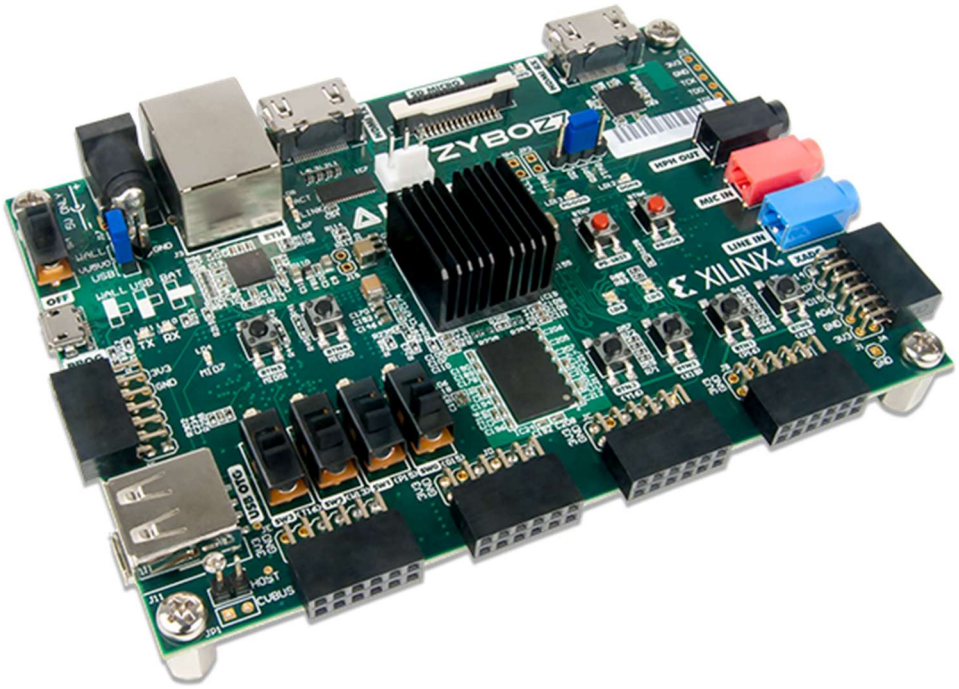
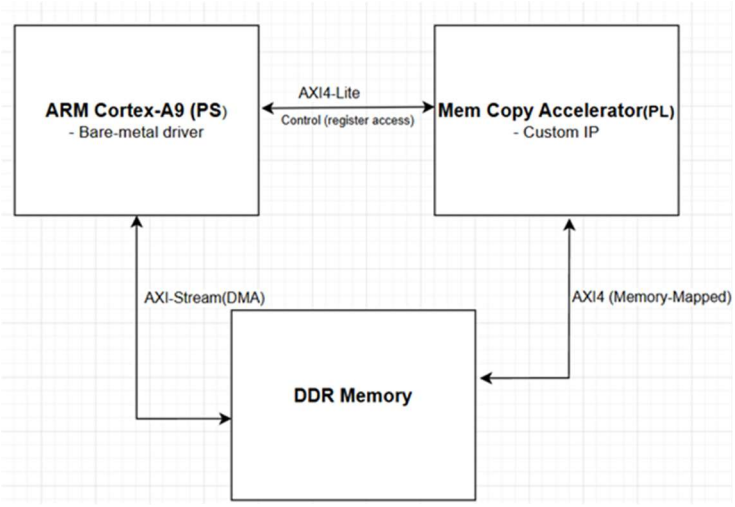
Design and Implementation of an Memory Copy Accelerator on Zybo Z7-10
Step-by-Step Guide



1. Introduction

This guide walks through the design and implementation of a **Memory Copy Accelerator** using **Vitis HLS**, **Vivado**, and **Vitis IDE** on the **Zybo Z7-10** board. The accelerator is designed to offload memory copy operations from the CPU and demonstrate interrupt-driven hardware-software co-design.



System Block Diagram Overview

System Overview

- **Platform:** Xilinx Zybo Z7-10 (Zynq-7000 SoC)
- **Processor:** ARM Cortex-A9
- **FPGA Fabric:** Custom AXI4-Lite IP for memory copy
- **Memory Interface:** DDR3 via AXI ports

2. Project Overview

Objective

Design a hardware accelerator to perform block memory copy (memcpy) faster than the software equivalent running on the ARM Cortex-A9 processor.

Features

- Implemented in **C/C++ using Vivado HLS**.
- AXI interfaces for DDR access (M_AXI) and control (S_AXI_LITE).
- **Interrupt-driven completion** (ap_ctrl_hs + interrupt output).
- Integrated and tested on **Zybo Z7-10** via **Vitis bare-metal application**.

Tools & Versions

- **Vivado HLS / Vitis HLS** 2020.2 to 2022.1
- **Vivado Design Suite** 2020.2 or 2022.1
- **Vitis IDE** for application development

Installation & Getting Started

For installation instructions, Digilent board file setup, and beginner tutorials, visit the official Digilent guide: 👉 https://digilent.com/reference/programmable-logic/guides/start

This resource provides detailed steps for:

- Installing **Vivado** and **Vitis**
- Setting up **Digilent Board Files**
- Learning from simple example projects and tutorials

Development Flow

1. **Vivado HLS**
    - Write and verify memcopy_accel.cpp
    - Simulate with memcopy_accel_tb.cpp
    - Synthesize and export as IP core
2. **Vivado Block Design**
    - Integrate HLS IP via AXI4-Lite Connect to PS
    - Generate bitstream
3. **Vitis Software Development**
    - Create bare-metal application `memcopy_accel(ip_base, src, dst, len);`
    - Compare performance vs `memcpy()`
4. **Testing & Validation**
    - Measure latency and throughput
    - Verify correctness (data integrity check)

3. AXI Protocol Overview

Reference: [AXI Basic Blog Series – AMD Adaptive Support]

What is AXI?

The **Advanced eXtensible Interface (AXI)** is part of the ARM AMBA protocol family used for high-performance and high-frequency data transfers between IP blocks in SoCs like the Zynq-7000.

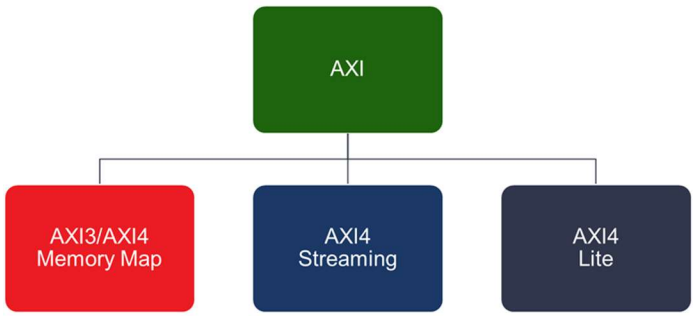


Figure 1. Three types of AXI4-Interfaces

AXI Types Used in This Project

Interface	Description	Used For
AXI4 (Full)	High-throughput memory-mapped interface supporting bursts up to 256 data beats.	Data transfer between accelerator and DDR (M_AXI_SRC / M_AXI_DST)
AXI4-Lite	Simplified version of AXI4 with single 32-bit transfers, ideal for control registers.	Register interface between PS and accelerator (S_AXI_CTRL)
AXI4-Stream	Stream-based interface for continuous data flow without addressing overhead.	Not used in this design, but suitable for future DMA-style streaming.

AXI4 Handshake Signals

Each AXI channel uses VALID and READY signals for synchronization:

- **VALID**: Indicates the master has placed valid data or address on the bus.
- **READY**: Indicates the slave is ready to accept data or address.
- A transfer occurs only when both **VALID** and **READY** are high in the same clock cycle.

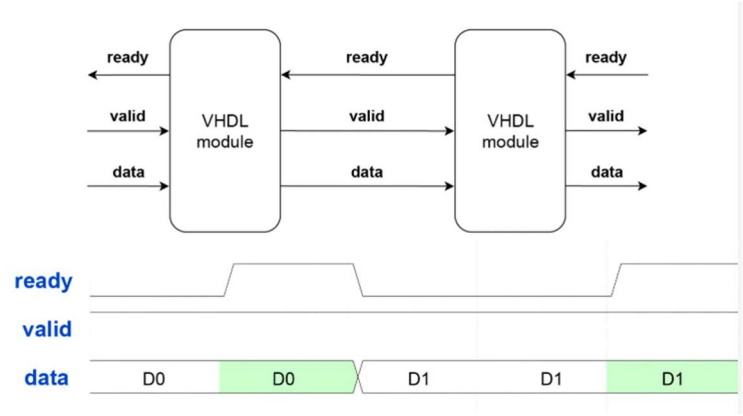


Figure 2. AXI ready/valid handshake works[link]

AXI Channels

The AXI protocol defines 5 channels:

- 2 are used for Read transactions
- 3 are used for Write transactions

Channel	Direction	Function
Read Address (AR)	Master → Slave	Contains read address and control info
Read Data (R)	Slave → Master	Returns read data
Write Address (AW)	Master → Slave	Contains write address and control info
Write Data (W)	Master → Slave	Sends write data
Write Response (B)	Slave → Master	Confirms write completion

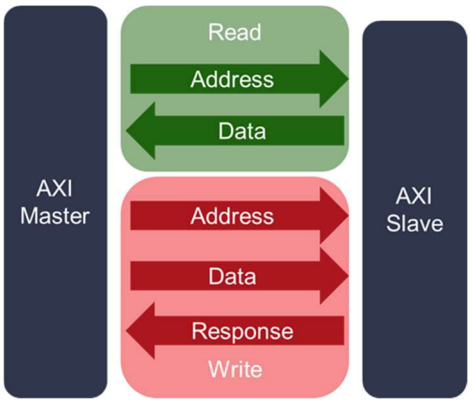


Figure 3. AXI Read and Write Channels

AXI in This Accelerator

- The **memcopy_accel** IP acts as an **AXI Master** on the `M_AXI_SRC` and `M_AXI_DST` ports to read and write DDR memory directly.
- The **Zynq PS** configures the IP via **AXI-Lite**, setting source/destination addresses and transfer length.
- This architecture allows the accelerator to transfer data **independently of the CPU** once started.

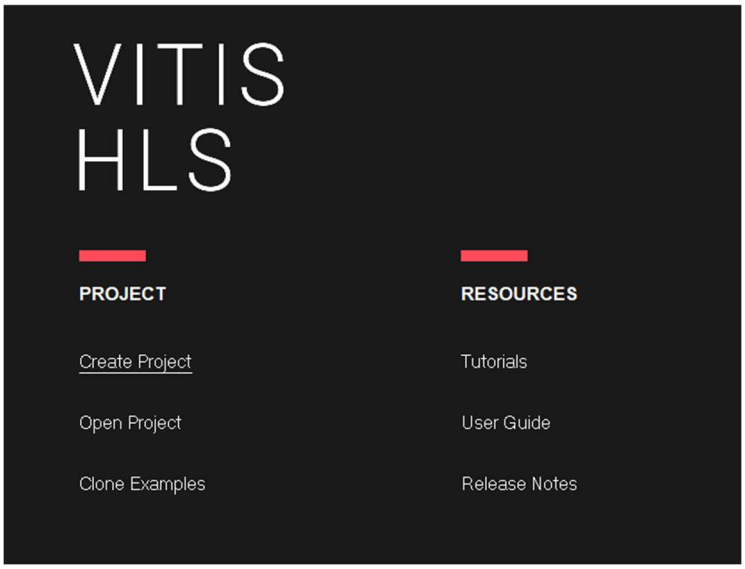
4. HLS Design with Vitis HLS

4.1 Creating a Vitis HLS Project (GUI) — Step-by-Step

The steps below show exactly how to create, configure, and build the HLS IP in **Vitis HLS** for Zybo Z7-10. *(Screenshots indicated below — insert your captures where noted.)*

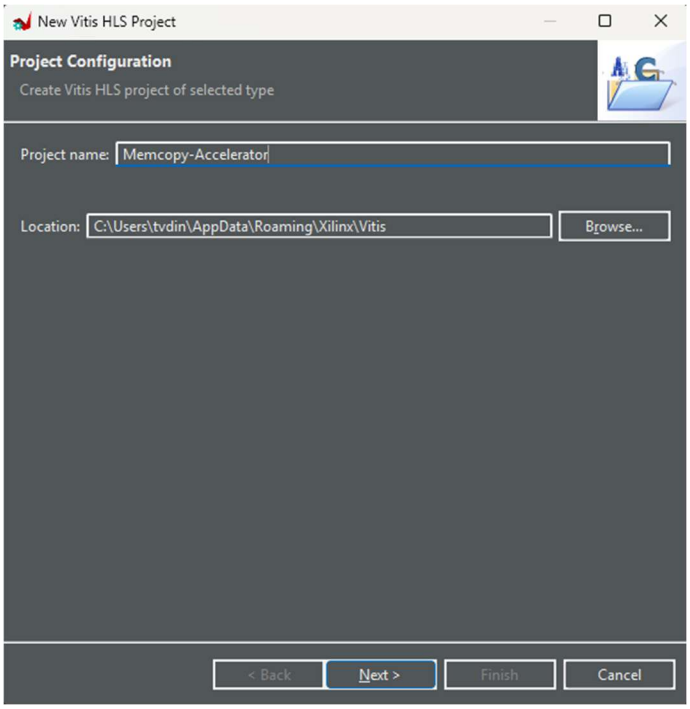
1. Launch Vitis HLS

- Open the **Vitis HLS** GUI. From the welcome page, click **Create Project**.



