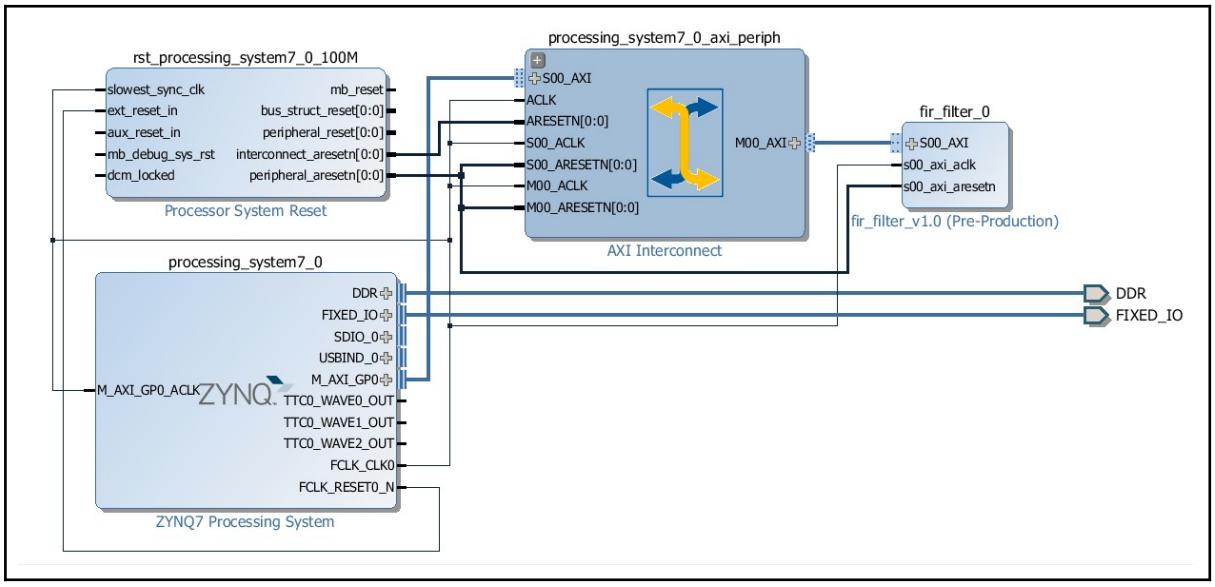
Digital Systems VLSI

Nanos Georgios

Initially, we created the requested Hardware / Software system at ZYBO following the instructions of the exercise and the suggested guides. The resulting Block Design is as follows:



For the IP of FIR FILTER we needed the following files:

fir_filter_v1_0_S00_AXI.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity fir_filter_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_filter_v1_0_S00_AXI;

architecture arch_imp of fir_filter_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 fir_out :std_logic_vector(31 downto 0);
   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);
      y: out std_logic_vector(15 downto 0);
      valid_out: out std_logic);
      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';
            else
              if (axi_awready = '0' and S_AXI_AWVALID = '1' and S_AXI_WVALID = '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';
              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') 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') 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_aready <= '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, fir_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
        if S_AXI_ARESETN = '0' then
          reg_data_out  <= (others => '1');
        else
          -- 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_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 if;
        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
   fir_filter0 : fir_filter
   port map (
      clk => S_AXI_ACLK,
      x => slv_reg0(7 downto 0),
      valid_in => slv_reg0(8),
      rst => '0', --slv_reg0(9),
      y => fir_out(15 downto 0),
      valid_out => fir_out(24));
        -- User logic ends

end arch_imp;
```

The Fir Filter and its components:

- **fir_filter.vhd**

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

```
entity fir_filter is
    generic (
        data_width : integer := 8;      -- width of data (bits)
        coeff_width : integer := 8      -- width of coefficients (bits)
    );
    port (
        clk: in std_logic;
        rst: in std_logic;
        valid_in: in std_logic;
        x: in std_logic_vector(7 downto 0);
        y: out std_logic_vector(15 downto 0);
        valid_out: out std_logic);
end fir_filter;

architecture Structural of fir_filter is


    component control_unit is
        port (
            clk: in std_logic;
            rst: in std_logic;
            pause: out std_logic;
            valid_in: in std_logic;
            rom_addr: out std_logic_vector(2 downto 0);
            ram_addr: out std_logic_vector(2 downto 0);
            mac_init: out std_logic;
            valid_out: out std_logic
        );
    end component;

