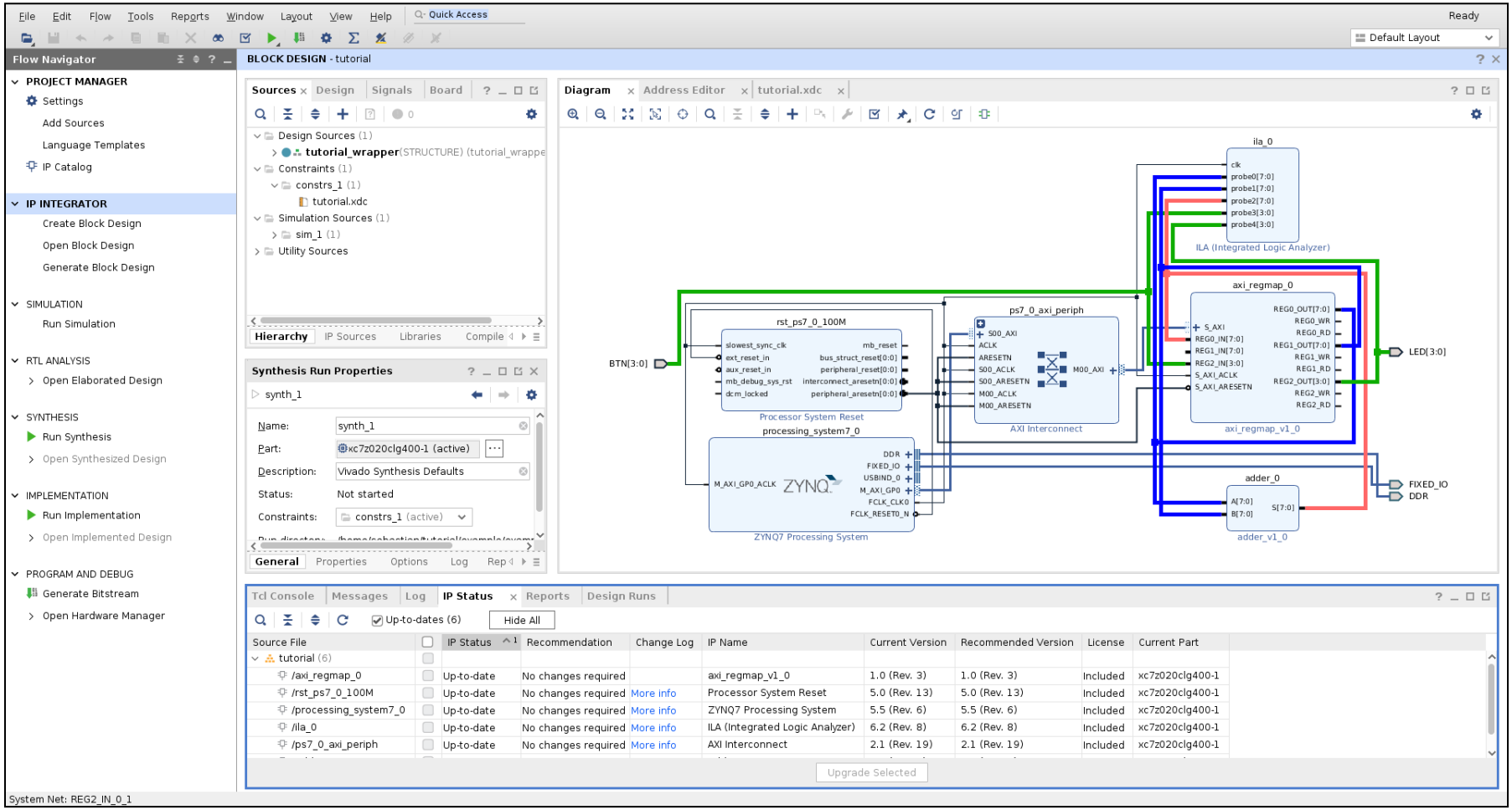
Click ***Skip***.
NOTE: This process is automatically performed during bitstream generation.



Click ***Rerun***.



Vivado did not find any more IP to upgrade. Continue.

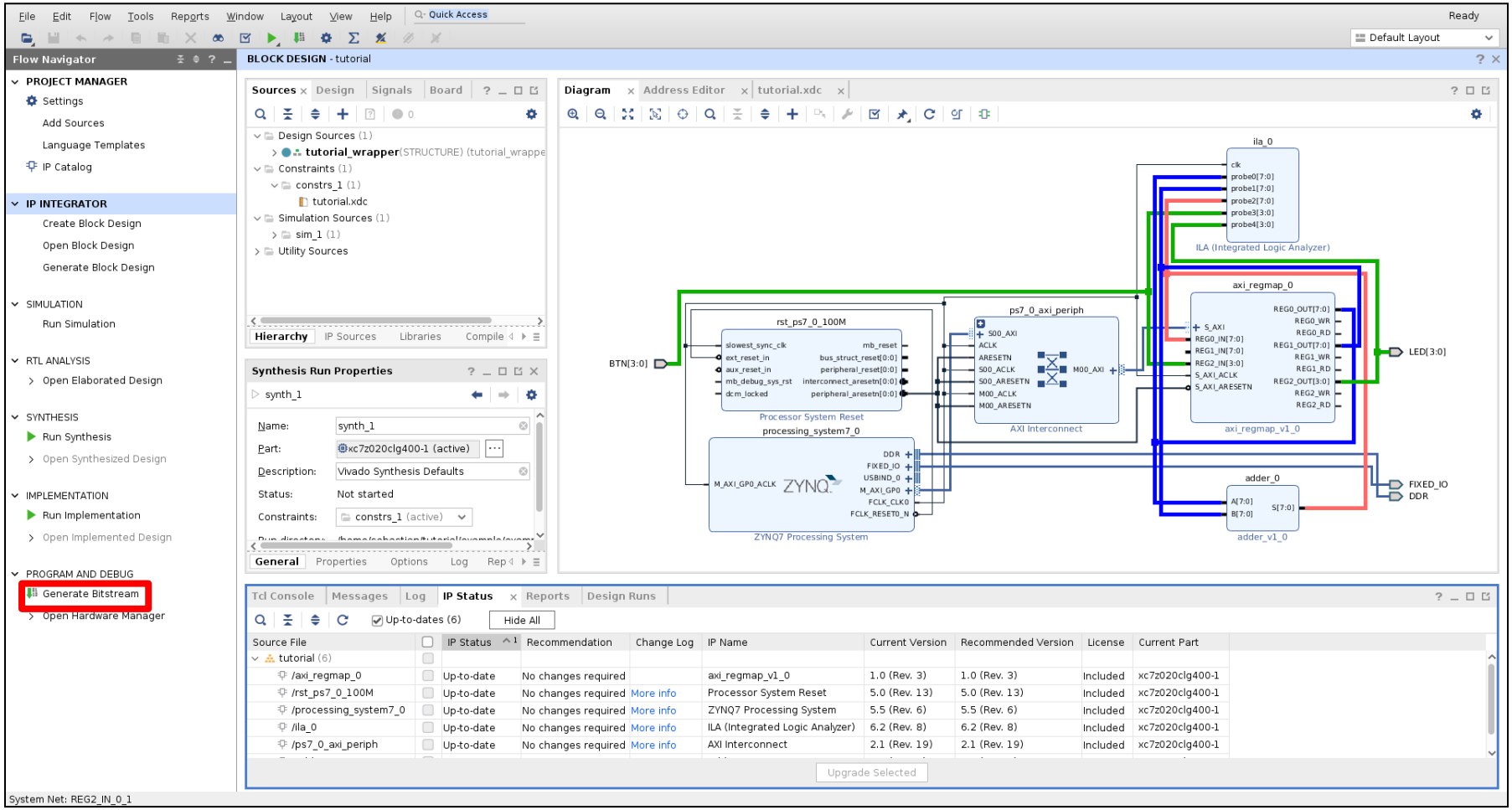


Part 6: Generating a Design Bitstream

Next, we will generate a bitstream and analyze the implemented design. The generated outputs will be transferred to a new SDK project for software development.

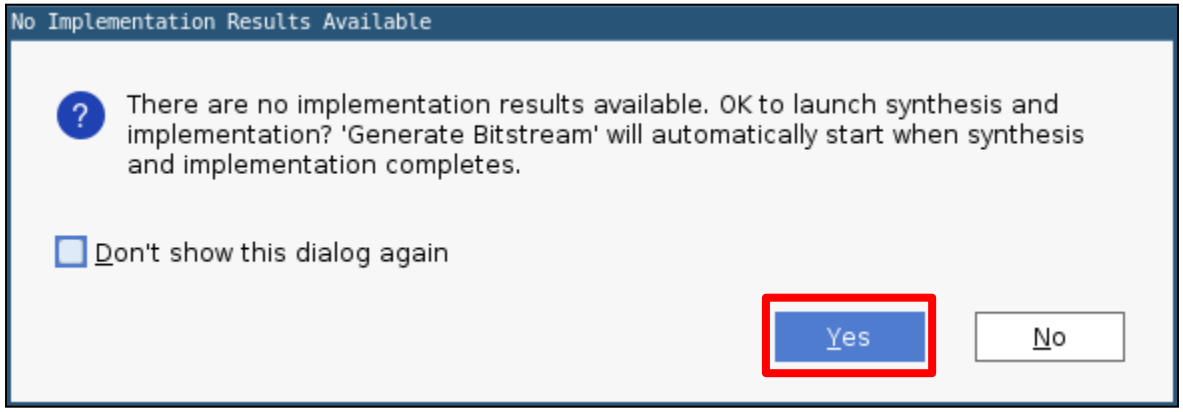
Click ***Generate Bitstream***.

NOTE: This process will generate the design bitstream (including synthesis and implementation of the design).

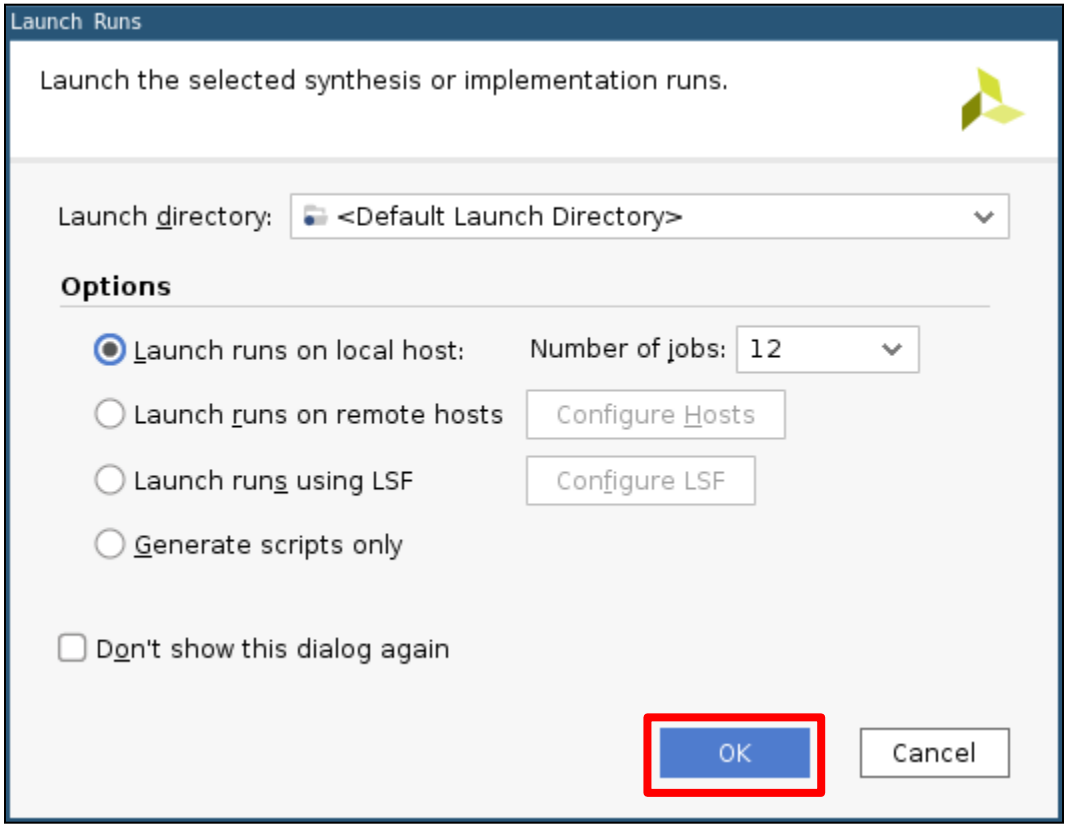


Source File	IP Status	Recommendation	Change Log	IP Name	Current Version	Recommended Version	License	Current Part
tutorial (6)								
/axi_regmap_0	Up-to-date	No changes required		axi_regmap_v1_0	1.0 (Rev. 3)	1.0 (Rev. 3)	Included	xc7z020clg400-1
/rst_ps7_0_100M	Up-to-date	No changes required	More info	Processor System Reset	5.0 (Rev. 13)	5.0 (Rev. 13)	Included	xc7z020clg400-1
/processing_system7_0	Up-to-date	No changes required	More info	ZYNQ7 Processing System	5.5 (Rev. 6)	5.5 (Rev. 6)	Included	xc7z020clg400-1
/ila_0	Up-to-date	No changes required	More info	ILA (Integrated Logic Analyzer)	6.2 (Rev. 8)	6.2 (Rev. 8)	Included	xc7z020clg400-1
/ps7_0_axi_periph	Up-to-date	No changes required	More info	AXI Interconnect	2.1 (Rev. 19)	2.1 (Rev. 19)	Included	xc7z020clg400-1

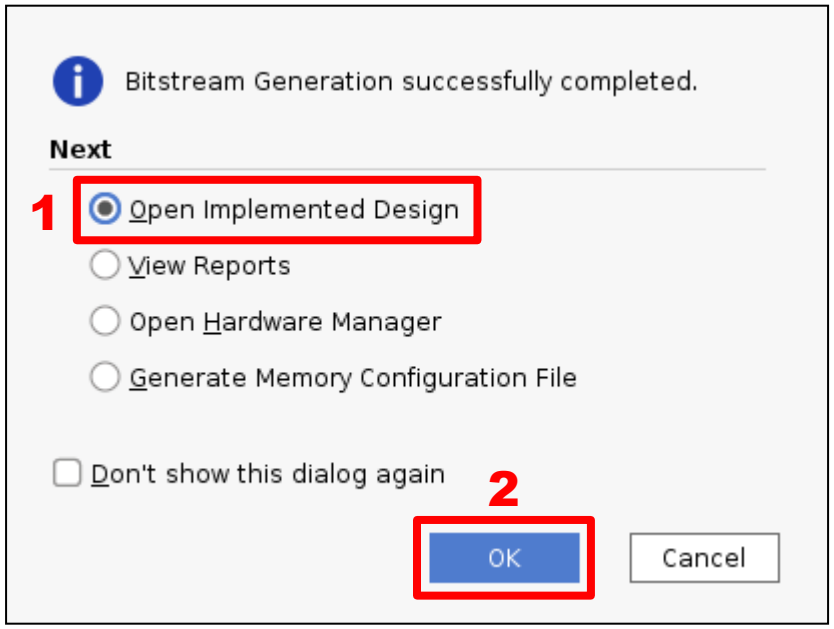
Click ***Yes***.



