Politecnico di Torino
III Facoltà di Ingegneria

Finite Impulse Response Filter
Integrated Systems Architectures

Master degree in Electronic Engineering

Group: 02

Alessi Valeria 198141
Renzi Alessandro 197783
Tiralongo Antonio 200021

February 6, 2014

Contents

CHAPTER 1

Introduction

A finite impulse response (FIR) filter is a particular type of digital filter, where the unit-sample response of the system contains a finite number of non-zero samples, i.e. h(n) is of finite duration.
FIR filters are designed to modify the frequency properties of the input signal x[n] to meet specific design requirements.
The properties of the filter are completely described by its unit-sample response h[n] that multiplied by x[n] gives the output signal y[n]:

$$y[n] = \sum_{j=0}^{n_t-1} H_j \cdot x[n-j]$$

The parameter n_t is the number of taps of the filter. Since in our case it is equal to 5, the complete expression for y[n] is the following:

$$y[n] = H_0 \cdot x[n] + H_1 \cdot x[n-1] + H_2 \cdot x[n-2] + H_3 \cdot x[n-3] + H_4 \cdot x[n-4]$$

The goal of this lab is to design in VHDL a pipelined architecture of the FIR filter, that receives input samples on 13 bits, represented as 2-complement normalized-fixed point values. The number of required pipeline levels is equal to 2.
In the following chapter, a block diagram implementation of the filter will be presented.

CHAPTER 2

FIR architecture

The 5-tap FIR filter (fig: 2.1) can be implemented using:

- 5 multipliers
- 4 adders
- 4 delay elements to propagate D_{IN}
- 5 pipeline registers
- 4 registers to propagate V_{IN}
- 2 registers to store the input sample D_{IN} and the associated enable signal V_{IN}
- 2 registers to store the output sample D_{OUT} and the associated enable signal V_{OUT}

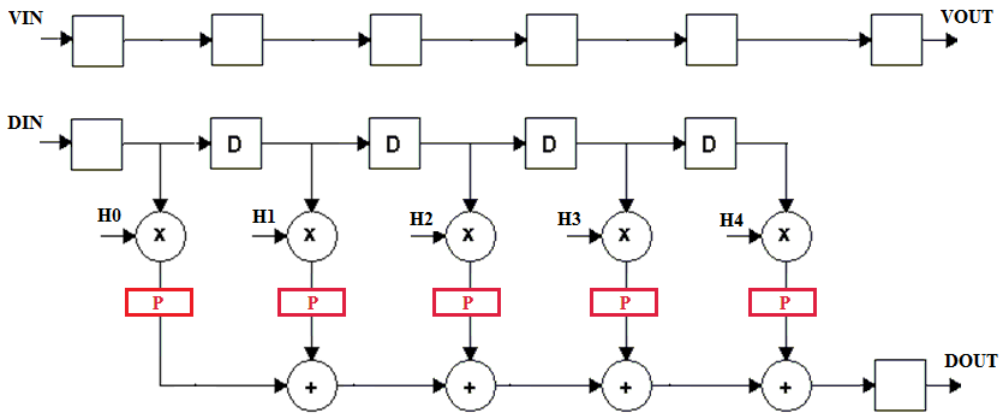

Figure 2.1: A 5-tap FIR filter

D_{IN} is the input sample that is stored in the first register every clock cycle, only if the enable signal V_{IN} is equal to 1. This signal has to be shifted as the input sample,

through all the delay elements. For this reason, a chain of flip-flops has been added to propagate V_{IN} to the output V_{OUT}.
It is important to clarify that the registers not marked with the letter 'D' are not proper to the algorithm, but have been added in order to store the input and output signals at every clock cycle, and to propagate the enable signal.
Pipeline registers have been highlighted in red and are used to divide the algorithm in two stages: the first one is the multiplication between the constant value H and the delayed input sample, while the second one is the addition between the product produced by the current tap and the one produced by the previous tap.
By introducing a level of pipeline registers, it is possible to reduce the critical path, which can be exploited to either increase the clock frequency or to reduce the power consumption at the same speed. In this case, the critical path delay is the maximum between the combinational delay of the adder and the combinational delay of the multiplier.

$$T_{CP} = max(T_A, T_M)$$

As we will see later, the second term is the dominant one, so we can state that:

$$T_{CP} = T_M$$

While pipelining reduces the critical path, it leads to a penalty in terms of an increase in latency. Latency is the difference in the availability of the first output data in the pipelined system and in the sequential system. Since we have introduced one level of pipeline registers, the latency is increased by one clock cycle.
Let's now have a look in detail at the single hardware components of the filter.

2.1 Multiplier

Each multiplication is computed by a Booth multiplier, which examines adjacent pairs of bits of the 13-bit multiplier (in 2-complement representation), including an implicit bit below the least significant bit, that is assumed equal to 0. The algorithm in pseudo-code can be written as:

```
i = 0
P = 0
while i < M-2 loop
   P <= P + Vp(b[i+1], b[i], b[i-1])
   A <= A * 4
   i <= i + 2
end loop
```

The basic idea is that, instead of shifting and adding for every column of the multiplier term and multiplying by 1 or 0, we only take every second column, and multiply by ±1, ±2, or 0, to obtain the same results.
Vp is a partial value corresponding to a tern of consecutive multiplicand, defined as follows:

b[i+1]	b[i]	b[i-1]	Vp
0	0	0	0
0	0	1	+A
0	1	0	+A
0	0	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

According to the above table, Booth recodes the multiplier term, with three bits in one block as b[i+1], b[i], b[i-1], such that each block overlaps the previous block by one bit.
The hardware implementation of the Booth's Algorithm is shown in figure 2.2.

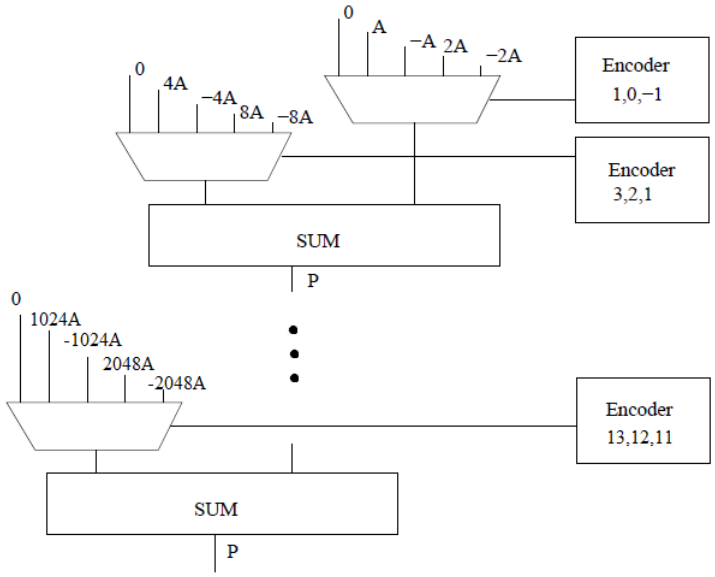

Figure 2.2: Booth multiplier

The encoders are designed in a behavioral way and are in charge of selecting the right input of the multiplexers, according to the table reported above. Moreover, the input signals of the multiplexers (2A, 4A ...) can be easily obtained in parallel, because

only left shifts are required.
Finally, the SUM blocks are ripple carry adders on 26 bits, since the final result of the multiplication will be represented on twice the number of bits of the input signals.
Here is reported a Booth multiplier instantiation in VHDL:

```
Mi: BOOTH_MUL
        generic map (N => nb)
        port map (A => S(i), B => SH(i), P => SM(i));
```

2.2 Adder

Each addition between the product of the current tap and the product of the previous tap is performed by a ripple carry adder, which for two n-bit operands, requires n full adders. Even though the result of the multiplication is represented on 26 bits, the adders are instantiated on 28 bits, to take into account the possible overflow that could arise in the addition chain.
The maximum value that can appear at the output of a multiplier is:

$$MAX\_P = (-2^{n_b-1}) \cdot (-2^{n_b-1}) = (-2^{12}) \cdot (-2^{12}) = 2^{24}$$

Supposing that at the same time all multipliers generate this value, the maximum value for the output sample D_{OUT} is:

$$MAX\_D_{OUT} = 5 \cdot 2^{24}$$

that is representable, as a 2-complement, on 28 bits.
Since all products are on 26 bits, they have to be sign-extended before applying them at the inputs of the corresponding adders. This operation can be easily written in VHDL as:

```
SP_EXTENSION: for i in 0 to nt-1 generate
        SP_EXT(i)(2*nb-1 downto 0) <= SP(i);
        SP_EXT(i)((2*nb+2)-1 downto 2*nb) <= (others => SP(i)(2*nb-1));
end generate;
```

The MSB of SP signals is replicated for bit 27 and 26 of SP_EXT signals.
In figure 2.3 the first RCA of the filter is shown.

