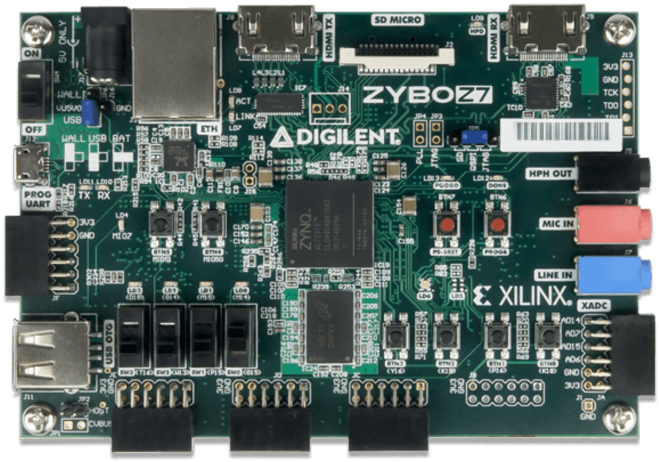
Ψηφιακά Συστήματα VLSI

8ο εξάμηνο, Ακαδημαϊκή περίοδος 2024-2025

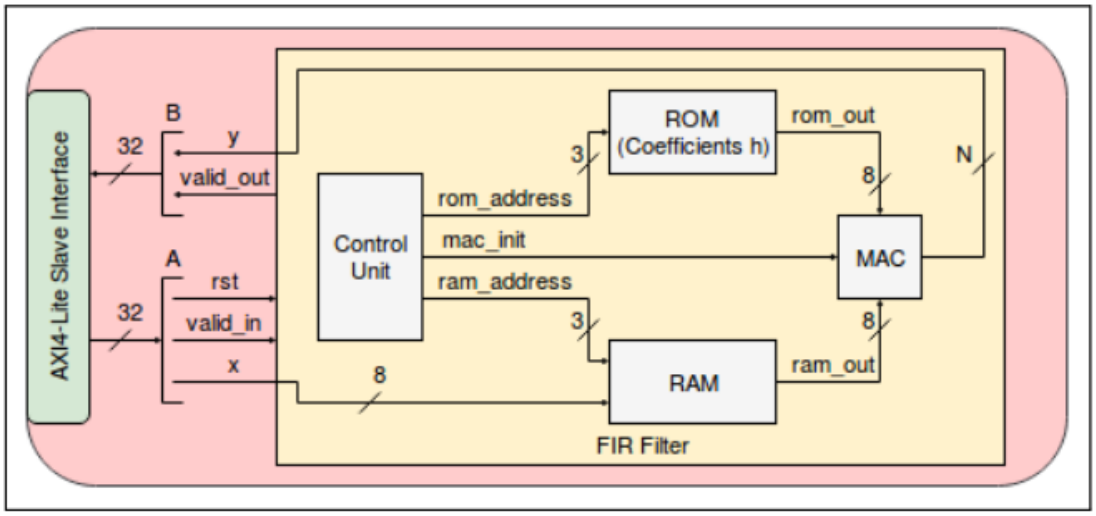
5η Εργαστηριακή Άσκηση

Δημήτρης Καμπανάκης: 03121012
Αγγελική Ζέρβα: 03121101



Εισαγωγή

Σε αυτήν την εργαστηριακή άσκηση τροποποιήσαμε τον κώδικα της προηγούμενης άσκησης, ο οποίος υλοποιεί ένα fir φίλτρο, ώστε να προγραμματίσουμε την αναπτυξιακή πλακέτα ZYBO για την υλοποίηση. Τα δεδομένα στέλνονται από τον ενσωματομένο επεξεργαστή (ARM) στο FPGA για επεξεργασία και τα αποτελέσματα θα ακολουθούν την αντίθετη διαδρομή. Το φίλτρο που θα υλοποιήσουμε με διεπαφή AXI4-Lite θα έχει την παρακάτω μορφή:



AXI4-Lite IP

Αρχικά, στο project δημιουργούμε το block design, στο οποίο προσθέτουμε το ZYNQ processing system και ένα AXI4-Lite IP, στο οποίο θα υλοποιήσουμε το fir filter. Οι κώδικες μας στο IP φαίνονται παρακάτω:

Κώδικας AXI4-Lite interface:

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

entity 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 fir_v1_0_S00_AXI;

architecture arch_imp of fir_v1_0_S00_AXI is


    component fir_filter
        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;