2. Name & Location

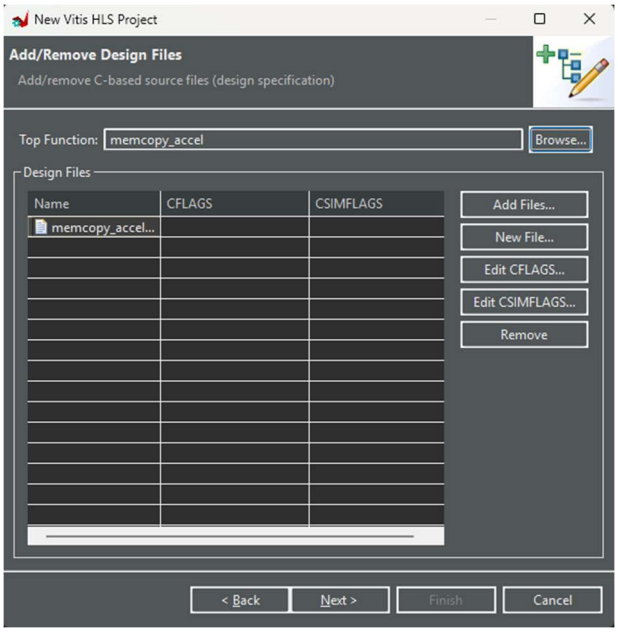
- In the pop-up, enter a project name (e.g., **Memcopy-Accelerator**) and choose a location. Click **Next**.



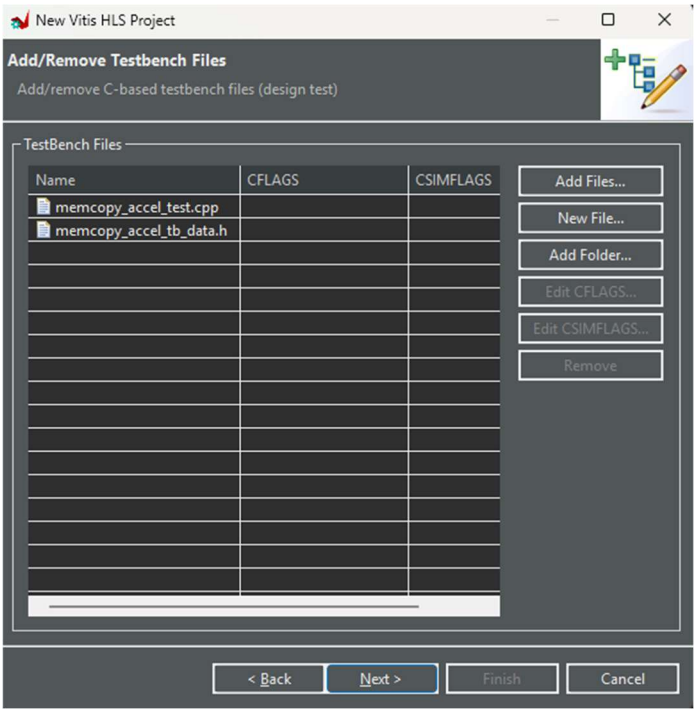
3. Add Sources

- Click **Add Files…** and select the HLS source file `memcopy_accel.cpp` cloned from Github https://github.com/vandinhtranengg/zybo_memcopy_accel_interrupt

- In **Top Function** field, type: **memcopy_accel**. Click **Next**.

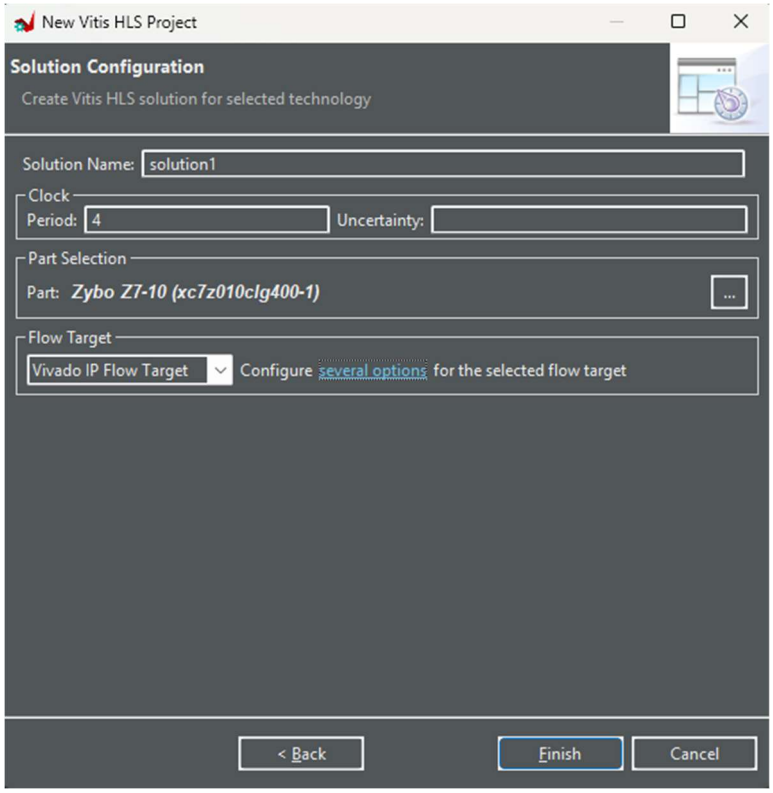


- Add the testbench files `memcopy_accel_test.cpp`, `memcopy_accel_tb_data.h`. Click **Next**.



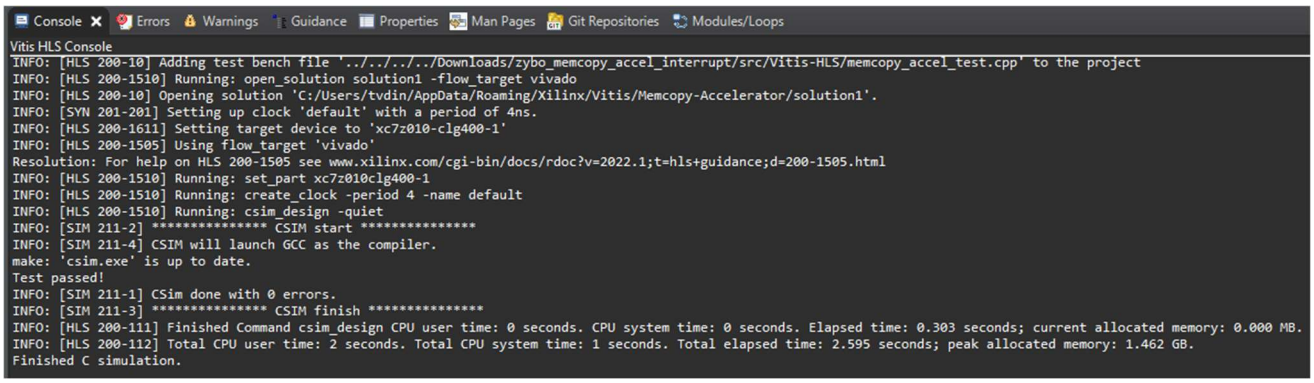
4. Solution & Part

- Create **Solution 1** (default).
- Set Clock Period to 4ns (250 MHz)
- Set **Part** to Zybo Z7-10 device: **xc7z010clg400-1**.
- Choose Vivado IP Flow Target. Click **Finish**.



5. Run C Simulation(csim)

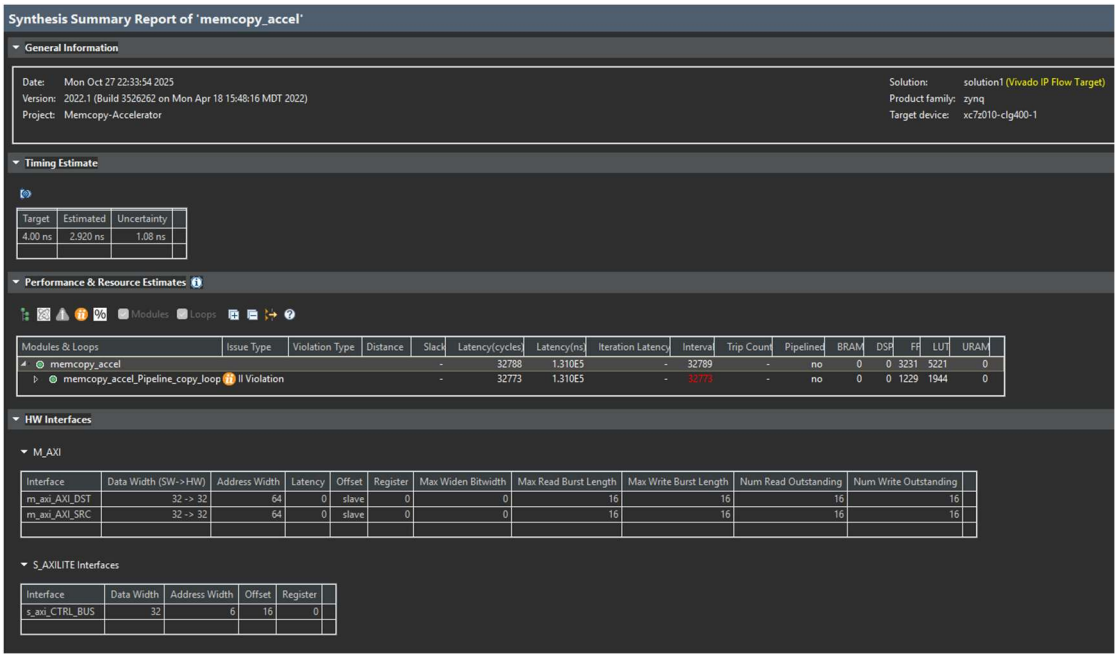
- Click **C Simulation**. Use the default testbench (`memcopy_accel_test.cpp`) to verify functional correctness.
- Check the console for Simulation result.



6. Synthesis (csynth)

- Click **Run C Synthesis** to generate RTL.

- Open the **Synthesis Report** to review resource utilization and performance (II, latency).



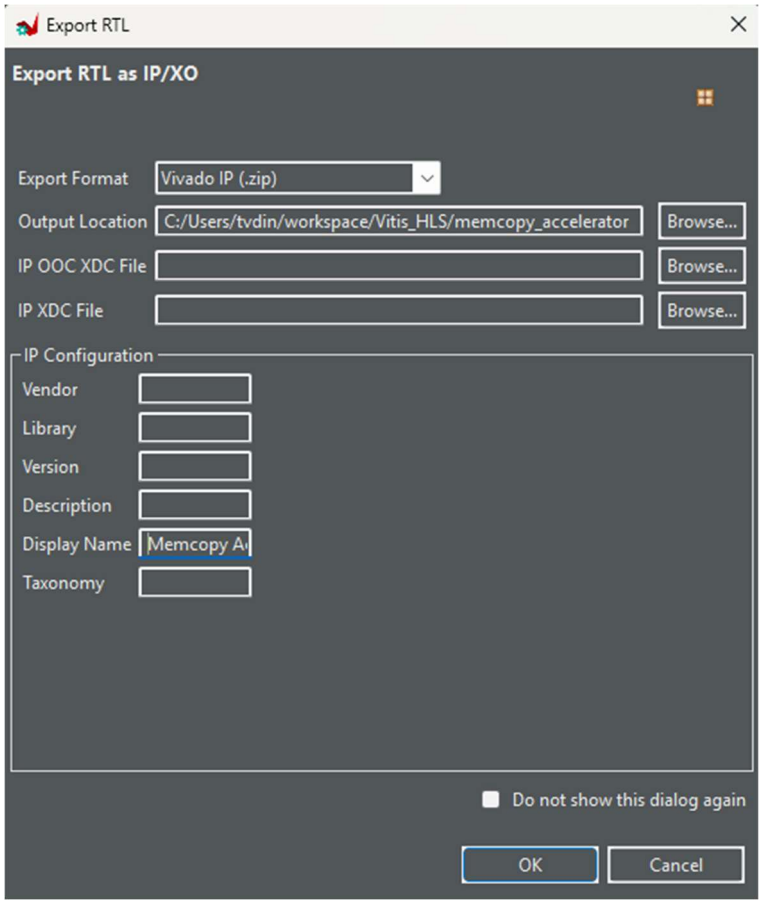
- Please make sure to note the S_AXILITE registers table in the report, as we will use it later.

