PYNQ Acceleration Kernels Tutorial using Vitis HLS and Vivado flow

Digital System Design Lab, National Formosa University

April, 2022

Contents

Introduction

This documentation guides user how to build acceleration kernels running on PYNQ framework for Xilinx platforms. Xilinx Vitis HLS will be used to synthesize the kernel and export it as a RTL intellectual property core. Then Xilinx Vivado will be used to create the system block design and generate the bitstream which will be realized in Programmable Logic. An example Jupyter notebook will be provided to show how we use the acceleration kernel. This approach makes the acceleration kernel reusable on various Xilinx devices using PYNQ framework.

Material files are included with this document. They are:

1. resize_accel.cpp, rgb2yuv4_accel: 2 acceleration kernel source code files using Vitis Vision Library (https://xilinx.github.io/Vitis_Libraries/vision/2020.2/index.html).
2. resize_test.ipynb, rgb2yuv4_test.ipynb: 2 Jupyter notebooks example demonstrating how to use the kernel on PYNQ framework.
3. board.jpg: a sample image.

Setup Workflow (run once before development)

First, setup a development system.
Second, setup a host machine.
Third, setup a target device.

Development Flow

First, synthesize an acceleration kernel using Vitis HLS on the host machine.
Second, create a system block design and generate the bitstream using Vivado on the host machine.
Third, run an example.

Host Machine Requirements

Recommendation: Intel Core i5, 16GB RAM, 200 GB SSD.

Supported Operating Systems on Host Machine

Xilinx® supports the following operating systems on x86 and x86-64 processor architectures.
Microsoft Windows Professional/Enterprise 10.0 1809 Update; 10.0 1903 Update; 10.0 1909 Update; 10.0 2004 Update
Red Hat Enterprise Workstation/Server 7.4 - 7.8, and 8.2 (64-bit), English/Japanese
CentOS 7.4 - 7.8, and 8.2 (64-bit), English/Japanese
SUSE Linux Enterprise 12.4 (64-bit), English/Japanese
Ubuntu Linux 16.04.5 LTS;16.04.6 LTS; 18.04.1 LTS; 18.04.2 LTS, 18.04.3 LTS; 18.04.4 LTS; and 20.04 LTS (64-bit), English/Japanese.

This tutorial was made on **Ubuntu Desktop 18.04.4 LTS**.

Target Device

1. Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit (XCZU7EV-2FFVC1156).
2. TUL Embedded PYNQ ™ -ZU Board (XCZU5EG-1SFVC784)

Reading Guide

1. Follow section 1, 2, 3 to setup the development system. Perform this setup once before development phase.

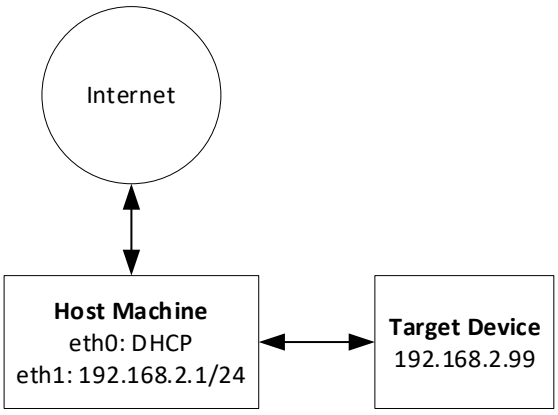
2. Follow section 4 to getting started with the first example running on ZCU104: resize. We suggest practicing this section several times to well understand the development flow.

3. Follow section 5 to continue with the second running on ZCU104: rgb2yuv4.

4. Follow section 6 to make the resize example running on other Zynq UltraScale+ MPSoC devices.

1. System Setup

We will setup the development system as shown in this diagram:



On Host Machine, we can use an USB to Ethernet adapter to implement the eth1 interface. Set it to 192.168.2.1/24 that we can connect to the target device through its default IP address.

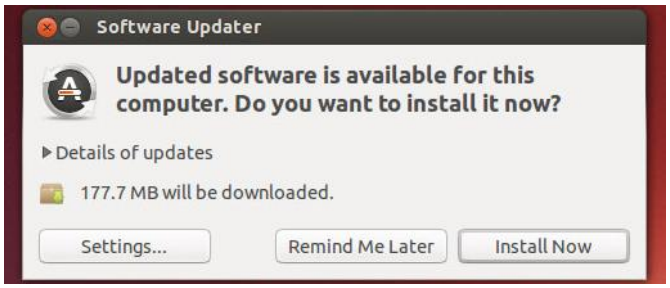
2. Host Machine Setup

2.1. Update Ubuntu APT and install some tools

Open terminal and type these commands

```
sudo apt-get update
sudo apt-get install build-essential git gtkterm
```

If Software Update dialog appears, click "Remind Me Later". Avoid using "sudo apt-get upgrade" command. Newer Ubuntu versions than 16.04.6, 18.04.4 does not work.



2.2. Install Vitis Unified Software 2020.2

Step 1: Open web browser to download installation file (see below link) and save it as ~/Xilinx_Unified_2020.2_1118_1232_Lin64.bin.
Official direct download link:

https://www.xilinx.com/member/forms/download/xef.html?filename=Xilinx_Unified_2020.2_1118_1232_Lin64.bin

Or go to Xilinx download page: https://www.xilinx.com/support/download.html, choose version 2020.2 and download "Xilinx Unified Installer 2020.2: Linux Self Extracting Web Installer (BIN - 354.08 MB)".

Step 2: You will be asked for a Xilinx account. If you don't have, just create a new account, then log in.

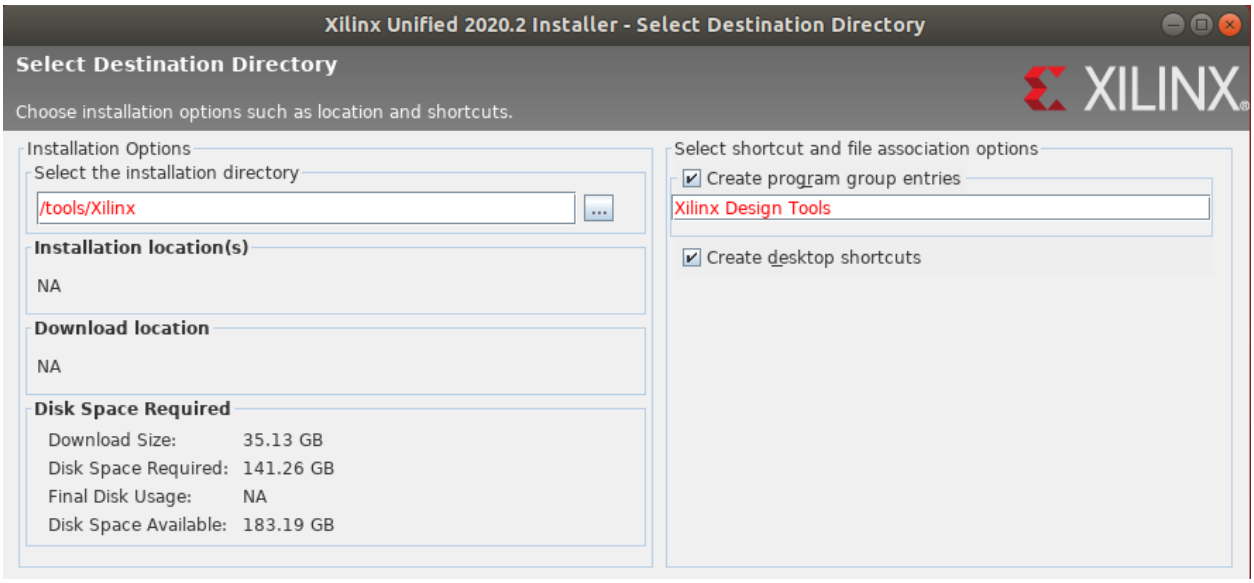
Step 3: You will be asked for U.S. Government Export Approval. Fill in your information then scroll down to click "Download".

Step 4: When the download is finished, open the terminal, type these commands to launch the installer.

```
cd ~
chmod 755 Xilinx_Unified_2020.2_1118_1232_Lin64.bin
mkdir Xilinx_Unified_2020.2
sudo ./Xilinx_Unified_2020.2_1118_1232_Lin64.bin --target Xilinx_Unified_2020.2
```

Step 5: Installation Wizard

- "Xilinx Unified 2020.2 Installer - Welcome" dialog will display, click Next.
- On "Select Install Type" dialog, choose "Download and Install Now", then click Next.
- On " Select Product to Install" dialog, choose "Vitis", then click Next.
- On " Vitis Unified Software Platform" dialog, keep the default, then click Next.
- On " Accept License Agreements" dialog, select all "I Agree", then click Next.
- On "Select Destination Directory" dialog, set the installation path to /tools/Xilinx.
- A dialog might appear to ask for creating new directory for the installation, choose "Yes".
- On "Installation Summary" dialog, review the information again, click "Install" and wait until the installation is completed.



2.3. Install balenaEtcher

Open terminal and type these commands:

```
curl -1sLf \
  'https://dl.cloudsmith.io/public/balena/etcher/setup.deb.sh' \
  | sudo -E bash
sudo apt-get update
sudo apt-get install balena-etcher-electron
```

3. Target Device Setup

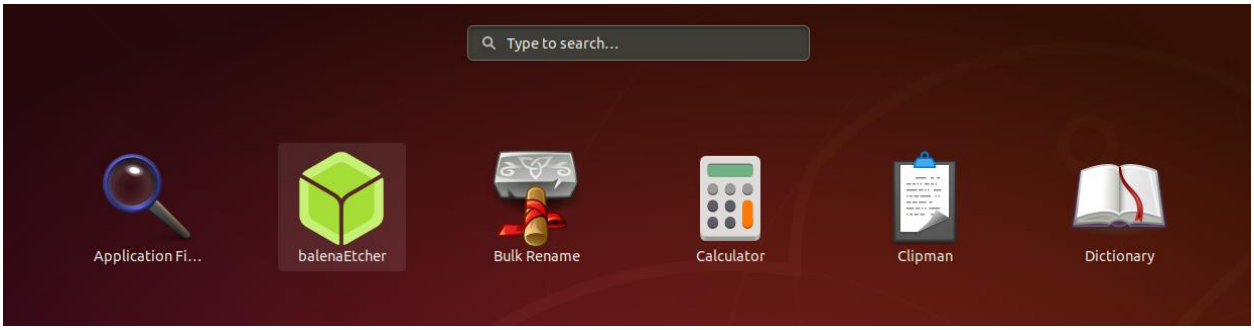
3.1. Download PYNQ image v2.7

Download the PYNQ image version 2.7 for ZCU104 from this official link https://bit.ly/zcu104_2_7, save it as $HOME/zcu104_v2.7.0.zip.
Or search for **ZCU104 v2.7.0 SDCard image** from https://github.com/Xilinx/PYNQ/releases.

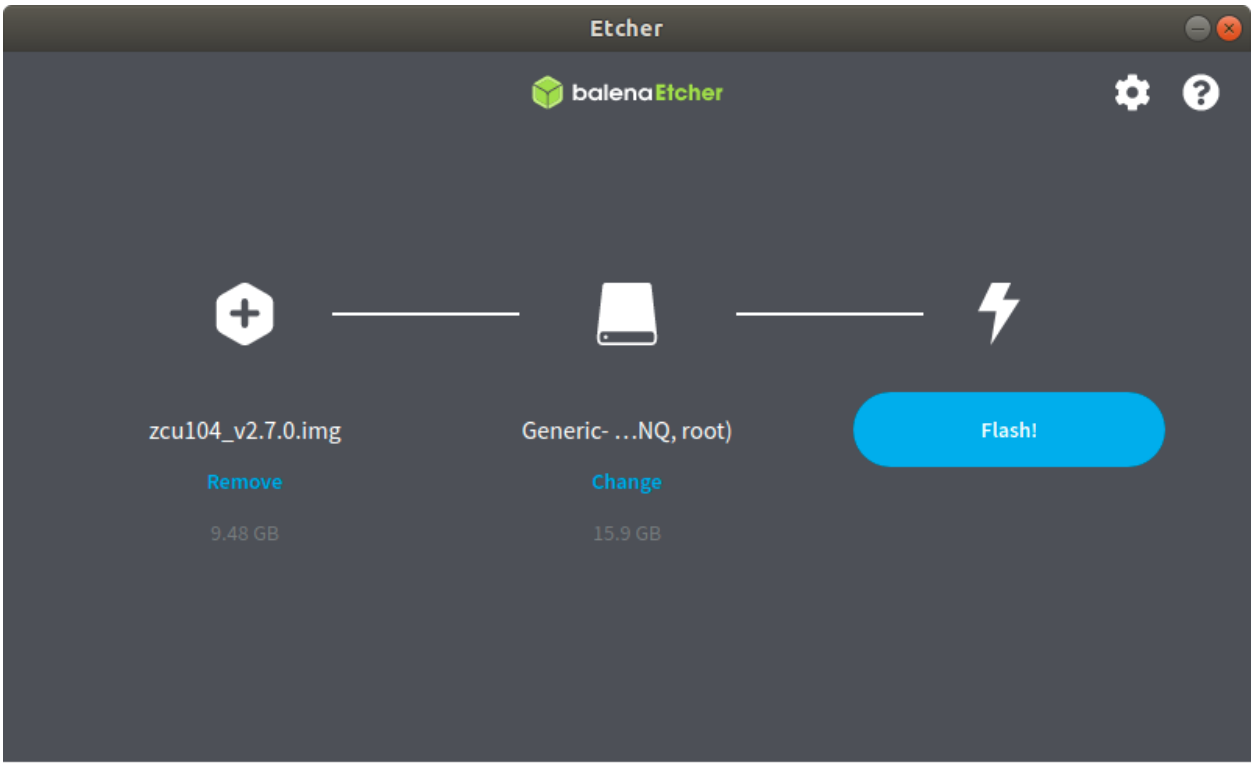
3.2. Flash the PYNQ image onto SD card

First, prepare a minimum 16 GB SD card then plug it to the host machine, then following below steps.

Step 1: Launch the balenaEtcher from Application Dock



Step 2: Click "Flash from file" to open the zcu104_v2.7.0.zip.
Step 3: Click "Select target" to select the SD card. Be careful! Choosing wrong SD card will lose all your data.



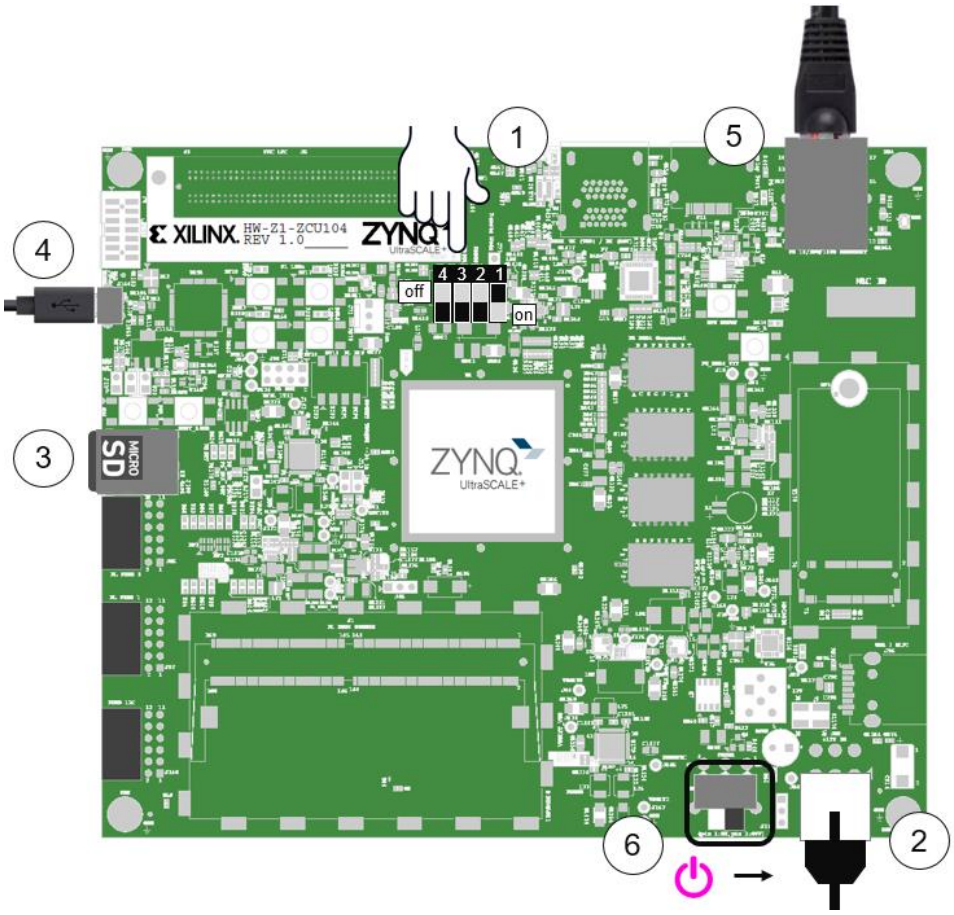
Step 4: Click "Flash".
Step 5: A dialog will appear to ask for root password, type the password and press Enter to start flashing.

3.3. Hardware Configuration

3.3.1. Prerequisites

ZCU104 board
Computer with compatible browser (Supported Browsers)
Ethernet cable
Micro USB cable (optional)
Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)

3.3.2. Board Setup



(1) Set the Boot Dip Switches (SW6) to the following positions:
(This sets the board to boot from the Micro-SD card)
Dip switch 1 (Mode 0): On (down position in diagram)
Dip switch 2 (Mode 1): Off (up position in diagram)
Dip switch 3 (Mode 2): Off (up)
Dip switch 4 (Mode 3): Off (up)
(2) Connect the 12V power cable. Note that the connector is keyed and can only be connected in one way.
(3) Insert the Micro SD card loaded with the appropriate PYNQ image into the MicroSD card slot underneath the board
(4) (Optional) Connect the USB cable to your PC/Laptop, and to the USB JTAG UART MicroUSB port on the board
(5) Connect the Ethernet port by following the instructions below
(6) Turn on the board and check the boot sequence by following the instructions below

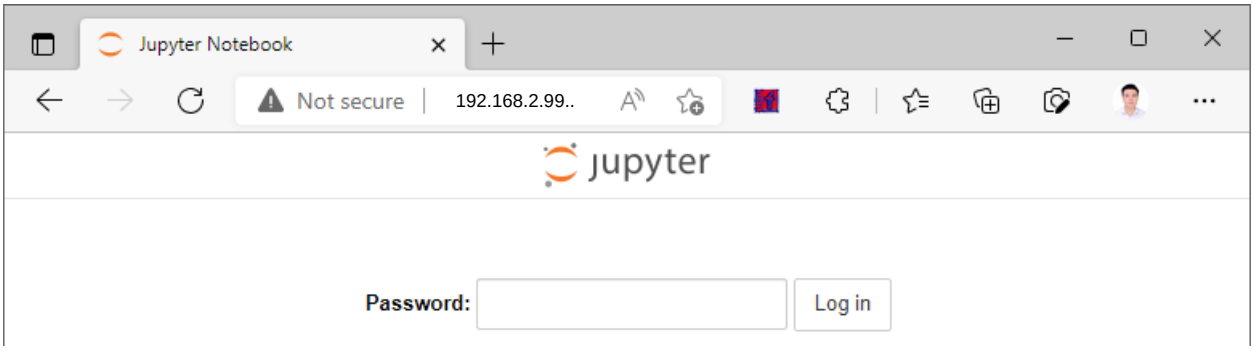
3.3.3. Turning On the ZCU104

As indicated in step 6, slide the power switch to the ON position to turn on the board. A Red LED and some additional yellow board LEDs will come on to confirm that the board has power.

After a few seconds, the red LED will change to Yellow. This indicates that the bitstream has been downloaded and the system is booting.

3.4. JupyterLab

Open web browser on host machine, navigate to 192.168.2.99. Type "xilinx" for the password.



Jupyter home page

4. The first example: Resize

4.1. Project Structure

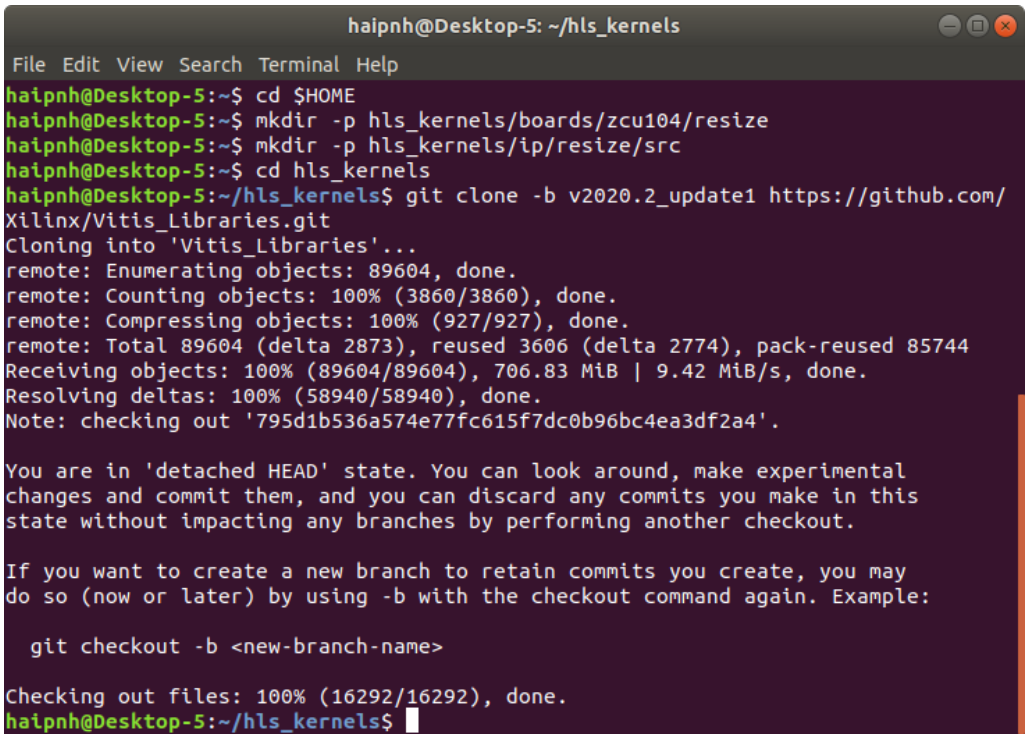
At $HOME directory, we will create the project having structure as following:

```
hls_kernels
├──── boards
│       └──── zcu104
│               └──── resize: Vivado project's workspace
├──── ip
│       └──── resize: Vitis HLS project's workspace
│               └──── src: Kernel source code
└──── Vitis_Libraries
        ├──── vision: Vitis Vision library
        └──── <other domains>
```

4.2. Acceleration Kernel Synthesis using Vitis HLS

Step 1: Open terminal and run these commands:

```
cd $HOME
mkdir -p hls_kernels/boards/zcu104/resize
mkdir -p hls_kernels/ip/resize/src
cd hls_kernels
git clone -b v2020.2_update1 https://github.com/Xilinx/Vitis_Libraries.git
```

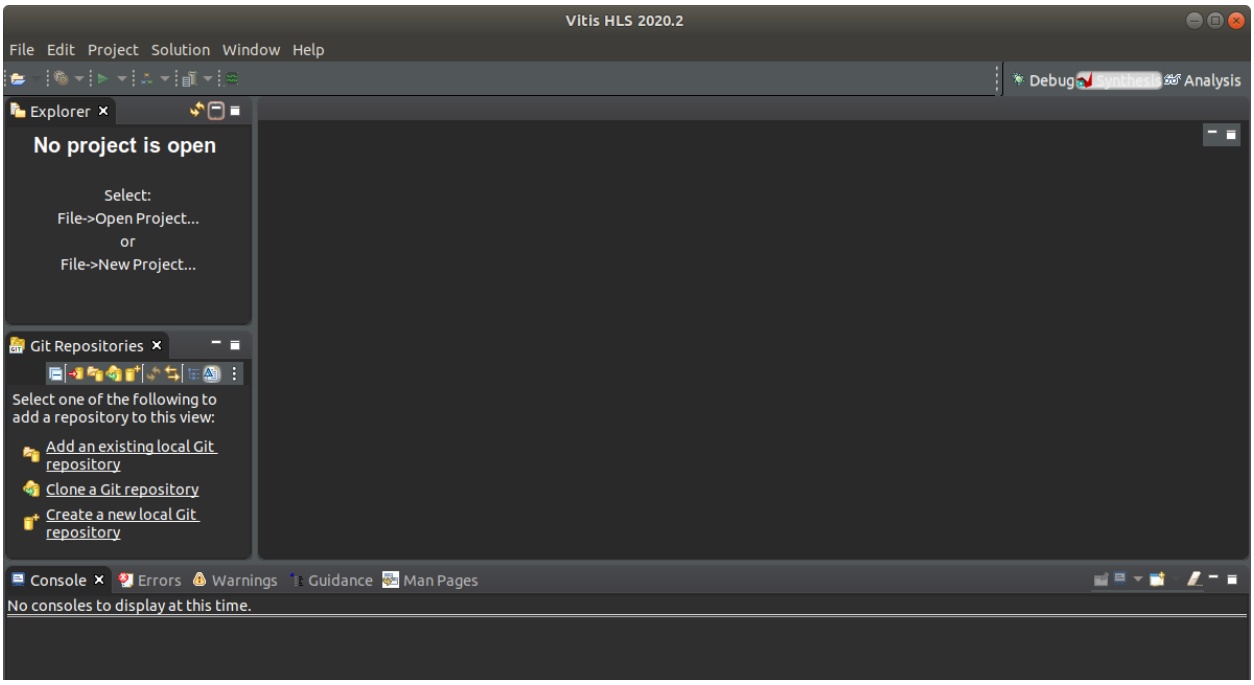


Step 2: Copy the source code file **resize_accel.cpp** to $HOME/hls_kernels/ip/resize/src.

Step 3: Continue executing these commands to open Vitis HLS.

```
cd $HOME/hls_kernels/ip/resize
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vitis_hls
```

The Vitis HLS 2020.2 should show as following.



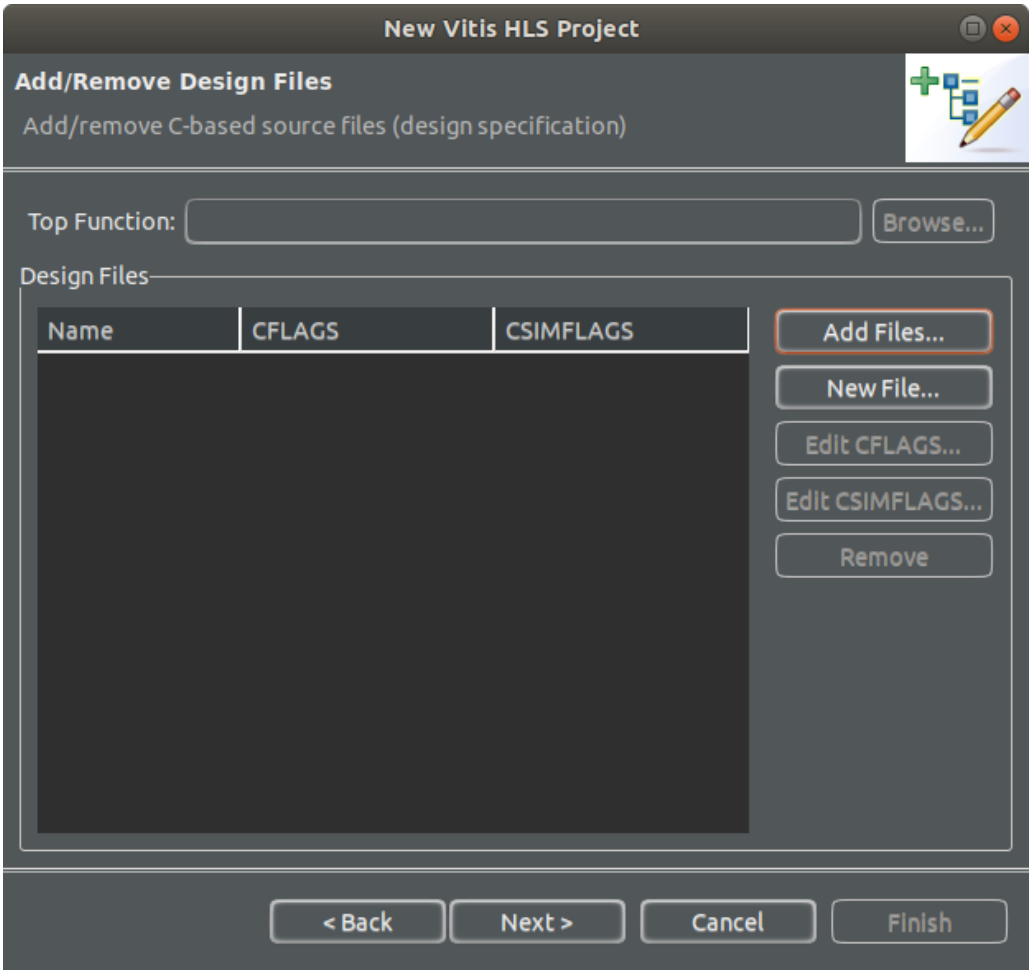
Step 4: Create new project by select **File ➔ New Project...**

Step 5: Fill the information:

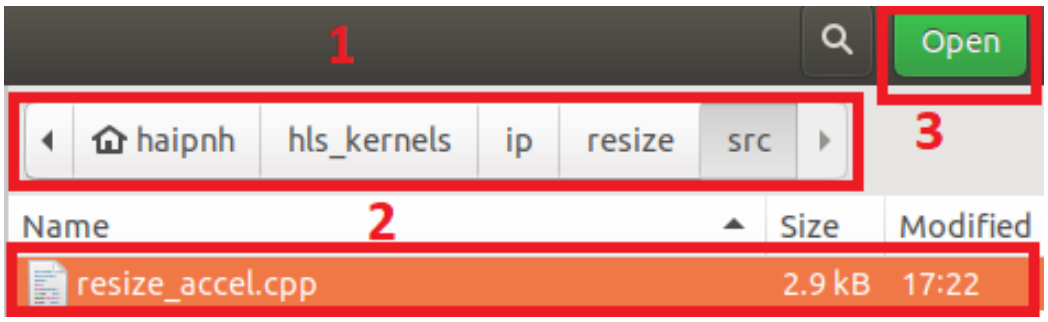
- Project name: **resize**
- Location: keep it default, check if it is $HOME/hls_kernels/ip/resize

Click **Next**.

