**Project Overview**

This project implements a real-time coincidence trigger acquisition system for **four 1 GSPS ADC channels** (1 ns per sample). A programmable **threshold** is applied to each channel. A trigger event is generated only when **all four channels exceed the threshold within a 100 ns coincidence window**.

Once triggered, the system captures and stores the corresponding waveforms, preserving **100 ns of pre-trigger data and 100 ns of post-trigger data** (approximately a 200 ns total capture window) for further analysis.

By co-designing with **Vitis HLS** and **Vivado Block Design**, the system can be implemented **without writing any Verilog**. Only **C++ code** (to generate the HLS IP) and **Vivado block connections/configuration** are required to complete the design.

**Vitis HLS trigger logic IP generation (step-by-step)**

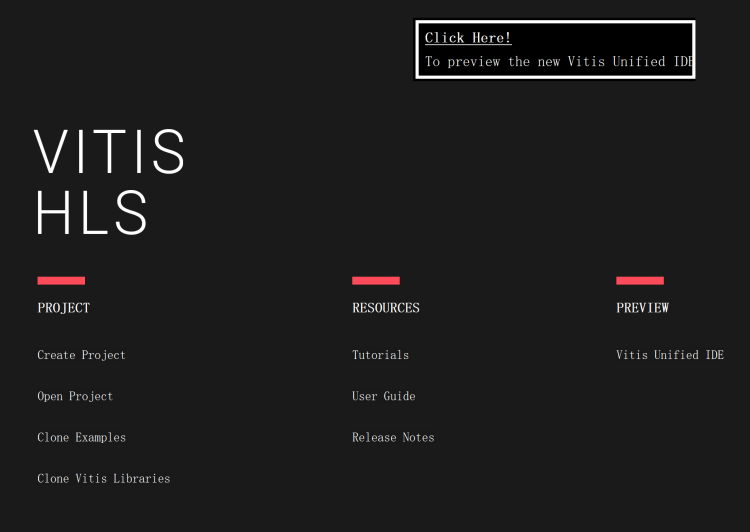
“trigger\_capture\_hls.cpp”: C++ source code used to generate the trigger logic. If it is necessary to change from a single-threshold trigger to a double-threshold trigger in the future, only this code needs to be modified, which is straightforward.

“tb\_trigger\_capture.cpp”: Test bench for functional verification.

1. Create a new HLS project

* Open **Vitis HLS**

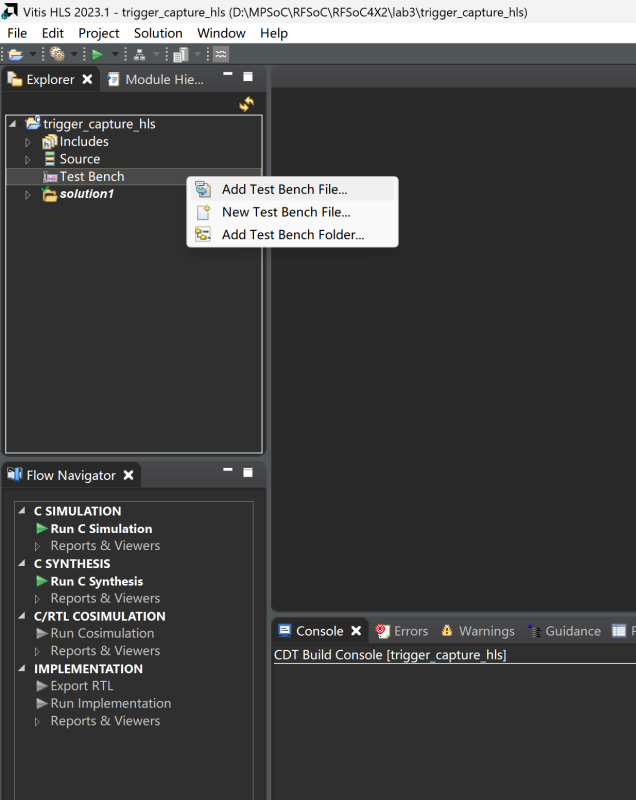
1. New Project → choose a project name/location



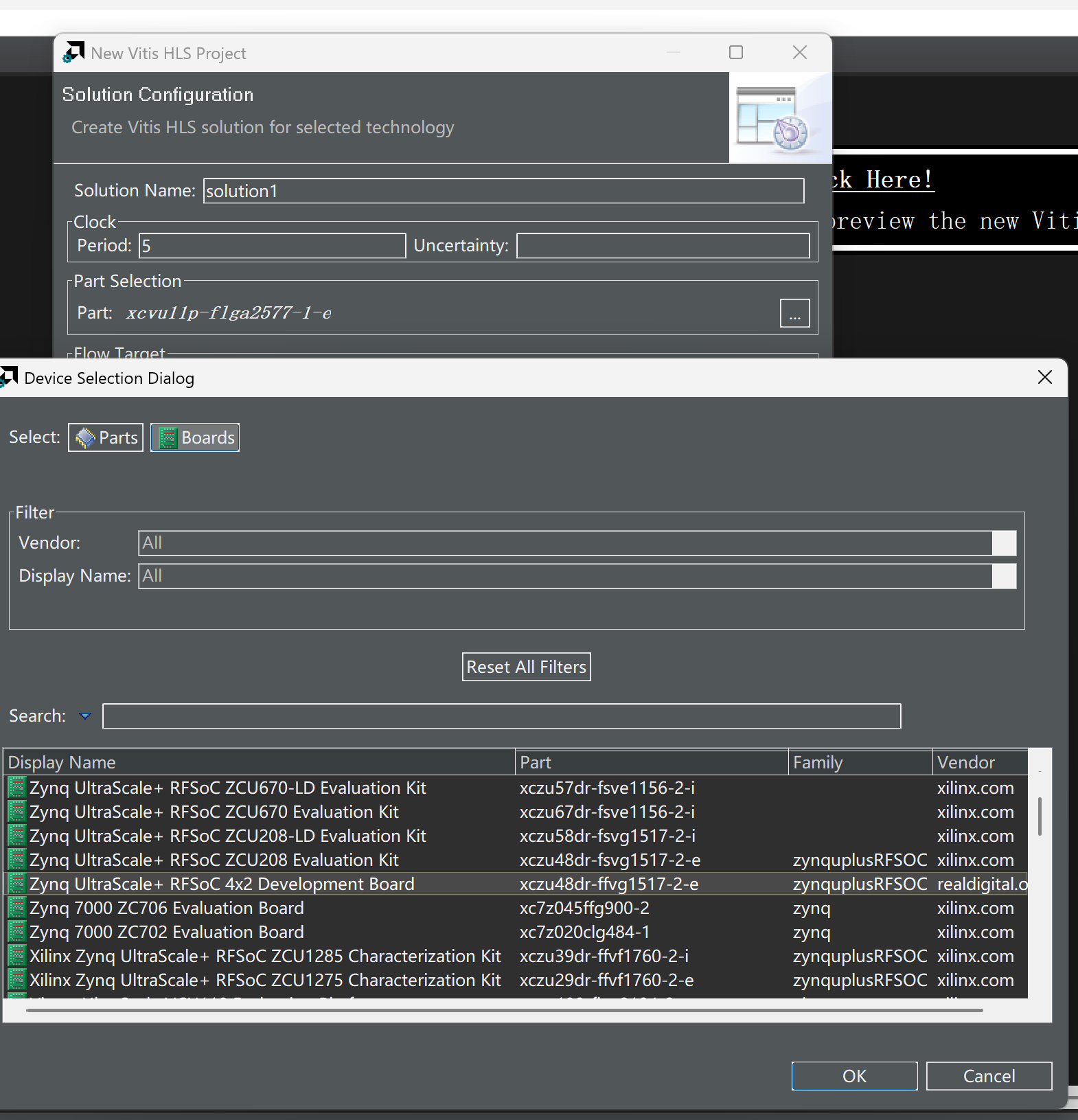
1. Add your top C++ file (e.g., trigger\_capture\_hls.cpp) and (optional) testbench (e.g., tb\_trigger\_capture.cpp)
2. Set the top function. In the project settings, set Top Function to your top-level function name (e.g., trigger\_capture\_hls)



You can add Test Bench later, sometime it will false if you at it in project creation (Strange Bug of Vitis):



1. Configure the target device/clock. Select the same FPGA part/board as your Vivado design (MPSoC/RFSoC device). Set the clock period to match your intended PL clock (e.g., 4 ns for 250 MHz, 8 ns for 125 MHz)

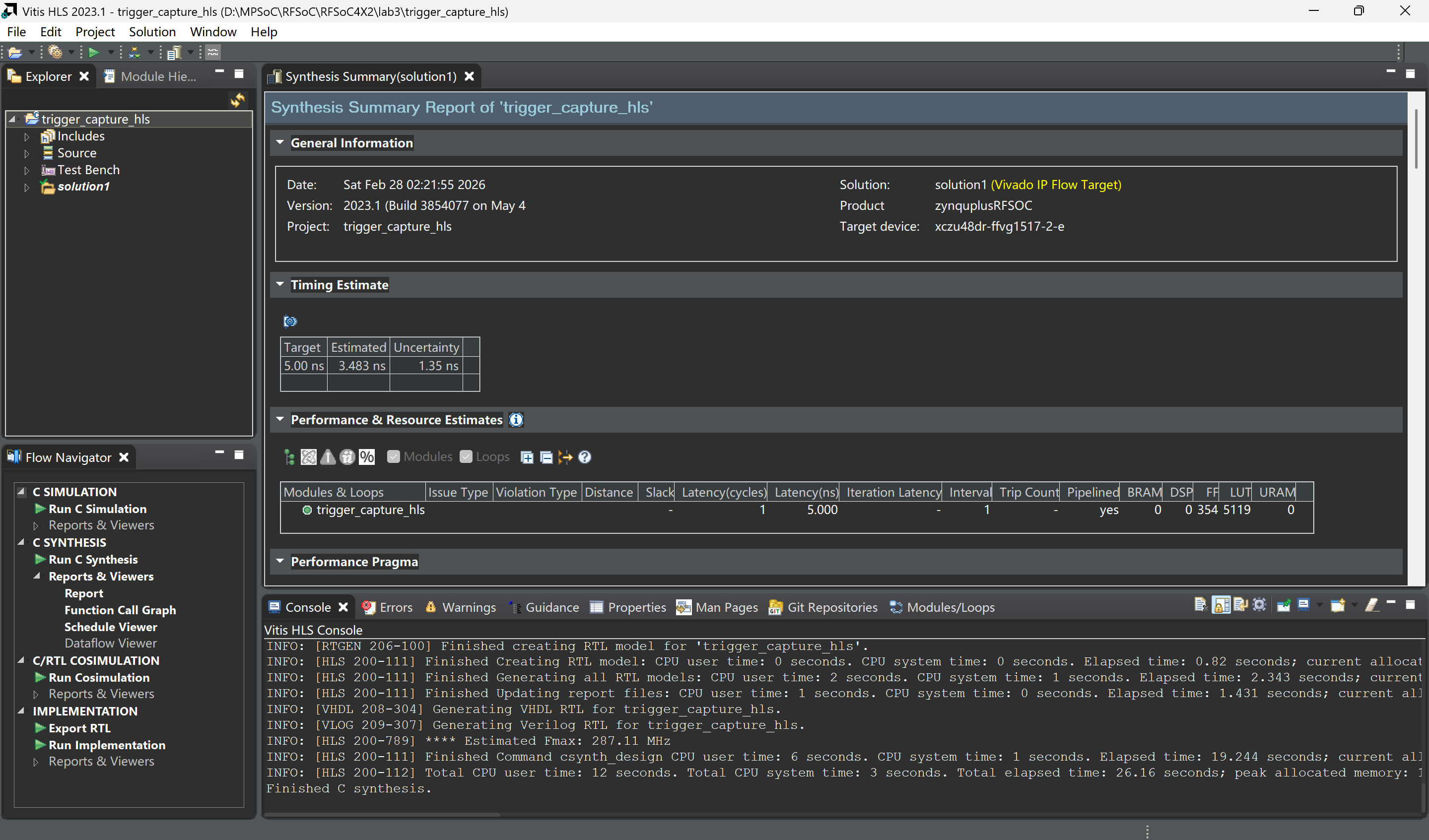


1. Run C Simulation (optional but recommended). Run C Simulation to verify the trigger behavior and waveform window logic in software.



Result

1. Review interfaces. Confirm your pragmas/interfaces generate: AXI4-Stream ports for data in/out (and per-channel ports if you have 4 inputs/4 outputs); AXI-Lite control bundle for registers (thresholds, window length, clear/status, etc.); Control protocol as required (ap\_ctrl\_hs)
2. Run C-Synthesis. Check: **Latency/II** (goal: **II=1** for streaming); Interfaces inferred correctly (AXIS + AXI-Lite); No unexpected stalls



Result

1. Run RTL Co-simulation (recommended), Some bug in Vitis.

If error happen:

Nano “D:\MPSoC\RFSoC\RFSoC4X2\lab1\trigger\_capture\_hls\solution1\sim\verilog\xsim.dir\trigger\_capture\_hls\xsim\_script.tcl”  
Then insert:

“# --- ensure tv junction exists (Windows) ---

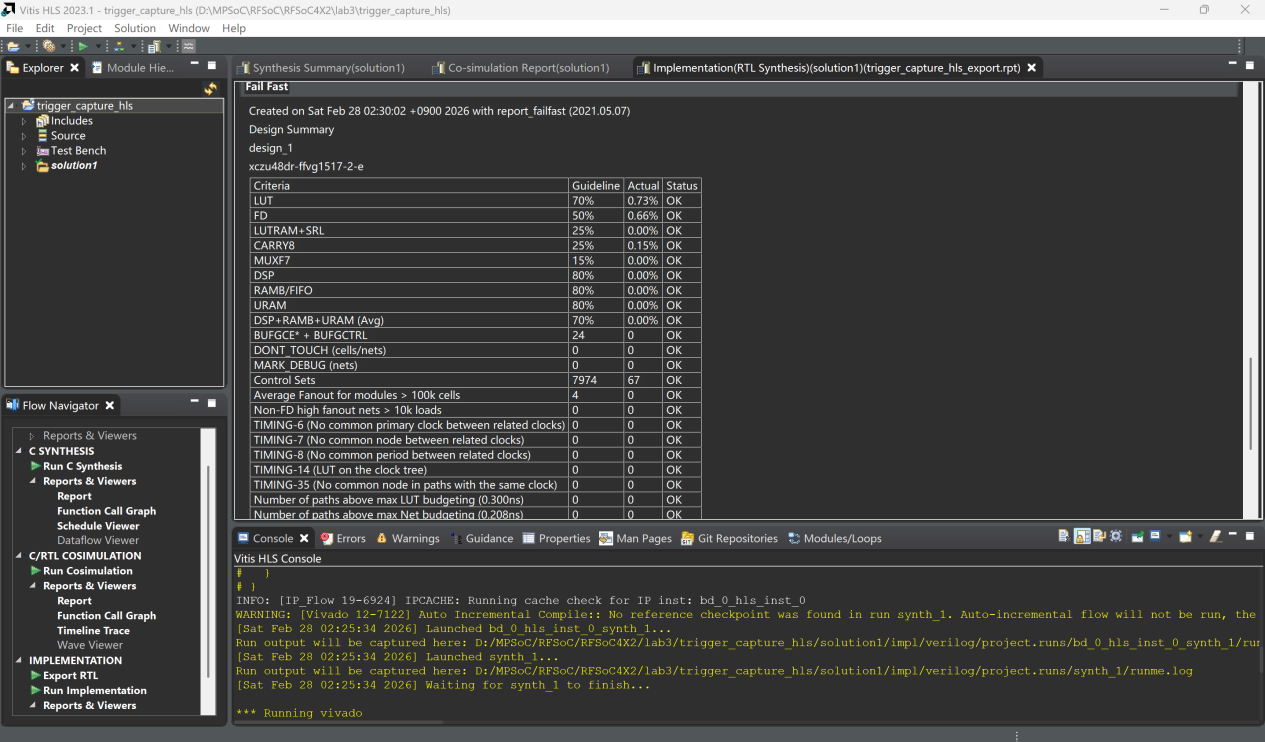
catch {exec cmd /c "cd /d .. & if not exist xsim.dir mkdir xsim.dir & rmdir /s /q xsim.dir\\tv 2>nul & mklink /J xsim.dir\\tv ..\\tv >nul"} msg

