ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδίαση Συστημάτων Υλικού - Λογισμικού

Εργαστήριο 1
Εξοικείωση με το εργαλείο Vitis HLS

Α. ΑΘΑΝΑΣΙΑΔΗΣ - Δ. ΚΑΡΑΝΑΣΣΟΣ

Διδάσκων: Ιωάννης Παπαευσταθίου

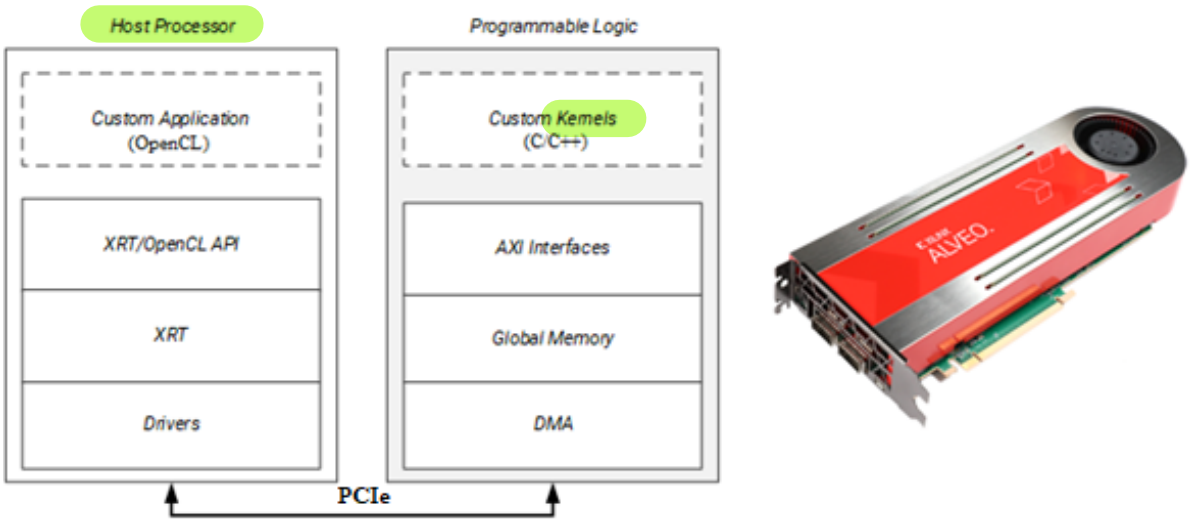
Version 0.5

Νοέμβριος 2025

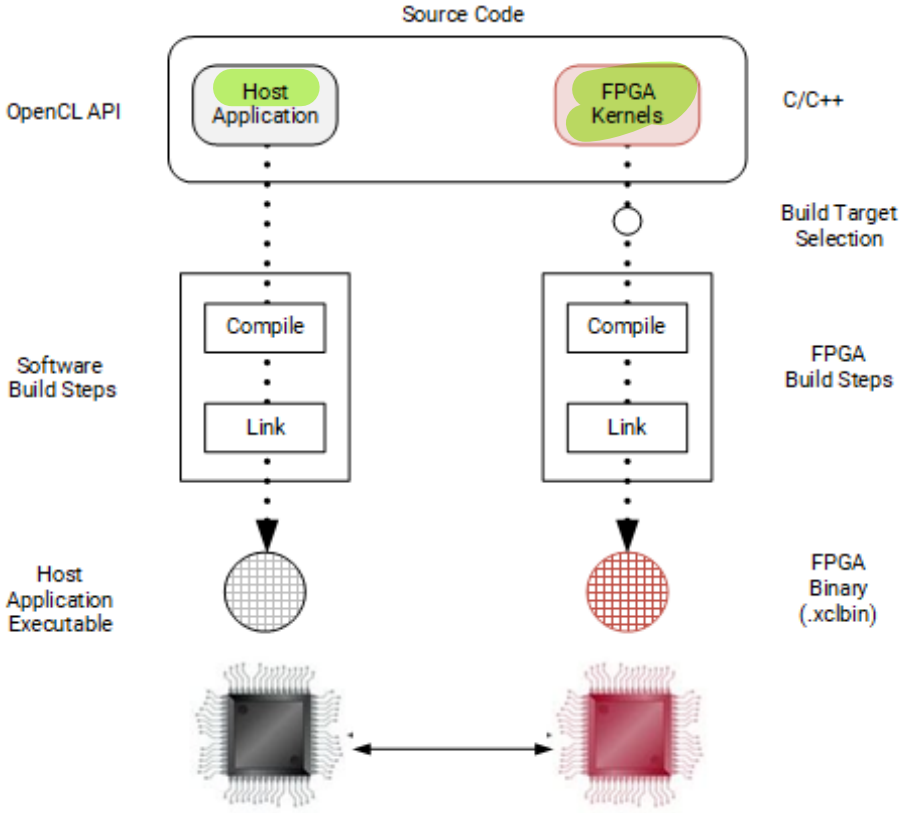
1. Εισαγωγή

Σε αντίθεση με το παρελθόν, όπου οι επιταχυντές υλικού (accelerators) σχεδιάζονταν κυρίως με τις γλώσσες περιγραφής υλικού (Hardware Description Languages - HDLs), σήμερα σχεδιάζονται και με μεθόδους σύνθεσης υψηλού επιπέδου (High-Level Synthesis - HLS). Η σχεδίαση με μεθόδους HLS αφορά τη χρήση κάποιας γλώσσας υψηλού επιπέδου (όπως C/C++) για την περιγραφή ψηφιακών κυκλωμάτων. Οι λόγοι χρήσης μεθόδων HLS αφορούν τη μείωση του χρόνου ανάπτυξης, την καλή ποιότητα της παραγόμενης σχεδίασης και την πολύ μεγάλη ευκολία διαχείρισης και μεταβολής της αρχικής σχεδίασης ώστε να προσαρμοστεί σε νέες απαιτήσεις.

Μετά το πέρας της σχεδίασης και της προσομοίωσης (simulation) ενός επιταχυντή υλικού ακολουθεί η υλοποίησή του σε πραγματικό υλικό (π.χ. μία πλακέτα που περιλαμβάνει ένα FPGA SoC) για να ελεγχθεί εάν ο υλοποιημένος επιταχυντής τηρεί τις αρχικές προδιαγραφές σχεδίασης όταν υλοποιηθεί σε υλικό. Για το σκοπό αυτό, συνήθως γράφεται μια εφαρμογή η οποία τρέχει στο επεξεργαστικό σύστημα (Processing System - PS, δηλ. τη CPU) και καλεί τον επιταχυντή που έχει υλοποιηθεί στο τμήμα προγραμματιζόμενης λογικής (Programmable Logic - PL, δηλ. το FPGA) του FPGA SoC. Στο παρακάτω σχήμα φαίνεται η απλοποιημένη αρχιτεκτονική του FPGA Alveo U200 όπου δεξιά (Programmable Logic) φαίνονται τα κυρίως τμήματα της πλακέτας, ενώ στα αριστερά (Host Processor) το runtime σύστημα (Xilinx RunTime – XRT, drivers κτλ.) έτσι ώστε να μπορέσει να επικοινωνήσει ο Host (ο επεξεργαστής, δηλ. το PS) με τη πλακέτα FPGA. Ειδικότερα η εφαρμογή χωρίζεται μεταξύ του τμήματος της εφαρμογής που εκτελείται στη πλευρά του Host (Custom Application) και του τμήματος της εφαρμογής που εκτελείται στην FPGA για εκτέλεση επιταχυντών (Custom Kernel). Αυτά τα δύο τμήματα είναι συνδεδεμένα μεταξύ τους με PCIe κανάλι επικοινωνίας. Το πρόγραμμα του κεντρικού υπολογιστή είναι γραμμένο σε OpenCL και εκτελείται σε έναν επεξεργαστή (x86 ή Arm αρχιτεκτονικής), ενώ οι πυρήνες του επιτάχυνση εκτελούνται στη προγραμματιζόμενη λογική (PL) του Alveo U200.



Στην εικόνα που ακολουθεί μπορείτε να δείτε τη διαδικασία εκτέλεσης της εφαρμογής (αριστερά είναι ο επεξεργαστής γενικού σκοπού και δεξιά η FPGA). Περισσότερες πληροφορίες μπορείτε να βρείτε εδώ[1].



Σε αυτό το εργαστήριο θα ασχοληθούμε με τη **σχεδίαση ενός επιταχυντή (accelerator)** με το εργαλείο Vitis HLS. Αρχικά θα παρουσιαστεί μια ακολουθία βημάτων η οποία αποσκοπεί

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

σε μια αρχική γνωριμία με το εργαλείο Vitis HLS το οποίο υποστηρίζει την σχεδίαση κυκλωμάτων προς υλοποίηση σε πλατφόρμα FPGA SoC κατασκευής Xilinx. Στη συνέχεια θα σας ζητηθεί να κατασκευάσετε και να βελτιώσετε (optimize) έναν δικό σας επιταχυντή (accelerator). Το τελικό προϊόν αυτού του εργαστηρίου είναι αυτό που θα χρησιμοποιήσετε ως αφετηρία για τη δημιουργία της εφαρμογής κατά τη διάρκεια της software σχεδίασης του συστήματος σας στα επόμενα εργαστήρια.