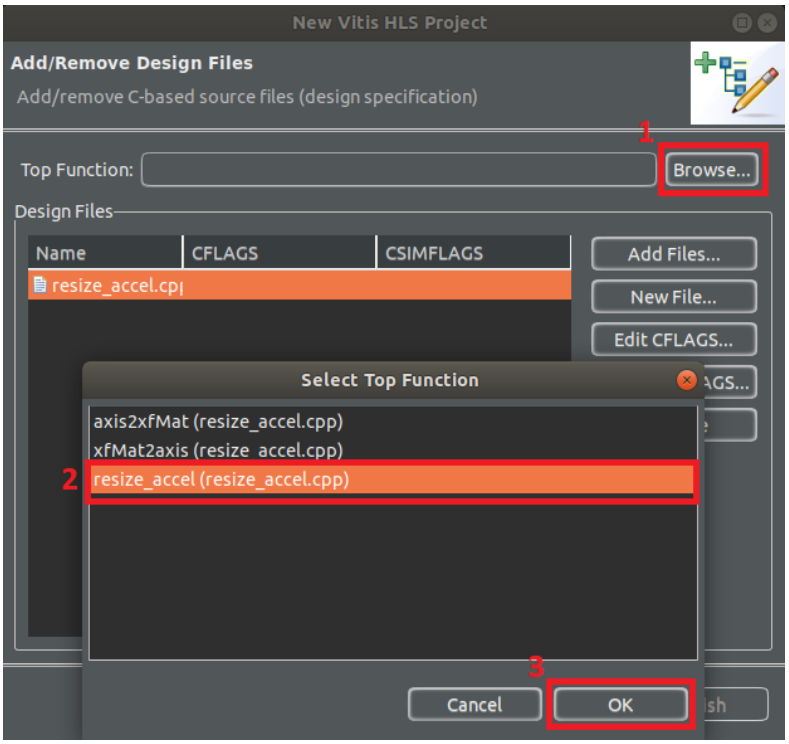
Step 6: In this step, we will add the acceleration kernel source code to the project. Click “Add Files…” in the below dialog.



Step 7: Navigate to $HOME/hls_kernels/ip/resize/src, select **resize_accel.cpp**, then click **Open**.

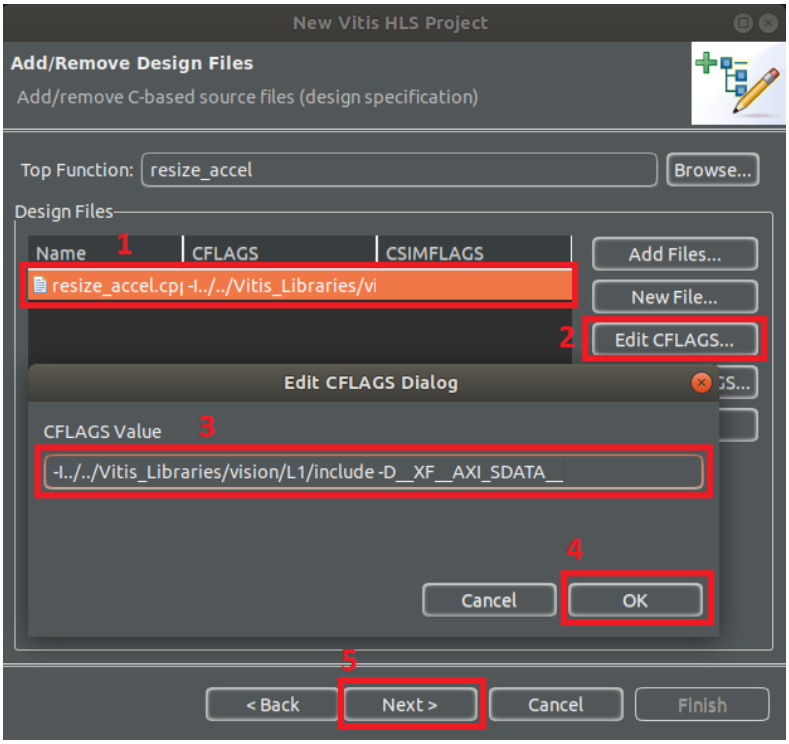


Step 8: Choose the top function in the source code file. This is an important step.

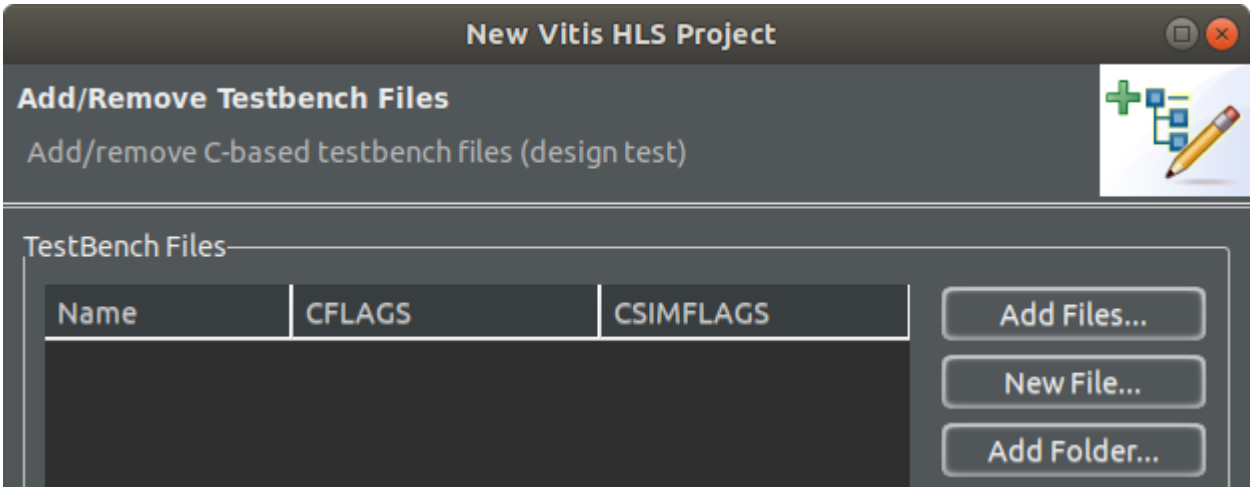


Step 9: Add following CFLAGS of the source code to include the Vitis Vision Library.

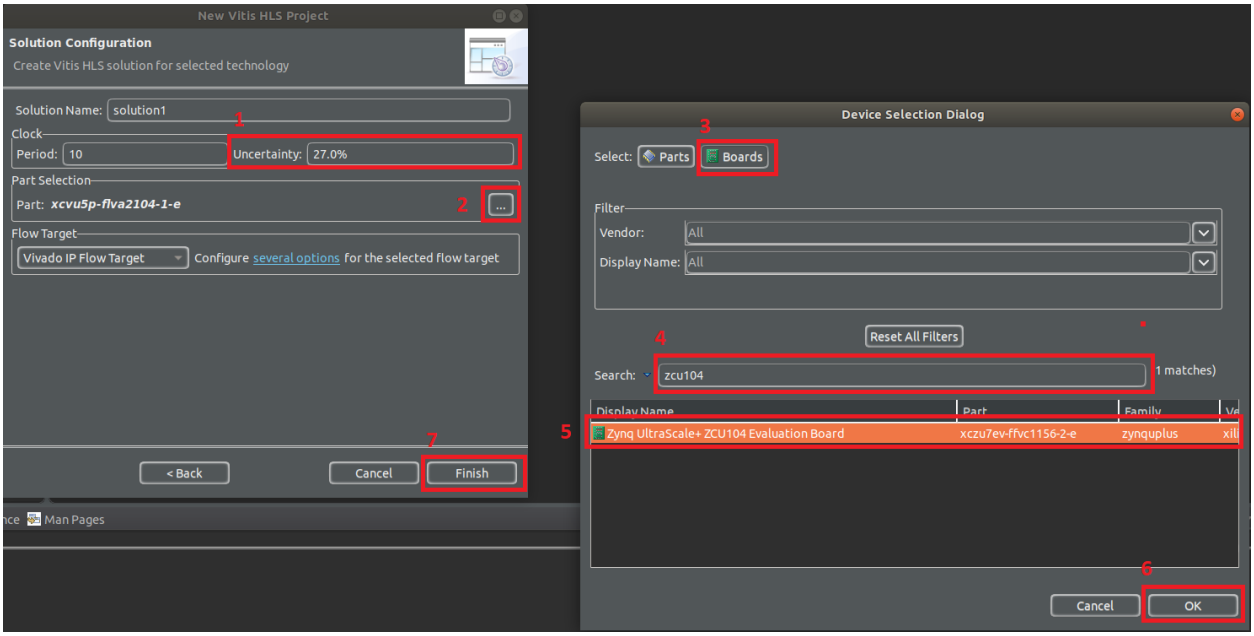
```
-I../../Vitis_Libraries/vision/L1/include -D__XF__AXI_SDATA__
```



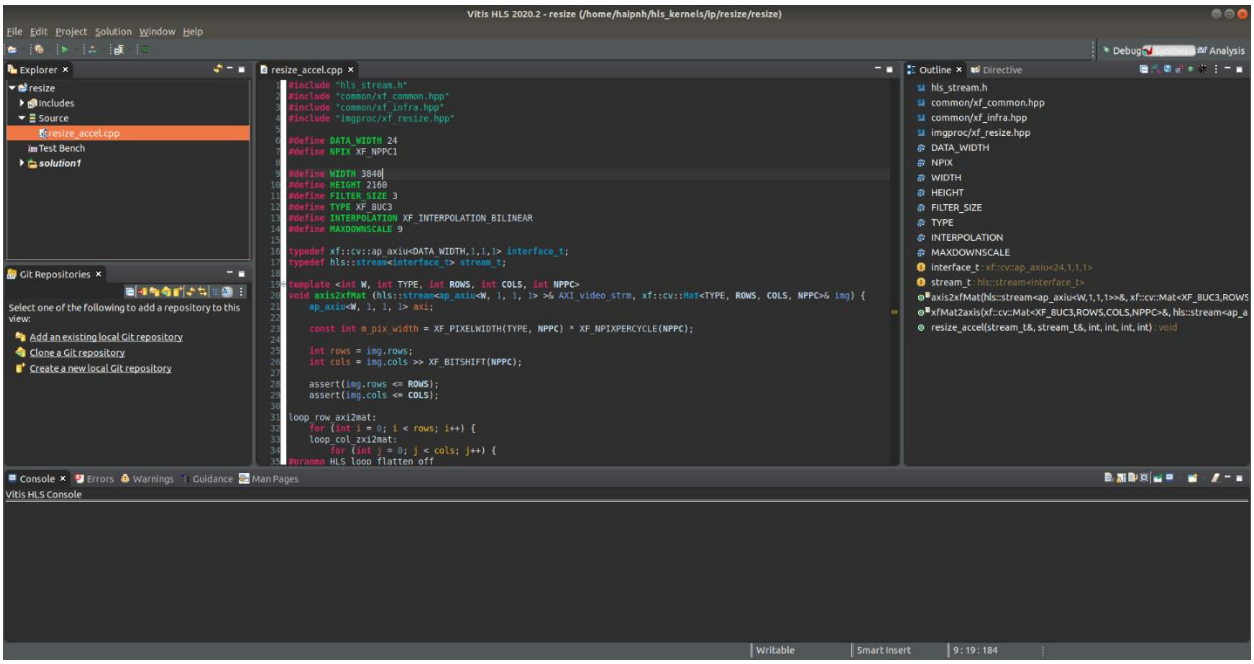
Step 10: We can skip choosing the testbench file for this project. Click Next.



Step 11: A Vitis HLS project might contain several solutions, starting with solution1. Perform following step to configure the solution1. We fill 27.0% to the Uncertainty, select ZCU104 board as target. It will result as selected **Part: xczu7ev-ffvc1156-2-e**. The configuration is finishing now.

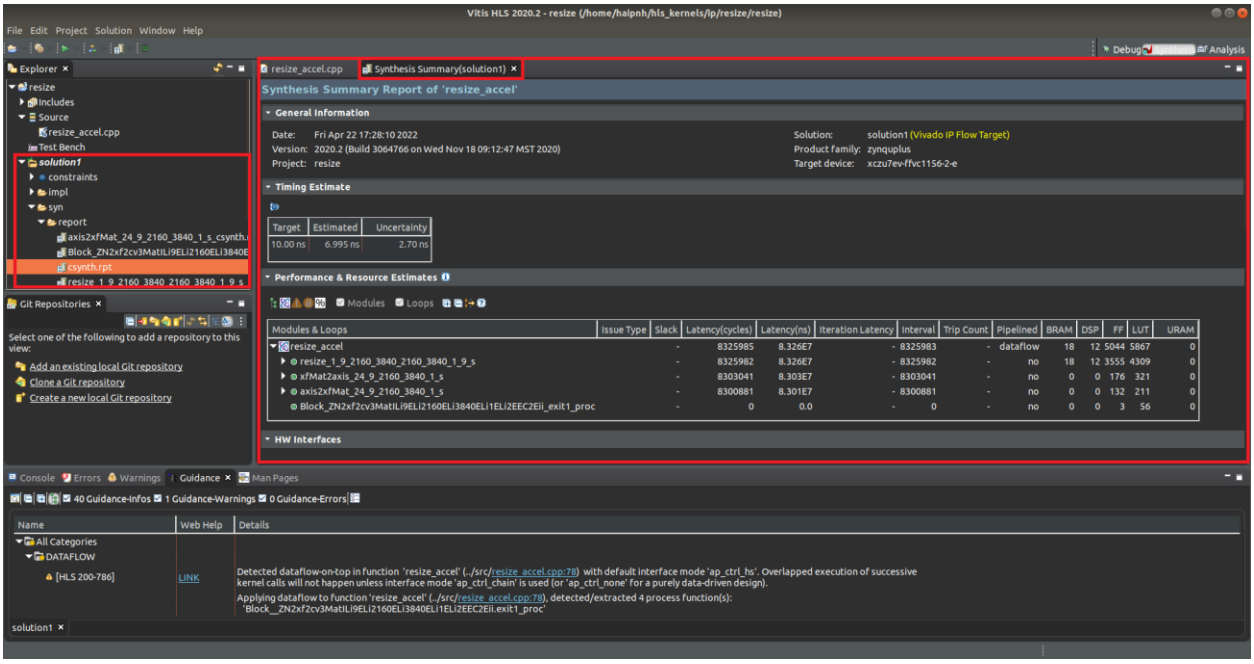


Step 12: In **Explorer**, open the **resize/Source/resize_accel.cpp** to review.

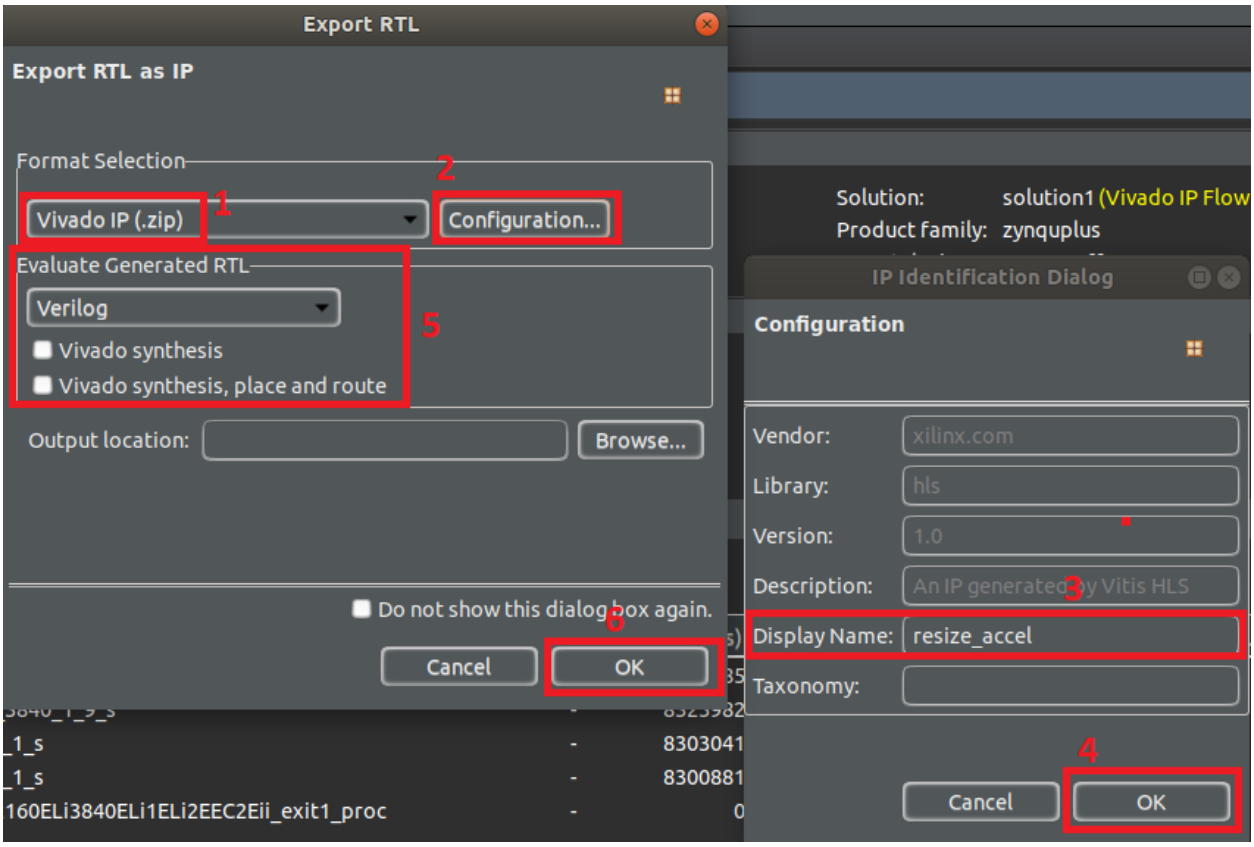


Step 13: It's ready for synthesis. To do that, open **Solution ➔ Run C Synthesis ➔ Active Solution.**

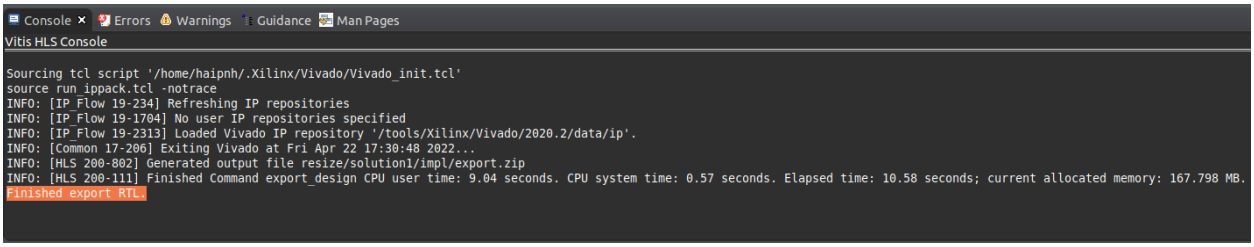
Step 14: Synthesis report can be found in **resize/solution1/syn/report/csynth.rpt**.



Step 15: Open **Solution ➔ Export RTL** to export the kernel. Configure as following.



Step 16: Wait for the exporting and the console should show the following message. Now we can close Vitis HLS.

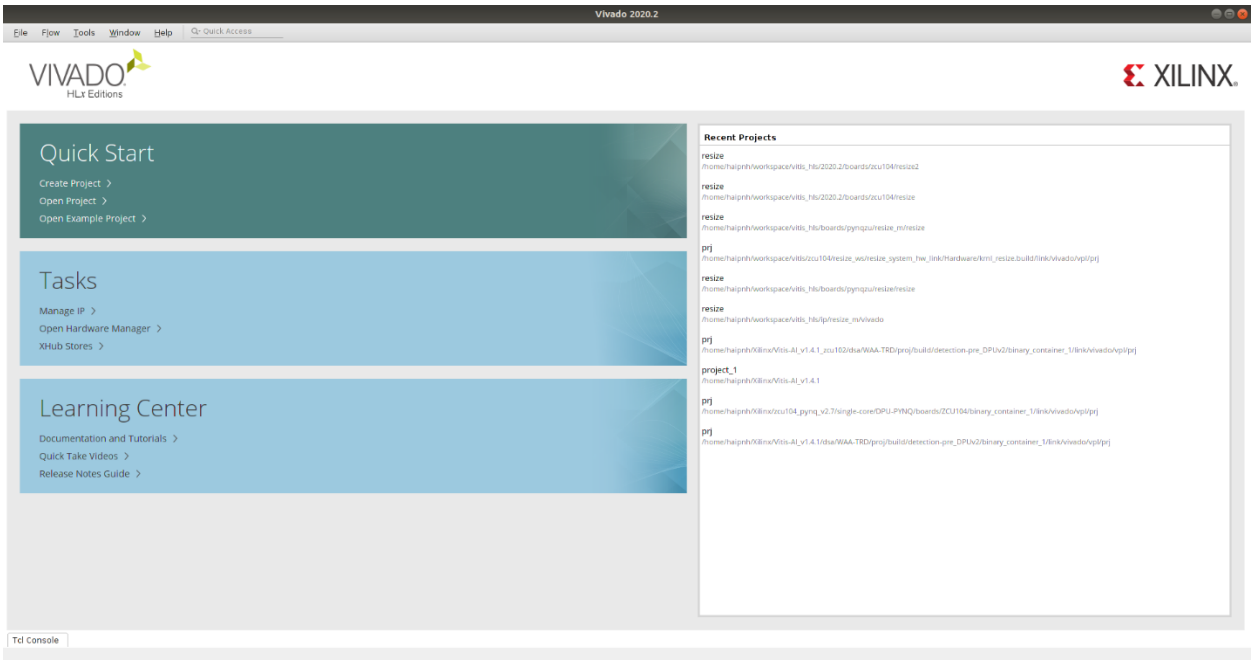


4.3. System Design and Generate Bitstream using Vivado

Step 1: Open terminal, run these commands to open Vivado.

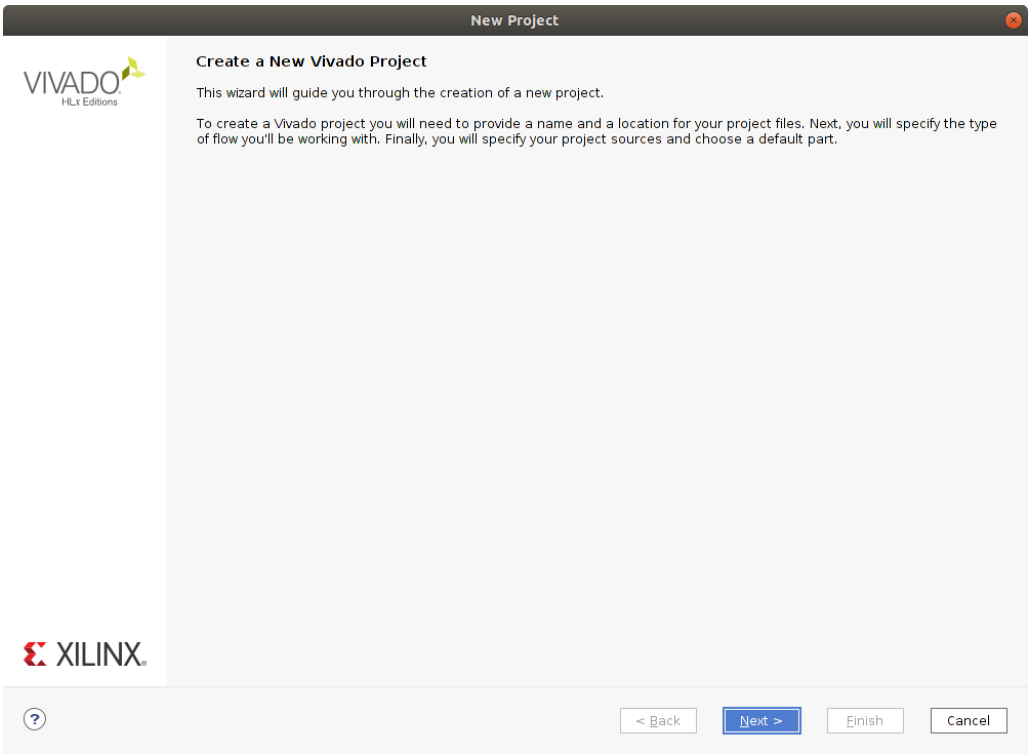
```
cd $HOME/hls_kernels/boards/zcu104/resize
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vivado
```

The Vivado window should be shown as following.



Step 2: Create new Vivado project, open **File ➔ New...**

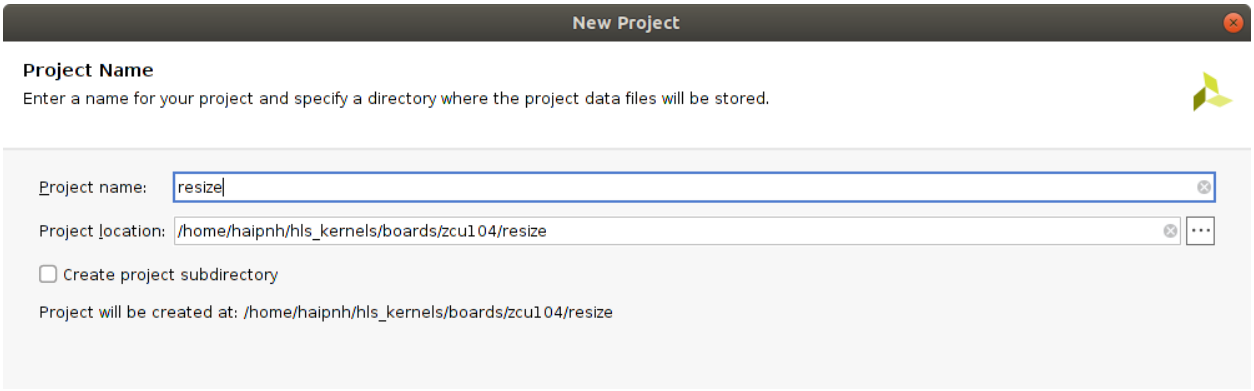
Step 3: Click **Next** on New Project dialog.



Step 4: Fill the information as following.

- Project name: **resize**
- Project location: in $HOME/hls_kernels/boards/zcu104/resize
- Uncheck Create project subdirectory

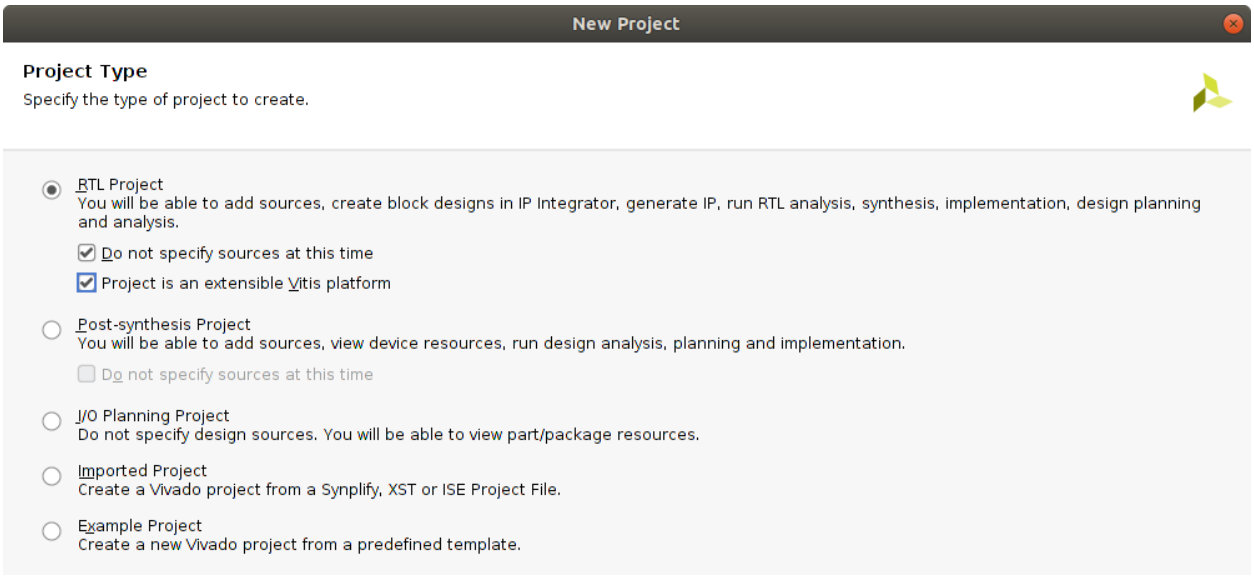
Click **Next**.



