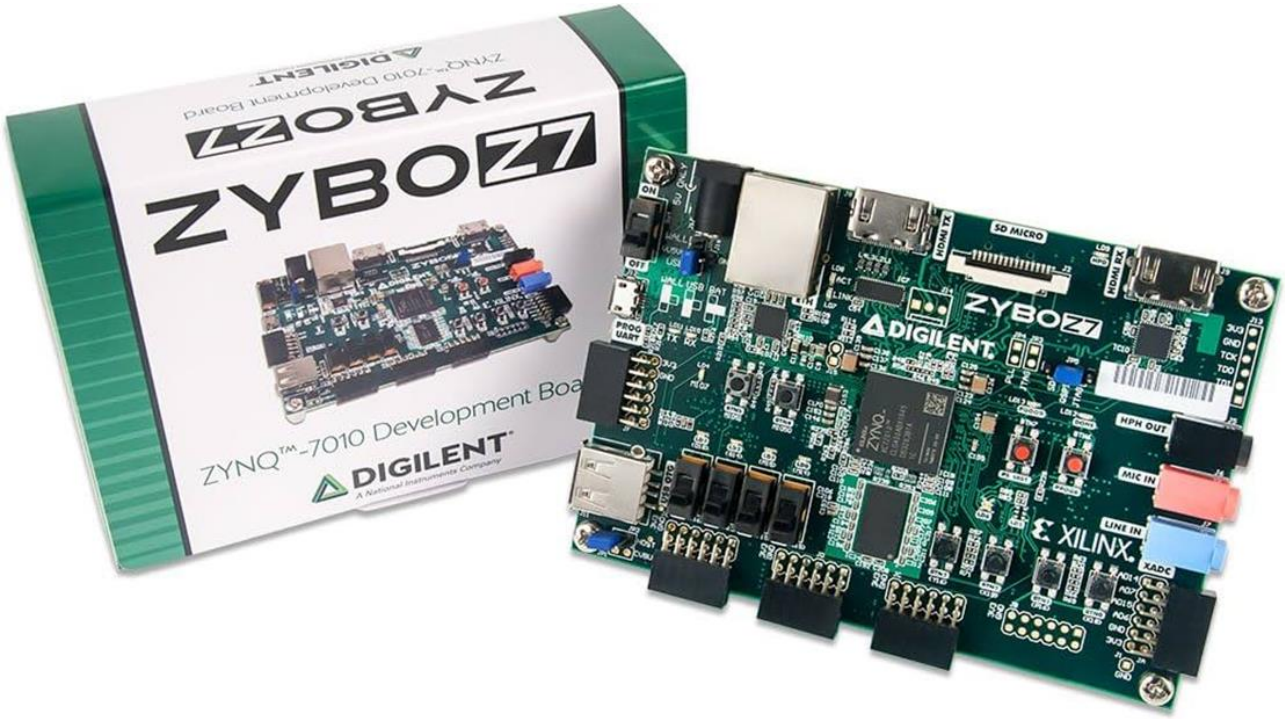
Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών



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

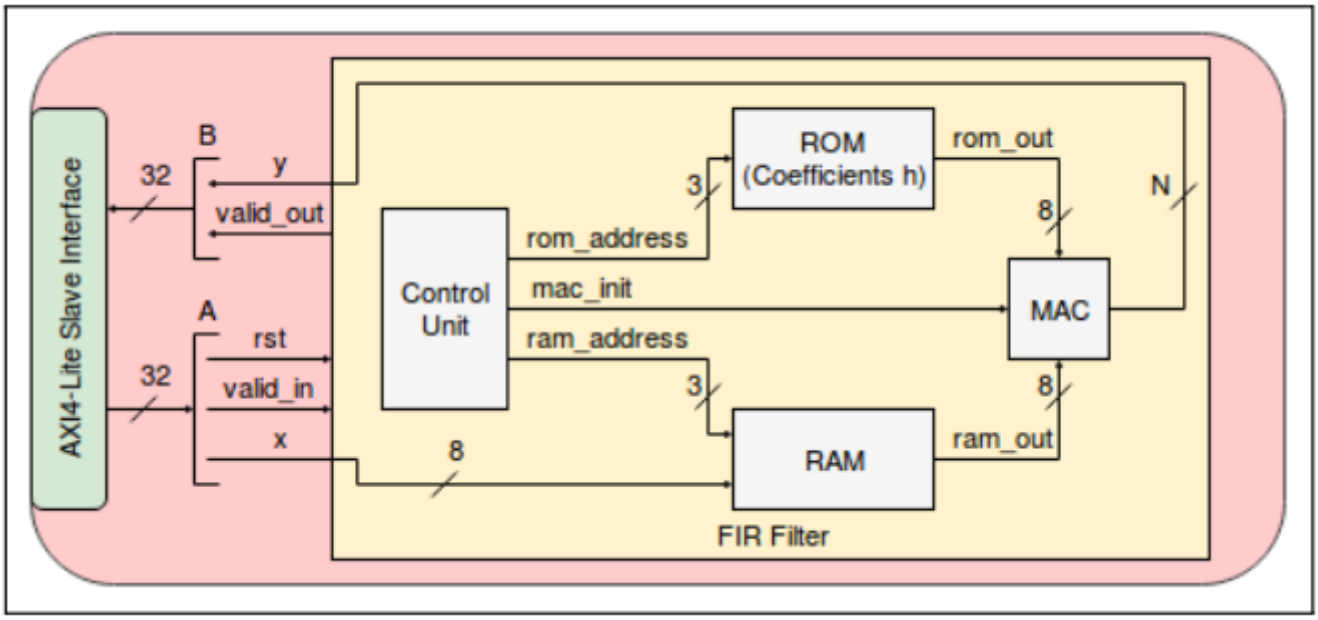
Σκοπός της παρούσας εργαστηριακής άσκησης είναι η υλοποίηση ενός FIR φίλτρου με AXI4-Lite διεπαφή και στη συνέχεια η εισαγωγή του στο Zybo SoC.