# puts $msg”



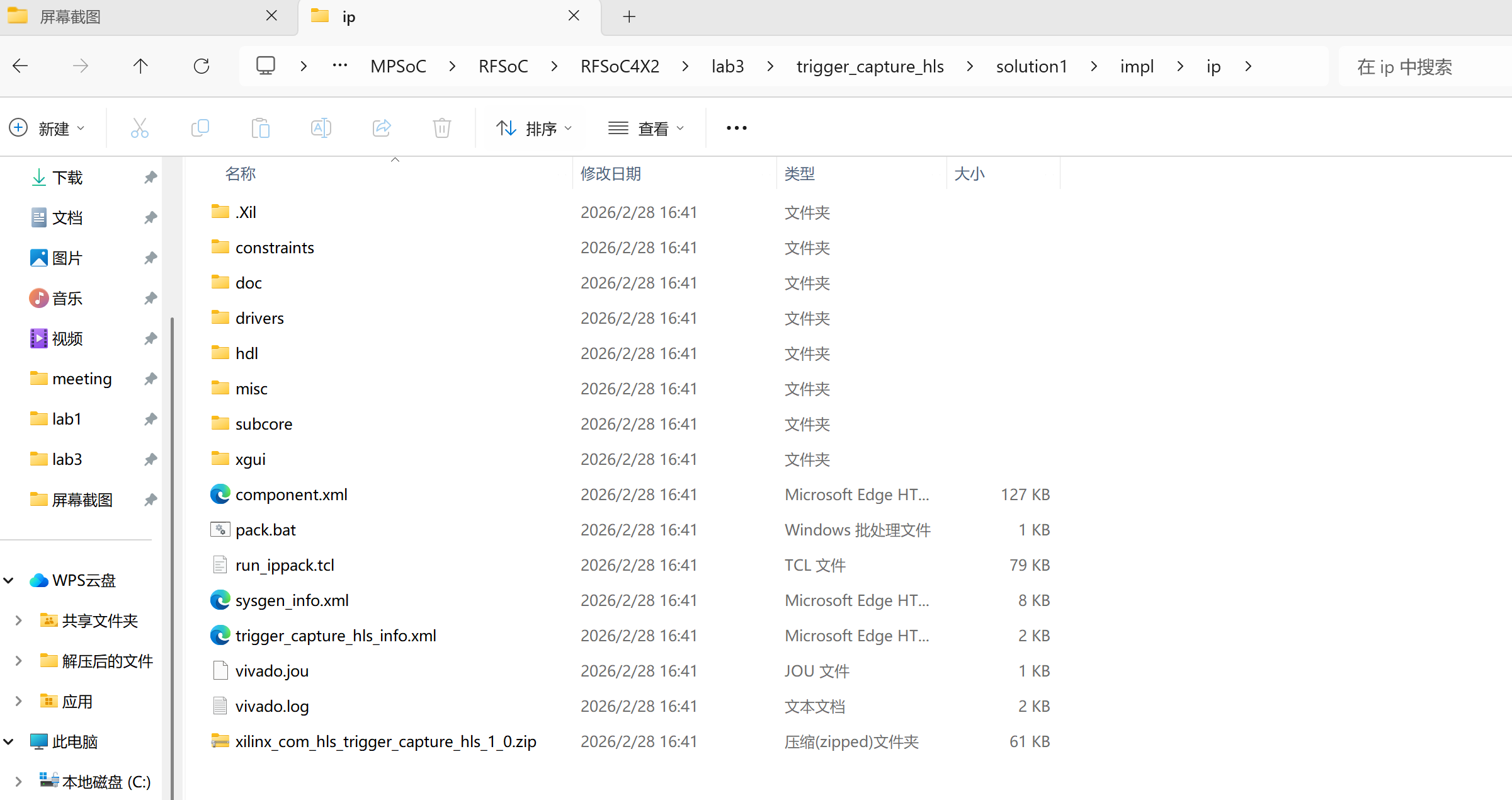
Result

1. Run IMPLEMENTATION, See that only <1% of resource used for trigger function.



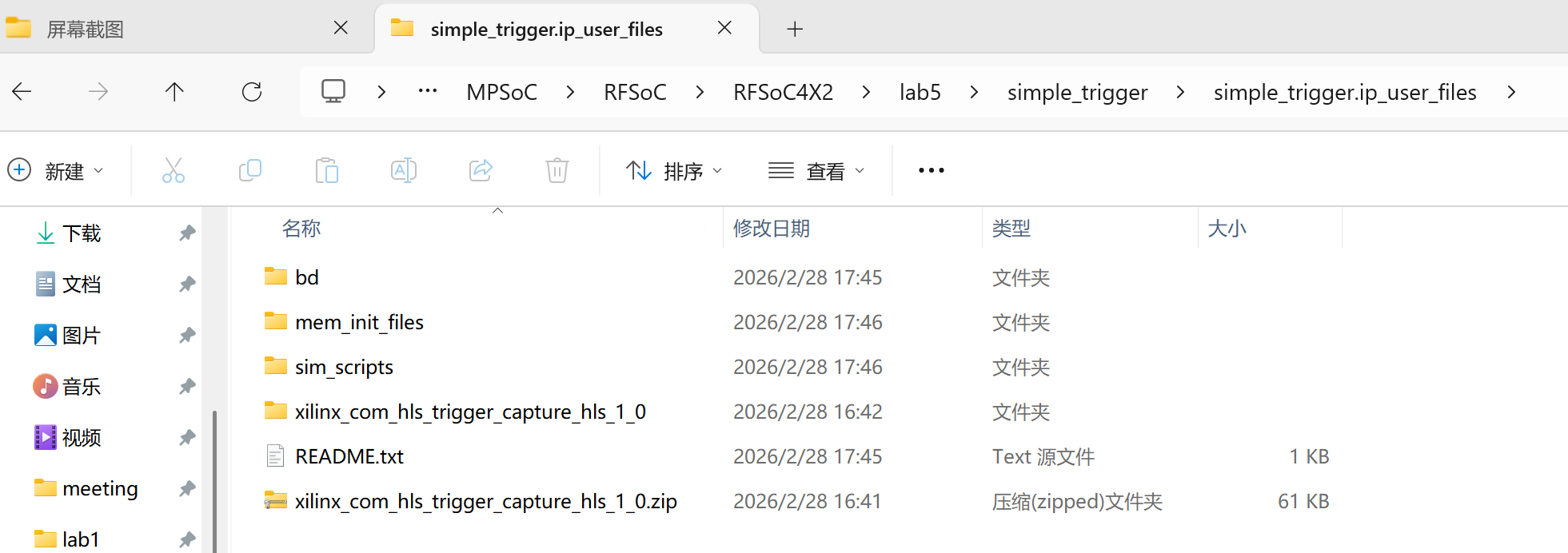
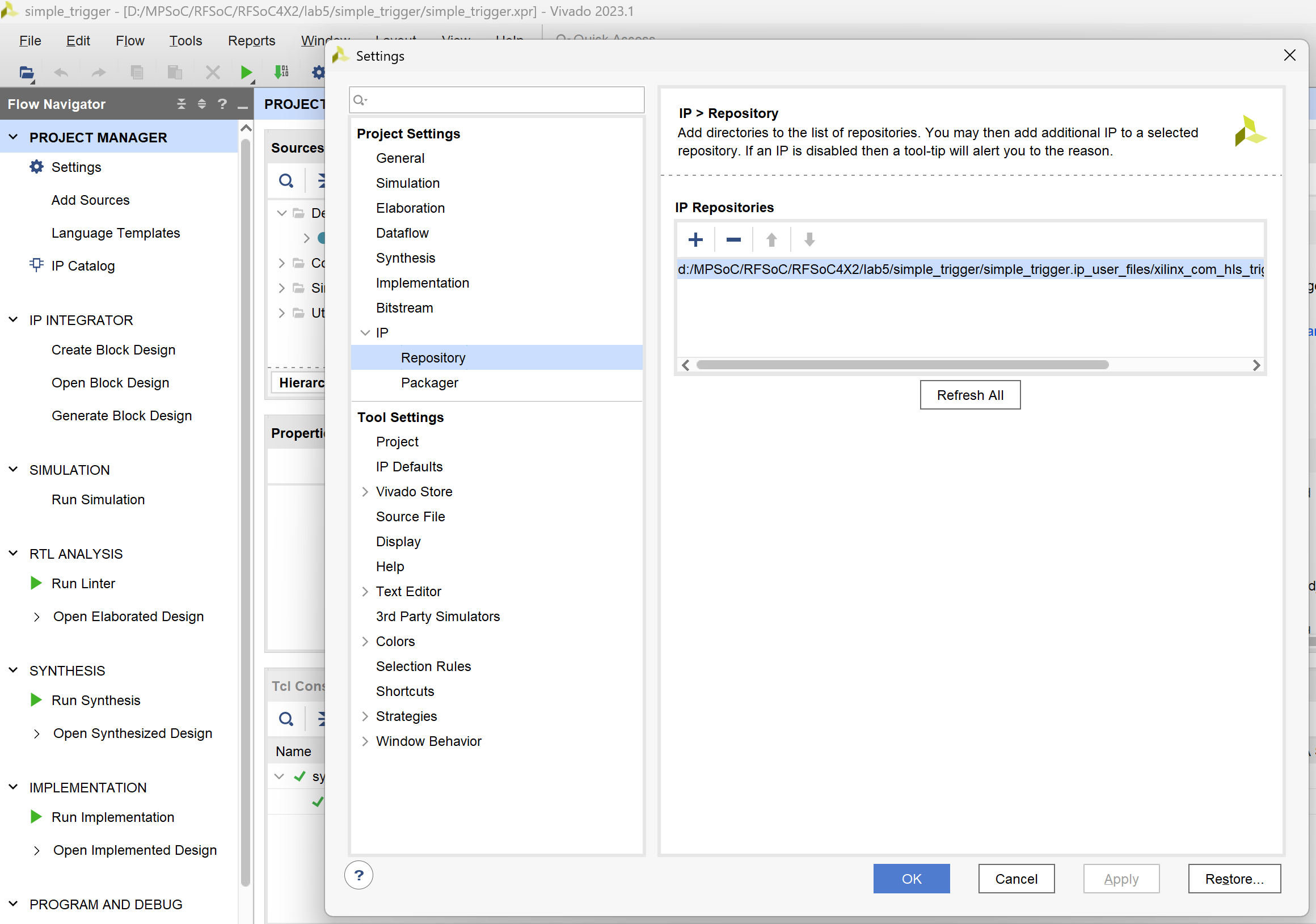
Result

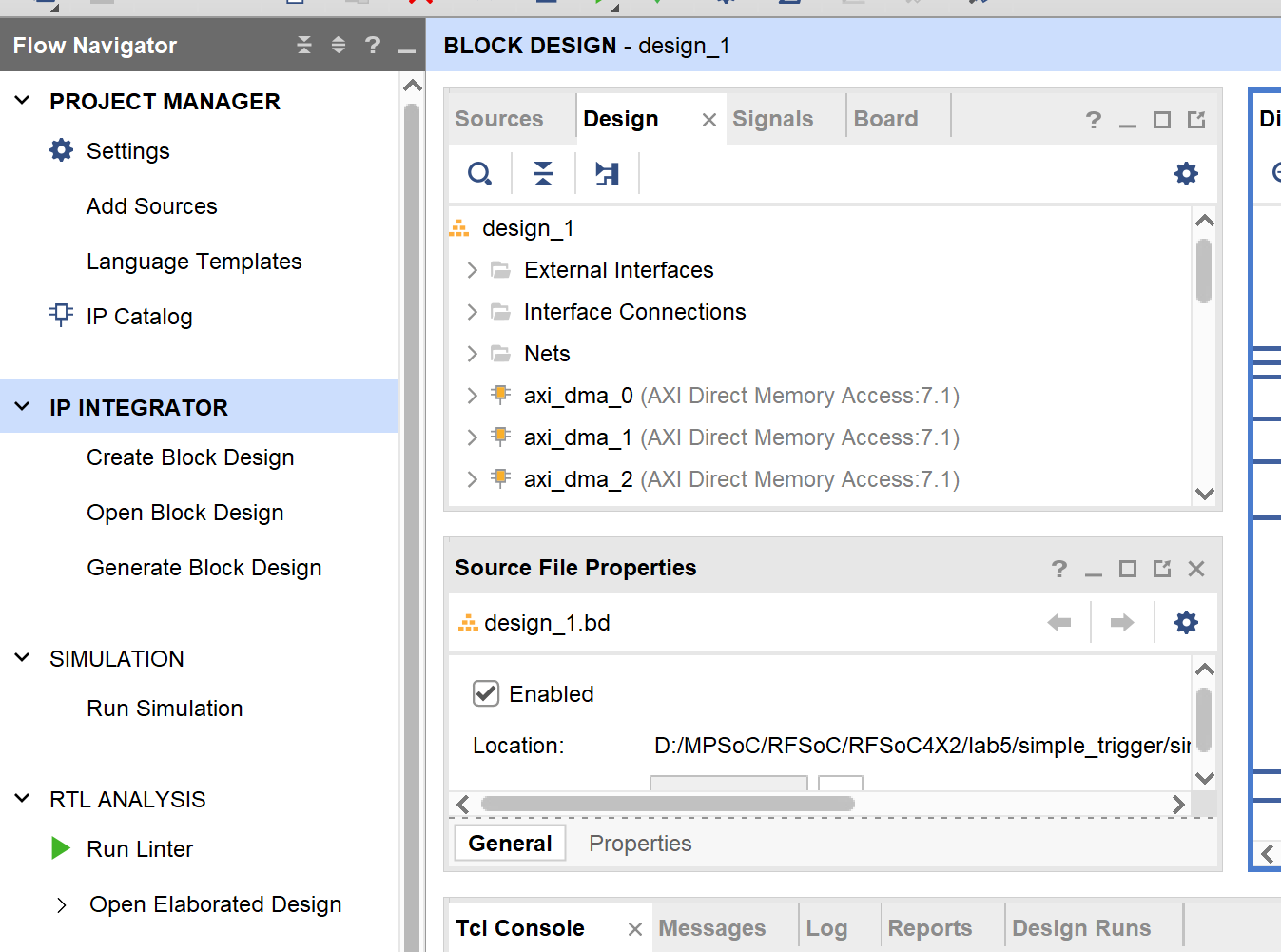
1. Export the IP, So that it can be used by Vivado as a block diagram. See the IP in:

