Exercise 1/B

Design and Implementation of Digital Filters

Balázs Bank and Kristóf Horváth
BME Department of Measurement and Information Systems

1 Introduction

The goal of this measurement is the implementation and testing of FIR and IIR filters using the ADSP-BF537 EZ-Kit Lite evaluation board with 16 bit fractional arithmetic.

Filtering is one of the most commonly used building blocks of digital signal processing algorithms and applications. A digital filter implements a linear (and usually time invariant) process that can be described by its impulse response $h[n]$: the output of the filter $y[n]$ is the convolution of the input signal $x[n]$ and the impulse response of the filter $h[n]$, that is, $y[n] = h[n] * x[n]$. In the frequency domain, the Fourier transform (or spectrum) of the output $Y(\vartheta)$ is the product of the Fourier transform of the input $X(\vartheta)$ and the Fourier transform of the filter impulse response $H(\vartheta)$, that is, $Y(\vartheta) = H(\vartheta)X(\vartheta)$. The variable $H(\vartheta)$ is actually the transfer function of the filter, whose magnitude $|H(\vartheta)|$ thus gives a direct indication of how the different frequency components of the input are emphasized or attenuated by the filter.

Note that in a strict sense, the term *filter* is used for such transfer functions that remove or filter out certain components of the signal and pass through others ideally without any modification. In a more general sense, a digital filter can be used to implement any kind of impulse response or transfer function, an example can be modeling the transfer function of a loudspeaker, or equalizing its response so that it becomes closer to ideal. In this laboratory we will only deal with simple low-pass, high-pass, band-pass or band-reject filters.

In this document only a short summary is given about FIR and IIR filters and digital filter design and students are referred to the course this laboratory is based on: Érzékelők és jelfeldolgozás – Perception and Signal Processing. Checking the documentation of the MATLAB commands mentioned in this document is also recommended.

2 Digital filters

2.1 FIR filters

A finite impulse response (FIR) filter is a filter whose impulse response is limited in time. The direct form implementation of a FIR filter can be found in Fig. 1.

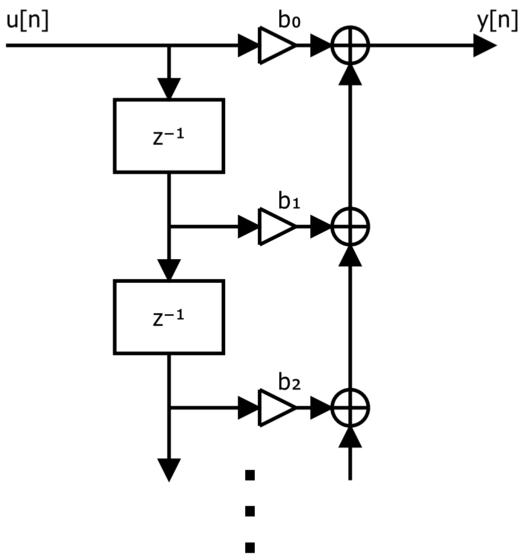

Figure 1: Direct form FIR filter structure.

The output of the FIR filter is calculated using the following formula:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k], \tag{1}$$

where N is the filter order, b_i are the filter coefficients and $x[k]$ is the input signal. This formula is also known as discrete convolution. The $x[n-k]$ in Eq. (1) is implemented as a delay line, whose outputs are commonly referred as taps.

FIR filters are commonly used because of their ease of implementation, inherent stability and low numerical noise. Additionally, FIR filters can be designed to have linear phase characteristics, so their delay becomes frequency-independent, which is important in preserving the shape of the input signal. Their main disadvantage is that significantly higher filter orders are required compared to IIR filters with similar specifications.