Obviously, only the upper 13 bits of the output of the last adder will be transferred on D_{OUT}. However, with this choice we do not lose anything in terms of precision, until the last addition. The result will have 2 bits for the integer part, and the remaining bits for the fractional part.

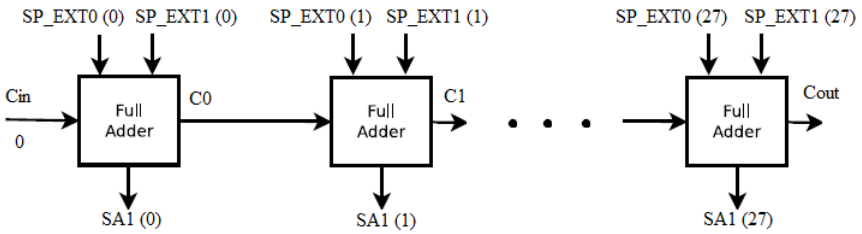

Figure 2.3: Ripple carry adder

2.3 Delay elements

Delay elements are used to shift the input sample at every clock cycle, according to the expression of y[n]. They are implemented using 13-bit shift registers, that store the input value only if the enable signal V_{IN} is high.
The VHDL implementation of the delay element is shown below:

```
PSYNCH: process(CK,RESET)
   begin
     if CK'event and CK='1' then
       if RESET='0' then
         Q <= (OTHERS => '0');
       elsif ENABLE ='1' then
         Q <= D;
       end if;
     end if;
   end process;
```

2.4 Pipeline registers

Pipeline registers are placed between multipliers and adders to divide the architecture in two separate stages. They are similar to delay elements, with the only difference that the enable signal is not present. When a rising edge of the clock arrives, the output of the multiplier is transferred to the output of the corresponding pipeline register.
The VHDL implementation of the pipeline register is shown below:

```
PSYNCH: process(CK,RESET)
   begin
     if CK'event and CK='1' then
       if RESET='0' then
         Q <= (OTHERS => '0');
       else
         Q <= D;
       end if;
     end if;
   end process;
```

2.5 Registers for enable signal

This chain of registers has been introduced to propagate the enable signal V_{IN} to the output V_{OUT}. If V_{OUT} is equal to 1, it means that the output sample is valid and can be written on file.
Actually, since the enable signal is only one bit, the chain is made of FFs. However, to not generate an additional component, we have used the pipeline register component, specifying N=1 inside the generic map.

```
Vi: REGISTER_PIPELINE
     generic map(N => 1)
     port map(D => V(i-1), CK =>CK, RESET => RST_n, Q => V(i));
```

CHAPTER 3

Test-bench

The test-bench is made up of three components:

- Data generator
- FIR filter
- Save data

The FIR filter has been analyzed in the previous chapter. Let's now focus on the other two blocks.

3.1 Data generator

The data generator is the HW block in charge of generating the clock, the enable V_{IN}, the reset, the signal to stop the simulation END_SIM and of passing the input samples and the constants $H_0 - H_4$ to the filter.

3.1.1 Clock generation

The clock is generated using a process that toggles the clock signal CK_i every nanosecond. Then, CK_i is put in AND with the negated end simulation signal to obtain the final clock CK_p, that will be applied to the filter. In this way, when the simulation is ended, END_SIM is equal to 1 and the clock remains stable at 0.

```
process
begin
   if (CK_i = 'U') then
      CK_i <= '0';
   else
        CK_i <= not(CK_i);
   end if;
   wait for Period/2;
end process;

CK_p <= CK_i and (not(END_SIM_i));
```

3.1.2 Input samples

Input samples are read from a text file using a process that reads every line of the file, converts the value into a std_logic_vector of 13 bits and finally assigns the new value to the SAMPLE output.
Since the simulation is based on 100 input samples, when the counter reaches 100, END_SIM is put equal to 1 and the clock is stopped.

```
process (CK_p, RESET)
file infile : text is in "inputs.txt";
variable inline: line;
variable counter : integer := 0;
variable in_sample : integer;
begin
   if RESET = '0' then
      VIN_i <= '0';
      END_SIM_i <= '0';
    elsif CK_p'event and CK_p = '1' then
      readline(infile, inline);
      read(inline, in_sample);
      SAMPLE <= conv_std_logic_vector(in_sample,nb);
      counter := counter+1;
      VIN_i <= '1';
      if(counter = NUM_SAMPLES) then
         END_SIM_i <= '1';
      else
         END_SIM_i <= '0';
      end if;
   end if;
end process;
```

3.1.3 Filter coefficients

Filter coefficients, called also tap weights, are constant values that make up the impulse response. They have been computed using MATLAB, in the case of a low-pass filter with linear phase.

```
H0 <= conv_std_logic_vector(34,nb);
H1 <= conv_std_logic_vector(893,nb);
H2 <= conv_std_logic_vector(2241,nb);
H3 <= conv_std_logic_vector(893,nb);
H4 <= conv_std_logic_vector(34,nb);
```

3.2 Save data

This block is in charge of writing the output samples on a text file. However, the operation is enabled only if V_{OUT} is equal to 1, so that in the first five clock cycles no result is written, since the pipeline has firstly to be filled.

```
process (CK, RST_n)
file fp_out : text open WRITE_MODE is "./results";
variable buf_o: line;
begin
   if RST_n = '0' then
      null;
    elsif CK'event and CK = '1' then
      if VOUT = '1' then
           write(buf_o, conv_integer(signed(DOUT)));
           writeline(fp_out, buf_o);
      end if;
   end if;
end process;
```

CHAPTER 4

Simulation

When designing any project, the simulation is necessary in order to check the device's behavior according to the project specification.
As just mentioned in Sec. 3.1, the simulation made is based on the generation of 100 different input signals, by reading them from a file, and the analysis of the file containing the output signals. The simulation was made by running the test bench, mentioned above, on Modelsim. As said before, the simulation outputs have been written on a file called "results". The file has been checked by hand using the following formula:

$$y[n] = H_0 \cdot x[n] + H_1 \cdot x[n-1] + H_2 \cdot x[n-2] + H_3 \cdot x[n-3] + H_4 \cdot x[n-4]$$

Using the filter coefficients (H_i) computed, the final expression is the following:

$$y[n] = 34 \cdot x[n] + 893 \cdot x[n-1] + 2241 \cdot x[n-2] + 893 \cdot x[n-3] + 34 \cdot x[n-4]$$

A screenshot of the simulation is shown in the figure that follows 4.1.