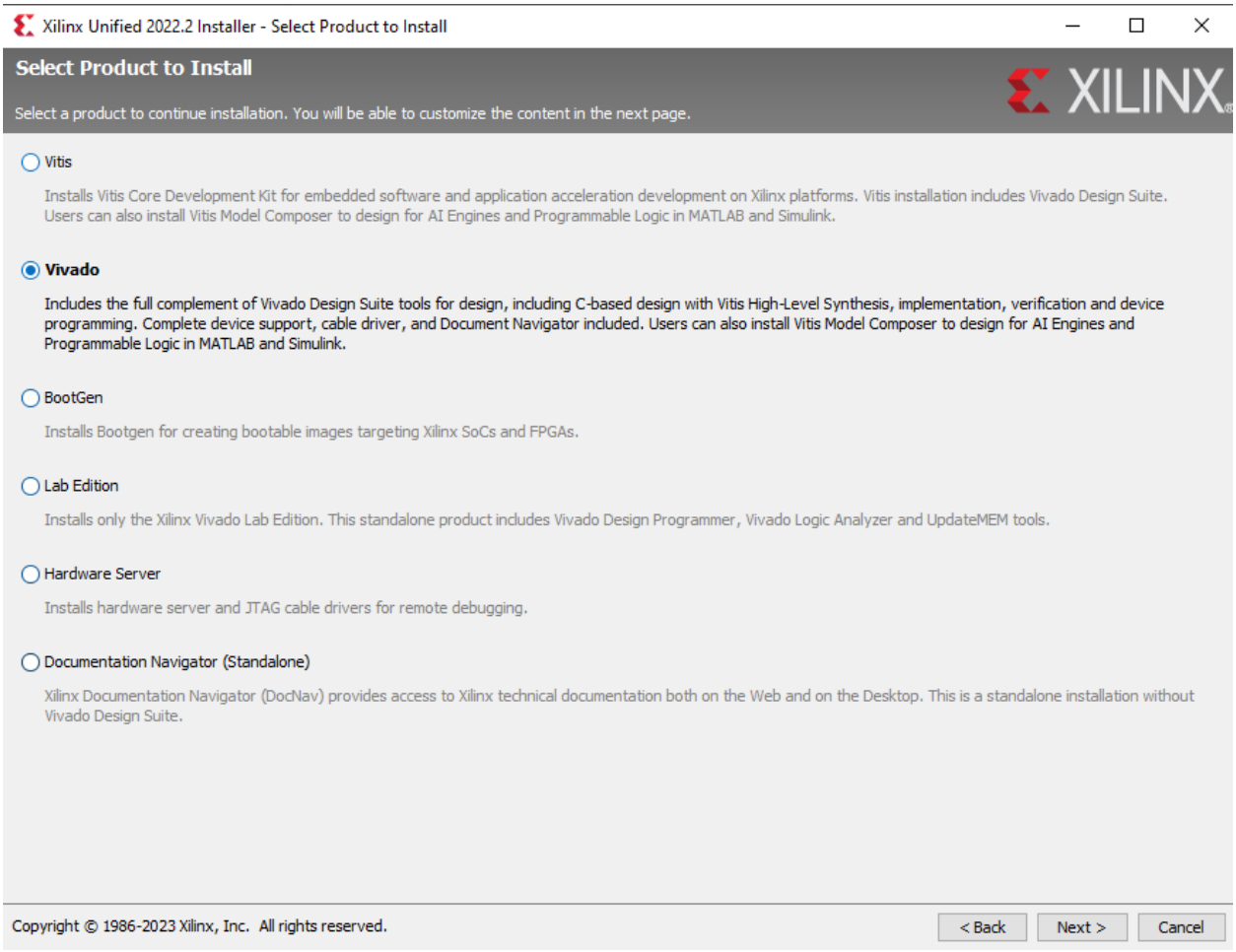
2. Έκδοση εργαλείων

Τα εργαλεία που θα χρησιμοποιήσουμε είναι αυτά της Xilinx και θα βασιστούμε στην έκδοση Vivado 2022.2 (υπάρχει πιθανότητα να αναφέρεται και ως Vitis HLS). Το Vitis HLS[2] είναι υπο-εργαλείο και υποστηρίζει την δημιουργία επιταχυντών custom λειτουργικότητας προς υλοποίηση σε FPGA ξεκινώντας με γλώσσες περιγραφής υψηλού επιπέδου. Συγκεκριμένα υποστηρίζει ανάπτυξη κώδικα C, C++, SystemC και OpenCL. Μπορείτε να βρείτε περισσότερες λεπτομέρειες σχετικά με το Vitis HLS στο User Guide[3]. Σας συνιστούμε να κατεβάσετε το ολοκληρωμένο πακέτο εργαλείων της Xilinx, Vitis 2022.2. Ειδικότερα μπορείτε να βρείτε την έκδοση τόσο για Windows όσο για Linux[4]:

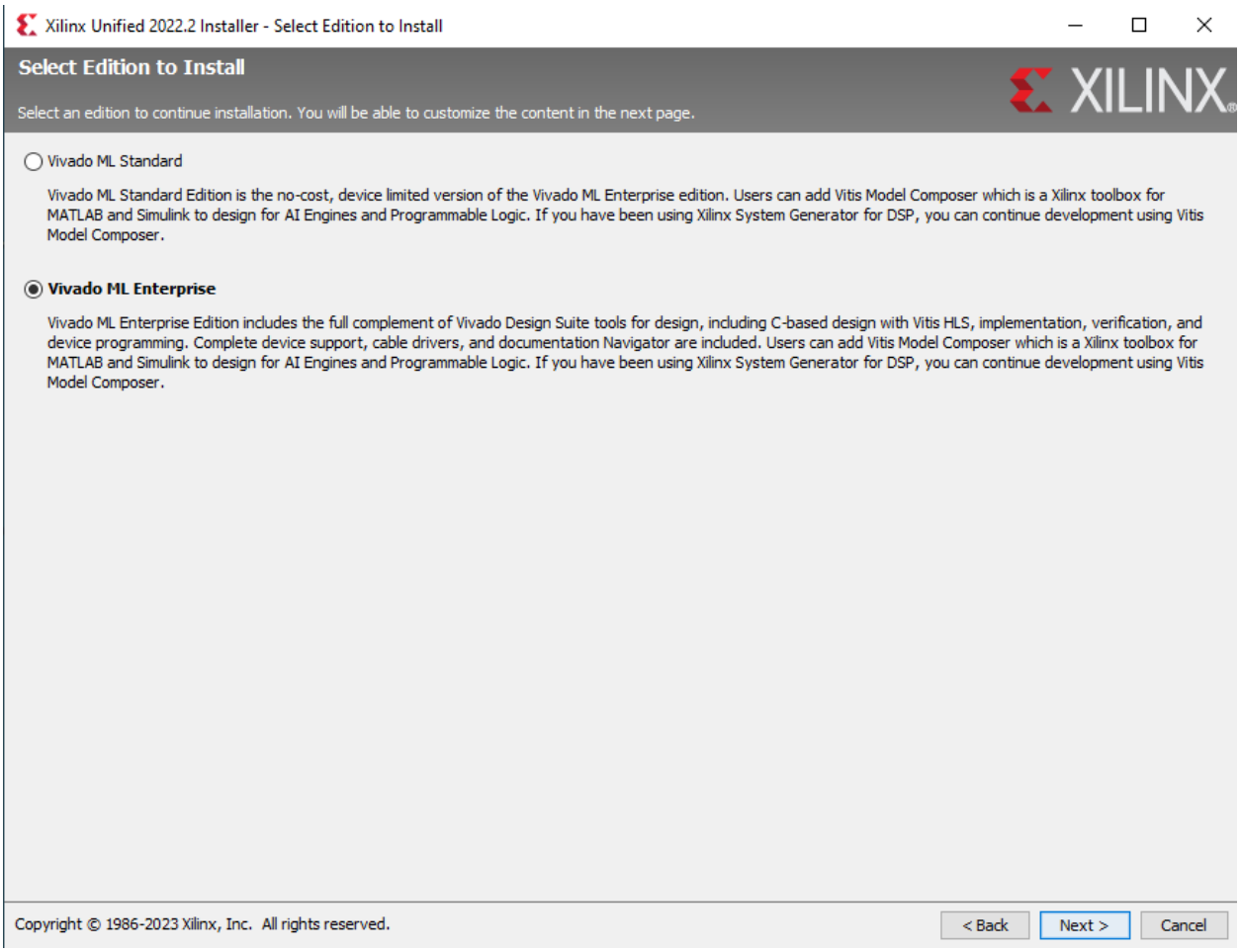
- Xilinx Unified Installer 2022.2: **Windows** Self Extracting Web Installer
- Xilinx Unified Installer 2022.2: **Linux** Self Extracting Web Installer

Κατά τη διαδικασία εγκατάστασης επιλέξτε το Vivado Suite (όπως φαίνεται στην ακόλουθη Εικόνα) καθώς το Vitis θα το χρησιμοποιήσετε από Server στον οποίο είναι εγκατεστημένο στο εργαστήριο (δεν έχετε license για χρήση από το δικό σας υπολογιστή).

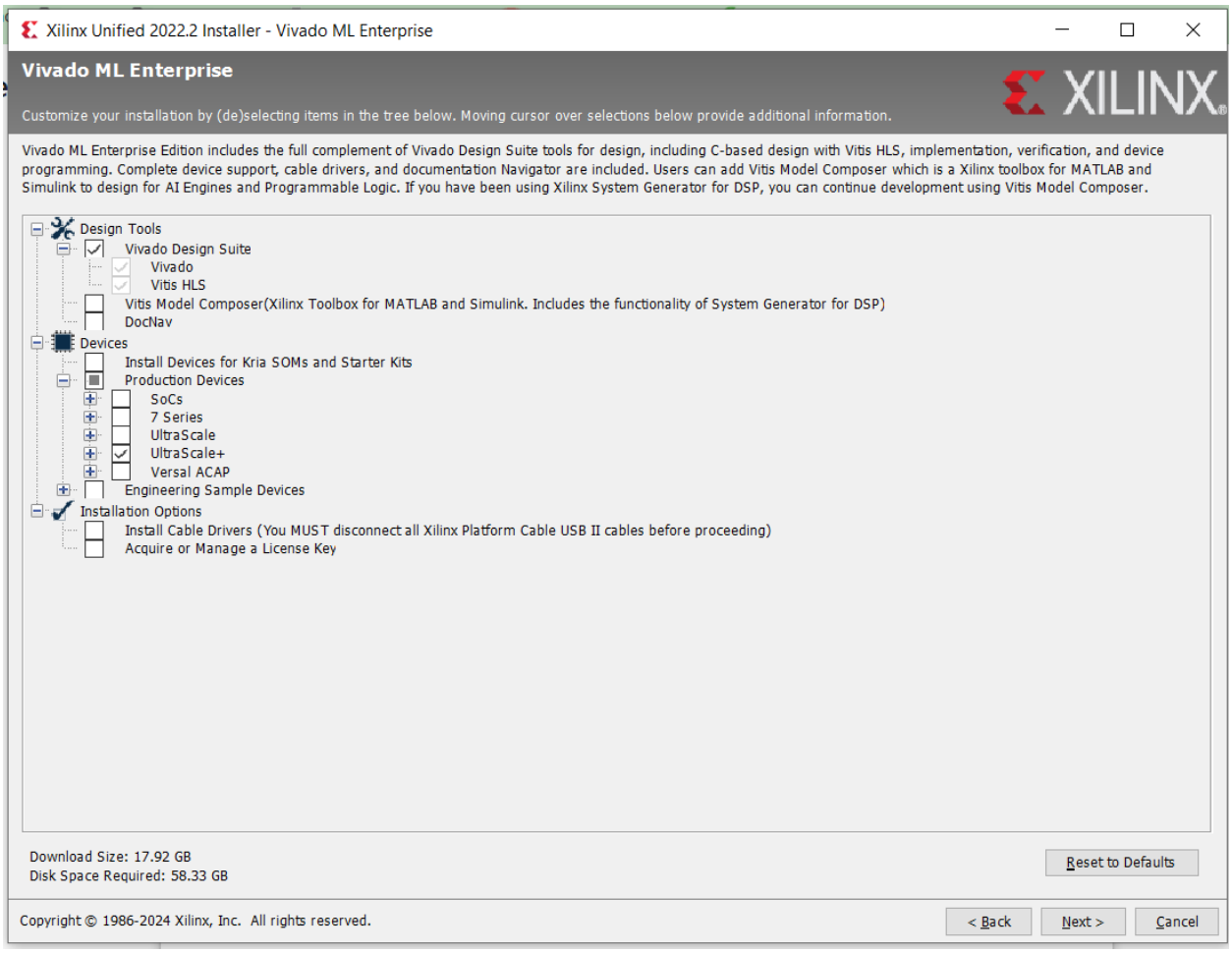
[2] https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html
[3] https://docs.xilinx.com/r/2022.2-English/ug1399-vitis-hls/Introduction
[4] https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/ -design-tools/archive.html



Στη συνέχεια επιλέξτε την εγκατάσταση της πλήρης έκδοσης του εργαλείου (Vivado ML Enterprise) η οποία περιλαμβάνει την υποστήριξη της κάρτας που θα χρησιμοποιήσετε στα επόμενα εργαστήρια.



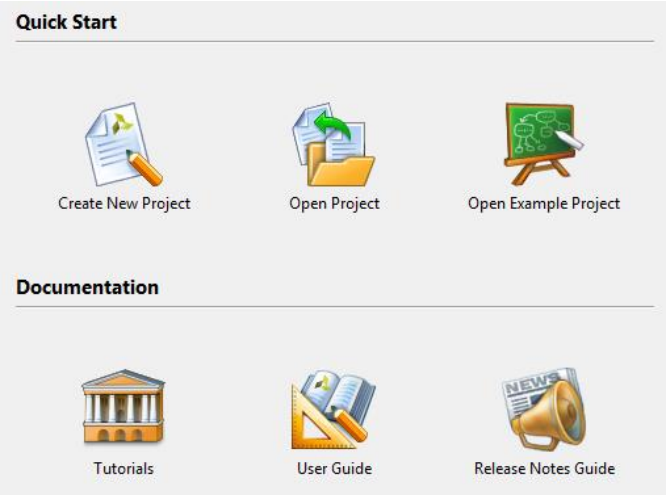
Κάνοντας τις ακόλουθες επιλογές κατά την εγκατάσταση του εργαλείου, μπορείτε να μειώσετε αισθητά το χώρο που θα καταλάβει στη συσκευή σας.



