Politecnico di Torino

Integrated Systems Architecture

A.Y. 2025/2026

Laboratory 1

Design and implementation of a digital filter

Delivery instructions

A template for the homework report is available on the *Portale della Didattica*. Students are required to follow the **provided structure** and ensure that the report is **self-contained**, **complete**, **clear**, and **accurate**. **Diagrams**, **graphs**, and **figures** that facilitate the understanding of the design must be included in the report.

Each project prepared for this course must be submitted by uploading it to the *Portale della Didattica*, section *Elaborati*. Only one member of the group must submit the deliverable.

The deliverable must consist of a `.zip` archive containing all the **source files**, the **scripts**, and the **main report**. The `.zip` must be called `isa<nn>_lab1.zip`, where `<nn>` represents the group number (for example, group 8 should name the file `isa08_lab1.zip`).

Work folders, temporary files, and intermediate files must not be included. Only the **input** and **output files** are required, together with **clear instructions** on how to reproduce the reported results using the uploaded material.

Delivery deadline: November 23 at 23:59

1 Reference model development

Design a Finite Impulse Response (FIR) filter with a cut-off frequency of 2 kHz. Set the sampling frequency to 10 kHz.

1.1 Filter design and coefficient quantization using Matlab/Octave

You can use the *fir1* function available in Matlab (or Octave) to design the FIR. 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 the *fir1* function (*myfir_design.m*). These files require two parameters:

1. The order of the filter;
2. The number of bits to represent the coefficients.

To set these two parameters, use the following algorithm:

1. Reverse-order (alphabetically) the surnames of the members of your group: e.g. Palmer, Lake, Emerson;
2. Let x and y be the number of characters in the first two reverse-ordered surnames (e.g. $x = 6$ for Palmer and $y = 4$ for Lake), then obtain the order of the filter (N) and the number of bits (n_b) as follows:

$$N = 2 \cdot (x \bmod 2) + 8, \quad (1)$$
$$n_b = (y \bmod 7) + 9. \quad (2)$$

Example (Palmer, Lake, ...) leads to $x = 6$ and $y = 4$, so $N = 8$ 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 script *my_fir_filter.m*, which creates a signal made of two sinusoidal waves at two different frequencies (one falls in-band and one out-of-band). Then, the script:

1. Calls the proper function (*myfir_design*);
2. Applies the filter;
3. Displays the results.

Moreover, the script saves both the input and output samples as quantized data represented on $n_b - 1$ bits for the fractional part and one bit for the integer part, as integer values (*samples.txt* and *resultsm.txt*). Check that $|b_j| < 1$ for every i so that the quantized values (*bq*) and the corresponding integer values (*bi*) are correctly represented.

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:

$$y_i = \sum_j x_{i-j} \cdot b_j, \quad (3)$$

where x_l, y_l and b_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.** Direct form for an FIR filter can be straightforwardly derived from *myfilter.c*.

This program:

1. Helps you in evaluating the performance of the fixed point implementation with respect to the Matlab/Octave one;

2. Is the reference model you will use to develop, debug and test the digital filter as an hardware architecture (see Section 2).

Please note that the example already reduces the number of bits after each multiplication, to reduce the bitwidth (and so the area) of your filter. **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, as in (3), see the help of the Matlab/Octave *filter* function for details.

Then, evaluate the Total Harmonic Distortion (THD) of the result y of your fixed point implementation. You can use the *thd* function in Matlab to obtain it. Choose the number of bits to discard after each multiplication ($SHAMT$) to minimize the area of your architecture with a maximum THD of -40dB.

Assignment

1. Design the filter with Matlab/Octave and represent the coefficients with the number of bits which have been assigned to your group. Put in the report a figure showing the frequency response for both floating point and fixed point coefficients. Comment it.
2. Develop the fixed point model as a C program and reduce the precision of the multiplications reaching a maximum THD of -40 dB.
3. Compare and comment the results.

2 VLSI Implementation

2.1 Starting Architecture Development

Draw the block diagram of the architecture and the corresponding timing diagram, following the requirements specified in Section 1. For timing diagrams, use `https://wavedrom.com`. Implement the designed filter architecture in HDL, ensuring that all flip-flops and registers include an *enable* signal. Either VHDL or SystemVerilog can be used; however, SystemVerilog is strongly recommended, especially for beginners, as it will be employed in subsequent laboratory activities.

Note: At this stage, do not choose the specific architecture of adders and multipliers. Use the operators '*' and '+' instead.

The filter interface is illustrated in Fig. 1(a): the input samples (DIN) arrive one per clock cycle, accompanied by a validation signal (VIN). When VIN = '1', a new sample is valid and can be loaded into the architecture. The output $DOUT$ contains the filtering result (y), and $VOUT$ is the corresponding validation signal. $VOUT$ = '1' indicates that $DOUT$ is valid.

Note: All filter inputs and outputs must be stored in registers.

All data are represented as 2's-complement normalized fixed-point values: the most significant bit (MSB) has weight -2^0, and the least significant bit (LSB) has weight 2^{-n_b+1}.

Note: For VHDL, use `std_logic_vector`, `signed`, or `unsigned` for fixed-point representation. For SystemVerilog, instead, use `logic` or `logic signed`.

Finally, prepare a hierarchical testbench where the data generator (*data_maker.vhd*), the check data (*data_sink.vhd*) and the clock generator (*clk_gen.vhd*) are implemented in VHDL, the filter is described in VHDL or SystemVerilog and the top-level entity is described in Verilog (see Fig. 1(b)). An example of the required files for creating a mixed VHDL/SystemVerilog testbench is available from the *Portale della didattica*.

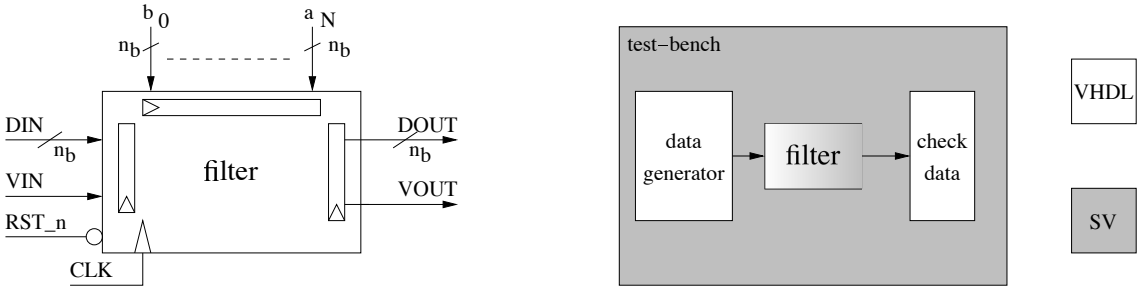
Note: To simplify the design flow, avoid using the *generic* statement and user-defined types in the **top** entity of the filter. Generics may be used for internal modules. If the top-level hierarchy must be parametric, use a package instead.

2.2 Simulation

Simulate and verify the system using ModelSim/QuestaSim (see *Simulation with Siemens QuestaSim* in the file *design_flow_documents.pdf* on *Portale della didattica*), comparing the HDL results with those from the fixed-point C model (Section 1).

Note: The outputs must be *identical* between the two models.

Suggested directory structure:



(a) (b)

Figure 1

- `src` — HDL source files of the filter.
- `tb` — SystemVerilog files for the testbench.
- `sim` — ModelSim/QuestaSim project files, input data files (e.g., `samples.txt`), and simulation scripts.

Note: The testbench must demonstrate correct behavior even when *VIN* goes low, as shown in Fig. 2.

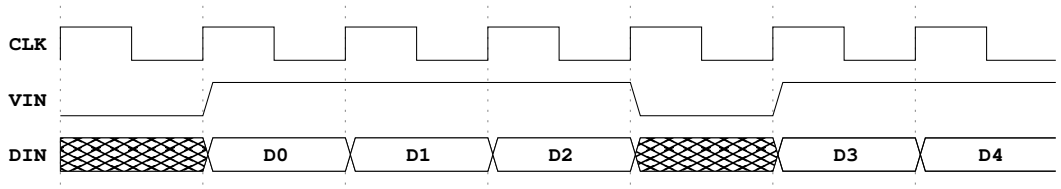


Figure 2

Assignment

1. Implement the filter architecture with the appropriate number of registers, adders, multipliers, and control logic. Report the block diagram and the timing diagram demonstrating compliance with the specifications (use `https://wavedrom.com`).
2. Develop the HDL model and verify it against the fixed-point C model with a proper testbench. The outputs of the HDL model must match exactly those of the C model, including cases where *VIN* transitions from '1' to '0' and back to '1'. Report a snapshot of this behaviour. Ensure that the interface of your architecture matches exactly the one in Fig. 1(a), as an automated tool will be used to check the design.
3. Measure the total time from the first rising clock edge with $VOUT$ = '1' to the last such event.

2.3 Implementation

2.3.1 Logic Synthesis

Synthesize the filter with Synopsys Design Compiler according to the following steps (see *Logic synthesis with Synopsys Design Compiler* in the file *design_flow_documents.pdf* on *Portale della didattica*):

1. Inspect messages generated during *analyze*, *elaborate*, and *compile* phases, addressing any issues such as incomplete sensitivity lists, inferred latches, or combinational loops.

2. Determine the maximum achievable clock frequency f_M (minimum period with zero slack) and record the corresponding area.

3. Verify the netlist via simulation at $f_{\text{clk}} = f_M$, ensuring no setup/hold violations. Adjust the testbench if necessary. Measure the latency from the first to the last valid output. Outputs must match the original HDL results.

4. Obtain switching activity data to estimate power consumption (see *Switching-activity-based power consumption estimation* in the file *design_flow_documents.pdf* on *Portale della didattica*).

5. Re-synthesize the design with $f_{\text{clk}} = f_M/2$. Record the corresponding area and re-estimate power consumption.

6. Synthesize the design with clock gating enabled. Thus, you need to evaluate a new clock frequency f_M. Record the corresponding area and re-estimate power consumption.

7. Re-synthesize the clock-gated design at $f_{\text{clk}} = f_M/2$. Record the corresponding area and re-estimate power consumption.

Suggested directory structure:

- `syn` — synthesis scripts and working files.
- `netlist` — output netlists from synthesis.

Reminder: Delete and recreate the *work* directory before each synthesis run.

Assignment

1. Perform logic synthesis to determine f_M, and report the maximum frequency, total cell area, and estimated power consumption.
2. Re-synthesize the design with $f_{\text{clk}} = f_M/2$ and record the results.
3. Repeat both syntheses steps with clock gating enabled.
4. Compare and comment on the results, highlighting differences in performance, area, and power consumption.

2.3.2 Place & Route

Perform Place & Route of the clock-gated design using Cadence Innovus with the following steps (see *Place and route with Cadence Innovus* in the file *design_flow_documents.pdf* on *Portale della didattica*):

1. Review Innovus messages at each stage (setup, floorplanning, placement, routing, etc.).
2. Set $f_{\text{clk}} = f_M/2$ in the `.sdc` file and record the design area.
3. Verify the post-layout netlist via simulation at $f_{\text{clk}} = f_M/2$, ensuring results match the original HDL.
4. Extract switching activity (at $f_{\text{clk}} = f_M/2$) for power estimation (see *Post place and route simulation and switching-activity-based power consumption estimation* in the file *design_flow_documents.pdf* on *Portale della didattica*).

Suggested directory structure:

- `innovus` — Place & Route files and results.

Assignment

1. Place & Route the design at $f_{\text{clk}} = f_M/2$, record the area, verify operation, measure simulation time, and estimate power consumption.
2. Compare the result to the one obtained after the synthesis at $f_{\text{clk}} = f_M/2$ with clock gating and comment it.

3 Advanced architecture development

You need now to improve your architecture. The technique you need to apply depends on your group number:

- **3-unfolding**, if the group number is even;
- **pipelining**, if the group number is odd.

For the new architecture, repeat all the steps detailed in Sections 2.1, 2.2 and 2.3.

Note for 3-unfolding: the final architecture must have the interface shown in Fig. 3, you can use the *unfolded* version of the testbench files available on *Portale della didattica* to test the unfolded architecture.

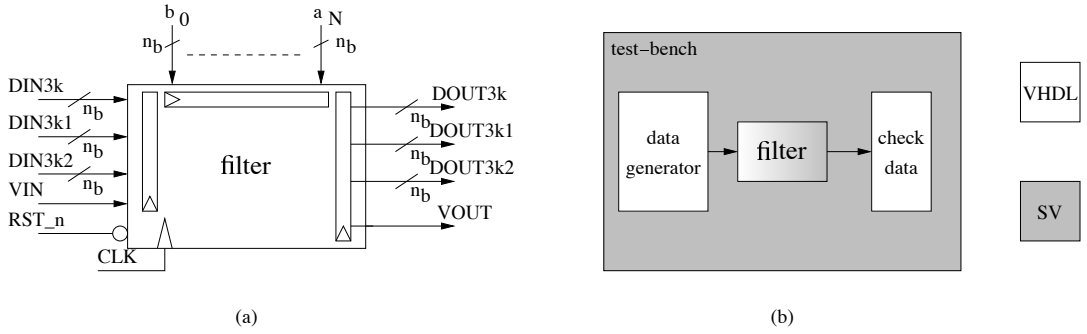


Figure 3

Note for pipelining: choose the number of pipeline stages to achieve the maximum possible throughput **without adding pipeline to the arithmetic blocks**.

Assignment

1. Repeat all the steps in Section 2 with the advanced architecture.