FIR filters can be most simply designed by the window method where the ideal impulse response of the filter – computed as the inverse Fourier transform of the target frequency response – is windowed to a certain length given by the filter order N. This type of design is achieved by the `fir1` and `fir2` commands in MATLAB. It is also possible to design FIR filters by numerical optimization, for example, the `firls` command uses a least-squares design that minimizes the mean squared error between the frequency response of the filter and the target response. Alternatively, the `firpm` implements the Parks-McClellan algorithm that uses the Remez exchange algorithm to provide an equiripple approximation, meaning that the ripple of the filter response is even both in the passband and stopband. The amount of ripple can be controlled by adding different weights to the pass-band and stop-band: adding more weight to the pass-band decreases the pass-band ripple since it weighs the pass-band errors more, while a larger weight at the stop-band decreases the stop-band ripple. Obviously, increasing the filter order N leads to better approximation for all types of FIR filter design with the expense of larger computational complexity.

2.2 IIR filters

Contrary to FIR filters, infinite impulse response (IIR) filters have feedback in their structure, thus their impulse response does not become zero after some time.

There are many implementations available for IIR filters. Direct form, or canonical realizations have the simplest structure with the transfer function of

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots b_N z^{-N}}{1 - a_1 z^{-1} - a_2 z^{-2} + \ldots + a_N z^{-N}}. \tag{2}$$

Fig. 2. shows the Direct-form 1 (DF1) structure, which is recommended for implementing IIR filters using fixed-point arithmetic.

The output of an IIR filter in direct form is calculated using the following formula:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k], \tag{3}$$

where N is the filter order, b_i and a_i are filter coefficients and $x[k]$ is the input signal.

The main advantage of IIR filters compared to FIR filters is that they can realize the typical low-pass, high-pass, band-bass and band-reject filter specification using much lower filter orders. On the other hand, designing IIR filters is more complicated, and their implementation can suffer from stability and numerical noise issues.

The most common way of designing IIR filters is based on the design of analog filters and then conversion to digital. In MATLAB this is achieved by the `butter`, `cheby1`, `cheby2` and `ellip` commands. The simplest is the Butterworth filter (`butter`) with a flat passband and monotonically decaying stop-band response, with a disadvantage of a relatively moderate slope at the transition from the pass-band to the stop-band. The steepness of the transition can be increased (and thus the transition region decreased) by allowing some ripples in the

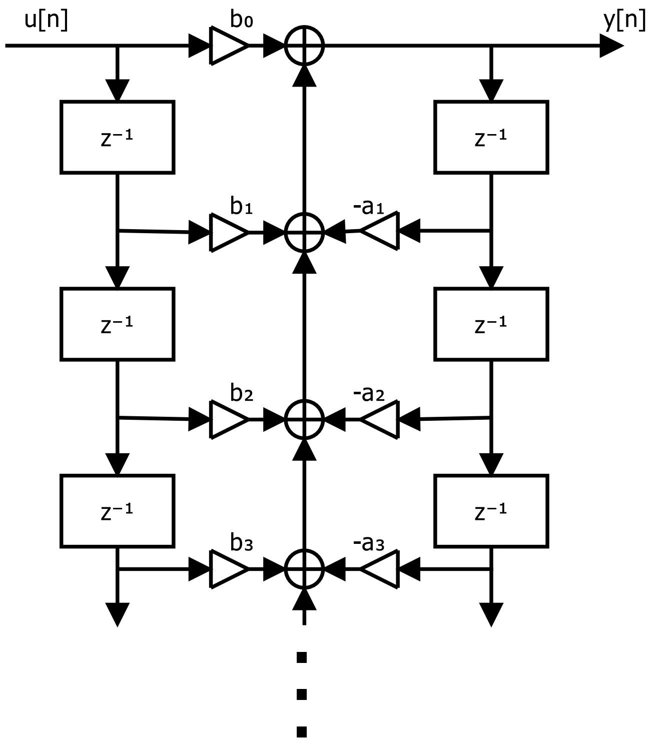

Figure 2: Direct-form 1. structure.

pass- or stop-band. Chebyshev 1 filters (`butter`) allow a pass-band ripple, Chebyshev 2 filters (`butter`) allow a stop-band ripple, while elliptic filters (`ellip`) add both for a steeper transition.

