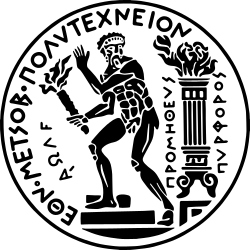
**Εθνικό Μετσόβιο Πολυτεχνείο** 

**Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών**

**Υπολογιστών**

Έτος: 2024- 2025

*Ψηφιακά VLSI*

**Ομάδα 45**

*Παπαδόπουλος Χαράλαμπος* (ΑΜ: 03120199)

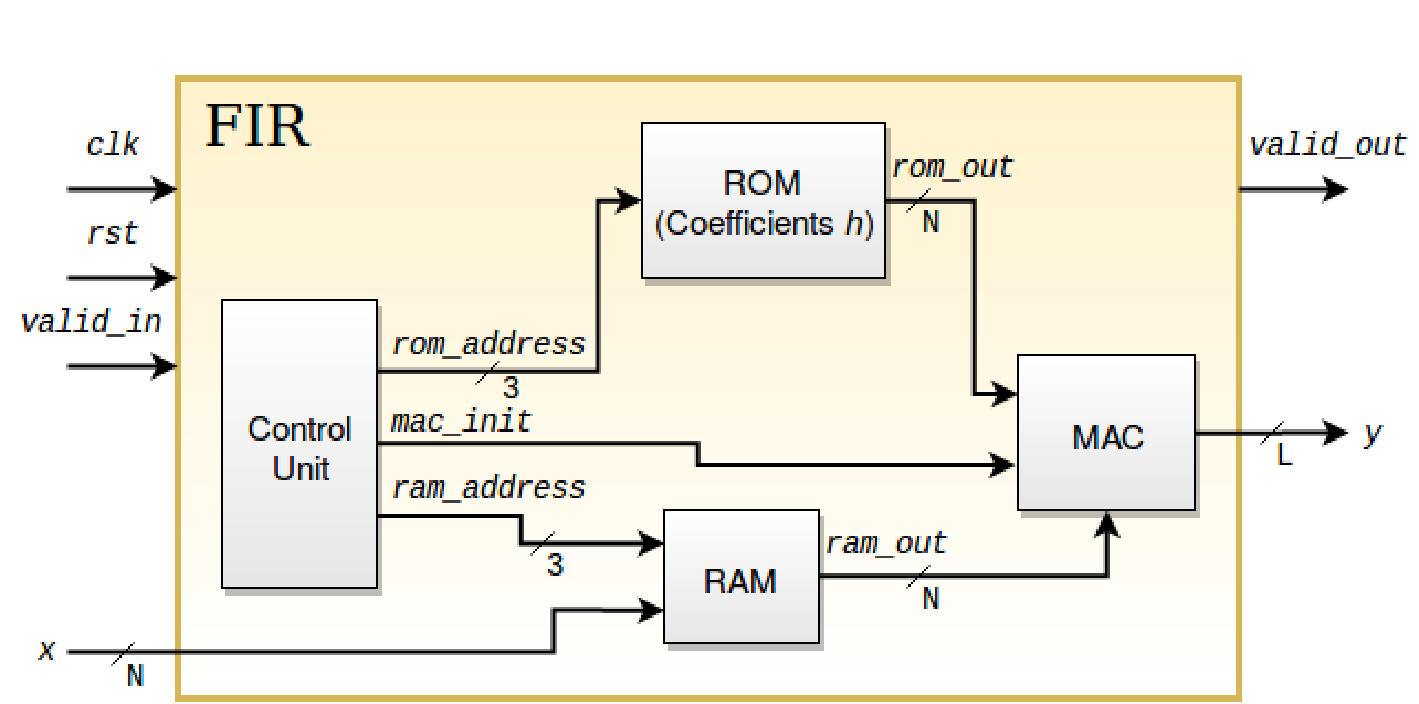
*Νικόλαος Παπακωνσταντόπουλος* (ΑΜ: 03120069)

# **Υλοποίηση FIR φίλτρου με AXI διεπαφή σε ZYNQ SoC FPGA**

Σκοπός αυτής της Άσκησης ήταν ο προγραμματισμός της αναπτυξιακής πλακέτας ZYBO, ώστε

να υλοποιεί ένα FIR φίλτρο, στο οποίο τα δεδομένα εισόδου θα αποστέλλονται από τον ενσωματωμένο επεξεργαστή (ARM) προς το FPGA για επεξεργασία, και αντίστροφα για τα αντίστοιχα αποτελέσματα. Η επικοινωνία επεξεργαστή-FPGA βασίζεται στο πρωτόκολλο AXI.