        -- 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


    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 => slv_reg1(19),
        y => slv_reg1(18 downto 0)
    );


    -- 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;

--                    if slv_reg0(8) = '1' then
--                       slv_reg0(8) <= '0';
--                    end if;

--                  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

    -- User logic ends

end arch_imp;
```

Κώδικας control-unit:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity control_unit is
    Port (
           clk            : in STD_LOGIC;
           rst            : in STD_LOGIC;
           valid_in       : in STD_LOGIC;
           rom_address    : out STD_LOGIC_VECTOR (2 downto 0);
           ram_address    : out STD_LOGIC_VECTOR (2 downto 0);
           mac_init       : out STD_LOGIC;
           mac_en         : out STD_LOGIC;
           done           : out STD_LOGIC
        );
end entity;

architecture behavioral of control_unit is
    signal up_counter     : std_logic_vector(2 downto 0) := "000";
    signal en             : std_logic := '0';
    signal valid_in_reg   : std_logic := '0';
    signal done_reg, done_reg_2       : std_logic := '0';
begin
    process (clk, rst)
    begin
        if rst = '1' then
            up_counter    <= "000";
            mac_init      <= '0';
            done_reg      <= '0';
            en            <= '0';
            valid_in_reg  <= '0';

        elsif rising_edge(clk) then
            valid_in_reg <= valid_in;

            -- Start MAC on rising edge of valid_in
            if valid_in = '1' and valid_in_reg = '0' then
                en <= '1';
                up_counter <= "000";
            end if;

            if en = '1' then
                mac_en <= '1';
                if up_counter = "000" then
                    mac_init <= '1';  -- Start accumulation
                else
                    mac_init <= '0';
                end if;

                if up_counter = "111" then
                    en <= '0';        -- Stop after 8 cycles
```

```
                    done_reg <= '1';  -- Pulse done
                else
                    done_reg <= '0';
                end if;

--                if done_reg = '1' then
--                    done_reg_2 <= '1';
--                else
--                    done_reg_2 <= '0';
--                end if;

                up_counter <= up_counter + 1;
            else
                mac_init <= '0';
                done_reg <= '0';
                mac_en <= '0';
            end if;
        done_reg_2 <= done_reg;
        end if;
    end process;
    rom_address <= up_counter;
    ram_address <= up_counter;
    done        <= done_reg_2;
end behavioral;
```

Κώδικας fir filter:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity 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 fir_filter;



architecture structural of fir_filter is
```

```
component control_unit is

    Port (
        clk         : in STD_LOGIC;
        rst         : in STD_LOGIC;
        valid_in    : in STD_LOGIC;
        rom_address : out STD_LOGIC_VECTOR (2 downto 0);
        ram_address : out STD_LOGIC_VECTOR (2 downto 0);
        mac_init    : out STD_LOGIC;
        mac_en      : out STD_LOGIC;
        done        : out STD_LOGIC

    );

end component;



component mlab_rom is

    generic ( coeff_width : integer := 8 );

    Port (

        clk         : in  STD_LOGIC;
        en          : in  STD_LOGIC;
        rom_address : in  STD_LOGIC_VECTOR (2 downto 0);
        rom_out     : out STD_LOGIC_VECTOR (7 downto 0)

    );

end component;



component mlab_ram is

    generic ( data_width : integer := 8 );

    port (
        clk         : in std_logic;
        rst         : in std_logic;
        we          : in std_logic;
        en          : in std_logic;
        ram_address : in std_logic_vector(2 downto 0);
        ram_in      : in std_logic_vector(7 downto 0);
        ram_out     : out std_logic_vector(7 downto 0)
    );

end component;
```

```
component mac is

    Port (
        clk      : in STD_LOGIC;
        mac_en   : in STD_LOGIC;
        mac_init : in STD_LOGIC;
        rom_out  : in STD_LOGIC_VECTOR (7 downto 0);
        ram_out  : in STD_LOGIC_VECTOR (7 downto 0);
        y_out    : out STD_LOGIC_VECTOR (18 downto 0)

