Physics of Data, UNIPD
Management and Analysis of Physics Datasets
- VHDL Final Project -

Agnolon Marco, Frazzetto Paolo, Perin Andrea, Polato Carlo, Tabarelli Tommaso

April, 2019

1 FIR Filter

In this homework we developed a 5 tap digital low-pass filter. The corresponding coefficients are: $C_0 = C_4 = 0.19335315$, $C_1 = C_3 = 0.20330353$ and $C_2 = 0.20668665$.

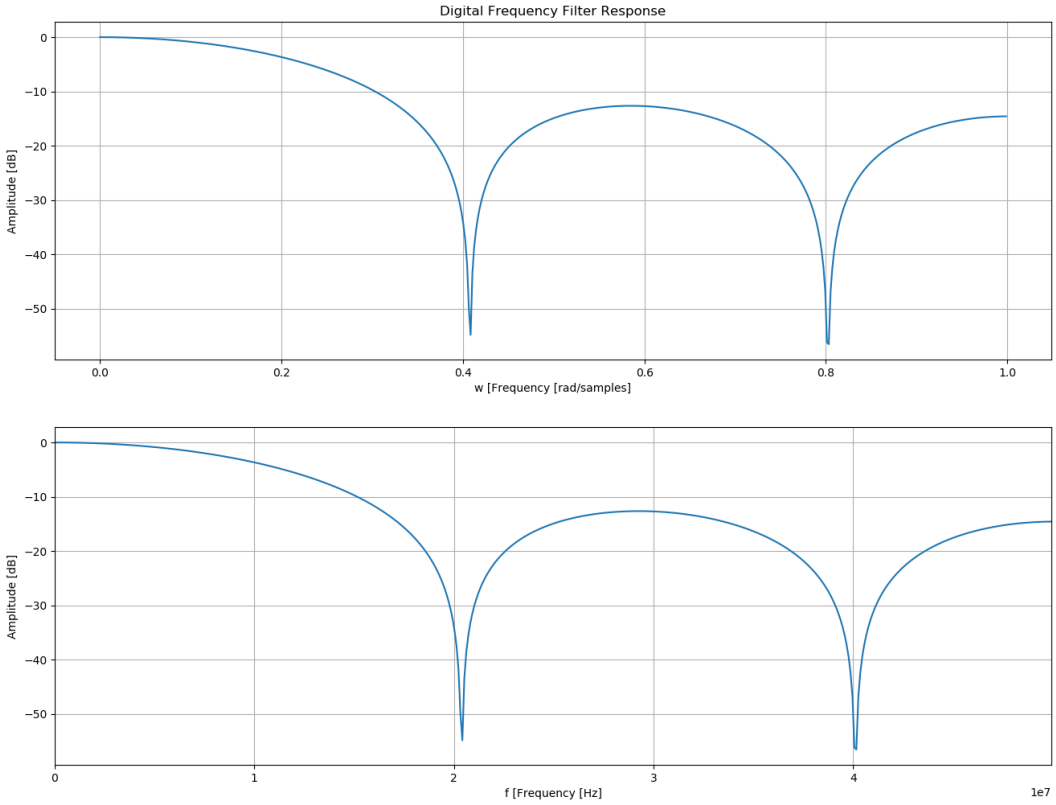

Figure 1: Expected frequency response of the FIR filter.

1.1 Our Results

We tested our FPGA implementation on square waves with different periods and duty cycles. The final results perfectly agree with theory.

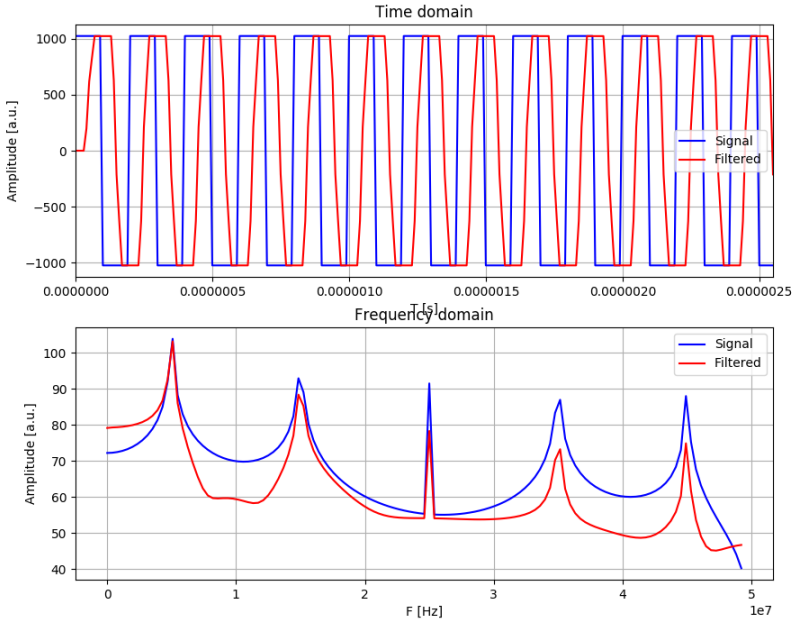

Figure 2: Period = 20, Duty Cycle= 50

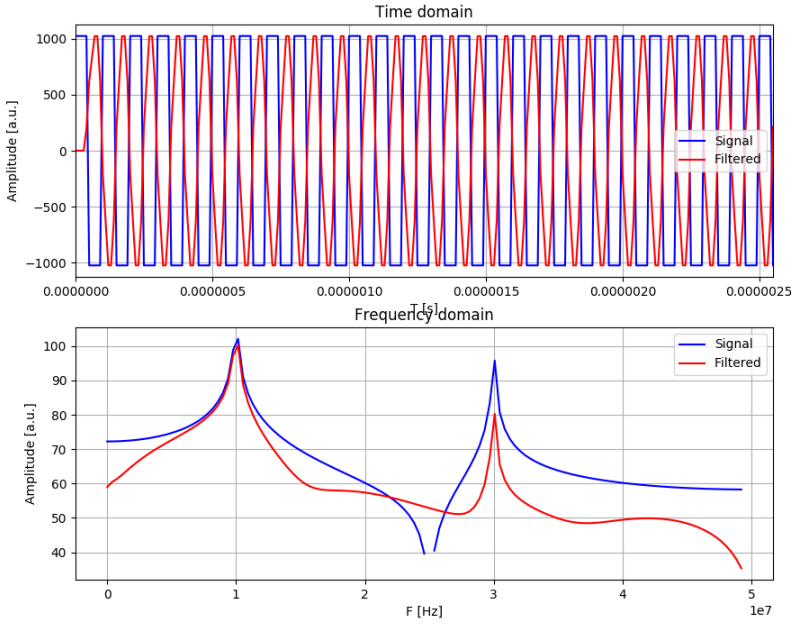

Figure 3: Period = 10, Duty Cycle= 50

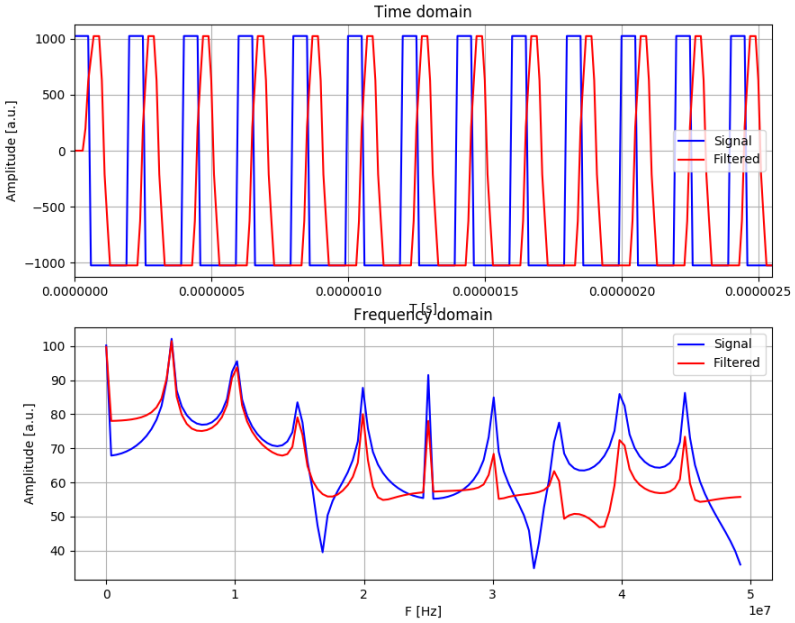

Figure 4: Period = 20, Duty Cycle= 30

2 VIO Policy

All the design blocks are driven with a VIO core. To achieve this, we realized a dedicated VIO FSM for two unique *start* and *reset* signals, such that after the SPI flash memory reading starts with the square wave generation it starts immediately with the fir processing.

```
------------------------------------------------------------------
-----------   FSM FOR VIO POLICY   ---------------------------
------------------------------------------------------------------

p_vio_start : process(clk_base_xc7a_i, vio_start, vio_rst)  -- deleting clock: clk_base_xc7a_i
begin
   if vio_rst = '1' then
      FSM_vio <= s_idle;
      start_spi_s <= '0';
      start_sq_s <= '0';
      start_fir_s <= '0';
      -- vio_start_p <= '0';
      spi_ready_vio_p <= '0';
      we_sq_p <= '0';

  elsif rising_edge(clk_base_xc7a_i) then

      vio_start_p <= vio_start;
      spi_ready_vio_p <= spi_ready_vio;
      we_sq_p <= we_sq;

      case FSM_vio is

      when s_idle =>
          if vio_start = '1' then
              --FSM_vio <= s_read_flash;        -- using previous code ignoring FLASH_READER
              FSM_vio <= s_sq_wave;
          else
              FSM_vio <= s_idle;
          end if:
          start_spi_s <= '0';
          start_sq_s <= '0';
          start_fir_s <= '0';

      -- USING PREVIOUS CODE IGNORING THE FLASH READER
      --when s_read_flash =>
      --    if (spi_ready_vio_p = '0' and spi_ready_vio = '1') then
      --        start_spi_s <= '0';
      --        FSM_vio <= s_sq_wave;
      --    else
      --        FSM_vio <= s_read_flash;
      --        start_spi_s <= '1';
      --    end if;


      when s_sq_wave =>
          if (we_sq_p = '1' and we_sq = '0') then      -- Using falling edge of we_sq
              FSM_vio <= s_fir;
              start_sq_s <= '0';
          else
              FSM_vio <= s_sq_wave;
              start_sq_s <= '1';
          end if;

      when s_fir =>
          start_fir_s <= '1';

      when others =>
          FSM_vio <= s_idle;

      end case;
  end if;


  ------------------------------------------------------------------
  ------------    VIO    -------------------------------------------
  ------------------------------------------------------------------

  -- adding
  my_vio : vio_0
    PORT MAP (
      clk => clk_base_xc7a_i,
      probe_out0(0) => vio_rst,
      probe_out1(0) => vio_start
    );
```

3 Requested Code Fragments

3.1 SPI Master Flash

```
process(clock, reset) -- clock frequency 25MHz
    variable bcnt : integer;
    variable cnt  : integer;
    variable cnt_o : integer;
    variable start_p : std_logic;

    -- adding
    variable w_pos : integer;

  begin
    if rising_edge(clock) then        -- maybe there should be a slow clock
      if reset = '1' then
        bcnt    := N_BYTES - 1;     -- bcnt goes through the 4 bit of the memory (reading process "jumps" every 8 bits
        cnt     := 0;
        state   <= s_idle;
        s_start <= '0';
        ready_s <= '0';
        cnt_o := 0;
        s_word  <= (others => '0');
        we_out <= '0';

      else
        case state is

        when s_idle =>
          s_start <= '0';
          ready_s <= '0';
          -- state   <= s_getbyte; -- this is unnecessary
          cnt := cnt + 1;
          -- the following condition statement stands for "less or equal"
          if cnt <= WTIME then
            state <= s_idle;
          else
            state <= s_getbyte;
          end if;

        when s_getbyte =>       -- reading
            -- this ready_s is the "ready" signal of THIS fsm:
            -- it is created for convention; the user can also not use it (in the top_level
            --  it is "open" (ignored) ).
            ready_s <= '0';

            -- s_ready_fsm is the spi_master ready (that flashes once fsm
            --  has finished writing): so if it can write, then it goes and write before
            --  reading other stuff
            if s_ready_fsm = '1' then
              s_start <= '0';
              state <= s_buildword;

            else
              -- s_start is a signal that flags the starting signal (rising edge of start) for the spi_master:
              -- when it becomes 1 (it is set to 0 both in s_idle and when finished to write), the spi_master
              --  starts again to read (IT IS BUILT IN THIS WAY TO ALLOW MORE READING ITERATIONS)
              s_start <= '1';

              -- N.B.: s_txd is the instructions to give to the spi_master: then spi_master send this instruction
              --       and receive what is in the memory at that address; for this reason, IN THIS PIECE OF CODE we can simply
              --       update the s_txd summing bcnt and not "accessing" the respective memory (this is done by spi_master)

              -- saving txd input in s_txd (of course it will appear changed at the next clock iteration)
              s_txd(TXBITS-1 downto TXBITS-1 - 7) <= txd(TXBITS-1 downto TXBITS-1 - 7);

              -- inline trick to increase the address adding bcnt
              s_txd(TXBITS-1 -8 downto 0) <= std_logic_vector(to_unsigned(to_integer(unsigned(txd(TXBITS-1 -8 downto 0)) + bcnt), TXBITS -8 ));

              -- N.B.: at this point the reading process has taken place (if it took place more than once, it always picked up the same addres
              --       because bcnt does not change in this section!!!

              -- s_txd is now a temporary signal read from outside: it is ready to be written somewhere
            end if;

        when s_buildword =>      -- writing
            if cnt_o < 3 then
              -- adding
              if bcnt >= 0 then      -- checking if the position is invalid
                s_word((bcnt+1)*RXBITS -1 downto (bcnt)*RXBITS) <= s_rxd;     -- geting "read" signal
              end if;

              we_out <= '1';    -- write enabled (keep writing, but actually we overwrite it every time (because the addr is 0)...)
              cnt_o := cnt_o + 1;
            else
              if bcnt = 0 then
                bcnt  := N_BYTES - 1;
                -- adding
                word <= s_word;  -- passing the concatenation to the output port

                state <= s_stop;
                ready_s <= '1';   -- stop all process (the bit to read are finished)
              else
                bcnt  := bcnt - 1;
                state <= s_getbyte;
                ready_s <= '0';   -- ready goes to 0: the FSM now goes to read again
              end if;
              we_out <= '0';       -- write disabled
              cnt_o := 0;          -- resetting writing counter
            end if;

        when s_stop =>
             ready_s <= '1';
             s_start <= '0';
             state <= s_stop;

        when others =>
            state <= s_idle;
        end case;

        start_p := start;
      end if;
    end if;
  end process;
```

3.2 MUX Implementation

```
-------------------------------------------------------------------
------------    MUX_21     ----------------------------------------
-------------------------------------------------------------------

top_mux : mux21
    port map (
            a_in => addr_sq,
            b_in => addr_fir_read,
            sel_in => sel_mux,
            y_out => addr_mux21
            );

    --- in file mux21.vhd
    entity mux21 is
      Port (a_in   : in  std_logic_vector(9 downto 0);
            b_in   : in  std_logic_vector(9 downto 0);
            sel_in : in  std_logic;
            y_out  : out std_logic_vector(9 downto 0)
            );
    end mux21;

    architecture rtl of mux21 is

    begin

    process (a_in, b_in, sel_in) is
    begin
       if sel_in = '0' then
          y_out <= a_in;
       else
          y_out <= b_in;
       end if;
    end process;

    end rtl;
```

3.3 FSM FIR

```
signal start_fir_p : std_logic;

type state_fir is (s_idle, s_read, s_processing, s_write);
signal fir_sq : state_fir;

begin

cf(0) <= to_signed(1584, N_coef); -- 2^13*0.19335315
cf(1) <= to_signed(1665, N_coef); -- 2^13*0.20330353
cf(2) <= to_signed(1693, N_coef); -- 2^13*0.20668665
cf(3) <= to_signed(1665, N_coef); -- 2^13*0.20330353
cf(4) <= to_signed(1584, N_coef); -- 2^13*0.19335315


p_fir : process(clk, rst, start_fir) is
variable cnt : integer := 0;    -- counter

begin
   if rst = '1' then
      fir_sq <= s_idle;
      temp_out <= (others => '0');
      cnt := 0;
      sel_mux_fir <= '0';
      -- start_fir_p <='0';

   elsif rising_edge(clk) then
      start_fir_p <= start_fir;

      case fir_sq is

            when s_idle =>
               if start_fir = '1' and start_fir_p = '0' then  -- if start_fir has a rising edge then start to generate the signal
                                                              -- Care: the signal starts to 0 with WTIME, then there is the 1st pulse
                  fir_sq <= s_read;
                  sel_mux_fir <= '1';
                  we_fir <= '0';
               else
                  fir_sq <= s_idle;
                  sel_mux_fir <= '0';
               end if;
               cnt:= 0;


           when s_read =>                     -- reading from dpram
              if cnt < SAMPLE_FIR then
                 -- to MUX_21
                 addr_read(9 downto 0) <= std_logic_vector(to_unsigned(cnt, 10));
                 sel_mux_fir <= '1';
                 we_fir <= '0';
                 -- here q_sq is collected from the dp-ram
                 --cnt := cnt + 1;        -- can not imcrement the counter here otherwise the write address will be wrong.

                 fir_sq <= s_processing;
              else
                 fir_sq <= s_idle;
              end if;


          when s_processing =>

             xd(0) <= q_sq;

             mult_result(0) <= cf(0) * signed(xd(0));
             mult_result(1) <= cf(1) * signed(xd(1));
             mult_result(2) <= cf(2) * signed(xd(2));
             mult_result(3) <= cf(3) * signed(xd(3));
             mult_result(4) <= cf(4) * signed(xd(4));

             sum_result(0) <= mult_result(0);
             sum_result(1) <= sum_result(0)  + mult_result(1);
             sum_result(2) <= sum_result(1)  + mult_result(2);
             sum_result(3) <= sum_result(2)  + mult_result(3);
             sum_result(4) <= sum_result(3)  + mult_result(4);

             temp_out <= std_logic_vector(resize(shift_right(sum_result(4), N_coef),N));

             fir_sq <= s_write;


          when s_write =>
              we_fir <= '1';
              addr_write(9 downto 0) <= std_logic_vector(to_unsigned(cnt, 10));
              cnt := cnt +1;

              fir_sq <= s_read;

           when others =>       -- there are no other possibilities: if there are, they are unknown and should lead to idle state
              fir_sq <= s_idle;
       end case;
    end if;
  end process;
```

3.4 Square Wave - `s_low`

```
when s_low =>                   -- the pulse stays to 1 untill PULSE_WIDTH has elapsed
   if cnt < SAMPLE_N then
       if h_cnt < PERIOD then
          h_cnt := h_cnt + 1;
          square <= s_low;
          -- TO WRITE
          we_out <= '1';
          address_out(9 downto 0) <= std_logic_vector(to_unsigned(cnt, 10));
          cnt := cnt + 1;
       else
          h_cnt := 0;  -- restarting from s_high
          square <= s_high;
       end if;
   else
       square <= s_idle;
       we_out <= '0';
       -- cnt := 0;
   end if;

   -- LOW LOGIC LEVEL
   y <= std_logic_vector(to_signed(-1024, y'length));
```