AN02034: Making a sample rate converter for xcore

Publication Date: 2025/10/17
Document Number: XM-015244-AN v1.1.0

IN THIS DOCUMENT

XMOS provides a library, lib_src which provides audio sample rate conversion (SRC) functions for use on *XCORE* devices. This library supports most of the standard audio input and output sample rates and has both Asynchronous (ASRC) and Synchronous (SSRC) functions.

This application note provides a basic introduction to sample rate conversion and contains example up- and down-samplers that enable the reader to create their sample rate converters if the desired conversion is not provided by `lib_src`.

> **Note**
>
> This document covers the design process for the Synchronous Sample Rate Conversion only. Asynchronous Sample Rate Conversion requires a more complex algorithm, users are advised to look at the ASRC component in `lib_src`. If this component does not support the required rates, it is possible to use the synchronous conversion functions described below to match your specific signals to those supported by the ASRC.

1 Introduction to sample rate conversion

Digital processors typically store analogue signals as a sequence of Pulse Code Modulated, or PCM, samples. The PCM samples are sampled along a defined and precise sample rate (for example, 48 kHz), and these points approximate the analogue signal as shown in Fig. 1.

Sample rate conversion means that given a stream of samples sampled at one sample-rate, they can be converted to a stream with a different sample-rate; so that it still sounds (more or less) the same. The problem can be broken down into three steps:

- Down-sampling a signal by a precise integer value, for example 96 kHz to 48 kHz or 96 kHz to 32 kHz.
- Up-sampling a signal by a precise integer value, for example 48 kHz to 96 kHz or 32 kHz to 96 kHz.

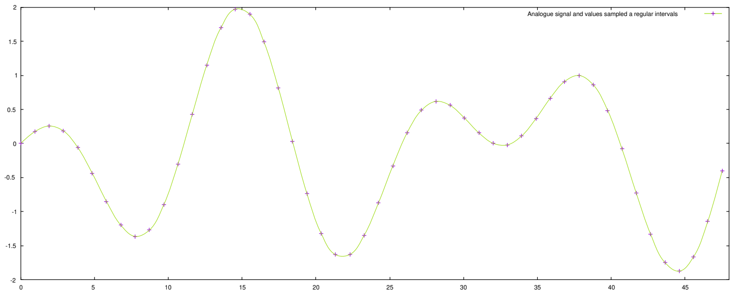

Fig. 1: An analogue signal (green line) sampled at regular intervals (purple plusses).

- Re-sampling a signal by a fractional value, for example, 32 kHz to 48 kHz or 48 kHz to 44.1 kHz;

This document restricts itself to sample rates that are synchronous to each other. Synchronous sample rate conversion assumes that the source and target frequencies are derived from the same source clock.

Asynchronous sample rate conversion deals with sample rates based on two independent clock sources with a variable skew between them.

1.1 Basics of down-sampling by a precise integer value

The principle of down-sampling is simple: remove some of the samples. For example, if it is desired to down-sample by 2x, every other sample should be deleted. In general, to down-sample by a factor of *N*, *N-1* samples out of every *N* should be deleted.

The problem is that this downsampling process creates *aliases* of high-frequency components in the input signal in the down-sampled signal. For example, take the signal shown in Fig. 2. Assuming downsampling from 96 kHz to 48 kHz, and that every other sample is deleted.

The original signal comprised two sine waves; a 4 kHz sine wave and a 47 kHz sine wave. The down-sampling process has not affected the 4 kHz sine wave; however, the 47 kHz sine wave has been converted into a 1 kHz sine wave. Removing *N-1* out of every *N* samples folds up higher frequencies into the lower frequencies.

This folding is shown graphically in Fig. 3. This figure shows a spectrogram of signal strength at various frequencies, and show how these are folded up. In this example the signal is down-sampled by 4x, and each quarter of the spectrum is folded onto the previous quarters much like a paper map would fold up. None of the higher frequencies disappear as such, and they are folded onto lower frequencies. Any higher frequencies may obliterate the signal in the lower frequencies the user is interested in.

When downsampling by an even factor any very high-frequency signals close to the Nyquist frequency (the highest frequency that can be represented at the input sample rate), get folded to a value close to 0 kHz. When downsampling by an odd factor, the highest frequencies end up on the new Nyquist frequency.

To avoid audible artefacts, one must therefore filter out any high-frequency signals before the downsampling. That is, if down-sampling from 96 to 24 kHz, first filter out any signals between 12 kHz and 48 kHz in the input 96 kHz signal, and then every other sample can be safely removed. As will be seen later, there are trade-offs to be made in this filtering process.

To down-sample by 8x there are four choices:

- Three times down-sample by 2x (2x2x2 = 8)

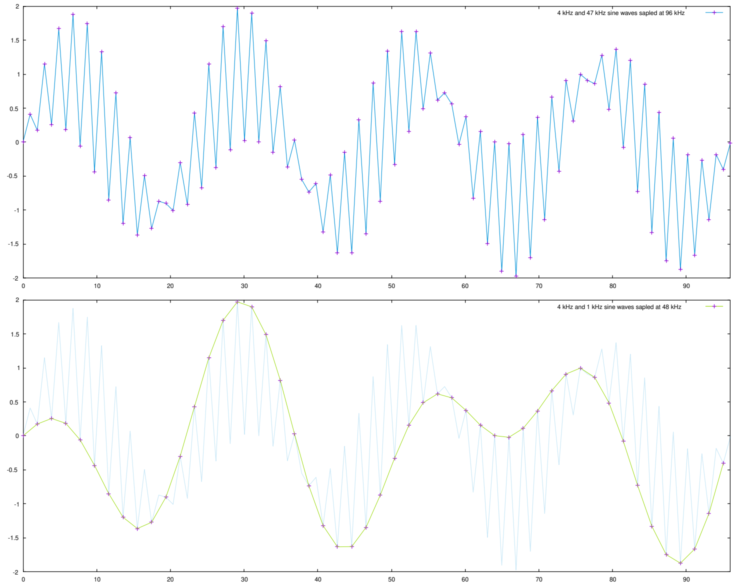

Fig. 2: A signal sampled at 96 kHz (top image) and 48 kHz (bottom image) by removing every other sample.

