Integrated Systems Architectures

Lab 1: design and implementation of a digital filter
General description

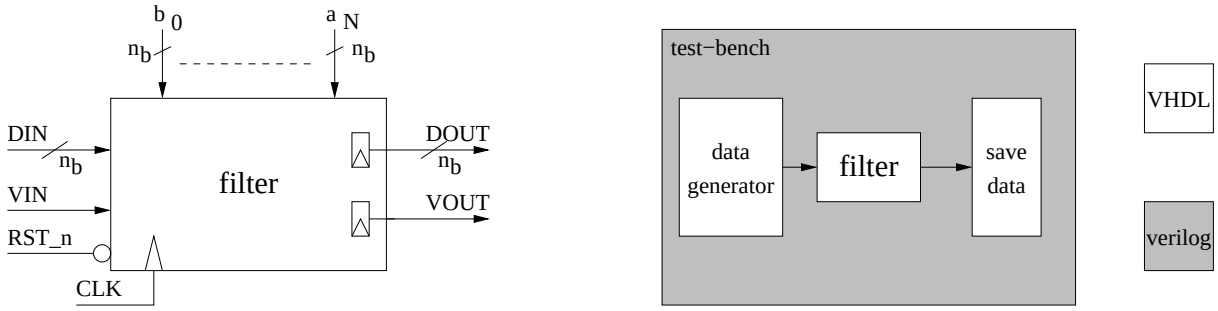

Figure 1

1 Reference model development

Design a digital filter with a cut-off frequency of 2 kHz. Set the sampling frequency to 10 kHz. The filter you have to design can be either a Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) filter depending on your group number, through parameter p, which is defined as follows: if the group number is odd then $p = 1$ and you have to design an FIR filter; otherwise $p = 0$ and you have to design an IIR filter.

1.1 Filter design and coefficient quantization using Matlab/Octave

You can use the *fir1* or *butter* functions available in Matlab or Octave to design the FIR or IIR filter respectively. Please note that Octave is an open source project and that Matlab is available for free for Polito students, check the page "MathWorks - Total Academic Headcount" on:
`https://www.areait.polito.it`.

On *Portale della didattica* you can download an example showing you the use of both the *fir1* and *butter* functions (*myfir_design.m* and *myiir_design.m*). These files require two parameters: i) the order of the filter, ii) the number of bits to represent the coefficients.

To set these two parameters use the following algorithm.

1. order (alphabetically) the surnames of the members of your group;
2. let x and y be the number of characters in the first two surnames, then obtain the order of the filter (N) and the number of bits (n_b) as follows:

$$N = 2^p \cdot [(x \bmod 2) + 1] + 6 \cdot p, \quad (1)$$
$$n_b = (y \bmod 7) + 8, \quad (2)$$

where p has been defined at the beginnig of Section 1.

Example (Bianchi, Rossi, ...) leads to $x = 7$ and $y = 5$, so $N_{p=0} = 2$, $N_{p=1} = 10$ and $n_b = 13$.

1.2 Testing the filter and fixed point implementation

1.2.1 First step: Matlab/Octave pseudo-fixed-point

Now, you have to test your filter on a signal. To this purpose on *Portale della didattica* you can download the scripts *my_fir_filter.m* and *my_iir_filter.m*, which create a signal made of two sinusoidal waves at two different frequencies (one falls in band and one out of the band). Then, each script i) calls the proper function (*myfir_design* or *myiir_design*), ii) applies the filter and iii) displays the results. Moreover, each script saves both the input and output samples as quantized data represented on n_b bits as integer values (*samples.txt* and *resultsm.txt*).

1.2.2 Second step: fixed-point C model

At this point, you have to write in C language a fixed point implementation of the filtering operation, namely:

$$y_i = \sum_j x_{i-j} \cdot b_j - \sum_k y_{i-k} \cdot a_k \tag{3}$$

where x_l, y_l, b_l and a_l are the input samples, output samples and filter coefficients respectively. There are several possibilities to implement (3), but **you must use direct form for FIR filters and direct form II (canonical direct form) for IIR filters.** Please note that on *Portale della didattica* you can download two examples (*myfilter.c* and *myfilterII.c*) where an IIR filter is implemented as fixed point direct form I (*myfilter.c*) and direct form II (*myfilterII.c*). Direct form for an FIR filter can be straightforwardly derived from *myfilter.c*. These programs i) help you in evaluating the performance of the fixed point implementation with respect to the Matlab/Octave one and ii) are the reference model you will use to develop, debug and test the digital filter as an hardware architecture (see Section 2). **Note: the notation used in this document, in the Matlab/Octave files and C fixed point model is the one used by Matlab**, i.e. b_i are the coefficients of the moving-average part and a_i are the coefficients of the auto-regressive part, as in (3).

2 VLSI implementation

2.1 Starting architecture development

Develop in VHDL the architecture of the filter you designed according to the requirements specified in Section 1. **Note: do not choose the architecture of adders and multipliers at this time. Use '*' and '+' instead.** The filter interface is shown in Fig. 1 (a): the samples (*DIN*) enter one each clock cycle with a validation signal (*VIN*). When *VIN*='1' a new sample is loaded into the architecture The output *DOUT* contains the result of the filtering (y) and *VOUT* is a validation signal: *VOUT*='1' when *DOUT* is ready. **Note:** all the inputs and the outputs of the filter must be loaded/produced by registers. All the data are represented as 2-complement normalized-fixed-point values: the weight of the most significant bit (MSB) is -2^0, the weight of the least significant bit (LSB) is 2^{-n_b+1}. **Note:** use `std_logic_vector`, `signed` or `unsigned` for fixed point data representation.

Finally, prepare a hierarchical test-bench, where the signal generation is implemented in VHDL, whereas the top of the hierarchy is a verilog file (see Fig. 1 (b)). From *Portale della didattica* you can download an example of the files required to create a mixed VHDL/verilog testbench.

Note: To ease the design flow we suggest to avoid the use of the *generic* statement in the **top** entity of the filter. You can use the *generic* for the modules which are inside the design. If you want the top of the hierarchy to be parametric, then please use a package instead.

2.2 Simulation

Simulate and verify with Modelsim the system you described. In particular, you have to compare the results produced by your VHDL with the ones you obtained with your fixed point model

developed in C language (see Section 1). **Note: the results produced by the VHDL must be equal to the ones obtained in C.** *Suggestion* structure your design environment as follows. Create a working directory (e.g. `lab1`). In `lab1` create:

- `src` as the directory containing the VHDL file/files of the filter;
- `tb` as the directory containing the VHDL and the verilog of the test-bench;
- `sim` as the directory containing the project files created by Modelsim and your simulation scripts (if any).

Note: the testbench must show that the circuit is working correctly even when *VIN* goes to zero, as shown in Fig. 2.

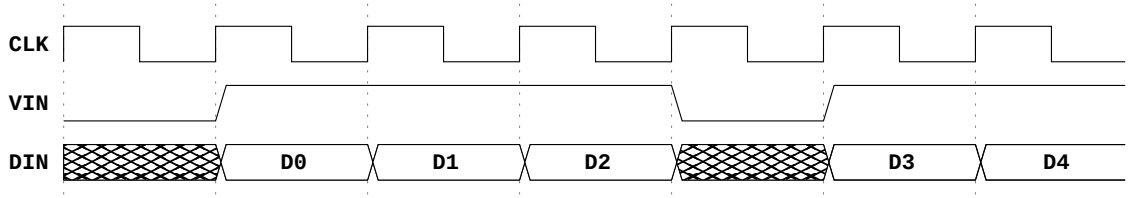

Figure 2

2.3 Implementation

2.3.1 Logic synthesis

Synthesize the filter with Synopsys Design Compiler (see the documentation on *Portale della didattica*) according to the following specifications.

1. Find the maximum clock frequency your design achieves (f_M) and the corresponding area.
2. Set the clock frequency to $f_M/4$, and verify the netlist via simulation. **The output results must be the same as the ones obtained with the VHDL.**
3. Obtain the switching activity (for the $f_{clk} = f_M/4$ case) to estimate the power consumption of your design (see the documentation on *Portale della didattica*).

Suggestion structure your design environment as follows. Create in `lab1`:

- `syn` as the directory containing the synthesis scripts and the working files produced by the logic synthesizer;
- `netlist` as the directory containing the output of the logic synthesizer.

2.3.2 Place & Route

Place and route the filter with Cadence SOC Innovus (see the documentation on *Portale della didattica*) according to the following specifications.

1. Set $f_{clk} = f_M/4$ and find the area of your design.
2. Verify the netlist via simulation. **The output results must be the same as the ones obtained with the VHDL.**
3. Obtain the switching activity to estimate the power consumption of your design (see the documentation on *Portale della didattica*).

Suggestion structure your design environment as follows. Create in `lab1`:

- `innovus` as the directory containing the place and route files and results.

2.4 Advanced architecture development

Improve your architecture by applying the following techniques:

- L-unfolding and pipelining for FIR filters,
- J-look-ahead for IIR filters,

where $L = 3$ and $J = 1$. Choose the number of pipeline stages to achieve the maximum possible throughput **without adding pipeline to the arithmetic blocks**, namely draw your architecture on paper, put registers between the arithmetic blocks and make experiments with Design Compiler. For the new architecture repeat all the steps detailed in Sections 2.2 and 2.3.

Note for IIR filters: derive the J-look-ahead architecture from the direct form II representation. Moreover, with the look-ahead technique 'new' coefficients, such as $b_0 \cdot a_1$, are required. As a consequence, precision and bit-width could change with respect to the first architecture.