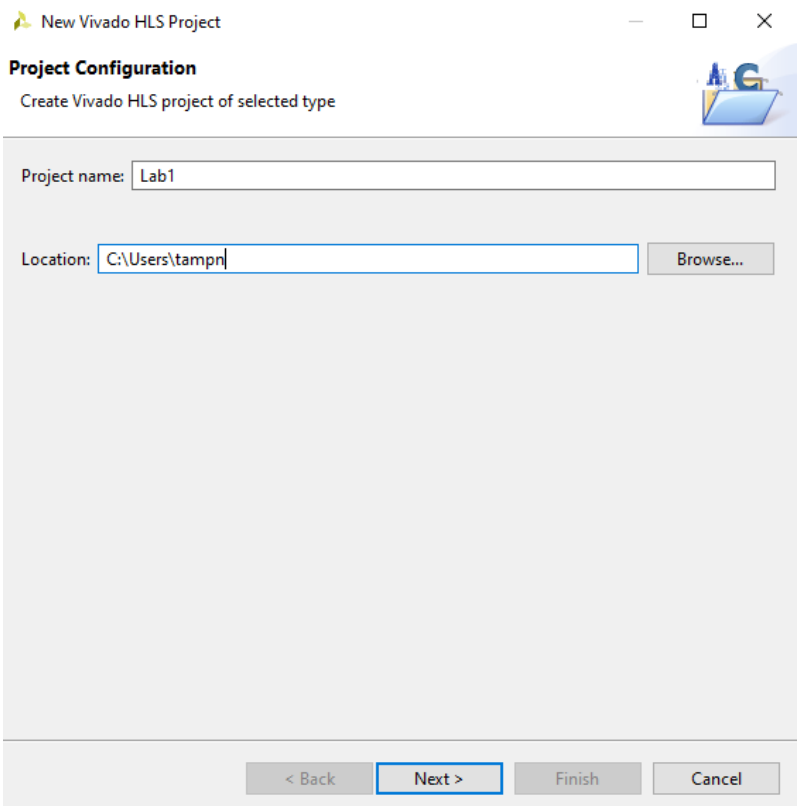
Για τη κάρτα Alveo U200 κατεβάστε το αρχείο au200 (από το elearning) και κάντε το extract στον ακόλουθο φάκελο:
C:\xilinx\Vivado\2022.2\data\xhub\boards\XilinxBoardStore\boards\Xilinx\

3. Εισαγωγή στο Vitis HLS (20%)

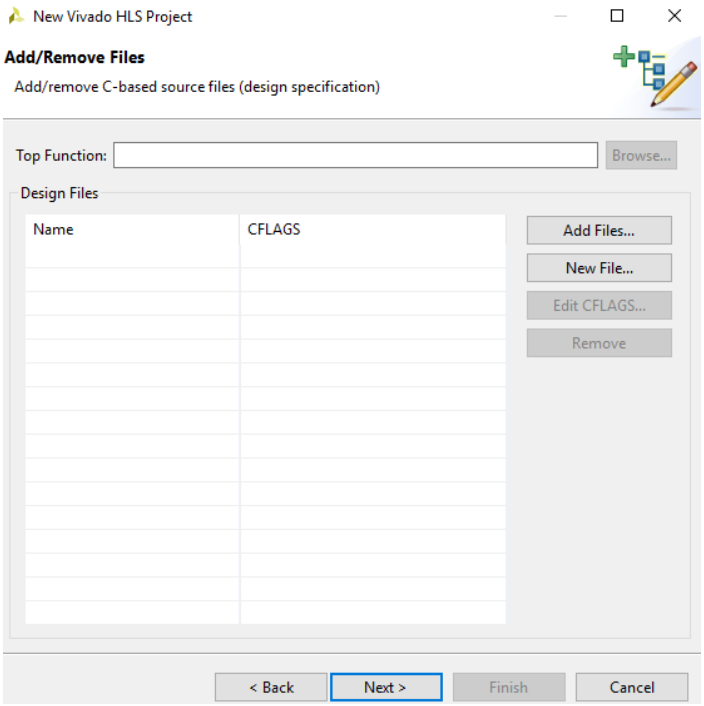
Τώρα ήρθε η στιγμή να ξεκινήσετε το Vitis HLS και στο σχετικό παράθυρο επιλέξτε τη δημιουργία νέου project, ***Create New Project***.



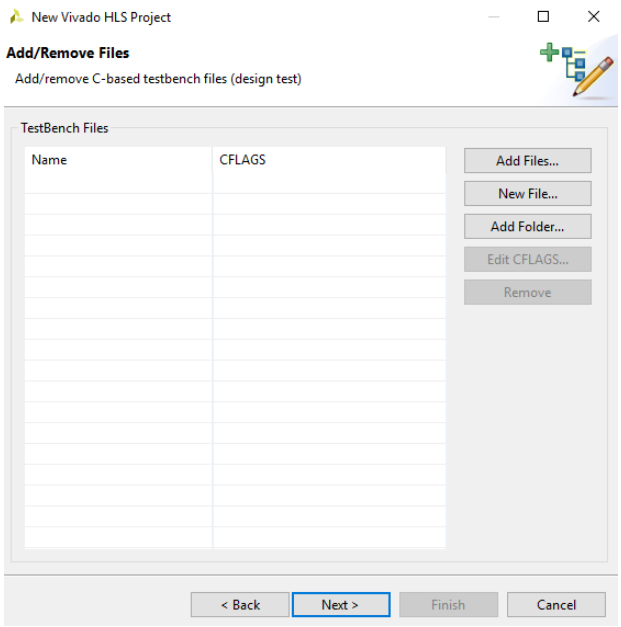
Στην επιλογή που ακολουθεί, πρέπει να επιλέξετε το όνομα του project σας καθώς και τη τοποθεσία αποθήκευσης. Όταν συμπληρώσετε τα σχετικά στοιχεία, πατήστε ***Next***.



Το επόμενο παράθυρο που προκύπτει μας ζητάει να προκαθορίσουμε ήδη υπάρχοντα αρχεία σχετικά με την υλοποίηση που θέλουμε να δημιουργήσουμε. Στην περίπτωση μας δεν έχουμε κάποια έτοιμα επομένως αγνοούμε το βήμα αυτό και επιλέγουμε κατευθείαν ***Next***.

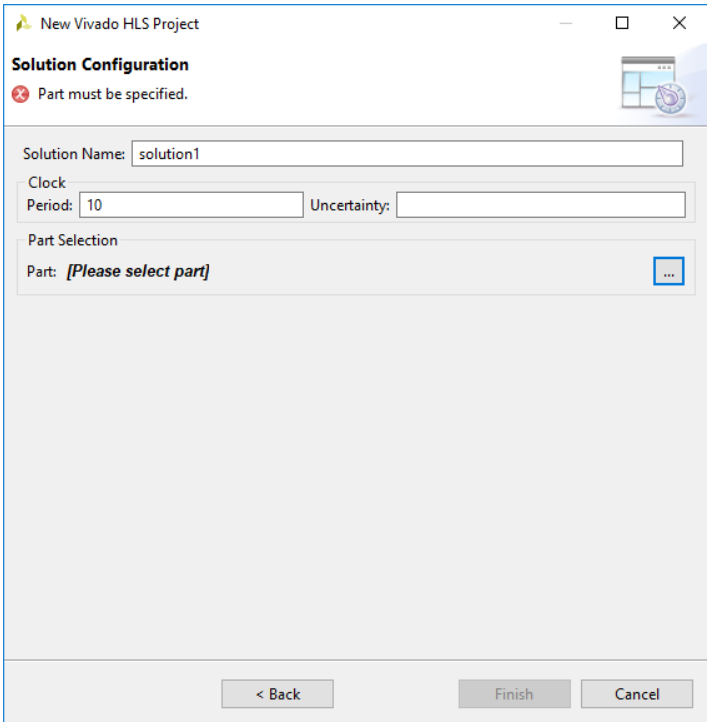


Παρόμοιος είναι και ο σκοπός του βήματος που ακολουθεί. Αυτή τη φορά μας ζητείται να ορίσουμε αρχεία που πρόκειται να λειτουργήσουν ως testbench καθώς και αρχεία δεδομένων τα οποία θα λειτουργήσουν ως golden (σημείο αναφοράς και σύγκρισης) προς επιβεβαίωση και επαλήθευση της λειτουργικότητας που πρόκειται να περιγράψουμε. Όπως πριν, δεν συμπληρώνουμε κάποιο στοιχείο και επιλέγουμε κατευθείαν ***Next***.

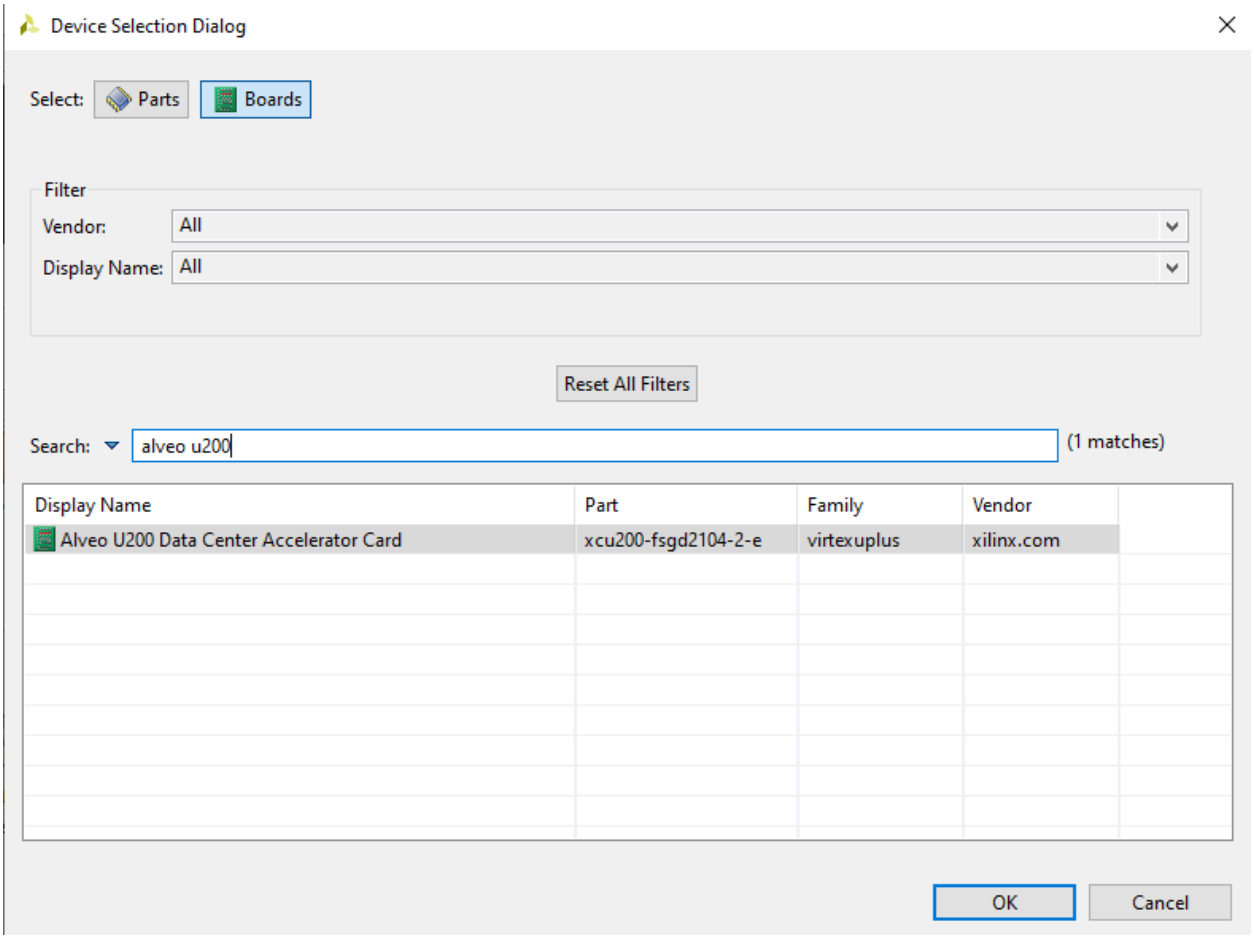


Στο επόμενο παράθυρο που προκύπτει, πρέπει να συμπληρώσετε ένα αριθμό από στοιχεία. Αρχικά αποδέχεστε το default όνομα για solution (solution - Καθεστώς συγκεκριμένων χαρακτηριστικών και παραμέτρων υλοποίησής της λειτουργικότητας σας – μπορείτε να έχετε πάνω από ένα στο ίδιο project).