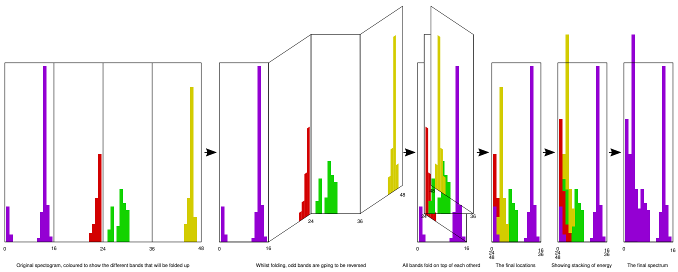

Fig. 3: Frequency folding when downsampling 4x, from 96 to 24 kHz sample rates.

- First down-sample by 2x and then 4x (2x4 = 8)
- First down-sample by 4x and then 2x (4x2 = 8)
- or down-sample by 8x in one step

There is no easy answer to which one is best; typically, going in smaller steps is computationally cheaper. This will be returned to later in the *filter design section*.

1.2 Basics of up-sampling by a precise integer value

Up-sampling is as easy as down-sampling. To up-sample by a factor *N*, *N-1* zero values should be introduced after each input sample, and multiply each input sample by *N*. The former creates the new sample rate and ensures no attenuation of the input volume.

If the simplest case is examined, up-sampling by a factor of 2, a zero is introduced in every other sample, it can be seen that a significant amount of high-frequency noise is added into the input signal. Similar to the down-sampling case, aliases of the input signal have been created into the output signal, but the spectrum has now been unfolded.

That is, in the case of 48 kHz to 96 kHz, all frequencies between 0 and 24 kHz are aliased onto frequencies between 24 kHz and 48 kHz. In the case of going from 32 kHz to 96 kHz, the input signals between 0 and 16 kHz are unfolded onto 32 kHz to 16 kHz and 32 kHz to 48 kHz.

To up-sample faithfully, a low-pass filter should be run over the output of the up-sampler, and remove any frequencies above the original Nyquist frequency. The same trade-offs apply to these filters. Note that on down-sampling the *input signal* is filtered, whereas on up-sampling the *output signal* is filtered.

1.3 Resampling by a fractional value

In many cases input and output sample rates are not integer multiples of each other. For example, it may be desired convert from 32 kHz to 48 kHz or from 48 kHz to 44.1 kHz; the former is a ratio of 1.5, and the latter is a ratio of 0.91875; an integer ratio of 147/160.

In order to re-sample by a fractional value, firstly up-sample by a whole integer value, and then down-sample by a whole integer value. Up-sampling should be to the Least-Common-Multiple (LCM) of the two frequencies (the smallest integer that is a multiple of the input and output sample rates). In between two low-pass filters need to be run, one after the up-sampler, and one before the down-sampler. These filters may be combined into a single filter with a cut-off at the lowest of the two Nyquist frequencies.

For example, the first example case (32 -> 48 kHz) can be achieved by up-sampling from 32 kHz to 96 kHz and then down-sampling to 48 kHz. This requires an upsampling by a factor of 3 (insert two zeroes between every sample), then filter to remove anything below 16 kHz, and then downsampling by a factor of 2 (remove every other sample). The net effect is that one sample is added every two samples.

The second example case (48 -> 44.1 kHz) will require an up-sample by a factor 147 to a sample rate of 7,056,000 Hz; then to filter this signal to remove anything below 22,050 Hz, and then down-sample by a factor of 160 down to 44.1 kHz. This may appear like a vast amount of work, but the filter has a larger number of zeroes as inputs and can be optimised to a manageable size. This will be discussed later in the document.

2 Finite Impulse Response (FIR) filters

A common and straightforward method to construct a filter is to create an FIR filter. An FIR filter has *N* “taps”; a tap is simply a number that is multiplied by. When applying a filter, one multiplies tap *k* with sample *k*, and sum all the results together. That is, one computes the inner product of the filter *f* with the last *N* samples:

```
o[k] = i[k-N+1] * f[0] + i[k-N+2] * f[1] + i[k-N+3] * f[2] + ... + i[k] * f[N-1]
```

In other words, one needs the most recent *N* input samples and to calculate an inner product with the filter of *N* elements to compute one output sample. Another way to look at it is shown in Fig. 4 where the input comes from the left, and is delayed by one sample by each blocks marked z^{-1}. The current sample is multiplied by `f[6]`, the next most recent sample is multiplied with `f[5]` etc. The results are added together and form the output value.

The values of *f* govern what filter is created. For example, a (very bad) averaging filter can be created by setting f[k] to be 1/N. The next section will deal with filter design.

