A black background with a black square

Description automatically generated with medium confidence **Εθνικό Μετσόβιο Πολυτεχνείο**

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εικόνα που περιέχει ηλεκτρονικές συσκευές, ηλεκτρονικό εξάρτημα, στοιχείο κυκλώματος, παθητικό στοιχείο κυκλώματος

Το περιεχόμενο που δημιουργείται από τεχνολογία AI ενδέχεται να είναι εσφαλμένο.

**Μάθημα: Ψηφιακά Συστήματα VLSI**

**Θέμα: 5η Εργαστηριακή Άσκηση – Υλοποίηση FIR φίλτρου με AXI διεπαφή σε ZYNQ SoC FPGA**

Εξάμηνο: 8ο

Ομάδα: 6

Συνεργάτες:

* Ακύλας Αντώνιος 03121152
* Κουμπιάς Ιωάννης-Χρυσοβαλάντης 03121053

**Εισαγωγή**

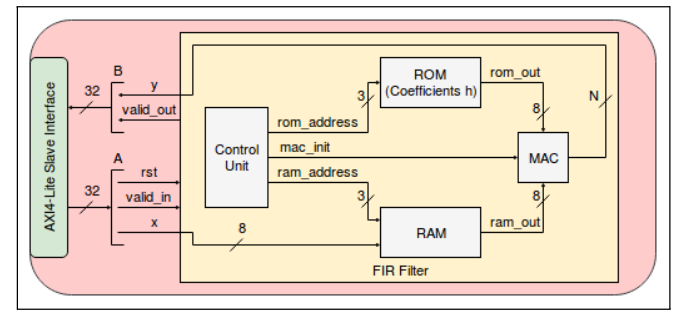
Σκοπός της παρούσας εργαστηριακής άσκησης είναι η υλοποίηση ενός FIR φίλτρου με ΑΧΙ4-Lite διεπαφή και στη συνέχεια η εισαγωγή του στο Zybo SoC.

**Υλοποίηση FIR Filter με διεπαφή AXI4-Lite**

Η αρχιτεκτονική του φίλτρου ταυτίζεται με την υλοποίηση που έχουμε ήδη φτιάξει σε προηγούμενο εργαστήριο.

Για να ενσωματώσουμε τη διεπαφή AXI4-Lite δημιουργούμε ένα νέο IP στο Vivado με βάση το δεδομένο template.

Το πρώτο βήμα είναι να εισάγουμε τα αρχεία κώδικα του fir (σε γλώσσα VHDL) και να ορίσουμε το component fir\_filter καθώς και μερικά απαραίτητα σήματα. Έπειτα πρέπει να δημιουργήσουμε ένα instance του φίλτρου συνδέοντας κατάλληλα τα σήματα εισόδου και εξόδου του.



Πιο συγκεκριμένα τόσο η είσοδος x όσο και τα σήματα ελέγχου reset και valid\_in εισάγονται στο σήμα Α το οποίο κατευθύνεται στον καταχωρητή slv\_reg0 του AXI4-Lite. Αντίστοιχα η έξοδος y του φίλτρου μαζί με το σήμα εξόδου valid\_out εισάγονται στο σήμα Β το οποίο κατευθύνεται στον καταχωρητή εξόδου data\_reg\_out στη διεύθυνση που αντιστοιχεί στην τιμή ‘01’ του σήματος loc\_addr.

Ωστόσο, πέρα από αυτές τις συνδέσεις χρειάζεται να κάνουμε και μερικές επιπλέον αλλαγές ώστε να ολοκληρωθεί η ενσωμάτωση της ΑΧΙ διεπαφής με τη λογική του φίλτρου μας.

Αρχικά πρέπει να εξασφαλίσουμε ότι η εγγραφή νέων δεδομένων θα γίνεται σε έναν κύκλο, με άλλα λόγια το valid\_in bit πρέπει να μηδενίζεται όταν δεν γίνεται εγγραφή. Για τον λόγο αυτό προσθέτουμε έναν έλεγχο ο οποίος μηδενίζει το bit 8 του καταχωρητή εισόδου όταν το σήμα slv\_reg\_wren είναι ‘0’.