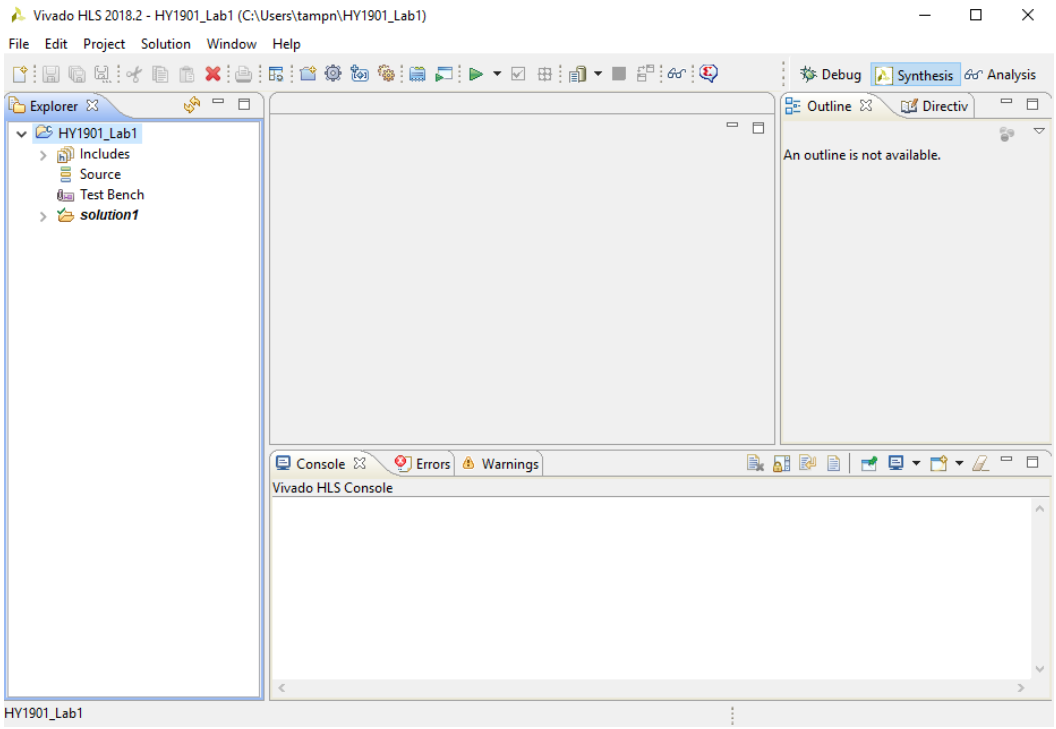
Εδώ επιλέγετε και την περίοδο του ρολογιού που επιθυμείτε για την υλοποίηση σας καθώς και το περιθώριο απόκλισης από αυτήν (αν δεν δώσετε συγκεκριμένη τιμή, η default είναι 12.5% της περιόδου του ρολογιού).



Τέλος, ως Board Selection πρέπει να επιλέξετε το Alveo U200 Board όπως φαίνεται στην επόμενη εικόνα.



Με την ολοκλήρωση της επιλογής πλατφόρμας μπορείτε να πατήσετε την **OK**. Αυτό θα ολοκληρώσει την διαδικασία δημιουργίας ενός νέου project και θα οδηγήσει στην εμφάνιση της παρακάτω εικόνας η οποία απεικονίζει το περιβάλλον εργασίας του Vitis HLS.

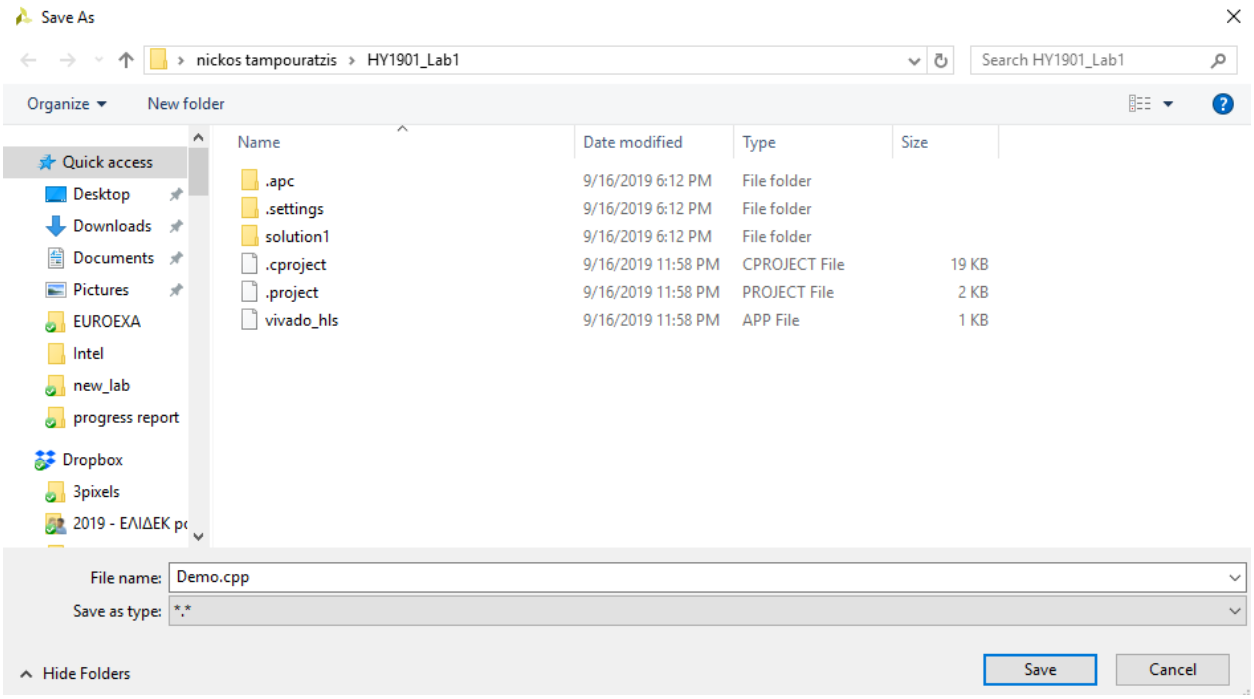


Εδώ μπορούμε να παρατηρήσουμε έναν αριθμό από πράγματα. Καταρχήν, το όνομα του project εμφανίζεται στην επάνω γραμμή του παραθύρου *Explorer*. Επιπλέον, το Vitis HLS ρυθμίζει τις πληροφορίες του project με ιεραρχικό τρόπο. Εδώ έχουμε πληροφορία σχετική με τον **source** κώδικα, το testbench αρχείο και τα διάφορα **solutions** (σενάρια υλοποίησης).

Το ίδιο το solution περιέχει πληροφορίες σχετικές με την πλατφόρμα υλοποίησης, σχεδιαστικά directives, και αποτελέσματα (προσομοίωσης, synthesis, IP export και άλλα).

Το πρώτο βήμα μας είναι να κάνουμε ***δεξί κλικ*** στην επιλογή **Source** και να επιλέξουμε ***New File***. Στο παράθυρο που θα εμφανιστεί, πρέπει να επιλέξετε το όνομα του αρχείου σας, τη τοποθεσία καθώς και τον τύπο αρχείου. Στη περίπτωση μας η τοποθεσία παραμένει η προτεινόμενη και ο τύπος του αρχείου θα είναι .cpp δηλαδή C++. Για όνομα επιλέγετε αυτό της αρεσκείας σας.

Τέλος επιλέγουμε ***Save***.



Αυτή η διαδικασία θα μας προσθέσει το αρχείο Demo.cpp στα Sources του project και θα μας το ανοίξει αυτόματα. Φυσικά πρόκειται για ένα κενό αρχείο στο οποίο μέσα πρέπει να περιγράψετε τη λειτουργικότητα που επιθυμείτε να υλοποιήσετε σε υλικό (hardware). Για το σκοπό αυτής της εισαγωγικής ενότητας, στόχος μας είναι η λειτουργικότητα που βλέπετε παρακάτω και πρέπει να την ***περάσετε μέσα στο αρχείο σας***.

Demo.cpp

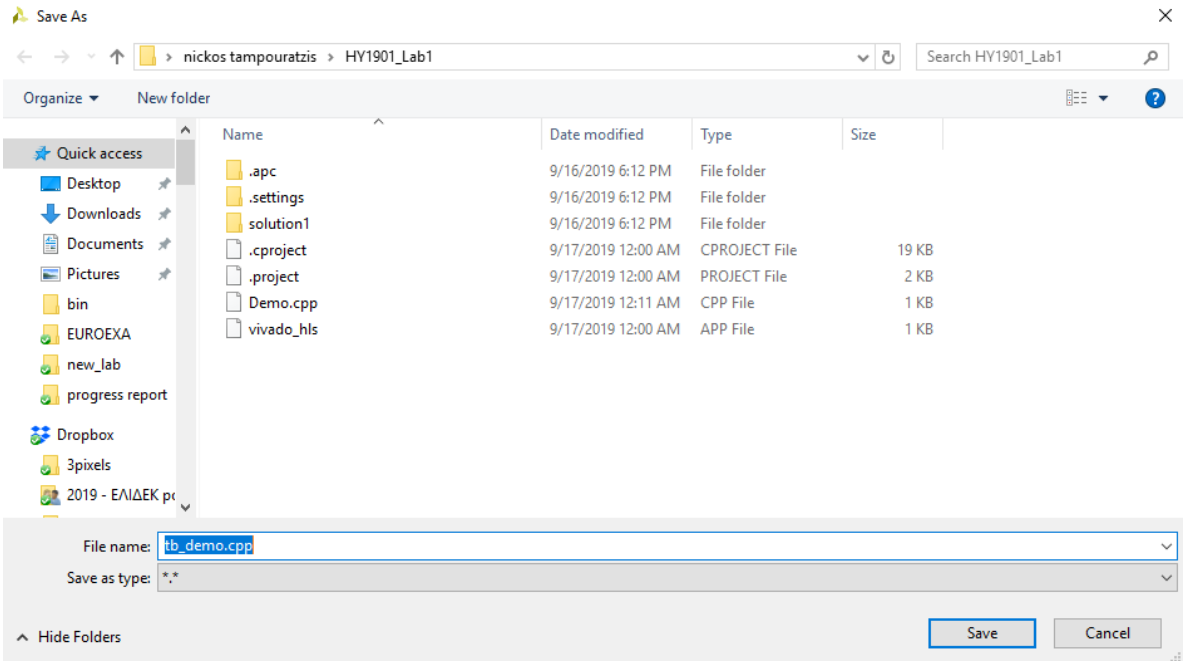
```
void simpleALU(int A, int B, int op, int *C){
    switch (op){
        case 0:{
            *C = A + B;
            break;
        }
        case 1:
        {
            *C = A - B;
            break;
        }
        case 2:
        {
            *C = A * B;
            break;
        }
        case 3:
        {
            *C = A / B;
            break;
        }
    }

}
```

Στην περίπτωση μας, αυτό που υλοποιούμε είναι ένα απλό ALU (Arithmetic Logic Unit) το οποίο σου δίνει τη δυνατότητα να πραγματοποιήσεις πρόσθεση, αφαίρεση, πολλαπλασιασμό και διαίρεση μεταξύ δύο μεταβλητών περνώντας το αποτέλεσμα της πράξης σε μια τρίτη μεταβλητή.