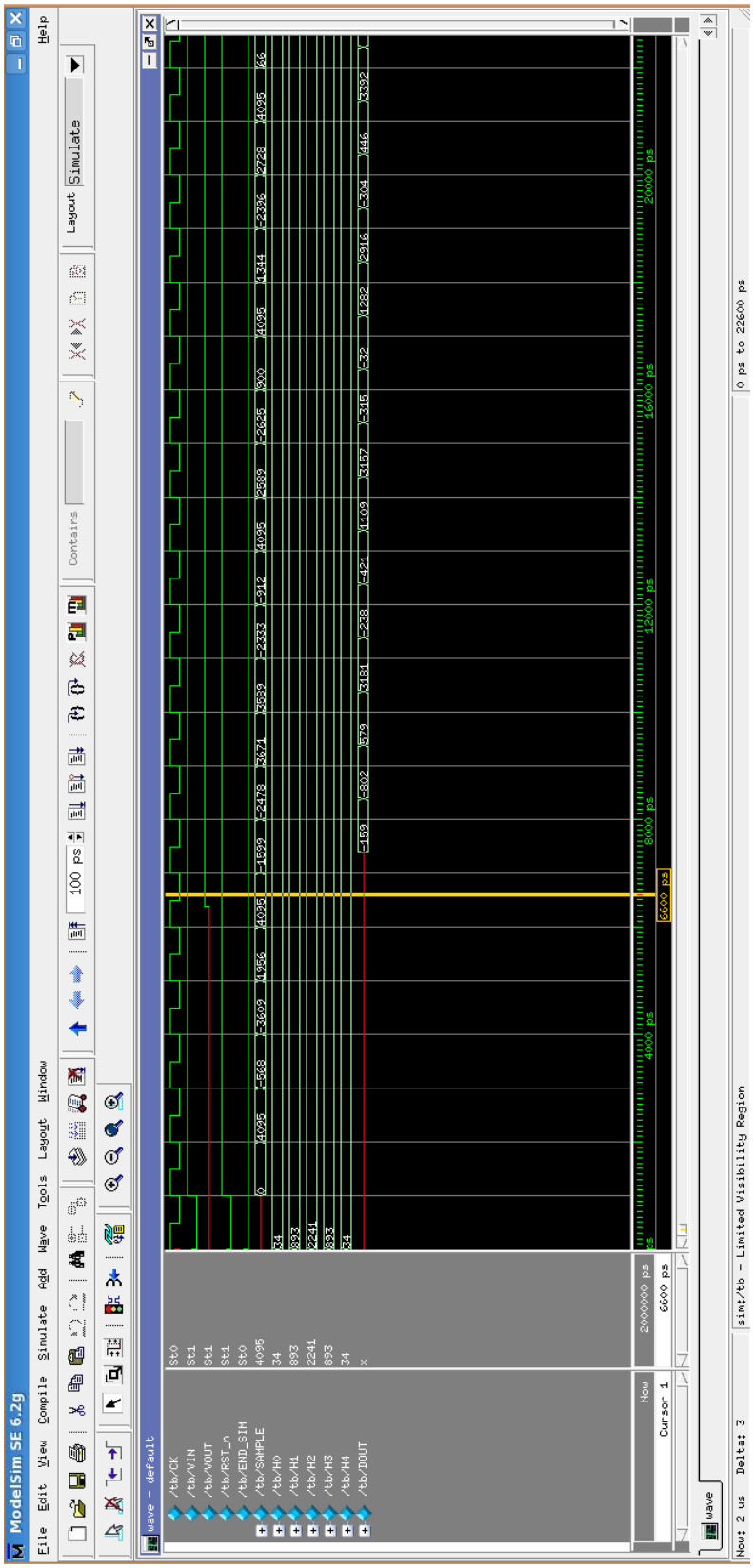

Figure 4.1: Simulation

CHAPTER 5

Logic Synthesis

Logic synthesis has been carried out using Design Compiler, the RTL Synthesis tool by Synopsys.
As a preliminary step, it was necessary to copy in the working directory the synopsys_dc.setup file, that is used for initializing design parameters and variables, declare design libraries, and so on. The commands in this file are executed when Design Compiler is invoked.
The first command in the synthesis flow is analyze that reads the HDL source files and checks for syntactical errors. For example:

```
analyze -f vhdl -lib WORK ../src/constants.vhd
```

Then, the elaborate command is invoked and it is in charge to:

- Translate the design into a technology-independent design (GTECH), starting from the intermediate files generated during analysis.
- Allow changing of parameter values (generics) defined in the source code.
- Replace the VHDL arithmetic operators in the code with DesignWare components.

At this point, if the elaboration completed successfully, the design is represented in GTECH format, which is an internal, equation-based, technology-independent design format.
Before synthesizing the design with compile_ultra command the clock signal with a period of 10 ns was generated with the following command:

```
create_clock -name CK -period 10.0 CK
```

Moreover, other parameters like clock uncertainty, input and output delay, output load and the wire model were specified. After synthesis we have observed the following results:

- Timing
- Area
- Power

5.1 Timing

```
TIMING ANALYSIS
-------------------------------------------------------------
  Startpoint: H2[3] (input port clocked by CLOCK)
  Endpoint: Pi_2/Q_reg[25]
            (rising edge-triggered flip-flop clocked by CLOCK)
  Path Group: CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------------
  FIR                tsmc090_wl40          fast

  Point                                    Incr       Path
  -------------------------------------------------------------
  clock CLOCK (rise edge)                  0.00       0.00
  clock network delay (ideal)              0.00       0.00
  input external delay                     0.50       0.50 r
  H2[3] (in)                               0.00       0.50 r
  U7565/Y (INVX5)                          0.13       0.63 f
  U7766/Y (NOR3X4)                         0.18       0.81 r
  .
  .
  .
  U5152/Y (OAI21X4)                        0.06       9.77 f
  U7706/Y (AOI21X2)                        0.15       9.91 r
  Pi_2/Q_reg[25]/D (DFFQX2)                0.00       9.91 r
  data arrival time                                   9.91

  clock CLOCK (rise edge)                 10.00      10.00
  clock network delay (ideal)              0.00      10.00
  clock uncertainty                       -0.07       9.93
  Pi_2/Q_reg[25]/CK (DFFQX2)               0.00       9.93 r
  library setup time                      -0.02       9.91
  data required time                                  9.91
  -------------------------------------------------------------
  data required time                                  9.91
```

```
  data arrival time                                  -9.91
  -----------------------------------------------------------
  slack (MET)                                         0.00
```

The delay report shows delay calculation in two sections: the first section for data arrival time and the second for data required time.
The data arrival time is the time required for a signal to travel from a starting point to an end point of a path.
The data required time is the maximum time a signal has for travelling that path.
The time difference between data required time and data arrival time is called slack or timing margin of the path. If the slack is negative, there is a timing violation on that path.
In our case the slack is equal to 0, which means that the maximum clock frequency is 100 Mhz. However, it is important to highlight that during this phase we are not considering parasitics.

5.2 Area

```
AREA ANALYSIS
---------------------------------------------------------------
Number of ports:              95
Number of nets:             7175
Number of cells:            6309
Number of references:        122

Combinational area:       41482.929559
Noncombinational area:    4065.667150
Net Interconnect area:    3043242.000000

Total cell area:          45548.597656
Total area:               3088790.597656
---------------------------------------------------------------
```

The above report simply shows the total area of the design. It is the sum of three components: combinational, noncombinational, and net interconnect area.
The area due to logic cells in the design is made by the combinational (basic logic gates like ANDs, ORs, etc) and the noncombinational (registers) factors. The third factor affecting the area (net interconnect area) is due to the wires connecting these cells.
It is the dominant value, since heavy routing is required for the multipliers and the adders.

5.3 Power

```
POWER ANALYSIS
------------------------------------------------------------
Cell Internal Power  =   1.6906 mW   (17%)
Net Switching Power  =   8.1079 mW   (83%)
                         ---------
Total Dynamic Power  =   9.7985 mW  (100%)

Cell Leakage Power   = 145.2489 uW
------------------------------------------------------------
```

The internal power is caused by the charging of internal loads as well as by the short-circuit current between N and P transistors of a gate, when both are on. As we can see from the report, it is the dominant component of the dynamic power, dissipated any time the capacitive load of a net charges or discharges.
Dynamic power has been computed assuming a switching activity equal to 0.5, which is a non realistic value, since some nodes could switch more frequently than others. In the following chapter we will make a more accurate power analysis.

CHAPTER 6

Switching-activity-based power consumption estimation

Switching activity is a parameter that indicates the probability of a node to toggle its value. It influences the dynamic power, that can be written as:

$$P_{SW} \propto \frac{1}{2} {V_{DD}}^2 f C_L E_{SW}$$

To extract the switching activity of the circuit we used Modelsim, that after all steps generates the **.saif** backward annotation file. This file has to be read (together with the verilog netlist of the circuit) by Design Compiler, and includes the switching activity for all nodes of the circuit, recorded during simulation. However, before running Modelsim with options to record switching activity, we had to modify the test-bench by specifying the region to be monitored, that has the same name of the filter instance, placed in test-bench.

```
initial begin
   $read_lib_saif("../saif/fast.saif");
   $set_gate_level_monitoring("on");
   $set_toggle_region(FILTER);
   $toggle_start;
end
```

The monitoring is stopped when the end simulation signal END_SIM becomes equal to '1', i.e. after reading 100 input samples.

```
always @ ( END_SIM ) begin
   if (END_SIM) begin
      $toggle_stop;
      $toggle_report("../saif/myfir_back.saif", 1.0e-9, "tb.FILTER");
   end
end
```

At this point, the .saif file is read by Design Compiler and a new power report is generated.

```
POWER ANALYSIS
------------------------------------------------------------
Cell Internal Power  =  51.6594 mW   (19%)
Net Switching Power  = 216.1070 mW   (81%)
                         ---------
Total Dynamic Power  = 267.7664 mW  (100%)

Cell Leakage Power   = 157.5060 uW
------------------------------------------------------------
```

As we can see, the total dynamic power is much higher than the one computed with a switching activity equal to 0.5. This means that in the previous case we were underestimating the number of toggles of each node of the circuit.