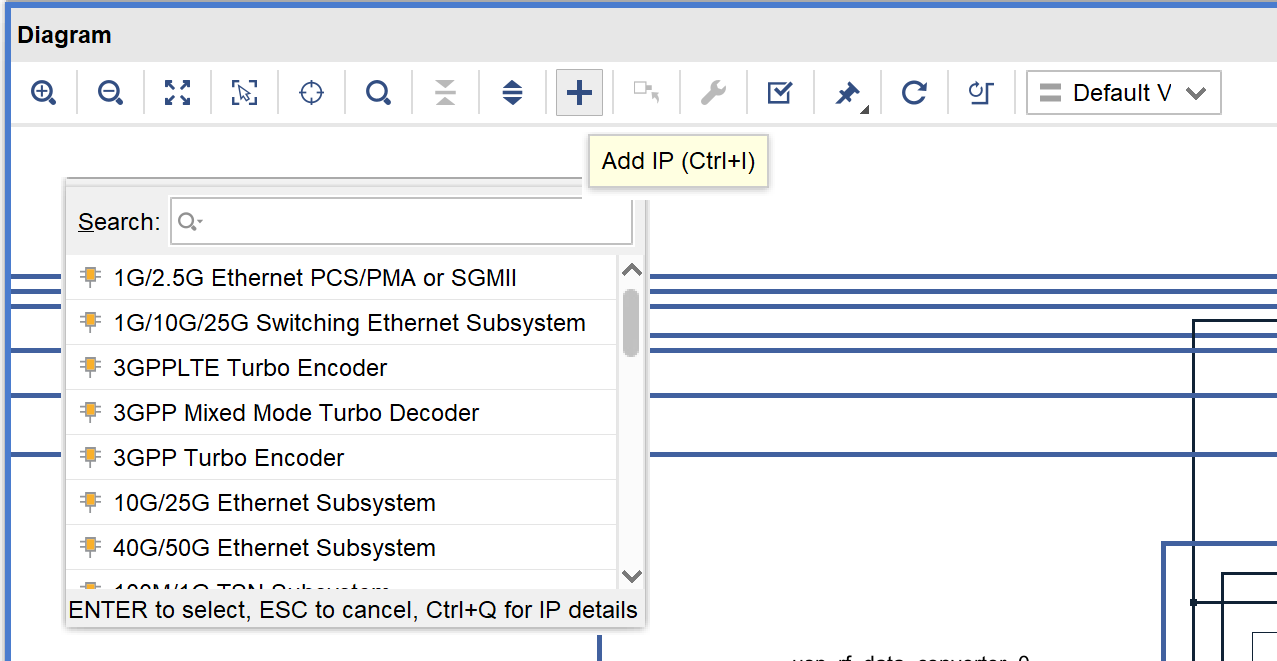


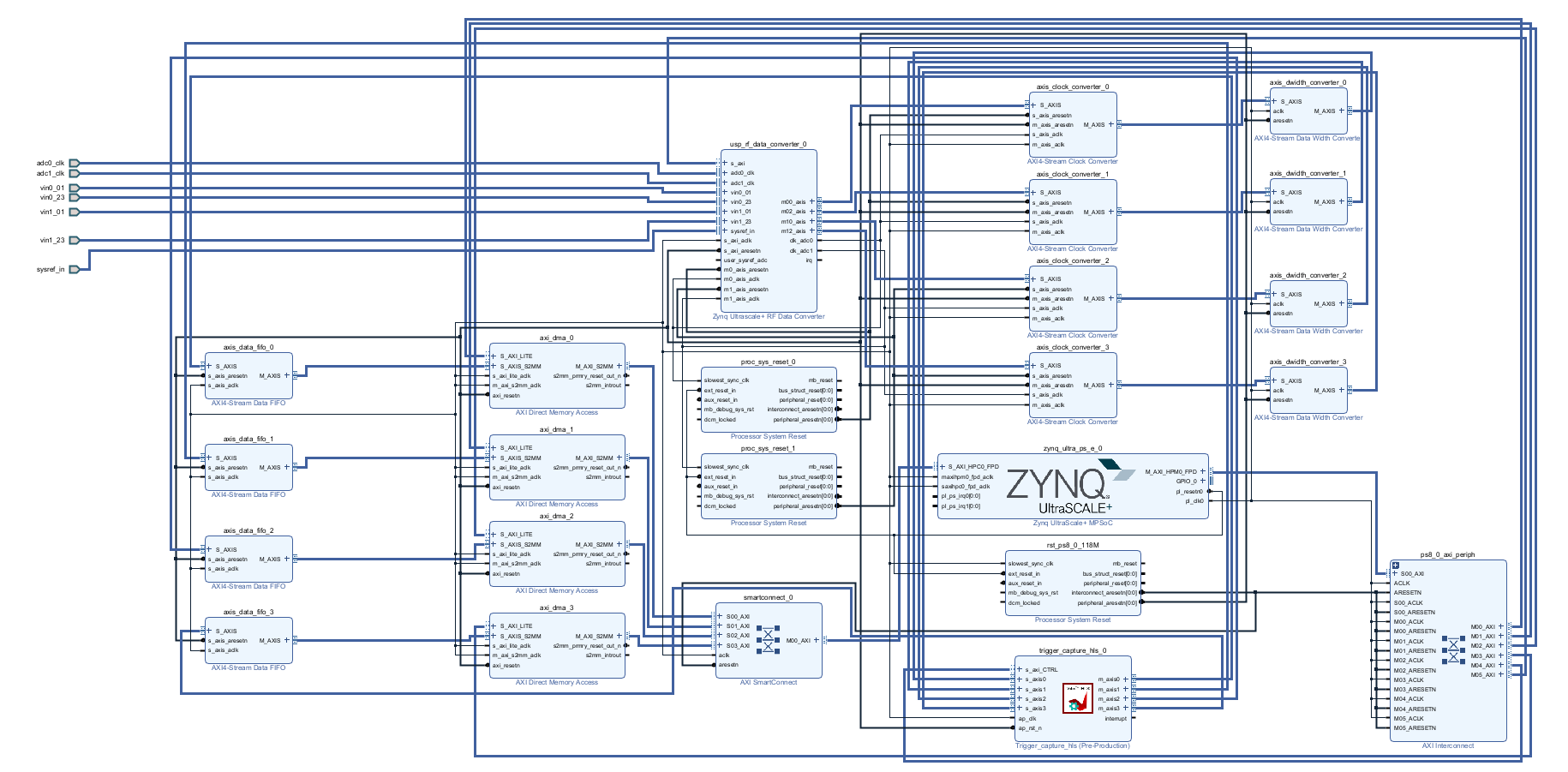
Now we have made our trigger logic, and then we are going to use it in Vivado.

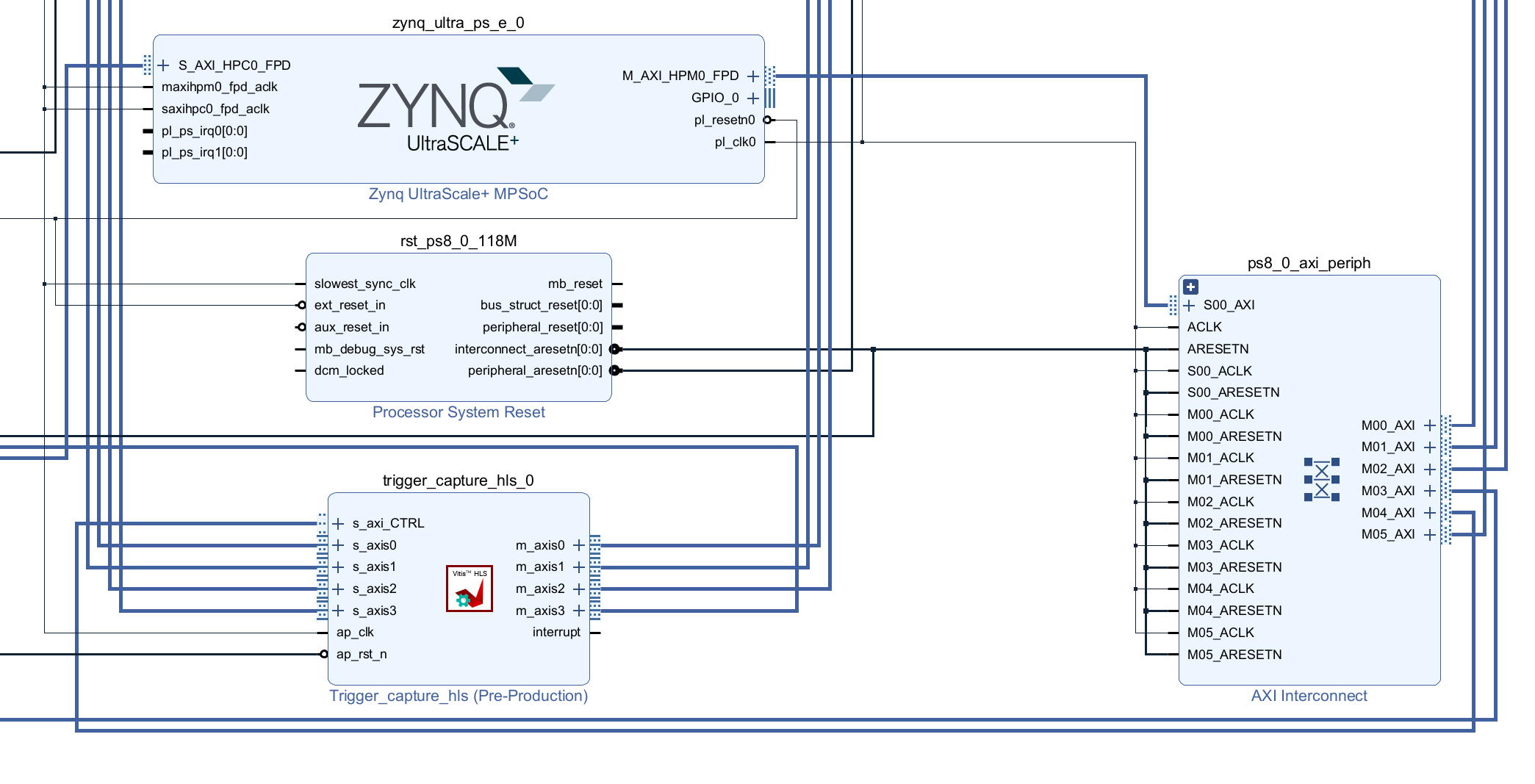
**Vivado block building (step-by-step)**

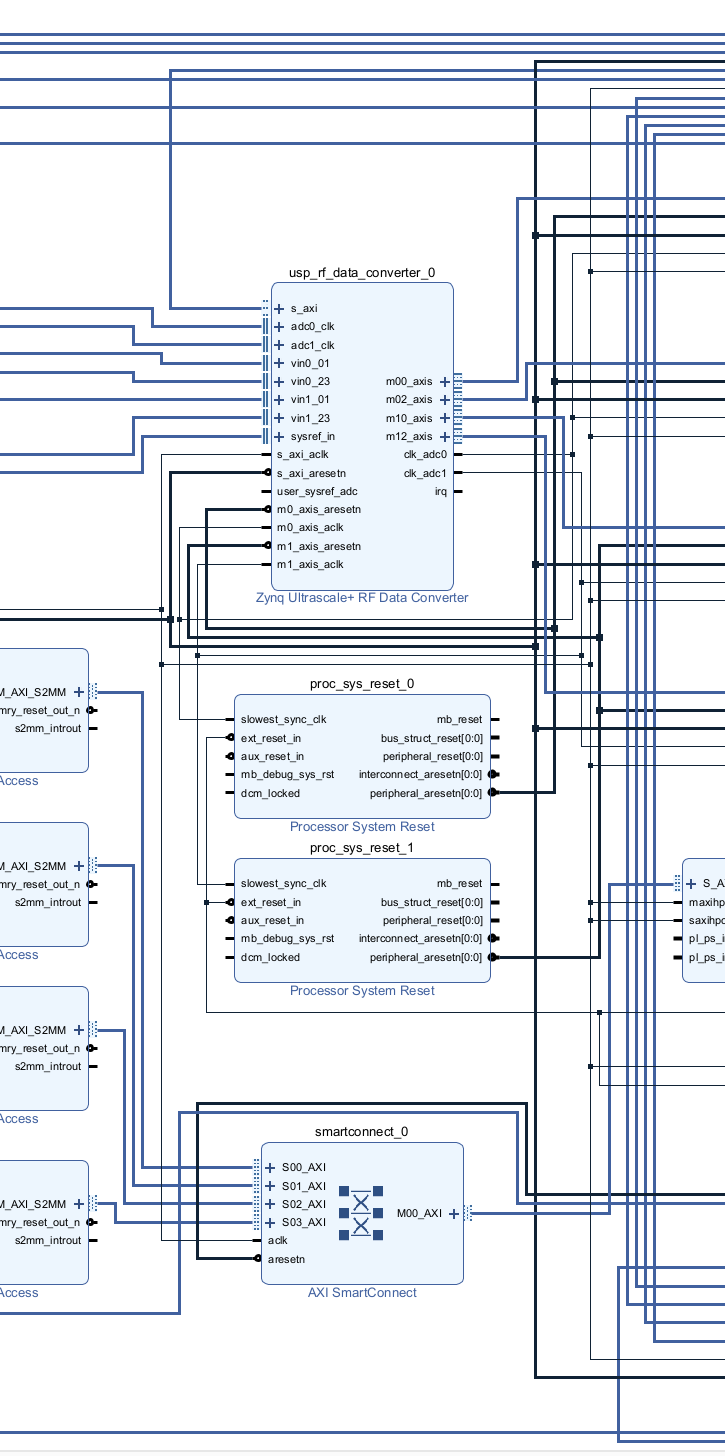
1. Create a project, choose RFSoC 4X2 board
2. Exact trigger\_hls IP generated by Vitis HLS to here:  
   