Το αμέσως επόμενο μας βήμα είναι να δημιουργήσουμε ένα testbench αρχείο έτσι ώστε να μπορέσουμε να επαληθεύσουμε τη λειτουργικότητα του επιταχυντή μας. Για να γίνει αυτό πρέπει να επιλέξουμε με ***δεξί κλικ*** το ***Test Bench*** στο ***Explorer Panel*** και μετά να πατήσουμε ***New File***.

Στο παράθυρο που προκύπτει πρέπει να δώσουμε ένα όνομα στο αρχείο που θα λειτουργήσει ως testbench, τον τύπο του (π.χ. *c* ή *cpp*) και την τοποθεσία αποθήκευσης. Για να βρίσκεται αυτόματα από το εργαλείο μπορούμε να το σώσουμε στο project directory και για ονοματοδοσία συνήθως επιλέγουμε το ίδιο με αυτό του Source αρχείου προσθέτοντας όμως τα αρχικά *tb* έτσι ώστε να είναι εύκολο να αναγνωρίσουμε ότι πρόκειται περί testbench.

Στο τέλος επιλέγουμε ***Save***.



Το testbench θα ανοίξει αυτόματα και εντός αυτού πρέπει να συμπληρώσετε κώδικα ο οποίος θα σας επιτρέψει να επαληθεύσετε τη λειτουργικότητα του επιταχυντή σας. Στην ειδική περίπτωση του project μας, μπορείτε να ***χρησιμοποιήσετε τον παρακάτω κώδικα***.

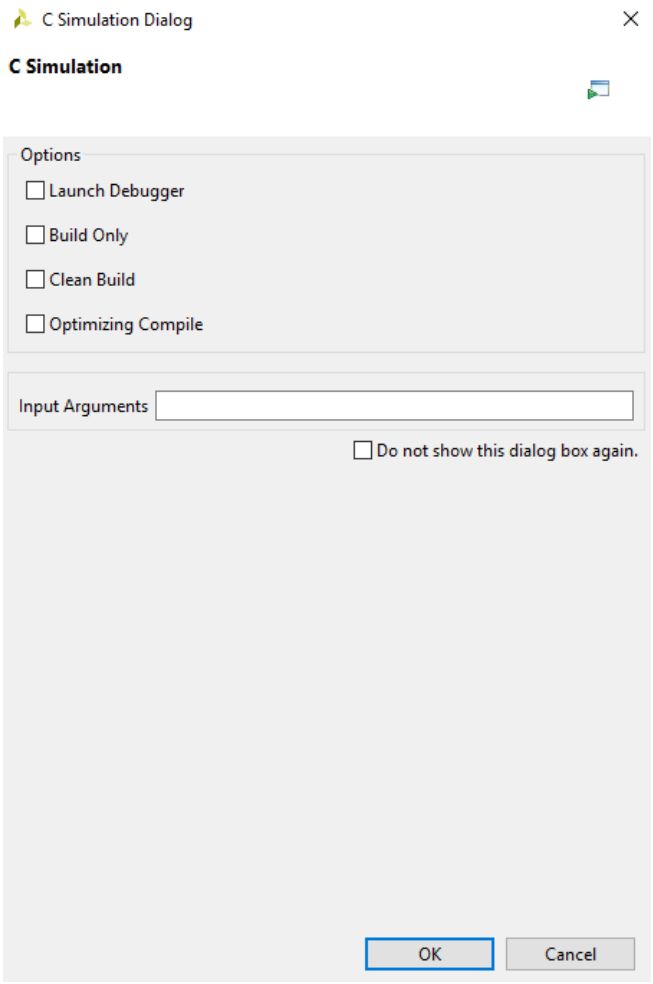
tb_demo.cpp

```
#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;

}
```

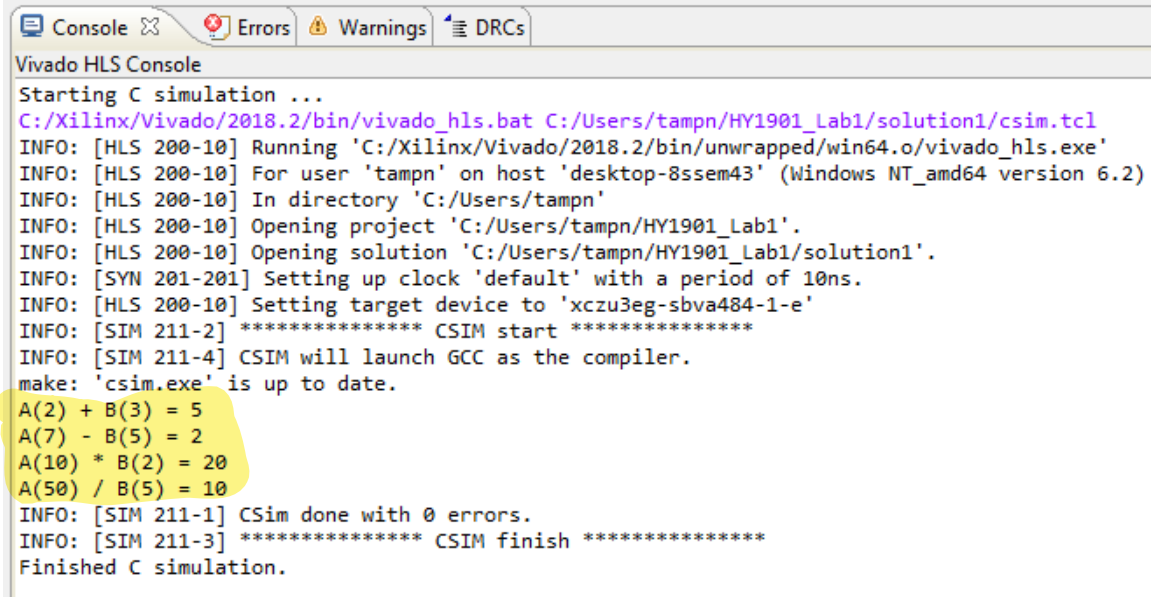
Ο κώδικας του testbench θα σας επιτρέψει να δοκιμάσετε και τις τέσσερις διαφορετικές πράξεις που προσφέρει το υλικό που θα δημιουργήσετε.

Το επόμενο βήμα στη σχεδιαστική μας ροή είναι να τρέξουμε μια πρώτη προσομοίωση σε αυτό το αλγοριθμικό επίπεδο σχεδίασης έτσι ώστε να επαληθεύσουμε ότι πραγματοποιεί την αναμενόμενη λειτουργικότητα.

Για να γίνει αυτό, πρέπει να πατήσετε το κουμπί ***Run C Simulation*** ή να χρησιμοποιήσετε το μενού ***Project > Run C Simulation***. Αυτή η επιλογή θα κάνει πρώτα compile τον κώδικα σας και μετά θα εκτελέσει, ως απλό software πρόγραμμα, τη σχεδίασή σας. Το παρακάτω παράθυρο ανοίγει όταν επιλέξετε *Run C Simulation* και για την ώρα μπορούμε να αγνοήσουμε τις επιπλέον επιλογές (*Options*). Πατάμε ***OK*** για να τρέξει η προσομοίωση μας.



Ακολούθως, η προσομοίωση ολοκληρώνεται με επιτυχία ή αποτυχία. Στο ***Console Panel*** μπορείτε να εξετάσετε τα *printf* του testbench σας για να επιβεβαιώσετε ότι οι πράξεις έχουν ολοκληρωθεί σωστά καθώς και να εξετάσετε τυχόν μηνύματα τύπου *warning* ή *error*.

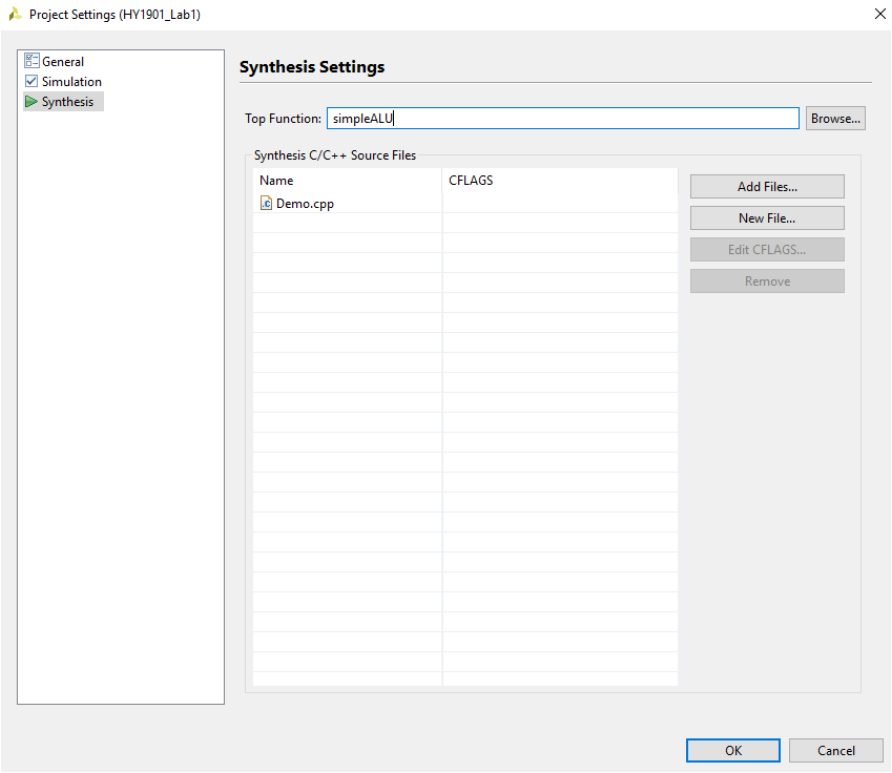


Εάν το επιθυμείτε, στο παράθυρο επιλογών για τη προσομοίωση, μπορείτε να επιλέξετε την επιλογή Debug και μέσω του debugger να πραγματοποιήσετε την επίλυση τυχόν προβλημάτων.

