ARISTOTLE UNIVERSITY OF THESSALONIKI

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Lab 1
Introduction to the Vitis HLS Tool

A. Athanasiadis – D. Karanassos

Instructor: Ioannis Papaefstathiou

Version 0.5

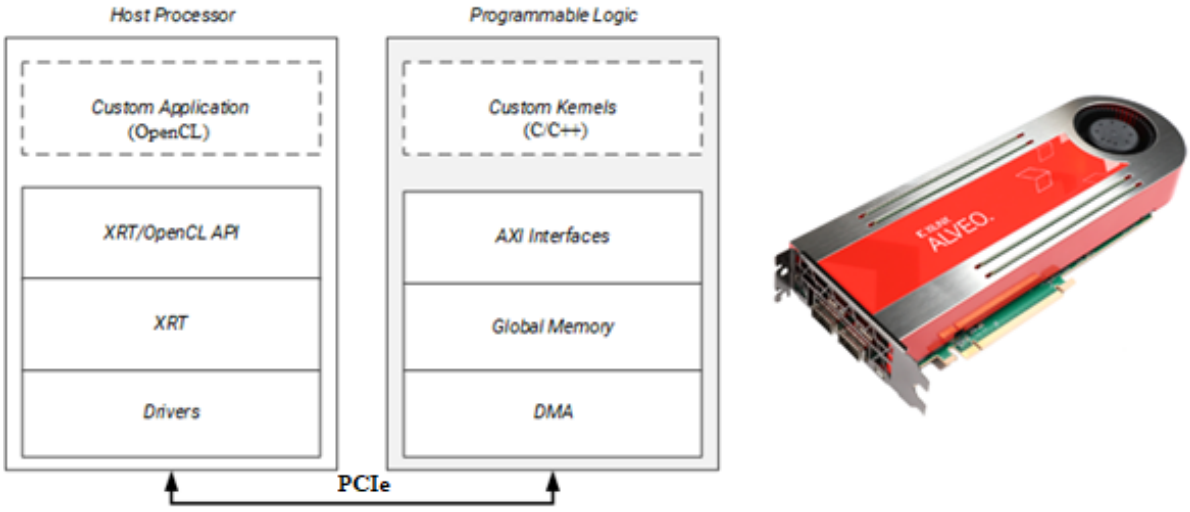
November 2025

1. Introduction

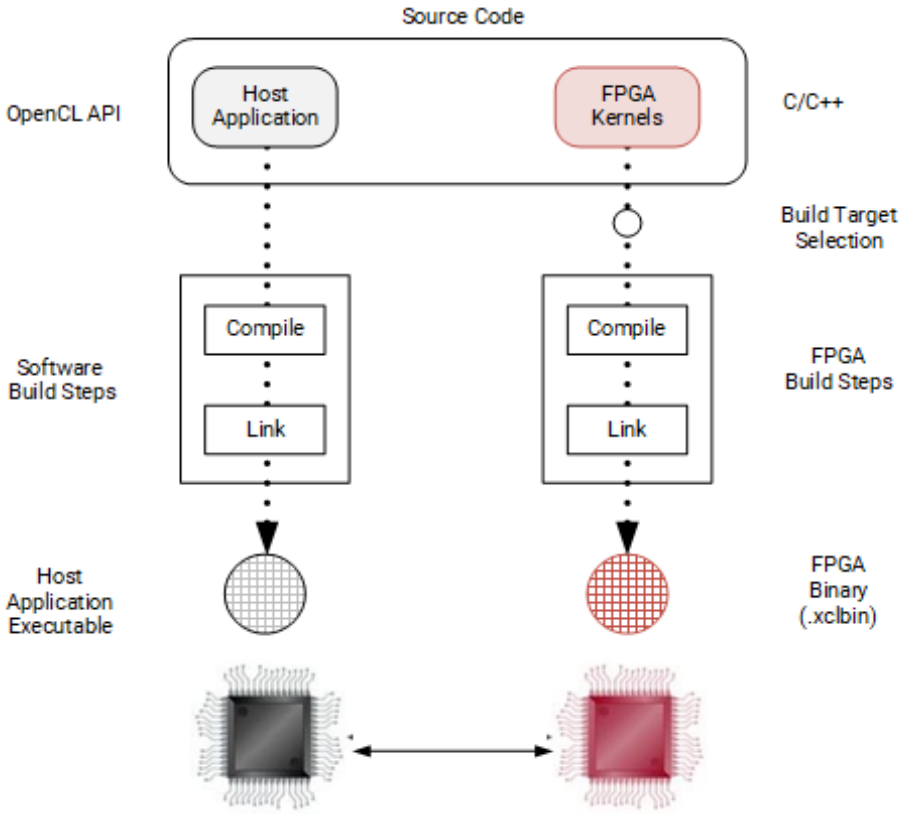
In contrast to the past, where hardware accelerators were mainly designed using Hardware Description Languages (HDLs), today they are also designed using High-Level Synthesis (HLS) methodologies. HLS design refers to the use of high-level programming languages (such as C/C++) for the description of digital circuits. The main reasons for adopting HLS methodologies include reduced development time, high quality of the generated designs, and the great flexibility in managing and modifying an initial design in order to adapt it to new requirements.

After completing the design and simulation of a hardware accelerator, the next step is its implementation on real hardware (e.g., a board that includes an FPGA SoC), in order to verify that the implemented accelerator meets the original design specifications when realized in hardware. For this purpose, an application is usually developed that runs on the Processing System (PS), i.e., the CPU, and invokes the accelerator implemented in the Programmable Logic (PL), i.e., the FPGA, of the FPGA SoC.

The following figure presents a simplified architecture of the Alveo U200 FPGA, where the main components of the board are shown on the right (Programmable Logic), while on the left (Host Processor) the runtime system (Xilinx Runtime – XRT, drivers, etc.) is depicted, enabling communication between the Host (the processor, i.e., the PS) and the FPGA board. More specifically, the application is divided into a part that runs on the Host side (Custom Application) and a part that runs on the FPGA for accelerator execution (Custom Kernel). These two parts are connected via a PCIe communication channel. The host program is written in OpenCL and runs on a processor of x86 or Arm architecture, while the acceleration kernels are executed in the programmable logic (PL) of the Alveo U200.



In the following image, the execution flow of the application is illustrated (the general-purpose processor is shown on the left and the FPGA on the right). Additional information can be found at the corresponding documentation[1].



In this laboratory, we will focus on the design of a hardware accelerator using the Vitis HLS tool. Initially, a sequence of steps will be presented, aiming at an introductory familiarization with Vitis HLS, which supports circuit design targeting Xilinx FPGA SoC platforms. Subsequently, you will be asked to design and optimize your own accelerator. The final outcome of this laboratory will serve as the starting point for developing the application during the software design phase of the system in the following laboratories.

[1] https://docs.xilinx.com/v/u/2022.2-English/ug1416-vitis-documentation

2. Tool Versions

The tools that will be used are provided by Xilinx, and the laboratory will be based on Vivado version 2022.2 (which may also be referred to as Vitis HLS). Vitis HLS[2] is a sub-tool that supports the creation of custom accelerators implemented on FPGAs using high-level programming languages. Specifically, it supports development in C, C++, SystemC, and OpenCL. More details regarding Vitis HLS can be found in the corresponding User Guide[3]. t is recommended to download the complete Xilinx tool package, Vitis 2022.2. The installer is available for both Windows and Linux platforms[4]:

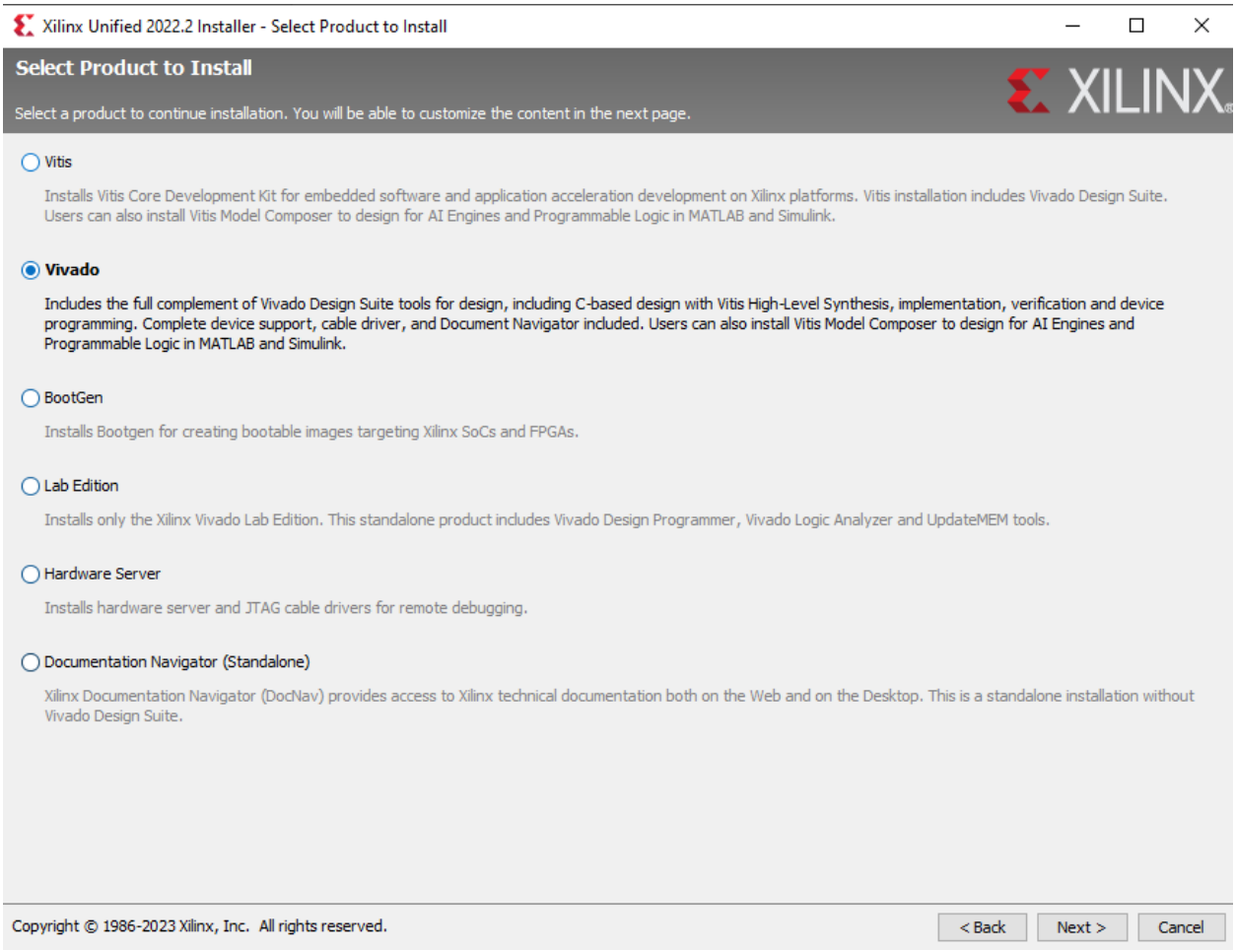
- Xilinx Unified Installer 2022.2: **Windows** Self Extracting Web Installer
- Xilinx Unified Installer 2022.2: **Linux** Self Extracting Web Installer