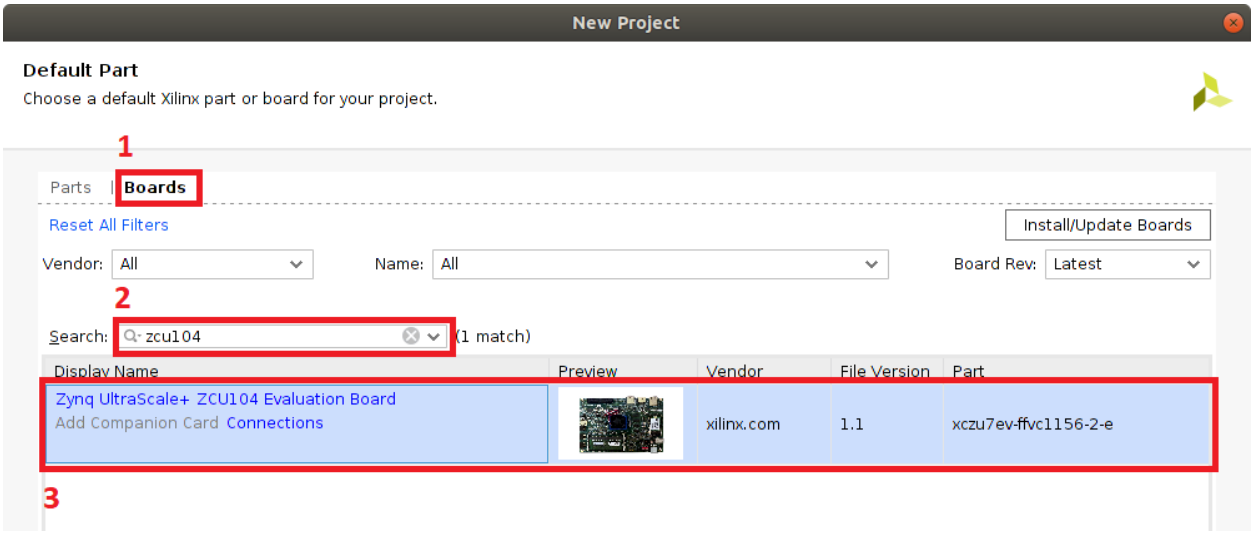
Step 5: Select **RTL Project** type

- Check **Do not specify sources at this time**
- Check **Project is an extensible Vitis platform**

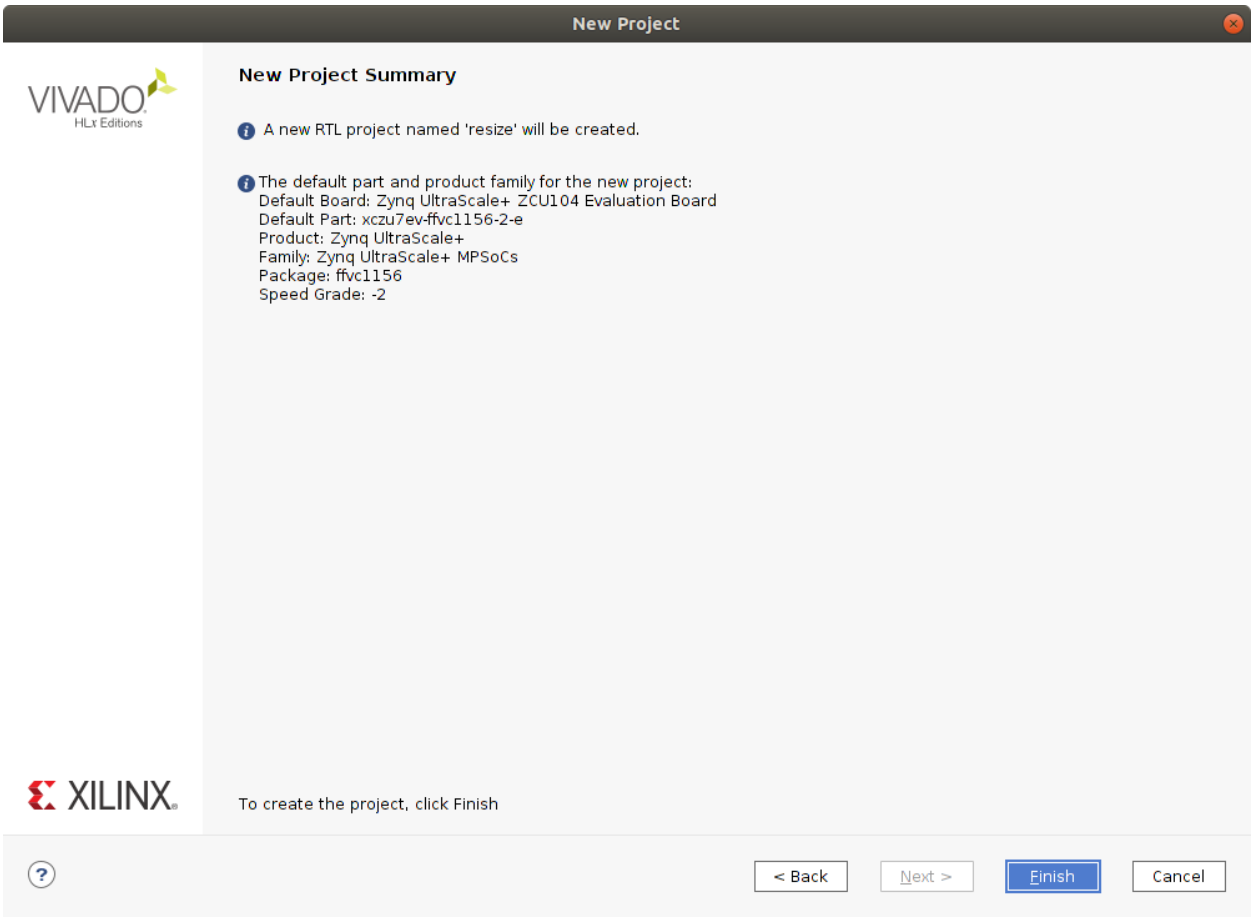
Click **Next**.



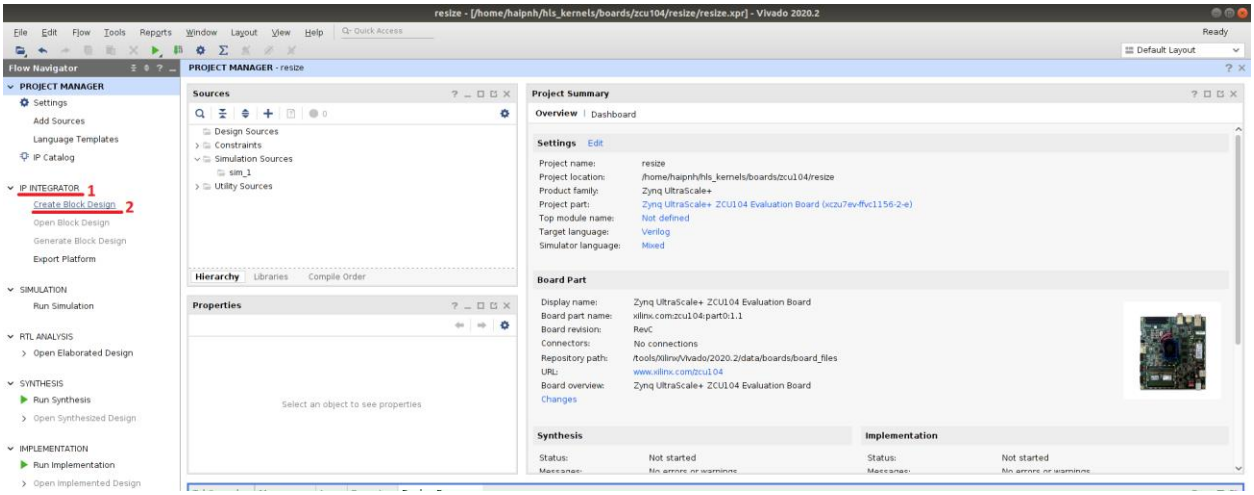
Step 6: Select **Boards** ➔ Search for **zcu104** ➔ Then select the **Zynq UltraScale+ ZCU104 Evaluation Board** ➔ Click **Next**.



Step 7: Review the **New Project Summary**. It should be as following. Then click **Finish**.

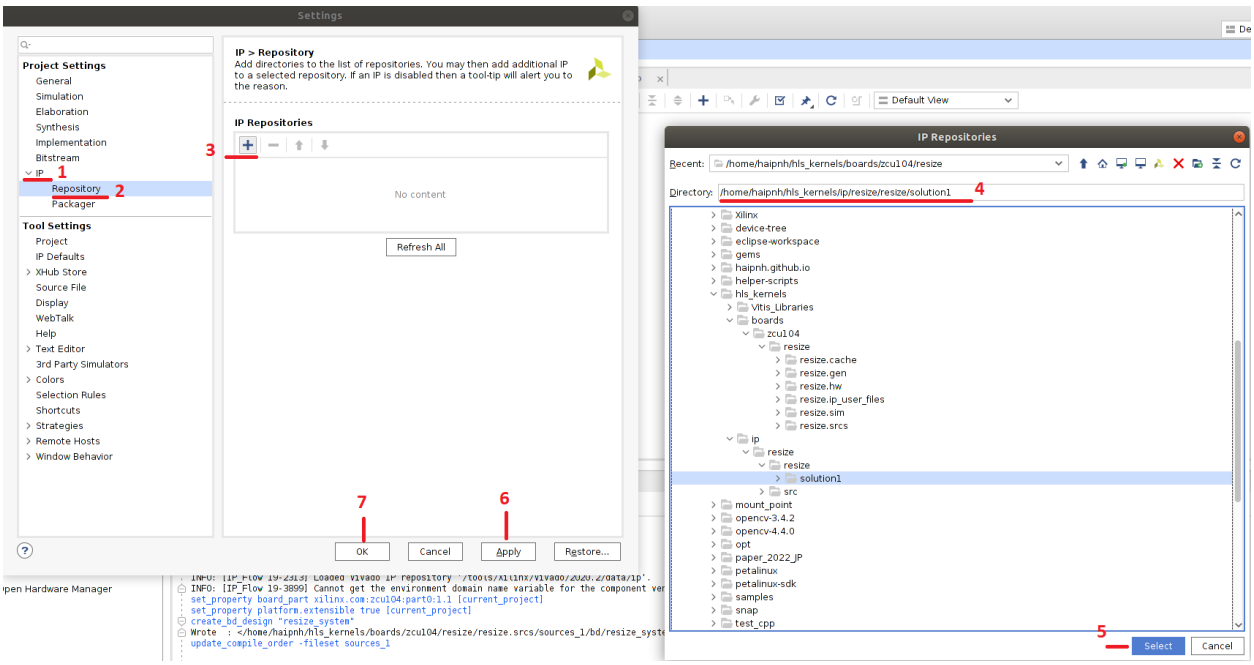


This is how a new project looks like.

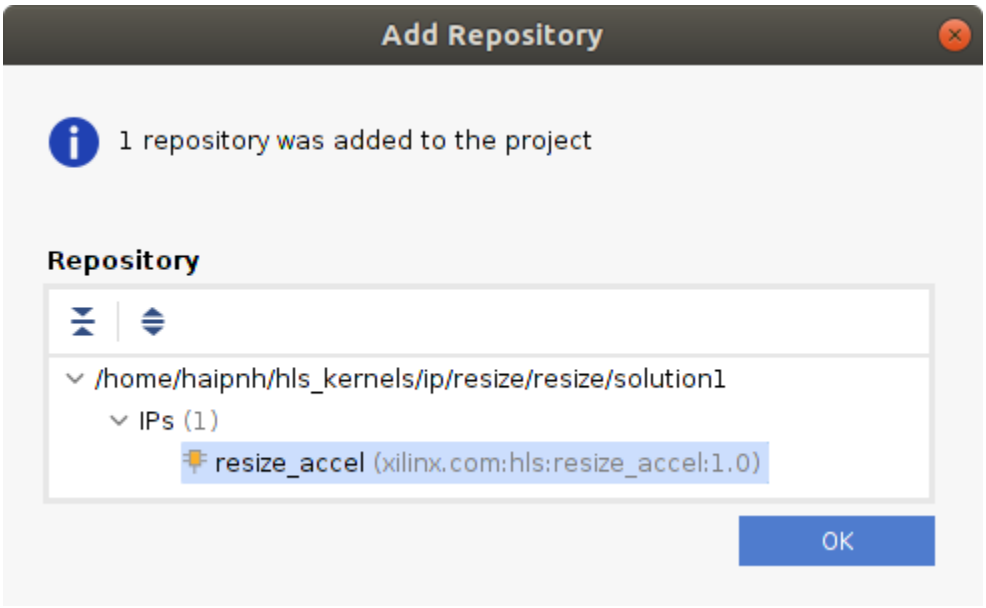


Step 8: Click **Tools ➔ Settings…** to open **Project Settings**.

Step 9: Follow these steps to add **User IP Repositories**.
The directory is **$HOME/hls_kernels/ip/resize/resize/solution1** (see below screenshot).

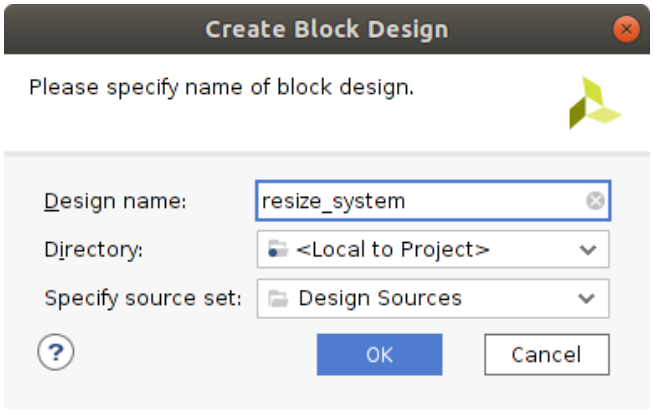


Step 10: The exported IP should be found: **resize_accel**.

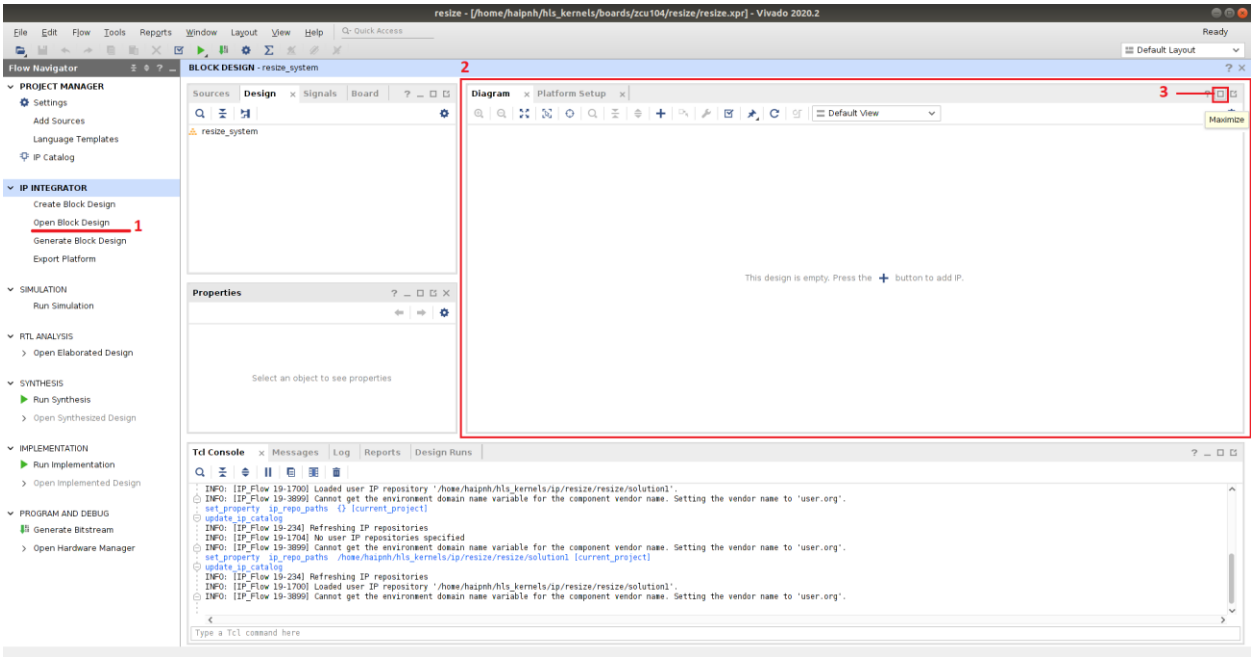


Step 11: Click **IP INTEGRATOR ➔ Create Block Design**

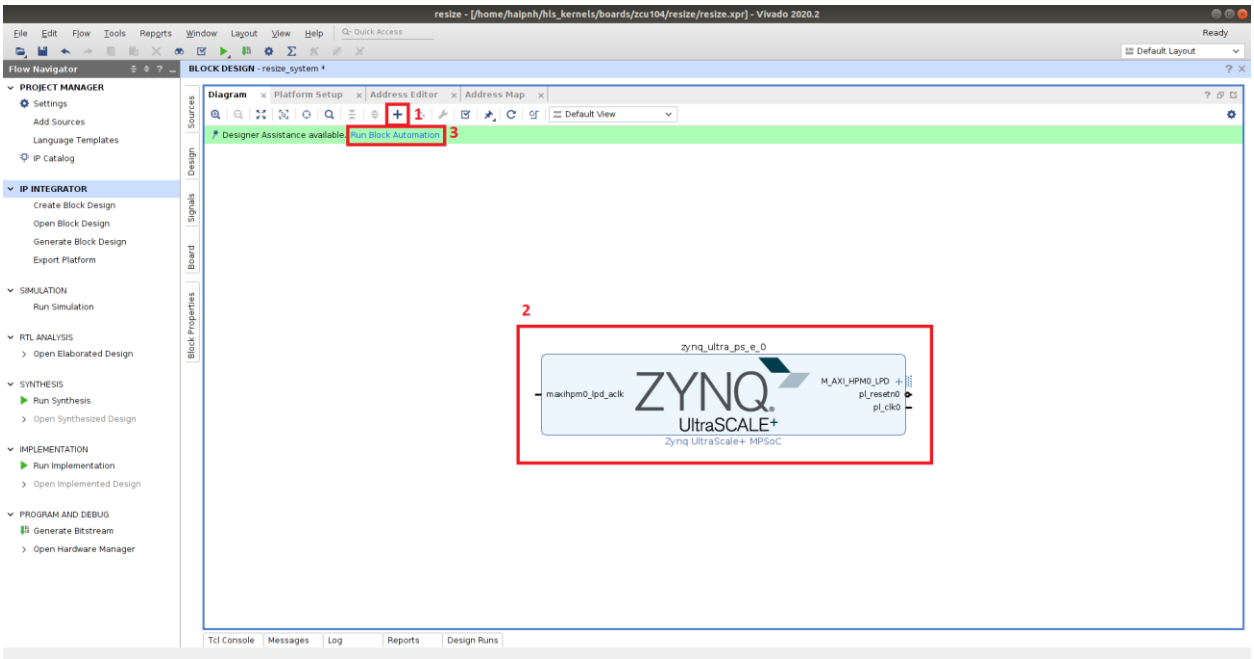
Step 12: Set the **Design name** to **resize_system** ➔ Click OK



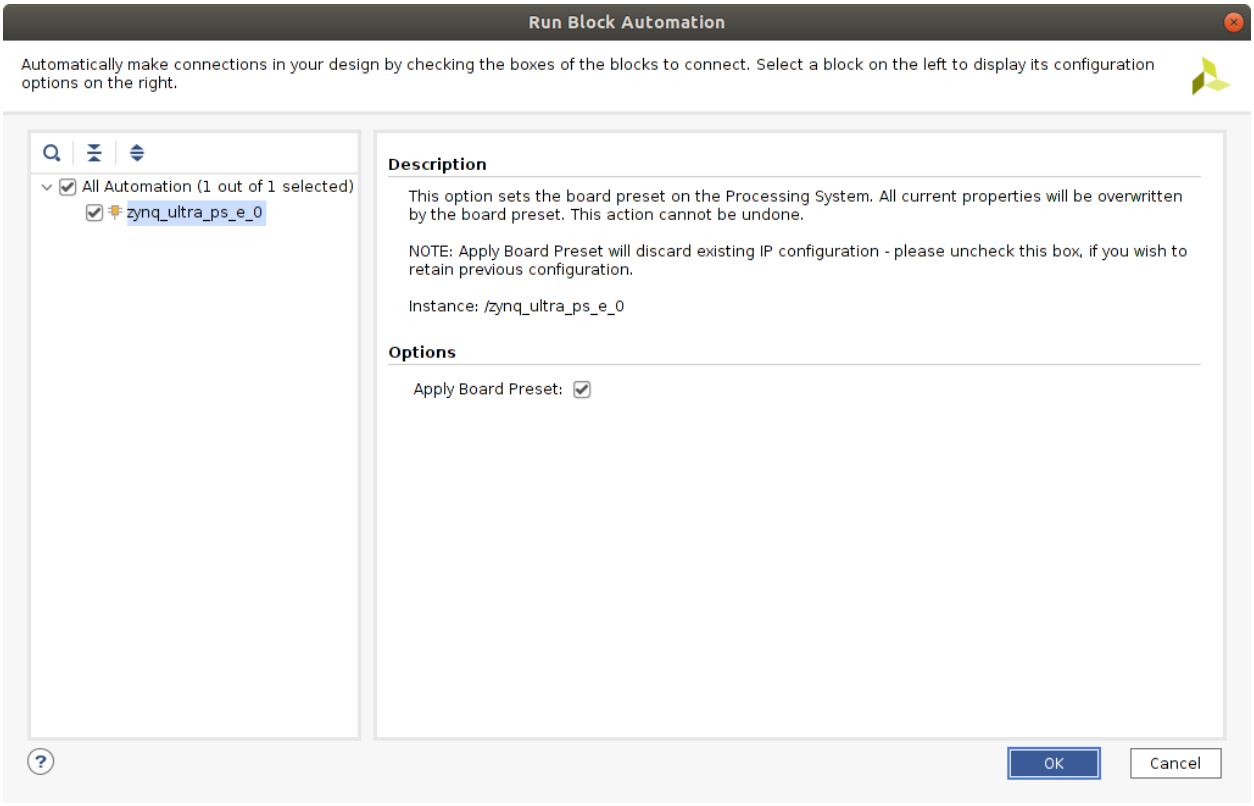
Step 13: Click **IP INTEGRATOR ➔ Open Block Design**. An empty diagram should be shown. You might want to maximize the **Diagram** tab for more convenient.



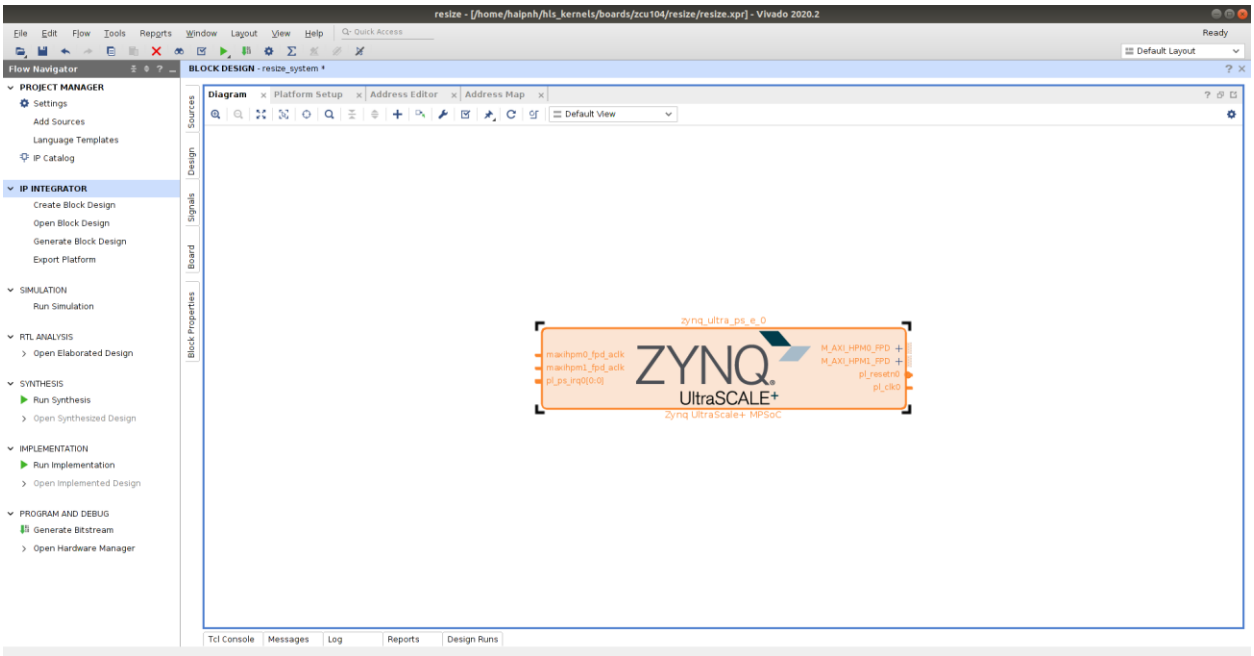
Step 14: Follow below steps to add the first block: a **Zynq UltraScale+ MPSoC**. It will have the instance name **zynq_ultra_ps_e_0**. Then click **Run Block Automation**.



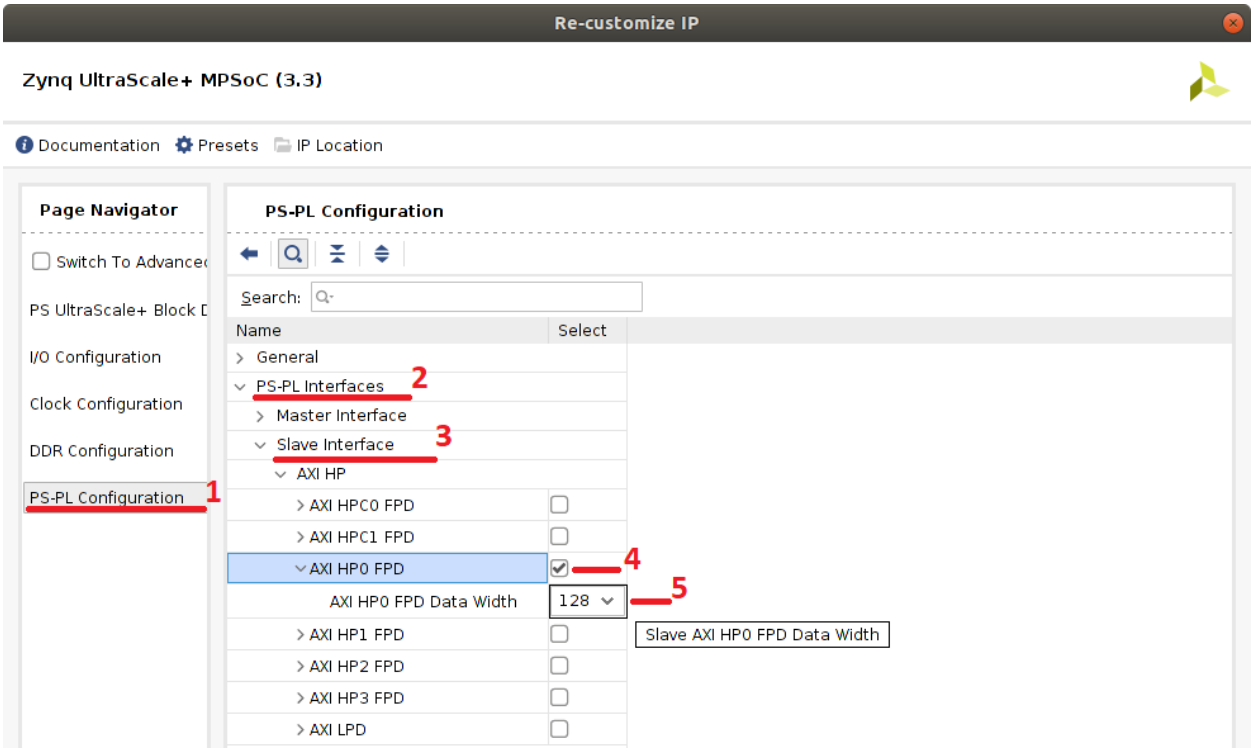
Step 15: Check **All Automation**, confirm **Apply Board Preset** is checked ➔ Click OK.



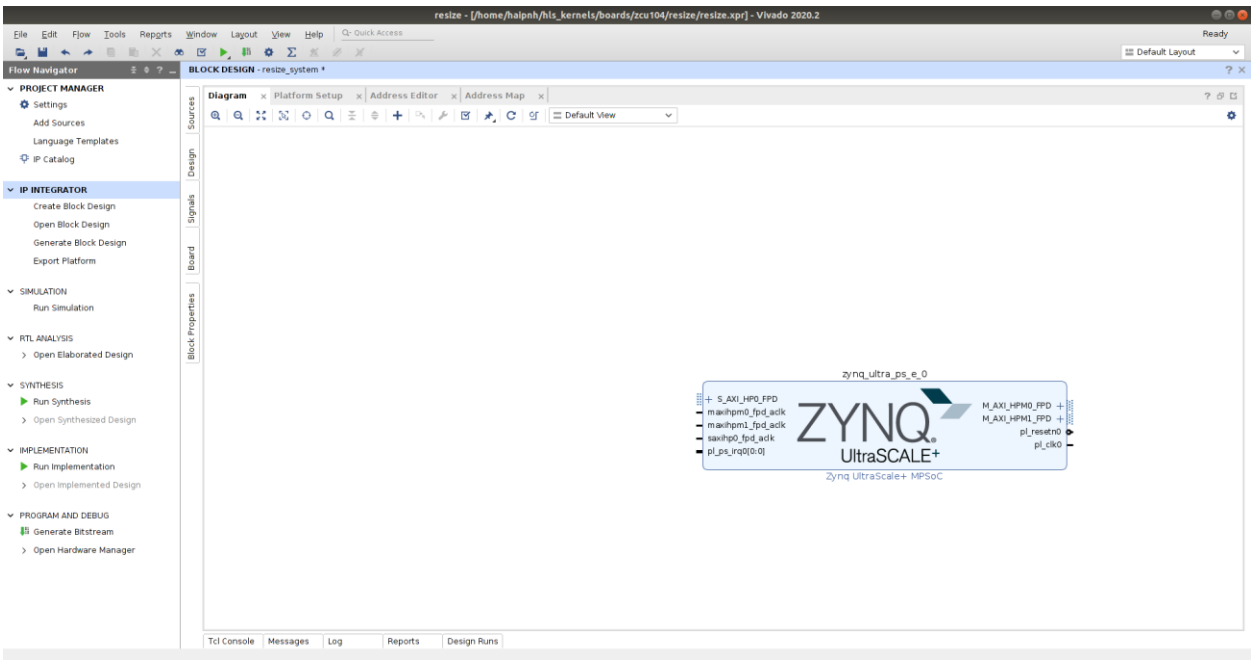
Step 16: The result is as following. Right click on **zynq_ultra_ps_e_0 ➔ Customize Block**



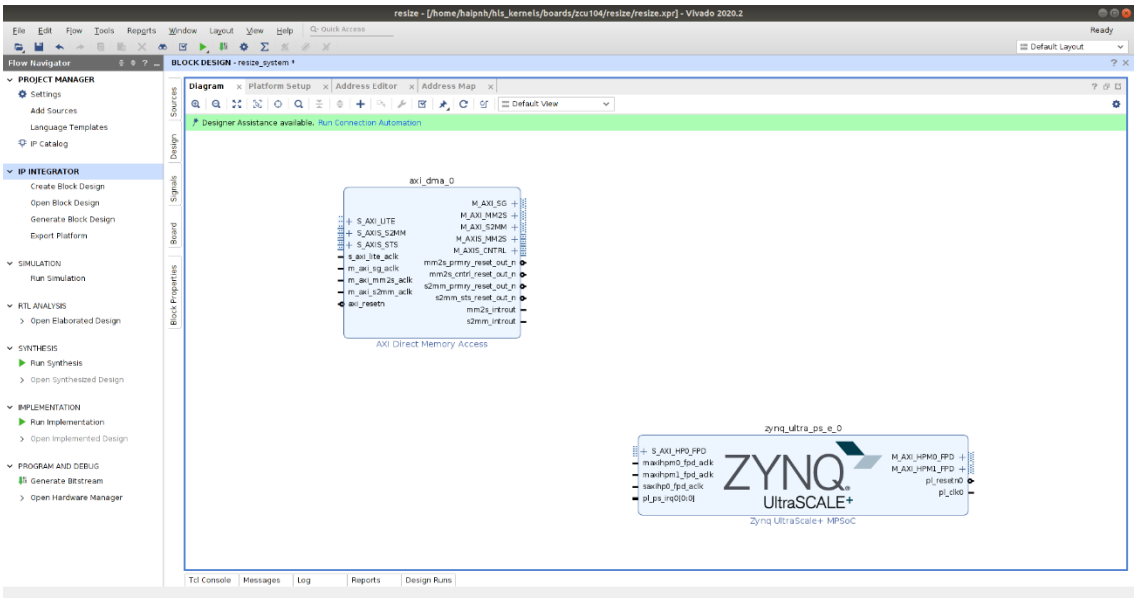
Step 17: Go to **PS-PL Configuration ➔ PS-PL Interfaces ➔ Slave Interface ➔ AXI HP.** Select the **AXI HP0 FPD**, set its **Data Width** to 128. Then click **OK**.



As a result, **S_AXI_HP0_FPD** interface is available now.

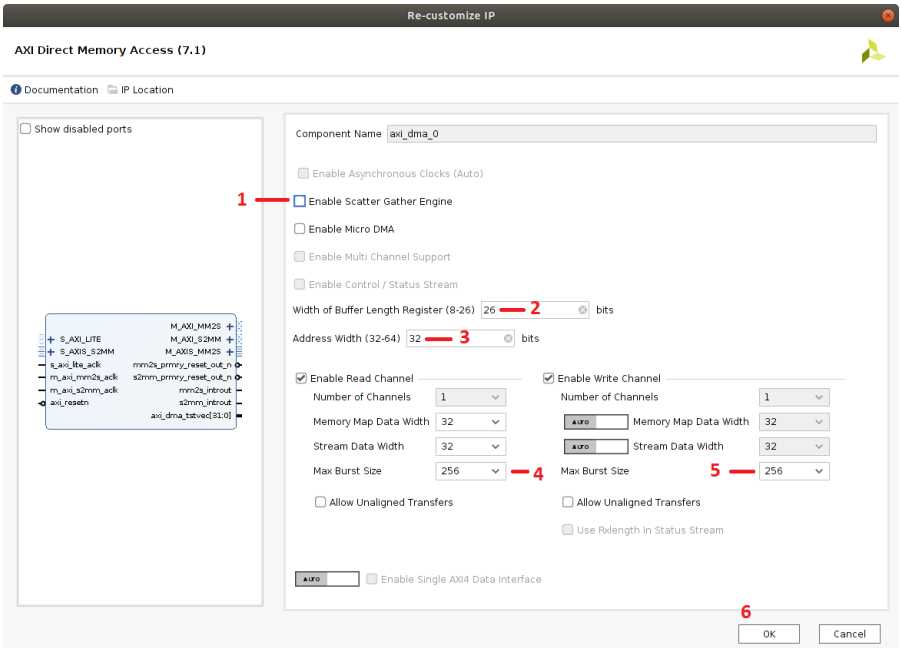


Step 18: Add a **AXI Direct Memory Access** block.

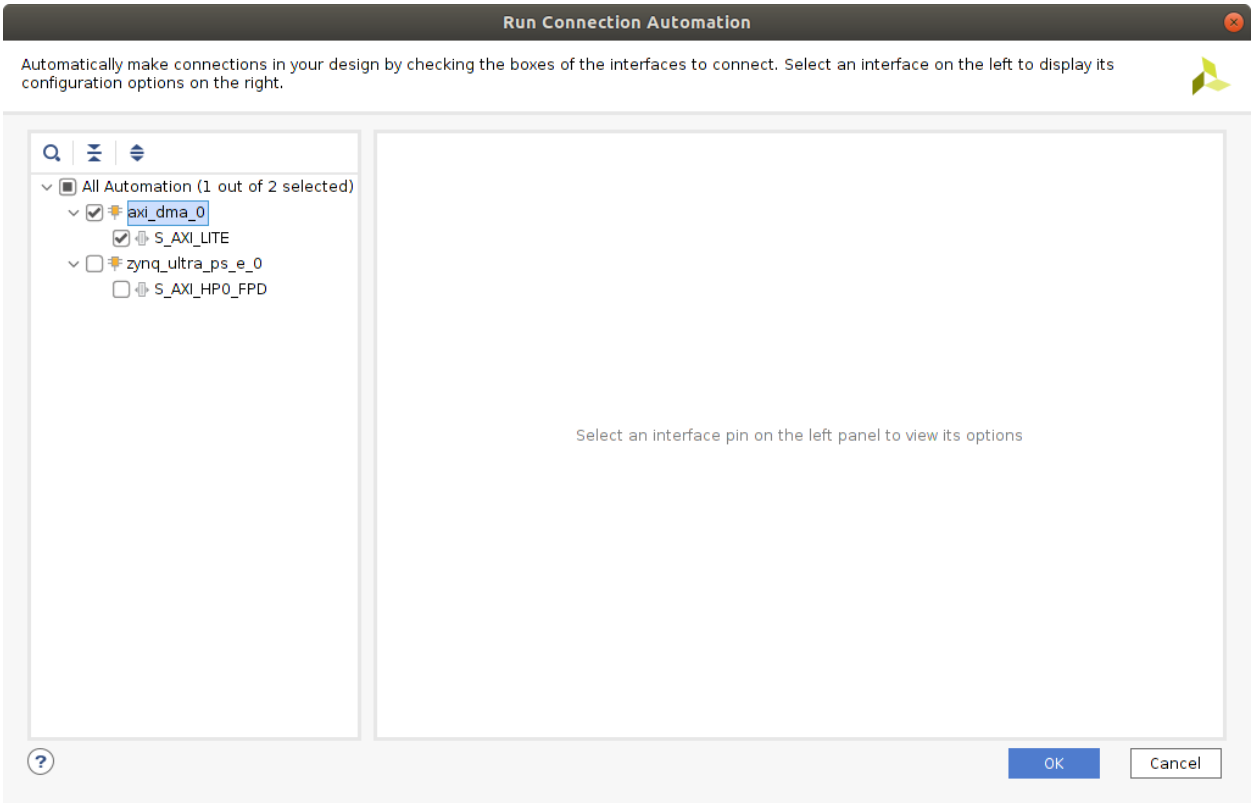


Step 19: Right click on **axi_dma_0** ➔ **Customize Block**. Configure as following:

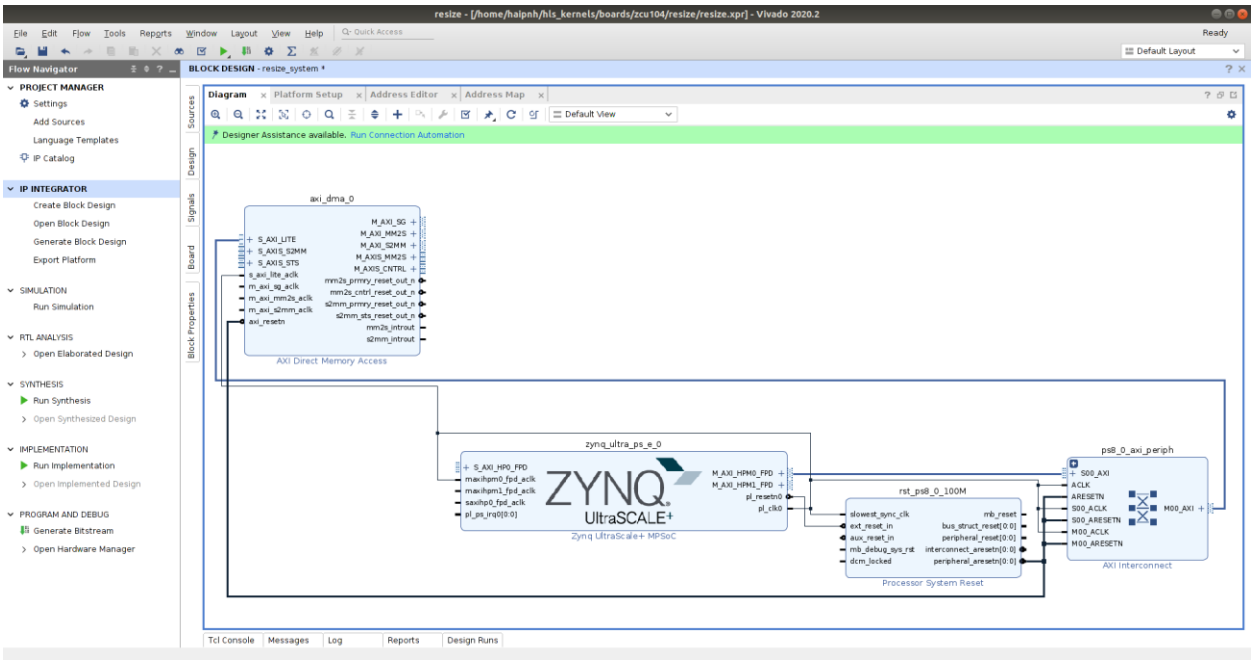
- Disable **Enable Scatter Gather Engine**
- Width of Buffer Length Register: 26 bits
    - Address Width: 32 bits
- Enable Read Channel
    - Max Burst Size: 256
- Enable Write Channel
    - Max Burst Size: 256



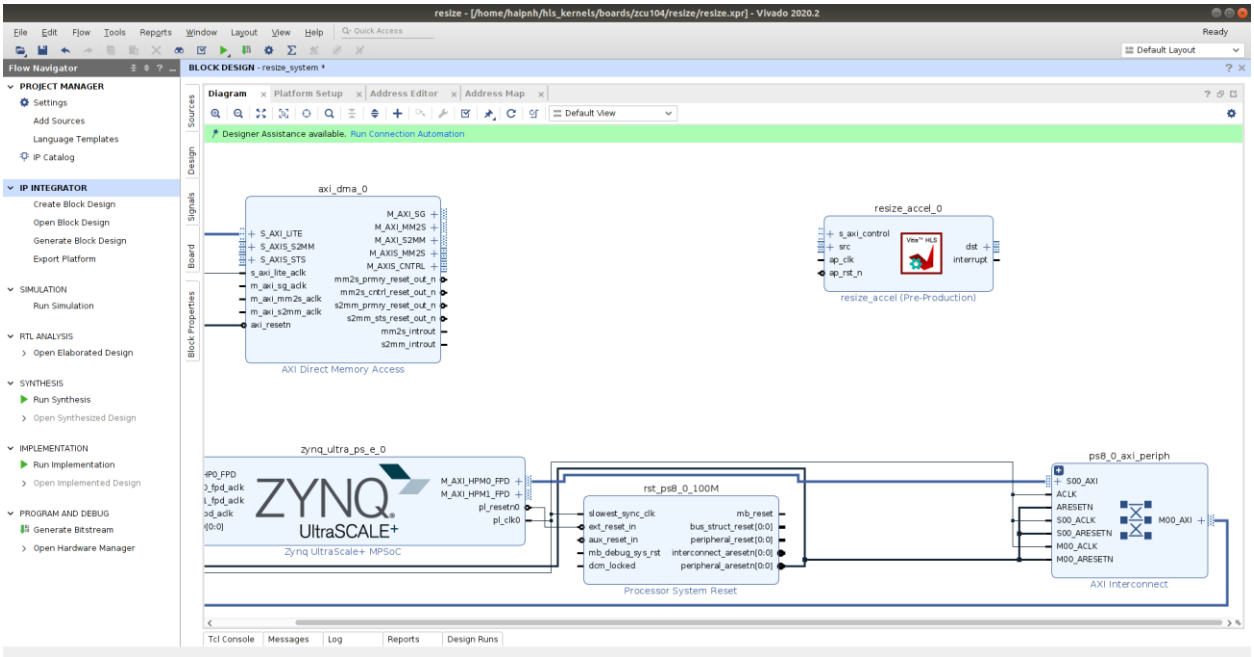
Step 20: Click **Run Connection Automation** with the following configuration.



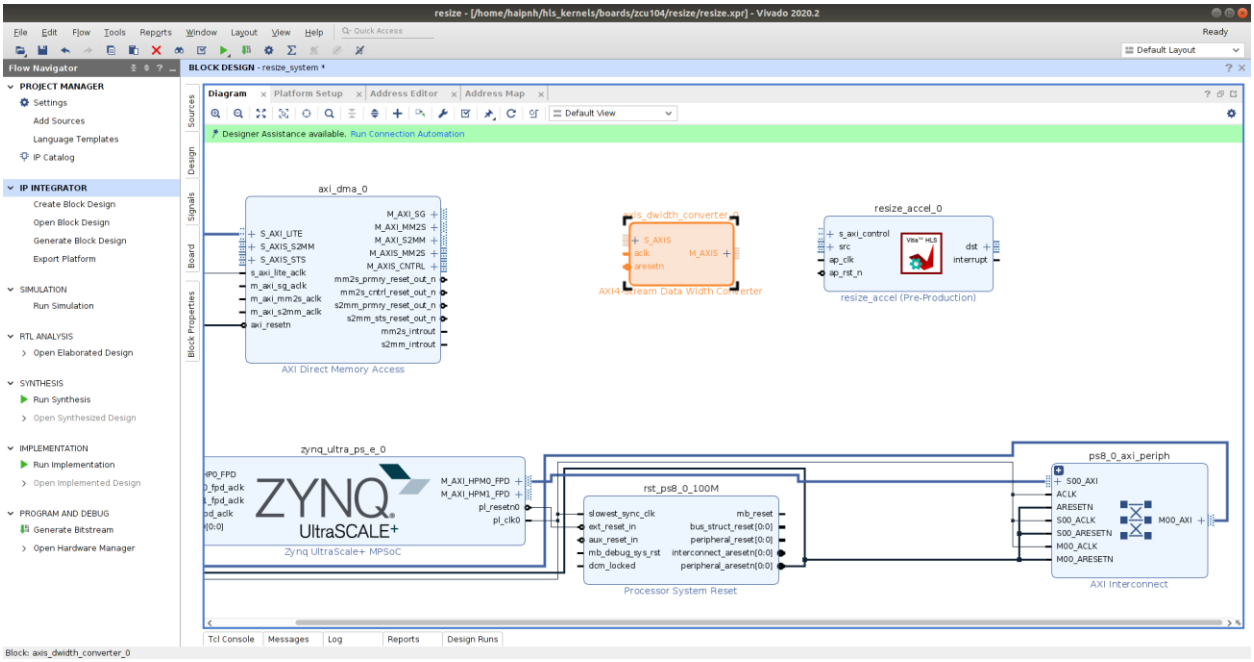
The following 4 blocks should be shown. Ignore the below **Run Connection Automation** suggestion.



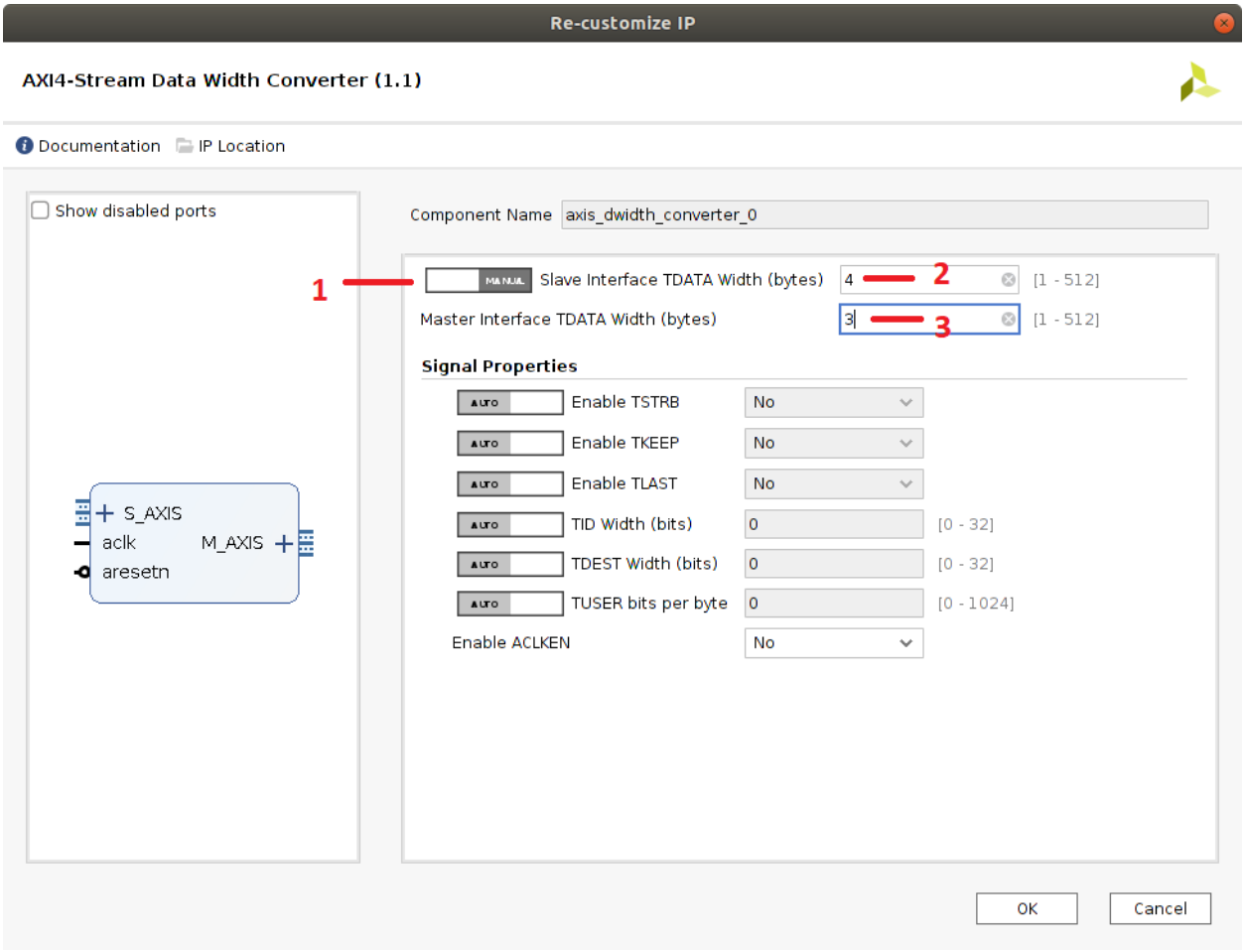
Step 21: Add **resize_accel** block.



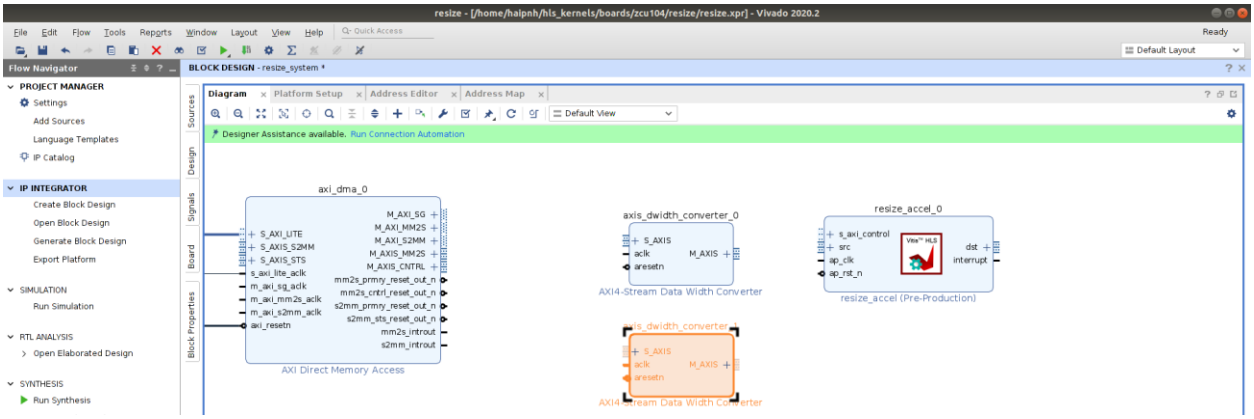
Step 22: Add a **AXI4-Stream Data Width Converter**, instanced as **axis_dwidth_converter_0**.



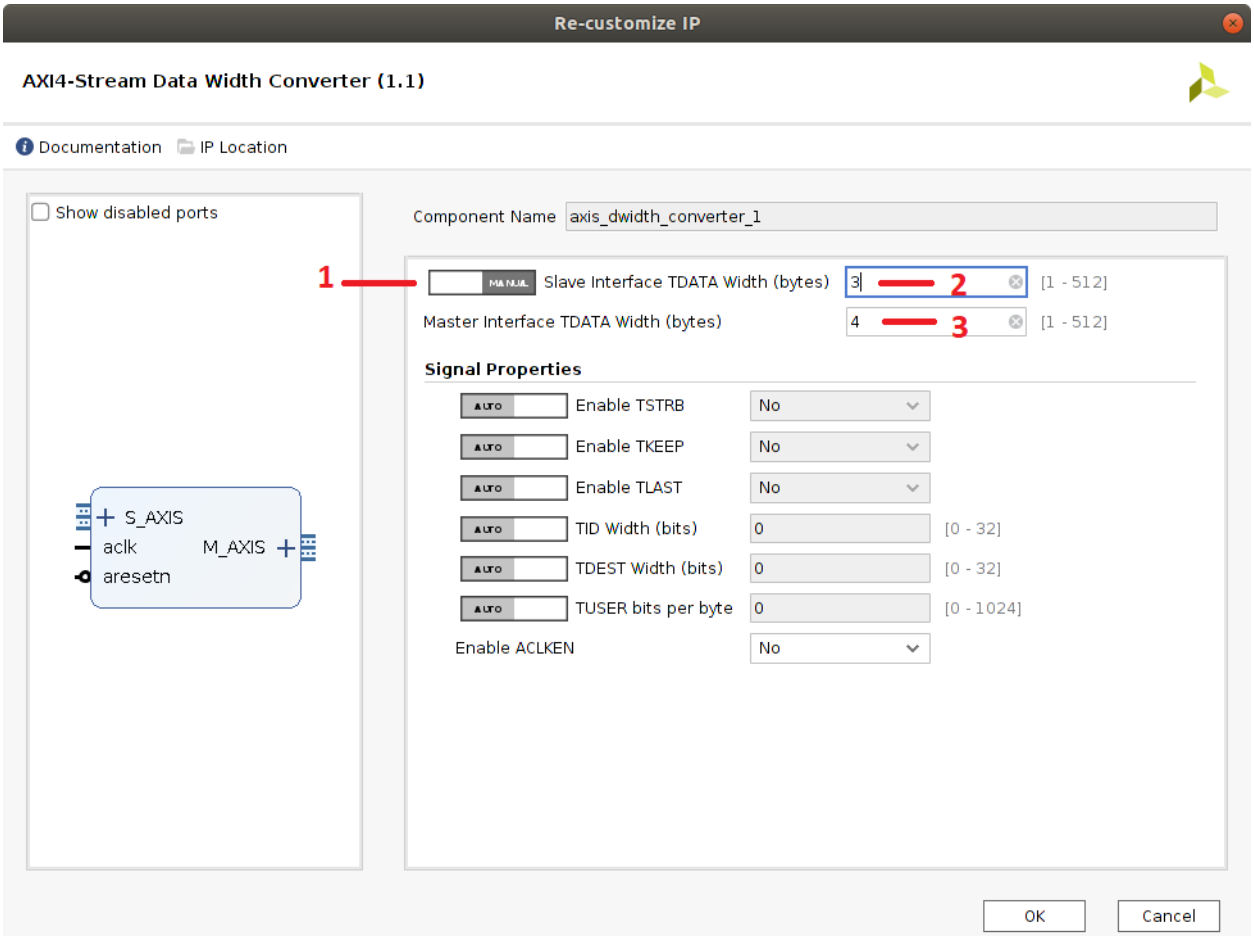
Step 23: Right click **axis_dwidth_converter_0 ➔ Customize Block…., ** configure as following, then click **OK**.



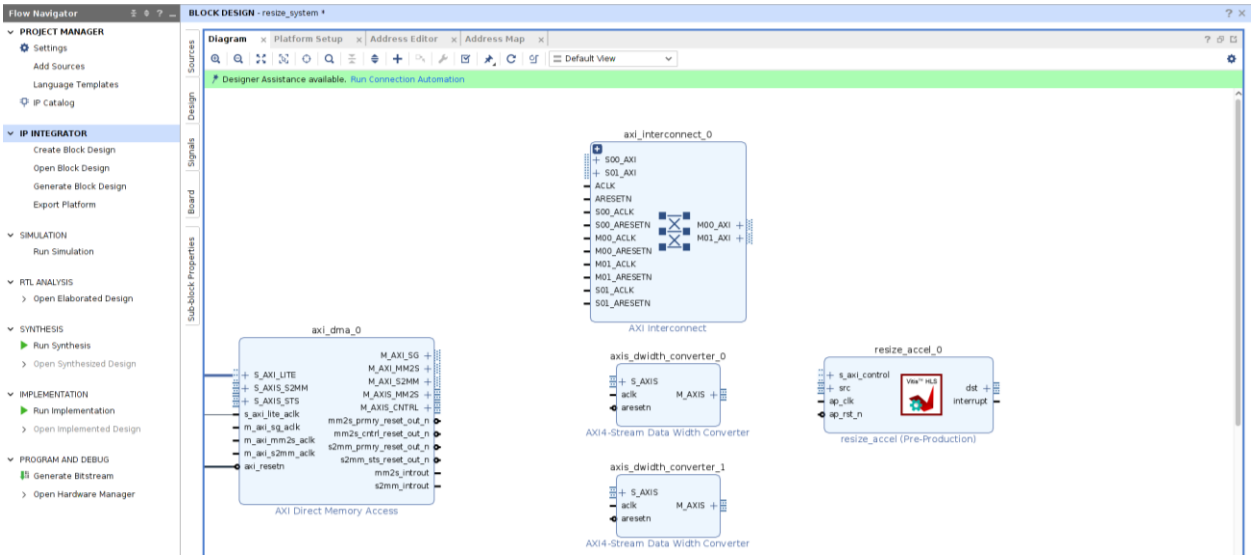
Step 24: Click **axis_dwidth_converter_0**, press Ctrl+C then press Ctrl+V to create **axis_dwidth_converter_1.**



Step 25: Right click **axis_dwidth_converter_1 ➔ Customize Block…..,** configure as following, then click **OK**.

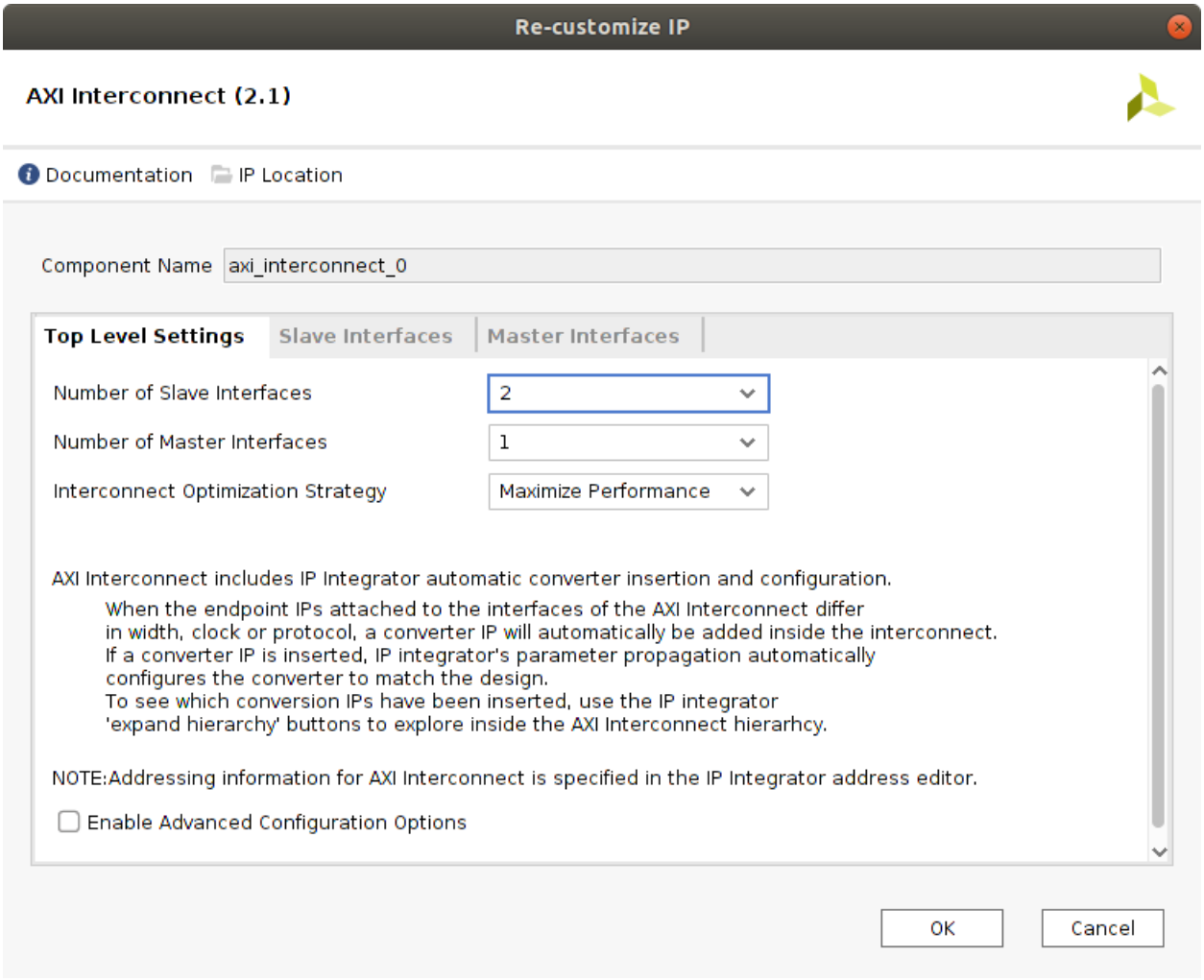


Step 26: Add a **AXI Interconnect**, instanced as **axi_interconnect_0**.



Step 25: Right click **axi_interconnect_0 ➔ Customize Block…., ** configure as following, then click **OK**.

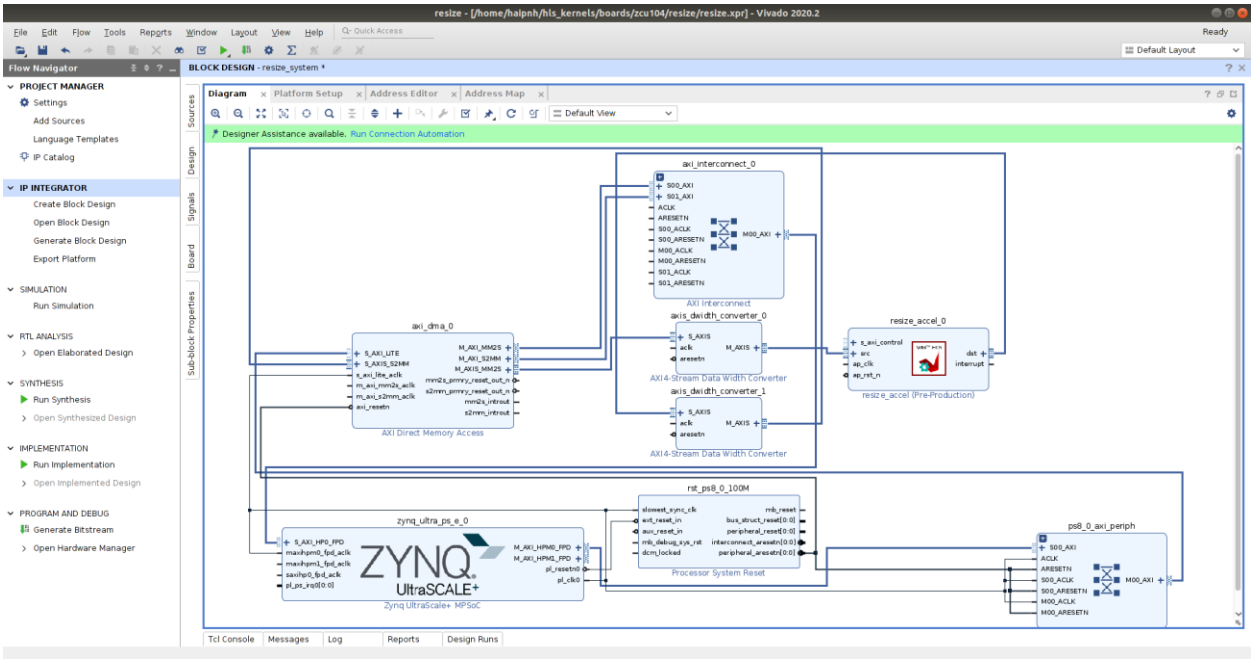
- Number of Slave Interfaces: 2
- Number of Master Interfaces: 1
- Interconnect Optimization Strategy: Maximize Performance



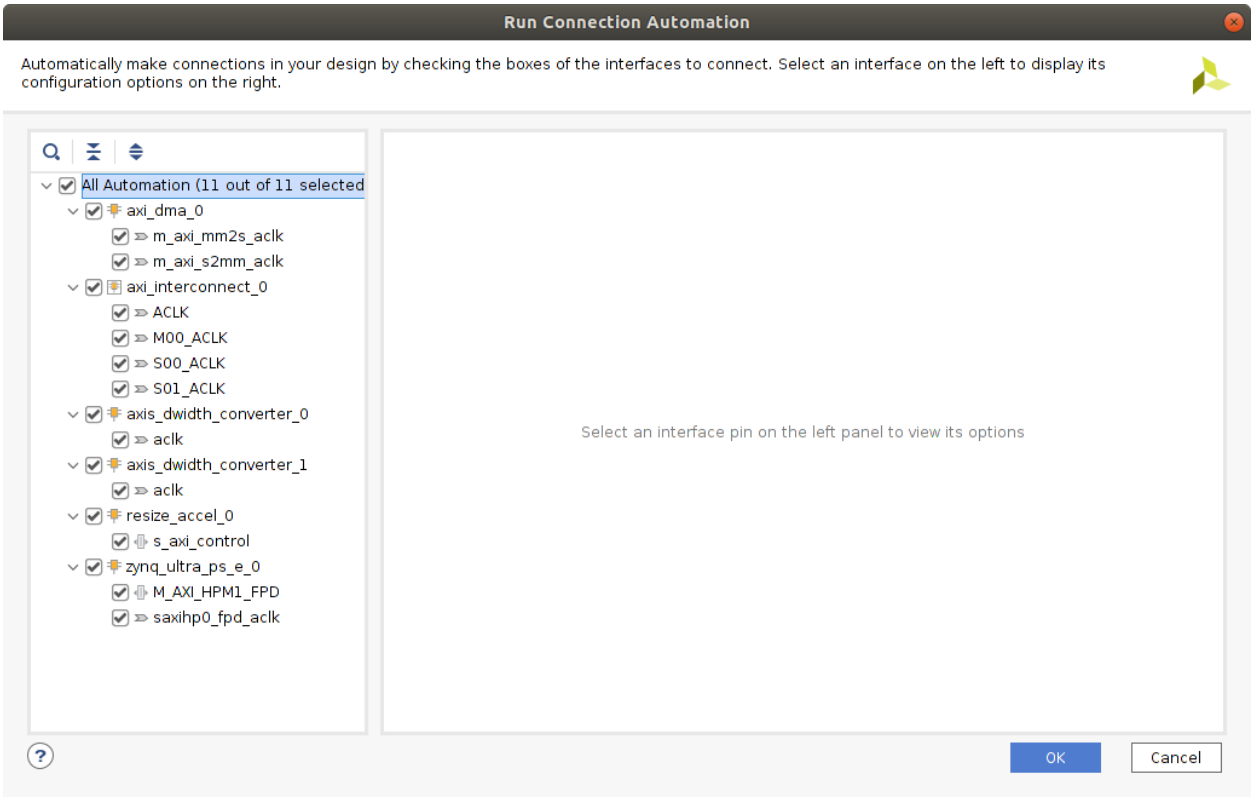
Step 26: Wire the interfaces of these blocks as following:

- zynq_ultra_ps_e_0 / S_AXI_HP0_FPD <-> axi_interconnect_0 / M00_AXI
- axi_interconnect_0 / S00_AXI <-> axi_dma_0 / M_AXI_MM2S
- axi_interconnect_0 / S01_AXI <-> axi_dma_0 / M_AXI_S2MM
- axi_dma_0 / M_AXIS_MM2S <-> axis_dwidth_converter_0 / S_AXIS
- axi_dma_0 / S_AXIS_S2MM <-> axis_dwidth_converter_1 / M_AXIS
- axis_dwidth_converter_0 / M_AXIS <-> resize_accel_0 / src
- axis_dwidth_converter_1 / S_AXIS <-> resize_accel_0 / dst

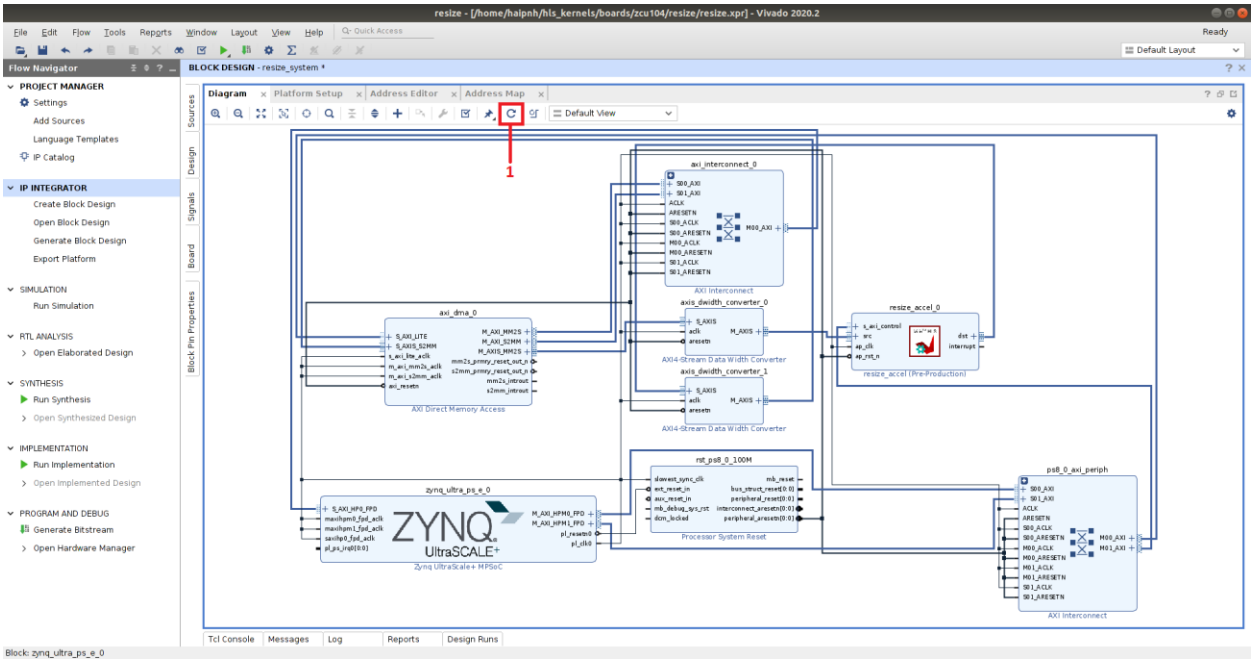
This is the result after wiring.



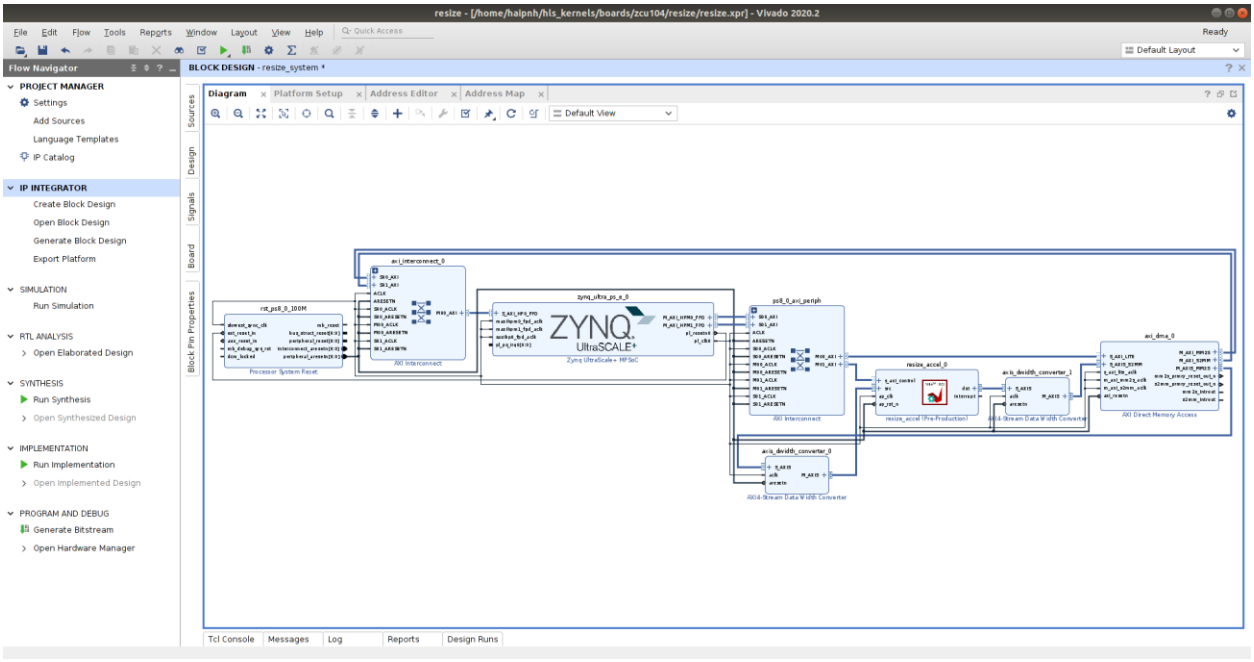
Step 27: Click **Run Connection Automation**, select **All Automation**, then click OK.



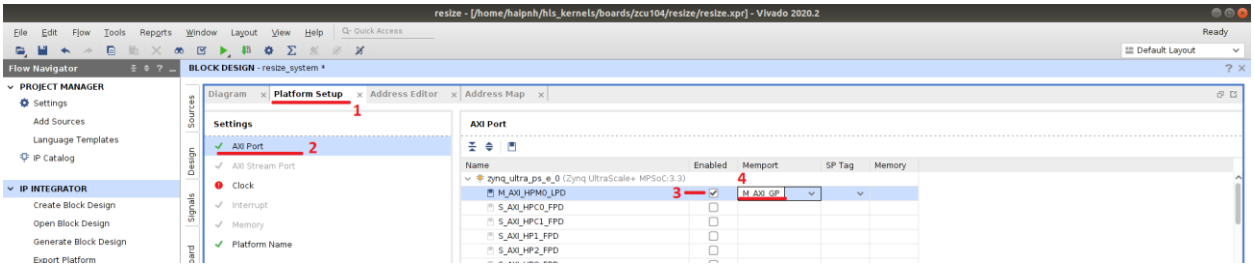
Step 28: The system is now fully wired. Click the **Regenerate Layout** icon.



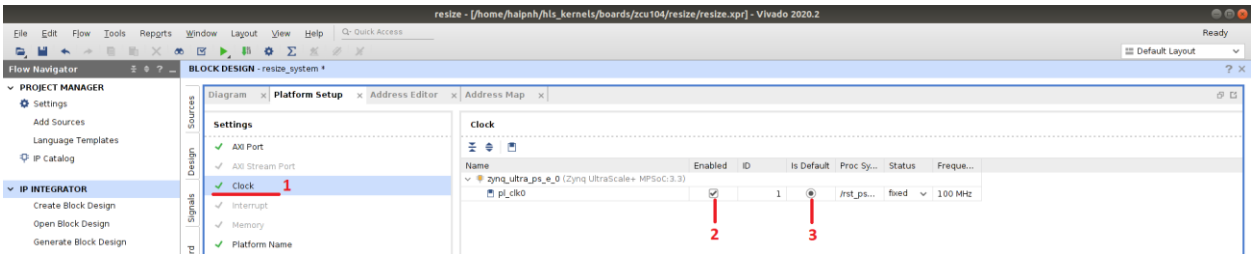
Here is the final layout.



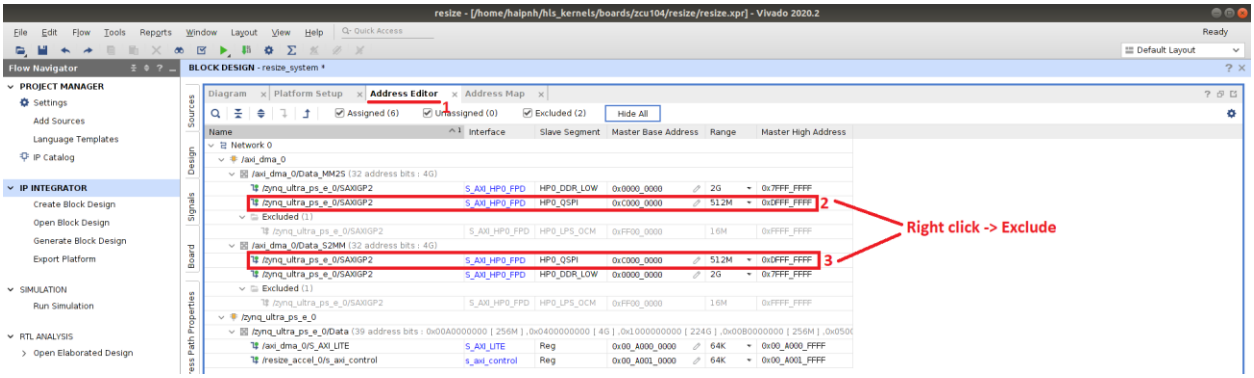
Step 29: Open **Platform Setup** ➔ **AXI Port** ➔ Enable **M_AXI_HPM0_LPD** ➔ Select Memport **M_AXI_GP**.



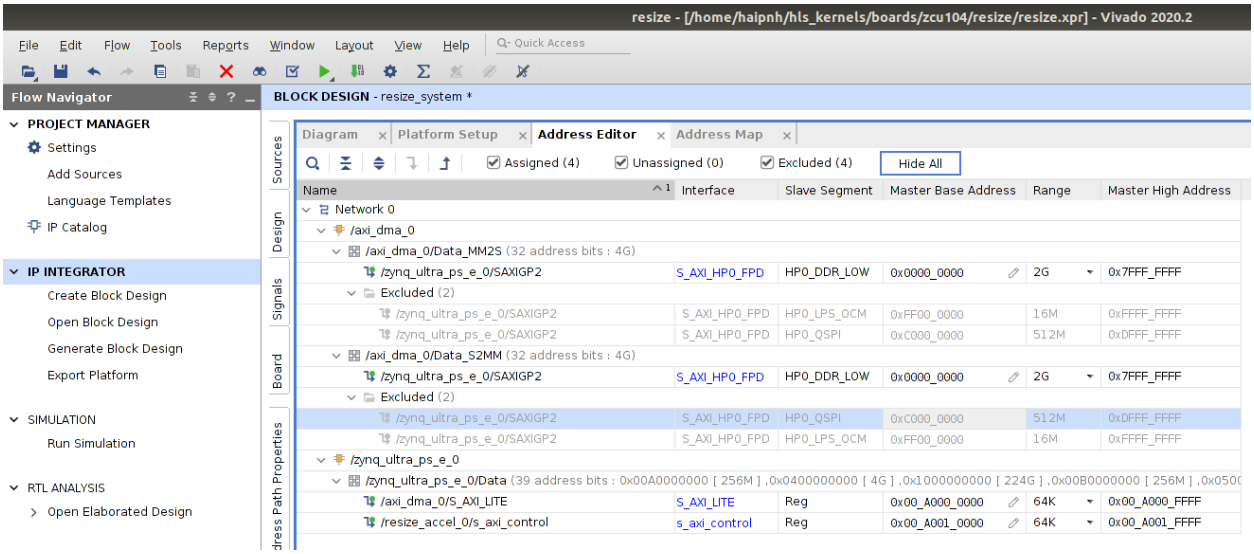
Step 30: On **Platform Setup**, open **Clock** ➔ Enable **pl_clk0** ➔ Select **Is Default**.



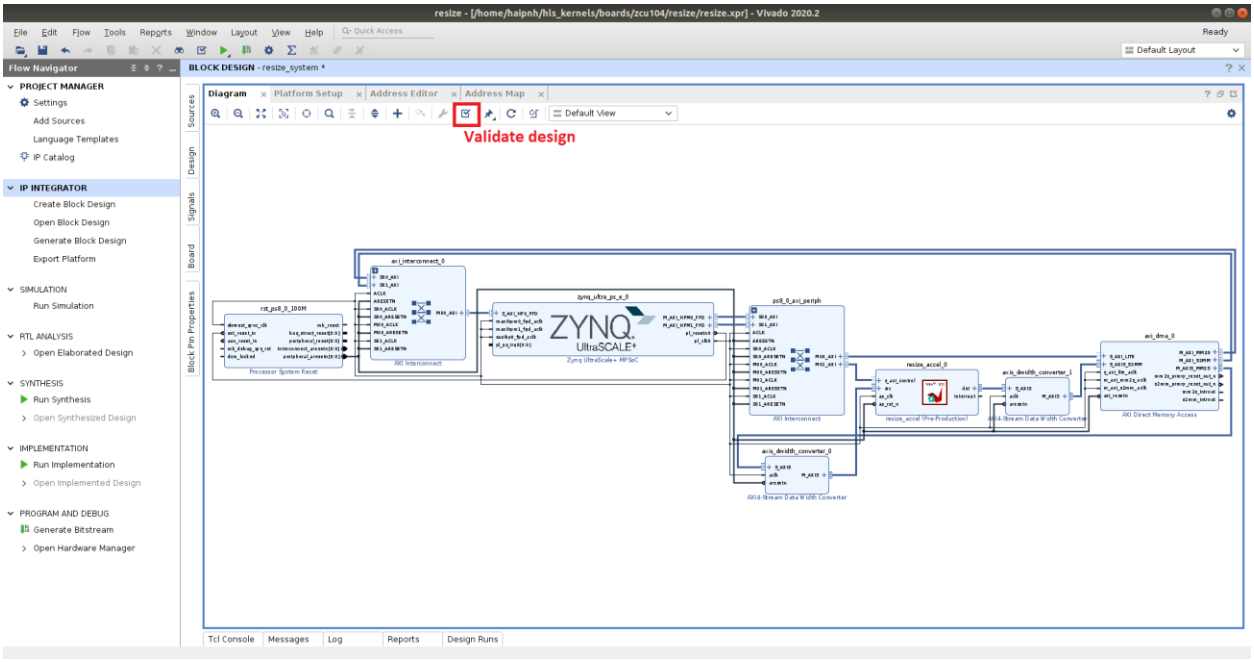
Step 31: Open **Address Editor**, right click on these **2 interfaces** using **HP0_QSPI** slave segment to exclude them.



The Address configuration should be as following.

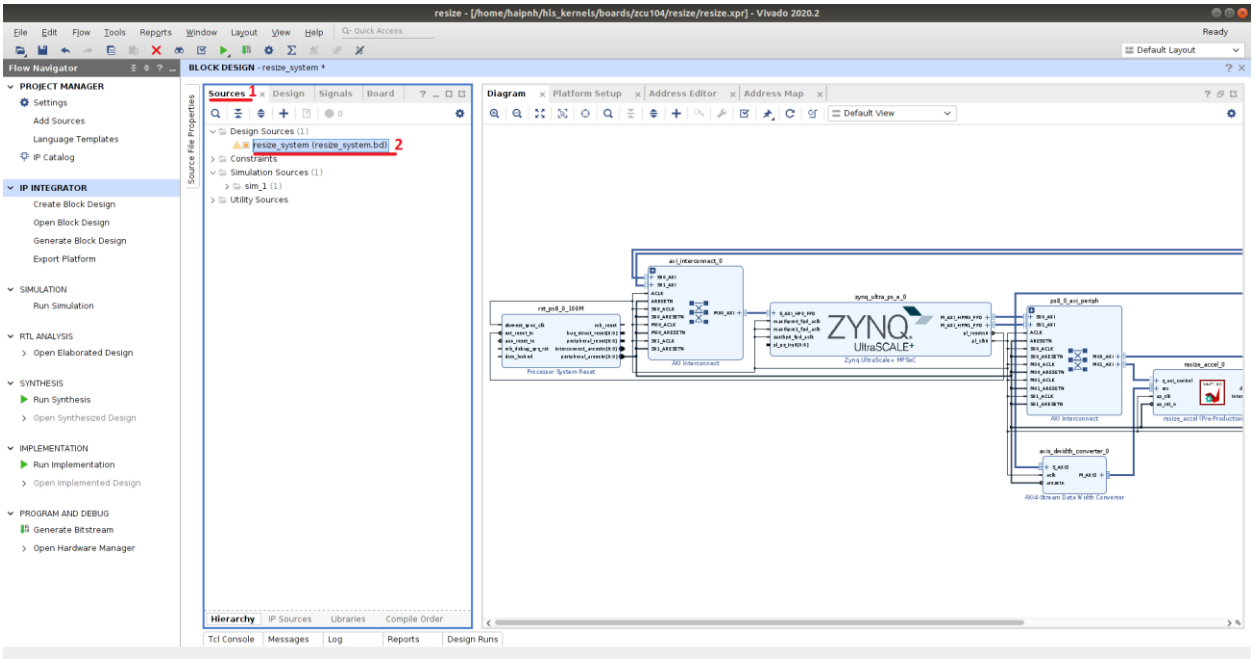


Step 32: Validate the design by clicking the icon as following.

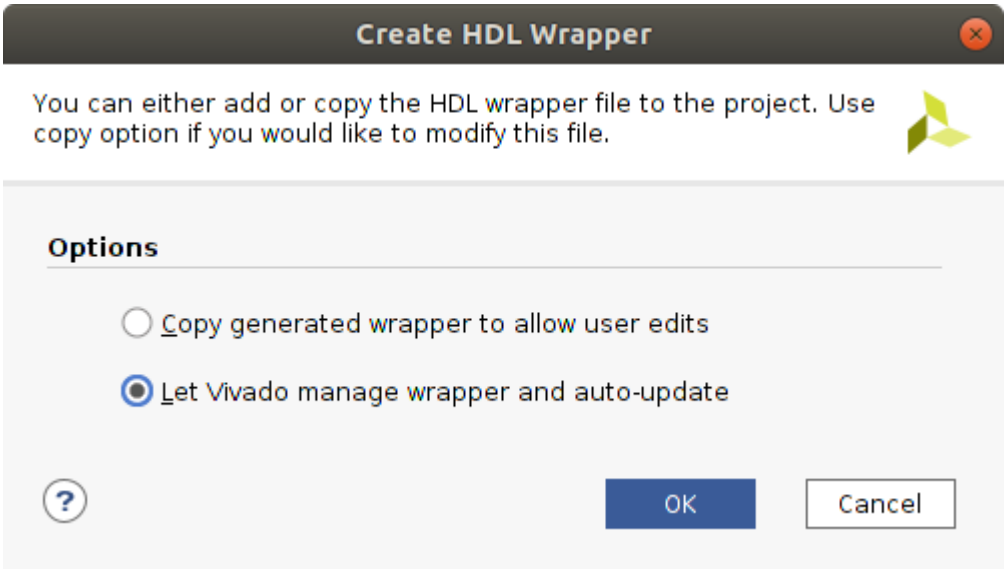


It should be successful.

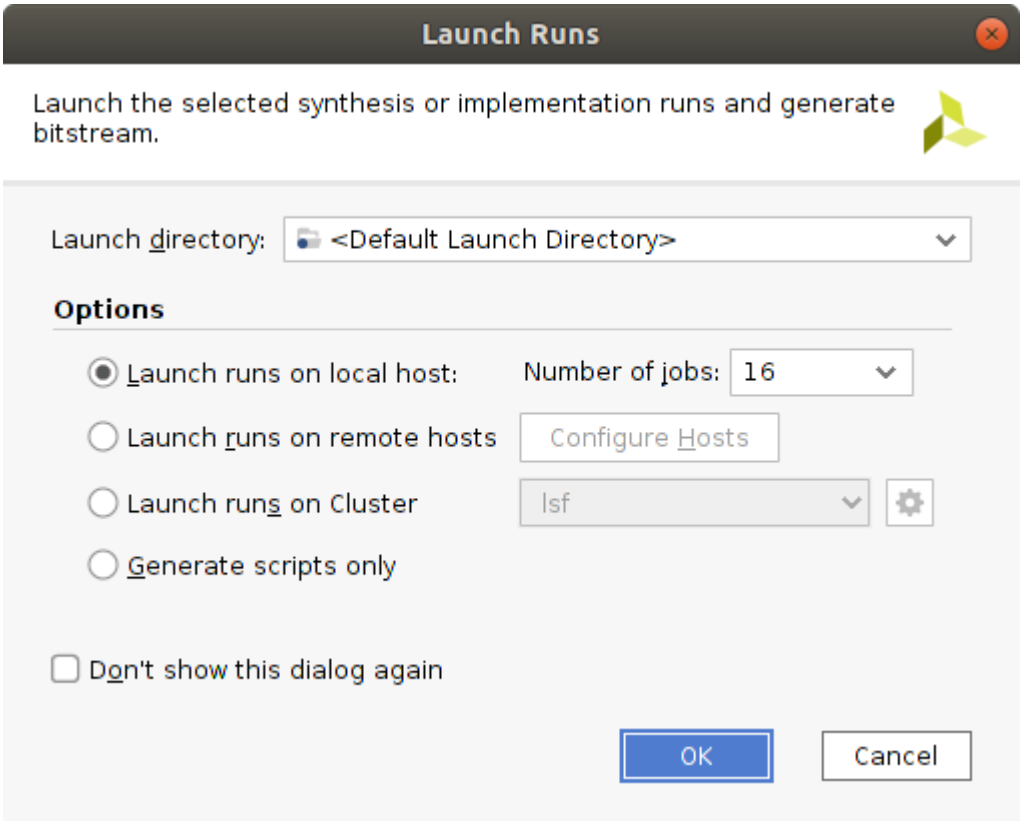
Step 33: In **Sources ➔ Design Sources**, right click on **resize_system** and choose **Create HDL Wrapper.**



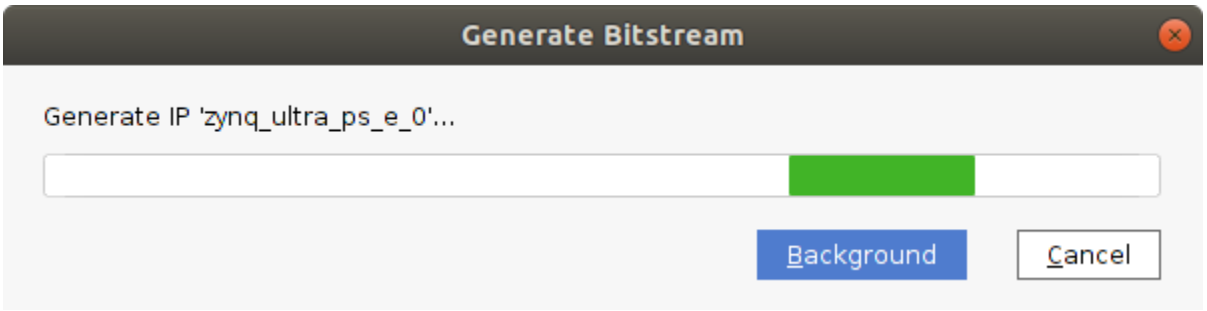
Step 34: Choose **Let Vivado manage wrapper and auto-update**.



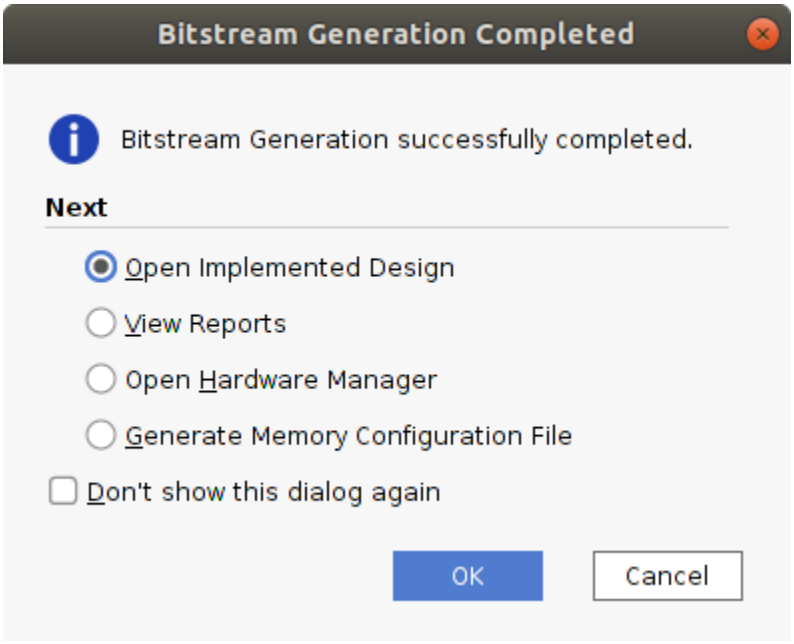
Step 35: Open **Flow ➔ Generate Bitstream** to generate the system bitstream. Keep it as following click **OK**.



Wait for the bitstream to be generated. It could take 30 minutes to hours depending on the performance of your host machine.



Step 36: A **Bitstream Generation Completed** dialog will display. You can **Cancel** it or review the implemented design, reports, etc. Vivado can be closed now.

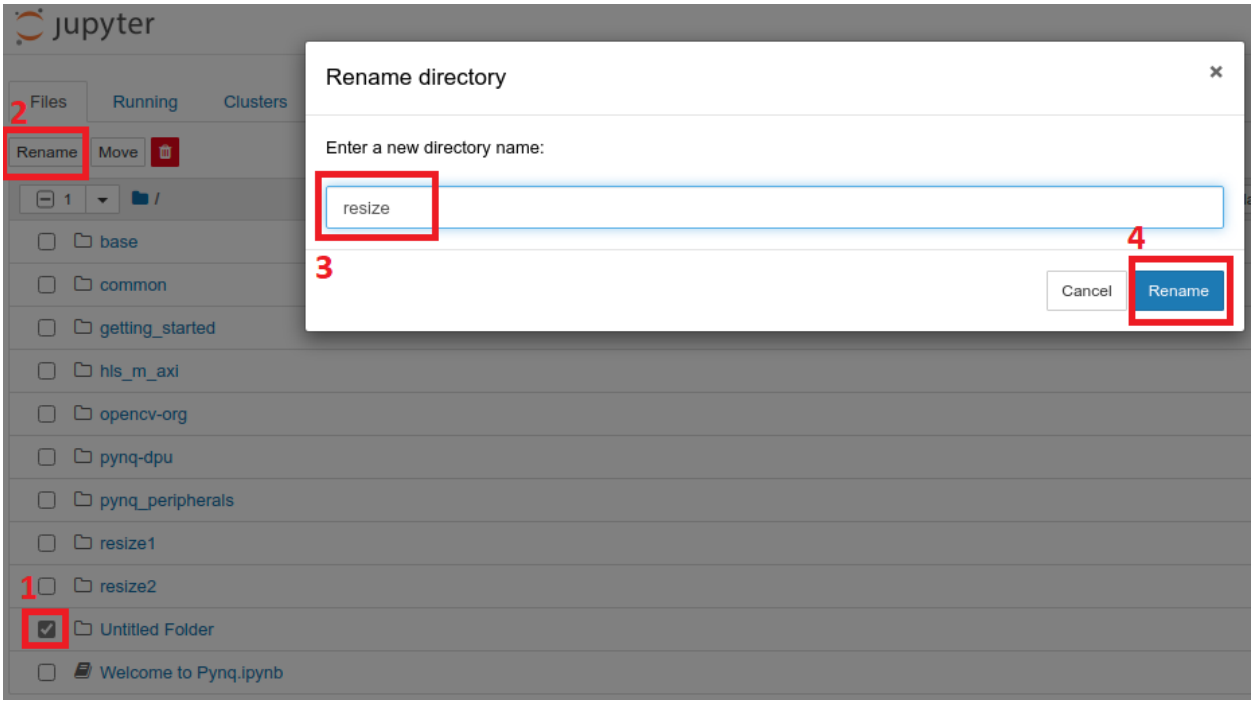


4.4. Run Jupyter Notebook example

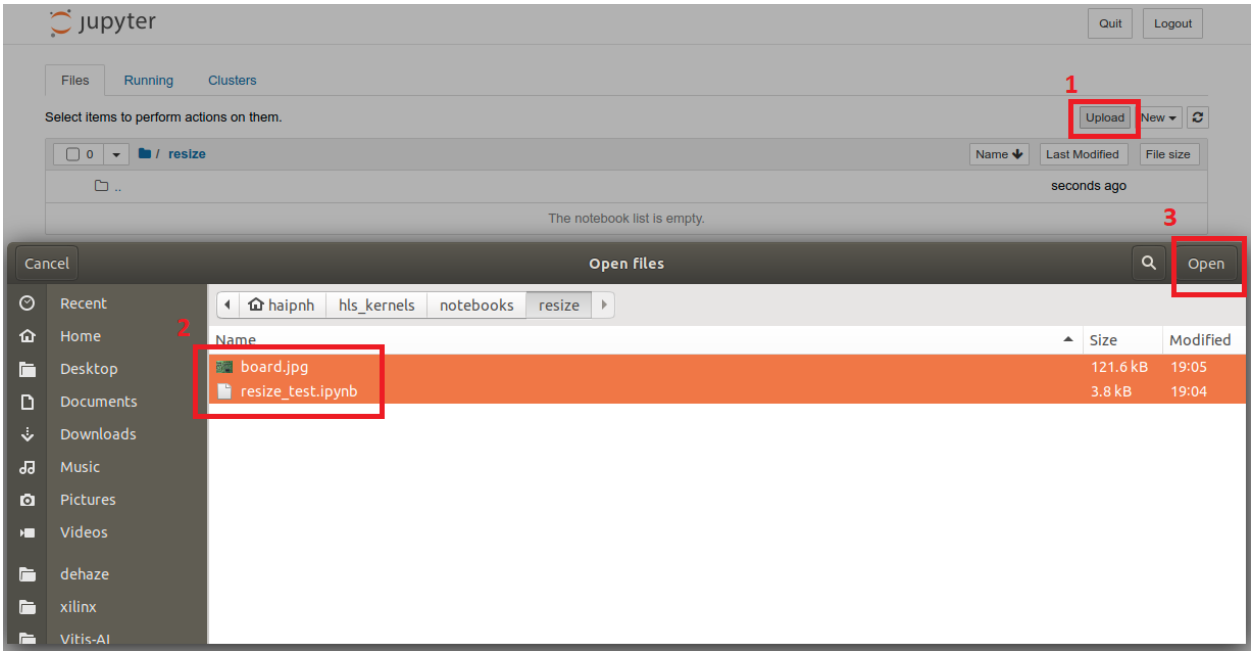
Step 1: Boot up the ZCU104, open web-browser and navigate to 192.168.2.99.

Step 2: Create a new folder as following.

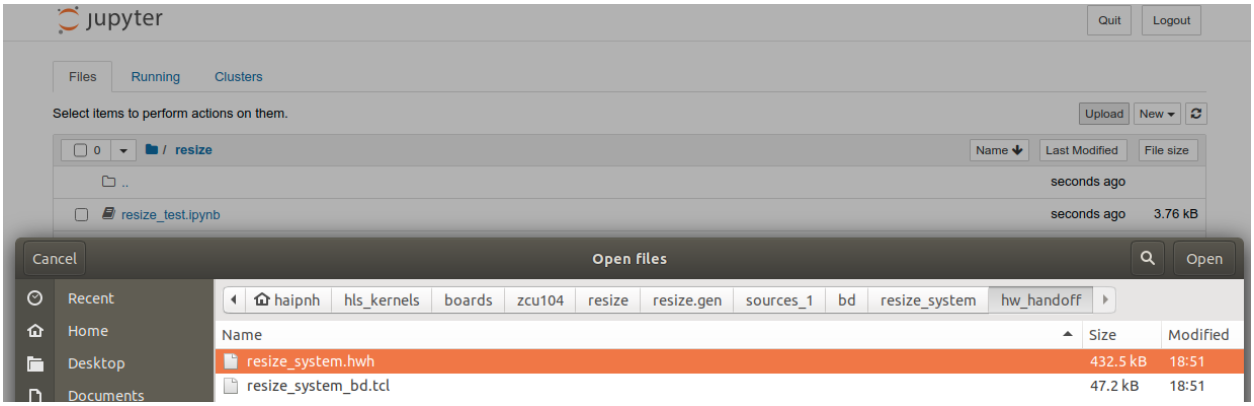
Step 3: Rename the folder from **Untitled Folder** to **resize** as following.



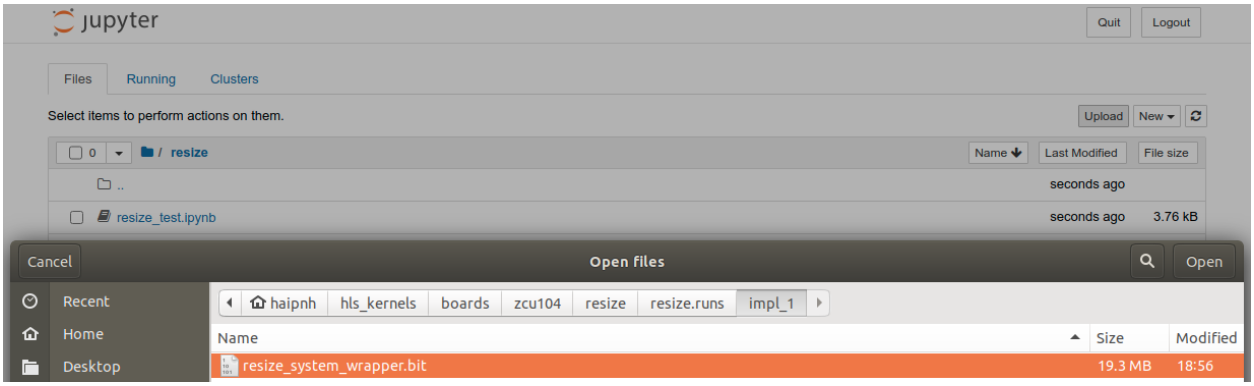
Step 4: Open **resize** folder, upload the example Jupyter notebook and sample image to this folder.



Step 5: Upload the **resize_system.hwh** file. This file is located at **$HOME/hls_kernels/boards/zcu104/resize/resize.gen/sources_1/bd/resize_system/hw_handoff**.



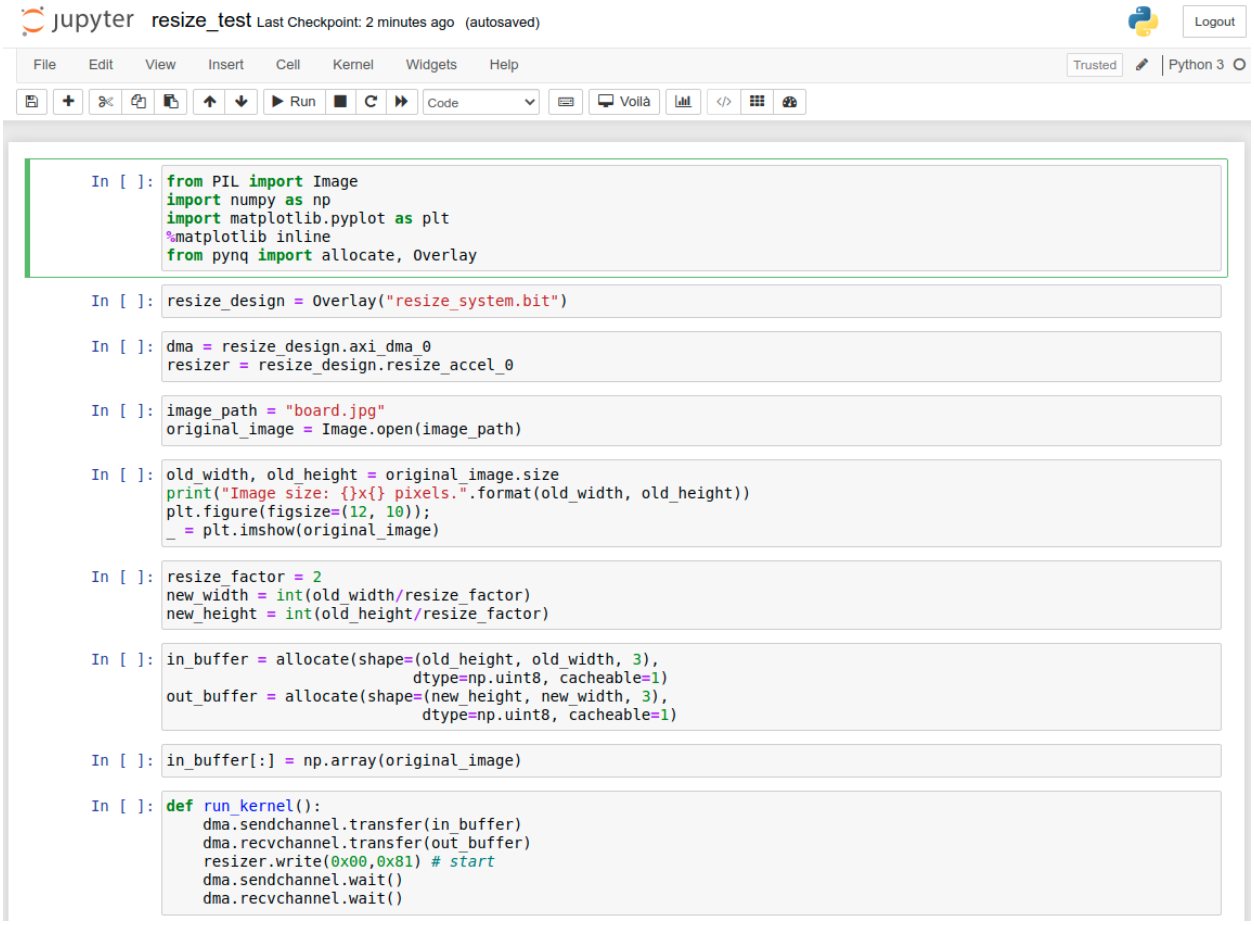
Step 6: Upload the **resize_system_wrapper.bit** file. This file is located at **$HOME/ hls_kernels/boards/zcu104/resize/resize.runs/impl_1**.



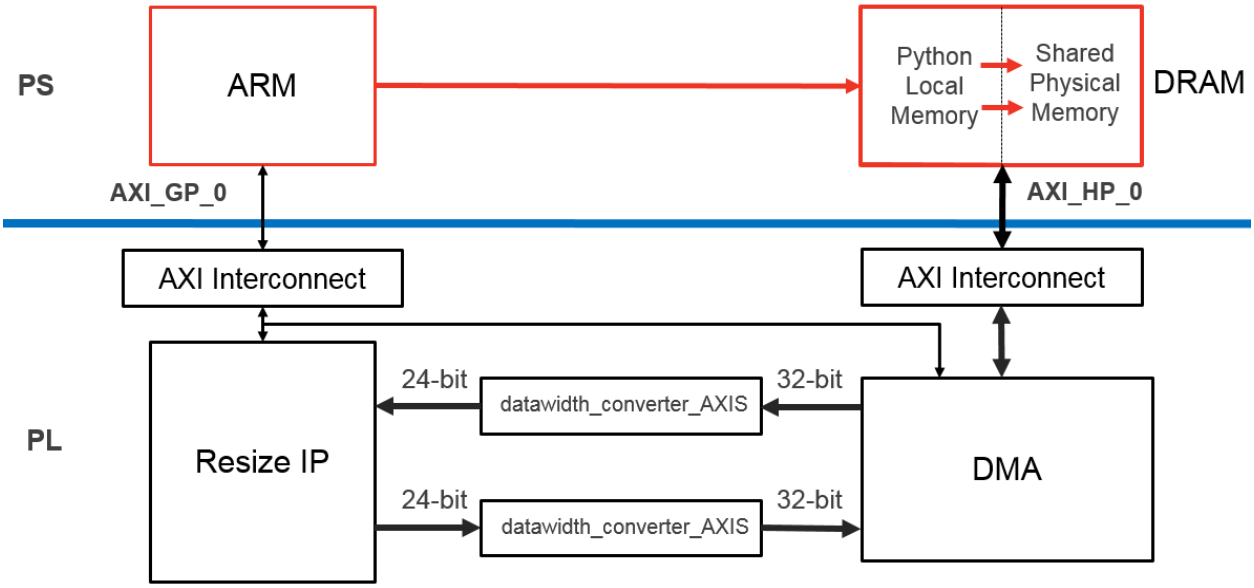
Step 7: Upload **resize_system_wrapper.bit** as **resize_system.bit**.

Required files should exist as following.

Step 8: Open **resize_test.ipynb** to examine.



In this example, 2 buffers will be allocated in PS memory and DMA block will transfer the data from PS memory to resize kernel to process, then transfer the output from the kernel back to PS memory.



Step 9: Run all the cells step-by-step. This is the final result.

```
In [11]: run_kernel()
         resized_image = Image.fromarray(out_buffer)

In [12]: print("Image size: {}x{} pixels.".format(new_width, new_height))
         plt.figure(figsize=(12, 10));
         _ = plt.imshow(resized_image)

Image size: 320x240 pixels.
```


```
In [13]: del in_buffer
         del out_buffer
```

5. The second example: RGB2YUV

We suggest practicing the first example to understand the workflow. From now, only key points will be listed.

5.1. Project Structure

At $HOME/hls_kernels directory, we will add new directories as following:

- boards/zcu104/rgb2yuv4
- ip/rgb2yuv4/src

```
hls_kernels
├── boards
│   └── zcu104
│       ├── resize
│       └── rgb2yuv4: New Vivado project's workspace
├── ip
│   ├── resize
│   │   └── src
│   └── rgb2yuv4: New Vitis HLS project's workspace
│       └── src: Kernel source code
└── Vitis_Libraries
    ├── vision: Vitis Vision library
    └── <other domains>
```

5.2. Acceleration Kernel Synthesis using Vitis HLS

Step 1: Copy the source code file rgb2yuv4_accel.cpp to $HOME/hls_kernels/ip/rgb2yuv4/src.

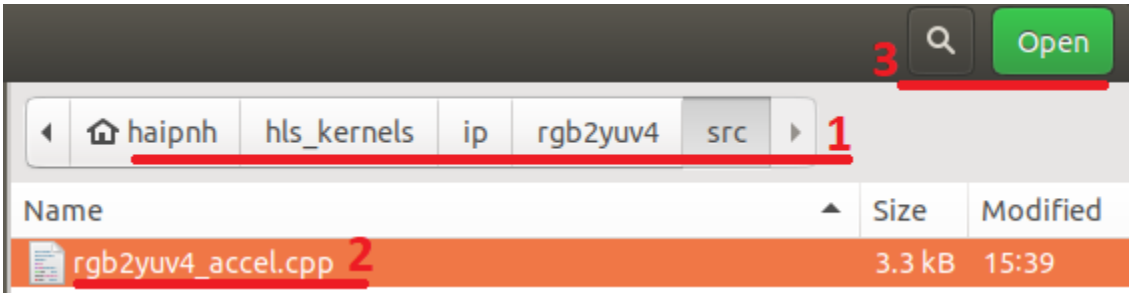
Step 2: Open Vitis HLS

```
cd $HOME/hls_kernels/ip/rgb2yuv4
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vitis_hls
```

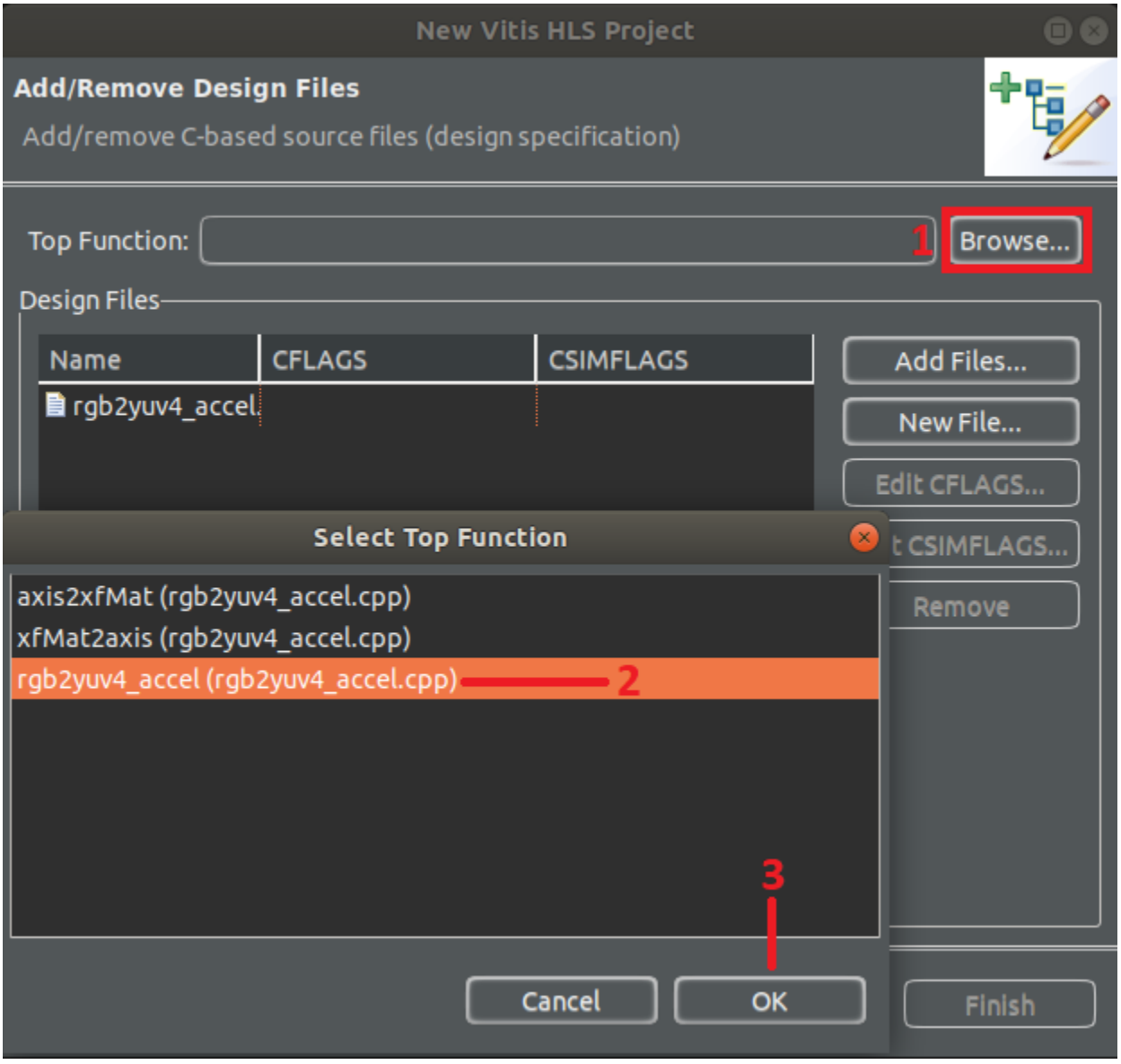
Step 3: Create new project with following information.

- Project name: **rgb2yuv4**
- Location: keep it default, check if it is $HOME/hls_kernels/ip/rgb2yuv4

Step 4: Add kernel source code file to the project: **rgb2yuv4_accel.cpp**.

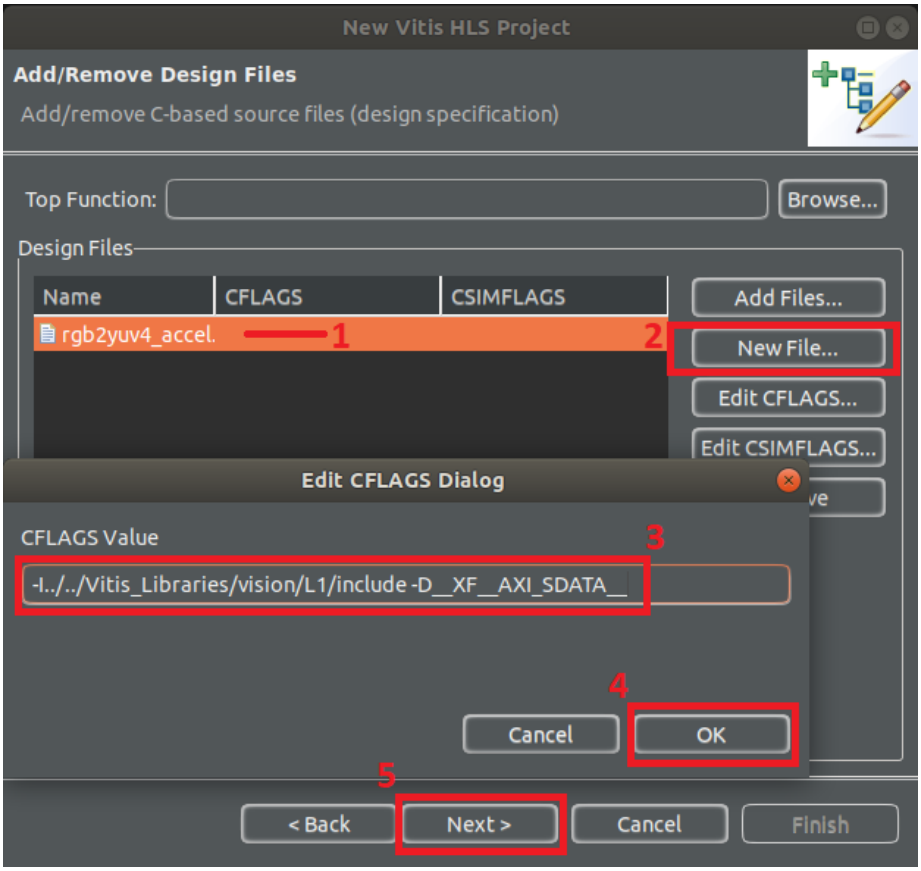


Step 5: Choose top function: **rgb2yuv_accel (rgb2yuv4_accel.cpp)**



Step 6: Add following CFLAGS of the source code to include the Vitis Vision Library.

```
-I../../Vitis_Libraries/vision/L1/include -D__XF__AXI_SDATA__
```



Step 7: Skip choosing the testbench ➔ Click **Next**.

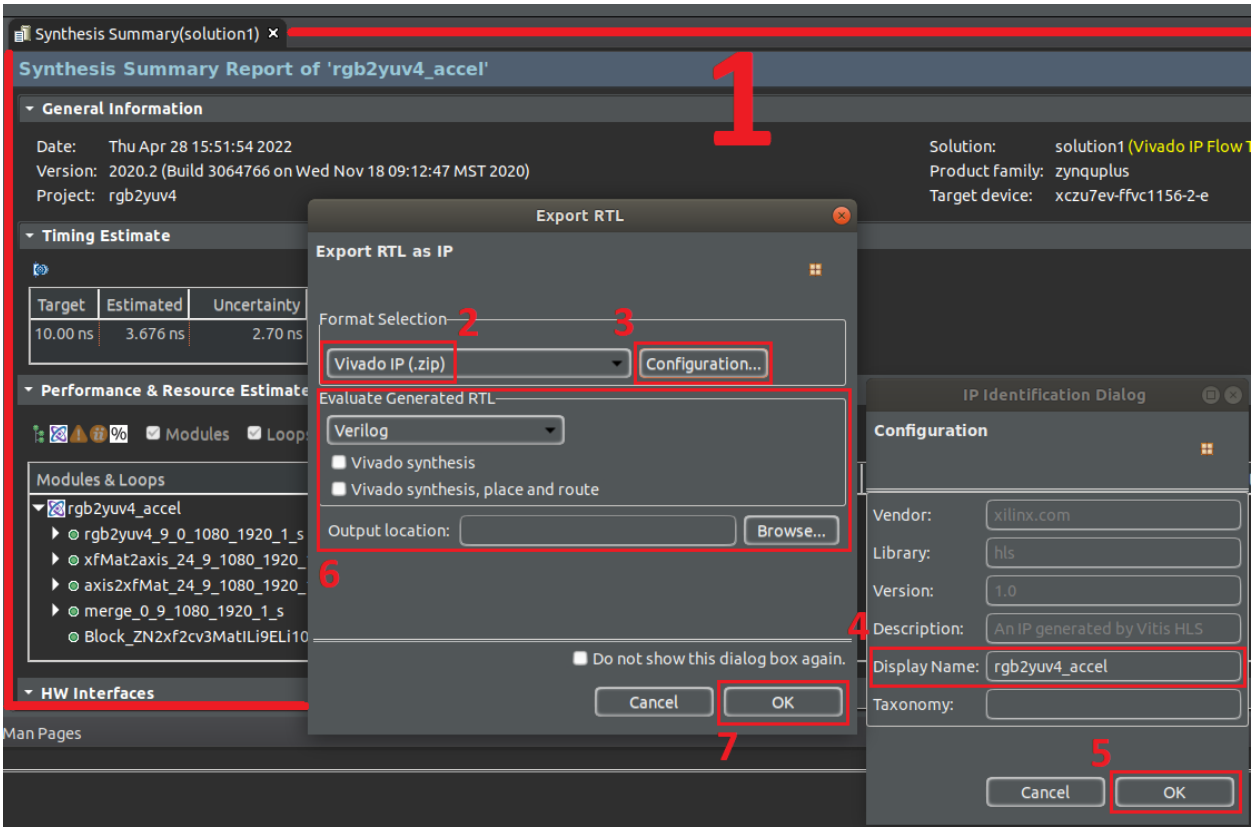
Step 8: In **Solution Configuration** dialog, set the **Uncertainty** to 27.0%. Then select **ZCU104** board as target, which will result as selected **Part: xczu7ev-ffvc1156-2-e**.

Step 9: In **Explorer**, open the **rgb2yuv4/Source/rgb2yuv4_accel.cpp** to review.

Step 10: Start Synthesis: **Solution ➔ Run C Synthesis ➔ Active Solution**.

Step 11: Review synthesis report in **rgb2yuv4/solution1/syn/report/csynth.rpt**.

Step 12: After the synthesis is finished and **Synthesis Summary** will show, open **Solution ➔ Export RTL** to export the kernel. Configure as following.



Step 13: Wait for the exporting and the message **"Finished export RTL"** in the console. Now we can close Vitis HLS.

5.3. System Design and Generate Bitstream using Vivado

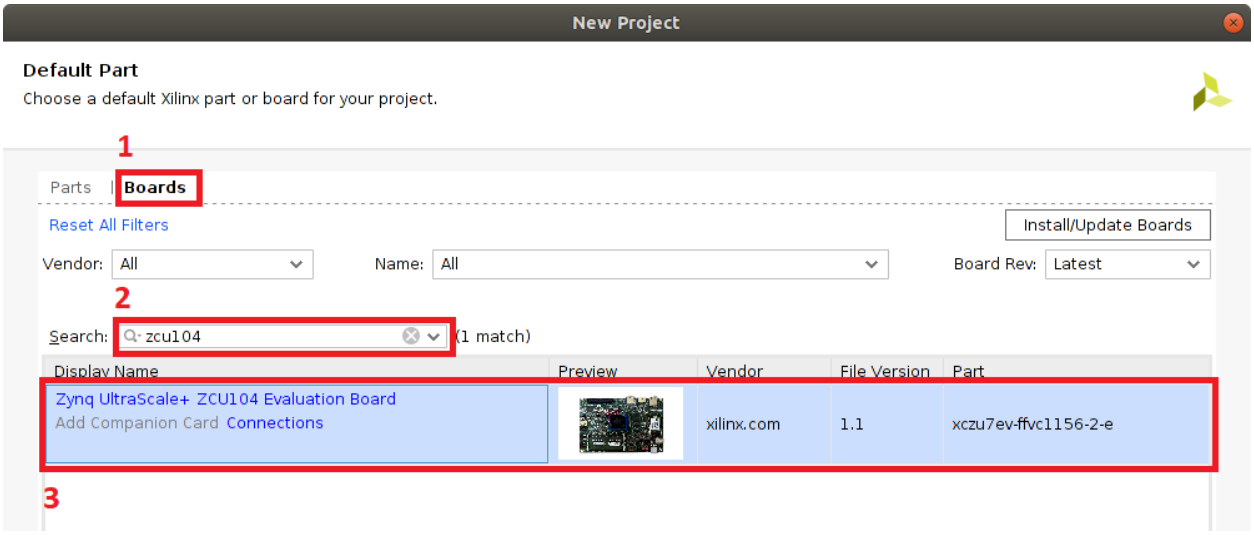
Step 1: Open Vivado

```
cd $HOME/hls_kernels/boards/zcu104/rgb2yuv4
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vivado
```

Step 2: Create new Vivado project with following information.

- Project name: **rgb2yuv4**
- Project location: in $HOME/hls_kernels/boards/zcu104/rgb2yuv4
- Uncheck **Create project subdirectory**
- Project Type: **RTL Project**
    - Check **Do not specify sources at this time**
    - Check **Project is an extensible Vitis platform**

Step 3: Select **Boards** ➔ Search for **zcu104** ➔ Then select the **Zynq UltraScale+ ZCU104 Evaluation Board** ➔ Click **Next**.

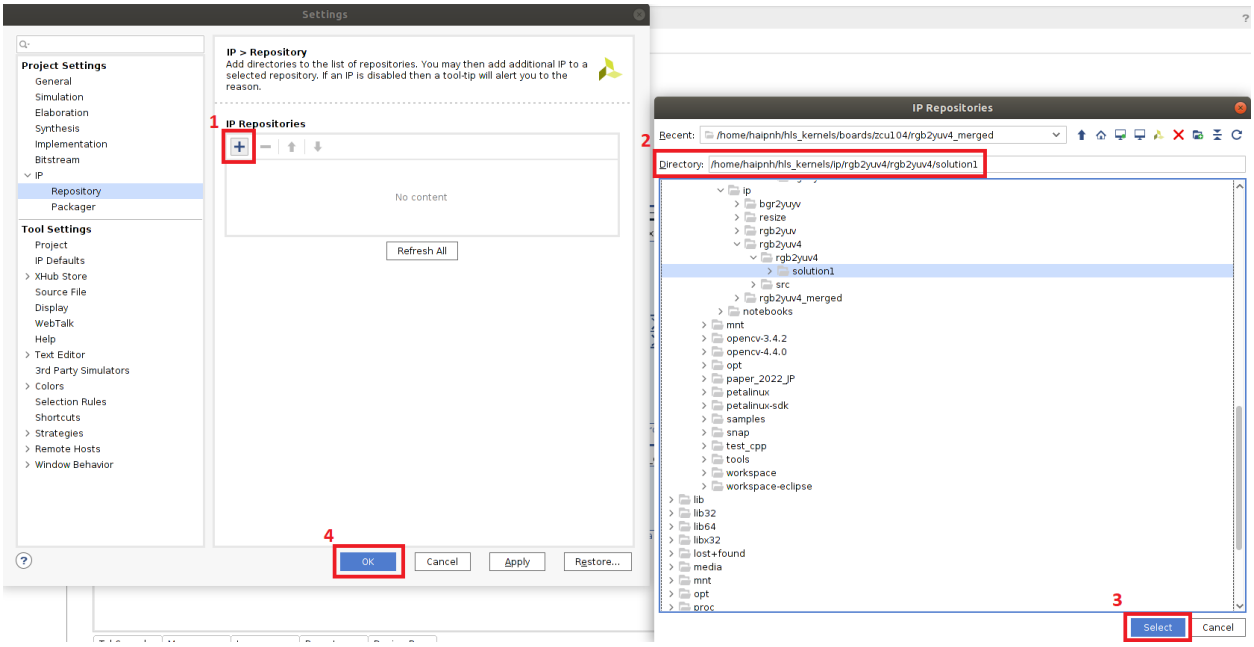


Step 4: Review the **New Project Summary**. Then click **Finish**.

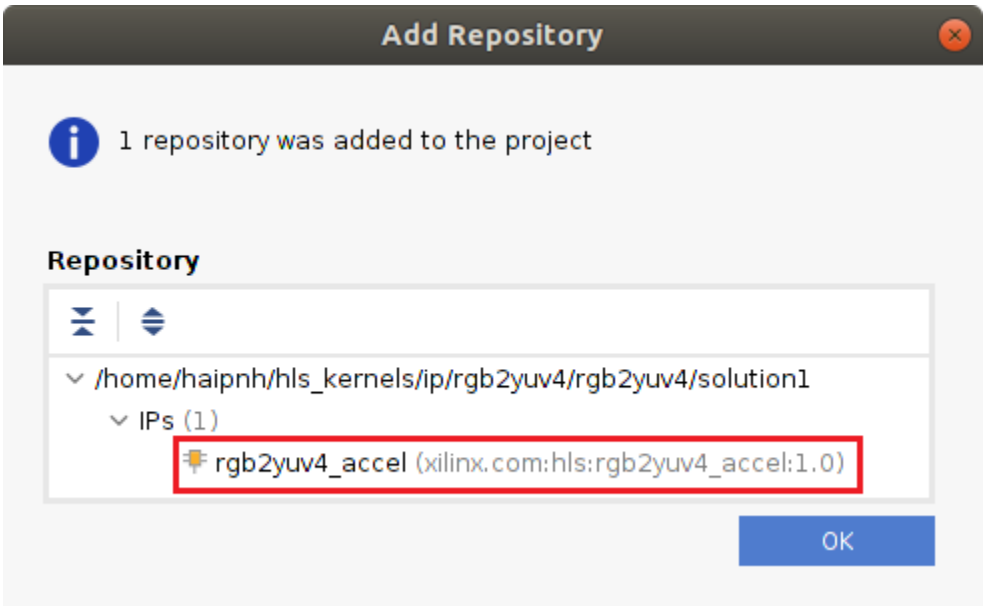
Step 5: Click **Tools** ➔ **Settings…** to open **Project Settings** ➔ **IP** ➔ **Repository**.

Step 6: Follow these steps to add **User IP Repositories**.

The directory is **$HOME/hls_kernels/ip/rgb2yuv4/rgb2yuv4/solution1** (as below screenshot).



Step 7: The exported IP should be found: **rgb2yuv4_accel.**



Step 8: Click **IP INTEGRATOR ➔ Create Block Design**. Set the **Design name** to **rgb2yuv4_system**.

Step 9: Click **IP INTEGRATOR ➔ Open Block Design**.

Step 10: Add one **Zynq UltraScale+ MPSoC** block, instanced as **zynq_ultra_ps_e_0**. Then **Run Block Automation** to **Apply Board Preset**.

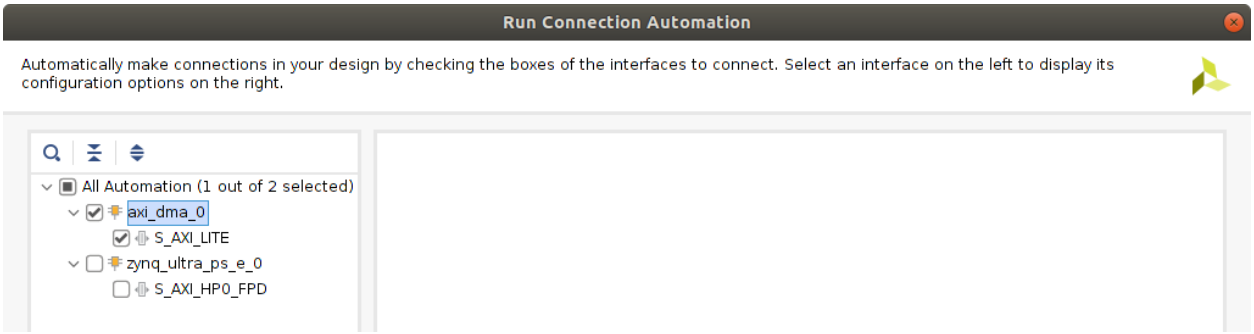
Step 11: Customize **zynq_ultra_ps_e_0** with following configuration.

- **PS-PL Configuration ➔ PS-PL Interfaces ➔ Slave Interface ➔ AXI HP**
    - Enable **AXI HP0 FPD**
        - **Data Width**: 128

Step 12: Add a **AXI Direct Memory Access**, instanced as **axi_dma_0**. Customize it with following configuration.

- Disable **Enable Scatter Gather Engine**
- Width of Buffer Length Register: 26 bits
    - Address Width: 32 bits
- Enable Read Channel
    - Max Burst Size: 256
- Enable Write Channel
    - Max Burst Size: 256

Step 13: Click **Run Connection Automation** with following configuration.



Step 14: Add a **rgb2yuv4_accel** block.

Step 15: Add a **AXI4-Stream Data Width Converter**, instanced as **axis_dwidth_converter_0**. Then customize it with following configuration.

- Slave Interface TDATA Width (bytes): 4
- Master Interface TDATA Width (bytes): 3

Step 16: Add a **AXI4-Stream Data Width Converter**, instanced as **axis_dwidth_converter_1**. Then customize it with following configuration.

- Slave Interface TDATA Width (bytes): 3
- Master Interface TDATA Width (bytes): 4

Step 17: Add a **AXI Interconnect**, instanced as **axi_interconnect_0**. Then customize it with following configuration.

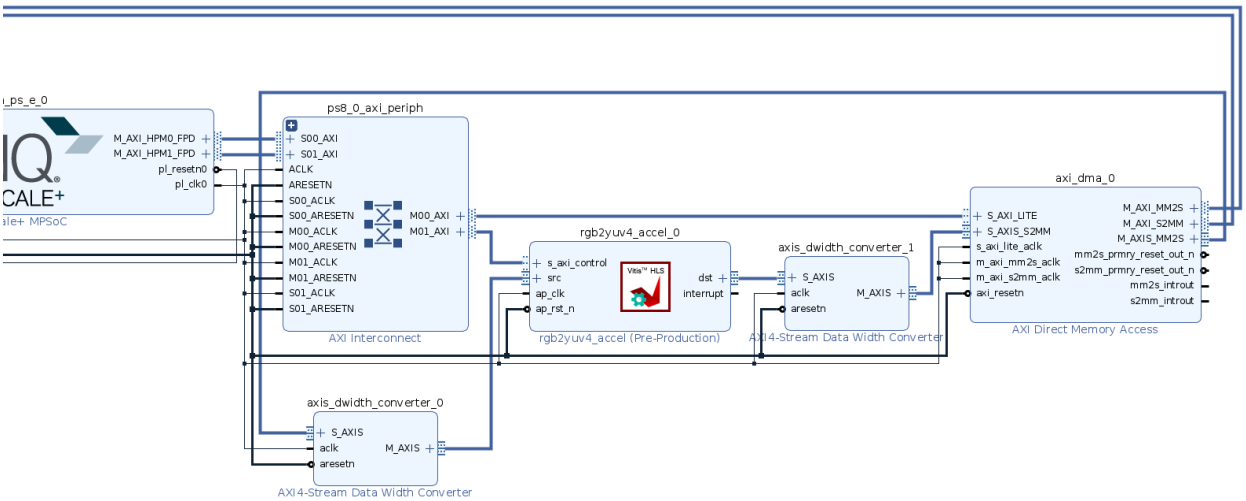
- Number of Slave Interfaces: 2
- Number of Master Interfaces: 1
- Interconnect Optimization Strategy: Maximize Performance

Step 18: Wire the interfaces of these blocks as following.

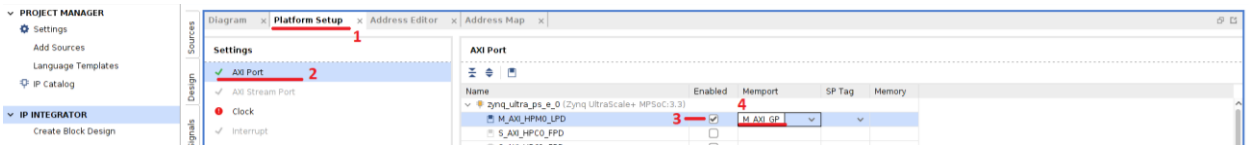
- zynq_ultra_ps_e_0 / S_AXI_HP0_FPD <-> axi_interconnect_0 / M00_AXI
- axi_interconnect_0 / S00_AXI <-> axi_dma_0 / M_AXI_MM2S
- axi_interconnect_0 / S01_AXI <-> axi_dma_0 / M_AXI_S2MM
- axi_dma_0 / M_AXIS_MM2S <-> axis_dwidth_converter_0 / S_AXIS
- axi_dma_0 / S_AXIS_S2MM <-> axis_dwidth_converter_1 / M_AXIS
- axis_dwidth_converter_0 / M_AXIS <-> rgb2yuv4_accel_0 / src
- axis_dwidth_converter_1 / S_AXIS <-> rgb2yuv4_accel_0 / dst

Step 19: Click **Run Connection Automation**, select **All Automation**, then click **OK**.

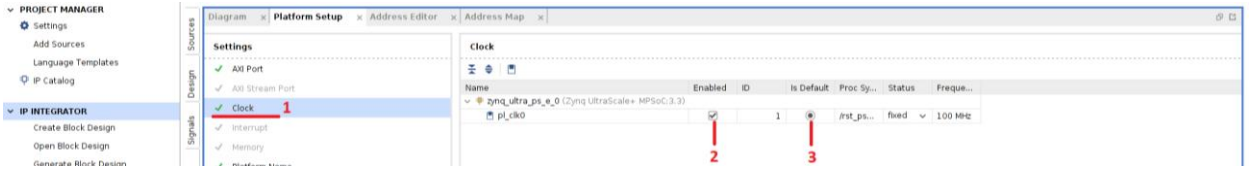
Step 20: After fully wiring the system, it should result as below.



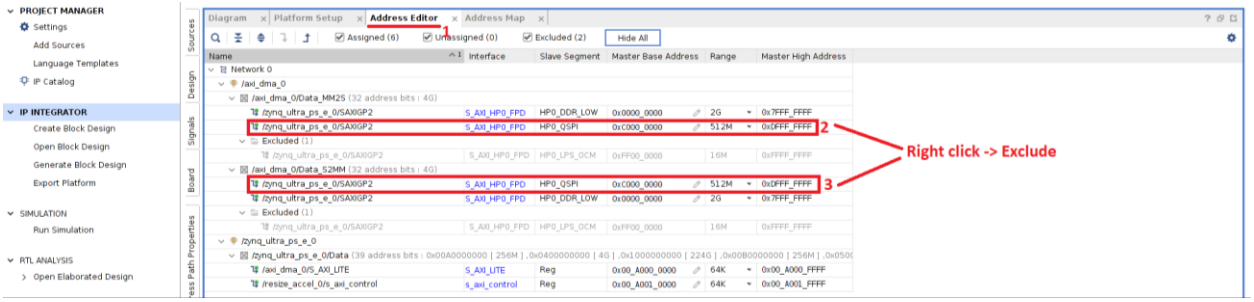
Step 21: Open **Platform Setup ➔ AXI Port ➔** Enable **M_AXI_HPM0_LPD ➔** Select Memport **M_AXI_GP**.



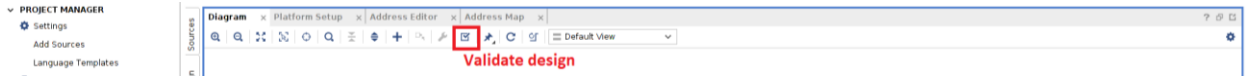
Step 22: On **Platform Setup**, open **Clock ➔** Enable **pl_clk0 ➔** Select **Is Default**.



Step 23: Open **Address Editor**, right click on these 2 interfaces using **HP0_QSPI** slave segment to exclude them.



Step 24: Validate the design.



Step 25: In **Sources ➔ Design Sources,** right click on **rgb2yuv4_system** and choose **Create HDL Wrapper.**

Step 25: Generate the bitstream, open **Flow ➔ Generate Bitstream**.

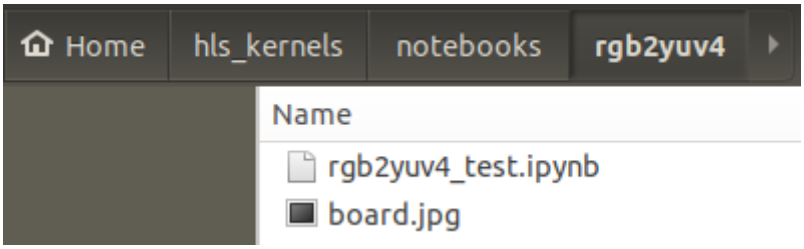
Step 26: A **Bitstream Generation Completed** dialog will display. You can **Cancel** it or review the implemented design, reports, etc. Vivado can be closed now.

5.4. Run Jupyter Notebook example

Step 1: Open Jupyter home page.

Step 2: Create New Folder, rename it to **rgb2yuv4**, then open it.

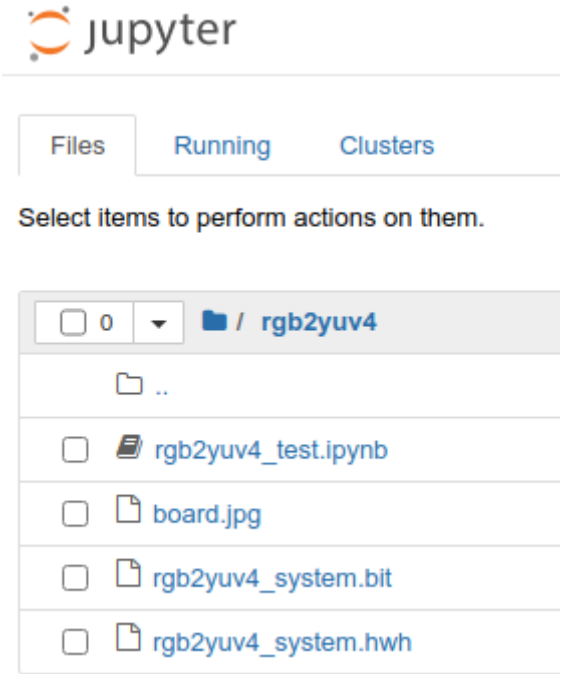
Step 3: Upload the example Jupyter notebook and sample image to the folder.



Step 4: Upload these files from $HOME/hls_kernels/boards/zcu104/rgb2yuv4

- rgb2yuv4.gen/sources_1/bd/rgb2yuv4_system/hw_handoff/**rgb2yuv4_system.hwh**
- rgb2yuv4.runs/impl_1/**rgb2yuv4_system_wrapper.bit**
  ➔ Rename it as **rgb2yuv4_system.bit**

Step 5: Required files should exist as following.



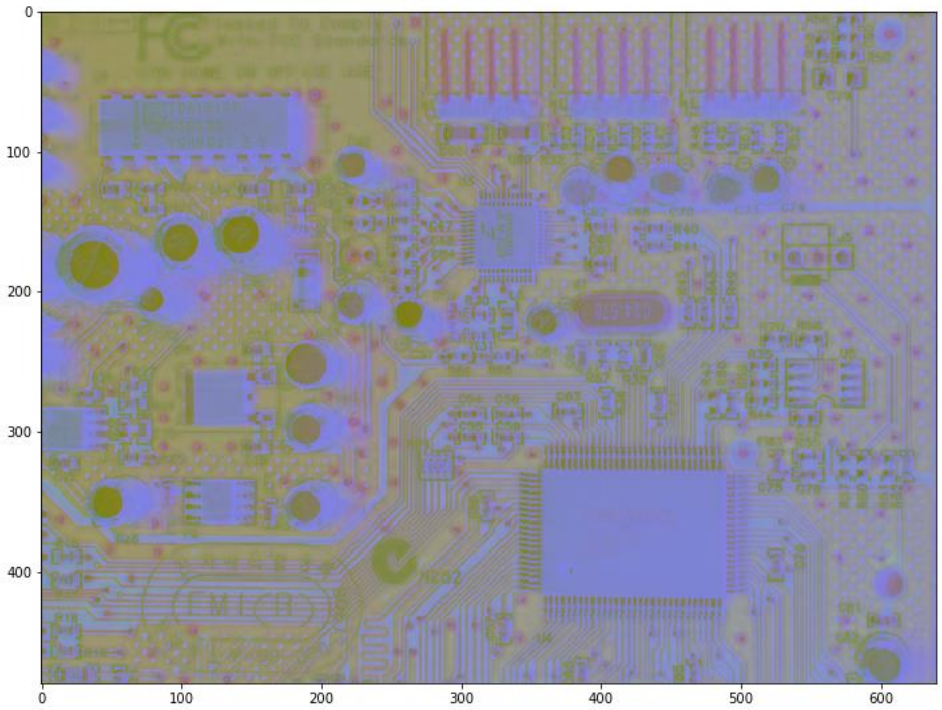
Step 6: Open **rgb2yuv4_test.ipynb** to examine, then run it.

The result should be as following.

```
In [10]: run_kernel()

In [11]: output[:] = out_buffer

In [12]: plt.figure(figsize=(12, 10));
         _ = plt.imshow((output * 255).astype(np.uint8))
```




```
In [13]: del in_buffer
         del out_buffer
```

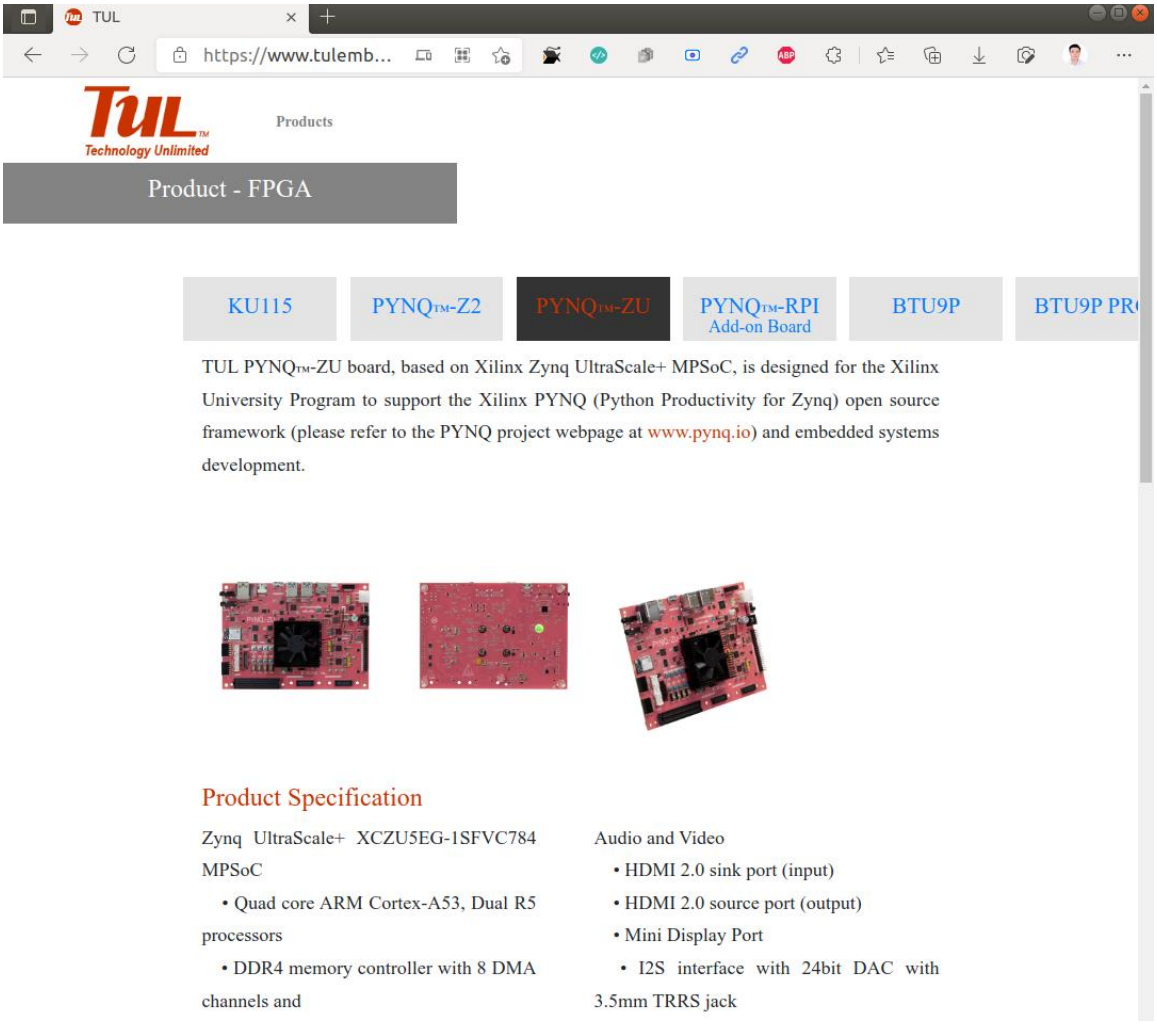
6. Run on other Zynq UltraScale+ MPSoC devices

6.1. Getting started to TUL Embedded PYNQ ™ -ZU Board

Once an acceleration kernel is developed and exported to RTL, it is reusable that can be employed in the system design on various Zynq UltraScale+ MPSoC devices.

We examine that capability using TUL Embedded PYNQ ™ -ZU Board. This device is based on XCZU5EG-1SFVC784, the same Zynq UltraScale+ MPSoC platform with ZCU104.

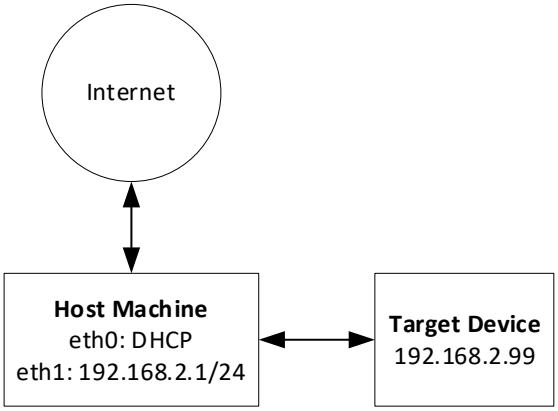
The detailed information about this product can be found at: https://www.tulembedded.com/FPGA/ProductsPYNQ-ZU.html (Last access: April 29, 2022).



User is recommended to download and read other documents to know more about the product, such as User Manual, Datasheet, Schematics, etc. However, we can skip this step and do it later.

6.2. System Setup

Setup the development system as following.



On Host Machine, we can use an USB to Ethernet adapter to implement the eth1 interface. Set it to 192.168.2.1/24 that we can connect to the target device through its default IP address.

Since the PYNQ-ZU does not have built-in RJ45 port, user might use an USB to Ethernet adapter, then connect the PYNQ-ZU to the Host Machine.

Notice that the Host Machine is already set up in the section 2. Host Machine Setup.

6.3. Target Device Setup

6.3.1. Download PYNQ image v2.7

Download the PYNQ image version 2.7 for PYNQ-ZU from this official link https://bit.ly/pynqzu_2_7, save it as $HOME/pynq_zu_v2.7.0.zip.
Or search for **PYNQ-ZU v2.7.0 SDCard image** from https://github.com/Xilinx/PYNQ/releases.

6.3.2. Flash the PYNQ image on to SD Card

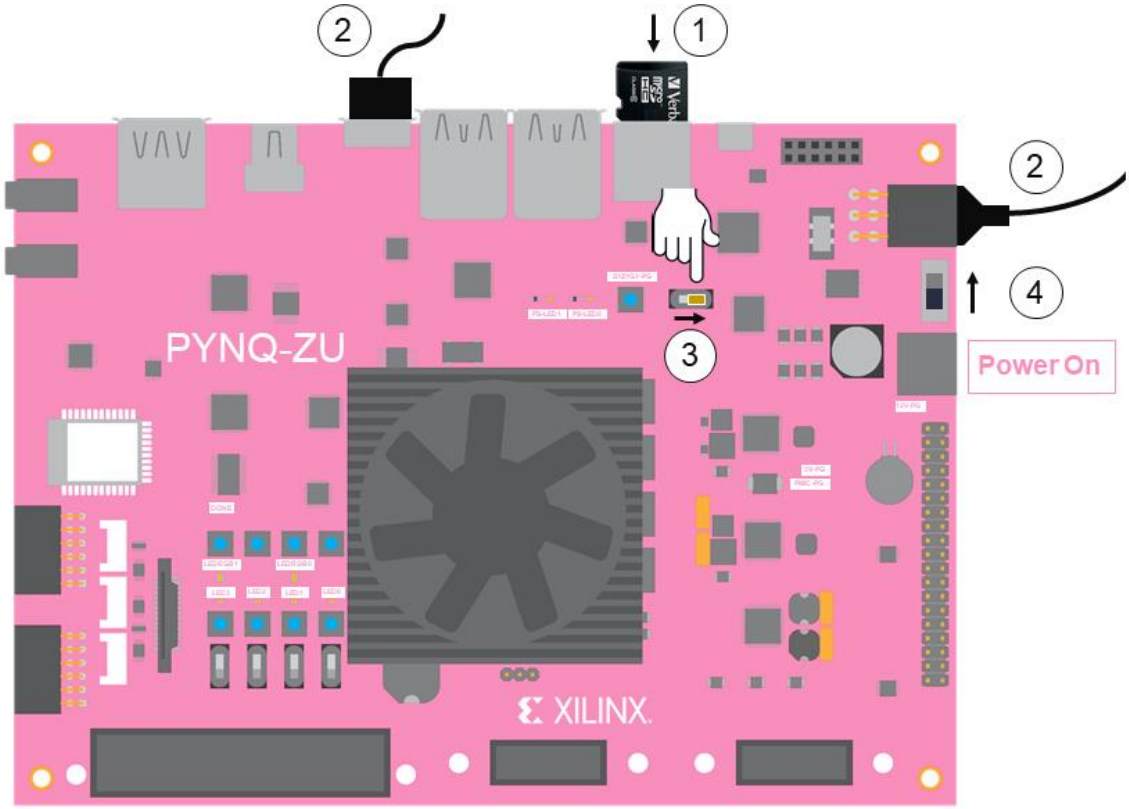
User follows the instructions from 3.2. Flash the PYNQ image onto SD card to flash the image pynq_zu_v2.7.0.zip onto SD Card (8GB or more), using balenaEtcher tool.

6.3.3. Hardware Configuration

Prerequisites

- PYNQ-ZU board
- Micro SD card (flashed with PYNQ image)
- Power supply for PYNQ-ZU board
- Optional: Micro USB cable (for serial terminal)

Setup the board



1. Insert the Micro SD card (pre-loaded with the PYNQ-ZU image)
2. Connect a Micro USB 3.0 cable to the board and your computer, and plug in the power cable
3. Set the Switch to the SD position to boot from the SD card
4. Power on the board
5. PYNQ Boot sequence
6. After you setup and power on the board, you should see the following sequence:
    - Status LEDs will turn on. Check the the “12V-PG” LED turns on. This indicates the board is getting power. You will also see one of the white LEDs (PS-LED0) flash in a

"heartbeat" pattern. This is a good indication the board is alive and the boot is in progress.

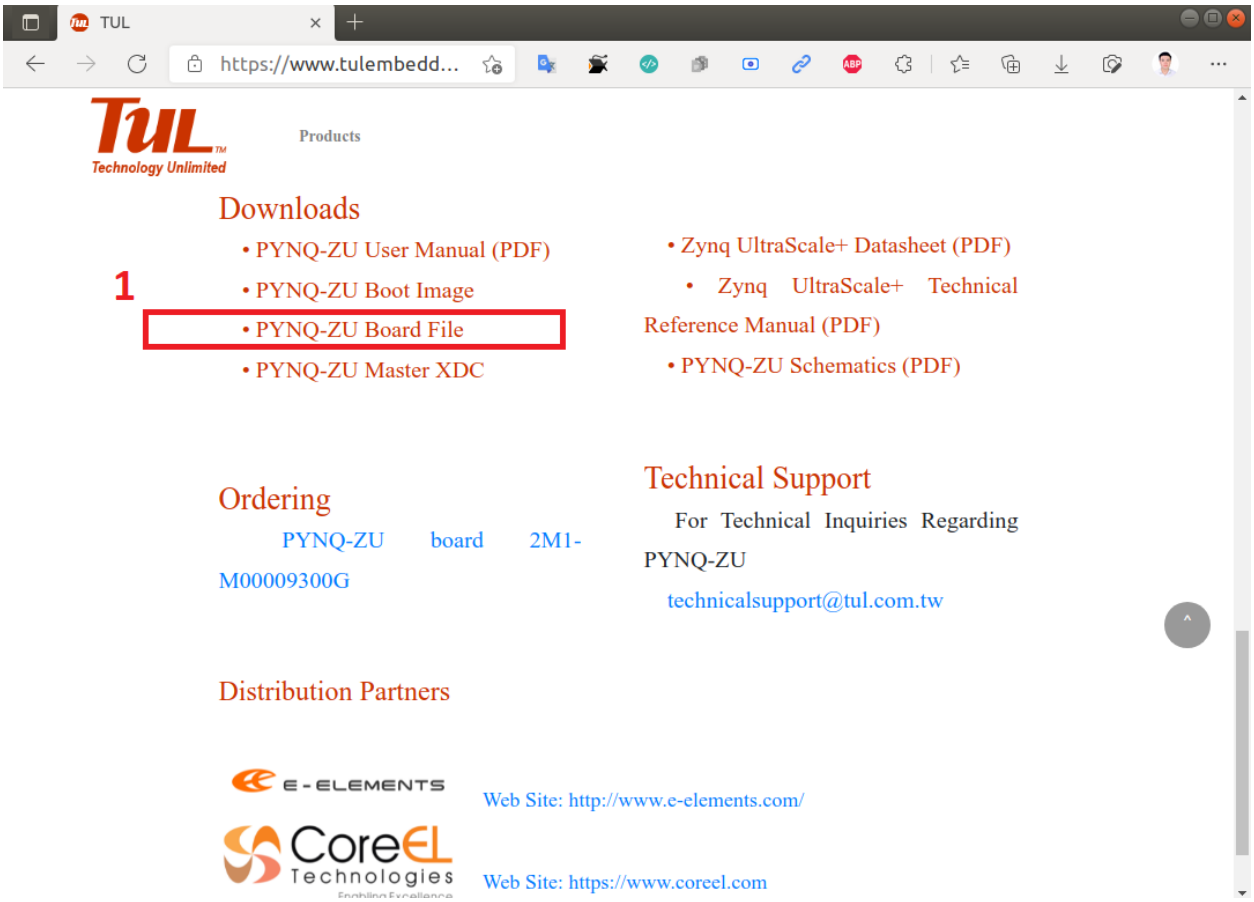
- After about 40 seconds, you should see the "DONE" LED turn on. This is the FPGA done signal, and indicates a bitstream has been downloaded.
- A few seconds later, the PYNQ image will flash the 4 white user LEDs (LED0-LED3), and the 2 RGB LEDs (LEDRGB0 and LEDRGB1) will flash blue to indicate the board is ready.

6.4. JupyterLab

After the board booting up, use should be able to access the JupyterLab at http://192.168.2.99. Default login password is **xilinx**.

6.5. System Design and Generate Bitstream using Vivado

Step 1: On the product's website, download the board file to be used in Vivado. At the time writing this tutorial, the version of the board file is v1.1. The downloaded file is **pynq_zu_v1.1.zip**. Save it at **$HOME/Downloads**.



Step 2: Extract the file, we will get these files:

```
├──── pynq_zu_v1.1: Extracted files
│       └──── pynqzu
│               └──── 1.1
│                       ├──── board.xml
│                       ├──── part0_pins.xml
│                       ├──── preset.xml
│                       ├──── pynqzu_board.png
│                       └──── xitem.json
└──── pynq_zu_v1.1.zip: Downloaded file
```

Step 3: Open the terminal to copy the board files to Vivado's board files location.

```
cd $HOME/Downloads
sudo cp -r pynq_zu_v1.1/pynqzu /tools/Xilinx/Vivado/2020.2/data/boards/board_files
```

Step 4: Create new directory as following.

- $HOME/hls_kernels/boards/pynq-zu/resize

```
hls_kernels
├──── boards
│       ├──── zcu104
│       │       ├──── resize
│       │       └──── rgb2yuv4
│       └──── pynq-zu
│               └────resize
├──── ip
│       ├──── resize
│       └──── rgb2yuv4
└──── Vitis_Libraries
        ├──── vision: Vitis Vision library
        └──── <other domains>
```

As mentioned, the exported RTL cores are reusable. We create only 1 new directory for the system design using Vivado.

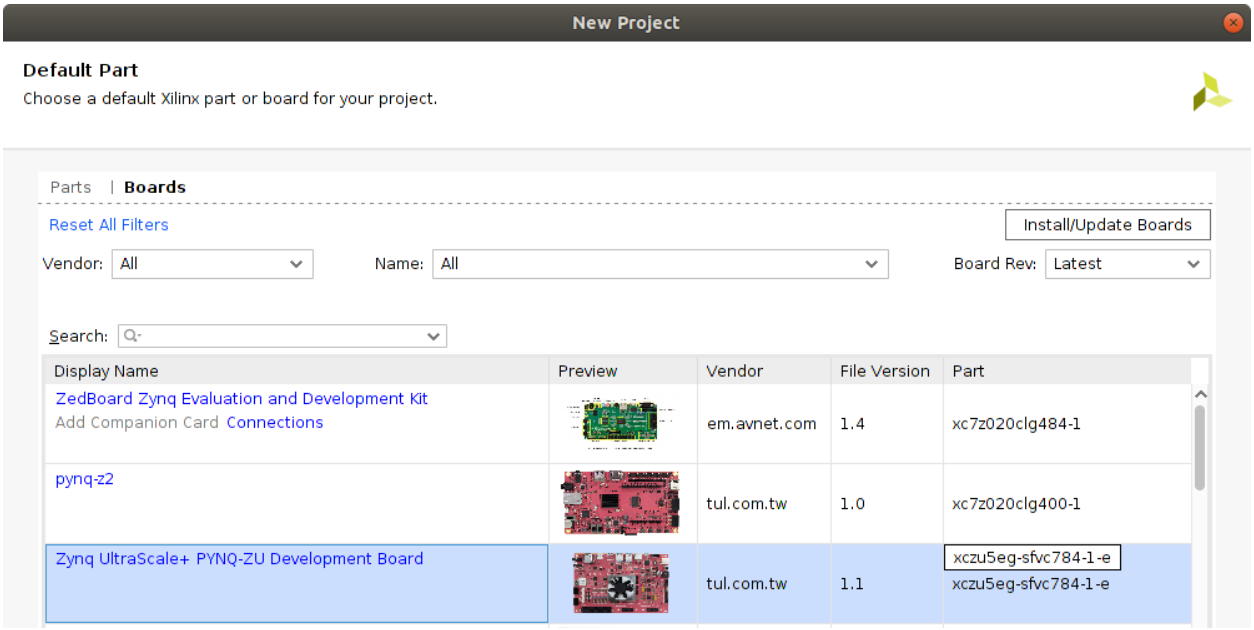
Step 5: Run these commands on terminal to open Vivado.

```
cd $HOME/hls_kernels/boards/pynq-zu/resize
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vivado
```

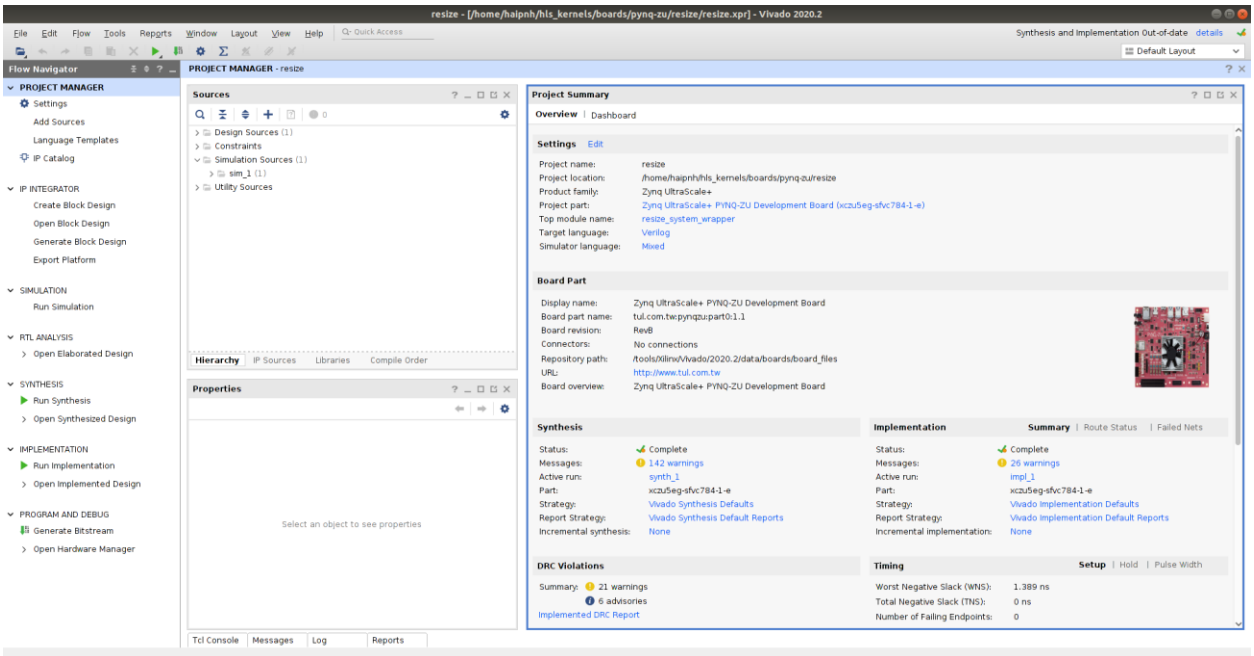
Step 6: Create new project with following information.

- Project name: **resize**
- Project location: in $HOME/hls_kernels/boards/pynq-zu/resize
- Uncheck **Create project subdirectory**
- Project Type: **RTL Project**
    - Check **Do not specify sources at this time**
    - Check **Project is an extensible Vitis platform**

Step 7: When choosing **Default Part**, PYNQ-ZU is available should be available as following.



Step 8: Finish the procedure to create new project, it should result as following.

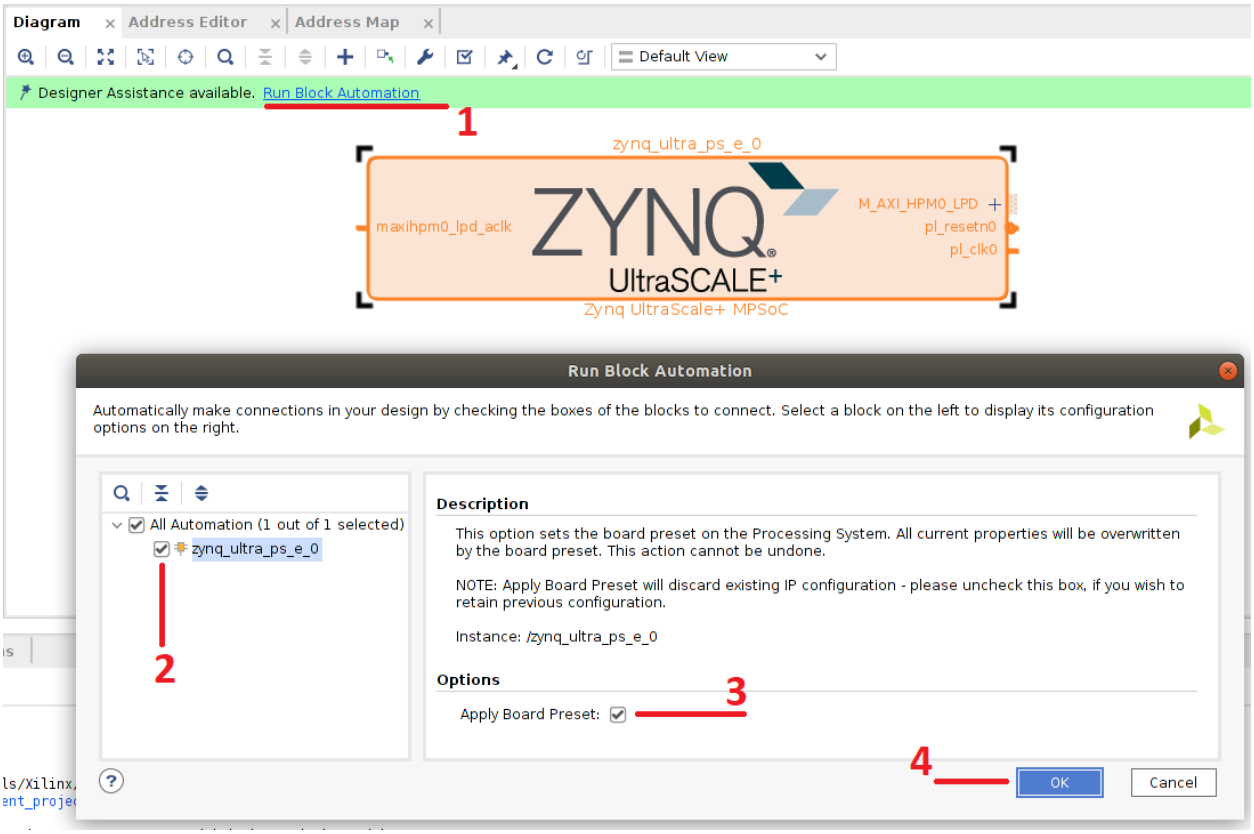


Step 12: Click **Tools ➔ Settings…** to open **Project Settings ➔ IP ➔ Repository**. Add following path: **$HOME/hls_kernels/ip/resize/resize/solution1**.

Step 13: The exported IP should be found: **resize_accel (xilinx.com:hls:resize_accel:1.0)**.

Step 14: Create a new system design, name it as **resize_system,** then open it.

Step 15: Add a **Zynq UltraScale+ MPSoC** block, instanced as **zynq_ultra_ps_e_0**. Then click **Run Block Automation** as following.

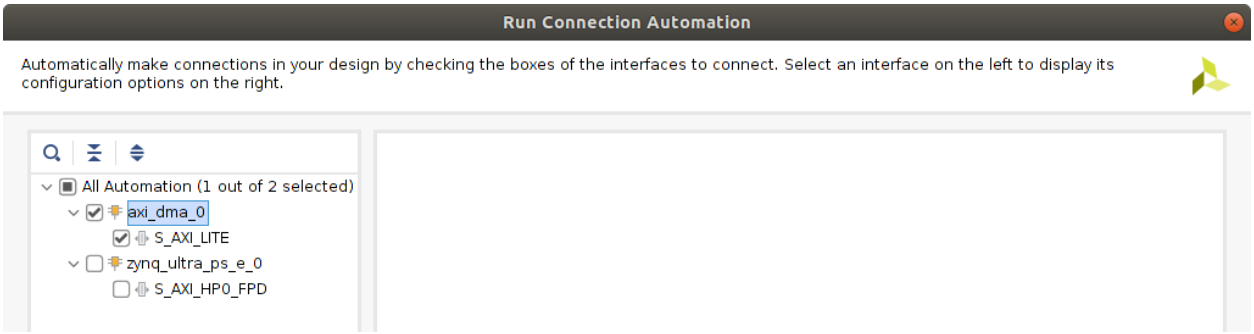


Step 17: Customize the **zynq_ultra_ps_e_0** as following: **PS-PL Configuration ➔ PS-PL Interfaces ➔ Slave Interface ➔ AXI HP ➔** Enable **AXI HP0 FPD ➔ Data Width: 128.**

Step 18: Add a **AXI Direct Memory Access**, instanced as **axi_dma_0**. Customize it with following configuration.

- Disable **Enable Scatter Gather Engine**
- Width of Buffer Length Register: 26 bits
    - Address Width: 32 bits
- Enable Read Channel
    - Max Burst Size: 256
- Enable Write Channel
    - Max Burst Size: 256

Step 13: Click **Run Connection Automation** with following configuration.



Step 14: Add a **resize_accel** block, instanced as **resize_accel_0**.

Step 15: Add a **AXI4-Stream Data Width Converter**, instanced as **axis_dwidth_converter_0**. Then customize it with following configuration.

- Slave Interface TDATA Width (bytes): 4
- Master Interface TDATA Width (bytes): 3

Step 16: Add a **AXI4-Stream Data Width Converter**, instanced as **axis_dwidth_converter_1**. Then customize it with following configuration.

- Slave Interface TDATA Width (bytes): 3
- Master Interface TDATA Width (bytes): 4

Step 17: Add a **AXI Interconnect**, instanced as **axi_interconnect_0**. Then customize it with following configuration.

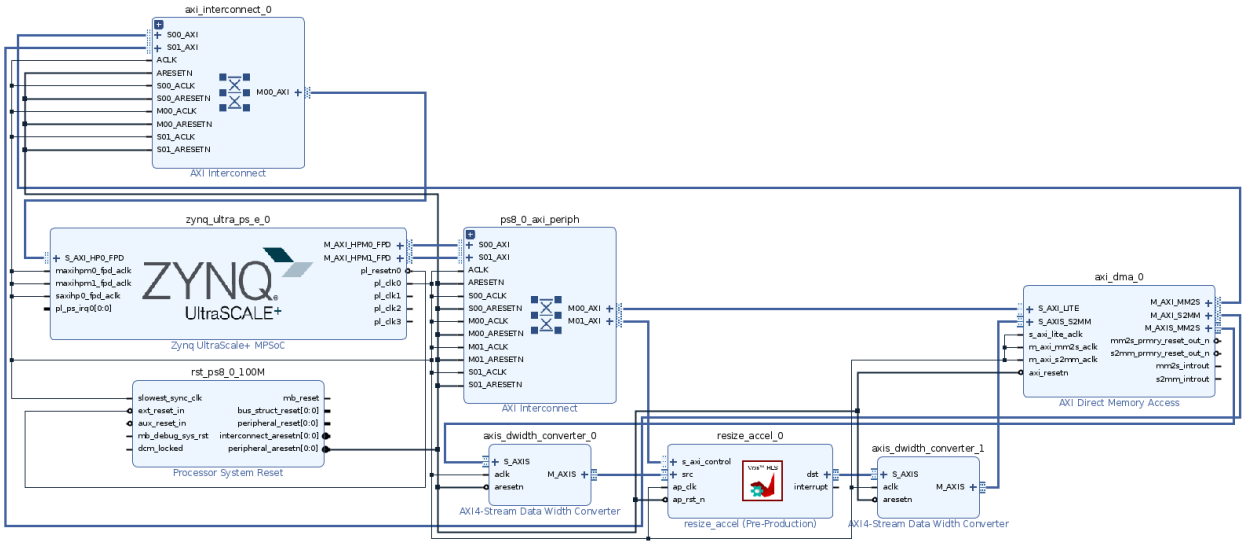
- Number of Slave Interfaces: 2
- Number of Master Interfaces: 1
- Interconnect Optimization Strategy: Maximize Performance

Step 18: Wire the interfaces of these blocks as following.

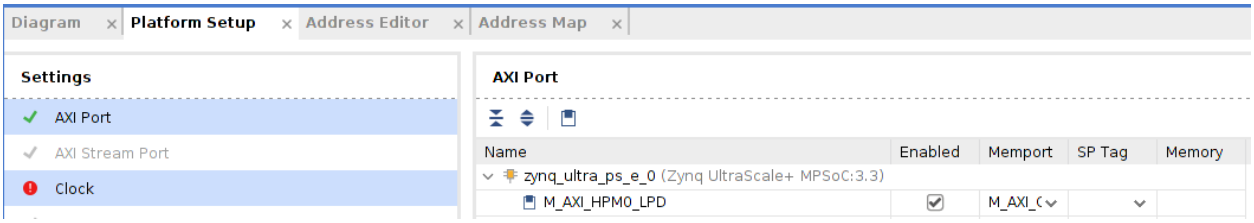
- zynq_ultra_ps_e_0 / S_AXI_HP0_FPD <-> axi_interconnect_0 / M00_AXI
- axi_interconnect_0 / S00_AXI <-> axi_dma_0 / M_AXI_MM2S
- axi_interconnect_0 / S01_AXI <-> axi_dma_0 / M_AXI_S2MM
- axi_dma_0 / M_AXIS_MM2S <-> axis_dwidth_converter_0 / S_AXIS
- axi_dma_0 / S_AXIS_S2MM <-> axis_dwidth_converter_1 / M_AXIS
- axis_dwidth_converter_0 / M_AXIS <-> resize_accel_0 / src
- axis_dwidth_converter_1 / S_AXIS <-> resize_accel_0 / dst

Step 19: Click **Run Connection Automation**, select **All Automation**, then click **OK**.

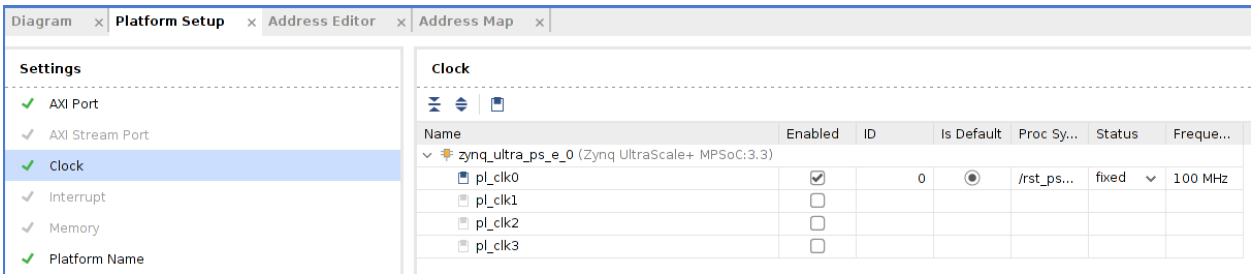
Step 20: After fully wiring the system, and generating the layout, it should result as below.



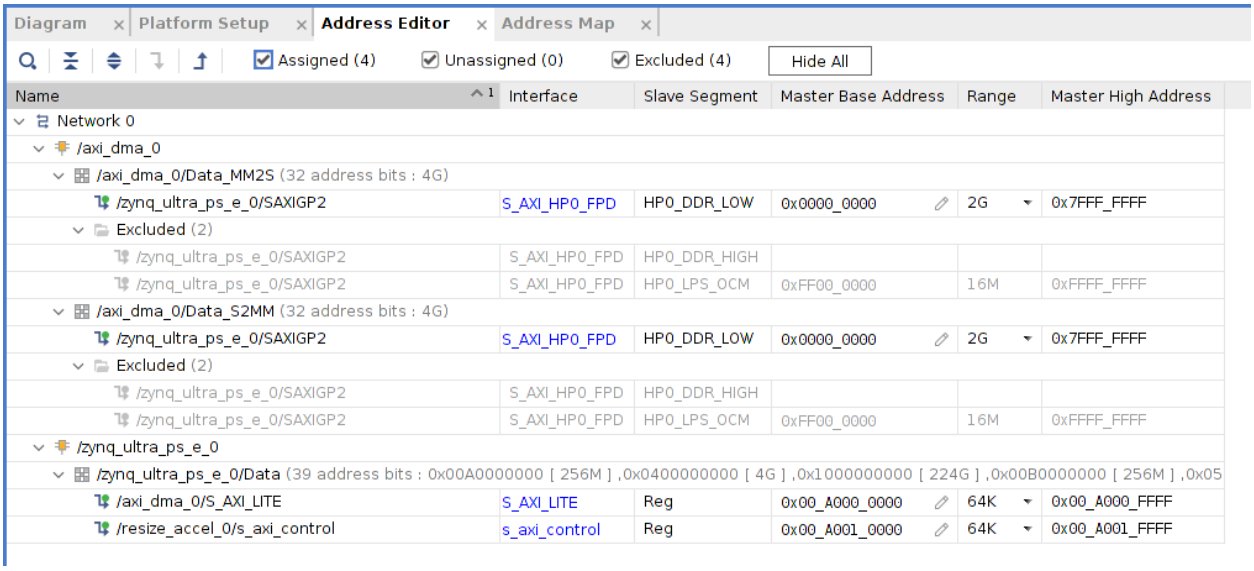
Step 21: Open **Platform Setup ➔ AXI Port ➔** Enable **M_AXI_HPM0_LPD ➔** Select Memport **M_AXI_GP**.



Step 22: On **Platform Setup**, open **Clock ➔** Enable **pl_clk0 ➔** Select **Is Default**.



Step 23: Go to **Address Editor**, exclude unused **Slave Segments**, the result should be as following.



Step 24: Validate the design.

Step 25: Create HDL wrapper.

Step 26: Generate the bitstream.

After generating the bitstream, the design is ready.

6.6. Run Jupyter Notebook example

Step 1: Open Jupyter home page, create a new folder, rename it as **resize**.

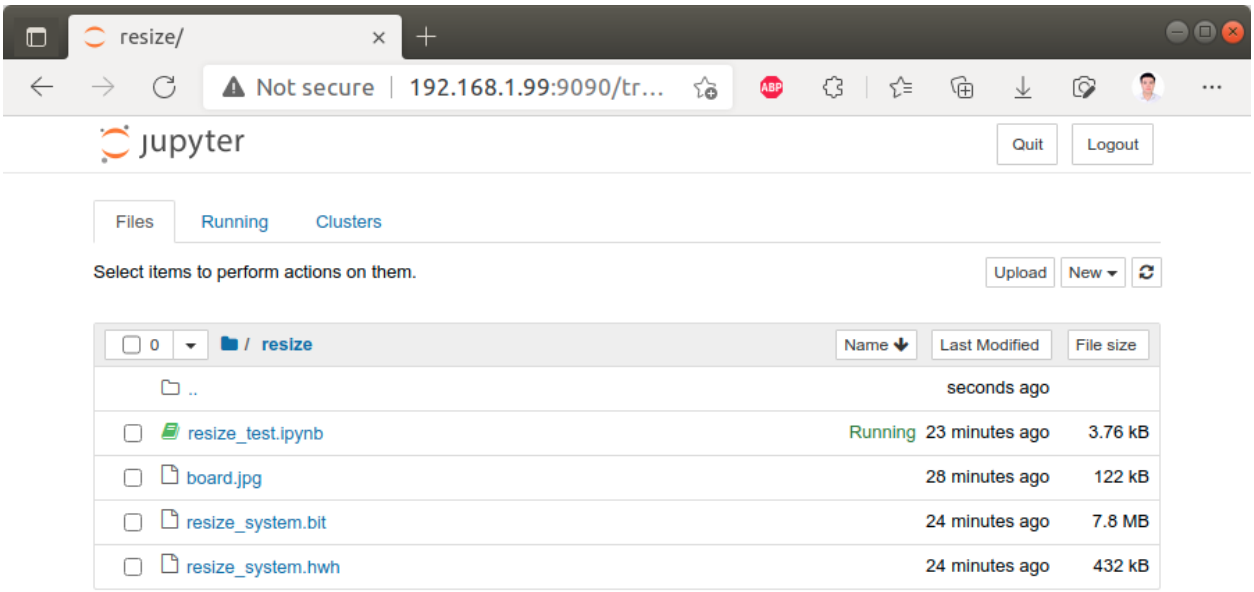
Step 2: Upload these files from **$HOME/hls_kernels/notebooks/resize**:

- **resize_test.ipynb**
- **board.jpg**

Step 3: Upload these files from **$HOME/hls_kernels/boards/pynq-zu/resize**:

- resize.gen/sources_1/bd/resize_system/hw_handoff/**resize_system.hwh**
- resize.runs/impl_1/**resize_system_wrapper.bit** ➔ rename to **resize_system.bit**

Step 4: Review uploaded items.



Step 5: Examine the **resize_test.ipynb**. Run it. The result should be as following.

In [11]:
```
run_kernel()
resized_image = Image.fromarray(out_buffer)
```

In [12]:
```
print("Image size: {}x{} pixels.".format(new_width, new_height))
plt.figure(figsize=(12, 10));
_ = plt.imshow(resized_image)
```

```
Image size: 320x240 pixels.
```

In [13]:
```
del in_buffer
del out_buffer
```

Reference

1. PYNQ - Python productivity for Zynq
2. Vitis High-Level Synthesis User Guide (UG1399)
3. Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit
4. ZCU104 Setup Guide — Python productivity for Zynq (Pynq)
5. TUL Embedded PYNQ ™ -ZU Board
6. Getting started with your PYNQ-ZU
7. GitHub - Xilinx/PYNQ-HelloWorld: This repository contains a "Hello World" introduction application to the Xilinx PYNQ framework.