XAPP936 (v1.1) March 5, 2007

Continuously Variable Fractional Rate Decimator

Author: Sean Caffee

Summary

This application note focuses on the baseband demodulation of Quadrature Amplitude Modulation (QAM) signals and, more specifically, on the use of a fractional rate decimator block. This application note also reviews polyphase decimating filter architectures and discusses the fractional rate decimator, its Xilinx System Generator 8.1i implementation, and its results.

Overview

The modern software-defined radio (SDR) system often must transmit and receive a variety of different digital waveforms. These waveforms might be different modulation formats (Quadrature Phase Shift Keying (QPSK), 16-QAM, etc.), different symbol rates (bandwidths), or even at different center frequencies (cellular band, Personal Communication Systems (PCS), band, unlicensed bands, etc.). Instead of building separate processing chains to deal with the many different waveforms that must be processed, a more hardware efficient method is to build a single processing chain that has enough programmable flexibility to process all required formats.

QAM Demodulator

A basic block diagram of a candidate demodulator is shown in Figure 1.

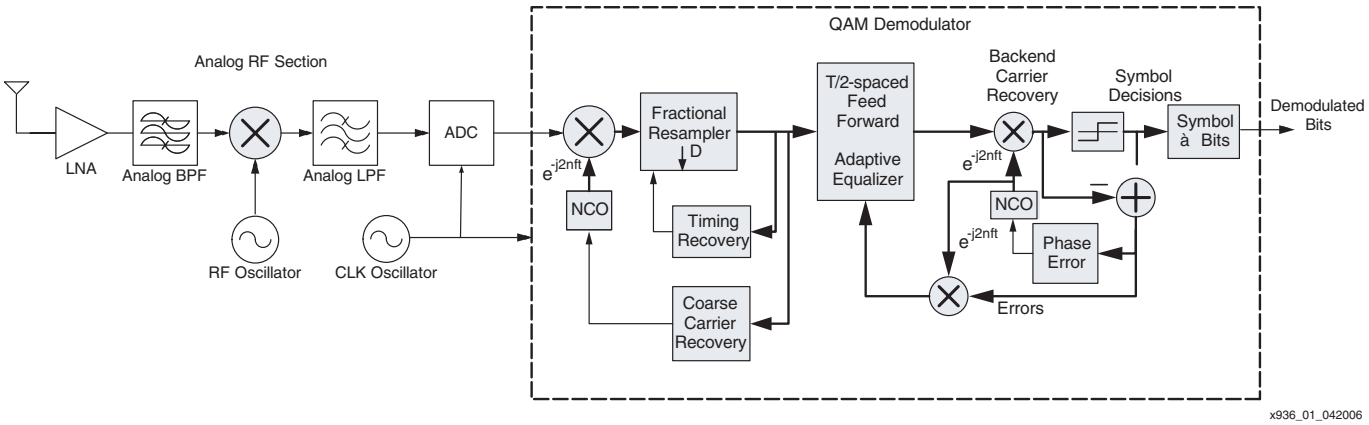

Figure 1: **Passband Sampling QAM Demodulator**

The QAM demodulator shown in Figure 1 consists of an analog Radio Frequency (RF) section and a digital section (within dashed line). The chosen architecture is a passband sampling arrangement where the signal is RF down converted to a particular Intermediate Frequency (IF) and then sampled at a particular (constant) frequency for all supported signals. The sample rate clock that drives the Analog-to-Digital Converter (ADC) is also used to clock the digital processing block. There are a few requirements on the appropriate sampling frequency, and

one of them is that it must satisfy the Nyquist requirement of being at least twice as high as the highest frequency in the (baseband) signal. Since this SDR demodulator is intended to receive signals of different bandwidths, the sampling frequency must be greater than or equal to twice the *widest* bandwidth signal of interest. Since the adaptive equalizer in Figure 1 normally operates on an input that is sampled at twice the symbol rate of the signal, there is a requirement to decimate the signal from the input sample rate to twice the symbol rate. A diagram zooming into this detail is shown in Figure 2.

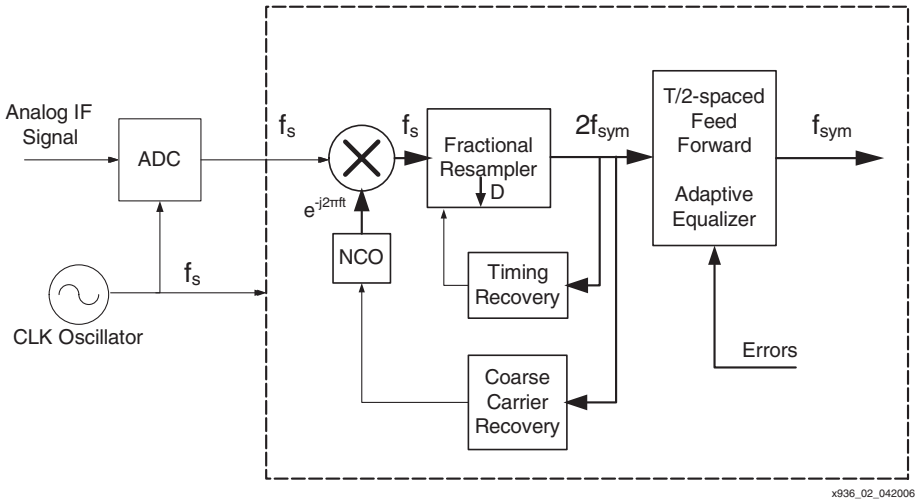

Figure 2: **Rate Change Detail**

If the ADC samples the signal at f_s and must enter the adaptive equalizer block at $2*f_{sym}$, then the resampler block must decimate (digitally lower the sample rate) the signal by a factor of $D = f_s / (2*f_{sym})$. When D is an integer, the problem is fairly straightforward. However, in general, D will not be an integer. For a demodulator designed for a *particular* signal bandwidth, the sample rate, f_s, can be chosen such that D is an integer. However, for a demodulator designed to support a variety (or a continuum) of signal bandwidths, D will be some fractional number greater than or equal to 1 (up to the designed limit). The maximum value that D is designed to take on dictates the bandwidth of the narrowest signal that can be demodulated. Whereas, when D is equal to 1, no decimation occurs and the widest bandwidth signal is allowed to pass through and be processed (in this case, the sample rate, f_s, is equal to twice the symbol rate).

