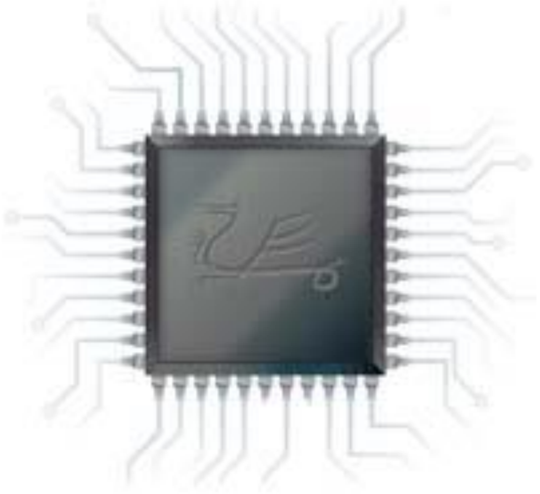
Convolution and Finite Impulse Response Filters



Simon Southwell

September 2023

Preface

This document brings together two articles written in September 2023 and published on LinkedIn, that is a look at Finite Impulse Response filters, based on convolution, with a practical HDL implementation for audio signal processing and a program for designing the tap values for these filters.

Simon Southwell
Cambridge, UK
September 2023

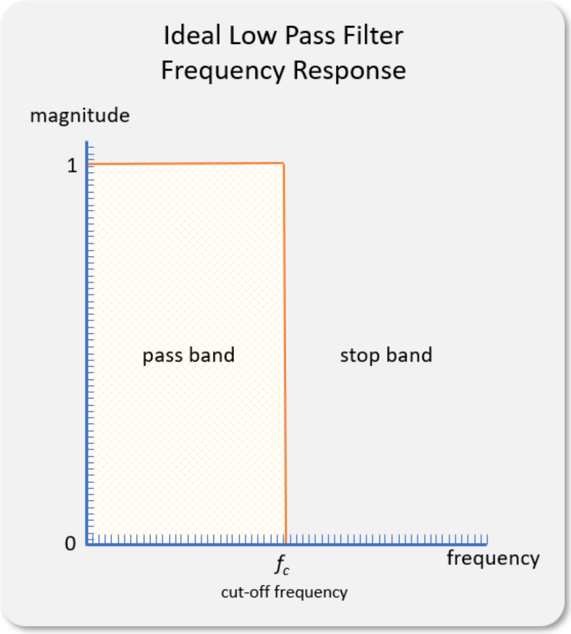
Contents

Part 1: Convolution and an HDL Implementation

Introduction

In this document on convolution and finite impulse response filters I want to move into the world of digital signal processing. Strictly speaking my articles on data compression, including image compression with JPEG, fall under this category but, I think, this document on finite impulse response filters is what most people would consider as definitely in the realm of DSP.

Finite impulse response filters (or FIRs) are based around the process of 'convolution', and we will have to have a look at some equations to explain what's going on, but I promise that there is nothing particularly taxing about this. When we get to a practical solution, we will be down to multiply-and-accumulate and nothing more complicated than that. We will also look at impulse responses—that is, the output you get from a system when an impulse (or delta) is input—which tend to be infinite (assuming unlimited precision). It turns out that if you convolve the impulse response of the desired filter with your signal it will filter that signal as per the filter characteristics. In fact, convolving in the time domain is the same as multiplying in the frequency domain (and vice versa). So, if we wanted a low pass filter, say, in the frequency domain this means that all frequencies from 0 to f_c we want to multiply by 1, and all frequencies above f_c we want multiplied by 0. This is the ideal frequency response, and the diagram below shows this for a low pass filter.

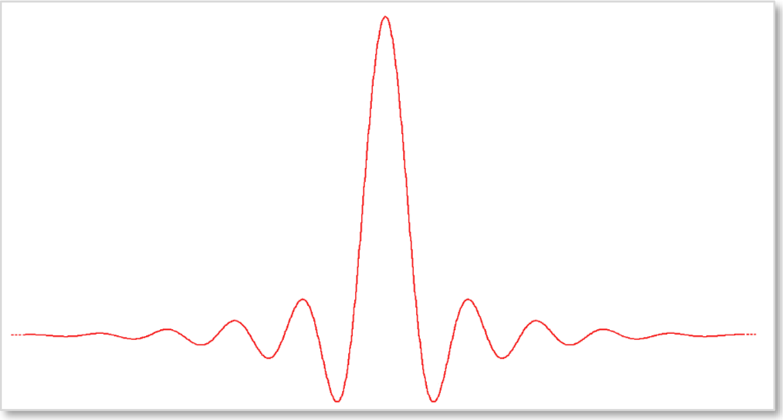


Even more handy, multiplying everything by 1 in the frequency domain is equivalent convolving a 'delta function' in the time domain—i.e., an impulse. Thus, if we know the impulse response of the filter, we can convolve this with our signal and it will filter it, just as if we had taken the signal into the frequency domain, multiplied it by our frequency response (by 1 for low frequencies, and by 0 for high frequencies), and put it back in the time domain again. In fact, this is another way to filtering which is outside of the scope of this document, but we might visit this again when looking digital fourier transforms and the fast fourier transform algorithm in the future. So, we 'simply' need to work out the impulse response values of our desired filter and convolve these with our signal. We will look at this in this document and a handy open-source program is provided to help us do just this.

In practical solutions, dealing with infinity is somewhat of problem, but we can truncate things as values tend towards zero, making things finite—and this is where the finite part of the FIR comes from, with the 'IR' from the impulse response. The truncation of impulse response values effectively gives a 'window' on that response, with everything outside of that window set to zero. Now, simply truncating the values causes unwanted artifacts and things can be improved if the window on the impulse response is shaped in a more gentle type of roll-off. There are many types of window functions that have been discovered/developed over time that improve the situation of reducing the artifacts of truncating an infinite impulse response, and we shall be looking at some of these and what the trade-offs are. The program I mentioned implements many of these (though we shall restrict ourselves to just a few) and so the different performances of the window functions can be explored.

Another source of undesired artifacts come from the fact that we can't have infinite precision in the values we use for our impulse response but must 'quantise' these to be of finite precision, with a set number of bits, such as 8- or 16-bit values. As we shall see, this tends to limit the amount of attenuation in the 'stop-band' that can be achieved, and the number of bits chosen needs to match the requirements of the filter being implemented.

Finally, the impulse responses of ideal filters are made up with '*sinc'* functions, or combinations thereof. A *sinc* function has a general form of *sine(x)/x*, which is not too complicated. The diagram below shows a plot for a generic *sinc* function.



As can be seen, the *sine* component provides the oscillations and the 1/*x* component gives the decay with the plot centred on 0, thus negative x on the left and positive x on the right. The mathematicians amongst you may have spotted the if *x* is 0 the function is infinite, but that is not what's shown. As *x* tends to 0 the plot approaches 1, and so this limit is the value at 0. (A bit of a mathematical cheat, I known, but I'm not mathematician enough to know why this is okay.)

Using these functions, as we shall see in the second part of the document, gives the infinite impulse responses of our ideal filter and multiplying these with a window function to make finite gives us our finite impulse response. Thus, these are sometimes called windowed-sinc filters.

Now, don't worry if this introduction has introduced a lot of concepts in a short space of time as we will revisit these in some more detail in the rest of the sections. And, as mentioned before, when we get to an implementation in HDL, we will only be doing multiplication and addition. All the messing about with trigonometrical functions can be done off-line to calculate the values we will need in the logic, and the provided program will do this for us and we could use this 'turn key', though I'd very much like you to understand what's going on.

The document is divided into two parts. In this first part we will introduce convolution for digital signals and move straight to discussing an HDL implementation. By the end we will have a solution that can be used to construct an FIR but won't know what values to configure it with to do the filtering that we desire. That will have to wait for the next part of the document, where we will look at *sinc* functions and their use in constructing impulse responses for our filters and at window functions to modify the infinite impulse responses to a finite set of values.

Before diving straight into convolution, I want to talk briefly about the general characteristics of FIRs and their advantages and disadvantages.

FIR Characteristics

There is not space in this document to give a full treatment to the alternative to FIRs, which are infinite impulse response (IIR) filters. Without going into too much detail IIRs are characterised by the current output being a function of both the current input and also by previous output values (and inputs). This gives them a potential infinite impulse response, which has many advantages. Compared to FIRs there are fewer coefficients required and thus smaller memory requirements to get similar characteristics in terms of cut off frequency transition and stop-band attenuation. Because there are fewer coefficients, the latency is smaller than for FIRs. Also, IIRs can be constructed in such a way as to be equivalent to analogue filters in terms of mapping between the s (analogue) and z (digital) planes (for those that know what this means—which we can't cover here). This is handy if digitizing a previous analogue system. There is no such equivalent mapping for FIRs.

So, if I were a salesman selling FIRs, I've just convinced you to go down the IIR route, right?

Well, unlike FIRs, IIRs have non-linear phase characteristics and in some applications, such as processing audio or biometric data, this can be quite undesirable. Due to the fact that IIRs have feedback paths, using previous output values, they can become unstable, whereas FIRs can't become unstable for any input signal since they are a function of input values and impulse response only. Although this document will limit what filter response we will investigate, none-the-less, FIRs can have an arbitrary frequency response. FIRs must handle numerical overflow to some extent, but this is easier than with IIRs, making IIR design more complex, and the effects of quantisation more severe.

So which one is used for any given application will depend on the requirements, resources, complexity, and all the normal engineering trade-offs in choosing between competing solutions. In this document, we will explore FIRs.

Convolution

I mentioned in the introduction that convolution was at the heart of FIRs, and so we need to understand this. However, since we will be implementing a logic based convolution solution, I want to move straight on to that implementation afterwards, in a break from a traditional text ordering, because, once we have this solution, making a desired filter from it is simply a matter setting the ‘right numbers’, and the next part of the document will deal with this. Thus, if one is only interested in a filter implementation, and are happy to get the right numbers from the provided winfilter program, then you can skip the part—though I genuinely hope you don’t.

What is Convolution?

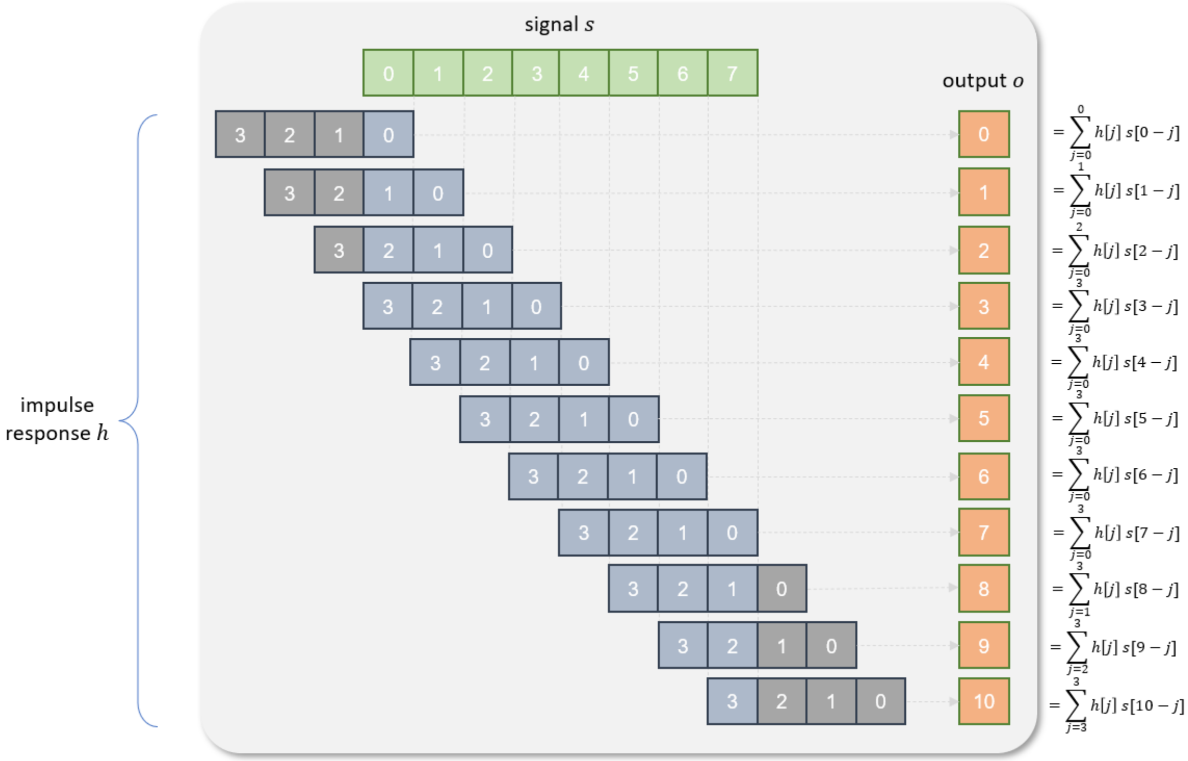
The general form of the convolution of two signals (e.g., an input signal s and an impulse response signal h) is given as:

$$o[n] = s[n] * h[n]$$

If the input signal has N points and the impulse response has M points, then the resultant output (o) will be an N+M-1 point signal. What does this actually mean? Well, each output $o[i]$ is calculated by multiplying the impulse response values from 0 to M-1 with the signal values from i to i-M-1 and adding all these up. This is summarised in the equation below.

$$o[i] = \sum_{j=0}^{M-1} h[j]\, s[i-j]$$

In other words, if we think of our impulse response as an M point array of values, and our signal as an N point array of signals, then for each output we multiply the M array elements of the impulse response with the last M elements of the signal, starting from i and running backwards, and add all the M multiplication results together. If the signal is finite in length (N), then if a signal value is indexed outside of its valid range, then it is assumed 0, and the multiplication is 0, and thus no effect on the output value. Let’s try and picture this. The diagram below has an 8 point signal to be convolved with a 4 point impulse response. This should yield an 11 point output.

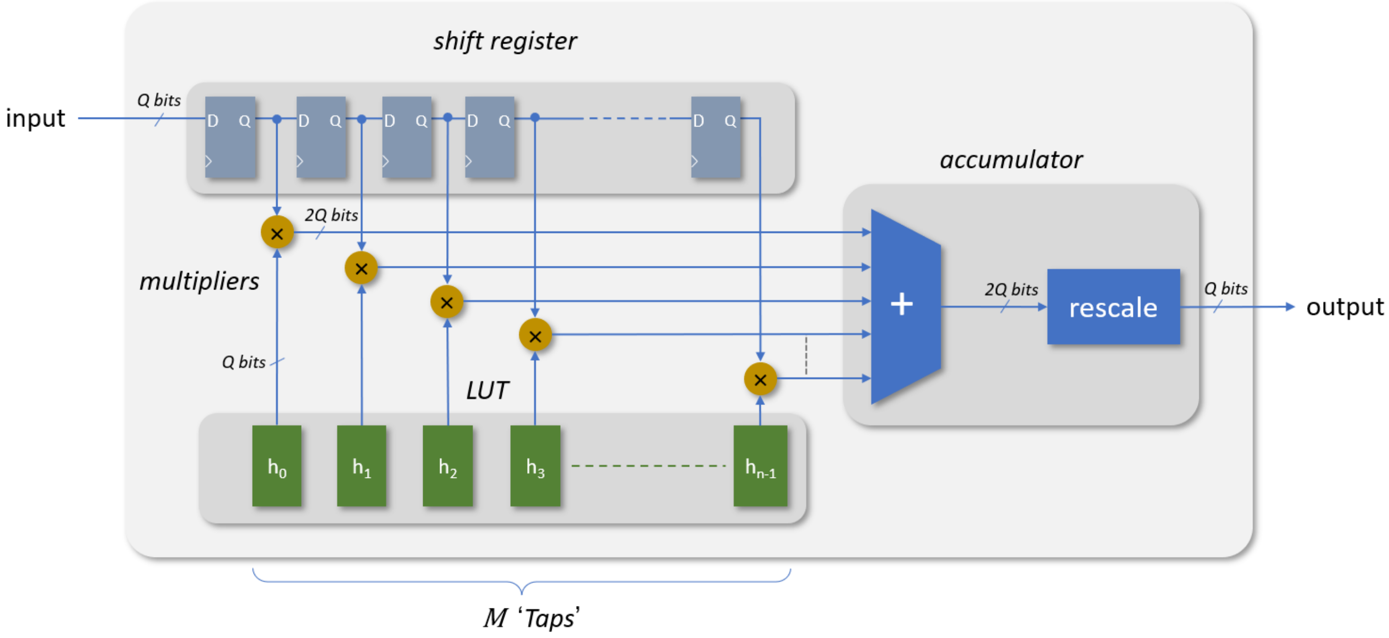

signal s
0 1 2 3 4 5 6 7
output o
impulse response h
3 2 1 0
0 1 2 3 4 5 6 7 8 9 10
$= \sum_{j=0}^{0} h[j]\, s[0-j]$
$= \sum_{j=0}^{1} h[j]\, s[1-j]$
$= \sum_{j=0}^{2} h[j]\, s[2-j]$
$= \sum_{j=0}^{3} h[j]\, s[3-j]$
$= \sum_{j=0}^{3} h[j]\, s[4-j]$
$= \sum_{j=0}^{3} h[j]\, s[5-j]$
$= \sum_{j=0}^{3} h[j]\, s[6-j]$
$= \sum_{j=0}^{3} h[j]\, s[7-j]$
$= \sum_{j=1}^{3} h[j]\, s[8-j]$
$= \sum_{j=2}^{3} h[j]\, s[9-j]$
$= \sum_{j=3}^{3} h[j]\, s[10-j]$

Each box is an element in an array for the signal (green), impulse response (blue) and output (orange). Each output element is calculated as the sum of the overlapping signal and impulse response elements multiplied together. I've tried to relate this to the previous equation, showing the specific summation of multiplications on the right by each output element. So, we've 'swept' the impulse response across the signal for each output calculation, multiplying and accumulating the overlapping elements. Note that the indexes for the response and the signal run in opposite directions, and this is important for the general case of convolution. It is likely that a signal will be asymmetric, and the secondary signal (an impulse response for our purposes but may not be for other uses of convolution) may also be asymmetrical. Thus, to convolve, one or the other signal must be reversed. It doesn't matter which one, as we could treat the impulse response as our signal, and the signal as the values to be convolved. As I said, the two sets of values need not be for FIR purposes. As we shall see later, the filter impulse responses will be symmetrical, so it won't matter, but an implementation should reversal do this so that it can be used for any convolution purpose.

What About Arbitrarily Long Signals?

The example in the diagram assumed a finite set of signal values, but in a real DSP usage, the signal will probably be an unending set of samples to be processed. Notice from the diagram that the impulse response only ever overlaps the signal values over a limited range. Basically, the length of the impulse response. So, so long as the values to be convolved with the signal are of finite length then, if we remember the last M signal values, we can keep producing output values indefinitely.

Now this is beginning to suggest a solution in logic. A multi-bit shift register is good for remembering the last M values of an input. For our purposes, the impulse response we want to convolve with our signal is fixed, and so we just need a look-up-table of values. Then, to convolve the impulse response with the signal for a new output value, we simply multiply and accumulate the elements of the LUT and the shift register. When a new input arrives, and the shift register is shifted, with the oldest value disposed of and the new value added, the system is ready to generate a new output. It's that simple!



There are some practical considerations to take into account. The above diagram suggests that there are as many multipliers as there are entries in the LUT (known as 'taps'). A practical impulse response might need to have as many as 256 taps or more, and the accumulator would have to be able to add that many multiplication results at once. If the rate of new input samples is low relative to the maximum clock frequency that, say, the multiplier logic can run at, then the implementation can do a multiply and accumulate once per cycle for M cycles, and if that period is less than the input sample rate, only one multiplier and one adder is required. In between this, if M cycles is too long, then two circuits running in parallel, doing half the taps (bottom/top or odd/even, for example) can be employed. This would take two multipliers and two adders plus an additional adder to add the two final results.

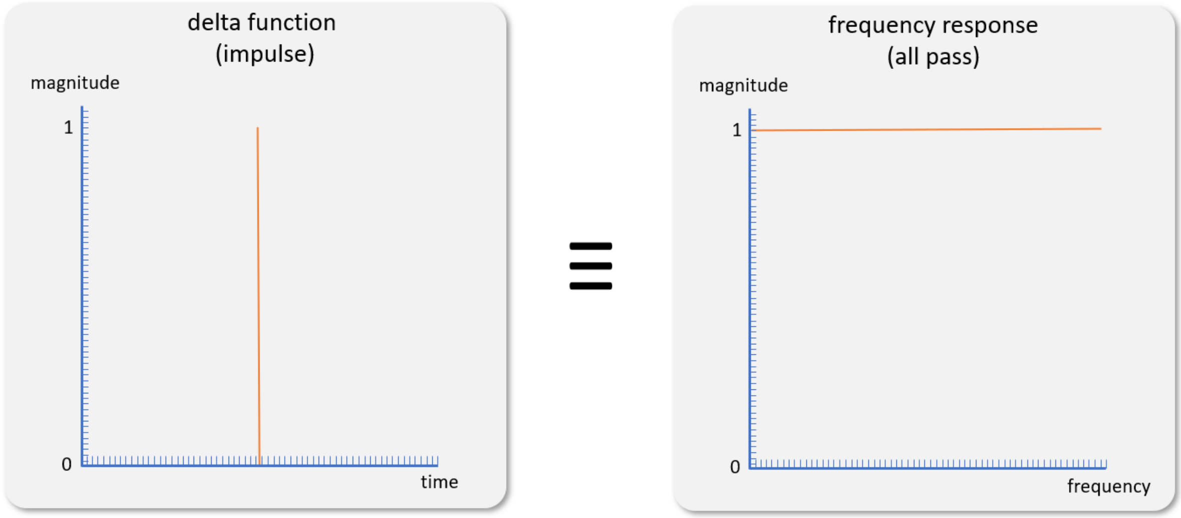
Hopefully you can see that this bifurcation can be extended for quarter, eight, sixteenth etc. parallel calculations, trading off speed against logic resources, depending on the requirements. In the example implementation we will discuss later, the clock rate is sufficiently fast that we can use a single multiply-accumulate logic block to process all the taps with time to spare for the target functionality.

Another consideration is the quantisation Q of the signal and 'tap' values. The choice here is dependent on the requirements and could range from double precision floating point (64 bits) to 8 bit integers or fixed point values. Having chosen the number of Q bits, it must also be noted that multiplying two numbers of Q bits wide gives a result width of $2Q$ bits and the adder must be able to take this size input. Also, when adding numbers, the result can be one bit larger than the inputs (let's assume they're both the same width). Thus, *potentially,* to avoid overflow (or underflow), the adder should allow additional bits for each addition—i.e., an additional M bits. In practice, this becomes impractical, particularly if M is configurable (as our example will be, via a parameter). Various ways of dealing with this are available. Firstly, do nothing and let the accumulator rollover. This may seem 'lazy', but it might be that upstream processing makes certain range guarantees that means the accumulation calculation can't over- or under-flow, and additional logic to deal with it is redundant. The next level is to detect under- or overflow and flag an error. This might come in the form of detecting that an addition would change the sign of the result when the input signs match. I.e., adding two positive numbers should not result in a negative number and, similarly, adding two negative numbers should not result in a positive number. Having detected over/underflow one could then take a further step and 'saturate' the result so that the overflowed or underflowed result is discarded to be replaced with the maximum positive or negative number as appropriate.

Having dealt with these problems, we have a result that is $2Q$ bits wide but, for many applications, we want the result to be Q bits wide to match the input. We could just take the top Q bits from the result, but we can do better than this and introduce less quantisation noise if we add the most significant bit of the truncated bits to the rescaled value to round up. This, then, picks the nearest rescaled value to the original.

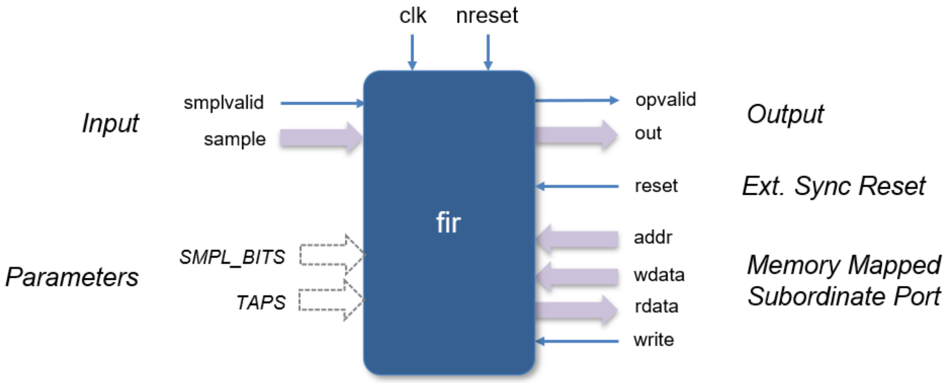
Earlier in the document I made the bold claim that convolving a signal with the delta function (an impulse) in the time domain results in multiplying all frequencies by 1 in the frequency domain—in other words and all-pass filter, which doesn't alter the signal. Our solution gives us an insight into why that might be. Imagine our impulse response is a delta function, with just one of the taps set to 1.0 (or the equivalent in binary). Now, as we sweep this over our signal, the single tap with the 1 in it will pick out only one value from the signal and will pick each one in turn as it sweeps through. All other values will be masked to 0, being multiplied by 0 from the impulse response. The output, then, being the addition of all the results, and all but one of the multiplications being zero, will simply reproduce the input value unaltered. This is true for whatever input signal we choose to put into the convolution. Thus, this acts as an all pass filter.

Conversely, if we make our signal a delta function (impulse), when we convolve this with our impulse response tap values, it will reproduce the impulse response at the output, as it picks one value at a time as it sweeps through. So here we see, in our solution, that convolving in the time domain is the same as multiplying in the frequency domain, and that an impulse in the time domain is the same as an all pass filter.



HDL Implementation

I have put together a configurable Verilog solution, firfilter, on github. Firstly, let's have a look at its ports. The diagram below shows the `fir` module block diagram with its signals and parameters.



The module has a clock (`clk`) and power-on reset (`nreset`, active low). It also has a synchronous external reset (`reset`, active high) so that external logic can reset the module outside of POR. There is also a simple subordinate memory mapped bus port which allows the tap values to be programmed from an external source.

The other two ports consist of the input samples (`sample`) with an accompanying valid (`smplvalid`), and the output values (`out`) with its own valid signal (`opvalid`). The number of bits for the input and output values is determined by the `SMPL_BITS` parameter (with a default value of 12). Needless to say, these values are going to be treated as signed numbers. The `TAPS` parameter (with a default of 127) determines the number of logical taps the impulse response LUT will have. It will also determine the width of the address (`addr`) on the memory mapped port to be able to index all the tap entries.

Actually, as this implementation is targeting an FIR implementation, and because the impulse responses will be symmetrical about the centre value, an optimisation is made to only store half the data, plus the centre value. Thus, the actual storage defined is `TAP/2+1` tap entries and logic will be used to recreate the full impulse response. If the size of the LUT is not a limiting factor, then this need not be done, saving on a little bit of logic, and making the implementation a more generic convolution solution. However, I wanted to illustrate some optimisation considerations that might be required when looking at the logic.

One other deviation in this implementation from the generic diagram seen earlier is that the taps are actually `SMPL_BITS+1` wide. This allows for the equivalent of 1.0 to be stored, otherwise the maximum value is equivalent to 0.111111111111... (in binary) which gives a slight scaling to the output (an attenuation), the degree of which is dependent of the magnitude of `SMPL_BITS` (the larger, the less of an error). This may not be an issue for some applications, but this will make things easier to cross-correlate results to what we've discussed and not introduce any scaling errors.

Indexing the Tap Values

Before jumping into the convolution logic, we need some logic to extract values from the tap LUT (`taps`—an array of signed values). These will be programmed with the impulse response values from index 0 to N/2. So, for the default `TAPS` value of 127, this is 0 to 64. When a new input value arrives (`smplvalid` is 1 for a cycle), a register `count` is set to `TAPS` and counts down to zero, decrementing each clock cycle. For the first half we want to index the LUT from 0 to N/2, and then count back down from N/2-1 down to 0. A combinatorial bit of logic does this calculation assigning a wire `tapidx`:

```
wire [$clog2(TAPS)-1:0] tapidx = (countdlyd > (TAPS+1)/2-1) ?
                                 ~countdlyd :
                                  count[$clog2(TAPS)-2:0];
```

Note that `countdlyd` is the `count` value delayed by one cycle, which is equivalent to `count+1` when the `count` is active, saving an adder, aligning the value, and relieving the timing.

Multiply and Accumulate

The heart of the convolution is a multiply and accumulate function. In many FPGAs, 'DSP' functions are included. For example, the Intel Cyclone V or the AMD Zynq 7000 series FPGAs, both fairly low cost solutions. These DSP blocks are basically multiply and accumulate blocks (though they can be used as just multipliers)—just what we need. One could instantiate these blocks as modules, but the synthesis tools can infer the functionality and construct the blocks we need for us. It does help, though, if we construct the logic to look like a multiply and accumulate equation. In the `fir` module, the synchronous process has the code:

```
if (countdlyd)
begin
    accum <= (tapval * validsample) + accum;
end
```

The `accum` register is just what you'd expect, accumulating the values over the convolution. The `tapval` is the registered extracted LUT value (`taps[tapidx]`). The `validsample` signal is registered version of the shift register (`smpls[]`) values, as indexed by `count`, but also validated as containing a value. After POR, the shift register has no valid values and could be random. POR could reset them all, but this could be a lot of bits (TAPS × *Q*+1) so, in this implementation, a register, `smplsvalid`, of `TAPS` bit wide is cleared on reset, and is used as a single bit shift register, filling with 1s at each shift in of the input samples. Thus, `validsample` is the indexed shift register value if the equivalent bit of `smplsvalid` is set, otherwise it's `0`. The update of `accum` happens whilst the count value (delayed) is non-zero. It gets zeroed each time a new input sample arrives.

Rescaling

Lastly, when in the last cycle, the output is updated with the `accum` value rescaled. The code fragment below shows this:

```
if (lastcycle)
begin
    out     <= accum[SMPL_BITS*2-1:SMPL_BITS-1] + accum[SMPL_BITS-2];
    opvalid <= 1'b1;
end
```

Note that the `opvalid` is set to a default value of `1'b0` at the beginning of the synchronous process so that, when set here, it pulses for just one cycle, being cleared on the next. And that's all there is to it. We now have the basis of a finite impulse response filter, if only we knew what to put in the tap values. That will be the subject of the part 2 of the document.

With this design, having a single multiply and accumulate function, a new output takes `TAPS` + 3 clock cycles to calculate. The extra three cycles allow for registering at both the input and output, and around the multiply-and-accumulate function. Thus, for a 100MHz clock this would be, for the default `TAPS` size of 127, 1.3μs or a rate of 769KHz. This 1.3μs is also the latency through the filter.

Conclusions

In this first part of the document, digital convolution was introduced showing how two signals can be convolved in the time domain and what this means in the frequency domain. By making one of the signals a finite impulse response we can use convolution to filter a signal.

A generic solution was explored, where there were no restrictions on resources such as memory, multipliers, or the number of terms in an addition. This was used to demonstrate an intuitive understanding of why convolution in the time domain is equivalent to multiplication in the frequency domain, and why a delta function (or impulse) in the time domain is an 'all-pass' filter—i.e., multiples everything in the frequency domain by 1.

From this idealised solution some practical constraints were formalised, and a Verilog based implementation was discussed. The solution requires only multiply and accumulate functionality over and above normal logic, and the solution is, I hope, simple and easy to understand. We left this HDL implemented but with no knowledge of how to configure this logic to act as a filter. We still need the numbers.

In part 2 of the document, I want to concentrate on generating the tap values required to turn our HDL implementation into a filter. It will cover generating impulse responses using *sinc* functions, 'windowing' the *sinc* function to make finite, designing these to meet our requirement, and analysing these to validate the responses. To do this we will take a specification from digital audio processing as an example, for filtering 48KHz samples with a 4 × oversampled low-pass filter, a cut off at 20KHz and a stop -band attenuation target of -60dB. We will use my `winfilter` program to help us do this but will discuss the calculations behind all this for a fuller understanding.

Part 2: Sinc Functions and Windows

Introduction

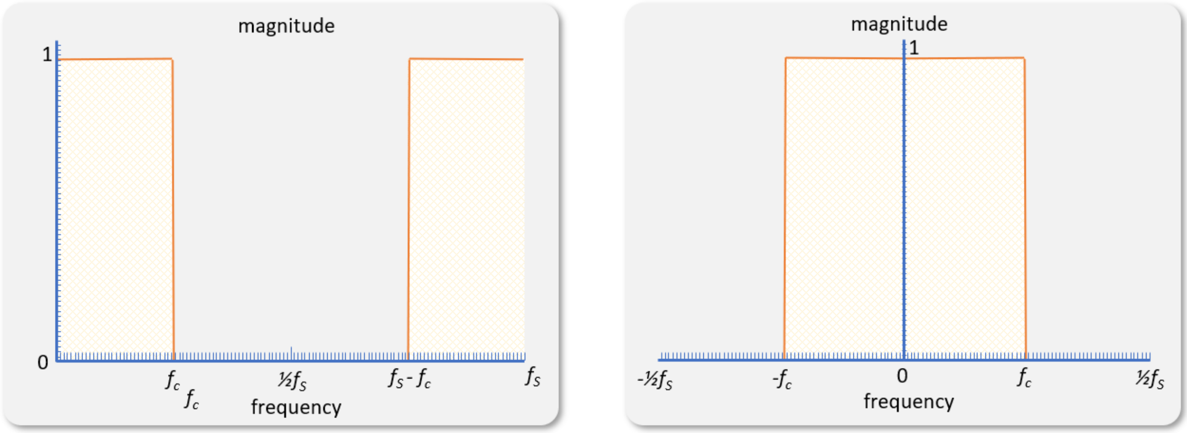
In part 1, I introduced the subject of finite impulse response filters (FIRs) and had a look a convolution. An HDL implementation was then discussed showing just how easy it is to construct logic to implement a solution for convolving two signals—for our purposes a finite impulse response with a continuous stream of digital signal samples. However, I left things hanging as the implementation needs configuring its 'taps' (or coefficients) to program the impulse response into the logic, and we don't know how to do that yet.

In this part of the document I want to look more closely at the *sinc* function and explain why we're messing around with it, and then window functions to turn the infinite nature of the *sinc* into a finite set of values we can configure into the logic implementation. Using a digital audio example to set some requirements on a practical design, we will then use `winfilter` to generate tap values, looking at the effects of the different window functions, quantisation bit width and the number of taps has on things like the filter's roll-off rate, stop-band attenuation and ripple in the pass-band. We will be able to plot the frequency response, the phase, the impulse response, and even the window function to visualise the functions and the filter performance. Although we will explore a low pass filter in detail as a practical example, we will also look briefly at highpass, bandpass and bandstop filters as well, and how we can construct these. I mentioned in the first part that, actually, an arbitrary frequency response can be implemented using FIRs, but space restricts what we can discuss here, but I will give some references to allow adventurers to continue to explore on their own.

Once we have some taps values for our logic implementation, we can then test how the filter actually performs against our predictions for the digital audio example, and some simulation results will be analysed and cross-referenced. A simulation environment is provided with the Verilog implementation which should be adaptable for a preferred logic simulator.

The *sinc* Function and Impulse Response

I mentioned in the first part of the document that the impulse response for a low pass filter is a *sinc* function of general form *sine(x)/x*. Firstly, this isn't strictly true as it's *sine(πx)/πx*, and secondly why is it a *sinc* function? In the introduction to the first part of the document there was a diagram of an idealised low pass filter. This diagram is actually only half the story. It only plots the frequency axis from 0 to the sample frequency divided by 2 (f_s/2). Without going into too much detail, when sampling data at a given sample frequency, only signals up to half that rate can be represented—this is the Nyquist limit. Any higher frequencies get 'aliased', and fold-back into the lower frequencies. What we see when we plot the entire frequency range is something like that shown in the diagram below:



If we simply extend the diagram from the first part of the document to f_s, we get a plot like that shown on the left of the diagram above, which is a reflection of the first half. If we kept plotting higher and higher frequencies this would actually repeat periodically, both in the positive and negative directions. The diagram on the right, then, shows what you get if plot from - f_s/2 to + f_s/2—i.e., a rectangle. Now, the fourier transform of a rectangle is a *sinc* function. I'm not going to prove this (others who know more mathematics than I do have done so already), but this is why we want a *sinc* function for our impulse response. The phase for a *sinc* function centred around 0, is 0 for all frequencies. Even if the *sinc* function is offset from 0 the phase remains linear, with a slope a function by the offset. This linear phase is one of the defining characteristics of an FIR, as mentioned in the first part. We will look at a phase plot when we discuss the audio example.

The diagrams above don't specify what the sample or cut-off frequencies are, but as we increase and decrease the passband as a proportion of the sample frequency, we can get a ratio f_c/f_s. This can now be mapped into our *sinc* function to get the impulse response for the cut off frequency desired as a fraction of sample frequency:

$$h[i] = \begin{cases} \dfrac{\sin\left(2\pi \dfrac{f_c}{f_s} i\right)}{i\pi} & when\ i \neq 0 \\ \dfrac{2f_c}{f_s} & when\ i = 0 \end{cases}$$

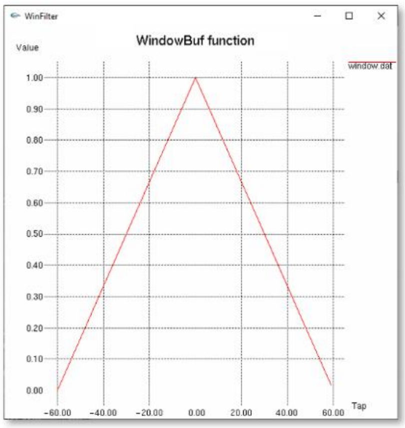
So, f_c/f_s can be anything from 0 to 0.5 to determine the low pass filter pass- and stopbands. For example, if our sample rate was 100KHz, and the desired cut off frequency is, say, 12.5KHz, then f_c/f_s = 0.125. Plug this into the equation and we can calculate the impulse response for all values of *i*, to plus and minus infinity.

The value when *i* is zero isn't just 1—previously, in the first part of the document, we used nice round numbers. The *sine* function becomes linear the nearer we approach zero radians—i.e., sin(θ) → θ. Therefore, removing the *sine*, and dividing by $i\pi$, leaves the value as shown in the equation for *i* at zero. So now we'd better deal with infinity.

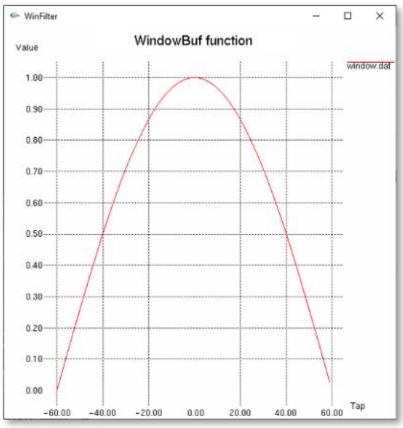
Windowing and Window Functions

In the first part of this document, I mentioned that truncating the infinite impulse response gave a 'window' on that response. I also mentioned that this simple truncation adds undesirable artifacts and that things can be improved with window functions that are a more rounded roll-off. The simple truncation, in fact, is known as a 'uniform' window. Over the years, researchers have experimented with various functions to improve the response from the basic uniform window, from a triangle (aka Bartlett window), a cosine window, a raised cosine window (aka Hamming window), through to some quite elaborate functions such as a Kaiser window. We can only look at a few, but the `winfilter` program supports quite a few, and the functions' definitions are all in the source code (see `windows.c`). Starting from `winfilter`'s default settings, let's look at a few example windows:

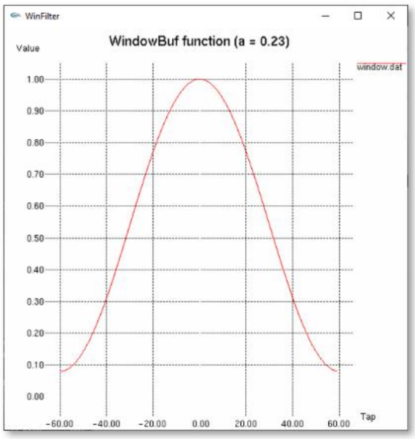
Bartlett



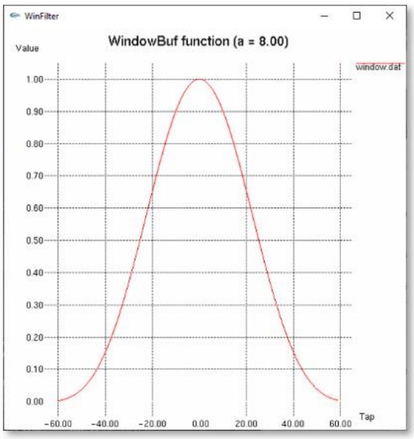
Cosine



Hamming (α = 0.23)



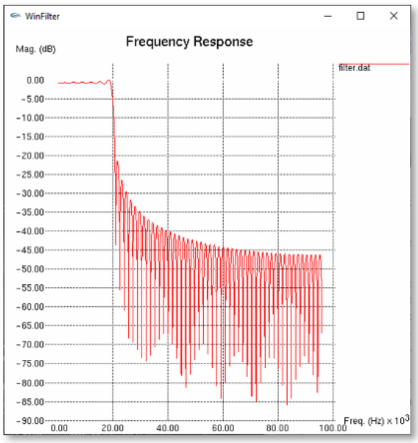
Kaiser (α = 8.0)



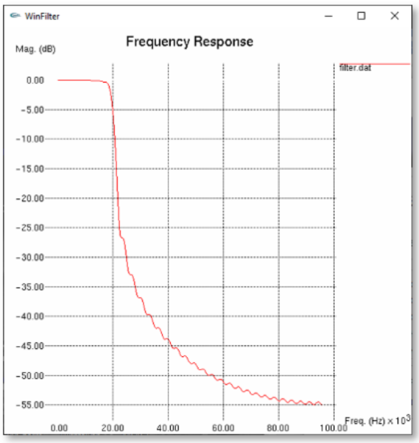
As you can see, the shapes of these windows more or less have a similar form, starting at 1 for the centre point and reducing either side towards 0, and it gets harder to spot the difference between them, though they do make a difference as we shall see. I haven't plotted the uniform window, as I assume you can picture multiplying by 1 over the number of taps and 0 everywhere else.

If we now take the *sinc* impulse response that we calculated from the last section and multiply it by the selected window function, we get the finite windowed-sinc filter tap coefficients (sometimes know as the filter's kernel) with which we need to program our HDL implementation. Plotting the frequency responses (in dBs) for these windowed-sinc impulse responses (including for the uniform window), we get the following results:

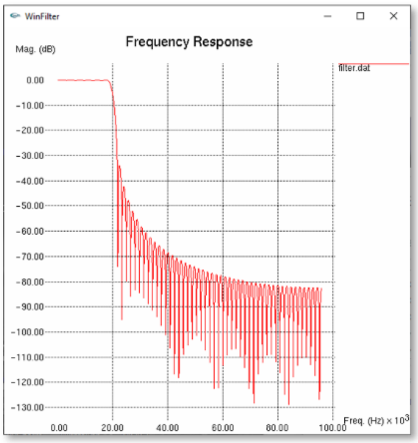
Uniform



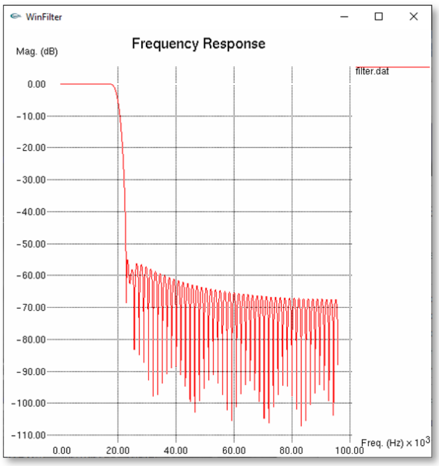
Bartlett



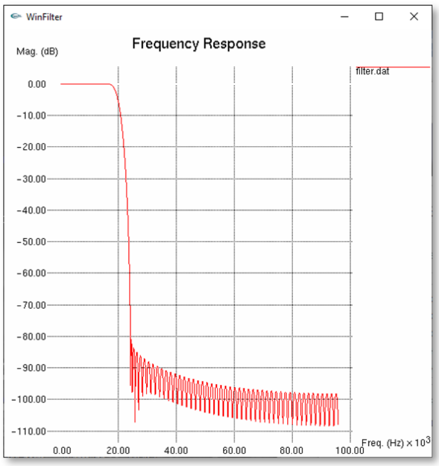
Cosine



Hamming (α = 0.23)



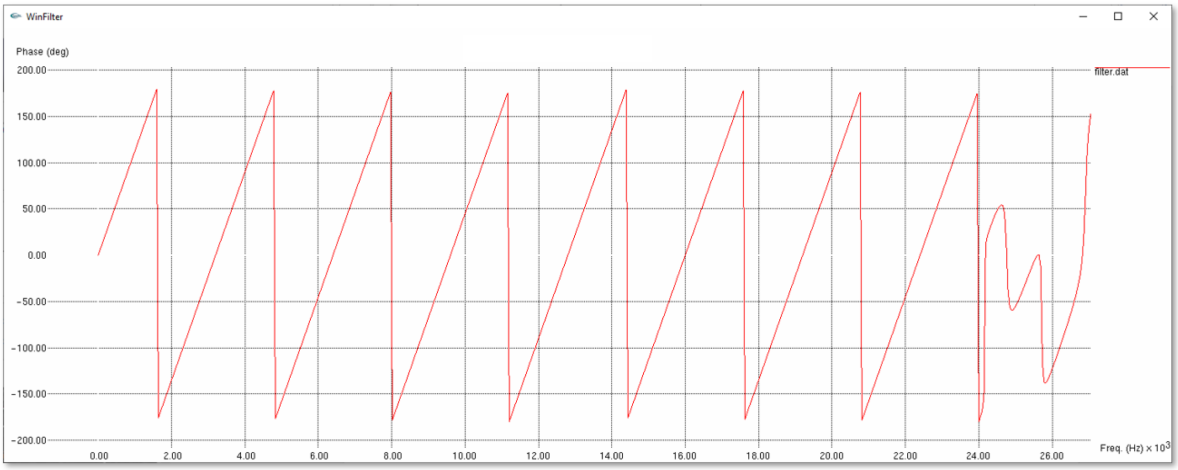
Kaiser (α = 8.0)



I've chosen the order of the types of windowed-sinc impulse responses with improved performance from first to last (left to right, then down). The uniform window has ripple in the passband and tails off slowly to an attenuation of -46dB or so by the $f_s/2$ point. The Bartlett window is not much better, reaching around -55dB at $f_s/2$, but with a slightly better roll-off. The cosine window improves things again, but with the Hamming window we now have a very decent response, reaching -60dB (for the given Alpha value) with a superior roll-off rate and maintaining that attenuation in the stopband. The Kaiser window, like the Hamming window, has an Alpha (α) value that can be manipulated to trade off passband ripple, roll-off rate, and stopband attenuation. With the values I've configured for the example, you can see that superior stopband attenuation can be achieved, with good roll-off and minimal passband ripple.

Phase Characteristics

The diagram below shows the phase plot for the example Kaiser windowed-sinc impulse response over the passband (0 to 24KHz).



The phase axis is in degrees and runs from -180° to + 180°, where the plot then wraps. If you imagine, though, that the axis goes to infinity, then you would see a straight line extending in a linear fashion.

This confirms what was suggested in the first part of the document that the phase is linear. This remains true regardless of the number of taps, the window function used or the quantisation.

Window Mathematics

So far, I've taken the window functions as read without describing how they are calculated. The uniform window should be intuitive, as is, I hope, is the triangular window. For the rest, I want to look at the examples for those discussed above, but the comments above each window function in the `winfilter` source code details the mathematics for all the window functions not explored here. This section can be skipped for those not interested in the details, but just want the tap coefficient values, but I think it is interesting to have a look for informative purposes.

The cosine window isn't as straight forward as plotting for ±½π, but gets raised to a power, defined by the Alpha setting (1.0 in the case of the example above). I.e.

$$h[i] = \cos(\frac{i\pi}{\text{TAPS}})^{\alpha}$$

The Hamming window is a raised cosine where the Alpha (α) value defines how much above 0 the cosine is raised:

$$h[i] = 2\alpha\cos(\frac{2\pi i}{TAPS}) + 1 - 2\alpha$$

A Kaiser window is now going to pile on the mathematics to some degree, using a 'modified order 0 Bessel function'. The window is calculated as:

$$h[i] = \frac{I_0\left(\propto\left(1 - (\frac{i}{\mathrm{TAPS/2}})^2\right)^{½}\right)}{I_0(\propto)}$$

Where I_0 is a modified order 0 Bessel function, I_0, is calculated as:

$$I_0(x) = 1 + \sum_{k=1}^{\infty}(\frac{1}{k!}(\frac{x}{2})^k)^2$$

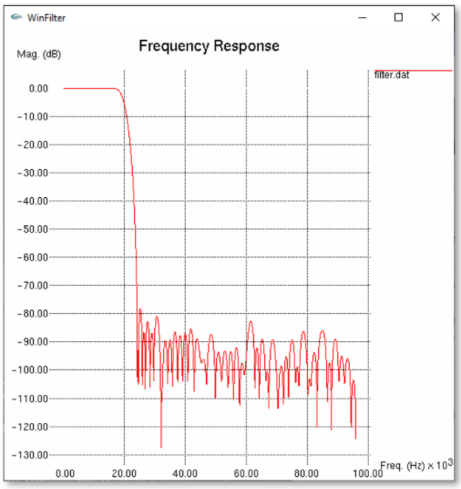
The Bessel function runs with k from 0 to infinity which, obviously, can't be computed, but the terms converging towards 0 as k increases, and a value of 69 terms was heuristically chosen to be used in `winfilter`.

So you can see, things can get quite elaborate. The Chebyshev window, for example, is actually calculated in the frequency domain, and then converted into the time domain with an inverse discrete fourier transform. I've included some examples in this section so that there is an idea of what's involved. The good thing, of course, is that these calculations do not have to be done as part of the implementation, but can be calculated offline, with the tap coefficients as the result to be configured into the FIR logic implementation. The source code documents all the windows, with C implementation for those that wish to study this more deeply.

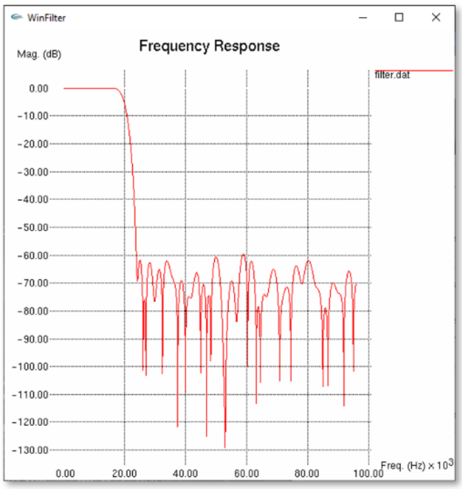
Quantisation

All the plots we've seen so far all use double precision numbers for maximum accuracy but quantising the impulse response taps to a fixed number of bits (either signed integers or signed fixed-point values—it makes no odds) has an effect on the resultant filter performance. In particular, the stopband attenuation performance suffers. Taking the Kaiser windowed-sinc filter example from before, which reached better than -80dB in the stopband, we can change the quantisation setting and replot the frequency response.

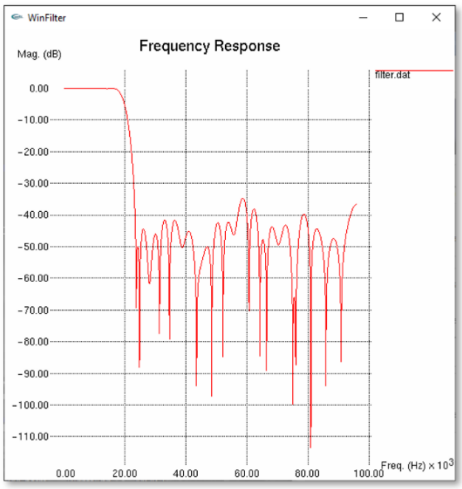
Q = 16



Q = 12



Q = 8



With 16-bit quantisation, we still reached around -80dB attenuation, if with a little messier stopband ‘lobes’, but with 12-bit precision, this becomes around -60dB and with 8 bits around -45dB but rises above this further up the stopband frequencies. Thus the choice of window function and an Alpha value is not enough to reach a particular target performance, and the number of quantisation bits has to be chosen to sufficiently meet the requirements as well.

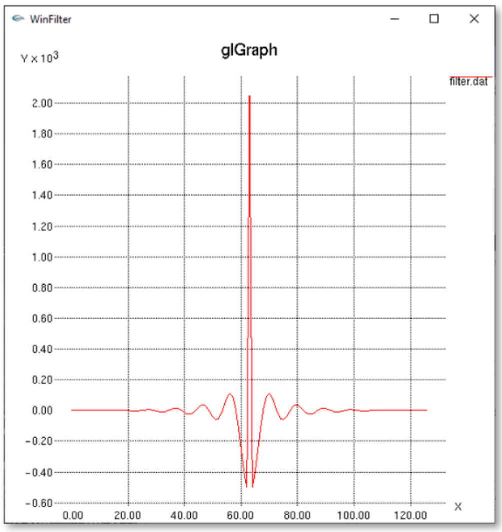
In theory, the achievable signal to noise ratio for a given number of bits (Q) is around $6Q + 1.76$ dB. So a Q of 12 should yield -74dB. In the $Q = 12$ plot above, it actually does no better than -60dB across the stopband. What I see is that the initial transition will reach around the theoretical limit, but the uneven lobes will often bring this back up above this, so plotting the actual response is important to verify performance target have been met.

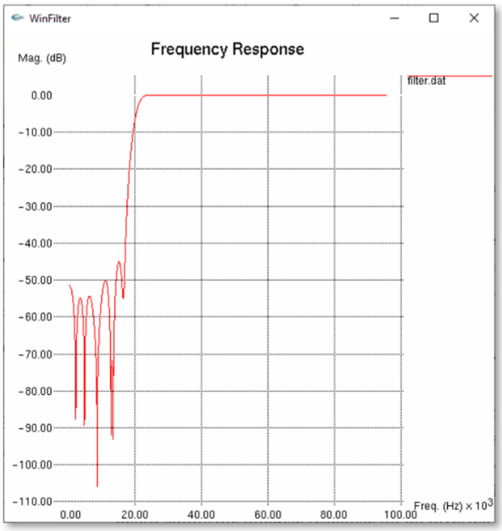
High Pass, Band Pass and Band Stop Filters

Until now only low pass filters have been discussed, with the *sinc* function being the fourier transform of the rectangular shape of the LPF, but what about highpass and bandpass filters?

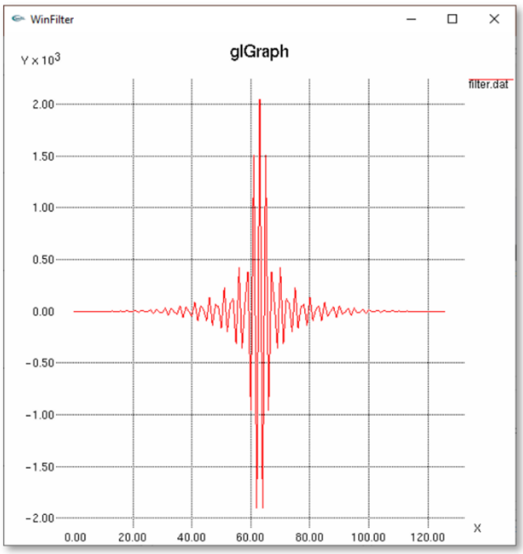
For highpass filters we could think of this in terms of lowpass filters in one of two ways. Firstly, if we invert an LPF in the frequency domain, i.e. flip it in the y axis, it will suddenly become a high pass filter. Also, if we reversed the frequency response about the $f_s/2$ midpoint, flipping it about the x axis, so that the stopband starts first ending in the pass band, that would also be a highpass filter. What does this mean for in the impulse responses?

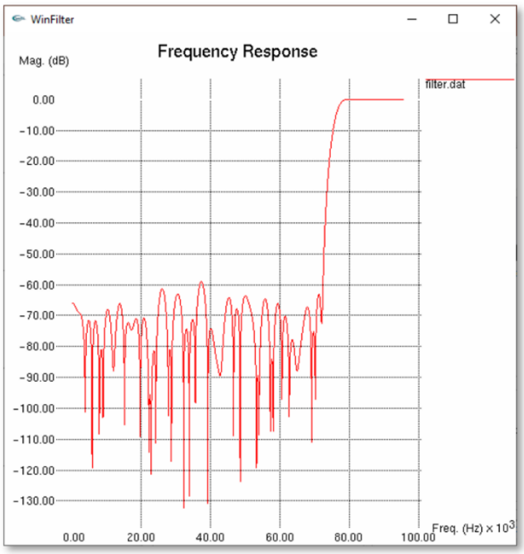
For the inversion, in the frequency domain, if we subtracted the low pass filter response from an all-pass filter response (i.e., 1) this would achieve an inversion. We saw before that an all-pass filter in the frequency domain is a delta function (an impulse of magnitude 1) in the time domain. So subtracting our low pass impulse response from the delta function will invert the response. This means multiplying all the values by -1, except the centre point, where the delta occurs which is subtracted from 1. The diagram below shows the impulse response for an inverted LPF and its frequency response based on the Kaiser windowed examples from above.





To reverse an LPF, recalling the earlier idealised response diagram and the rectangular shape centred on 0 frequency, if we could shift this so that it was centred on $f_s/2$ instead, that would reverse the response, with (from the earlier diagram) the response from -$f_s/2$ to 0—a reflection of that from 0 to $f_s/2$—now sitting from 0 to $f_s/2$. We can do this if we convolve the LPF frequency response with a delta function at $f_s/2$. A delta function in one domain is a 1 in the other *if* it is at point 0. If it is shifted, the signal in the other domain becomes rectangular between -1 and 1. A delta function at $f_s/2$ produces a signal in the time domain at a frequency of $f_s/2$, and we know that convolving in one domain means multiplying in the other domain. So if we multiply every other coefficient of an LPF impulse response by -1, we will reverse the frequency domain. This is shown below:



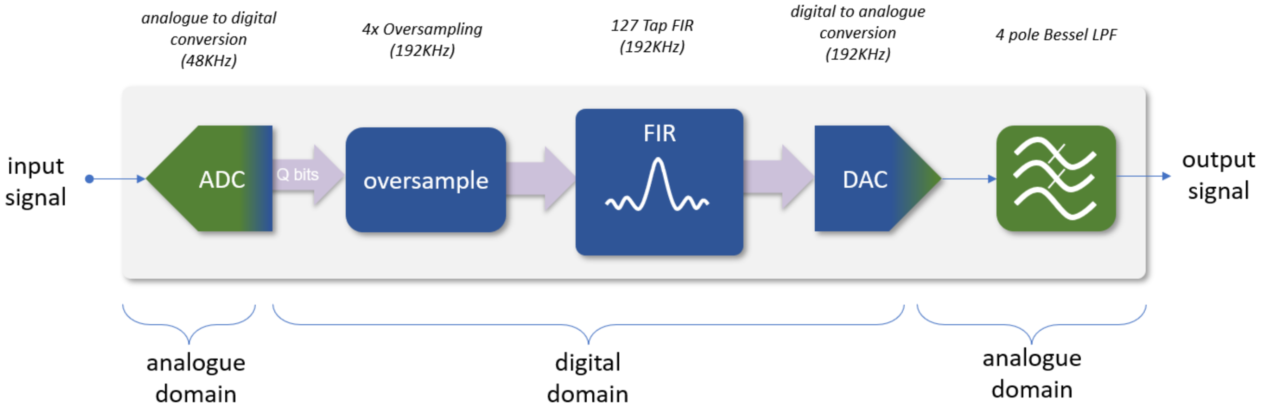


Now that we can make highpass filters, we can combine these to make bandpass and bandstop filters. For a highpass filter, imagine having an LPF and an HPF such that, in the frequency domain, they overlap. If you multiply these responses together then, where they overlap, there will be a pass band and elsewhere a stop band. Multiplying in the frequency domain is the same as convolving in the time domain, so we can take the impulses from the low- and highpass filters and convolve them together to generate the impulse response for the band pass filter. If we invert this band pass kernel, in the manner discussed above, this will become a band stop filter. So, by using convolution, multiplication, and lowpass filters, we can generate these other types of filters as well. This is why the bulk of this document has concentrated on low pass filters, as the others are just a function of these.

Designing the Filter

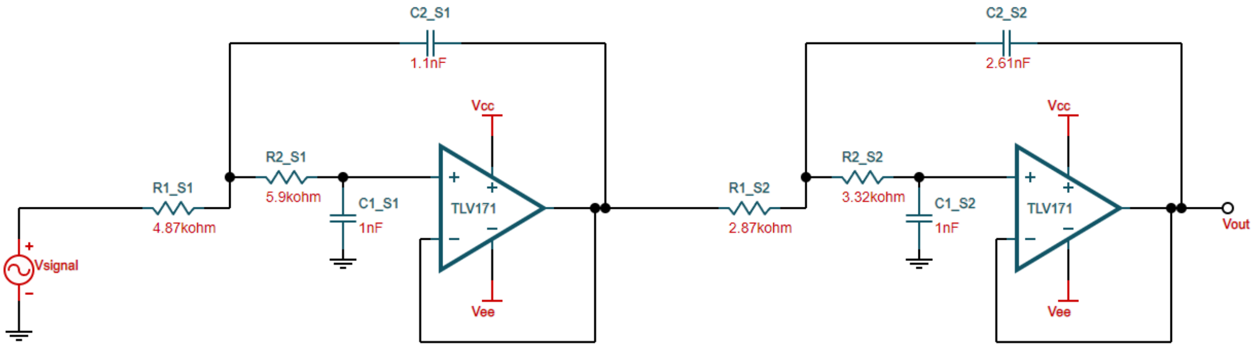
To put what we've looked at so far together, we will work though an example filter design, using `winfilter`, from the world of digital audio. Digital audio sample rates perhaps more familiar to most people from consumer electronics and streaming services might be 44.1KHz from Compact Discs (CDs), or 48KHz from DVDs and (in an obsolete reference to my own past) Digital Audio Tape (DAT). Blu-ray supports many audio formats, with varying number of channels at rates from 48KHz to 192KHz. These audio formats have sample widths ranging from 16- to 24-bits. The sample rates for audio will not tax the HDL implementation, which can run at over 750KHz with a 100MHz clock, and so we'll target an audio application.

As a side note, an FIR that runs at the higher rate of, say, 192KHz can still process signals at a lower (integer) rate such as 48KHz by 'oversampling' the signal. That is, inputting an individual sample multiple times (in this case four, for 4 times oversampling) and the filter will work just fine. The advantage of this is that the digital noise of the signal is spread over the larger frequency range. When the signal is converted back to the analogue domain and low pass filtered, the noise at the higher frequencies is attenuated, leaving less noise in the passband than would otherwise have been without oversampling. The analogue anti-aliasing filter specification can also be relaxed as it has a much longer attenuation region with which to reach its attenuation target to filter aliased frequencies. A simplified system is shown in the diagram below:



In the diagram I've ignored any clock synchronisation between the sampling rate and the internal clock, but this would need considering—either the system clock rate can be a synchronous integer multiple of the sample rate, or CDC logic added. CDC can add some time domain jitter to the samples which would need analysing to ensure acceptable (beyond the scope of this document).

With oversampling, the specification of the analogue filter can also be relaxed, as mentioned above, as it can roll off more slowly from the cut off frequency, as the space between this and its reflected frequency response between $f_s/2$ to f_s is wider than without oversampling. But the analogue LPF must be present to remove the reflected aliased signals. A typical analogue LPF used for this might be a Bessel filter which has a linear phase delay (aka maximally flat group delay) preserving the phase linearity of the FIR. An example 4-pole Bessel filter is shown below, courtesy of the Texas Instruments filter design tool.

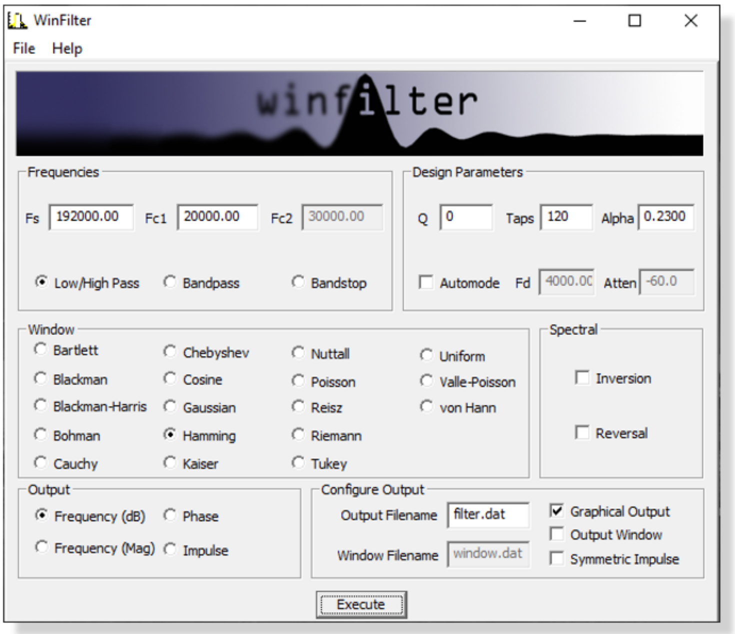


Before setting out on the specific audio example, a quick summary of the `winfilter` program usage is in order.

Note on Using winfilter

Firstly, a note on the [`winfilter`] source code. The `winfilter` source code (originally `xfilter` for Linux and X-windows) has its origins from more than twenty years ago. In fact, some parts of the code, particularly the DFT/FFT parts, originate from my undergraduate dissertation work done in 1988/'89. Therefore, it's written in C, rather than C++, and its style is not necessarily how I'd construct code nowadays. What I was gratified to see, when revisiting the code, was that there are plenty of comments, including the mathematics for things such as the window functions alongside the code that implements them. So, if a bit rough around the edges in terms of coding best practices, it is still a useful reference for all the details we can't cover in this document (we didn't explore every window function or the DFT and FFTs functionality—this latter we'll save for another article).

As the name implies, `winfilter` is a Windows based graphical program as shown in the diagram below with the default values:



Frequencies

Fs 192000.00 Fc1 20000.00 Fc2 30000.00

● Low/High Pass ○ Bandpass ○ Bandstop

Design Parameters

Q 0 Taps 120 Alpha 0.2300

□ Automode Fd 4000.00 Atten -60.0

Window

○ Bartlett ○ Chebyshev ○ Nuttall ○ Uniform
○ Blackman ○ Cosine ○ Poisson ○ Valle-Poisson
○ Blackman-Harris ○ Gaussian ○ Reisz ○ von Hann
○ Bohman ● Hamming ○ Riemann
○ Cauchy ○ Kaiser ○ Tukey

Spectral

□ Inversion
□ Reversal

Output

● Frequency (dB) ○ Phase
○ Frequency (Mag) ○ Impulse

Configure Output

Output Filename filter.dat
Window Filename window.dat
☑ Graphical Output
□ Output Window
□ Symmetric Impulse

The documentation that comes with `winfilter` explains in detail each of the functions on the GUI and what they are all for, but a brief summary here is worthwhile, I think.

My intention for the program was to be able to explore FIRs, all the different windows functions, and the effects of the different parameters on filter performance. In terms of actually generating some tap coefficients we can actually use, my intention is to do a two phase approach. The first is to select parameters and alter them until we meet the specification required, minimising parameters that cost in terms of logic, such as number of quantisation bits and the number of taps. For this we can plot frequency magnitudes to verify predicted performance. Once satisfied with these results, we can then plot the impulse response. The program will always output the plot values to the file specified in the 'Configure Output' box. This would normally be both X and Y values so the glGraph library I'm using can read these and display the graph. A new checkbox has been added, labelled 'Symmetric Impulse', which can be checked and, so long as the quantisation value, Q, is non-zero, the output will just be a list of signed hexadecimal numbers of the correct width for the Q value, suitable for configuring the memory in the HDL implementation.

The program does come with an automode design function. This is only available for the Kaiser window function and so this is automatically selected. In this mode, instead of setting the number of taps and an 'Alpha' value and seeing what performance is achieved, one can specify the pass- to stopband transition (the roll off) and a target attenuation in dBs. When executed it will update the Taps and Alpha boxes that achieve that specification, which then informs you the number of taps the HDL implementation needs to be configured for. Note that the Q value for quantisation will also have an effect.

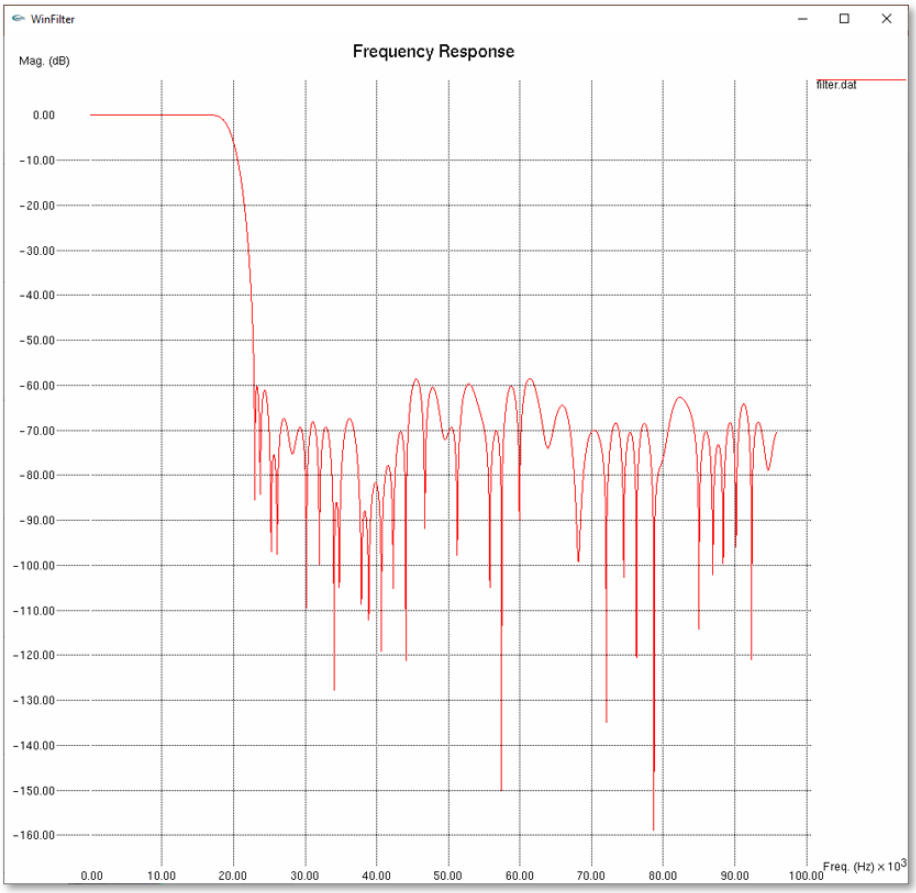
By default, the quantisation value Q is set to 0. This means no quantisation, but actually the program will use double precision floating point values, which is the best we can do. Using Automode when Q is 0 will, indeed achieve the specification configured. However, if you specify a target of -100dB attenuation with a 1KHz roll-off period (other parameters being default), and then set the quantisation to 8 bits, you have no chance of achieving that target, and this parameter must be experimented with as well.

Example Specification

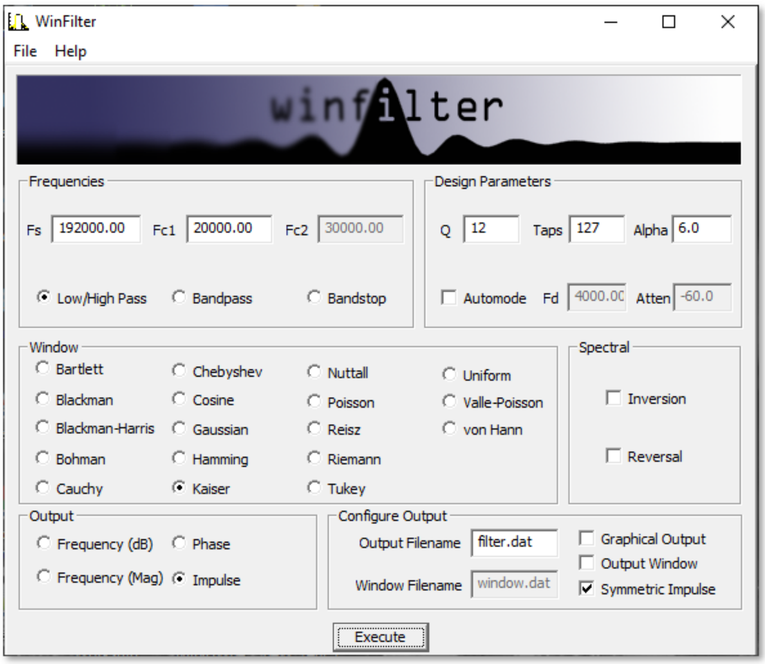
So, for our audio example let's set some parameters.

- 192KHz sampling rate
- Kaiser windowed-sinc filter
- Pass- to stopband transition < 3KHz
- -60dB attenuation in the stopband

The first parameter we can set from this specification in the quantisation Q. For an attenuation of -60dB we saw from our quantisation plots from before that 12 bits is the smallest we can practically get away with. The number of taps will determine the frequency transition width and we can use the Kaiser window automode to set the 3KHz transition and the target attenuation. This gives 116 taps and an Alpha of 5.6533. If we make the taps 127 (the implementation module's default parameter setting) and set an Alpha value of 6.0, along with a quantisation of 12 bits, we get a frequency response as shown below:



This appears to meet the brief well. At some of the higher frequencies the lobes do push slightly above -60dB, but the analogue anti-aliasing LPF will be attenuating by those frequencies, bringing them back into line. Now we have our settings we can generate the tap coefficients by selecting 'Symmetrical Impulse' in the Configure Output box. The `winfilter` GUI will look like the following:



Frequencies

Fs 192000.00 Fc1 20000.00 Fc2 30000.00

● Low/High Pass ○ Bandpass ○ Bandstop

Design Parameters

Q 12 Taps 127 Alpha 6.0

□ Automode Fd 4000.00 Atten -60.0

Window

○ Bartlett ○ Chebyshev ○ Nuttall ○ Uniform
○ Blackman ○ Cosine ○ Poisson ○ Valle-Poisson
○ Blackman-Harris ○ Gaussian ○ Reisz ○ von Hann
○ Bohman ○ Hamming ○ Riemann
○ Cauchy ● Kaiser ○ Tukey

Spectral

□ Inversion
□ Reversal

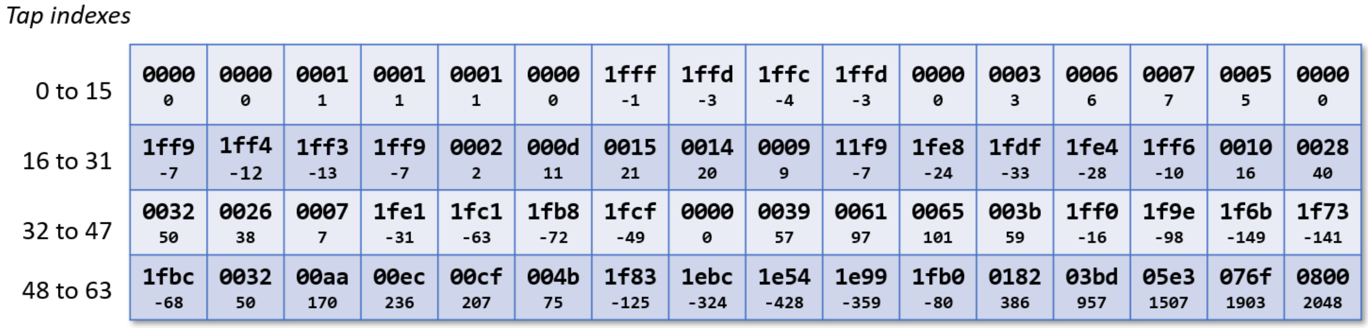
Output

○ Frequency (dB) ○ Phase
○ Frequency (Mag) ● Impulse

Configure Output

Output Filename filter.dat
Window Filename window.dat
□ Graphical Output
□ Output Window
☑ Symmetric Impulse

If the program is executed with these settings, the generated filter.dat file will now contain 127 signed hexadecimal 13 bits numbers—in the first part of the document we made the HDL LUT values one bit larger that Q so that an equivalent 1.0 can be stored (0x800). The table below shows the first 64 hexadecimal values, remembering we will only store half the table, with the impulse response reflected about the value at index 63:



Tap indexes

0 to 15	0000 0	0000 0	0001 1	0001 1	0001 1	0000 0	1fff -1	1ffd -3	1ffc -4	1ffd -3	0000 0	0003 3	0006 6	0007 7	0005 5	0000 0
16 to 31	1ff9 -7	1ff4 -12	1ff3 -13	1ff9 -7	0002 2	000d 11	0015 21	0014 20	0009 9	11f9 -7	1fe8 -24	1fdf -33	1fe4 -28	1ff6 -10	0010 16	0028 40
32 to 47	0032 50	0026 38	0007 7	1fe1 -31	1fc1 -63	1fb8 -72	1fcf -49	0000 0	0039 57	0061 97	0065 101	003b 59	1ff0 -16	1f9e -98	1f6b -149	1f73 -141
48 to 63	1fbc -68	0032 50	00aa 170	00ec 236	00cf 207	004b 75	1f83 -125	1ebc -324	1e54 -428	1e99 -359	1fb0 -80	0182 386	03bd 957	05e3 1507	076f 1903	0800 2048

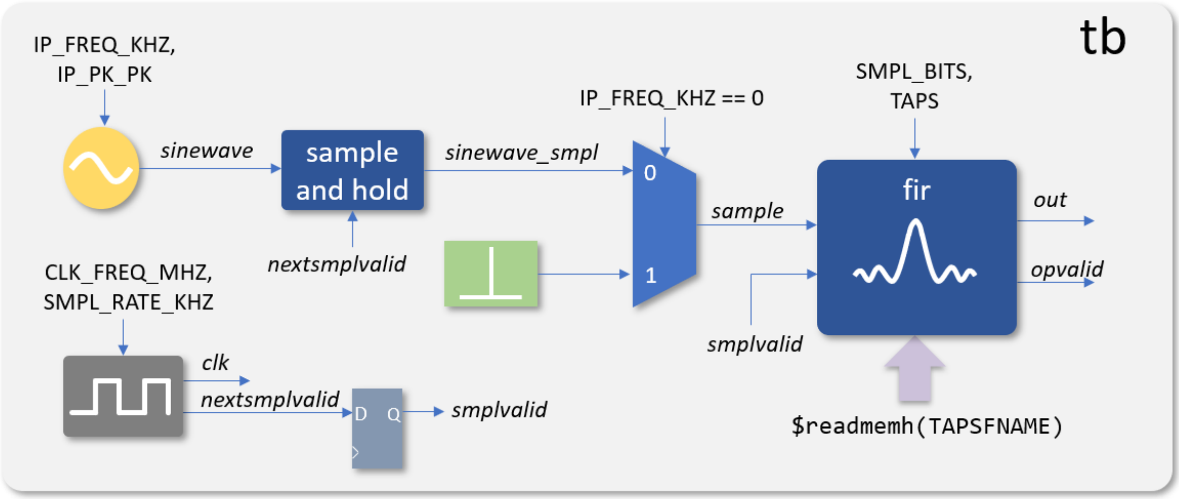
These numbers have been scaled by the winfilter program to give maximum range, and thus resolution, to the impulse response, with the peak of the impulse response at 2048 (at index 63). This will add a gain to the filter. When winfilter uses floating point values (Q set to 0) the impulse response generated gives unity gain at DC, but when quantised this would reduce the resolution, and so the numbers are rescaled to make the peak the maximum quantised value. This can be set back to unity, if needed, by external means, such as the gain of the anti-aliasing filter, or reduce the filter kernel values to normalise them. This latter, though, will reduce the filter's response performance.

So, after all this effort, we simply need put these numbers into the TAP lookup table of the logic implementation and we have a filter to our specification. Let's check if it works.

Performance

To test the FIR implementation, a simple test bench (tb) is provided in the github repository to simulate the module and check the results. This test bench has a set of user definable parameters to configure the test.

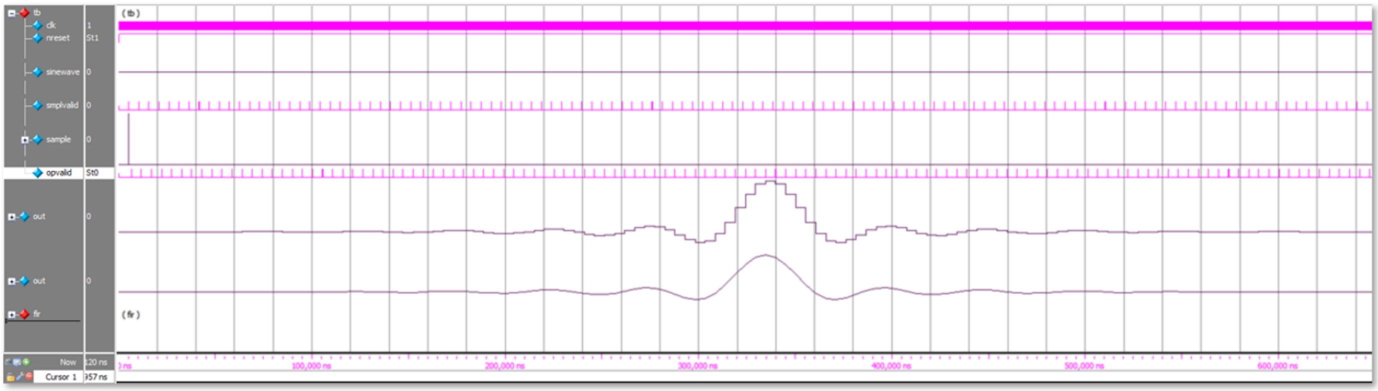
The input signal to the UUT can be selected to be either an impulse (IP_FREQ_KHZ is 0), or a sine wave with a frequency defined by IP_FREQ_KHZ, and a peak-to-peak value defined by IP_PK_PK. The system clock rate is defined with CLK_FREQ_HZ, and the input signal sample rate is defined with SMPL_RATE_KHZ. The default values are set up so that the clock rate is a multiple of the sample rate. I.e., a 96MHz clock and a 192KHz sample rate. The filter's parameters, `SMPL_BITS` and `TAPS` are also exposed for configuration with the test bench parameters. The illustration below shows a simplified block diagram of the test bench setup.



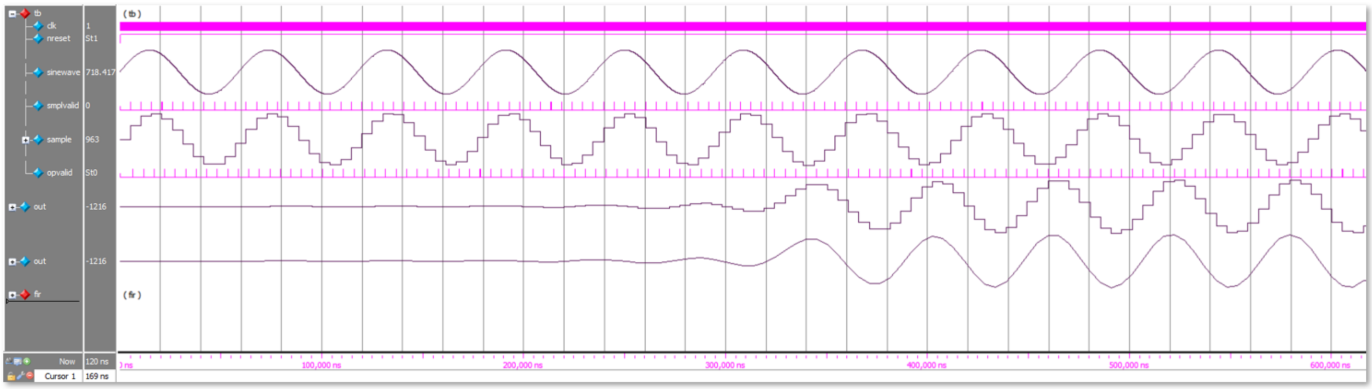
Simulation Results

For the purposes of this document, three tests were performed for the example configuration discussed in the last section: an impulse input, a sinewave at 15KHz in the passband, and a sine wave at 24KHz in the stopband.

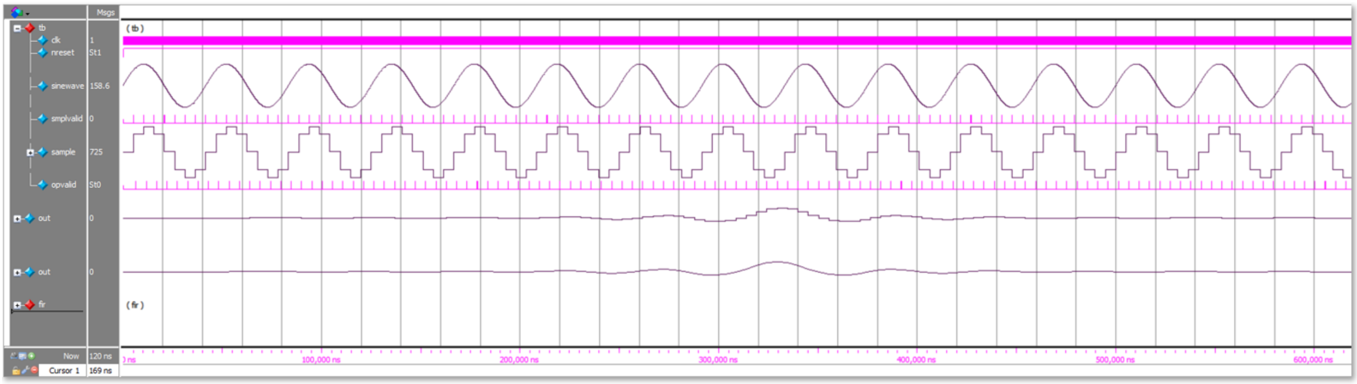
The diagram below shows the waveform trace for the impulse response. The signals displayed (from top to bottom) are a clock and reset followed by the `smplvalid` pulses at the sample rate. The impulse signal, on `sample`, is next (look to the left), followed by `opvalid` and then `out` from the FIR filter. The last signal uses the simulators 'interpolation' mode to emulate a DAC and analogue anit-aliasing filter to remove the steps. As can be seen, we do, indeed, get the impulse response at the FIR's output. This also illustrates the latency of the filter.



The next plot is for the 15KHz signal, which is at a frequency just before the predicted filter response starts to attenuate. Here, a sine wave is generated and then sampled to input to the filter. As expected, we do see this signal passing through the filter after the latency period.



The last plot does the same thing as the previous test, but the input signal frequency is now 24KHz, the lowest frequency in the predicted stopband. Here we see that the signal is filtered and does not pass through. There is some initial response after the filter latency, and this is due to the non-linearity of the transition from no input samples to a sine wave input temporarily generating frequencies in the passband, but it soon settles down to attenuating the input signal.



The test bench, bundeled with the FIR source code, is set up for ModelSim, but it should be simple enough to migrate to whichever simulator is convenient. Please have a go at trying different kernels and verifying the reponses you get for different frequencies, or add more complex signals to filter.

Conclusions

In this second part of the document, we built on the foundations of the first part, discovering convolution, and with an HDL implementation ready to program with some numbers. In this part was discussed how to generate those numbers for a lowpass filter, from the *sinc* function being the digital fourier transform of the rectangular frequency response to give an infinite impulse response. This was turned into a finite impulse response using ‘window’ functions of varying degrees of complexity and performance. To make practical for the Verilog implementation the finite impulse response was then quantised, and the limitations explored by this step.

Having generated a practical set of coefficients for a low pass filter, we then looked at how to manipulate and combine the tap coefficient to produce high pass and band pass filters.

With all this knowledge a practical example from the world of digital audio was specified and a simulation constructed to test the FIR implementation with some filter tap coefficients to match that specification. The results did indeed confirm the filter’s response to impulses, passband frequencies and stopband frequencies to be as predicted.

The convolution/FIR logic can be seen, I hope to be fairly simple, with the complexity in the calculation of the tap coefficients. Although more complex, the mathematics of this is not, I’d like to think, at the realm of professors of pure mathematicians, and at least means none of it ends up in the logic implementation and can be done offline. The resultant FIR, though, has impressive characteristics for such simple functionality, and is useful in a wide range of applications where stable and linear filtering is required.

More Information

For a very large section of my career, the foundation for the DSP knowledge I have has come from a book “*The Scientist and Engineer's Guide to Digital Signal Processing*”, By Steven W. Smith. I cannot recommend this book highly enough for anyone starting out in digital signal processing. It’s writing style and clarity is something that I have tried to emulate in my own documentation and these articles. Even, better, you can get hold it for free, on Steven Smith’s website at www.dspguide.com. This book includes way more information and detail that I can fit in a few articles but includes much more than FIR filters. I still refer to this text (I actually bought a hard copy, despite it being free). As far as implementation goes, it focuses more on digital signal processors, where I aim towards logic implementation, but is very much relevant for both disciplines. So, if you enjoyed these articles, and want to explore DSP more, check out this book.