S_AXILITE Registers

Interface	Register	Offset	Width	Access	Description	Bit Fields
s_axi_CTRL_BUS	CTRL	0x00	32	RW	Control signals	0=AP_START 1=AP_DONE 2=AP_IDLE 3=AP_READY 7=AUTO_RESTART 9=INTERRUPT
s_axi_CTRL_BUS	GIER	0x04	32	RW	Global Interrupt Enable Register	0=Enable
s_axi_CTRL_BUS	IP_IER	0x08	32	RW	IP Interrupt Enable Register	0=CHAN0_INT_EN 1=CHAN1_INT_EN
s_axi_CTRL_BUS	IP_ISR	0x0c	32	RW	IP Interrupt Status Register	0=CHAN0_INT_ST 1=CHAN1_INT_ST
s_axi_CTRL_BUS	src_1	0x10	32	W	Data signal of src	
s_axi_CTRL_BUS	src_2	0x14	32	W	Data signal of src	
s_axi_CTRL_BUS	dst_1	0x1c	32	W	Data signal of dst	
s_axi_CTRL_BUS	dst_2	0x20	32	W	Data signal of dst	
s_axi_CTRL_BUS	len	0x28	32	W	Data signal of len	

7. Export RTL as IP

- Click Solution → Export RTL to open the Export RTL dialog box.
- The dialog box should open as seen in the screen capture below.
  - Choose 'Export Format' as Vivado IP.zip, the outputted ZIP file will contain your RTL IP, which can be imported into Vivado Design Suite. At the Output Location tag eelect the 'Browse' button to choose a location and a name for the outputted 'Vivado IP (.zip)' file.
  - The IP Configuration box is option to add details to your new RTL IP. Change the display name to 'Memcopy Accel', and select 'OK'. This name will appear when you open your IP in Vivado Design Suite.



- In section 5 we will import the generated IP into Vivado.

4.2 Explanation of the Main Point in the HLS Code

This part explains the core logic and HLS optimizations used in the `memcopy_accel` function.

Header and Definitions

```
#include <stdint.h>
#include <hls_stream.h>
#include <ap_int.h>

#define BURST_LEN 32  // number of 32-bit words per burst
```

- `stdint.h`: Provides fixed-width integer types (e.g., `uint32_t`).
- `hls_stream` and `ap_int`: HLS-specific libraries for hardware data types and stream operations.
- `BURST_LEN`: Defines the burst size (number of 32-bit words per transaction) used for efficient AXI transfers.

Function Prototype

```
void memcopy_accel(uint32_t* src, uint32_t* dst,  uint32_t len)
```

- `src` and `dst` are pointers to source and destination DDR memory.
- `len` is the total byte length to copy.

Interface Pragmas

```
#pragma HLS INTERFACE m_axi     port=src offset=slave bundle=AXI_SRC depth=1024
#pragma HLS INTERFACE m_axi     port=dst offset=slave bundle=AXI_DST depth=1024
#pragma HLS INTERFACE s_axilite port=src       bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=dst       bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=len       bundle=CTRL_BUS
#pragma HLS INTERFACE s_axilite port=return    bundle=CTRL_BUS
```

- Defines **AXI4 Master** interfaces for `src` and `dst` to read/write DDR.
- Defines **AXI4-Lite** control interface for the processor to set parameters and control execution.
- These pragmas automatically generate proper AXI buses during synthesis.

Main Variables

```
uint32_t num_words = len >> 2;  // divide by 4
uint32_t buffer[BURST_LEN];
#pragma HLS ARRAY_PARTITION variable=buffer complete dim=1
```

- Converts `len` in bytes to `num_words` (32-bit words).
- Creates an on-chip buffer for burst data.
- `ARRAY_PARTITION complete` allows parallel access to each buffer element during loop unrolling.

Loop Structure

1. *Copy Loop*

```
for (uint32_t i = 0; i < num_words; i += BURST_LEN)
```

- Processes data in chunks of `BURST_LEN` words.
- `#pragma HLS PIPELINE II=1` allows continuous initiation every cycle.

2. *Branchless Chunk Computation*

```
uint32_t diff = num_words - i;
uint32_t mask = (diff >= BURST_LEN); // 1 if full burst, 0 otherwise
mask = -mask; // 0xFFFFFFFF if true, 0x0 if false
uint32_t chunk = (mask & BURST_LEN) | (~mask & diff);
```

- This piece of code is equivalent to
  `uint32_t chunk = (i + BURST_LEN <= num_words) ? BURST_LEN : (num_words - i);`
  The purpose is to Eliminates the conditional branch, which is not hardware-friendly should be avoid in the HLS design, to determine the last burst size. It is HLS-safe synthesis technique. Ensures synthesis produces predictable hardware with no conditional paths.

3. *Read Loop*

```
    read_loop:
        for (uint32_t j = 0; j < BURST_LEN; j++) {
#pragma HLS UNROLL
            bool valid = (j < chunk);
            uint32_t vmask = -((uint32_t)valid);
            buffer[j] = src[i + j] & vmask;  // only valid data copied
        }
```

- Fully unrolled to create `BURST_LEN` parallel reads.
- Uses a **masking technique** (`vmask`) to avoid conditional `if` statements.
- Invalid indices automatically zeroed out.

4. *Write Loop*

```
    write_loop:
        for (uint32_t j = 0; j < BURST_LEN; j++) {
#pragma HLS UNROLL
            bool valid = (j < chunk);
            uint32_t vmask = -((uint32_t)valid);
            dst[i + j] = buffer[j] & vmask;  // only valid data written
        }
```

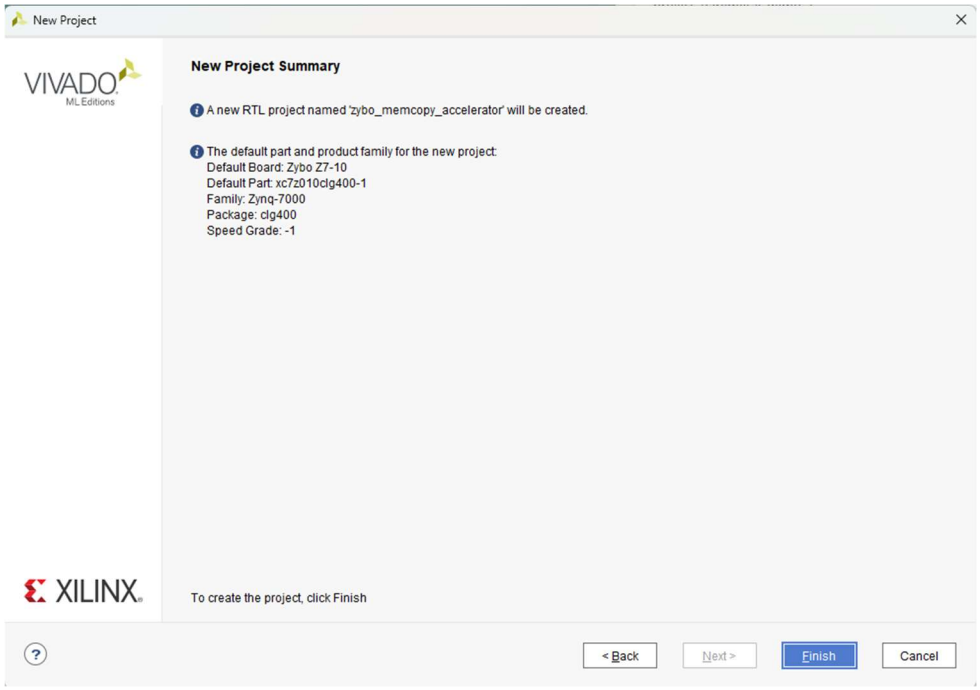
- Similar to the read loop but writes from the buffer to destination DDR.
- Both read and write loops run in parallel during pipelining for higher throughput.

5. Vivado Integration

Reference: Digilent – Getting Started with IPI:
https://digilent.com/reference/programmable-logic/guides/getting-started-with-ipi

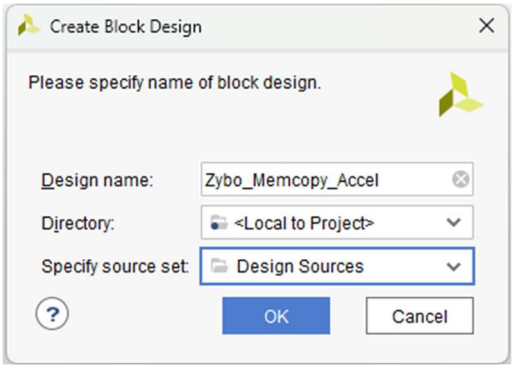
1. Create a New Vivado Project

- *File → New Project →* **Next** → Name your project (e.g., `zybo_memcopy_accelerator`).
- Choose **RTL Project** (Do not specify sources at this time).
- Select **Board**: *Zybo Z7-10* (requires Digilent board files) **or Part**: `xc7z010clg400-1`



2. Create an IPI Block Design

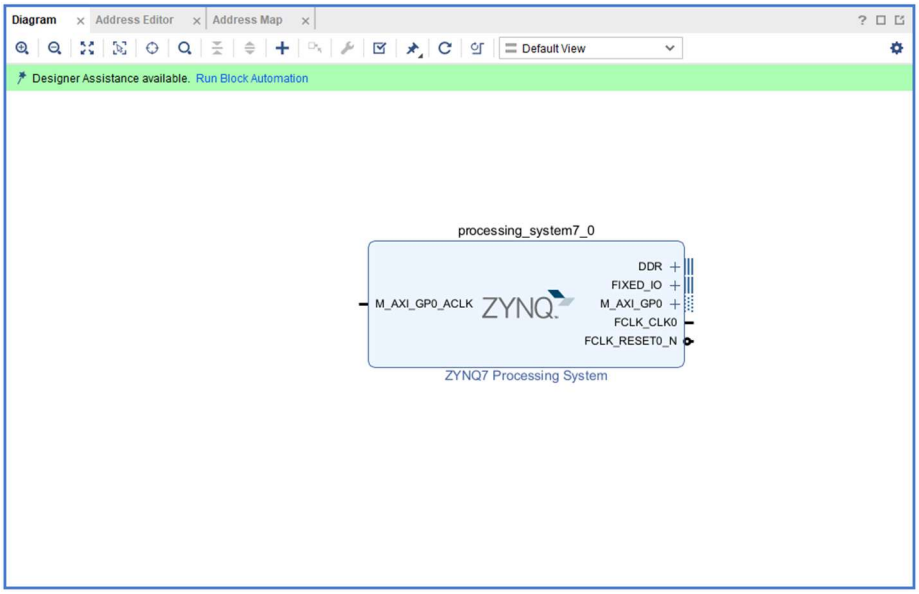
- *Flow Navigator* → *IP Integrator* → *Create Block Design* → name: `Zybo_Memcopy_Accel`.



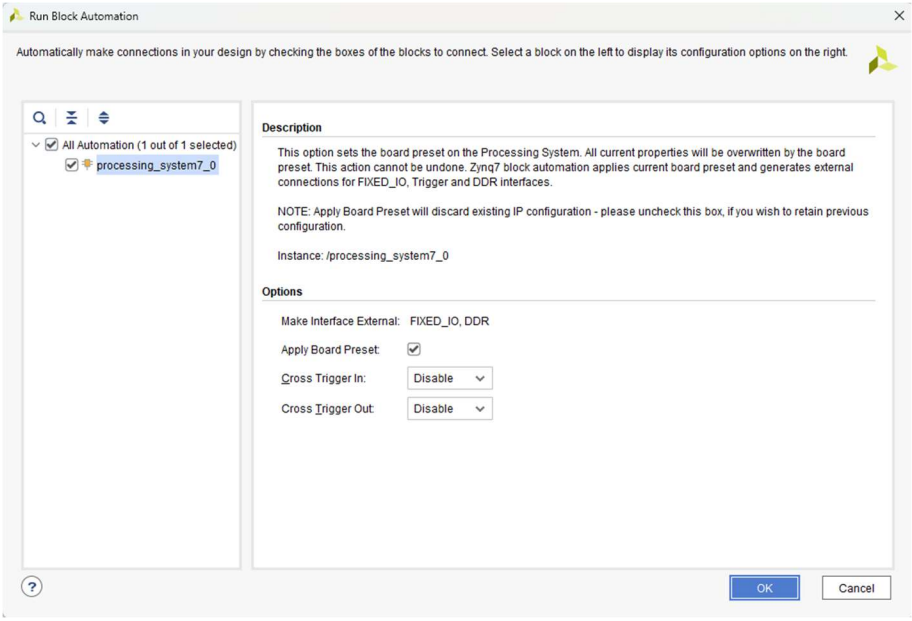
3. Add the Zynq Processing System (PS)

- Click **Add IP** button → Search **ZYNQ7 Processing System** → Add.