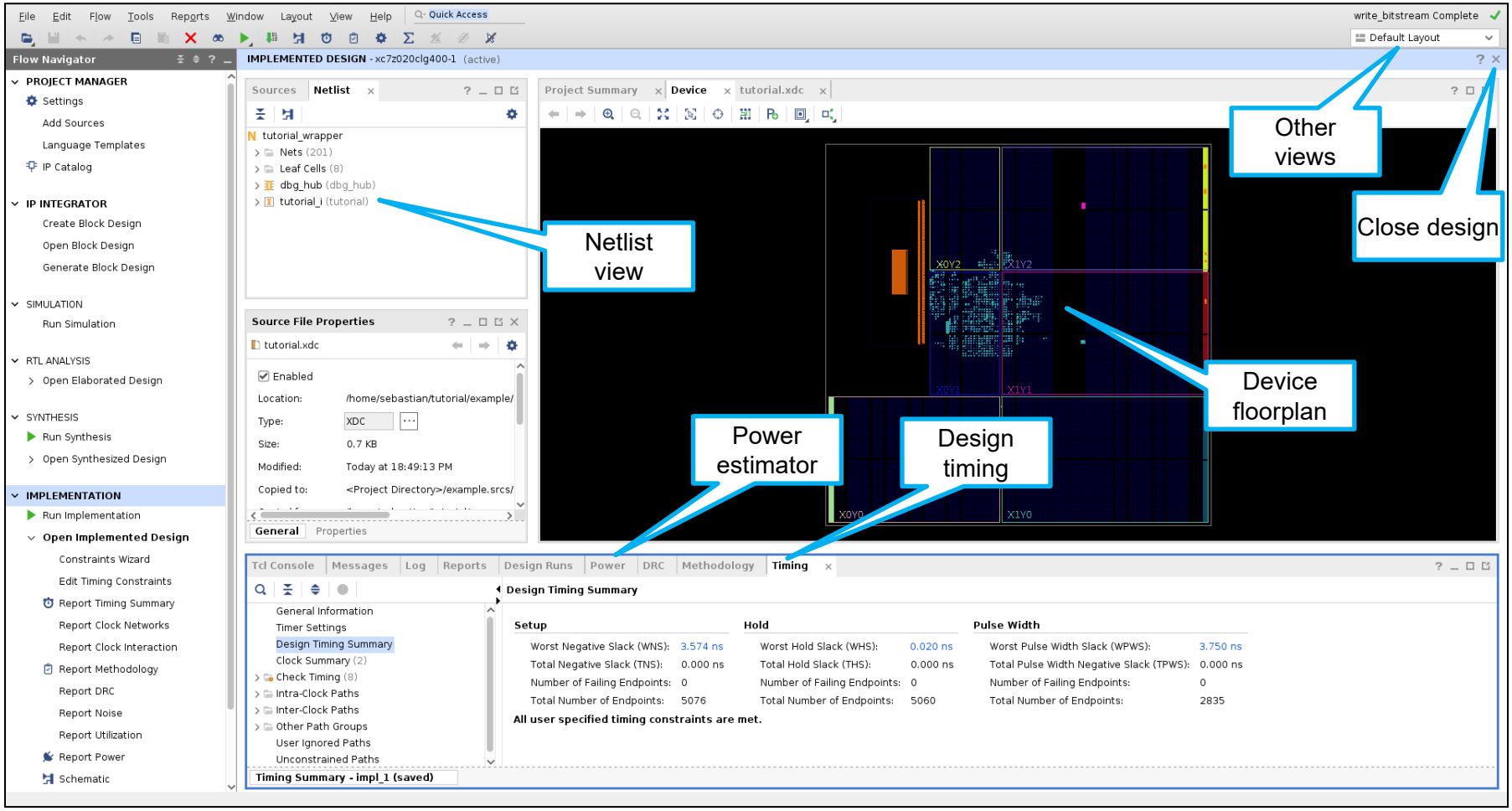
Click ***OK***.



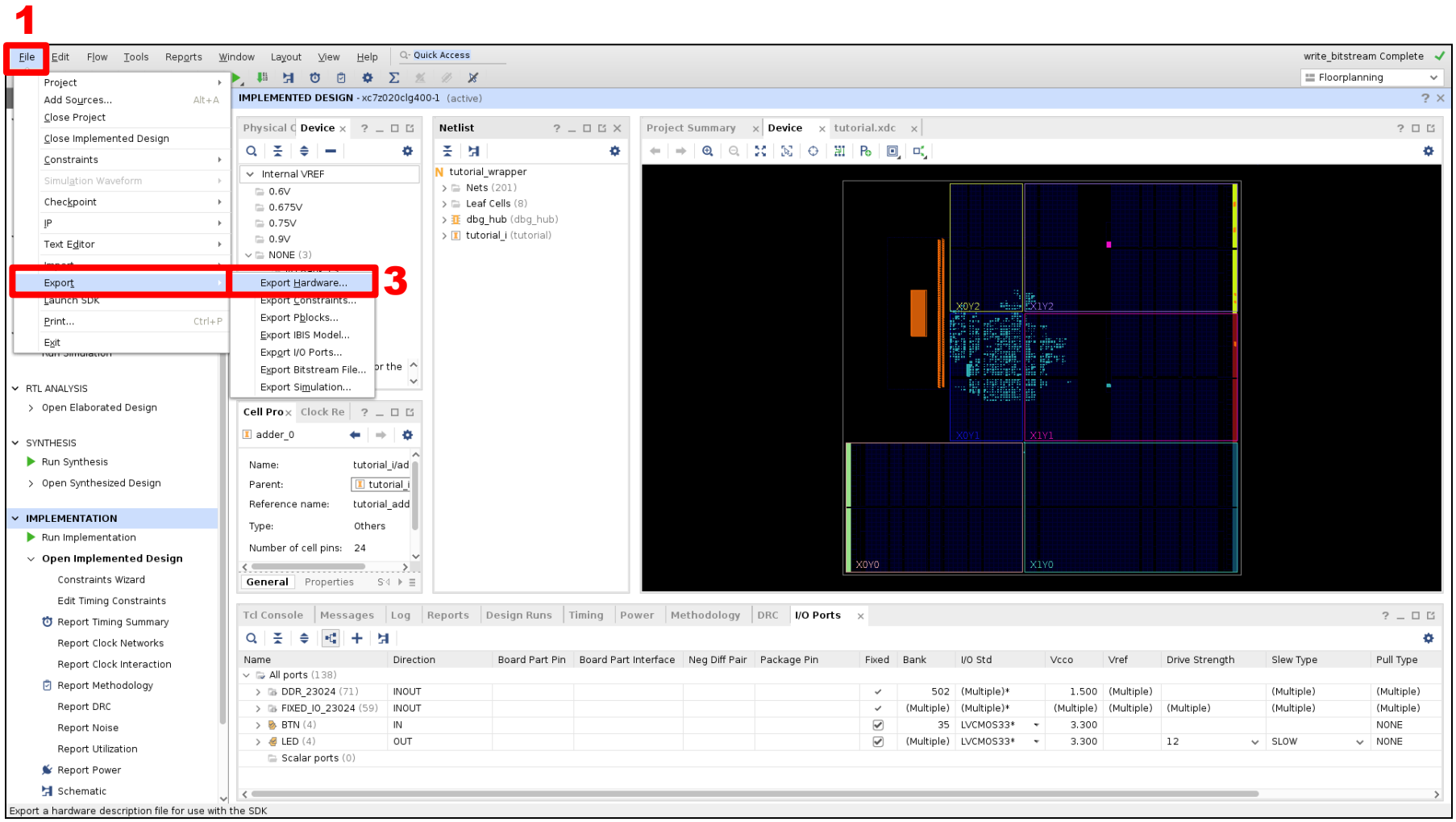
Wait until bitstream generation has completed. Select ***Open Implemented Design***. Click ***OK***.



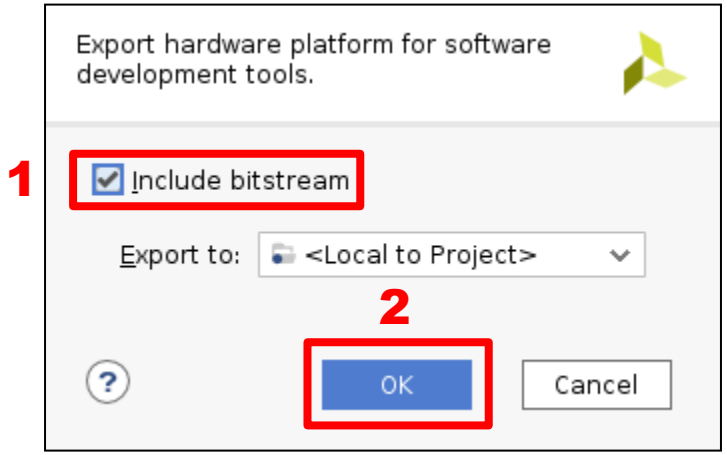
This is the Implemented Design GUI. Continue.



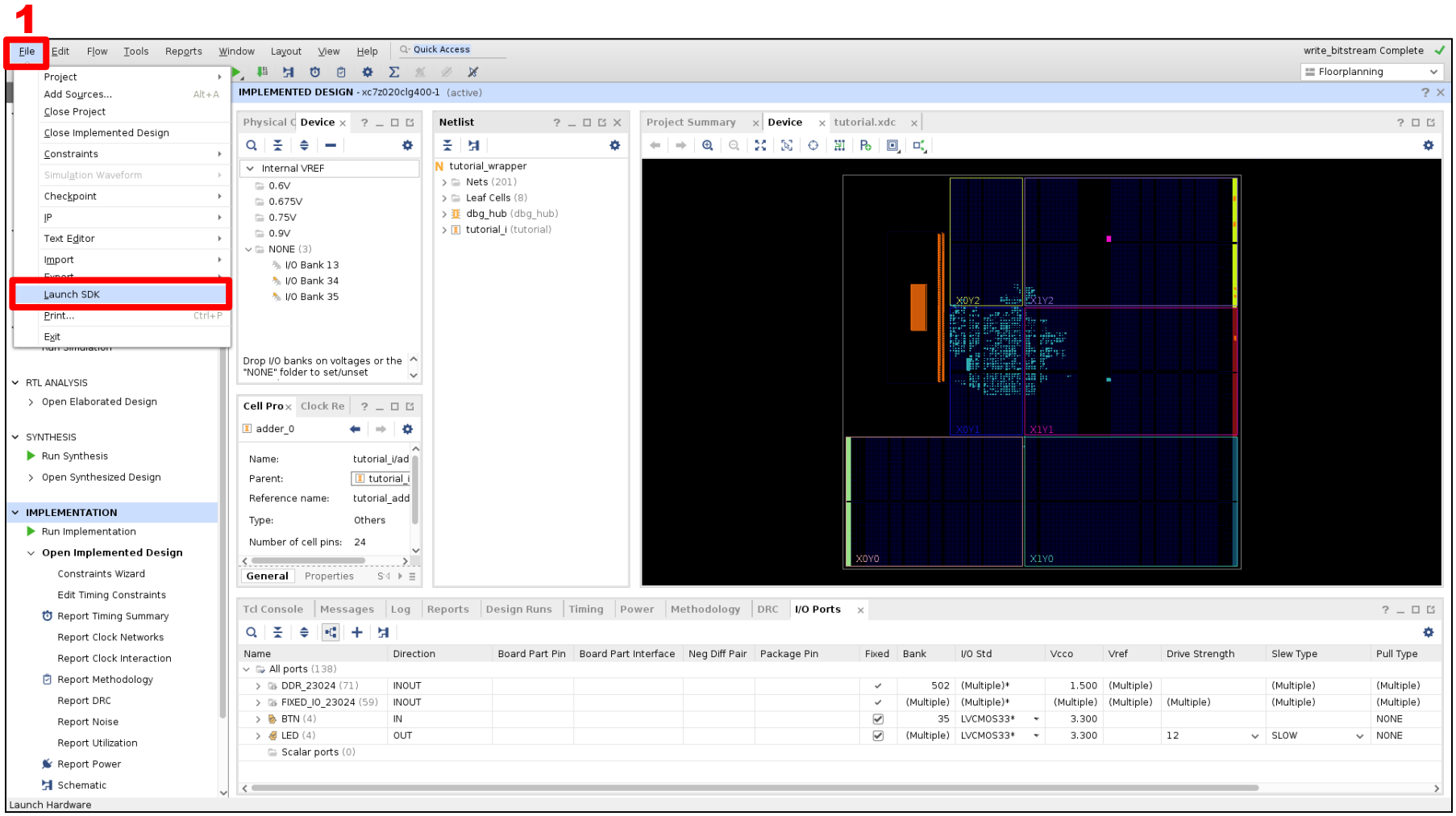
Click ***File***. Click ***Export***. Click ***Export Hardware…***.
NOTE: This process will export the design and launch SDK.



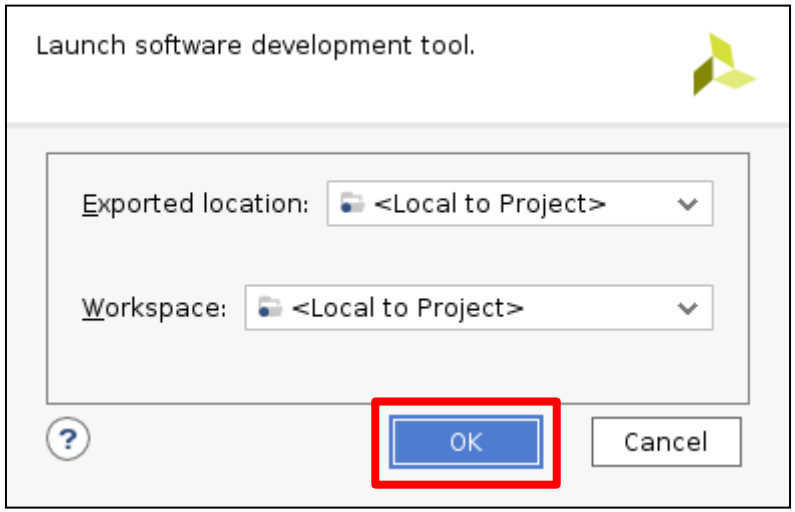
Check ***Include bitstream***. Click ***OK***.



Click ***File***. Click ***Launch SDK***.



Click ***OK***.