    );

end component;



signal rom_addr, ram_addr : std_logic_vector(2 downto 0);
signal rom_out, ram_out   : std_logic_vector(7 downto 0);
signal mac_init, mac_en   : std_logic;
signal done               : std_logic;
signal y_temp             : std_logic_vector(18 downto 0);
signal valid_out_latched  : std_logic := '0';
signal write_enable, valid_in_reg       : std_logic := '0';


begin



    control : control_unit port map(
        clk         => clk,
        rst         => rst,
        valid_in    => write_enable,
        rom_address => rom_addr,
        ram_address => ram_addr,
        mac_init    => mac_init,
        mac_en      => mac_en,
        done        => done
    );


    rom : mlab_rom port map(
        clk         => clk,
        en          => '1',
        rom_address => rom_addr,
        rom_out     => rom_out
    );



    ram : mlab_ram port map(
        clk         => clk,
```

```
        rst         => rst,
        we          => write_enable,
        en          => mac_en,
        ram_address => ram_addr,
        ram_in      => x,
        ram_out     => ram_out
    );



    multac : mac port map(
        clk      => clk,
        mac_en   => mac_en,
        mac_init => mac_init,
        rom_out  => rom_out,
        ram_out  => ram_out,
        y_out    => y_temp
    );



    process(clk, rst)
    begin
        if rst = '1' then
            valid_out <= '0';
            valid_out_latched <= '0';
            y <= (others => '0');
        elsif rising_edge(clk) then
            valid_in_reg <= valid_in;
--            if valid_in = '1' and valid_in_reg = '0' then
--                write_enable <= '1';
--            else
--                write_enable <= '0';
--            end if;

            if done = '1' then
                y <= y_temp;
                valid_out <= '1';
                valid_out_latched <= '1';
--          elsif valid_in_reg = '0' and valid_out_latched = '1' then (previous version)
            elsif write_enable = '1' and valid_out_latched = '1' then
                valid_out <= '0';
                valid_out_latched <= '0';
            end if;
        end if;

        write_enable <= valid_in and (not valid_in_reg);


    end process;