Ακόμη παρατηρήσαμε ότι ο καταχωρητής εξόδου reg\_data\_out ενημερώνεται μόνο όταν υπάρχει κάποιο read request. Αυτό σημαίνει ότι ο χρήστης πρέπει να συγχρονίσει πλήρως το αίτημά του με την παραγωγή των δεδομένων εξόδου (κατά την οποία το valid\_out είναι ‘1’) για να πάρει έγκυρα δεδομένα, κάτι που είναι πρακτικά αδύνατο. Συνεπώς πρέπει να προσθέσουμε έναν καταχωρητή στον οποίο θα αποθηκεύονται όλες οι έγκυρες τιμές εξόδου του fir καθώς και μία νέα διεργασία η οποία θα τρέχει παράλληλα με τις υπόλοιπες και θα ελέγχει διαρκώς την τιμή του valid\_out. Μόλις αυτό ενεργοποιηθεί η τιμή του ενδιάμεσου καταχωρητή ενημερώνεται και τα δεδομένα είναι πλέον διαθέσιμα στον χρήστη οποιαδήποτε χρονική στιγμή (μέχρι να αντικατασταθούν από νεότερο αποτέλεσμα).

Ο συνολικός κώδικας του ΙP φαίνεται παρακάτω:

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity fir\_ip\_v1\_0\_S00\_AXI is

generic (

-- Users to add parameters here

-- User parameters ends

-- Do not modify the parameters beyond this line

-- Width of S\_AXI data bus

C\_S\_AXI\_DATA\_WIDTH : integer := 32;

-- Width of S\_AXI address bus

C\_S\_AXI\_ADDR\_WIDTH : integer := 4

);

port (

-- Users to add ports here

-- User ports ends

-- Do not modify the ports beyond this line

-- Global Clock Signal

S\_AXI\_ACLK : in std\_logic;

-- Global Reset Signal. This Signal is Active LOW

S\_AXI\_ARESETN : in std\_logic;

-- Write address (issued by master, accepted by Slave)

S\_AXI\_AWADDR : in std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

-- Write channel Protection type. This signal indicates the

-- privilege and security level of the transaction, and whether

-- the transaction is a data access or an instruction access.

S\_AXI\_AWPROT : in std\_logic\_vector(2 downto 0);

-- Write address valid. This signal indicates that the master signaling

-- valid write address and control information.

S\_AXI\_AWVALID : in std\_logic;

-- Write address ready. This signal indicates that the slave is ready

-- to accept an address and associated control signals.

S\_AXI\_AWREADY : out std\_logic;

-- Write data (issued by master, accepted by Slave)

S\_AXI\_WDATA : in std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

-- Write strobes. This signal indicates which byte lanes hold

-- valid data. There is one write strobe bit for each eight

-- bits of the write data bus.

S\_AXI\_WSTRB : in std\_logic\_vector((C\_S\_AXI\_DATA\_WIDTH/8)-1 downto 0);

-- Write valid. This signal indicates that valid write

-- data and strobes are available.

S\_AXI\_WVALID : in std\_logic;

-- Write ready. This signal indicates that the slave

-- can accept the write data.

S\_AXI\_WREADY : out std\_logic;

-- Write response. This signal indicates the status

-- of the write transaction.

S\_AXI\_BRESP : out std\_logic\_vector(1 downto 0);

-- Write response valid. This signal indicates that the channel

-- is signaling a valid write response.

S\_AXI\_BVALID : out std\_logic;

-- Response ready. This signal indicates that the master

-- can accept a write response.

S\_AXI\_BREADY : in std\_logic;

-- Read address (issued by master, accepted by Slave)

S\_AXI\_ARADDR : in std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

-- Protection type. This signal indicates the privilege

-- and security level of the transaction, and whether the

-- transaction is a data access or an instruction access.

S\_AXI\_ARPROT : in std\_logic\_vector(2 downto 0);

-- Read address valid. This signal indicates that the channel

-- is signaling valid read address and control information.

S\_AXI\_ARVALID : in std\_logic;

-- Read address ready. This signal indicates that the slave is

-- ready to accept an address and associated control signals.

S\_AXI\_ARREADY : out std\_logic;

-- Read data (issued by slave)

S\_AXI\_RDATA : out std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

-- Read response. This signal indicates the status of the

-- read transfer.

S\_AXI\_RRESP : out std\_logic\_vector(1 downto 0);

-- Read valid. This signal indicates that the channel is

-- signaling the required read data.

S\_AXI\_RVALID : out std\_logic;

-- Read ready. This signal indicates that the master can

-- accept the read data and response information.