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 A 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 B 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 AXI 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 IP 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
```

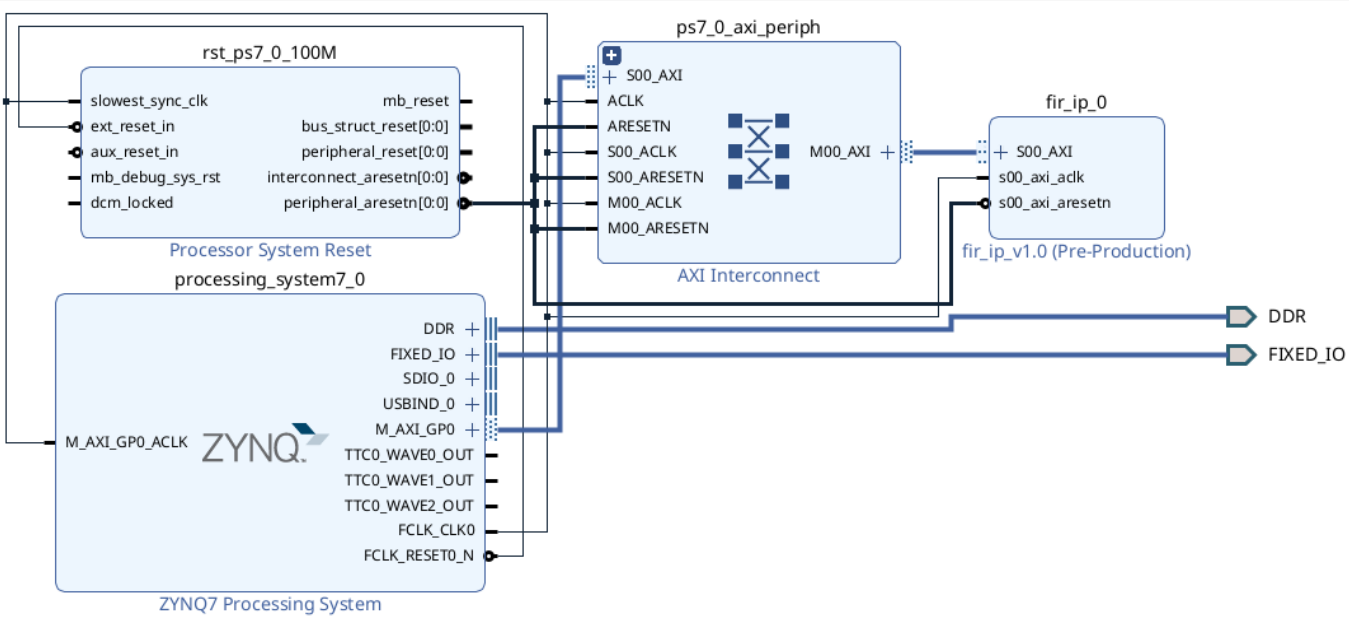
```
	-- de-asserted when reset (active low) is asserted.
	-- The read address is also latched when S_AXI_ARVALID is
	-- asserted. axi_araddr is reset to zero on reset assertion.

	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_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
    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.

Το 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.