3. Import the IP:  
   
4. Create Block Design

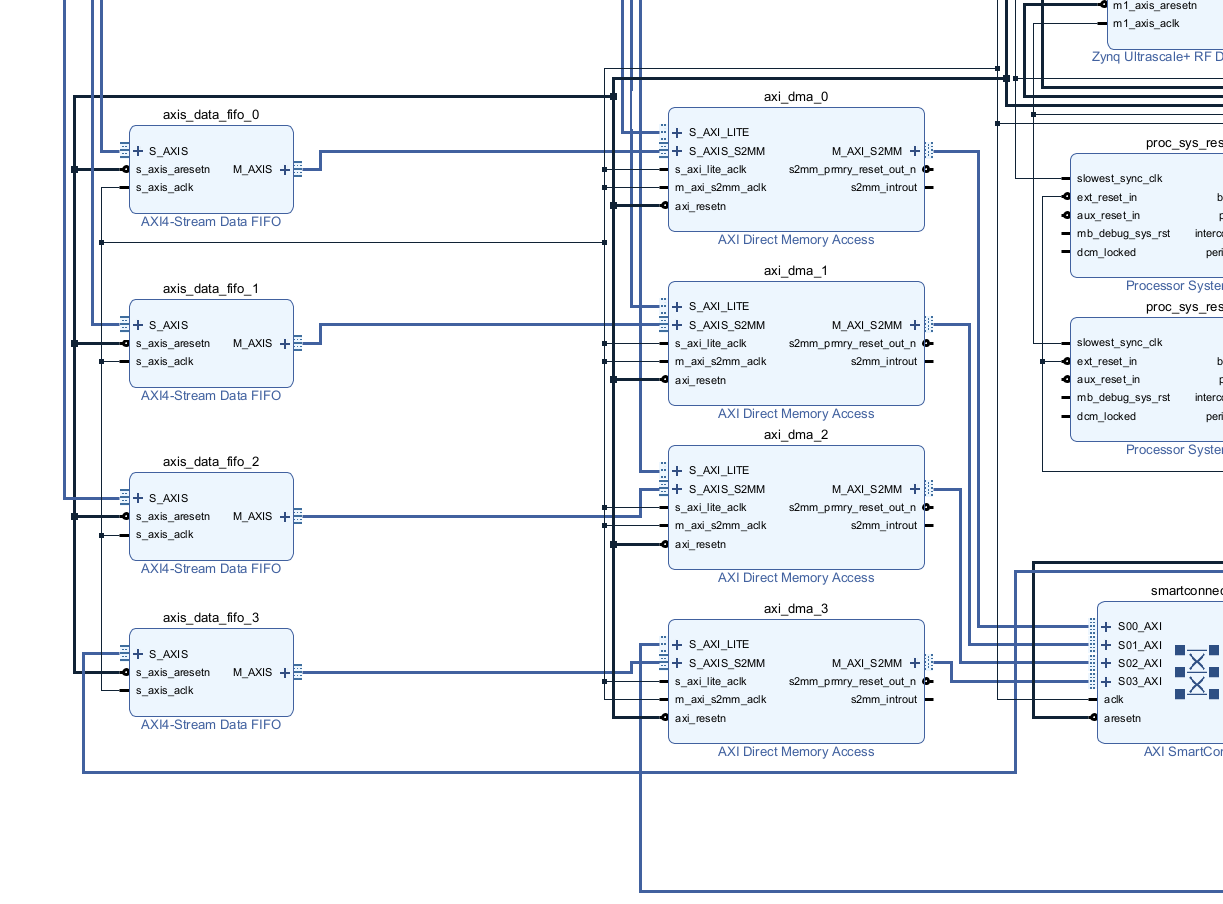


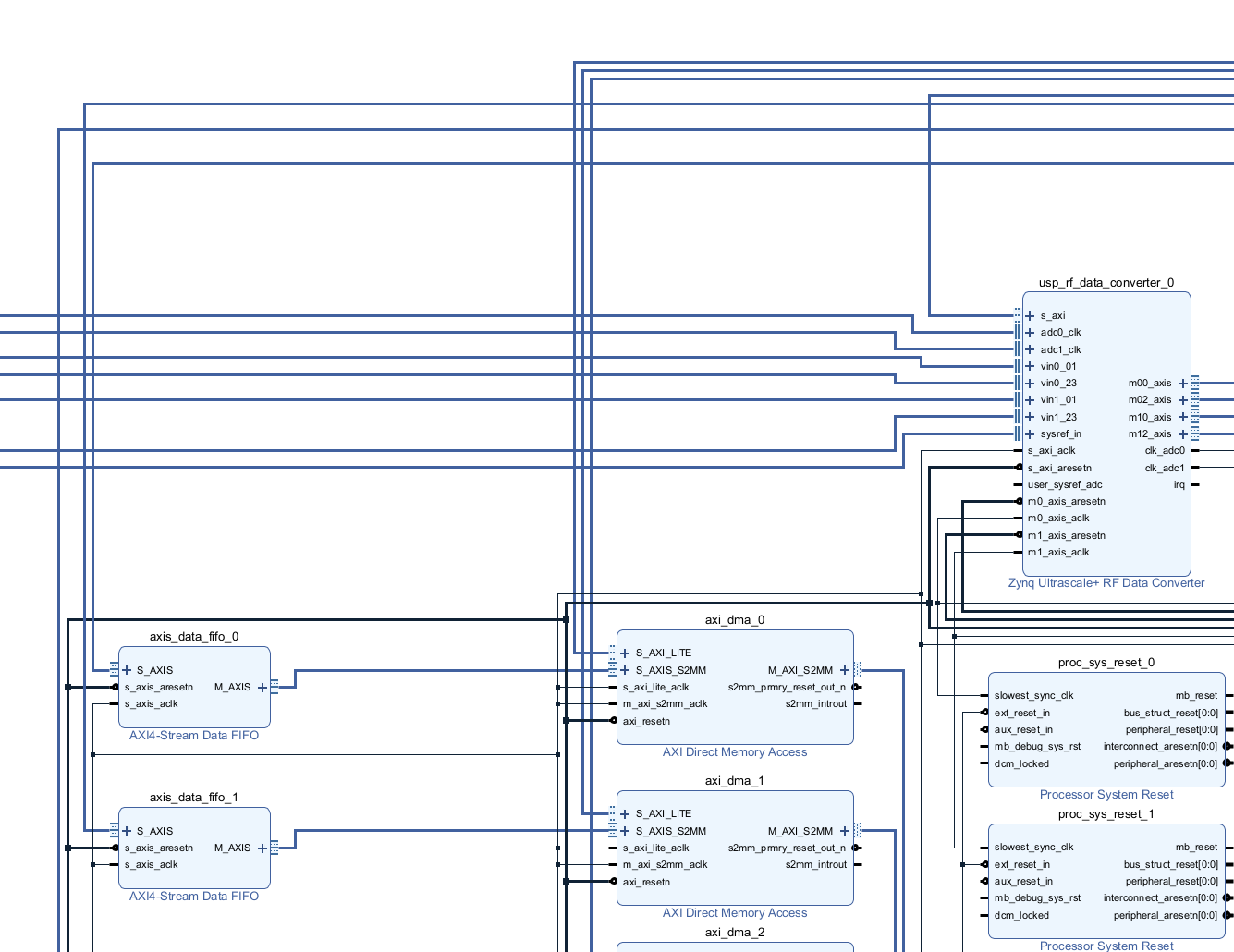
1. Add IPs:  
   