    component ram is
        port (
            clk: in std_logic;
            rst: in std_logic;
            we: in std_logic;
            en: in std_logic;
            ram_addr: in std_logic_vector(2 downto 0);
            data_input: in std_logic_vector(data_width-1 downto 0);
            ram_out: out std_logic_vector(data_width-1 downto 0)
        );
    end component;

    component rom is
        port (
            clk: in  std_logic;
            en: in  std_logic;      -- operation enable
            rom_addr: in  std_logic_vector(2 downto 0);     -- memory address
            rom_out: out  std_logic_vector(coeff_width-1 downto 0)  -- output data
        );
    end component;

    component mac is
        port (
            clk: in std_logic;
```

```
        pause: in std_logic;
        rom_out: in std_logic_vector (coeff_width-1 downto 0);
        mac_init: in std_logic;
        ram_out: in std_logic_vector (data_width-1 downto 0);
        y: out std_logic_vector (15 downto 0)
    );
  end component;

  signal ram_addr: std_logic_vector(2 downto 0);
  signal rom_addr: std_logic_vector(2 downto 0);
  signal mac_init: std_logic;
  signal rom_out: std_logic_vector(7 downto 0);
  signal ram_out: std_logic_vector(7 downto 0);
  signal pause : std_logic;
begin
  control_unit_p: control_unit
  port map (
    clk => clk,
    rst => rst,
    pause => pause,
    valid_in => valid_in,
    ram_addr => ram_addr,
    rom_addr => rom_addr,
    mac_init => mac_init,
    valid_out => valid_out
  );

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

  ram_p : ram
  port map (
    clk => clk,
    rst => rst,
    we => mac_init,
    en => '1',
    data_input => x,
    ram_addr => ram_addr,
    ram_out => ram_out
  );

  mac_p: mac
  port map (
    clk => clk,
    pause => pause,
    mac_init => mac_init,
    rom_out => rom_out,
    ram_out => ram_out,
    y => y
```

```
    );

end Structural;
```

- **control_unit.vhd**

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

entity control_unit is
    port (
        clk: in std_logic;
        rst: in std_logic := '0';
        valid_in: in std_logic;
        pause: out std_logic;
        rom_addr: out std_logic_vector(2 downto 0);
        ram_addr: out std_logic_vector(2 downto 0);
        mac_init: out std_logic ;
        valid_out: out std_logic
    );
end control_unit;

architecture Behavioural of control_unit is
    signal counter: std_logic_vector(2 downto 0) := "000";
--    signal prev_counter: std_logic_vector(2 downto 0) := (others=>'0');
    signal valid_in_flag: std_logic := '0';

begin

--    ovehflo: process (counter)
--    begin
--        if prev_counter = "000" and counter = "001" then
--                valid_out <= '1';
--        else
--                valid_out <= '0';
--        end if;
--    end process;

    edge: process(clk,rst, valid_in)
    begin
        if valid_in'event and valid_in='1' then
            valid_in_flag <= '1';
        end if;

        if valid_in'event and valid_in='0' then
            valid_in_flag <= '0';
```

```
    end if;

    if rst='1' then
       counter <= (others=>'0');
    elsif clk'event and clk='1' then
       if counter = "000" and valid_in_flag = '1' then
          mac_init <= '1';
          valid_out <= '1';
          valid_in_flag <= '0';
          counter <= counter+1;
          pause <= '0';
       elsif counter = "000" and valid_in_flag = '0' then
          pause <= '1';
          valid_out <= '1';
       else
          pause <= '0';
          mac_init <= '0';
          valid_out <= '0';
          counter <= counter+1;
    end if;
    end if;
  end process;

  ram_addr <= counter;
  rom_addr <= counter;


end Behavioural;
```

- **rom.vhd**

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

entity rom is
  generic (
    coeff_width: integer := 8       -- width of coefficients (bits)
  );
  port (
    clk: in  std_logic;             -- clock
    en: in  std_logic;              -- operation enable
    rom_addr: in  std_logic_vector(2 downto 0);             -- memory address
    rom_out: out  std_logic_vector(coeff_width-1 downto 0)
    );
end rom;

architecture Behavioural of rom is

  type rom_type is array(7 downto 0) of std_logic_vector(coeff_width-1 downto 0);
```