Χρησιμοποιήθηκε η υλοποίηση ενός 8-tap FIR για N=8 bits εύρος δεδομένων, όπως είχε υλοποιηθεί στην προηγούμενη άσκηση, σύμφωνα με την ακόλουθη αρχιτεκτονική.



Έγινε αντιληπτό ότι το φίλτρο πρέπει να τροποποιηθεί κατάλληλα ώστε να επικοινωνεί με τον ARM και να περιμένει να διαβάσει/γράψει δεδομένα στους καταχωρητές που χρησιμοποιούνται για είσοδο/έξοδο του φίλτρου. Συγκεκριμένα, γράφτηκε το fir\_wrapper.vhd που χρησιμοποιεί τα παραπάνω components με βάση τις παρακάτω παρατηρήσεις:

- Το valid\_out άπαξ και γίνει 1 (έχει παραχθεί έγκυρο αποτέλεσμα), πρέπει να παραμένει 1 μέχρις ότου διαβαστεί από το πρόγραμμα C που τρέχει στον ARM και αποσταλεί νέο δεδομένο προς επεξεργασία για το φίλτρο. Στη συνέχεια, όταν το valid\_in μεταβληθεί από 0 σε 1 (δίνεται νέο δεδομένο στο φίλτρο), πρέπει να μηδενίζεται το valid\_out του φίλτρου (αυτή η τροποποίηση έγινε με χρήση του **last\_stage\_reg**)

- Επειδή το valid\_in γράφεται σε 1 όταν δίνεται είσοδος στο φίλτρο και δεν γνωρίζουμε πόσοι κύκλοι περνούν μέχρις ότου να τροποποιηθεί σε 0, πρέπει το εσωτερικό component FIR να αντιλαμβάνεται ότι το valid\_in είναι 1 μόνο κατά την είσοδο του δεδομένου που δόθηκε από τον ARM (μεταβολή τιμής από 0 σε 1) και μετά 0. Αυτό διότι, αν μετά την ολοκλήρωση των 8 κύκλων που χρειάζονται για τον υπολογισμό του αποτελέσματος, το φίλτρο “βλέπει” valid\_in 1 θα συνεχίσει να κάνει πράξη βάζοντας πάλι το ίδιο δεδομένο στη RAM και τα αποτελέσματα θα είναι άκυρα - μη αναμενόμενα. Συνεπώς, με χρησιμοποιήθηκε το **modify\_valid\_in\_proc** ώστε το εσωτερικό valid\_in γίνεται 1 μόνο όταν το εξωτερικό valid\_in μεταβάλλεται από 0 σε 1.

## fir\_wrapper.vhd

library IEEE;

use IEEE.std\_logic\_1164.ALL;

use IEEE.numeric\_std.all;

entity fir\_wrapper is

Port (

valid\_in : in std\_logic;

clk : in std\_logic;

rst : in std\_logic;

x : in std\_logic\_vector(8-1 downto 0);

valid\_out : out std\_logic;

y : out std\_logic\_vector(19-1 downto 0)

-- deleted debug ports and additional ones

);

end fir\_wrapper;

architecture structural of fir\_wrapper is

component fir is

Port (

valid\_in : in std\_logic;

clk : in std\_logic;

rst : in std\_logic;

x : in std\_logic\_vector(8-1 downto 0);

valid\_out : out std\_logic;

y : out std\_logic\_vector(19-1 downto 0);

-- Debug ports

dbg\_counter : out std\_logic\_vector(2 downto 0);

dbg\_rom\_out : out std\_logic\_vector(7 downto 0);

dbg\_ram\_out : out std\_logic\_vector(7 downto 0);

dbg\_mac\_init : out std\_logic;

dbg\_we : out std\_logic;

-- Additional debug ports

dbg\_x\_register : out std\_logic\_vector(7 downto 0);

dbg\_ram\_di : out std\_logic\_vector(7 downto 0);

dbg\_ram\_addr : out std\_logic\_vector(2 downto 0);

dbg\_mac\_acc : out std\_logic\_vector(18 downto 0)

);

end component;