2. Connect All the Blocks as follow:

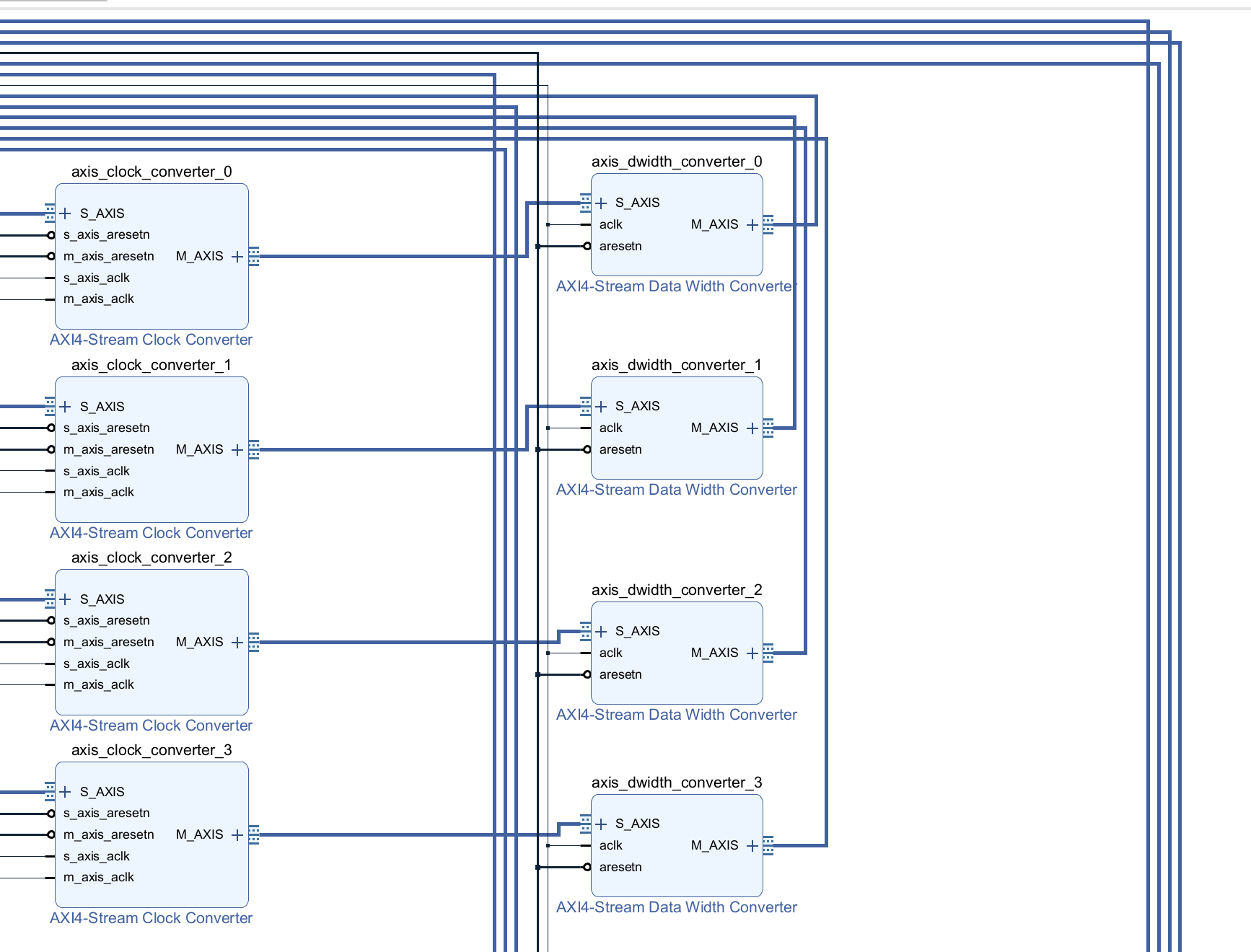


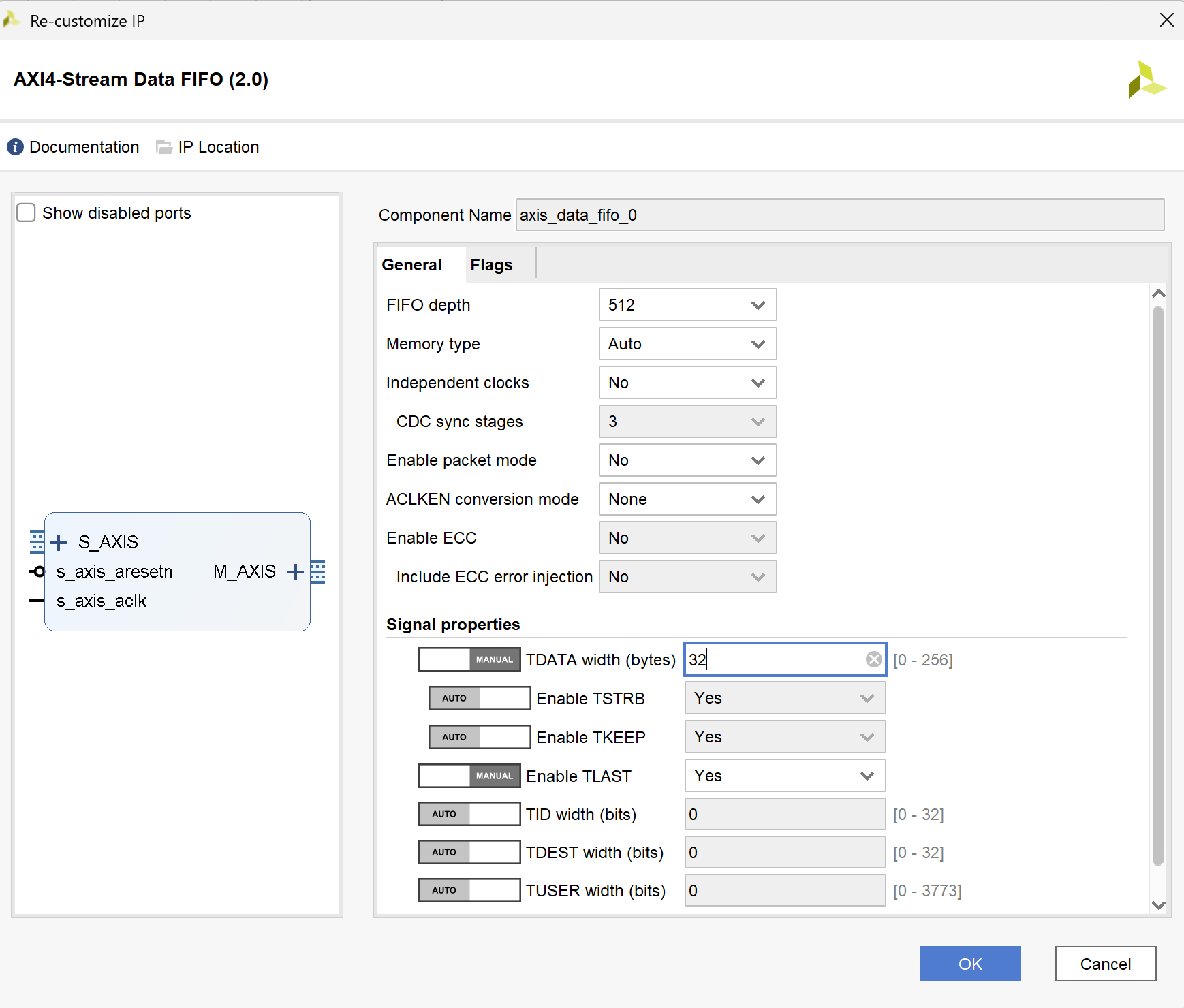


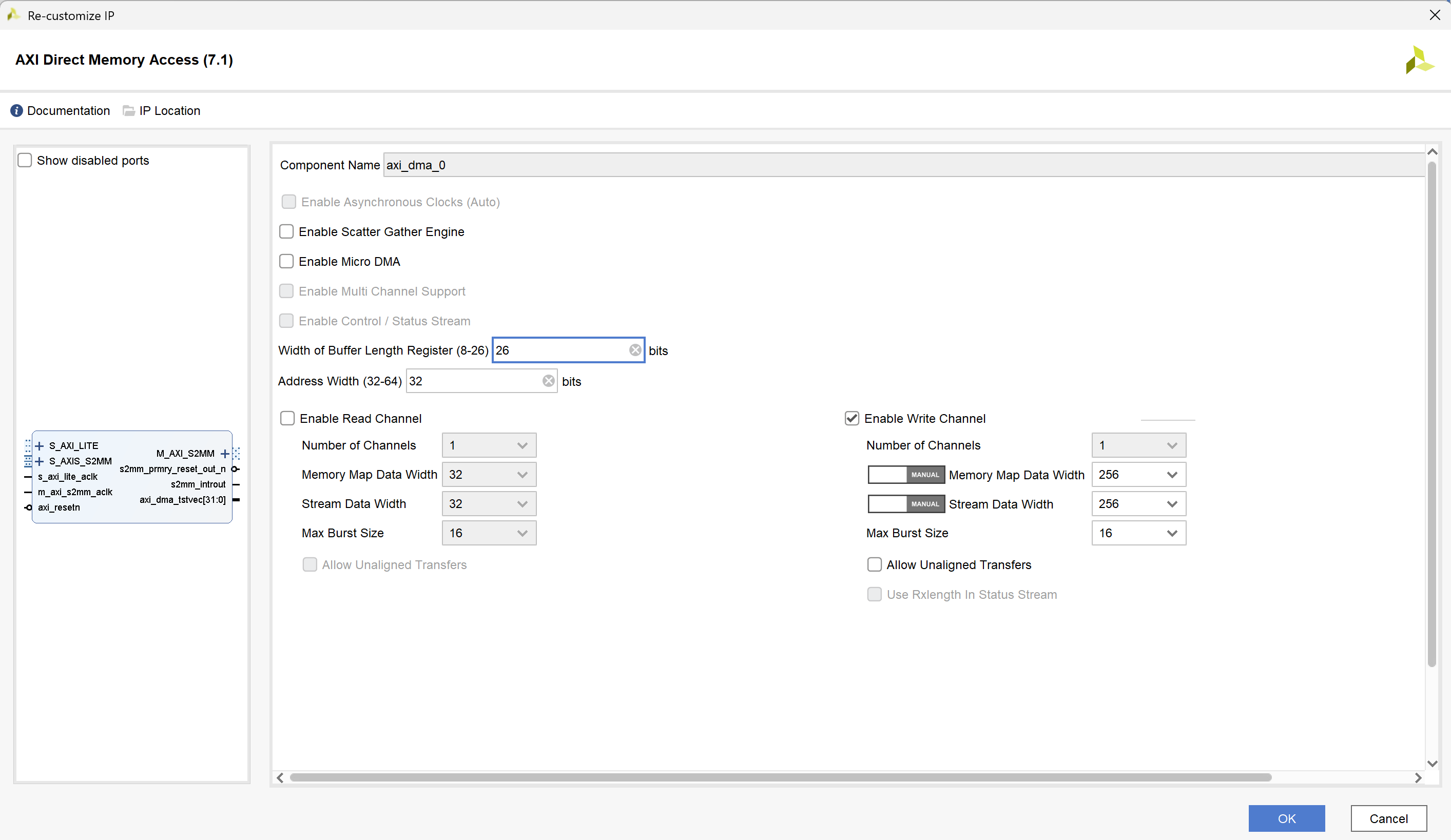


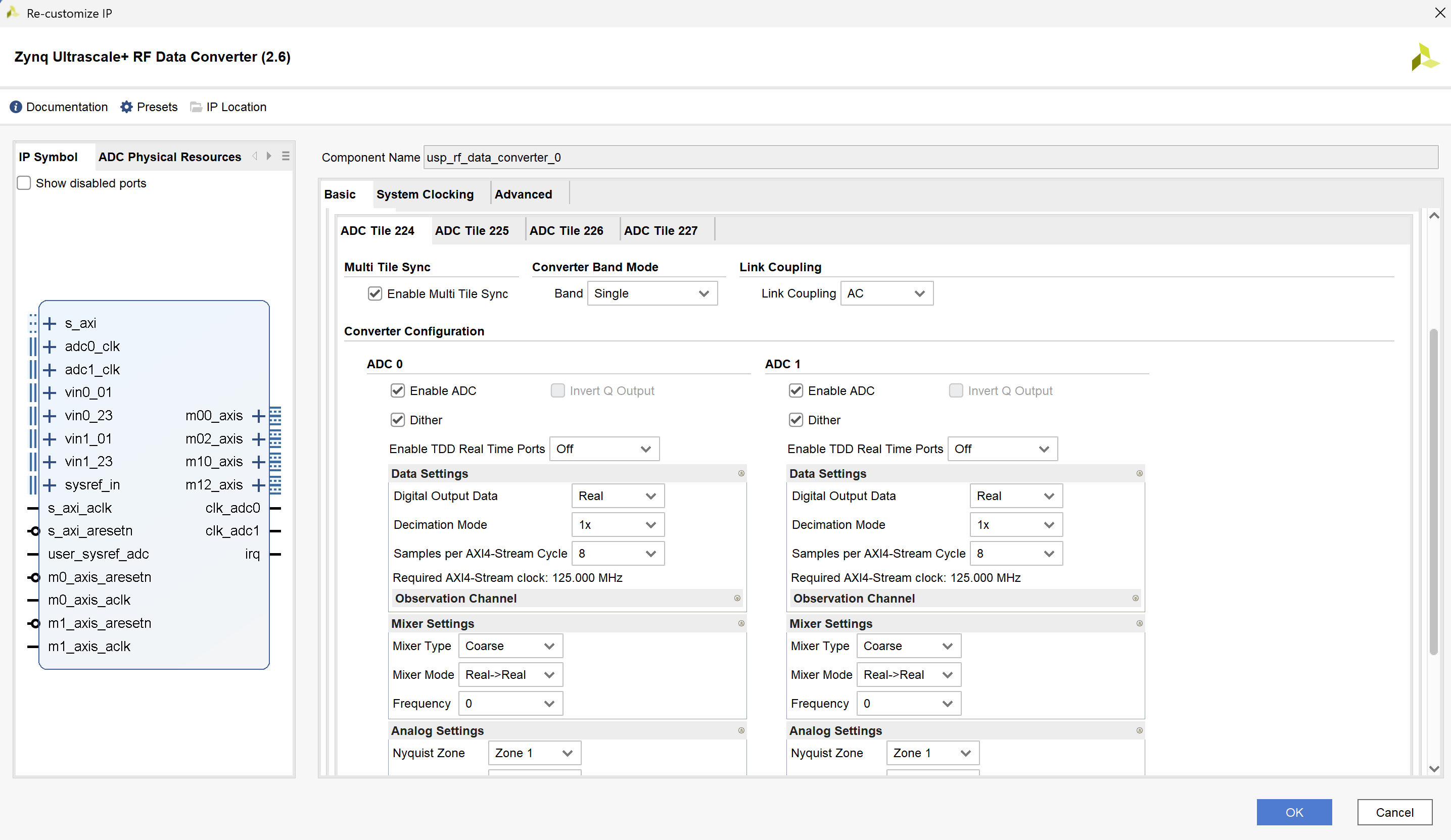


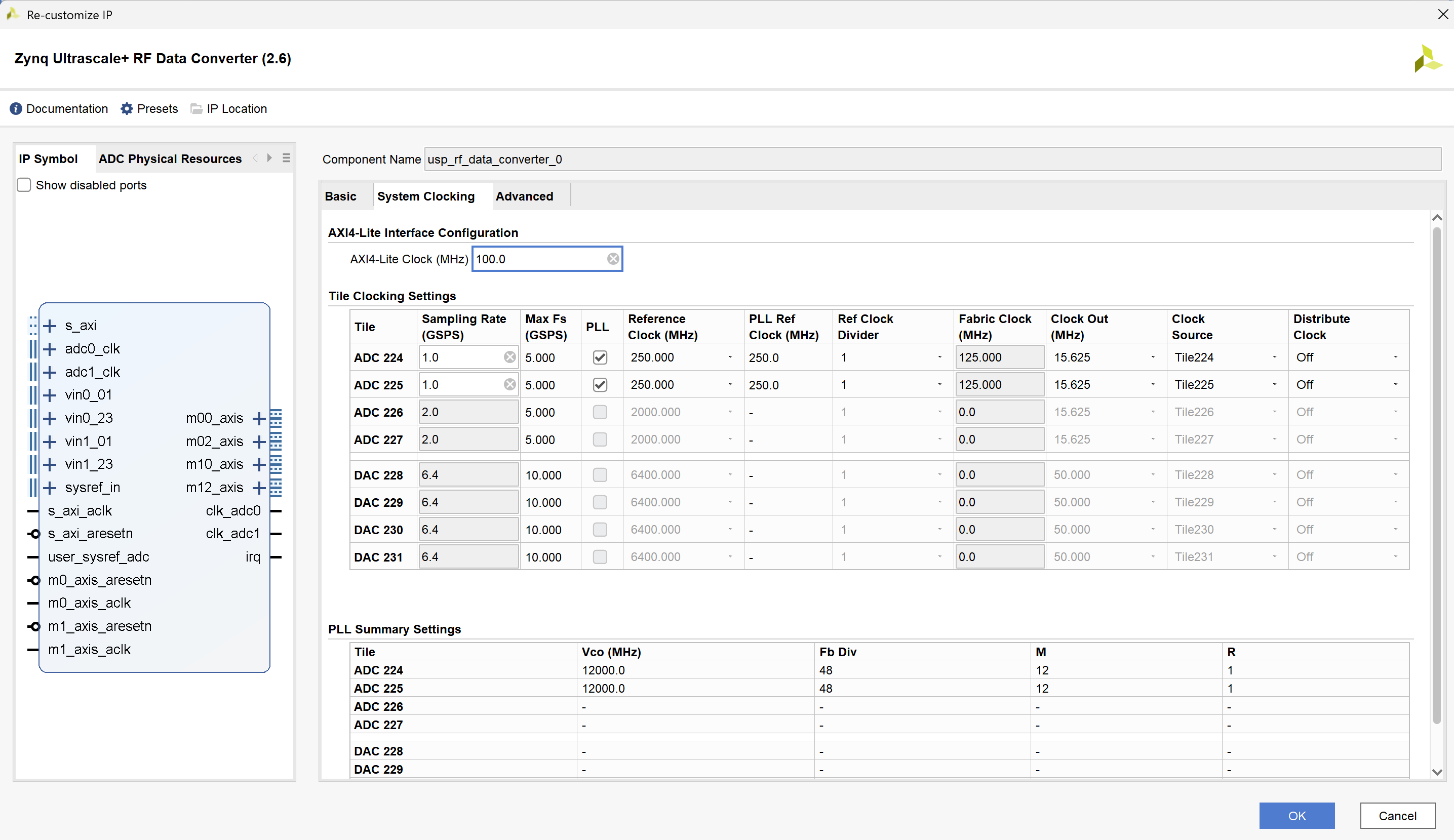


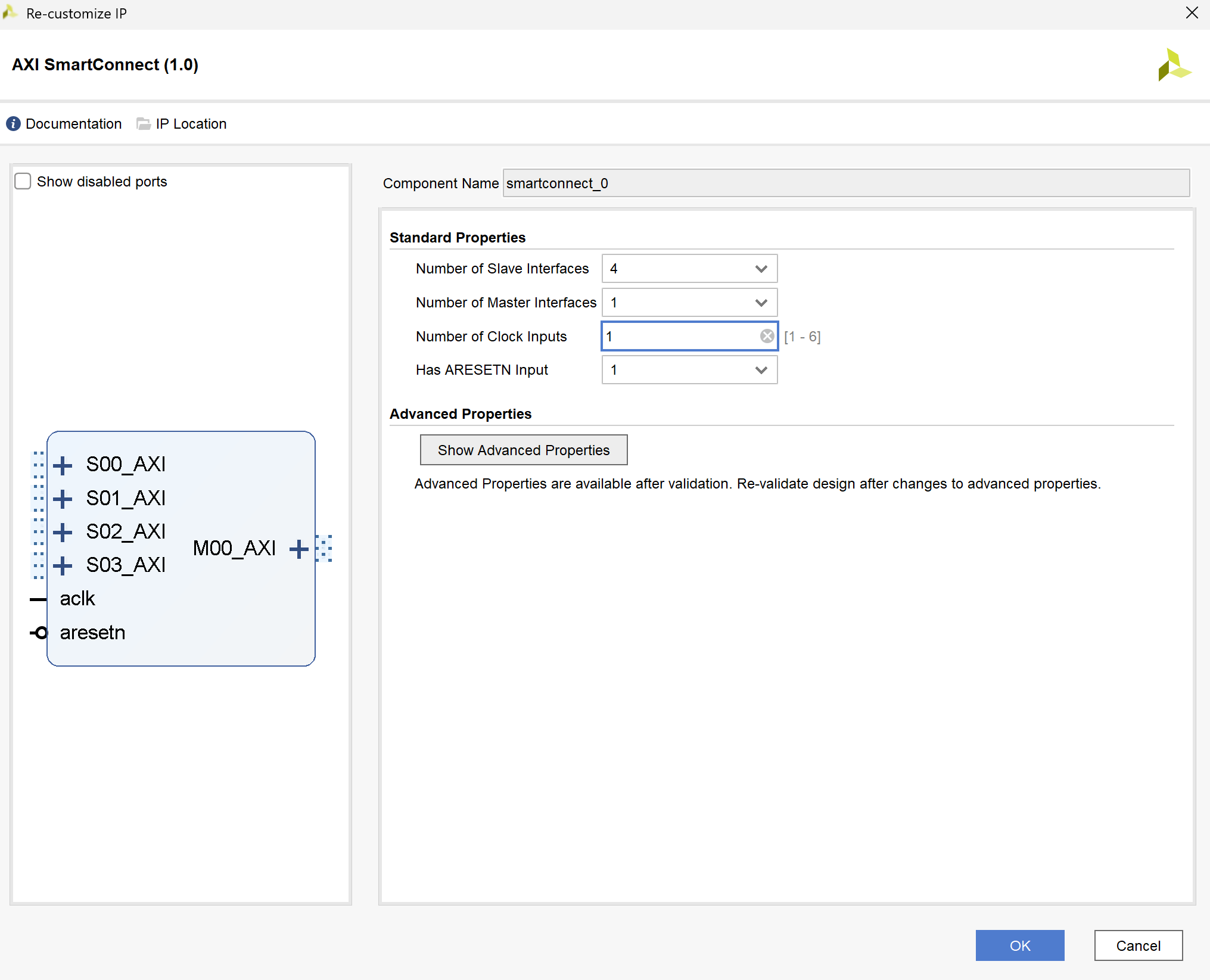


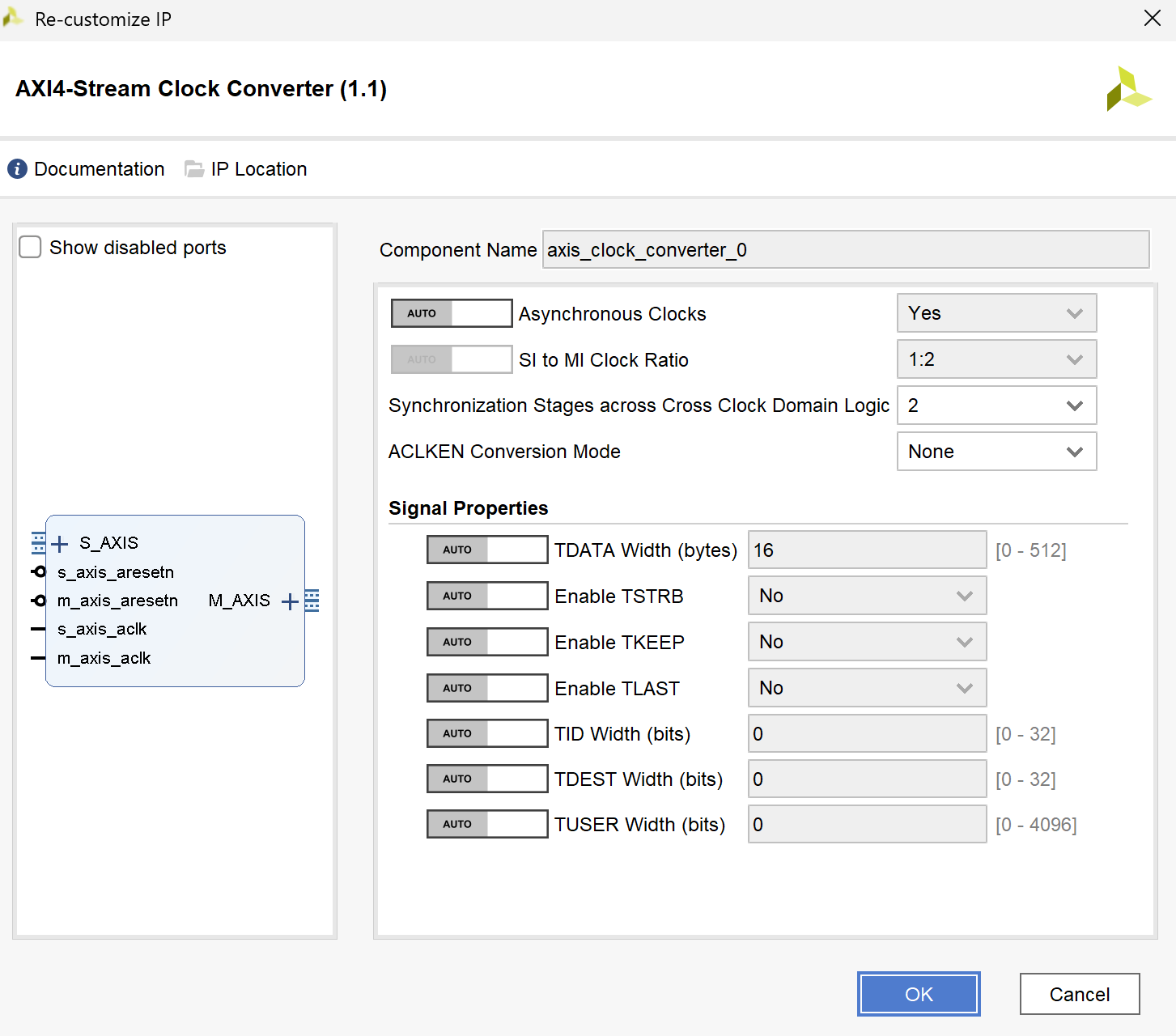
1. See all the setting as below:  
   

