Digital Systems VLSI

Nanos Georgios

1) MAC Implementation: Each time the clock strikes a buffer, it sums the product of the rom (shock repulsion) with ram (input signal) and resets the buffer when mac_init is 1.

```
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;
     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) := (others=>'0')
   );
end mac;

architecture Behavioural of mac is

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

```
  apply_filter: process(clk)
  begin
    if clk'event and clk='1' then
      if mac_init='1' then
        y_buf <= rom_out*ram_out;
      else
        y_buf <= y_buf+rom_out*ram_out;
      end if;
    end if;
  end process;

  y <= y_buf;

end Behavioural;
```

ROM Implementation: In each clock cycle it extracts the factor corresponding to the value of the rom_addr vector.

```
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) := (others=>'0') -- output data
    );
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 rdata: std_logic_vector(coeff_width-1 downto 0) := (others=>'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 implementation: Outputs the input value corresponding to the rom_addr signal in each clock cycle. Whenever we equals ‘1’ it makes a shift to the stored values and discards the older value.

```
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;
```

Controller Implementation: The operation and synchronization of the above units is based on a counter 0-7 that shows which position the memories should enter, and what value mac_init should take. A prev_counter signal is used to detect overflows (used to make the system work properly in case the user enters less than 8 cycles)

```
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;
      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 := '0';
      valid_out: out std_logic
   );
end control_unit;

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

begin

   overflow: 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)
   begin
      if rst='1' then
         counter <= (others=>'0');
```

```
    elsif clk'event and clk='1' then
      if valid_in='1' then
        prev_counter <= counter;
        counter <= "001";
      else
        prev_counter <= counter;
        counter <= counter+1;
      end if;

      case counter is
      when "000" =>
        mac_init <= '1';
      when others =>
        mac_init <= '0';
      end case;
    end if;
  end process;

  ram_addr <= counter;
  rom_addr <= counter;

end Behavioural;
```

Implementation of L: To calculate the number of digits of the output, a simple logic unit is used where it checks which is the first digit of the output that has the value ‘1’. Because the operation of the unit is similar to the encoder, it was named PseudoCoder.

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

entity PseudoCoder is
    port (
        input_digits: in std_logic_vector (15 downto 0);
        number_of_digits: out std_logic_vector(4 downto 0)
        );
```

```
end PseudoCoder;

architecture Dataflow of PseudoCoder is

begin

   find_L: process(input_digits)
   begin
      if input_digits(15)='1' then
         number_of_digits <= "10000";
      elsif input_digits(14)='1' then
         number_of_digits <= "01111";
      elsif input_digits(13)='1' then
         number_of_digits <= "01110";
      elsif input_digits(12)='1' then
         number_of_digits <= "01101";
      elsif input_digits(11)='1' then
         number_of_digits <= "01100";
      elsif input_digits(10)='1' then
         number_of_digits <= "01011";
      elsif input_digits(9)='1' then
         number_of_digits <= "01010";
      elsif input_digits(8)='1' then
         number_of_digits <= "01001";
      elsif input_digits(7)='1' then
         number_of_digits <= "01000";
      elsif input_digits(6)='1' then
         number_of_digits <= "00111";
      elsif input_digits(5)='1' then
         number_of_digits <= "00110";
      elsif input_digits(4)='1' then
         number_of_digits <= "00101";
      elsif input_digits(3)='1' then
         number_of_digits <= "00100";
      elsif input_digits(2)='1' then
         number_of_digits <= "00011";
      elsif input_digits(1)='1' then
         number_of_digits <= "00010";
      elsif input_digits(0)='1' then
         number_of_digits <= "00001";
      end if;
   end process;
end Dataflow;
```

Implementation of FIR: Combination of the above. The we of RAM corresponds to the valid_in given by the user. Obviously, the interval of 8 cycles required for the filter to work properly depends on the user. But if the user enters faster than 8 cycles, the system will save the input but will not output valid_out, since the output value will be incorrect.

```
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;
        L: out std_logic_vector(4 downto 0)
    );
end fir_filter;

architecture Structural of fir_filter is

    component PseudoCoder is
        port (
        input_digits: in std_logic_vector (15 downto 0);
        number_of_digits: out std_logic_vector(4 downto 0)
        );
    end component;



    component control_unit is
        port (
            clk: in std_logic;
            rst: in 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;
      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) := (others => '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 output_buffer: std_logic_vector(15 downto 0);

begin

    PseudoCoder_p: PseudoCoder
    port map (
        input_digits => output_buffer,
        number_of_digits => L
        );

    control_unit_p: control_unit
    port map (
      clk => clk,
      rst => rst,
      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 => valid_in,
    en => '1',
    data_input => x,
    ram_addr => ram_addr,
    ram_out => ram_out
  );

  mac_p: mac
  port map (
    clk => clk,
    mac_init => mac_init,
    rom_out => rom_out,
    ram_out => ram_out,
    y => y
  );

  y <= output_buffer;

end Structural;
```

2) The testbench used in the test together with an example of proper operation in case of entry with an interval of less than 8 cycles is the following:

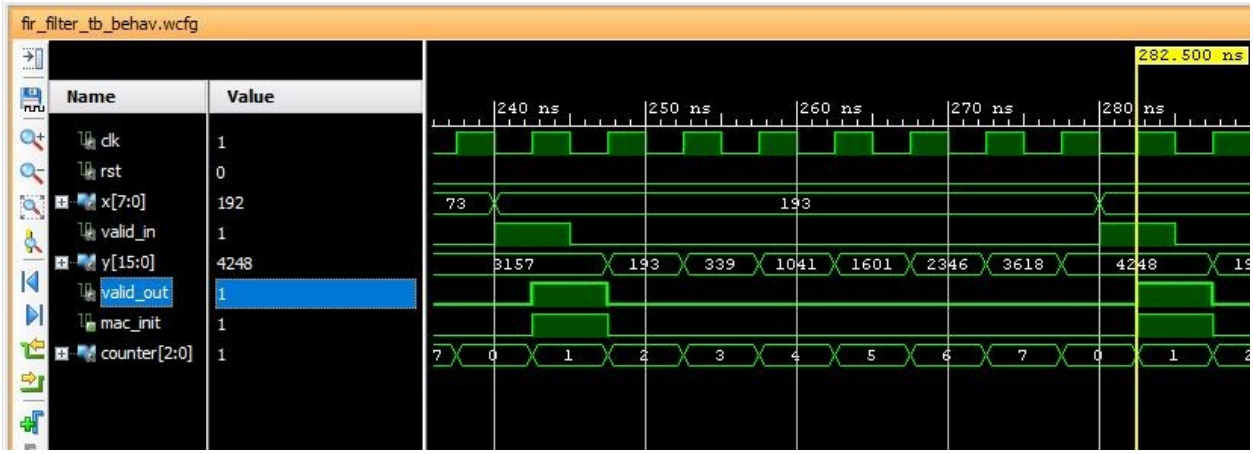
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fir_filter_tb is
end fir_filter_tb;

architecture bench of fir_filter_tb is

  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);
    y: out std_logic_vector(15 downto 0);
    valid_out: out std_logic
  );
  end component;

  signal clk: std_logic;
  signal rst: std_logic;
  signal valid_in: std_logic;
  signal valid_out: std_logic;
  signal x: std_logic_vector (7 downto 0);
  signal y: std_logic_vector (15 downto 0);

  constant CLOCK_PERIOD : time := 5 ns;

  type input_type is array(19 downto 0) of integer;
  signal input_sig: input_type := (90, 212, 149, 140, 234, 73, 193, 192, 97, 145, 19, 13, 135, 199, 239, 33, 145, 120, 3, 86);


begin

  filter: fir_filter
  port map (
    clk => clk,
    rst => rst,
    valid_in => valid_in,
    x => x,
    y => y,
    valid_out => valid_out
  );

  simulation : process
  begin
    rst <= '0';
```

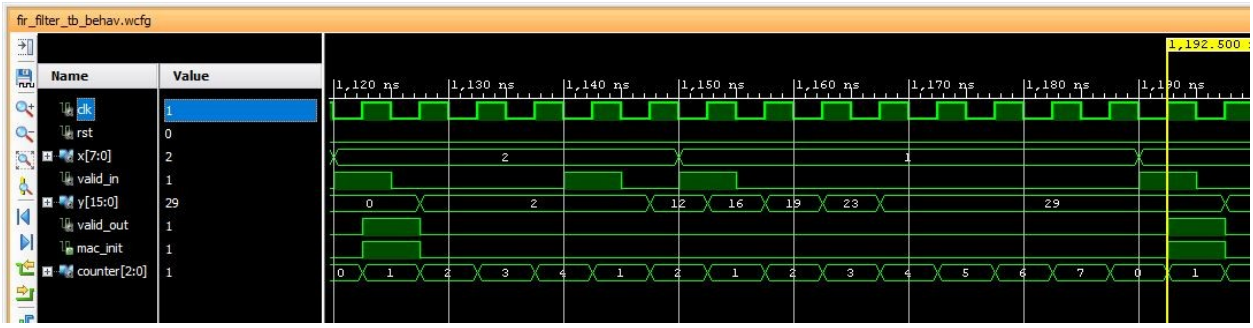
```
    for i in 19 downto 0 loop
        valid_in <= '1';
        x <= std_logic_vector(to_unsigned(input_sig(i), 8));
        wait for CLOCK_PERIOD;
        valid_in <= '0';
        wait for 7*CLOCK_PERIOD;
    end loop;

    for i in 0 to 7 loop
        valid_in <= '1';
        x <= std_logic_vector(to_unsigned(0, 8));
        wait for CLOCK_PERIOD;
        valid_in <= '0';
        wait for 7*CLOCK_PERIOD;
    end loop;

    valid_in <= '1';
    x <= "00000010";
    wait for CLOCK_PERIOD;
    valid_in <= '0';
    wait for 3*CLOCK_PERIOD;
    valid_in <= '1';
    wait for CLOCK_PERIOD;
    valid_in <= '0';
    wait for CLOCK_PERIOD;

    for i in 1 to 8 loop
        valid_in <= '1';
        x <= std_logic_vector(to_unsigned(i, 8));
        wait for CLOCK_PERIOD;
        valid_in <= '0';
        wait for 7*CLOCK_PERIOD;
    end loop;

    wait for 7*CLOCK_PERIOD;
    wait;
end process;
```

```
  generate_clock : process
  begin
    clk <= '0';
    wait for CLOCK_PERIOD/2;
    clk <= '1';
    wait for CLOCK_PERIOD/2;
  end process;

end bench;
```

We see that from the moment the user enters (valid_in) the counter is reset, so that in the next 8 cycles it gives the correct addresses to the memories and the operations are done on the mac. The intermediate results of the operations come out at the output y, but are considered valid only when valid_out = '1' that occurs when the counter goes from 0 to 1 (which in our system translates that the counter made a complete circle and we have an overflow), after from 8 cycles.



In the following case we notice that if the user enters the inputs faster than the interval of 8 cycles, the system exits the correct output after 8 cycles from the last valid_in.



3) The following resources were used in the implementation:

Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	71	35200	0.20
LUT	47	17600	0.27
I/O	33	100	33.00
DSP48	1	80	1.25
BUFG	1	32	3.12

Graph **Table**