-- signal declarations

signal previous\_valid\_in: std\_logic := '0';

signal valid\_in\_modified : std\_logic := '0';

signal x\_register: std\_logic\_vector(7 downto 0) := (others => '0');

signal y\_internal: std\_logic\_vector(19-1 downto 0) := (others => '0');

signal valid\_out\_internal: std\_logic ;

begin

fir\_inst: fir

port map (

valid\_in => valid\_in\_modified,

clk => clk,

rst => rst,

x => x\_register,

valid\_out => valid\_out\_internal,

y => y\_internal

);

modify\_valid\_in\_proc: process(clk, rst)

begin

if rst = '1' then

valid\_in\_modified <= '0';

elsif rising\_edge(clk) then

if valid\_in = '1' and previous\_valid\_in = '0' then

valid\_in\_modified <= '1';

else

valid\_in\_modified <= '0';

end if;

previous\_valid\_in <= valid\_in;

end if;

end process;

x\_reg\_proc: process(clk, rst)

begin

if rst = '1' then

x\_register <= (others => '0');

elsif rising\_edge(clk) then

x\_register <= x;

end if;

end process;

last\_stage\_reg: process(clk, rst)

begin

if rst = '1' then

y <= (others => '0');

valid\_out <= '0';

elsif rising\_edge(clk) then

if valid\_in\_modified = '1' then

y <= (others => '0');

valid\_out <= '0';

elsif valid\_out\_internal = '1' then

y <= y\_internal;

valid\_out <= '1';

end if;

end if;

end process;

end structural;

## FIR\_v1\_0\_AXI.vhd

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity FIR\_v1\_0\_FIR\_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, acceped 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, acceped 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, acceped 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\_v1\_0\_FIR\_AXI;

architecture arch\_imp of FIR\_v1\_0\_FIR\_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;

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 registor 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 registor 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 registor 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 registor 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;

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 acceped 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, slv\_reg1, 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 <= slv\_reg1;

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

fir\_ip: entity work.fir\_wrapper Port map

(

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

clk => S\_AXI\_ACLK,

rst => slv\_reg0(9),

x => slv\_reg0(8-1 downto 0),

valid\_out => slv\_reg1(19),

y => slv\_reg1(19-1 downto 0)

);

-- User logic ends

end arch\_imp;

Στο τέλος του κώδικα φαίνεται η λογική αντιστοίχισης των καταχωρητών Α -slv\_reg0 (εισόδου) και Β - slv\_reg1 (εξόδου) με τα αντίστοιχα pin του FIR.

Χρειάστηκε να αφαιρεθεί μια γραμμή που μηδένιζε το aw\_en ώστε να γίνεται σωστά η επικοινωνία με το AXI καθώς και κάποιες γραμμές που έγραφαν το slv\_reg1 (για λόγους conflict ταυτόχρονο γράψιμο με το FIR module)

## FIR\_v1\_0.vhd

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity FIR\_v1\_0 is

generic (

-- Users to add parameters here

-- User parameters ends

-- Do not modify the parameters beyond this line

-- Parameters of Axi Slave Bus Interface FIR\_AXI

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

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

);

port (

-- Users to add ports here

-- User ports ends

-- Do not modify the ports beyond this line

-- Ports of Axi Slave Bus Interface FIR\_AXI

fir\_axi\_aclk : in std\_logic;

fir\_axi\_aresetn : in std\_logic;

fir\_axi\_awaddr : in std\_logic\_vector(C\_FIR\_AXI\_ADDR\_WIDTH-1 downto 0);

fir\_axi\_awprot : in std\_logic\_vector(2 downto 0);

fir\_axi\_awvalid : in std\_logic;

fir\_axi\_awready : out std\_logic;

fir\_axi\_wdata : in std\_logic\_vector(C\_FIR\_AXI\_DATA\_WIDTH-1 downto 0);

fir\_axi\_wstrb : in std\_logic\_vector((C\_FIR\_AXI\_DATA\_WIDTH/8)-1 downto 0);

fir\_axi\_wvalid : in std\_logic;

fir\_axi\_wready : out std\_logic;

fir\_axi\_bresp : out std\_logic\_vector(1 downto 0);

fir\_axi\_bvalid : out std\_logic;

fir\_axi\_bready : in std\_logic;

fir\_axi\_araddr : in std\_logic\_vector(C\_FIR\_AXI\_ADDR\_WIDTH-1 downto 0);

fir\_axi\_arprot : in std\_logic\_vector(2 downto 0);

fir\_axi\_arvalid : in std\_logic;

fir\_axi\_arready : out std\_logic;

fir\_axi\_rdata : out std\_logic\_vector(C\_FIR\_AXI\_DATA\_WIDTH-1 downto 0);

fir\_axi\_rresp : out std\_logic\_vector(1 downto 0);

fir\_axi\_rvalid : out std\_logic;

fir\_axi\_rready : in std\_logic

);

end FIR\_v1\_0;

architecture arch\_imp of FIR\_v1\_0 is

-- component declaration

component FIR\_v1\_0\_FIR\_AXI is

generic (

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

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

);

port (

S\_AXI\_ACLK : in std\_logic;

S\_AXI\_ARESETN : in std\_logic;

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

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

S\_AXI\_AWVALID : in std\_logic;

S\_AXI\_AWREADY : out std\_logic;

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

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

S\_AXI\_WVALID : in std\_logic;

S\_AXI\_WREADY : out std\_logic;

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

S\_AXI\_BVALID : out std\_logic;

S\_AXI\_BREADY : in std\_logic;

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

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

S\_AXI\_ARVALID : in std\_logic;

S\_AXI\_ARREADY : out std\_logic;

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

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

S\_AXI\_RVALID : out std\_logic;

S\_AXI\_RREADY : in std\_logic

);

end component FIR\_v1\_0\_FIR\_AXI;

begin

-- Instantiation of Axi Bus Interface FIR\_AXI

FIR\_v1\_0\_FIR\_AXI\_inst : FIR\_v1\_0\_FIR\_AXI

generic map (

C\_S\_AXI\_DATA\_WIDTH => C\_FIR\_AXI\_DATA\_WIDTH,

C\_S\_AXI\_ADDR\_WIDTH => C\_FIR\_AXI\_ADDR\_WIDTH

)

port map (

S\_AXI\_ACLK => fir\_axi\_aclk,

S\_AXI\_ARESETN => fir\_axi\_aresetn,

S\_AXI\_AWADDR => fir\_axi\_awaddr,

S\_AXI\_AWPROT => fir\_axi\_awprot,

S\_AXI\_AWVALID => fir\_axi\_awvalid,

S\_AXI\_AWREADY => fir\_axi\_awready,

S\_AXI\_WDATA => fir\_axi\_wdata,

S\_AXI\_WSTRB => fir\_axi\_wstrb,

S\_AXI\_WVALID => fir\_axi\_wvalid,

S\_AXI\_WREADY => fir\_axi\_wready,

S\_AXI\_BRESP => fir\_axi\_bresp,

S\_AXI\_BVALID => fir\_axi\_bvalid,

S\_AXI\_BREADY => fir\_axi\_bready,

S\_AXI\_ARADDR => fir\_axi\_araddr,

S\_AXI\_ARPROT => fir\_axi\_arprot,

S\_AXI\_ARVALID => fir\_axi\_arvalid,

S\_AXI\_ARREADY => fir\_axi\_arready,

S\_AXI\_RDATA => fir\_axi\_rdata,

S\_AXI\_RRESP => fir\_axi\_rresp,

S\_AXI\_RVALID => fir\_axi\_rvalid,

S\_AXI\_RREADY => fir\_axi\_rready

);

-- Add user logic here

-- User logic ends

end arch\_imp;

## helloworld.c

#include <stdio.h>

#include "platform.h"

#include "xil\_printf.h"

#include "xparameters.h"

#include "FIR.h"

#include <xil\_io.h>

#include <unistd.h>

int main()

{

init\_platform();

setvbuf (stdin, NULL, \_IONBF, 0);

FIR\_mWriteReg(XPAR\_FIR\_0\_FIR\_AXI\_BASEADDR, FIR\_FIR\_AXI\_SLV\_REG0\_OFFSET, (uint32\_t) (1 << 9) );

sleep(1);

FIR\_mWriteReg(XPAR\_FIR\_0\_FIR\_AXI\_BASEADDR, FIR\_FIR\_AXI\_SLV\_REG0\_OFFSET, (uint32\_t) 0 );

sleep(1);

putchar('a');

printf("initialized \n\n");

while (1) {

int temp\_c, temp\_flags;

unsigned char c, flags;

scanf("%d", &temp\_c);

temp\_flags = 1;

c = (unsigned char)temp\_c;

flags = (unsigned char)temp\_flags;

printf("\n c = %d, flags = %d only flags = %d \n", c, flags, flags & 0x03);

FIR\_mWriteReg(XPAR\_FIR\_0\_FIR\_AXI\_BASEADDR, FIR\_FIR\_AXI\_SLV\_REG0\_OFFSET, (uint32\_t) (((flags & 0x02) >> 1) << 9) );

printf("b ");

uint32\_t n;

sleep(1);

FIR\_mWriteReg(XPAR\_FIR\_0\_FIR\_AXI\_BASEADDR, FIR\_FIR\_AXI\_SLV\_REG0\_OFFSET, (((flags & 0x02) >> 1) << 9) | ((flags & 0x01) << 8) | (uint32\_t) c);

printf("c\n");

/\*for (int i=0; i<20; i++)\*/

while (1){

n = FIR\_mReadReg(XPAR\_FIR\_0\_FIR\_AXI\_BASEADDR, FIR\_FIR\_AXI\_SLV\_REG1\_OFFSET);

//printf("valid\_out = %d \n", n, n & (1<<19));

if (n & (1<<19)) break;

}

printf("%lu ", n & ((1<<19)-1));

}

cleanup\_platform();

return 0;

}

Στην έναρξη της main() κάνουμε reset με εγγραφή λογικού 1 στο αντίστοιχο - bit (bit 9) του AXI register 0 για την είσοδο Α του FIR και μετά απενεργοποιούμε το reset γράφοντας στο bit αυτό 0. Ύστερα σε κάθε επανάληψη:

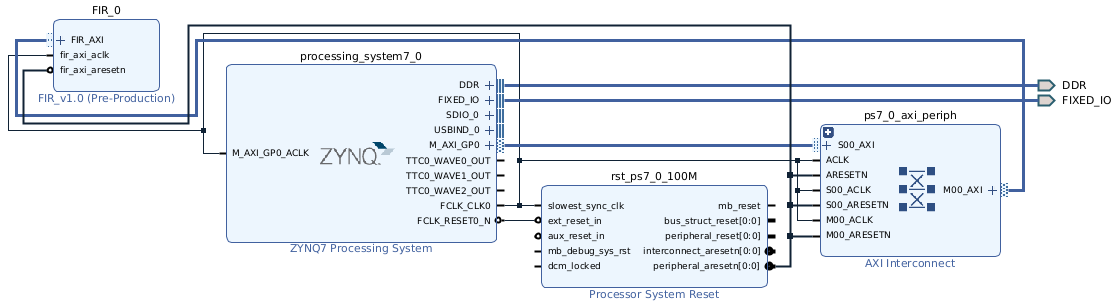
- διαβάζουμε αριθμό από σειριακή κονσόλα (χρήστη) για το νέο δεδομένο εισόδου του φίλτρου

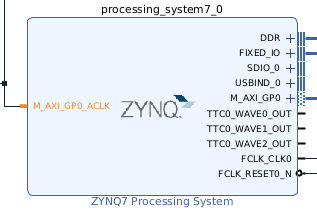
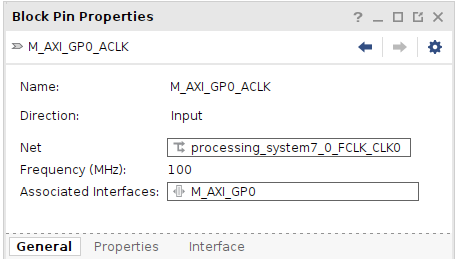
- πρώτα κάνουμε το valid\_in (bit 8) ίσο με 0 (FIR\_mWriteReg μονο με το rst flag και τα άλλα bits είναι 0)

- μετά γράφουμε το νέο αριθμό εισόδου μαζί με rst και valid\_in όπως είναι στα flags (εδώ έχει δηλωθεί rst = 0, valid\_in = 1)

- έπειτα διαβάζουμε διαρκώς με FIR\_mReadReg τον ΑΧΙ register 1 που αντιστοιχεί στην έξοδο Β του FIR μέχρις ότου το valid\_out (bit 19) να είναι 1 (ελήφθη έγκυρο αποτέλεσμα) και τότε εκτυπώνουμε το αποτέλεσμα (bits 0 - 18)

## Block Design





Συχνότητα Λειτουργίας 100 MHz (πατώντας στο αντίστοιχο pin - clock του AXI)