end structural;
```

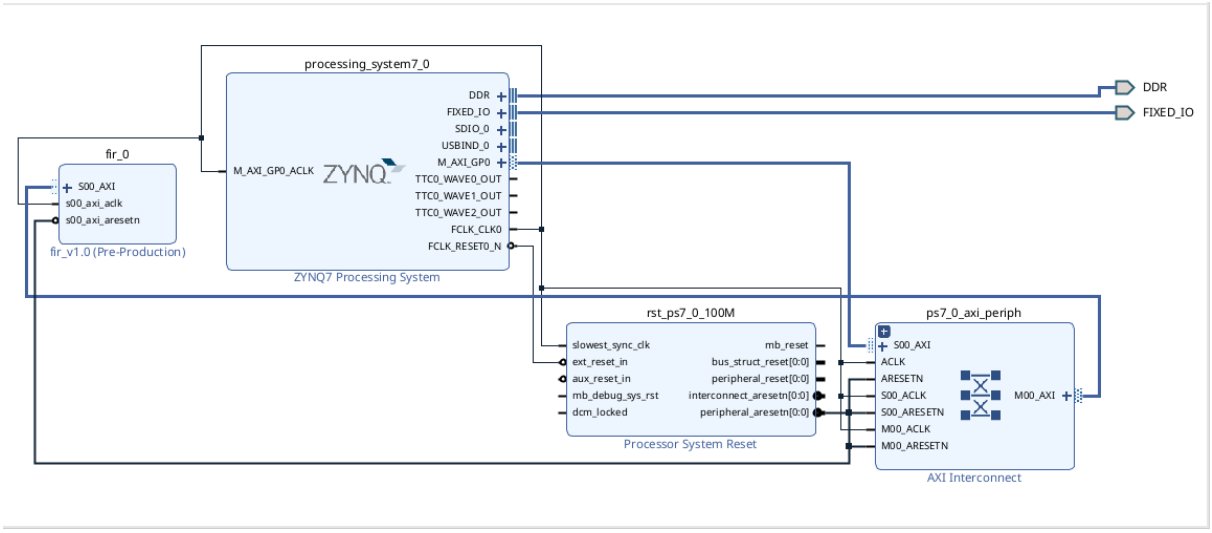
Οι κώδικες των υπόλοιπων κομματιών της υλοποίησης μας δεν έχουν αλλάξει από την υλοποίηση της προηγούμενης εργαστηριακής άσκησης. Στα παραπάνω, στον κώδικα του AXI4-Lite interface προσθέσαμε ως component το δικό μας fir filter και κάναμε το παρακάτω port map και εμποδίσαμε την εγγραφή στον slv_reg1 που αποτελεί την έξοδο του φίλτρου μας:

- A[7:0] = x
- A[8] = valid_in
- A[9] = rst
- A[31:10] = not used
- B[N-1:0] = y
- B[N] = valid_out
- B[31:N+1] = not used

όπου εμείς έχουμε αντιστοιχίσει το A στο slv_reg0 και το B στο slv_rev1. Στα control-unit και fir filter έχουμε αλλάξει λίγο την λογική μας ώστε να μπορεί να διαχειριστεί ένα valid in που διαρκεί περισσότερους κύκλους, όπως θα είναι το σήμα που έρχεται από τον arm πεξεργαστή και τον valid out σήμα να είναι 1 για περισσότερους κύκλους, ώστε α προλαβαίνει ο ενσωματωμένος επεξεργαστής να διαβάσει, καθώς λειτουργεί πιο αργά από το FPGA.

Υλοποίηση συστήματος και παραγωγή bitstream

Αφού έχουμε ολοκληρώσει τον κώδικα για το φίλτρο, κάνουμε package το IP. Πραγματοποιώντας τις διασυνδέσεις του ZYNQ με την διεπαφή AXI4-Lite προκύπτει το παρακάτω block design:

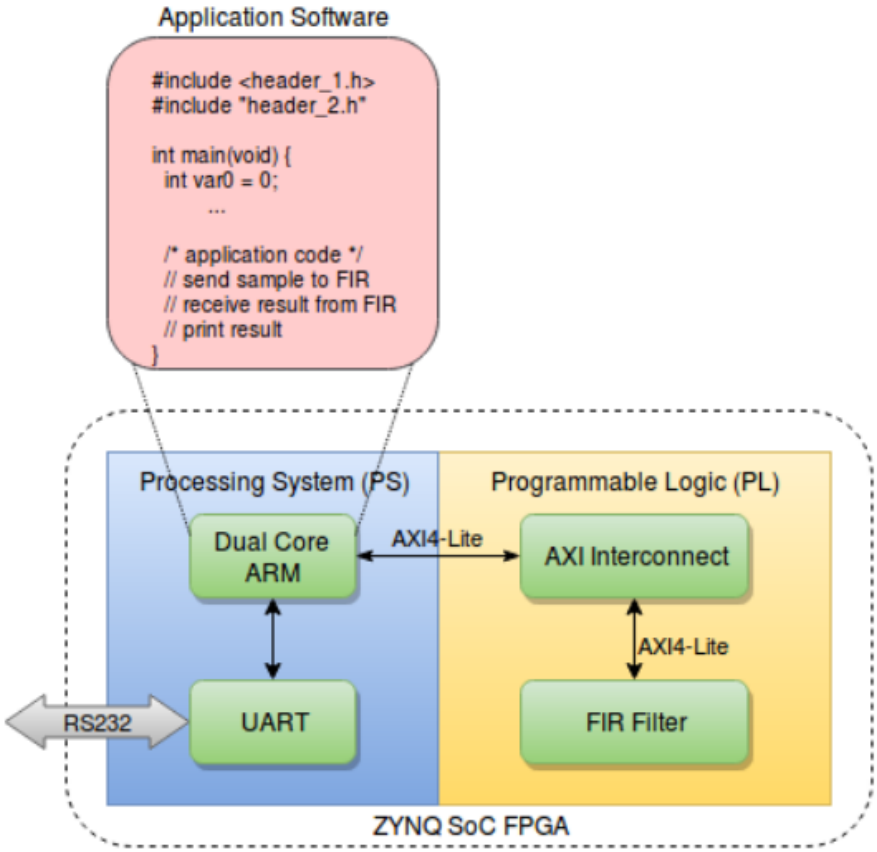


Παράγουμε το bitstream, προσέχοντας το τελικό μας bitstream να μην έχει critical warnings (και κατά προτίμηση ούτε warnings).

Εφόσον έχουμε παράξει το bitstream, μπορούμε να χρησιμοποιήσουμε το Vitis IDE για να προγραμματίσουμε το ZYBO.

Προγραμματισμός του ZYNQ SoC FPGA και εκτέλεση της εφαρμογής

Παρακάτω φαίνεται η τελική αρχιτεκτονική του project μας και ο κώδικας που πραγματοποιεί την επικοινωνία με το FPGA, που αποτελεί την εφαρμογή λογισμικού, η οποία είναι υπεύθυνη να στέλνει τα δεδομένα και να λαμβάνει τα αποτελέσματα και να τα τυπώνει στο τερματικό, αφού ελέγξει την εγκυρότητά τους.




