Filtering and Decimation by Eight in an FPGA for SDR and Other Applications

Larry Doolittle, LBNL
November, 2006

Introduction

There is currently a good match between telecommunication-grade ADCs and budget FPGAs. These ADCs (e.g., ADS809, LTC2249, AD9236) are available for tens of US Dollars, have 12 to 14 bit accuracy, and can be clocked at 65 to 125 MS/s. The corresponding FPGAs (e.g., Xilinx Spartan3, Altera Cyclone II) are available for a similar price (or more, if you write sloppy code and therefore need a lot of gates), and have no problems clocking at the same rate as the ADC.

The ultimate consumer of the digitized information is usually constrained to a much lower data rate than the 100-plus megabytes per second that each ADC can produce. One of the jobs of the FPGA is to select a subset of the information to pass on to its consumer. One such data reduction process is to select a frequency band of interest. The normal technique is to digitally mix with a virtual local oscillator (often implemented with CORDIC[1] techniques), and low-pass filter and decimate the data stream.

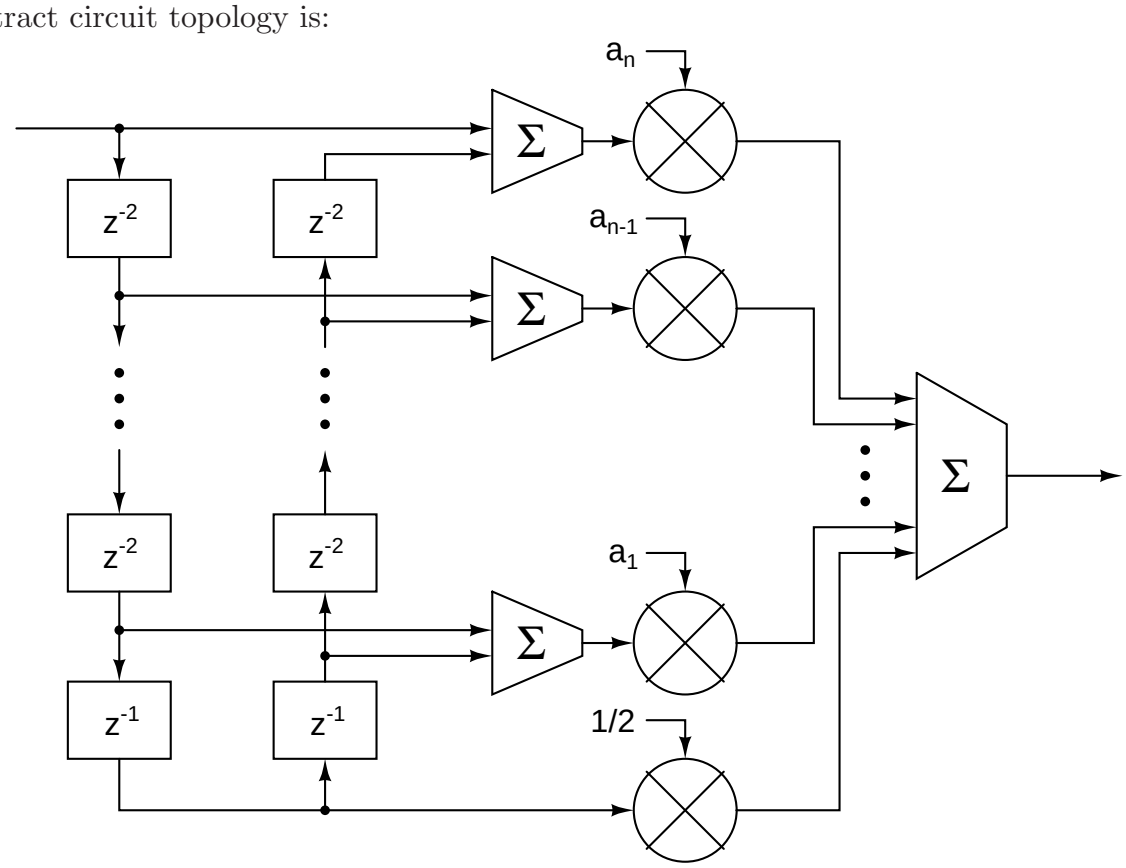
I point out two interesting reference designs: USRP[2] and LLRF4[3]. USRP is nice in that it is both a supported commercial hardware product, and the software to run it (gnuradio) is fully open source and actively maintained. LLRF4 is a more up-to-date implementation of similar concepts, but incomplete and for a non-SDR (Software Defined Radio) application. Supporting LLRF4 is the motivation for this study.

