5η Εργαστηριακή Άσκηση στο μάθημα

Ψηφιακά Συστήματα VLSI

Ομάδα 13 :

Γρίβας Αριστοτέλης – el19889

Αυγουστής Μολτσάνοβ Εμίλ – el17064

Άσκηση :

Ζητούμενο της εργαστηριακής άσκησης είναι 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 θα βασίζεται στο πρωτόκολλο AXI. Η υλοποίηση του συστήματος χωρίζεται στα παρακάτω βήματα:

1) Εισαγωγή του ZYNQ Processing System (PS).

2) Εισαγωγή της AXI4-LIte διεπαφής στο FPGA (PL) για την πραγματοποίηση της επικοινωνίας ARM-FPGA.

3) Διασύνδεση του PS με PL.

4) Σύνθεση και υλοποίηση του συστήματος και παραγωγή του bitstream αρχείου.

5) Εξαγωγή της περιγραφής του συστήματος και δημιουργία της εφαρμογής.

6) Προγραμματισμός του ZYNQ SoC FPGA και εκτέλεση της εφαρμογής.

Αρχικά, φτιάχνοντας σαν custom ip το FIR φίλτρο χρησιμοποιώντας τον ίδιο κώδικα με την 4η εργαστηριακή άσκηση δίχως να κάνουμε κάποια αλλαγή, παρατηρήθηκε ότι όταν τρέχουμε το SDK, τα outputs δεν ήταν ορθά. Για παράδειγμα αν αρχικά X(0)=1, το πρόγραμμα αντί να βγάζει output Y=1, έβγαζε Y=36. Όμως, 36=1+2+3+4+5+6+7+8=h(0)x(0)+ h(1)x(0)+ h(2)x(0)+ h(3)x(0)+ h(4)x(0)+ h(5)x(0)+ h(6)x(0)+ h(7)x(0). Δηλαδή, μέχρι το sdk να εμφανίσει το output, περνούσαν αρκετοί κύκλοι ρολογιού ώστε και οι 8 θέσεις της ram να προλάβουν να πάρουν την τιμή της εισόδου x(0), η οποία διατηρούνταν σταθερή.

Οπότε, η λύση που σκεφτήκαμε είναι όταν περάσουν οι πρώτοι 8 κύκλοι ρολογιού από τη στιγμή που το valid_in έγινε 1, θα πρέπει να εκτελούνται τρία πράγματα:

Πρώτον, θα πρέπει η είσοδος we της ram να γίνεται 0 και να παραμένει 0 όσο το valid_in συνεχίζει να είναι 1.

Δεύτερον, η έξοδος Y που υπολογίστηκε στο τέλος των 8 παλμών θα πρέπει να παραμένει σταθερή όσο το valid_in συνεχίζει να είναι 1.

Τρίτον, όταν το valid_out γίνει 1 θα πρέπει να παραμένει σταθερό όσο το valid_in συνεχίζει να είναι 1.

Το πρώτο και τρίτο ζητούμενο μπορούν να υλοποιηθούν με δύο ασύγχρονους μετρητές του 1 bit (T flip flop με είσοδο 1) και δύο πολυπλέκτες. Η λογική είναι ότι στην 1η θύρα του πολυπλέκτη μπαίνει ένα σήμα από την control unit, που ανιχνεύει πότε πέρασαν 8 παλμοί ρολογιού από τη στιγμή που το valid_in έγινε 1, και στη 2η θύρα μπαίνει ένα 0.Η έξοδος του πολυπλέκτη συνδέεται στην είσοδο ρολογιού του T flip flop, και στον επιλογέα του πολυπλέκτη συνδέεται η έξοδος του T flip flop. Επίσης το T flip flop έχει μία