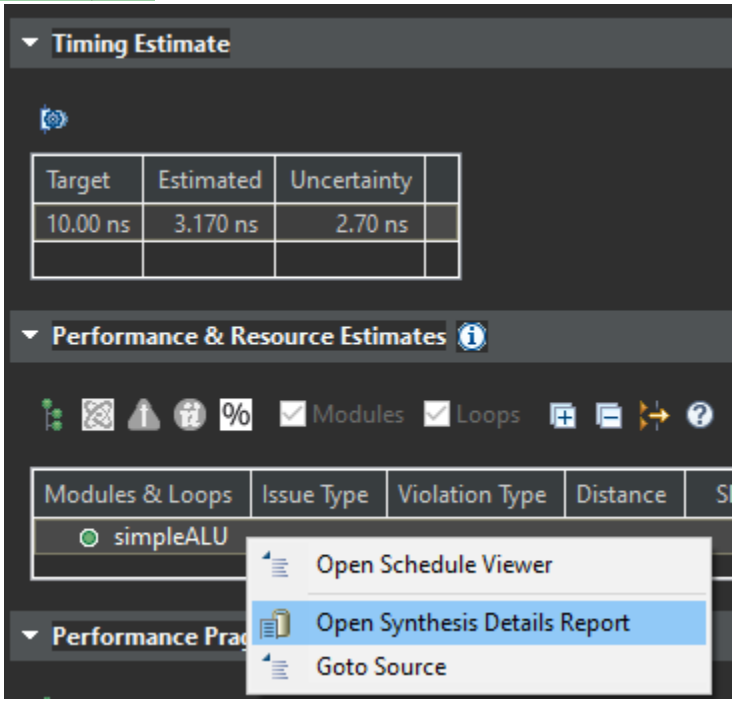
Στη περίπτωση που η διαδικασία ολοκληρωθεί με επιτυχία, η σχεδίασή σας είναι τώρα έτοιμη για σύνθεση (synthesis). Σε αυτό το βήμα μετατρέπετε τον επιταχυντή σας που βρίσκεται σε υψηλό επίπεδο περιγραφής βασισμένος στη γλώσσα C++ σε περιγραφή χαμηλότερου επιπέδου (RTL) και σε γλώσσες VHDL, Verilog και SystemC.

Προτού όμως προχωρήσετε στο βήμα αυτό, πρέπει να βεβαιωθείτε ότι το εργαλείο γνωρίζει ποια συνάρτηση θα θεωρηθεί ως η top level έτσι ώστε να ξέρει με ποια θα επικοινωνήσει το testbench σας. Στη περίπτωση του project μας, χρησιμοποιούμε ως μια και μοναδική συνάρτηση αυτή με το όνομα ***simpleALU*** (επομένως πρέπει να την προσδιορίσουμε και στο σωστό σημείο).

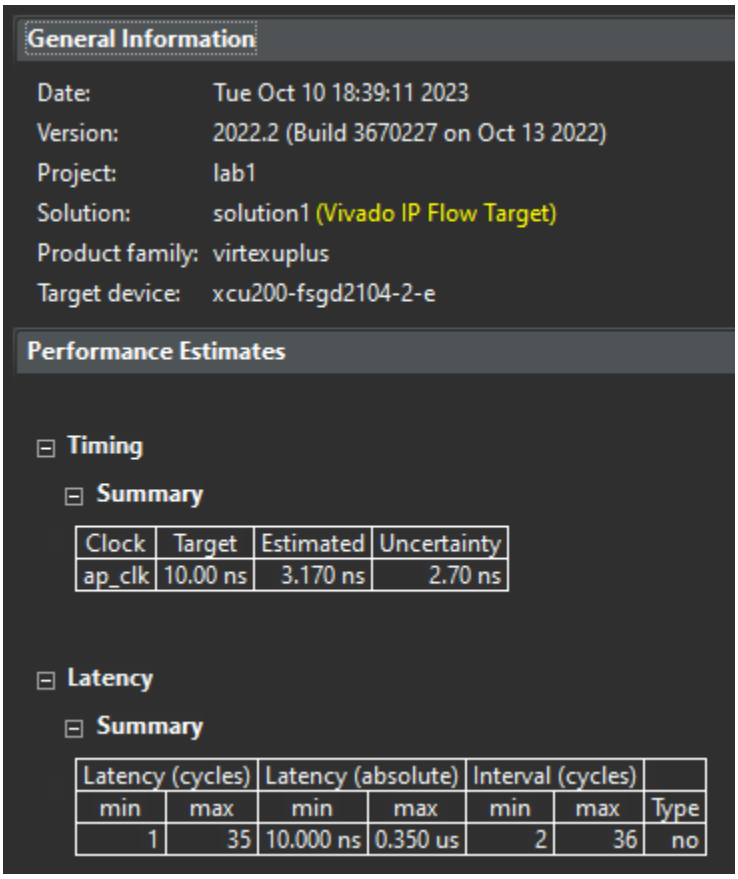
Επιλέξτε ***Project > Project Settings*** και θα ανοίξει το παρακάτω παράθυρο. Εδώ και στην επιλογή ***Synthesis*** πρέπει να συμπληρώσετε το σωστό στοιχείο αναφορικά με ***Top Function*** και μετά πατήστε ***OK***.



Μετά επιλέξτε την επιλογή ***Run C Synthesis*** από το toolbar ή χρησιμοποιήστε το μενού ***Solution > Run C Synthesis***. Με την επιτυχή ολοκλήρωση του βήματος αυτού, ανοίγει αυτόματα το αρχείο ***Synthesis Summary***. ***Επιλέξτε δεξί κλικ και στο simpleALU και πατήστε Open Synthesis Details Report.***



Εξετάζοντας προσεκτικά το αρχείο αυτό, αποκτούμε μια πρώτη εντύπωση σχετικά με την hardware διάσταση της υλοποίησης μας. Αφιερώστε μερικά λεπτά έτσι ώστε να δείτε κάποια βασικά χαρακτηριστικά του design σας.



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

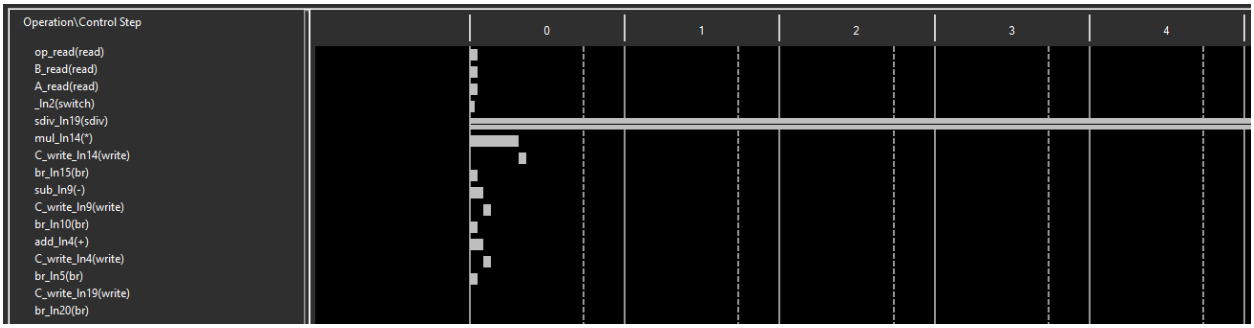
Εδώ μπορούμε να παρατηρήσουμε έναν αριθμό από πράγματα όπως:

Το ρολόι το οποίο θέσαμε ως στόχο, έχουμε καταφέρει να το επιτύχουμε μιας και το Estimated μαζί με το Uncertainty δεν ξεπερνούν το μέγεθος που ορίσαμε εμείς των 10ns.

Έχουμε ένα ελάχιστο latency ενός κύκλου και ένα μέγιστο latency 35 κύκλων. Με τη βοήθεια του Analysis θα εξετάσουμε σε ποια πράξη (πρόσθεση, αφαίρεση κτλ.) αναφέρετε το κάθε ένα από αυτά.

Επίσης, το interval μας λέει πότε διαβάζεται το επόμενο input. Στη ελάχιστη περίπτωση, αυτό είναι μετά από 1 κύκλο του ρολογιού ενώ στη μέγιστη περίπτωση αυτό γίνεται μετά από 35 κύκλους. Επομένως το σχέδιο μας ΔΕΝ είναι pipelined μιας και για να ξεκινήσει ένα transaction θα πρέπει να έχει ολοκληρωθεί το προηγούμενο. Κανονικά μια τέτοια συμπεριφορά δεν μας ικανοποιεί και προχωρούμε προς τροποποίηση της λειτουργικότητας όπως θα κάνετε εσείς στο επόμενο σκέλος αυτού του εργαστηρίου.

Σχετικά με τον προσδιορισμό του μέγιστου latency, επιλέξτε το ***Solution*** στην επάνω δεξιά μεριά του εργαλείου και επιλέξτε ***Open Schedule Viewer***.



Εδώ παρατηρούμε ότι οι 3 πράξεις, πρόσθεση, αφαίρεση και πολλαπλασιασμός, χρειάζονται ένα κύκλο του ρολογιού για να ολοκληρωθούν, ενώ εάν προχωρήσουμε προς τα δεξιά θα προσέξουμε ότι τους 35 κύκλους latency τους χρειάζεται η πράξη της διαίρεσης.

Επιστρέψτε στο ***Synthesis Summary***.

Παρατηρήστε τα δεδομένα στο Summary του **Utilisation Estimates**. Εδώ μπορείτε να αποκτήσετε μια πρώτη εκτίμηση σχετικά με το πόσους πόρους της FPGA που έχουμε θέσει ως στόχο πρόκειται να χρησιμοποιήσουμε. Τα πεδία που έχουν το περισσότερο βάρος είναι αυτά των LUTs, FFs, DSPs και BRAMs.

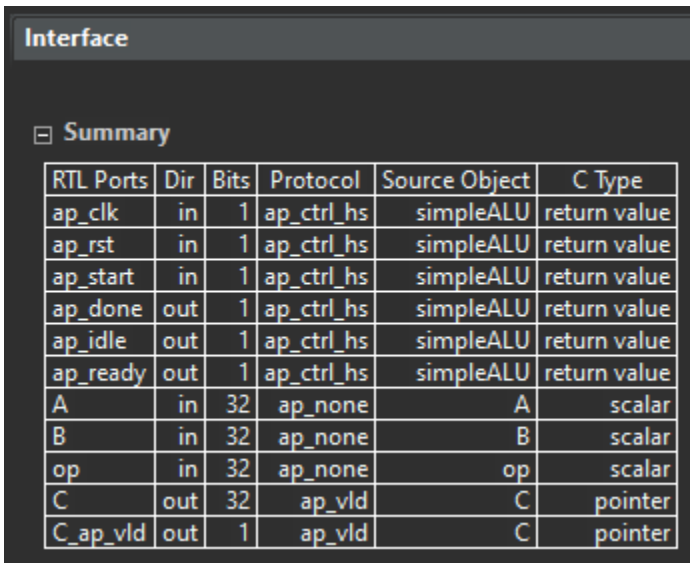
Summary

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

Να θυμάστε ότι αυτές οι τιμές είναι εκτιμήσεις και είναι πολύ πιθανό στην πραγματικότητα να αλλάξουν όταν πραγματοποιήσετε το βήμα Synthesis επιπέδου RTL μέσα από το εργαλείο Vitis (Εργαστήριο 2). Να επισημάνουμε ότι η κάρτα U200 έχει 3 Super Logic Region (SLR). Το Super Logic Region (SLR) όπως αναφέρει η Xilinx είναι το μικρότερο ενοποιημένο FPGA που έχει μέσα το U200. Στα πλαίσια των εργαστηρίων εμείς θα αναπτύξουμε την

υλοποίηση μας σε ένα SLR συνεπώς μας ενδιαφέρει το Utilization SLR (%) καθώς για να εκμεταλλευτούμε το συνολικό Utilization του U200 απαιτούνται προχωρημένες γνώσεις.

Τέλος, προχωρήστε παρακάτω για να δείτε την σύνοψη για το Interfacing του σχεδίου σας.