Note that IIR filters can also be designed by numerical algorithms that optimize the filter coefficients in such a way that the transfer function approximates the desired specification. This is usually done by minimizing the mean squared error between the transfer function of the filter and the target response, called least-squares filter design, but this out of the scope of this laboratory.

3 Digital signal processors (DSPs)

Digital Signal Processors are specialized microcontrollers that are optimized for signal processing applications. The main difference between general-purpose microcontrollers and DSPs lies in their arithmetic-logic units (ALUs). DSPs can perform single instruction multiple data (SIMD) operations, and have accumulator registers, using which multiply-and-accumulate (MAC) operations can be performed efficiently. This way, implementing a FIR filter tap can be done in one instruction cycle.

Additionally, DSPs have special addressing modes for increasing the efficiency of signal processing algorithms. For example, modulo addressing is useful for implementing circular buffers, as well as implementing FIR filters. Bit-reversed addressing is used for implementing the fast Fourier-transform (FFT) algorithm.

3.1 Floating-point signal processors

High-performance DSPs often have a floating point unit (FPU) which makes single-cycle floating-point operations possible. The advantages of floating-point number representation include wide dynamic range of values possible and ease of use compared to fixed-point numbers. Disadvantages include difficult rounding rules and complex FPU hardware design which results in higher price.

3.2 Fixed-point signal processors

In this laboratory we use fixed-point signal processors. Fixed-point DSPs are close to high-performance microcontrollers, except that their ALU has accumulators whose widths are at least twice the word length of the native word format. Using an accumulator, MAC instructions are performed without rounding or truncation after each multiplication, which results in low numerical noise power in digital filter implementations. For example, a FIR filter or an IIR DF1 filter can be implemented in such way that the accumulator is rounded only at the output of the filter.

Usually, signal processors have limited set of natively supported fractional types. The Analog Devices Blackfin series used in this lab support signed 16-bit `fract` and 40-bit `accum` types with Q0.15 and Q8.31 format, respectively. Although the compiler allows to use `accum` type for storing values, for efficient implementation `accum` should only be used as accumulator variable.

4 Fixed-point arithmetic

In fixed-point number arithmetic numbers are represented by storing a predetermined number of digits of their integer and fractional parts. In computing, binary representation is used, thus numbers are represented as the integer multiples of 2^{-k}, where k corresponds to the position of the fractional point. This is also known as binary scaling. For example, with $k = 0$ choice the numbers represented are integers, but with $k = 3$ the numbers are multiplies of 0.125. Practically, k represents the number of fractional bits.

Fixed-point number formats are denoted by Q-notation: Q$j.k$ means a fixed-point number with j-bit integer part and k-bit fractional part. Signed numbers use two's complement to represent negative values, in which case the upper bit representing the sign is omitted from the notation. For example, a signed Q3.4 number is 8 bits wide: 1 sign bit + 3 integer bits + 4 fractional bits.

It should be mentioned that regardless of the position of the fractional point, the numbers are stored in memory as integers. It is the programmer's task to keep in mind the binary scaling of each numbers.

Trivially, one extreme for binary scaling is integer representation ($k = 0$). The other extreme is when the number has only fractional digits ($k = B$, where B is the number of bits). This way the range of possible numbers is $[-1...1 - 2^{-B}]$.

Real value	Format	Memory
0.25	Q3.5 unsigned	$(00001000)_2$
0.25	Q3.4 signed	$(00000100)_2$
-0.25	Q3.4 signed	$(11111100)_2$

Table 1: Examples for fixed-point numbers.

In fixed-point arithmetic addition is only possible if both numbers have the same binary scaling. Therefore, for efficient algorithms, numbers should have the same format at each summing points, to avoid format conversions.