ασύγχρονη είσοδο reset, στην οποία συνδέεται ένα σήμα που γίνεται 1 μόνο όταν το valid_in γίνεται 0 και έχουν ήδη περάσει 8 παλμοί ρολογιού από τη στιγμή που το valid_in έγινε 1.Οπότε η λογική είναι ότι αρχικά η έξοδος του T flip flop είναι 0, οπότε στην είσοδο ρολογιού του περνάει η 1η θύρα του πολυπλέκτη. Μόλις εκτελεστούν 8 παλμοί ρολογιού από τη στιγμή που το valid_in έγινε 1, μέσω του σήματος που συνδέεται στην 1η θύρα, η είσοδος του ρολογιού θα πάρει την τιμή 1 και η έξοδος του T flip flop θα αντιστραφεί από 0 σε 1.Οπότε τώρα στην είσοδο ρολογιού θα περνάει η 2η θύρα, δηλαδή το 0. Οπότε η έξοδος του flip flop κλείδωσε στο 1 και θα παραμένει συνεχώς όσο το valid_in παραμένει 1.Μόνο όταν γίνει 0 το valid_in το flip flop θα αρχικοποιείται στο 0 ασύγχρονα. Την έξοδο του T flip flop τώρα μπορούμε να την συνδέουμε στον επιλογέα ενός άλλου πολυπλέκτη, στην 1η θύρα του οποίου συνδέεται το σήμα που πριν έκανε enable την εγγραφή στη ram, στην 2η θύρα συνδέεται ένα 0, και η έξοδος του συνδέεται στην είσοδο we της ram. Οπότε έτσι όταν κλειδώνει η έξοδος του φλιπ φλοπ στο 1, κλειδώνει η είσοδος we στο 0, οπότε δεν γίνεται καινούργια εγγραφή στη ram όσο το valid_in παραμένει 1. Ακριβώς παρόμοια λογική έχουμε και με το valid_out. Όταν κλειδώνει η έξοδος του 2ου φλιπ φλοπ στο 1, κλειδώνει και η έξοδος valid_out στο 1, όσο το valid_in παραμένει 1. Τέλος, για το 2ο ζητούμενο, αυτό που χρειάζεται να κάνουμε είναι στην μονάδα mac είναι στο D flip flop(19 bit) που η έξοδος του ήταν το Y, την είσοδο αυτού του φλιπ φλοπ πρέπει να την συνδέσουμε σε έναν πολυπλέκτη, στην 1η θύρα του οποίου συνδέεται το αποτέλεσμα του πολλαπλασιασμού και της πρόσθεσης, στην 2η θύρα συνδέεται η έξοδος του D flip flop,και σαν επιλογέα έχουμε το valid_out. Έτσι, όσο valid_out=1 το D flip flop θα διατηρεί το περιεχόμενό του σταθερό, δηλαδή την έξοδο Y.

Με αυτές τις αλλαγές, θα έχουμε τα ακόλουθα:

Αρχικά αν πατήσουμε rst, όλα θα μηδενιστούν.

Έπειτα αν δώσουμε είσοδο χ(0) και κάνουμε το valid_in 1, μετά από 8 παλμούς θα έχουμε έξοδο Y=h(0)x(0), η οποία θα παραμένει σταθερή μέχρι το valid _in να μηδενιστεί.

Έπειτα αν μηδενίσουμε το valid_in, δώσουμε είσοδο χ(1) και κάνουμε το valid_in πάλι 1, μετά από 8 παλμούς θα έχουμε έξοδο Y=h(0)x(1)+h(1)x(0), η οποία θα παραμένει σταθερή μέχρι το valid _in να μηδενιστεί.

Έπειτα αν μηδενίσουμε το valid_in, δώσουμε είσοδο χ(2) και κάνουμε το valid_in πάλι 1, μετά από 8 παλμούς θα έχουμε έξοδο Y=h(0)x(2)+h(1)x(1)+h(2)x(0), η οποία θα παραμένει σταθερή μέχρι το valid _in να μηδενιστεί...κτλ.

Ανανεομένος κώδικας vhdl του FIR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity FIR is
port(
clk,rst,valid_in: in std_logic;
x : in std_logic_vector(7 downto 0);
valid_out: out std_logic;
--control_out : out std_logic_vector(3 downto 0);
```

```
y : out std_logic_vector(18 downto 0));
end FIR;

architecture Behavioral of FIR is

component dff_1bit is
port(
D,rst,clk : in std_logic;
Q : out std_logic);
end component;

component tff is
port(
T,rst,clk : in std_logic;
Q : out std_logic);
end component;

component mac is
port(
init,clk,rst,mux_dff : in std_logic;
x,h  : in std_logic_vector(7 downto 0);
y_out    : out std_logic_vector(18 downto 0));
end component;

component control is
```

```
port(
clk,rst,en : in std_logic;
address : out std_logic_vector(3 downto 0);
mac, to_tff, to_control : out std_logic);
end component;

component mlab_ram is
port (
rst  : in std_logic;
clk  : in std_logic;
we   : in std_logic;
addr : in std_logic_vector(2 downto 0);
di   : in std_logic_vector(7 downto 0);
do   : out std_logic_vector(7 downto 0));
end component;

component mlab_rom is
Port(
clk : in  STD_LOGIC;
addr : in  STD_LOGIC_VECTOR (2 downto 0);
rom_out : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component asynq_counter is
port(
```

