# Integrated System Architecture
# Lab session 1 report

Marco Andorno (247222)
Michele Caon (253027)
Matteo Perotti (251453)
Giuseppe Sarda (xxxxxx)

December 20, 2018

This report along with all the source files, scripts, reports and diagrams for the project can be found on GitHub at `https://github.com/mksoc/ISA-filter-design`.

## Contents

## 1 Filter design

Following the rules given in the assignment, the main specifications were derived:

- Filter type: IIR
- Filter order: $N = 2$
- Cutoff frequency: $f_c = 2\,\text{kHz}$
- Sampling frequency: $f_s = 10\,\text{kHz}$
- Data parallelism: $n_b = 12$

Then, using the provided example MATLAB script, the filter coefficients were found by means of the `butter` function. Real coefficients are then quantized as fixed point fractional number in the format $Q1.(n_b - 1)$ ($Q1.11$ in our case) and expressed as integers on $n_b$ bits for the future C model and hardware filter. We will discover later that some care has to be taken when performing operations on the integer representation of fixed point numbers.

Quantization is performed by truncation (`floor` function), so that the maximum error is equal to:

$$\varepsilon_{max} = 2^{-(n_b-1)} = 2^{-11} = 0.049\% \tag{1}$$

We accept that this error could be reduced by rounding and that truncation introduces a negative bias by approximating always towards $-\infty$, because on the other hand truncation is much easier to implement in hardware, where it just represent an arithmetic shift.

The resulting difference equation is in the end:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2]$$

where

$b_0 = 0.20654 = 423$

$b_1 = 0.41309 = 846$

$b_2 = 0.20654 = 423$

$a_1 = -0.36963 = -757$

$a_2 = 0.19580 = 401$

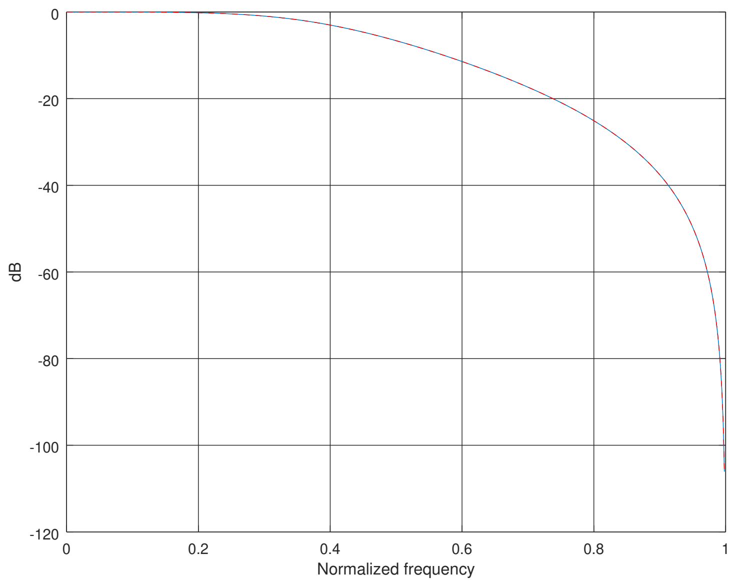


Figure 1: Bode plot of the filter frequency response

Figure 1 shows the transfer function of the filter computed from both the original coefficients and the quantized ones. The two curves cannot be told apart because the error is too small ($5.42 \cdot 10^{-7}$ in the worst case).

Figure 2 on the other hand shows the time domain waveforms of an input signal $x_1(t)$, in blue, containing two frequency components one in band and one out of band, and the corresponding output $x_2(t)$, in red, where only the in-band component survives.

## 2 Fixed-point C model

The next step consists of writing a software model of the filter in C, which mirrors the behavior of the hardware architecture to be designed next. In this regard, the main difference between the MATLAB model and this one is that the former uses quantized coefficients but performs the internal computation using the maximum precision allowed by the machine, while the latter performs computation always resorting to the original fixed parallelism of data (12 bits here).