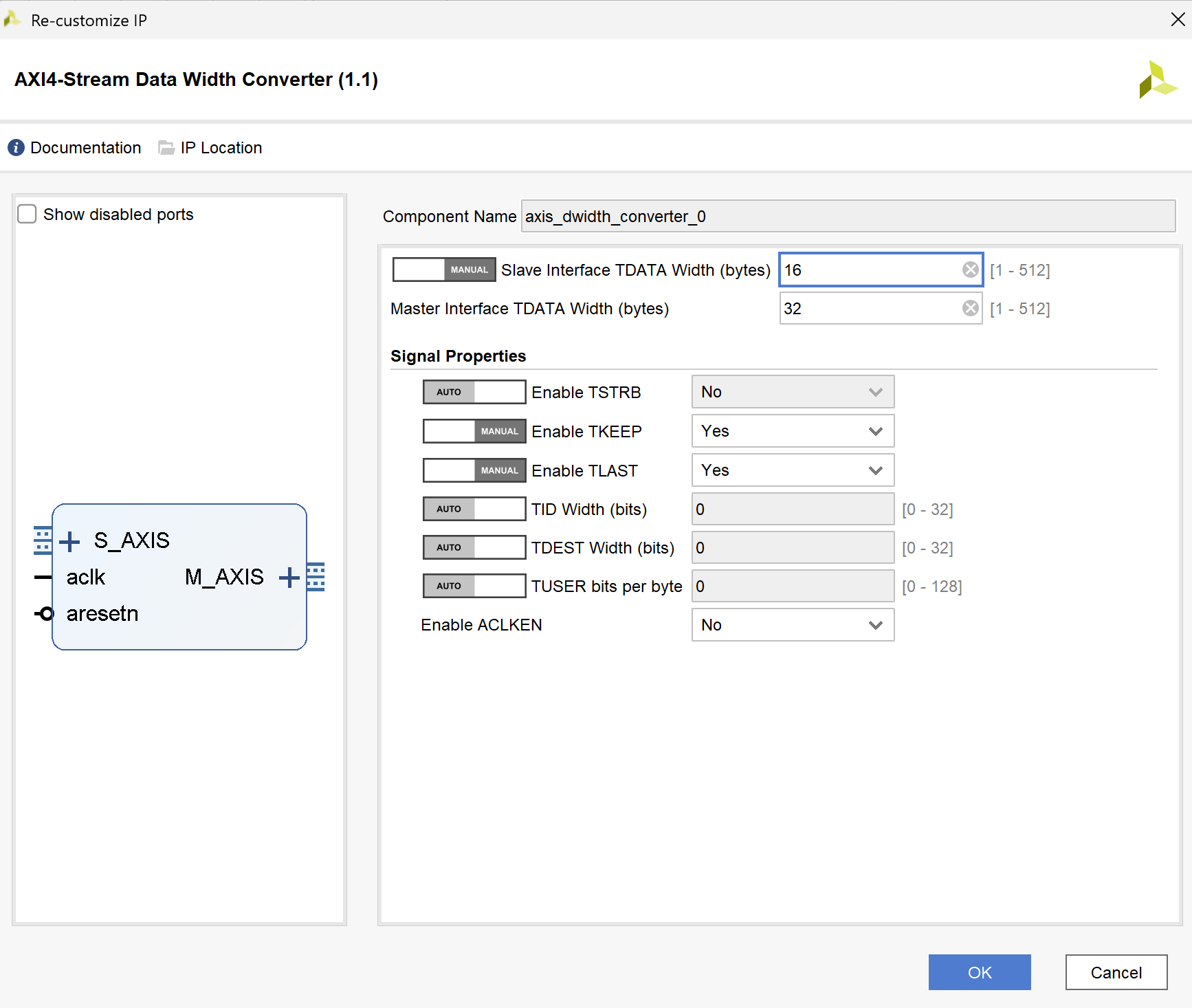


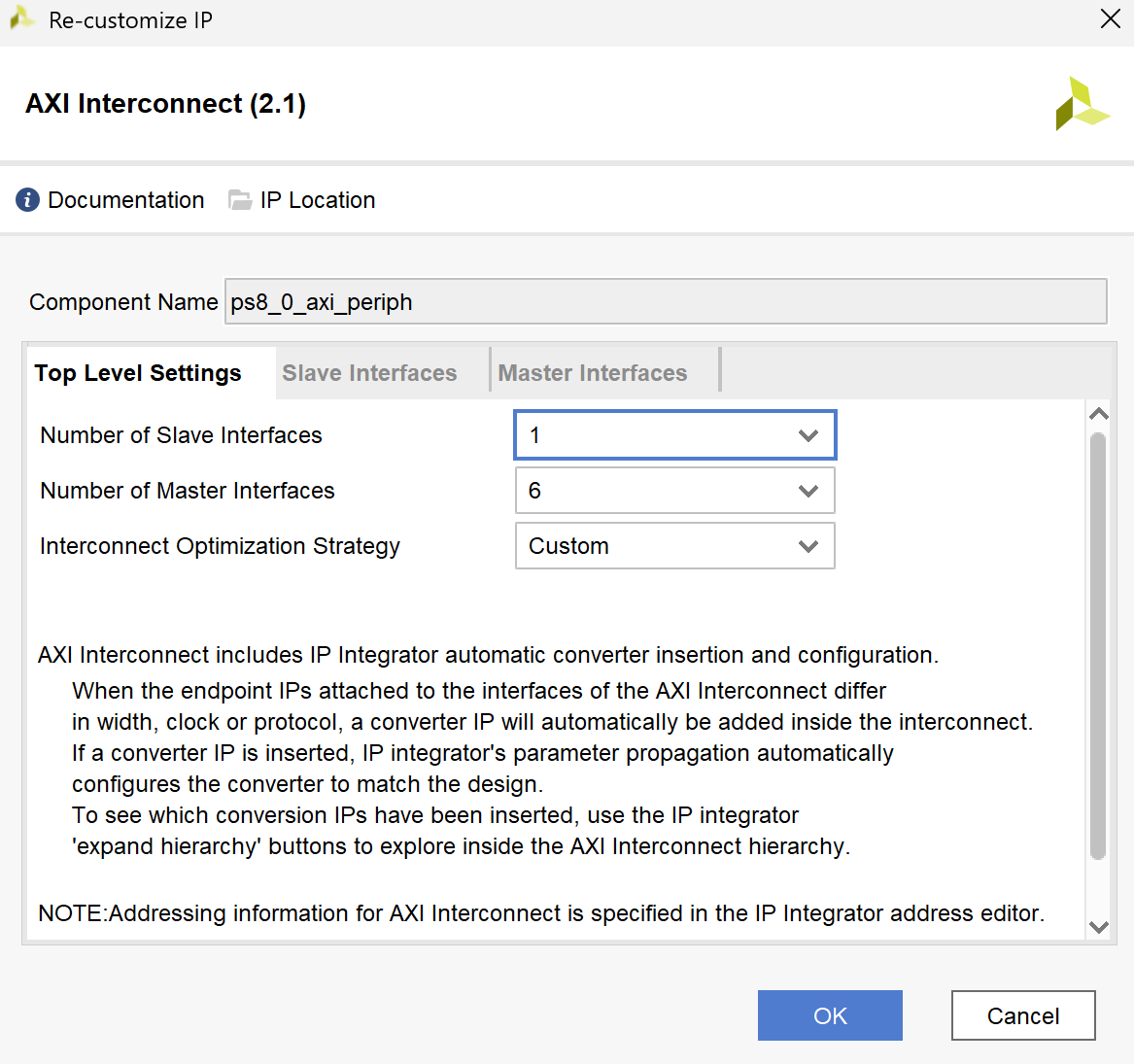


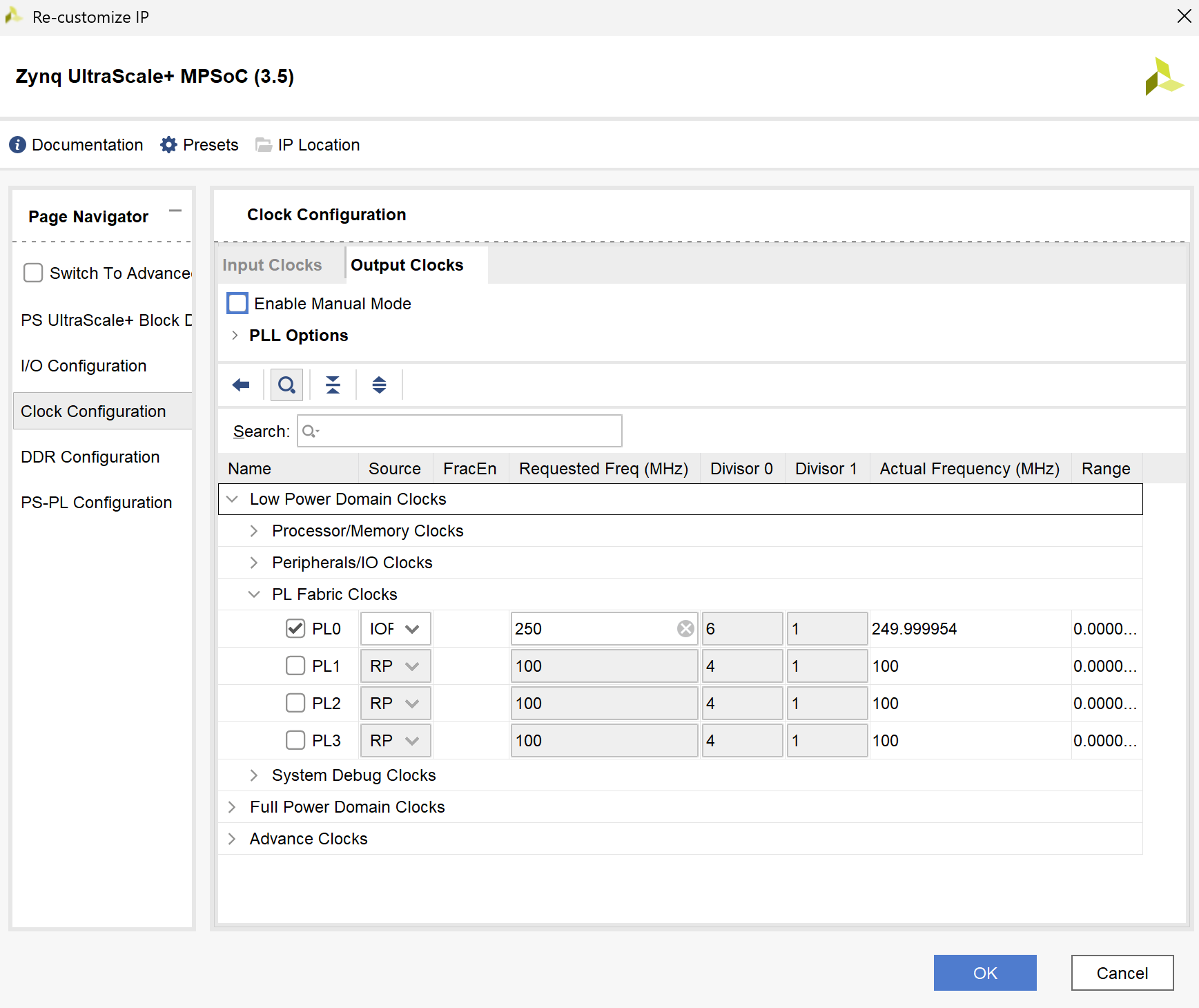


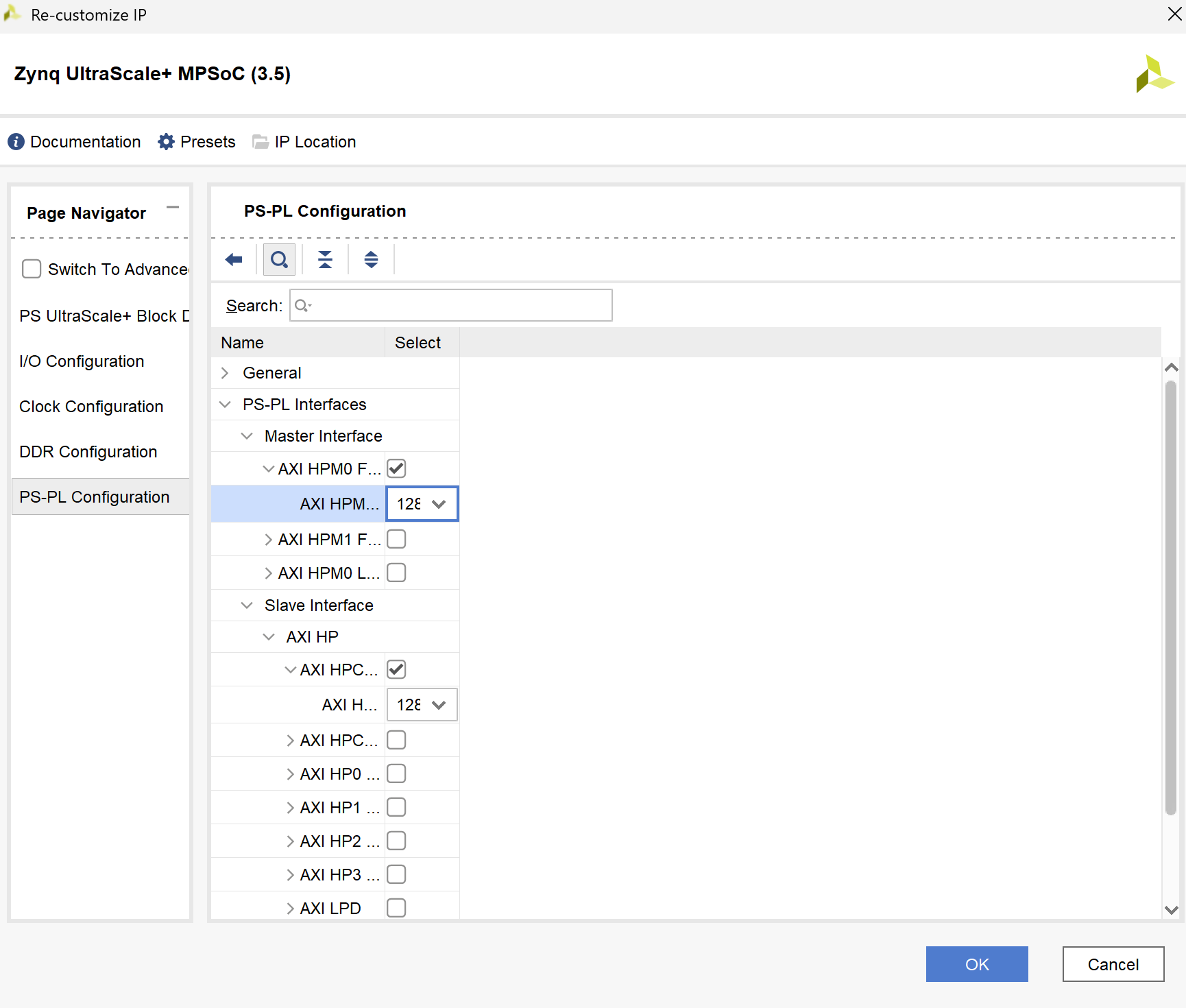




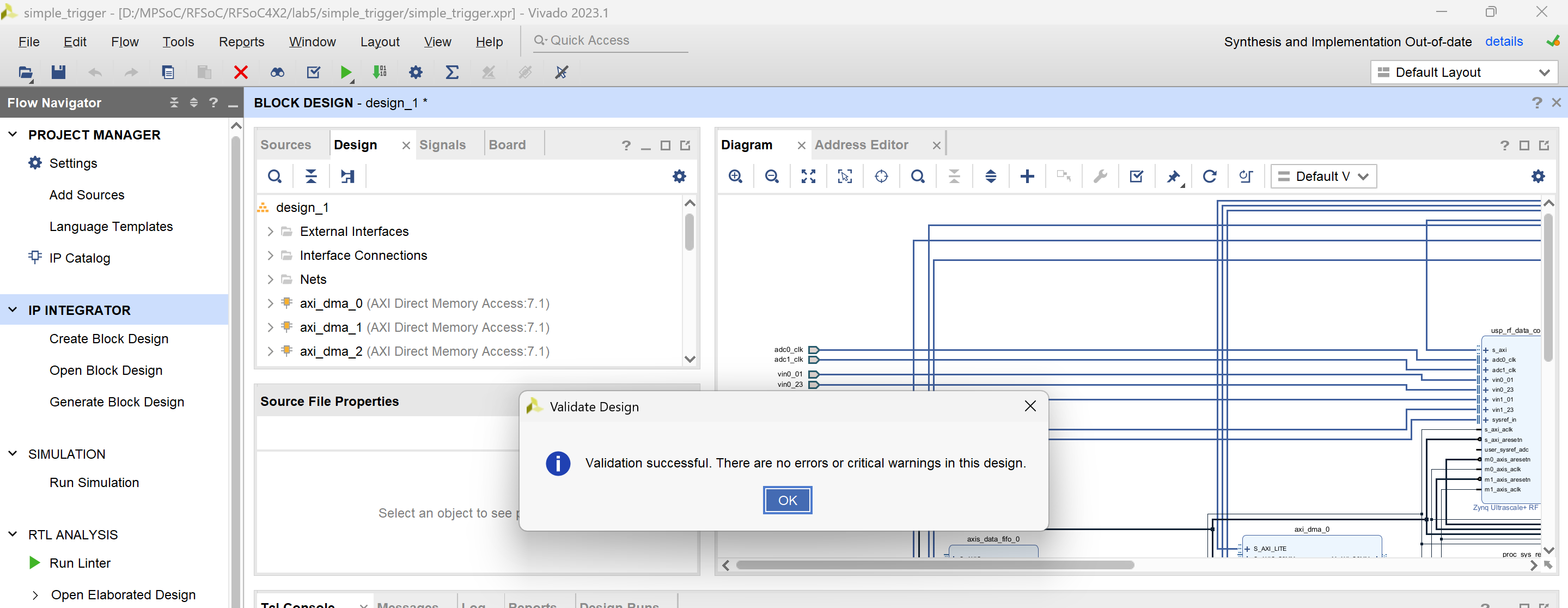




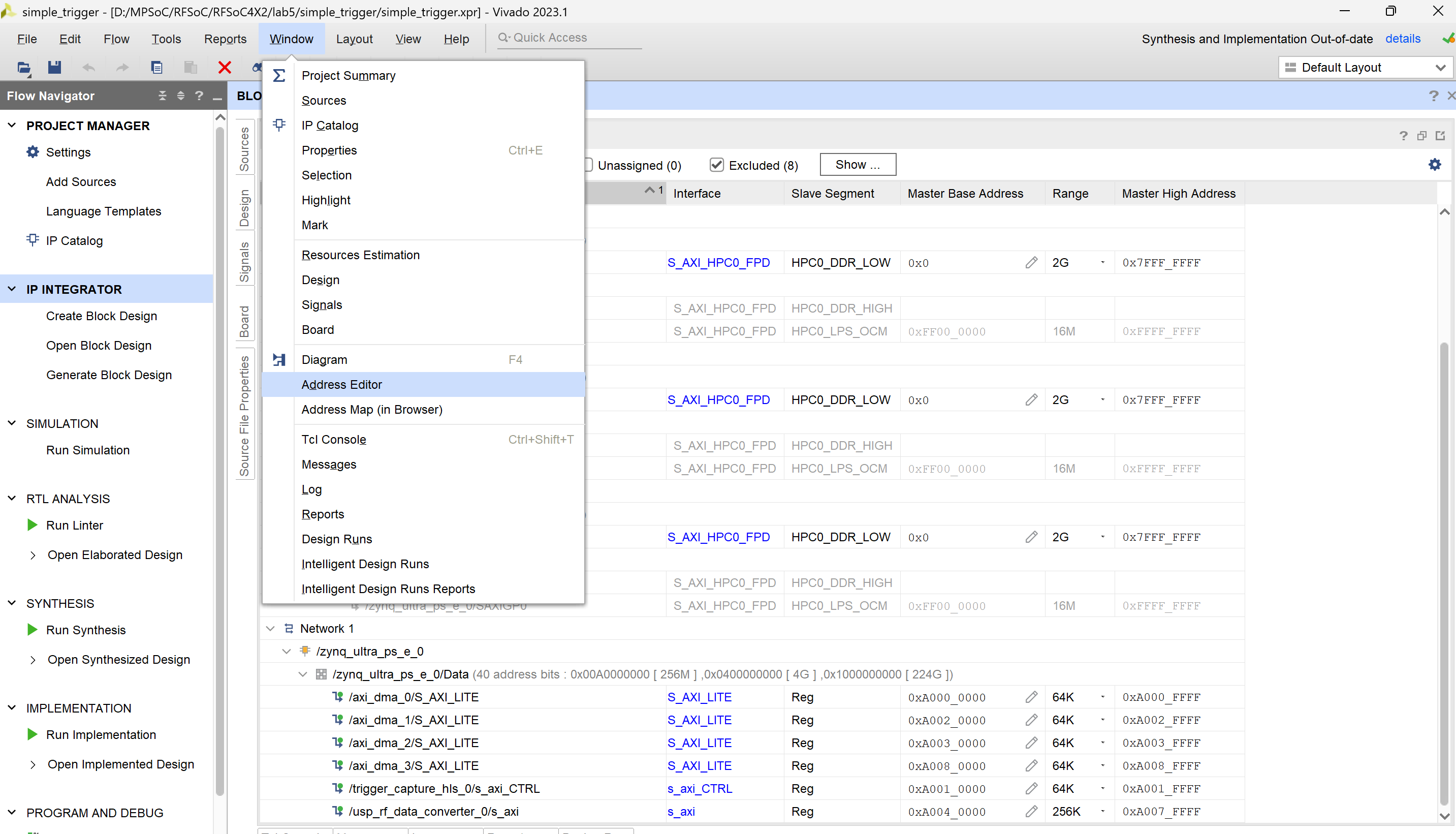