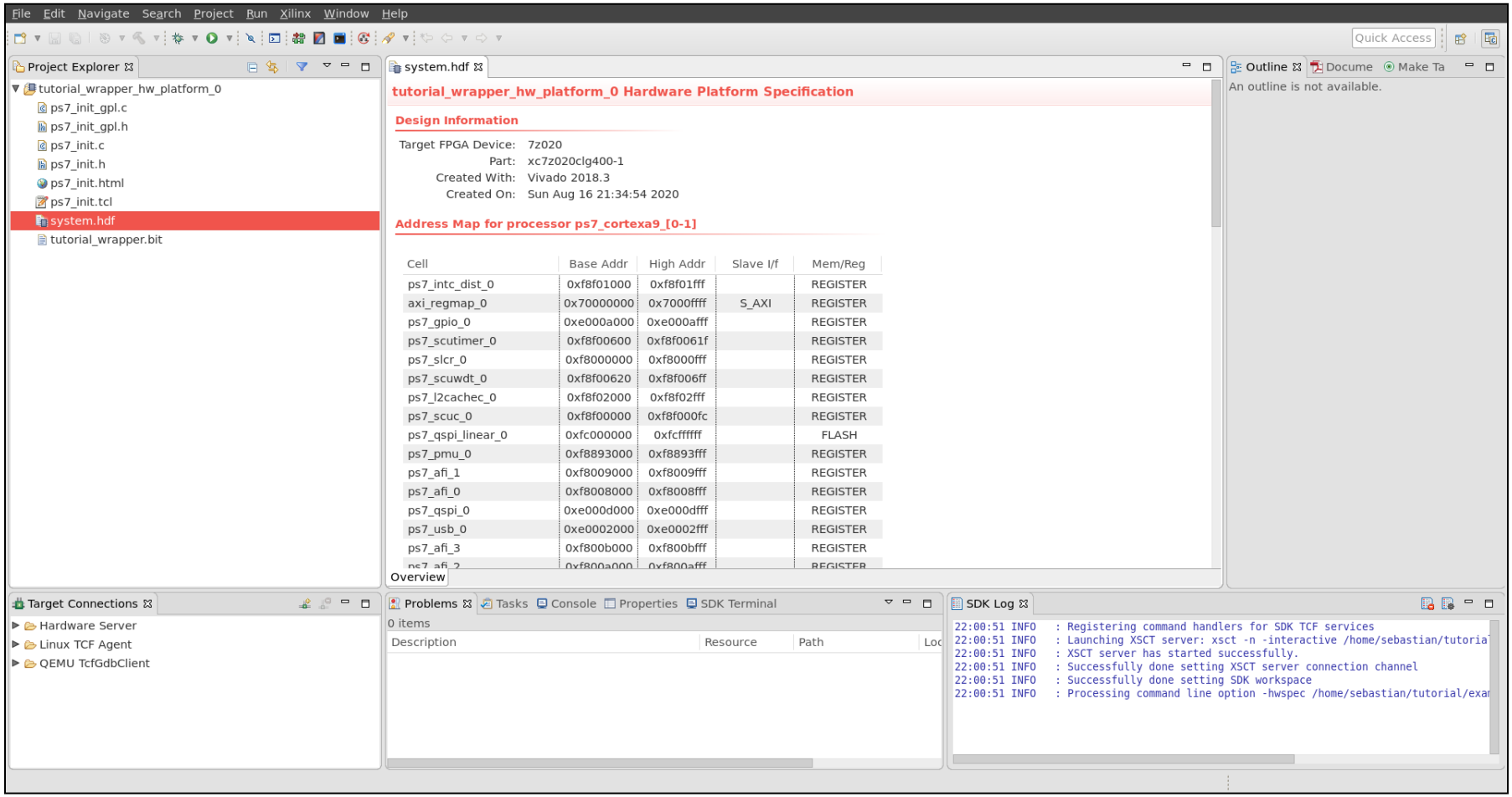


Xilinx SDK Tutorial

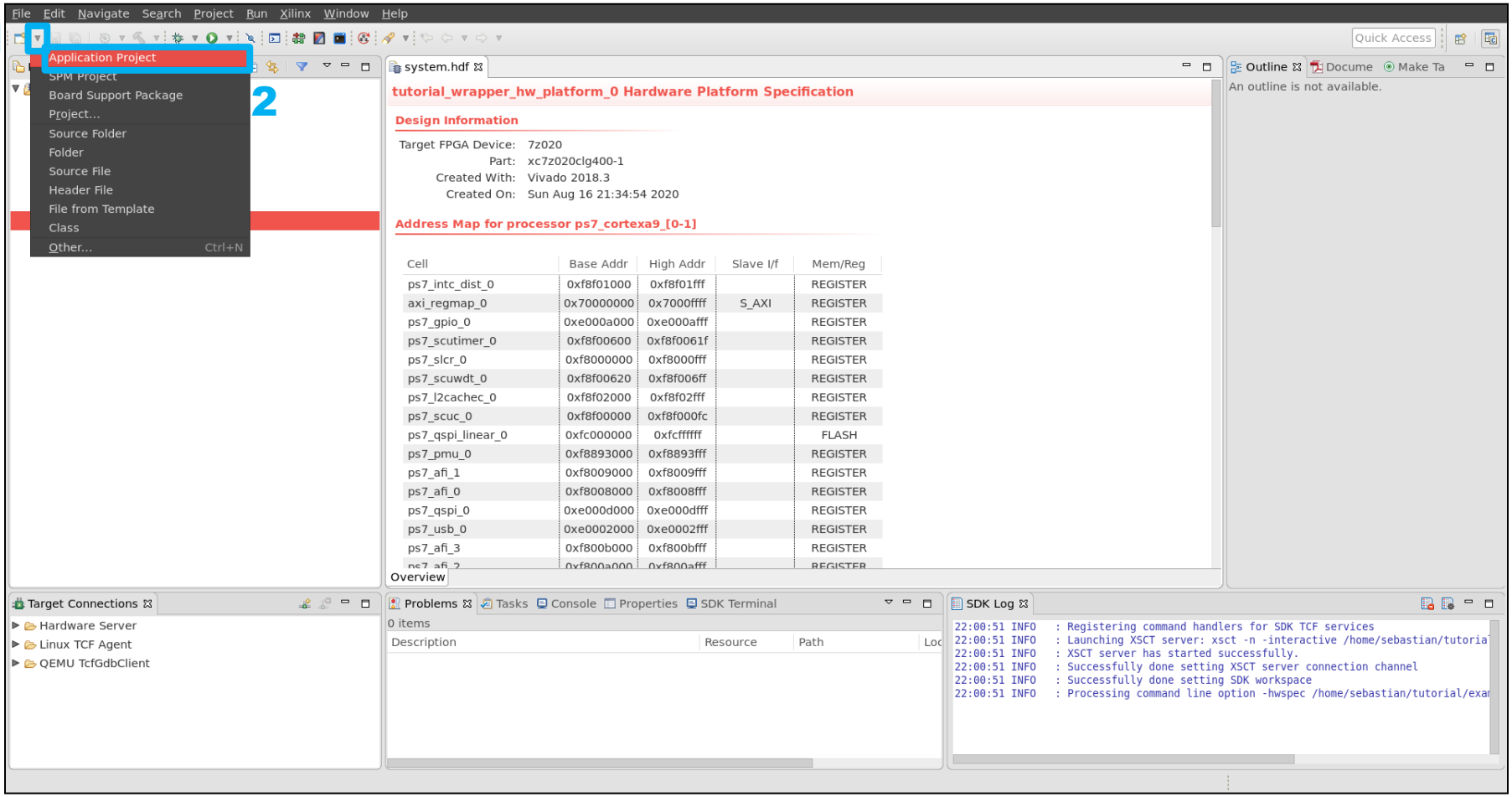
Part 1: Creating Application Projects

We will create two application projects: **adder** and **io**. The **adder** app will test the **adder** IP, and the **io** app will echo button inputs to LED outputs.

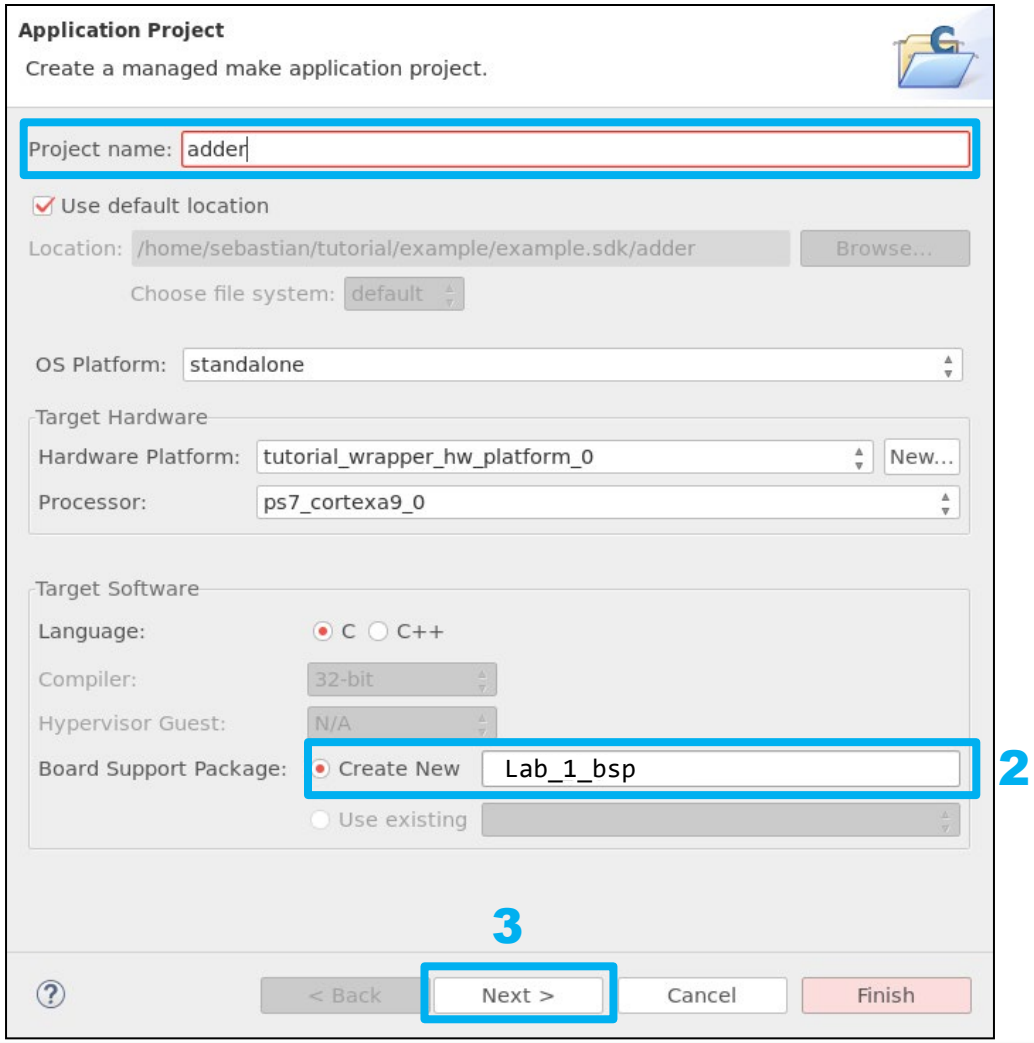
This is the SDK GUI. Continue.



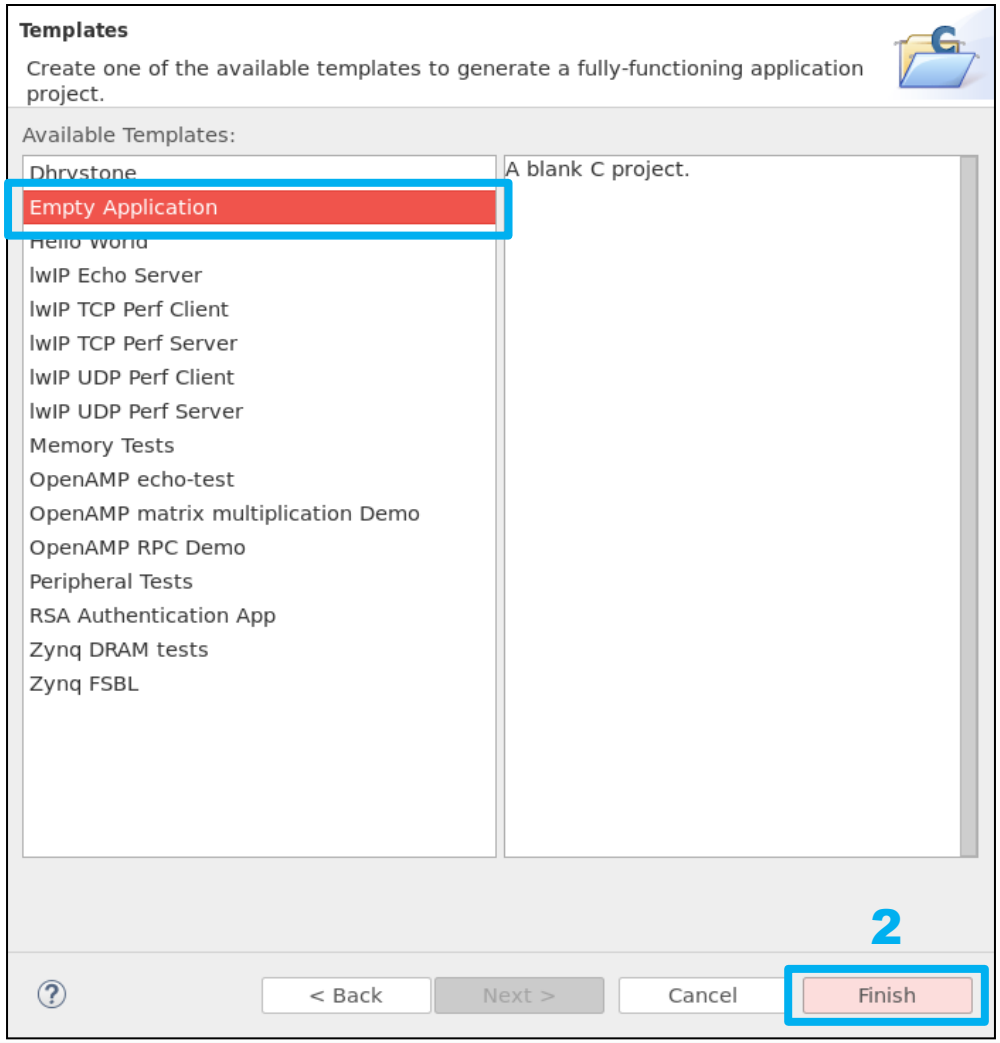
Click the ***New (drop-down)***. Click ***Application Project***.



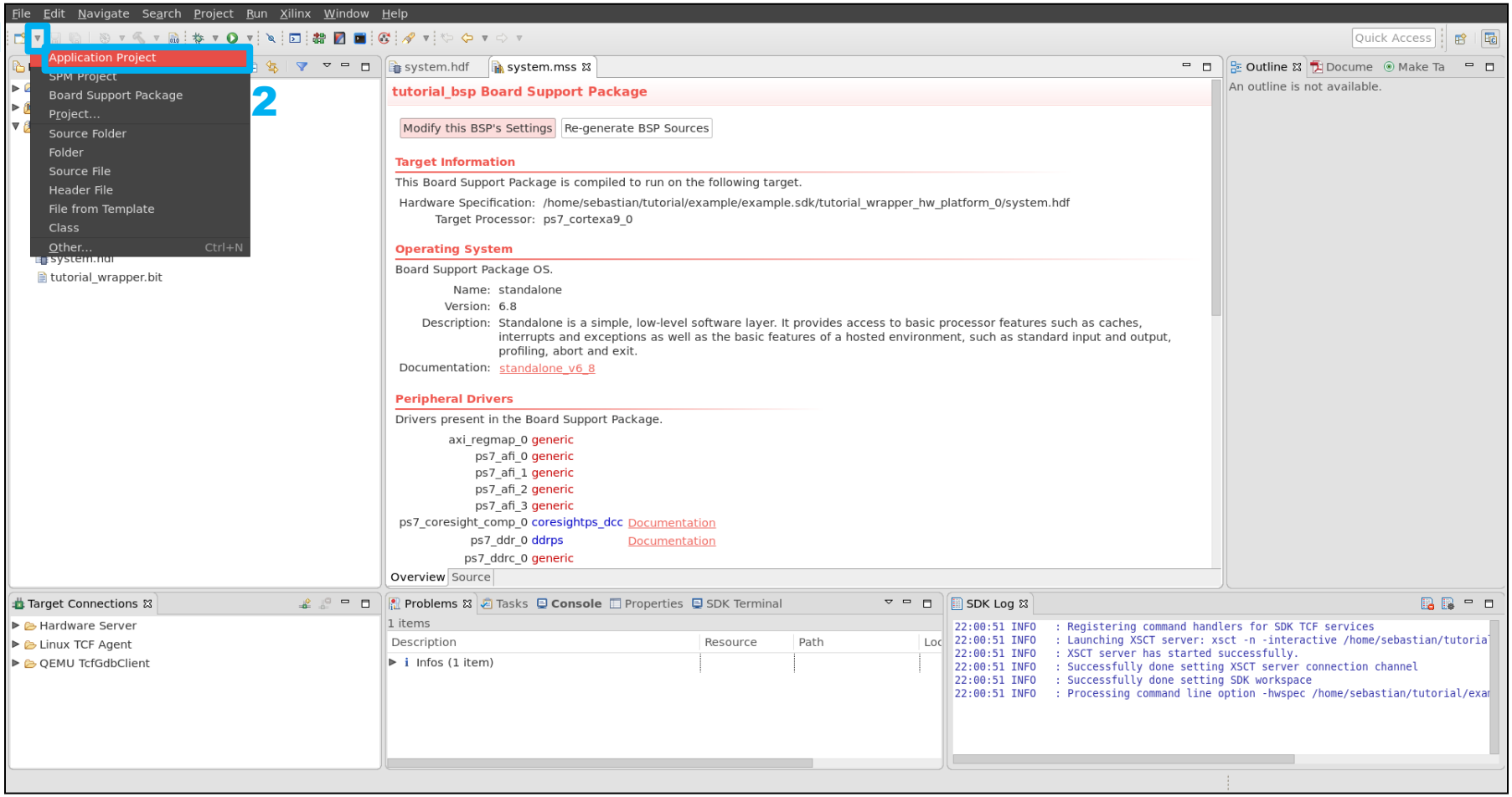
Enter project name **adder**. Select to ***Create New*** BSP and enter **Lab_1_bsp**. Click ***Next***. Note" naming might be different than the picture



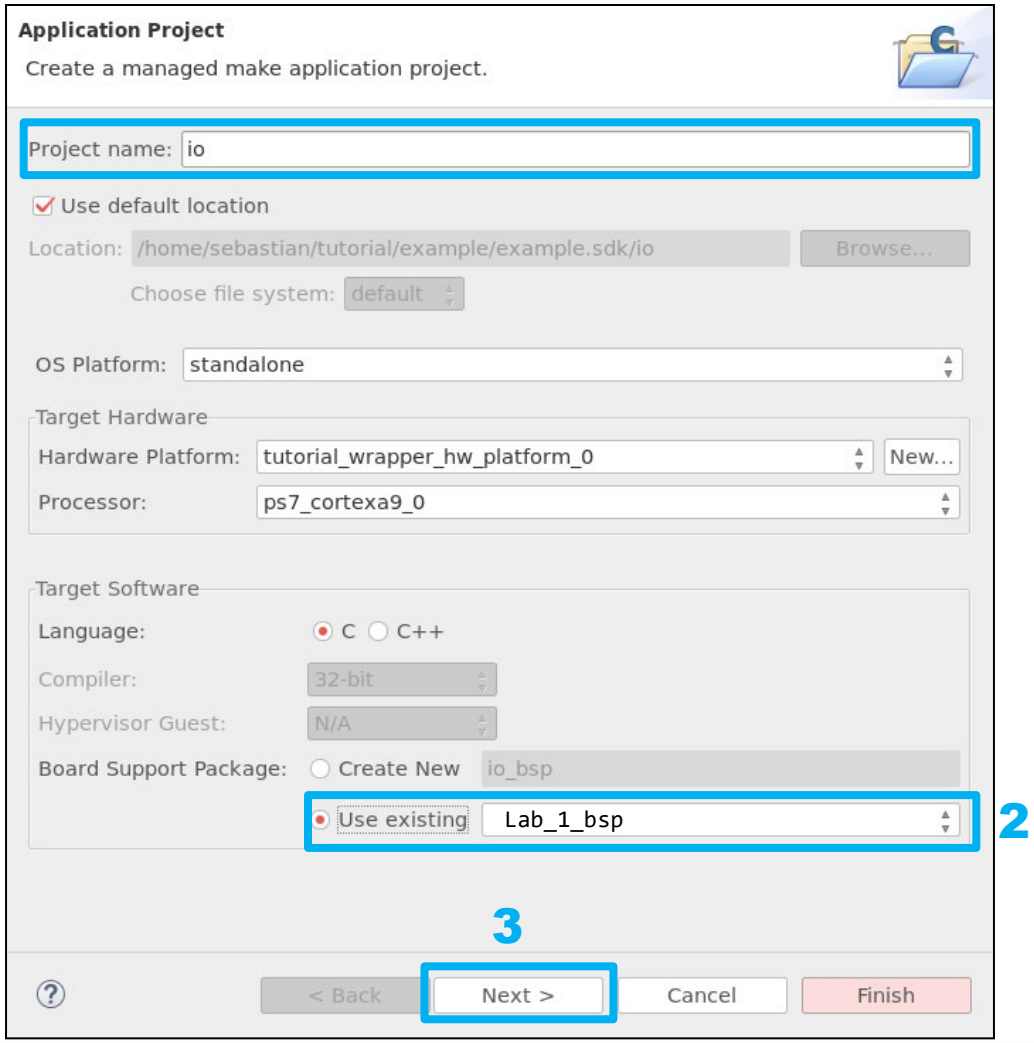
Select ***Empty Application***. Click ***Finish***.