For now, it is assumed that all values are real numbers, and that the coefficients of *f* add up to 1.0. That way, the average signal strength will not change. For an efficient implementation, one may use integer arithmetic, and one may choose to implement a gain by multiplying all values of *f* by some constant.

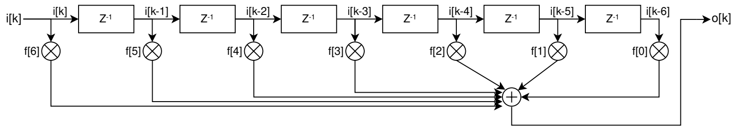

Fig. 4: Filtering with a 7 tap filter.

It must be ensured that the coefficients are applied in the correct order. A convention may be that the newest sample is in i[k] and the oldest sample is in i[k-N+1], and that the coefficients of *f* are stored with f[0] applied to the oldest sample and f[N-1] to the newest sample. Ordering is not an issue if *f* is designed to be symmetric, but, as will be seen later, polyphase filters are never symmetric.

3 Filter design

All three re-sampling methods require a *low-pass* filter to be designed to remove any frequencies above the *Nyquist frequency* of the input or output signal. The filter may be run before down-sampling, after up-sampling, or between up-sampling and down-sampling.

3.1 Design of a low-pass filter and its application to down-sampling

A filter is called a low-pass filter if it lets through all signals with frequencies below some cutoff frequency, and it removes all signals above that frequency. An ideal low-pass filter cannot be constructed as that would require infinite compute, so instead low pass filters are approximated to have the following properties:

- A gain of 0 dB below a first cut-off frequency; this is the *pass-band*.
- A *ripple* of no more than, say, 0.1 dB for any frequencies in this pass-band. That is, the low-frequency signals may have a little bit of gain or attenuation, but at a level that is not perceptible.
- A strong attenuation of, say, -120 dB of signals with a frequency above a second cut-off frequency, called the *stop-band*. Those signals are attenuated so strongly that any aliases will not be perceptible in the output signal.
- Some attenuation between the two cut-off frequencies; this is an area of no concern.

An example low-pass filter response is shown in Fig. 5. This filter was designed using an on-line FIR design tool <http://t-filter.engineerjs.com>. This particular filter was designed to convert from 384 kHz to 192 kHz and not attenuate any signals below 24 kHz. Note that the attenuation of signals over 96 kHz is -150 dB or better, the ripple below 24 kHz is tiny (0.07 dB), and signals between 24 kHz and 96 kHz is variable - a "don't care". The final thing to note is that this filter requires a very modest number of FIR filter taps: only 31. The tap values for this filter are as follows:

```
0.000012655763070268956
0.00011443689561270061
0.0005222575167540901
0.0015503056579715265
0.003206743194403911
0.004441310178041441
0.0027511574458392177
-0.004624415198892449
-0.0174382058234831
-0.028890699902555182
-0.025632812530463855
0.005899831447759155
0.06892545234870025
0.14890438350333784
0.2167916423815637
0.243487446484238895
```

(continues on next page)

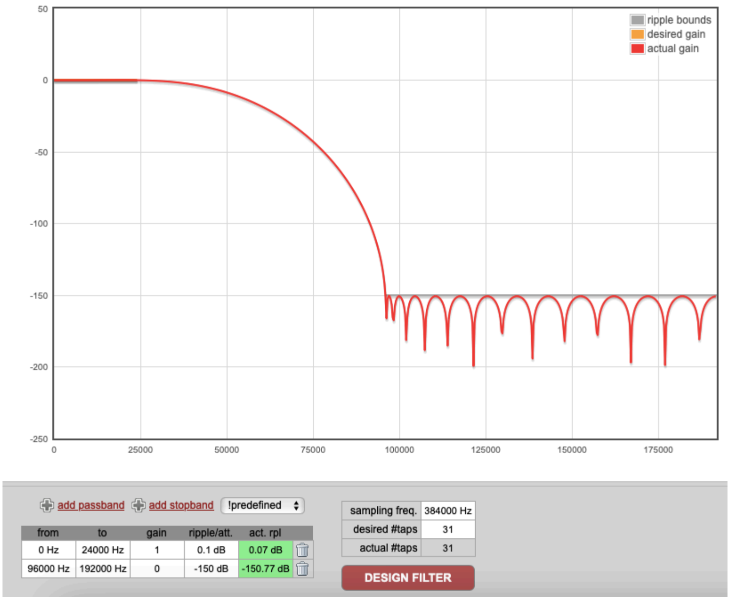

Fig. 5: Frequency response of a low-pass filter, 384 -> 192 kHz.

(continued from previous page)

```
0.2167916423815637
0.14890438350333784
0.06892545234870025
0.005899831447759155
-0.025632812530463855
-0.028890699902555182
-0.0174382058234831
-0.004624415198892449
0.002751157445839217 7
0.0044413101780414 41
0.0032067431944039 11
0.0015503056579715265
0.0005222575167540901
0.0001144368956127061
0.000012655763070268956
```

Whilst this filter is designed to go from 384 kHz to 192 kHz, it can also be used to go from, say 768 kHz to 384 kHz (it will not attenuate signals down to 48 kHz in that case). However, it cannot be used to go from 192 kHz to 96 kHz, as that would start attenuating audible signals in the 12 kHz range.

A filter can be designed that goes from 192 kHz to 96 kHz that will not attenuate signals below 24 kHz, and an example is shown in Fig. 6. Note that to achieve the same levels of ripple and attenuation using more taps (more compute, memory, and a higher signal latency) are required: 48 taps rather than 31.