Not only will the decimation value (D), in general, be a fractional number, but it also needs to be dynamically changed by the timing recovery block to resample the symbols at the proper time instant (or phase), not just frequency. For example, when outputting two samples per symbol from the resampler, one will ideally be at *top dead center* of the symbol, and one will be at the transition sample between symbols.

The remainder of this application note reviews polyphase decimating filter architectures and discusses the fractional rate decimator and its Xilinx System Generator 8.1i implementation and results.

Polyphase Decimating Filter Architectures

Before describing the fractional rate decimating filter, it is helpful to review decimating filters in general and polyphase decimating filters in particular.(1) The two basic operations involved in decimation are filtering and downsampling as shown in Figure 3.

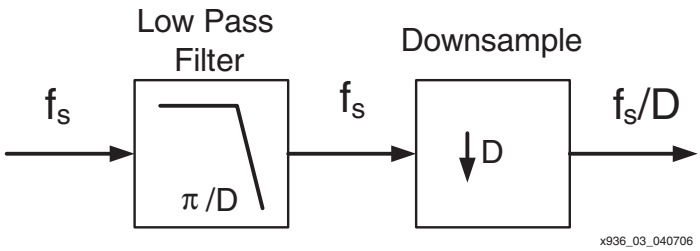

Figure 3: **Basic Decimating Filter Operation**

The signal's bandwidth must first be limited to half the sampling rate (π) divided by the decimation rate, D, otherwise aliasing occurs after downsampling. The downsampling step discards every D-1 samples (keeps every D^{th} sample), which reduces the output rate by a factor of D.

Note that in Figure 3, the low pass filter that limits the bandwidth, which can sometimes be quite long due to stringent image rejection requirements, operates at the input sample rate (the high rate), but only every D^{th} output of the downsampler is kept for follow on processing. So, the filter is performing calculations that are simply discarded. It is due to this inefficiency that polyphase filters are generally used to implement rate changes.

The same decimation by a factor of D (D is an integer for this example) operation can be implemented in a polyphase architecture by breaking the N-tap low pass filter in Figure 3 into D parts (or phases), each with N/D taps (if N is not evenly divisible by D, then pad with zero taps). This architecture is shown in Figure 4.

Each of the subfilters in Figure 4 is created by taking every D^{th} tap of the original filter and is N/D taps long. For example, if decimating by 4 and the decimation filter is 16 taps long, h_0 is made up of taps 0, 4, 8, and 12. Likewise h_1 would be taps 1, 5, 9, and 13 of the original 16 tap filter, and so on.

Note that each of the subfilters in Figure 4 now only operates at $1/D^{th}$ the input sample rate, but there are D of them. Since each of the subfilters only operates at $1/D^{th}$ the input sample rate, a single FIR filter structure (that is N/D taps long) can be operated at the full input sample rate, but with the coefficients changing every cycle and accumulating the outputs for D cycles, this is shown in Figure 5. As compared with the inefficient decimating filter of Figure 3, the polyphase implementation cuts the computational workload (or hardware resources) down by a factor of D (with the addition of a small amount of control logic required to properly sequence the stored coefficients to the tap delay line).

This provides an efficient way of decimating by an integer factor, but how is decimating done by a non-integer factor?

The classical way of performing rational ratio resampling is to upsample by an integer factor, low pass filter the result to remove images, and then downsample by another integer factor. This is shown in Figure 6. The polyphase partition of the interpolator has a dual form (directional arrows reversed, input and output signals reversed, and summing junctions replaced by nodes) to that of Figure 4. The resampling method in Figure 6 works fine for fixed ratios, but is still inefficient, because interpolated samples are computed and then some of them are discarded at the downsample block. The next section discusses a way to efficiently

1. *Multirate Signal Processing for Communications Systems* by fredric j. harris, published by Prentice Hall PTR, 2004, is an excellent reference for all of the multi-rate signal processing details that are not covered here.

resample an input waveform for an arbitrary ratio (or fractional number) and be able to continuously vary that ratio.

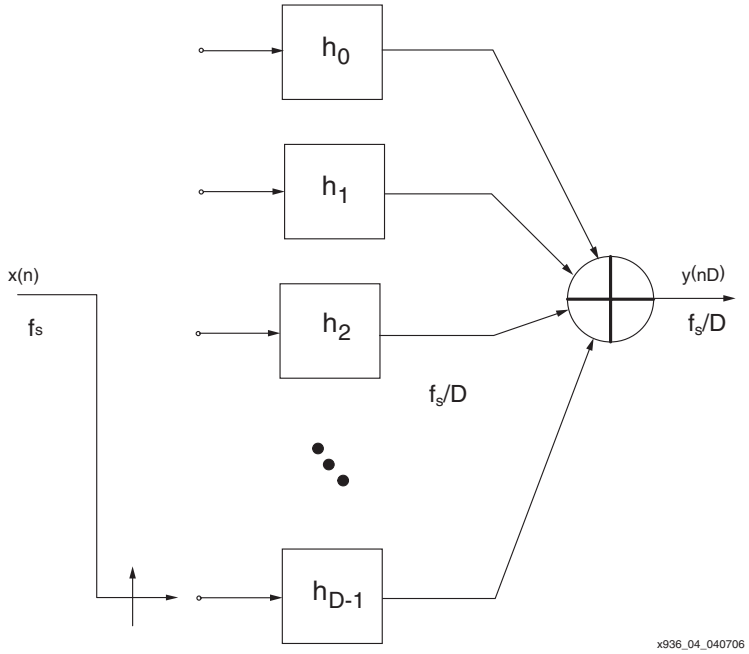

Figure 4: **Polyphase Partition for Decimation Filter**

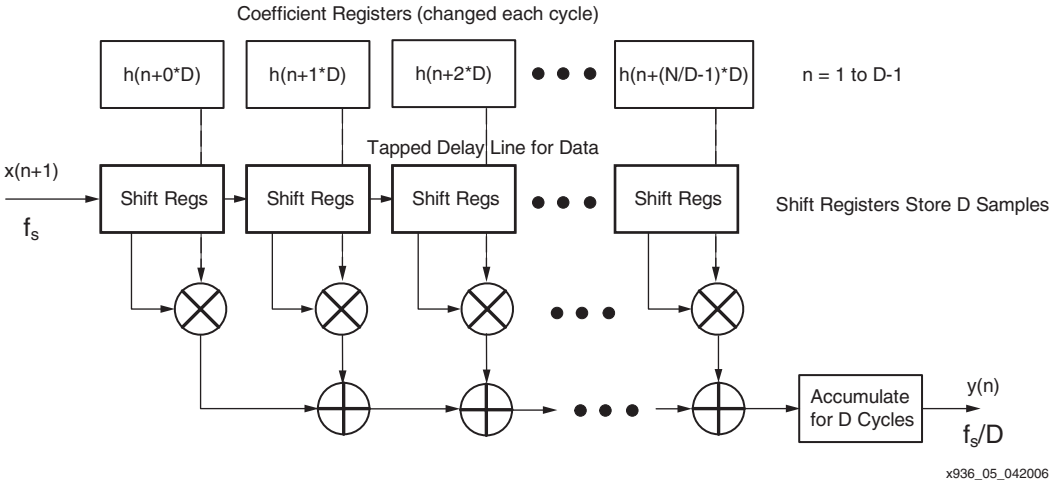

Figure 5: **Efficient Polyphase Filter Implementation**

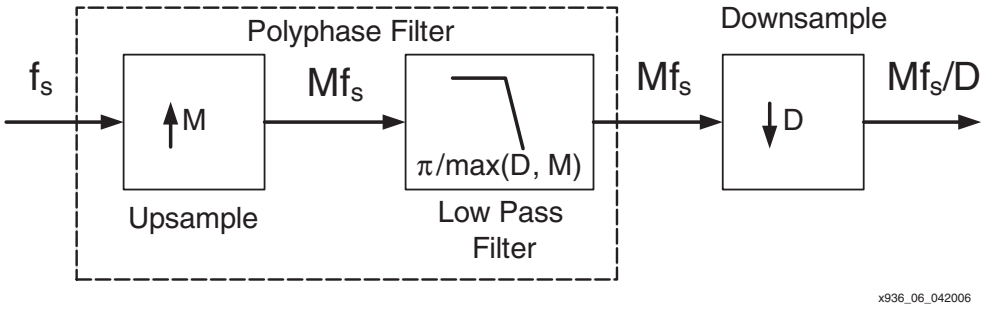

Figure 6: **Rational Ratio Resampling Operation**

Polyphase Fractional Decimating Filters

To develop the fractional decimating filter architecture, first redraw Figure 5 slightly by adding some detail and using the transposed form of the filter (instead of the transversal or tapped delay line version). This is shown in Figure 7 for a four phase, four taps/phase implementation. The transposed form of the filter is used to avoid the D length shift registers in Figure 5. If one ultimately wants a variable and fractional decimation rate, the fixed, integral length shift register do not work.

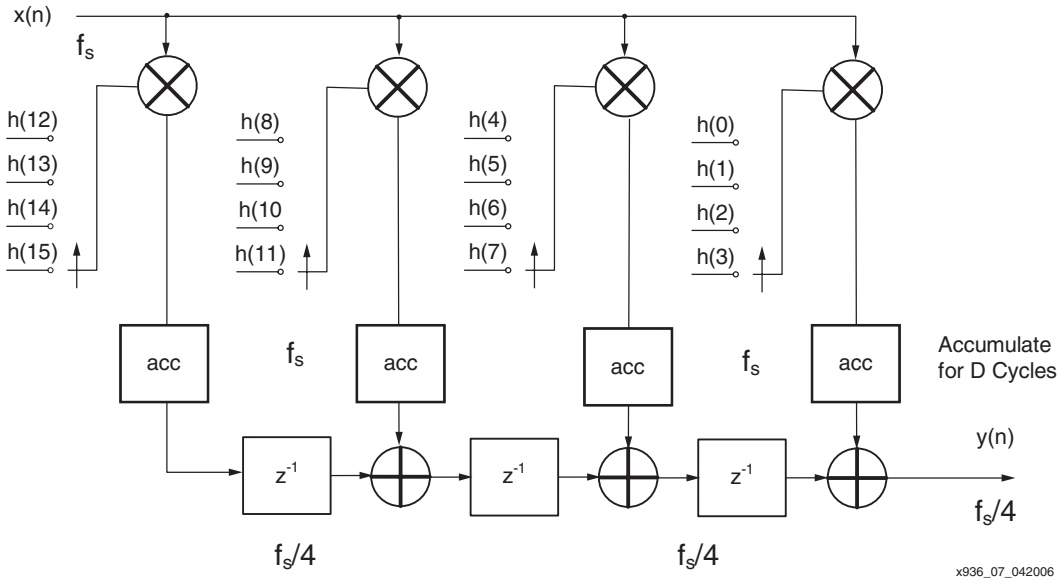

Figure 7: **Transposed 4 Phase Decimating Filter**

Some things to note about the filter structure in Figure 7 are that the data, x(n), comes into the filter at the sample rate, f_s, and is applied to all of the tap multipliers at the same time (transposed form of FIR filter). The coefficients are cycled through their values at the sample rate, but the indexes of the coefficients at any given time are separated by D, the decimation rate, or 4 in the example. As an example, take the line that begins with h(12) in Figure 7 on the left; reading across to the right are h(8), h(4), and h(0)—each coefficient index is separated by 4. The accumulators function at the input sample rate, but their output and that of the adder chain is clocked (or clock enabled to be more precise) at the decimated, or output, rate.

What is not shown explicitly in Figure 7 is the address generation circuitry that selects the proper coefficient from storage at the proper time. For this example, it is simply a 2-bit down counter (to address h(0) – h(3)) for each multiplier's local coefficient storage.

The architecture in Figure 7 can be used exactly as shown to decimate by a variable factor by storing a larger number of filter taps and having the address generator down count by a factor related to the decimation rate that is desired (the actual value is (N/tpp)/D, where N is the total length of the filter, tpp is taps per phase, and D is the decimation rate). Fractional rates can be accommodated by allowing the address generator (accumulator) to subtract fractional values and to accumulate the remainder as the accumulator wraps around its max/min values.

Xilinx System Generator Fractional Decimator Design

Figure 8 shows a Xilinx System Generator™ model of a fractional decimator design that closely matches the structure in Figure 7. This design allows continuous fractional decimation rate changes while operating and is highly parameterized, including input bit width, coefficient bit width, intermediate bit widths, and maximum decimation rate (the maximum decimation rate discussed below is 128. However, the maximum decimation rate parameter can be changed to 1024 without significantly increasing the FPGA resources needed.).

System Generator is a DSP design entry tool that works in conjunction with The Mathworks MATLAB® and Simulink® tools. System Generator accelerates the entry, simulation, and implementation of DSP designs by abstracting away many of the details of the low-level FPGA design, while still giving the designer control over the implementation to ensure high performance and low resource usage. After a design is entered and simulated using System Generator, a bit and cycle accurate implementation (VHDL or Verilog) can be generated automatically by clicking a button on the GUI. Complex DSP designs can be designed and implemented in hours or days instead of weeks or months using alternative design methodologies.

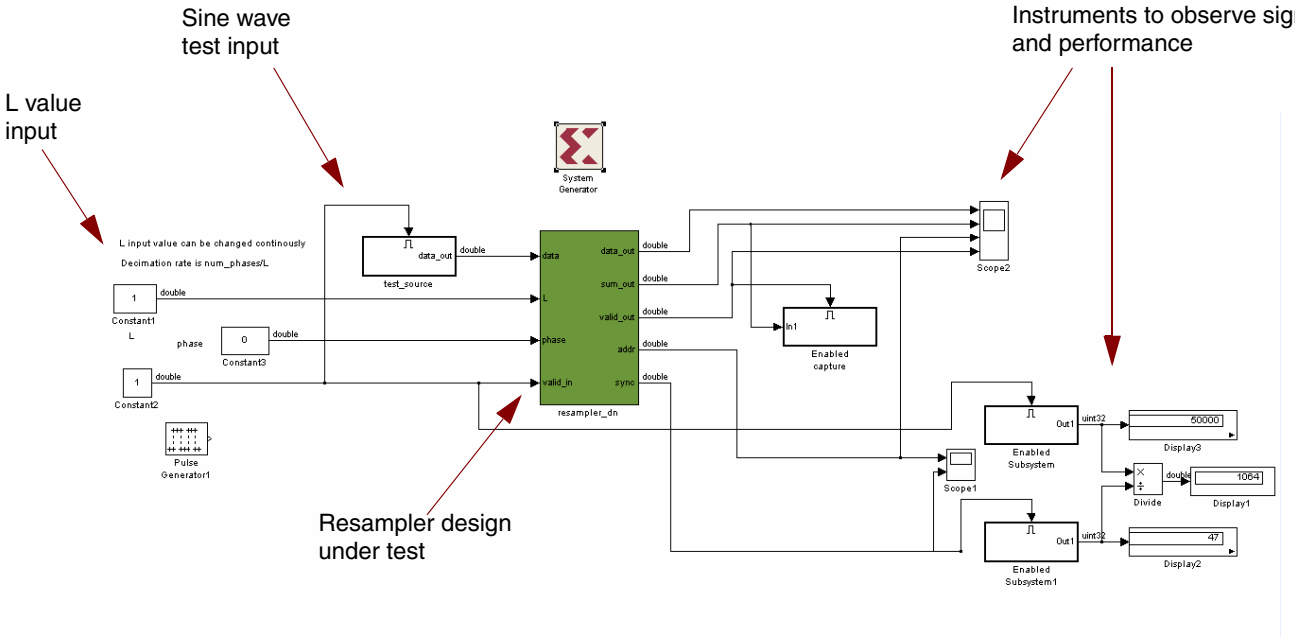

Figure 8: **Top-Level System Generator Testbench**

Figure 8 is the top-level design that contains the design under test (the decimator) and the input sources and output instrumentation of the testbench. The L value input is analogous to the desired decimation rate and is described in more detail in a few paragraphs below.

Figure 9 shows the top-level of the fractional decimator design, which consists of an address generator and a filter block. The small amount of logic (blue blocks) that are not contained in these two blocks simply changes the address generator output from an up counter to a down counter (it was more efficient to turn an up counter into a down counter than to start with a down counter).

The input and output signals that have Gateway blocks on them (yellow rectangles) translate into ports (or pins if the design is the top-level FPGA design) on the generated output design. The Gateway blocks translate between the floating point simulation environment of Simulink (testbench) and the fixed-point implementation environment of System Generator and delineate the portion of the design that will be implemented in an FPGA.

The output signals in Figure 9 that do not have Gateway blocks on them are simply internal signals provided for debug and observation and do not turn into ports or pins on the generated output design.

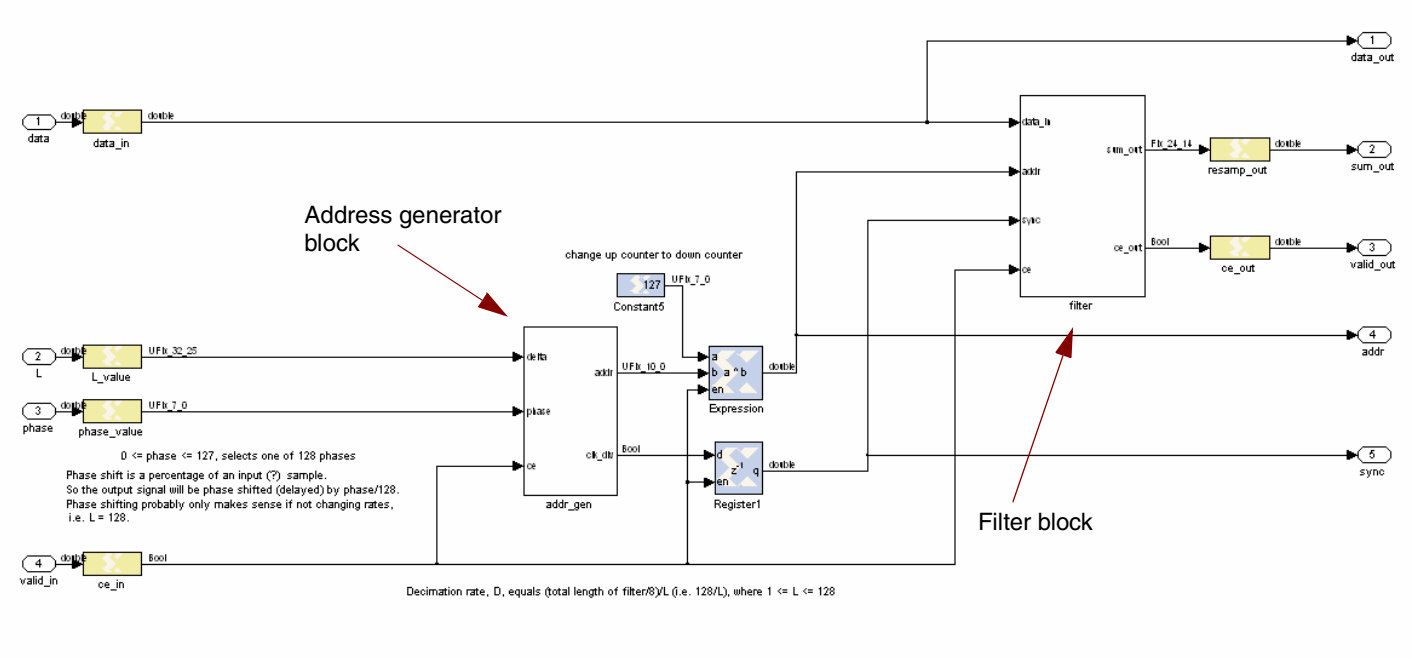

Figure 9: **Top-level of Fractional Decimator Design**

Figure 10 illustrates the address generator circuit, which is simply a phase accumulator that is set up to have a fractional as well as an integer portion. The accumulator continues to accumulate the *L value* input (which is equal to 128 divided by the desired decimation rate for this particular design) on each cycle. However, when it overflows or wraps, it generates a sync signal that is used in the filter block and becomes the clock enable output (valid output indicator) for the decimator. The integer portion of the accumulator is used in the filter block to address the proper coefficients.