A single ADC in each case puts out about 128 MBytes/sec. A downconversion step temporarily doubles that to 256 MBytes/sec. Filtering and decimation by a factor of eight will take that data rate down to 32 MByte/sec, within the data transfer capability of the USB 2.0 interface to the host computer. The rest of this paper discusses the filtering and decimation process that applies to each of the two data streams (I and Q) that come out of the downconverter at the full ADC sample rate.

Two filter architectures get pulled from DSP toolbox to accomplish this step: half-band filters, and cascaded integrator-comb (CIC) filters. The latter is nicely discussed by Matthew Donadio[4], and I will only briefly show its application. I did not find a similar document describing the fundamentals of half-band filters; the best hints were in a discussion[5] strongly keyed to closed-source software, so I will pursue that angle here.

Half-Band Filters

The half-band filter architecture is a linear-phase finite-impulse-response (FIR) filter. Its abstract circuit topology is:

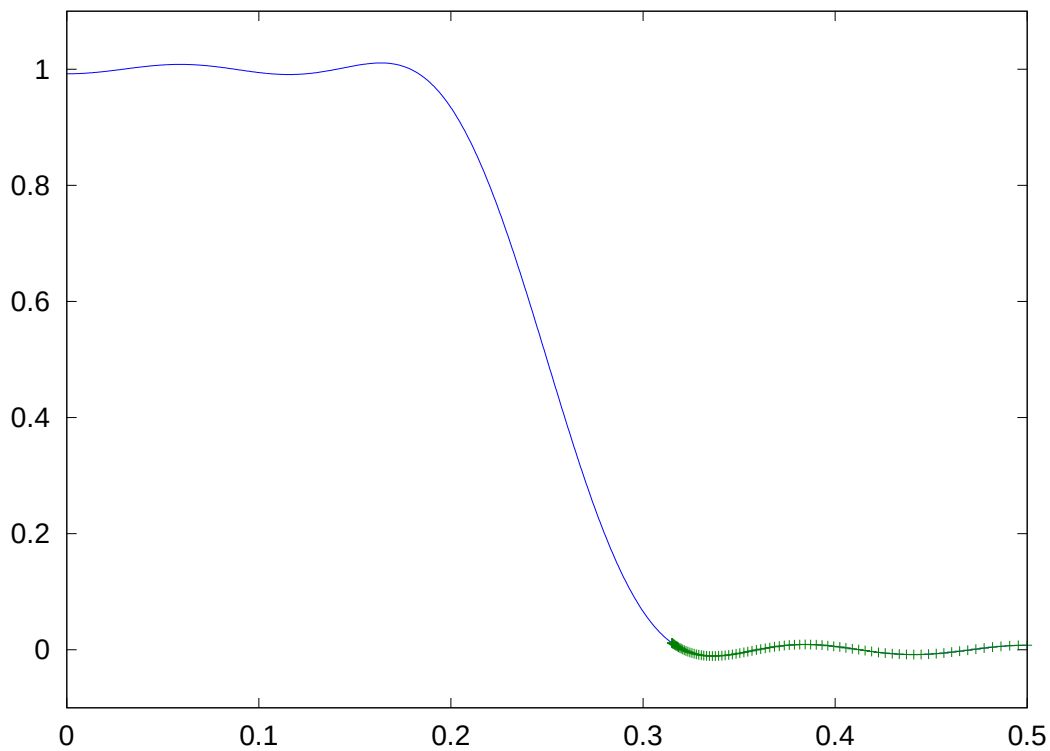


Its frequency response is

$$A = 0.5 + \sum_{k=1}^{2n-1} a_k \cos(kx)$$

where x is the normalized digital frequency, ωT. When followed by a decimator, its purpose is to have as small a response as possible within the high frequencies that will alias onto the pass-band after decimation.

An initial approximation to such a filter can be constructed by selecting points within the reject window, and forming a linear-least squares fit (to zero) of the above function. An $n = 4$ fit to points within 74% of the Nyquist frequency is shown here.