During installation, select the Vivado Suite (as shown in the following figure), since Vitis will be used via a laboratory server where it is already installed (you do not have a license to use it locally on your own computer).

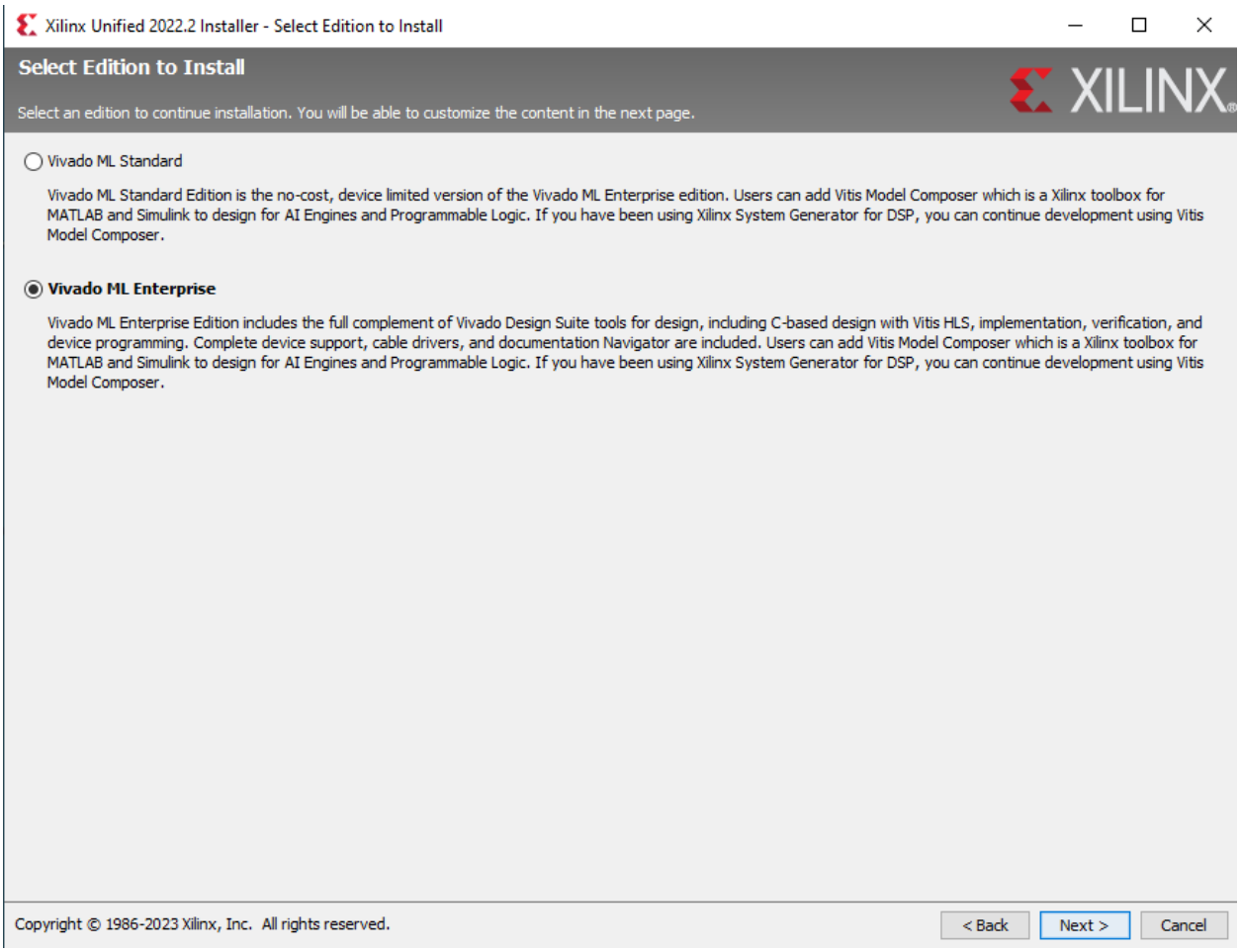
[2] https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

[3] https://docs.xilinx.com/r/2022.2-English/ug1399-vitis-hls/Introduction

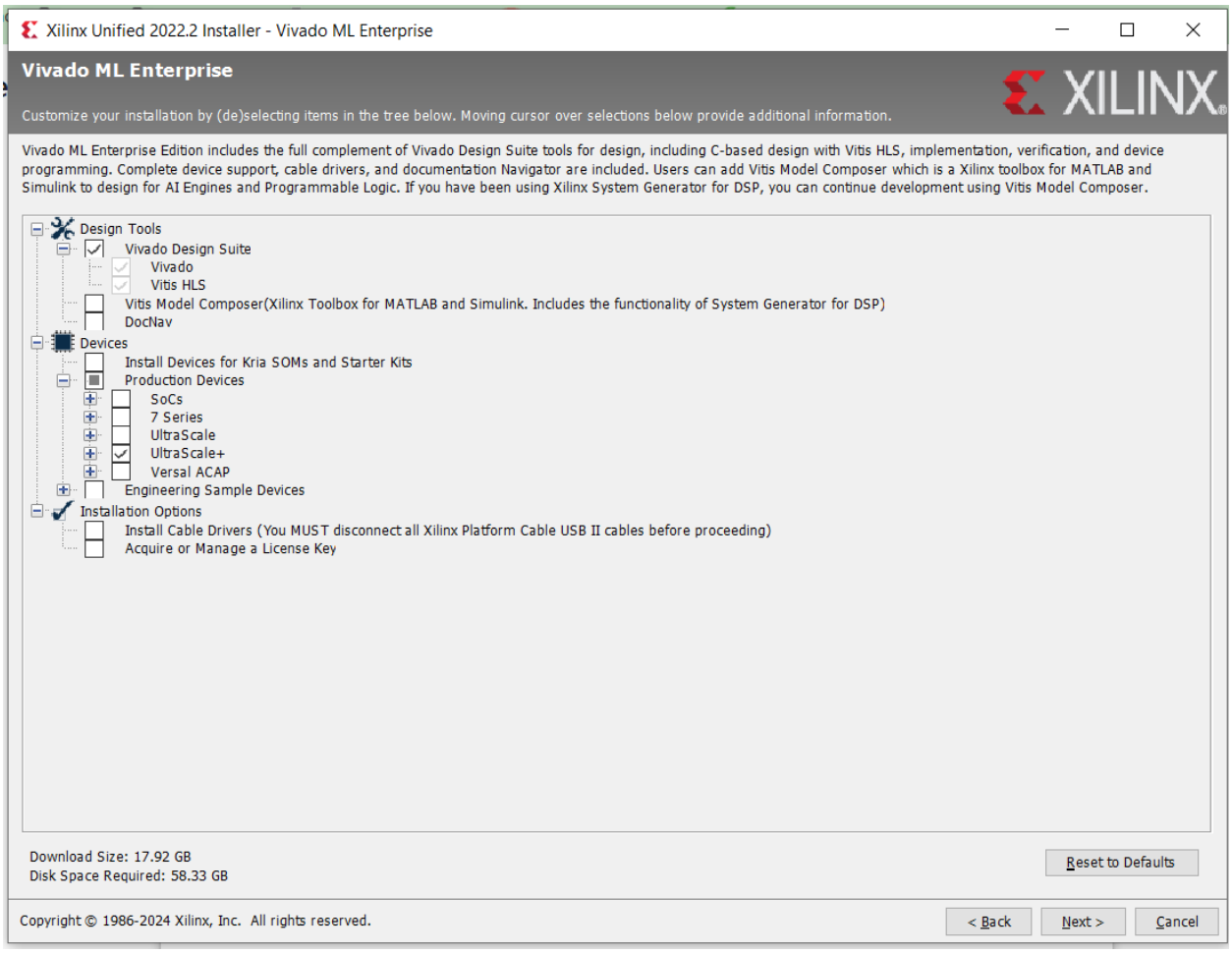
[4] https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/ -design-tools/archive.html



Next, select the installation of the full tool version (Vivado ML Enterprise), which includes support for the FPGA board that will be used in the following laboratories.