```
    #include <stdio.h>
#include "xil_io.h"
#include "xparameters.h"
#include "sleep.h"

#define FIR_BASEADDR      0x43C00000
#define REG_CONTROL       (FIR_BASEADDR + 0x00) // slv_reg0
#define REG_OUTPUT        (FIR_BASEADDR + 0x04) // slv_reg1

//int data_x[16] = {
//    0xBE, 0x01, 0x58, 0xC7, 0x6D, 0x9A, 0x13, 0xF5,
//    0x2E, 0xD6, 0x4B, 0x89, 0xE1, 0x7C, 0xA2, 0x3F
//};

length = 20;

int data_x2[20] = {
```

```
    0xD0, 0xE7, 0x20, 0xE9, 0xA1, 0x18, 0x47, 0x8C,
    0xF5, 0xF7, 0x28, 0xF8, 0xF5, 0x7C, 0xCC, 0x24,
    0x6B, 0xEA, 0xCA, 0xF5
};

int main() {
    printf("Starting FIR filter test...\n");

    // ---------- RESET FIRST ----------
    printf("Applying reset...\n");
    Xil_Out32(REG_CONTROL, (1 << 9));  // rst = 1 (bit 9)
    usleep(10);
    Xil_Out32(REG_CONTROL, 0);         // rst = 0
    printf("Reset complete.\n");

    // ---------- FIRST TEST LOOP ----------
    for (int i = 0; i < length; i++) {
        printf("1st loop - sending sample %2d\n", i);

        u32 control_word = (1 << 8) | (data_x2[i] & 0xFF);  // valid_in = 1
        Xil_Out32(REG_CONTROL, control_word);
        usleep(10);  // short pulse
        Xil_Out32(REG_CONTROL, data_x2[i] & 0xFF); // valid_in = 0

        while ((Xil_In32(REG_OUTPUT) & (1 << 19)) == 0); // wait for valid_out

        u32 result = Xil_In32(REG_OUTPUT) & 0x7FFFF; // 19-bit output
        printf("1st loop Input[%2d] = %3d → FIR Output = %5d\n", i, data_x2[i], result);

        usleep(1000);
    }

    // ---------- APPLY RESET BEFORE SECOND LOOP ----------
    printf("\nApplying reset again before second test loop...\n");
    Xil_Out32(REG_CONTROL, (1 << 9));  // rst = 1
    usleep(10);
    Xil_Out32(REG_CONTROL, 0);         // rst = 0
    printf("Reset complete.\n");

    // ---------- SECOND TEST LOOP ----------
    for (int i = 0; i < length; i++) {
        printf("2nd loop - sending sample %2d\n", i);

        u32 control_word = (1 << 8) | (data_x2[i] & 0xFF);  // valid_in = 1
        Xil_Out32(REG_CONTROL, control_word);
        usleep(10);  // short pulse
        Xil_Out32(REG_CONTROL, data_x2[i] & 0xFF); // valid_in = 0

        while ((Xil_In32(REG_OUTPUT) & (1 << 19)) == 0); // wait for valid_out

        u32 result = Xil_In32(REG_OUTPUT) & 0x7FFFF; // 19-bit output
        printf("2nd loop Input[%2d] = %3d → FIR Output = %5d\n", i, data_x2[i], result);

        usleep(1000);
```

```
    }

    printf("\nFIR test complete.\n");
    return 0;
}
```

Δοκιμάζουμε τα αποτελέσματα και στην συνέχεια πραγματοποιούμε reset και ξαναπροσπαθούμε. Σημειώνουμε ότι τα δεδομένα εισόδου και εξόδου είναι τα παρακάτω:
x = {208, 231, 32, 233, 161, 24, 71, 140, 245, 247, 40, 248, 245, 124, 204, 36, 107, 234, 202, 245}

y = {208, 647, 1118, 1822, 2687, 3576, 4536, 5636, 5109, 4414, 5319, 4631, 4603, 5771, 6696, 6965, 6256, 5518, 6598, 6011}

Οι παράγοντες του fir filter είναι:
h = {1, 2, 3, 4, 5, 6, 7, 8}
Τα αποτελέσματα φαίνονται παρακάτω, όπου παρατηρούμε την σωστή υλοποίηση του φίλτρου μας:

```
Starting FIR filter test...
Applying reset...
Reset complete.
1st loop - sending sample  0
1st loop Input[ 0] = 208 → FIR Output =   208
1st loop - sending sample  1
1st loop Input[ 1] = 231 → FIR Output =   647
1st loop - sending sample  2
1st loop Input[ 2] =  32 → FIR Output =  1118
1st loop - sending sample  3
1st loop Input[ 3] = 233 → FIR Output =  1822
1st loop - sending sample  4
1st loop Input[ 4] = 161 → FIR Output =  2687
1st loop - sending sample  5
1st loop Input[ 5] =  24 → FIR Output =  3576
1st loop - sending sample  6
1st loop Input[ 6] =  71 → FIR Output =  4536
1st loop - sending sample  7
1st loop Input[ 7] = 140 → FIR Output =  5636
1st loop - sending sample  8
1st loop Input[ 8] = 245 → FIR Output =  5109
1st loop - sending sample  9
1st loop Input[ 9] = 247 → FIR Output =  4414
1st loop - sending sample 10
1st loop Input[10] =  40 → FIR Output =  5319
1st loop - sending sample 11
1st loop Input[11] = 248 → FIR Output =  4631
1st loop - sending sample 12
1st loop Input[12] = 245 → FIR Output =  4603
1st loop - sending sample 13
1st loop Input[13] = 124 → FIR Output =  5771
1st loop - sending sample 14
1st loop Input[14] = 204 → FIR Output =  6696
1st loop - sending sample 15
1st loop Input[15] =  36 → FIR Output =  6965
1st loop - sending sample 16
1st loop Input[16] = 107 → FIR Output =  6256
1st loop - sending sample 17
1st loop Input[17] = 234 → FIR Output =  5518
1st loop - sending sample 18
1st loop Input[18] = 202 → FIR Output =  6598
1st loop - sending sample 19
1st loop Input[19] = 245 → FIR Output =  6011
```

```
Applying reset again before second test loop...
Reset complete.
2nd loop - sending sample  0
2nd loop Input[ 0] = 208 → FIR Output =   208
2nd loop - sending sample  1
2nd loop Input[ 1] = 231 → FIR Output =   647
2nd loop - sending sample  2
2nd loop Input[ 2] =  32 → FIR Output =  1118
2nd loop - sending sample  3
2nd loop Input[ 3] = 233 → FIR Output =  1822
2nd loop - sending sample  4
2nd loop Input[ 4] = 161 → FIR Output =  2687
2nd loop - sending sample  5
2nd loop Input[ 5] =  24 → FIR Output =  3576
2nd loop - sending sample  6
2nd loop Input[ 6] =  71 → FIR Output =  4536
2nd loop - sending sample  7
2nd loop Input[ 7] = 140 → FIR Output =  5636
2nd loop - sending sample  8
2nd loop Input[ 8] = 245 → FIR Output =  5109
2nd loop - sending sample  9
2nd loop Input[ 9] = 247 → FIR Output =  4414
2nd loop - sending sample 10
2nd loop Input[10] =  40 → FIR Output =  5319
2nd loop - sending sample 11
2nd loop Input[11] = 248 → FIR Output =  4631
2nd loop - sending sample 12
2nd loop Input[12] = 245 → FIR Output =  4603
2nd loop - sending sample 13
2nd loop Input[13] = 124 → FIR Output =  5771
2nd loop - sending sample 14
2nd loop Input[14] = 204 → FIR Output =  6696
2nd loop - sending sample 15
2nd loop Input[15] =  36 → FIR Output =  6965
2nd loop - sending sample 16
2nd loop Input[16] = 107 → FIR Output =  6256
2nd loop - sending sample 17
2nd loop Input[17] = 234 → FIR Output =  5518
2nd loop - sending sample 18
2nd loop Input[18] = 202 → FIR Output =  6598
2nd loop - sending sample 19
2nd loop Input[19] = 245 → FIR Output =  6011

FIR test complete.
```

Τελικό Συμπέρασμα για το AXI4-Lite

Η διεπαφή AXI4-Lite μας επιτρέπει την λειτουργία του FPGA και την επικοινωνία του με την cpu του SoC με απλό τρόπο και χωρίς να καταναλώνει πολλούς πόρους στο FPGA. Το πρωτόκολλο αυτό όμως έχει το μειονέκτημα ότι επιτρέπει μεταφορά των δεδομένων ένα την φορά με μια λογική χειραψίας, χωρίς να επιτρέπει bursts δεδομένων. Αυτό μπορεί να αποτελέσει πρόβλημα για εφαρμογές που θα απαιτούσαν μεγαλύτερο throughput.