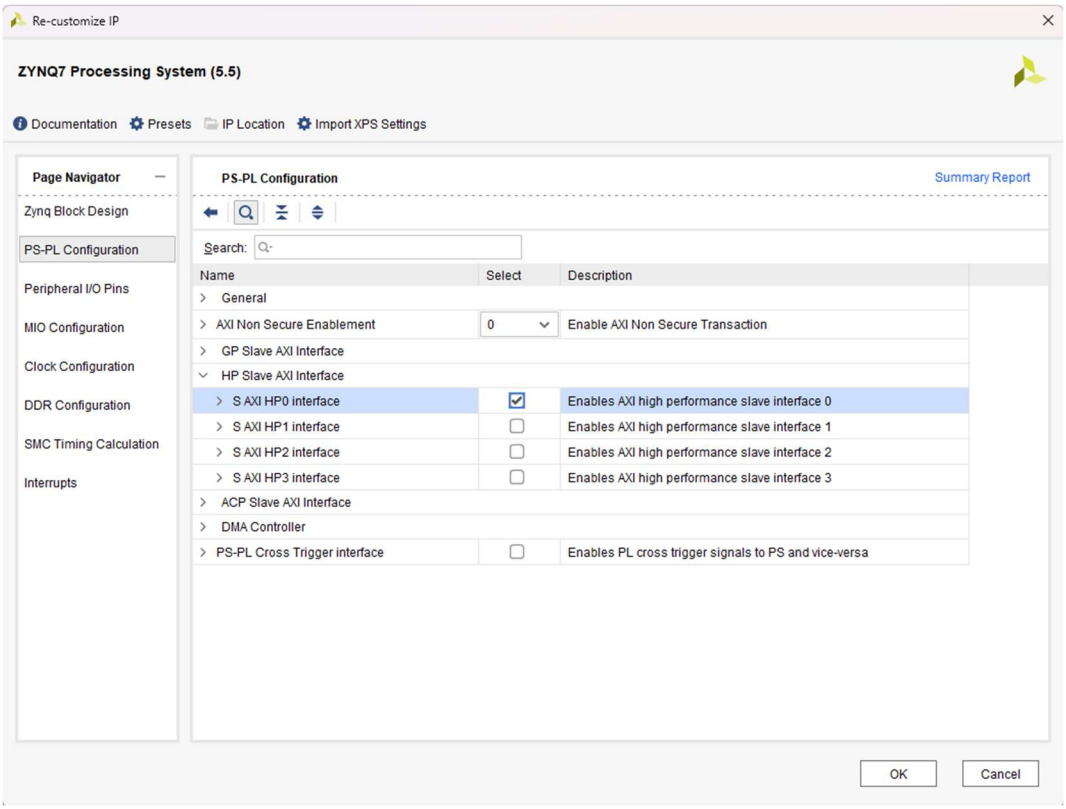
- Click **Run Block Automation** in the Design Assistance banner (the green bar).



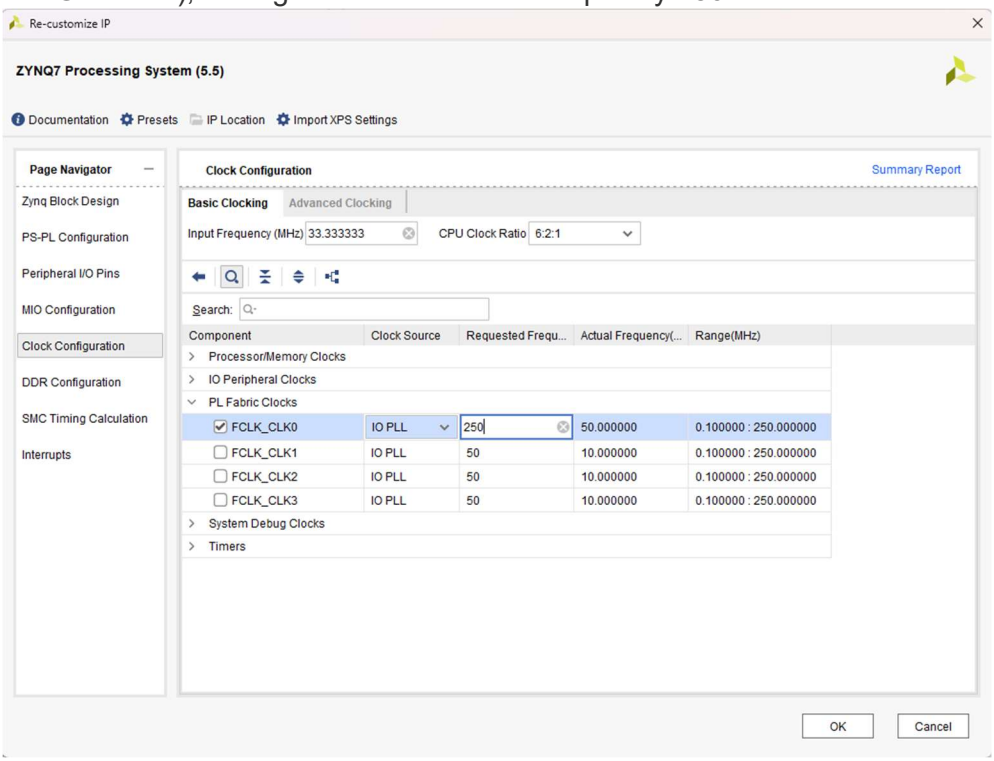
- In the dialog that pops up, leave all settings as defaults. *Apply Board Preset* should be checked.



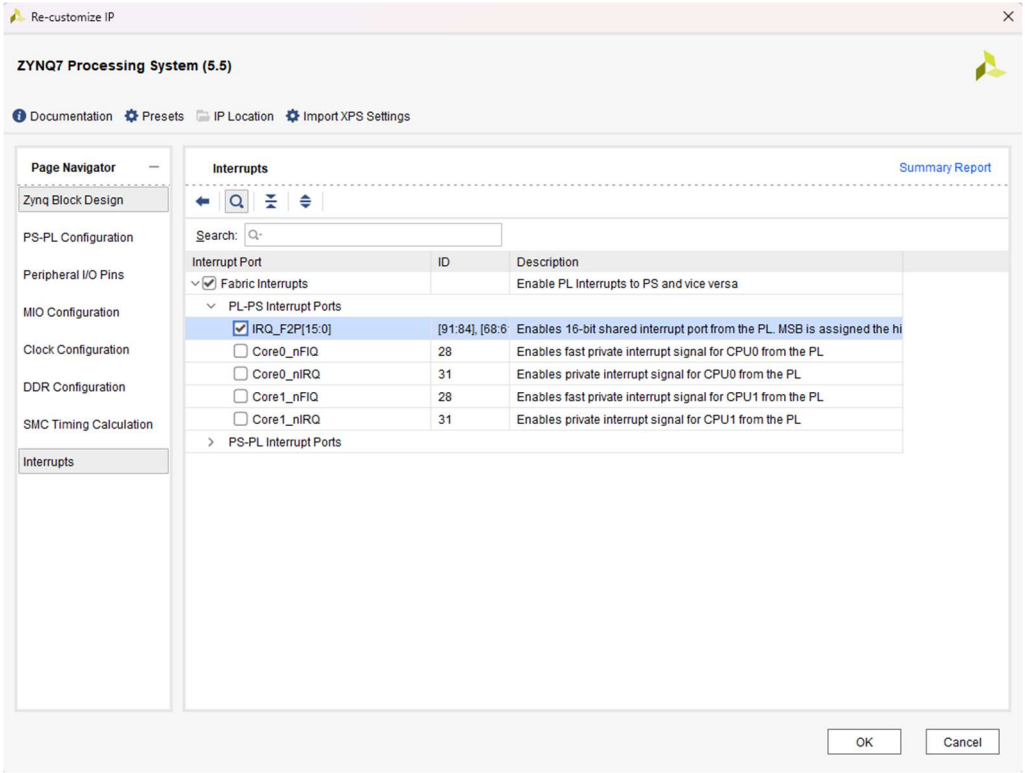
- The needs of the project require that change some of the default settings of the Zynq PS. To edit its settings, double click on it to open the configuration wizard. Make changes as in the pictures below:
  - AXI HP ports (for DDR access)



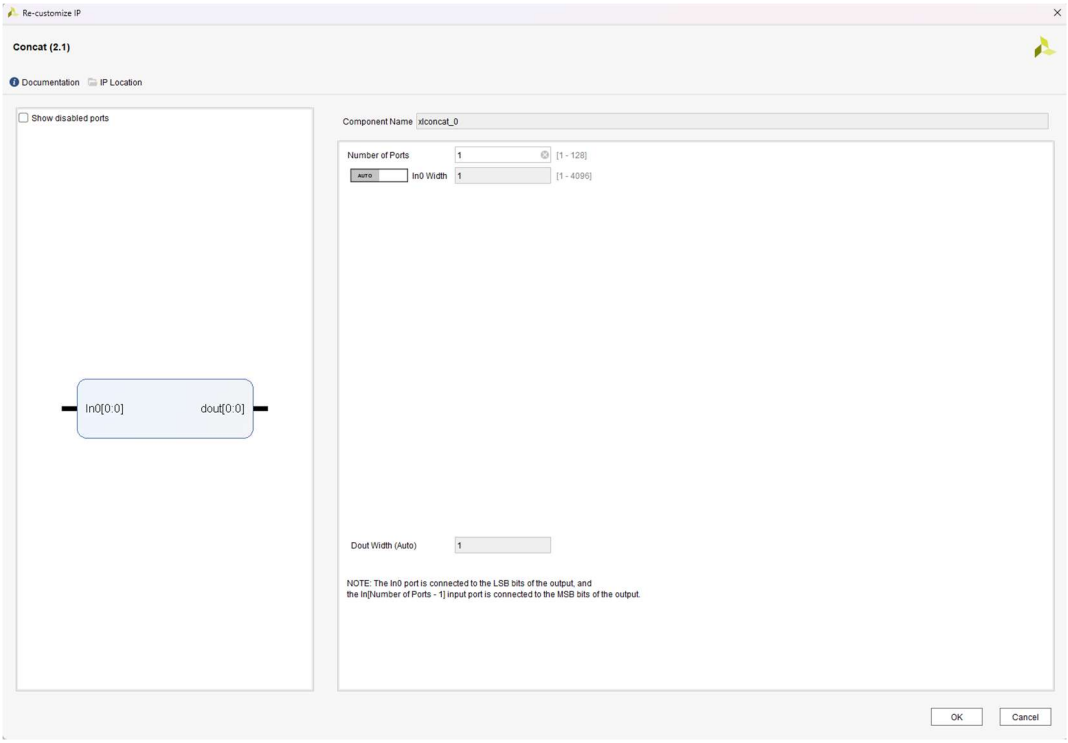
- FCLKs (The Zynq PS can generate multiple clocks that are then provided to the FPGA fabric), setting at maximum clock frequency 250MHz.



- IRQ_F2P port (Zynq devices can also use interrupts generated in FPGA fabric to trigger interrupts within the Processing System).