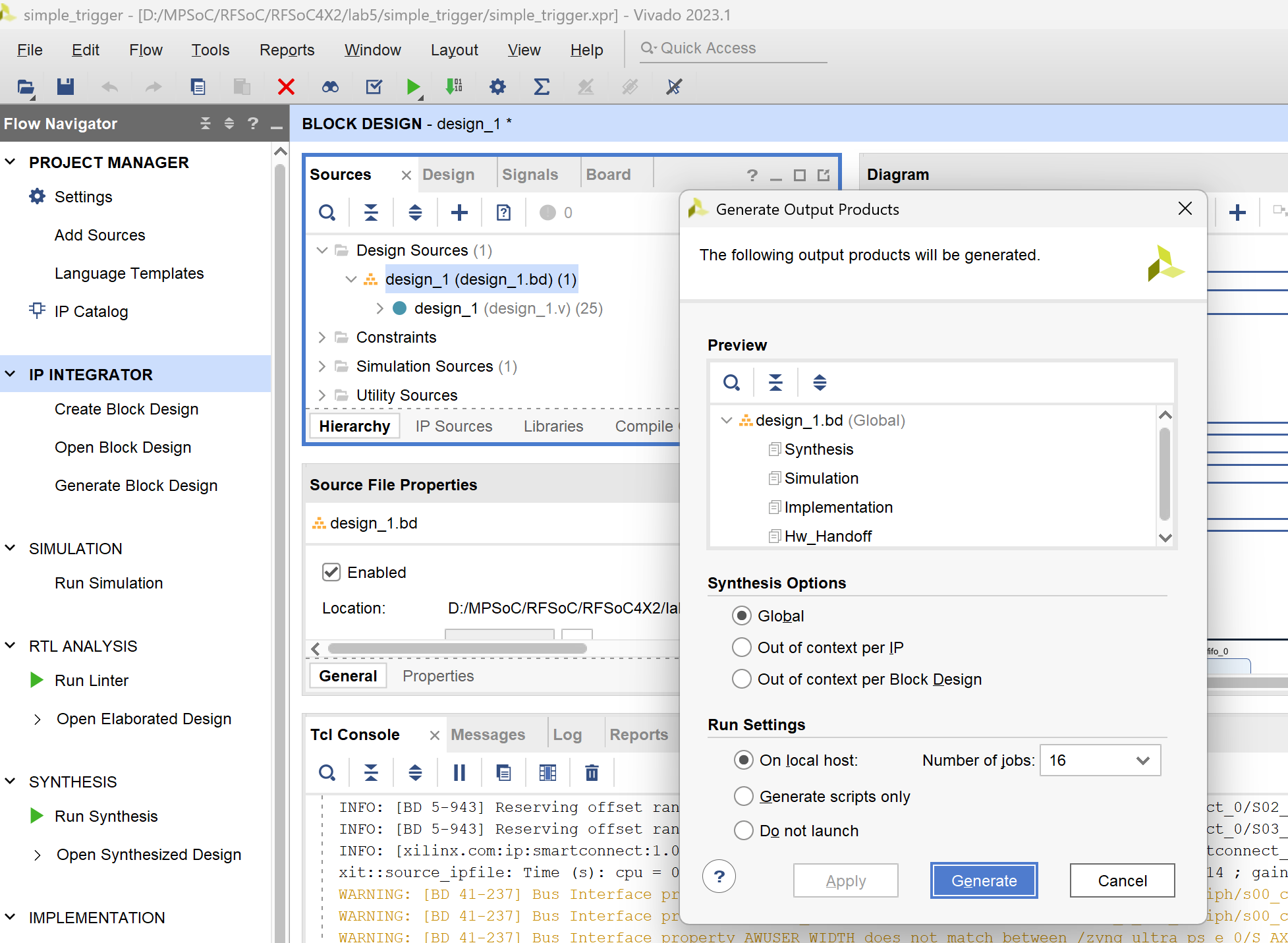
1. Validate Design, it should be no errors or critical warnings



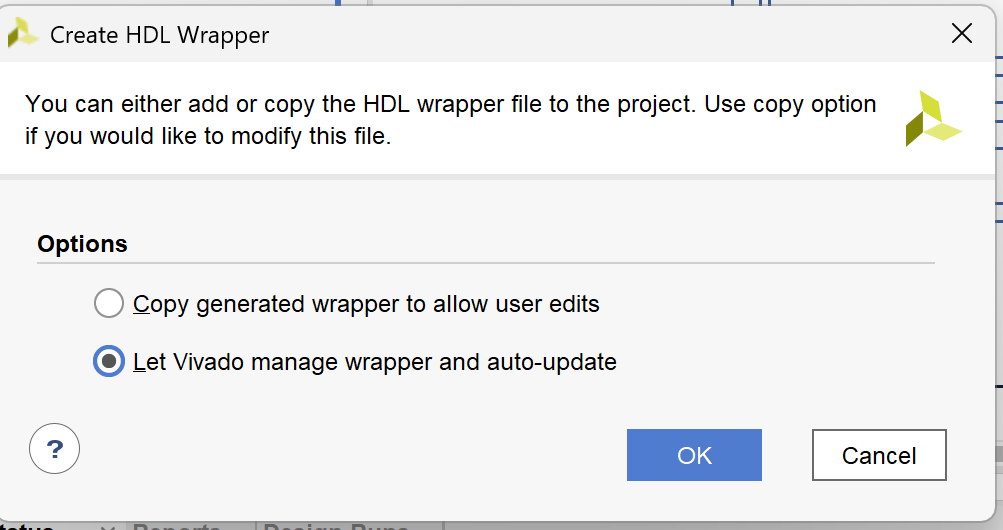
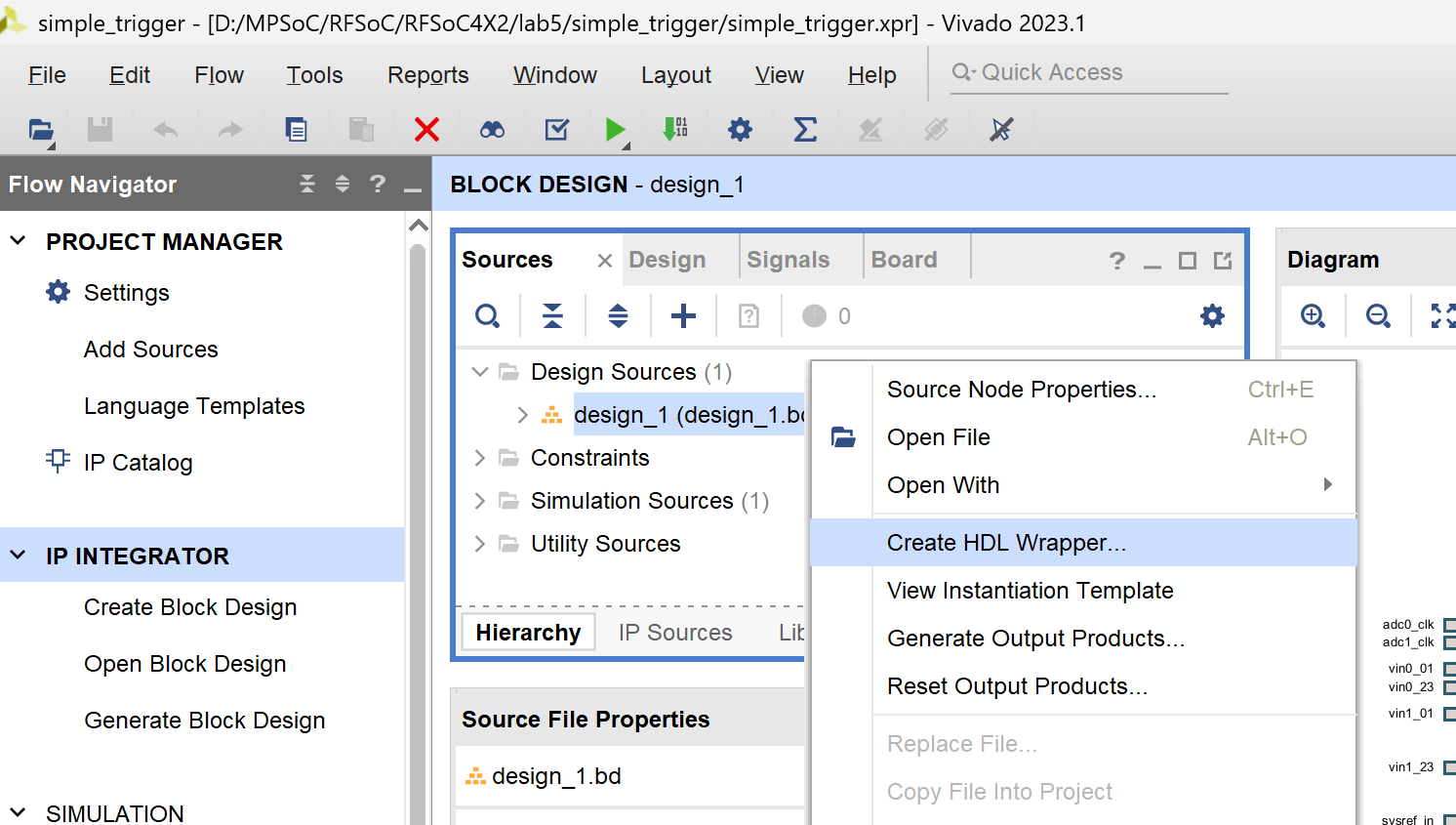
1. Open address editor, properly set the address.