CHAPTER 7

Place and route

To perform the Place and Route we used Cadence SoC Encounter, the steps needed are:

- Importing the design
- Floorplanning
- Power planning and routing
- Cell placing
- Clock tree synthesis
- Filler Placement
- Signal routing
- Timing analysis
- Design analysis and verification

Importing the design First we need to import the synthesized netlist generated by Design Compiler. To do so we customized the configuration file specifying all the needed files like the libraries and the same netlist.

Floorplanning Within this step the software set the area to be assigned to the design, in our case it is set in the center of the chip with a margin with respect to the die boundary.

Power planning and routing The power planning aim is to place the necessary metal stripes to provide power to the entire chip. First two rings are added around the chip boundary for VCC and GND, then some vertical stripes are placed in order to reduce the path length of the current, and so the voltage drop along the power lines. At this point the horizontal wires are placed to power the standard cells and complete the power planning.

Cell placing Now the library cells will be placed according to the synthesized netlist. Encounter will try to place these cells in the best possible way to limit the routing congestion.

Clock tree synthesis Clock tree synthesis (CTS) is the process of insertion of buffers or inverters along the clock paths of the design, in order to achieve minimum skew or balanced skew. The goal of CTS is to minimize the load and the delay of the clock tree.
In the *.ctstch* file we specify a tree of three levels, where the first level uses CLKBUFX1 buffers (where X1 indicates that they are able to drive only one gate), the second level uses CLKBUFX4 buffers, while the third one uses CLKBUFX8 buffer. As we go up in the tree the load capacitance seen by buffers increases, so we need to increase also their driving capability.
After clock synthesis all results are stored in the report file and a verilog file, with all buffers that have been added to the design, is generated.

```
------------------------------Clock Tree Report--------------------------------
Nr. of Subtrees                : 0
Nr. of Sinks                   : 214
Nr. of Buffer                  : 25
Nr. of Level (including gates) : 3
Max trig. edge delay at sink(R): P0_Q_reg_24_/CK 254.6(ps)
Min trig. edge delay at sink(R): R0_Q_reg_7_/CK 246.2(ps)

                                  (Actual)              (Required)
Rise Phase Delay               : 246.2~254.6(ps)       0~10000(ps)
Fall Phase Delay               : 281.2~290.8(ps)       0~10000(ps)
Trig. Edge Skew                : 8.4(ps)               10000(ps)
Rise Skew                      : 8.4(ps)
Fall Skew                      : 9.6(ps)
Max. Rise Buffer Tran          : 158(ps)               10000(ps)
Max. Fall Buffer Tran          : 152(ps)               10000(ps)
Max. Rise Sink Tran            : 46.9(ps)              10000(ps)
Max. Fall Sink Tran            : 43.7(ps)              10000(ps)
```

```
***** NO Transition Time Violation *****

***** NO Capacitance Violation *****
--------------------------------------------------------------------------------
```

25 buffers have been inserted with a rise skew of 8.4 ps and a fall skew of 9.6 ps.

- Rise skew is calculated based on rise edge at the clock root.
- Rise skew is calculated based on rise edge at the clock root.
- Fall skew is calculated based on fall edge at the clock root.
- Triggering edge Skew is calculated based on arrival times of active signals on clock pins.
- Transition time is the time taken by the clock signal to make a transition from 20% to 80% of the maximum value.

The clock skew is not equal to 0 because paths are not perfectly balanced. However, it is a small value with respect to the clock period.
Main causes of clock skew could be:

- unequal wire length;
- unequal buffer delay;
- unequal load;
- IR-drop.

Filler placement This step is required for technological reasons to guarantee continuity in N and P wells in each row. It consists of filling the holes on the die with filler cells.

Signal routing This step is divided into two phases: the first phase is a sort of raw routing, a planning of the wire position. The second phase is fine routing of the wires, which connect all the cells; here Encounter will try to find the best solution for the wires positioning.

Timing analysis To do the timing analysis we first have to specify the operating conditions like temperature, power supply voltage and the process variations, then we must extract the parasitics (resistance and capacitance) of each wire in our design. The **ExtractRC** command returns two files *.setload* and *.setres* that are used to set capacitance and resistance for each net respectively. They have to be included in the *.sdc* file, that is used to fix timing constraints.

```
#/***************************************************
# * Timing constraint file in SDC format
# ***************************************************/
set_wire_load_model -name tsmc090_wl40 -library fast
create_clock [get_ports CK]  -name CLOCK  -period 10  -waveform {0 5}
source myfir.setload
source myfir.setres
```

Now we are ready for the timing analysis, we load the timing constraints (used also in the synthesis) and launch the analysis.
As we anticipated in the Logic Synthesis chapter, the maximum operating frequency, found in that phase, is not a real value, since we were not considering resistance and capacitance parasitic values for metal wires. Starting from the minimum declared period (10 ns) we performed Timing Analysis and we found no violating paths.

```
-----------------------------------------------------------
*info: Report constrained paths
*      Path type: max (data)
*      Format: long
*** Found 0  violating paths ***
-----------------------------------------------------------
```

It means that the FIR can really be clocked at 100 Mhz.

Design analysis and verification Since there are no timing violations we can proceed with the design verification. It checks if there are floating wires and if there are geometric issues related to the design rules, imposed by the technology.

```
-----------------------------------------------------------
Begin Summary
  Cells       : 0
  SameNet     : 0
  Wiring      : 0
  Antenna     : 0
  Short       : 0
  Overlap     : 0
End Summary

No DRC violations were found
-----------------------------------------------------------
```

Finally, the total number of gates is reported below.

```
---------------------------------------------------------
Gate area 2.1168 um^2
Level 0 Module FIR  Gates = 21585
Cells = 6334
Area = 45691.8 um^2
---------------------------------------------------------
```

CHAPTER 8

Post place and route simulation and switching-activity-based power consumption estimation

In order to perform the post place-and-route simulation, it's necessary to simplify the verilog test-bench, because in this case the procedures are simpler than post-synthesis ones. However, we removed the initial section from the test-bench:

```
initial begin
...
end
```

And we also removed the toggle related process:

```
always @ ( END SIM i ) begin
...
end
```

To obtain an accurate switching activity report, we also linked the delay file (.sdf) generated by Cadence Encounter and we wrote the switching activity informations in a .vcd file using Modelsim. This file has then been used during the power consumption estimation in Cadence Encouter, in order to produce the following report:

```
##############################################
# The Power Analysis Report for VDD net      #
##############################################
power supply: 1.1 volt
average power between 0.0000e+00 S and 9.9900e-08 S
Total id in vcd file: 7198
    In module tb/FILTER valid id: 7198
```

```
         redundant id: 0
    In module tb/FILTER invalid id: 0
        redundant id: 0
Total activity in vcd file: 563755
    In module tb/FILTER valid activity: 563754
    In module tb/FILTER invalid activity: 1
average power(default): 7.1291e+01 mw
    average switching power(default): 2.4171e+01 mw
    average internal power(default): 4.6967e+01 mw
    average leakage power(default): 1.5277e-01 mw
    user specified power(default): 0.0000e+00 mw
average power by cell category:
    core:  7.1291e+01 mw
    block: 0.0000e+00 mw
    io:    0.0000e+00 mw
biggest toggled net: RST_n
    no. of terminal: 188
    total cap: 6.2511e+02 ff
```

In this case, the average switching power is less than in the post-synthesis power report. This is due to the place-and-route and to the switching activity computed by Modelsim.

CHAPTER 9

MATLAB

We used Matlab as a codesign tool for data generation and analysis.