To go from 96 kHz to 48 kHz another trade-off needs to be made; the lower cut-off frequency has to be reduced from 24 kHz, as the pass-band and stop-band must have a gap between them. A pass-band of 0..20 kHz has been selected, and a stop-band of 24..48 kHz as a compromise. Note that the stop-band must include the Nyquist frequency to attenuate those signals, and the compromise involves allowing some of the frequencies that are close to audible to be removed. This filter needs 169 taps and the response is shown in Fig. 7.

A path has now been constructed to down-convert from 768 kHz to 48 kHz as follows:

- 768 kHz input sample rate, 31-tap filter, down-sample 2x, 384 kHz output

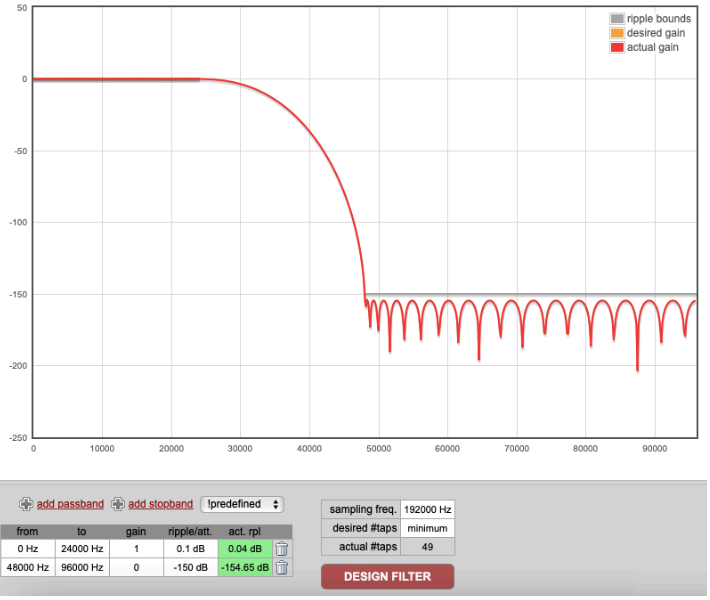

Fig. 6: Frequency response of a low-pass filter, 192 -> 96 kHz.

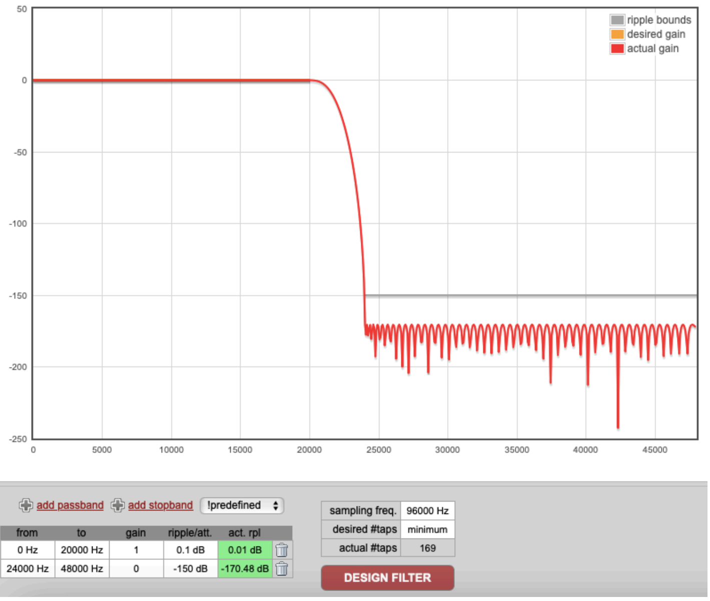

Fig. 7: Frequency response of a low-pass filter, 96 kHz -> 48 kHz.

- 384 kHz input sample rate, 31-tap filter, down-sample 2x, 192 kHz output
- 192 kHz input sample rate, 48-tap filter, down-sample 2x, 96 kHz output
- 96 kHz input sample rate, 169-tap filter, down-sample 2x, 48 kHz output

Note that the filters only have to compute the samples that are actually used, so the filters run at the output sample rate. That means that computationally the following is required:

- 31 x 384,000 = 11,904,000 taps/second for the first filter, 40 us delay
- 31 x 192,000 = 5,952,000 taps/second for the second filter, 80 us delay
- 48 x 96,000 = 4,608,000 taps/second for the third filter, 161 us delay
- 169 x 48,000 = 8,112,000 taps/second for the final filter, 3.5 ms delay
- A total of 30,576,000 taps/second, a total delay of 3.9 ms.

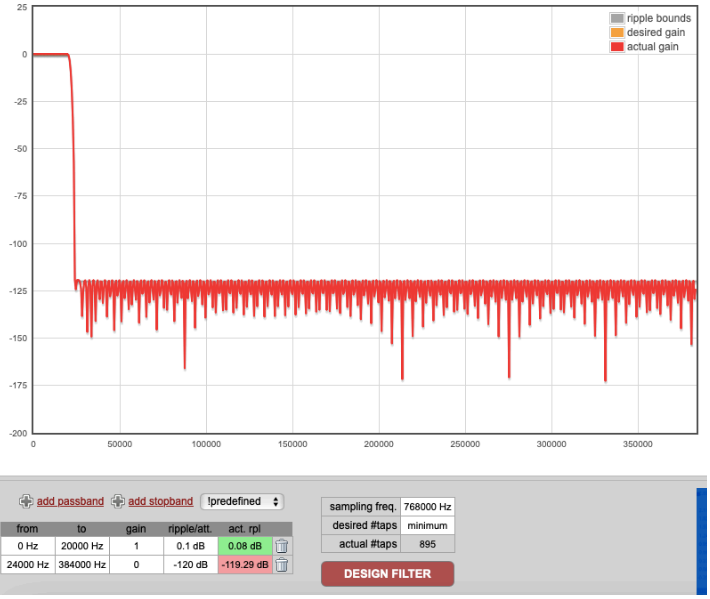

Fig. 8: Frequency response of a low-pass filter, from 768 kHz -> 48 kHz.

