RF Engines Ltd,
Innovation Centre
St Cross Business Park
Newport, Isle of Wight
PO30 5WB
Tel +44 (0)1983 550330
Fax +44 (0)1983 550340
E-Mail Info@rfel.com

Introduction to Digital Resampling

By Dr Mike Porteous
Principal Digital Systems Engineer, RF Engines Ltd

Overview

This white paper provides an introduction to the digital signal processing technique of resampling. Resampling is required whenever it is necessary to convert sampled data from one sample rate to another. The resampling process involves the signal processing operations of interpolation (increase in sample rate) and decimation (reduction in sample rate). An analysis of these two operations leads to the polyphase decomposition of both the sampled data and the filter required for removing the images created by interpolation and the prevention of aliases created by decimation. The operations of interpolation and decimation are combined to form a resampler that is capable of implementing sample rate changes with arbitrary rational factors. The polyphase decomposition of the sampled data and filters allows efficient resampler architectures to be designed and implemented in hardware.

Worked examples are used to explain polyphase decomposition and show how an efficient architecture for a fixed sample rate change can be obtained. Equations are derived that show how a polyphase architecture can be used for implementing arbitrary rational sample rate change factors. The paper concludes with a brief description of RFEL's efficient and flexible resampler architecture.

Introduction

A fundamental part of digital signal processing is the need to convert sampled signals from one rate to another. For example converting CD audio data sampled at 44.1kHz to DAT audio data sampled at 48kHz (the two sample rates are related by the rational fraction 147/160).

In military applications, high speed ADC devices mean that ever increasing bandwidths can be digitised by a single device. This means that digitised signals can be sampled at much higher rates than is required by the Nyquist sampling theory. If signals are to be isolated for further analysis, the sample rate is often reduced to avoid processing unnecessary bandwidth. The signals of interest may have differing bandwidths and therefore the optimum sample rate will be different for each signal type. Producing digital receivers that can support many different sample rates is therefore desirable.

The sample rates employed are often based on independent legacy systems and it is sometimes desirable to have these legacy systems present in the same hardware e.g. for multi-standard mobile phones and software defined radio (SDR). Ideally the same sampling clock is used in the hardware and hence there is a need to convert between signals generated using different sample rates.

Resampling (also known as sample rate conversion) is a general term that refers to changing the sample rate of a digital representation of a signal whilst preserving, as closely as possible, the information contained in the original signal.

Resampling is based around two signal processing operations:

1. Interpolation – increasing the sample rate of a digital signal
2. Decimation – decreasing the sample rate of a digital signal.

If interpolation and decimation are used together then the resultant system is known as a resampler (or sample rate converter (SRC)). For a resampler, the overall change in sample rate is given by:

$$R = \frac{L}{M}$$

where *L* represents the increase in sample rate due to interpolation and *M* represents the decrease in sample rate due to decimation. In this white paper, only rational rate changes are discussed which means that both *L* and *M* are constrained to have integer values. This may seem to be a limiting feature, however architectures exist that are capable of implementing large values of L and M so the range of rate changes is virtually limitless. Therefore these architectures can be used to approximate irrational rate changes.

The architectures described in this paper are based on polyphase filters and are aimed at implementation on field programmable gate array (FGPA) devices. A more rigorous mathematical treatment of the architectures described in this paper can be found in [1] and [2].

Alternative architectures for resamplers based on cascaded integrator comb (CIC) filters are described in [3]. These architectures are generally less suitable for implementation on modern FPGAs due to the availability of many multiply-accumulate blocks that are ideal for implementing polyphase filter structures. Other disadvantages of the CIC approach are the large bit widths that are required to prevent overflow, the narrow alias-free bandwidth compared to the sample rate and the passband ‘droop’ (which is normally corrected by a FIR filter). However, CIC filters do still have a role to play in certain applications, especially where implementation resources are not pre-prescribed, such as in ASIC devices such as the Analog Devices AD6620 digital receive processor chip [4].

The purpose of this white paper is to give an insight into how FPGA-efficient polyphase resampler architectures can be designed. The signal processing operations of interpolation and decimation are discussed individually below before discussing the development of resampler architectures.

Interpolation

Interpolation is achieved in two stages as illustrated in Figure 1 for L=3:

1. Upsampling - -the placing of L-1 (in this case 2) zeros between the samples in the original signal.
2. Filtering - smoothing the upsampled signal using a filter.

Figure 1 a) shows the original sampled signal, upsampling is achieved by placing 2 zeros between the original samples as shown in Figure 1 b). The upsampled signal is then smoothed using a filter as shown in Figure 1 c).

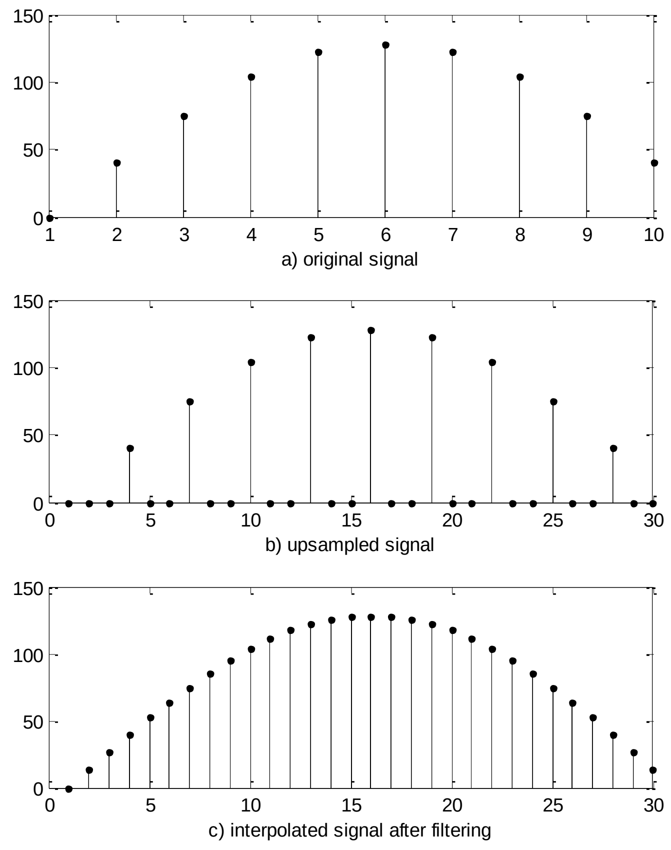


Figure 1: Interpolation by a factor of L = 3.

One question not yet answered is what characteristics are required of the filter? This question can be answered by looking at how the interpolation process behaves in the frequency domain; this is illustrated in Figure 2.

Figure 2 a) shows the spectrum of the original sampled signal. The spectrum contains images of the signal spectrum at multiples of the sample rate; this is a consequence of the original sampling process. Figure 2 b) shows the spectrum after upsampling. Some of the images now lie within the new (higher) sampling rate. The filter is used to remove these images as shown in Figure 2 c).

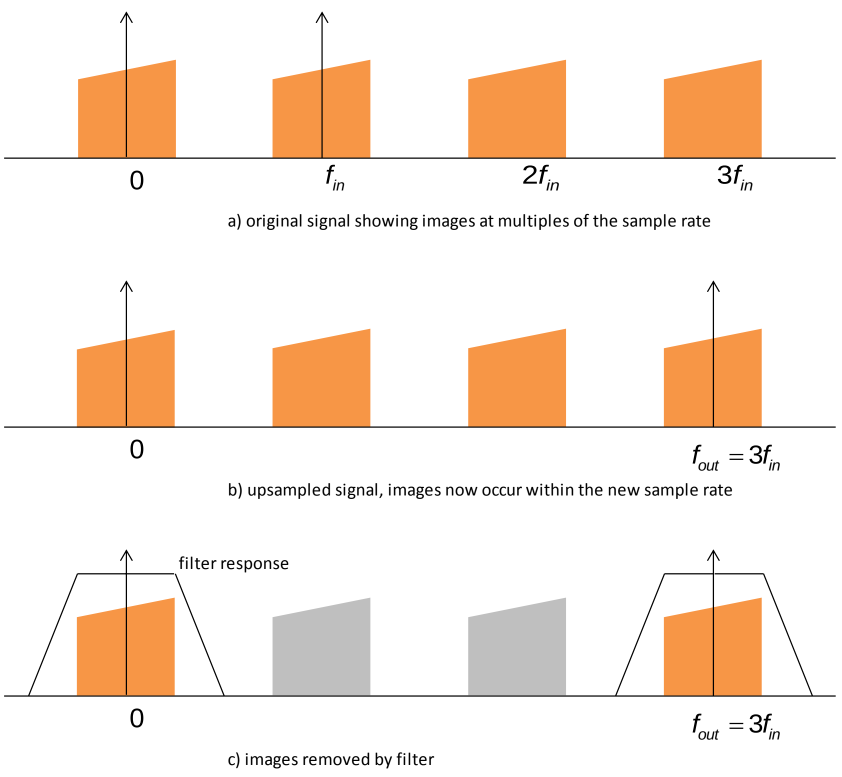

0
f_{in}
$2f_{in}$
$3f_{in}$
a) original signal showing images at multiples of the sample rate
0
$f_{out} = 3f_{in}$
b) upsampled signal, images now occur within the new sample rate
filter response
0
$f_{out} = 3f_{in}$
c) images removed by filter

Figure 2: Frequency domain interpretation of interpolation

The filter is therefore designed to remove these images. This filter is known as an *anti-imaging* filter. In practical systems, some degree of oversampling is required for the signal to be interpolated. This is to allow practical filter designs to be used for removing the images. The stop band response of the filter defines the level of *image rejection*.

Note that for a given interpolation factor, it is always possible to design a filter to remove the images.

A system level diagram for an interpolator is shown in Figure 3.

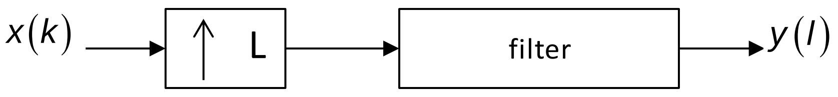


Figure 3: Interpolator system level diagram

The box in Figure 3 containing the up arrow represents the upsampling operation. For a practical implementation, this architecture is inefficient. This is because after the upsampling operation, some of the coefficients in the filter are multiplied by the introduced zeros in the data. The filter also has to be clocked at the higher output sample rate. This inefficiency is explained in more detail as follows.

If the filter is implemented as a finite impulse response (FIR) filter, then the interpolator of Figure 3 can be realised in hardware as shown in Figure 4.

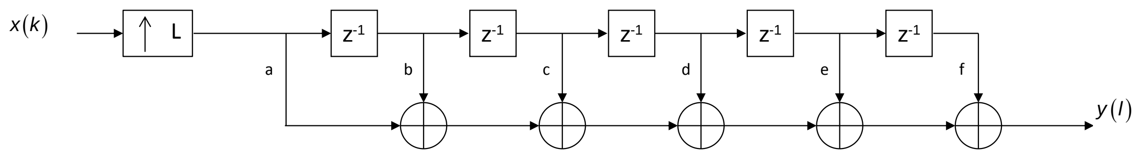


Figure 4: Interpolator architecture using a FIR filter

where the filter response is given by *h(n)={ a, b, c, d, e, f }.* In the architecture of Figure 4, the filter coefficients are ordered from left to right along the length of a tapped delay line. The data samples to be filtered are ordered from right to left as the data enters the filter from the left. This time reversal between the input data samples and the filter coefficients means that the delay line implements a *convolution* i.e. it implements a filtering operation.

k	a	b	c	d	e	f	o/p
0	0	.	.	.	.	.	y0
1	x	0	.	.	.	.	y1
2	x	x	0	.	.	.	y2
3	1	x	x	0	.	.	y3
4	x	1	x	x	0	.	y4
5	x	x	1	x	x	0	y5
6	2	x	x	1	x	x	y6
7	x	2	x	x	1	x	y7
8	x	x	2	x	x	1	y8

Figure 5: Tapped delay line values

Figure 5 shows the status of the tapped delay line in Figure 4 for each instant of time *k* as defined by the input sample rate. The x's in the diagram represent the zeros introduced by the upsampling operation, and the numerical values represent the input data sample indices prior to the upsampling operation (so that 0 represents sample *x(0)*, 1 represents sample *x(1)* etc.). So for example, at time *k=5*, input sample *x(0)* is aligned with filter coefficient *f* and input sample number *x(1)* is aligned with filter coefficient *c*. The dots in Figure 5 represent the values in the delay line prior to the arrival of the first data sample. In practical implementations, the delay line is usually filled with zeros prior to the arrival of the input data samples.

The outputs of the filter are given by Equation 1 ignoring any multiplies that generate a zero output:

$$
\begin{aligned}
y0 &= a0 \\
y1 &= b0 \\
y2 &= c0 \\
y3 &= a1 + d0 \\
y4 &= b1 + e0 \\
y5 &= c1 + f0 \\
y6 &= a2 + d1 \\
y7 &= b2 + e1 \\
y8 &= c2 + f1
\end{aligned}
$$

Equation 1: Filter outputs

where *y0* to *y8* represent the first nine interpolator output data samples.

As shown in Figure 5, for every input sample in the delay line, there are also *L-1* (2 in this case) zero values in the delay line from the upsampling (shown by the x's) and so not every filter coefficient is used to generate a particular output sample. This redundancy leads to an inefficient architecture.

Equation 1 shows that each interpolator output sample is generated by a subset of the filter coefficients, and the subsets that are used vary periodically i.e. the subsets used are {*a,d*}, {*b,e*} and {*c,f*}; these subsets are used in a repeated pattern.

This periodic variation over the filter coefficient subsets gives a clue as to how to develop a more efficient architecture for the interpolator. If the filter coefficients are decomposed according to:

$$h_p(n) = h(p + Ln) \quad p = 0 \text{ to } L - 1$$

Equation 2: Polyphase decomposition of the interpolation filter

where *h(n)* is the filter impulse response defined at the output sample rate. Equation 2 gives the following decomposition for the filter of Figure 4:

$$
\begin{aligned}
h_0(n) &= a, d \\
h_1(n) &= b, e \\
h_2(n) &= c, f
\end{aligned}
$$

Equation 3: Polyphase filters

A comparison with Equation 1 shows that this decomposition gives the same subsets of filter coefficients. The decomposition defined by Equation 2 is known as *polyphase decomposition*.

The use of polyphase decomposition leads to the interpolator architecture shown in Figure 6.

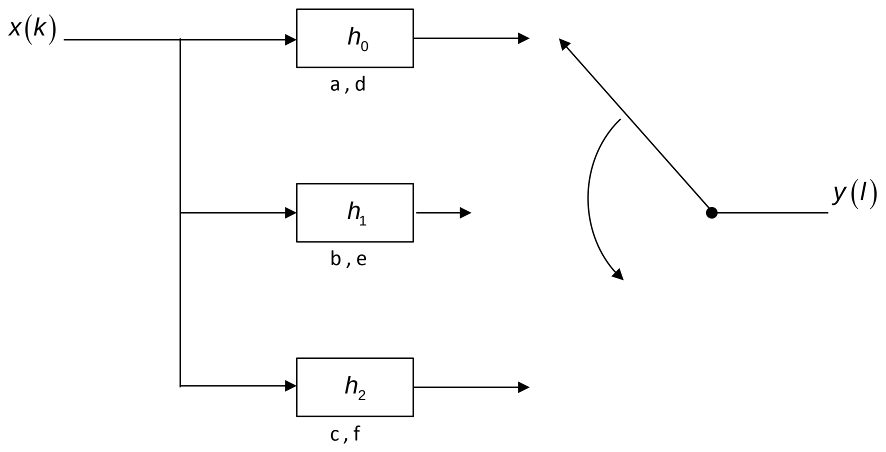


Figure 6: Polyphase interpolator

In Figure 6, the input is simultaneously filtered by all of the polyphase filters. At the output, an anti-clockwise *commutator* sweeps around the filters at the output sample rate and at each time instant, selects one of the filter outputs. Note that the polyphase filters are clocked at the lower input sample rate and so the only part of this architecture that operates at the higher output rate is the commutator. If the commutator starts at time 0 with the output of filter h_0, then it is easy to show using Equation 3 that the architecture of Figure 6 gives the same output as the architecture of Figure 4 (as shown by Equation 1).

This is now an efficient architecture as the zeros introduced by the upsampling operation do not appear in the architecture of Figure 6; this architecture is also clocked at the lower input sample rate. The commutator can be implemented in hardware as a multiplexer if a serial output data stream is required. For a parallel architecture, the commutator can be removed if an output parallelism of L can be used for the output data samples.

The parallelism (number of polyphase filters) of the polyphase architecture equals the interpolation factor L. For large values of L, the required parallelism becomes impractical to implement. For example, for L=64, 64 polyphase filters would be required. For large interpolation factors, the interpolation can be applied in stages, so an interpolation factor of 64 could be implemented with two interpolators each with an interpolation factor of 8; i.e. L can be factored as

$$L = L_1 L_2 \ldots L_n$$

Equation 4: Factoring large interpolation factors

An alternative interpolator architecture would be to implement a single time varying filter. At each time instant, the filter coefficients within the tapped delay line are re-programmed with the next set of polyphase filter coefficients prior to generating the next output sample. This avoids the need for a large degree of parallelism when L is large, however an extra level of complexity is required to ensure that the filter coefficients within the delay line are programmed correctly. Unlike the polyphase interpolator in Figure 6, the time varying filter has to be clocked at the higher output sample rate.

Decimation

Decimation by a factor of M is achieved by downsampling, i.e. retaining every Mth sample from the input as shown in Figure 7 b) for a decimation factor of M=4.

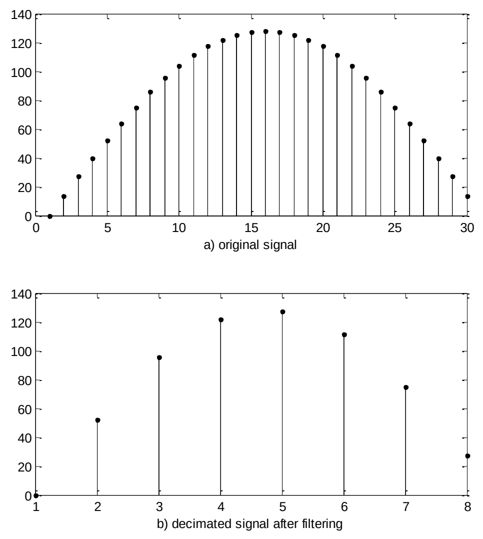
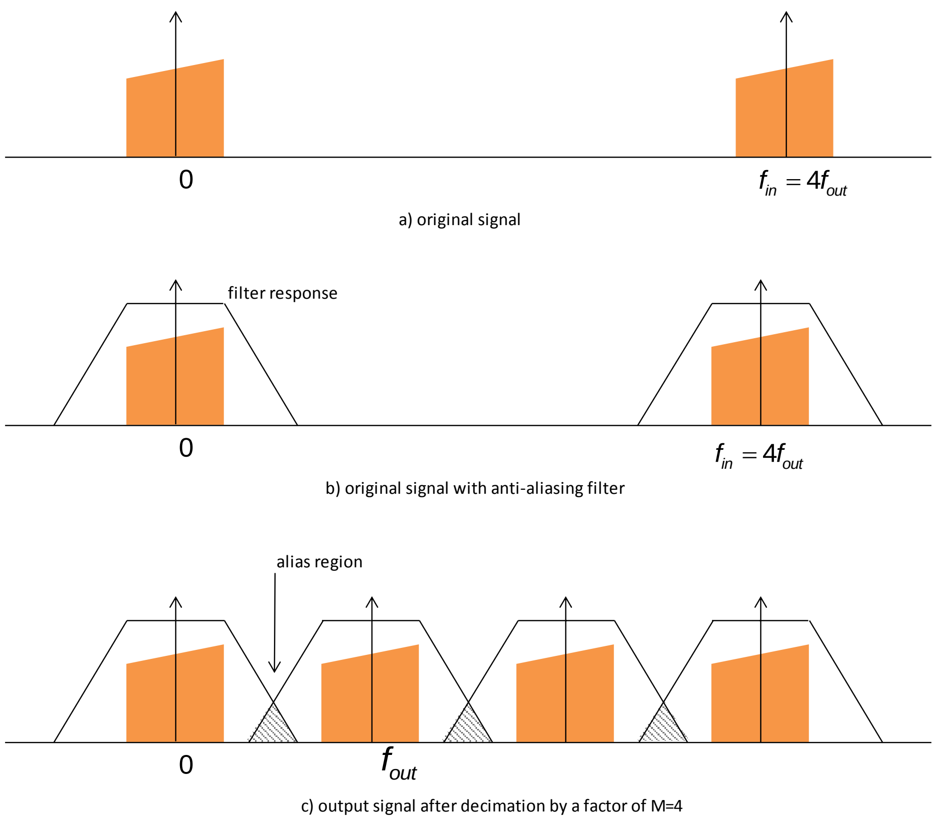


Figure 7: Decimation by a factor of M=4

However for decimation to work correctly, the input signal has to be appropriately conditioned so that distortion is not introduced into the output signal. To avoid distortion, the output signal from the decimator must obey the Nyquist sampling criterion to avoid the introduction of *aliasing*. Filtering prior to the downsampling operation is used to prevent aliasing from appearing in the output signal. To understand how aliasing can occur, it is useful to understand how decimation behaves in the frequency domain. This is illustrated in Figure 8.

Figure 8 shows that the downsampling operation moves the spectral images closer to one another. If the decimation factor is too high then there is a risk that the images will start to overlap, this overlap leads to aliasing as shown by the shaded regions in Figure 8 c). The purpose of the filter shown in Figure 8 b) is to remove the parts of the spectrum that would lead to aliasing. This filter is known as an *anti-alias* filter. The stop band response of this filter defines the level of *alias rejection*.

Note that once aliasing occurs it cannot be removed, this is in contrast to the interpolator where images can always be removed.



0
$f_{in} = 4f_{out}$
a) original signal
filter response
0
$f_{in} = 4f_{out}$
b) original signal with anti-aliasing filter
alias region
0
f_{out}
c) output signal after decimation by a factor of M=4

Figure 8: Frequency domain interpretation of decimation

A system level diagram for a decimator is shown in Figure 9.

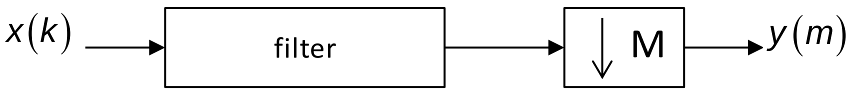


Figure 9: Generalised decimator

The box containing the down arrow represents the downsampling operation; i.e. to retain every Mth sample, and discard all other samples. If the filter is implemented as a FIR filter, the decimator can be implemented using the architecture shown in Figure 10.

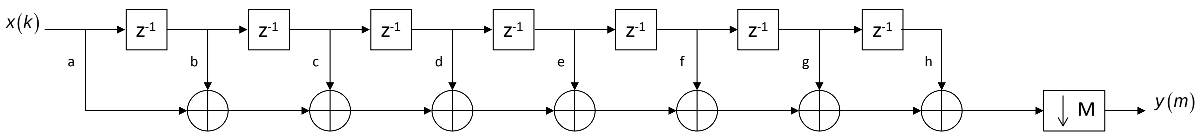


Figure 10: Decimator architecture using a FIR filter

where the filter response is given by *h(n) = { a, b, c, d, e, f, g, h }*. Like the interpolator described previously, this architecture is inefficient. All of the input data is filtered before the downsampler throws away most of the filter outputs. In this example for *M=4*, 3 out of every 4 filter outputs are discarded.

Like the interpolator, a clue to improving the decimator efficiency can be found by looking at the behaviour of the tapped delay line within the filter.

k	a	b	c	d	e	f	g	h	o/p
0	0	.	.	.	.	.	.	.	x
1	1	0	.	.	.	.	.	.	x
2	2	1	0	.	.	.	.	.	x
3	3	2	1	0	.	.	.	.	y0
4	4	3	2	1	0	.	.	.	x
5	5	4	3	2	1	0	.	.	x
6	6	5	4	3	2	1	0	.	x
7	7	6	5	4	3	2	1	0	y1
8	8	7	6	5	4	3	2	1	x
9	9	8	7	6	5	4	3	2	x
10	10	9	8	7	6	5	4	3	x
11	11	10	9	8	7	6	5	4	y2
12	12	11	10	9	8	7	6	5	x
13	13	12	11	10	9	8	7	6	x
14	14	13	12	11	10	9	8	7	x
15	15	14	13	12	11	10	9	8	y3

Figure 11: Tapped delay line values

Figure 11 shows that all of the input data and all of the filter coefficients are required to generate the decimated output. The x's in the last column of Figure 11 show which filter outputs are discarded by the decimation operation. The dots are used to represent the values in the delay line prior to the input data arriving.

In this example, the first output sample is generated at time *k=3* when the first *M=4* input samples are present in the tapped delay line. The reason for this choice of decimation phase will become clear later in the following discussion.

The outputs retained from the filter are given by Equation 5:

$$\begin{aligned}
y0 &= a3 + b2 + c1 + d0 \\
y1 &= a7 + b6 + c5 + d4 + e3 + f2 + g1 + h0 \\
y2 &= a11 + b10 + c9 + d8 + e7 + f6 + g5 + h4 \\
y3 &= a15 + b14 + c13 + d12 + e11 + f10 + g9 + h8
\end{aligned}$$

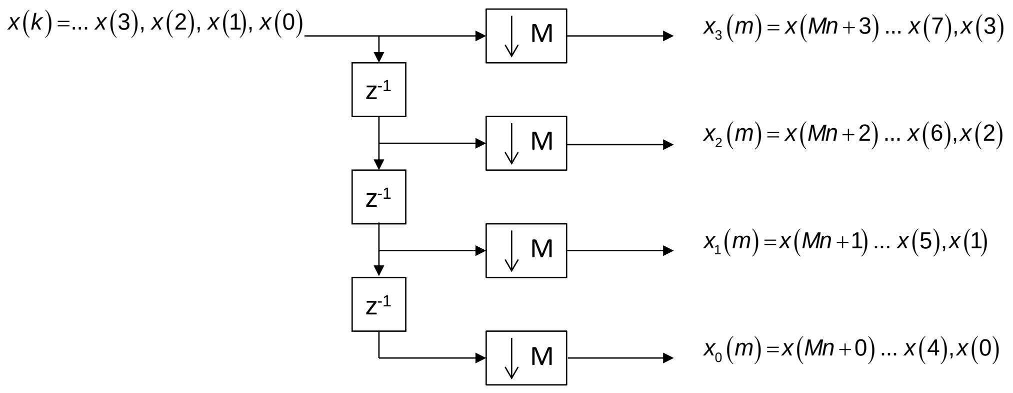
Equation 5: Decimator outputs

Equation 5 shows that each output requires all of the filter coefficients; however each filter coefficient only multiplies every Mth input sample. In this example *for M=4*, coefficient *a* only multiplies the 3rd, 7th, 11th and 15th input samples and coefficient *c* only multiplies the 1st, 5th, 9th and 13th input samples. It appears that to produce a decimated output, the filter needs to operate over a polyphase decomposition of the input data samples.

If we analyse Equation 5 further, another polyphase decomposition appears. Notice that input sample 7 is only multiplied by coefficients {a,e}, and input sample number 10 is only multiplied by coefficients {b,f}. This looks like a polyphase decomposition of the filter coefficients.

A decimator appears to use polyphase decomposition on both the input data samples and the filter coefficients. A polyphase architecture for the decimator can be obtained as follows.

The input data samples can be decomposed into *M=4* phases as shown in Figure 12.



$x(k)=\ldots x(3), x(2), x(1), x(0)$
z^{-1}
z^{-1}
z^{-1}
$\downarrow M$
$\downarrow M$
$\downarrow M$
$\downarrow M$
$x_3(m)=x(Mn+3)\ldots x(7),x(3)$
$x_2(m)=x(Mn+2)\ldots x(6),x(2)$
$x_1(m)=x(Mn+1)\ldots x(5),x(1)$
$x_0(m)=x(Mn+0)\ldots x(4),x(0)$

Figure 12: Polyphase decomposition of the input data samples

Each output $x_p(m)$ in Figure 12 is a decimated (by a factor of *M=4*) version of the input, with each output starting on a different sample or *phase*. Note that the order of the phases is reversed with $x_0(m)$ appearing at the bottom of Figure 12.

The filter coefficients are decomposed using:

$$h_p(n)=h(p+Mn) \quad p=0 \text{ } to \text{ } M-1$$

Equation 6: Polyphase decomposition of the decimation filter

where *h(n)* is the filter impulse response defined at the decimated output rate. Decomposing the filter coefficients by a factor of *M=4* using Equation 6 gives the following polyphase filters:

$$\begin{aligned} h_0(n)&=a,e\\ h_1(n)&=b,f\\ h_2(n)&=c,g\\ h_3(n)&=d,h \end{aligned}$$

Equation 7: Polyphase filters

The polyphase filters are applied to the polyphase decomposition of the input sample data of Figure 12 as follows:

$$\begin{aligned}
y_3(m) &= h_o(m) \otimes x_3(m) \\
y_2(m) &= h_1(m) \otimes x_2(m) \\
y_1(m) &= h_2(m) \otimes x_1(m) \\
y_0(m) &= h_3(m) \otimes x_0(m)
\end{aligned}$$

$$y(m) = y_3(m) + y_2(m) + y_1(m) + y_0(m)$$

Equation 8: Polyphase filters applied to polyphase input data

Where $\otimes$ denotes the convolution (filtering) operation. Note from Equation 8, the order of the polyphase filters is the reverse of the order of the polyphase input data. The following worked example shows that Equation 8 leads to the same decimator output as given by Equation 5 (only output samples 1 and 2 are calculated here):

$$\begin{aligned}
y_3 1 &= a7 + e3 & \qquad y_3 2 &= a11 + e7 \\
y_2 1 &= b6 + f2 & \qquad y_2 2 &= b10 + f6 \\
y_1 1 &= c5 + g1 & \qquad y_1 2 &= c9 \ + g5 \\
y_0 1 &= d4 + h0 & \qquad y_0 2 &= d8 \ + h4
\end{aligned}$$

$$\begin{aligned}
y1 &= y_3 1 + y_2 1 + y_1 1 + y_0 1 & \qquad y2 &= y_3 2 + y_2 2 + y_1 2 + y_0 2 \\
y1 &= a7 + b6 + c5 \ + d4 \ + e3 + f2 \ + g1 + h0 & \qquad y2 &= a11 + b10 + c9 \ + d8 \ + e7 + f6 \ + g5 + h4
\end{aligned}$$

Equation 9: Worked example for the polyphase decimator

Equation 8 thus leads to the following polyphase architecture for the decimator as shown in Figure 13.

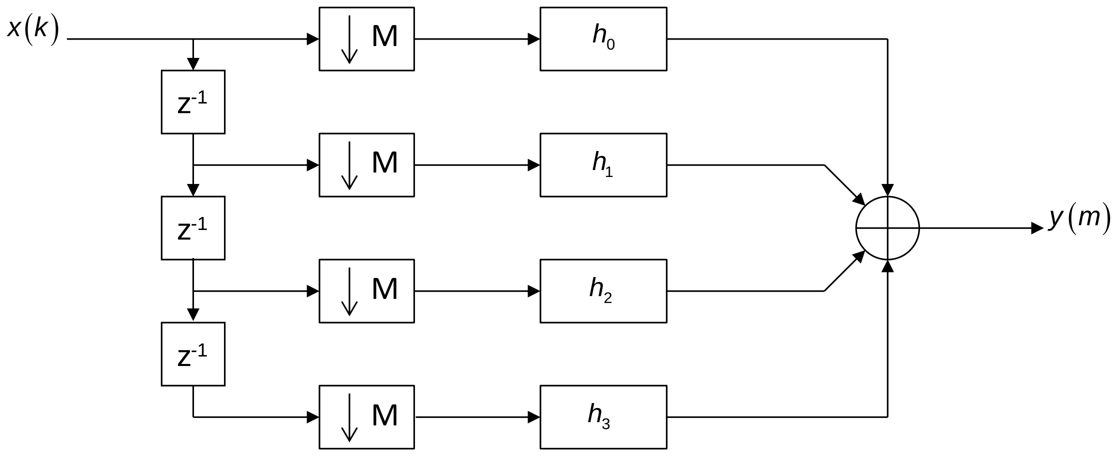


Figure 13: Polyphase decimator architecture

The derivation of this architecture is the reason the first decimator output sample was generated at time *k=3* as shown in Figure 11. If the first decimator output sample was generated at time *k=0*, then it is easy to show using Figure 11 that Equation 8 is modified to become:

$$\begin{aligned} y_0(m) &= h_o(m) \otimes x_0(m) \\ y_3(m) &= h_1(m) \otimes x_3(m) \\ y_2(m) &= h_2(m) \otimes x_2(m) \\ y_1(m) &= h_3(m) \otimes x_1(m) \end{aligned}$$

$$y(m) = y_0(m) + y_3(m) + y_2(m) + y_1(m)$$

Equation 10: Modified polyphase decimator equations

Equation 10 shows that the order of the input data phases is changed from {3, 2, 1, 0} to {0, 3, 2, 1}. This is still a valid decimator; however the order of the input data phases is less convenient for a practical implementation.

Note that a valid decimator can be obtained by generating the first output sample at time *k*=0, 1, 2, or 3. This gives *M=4* different versions of the same decimator. In general, the decimator output can be started by aligning the output with any one of the M phases in the polyphase input data.

An alternative architecture to Figure 13 is to replace the delay line structure prior to the filters with an anti-clockwise commutator as shown in Figure 14.

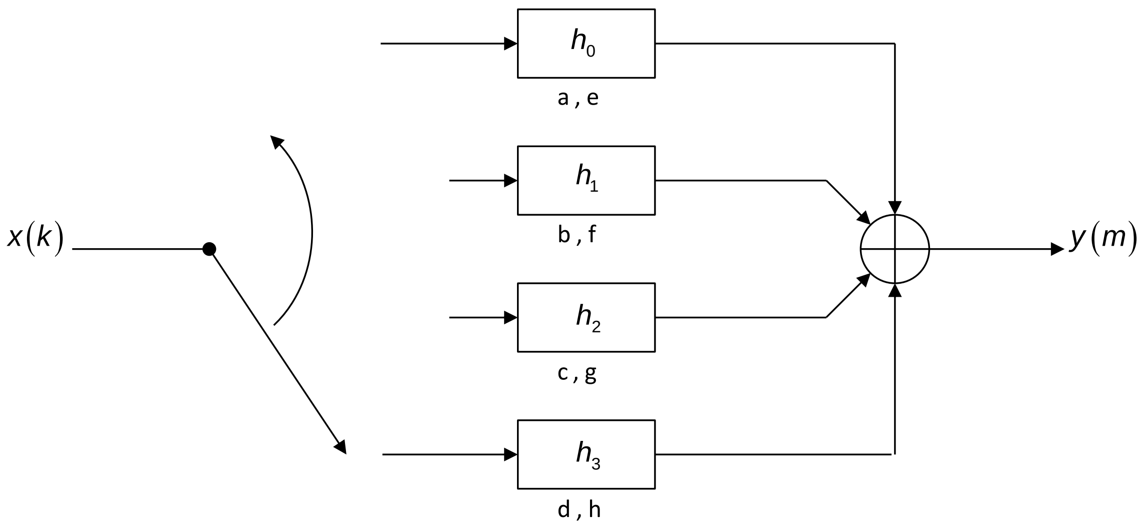


Figure 14: Polyphase decimator with anti-clockwise commutator

If the commutator starts by feeding the first input sample to the filter at the bottom of Figure 14, then it is easy to show that this architecture will produce the same output as Equation 9. Note that the only part of this architecture that operates at the higher input rate is the commutator, the filters and the final summation can all be clocked at the lower output sample rate. The commutator can be implemented in hardware using a demultiplexer, or if an input parallelism of *M* can be used, it can be removed entirely.

This is now an efficient architecture as only the input samples and coefficients required by each of the output samples are used; the filters in the architecture are also clocked at the lower output sample rate.

Similar to the interpolator described earlier, large decimation factors leads to a large degree of parallelism in the polyphase decimator architecture. The decimation can be implemented in stages by factoring the decimation factor as

$$M = M_1 M_2 \ldots M_n$$

Equation 11: Factoring the decimation factor

In a similar manner to the interpolator described previously, a decimator can be implemented using a single filter with time varying coefficients. This removes the need for the parallel architecture as shown in Figure 14, however the filter is clocked at the higher input sample rate.

Resampling with arbitrary rational factors

The two previous sections have described how efficient architectures for interpolation and decimation can be obtained. To achieve resampling by arbitrary rational factors, it is necessary to combine interpolation and decimation. However, in what order should these operations be performed?

The answer lies in the nature of the images and aliasing that occurs with interpolation and decimation. For the interpolator, it was stated that the images produced by the upsampling process can always be removed using a filter. For the decimator, if aliasing occurs, it cannot be removed by filtering. Therefore for resampling, interpolation is always performed first, and the filters are designed to *remove* images and to *prevent* aliasing.

A system level diagram for a resampler is shown in Figure 15:

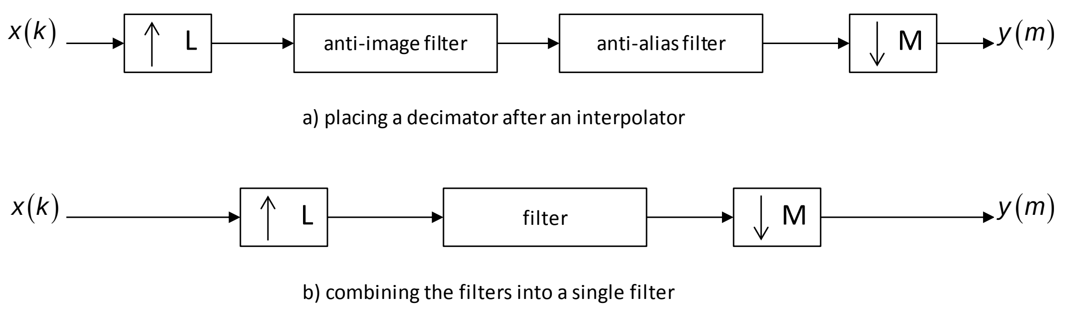

x(k)
↑ L
anti-image filter
anti-alias filter
↓ M
y(m)
a) placing a decimator after an interpolator
x(k)
↑ L
filter
↓ M
y(m)
b) combining the filters into a single filter

Figure 15: Resampler formed by combining an interpolator with a decimator

Figure 15 b) shows that the anti-image and anti-alias filters can be combined into a single filter. Note that the filter is designed for operation at the higher sample rate (after the upsampling). The main design criterion for this filter is the prevention of aliasing through the decimator. The passband of this filter is specified at the higher sample rate (Lf_{in}) and must roll-off sufficiently quickly so that aliasing does not occur at the output ($f_{out} = (L/M)\ f_{in}$) after the downsampling operation.

As with the interpolator and decimator, large sample rate changes or large integer ratios can be implemented in stages:

$$R = \frac{L}{M} = \frac{L_1}{M_1}\frac{L_2}{M_2}\cdots\frac{L_n}{M_n}$$

Equation 12: Factoring a large sample rate change

Note that not all of the stages need to be fractional, in most practical systems; *R* is usually factored into an integer component and a fractional component. So for example, for an overall reduction in sample rate, *R* may be factored as:

$$R = \frac{1}{M_1}\frac{1}{M_2}\cdots\frac{1}{M_3}\frac{L_1}{M_4}\cdots\frac{L_m}{M_n}$$

Equation 13: Factoring *R* into integer and fractional rate changes

To gain insight into how resamplers can be implemented efficiently, a worked example is given based on the interpolator and decimator examples described previously. A sample rate change of 3/4 is required using a filter with an impulse response given by *h(n) ={a, b, c, d, e, f, g, h, i}*. The same type of tapped delay line calculations as shown in Figure 5 and Figure 11 are used and lead to the tapped delay line values shown in Figure 16.

k	a	b	c	d	e	f	g	h	i	op
0	0	.	.	.	.	.	.	.	.	y0
1	x	0	.	.	.	.	.	.	.	x
2	x	x	0	.	.	.	.	.	.	x
3	1	x	x	0	.	.	.	.	.	x
4	x	1	x	x	0	.	.	.	.	y1
5	x	x	1	x	x	0	.	.	.	x
6	2	x	x	1	x	x	0	.	.	x
7	x	2	x	x	1	x	x	0	.	x
8	x	x	2	x	x	1	x	x	0	y2
9	3	x	x	2	x	x	1	x	x	x
10	x	3	x	x	2	x	x	1	x	x
11	x	x	3	x	x	2	x	x	1	x
12	4	x	x	3	x	x	2	x	x	y3
13	x	4	x	x	3	x	x	2	x	x
14	x	x	4	x	x	3	x	x	2	x
15	5	x	x	4	x	x	3	x	x	x
16	x	5	x	x	4	x	x	3	x	y4
17	x	x	5	x	x	4	x	x	3	x
18	6	x	x	5	x	x	4	x	x	x
19	x	6	x	x	5	x	x	4	x	x
20	x	x	6	x	x	5	x	x	4	y5

Figure 16: Tapped delay line values for a R=3/4 resampler

The outputs of the resampler are thus given by:

$$\begin{aligned}
y0 &= a0 \\
y1 &= b1 + e0 \\
y2 &= c2 + f1 + i0 \\
y3 &= a4 + d3 + g2 \\
y4 &= b5 + e4 + 3h \\
y5 &= c6 + f5 + i4
\end{aligned}$$

Equation 14: Resampler outputs

Like the decimator example described previously, the resampler output can be generated using a polyphase decomposition of both the input sample data and the filter impulse response.

Equation 14 shows that *L=3* different filters are required, and that input data is decomposed by a factor of *M=4* (i.e. coefficient *i* multiplies input sample numbers 0, 4, …, coefficient *f* multiplies input sample numbers 1, 5, … etc.).

Equation 14 leads to the architecture shown in Figure 17.

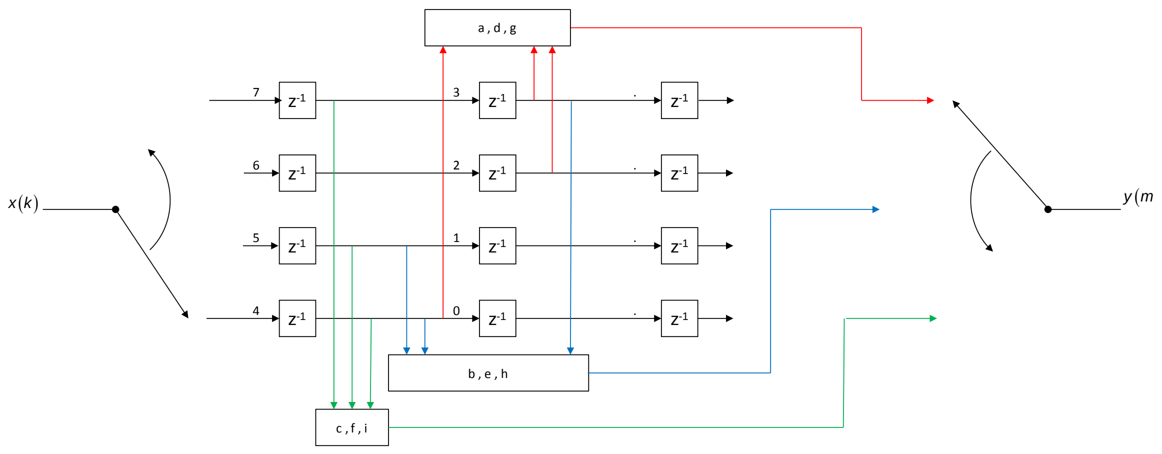


Figure 17: Resampler architecture for R=3/4

In this architecture, the input commutator is clocked at the input sample rate f_{in}, the delay lines and filters are all clocked at a rate of f_{in}/M, and the output commutator is clocked at the output rate $(L/M)f_{in}$. No part of this design is clocked at the interpolated rate of Lf_{in}. Note that unlike the interpolator and decimator described previously, the polyphase filters now lie *across* the phases of the polyphase input data samples.

In general, for a sample rate change *of R=L/M*, the input data samples are decomposed into *M* phases and *L* polyphase filters are required. What is not clear is how to align the filters with the input sample data for the general case. Figure 18 gives an insight into the alignment of the filters with the input data samples.

a)	8	x	x	7	x	x	6	x	x	5	x	x	4	x	x	3	x	x	2	x	x	1	x	x	0	input samples
b)	6	x	x	x	5	x	x	x	4	x	x	x	3	x	x	x	2	x	x	x	1	x	x	x	0	output samples
c)	8				6				5				4				2				1				0	last sample in delay line
d)	0				2				1				0				2				1				0	polyphase filter

Figure 18: R=3/4 resampler data alignment

Figure 18 a) and b) show the time alignment for the input and output sample data as seen at the interpolated sample rate of Lf_{in}. Figure 18 c) shows the last input data sample placed into the delay line of the filter (obtained from Figure 16) at the time an output sample is generated. This sample is also the last sample to be placed into one of the the polyphase filters as shown by Equation 14. The polyphase filter used to generate an output sample is shown in Figure 18 d). Figure 18 was used to construct the resampler architecture shown in Figure 17.

Figure 18 c) and Figure 18 d) can be calculated mathematically as shown in Figure 19.

a)	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	counter at rate L
b)	8	7	7	7	6	6	6	5	5	5	4	4	4	3	3	3	2	2	2	1	1	1	0	0	0	floor(x/3)
c)	8				6				5				4				2				1				0	decimate by 4
d)	0				2				1				0				2				1				0	mod(x,4), start phase
e)	6				5				4				3				2				1				0	counter at rate M
f)	24				20				16				12				8				4				0	multiply by 4
g)	0				2				1				0				2				1				0	mod(x,3), polyphase filter

Figure 19: Calculating the polyphase filter alignment

The shaded rows of Figure 19 show which polyphase filter aligns with which phase of the polyphase input sample data. In this example, the start phase and the polyphase filter order are the same. This is not true in the general case.

As a further example, Figure 20 shows the polyphase filter alignment for a *R=5/3* resampler. The calculations can be verified by carrying out the same analysis described above for the *R=3/4* resampler. Note that in this example, the start phases and polyphase filter order are now different.

In general, the start phases for the filter inputs are given by:

$$x_p = \mathrm{mod}\left(\left\lfloor \frac{Mp}{L} \right\rfloor, M\right) \quad 0 \le p \le M-1$$

Equation 15: Filter start phases for a R=L/M resampler

where $\lfloor . \rfloor$ denotes the *floor()* operation, i.e. rounding down to the nearest integer. The polyphase filter order is given by:

$$h_p = \mathrm{mod}(Mp, L) \quad 0 \le p \le L-1$$

Equation 16: Polyphase filter order for a R=L/M resampler

a)	5	x	x	x	x	4	x	x	x	x	3	x	x	x	x	2	x	x	x	x	1	x	x	x	x	0	input samples
b)	x	8	x	x	7	x	x	6	x	x	5	x	x	4	x	x	3	x	x	2	x	x	1	x	x	0	output samples
c)		4			4			3			3			2			1			1			0			0	last sample in delay line
d)		4			1			3			0			2			4			1			3			0	polyphase filter

a)	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	counter at rate L
b)	5	4	4	4	4	4	3	3	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0	0	0	0	floor(x/5)
c)		4			4			3			3			2			1			1			0			0	decimate by 3
d)		1			1			0			0			2			1			1			0			0	mod(x,3), start phase
e)		8			7			6			5			4			3			2			1			0	counter at rate M
f)		24			21			18			15			12			9			6			3			0	multiply by 3
g)		4			1			3			0			2			4			1			3			0	mod(x,5), polyphase filter

Figure 20: Filter alignment for a R=5/3 resampler

The resampler architecture of Figure 17 is efficient in the sense that only the input samples and filter coefficients required for an output data sample are used. This type of architecture is suitable for fixed resampling rates and where the values of *L* and *M* are small. This type of architecture is not suitable if the resampling rate is required to be programmable nor is it suitable for large values of *L* and *M*.

The key to implementing flexible resampler architectures is to recognise that each output sample is calculated using a different polyphase filter. Equation 15 and Equation 16 show how to align the polyphase filters with the input data samples. Note that in the resampler, the polyphase filters lie across the M phase input delay line. Therefore in practical implementations, the input data does not need to be polyphase (i.e. parallel): the important thing is to ensure the input data is correctly aligned with the polyphase filters.

RFEL Flexible Resampler

RFEL has developed a flexible resampler architecture that is capable of implementing arbitrary rational rate changes. The resampler can be configured to produce an overall increase in sample rate (upsampler mode) or an overall decrease in sample rate (downsampler mode).

For the downsampler mode, the available sample rate changes are given by:

$$R = \frac{L}{M} \quad \text{where}\, M \,\text{is fixed and}\, 1 < L < M$$

For the upsampler mode, the available sample rate changes are given by:

$$R = \frac{L}{M} \quad \text{where } L \text{ is fixed and } 1 < M < L$$

For large sample rate changes and in particular for large interpolation and decimation rates, the filters required for anti-imaging and anti-aliasing can become extremely large (billions of taps in the extreme). Filters this large become impractical to implement in hardware due to the large amounts of storage and routing required for implementing them. RFEL use virtual ROMs for generating the filter coefficient subsets required to generate the output samples. The virtual ROM calculates the required polyphase filter coefficients 'on the fly' and avoids the need for large amounts of memory for storing the filter prototype.

The RFEL resampler has the following properties:

- Large values of *L* and *M* are possible. For example, based on a 32 bit modulus and a 200MHz input sample rate, sub-Hz rate resolutions are achievable.
- The resampler can be implemented using resource sharing techniques that allow several independent input channels to be interleaved and processed by a single resampler. Each channel can have its own independent fixed or programmable rate change.
- For an overall reduction in sample rate, if the input sample rate is lower than the FPGA clock rate, filter arithmetic can be shared reducing FPGA resources even further.
- For multiple parallel channels requiring the same resampling factor, resource sharing can be used resulting in a architecture that consumes less resources than using multiple parallel cores.
- If *L* and *M* set to be the same value, then start phase (a third runtime programmable parameter in the range 0 to *M*-1) can be programmed to enable sub-sample delays to be implemented.
- *L*, *M* and the start phase can all be made runtime programmable.

For more information, please see www.rfel.com

References

[1] Digital Signal Processing, Principles Algorithms and Applications. J G Proakis and D G Manolakis, 4th Edition, Prentice Hall, 2007.

[2] Sample Rate Conversion for Software Radio. T Hentschel and G. Fettweis, IEEE Communications. Magazine., vol. 38, no. 8, pp. 142-150, 2000.

[3] An Economical Class of Digital Filters for Decimation and Interpolation, IEEE Transactions on ASSP, vol 29, no. 2, pp. 155-162, 1981.

[4] Analog Devices AD6620 data sheet, Analog Devices, 2001.

Author biography

Mike Porteous received an Electronic Engineering honours degree from the University of Leeds and a PhD from the University of Birmingham on cyclostationary signal processing. This research was funded by the Defence Science and Technology Laboratory (Dstl), part of the UK Ministry of Defence. He spent a total of 12 years working for Dstl working in Electronic Warfare research, specialising in signal processing techniques for Communications and Radar Electronic Surveillance (ES) systems. He has worked for RFEL for three years as a Principal Digital Systems Engineer and his current role involves the design and modelling of signal processing architectures with an emphasis on defence applications.