9.1 Synthesis

As a first step we created a script to synthesize the fir filter's coefficients. This script exploited a Matlab's built-in function to do so, then we converted these coefficients to a format compatible with Modelsim's simulation. To perform this conversion we first converted the floating point coefficients into fixed point numbers, using the built-in datatypes, with the wanted format in terms of bits number for integer part and fractional part. Then, we converted again the fixed point binary number to a decimal representation, understandable by Modelsim. We assumed a filter with a cut-off frequency of 10KHz working at 48KHz. The code is the following.

```
b=fir1(M-1,ft/(fs/2)) % Coefficient generation (M taps -> M-1 order, cut
    frequency normalized with respect to the Nyquist frequency)
f=sfi(b.',word_length,fractional_length); % convert to fixed point for
    saving
h=twos2dec(bin(f));
dlmwrite('h', h, 'delimiter', '\n');
dlmwrite('b', b, 'delimiter', '\n');
```

9.2 Data generation

The second step was the data generation. We made different scripts for different waveforms, such as square wave, sawtooth wave, two tones signal and random signal. As before the generated data has been converted into integer representation for compatibility with Modelsim. The following is a code sample of the sawtooth generator.

```
signal_frequency = 4*ft;

t = 0:1/fs:n/fs;

x = sawtooth(t*signal_frequency);
f=sfi(x.',word_length, fractional_length); % convert to fixed point for
    saving
w=twos2dec(bin(f)); % binary conversion of the fixed point converted to
    decimal
dlmwrite('input_sawtooth', w, 'delimiter', '\n');
```

9.3 Spectral Analysis

Once all the generated inputs have been simulated with Modelsim, we took the output and analyzed it to check if the filtering was actually performed.

```
x = dlmread('input_2sine', '\n');
z = dlmread('result_modelsim', '\n');
b = dlmread('b', '\n');
a = 1;
N = length(z);
f=[0:fs/(N-1):fs];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Filter response
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[H,F]=freqz(b,a,N,fs);
modH=20*log(abs(H));
figure, plot(F,modH);
title('Filter response');
xlabel('Frequency [Hz]');
ylabel('Amplitude [dB]');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Original
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S=1/(N*fs)*(abs(fft(x(1:N))).^2);
figure, subplot(2,1,1), plot(f,S);
title('Original spectrum');
xlabel('f');
ylabel('S(f)');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Filtered
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S=1/(N*fs)*(abs(fft(z(1:N))).^2);
subplot(2,1,2), plot(f,S);
title('Filtered spectrum');
xlabel('f');
ylabel('S(f)');
```

The script is structured in three sections: the first plots the frequency response of the filter, which is the following:

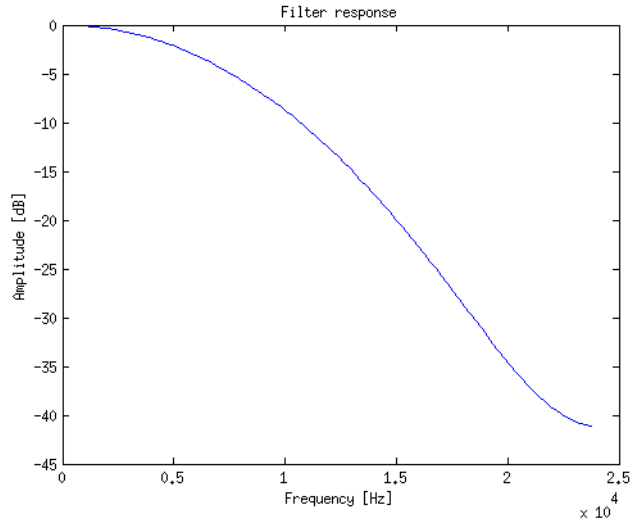

Figure 9.1: Filter's frequency response

The second plots the original spectrum of the non-filtered signal and the third is the spectrum of the filtered signal. We present the results for every generated signal.

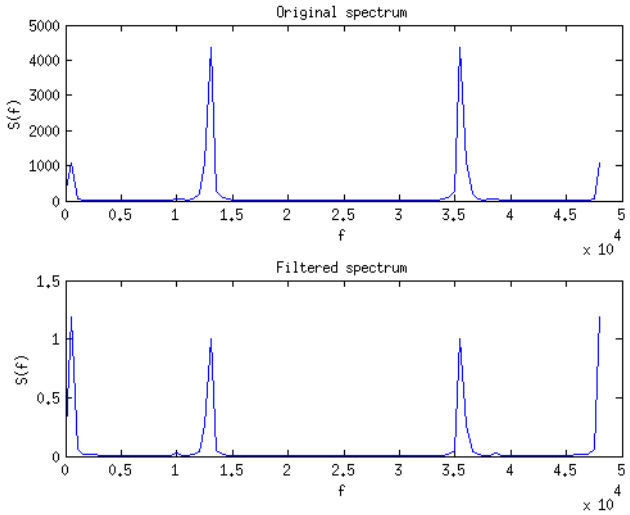

Figure 9.2: Two tones signal spectrum

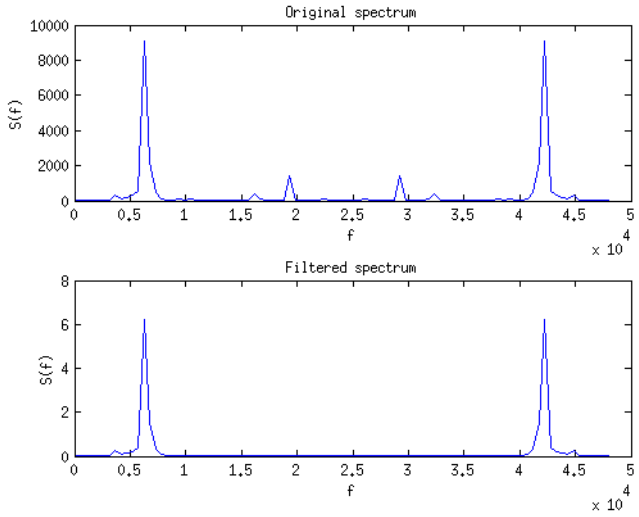

Figure 9.3: Square wave signal spectrum

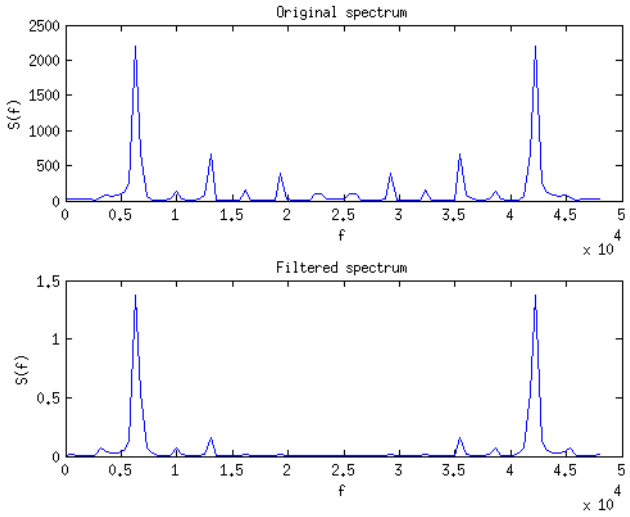

Figure 9.4: Sawtooth wave signal spectrum

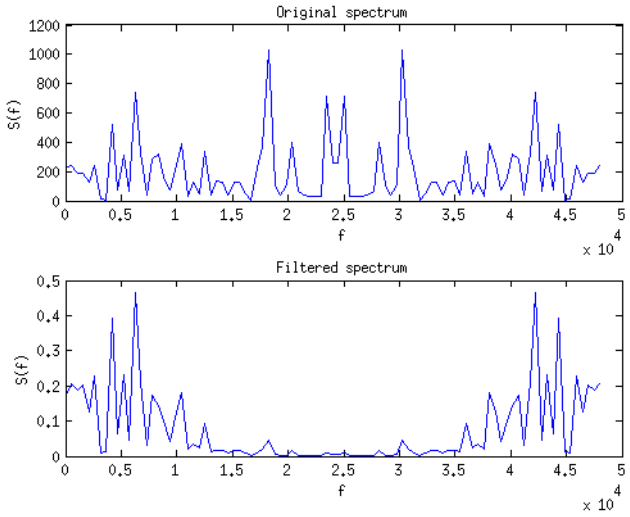

Figure 9.5: Random signal spectrum

APPENDIX A

fir.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use WORK.constants.all; -- libreria WORK user-defined

entity FIR is
   port (  DIN: in std_logic_vector(nb-1 downto 0);
      VIN: in std_logic;
      RST_n:  in std_logic;
      CK:  in std_logic;
      H0:  in std_logic_vector(nb-1 downto 0);
      H1:  in std_logic_vector(nb-1 downto 0);
      H2:  in std_logic_vector(nb-1 downto 0);
      H3:  in std_logic_vector(nb-1 downto 0);
      H4:  in std_logic_vector(nb-1 downto 0);
      DOUT: out  std_logic_vector(nb-1 downto 0);
      VOUT: out  std_logic);
end FIR;

architecture STRUCTURAL of FIR is

   type S_TYPE is array (nt-1 downto 0) of std_logic_vector (nb-1 downto 0);
   type SH_TYPE is array (nt-1 downto 0) of std_logic_vector (nb-1 downto 0);
   type SM_TYPE is array (nt-1 downto 0) of std_logic_vector ((2*nb)-1
      downto 0);
```