Instead of this four-stage approach, other approaches can be attempted. For example, a filter could instead be designed that is suitable to go from 768 kHz to 48 kHz in one step, shown in Fig. 8. The filter design software struggles with this, and a compromise on the attenuation of the stop-band is required - it is set to -120 dB. At this point, a filter is constructed that requires 895 taps, hence:

- 768 kHz input sample rate, 895-tap filter, down-sample 16x, 48 kHz output

The filters run at the output sample rate. That means that the computational requirements are:

- 895 x 48,000 = 42,960,000 taps/second for the filter, 1.16 ms delay

Note that the computational cost has increased slightly; performance is poorer, but it benefits from a smaller delay in the signal. Indeed, to get the same performance, many more taps would be required in the filter.

3.2 Using the same filters for up-sampling

The same filters that were designed for down-sampling can be used for upsampling. The computational requirements are slightly different, so it is assumed that the filter coefficients discovered before are used, but implement filters specific for upsampling.

As previously described, when up-sampling, first insert zeroes and then apply the filter. This means that the filter will multiply with a signal that has a zero in every other input value (assuming an up-sample by 2x). If a signal is upsampled with sample values i[0], i[1], i[2], i[3], ... and, say, a 7-tap filter is applied - one gets the following outputs o0, o1, o2, o3:

```
...
o[6] = i[0] x f[0] + 0 x f[1] + i[1] x f[2] + 0 x f[3] + i[2] x f[4] + 0 x f[5] + i[3] x f[6]
o[7] = 0 x f[0] + i[1] x f[1] + 0 x f[2] + i[2] x f[3] + 0 x f[4] + i[3] x f[5] + 0 x f[6]
o[8] = i[1] x f[0] + 0 x f[1] + i[2] x f[2] + 0 x f[3] + i[3] x f[4] + 0 x f[5] + i[4] x f[6]
o[9] = 0 x f[0] + i[2] x f[1] + 0 x f[2] + i[3] x f[3] + 0 x f[4] + i[4] x f[5] + 0 x f[6]
...
```

Note that for the even output values a four-tap filter is applied `f[0], f[2], f[4], f[6]` , whereas for the odd output values a three-tap filter is applied `f[1], f[3], f[5]`. This is called *poly-phase filtering*, and in this case there are two phases with even and odd filter values. A diagram of this is shown in Fig. 9. If, instead, it was up-sampled by 3x, this would result in three phases a first phase `f[0], f[3], f[6]`, a second phase `f[2], f[5]`, and a third phase `f[1], f[4]`.

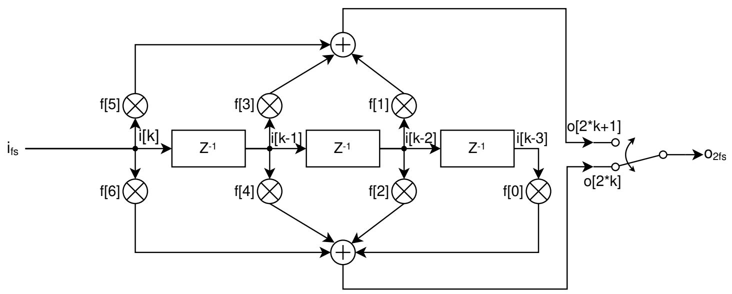

Fig. 9: Polyphase upsampling by a factor of 2 with a 7-tap filter.

It is interesting to note that the total number of taps required is the input sample rate times the filter length, and that these are spread out over the output samples. This can now be summarised and an up-sampler from 48 to 768 kHz can be created as follows:

- 48 kHz input sample rate, up-sample 2x, 169-tap filter, 96 kHz output
- 96 kHz input sample rate, up-sample 2x, 48-tap filter, 192 kHz output
- 192 kHz input sample rate, up-sample 2x, 31-tap filter, 384 kHz output
- 384 kHz input sample rate, up-sample 2x, 31-tap filter, 768 kHz output

Note that the filters only have to compute the samples actually used, so the filters run at the output sample rate. That means that computationally the following is required:

- 169 x 48,000 = 8,112,000 taps/second for the final filter, 3.5 ms delay
- 48 x 96,000 = 4,608,000 taps/second for the third filter, 161 us delay
- 31 x 192,000 = 5,952,000 taps/second for the second filter, 80 us delay
- 31 x 384,000 = 11,904,000 taps/second for the first filter, 40 us delay
- A total of 30,576,000 taps/second, a total delay of 3.9 ms.

3.3 Fractional sample-rate conversion

This is largely left as an area for the reader to explore. However, to give an idea, 48,000 Hz to 44,100 Hz conversion is briefly examined. As previously stated, this comprises upsampling by 147, filtering, and downsampling by 160. Note that 147 and 160 are relatively prime (if not, common factors would have to be divided out), so the filter will need to notionally operate at 7,056,000 Hz and have a cut-off frequency of approximately 20,000 to 22,050 Hz.