```
--    signal rom: rom_type:= ("00001000", "00000111", "00000110", "00000101", "00000100",
"00000011", "00000010",
--                              "00000001");   -- initialization of rom with user data
       signal rom: rom_type:= ("00000001", "00000001", "00000001", "00000001", "00000001",
"00000001", "00000001",
                          "00000001");
   signal rdata: std_logic_vector(coeff_width-1 downto 0);

begin

   rdata <= rom(conv_integer(rom_addr));

   process (clk)
   begin
      if (clk'event and clk='1') then
         if (en='1') then
            rom_out <= rdata;
         end if;
      end if;
   end process;

end Behavioural;
```

- **ram.vhd**

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

entity ram is
   generic (
      data_width: integer :=8     -- width of data (bits)
   );
   port (
      clk: in std_logic;
      rst: in std_logic;
      we: in std_logic;          -- memory write enable
      en: in std_logic;          -- operation enable
      ram_addr: in std_logic_vector(2 downto 0);              -- memory address
      data_input: in std_logic_vector(data_width-1 downto 0);      -- input data
      ram_out: out std_logic_vector(data_width-1 downto 0) := (others=>'0')
      );   -- output data
end ram;

architecture Behavioural of ram is

   type ram_type is array(7 downto 0) of std_logic_vector(data_width-1 downto 0);
   signal ram: ram_type := (others=>(others=>'0'));
```

```
begin

   process (clk, rst)
   begin
      if rst='1' then
         ram <= (others=>(others=>'0'));
      elsif clk'event and clk='1' then
         if en='1' then
            if we='1' then                              -- write operation
               ram(7 downto 1) <= ram(6 downto 0);
               ram(0) <= data_input;
               ram_out <= data_input;
            else                                        -- read operation
               ram_out <= ram(conv_integer(ram_addr));
            end if;
         end if;
      end if;
   end process;

end Behavioural;
```

- **mac.vhd**

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

entity mac is
          generic (
      coeff_width: integer := 8;      -- width of coefficients (bits)
      data_width: integer := 8        -- width of data (bits)
   );
   port (
      clk: in std_logic;
      pause: in std_logic;
      mac_init: in std_logic;
      rom_out: in std_logic_vector(coeff_width-1 downto 0);
      ram_out: in std_logic_vector(data_width-1 downto 0);
      y: out std_logic_vector(15 downto 0)
   );
end mac;

architecture Behavioural of mac is

   signal y_buf: std_logic_vector(15 downto 0) := (others => '0');

begin

   apply_filter: process(clk, pause)
```

```
   begin
      if pause = '0' then
         if clk'event and clk='1' then
            if mac_init='1' then
               y_buf <= rom_out*ram_out;
            else
               -- mporei kai na mhn douleuei
               y_buf <= y_buf+rom_out*ram_out;
            end if;
         end if;
      end if;

   end process;

   y <= y_buf;



end Behavioural;
```

Finally, helloworld.c is the executable file in C that we used in the SDK:

helloworld.c

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xbasic_types.h"
#include "xil_types.h"
#include "xil_exception.h"
#include "xscugic.h"
#include <stdlib.h>
#include <string.h>
#include "fir_filter.h"

u32 *baseaddr_p = (u32 *)XPAR_FIR_FILTER_0_S00_AXI_BASEADDR;

int main()
{
           init_platform();

xil_printf("Welcome to our filter!\n\r");

while(1){
```

```
	int er=0;
	scanf("%d\n\r",&er);
	u32 x;
	if(er>=512){
		Xil_Out32((XPAR_FIR_FILTER_0_S00_AXI_BASEADDR+0x0), 512);
		//Xil_Out32((XPAR_FIR_FILTER_0_S00_AXI_BASEADDR+0x0), 256);
	}
	else{
		x = er+256;
	}

	Xil_Out32((XPAR_FIR_FILTER_0_S00_AXI_BASEADDR+0x0), x);

	usleep(100);

	u32 y,valid;
	y = Xil_In32(XPAR_FIR_FILTER_0_S00_AXI_BASEADDR+0x00000008);
	valid = y & 0x00100000;
	if (valid == 0x00100000 ){
		xil_printf("\nOutput is: %d \n\r", y & 0x000FFFFF);  }
	else{
		xil_printf("No new input\n\r");

	}
	Xil_Out32((XPAR_FIR_FILTER_0_S00_AXI_BASEADDR+0x0), 0);
}


cleanup_platform();
return 0;
}
```