Unlike addition, multiplication is always possible between different format fixed-point numbers, but the programmer must keep track of the fractional point. By multiplying two B-bit wide numbers the result is $2B$-bit wide. In order to store, or perform additional operations on the number, it must be rounded, or truncated to B bits. The significant part of the result depends on the number format. If the two (signed) multiplicands are in Q$x.y$ and Q$n.m$ representations, the result will be in Q$x + n + 1.y + m$ format. In order to store the result in Q$x.y$, it should be shifted right by $n + 1$ and rounded to the B most significant bits.

Compared to floating-point numbers of the same size, fixed-point number representation has limited range, thus addition can result in overflow (see Fig. 3.), which may result in limit cycles. On the contrary to general purpose microcontrollers, DSPs have saturation mode that sets the result to the maximum (or minimum) possible number depending on whether overflow or underflow has happened. This has less severe effect on the result compared to numerical overflow.

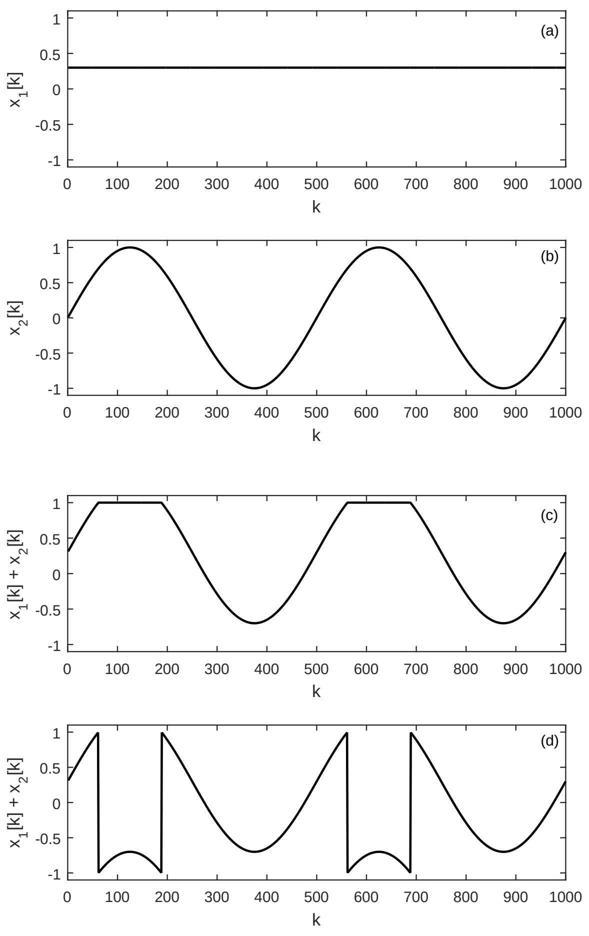

Figure 3: Effect of overflow when adding two signals represented in Q0.15. a.) and b.) are the signals to be added, c.) is the result when using saturation arithmetic, d.) is when saturation arithmetic is not available.

5 Numerical effects in digital filters

5.1 Coefficient quantization

Usually, filters are designed on PC using a specialized software, for example, MATLAB. The design software most commonly uses double-precision floating-point number format, which must be converted to lower precision floating-point or fixed-point numbers to be implemented on DSPs. This quantization modifies the transfer function of the filter, which is undesirable.

IIR filters are much more sensitive to coefficient quantization compared to FIR filters due to their feedback structure. In transfer function terms, not only the zeros, but also the poles are moved by coefficient quantization. In extreme cases, the poles are moved outside of the unit circle and the filter becomes unstable. This is particularly common in high-order direct-form IIR filters. If a high-order IIR filter cannot be implemented in direct-form due to stability issues, it may be converted to a series or parallel set of second-order IIR filters before implementation that has less severe quantization artifacts due to its lower filter order.

It should be noted that filter coefficients may not fit the native `fract` number format. In this case the coefficients should be shifted to the right after filter design, and the firmware should shift to the left the results when taking it out of the accumulator. Note that the signal should also be shifted right before adding to a summation point with scaled-down coefficients.

5.2 Overflow in filters

If both the input and output is represented in the same numerical range, filtering may result in overflow/saturation if the filter amplifies certain frequencies – if the input is a full-scale signal, it will lead to a distorted output if it is amplified even further. A simple method to avoid overflow is to specify and design the filter in such a way that its magnitude response does not exceed 0 dB at any frequencies, or simply scale it down after design. This will ensure that no overflow will happen for sinusoidal inputs. (For ensuring that no overflow happens for any kind of input signals, the constraint is a bit more complicated: one must ensure that the absolute sum of the impulse response of the filter is below unity.) This kind of output overflow does not depend on the implementation, and can be avoided by ensuring that the above constraints are met.

However, in digital filters internal overflow may also happen, even if the net transfer function is properly scaled. A simple example can be increasing the signal by 10 times, and then dividing by 10: from the outside the system looks like having 0 dB gain, but internally it clips all input signals that are above 0.1 or below -0.1 (assuming a fractional representation in the range of ± 1.)

Internal overflow may happen if there is an overflow between the input of the signal and any of the summation points, that is, the transfer function between the input of the filter and a summation point is larger that 0 dB. One of the main advantages of Direct-form 1 structure is that it only has one summation point that equals its output, thus if no overflow happens at the output, one can be sure that there was no internal overflow either. The same is true for FIR filters as well.

Other filter structures are prone to internal overflow, including series second-order sections built from Direct-form 1 components. Internal overflow can be mitigated by scaling down the input of the filter and scaling up its output. This requires the analysis of the filter transfer function for all summation points which of course can be automated for a given filter structure.

5.3 Numerical noise

Numerical noise is caused by rounding (or truncation) after multiplication (including MAC). Although quantization is a deterministic, nonlinear operation, it is easier to be modeled with white noise added to the quantization point (see Fig. 4.). In this case, the added white noise has uniform distribution with variance of $2^{-2B}/12$. If the rounding point is inside a feedback loop in the filter structure (this is the case for all IIR filters), the noise spectrum at the output is affected by the transfer function of the feedback loop.

Note that measurement of numerical noise is out of the scope of this laboratory course, thus numerical noise is mentioned here only for completeness.

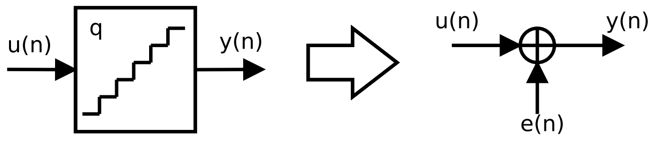

Figure 4: Rounding is modeled as an added white noise.

6 Measurement tasks

6.1 Design and measurement of an FIR filter

Design an FIR filter with a low-pass characteristic using the `firpm` command in MATLAB that meets the following requirements:

Cutoff frequency	5 kHz
Stop frequency	6 kHz
Passband ripple	2 dB
Stopband attenuation	30 dB

Note that you will need to do several iterations setting the input parameters of the `firpm` command and plotting the results. Once the proper set of coefficients is found, export the filter coefficients to a file and import it to the C code of the `BlackFin_FIR_fract` project.

Check the transfer function of the filter running on the BlackFin processor using a digital function generator and the FFT function of the oscilloscope. The most straightforward way of such a measurement is using an excitation signal with a flat spectrum so that the output spectrum directly corresponds to the transfer function of the filter. One such signal is the linearly swept sine (also called chirp) which is a sine wave whose frequency increases linearly in time from a given start frequency to the given stop frequency. First, display the swept sine on the oscilloscope by also connecting the sync output of the generator to the oscilloscope for synchronized (stable) plot. Set the start- and stop frequencies so that the input spectrum is wider than the expected transfer function of the filter. Display the FFT of the swept sine and check if it corresponds to the settings. Finally, connect the swept sine signal to the input of the evaluation board and check the spectrum of the output signal. Observe if the measured transfer function meets the desired specification.

Connect the input of the evaluation board to an audio source (e.g., computer soundcard), and its output to a loudspeaker and listen how low-pass filtering affects the audio signal.

6.2 Implementation of an IIR filter routine

Implement an IIR filter in C starting from the code of the `BlackFin_FIR_fract` project based on the block diagram of the Direct-form 1 (DF1) filter implementation in Fig. 2. While Fig. 2 only shows a third-order filter, the code should work for arbitrary order N.

Throw a dice, and based on the numbers 1–6, chose the set of your coefficients according to Table 2.

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
b_0	0.0157356560693239	0.00257643442532263	0.152843243788085	0.499814997569879	0.0976310729378175	0.292893218813452
b_1	0.0629426242772956	0.0103057377012905	-0.611372975152342	-1.99925999027952	0	-0.428825434727671
b_2	0.0944139364159433	0.0154586065519358	0.917059462728512	2.99888998541927	-0.195262145875635	0.742747440526138
b_3	0.0629426242772956	0.0103057377012905	-0.611372975152342	-1.99925999027952	0	-0.428825434727671
b_4	0.0157356560693239	0.00257643442532263	0.152843243788085	0.499814997569879	0.0976310729378175	0.292893218813452
a_1	-1.88652023640481	-2.63862774389125	-0.651471653594406	-2.63862774389125	-1.93907168024787	-0.732050807568879
a_2	2.24014790055199	2.76930978615149	0.620472122486497	2.76930978615149	1.93105031477870	0.156961002899234
a_3	-1.48589164595573	-1.33928076126520	-0.147379460605609	-1.33928076126520	-1.06051351678402	-0.125600061886466
a_4	0.531293328739442	0.249821669810126	0.0261686639228548	0.249821669810126	0.333333333333334	0.171572875253810

Table 2: IIR filter coefficients.

Measure the transfer function of the IIR filter using the function generator and the oscilloscope in a similar way as was done for the FIR filter above. What kind of transfer function are you seeing? Low-pass, high-pass, band-pass, or band-reject? Measure the cutoff frequency(ies).

6.3 Design of a Butterworth IIR filter

Design a Butterworth IIR filter using the `butter` command in MATLAB with a filter order $N = 4$ and a cutoff frequency of $f_c = 12$ kHz and plot the frequency response in MATLAB.

Export the coefficients in such a way so that it can be implemented on the DSP evaluation board – pay attention that scaling of the coefficients will be necessary as discussed in Sec. 5.1. Measure the transfer function of the IIR filter using the function generator and the oscilloscope and check if it corresponds to the response displayed in MATLAB.

6.4 Design of a Chebyshev 1 IIR filter

Design a Chebyshev 1 IIR filter using the `cheby1` command in MATLAB with a filter order $N = 4$, a cutoff frequency of $f_c = 4$ kHz and a passband ripple of 2 dB and plot the frequency response in MATLAB. Compare the transfer function with that of the Butterworth filter.

Export the coefficients with appropriate scaling to the DSP evaluation board and measure the transfer function of the IIR filter using the function generator and the oscilloscope.

6.5 Supplementary task

Design a Chebyshev 1 IIR filter using the `cheby1` command in MATLAB with a filter order $N = 10$, a cutoff frequency of $f_c = 4$ kHz and a passband ripple of 2 dB and plot the frequency response in MATLAB. Compare the frequency response with the 4th order Chebyshev 1 filter designed above.

Export the coefficients with appropriate scaling to the DSP evaluation board, and measure the response of the IIR filter using the function generator and the oscilloscope. You may notice that the filter becomes unstable due to coefficient rounding.

A solution to this is implementing the filter as a series of second-order sections. For this, the `cheby1` command should be used in such a way that it gives the poles and zeros of the filter, and from the poles and zeros the `zp2sos` command in MATLAB can be used to obtain the coefficients of the second-order IIR filters. Once done, export the coefficients and measure the response of the filter running on the DSP using the function generator and the FFT function of the oscilloscope.