S\_AXI\_RREADY : in std\_logic

);

end fir\_ip\_v1\_0\_S00\_AXI;

architecture arch\_imp of fir\_ip\_v1\_0\_S00\_AXI is

-- AXI4LITE signals

signal axi\_awaddr : std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

signal axi\_awready : std\_logic;

signal axi\_wready : std\_logic;

signal axi\_bresp : std\_logic\_vector(1 downto 0);

signal axi\_bvalid : std\_logic;

signal axi\_araddr : std\_logic\_vector(C\_S\_AXI\_ADDR\_WIDTH-1 downto 0);

signal axi\_arready : std\_logic;

signal axi\_rdata : std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal axi\_rresp : std\_logic\_vector(1 downto 0);

signal axi\_rvalid : std\_logic;

-- Example-specific design signals

-- local parameter for addressing 32 bit / 64 bit C\_S\_AXI\_DATA\_WIDTH

-- ADDR\_LSB is used for addressing 32/64 bit registers/memories

-- ADDR\_LSB = 2 for 32 bits (n downto 2)

-- ADDR\_LSB = 3 for 64 bits (n downto 3)

constant ADDR\_LSB : integer := (C\_S\_AXI\_DATA\_WIDTH/32)+ 1;

constant OPT\_MEM\_ADDR\_BITS : integer := 1;

------------------------------------------------

---- Signals for user logic register space example

--------------------------------------------------

---- Number of Slave Registers 4

signal slv\_reg0 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg1 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg2 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg3 :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal slv\_reg\_rden : std\_logic;

signal slv\_reg\_wren : std\_logic;

signal reg\_data\_out :std\_logic\_vector(C\_S\_AXI\_DATA\_WIDTH-1 downto 0);

signal byte\_index : integer;

signal aw\_en : std\_logic;

signal fir\_out : std\_logic\_vector(31 downto 0) := (others => '0');

signal fir\_valid\_out : std\_logic\_vector(31 downto 0) := (others => '0');

component fir\_filter is

port(

clk: in std\_logic;

rst: in std\_logic;

valid\_in: in std\_logic;

x: in std\_logic\_vector(7 downto 0);

valid\_out: out std\_logic ;

y: out std\_logic\_vector(18 downto 0)

);

end component;

begin

-- I/O Connections assignments

S\_AXI\_AWREADY <= axi\_awready;

S\_AXI\_WREADY <= axi\_wready;

S\_AXI\_BRESP <= axi\_bresp;

S\_AXI\_BVALID <= axi\_bvalid;

S\_AXI\_ARREADY <= axi\_arready;

S\_AXI\_RDATA <= axi\_rdata;

S\_AXI\_RRESP <= axi\_rresp;

S\_AXI\_RVALID <= axi\_rvalid;

-- Implement axi\_awready generation

-- axi\_awready is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are asserted. axi\_awready is

-- de-asserted when reset is low.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_awready <= '0';

aw\_en <= '1';

else

if (axi\_awready = '0' and S\_AXI\_AWVALID = '1' and S\_AXI\_WVALID = '1' and aw\_en = '1') then

-- slave is ready to accept write address when

-- there is a valid write address and write data

-- on the write address and data bus. This design

-- expects no outstanding transactions.

axi\_awready <= '1';

aw\_en <= '0';

elsif (S\_AXI\_BREADY = '1' and axi\_bvalid = '1') then

aw\_en <= '1';

axi\_awready <= '0';

else

axi\_awready <= '0';

end if;

end if;

end if;

end process;

-- Implement axi\_awaddr latching

-- This process is used to latch the address when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are valid.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_awaddr <= (others => '0');

else

if (axi\_awready = '0' and S\_AXI\_AWVALID = '1' and S\_AXI\_WVALID = '1' and aw\_en = '1') then

-- Write Address latching

axi\_awaddr <= S\_AXI\_AWADDR;

end if;

end if;

end if;

end process;

-- Implement axi\_wready generation

-- axi\_wready is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_AWVALID and S\_AXI\_WVALID are asserted. axi\_wready is

-- de-asserted when reset is low.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_wready <= '0';

else

if (axi\_wready = '0' and S\_AXI\_WVALID = '1' and S\_AXI\_AWVALID = '1' and aw\_en = '1') then

-- slave is ready to accept write data when

-- there is a valid write address and write data

-- on the write address and data bus. This design

-- expects no outstanding transactions.

axi\_wready <= '1';

else

axi\_wready <= '0';

end if;

end if;

end if;

end process;

-- Implement memory mapped register select and write logic generation

-- The write data is accepted and written to memory mapped registers when

-- axi\_awready, S\_AXI\_WVALID, axi\_wready and S\_AXI\_WVALID are asserted. Write strobes are used to

-- select byte enables of slave registers while writing.