```
type SP_TYPE is array (nt-1 downto 0) of std_logic_vector ((2*nb)-1
    downto 0);
type SP_EXT_TYPE is array (nt-1 downto 0) of std_logic_vector ((2*nb+2)-1
    downto 0);
type SA_TYPE is array (nt-1 downto 0) of std_logic_vector ((2*nb+2)-1
    downto 0);
type V_TYPE is array (nt-1 downto 0) of std_logic_vector(0 downto 0);

signal S: S_TYPE;
signal SH: SH_TYPE;
signal SM: SM_TYPE;
signal SP: SP_TYPE;
signal SP_EXT: SP_EXT_TYPE;
signal SA: SA_TYPE;
signal V: V_TYPE;
signal VIN_i : std_logic_vector (0 downto 0);
signal VOUT_i : std_logic_vector (0 downto 0);
signal DOUT_i : std_logic_vector (nb-1 downto 0);

component RCA_GENERIC is
   generic ( N   : integer);
   port (  A: In std_logic_vector(N-1 downto 0);
      B: In std_logic_vector(N-1 downto 0);
      Ci:   In std_logic;
      S: Out  std_logic_vector(N-1 downto 0);
      Co:   Out   std_logic);
end component;

component BOOTH_MUL is
   generic (N: integer);
   port (  A: In std_logic_vector(N-1 downto 0);
      B: In std_logic_vector(N-1 downto 0);
      P: Out  std_logic_vector(2*N-1 downto 0));
end component;

component REGISTER_ALGO is
   generic (N: integer);
   port (  D: In std_logic_vector(N-1 downto 0);
      CK:   In std_logic;
      RESET:  In std_logic;
      ENABLE: In std_logic;
      Q: Out  std_logic_vector(N-1 downto 0));
end component;
```

```
   component REGISTER_PIPELINE is
      generic (N: integer);
      port (  D: In std_logic_vector(N-1 downto 0);
         CK:   In std_logic;
         RESET:  In std_logic;
         Q: Out  std_logic_vector(N-1 downto 0));
   end component;
begin

   VIN_i(0) <= VIN;
   SH(0) <= H0;
   SH(1) <= H1;
   SH(2) <= H2;
   SH(3) <= H3;
   SH(4) <= H4;

   R0: REGISTER_ALGO
      generic map(N => nb)
      port map(D => DIN, CK =>CK, RESET => RST_n, ENABLE => VIN, Q => S(0));
   M0: BOOTH_MUL
      generic map (N => nb)
      port map (A => S(0), B => SH(0), P => SM(0));
   P0: REGISTER_PIPELINE
      generic map(N => 2*nb)
      port map(D => SM(0), CK =>CK, RESET => RST_n, Q => SP(0));

   V0: REGISTER_PIPELINE
      generic map(N => 1)
      port map(D => VIN_i, CK =>CK, RESET => RST_n, Q => V(0));

   SA(0) <= SP_EXT(0);   --No addition in the first tap


   COMPONENT_INSTANTIATION: for i in 1 to nt-1 generate
      Ri: REGISTER_ALGO
         generic map(N => nb)
         port map(D => S(i-1), CK =>CK, RESET => RST_n, ENABLE => VIN, Q =>
            S(i));

      Mi: BOOTH_MUL
         generic map (N => nb)
         port map (A => S(i), B => SH(i), P => SM(i));
```

```
      Pi: REGISTER_PIPELINE
         generic map(N => 2*nb)
         port map(D => SM(i), CK =>CK, RESET => RST_n, Q => SP(i));

      Ai: RCA_GENERIC
         generic map (N => (2*nb+2))
         port map (A => SA(i-1), B => SP_EXT(i), Ci => '0', S => SA(i), Co
            => OPEN);

      Vi: REGISTER_PIPELINE         -Register used to shift VIN up to VOUT
         generic map(N => 1)
         port map(D => V(i-1), CK =>CK, RESET => RST_n, Q => V(i));

   end generate;

   SP_EXTENSION: for i in 0 to nt-1 generate
        SP_EXT(i)(2*nb-1 downto 0) <= SP(i);
        SP_EXT(i)((2*nb+2)-1 downto 2*nb) <= (others => SP(i)(2*nb-1));
           --Sign extension
   end generate;

   DOUT_i <= SA(4)((2*nb+2)-1 downto nb+2); --Only the upper 13 bits are
       considered

   DOUT_REG: REGISTER_PIPELINE           --Output register for DOUT
        generic map(N => nb)
        port map(D => DOUT_i, CK =>CK, RESET => RST_n, Q => DOUT);

   VOUT_REG: REGISTER_PIPELINE           --Output register for VOUT
        generic map(N => 1)
        port map(D => V(4), CK =>CK, RESET => RST_n, Q => VOUT_i);

   VOUT <= VOUT_i(0);

end STRUCTURAL;
```

constants.vhd

```
package CONSTANTS is
   constant nb : integer := 13;
   constant nt : integer := 5;
   constant NUM_SAMPLES : integer := 100;
```

```
end CONSTANTS;
```

register_algo.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.constants.all;

entity REGISTER_ALGO is
   generic(N: integer);
   Port (  D: In std_logic_vector(N-1 downto 0);
      CK:   In std_logic;
      RESET:  In std_logic;
      ENABLE: In std_logic;
      Q: Out  std_logic_vector(N-1 downto 0));
end REGISTER_ALGO;


architecture SYNCH of REGISTER_ALGO is -- flip flop D with syncronous reset

begin
   PSYNCH: process(CK,RESET)
   begin
     if CK'event and CK='1' then -- positive edge triggered:
       if RESET='0' then -- active low reset
         Q <= (OTHERS => '0');
       elsif ENABLE ='1' then -- active high enable
         Q <= D; -- input is written on output
       end if;
     end if;
   end process;

end SYNCH;
```

register_pipeline.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.constants.all;

entity REGISTER_PIPELINE is
   generic(N: integer);
```

```
   Port (  D: In std_logic_vector(N-1 downto 0);
      CK:   In std_logic;
      RESET:  In std_logic;
      Q: Out  std_logic_vector(N-1 downto 0));
end REGISTER_PIPELINE;


architecture SYNCH of REGISTER_PIPELINE is -- flip flop D with syncronous
    reset

begin
   PSYNCH: process(CK,RESET)
   begin
     if CK'event and CK='1' then -- positive edge triggered:
       if RESET='0' then -- active low reset
         Q <= (OTHERS => '0');
       else
         Q <= D; -- input is written on output
       end if;
     end if;
   end process;