These sorts of filters may contain many thousands of taps, but given that they operate on a signal that is almost exclusively zeroes (146 zeroes for every non-zero), and given that only one in 160 values actually needs computing, these filters mostly use a lot of memory to store but not a prohibitive amount of compute despite the very high notional intermediate frequency. The thousands of taps will be cut into 147 different phases, each phase may be 31 or 32 taps only (assuming a 4,600 tap filter).

Normally the phases are applied in decreasing order phase 146, phase 145, phase 144, ..., phase 1, phase 0, phase 145 etc. But given that the task is downsampling and therefore only interested in every 160th sample, phase 146 is initially applied, then phase (146-160) mod 147 = 133, then phase (133-160) mod 147 = 120, etc. Tracking which samples these phases are applied to is slightly fiddly.

One will typically use one of the pre-defined filters in `lib_src` for this purpose.

4 Implementing a down-sampling filter

There are many ways to practically implement a down-sampling filter for *XCORE*. This section describes three options:

- Using standard "C" code
- Use the *XMOS* lib_xcore_math library
- Use the *XCORE* assembly language

Most of these will use integer arithmetic. One could use floating point in C, but given that the signal arrives as an integer sample (over any audio interface), and leaves as an integer sample, one may as well do all the arithmetic in the integer domain.

The convention used when calculating a FIR is that the numbers are represented as a sign bit, a magnitude bit, a binary point, and 30 bits precision below the binary point. As long as all FIR coefficients are stored in this format, the maths will work out just fine.

This section discusses a single down-sampling filter with 31 taps, as that suffices to convert from 768 to 192 kHz; from there on one can use `lib_src` to go to lower frequencies, as that has been optimised to convert to Hi-Fi standards.

4.1 Implementing a down-sampler in C

The simplest and easiest to understand approach is to write the down-sampler in plain C. The code for this is shown below:

```
int32_t ds_history[DS_COEFFICIENTS];

int ds_simple(int sample0, int sample1) {
    ds_history[DS_COEFFICIENTS-2] = sample0;
    ds_history[DS_COEFFICIENTS-1] = sample1;
    int64_t accumulator = 0;
    for(int i = 0; i < DS_COEFFICIENTS; i++) {
        accumulator += ((ds_coefficients[i] * (int64_t) ds_history[i]));
    }
    memmove(ds_history, ds_history + 2, (DS_COEFFICIENTS -2) * sizeof(int32_t));
    return (accumulator + (1<<29)) >> 30;
}
```

An array is declared that stores the last 31 samples, the two samples are stored at the end of this array; the code calculates the inner product (explained below), and finally the array is copied down a bit using `memmove`.

In order to calculate the inner product the code creates a 64-bit accumulator, and adds a full 32 x 32 into 64 bit product to the accumulator, whilst iterating over the data and the coefficients. Once it has calculated a full precision sum a rounding bit is added to it, and then it is shifted down by 30 bits because each of the coefficients had been shifted up by 30 bits.

This solution is simple to explain, but it is not very fast; it takes approximately 834 instructions for each call to this function. Assuming a fully loaded 600 MHz *XCORE* that would be 11.12 us per sample, limiting this to a 90,000 Hz target sample rate.

4.2 Implementing a down-sampler using lib_xcore_math

Another reasonably simple method uses `lib_xcore_math`; a general purpose library that uses the vector unit on *XCORE.AI* to speed up computations. The code for this is shown below:

```
filter_fir_s32_t filter_fast;

int32_t fast_history[DS_COEFFICIENTS];

void ds_fast_init() {
    filter_fir_s32_init(&filter_fast, fast_history, DS_COEFFICIENTS, ds_coefficients, 0);
}

int ds_fast(int sample0, int sample1) {
    filter_fir_s32_add_sample(&filter_fast, sample0);
    return filter_fir_s32(&filter_fast, sample1);
}
```

Like before, the code declares an array that stores the last 31 samples, but it must also declare a filter variable of type `filter_fir_s32_t`. The code initialises the filter with the history array, coefficients, and number of taps, and after that it can compute the result of the filter using a call to `filter_fir_s32()`. Before it does that it needs to add the first sample using `filter_fir_s32_add_sample()`.

This solution is nearly an order of magnitude faster than the simple C code, it takes 95 instructions for each call to this function. Assuming a fully loaded 600 MHz *XCORE* that would be 1.25 us per sample, limiting this to an 800,000 Hz target sample rate. Fast enough for a stereo 786 kHz source to a 384 kHz target.

4.3 Implementing a down-sampler using assembly and the vector-unit

Finally, one can resort to assembly code for full performance. This uses exactly the same algorithm that `lib_xcore_math` uses, but is specialised to just work for filters of size 31; and it assumes that nothing overflows.

The code for this is shown in `ds_vpu.S`, and an explanation is beyond the scope of this document. The ISA explains the operation of each of the instructions. It is included to show the performance of this solution: it is just more than twice as fast as `lib_xcore_math`, taking approximately 44 thread cycles per call. Assuming a fully loaded 600 MHz *XCORE* that would be 0.58 us per sample, limiting this to an 1,700,000 Hz target sample rate. Fast enough for a four-channel 786 kHz source to a 384 kHz target.

5 Implementing an up-sampling filter

All strategies for downsampling can also be applied to upsampling. The same conventions are used for the FIR coefficients, this section implements an up-sampler with the same coefficients.