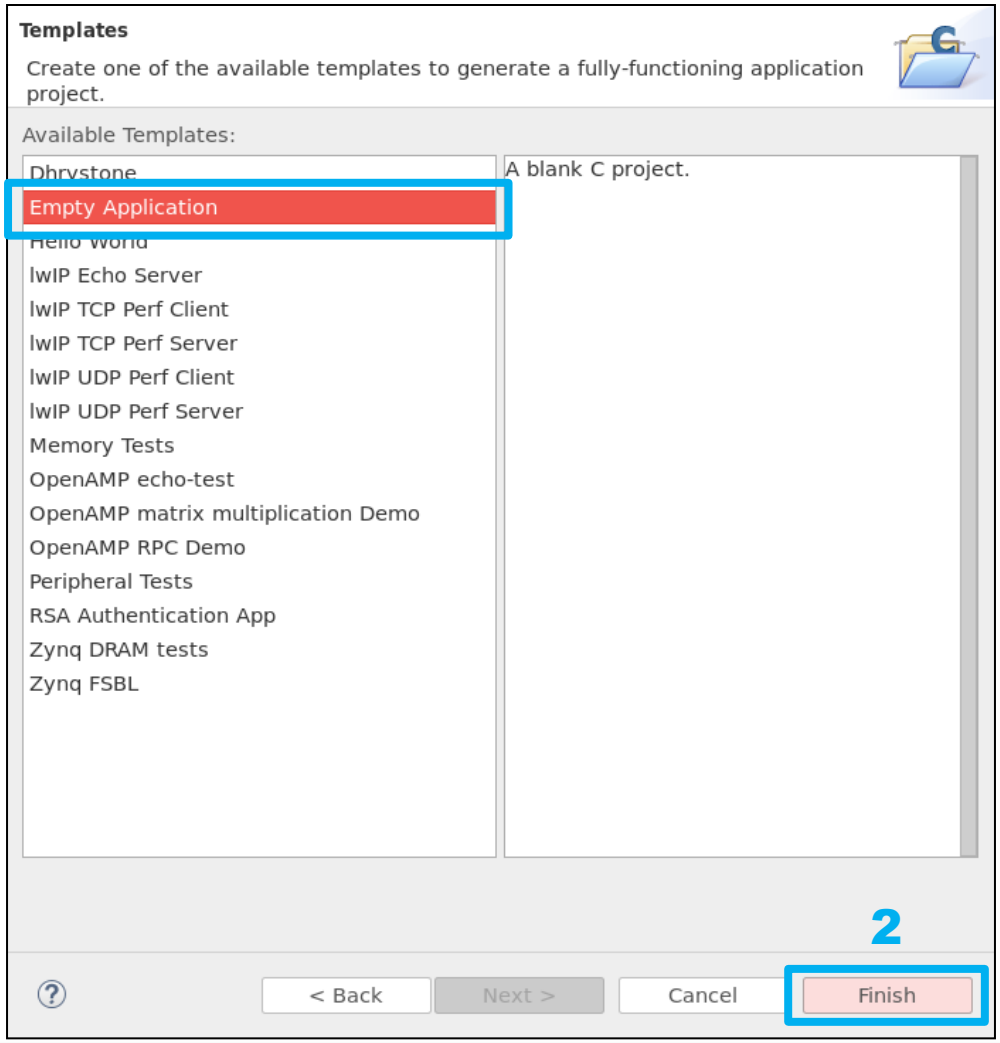
Click the ***New (drop-down)***. Click ***Application Project***.



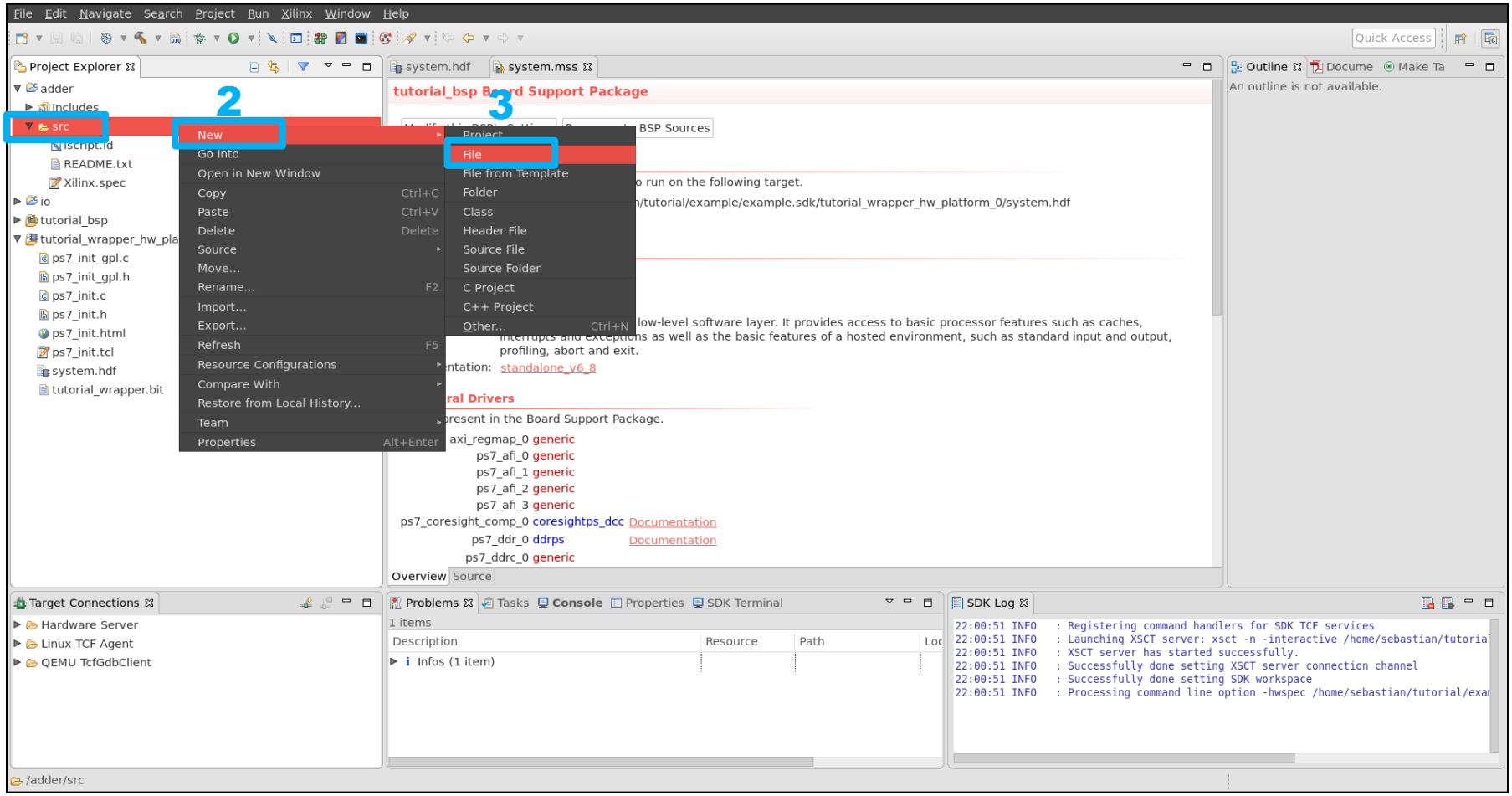
Enter project name **io**. Select to ***Use existing*** BSP and select **Lab_1_bsp**. Click ***Next***.



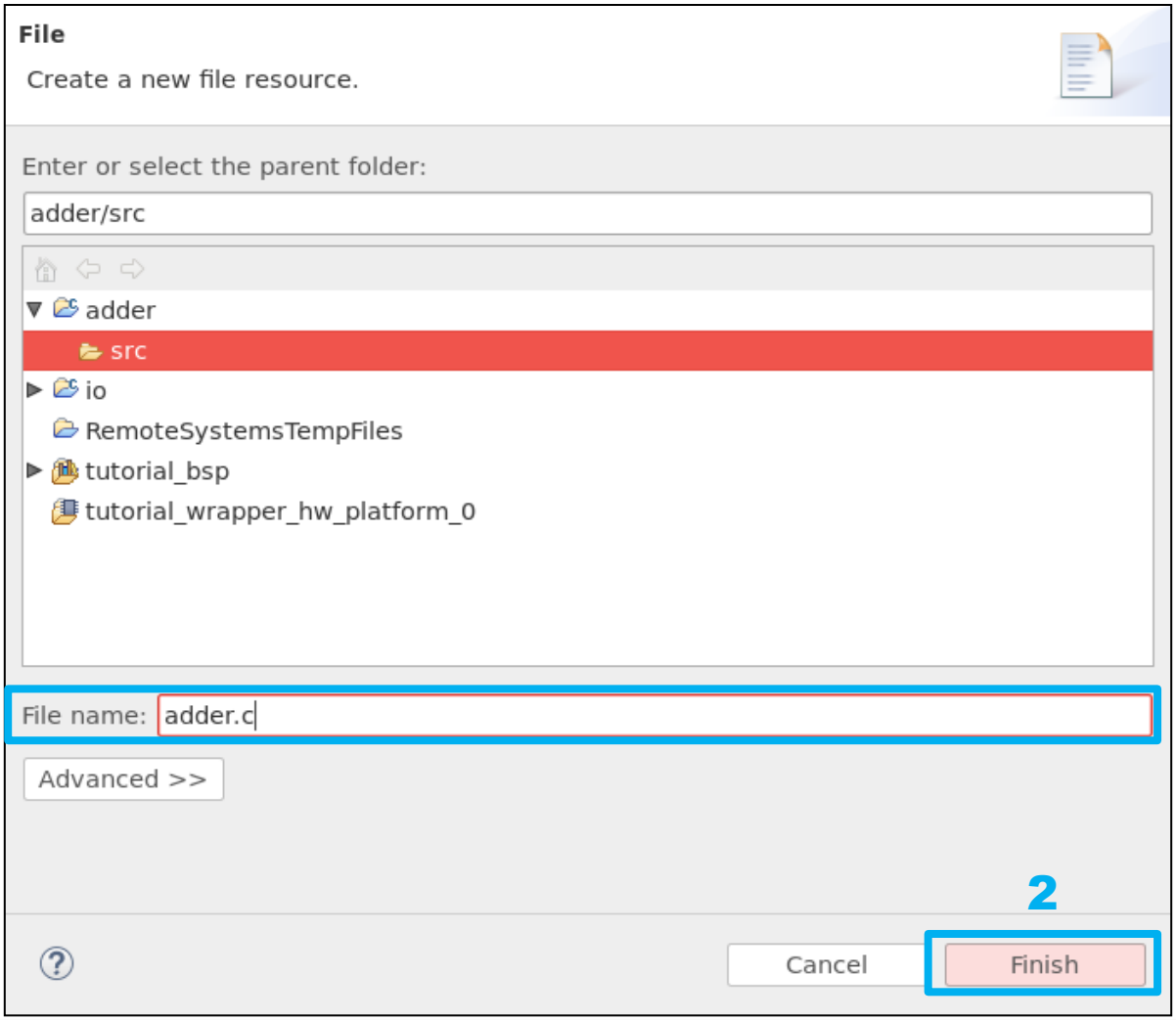
Select ***Empty Application***. Click ***Finish***.



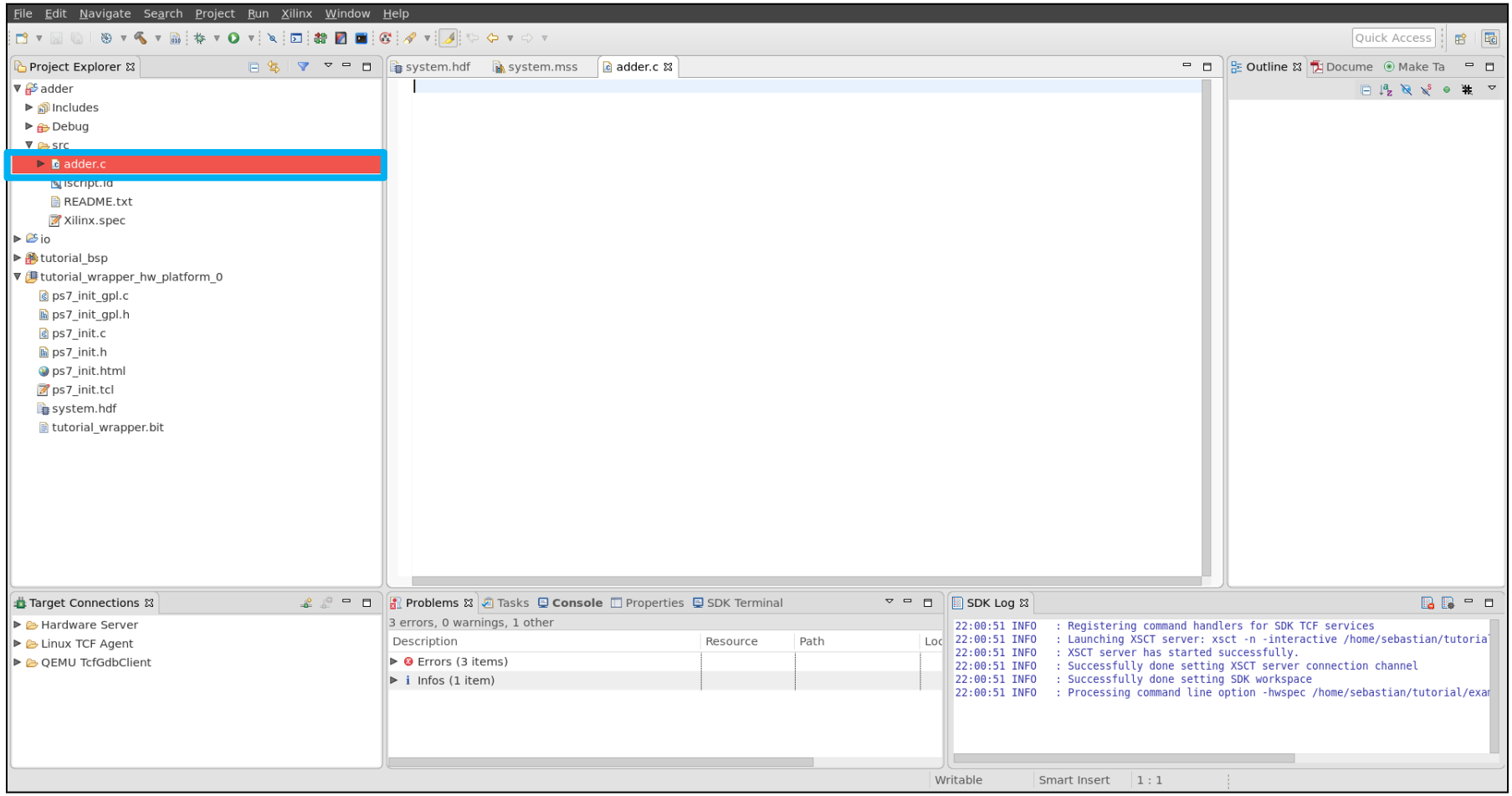
Right-click **`src`** directory of the **adder** application project. Click ***New***. Click ***File***.



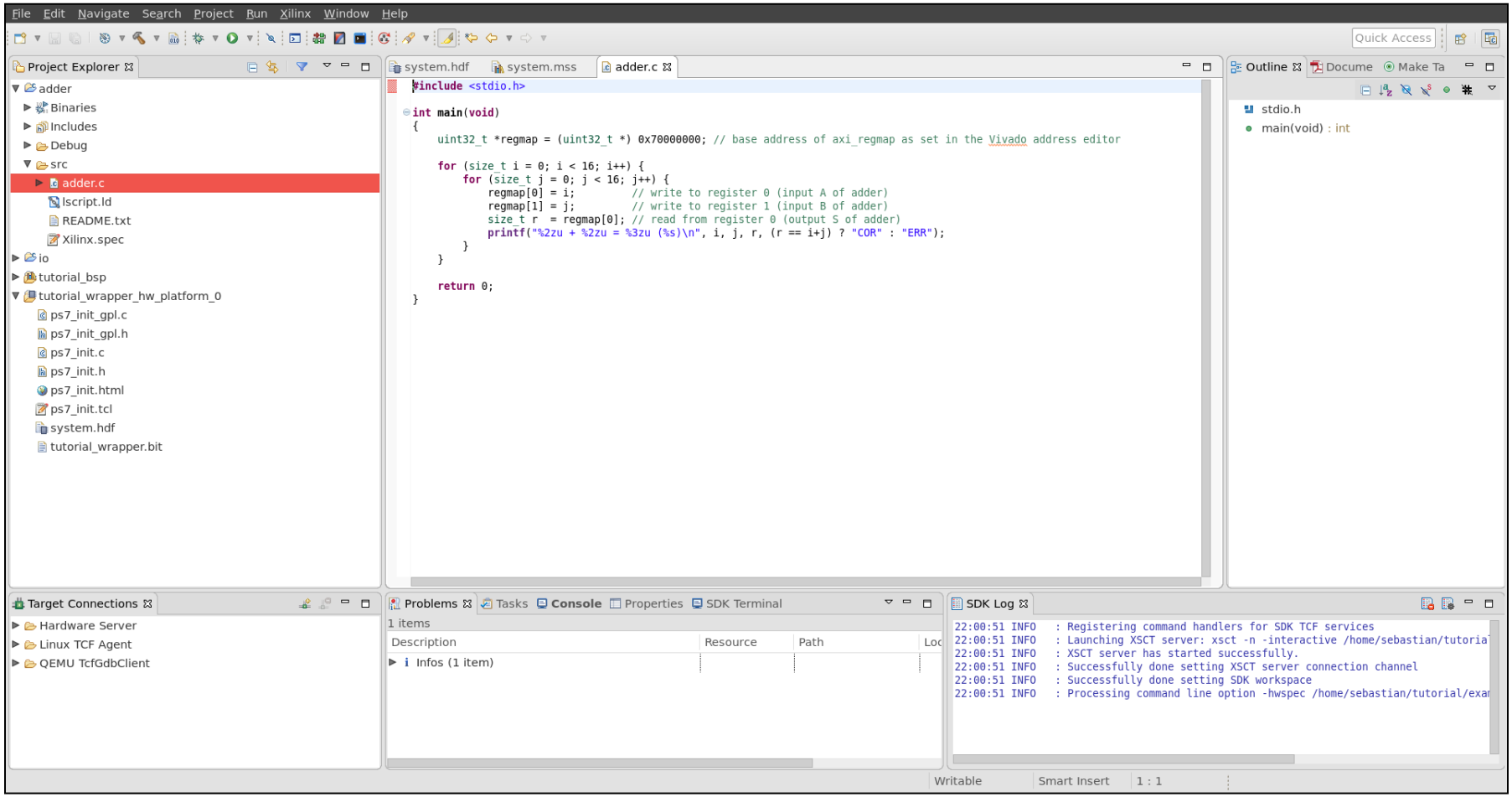
Enter file name **`adder.c`**. Click ***Finish***.



Double-click **`src/adder.c`** file of the **adder** application project to open it in the editor.



Copy-paste contents of **`Lab_1_repo/resources/adder.c`** to **`adder.c`** in the editor.



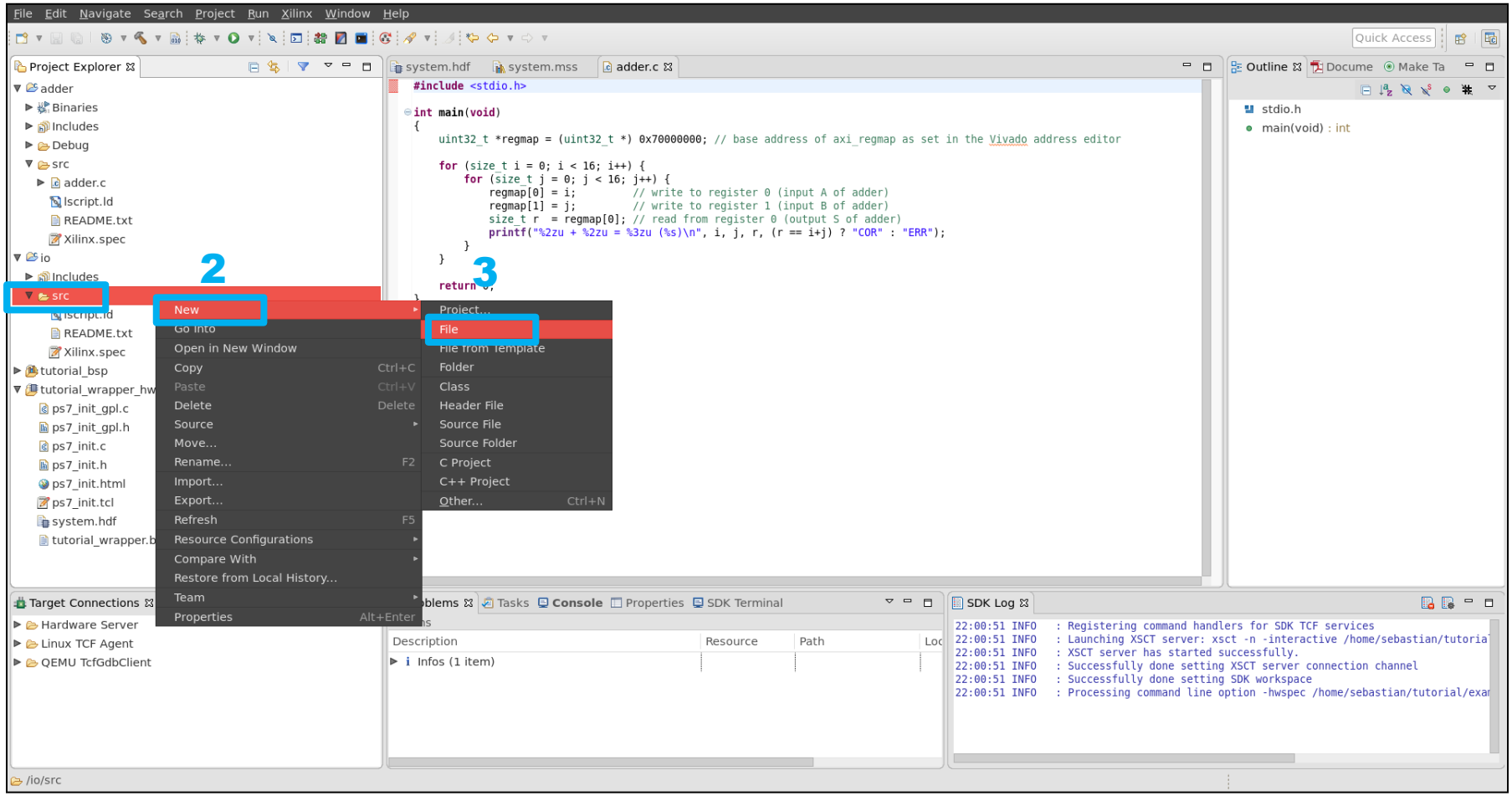
adder.c. The main loop assigns iterators to adder inputs via writes to registers 0 and 1, respectively, and retrieves adder result via reads from register 0. Continue.
NOTE: The base address **0x7000_0000** matches the base address set in Vivado.

```c
#include <stdio.h>

int main(void)
{
    // base address of axi_regmap as set in the Vivado address editor
    uint32_t *regmap = (uint32_t *) 0x70000000;

    for (size_t i = 0; i < 16; i++) {
        for (size_t j = 0; j < 16; j++) {
            regmap[0] = i;          // write to register 0 (REG0_OUT, input A of adder)
            regmap[1] = j;          // write to register 1 (REG1_OUT, input B of adder)
            size_t r  = regmap[0]; // read from register 0 (REG0_IN, output S of adder)
            printf("%2zu + %2zu = %3zu (%s)\n", i, j, r, (r == i+j) ? "COR" : "ERR");
        }
    }

    return 0;
}
```

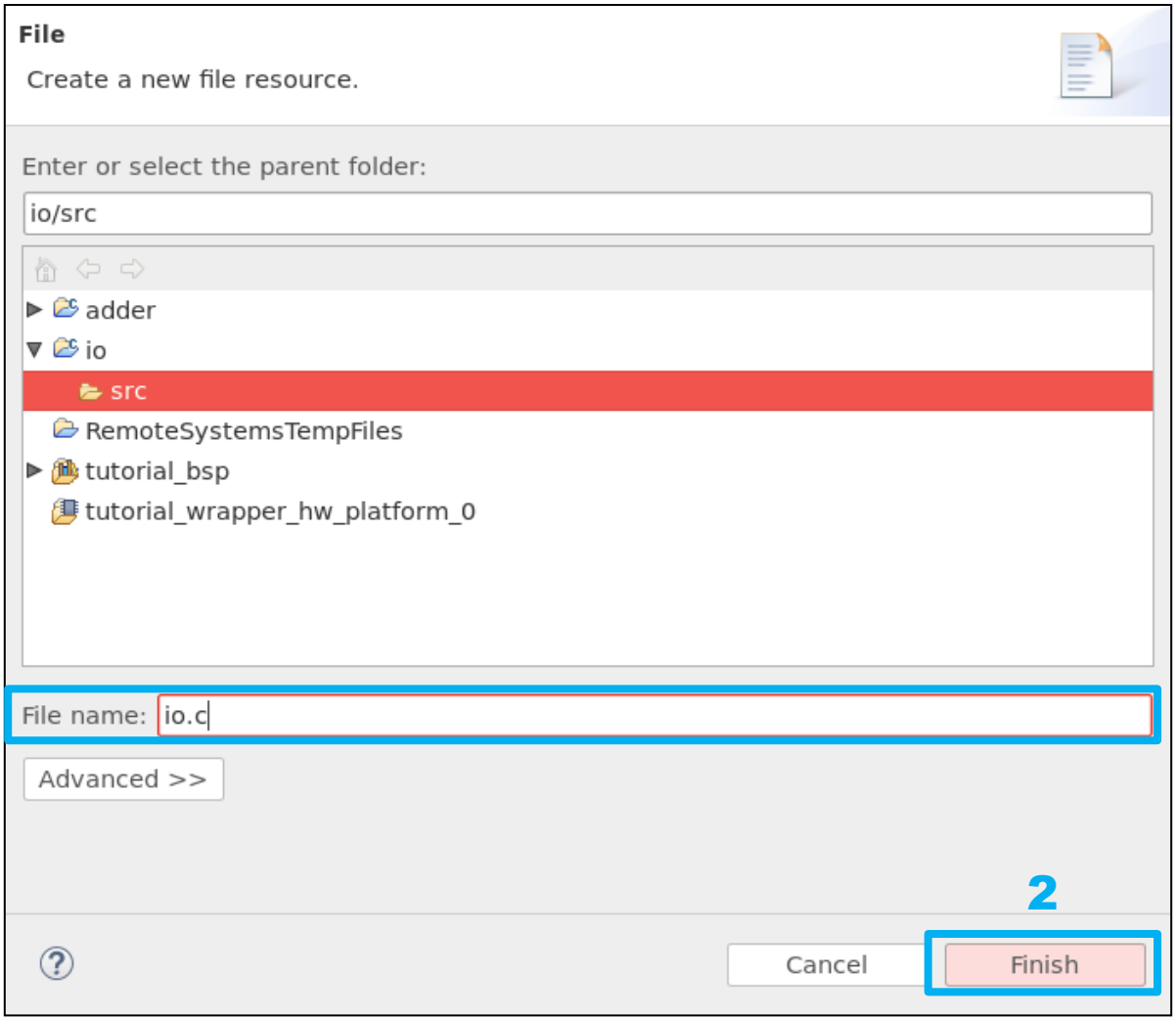
Register Map

Index	Offset	Read Access	Write Access
[0]	0x00	S (adder output)	A (adder input)
[1]	0x04	Zero	B (adder input)
[2]	0x08	BTN	LED

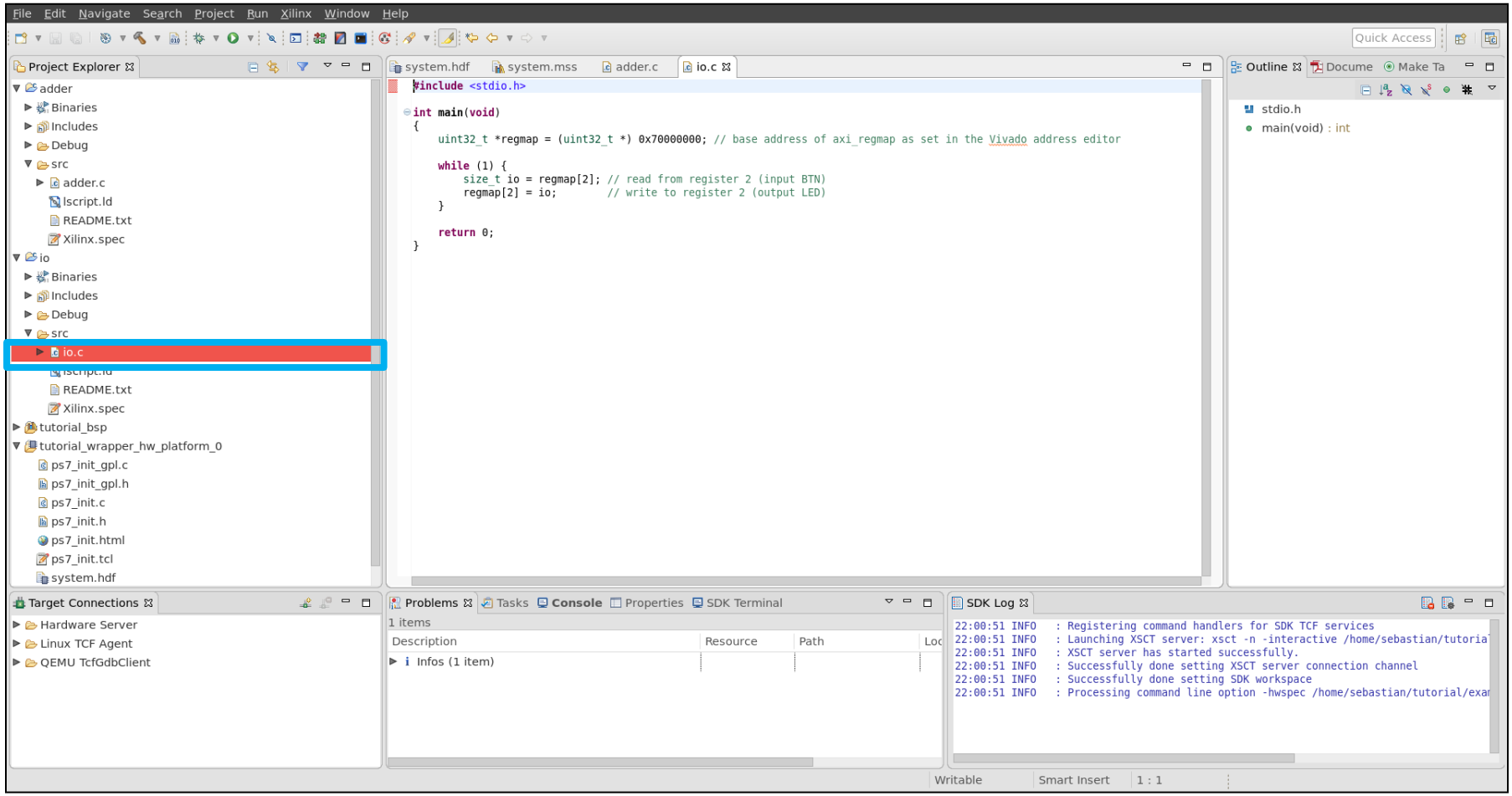
Right-click **`src`** directory of the **io** application project. Click ***New***. Click ***File***.



Enter file name **`io.c`**. Click ***Finish***.