The development of this software model started with the example provided in the assignment, tailored to our specifications. A Python script was developed and used to compare the results file of the two models. As expected, the comparison shows that the two models differ at most of one unity, that is the previously computed $\varepsilon_{max}$ (1) in fractional form. Furthermore, results from the MATLAB model, when different, are always greater than the results from the C model, as the latter performs multiple truncations (rounding towards $-\infty$) in its computations.

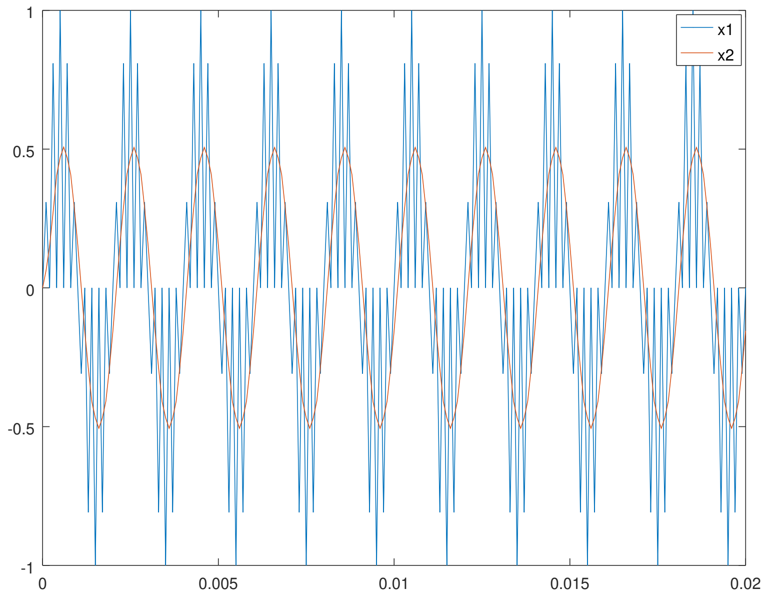


Figure 2: Input and output waveforms

## 3 Base architecture design

The development of the first architecture began with the definition of the Data Flow Graph of the filter, shown in figure 3, from which the number of hardware resources needed can be derived (i.e. two delay elements, four adders and five multipliers).

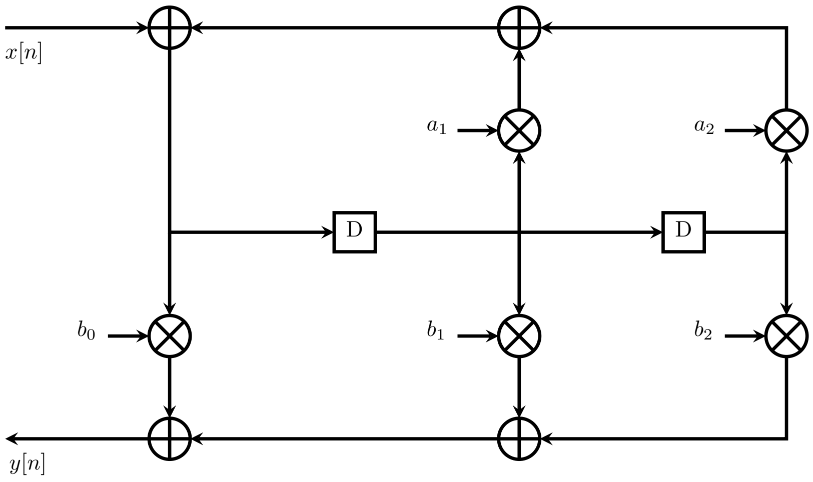


Figure 3: DFG of the filter

### 3.1 Datapath

From the generic DFG, a full datapath architecture was designed, shown in figure 4. Compared to the simple DFG of figure 3, this datapath explicitly shows all signal names used throughout the design and their parallelism, along with additional interface registers required by the specifications. The following paragraphs detail the key points of the design.

