AN 623: Using the DSP Builder Advanced Blockset to Implement Resampling Filters

AN-623-1.0 Application Note

This application note discusses various design techniques for implementing resampling filters using the Altera® DSP Builder advanced blockset. The DSP Builder advanced blockset supports constraint-based high-level synthesis and is particularly efficient for implementing multiple channel, high-performance resampling filters. You can use the DSP Builder advanced blockset to quickly map highly abstract algorithms to model-based Simulink designs and to generate near optimal HDL code based on your design requirements. In addition, because you can parameterize the DSP Builder advanced blockset components, you can quickly update your design if the specification changes.

Prerequisites

This application notes assumes that you have general knowledge of DSP algorithms and tools. In particular, it assume that you have basic knowledge of the following topics:

- Decimation and interpolation filters, including polyphase decomposition
- The Mathworks MATLAB and Simulink tools
- DSP Builder

This prerequisite knowledge ensures that you understand the algorithms and architecture of various resampling filters and design examples described in this application note.

The design examples used in this application note can be found on the DSP Design Examples page of the Altera website.

The remainder of this application note disccusses the following topics:

- Resampling Filter Basics
- Resampling Filter Examples in DSP Builder Advanced Blockset

Resampling Filter Basics

Sample rate conversion has a wide range of applications including wireless communications, medical imaging, and military applications. Sample rate conversion is often computationally intensive and requires parallel processing of a large of number of independent data channels, making a high-performance resampling filter a suitable candidate for FPGA implementations. Altera's DSP Builder advanced blockset is a high-level synthesis tool which is particularly useful in creating fast implementations of sample rate conversions applications in FPGAs.

101 Innovation Drive
San Jose, CA 95134
www.altera.com

Wireless Communications

The trend in 3G and 4G wireless communications terminals is to support multimode, multi-bands and to be capable of changing between multiple standards. One of the major challenges of multimode design is that the sampling rate requirements are completely different. For example, the baseband sampling frequency of a Global System for Mobile Communications (GSM) is a multiple of 270.833 KHz, while the sampling frequencies of 3 GPP Long Term Evolution (LTE) and Wideband Code Division Multiple Access (W-CDMA) systems are multiples of 3.84 MHz. In the remote radio head design for the multi-mode systems, an efficient solution requires reuse of a portion of the hardware resources for different standards. Typical hardware reuse is in digitalup converter (DUC) and digitaldown converter (DDC) portion of remote radio head designs. A resampling filter processes data from multiple standards using the same filter chain. The ratio of LTE and WCDMA baseband sample rates to that of GSM systems is 4608/325. This rate change factor makes the conventional polyphase implementation almost impossible.

Medical Imaging

In medical imaging applications, such as ultrasound and MRI, you can apply a variable rate decimation filter to base band echo signals to adjust the sample rate. Highly oversampled original echo signals are downsampled by an integer factor, which can change at run time. Usually, as the decimation rate increases, the filter length grows proportionally. The polyphase structure is an efficient hardware solution for the filter. The polyphase structure uses a fixed number of multipliers, thus it can handle a wide range of integer rate change factors. One challenge in designing resampling filters for medical imaging is the support to process a large number of parallel channels simultaneously. While current commercial MRI scanners typically support 32 channels of data, next generation scanners may support up to 128 channels of data concurrently.

Different resampling filter requirements warrant different architectures. Innovative implementation schemes for different applications exist in numerous patents and in the research literature. This application note discusses a few basic resampling filter architectures and focuses on their implementation using the Altera DSP Builder advanced blockset.

The Altera DSP Builder advanced blockset is a high-level synthesis tool that is integrated with the Mathworks Simulink tool. The advanced blockset allows you to quickly design an algorithm, such as a high performance resampling filter, without regard for the hardware implementation details. The DSP Builder advanced blockset automatically schedules resource sharing typically seen in a multiple channel resampling filter. It also can update key parameters at run time using the Avalon® Memory Mapped (Avalon-MM) interface.

For more information about the Avalon-MM protocol, including timing diagrams, refer to the *"Avalon Memory-Mapped Interfaces"* chapter in the *Avalon Interface Specifications*.

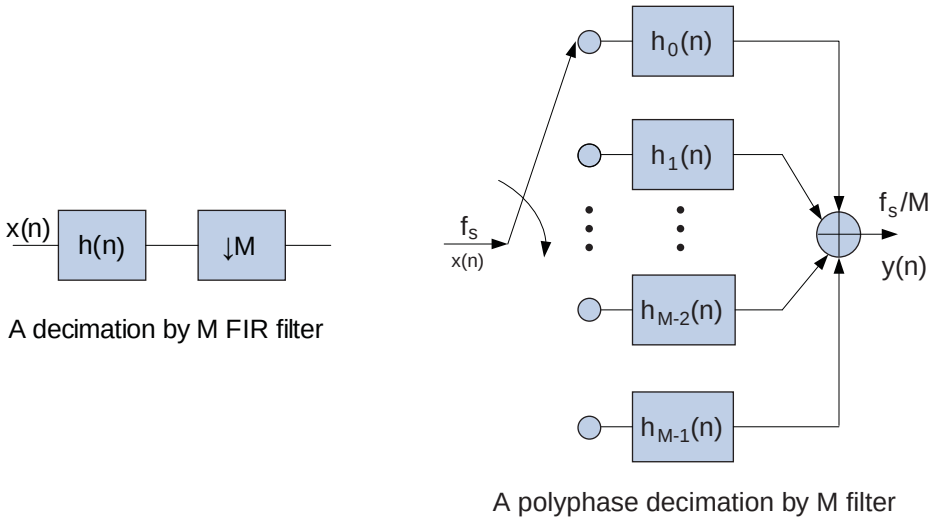
The following sections discuss a few different architectures for resampling filters and demonstrate how you can implement high performance resampling filters quickly using the DSP Builder advanced blockset.

Polyphase Decomposition

Sample rate conversion refers to decimation, interpolation, or a combination of downsampling and upsampling for fractional rate change factors. Decimation requires low-pass filtering on high rate data first, followed by periodic downsampling to remove unused data. Low-pass filtering shapes the signal spectrum to prevent aliasing. After downsampling, the output signal is at a lower sample rate. Interpolation inserts zeros into the input low rate data, then applies a low-pass filter to smooth out the signal transitions so that the inserted zeros become interpolated data.

Implementing multi-rate filters directly wastes considerable resources. Decimation throws away filtered data, as illustrated by the filter structure on the left in Figure 1.

Figure 1. A decimation by M filter and its polyphase decompositio

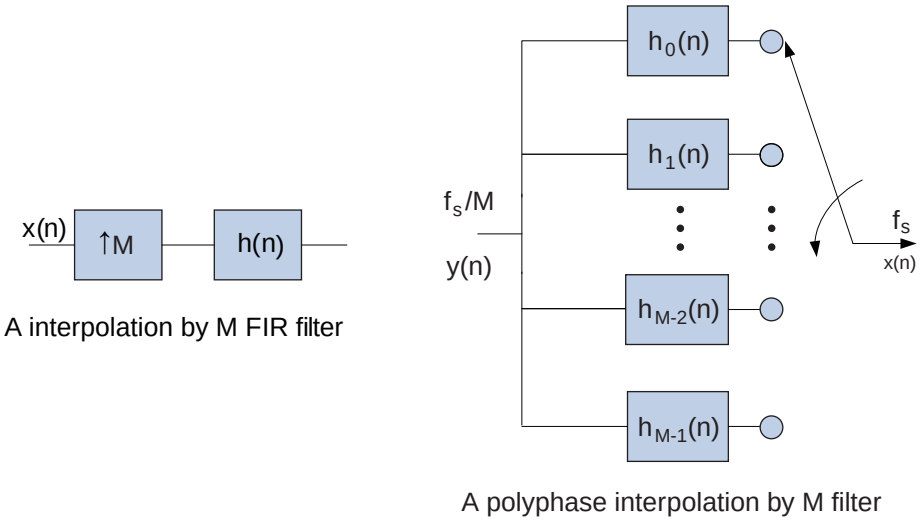


The interpolation filter has many zeros in its input signal, as illustrated by the direct interpolation filter structure on the left in Figure 2. Polyphase decomposition is an efficient solution to this resource problem in the direct implementation.

Figure 1 illustrates a polyphase decimation filter at an integer rate. The original length <*M*×*L*> filter is broken down into <*M*> polyphase components $h_0(n)$–$h_{M-1}(n)$, where each polyphase has <*L*> taps. In the polyphase implementation on the right of Figure 1, input data are clocked in at a rate of one unique sample per unit time, corresponding to a sample rate of <f_s>. The new samples are delivered to polyphase components sequentially. Therefore, new data arrive at each polyphase at a rate of <f_s/M>. An input commutator is used in Figure 1 to represent the delay then down sample operation of the input data. The direct implementation on the left requires <*M*×*L*> multiplications and <*M*×*L-1*> additions per unit time at <f_s>. In the polyphase decomposed version on the right, each filter is <*L*> taps long, and its input is at rate <f_s/M>. Therefore, each polyphase requires <*L/M*> multiplications and <*(L-1)/M*> additions per unit time. The entire filter requires <*L*> multiplications and <*(L-1)+(M-1)*> additions per unit time.

The interpolation filter can be evaluated in a similar manner. Figure 2 shows the polyphase decomposition of an integer rate interpolation filter. The input data at low rate <f_s/M> is delivered to <M> polyphases simultaneously, and a high speed switch samples the outputs of the polyphase filters sequentially at rate <f_s>. Just as in the decimation filter case, the polyphase structure is more efficient than the direct implementation because computations are done at the low sampling rate.

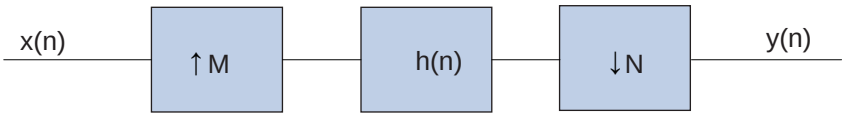
Figure 2. An interpolation by M filter and its polyphase decomposition.



Fractional Rate Resampling Filter

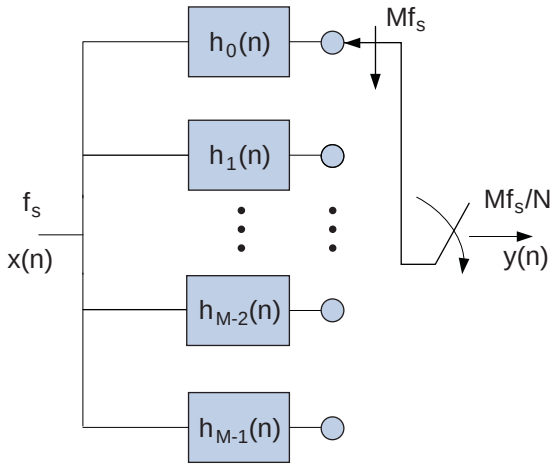
Some applications require a sample rate change of <M/N>, as shown in Figure 3. The resampling can be realized in a two-step process. First, the sample rate is raised to <Mf_s> via standard polyphase decomposition by the <M> interpolation filter. Then, <N>-to-1 downsampling reduces the sample rate to <$(M/N)f_s$>.

Figure 3. An M/N resampling filter



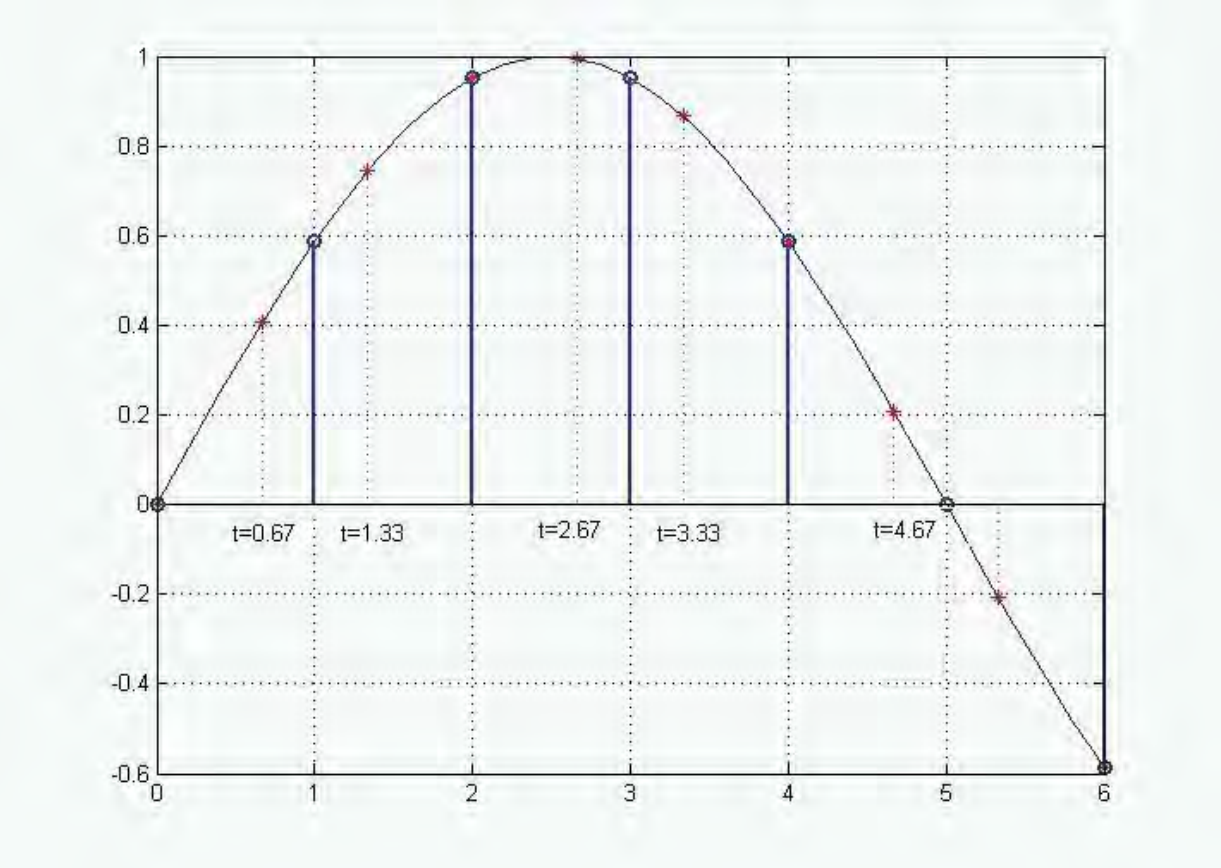
This structure can be implemented as a polyphase decomposed <×*M*>-filter followed by an output commutator with a stride of length of <*N*>, as shown in Figure 4.

Figure 4. The polyphase structure of an M/N resampling filter



Resampling to a fractional rate at <*M/N*> of the input sample rate means the output time location falls between the integer sample indexes on the time axis as Figure 5 illustrates. Figure 5 shows the input and output sample time as well as interpolated sample values of a 3/2 fractional rate sampling filter. The blue circles represent the original signal as sample rate < f_s>. The red asterisks represent the interpolated output samples. At a rate of 3/2, the output sample points can overlap with input samples, and, at times, they fall between sample points <*k*> and <*k+1*>. The distance between input sample time <*k*> and the fractional output sample time is <*k*+Δ>. For the example in Figure 5, Δ is from a finite set of values [0, 2/3, 1/3]. This delta represents the time jitter during fractional resampling.

Figure 5. Input and Output Sample Time of a 3/2 Fractional Resampling Filter



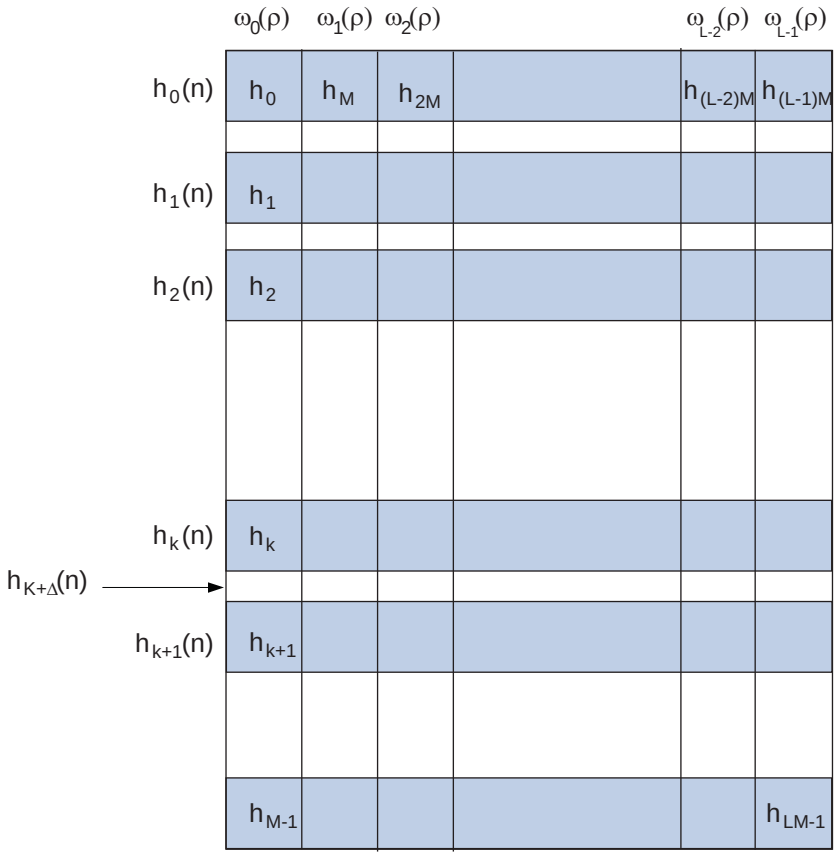
Reducing phase jitter artifacts and achieving a finer grained interpolation requires a large number of polyphases, <*M*>. However, the upsampling then downsampling structure shown in Figure 3 is not always feasible for large values of <*M*>, because the interpolation rate can well exceed the maximum FPGA clock frequency in a high performance system. For example, a multi-mode remote radio head design supporting both GSM and LTE standards requires a sample rate change of 4608/325. A direct upsampling by 4608 requires a system clock rate of 1.7 GHz, which is not feasible.

On the other hand, due to the polyphase structure, increasing <*M*> does not increase the computational complexity of the resampler, as long as the filter length is proportional to the number of polyphases. However, a large <*M*> requires more filter coefficients storage in the memory if you choose to pre-store all filter coefficients.

As an alternative, when the number of polyphases <*M*> is large, you can use polynomial approximation for the prototype filter response. First, consider the polyphase coefficients of a total length <*L*×*M*> interpolation by <*M*> filter used in the fractional rate resampler shown in Figure 4 on page 5. Using this filter as a prototype, the filter coefficients are mapped to the matrix shown in Figure 6. Each row of the coefficients matrix in Figure 6 is a polyphase component of the prototype filter. Row <*k*> computes output samples at time <*n* + *k*/*M*> (See "F.J. Harris, Multirate Signal Processing for Communication Systems, Prentice Hall, 2004." on page 27.), where <*n*>

is the integer time index 0, 1, 2, and so on. Row <*k+1*> computes output samples at time <*n + (k+1)/M*>. Conceptually, samples corresponding to a fractional time offset Δ, at <*n + (k+*Δ*)/M*>, can be computed using an imaginary row <*k+*Δ> located between row <*k*> and <*k+1*>. This is the pointer in Figure 6. The weight function corresponding to this imaginary row can be interpolated from its nearest neighbors <$h_k(n)$> and <$h_{k+1}(n)$>. This new subfilter is denoted as <$h_{k+\Delta}(n)$>, and it maps to the prototype filter by <$h_{k+\Delta}(n) = h(k + \Delta + nM)$>. When <*M*> is large, the coefficients of <$h_k(n)$> and <$h_{k+1}(n)$> can be quite similar. The goal is to interpolate these two polyphase components at time offset Δ so that the fractional rate output sample at time offset Δ can be computed. The original problem of interpolating output samples now becomes the problem of interpolating filter weights. This manipulation results in a very hardware efficient structure of a fractional rate resampling filter called a Farrow filter.

Figure 6. Coefficients mapping M-stage polyphase filter



Farrow Filter

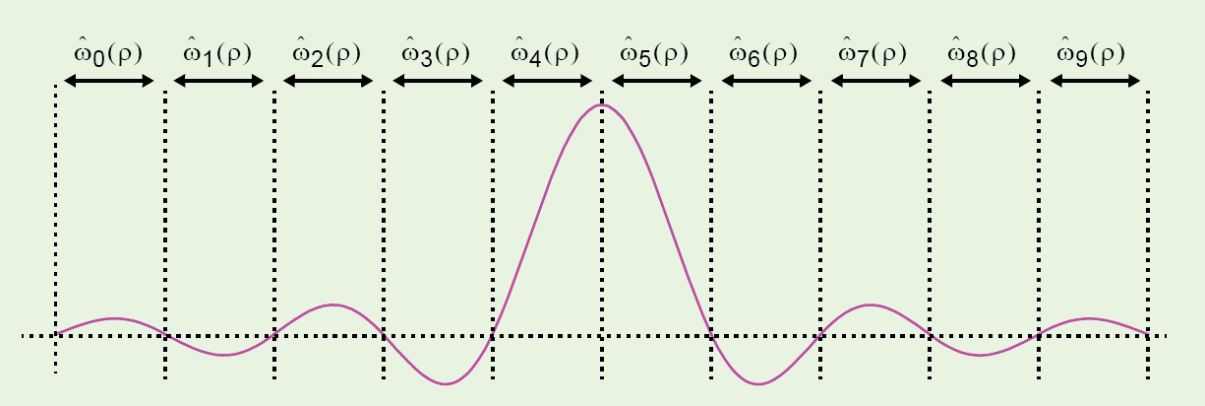
The filter weights $<h_{k+\Delta}(n)>$, come from the polyphase coefficients matrix of the prototype filter shown in Figure 6. Each column of this filter is a section of the impulse response of the prototype filter. A low order polynomial can approximate it. If the polynomial order is $<P>$, Equation 1 provides an approximation where Δ is used to quantify the sampling phase difference between the current input and desired output sample.

Equation 1. Polynomial Approximation

$$\omega_n(\Delta) = \alpha_{n,0} + \alpha_{n,1}\Delta + \alpha_{n,2}\Delta^2 + \ldots + \alpha_{n,P}\Delta^P$$

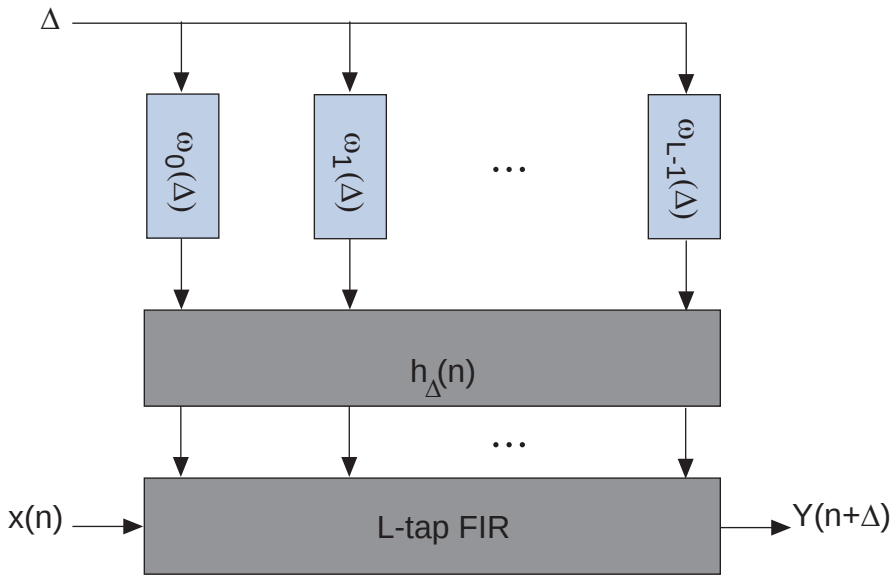
Figure 7 illustrates the polynomial approximation. Given a prototype filter response, the polynomial can be identified before implementing the actual resampling filter.

Figure 7. Polynomial Approximation by Sections of the Prototype Filter Impulse Response



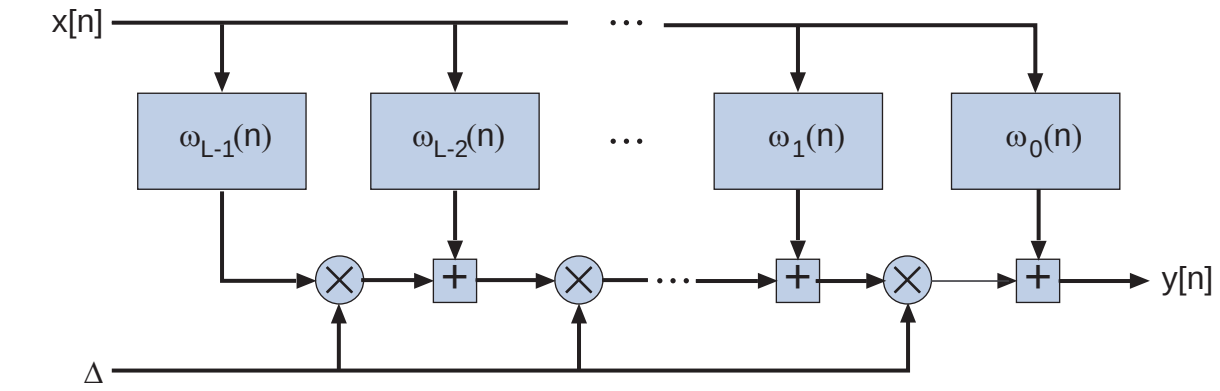
With this polynomial approximation, the arbitrary fractional rate resampling filter can be implemented in the two steps shown in Figure 8. First, determine the weight function corresponding to a time offset of input and output sample time. Then, use the estimated weight function to process input signals.

Figure 8. Polynomial Form of a Fractional Rate Resampling Filter



This modification results in significant resource saving. For instance, for a 200-tap prototype filter implementing interpolation by 50 (<*M*> = 50), 200 coefficients are needed. Approximating it with 4 polynomials of order 4, requires only 20 coefficients. In addition, the control mechanism that operates at <Mf_s> for the first step upsampling is eliminated. The datapath can operate at the output sample rate at <$(M/N)f_s$>. This filter structure can be further simplified by manipulating the Taylor series expansion of the interpolated output signal and exchanging the order of polynomial evaluation and FIR filtering. The final simplification comes from applying Horner's Rule to the polynomial evaluation, which gives the Farrow filter structure in Figure 9.

Figure 9. Farrow Filter Structure



In the Farrow structure, <*L*> low order polynomials replace the prototype filter and they filter the input data first. The outputs of these polynomials are the Taylor series expansion of the input signal. For instance, $\omega_0(n)$ evaluation the DC terms of the input data, and $\omega_1(n)$ extracts the first order derivative of the input signal. They are then weighted by the time offset Δ and combined to generate an interpolated output. The hardware efficient structure in Figure 9 requires <*L(P+1) + (L-1)*> multiplications to calculate one output sample and the design can operate at output sample rate $<(M/N)f_s>$ or a combination of $<f_s>$ and $<(M/N)f_s>$. It is particularly efficient for processing multiple channels or multiple parallel datapaths, where all channels or datapaths require the same set of filter coefficients.

Resampling Filter Examples in DSP Builder Advanced Blockset

You can implement these resampling filters using the Altera DSP Builder advanced blockset. Designing with DSP Builder advanced blockset typically includes the following steps:

1. Define the clock rate, bit width, and filter design in MATLAB.
2. Map the algorithm in the most natural and intuitive way in a Simulink model using DSP Builder advanced blockset library blocks.
3. Run a simulation, debug, and verify your design.

DSP Builder generates synthesizable HDL code when you start a simulation. You can leave most, if not all, hardware optimization considerations to the tool. Let the DSP Builder advanced blockset optimize the following features of the design:

- Pipeline stages required to meet timing
- Memory versus logic tradeoffs based on device selection
- State machines to match pipeline latencies
- Routing registers to account for routing delays

You can also allow DSP Builder to determine optimal resource sharing and time division multiplexing (TDM). Defining the algorithm using a tool that provides high-level abstractions creates a design that is portable across many devices and parameterizable when the design specification changes. In addition, it significantly shortens the design cycle and improves productivity.

This application note demonstrates the tool flow and provides information about using the DSP Builder advanced blockset in the following three example designs:

- Reconfigurable Decimation Filter
- Variable Rate Decimation Filter
- Multichannel Farrow Filter

Reconfigurable Decimation Filter

Many medical imaging systems, including ultrasound and MRI, require a reconfigurable decimation filter to reduce the echo data sample rate. The input data has a fixed sample rate; however, the integer decimation rate must be changed in real time. Furthermore, the total filter length grows linearly with the decimation rate. Wireless communications applications may have similar requirements. A polyphase structure is highly optimized for these applications because the multiplier count is fixed at compile time and does not grow with the rate increase. The key optimization is in the variable length delay taps and efficient filter coefficients storage.

Features

This design example has the following key features:

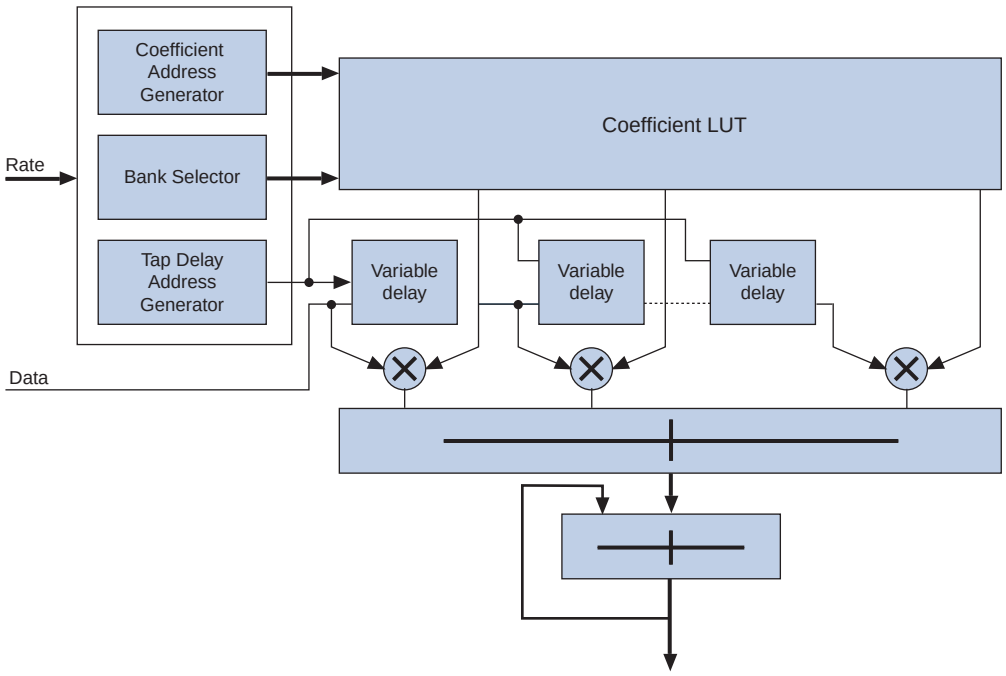
- Supports an arbitrary integer decimation rate (including the cases without rate change), an arbitrary number of channels, and an arbitrary clock rate and input sample rate, as long as the clock rate is high enough to process all channels in a single datapath
- Supports run-time reconfiguration of decimation rate
- Uses two memory banks for filter coefficients storage instead of prestoring coefficients for all rates in memory. Updates one memory bank while the design is reading coefficients from the other bank
- Provides real-time control of scaling in the FIR datapath

You can download the design files for this example from the Reconfigurable Decimation Filter Design Example Using DSP Builder Advanced Blockset page of the Altera website.

Functional Description

The design uses a direct form polyphase decimation filter structure as illustrated in Figure 10. The address controller generates the read address of the coefficients stored in memory, a bank selector, and the write addresses of the variable delay taps. The coefficients are stored in on-chip memory RAM blocks. The variable delay taps are also implemented in dual-port memories. The current decimation rate controls the delay tap pointer. The design uses a fixed number of multipliers.

Figure 10. Block Diagram of the Variable Rate Decimator



The setup script for this design defines the clock rate, decimation rate, filter length, and multiplier engine (polyphase components FIR length). Scripts also provide the parameters for bit width management. Table 1 defines key parameters.

Parameters

You can modify all parameters to target a different design. New HDL codes is generated based on the updated parameters.

Table 1. Parameters for variable rate decimation filter example (Part 1 of 2)

Parameter	Definition
`ClockRate`	FPGA Clock rate. It should be the target f_{MAX}.
`SampleRate`	Input data sample rate.
`Period`	Number of cycles available between unique input samples. Equals `ClockRate/SampleRate`. It is a compile time parameter.
`ChanCount`	Number of input channels. It should not exceed `Period`.

Table 1. Parameters for variable rate decimation filter example (Part 2 of 2)

Parameter	Definition
Rmax	Maximum decimation rate the design supports.
R	Current decimation rate. It should not exceed Rmax.
L	FIR kernel size, that is, the multiplier engine size. It is the polyphase FIR filter length.

Variable Tap Delay Lines

The variable tap delay blocks in this example have a run time reconfigurable depth. They are implemented as elastic memories using on-chip RAM blocks. Each delay tap is allocated based on the worst case, and it is Rmax × Period words deep. The actual number of delays through a delay tap block is R × Period, which is based on current decimation rate. Period accounts for the multiple channel support and the case when FPGA is running faster than the input sample rate.

A single pointer or address signal reads and writes into the delay tap using a two-port RAM. This RAM is configured to read out old data and write new data to the same location in a single cycle, realizing a delay of R × Period cycles.

The elastic memory pointer cycles through zero to R × Period. If the rate changes, it immediately reverts back to zero, so that a new rate initiaties writes to the beginning of a delay tap block. Each delay tap block includes a clear signal. The clear signal resets the delay tap output to all zeros for R × Period cycles immediately after a rate change, clearing the shift register chain at each rate transition. Note that the length of the clear signal assertion is based on the new decimation rate, instead of the old rate. If you do not need to clear the delay taps chain, you can bypass the reset stage which may improve your f_{MAX} slightly. It takes R × Period cycles to completely flush the contents of a delay tap, and it takes L × R × Period cycles, or the total length of the delay tap chain where L is the number of delay tap blocks, to flush out the entire delay tap chain.

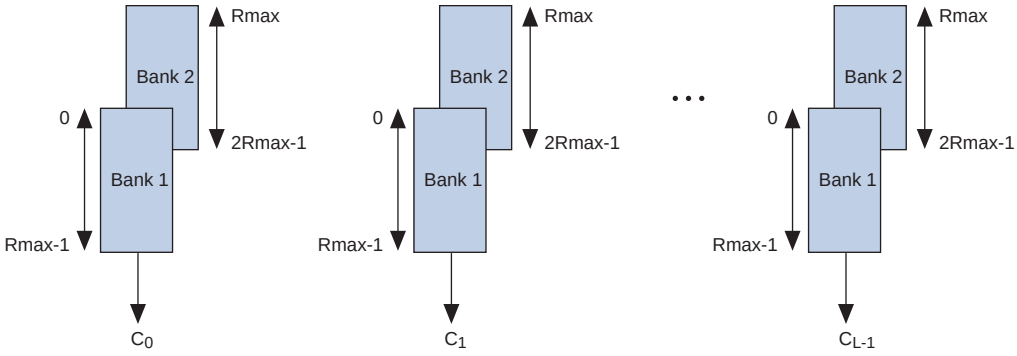
Dual Coefficient Bank

There are many ways to store FIR filter coefficients. If your design is not memory-limited, or you do not want to have a processor to update coefficients, you can prestore entire coefficient sets corresponding to all possible rates in memory. Each memory bank stores a coefficient set. Supporting a large number of rates can be very memory consuming. If you allocate coefficient banks based on the maximum rate supported, you can use a simple addressing scheme because all banks are the same size. However, this allocation scheme wastes memory because the filter length varies for different rates. If you allocate banks based on individual rates, the address scheme may be rather complex, especially if you support a large of number of rates.

This example uses two memory banks, with one updating while the other is accessed. Coefficients are reloaded at run time using the processor interface. Figure 11 illustrates the structure of coefficient storage in memory. Each memory bank is allocated for the maximum rate supported, although its valid contents is only R-deep, where R is the current rate. L such blocks are required where L is the multiplier engine size. The total memory usage for coefficients is 2(Rmax × L). A simple address counter and a bank selector are sufficient to index both banks. Note that for multiple-channel cases, the coefficient address counter updates every Period cycles so that all channels use the same coefficients.

The bank signal is an output of the top level design, so that a controller can monitor it and decide which bank it can reload with new coefficients.

Figure 11. Dual Coefficient Bank Setup for the Variable Rate Decimation Filter



Real Time Reconfiguration of Filter Coefficients and Control Registers

You can use the Nios® II processor or a different processor to reload the coefficients and change the decimation rate. The processor interface uses the Avalon-MM interface to update the following parameters and variables:

- The current decimation rate
- The coefficients bank
- Scaling for the FIR filter multi-port adder
- Scaling for the FIR decimation filter final accumulator

Using control registers and the Avalon-MM blocks in DSP Builder advanced blockset is easy. You can drag and drop registers or memories into your design. Then, you specify the relative base address of control registers and memory blocks. DSP Builder advanced blockset automatically generates address decoding logic and the appropriate system interconnect fabric.

Signals

Table 2 lists the top-level interface signals of the synthesizable design.

Table 2. Interface signals of the variable rate decimation filter

Signal	Direction	Description
av	Input	Input valid signal
ac	Input	Input channel signal
a	Input	Input data signal
v	Output	Output valid signal
c	Output	Output channel signal
data_out	Output	Output data signal
rate_out	Output	Current decimation rate corresponding to the output data
bank_out	Output	Current coefficient bank used by the output data

Example Walkthrough

The following example presents two instances of the reconfigurable decimation filter. It demonstrates how you can easily migrate similar designs to meet different specifications. In most cases, you only need to modify the setup script text file to reflect specification changes. Even if you need to make changes to model design file, the modification often is minimal.

Example 1

The first example has the following parameters:

- Total number of channels—16
- Input sample rate of each channel—16 MHz
- Decimation rate—1 (no rate change) to 16.
- Multiplier engine size—10.

This specification results in the following settings:

- FPGA clock rate—256MHz, allowing all channels to be processed on a single datapath.
- Total filter length range—10 (single rate) to 160 (decimation by 16). MATLAB **fir1.m** is used for filter design.

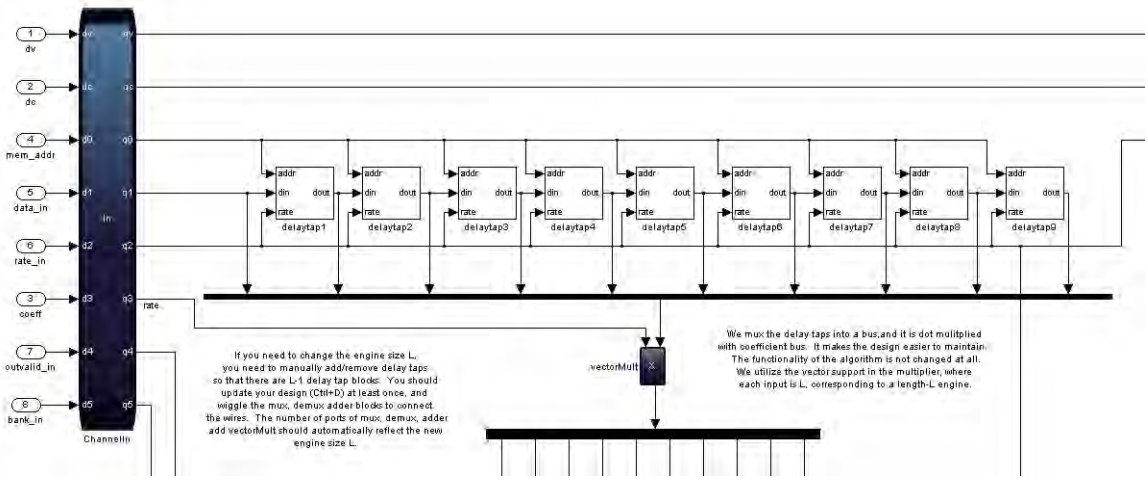
Example 1 shows how to initialize these parameters in an **.m** file before simulating.

Example 1. Script to Initialize Parameters in Example 1

```
% File: setup_vardownsampler.m
% Description: Script to set variables in Matlab workspace to configure vardownsample
model
%               This design assumes fixed input sample rate and variable
%               decimation rate.  It has a polyphase structure, and the
%               kernel size (multiplier count) is fixed.  The total filter
%               length grows linearly with the decimation rate.

%% Multichannel setup
ChanCount=16;

%% clock and sample rate setup
ClockRate=256;
% NOTE: input sample rate must be fixed at compile time and should divide
% ClockRate
SampleRate= 16;
Period=floor(ClockRate / SampleRate);
SampleTime = 1;
%SampleTime = 1 / (ClockRate * 1e6);   % uncomment this line to simulate
%the model with realworld time
ClockMargin = 0.0;

%% Decimation rate setup
Rmax = 16; % maximum sample rate the design supports
R = [1 2 5 8 2]; %sample rates being tested in the test bench
numRates = length(R);

%% Filter setup
% Option 1: fixed kernel size;
%           derive total filter length from kernal size
% Spec: total filter length = totlen;
L = 10; % muliplier engine (kernal size): number of multipliers;
fLen = L*R; % Filter length for each sample rate; linearly grows with Rate
fLenmax = L*Rmax; % worst case filter length; ie maximum filter length
```

Elastic memories, multipliers, a multi-port adder, and an accumulator implement the direct form FIR as Figure 12 illustrates. A data bus aggregates the variable length delay taps to take advantage of the vector support of the DSP Builder advanced blockset. The output of the data bus is an <*L*>-element vector. It is multiplied with the <*L*>--element coefficient vector using element-by-element multiplication which is equivalent to a dot product in MATLAB. Therefore, Figure 12 only shows one multiplier with vector support, making the design much more portable, clean, and easier to manage.

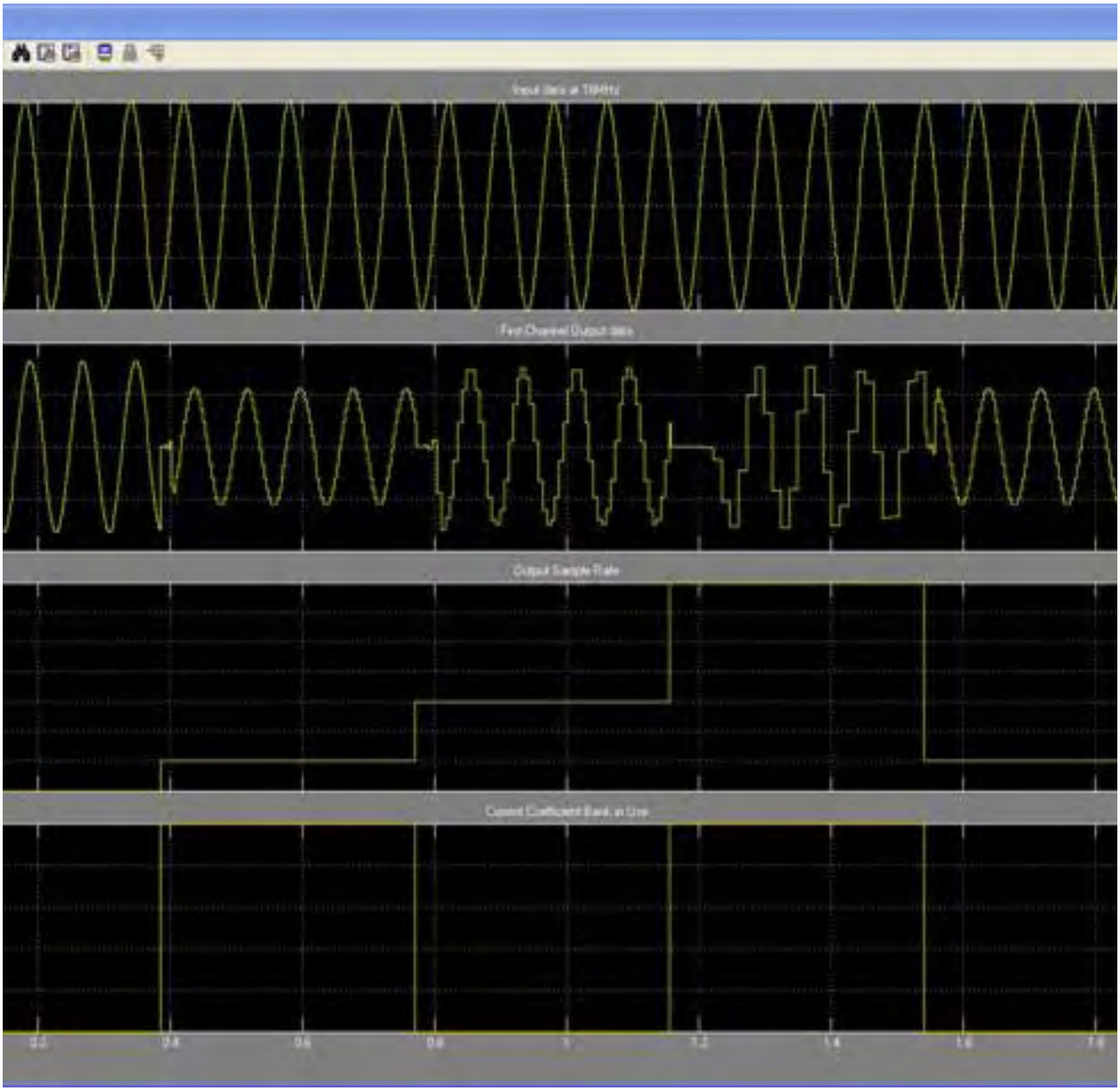
Figure 12. DSP Builder Advanced Block Set Implementation of the Variable Integer Rate Decimation Filter—Example 1



The input signal dimension of the data multiplexer/demultiplexer, multiplier, and adder are all parameterized with <*L*>, the kernel size.

Figure 13 plots the downsampled output when input is a sine wave. The delay tap chain is cleared at each rate transition.

Figure 13. Sample Variable integer Rate Decimation Filter Output for Example 1 ***(Note 1)***



Note to Figure 13:

(1) If you multiplier kernel size does not change, you can change the clock rate, sample rate, and channel count parameters in the setup script. You do not need to make any changes to the model. New HDL code is generated at the start of the simulation based on your updated parameters.

Example 2

The second example has the following parameters:

- Input—single channel.
- Input sample rate—20 MHz.

- Decimation rate—4 to 20.
- Total filter length—32 to 160.

The specification results in the following settings:

- FPGA clock —160MHz. This selection is arbitrary. You can choose whatever rate that is best for your system.
- FIR multiplier engine size—8. This size is derived from the total filter length and decimation rate. It is the same across all rates.

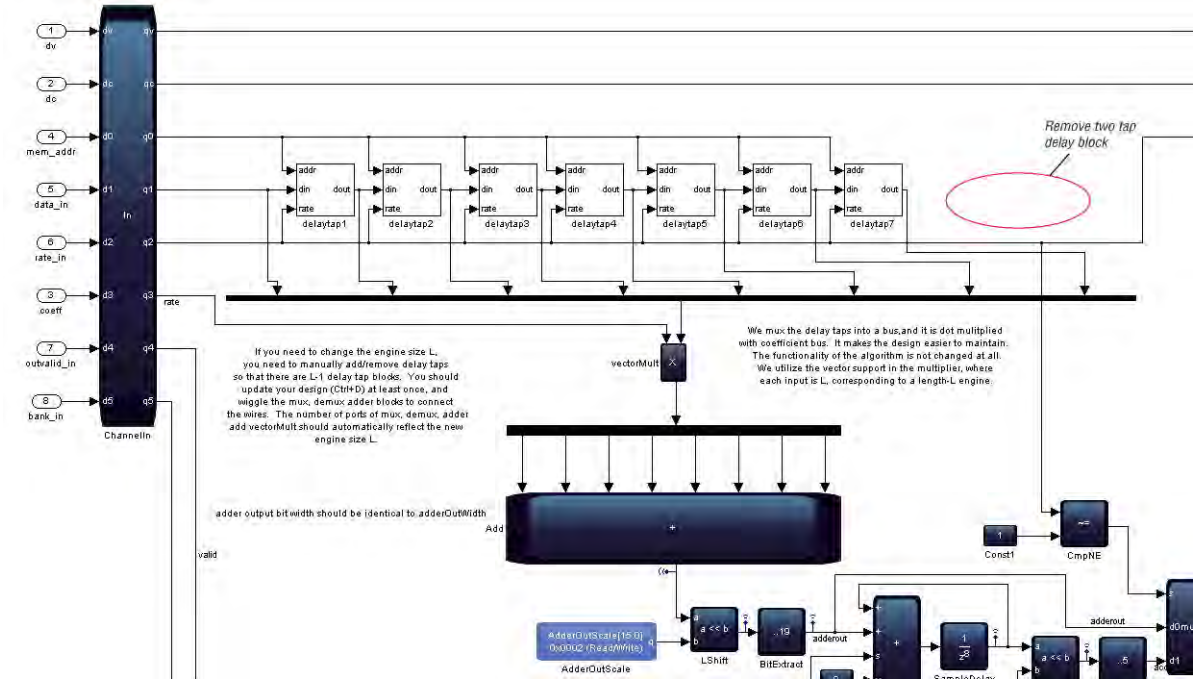
Even though Example 2 is quite different than Example 1, the most efficient approach to creating this example is to copy and modify the Example 1 design. The new specification is reflected in the updated setup script shown in Example 2.

Example 2. Script to Initialize Parameters in Example 2

```
% File: setup_vardownsampler.m
% Description: Script to set variables in Matlab workspace to configure vardownsample
model
%               This design assumes fixed input sample rate and variable
%               decimation rate.  It has a polyphase structure, and the
%               kernal size (multiplier count) is fixed.  The total filter
%               length grows linearly with the decimation rate.

%% Multichannel setup
ChanCount=1;

%% clock and sample rate setup
ClockRate=160;
% NOTE: input sample rate must be fixed at compile time and should divide
% ClockRate
SampleRate= 20;
Period=floor(ClockRate / SampleRate);
SampleTime = 1;
%SampleTime = 1 / (ClockRate * 1e6);   % uncomment this line to simulate
%the model with realworld time
ClockMargin = 0.0;

%% Decimation rate setup
Rmax = 20; % maximum sample rate the design supports
R = [4 5 7 8 9]; %sample rates being tested in the test bench
numRates = length(R);

%% Filter Setup
% Option 2: fixed total filter length;
%          derive kernal size from rate and total filter length
fLen = [32 40 56 64 72];
L = fLen(1)/R(1); % muliplier engine (kernal size): number of multipliers;
```

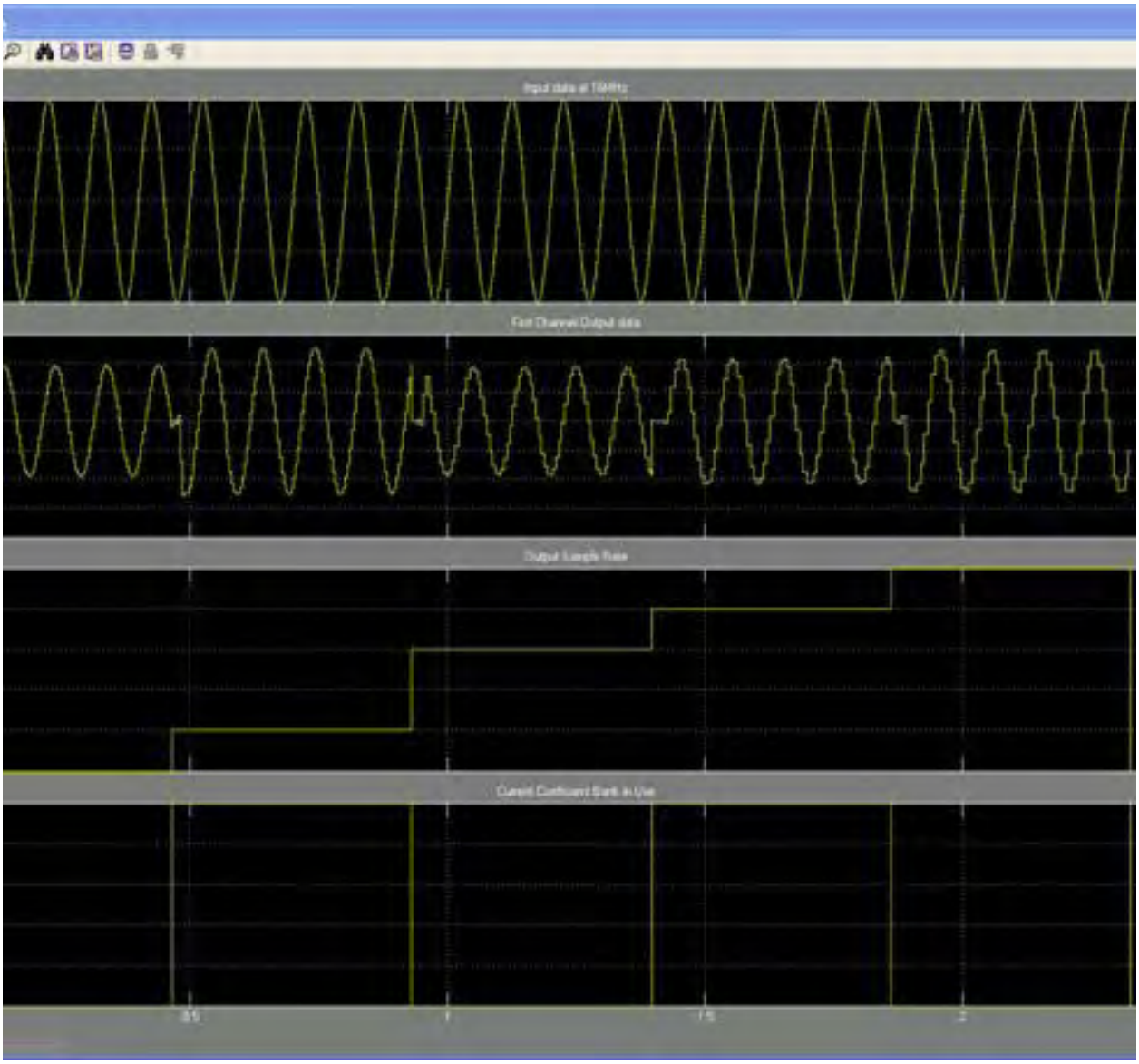
The only change you must make to the model is to remove the two extra tap delay blocks in the FIR filter, as shown in Figure 13. Because all blocks are parameterized, you do not need to make any other manual changes. New HDL code is generated when you run simulation.

Figure 14. DSP Builder Advanced Block Set Implementation of the Variable Integer Rate Decimation Filter—Example 2



As in Example 1, you can explore many combinations of clock rate, sample rate, channel count and decimation rate with this design by modifying the setup script.

Figure 15. Sample Variable integer Rate Decimation Filter Output for Example 2



Variable Rate Decimation Filter

You can modify the polyphase structure of the direct form FIR with an accumulator to support both the integer and fractional rate decimation filter.

You can download the design files for this example from the Variable Integer Rate Decimation Filter Design Example page od the Altera website.

Functional Description

Conventional decimation by <*N*> filters can be efficiently implemented via polyphase decomposition with an input commutator and <*N*> parallel paths. Each path is a polyphase of the original prototype filter, as illustrated in Figure 1 on page 3. If the input commutator skips every other phase, instead of going through all <*M*> paths, the decimation rate becomes <*N/2*>, and so on.

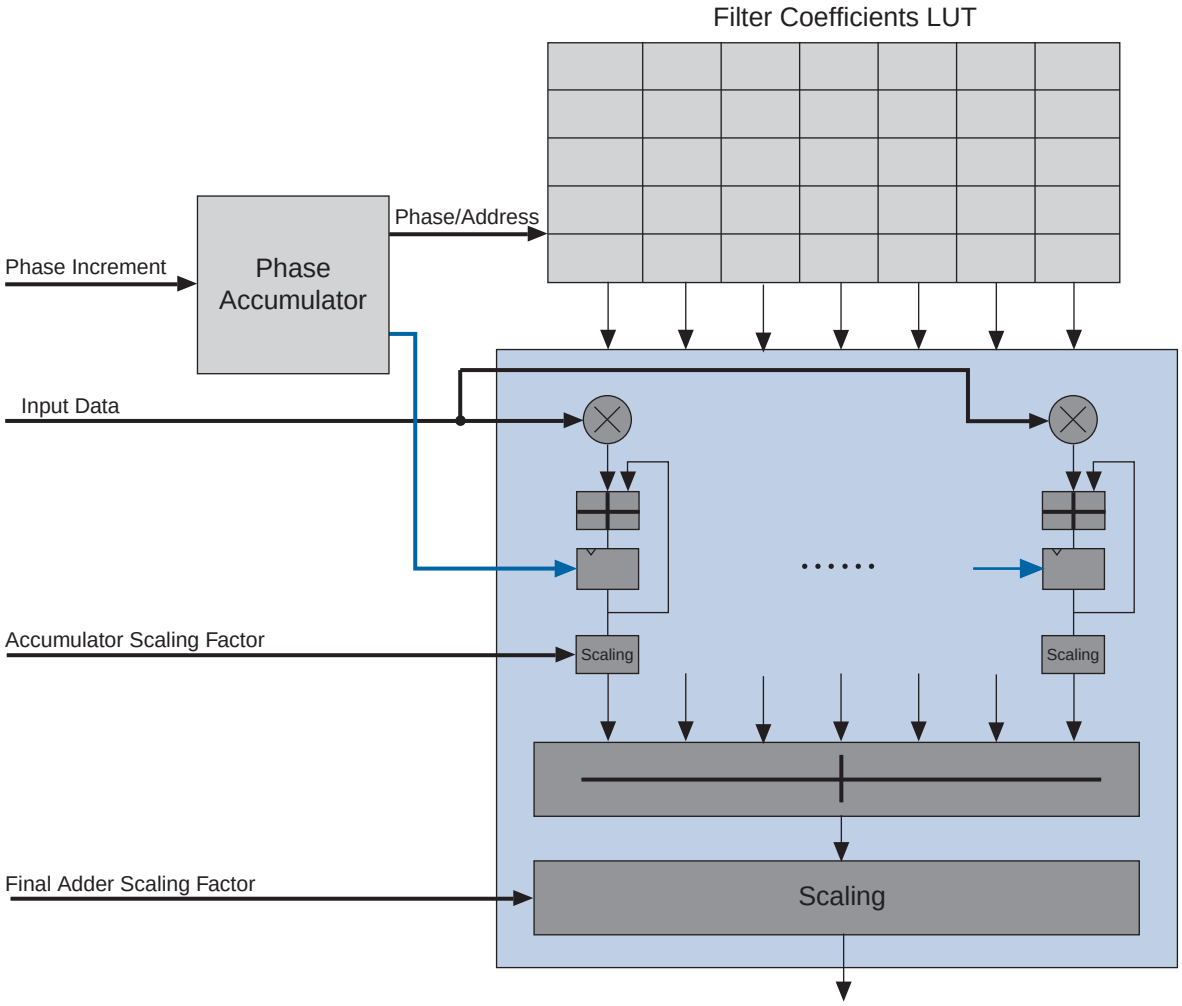
The actual design includes a single polyphase FIR, where the coefficients corresponding to the polyphases change every cycle at the input sample rate instead of a commutator and a parallel bank of FIR paths. This implementation delivers consecutive input samples to a parallel bank of polyphases. The polyphase coefficients are stored in memory and use an input accumulator like the one used in numerically controlled oscillator (NCO) to control which phase is currently being read and sent to the FIR path. The step size of the phase accumulator controls how fast the system cycles through the polyphases, consequently the decimation rate. The overflow signal of the accumulator is asserted when a valid output sample has been generated at the lower sample rate.

This design is shown in Figure 16. It implements a <*M/N*> decimation filter, where <*M*> is the number of polyphases skipped when switching polyphase. <*M*> must not exceed <*N*>. If <*M*> is divisble by <*N*>, the decimation filter rate is an integer. If <*M*> and <*N*> are coprime, the decimation filter rate is fractional.

The FIR path in Figure 16 is a modified direct form FIR filter. The filter coefficients are from the coefficients look-up table (LUT) and change from one polyphase to another. Each multiplier output has an accumulator which accumulates the outputs of all polyphases at its tap until the rollover signal indicates that all phases have been visited and an output is due. At that moment, a multiport adder sums all accumulator outputs and generates a final decimated sample. At the same time, the accumulators clear the contents for the next accumulation cycle.

Note that in this architecture a single filter is designed and stored in the memory. It is designed according to the highest decimation rate, <*N*>.

Figure 16. Block Diagram of the Arbitrary Rate Decimation Filter



You can reconfigure the decimation rate change by varying the accumulator phase increment or step size at run time. You can update the phase increment in real time via the Avalon-MM interface. When the decimation rate changes, the number of polyphases accumulated in the FIR path also changes. To maximize the dynamic range, you can supply a reconfigurable scaling factor to the multiply-and-accumulate units and to the final adder output. This functionality is also reconfigured via the Avalon-MM interface. The parameters and signals of this arbitrary rate decimator are similar to the integer rate decimator described in the previous section.

Multichannel Farrow Filter

This section shows how to use the DSP Builder advanced blockset to implement a multilchannel sample rate conversion filter based on a Farrow structure.

Features

The Farrow filter design example has the following key features:

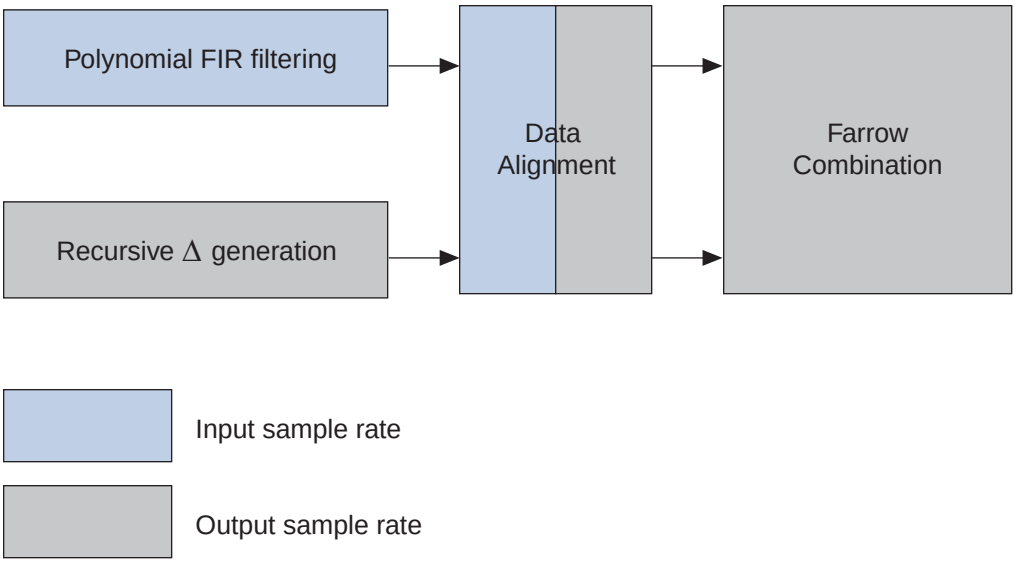
- Supports both decimation and interpolation
- Supports almost any rational sample rate change factor
- Supports up to 16 channels although you can modify the design to support more channels
- Supports automatic folding, allowing time-division multiplexing (TDM) of multipliers, adders, and other hardware resources

You can download the design files for this example from the Multichannel Farrow Filter Design Example page of the Altera website.

Functional Description

Figure 17 provides a functional block diagram of a Farrow filter. All modules are use primitive blocks from the DSP Builder advanced blockset.

Figure 17. Block Diagram of a Farrow Filter ***(Note 1)***, ***(Note 2)***



Notes to Figure 17:

(1) The current version has the FPGA clock rate as an integer multiple of both the input sample rate and output sample rate. There is no structural change for the different sample rates it supports.

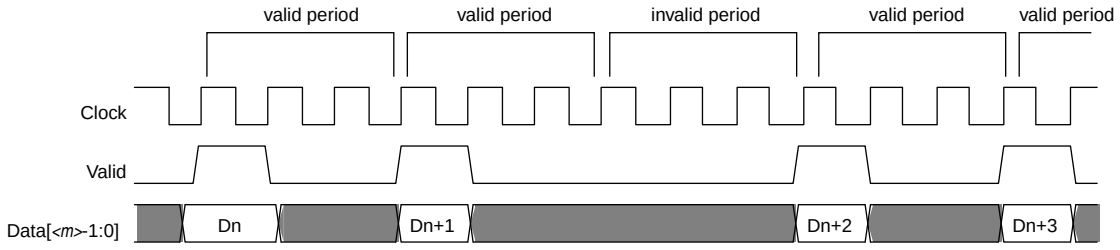
(2) All channels should be processed by one datapath, that is a single wire in the DSP Builder advanced blockset design. If you have a large number of channels, you can increase your FPGA clock rate so that you do not have to split the data channels into multiple parallel datapaths. To support multiple wires or multiple data paths, you can modify the data alignment block, which is not covered in this example.

Sample Rate Management

The DSP Builder advanced blockset supports a single clock domain. As a result, the assertion and deassertion of the `valid` signal identifies different sample rates. The Farrow filter polynomial FIRs operate at the input sample rate, while the time offset generation is at output sample rate. The `period` parameter, or number of cycles between two unique data samples, is different for the polynomial FIRs and time offset Δ generation. Therefore, you cannot directly combine the two datapaths in the Farrow structure. In the case of multiple channels, the latency introduced in the polynomial FIRs means that the `valid` signals of the top and bottom datapaths in Figure 17 may be misaligned. Before they are combined, you must synchronize the channel alignment. In this example, the data alignment module provides channel alignment and `valid` signal provides synchronization.

Alternatively, you can deassert the `valid` signal more frequently if the average sample rate and system clock rate are not integer multiples. For instance, at 180 MHz clock rate, you can represent a 40 MHz signal by asserting the `valid` signal for 2 valid periods, followed by 1 period of `valid` deassertion. Each valid period spans 3 cycles as Figure 18 illustrates. DSP algorithms typically use the `valid` signal to qualify data; however, for Farrow algorithms the synchronization with time offset generation is more complex and so that the difference between sample rates cannot be resolved by using the `valid` signal to qualify data. Instead, a channel alignment module is always required.

Figure 18. Use of Valid Signal in Single Clock Domain Applications



Time Offset Generation

The symbol Δ quantifies the sampling phase difference between the current input and output sample. The value is normalized between 0 and 1. Equation 3 shows how to calculate Δ.

Equation 2. Delta Calculation

$$\Delta = (\text{Output\_time} - \text{Input\_time}) \times \text{Input\_Sampling\_Frequency}$$

The time offset, or phase difference, can be generated by using an NCO-based closed loop. Recognizing the recursive nature of the time offset tracking, this example uses a simple recursive time offset update without NCO or filtering. Equation 3 shows the calculation.

Equation 3. Time Offset Update Calculation ***(Note 1), (2)***

$$\Delta_{n+1} = \Delta_n + \begin{cases} C_1 & \text{if } \Delta_n < C_2 \\ -C_2 & \text{if } \Delta_n \geq C_2 \end{cases}$$

Notes to Equation 3:

(1) *<n>* stands for the number of samples or discrete time stamp.

(2) *<C_1>* is defined as the fractional part of the inverse of rate change factor as shown in Equation 4.

Equation 4 shows the fractional part of the inverse rate change factor.

Equation 4. Fractional Part of Inverse Rate Change Factor ***(Note 1), (2), (3)***

$$C_1 = \frac{1}{R} - \text{floor}\left(\frac{1}{R}\right)$$

and

$$C_2 = 1 - C_1$$

Notes to Equation 4:

(1) *<n>* stands for the number of samples or discrete time stamp.

(2) *<C_1>* is defined as the fractional part of the inverse of rate change factor.

(3) *<R>* stands for the sampling rate and can be both interpolation and decimation.

Utilizing Folding Feature of DSP Builder advanced blockset

Folding is closely related to time division multiplexing. When the system clock rate is faster than the sample rate, you can typically reuse one hardware block, such as a multiplier, to process multiple data points. Different data points access the shared hardware resource via TDM. Similarly, in a system with multiple data channels, instead of duplicating hardware for each channel, you can frequently use one datapath to process multiple data channels. Folding allows multiple channels to access system resources such as multipliers and adders in a TDM fashion, saving resources.

The DSP Builder advanced blockset ModelIP blocks automatically support folding. For primitive subsystems such as this example, you can enable folding by clicking the **Folding enabled** option for the **ChannelIn** and **ChannelOut** blocks of a subsystem. When you enable folding, you can edit the **Number of used TDM slots** and **Sample rate** parameters. For the polynomial FIRs, the number of TDM slots used is the number of input data channels. The sample rate is the input data sample rate. For the Farrow structure, the number of TDM slots used is also the channel count. The sample rate refers to the output sample rate because it generates the final output data. Turning on folding allows the different data channels to share multipliers.

You do not need to enable folding for time offset generation and synchronization subsystems because the same offset applies to all channels.

Related Documents

- A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, 1999.
- F.J. Harris, *Multirate Signal Processing for Communication Systems*, Prentice Hall, 2004.
- Stephen et al, *Methods and apparatus for variable-rate down-sampling filters for discrete-time sampled systems using a fixed sampling rate*, US Patent 6014682, 2000.
- F.M. Gardner, "Interpolation in digital modems- Part I: Fundamentals," *IEEE Trans. Commun.*,vol. 41, pp. 502-508, Mar. 1993.
- L. Erup, R.M. Gardner, and R.A. Harris, "Interpolation in digital modems - Part 11: Implementation and performance," *IEEE Trans. Comm.* vol. 41, pp. 998-1008, 1993.

Document Revision History

Table 3 shows the revision history for this document.

Table 3. Document Revision History

Date	Version	Changes
August 2010	1.0	Initial release.