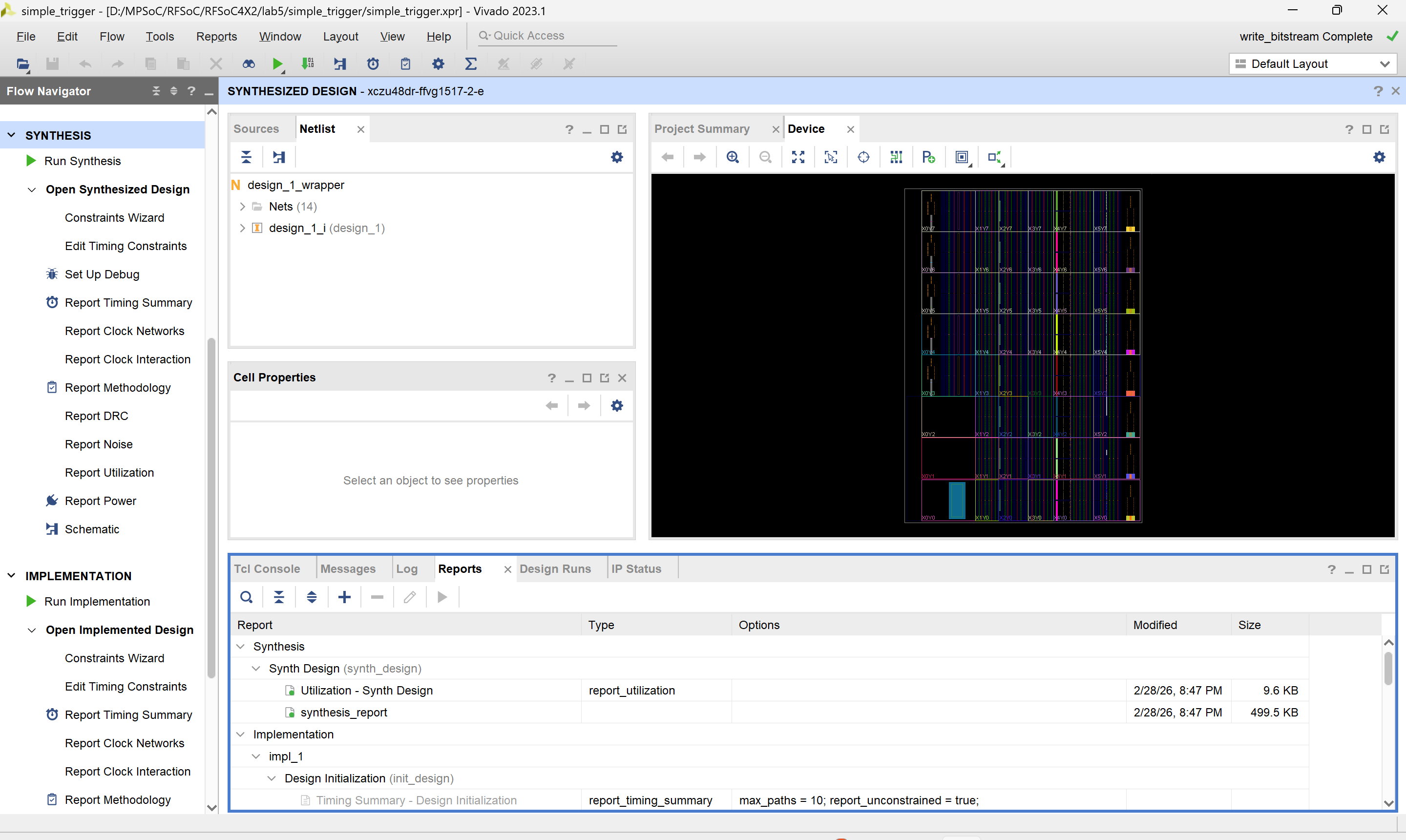
1. Generate output products



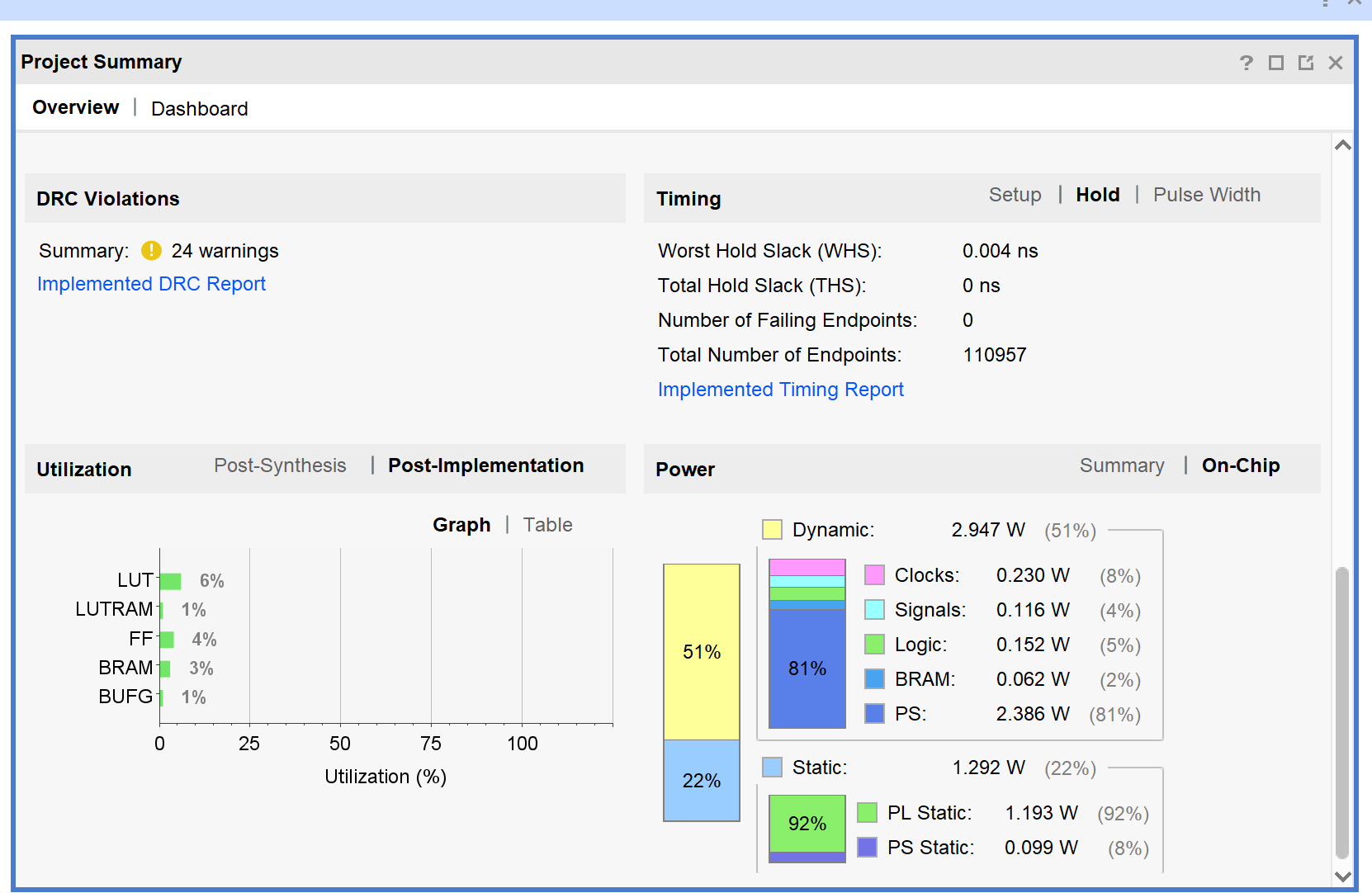
1. Create HDL Wrapper



1. Run Synthesis & Implementation



1. See the result, only about 5W and use <6% of resource.



1. Generate bitstream

**How to use it in PYNQ**

1. Rename “design\_1\_wrapper.bit” & “design\_1.hwh” to “capture.bit / capture.hwh”, be careful that their name should be the same. Create a folder (recommended), e.g. capture\_overlay/. And upload them into RFSoC SD card through Jupyter Notebook interface.
2. Create a new notebook (Python 3) in capture\_overlay/.
3. Use it like a normal python function, the below figure just a illustration, real code see next item:  
   
4. Inside Notebook:

# Notebook Cells (copy/paste)

## Cell 1 — Imports + Load Overlay

from pynq import Overlay, allocate  
import numpy as np  
import time  
  
# Load the overlay (.bit and .hwh must be in the same directory)  
ol = Overlay("capture.bit")  
  
# List available IP blocks in the design  
print("IPs in this overlay:")  
print(list(ol.ip\_dict.keys()))

## Cell 2 — Get handles to HLS IP and DMA IP(s)

You need to adjust the IP names based on what you see in Cell 1 output.

### Option A: Common names (try this first)

# Try the most common names; change if your design uses different names.  
hls = ol.trigger\_capture\_hls\_0  
  
# If you have only one DMA:  
dma0 = ol.axi\_dma\_0  
  
print("HLS register map:")  
print(hls.register\_map)

### Option B: If you have 4 DMAs (typical for 4 ADC channels)

hls = ol.trigger\_capture\_hls\_0  
  
dma0 = ol.axi\_dma\_0  
dma1 = ol.axi\_dma\_1  
dma2 = ol.axi\_dma\_2  
dma3 = ol.axi\_dma\_3  
  
print("HLS register map:")  
print(hls.register\_map)

If the attribute names above don’t exist, use this to pick the correct ones:

# Helper: show IP types and addresses so you can identify the correct blocks  
for name, meta in ol.ip\_dict.items():  
 print(f"{name}: type={meta.get('type')} addr={hex(meta.get('phys\_addr', 0))}")

## Cell 3 — Start the HLS core (ap\_ctrl\_hs: auto-restart + ap\_start)

### Preferred (field-based) method

# Print CTRL fields to confirm names (AP\_START, AUTO\_RESTART, etc.)  
print(hls.register\_map.CTRL)  
  
# Enable auto-restart, then start  
hls.register\_map.CTRL.AUTO\_RESTART = 1  
hls.register\_map.CTRL.AP\_START = 1

### Fallback (raw write) method (works even if fields differ)

# HLS control register usually at offset 0x00:  
# bit0 = ap\_start  
# bit7 = auto\_restart  
hls.write(0x00, (1 << 0) | (1 << 7))

## Cell 4 — Configure thresholds / window parameters (edit names as needed)

First, look at print(hls.register\_map) output, then set the correct register names.

Example:

# Example values (edit to match your design)  
# Use the exact register names shown by hls.register\_map  
  
if hasattr(hls.register\_map, "thr0"):  
 hls.register\_map.thr0 = 1200  
 hls.register\_map.thr1 = 1200  
 hls.register\_map.thr2 = 1200  
 hls.register\_map.thr3 = 1200  
  
if hasattr(hls.register\_map, "window\_clk"):  
 hls.register\_map.window\_clk = 25  
  
if hasattr(hls.register\_map, "post\_clk"):  
 hls.register\_map.post\_clk = 25  
  
# Optional: clear sticky flags (only if you have such a register)  
if hasattr(hls.register\_map, "clear\_trigger"):  
 hls.register\_map.clear\_trigger = 1  
 hls.register\_map.clear\_trigger = 0  
  
print("Configured.")

## Cell 5 — Allocate DMA buffers (256-bit stream ⇒ 32 bytes per beat)

You must match the DMA transfer size to your **frame length** (TLAST).  
If your TLAST ends the frame after beats\_per\_frame beats:

bytes = beats\_per\_frame \* 32

uint32 words = bytes / 4 = beats\_per\_frame \* 8

# Set this to your expected frame length in beats (must match TLAST framing)  
beats\_per\_frame = 1024 # <-- change to your real frame length  
  
words\_per\_frame = beats\_per\_frame \* 8 # 256-bit(32B)=8\*uint32  
buf0 = allocate(shape=(words\_per\_frame,), dtype=np.uint32)  
  
# If you have 4 DMAs:  
buf1 = allocate(shape=(words\_per\_frame,), dtype=np.uint32)  
buf2 = allocate(shape=(words\_per\_frame,), dtype=np.uint32)  
buf3 = allocate(shape=(words\_per\_frame,), dtype=np.uint32)  
  
print("Buffers allocated.")  
print("words\_per\_frame =", words\_per\_frame, " (uint32)")  
print("bytes\_per\_frame =", beats\_per\_frame \* 32, " (bytes)")

## Cell 6 — Capture ONE frame (single DMA)

# Optional: wait for a trigger/event register (only if your design provides one)  
def wait\_for\_trigger(timeout\_s=5.0):  
 t0 = time.time()  
 while True:  
 if hasattr(hls.register\_map, "trigger\_event"):  
 if int(hls.register\_map.trigger\_event) == 1:  
 return True  
 if time.time() - t0 > timeout\_s:  
 return False  
  
# Wait for trigger (optional)  
got = wait\_for\_trigger(timeout\_s=5.0)  
print("Trigger seen?" , got)  
  
# Start DMA receive  
dma0.recvchannel.transfer(buf0)  
dma0.recvchannel.wait()  
  
# Cache invalidate before reading  
buf0.invalidate()  
  
frame0 = np.array(buf0) # copy into a normal numpy array  
print("Captured frame0 words:", frame0.size)  
print("First 16 words:", frame0[:16])

## Cell 7 — Capture ONE frame (4 DMAs in parallel)

# Optional: wait for trigger  
got = wait\_for\_trigger(timeout\_s=5.0)  
print("Trigger seen?" , got)  
  
dmas = [dma0, dma1, dma2, dma3]  
bufs = [buf0, buf1, buf2, buf3]  
  
# Kick off all transfers  
for i in range(4):  
 dmas[i].recvchannel.transfer(bufs[i])  
  
# Wait all complete  
for i in range(4):  
 dmas[i].recvchannel.wait()  
 bufs[i].invalidate()  
  
frames = [np.array(b) for b in bufs]  
  
for i, f in enumerate(frames):  
 print(f"Captured frame{i} words:", f.size, "first 8:", f[:8])

## Cell 8 — Save captured data to disk (optional)

# Save as .npy files  
np.save("frame0.npy", frame0)  
  
# If 4 channels:  
for i, f in enumerate(frames):  
 np.save(f"frame\_ch{i}.npy", f)  
  
print("Saved.")