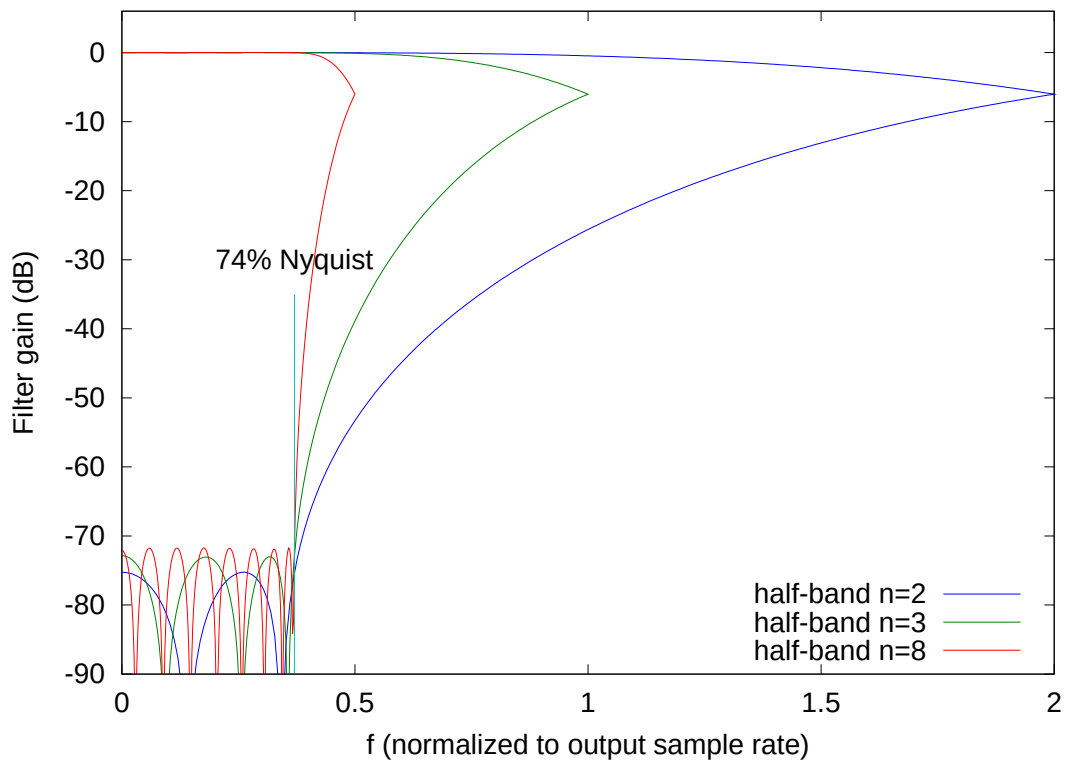
Properly refining the coefficients to make each ripple exactly the same size is a bothersome exercise, but one with a well established lore in curve-fitting circles. The final shape within the stop band (and, by symmetry, the pass band) is patterned after a Chebyshev polynomial of order $2n$.

After decimation, signal in the stop band that leaks through the filter is aliased on top of the pass band, and represents error, or corruption, of the desired signal.