The first thing that needs to happen is to construct the phases of the polyphase filter. As the task is to upsample by a factor of 2 this results in two phases. The first phase uses all the odd elements of the filter, and the second phases uses all the even elements of the filter. Given that the code adds samples to the end of the buffer, and notionally it will add N-1 zero samples after each input. So the first element that is desired to be computed shall use the last coefficient, and then in steps of N, the lower coefficients. The second element desired to be computed shall use the last-but-one coefficient, and then in steps of N lower coefficients, and so on. This leads to the following two arrays of coefficients:

```
#define CONVERT_FP(x)  ((int)((2*x) * (1<<30)))

int32_t us_coefficients_phase0[(US_COEFFICIENTS+1)/2] = {
    CONVERT_FP(0.0001144368956127061),
    CONVERT_FP(0.0015503056579715265),
    CONVERT_FP(0.004441310178041441),
    CONVERT_FP(-0.004624415198892449),
    CONVERT_FP(-0.028890699902555182),
    CONVERT_FP(0.005899831447759155),
    CONVERT_FP(0.14890438350333784),
    CONVERT_FP(0.24348744648423895),
    CONVERT_FP(0.14890438350333784),
    CONVERT_FP(0.005899831447759155),
    CONVERT_FP(-0.028890699902555182),
    CONVERT_FP(-0.004624415198892449),
    CONVERT_FP(0.004441310178041441),
    CONVERT_FP(0.0015503056579715265),
    CONVERT_FP(0.0001144368956127061),
    0
};

int32_t us_coefficients_phase1[(US_COEFFICIENTS+1)/2] = {
    CONVERT_FP(0.000012655763070268956),
    CONVERT_FP(0.0005222575167540901),
    CONVERT_FP(0.003206743194403911),
    CONVERT_FP(0.0027511574458392177),
    CONVERT_FP(-0.0174382058234831),
    CONVERT_FP(-0.025632812530463855),
    CONVERT_FP(0.06892545234870025),
    CONVERT_FP(0.2167916423815637),
    CONVERT_FP(0.2167916423815637),
    CONVERT_FP(0.06892545234870025),
    CONVERT_FP(-0.025632812530463855),
    CONVERT_FP(-0.0174382058234831),
    CONVERT_FP(0.0027511574458392177),
    CONVERT_FP(0.003206743194403911),
    CONVERT_FP(0.0005222575167540901),
    CONVERT_FP(0.000012655763070268956),
};
```

Both phases are half the length, but as inserting half zeroes would apply a 0.5x gain to the signal; this is compensated for by multiplying all coefficients by 2x.

To up-sample the system needs to apply the first phase on the historical data to calculate the first output sample, and then apply the second phase on the same historical data to calculate the second output sample. One can see that for N phases, N output samples can be computed for each input sample.

5.1 Implementing an up-sampler in C

Similar with the down-sampler, the simplest approach is to write the up-sampler in plain C. The code for this is shown below:

```
int32_t us_history[US_COEFFICIENTS/2];

static int us_inner_product(int32_t coefficients[]) {
    int64_t accumulator = 0;
    for(int i = 0; i < US_COEFFICIENTS/2; i++) {
        accumulator += ((coefficients[i] * (int64_t) us_history[i]));
    }
    return (accumulator + (1<<29)) >> 30;
}

void us_simple(int out[2], int in_sample) {
    us_history[US_COEFFICIENTS/2-1] = in_sample;
    out[0] = us_inner_product(us_coefficients_phase0);
    out[1] = us_inner_product(us_coefficients_phase1);
    memmove(us_history, us_history + 1, (US_COEFFICIENTS/2 - 1) * sizeof(int32_t));
}
```

The code declares an array that stores the last 16 samples; since it is notionally storing zeroes between each sample it only needs half the filter-length. The code stores the new sample at the end of the array, and then calculates the inner products with each of the two phases to produce two outputs (explained below), and finally the code needs to copy the array down one sample using `memmove`.

To calculate the inner product the code creates a 64-bit accumulator, and adds a full 32 x 32 into 64-bit product to the accumulator, whilst iterating over the data and the coefficients. Once it has calculated a full precision sum it adds a rounding bit to it, and then it shifts down by 30 bits because each of the coefficients had been shifted up by 30 bits.

This solution is simple to explain, but not very fast; it takes approximately 600 instructions for each call to this function. Assuming a fully loaded 600 MHz *XCORE* that would be 8 us per sample, limiting this to a 125,000 Hz source sample rate.

5.2 Implementing an up-sampler using lib_xcore_math

Code using `lib_xcore_math` is shown below:

```
filter_fir_s32_t filter0_fast;
filter_fir_s32_t filter1_fast;

int32_t fast0_history[(US_COEFFICIENTS+1)/2];
int32_t fast1_history[(US_COEFFICIENTS+1)/2];

void us_fast_init() {
    filter_fir_s32_init(&filter0_fast, fast0_history, (US_COEFFICIENTS+1)/2, us_coefficients_phase0, 0);
    filter_fir_s32_init(&filter1_fast, fast1_history, (US_COEFFICIENTS+1)/2, us_coefficients_phase1, 0);
}

void us_fast(int out[2], int in_sample) {
    out[0] = filter_fir_s32(&filter0_fast, in_sample);
    out[1] = filter_fir_s32(&filter1_fast, in_sample);
}
```

The code needs two history buffers for each of the two filters, and to initialise both filters. The code needs to initialise each filter with the history array, coefficients, and number of taps, and after that it can compute the result of the filter using a call to `filter_fir_s32()`. Note that it can calculate the two phases in the opposite order; this is because `lib_xcore_math` assumes that `coefficient[0]` is used for the most recent sample, whereas our plain C code assumes that `coefficient[N]` is used for the most recent sample.

