Preliminary

The following sections in which I present a design for a ratio detector and ramp filter is firmly based in the design present by Adams and Kwan in their paper “A Stereo Asynchronous Digital Sample-Rate Converter for Digital Audio” which was published in 1994, and elements taken from their patent emanating from their work in this field. I don't claim any ownership of these ideas, but chose to use their design as the basis for a ratio detector and ramp filter since it is perhaps one of the most elegant solutions to the problem that I've seen. Whilst I did come up with my own scheme for achieving the same solution, which was equally effective, I chose to utilise the Adam's/Kwan design since it is much more space efficient with respect to an FPGA implementation.

The Sample Rate Conversion algorithm is my own, although many before me have utilised similar schemes. I must say that Adams et. al. do mention that higher order interpolation algorithms might be used than his presented (2nd order Lagrangian, or linear interpolation), but this would occur at the expense of increased computational overhead. And indeed this is so – in looking at his algorithm, I've seen that it can be implemented with a mere 6 DSP slices on a Xilinx Spartan 6. It's difficult to get his coefficient ROM just right so that the THD figures match up against his – and tweaking it would require more time than I have to dedicate to the problem, although my research in this area is still ongoing. So, my implementation uses a 3rd order Lagrange interpolator, which does utilise an extra 4 DSP slices (10 in total), but actually requires less RAM for more or less the same result as the AD1896 (the most recent iteration of the Adams design).

Ratio Detector

The ratio detector is a fairly simple circuit. It is displayed in the figure below, and is nothing new. It's simply an event counter – it counts how many time event A occurs over a period of N event Bs. When you think about it, it's exactly what we want. We have input frames (left/right samples) occurring at the sample rate, and 64 clocks per output sample frame, so we have complete information regarding the output to input sample rate ratio.

The lower register counts up every input frame sync pulse to a terminal value, which triggers another register to latch the value of a counter that increments every time an output clock event occurs. The following register is only enabled upon terminal count, and if its output is greater or less than the value of the Fso count register. In this way we implement what is known as hysteresis – which is a fancy term for the fact that there is some slack in the value, so that only significant changes in the input give rise to a change in the output. At the output of this register we have the input to output sample rate ratio with an integer range of 0 to 15.

Following on from this, we apply the ratio to a very simple low pass filter. Actually, this filter is what is known as a leaky integrator (I've never questioned why it is called leaky, Google probably knows why). The integrator output slowly changes with response to an input stimulus until the input and output are equal in value. How does it work I hear you ask? Conceptually it is very simple! Firstly, to understand what the circuit does, we need to understand what integration is. Integration is a mathematical concept where, given a continuous line with X and Y coordinates, we calculate the area under the line. In discrete time systems, this amounts to continually adding values of a signal to itself. This is achieved by adding a register output to its input – producing an error term. Sounds simple really, and it is.

Now to explain the circuit. The output of the integrator is subtracted from the input signal – producing an error term. We apply a gain to this error term, which then is added to the output of the integrator (this is how it performs integration). The error term is registered to be output on the next clock cycle. What does this achieve? Well, in terms of the ratio detector, we are producing a signal that is constant (i.e. a DC value), and so we expect the integrator to slowly rise or fall from its current value to the value presented to its input just like a low pass filter would. How fast it does this is dependent on its programmed gain, and the result is very much like a simple RC low pass filter. So, by utilising this integrator, we only allow the output to change slowly with any given input. The main purpose of the integrator is to reject spurious changes in either the input or output sample rates, which are expected to be rather high frequency when compare to the cut-off frequency of the filter.

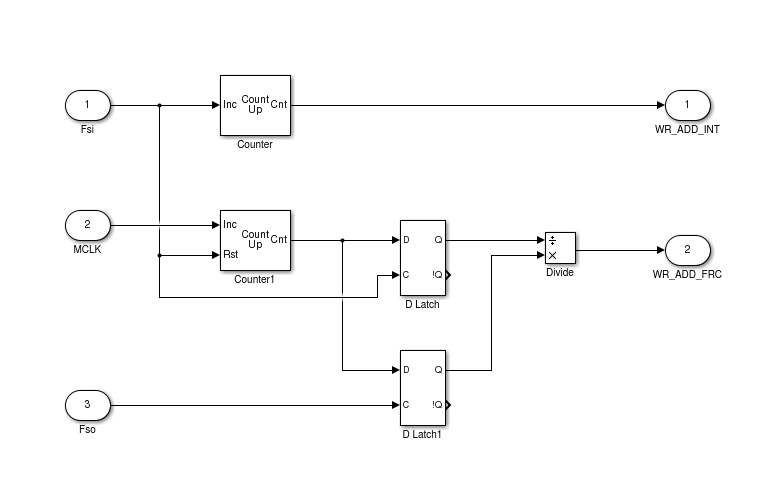
Finally, the output of the integrator is presented to a limiter. Earlier I mentioned that the calculated ratio could have an integer value in the range of 0 to 15, well the limiter restricts the total range of the ratio output to values between zero and one. The fractional portion is critically important, and I'll explain that later when I discuss the SRC algorithm, but form now, it's only important to know that the output of the ratio detector cannot be greater than one.

Now, I should mention that whilst none of these components are novel in and of themselves, they were first presented in this format in US patent 5471411 back in 1995. At least some of the inventors worked for Analog Devices at the time this patent was granted, and maybe still do. You may have heard of the AD1896? I believe that, at least from what I can tell from the datasheet, this circuit is still in use. Thanks go to Adams, Kwan, and Coln for this work.

Ramp Filter

The ramp filter is what is known as a first order integrator, whose input is an input sample request relative to an output sample request. It's important to remember that in an output sample request could arrive at any time between to successive input samples, there could be several output sample requests (in the case of up-sampling), or even none at all (in the case of down-sampling. Also, given the asynchronous relationship between input samples arriving and output sample requests, in order to execute decent sample rate conversion, we need very precise information regarding the time (or phase) relationship between an output sample request, and the arrival of a previous input sample.

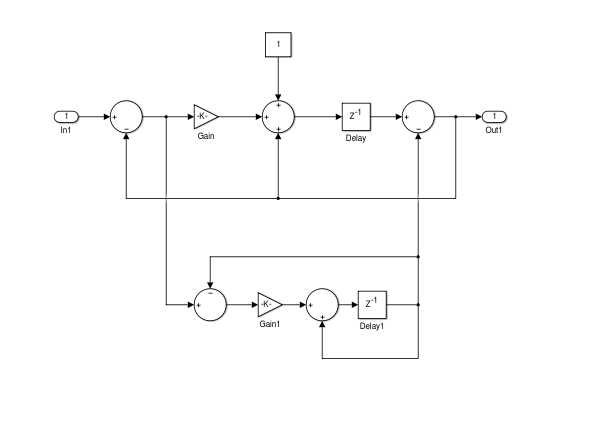
So, how do we calculate this? Below is a figure of a circuit that calculates just this – the usefulness of all the information generated will be explained just a little later.

FIGURE

Considering this picture, we can see two counters. The upper counter simply counts how many input samples frames have arrived. This is passed to a ring buffer as a write address, and also onto the filtering function which will be discussed soon.

The second counter counts master clocks, and is reset whenever an input sample frame arrives. This is followed by two latches. The upper latch simply takes the value of the counter every time an input sample arrives, thus giving us the number of master clocks occurring between successive input samples. The lower latch takes the value of the counter every time an output sample is requested, and remembering that the counter is reset at the time of input sample arrival, this latch gives us the number of master clocks occurring between an output sample request and the previous input sample arrival. If we divide these two latches (lower/upper), we get the a fractional input pointer value. If we append the result of the division to the upper counter, we have a complete input pointer, that increments by Fsi/Fso each time an output sample request is made.

As a note, the result of this division can never be equal to or greater than one. So as an optimisation, I have implemented a radix 2 divider that only produces the remainder of the division of its inputs. This results in a significant saving with respect to processing cycles.

Now let's look at the first order integrator, into which the input pointer is fed.

Firstly, let's start by considering the output delay element (Z^-1). The output of this latch is fed back is added to an input from the gain block. The latch takes this input as its value every time an output sample request is made. In this way we have constructed an integrator – it's an integrator because it is effectively accumulating the input (albeit scaled by the gain block).

Now let's try to understand the input to the integrator. As said above, an Fsi/Fso accumulator is fed to the input of this filter, which has the output of the integrator subtracted from it. In this way we have produced an error term which should be a constant when the filter has reached steady state. How do I know that? Well, the input is pretty much an accumulator of Fsi/Fso which implies that it is increasing at a constant rate of Fsi with Fso, and Fso which is the rate at which the latch is enabled. So, if it accumulates Fsi/Fso, and latched at rate Fso, then what we have as an output is actually an interpolated that provides read pointer to RAM, ROM, and an interpolation factor which is valid at each output sample request.

So, the difference of the integrator output, and the Fsi/Fso accumulator input should therefore be constant, and reflect the magnitude that the integrator should increment by at each Fso. We attenuate this error with the gain block of course, which simply affects the speed with which the the filter will settle – more feedback results in a faster settling time, and conversely, less feedback results in a slower settling time. It also affects how the filter reacts to sudden changes in its input – the slower the filter, the more it is able to reject spurious changes in the input. And this is a good thing, because the result is a filter that is able to reject jitter on its input, which in turn leads to averaged timing information with little variance.

So we have a filter, but what is its cut-off frequency? Well, as implied above, the cut-off frequency is dictated by the gain only. The equation that relates the cut-off frequency to gain is as follows:

G = T/RC

Where G is the gain, T is the inverse of Fso (1/Fso), and RC is the time constant of the filter. Plugging in the values for an Fso of 50 kHz and a gain of 1/2^11, we get:

(1/2048)=(1/44.1e3)/(RC)

Rearranging for RC:

RC = 2048/44.1e3 = 0.046

Which still isn't a cut-off frequency, but the rest is as simple as using the -3dB RC frequency formula:

fc = 1/(2\*pi\*RC) = 3.4 Hz

This is what the AD1896 advertises as the ramp filter's cut-off, and pretty much proves that AD are still using this filter in their implementation.

Now, how about the time resolution of the design. Pretty simple again. We are producing a fractional portion of the read address at a maximum of 192 kHz (highest Fso rate) that is 20 bits wide. Therefore the resolution of ramp filter is:

Res = Tso/Wlen=(1/192e3)/2^20 = 1/(192e3\*2^20) = 4.97 ps

If you read the datasheet for the AD1896, you'll see this figure pop up. Again proof that this ramp filter is in fact in use.

But we still haven't considered the bottom half of the ramp filter.

The bottom half of the ramp filter is a feed forward cancellation mechanism. Since the upper filter is only a first order filter, there will be some steady state error introduced – which is problematic because it may lead to read and write pointers overlapping during our SRC filtering process. At steady state, it subtracts a constant value from the output of the integrator, which is desirable since we never want the output of the ramp filter to cross over the actually ring buffer write address under dynamic conditions. It's a bit more complicated than this, because simply subtracting the feed forward value results in the distance between the integer portions of the read and write pointers separating by a degree that varies with the gain applied – but a more accurate fractional portion. The varying degree is problematic since if the distance between pointers grows significantly, we may write into a RAM location that is being used by in the convolution being performed by the SRC engine. This results, predictably, in severe distortion. Conversely, adding the feed forward value does the opposite – the integer portions of the pointers track very well (distance approaching zero), but there is a lot of noise in the fractional portion. Taking the best of both worlds, we provide an adder and subtracter and form our read pointer from each.

What we should note here is that there is one significant drawback with the circuit presented. That is, at relatively high gains, we can see some oscillation in the fractional portion of the read address pointer. This is mainly due to the architecture of the integrator involved. It's a first order low pass, in fact a leaky integrator just as before, and as such is an infinite impulse response filter. These filters are notorious for introducing what is known as limit cycling (the oscillating we observe), and is caused by quantisation within the filter. The error introduced due is to the gain, or shift, operation which effectively quantises the output of the input adder, which in turn is fed to the delay element and then fed back to the input adder. The integrator will produce an error term which includes the quantisation error, and it is this cumulative error that introduces a low grade oscillation. The solution to this is to simply reduce the amount of gain applied. There are other methods for tackling limit cycling, but they are pretty complex, and only serve to increase the complexity of the implementation.

The output of the ramp filter has two main parts to it:

1. An integer part that can be used as an initial address into out ring buffer RAM that we'll perform a convolution over; and
2. A fractional part which itself can be broken up into two parts:
   1. The seven most significant bits provide a reference to the first address within our polyphase filter that we'll convolve with the RAM elements mentioned above; and
   2. The 13 least significant bits which provide an interpolation factor that we'll use to interpolate our polyphase filter.

This technique of extracting addressing and interpolation data from a fractional input pointer was first presented, to my knowledge, by Smith and Gossett in their paper “A Flexible Sample-Rate Conversion Method,” 1984.

Note that I have omitted the mathematical analysis of the circuit here. The interested reader is encouraged to model it themselves. If suitable interest exists I can present present it, although it will take a while to get together.

Lock Detection

Ratio Detector Problem

So, the problem here is, how do we know when the SRC has obtained lock? What does it mean to have lock? What are we locking to?

Well the latter two parts of the problem are fairly easy to explain. Simply put, we have lock when the output sample rate to input sampling rate detector becomes stable. A fairly useless explanation in the sense that we haven't defined stability, so we'll describe what that means first. Here's out stability criteria:

1. Input and output clocks should be present. With respect to the input clock, we actually only need the left/right strobe signal – but the SRC is designed so that you can use either by synthesis configuration;
2. The output to input sample rate ratio should be greater than 0.125. Why? Because we simply don't have enough RAM available internally to accommodate down-sampling by a factor greater than 7.5:1 – and although 0.125 implies down-sampling by 8, it's close enough (and simple enough to implement);
3. The first derivative of the input to output of the integrator used for ratio tracking should be close to zero. This might require some explaining, and actually, the explanation is probably more difficult that the actual implementation.

To explain point 3 above, let's consider how the integrator behaves. Below is an image of the output of the integrator.

FIGURE

When unchanging input and output clocks are present the integrator is presented with a constant input. When the integrator has reached steady state, its output should reflect the input precisely. In other words, the output should eventually become constant. Now, the derivative of the output simply looks at its rate of change over time – and since it is constant, this should settle to zero (and will therefore be non-zero when I/o clocks change).

In mathematical terms, under steady state conditions, we can see that

f(x) = C

Where x is the ratio function. Therefore, taking the derivative of f(x) we can see that:

dx/dt = 0

Implementing such a thing is really quite simple. We simply subtract the input of the integrator from its output (i.e. the rate of change of its input and output) and look at the absolute value of the result. This gives us positive value to compare against a threshold for ratio lock.

Ramp Filter Problem

This is only part of the picture though. Remember, we have a ramp filter (or what Adams calls an auto-centring circuit) to consider too, and if this filter hasn't reached steady state then it's possible that we'll hear what is known as a doppler of the input signal.

So, we ideally want the ramp filter to have settled to some value too. And this makes sense when we consider that the output of the ramp filter provides critical timing information, including:

1. The initial address for RAM lookup (our ring buffer into which input samples are stored);
2. The initial address for our filter coefficient ROM; and
3. The interpolation factor that we use to interpolate the filter.

So having the ramp filter settle is really critical. It can be seen that the ramp filter has a fast response that we use to coarsely track to our intended steady state, and a slow response so that the filter doesn't respond overly to perturbations in its input (i.e. jitter). The selection of fast versus slow filter response is controlled by the ratio lock detector outlined above, but once in slow mode, how can we tell if the filter has reached steady state?

The answer is similar to the method used in the ratio lock detector – except this time we look at the second derivative of the ramp filter output. Why the second derivative? Well, remember from above that if a signal is unchanging at steady state, then the derivative of that signal will be zero. In this case we have a ramp to consider, and the first derivative of the this signal will be some constant greater than zero – so we take this and call it the first derivative of the ramp output. If we take the second derivative, baring in mind that the first derivative is some non-zero constant, then the result should be zero under steady state conditions.

The maths makes this pretty easy to understand. The formula for a straight line in:

f(x) = mt + C

Where f(x) is the output of the ramp filter, m is some slope of a straight line, and C is some constant. The first derivative, therefore, is:

dx/dt = m

And then, just as describe above, the derivative of this is:

d2x/dt2 = 0

However, we need to consider the fact that if the filter has settled in fast response mode, then the second derivative may have gone to zero at the time of mode switch to slow mode. Switching response modes will upset the ramp filter, so we wait for the second derivative to climb above some threshold, and then approach zero again before we declare that the SRC as a whole has achieved lock. Let's look at how the actual ramp filter behaves in practice.

FIGURE

Here, in zone 1 we can see the ramp filter rise from reset conditions and settle quite quickly. Zone 2 shows the reaction of the filter to the sudden gain change caused by the change from fast response, to slow response. The filter changes to some peak before settling around zero. So, when in slow mode, we detect that the second derivative of the ramp filter has risen above some value and settled again before SRC lock is declared.

When does the SRC leave lock mode?

Simple, if either of the ratio or ramp lock detector derivative values rise above some threshold then we say that we have left lock. Also, should either the input or output sample clocks halt, when we declare the ratio lock detector to have “unlocked” and therefore the SRC to leave locked state.

Polyphase Lock Loop Theory

Lagrange Interpolation

As some of you may know, the AD1896, or at least Adams' initial design, utilises a linear interpolator to calculate filter coefficients on the fly. A significant portion of his patent goes into how a filter is generated, and how linear interpolation can yield very good results. The payoff for such a thing is space – he utilises more space for filter storage than high order interpolators need.

I've chose a different design option. Some of you may not know this, so I'll mention it anyway, that linear interpolation is merely a second order Lagrange interpolator – so a special case wih respect to Lagrange interpolation where you only need two points in a series to calculate an intermediate value. I've chosen to go a little further, and use a third order Lagrange interpolator. It turns out that Lagrange interpolators are pretty good at estimating intermediate values of a filter – proof is in the pudding, AD1896.

I can hear you asking what exactly a third order Lagrange interpolator is. Well, unfortunately we can't really enter into a discussion without looking at some maths. So I'm going to hit you with the Lagrange Interpolation formula right of the bat. Don't worry though, I'll break it down into its constituent parts so you can understand exactly what's going on, derive the formula for a third order interpolator, and then show you a nifty optimisation to save a bit of logic when implemented in an FPGA.

Now Lagrange's interpolation formula …

What does this mean? Simple, that the Lagrange interpolation polynomial is the sum of n functions (Pj, which in our case is three functions). So, what is Pj? Another formula …

Now what does this mean? It's a bit tougher to explain, but basically it consists of a numerator (the top bit), and a denominator (bottom bit).

Firstly the top bit. Given x, which exists at some point between our three chosen filter coefficients, it provides the distance between x and each coefficient of interest. We exclude the case where the coefficient index is equal to the index of the polynomial we are calculating. So,for example, if we are looking at calculating the first polynomial then we don't consider the distance of x to the first coefficient from our filter – it would lead to a divide by zero in the denominator and the universe would probably come to an end.

Now for the denominator. This looks at the distance of each coefficient with each other coefficient – obviously the distance between one coefficient and itself is zero, so we can't do that. Now, it's worth noting here that in an FIR filter, each coefficient is uniformly spaced, so we can actually normalise these distances and to the address of the first coefficient. In other words, the distance between the first and second coefficients is one, the second and third is one, and the first and thrid is two. This makes the maths incredibly easy, because, as you'll see in a minute, we only consider 3 denominator values: 1, 2, and -1. The divide by two can be realised with a simple right shift by one since we're using base two maths.