-- These registers are cleared when reset (active low) is applied.

-- Slave register write enable is asserted when valid address and data are available

-- and the slave is ready to accept the write address and write data.

slv\_reg\_wren <= axi\_wready and S\_AXI\_WVALID and axi\_awready and S\_AXI\_AWVALID ;

process (S\_AXI\_ACLK)

variable loc\_addr :std\_logic\_vector(OPT\_MEM\_ADDR\_BITS downto 0);

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

slv\_reg0 <= (others => '0');

slv\_reg1 <= (others => '0');

slv\_reg2 <= (others => '0');

slv\_reg3 <= (others => '0');

else

loc\_addr := axi\_awaddr(ADDR\_LSB + OPT\_MEM\_ADDR\_BITS downto ADDR\_LSB);

if (slv\_reg\_wren = '1') then

case loc\_addr is

when b"00" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave register 0

slv\_reg0(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"01" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave register 1

slv\_reg1(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"10" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave register 2

slv\_reg2(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when b"11" =>

for byte\_index in 0 to (C\_S\_AXI\_DATA\_WIDTH/8-1) loop

if ( S\_AXI\_WSTRB(byte\_index) = '1' ) then

-- Respective byte enables are asserted as per write strobes

-- slave register 3

slv\_reg3(byte\_index\*8+7 downto byte\_index\*8) <= S\_AXI\_WDATA(byte\_index\*8+7 downto byte\_index\*8);

end if;

end loop;

when others =>

slv\_reg0 <= slv\_reg0;

slv\_reg1 <= slv\_reg1;

slv\_reg2 <= slv\_reg2;

slv\_reg3 <= slv\_reg3;

end case;

else

--disable VALID\_IN when not writing

slv\_reg0(8) <= '0';

end if;

end if;

end if;

end process;

-- Implement write response logic generation

-- The write response and response valid signals are asserted by the slave

-- when axi\_wready, S\_AXI\_WVALID, axi\_wready and S\_AXI\_WVALID are asserted.

-- This marks the acceptance of address and indicates the status of

-- write transaction.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_bvalid <= '0';

axi\_bresp <= "00"; --need to work more on the responses

else

if (axi\_awready = '1' and S\_AXI\_AWVALID = '1' and axi\_wready = '1' and S\_AXI\_WVALID = '1' and axi\_bvalid = '0' ) then

axi\_bvalid <= '1';

axi\_bresp <= "00";

elsif (S\_AXI\_BREADY = '1' and axi\_bvalid = '1') then --check if bready is asserted while bvalid is high)

axi\_bvalid <= '0'; -- (there is a possibility that bready is always asserted high)

end if;

end if;

end if;

end process;

-- Implement axi\_arready generation

-- axi\_arready is asserted for one S\_AXI\_ACLK clock cycle when

-- S\_AXI\_ARVALID is asserted. axi\_awready is

-- de-asserted when reset (active low) is asserted.

-- The read address is also latched when S\_AXI\_ARVALID is

-- asserted. axi\_araddr is reset to zero on reset assertion.

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_arready <= '0';

axi\_araddr <= (others => '1');

else

if (axi\_arready = '0' and S\_AXI\_ARVALID = '1') then

-- indicates that the slave has accepted the valid read address

axi\_arready <= '1';

-- Read Address latching

axi\_araddr <= S\_AXI\_ARADDR;

else

axi\_arready <= '0';

end if;

end if;

end if;

end process;

-- Implement axi\_arvalid generation

-- axi\_rvalid is asserted for one S\_AXI\_ACLK clock cycle when both

-- S\_AXI\_ARVALID and axi\_arready are asserted. The slave registers

-- data are available on the axi\_rdata bus at this instance. The

-- assertion of axi\_rvalid marks the validity of read data on the

-- bus and axi\_rresp indicates the status of read transaction.axi\_rvalid

-- is deasserted on reset (active low). axi\_rresp and axi\_rdata are

-- cleared to zero on reset (active low).