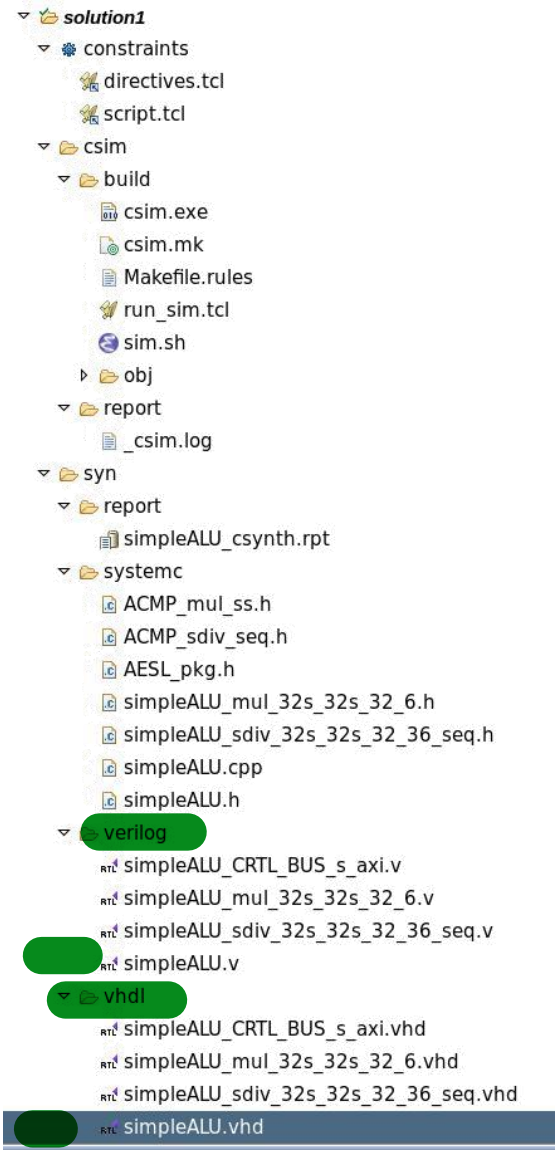
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	simpleALU	return value
ap_rst	in	1	ap_ctrl_hs	simpleALU	return value
ap_start	in	1	ap_ctrl_hs	simpleALU	return value
ap_done	out	1	ap_ctrl_hs	simpleALU	return value
ap_idle	out	1	ap_ctrl_hs	simpleALU	return value
ap_ready	out	1	ap_ctrl_hs	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

Εδώ μπορούμε να παρατηρήσουμε τα σήματα και τα σχετικά πρωτόκολλα που θα χρησιμοποιηθούν για τη διεπαφή του επιταχυντή με το σύστημα επεξεργασίας.

Αρχικά έχουμε ένα σήμα **clock** και ένα σήμα **reset**, τα οποία σχετίζονται με το Source Object simpleALU, δηλαδή με τη σχεδίασή μας. Επιπλέον σήματα θα χρησιμοποιηθούν για το σκοπό του ελέγχου του επιταχυντή. Αυτά τα σήματα έχουν μπει αυτοματοποιημένα από το εργαλείο κατά τη διάρκεια της εκτέλεσης του βήματος synthesis.

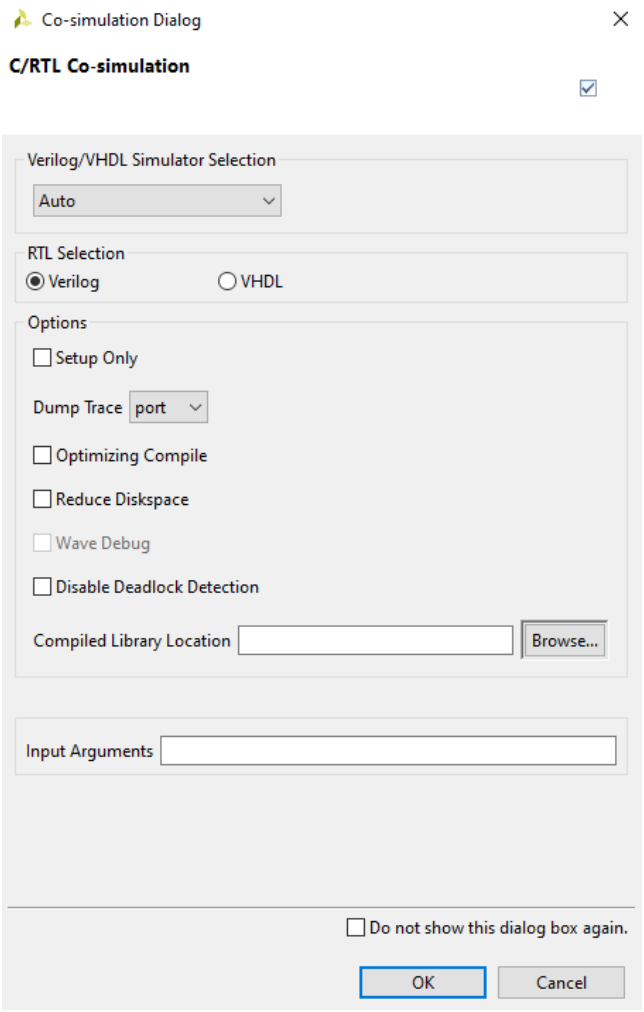
Τα input A, B, op και output C σήματα εξυπηρετούνται από διαύλους εύρους 32-bit ενώ το output έχει ένα επιπλέον σήμα (**C_ap_vld**) που ορίζει πότε τα δεδομένα εξόδου είναι valid. Τέλος, η επικοινωνία των input και output δεδομένων μας, αυτή τη στιγμή έχει υλοποιηθεί χωρίς να υπάρχει κάποιο I/O πρωτόκολλο (ap_none).

Σημειώστε ότι ψάχνοντας μέσα στο solution1 του project σας, μπορείτε να βρείτε όλα τα αρχεία που προέκυψαν από τη σύνθεση και για τις τρεις γλώσσες που προαναφέραμε (VHDL, Verilog και SystemC) και έτσι μπορεί κάποιος να μελετήσει τους σχετικούς κώδικες που δημιούργησε το εργαλείο.

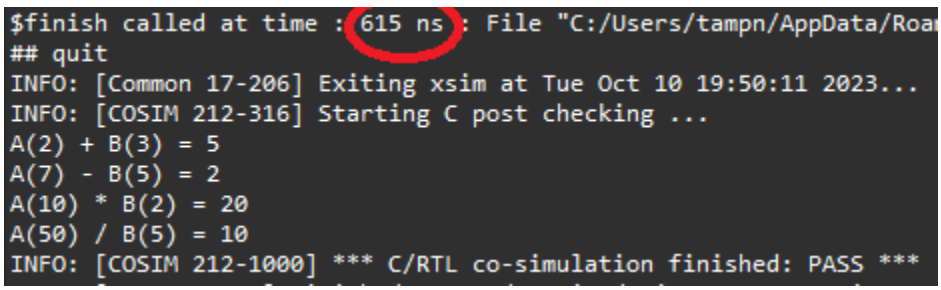


Τώρα ήρθε η ώρα της επαλήθευσης του RTL κώδικα που προέκυψε μέσα από το στάδιο του synthesis. Εδώ θα χρησιμοποιηθεί το testbench αρχείο μας για να επαληθεύσουμε τις RTL περιγραφές και τα αποτελέσματα αυτής της διαδικασίας θα συγκριθούν με αυτά της προσομοίωσης που έλαβε χώρα νωρίτερα.

Πατήστε την επιλογή ***Run C/RTL Cosimulation*** από το toolbar, εναλλακτικά χρησιμοποιήστε το μενού πηγαίνοντας ***Solution > Run C/RTL Cosimulation***. Το παρακάτω παράθυρο θα εμφανιστεί.

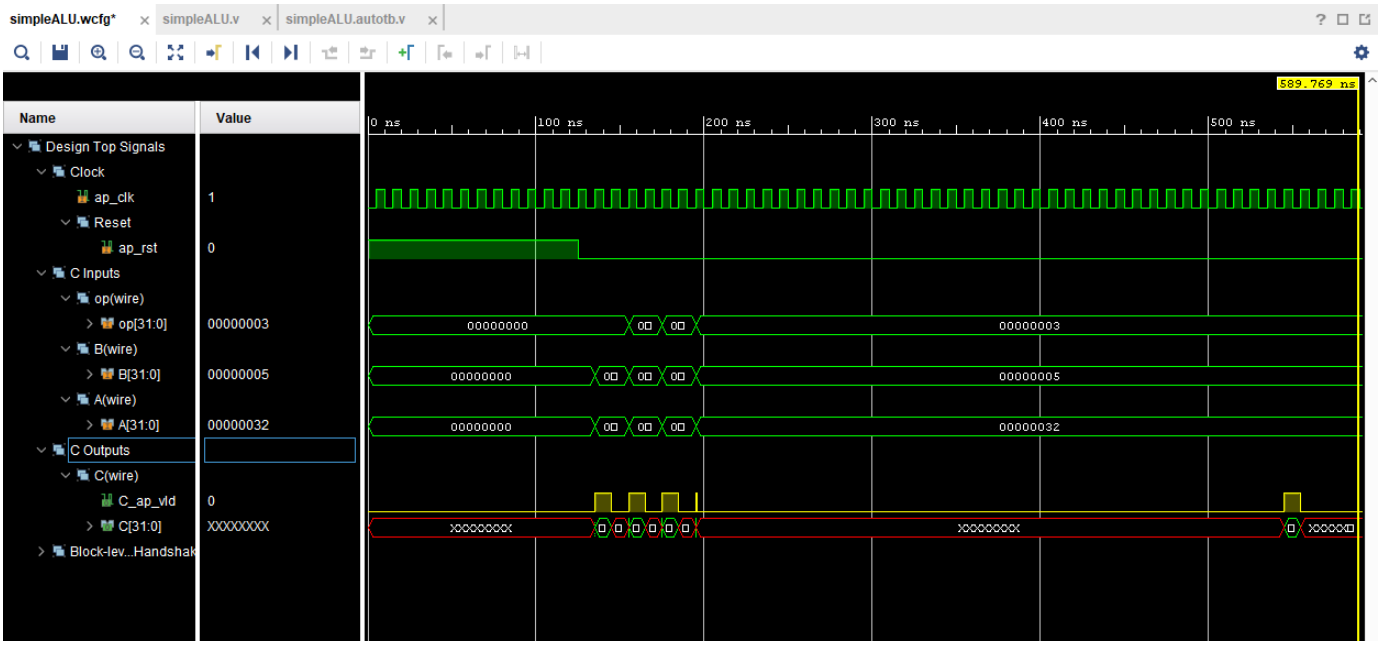


Εδώ το εργαλείο μας παρουσιάζει μια σειρά από άμεσες δυνατότητες όπως το να ορίσουμε συγκεκριμένα τον προσομοιωτή που θα χρησιμοποιηθεί και το εάν θέλουμε να χρησιμοποιήσει την Verilog ή την VHDL περιγραφή επιπέδου RTL στη διαδικασία επαλήθευσης. Μια άλλη σημαντική δυνατότητα είναι αυτή του να μας αποθηκεύσει τη δραστηριότητα σε επίπεδο σημάτων για να μπορέσουμε να παρατηρήσουμε οπτικά τι έχει συμβεί κατά τη διάρκεια της προσομοίωσης/επαλήθευσης του RTL κώδικα μας. Βεβαιωθείτε ότι η επιλογή ***port*** είναι ενεργοποιημένη στο πεδίο ***Dump Trace***.