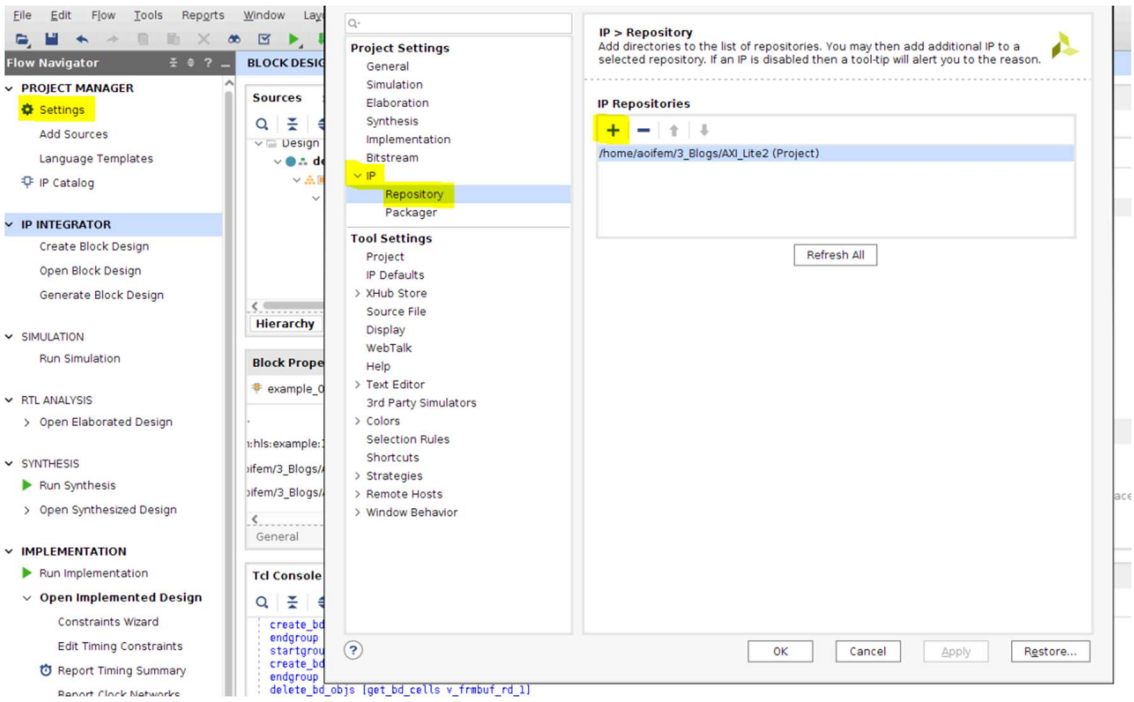
- Since the IRQ_F2P port is a vectored interface, the typically single-bit interrupts signal from peripherals will need to be vectorized in order to be connected.This is performed



using the IP Integrator Concat IP block. Add the Concat IP block to the block diagram and configure the number of desired interrupt inputs. [ref]

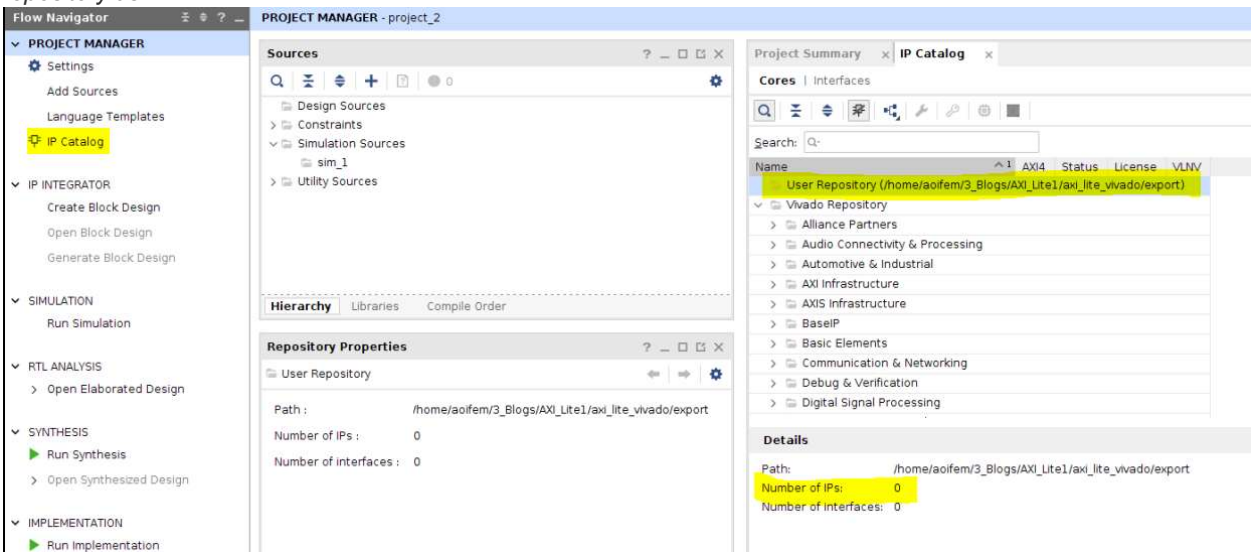
4. Add the HLS IP (memcopy_accel)

- To use the generated IP in a Vivado project, we first have to add your new IP repository to the Vivado project. This is the folder that contains the exported .zip file from Vitis HLS. To do this,

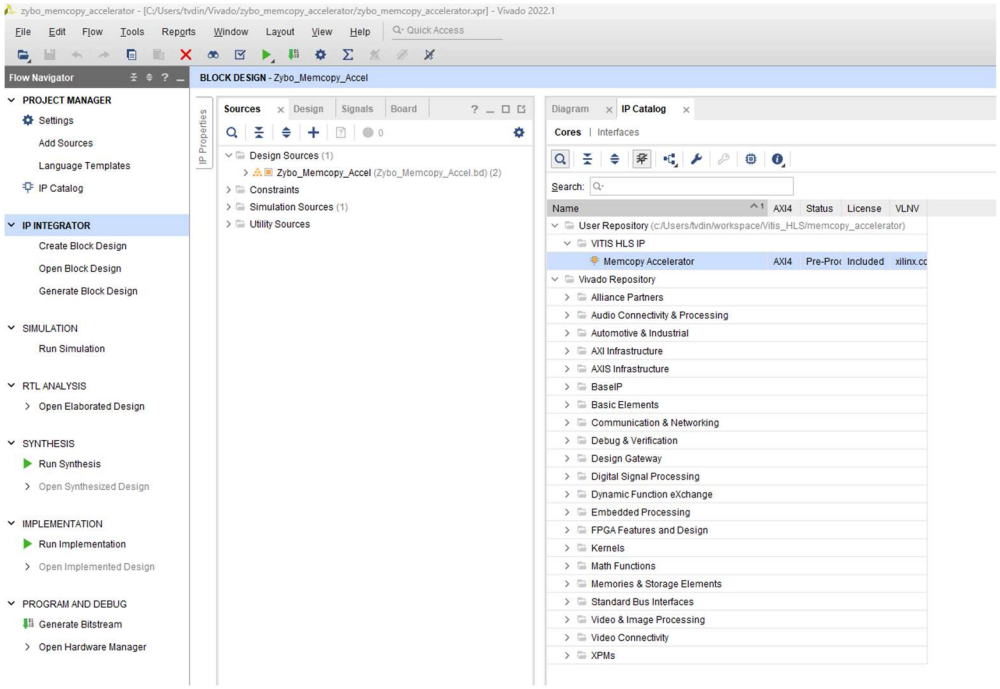


select Settings → IP → Repository. Select the + button and select the location where you exported The Memcopy Accelerator IP in Vitis HLS (the one generated in section 4).

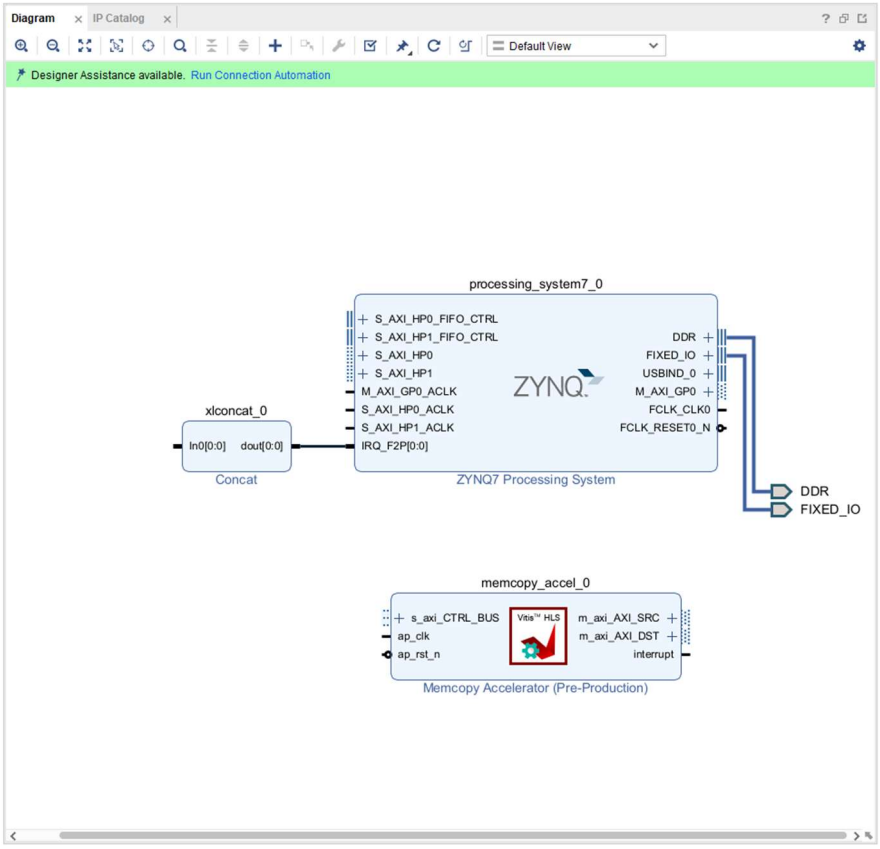
- If the IP was imported successfully, the number of IPs listed in the details window should now be '1'.
  Note: If there are 0 IPs listed, you can right click on your new repository and select 'Add IP to Repository'. Select the ZIP file you exported from Vitis HLS. You should then see the number of IPs in that repository as '1'



- From **IP Catalog**, under User Repository add Memcopy Accelerator to the diagram.

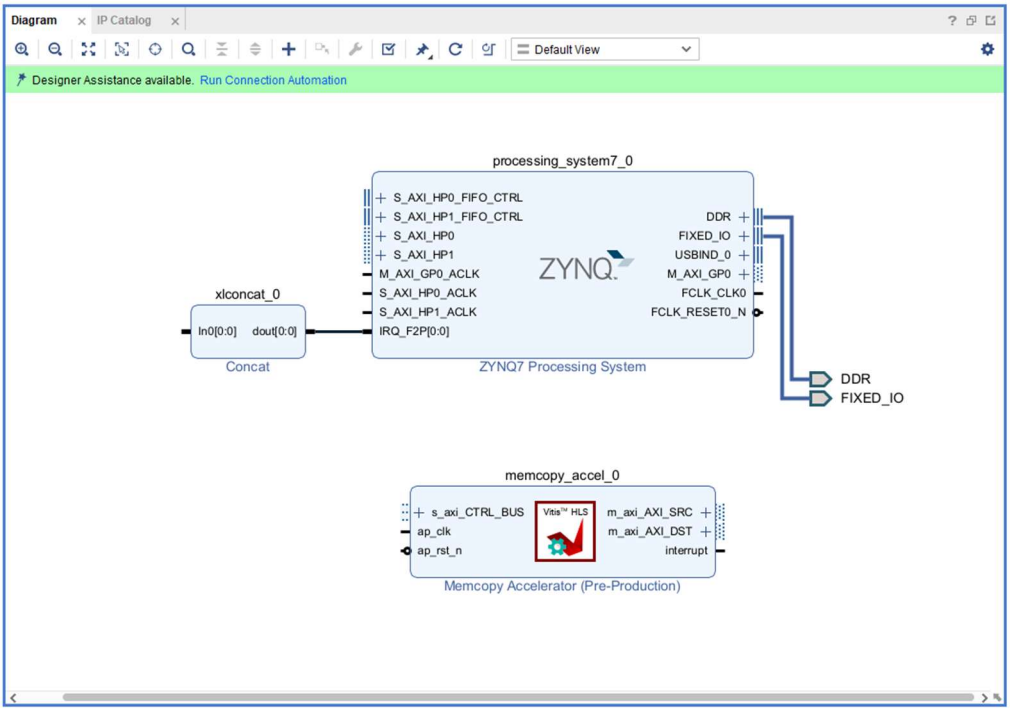


- Now the Vivado Diagram should look like this

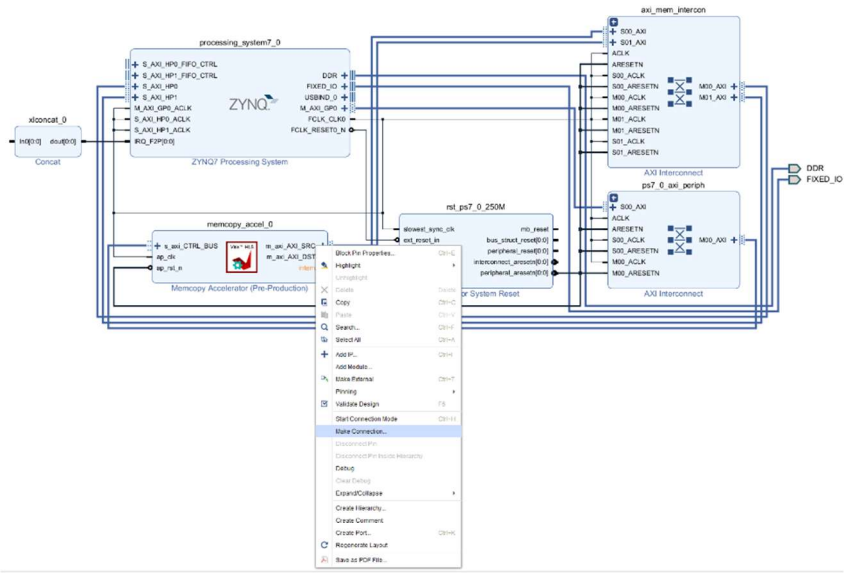


5. Make Connections

- Click the **Run Connection Automation** button in the green *Designer Assistance* bar.