Double-click **src/io.c** file of the **io** application project to open it in the editor. Copy-paste contents of **Lab_1_repo/resources/io.c** to **io.c** in the editor.



io.c. The main loop inputs from buttons via reads from register 2 and outputs to LEDs via writes to register 2. Continue.

```c
#include <stdio.h>

int main(void)
{
    // base address of axi_regmap as set in the Vivado address editor
    uint32_t *regmap = (uint32_t *) 0x70000000;

    while (1) {
        size_t io = regmap[2]; // read from register 2 (REG2_IN, input BTN)
        regmap[2] = io;        // write to register 2 (REG2_OUT, output LED)
    }

    return 0;
}
```

Register Map			
Index	Offset	Read Access	Write Access
[0]	0x00	S (adder output)	A (adder input)
[1]	0x04	Zero	B (adder input)
[2]	0x08	BTN	LED

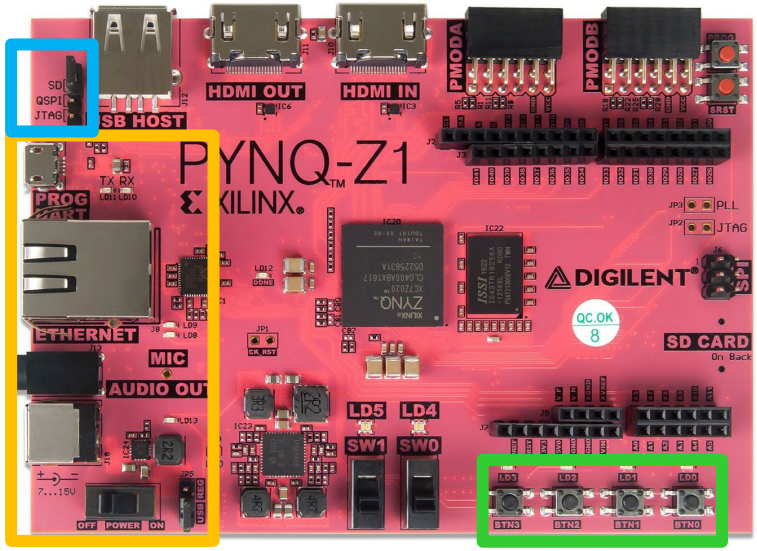
Tutorial Demo

Part 1: Board Setup

Digilent PYNQ-Z1

Boot Mode

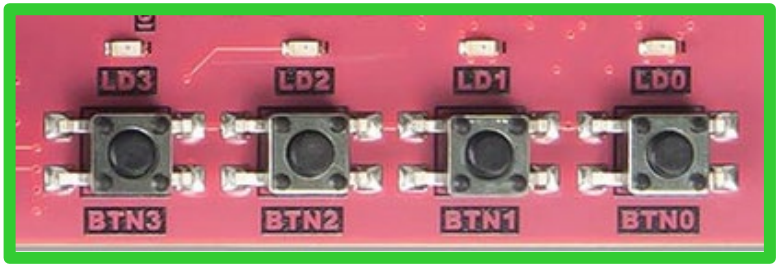
JTAG/UART



Power Switch

Power Source Select

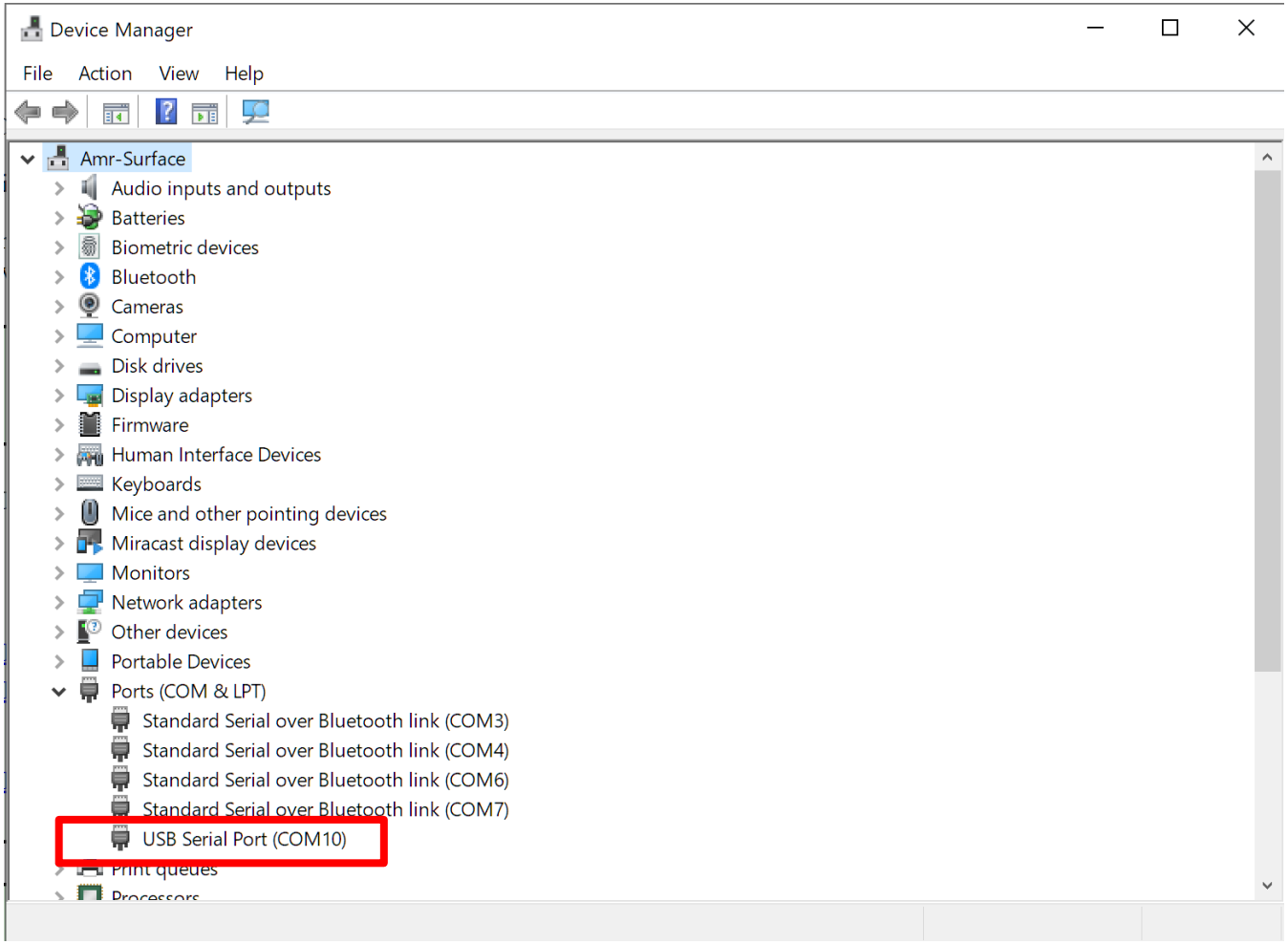
Buttons and LEDs



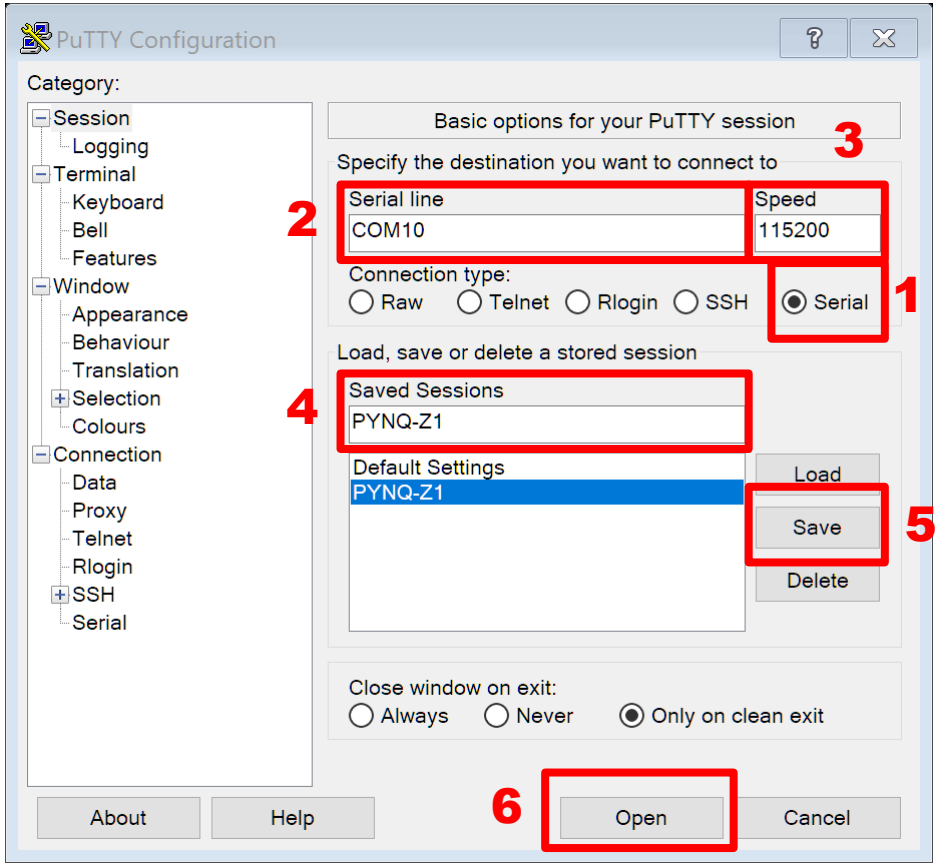
Go to the following link to download PuTTY. It will be used for opening a serial session with the target board.
https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

Connect the board via USB cable and turn it on. Open the device manager and record the COM port that corresponds to the board.

Note: if you can't figure it out, while the device manager window is opened, turn off the board and back on again. Note the COM# added as "USB Serial Port"



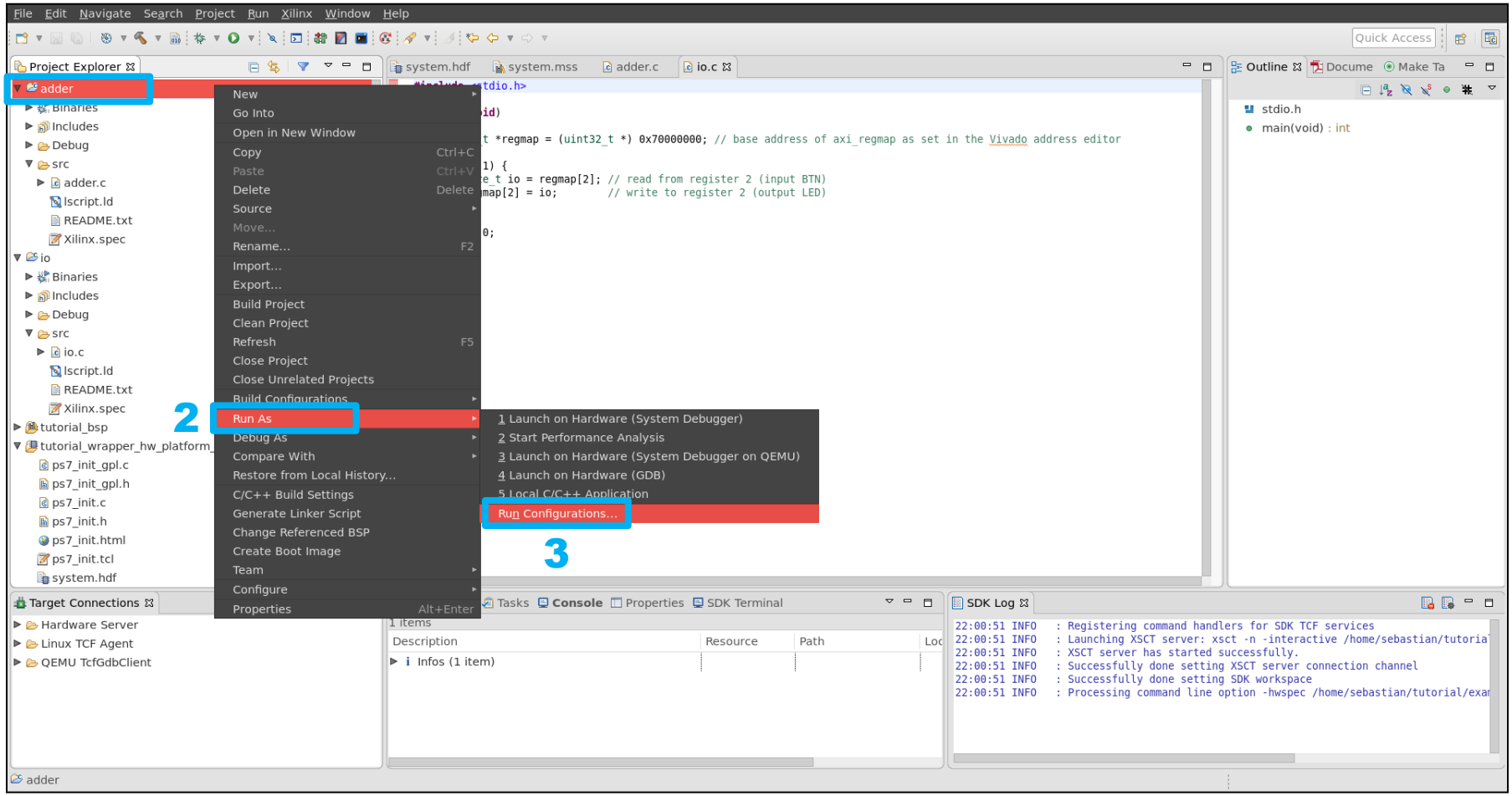
Open PuTTY. Select **Serial**, add **COM#** under **Serial Line**, change **Speed** to **115200**. In order to save those settings, Enter **PYNQ-Z1** under **saved sessions** and click **save**. Click **Open** to open a terminal connected to the board.



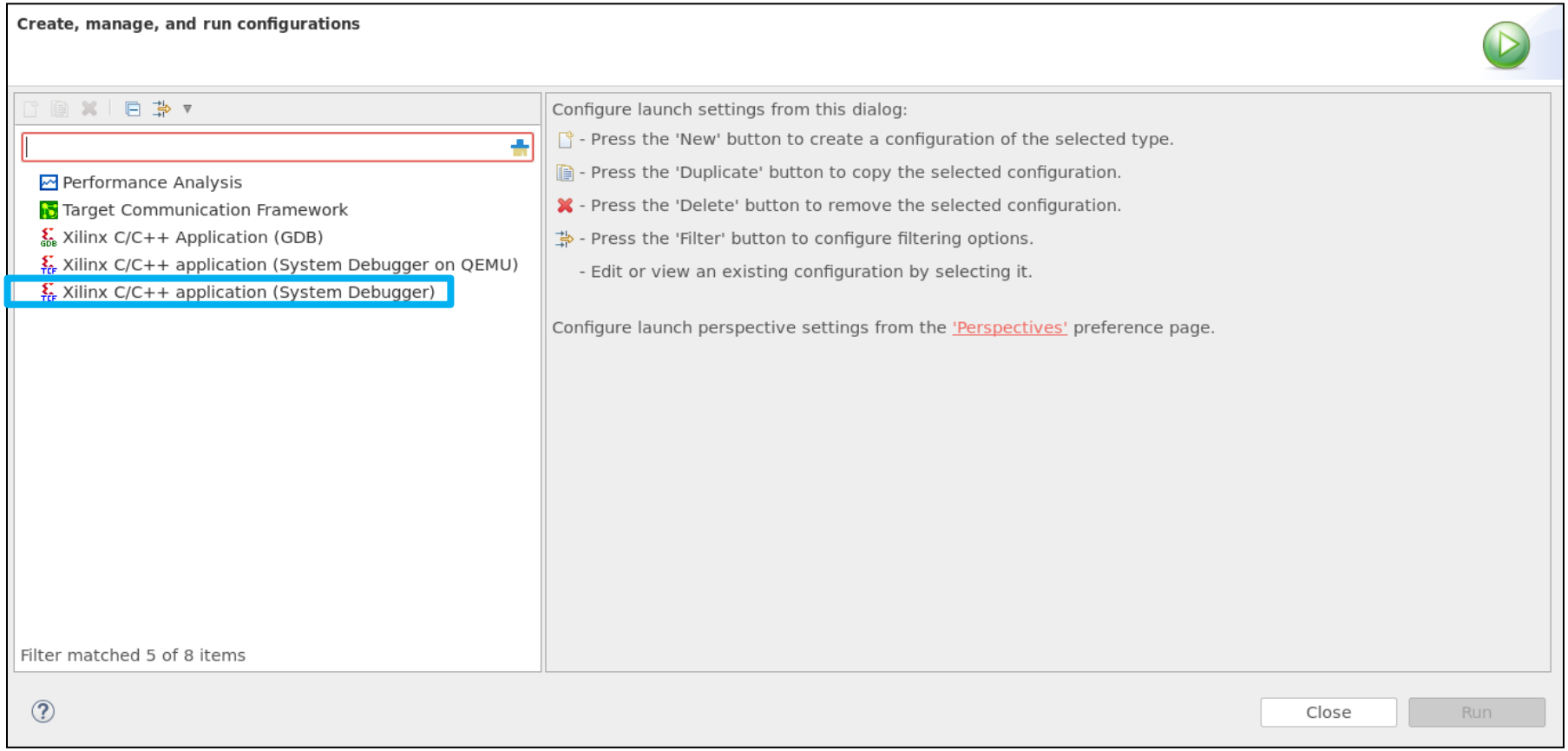
Part 2: Creating a Run Configuration

A run configuration is a profile for how an application project is programmed into the board. We will create one run configuration for each application.