process (S\_AXI\_ACLK)

begin

if rising\_edge(S\_AXI\_ACLK) then

if S\_AXI\_ARESETN = '0' then

axi\_rvalid <= '0';

axi\_rresp <= "00";

else

if (axi\_arready = '1' and S\_AXI\_ARVALID = '1' and axi\_rvalid = '0') then

-- Valid read data is available at the read data bus

axi\_rvalid <= '1';

axi\_rresp <= "00"; -- 'OKAY' response

elsif (axi\_rvalid = '1' and S\_AXI\_RREADY = '1') then

-- Read data is accepted by the master

axi\_rvalid <= '0';

end if;

end if;

end if;

end process;

-- Implement memory mapped register select and read logic generation

-- Slave register read enable is asserted when valid address is available

-- and the slave is ready to accept the read address.

slv\_reg\_rden <= axi\_arready and S\_AXI\_ARVALID and (not axi\_rvalid) ;

process (slv\_reg0, fir\_valid\_out, slv\_reg2, slv\_reg3, axi\_araddr, S\_AXI\_ARESETN, slv\_reg\_rden)

variable loc\_addr :std\_logic\_vector(OPT\_MEM\_ADDR\_BITS downto 0);

begin

-- Address decoding for reading registers

loc\_addr := axi\_araddr(ADDR\_LSB + OPT\_MEM\_ADDR\_BITS downto ADDR\_LSB);

case loc\_addr is

when b"00" =>

reg\_data\_out <= slv\_reg0;

when b"01" =>

reg\_data\_out <= fir\_valid\_out;

when b"10" =>

reg\_data\_out <= slv\_reg2;

when b"11" =>

reg\_data\_out <= slv\_reg3;

when others =>

reg\_data\_out <= (others => '0');

end case;

end process;

-- Output register or memory read data

process( S\_AXI\_ACLK ) is

begin

if (rising\_edge (S\_AXI\_ACLK)) then

if ( S\_AXI\_ARESETN = '0' ) then

axi\_rdata <= (others => '0');

else

if (slv\_reg\_rden = '1') then

-- When there is a valid read address (S\_AXI\_ARVALID) with

-- acceptance of read address by the slave (axi\_arready),

-- output the read dada

-- Read address mux

axi\_rdata <= reg\_data\_out; -- register read data

end if;

end if;

end if;

end process;

-- Add user logic here

FILTER: fir\_filter

port map(

clk => S\_AXI\_ACLK,

rst => slv\_reg0(9),

valid\_in => slv\_reg0(8),

x => slv\_reg0(7 downto 0),

valid\_out => fir\_out(19),

y => fir\_out(18 downto 0)

);

-- A process to update the output only when Valid\_Out is set

process(S\_AXI\_ACLK ,fir\_out) is

begin

if (rising\_edge (S\_AXI\_ACLK)) then

if fir\_out(19) = '1' then

fir\_valid\_out <= fir\_out;

end if;

end if;

end process;

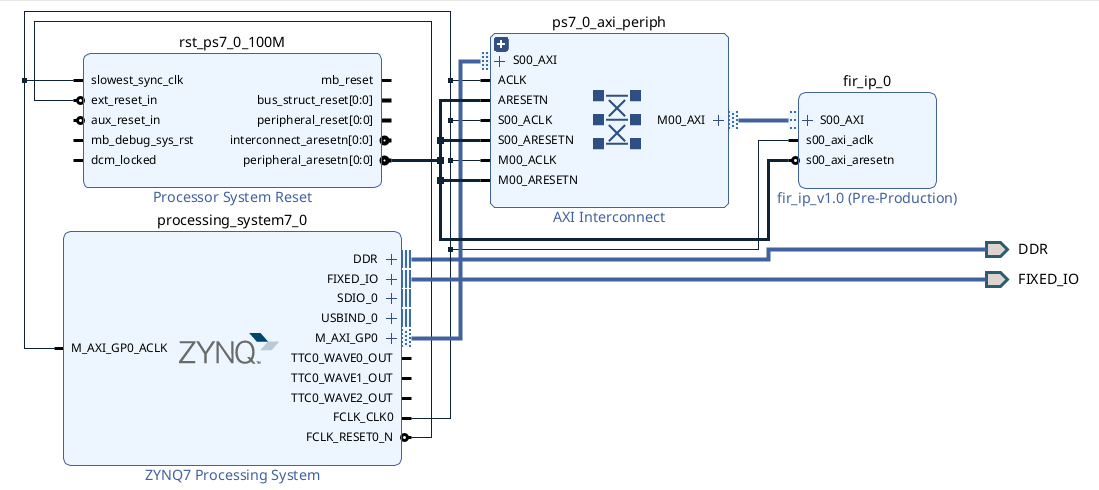
-- User logic ends

end arch\_imp;

Με τις παραπάνω αλλαγές μπορούμε να "πακετάρουμε" το IP που μόλις δημιουργήσαμε και να το χρησιμοποιήσουμε σε κάποιο Vivado Project.