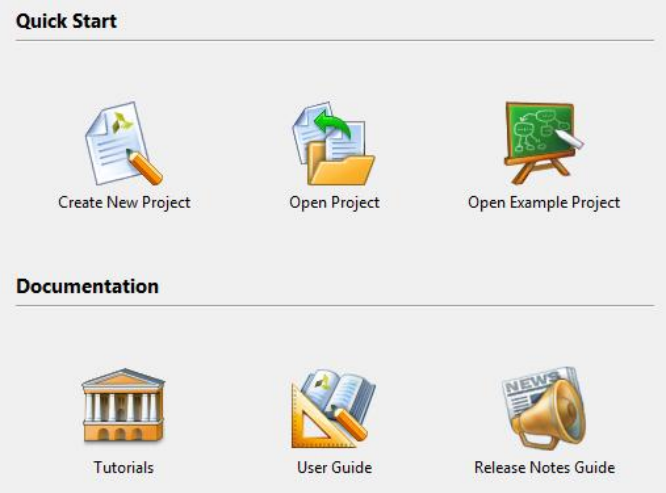
By making the following selections during installation, you can significantly reduce the disk space required on your system.



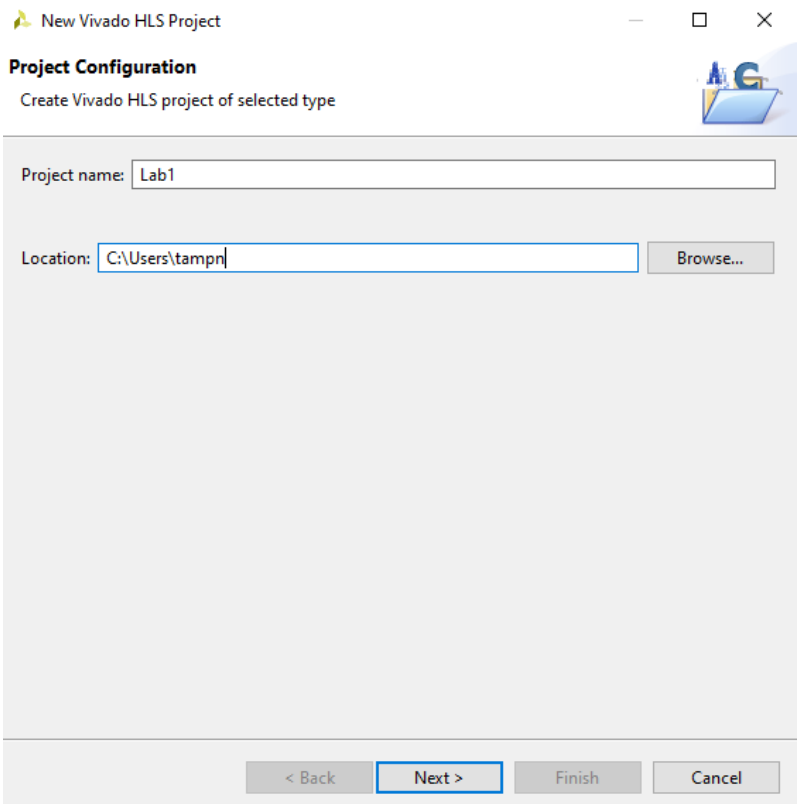
For the Alveo U200 board, download the file au200 (from the e-learning platform) and extract it into the following directory:
C:\xilinx\Vivado\2022.2\data\xhub\boards\XilinxBoardStore\boards\Xilinx\

3. Introduction to Vitis HLS (20%)

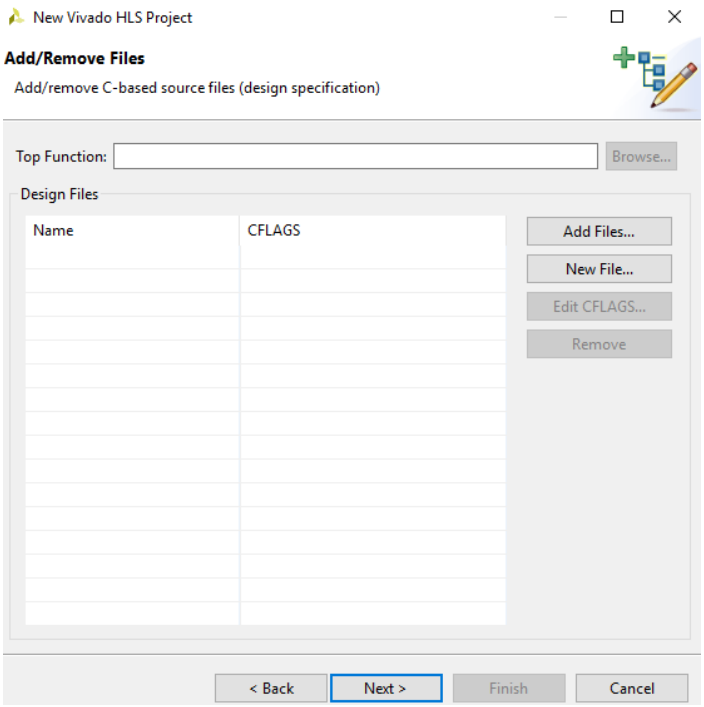
Now it is time to start Vitis HLS. In the main window, select ***Create New Project***.



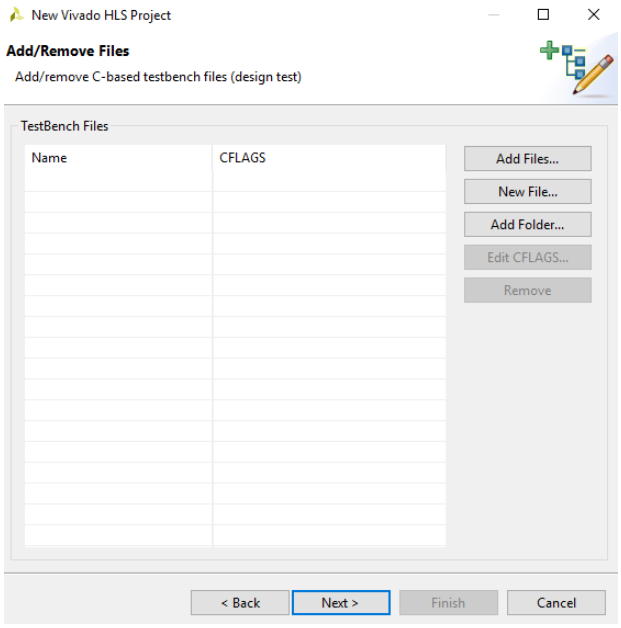
In the next step, choose the project name and storage location. After filling in the required information, click ***Next***.



The following window prompts you to specify existing source files related to the design you want to create. In our case, no such files exist, so skip this step and click ***Next***.

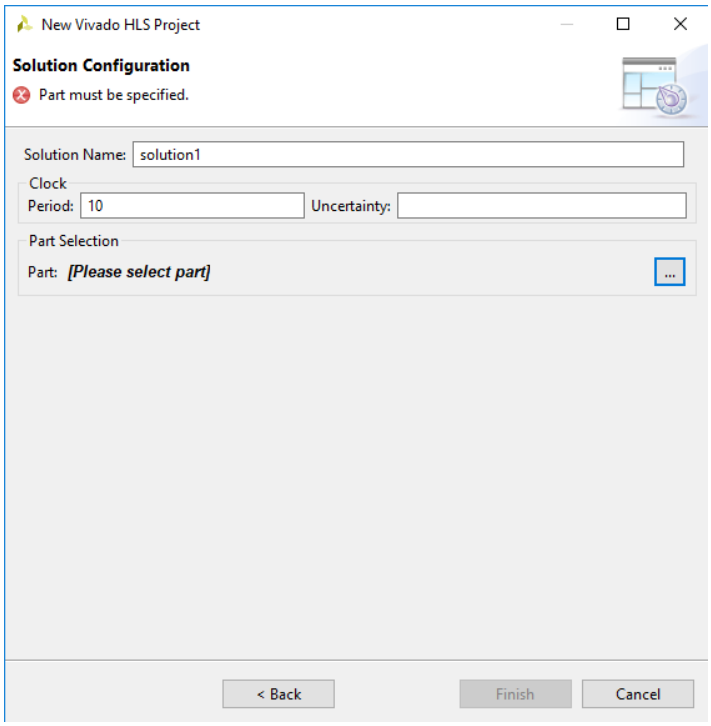


A similar purpose applies to the next step, where you are asked to define testbench files and data files that will serve as golden references for verification. Again, do not specify any files and proceed by clicking ***Next***.