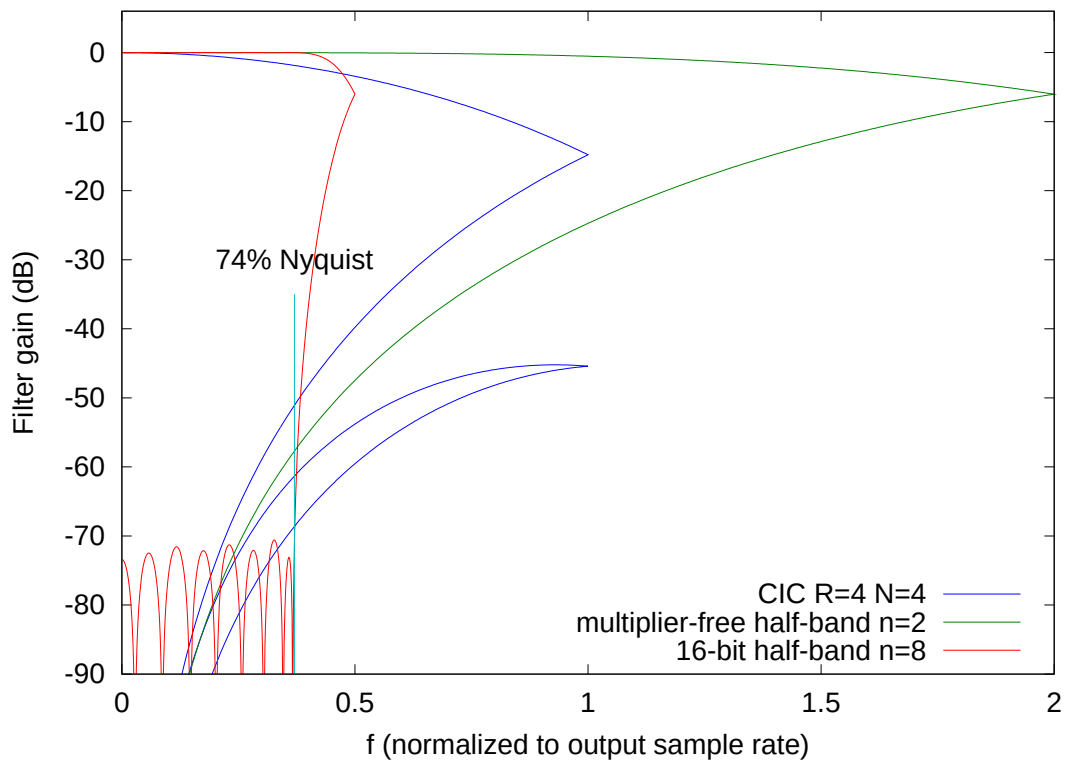
This step is most efficiently repeated when each step decimates by a factor of two. Early filtering steps need relatively small order, because the final pass band is a small fraction of its input band, and later steps operate with a reduced data rate.

Combining Filters

Such a three-step (decimation by eight) process is represented here, where each filter's response is shown after decimation, with the stop band folded over (aliased) onto the pass band, as it as after decimation.



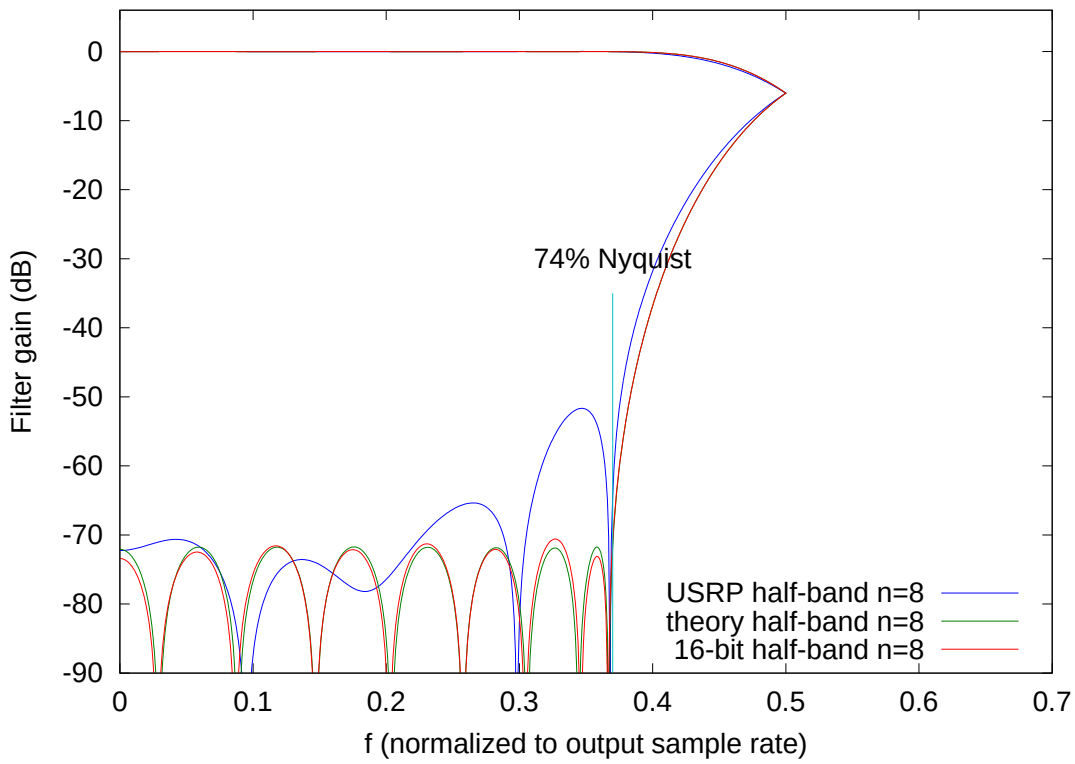
The output of first filter ($n = 2$) is decimated by a total of 2, and therefore only need to yield one result every two clock cycles. Thus a single multiplier can be used. The second filter ($n = 3$) decimates by an additional factor of two, and therefore only needs to yield one result every four clock cycles, so it too can be implemented with a single multiplier. Likewise for the third filter, which must accomplish eight multiplies in eight clock cycles. This set of filters attenuates all aliased signals by at least 71.7 dB.