end SYNCH;
```

booth_mul.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use WORK.constants.all; -- libreria WORK user-defined

--Before starting, we want to specify that this implementation of the Booth
    Algorithm is valid also if the MSB of the operand B is equal to 1.
--However, we had to internally extend the operand B on N+2 bits and this
    forced the instantiantion of an additional stage (encoder+adder+mux), in
    order to obtain a correct result.

entity BOOTH_MUL is
   generic (N: integer);
   port (  A: In std_logic_vector(N-1 downto 0);
      B: In std_logic_vector(N-1 downto 0);
```

```
     P: Out  std_logic_vector(2*N-1 downto 0));
end BOOTH_MUL;

architecture STRUCTURAL of BOOTH_MUL is

   type ResultSIgnals is array (N/2 downto 0) of std_logic_vector (2*N-1
      downto 0);   --If N is the number of bits, the number of MUXs is N/2+1
   type InMuxSignals is array ((4*(N/2+1))-1 downto 0) of std_logic_vector
      (2*N-1 downto 0); --Each Mux has 4 inputs different from 0, so the
      total number of signals is 4*Number_of_MUXs
   type EncoderSignals is array (N/2 downto 0) of std_logic_vector (2 downto
      0);      --If N is the number of bits, the number of encoders is N/2+1
   signal OP_A: std_logic_vector(2*N-1 downto 0);              --Op_A is
      the operand A extended on 2*N bits. Since the product is on 2N bits,
      the output of the adders has to be on 2*N bits, and
            --so also the 2 inputs of the adders have to be on 2*N bits
   signal ZERO: std_logic_vector(N downto 0);                --ZERO is a
      signal of all 0s, used to extend the operand A and also to perform
      shift left operations.
   signal B_one: std_logic_vector (2 downto 0);              --The input of
      the first decoder is B1/B0/B-1- Since B-1 is always 0 we use this
      signal to concatenate B1/B0 and '0'.
   signal B_extended: std_logic_vector (N+1 downto 0);            --This
      signal is used to extend the B operand on N+2 bits, in order to take
      into account also the possibility of MSB = 1
   signal OUT_MUX,SUM: ResultSignals;                   --SUM = outputs of
      the adders, OUT_MUX = outputs of the muxs
   signal OUT_ENC: EncoderSignals;                      --OUT_ENC = outputs
      of the encoders that work as selection signals for the muxs
   signal MUX_IN: InMuxSignals;                        --MUX_IN = inputs for
      the muxs

   component ENCODER
   port (  S: In std_logic_vector(2 downto 0);
     Y: Out  std_logic_vector(2 downto 0));
   end component;


   component MUX51_GENERIC is
   generic (N: integer);
   Port (  A0:  In std_logic_vector(N-1 downto 0);
     A1:  In std_logic_vector(N-1 downto 0);
     A2:  In std_logic_vector(N-1 downto 0);
```

```
      A3:   In std_logic_vector(N-1 downto 0);
      A4:   In std_logic_vector(N-1 downto 0);
      S: In std_logic_vector(2 downto 0);
      Y: Out  std_logic_vector(N-1 downto 0));
   end component;

   component RCA_GENERIC is
   generic ( N   : integer);
   Port (  A: In std_logic_vector(N-1 downto 0);
      B: In std_logic_vector(N-1 downto 0);
      Ci:   In std_logic;
      S: Out  std_logic_vector(N-1 downto 0);
      Co:   Out   std_logic);
   end component;

begin

   ZERO <= (others => '0');              --ZERO is initialized with all
      zeros
   OP_A(N-1 downto 0) <= A;
   OP_A(2*N-1 downto N) <= (others => A(N-1));       --Sign extension for
      operand A
   MUX_IN(0) <= OP_A;                  --Second input of the first mux is
      always OP_A
   B_extended(N-1 downto 0) <= B;               --The first N bits of B
      extended coincide with operand B
   B_extended(N+1 downto N) <= (others => '0');      --The 2 most
      significant bits are forced to '0'
   B_one(2 downto 1) <= B(1 downto 0);          --B_one(2) = B(1) , B_one(1)
      = B(0)
   B_one(0) <= '0';                      --B_one(0) = '0'

   SIGNAL_CONNECTION: for I in 1 to ((4*(N/2+1))-1) generate
                SIG:  if (I MOD 2) = 0 generate        --The index is even
                   ---> do a shift left using concatenation method
                  --MUX_IN(I) <= to_StdLogicVector((to_bitvector(OP_A)) sll
                     (I/2));
                MUX_IN(I) <= OP_A(2*N-1-I/2 downto 0) & ZERO (I/2-1 downto
                   0); --We could use also sll. Since we used it in the
                   first lab, we decided to use concatenation
             end generate;
```

```
      SIG_COMPL:    if (I MOD 2) = 1 generate        --The index is odd
         -----> calculate the 2's complement of the previous signal
           MUX_IN(I) <= ((NOT(MUX_IN(I-1))) + '1');
           end generate;
         end generate;


MUX_GENERATION: for I in 0 to N/2 generate
              MUX_I: MUX51_GENERIC
           generic map (N=>2*N)
           port map (A0=>(others => '0'), A1=>MUX_IN(4*I),
               A2=>MUX_IN(4*I+1),
               A3=>MUX_IN(4*I+2),A4=>MUX_IN(4*I+3),S=>OUT_ENC(I),Y=>OUT_MUX(I));
      end generate;


ENCODER_GENERATION: for I in 0 to N/2 generate
              ENC0:  if I = 0 generate
              ENCODER0: ENCODER
           port map (S=>B_one, Y=>OUT_ENC(I));
              end generate;

         ENC_I: if I > 0 generate
           ENCODER_I : ENCODER
           port map (S=>B_extended(2*I+1 downto 2*I-1), Y=>OUT_ENC(I));
           end generate;
         end generate;

SUM_GENERATION: for I in 1 to N/2 generate
              ADDER0: if I = 1 generate
              RCA0: RCA_GENERIC
           generic map (N => 2*N)
           Port Map (A=>OUT_MUX(I), B=>OUT_MUX(I-1), Ci=>'0',
               S=>SUM(I),Co=>OPEN);
              end generate;

         ADDER_I:if I > 1 generate
           RCA_I : RCA_GENERIC
           generic map (N => 2*N)
           Port Map (A=>OUT_MUX(I), B=>SUM(I-1), Ci=>'0',
               S=>SUM(I),Co=>OPEN);
              end generate;
         end generate;
```

```
   P <= SUM(N/2);

end STRUCTURAL;


configuration CFG_BOOTH_MUL_STRUCTURAL of BOOTH_MUL is
   for STRUCTURAL

     for SUM_GENERATION
      for all : RCA_GENERIC
         use configuration WORK.CFG_RCA_GENERIC_STRUCTURAL;
      end for;
     end for;

     for MUX_GENERATION
      for all : MUX51_GENERIC
         use configuration WORK.CFG_MUX51_GEN_BEHAVIORAL;
      end for;
     end for;

     for ENCODER_GENERATION
      for all : ENCODER
         use configuration WORK.CFG_ENCODER_BEHAVIORAL;
      end for;
     end for;
   end for;
end CFG_BOOTH_MUL_STRUCTURAL;
```

encoder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use WORK.constants.all; -- libreria WORK user-defined

entity ENCODER is
   port (  S: In std_logic_vector(2 downto 0);
      Y: Out  std_logic_vector(2 downto 0));
end ENCODER;

architecture BEHAVIORAL of ENCODER is
```

```
begin

Y <=  "000" when S ="000" else --S=000 select '0'
      "001" when S ="001" else --S=001 select '+A'
      "001" when S ="010" else --S=010 select '+A'
      "011" when S ="011" else --S=011 select '+2A'
      "100" when S ="100" else --S=100 select '-2A'
      "010" when S ="101" else --S=101 select '-2A'
      "010" when S ="110" else --S=110 select '-A'
      "000" ; --S = "111"   --S=111 select '-A'

end architecture BEHAVIORAL;


configuration CFG_ENCODER_BEHAVIORAL of ENCODER is
   for BEHAVIORAL
   end for;
end  CFG_ENCODER_BEHAVIORAL;
```

mux51_generic.vhd

```
library IEEE;
use IEEE.std_logic_1164.all; -- libreria IEEE con definizione tipi standard
    logic
use WORK.constants.all; -- libreria WORK user-defined


entity MUX51_GENERIC is
   Generic (N: integer);
   Port (  A0:   In std_logic_vector(N-1 downto 0);
      A1:   In std_logic_vector(N-1 downto 0);
      A2:   In std_logic_vector(N-1 downto 0);
      A3:   In std_logic_vector(N-1 downto 0);
      A4:   In std_logic_vector(N-1 downto 0);
      S: In std_logic_vector(2 downto 0);
      Y: Out  std_logic_vector(N-1 downto 0));
end MUX51_GENERIC;


architecture BEHAVIORAL of MUX51_GENERIC is

begin
```

```
   with S select
   Y <=  A0 when "000",
         A1 when "001",
         A2 when "010",
         A3 when "011",
         A4 when "100",
         A0 when others;

end BEHAVIORAL;


configuration CFG_MUX51_GEN_BEHAVIORAL of MUX51_GENERIC is
   for BEHAVIORAL
   end for;
end CFG_MUX51_GEN_BEHAVIORAL;
```

rca_generic.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use WORK.constants.all; -- libreria WORK user-defined

entity RCA_GENERIC is
   generic (N : integer);
   Port (  A: In std_logic_vector(N-1 downto 0);
      B: In std_logic_vector(N-1 downto 0);
      Ci:   In std_logic;
      S: Out  std_logic_vector(N-1 downto 0);
      Co:   Out   std_logic);
end RCA_GENERIC;

architecture STRUCTURAL of RCA_GENERIC is

  signal STMP : std_logic_vector(N-1 downto 0);
  signal CTMP : std_logic_vector(N downto 0);

  component FA
  Port ( A: In std_logic;
    B:   In std_logic;
    Ci:  In std_logic;
    S:   Out   std_logic;
```

```
    Co: Out  std_logic);
  end component;

begin

  CTMP(0) <= Ci;
  S <= STMP;
  Co <= CTMP(N);

  ADDER1: for I in 1 to N generate
    FAI : FA
     Port Map (A(I-1), B(I-1), CTMP(I-1), STMP(I-1), CTMP(I));
  end generate;

end STRUCTURAL;


configuration CFG_RCA_GENERIC_STRUCTURAL of RCA_GENERIC is
  for STRUCTURAL
    for ADDER1
      for all : FA
        use configuration WORK.CFG_FA_BEHAVIORAL;
      end for;
    end for;
  end for;
end CFG_RCA_GENERIC_STRUCTURAL;
```

fa.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use WORK.constants.all; -- libreria WORK user-defined

entity FA is
   Port (  A: In std_logic;
      B: In std_logic;
      Ci:  In std_logic;
      S: Out  std_logic;
      Co:  Out  std_logic);
end FA;

architecture BEHAVIORAL of FA is
```

```
begin

  S <= A xor B xor Ci;
  Co <= (A and B) or (B and Ci) or (A and Ci);
  -- Co <= (A and B) or (B and Ci) or (A and Ci);

end BEHAVIORAL;

configuration CFG_FA_BEHAVIORAL of FA is
  for BEHAVIORAL
  end for;
end CFG_FA_BEHAVIORAL;
```

tb.v

```
module tb; //Module is self-contained. There is no port list.

parameter N = 13;             //N = nb

wire CK, VIN, VOUT, RST_n, END_SIM;        //Signal equivalent in VHDL
wire [N-1:0] SAMPLE, H0, H1, H2, H3, H4, DOUT;

DATA_GENERATOR DATA_GEN(.CK(CK),      //Instantiation of DATA_GENERATOR
    component
         .VIN(VIN),
         .RST_n(RST_n),
         .END_SIM(END_SIM),
         .H0(H0),
         .H1(H1),
         .H2(H2),
         .H3(H3),
         .H4(H4),
         .SAMPLE(SAMPLE));

FIR FILTER( .DIN(SAMPLE),          //Instantiation of FIR component
            .VIN(VIN),
            .RST_n(RST_n),
            .CK(CK),
         .H0(H0),
         .H1(H1),
         .H2(H2),
         .H3(H3),
         .H4(H4),
```

```
        .DOUT(DOUT),
        .VOUT(VOUT));

SAVE_DATA SAVE( .CK(CK),           //Instantiation of SAVE_DATA component
        .RST_n(RST_n),
        .VIN(VIN),
        .DIN(SAMPLE),
        .VOUT(VOUT),
        .DOUT(DOUT));

endmodule //End Of Module tb
```

tb_power.v

```
module tb; //Module is self-contained. There is no port list.

parameter N = 13;             //N = nb

wire CK, VIN, VOUT, RST_n, END_SIM;        //Signal equivalent in VHDL
wire [N-1:0] SAMPLE, H0, H1, H2, H3, H4, DOUT;

initial begin
   $read_lib_saif("../saif/fast.saif");
   $set_gate_level_monitoring("on");
   $set_toggle_region(FILTER);
   $toggle_start;
end


always @ ( END_SIM ) begin
   if (END_SIM) begin
      $toggle_stop;
      $toggle_report("../saif/myfir_back.saif", 1.0e-9, "tb.FILTER");
   end
end



DATA_GENERATOR DATA_GEN(.CK(CK),      //Instantiation of DATA_GENERATOR
    component
        .VIN(VIN),
        .RST_n(RST_n),
        .END_SIM(END_SIM),
```

```
        .H0(H0),
        .H1(H1),
        .H2(H2),
        .H3(H3),
        .H4(H4),
        .SAMPLE(SAMPLE));

FIR FILTER( .DIN(SAMPLE),          //Instantiation of FIR component
           .VIN(VIN),
           .RST_n(RST_n),
           .CK(CK),
        .H0(H0),
        .H1(H1),
        .H2(H2),
        .H3(H3),
        .H4(H4),
        .DOUT(DOUT),
        .VOUT(VOUT));

SAVE_DATA SAVE( .CK(CK),           //Instantiation of SAVE_DATA component
        .RST_n(RST_n),
        .VIN(VIN),
        .DIN(SAMPLE),
        .VOUT(VOUT),
        .DOUT(DOUT));

endmodule //End Of Module tb
```

data_generator.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.all;
use ieee.std_logic_textio.all;

library std;
use std.textio.all;
use WORK.constants.all; -- libreria WORK user-defined


entity DATA_GENERATOR is
```

```
   Port (  CK : out std_logic;
      VIN   : out std_logic;
      RST_n : out std_logic;
      END_SIM : out std_logic;
      H0 : out std_logic_vector(nb-1 downto 0);
      H1 : out std_logic_vector(nb-1 downto 0);
      H2 : out std_logic_vector(nb-1 downto 0);
      H3 : out std_logic_vector(nb-1 downto 0);
      H4 : out std_logic_vector(nb-1 downto 0);
      SAMPLE  : out std_logic_vector(nb-1 downto 0));
end DATA_GENERATOR;


architecture STRUCTURAL of DATA_GENERATOR is

   constant Period: time := 1 ns;     -- Clock period (1 GHz)
   signal CK_i, CK_p : std_logic := '0';
   signal RESET,VIN_i,END_SIM_i: std_logic;


begin

   --Reset conditions
   RESET <= '0', '1' after Period;

   --Clock generation process
   process
   begin
        if (CK_i = 'U') then
             CK_i <= '0';
        else
             CK_i <= not(CK_i);
        end if;
        wait for Period/2;
   end process;

   CK_p <= CK_i and (not(END_SIM_i));  --Stop the clock when END_SIM_i
      (generated by the LFSR) is equal to '1'

   --Process to read input sample from txt file
   process (CK_p, RESET)
      file infile : text is in "inputs.txt";
```

```
    variable inline: line;        --buffer used to store the character to
        be written on the file
    variable counter : integer := 0;  --counter used to index samples
    variable in_sample : integer;
  begin -- process

      if RESET = '0' then                  -- asynchronous reset (active
          low)
      VIN_i <= '0';
      END_SIM_i <= '0';
      elsif CK_p'event and CK_p = '1' then     -- rising clock edge
      readline(infile, inline);           --reading a line from the file.
      read(inline, in_sample);      --put the line value into a variable
      SAMPLE <= conv_std_logic_vector(in_sample,nb); --assign the
          variable to a signal.
      counter := counter+1;
      VIN_i <= '1';
      if(counter = NUM_SAMPLES) then     --if 100 samples have been read
          stop the simulation
        END_SIM_i <= '1';
      else
           END_SIM_i <= '0';
      end if;
       end if;
  end process;

  CK <= CK_p;
  RST_n <= RESET;
  VIN <= VIN_i;
  END_SIM <= END_SIM_i;

  H0 <= conv_std_logic_vector(34,nb);
  H1 <= conv_std_logic_vector(893,nb);
  H2 <= conv_std_logic_vector(2241,nb);
  H3 <= conv_std_logic_vector(893,nb);
  H4 <= conv_std_logic_vector(34,nb);
end STRUCTURAL;
```

save_data.vhd

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

```
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;

library std;
use std.textio.all;
use WORK.constants.all; -- libreria WORK user-defined


entity SAVE_DATA is
   port (  CK    : in std_logic;
         RST_n   : in std_logic;
         VIN     : in std_logic;
         DIN     : in std_logic_vector(nb-1 downto 0);
         VOUT    : in std_logic;
         DOUT    : in std_logic_vector(nb-1 downto 0));
end SAVE_DATA;

architecture BEHAVIORAL of SAVE_DATA is

begin

   --Process for results file
   process (CK, RST_n)
      file fp_out : text open WRITE_MODE is "./results";
      variable buf_o: line;         --buffer used to store the character to
          be written on the file
      variable counter_o : integer := 0;  --counter used to index samples
   begin -- process

         if RST_n = '0' then              -- asynchronous reset (active low)
         null;
         elsif CK'event and CK = '1' then  -- rising clock edge
               if VOUT = '1' then
               write(buf_o, conv_integer(signed(DOUT))); --write OUTPUT
                  SAMPLE in the buffer
               writeline(fp_out, buf_o);     --write all the content of
                  buffer on the file
               end if;
          end if;
   end process;

end BEHAVIORAL;
```