Because the filters are so small, the overhead of using `lib_xcore_math` is significant, and it is only 4x faster than the plain C code. It takes 148 instructions for each call to this function. Assuming a fully loaded 600 MHz *XCORE*, that would require 2 us per sample, limiting this to a 500,000 Hz source sample rate. Fast enough for a mono 384 kHz to 768 kHz up-sampler.

5.3 Implementing an up-sampler using assembly and the vector-unit

Finally, one can resort to assembly code for full performance. This uses exactly the same algorithm that `lib_xcore_math` uses, but is specialised to up-sampling. In particular, it uses only one history buffer that is used to *simultaneously* calculate both phases of the filter. The code given here works only for filters of size 16 and assumes that nothing overflows.

For this to work, it needs to interleave the two phases of the filters, in blocks of eight coefficients. This is shown below:

```
int32_t us_coefficients_interleaved[US_COEFFICIENTS+1] = {
    CONVERT_FP(0.0001144368956127061),
    CONVERT_FP(0.0015503056579715265),
    CONVERT_FP(0.004441310178041441),
    CONVERT_FP(-0.004624415198892449),
    CONVERT_FP(-0.028890699902555182),
    CONVERT_FP(0.005899831447759155),
    CONVERT_FP(0.14890438350333784),
    CONVERT_FP(0.24348744648423895),
    CONVERT_FP(0.000012655763070268956),   // phase 1
    CONVERT_FP(0.0005222575167540901),    // phase 1
    CONVERT_FP(0.003206743194403911),    // phase 1
    CONVERT_FP(0.0027511574458392177),    // phase 1
    CONVERT_FP(-0.0174382058234831),    // phase 1
    CONVERT_FP(-0.025632812530463855),    // phase 1
    CONVERT_FP(0.06892545234870025),    // phase 1
```

(continues on next page)

(continued from previous page)

```
    CONVERT_FP(0.2167916423815637),   // phase 1
    CONVERT_FP(0.148904383503337 84),
    CONVERT_FP(0.0058998314477591 55),
    CONVERT_FP(-0.028890699902555182),
    CONVERT_FP(-0.004624415198892449),
    CONVERT_FP(0.004441310178041441),
    CONVERT_FP(0.001550305657971 5265),
    CONVERT_FP(0.0001144368956127061),
    0,
    CONVERT_FP(0.2167916423815637),   // phase 1
    CONVERT_FP(0.06892545234870025),   // phase 1
    CONVERT_FP(-0.025632812530463855),   // phase 1
    CONVERT_FP(-0.0174382058234831),   // phase 1
    CONVERT_FP(0.002751157445839 2177),   // phase 1
    CONVERT_FP(0.003206743194403911),   // phase 1
    CONVERT_FP(0.0005222575167540901),   // phase 1
    CONVERT_FP(0.000012655763070268956),   // phase 1
};
```

The code for this is shown in `us_vpu.S`, and an explanation is beyond the scope of this document. The XCORE.AI Instruction Set Architecture (ISA) explains the operation of each of the instructions. It is included to show the performance of this solution: it is nearly 5x faster than `lib_xcore_math`, taking approximately 34 thread cycles per call. Assuming a fully loaded 600 MHz *XCORE*, that would be 0.45 us per sample, limiting this to a 2,200,000 Hz source sample rate. Fast enough for a five-channel 384 kHz source to a 786 kHz target.

6 Headroom Considerations

This application note has considered samples to be numbers with no top value. In a real system there is a maximum value for samples; for example +/- 1.0 when representing them in floating point, or may be [-2**31..2**31-1] when using 32-bit integers. Sample values cannot be outside this range, and if a signal is outside this range it will be clipped.

When re-sampling signals close to full scale, it is possible to run out of headroom in the resulting signal. Consider a sine wave at *fs/4*. This can be validly sampled as `[+1, +1, -1, -1]`. It can be seen that the maximum amplitude of the continuous sine wave will exceed `1`, but due to the sampling locations headroom has not been exhausted.

When upsampling this signal by a factor of 2, one gets the sequence `[+1, +1.414, +1, 0, -1, -1.414, -1, 0]`. It can be seen that this exceeds the maximum sampled amplitude by a factor of 1.414 (3.01 dB), and will result in distortion.

The example shown above is arguably contrived, but in many cases re-sampling and filtering may slightly increase the magnitude of the signal (even though the energy has not increased). Sufficient headroom should be left on the signal before re-sampling to avoid clipping.

7 Example application

7.1 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `an02034` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an02034
cd app_an02034
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `app_an02034/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

7.2 Running the example

From an XTC command prompt, the following command should be run from the `an02034/app_an02034` directory:

```
xrun --xscope ./bin/app_an02034.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/app_an02034.xe
```

8 Summary

This application note presents how to construct a sample rate converter on *XCORE*. The most important step is the filter design; it governs how much of the higher frequencies end up in the noise floor (for down-sampling), or how much of the lower frequencies end up in the high-frequency bands (for up-sampling).

Once the filter is designed, it can simply be applied when down-sampling. For up-sampling, a polyphase filter has to be constructed. In terms of computational performance, the examples presented enable multi-channel 384 to 768 kHz or 768 kHz to 384 kHz re-samplers.

9 Further reading

- *XMOS* XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest