Departamento de Engenharia Eletrotécnica e de Computadores
Instituto Superior Técnico
Universidade de Lisboa

Hardware/Software Co-Design

Laboratory Guide

José T. de Sousa

April 2025

Contents

1 Introduction **4**

2 Design an AXI-Lite IP Using HLS **4**
- 2.1 Create a New HLS Component . . . 5
- 2.2 Verify the C-Based Hardware Specification (C Simulation) . . . 7
- 2.3 Synthesize the Design . . . 9
- 2.4 C/RTL Co-Simulation . . . 9
- 2.5 Package the IP Component . . . 10

3 Zynq Project in Vivado **10**
- 3.1 Create a New Project targeting the FPGA board . . . 10
  - 3.1.1 Open Vivado . . . 10
  - 3.1.2 Create a New Project . . . 10
  - 3.1.3 Name the project . . . 10
  - 3.1.4 Select project type . . . 12
  - 3.1.5 Skip Add Sources and Constraints . . . 12
  - 3.1.6 Board Selection . . . 12
- 3.2 Use IP Integrator to create a new Block Design, with the ZYNQ processing system . . . 12
  - 3.2.1 Create a Block Design . . . 12
  - 3.2.2 Add the Processing System . . . 12
  - 3.2.3 Configure the processing block with only one UART peripheral . . . 14
  - 3.2.4 PS-PL Configuration . . . 14
  - 3.2.5 Clock Configuration . . . 14
  - 3.2.6 Finalize the PS Configuration . . . 17
- 3.3 Specify IP Repository . . . 17
- 3.4 Add your IP to the Zynq design . . . 17
  - 3.4.1 Add New AXI-Lite IP . . . 17
  - 3.4.2 Connect the blocks . . . 17
- 3.5 System Hardware Generation . . . 21
  - 3.5.1 Generate Block Design . . . 21
  - 3.5.2 Create HDL Wrapper . . . 21
  - 3.5.3 In the Sources panel, right-click on *design_1.bd* and select Create HDL Wrapper to generate the top-level VHDL model. Leave the "*Let Vivado manager wrapper and auto-update*" option selected. . . . 21
  - 3.5.4 Design Implementation . . . 22
  - 3.5.5 Export Hardware to the Vitis Software Platform . . . 22

4 Software Application **24**
- 4.1 Create your C Project . . . 24

5 Verify the Design in Hardware **26**
- 5.1 Connect the board and establish serial communication from Vitis's Terminal 26
- 5.2 Debug your software using the Application Debugger . . . 28

6 Memory Initialization from a Binary File **28**

7 Measure the execution time of your application **29**

8 Course Project **30**
8.1 Data input/output 30
8.2 Project schedule 30

References **31**

List of Figures

1 Vitis main window — Create Empty HLS Component 5
2 Choose the component name and location 6
3 Source Files tab — adding design and testbench files and selecting the top function 6
4 Hardware tab — selecting the target FPGA part 7
5 Settings tab — HLS synthesis configuration 8
6 C simulation output — dot-product IP result matches the software reference 8
7 Running HLS synthesis 9
8 Open Vivado — without closing Vitis 11
9 Open Vivado — Create a New Project 11
10 Open Vivado — Name the project 12
11 Select project type — path and in the filenames 13
12 Create a Block Design — with the design name design_1 14
13 Create a Block Design — System and add it to the design 15
14 Create a Block Design — Click on Run Block Automation with the default settings by pressing OK 16
15 Create a Block Design — applied, which can be customized 16
16 PS-PL Configuration — option (General → Enable Clock Resets) selected 17
17 Clock Configuration — GP0, clock, and reset ports: 18
18 Specify IP Repository — Add your IP repository folder to the IP repository list in the 18
19 Add New AXI-Lite IP — to the design 19
20 Connect the blocks — connections: 19
21 Connect the blocks — Interconnect, have been automatically added to the design 20
22 Connect the blocks — Note: the mapping addresses generated by 20
23 Create HDL Wrapper — right-click design_1.bd and select Create HDL Wrapper 21
24 Create HDL Wrapper — to see the content in the Auxiliary pane.) 22
25 Design Implementation — Utilization Report and the Timing Summary Report 22
26 Design Implementation — Utilization Report and the Timing Summary Report 23
27 Design Implementation — the export path corresponds to your project folder 23
28 Design Implementation — the export path corresponds to your project folder 24
29 Software Application — choose a suitable location. Click Next 25
30 Software Application — (design_1_wrapper.xsa) and select it. Click Next 25
31 Create your C Project — memory space (starting with base address 0x00000000) 27
32 Memory Initialization — editing ps7_init.tcl with binary file, memory address and file size 28

1 Introduction

This lab guide supports the Hardware/Software Co-Design course at Instituto Superior Técnico. It covers three laboratory assignments that progressively build a complete embedded HW/SW system on a Xilinx Zynq-7000 SoC development board.

Lab 0 A short tutorial on the Xilinx Vitis and Vivado design tools and the development board. Students become familiar with the HLS and FPGA design flows before tackling the main project.

Lab 1 A HW/SW co-processing architecture based on a uniprocessor Zynq system with a hardware accelerator designed using HLS. The accelerator is connected to the ARM processor via an AXI-Lite interface. The target application is a 2D image convolution kernel, representative of the convolutional layers used in neural networks [1].

Lab 2 An embedded heterogeneous multiprocessor system on the Zynq PL fabric, with one or more hardware accelerators. Students must demonstrate parallelization and measure the resulting speedup.

The course project (Section 8) targets Convolutional Neural Network (CNN) image classification using the STL-10 dataset. Students implement and optimize a 2D convolution engine as an HLS-based AXI-Lite IP, integrate it into a Zynq block design in Vivado, develop the bare-metal software application in Vitis, and measure execution time to evaluate their hardware acceleration.

This guide walks through each step of the design flow: designing the AXI-Lite IP with Vitis HLS (Section 2), building the Zynq block design in Vivado (Section 3), writing and running the software application (Section 4), verifying the design on hardware (Section 5), loading data from a binary file into memory (Section 6), and measuring execution time (Section 7).

Recommended supporting material for this course is listed in the References section [2, 3, 4, 1, 5].

Note: some screenshots in this guide may differ slightly from what you see on screen due to tool updates. Please consult the course website for the latest version of this guide and the associated project files. Xilinx/AMD tools can occasionally be unstable; do not hesitate to ask for help.

2 Design an AXI-Lite IP Using HLS

In this section, you will use the Xilinx Vitis High-Level Synthesis (HLS) tool to design a hardware accelerator and package it as an AXI-Lite IP block. The accelerator implements a multiply-accumulate (MAC) operation on two vectors, i.e., the dot product, which serves as a simple but representative kernel for hardware acceleration.

AXI-Lite is a lightweight member of the AMBA AXI bus family. It provides a simple register-mapped interface between a processor and a hardware accelerator, making it straightforward to control the accelerator from software. In this lab, the Zynq Processing System (PS) will communicate with the dot-product accelerator through an AXI-Lite interface.

Vitis HLS raises the abstraction level by allowing hardware to be described in C/C++.

The tool synthesizes the C description into RTL (Register Transfer Level) HDL, estimates performance and resource utilization, and packages the result as a reusable IP block that can be instantiated in a Vivado block design.

2.1 Create a New HLS Component

Follow the steps below to create a new HLS component targeting the FPGA on your board.

1. Launch Vitis from the desktop shortcut or from the shell. On the welcome screen, select **HLS Development** → **Create Component** → **Create Empty HLS Component**.

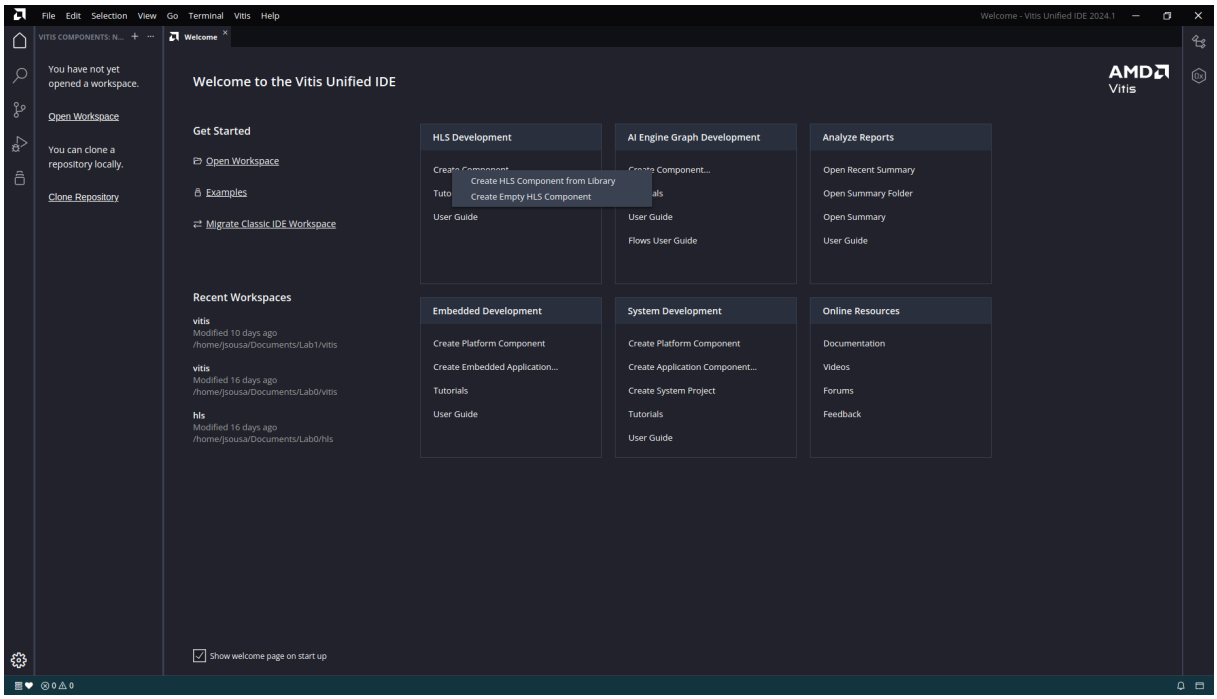


Figure 1: Vitis main window — Create Empty HLS Component

2. Enter a component name (e.g., `axil_macc`) and choose a suitable location on the filesystem. Click **Next**.

3. In the **Configuration File** tab, select **Empty File**, leave the default configuration filename unchanged, and click **Next**.

4. In the **Source Files** tab, add the provided source files:

   - Add `axil_macc.cpp` as the *design file* — this contains the C description of the dot-product hardware accelerator.
   - Add `tb_axil_macc.cpp` as the *testbench file* — this contains the `main()` function used for C simulation only and is **not** synthesized into hardware.

   Click **Browse** to select `axil_macc` as the *top-level function* (the function that will be synthesized into hardware), then click **Next**.

5. In the **Hardware** tab, search for and select the FPGA part corresponding to your board:

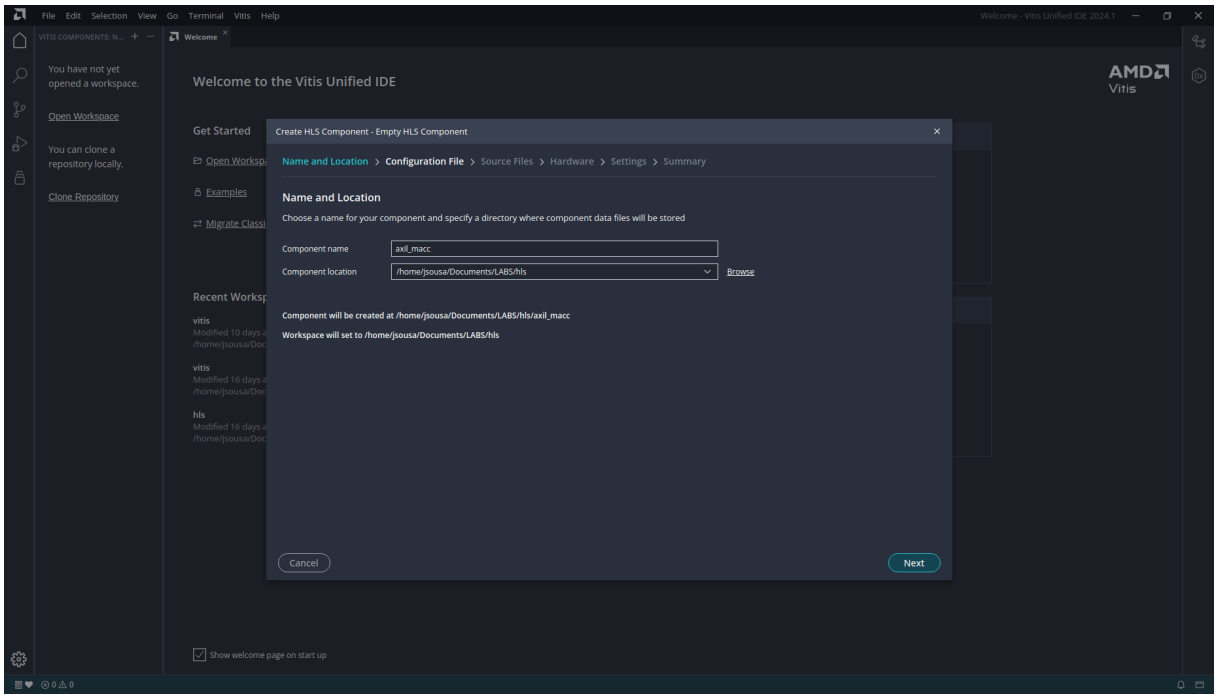


Figure 2: Choose the component name and location

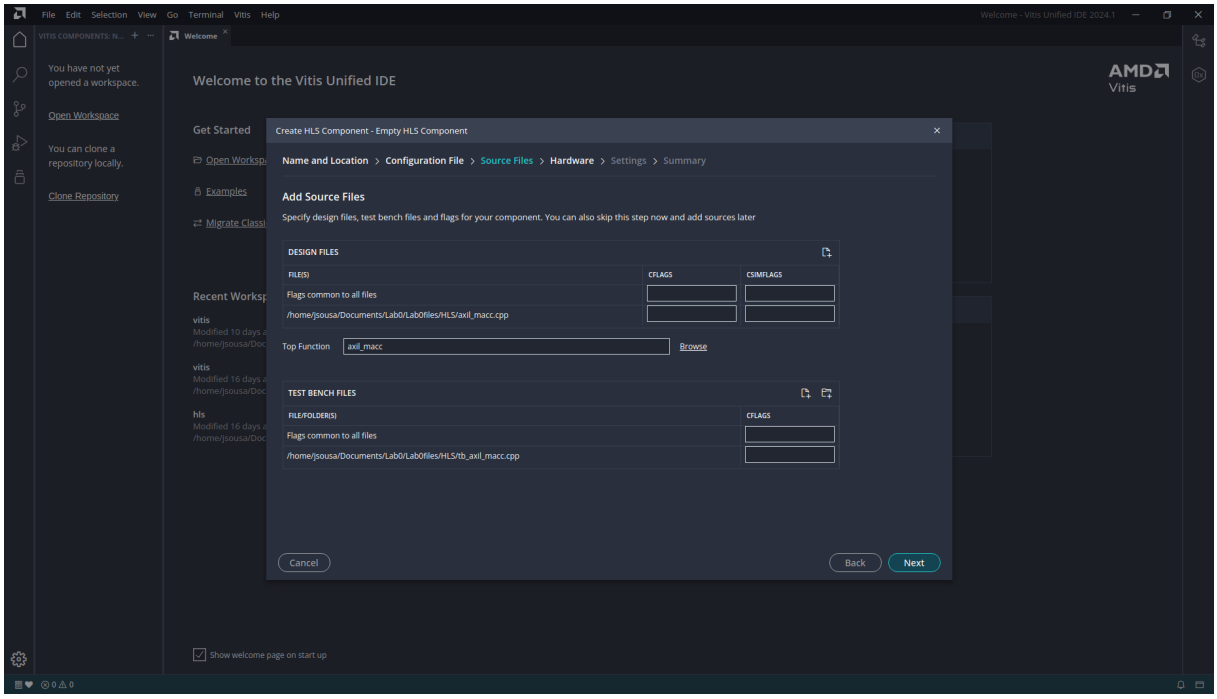


Figure 3: Source Files tab — adding design and testbench files and selecting the top function

- `xc7z010clg400-1` — for boards equipped with the Zynq-7010 (e.g., the Zybo-Z7-10).
- `xc7z020clg400-1` — for boards equipped with the Zynq-7020 (e.g., the Zybo-Z7-20 or PYNQ-Z1/Z2).

Click **Next**.

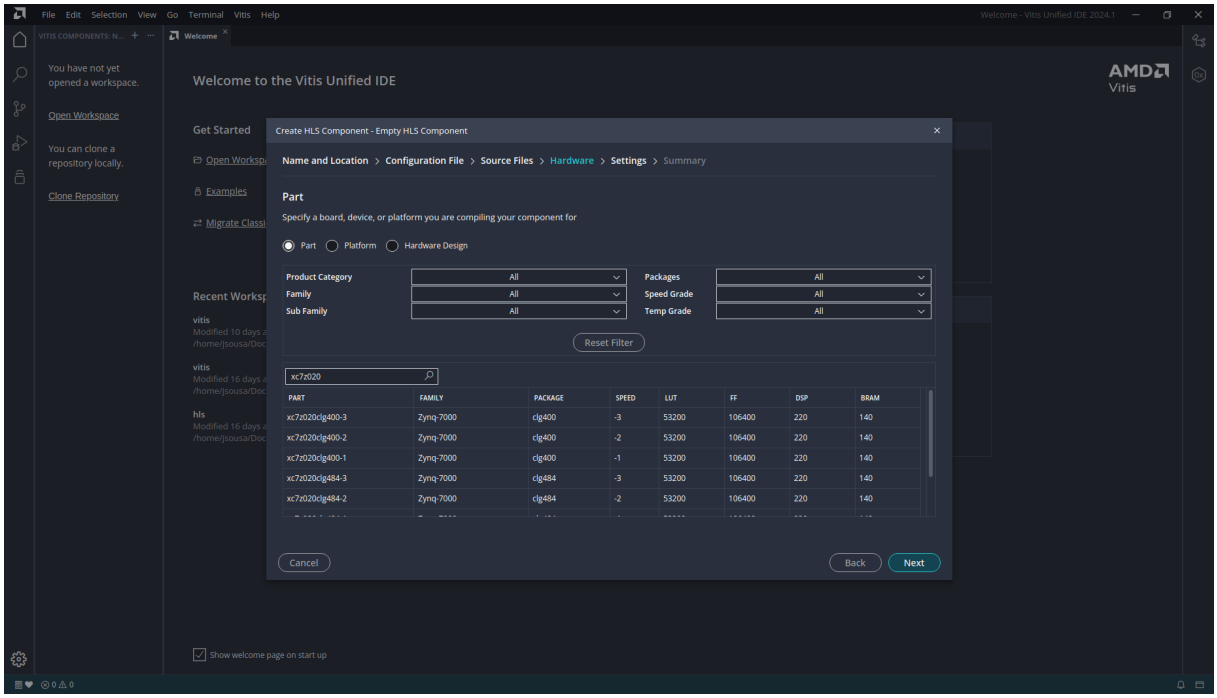


Figure 4: Hardware tab — selecting the target FPGA part

6. In the **Settings** tab, configure the HLS options as shown in the figure below (clock period, flow target, etc.) and click **Next**.
7. In the **Summary** tab, review all your selections and click **Finish** to create the component.

2.2 Verify the C-Based Hardware Specification (C Simulation)

Before synthesizing the design into RTL, it is good practice to verify that the C description is functionally correct using *C simulation*. This step compiles and runs the C code natively on the host machine, using the testbench to provide stimulus and check outputs. Because this simulation runs before any hardware generation, it executes very quickly and allows rapid debugging of algorithmic errors.

Run **Flow** → **C Simulation** → **Run** with the default settings (no options selected). Inspect the console output and verify that the dot-product result produced by `axil_macc` matches the reference result computed by the software checking function in the testbench. A *PASS* message confirms functional correctness.

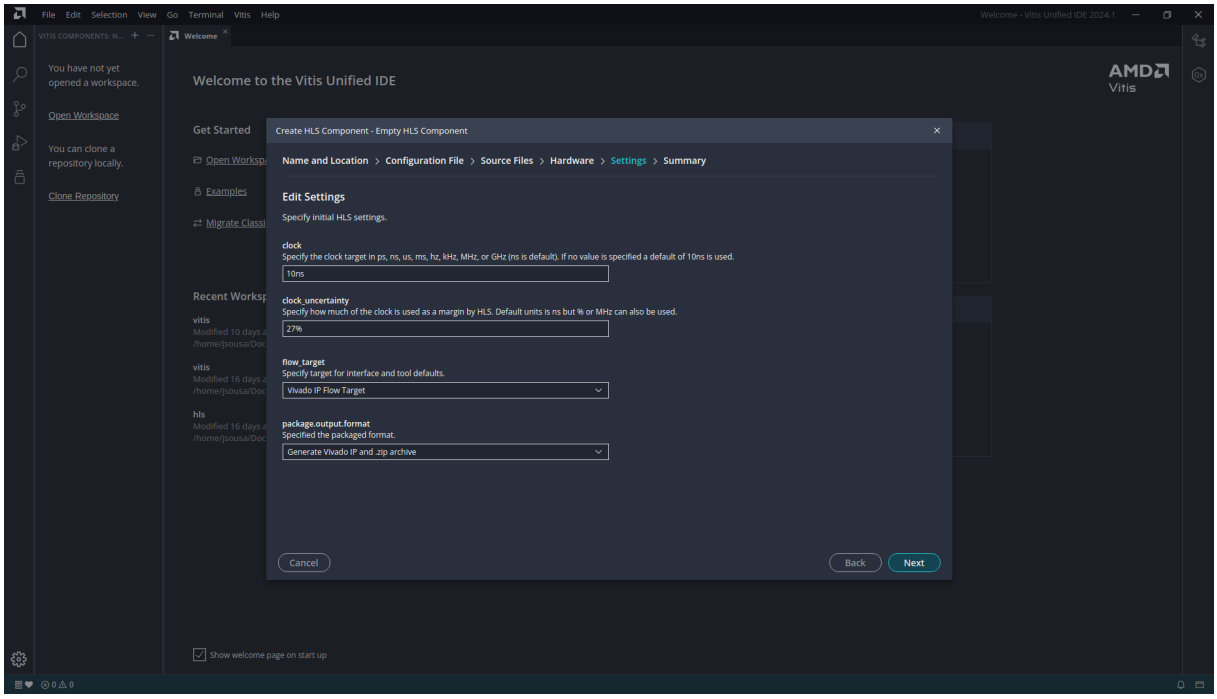


Figure 5: Settings tab — HLS synthesis configuration

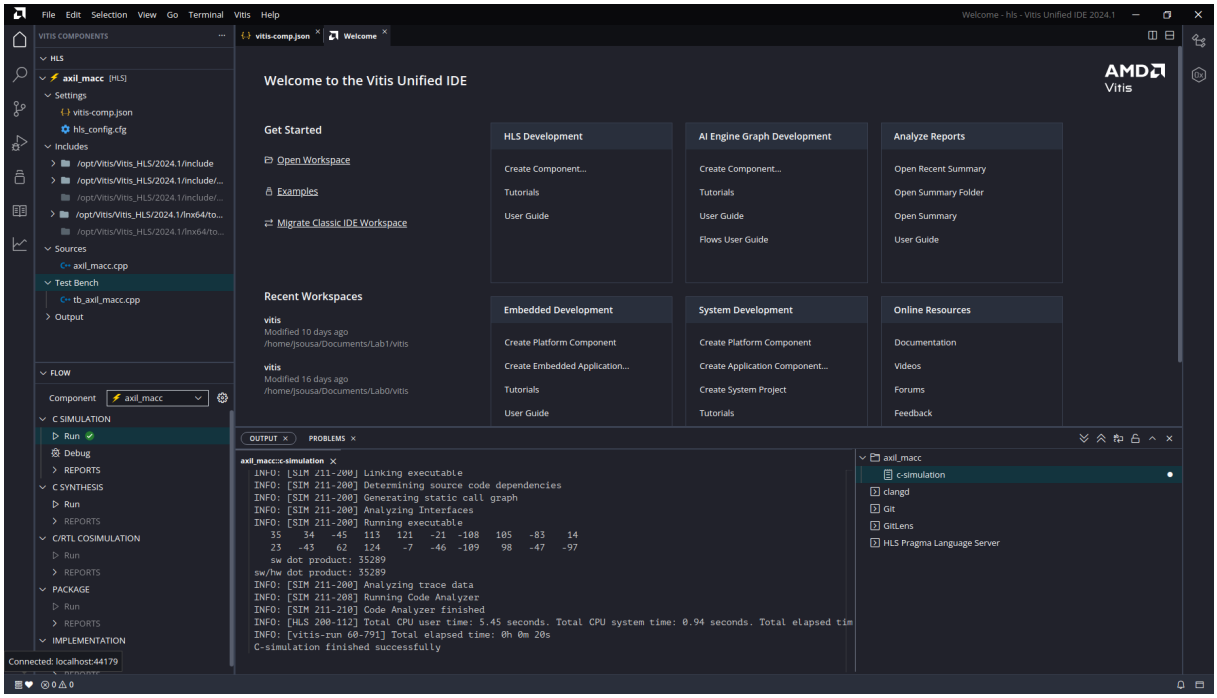


Figure 6: C simulation output — dot-product IP result matches the software reference

2.3 Synthesize the Design

HLS synthesis translates the C description of `axil_macc` into an RTL implementation. The tool schedules operations across clock cycles, allocates hardware resources (DSP blocks, BRAMs, LUTs, flip-flops), and inserts the AXI-Lite interface logic.

Run **Flow** → **Synthesis** → **Run**.

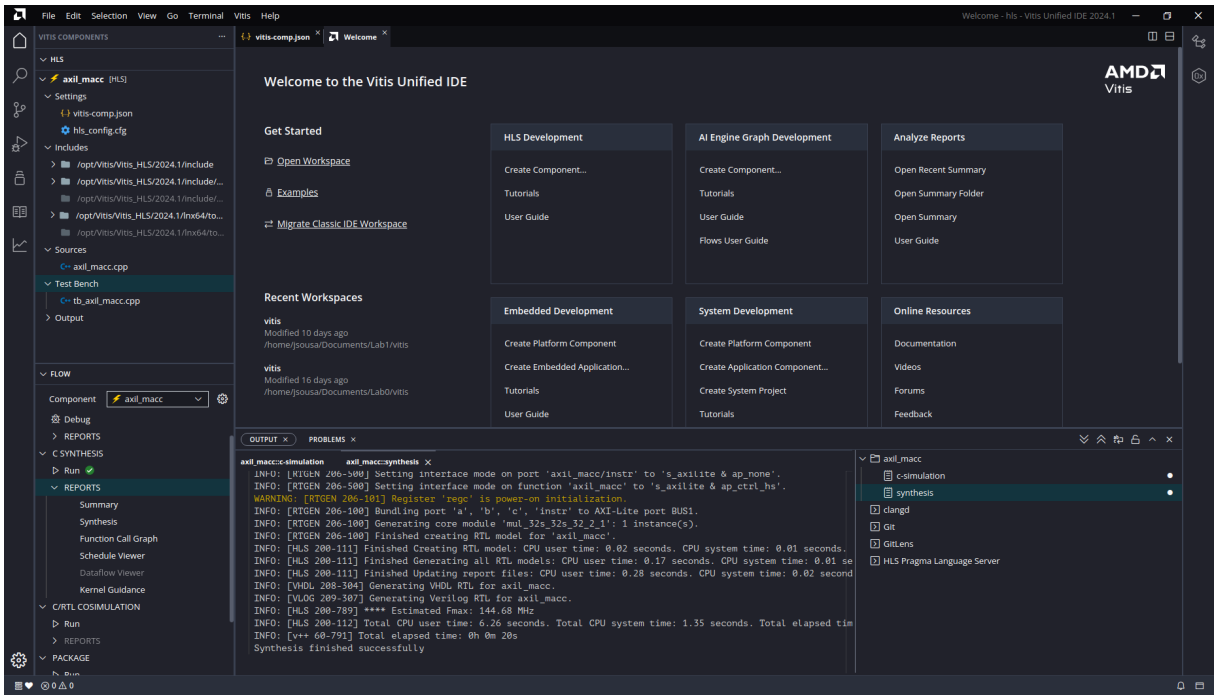


Figure 7: Running HLS synthesis

Once synthesis completes, carefully examine the synthesis report:

- **Performance estimates** — the estimated clock period (which must meet the target constraint) and the initiation interval and latency (in clock cycles) of the top-level function.
- **Resource utilization estimates** — the number of LUTs, flip-flops, DSP blocks, and BRAMs consumed. Compare these figures with the resources available on the target device.
- **Interface summary** — lists the ports and protocols generated for the top function, including the AXI-Lite slave interface and any additional control signals.

Also open the **Schedule Viewer** to understand how operations are mapped to clock cycles and identify any bottlenecks or pipeline stages.

Note: The `sync` folder (inside the `hls` directory in the Explorer pane) contains `report`, `verilog`, and `vhdl` sub-folders with the generated synthesis reports and RTL source files.

2.4 C/RTL Co-Simulation

C/RTL co-simulation provides a post-synthesis functional verification of the generated RTL. The same testbench used in C simulation is reused to drive the RTL model, ensuring

that the synthesized design is cycle-accurate and produces the same results as the original C description. This step is a *non-exhaustive* verification — it does not replace formal verification or timing simulation, but it catches common synthesis mismatches.

Before running co-simulation, enable waveform dumping so that you can inspect signal transitions in Vivado:

1. Open `HLS/axil_macc/Settings/hls_config.cfg`.
2. Locate the co-simulation settings and set `trace_level=all` and enable `wave_debug`.

Run **Flow** → **C/RTL Co-simulation**.

After the run completes, open the generated waveform in the Vivado simulator. Identify the AXI-Lite read and write transactions on the interface signals, and verify that the simulation produces a *PASS* result.

2.5 Package the IP Component

Once the design has been verified through C simulation, synthesis, and co-simulation, it can be packaged as a reusable IP block for use in Vivado.

Run **Package** → **Run** to export and package the synthesized hardware component into a standardized Xilinx IP format.

When the packaging step completes, a new `hls/impl/ip` sub-folder appears in the Explorer pane. This folder contains all files that define the IP block, including:

- The RTL source files (Verilog/VHDL).
- The IP metadata file (`component.xml`), which describes the interfaces, parameters, and bus definitions to Vivado.
- The **drivers** sub-folder, which contains automatically generated bare-metal C driver files. These drivers provide high-level functions to initialize the IP, start a transaction, and read results from software running on the Zynq PS.

The IP is now ready to be added to a Vivado repository and instantiated in the Zynq block design, as described in the next section.

3 Zynq Project in Vivado

Start the Vivado application from the desktop shortcut or command line without closing Vitis.

3.1 Create a New Project targeting the FPGA board

3.1.1 Open Vivado

3.1.2 Create a New Project

3.1.3 Name the project

Note: Avoid using spaces and/or special characters in your directory path and in the filenames.

Figure 8: Open Vivado — without closing Vitis

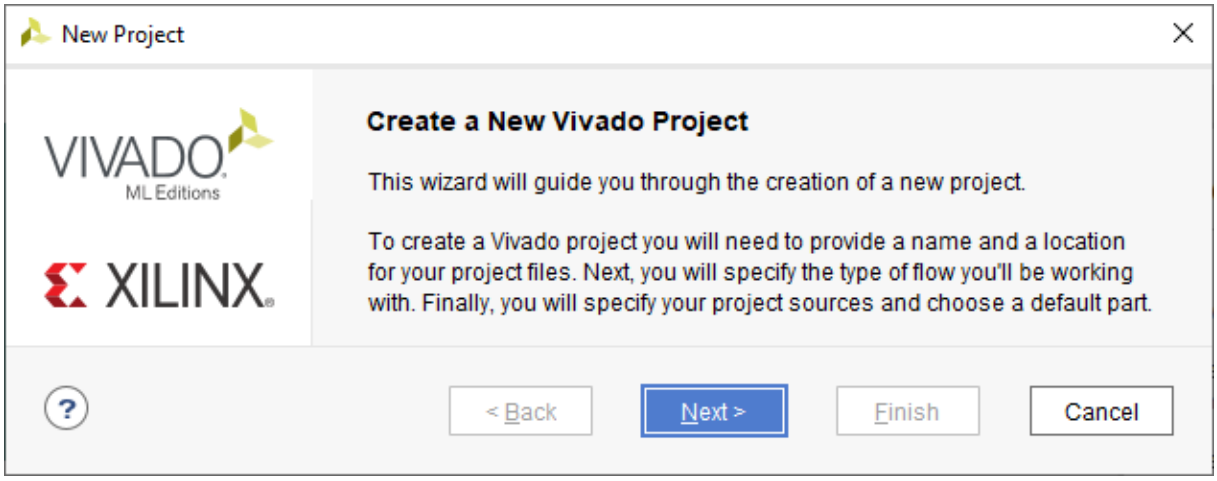


Figure 9: Open Vivado — Create a New Project

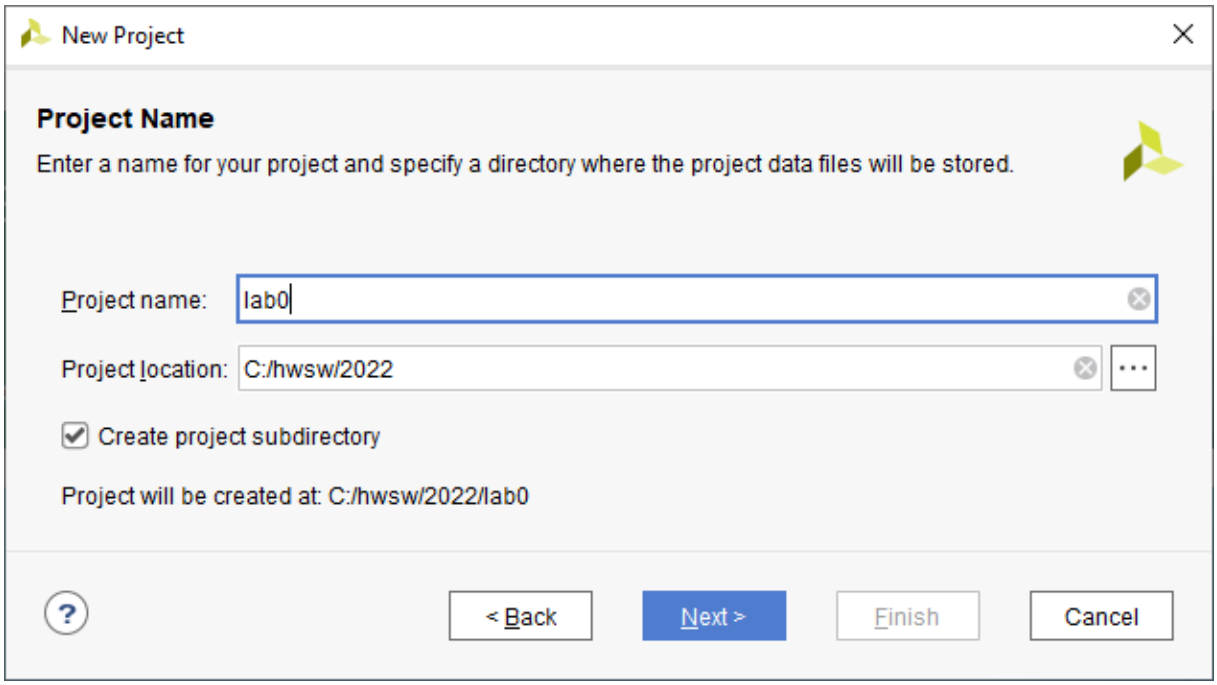


Figure 10: Open Vivado — Name the project

3.1.4 Select project type

3.1.5 Skip Add Sources and Constraints

Click Next twice to skip Add Source and Add Constraints.

3.1.6 Board Selection

In the Default Part form, select Boards and then choose the board you are using: Zybo, Zybo Z7-10 or Pynq-Z2 (check the version of your board).

Note: if the board is not in the board list, go to Tools/Vivado Store to install it.

Click Next, check the Project Summary and click Finish to create an empty Vivado project.

3.2 Use IP Integrator to create a new Block Design, with the ZYNQ processing system

3.2.1 Create a Block Design

In the Flow Navigator, click Create Block Design (under IP Integrator) with the design name **design_1**.

3.2.2 Add the Processing System

IP components can be added from the catalog by clicking the Add IP icon (+) in the block diagram, or by right-clicking anywhere in the Diagram workspace and selecting Add IP. Select the Zynq7 Processing System and add it to the design.

Click on Run Block Automation with the default settings by pressing OK.

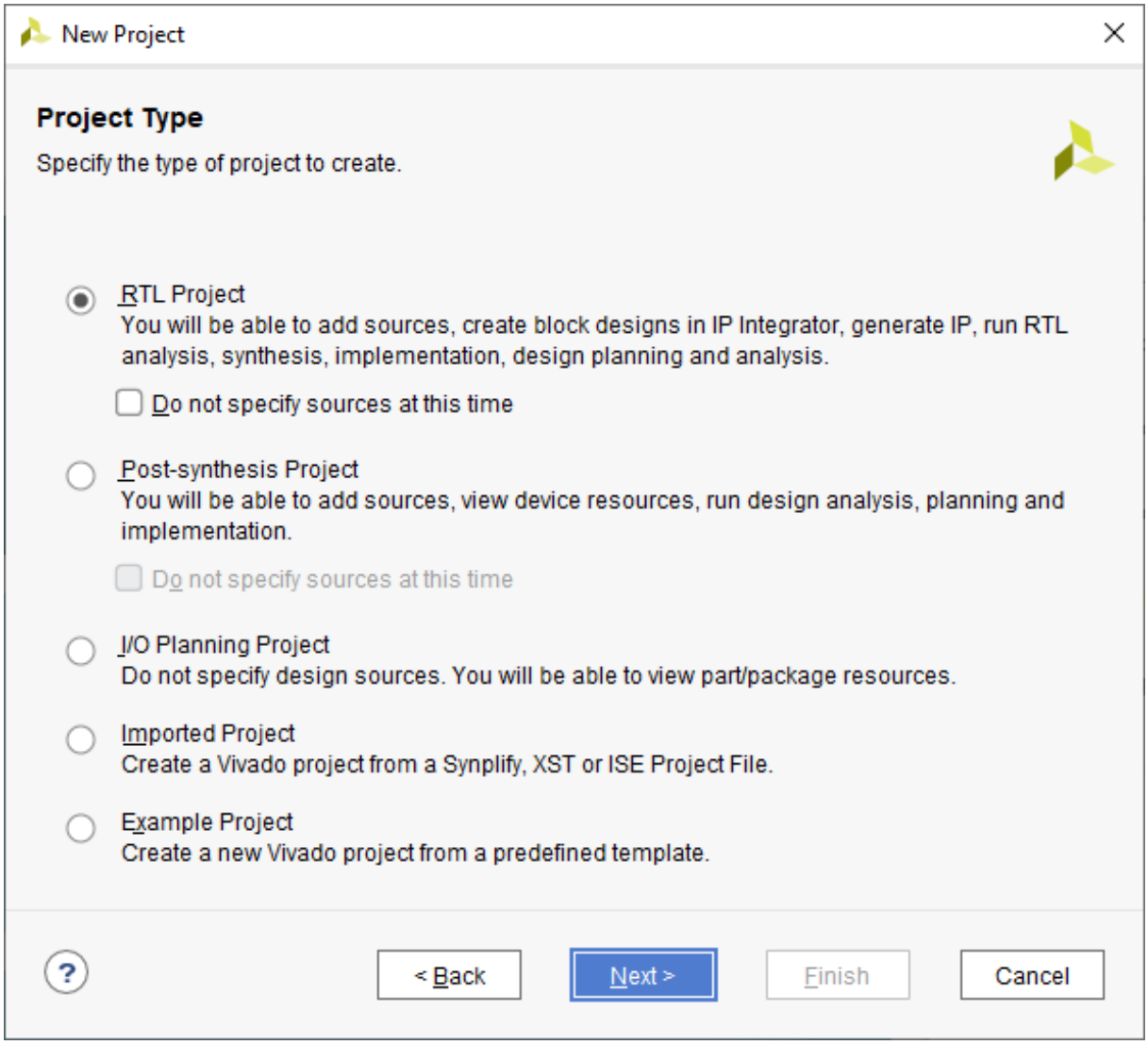


Figure 11: Select project type — path and in the filenames

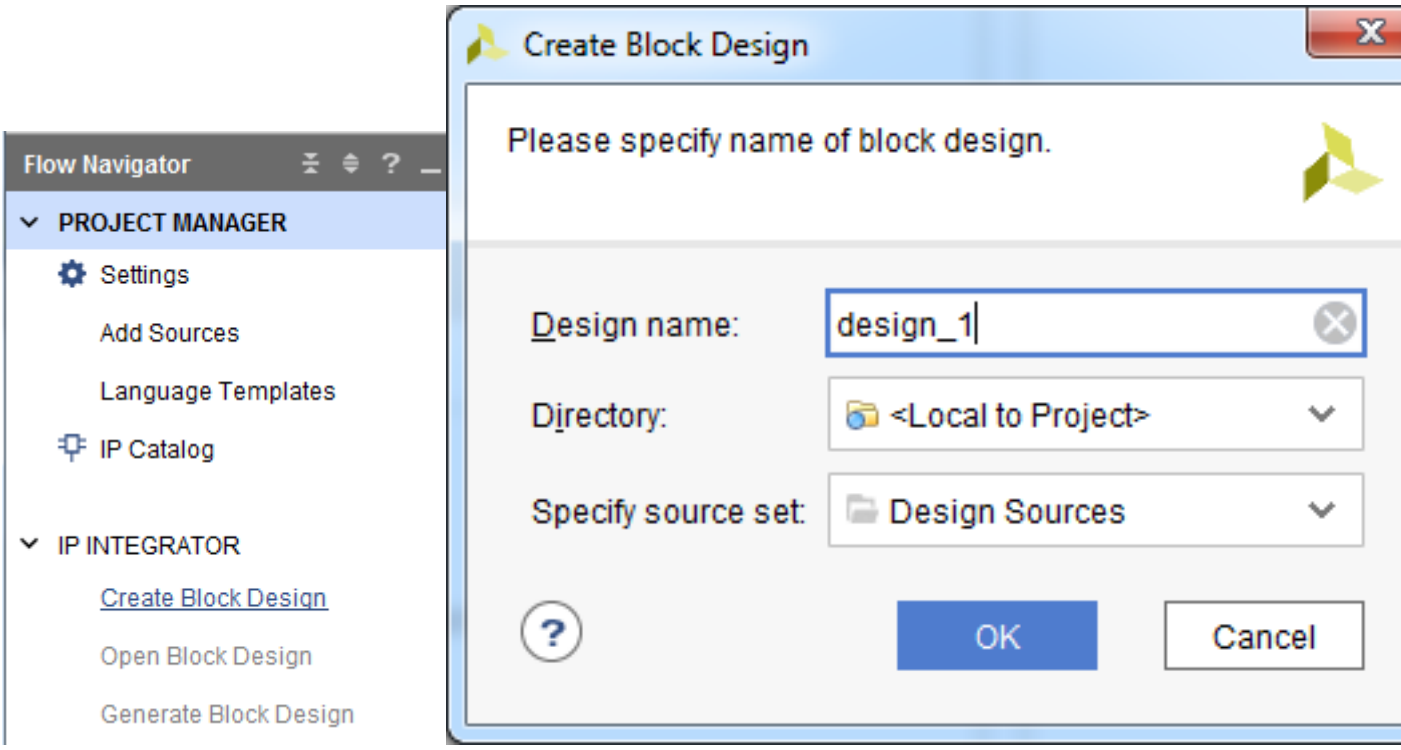


Figure 12: Create a Block Design — with the design name design_1

Ports are automatically added for the DDR and Fixed IO. An imported configuration for the Zynq PS system related to the Zybo board has been applied, which can be customized.

3.2.3 Configure the processing block with only one UART peripheral

Double-click on the PS block to open its customization window.

Open the MIO configuration form and ensure **all the I/O are deselected except UART 1**, i.e., ENET 0, USB 0, SD 0 (I/O Peripherals), GPIO MIO (GPIO), Quad SPI Flash (Memory Interfaces), and Timer 0 (Application Processor Unit) must be deselected.

3.2.4 PS-PL Configuration

On the PS-PL Configuration window, keep the M AXI GP0 interface (AXI Non Secure Enablement → GP Master AXI interface) and the FCLK_RESET0_N option (General → Enable Clock Resets) selected.

Figure : PS-PL Configuration.

3.2.5 Clock Configuration

On the Clock Configuration window PL Fabric Clocks, keep the clock configuration with the FCLK_CLK0 clock enabled, and select a requested frequency of 100 MHz.

Do not change the Input Frequency, which is 50 MHz on the Pynq board and 33.(3) MHz on the Zybo-Z7-10 boards.

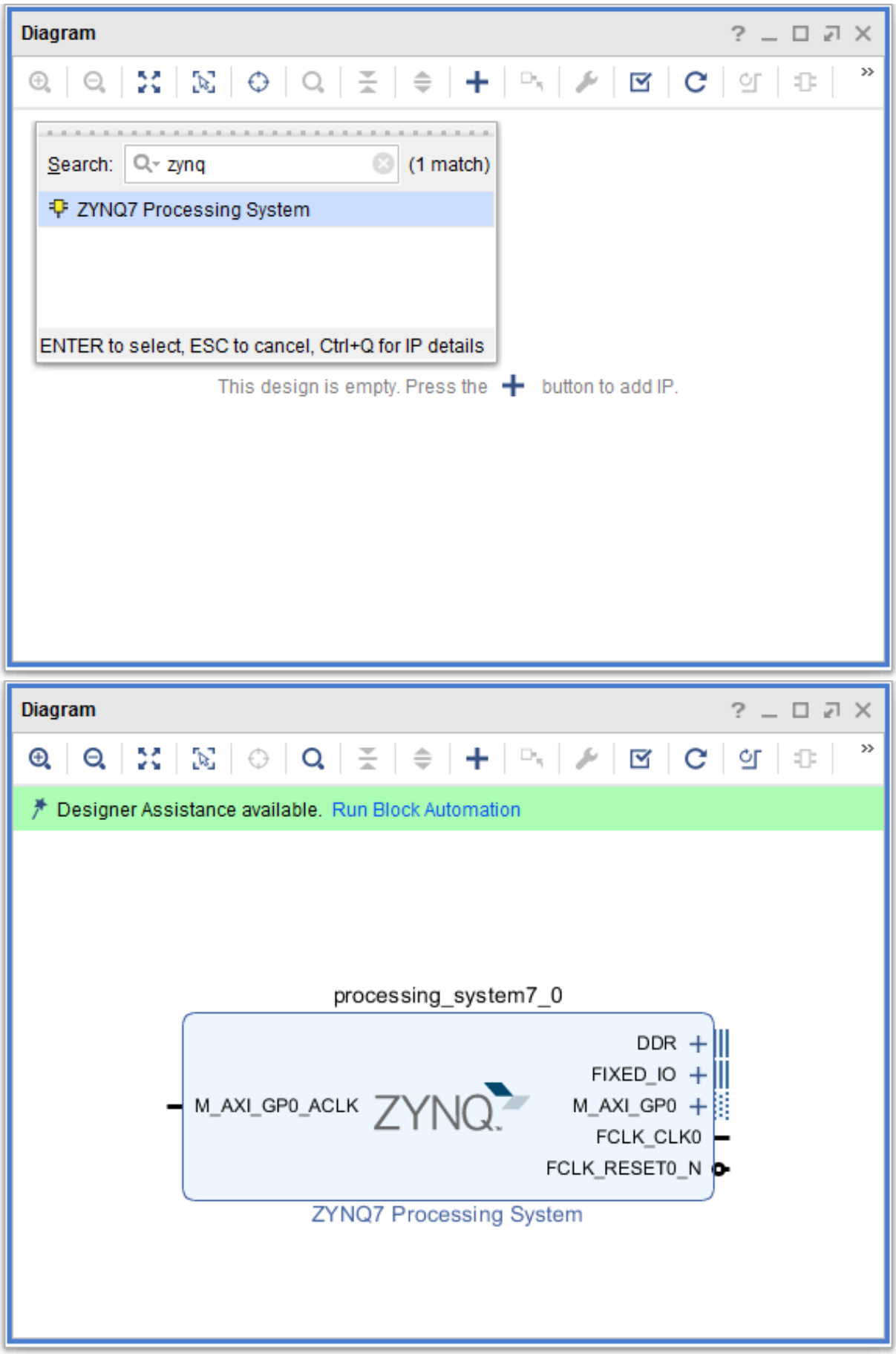


Figure 13: Create a Block Design — System and add it to the design

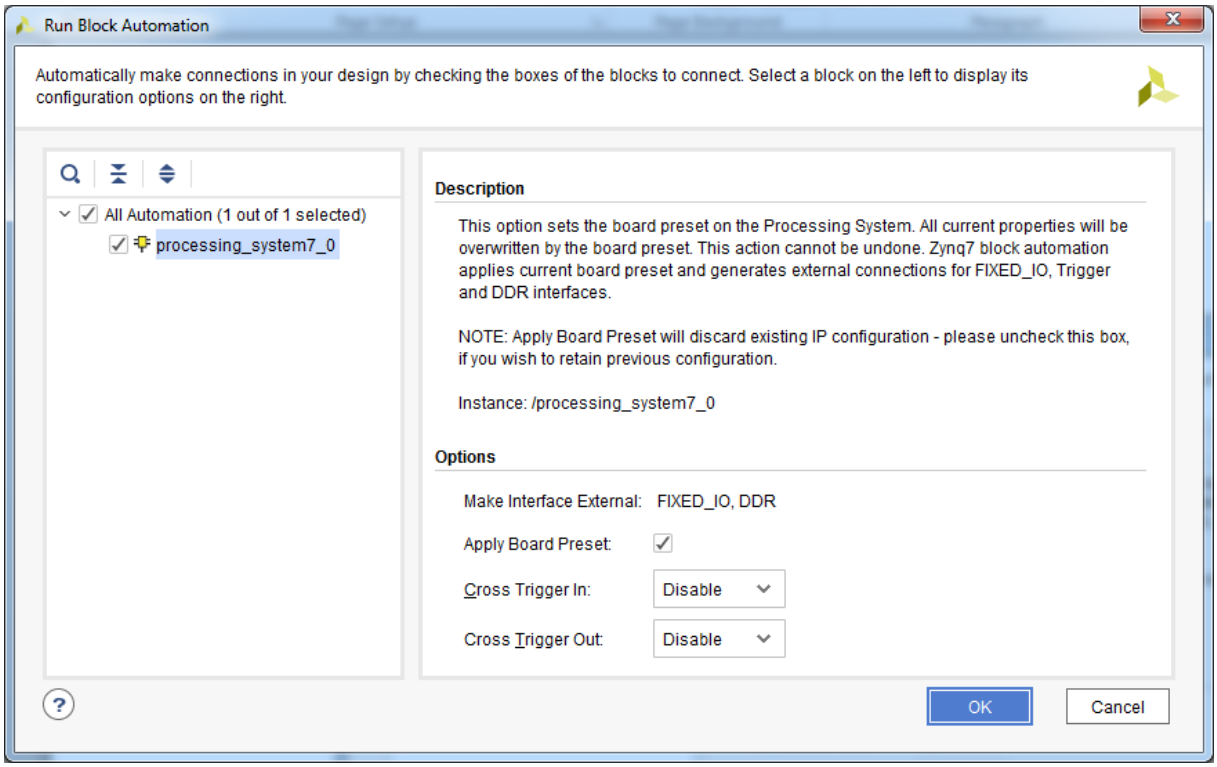


Figure 14: Create a Block Design — Click on Run Block Automation with the default settings by pressing OK

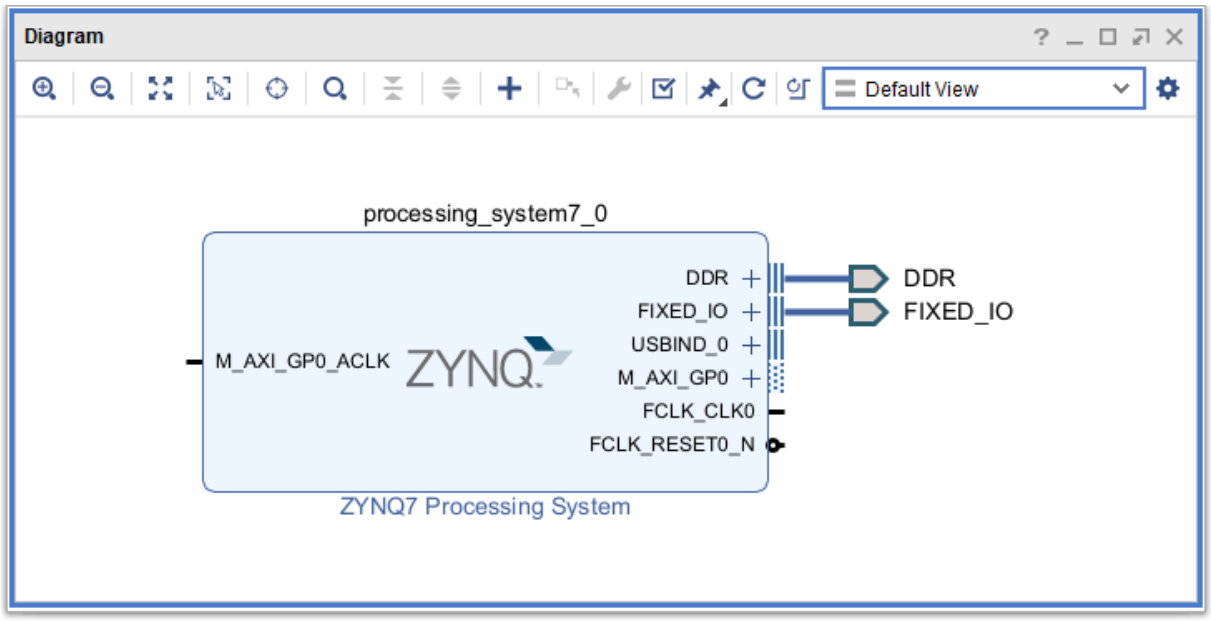


Figure 15: Create a Block Design — applied, which can be customized

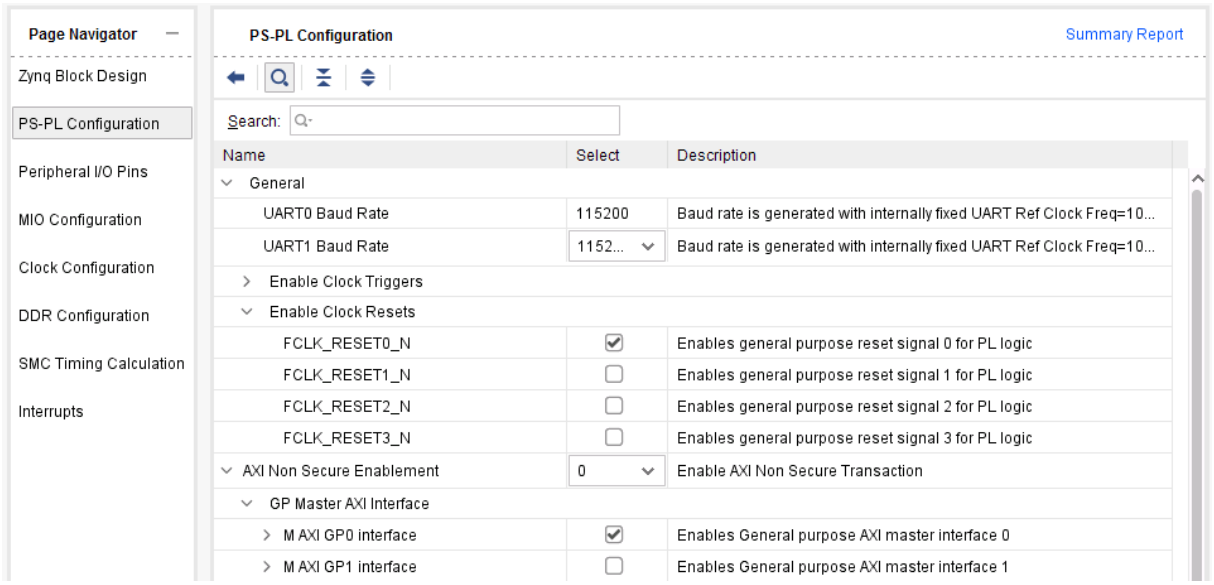


Figure 16: PS-PL Configuration — option (General → Enable Clock Resets) selected

3.2.6 Finalize the PS Configuration

Apply the customization and note that the Zynq PS block includes the GP0, clock, and reset ports:

3.3 Specify IP Repository

Add your IP repository folder to the IP repository list in the ***Project Settings***.

3.4 Add your IP to the Zynq design

3.4.1 Add New AXI-Lite IP

Click the ***Add IP icon***, search for your new IP and add it to the design.

3.4.2 Connect the blocks

Complete the design by automatically connecting the PS and PL blocks: run ***Connection Automation*** (and select /axil_macc_0/s_axi_BUS1) to automatically make the required connections:

Note that two additional blocks, Processor System Reset and AXI Interconnect, have been automatically added to the design.

Regenerate the layout .

Validate the design .

Note: if a warning message appears on "PCW_UIPARAM_DDR_DQS_TO_CLK_DELAY_0 has negative value", you may ignore it.

Check the Address Editor tab and check which address region has been assigned to the axil_macc_0 s_axi_BUS_A Registers.

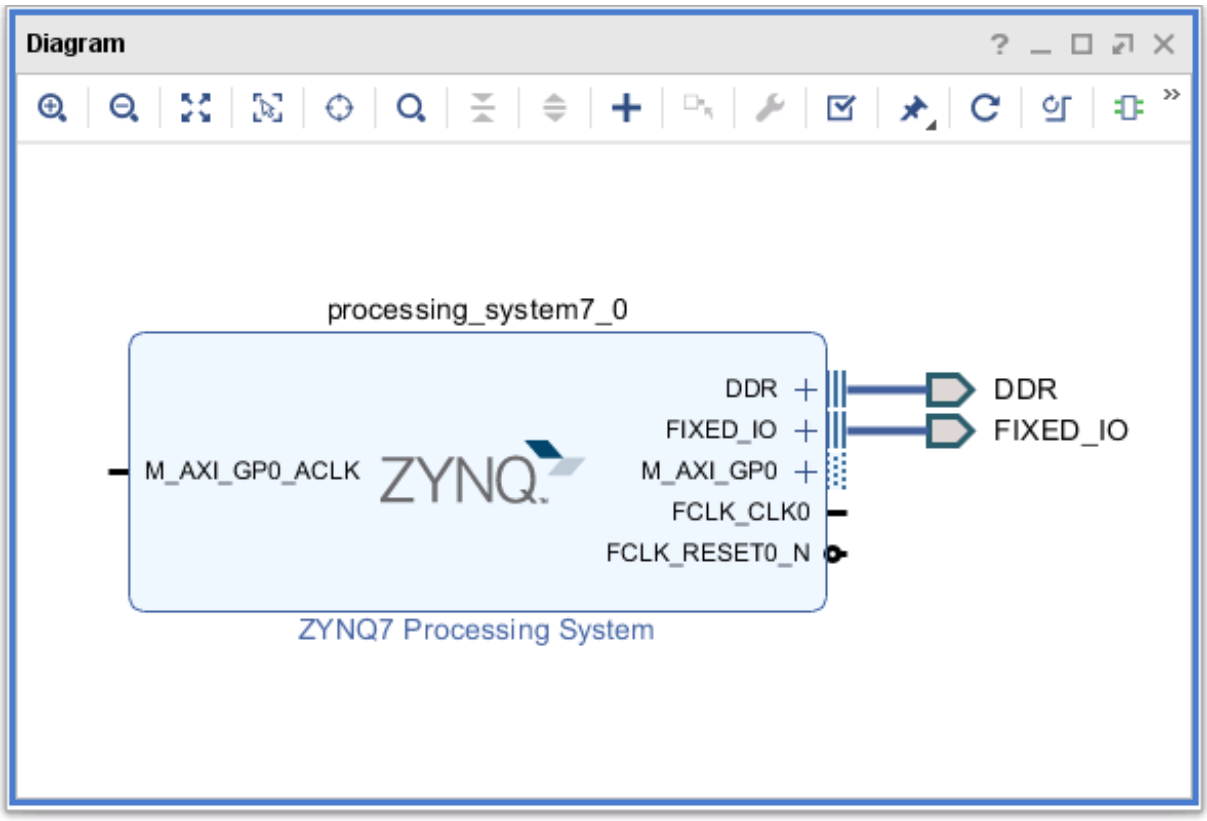


Figure 17: Clock Configuration — GP0, clock, and reset ports:

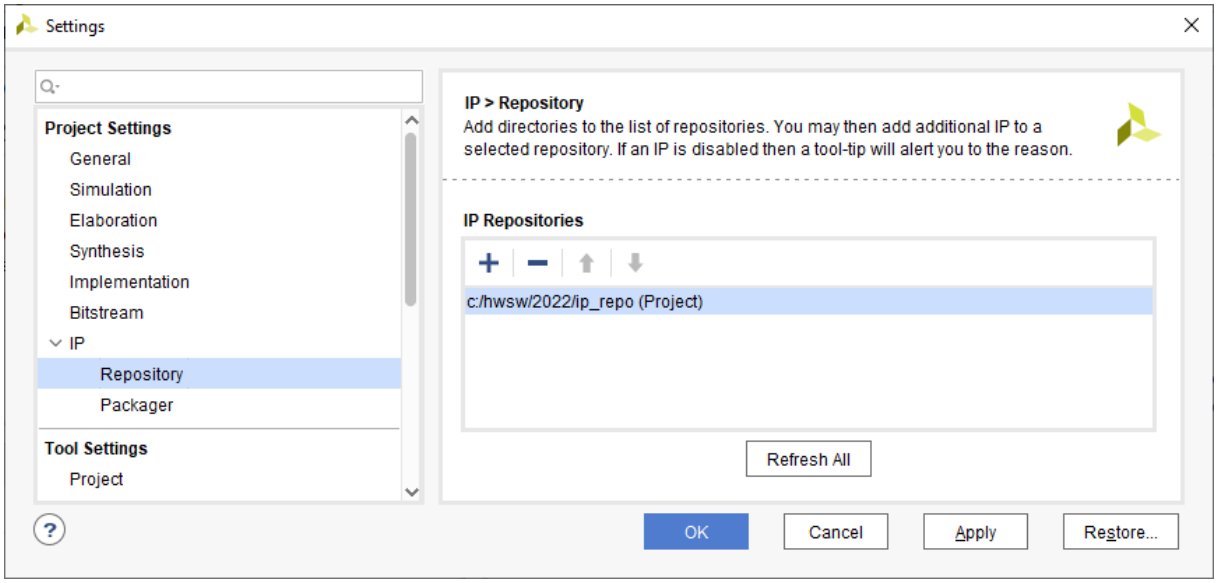


Figure 18: Specify IP Repository — Add your IP repository folder to the IP repository list in the

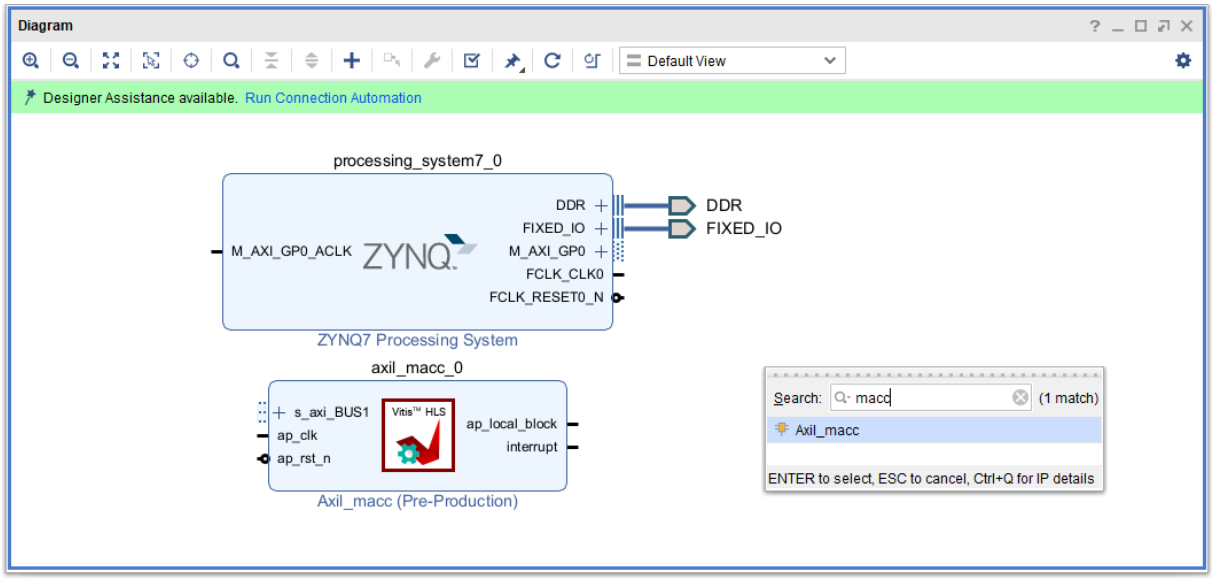


Figure 19: Add New AXI-Lite IP — to the design

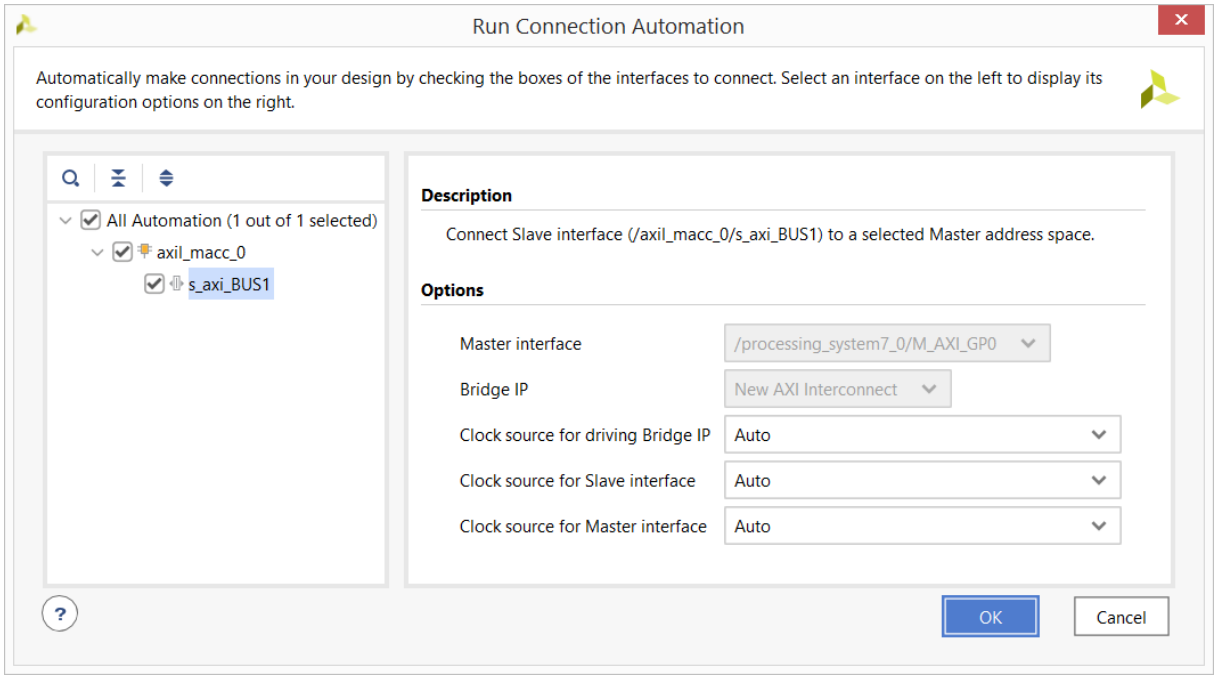


Figure 20: Connect the blocks — connections:

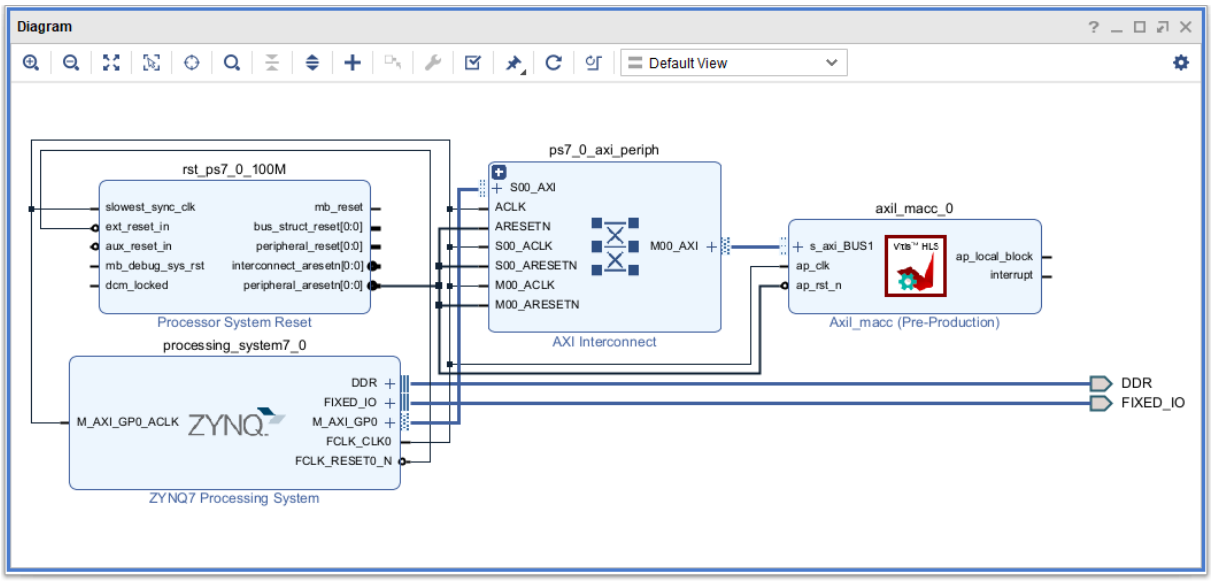


Figure 21: Connect the blocks — Interconnect, have been automatically added to the design

Note: ***Do not change*** the mapping addresses generated by Vivado.

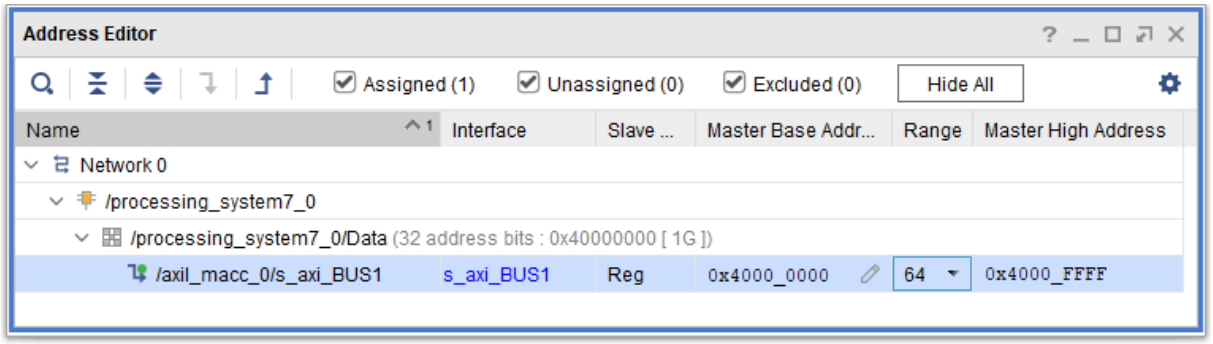
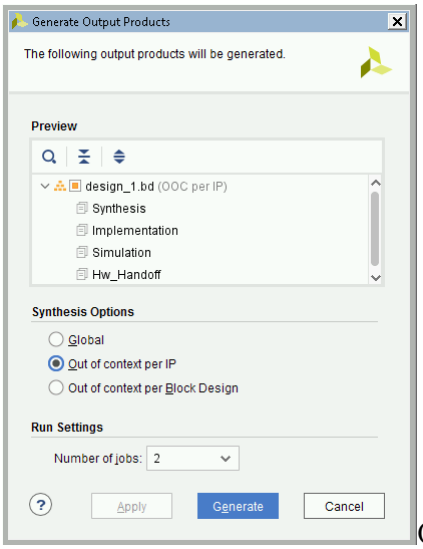


Figure 22: Connect the blocks — Note: the mapping addresses generated by

3.5 System Hardware Generation

3.5.1 Generate Block Design



Click on Generate Block Design (on the Flow Navigator) to generate the Implementation, Simulation and Synthesis files for the design.

3.5.2 Create HDL Wrapper

3.5.3 In the Sources panel, right-click on *design_1.bd* and select Create HDL Wrapper to generate the top-level VHDL model. Leave the "*Let Vivado manager wrapper and auto-update*" option selected.

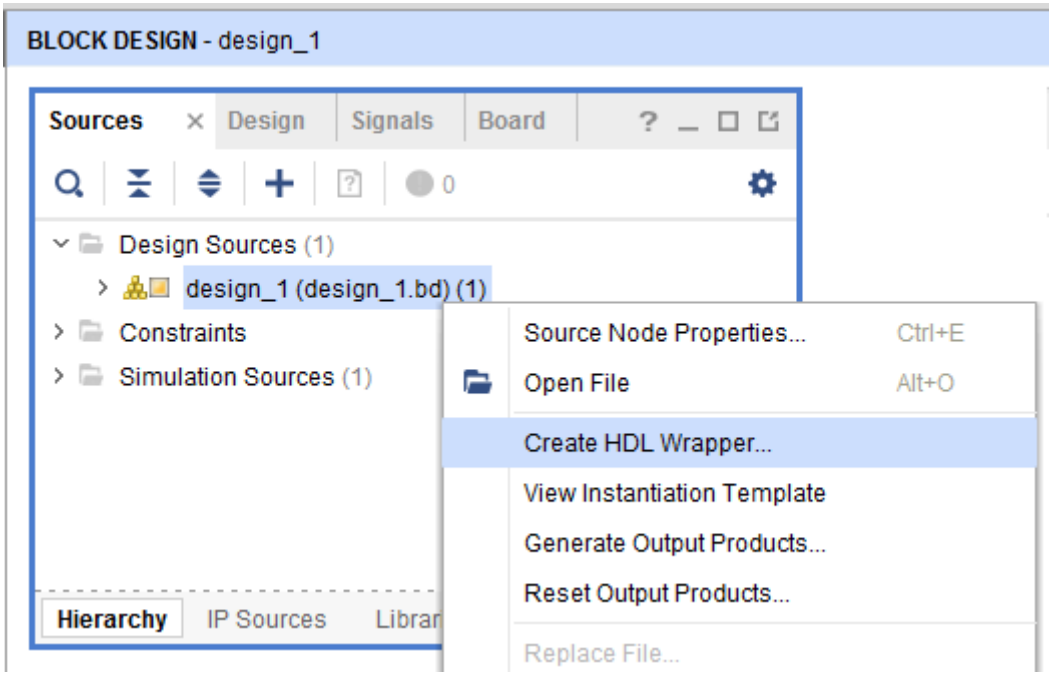


Figure 23: Create HDL Wrapper — right-click design_1.bd and select Create HDL Wrapper

The *design_1_wrapper.vhd* file is created, added to the project, and set as the top module in the design. (Double-click on the file name to see the content in the Auxiliary pane.)

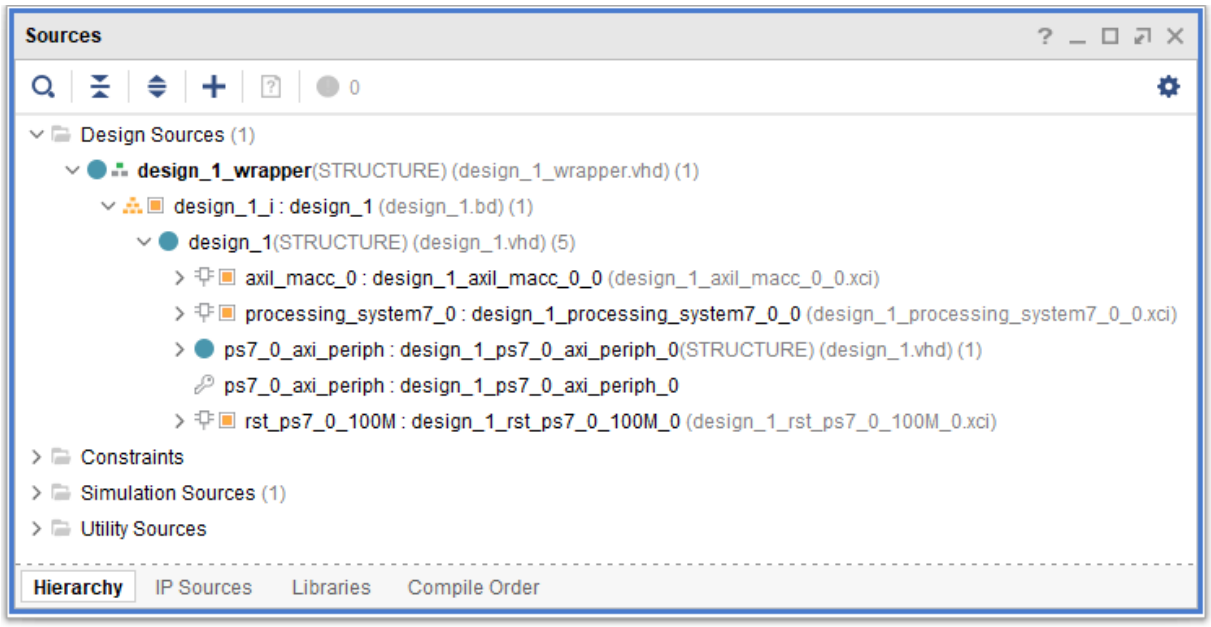


Figure 24: Create HDL Wrapper — to see the content in the Auxiliary pane.)

3.5.4 Design Implementation

Synthesize the design (***Run Synthesis***), implement it (***Run Implementation***), and generate the bitstream (***Generate Bitstream***).

The hardware system has been generated and can now be exported to the Vitis IDE to develop the embedded software and verify the application on the development board.

After implementation, you may check the Vivado reports, namely the *Utilization Report* and the *Timing Summary Report*.

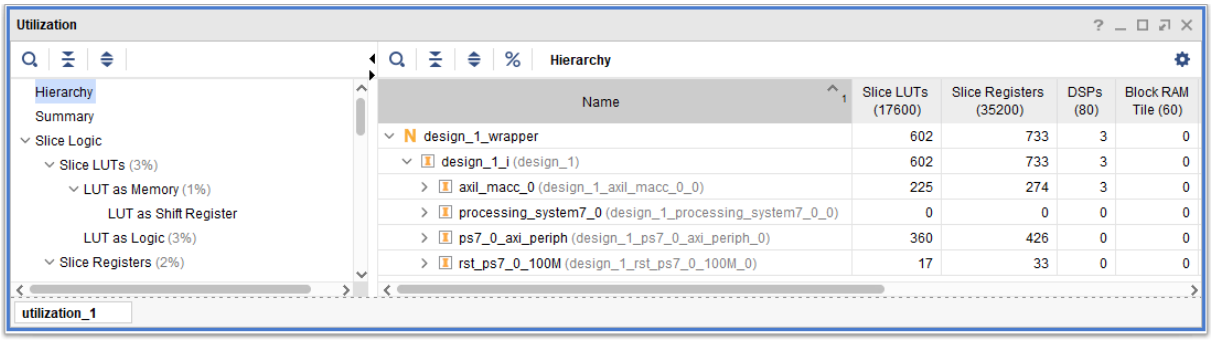


Figure 25: Design Implementation — Utilization Report and the Timing Summary Report

3.5.5 Export Hardware to the Vitis Software Platform

In Vivado, click ***File → Export → Export Hardware***.

This design has hardware in the Programmable Logic (PL), therefore, you must include the bitstream to be generated and included. Make sure that the export path corresponds to your project folder.

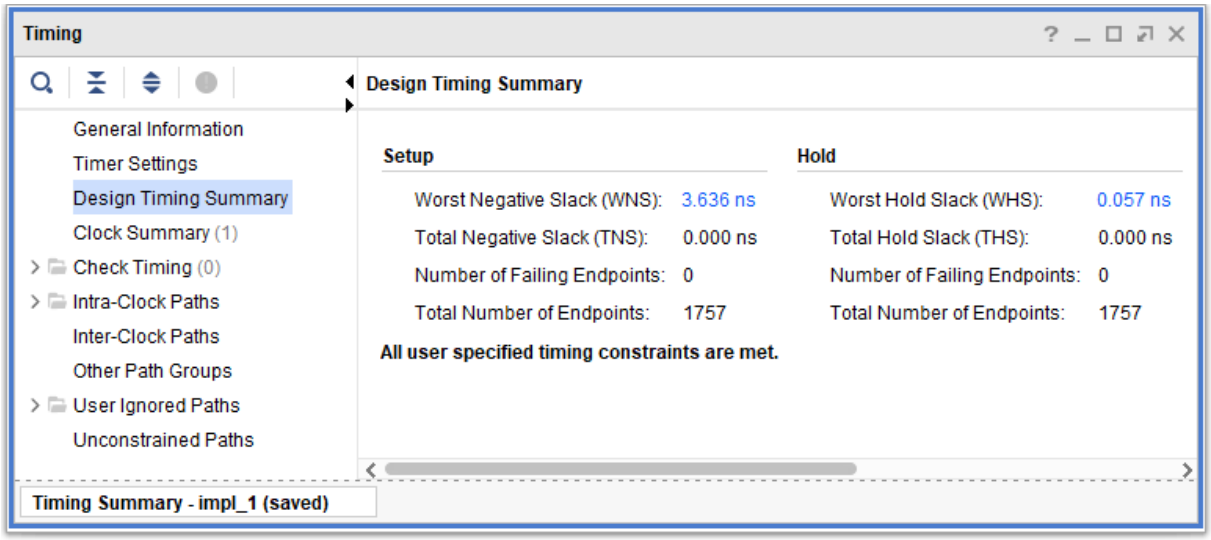


Figure 26: Design Implementation — Utilization Report and the Timing Summary Report

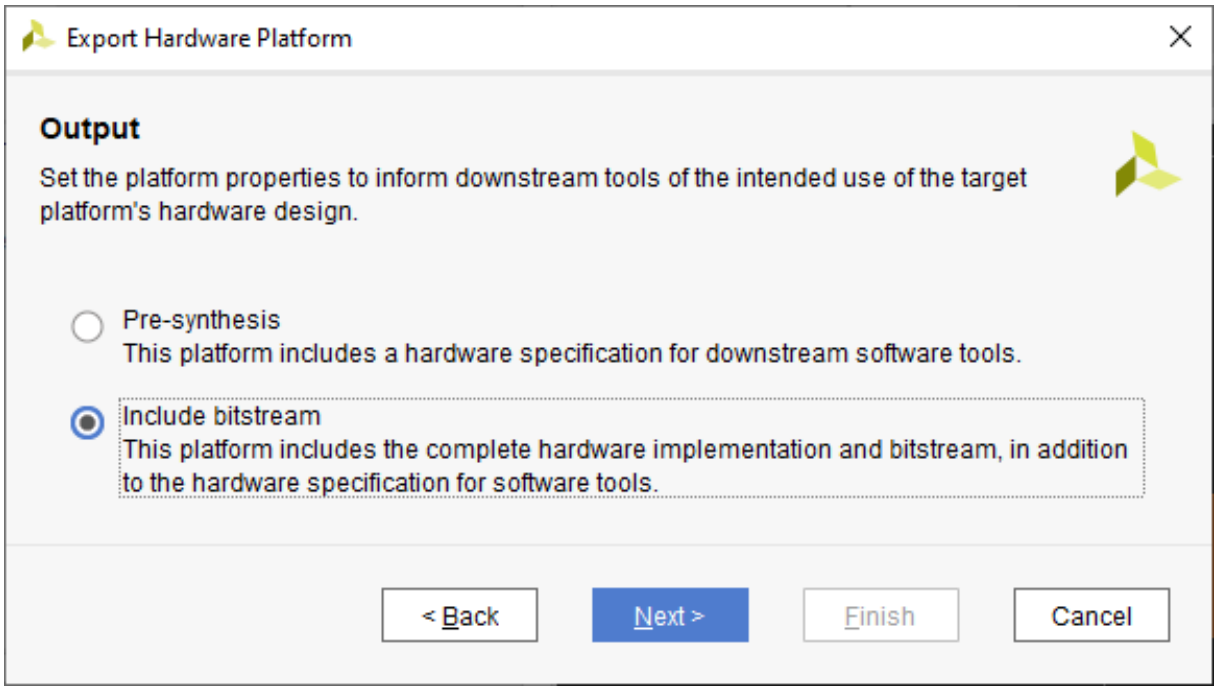


Figure 27: Design Implementation — the export path corresponds to your project folder

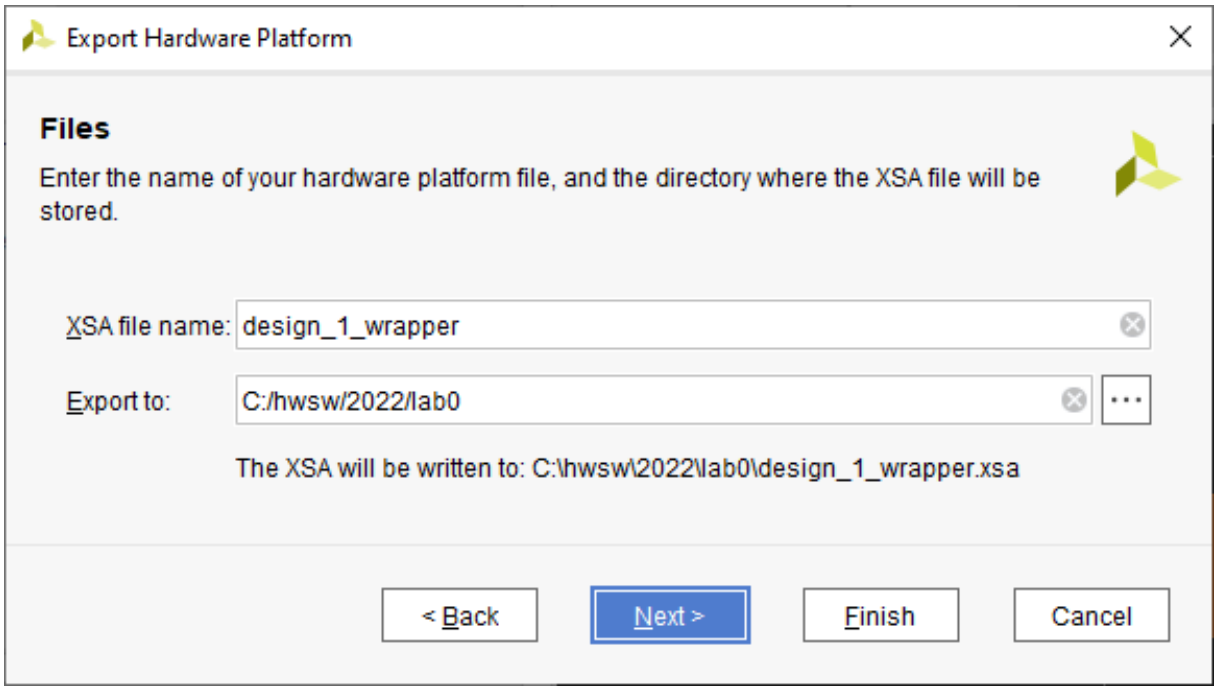


Figure 28: Design Implementation — the export path corresponds to your project folder

4 Software Application

Now return to the Vitis application you left open before.

1. Click ***Embedded Development → Create Platform Component to create a new platform project from the Xilinx Shell Archive (XSA) previously created in*** Vivado. Enter the platform component name and choose a suitable location. Click Next.
2. Browse to the hardware specification file (*design_1_wrapper.xsa) and select it. Click Next.*
3. The *Software Specification* fields (Operating system and Processor) are updated to *standalone* and *ps7_cortexa9_0,* respectively. Click Next, review the settings in the next screen and click Finish.
4. Build the platform project by clicking Flow/*Build.*

After the project builds, you can now start developing your software application.

4.1 Create your C Project

In this example, you will use a previously coded C program "*dotprod_v0.c*" that multiplies two vectors (software-only).

1. Create a new C project by clicking ***File → New Component → Application Project***. Choose a name and a suitable location for your software component, and click Next. Target the existing hardware platform you have previously created, and click Next. Leave the default Domain, click Next. Review and click Finish. The C project will be created and shown in the Vitis Project Components window.

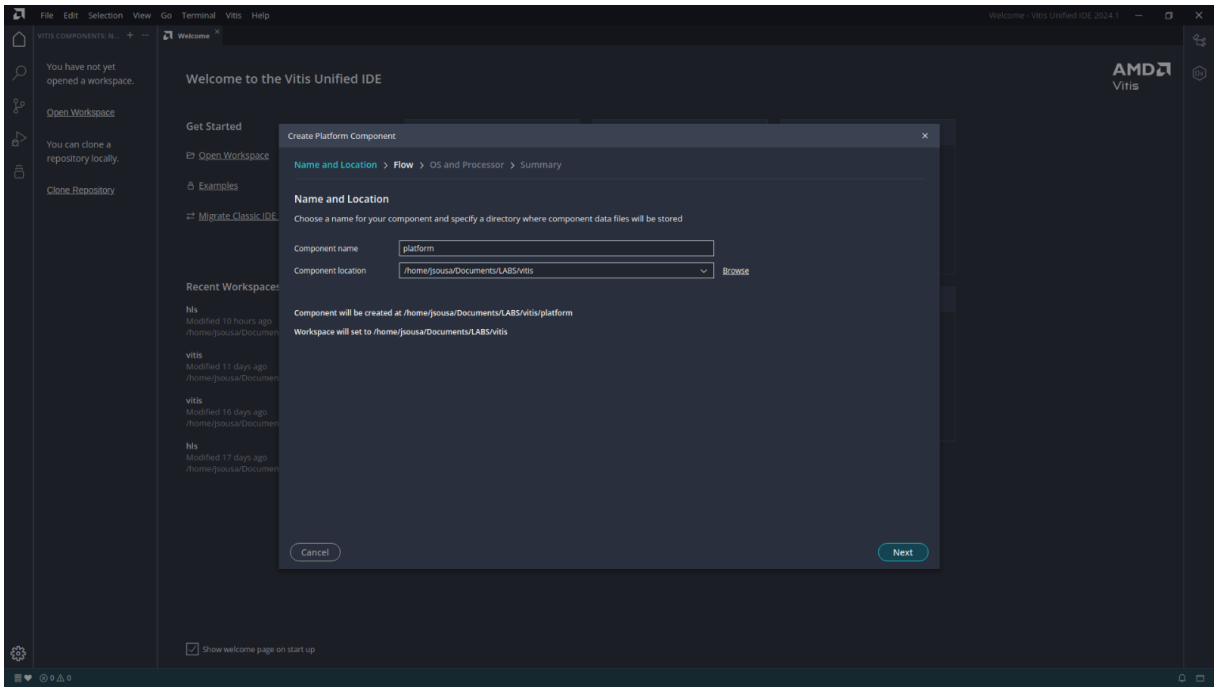


Figure 29: Software Application — choose a suitable location. Click Next

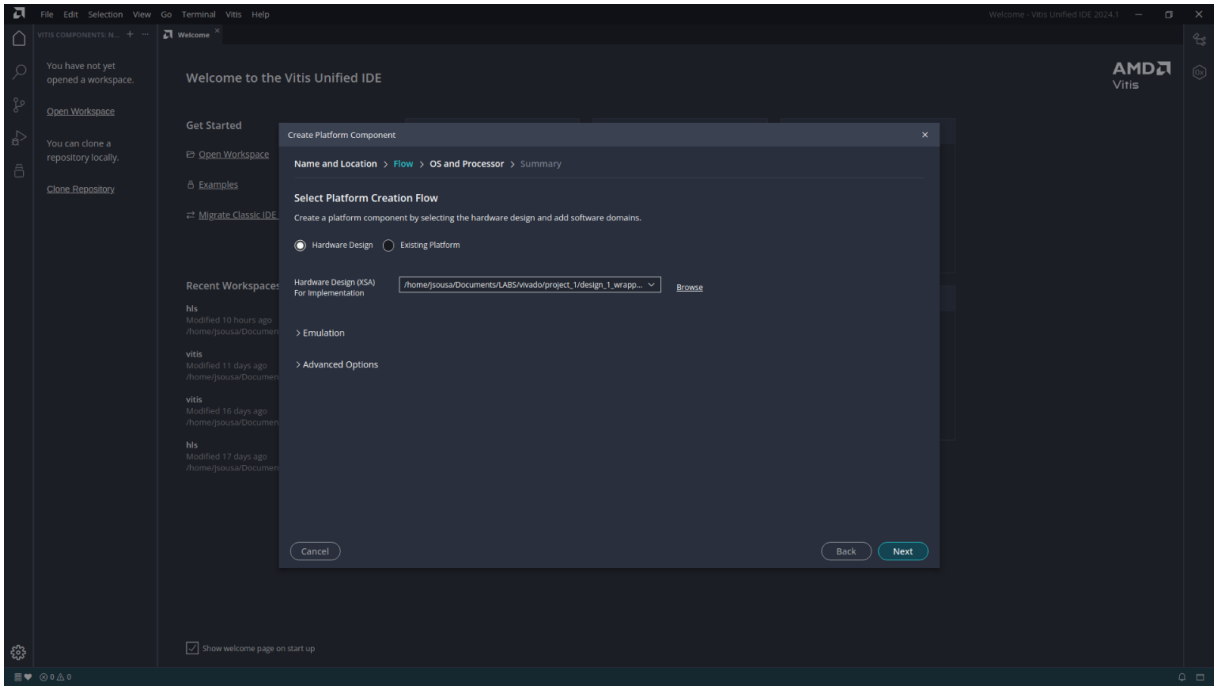


Figure 30: Software Application — (design_1_wrapper.xsa) and select it. Click Next

2. Now you can import C source files, using the C code provided as in this example. To import a new C source file, select the *src* folder in the Components view, right-click, and select ***Import->Files...***. Browse and import file *dotprod_v0.c*.

You can set the GCC compiler settings, including the optimization level, by selecting Settings (below your application) and clicking the *UserConfig.cmake* item. *For now, leave them as default, namely the Optimization Level set to None (-O0).*

You can also view the Linker Script by double-clicking on *Sources/src/lscript.ld*. Here you can define where your code's data and instruction sections will be stored. Place all the sections of your program on the OCM and leave the heap and stack sizes as 1K and 10K bytes, respectively.

Note: if you use the printf() function extensively, you may need to increase the heap size (e.g., to heap size = 10kB)

3. *Build* the project to generate *the* executable file. The linker combines the compiled applications, including libraries and drivers, and produces an executable file, in Executable Linked Format (ELF), that is ready to execute on your processor hardware platform.

Opening the ***.elf*** (see Components/Output), you can see the instructions generated for your C code, including their instruction memory addresses. At the top, you can view a list of the sizes and starting addresses of the various sections of the program, where ***size*** indicates the size of each section and ***LMA*** is the Loadable Memory Address (start address for each section). Note that, in this case, all sections reside in the OCM memory space (starting with base address 0x00000000).

You can check the size occupied by your program by opening the file ***dotprod0.elf.size***. In this example it occupies about 70 kB.

text data bss dec hex filename

51111 2548 17600 71259 1165b dotprod_0.elf

Note: your program must fit in with the memory you select to store it in!
Also, you must avoid trying to store multiple programs and/or data in the same memory zones.
All the memory management is done by the designer, you!

5 Verify the Design in Hardware

5.1 Connect the board and establish serial communication from Vitis's Terminal

Connect and power up the FPGA board. You will use one cable to connect to the PROG/UART port of the board.

Warning: Do not press hard when connecting the cable, as the board connector is fragile.

Open a terminal with

Vitis → Serial Monitor Terminal.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00008390  00000000  00000000  00010000  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .init         0000000c  00008390  00008390  00018390  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini         0000000c  0000839c  0000839c  0001839c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata       000003f3  000083a8  000083a8  000183a8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         000009ec  000087a0  000087a0  000187a0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .eh_frame     00000004  0000918c  0000918c  0001918c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .mmu_tbl      00004000  0000c000  0000c000  0001c000  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .ARM.exidx    00000008  00010000  00010000  00020000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .init_array   00000004  00010008  00010008  00020008  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .fini_array   00000004  0001000c  0001000c  0002000c  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 10 .ARM.attributes 00000033  00010010  00010010  00020010  2**0
                  CONTENTS, READONLY
 11 .bss          000000bc  00010010  00010010  00020010  2**2
                  ALLOC
 12 .heap         00000404  000100cc  000100cc  00020010  2**0
                  ALLOC
 13 .stack        00004000  000104d0  000104d0  00020010  2**0
                  ALLOC
 14 .comment      00000012  00000000  00000000  00020043  2**0
                  CONTENTS, READONLY
 15 .debug_info   000026f4  00000000  00000000  00020055  2**0
                  CONTENTS, READONLY, DEBUGGING
 16 .debug_abbrev 0000109c  00000000  00000000  00022749  2**0
                  CONTENTS, READONLY, DEBUGGING
 17 .debug_aranges 000002c0  00000000  00000000  000237e8  2**3
                  CONTENTS, READONLY, DEBUGGING
 18 .debug_macro  00001a1f  00000000  00000000  00023aa8  2**0
                  CONTENTS, READONLY, DEBUGGING
 19 .debug_line   00001d37  00000000  00000000  000254c7  2**0
                  CONTENTS, READONLY, DEBUGGING
 20 .debug_str    00007506  00000000  00000000  000271fe  2**0
                  CONTENTS, READONLY, DEBUGGING
 21 .debug_frame  00000444  00000000  00000000  0002e704  2**2
                  CONTENTS, READONLY, DEBUGGING
 22 .debug_loc    00000991  00000000  00000000  0002eb48  2**0
                  CONTENTS, READONLY, DEBUGGING
 23 .debug_ranges 00000150  00000000  00000000  0002f4d9  2**0
                  CONTENTS, READONLY, DEBUGGING
```

```
    void SW_dot_product()
    {
         6c0:   e92d4800    push    {fp, lr}
         6c4:   e28db004    add fp, sp, #4
         6c8:   e24dd008    sub sp, sp, #8
        int i;
        for (vdotp1=0, i=0; i<VEC_SIZE; i++) {
         6cc:   e300307c    movw    r3, #124    ; 0x7c
         6d0:   e3403001    movt    r3, #1
         6d4:   e3a02000    mov r2, #0
         6d8:   e5832000    str r2, [r3]
         6dc:   e3a03000    mov r3, #0
         6e0:   e50b3008    str r3, [fp, #-8]
         6e4:   ea000012    b   734 <SW_dot_product+0x74>
            vdotp1 += v1[i]*v2[i];
         6e8:   e300302c    movw    r3, #44 ; 0x2c
         6ec:   e3403001    movt    r3, #1
```

Select the serial port and baud rate. After clicking ***Run***, the application will execute, and you will see the *stdout* output in the Terminal console created before.

Note that this *dotprod_0* application is software-only (only the processor system is used).

5.2 Debug your software using the Application Debugger

To execute the program in debugging mode, select *Debug instead of Run.* The application will start and stop in the program's first instruction, and the IDE automatically changes to Debug view (you can change between Design and Debug view using the top-right buttons).

Breakpoints can be set/unset by double-clicking on the blue bar on the left of the program window.
The top taskbar's resume and step control buttons can control the program flow.
The program variables and memory zones can be monitored on the right-side windows.

6 Memory Initialization from a Binary File

You can debug your applications by initializing part of your system's memory using data from a binary file in your computer. This basic procedure will be used in your future projects to demonstrate your designs. Edit the file ps7_init.tcl as in the figure, where images.bin is the binary file, the first number is the memory address, and the last number is the file size.

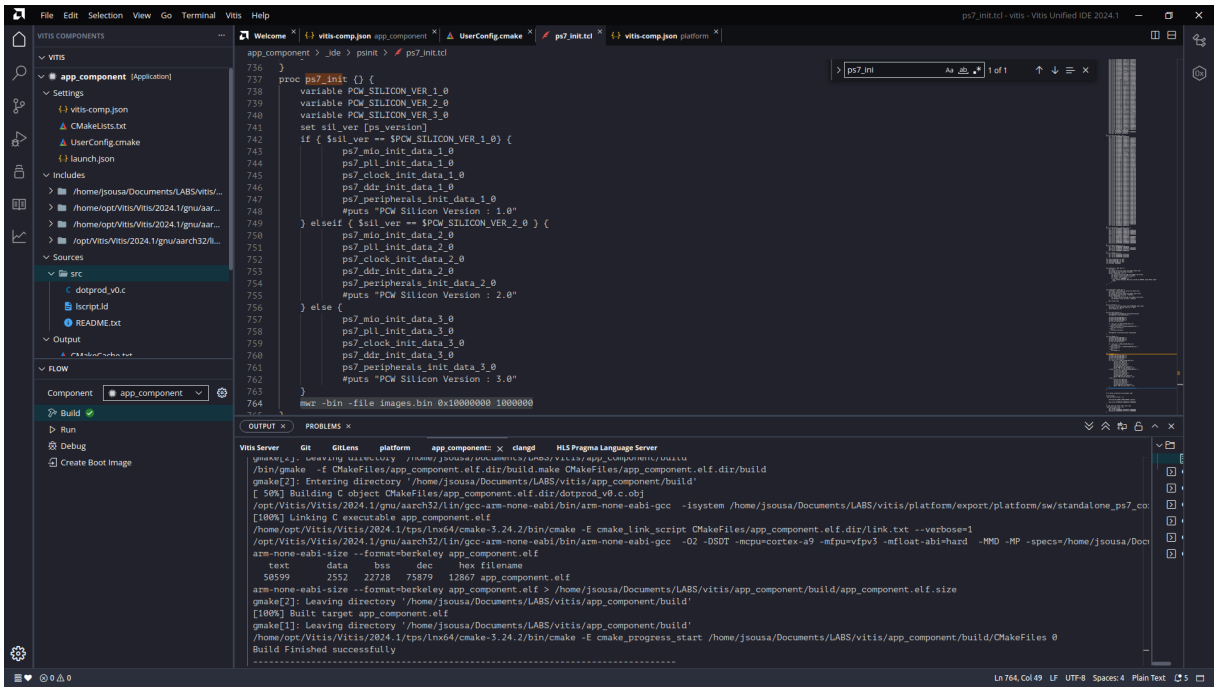


Figure 32: Memory Initialization — editing ps7_init.tcl with binary file, memory address and file size

You must select the memory address zone so that it corresponds to a valid data memory zone in your system and is free to store your data. Choose a free memory zone in your system to store your data. This zone must be adequate for the required storage and not

interfere with the memory zones used for other storage, namely for the program code and data. **Remember that you do the memory management.**

In the figure example, the base address selected, 0x10000000, defines a memory area to be used in the external memory (DDR).

Then and for example, you can set the base address for your vector's storage in your C program, and use it to set constant pointers to address the vectors:

```
#define VEC1_START_ADD 0x10000000
#define VEC2_START_ADD (VEC1_START_ADD+4*VEC_SIZE)
int *v1 = (int *)(VEC1_START_ADD);
int *v2 = (int *)(VEC2_START_ADD);
```

7 Measure the execution time of your application

You can use the time-specific function ***XTime_GetTime (XTime *xtime)*** (available in the standalone BSP) to evaluate the execution time of your application. This function provides direct access to the
64-bit Global Counter in the PMU. This global timer (GT) is a 64-bit incrementing counter with an auto-incrementing feature accessible to both Cortex-A9 processors. The global timer is consistently clocked at 1/2 of the CPU frequency (the counter increments every two processor cycles). The pre-defined value COUNTS_PER_SECOND directly indicates the GC clock frequency (number of counts per second).

```
#include <stdio.h>
#include "xiltimer.h"
int main()
{
XTime tStart, tEnd;
// initialize the application
XTime_GetTime(&tStart); // start measuring time
// process the application
XTime_GetTime(&tEnd); // end measuring time
// finalize the application
printf("Execution took %llu clock cycles.\n",
2*(tEnd - tStart));
printf("Output took %.2f us.\n",
1.0*(tEnd - tStart) * 1000000/(COUNTS_PER_SECOND));
return 0;
}
```

Note: when measuring the performance of (parts of) your application, be careful not to measure the time of the printfs!

8 Course Project

- (CNN) Convolutional Neural Network classification of images.
  http://neuralnetworksanddeeplearning.com/chap6.html
  https://cs.stanford.edu/~acoates/stl10/

  1st part: 3×3 image convolution of images from the STL-10 images data set (88×88 RGB images).
  The elements of the convolution kernel will be 8-bit integers.

  2nd part: classification of images from the STL-10 images data set
  A baseline trained network will be provided (at the Support Material web page).
  The network weights provided will be single-precision floating-point.

8.1 Data input/output

The projects will be demonstrated by initializing the external memory with the appropriate data input and calculating the results.

The design performance will be evaluated by measuring the total processing time of the algorithm, starting from the first read of input data from the external memory and ending when all the resulting data is written back to the external memory.

8.2 Project schedule

The project will be implemented in two phases.

Lab 1: HW / SW co-processing architecture ~ 2 weeks (35%)

Embedded uniprocessor with (simpler) hardware accelerator (using GP).
Must demonstrate application functionality (simpler processing) and verification competencies (HLS IP co-simulation and, optionally, Integrated Logic Analysis), and evaluate performance.

i. Introductory application: implement and evaluate integer convolution 2D using the application and testbenches provided in the classes.

ii. Base application: Compute the 2D convolution for input images with 8-bit integer pixels.

- Milestone 1: Demonstrate a working software-only application.
- Milestone 2: demonstrate HW/SW application using the axil_conv2D IP connected to the GP port, including:

  a. HLS C/RTL co-simulation using simple testbench with 6×6 images and 3×3 kernel

b. HLS C-simulation with the full application as a testbench, with the 88×88 images and 3×3 kernels extended to simulate the use of IP.

c. HW/SW embedded application executing on the Zynq PS/PL system (using the IP and one ARM processor).

iii. (Optional) Simple optimization of the HW/SW application (software or/and hardware component) [2 val]

iv. (Optional) Integrated Logic Analysis [1 val]

Lab 2: Multiprocessor system in an FPGA ~ 3 weeks (50%)

- Embedded (heterogeneous) multiprocessor(s) with hardware accelerator(s). Must demonstrate functionality and show parallelization/hardware acceleration.

References

[1] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, Glasgow, UK, 2014. Free PDF available at http://www.zynqbook.com.

[2] José T. de Sousa. Hardware/software co-design – course slides. Instituto Superior Técnico, Universidade de Lisboa, 2025. Available on the course website.

[3] AMD/Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*. AMD/Xilinx, 2023. Available at https://docs.amd.com/r/en-US/ug998-vivado-intro-fpga-design-hls.

[4] AMD/Xilinx. *Vitis High-Level Synthesis User Guide*. AMD/Xilinx, 2023. Available at https://docs.amd.com/r/en-US/ug1399-vitis-hls.

[5] José T. de Sousa. Lab guides and project files for Lab 0, Lab 1 and Lab 2. Instituto Superior Técnico, Universidade de Lisboa, 2025. Distributed via the course website.