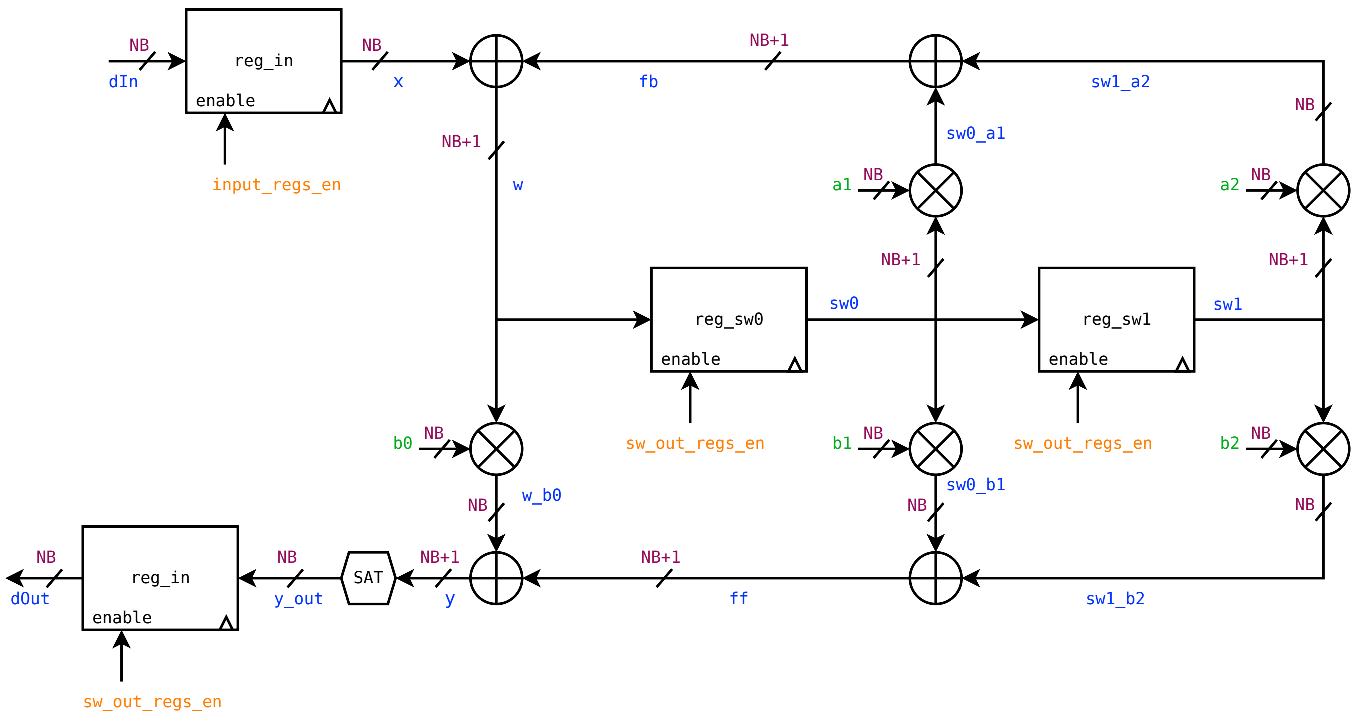


Figure 4: Datapath

### 3.1.1 Color legend

All datapath schemes used in this report follow the same color coding:

- Blue signals are data signals
- Green signals are external coefficients
- Purple labels define the parallelism of the corresponding signal
- Orange signals are control signals coming from the Control Unit

### 3.1.2 Registers

In addition to the two core delay elements of the filter DFG, the complete datapath includes also border registers for each piece of data coming in or out of the architecture, specifically the input and output sample stream and all filter coefficients. Input registers for the latter are not actually shown in figure 4 to avoid too much visual clutter, but assume that each coefficient in green is actually the output of a register.

Each register has implicit clock and asynchronous reset signals, as well as individual enable signals. No synchronous clear is provided as it is of no use for the purposes of this design.

### 3.1.3 Arithmetic operators

As per specifications, arithmetic operators at this stage of the design are described as behavioral operators in VHDL. While no rounding is requested after addition, each multiplication must truncate back to the original number of bits, a VHDL package was written, containing a function called `multiplyAndRound` which performs potential sign extension of the operands, behavioral multiplication and truncation. In the same package, some constants and custom data types to make the design parametric were defined as well.

Addition is more easily implemented as simply the behavioral operator '+', but the fact that addition can overflow and need an additional bit on the output posed the need for some considerations about internal parallelism. By performing an analysis on the specific values of filter coefficients and the range of data stream values, the conclusion was that the maximum internal parallelism that this specific architecture can need is:

$$n_b + 1 = 13\,\text{bit}$$

Contrary to the initial expectations however, the output value $y$ was found to need $(n_b + 1)$ bits as well because of occasional overflow, while in the beginning it was expected to stay in the $n_b$ range. Overflow on the output would of course cause very large errors in the result because of wrapping. So it was decided to implement a saturation approach which limits the error instead, by taking the maximum or minimum value allowed on $n_b$ bits if the actual results is too large or too small to be represented correctly.

This decision led to the need of modifying the fixed-point C model as well, to provide the same results of the VHDL description to compare during simulation and testing.

## 3.2 Control unit

Given the simplicity of the design, it was initially thought that no explicit control unit would be needed. While it is actually possible to avoid it completely, some considerations about the ease of extending the design with subsequent transformations and pipelining (section **??**) as well as the preference for modularity and clarity led to the decision to implement a simple control unit, which FSM is shown in figure 5.

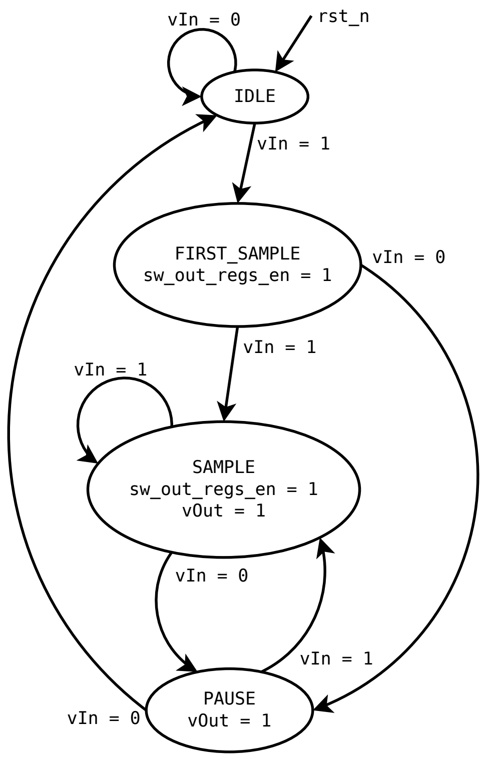


Figure 5: Filter control unit FSM

This control unit satisfies the specifications about the generation of the signal `vOut` to notify the presence of new data at the output and the need to take into account possible pauses during the input data stream. In practice, figure 6 shows the timing diagram that this FSM implements.

## 3.3 Simulation

All simulations were performed using ModelSim both on our local machines and on the remote server, by means of a mixed-language testbench partly already provided in the course material.

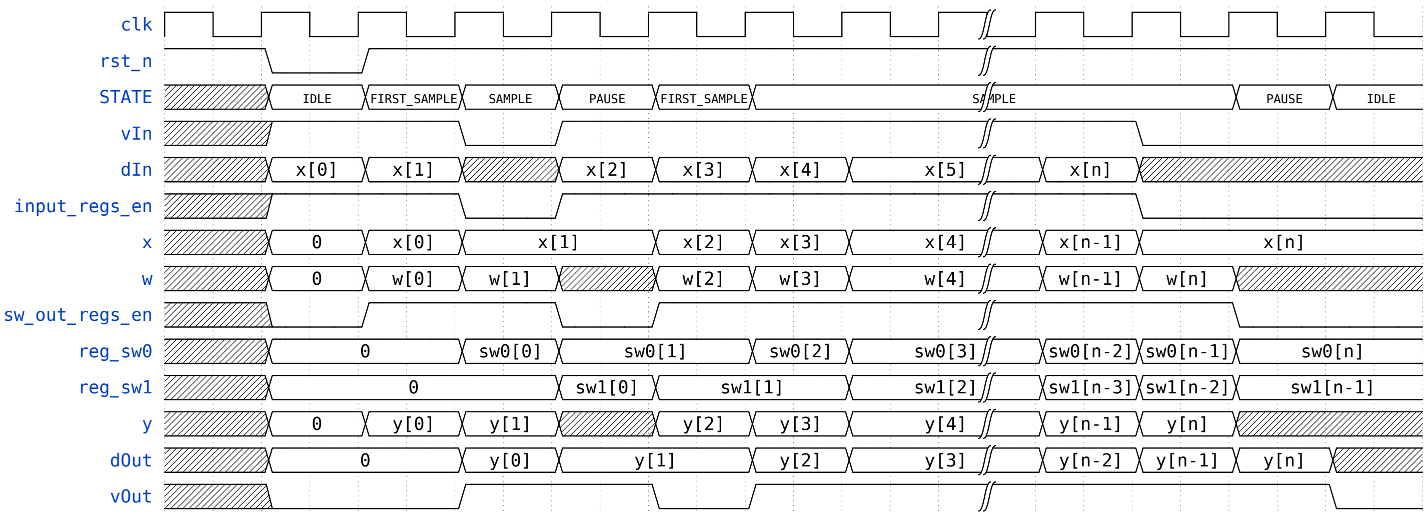


Figure 6: Timing diagram

The procedure and tools described in the following sections apply to the simulation of all the RTL description as well as the netlists generated by other tools used for synthesis or place and route.

Each design or netlist has been simulated extensively multiple times using up to 100000 samples generated randomly or with special values, which would normally take around 20 minutes.

Regarding the simulation of the RTL description, no particular problems arose apart from the usual initial debugging and in the end the design worked as expected.

### 3.3.1 Testbench

The testbench is made up of several independent modules:

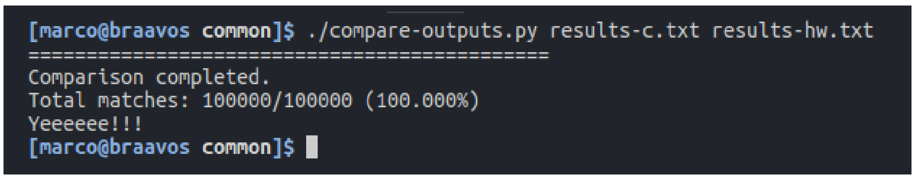
- A *clock generator* in charge of generating the initial reset signal and a periodic clock of specified period until an external `end_sim` signal arrives to notify the end of the simulation. This is useful to automatically stop the simulation process when all input data has been processed instead of fixing a specific time span a priori.
- A *data maker* which reads input data from a text file, converts it to a suitable format (`signed` in this case) and feeds it to the hardware description of the filter. It also allows to insert pauses in between samples to verify that such situation is handled correctly by the design. The number of pauses can be hard-coded or random, according to the value of a constant defined in the package.
- A *data sink* to read output data stream from the hardware block and write it to another text file.
- A top module written in Verilog (instead of VHDL as the previous ones) that simply connects the instance of the filter and all other testbench modules together. The need to describe this block in Verilog arises from the fact that synthesis and place and route tools generate Verilog netlists and are more comfortable using Verilog files.

### 3.3.2 Scripts

In order to try to automate as much as possible the simulation process, a number of scripts were developed, which can be found in the GitHub repository or in the material provided:

- `samples-generator.py` to generate a user-specified number of samples either random or special (meaning extremes and zero only).
- `auto-scp.sh` to automate the copy of source files and input samples on the server and results and netlists from the server.

- `compare-results.py` which compares the results of the simulation against the ones produced by the C model (figure 7).
- `auto-simulate.sh` which allows, by means of defining environmental variables, to choose whether to simulate locally or remotely and which design to simulate (RTL description, post-synthesis netlist or post-place and route netlist) and then performs several steps to automatically carry out the simulation, namely:
  - Generates samples using `samples-generator.py`.
  - Runs the C model on these samples.
  - Connects to the server (if the user has chose to run the simulation remotely) and copies the samples file.
  - Runs ModelSim CLI to perform the simulation.
  - Retrieves output files and compare the results.



```
[marco@braavos common]$ ./compare-outputs.py results-c.txt results-hw.txt
==============================================
Comparison completed.
Total matches: 100000/100000 (100.000%)
Yeeeeee!!!
[marco@braavos common]$ 
```

Figure 7: Successful comparison