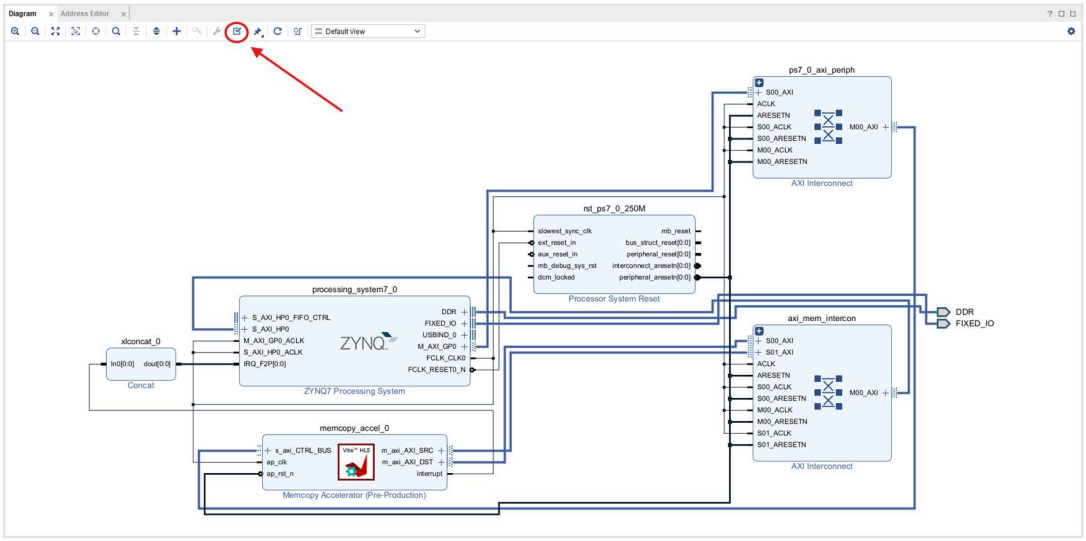
- In the dialog that pops up, check all boxes → Click OK to run connection automation and connect the PL accelerator block to the PS Arm processor. *Note that repeat the Connection Automation step until it disappears*.
- Next, right click on the 'interrupt' port of the Accelerator



- In the dialog that pops up, check the xlconcat_0 box (connecting PL-PS interrupt)

6. Validate a Block Design

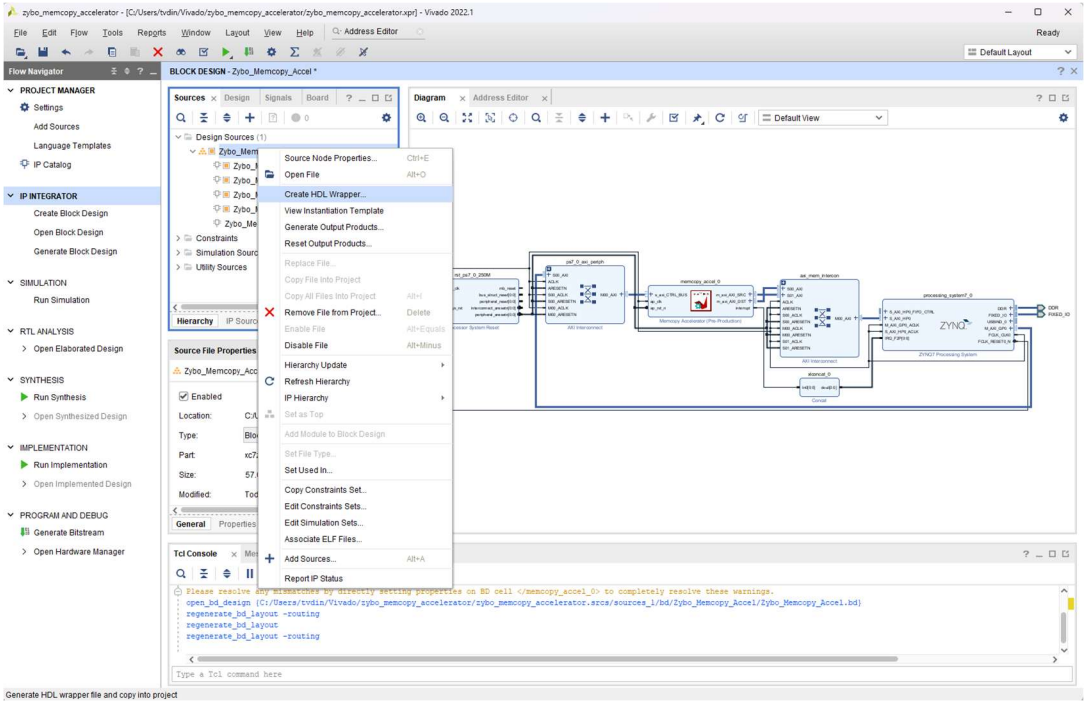
- Before the Vivado project can be built, the block design must be validated. Click the **Validate Design** button ( ) in the Diagram pane's toolbar (or press the F6 key).



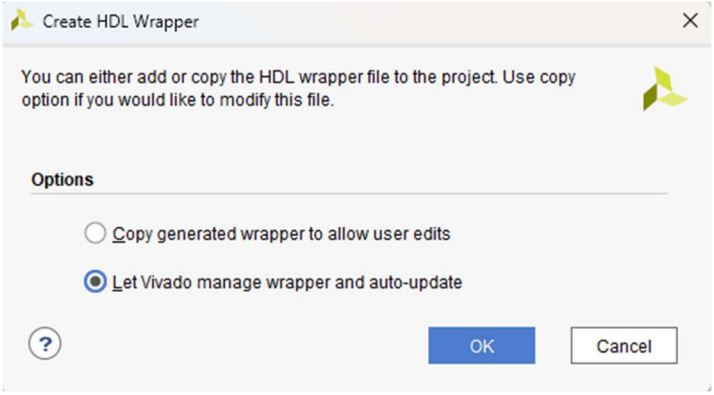
- **Note:** *Some Zynq boards may produce critical warnings at this stage relating to PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY parameters. These warnings are ignorable and will not affect the functionality of the project*

7. Create an HDL Wrapper

- An HDL wrapper must be created for the block design. This process translates the block design into a source file that can be read by the Vivado tools, and is used to build the actual design.
- Open the *Sources* pane and locate the block design file (.bd) under the *Design Sources* dropdown. Right click on it and select **Create HDL Wrapper**.

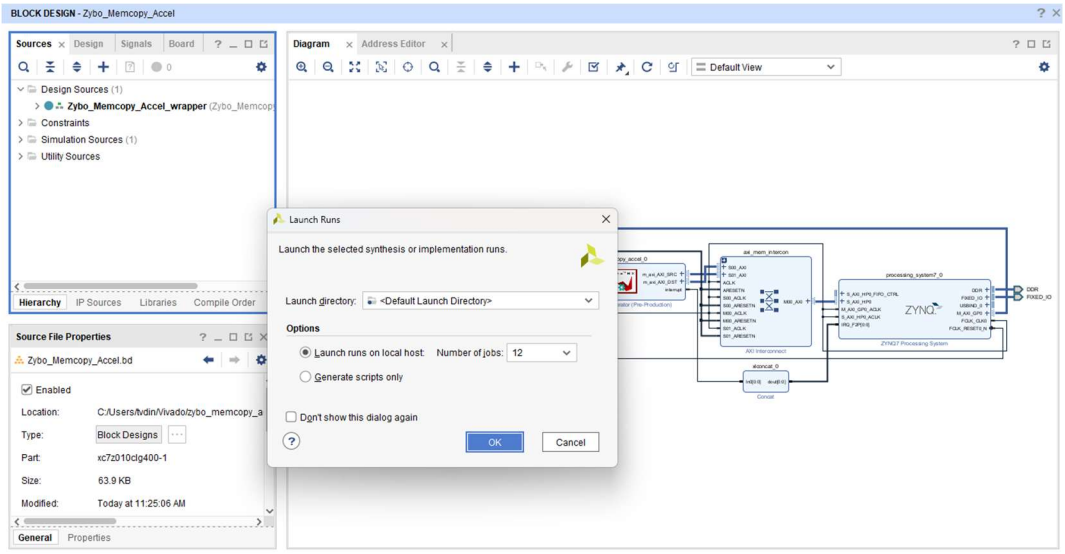


- In the dialog that pops up, choose *Let Vivado manage wrapper and auto-update* is recommended.



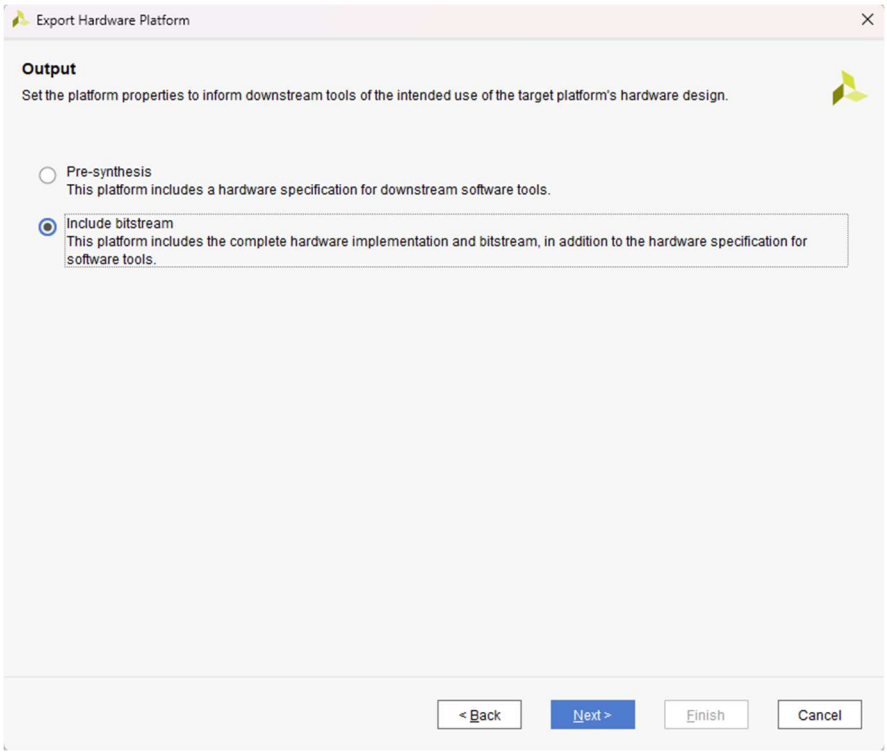
8. Build a Vivado Project

- At this point, the Vivado Project is ready to be built, by running it through Synthesis and Implementation, and finally generating a bitstream.
- Run Synthesis → Run Implementation → Generate Bitstream

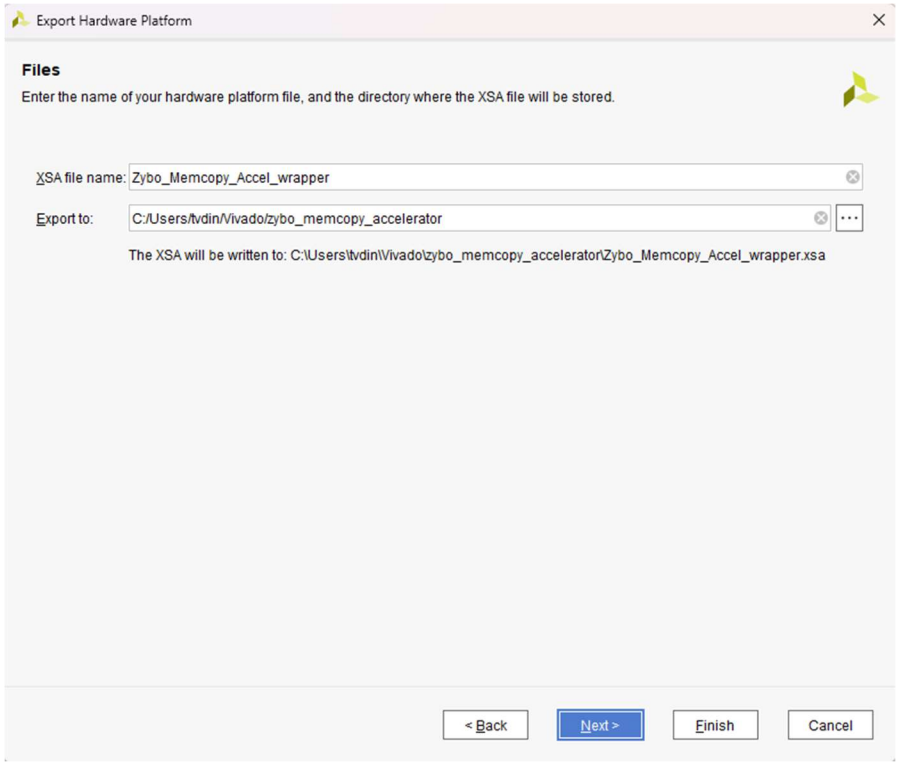


9. Export a Fixed Post-Synthesis Hardware Platform

- Once the project has been built, the design must be exported from Vivado so that Vitis has access to information about the hardware that a software application is being developed for. This includes the set of IP connected to the processor, their drivers, their addresses, and more. Exporting hardware after the bitstream has been generated allows us to program the ZyboZ7 board directly from within Vitis.
- *File → Export → Export Hardware* → Choose the option **Include bitstream**



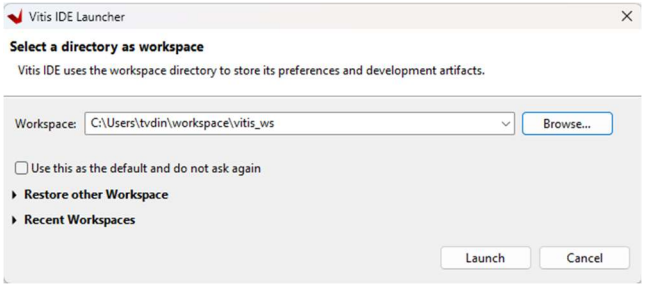
- The Files screen gives the option to choose a name for the Xilinx Shell Architecture (XSA) file, and provide a path to a folder that the file will be placed within. Give the XSA file a name, and choose a memorable location to place it in. This file will later be imported into Vitis, so take a note of where it is placed and what it is called.



6. Vitis Bare-Metal Application

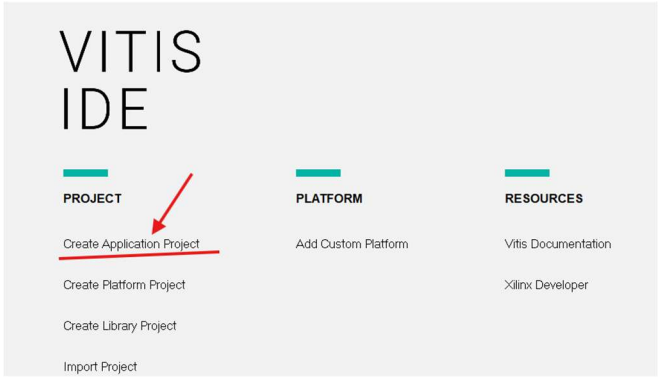
1. Launch Vitis IDE

- Create a new workspace, e.g., `vitis_ws`.

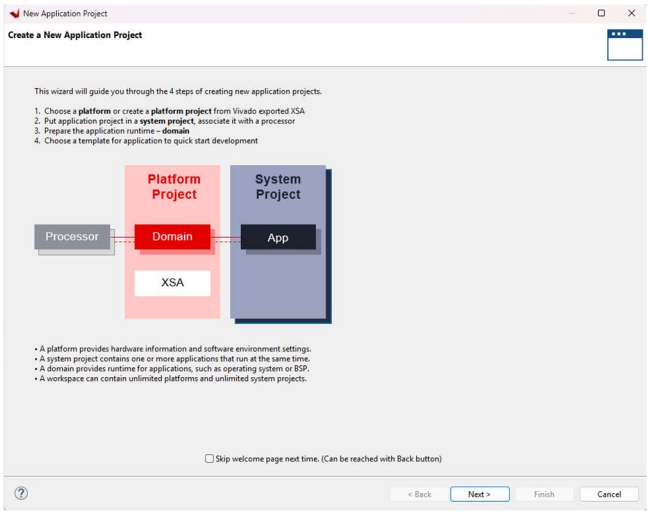


2. Create Application Project

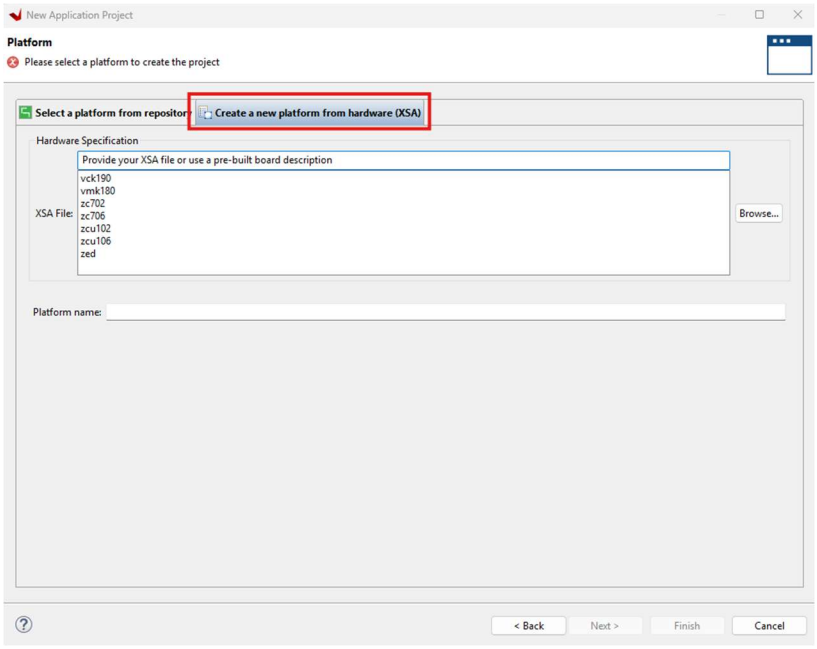
- On Vitis IDE' welcome screen, click Create Application Project.



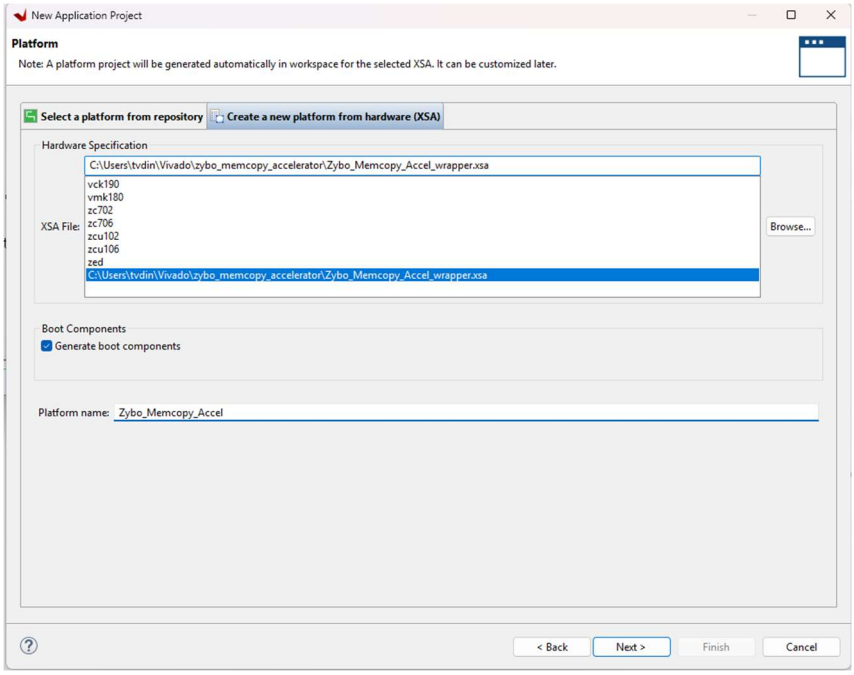
- The first screen of the wizard is a welcome page, which summarizes what each of the components of a software design are. Click Next to continue.



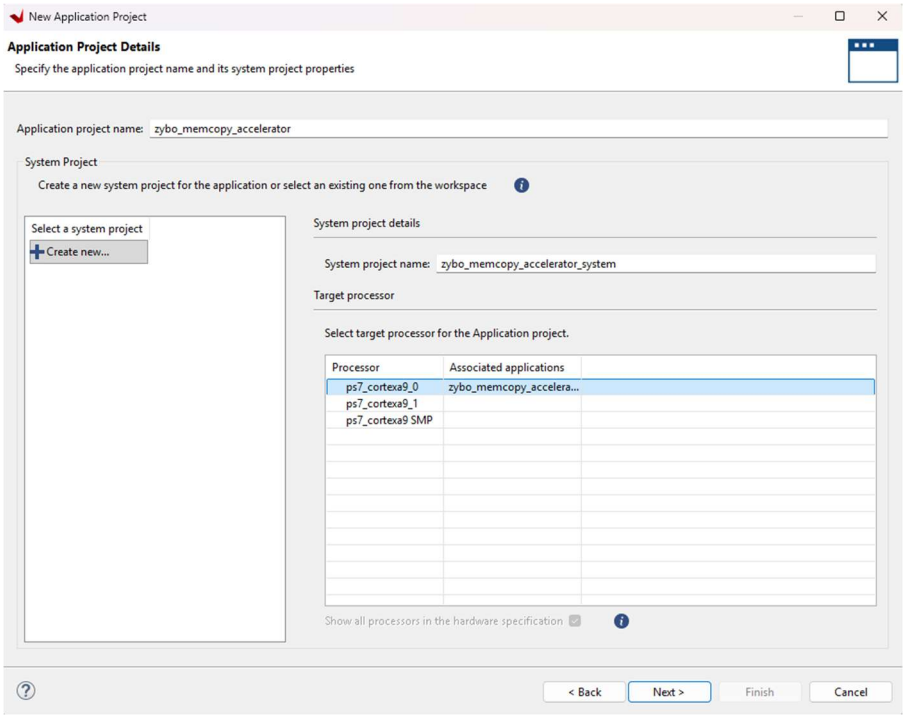
- Next, the platform that the application targets must be created. Open the **Create a new platform…** tab.



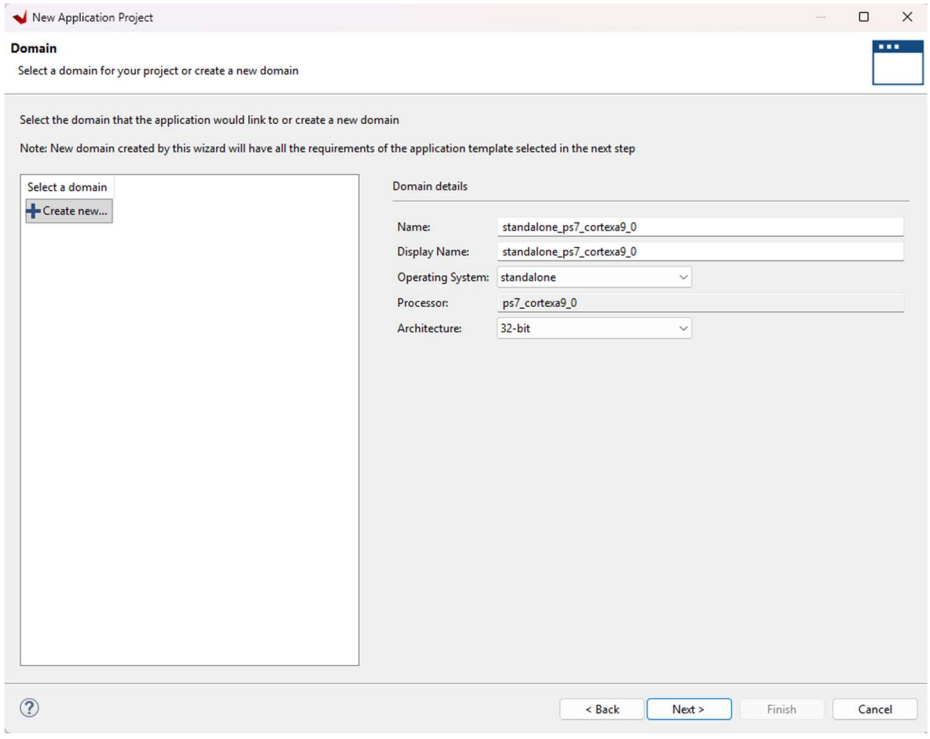
- **Browse** your file system to find the Xilinx Shell Architecture (XSA file) previously exported from Vivado (*change the name if you do not like the default).* Click **Next** to continue.



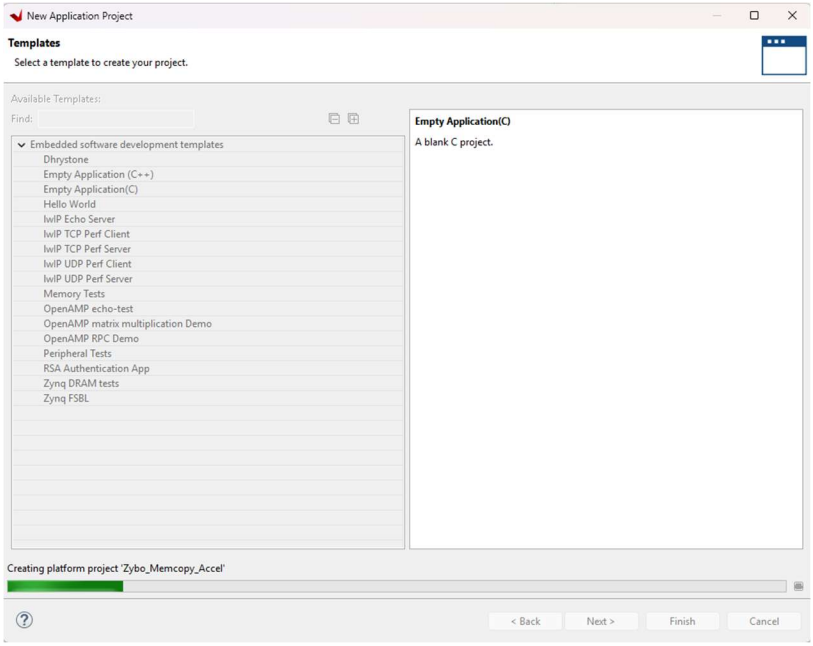
- The next screen is used to set some options for the application project and the system project. The names of both projects can be set, as well as which processor core will be used to run the application. All settings can be left as defaults. Click **Next** to continue.



- Next, the domain that the application project operates in will be defined. In this case all default settings will be used. Click **Next** to continue.

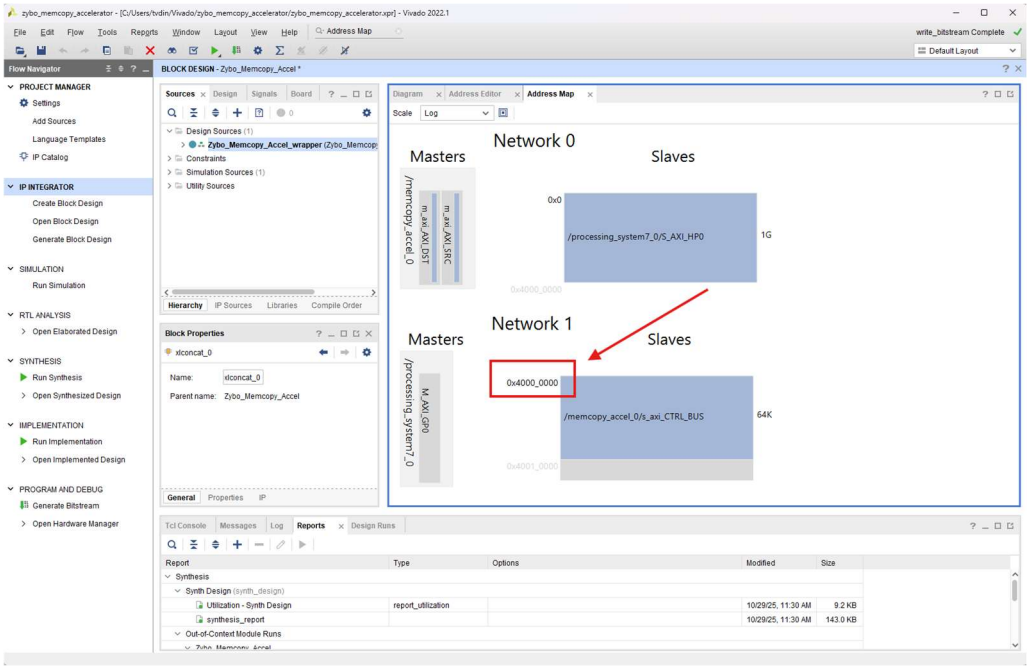


- Next, choose Empty Application (C/C++). The source files will be added later. Click **Finish** to finish creating the project.



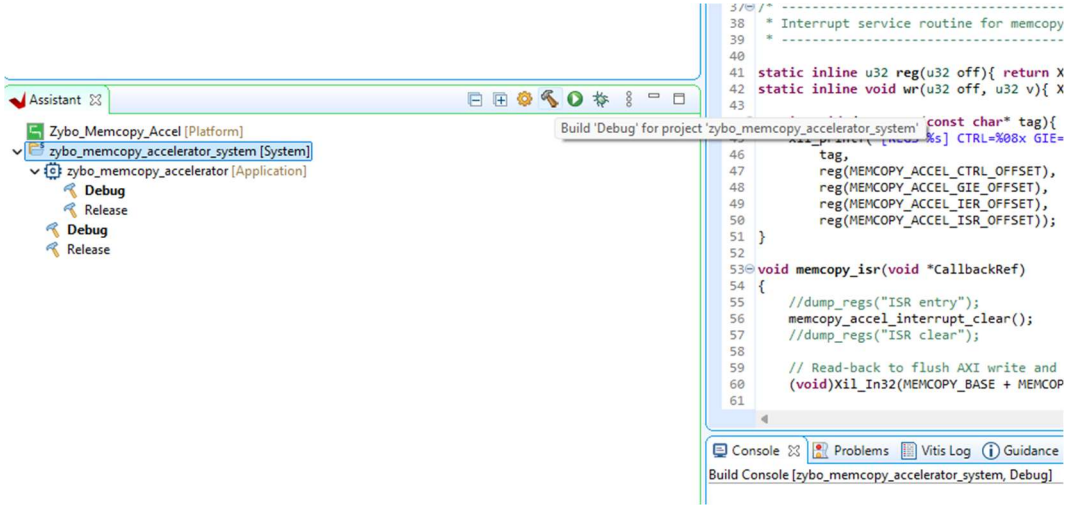
3. Add Source Files

- In the app project `src/`, add `main.c`, `memcopy_accel.c`, `memcopy_accel.h` from the clonned folder `src/ Vitis-BareMetal.`
- Ensure the base address macro `MEMCOPY_ACCEL_BASEADDR` in the header file `memcopy_accel.h` matches with the IP address in Vivado.



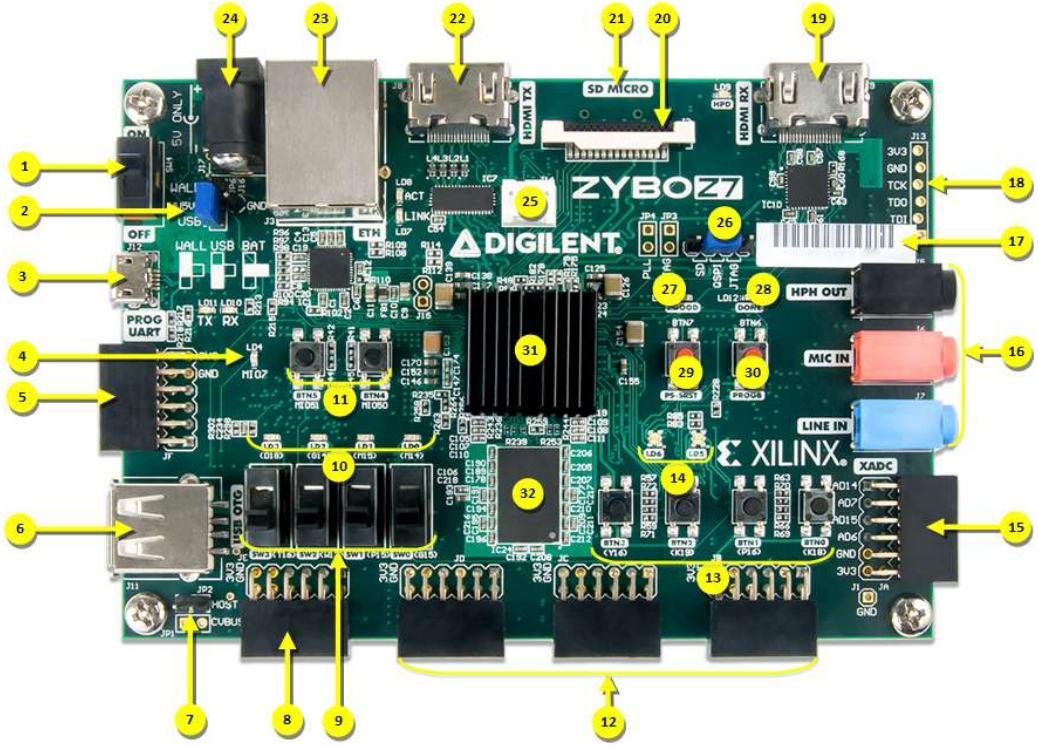
4. Build a Vitis Application

- To build the project and all of its dependencies, select the **[System]** project in the *Assistant* pane, and either click the **Build** button , or press Ctrl-B on your keyboard.



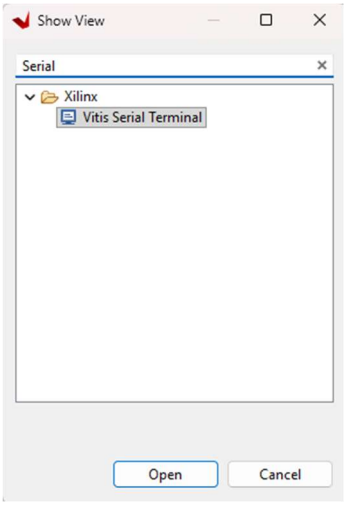
5. Launch a Vitis Baremetal Software Application

- Make sure the board is set to boot from JTAG before it's powered on (check No.26 Programming mode select jumper)

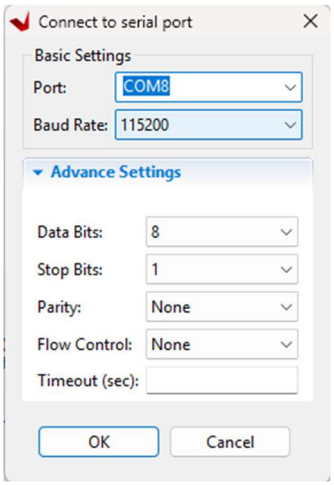


- The applications require that a serial console is connected to the board, so that standard output (from print statements) can be viewed. For this purpose, a serial terminal should be used. Use a serial terminal application to connect to the board's serial port. Vitis IDE has a built in serial terminal or you can use Tera Term or Puty whatever you prefer. Zynq designs use a baud rate of 115200.

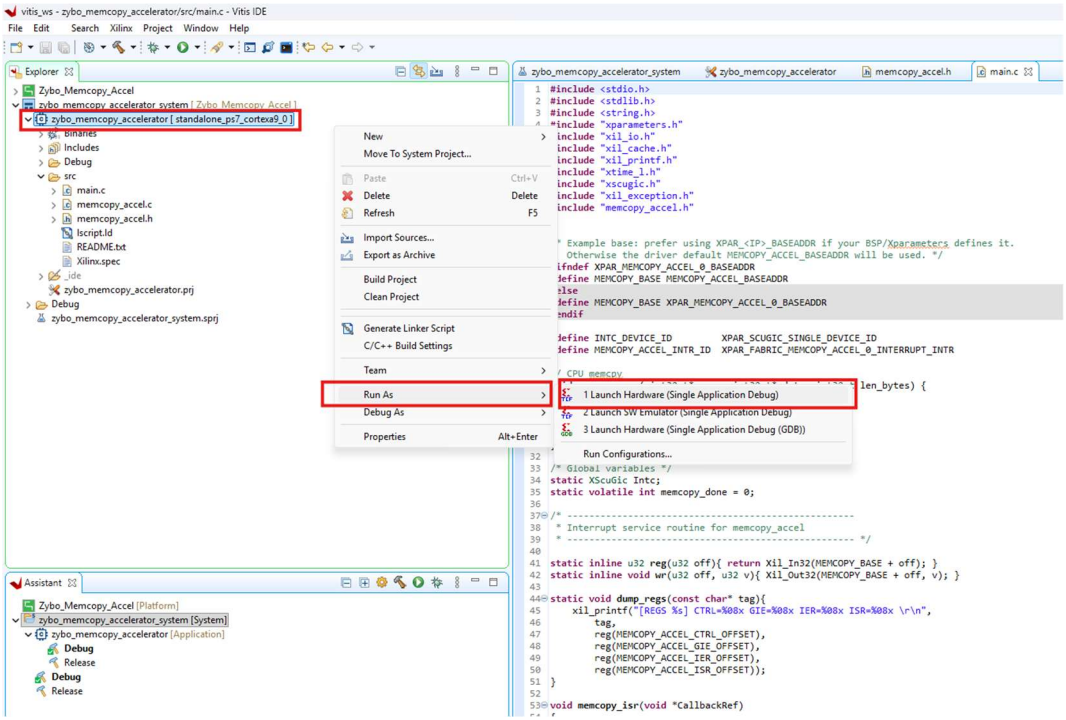
- To open Vitis Serial Terminal, in the toolbar *Window* → *Show View* then searching for *Serial Terminal*



- Click the + Button on the Vitis Serial Terminal section an select the right COM Port and set baud rate.



- In the *Explorer* pane at the left side of the screen, right click on the application or system project that is to be run, and select *Run as → 1 Launch on Hardware (Single Application Debug)*. The FPGA will be programmed with the bitstream, the ELF file created by the software build is loaded into system memory, and the application project will begin to run.



- Check the Vitis Serial Terminal for the result.