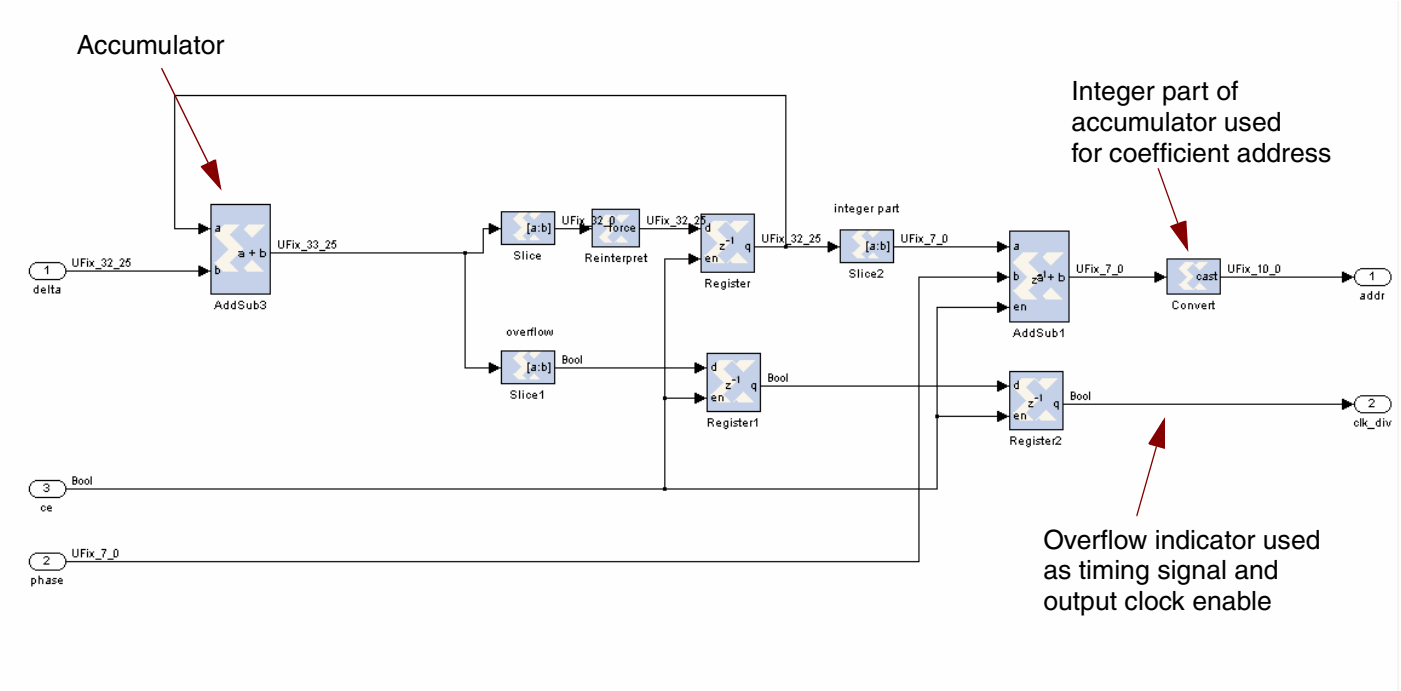

Figure 10: **Coefficient Address Generation Block**

Figure 11 shows the overall filter block, which is made up of four two-tap blocks and some individual registers and gates (blue blocks).

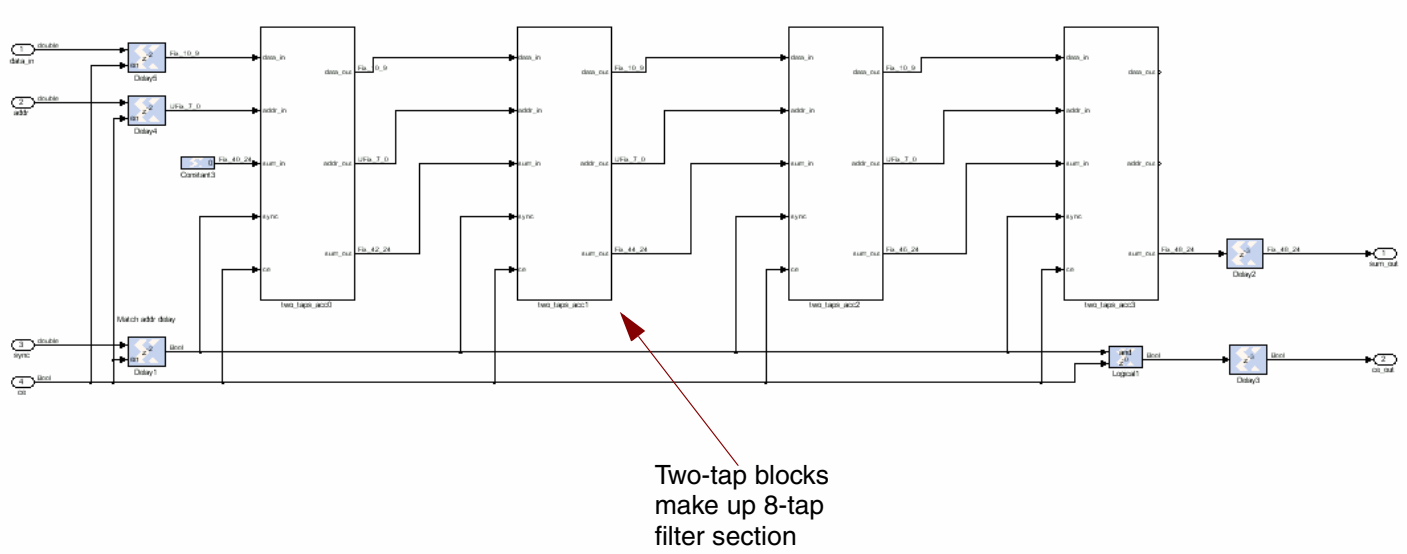

Figure 11: **Filter Block**

Figure 12 shows the detail of the two-tap block. This block consists of two single-port RAMs that store the coefficients, a multiplier and accumulator for each tap, and an adder chain along the bottom of the diagram. A single-port RAM block for each tap initialized with the filter coefficients(1) can supply each multiplier with the proper coefficient. The accumulator accumulates multiplier outputs for D (the decimation rate) cycles. Because D can be a fractional number, the actual number of cycles can be higher or lower than D, but will average out to D. Every component is clocked at the input rate (high rate) except for the capture register in the accumulator and the adder chain, which are clock enabled at the decimated rate.

The entire design utilizes a single fast, input sample rate clock, and outputs decimated samples along with a clock enable (valid output) to indicate which samples should be used for follow on processing.

1. Coefficient values and other parameters are initialized in the InitFcn callback under Model Properties. The filter used can be any 1024 tap low pass filter that has a passband out to 2/128 of the input sampling rate. Using normal MATLAB normalized cut-off frequencies, the filter cutoff would be 1/128.

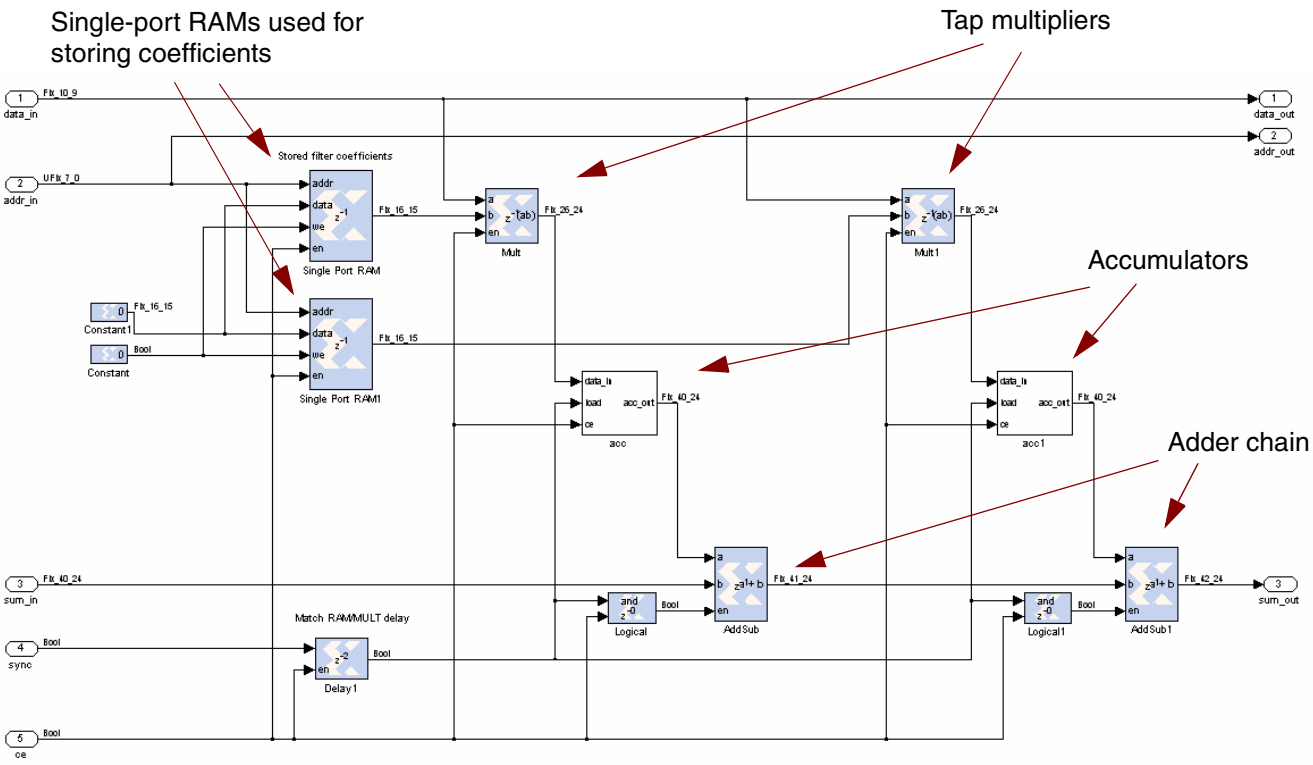

Figure 12: **Two-Tap Block**

When simulating the design, an oscilloscope in the testbench captures various test signals. For this simulation run, an L value of 2.63 was used, which equates to a decimation factor of 128/2.63, or 48.6692. A screen shot of the signals on the oscilloscope is shown in Figure 13. The top signal is a zoomed in portion of the input sine wave, the next lower signal is the decimated output signal. Note that the decimated output signal is changing at a slower rate and thus holds its value steady for many clock cycles.

The signal under the decimated output signal in Figure 13 is a zoomed in look at the top integer values of the address generator output. Since 2.63 does not divide evenly into 128, the accumulator overflows at slightly different values each period. The cycle that the accumulator overflows is captured as a sync pulse or valid out in the lowest signal on the oscilloscope screen shot. Because the accumulator does not overflow at the same point each period, the valid out signal's period dithers between the two nearest integers of the actual decimation rate, 48 and 49 in this example. The average period of the valid out signal is the fractional decimation rate (i.e., it spends more time at a period of 49 than 48, in this example, to average out to 48.6692).

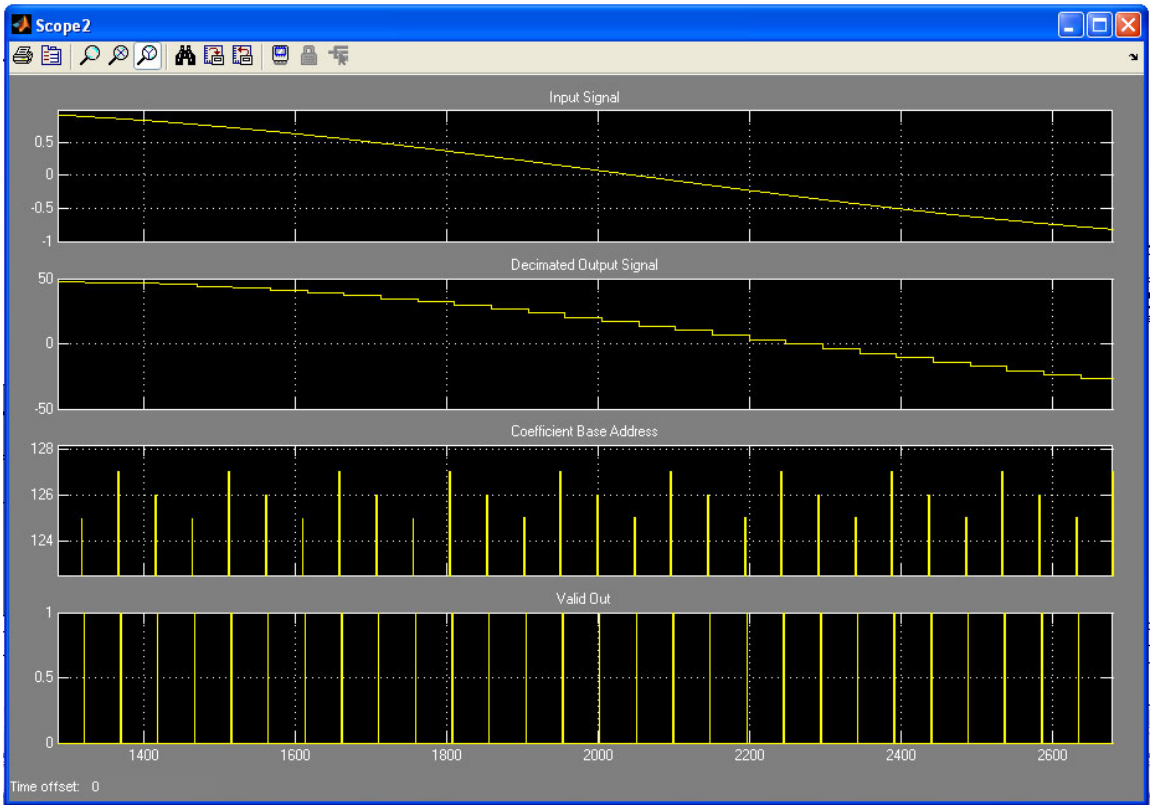

Figure 13: **Decimator Test Signals Screenshot**

Performance Results

The resampled output waveform fidelity is dependent upon a variety of factors including:

- the parameterized bit widths of the input data, the filter coefficients, and the partial sums in the filter;
- the number of taps in the resampling filter (the more taps, the closer the final output value is to the ideal sample point); and
- the desired decimation rate (or L value in the design example).

Table 1 was generated for a few decimation rates to give the reader a feel for the signal-to-noise ratio (SNR) that can be expected. The table was generated by injecting a sine wave into the fractional rate decimator for the different L values and observing the FFT of the output sine wave. Data input was 10-bits, coefficients used were 16-bits, and partial sums in the filter were quantized to 24-bits. The SNR was approximated by taking the difference in dB between the peak of the tonal impulse and the highest spur or noise floor. Using this criterion, the input sine wave itself was measured at about 82 dB SNR (4096 points quantized to 10-bits). So the degradation due to the resampler implementation (implementation noise) is the difference between 82 dB and the value shown in Table 1.

Table 1: **Output Signal Fidelity**

L Value	Equivalent Decimation Rate (128/L)	SNR (dB)
1.0	128.0	46
1.1	116.36	48
64.0	2.0	76
127.9	1.00078	62
128.0	1.0	82

The lower SNRs shown for the higher decimation rates (128 and 116.36) are attributable to round-off noise in the numerous accumulation cycles at these rates. Widening the allowed output bit width would improve these values. The lower decimation rates result in commensurately higher SNRs due to fewer computations being performed. The SNR for decimating at a rate of 1.00078 is lower than might be expected. This is because the ideal coefficients to use are not present in memory and the closest ones are used (the address generator truncates the ideal address to the nearest integer) instead. This causes a certain amount of jitter, which shows up as a lower SNR (noisier output sine wave). Increasing the number of taps in the filter would improve results for these situations.

Implementation Results

Using a Spartan™-3 XC3S200-5 device, 100 MHz operation was achieved with no special constraints while using only 8 embedded MULTs, 8 18 Kb block RAM blocks, and 378 slices. When targeting the Virtex™-4 architecture, a Virtex-4 XC4VSX25 -10 device achieved 200 MHz operation while only using 17 DSP48 slices, 8 18 18Kb block RAM blocks, and 134 slices. Because the design has been implemented with generic multiplier and add blocks so that any device may be targeted, optimal DSP48 mapping during the synthesis step is generally not achieved.

To achieve near 400 MHz operation in a Virtex-4 device (or 450 MHz in a Virtex-5 device), several areas of the design need to be changed slightly. The single-port RAM blocks should be changed to have a latency of 2 on the output (2 output registers). The multiplier and accumulator block should be combined into a single DSP48(E) with a separate capture register on the output, using either a DSP48 primitive block or a DSP48 Macro block in the DSP folder in System Generator. The adder chain adders and registers should be combined into a chain of DSP48(E)s using the PCOUT--> PCIN routing. And, lastly, the address generator accumulator needs to be floorplanned carefully for maximum speed. For more detailed information on using the DSP48 slices and DSP48E slice, consult the *XtremeDSP™ for Virtex-4 FPGAs User Guide* (UG073) and the *Virtex-5 XtremeDSP User Guide* (UG193), respectively.

Reference Design Files

The fractional decimator design described in this application note as well as a complementary fractional interpolator design can be downloaded at: xapp936.zip

The designs use parameters for input bit width, coefficient bit width, and intermediate results so changing those values in the initialization script (coefficient values and other parameters are initialized in the InitFcn callback under Model Properties) changes them globally in the design. This allows the design to serve as a starting point and be tailored easily to meet system requirements for different applications.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/27/06	1.0	Initial Xilinx release.
07/06/06	1.0.1	Changed the link to the Reference Design Files.
07/10/06	1.0.2	Corrected part number in Implementation Results.
03/05/07	1.1	Replaced Figure 8, Figure 11, and Figure 12. Revised paragraph under Figure 12 and the “Implementation Results.” The reference design files were updated. See the readme file in xapp936.zip.