```
rst,clk : in std_logic;

Q : out std_logic_vector(9 downto 0));

end component;


signal
zero_ff,valid_and,valid_out_intr,mac_intr,clk_tff,we_intr,tff_to_dff,
en_intr,
not_zero,initialize,rst_2nd_tff,rst_3rd_tff,clk_2nd_tff,clk_3rd_tff,val
id_out_intr2 : std_logic;

signal
asynq0,asynq1,asynq2,asynq3,asynq4,asynq5,asynq6,asynq7,asynq8
,asynq9,reset_asynq, disable_we,we_ram : std_logic;

signal address_intr : std_logic_vector(3 downto 0);

signal ram_out,rom_out_intr : std_logic_vector(7 downto 0);


begin


valid_and <= valid_in and valid_out_intr;

we_intr <= mac_intr and valid_in; -- read input if (counter =0 and
valid_in=1)

zero_ff <= rst or valid_out_intr;  --reset ffs if rst=1 or if valid_out=1

en_intr <= valid_in or not_zero;  --enable counter if valid_in=1 or if
counter/=0



c1: control port map(clk=>clk, rst=>rst, en => en_intr,
address=>address_intr, mac=>mac_intr,to_tff=>clk_tff, to_control
=> not_zero);

t1: tff port map(clk=> clk_tff, T=>'1',rst=>zero_ff, Q=> tff_to_dff);
```

```
d1: dff_1bit port map(clk => clk, rst=>rst, D => tff_to_dff, Q =>
valid_out_intr);




ram1: mlab_ram port map(rst=>rst, clk=>clk, we=>we_ram,
addr=>address_intr(2 downto 0),di=>x,do=>ram_out);

rom1: mlab_rom port map(clk=>clk, addr=>address_intr(2 downto
0),rom_out=>rom_out_intr);

m1  : mac port map(clk=>clk, rst=>rst, init=> we_intr,
mux_dff=>valid_out_intr2, x=>ram_out, h=>rom_out_intr,
y_out=>y);



rst_2nd_tff <= rst or ((not valid_in) and disable_we and mac_intr);


t2: tff port map(clk=> clk_2nd_tff, T=>'1',rst=>rst_2nd_tff, Q=>
disable_we);


with disable_we select

clk_2nd_tff <= tff_to_dff when '0',

        '0' when '1',

       '0' when others;


with disable_we select

we_ram <= we_intr when '0',

        '0' when '1',
```

```
        '0' when others;


with valid_out_intr2 select
clk_3rd_tff <= valid_out_intr when '0',
              '0' when '1',
              '0' when others;


rst_3rd_tff <= rst or ((not valid_in) and valid_out_intr2 and
mac_intr);


t3: tff port map(clk=> clk_3rd_tff, T=>'1',rst=>rst_3rd_tff, Q=>
valid_out_intr2);


valid_out <= valid_out_intr2;


end Behavioral;
```

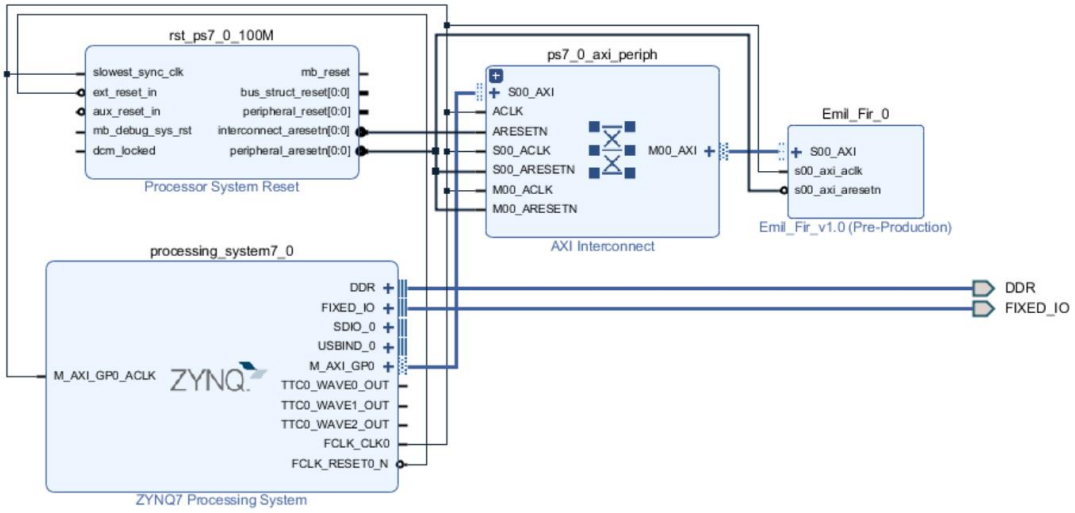
Ανανεομένος κώδικας vhdl της MAC

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;


entity mac is
port(
```

```
init,clk,rst,mux_dff : in std_logic;
x,h  : in std_logic_vector(7 downto 0);
y_out    : out std_logic_vector(18 downto 0));
end mac;

architecture Behavioral of mac is

component dff is
port(
D       : in std_logic_vector(18 downto 0);
clk,rst     : in std_logic;
Q       : out std_logic_vector(18 downto 0));
end component;

signal y_inter : std_logic_vector(18 downto 0);
signal p : std_logic_vector(15 downto 0);
signal sum,sig,sig_dff,gate: std_logic_vector(18 downto 0);

begin

with init select

gate <= "0000000000000000000" when '1',
        "1111111111111111111" when '0',
        "0000000000000000000" when others;
```

```
p <= x * h;
sum <= p+y_inter;
sig <= sum and gate;

with mux_dff select

sig_dff <= sig when '0',
           y_inter when '1',
           sig when others;

d1: dff port map(clk => clk, D => sig_dff, Q => y_inter, rst=>rst);

y_out <= y_inter;

end Behavioral;
```

Παραθέτουμε το design Schematic στο Vivado(αποτελείται από το Fir IP καθώς και το Zynq):



Καθώς και τον κώδικα του IP στον οποίο προσθέσαμε δικά μας στοιχεία. Στον παρακάτω κώδικα,αυτά που αλλάξαμε είναι τα εξής :

-Ορίσαμε ως component το FIR φίλτρο μας και το προσθέσαμε στο IP,όπως θα κάναμε σε μία structural αρχιτεκτονική.

-Προσθέσαμε ένα σήμα “output : std_logic_vector(31 downto0)” για το αποτέλεσμα.

-Ορίσαμε τις εισόδους στον καταχωρητή slv_reg_0, με βάση την εκφώνηση της άσκησης.

--Αντικαταστήσαμε τον slv_reg_1 με το σήμα output.

--Ορίσαμε τις εξόδους στον καταχωρητή output, με βάση την εκφώνηση της άσκησης.

Ο κώδικας του IP ακολουθεί παρακάτω :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Emil_Fir_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, 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 Emil_Fir_v1_0_S00_AXI;
```

```
architecture arch_imp of Emil_Fir_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 output: std_logic_vector(31 downto 0);


	component FIR is

  port(

  clk,rst,valid_in: in std_logic;

  x : in std_logic_vector(7 downto 0);

  valid_out: out std_logic;
```

```
   --control_out : out std_logic_vector(3 downto 0);
   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';
          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_aready 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_aready = '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_aready and S_AXI_ARVALID and (not axi_rvalid) ;

  process (slv_reg0,output, 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 <= output;
            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_aready),
                -- 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
F1: FIR port map(clk=>S_AXI_ACLK,x=>slv_reg0(7 downto
0),valid_in=>slv_reg0(8),rst=>slv_reg0(9),y=>output(18 downto
0),valid_out=>output(19));
  -- User logic ends


end arch_imp;
```

Τέλος,αφού τρέξουμε το “generate bitstream” και βεβαιωθούμε ότι όλα δουλεύουν όπως θα θέλαμε,ανοίγουμε το SDK ώστε να βεβαιωθούμε για την ορθή λειτουργία του FIR. Ακολουθεί ο κώδικας C στο SDK:

Κώδικας C:

```
#include "xparameters.h"

#include "xil_io.h"

#include "xbasic_types.h"

#include "sleep.h"


int main()

{

    uint32_t output;

    uint32_t a[20] =

    {0x000001a7,0x00000109,0x000001d9,0x000001ef,


    0x000001ad,0x000001c1,0x000001be,0x00000164,0
x000001a7,


    0x0000012b,0x000001b4,0x00000108,0x00000146,0
x0000010b,
```

```
    0x00000118,0x000001d2,0x000001b1,0x00000151,0
x000001f3,0x00000108};



    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
0x00000200);

    usleep(20000);

    output =
Xil_In32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR+4);

    xil_printf("output is equal to %d \n",output);


    for(int i = 0; i<20; i++){

    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
0x00000000);

    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
a[i]);

    output =
Xil_In32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR+4);

    xil_printf("output is equal to %d
\n",output);

    usleep(20000);


    }
```

```
    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
0x00000200);

    usleep(20000);

    output =
Xil_In32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR+4);

    xil_printf("output is equal to %d \n",output);


    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
0x000000a7);

    Xil_Out32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR,
0x000001a7);

    usleep(20000);

    output =
Xil_In32(XPAR_EMIL_FIR_0_S00_AXI_BASEADDR+4);

    xil_printf("output is equal to %d \n",output -
524288);


    return 0;

}
```

Σχόλια :

Στον παραπάνω κώδικα κάνουμε τα εξής :

Αρχικά αποθηκεύουμε τις τιμές εισόδου στον πίνακα “a”.Οι τιμές είναι αυτές που δόθηκαν στο μάθημα για την επαλήθευση της ορθής λειτουργίας του FIR φίλτρου της προηγούμενης εργαστηριακής άσκησης.

Στη συνέχεια, κάνουμε reset στο κύκλωμα στέλνοντας στη διεύθυνση

“XPAR_EMIL_FIR_0_S00_AXI_BASEADDR” (slave_reg_0), την τιμή εισόδου “0x00000200” (reset = 1), και διαβάζουμε το αποτέλεσμα από τη διεύθυνση **“XPAR_EMIL_FIR_0_S00_AXI_BASEADDR+4”,**η οποία είναι η διεύθυνση της μεταβλητής εξόδου που ορίσαμε στη θέση του slave_reg_1 και είναι 4 θέσεις(32 bits),μετά από την διεύθυνση εγγραφής(η τιμή εξόδου είναι αναμενόμενα 0).

Έπειτα, δημιουργούμε μία loop η οποία κάνει το valid_in = 0 ώστε να μπορεί να αρχίσει νέα εγγραφή,και στη συνέχεια valid_in = 1 ώστε να γίνει η εγγραφή της νέας τιμής (η οποία δίνεται από τον πίνακα a), και το αποτέλεσμα τυπώνεται όπως περιγράψαμε παραπάνω.

Συνεπώς,η loop αυτή στέλνει στο FIR όλες τις τιμές εισόδου του πίνακα a,και τυπώνει τα αντίστοιχα αποτελέσματα .

Μετά τη loop κάνουμε άλλη μία φορά reset (στέλνοντας 0x00000200),και στέλνουμε άλλη μία τιμή.

Παρατηρούμε,ότι όλες οι λειτουργίες δουλεύουν όπως ήταν επιθυμητό.

Τέλος,ιδιαίτερα σημαντικό είναι το γεγονός πως στη έξοδο δίνουμε την τιμή “B[31...0] - 524288” (όπου

B ο καταχωρητής της τιμή εξόδου).Αυτό το κάνουμε επειδή,η τιμή B[19] = 524288 είναι η τιμή του valid_out,η οποία είναι 1 όταν τυπώνουμε,με αποτέλεσμα η έξοδος μας να έχει ένα offset αυτής της τάξης. Επομένως,για να βλέπουμε καθαρά και μόνο το αποτέλεσμα του FIR (δηλ. B[18...0]), αφαιρούμε το Valid_out offset.

Οι τιμές εξόδου δεν περιέχονται στην αναφορά,καθώς δεν έχουμε την πλακέτα ZYBO στη διάθεση μας για να δούμε τα αποτελέσματα,ωστόσο κατά την εξέταση της άσκησης έγινε επίδειξη των αποτελεσμάτων,τα οποία ήταν και τα σωστά.