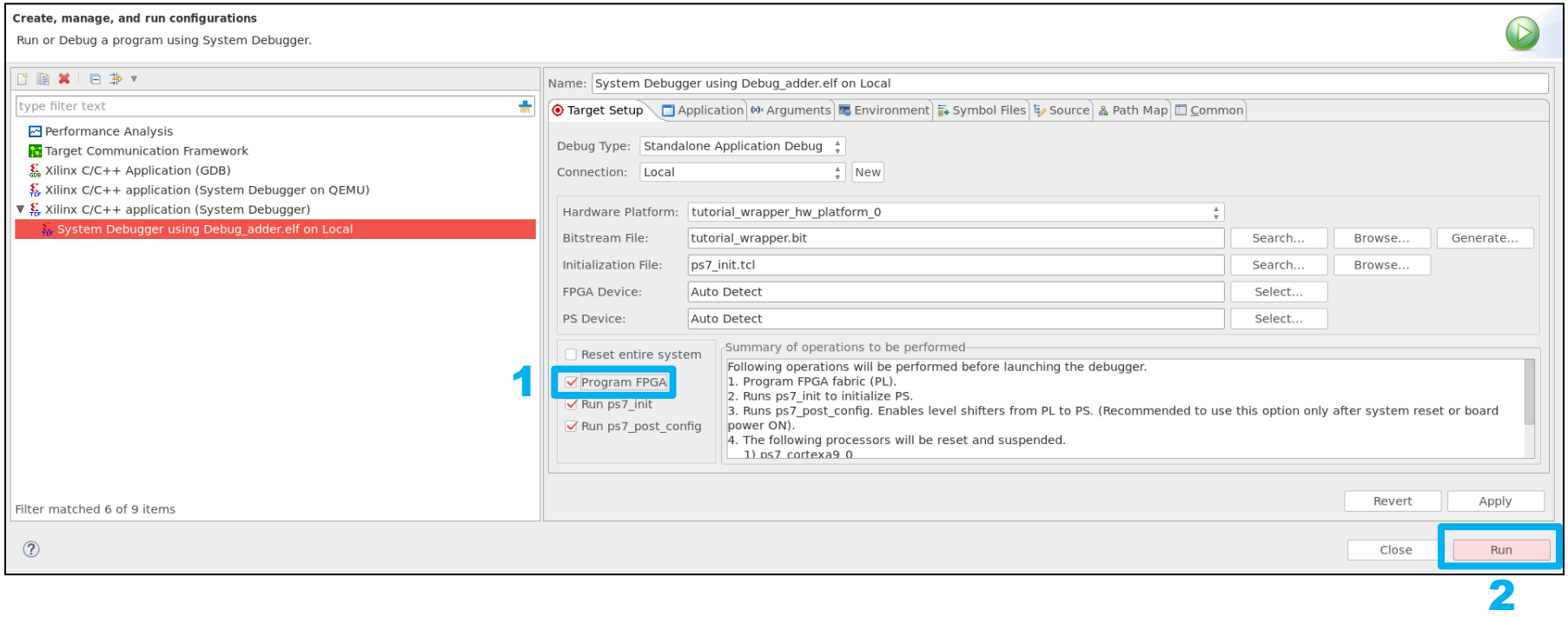
Right-click the **adder** application project. Click ***Run As***. Click ***Run Configurations…***.



Double-click ***Xilinx C/C++ application (System Debugger)***.



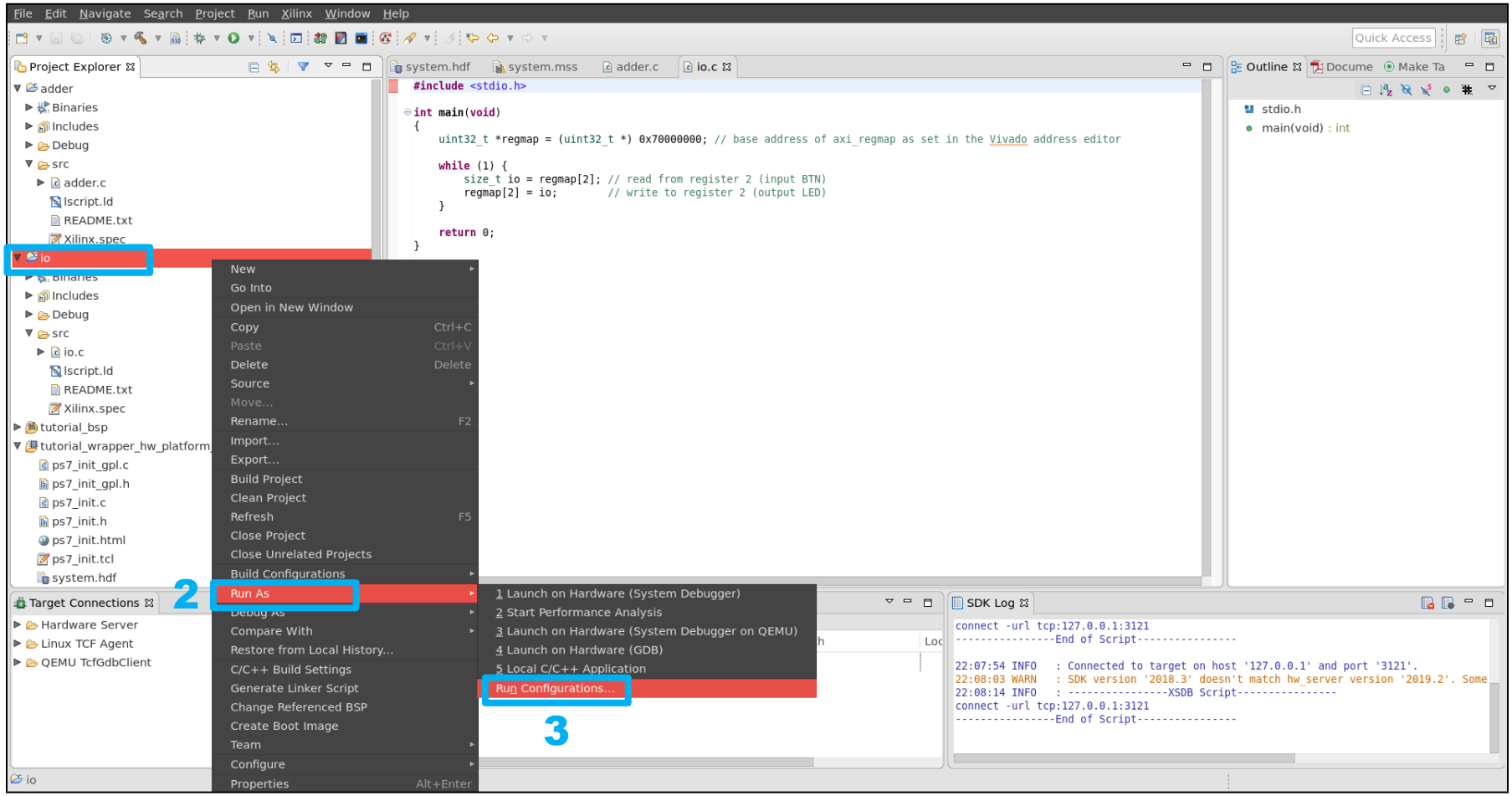
A profile is generated for the **adder** application. Check ***Program FPGA***. Click ***Run***.
NOTE: This will program the FPGA, then load and run the binary application.



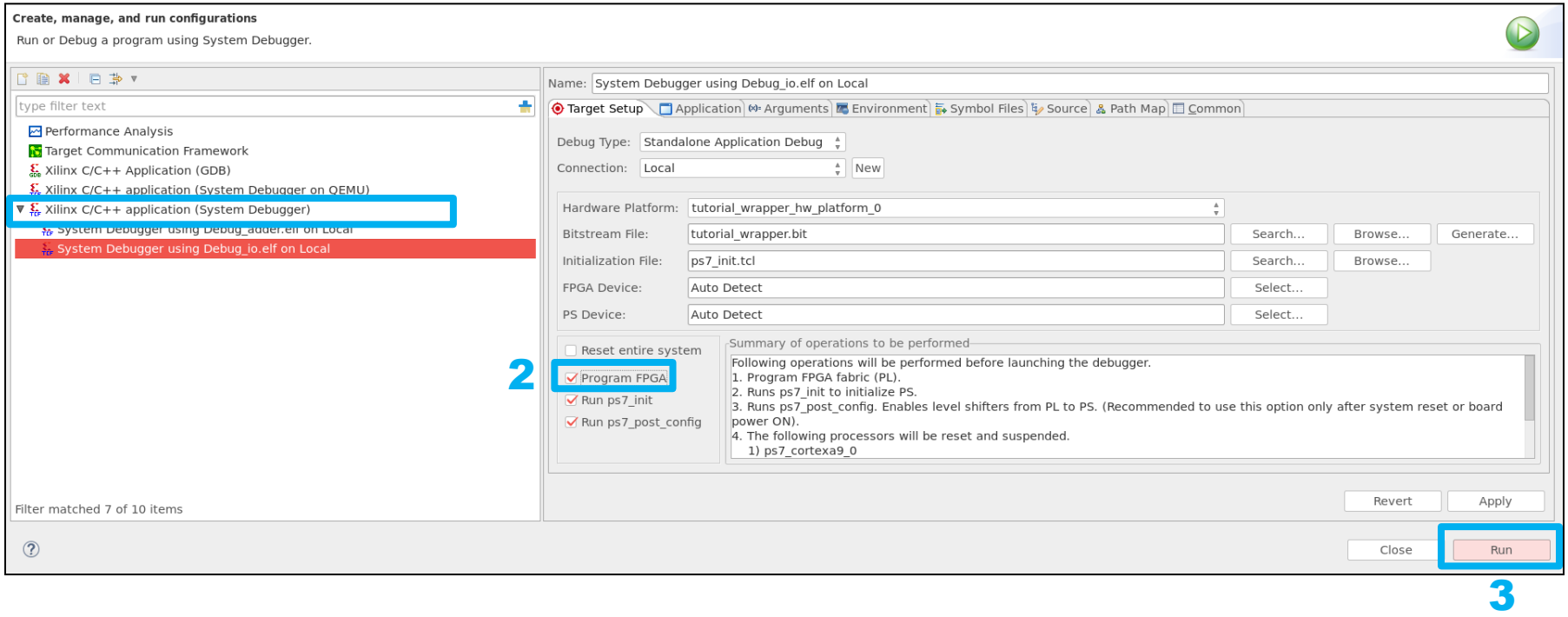
This is a snippet of the printout of the **adder** application. Continue.

```
0 +  0 =   0 (COR)
0 +  1 =   1 (COR)
0 +  2 =   2 (COR)
0 +  3 =   3 (COR)
0 +  4 =   4 (COR)
0 +  5 =   5 (COR)
0 +  6 =   6 (COR)
0 +  7 =   7 (COR)
0 +  8 =   8 (COR)
0 +  9 =   9 (COR)
0 + 10 =  10 (COR)
0 + 11 =  11 (COR)
0 + 12 =  12 (COR)
0 + 13 =  13 (COR)
0 + 14 =  14 (COR)
0 + 15 =  15 (COR)
1 +  0 =   1 (COR)
1 +  1 =   2 (COR)
1 +  2 =   3 (COR)
1 +  3 =   4 (COR)
1 +  4 =   5 (COR)
1 +  5 =   6 (COR)
1 +  6 =   7 (COR)
1 +  7 =   8 (COR)
1 +  8 =   9 (COR)
1 +  9 =  10 (COR)
```

Right-click the **io** application project. Click ***Run As***. Click ***Run Configurations…***.



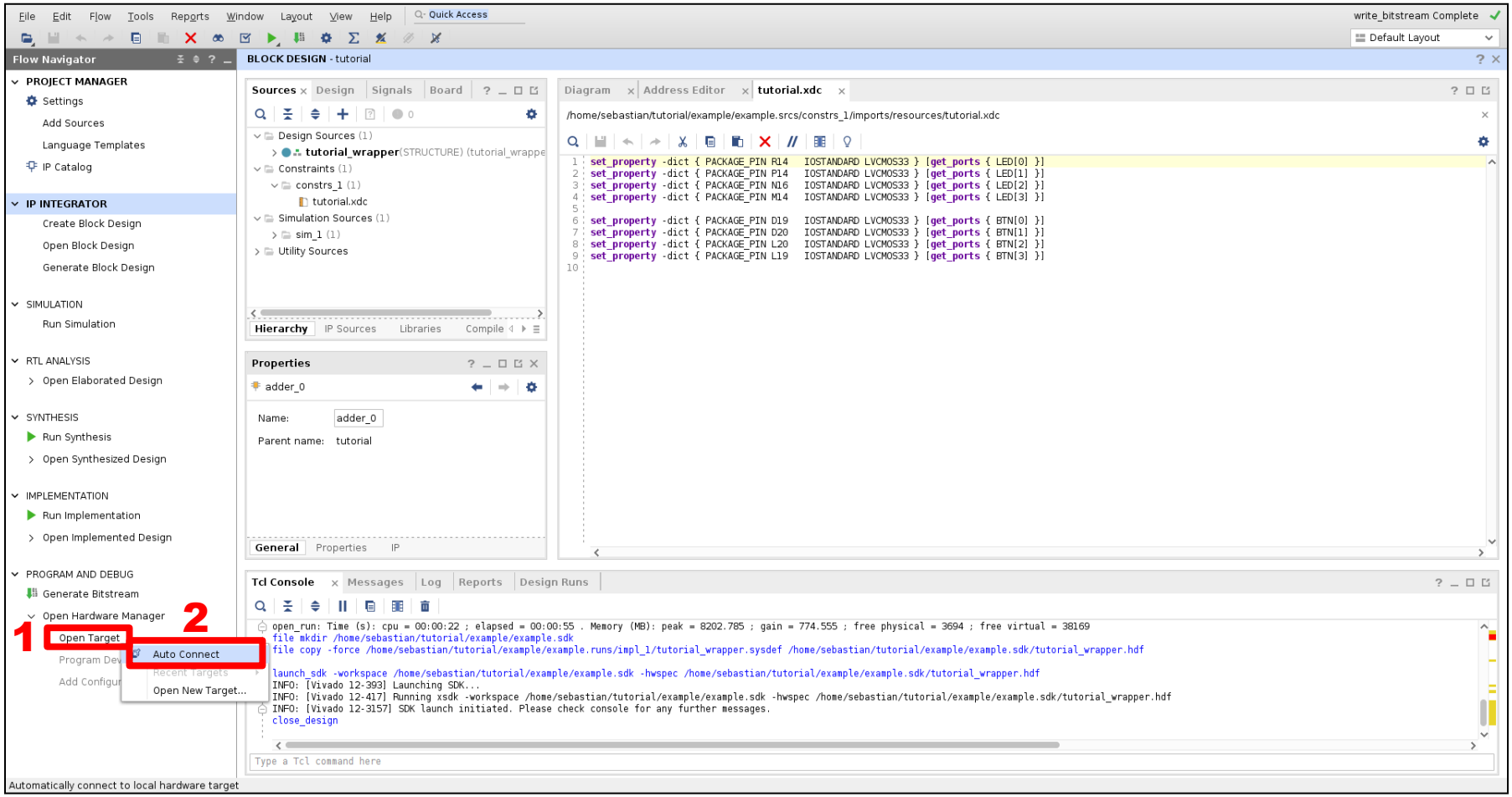
Double-click *Xilinx C/C++ application (System Debugger)*. A profile is generated for the **adder** application. Check ***Program FPGA***. Click ***Run***.
NOTE: This will program the FPGA, then load and run the binary application.



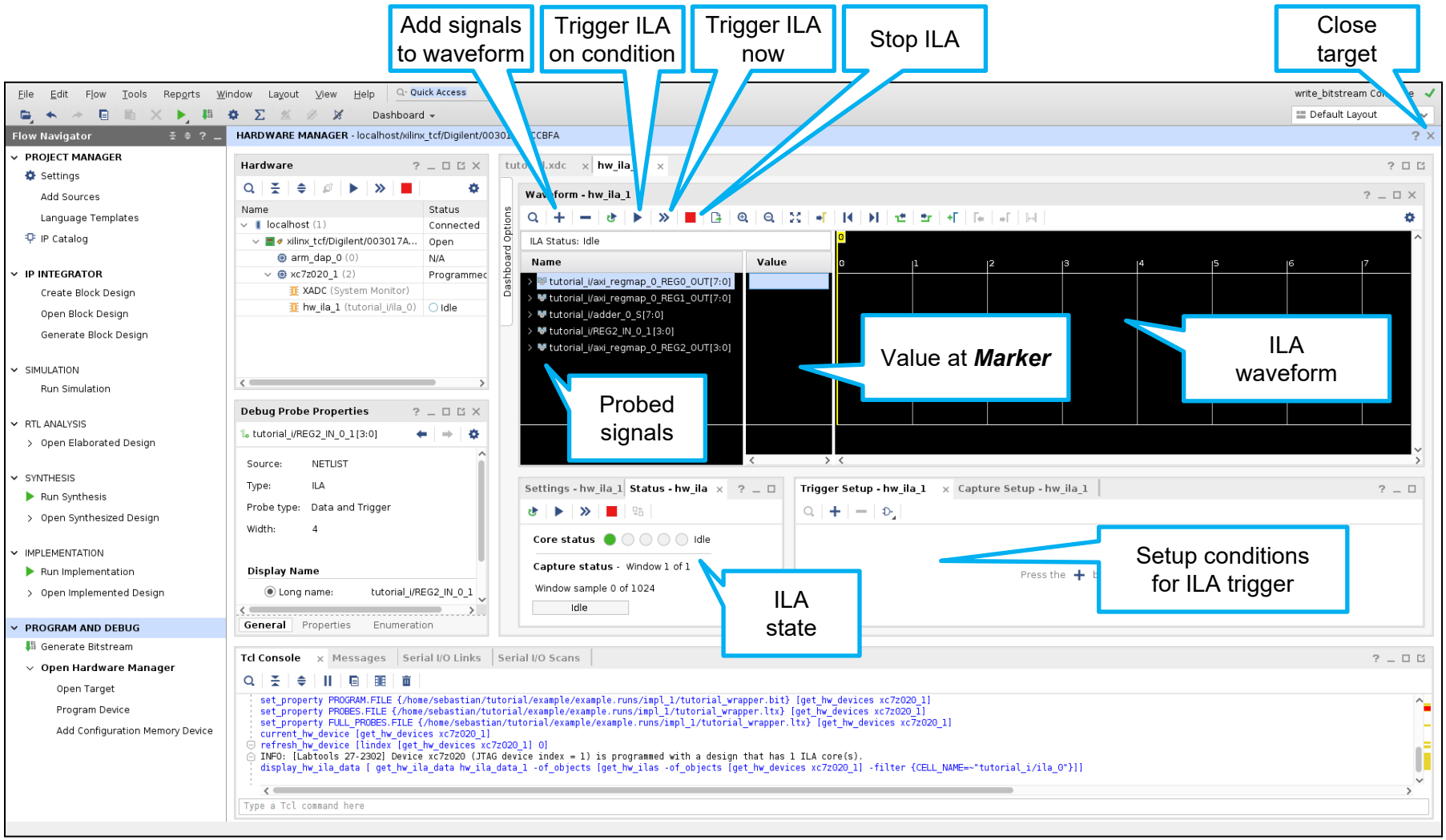
Part 3: Using the Integrated Logic Analyzer

The ILA is a powerful debugging tool that allows internal signals to be probed and analyzed at runtime. We will demonstrate the ILA for the **io** app (the app assumed to be running from the previous step).

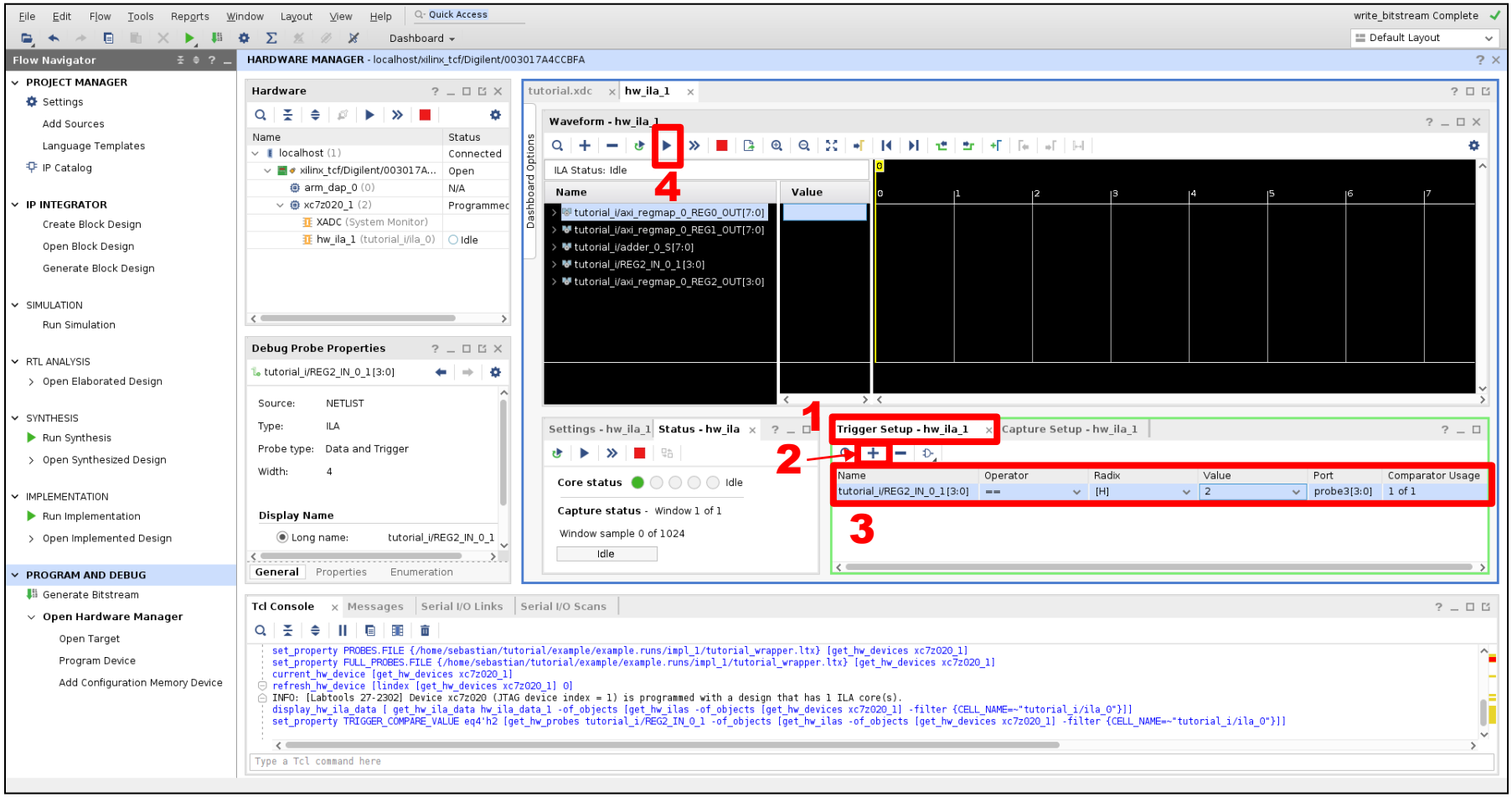
Go back to **Lab_1** project Instance. Click ***Open Target***. Click ***Auto Connect***.



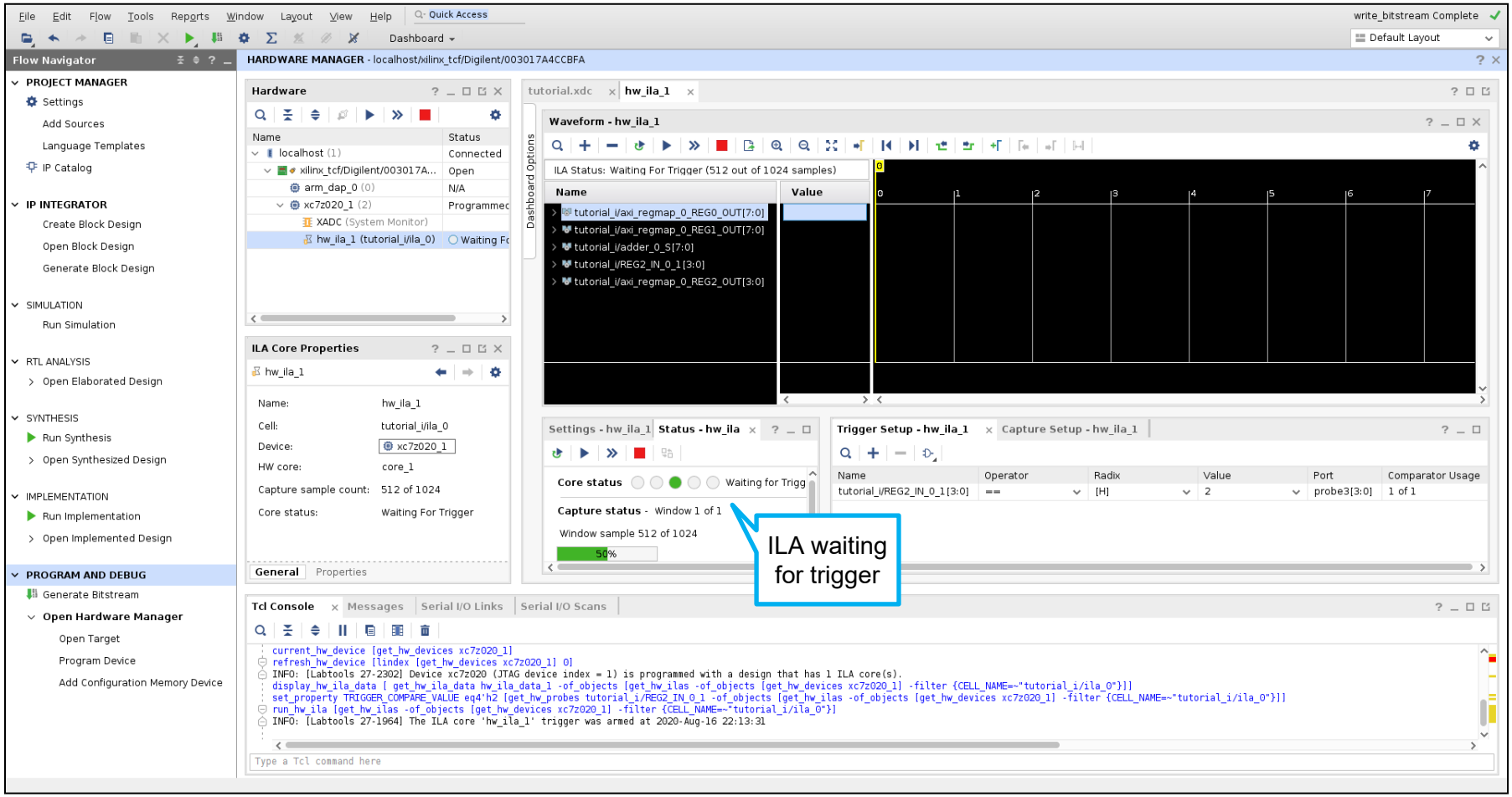
This is the ILA GUI. Continue.



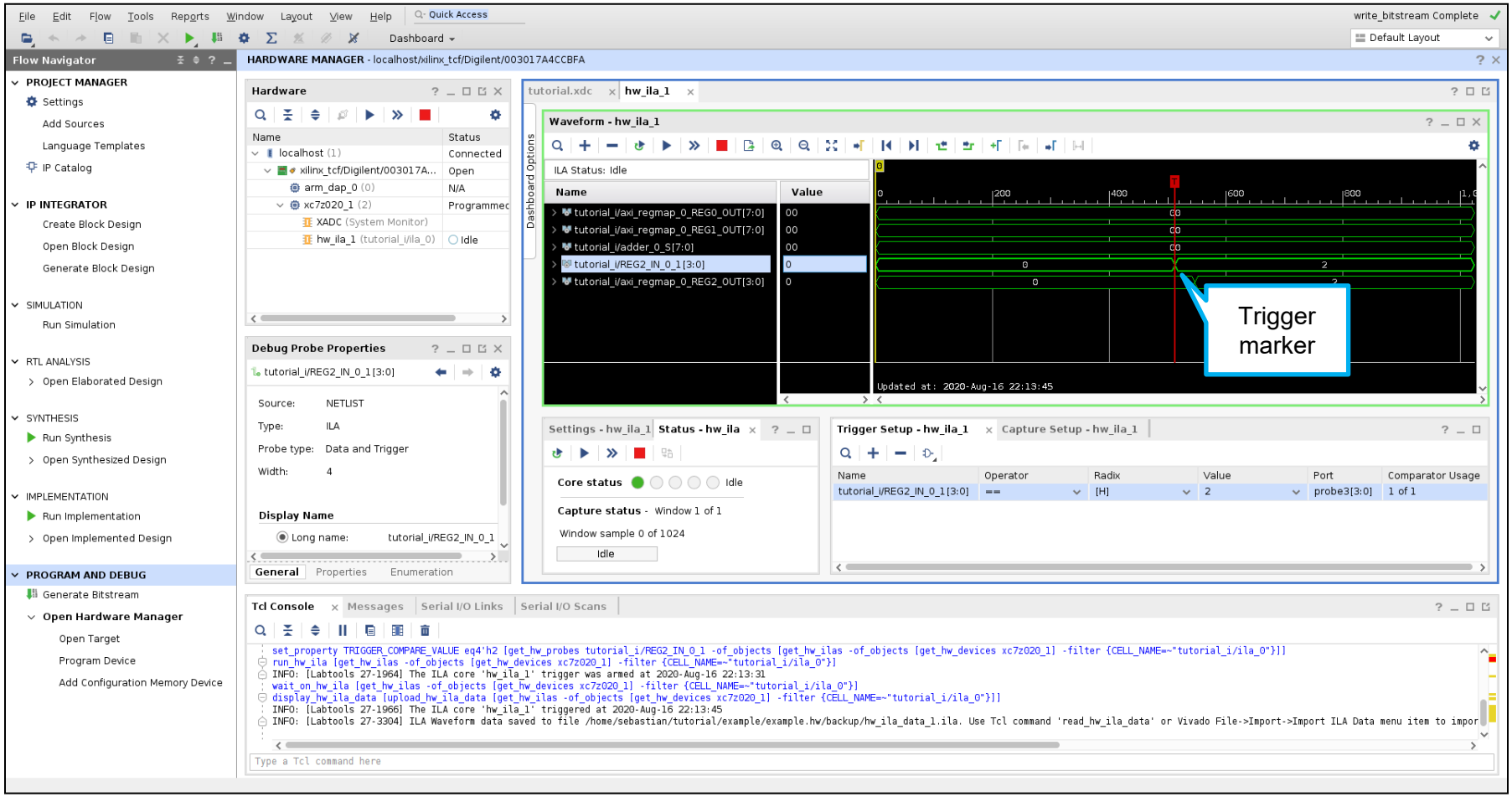
Click the ***Trigger Setup – hw_ila_1*** tab. Click ***+ (Add probe(s))*** and add the net ***REG2_IN***. Set the trigger parameters ***Operator*** → **==** and ***Value*** → **2**. Click ►. This means that the ILA will starting capturing the probed signals once the trigger condition happens.



ILA is waiting for trigger condition. Continue.



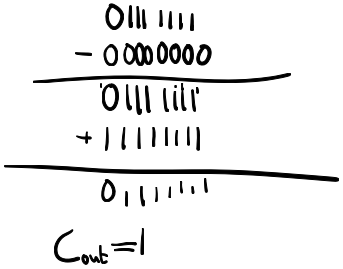
Set board **BTN[3:0] → 0010 (Value = 2)** to trigger ILA. (Click on **BTN1)**

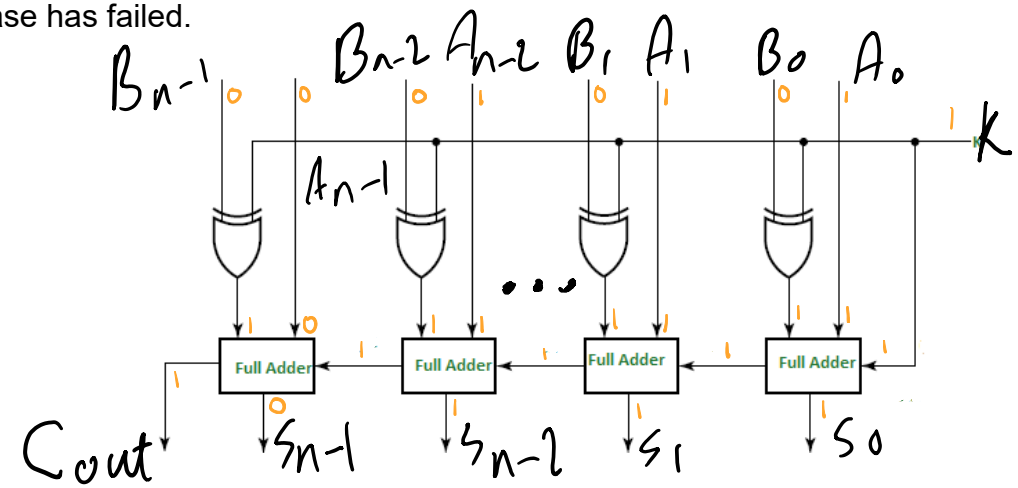


What to do

$-0x80000000 = -2^{n-1}$
$0x7FFF FFFF = 2^{n-1}-1$

- Design a generic 32 bit adder/subtractor using:
  - Generic statement
  - For-Generate statement
- Write a testbench to fully test your VHDL design, verify the functionality using the behavioral simulation. (refer to assert_tb.vhd)
- Write C code to fully test your design. Make sure to cover both the addition and subtraction cases. (use random numbers. Refer to this blog for more information)
  Note: no need to print to the terminal here, as it will keep printing for a very long time. Instead, only print statement, when you pass all the test cases, or if any test case has failed.





https://www.geeksforgeeks.org/4-bit-binary-adder-subtractor/

Deliverables

- You will be delivering all the VHDL and C codes that you have written, including the design files, testbenches, and the C codes. The submission will on Canvas Assignment for Lab 1.

- If you Want, you can also just Zip the top Vivado project folder that has all the files that was mentioned above. (please make sure that it has all the files required).

- Check offs will be done on the due date for this assignment. The rubric for the check-offs will be provided later, but you should expect that we would:
  - Look at your design files and make sure that you used the required design method.
  - Check the VHDL testbench simulation.
  - Check the C code and make sure it has enough test cases.
  - Check that you can generate a bitstream and configure the design on the board.

End of Lab 1