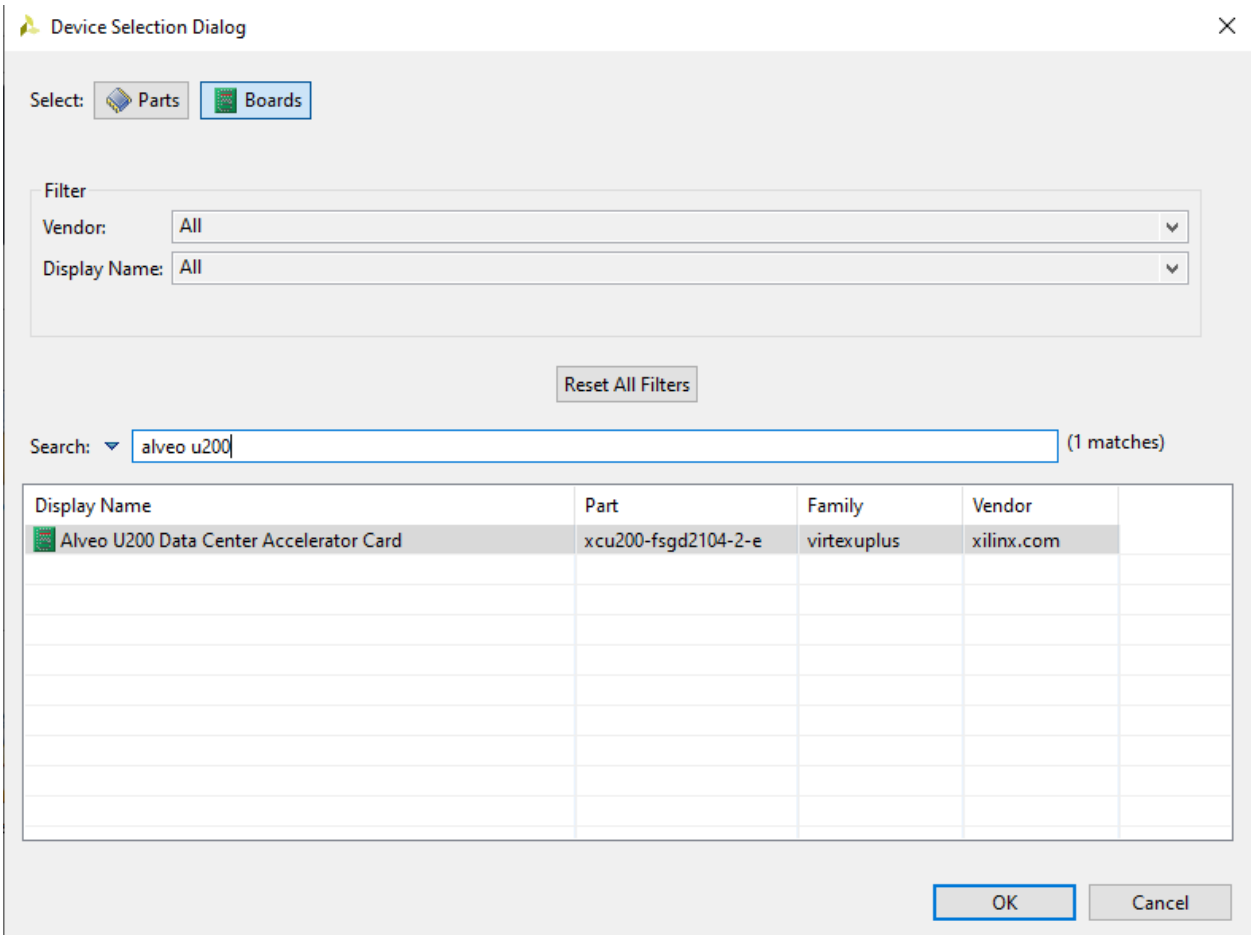


In the next window, you must specify several parameters. First, accept the default solution name (a solution represents a specific configuration and set of implementation parameters—you can have multiple solutions within the same project).

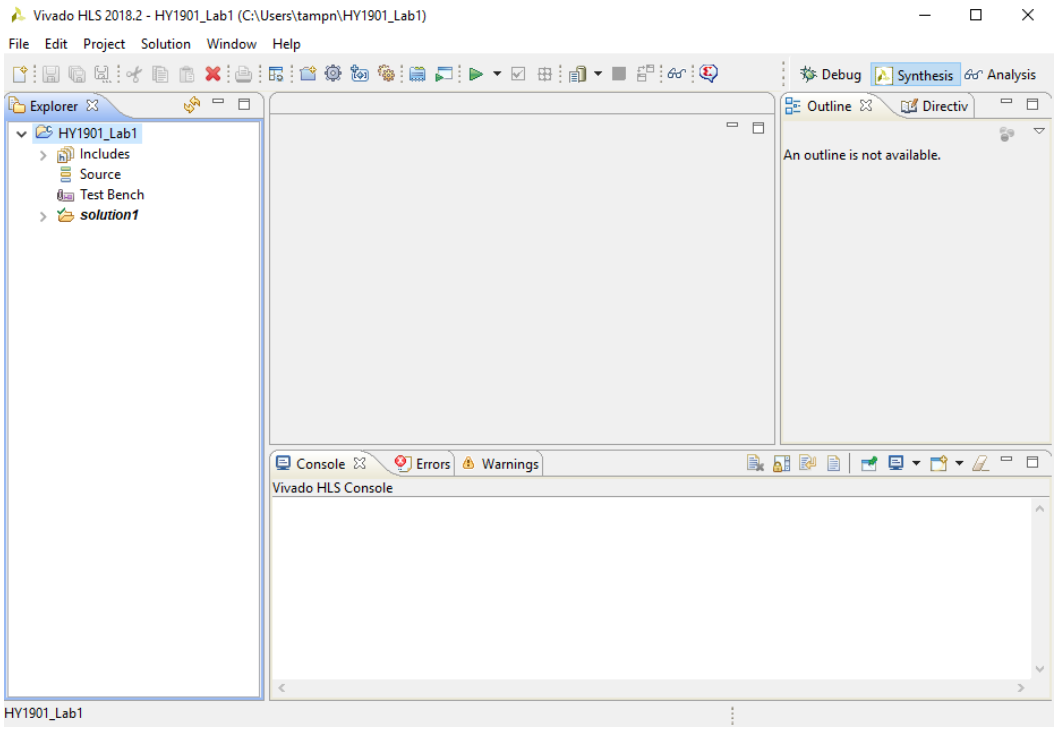
Here, you also select the target clock period for your implementation, as well as the uncertainty margin (if not specified, the default is 12.5% of the clock period).



Finally, under Board Selection, choose the Alveo U200 board, as shown in the next figure.



Once the platform selection is complete, click **OK**. This completes the project creation process and opens the Vitis HLS workspace.

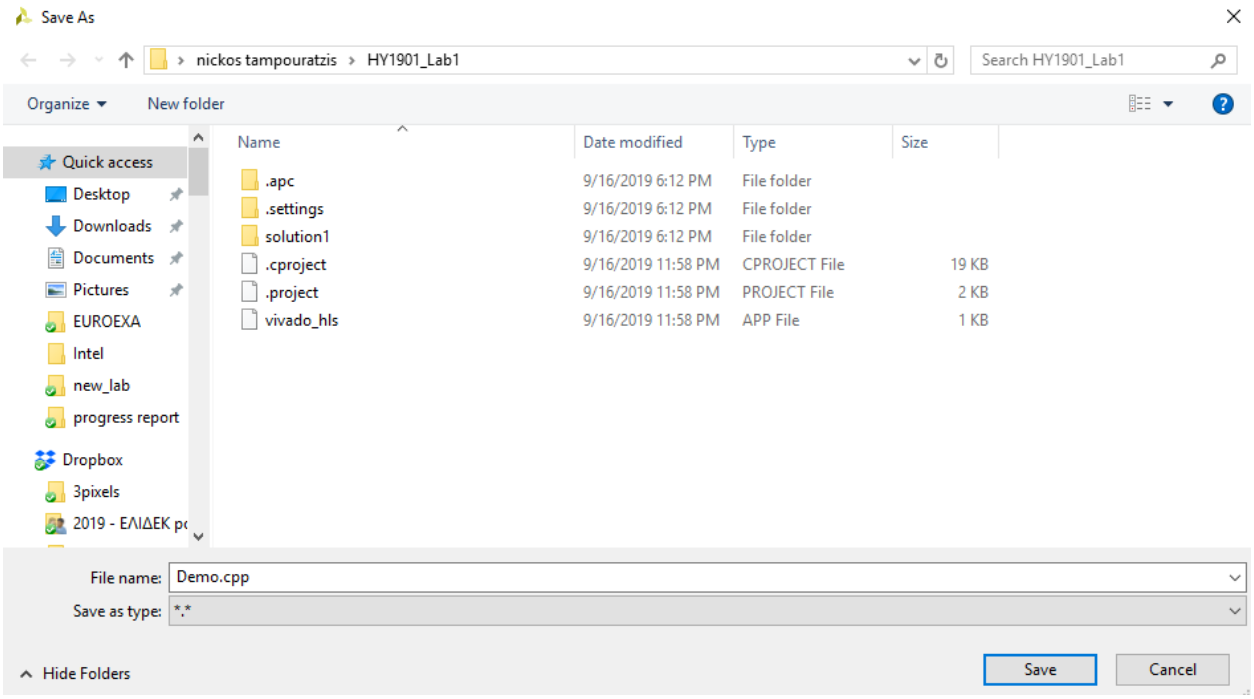


At this point, several things can be observed. The project name appears at the top of the Explorer panel. Vitis HLS organizes project information hierarchically, including **source** code, **testbench** files, and different **solutions**.

Each **solution** contains information related to the target platform, design directives, and results (simulation, synthesis, IP export, etc.).

The first step is to **right-click** on **Source** and select **New File**. In the window that appears, specify the file name, location, and type. In our case, the file type will be .cpp (C++). The location can remain the default, and you may choose any name you prefer.

Click **Save**.