Με την επιτυχή ολοκλήρωση και αυτού του βήματος, είμαστε έτοιμοι να εξετάσουμε τη συμπεριφορά του συστήματος μας στο χρόνο, σύμφωνα πάντα με το testbench που ορίσαμε νωρίτερα. Στη παραπάνω εικόνα μπορούμε να δούμε και το συνολικό χρόνο (execution time) που χρειάστηκε το design μας. Αν θέλουμε περισσότερη ανάλυση επιλέγουμε από το toolbar το ***Open Wave Viewer…*** ή το βρίσκουμε από το μενού ακολουθώντας ***Solution > Open Wave Viewer…***

Περιμένουμε λίγα λεπτά και το Vivado μας ανοίγει το αποτέλεσμα της προσομοίωσης. Αφιερώστε μερικά λεπτά έτσι ώστε να καταλάβετε πώς ακριβώς έτρεξε η προσομοίωση. Η συνάρτηση που έχετε υλοποιήσει είναι απλή οπότε θα πρέπει να μπορείτε καταλάβετε διεξοδικά το τι απεικονίζεται.



Όταν τελειώσετε, βεβαιωθείτε ότι έχετε κλείσει το Vivado επιλέγοντας File > Exit.

4. Σχεδίαση H/W accelerator χρησιμοποιώντας το Vitis HLS (80%)

Χρησιμοποιώντας το Vitis HLS, σχεδιάστε το hardware accelerator IMAGE_DIFF_POSTERIZE, ο οποίος θα δέχεται ως είσοδο δύο γκρι κλίμακας εικόνες (πίνακες) A και B διαστάσεων HEIGHT × WIDTH και θα παράγει μία εικόνα εξόδου C ίδιων διαστάσεων.
Οι πίνακες A, B και C μπορούν να θεωρηθούν ως γκρι εικόνες (grayscale images), όπου κάθε στοιχείο αναπαριστά τη φωτεινότητα ενός pixel στο εύρος 0–255.
Σκοπός του επιταχυντή είναι να "τονίζει" τις διαφορές ανάμεσα στις δύο εικόνες, εφαρμόζοντας μια απλή μη-γραμμική μετασχηματίση **posterization** πάνω στο μέτρο της διαφοράς των pixels.

Λειτουργικότητα του επιταχυντή IMAGE_DIFF_POSTERIZE

Κάθε στοιχείο των πινάκων A, B και C είναι 8-bit μη προσημασμένος ακέραιος (unsigned char ή uint8_t) στο εύρος 0–255.

Για κάθε θέση (i, j) ορίζουμε:

- A[i][j] : τιμή φωτεινότητας από την πρώτη εικόνα
- B[i][j] : τιμή φωτεινότητας από τη δεύτερη εικόνα
- C[i][j] : τιμή εξόδου

Αρχικά υπολογίζουμε τη διαφορά των δύο pixels:

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

και στη συνέχεια εφαρμόζουμε posterization χρησιμοποιώντας δύο κατώφλια T1 και T2, με $0 \le T1 < T2 \le 255$:

- Αν $D(i,j) < T1$, τότε $C[i][j] = 0$
- Αν $T1 \le D(i,j) < T2$, τότε $C[i][j] = 128$
- Αν $D(i,j) \ge T2$, τότε $C[i][j] = 255$

Έτσι:

- Περιοχές όπου οι δύο εικόνες είναι σχεδόν ίδιες (μικρή διαφορά) εμφανίζονται μαύρες (0),
- Περιοχές με μέτριες διαφορές εμφανίζονται σε μεσαίο γκρι (128),
- Περιοχές με μεγάλες διαφορές εμφανίζονται λευκές (255).

Οπτικά, η εικόνα C μπορεί να θεωρηθεί ως "χάρτης αλλαγών" μεταξύ των δύο εικόνων A και B.

Υποθέσεις – Ορισμοί

- Κάθε στοιχείο των πινάκων A, B και C είναι τύπου unsigned char ή uint8_t.
- Οι διαστάσεις των πινάκων (εικόνων) ορίζονται ως: 2 < WIDTH < 512
  - WIDTH = 2^lW, όπου lW είναι ακέραιος ώστε 1 ≤ lW ≤ 9 (π.χ. WIDTH = 2^8 = 256)
  - HEIGHT = 2^lH, όπου lH είναι ακέραιος ώστε 1 ≤ lH ≤ 9 (π.χ. HEIGHT = 2^8 = 256)
- Μπορείτε να αναπαραστήσετε τις εικόνες είτε ως δισδιάστατους πίνακες *A[HEIGHT][WIDTH], B[HEIGHT][WIDTH], C[HEIGHT][WIDTH],*
- είτε ως μονοδιάστατους πίνακες *A[HEIGHT*WIDTH], B[HEIGHT*WIDTH], C[HEIGHT*WIDTH]*

  με διάταξη row-major (γραμμή-προς-γραμμή). Ο ορισμός πρέπει να είναι συνεπής τόσο στη HW top-level συνάρτηση όσο και στο testbench.
- Τα thresholds T1 και T2 μπορούν να οριστούν ως σταθερές (π.χ. #define) ή ως παράμετροι εισόδου της top-level συνάρτησης, αρκεί να είναι ίδιες στη HW και στη SW reference υλοποίηση που θα χρησιμοποιήσετε στο testbench.
- Για τις ανάγκες του εργαστηρίου, θεωρούμε σταθερές τιμές T1 = 32 και T2 = 96, ορισμένες π.χ. ως #define T1 32 και #define T2 96 στον κώδικά σας. Αν επιθυμείτε, μπορείτε να πειραματιστείτε και με άλλες τιμές, αρκεί να είναι ίδιες στη HW και στη SW υλοποίηση.
- Δεν θα χρησιμοποιήσετε scanf() ή άλλον μηχανισμό εισόδου από πληκτρολόγιο για αρχικοποίηση. Οι τιμές των A και B θα δημιουργούνται μέσα στο testbench (π.χ. ψευδο-τυχαίες τιμές ή απλές συναρτήσεις των i, j).

Ερώτημα 1 (30%)

Γράψτε C κώδικα για τη συνάρτηση IMAGE_DIFF_POSTERIZE στο Vitis HLS σύμφωνα με τους παραπάνω ορισμούς και αποθηκεύστε τον στο source πεδίο του Vitis HLS.

Η συνάρτηση αυτή θα πρέπει:

a) Να δέχεται ως είσοδο δύο πίνακες A και B ίδιων διαστάσεων (HEIGHT × WIDTH).
b) Να παράγει ως έξοδο έναν πίνακα C ίδιων διαστάσεων.
c) Για κάθε στοιχείο (i, j) να υπολογίζει:
   i. Τη διαφορά $D(i,j)=|A[i][j]-B[i][j]|$ χρησιμοποιώντας κατάλληλο ενδιάμεσο τύπο (π.χ. int16_t ή int) ώστε να αποφευχθεί overflow κατά την αφαίρεση.
   ii. Την τιμή C[i][j] ανάλογα με τα κατώφλια T1, T2:
      1. αν $D(i,j) < T1 \rightarrow$ C[i][j] = 0
      2. αν $T1 \leq D(i,j) < T2 \rightarrow$ C[i][j] = 128
      3. αν $D(i,j) \geq T2 \rightarrow$ C[i][j] = 255

Επιπλέον, γράψτε ένα testbench σε C και προσθέστε το στο Vitis HLS, το οποίο:

a) Αρχικοποιεί τους πίνακες A και B με τιμές στο εύρος 0–255. Μπορείτε, για παράδειγμα, να χρησιμοποιήσετε:
   i. ένα απλό "gradient" ως προς i και j (π.χ. A[i][j] = f(i,j), B[i][j] = g(i,j)), ή

ii. ψευδο-τυχαίες τιμές (με σταθερό seed για αναπαραγωγιμότητα).

b) Υπολογίζει τα αποτελέσματα με δύο τρόπους:
   i. Μια καθαρά S/W υλοποίηση (reference function σε C) που υλοποιεί ακριβώς την παραπάνω λογική.
   ii. Την H/W υλοποίηση μέσω της top-level συνάρτησης IMAGE_DIFF_POSTERIZE στο Vitis HLS.

c) Συγκρίνει τα αποτελέσματα στοιχειομετρικά (για όλα τα στοιχεία C[i][j]) και εκτυπώνει ευανάγνωστο μήνυμα, π.χ. «Test Passed» σε περίπτωση που όλα τα στοιχεία συμφωνούν, ή κατάλληλο μήνυμα σφάλματος αν εντοπιστούν διαφορές.

Προαιρετικά, μπορείτε να εκτυπώσετε ένα μικρό υποσύνολο της εικόνας (π.χ. τα πρώτα 8×8 στοιχεία) ή στατιστικά (π.χ. πόσα pixels πήραν τιμή 0, 128, 255).

Ερώτημα 2 (5%)

Κάντε σύνθεση (C Synthesis) της παραπάνω σχεδίασής σας (χωρίς να έχετε προσθέσει ακόμη HLS directives, πέρα από ό,τι απαιτείται για το interface) με default settings και συμπληρώστε τα ακόλουθα (για μια συγκεκριμένη επιλογή διαστάσεων, π.χ. 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: _____

Στο report ή/και στην αναφορά σας να αναφέρετε με σαφήνεια ποιες διαστάσεις πινάκων χρησιμοποιήσατε.

Ερώτημα 3 (5%)

Τρέξτε C/RTL cosimulation και βεβαιωθείτε ότι η σχεδίαση σας περνάει το test επιτυχώς και συμπληρώστε τα ακόλουθα(για την ίδια επιλογή διαστάσεων, π.χ. HEIGHT = WIDTH = 256):

Total Execution Time: _____
Min latency: _____
Avg. latency: _____
Max latency: _____

Χρησιμοποιήστε τα αντίστοιχα reports του Vitis HLS (π.χ. Cosimulation Report, Performance Estimates) για να εντοπίσετε τις τιμές αυτές.

Ερώτημα 4 (40%)

Εφαρμόστε Vitis HLS directives (π.χ. ARRAY_PARTITION, PIPELINE, UNROLL) έτσι ώστε να βελτιώσετε όσο πιο πολύ μπορείτε το execution time[5]. Πληροφορίες για τα directives μπορείτε να βρείτε εδώ[6]. Πειραματιστείτε με διάφορες διαστάσεις των πινάκων και **απαντήστε στα ακόλουθα**:

i) Κρατώντας σταθερό το ένα μέγεθος (π.χ. HEIGHT) και μεταβάλλοντας το άλλο (π.χ. WIDTH = 64, 128, 256, 512), τι παρατηρείτε όσον αφορά:

- το συνολικό latency (σε κύκλους),
- τον συνολικό χρόνο εκτέλεσης (execution time)
- οποιαδήποτε άλλη παρατήρηση σας έκανε εντύπως.

ii) Αφού δοκιμάσετε διάφορους συνδυασμούς directives (π.χ. #pragma HLS PIPELINE στο εσωτερικό loop σάρωσης των στοιχείων, πιθανή χρήση UNROLL σε κατάλληλο επίπεδο, ή/και ARRAY_PARTITION σε πίνακες εφόσον κρίνετε ότι βοηθά), επιλέξτε την καλύτερη (κατά την κρίση σας) υλοποίηση για την περίπτωση HEIGHT = WIDTH = 256.
Για τη βέλτιστη αυτή υλοποίηση, συμπληρώστε:

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: _____

και περιγράψτε συνοπτικά ποια directives χρησιμοποιήσατε (σε ποιο loop βάλατε PIPELINE, αν κάνατε UNROLL και με ποιο factor, αν χρησιμοποιήσατε ARRAY_PARTITION κ.λπ.) και γιατί θεωρείτε ότι αυτά βελτίωσαν τις επιδόσεις.

iii) Τέλος υπολογίστε την επιτάχυνση (speed-up) της βέλτιστης hardware υλοποίηση σας συγκριτικά με την αρχική σχεδίαση σας σε hardware (χωρίς directives).

[5] Αν χρειαστεί εφαρμόστε το TRIPCOUNT pragma για να ορίσετε τα όριο των επαναλήψεων.
[6] https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis