Audio DSP

Guillaume W. Bres

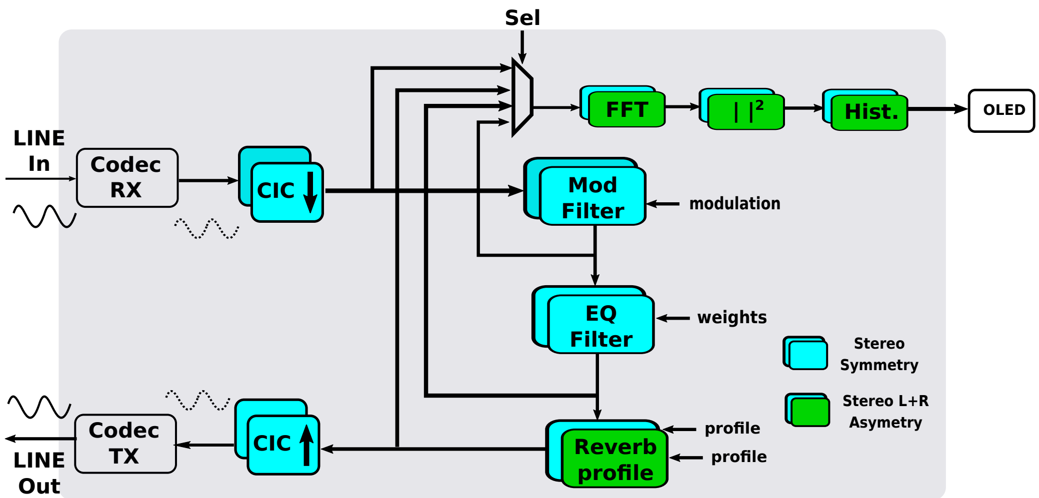
July 19, 2020

This project is an application of basic signal processing on audio signal, using zc702/zc706 development platforms.

Contents

1 System

1.1 Bloc design



Bloc design

All processing is real time and synchronous to the input stream. The initial sample rate is 48 MHz, making the input data rate 1.152 Gb/s. This is way more more than needed. In this setup we would need gigantic FFT to reach a decent resolution. We would also have to implement enormous filters to obtain a meaningful output.

We use a CIC decimation filter with R = 128 to reduce the sample rate to 24 kHz (a little more than needed). This reduces the bit rate to 576 kB/s.

One modulation can be applied on the signal, as well as one EQ preset made of 16 subbands and one Reverb/room profile can be emulated. These are all implemented & controlled in real time from the user interface. For each step refer to its own documentation down below.

The modulation & EQ will be applied using an FIR type of implemntation. The Reverb profile will be implemented by a 10 tap IIR type of filter.

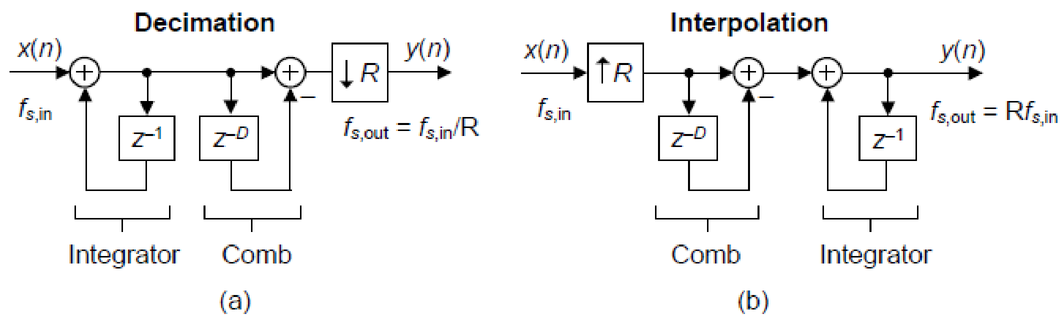
A power spectral analysis is performed in real time, either on the direct input line or the final output line. Selection is done in real time using the User Interface. Refer to following section for further details. The CIC decimator gives us a 187 Hz resolution in the spectral analysis. The result is displayed in real time in the form of a histogram, on the 'OLed' display.

By selecting either the direct input line or the output line to feed the FFT, we can choose to see the effect of the current filters on the signal in spectral domain.

A final interpolation is obviously needed to come back to the initial 1.152 Gb/s rate.

1.2 CIC filters

CIC filters stand for Cascaode of Integrator and Comb Filters:



They are defined by 3 parameters:

- R: decimation/interpolation factor
- M: time delay, can either be 1 or 2 cycle
- N: number of both integration & comb stages

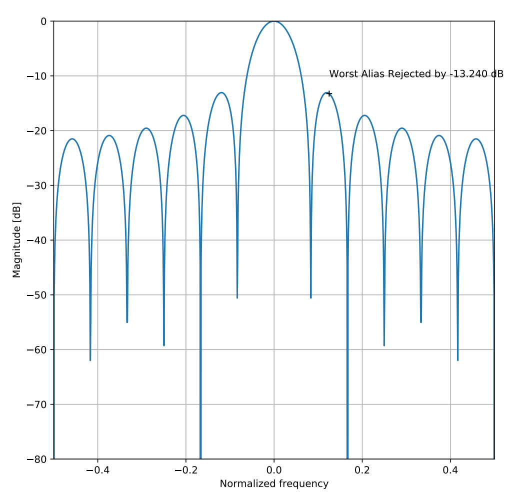
An integrator is defined by $H(z) = \frac{1}{1-z^{-1}}$. A comb filter is defined by $H(z) = 1 - z^{-M}$.

The total CIC filter is therefore defined by

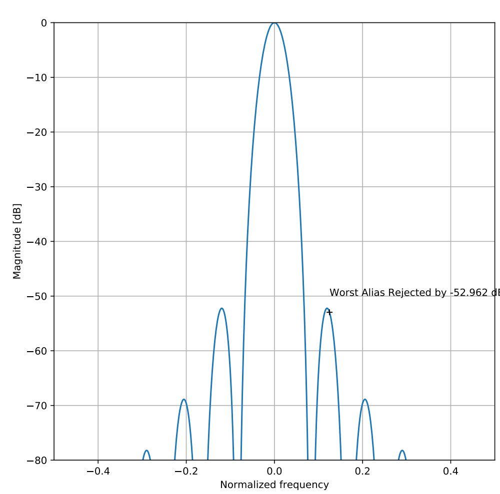
$$H(z) = \left| \frac{1 - z^{-M}}{1 - z^{-1}} \right|^N \tag{1}$$

The total filter gain is $(RM)^N$, hence the total bit growth in our implementation will be $\left[log_2\left(RM^N\right)\right]_{\text{ceil}}$. It is very important to take into account the bit growth, at least in the itegrators. Ideally itegrators should implement a 2's complement format with saturation, this is how our IP operates.

It is possible to approximate the magnitude response of a CIC filter to $H(\nu) = \left|\frac{\sin(RM\pi\nu)}{RM\sin(\pi\nu)}\right|^N$ $\forall\nu$ normalized frequency. Hence the magnitude response has a $\frac{\sin(\nu)}{\nu}$ shape:



`$git/python/dsp/cic.py R=12 N=2`
the worst alias is rejected by -13 dB



`$git/python/dsp/cic.py R=12 N=8`
increasing the number of stages to 8 improves the worst alias rejection to -53 dB

The worst alias is encountered at the peak of the 2nd lobe of the $\frac{\sin(\nu)}{\nu}$ shape, located at frequency $\nu = \frac{3}{2MR}$

Because of this $f(\nu) = \frac{sin(\nu)}{\nu}$ shape, we completely lose the flatness within the system bandwidth. The transition bandwidth is also very slow. To compensate for the flatness loss and to reduce the transition, we generally use a compensation filter (in the form of an FIR filter) after the CIC decimation filter and prior the CIC interpolation filter.

In our system we will fix R=128 to reduce the sample rate to 24 kHz and making our application possible. We will see in the next section how to design the compensation filter for this given CIC filter.

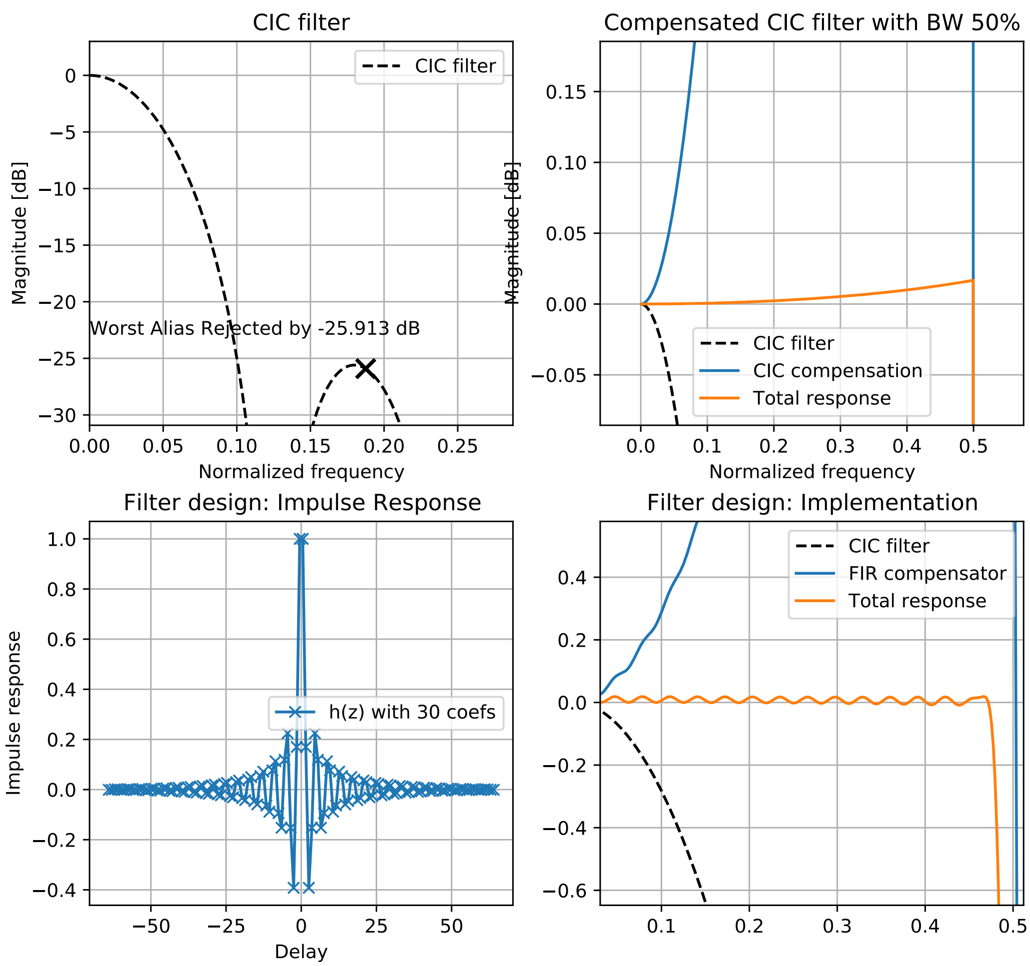
1.3 CIC compensation filter

The compensation filter response is the exact inverse of the CIC filter response $G(\nu) = \frac{1}{H(\nu)}$ within the new established band, ranging from 0 to $\frac{f_s}{R}$, $\frac{f_s}{R}$ being the first null of the $\frac{sin(\nu)}{\nu}$ response.

The CIC compensation filter is designed using the python script `$git/dsp/cic-compensator.py`.

```
$git/dsp/cic-compensator.py R=4 N=12 BW=30
$git/dsp/cic-compensator.py R=8 N=4 BW=45 ncoef=128
```

Use BW (in %) to set the pass band edge location within the new established Nyquist band. The ncoef parameter allows to compare different implementation of the FIR filter.



The CIC filter compensator designer designs a CIC filter to compensate for the CIC filtering stage. One can compare the effect of the BW cut off frequency (design the new user bandwidth) and the ripple effect, for different FIR filters implementation.

Upper left: theoretical CIC filter as described in previous section.

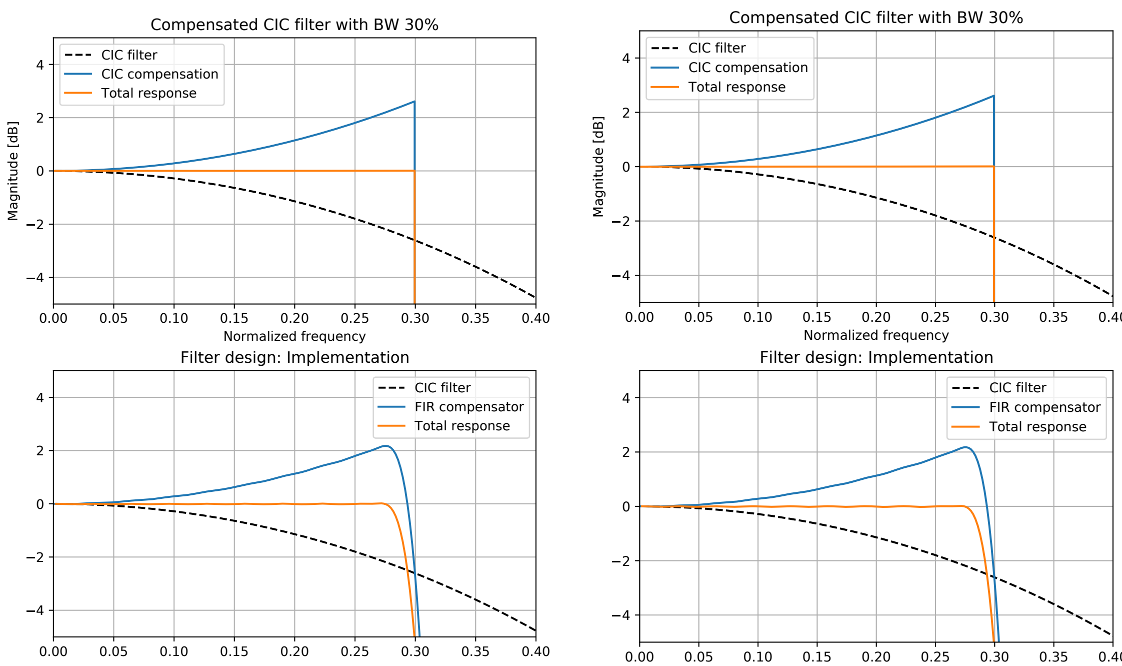
Upper right: theoretical CIC filter versus its optimum compensation, for a given BW parameter. Total theoretical response is CIC+FIR.

Below - left: designed Impulse Response to be implemented, for given BW and ncoef parameters.

Below - right: filter implementation: theoretical CIC filter compensated by implemented FIR filter. Total implemented response is CIC+FIR.

CIC filter compensation will be implemented in both the decimation & the interpolation filter, using Xilinx's optimized FIR filter IP core.

The CIC decimation is very important to our system, without heavily downsampling the input signal (we chose R=128), the displayed FFT resolution would be 37.5 MHz, which is totaly unusable for audio data. The Modulation Filter & EQ filter would require so many bands to be computated that it would also be impossible to implement them in the system.



Direct cut off is impossible to reach in the implemented form. This is a 16 tap FIR filter.

Increasing to 128 taps dramatically reduces the distortion within the band.

1.3.1 CIC interpolation filter

Using previously designed compensated CIC decimator, we reduce the input rate by a factor of 128. We need to interpolate by a factor of 128 prior outputing the signal to the DAC.

1.4 Equalization filter

The EQ filter allows the user to modify the signal's frequency response, in real time.

The frequency response is divided into 16 bands, considering our internal sample rate of $f_s = \frac{48e8}{128} = 24$ kHz, each band is 1.5 kHz wide.

The user interface allows the user to define the weighting of each band, so define the custom frequency response $H(\nu)$.

The GUI transforms $H(\nu)$ into $h(z)$ to be applied in the FIR filter:

$$h(z) = FFT^{-1}\left(H(z)\right) = Bilienear(H(\nu)) \tag{2}$$

1.5 FFT Histogram

The Histogram IP core converts the power spectrum from the magnitude IP core to a histogram to be displayed on the OLED.

It is a simple mapping of the resulting power spectrum to a 2D array (X,Y) describing the power spectrum along a frequency axis.

2 Simulation

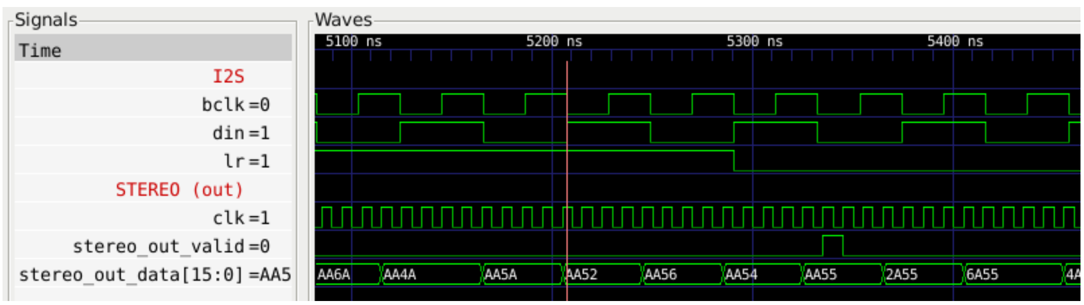
2.1 GHDL

Most IPs are simple unitary functions and can be simulated using `ghdl`. Any IP that comes with a `/sim/Makefile` can be simulated with `ghdl`.
Install `ghdl` to easily simulate modules that come with a testbench:

```
git clone https://github.com/ghdl/ghdl
cd ghdl
./configure
make
sudo make install
```

Once this is done, you can safely simulate a module that comes with a testbench by using *make* in its `sim` folder. For example:

```
cd $git/ip/adau1761/sim/rx
make
```



`GHDL` & `GTK-wave` are convenient tools to quickly run a simulation on a unitary IP

2.2 Vivado - `xsim`

IPs that require Xilinx's dedicated functions, such as BRAM or FIFOs to buffer the input/output data, can only be simulated in Vivado.
To simulate those IPs on your side, either import the IP sources & dependencies and the testbench into Vivado, or use the `sim-project.tcl` if it is delivered with the IP.