Δημιουργούμε, λοιπόν, ένα νέο project και σχεδιάζουμε ένα block design στο οποίο συμπεριλαμβάνουμε το fir\_ip που μόλις φτιάξαμε και ένα Zynq Processing System. Στη συνέχεια χρησιμοποιούμε τα αυτοματοποιημένα εργαλεία του προγράμματος για να ολοκληρώσουμε το design μας.

To τελικό block design φαίνεται παρακάτω:



Τέλος, αφού επιβεβαιώσουμε ότι δεν υπάρχουν σφάλματα μέσω validation, παράγουμε έναν wrapper και προχωράμε σε synthesis, implementation, bit-stream generation και hardware export.

**Υλοποίηση εφαρμογής λογισμικού**

Για να επιβεβαιώσουμε την ορθή λειτουργία του συστήματος φτιάχνουμε ένα project σε Vitis και γράφουμε κατάλληλο πρόγραμμα σε C το οποίο εκτελείται στον ARM επεξεργαστή της πλακέτας και στέλνει τις εισόδους x ως input στο FPGA ενώ κατόπιν διαβάζει την έξοδο y που παράγεται. Η επικοινωνία μεταξύ του ARM και του επιταχυντή υλικού γίνεται μέσω της διεπαφής AXI4-Lite που υλοποιήσαμε προηγουμένως. Ακολουθεί ο κώδικας της εφαρμογής:

#include <stdio.h>

#include "platform.h"

#include "xil\_printf.h"

#include "xil\_types.h"

#include "xparameters.h"

#include "sleep.h"

// Offsets for control bits

#define VALID\_IN 8

#define RESET 9

#define VALID\_OUT 19

// Write to the first register

#define INPUT\_REG XPAR\_FIR\_IP\_0\_S00\_AXI\_BASEADDR

// Read from the second register

#define OUTPUT\_REG XPAR\_FIR\_IP\_0\_S00\_AXI\_BASEADDR + 4

//uint8\_t values[] = {213,107,172,58,147,225,92,39,205,26,180,99,248,15,134,73};

uint8\_t values[] = {208,231,32,233,161,24,71,140,245,247,40,248,245,124,204,36,107,234,202,245};

void reset(){

uint32\_t input\_data = 0;

// Set Reset Bit

input\_data = 1 << RESET ;

Xil\_Out32(INPUT\_REG,input\_data);

// Wait 1usec

usleep(1);

// Unset Reset Bit

input\_data &= ~(1 << RESET);

Xil\_Out32(INPUT\_REG,input\_data);

xil\_printf("Reset Done\n\r");

}

void write\_data(uint8\_t num){

uint32\_t input\_data = 0;

input\_data = (uint32\_t) num;

// Enable Valid bit

input\_data |= (1 << VALID\_IN );

// Disable Reset bit

input\_data &= ~(1 << RESET);

Xil\_Out32(INPUT\_REG,input\_data);

// Mask

input\_data &= 0xFF;

//xil\_printf("Wrote: %d\n\r",input\_data);

}

void read\_data(){

uint32\_t output = 0;

uint32\_t valid\_out = 0;

while(!valid\_out){

output = Xil\_In32(OUTPUT\_REG);

valid\_out = (output >> VALID\_OUT) & 0x01;

}

// Mask output

output &= 0x7FFFF;

xil\_printf("Read : %d\n\r",output);

}

int main(){

/\* Initialize platform \*/

init\_platform();

sleep(1);

reset();

xil\_printf("Fir Test\n\r");

for(int i=0; i < sizeof(values); ++i){

write\_data(values[i]);

usleep(1);

read\_data();

}

xil\_printf("End of test\n\r");

return 0;

}

**Τελικό Αποτέλεσμα**

Φορτώνουμε το παραπάνω πρόγραμμα στο Zybo το οποίο συνδέεται στη θύρα USB UART του υπολογιστή μας και ελέγχουμε τα αποτελέσματα μέσω serial monitor. Συγκρίνουμε τις τιμές αυτές με τα αποτελέσματα που έχουμε παράξει μέσω του python script της προηγούμενης εργαστηριακής άσκησης και επιβεβαιώνουμε την ορθή λειτουργία ολόκληρου του συστήματος.