This process adds the file Demo.cpp to the project sources and opens it automatically. The file is initially empty, and you must describe the functionality that you wish to implement in hardware. For this introductory section, the target functionality is shown below and must be implemented in your source file.

Demo.cpp

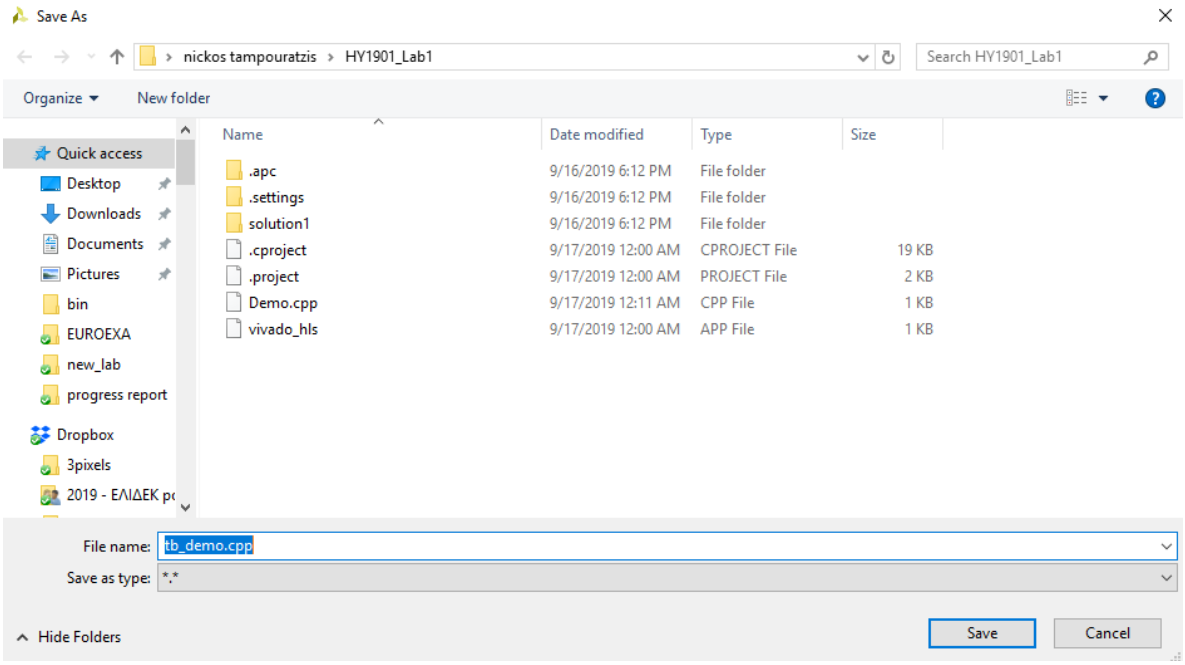
```
1  void simpleALU(int A, int B, int op, int *C){
2      switch (op){
3          case 0:{
4              *C = A + B;
5              break;
6          }
7          case 1:
8          {
9              *C = A - B;
10             break;
11         }
12         case 2:
13         {
14             *C = A * B;
15             break;
16         }
17         case 3:
18         {
19             *C = A / B;
20             break;
21         }
22     }
23
24 }
```

In this case, we implement a simple ALU (Arithmetic Logic Unit) that supports addition, subtraction, multiplication, and division between two input variables, producing the result in a third variable.

The next step is to create a testbench file in order to verify the functionality of the accelerator. To do so, right-click on **Test Bench** in the Explorer panel and select **New File**.

In the window that appears, specify the testbench file name, type (e.g., C or C++), and storage location. To ensure that the tool detects it automatically, save it in the project directory. A common naming convention is to use the same name as the source file, prefixed with tb.

Click **Save**.



The testbench file opens automatically. Inside this file, you must write code that allows verification of the accelerator functionality. For this specific project, you may use **the following code**.

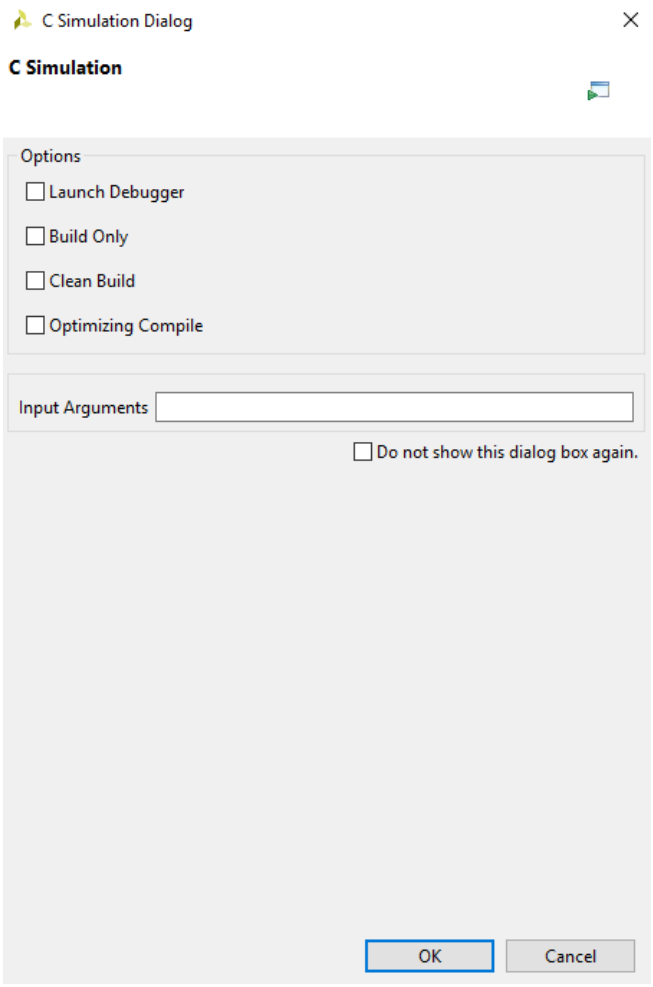
tb_demo.cpp

```
#include <stdio.h>

void simpleALU(int A, int B, int op, int *C);

int main(){

    int A,B,C;
    A=2;
    B=3;
    simpleALU(A,B,0,&C);
    printf("A(%d) + B(%d) = %d\n", A,B,C);

    A=7;
    B=5;
    simpleALU(A,B,1,&C);
    printf("A(%d) - B(%d) = %d\n", A,B,C);

    A=10;
    B=2;
    simpleALU(A,B,2,&C);
    printf("A(%d) * B(%d) = %d\n", A,B,C);

    A=50;
    B=5;
    simpleALU(A,B,3,&C);
    printf("A(%d) / B(%d) = %d\n", A,B,C);

    return 0;

}
```

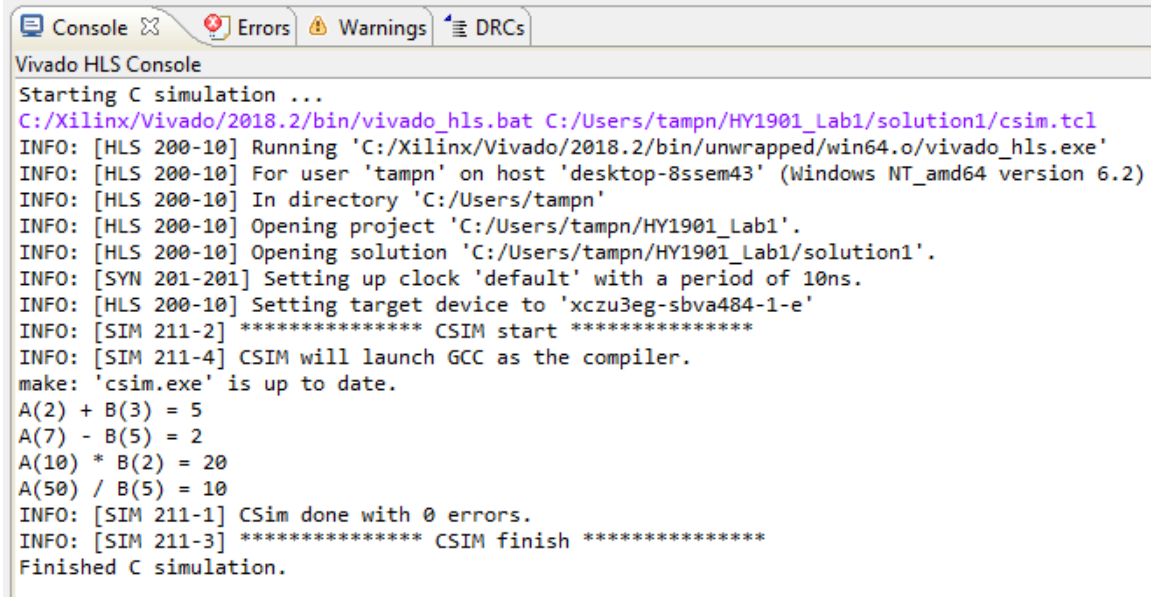
The testbench code allows you to test all four operations supported by the hardware you will create.

The next step in the design flow is to run an initial simulation at the algorithmic level, in order to verify that the expected functionality is achieved.

To do this, click **Run C Simulation** or use the menu **Project > Run C Simulation**. This action compiles the code and executes it as a software program. The following window appears when selecting Run C Simulation; for now, you can ignore the additional options and click **OK**.



After completion, the simulation either succeeds or fails. In the Console panel, you can examine the printf outputs from the testbench and check for warning or error messages.

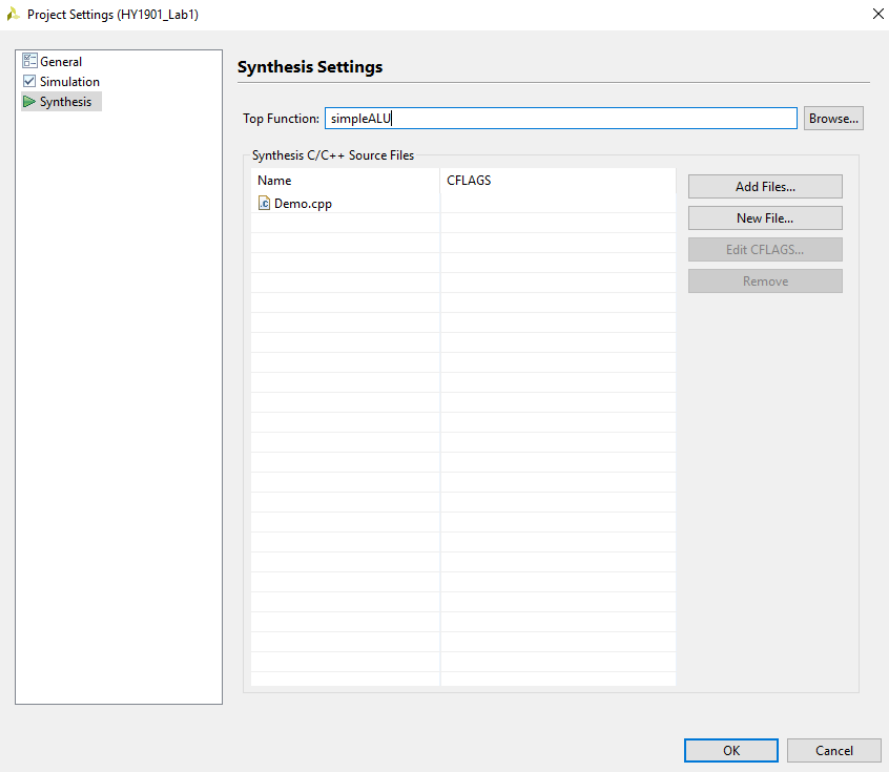


If desired, you can enable debugging in the simulation options and use the debugger to resolve any issues.

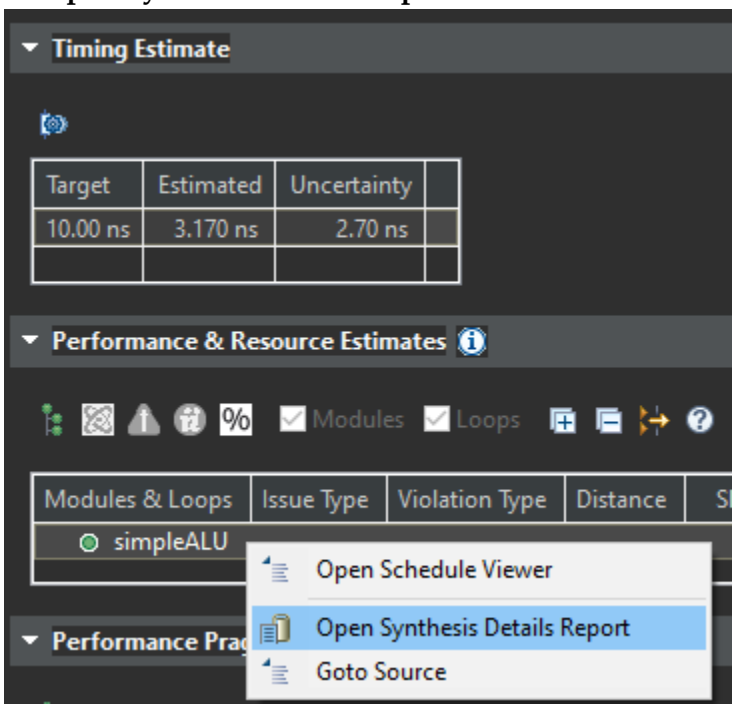
If the process completes successfully, the design is now ready for synthesis. During synthesis, the high-level C++ description of the accelerator is converted into a lower-level RTL description in VHDL, Verilog, and SystemC.

Before proceeding, ensure that the tool knows which function is the top-level function. In this project, the only function is simpleALU, which must be specified accordingly.

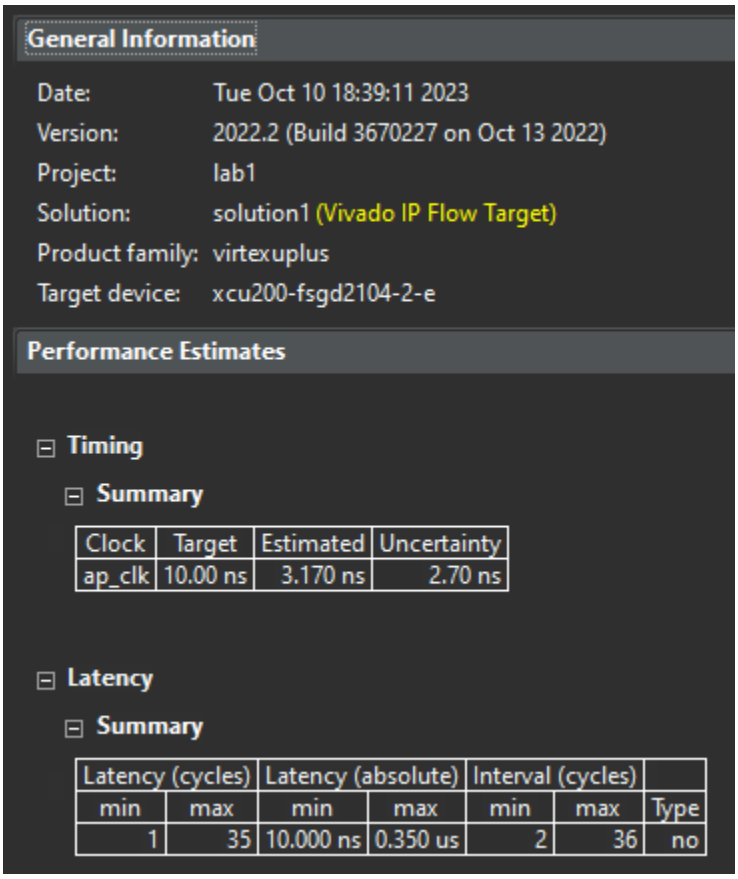
Select **Project > Project Settings**, and in the **Synthesis** section specify the correct Top Function, then click **OK**.



Next, select **Run C Synthesis** from the toolbar or use **Solution > Run C Synthesis**. Upon successful completion, the Synthesis Summary file opens automatically. Right-click on **simpleALU** and select **Open Synthesis Details Report.**



By examining this report, you gain a first insight into the hardware characteristics of your implementation. Take a few minutes to observe the key features of the design.



General Information

Date:	Tue Oct 10 18:39:11 2023
Version:	2022.2 (Build 3670227 on Oct 13 2022)
Project:	lab1
Solution:	solution1 (Vivado IP Flow Target)
Product family:	virtexuplus
Target device:	xcu200-fsgd2104-2-e

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	3.170 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1	35	10.000 ns	0.350 us	2	36	no

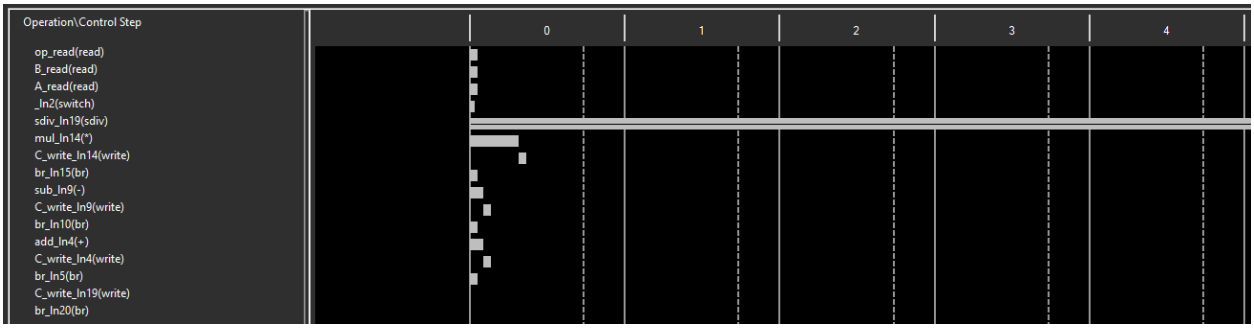
Εδώ μπορούμε να παρατηρήσουμε έναν αριθμό από πράγματα όπως:

You can observe that the target clock period has been met, since the estimated period plus uncertainty does not exceed the specified 10 ns.

The minimum latency is one clock cycle, while the maximum latency is 35 cycles. Using the analysis tools, you can determine which operation (addition, subtraction, etc.) corresponds to each latency.

The initiation interval indicates when the next input can be accepted. In the best case, this occurs after one clock cycle, while in the worst case after 35 cycles. Therefore, the design is not pipelined, since a new transaction can only begin after the previous one is completed. Typically, this behavior is not desirable and motivates further optimization, which will be addressed in the next part of the laboratory.

To identify the source of the maximum latency, select the solution and open the **Schedule Viewer**.



You will observe that addition, subtraction, and multiplication require one clock cycle, whereas division requires 35 cycles.

Return to the **Synthesis Summary**.

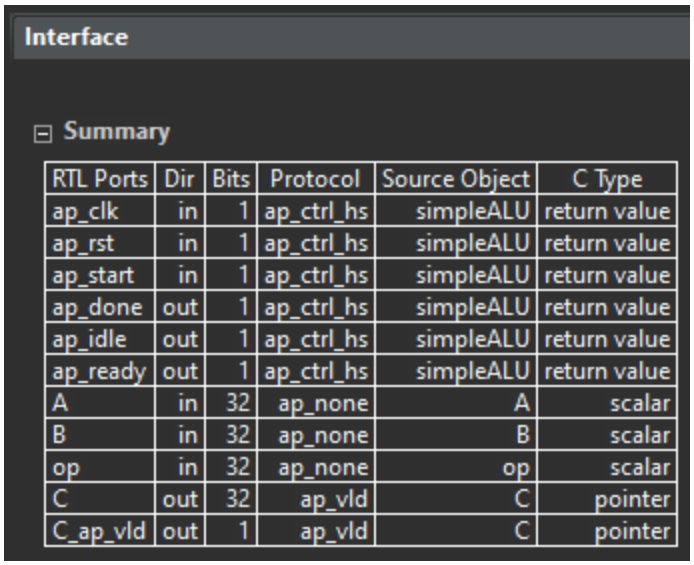
Examine the Utilization Estimates section, which provides an initial estimation of FPGA resource usage. The most important resources are LUTs, FFs, DSPs, and BRAMs.

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	78	-
FIFO	-	-	-	-	-
Instance	-	3	394	258	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	192	-
Register	-	-	36	-	-
Total	0	3	430	528	0
Available	4320	6840	2364480	1182240	960
Available SLR	1440	2280	788160	394080	320
Utilization (%)	0	~0	~0	~0	0
Utilization SLR (%)	0	~0	~0	~0	0

Keep in mind that these values are estimates and may change during RTL-level synthesis in Vitis (Laboratory 2). Note that the U200 card contains three Super Logic Regions (SLRs). An SLR is the smallest unified FPGA within the U200. In this laboratory, the implementation will target a single SLR; therefore, the Utilization SLR (%) metric is of interest. Utilizing the full U200 requires advanced knowledge.

Finally, scroll down to examine the interface summary of your design.



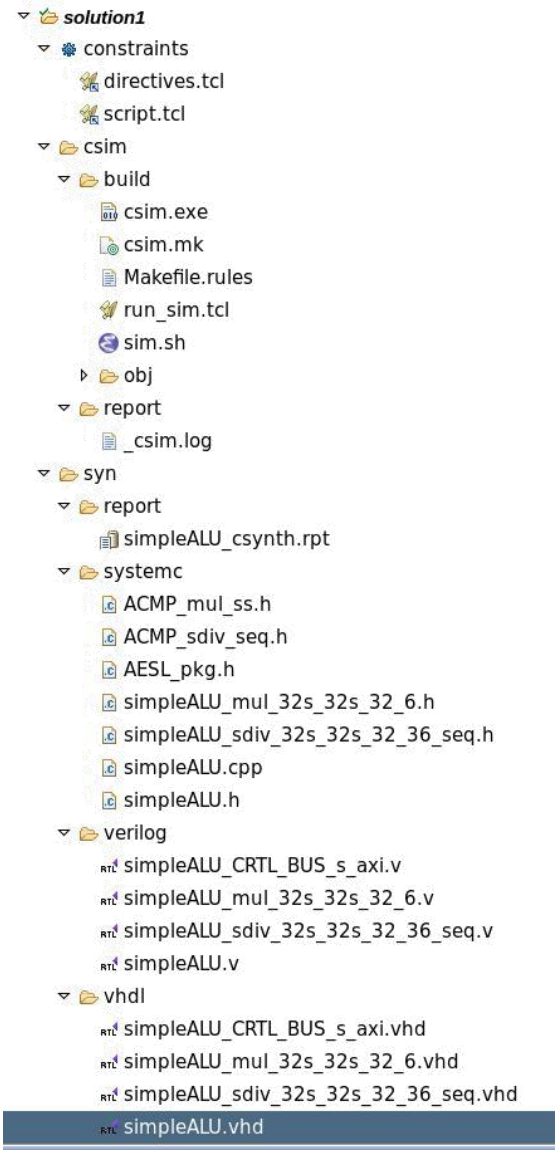
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	simpleALU	return value
ap_rst	in	1	ap_ctrl_hs	simpleALU	return value
ap_start	in	1	ap_ctrl_hs	simpleALU	return value
ap_done	out	1	ap_ctrl_hs	simpleALU	return value
ap_idle	out	1	ap_ctrl_hs	simpleALU	return value
ap_ready	out	1	ap_ctrl_hs	simpleALU	return value
A	in	32	ap_none	A	scalar
B	in	32	ap_none	B	scalar
op	in	32	ap_none	op	scalar
C	out	32	ap_vld	C	pointer
C_ap_vld	out	1	ap_vld	C	pointer

Here, you can observe the signals and protocols used to interface the accelerator with the processing system.

The design includes clock and reset signals associated with the simpleALU source object. Additional control signals are automatically inserted by the tool during synthesis.

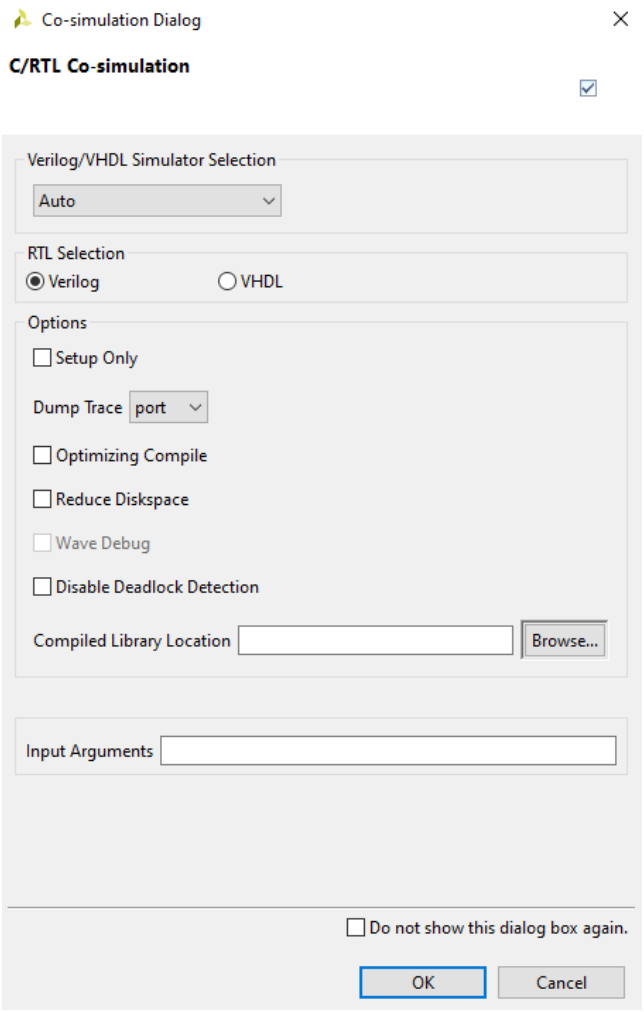
Inputs A, B, and op, as well as output C, are served via 32-bit buses. The output includes an additional valid signal (C_ap_vld) indicating when output data is valid. Currently, input and output communication is implemented without a specific I/O protocol (ap_none).

By exploring the solution1 directory, you can find all generated synthesis files in VHDL, Verilog, and SystemC, allowing you to study the automatically generated RTL code.



Now it is time to verify the RTL code generated during the synthesis stage. At this point, the previously created testbench file will be used to verify the RTL descriptions, and the results of this process will be compared against those obtained from the earlier C-level simulation.

Select **Run C/RTL Co-simulation** from the toolbar, or alternatively use the **menu Solution > Run C/RTL Co-simulation**. The following window will appear.



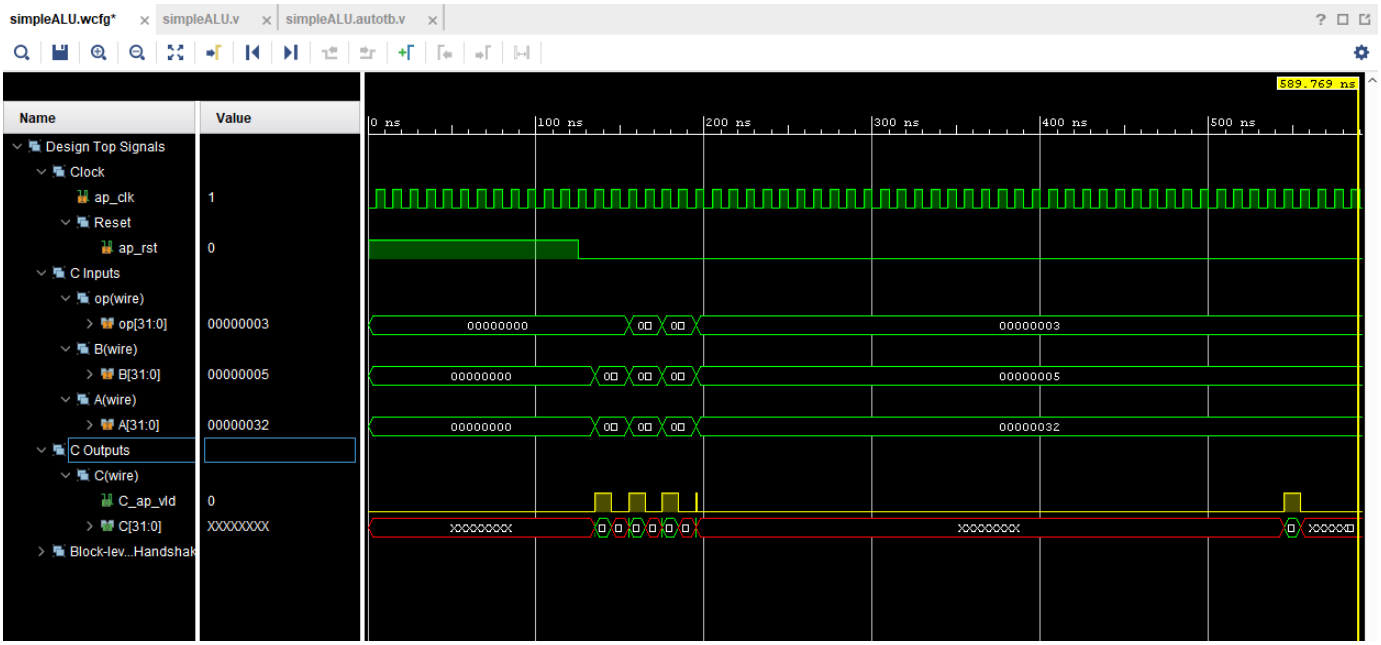
In this window, the tool provides several configuration options, such as selecting the simulator to be used and choosing whether the Verilog or VHDL RTL description will be employed during the verification process. Another important option is the ability to store signal-level activity, allowing visual inspection of the simulation behavior. Make sure that the **port** option is enabled in the **Dump Trace** field.

```
$finish called at time : 615 ns : File "C:/Users/tampn/AppData/Roa
## quit
INFO: [Common 17-206] Exiting xsim at Tue Oct 10 19:50:11 2023...
INFO: [COSIM 212-316] Starting C post checking ...
A(2) + B(3) = 5
A(7) - B(5) = 2
A(10) * B(2) = 20
A(50) / B(5) = 10
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

Upon successful completion of this step, the system is ready for time-based behavior analysis, based on the previously defined testbench. In the above figure, the total execution time of the

design can also be observed. For more detailed analysis, select **Open Wave Viewer**… from the toolbar, or navigate through **Solution > Open Wave Viewer**….

After a short delay, Vivado opens the waveform viewer displaying the simulation results. Spend some time examining how the simulation was executed. Since the implemented function is relatively simple, you should be able to thoroughly understand the depicted behavior.



Once finished, make sure to close Vivado by selecting **File > Exit**.

4. Design of an H/W Accelerator Using Vitis HLS (80%)

Using Vitis HLS, design the hardware accelerator IMAGE_DIFF_POSTERIZE, which accepts as input two grayscale images (arrays) A and B of dimensions HEIGHT × WIDTH and produces an output image C of the same dimensions.
The arrays A, B, and C can be interpreted as grayscale images, where each element represents the brightness of a pixel in the range 0–255.
The purpose of the accelerator is to highlight the differences between the two images by applying a simple non-linear posterization transformation to the absolute difference of the pixel values.

Functionality of the IMAGE_DIFF_POSTERIZE Accelerator

Each element of arrays A, B, and C is an 8-bit unsigned integer (unsigned char or uint8_t) in the range 0–255.

For each position (i, j), we define:

- A[i][j]: brightness value from the first image
- B[i][j]: brightness value from the second image
- C[i][j]: output value

First, the absolute difference between the two pixels is computed:

$$D(i,j)=|A[i][j]-B[i][j]|$$

Then, posterization is applied using two thresholds T1 and T2, with 0 ≤ T1 < T2 ≤ 255:

- Αν $D(i,j) < T1$, then $C[i][j] = 0$
- Αν $T1 \leq D(i,j) < T2$, then $C[i][j] = 128$
- Αν $D(i,j) \geq T2$, then $C[i][j] = 255$

Thus:

- Regions where the two images are nearly identical (small difference) appear black (0).
- Regions with moderate differences appear in mid-gray (128).
- Regions with large differences appear white (255).

Visually, the output image C can be considered a “change map” between images A and B.

Assumptions – Definitions

- Each element of arrays A, B, and C is of type unsigned char or uint8_t.
- The image dimensions are defined as:
  - WIDTH = 2^W, where W is an integer such that 1 ≤ W ≤ 9 (e.g., WIDTH = 2^8 = 256)
  - HEIGHT = 2^H, where H is an integer such that 1 ≤ H ≤ 9 (e.g., HEIGHT = 2^8 = 256)

- You may represent the images either as two-dimensional arrays: A[HEIGHT][WIDTH], B[HEIGHT][WIDTH], C[HEIGHT][WIDTH]
  or as one-dimensional arrays:
  A[HEIGHT * WIDTH], B[HEIGHT * WIDTH], C[HEIGHT * WIDTH]
  using row-major ordering. The chosen representation must be consistent between the hardware top-level function and the testbench.
- The thresholds T1 and T2 may be defined as constants (e.g., using #define) or passed as input parameters to the top-level function, provided that they are identical in both the hardware and software reference implementations used in the testbench.
- For the purposes of this laboratory, assume fixed values T1 = 32 and T2 = 96, defined for example as: #define T1 32 #define T2 96. You may experiment with different values, as long as the same values are used in both the hardware and software implementations.
- You must not use scanf() or any other keyboard input mechanism for initialization. The values of arrays A and B must be generated within the testbench (e.g., pseudo-random values or simple functions of i and j).

Question 1 (30%)

Write C code for the function IMAGE_DIFF_POSTERIZE in Vitis HLS according to the above definitions and store it in the source section of Vitis HLS.

The function must:

a) Accept as input two arrays A and B of identical dimensions (HEIGHT × WIDTH).
b) Produce as output an array C of the same dimensions.
c) For each element (i, j), compute:
   i. The absolute difference $D(i,j)=|A[i][j]-B[i][j]|$ using an appropriate intermediate type (e.g., int16_t or int) to avoid overflow during subtraction.
   ii. The output value C[i][j] based on thresholds T1 and T2:
      1. if $D(i,j) < T1 \rightarrow$ C[i][j] = 0
      2. if $T1 \leq D(i,j) < T2 \rightarrow$ C[i][j] = 128
      3. if $D(i,j) \geq T2 \rightarrow$ C[i][j] = 255

In addition, write a C testbench and add it to Vitis HLS, which:

a) Initializes arrays A and B with values in the range 0–255. For example, you may use:
   i. A simple gradient with respect to i and j (e.g., A[i][j] = f(i, j), B[i][j] = g(i, j)), or
   ii. Pseudo-random values (with a fixed seed for reproducibility).
b) Computes the results in two ways:
   i. A purely software reference implementation (a C function) that implements exactly the same logic.
   ii. The hardware implementation through the top-level function IMAGE_DIFF_POSTERIZE in Vitis HLS.

c) Compares the results element-by-element (for all C[i][j]) and prints a clear message, e.g., "Test Passed", if all elements match, or an appropriate error message if discrepancies are detected.

Optionally, you may print a small subset of the image (e.g., the first 8×8 elements) or statistics (e.g., how many pixels have values 0, 128, and 255).

Question 2 (5%)

Perform C synthesis of the above design (without adding HLS directives, except those required for the interface) using default settings and fill in the following fields (for a specific choice of dimensions, e.g., HEIGHT = WIDTH = 2^8 = 256):

Estimated clock period: _____
Worst case latency: _____
Number of DSP48E used: _____
Number of BRAMs used: _____
Number of FFs used: _____
Number of LUTs used: _____

In your report, clearly state the array dimensions used.

Question 3 (5%)

Run C/RTL co-simulation and verify that your design passes the test successfully. Fill in the following fields (for the same choice of dimensions, e.g., HEIGHT = WIDTH = 256):

Total Execution Time: _____
Min latency: _____
Avg. latency: _____
Max latency: _____
Use the appropriate Vitis HLS reports (e.g., Co-simulation Report, Performance Estimates) to identify these values.

Question 4 (40%)

Apply Vitis HLS directives (e.g., ARRAY_PARTITION, PIPELINE, UNROLL) to optimize the execution time[5]. Information about directives can be found in the official documentation[6]. Experiment with different array dimensions and answer the following:
i) Keeping one dimension fixed (e.g., HEIGHT) and varying the other (e.g., WIDTH = 64, 128, 256, 512), describe what you observe regarding:

- Total latency (in clock cycles).
- Total execution time.

[5] If needed you can use TRIPCOUNT pragma to define the number of iterations.
[6] https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis

- Any other observation you found noteworthy.

ii) After experimenting with various combinations of directives (e.g., applying #pragma HLS PIPELINE inside the element-scanning loop, possibly using UNROLL at an appropriate level, and/or ARRAY_PARTITION on arrays if deemed beneficial), select the best implementation (according to your judgment) for the case HEIGHT = WIDTH = 256.
For this optimal implementation, fill in the following:

Estimated clock period: _____
Number of DSP48E used: _____
Number of BRAMs used: _____
Number of FFs used: _____
Number of LUTs used: _____
Total Execution Time: _____
Min latency: _____
Avg. latency: _____
Max latency: _____

and briefly describe which directives you used (e.g., where PIPELINE was applied, whether UNROLL was used and with which factor, whether ARRAY_PARTITION was applied, etc.) and why you believe they improved performance.

iii) Finally, compute the speed-up of your optimal hardware implementation compared to the initial hardware design without directives.