This figure above shows two other filters that use fewer multipliers than the full half-band filter. The fourth-order CIC filter (as used in the USRP) decimates by a factor of four, so more frequencies alias onto the pass band. This effect is shown by additional folding in its response curve. It is versatile, with its decimation ratio easily configured on-the-fly.

The multiplier-free half-band filter has the nominal $n = 2$ coefficients from before [-0.0666 0 0.5664 1 0.5664 0 -.0666] approximated by [-1/16 0 9/16 1 9/16 0 -1/16].

The $n = 8$ half-band filter is shown for comparison, this time with its coefficients rounded to a 16-bit binary representation.



Finally, I can see that the author of the $n = 8$ half-band filter incorporated in the USRP code base has not read this paper (since it wasn't written yet). The concept is sound, but the (non-zero) filter coefficients would better changed from [-16 74 -254 669 -1468 2960 -6158 20585] to [-49 165 -412 873 -1681 3135 -6282 20628].

Conclusion

An Octave[6] script to compute halfband filters is shown in the appendix. halfgen4 accepts a stopband width (up) and order (N), and produces the full set of FIR coefficients ($4N-1$) for the half-band filter. The stopband width is given as a fraction of the input sampling rate, so 74% of output Nyquist used for the $N = 8$ case of this paper is represented as up=0.185.

This paper, and all the files used to generate it, are posted online[7].

References

[1] Coordinate Rotation Digital Computer,
http://dspguru.com/dsp/faqs/cordic

[2] Universal Software Radio Peripheral,
http://www.ettus.com/products

[3] http://recycle.lbl.gov/llrf4/

[4] CIC Filter Introduction, Matthew P. Donadio, m.p.donadio@ieee.org, 18 July 2000,
http://dspguru.com/dsp/tutorials/cic-filter-introduction

[5] http://www.dsprelated.com/showmessage/31018/1.php

[6] http://www.gnu.org/software/octave/

[7] http://recycle.lbl.gov/∼ldoolitt/halfband/

Appendix

```
function A=halfgen4(up,N);
% up is the stopband width, as a fraction of input sampling rate
% N is the order of half-band filter to generate
% A is the full set of FIR coefficients, 4*N-1 long
npt=N*20;
wmax=2*pi*up;
x0=([0:npt]-.0)'/npt;
yfit=1-x0.^2;   % possibly bogus, but good enough to get started
wfit=yfit*wmax;
q=[1:2:(2*N-1)];
target=.5*ones(length(wfit),1);
basis=cos(wfit*q);
l=basis\target;
% Not Pretty.  I got frustrated and just brute-forced the weights.
% I have no reason to think the process is stable.
weight=1+wfit*0;
hold on
for iter=[1:35]
    err=abs(basis*l-.5);
    maxerr=max(err);
    ix=find(err>maxerr*0.99);
    boost=1+1.5/(iter+10);
    weight(ix)=weight(ix)*boost;
    wbasis=basis.*(weight*ones(1,N));
    wtarget=target.*weight;
    l=wbasis\wtarget;
end
err=abs(basis*l-.5);
maxerr=max(err);
ix=find(err>maxerr*0.99);
% Expand the N-long l array to the (4*N-1)-long A array,
% suitable for polyval(A,z) computation of filter behavior
A=reshape([l';l'*0],1,length(l)*2);
A=A(1:length(A)-1);
A=[fliplr(A) 1 A]/2;
```